云原生

Kubernetes

实战手册

（Rocky Linux 9 + Kubernetes）



作者：王良明

https://www.veryopen.org

目录

前言 **1**

------- 上 篇 ------- **5**

第 1 章 系统设计 **6**

第 2 章 准备四个节点 **10**

§2.1 安装 Rocky 10

§2.2 远程登录 12

§2.3 基本配置 17

§2.4 准备工作节点 22

§2.5 创建快照 25

§2.6 拓展学习 25

第 3 章 容器实战 **26**

§3.1 docker 容器 30

§3.1.1 安装 30

§3.1.2 体验一遍 33

§3.1.3 定制镜像 37

§3.1.4 构建镜像 40

§3.1.5 使用外部存储 42

§3.1.6 容器组合 43

§3.1.7 容器网络 53

§3.1.8 实战：容器里的 DNS 57

§3.1.9 部署大语言模型 60

§3.1.10 可视化管理工具 63

§3.2 podman+quadlet 容器 66

§3.2.1 安装 67

§3.2.2 实战 69

§3.2.3 可视化管理工具 74

§3.3 containerd+nerdctl 容器 75

§3.3.1 安装环境 75

§3.3.2 containerd 名字空间 76

§3.3.3 跑应用 76

§3.4 拓展学习 78

第 4 章 防火墙 **79**

§4.1 基础知识 79

§4.2 内核模块 netfilter 80

§4.3 配置规则 82

§4.4 场景实验 88

第 5 章 SELinux 安全模块 **92**

§5.1 SELinux 策略 92

§5.2 布尔参数 95

§5.3 查看 SELinux 日志 96

第 6 章 程序运行时 **97**

§6.1 动态编译 97

§6.2 静态编译 98

§6.3 chroot 实战 100

第 7 章 控制组 cgroup **106**

https://www.veryopen.org

§7.1 起底 cgroup v2 107
§7.1.1 简介 107
§7.1.2 资源控制器 111
§7.2 cgroup2 实战 115
§7.2.1 cpu 配额 116
§7.2.2 内存配额 117
§7.2.3 磁盘 IO 配额 119

第 8 章 名字空间 122
§8.1 unshare/nsenter 命令 123
§8.1.1 unshare 命令用法 123
§8.1.2 nsenter 命令用法 125
§8.2 网络名字空间 126
§8.2.1 tap/tun 127
§8.2.2 veth 131
§8.2.3 bridge 133
§8.2.4 一个简单的 VPN 136
§8.3 用户名字空间 139
§8.4 文件系统名字空间 140
§8.5 进程名字空间 142
§8.6 主机名字空间 144
§8.7 时钟名字空间 145
§8.8 IPC 名字空间 146
§8.9 挂载点名字空间 147
§8.10 手工创建准容器 147
§8.10.1 准容器实战 147
§8.10.2 解剖容器 153

第 9 章 Go 语言 159
§9.1 Go 语言简介 159
§9.2 编程环境 160
§9.3 go 语法 162
§9.3.1 定义变量 162
§9.3.2 控制语句 163
§9.3.3 函数 164
§9.3.4 指针 165
§9.4 综合例子 168
§9.4.1 多态结构 168
§9.4.2 制作容器 169

第 10 章 数字证书 171
§10.1 公/私钥 172
§10.2 数字签名 179
§10.3 数字证书 182
§10.4 openSSL/GPG 190
§10.4.1 openssl 工具 191
§10.4.2 gpg2 命令 202

第 11 章 etcd 数据库 204
§11.1 安装 204
§11.2 增加证书 206
§11.3 创建 etcd 集群 210
§11.3.1 启用节点证书 211

https://www.veryopen.org

§11.3.2 加入 worker1 节点 213
§11.4 使用 etcd 218
§11.4.1 权限管理 218
§11.4.2 内容操作 220
§11.4.3 备份和恢复 223

第 12 章 Ceph 存储 226
§12.1 系统设计 226
§12.1.1 ceph 简介 226
§12.1.2 实验架构 229
§12.2 安装 Ceph 231
§12.2.1 初始化最小集群 231
§12.2.2 添加主机 235
§12.2.3 添加 OSD 237
§12.3 使用 ceph 存储 240
§12.3.1 使用 Ceph 块设备 240
§12.3.2 使用 Ceph 文件系统 243
§12.3.3 使用对象存储 244
§12.4 管理 Ceph 248
§12.4.1 管理命令 248
§12.4.2 管理页面 248
§12.4.3 监控 Ceph 248

------- 下篇 ------- 253

第 13 章 Kubernetes 简介 254
§13.1 体系结构 254
§13.2 K8s 中的组件 257
§13.2.1 API Server 257
§13.2.2 Controller Manager 258
§13.2.3 Scheduler 259
§13.2.4 Kubelet 259
§13.2.5 Kube Proxy 260
§13.2.6 Cloud Controller Manager 260

第 14 章 安装基础服务 261
§14.1 设置防火墙规则 261
§14.2 时间同步服务 262
§14.3 安装容器运行时 264
§14.4 总结 268

第 15 章 部署 k8s 集群 270
§15.1 安装 k8s 程序 271
§15.2 初始化控制面 272
§15.3 了解集群 278
§15.4 工作节点入群 281
§15.5 安装网络插件 283
§15.6 操纵 etcd 292
§15.7 集群中的证书 294
§15.8 检查集群状态 298

第 16 章 K8s 管理基础 301
§16.1 k8s 中的网络 301
§16.1.1 节点网 301

https://www.veryopen.org

§16.1.2 pod 网络 301
§16.1.3 网络插件 CNI 302
§16.1.4 南北网络 302
§16.2 k8s 中的资源对象 303
§16.2.1 资源对象介绍 303
§16.2.2 定义资源对象 306
§16.3 RBAC 机制 324
§16.3.1 基础 324
§16.3.2 RBAC 实战 329
§16.4 k8s 中的存储 333
§16.4.1 基础 333
§16.4.2 NFS 实战 336
§16.5 管理 k8s 集群 343
§16.5.1 kubectl/ctr 命令 343
§16.5.2 helm 命令 348
§16.5.3 K8s Dashboard 354
§16.5.4 K8S 集群运维 359

第 17 章 部署应用 369
§17.1 busybox 369
§17.2 web 应用 373
§17.2.1 部署 web 服务 373
§17.2.2 访问网页 374
§17.2.3 向外暴露应用 375
§17.3 数据库 383
§17.4 wordpress 387
§17.4.1 部署数据库 MariaDB 387
§17.4.2 部署 WEB 服务 389
§17.5 网络连通性 391
§17.6 验收 395
§17.7 部署 Gemma4 396
§17.7.1 环境检查 396
§17.7.2 部署 vLLM 推理服务 397
§17.7.3 部署 Open WebUI 398
§17.7.4 清理 399
§17.8 Wasm 应用 399
§17.8.1 环境准备 399
§17.8.2 编写 Hello World 服务 400
§17.8.3 本地编译与验证 400

第 18 章 集群伸/缩 404
§18.1 伸/缩工作节点 404
§18.2 伸/缩控制面节点 404
§18.2.1 增加负载均衡器 404
§18.2.2 增加控制面节点 404

第 19 章 Service Meshes 405
§19.1 Istio 405
§19.2 Gateway API 408
§19.2.1 GatewayClass 409
§19.2.2 Gateway 409
§19.2.3 HTTPRoute 410
§19.3 实战 412

https://www.veryopen.org

第 20 章 Serverless and Event-Driven Applications 414

第 21 章 二进制方式安装 K8S 415

§21.1 部署 etcd 415

§21.2 部署 kubernetes 418

§21.2.1 准备二进制程序 418

§21.2.2 部署 kube-apiserver 418

第 22 章 搭建本地源 422

§22.1 操作系统安装源 422

§22.2 k8s/containerd 源 424

§22.3 容器镜像库 425

§22.4 Helm Chart 库 430

第 23 章 排错 433

§23.1 docker 容器网络不通 433

§23.2 ~~Docker 官方镜像访问失败~~ 435

§23.2.1 ~~更改为国内镜像源~~ 435

§23.3 初始化时下拉镜像失败 435

§23.4 手工拉下容器镜像 436

§23.5 在 pod 中无法 ping 通外部的域名 437

§23.6 kubeadm init 报错 438

§23.7 手工导入镜像时报错 439

§23.8 部署应用时报错：下载镜像超时 439

§23.9 pod 之间网络不通 439

§23.10 pod 异常 440

§23.11 重置 etcd 集群 440

§23.12 无法访问 wordpress 页面 442

https://www.veryopen.org

https://www.veryopen.org

前言

一、容器的新变化

容器技术这一年（2025 下半年到 2026 上半年）确实发生了不少结构性变化，核心主线是：Kubernetes 全面转向 AI 基础设施、容器运行时进一步“去 Docker 化”、隔离与安全成为默认项、以及 Wasm 作为轻量运行时正式进入 CRI（Container Runtime Interface） 生态。

1. Kubernetes 1.36「Haru」（2026 年 4 月发布）

这是 2026 年第一个大版本，风格上不追求炫技特性，而是把几个拖了多年的老问题收尾，重点在稳定性和 AI/GPU 工作负载：

- DRA（Dynamic Resource Allocation）持续深化：这是这一版最大的看点。DRA 用基于属性（CEL 表达式）的 ResourceClaim 取代了老的 Device Plugin 模式，比如可以直接声明“任意一块显存 ≥ 40GB 的 GPU”，而不用绑死具体的 vendor/型号。v1.36 版本让 DRA 扩展到内存、CPU 这类原生资源，并支持 PodGroup 级别的 ResourceClaim，另外四个 DRA 相关特性进入 GA，其中最重要的是设备级维护支持——以前给 GPU 做维护必须整节点 drain，现在可以做更精细的滚动维护而不用打断分布式训练任务。值得关注的是 NVIDIA 在 KubeCon Europe 2026 把 DRA GPU 驱动捐给了 CNCF，Google 同时开源了 DRA TPU 驱动，说明 DRA 已经成为主流硬件厂商认可的标准接口，不再是 K8s 自己的孤岛方案。
- User Namespaces for Pods 转 Stable：这个特性把容器内的 UID 映射到宿主机上的非特权 UID，即便容器内进程以 root 身份运行，逃逸后在宿主机上也只是普通用户，算是多年安全债的一次了结。
- 其他 Stable 项：Fine-Grained Kubelet API Authorization、Service Account Token 外部签名 API、Mutating Admission Policies（用 CEL 表达式直接定义准入变更逻辑，不用再单独维护 webhook）、递归 SELinux 标签修改加速、VolumeGroupSnapshot、OCI VolumeSource、基于 cgroup v2 的 PSI 支持都进了 Stable。
- Ingress-Nginx 退场：网络这块有个方向性转折，Ingress-Nginx 逐步退休，Gateway API 成为更具扩展性的官方替代路径。如果你的集群还在用传统 ingress-nginx，这是需要提前规划迁移的事项。

2. 容器运行时：Docker 进一步被“架空”

Docker 本体没死，但它在整个技术栈里的位置持续被稀释：

- K8s 从 v1.24 起彻底移除 dockershim，不再原生支持 Docker Engine 作为运行时，containerd 或 CRI-O 才是现在的标准配置，这已经是运行了几年的生产常态，不是新闻。大规模集群里 containerd 2.0 或 CRI-O 1.31 是默认选择，AWS EKS、GKE、AKS 都默认 containerd。
- 但要澄清一个常见误解：K8s 弃用的是 Docker Engine 这个运行时，不是 Docker 镜像格式——你用 Dockerfile 构建的镜像照样能跑，因为 OCI 标准统一了镜像层。
- 本地开发环境：Podman 5 和 OrbStack 正在快速蚕食 Docker Desktop 的份额，主要卖点是 rootless 架构。Podman 默认以非 root 用户运行容器，2026 年一项针对 500 家企业的调查显示，Podman 5 的 rootless 模式比 Docker 27 少了 27%的安全漏洞，背景是 2025 年 8 月 Docker Desktop 曝出的 CVSS 9.3 容器逃逸漏洞（CVE-2025-9074）确实推了一把这个迁移趋势。不过 Podman 在 Windows 上性能仍明显弱于 Linux，桌面体验也不如 Docker 成熟。

3. 强隔离运行时成为多租户/Serverless 标配

多租户 SaaS 和 serverless 场景里，gVisor、Kata Containers、Firecracker MicroVM 已经成为标准隔离手段。简单区分一下适用场景：

- gVisor：用户态内核拦截 syscall，宿主机内核对容器进程完全不可见，性能有损耗但安全收益很大，

https://www.verypen.org

2026 年已是多个托管沙箱服务的默认运行时。

- Kata Containers：用 Firecracker 或 Cloud Hypervisor 之类的 microVM 做硬件级隔离，同时保持容器级的 API 兼容性和轻量 footprint，适合"要 VM 级隔离但不想丢 Docker/K8s 生态"的场景。
- Confidential Containers：用 AMD SEV-SNP、Intel TDX 做内存加密，把机密计算也纳入容器隔离范畴，这个方向在金融、医疗类多租户场景开始落地。
- 运行时层面，Sandbox API 已经稳定，gVisor/Kata 这类隔离运行时都通过这个标准 API 集成。

4. WebAssembly 正式进入容器生态（而不是替代它）

这是个容易被低估的变化：WasmEdge、Wasmtime、Wasmer 这些 Wasm 运行时通过 runwasi shim 正式接入 Kubernetes，作为 RuntimeClass 注册。实际用法是 Wasm 模块作为 OCI 兼容 artifact 分发，通过 containerd shim（如 io.containerd.wasmedge.v1）调度，和普通容器共存于同一个集群、同一套调度逻辑下。适用场景主要是边缘计算、插件化沙箱（比如 Docker 沙箱里跑不受信代码时，用 Docker 命名空间加上 Wasm 的内存安全做双重沙箱），以及需要毫秒级冷启动的 serverless 函数——它不是要取代 Linux 容器，而是多了一档更轻量的隔离选项。

5. Kubernetes 变成“AI 基础设施操作系统”

这是最大的叙事转变。根据 CNCF 2026 年 1 月的年度调研，82%的容器用户已在生产环境用 K8s，部署生成式 AI 模型的企业中有 66%会把部分或全部推理负载放在 K8s 上。行业讨论焦点已经从无状态 Web 应用转向分布式训练、LLM 推理、自主 AI Agent，逻辑是如果把数据处理、训练、推理、Agent 运行放在不同基础设施上运维复杂度会指数级上升，K8s 恰好给这些异构负载提供了统一底座。这也解释了为什么这一版 DRA、GPU 维护、gang scheduling 会被当作头等大事来推。

6. 供应链安全成为默认要求

2026 年供应链攻击已是最大威胁之一，最佳实践要求每次镜像构建自动生成 SBOM，用 Sigstore 等工具签名，集群准入控制器严格拦截未签名或缺 SBOM 的镜像，同时镜像基座全面转向 Distroless/Scratch，把体积压到几十 MB 并缩小攻击面。多集群方面，Karmada 这类方案在做跨云、跨地域的联邦调度和无感故障转移也在成为标配而不是加分项。

二、本书的一些约定

注意：不要使用微软的 edge 浏览器打开本文档，在拷贝命令的时候 edge 浏览器会在末尾增加空格，最好用谷歌浏览器 Chrome 或火狐浏览器 firefox。

一般采用 OpenStack 搭建 IaaS 云，即 OpenStack 用于管理虚拟机，而采用 kubernetes 搭建 PaaS 云，即 kubernetes 用于管理容器。现在最流行的云原生软件工程架构就是基于 kubernetes+containerd+devops 的。作为当代的 IT 学员和从业人员，在 AI 时代，掌握云原生这套技术大有裨益！

关于 OpenStack 的实战，请参考《OpenStack 云平台实战手册》，可从我的网站 https://www.veryopen.org 下载。

本书中的一些约定：

1. 命令行上的提示字符串，说明如下：

“[root@master1 ~]# ”表示在第 1 个管理节点上以 root 用户操作。

“[root@worker1 ~]# ”表示在第 1 个工作节点上以 root 用户操作。

https://www.veryopen.org

“[root@allnodes ~]# ”表示要在全部节点上以 root 用户操作。

“[root@worker? ~]# ”表示在所有工作节点上以 root 用户操作。

诸如如下的命令行：

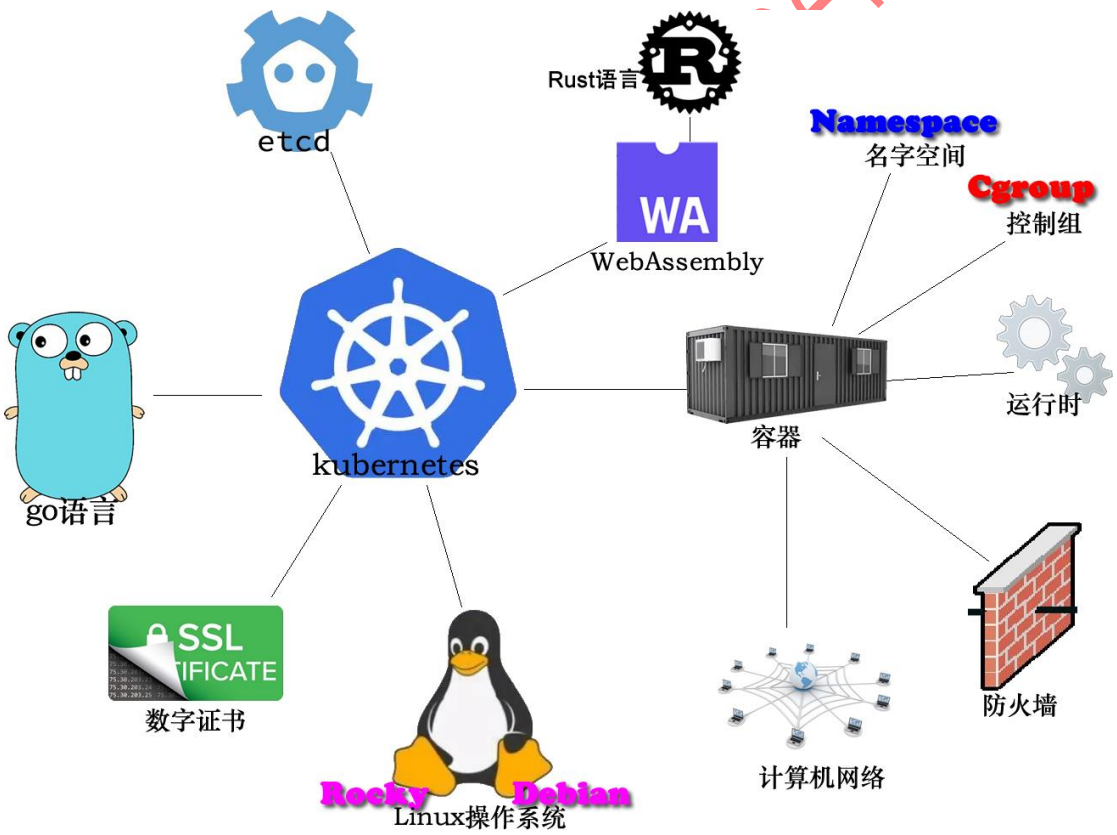
```
[root@master1 ~]# dnf install vim -y                                    #安装 vim
Last metadata expiration check: 5:11:12 ago on Fri 11 Aug 2023 11:30:24 AM CST.
Package vim-enhanced-2:8.2.2637-20.el9.x86_64 is already installed
……
```

最右侧的“#安装 vim”是解释，操作时不要输入，“[root@master1 ~]# ”是命令行提示符，会自动显示，不用输入。真正需要输入的是中间绿色字体部分“dnf install vim -y”，这才是命令本身。命令的输出用灰色字体表示。

veryopen.org

2．围绕 kubernetes 的知识点：

想要学好 kubernetes，还得学习其周边知识，还得学习周边的周边知识，……，参见下图。



图：Kubernetes 的周边知识点

课堂上我们讲解：容器、运行时、Go 语言、数字证书、etcd 数据库和 kubernetes。如果时间允许，再讲名字空间。

3．版权所求：

对本文的引用，请注明“来自 https://www.veryopen.org”，谢谢。

https://www.veryopen.org

------- 上 篇 -------

kubernetes 前导知识。

https://www.veryopen.org

第1章 系统设计

我们采用八台机器来部署 Kubernetes 集群，其中控制面三台，分别取名 master1、master2 和 master3，工作节点三台，分别取名 worker1、worker2 和 worker3，两台负载均衡器，分别取名 balancer1 和 balancer2，当然也可以采用更多的计算节点和控制面节点，计算节点是用来跑 pod 的。在生产环境中，为了提高可用性，一般采用至少三台控制面节点和至少五台的计算节点。不过做实验时，整个集群只用一台机器也是可以的。

由于实验室的条件限制，所以我们的实验环境采用虚拟机，设计如图 1-1 所示，虚拟机不采用桥接网卡，因为桥接网卡需要配置实验室的局域网 ip，而实验室的 ip 数量有限，不能满足几十位同学一起做实验的要求。所以全部的虚拟机网卡都是 NAT 模式。

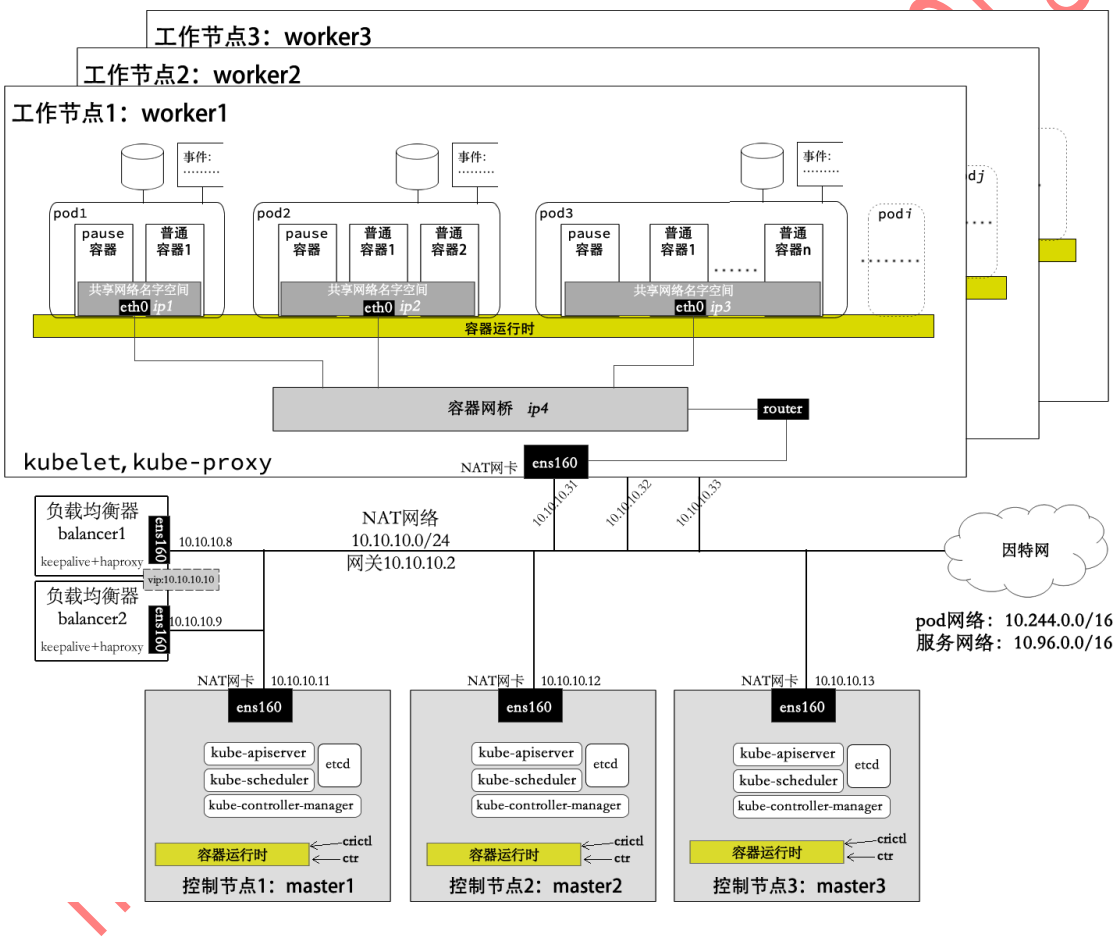


图 1-1 实验环境拓扑图

我们的实验环境是这样的：在一台配置 16GB 内存的电脑上安装 windows 10，在 windows 上安装 VMware Workstation pro 17，然后在 VMware 里创建八台虚拟机，虚拟机的配置参考表 1-1。

表 1-1 八台虚拟机的配置

角色	主机名	内存	磁盘	ip	进程
控制面	master1	4G	32G	10.10.10.11	kube-apiserver/kube-controller-manager/kube-proxy/etcd/kube-scheduler/kubelet/containerd
	master2	4G	32G	10.10.10.12	kube-apiserver/kube-controller-manager/kube-proxy/etcd/kube-scheduler/kubelet/containerd
	master3	4G	32G	10.10.10.13	kube-apiserver/kube-controller-manager/kube-proxy/etcd/kube-scheduler/kubelet/containerd
工作节点	worker1	3G	32G	10.10.10.31	kube-proxy/kubelet/containerd /业务 pod
	Worker2	3G	32G	10.10.10.32	kube-proxy/kubelet/containerd /业务 pod
	Worker3	3G	32G	10.10.10.33	kube-proxy/kubelet/containerd /业务 pod
负载均衡 10.10.10.10	balancer1	1G	8G	10.10.10.8	keepalive/haproxy
	balancer2	1G	8G	10.10.10.9	keepalive/haproxy

控制面节点的内存不能少于 3G，工作节点的内存由上面跑的容器决定，一般不能少于 2.5GB。我们做实验分成三步：首先采用四个节点（master1、worker1、worker2 和 worker3）搭建基本的 k8s 环境，然后增加两个空面节点（master2 和 master3），最后增加负载均衡器（balancer1 和 balancer2）。

参考图 1-2 修改 VMware 的 NAT 网络参数。

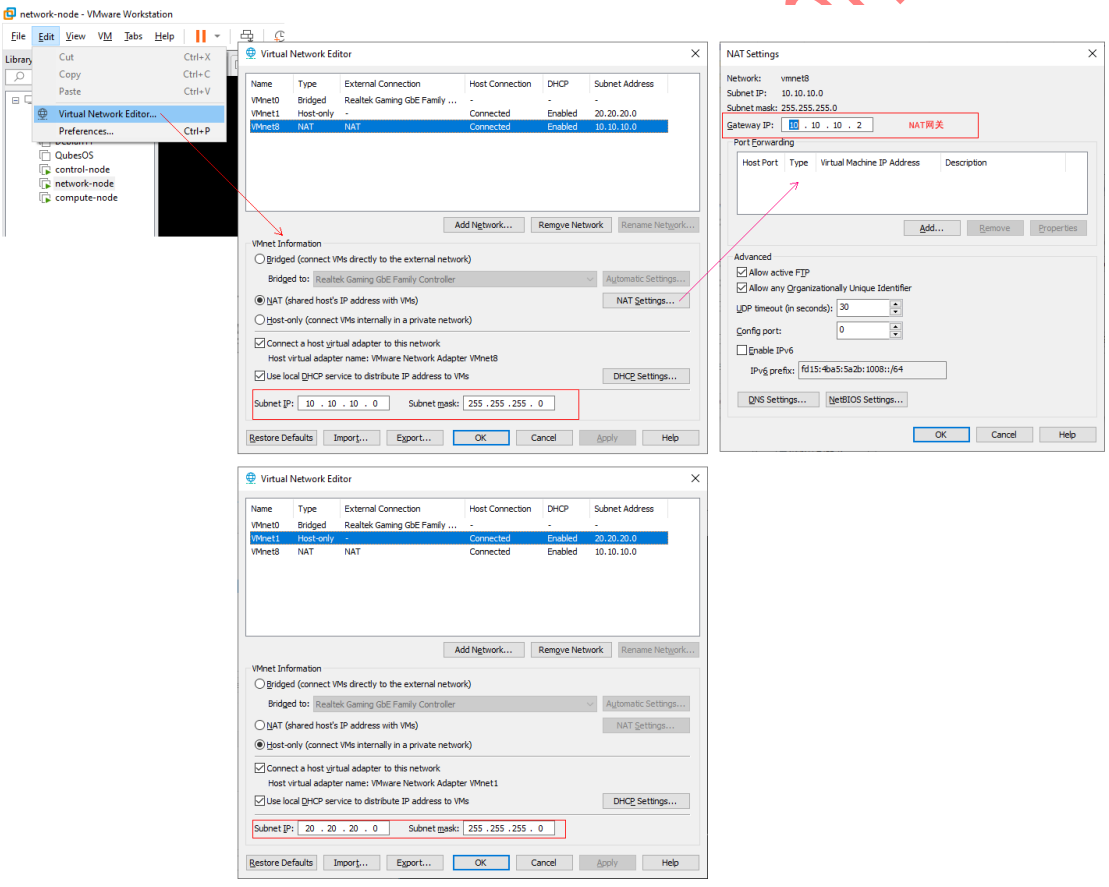


图 1-2 设置 VMware 的 NAT 网络参数

NAT 网卡的静态 Ip 要避开动态地址池，比如图 1-3 所示的动态地址池为 128 到 254，所以小于 128 的都是静态地址。

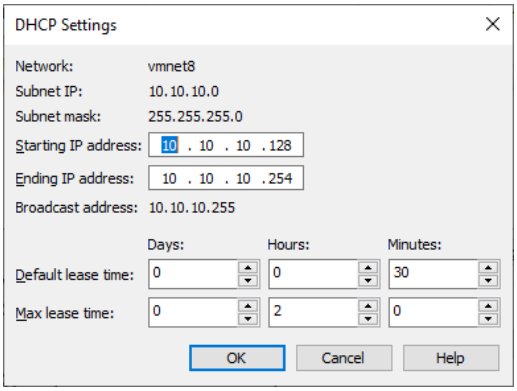


图 1-3 NAT 网络的动态地址池

要确保宿主机（即 windows 机器）上的虚拟网卡 Vmnet8（虚拟机上的 NAT 模式的网卡需要用到）被启用，参考图 1-4 所示。

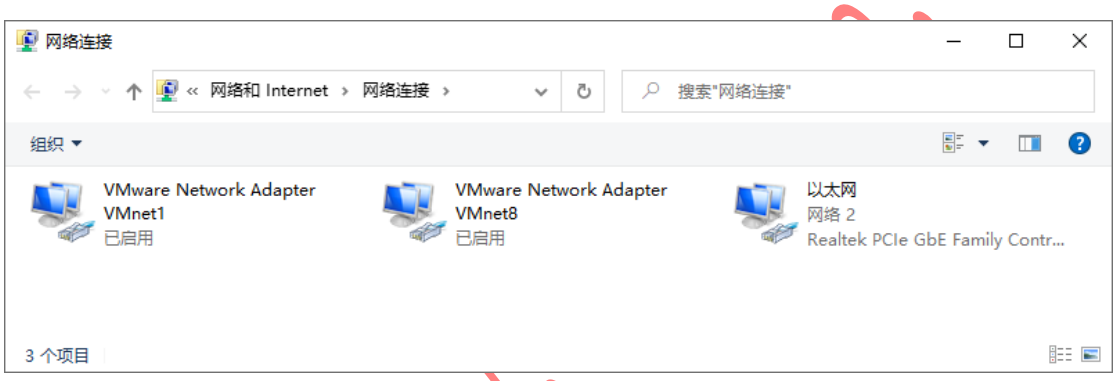


图 1-4 启用宿主机上的 Vmnet1 和 Vmnet8 虚拟机网卡

最终的虚拟机如图 1-5 所示。

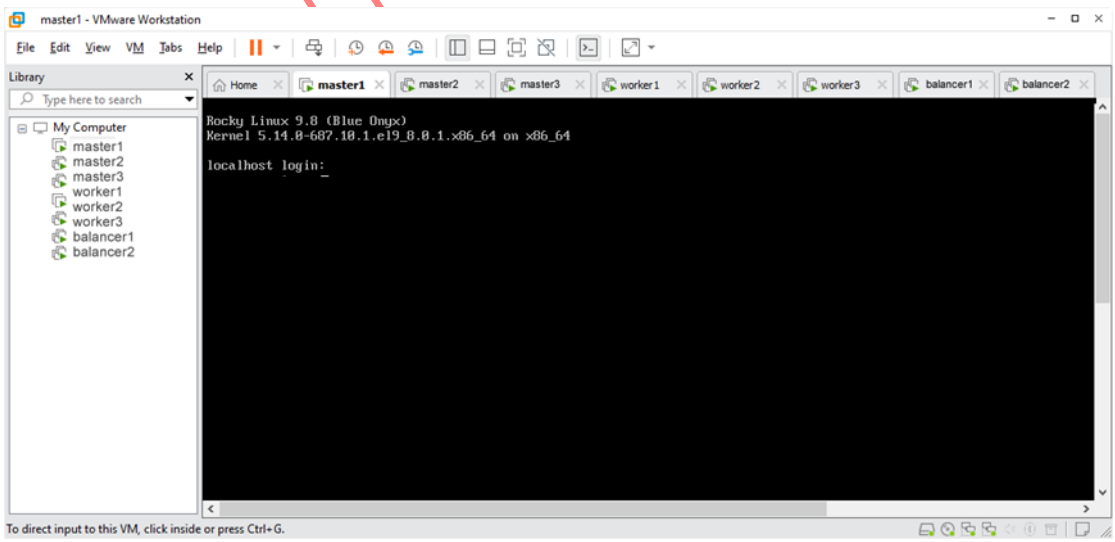


图 1-5 八台虚拟机

对于一台已经安装了操作系统的虚拟机，如果想再次从光盘启动重新安装 Linux 操作系统，可以这样做：关闭虚拟机，然后启动虚拟机的 3 秒钟内，不断往虚拟机屏幕里点鼠标的同时不断按 F2 键，进入虚拟机的 BIOS 设置界面，然

后进入菜单 Boot，再按上下光标键选中 Hard Drive 后按-（减号）下移它，让 CD-ROM Drive 位于上面，这样下次开机就从光盘启动虚拟机了。最后按 F10 存盘退出。操作请参考图 1-6。

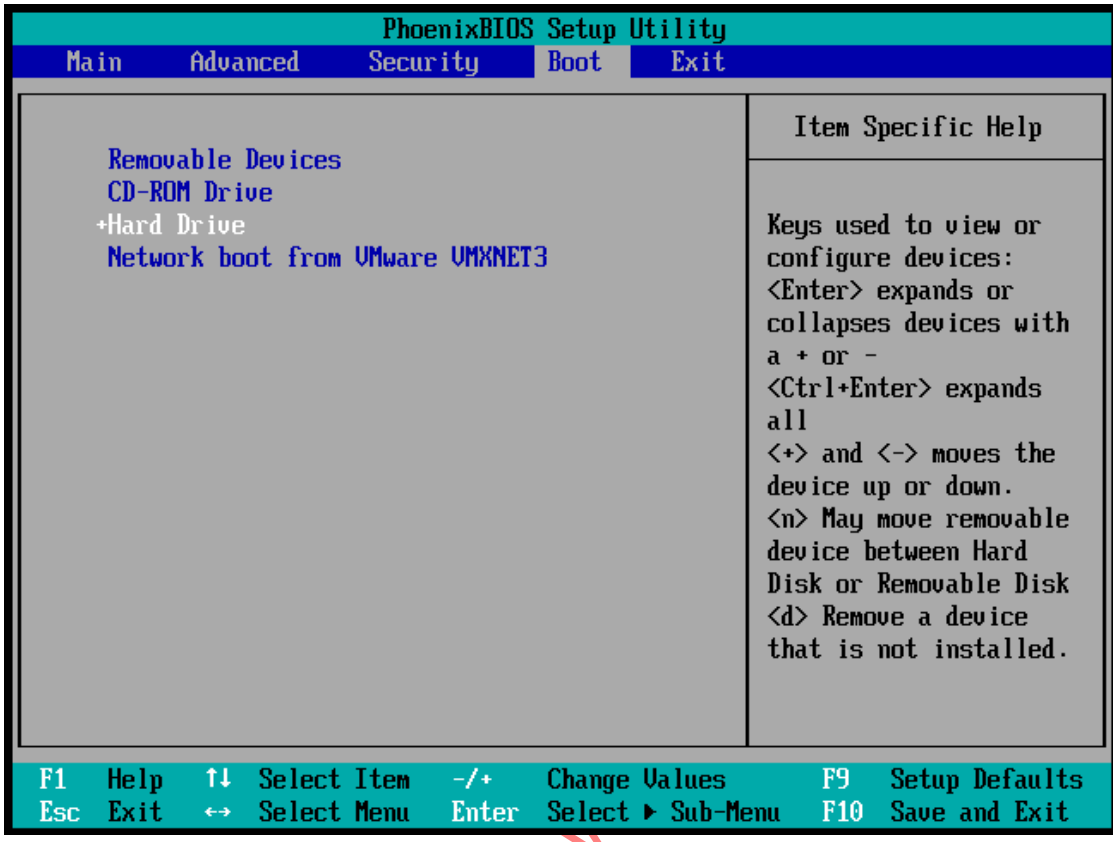


图 1-6 修改虚拟机的启动磁盘顺序

在具体实战过程中，我们可以先用一台控制面节点 master1 → 然后加入三台工作节点 → 部署应用 → 加入另外两台控制面节点 master2 和 master3，启用一台负载均衡节点 balancer1 → 模拟控制面节点故障（比如关闭 master3） → 启用另外一台负载均衡节点 balancer2 → 模拟从外面访问应用 → 模拟一台负载均衡故障（比如关闭 balancer1）。

https://www.

第2章 准备四个节点



大概的步骤：参考第一章创建一台虚拟机——命名为 master1，虚拟机里安装 Rocky Linux 9，升级操作系统到最新状态，关闭 SELinux 安全模块，设置网卡参数、主机名、NAT 网卡的 Ip，默认网关和 DNS 设在 NAT 网卡上。假设虚拟机的镜像文件位于宿主机的 D:\wlm\master1 目录中。

然后克隆三台虚拟机，分别取名 worker1、worker2、worker3。

§2.1 安装 Rocky

安装 Rocky Linux 9（可从 https://mirrors.nju.edu.cn/rocky/9.8/isos/x86_64/Rocky-9-latest-x86_64-minimal.iso 下载）。最好安装中文和英语两种语言，不要交换分区，参考图 2-1 所示的安装摘要。

https://www.veryopen.



图 2-1 Rocky Linux 9 的安装摘要

点击上图中的“安装目的地”选择手工硬盘分区，只分一个根分区即可，交换区不需要，参考图 2-2 所示操作（按照顺序号操作）。kubernetes 集群中的全部节点都要关闭交换区，后面会讲到如何关闭。当然如果安装操作系统时没有创建交换区，那么交换区已经是关闭了。



图 2-2 手工分区

虚拟机的硬盘只有 32GB，要把整个硬盘给根文分区，则在输入“期望容量”时只要大于 32GB 即可让根分区占满整块硬盘。

§2.2 远程登录

强烈建议从宿主机上（windows 10）远程登录到虚拟机中操作，这样内外复制/粘贴都很方便——windows 这边按 Ctrl+c 复制，Linux 那边点鼠标右键粘贴，或者 linux 那边选中内容并点右键（复制），windows 这边按 Ctrl+v 粘贴。

windows 自带了 ssh 命令，允许登录到虚拟机，参考图 2-3 所示，如果是第一次登录，需要输入 yes 确定指纹数据。

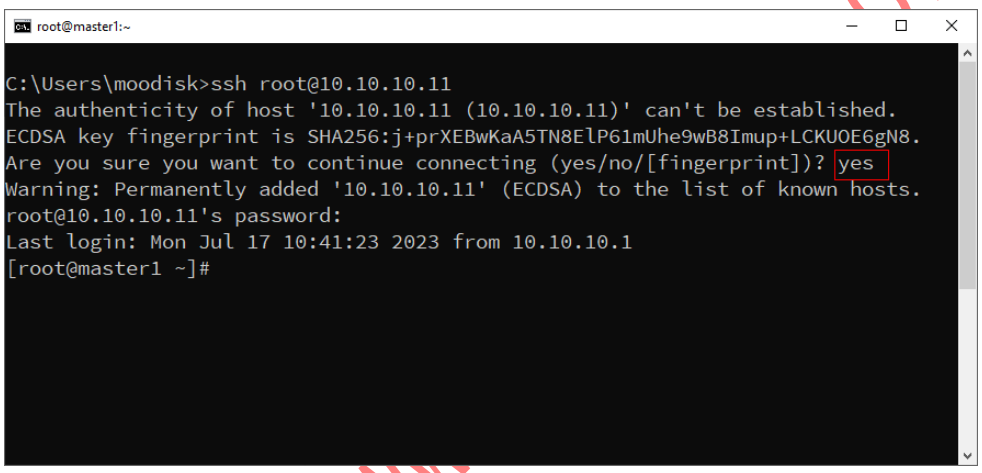


图 2-3 从 windows 登陆到 Linux

如果没法用 root 用户登陆，那是因为在图 2-1 中没有勾选“允许 root 用户使用密码进行 ssh 登陆”。请在虚拟机中执行下面的命令即可：

```
[root@master1 ~]# echo "PermitRootLogin yes" > /etc/ssh/sshd_config.d/01-permitrootlogin.conf
[root@master1 ~]# systemctl restart sshd
```

上面第一条命令产生新文件/etc/ssh/sshd_config.d/01-permitrootlogin.conf，文件内的只有一行 PermitRootLogin yes。

windows 自带的 ssh 命令有一个缺陷，就是滚动屏幕的幅度有限（稍微靠前一些的历史内容就看不到了）。如果你的 windows 没有 ssh 命令，就从网上下载登录小工具 putty.exe，不需要安装，直接运行它即可，参考图 2-4。

https://www.org

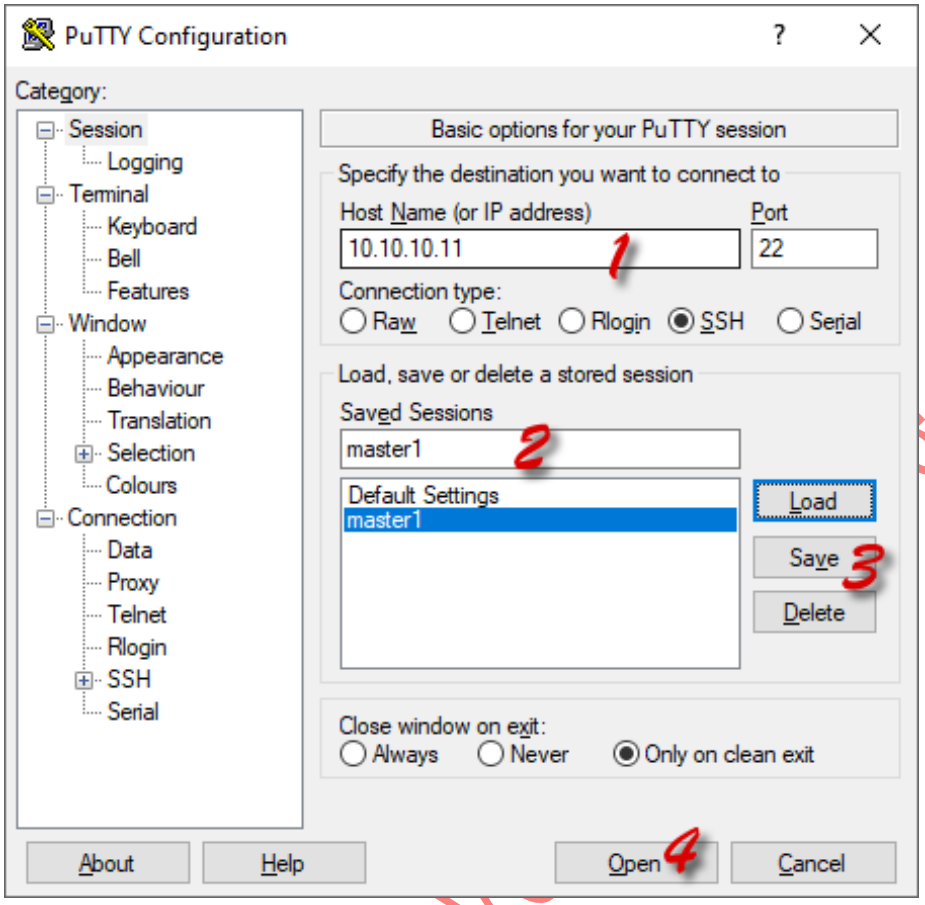


图 2-4 putty.exe 登录 Linux

把经常要登录的主机取名（比如 master1），并保存 ip 地址等参数，以后只要双击 master1 就可以登录。windows 上另一个超级好用的登录工具是 mobaXterm，可从 https://mobaxterm.mobatek.net/download-home-edition.html 下载免费版，它支持 ssh 最新版的公钥登录，而 putty.exe 不支持。可参考图 2-5 所示配置字体和字体大小。

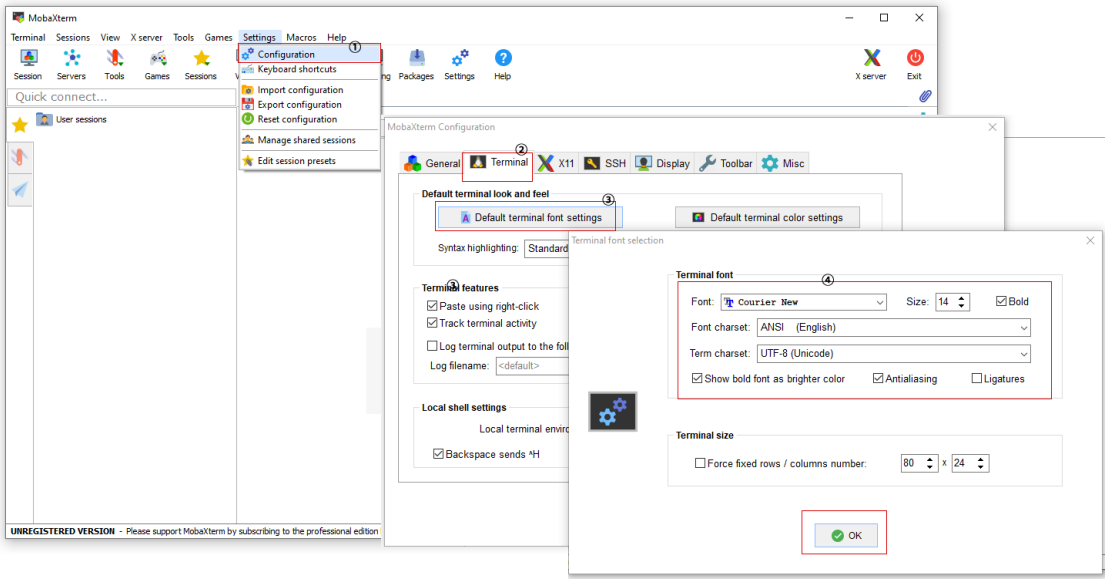


图 2-5 配置 mobaXterm 字体

然后登录，参考图 2-6 操作：

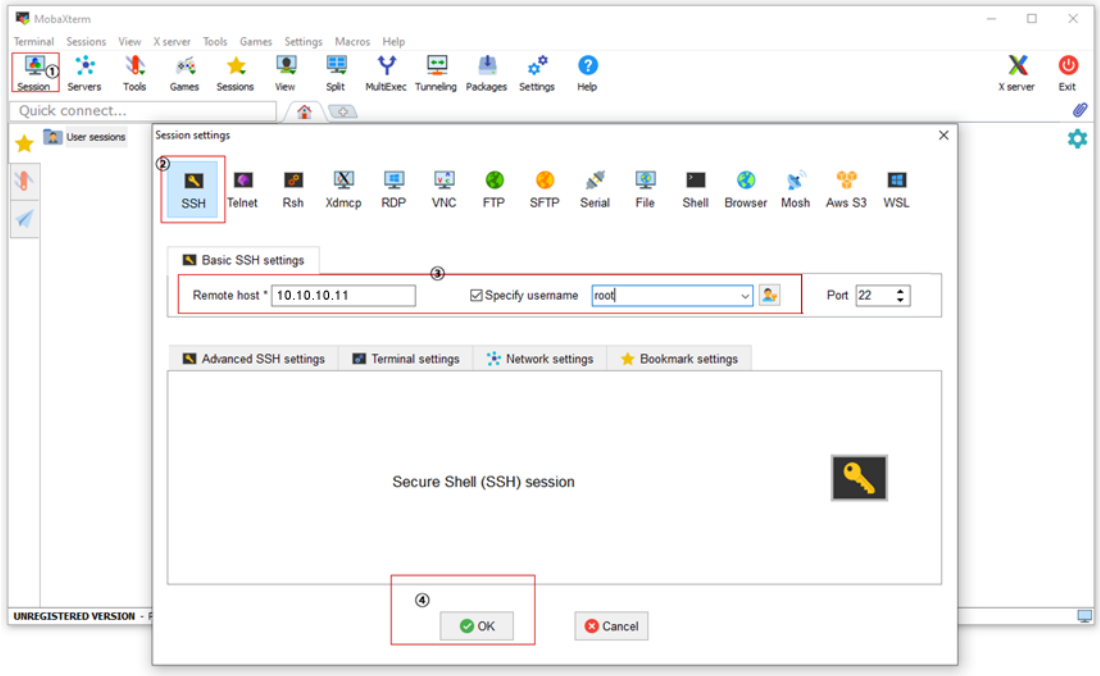


图 2-6 mobaXterm 登录会话

登录之后的画面参考图 2-7 所示。

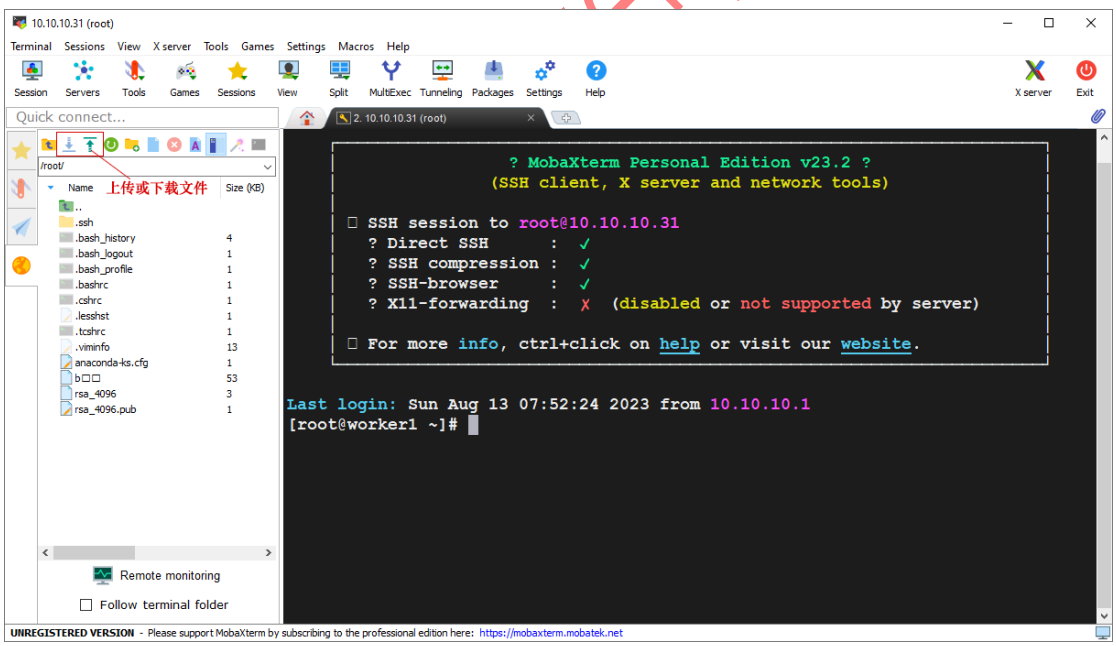


图 2-7 登录之后的界面

以后第一次按鼠标邮件会弹出如图 2-8 所示的画面，点选“Paste”指示以后点击鼠标右键就是粘贴。

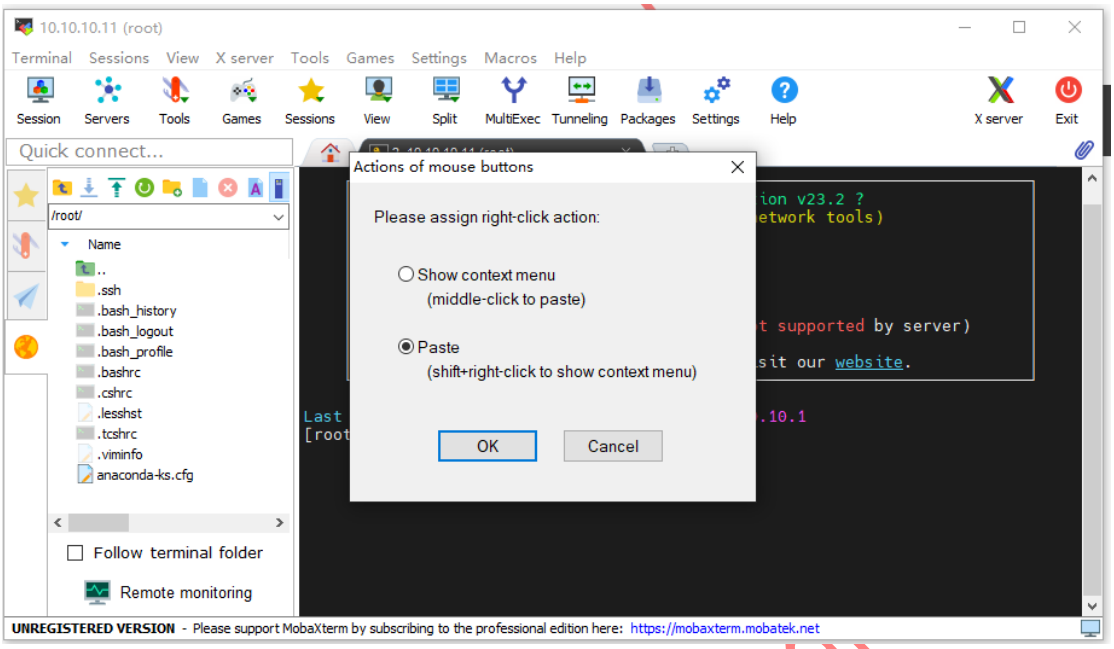


图 2-8 设置鼠标邮件为粘贴

现在启动 master1。

如果需要从其他电脑登录到虚拟机（比如老师要从教师机登入到学生的虚拟机内排错），需要在 VMware 虚拟化软件中设置 NAT 转换，参考图 2-9 操作。

https://www.veryopen

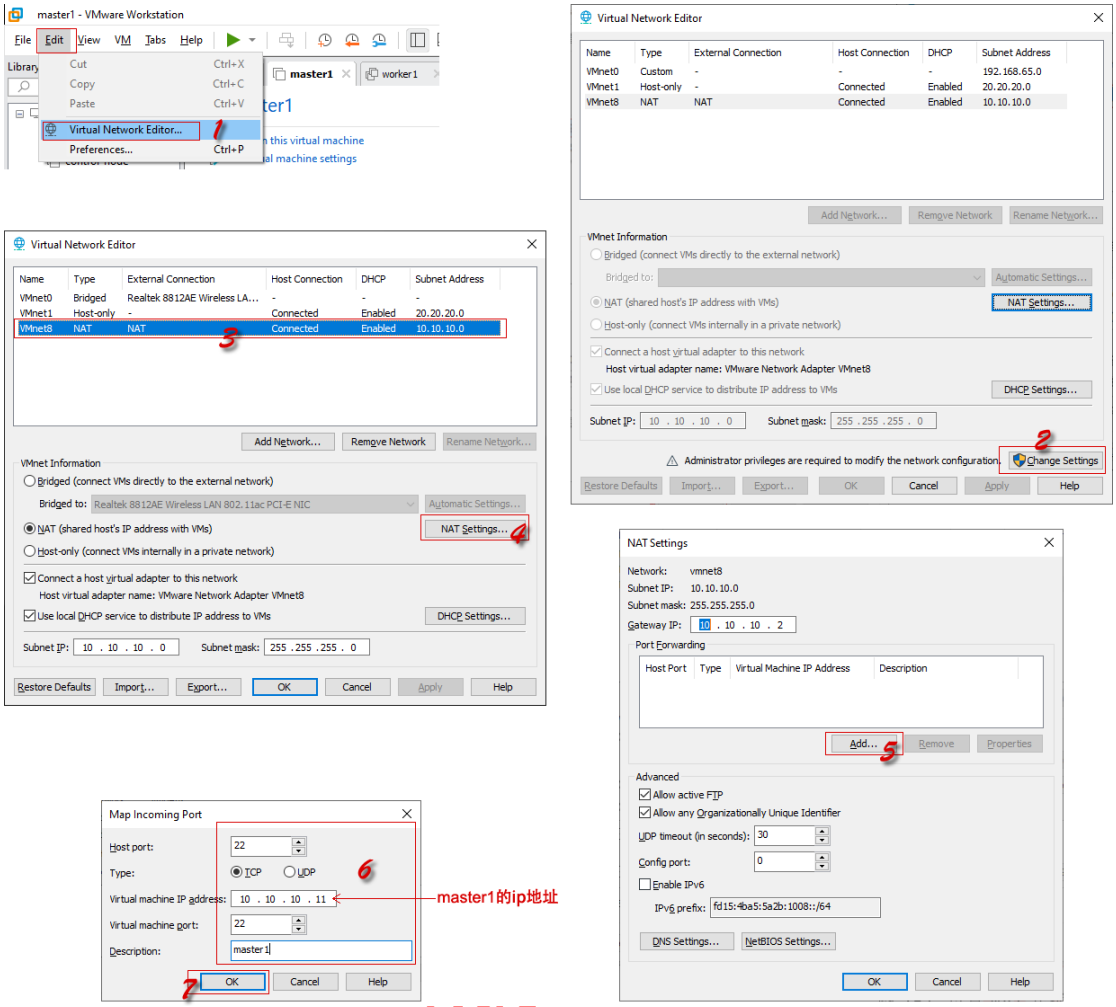


图 2-9 设置 NAT 转换

以后就可以通过登陆到宿主机（windows）的 ip 进入控制节点了，比如参考类似图 2-10 所示查看 windows 的网卡 ip 为 10.3.106.80，因此从其他电脑采用命令 ssh root@10.3.106.80 登录到控制节点了。

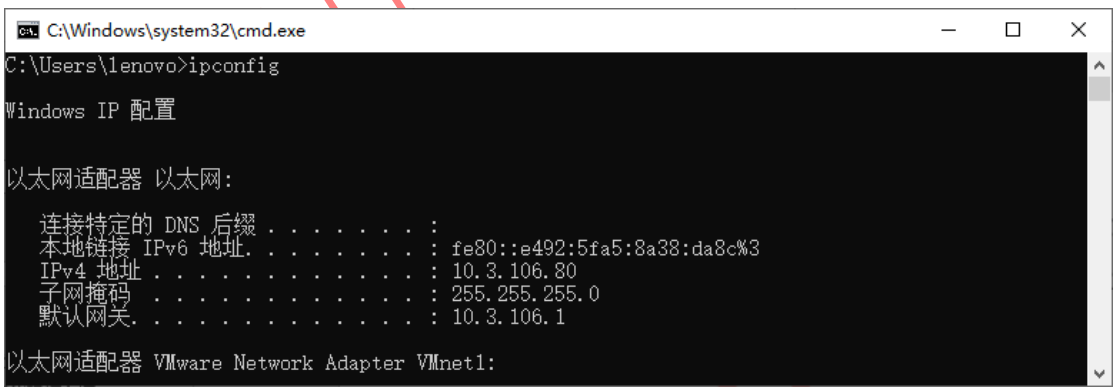


图 2-10 查看 windows 的网卡 Ip

如果不能登录，可能是 windows 的防火墙阻止了，请关闭防火墙或者放行端口 22/tcp。

§2.3 基本配置

1. 设置主机名

```
[root@localhost ~]# hostnamectl set-hostname master1
[root@localhost ~]# su -
[root@master1 ~]#
```

控制节点设置主机名 master1。

2. 配置网卡参数（如果在安装时已经设置了网卡 ip，现在就不要做）：

```
[root@master1 ~]# nmcli c                                                    #查看网卡的名字
NAME    UUID                                  TYPE      DEVICE
ens160  52608ad1-ce77-3762-aa54-332e8b661466  ethernet  ens160
lo      41122259-6e92-43f8-95e4-5c56ada17f9a  loopback  lo
```

或者采用命令 ip a 查看网卡的名字。

Rocky 9 开始使用 NetworkManager 管理网络，网卡的配置文件放在/etc/NetworkManager/system-connections/目录中，可以直接修改这些网卡配置文件来设置 ip 地址。我们这里采用命令 nmclic 来修改：

```
[root@master1 ~]# nmcli device status
DEVICE  TYPE      STATE                   CONNECTION
ens160  ethernet  connected               ens160
lo      loopback  connected (externally)  lo

[root@master1 ~]# nmcli connection modify ens160 ipv4.method manual                    #配置静态 ip
[root@master1 ~]# nmcli connection modify ens160 ipv4.addresses 10.10.10.11/24 ipv4.gateway 10.10.10.2
[root@master1 ~]# nmcli connection modify ens160 ipv4.dns "8.8.8.8 9.9.9.9"
[root@master1 ~]# nmcli connection up ens160                                  #使新的 ip 起作用

[root@master1 ~]# more /etc/NetworkManager/system-connections/ens160.nmconnection
[connection]
id=ens160
uuid=52608ad1-ce77-3762-aa54-332e8b661466
type=ethernet
autoconnect-priority=-999
interface-name=ens160
timestamp=1689553683

[ethernet]

[ipv4]
address1=10.10.10.11/24
dns=8.8.8.8;9.9.9.9;
gateway=10.10.10.2
method=manual

[ipv6]
addr-gen-mode=eui64
method=auto

[proxy]
```

nmcli 是用来管理网络的强大工具，此命令的参数在不产生歧义的情况下允许采用简略字符，如 connection 简略为 c，modify 简略为 mod，addresses 简略为 addr，所以上面的命令“nmcli connection up ens160”与“nmcli c up ens160”等价。网卡配置文件分为若干节，如[connection]、[ipv4]等，在 nmcli 命令中通过格式“节名.参数”指定具体要修改的参数，如上述命令中的“ipv4.method manual”表示把[ipv4]节中的参数 method 的值改为 manual，同样如果要修改[connection]节中的 id 参数，可用命令“nmcli c mod connection.id ens160-nat”。有的参数允许多值，这时可采用这样的格式来修改参数的值：+节名.参数 值、-节名.参数 值、节名.参数 值。添加“+”前缀表示增加值，“-”前缀表示删除值，没有“+”或“-”前缀表示设定值，例如“nmcli c mod ens160 +ipv4.addr 10.10.10.10/24”给网卡再增加一个 ip 地址，“nmcli c mod ens160 ipv4.addr 10.10.10.11/24”把网卡的 ip 设成 10.10.10.11。

DNS 务必要采用 8.8.8.8 和 9.9.9.9，国内提供的其他 DNS 污染了一些国外域名，会导致诸如容器镜像无法下拉的问题。

如果网卡配置乱了，ip 地址总是起不来，建议先删除现有的链接（采用命令 nmlci c delete <网卡 uuid>、再 rm /etc/NetworkManager/system-connections/ens160.nmconnection），然后重新添加链接（采用命令 nmcli c add type ethernet ifname ens160 con-name ens160），最后采用上面的 nmlci connection …系列命令修改网卡参数即可。

把下面四行加入到/etc/hosts 文件的末尾，这样就实现了主机到 Ip 的解析了。

```
[root@master1 ~]# vi /etc/hosts
……
10.10.10.11                    master1
10.10.10.31                    worker1
10.10.10.32                    worker2
10.10.10.33                    worker3
```

如果重启虚拟机后发现第一块网卡没有 Ip，不妨运行“nmcli n on”命令试试。

3．关闭 SELinux 安全模块

SELinux 为安全增强 Linux 系统，利用它能打造一台军用安全级别的 Linux 电脑，但是配置起来很麻烦，设置不当很容易导致各种问题，所以为了简化我们的实验难度，建议关闭它。

(1)．先查看 SELinux 安全系统的状态：

```
[root@master1 ~]# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   enforcing
Mode from config file:          enforcing
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Memory protection checking:     actual (secure)
Max kernel policy version:      33
```

表明 SELinux 的状态是激活的，并且是增强模式，在实验环境中，我们把它关闭，但在生产环境中，最好不要关闭。在开启了 SELinux 安全系统后，部署的每一个软件都需要花费大量的精力做相应的安全配置才行。

(2)．把 SELINUX=enforcing 改为 SELINUX=disabled：

```
[root@master1 ~]# vi /etc/sysconfig/selinux
……
SELINUX=disabled
……
```

重启电脑后再查看 SELinux 的状态：

```
[root@master1 ~]# sestatus
SELinux status:                 disabled
```

4．配置 Rocky 安装源

我们学校有 Rocky 9 的本地镜像库，做实验时如能访问 10.8.20.8 就配置本地镜像库，否则请配置南京大学的镜像库。

下面两种方法请选择一种。

【选择 1】配置本地镜像库：

```
[root@master1 ~]# cd /etc/yum.repos.d
[root@master1 yum.repos.d]# mv rocky.repo rocky.repo.bak                    #rocky.repo 改名为 rocky.repo.bak
[root@master1 yum.repos.d]# mv rocky-addons.repo rocky-addons.repo.bak
[root@master1 yum.repos.d]# mv rocky-extras.repo rocky-extras.repo.bak
[root@master1 yum.repos.d]# cd

[root@master1 ~]# vi /etc/yum.repos.d/rocky.repo
```

https://www.open.org

```
[baseos]
name=Rocky Linux $releasever - BaseOS
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/baseos/
gpgcheck=0
enabled=1

[appstream]
name=Rocky Linux $releasever - AppStream
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/appstream/
gpgcheck=0
enabled=1

[crb]
name=Rocky Linux $releasever - CRB
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/crb/
gpgcheck=0
enabled=1

[root@master1 ~]# vi /etc/yum.repos.d/rocky-extras.repo
[extras]
name=Rocky Linux $releasever - Extras
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/extras/
gpgcheck=0
enabled=1


[root@master1 ~]# vi /etc/yum.repos.d/rocky-addons.repo
[resilientstorage]
name=Rocky Linux $releasever - ResilientStorage
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/resilientstorage/
gpgcheck=0
enabled=1

[rt]
name=Rocky Linux $releasever - RT
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/rt/
gpgcheck=0
enabled=1
```

【选择 2】配置南京大学的镜像库：

```
[root@master1 ~]# sed -e 's|^mirrorlist=|#mirrorlist=|g' \
 -e 's|^#baseurl=http://dl.rockylinux.org/$contentdir|baseurl=https://mirror.nju.edu.cn/rocky|g' \
 -i.bak \
 /etc/yum.repos.d/rocky*.repo
```

每个安装源配置文件包含若干节（中括号内的就是节名），每一节代表一个软件库，节里的配置主要包括网址和激活状态，enabled=1 表示激活，enabled=0 表示禁用，安装软件时只在激活的软件库中查找并下载。

注意：每一次修改了安装源配置文件，必须要重新下载安装源的配置信息(采用命令 dnf makecache)。

```
[root@master1 ~]# dnf clean all                                                    #清除本地磁盘中旧的缓存
[root@master1 ~]# dnf makecache -y                                                 #更新安装源缓存
[root@master1 ~]# dnf repolist --enabled                                           #查看可用的安装源
仓库 id             仓库名称
appstream                                                 Rocky Linux 9 - AppStream
baseos                                                    Rocky Linux 9 - BaseOS
crb                                                       Rocky Linux 9 - CRB
extras                                                    Rocky Linux 9 - Extras
resilientstorage                                          Rocky Linux 9 - ResilientStorage
rt                                                        Rocky Linux 9 - RT
```

也可以采用其他国内的安装源，比如上海交通大学（mirror.sjtu.edu.cn）、中科大（mirrors.ustc.edu.cn）或者阿里云（mirrors.aliyun.com）。

打上操作系统的全部补丁并重启：

```
[root@master1 ~]# dnf upgrade -y
[root@master1 ~]# shutdown -r now
```

5. 安装常用的软件

下载工具 wget、增强的编辑器 vim、网站页面源码下载工具 curl、网络工具 nmap-ncat、网络抓包工具 tcpdump、JSON 分析工具 jq、时间同步工具 chrony 和以树形格式现实目录结构 tree 等，后面需要用到这些，请安装它们：

```
[root@master1 ~]# dnf -y install wget vim curl nmap-ncat tcpdump jq chrony tree man\* git iproute-tc tar
```

man*是在线帮助文件。

配置 vim 的工作环境：

```
[root@master1 ~]# vim ~/.vimrc
set autoindent
set cindent
set smartindent
syntax enable
syntax on
set nu
set nowrap
set softtabstop=-1
set shiftwidth=4
set expandtab
set tabstop=4
set hlsearch
set ruler
set showcmd
set cinoptions={0,1s,t0,n-2,p2s,(03s,=.5s,>1s,=1s,:1s
filetype on
colorscheme desert
```

显示行号（set nu）、语法高亮（syntax on）、不折行（set nowrap）、自动缩进（set autoindent）、Tab 键转换为四个空格（set expandtab）、根据文件扩展识别语言（filetype on）等。

6．检查时区

```
[root@master1 ~]# timedatectl
               Local time: Sat 2022-04-23 14:58:53 CST
           Universal time: Sat 2022-04-23 06:58:53 UTC
                 RTC time: Sat 2022-04-23 06:58:52
                Time zone: Asia/Shanghai (CST, +0800)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: no
```

如果时区不是亚洲的上海时区，请采用下面的命令修改：

```
[root@master1 ~]# timedatectl set-timezone Asia/Shanghai
```

7．确保防火墙已经启动

```
[root@master1 ~]# firewall-cmd --state
running                                                          #表名防火墙正在运行
[root@master1 ~]# firewall-cmd --list-all                        #查看防火墙规则
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: ens160
  sources:
  services: cockpit dhcpv6-client ssh
  ports:
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

确保网卡 ens160 已经成为 public 区的网络接口，ssh 协议（远程登录）允许通过防火墙。

如果没有启动防火墙，请采用下面的命令激活并启动它：

```
[root@master1 ~]# systemctl --now enable firewalld
```

!!! 在开启防火墙的情况下，各个 pod 之间 ping 不通，关闭防火墙就 ping 得通，继续找原因!!!

8．确保交换区已经关闭

```
[root@master1 ~]# free
               total        used        free      shared  buff/cache   available
Mem:         3980604      441084     3604712        9192      118824     3539520
Swap:              0           0           0
```

如果没关闭，请执行下面的命令：

```
[root@master1 ~]# swapoff -a
[root@master1 ~]# vim /etc/fstab
……
#/dev/mapper/cs-swap     none                    swap    defaults        0 0
```

即在行首增加“#”注释掉交换区（下次开机就不会自动挂载）。

9．加载两个额外的 Linux 内核模块

kubernetes 需要用到 overlay 和 br_netfilter 两个 linux 内核模块，需要开机时自动加载它们：

```
[root@master1 ~]# vim /etc/modules-load.d/containerd.conf                    #加入文件的模块，下次开机时会自动加载
overlay
br_netfilter
```

并开启一些内核参数，使得内核支持转发（虚拟机路由器）：

```
[root@master1 ~]# vim /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
```

在机器启动时需要自动加载的内核模块放在/etc/modules-load.d/目录下的任何文件中即可。在电脑启动时需要自动启用的内核参数放在/etc/sysctl.d/目录下的任何文件中即可，文件名中开头的数字代表启用的先后顺序，因为有些内核参数存在关联性，比如参数 A 一定要在参数 B 启用后才能企用，所以最佳的实践是文件取名类似“99-…….conf”这样的名字。

k8s 需要额外用到 overlay 和 br_netfilter 两个模块，前者实现联合文件系统，后者将桥接的流量转发至 iptables 链（在虚拟网桥上实现网络过滤）。三个内核参数 net.bridge.bridge-nf-call-iptables 表示网桥设备在二层转发时是否去调用 iptables 配置的三层规则（包含 conntrack），值为 1 表示开启，从而实现了同一节点内的服务通信。net.bridge.bridge-nf-call-ip6tables 内核参数针对 ipv6 的，而 net.ipv4.ip_forward=1 表示启用 Linux 的 ipv4 转发，从而实现路由功能。

10．禁用图形桌面

如果安装并默认启动了图形桌面，会消耗很多内存，建议采用下面的命令禁用图形桌面：

```
[root@master1 ~]# systemctl set-default multi-user.target
[root@master1 ~]# dnf group remove "Server with GUI"                         #卸载图形桌面
```

11．重启然后关机：

```
[root@master1 ~]# shutdown -r now                                            #重启
[root@master1 ~]# shutdown -h now                                            #关机
```

到此我们准备好了 master1 节点的基本系统。

https://www.very...org

§2.4 准备工作节点

关闭 master1 虚拟机，然后克隆另外三台虚拟机，分别取名 worker1、worker2 和 worker3。克隆过程请参考图 2-11 操作。

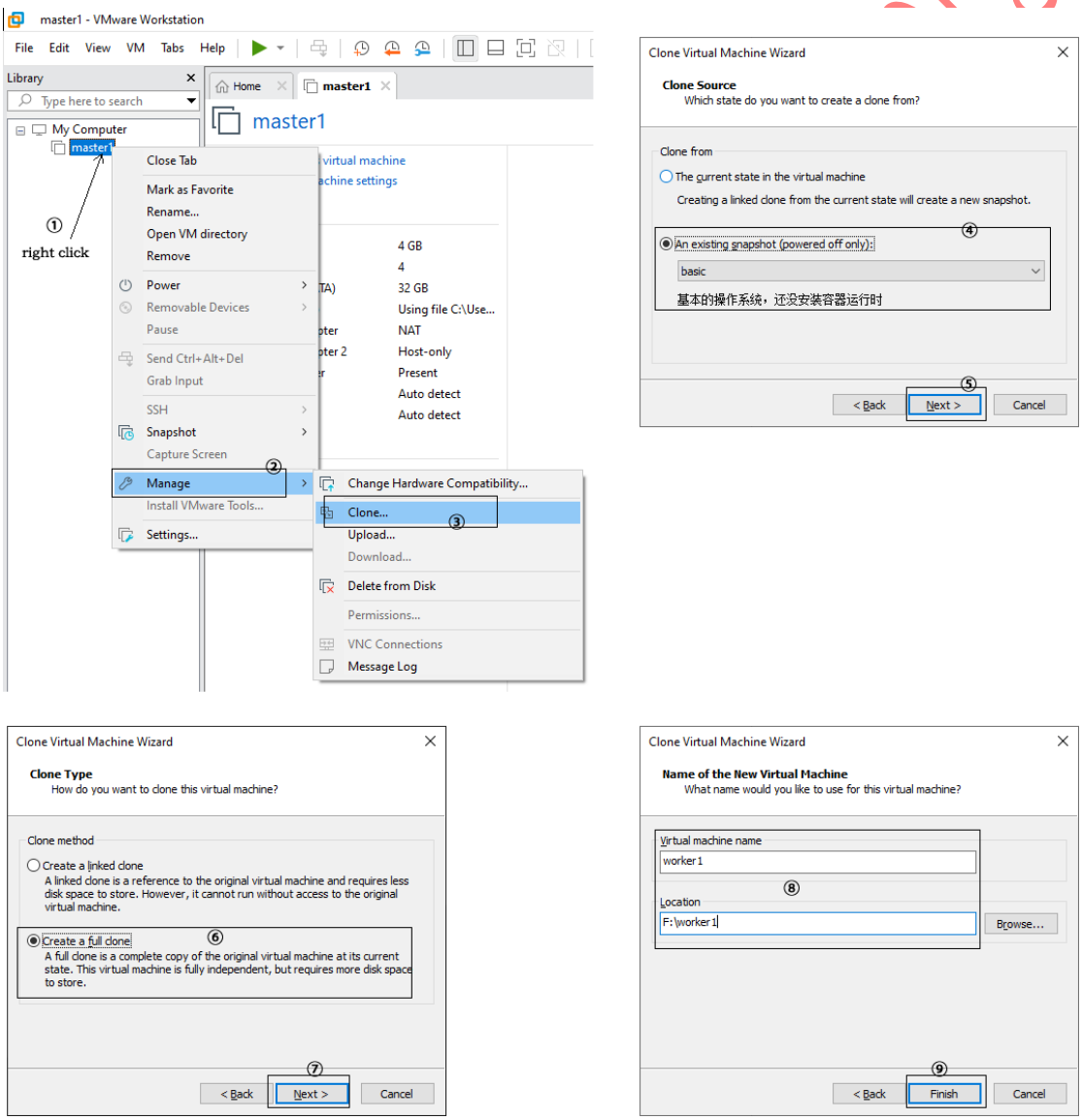


图 2-11 克隆虚拟机

把内存调整为 3GB，重新生成 worker1 的网卡 MAC 地址，参考图 2-12 操作。

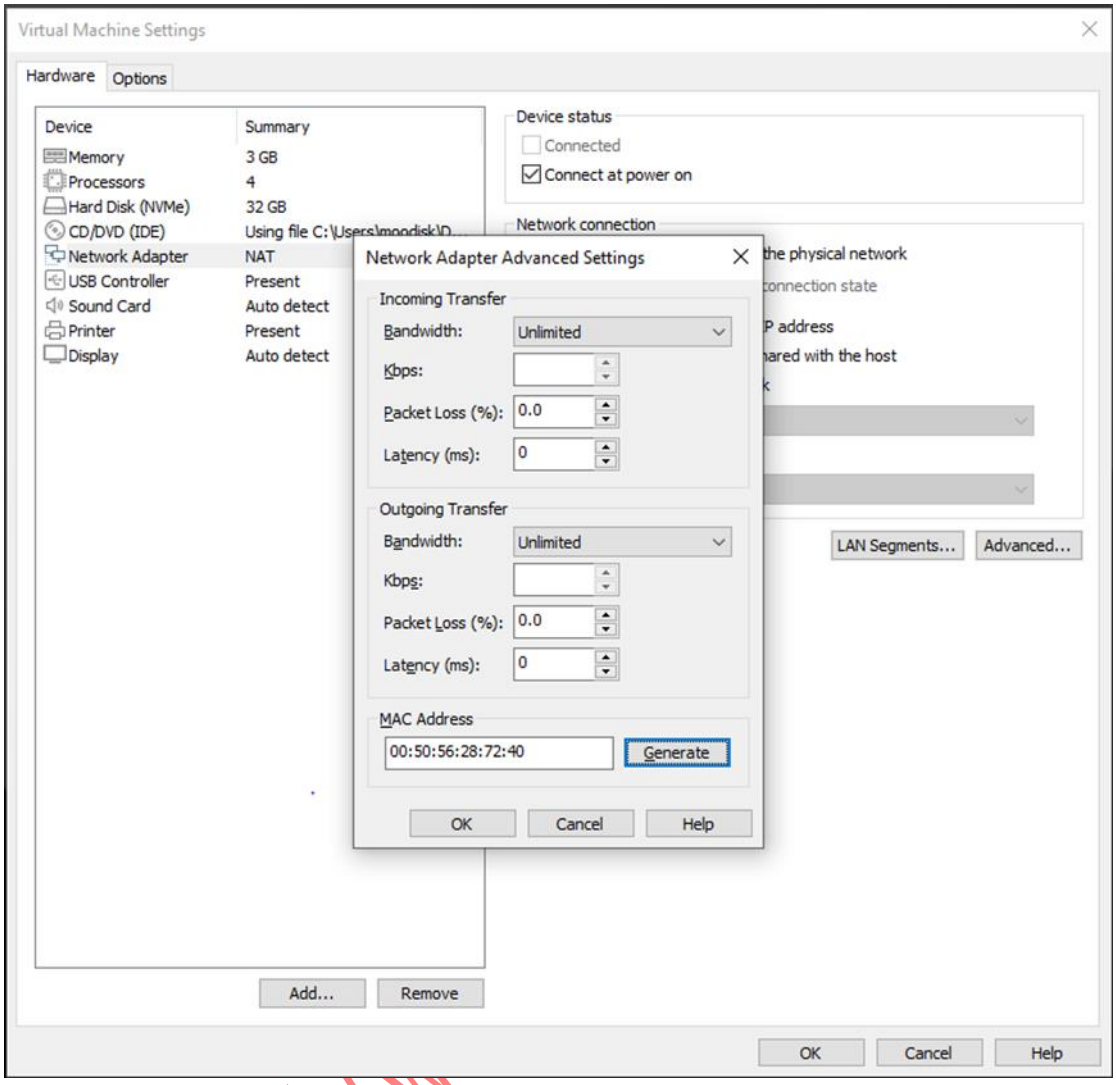


图 2-12 重新生成网卡 MAC

启动 worker1，配置主机名和网卡 ip 地址。

```
[root@master1 ~]# hostnamectl set-hostname worker1
[root@master1 ~]# su -

[root@worker1 ~]# uuidgen                                                    #产生 uuid
fc1223e7-90b4-40a6-9082-20a484d4f21b

[root@worker1 ~]# vim /etc/NetworkManager/system-connections/ens160.nmconnection
[connection]
id=ens160
uuid=fc1223e7-90b4-40a6-9082-20a484d4f21b
type=ethernet
autoconnect-priority=-999
interface-name=ens160
timestamp=1689553683

[ethernet]

[ipv4]
address1=10.10.10.31/24,10.10.10.2
dns=8.8.8.8;9.9.9.9;
method=manual
……

[root@worker1 ~]# reboot
[root@worker1 ~]# shutdown -h now
```

参照类似方法再克隆 worker2，把内存调整为 3GB，参考图 2-11 修改网卡 MAC 地址，最后启动修改主机名和网卡 ip。

```
[root@master1 ~]# hostnamectl set-hostname worker2

[root@master1 ~]# uuidgen                                                    #每次产生的 uuid 都不同
b44b3740-33b5-4420-a8c2-0d363d67a201

[root@master1 ~]# vim /etc/NetworkManager/system-connections/ens160.nmconnection
[connection]
id=ens160
uuid=b44b3740-33b5-4420-a8c2-0d363d67a201
type=ethernet
autoconnect-priority=-999
interface-name=ens160
timestamp=1689553683

[ethernet]

[ipv4]
address1=10.10.10.32/24,10.10.10.2
dns=8.8.8.8;9.9.9.9;
method=manual
……

[root@master1 ~]# reboot
[root@master1 ~]# shutdown -h now
```

再克隆 worker3，ip 地址设为 10.10.10.33。

最后看到的四台虚拟机如图 2-13 所示。

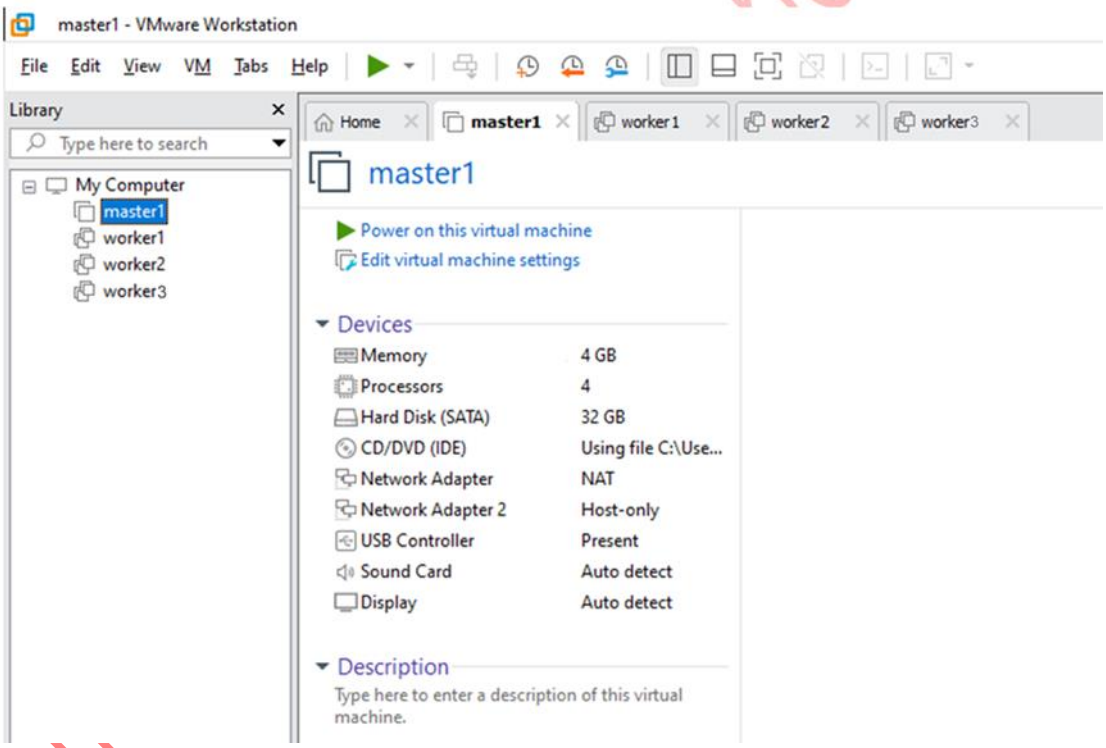


图 2-13 四台虚拟机就绪

§2.5 创建快照

四台虚拟机关闭后，我们给这些虚拟机创建快照，快照可取名 basic，描述为“第三章前”，具体操作参考图 2-14 操作。

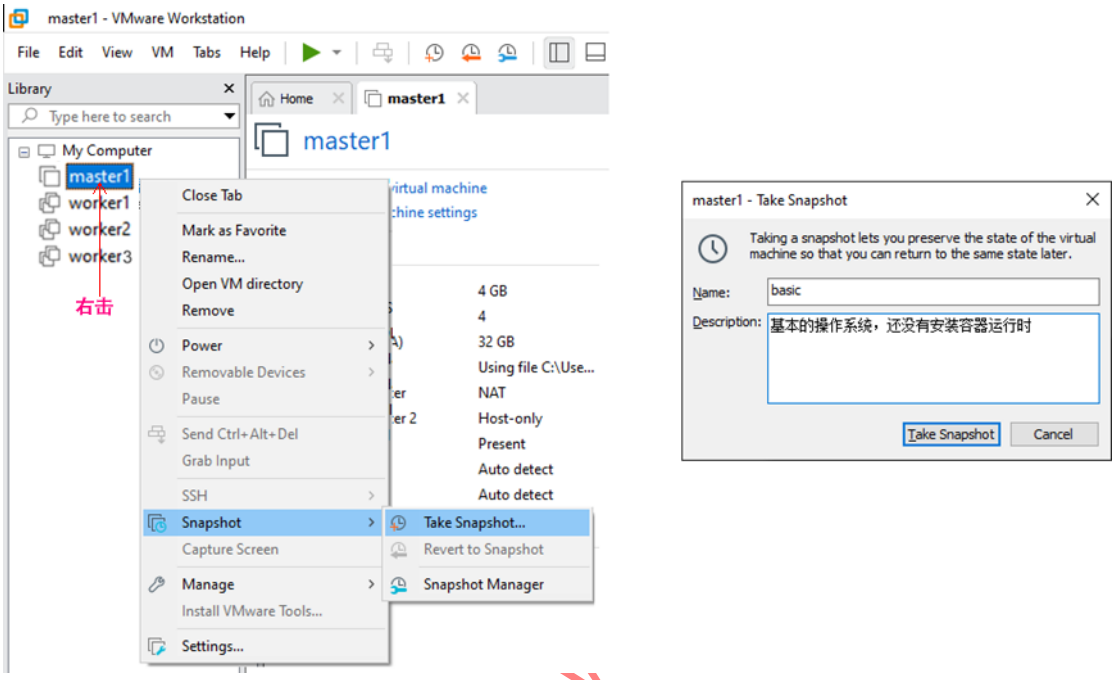


图 2-14 创建快照

有了快照后，以后可随时回滚到最初的状态，也方便以后从快照克隆更多的虚拟机。建议把 master1 虚拟机压缩备份到 U 盘上。

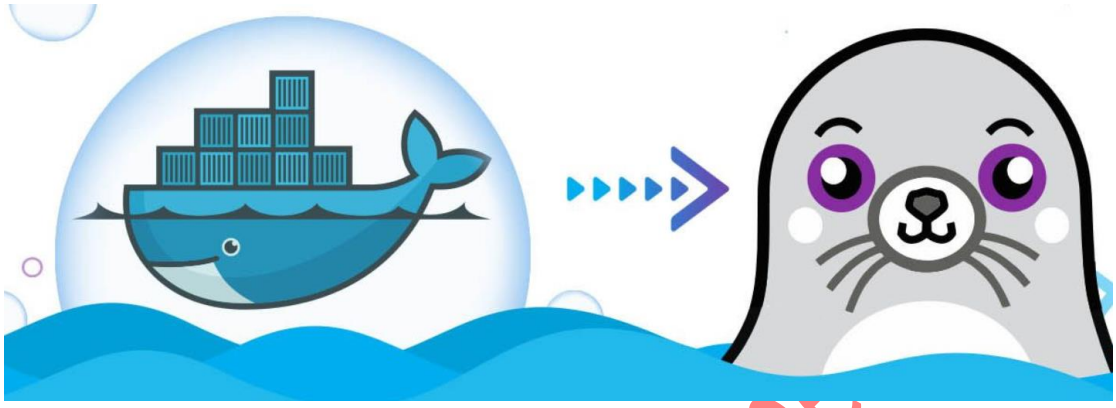
接下来的实战暂时用不到 worker1、worker2 和 worker3，所以现在只需启动 master1 就可以了。

https://www.

§2.6 拓展学习

1. SELinux/AppArmor/Seccomp 安全系统。

第3章 容器实战



由于每台虚拟机里都要安装和运行操作系统，这样就浪费了很多计算资源，为此人们推出了容器技术。举一个简单的例子：假如一台计算机的配置是双核 3GHz 的 CPU、8GB 的内存、500GB 的硬盘，现在在这台计算机上创建 6 台虚拟机，每台虚拟机分配 1GB 内存、64GB 硬盘、1GHz CPU，虚拟机都安装 Windows 7 操作系统。那么当全部虚拟机启动后，几乎就很难再运行应用程序了，因为内存和 CPU 资源都被操作系统本身消耗掉了。为此，有公司专门推出了应用软件容器产品，即在操作系统层上创建一个个容器，这些容器共享下层的操作系统内核和硬件资源，但是每个容器可单独限制 CPU、内存、硬盘和网络带宽容量，并且拥有单独的 IP 地址和操作系统管理员账户，可以关闭和重启。与虚拟机最大的不同是，容器里不用再安装操作系统，因此浪费的计算资源也就大大减少了，这样同样一台计算机就可以服务于更多的租户，示意图如图 3-1 所示。

https://www.verypo

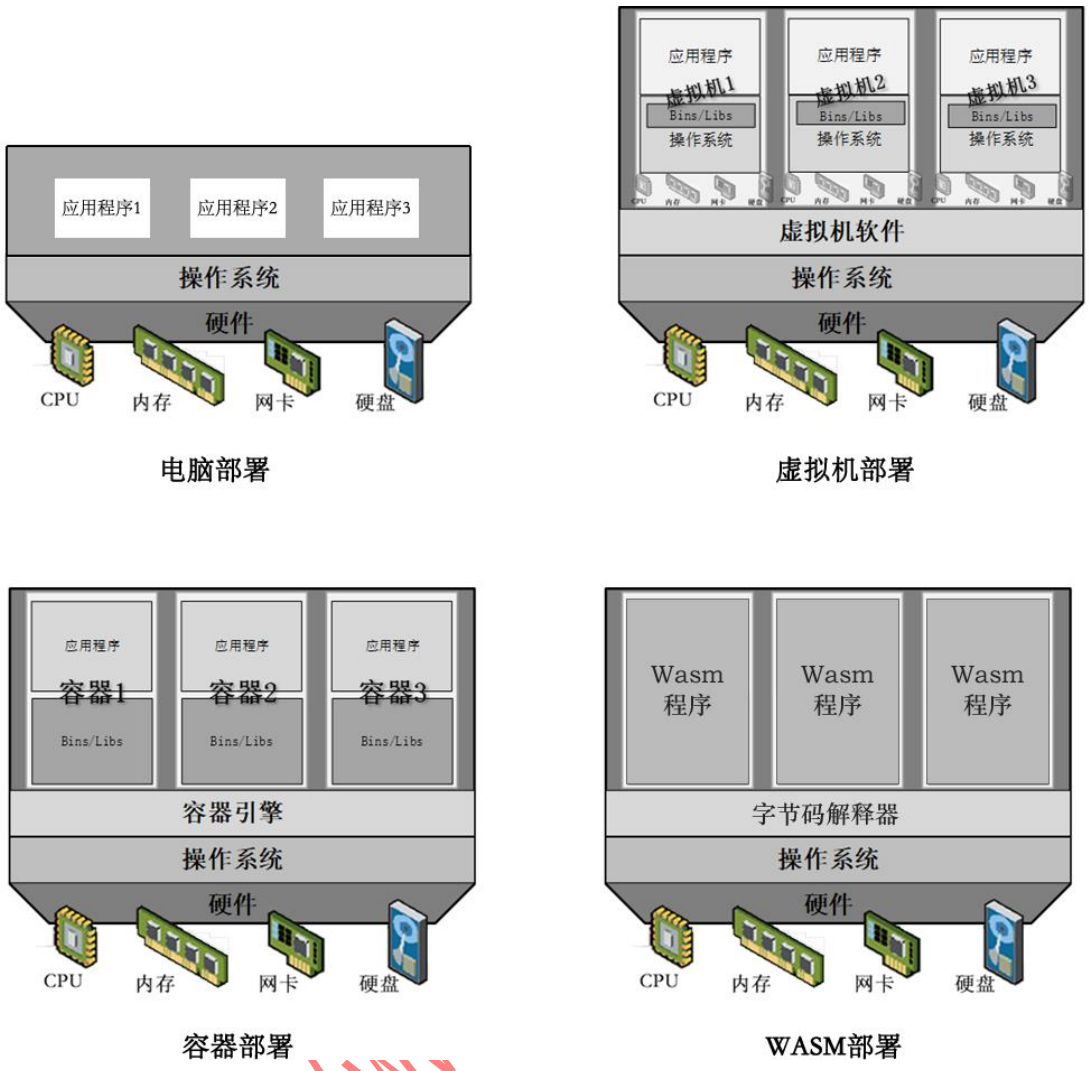


图 3-1 运行程序的四种方法

从图 3-1 还可以看出虚拟机和容器里都有“应用程序”和“Bins/Libs”，应用程序很容易理解，比如网站服务、数据库服务、ERP 等。而 Bins 就是 Binaries 的缩写，即二进制可执行程序（比如容器中的 bash、sh、ls、ip 命令等），Libs 为 Libaries 的缩写，即动态库或调用库或共享库（比如 /lib64/libc.so.6、C:\Windows\system32.dll 等），二进制可执行程序运行时需要使用动态库，关于这方面的内容请参考“第 6 章 程序运行时”。不过容器中的二进制可执行程序和动态库文件相比于虚拟机来说，要少很多，也小很多。但是容器里的程序仍然与底层操作系统紧密捆绑在一起，比如 Linux 容器就无法跑在 windows 宿主机上，另外容器镜像虽然比虚拟机小，但是成百上千兆的容量最常见，这样冷启动容器耗时也比较长（几百毫秒以上），这样对于边沿计算（如 serverless 代码）来说代价就很大。因此人们引入了 WebAssembly（简称为 Wasm）技术，它是一个与硬件、操作系统无关的沙箱化虚拟机指令集，Wasm 程序能在浏览器里能跑，也能在服务器上跑。Wasm 程序就是中间代码，类似于 Java 的字节码，真正做到了编译一次到处运行。WebAssembly 被引入到 k8s 生态，不是要取代容器，正如容器不是要取代虚拟机一样，它们各有自己的应用场景。Rust 是编写 Wasm 程序最佳语言，具体参考“§17.8 Wasm 应用”。

但是容器的隔离效果比虚拟机差，在安全性要求高的场合（比如公有云）中，虚拟机使用率要远远高于容器，而安全性要求不高的场合（比如私有云、公司内部 IT 环境）中，容器的使用率会越来越高。谷歌云目前的做法是在虚拟机里跑容器，并把 kubernetes 交给租户使用。开源产品 kata 和 gVisor 以及微软的闭源产品 Hyper-V Container 都是跨界产品——既接近于容器的轻量性，又接近于虚拟机的安全性。gVisor 是谷歌主推的开源项目，具备良好的发展势头。

一个容器本质上就是若干被隔离并且受限制运行的进程，采用名字空间实现隔离（参考第 8 章），采用控制组实现资源限制（参考第 7 章）。

一个容器原则上只运行一至两个应用，比如网站服务、邮件等，但是非常重型的应用不适合跑在容器中，比如 Oracle 数据库。使用容器的目的是方便开发、测试、发布、隔离和在集群中快速迁移应用程序，使得同一台电脑上跑更多的应用程序而互不干扰，有点类似人们常说的绿色软件。所以说容器不是虚拟机的替代品，它们各有自己应用的领域，虚拟机里可以跑容器，但是反过来就不行。

目前有很多技术可以在 Linux 操作系统上实现容器，比如 OpenVZ、Linux-VServer、LXC、Docker、FreeBSD jail、Solaris Containers 和 podman 等，其中 Docker 几年前在谷歌公司的大力推动下发展迅速，而且谷歌发布了构建于 Docker 之上的开源的 Kubernetes 管理平台，这个平台使得管理运行在成千上万台电脑上的数十万个 Docker 容器变得异常轻松和简单。而由红帽推出的 podman 是后起之秀，它不用守护进程，也可在非 root 用户下运行，podman 的用法几乎与 docker 完全一样。不过最近 k8s 已经抛弃了 docker，转而完全采用 containerd。

在 docker 中，容器就是镜像（image）的实例化，镜像与容器的关系就像面向对象编程语言中的类与实例化的关系，也类似程序和进程的关系，一个镜像允许实例化成多个容器，每个容器具有唯一的 ID 号，正如多次运行同一个程序从而产生的多个进程一样，每个进程具有唯一的进程号。镜像和容器的关系参考图 3-2。

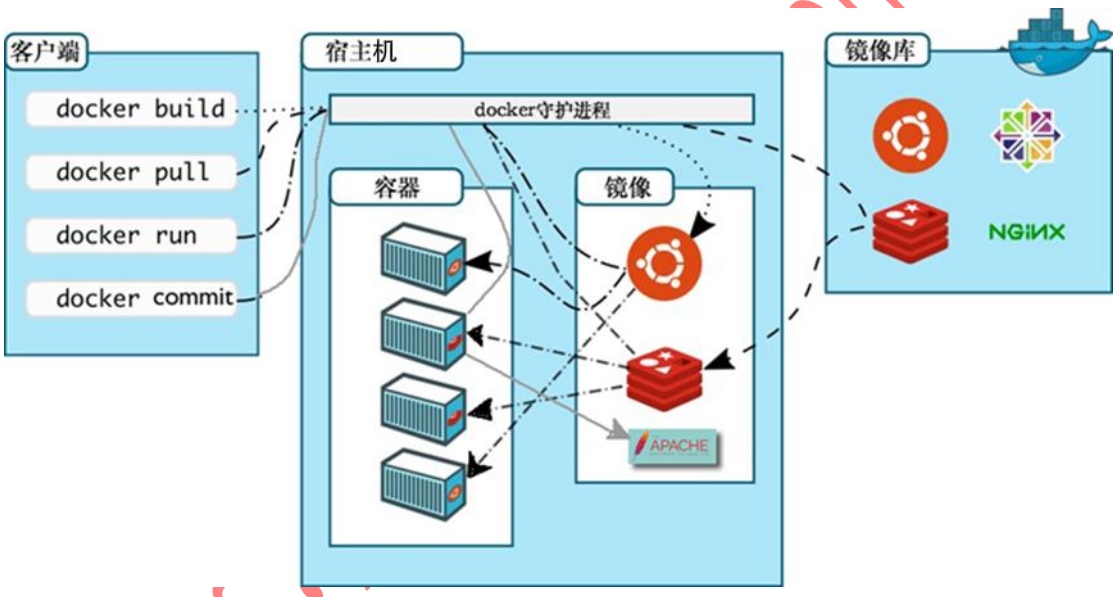


图 3-2 镜像和容器

镜像库一般位于公网上（当然也可以自建本地镜像库），用户可以把镜像下载到本地（即上图中的宿主服务器）（采用 docker pull 客户端命令），或者自己创建镜像（采用 docker build 客户端命令），或者从容器生成镜像（采用 docker commit 客户端命令）。注意图中的线型区别，比如用户执行命令 docker pull 向 docker 守护进程发指示，docker 守护进程就从网上下载镜像。

容器具有七种状态，状态转换参考图 3-3。

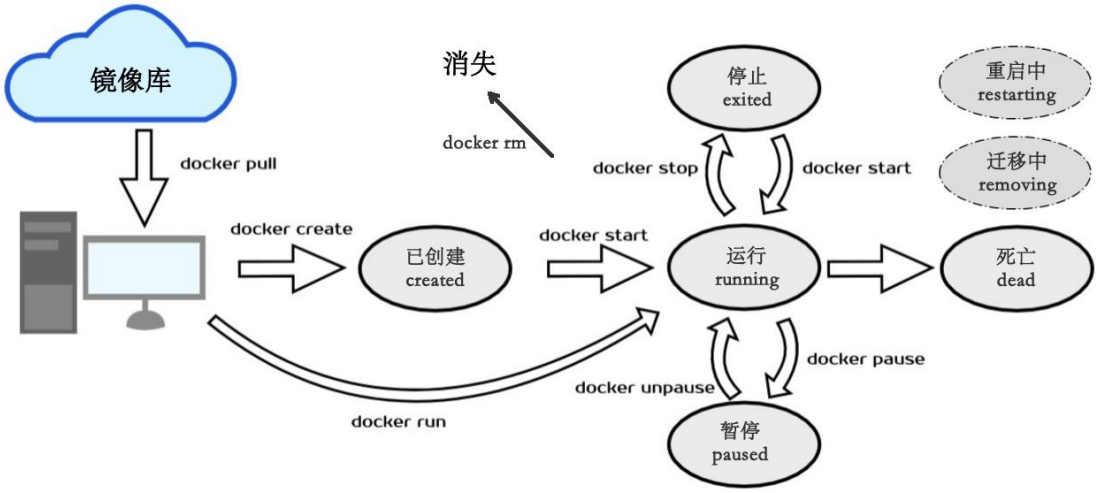
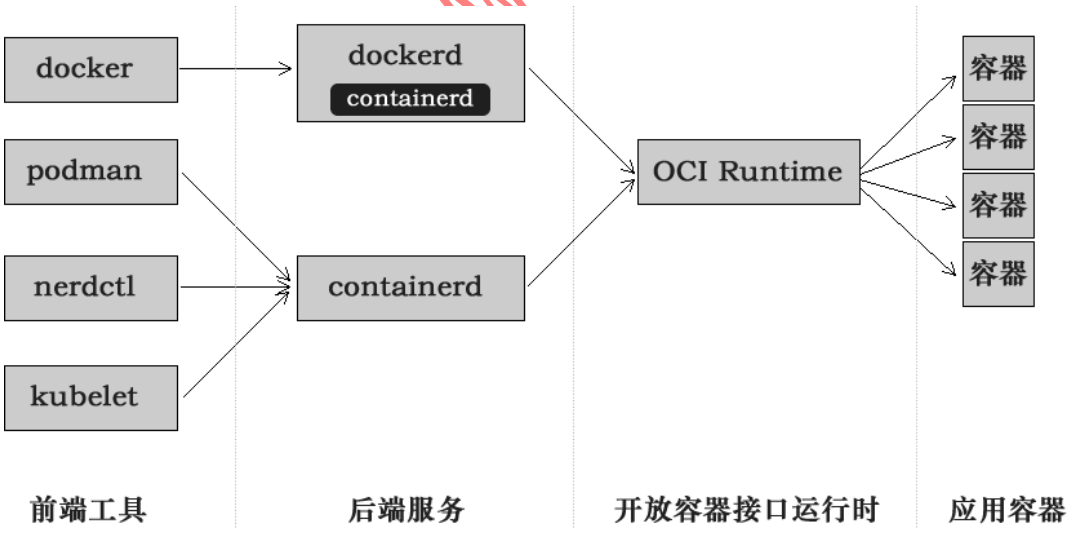


图 3-3 容器的状态

七种状态分别是已创建（created）、运行（running）、停止（exited）、暂停（paused）、死亡（dead）、重启中（restarting）和迁移中（removing），其中最后两种状态是过度状态。状态变换都是人为驱动的（运行相应的命令 docker xxx）。

Kubernetes 和 OpenStack 类似于 windows 及 Linux 操作系统，windows 和 Linux 操作系统管理进程，OpenStack 是专门管理虚拟机的，而 Kubernetes 是专门用来管理容器的。

容器环境如下图所示。



§3.1 docker 容器



§3.1.1 安装

在 Rocky Linux 9 上安装 docker 的官方文档在这里。

增加 docker 安装源配置文件 docker-ce.repo，下载后放在/etc/yum.repos.d 目录下：

```
[root@master1 ~]# dnf -y install dnf-plugins-core
[root@master1 ~]# dnf config-manager --add-repo https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
添加仓库自: https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

[root@master1 ~]# ll /etc/yum.repos.d/
总用量 24
-rw-r--r--. 1 root root  891  8 月 22 19:24 docker-ce.repo
-rw-r--r--. 1 root root 6520  8 月 23 00:16 rocky-addons.repo
-rw-r--r--. 1 root root 6610  5 月 23 23:27 rocky-addons.repo.bak
-rw-r--r--. 1 root root 1150  8 月 23 00:16 rocky-devel.repo
-rw-r--r--. 1 root root 1165  5 月 23 23:27 rocky-devel.repo.bak
-rw-r--r--. 1 root root 2357  8 月 23 00:16 rocky-extras.repo
-rw-r--r--. 1 root root 2387  5 月 23 23:27 rocky-extras.repo.bak
-rw-r--r--. 1 root root 3372  8 月 23 00:16 rocky.repo
-rw-r--r--. 1 root root 3417  5 月 23 23:27 rocky.repo.bak
-rw-r--r--. 1 root root 1410  8 月 23 00:16 rocky-security.repo
-rw-r--r--. 1 root root 1425  5 月 23 23:27 rocky-security.repo.bak
```

即/etc/yum.repos.d 目录中多了一个文件 docker-ce.repo。

~~修改成中科大的安装源：~~

```
~~[root@master1 ~]# sed -i 's|download.docker.com|mirrors.ustc.edu.cn/docker-ce|' /etc/yum.repos.d/docker-ce.repo~~
```

~~此命令就是把 /etc/yum.repos.d/docker-ce.repo 文件中的 download.docker.com 替换成 mirrors.ustc.edu.cn/docker-ce。当然替换成清华的也行 mirrors.tuna.tsinghua.edu.cn/docker-ce。~~

更新本地缓存：

```
[root@master1 ~]# dnf makecache -y
```

列出安装源的库：

```
[root@master1 ~]# dnf repolist --enabled
仓库 id                                                 仓库名称
appstream                                               Rocky Linux 9 - AppStream
baseos                                                  Rocky Linux 9 - BaseOS
docker-ce-stable                                        Docker CE Stable - x86_64
extras                                                  Rocky Linux 9 - Extras packages
```

http://veryopen.

```
resilientstorage                                Rocky Linux 9 - ResilientStorage
```

上面黄色字体的两个安装源必需被激活，如果没有激活，请使用命令它，比如激活 extras 的命令：dnf config-manager --enable extras。

不管有没有安装老版本的 docker，建议执行下面的命令卸载掉（忽略报错信息）：

```
[root@master1 ~]# dnf remove docker docker-client docker-client-latest docker-common docker-latest \
 docker-latest-logrotate docker-logrotate docker-engine
[root@master1 ~]# rm -rf /var/lib/docker/
```

老版本 docker 的镜像、容器、卷和网络保存在/var/lib/docker 目录中，而新版本的容器放在/var/lib/containerd 目录下。

安装 docker-ce（有在线安装和离线安装两种方法，这里采用在线安装方法，离线方法这里）：

```
[root@master1 ~]# dnf install docker-ce -y                                    #安装 docker-ce
```

会自动安装这些依赖软件：docker-ce-cli、containerd.io、docker-buildx-plugin、docker-compose-plugin、container-selinux 和 docker-ce-rootless-extras。

默认的官方容器镜像库在国外，国内无法下拉（被防火墙阻断），所以需要更换为国内的镜像库：

```
[root@master1 ~]# vim /etc/docker/daemon.json
{
    "registry-mirrors": [
        "https://docker.mirrors.tuna.tsinghua.edu.cn",
        "https://docker.mirrors.ustc.edu.cn",
        "https://docker.mirrors.nju.edu.cn",
        "https://docker.xuanyuan.me",
        "https://docker.m.daocloud.io"
    ],
    "insecure-registries":[
        "10.8.20.8:5000"
    ]
}
```

下拉镜像时会依次从上往下尝试直到成功或最后失败，即先从 https://docker.mirrors.tuna.tsinghua.edu.cn 下拉，如果不行在试 https://docker.mirrors.ustc.edu.cn、……、https://docker.m.daocloud.io。10.8.20.8:5000 是校内镜像库，只存放了实验需要用到的容器镜像。

```
[root@master1 ~]# systemctl restart docker
[root@master1 ~]# docker info                                        #查看生效的镜像加速库
……
Experimental: false
 Insecure Registries:
  10.8.20.8:5000
  127.0.0.0/8
 Registry Mirrors:
Registry Mirrors:
  https://docker.mirrors.tuna.tsinghua.edu.cn/
  https://docker.mirrors.ustc.edu.cn/
  https://docker.mirrors.nju.edu.cn/
  https://docker.xuanyuan.me/
  https://docker.m.daocloud.io/
Live Restore Enabled: false
……
```

镜像加速库对搜索（docker search…）无效，搜索会继续走官方库。可以浏览 https://xuanyuan.cloud/页面并搜索需要的镜像，再来下拉。对于一些大型的镜像，建议直接从校园内拉取，例如 mariadb:latest 镜像有 400MB+，从外网拉太慢，这时可从内网下载：docker pull 10.8.20.8:5000/library/mariadb:latest，速度会快很多。注意：从内网拉镜像时一定要知名具体的版本，即使 latest 也不能省略，否则需要多次拉取才可能成功一次。

```
[root@master1 ~]# systemctl enable --now docker                          #激活并启动 docker 服务
[root@master1 ~]# ip a show docker0                                      #查看网桥 docker0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:ef:dc:8b:f2 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
```

网桥 docker0 上配置 172.17.8.1，这是容器的网关。还会新建防火墙区 docker，docker0 加入了此区，但没加入默认区 public：

```
[root@master1 ~]# firewall-cmd --zone=docker --list-all
docker (active)
  target: ACCEPT
  icmp-block-inversion: no
  interfaces: docker0
  ……

[root@master1 ~]# firewall-cmd --list-all
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: ens160
  ……
```

```
[root@master1 ~]# docker version                                    #显示 docker 的版本信息
Client: Docker Engine - Community
 Version:           29.7.2
 API version:       1.55
 Go version:        go1.26.5
 Git commit:        a7dcaa6
 Built:             Wed Aug  5 18:32:45 2026
 OS/Arch:           linux/amd64
 Context:           default

Server: Docker Engine - Community
 Engine:
  Version:          29.7.2
  API version:      1.55 (minimum version 1.40)
  Go version:       go1.26.5
  Git commit:       6a43e3d
  Built:            Wed Aug  5 18:28:25 2026
  OS/Arch:          linux/amd64
  Experimental:     false                                           #不是实验版，是正式版本
containerd:
  Version:          v2.3.3
  GitCommit:        aad11006b869517fcd3009450b6f82da282e1a9b
 runc:
  Version:          1.4.3
  GitCommit:        v1.4.3-0-gbb14dabe
 docker-init:
  Version:          0.19.0
  GitCommit:        de40ad0
```

运行一个 hello world 的容器试试安装是否成功：

```
[root@master1 ~]# docker run hello-world
Unable to find image hello-world:latest' locally
latest: Pulling from hello-world
c1ec31eb5944: Pull complete
Digest: sha256:d37ada95d47ad12224c205a938129df7a3e52345828b4fa27b03a98825d1e2e7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

如果长时间没反应，最后报错：docker: Error response from daemon: Get "https://registry-1.docker.io/v2/": net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers).，说明没法访问 docker 官方镜像库，有几种方法解决此问题，但是国内的很多镜像源不能访问了，最好的方法是自己搭建私有镜像源，学校的镜像库的 ip 和端口为 10.8.20.8:5000，然后运行容器：docker run 10.8.20.8:5000/hello-world:latest。

可采用这些命令卸载 docker：dnf remove docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin docker-ce-rootless-extras 、 rm -rf /var/lib/{docker,

containerd}。下面还要继续实战容器，所以先不卸载。

§3.1.2 体验一遍

从镜像库中下拉容器镜像，然后启动容器。

1. 在镜像库中搜索包含 debian 关键字的镜像：

```
[root@master1 ~]# docker search debian
NAME                                      DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
ubuntu                                    Ubuntu is a Debian-based Linux operating sys…   16416     [OK]
debian                                    Debian is a Linux distribution that's compos…   4797      [OK]
neurodebian                               NeuroDebian provides neuroscience research s…   104       [OK]
bitnami/debian-base-buildpack             Debian base compilation image                   2                    [OK]
kasmweb/debian-bullseye-desktop           Debian Bullseye desktop for Kasm Workspaces     5
mirantis/debian-build-ubuntu-xenial                                                       0
rancher/debianconsole                                                                     1
datadog/debian-i386                                                                       0
mirantis/debian-build-ubuntu-trusty                                                       0
kasmweb/debian-bookworm-desktop                                                           0
osrf/debian_arm64                         Debian arm64 Base Images                        1
……
```

如果超时报错，请检查 DNS 是不是 8.8.8.8 和 9.9.9.9，另一个原因可能是没法访问官方镜像库。我们在 daemon.json 文件中配置的镜像地址，只会在 docker Pull 时被使用，而 docker search 时使用的仍然是 Docker 默认搜索地址，在国内就会因超时而失败。这是要么搜索时指定源，例如“docker search docker.1ms.run/debian”，要么干脆去国内网站（如 https://xuanyuan.cloud/）搜索。列出本地镜像库 10.8.20.8:5000 里的镜像：

```
[root@master1 ~]# curl -XGET http://10.8.20.8:5000/v2/_catalog | jq
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   174  100   174    0     0  43500      0 --:--:-- --:--:-- --:--:-- 58000
{
  "repositories": [
    "library/busybox",
    "library/debian",
    "library/grafana/grafana",
    "library/hello-world",
    "library/mariadb",
    "library/nginx",
    "library/wordpress",
    "wasm/hello-wasm"
  ]
}
```

2. 下拉 debian 镜像：

```
[root@master1 ~]# docker pull 10.8.20.8:5000/library/debian:latest
Using default tag: latest
latest: Pulling from library/debian
785ef8b9b236: Pull complete
Digest: sha256:9f76a008888da28c6490bedf7bdaa919bac9b2be827afd58d6eb1b916e1e5918
Status: Downloaded newer image for debian:latest
docker.io/library/debian:latest

[root@master1 ~]# docker images                                                    #查看本地的镜像
IMAGE                                     ID             DISK USAGE   CONTENT SIZE   EXTRA
10.8.20.8:5000/library/debian:latest      d8f17b92dc7f       181MB         49.3MB
hello-world:latest                        5e2309035332      21.8kB         9.49kB    U

[root@master1 ~]# docker tag 10.8.20.8:5000/library/debian:latest debian:latest                  #打标
```

如果要拉取特定的版本，请指明版本号，比如下来 12 版：docker pull debian:12。如果从互联网下拉，就直接 docker pull debian:latest 即可，也可省略:latest，默认就是拉取最新版。

3. 列出本地的镜像：

```
[root@master1 ~]# docker images
IMAGE                                     ID             DISK USAGE   CONTENT SIZE   EXTRA
```

```
10.8.20.8:5000/library/debian:latest   d8f17b92dc7f        181MB         49.3MB
debian:latest                          d8f17b92dc7f        181MB         49.3MB
hello-world:latest                     5e2309035332       21.8kB         9.49kB    U
```

4．查看镜像的详细说明：

```
[root@master1 ~]# docker inspect debian
[
    {
        "Id": "sha256:999ffdddc1528999603ade1613e0d336874d34448a74db8f981c6fae4db91ad7",
        "RepoTags": [
            "debian:latest"
        ],
        "RepoDigests": [
            "debian@sha256:833c135acfe9521d7a0035a296076f98c182c542a2b6b5a0fd7063d355d696be"
        ],
        "Parent": "",
        "Comment": "debuerreotype 0.16",
        "Created": "2025-09-08T00:00:00Z",
        "DockerVersion": "",
        "Author": "",
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 119847939,
        "GraphDriver": {
            "Data": {
                "MergedDir": "/var/lib/docker/overlay2/0955ff537b61b2ab2a509a2e15a84ea52ec2f5d715ed8b7a303b378c838b303b/merged",
                "UpperDir": "/var/lib/docker/overlay2/0955ff537b61b2ab2a509a2e15a84ea52ec2f5d715ed8b7a303b378c838b303b/diff",
                "WorkDir": "/var/lib/docker/overlay2/0955ff537b61b2ab2a509a2e15a84ea52ec2f5d715ed8b7a303b378c838b303b/work"
            },
            "Name": "overlay2"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:185e04da9d947141fd703dbf36361bdc2ff77cc27cbf500fb9f4881cb5ddbe95"
            ]
        },
        "Metadata": {
            "LastTagTime": "0001-01-01T00:00:00Z"
        },
        "Config": {
            "Cmd": [
                "bash"
            ],
            "Entrypoint": null,
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
            ],
            "Labels": null,
            "OnBuild": null,
            "User": "",
            "Volumes": null,
            "WorkingDir": ""
        }
    }
]
```

其中黄色字体部分说明了启动容器后默认运行的程序就是 bash（即/bin/bash），所以命令 docker run -it debian 与命令 docker run -it debian bash 以及命令 docker run -it debian /bin/bash 都是等价的。

5．启动容器并运行容器中的/bin/bash（进入容器），然后退出容器：

```
[root@master1 ~]# docker run -it debian /bin/bash
root@0e0df8387363:/# uname -a
Linux 0e0df8387363 5.14.0-333.el9.x86_64 #1 SMP PREEMPT_DYNAMIC Wed Jun 28 09:47:27 UTC 2023 x86_64 GNU/Linux
root@0e0df8387363:/# exit                                                    #退出并关闭容器
```

只要运行容器时指定的程序结束了，那么容器就被关闭。上面的 exit 表示退出/bin/bash 程序，所以容器也被关闭。

当然可以直接运行容器，而不用事先下拉容器镜像。

6．列出处于任何状态的容器：

```
[root@master1 ~]# docker ps -a
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS                     PORTS     NAMES
0e0df8387363   debian    "/bin/bash"   5 minutes ago   Exited (0) 2 minutes ago             adoring_curran
```

7．再次启动容器 0e0df8387363：

```
[root@master1 ~]# docker start 0e0df8387363
0e0df8387363

[root@master1 ~]# docker ps -a
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS          PORTS     NAMES
0e0df8387363   debian    "/bin/bash"   9 minutes ago   Up 46 seconds             adoring_curran
```

8．进入处于运行状态的容器：

```
[root@master1 ~]# docker attach 0e0df8387363
root@0e0df8387363:/# ls /
```

```
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr
var
root@0e0df8387363:/# cat /etc/issue
Debian GNU/Linux 12 \n \l

root@c02f2354b88c:/#                                                    #按 Ctrl+p，Ctrl+q 退出容器，但容器继续运行

[root@master1 ~]# docker ps -a
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS          PORTS     NAMES
0e0df8387363   debian    "/bin/bash"   14 minutes ago   Up 5 minutes              adoring_curran
```

一旦执行 exit 命令退出，容器就被停止。如果退出后让容器继续运行，请按键 Ctrl+p、Ctrl+q 离开容器。

9. 从同一个镜像再启动一个容器：

```
[root@master1 ~]# docker run debian /usr/bin/sleep 20m&                    #容器中运行 sleep 20m（睡眠 20 分钟）
[1] 3093

[root@master1 ~]# docker ps                                                                    #查看正在运行的容器
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS     NAMES
fe0065b3521f   debian    "/usr/bin/sleep 20m"     52 seconds ago   Up 51 seconds             crazy_black
0e0df8387363   debian    "/bin/bash"              28 minutes ago   Up 19 minutes             adoring_curran
```

采用命令“docker exec -it fe0065b3521f /bin/bash”可再次进入容器，如果一个容器本身就在执行 /bin/bash 命令（比如上面第二行所示的容器），可采用命令“docker attach 0e0df8387363”再次进入容器。

10. 查看运行中的容器说明信息：

```
[root@master1 ~]# docker inspect fe0065b3521f
……
    "Status": "running",
……
    "Id": "0e0df8387363bf96184839b6f6d8d069d96bb31b93ea999b5bb45878adc43f19",
……
    "Image": "sha256:278727474c2935fca471835f8e09a78631bf93d6a414979f038abebc315fb8da",
    "ResolvConfPath":
"/var/lib/docker/containers/0e0df8387363bf96184839b6f6d8d069d96bb31b93ea999b5bb45878adc43f19/resolv.conf",
    "HostnamePath": "/var/lib/docker/containers/0e0df8387363bf96184839b6f6d8d069d96bb31b93ea999b5bb45878adc43f19/hostname",
    "HostsPath": "/var/lib/docker/containers/0e0df8387363bf96184839b6f6d8d069d96bb31b93ea999b5bb45878adc43f19/hosts",
……
    "LowerDir": "/var/lib/docker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69-
init/diff:/var/lib/docker/overlay2/08621f4e7f3586e162d1363547935c427e0c47e213ba382a25977a5abfbbb445/diff",
    "MergedDir": "/var/lib/docker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/merged",
    "UpperDir": "/var/lib/docker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/diff",
    "WorkDir": "/var/lib/docker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/work"
……
    "Cmd": [
            "/usr/bin/sleep",
            "20m"
    ],
    "Image": "debian",
……
    "IPAddress": "172.17.0.2",
    "IPPrefixLen": 16,
    "IPv6Gateway": "",
    "MacAddress": "02:42:ac:11:00:03",
……

[root@master1 ~]# df -T
Filesystem     Type     1K-blocks    Used Available Use% Mounted on
devtmpfs       devtmpfs      4096       0      4096   0% /dev
tmpfs          tmpfs      1990304       0   1990304   0% /dev/shm
tmpfs          tmpfs       796124    9380    786744   2% /run
/dev/sda1      xfs       33487872 3079780  30408092  10% /
tmpfs          tmpfs       398060       0    398060   0% /run/user/0
overlay        overlay   33487872 3079780  30408092  10% /var/lib/docker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/merged
overlay        overlay   33487872 3079780  30408092  10% /var/lib/docker/overlay2/f12f0b5108d431d3e4a8da05a12a979eff1a85befe8a9639b1811546bdd79a57/merged

[root@master1 ~]# mount | grep overlay
overlay on /var/lib/docker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/merged type overlay
(rw,relatime,lowerdir=/var/lib/docker/overlay2/l/AFUVKIFZSHQT36Y4N5IEZ3HS76:/var/lib/docker/overlay2/l/LZWENGH7KIUIDHRFLY4EYFU3DS,upperdir
=/var/lib/docker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/diff,workdir=/var/lib/docker/overlay2/30a7f3868
10d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/work)
```

docker 容器会用到 overlay 文件系统，它是一个联合文件系统——把多个宿主机上的目录叠加合并后挂载到一个空目录上。在联合文件系统中，底层目录是只读的，只有最上层目录是可写的。参考图 3-4 作进一步理解。

```
mount -t overlay overlay -o lowerdir=./lower2:./lower1,upperdir=./upper,workdir=./work ./merged
```

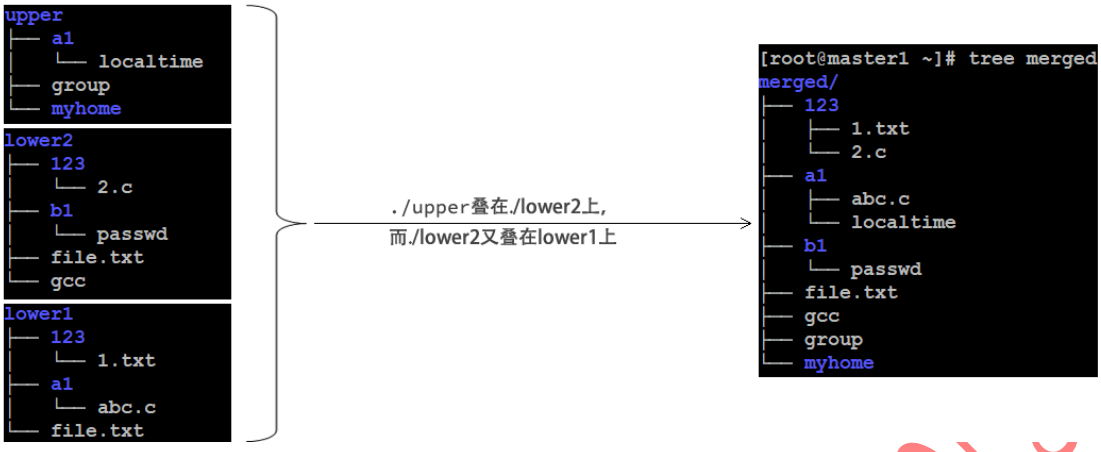


图 3-4 overlay 文件系统示例

左侧是将被叠加的三个目录的结构，右侧是联合文件系统的目录结构。从图可以发现：合并同名目录中的文件（参考图 3-4 中的 a1 和 123 目录），上层文件覆盖下层的同名文件（例如图 3-4 中的 file.txt 文件）。以后往 merged 目录写东西就是往 upper 目录写东西。

下面我们通过实战验证上面的例子：

```
[root@master1 ~]# mkdir -p upper/{a1,myhome} lower2/{123,b1} lower1/{123,a1} worker merged                #创建需要的目录
[root@master1 ~]# tree lower1 lower2 upper merged worker                                                 #显示目录结构
lower1
├── 123
└── a1
lower2
├── 123
└── b1
upper
├── a1
└── myhome
merged
worker

[root@master1 ~]# cd upper/
[root@master1 upper]# cp /etc/group ./ ------------------------------------------------------------------------------#拷贝文件
[root@master1 upper]# touch a1/localtime                                                                #新建空文件 localtime
[root@master1 upper]# cd ../lower2
[root@master1 lower2]# echo ggcgccggcc > gcc ------------------------------------------#创建文件 gcc，文件内容为 ggcgccggcc
[root@master1 lower2]# echo 111111 > file.txt
[root@master1 lower2]# cp /etc/passwd b1/
[root@master1 lower2]# ls > 123/2.c
[root@master1 lower2]# cd ../lower1
[root@master1 lower1]# echo 22222 > file.txt
[root@master1 lower1]# ls / > 123/1.txt
[root@master1 lower1]# echo include > a1/abc.c
[root@master1 lower1]# cd ../
[root@master1 ~]# tree upper lower2 lower1
upper
|-- a1
|   `-- localtime
|-- group
`-- myhome
lower2
|-- 123
|   `-- 2.c
|-- b1
|   `-- passwd
|-- file.txt
`-- gcc
lower1
|-- 123
|   `-- 1.txt
|-- a1
|   `-- abc.c
`-- file.txt
```

准备好了三个目录后，下面创建联合文件系统：

```
[root@master1 ~]# mount -t overlay overlay -o lowerdir=./lower2:./lower1,upperdir=./upper,workdir=./worker ./merged
```

查看联合文件系统的目录结构：

```
[root@master1 ~]# tree ./merged
./merged
|-- 123
|   |-- 1.txt
|   `-- 2.c
|-- a1
|   |-- abc.c
|   `-- localtime
|-- b1
|   `-- passwd
|-- file.txt
|-- gcc
|-- group
`-- myhome
```

执行命令“mount -t overlay …”前要确保 Linux 内核模块 overlay 已经加载，采用命令“lsmod | grep overlay”查看（参考§2.3 小节的中的“9. 加载两个额外的 Linux 内核模块”）。

根据前面容器的说明信息和其他命令（如 df -T、mount | grep overlay）的输出，我们可以还原启动容器时挂载联合文件系统的命令如下：

```
mount -t overlay overlay -o \
lowerdir=/var/lib/docker/overlay2/l/AFUVKIFZSHQT36Y4N5IEZ3HS76:/var/lib/docker/overlay2/l/LZWEN6H7KIUIDHRFLY4EYFU3DS,upperdir=/var/lib/doc
ker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/diff,workdir=/var/lib/docker/overlay2/30a7f386810d5da5297e19
1f83aaa1175ff53228878cb4a292735e66eee8da69/work \
/var/lib/docker/overlay2/30a7f386810d5da5297e191f83aaa1175ff53228878cb4a292735e66eee8da69/merged
```

课堂实战练习：运行一个 busybox 容器。提示：在线上镜像库中搜索 busybox，然后下拉容器镜像，最后运行（运行时不用指定执行的命令）。

§3.1.3 定制镜像

接下来我们定制容器镜像（修改镜像），在容器中运行网站服务。

1. 首先在容器中安装 ngnix 网站服务器：

```
[root@master1 ~]# docker run -i -t debian /bin/bash                                    #启动并进入 debian 容器

配置网易的安装源：
root@0e0df8387363:/# sed -i 's|deb.debian.org|mirror.nju.edu.cn|g' /etc/apt/sources.list.d/debian.sources       #使用南京大学源

root@0e0df8387363:/# apt update
```

上面第二条命令就是把/etc/apt/sources.list.d/debian.sources 文件中的“deb.debian.org”替换成“mirror.nju.edu.cn”（即南京大学的源），也可以替换成清华大学的源（mirrors.tuna.tsinghua.edu.cn）。如果这些源都不能用，很可能就是容器的网络不通，请参考“§23.1 docker 容器网络不通”小节解决。对于容器镜像，从 Debian 12 开始，其软件源配置文件变更为 DEB822 格式，路径为/etc/apt/sources.list.d/debian.sources。一般情况下，将对应文件中 Debian 默认的源地址 http://deb.debian.org/ 替换为镜像地址即可。

安装 web 服务 nginx 并激活它：

```
root@0e0df8387363:/# apt install nginx -y                                              #安装 nginx

root@0e0df8387363:/# ip a | grep inet                                                  #查看网卡的 ip
```

https://www.veryopen.org

```
    inet 127.0.0.1/8 scope host lo
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0

root@0e0df8387363:/# update-rc.d nginx enable                                    #开机时 nginx 自动启动
root@0e0df8387363:/# exit
```

退出（exit）容器后，容器就停止了。

2．从容器创建镜像：

```
[root@master1 ~]# docker ps -a                                                                                  #存在哪些容器
CONTAINER ID   IMAGE     COMMAND                  CREATED       STATUS                          PORTS     NAMES
0e0df8387363   debian    "/bin/bash"              3 hours ago   Exited (0) About a minute ago             adoring_curran
2016e14bc530   debian    "/usr/bin/sleep 20m"     2 hours ago   Exited (137) 2 hours ago                  relaxed_merkle

[root@master1 ~]# docker commit -m="Nginx Web Server" -a="李木生" 0e0df8387363 debian:www                       #创建容器镜像
sha256:ced4634b22cce5fe2af7056093d5df965339244ecefb0013dee614becf612ebd

[root@master1 ~]# docker image list                                                                             #列出本地存在的镜像
IMAGE                          ID             DISK USAGE   CONTENT SIZE   EXTRA
hello-world:latest             5e2309035332       21.8kB         9.49kB    U
busybox:latest                 dc2d74b28e4c       6.74MB         2.23MB    U
debian:latest                  d8f17b92dc7f        181MB          49.3MB    U
debian:www                     a1723137ddc4        237MB          70.9MB
```

从容器生成镜像（上面的 docker commit … 命令），并备注为“Nginx Web Server”，指名作者为“李木生”，新产生的镜像名为 debian:www。注意：容器 id 不要搞错了，一般是最前面那个。

现在可以启动定制的镜像来对外提供 web 服务了。由上面最后一条命令可知，我们定制的容器镜像为 debian:www，id 号为 ced4634b22cc。

3．启动容器里的 nginx 服务：

```
[root@master1 ~]# docker run debian:www /usr/bin/whereis nginx                                  #查看 nginx 程序的全路径
nginx: /usr/sbin/nginx /usr/lib/nginx /etc/nginx /usr/share/nginx /usr/share/man/man8/nginx.8.gz

[root@master1 ~]# docker run -t -d -p 8081:80 debian:www /usr/sbin/nginx -g "daemon off;"       #启动容器中的 Nginx 服务
d3b245507122543ada18ebca256816c85119e17b2f32ed20cb0176e2af60a393
```

这时我们可以在 windows 的浏览器中打开网站 http://10.10.10.11:8081，看到容器中的默认静态页面，参考图 3-5。



图 3-5 容器中 web 服务器的默认页面

```
[root@master1 ~]# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                                       NAMES
d3b245507122   debian:www     "/usr/sbin/nginx -g …"   53 seconds ago   Up 52 seconds   0.0.0.0:8081->80/tcp, :::8081->80/tcp       boring_bhaskara
```

```
产生一个简单的页面文件：
[root@master1 ~]# docker exec d3b245507122 /bin/bash -c 'echo "<h1>Hello from Nginx in Docker Container</h1>" > /var/www/html/index.html'
```

上面这条命令表示在容器 d3b245507122 中执行“/bin/bash -c 'echo "<h1>Hello from Nginx in Docker Container</h1>" > /var/www/html/index.html'”，而/bin/bash -c '…'表示用/bin/bash 程序来执行单引号内的命令，而 echo "<h1>Hello from Nginx in Docker Container</h1>" > /var/www/html/index.html'就是产生新文件/var/www/html/index.html，内容只有一行，即<h1>Hello from Nginx in Docker Container</h1>。/var/www/html/index.html 就是容器的 web 服务器默认的主页面文件。

4．访问 Web 页面：

```
[root@master1 ~]# curl localhost:8081
#浏览页面
<h1>Hello from Nginx on Docker Container</h1>

[root@master1 ~]# docker inspect d3b245507122 |grep \"IPAddress                                    #查看容器的 ip
            "IPAddress": "172.17.0.2",
                    "IPAddress": "172.17.0.2",

[root@master1 ~]# curl 172.17.0.2
<h1>Hello from Nginx on Docker Container</h1>
```

启动容器时，我们做了端口映射，即宿主机上的 8081 端口映射到容器中的 80 端口，因此可以用两种方式访问容器中的 web 服务：⑴ http://<宿主机 ip>:8081；或者⑵ http://<容器的 ip>。

例如：直接在浏览器中打开 http://<宿主机的 ip>:8081，参考图 3-6 所示。

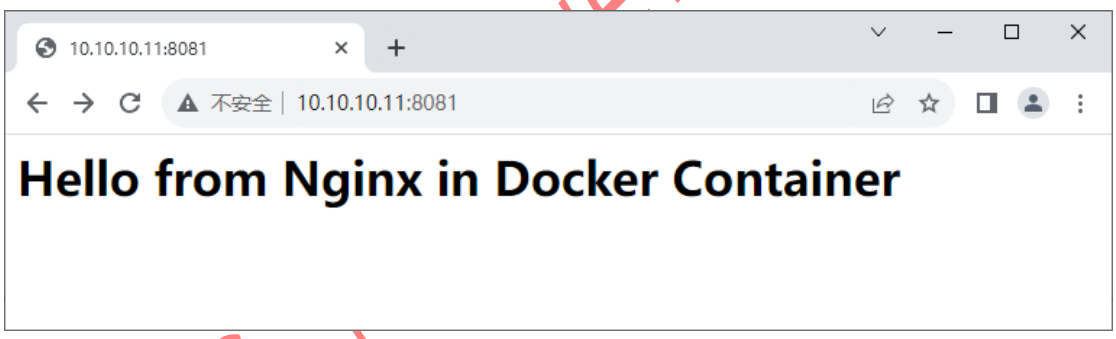


图 3-6 浏览页面

5．发布镜像

备份或发布容器镜像：

```
[root@master1 ~]# docker images
IMAGE                          ID             DISK USAGE   CONTENT SIZE   EXTRA
hello-world:latest             5e2309035332       21.8kB         9.49kB    U
busybox:latest                 dc2d74b28e4c       6.74MB         2.23MB    U
debian:latest                  d8f17b92dc7f        181MB          49.3MB    U
debian:www                     a1723137ddc4        237MB          70.9MB

[root@master1 ~]# docker image save -o ./NginxDocker.tar debian:www                    #备份
[root@master1 ~]# xz NginxDocker.tar                                                   #压缩
[root@master1 ~]# ls -l NginxDocker.tar.xz
-rw------- 1 root root 62176252 Jul 31 11:21 NginxDocker.tar.xz
```

这样就可以把文件 NginxDocker.tar.xz 拷贝出来，或者分发给别人了。不过要注意的是：备份时最好不要用镜像 id，而直接指明镜像名称和标记，这样就把名称和标记都备份出来了。其他人得到这个文件后就可以恢复成容器镜像，参考下面的步骤。

6．恢复成容器镜像：

```
[root@localhost ~]# xz -d NginxDocker.tar.xz                                    #解压
[root@localhost ~]# docker load -i NginxDocker.tar                              #恢复
Loaded image ID: sha256:bc32c2703bac34fbf9ac9d5d0c209a7d3e53344b5587a58914140f2b0d0f3bda

[root@localhost ~]# docker images -a
IMAGE                                ID             DISK USAGE   CONTENT SIZE   EXTRA
debian:www                           a1723137ddc4       237MB         70.9MB
```

有了容器镜像之后就可以启动容器了：

```
[root@localhost ~]# docker run -t -d -p 8082:80 debian:www /usr/sbin/nginx -g "daemon off;"
```

可以采用命令“docker rm 容器id”删除容器，采用命令“docker rmi 镜像id”删除镜像。

课堂实战练习：在debian镜像的基础上定制一个跑mariadb数据库服务的容器镜像。

§3.1.4 构建镜像

本小节，我们手工来构建一个容器镜像，之前我们用别人构建好了容器镜像，或者在已有容器镜像的基础上修改。下面以rockylinux-9为蓝本，构建一个运行web服务的镜像。

1．新建一个文件Dockerfile：

```
[root@master1 ~]# vim Dockerfile
#FROM dokken/rockylinux-9
FROM 10.8.20.8:5000/rockylinux-9

RUN dnf makecache
RUN dnf -y install httpd
RUN echo "<h1>Dockerfile Test on Aapche httpd 哈哈</h1>" > /var/www/html/index.html

EXPOSE 80
CMD ["-D", "FOREGROUND"]
ENTRYPOINT ["/usr/sbin/httpd"]
```

文件中是一系列指令：FROM指令定义基础镜像；RUN指令定义在容器中执行的命令；EXPOSE指令设置web服务器监听80端口；CMD和ENTRYPOINT指令定义以后启动容器时运行的程序，即“/usr/sbin/httpd -D FORCEGROUN”。

上面这个文件的意思是：以rockylinux-9发行版为蓝本(FROM指示器)，在镜像中安装httpd网站服务器(RUN指示器)，并创建了一个简单的页面（RUN指示器），以标题1显示“Dockerfile Test on Aapche httpd哈哈”。从镜像启动容器时运行程序/usr/sbin/httpd -D FORCEGROUND，此程序监听80端口。

如果采用公网镜像源，则FROM指令为dokken/rockylinux-9。

2．开始构建镜像。采用命令docker build -t [image name]:[tag]：

```
[root@master1 ~]# docker build -t rocky-httpd:latest ./
[+] Building 88.1s (6/7)                                                docker:default
```

http://www.veryopen.org

```
[+] Building 88.3s (6/7)                                                    docker:default
[+] Building 88.4s (6/7)                                                    docker:default
[+] Building 88.6s (7/8)                                                    docker:default
[+] Building 88.7s (7/8)                                                    docker:default
[+] Building 88.8s (8/8) FINISHED                                           docker:default
=> [internal] load build definition from Dockerfile
0.0s3  => => transferring dockerfile: 293B
0.0s3  => [internal] load metadata for 10.8.20.8:5000/rockylinux-9:latest
0.0s3  => [internal] load .dockerignore
0.0s3  => => transferring context: 2B
0.0s3  => [1/4] FROM 10.8.20.8:5000/rockylinux-9:latest@sha256:d644d203142cd5b54ad2a83a203e1dee68af2229f8fe32f52a30c6e1d3c3a9e0
4.9s3  => => resolve 10.8.20.8:5000/rockylinux-9:latest@sha256:d644d203142cd5b54ad2a83a203e1dee68af2229f8fe32f52a30c6e1d3c3a9e0
0.0s3  => => sha256:446f83f14b236772583d069e6f46a75e7e5456add656d1415a452618189fb825 64.31MB / 64.31MB
0.7s
 => => extracting sha256:446f83f14b236772583d069e6f46a75e7e5456add656d1415a452618189fb825
4.2s
 => [2/4] RUN dnf makecache
112.7s
 => [3/4] RUN dnf -y install httpd
38.5s
 => [4/4] RUN echo "<h1>Dockerfile Test on Aapche httpd 哈哈</h1>" > /var/www/html/index.html
0.3 s=> => exporting attestation manifest sha256:9c985b696945c93581f0c0e3cdb9f1b052a40b501433befd361be337c7da3b17
0.0s
 => exporting to image
5.7s
 => => exporting layers
3.8s
 => => exporting manifest sha256:9aa85ee39c9d6e02f459c993e35a4858363974724d074e33ae368aec06ae0808
0.0s
 => => exporting config sha256:50b05694fbd5fa20a5300727db975b2e8381366f7c48a775a58b1f6eba2fa87e
0.0s
 => => exporting attestation manifest sha256:9c985b696945c93581f0c0e3cdb9f1b052a40b501433befd361be337c7da3b17
0.0s
 => => exporting manifest list sha256:32f83ef8db2a3f7ba1141361e24ef04d9ff3931099840f83fc013228007b09c3
0.0s
 => => naming to docker.io/library/rocky-httpd:latest
0.0s
 => => unpacking to docker.io/library/rocky-httpd:latest
```

命令“docker build ...”根据当前目录下的 Dockerfile 文件构建镜像，如果文件名不同或者在其他目录中，可通过参数-f 指定，比如 docker build -f /tmp/build.docker，最终产生的镜像名为 rocky-httpd，标签为 latest。

3. 启动容器：

```
[root@master1 ~]# docker images                                          #先看看本地的镜像有哪些
IMAGE                                  ID             DISK USAGE   CONTENT SIZE   EXTRA
10.8.20.8:5000/library/debian:latest   d8f17b92dc7f       181MB         49.3MB    U
busybox:latest                         dc2d74b28e4c      6.74MB         2.23MB    U
debian:latest                          d8f17b92dc7f       181MB         49.3MB    U
debian:www                             a1723137ddc4       237MB         70.9MB
hello-world:latest                     5e2309035332      21.8kB         9.49kB    U
rocky-httpd:latest                     32f83ef8db2a       494MB          159MB

[root@master1 ~]# docker run -d -p 8080:80 rocky-httpd                   #启动容器，并把容器内的80端口映射到宿主机的8080
ae104665b2a61d73f987b836a66b43e51df00509a9b074d4e7e1fd1d7f61d7b7

[root@master1 ~]# docker ps                                              #显示正在运行的容器
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                                       NAMES
ae104665b2a6   rocky-httpd    "/usr/sbin/httpd -D …"   15 seconds ago   Up 14 seconds   0.0.0.0:8080->80/tcp, :::8080->80/tcp       happy_boyd
d3b245507122   debian:www     "/usr/sbin/nginx -g …"   3 hours ago      Up 3 hours      0.0.0.0:8081->80/tcp, :::8081->80/tcp       boring_bhaskara

[root@master1 ~]# curl localhost:8080                                    #浏览页面
Dockerfile Test on Aapche httpd
```

或者在浏览器中打开 http://10.10.10.11:8080，参考图 3-7。



图 3-7 浏览容器中的页面

如果想要修改页面的内容，请进入容器直接修改/var/www/html/index.html，参考下面的命令：

```
[root@master1 ~]# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                    ……
5be72a68411c   rocky-httpd    "/usr/sbin/httpd -D …"   3 seconds ago    Up 2 seconds    0.0.0.0:8080->80/tcp,……
[root@master1 ~]# docker exec -it 5be72a68411c /bin/bash
[root@5be72a68411c /]# vi /var/www/html/index.html                                          #修改页面内容
```

§3.1.5 使用外部存储

容器一旦被删除，里面的数据全部丢失了，所以容器中的重要数据需要保存在外部的存储设备上，比如跑数据库服务的容器，需要把数据库数据保存在容器之外。

方法 1：通过映射目录

就是把宿主机上的一个目录 X 和容器里的一个目录 Y 建立映射关系，此后在容器中往 Y 目录中拷贝文件实际上就是往 X 目录拷贝文件，在宿主机上往 X 目录拷贝文件，同样在容器里的 Y 目录看得到。

```
[root@master1 ~]# mkdir -p /var/lib/docker/disk01
[root@master1 ~]# echo "永久保存的数据" > /var/lib/docker/disk01/testfile.txt


[root@master1 ~]# docker images                                                             #查看本地有哪些镜像
IMAGE                                  ID             DISK USAGE   CONTENT SIZE   EXTRA
10.8.20.8:5000/library/debian:latest   d8f17b92dc7f       181MB         49.3MB    U
busybox:latest                         dc2d74b28e4c      6.74MB         2.23MB    U
debian:latest                          d8f17b92dc7f       181MB         49.3MB    U
debian:www                             a1723137ddc4       237MB         70.9MB
hello-world:latest                     5e2309035332      21.8kB         9.49kB    U
rocky-httpd:latest                     32f83ef8db2a       494MB          159MB    U

从 busybox 镜像运行容器，把外部的目录挂载到容器中的/mnt 目录上：
[root@master1 ~]# docker run -it -v /var/lib/docker/disk01:/mnt busybox /bin/sh
/ # df -hT
Filesystem     Type     Size  Used Avail Use% Mounted on
overlay        overlay   32G  3.8G   29G  12% /
tmpfs          tmpfs     64M     0   64M   0% /dev
shm            tmpfs     64M     0   64M   0% /dev/shm
/dev/sda1      xfs       32G  3.8G   29G  12% /mnt
tmpfs          tmpfs    1.9G     0  1.9G   0% /proc/asound
tmpfs          tmpfs    1.9G     0  1.9G   0% /proc/acpi
tmpfs          tmpfs    1.9G     0  1.9G   0% /proc/scsi
tmpfs          tmpfs    1.9G     0  1.9G   0% /sys/firmware

/ # cat /mnt/testfile.txt
永久保存的数据
/ # echo "The data from container" >> /mnt/testfile.txt
/ # cp /etc/resolv.conf /mnt/
/ # ls -l /mnt
total 8
-rw-r--r-- 1 root root 2123 Jul 31 06:30 resolv.conf
-rw-r--r-- 1 root root   46 Jul 31 06:30 testfile.txt
/ # exit                                                                                    #退出 bash 程序，容器也被停止了
```

看看刚才从容器中产生的文件：

```
[root@master1 ~]# ls -l /var/lib/docker/disk01/
total 8
-rw-r--r-- 1 root root 2123 Jul 31 14:30 resolv.conf
-rw-r--r-- 1 root root   46 Jul 31 14:30 testfile.txt
[root@master1 ~]# cat /var/lib/docker/disk01/testfile.txt
永久保存的数据
The data from container
```

容器虽然停止了，但是产生的数据还在。

方法 2：通过卷

```
# 创建卷 volume01:
[root@master1 ~]# docker volume create volume01
[root@master1 ~]# docker volume ls
DRIVER    VOLUME NAME
local     volume01

# 查看卷的详细信息:
[root@master1 ~]# docker volume inspect volume01
[
    {
        "CreatedAt": "2023-07-31T14:40:38+08:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/volume01/_data",
        "Name": "volume01",
        "Options": null,
        "Scope": "local"
    }
]
```

可见卷也是对应宿主机上的一个目录（如/var/lib/docker/volumes/volume01/_data）。

运行容器，并把卷挂载到/mnt 目录上：

```
[root@master1 ~]# docker images
REPOSITORY               TAG       IMAGE ID       CREATED         SIZE
rocky-httpd              latest    4d676aa8268d   2 hours ago     382MB
debian                   www       ced4634b22cc   4 hours ago     175MB
dokken/rockylinux-9      latest    65a1594065ee   7 hours ago     302MB
debian                   latest    278727474c29   3 days ago      116MB

[root@master1 ~]# docker run -it -v volume01:/mnt debian:latest /bin/bash
root@6683d1b3f86f:/# df -hT /mnt
Filesystem     Type  Size  Used Avail Use% Mounted on
/dev/sda1      xfs    32G  3.8G   29G  12% /mnt
root@6683d1b3f86f:/# echo "Hello, I am coming here from container!" > /mnt/hello.txt
root@6683d1b3f86f:/# exit
exit
[root@master1 ~]# cat /var/lib/docker/volumes/volume01/_data/hello.txt
Hello, I am coming here from container!
```

同一个卷还可以挂载到其他容器中：

```
[root@master1 ~]# docker run -v volume01:/var/volume01 debian:www /usr/bin/cat /var/volume01/hello.txt
Hello, I am coming here from container!
```

删除卷：

```
[root@master1 ~]# docker volume rm volume01
Error response from daemon: remove volume01: volume is in use - [6683d1b3f86f55087a0ab2796b8e76cd23d9ebab46583d5c8025effad3fff7b9,
4e41d9a2352dc0859ac7f1ffd75c207ea7054b355a4f19b15f05f6d75ed5107e]
```

被删除的卷正被容器使用，所以不能删，得先删除容器：

```
[root@master1 ~]# docker rm 6683d1b3f86f55087a0ab2796b8e76cd23d9ebab46583d5c8025effad3fff7b9
6683d1b3f86f55087a0ab2796b8e76cd23d9ebab46583d5c8025effad3fff7b9
[root@master1 ~]# docker rm 4e41d9a2352dc0859ac7f1ffd75c207ea7054b355a4f19b15f05f6d75ed5107e
4e41d9a2352dc0859ac7f1ffd75c207ea7054b355a4f19b15f05f6d75ed5107e
[root@master1 ~]# docker volume rm volume01
volume01
```

§3.1.6 容器组合

一个完整的应用可能涉及多个服务，比如企业的门户网站包括 Web 服务和数据库服务，前面已经讲过，一个容器一般只跑一个服务，这样就有必要组合两个容器来实现企业门户网站了，方便组合在一起的容器“共命运同生死”。采用

docker compose 命令很容易实现多个容器组合起来实现一个具体的应用。本小节以实际例子来演示如何进行容器组合。

1. 构建 Web 服务容器的配置文件：

```
[root@master1 ~]# vim Dockerfile                                                    #构建镜像的配置文件
FROM 10.8.20.8:5000/library/debian:latest

RUN echo "deb http://mirror.nju.edu.cn/debian/ bookworm main contrib non-free non-free-firmware" > /etc/apt/sources.list
RUN apt update
RUN apt install nginx -y
RUN rm -rf /var/www/html/*

EXPOSE 80
CMD ["/usr/sbin/nginx", "-g", "daemon off;"]
```

即以 debian 镜像为基础，额外安装 nginx 网站服务，并把容器内的 80 端口暴露给宿主机，容器默认的入口程序为 /usr/sbin/nginx -g 'gaemon off;'。如果采用私有镜像库，请把第一行替换成“FROM 10.8.20.8:5000/debian”。

再新建一个定义容器组合的文件：

```
[root@master1 ~]# vim docker-compose.yaml                                           #定义应用
services:
  db_service:                                                                       #服务名称
    image: 10.8.20.8:5000/library/mariadb:latest                                    #容器镜像
    volumes:                                                                        #外部存储
      - /var/lib/docker/disk01:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: RootPass                                                 #MySQL 数据库的 root 用户密码
      MYSQL_USER: john
      MYSQL_PASSWORD: JohnPass
      MYSQL_DATABASE: student_db
    ports:                                                                          #内外端口映射
      - "3306:3306"
  web_service:                                                                      #服务名称
    build: ./                                                                       #从本地文件 Dockerfile 构建容器
    ports:
      - "8081:80"                                                                   #容器中的 80 映射到宿主机的 8081
    volumes:
      - /var/lib/docker/disk02:/var/www/html
```

yaml 语言完全靠缩进量来体现父子兄弟关系，所以 services 定义了两个服务（两个儿子）db_service 和 web_service。

知识点讲解：

docker-compose.yaml 文件包含三个节，它们是 services、networks 和 volumes，分别定义服务、网络和存储卷，其中 networks 和 volumes 节允许省略。yaml 文件的整体结构如下：

```
services:
    service1:
        配置参数
    service2:
        配置参数
    ……

networks:
    frontend:
    backend:

volumes:
    db-data:
```

从 2023 年 6 月底开始，Compose V1 将不再被支持，并将从所有 Docker 桌面版本中移除。

yaml 语言的文件中只可能包含 Map（键:值对）、List（列表）、#注释行、---分割行、标量和空行。字母大小写敏感，使用缩进表示层级关系，缩进只能用空格，缩进一次采用多少空格没有严格规定，只要相同层级的元素左对齐即可。Map 中的“值”和 List 中的项目可为标量、Map 或 List，即 Map 和 List 可相互嵌套，而且嵌套的层数没有限

定。看一个例子：

```
[root@master1 ~]# cat example.yaml
#这是注释行
#下面是一个Map:
name: "Alice"
#下面是一个Map，值为List:
scores:
  - 60
  - 80
  - 100
#下面是分割行
---

jack:
  phone: 1368044320
  sex: male
  age: 20
students:
  - "Alice"
  - John
  - "Jack"
  - other: "Zsan Lisi"
---
gdpi:
  - computer:
      teacher1: "wangwei"
      teacher2: "rudingzhu"
  - english:
      worker1: "abc1"
      worker2: "moliping"
  - "others"
```

后面在 kubernetes 实战时会大量采用 yaml 语言定义资源对象，关于 yaml 语言更详细的介绍请去互联网上搜索学习。yaml 代码和 json 代码很容易相互转换，这是一个在线转换的网站 https://www.lddgo.net/convert/yaml-to-json。

分割行“---”的上下各是一个完整的 json 代码块，所以上面的 yaml 代码对应图 3-8 中三块 json 代码。

```
{
    "name": "Alice",
    "scores": [
        60,
        80,
        100
    ]
}
```

```
{
    "jack": {
        "phone": 1368044320,
        "sex": "male",
        "age": 20
    },
    "students": [
        "Alice",
        "John",
        "Jack",
        {
            "other": "Zsan Lisi"
        }
    ]
}
```

```
{
    "gdpi": [
        {
            "computer": {
                "teacher1": "wangwei",
                "teacher2": "rudingzhu"
            }
        },
        {
            "english": {
                "worker1": "abc1",
                "worker2": "moliping"
            }
        },
        "others"
    ]
}
```

图 3-8 Yaml 转换后的 Json 代码块

2．构建容器组合并运行：

```
[root@master1 ~]# docker compose config -q                    #检查 docker-compose.yaml 文件有无语法错
```

如果没有输出，表示没有语法错误，否则请改正。

```
[root@master1 ~]# docker compose pull                                  #最好手工下拉好容器镜像
[root@master1 ~]# docker compose up -d                  #根据当前目录中的 docker-compose.yml 文件构建容器组合
[+] Building 35.8s (9/9) FINISHED
 => [web_service internal] load build definition from Dockerfile                                                    0.0s
 => => transferring dockerfile: 339B                                                                                0.0s
 => [web_service internal] load .dockerignore                                                                       0.0s
 => => transferring context: 2B                                                                                     0.0s
 => [web_service internal] load metadata for docker.io/library/debian:latest                                        0.0s
 => CACHED [web_service 1/5] FROM docker.io/library/debian                                                          0.0s
 => [web_service 2/5] RUN echo "deb http://mirror.nju.edu.cn/debian/ bookworm main contrib non-free non-free-firmware" > /etc/apt/sources.list   0.3s
 => [web_service 3/5] RUN apt update                                                                               22.6s
 => [web_service 4/5] RUN apt install nginx -y                                                                     12.3s
 => [web_service 5/5] RUN rm -rf /var/www/html/*                                                                    0.4s
 => [web_service] exporting to image                                                                                0.2s
 => => exporting layers                                                                                             0.2s
 => => writing image sha256:db2652b186796b3c793e81cc73ec1d81bdcc0a2d3cfc883eadd30fdc0952a7ed                        0.0s
 => => naming to docker.io/library/root-web_service                                                                 0.0s
[+] Running 3/3
 ✔ Network root_default          Created                                                                            0.2s
 ✔ Container root-web_service-1  Started                                                                            0.5s
 ✔ Container root-db_service-1   Started                                                                    0.3s
```

“docker compose up ...”命令根据当前目录下的文件 docker-compose.yml 和 Dockerfile 构建容器并组合在一起，最后启动容器组合。参数-d 让容器在后台运行。

根据命令输出，我们发现：创建了网络 root_default，启动了两个容器 root-web_service-1 和 root-db_service-1，两个容器组合在一起，它们通过网络 root_default 互通。使用下面的命令进一步查看网络 root_default：

```
[root@master1 ~]# docker network inspect root_default
[
    {
        "Name": "root_default",
        "Id": "03c4faf5c217747668bda8e4d63488fd02fd1c910243f6675a8bd1eca9c1b31e",
        "Created": "2025-09-19T15:29:14.55660124+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv4": true,
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "12c5cf2095cfbe82c8573ffb27f38efee625033463405db8786949c4d4d26d28": {
                "Name": "root-db_service-1",
                "EndpointID": "565d33ddc18c693369b4093f2257ff67000ab4cc75a25f87d2f1b2127915b002",
                "MacAddress": "0a:dc:2f:c4:c1:c2",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            },
            "e3a718dab682f59ac4ecf7c8a39c512313d93c0043f4c7893c98f02ad42d3cda": {
                "Name": "root-web_service-1",
                "EndpointID": "32a0a6b9931696840c0a89177702ff0df211cfa256c0c30fbd11424e2b4cb4fc",
                "MacAddress": "96:84:b3:9e:81:8e",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
            "com.docker.compose.config-hash": "d920c4b7b178f40a0538499fc7463075fbd4b0a267bad431d315cb5126b88f4a",
            "com.docker.compose.network": "default",
            "com.docker.compose.project": "root",
            "com.docker.compose.version": "2.39.2"
        }
    }
]
```

可见为组合中的两个容器分配的 ip 地址分别是 172.18.0.2 和 172.18.0.3。

```
[root@master1 ~]# docker compose ps                                                                                    #查看正在运行的容器
CONTAINER ID   IMAGE            COMMAND                  CREATED          STATUS          PORTS                                          NAMES
252393940eb9   mariadb          "docker-entrypoint.s…"   5 minutes ago    Up 5 minutes    0.0.0.0:3306->3306/tcp, :::3306->3306/tcp      root-db_service-1
7a0d022b4e20   root-web_service "/usr/sbin/nginx -g …"   5 minutes ago    Up 5 minutes    0.0.0.0:8081->80/tcp, :::8081->80/tcp          root-web_service-1
```

如果发现某些容器没有启动，请仔细检查 Dockerfile 和 docker-compose.yml 文件，如果修改了 Dockerfile 文件，请运行命令“docker compose up -d --build”重新构建 web 服务所在的容器镜像并启动容器组合。

```
[root@master1 ~]# docker network ls                                                                                    #查看网络
NETWORK ID     NAME           DRIVER    SCOPE
e1c3e14039d7   bridge         bridge    local
f81d18f5f735   host           host      local
566c88aea10c   none           null      local
9eb8402a3bb2   root_default   bridge    local

[root@master1 ~]# ss -tnlp                                                                                             #查看宿主机上被监听的 tcp 端口
State  Recv-Q Send-Q  Local Address:Port    Peer Address:Port Process
LISTEN 0      128           0.0.0.0:22           0.0.0.0:*     users:(("sshd",pid=790,fd=3))
LISTEN 0      4096      127.0.0.1:38555          0.0.0.0:*     users:(("containerd",pid=796,fd=13))
LISTEN 0      4096          0.0.0.0:3306         0.0.0.0:*     users:(("docker-proxy",pid=11376,fd=4))
LISTEN 0      4096          0.0.0.0:8081         0.0.0.0:*     users:(("docker-proxy",pid=11425,fd=4))
LISTEN 0      128              [::]:22              [::]:*     users:(("sshd",pid=790,fd=4))
LISTEN 0      4096             [::]:3306            [::]:*     users:(("docker-proxy",pid=11381,fd=4))
LISTEN 0      4096             [::]:8081            [::]:*     users:(("docker-proxy",pid=11432,fd=4))
```

由此可知：最终构建了两个容器，一个容器跑数据库，另一个容器跑 nginx 网站服务器，且把容器内监听的端口映射到了宿主机上，分别是 3306（数据库）、8081（网站服务）。

3．验证

首先验证在宿主机上访问容器里的数据库：

```
[root@master1 ~]# dnf install mariadb -y                                                    #安装数据库客户端

[root@master1 ~]# mariadb -h 127.0.0.1 -u root -p -e "show variables like 'hostname';"
Enter password: <输入密码 RootPass，已经在 docker-compose.yml 中定义>
+---------------+--------------+
| Variable_name | Value        |
+---------------+--------------+
| hostname      | 252393940eb9 |
+---------------+--------------+

[root@master1 ~]# mariadb -h 127.0.0.1 -u john -p -e "show databases;"
Enter password:<输入密码 JohnPass>
+--------------------+
| Database           |
+--------------------+
| information_schema |
| student_db         |
+--------------------+
```

再来验证容器里的 web 服务器，默认页面不存在，所以先产生一个页面文件：

```
[root@master1 ~]# echo "<h1>Hello Docker Compose World! </h1>" > /var/lib/docker/disk02/index.html
[root@master1 ~]# curl 127.0.0.1:8081
<h1>Hello Docker Compose World! </h1>
```

也可以用浏览器浏览 http://10.10.10.11:8081。

课后实战练习：在跑网站服务的那个容器里安装 php，然后写一个动态网页，显示数据库中某个表的信息。提示：执行命令“docker exec -it root-web_service-1 /bin/bash”进入容器，然后在容器中执行“apt install php -y”安装 php。

4．针对容器组合的其他操作：

```
# 查看容器组合：
[root@master1 ~]# docker compose ps
NAME                IMAGE               COMMAND                  SERVICE             CREATED             STATUS              PORTS
root-db_service-1   mariadb             "docker-entrypoint.s…"   db_service          25 minutes ago      Up 25 minutes       0.0.0.0:3306->3306/tcp, :::3306->3306/tcp
root-web_service-1  root-web_service    "/usr/sbin/nginx -g …"   web_service         25 minutes ago      Up 25 minutes       0.0.0.0:8081->80/tcp, :::8081->80/tcp

# 查看容器组合的日志：
[root@master1 ~]# docker compose logs
root-db_service-1  | 2023-07-31 08:23:52+00:00 [Note] [Entrypoint]: Entrypoint script for MariaDB Server 1:11.0.2+maria~ubu2204 started.
root-db_service-1  | 2023-07-31 08:23:52+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
root-db_service-1  | 2023-07-31 08:23:52+00:00 [Note] [Entrypoint]: Entrypoint script for MariaDB Server 1:11.0.2+maria~ubu2204 started.
root-db_service-1  | 2023-07-31 08:23:52+00:00 [Note] [Entrypoint]: Initializing database files
……
root-db_service-1  | 2023-07-31  8:23:57 0 [Note] Server socket created on IP: '0.0.0.0'.
root-db_service-1  | 2023-07-31  8:23:57 0 [Note] Server socket created on IP: '::'.
root-db_service-1  | 2023-07-31  8:23:57 0 [Note] mariadbd: ready for connections.
root-db_service-1  | Version: '11.0.2-MariaDB-1:11.0.2+maria~ubu2204'  socket: '/run/mysqld/mysqld.sock'  port: 3306  mariadb.org

# 在容器中执行命令，容器所对应的服务名称就是在 docker-compose.yml 文件中定义的：
[root@master1 ~]# docker compose exec db_service /bin/bash                                  #进入跑数据库的容器
root@aa25802470a0:/#

# 停止容器组合：
[root@master1 ~]# docker compose stop
[+] Stopping 2/2
 ✔ Container root-web_service-1  Stopped                                                                    0.3s
 ✔ Container root-db_service-1   Stopped                                                                    0.5s

# 启动应用中的一个服务，顺带启动依赖服务：
[root@master1 ~]# docker compose up -d web_service
[+] Running 1/1
 ⠿ Container root-web_service-1  Started                                                                    0.4s
或者：
[root@master1 ~]# docker compose start web_service db_service
```

www.verybe

```
[root@master1 ~]# docker compose ps                                                    #带-a 参数查看所有的组合
NAME                  IMAGE                 COMMAND                  SERVICE           CREATED             STATUS              PORTS
root-web_service-1    root-web_service      "/usr/sbin/nginx -g …"   web_service       38 minutes ago      Up 2 minutes
0.0.0.0:8081->80/tcp, :::8081->80/tcp

# 删除容器组合以及持久存储:
[root@master1 ~]# docker compose down --volumes
[+] Stopping 2/2
 ✔ Container root-web_service-1  Stopped                                                                                0.3s
 ✔ Container root-db_service-1   Stopped                                                                                0.0s
```

命令 `docker compose --help` 获取更多的帮助信息。

课堂实战练习 1：5 分钟部署 wordpress

wordpress 是一个著名的搭建网站的开源项目，尤其被全世界的 IT 人员所喜爱，常用于搭建个人博客网站。

```
[root@master1 ~]# mkdir my_wordpress
[root@master1 ~]# cd my_wordpress/
[root@master1 my_wordpress]# vim docker-compose.yml                                    #新建容器组合配置文件
services:
  db_service:                                                                          #数据库服务
    # 我们使用 MariaDB 数据库:
    image: 10.8.20.8:5000/library/mariadb
    # 如果你真的想使用 MySQL 数据库，请注释掉上面一行，反注释下面一行。
    #image: mysql
    command: '--default-authentication-plugin=mysql_native_password'
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=Pass654321                                                 #数据库的 root 用户密码
      - MYSQL_USER=wordpress_id
      - MYSQL_PASSWORD=A1b2C3
      - MYSQL_DATABASE=wordpress_db
    expose:
      - 3306                                                                           #容器中数据库服务端口
  wordpress:                                                                           #网站服务
    image: 10.8.20.8:5000/library/wordpress
    volumes:
      - wp_data:/var/www/html
    ports:
      - 80:80                                                                          #容器中网站服务端口 80 映射到宿主机端口 80
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db_service
      - WORDPRESS_DB_USER=wordpress_id
      - WORDPRESS_DB_PASSWORD=A1b2C3
      - WORDPRESS_DB_NAME=wordpress_db
volumes:
  db_data:                                                                             #数据库服务的持久卷
  wp_data:                                                                             #网站服务的持久卷
```

持久卷就是容器中的某个目录映射到宿主机上的一个目录，这样容器就可以把数据保存早宿主机上，从而实现持久化存储，即使容器被销毁了，保存在宿主机上的数据还在。

```
[root@master1 my_wordpress]# docker compose config -q                                  #检查 docker-compose.yml 文件有无语法错
```

如果没有输出，表示没有语法错误，否则请改正。

```
[root@master1 my_wordpress]# docker compose up -d --remove-orphans                     #构建并启动容器组合
[+] Running 33/2
 ✔ wordpress 21 layers [⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿]      0B/0B      Pulled                                  170.1s
 ✔ db_service 10 layers [⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿⣿] 0B/0B      Pulled                                                          236.6s
[+] Running 5/5
 ✔ Network my_wordpress_default        Created                                                                          0.3s
 ✔ Volume "my_wordpress_db_data"       Created                                                                          0.0s
 ✔ Volume "my_wordpress_wp_data"       Created                                                                          0.0s
 ✔ Container my_wordpress-wordpress-1  Created                                                                         25.5s
 ✔ Container my_wordpress-db_service-1 Created                                                                         25.5s
Attaching to my_wordpress-db_service-1, my_wordpress-wordpress-1
my_wordpress-db_service-1 | 2023-09-26 01:29:38+00:00 [Note] [Entrypoint]: Entrypoint script for MariaDB Server 1:10.6.4+maria~focal started.
my_wordpress-wordpress-1  | WordPress not found in /var/www/html - copying now...
my_wordpress-db_service-1 | 2023-09-26 01:29:39+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
my_wordpress-db_service-1 | 2023-09-26 01:29:39+00:00 [Note] [Entrypoint]: Entrypoint script for MariaDB Server 1:10.6.4+maria~focal started.
my_wordpress-db_service-1 | 2023-09-26 01:29:39+00:00 [Note] [Entrypoint]: Initializing database files
my_wordpress-wordpress-1  | Complete! WordPress has been successfully copied to /var/www/html
……
```

从构建容器组合的显示结果发现：首先下拉数据库和 wordpress 镜像，然后创建两个持久卷，再分别从下拉的两个镜像启动两个容器，最后进入容器作配置。

```
[root@master1 my_wordpress]# docker network ls
```

```
NETWORK ID     NAME                   DRIVER    SCOPE
ab44d42a39c1   bridge                 bridge    local
81eb8c49ca3b   host                   host      local
1a0db3618555   my_wordpress_default   bridge    local
7a1b79034b16   none                   null      local
```

其中 my_wordpress_default 是本组合新建的网桥，用下面的命令进一步查看网桥信息：

```
[root@master1 my_wordpress]# docker network inspect my_wordpress_default
[
    {
        "Name": "my_wordpress_default",
        "Id": "1a0db36185553587797a15649805d04db22405a481a10eeb4f73501822d6d46d",
        "Created": "2025-09-25T10:55:53.639279921+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv4": true,
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "22997363549bd9f104e56e611c8842a45a43c6153f1e5bfacf36f16a02475eac": {
                "Name": "my_wordpress-db-1",
                "EndpointID": "96177b3300eab4769ab7ed491f6de382512c1562fec038cfc34e08f4cd17b6cc",
                "MacAddress": "be:f6:48:96:2d:53",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            },
            "aed37a77e84bfbd78bb8c325b56a63055cc2ccdea34ae17b32f44683fff56207": {
                "Name": "my_wordpress-wordpress-1",
                "EndpointID": "878054b2aede0010d8d1208fdf2926a7e55c825d778bf07ba09fd50bb9b35712",
                "MacAddress": "8e:bf:0d:90:3c:a4",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
            "com.docker.compose.config-hash": "83a3beb3cd1509b448f32fc5161f03356d2a35cec427728afa1c0a80d5068a43",
            "com.docker.compose.network": "default",
            "com.docker.compose.project": "my_wordpress",
            "com.docker.compose.version": "2.39.2"
        }
    }
]
```

可见网段是 172.18.0.0/16，已经给两个容器分配了 ip 地址 172.18.0.2 和 172.18.0.3。

```
[root@master1 my_wordpress]# docker compose ls                                          #查看正在运行的容器组合
NAME                STATUS              CONFIG FILES
my_wordpress        running(2)          /root/my_wordpress/docker-compose.yml

[root@master1 my_wordpress]# docker ps                                                  #查看正在运行的容器
CONTAINER ID   IMAGE                COMMAND                  CREATED         STATUS         PORTS                                  NAMES
18d0467c5ae5   wordpress:latest     "docker-entrypoint.s…"   4 minutes ago   Up 3 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp      my_wordpress-wordpress-1
fdabf459ad95   mariadb:10.6.4-focal "docker-entrypoint.s…"   4 minutes ago   Up 3 minutes   3306/tcp                               my_wordpress-db-1

[root@master1 my_wordpress]# ss -tnlp                                                   #查看被监听的 tcp 端口
State       Recv-Q      Send-Q            Local Address:Port             Peer Address:Port      Process
LISTEN      0           128                     0.0.0.0:22                    0.0.0.0:*          users:(("sshd",pid=664,fd=3))
LISTEN      0           4096                    0.0.0.0:80                    0.0.0.0:*          users:(("docker-proxy",pid=3942,fd=4))
LISTEN      0           128                        [::]:22                       [::]:*          users:(("sshd",pid=664,fd=4))
LISTEN      0           4096                       [::]:80                       [::]:*          users:(("docker-proxy",pid=3949,fd=4))

[root@master1 my_wordpress]# docker volume ls                                           #查看持久卷
DRIVER    VOLUME NAME
local     my_wordpress_db_data
local     my_wordpress_wp_data

[root@master1 my_wordpress]# docker volume inspect my_wordpress_db_data
[
    {
        "CreatedAt": "2023-09-26T09:48:01+08:00",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.project": "my_wordpress",
```

```
            "com.docker.compose.version": "2.21.0",
            "com.docker.compose.volume": "db_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/my_wordpress_db_data/_data",----------#持久卷对应宿主机上的目录
        "Name": "my_wordpress_db_data",
        "Options": null,
        "Scope": "local"
    }
]

[root@master1 my_wordpress]# docker volume inspect my_wordpress_wp_data
[
    {
        "CreatedAt": "2023-09-26T09:48:01+08:00",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.project": "my_wordpress",
            "com.docker.compose.version": "2.21.0",
            "com.docker.compose.volume": "wp_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/my_wordpress_wp_data/_data",
        "Name": "my_wordpress_wp_data",
        "Options": null,
        "Scope": "local"
    }
]

在宿主机上显示 wordpress 的 php 代码文件:
[root@master1 my_wordpress]# ll /var/lib/docker/volumes/my_wordpress_wp_data/_data/
总用量 244
-rw-r--r--  1 33 tape   405  2 月  6  2020 index.php
-rw-r--r--  1 33 tape 19915  1 月  1  2023 license.txt
-rw-r--r--  1 33 tape  7399  7 月  6 01:41 readme.html
-rw-r--r--  1 33 tape  7211  5 月 13 05:35 wp-activate.php
drwxr-xr-x  9 33 tape  4096  8 月 29 22:14 wp-admin
-rw-r--r--  1 33 tape   351  2 月  6  2020 wp-blog-header.php
-rw-r--r--  1 33 tape  2323  6 月 14 22:11 wp-comments-post.php
-rw-rw-r--  1 33 tape  5492  9 月 21 15:01 wp-config-docker.php
-rw-r--r--  1 33 tape  5596  9 月 26 09:48 wp-config.php
-rw-r--r--  1 33 tape  3013  2 月 23  2023 wp-config-sample.php
drwxr-xr-x  8 33 tape   112  9 月 26 10:11 wp-content
-rw-r--r--  1 33 tape  5638  5 月 31 02:48 wp-cron.php
drwxr-xr-x 27 33 tape 12288  8 月 29 22:14 wp-includes
-rw-r--r--  1 33 tape  2502 11 月 27  2022 wp-links-opml.php
-rw-r--r--  1 33 tape  3927  7 月 16 20:16 wp-load.php
-rw-r--r--  1 33 tape 49441  7 月 17 21:18 wp-login.php
-rw-r--r--  1 33 tape  8537  6 月 22 22:36 wp-mail.php
-rw-r--r--  1 33 tape 25602  7 月 25 14:35 wp-settings.php
-rw-r--r--  1 33 tape 34385  6 月 20 02:27 wp-signup.php
-rw-r--r--  1 33 tape  4885  6 月 22 22:36 wp-trackback.php
-rw-r--r--  1 33 tape  3236  6 月 14 22:11 xmlrpc.php
```

好了，现在可以浏览初始化页面，打开 http://10.10.10.11 网站（其中 10.10.10.11 为宿主机的 Ip），具体参考图 3-7、图 3-8、图 3-9 所示操作。

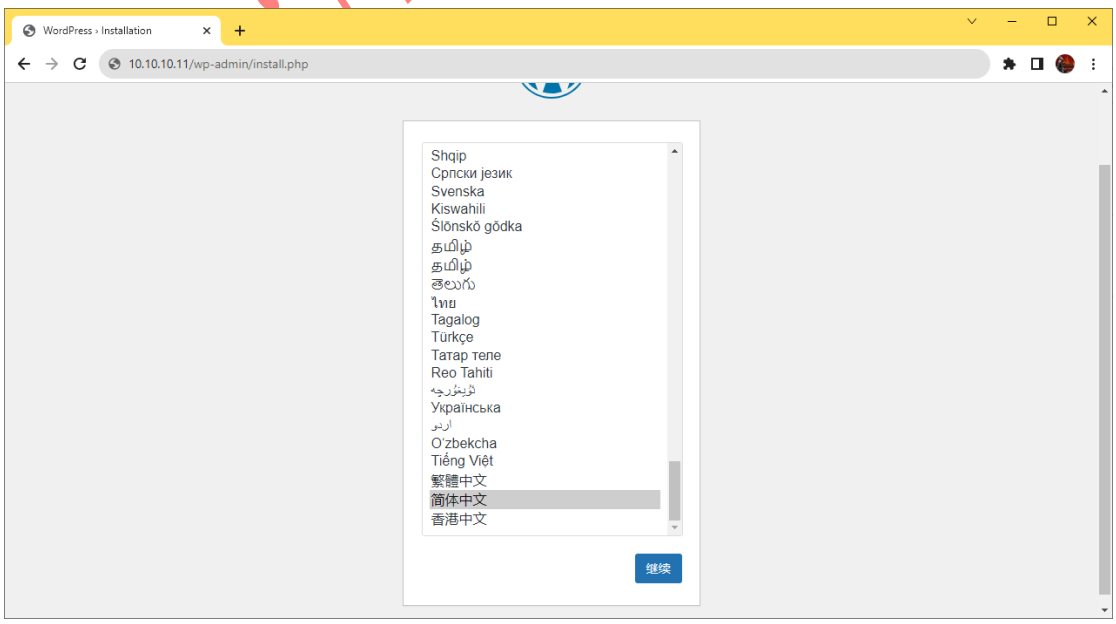


图 3-7 初始化 wordpress 之一



图 3-8 初始化 wordpress 之二



图 3-9 初始化 wordpress 之三

接下来就可以登录网站后台了，打开网址 http://10.10.10.11/wp-admin，参考图 3-10 所示。

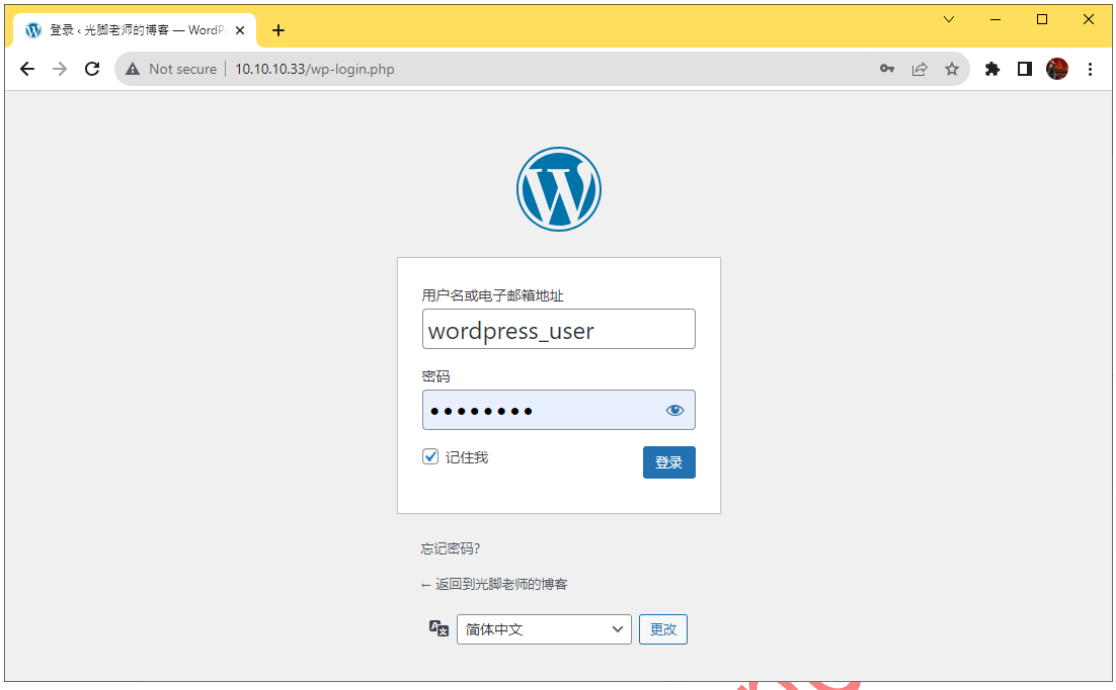


图 3-10 登录网站后台

在后台你可以设置页面布局，添加页面内容等，总之，功能恨强悍，参考图 3-11。



图 3-11 wordpress 的管理后台

点击上图中左侧的“文章”，你就可以写博客了，然后浏览 10.10.10.11 就能看到你的个人网站了。

每位同学写一篇博客文章，在页面上看到文章就算完成实验，平时实验分就加 10 分。

§3.1.7 容器网络

一个容器到底怎么接入外网呢？这是本节要介绍的内容。一共存在四种网络类型，分别是：

1. host(宿主机)模式

容器直接使用宿主机的 ip 与外界通信，所以服务端口不能与宿主机的服务端口冲突。运行容器时通过参数--network=host 指定主机模式，参考图 3-7 进一步理解 host 模式，注意 ip 地址相同。

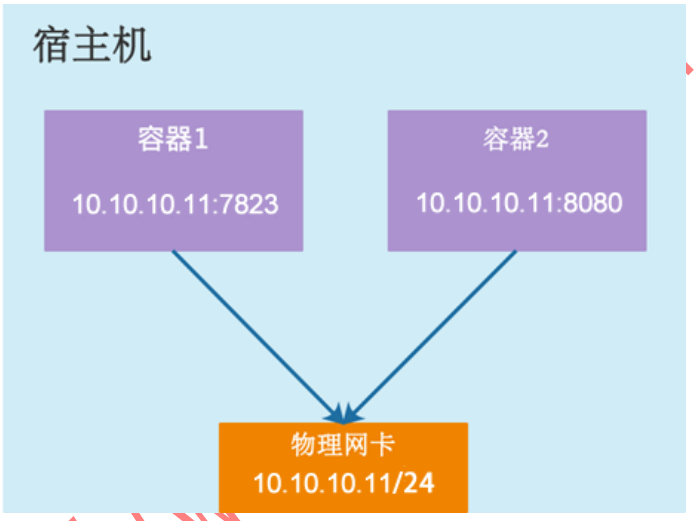


图 3-7 主机模式

下面的例子从 busybox 镜像启动容器，并使用宿主机的 ip：

```
[root@master1 ~]# docker run -it --network=host busybox /bin/sh
```

2. container(容器)模式

和另一个容器共享 ip 地址和端口范围，运行容器时通过参数--network=container:<NAME_or_ID>指定容器模式，其中<Name_or_ID>为另一个容器的名称或者 id 号。请参考图 3-8 进一步理解。

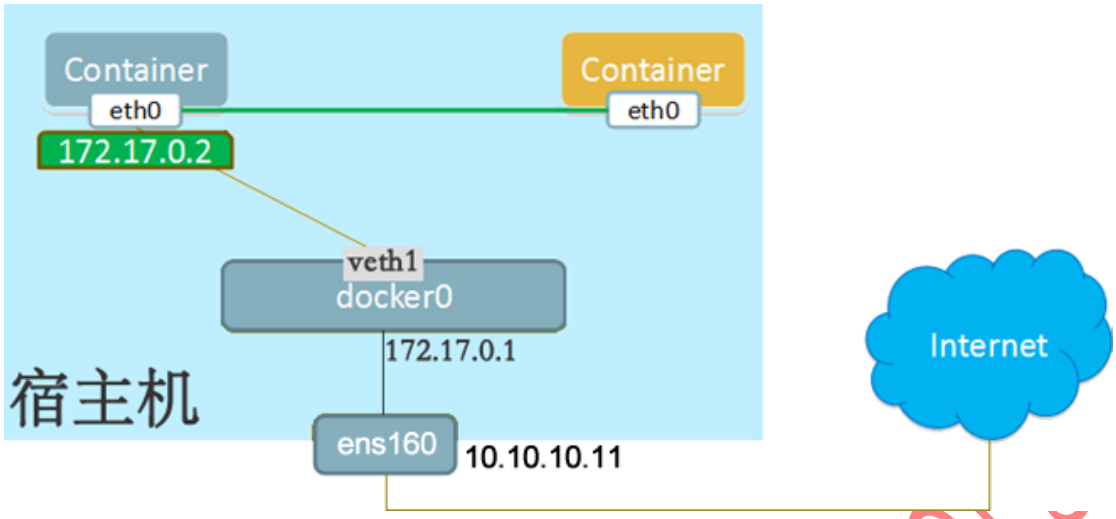


图 3-8 容器模式

图 3-8 中的右上角的容器为容器模式网络，它共享左上角的容器的 ip（即 172.17.0.2）。在 kubernetes 中，pod 中的其他容器都是共享 pause 容器的 ip。

```
[root@master1 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED          STATUS          PORTS     NAMES
fe0065b3521f   debian    "/usr/bin/sleep 20m"     52 seconds ago   Up 51 seconds             crazy_black
0e0df8387363   debian    "/bin/bash"              28 minutes ago   Up 19 minutes             adoring_curran

[root@master1 ~]# docker run -it --network=container:fe0065b3521f busybox /bin/sh
```

3. none 模式

none 模式的容器没有网卡、IP、路由等信息，需要我们自己事后为容器添加网卡、配置 IP 等。运行容器时通过参数 --network=none 指定 none 模式，参考图 3-9 所示。

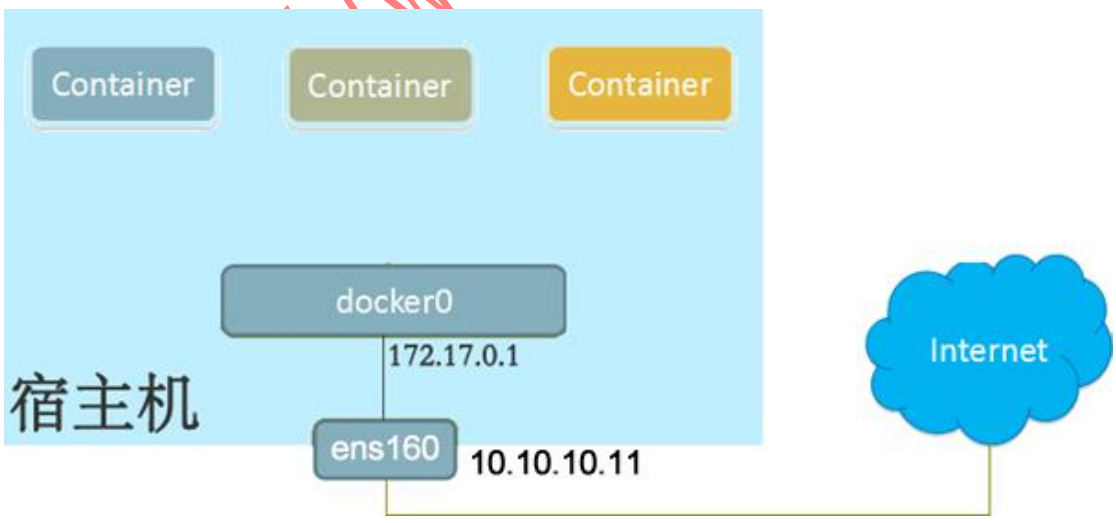


图 3-9 none 模式

4. bridge(网桥)模式

这是默认的联网方式，如果要指定参数，格式为--network=bridge。当 Docker 服务进程启动时，会在宿主机上创建一个名为 docker0 的虚拟网桥并给网桥配置一个 ip 地址，容器会连接到这个虚拟网桥上。虚拟网桥类似于物理交换机，这样宿主机上的所有容器就通过此“交换机”连在了一个二层网络中。从 docker0 子网中分配一个 IP 给容器使用，并设置 docker0 的 IP 地址为容器的默认网关，参考图 3-10 所示。

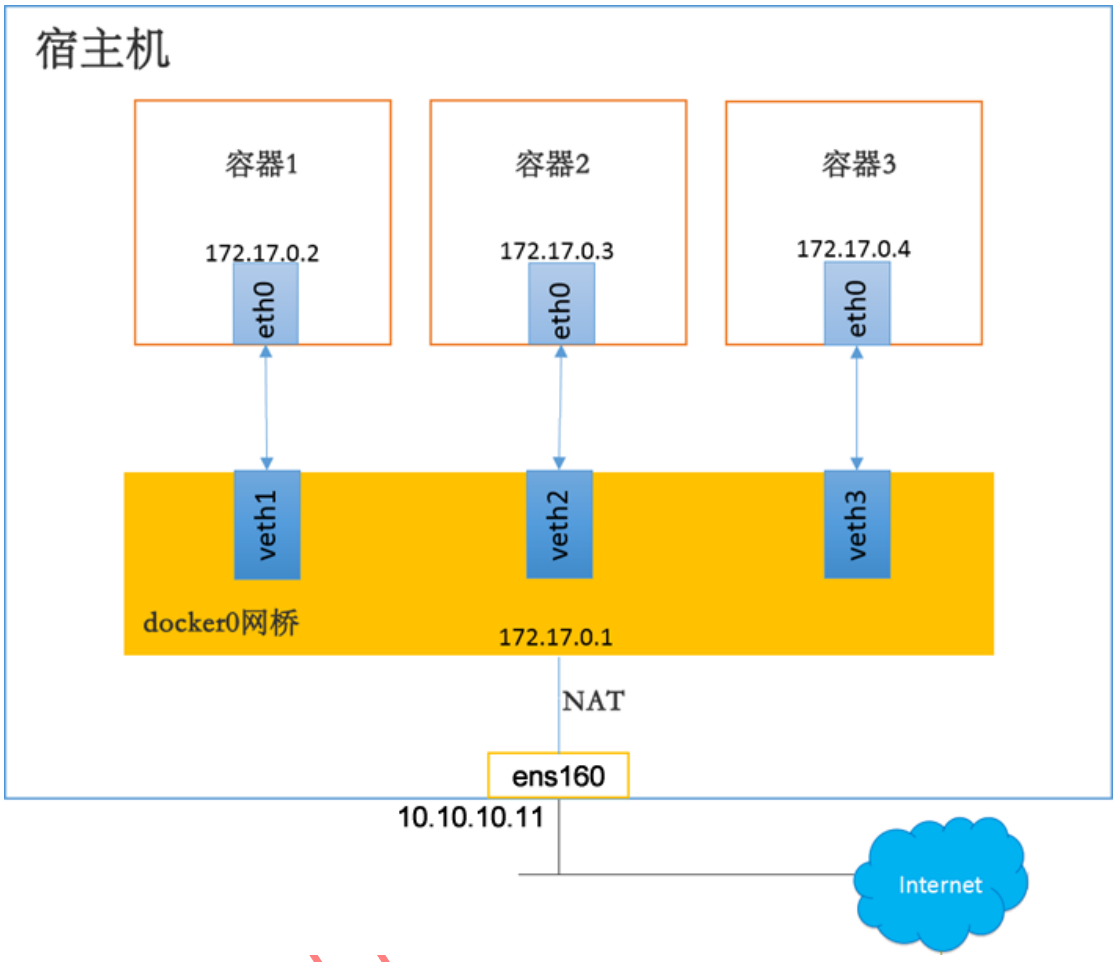


图 3-10 网桥模式

关于图 3-10 中的虚拟网元 veth?，请参考“§7.2 网络名字空间”小节。

显示网络：

```
[root@master1 ~]# docker network ls
NETWORK ID     NAME           DRIVER    SCOPE
45d5d71e96ab   bridge         bridge    local
386ea8c01278   host           host      local
6a5c7de4a23c   none           null      local
54c656e33c21   root_default   bridge    local
```

查看网桥的具体信息：

```
[root@master1 ~]# docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "45d5d71e96ab245a9f49cf3441e0777813fd9deb48d2a0485e4ce5e85782fd7a",
        "Created": "2023-07-31T07:13:08.915936855+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
```

```
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

创建一个网络：

```
[root@master1 ~]# docker network create --subnet 192.168.100.0/24 network01
13e6068de7d41d23bf70f0a99ba1e3d1774afb21f9d570ec93cdd3519a322528
```

默认的类型为网桥，网关为 192.168.100.1，创建网络时可以采用参数--driver 和--gateway 指定类型和网关。创建 bridge 类型的网络实际上创建了一个网桥，网桥的 ip 就是新建网络的网关。

运行容器并指定采用 network01 网络：

```
[root@master1 ~]# docker run -i -t --net network01 busybox /bin/sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
69: eth0@if70: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:64:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.100.2/24 brd 192.168.100.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # ip r
default via 192.168.100.1 dev eth0
192.168.100.0/24 dev eth0 proto kernel scope link src 192.168.100.2
[root@9b1edb37436c /]# ping -c 3 www.bing.com
PING dual-a-0001.a-msedge.net (2.16.241.157) 56(84) bytes of data.
64 bytes from a2-16-241-157.deploy.static.akamaitechnologies.com (2.16.241.157): icmp_seq=1 ttl=127 time=318 ms
64 bytes from a2-16-241-157.deploy.static.akamaitechnologies.com (2.16.241.157): icmp_seq=2 ttl=127 time=329 ms
64 bytes from a2-16-241-157.deploy.static.akamaitechnologies.com (2.16.241.157): icmp_seq=3 ttl=127 time=312 ms

--- dual-a-0001.a-msedge.net ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 5505ms
rtt min/avg/max/mdev = 312.451/319.738/328.876/6.847 ms
/ #                                                                    按 Ctrl+p, Ctrl+q 退出容器

[root@master1 ~]# docker ps
CONTAINER ID   IMAGE                      COMMAND                  CREATED          STATUS
c6f71d9a5036   busybox                    "sh"                     24 seconds ago   Up 23 seconds
22997363549b   10.8.20.8:5000/mariadb     "docker-entrypoint.s…"   27 hours ago     Up 5 minutes
aed37a77e84b   10.8.20.8:5000/wordpress   "docker-entrypoint.s…"   27 hours ago     Up 5 minutes    0.0.0.0:
```

给正在运行的容器（假如容器的 Id 号为 c6f71d9a5036）附加另一个网络上的 ip：

```
[root@master1 ~]# docker network connect --ip 192.168.100.10 network01 c6f71d9a5036
[root@master1 ~]# docker exec c6f71d9a5036 ip route
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 proto kernel scope link src 172.17.0.2
192.168.100.0/24 dev eth1 proto kernel scope link src 192.168.100.10
```

此时容器实际上同时接入了两个网络。

从容器上断开网络：

```
[root@master1 ~]# docker network disconnect network01 c6f71d9a5036
```

删除网络（网络上如果还有容器就不能删除，可用 docker network inspect network01 查看）：

```
[root@master1 ~]# docker network rm network01

[root@master1 ~]# docker network prune                                          #删除全部不再使用的网络
WARNING! This will remove all custom networks not used by at least one container.
Are you sure you want to continue? [y/N] y
Deleted Networks:
root_default
```

以 host 网络模式运行容器：

```
[root@master1 ~]# docker run -d --network=host rocky-httpd
39989a03b9ec3ec65cb4677cda223e72abab9ea5a873c92bb215717354d5460b
[root@master1 ~]# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS     NAMES
39989a03b9ec   rocky-httpd    "/usr/sbin/httpd -D …"   8 seconds ago    Up 6 seconds              upbeat_elbakyan

[root@master1 ~]# ss -tnlp
State       Recv-Q      Send-Q            Local Address:Port             Peer Address:Port      Process
LISTEN      0           128                     0.0.0.0:22                    0.0.0.0:*          users:(("sshd",pid=1014,fd=5))
LISTEN      0           128                        [::]:22                       [::]:*          users:(("sshd",pid=1014,fd=7))
LISTEN      0           128                           *:80                          *:*          users:(("httpd",pid=8872,fd=4)...

[root@master1 ~]# curl localhost
Dockerfile Test on Aapche httpd
```

§3.1.8 实战：容器里的 DNS

下面我们在 docker 容器中部署域名服务，然后解析 www.abc.com 域名。人们常使用 bind 开源软件实现 DNS 服务，下面我们就采用别人已经构建好的 bind 容器来实现 DNS 服务。

```
[root@master1 ~]# docker pull 10.8.20.8:5000/sameersbn/bind:latest
[root@master1 ~]# docker tag 10.8.20.8:5000/sameersbn/bind:latest sameersbn/bind:latest
```

如果不能访问局域网 10.8.20.8，请采用命令 docker pull sameersbn/bind 下拉。

```
[root@master1 ~]# mkdir -p /opt/docker/dns-server
[root@master1 ~]# docker run --name dns-server -d --restart=always --publish 53:53/tcp --publish 53:53/udp \
 --publish 10000:10000/tcp --volume /opt/docker/dns-server:/data sameersbn/bind:latest
```

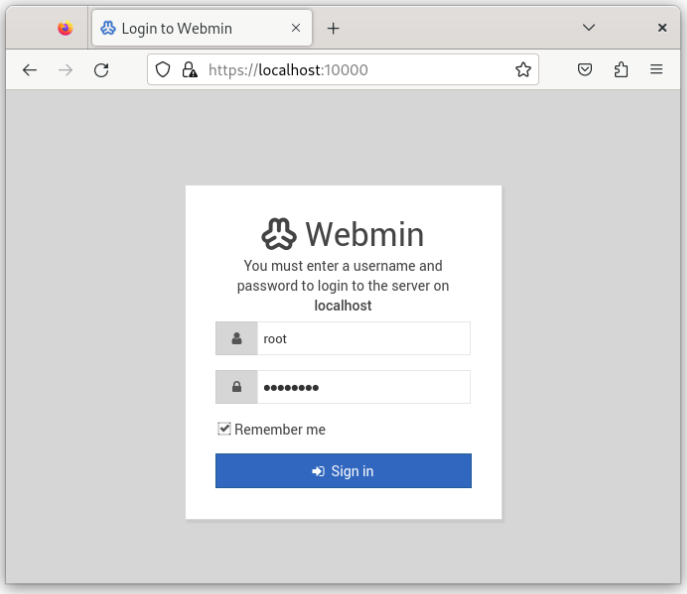
创建的容器名称为 dns-server，参数-p 53:53/udp 绑定容器 53 端口到宿主机的 53 端口，DNS 默认端口；-p 10000:10000 图形化界面管理器端口；-volume /dns-server:/data 挂载本地目录作为 dns 配置存储。

如果启用了防火墙，请放行 10000/tcp、53/tcp 和 53/udp 端口，执行命令：

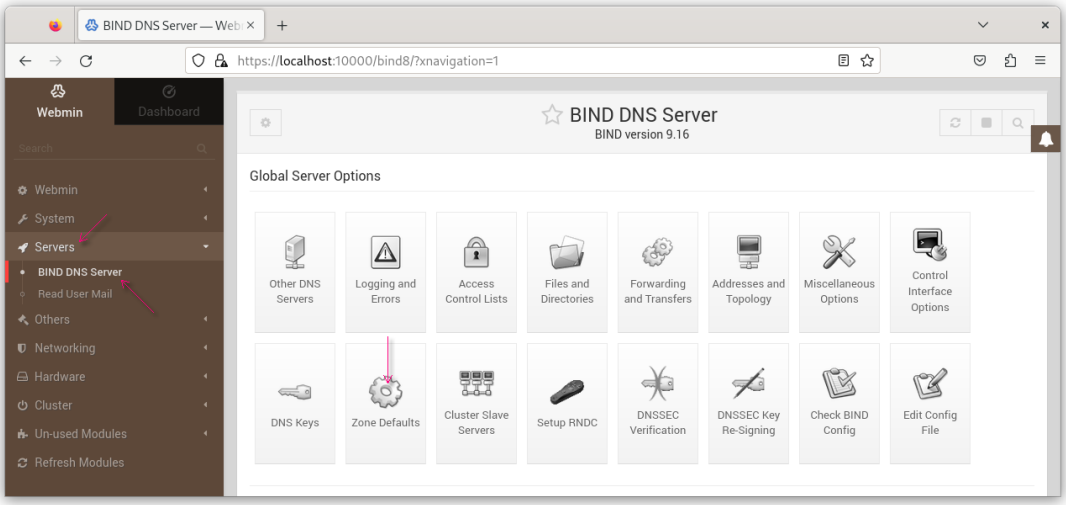
```
[root@master1 ~]# firewall-cmd --add-port={53,10000}/tcp --add-port=53/udp -permanent
[root@master1 ~]# firewall-cmd --reload
```

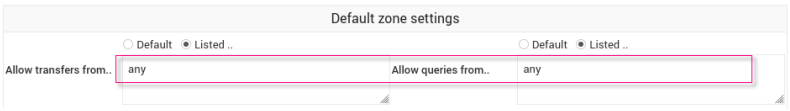
然后访问地址 https://10.10.10.11:10000，默认账户是 root，默认的密码是 password，登录界面如下。

https://www.veryopen.cn

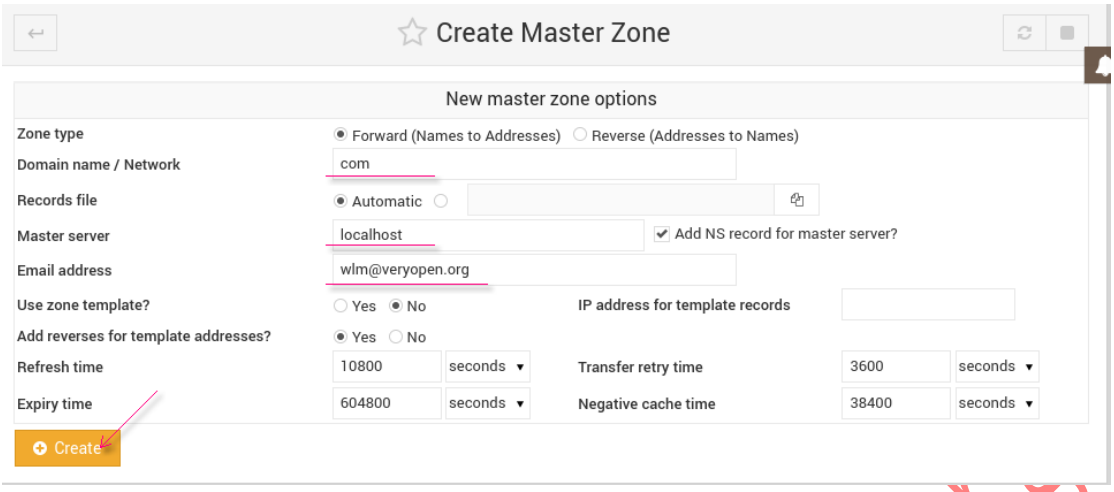


登录之后修改默认设置，参考下图：

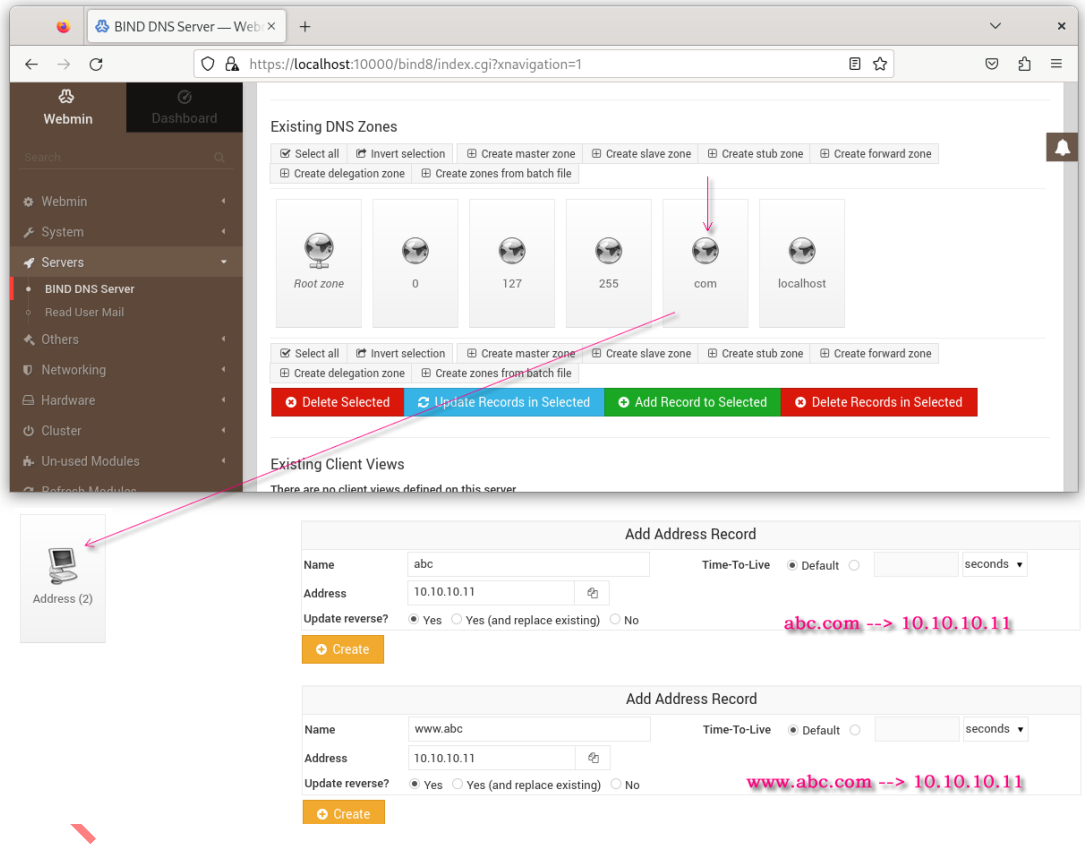




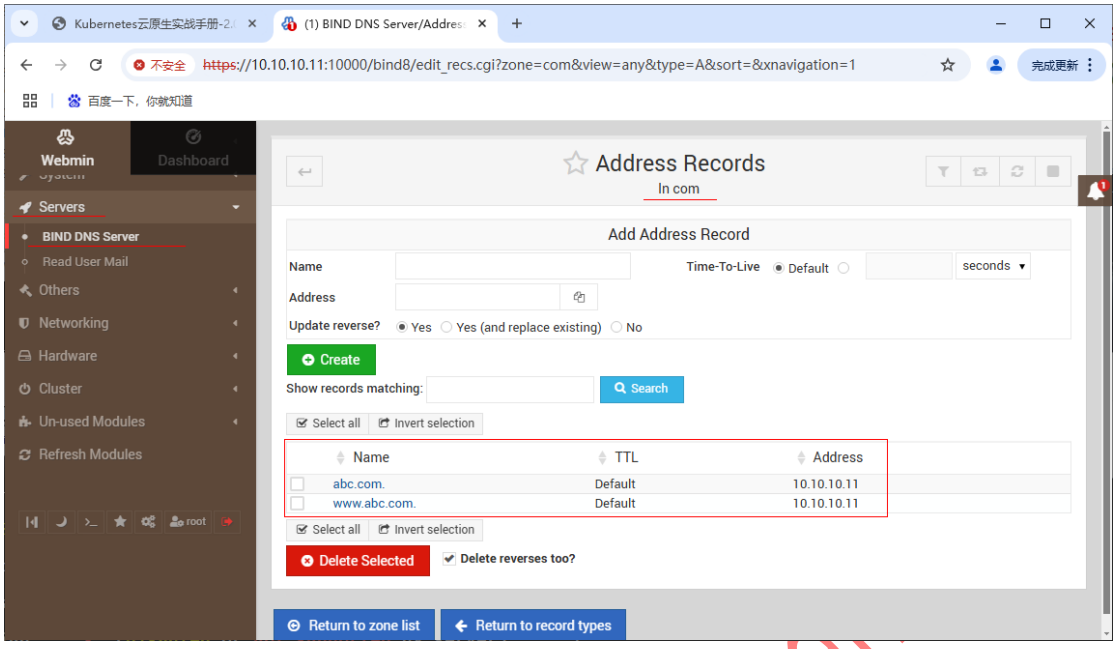
然后创建新的主区域，点击左侧 Servers>BIND DNS Server，再点击右侧的“Create master zone”按钮：



最后在 com 区域增加 A 记录：



最终看到如下页面：

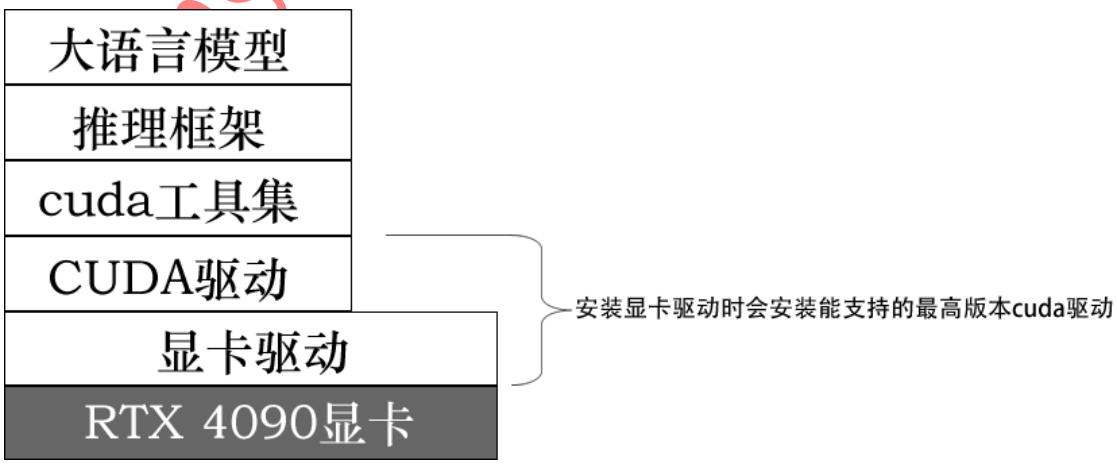



```
[root@master1 ~]# docker restart dns-server                    #重启容器
```

在其他需要使用 dns 的电脑上，设置第一个 dns 为 10.10.10.11 即可。比如在 Rocky 的电脑上执行命令修改 DNS：nmcli c modify ens160 ipv4.dns "10.10.10.11 9.9.9.9"，然后重启电脑之后直接 ping www.abc.com 或者 ping abc.com，看看是否解析成 10.10.10.11，但是 ping 不一定准确。更好的方法是：或者在 windows 上执行 ipconfig /flushdns、nslookup -type=A www.abc.com. 10.10.10.11，或者在 linux 上执行 dnf install -y bind-utils 、dig @10.10.10.11 www.abc.com. A +short。

§3.1.9 部署大语言模型

技术选型：一块英伟达 4090d 24GB 显卡 + Open WebUI + vLLM + gemma-4-26B-A4B-it-AWQ-4bit。



假设显卡驱动、cuda 驱动和 cuda 工具集都已经部署好了，且谷歌开源大语言模型 gemma-4-26B-A4B-it-AWQ-4bit 已经下载到 models/gemma-4-26B-A4B-it-AWQ-4bit 目录中。关于如何安装显卡驱动等，请参考《本地运行和微调 LLMs》，可从 https://www.veryopen.org 下载。

https://www.veryopen.org

1. 添加 INVIDIA 运行时（会修改/etc/docker/daemon.json 文件）：

```
root@fujunservers:~# nvidia-ctk runtime configure --runtime=docker
root@fujunservers:~# more /etc/docker/daemon.json
{
  "registry-mirrors": [
    "https://docker.1panel.live",
    "https://docker.1ms.run",
    "https://dytt.online",
    "https://docker-0.unsee.tech",
    "https://lispy.org",
    "https://docker.xiaogenban1993.com",
    "https://666860.xyz",
    "https://hub.rat.dev",
    "https://docker.m.daocloud.io",
    "https://demo.52013120.xyz",
    "https://proxy.vvvv.ee",
    "https://registry.cyou",
    "https://mirror.ccs.tencentyun.com",
    "https://no3hfwj9.mirror.aliyuncs.com"
  ],

    "runtimes": {
        "nvidia": {
            "args": [],
            "path": "nvidia-container-runtime"
        }
    }
}

root@fujunservers:~# systemctl restart docker
```

2. 测试能否在 gpu 中运行：

```
llm@fujunservers:~$ sudo docker pull swr.cn-north-4.myhuaweicloud.com/ddn-k8s/docker.io/nvidia/cuda:12.2.0-base-
ubuntu22.04
docker image tag swr.cn-north-4.myhuaweicloud.com/ddn-k8s/docker.io/nvidia/cuda:12.2.0-base-ubuntu22.04
nvidia/cuda:12.0-base-ubuntu22.04

docker run --rm --runtime=nvidia --gpus all nvidia/cuda:12.0-base-ubuntu22.04 nvidia-smi
Sun Apr 19 07:21:53 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 595.58.03              Driver Version: 595.58.03      CUDA Version: 13.2     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 4090 D      Off |   00000000:AF:00.0 Off |                  Off |
| 31%   28C    P8              4W /  425W |      15MiB /  24564MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

能看到 rtx 显卡信息表，说明一切正常。

3. 编写容器组合文件 docker-compose.yml：

```
llm@fujunservers:~$ vim docker-compose.yml
services:
  # ─── 1. vLLM 推理服务 ──────────────────────────────────────────────
  vllm:
    image: vllm/vllm-openai:gemma4-cu130
    container_name: llm-stack-vllm
    runtime: nvidia
    ipc: host
    ports:
      - "8000:8000"
    volumes:
      - ./models/gemma-4-26B-A4B-it-AWQ-4bit:/models
    command: >
      /models
      --quantization compressed-tensors
      --dtype float16
      --max-model-len 32768 ------------------------------------------------------------#16k 上下文（K-V 缓存，占用显存）
      --gpu-memory-utilization 0.88 ------------------------------------------------------------ #模型占用显存 88%
      --served-model-name gemma-4-26B-A4B-it-AWQ-4bit
      --api-key ILoveGemma4
```

```
      --trust-remote-code
      --reasoning-parser gemma4
      --tool-call-parser gemma4
      --enable-auto-tool-choice
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]
    healthcheck:
      test: [
        "CMD", "curl", "-f",
        "-H", "Authorization: Bearer ILoveGemma4",
        "http://localhost:8000/v1/models"
      ]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 180s
    environment:
      - HF_ENDPOINT=https://hf-mirror.com                                    #hugging face 国内镜像
    restart: unless-stopped
    networks:
      - llm-network

  # ── 2. Open WebUI ──────────────────────────────
  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: llm-stack-open_webui
    ports:
      - "3000:8080"
    volumes:
      - ./open_webui_data:/app/backend/data
    environment:
      - OPENAI_API_BASE_URL=http://vllm:8000/v1
      - OPENAI_API_KEY=ILoveGemma4
      - WEBUI_SECRET_KEY=ILoveOpenWebUI
      - ENABLE_SIGNUP=true
      - HF_ENDPOINT=https://hf-mirror.com
    restart: unless-stopped
    depends_on:
      vllm:
        condition: service_healthy
      pipelines:
        condition: service_started
    networks:
      - llm-network

# ── 容器网络与数据卷定义 ──────────────────────────────────────────
networks:
  llm-network:
    driver: bridge
```

检查 docker-compose-llm.yaml 的语法错误：

```
llm@fujunservers:~$ docker compose docker-compose.yml config -q
```

如果没有错误，就先下拉好容器镜像：

```
llm@fujunservers:~$ sudo docker compose pull
llm@fujunservers:~$ sudo docker images
IMAGE                                      ID             DISK USAGE   CONTENT SIZE   EXTRA
ghcr.io/open-webui/open-webui:main         c2e4723fdbca       6.7GB         1.72GB   U
vllm/vllm-openai:gemma4-cu130              0d152595cd94      27.6GB          7.6GB   U
```

vllm/vllm-openai:latest 镜像很大，有 9.61GB，可去 https://hub.docker.com/r/vllm/vllm-openai/tags?name=gemma4 查看其他镜像。

4. 正式启动

以后台模式启动两个容器（刚开始建议先启动一个 vllm 服务：sudo docker compose up -d vllm，看看有没有错）：

```
llm@fujunservers:~$ sudo docker compose up -d
```

以后如果只修改了其中一个服务，照样可以用上面的命令重建那个服务，而没有修改的服务不会重建。

实时查看启动日志（按 Ctrl+C 停止查看，容器继续运行）

```
llm@fujunservers:~$ docker compose logs -f
```

5. 开始聊天

浏览 http://<Open WebUI 容器的宿主机 ip>:3000，首次浏览会提示创建一个用户（第一次创建的用户就是管理员），以后再登陆就不能自行创建用户了，需要管理员协助新建用户（管理员登陆后的管理员面板上管理用户）。

§3.1.10 可视化管理工具

Portainer 是一款轻量级的应用，它提供了图形化界面，可让我们轻松构建和管理 Docker、Docker Swarm、Kubernetes 和 Azure ACI 中的容器。它分为社区版（CE）和商业版（BE）。

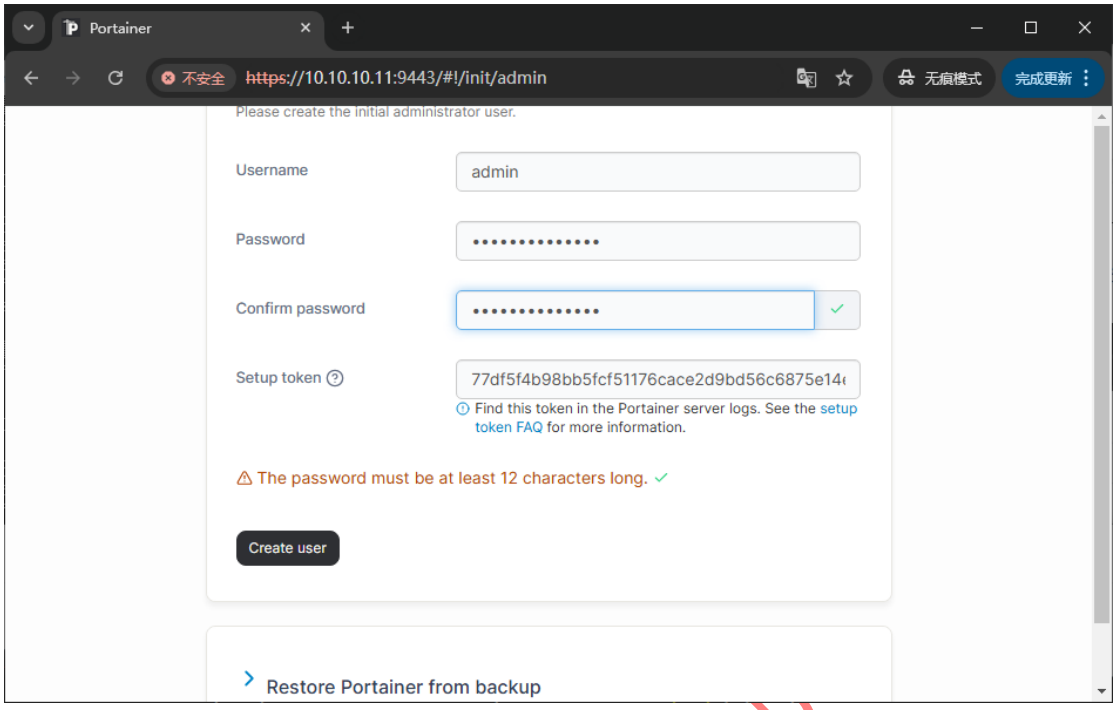
```
[root@master1 ~]# docker volume create portainer_data                                          #创建存储卷

[root@master1 ~]# docker pull 10.8.20.8:5000/portainer/portainer-ce:latest                     #下拉镜像
[root@master1 ~]# docker tag 10.8.20.8:5000/portainer/portainer-ce:latest portainer/portainer-ce:latest      #打标

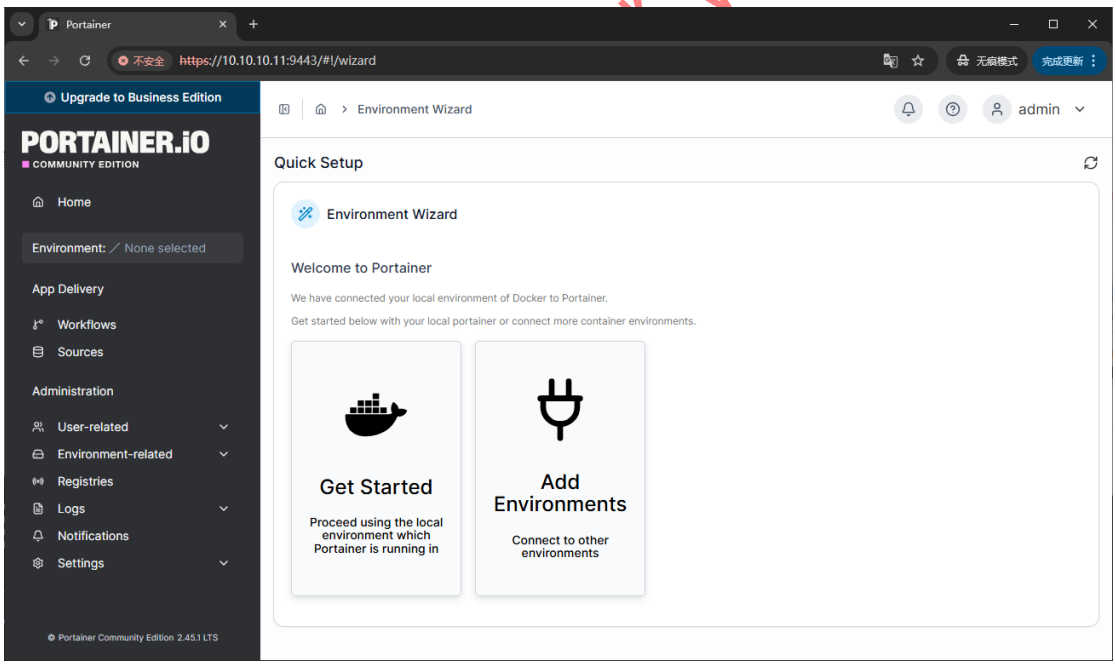
[root@master1 ~]# docker run -d -p 8000:8000 -p 9443:9443 -p 9000:9000 --name portainer --restart=always \
 -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce:latest
[root@master1 ~]# firewall-cmd --add-port={9000,9443}/tcp
[root@master1 ~]# docker logs portainer 2>&1|grep setup_token                          #查看 token，首次登陆时需要用到
setup_token=935773da5c6f56eb629e7f17a82e7df6e529bac2a72063e0f8d7c059ba708b4c
```

接下来就可以用浏览器打开网址 https://<宿主机的 ip>:9443 或者 http://<宿主机的 ip>:9000，第一次访问页面需要给 admin 用户设置密码（至少 12 个字符）：

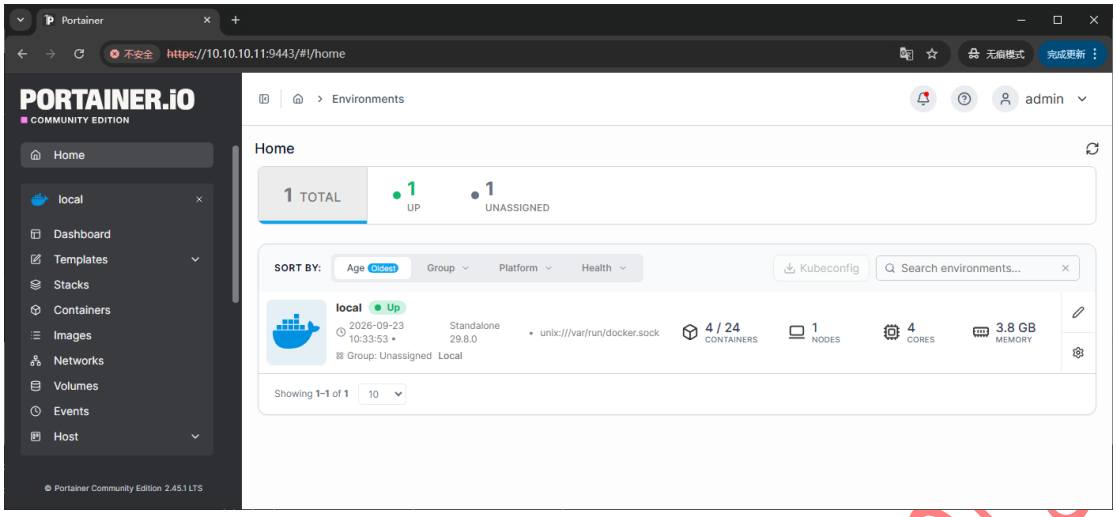
https://www.ityopen.org



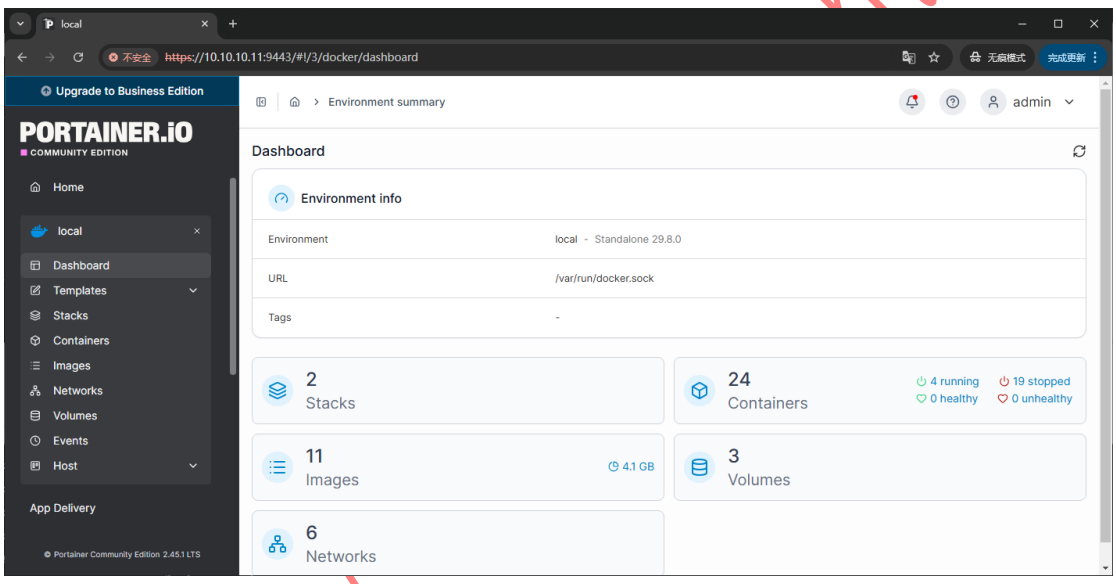
点击“Create user”按钮后下一界面上点击“Continue”按钮即可进入管理后台：



点击上面的“Get Started”按钮：



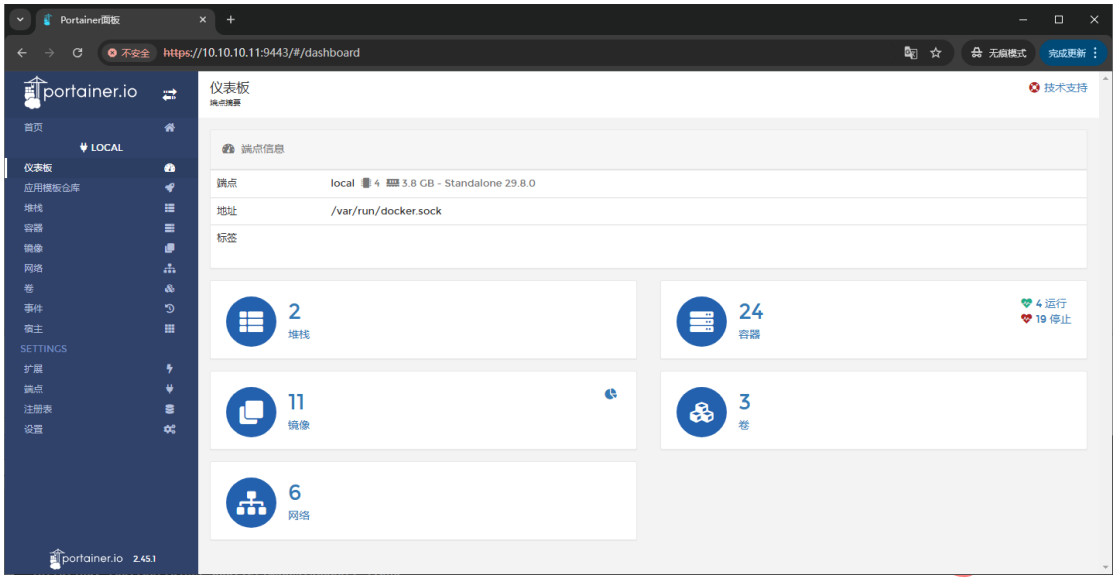
点击上图中的右侧的“local”：



这里就可以管理本机的容器等资源了。由于 portainer 官方默认只支持英语，下面我们通过其他途径来汉化。

```
[root@master1 ~]# wget https://raw.githubusercontent.com/renyinping/portainer-cn/master/Portainer-CN.zip
[root@master1 ~]# dnf install unzip
[root@master1 ~]# unzip Portainer-CN.zip  -d /opt/portainer-public
[root@master1 ~]# docker stop portainer
[root@master1 ~]# docker rm portainer
[root@master1 ~]# docker run -d -p 8000:8000 -p 9443:9443 -p 9000:9000 --name portainer --restart=always -v \
  /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data -v /opt/portainer-public:/public \
  portainer/portainer-ce:latest
```

然后刷新页面并重新登陆后可以看到中文页面了：



不过汉化后发现存在一些小问题，所以建议不要汉化。

§3.2 podman+quadlet 容器



2015 年 Docker 公开了 OCI（开放容器倡议），红帽公司由此在 docker 和 LVC 的基础上开发了 Podman。Podman 是一种无守护进程、开源的 Linux 原生工具，旨在使用符合 OCI 规范的容器和容器映像轻松查找、运行、构建、共享和部署应用程序。Podman 提供了任何使用过 Docker 容器引擎的人都熟悉的命令行界面 (CLI)，大多数用户可以简单地将 Docker 作为 Podman 的别名使用（alias docker=podman）而不会出现任何问题。与其他常见的容器引擎（Docker、CRI-O、containerd）类似，Podman 依赖于符合 OCI 的容器运行时（runc、crun、runv 等）来与操作系统交互并创建正在运行的容器。这使得 Podman 创建的容器与任何其他常见容器引擎创建的容器几乎没有区别。

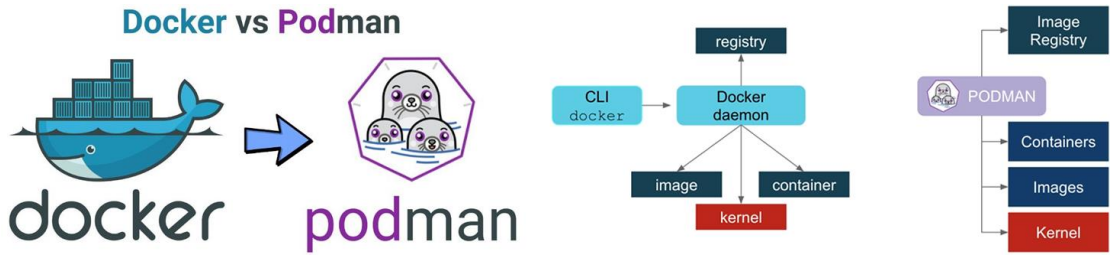


图 3-11 docker 与 podman 比较

与 docker 相比， podman 存在这些优势：

(1) rootless：容器在宿主机上的进程允许以普通用户运行，而经典的 docker 必须以 root 用户运行，不过现在也可以以 rootless 模型运行（执行命令 dockerd-rootless-setuptool.sh install 安装即可）。
(2) Daemonless：Docker 采用了“客户端/服务器”架构，Docker 客户端负责与 Docker 守护进程进行通信，守护进程负责管理容器的创建、运行和销毁。而 Podman 采用了无守护进程的体系结构，每个 Podman 容器都是一个独立的进程，没有守护进程来管理容器。这种架构使得 Podman 更加轻量且适合于多个容器之间的隔离。参考图 3-11。
(3) 与 SELinux 安全系统兼容更好：SELinux 和 podman 都是红帽公司开发的，天生具备更好的兼容性。
(4) 更轻量化：它在资源利用和开发效率上具备独特优势，让我们能够更轻松地管理和运行容器，提高工作效率并节省成本。
(5) 在防火墙策略设置方面更具灵活性。

由于 podman 项目相对来说比较新，所以在生态和社区支持方面还是不如 docker 的，这不过需要更长的时间来裁决。

podman 是 Pod Manager 单词的缩写，意为 pod 管理器，一个 pod 包含若干个容器，这是一个很重要的概念，在 kubernetes 中大量采用 Pod。

§3.2.1 安装

安装前建议先把虚拟机回滚到 docker 实战之前的快照。

1. 安装：

把以前安装的其他的容器版本卸掉：

```
[root@master1 ~]# dnf remove docker docker-engine docker.io podman runc docker-ce
```

首先采用下面的命令安装 podman 容器引擎：

```
[root@master1 ~]# dnf install podman buildah skopeo -y
```

其中 podman 为核心容器引擎，buildah 是镜像构建工具（Podman 底层依赖之一，也可独立使用），skopeo 为镜像检查/复制工具。epel 库中的 podman-compose 是兼容 docker-compose.yml 的工具，这里先不用它。

2. 配置私有镜像源（因为官方源无法访问，国内源又补稳定，所以搭建了私有镜像库）。如果能访问官网镜像库，就不要配置：

```
[root@master1 ~]# vim /etc/containers/registries.conf
```

https://www.veryopen.org

```
#搜索镜像库的顺序:
unqualified-search-registries = ["registry.access.redhat.com", "registry.redhat.io", "docker.io","10.8.20.8:5000"]

[[registry]]
location = "docker.io"

[[registry.mirror]]
location = "10.8.20.8:5000"
insecure = true

[[registry]]
location = "registry.k8s.io"

[[registry.mirror]]
location = "10.8.20.8:5001"
insecure = true

[[registry]]
location = "quay.io"

[[registry.mirror]]
location = "10.8.20.8:5002"
insecure = true

[[registry]]
location = "ghcr.io"

[[registry.mirror]]
location = "10.8.20.8:5003"
insecure = true

short-name-mode = "enforcing"
```

因为我的私有源是 http 协议，所以 insecure=true。修改文件时，建议搜索[[registry]]、location 这些关键字，然后再修改。不过要注意 prefix、insecure 和 location 三个参数都在[[registry]]节下。有了 prefix="10.8.20.8:5000" 后，以后使用镜像时就可以省略前缀，比如 podman pull 10.8.20.8:5000/busybox 可以简化为 podman pull busybox。

使用命令 podman 来管理容器和容器镜像，这个命令的语法是：

```
podman [参数] 子命令
```

一般采用默认参数即可，子命令很多，表 3-1 列出了大部分常用的子命令。

表 3-1 podman 的子命令

子命令	说明
attach	连接到一个正在运行的容器
auto-update	根据容器本身的升级策略升级容器
build	使用 Containerfile 文件构建容器镜像
commit	从一个已经改变过的运行中的容器创建容器镜像
container	管理容器
cp	在容器和本地文件系统之间拷贝文件
create	创建新的容器
diff	检查容器或者镜像的文件系统的差异
events	监视 Podman 事件
exec	在运行的容器中执行命令
export	把容器的文件系统输出为一个 tar 归档文件
healthcheck	对容器健康检查
history	显示一个镜像的历史
image	管理镜像
images	列出本地的镜像
init	初始一个或多个容器
inspect	显示配置信息
kill	杀死一个或多个容器中的主进程
load	从 tar 归档恢复到容器，与 export 子命令作用相反

https://www.veryyoe

login	登录到容器
logout	从容器登出
logs	显示一个或多个容器的日志
mount	挂载容器的根目录
network	管理 Podman 网络
pause	暂停一个或多个容器
pod	管理 pod（容器组）
port	列出容器的端口
ps	列出容器的信息
pull	下载容器镜像
push	上传容器镜像
rename	修改容器名
restart	重启一个或多个容器
rm	删除一个或多个容器
rmi	删除一个或多个本地镜像
run	在一个新容器中运行命令
search	搜索镜像
start	启动一个或多个容器
stats	统计容器中的资源使用情况
stop	关闭一个或多个容器
top	显示容器中的进程
unmount	卸载工作容器的根目录，与 mount 子命令作用相反
unpause	为 pause 子命令的反方向操作
wait	等待一个或多个容器的关机动作，并显示退出状态代码

这些子命令都可以采用 man podman-*的形式获取帮助，比如 man podman-search，也可以类似这样 podman pull --help，还可以获取 podman 的全盘帮助，如 man podman。

https://www.verypen.org

§3.2.2 实战

实验一：用 Podman 跑一个网站容器

1. 直接运行一个 Nginx 网站：

```
[root@master1 ~]# podman run -d --name my-nginx -p 8080:80 docker.io/library/nginx:latest
```

参数说明（和 Docker 完全一致）： -d：后台运行 --name：容器命名 -p 8080:80：宿主机 8080 端口映射到容器内 80 端口。镜像地址建议写全 docker.io/library/nginx，避免 Podman 因为未配置默认 registry 而报“是否要从多个源里选一个”的交互提示。

验证：

```
[root@master1 ~]# podman ps
[root@master1 ~]# curl http://localhost:8080
```

浏览器访问 http://<虚拟机 IP>:8080，应该能看到 Nginx 默认欢迎页。

2. 挂载自定义网页内容

在宿主机准备一个网页文件：

```
[root@master1 ~]# mkdir -p ~/mysite
[root@master1 ~]# cat > ~/mysite/index.html << 'EOF'
<html><body><h1>Hello from Podman on Rocky Linux 9</h1></body></html>
EOF
```

停掉旧容器，重新用挂载卷的方式运行：

```
[root@master1 ~]# podman stop my-nginx && podman rm my-nginx

[root@master1 ~]# podman run -d --name my-nginx -p 8080:80 -v ~/mysite:/usr/share/nginx/html \
  docker.io/library/nginx:latest
```

刷新浏览器，应该看到自定义的 “Hello from Podman” 页面。

3. 常用管理命令

```
podman ps -a                # 查看所有容器（含已停止的）
podman logs my-nginx        # 查看日志
podman exec -it my-nginx bash  # 进入容器内部
podman stop my-nginx        # 停止
podman start my-nginx       # 启动
podman rm my-nginx          # 删除容器
podman images               # 查看本地镜像
```

可以看到，命令几乎和 Docker 一字不差，alias docker=podman 在很多场景下是可以直接用的。

实验二：用 Quadlet 把容器变成 systemd 服务

1. 为什么需要 Quadlet

实验一里的容器，一旦虚拟机重启就没了，需要手动重新 podman run。生产环境里我们希望容器：1)开机自动启动；2)异常退出后自动重启；3)能用 systemctl status/start/stop 统一管理，和管理其他系统服务的方式一致。

早期做法是 podman generate systemd 生成 unit 文件，但这种方式生成的文件是“死”的，Podman 升级后需要手动重新生成。Quadlet 是 Podman 4.6+ 内置的更现代的方案：你写一个类似 systemd unit 格式的声明式配置文件（.container），systemd 在启动时通过内置的生成器自动转换成正式的 service，且会跟随 Podman 版本自动更新生成逻辑，不需要你手动维护。

2. Quadlet 文件的存放位置

模式	目录	管理命令前缀
系统级（root）	/etc/containers/systemd/	systemctl
用户级（rootless）	~/.config/containers/systemd/	systemctl --user

课堂实验统一采用系统级（root），因为用户级模式还涉及 loginctl enable-linger 才能在用户未登录时保持服务运行，对初学者来说多一层不必要的复杂度。生产环境中如果追求更高安全性，再引导学生了解用户级模式。

3. 第一个 Quadlet 实验：Nginx 网站服务化

先清理实验一遗留的容器（Quadlet 会接管容器生命周期，避免手动启动的容器和 Quadlet 冲突）：

```
[root@master1 ~]# podman stop my-nginx && podman rm my-nginx
```

创建系统级 Quadlet 目录并编写配置文件：

```
[root@master1 ~]# mkdir -p /etc/containers/systemd

[root@master1 ~]# tee /etc/containers/systemd/mysite.container << 'EOF'
[Unit]
Description=My Nginx Website
After=network-online.target
```

https://www.veryopen.org

```
[Container]
Image=docker.io/library/nginx:latest
ContainerName=mysite
PublishPort=8080:80
Volume=/home/%u/mysite:/usr/share/nginx/html

[Service]
Restart=always

[Install]
WantedBy=multi-user.target
EOF
```

注意：Volume= 这一行路径要写绝对路径，%u 这种 systemd specifier 在系统级 unit 里不一定按预期展开，教学时建议直接写死实际路径，比如 /home/student/mysite:/usr/share/nginx/html。

让 systemd 重新扫描 Quadlet 目录并生成对应 service：

```
[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl start mysite.service
[root@master1 ~]# systemctl enable mysite.service
```

注意生成的 service 名字是文件名去掉扩展名 + .service，例如：mysite.container → mysite.service。

验证：

```
[root@master1 ~]# systemctl status mysite.service
[root@master1 ~]# curl http://localhost:8080
[root@master1 ~]# podman ps                                    # 会看到 systemd 拉起来的容器，名字是 mysite
```

试一试重启虚拟机（或者 systemctl stop podman 模拟异常场景后手动 kill 容器进程），验证服务能否自动恢复：

```
[root@master1 ~]# podman kill mysite
[root@master1 ~]# sleep 3
[root@master1 ~]# podman ps                                    # 应该能看到 systemd 已经自动把它重新拉起来了
```

这就是 Quadlet 相比手动 podman run 的核心价值——容器生命周期完全交给 systemd 管理，行为和管理任何其他 Linux 系统服务完全一致。

4. Quadlet 常用管理命令

```
[root@master1 ~]# systemctl status mysite.service
[root@master1 ~]# systemctl stop mysite.service
[root@master1 ~]# systemctl restart mysite.service
[root@master1 ~]# systemctl disable mysite.service
[root@master1 ~]# journalctl -u mysite.service -f              # 查看容器日志（等价于 podman logs -f）
```

注意：每次修改了 .container 文件之后，必须重新 daemon-reload 才会生效：

```
[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl restart mysite.service
```

实验三：多容器协作——WordPress + MySQL

这是本次实验的核心案例：WordPress 需要两个容器协同工作——一个跑 MySQL（数据存储），一个跑 WordPress 本体（Web 服务），两者需要能够互相通信。

1. 两种容器协作方式的区别

方式	原理	Quadlet 对应文件
共享网络（Network）	两个独立容器，各自有自己的网络命名空间，通过 Podman 自建的虚拟网络+内置 DNS 互相访问	.network + 多个 .container
共享 Pod（命名空间）	多个容器共享同一个网络命名空间（类似 K8s 里的 Pod 概念），容器间用 localhost 直接通信	.pod + 多个 .container

先做“共享网络”方式（更接近 docker-compose 的思路，也是目前更常见的 Quadlet 实践），再作为拓展介绍“共享 Pod”方式（直接对应 K8s 的 Pod 概念，方便后续过渡到 K8s 章节）。

https://www.devopsroad.org

2. 方式一：共享 Network 实现 WordPress + MySQL

第一步：创建 Quadlet 网络单元

```
[root@master1 ~]# tee /etc/containers/systemd/wp-net.network << 'EOF'
[Network]
NetworkName=wp-net
EOF
```

第二步：创建 MySQL 容器单元

```
[root@master1 ~]# mkdir -p /opt/wordpress/mysql-data

[root@master1 ~]# tee /etc/containers/systemd/wp-db.container << 'EOF'
[Unit]
Description=WordPress MySQL Database
After=network-online.target wp-net.service

[Container]
Image=docker.io/library/mysql:8.0
ContainerName=wp-db
Network=wp-net.network
Volume=/opt/wordpress/mysql-data:/var/lib/mysql:Z
Environment=MYSQL_ROOT_PASSWORD=RootPass123
Environment=MYSQL_DATABASE=wordpress
Environment=MYSQL_USER=wpuser
Environment=MYSQL_PASSWORD=WpPass123

[Service]
Restart=always

[Install]
WantedBy=multi-user.target
EOF
```

第三步：创建 WordPress 容器单元

关键点：WordPress 容器通过 WORDPRESS_DB_HOST=wp-db 连接数据库——容器名就是网络内的 DNS 名称，这是 Podman 自建网络自带的服务发现能力，不需要手动配置 IP。

```
[root@master1 ~]# mkdir -p /opt/wordpress/wp-data

[root@master1 ~]# tee /etc/containers/systemd/wp-web.container << 'EOF'
[Unit]
Description=WordPress Web
After=network-online.target wp-db.service
Requires=wp-db.service

[Container]
Image=docker.io/library/wordpress:latest
ContainerName=wp-web
Network=wp-net.network
PublishPort=8081:80
Volume=/opt/wordpress/wp-data:/var/www/html:Z
Environment=WORDPRESS_DB_HOST=wp-db
Environment=WORDPRESS_DB_NAME=wordpress
Environment=WORDPRESS_DB_USER=wpuser
Environment=WORDPRESS_DB_PASSWORD=WpPass123

[Service]
Restart=always

[Install]
WantedBy=multi-user.target
EOF
```

Requires=wp-db.service + After=wp-db.service 保证了启动顺序——systemd 会先拉起数据库，再启动 WordPress，避免 WordPress 启动时数据库还没就绪导致连接失败。

第四步：加载并启动

```
[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl start wp-db.service
[root@master1 ~]# systemctl start wp-web.service
[root@master1 ~]# systemctl enable wp-db.service wp-web.service
```

第五步：验证

```
[root@master1 ~]# podman ps                                    # 应该看到 wp-db 和 wp-web 两个容器都在运行

[root@master1 ~]# podman network inspect wp-net                # 查看两个容器是否都在这个网络里
```

浏览器访问 http://<虚拟机 IP>:8081，应该能看到 WordPress 的安装向导页面，跟着向导走完初始化，能成功建站说明数据库连接是通的。

排错提示：

如果 WordPress 页面报数据库连接错误：

```
[root@master1 ~]# journalctl -u wp-db.service -n 50                                  # 看数据库有没有正常初始化完成
[root@master1 ~]# journalctl -u wp-web.service -n 50                                  # 看 WordPress 报的具体错误
[root@master1 ~]# podman exec -it wp-db mysql -uwpuser -pWpPass123 -e "SHOW DATABASES;" #手动验证数据库账号密码是否正确
```

常见原因：MySQL 容器首次初始化数据库需要一点时间（尤其虚拟机资源有限时），WordPress 容器如果启动过快可能连接失败，可以稍等几十秒后 systemctl restart wp-web.service 重试。

3. 方式二（拓展）：共享 Pod 实现 WordPress + MySQL

Pod 是把多个容器放进同一个网络命名空间，容器之间直接用 localhost 通信，这和 Kubernetes 里 Pod 的概念完全一致——这也是为什么学完这部分，理解 K8s 的 Pod 会容易很多。

第一步：创建 Pod 单元

```
[root@master1 ~]# tee /etc/containers/systemd/wp-pod.pod << 'EOF'
[Pod]
PodName=wp-pod
PublishPort=8082:80
EOF
```

第二步：数据库容器加入这个 Pod

```
[root@master1 ~]# tee /etc/containers/systemd/wp-pod-db.container << 'EOF'
[Unit]
Description=WordPress DB (in Pod)
After=network-online.target

[Container]
Image=docker.io/library/mysql:8.0
ContainerName=wp-pod-db
Pod=wp-pod.pod
Volume=/opt/wordpress/mysql-data-pod:/var/lib/mysql:Z
Environment=MYSQL_ROOT_PASSWORD=RootPass123
Environment=MYSQL_DATABASE=wordpress
Environment=MYSQL_USER=wpuser
Environment=MYSQL_PASSWORD=WpPass123

[Service]
Restart=always

[Install]
WantedBy=multi-user.target
EOF
```

第三步：WordPress 容器也加入同一个 Pod

因为共享网络命名空间，这里数据库地址直接写 127.0.0.1，而不是容器名：

```
[root@master1 ~]# tee /etc/containers/systemd/wp-pod-web.container << 'EOF'
[Unit]
Description=WordPress Web (in Pod)
After=wp-pod-db.service
Requires=wp-pod-db.service

[Container]
Image=docker.io/library/wordpress:latest
ContainerName=wp-pod-web
Pod=wp-pod.pod
Volume=/opt/wordpress/wp-data-pod:/var/www/html:Z
Environment=WORDPRESS_DB_HOST=127.0.0.1
Environment=WORDPRESS_DB_NAME=wordpress
Environment=WORDPRESS_DB_USER=wpuser
Environment=WORDPRESS_DB_PASSWORD=WpPass123

[Service]
Restart=always

[Install]
```

```
WantedBy=multi-user.target
EOF
```

注意端口映射写在 Pod 单元里（wp-pod.pod 的 PublishPort），而不是某个具体容器里——因为端口本质上是 Pod 共享网络命名空间对外暴露的，这一点和 K8s 里”Service 挂在 Pod 上而不是某个 container 上”的逻辑是一致的。

第四步：启动验证

```
[root@master1 ~]# mkdir -p /opt/wordpress/mysql-data-pod /opt/wordpress/wp-data-pod
[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl start wp-pod-db.service
[root@master1 ~]# systemctl start wp-pod-web.service

[root@master1 ~]# podman pod ps                                                    # 查看 Pod
[root@master1 ~]# podman ps --pod                                                  # 查看容器及其所属 Pod
```

浏览器访问 http://<虚拟机 IP>:8082，同样应该看到 WordPress 安装页面。

§3.2.3 可视化管理工具

也可用红帽或 Rocky 图形页面来管理容器，不过先要安装它：

```
[root@master1 ~]# dnf install cockpit\* -y

[root@master1 ~]# systemctl --now enable cockpit.socket
Created symlink /etc/systemd/system/sockets.target.wants/cockpit.socket → /usr/lib/systemd/system/cockpit.socket.

[root@master1 ~]# ss -tnlp
State  Recv-Q Send-Q Local Address:Port   Peer Address:Port Process
LISTEN 0      4096         0.0.0.0:8081        0.0.0.0:*     users:(("conmon",pid=5462,fd=5))
LISTEN 0      128          0.0.0.0:22          0.0.0.0:*     users:(("sshd",pid=7004,fd=3))
LISTEN 0      4096         0.0.0.0:111         0.0.0.0:*     users:(("systemd",pid=1,fd=51))
LISTEN 0      4096       127.0.0.1:39181       0.0.0.0:*     users:(("containerd",pid=844,fd=13))
LISTEN 0      128             [::]:22             [::]:*     users:(("sshd",pid=7004,fd=4))
LISTEN 0      4096            [::]:111            [::]:*     users:(("systemd",pid=1,fd=53))
LISTEN 0      4096               *:9090              *:*     users:(("systemd",pid=1,fd=56))
```

其中软件包 cockpit-podman 和 cockpit-machines 分别对应页面上的容器和虚拟机超链接，也可以单独安装它们。默认不能从其他电脑采用 root 登录的，我们做如下修改（注释掉 root 那行，或者干脆删掉）：

```
[root@master1 ~]# vim /etc/cockpit/disallowed-users
# List of users which are not allowed to login to Cockpit
#root
```

这样可以从任何电脑以 root 登录图形管理界面了。

然后用浏览器打开 https://10.10.10.11:9090，输入 root 和它的密码即可登录，参考图 3-12 所示。

https://www.oben.org

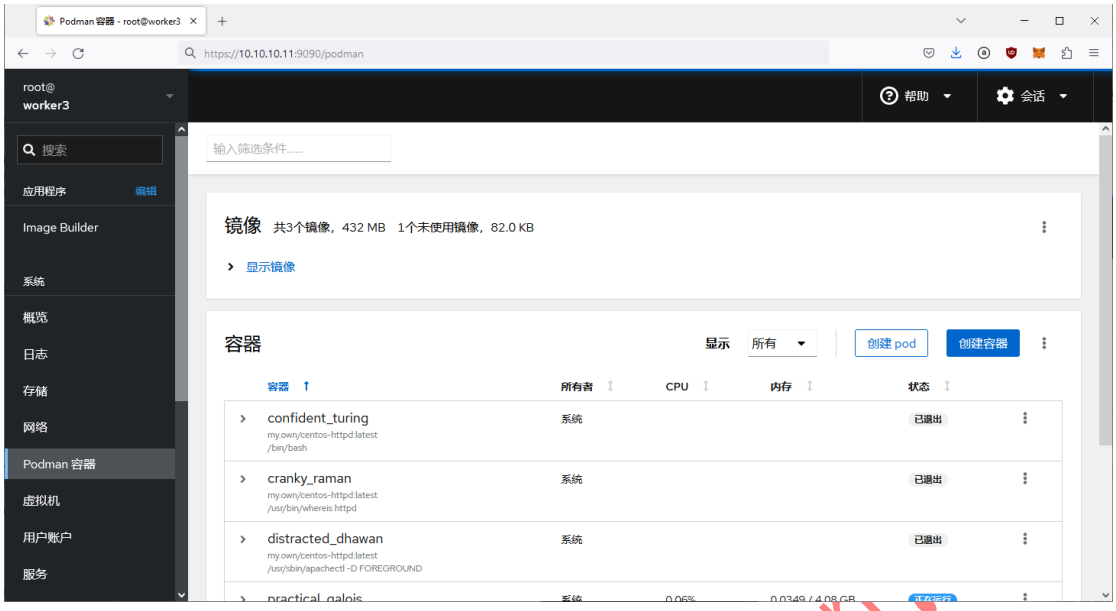


图 3-12 可视化管理容器

§3.3 containerd+nerdctl 容器

containerd 是后台容器引擎，nerdctl 的前台管理工具。K8s 现在默认采用 containerd。

§3.3.1 安装环境

浏览 https://github.com/containerd/nerdctl/releases/latest，下载最新本版 nerdctl-full 官方发布包——这是一个“全家桶”压缩包，把 containerd、runc、CNI 插件、BuildKit、RootlessKit 全部打包在一起，解压即用，特别适合教学环境快速搭建、避免各组件版本搭配不兼容的问题。

```
[root@master1 ~]# wget https://github.com/containerd/nerdctl/releases/download/v2.4.0/nerdctl-full-2.4.0-linux-amd64.tar.gz

[root@master1 ~]# tar -C /usr/local -xzf nerdctl-full-2.4.0-linux-amd64.tar.gz
```

这个 full 包解压后会把二进制放到 /usr/local/bin，systemd 单元文件放到 /usr/local/lib/systemd/system/，CNI 插件放到 /usr/local/libexec/cni，在线帮助文件放在 /usr/local/share/doc/nerdctl/。

```
[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl enable --now containerd
[root@master1 ~]# systemctl enable --now buildkit

[root@master1 ~]# systemctl status containerd

[root@master1 ~]# nerdctl version
Client:
 Version:       v2.4.0
 OS/Arch:       linux/amd64
 Git commit:    347782c28efe1753d5cc9652f6304752ed431221
 buildctl:
```

https://www.veryopen.

```
  Version:      v0.31.2
  GitCommit:    e42e1bfd389af7203238cce77b1f7dad447285e9

Server:
 containerd:
  Version:      v2.3.3
  GitCommit:    aad11006b869517fcd3009450b6f82da282e1a9b
 runc:
  Version:      1.5.1
  GitCommit:    v1.5.1-0-g8f2685a

[root@master1 ~]# nerdctl info
```

配置本地镜像仓库加速请参考“§14.3　安装容器运行时”小节。

§3.3.2　containerd 名字空间

名字空间类似硬盘上的目录，用于隔离各种资源对象，containerd 默认存在三个名字空间：default（nerdctl 默认使用这个命名空间）、moby（Docker 使用的命名空间，这就是为什么 docker images 看不到别的命名空间的内容）、k8s.io（kubelet/kubeadm/crictl 使用的命名空间）和 buildkit。

做实验验证一下：

```
# nerdctl 默认在 default 命名空间操作
[root@master1 ~]# nerdctl images

# 明确指定查看 k8s.io 命名空间（如果这台机器上也跑过 K8s 实验）
[root@master1 ~]# nerdctl -n k8s.io images

# 下拉镜像 busybox:
[root@master1 ~]# nerdctl pull busybox

# 列出 containerd 当前有哪些命名空间
[root@master1 ~]# nerdctl namespace list
buildkit    0             0         0
default     0             1         0
k8s.io      0             0         0
moby        0             0         0
```

§3.3.3　跑应用

实验一：用 nerdctl 跑一个网站容器

```
[root@master1 ~]# nerdctl run -d --name my-nginx -p 8080:80 nginx:latest

[root@master1 ~]# nerdctl ps
[root@master1 ~]# curl http://localhost:8080
```

命令风格和 Docker 几乎一模一样，这是 nerdctl 的设计目标——“给 containerd 套一层 Docker 兼容的用户体验”。nerdctl 的其他用法：nerdctl ps -a（查看容器）、nerdctl logs -f my-nginx（查看日志）、nerdctl exec -it my-nginx sh（进入容器）、nerdctl stop/rm my-nginx（停止/删除容器）等。

修改页面：

```
[root@master1 ~]# mkdir -p ~/mysite
[root@master1 ~]# echo "<h1>Hello from nerdctl on Rocky Linux 9</h1>" > ~/mysite/index.html

[root@master1 ~]# nerdctl rm -f my-nginx

[root@master1 ~]# nerdctl run -d --name my-nginx \
  -p 8080:80 \
  -v ~/mysite:/usr/share/nginx/html \
  nginx:latest

[root@master1 ~]# curl http://localhost:8080
```

实验二：从镜像构建（nerdctl + BuildKit）

nerdctl 内置对接了 BuildKit（我们前面启动 buildkit.service 就是为这个准备的），可以直接执行 nerdctl build，用法和 docker build 一致：

```
[root@master1 ~]# mkdir -p ~/nerdctl-demo && cd ~/nerdctl-demo

[root@master1 nerdctl-demo]# vim Dockerfile
FROM nginx:latest
RUN echo "<h1>Built with nerdctl + BuildKit</h1>" > /usr/share/nginx/html/index.html

[root@master1 nerdctl-demo]# nerdctl build -t my-custom-nginx:v1 .
[root@master1 nerdctl-demo]# nerdctl run -d --name custom-nginx -p 8081:80 my-custom-nginx:v1
[root@master1 nerdctl-demo]# curl http://localhost:8081
```

这一步是 Podman 实验里没有专门涉及的内容，Docker 的构建能力内置在 dockerd 服务里，Podman 用独立的 buildah 做构建，而 nerdctl 是通过对接独立的 BuildKit 服务来做构建的。

实验三：多容器协作——nerdctl compose 跑 WordPress + MySQL

nerdctl 原生支持 docker-compose.yml 语法，这一点和 Podman 需要额外装 podman-compose 不同，nerdctl compose 是内置子命令，不用装额外工具。

方式	工具	配置文件格式
Docker	docker compose	docker-compose.yml
Podman	Quadlet	.container/.network/.pod（systemd 风格）
containerd	nerdctl	compose docker-compose.yml（兼容 Docker 语法，内置支持）

编写 compose 文件

```
[root@master1 ~]# mkdir -p ~/wordpress-nerdctl && cd ~/wordpress-nerdctl

[root@master1 wordpress-nerdctl]# vim docker-compose.yml
services:
  db:
    image: mysql:8.0
    container_name: wp-db
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: RootPass123
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wpuser
      MYSQL_PASSWORD: WpPass123
    volumes:
      - ./mysql-data:/var/lib/mysql

  wordpress:
    image: wordpress:latest
    container_name: wp-web
    restart: always
    depends_on:
      - db
    ports:
      - "8082:80"
    environment:
      WORDPRESS_DB_HOST: db
      WORDPRESS_DB_NAME: wordpress
      WORDPRESS_DB_USER: wpuser
      WORDPRESS_DB_PASSWORD: WpPass123
    volumes:
      - ./wp-data:/var/www/html
```

关键点：WORDPRESS_DB_HOST: db ——这里的 db 是 compose 文件里定义的服务名，nerdctl compose 会自动给这组容器创建一个专属网络，并提供基于服务名的 DNS 解析，这和 Docker Compose、以及之前 Podman 那个“共享 Network”方案的服务发现原理是一致的。

```
[root@master1 wordpress-nerdctl]# nerdctl compose up -d
[root@master1 wordpress-nerdctl]# nerdctl compose ps
[root@master1 wordpress-nerdctl]# nerdctl compose logs -f wordpress
```

浏览器访问 http://10.10.10.11:8082，应该能看到 WordPress 安装向导页面。

```
[root@master1 wordpress-nerdctl]# nerdctl compose stop
[root@master1 wordpress-nerdctl]# nerdctl compose start
```

verypen.org

```
[root@master1 wordpress-nerdctl]# nerdctl compose down                    # 停止并删除容器（不删数据卷）
[root@master1 wordpress-nerdctl]# nerdctl compose down -v                 # 连数据卷一起删除，彻底清空
```

（可选拓展）让容器开机自启——手动包一层 systemd

nerdctl 本身没有像 Podman Quadlet 那样的原生 systemd 集成方案，如果想要类似效果，需要手工写 systemd unit：

```
[root@master1 ~]# vim /etc/systemd/system/nerdctl-nginx.service
[Unit]
Description=Nerdctl managed Nginx
After=containerd.service
Requires=containerd.service

[Service]
ExecStartPre=-/usr/local/bin/nerdctl rm -f my-nginx
ExecStart=/usr/local/bin/nerdctl run --name my-nginx -p 8080:80 nginx:latest
ExecStop=/usr/local/bin/nerdctl stop my-nginx
Restart=always

[Install]
WantedBy=multi-user.target

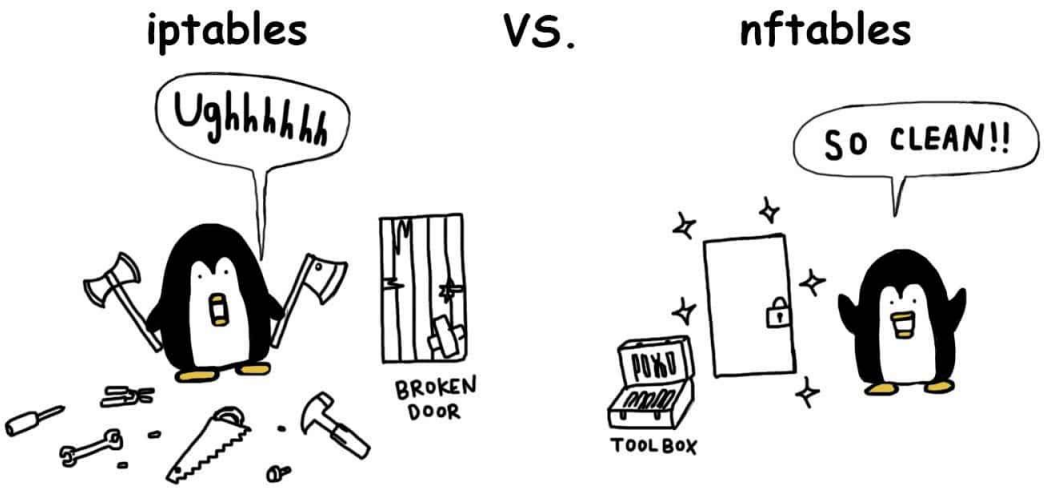
[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl enable --now nerdctl-nginx.service
```

https://www.veryopen.

§3.4 拓展学习

1. overlay 文件系统。
2. docker 命令的用法。
3. 熟悉 cockpit 可视化管理页面。

第4章 防火墙



§4.1 基础知识

绝大多数防火墙工作在网络层，其主要任务就是设“关卡”来审查流经防火墙的数据包，将原来完全互通的网络分割成不同的区域（zone）——内网区、外网区、公共区、DMZ 区、信任区等等。通过在防火墙内设置一系列的审查规则来判断哪些数据包允许通过，哪些数据包必须拦截，哪些数据包经过修改后放行，等等。

一个服务采用“ip 地址:端口”来唯一指定，比如“192.168.0.10:22”定义了 ip 地址为 192.168.0.10 这台计算机上的 ssh 服务（此服务监听端口 22）。另一个重要的概念是建立 tcp 链接需要三次握手，第一次握手时首先“伸手”的一方称为发起方，一般为客户端，比如 sshd 服务启动后监听 22 号端口，被动地等待客户端发起握手以便建立登录连接。图 4-1 描述了 tcp/ip 的通讯过程。

https://www.very

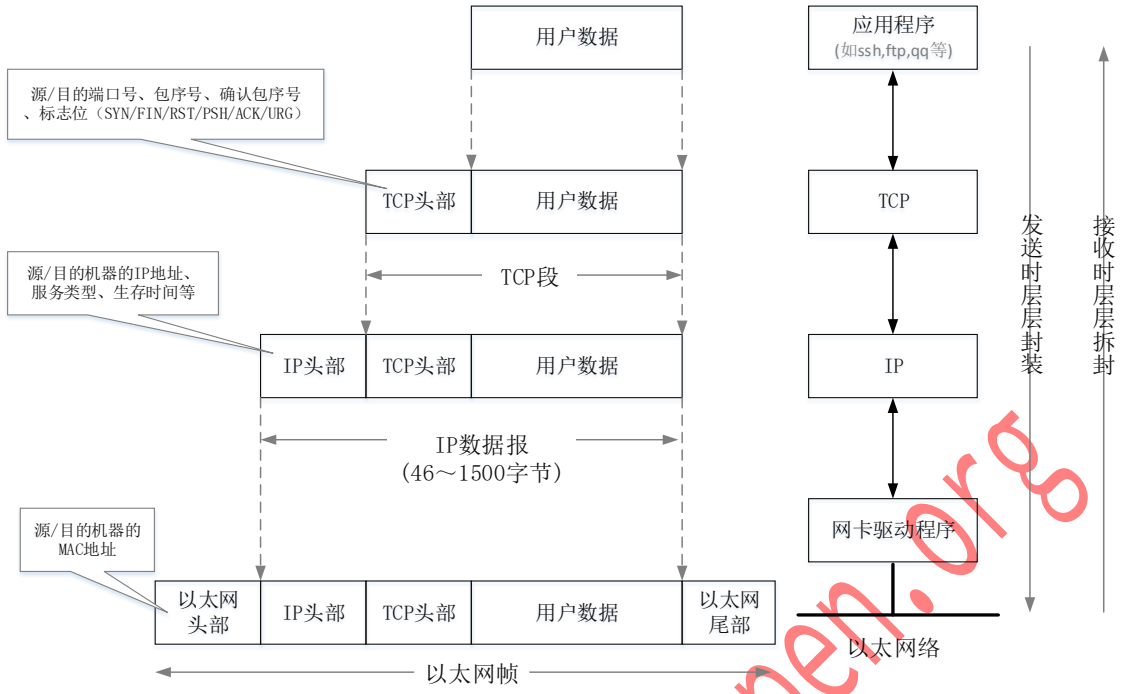


图 4-1 tcp/ip 通讯示意图

发送数据时从上（应用程序）到下（以太网络）层层封装，相反，接收数据就从下到上层层拆封。我们编写的网络应用程序中，往往存在如下所示的发送数据的语句：

```
write(socket_fd,"嗨，我是马列毛！\n",17)
```

其中“嗨，我是马列毛！\n”是用户数据，层层封装的操作由 write()函数完成，应用程序开发人员不用管它。再来看看接收数据的语句：

```
iBytes = read(accept_fd, buf, 256);
```

用户数据最终被放在 buf 变量里，而层层拆封的操作也不用应用程序开发人员理会，由 read()函数自动完成。

Linux 的 netfilter 防火墙模块工作在网卡驱动程序的上面，所以能看到 IP 数据报，报文里面的 IP 头部和 TCP 头部包含了重要的可用于制定审查规则的信息，如源/目的端口号、源/目的机器的 IP 地址、包序号、标志位等等。

https://www.veryopen.org

§4.2 内核模块 netfilter

在 Linux 系统中，tcp/ip 协议栈、netfilter nft 子系统、nf_tables 模块和 firewalld 的关系见图 4-2 所示。

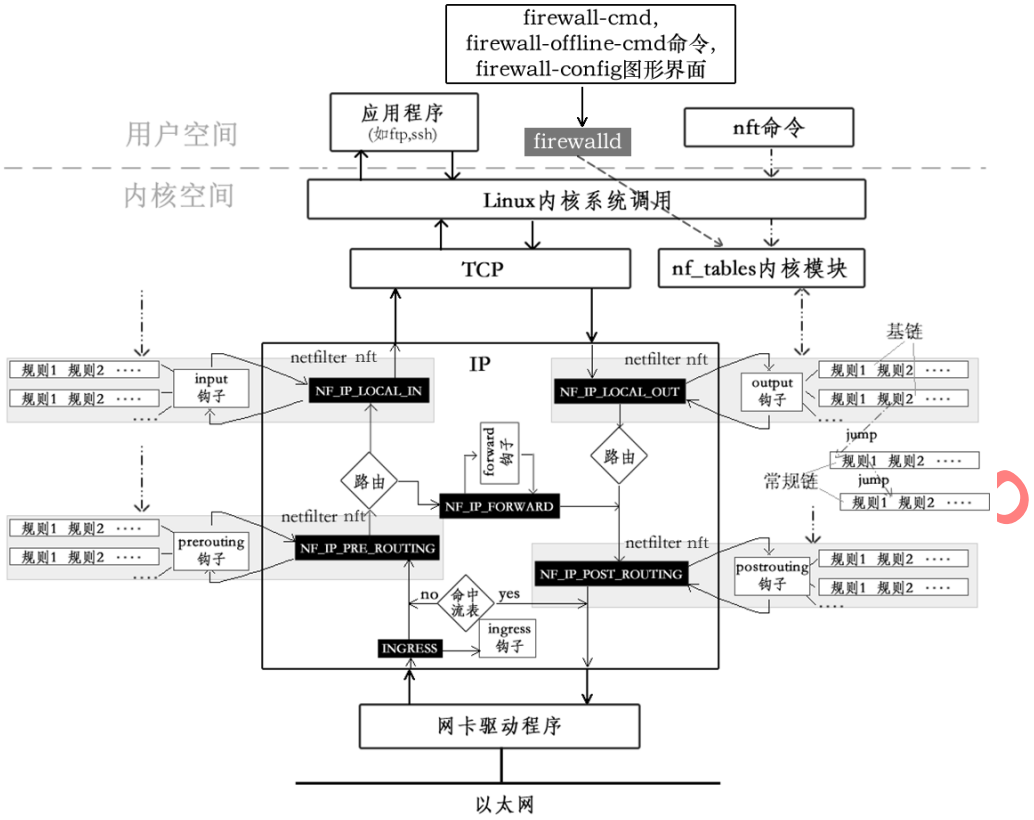


图 4-2 tcp/ip、netfilter 和 nf_tables 的关系

现代 Linux 操作系统的防火墙默认后端是 nftables，而不再使用 iptables 了，当然用户可以自行选择后端。nftables 比 iptables 相比具有明显的优势，比如速度就快很多。Netfilter 是 Linux 2.4.x 内核开始引入的一个通用的、抽象的框架，它提供一整套的钩子函数（hook）的管理机制，使得诸如数据包过滤、网络地址转换（NAT）和基于协议类型的连接跟踪的用户编程变得简化。Netfilter 在 TCP/IP 协议栈的 IP 层中的数据包路径上设置若干陷阱（如图 4-2 中的 INGRESS、NF_IP_PRE_ROUTING、NF_IP_LOCAL_IN、NF_IP_FORWARD、NF_IP_LOCAL_OUT 和 NF_IP_POST_ROUTING），数据包一旦进入陷阱，就会被预先注册的各个钩子处理，钩子根据防火墙规则决定每个包的命运，要么放行，要么销毁，要么修改后放行，要么旁路等。

由图 4-2 可知，进入电脑的网络数据包会依次通过 INGRESS、NF_IP_PRE_ROUTING 和 NF_IP_LOCAL_IN 三个陷阱，从而流经 ingress、prerouting 和 input 三个钩子；而出去的数据包依次通过 NF_IP_LOCAL_OUT 和 NF_IP_POST_ROUTING 两个陷阱，流经 output 和 postrouting 两个钩子；转发的数据包要通过 INGRESS、NF_IP_PRE_ROUTING、NF_IP_FORWARD 和 NF_IP_POST_ROUTING 四个陷阱，流经 ingress、prerouting、forward 和 postrouting 四个钩子。我们可进一步发现只有进入的数据包才通过 NF_IP_LOCAL_IN，只有出去的数据包才通过 NF_IP_LOCAL_OUT，只有转发的数据包才通过 NF_IP_FORWARD，而其他的陷阱都会影响到多种数据包类型。牢记网络数据包通过这些陷阱的先后次序很重要，否则设计的防火墙规则可能会导致奇怪的结果，尤其当规则多了以后更是如此。

nftables 具体包含 nf_tables 内核模块和 nft 用户工具，nf_tables 模块构建于 netfilter 框架之上，它其实就是一台虚拟机，由 nft 把用户输入的配置内容（源代码）编译成 nf_tables 认识的指令并喂给它，nf_tables 执行这些指令并把结果输出到 netfilter。用户采用 nft 工具编写防火墙规则相当于是编程——一种新的计算机编程语言，不过比 C 语言要简单得多。

firewalld 为防火墙服务，用户使用命令 firewall-cmd、firewall-config 等设置的防火墙规则通过 firewalld 服务与 nf_tables 打交道，所以 firewalld 服务相当于一个中间人。而 nft 命令绕过了这个中间人。

在 Rocky Linux 9，虽然 iptables 命令继续可用，但它是/usr/sbin/xtables-nft-multi 命令的符号连接，

后者还是直接跟 nf_tables 内核模块打交道的，就像 nft 命令一样。新版 Linux 之所以保留 iptables 命令，是为了照顾用户的老习惯，不过建议大家使用 nft 命令。

§4.3 配置规则

在 nftables 中，一张表是一个名字空间，用作名字隔离，用户可以任意创建表，表中可包含链、集合、状态对象和变量等。一个表必须依附于一个具体的地址簇：ip（ipv4）、ip6（ipv6）、inet（ipv4/ipv6）、arp（ipv4 ARP）、bridge、netdev，因此表的唯一性引用就是“地址簇 表名”，比如“ip filter1”、“arp myRaw”。链是规则的容器，有两种类型的链，一种称为基链，另一类称为常规链，基链必须挂在 netfilter 的某个钩子上，一共有 6 个钩子，分别是 prerouting、input、forward、output、postrouting 和 ingress（参见图 4-2），ingress 钩子只对 inet 地址簇有效。基链能看到 TCP/IP 协议栈上的数据包，被挂载在同一个钩子上的多条基链通过各自的优先级来排序处理。基链的优先级为有符号整数（数越小优先级越高），挂在同一个钩子上的多条基链，优先级高的被优先处理（比如优先级为-10 的基链比 20 的基链先处理），但是如果某条基链拒绝或者删除了数据包，则后续的基链不再被处理。而常规链不挂在钩子上，是通过从其他链 jump 或 goto 过来的，一个不被 jump 或 goto 到的常规链没有什么作用，就像一个不被调用的函数那样永远得不到执行。被 jump 到的链需要返回，而 goto 不用返回。这里的规则就是防火墙规则，一条规则由表达式和语句组成，如果一个经过的数据包与表达式匹配成功，那么就对该数据包执行规则中的语句。规则集、地址簇、表、链和规则的关系参见图 4-3。



图 4-3 nftables 中的规则集、地址簇、表、链和规则的关系

假设 netfilter 子系统的 prerouting 钩子上挂了 3 条基链，分别命名为基链 1、基链 2 和基链 3，优先级依次增大，基链 2 的某条规则 jump 到常规链 10。因此我们可以这样描述数据包被 nftables 处理的情景：一个数据包到达 netfilter 子系统的 prerouting 钩子（参见图 4-2），先由基链 1 处理数据包，基链 1 通过后再由基链 2 处理数据包，基链 2 中的某条规则 jump 到常规链 10，因此常规链 10 处理数据包，然后数据包被常规链 10 中某条规则删除了，因此数据包不会到达基链 3。

www.verypdf.org

表对名字的隔离性给我们带来了极大的方便，应用程序可以单独创建自己的表、链和规则，不用考虑是否会影响其他应用，iptables 是没有这这个优势的。不过同一地址簇上的表名不能相同，比用户甲安装 a 程序时自动在 ip 地址簇上创建表 abc，那么用户乙就不能再在 ip 地址簇上创建表 abc。

一条规则的语法如下：

```
[表达式 [ 表达式 表达式 … ]] 操作语句 [ 操作语句 … ]
```

即：如果一个“到达”本规则的网络数据包匹配全部“表达式”，那么该数据包就被执行“操作语句”，可用的语句有：accept（通过）、drop（删除）、jump（调用其他常规链，会返回）、goto（跳转到其他常规链，不会返回）、reject（拒绝）、log（记录日志）、snat（源地址转换）、dnat（目的地址转换）、meta（设置元数据：标记、优先级或跟踪线索）、queue（排队）、dup（复制）、mangle（修改包的头部）、counter（计数）、notrack（不做连接跟踪）等（详见 https://wiki.nftables.org/wiki-nftables/index.php/Main_Page）。

用户可以使用这些命令编写规则：nft、firewall-cmd、firewall-offline-cmd 和 firewall-config 等，其中 nft 命令比较复杂，firewall-config 是图形界面。

1. firewall-cmd 命令

firewall-cmd 命令直接与防火墙服务（即 firewalld）打交道，可以实时改变运行时的防火墙策略，也可以改变永久生效的防火墙策略。firewalld 引入了区域（zone）的概念——可以理解为应用场景，区域中包含若干条防火墙策略，可通过应用不同的区域来达到快速切换防火墙策略的目的。firewall-cmd 命令的参数支持 Tab 键补齐，比如我输入“firewall-cmd --”，然后连续按两次 Tab 键，会显示全部的参数，再比如输入“firewall-cmd --add-”后连续按两次Tab 键，会显示所有以--add-开头的参数。在线文档man firewall-cmd 提供了详细的说明，下面举例说明，以抛砖引玉。

（1）查看防火墙服务是否正在运行：

```
[root@master1 ~]# firewall-cmd --state
```

（2）查看预设的区域有哪些：

```
[root@master1 ~]# firewall-cmd --get-zones
block dmz drop external home internal nm-shared public trusted work
```

预设的区域有 block、dmz、drop、external、home、internal、nm-shared、public、trusted、work 和 workstation，可以根据实际的应用场景选择合适的预设区域，一块网卡每次只能绑定一个区域。这些预设区域的规则参见表 10.4。

表 4-1 预设区域的规则

序号	区域（noze）	默 认 策 略 规 则
1	trusted	允许所有的数据包进出（不设防）
2	home，internal	拒绝进入的流量，除非与出去的流量相关，但允许与 ssh、mdns、ipp-client、amba-client 与 dhcpv6-client 服务相关的数据包进入
3	work	拒绝进入的流量，除非与出去的流量相关，但允许与 ssh、ipp-client 与 dhcpv6-client 服务相关的数据包进入
4	public	拒绝进入的流量，除非与出去的流量相关，但允许与 ssh、dhcpv6-client 服务相关的数据包进入
5	external,dmz	拒绝进入的流量，除非与出去的流量相关；而如果流量与 ssh 服务相关，则允许进入
6	block，drop	拒绝进入的流量，除非与出去的流量相关

可以修改预设区域内的防火墙规则以便更好契合实际的应用场景，当然也可以新增区域。

https://www.verypen.org

（3）查看当前正在使用的区域：

```
[root@master1 ~]# firewall-cmd --get-active-zones
public
  interfaces: ens160
```

表明区域 public 被绑定到网卡 enp160 上，即当前正在使用 pulib 区域。

（4）使用 dmz 区域：

```
[root@master1 ~]# firewall-cmd --zone=public --remove-interface=enp160
[root@master1 ~]# firewall-cmd --zone=dmz --add-interface=enp160
```

（5）查看 home 区域内的防火墙规则：

```
[root@master1 ~]# firewall-cmd --info-zone=home
……
    services: cockpit dhcpv6-client mdns samba-client ssh
……
```

表明允许外面的计算机与这些服务建立链接。

（6）查看默认的区域：

```
[root@master1 ~]# firewall-cmd --get-default-zone
```

（7）设置默认区域为 dmz：

```
[root@master1 ~]# firewall-cmd --set-default-zone=dmz
```

（8）新建区域 caidi：

```
[root@master1 ~]# firewall-cmd --permanent --new-zone=caidi
```

参数--permanent 表示操纵永久生效的防火墙策略，否则就是临时的（重启计算机就没有了）。新建的区域是空的，表名会话只能出不能进。

（9）使用 caidi 区域：

```
[root@master1 ~]# firewall-cmd --reload                                    #加载永久性
的防火墙策略
[root@master1 ~]# firewall-cmd --set-default-zone=caidi                     #使用
caidi 区域
```

可以验证，此时不能从外面登录到本机了。

（10）删除区域 mySecure：

```
[root@master1 ~]# firewall-cmd --permanent --delete-zone=mySecure
```

注意：诸如 home、work 等这种预设的区域不能删除。

（11）查看预设了防火墙策略的服务：

```
[root@master1 ~]# firewall-cmd --get-services
…… cratedb ctdb dhcp dhcpv6 dhcpv6-client distcc dns dns-over-tls docker-registry docker-swarm dropbox-lansync
elasticsearch etcd-client etcd-server finger foreman foreman-proxy freeipa-4 freeipa-ldap freeipa-ldaps freeipa-
replication freeipa-trust ftp galera ganglia-client ganglia-master git gpsd grafana gre high-availability http
http3 https ident imap imaps ipfs ipp ipp-client ipsec irc ircs iscsi-target isns jellyfin jenkins kadmin
kdeconnect kerberos kibana klogin kpasswd ……
```

针对常见的应用服务（如 dhcp、dns、ftp、http、https、postgresql、pop3、nfs、ntp、mysql、rdp、samba、smtp、ssh、tftp 等等），firewalld 预设了策略，我们直接在区域中加入这样的服务即可开通从外面进来的数据包，看下面的例子。

（12）在 caidi 区域中加入 ssh 服务，这样外面就能登录了：

```
[root@master1 ~]# firewall-cmd --zone=caidi --add-service ssh
```

如果想永久生效，就加上--permanent 参数，如果去掉参数--zone=caidi，就是操纵当前正在使用的区域。

（13）在 8 小时内允许外面访问本机的网站：

https://www.verypen.org

```
[root@master1 ~]# firewall-cmd --add-service=http --timeout=8h
```

注意：--permanent 和--timeout 参数不能同时使用。

（14）不允许从外面登录本机：

```
[root@master1 ~]# firewall-cmd --remove-service=ssh
```

（15）打通 443 端口，即允许从外面访问本机的 https 网站：

```
[root@master1 ~]# firewall-cmd --add-port=443/tcp
```

（16）在 work 区域中，开通从 5901 至 5910 的 udp 端口进入本机：

```
[root@master1 ~]# firewall-cmd --permanent --zone=work --add-port=5901-5910/udp
```

（17）drop 区域内允许 icmp 协议流量进入，即允许 ping：

```
[root@master1 ~]# firewall-cmd --zone=drop --add-protocol=icmp
```

（18）work 区域允许来自 192.168.0.21 机器的任何会话：

```
[root@master1 ~]# firewall-cmd --zone=work --add-source=192.168.0.21
```

如果省略--zone 参数，就修改默认区域。还可以定义网段，如--add-source=10.10.2.0/24 表示接受来自 10.10.2.0/24 网络的任何链接。

注意：--add-*、--remove-*和--list-*分别表示增加、删除和显示相关策略的参数，如--add-port、--remove-port 和--list-ports。

（19）将访问本机 888 端口的流量转发到本机 22 端口：

```
[root@master1 ~]# firewall-cmd --add-forward-port=port=888:proto=tcp:toport=22
```

不过要先开通 888 端口（参考例子(15)）。

（20）将访问本机 999 端口的流量转发到 ip 为 192.168.0.21 的主机的 22 端口（需要开启 masquerade）：

```
[root@master1 ~]# firewall-cmd --add-masquerade
[root@master1 ~]# firewall-cmd --add-port=999/tcp
[root@master1 ~]# firewall-cmd --add-forward-port=port=999:proto=tcp:toport=22:toaddr=192.168.0.10
```

（21）每分钟允许 10 个新连接访问 ftp 服务：

```
[root@master1 ~]# firewall-cmd --add-rich-rule="rule service name=ftp limit value=10/m accept"
```

（22）伪装，将来自局域网 192.168.2.0/24 网段访问外网的流量映射为网络出口公网 IP，即修改源 IP 地址（SNAT）：

```
[root@master1 ~]# firewall-cmd --add-masquerade
[root@master1 ~]# firewall-cmd --add-rich-rule="rule family=ipv4 source address=192.168.2.0/24 masquerade"
```

rich rule 称为富规则，使用富规则我们可以精细化定义防火墙规则，详情请参考在线文档 man 5 firewalld.richlanguage。

（23）进入紧急维护模式（中断所有对外的链接，仅本机控制台能登录）：

```
[root@master1 ~]# firewall-cmd --panic-on
```

（24）退出紧急维护模式：

```
[root@master1 ~]# firewall-cmd --panic-off
```

2. nft 命令

https://www.veryopen.org

nft 命令直接操纵内核中的 nf_tables 模块，它有三种用法：一是直接带参数一条一条执行 nft 命令，二是运行“nft -i”进入交互界面编写规则，三是编写规则文件然后运行下面的命令一次性提交（文件的首行必须是#!/usr/sbin/nft -f）。

```
nft -f myFile.nft
```

1). 管理 ruleset

```
nft [-a] {list | flush} ruleset [地址族]
```

参数-a 可与 list 配合使用，显示每个对象的句柄。在 nftables 中，每一个表、链和规则都是对象，每个对象都有一个唯一的 id 号（称为句柄）。

（1）显示系统中的防火墙规则：

```
[root@master1 ~]# nft list ruleset
table inet firewalld {
        chain mangle_PREROUTING {
                type filter hook prerouting priority mangle + 10; policy accept;
                jump mangle_PREROUTING_ZONES
        }

        chain mangle_PREROUTING_POLICIES_pre {
                jump mangle_PRE_policy_allow-host-ipv6
        }

        chain mangle_PREROUTING_ZONES {
                iifname "ens192" goto mangle_PRE_public
                iifname "ens160" goto mangle_PRE_public
                goto mangle_PRE_public
        }

        chain mangle_PREROUTING_POLICIES_post {
        }
        ……
}
```

加上参数-a 就能显示句柄，如 nft -a list ruleset。默认只有一个表“inet firewalld”，表中存在若干条链，即“chain <链名>”，链里的防火墙规则可为 0 个或多个。

（2）清除系统中的防火墙规则：

```
[root@master1 ~]# nft flush ruleset
```

（3）显示 ip 地址族的防火墙规则：

```
[root@master1 ~]# nft list ruleset ip
```

（4）清除 bridge 地址族的防火墙规则：

```
[root@master1 ~]# nft flush ruleset bridge
```

2). 管理表

```
nft {add | create} table [地址族] 表名 [ { flags dormant \;} ]
nft {delete | list | flush} table [地址族] 表名
nft delete table [地址族] handle 表句柄
nft list tables [family]
```

add 与 create 的区别是当表存在时 create 会返回错误，而 add 会修改现存的表。如果省略“地址族”，默认是 ip（即 ipv4）。如果指明了 dormant，表示本表暂时失效。

（1）在 ip 地址族中创建一个表 myFilter：

```
[root@master1 ~]# nft add table myFilter
```

（2）在 arp 地址族中新建表 arp_tb：

```
[root@master1 ~]# nft create table arp arp_tb
```

https://www.v

（3）暂时禁用 ip 地址族中的 myFilter 表（表中的防火墙规则不起作用）：

```
[root@master1 ~]# nft add table myFilter { flags dormant \; }
```

如果要启用之前被禁用的表，可采用类类似命令“nft add table myFilter”即可。

（4）删除 arp 地址族中的表 arp_tb：

```
[root@master1 ~]# nft delete table arp arp_tb
```

（5）清除 ip 地址族中的表 myFilter 中的全部规则：

```
[root@master1 ~]# nft flush table ip myFilter
```

（6）删除 100 号表：

```
[root@master1 ~]# nft delete table handle 100
```

3)．管理链

```
nft {add | create} chain [地址族] 表名 链名 [ { type 类型 hook 钩子 [device 设备] priority 优先级 \; [policy 策略 \;] } ]
nft {delete | list | flush} chain [地址族] 表名 链名
nft delete chain [地址族] 表名 handle 链句柄
nft rename chain [地址族] 表名 链名 新链名
```

add 和 create 的区别参见上面的“2、管理表”。“设备”指网卡名称，如 enp0s3、ens33 等，如果“钩子”为 ingress，那么“设备”不能省略。如果同一个勾子上存在多条链，将根据它们的优先级来处理，代表优先级的数字（可正可负）越小越优先。“策略”只允许定义在基链上，定义本链中的规则如果没有明确定义“接受”或者“拒绝”语句，那么就执行“策略”，“策略”的值可为 accept 或者 drop，如果基链中没有定义策略，默认就是 accept。创建基链时，类型、钩子和优先级不能省略，基链被直接挂接在钩子上，流经的网络数据包首先被基链处理，在上面的第一条语法中，如果省略了大括号中的内容，那么就是创建常规链。地址族、类型和钩子的有效组合见表 4-2。

表 4-2 类型、地址簇和钩子的有效组合

类　型	地址族	钩　子	说　明
filter	全部	全部	filter 类型的链用于过滤数据包。但是 netdev 地址族只能与 filter 和 ingress 组合，arp 地址族只能与 input，output 和 filter 组合
nat	ip，ip6，inet	prerouting，input，output，postrouting	nat 类型的链用于地址转换，只有会话的第一个包会被本链处理
route	ip，ip6	output	如果一个数据包的 ip 头部被修改，那么本类型的链会执行一个新的路由查找。在 nftables 中，route 类型的链通常用于实现策略路由

（1）在 ip myFilter 表中增加一条常规链 myChain：

```
[root@master1 ~]# nft add chain ip myFilter myChain
```

（2）在 ip myFilter 表中增加一条基链 baseChain：

```
[root@master1 ~]# nft add chain ip myFilter baseChain { type filter hook input priority 10 \; policy accept \; }
[root@master1 ~]# nft -a list chain ip myFilter baseChain
table ip myFilter {
        chain baseChain { # handle 2
                type filter hook input priority filter + 10; policy accept;
        }
}
```

链的类型为 filter，放置于 input 钩子上，默认策略是 accept，优先级为 10。

（3）删除 ip myFilter 中的一条常规链 myChain：

```
[root@master1 ~]# nft delete chain ip myFilter myChain
```

（4）清除 ip myFilter 表中的基链 baseChain 中的规则：

```
[root@master1 ~]# nft flush chain ip myFilter baseChain
```

https://www.verypeer.org

（5）删除 ip myFilter 中的基链 baseChain：

```
[root@master1 ~]# nft delete chain ip myFilter handle 2
```

4)．管理规则

```
nft {add | insert} rule [地址族] 表名 链名 [ handle 句柄 | index 索引] statement... [ comment 备注 ]
nft replace rule [地址族] 表名 链名 handle 句柄 statement... [ comment 备注 ]
nft delete rule [地址族] 表名 链名 handle 句柄
```

一条链里的规则的索引从 0 开始，假如有 10 条规则，索引就是从 0 到 9，同一条规则在不同的时刻可能对应不同的索引，但是规则句柄是固定的，所以建议少用索引多使用句柄。add 和 insert 的区别是：前者在链的末尾或者指定的规则后面增加一条规则，后者是在链的开头或者指定的规则前面增加一条规则。statement 是规则的主体部分，包含若干表达式和若干操作语句，当到达的数据包匹配表达式时就执行操作语句。

（1）在 ip myFilter 表里的链 baseChain 的末尾追加一条规则：

```
[root@master1 ~]# nft add rule ip myFilter baseChain ip daddr 192.168.0.0/24 accept
```

（2）在 MyChain 链中的规则句柄 4 后面插入一条规则：

```
[root@master1 ~]# nft add rule myFilter myChain handle 4 ip saddr 127.0.0.9 drop
```

（3）建表 mytable，在表中创建基链 myin，再在链中建规则，以交互式执行 nft 命令：

```
[root@master1 ~]# nft -i
nft> create table inet mytable
nft> add chain inet mytable myin { type filter hook input priority -10; }
nft> add rule inet mytable myin counter
nft> list chain inet mytable myin
table inet mytable {
        chain myin {
                type filter hook input priority filter - 10; policy accept;
                counter packets 14 bytes 1152
        }
}
nft> quit
```

规则的编写非常复杂，里面可以引用变量、集合、状态对象、映射等，更详细的资料请参考官方文档。

§4.4 场景实验

首选采用下面的命令安装防火墙：

```
[root@master1 ~]# dnf install firewalld nftables firewall-config -y
```

同时会安装几个依赖包，诸如 ipset。firewall-config 是图形化的配置工具，可以不安装它。firewalld 主配置文件为/etc/firewalld/firewalld.conf，防火墙策略放在/usr/lib/firewalld/目录中。下面我们以几个应用场景来做实验。

场景 1：个人工作站

https://www.nen.org

对于 Linux 工作站，防火墙的策略一般是：允许出去，只允许外面 ssh 远程登录进来，能访问本机的网站，能 ping 本机。这里给出两种实现方法：

（1）采用 firewall-cmd 命令直接在当前区域中增加规则：

```
[root@master1 ~]# firewall-cmd  --add-service=http --add-service=https --add-service=ssh
success
[root@master1 ~]# firewall-cmd --add-protocol=icmp
success
[root@master1 ~]# firewall-cmd --runtime-to-permanent                                #把运行状态的规则保存到
磁盘
success
```

或者（2）采用 nft 命令，把下面的内容输入到文件 workstation.nft：

```
[root@master1 ~]# vim workstation.nft
#!/usr/sbin/nft -f

flush ruleset
table ip worker_tb {
        chain input_ch {
                type filter hook input priority 0; policy drop;
                tcp dport { ssh, http, https } accept
                ip protocol icmp icmp type { echo-request,echo-reply } accept
        }
}
```

即在 ip 地址簇上创建表 worker_tb，表中有一条基链 input_ch，挂在 input 钩子上，基链的优先级为 0，默认动作是 drop（禁止通行），也即除了链中其他规则放行的网络数据包外，都禁止通行。

然后执行下面的命令导入规则：

```
[root@master1 ~]# nft -f workstation.nft

[root@master1 ~]# nft -a list table ip worker_tb
table ip worker_tb { # handle 10
        chain input_ch { # handle 1
                type filter hook input priority filter; policy drop;
                tcp dport { 22, 80, 443 } accept # handle 4
                icmp type { echo-reply, echo-request } accept # handle 5
        }
}
```

场景 2：Web 服务器

要求：外面能 ssh 登录服务器，能访问 http/https 页面，能 ping 通服务器，服务器出去不受限制，不做转发服务。

```
[root@master1 ~]# vim web-server.nft
#!/usr/sbin/nft -f

flush ruleset

table inet firewall {

    chain inbound_ipv4 {
        # 每秒接受 5 个 ping 包
        icmp type echo-request limit rate 5/second accept
    }

    chain inbound_ipv6 {
        icmpv6 type { nd-neighbor-solicit, nd-router-advert, nd-neighbor-advert } accept
        icmpv6 type echo-request limit rate 5/second accept
    }

    chain inbound {
        type filter hook input priority 0; policy drop;

        # Allow traffic from established and related packets, drop invalid
        ct state vmap { established : accept, related : accept, invalid : drop }
        iifname lo accept
        meta protocol vmap { ip : jump inbound_ipv4, ip6 : jump inbound_ipv6 }
        tcp dport { 22, 80, 443} accept
    }

    chain forward {
        type filter hook forward priority 0; policy drop;
    }
}
```

```
[root@master1 ~]# nft -f web-server.nft
```

场景 3：边界防火墙

如图 10.27 所示的边界防火墙是最流行的布局，我们用一台安装 Linux 和 nftables 的计算机充当防火墙。在边界防火墙上执行 nft -f edge.nft 即可。

图 10.27 边界防火墙

（1）文件 edge.nft 内容：

```
[root@master1 ~]# vim edge.nft
#!/usr/sbin/nft -f

flush ruleset

include "./defines.nft"

table inet filter_tb {
        chain global_ch {
                ct state established,related accept
                ct state invalid drop
                ip protocol icmp accept
                ip6 nexthdr icmpv6 accept
                udp dport 53 accept
        }

        include "./inet-filter-sets.nft"
        include "./inet-filter-forward.nft"
        include "./inet-filter-local.nft"
}
```

（2）文件 defines.nft 内容：

```
[root@master1 ~]# vim defines.nft

# interfaces
define nic_inet = ens33
define nic_dmz = bond0
define nic_lan = bond1

# network ranks
define net_ipv4_dmz = 10.0.1.0/24
define net_ipv6_dmz = fe00:1::/64
define net_ipv4_lan = 10.0.2.0/24
define net_ipv6_lan = fe00:2::/64

# some machines
define server1_ipv4 = 10.0.1.2
define server1_ipv6 = fe00:1::2
define workstation1_ipv4 = 10.0.2.2
define workstation1_ipv6 = fe00:2::2
```

http://www.everyopen.org

（3）文件 inet-filter-sets.nft 内容：

```
[root@master1 ~]# vim inet-filter-sets.nft
set myset_ipv4 {
         type ipv4_addr;
         elements = { $server1_ipv4 , $workstation1_ipv4 }
}

set myset_ipv6 {
         type ipv6_addr;
         elements = { $server1_ipv6 , $workstation1_ipv6 }
}
```

（4）文件 inet-filter-forward.nft 内容：

```
[root@master1 ~]# vim inet-filter-forward.nft
chain dmz_in {
        # your rules for traffic to your dmz servers
        ip saddr @myset_ipv4
        ip6 saddr @myset_ipv6
}

chain dmz_out {
        # your rules for traffic from the dmz to internet
}

chain lan_in {
        # your rules for traffic to your LAN nodes
}

chain lan_out {
        # your rules for traffic from the LAN to the internet
}

chain forward_ch {
        type filter hook forward priority 0; policy drop;
        jump global_ch
        oifname vmap { $nic_dmz : jump dmz_in , $nic_lan : jump lan_in }
        oifname $nic_inet iifname vmap { $nic_dmz : jump dmz_out , $nic_lan : jump lan_out }
}
```

（5）文件 inet-filter-local.nft 内容：

```
[root@master1 ~]# vim inet-filter-local.nft
chain input_ch {
        type filter hook input priority 0 ; policy drop;
        jump global_ch
        # your rules for traffic to the firewall here
}

chain output_ch {
        type filter hook output priority 0 ; policy drop;
        jump global_ch
        # your rules for traffic originated from the firewall itself here
}
```

拓展学习：

1.

第5章 SELinux 安全模块

SELinux 是 Linux 的一个安全增强内核模块，基于常见的 RBAC 机制，控制主体（进程）如何访问客体（文件、目录、进程和网络端口等），以确保主体拥有且只拥有完成任务的最小权限。SELinux 的运行模式有三种：disabled（禁用模式）、permissive（宽容模式）和 enforcing（增强模式），disabled 就是关闭 SELinux 内核模块，也就是不使用它，permissive 只是记录访问日志，并不做任何权限控制，而 enforcing 不但记录访问日志，还做权限控制，请参考图 5-1。

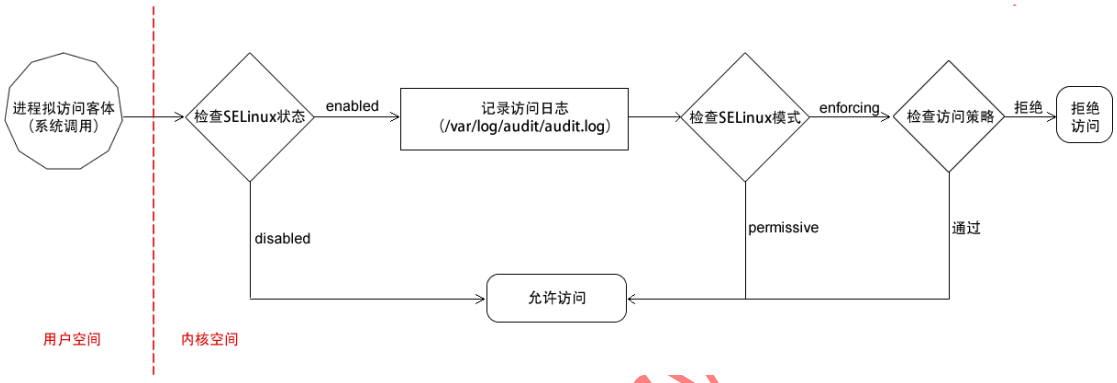


图 5-1 SELinux 原理

Linux 操作系统中的 SELinux 的默认模式是 enforcing，可修改文件/etc/selinux/config 中的 SELINUX 参数来改变电脑重启后的 SELinux 的运行模式（/etc/sysconfig/selinux 是符号链接文件）。permissive 和 enforcing 之间可用命令 setencforce 0|1 临时切换，如 setenforce 0 从 enforcing 切换到 permissive，但没法临时切换到 disabled，也不能临时从 disabled 切换到两外两种模式。可用命令 getenforce 查看 SELinux 的当前模式。

为了完成本章的实战，请加装下面的软件：

```
[root@master1 ~]# dnf install setools-console policycoreutils-python-utils audit -y
```

https://www.very

§5.1 SELinux 策略

在增强状态下，进程每次访问客体，都要检查访问策略。SELinux 策略就是一些规则的集合，它定义哪个主体允许访问哪些客体。进程就是运行的程序，而程序开发人员是最清楚开发的程序需要访问哪些客体的，因此 SELinux 策略一般是由程序开发人员编写并随同程序一同发布的，当然运维人员最好严格审查这些策略。

在增强状态下，SELinux 存在三种策略类型，它们分别是：

1. targeted：这是默认值。本策略针对系统中的服务进程做访问控制，同时，它还可以限制其他进程和用户。服务进程被放入沙盒中，在此环境中，服务进程会被严格限制，以便使通过此类进程所引发的恶意攻击不会影响到其他服务或 Linux 系统。沙盒（sandbox）是一种环境，在此环境中的进程对其他进程或资源的访问会被严格控制。换句

话说，位于沙盒中的各个进程，都只是运行在自己的域（进程所运行的区域被称为“域”）内，它们无法访问其他进程或资源（除非被授予特殊的权限）。通过使用此策略，可以更加安全地共享打印服务器、文件服务器、Web 服务器或其他服务，同时降低因访问这些服务而对系统中其他资源造成不利影响的风险。

2. MLS：是 Multi-Level Security 的缩写，该策略会对系统中的所有进程进行控制。启用 MLS 之后，用户即便执行最简单的指令（如 ls），都会报错。

3. Minimum：意思是“最小限制”，该策略最初是针对低内存计算机或者设备（比如智能手机）而创建的。从本质上来说，Minimun 和 Target 类似，不同之处在于，它仅使用基本的策略规则包。对于低内存设备来说，Minumun 策略使得 SELinux 不会消耗过多内存。

策略类型由/etc/selinux/config 中的 SELINUXTYPE 参数定义。采用 sestatus 命令可查看当前的状态和策略：

```
[root@master1 ~]# sestatus
SELinux status:                 enabled
#启用状态
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
#策略类型
Current mode:                   enforcing                                    #运行模
式
Mode from config file:          enforcing
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Memory protection checking:     actual (secure)
Max kernel policy version:      33
```

SELinux 使用上下文（context）来识别与应用程序关联的客体，上下文是在 Linux 系统中的每个客体上分配的有关安全信息的集合。使用 ls、ps、ss 命令时带上参数-Z 就能显示客体的上下文。

```
[root@master1 ~]# ls -lZ /boot
总用量 176632
-rw-r--r--. 1 root root system_u:object_r:boot_t:s0        218241  8 月 11 05:12 config-5.14.0-354.el9.x86_64
drwxr-xr-x. 3 root root system_u:object_r:boot_t:s0            17  8 月 28 08:59 efi
drwx------. 5 root root system_u:object_r:boot_t:s0            97  8 月  2 23:19 grub2
-rw-------. 1 root root system_u:object_r:boot_t:s0      83747872  8 月 28 09:01 initramfs-0-rescue-0f78834755b847e9b97cf91153ff8d47.img
-rw-------. 1 root root unconfined_u:object_r:boot_t:s0  31784695  8 月 28 09:30 initramfs-5.14.0-354.el9.x86_64.img
-rw-------. 1 root root system_u:object_r:boot_t:s0      30722048  8 月 28 09:31 initramfs-5.14.0-354.el9.x86_64kdump.img
drwxr-xr-x. 3 root root system_u:object_r:boot_t:s0            21  8 月 28 09:00 loader
lrwxrwxrwx. 1 root root unconfined_u:object_r:boot_t:s0        45  8 月 28 09:30 symvers-5.14.0-354.el9.x86_64.gz
-rw-------. 1 root root system_u:object_r:system_map_t:s0  8486371  8 月 11 05:12 System.map-5.14.0-354.el9.x86_64
-rwxr-xr-x. 1 root root system_u:object_r:boot_t:s0       12543240  8 月 28 09:01 vmlinuz-0-rescue-0f78834755b847e9b97cf91153ff8d47
-rwxr-xr-x. 1 root root system_u:object_r:boot_t:s0       13357000  8 月 11 05:12 vmlinuz-5.14.0-354.el9.x86_64

[root@master1 ~]# ss -tnlpZ
...Send-Q  Local Address:Port    Peer Address:Port Process
...128           0.0.0.0:22          0.0.0.0:*     users:(("sshd",pid=655,proc_ctx=system_u:system_r:sshd_t:s0-s0:c0.c1023,fd=3))
...4096      127.0.0.1:45517       0.0.0.0:*     users:(("containerd",pid=659,proc_ctx=system_u:system_r:container_runtime_t:s0,fd=15))
...4096      127.0.0.1:10248       0.0.0.0:*     users:(("kubelet",pid=665,proc_ctx=system_u:system_r:kubelet_t:s0-s0:c0.c1023,fd=11))
...4096      127.0.0.1:10249       0.0.0.0:*     users:(("kube-proxy",pid=1327,proc_ctx=system_u:system_r:spc_t:s0,fd=10))
...4096            *:10250             *:*     users:(("kubelet",pid=665,proc_ctx=system_u:system_r:kubelet_t:s0-s0:c0.c1023,fd=24))
...4096            *:10256             *:*     users:(("kube-proxy",pid=1327,proc_ctx=system_u:system_r:spc_t:s0,fd=8))
...128          [::]:22             [::]:*     users:(("sshd",pid=655,proc_ctx=system_u:system_r:sshd_t:s0-s0:c0.c1023,fd=4))

[root@master1 ~]# ps -Zaux
LABEL                                USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
system_u:system_r:init_t:s0          root           1  0.0  0.9 171832 17408 ?        Ss   10:59   0:00 /usr/lib/systemd/syste ...
system_u:system_r:kernel_t:s0        root           2  0.0  0.0      0     0 ?        S    10:59   0:00 [kthreadd]
system_u:system_r:kernel_t:s0        root           3  0.0  0.0      0     0 ?        I<   10:59   0:00 [rcu_gp]
system_u:system_r:kernel_t:s0        root           4  0.0  0.0      0     0 ?        I<   10:59   0:00 [rcu_par_gp]
……
system_u:system_r:kernel_t:s0        root       11172  0.0  0.0      0     0 ?        I    11:35   0:00 [kworker/1:2-ata_sff]
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 root 11739 0.0  0.3 18268 6300 pts/0 R+ 11:37   0:00 ps -Zaux
```

上下文包含五部分，用分号隔开，它们是：

用户:角色:域或类型:敏感度[:类别]

其中：

1. “用户”部分常带有“_u”后缀，表示 SELinux 用户账号。SELinux 用户账号不同于 Linux 操作系统的用户账号，它直接在策略中定义，在电脑启动时被载入内存中。Linux 操作系统的用户账号在策略中可被映射成 SELinux 用户账号。可用 semanage user -l 命令列出当下全部的用户身份。

```
[root@master1 ~]# semanage user -l

                Labeling   MLS/       MLS/
SELinux User    Prefix     MCS Level  MCS Range                    SELinux Roles
```

```
guest_u         user       s0         s0                          guest_r
root            user       s0         s0-s0:c0.c1023              staff_r sysadm_r system_r unconfined_r
staff_u         user       s0         s0-s0:c0.c1023              staff_r sysadm_r system_r unconfined_r
sysadm_u        user       s0         s0-s0:c0.c1023              sysadm_r
system_u        user       s0         s0-s0:c0.c1023              system_r unconfined_r
unconfined_u    user       s0         s0-s0:c0.c1023              system_r unconfined_r
user_u          user       s0         s0                          user_r
xguest_u        user       s0         s0                          xguest_r
```

guest_u：这种类型的用户无权访问网络，在 /home 中没有脚本执行权限，也无法使用 sudo 或 su 命令来获得更高的权限，它只能使用 guest_r 角色。

staff_u：映射到此 SELinux 用户的 Linux 操作系统用户可以访问 GUI、网络以及使用 sudo 命令来获取权限，它允许在 stuff_r、sysadm_r、system_r 和 unconfined_r 角色之间切换，命令 newrole -r <new role>切换。

sysadmin_u：在 staff_u 的基础上，还可以使用 su 命令。它只能被赋予 sysadm_r 角色。

system_u：只能分配给系统服务的用户（Linux 系统用户），不应将任何 Linux 操作系统的普通用户映射到它。系统用户用于跑服务程序（如 sshd 用户用来跑 ssh 服务），而普通用户分配给职员用于登录。

user_u：只分配给 Linux 的普通用户。

unconfined_u：这种类型的用户没有任何限制，它能赋予 unconfined_r 和 system_r 角色。Linux 操作系统用户 root 默认被映射到 unconfined_u，可用命令 semanage login -l 列出映射关系。

xgust_u：该 SELinux 用户可以访问 GUI 和网络，但只能通过 Firefox 浏览器访问。它对 /home 下的文件没有执行权限，只能赋予 xguest_r 角色。

实验：

```
[root@master1 ~]# useradd alice
[root@master1 ~]# passwd alice
[root@master1 ~]# semanage login -a -s guest_u alice;

然后 alice 用户登录（不能用 su - alice 切换）后执行:
[alice@master1 ~]$ ping www.veryopen.org
[alice@master1 ~]$ ping 127.0.0.1
[alice@master1 ~]$
```

ping 任何主机都没有反馈，因为 alice 被映射到 guest_u，所以 alice 运行的进程上下文是 guest_u，不允许使用网路。“semanage user ...”，命令管理 SELinux 用户。

2. “角色”部分一般带有“_r”后缀，定义主体可以对指定域中的客体执行的操作。常见的值有 object_r（文件或目录）、system_r（进程）。查看一个角色能访问哪些域和类型的命令是 seinfo -r <角色> -x，如：

```
[root@master1 ~]# seinfo -r staff_r -x
Roles: 1
   role staff_r types { abrt_helper_t alsa_home_t antivirus_home_t audio_home_t auth_home_t bluetooth_helper_t ...}
```

3. “域或类型”部分一般带有“_t”后缀，定义主体的域或客体的类型。比如文件的类型有：/etc 目录文件、页面文件、设备文件、套接字文件等。“域或类型”是安全上下文中最重要的字段，进程是否可以访问文件，主要就是看进程的安全上下文域字段是否和文件的安全上下文类型字段相匹配，如果匹配成功则可以访问。“域或类型”字段在文件或目录的安全上下文中被称作类型（type），但是在主体的安全上下文中被称作域（domain）。只有主体的域匹配客体的类型才能访问，“匹配”在策略规则中定义。比如 httpd 进程访问/var/www/html 目录的例子：

```
[root@master1 ~]#  ps auxZ | grep httpd
system_u:system_r:httpd_t:s0    root       58491  1.5  0.6  20176 11484 ?        Ss   16:32   0:00 /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0    apache     58493  0.0  0.4  21608  7176 ?        S    16:32   0:00 /usr/sbin/httpd -DFOREGROUND
system_u:system_r:httpd_t:s0    apache     58494  0.0  1.0 2521236 19080 ?       Sl   16:32   0:00 /usr/sbin/httpd -DFOREGROUND
[root@master1 ~]# ls -dZ /var/www/html/
system_u:object_r:httpd_sys_content_t:s0 /var/www/html/
[root@master1 ~]# sesearch -A -s httpd_t -t httpd_sys_content_t | grep httpd_sys_content_t
allow daemon httpd_sys_content_t:dir { getattr open search };
allow httpd_t httpd_sys_content_t:dir { ioctl lock read };
allow httpd_t httpd_sys_content_t:lnk_file { getattr read };
```

即 httpd 进程的域为 httpd_t，目录/var/www/html 的类型为 httpd_sys_content_t，这二者在策略规则中做了匹配，参见上面黄色字体部分，第一条规则表示域为 httpd_t 的进程允许对类型为 httpd_sys_content_t 的目录的操作：ioctl、lock 和 read，第二条规则表示域为 httpd_t 的进程允许对类型为 httpd_sys_content_t 的普通文件或链接文件的操作：getattr 和 read，其中 getattr 获取文件的远信息（如文件名、修改时间等），read 读文件内容。

https://www.veryopen.org

新建的文件继承所在目录的安全上下文，可采用命令 chcon 修改文件的安全上下文，如：

```
[root@master1 ~]# chcon staff_u:object_r:user_home_t:s0 file123.txt
[root@master1 ~]# chcon -t user_home_t /var/www/html/test.html
```

chcon 命令修改的上下文是临时的，重启电脑后就没有了。采用 semanage fcontext 可永久修改，如：

```
[root@master1 ~]# semanage fcontext -a -t httpd_sys_content_t /home/egdoc/test(/.*)?
[root@master1 ~]# restorecon -vR /home/egdoc/test                    #使修改后的上下文生效
```

4. “敏感度”定义了分层安全级别，一共有 15 层（0~14），0 层为最低级别，用“s”做前缀，如 s0、s5 等。范围运算符“-”表示从哪个级别到哪个级别，如 s0-s3。对于 targeted 策略类型，只采用 s0。

5. “类别”定义平行的（非分层）类别，用 c0~c1023 表示。仅当 SELinux 策略类型设置为 MLS（Multi-Level Security） 模式时，才用到安全类别。MLS 模式是为需要非常高安全性的部门开发的，如军事、国防部等。民用领域通常不使用这种细粒度的安全级别模型，而采用适当布尔值的 targeted 模式就足够了。一个对象允许指定多个类别，“.”表示范围，“,”为分隔符，如“c0,c5.c8”一共指定了 5 个类别，即 c0、c5、c6、c7、c8。

可采用命令 semanage login --modify ...修改敏感度和类别，如：

```
[root@master1 ~]# semanage login --modify --range s0:c100 alice
```

前提是 alice 已经映射到 SELinux 用户，在 targted 模式下，敏感度只能是 s0。

命令 runcon 以指定的上下文运行程序，例如：

```
[root@master1 ~]# runcon                                             #显示当前的安全上下文
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[root@master1 ~]# runcon guest_u:guest_r:guest_t:s0 ping 8.8.8.8     #以其他上下文运行 ping 命令
runcon: 'ping': Permission denied
[root@worker1 ~]# runcon -u staff_u -r system_r ping 9.9.9.9
runcon: 'ping': Permission denied
```

注意：用户、角色、域和敏感度不能随便搭配，其中用户、角色和敏感度的搭配参考“semanage user -l”命令的输出，可搭配的域参考命令“seinfo -r <角色> -x”的输出。

§5.2 布尔参数

布尔参数能改变 SELinux 的行为，可用 semanage boolean -l 或 getsebool -a 列出全部的布尔参数，如：

```
[root@master1 ~]# semanage boolean -l | more
SELinux boolean                 State  Default Description

abrt_anon_write                 (off  ,  off)  Allow abrt to anon write
abrt_handle_event               (off  ,  off)  Allow abrt to handle event
abrt_upload_watch_anon_write    (on   ,   on)  Allow abrt to upload watch anon write
antivirus_can_scan_system       (off  ,  off)  Allow antivirus to can scan system
antivirus_use_jit               (off  ,  off)  Allow antivirus to use jit
auditadm_exec_content           (on   ,   on)  Allow auditadm to exec content
authlogin_nsswitch_use_ldap     (off  ,  off)  Allow authlogin to nsswitch use ldap
……


[root@master1 ~]# getsebool -a | more
abrt_anon_write --> off
abrt_handle_event --> off
abrt_upload_watch_anon_write --> on
antivirus_can_scan_system --> off
antivirus_use_jit --> off
auditadm_exec_content --> on
authlogin_nsswitch_use_ldap --> off
authlogin_radius --> off
authlogin_yubikey --> off
awstats_purge_apache_log_files --> off
boinc_execmem --> on
cdrecord_read_content --> off
……
```

https://www.weiqiaoen.org

设置布尔参数的值可采用命令“semanage boolean -m -1|-0 <布尔参数>” 或者“setsebool [-P|-N] <布尔参数> 1|0”或者“setsebool [-P|-N] 布尔参数 1=1|0 布尔参数 2=1|0 …”，其中-P 表示永久有效。如：

```
[root@master1 ~]# setsebool -P httpd_enable_ftp_server 1
此命令等价于:
[root@master1 ~]# semanage boolean -m -1 httpd_enable_ftp_server
```

§5.3 查看 SELinux 日志

SELinux 把日志登记在/var/log/audit/audit.log 文件中，除了用 vim 打开查看外，还额外提供了几个命令。

1. audit2why

2. audit2allow

3. sealert

https://www.veryopen.org

第6章 程序运行时

作为计算机专业的学生，都应该开发过应用程序，写程序无非就是三部曲：写源代码，然后编译源程序，最后运行程序。那到底存在几种编译方法？运行程序时又需要什么？这是本章要回答的问题。

§6.1 动态编译

首先安装 c 语言编译环境：

```
[root@master1 ~]# dnf install gcc glibc-static -y                    #安装 c 语言编译器和函数库
```

在 Rocky 上安装 c 语言开发环境的另一个方法是运行命令“dnf install "@Development Tools"”，不过这个方法还会安装其他语言的开发环境，会占用较大的磁盘空间。

1. 编写打印“Hello World”的 c 语言程序：

```
[root@master1 ~]# vim hello.c                                        #编辑 c 语言源程序
#include <stdio.h>

int main()
{
        printf("Hello World!\n");
}


[root@master1 ~]# gcc hello.c                                        #动态编译
[root@master1 ~]# ls -l a.out
-rwxr-xr-x 1 root root 24272 Jul 19 11:34 a.out                      #编译后产生的可执行程序

[root@master1 ~]# ./a.out                                            #执行刚编译的程序
Hello World!
```

编译时会把头文件 stdio.h（对应的文件路径/usr/include/stdio.h）的内容包含进来，里面包含了函数 printf()的声明：

```
......
extern int printf (const char *__restrict __format, ...);
......
```

但这个函数具体实现的代码以二进制格式提供，这些可执行的二进制代码包含在/lib64/libc.so.6 文件中，而此文件又包含在 glibc 软件包中，在安装操作系统时会自动安装这个软件包。

动态编译时会在目标程序（比如上面的 a.out 程序）中产生一个指针，指到/lib64/libc.so.6 文件中具体实现 printf 函数的二进制代码块，因此动态编译后的程序（如 a.out）都比较小。/lib64/libc.so.6 称为动态库，里面除了 printf 函数的二进制代码外，还包含大量其他函数的二进制代码，这些代码在程序（比如 a.out、ls、vim 等）执行时被调用，如果不小心删除了/lib64/libc.so.6 文件，很多命令是无法执行的。什么？你不信？那你现在就删掉它试试！

2. 查看程序需要的动态库：

```
[root@master1 ~]# ldd a.out
        linux-vdso.so.1 (0x00007ffcb65d0000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fc4acc00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fc4acecb000)
```

https://www.yoyen.org

```
[root@master1 ~]# which ls
alias ls='ls --color=auto'
        /usr/bin/ls

[root@master1 ~]# ldd /usr/bin/ls
        linux-vdso.so.1 (0x00007ffc58bfa000)
        libselinux.so.1 => /lib64/libselinux.so.1 (0x00007fbbf58f7000)
        libcap.so.2 => /lib64/libcap.so.2 (0x00007fbbf58ed000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fbbf5600000)
        libpcre2-8.so.0 => /lib64/libpcre2-8.so.0 (0x00007fbbf5851000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fbbf594e000)

[root@master1 ~]# ldd /usr/bin/vim
        linux-vdso.so.1 (0x00007ffdaa197000)
        libm.so.6 => /lib64/libm.so.6 (0x00007fd5fb125000)
        libselinux.so.1 => /lib64/libselinux.so.1 (0x00007fd5fb614000)
        libtinfo.so.6 => /lib64/libtinfo.so.6 (0x00007fd5fb5e5000)
        libacl.so.1 => /lib64/libacl.so.1 (0x00007fd5fb5da000)
        libgpm.so.2 => /lib64/libgpm.so.2 (0x00007fd5fb11d000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fd5fae00000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fd5fb647000)
        libpcre2-8.so.0 => /lib64/libpcre2-8.so.0 (0x00007fd5fb081000)
        libattr.so.1 => /lib64/libattr.so.1 (0x00007fd5fb079000)
```

即红色字体部分就是动态库文件，要运行某个程序，这些相应的动态库文件必须存在。

现在做个实验，把最常用的动态库文件/lib64/libc.so.6 删掉，会发生什么情况。

```
[root@master1 ~]# rm -rf /lib64/libc.so.6

[root@master1 ~]# ls
ls: error while loading shared libraries: libc.so.6: cannot open shared object file: No such file or directory

[root@master1 ~]# vim hello.c
vim: error while loading shared libraries: libc.so.6: cannot open shared object file: No such file or directory

[root@master1 ~]# shutdown -h now
shutdown: error while loading shared libraries: libc.so.6: cannot open shared object file: No such file or
directory
```

这下糟了，几乎全部命令都无法执行了，也没法关机，只有强行关机，然后恢复快照了。

一个程序在执行时，所需要的所有动态库文件组成为该程序的运行环境，简称为程序运行时。所以容器里运行的服务程序，其运行环境必须在容器中。

§6.2 静态编译

既然存在动态编译，聪明的你可能会疑惑：莫非还有静态编译？没错，真的有，下面我们来静态编译 hello.c：

```
[root@master1 ~]# gcc -static -o myhello hello.c                                    #编译后的程序名为 hello

[root@master1 ~]# ls -l a.out myhello
-rwxr-xr-x 1 root root   24272 Jul 19 14:05 a.out                                   #之前动态编译的程序
-rwxr-xr-x 1 root root 1689272 Jul 19 14:05 myhello                                 #刚刚静态编译的程序

[root@worker3 ~]# ./myhello                                                         #运行
Hello World!

[root@worker3 ~]# ldd myhello                                                       #查看运行时
        not a dynamic executable
```

gcc 命令带参数“-static”就是静态编译程序，“-o myhello”参数指明产生的程序名为 myhello。静态编译会把静态库文件中对应的代码复制到目标程序中（比如 myhello），所以编译后的程序特别大，比如编译同一个源程序，动态编译的程序 a.out 只有 24272 个字节，而静态编译的程序 myhello 有 1689272 个字节。在 Linux 中，以.a 为后缀的文件为静态库文件，比如/usr/lib64/libc.a 文件中包含了绝大多数 c 语言函数的二进制代码，它是随

glibc-static 软件包安装进来的，如果没有安装这个包，就不能静态编译 c 语言程序，但是一旦编译完成，执行时就不再依赖静态库了，更不用动态库。

上面的最后一条命令“ldd myhello”查看程序的运行时，反馈说“不是一个动态可执行程序”，意为一个静态可执行程序。静态可执行程序没有程序运行时，所以可拷贝到任何 Linux 系统中去执行，不过在 windows 上不能执行，因为两个操作系统的可执行程序格式完全不同。人们常说的“绿色软件”就是指不用安装的软件，直接拷贝过来就可执行，要我说，静态可执行软件才算真正的“纯绿色软件”。

动态编译和静态编译示意图请参考图 6-1 所示。

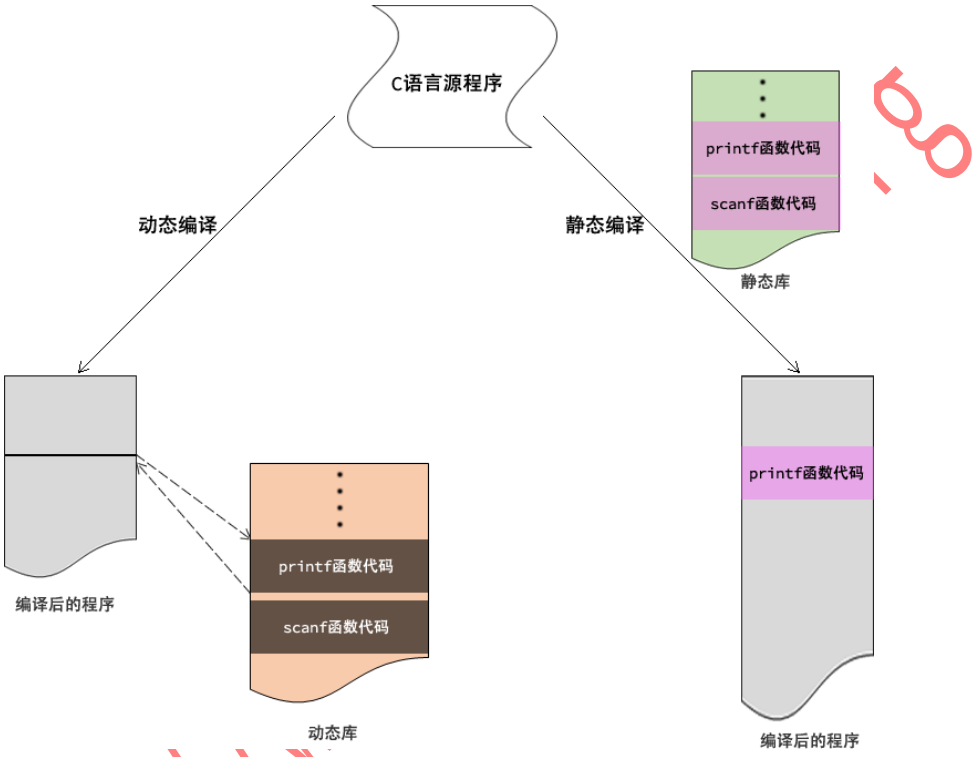


图 6-1 动态编译和静态编译示意图

由于动态编译后的可执行程序需要依赖额外的动态库文件，尤其当程序很大时会依赖很多的动态库文件，虽然在 Linux 系统中可通过依赖软件包来解决这个问题，比如大家经常遇到的情况是：我明明只安装一个软件包（比如 dnf install vim），可最终却安装了很多软件包，其中一部分依赖包就是程序运行时（即动态库）。但是我们也经常为“缺少 xx 依赖软件包”的报错而伤透了心，为此，谷歌公司开发的 go 语言只采用静态编译方法——一次编译，随处可用。以前之所以采用动态编译，重要的原因之一是减小程序大小来降低磁盘消耗和内存占用，并在执行时减少加载到内存的时间，但是降低了执行效率，即用空间换时间。今天的磁盘、内存和 CPU 都已经足够大足够快了，更好的用户体验就显得尤为重要，所以静态编译逐渐流行起来。go 语言自诞生以来，在计算机编程语言排行榜上节节攀升（参见网站 https://www.tiobe.com/tiobe-index/）。kubernetes 就是用 go 语言编写的。

PaaS 云产品的一个重要特点就是提供了某些编程语言的程序运行时，但一款 PaaS 产品很难做到所有编程语言的程序运行时，因为计算机编程语言实在大多了。

容器里只包含了应用程序和该应用程序的运行时，而不像虚拟机那么笨重，因为虚拟机包含了整个操作系统。

§6.3 chroot 实战

命令 chroot 实现在一个新的根目录中执行命令。平时我们登录到 Linux 系统，执行的所有命令都是在根目录（即“/”目录）所在的文件系统中完成的，比如列出/boot 目录下文件，把/etc/profile 文件拷贝到/tmp/目录中，显示/var/log/syslog 文件的内容。而一旦通过 chroot 切换到另一个目录 X 之后，那个目录 X 就成了新的根目录，这时/boot 可能不存在，/bin 目录中的文件也不同于宿主根下/bin 目录中的文件。当然能作为新根的目录，必须满足一定的目录结构才行。

比如执行命令“chroot /opt/sandbox /bin/bash”，首先会把/opt/sandbox 切换为新根，然后去执行新根中的/bin/bash 命令，参考图 6-2 所示。

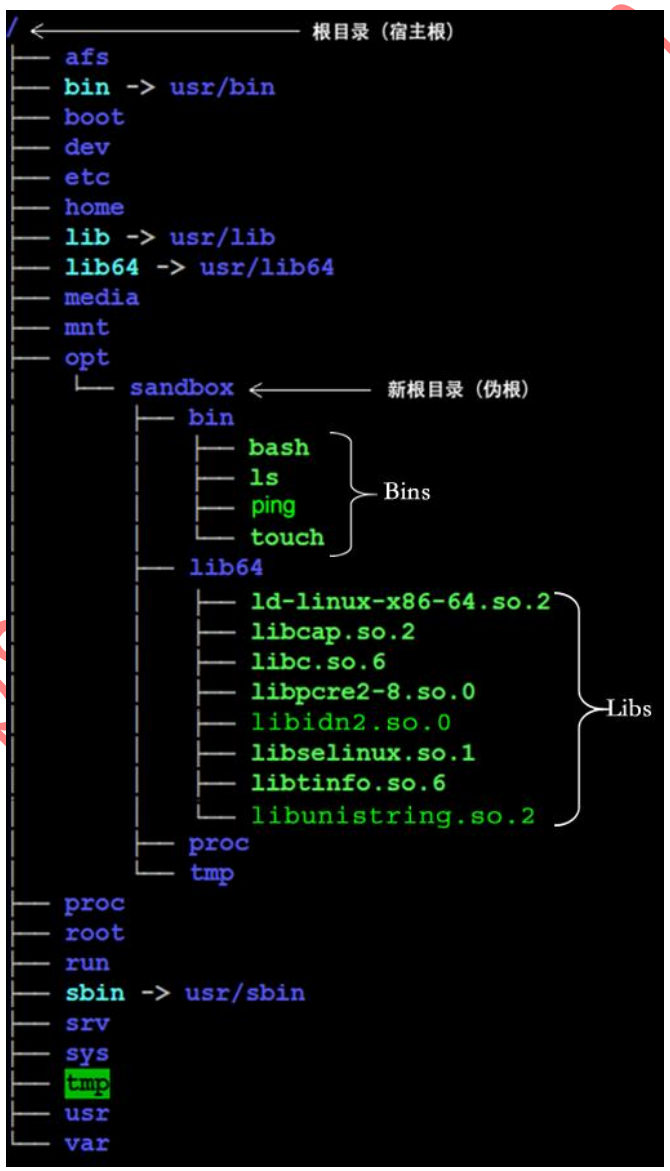


图 6-2 新根目录结构示意图

上图中的 Bins 和 Libs 就是指 Binaries 和 Libraries，即二进制课执行程序和动态库，就是图 3-1 中的虚拟机和容器中的 Bins/Libs。

下面我们先来做一个实验。

1. 构造新根目录结构：

```
创建新根目录结构：
[root@master1 ~]# mkdir -p /opt/sandbox/{bin,lib64,tmp,proc}                    #在/opt/sandbox 下创建了四个子目录

把 bash、ls、touch 三个程序拷贝到新根的 bin 目录中：
[root@master1 ~]# cp -v /usr/bin/{bash,ls,touch} /opt/sandbox/bin/
'/usr/bin/bash' -> '/opt/sandbox/bin/bash'
'/usr/bin/ls' -> '/opt/sandbox/bin/ls'
'/usr/bin/touch' -> '/opt/sandbox/bin/touch'

[root@master1 /]# which ping
/usr/sbin/ping
[root@master1 /]# cp -v /usr/sbin/ping /opt/sandbox/bin/
'/usr/sbin/ping' -> '/opt/sandbox/bin/ping'

查看 bash 命令的程序运行时：
[root@master1 ~]# ldd /usr/bin/bash | egrep -o '/lib.*\.[0-9]+'
/lib64/libtinfo.so.6
/lib64/libc.so.6
/lib64/ld-linux-x86-64.so.2

把/bin/bash 的程序运行时拷贝到行新根的 lib64 目录下：
[root@master1 ~]# cp -v /lib64/{libtinfo.so.6,libc.so.6,ld-linux-x86-64.so.2} /opt/sandbox/lib64/
'/lib64/libtinfo.so.6' -> '/opt/sandbox/lib64/libtinfo.so.6'
'/lib64/libc.so.6' -> '/opt/sandbox/lib64/libc.so.6'
'/lib64/ld-linux-x86-64.so.2' -> '/opt/sandbox/lib64/ld-linux-x86-64.so.2'

查看 ls 命令的程序运行时：
[root@worker3 ~]# ldd /usr/bin/ls | egrep -o '/lib.*\.[0-9]+'
/lib64/libselinux.so.1
/lib64/libcap.so.2
/lib64/libc.so.6
/lib64/libpcre2-8.so.0
/lib64/ld-linux-x86-64.so.2

把/bin/ls 的程序运行时拷贝到行新根的 lib64 目录下（其中两个文件前面已经拷贝了）：
[root@master1 ~]# cp -v /lib64/{libselinux.so.1,libcap.so.2,libpcre2-8.so.0} /opt/sandbox/lib64/
'/lib64/libselinux.so.1' -> '/opt/sandbox/lib64/libselinux.so.1'
'/lib64/libcap.so.2' -> '/opt/sandbox/lib64/libcap.so.2'
'/lib64/libpcre2-8.so.0' -> '/opt/sandbox/lib64/libpcre2-8.so.0'

查看 touch 命令的程序运行时（前面都已经拷贝了）：
[root@master1 ~]# ldd /usr/bin/touch | egrep -o '/lib.*\.[0-9]+'
/lib64/libc.so.6
/lib64/ld-linux-x86-64.so.2

查看 ping 命令的程序运行时
[root@master1 /]# ldd /usr/sbin/ping | egrep -o '/lib.*\.[0-9]+'
/lib64/libcap.so.2
/lib64/libidn2.so.0
/lib64/libc.so.6
/lib64/libunistring.so.2
/lib64/ld-linux-x86-64.so.2

[root@master1 /]# cp -v /lib64/{libidn2.so.0,libunistring.so.2} /opt/sandbox/lib64/
'/lib64/libidn2.so.0' -> '/opt/sandbox/lib64/libidn2.so.0'
'/lib64/libunistring.so.2' -> '/opt/sandbox/lib64/libunistring.so.2'

最后看看新根的目录结构（与图 6-2 中的/opt/sandbox 目录一致）：
[root@master1 ~]# tree /opt/sandbox/
/opt/sandbox/
├── bin
│   ├── bash
│   ├── ls
│   ├── ping
│   └── touch
├── lib64
│   ├── ld-linux-x86-64.so.2
│   ├── libcap.so.2
│   ├── libc.so.6
│   ├── libidn2.so.0
│   ├── libpcre2-8.so.0
│   ├── libselinux.so.1
│   ├── libtinfo.so.6
│   └── libunistring.so.2
├── proc
└── tmp
```

2. 切换到新根/opt/sandbox 并执行其中的/bin/bash 程序（类似启动一个容器并进入容器）：

```
[root@master1 ~]# chroot /opt/sandbox /bin/bash
bash-5.1# pwd                                                    #已经进入了新根并执行了 bash
/
bash-5.1# ls
bash: ls: command not found
bash-5.1# export PATH=/bin
bash-5.1# ls
bin  lib64  proc  tmp
bash-5.1# ls -l
total 0
drwxr-xr-x 2 0 0  41 Jul 20 01:21 bin
drwxr-xr-x 2 0 0 137 Jul 20 01:24 lib64
drwxr-xr-x 2 0 0   6 Jul 20 00:46 proc
drwxr-xr-x 2 0 0   6 Jul 20 00:46 tmp
bash-5.1# cd /
bash-5.1# ls -l
total 0
drwxr-xr-x 2 0 0  41 Jul 20 01:21 bin
drwxr-xr-x 2 0 0 137 Jul 20 01:24 lib64
drwxr-xr-x 2 0 0   6 Jul 20 00:46 proc
drwxr-xr-x 2 0 0   6 Jul 20 00:46 tmp
bash-5.1# cd /bin
bash-5.1# ls -l
total 1592
-rwxr-xr-x 1 0 0 1388920 Jul 20 01:21 bash
-rwxr-xr-x 1 0 0  140760 Jul 20 01:21 ls
-rwxr-xr-x 1 0 0   94048 Jul 20 01:21 touch

bash-5.1# ping 9.9.9.9
PING 9.9.9.9 (9.9.9.9) 56(84) bytes of data.
64 bytes from 9.9.9.9: icmp_seq=1 ttl=128 time=213 ms
64 bytes from 9.9.9.9: icmp_seq=2 ttl=128 time=214 ms
^C

bash-5.1# exit
exit
[root@wmaster1 ~]#
```

chroot 让程序在新根中运行有一个好处，那就是隔离了文件系统，那个程序（在新根中运行的程序）如果内置了木马想盗取系统中重要的文件（比如/etc/passwd）或者销毁重要的数据，是不可能的了。

课堂实战练习 1：在以上的“容器”里再放入 vim 命令。

chroot 命令的语法如下：

```
chroot    [参数]    <新根目录>    [程序    [程序的参数]  ]
          |
          --userspec=用户:用户组      #指定切换新根之后运行程序的用户和组，如果省略，就用当前用户和组。
          --groups=附加组1,附加组2,...  #给切换到新根之后的用户指定更多的附加用户组。
          --skip-chdir               #当新根还是“/”时，不改变当前的目录。如果省略本参数，则改变到/目录。
```

语法中的中括号部分表示可以省略，尖括号部分不能省略，因此简单的用法就是类似“chroot newroot”命令，它表示切换到新根目录 newroot，然后以当前用户（执行 chroot 命令的用户）执行“"$SHELL" -i”程序。环境变量 SHELL 保存了当前用户的登录 shell 程序，默认是/bin/bash，可用下面的命令显示 SHELL 环境变量的值：

```
[root@master1 ~]# echo $SHELL
/bin/bash
```

切换到新根之后如果要以不同的用户和用户组运行程序，可通过参数--userspec 指定，如--userspec=alice:class2，不过用户 alice 和组 class2 必须在宿主根中存在。参数--groups 指定更多的附加组，如--groups=x86,sales,grade1，用户组之间用逗号分开，不能有空格，指定的附加组必须存在。参数--skip-chdir 只在新根为“/”时才有效，表示切换到新根之后不改变当前目录，执行 chroot 命令时的目录称为当前目录，看例子：

```
[root@master1 log]# pwd
```

```
/var/log                                                                                                    #当前目录
[root@master1 log]# chroot --skip-chdir /
[root@master1 log]# pwd
/var/log                                                                                                    #当前目录没改变
[root@master1 log]# exit
exit
[root@master1 log]# pwd
/var/log ---------------------------------------------------------------------------------------------------- #当前目录
[root@master1 log]# chroot /
[root@master1 /]# pwd
/ ---------------------------------------------------------------------------------------------------- #当前目录改变为“/”
[root@master1 /]# exit
exit
[root@master1 log]#
```

下面的例子切换到新根之后以 john 和 sales 组运行程序：

```
[root@master1 ~]# cp /usr/bin/id /opt/sandbox/bin/
[root@master1 ~]# groupadd sales ---------------------------------------------------------------------------- #创建用户组 sales
[root@master1 ~]# useradd -G sales john                                    #创建用户 john，并让它加入组 sales
[root@master1 ~]# chroot --userspec=john:sales /opt/sandbox /bin/bash -i  ------------#在新根中运行程序“/bin/bash -i”
bash-5.1$ pwd
/
bash-5.1$ /bin/ls -l
total 0
drwxr-xr-x 2 0 0  41 Jul 20 01:21 bin
drwxr-xr-x 2 0 0 137 Jul 20 01:24 lib64
drwxr-xr-x 2 0 0   6 Jul 20 00:46 proc
drwxr-xr-x 2 0 0   6 Jul 20 00:46 tmp
bash-5.1$ /bin/id
uid=1001 gid=1002 groups=1002 ------------------------------------------------------------------------------ #不是 root 用户了
bash-5.1$ exit
exit
[root@master1 ~]#
```

一般是在新根中运行应用程序，比如数据库服务。

课堂实战练习 2：忘记了 root 用户的密码怎么办？

利用本小节的知识，我们就能轻松解决这样的一个问题：忘记了 Linux 系统的 root 用户密码，且又没有能执行 sudo 命令的普通用户，此时如何重设 root 用户的密码？

图 6-3 是重设 Rocky Linux 9 的 root 用户密码，大致的过程是：需要先用安装光盘启动电脑并进入命令行，然后把硬盘根分区挂在到/mnt 目录上，接下来采用命令 chroot 把根切换到/mnt，最后修改 root 密码。

https://www.veryop

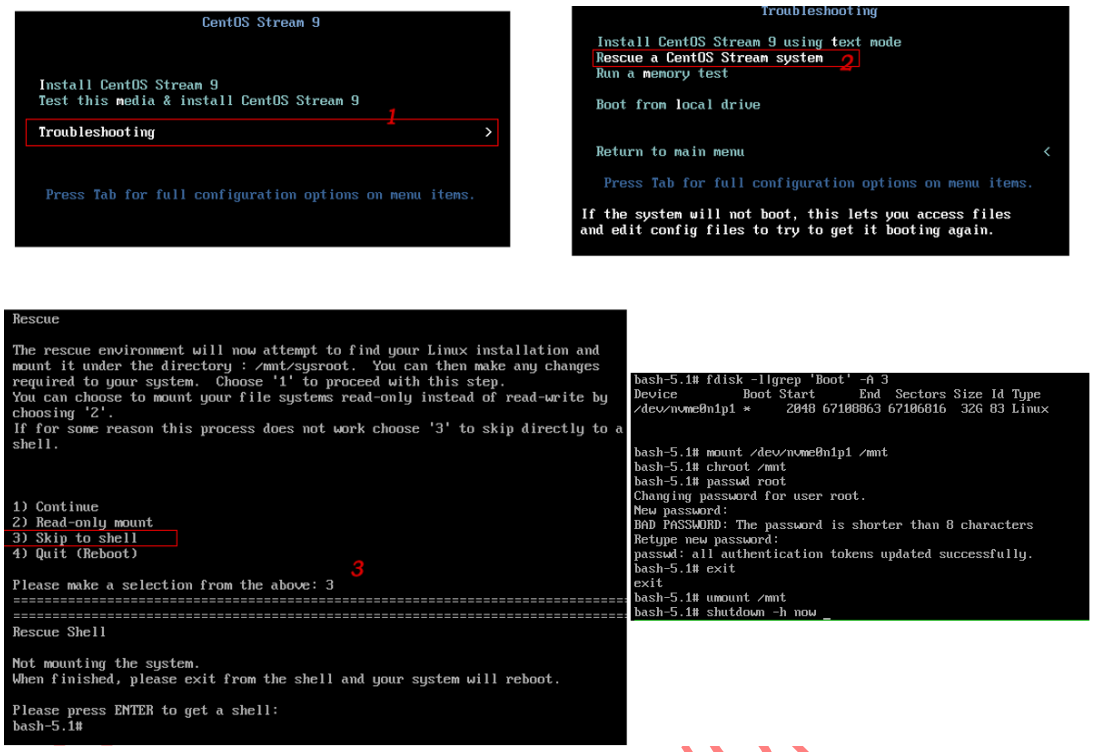


图 6-3 重设 Rocky Linux 9 的 root 密码

图中的命令 fdisk -l | grep ‘Boot’ -A 3 是找出硬盘的根分区，根分区的 Boot 标识一定是‘*’，所以本例中的根分区为/dev/nvme0n1p1。

如果在图 6-3 中左下部分选择“1) Continue”，那么会自动把根文件系统挂载到/mnt/sysroot 目录上，我们直接切换伪根并修改 root 密码即可，参考图 6-4 所示操作。

https://www.verycd.com

```
1) Continue
2) Read-only mount
3) Skip to shell
4) Quit (Reboot)

Please make a selection from the above: 1
================================================================================
================================================================================
Rescue Shell

Your system has been mounted under /mnt/sysroot.

If you would like to make the root of your system the root of the active system,
run the command:

        chroot /mnt/sysroot

When finished, please exit from the shell and your system will reboot.

Please press ENTER to get a shell:
bash-5.1# chroot /mnt/sysroot
bash-5.1# passwd root
Changing password for user root.
New password:
BAD PASSWORD: The password is shorter than 8 characters
Retype new password:
passwd: all authentication tokens updated successfully.
bash-5.1# exit
exit
bash-5.1# cd /
bash-5.1# umount /mnt/sysroot
bash-5.1# shutdown -h now
```

图 6-4 重设 root 密码

https://www.ve

第7章 控制组 cgroup

容器技术&云原生生态的大致发展历程参考图 7-1。

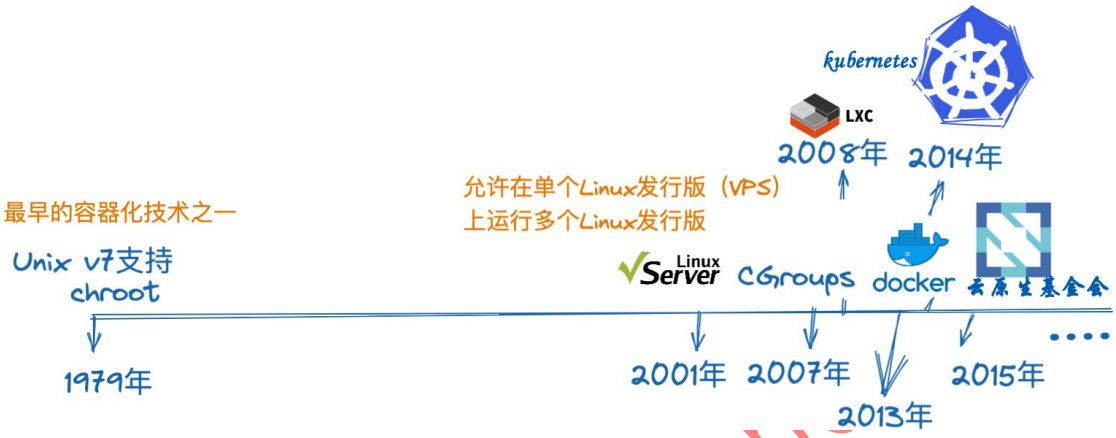


图 7-1 云原生的前生今世

https://www.veryop

§7.1 起底 cgroup v2

§7.1.1 简介

控制组（control group，简称 cgroup） 是 Linux 内核的一个特性（内核 2.6 引入 cgroup v1，4.5 引入 cgroup v2），用于限制、记录和隔离一组进程所消耗的资源，在 linux 系统中，常见的资源有 cpu、内存、I/O、进程数、RDMA 等）。cgroup 具有以下作用：

1. 资源限制：通过配置 cgroup 来限制进程消耗特定资源的上限。
2. 优先级：当资源稀缺时，控制进程获取资源的优先级。
3. 记录：在 cgroup 级别监控和报告资源使用状况。
4. 控制：可以使用单个命令更改 cgroup 中所有进程的状态（冻结、停止或重新启动）。

一个容器中通常运行了多个进程，并且您需要对这些进程实施统一控制，因此 cgroup 是容器的关键组件。学好 kubernetes 必先掌握容器技术，要掌握容器技术又必先精通 cgroup！因为 Kubernetes 环境使用 cgroup 在 pod 级别上配置资源请求和限制以及对应的 QoS 类。

Rocky Linux 9 默认采用 cgroup v2，版本大于 4.5 的 linux 内核支持 cgroup v2，但是集成了高版本内核的 Linux 发行版不一定支持 cgroup v2。可采用下面的两个命令之一检查你正在使用的 linux 支持 cgroup 的版本：

```
[root@master1 ~]# stat -fc %T /sys/fs/cgroup/
cgroup2fs

[root@master1 ~]# mount | grep '/sys/fs/cgroup'
cgroup2 on /sys/fs/cgroup type cgroup2 (rw,nosuid,nodev,noexec,relatime)
```

说明支持 cgroup v2。

强烈建议采用 cgroup v2 版本，它比 v1 版本简洁多了。如果当前 linux 不支持 cgroup v2，可采用下面的方法开启 cgroup v2（确保存在红色字体的那一行）：

```
[root@master1 ~]# vim /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="cgroup_no_v1=all"
GRUB_DISABLE_RECOVERY="true"
GRUB_ENABLE_BLSCFG=true

[root@master1 ~]# grub2-mkconfig -o /boot/grub2/grub.cfg                    #重构 grub 配置
Generating grub configuration file ...
Adding boot menu entry for UEFI Firmware Settings ...
done
```

Linux 启动时会在/sys/fs/cgroup 目录上挂载一个 cgroup2 文件系统，因此我们就从这个目录的结构开始认识 cgroup2。

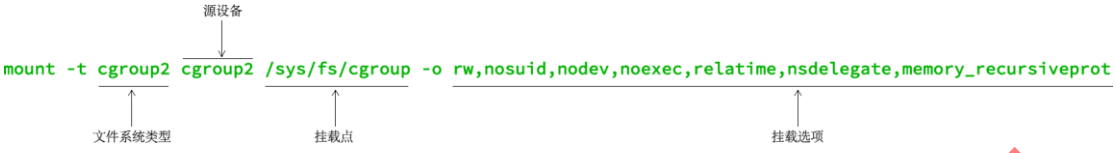
先看看挂载的文件系统树：

```
[root@master1 ~]# findmnt
TARGET                                SOURCE     FSTYPE     OPTIONS
/                                     /dev/sda1  xfs        rw,relatime,attr2,inode64,logbufs=8,logbsize=32k,noquota
├─/proc                               proc       proc       rw,nosuid,nodev,noexec,relatime
│ └─/proc/sys/fs/binfmt_misc          systemd-1  autofs
rw,relatime,fd=29,pgrp=1,timeout=0,minproto=5,maxproto=5,direct,pipe_ino=14864
├─/sys                                sysfs      sysfs      rw,nosuid,nodev,noexec,relatime
│ ├─/sys/kernel/security              securityfs securityfs rw,nosuid,nodev,noexec,relatime
│ ├─/sys/fs/cgroup                    cgroup2    cgroup2    rw,nosuid,nodev,noexec,relatime,nsdelegate,memory_recursiveprot
│ ├─/sys/fs/pstore                    pstore     pstore     rw,nosuid,nodev,noexec,relatime
│ ├─/sys/fs/bpf                       bpf        bpf        rw,nosuid,nodev,noexec,relatime,mode=700
│ ├─/sys/kernel/debug                 debugfs    debugfs    rw,nosuid,nodev,noexec,relatime
│ ├─/sys/kernel/tracing               tracefs    tracefs    rw,nosuid,nodev,noexec,relatime
│ ├─/sys/fs/fuse/connections          fusectl    fusectl    rw,nosuid,nodev,noexec,relatime
│ └─/sys/kernel/config                configfs   configfs   rw,nosuid,nodev,noexec,relatime
├─/dev                                devtmpfs   devtmpfs   rw,nosuid,size=4096k,nr_inodes=492442,mode=755,inode64
│ ├─/dev/shm                          tmpfs      tmpfs      rw,nosuid,nodev,inode64
│ ├─/dev/pts                          devpts     devpts     rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000
│ ├─/dev/hugepages                    hugetlbfs  hugetlbfs  rw,relatime,pagesize=2M
```

https://www.cnblogs.com/

```
│ └─/dev/mqueue                                      mqueue     mqueue     rw,nosuid,nodev,noexec,relatime
└─/run                                               tmpfs      tmpfs      rw,nosuid,nodev,size=796124k,nr_inodes=819200,mode=755,inode64
  ├─/run/credentials/systemd-tmpfiles-setup-dev.service none      ramfs      ro,nosuid,nodev,noexec,relatime,mode=700
  ├─/run/credentials/systemd-sysctl.service          none      ramfs      ro,nosuid,nodev,noexec,relatime,mode=700
  ├─/run/credentials/systemd-tmpfiles-setup.service  none      ramfs      ro,nosuid,nodev,noexec,relatime,mode=700
  └─/run/user/0                                      tmpfs      tmpfs      rw,nosuid,nodev,relatime,size=398060k,nr_inodes=99515,mode=700,inode64
```

注意上面红色字体那行，挂载点/sys/fs/cgroup、文件系统 cgroup2、源设备 cgroup2，以及众多的挂载参数 rw,nosuid,nodev,noexec,relatime,nsdelegate,memory_recursiveprot，因此可以推导出电脑启动时的挂载命令为：



我们还可以再次执行命令 mount -t cgroup2 cgroup2 ...在任何目录上挂载 cgroup2 文件系统，具体见后面的实战部分。

再看看/sys/fs/cgroup 的目录结构，参考图 7-2 所示。

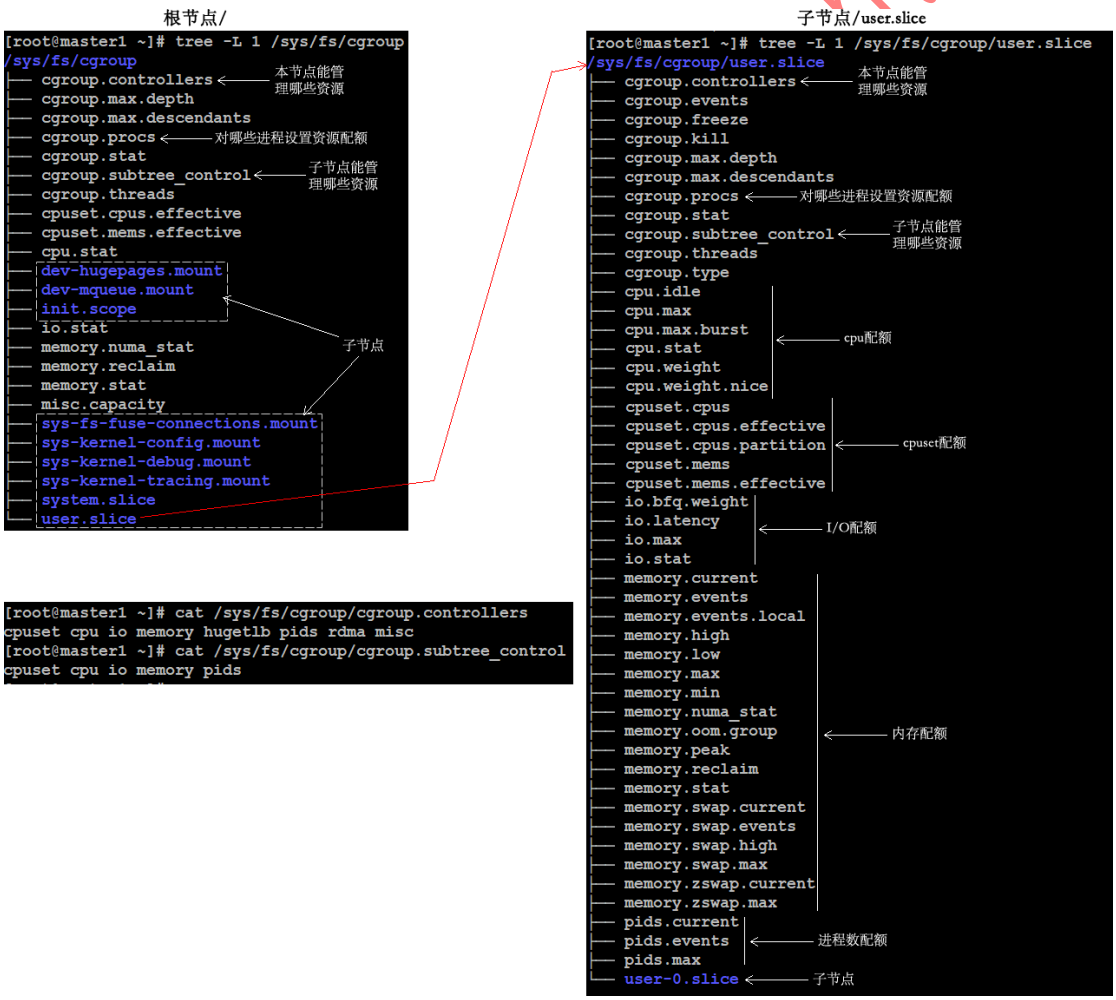


图 7-2 cgroup2 的节点结构

可以发现/sys/fs/cgroup 是一个目录树，目录里有文件和更多的目录。这里的目录我们称之为节点，其中 /sys/fs/cgroup 称为根节点，其他的目录都称为子节点，没有包含子节点的节点又称为叶子节点，X 节点的子节点及子节点的子节点……统称为 X 节点的后代节点，除了根节点和叶子节点之外的其他节点也称为中间节点，同一个父节点的节点称为兄弟节点。参考图 7-3 所示，左边是目录结构，右边是对应的节点树，红色字体为根节点，绿色字体

为叶子节点，其他的都为中间节点。

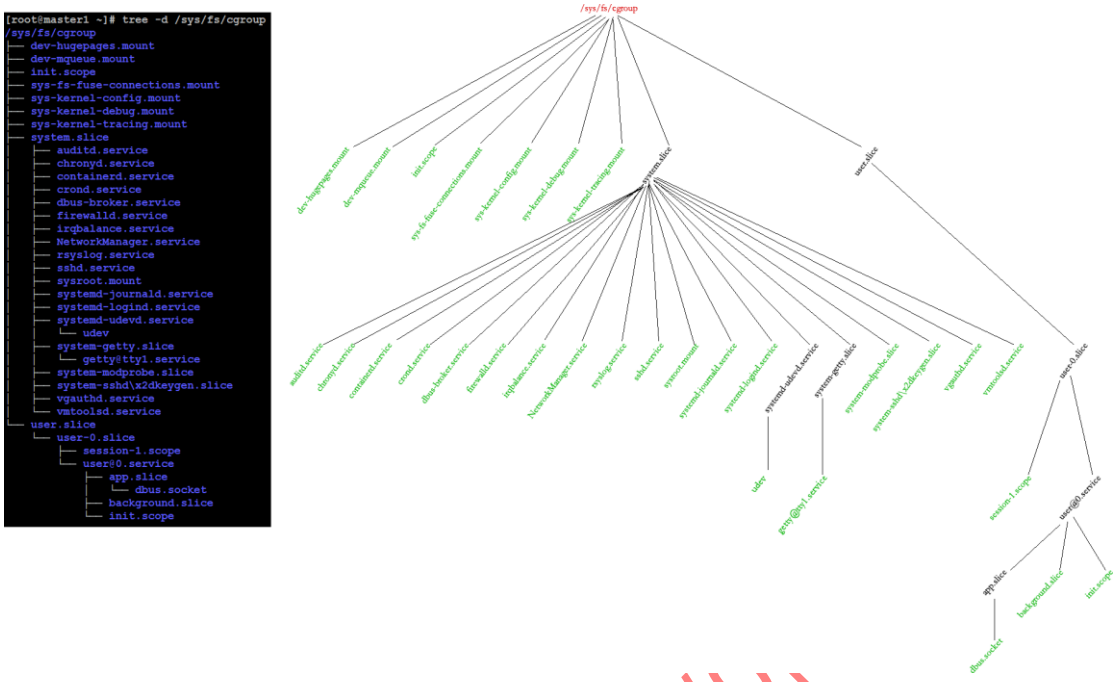


图 7-3 cgroup 的节点树

1．所有的节点中都包含以下几个重要文件：

cgroup.controllers：此文件包含本节点和后代节点需要设置配额的资源（用空格分隔的资源控制器），如根节点中的 cgroup.controllers 的内容为 cpuset cpu io memory hugetlb pids rdma misc，子节点 user.slice 中的 cgroup.controllers 的内容为 cpu memory，也就是说 user.slice 子节点只能对 cpu 和内存做配额限制。这是个只读文件。

cgroup.subtree_control：本文件包含子节点能做配额限制的资源控制器，必须是 cgroup.controllers 文件所列举的资源控制器集合的子集，与子节点中的 cgroup.controllers 文件具有相同的内容。

cgroup.procs：包含需要施行资源配额限制的进程号，一个进程号占一行，根节点中的 cgroup.procs 文件包含系统全部进程，但根节点有点特殊，不对任何进程做任何资源配额限制。一般都在叶子节点上对某些进程施行资源配额限制，所以中间节点的 cgroup.procs 为空文件。如果你要强行在某个中间节点（比如 α）上对进程做配额限制（即 cgroup.procs 文件不空），那么 α 节点的全部后代节点就不起作用了，因为如果 cgroup.procs 文件不空，那么 cgroup.subtree_control 文件必空，这样就导致后代节点不能施行任何资源限制了（因为没有资源控制器），这点要特别小心。注意文件中的进程号是无序的，一个进程只能绑定到一个节点，要么根本不绑定，可以采用命令“cat /proc/<进程号>/cgroup”查看进程绑定的节点，例如下面的例子查看当前进程（bash）绑定节点：

```
[root@master1 ~]# cat /proc/self/cgroup
0::/iol
```

即绑定到 cgroup 节点/sys/fs/cgroup/io1 上，即根节点就是/sys/fs/cgroup。

cgroup.max.descendants：本节点最多能拥有多少后代节点，默认值是 max，即没有限制。假如设为 2，那么最多拥有 2 个后代，要么两个子节点，要么一个子节点加一个孙节点。

cgroup.max.depth：本节点最多能拥有的后代辈数，默认值是 max（没有限制）。假如设为 1，那么最多只能有一代后辈，即儿子节点，没有孙辈了。

https://www.verycd

cgroup.stat：关于后代节点的统计信息，目前只统计了两方面的信息：nr_descendants（活跃后代的数量）、nr_dying_descendants（被用户删除即将被系统销毁的后代节点数量）。这是个只读文件。类似的其他*.stat 文件都是对后代节点统计某些方面的信息。

2. 对于非根节点，额外包含如下一些重要文件：

cgroup.type：定义本节点是针对进程还是线程做配额限制，可取的值有：domain（针对进程）、threaded（针对线程）、domain threaded（针对线程子树的根。一个线程可以繁衍很多子子孙孙线程）、domain invalid（无效状态。但可改为 threaded）。这个文件可读写。

cgroup.threads：对于线程节点，本文件中保存了需要施行资源配额的线程号，一个线程号占据一行。

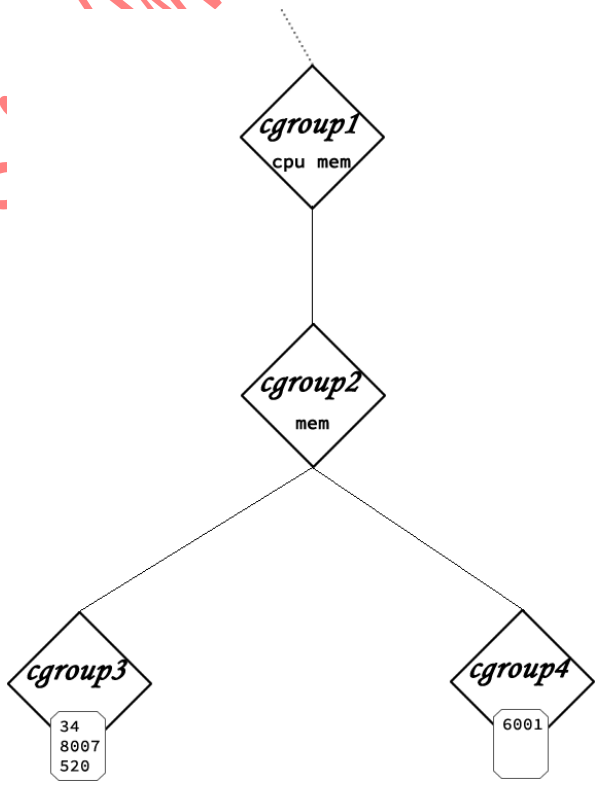
cgroup.freeze：读写单值文件，允许的值为“0”和“1”，默认值为“0”。将“1”写入文件会导致本节点和所有后代节点被冻结，这意味着所有所属进程都将停止（不是退出），直到节点被解冻。 完成“冻结”任务需要一些时间，当此操作完成后，另一个文件 cgroup.events 中的“frozen”值将被更新为“1”，并发出相应的通知。一个节点遭到冻结可能的原因有两个：自身或祖先节点的 cgroup.freeze 文件的内容被置为“1”。冻结节点中的进程还能接受诸如 SIGKILL 信号而退出，或者移出冻结节点后则会继续运行。如果一个进程被移入到冻结节点，它就会停止。把进程号加入节点的 cgroup.procs 文件中表示移入到节点，相反，把 cgroup.procs 文件中的某个进程号删除，就是移出。处于冻结状态的节点不会影响任何针对节点树的操作，比如删除冻结节点、创建新的子节点等。

cgroup.events：除非另有指定，否则更改此文件中的内容会产生文件修改事件。参数 populated 取值 1（本节点或后代节点包含活动进程），否则取值 0；参数 frozen 取值 1（如果节点被冻结），否则取值 0。默认取值看下面的命令：

```
[root@master1 ~]# more /sys/fs/cgroup/user.slice/cgroup.events
populated 1
frozen 0
```

cgroup.kill：往文件中写入 1 时，会向本节点和后代节点所属的全部进程发送 SIGKILL 信号，即进程立即退出。

最后我以图 7-4 来总结一下。



https://www.verypen.org

图 7-4 cgroup 实例

如图 6-4 所示，cgroup1 节点中限制了使用 cpu 及 内存资源，它将控制后代节点的 CPU 周期和内存分配（即，限制 cgroup2、cgroup3、cgroup4 中的 cpu 及内存资源分配）。cgroup2 中启用了内存限制，但是没有启用 cpu 的资源限制，这就导致了 cgroup3 和 cgroup4 的内存资源受 cgroup2 的内存配额的限制，cgroup3 和 cgroup4 会自由竞争在 cgroup1 节点中设定的 cpu 资源配额。

由此可见，cgroup 资源是自上而下分布约束的，只有当资源已经从先辈节点分发下来时，后辈节点才能进一步瓜分资源。所有非根节点的 cgroup.subtree_control 文件只能包含在父节点的 cgroup.subtree_control 文件中启用的资源控制器。

§7.1.2 资源控制器

本节对一些重要的资源控制器做个介绍，这些资源控制器包括：cpu、cpuset、memory、io、pids、rdma、hugetlb 和 misc，每个资源控制器通过节点中的若干个文件来配置，而一个节点中有没有相应资源控制器的配置文件是由父节点中的 cgroup.subtree_control 文件的内容决定的，比如父节点的 cgroup.subtree_control 文件中没有 cpu 资源控制器，那么本节点内就没有 cpu 资源控制器的配置文件。另外资源控制器的配置文件不会出现在根节点中，因为根节点是针对整台电脑的资源针对所有进程。

下面对介绍一些常用的控制器。

1. cpu 控制器

cpu 控制器管理进程消耗 cpu 资源的配额，具体通过文件 cpu.max、cpu.max.burst、cpu.stat、cpu.weight 和 cpu.weight.nice 等来管理，节点中有没有这些文件是有父节点中的文件 cgroup.subtree_control 的内容决定的，如果父节点中的 cgroup.subtree_control 文件内容没有 cpu 控制器，那么本节点根本没有这些文件。

cpu.weight：定义消耗 cpu 资源的权重，取值在[1,10000]之内，默认值 100。显然值越大允许消耗的 cpu 资源就越多。

cpu.weight.nice：定义消耗 cpu 资源的优先级，在[-20,19]范围取值，值越小优先级越高，默认值 0。cpu.weight 和 cpu.weight.nice 是两种设置优先级的方法，修改任何一个文件的内容会自动影响另一个文件的内容，请看下面的命令：

```
[root@master1 ~]# echo 650 > /sys/fs/cgroup/user.slice/cpu.weight
[root@master1 ~]# cat /sys/fs/cgroup/user.slice/cpu.weight.nice
-8
[root@master1 ~]# echo 10 > /sys/fs/cgroup/user.slice/cpu.weight.nice
[root@master1 ~]# cat /sys/fs/cgroup/user.slice/cpu.weight
11
```

cpu.max：文件内容只有一行，格式是$MAX $PERIOD，表示在 PERIOD 个时间周期内，最多可消耗$MAX 个时间周期。默认值为 max 100000，表示没有限制。其实相当于设定消耗 cpu 的百分数，即$MAX/$PERIOD%，不过允许 $MAX 大于$PERIOD，当大于 100%时其实就是没有限制了。

cpu.max.burst：使用之前未使用的积累下来的周期数，取值范围[0,$MAX]，默认值为 0。打个比方：我每月的手机流量是 10G，但是半年来每月实际使用的流量不到 6G，也就是说到了第七个月，我之前半年未使用完的流量累计有 24G 之多，第七个月我要看几部大片，所以尽管仍然限流 10GB，而且我还可以再使用之前未用完的流量，但是移动公

https://www.verypen.org

司不讲武德，说“你最多能额外再用 10G。”这时相当于 cpu.max.burst=10。如果之前积累下来的未用完的流量不到 10GB，比如 5G，那你七月最多只能用 15G。

cpu.stat：只读文件，当前节点内进程消耗 cpu 的统计信息：usage_usec（占用 cpu 总时间）、user_usec（用户态占用时间）、system_usec（内核态占用时间）、nr_periods（周期计数）、nr_throttled（周期内的限制计数）、throttled_usec（限制执行的时间）。

2. CPUSET 控制器

cpuset 控制器限制进程使用 cpu 核心。现代 cpu 一般都是多核 cpu，比如 2 核、4 核、8 核等，核心编号从 0 开始，比如 4 核 cpu 的核心编号为 0、1、2、3，可执行命令 cat /proc/cpuinfo 查看 cpu 各个核心的参数。允许把进程绑定在具体的核心上。

cpuset.cpus：列举能使用的 cpu 核，用逗号分开的一组数字，如 0-4,6,8-10。默认值为空，表示使用最近的 cpuset.cpus 文件非空的那个父辈节点所列出的 cpu 核。本文件允许修改。比如命令 echo 0,2 > cpuset.cpus 限制只能使用 0 和 2 号核心。

cpuset.cpus.effective：由父节点实际授予的真实可用的 cpu 核。如果 cpuset.cpus 文件非空，那么 cpuset.cpus.effective 就是 cpuset.cpus 的子集，否则就是由父节点授予的全部 cpu 核。本文件为只读文件。

cpuset.mems 和 **cpuset.mems.effective** 是针对 numa 架构的服务器做配额的。非统一内存访问（NUMA）是一种用于多处理器的电脑内存体设计，内存访问时间取决于处理器的内存位置，处理器访问它自己的本地存储器的速度比非本地存储器快一些。这两个文件的定义与 cpuset.cpus、cpuset.cpus.effective 一样。

cpuset.cpus.partition：文件只有一行，可取值 root、member 和 isolated，默认根节点是 root，其他节点都为 member。当某个节点（比如 α）为 root 时，就开始了一个新的分区，此分区由 α 及满足这些条件的后代节点组成，这些条件就是：cpuset.cpus.partition 为 member 且不在 root 或 isolated 的子树上，例如图 7-5 左侧部分，灰底的节点组成一个分区。

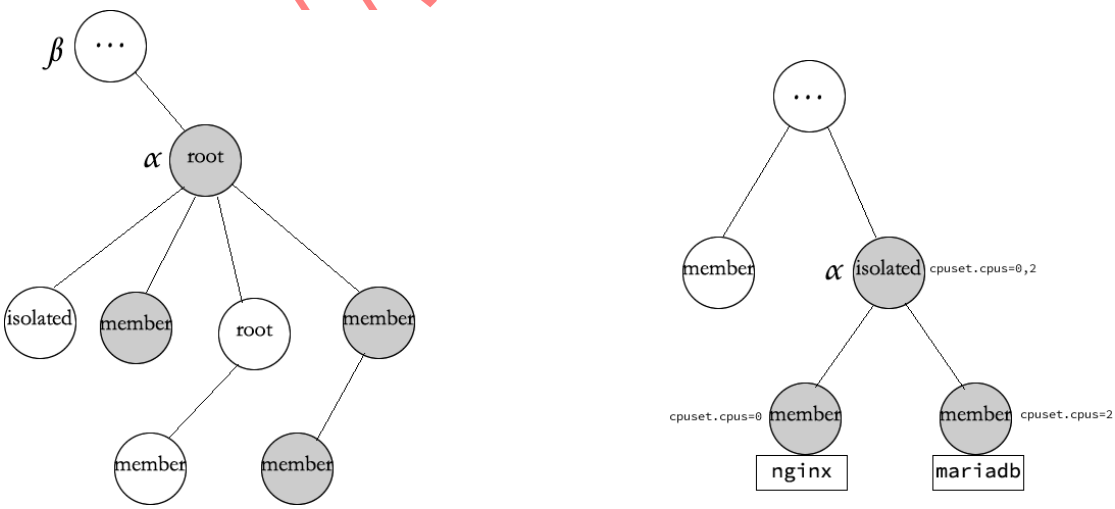


图 7-5 分区示意图

一个分区独占分配给它的 cpu 核，这些 cpu 核被列在分区根节点中的 cpuset.cpus 文件内，并从分区跟节点的父节点的 cpuset.cpus.effective 文件内删除这些 cpu 核。比如在图 4-5 中，假设 α 节点的 cpuset.cpus 文件内容为 2，那么 β 节点的 cpuset.cpus.effective 文件内容就不能包含 2。当某个节点的 cpuset.cpus.partition 文件的内容为 isolated 时，与 root 值一样，开辟了一个新的分区，但是授予的 cpu

www.verycpen.org

核在本分区内不会由系统均衡调度，而是要人为调度。所以要特别小心，如果人为调度不当，可能会导致某个 cpu 核特别忙碌，而另一个 cpu 核又特别空闲的现象。比如在图 4-5 右侧部分所示，分配给 α 分区 0 和 2 两个 cpu 核，然后人为指定 nginx 网站服务进程只在 0 号 cpu 核上运行，而 mariadb 数据库服务进程只在 2 号 cpu 核上运行，尽管 nginx 和 mariadb 都是多进程服务，但是 mariadb 的进程不能在 0 号 cpu 核上运行，尽管可能 0 号 cpu 核很空闲。

当显示 cpuset.cpus.partition 文件的内容时，可能的结果有：root、isolated、member、root invalid (<分区根无效的原因>)、isolated invalid (<孤立分区根无效的原因>)。比如下面例子：

```
[root@master1 ~]# cat /sys/fs/cgroup/cg2/cpuset.cpus.partition
root invalid (cpuset.cpus is empty)

[root@master1 ~]# echo 1 > /sys/fs/cgroup/cg2/cpuset.cpus
[root@master1 ~]# cat /sys/fs/cgroup/cg2/cpuset.cpus.partition
root
```

第一条命令返回的结果表明 cpuset.cpus 文件是空的，所以不能成为分区根，第二条命令授予 1 号 cpu 核，这样再显示/sys/fs/cgroup/cg2/cpuset.cpus.partition 文件内容，就是合法的分区根了。

成为合法分区的根节点必须满足以下几个条件：

(1) cpuset.cpus 的内容与兄弟节点互斥。
(2) 父节点是一个有效的分区根节点。
(3) cpuset.cpus 不能为空，且为父节点 cpuset.cpus 的子集。
(4) cpuset.cpus.effective 不能为空，除非分区没有绑定进程。

当把一个节点由 root 改为 member 时要特别小心，因为这种改动会导致子分区（如果存在）无效，从而影响到绑定的进程运行异常。

3. memory 控制器

内存控制器用来对进程做内存资源配额限制。具体通过文件 memory.current、memory.events.local、memory.low、memory.min、memory.oom.group、memory.reclaim、memory.swap.current、memory.swap.high、memory.zswap.current、memory.events、memory.high、memory.max、memory.numa_stat、memory.peak、memory.stat、memory.swap.events、memory.swap.max 和 memory.zswap.max 来管理。

所有内存计数单位为字节，如果设置的数字不是内存页大小的整数倍，就会被自动往下调整为整数倍，linux 系统中内存页的大小是 4k，即 4096 字节。比如你设置 memory.min 为 8000，那么最终被自动调整为 4096。

memory.max：使用内容的上限，不能超过这个值，默认值是 max，即没有限制。

memory.min：这是内存的硬保护机制。如果当前节点的内存使用量在 min 值以内，则任何情况下都不会对这部分内存进行回收。默认值为 0。

memory.high：内存使用的高位阈值，当超出这个高位阈值时，内核会使用各种手段回收内存，从而设法保持内存的用量在高位阈值以下。默认值为 max。

memory.low：内存使用如果低于这个值，则内存将尽量不被回收，除非无法从任何未受保护的节点回收内存。这是一种是尽力而为的内存保护措施，是“软保护”。默认值为 0。

min、low、high 和 max 这个四个值的关系参考图 7-6。

https://www.veryopen.org

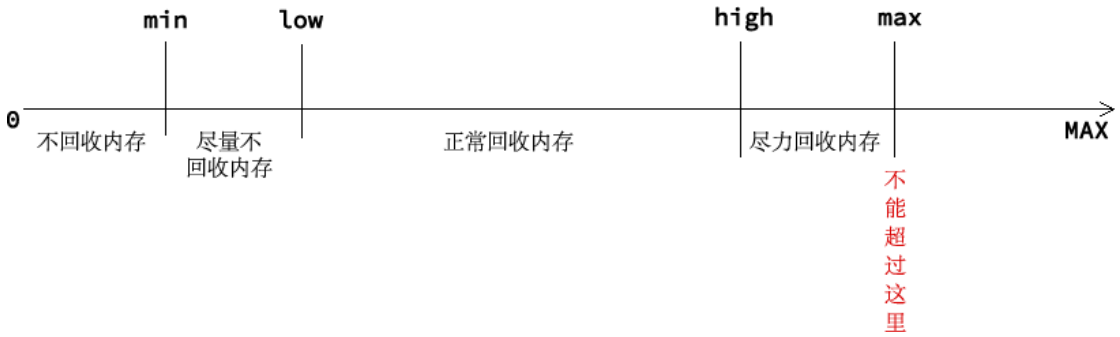


图 7-6 min、low、high 和 max 的关系

memory.current：本节点及后代节点使用内存的总数，以字节为单位，是内存页大小的整数倍。linux 的内页大小为 4k（4096 个字节）。

memory.peak：自从启用本节点以来消耗的内存峰值。

memory.reclaim：只写文件，触发本节点的内存回收机制。如 echo "1G" > memory.reclaim 回收 1G 的内存。

memory.swap.max：能使用的交换区上限。

memory.swap.high：交换区高位阈值，

memory.swap.current：本节点及后代节点使用交换区的总数，以字节为单位。

memory.zswap.max：内存压缩区的使用量。zswap 是 Linux 内核实现的内存压缩机制，当物理内存不够时，传统的做法是把部分内存页内容换出到磁盘（交换区），等以后要用到这些被换出的内容时再换入。读写磁盘相对于读写内存来说要慢得多，所以当电脑开始使用交换区时，就很卡。内存压缩技术就是尽量不换出内存，而是通过压缩拟换出内存页的内容来腾出更多的内存。大部分 linux 发行版，默认是没有启用 zswap 的。具体怎么启用请参考§2.1 小节。

memory.zswap.current：当前内存压缩区的实际使用量。

4. IO 控制器

io 控制器主要针对磁盘做配额限制，磁盘 io 的调度器不同，IO 控制器的配置文件名也有所不同。现在流行的磁盘 io 调度器有 none、mq-deadline、kyber、cfq 和 bfq，磁盘 io 调度器任务是如何更高效的读写硬盘。每个调度器各有适合的磁盘接口类型，比如现在逐渐流行的 NVMe 接口的固态盘来说，none 调度器（其实是不用调度器）是最合适的，而 bfq 最适合当下的 sata 接口类型的磁盘。可用命令 dmesg | grep "io scheduler"查看操作系统加载了哪些调度器，采用命令 cat /sys/block/sda/queue/scheduler 查看 sda 磁盘使用哪个调度器（显示结果中的中括号内的调度器），如果要临时修改调度器，可采用命令 echo bfq > /sys/block/sda/queue/scheduler，即采用 bfq 调度器。如果要永久修改，可采用命令 grubby --grub2 --update-kernel=ALL --args="elevator=bfq"，然后重启电脑即可。

io.weight：对磁盘 io 的权重隔离。文件内容为若干行，每行类似“键 值”这样的格式，“键”可为“default”（默认）或者“主设备号:次设备号”，而“值”的取值范围是[1,10000]。默认内容只有一行，即 defaut 100。权重越大读取磁盘的优先级也就越高。命令 lsblk 可查看硬盘的主/次设备号，参考下面的例子：

```
[root@master1 ~]# lsblk
NAME    MAJ:MIN   RM   SIZE RO TYPE  MOUNTPOINTS
```

https://www.veryopen.org

```
sda       8:0    0    32G  0 disk
└─sda1    8:1    0    32G  0 part  /
sr0      11:0    1   9.1G  0 rom
```

采用命令 echo 200 > io.bfq.weight 或者 echo default 200 > io.bfq.weight 修改默认权重为 200。

io.bfq.weight：专门针对 bfq 调度器，用法与 io.weight 相同。

io.max：同时实现了 BPS（读写磁盘缓存）和 IOPS（读写磁盘）的流量限制。文件包含若干行，每行的格式为“主设备号:次设备号 参数=值 参数=值 ...”，“主设备号:次设备号”指定哪块硬盘，不能指定磁盘分区，“参数”可为：rbps（每秒最多读磁盘的字节数）、wbps（每秒最多写磁盘的字节数）、riops（每秒最多读磁盘的次数）、wiops（每秒最多写磁盘的次数），“值”为具体的数字或者 max，如果不做限制，就置为 max。注意：io.max 只控制磁盘的读写，读写磁盘缓存无效。看例子：

```
[root@master1 ~]# mkdir /sys/fs/cgroup/cg2 && cd /sys/fs/cgroup/cg2
[root@master1 cg2]# echo "8:0 rbps=2097152 wiops=120" > io.max                    #每秒至多读 2M,写 120
次
[root@master1 cg2]# more io.max
8:0 rbps=2097152 wbps=max riops=max wiops=120
[root@worker3 cg2]# echo "11:0 riops=90 rbps=1572864" > io.max                   #每秒至多读 90 次，读
1.5M
[root@worker3 cg2]# more io.max
11:0 rbps=1572864 wbps=max riops=90 wiops=120
8:0 rbps=2097152 wbps=max riops=max wiops=120
```

io.stat：磁盘 io 的统计信息，每行的格式为“主设备号:次设备号 参数=值 参数=值 ...”，其中参数有：rbytes（一共读了多少字节）、wbytes（一共写了多少字节）、rios（一共读了多少次）、wios（一共写了多少次）、dbytes（因超过配额限制而丢失的字节数）、dios（因超过配额限制而阻止的读写磁盘次数）。如：

```
8:16 rbytes=1459200 wbytes=314773504 rios=192 wios=353 dbytes=0 dios=0
8:0 rbytes=90430464 wbytes=299008000 rios=8950 wios=1252 dbytes=50331648 dios=3021
```

5. PIDS 控制器

pids 控制器限制能产生的进程数量。

pids.max：当前节点机器后代节点能产生的最大进程数量，默认值为 max，表示没有限制。执行命令 echo 100 > pids.max，则限制进程数 100，当进程数已达 100，如果还要产生进程，就被阻止。

pids.current：当前节点机器后代节点中的进程数。

参考网站：https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html

https://www.verypeng

§7.2 cgroup2 实战

为了不影响 linux 操作系统正在使用的 cgroup2（/sys/fs/cgroup 目录），我们单独在另一个目录上挂载 cgroup2 文件系统。下面的命令在/opt/mycgroup 目录上挂载 cgroup2 文件系统：

```
[root@master1 ~]# mkdir /opt/mycgroup2
[root@master1 ~]# mount -t cgroup2 cgroup2 /opt/mycgroup
[root@master1 ~]# findmnt cgroup2 ------------------------------------------------------------------#显示已挂载的 cgroup2 文件系统
TARGET         SOURCE  FSTYPE  OPTIONS
/sys/fs/cgroup cgroup2 cgroup2 rw,nosuid,nodev,noexec,relatime
/opt/mycgroup  cgroup2 cgroup2 rw,relatime
```

安装 c 语言程序编译器：

```
[root@master1 ~]# dnf install gcc glibc-static -y
```

§ 7.2.1 cpu 配额

给进程施行 cpu 配额主要通过 cpu 和 cpuset 两个控制器完成，即重点通过 cpu.max、cpuset.cpus 和 cpuset.cpus.partition 文件做配额限制。

```
[root@master1 ~]# cat /opt/mycgroup/cpu1/cpu.max
max 10000
[root@master1 ~]# echo "5000 10000" > /opt/mycgroup/cpu1/cpu.max ----------------------#最多能使用 50%的 cpu 时钟周期数
[root@master1 ~]# cat /opt/mycgroup/cpu1/cpu.max
5000 10000
[root@master1 ~]# echo 1 > /opt/mycgroup/cpu1/cpuset.cpus ---------------------------------- #安排在 1 号 cpu 核上运行

[root@master1 ~]# vim cpu1.c
#include <stdio.h>

int main()
{
        for(;;){
                ;
        }
}

[root@master1 ~]# gcc -o cpu1 cpu1.c ------------------------------------------------------------#编译 c 语言，产生可执行程序 cpu1

[root@master1 ~]# ./cpu1 & ------------------------------------------------------------------------------------ #在后台运行 cpu1
[1] 79562

[root@master1 ~]# echo 79562 > /opt/mycgroup/cpu1/cgroup.procs                                          #绑定进程
[root@master1 ~]# cat /opt/mycgroup/cpu1/cgroup.procs
79562

[root@master1 ~]# top                                                                  #运行 top 命令后按 1 查看每个 cpu 核的使用率
top - 16:19:39 up 36 min,  1 user,  load average: 0.60, 0.40, 0.24
Tasks: 210 total,   2 running, 208 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 47.1 us,  0.0 sy,  0.0 ni, 52.6 id,  0.0 wa,  0.3 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.3 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3887.3 total,   3213.6 free,    522.5 used,    378.6 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3364.8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  79580 root      20   0    2496    964    872 R  46.2   0.0   0:36.28 cpu1
    784 root      20   0 1504060  53980  29412 S   0.3   1.4   0:10.27 containerd
      1 root      20   0   21420  12652   9744 S   0.0   0.3   0:00.95 systemd
......
```

按 q 退出 top 命令。虽然配额上限是 50%，但实际情况是小于但接近 50%。我们可以修改 /opt/mycgroup/cpu1/cpu.max 和/opt/mycgroup/cpu1/cpuset.cpus 文件的内容，再运行 top 命令观察实验结果。

接下来再看看 cpuset.cpus.partition 文件，我们构造一个只包含根和一个叶子节点的分区，参考下面的命令：

```
[root@master1 ~]# echo root > /opt/mycgroup/cpu1/cpuset.cpus.partition
[root@master1 ~]# cat /opt/mycgroup/cpu1/cpuset.cpus.partition
root
[root@master1 ~]# echo "1,3">/opt/mycgroup/cpu1/cpuset.cpus
[root@master1 ~]# cat /opt/mycgroup/cpu1/cpuset.cpus
1,3
[root@master1 ~]# echo "7000 10000" > /opt/mycgroup/cpu1/cpu.max ------------------------------------ #不限制 cpu
[root@master1 ~]# mkdir /opt/mycgroup/cpu1/leaf
[root@master1 ~]# vim cpu2.c
#include <stdio.h>
#include <unistd.h>
```

```
int main()
{
        pid_t child_id;
        sleep(60);
        child_id = fork();
        if(child_id==0){
                //子进程
                while(1){};
        }else if(child_id>0){
                //父进程
                for(;;);
        }else{
                printf("for failed!\n");
        }
}

[root@master1 ~]# ./cpu2 &
[1] 79725
[root@master1 ~]# echo 79725 > /opt/mycgroup/cpu1/leaf/cgroup.procs
[root@master1 ~]# top
top - 17:22:29 up  1:39,  1 user,  load average: 0.01, 0.03, 0.09
Tasks: 210 total,   3 running, 207 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 30.3 us,  0.0 sy,  0.0 ni, 69.4 id,  0.0 wa,  0.3 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 33.3 us,  0.0 sy,  0.0 ni, 66.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3887.3 total,   3204.3 free,    526.6 used,    383.9 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3360.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  79727 root      20   0    2496     88      0 R  30.9   0.0   0:02.00 cpu2
  79725 root      20   0    2496    960    872 R  28.6   0.0   0:01.72 cpu2
  79563 root      20   0       0      0      0 I   0.3   0.0   0:00.29 kworker/3:0-events_freezable_power_
      1 root      20   0   21420  12652   9744 S   0.0   0.3   0:00.96 systemd
    ......
```

按 q 退出 top，在运行命令 fg 把 cpu1 切换到前台，然后按 ctrl+c 终止 cpu2 程序。接下来再实验 isolated 分区：

```
[root@master1 ~]# echo isolated > /opt/mycgroup/cpu1/cpuset.cpus.partition
[root@master1 ~]# ./cpu2 &
[1] 79786
[root@master1 ~]# echo 79786 > /opt/mycgroup/cpu1/leaf/cgroup.procs
[root@master1 ~]# top
top - 19:55:23 up  2:06,  2 users,  load average: 1.45, 0.83, 0.58
Tasks: 213 total,   3 running, 210 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 59.9 us,  0.0 sy,  0.0 ni, 39.7 id,  0.0 wa,  0.3 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3887.3 total,   3189.9 free,    540.4 used,    384.9 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3347.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  79786 root      20   0    2496    960    872 R  29.7   0.0   0:45.52 cpu2
  79788 root      20   0    2496     88      0 R  29.4   0.0   0:45.50 cpu2
      1 root      20   0  111544  12824   9872 S   0.0   0.3   0:00.99 systemd
......
```

https

§7.2.2 内存配额

现在我们来实战 cgroup2 的内存控制器，进一步探究这几个配置文件：memory.low、memory.min、memory.high、memory.max。

首先按图 7-7 所示设置几个配置文件的内容。

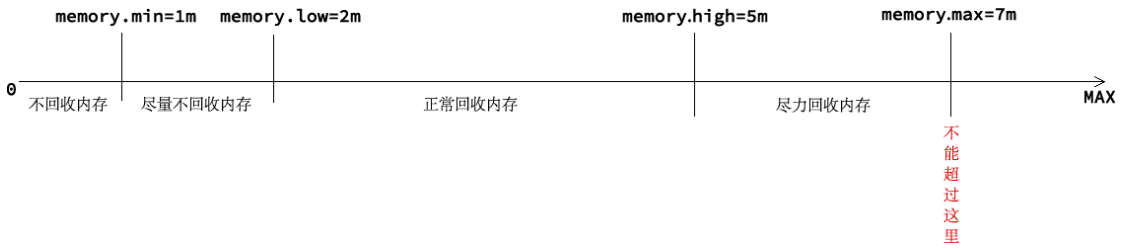


图 7-7 内存配额

```
[root@master1 ~]# shutdown -r now                                                    #使前一次的实战全部作废
[root@master1 ~]# mkdir -p /opt/mycgroup
[root@master1 ~]# mount -t cgroup2 cgroup2 /opt/mycgroup/
[root@master1 ~]# mkdir /opt/mycgroup/mem1 ------------------------------------------------------------------ #创建子节点 mem1
[root@master1 ~]# echo 1048576 > /opt/mycgroup/mem1/memory.min
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.min
1048576
[root@master1 ~]# echo 2097152 > /opt/mycgroup/mem1/memory.low
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.low
2097152
[root@master1 ~]# echo 5242880 > /opt/mycgroup/mem1/memory.high
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.high
5242880
[root@master1 ~]# echo 7340050 > /opt/mycgroup/mem1/memory.max
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.max
7340032
```

大家注意到没有，明明给 memory.max 设置了 7340050，为什么显示出来的结果是 7340032 呢？那是因为 7340050 不是内存页（4086 字节）的整数倍，所以自动往小调整为 7340032（刚好为 4096 的整数倍）。

绑定当前 bash 程序到 cgroup 节点 mem1：

```
[root@master1 ~]# echo $$ > /opt/mycgroup/mem1/cgroup.procs
```

$是一个特殊变量，保存了当前进程号，而$$就是引用$变量的值。执行命令“ps”可查看当前 bash 程序的进程号，再运行命令“echo $$”，看看结果是不是一样。

看看现在 bash 进程用了多少内存：

```
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
450560
```

不大，才四百多 KB。

下面我们来创建一个 10MB 的内存硬盘，即 RAMDISK 磁盘，然后往内存硬盘里拷贝文件或删除文件，网内存磁盘拷贝文件就会消耗内存，删除文件就会释放内存。因为当前 bash 被绑定到 cgroup 节点 mem1 上，所以我接下来的操作都会影响 bash 进程消耗的内存。

```
[root@master1 ~]# cd                                                                 #回到家目录
[root@master1 ~]# mkdir mem-disk
[root@master1 ~]# mount -t tmpfs -o size=10m ramdisk ~/mem-disk                      #挂载 10 兆的 tmpfs 文件系统到 mem-disk 目录
[root@master1 ~]# df -T
Filesystem     Type     1K-blocks    Used Available Use% Mounted on
devtmpfs       devtmpfs      4096       0      4096   0% /dev
tmpfs          tmpfs      1990300       0   1990300   0% /dev/shm
tmpfs          tmpfs       796124    9188    786936   2% /run
/dev/sda1      xfs       33487872 2450620  31037252   8% /
tmpfs          tmpfs       398060       0    398060   0% /run/user/0
ramdisk        tmpfs        10240       0     10240   0% /root/mem-disk
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
561152
```

好了，现在有一个 10 兆的磁盘挂在家目录中的 mem-disk 目录上，由于磁盘是空的，所以还没有消耗更多的内存。

```
[root@master1 ~]# cp /var/log/messages mem-disk/
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
2220032
[root@master1 ~]# dd if=/dev/zero of=mem-disk/file.dat bs=1024 count=1024                    #产生 1m 的文件 file.dat
1024+0 records in
1024+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.000854513 s, 1.2 GB/s
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
3084288
[root@master1 ~]# ls -l mem-disk/
total 1768
```

https://www.verydoc

```
-rw-r--r-- 1 root root 1048576 Jul 25 08:12 file.dat
-rw------- 1 root root  761322 Jul 25 08:09 messages
```

继续往磁盘拷贝大文件：

```
[root@master1 ~]# find /usr/lib64 -mount -size -5M -a -size +3M                    #查找小于 5m 大于 3m 的文件
/usr/lib64/libpython3.9.so.1.0
/usr/lib64/systemd/libsystemd-shared-252.so
[root@master1 ~]# cp /usr/lib64/systemd/libsystemd-shared-252.so mem-disk/
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
6037504
[root@master1 ~]# ls -l mem-disk/
total 5324
-rw-r--r-- 1 root root 1048576 Jul 25 08:12 file.dat
-rwxr-xr-x 1 root root 3640312 Jul 25 08:20 libsystemd-shared-252.so
-rw------- 1 root root  761322 Jul 25 08:09 messages
```

现在消耗的内存介于 memory.high 和 memory.max 之间了，所以现在执行命令比较慢，因为系统都在设法回收内存。下面再往磁盘拷贝一个大文件，看看能否成功：

```
[root@master1 ~]# cp /usr/lib64/libpython3.9.so.1.0 mem-disk/
^C                                                                  #命令一直在等待，按 ctrl+c 强行中断
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
7147520
[root@master1 ~]# ls -l mem-disk/
total 6348
-rw-r--r-- 1 root root 1048576 Jul 25 08:12 file.dat
-rwxr-xr-x 1 root root 1048576 Jul 25 08:51 libpython3.9.so.1.0
-rwxr-xr-x 1 root root 3640312 Jul 25 08:20 libsystemd-shared-252.so
-rw------- 1 root root  761322 Jul 25 08:09 messages
```

接下来开始删除磁盘中的文件：

```
[root@master1 ~]# rm -rf mem-disk/libpython3.9.so.1.0
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
6082560
[root@master1 ~]# rm -rf mem-disk/libsystemd-shared-252.so
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
2445312
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
2363392
[root@master1 ~]# rm -rf mem-disk/file.dat
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
1490944
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
1474560
[root@master1 ~]# rm -rf mem-disk/messages
[root@master1 ~]# cat /opt/mycgroup/mem1/memory.current
720896
```

§7.2.3 磁盘 IO 配额

下面我们通过实战来加深理解 io 控制器的 io.max 文件，此文件包含若干行，每行的格式为“主设备号:次设备号 参数=值 参数=值 ...”，其中参数有：rbps（每秒最多读磁盘的字节数）、wbps（每秒最多写磁盘的字节数）、riops（每秒最多读磁盘的次数）、wiops（每秒最多写磁盘的次数）。由于存在磁盘缓存，第一次读取磁盘数据时会从物理磁盘读取，同时放入磁盘缓存中，当第二次读同样的数据时直接从磁盘缓存获取数据，而不要再次访问物理磁盘。另外写磁盘页遵循大致的过程：先写入磁盘缓存，到了何时的时机再写入物理磁盘。io.max 只控制读写物理磁盘，所以做实验时要特别小心。

1. 准备实验：

```
[root@master1 ~]# shutdown -r now
[root@master1 ~]# mkdir -p /opt/mycgroup
[root@master1 ~]# mount -t cgroup2 cgroup2 /opt/mycgroup
[root@master1 ~]# mkdir /opt/mycgroup/io1
[root@master1 ~]# echo $$ > /opt/mycgroup/io1/cgroup.procs
```

2．磁盘读限速实战

下面的实验限制读磁盘最多每秒 500K：

```
[root@master1 ~]# lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
sda      8:0    0   32G  0 disk
└─sda1   8:1    0   32G  0 part /
sr0     11:0    1  9.1G  0 rom
[root@master1 ~]# echo "8:0 rbps=512000" > /opt/mycgroup/io1/io.max                    #每秒最多读500KB
[root@master1 ~]# cat /opt/mycgroup/io1/io.max
8:0 rbps=512000 wbps=max riops=max wiops=max
[root@master1 ~]# sync                                                                #把磁盘缓存中的脏数据写入磁盘
[root@master1 ~]# echo 3 > /proc/sys/vm/drop_caches                                   #释放全部磁盘缓存
[root@master1 ~]# dd if=/boot/vmlinuz-5.14.0-333.el9.x86_64 of=/dev/null
24498+1 records in
24498+1 records out
12543240 bytes (13 MB, 12 MiB) copied, 24.5049 s, 512 kB/s
```

上面最后一条命令就是读/boot/vmlinuz-5.14.0-333.el9.x86_64 文件的内容并写到/dev/null 文件中，而/dev/null 文件相当于一个黑洞，即使往里面写太多的内容，也都消失得无影无踪。

注意上面最后一条命令，Linux 系统会把/boot/vmlinuz-5.14.0-333.el9.x86_64 文件放入磁盘缓存中。下面再次运行次命令看看效果如下：

```
[root@master1 ~]# dd if=/boot/vmlinuz-5.14.0-333.el9.x86_64 of=/dev/null
24498+1 records in
24498+1 records out
12543240 bytes (13 MB, 12 MiB) copied, 0.0118553 s, 1.1 GB/s
```

每秒能读 1.1GB 的数据，那是因为直接读磁盘缓存，所以 io.max 的限制无效。运行命令“echo 3 > /proc/sys/vm/drop_caches”先清空磁盘缓存，再来运行 dd 命令就限速了。

下面的实验限制每秒最多读 8 次：

```
[root@master1 ~]# echo "8:0 rbps=max riops=8" > /opt/mycgroup/io1/io.max              #每秒最多读20次
[root@master1 ~]# cat /opt/mycgroup/io1/io.max
8:0 rbps=max wbps=max riops=8 wiops=max
[root@master1 ~]# sync                                                                #把磁盘缓存中的脏数据写入磁盘
[root@master1 ~]# echo 3 > /proc/sys/vm/drop_caches                                   #释放全部磁盘缓存
[root@master1 ~]# dd if=/dev/sda of=/dev/null bs=512 count=10240
10240+0 records in
10240+0 records out
5242880 bytes (5.2 MB, 5.0 MiB) copied, 9.99994 s, 524 kB/s
[root@master1 ~]# sync
[root@master1 ~]# echo 3 > /proc/sys/vm/drop_caches
[root@master1 ~]# cat /boot/vmlinuz-5.14.0-333.el9.x86_64 > /dev/null
```

由于每秒读磁盘的次数不能超过 8 次，所以上面最后一条命令会执行很长时间。

3．磁盘写限速实战

为了测试磁盘写，我们需要一块新的硬盘，关闭虚拟机，添加一块 4G 的 sata 硬盘。

下面的实验限制每秒最多写 100KB：

```
[root@master1 ~]# lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
sda      8:0    0   32G  0 disk
└─sda1   8:1    0   32G  0 part /
sdb      8:16   0    4G  0 disk
sr0     11:0    1  9.1G  0 rom
[root@master1 ~]# echo "8:0 wbps=102400" > /opt/mycgroup/io1/io.max                   #每秒最多写100KB
[root@master1 ~]# cat /opt/mycgroup/io1/io.max
8:0 rbps=max wbps=max riops=max wiops=max
[root@master1 ~]# sync                                                                #把磁盘缓存中的脏数据写入磁盘
[root@master1 ~]# echo 3 > /proc/sys/vm/drop_caches                                   #释放全部磁盘缓存
[root@master1 ~]# dd if=/dev/zero of=~/zero.dat bs=512 count=10240                    #产生一个5m的文件/tmp/zero.dat
10240+0 records in
10240+0 records out
```

```
5242880 bytes (5.2 MB, 5.0 MiB) copied, 0.00898597 s, 583 MB/s
```

什么情况？没有限速！那是因为写到磁盘缓存了，linux 系统会在合适的时机再写到磁盘。下面我们直接写第二块硬盘试试：

```
[root@master1 ~]# echo "8:16 wbps=102400" > /opt/mycgroup/io1/io.max                    #每秒最多写 100KB
[root@master1 ~]# cat /opt/mycgroup/io1/io.max
8:16 rbps=max wbps=max riops=max wiops=max
[root@master1 ~]# sync                                                                  #把磁盘缓存中的脏数据写入磁盘
[root@master1 ~]# echo 3 > /proc/sys/vm/drop_caches                                     #释放全部磁盘缓存
[root@master1 ~]# dd if=/dev/zero of=/dev/sdb bs=512 count=10240                        #向磁盘写入 5m 的数据
10240+0 records in
10240+0 records out
5242880 bytes (5.2 MB, 5.0 MiB) copied, 51.3132 s, 102 kB/s
```

https://www.veryopen.org

第8章 名字空间



名字空间由 namespace 翻译而来，用来隔离对象，因此在不同的名字空间中允许存在相同名字的对象，比如 windows 上的目录用来隔离文件对象，两个目录中允许存在相同名字的文件。在 Linux 系统中，存在若干类型的对象，每一种类型的对象存在对应的名字空间，一共有 9 类名字空间，它们分别是网络名字空间、用户名字空间、文件系统名字空间、进程名字空间、主机名字空间、时钟名字空间、控制组名字空间、ipc 名字空间和挂载点名字空间。除了控制组名字空间已在第 7 章讲解了外，其他名字空间都在本章介绍。

一个网络名字空间内的对象有网卡名字、ip、路由表、localhost、DNS、防火墙策略、网关、网络掩码等，两个不同的网络名字空间拥有各自的对象，不用担心名字冲突，比如正在运行的容器就创建了自己的网络空间，所以容器有自己的 localhost 设备、网卡名称和 ip 等。

本章我们一起来学习 Linux 系统中的九大名字空间，在学习过程中需要实战练习，所以还得先学习两个命令：unshare 和 nsenter。

https://www.veryopen.org

§8.1 unshare/nsenter 命令

§8.1.1 unshare 命令用法

unshare 在新的名字空间中运行程序，当程序退出后 unshare 也退出。此命令比 chroot 命令要强大得多，能隔离 linux 系统中的全部名字空间。它的使用语法如图 8-1 所示。

```
unshare [参数] [程序 [程序的参数] ]
-m, --mount[=<file>]      #隔离挂载点名字空间。如果指明了file，则通过绑定一个挂载点创建持久名字空间，当umount file后销毁名字空间。
-u, --uts[=<file>]        #隔离主机名名字空间。
-i, --ipc[=<file>]        #隔离进程间通信名字空间。
-n, --net[=<file>]        #隔离网络名字空间。
-p, --pid[=<file>]        #隔离进程名字空间。如果指定file，那么也必须指定--fork参数。
-U, --user[=<file>]       #隔离用户和组名字空间。
-C, --cgroup[=<file>]     #隔离cgroup名字空间。
-T, --time[=<file>]       #隔离时间名字空间。一般再指定--monotonic或--boottime定义相应的时钟偏移。
-f, --fork                #unshare产生一个子进程，并在子进程中执行程序。一般与-p参数结合使用。
 --keep-caps              #当结合--user 参数时，确保在用户命名空间中授予的权能保留在子进程中。
--kill-child[=signame]    #当unshare终止时发给子进程的信号，默认信号为SIGKILL。隐式带有-p和-f参数。
--mount-proc[=mountpoint] #在运行程序前，把proc文件系统挂载到mountpoind上（默认是/proc），一般结合-p参数，同时隐式带有-m参数。
                            必须结合-f或--kill-child参数。
--map-user=uid|name       #以有效用户uid运行程序，隐式带有-U参数。
--map-group=gid|name      #以有效组gid运行程序，隐式带有-U和--setgroups=deny参数。
-r, --map-root-user       #以新用户名字空间中的root用户运行程序，以便于执行特权操作（如配置网卡）。相当于--map-user=0 --map-group=0，
                            本参数隐式带有--setgroups=deny和--user参数。
-c, --map-current-user    #以当前用户的uid运行程序（不过这个uid为新用户名字空间中对应的用户），本参数等价于
                            --map-user=$(id -ru) --map-group=$(id -rg)，隐式带有参数--setgroups=deny和--user参数。
--setgroups allow|deny    #在新的用户名字空间中允许或禁止调用setgroup。
-R, --root=dir            #dir作为新根目录运行程序。
-w, --wd=dir              #改变工作目录为dir。
-S, --setuid uid          #在进入的名字空间中设置用户的uid。
-G, --setgid gid          #在进入的名字空间中设置用户的gid，同时删除全部的附加组。
--monotonic offset        #在新的时间名字空间中设置CLOCK_MONOTONIC的值，需结合-T参数。CLOCK_MONOTONIC就是开机到现在的
                            时间，但不包含休眠的时间。值的单位为秒，可正可负，意为与宿主机偏移多少秒。
--boottime offset         #在新的时间名字空间中设置CLOCK_BOOTTIME的值，需结合-T参数。CLOCK_BOOTTIME就是开机到现在的
                            时间，且包含休眠的时间。值的单位为秒，可正可负，意为与宿主机偏移多少秒。
```

图 8-1 unshare 使用语法

如果省略了“[程序 [程序的参数]]”，默认就是运行$SHELL 程序，环境变量 SHELL 保存了用户登录 shell 程序，默认就是/bin/bash。如果省略了“[参数]”，就直接在当前进程的名字空间内运行程序，可用“echo $$”查看当前进程号。如果不创建新的名字空间（即不采用“-m,-u,-i,-n,-P,-U,-C,-T,--kill-child”中的任何一个参数）运行程序，使用 unshare 来运行程序就没多大意义，这时与直接运行程序没什么区别。

语法中隔离名字空间参数都有一个可选项<file>，如果指定了一个具体存在的文件，比如--mount=/opt/nsfile，那么就创建一个持久名字空间，即使“程序”结束，名字空间依然存在，而这个文件就成为持久名字空间的挂载点，以后通过此挂载点可多次进入这个名字空间，直到卸载这个挂载点后，名字空间就被销毁。

unshare 能隔离 Linux 系统的全部名字空间，可用命令 lsns 查看系统中现存的名字空间，下面显示了一台刚装完 Rocky Linux 9 的电脑上正在使用的名字空间：

```
[root@master1 ~]# lsns
        NS TYPE   NPROCS   PID USER   COMMAND
4026531834 time      246     1 root   /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531835 cgroup    246     1 root   /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531836 pid       246     1 root   /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531837 user      246     1 root   /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531838 uts       243     1 root   /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531839 ipc       246     1 root   /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531840 net       245     1 root   /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531841 mnt       235     1 root   /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531862 mnt         1    43 root   kdevtmpfs
4026532497 mnt         1   636 root   /usr/lib/systemd/systemd-udevd
4026532498 uts         1   636 root   /usr/lib/systemd/systemd-udevd
4026532524 mnt         1   640 root   /sbin/auditd
4026532525 mnt         2   693 dbus   /usr/bin/dbus-broker-launch --scope system --audit
4026532526 mnt         1   717 chrony /usr/sbin/chronyd -F 2
4026532527 mnt         1   702 root   /usr/sbin/irqbalance --foreground
4026532528 mnt         1   703 root   /usr/lib/systemd/systemd-logind
4026532530 uts         1   717 chrony /usr/sbin/chronyd -F 2
4026532531 uts         1   703 root   /usr/lib/systemd/systemd-logind
4026532570 mnt         1   733 root   /usr/sbin/NetworkManager --no-daemon
```

https://www.

```
4026532582 net         1   768 root   /usr/lib/systemd/systemd-hostnamed
4026532633 mnt         1   768 root   /usr/lib/systemd/systemd-hostnamed
4026532730 mnt         1   929 root   /usr/sbin/rsyslogd -n
```

第一列 NS 为名字空间的唯一 id 号。

第二列 TYPE 为名字空间类型，共有九大类型，它们分别是：time（时间名字空间）、cgroup（控制组名字空间）、pid（进程名字空间）、user（用户名字空间）、uts（主机名字空间）、ipc（进程间通讯名字空间）、net（网络名字）和 mnt（挂载点名字空间）。另外还有一个文件系统名字空间，一共九类。

第三列 NPROCS 为该名字空间内正在运行的进程数量。

第四列 PID 为创建该名字空间的进程号。

lsns 命令默认列出全部名字空间，如果只想列出某种类型的名字空间，比如 mnt，可以这样“lsns -t mnt”。

对于 unshare 的参数，不过要注意的是：类似-m, --mount=<file>参数中的<file>必须在传播属性（PROPAGATION）的值为非“shared”的文件系统中，可用命令“findmnt -o+PROPAGATION”查看文件系统的属性。请看下面的例子：

```
[root@master1 ~]# touch /opt/mnt

[root@master1 ~]# unshare --mount=/opt/mnt                                  #在单独的挂载点名字空间内执行/bin/bash
unshare: mount /proc/2077/ns/mnt on /opt/mnt failed: Invalid argument

[root@master1 ~]# findmnt -o+PROPAGATION | head -n 2
TARGET      SOURCE      FSTYPE     OPTIONS                                                  PROPAGATION
/           /dev/sda1   xfs        rw,relatime,attr2,inode64,logbufs=8,logbsize=32k,noquota  shared
```

因为文件/opt/mnt 在属性 PROPAGATION 的值为 shared 的根文件系统上，所以不能作为新名字空间的挂载点。下面修改为“private”：

```
[root@master1 ~]# mkdir /opt/namespaces
[root@master1 ~]# mount --bind /opt/namespaces /opt/namespaces
[root@master1 ~]# findmnt -o+PROPAGATION | grep -e PROPAGATION -e namespaces
TARGET             SOURCE                  FSTYPE  OPTIONS                                                  PROPAGATION
└─/opt/namespaces /dev/sda1[/opt/namespaces] xfs     rw,relatime,attr2,inode64,logbufs=8,logbsize=32k,noquota  shared

[root@master1 ~]# touch /opt/namespaces/mnt
[root@master1 ~]# mount --make-private /opt/namespaces ------------------------------------------------------#修改为 private
[root@master1 ~]# findmnt -o+PROPAGATION|grep -e PROPAGATION -e namespaces
TARGET             SOURCE                  FSTYPE  OPTIONS                                                  PROPAGATION
└─/opt/namespaces /dev/sda1[/opt/namespaces] xfs     rw,relatime,attr2,inode64,logbufs=8,logbsize=32k,noquota  private

[root@master1 ~]# PS1= "abc \W# " unshare --mount=/opt/namespaces/mnt          #在独立的挂载点名字空间内执行/bin/bash
abc ~# cd /
abc /# ls
afs  bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
abc /# exit
logout

[root@master1 ~]# PS1= "abc \W# " nsenter --mount=/opt/namespaces/mnt                                #再次进入名字空间
abc ~# exit
logout

[root@master1 ~]# umount /opt/namespaces/mnt                                                  #卸载挂载点，名字空间被销毁
[root@master1 ~]# nsenter --mount=/opt/namespaces/mnt
nsenter: reassociate to namespace 'ns/mnt' failed: Invalid argument

[root@master1 ~]# umount /opt/namespaces
[root@master1 ~]#
```

unshare 命令的其他名字空间参数的 file 允许在任意文件系统中，比如指定参数--uts=/opt/uts123，具体参考§8.6 小节。

§8.1.2 nsenter 命令用法

nsenter 在不同的名字空间中执行程序，其详细的使用语法如图 8-2 所示。

```
nsenter        [参数]          [程序       [程序的参数] ]

-a, --all                  #进入目标进程的名字空间。用特别指定的名字空间（比如额外使用参数--mount=file）覆盖目标进程相应的名字空间。
 -t, --target PID          #指定目标进程。进程的全部名字空间在/proc/PID/ns目录中，如/proc/PID/ns/net为网络名字空间。
-m, --mount[=<file>]       #隔离挂载点名字空间。如果指明了file，则通过绑定一个挂载点创建持久名字空间，当umount file后销毁名字空间。
-u, --uts[=<file>]         #隔离主机名名字空间。
-i, --ipc[=<file>]         #隔离进程间通信名字空间。
-n, --net[=<file>]         #隔离网络名字空间。
-p, --pid[=<file>]         #隔离进程名字空间。如果指定file，那么也必须指定--fork参数。
-U, --user[=<file>]        #隔离用户和组名字空间。
-C, --cgroup[=<file>]      #隔离cgroup名字空间。
--T, --time[=<file>]       #隔离时间名字空间。一般再指定--monotonic或--boottime定义相应的时钟偏移。
-G, --setgid gid           #在进入的名字空间中设置用户的gid，同时删除全部的附加组。
-S, --setuid uid           #在进入的名字空间中设置用户的uid。
--preserve-credentials     #进入用户名字空间后不修改用户的UID和GID。省略本参数，用户的UID和GID会改为0。
-r, --root[=directory]     #进入名字空间后的根目录。如果省略directory，就把“程序”所在的目录切换成根目录。
-w, --wd[=directory]       #设置工作目录。 如果省略directory，则将工作目录设置为“程序”的工作目录。 如果指定了directory，则将工作目录设为它
-F, --no-fork              #直接在当前进程中执行“程序”，如果省略本参数，则在子进程中执行“程序”。
-Z, --follow-context       #采用与目标进程相同的SELinux上下文。须结合-r参数。
```

图 8-2 nsenter 命令使用语法

不同于 unshare 命令，nsenter 命令不会创建新的名字空间，它只是借用现有进程（由“-t, --target PID”参数指定）的名字空间，或者由其他参数（如“-m, --mount[=file]”等）指定的现存名字空间。因此由 nsenter 执行的“程序”的全部名字空间可能借自不同的进程，比如命令“nsenter --uts=/opt/ns123 -t 928 -a /usr/sbin/sshd”执行的程序/usr/sbin/sshd，其名字空间除了主机名字空间外都借自 928 号进程（参数-t 928 -a），而主机名字空间借自/opt/ns123，它又是另一个进程的持久主机名字空间（参考 unshare 命令的语法）。另外 nsenter 允许借用其他进程的部分名字空间，比如“-t 928 -u -n”参数只借用 928 进程的主机名字空间和网络名字空间。

如果省略“[程序 [程序的参数]]”，就是执行程序$SHELL，环境变量 SHELL 保存了用户登录 shell 程序，默认就是/bin/bash。如果“[参数]”也省略，即直接输入 nsenter 并回车，则等价于命令“nsenter -t $$ -a $SHELL”，特殊变量$保存了当前 shell 程序的进程号，变量前加$表示引用一个变量的值。下面的命令显示$和 SHELL 变量的值：

```
[root@master1 ~]# echo $$ $SHELL
1287 /bin/bash

[root@master1 ~]# ps
    PID TTY          TIME CMD
   1287 pts/0    00:00:00 bash
   1697 pts/0    00:00:00 ps
```

一个进程使用的名字空间可在/proc/<PID>/ns 目录中查到，比如查看 1287 号进程的名字空间：

```
[root@master1 ~]# ls /proc/1287/ns
cgroup  ipc  mnt  net  pid  pid_for_children  time  time_for_children  user  uts
```

再看下面的例子。

首先创建若干新的名字空间并在这些新的名字空间中运行程序 sleep 1h&：

```
[root@master1 ~]# unshare -m -u -i -n -U -C -f -r /bin/sleep 1h&
[1] 1980
```

创建的新的名字空间由参数-m -u -i -n -U -C 指定（具体是哪些名字空间，请参考图 8-1 所示的 unshare 的语法自行判断），sleep 1h 表示睡眠 1 小时，程序后面增加符号“&”表示在后台运行。

然后可再次进入 sleep 进程的名字空间，比如下面的例子就是进入名字空间后修改主机名：

```
[root@master1 ~]# ps --forest
    PID TTY          TIME CMD
   1833 pts/0    00:00:00 bash
   1890 pts/0    00:00:00  \_ unshare
   1891 pts/0    00:00:00  |   \_ sleep
   1894 pts/0    00:00:00  \_ ps
```

```
[root@master1 ~]# nsenter -t 1891 -a hostname foolishme                #修改 1891 号进程所在的名字空间内的主机名为 foolishme
[root@master1 ~]# nsenter -t 1891 -u hostname
foolishme
[root@master1 ~]# nsenter -t 1891 -a------------------------------------#进入 1891 号进程所在的名字空间，然后执行程序/bin/bash
[root@foolishme /]# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
[root@foolishme /]# exit
logout
[root@master1 ~]#
```

拓展学习：

1. mount 命令的用法。

§8.2 网络名字空间

网络名字空间由 Net Namespace 翻译而来，用来隔离网络资源（网卡名、路由表、ARP 表、ip 地址表、iptables 表、ebtable 表、dns 等）。网络名字空间可以让各个进程拥有自己独立的 IP 地址、端口和网卡信息等，一个网络名字空间相当于一台独立的电脑，但是共享文件系统，示意图请参考图 8-3。

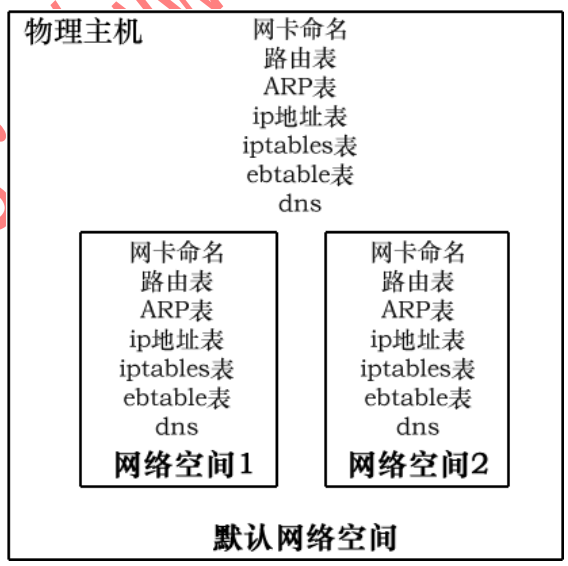


图 8-3 Linux 主机中的网络名字空间

例如宿主机的 IP 地址为 10.10.10.11，但在另一个网络名字空间中可以设置独立的 IP 地址为 192.168.1.1。验证：

```
[root@master1 ~]# PS1="abc# " unshare -n --fork /bin/bash
abc# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

https://www.veryopen.org

```
abc# exit
```

在 Linux 系统中，主要采用“ip netns ...”命令来管理网络名字空间，包括创建、删除、修改、查询、在网络名字空间中执行命令等操作，可执行命令 man ip-netns 获取详细的帮助。请看下面的实战：

```
[root@master1 ~]# ip netns list                                    #查看存在的网络名字空间（也可用命令 lsns -t net）
[root@master1 ~]# ip netns add ns1                                 #新建网络名字空间 ns1
[root@master1 ~]# ip netns add netns2
[root@master1 ~]# ip netns list
ns1
netns2
[root@master1 ~]# ls -l /var/run/netns                             #创建的网络名字空间对应/var/run/netns 目录中的一个文件
total 0
-r--r--r-- 1 root root 0 Aug  1 16:45 netns2
-r--r--r-- 1 root root 0 Aug  1 16:03 ns1

[root@master1 ~]# ip netns exec ns1 ip a                           #在 ns1 中执行命令“ip a”
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

[root@master1 ~]# ip netns exec ns1 ip link set lo up              #启用 ns1 中的 lo 设备
[root@master1 ~]# ip netns exec ns1 /bin/bash                      #进入 ns1
[root@master1 ~]# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
[root@master1 ~]# ip r
[root@master1 ~]# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
[root@master1 ~]# ps
    PID TTY          TIME CMD
    951 pts/0    00:00:00 bash
   1205 pts/0    00:00:00 bash
   1228 pts/0    00:00:00 ps
[root@master1 ~]# exit                                             #退出 ns1
exit

[root@master1 ~]# ip -all netns exec ip a                          #在全部名字空间中执行命令“ip a”

netns: netns2
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

netns: ns1
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    ......

[root@master1 ~]# ip netns delete ns1                              #删除 ns1 网络名字空间
[root@master1 ~]# ip -all netns delete                             #删除全部网络名字空间
[root@master1 ~]# ip netns list
```

Linux 下存在各种网元，比如 tap、tun、veth、bridge，都是用软件模拟出来的网络设备，网络名字空间不能单独存在，需要结合各种网元才有意义。下面各小节分别介绍这些网元。

§8.2.1 tap/tun

tap 等同于一个以太网设备（如以太网卡），它工作在数据链路层。tun 模拟了网络层设备，工作在第三层即网络层。

任何一个网络设备都有两端，比如一个物理以太网卡，一端连接内核 tcp/ip 协议栈，另一端是通过网线连接外界网络设备（如交换机）。tap/tun 这种虚拟网络设备也是如此，一端连接内核 tcp/ip 协议栈，另一端通过字符设备文件连接用户态的用户程序。图 8-4 描述了物理网卡、tap 和 tun 的工作原理。

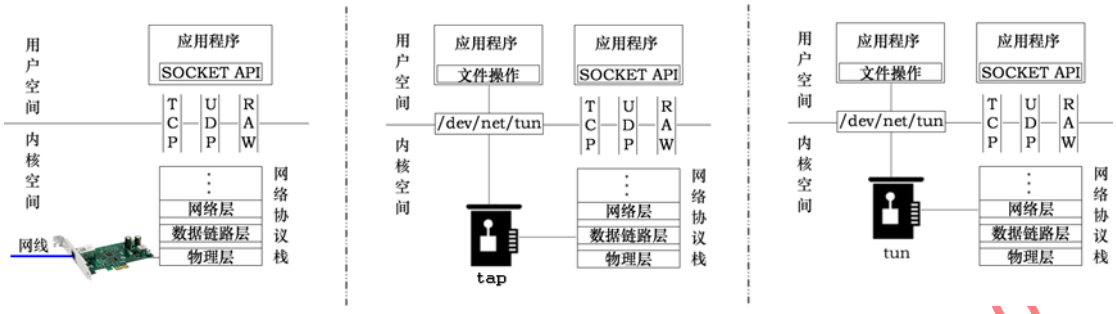


图 8-4 “网卡”的工作原理

由于 tap 工作在二层，能附加 MAC 地址和 IP 地址，所以能直接与物理网卡连接，因此 tap 设备常用来实现网桥和虚拟机交换机等。tun 工作在三层，只能附加 IP 地址而不能附加 MAC 地址，所以它不能与物理以太网卡连接，tun 设备常用于实现 vpn、tunnel、ipsec 之类的应用。

虚拟机一般采用 tap 设备来组网，图 8-5 所示就是典型的虚拟机网络通路。

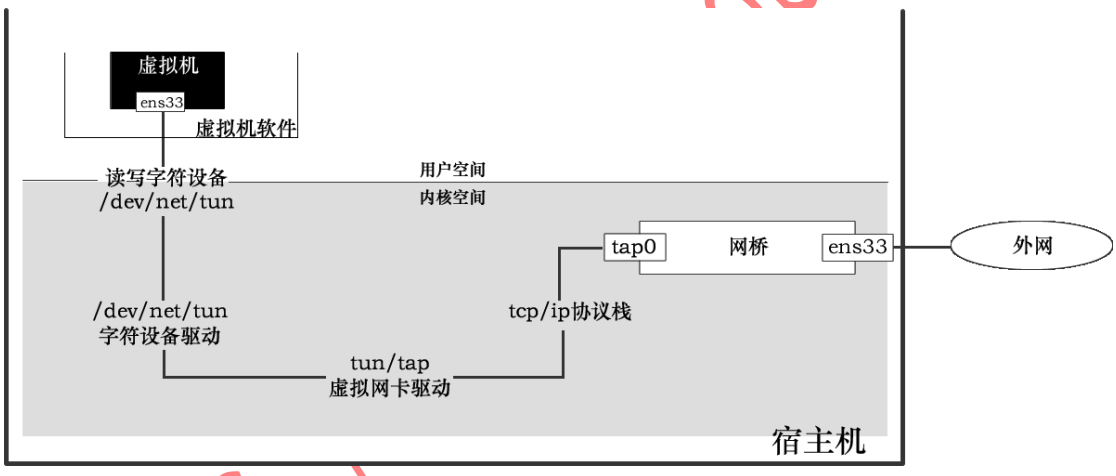


图 8-5 虚拟机的网络通路

虚拟机中的网卡 eth0 发送的包数等于 tap0 收到的包数，eth0 收到的包数等于 tap0 发送的包数。在宿主机上运行命令“ps -ef|grep qemu”，会发现类似“-netdev tap,fd=26”的信息，表明虚拟机软件通过 26 号文件句柄读取 tap 虚拟网卡的数据。

比如 vpn 的示意图请参考 8-6 所示。

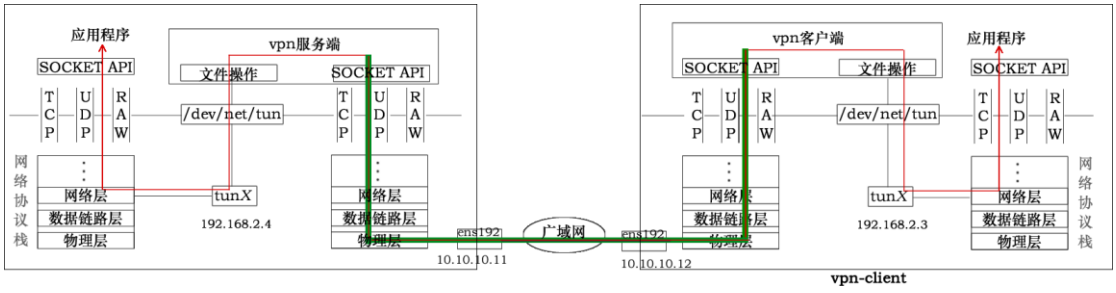


图 8-6 vpn 实现原理

一方面 vpn 客户端与 vpn 服务器建立连接从而构建隧道，另一方面双方以读写方式打开/dev/net/tun 设备文件。当本地的应用程序向对方发送数据（不妨设为 X）时（比如 vpn 服务端的应用程序向 192.168.2.3 发数据），三层数据包（不妨设为 Y）首先到达本地的 tun 设备，然后被 vpn 程序读取到，再然后把 Y 作为数据发送到对方的公网 ip（比如 202.113.198.3），对方 vpn 程序拿到 Y 后写到/dev/net/tun，最后对方的应用程序就获得了 X 了。至于如何编程实现 vpn，参考后面的实战。

实战 1：

在默认的网络名字空间中创建 tap/tun 设备，并设置 ip，参考图 8-7 所示。

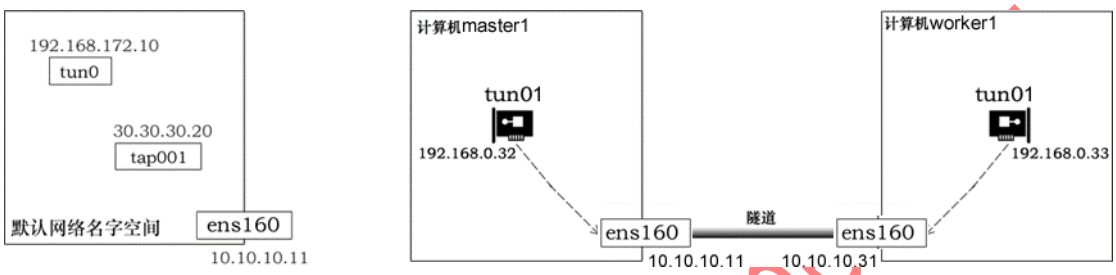


图 8-7 tap/tun 实战

具体操作如下。

实战 1.1：实现图 8-7 左侧部分。

(1). tun0 设备：

```
[root@master1 ~]# ip tuntap add mode tun name tun0                                  #创建 tun0 设备
[root@master1 ~]# ip address add 192.168.172.10/24 dev tun0                         #给 tun0 设置 ip
[root@master1 ~]# ip link set tun0 up                                               #启用 tun0 设备
[root@master1 ~]# ping 192.168.172.10
PING 192.168.172.10 (192.168.172.10) 56(84) bytes of data.
64 bytes from 192.168.172.10: icmp_seq=1 ttl=64 time=0.042 ms
64 bytes from 192.168.172.10: icmp_seq=2 ttl=64 time=0.126 ms
64 bytes from 192.168.172.10: icmp_seq=3 ttl=64 time=0.046 ms
^C
--- 192.168.172.10 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2079ms
rtt min/avg/max/mdev = 0.042/0.071/0.126/0.038 ms
[root@master1 ~]# ip addr show dev tun0
4: tun0: <NO-CARRIER,POINTOPOINT,MULTICAST,NOARP,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 500
    link/none
    inet 192.168.172.10/24 scope global tun0
       valid_lft forever preferred_lft forever
```

(2). tap001 设备：

```
[root@master1 ~]# ip tuntap add mode tap name tap001                                #创建 tap001 设备
[root@master1 ~]# ip address add 30.30.30.20/24 dev tap001                          #给 tap001 设置 ip
[root@master1 ~]# ip link set tap001 up                                             #启用 tap001 设备
[root@master1 ~]# ping 30.30.30.20
PING 30.30.30.20 (30.30.30.20) 56(84) bytes of data.
64 bytes from 30.30.30.20: icmp_seq=1 ttl=64 time=0.034 ms
64 bytes from 30.30.30.20: icmp_seq=2 ttl=64 time=0.124 ms
64 bytes from 30.30.30.20: icmp_seq=3 ttl=64 time=0.033 ms
^C
--- 30.30.30.20 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2063ms
rtt min/avg/max/mdev = 0.033/0.063/0.124/0.042 ms
[root@master1 ~]# ip addr
……
4: tun0: <NO-CARRIER,POINTOPOINT,MULTICAST,NOARP,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 500
    link/none
    inet 192.168.172.10/24 scope global tun0
       valid_lft forever preferred_lft forever
5: tap001: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 1000
    link/ether 62:c3:94:a0:6c:8d brd ff:ff:ff:ff:ff:ff
    inet 30.30.30.20/24 scope global tap001
       valid_lft forever preferred_lft forever
```

tun0 设备没有 MAC 地址，而 tap001 有。还可以修改 tap001 的 mac 地址：

```
[root@master1 ~]# ip link set dev tap001 address 62:88:77:66:55:44
[root@master1 ~]# ip link show dev tap001
5: tap001: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN mode DEFAULT group default qlen 1000
    link/ether 62:88:77:66:55:44 brd ff:ff:ff:ff:ff:ff
```

实战 1.2：实现图 8-7 右侧部分。

在两台电脑之间建立 gre 隧道，从而使得两块 tun01 通过隧道连通。

1. 在 master1 上操作：

```
[root@master1 ~]# modprobe ip_gre                                    #加载内核模块
[root@master1 ~]# lsmod | grep gre
ip_gre                 32768  0
ip_tunnel              36864  1 ip_gre
gre                    16384  1 ip_gre
```

加载了 ip_gre 的 Linux 内核模块后，新建了三个虚拟网络设备：gre0@NONE、gretap0@NONE 和 erspan0@NONE。两台电脑之间通过 ens160 网卡建立 gre 隧道（隧道网络数据包中的源地址 10.10.10.11，目的 ip 为 10.10.10.31）：

```
[root@master1 ~]# ip tunnel add tun01 mode gre remote 10.10.10.31 local 10.10.10.11 ttl 64 dev ens160
[root@master1 ~]# ip a
……
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:30:3b:e7 brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.11/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe30:3be7/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: gre0@NONE: <NOARP> mtu 1476 qdisc noop state DOWN group default qlen 1000
    link/gre 0.0.0.0 brd 0.0.0.0
4: gretap0@NONE: <BROADCAST,MULTICAST> mtu 1462 qdisc noop state DOWN group default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
5: erspan0@NONE: <BROADCAST,MULTICAST> mtu 1450 qdisc noop state DOWN group default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
6: tun01@ens160: <POINTOPOINT,NOARP> mtu 1476 qdisc noop state DOWN group default qlen 1000
    link/gre 10.10.10.11 peer 10.10.10.31

[root@master1 ~]# ip link set tun01 up
```

隧道内网络数据包中的源 ip（192.168.0.32）和目的 ip （192.168.0.33）：

```
[root@master1 ~]# ip address add 192.168.0.32/24 peer 192.168.0.33/24 dev tun01
[root@master1 ~]# ip address show dev tun01
7: tun0@ens192: <POINTOPOINT,NOARP,UP,LOWER_UP> mtu 1476 qdisc noqueue state UNKNOWN group default qlen 1000
    link/gre 10.10.10.11 peer 10.10.10.31
    inet 192.168.0.32 peer 192.168.0.33/24 scope global tun0
       valid_lft forever preferred_lft forever
    inet6 fe80::200:5efe:1414:1420/64 scope link
       valid_lft forever preferred_lft forever

[root@master1 ~]# sysctl -w net.ipv4.ip_forward=1                    #开启内核的转发功能（实现路由）
net.ipv4.ip_forward = 1
```

2. 在 worker1 上操作：

```
[root@worker1 ~]# modprobe ip_gre
[root@worker1 ~]# ip tunnel add tun01 mode gre remote 10.10.10.11 local 10.10.10.31 ttl 64 dev ens160
[root@worker1 ~]# ip link set tun01 up
[root@worker1 ~]# ip address add 192.168.0.33/24 peer 192.168.0.32/24 dev tun01
[root@worker1 ~]# sysctl -w net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

Linux 支持五种隧道类型，它们分别是 gre、ipip、isatap、sit 和 vti，gre 用得比较多。管理隧道的命令“ip tunnel ...”，执行命令“man ip-tunnel”可获得详细的帮助。

3. 验证：

```
在 master1 上经 tun01 设备 ping 对方：
[root@master1 ~]# ping -I tun01 192.168.0.33
PING 192.168.0.33 (192.168.0.33) from 192.168.0.32 tun01: 56(84) bytes of data.
64 bytes from 192.168.0.33: icmp_seq=1 ttl=64 time=2.11 ms
64 bytes from 192.168.0.33: icmp_seq=2 ttl=64 time=0.778 ms
64 bytes from 192.168.0.33: icmp_seq=3 ttl=64 time=1.01 ms
64 bytes from 192.168.0.33: icmp_seq=4 ttl=64 time=0.912 ms
……
```

如果 ping 不通，不妨在 worker1 上 ping 192.168.0.32 试试，然后再从 master1 上 ping 192.168.0.33。

在 worker1 抓取 ens160 网卡的网络数据包：

```
[root@worker1 ~]# tcpdump -i ens160 proto gre
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on ens160, link-type EN10MB (Ethernet), snapshot length 262144 bytes
17:18:06.252952 IP 20.20.20.32 > worker1: GREv0, length 88: IP 192.168.0.32 > worker1: ICMP echo request, id 5, seq 1, length 64
17:18:06.252986 IP worker1 > 20.20.20.32: GREv0, length 88: IP worker1 > 192.168.0.32: ICMP echo reply, id 5, seq 1, length 64
17:18:07.260178 IP 20.20.20.32 > worker1: GREv0, length 88: IP 192.168.0.32 > worker1: ICMP echo request, id 5, seq 2, length 64
17:18:07.260331 IP worker1 > 20.20.20.32: GREv0, length 88: IP worker1 > 192.168.0.32: ICMP echo reply, id 5, seq 2, length 64
17:18:08.283259 IP 20.20.20.32 > worker1: GREv0, length 88: IP 192.168.0.32 > worker1: ICMP echo request, id 5, seq 3, length 64
17:18:08.283302 IP worker1 > 20.20.20.32: GREv0, length 88: IP worker1 > 192.168.0.32: ICMP echo reply, id 5, seq 3, length 64
17:18:09.307245 IP 20.20.20.32 > worker1: GREv0, length 88: IP 192.168.0.32 > worker1: ICMP echo request, id 5, seq 4, length 64
……
```

可以发现 88 个字节大小的 GREv0 隧道报文，里面包含了 64 个字节大小的 ICMP 报文。

§ 8.2.2 veth

Veth（Virtual Ethernet devices）是 Linux 中一种用软件虚拟出来的模拟硬件网卡的设备，Veth 总是成对出现，所以一般也叫 Veth-Pair，用于点对点的虚拟连接。其作用非常简单：如果 veth0 和 veth1 是一对 veth 设备，veth0 发送的数据会由 veth1 收到，反之亦然，其实说白了，Veth 就是一根用“网线”直连的两块网卡，你从一头发数据，自然可以从另一头收到数据。示意图请参考图 8-8。

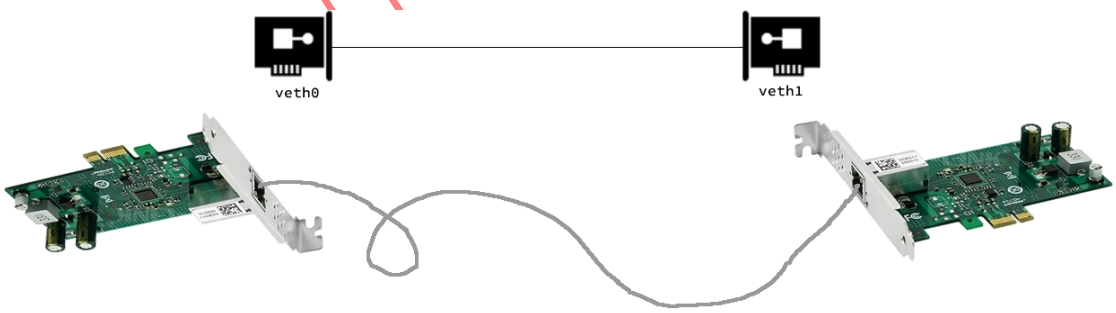


图 8-8 一对 veth 设备

veth 设备工作在二层，因此可以添加 MAC 和 IP。现在的容器一般都是采用 veth 设备来组网。

实战 2：

实现图 8-9 所示的效果。

www.veryopen.

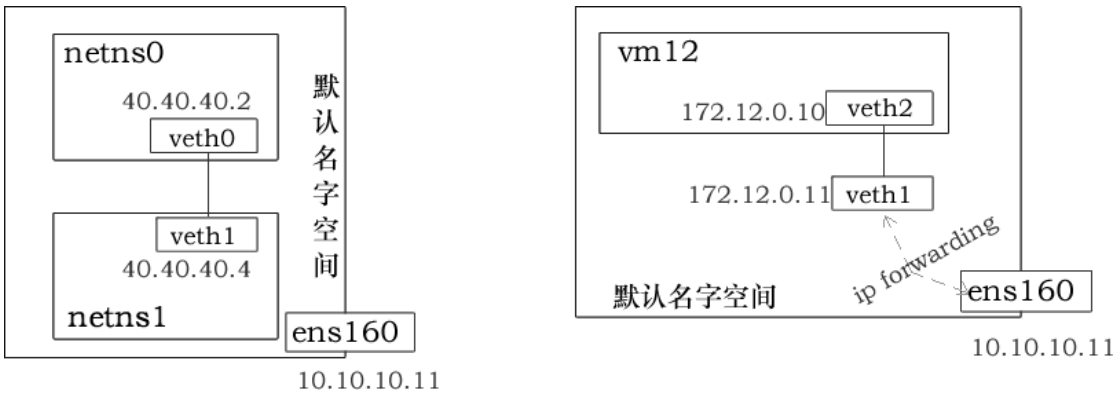


图 8-9 veth 实战

实战 2.1：参考图 8-9 左侧部分。

要创建两个网络名字空间 netns0 和 netns1，创建一对 veth 设备，每个 veth 设备放在在不同的网络名字空间内容，从而使得 netns0 和 netns1 连通。具体操作如下：

```
[root@master1 ~]# ip netns add netns0                                    #创建 netns0 网络名字空间
[root@master1 ~]# ip netns add netns1
[root@master1 ~]# ip link add type veth ---------- #创建一对 veth 设备，名字为 veth0 和 veth1
[root@master1 ~]# ip link set veth0 netns netns0                         #把 veth0 插入 netns0 中
[root@master1 ~]# ip link set veth1 netns netns1----------#把 veth1 插入 netns1 中
[root@master1 ~]# ip netns exec netns0 ip address add 40.40.40.2/24 dev veth0    #在 netns0 中配置 veth0 设备的 ip
[root@master1 ~]# ip netns exec netns1 ip address add 40.40.40.4/24 dev veth1
[root@master1 ~]# ip netns exec netns0 ip link set veth0 up----------#在 netns0 中启用 veth0 设备
[root@master1 ~]# ip netns exec netns1 ip link set veth1 up
[root@master1 ~]# ip netns exec netns0 ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
6: veth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether de:d7:60:e7:59:d9 brd ff:ff:ff:ff:ff:ff link-netns netns1
    inet 40.40.40.2/24 scope global veth0
       valid_lft forever preferred_lft forever
    inet6 fe80::dcd7:60ff:fee7:59d9/64 scope link
       valid_lft forever preferred_lft forever
```

可见 6 号网络接口为 veth0@if7，表示网络接口名为 veth0，对端是 7 号网络接口，可在 netns1 空间内用命令“ip link | grep -A1 ^7”查看 7 号网络接口，肯定是 veth1。

```
[root@master1 ~]# ip netns exec netns0 ping 40.40.40.4                   #在 netns0 中 ping 通 netns1
PING 40.40.40.4 (40.40.40.4) 56(84) bytes of data.
64 bytes from 40.40.40.4: icmp_seq=1 ttl=64 time=0.033 ms
64 bytes from 40.40.40.4: icmp_seq=2 ttl=64 time=0.044 ms
^C
--- 40.40.40.4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.033/0.038/0.044/0.005 ms
[root@master1 ~]# ip netns exec netns1 ping 40.40.40.2
PING 40.40.40.2 (40.40.40.2) 56(84) bytes of data.
64 bytes from 40.40.40.2: icmp_seq=1 ttl=64 time=0.040 ms
64 bytes from 40.40.40.2: icmp_seq=2 ttl=64 time=0.102 ms
^C
--- 40.40.40.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1034ms
rtt min/avg/max/mdev = 0.040/0.071/0.102/0.031 ms
[root@master1 ~]# ip netns exec netns1 ping 9.9.9.9                      #ping 不通外网
ping: connect: Network is unreachable
```

实战 2.2：参考图 8-9 右侧部分。

能从 vm12 网络空间 ping 通外网，具体操作如下：

```
[root@master1 ~]# ip -all netns delete                                   #删除全部网络名字空间
```

```
[root@master1 ~]# ip netns add vm12
[root@master1 ~]# ip link add veth1 type veth peer name veth2                    #添加一堆 veth 设备，取名 veth1/veth2
[root@master1 ~]# ip link
……
8: veth2@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether d6:ee:b4:ec:b0:1d brd ff:ff:ff:ff:ff:ff
9: veth1@veth2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 1e:e2:b9:cc:02:38 brd ff:ff:ff:ff:ff:ff

[root@master1 ~]# ip link set veth2 netns vm12                                   #把 veth2 插入 vm12 网络名字空间
[root@master1 ~]# ip address add 172.12.0.11/24 dev veth1
[root@master1 ~]# ip link set veth1 up                                           #激活默认网络名字空间内的 veth1
[root@master1 ~]# ip netns exec vm12 ip address add 172.12.0.10/24 dev veth2
[root@master1 ~]# ip netns exec vm12 ip link set veth2 up
[root@master1 ~]# ip netns exec vm12 ip route add default via 172.12.0.11       #在 vm12 中设备默认路由
[root@master1 ~]# sysctl -w net.ipv4.ip_forward=1                                #在默认网络空间内允许转发
```

做 SNAT 转换，只要把源 ip 为 172.12.0.0/24 的网络包改为网卡 ens160 的 ip：

```
[root@master1 ~]# firewall-cmd --direct --passthrough ipv4 -t nat -A POSTROUTING -s 172.12.0.0/24 -o ens160 -j MASQUERADE
success
[root@master1 ~]# ip netns exec vm12 ping 9.9.9.9                                #在 vm12 中 ping 外网
PING 9.9.9.9 (9.9.9.9) 56(84) bytes of data.
64 bytes from 9.9.9.9: icmp_seq=1 ttl=127 time=265 ms
64 bytes from 9.9.9.9: icmp_seq=2 ttl=127 time=271 ms
^C

[root@master1 ~]# PS1="vm12# " ip netns exec vm12 /bin/bash                      #进入 vm12 网络名字空间
vm12# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: veth2@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 9e:73:7d:88:a6:87 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.12.0.10/24 scope global veth2
       valid_lft forever preferred_lft forever
    inet6 fe80::9c73:7dff:fe88:a687/64 scope link
       valid_lft forever preferred_lft forever

vm12# ping www.baidu.com
PING www.wshifen.com (183.232.231.172) 56(84) bytes of data.
64 bytes from 183.232.231.172 (183.232.231.172): icmp_seq=1 ttl=127 time=24.8 ms
64 bytes from 183.232.231.172 (183.232.231.172): icmp_seq=2 ttl=127 time=19.5 ms
^C
vm12# exit
exit
[root@master1 ~]#
```

因为 vm12 与宿主共享文件系统，所以在 vm12 内也是能读取/etc/resolv.conf 文件的，里面包含了 DNS。

§8.2.3 bridge

在物理网络中，如果需要连接多个主机，我们会使用网桥，或者又称为集线器或交换机。Linux 也提供了网桥的虚拟实现——Linux Bridge。Linux 网桥有多个端口，数据可以从任何端口进入，从哪个出口出去取决于目的地 Mac 地址，原理与硬件交换机一样。

Linux 网桥工作在二层，网桥自身可以配置 ip 地址和 mac 地址，所以只有 tap、veth 和物理网卡允许插到网桥上，被插到网桥上的设备成为端口，所以不需要 ip 地址，且工作于混杂模式。

实战 3：

参考图 8-10，左侧部分实现能从外面登录到网络名字空间中，右侧部分实现两个网络名字空间通过第三个网络名字空间互联在一起。

https://www.

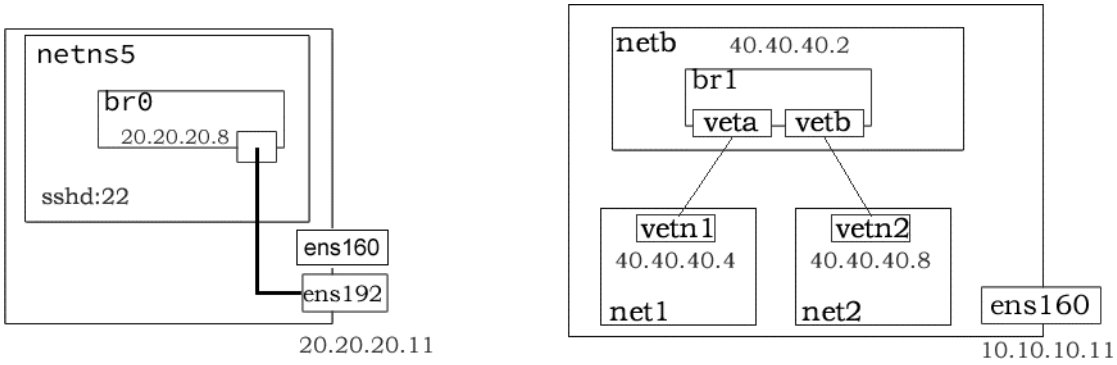


图 8-10 实战网桥

实战 3.1：实现图 8-10 左侧部分。

具体操作：

1. 首先给虚拟机增加一块主机模式的网卡，并设置静态 Ip 20.20.20.11：

```
[root@master1 ~]# ip a                                                    #查看第二块网卡的名字
……
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:39:27:ef brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.11/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe39:27ef/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:0c:29:8c:f9:bc brd ff:ff:ff:ff:ff:ff
    altname enp11s0
    inet 20.20.20.11/24 brd 20.20.20.255 scope global noprefixroute ens192
       valid_lft forever preferred_lft forever
    inet6 fe80::8cd9:d537:8266:35f9/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

我这里第二块网卡的名字为 ens192，有的名字为 ens224，要根据实际情况替换下面命令中的 ens192。

```
[root@master1 ~]# nmcli c add type ethernet ifname ens192 con-name ens192
Connection 'ens192' (786af0e3-db29-424f-a5c4-30da16cc3628) successfully added.

[root@master1 ~]# nmcli c modify ens192 ipv4.addresses 20.20.20.11/24
[root@master1 ~]# nmcli c modify ens192 ipv4.method manual
[root@master1 ~]# nmcli c up ens192
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnection/5)

[root@master1 ~]# ping 20.20.20.11                                        #确保能 ping 通
```

如果配置错了，请使用命令 nmcli c del ens192 或者 nmcli c del ens224 删除。然后再重复上面的命令配置网卡。

2. 然后再操作：

```
[root@master1 ~]# ip -all netns delete                                    #删除之前的全部网络名字空间
[root@master1 ~]# ip netns add netns5 ------------------------------------------------------------ #新建网络名字空间 netns5
[root@master1 ~]# ip netns exec netns5 ip link add br0 type bridge        #在 netns5 中增加网桥 br0
[root@master1 ~]# ip netns exec netns5 ip link set br0 up                 #激活网桥
[root@master1 ~]# ip link set dev ens192 netns netns5 -------------------------------------------#把物理网卡 ens192 加入 netns5
[root@master1 ~]# ip netns exec netns5 ip link set ens192 up
[root@master1 ~]# ip netns exec netns5 ip link set dev ens192 master br0---------#把 ens192 插入网桥并把它作为网桥的上行口
[root@master1 ~]# ip netns exec netns5 ip address add 20.20.20.8/24 dev br0-----------------------------------#给网桥配置 ip
[root@master1 ~]# ip netns exec netns5 /usr/sbin/sshd -o PidFile=/var/run/sshd-netns.pid      #在 netns5 中启动登录服务
[root@master1 ~]# ip netns exec netns5 ss -tnlp                           #查看 netns5 中被监听的端口（22 就是远程登录的端口号）
State    Recv-Q   Send-Q     Local Address:Port      Peer Address:Port   Process
LISTEN   0        128              0.0.0.0:22             0.0.0.0:*       users:(("sshd",pid=1529,fd=4))
LISTEN   0        128                 [::]:22                [::]:*       users:(("sshd",pid=1529,fd=3))
```

下面尝试远程登陆到 netns5 名字空间中：

```
[root@master1 ~]# ssh root@20.20.20.8
```

```
The authenticity of host '20.20.20.8 (20.20.20.8)' can't be established.
ED25519 key fingerprint is SHA256:i6+i+wS2exXmKsospCS9naY7pFCATpHMyk5AtKoHr10.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: worker1
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '20.20.20.8' (ED25519) to the list of known hosts.
root@20.20.20.8's password:
Last login: Wed Oct 16 15:09:40 2024 from 10.7.57.10
[root@master1 ~]# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 00:0c:29:3e:24:25 brd ff:ff:ff:ff:ff:ff
    inet 20.20.20.8/24 scope global br0
       valid_lft forever preferred_lft forever
    inet6 fe80::548c:61ff:fe92:2a1b/64 scope link
       valid_lft forever preferred_lft forever
3: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master br0 state UP group default qlen 1000
    link/ether 00:0c:29:3e:24:25 brd ff:ff:ff:ff:ff:ff
    altname enp11s0
    inet6 fe80::20c:29ff:fe3e:2425/64 scope link
       valid_lft forever preferred_lft forever
[root@master1 ~]# exit
logout
Connection to 20.20.20.8 closed.
[root@master1 ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:30:3b:e7 brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.11/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe30:3be7/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
[root@master1 ~]#
```

如果没法登陆，请检查 Vmware 虚拟化软件的主机模式网段有没有改成 20.20.20.0/24（具体参考图 1-2）。

由最后一条命令的输出可发现宿主机上没有了 ens192 网卡，因为它被加到 netns5 网络名字空间中去了。如果要把网卡从网桥上拔下可用命令“ip link set dev ens192 nomaster”。如果要删除网桥可用命令“ip link delete br0 type bridge”。

请重启虚拟机后再做下面的实验。

实战 3.2：实现图 8-10 右侧部分。

具体操作：

```
[root@master1 ~]# ip netns add net1                                                    #新建三个网络名字空间
[root@master1 ~]# ip netns add net2
[root@master1 ~]# ip netns add netb

[root@master1 ~]# ip link add type veth                                                #新建 veth0/veth1 对
[root@master1 ~]# ip link add type veth                                                #新建 veth2/veth3 对

[root@master1 ~]# ip link set dev veth0 name veta netns netb                #veth0 加入 netb 网络名字空间并改名为 veta
[root@master1 ~]# ip link set dev veth1 name vetn1 netns net1
[root@master1 ~]# ip link set dev veth2 name vetb netns netb
[root@master1 ~]# ip link set dev veth3 name vetn2 netns net2

[root@master1 ~]# PS1="netb# " ip netns exec netb /bin/bash                            #进入 netb
netb# ip link add br1 type bridge                                                      #新建 br1 网桥
netb# ip link set br1 up
netb# ip link set veta up
netb# ip link set vetb up
netb# ip link set dev veta master br1                                                  #把 veta 插入网桥
netb# ip link set dev vetb master br1
netb# bridge link                                                                      #显示桥 br1 上的端口
10: veta@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br1 state forwarding priority 32 cost 2
12: vetb@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br1 state forwarding priority 32 cost 2
netb# exit                                                                             #退出 netb
exit
[root@master1 ~]# ip netns exec net1 ip address add 40.40.40.4/24 dev vetn1
[root@master1 ~]# ip netns exec net2 ip address add 40.40.40.8/24 dev vetn2
[root@master1 ~]# ip netns exec net1 ip link set vetn1 up
```

```
[root@master1 ~]# ip netns exec net2 ip link set vetn2 up
[root@master1 ~]# ip netns exec net1 ping 40.40.40.8
PING 40.40.40.8 (40.40.40.8) 56(84) bytes of data.
64 bytes from 40.40.40.8: icmp_seq=1 ttl=64 time=0.044 ms
64 bytes from 40.40.40.8: icmp_seq=2 ttl=64 time=0.086 ms
^C
--- 40.40.40.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1068ms
rtt min/avg/max/mdev = 0.044/0.065/0.086/0.021 ms
[root@master1 ~]# ip netns exec net2 ping 40.40.40.4
PING 40.40.40.4 (40.40.40.4) 56(84) bytes of data.
64 bytes from 40.40.40.4: icmp_seq=1 ttl=64 time=0.046 ms
64 bytes from 40.40.40.4: icmp_seq=2 ttl=64 time=0.070 ms
^C
--- 40.40.40.4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1027ms
rtt min/avg/max/mdev = 0.046/0.058/0.070/0.012 ms
```

参考：深入理解 Linux 上软件实现的“交换机” - Bridge!（https://zhuanlan.zhihu.com/p/421276975）

实验：命令 bridge 的用法。

www.veryopen.org

§8.2.4 一个简单的 VPN

好了，在掌握了 Linux 各种虚拟网元之后，我们自己手工实现一个简易的 VPN，参考图 8-11 所示。

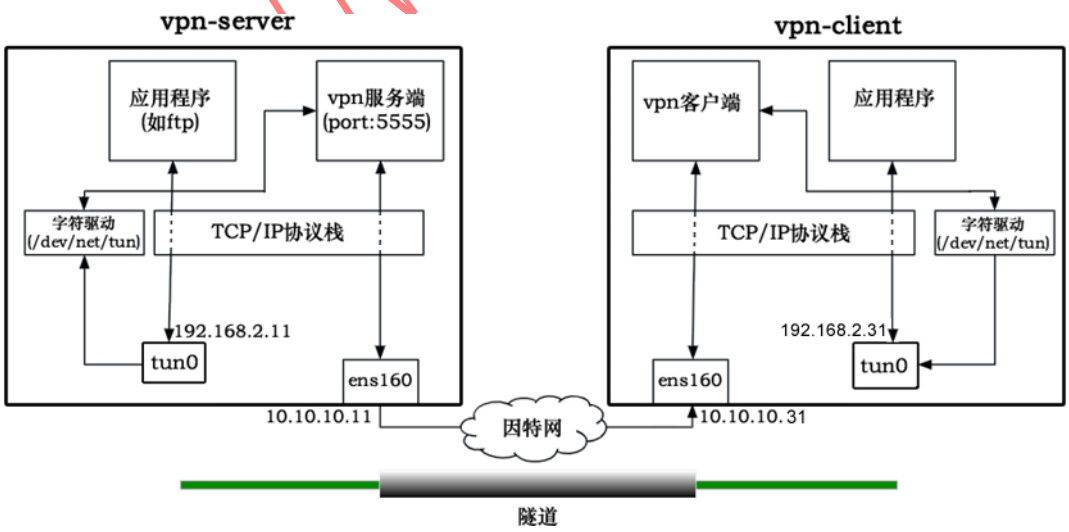


图 8-11 实战：手工实现 VPN

1. 首先编写一个 c 语言程序 simpletun.c，代码如下：

```
[root@master1 ~]# vim simpletun.c                                    #程序代码如下
/*
Server: ip tuntap add mode tun name tun9,ip link set tun9 up,ip address add 192.168.2.11/24 dev tun9,./simpletun tun9 -s
Client: ip tuntap add mode tun name tun9,ip link set tun9 up,ip address add 192.168.2.31/24 dev tun9,./simpletun tun9 -c
```

```
<server_ip>
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <net/if.h>
#include <linux/if_tun.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>
#include <stdarg.h>

#define BUFSIZE 2000
#define PORT 55555

int tun_alloc(char *dev) {

        struct ifreq ifr;
        int fd, err;
        char *clonedev = "/dev/net/tun";

        if( (fd = open(clonedev , O_RDWR)) < 0 ) {
                return fd;
        }
        memset(&ifr, 0, sizeof(ifr));
        ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
        if (*dev) {
                strncpy(ifr.ifr_name, dev, IFNAMSIZ);
        }
        if( (err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ) {
                close(fd);
                return err;
        }
        strcpy(dev, ifr.ifr_name);
        return fd;
}

int cread(int fd, char *buf, int n){

        int nread;
        if((nread=read(fd, buf, n)) < 0) exit(1);
        return nread;
}

int cwrite(int fd, char *buf, int n){

        int nwrite;
        if((nwrite=write(fd, buf, n)) < 0) exit(1);
        return nwrite;
}

int read_n(int fd, char *buf, int n) {

        int nread, left = n;
        while(left > 0) {
                if ((nread = cread(fd, buf, left)) == 0){
                        return 0 ;
                }else {
                        left -= nread;
                        buf += nread;
                }
        }
        return n;
}

int main(int argc, char *argv[]) {

        int tun_fd, option;
        int maxfd;
        uint16_t nread, nwrite, plength;
        char buffer[BUFSIZE];
        struct sockaddr_in local, remote;
        unsigned short int port = PORT;
        int sock_fd, net_fd, optval = 1;
        socklen_t remotelen;
        unsigned long int tun2net = 0, net2tun = 0;

        if ( (tun_fd = tun_alloc(argv[1])) < 0 )  exit(1);
        if ( (sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)  exit(1);

        if(strcmp(argv[2],"-c")==0) {//vpn 的客户端
                memset(&remote, 0, sizeof(remote));
                remote.sin_family = AF_INET;
                remote.sin_addr.s_addr = inet_addr(argv[3]);
                remote.sin_port = htons(port);
                if (connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0)  exit(1);
                net_fd = sock_fd;

        } else {//vpn 的服务端
                if(setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval, sizeof(optval)) < 0)  exit(1);
                memset(&local, 0, sizeof(local));
                local.sin_family = AF_INET;
                local.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
                local.sin_port = htons(port);
                if (bind(sock_fd, (struct sockaddr*) &local, sizeof(local)) < 0)  exit(1);
                if (listen(sock_fd, 5) < 0) exit(1);
                remotelen = sizeof(remote);
                memset(&remote, 0, remotelen);
                if ((net_fd = accept(sock_fd, (struct sockaddr*)&remote, &remotelen)) < 0)  exit(1);
        }

/*到此建立了两个句柄：tun_fd 针对/dev/net/tun 文件，net_fd 针对网络*/
        maxfd = (tun_fd > net_fd)?tun_fd:net_fd;
        while(1) {
                int ret;
                fd_set rd_set;

                FD_ZERO(&rd_set);
                FD_SET(tun_fd, &rd_set);
                FD_SET(net_fd, &rd_set);
                ret = select(maxfd + 1, &rd_set, NULL, NULL, NULL);/*等待/dev/net/tun 文件或者网络有数据可读*/
                if (ret < 0 && errno == EINTR) continue;
                if (ret < 0)  exit(1);

                if(FD_ISSET(tun_fd, &rd_set)) {/*读/dev/net/tun 文件、写网络*/
                        nread = cread(tun_fd, buffer, BUFSIZE);
                        tun2net++;
                        plength = htons(nread);
                        nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
                        nwrite = cwrite(net_fd, buffer, nread);
                }

                if(FD_ISSET(net_fd, &rd_set)) {/*读网络、写/dev/net/tun 文件*/
                        nread = read_n(net_fd, (char *)&plength, sizeof(plength));
                        if(nread == 0)  break;
                        net2tun++;
                        nread = read_n(net_fd, buffer, ntohs(plength));
                        nwrite = cwrite(tun_fd, buffer, nread);
                }
        }
        return(0);
}
```

客户端和服务器建立连接之后，不管是客户端还是服务器都在等待两个句柄（net_fd 对应网络、tun_fd 对应/dev/net/tun 文件）上是否有数据可读，从 net_fd 读到的数据写入 tun_fd，反之从 tun_fd 读到的数据写入 net_fd。而应用程序只要针对 192.168.2.0/24 网络操作（套接口编程）即可，应用程序向网络发送的数据会到达本机的/dev/net/tun 文件，从而触发客户端（或服务器）的 tun_fd 句柄有数据可读。

2. 然后编译并运行（master1 充当服务器，worker1 充当客户端）：

```
[root@master1 ~]# gcc simpletun.c -o simpletun                                  #编译成可执行程序 simpletun
[root@master1 ~]# scp simpletun worker1:./  ------------------------------------ #把程序传到 worker1
[root@master1|worker1 ~]# ip tuntap add mode tun name tun9 ---------------------#在两台电脑上新建 tun9 设备
[root@master1|worker1 ~]# ip link set tun9 up                                   #激活 tun9 设备
[root@master1 ~]# ip address add 192.168.2.11/24 dev tun9  ---------------------- #给 tun9 分配 ip 地址
[root@worker1 ~]# ip address add 192.168.2.31/24 dev tun9
```

新建其他的 tun 设备也行（比如 tun7），而且两台电脑上新建的 tun 设备名称不一定要相同。

如果启动了防火墙，必须放行 55555/tcp 端口（或者干脆关闭防火墙 systemctl stop firewalld）：

```
[root@master1 ~]# firewall-cmd --add-port=55555/tcp
```

下面正式启动 vpn：

```
[root@master1 ~]# ./simpletun tun9 -s ------------------------------------------#作为服务器运行 simpletun
[root@worker1 ~]# ./simpletun tun9 -c 10.10.10.11
```

3. 验证：

保留上面的登录会话不动，我们再额外登录到 master1 和 worker1 中，然后执行：

```
[root@worker1 ~]# ping -I tun9 192.168.2.11
PING 192.168.2.11 (192.168.2.11) from 192.168.2.12 tun0: 56(84) bytes of data.
64 bytes from 192.168.2.11: icmp_seq=1 ttl=64 time=47.3 ms
64 bytes from 192.168.2.11: icmp_seq=2 ttl=64 time=48.3 ms
64 bytes from 192.168.2.11: icmp_seq=3 ttl=64 time=48.3 ms
64 bytes from 192.168.2.11: icmp_seq=4 ttl=64 time=47.4 ms
……

[root@master1 ~]# tcpdump -i ens160 ------------------------------------------------------------------------------#抓取隧道报文
……
```

https://www.verylo

```
20:27:07.168979 IP master1.55555 > 10.10.10.12.53186: Flags [.], ack 2, win 510, options [nop,nop,TS val 1293273681 ecr 925015128], length 0
20:27:07.169737 IP 10.10.10.12.53186 > vpn-server.55555: Flags [P.], seq 2:86, ack 1, win 502, options [nop,nop,TS val 925015174 ecr 1293273681], length 84
20:27:07.169832 IP master1.55555 > 10.10.10.12.53186: Flags [.], ack 86, win 510, options [nop,nop,TS val 1293273682 ecr 925015174], length 0
20:27:07.170057 IP master1.55555 > 10.10.10.12.53186: Flags [P.], seq 1:3, ack 86, win 510, options [nop,nop,TS val 1293273682 ecr 925015174], length 2
......
^C

[root@master1 ~]# tcpdump -i tun9 ------------------------------------------------------------------------------#抓取 ICMP 报文
......
20:28:37.137961 IP 192.168.2.12 > master1: ICMP echo request, id 7, seq 110, length 64
20:28:37.138012 IP master1 > 192.168.2.12: ICMP echo reply, id 7, seq 110, length 64
20:28:38.145071 IP 192.168.2.12 > master1: ICMP echo request, id 7, seq 111, length 64
20:28:38.145090 IP master1 > 192.168.2.12: ICMP echo reply, id 7, seq 111, length 64
......
^C
```

另外测试，只要退出 simpletun 程序，就 ping 不通了。

当然也可以从 master1 通过 vpn 登陆到 worker1 上：

```
[root@master1 ~]# ssh root@192.168.2.31
The authenticity of host '192.168.2.31 (192.168.2.31)' can't be established.
ED25519 key fingerprint is SHA256:i6+i+wS2exXmKsospCS9naY7pFCATpHMyk5AtKoHr10.
This host key is known by the following other names/addresses:
    ~/.ssh/known_hosts:1: worker1
    ~/.ssh/known_hosts:4: 20.20.20.8
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.2.31' (ED25519) to the list of known hosts.
root@192.168.2.31's password:
Last login: Fri Oct 18 17:17:49 2024 from 10.10.10.11
[root@worker1 ~]#
```

参考：https://vtun.sourceforge.net/

https://www.veryopen

§8.3 用户名字空间

用户名字空间由 User Namespace 翻译而来，用于隔离用户和用户组的 ID（unshare 命令的参数-U,--user）。一个比较典型的应用场景就是在宿主机上以非 root 用户运行的进程可以在另一个用户名字空间中映射成 root 用户。使用 User Namespace 可以实现进程在容器内拥有 root 权限，而在宿主机上却只是普通用户。验证：

```
[root@master1 ~]# useradd alice
[root@master1 ~]# su - alice
[alice@master1 ~]$ unshare --user -r /bin/bash
[root@master1 ~]# id                                                                    #已经进入了新的用户名字空间
uid=0(root) gid=0(root) groups=0(root)-----------------------------------------------#alice 在这个新的用户名字空间成为了 root 用户
[root@master1 ~]# shutdown -h now
Call to PowerOff failed: Access denied
[root@master1 ~]# exit
exit
```

尽管在新的用户名字空间内，alice 变为了 root，但是还没有权限关机。再看下面的例子：

```
[alice@master1 ~]$ unshare --user --map-root-user bash -c whoami
root
[alice@master1 ~]$ exit
logout
[root@master1 ~]#
```

用户名字空间需要 Linux 内核的支持，如果命令“grep CONFIG_USER_NS /boot/config-$(uname -r)”返回 CONFIG_USER_NS=y，表示内核支持了。

§8.4 文件系统名字空间

为了安全起见，运行在容器中的进程是不能访问根目录的，因此根文件系统必须隔离，也就是容器中也能访问“根”目录，但这个“根”目录不是宿主机上的根目录，它只是一个伪根目录。伪根目录在宿主机看来就是自己文件系统中的一个普通目录，只不过这个普通目录须要满足一定的目录结构才行。

下面我们来构造这样一个目录，并切换成伪根。

1. 先在宿主机上构造一个符合要求的目录/opt/namespace/sandbox，里面只有几个程序和相应的动态库文件。

操作如下：

```
[root@master1 ~]# mkdir -p /opt/namespace/sandbox/{bin,lib64}
```

此命令创建了两个目录：/opt/namespace/sandbox/bin 和/opt/namespace/sandbox/lib64。

拷贝几个程序：

```
[root@master1 ~]# cp -v /usr/bin/bash /opt/namespace/sandbox/bin/
'/usr/bin/bash' -> '/opt/namespace/sandbox/bin/bash'

[root@master1 ~]# cp -v /usr/bin/ls /opt/namespace/sandbox/bin/
'/usr/bin/ls' -> '/opt/namespace/sandbox/bin/ls'

[root@master1 ~]# cp -v /usr/bin/touch /opt/namespace/sandbox/bin/
'/usr/bin/touch' -> '/opt/namespace/sandbox/bin/touch'
```

拷贝这些程序用到的动态库文件：

```
[root@master1 ~]# ldd /usr/bin/bash | egrep -o '/lib.*\.[0-9]+'                    #查看/usr/bin/bash 的程序运行时
/lib64/libtinfo.so.6
/lib64/libc.so.6
/lib64/ld-linux-x86-64.so.2

[root@master1 ~]# cp /lib64/libtinfo.so.6 /opt/namespace/sandbox/lib64/
[root@master1 ~]# cp /lib64/libc.so.6 /opt/namespace/sandbox/lib64/
[root@master1 ~]# cp /lib64/ld-linux-x86-64.so.2 /opt/namespace/sandbox/lib64/

[root@master1 ~]# ldd /usr/bin/ls | egrep -o '/lib.*\.[0-9]+'                      #查看/usr/bin/ls 的程序运行时
/lib64/libselinux.so.1
/lib64/libcap.so.2
/lib64/libc.so.6
/lib64/libpcre2-8.so.0
/lib64/ld-linux-x86-64.so.2

[root@master1 ~]# cp /lib64/{libselinux.so.1,libcap.so.2,libpcre2-8.so.0} /opt/namespace/sandbox/lib64/

[root@master1 ~]# ldd /bin/touch | egrep -o '/lib.*\.[0-9]+'                      #查看/usr/bin/touch 的程序运行时
/lib64/libc.so.6
/lib64/ld-linux-x86-64.so.2
```

/usr/bin/ls 程序的动态库文件有两个与/usr/bin/bash 程序相同，所以没拷贝，因为前面已经拷贝了。另外/usr/bin/touch 的程序运行时不用拷贝，因为与前面几个程序一样，都已经拷贝了。

最后现实的“根”目录的结构如下：

```
[root@master1 ~]# tree /opt/namespace/sandbox/
/opt/namespace/sandbox/
├── bin
│   ├── bash
│   ├── ls
│   └── touch
└── lib64
    ├── ld-linux-x86-64.so.2
    ├── libcap.so.2
    ├── libc.so.6
    ├── libpcre2-8.so.0
    └── libselinux.so.1
```

```
    └── libtinfo.so.6

2 directories, 9 files
```

如果报没有 tree 命令，请采用命令“dnf install tree -y”安装 tree。

2．切换到伪根文件系统：

```
[root@master1 ~]# unshare -R /opt/namespace/sandbox
bash-5.1# pwd
/
bash-5.1#
```

切换成功，命令提示符变了，下面我们在伪根文件系统中运行一些命令：

```
bash-5.1# export PATH=/bin
bash-5.1# cd /                                                                                                    #改变到根目录
bash-5.1# ls -l                                                                                  #咦，根目录下只有两个子目录？
total 0
drwxr-xr-x 2 0 0  41 Jul 19 08:09 bin
drwxr-xr-x 2 0 0 137 Jul 19 08:11 lib64
bash-5.1# cd /lib64
bash-5.1# ls -l
total 4180
-rwxr-xr-x 1 0 0  841768 Jul 19 08:11 ld-linux-x86-64.so.2
-rwxr-xr-x 1 0 0 2387016 Jul 19 08:11 libc.so.6
-rwxr-xr-x 1 0 0   36944 Jul 19 08:11 libcap.so.2
-rwxr-xr-x 1 0 0  637872 Jul 19 08:11 libpcre2-8.so.0
-rwxr-xr-x 1 0 0  175552 Jul 19 08:11 libselinux.so.1
-rwxr-xr-x 1 0 0  191600 Jul 19 08:10 libtinfo.so.6
bash-5.1# cd /bin
bash-5.1# touch Hello1                                                                          #创建一个空文件 hello1
bash-5.1# ls
Hello1  bash  ls  touch
bash-5.1# exit                                                                                                  #退出伪根文件系统
exit

[root@master1 ~]# ls -l /opt/namespace/sandbox/bin
total 1592
-rwxr-xr-x 1 root root 1388920 Jul 19 16:09 bash
-rw-r--r-- 1 root root       0 Jul 19 16:34 Hello1
-rwxr-xr-x 1 root root  140760 Jul 19 16:09 ls
-rwxr-xr-x 1 root root   94048 Jul 19 16:09 touch
```

其中文件 Hello1 就是刚才在伪根里创建的空文件。宿主机上的目录/opt/namespace/sandbox/对应伪根中的/目录，同样的，宿主机上的目录/opt/namespace/sandbox/bin 就是伪根中的/bin 目录。

3．在伪根文件系统中使用宿主文件系统的目录。

正常情况下，伪根文件系统与宿主机文件系统是隔离的，即在伪根中没法访问外面的文件系统（宿主文件系统），而宿主文件系统可以访问伪根文件系统。

下面我们来实现：伪根访问宿主文件系统上的目录/opt/share：

```
[root@master1 ~]# mkdir /opt/share
[root@master1 ~]# mkdir /opt/namespace/sandbox/data
[root@master1 ~]# mount --bind /opt/share /opt/namespace/sandbox/data                                  #绑定两个目录
[root@master1 ~]# cp /etc/profile /opt/share
[root@master1 ~]# unshare -R /opt/namespace/sandbox
bash-5.1# cd /data
bash-5.1# ls -l
bash: ls: command not found
bash-5.1# export PATH=/bin
bash-5.1# ls -l
total 4
-rw-r--r-- 1 0 0 1899 Jul 19 08:50 profile
bash-5.1# echo "Hi, I am John">file.txt                                          #新建文件 file.txt，内容为“Hi, I am John”
bash-5.1# ls -l
total 8
-rw-r--r-- 1 0 0   14 Jul 19 08:53 file.txt
-rw-r--r-- 1 0 0 1899 Jul 19 08:50 profile
```

上面的命令“mount --bind ……”把/opt/share 目录绑定到/opt/namespace/sandbox/data 目录，以后往/opt/namespace/sandbox/data 目录中拷贝文件也就是往/opt/share 拷贝文件，对/opt/share 目录中的文件

做任何改变，也同时在/opt/namespace/sandbox/data 目录中反映出来。可以认为/opt/namespace/sandbox/data 是/opt/share 的别名。

保留本命令窗口不动，另外再登录到 master1，查看/opt/share 目录是不是多了文件：

```
[root@master1 ~]# ls -l /opt/share
total 8
-rw-r--r-- 1 root root   14 Jul 19 16:53 file.txt
-rw-r--r-- 1 root root 1899 Jul 19 16:50 profile
[root@master1 ~]# cat /opt/share/file.txt
Hi, I am John
```

到此我们实现了一个隔离的根文件系统。

§8.5 进程名字空间

进程名字空间由 pid namespace 翻译而来，它用来隔离进程号（unshare 命令和 nsenter 命令的参数-p,--pid），在不同的进程名字空间中，进程可以拥有相同的进程号，由此可以实现每个容器的主进程为 1 号进程，而容器内的进程在宿主机中却拥有不同的进程号。

```
[root@master1 ~]# unshare --fork --pid --mount-proc /bin/bash
[root@master1 ~]# ps -ax
    PID TTY      STAT   TIME COMMAND
      1 pts/1    S      0:00 /bin/bash
     17 pts/1    R+     0:00 ps -ax
```

换一个登录屏幕，在宿主机中查看进程：

```
[root@master1 ~]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
……
root        2092    1921  0 20:38 pts/1    00:00:00 unshare --fork --pid --mount-proc /bin/bash
root        2093    2092  0 20:38 pts/1    00:00:00 /bin/bash
root        2112       2  0 20:40 ?        00:00:00 [kworker/3:0-ata_sff]
root        2113    2055  0 20:40 pts/2    00:00:00 ps -ef
```

发现/bin/bash 的进程号为 2093，而在另一个进程名字空间中它的进程号为 1，参考上面红色字体部分。下面的命令在新的进程名字空间中读取本进程的进程号：

```
[root@master1 ~]# unshare --fork --pid --mount-proc readlink /proc/self
1
```

下面进一步验证进程名字空间，在上一小节构造的伪根目录中增加 ps 和 kill 程序。

```
[root@master1 ~]# which ps
/usr/bin/ps
[root@master1 ~]# cp --parent -v /usr/bin/ps /opt/namespace/sandbox/
'/usr/bin/ps' -> '/opt/namespace/sandbox/usr/bin/ps'
[root@master1 ~]# which kill
/usr/bin/kill
[root@master1 ~]# cp --parent -v /usr/bin/kill /opt/namespace/sandbox/
'/usr/bin/kill' -> '/opt/namespace/sandbox/usr/bin/kill'

 [root@master1 ~]# ldd /usr/bin/ps | egrep -o '/lib.*\.[0-9]+'
/lib64/libprocps.so.8
/lib64/libc.so.6
/lib64/libsystemd.so.0
/lib64/ld-linux-x86-64.so.2
/lib64/libcap.so.2
/lib64/libgcrypt.so.20
/lib64/liblzma.so.5
```

https://www.yeyupen.org

```
/lib64/libzstd.so.1
/lib64/liblz4.so.1
/lib64/libgcc_s.so.1
/lib64/libgpg-error.so.0
[root@master1 ~]# cp /lib64/{libprocps.so.8,libsystemd.so.0,liblzma.so.5} /opt/namespace/sandbox/lib64/
[root@master1 ~]# cp /lib64/{libzstd.so.1,liblz4.so.1,libgcc_s.so.1} /opt/namespace/sandbox/lib64/
[root@master1 ~]# cp /lib64/{libgpg-error.so.0,libgcrypt.so.20} /opt/namespace/sandbox/lib64/

[root@master1 ~]# ldd /usr/bin/kill | egrep -o '/lib.*\.[0-9]+'
/lib64/libc.so.6
/lib64/ld-linux-x86-64.so.2

[root@master1 ~]# mkdir /opt/namespace/sandbox/proc

[root@master1 ~]# tree /opt/namespace/sandbox/                                  #再检查以下目录结构
/opt/namespace/sandbox/
├── bin
│   ├── bash
│   ├── ls
│   └── touch
├── lib64
│   ├── ld-linux-x86-64.so.2
│   ├── libcap.so.2
│   ├── libc.so.6
│   ├── libgcc_s.so.1
│   ├── libgcrypt.so.20
│   ├── libgpg-error.so.0
│   ├── liblz4.so.1
│   ├── liblzma.so.5
│   ├── libpcre2-8.so.0
│   ├── libprocps.so.8
│   ├── libselinux.so.1
│   ├── libsystemd.so.0
│   ├── libtinfo.so.6
│   └── libzstd.so.1
├── proc
└── usr
    └── bin
        ├── kill
        └── ps

5 directories, 19 files
```

然后切换到伪根：

```
[root@vpn-server ~]# unshare --fork --pid -R /opt/namespace/sandbox -r --mount-proc
-bash-5.1# export PATH=/bin:/usr/bin
-bash-5.1# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
0              1       0  0 03:54 ?        00:00:00 -bash
0              3       1  0 03:54 ?        00:00:00 ps -e
-bash-5.1# bash&
[1] 5
-bash-5.1# ps
    PID TTY          TIME CMD
      1 ?        00:00:00 bash
      5 ?        00:00:00 bash
      7 ?        00:00:00 ps
-bash-5.1# kill -9 5
-bash-5.1# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
0              1       0  0 03:55 ?        00:00:00 -bash
0              9       1  0 03:57 ?        00:00:00 ps -ef
[1]+  Killed                  bash
-bash-5.1# exit
logout
```

至于 unshare 命令的用法，请参考§8.1.1 小节。

拓展学习：

1. proc 文件系统

§8.6 主机名字空间

主机名字空间由 UTS namespace 翻译而来，它用来隔离主机名字和域名（unshare 命令和 nsenter 命令的参数-u,--uts），它允许在每个主机名字空间内拥有一个独立的主机名称。例如我们的宿主机名称为 master1，但在另一个主机名字空间内的主机名可称为 worker 或者其他任意自定义主机名。

验证 1：

```
[root@master1 ~]# unshare -u bash
[root@master1 ~]#
```

看上去什么也没发生，来看一下进程：

```
[root@master1 ~]# ps --forest
    PID TTY          TIME CMD
   1809 pts/0    00:00:00 bash
   1915 pts/0    00:00:00  \_ bash
   1931 pts/0    00:00:00      \_ ps
```

进程号为 1915 的 bash 是我们刚才创建的，现在我们正身处这个 shell 中。现在来看一下 namespace 情况：

```
[root@master1 ~]# lsns
        NS TYPE   NPROCS   PID USER   COMMAND
……
4026532685 uts         2  1915 root   bash
4026532837 mnt         1  1103 root   /usr/sbin/rsyslogd -n
```

注意，这里 NPROCS 是 2，除了 bash 外，另一个是正在执行的 lsns 命令。如果在外面的 shell 中执行该命令，会看到该值为 1。

虽然这里只多出了一条 uts 空间，并不代表该进程只有这么一个空间。使用 --task <PID> 查看指定进程：

```
[root@abc ~]# lsns --task 1915
        NS TYPE   NPROCS   PID USER COMMAND
4026531834 time      333     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531835 cgroup    333     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531836 pid       333     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531837 user      332     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531839 ipc       333     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531840 net       332     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026531841 mnt       322     1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026532685 uts         3  1915 root bash
```

可以看到，除了 uts 其它仍然是继承的父进程的空间。现在来改一下 hostname：

```
[root@master1 ~]# hostname abc
[root@master1 ~]#
```

看上去没什么变化，在当前这个 bash 再另外启动一个 bash 看看：

```
[root@master1 ~]# bash
[root@abc ~]#
```

而在宿主机上，机器名并没有改变。现在回到「容器」内的 bash， 连续两次 exit 退出整个 bash，然后再次执行 lsns，可以看到标识为 4026532685 的 uts 空间也一并没有了。

验证 2：

```
[root@master1 ~]# PS1="ns2# " unshare --fork --uts /bin/bash
ns2# hostname                                                                    #进入了新的主机名字空间
master1
ns2# hostname -b worker ----------------------------------------------------------------------------------------------#修改主机名
ns2# hostname
worker
ns2# su - ------------------------------------------------------------------------------#以当前用户再次登录，新的主机名就其作用了
Last login: Wed Jul 19 20:54:05 CST 2023 on pts/1
[root@worker ~]# exit
logout
ns2# exit
exit
```

可以进一步验证宿主的主机名没有变化，看例子：

```
[root@master1 ~]# touch /root/uts-ns
[root@master1 ~]# unshare --uts=/root/uts-ns hostname foolish-me        #创建新的持久性主机名字空间，主机名改为 foolish-me
[root@master1 ~]# nsenter --uts=/root/uts-ns hostname ---------------------------#再次进入刚才创建的主机名字空间并显示主机名称
foolish-me
[root@master1 ~]# nsenter --uts=/root/uts-ns                                #再次进入刚才创建的主机名字空间并运行 bash 程序
[root@foolish-me ~]# hostname
foolish-me
[root@foolish-me ~]# pwd
/root
[root@foolish-me ~]# exit
[root@master1 ~]# hostname
master1
[root@master1 ~]# umount /root/uts-ns                                       #销毁刚才创建的主机名字空间
```

§8.7 时钟名字空间

时钟名字空间由英语 time namespace 翻译而来，它用于隔离 CLOCK_MONOTONIC 和 CLOCK_BOOTTIME 这两个系统时钟。另一个常见的系统时钟是 CLOCK_REALTIME，它是全局性的，跟时钟名字空间没有关系。这些系统时钟会影响到 clock_gettime()、clock_nanosleep()、nanosleep()、timer_settime()、timerfd_settime()函数的返回值，因为函数的第一实参就是指定系统时钟，比如 clock_gettime(CLOCK_BOOTTIME, &get1)，同时还会影响/proc/uptime 文件的值。

CLOCK_REALTIME 为系统的真实时间，可人为修改或被时钟同步改变，它是全局性的，不受时钟名字空间的影响。

CLOCK_MONOTONIC 为单调不减时间，表示从系统重启到现在过去了多少秒。CLOCK_MONOTONIC 时钟不受系统时间不连续跳跃的影响（例如，如果系统管理员手动更改 CLOCK_REALTIME 时钟），但受 adjtime()函数调用或 NTP 执行的增量调整的影响。该时钟不计算系统暂停期间的时间。前后多次获取 CLOCK_MONOTONIC 时钟不会出现时间倒退的现象，所以说它是单调不减的。

CLOCK_BOOTTIME 时钟与 CLOCK_MONOTONIC 类似，只是它包括系统挂起期间的时间。

当一个时钟名字空间内没有进程时，允许调整 CLOCK_MONOTONIC 和 CLOCK_BOOTTIME，一旦有进程入驻，就不能改变它们了，所以只能通过 unshare 命令的参数--monotonic、--boottime 来修改。

文件/proc/<pid>/timens_offsets 记录了进程<pid>所在时钟名字空间与宿主机的时钟名字空间相比较，CLOCK_MONOTONIC 和 CLOCK_BOOTTIME 的偏移量，根宿主机内的/proc/<pid>/timens_offsets 总是为 0。

/proc/<pid>/timens_offsets 包含两行，格式为：

```
monotonic           <秒数>          <纳秒数>
boottime            <秒数>          <纳秒数>
```

第二列<秒数>可正可负，第三列为正整数，比如“monotonic -2 50”表示与宿主机相差-2*1000000000+50 纳秒。

文件/proc/<pid>/ns/time 指明进程<pid>入驻的时钟名字空间，而文件/proc/<pid>/ns/time_for_children 指明进程<pid>的子进程入驻的时钟名字空间，进程和它的子进程允许属于不通的时钟名字空间。

看实验：

```
[root@master1 ~]# cat /proc/self/timens_offsets                             #宿主机上总是为 0
```

https://www.veryopen.org

```
monotonic          0         0
boottime           0         0

新建时钟名字空间，CLOCK_MONOTONIC 比宿主机早 5 秒，而 CLOCK_BOOTTIME 快 8 天：
[root@master1 ~]# PS1="tns# " unshare -T --monotonic -5 --boottime $((8*60*60))
tns# cat /proc/self/timens_offsets
monotonic          -5         0
boottime        28800         0
tns# uptime --pretty
up 8 hours, 26 minutes
tns# echo "boottime -100" > /proc/$$/timens_offsets
-bash: echo: write error: Invalid argument
tns# readlink /proc/$$/ns/time_for_children
time:[4026532585]
tns# exit
logout
[root@master1 ~]# uptime --pretty
up 43 minutes
```

§8.8 IPC 名字空间

IPC 名字空间又称为进程间通讯名字空间，它的用来隔离 System V IPC 对象和 POSIX 消息队列的（unshare 命令和 nsenter 命令的参数-i,--ipc）。在不通的 IPC 名字空间内，以下文件的内容会不同：/proc/sys/fs/mqueue、/proc/sys/kernel/{msgmax,msgmnb,msgmni,sem,shmall,shmmax,shmmni,shm_rmid_forced} 、/proc/sysvipc。

IPC 名字空间需要内核支持，执行命令“grep CONFIG_IPC_NS /boot/config-$(uname -r)”返回 CONFIG_IPC_NS=y，表示内核支持了。

如果 PID 名字空间和 IPC 名字空间一起使用，则可以实现同一 IPC 名字空间内的进程彼此可以通信，不同 IPC 名字空间的进程不能通信的目的。

实战：

```
[root@master1 ~]# PS1="ipc# " unshare --fork --ipc /bin/bash
ipc# ipcs -q                                                    #查看进程间通信队列列表

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

ipc# ipcmk -Q                                                   #创建进程间通信队列
Message queue id: 0
ipc# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x9c52f7b9 0          root       644        0            0

ipc# exit
[root@master1 ~]# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

在宿主机上执行命令“ipcs -q”是看不到刚才创建的进程通信队列的。

c.biancheng.net/www.verypen.org

§8.9 挂载点名字空间

挂载点名字空间由 mount namespace 翻译而来。在一个隔离的挂载点名字空间中，挂载（mount）和卸载（umount）文件系统不会影响其他挂载点名字空间，即在不同的名字空间中看到不同的挂载目录，除非你在挂载时明确指明了“共享”参数（即 mount --make-shared）。unshare 带参数-m,--mount 即可新建挂载点名字空间。验证：

```
[root@master1 ~]# unshare --mount --fork /bin/bash
[root@master1 ~]# PS1="mnt# " ------------------------------------------------------------------------#已经进入另一个挂载点名字空间
mnt# mkdir /tmp/tmpfs
mnt# mount -t tmpfs -o size=20m tmpfs /tmp/tmpfs
mnt# df -T
Filesystem     Type     1K-blocks    Used Available Use% Mounted on
/dev/sda1      xfs       33487872 2157560  31330312   7% /
devtmpfs       devtmpfs      4096       0      4096   0% /dev
tmpfs          tmpfs      1990300       0   1990300   0% /dev/shm
tmpfs          tmpfs       796124    9208    786916   2% /run
tmpfs          tmpfs       398060       0    398060   0% /run/user/0
tmpfs          tmpfs        20480       0     20480   0% /tmp/tmpfs
```

为了验证宿主系统中并没有挂载此目录，我们新打开一个命令行窗口，同样执行 df 命令查看宿主机的挂载信息：

```
[root@master1 ~]# df -T
Filesystem     Type     1K-blocks    Used Available Use% Mounted on
devtmpfs       devtmpfs      4096       0      4096   0% /dev
tmpfs          tmpfs      1990300       0   1990300   0% /dev/shm
tmpfs          tmpfs       796124    9208    786916   2% /run
/dev/sda1      xfs       33487872 2157580  31330292   7% /
tmpfs          tmpfs       398060       0    398060   0% /run/user/0
```

同样在宿主中新挂载的文件系统，在另一个挂载点名字空间中也看不到，这就实现了挂载点名字空间的隔离。在两个挂载点名字空间执行命令“ls -l /proc/self/ns/”查看名字空间的 Id 号，发现只有 mnt 不同（表示不同的挂载点名字空间）。可执行命令“exit”退出挂载点名字空间。

§8.10 手工创建准容器

我们后面要接触的容器就是对应 linux 中的进程，只不过限制了该进程的根文件系统、网络名字空间和各种资源（内存和 CPU 等）的名字空间。本小节使用 linux 操作系统的一些命令，我们手工来构造一个准容器，以便加深对容器的理解。

§8.10.1 准容器实战

本小节我们采用 unshare 命令来具体构建一个准容器，以此来加深理解容器的底层原理。

https://www.verypo

1．准备文件系统

准备迷你型的文件系统有两种方法：第一种方法是从网上下载别人已经构造好的，第二种方法自己手工来构造。下面的两种方法建议你采用第二种。

第一种方法

从 http://dl-cdn.alpinelinux.org/alpine 下载最新的迷你型文件系统，然后解压。

```
[root@master1 ~]# wget http://dl-cdn.alpinelinux.org/alpine/v3.20/releases/x86_64/alpine-minirootfs-3.20.3-x86_64.tar.gz
[root@master1 ~]# mkdir -p /opt/rootfs
[root@master1 ~]# tar -xf alpine-minirootfs-3.20.3-x86_64.tar.gz -C /opt/rootfs              #解压到/opt/rootfs 目录中
[root@master1 ~]# du -sh /opt/rootfs
7.4M    /opt/rootfs_alpine ----------------------------------------------------------------------------------------------------------------#一共 7.4 兆
[root@master1 ~]# tree /opt/rootfs
/opt/rootfs_alpine
├── bin
│   ├── busybox
│   ├── cat -> /bin/busybox
│   ├── cp -> /bin/busybox
│   ├── hostname -> /bin/busybox
│   ├── ls -> /bin/busybox
│   ├── sh -> /bin/busybox
│   └── .....
├── dev
├── etc
│   ├── conf.d
│   ├── fstab
│   ├── group
│   ├── hostname
│   ├── hosts
│   ├── network
│   ├── passwd
│   ├── profile
│   ├── sysctl.conf
│   └── .....
├── home
├── lib
│   ├── libcrypto.so.3
│   ├── libssl.so.3
│   ├── libz.so.1.2.13
│   └── sysctl.d
│       └── 00-alpine.conf
├── media
│   ├── cdrom
│   ├── floppy
│   └── usb
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
│   ├── ip -> /bin/busybox
│   └── .....
├── srv
├── sys
├── tmp
├── usr
│   ├── bin
│   │   ├── id -> /bin/busybox
│   │   ├── vi -> /bin/busybox
│   │   └── .....
│   ├── lib
│   │   └── .....
│   ├── local
│   │   ├── bin
│   │   ├── lib
│   │   └── share
│   ├── sbin
│   │   ├── adduser -> /bin/busybox
│   │   └── .....
│   └── share
│       ├── man
│       ├── .....
│       └── udhcpc
│           └── default.script
└── var
    ├── cache
    │   └── misc
    ├── empty
    ├── local
    ├── lock
    │   └── subsys
```

```
    ├── log
    ├── mail
    ├── opt
    ├── run -> /run
    ├── spool
    │   ├── cron
    │   │   └── crontabs -> /etc/crontabs
    │   └── mail -> /var/mail
    └── tmp
```

可见 alpine 是采用开源项目 busybox 构建的，绝大多数的命令都是符号连接到/bin/busybox 程序，这个迷你型文件系统虽小，但“五官”俱全。

第二种方法

采用 supermin 命令手工构造迷你文件系统：

```
[root@master1 ~]# dnf install supermin -y
```

添加一些预装软件（bash,coreutils,dnf,iproute,vim-minimal,openssh-server,openssh-clients,iputils,procps-ng），并预备好目录 supermin.d：

```
[root@master1 ~]# supermin -v --prepare bash coreutils dnf iproute vim-minimal openssh-server openssh-clients \
 iputils procps-ng -o supermin.d
supermin: version: 5.3.3
supermin: rpm: detected RPM version 4.16
supermin: rpm: detected RPM architecture x86_64
supermin: package handler: fedora/rpm
supermin: prepare: bash coreutils dnf iproute vim-minimal openssh-server openssh-clients iputils procps-ng
supermin: packages specified on the command line:
  - bash-5.1.8-9.el9.x86_64
  - coreutils-8.32-36.el9.x86_64
  - dnf-4.14.0-17.el9.noarch
……
```

开始构建迷你文件系统，构建好的目录为/opt/rootfs：

```
[root@master1 ~]# rm -rf /opt/rootfs                                    #先确保/opt/rootfs 不能存在
[root@master1 ~]# supermin -v --build --format chroot supermin.d -o /opt/rootfs
supermin: version: 5.3.3
supermin: rpm: detected RPM version 4.16
supermin: rpm: detected RPM architecture x86_64
supermin: package handler: fedora/rpm
supermin: build: supermin.d
supermin: reading the supermin appliance
supermin: build: visiting supermin.d/base.tar.gz type gzip base image (tar)
supermin: build: visiting supermin.d/packages type uncompressed packages
supermin: mapping package names to installed packages
supermin: resolving full list of package dependencies
supermin: build: 162 packages, including dependencies
supermin: build: 31528 files
supermin: build: 31528 files, after matching excludefiles
supermin: build: 31528 files, after adding hostfiles
supermin: build: 15903 files, after removing unreadable files
supermin: build: 15932 files, after munging
supermin: renaming /opt/rootfs.p7nn7vco to /opt/rootfs

[root@master1 ~]# du -sh /opt/rootfs
339M    /opt/rootfs                                    #由于安装了一些大型软件，所以这个文件系统不算“迷你”了，呵呵
```

由于安装诸如 bash、openssh-server、openssh-clients、dnf、iputils 等较大型的软件，所以构建出来的“迷你”文件系统远远大于上面第一种方法构建的文件系统。

Debian 发行版也有一个构建迷你文件系统的命令 debootstrap，大家不妨去了解以下。

另外再登录一次到 master1 虚拟机，保持两个登录屏幕，下面要用到两个登录屏幕。

2．产生一个没有网卡的准容器

在 Linux 下，“nohup <命令> &”表示在后台执行“命令”，且命令与控制台分离——当退出登陆转台后不会影响命令的

执行。

```
[root@master1 ~]# nohup unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/rootfs --wd=/root \
 --mount-proc=/opt/rootfs/proc --map-root-user --kill-child /bin/sleep 5h &
[1] 14091

[root@master1 ~]# ps --forest
    PID TTY          TIME CMD
  14043 pts/0    00:00:00 bash
  14091 pts/0    00:00:00  \_ unshare
  14092 pts/0    00:00:00  |   \_ sleep
  14176 pts/0    00:00:00  \_ ps
```

新建了 8 个名字空间，它们分别是：--ipc（进程间通讯名字空间）、--uts（主机名字空间）、--net（网络名字空间）、--user（用户和组名字空间）、--mount（挂载点名字空间）、--pid（进程名字空间）、--cgroup（控制组名字空间）和--root（文件系统名字空间，即把/opt/rootfs 作为文件系统的根）。

参数--map-root-user 表示进入准容器后转换为 0 号用户（即 root），如果省略-map-root-user 参数，就转换为 nobody 用户，nobody 用户比 root 用户的权限小很多。参数“-w /root”表示进入准容器后的当前目录改为/root。参数--mount-proc 表示把 proc 文件系统挂载到/proc 目录（对应宿主文件系统中的/opt/rootfs/proc 目录）上。参数--kill-child 隐式包含了--fork 参数，表示在宿主系统中产生一个子进程并在子进程中运行/bin/sleep 5h 程序，当父进程（运行了 unshare 程序的那个进程）终止时，会向子进程发 SIGKILL 信号，这是杀死进程的信号。命令/bin/sleep 5h 表示睡眠 5 个小时，即我们创建的准容器能存活 5 个小时，在这 5 小时内足可以完成接下来的实验。由 unshare 新建名字空间并运行某个程序，当这个程序退出时，unshare 也就会结束并销毁全部新建的名字空间，所以这次运行的程序 sleep 5h 尽管在睡眠但并未退出，所以 unshare 能一直维持到睡眠结束。

查看进程：

```
[root@master1 ~]# ps -ef | grep -e /bin/sleep -e PID
UID          PID    PPID  C STIME TTY          TIME CMD
root       14091   14043  0 09:59 pts/0    00:00:00 unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/rootfs --wd=/root --mount-proc --map-root-user --
kill-child /bin/sleep 5h
root       14092   14091  0 09:59 pts/0    00:00:00 /bin/sleep 5h
root       14178   14043  0 10:10 pts/0    00:00:00 grep --color=auto -e /bin/sleep -e PID
```

注意第二列 PID（进程号）和第三列 PPID（父进程号），发现/bin/sleep 5h 的进程号为 14092，父进程号是 14091，而 14091 就是“unshare --ipc ……”的进程号，由此可见 unshare 产生了子进程并在子进程中运行/bin/sleep 5h，这就是--fork 参数的意义，因为--kill-child 隐式包含了--fork 参数。

查看相关的名字空间：

```
[root@master1 ~]# lsns | grep -e /bin/sleep -e COMMAND
        NS TYPE   NPROCS   PID USER   COMMAND
4026532688 user         4 14091 root   unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/rootfs …… --mount-proc --map-root-user --kill-child /bin/sleep 5h
4026532689 mnt          4 14091 root   unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/rootfs …… --mount-proc --map-root-user --kill-child /bin/sleep 5h
4026532690 uts          4 14091 root   unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/rootfs …… --mount-proc --map-root-user --kill-child /bin/sleep 5h
4026532691 ipc          4 14091 root   unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/rootfs …… --mount-proc --map-root-user --kill-child /bin/sleep 5h
4026532692 pid          2 14092 root   /bin/sleep 5h
4026532693 cgroup       4 14091 root   unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/rootfs …… --mount-proc --map-root-user --kill-child /bin/sleep 5h
4026532694 net          4 14091 root   unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/rootfs …… --mount-proc --map-root-user --kill-child /bin/sleep 5h
```

可通过/proc/<pid>/ns/<namespace>来访问进程 pid 的名字空间，如/proc/14092/ns/net 就是访问 14092 号进程的网络名字空间。后续的实验要用到这些名字空间。

下面再次进入准容器（注意：14092 就是 sleep 5h 的进程号，前面的命令已经显示出来了）：

```
[root@master1 ~]# PS1="container# " nsenter -t 14092 -a --root=/opt/rootfs --wd=/root /bin/sh
shell-init: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
container# cd
chdir: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
container# pwd
/root
container# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
container#
```

给变量 PS1 赋值"container# "表示进入容器后的命令行提示符改为 container# 。

3. 添加网卡

https://www.xryopen.org

图 8-12 为最终构造的网络图。

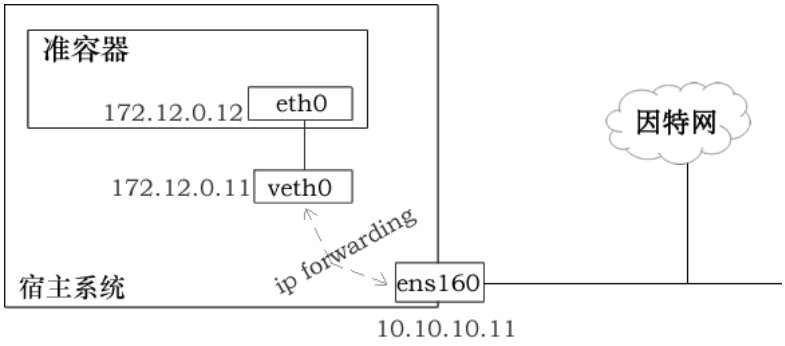


图 8-12 准容器连接外网

在宿主机上操作：

```
[root@master1 ~]# ip link add veth0 type veth peer name eth0                    #创建一对 veth 设备: veth0 <---> eth0
[root@master1 ~]# ip link set veth0 up                                          #激活 veth0
[root@master1 ~]# ip addr add 172.12.0.11/24 dev veth0                          #给 veth0 配置 ip: 172.12.0.11
[root@master1 ~]# ip link set eth0 netns /proc/14092/ns/net                     #把 eth0 移到/proc/14092/ns/net 网络名字空间中
```

14092 就是准容器中“/bin/sleep 5h”进程号，可在宿主机上用命令“ps -ef|grep -e sleep -e PID |grep -v unshare”查看进程号。

在准容器里操作：

```
container# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
4: eth0@if5: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN qlen 1000          #看到了网卡了
    link/ether fe:56:89:36:02:8c brd ff:ff:ff:ff:ff:ff
container# ip link set lo up
container# ip link set eth0 up
container# ip addr add 172.12.0.12/24 dev eth0
container# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP qlen 1000
    link/ether fe:56:89:36:02:8c brd ff:ff:ff:ff:ff:ff
    inet 172.12.0.12/24 scope global ceth0
       valid_lft forever preferred_lft forever
    inet6 fe80::fc56:89ff:fe36:28c/64 scope link
       valid_lft forever preferred_lft forever
container# ping -c 3 172.12.0.11
PING 172.12.0.11 (172.12.0.11): 56 data bytes
64 bytes from 172.12.0.11: seq=0 ttl=64 time=0.050 ms
64 bytes from 172.12.0.11: seq=1 ttl=64 time=0.083 ms
64 bytes from 172.12.0.11: seq=2 ttl=64 time=0.083 ms

--- 172.12.0.11 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.050/0.072/0.083 ms
container# ping 9.9.9.9
PING 9.9.9.9 (9.9.9.9): 56 data bytes
ping: sendto: Network unreachable
container#
```

但是准容器没法与外网连通，下面来解决这个问题。

4. 让准容器连接外网

在准容器中添加默认路由：

```
container# ip route add default via 172.12.0.11                                 #添加默认路由
container# ip r
default via 172.12.0.11 dev eth0
```

```
172.12.0.0/24 dev eth0 scope link  src 172.12.0.12
```

在宿主系统中执行（做 SNAT）下面的命令做 SNAT 策略：

```
[root@master1 ~]# echo 1 > /proc/sys/net/ipv4/ip_forward                                    #允许内核转发
[root@master1 ~]# firewall-cmd --direct --passthrough ipv4 -t nat -A POSTROUTING -s 172.12.0.0/24 -o ens160 \
 -j MASQUERADE
[root@master1 ~]# iptables -t nat -A POSTROUTING -s 172.12.0.0/24 -o ens160 -j MASQUERADE
[root@master1 ~]# firewall-cmd --add-rich-rule="rule family=ipv4 source address=192.168.2.0/24 masquerade"
```

在准容器中执行：

```
container# ping -c 3 9.9.9.9
PING 9.9.9.9 (9.9.9.9): 56 data bytes
64 bytes from 9.9.9.9: seq=0 ttl=127 time=75.481 ms
64 bytes from 9.9.9.9: seq=1 ttl=127 time=114.859 ms
64 bytes from 9.9.9.9: seq=2 ttl=127 time=82.151 ms

--- 9.9.9.9 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 75.481/90.830/114.859 ms

container# ping www.baidu.com
ping: bad address 'www.baidu.com'
```

现在能 ping 通外面的 ip，但是没法做 dns 解析，下面来解决这个问题。

5．让准容器能做 DNS 解析

```
container# vi /etc/resolv.conf                                                              #增加两个 DNS
nameserver 8.8.8.8
nameserver 9.9.9.9

container# ping www.baidu.com
PING www.baidu.com (119.63.197.139): 56 data bytes
64 bytes from 119.63.197.139: seq=0 ttl=127 time=106.145 ms
64 bytes from 119.63.197.139: seq=1 ttl=127 time=117.922 ms
64 bytes from 119.63.197.139: seq=2 ttl=127 time=102.316 ms
^C
--- www.baidu.com ping statistics ---
4 packets transmitted, 3 packets received, 25% packet loss
round-trip min/avg/max = 102.316/108.794/117.922 ms
```

6．在准容器种启动 sshd 服务

一旦准容器启动了 sshd 服务，我们就可以从远程登录到准容器了。

(1)．把宿主机上的/dev/目录绑定到容器中：

```
[root@master1 ~]# mount --rbind /dev /opt/rootfs/dev
[root@master1 ~]# mount --rbind /proc /opt/rootfs/proc
```

(2)．在容器中启动 sshd 服务：

```
container# ssh-keygen -A  ------------------------------------------------------------------ #产生主机公/私钥（在/etc/ssh/）

container# useradd -u 74 -d /usr/share/empty.sshd -s /sbin/nologin sshd 2>/dev/null                    #创建用户 sshd
container# mkdir /var/run/sshd
container# echo "PermitRootLogin yes" > /etc/ssh/sshd_config.d/01-permitrootlogin.conf                 #允许 root 登陆
container# /usr/sbin/sshd ------------------------------------------------------------------------#在后台运行 sshd 服务程序
[1] 9
container# ss -tnlp
State          Recv-Q         Send-Q                Local Address:Port                Peer Address:Port
Process
LISTEN         0              128                         0.0.0.0:22                       0.0.0.0:*
LISTEN         0              128                            [::]:22                          [::]:*
```

从最后这条命令的输出可知 22 号端口处于监听状态，不过还不能远程登录到准容器中，还需要在宿主机中做端口映射（DNAT）：

```
[root@master1 ~]# firewall-cmd --add-masquerade
[root@master1 ~]# firewall-cmd --add-forward-port=port=220:proto=tcp:toport=22:toaddr=172.12.0.12
```

即把宿主机上的 220/tcp 映射到 172.12.0.12:22/tcp，但是系统中并没有监听 220 端口（用 ss -tnlp 查看），

这是防火墙在请求进入系统前就拦截并转向了。现在就可以从其他电脑登录到我们的准容器了：

```
[root@worker1 ~]# ssh -p 220 root@10.10.10.11
```

但是报错：connection reset by ……，表示链接被服务器复位。在准容器里，执行命令 su 拨错 su: cannot set groups: Operation not permitted，因此首先需要解决这个问题，即在准容器中能执行 su 命令。

§8.10.2 解剖容器

请先参考§3.1.1 小节安装 docker，并配置好国内镜像源。下面我们来解剖一个容器镜像的结构，busybox 容器镜像很小，正好拿来解剖，麻雀虽小五脏俱全。

```
[root@worker3 ~]# docker pull busybox
Using default tag: latest
latest: Pulling from library/busybox
3f4d90098f5b: Pull complete
Digest: sha256:3fbc632167424a6d997e74f52b878d7cc478225cffac6bc977eedfe51c7f4e79
Status: Downloaded newer image for busybox:latest
docker.io/library/busybox:latest
[root@worker3 ~]# docker images
REPOSITORY     TAG       IMAGE ID       CREATED       SIZE
busybox        latest    a416a98b71e2   2 weeks ago   4.26MB
[root@worker3 ~]# docker image save -o busybox.tar busybox
[root@worker3 ~]# ll busybox.tar
-rw------- 1 root root 4501504 Aug  4 14:26 busybox.tar
[root@worker3 ~]# mkdir busybox
[root@worker3 ~]# tar -xf busybox.tar -C busybox/
[root@worker3 ~]# cd busybox
[root@worker3 busybox]# ll
total 12
drwxr-xr-x 2 root root   50 Jul 19 07:19 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77
-rw-r--r-- 1 root root 1457 Jul 19 07:19 a416a98b71e224a31ee99cff8e16063554498227d2b696152a9c3e0aa65e5824.json
-rw-r--r-- 1 root root  203 Jan  1  1970 manifest.json
-rw-r--r-- 1 root root   90 Jan  1  1970 repositories
```

先来看看 repositories 文件（用 vim 打开文件后输入:%!python3 -m json.tool 格式化代码）：

```
[root@worker3 busybox]# vim repositories
{
    "busybox": {
        "latest": "866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77"
    }
}

[root@worker3 busybox]# vim manifest.json
[
    {
        "Config": "a416a98b71e224a31ee99cff8e16063554498227d2b696152a9c3e0aa65e5824.json",
        "RepoTags": [
            "busybox:latest"
        ],
        "Layers": [
            "866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77/layer.tar"
        ]
    }
]


[root@worker3 busybox]# vim a416a98b71e224a31ee99cff8e16063554498227d2b696152a9c3e0aa65e5824.json
{
    "architecture": "amd64",
    "config": {
        "Hostname": "",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "Cmd": [
            "sh"
```

```
        ],
        "Image": "sha256:919de79e94ce464098cec8a4796f27aa2d0c93985e27f6a157c8bbb557568013",
        "Volumes": null,
        "WorkingDir": "",
        "Entrypoint": null,
        "OnBuild": null,
        "Labels": null
    },
    "container": "adfa050237a5fcf9f1e5ad6351b3665e4b99e0c12f33bdd3535b8fb001f9dd2e",
    "container_config": {
        "Hostname": "adfa050237a5",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "Cmd": [
            "/bin/sh",
            "-c",
            "#(nop) ",
            "CMD [\"sh\"]"
        ],
        "Image": "sha256:919de79e94ce464098cec8a4796f27aa2d0c93985e27f6a157c8bbb557568013",
        "Volumes": null,
        "WorkingDir": "",
        "Entrypoint": null,
        "OnBuild": null,
        "Labels": {}
    },
    "created": "2023-07-18T23:19:33.655005962Z",
    "docker_version": "20.10.23",
    "history": [
        {
            "created": "2023-07-18T23:19:33.538571854Z",
            "created_by": "/bin/sh -c #(nop) ADD file:7e9002edaafd4e4579b65c8f0aaabde1aeb7fd3f8d95579f7fd3443cef785fd1 in / "
        },
        {
            "created": "2023-07-18T23:19:33.655005962Z",
            "created_by": "/bin/sh -c #(nop)  CMD [\"sh\"]",
            "empty_layer": true
        }
    ],
    "os": "linux",
    "rootfs": {
        "type": "layers",
        "diff_ids": [
            "sha256:3d24ee258efc3bfe4066a1a9fb83febf6dc0b1548dfe896161533668281c9f4f"
        ]
    }
}

[root@worker3 busybox]# ll
total 12
drwxr-xr-x 2 root root   50 Jul 19 07:19 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77
-rw-r--r-- 1 root root 1457 Jul 19 07:19 a416a98b71e224a31ee99cff8e16063554498227d2b696152a9c3e0aa65e5824.json
-rw-r--r-- 1 root root  203 Jan  1  1970 manifest.json
-rw-r--r-- 1 root root   90 Jan  1  1970 repositories

[root@worker3 busybox]# cd 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77/
[root@worker3 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77]# ll
total 4396
-rw-r--r-- 1 root root    1134 Jul 19 07:19 json
-rw-r--r-- 1 root root 4492288 Jul 19 07:19 layer.tar
-rw-r--r-- 1 root root       3 Jul 19 07:19 VERSION
[root@worker3 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77]# more VERSION
1.0

[root@worker3 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77]# vim json
{
    "id": "866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77",
    "created": "2023-07-18T23:19:33.655005962Z",
    "container": "adfa050237a5fcf9f1e5ad6351b3665e4b99e0c12f33bdd3535b8fb001f9dd2e",
    "container_config": {
        "Hostname": "adfa050237a5",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "Cmd": [
            "/bin/sh",
            "-c",
            "#(nop) ",
            "CMD [\"sh\"]"
```

```
        ],
        "Image": "sha256:919de79e94ce464098cec8a4796f27aa2d0c93985e27f6a157c8bbb557568013",
        "Volumes": null,
        "WorkingDir": "",
        "Entrypoint": null,
        "OnBuild": null,
        "Labels": {}
    },
    "docker_version": "20.10.23",
    "config": {
        "Hostname": "",
        "Domainname": "",
        "User": "",
        "AttachStdin": false,
        "AttachStdout": false,
        "AttachStderr": false,
        "Tty": false,
        "OpenStdin": false,
        "StdinOnce": false,
        "Env": [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
        ],
        "Cmd": [
            "sh"
        ],
        "Image": "sha256:919de79e94ce464098cec8a4796f27aa2d0c93985e27f6a157c8bbb557568013",
        "Volumes": null,
        "WorkingDir": "",
        "Entrypoint": null,
        "OnBuild": null,
        "Labels": null
    },
    "architecture": "amd64",
    "os": "linux"
}

[root@worker3 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77]# mkdir /tmp/rootfs
[root@worker3 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77]# tar -xf layer.tar -C /tmp/rootfs
[root@worker3 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77]# du -sh /tmp/rootfs/
4.2M    /tmp/rootfs/
[root@worker3 866320838f32a620adb16ae0d1537e4792766eefcb8555850ec12850503d5a77]# cd /tmp/rootfs
[root@worker3 rootfs]# ll
total 20
drwxr-xr-x 2 root   root   12288 Jul 18 02:30 bin
drwxr-xr-x 2 root   root       6 Jul 18 02:30 dev
drwxr-xr-x 3 root   root     100 Jul 18 02:30 etc
drwxr-xr-x 2 nobody nobody     6 Jul 18 02:30 home
drwxr-xr-x 2 root   root    4096 Jul 18 02:30 lib
lrwxrwxrwx 1 root   root       3 Jul 18 02:30 lib64 -> lib
drwx------ 2 root   root       6 Jul 18 02:30 root
drwxrwxrwt 2 root   root       6 Jul 18 02:30 tmp
drwxr-xr-x 4 root   root      29 Jul 18 02:30 usr
drwxr-xr-x 4 root   root      30 Jul 18 02:30 var

[root@worker3 rootfs]# tree /tmp/rootfs
/tmp/rootfs
├── bin
│   ├── [
│   ├── [[
│   ├── arp
│   ├── busybox
│   ├── bzcat
│   ├── bzip2
│   ├── cal
│   ├── cat
│   ├── chat
│   ├── chattr
│   ├── chgrp
│   ├── passwd
│   ├── stty
│   ├── su
│   ├── ......                                                    #省略了 389
个命令
│   ├── sum
│   └── zcip
├── dev
├── etc
│   ├── group
│   ├── localtime
│   ├── network
│   │   ├── if-down.d
│   │   ├── if-post-down.d
│   │   ├── if-pre-up.d
│   │   └── if-up.d
│   ├── nsswitch.conf
│   ├── passwd
│   └── shadow
├── home
├── lib
│   ├── ld-linux-x86-64.so.2
```

```
|   ├── libc.so.6
|   ├── libm.so.6
|   ├── libnss_compat.so.2
|   ├── libnss_dns.so.2
|   ├── libnss_files.so.2
|   ├── libnss_hesiod.so.2
|   ├── libpthread.so.0
|   └── libresolv.so.2
├── lib64 -> lib
├── root
├── tmp
├── usr
|   ├── bin
|   |   └── env -> ../../bin/env
|   └── sbin
└── var
    ├── spool
    |   └── mail
    └── www

20 directories, 420 files
```

由于绝大多数的命令是/bin/busybox 的硬链接文件，所以尽管包含这么多的文件和目录，但总体大小还不到 5 兆。

下面启动容器镜像，然后再导出文件系统，看看结构有什么变化：

```
[root@master1 ~]# docker run -it busybox sleep 1h&
[1] 16048
[root@master1 ~]# docker ps
CONTAINER ID   IMAGE     COMMAND      CREATED         STATUS         PORTS     NAMES
0dc3668fe634   busybox   "sleep 1h"   8 seconds ago   Up 7 seconds             friendly_colden
[root@master1 ~]# docker export 0dc3668fe634 -o /tmp/busybox_runnning.tar
[root@master1 ~]# mkdir /tmp/busybox_runnning
[root@master1 ~]# tar -xf /tmp/busybox_runnning.tar -C /tmp/busybox_runnning/
```

看看运行中的容器的根文件系统的结构（/bin 和/lib 目录与前面的一样，所以我把它们删除了）：

```
[root@master1 ~]# tree /tmp/busybox_runnning
/tmp/busybox_runnning
├── dev
|   ├── console
|   ├── pts/
|   └── shm/
├── etc
|   ├── group
|   ├── hostname
|   ├── hosts
|   ├── localtime
|   ├── mtab -> /proc/mounts
|   ├── network
|   |   ├── if-down.d
|   |   ├── if-post-down.d
|   |   ├── if-pre-up.d
|   |   └── if-up.d
|   ├── nsswitch.conf
|   ├── passwd
|   ├── resolv.conf
|   └── shadow
├── home
├── lib64 -> lib
├── proc/
├── root
├── sys/
├── tmp
├── usr
|   ├── bin
|   |   └── env -> ../../bin/env
|   └── sbin
└── var
    ├── spool
    |   └── mail
    └── www

24 directories, 425 files
```

发现与镜像相比，多了四个目录和五个文件。

再进入 0dc3668fe634 容器看看挂载的设备：

```
[root@master1 ~]# docker exec -it 0dc3668fe634 sh
/ # mount
overlay on / type overlay
(rw,relatime,lowerdir=/var/lib/docker/overlay2/l/QFZS4G6EALD7MKNKTMA47FI4PQ:/var/lib/docker/overlay2/l/MZNZKPD2MCT3
RL7CBFALTAKRL4,upperdir=/var/lib/docker/overlay2/52e4e2779c42571a88e26bb401c113712a199037804b64ed153ff27a332782a0/d
iff,workdir=/var/lib/docker/overlay2/52e4e2779c42571a88e26bb401c113712a199037804b64ed153ff27a332782a0/work)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev type tmpfs (rw,nosuid,size=65536k,mode=755,inode64)
```

```
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=666)
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup type cgroup2 (ro,nosuid,nodev,noexec,relatime,nsdelegate,memory_recursiveprot)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime)
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=65536k,inode64)
/dev/sda1 on /etc/resolv.conf type xfs (rw,relatime,attr2,inode64,logbufs=8,logbsize=32k,noquota)
/dev/sda1 on /etc/hostname type xfs (rw,relatime,attr2,inode64,logbufs=8,logbsize=32k,noquota)
/dev/sda1 on /etc/hosts type xfs (rw,relatime,attr2,inode64,logbufs=8,logbsize=32k,noquota)
devpts on /dev/console type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=666)
proc on /proc/bus type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/fs type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/irq type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sys type proc (ro,nosuid,nodev,noexec,relatime)
proc on /proc/sysrq-trigger type proc (ro,nosuid,nodev,noexec,relatime)
tmpfs on /proc/asound type tmpfs (ro,relatime,inode64)
tmpfs on /proc/acpi type tmpfs (ro,relatime,inode64)
tmpfs on /proc/kcore type tmpfs (rw,nosuid,size=65536k,mode=755,inode64)
tmpfs on /proc/keys type tmpfs (rw,nosuid,size=65536k,mode=755,inode64)
tmpfs on /proc/timer_list type tmpfs (rw,nosuid,size=65536k,mode=755,inode64)
tmpfs on /proc/scsi type tmpfs (ro,relatime,inode64)
tmpfs on /sys/firmware type tmpfs (ro,relatime,inode64)
```

多出来的目录和文件清楚了：`/proc` 目录上挂载了 `proc` 文件系统，`/sys` 目录上挂载了 `sysfs` 文件系统，`/dev/pts` 上挂载了 `devpts` 文件系统，`/dev/shm` 目录上挂载了 `shm` 文件系统，这些都是虚拟文件系统，驻留在内存中。而多出来的几个文件是把宿主机上的相应文件绑定而来（采用类似这样的命令 `mount --bind /etc/resolv.conf /path/to/etc/resolv.conf`）。

而挂载虚拟文件系统和绑定宿主机上的文件到底什么时候执行的？这要看运行容器的具体过程了。

1. 从镜像库拉下 `debian` 镜像：

```
[root@master1 ~]# docker pull debian
[root@master1 ~]# docker images
REPOSITORY     TAG       IMAGE ID       CREATED       SIZE
debian         latest    278727474c29   7 days ago    116MB
```

2. 启动软件并安装一些软件：

```
[root@master1 ~]# docker run -it debian
root@e65c29c0757d:/# apt update -y

root@e65c29c0757d:/# apt install iproute2 vim-tiny -y
root@e65c29c0757d:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
15: eth0@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
root@e65c29c0757d:/# passwd root
New password:
Retype new password:
passwd: password updated successfully
root@e65c29c0757d:/# echo 'export PS1="container-sshd# "' >> ~/.bashrc
root@e65c29c0757d:/# echo 'export PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"' >> ~/.bashrc
```

3. 安装远程登录服务：

```
root@e65c29c0757d:/# apt install openssh-server -y
root@e65c29c0757d:/# mkdir /run/sshd
root@e65c29c0757d:/# vi /etc/ssh/sshd_config
......
PermitRootLogin yes
......

root@e65c29c0757d:/# /usr/sbin/sshd
root@e65c29c0757d:/# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 02:12 pts/0    00:00:00 bash
root        1317       1  0 02:24 ?        00:00:00 sshd: /usr/sbin/sshd [listener] 0 of 10-100 startups
root        1328       1  0 02:30 pts/0    00:00:00 ps -ef
root@e65c29c0757d:/# ss -tnlp
State    Recv-Q   Send-Q       Local Address:Port       Peer Address:Port   Process
LISTEN   0        128                0.0.0.0:22              0.0.0.0:*       users:(("sshd",pid=1317,fd=3))
```

```
LISTEN   0        128                [::]:22              [::]:*       users:(("sshd",pid=1317,fd=4))
```

4．换个登录屏幕看看能否登录到容器：

```
[root@master1 ~]# ssh root@172.17.0.2
root@172.17.0.2's password:
Linux e65c29c0757d 5.14.0-333.el9.x86_64 #1 SMP PREEMPT_DYNAMIC Wed Jun 28 09:47:27 UTC 2023 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
container-sshd# exit
logout
Connection to 172.17.0.2 closed.
```

退出容器：

```
root@e65c29c0757d:/# exit
```

5．将停止的容器导出来，并解压到/opt/debian_sshd 目录中：

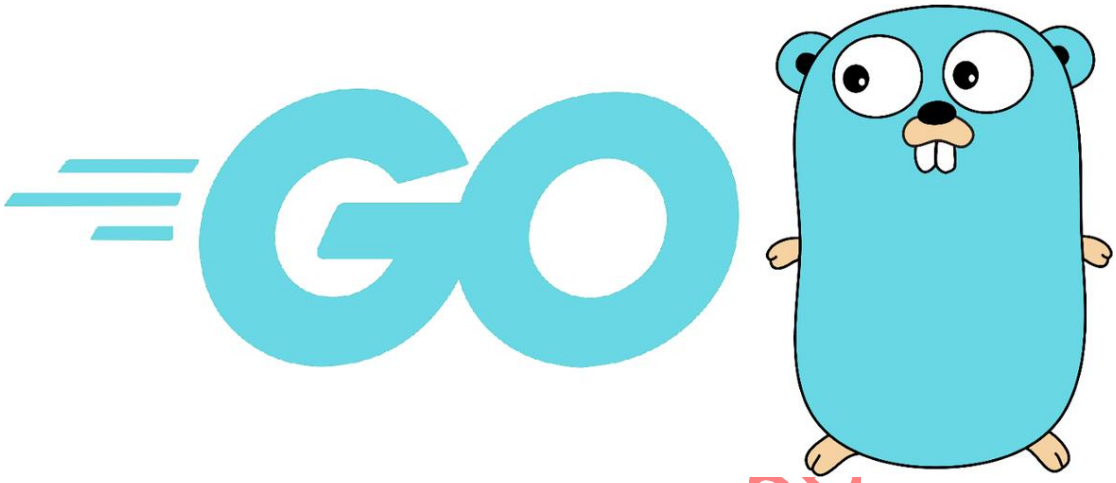
```
[root@master1 ~]# docker ps -a
CONTAINER ID   IMAGE          COMMAND   CREATED             STATUS                       PORTS     NAMES
e65c29c0757d   debian         "bash"    23 minutes ago      Exited (0) 12 seconds ago              hardcore_rhodes
[root@master1 ~]# docker export e65c29c0757d --output=debian_sshd.tar
[root@master1 ~]# ll debian_sshd.tar
-rw------- 1 root root 199396864 Aug  4 10:37 debian_sshd.tar
[root@master1 ~]# tar -xf debian_sshd.tar -C /opt/debian_sshd/
[root@master1 ~]# du -sh /opt/debian_sshd/
206M    /opt/debian_sshd/
```

6．产生一个准容器：

```
[root@master1 ~]# unshare --ipc --uts --net --user --mount --pid --cgroup --root=/opt/debian_sshd --wd=/root \
 --mount-proc --map-root-user --kill-child bash
container-sshd#
```

https://www.ver...en.org

第9章 Go 语言



§9.1 Go 语言简介

Go 语言（或 Golang）起源于 2007 年，并在 2009 年正式对外发布。Go 语言的发明者都是计算机科学领域的重量级人物，又有谷歌公司的加持，不能不说它是出生名门望族。Go 是非常年轻的一门语言，它的主要目标是“兼具 Python 等动态语言的开发速度和 C/C++ 等编译型语言的性能与安全性”，因此不能不说它填补了计算机编程语言的一个空白——兼顾解释型语言的开发速度和编译型语言的执行效率。

Go 语言是编程语言设计的又一次尝试，是对类 C 语言的重大改进，它不但能让你访问底层操作系统，还提供了强大的网络编程和并发编程支持。Go 语言的用途众多，可以进行网络编程、系统编程、并发编程、分布式编程等，在云计算、大数据和元宇宙时代会大放异彩。Go 语言的推出，旨在不损失应用程序性能的情况下降低代码的复杂性，具有“部署简单、并发性好、语言设计良好、执行性能好”等优势，目前国内诸多 IT 公司均已采用 Go 语言开发项目。

Go 语言有时候被描述为“C 类似语言”，或者是“21 世纪的 C 语言”。Go 从 C 语言继承了相似的表达式语法、控制流结构、基础数据类型、调用参数传值、指针等很多思想，还有 C 语言一直所引以为傲的编译后机器码的运行效率以及和现有操作系统的无缝适配。Go 语言没有类和继承的概念，“类”和“继承”使得很多初学者望而生畏，所以 go 抛弃了它们，转而采用简洁易懂的接口来实现诸如多态这种面向对象编程的特征。

因为 Go 语言没有类和继承的概念，所以它和 Java 或 C++看起来并不相同。但是它通过接口（interface）的概念来实现多态性。Go 语言有一个清晰易懂的轻量级类型系统，在类型之间也没有层级之说。因此可以说 Go 语言是一门混合型的语言。

此外，很多重要的开源项目都是使用 Go 语言开发的，其中包括 Docker、Go-Ethereum、Thrraform 和 Kubernetes 等。

也许有人会问，我已经会 c 语言或 Python 语言，为什么还要花时间和精力再去学 Go 语言呢？我的理由如下：

https://www.veryop

1． 如果你要创建系统程序，或者基于网络的程序，Go 语言是很不错的选择。作为一种相对较新的语言，它是由经验丰富且受人尊敬的计算机科学家设计的，旨在应对创建大型并发网络程序面临的挑战。

2． 在 Go 语言出现之前，开发者们总是面临非常艰难的抉择，究竟是使用执行速度快但是编译速度并不理想的语言（如：C++），还是使用编译速度较快但执行效率不佳的语言（如：.NET、Java），或者说开发难度较低但执行速度一般的动态语言（如 php、Python）呢？显然，Go 语言在这 3 个条件之间做到了最佳的平衡：快速编译，高效执行，易于开发。

3．Go 语言支持交叉编译，比如说你可以在运行 Linux 系统的计算机上开发可以在 Windows 上运行的应用程序。这是第一门完全支持 UTF-8 的编程语言，这不仅体现在它可以处理使用 UTF-8 编码的字符串，就连它的源码文件格式都是使用的 UTF-8 编码。Go 语言做到了真正的国际化！

4． Go 语言为并发而生！在云计算、大数据、元宇宙和人工智能时代，其实就是并发的时代——大规模并行计算的时代！你学会了 Go，你就跟上了时代，懂不？

本章对 go 语言做个简单介绍，至于更高级的如反射、并发等请参考其他书籍或者网上材料。

go 语言学习网站 https://www.runoob.com/go/go-tutorial.html 、http://c.biancheng.net/view/77.html

https://www.veryopen.org

§9.2 编程环境

1．安装 go 语言：

```
[root@master1 ~]# dnf install golang golang-docs -y
[root@master1 ~]# go env
GO111MODULE=""
GOARCH="amd64"
GOBIN=""
GOCACHE="/root/.cache/go-build"
GOENV="/root/.config/go/env"
GOEXE=""
GOEXPERIMENT=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOINSECURE=""
GOMODCACHE="/root/go/pkg/mod"
GONOPROXY=""
GONOSUMDB=""
GOOS="linux"
GOPATH="/root/go"
GOPRIVATE=""
GOPROXY="https://proxy.golang.org,direct"
GOROOT="/usr/lib/golang"
GOSUMDB="sum.golang.org"
GOTMPDIR=""
GOTOOLDIR="/usr/lib/golang/pkg/tool/linux_amd64"
……

[root@master1 ~]# go help                                          #获得全局帮助
Go is a tool for managing Go source code.

Usage:
```

```
        go <command> [arguments]

The commands are:

        bug         start a bug report
        build       compile packages and dependencies
        clean       remove object files and cached files
        doc         show documentation for package or symbol
        env         print Go environment information
        fix         update packages to use new APIs
        fmt         gofmt (reformat) package sources
        generate    generate Go files by processing source
        get         add dependencies to current module and install them
        install     compile and install packages and dependencies
        list        list packages or modules
        mod         module maintenance
        work        workspace maintenance
        run         compile and run Go program
        test        test packages
        tool        run specified go tool
        version     print Go version
        vet         report likely mistakes in packages

Use "go help <command>" for more information about a command.
……

[root@master1 ~]# go help build                                          #如何编译
usage: go build [-o output] [build flags] [packages]

Build compiles the packages named by the import paths,
along with their dependencies, but it does not install the results.
……
```

编译 go 语言的命令语法为“go build [-o output] [build flags] [packages]”，其中参数-o 指明编译后的程序文件名，packages 指明源程序。比如 go build abc.go 就是把 abc.go 编译为 abc 可执行程序，再比如 go build -o myHello hello.go 把 hello.go 编译为 myHello。

2. “Hello World!”例子：

```
[root@master1 ~]# vim hello.go
package main

import "fmt"

func main() {

        /*向世界问个好吧！*/
        fmt.Println("Hello World!")
}

[root@master1 ~]# go run hello.go                                        #运行程序
Hello World!
```

go 语言源程序可以直接执行，这点像 python 或 php 语言，当然也可以编译成可执行程序的，见下面一个例子，

3. 再来个 99 乘法表的例子：

```
[root@master1 ~]# vim 99.go
package main

import "fmt"

func main() {

        // 定义整型变量 y
        var y int

        // 遍历，决定处理第几行
        for x := 1; x <= 9; x++ {

                // 遍历，决定这一行有多少列
                for y = 1; y <= x; y++ {
                        fmt.Printf("%d*%d=%d ", y, x, x*y)
                }

                // 换行
                fmt.Println()
        }
}
```

```
[root@master1 ~]# go build -o my99 99.go                                    #编译成 my99 可执行程序
[root@master1 ~]# ll *99*
-rw-r--r-- 1 root root     316 Aug  5 07:42 99.go
-rwxr-xr-x 1 root root 1870070 Aug  5 08:30 my99

[root@master1 ~]# ./my99                                                    #运行程序
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6 3*3=9
1*4=4 2*4=8 3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81

[root@master1 ~]# ldd my99                                                  #检查 my99 程序的运行时
        not a dynamic executable
```

最后一条命令检查 my99 程序需要的运行时，反馈说是非动态可执行程序，也就是静态编译的。静态编译的程序在运行的时候不依赖任何动态库文件，所以允许随意拷贝到任何电脑上去执行，当然除了 windows 外。

§9.3 go 语法

§9.3.1 定义变量

1. go 语言的数据类型有：

(1). 布尔类型：bool，取值只能是 true 或 false；

(2). 整型：有符号整型 int8/byte、int16、int32、int64 和 int，无符号整型 uint8、uint16、uint32、uint64 和 uint。其中 int 和 uint 与 cpu 的字长相关，比如在流行 64 位 CPU 电脑上，一个 int 型变量能保存 8 个字节的数据。

(3). 浮点型：float32、float64。

(4). 复数：complex128（64 位实数和虚数）、complex64（32 位实数和虚数）。

(5). 字符串：string，为 UTF-8 编码标识 Unicode 文本。

(6). 结构体：普通变量的集合体。

```
type 结构体名 struct {                                          例子:          type Student struct
{
        成员名 1 数据类型                                                          name string
        成员名 1 数据类型                                                          age uint8
                ……                                                             sex bool
}                                                                          }
```

(7). 接口：方法的集合体。

```
type 接口名 interface {                                         例子:          type Shape interface {
    方法名 1( [参数列表 1] ) ( [返回值列表 1] )                                        area() float64
    方法名 2( [参数列表 2] ) ( [返回值列表 2] )                                        length() (float64)
```

https://www.veryopen.org

```
    …                                                                    }
}
```

2．定义变量的例子：

```
var i int                                                                                    //定义整型变量 i
var x, y float32                                                                             //定义两个浮点型变量 x,y
var firstname string = "Alice"
k := 10                                                                                      //定义整型变量 k 并初始化为 10
tag, str := 1024, "2 的 10 次方"                                    //定义整型变量 tag 并赋值 1024，定义字符串变量 str 并赋值"2 的 10 次方"

var a *int                                                                                   //定义整型指针变量 a
var scores [60]uint8                                                                         //定义 60 个元素的数组 scores
var numbers = [5]int{1, 2, 3, 4, 5}                                                          //通过赋值定义数组
balance := [...]float32{1000.0, 2.0, 3.4, 7.0, 50.0}                                        //根据值的个数确定数组元素个数

var (
        a int
        b bool
)                                                                                            //定义两个变量 a 和 b

type Book struct {
          name string
          author string
          price float32
}
var book1 Book
```

§9.3.2　控制语句

1.条件语句

两路分支 if：

```
if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
}

if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
} else {
  /* 在布尔表达式为 false 时执行 */
}

if 布尔表达式 1 {
        // do something
} else if 布尔表达式 2 {
        // do something else
} else {
        // catch-all or default
}
```

举例：

```
var ten int = 11
if ten > 10 {
    fmt.Println(">10")
} else {
    fmt.Println("<=10")
}
```

多路分支 switch：

```
switch var1 {
    case val1:
```

```
        ...
    case val2:
        ...
    default:
        ...
}
```

举例：

```
package main

import "fmt"

func main() {
   var grade string = "B"
   var marks int = 75

   switch marks/10 {
      case 9,10: grade = "A"
      case 8: grade = "B"
      case 5,6,7 : grade = "C"
      default: grade = "D"
   }
   fmt.Println("你的成绩是: ", grade);
}
```

分支语句还有 select。

2. 循环语句

for 循环：

```
for init; condition; post {
          //循环体
}

for 条件 { }

break                                                    //跳出循环
continue                                                 //继续下轮循环
goto                                                     //无条件跳转
```

举例：

```
var puzzle [10]int

for i := 0; i < 10; i++ {
          puzzle[i] = i
          fmt.Printf "%d ", puzzle[(i+7)%10])
}
```

https://www.[illegible]vopen.cn

§9.3.3 函数

声明函数的语法如下：

```
func 函数名( [形参列表] ) [(返回值列表)] {
   函数体
}

形参列表：变量 1 数据类型, 变量 2 数据类型, …, 变量 m, 变量 n, … 数据类型
返回值列表：数据类型, 数据类型, … | 变量 1 数据类型, 变量 2 数据类型, …

例如：
func mymax(x, y int) (int) { ... }
```

```
func abc1(name string, age int, amount, price float32) (float32, string) { … }
```

返回值列表只能二选一，即要么是“数据类型，数据类型，…”，要么是“变量 1 数据类型，变量 2 数据类型，…”。

看例子：

```
[root@master1 ~]# vim test1.go
package main

import "fmt"

func max1(x int, y int) (int) {
        if x > y {
                return x
        } else{
                return y
        }
}

func swap(x, y int) (int, int) {
        return y, x
}

func tst(x, y int, name string) (n string, z int) {
        z = x * y
        n = name
        return
}

func main() {
        var ret int
        ret = max1(10,99)
        fmt.Println(ret)

        i, j := swap(88, 77)
        fmt.Println(i,j)

        var name string
        var total int
        name, total = tst(i,j,"Alice")
        fmt.Println(name,total)
}

[root@master1 ~]# go run test1.go
99
77 88
Alice 6776
```

https://www

§9.3.4 指针

1. 内存单元

要掌握指针，你必须先了解内存单元、内存地址等。

一个字节由 8 位二进制组成，一个内存单元存储一个字节，每个内存单元有一个唯一的编号，这个编号称为内存地址，我们是通过内存地址来访问内存单元的，比如说“请把数字 8 保存在 128 号内存单元内，谢谢!”平时人们说自己的电脑有 8G 内存，更准确表述就是“我的电脑有 8G 个内存单元。”也就是 8589934592 个内存单元（8G=8*1024*1024*1024），因此给这些内存单元的编号（地址）依次为0,1,2,3,4,5,6,...., 8589934591。

另外我们平时所说的 64 位 cpu 是指什么呢？就是指这款 cpu 使用 64 位二进制来表示一个内存单元地址，用 64 位二进制来表示一个整数，分别对应 cpu 上的 64 根针脚（或触点）。表示内存地址的 64 个针脚又与电脑主板上 64 根地址总线连接在一起，表示整数的 64 个针脚另外与电脑主板上 64 根数据总线连接在一起。所以你如果拆开电脑仔细看主

板，就会发现线路板上布满密密麻麻的排线，这些排线主要是内存地址总线和数据总线。

用 64 位二进制来表示一个内存单元地址，所以总共能编号的内存单元总数达2^{64}，这几乎是个天文数字！而 32 位 CPU 至多能用到2^{32}个内存单元，也就是 4G 内存，不过现在的电脑一般都是 64 位的。

假如在 64 位的电脑上只插入 4G 的内存条，那么有效的内存单元地址范围就是：0,1,2,3,...,4294967291，现在如果 CPU 要读取 11 号内存单元里的数据，那么 CPU 仍然要向内存发送 64 位二进制表示的地址 11，即：

```
0000000000000000000000000000000000000000000000000000000000001011
```

十进制 11 转换成二进制就是 1011，所以前面需要补上 60 个零，一共凑齐 64 位二进制。但是在我们编程时很少直接采用二进制，更习惯使用十六进制表示的内存地址，不过需要添加 0x 前缀以示区别，因此 11 号内存单元的十六进制地址就是：

```
0x000000000000000b                                                  //0x 与 b 之间有 15 个十六进制 0
```

二进制 0000 转换为十六进制后还是 0，二进制 1011 转换为十六进制就是 b。

有人说“请把 123456 这个整数保存在 0x000000000000009a 号内存单元”（不妨假设在 64 位电脑上）。下面我们看看具体是怎么存储的。

在 64 位电脑上，一个整数就是用 64 位二进制表示，因此十进制数 123456 转换成 64 位二进制就是：

```
0000000000000000000000000000000000000000000000011110001001000000
```

64 位二进制数就是 8 个字节，所以需要 8 个内存单元才能存储这个数，显然需要使用从 0x000000000000009a 内存地址开始的连续 8 个内存单元。为了区分，不妨给不同的字节打上不同的颜色，如下所示：



这样颜色不同的 8 位二进制分别存储在不同的内存单元里，存储方式有两种，术语叫做“大端存储”或者“小端存储”，参见图 9-1 所示。

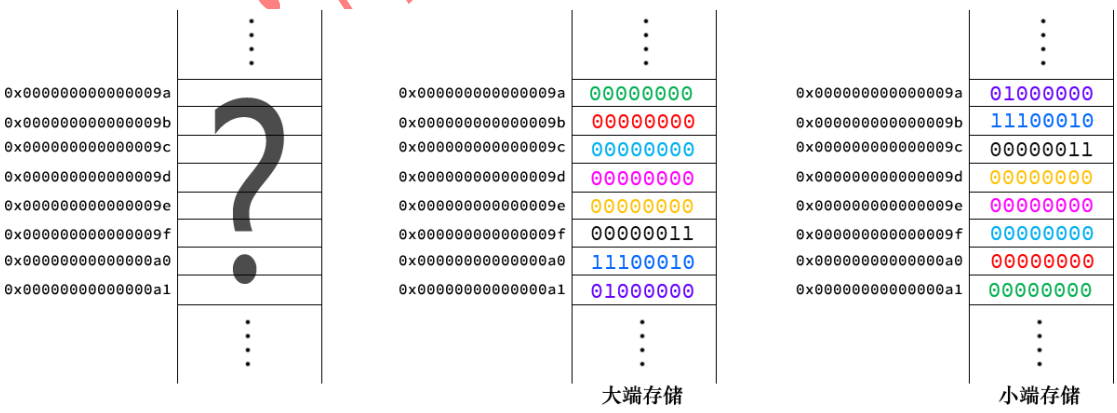


图 9-1 内存单元即存储方式

把高位字节存储在地址较小的内存单元就称为大端存储，而把低位字节存储在地址较小的内存单元称为小端存储，不同的操作系统会采用不同的存储方法，linux 操作系统上 go 语言采用小端存储。

2．指针变量

指针变量的值是内存地址。指针变量的定义语法如下：

```
var 变量名 *数据类型

如：
var ip *int32                                                    //定义整型 int32 指针变量 ip
var fp2 *float32                                                 //定义浮点型 float32 指针变量 fp2
```

指针是强类型的，比如不能把 int 类型的指针指向 uint16 型变量，但可以通过 unsafe.Pointer()来实现指针的类型转换，如：

```
var i uint16
var *pr uint8
pr = (*uint8)(unsafe.Pointer(&i))
```

看一个例子：

```
[root@master1 ~]# vim ptr.go
package main
import "fmt"

func main() {

        var i int32
        var ip1 *int32

        fmt.Println(i, ip1)

        ip1 = &i                                                 //获取 i 的内存地址并赋给 ip1

        i = 1989
        fmt.Println(i, *ip1, &i)

        *ip1 = 1234                                              //把 ip1 所指到的那个内存地址处的内存单元改为 1234
        fmt.Println(i, *ip1, ip1, &ip1)
}

[root@master1 ~]# go run ptr.go
0 <nil>
1989 1989 0xc0000180d0
1234 1234 0xc0000180d0   0xc000012028
```

i 变量在内存中的首地址为 0xc0000180d0，补齐地址中的高位 0，即 0x000000c0000180d0，转换为二进制就是：

```
0000000000000000000000001100000000000000000000011000000011010000
```

ip1 变量在内存中的地址是 0xc000012028。下面我们看看执行上面的语句过程中的内存变化。

(1)．执行这两条语句后的内存参考图 8-2 中①部分：

```
        var i int32
        var ip1 *int32
```

(2)．执行“ip1 = &i”语句，把变量 i 的内存地址保存到变量 ip1 中，请参考图 8-2 中②部分。&i 表示获取变量 i 的内存地址，众所周知，定义的变量都是保存在内存中。

(3)．执行“i = 1989”语句。十进制 1989 对应二进制为 011111000101，高位补齐 0 使得二进制位数达到 32（因为 i 是 32 位整数）：

```
00000000000000000000011111000101
```

一共四个字节，以小端存储后的内存参见图 8-2 中③部分。

(4)．最后执行“*ip1 = 1234”语句。由上面第(2)点可知，变量 ip1 存储了变量 i 的内存地址，所以*ip1 就等价于 i 了，因此给*ip1 赋值 1234 就等价于给 i 赋值 1234。大家可以试试这条语句“fmt.Println(&i, &(*ip1))”打印出来的两个值是不是相等。十进制 1234 转换成二进制并高位补足 0 就是：

```
00000000000000000000010011010010
```

最后内存的样子就是图 9-2 中④部分。

https://www.

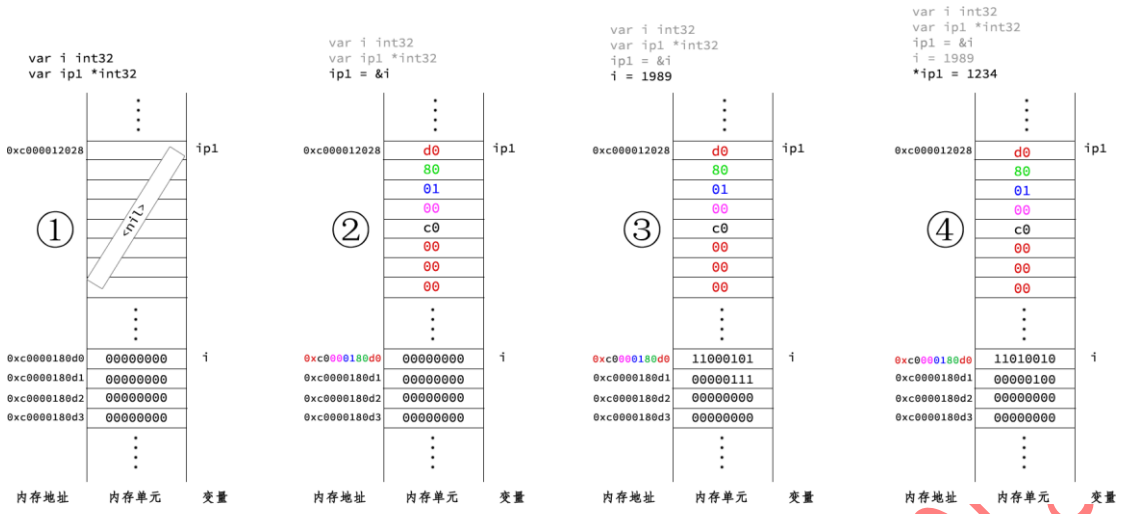


图 9-2 内存随着语句执行而发生变化

*ip1 表示“以变量 ip1 的值为内存地址，此地址所对应的内存单元”，而从图 9-2 中②部分可知，ip1 变量的值就是 0x000000c0000180d0，即变量 i 的内存地址 0xc0000180d0 高位补足 0 得到的。

下面的程序可以判断是小端还是大端存储：

```
[root@master1 ~]# vim borl.go
package main
import "fmt"
import "unsafe"

func main() {

        var i uint16
        var ip1 *byte

        ip1 = (*byte)(unsafe.Pointer(&i))

        i = 0x100

        if *ip1 == 0x1 {
                fmt.Println("大端存储")
        }else{
                fmt.Println("小端存储")
        }
}

[root@master1 ~]# go run borl.go
小端存储
```

ip1 为字节类型的指针，所以*ip1 代表一个内存单元的值，即变量 i 的低地址内存单元的值。

§9.4 综合例子

§9.4.1 多态结构

下面的例子用两个不同的结构体类型实现同一个接口。

```
[root@master1 ~]# vim com.go
```

```
package main

import "fmt"

type Shape interface {
        getArea() float64
        getPerimeter() float64
}

type Rectangle struct {
        width float64
        height float64
}

func (r Rectangle) getArea() float64 {                                  //Rectangle 结构体类型实现 getArea()方法
        return r.width * r.height
}

func (r Rectangle) getPerimeter() float64 {                             //Rectangle 结构体类型实现 getPerimeter()方法
        return 2 * ( r.width + r.height )
}

type Circle struct {
        radius float64
}

func (c Circle) getArea() float64 {                                     //Circle 结构体类型实现 getArea()方法
        return 3.14 * c.radius * c.radius
}

func (c Circle) getPerimeter() float64 {                                //Circle 结构体类型实现 getPerimeter()方法
        return 2 * 3.14 * c.radius
}

func main() {
        var s Shape
        var rec Rectangle
        var cir Circle = Circle{ radius: 3 }

        rec.width = 10
        rec.height = 5
        s = rec                                         //s 引用 Rectangle 结构体实例，s.getArea()就是 Rectangle 实现的方法
        fmt.Printf("矩形面积: %f, 边长: %f\n", s.getArea(),s.getPerimeter())

        s = cir
        fmt.Printf("圆形面积: %f, 周长: %f\n", s.getArea(),s.getPerimeter())
}

[root@master1 ~]# go run com.go
矩形面积: 50.000000, 边长: 30.000000
圆形面积: 28.260000, 周长: 18.840000
```

§9.4.2 制作容器

采用几行 go 代码产生一个容器。首先参考“§8.10.1 准容器实战”小节的准备文件系统的第一种方法，准备号文件系统的目录/opt/rootfs。

```
[root@master1 ~]# mkdir /opt/rootfs/mytemp
```

编写 go 语言程序 mydocker.go：

```
[root@master1 ~]# vim mydocker.go
package main

import (
        "fmt"
        "os"
        "os/exec"
        "path/filepath"
        "strconv"
        "syscall"
)

// 运行: go run main.go <cmd> <args>, 例如 go run mydocker.go /bin/sh
// 注: os.Args[0]是可执行文件本身的路径, 大概长这样: /tmp/go-build2286914496/b001/exe/ mydocker
func main() {
        fmt.Printf("Running %v \n", os.Args[1:])
        createCgroupFiles() // 创建 cgroup 相关的文件夹和文件
```

https://ww

```
        cmd := buildCmd()    // 构建命令
        prepareContainer()   // 准备容器
        defer unmount()      // 卸载，延迟进行
        must(cmd.Run())      // 执行命令，前面调用了 syscall.Chroot，这里实际执行的是：/opt/root/fs/bin/sh
}

/* 创建 cgroup 相关的文件夹和文件 */
func createCgroupFiles() {
        cgroups := "/sys/fs/cgroup/"
        _ = os.Mkdir(filepath.Join(cgroups, "wlm"), 0755) // 创建文件夹：/sys/fs/cgroup/pids/wlm
        // 写文件：/sys/fs/cgroup/wlm/pids.max，限制最大进程数为 20
        must(os.WriteFile(filepath.Join(cgroups, "wlm/pids.max"), []byte("20"), 0700))
        // 写文件：/sys/fs/cgroup/wlm/cgroup.procs，将当前进程的 pid 写入
        must(os.WriteFile(filepath.Join(cgroups, "wlm/cgroup.procs"), []byte(strconv.Itoa(os.Getpid())), 0700))
}

/* 构建命令 */
func buildCmd() *exec.Cmd {
        cmd := exec.Command(os.Args[1], os.Args[2:]...) // 示例：/bin/bash ，本例中 os.Args[2:] 为空
        cmd.Stdin = os.Stdin                            // 标准输入
        cmd.Stdout = os.Stdout                          // 标准输出
        cmd.Stderr = os.Stderr                          // 标准错误
        return cmd
}

/* 准备容器 */
func prepareContainer() {
        must(syscall.Sethostname([]byte("container"))) // 设置容器的主机名
        must(syscall.Chroot("/opt/rootfs"))            // 修改当前进程的根目录，这里的参数理论上不应该写死
        must(os.Chdir("/"))                            // 修改当前进程的工作目录
        // 挂载 proc 文件系统，挂载点为/proc，容器需要使用 proc 文件系统来获取进程相关信息；第 1 个参数不重要
        must(syscall.Mount("proc", "proc", "proc", 0, ""))
        // 挂载 tmpfs 文件系统，挂载点为/mytemp，容器需要使用 tmpfs 文件系统来创建临时文件
        must(syscall.Mount("thing", "mytemp", "tmpfs", 0, ""))
}

/* 卸载 */
func unmount() {
        must(syscall.Unmount("proc", 0))
        must(syscall.Unmount("mytemp", 0))
}

/* 要求操作必须成功，否则抛出异常（本例中会导致进程终止） */
func must(err error) {
        if err != nil {
                panic(err)
        }
}
```

运行 mydocker.go 创建容器：

```
[root@master1 ~]# go run mydocker.go /bin/sh
Running [/bin/sh]
/ # export PATH=$PATH:/bin
/ # ls -l
total 16
drwxr-xr-x    2 root     root          4096 Jun 18 15:37 bin
drwxr-xr-x    2 root     root             6 Jun 18 15:37 dev
drwxr-xr-x   19 root     root          4096 Jun 18 15:37 etc
drwxr-xr-x    2 root     root             6 Jun 18 15:37 home
drwxr-xr-x    7 root     root          4096 Jun 18 15:37 lib
drwxr-xr-x    5 root     root            44 Jun 18 15:37 media
drwxr-xr-x    2 root     root             6 Jun 18 15:37 mnt
drwxrwxrwt    2 root     root            40 Jul 15 01:56 mytemp
drwxr-xr-x    2 root     root             6 Jun 18 15:37 opt
dr-xr-xr-x  296 root     root             0 Jul 15 01:56 proc
drwx------    2 root     root            26 Jul 15 01:46 root
drwxr-xr-x    2 root     root             6 Jun 18 15:37 run
drwxr-xr-x    2 root     root          4096 Jun 18 15:37 sbin
drwxr-xr-x    2 root     root             6 Jun 18 15:37 srv
drwxr-xr-x    2 root     root             6 Jun 18 15:37 sys
drwxrwxrwt    2 root     root             6 Jun 18 15:37 tmp
drwxr-xr-x    7 root     root            66 Jun 18 15:37 usr
drwxr-xr-x   12 root     root           137 Jun 18 15:37 var
/ # exit
[root@master1 ~]#
```

第10章 数字证书



在现实生活当中，如何证明你就是你？是的，没错，请出示你的身份证，身份证上的照片像你，尽管现在的你看起来更老:)，身份证上的名字与开户的名字相符，没错你就是你！

你有没有想过一个问题：为什么人们就相信那张名片大小的卡片（号称身份证）上的相片和名字就是真的？而你自己制作的卡片（上面贴上相片，写上姓名）他们就不相信？答案是：身份证是国家权威机关制作的，并且难以作假，如果存疑，对方可以咨询公安机关以验证身份证的真伪。人们都信赖的部门就称为权威机构，在人们日常交往中，政府往往就是充当第三方的权威机构。

身份证和可信赖的第三方权威部门，构成了现实生活当中证明“我就是我”的有效手段之一，身份证和权威部门缺一不可。

在现实生活当中，还有一个领域，就是我怎么证明这份材料（比如合同）是你签发的？对，上面有你的亲笔签名和公章，因为我熟悉你的笔迹，或者我有你的签名存档，二者比对一下就知道了。

在网络上，又是怎么证明“你就是你”呢？早期就是用户名和密码，比如你登录邮箱系统，只要输入的用户名和密码对了，那就证明你就是你了，“请进去阅读你的邮件吧！”。后来为了增强安全性，额外增加了二次验证方法，比如发手机短信验证码或者发邮件验证码，这时要正确输入用户名、密码和验证码才能证明你就是你。

在网络上，又是如何实现签名的？如何达到更高安全级别（比如军事级别）的要求？在量子计算机诞生之前，普遍的做法是：采用公/私钥、数字签名和数字证书。

下面首先从公/私钥讲起……

https://www.v

§10.1 公/私钥

对于一些重要的文档资料，我一般会加密保存在自己电脑上。你瞧，亨特·拜登就不知道加密保存资料，结果被电脑修理店老板发现并举报给 FBI，你说他傻不傻啊，嗯？比如图 10-1 就是加密保存 word 文档的例子。

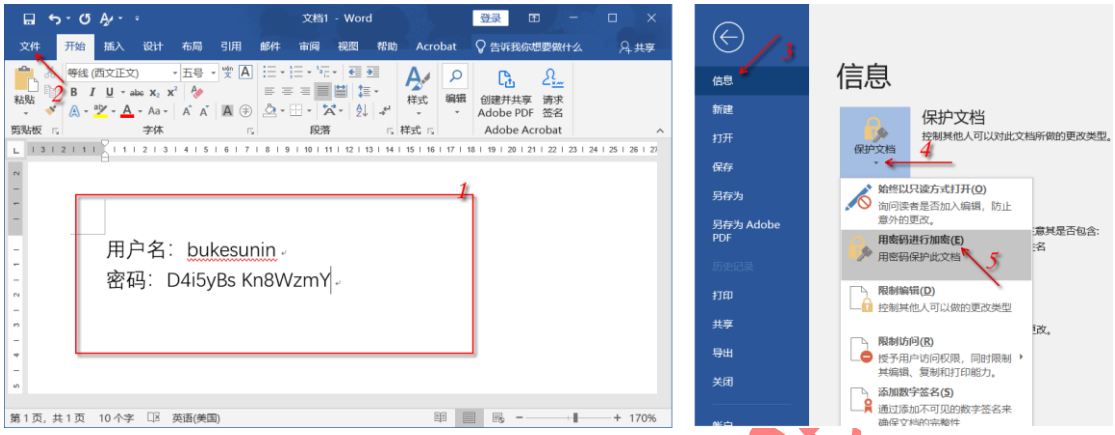


图 10-1 word 加密保存

图 10-2 把重要的图片加密压缩保存。

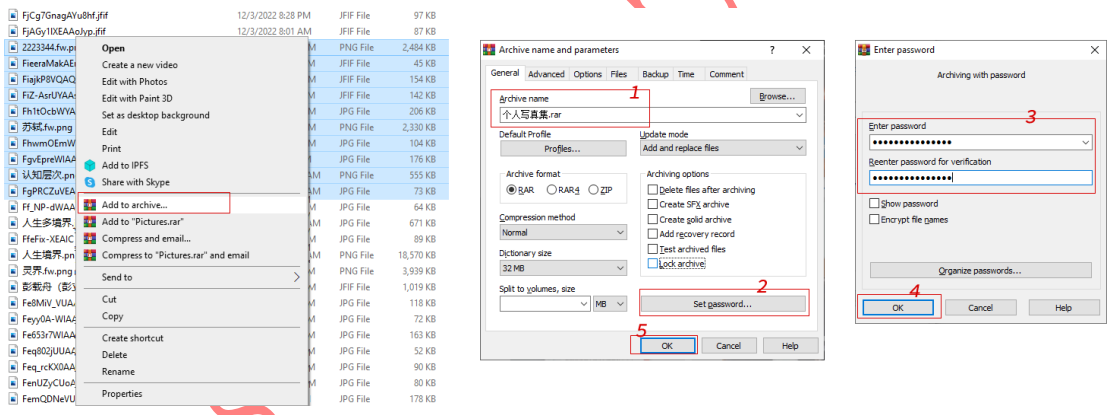


图 10-2 加密压缩保存

这两个例子都是采用对称密码——加密和解密的密码相同，比如我加密保存 word 文档的密码是 a1234567，那么下次打开这个 word 文档时我必须输入 a1234567，输错了就打不开。另外我如果把一份加密文件发给另外一个人，我必须告诉他密码才行，否则对方收到文件后打不开。一旦密码要通过媒介传输给对方，就存在泄密的可能性，当然这在民间日常的经济活动中，这种泄密带来的风险往往被人们忽视。但是在战争中，情报的加密密码如果还通过公开媒介传播，那就等于是找死。

有没有一种方法？我既可以加密文件，并把加密后的文件安全传送给对方，同时密码只有我一个人知道，换句话说，就是密码不用告诉对方。还真的存在这种方法，那就是大名鼎鼎的非对称密钥——加密和解密采用不同的密码！我用一个密码加密，你用另一个密码解密，我加密用到的密码称为公钥，你解密用到的密码称为私钥。公钥和私钥是一对的，简称为“公/私钥”对，公钥是从私钥中推导出来的，但是无法从公钥推导出私钥。对外公开公钥，私钥锁在我的保险柜里，反正打死我也不交出保险柜的钥匙！

以后别人如果要给我发加密的重要文件，他就用我事先公开的公钥加密，然后把加密后的文件发给我，最后我用私钥解密文件，就可以看到明文了。由于其他任何人都不知道私钥，所以没有人能看到明文，就算他获取到了加了密的文

件也没用。

也许有人会说：我就猜测私钥，先输入 123456，不行的话再输入 1234567，还不行再输入 abcdef，……总有猜中的时候。那我告诉你，我的私钥包含数字、大小写字母、标点符号，总共 64 位长度，就算你猜到子子孙孙十八代，也不可能成功！

在计算机处理公私钥的实际过程中，密钥长度一般称是多少位的，是指私钥或公钥是由多少个二进制组成的，比如比特币的私钥长度是 256 位二进制，公钥长度是 512 个二进制的。当然二进制位数越多，就越安全。256 位二进制长度的比特币私钥，就算你用现在最快的计算机暴力破解，也需要花费上百年的时间。暴力破解的原理很简单，就是猜一个 256 位二进制的数字，然后利用公开的算法推导公钥，最后比较两个公钥（另一个是比特币公开的公钥），如果相等就猜对了，否则继续猜下一个。

类似如图 10-3 所示为一个 256 位的私钥。



图 10-3 由二进制组成的私钥

256 位二进制数不好记，于是可以换算为 64 位的十六进制数（即 Hex 编码——由数字和字母 a,b,c,d,e,f 组成），类似下面的：

```
CF4D1030FECE1071B73246793D06D3D08EAD385481EE28CBF7B22859772798D1
```

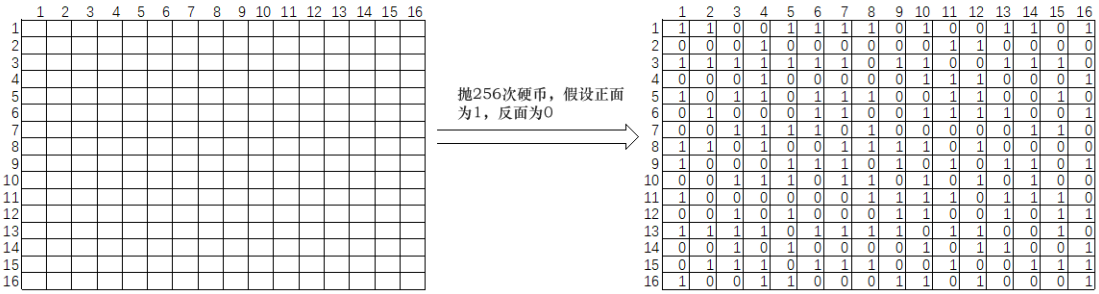
尼玛，这还不好记啊，那进一步转换为 Base64 编码的 44 个字符吧，类似下面的：

```
z00QMP7OEHG3MkZ5PQbT0I6tOFSB7ijL97IoWXcnmNE=
```

最后一个字符一定是“=”。我靠，你玩我啊，这怎么记？！那算了，我只能帮你到这里了。我告诉你，256 位二进制、64 位的十六进制数和 44 个字符的 Base64 编码，你只要记住任何一个就行，因为它们可以互相转化。

就拿一个 256 位的二进制数私钥来说，从最小的数（256 位全是 0）到最大的数（256 位全是 1），数的个数一共有 2^{256} 个（约等于 $1.15*10^{77}$，差不多等于宇宙中原子的个数）。所以随机产生的一个私钥就相当于随机在宇宙中挑选一个原子，这样两个人随机产生的私钥，它们相同的概率完全可以忽略不计。因此一个好的私钥就体现在“随机”二字上——挑选时越随机越好。在我们实际的生活当中，很多看似随机的事件却是必然的，比如你抛一枚硬币，其实是可以根据硬币离开手时的状态（速度、自转等）、风速等因素算出最终的正反面。目前为止，人类发现的真正随机的事件是原子中的电子接下来出现的位置，这才是真随机，而其他的都称为伪随机。利用现有的技术手段在有限的时间内（比如 100 年）几乎不可能重复的事件，我们就可以用来产生需要的私钥。

手工产生一个 256 位私钥最简单的方法就是抛 256 次硬币（假如规定正面为 1，反面为 0），从而产生 256 位二进制数的私钥，见图 10-4 所示。



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	0	0	1	1	1	1	0	1	0	0	1	1	0	1
2	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0
3	1	1	1	1	1	1	1	0	1	1	0	0	1	1	1	0
4	0	0	0	1	0	0	0	0	0	1	1	1	0	0	0	1
5	1	0	1	1	0	1	1	1	0	0	1	1	0	0	1	0
6	0	1	0	0	0	1	1	0	0	1	1	1	1	0	0	1
7	0	0	1	1	1	1	0	1	0	0	0	0	0	1	1	0
8	1	1	0	1	0	0	1	1	1	1	0	1	0	0	0	0
9	1	0	0	0	1	1	1	0	1	0	1	0	1	1	0	1
10	0	0	1	1	1	0	1	1	0	1	0	1	0	1	0	0
11	1	0	0	0	0	0	0	1	1	1	1	0	1	1	1	0
12	0	0	1	0	1	0	0	0	1	1	0	0	1	0	1	1
13	1	1	1	1	0	1	1	1	1	0	1	1	0	0	1	0
14	0	0	1	0	1	0	0	0	0	1	0	1	1	0	0	1
15	0	1	1	1	0	1	1	1	0	0	1	0	0	1	1	1
16	1	0	0	1	1	0	0	0	1	1	0	1	0	0	0	1

图 10-4 抛硬币产生私钥的过程

然后依次把各行中的二进制数拼接成一个 256 位的二进制数，就得到了一个很好的私钥，参见图 10-5 所示。

https://www.verypen.org

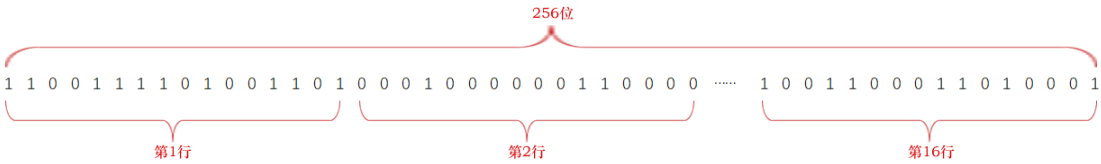


图 10-5 抛硬币产生的私钥

在本例子中，最后得到如下所示的 256 位的二进制私钥：

```
1100111101001101000100000011000011111110110011100001000001110001101101110011001001000110011111001001111010000011011010011110100001000111010101101001110000101010010000001111011100010100011001011111110111011001000101000010110010111011100100111100110001101010000
```

256 位二进制数不好记，使用也不方便，一般要转换为其他编码形式，比如常见的 HEX（十六进制）或 Base64 等。对于 HEX，每 4 位二进制对应一位十六进制数，可参考下表作转换。

二进制数	十进制数
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

因此上面抛硬币产生的随机数，通过纯手工方法转换，对应的 HEX 编码如下（一共 64 位）：

```
CF4D1030FECE1071B73246793D06D3D08EAD385481EE28CBF7B22859772798D1
```

参照上面的表格，换算的过程如图 10-6 所示。

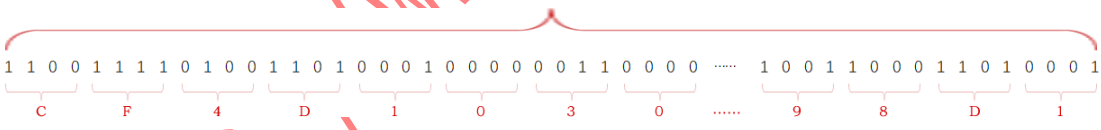


图 10-6 用十六进制表示的私钥

通过手工把二进制换算成十六进制数，很容易出错（不过出错关系也不大，反正你最终记住 HEX 编码的私钥就可以了）。

64 位的十六进制编码的私钥还是有点长，如果转换为 Base64 编码，长度会更短一些，Base64 编码机制是：以 24 位二进制数为一组，其中每 6 位二进制转换为 1 个字符，最后一组只有 16 位二进制（结果补 1 个“=”）。Base64 码表如下：

https://www.yopen.org

二进制	十进制	字符	二进制	十进制	字符	二进制	十进制	字符	二进制	十进制	字符
000000	0	A	010000	16	Q	100000	32	g	110000	48	w
000001	1	B	010001	17	R	100001	33	h	110001	49	x
000010	2	C	010010	18	S	100010	34	i	110010	50	y
000011	3	D	010011	19	T	100011	35	j	110011	51	z
000100	4	E	010100	20	U	100100	36	k	110100	52	0
000101	5	F	010101	21	V	100101	37	l	110101	53	1
000110	6	G	010110	22	W	100110	38	m	110110	54	2
000111	7	H	010111	23	X	100111	39	n	110111	55	3
001000	8	I	011000	24	Y	101000	40	o	111000	56	4
001001	9	J	011001	25	Z	101001	41	p	111001	57	5
001010	10	K	011010	26	a	101010	42	q	111010	58	6
001011	11	L	011011	27	b	101011	43	r	111011	59	7
001100	12	M	011100	28	c	101100	44	s	111100	60	8
001101	13	N	011101	29	d	101101	45	t	111101	61	9
001110	14	O	011110	30	e	101110	46	u	111110	62	+
001111	15	P	011111	31	f	101111	47	v	111111	63	/

参照此表，我们上面抛硬币产生的私钥编码成 base64 的过程如图 10-7 所示。

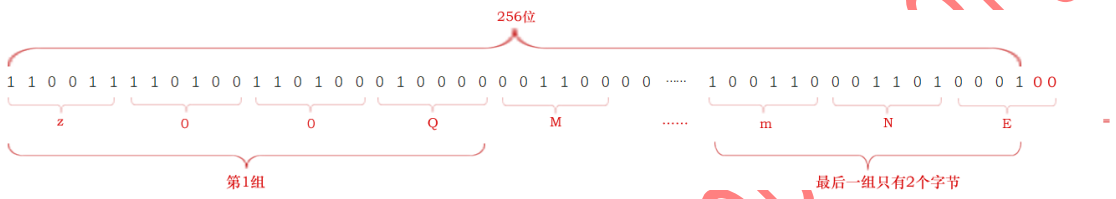


图 10-7 私钥编码成 base64 格式

最终的 Base64 编码为：

```
z00QMP7OEHG3MkZ5PQbT0I6tOFSB7ijL97IoWXcnmNE=
```

一共 44 个字符，相比于 HEX 编码少了 20 个。

从 256 位的私钥提取 512 位的公钥，存在多种方法，方法不同，提取出来的公钥也不同。提取公钥没法手工完成，需要用到专门的计算机算法程序，而这些算法是公开的。

一般很少人会采用纯手工方法制作私钥，大部分采用计算机指令完成。下面我们在 Linux 系统中利用命令 openssl 来制作自己的公/私钥。

1. 产生公/私钥：

```
产生 1024 位私钥，并保存在 private.key 文件中：
[root@master1 ~]# openssl genrsa -out private.key 1024

从私钥导出公钥，并保存在 public.key 中：
[root@master1 ~]# openssl rsa -in private.key -pubout -out public.key

[root@master1 ~]# ll *.key
-rw------- 1 root root 916 Aug 10 14:25 private.key                    #私钥文件
-rw-r--r-- 1 root root 272 Aug 10 14:25 public.key                     #公钥文件
```

也可以采用命令“cat private.key | base64 --wrap=0”把私钥编码成 base64。

好了，现在把公钥文件 public.key 分发给你的亲戚朋友，下次用得上。当很多人（比如客户）拥有你的公钥，并且日常的经济往来活动经常使用公钥后，妥善保管你的私钥就显得特别重要，你想想，万一私钥泄露了就麻烦了，你得赶快重做一对公/私钥，并且尽快广而告知新的公钥，这期间可能存在泄密风险——可能还有人继续用旧的公钥加密文件并发给你，但不幸的是被掌握你的旧私钥的人截获了密文。有没有方法对私钥进行加密？用到私钥的时候临时解密，这样就算别人获取到我的私钥，他没有密码，还是没法使用私钥。还真有加密的私钥，下面我们亲自做一对。

2.产生加了密的私钥：

```
产生 2048 位的私钥，同时对它进行加密，加密后的私钥保存在 private.pem 文件中：
[root@master1 ~]# openssl genrsa -aes256 -passout stdin -out private.pem 2048
A1b2C3                                                                 #加密私钥的密码
```

https://www.veryor.org

```
导出公钥，并把公钥保存在 public.pem 文件中：
[root@master1 ~]# openssl rsa -in private.pem -pubout -out public.pem
Enter pass phrase for private.pem:                                    #输入密码 A1b2C3
writing RSA key

[root@master1 ~]# ll *.pem
-rw------- 1 root root 1874 Aug 10 15:28 private.pem                  #加了密的私钥
-rw-r--r-- 1 root root  451 Aug 10 15:28 public.pem                   #公钥
```

公钥没有加密，可以公开。私钥加了密，以后每次使用私钥（比如签名）时必须输入密码才行，所以这时就算别人获得了我的私钥文件 private.pem，他也是没法使用私钥的，因为他不知道密码。

下面我们使用刚刚产生的公/私钥来加密和解密文档，记住：用公钥加密，私钥解密。公加私解→“公家四姐”，好记吧？！。

实战一、利用公私钥来加/解密文档。

当要通过公共媒介（比如互联网）传递重要资料时，最佳的做法是用公钥加密文档，对方收到后用私钥解密即可。

```
[root@master1 ~]# vim file.txt                                        #需转递的军事情报
今晚九点，五团、八团、九团联合同时向 396 高地发起总攻！切记!!

用公钥加密文件 file.txt，密文保存在 file.txt.encrypt 文件中：
[root@master1 ~]# openssl pkeyutl -encrypt -pubin -inkey public.key -in  file.txt -out file.txt.encrypt

[root@master1 ~]# ll file.txt*
-rw-r--r-- 1 root root  96 Aug 10 14:26 file.txt                      #明文文件，最好删除
-rw-r--r-- 1 root root 128 Aug 10 14:26 file.txt.encrypt              #密文文件

对方收到 file.txt.encrypt 文件后用私钥解密：
[root@master1 ~]# openssl pkeyutl -decrypt -inkey private.key -in file.txt.encrypt
今晚九点，五团、八团、九团联合同时向 396 高地发起总攻！切记!!
```

实战二、利用公私钥登录到另一台 linux 电脑。

我们租用的云上主机，一般都是安装 Linux 操作系统，平时运维都是远程登录过去，现在采用 ssh 协议的登录工具几乎是标配，而且 Linux 默认已经安装了 openssh-server 服务，不管是 windows 还是 linux 都有 ssh 这个客户端登录命令。ssh 协议支持两种登录方法，一是“用户名/密码”方式，二是“公/私钥”方式，显然“公/私钥”方法要安全得多——先上传公钥，此后用私钥登录。最佳的实践是：云上主机不允许 root 远程登录，改掉默认的登录端口 22（比如改成 7208），创建一个允许使用 sudo 命令的普通用户，在客户机上创建公私钥，把公钥上传到云上主机，最后修改 sshd 的配置，禁止用户密码登录。下面不妨把 master1 作为云上主机，worker1 作为登录客户机，具体操作如下（请根据命令行上的主机名判断在哪台电脑上操作）：

```
[root@master1 ~]# vim /etc/ssh/sshd_config
......
Port 7208                                                             #把默认端口 22 改为 7208
......
PermitRootLogin no                                                    #不允许 root 远程登录
......

[root@master1 ~]# systemctl reload sshd                               #使新配置生效

[root@master1 ~]# ss -tnlp|grep sshd
LISTEN 0      128          0.0.0.0:7208       0.0.0.0:*    users:(("sshd",pid=794,fd=3))
LISTEN 0      128             [::]:7208          [::]:*    users:(("sshd",pid=794,fd=4))

[root@master1 ~]# useradd -g wheel alice                              #加入 wheel 组，就能使用 sudo 命令
[root@master1 ~]# passwd alice
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

采用 ssh-keygen 命令生成公私钥：4096 位、rsa 类型、私钥密码 A1b2C3、私钥文件 rsa_4096、公钥文件 rsa_4096.pub：

```
[root@worker1 ~]# ssh-keygen -q -b 4096 -C "登录到我VPS" -t rsa -N A1b2C3 -f rsa_4096
[root@worker1 ~]# ll rsa_4096*
-rw------- 1 root root 3434 Aug 12 15:44 rsa_4096                     #私钥文件
-rw-r--r-- 1 root root  741 Aug 12 15:44 rsa_4096.pub                 #公钥文件
```

```
[root@worker1 ~]# ssh-copy-id -i rsa_4096.pub -p 7208 alice@master1                    #公钥上传到 master1
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "rsa_4096.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys
alice@master1's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh -p '7208' 'alice@master1'"
and check to make sure that only the key(s) you wanted were added.

[root@master1 ~]# su - alice
Last login: Sat Aug 12 16:26:38 CST 2023 on pts/0

[alice@master1 ~]$ more .ssh/authorized_keys                          #检查是否只有一份公钥
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAACAQC9ZiEP3ne8YV+g0D06WuP0TpjFNwlTueZAMmZnsZpNjKxegk6jZLxx+2pHK6bgZJ0hwFk7lBvqXloZ0bF4bVOYSOowwLK64
DEB3HgdpXM/xr7+Vn81ylgckV0ar7veq9dOXSaaAqg33yEvmi6VpcwrV9yKFngey/tQZ8y1f1zubRUuL6ebPh+IA4OATlce3vv
ecVvNR45FmfIYdnd+j70FjBr+yPCyPwiYaJN9QyPLyX2cJrtJCVH91/7uXs+cVzTX461Q1mOA+zHz0fYcKjMAXYSgotSuHUj4+/SMIZRyB7y9qui+tRyyz3zx/erP
2lImRTk/lQz2EE3cys/7tvhUP/ANDOUpQ/Jf2OAgcEF0Ge1sSuQyXF0zvU4IUNCkO8tIfIHpoIR2Ibp/CP7Yul0Il0U1tHijCYXW27/aAO
2k1V/Q3NAppltW9OotrJzw9ZGe8ogGizIN5RHioQQWGPbP5wPVUR76ri4X5I6CF/IF7j4XH92FWQan6LqLcRo8jHHPrH5hRAP9koRPP2Uol5VZ8DK8AUEjp8mqkx1
7ttRvi2yFCFPJXLohFUqc6s0K8utC7588tH19f3xq/T43FnnjHUC1x1xRUjI8/ONHS6pjbo6FbQoLNJWTgvA4mtqr+kFSttzFd8+Nh7aRf
PKbiiisfn75kC1+8sCDXl605Y1liQ== 登录到我 VPS
[alice@master1 ~]$ exit
logout
[root@master1 ~]# vim /etc/ssh/sshd_config
......
PasswordAuthentication no                                                          #禁止密码登录
......

[root@master1 ~]# systemctl reload sshd
```

下面测试公钥登录：

```
[root@worker1 ~]# ssh -i rsa_4096 -p 7208 alice@master1
Enter passphrase for key 'rsa_4096':                                  #输入私钥的保护密码 A1b2C3
Last login: Sat Aug 12 16:45:13 2023 from 10.10.10.31
[alice@master1 ~]$
```

使用公钥登录成功了。因为私钥加了密，加密密码是 A1b2C3，所以登录时要输入这个密码。参数-i 指明私钥文件名，如果不在当前目录，请带路径，如-i /opt/pki/private.ssh。参数-p 指明登录端口，默认是 22，如果改了其他端口号，必要用参数-p 指定。

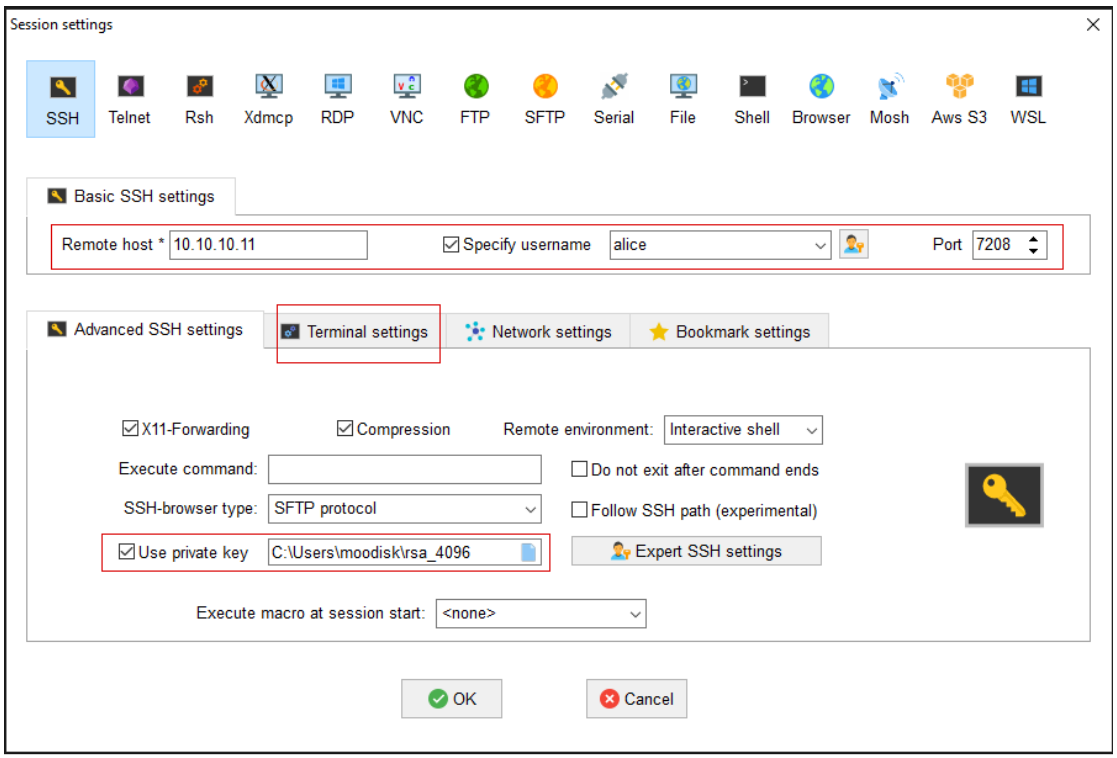
再看看能否用密码登录：

```
[root@worker1 ~]# ssh  -p 7208 alice@master1
alice@master1: Permission denied (publickey,gssapi-keyex,gssapi-with-mic).
```

不能用密码登录。

把私钥 rsa_4096 拷贝到 windows，然后直接可用 windows 的 ssh 命令登录到 master1。mobaXterm 工具使用私钥登录参考图 10-8。

https://www.very



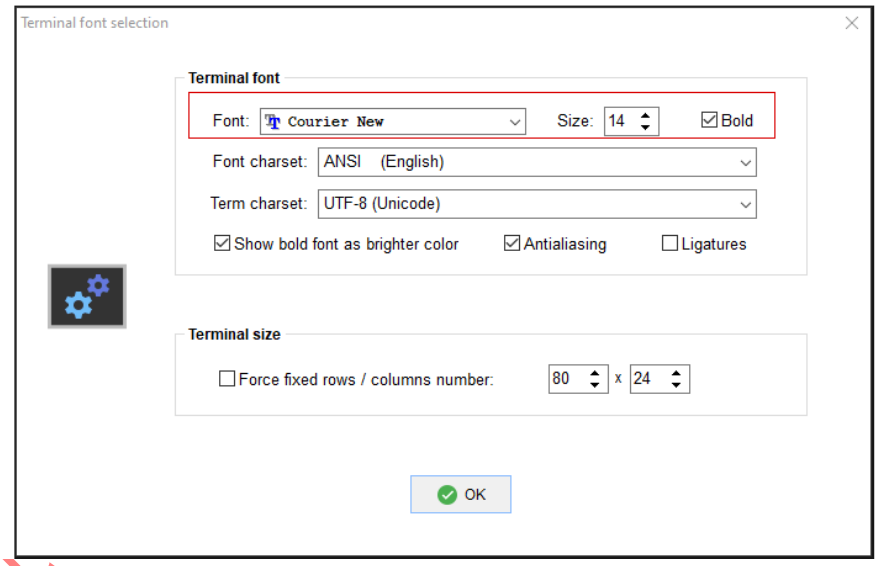


图 10-8 mobaXterm 使用私钥登录

§10.2 数字签名

数字签名的目的是杜绝抵赖，就像现实生活当中的手写签名，明明合同上面是你的亲笔签名，你还怎么抵赖。在网络生活当中，数字签名更没法抵赖。只能用私钥签名，而私钥只有你一个人知道，你还怎么抵赖。什么？你的私钥被别人盗用了？那也是你的问题，谁叫你保管不小心。一旦发现私钥泄露了，赶快做一个新的，同时把新的公钥公布于众。

对一份文件用私钥签名，用公钥验证，所以数字签名的前提还是你拥有一对公/私钥，并把公钥公布与众。签名和验证的示意图请参考图 10-7。

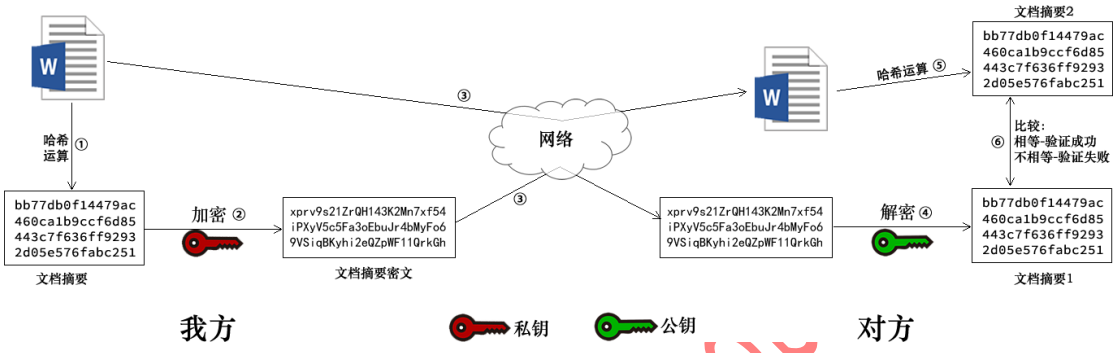


图 10-7 签名和验证

签名和验证大致的过程是这样的：

①给待签名的文档做哈希运算，产生文档摘要；
②用私钥对文档摘要加密，产生摘要密文；
③把文档和摘要密文发给对方；
④对方用公钥解密文档摘要，得到文档摘要 1；
⑤对方对文档做哈希运算，算出文档摘要 2；
⑥比较文档摘要 1 和文档摘要 2，如果相等表示签名验证成功，如果不相等则验证不成功。验证不成功可能的原因有二：一是私钥和公钥不是一对，二是文档被改动。

哈希运算就是不管处理什么电子资料（如文本文件、图片、视频、word 文档、程序等），计算的结果都是固定长度的一串字符，比如都是 32 个字符。哈希运算有以下三个特征：

1、相同的输入一定会得到相同的输出；2、不同的输入得到相同的输出是极小概率事件；3、即使对输入做极细微的改变，输出却完全不同。

第 1 个特征说明不管何时何地对没篡改的内容做哈希运算，永远会得到相同的结果，即验证是没有时效性和区域性限制的。第 2 个特征说明你最好打消这种侥幸心理：我修改了原内容，但碰巧得到相同的哈希值。输入不同而输出相同的现象称为哈希碰撞，当规定输出串足够长时（比如加密币是 32 个字符），碰撞的概率小到可以忽略不计，尽管理论上碰撞是一定存在的。第 3 个特征表明：即使你修改一个 word 文档（假如是一本上千页的书稿）中的一个标点符号，哈希运算的结果会完全不同，输出串中的每一个字符都变了，也就是说不可能从哈希值中寻找规律，从而反推算出原文是什么。

所以数字签名保证了：1.我签了名的文档，任何人不能改动，否则对方验证签名时肯定通不过；2.我不能抵赖，因为私钥只有我一个人拥有，如果用其他私钥签名，那么对方验证时肯定通不过。

下面我们实战一下数字签名。先准备一个 word 文档，比如“创建 ssh 隧道.docx”上传到 Linux 系统中。淡然任何类型的文档都可以签名，包括文本文件、图片文件、音频、视频、excel、ppt 等都行。

https://www.veryc

1. 用私钥对文档签名：

```
[root@master1 ~]# openssl dgst -sha256 -sign private.pem -out 创建ssh隧道.docx.sha256 创建ssh隧道.docx
Enter pass phrase for private.pem:

[root@master1 ~]# ll 创建ssh隧道.docx*
-rw-r--r-- 1 root root 1298015 Aug 11 07:38 创建ssh隧道.docx
-rw-r--r-- 1 root root     256 Aug 11 08:12 创建ssh隧道.docx.sha256
```

如果文档太大（超过 1K），需要先计算摘要，然后对摘要签名。

2. 然后把“创建 ssh 隧道.docx”和“创建 ssh 隧道.docx.sha256”两个文件发给对方，对方收到后用公钥验证签名：

```
[root@master1 ~]# openssl dgst -sha256 -verify public.pem -signature 创建ssh隧道.docx.sha256 创建ssh隧道.docx
Verified OK
```

下面我们修改“创建 ssh 隧道.docx”，再来验证签名看看是否成功：

```
[root@master1 ~]# echo 1 >> 创建ssh隧道.docx
[root@master1 ~]# openssl dgst -sha256 -verify public.pem -signature 创建ssh隧道.docx.sha256 创建ssh隧道.docx
Verification failure
80AB5B73307F0000:error:02000068:rsa routines:ossl_rsa_verify:bad signature:crypto/rsa/rsa_sign.c:430:
80AB5B73307F0000:error:1C880004:Provider routines:rsa_verify:RSA
lib:providers/implementations/signature/rsa_sig.c:817:
```

验证失败了。

下面我们再来做个实验：从 www.kernel.org 网站下载 Linux 内核，并验证签名。我们看看网站上的文件，参考图 10-11 所示。

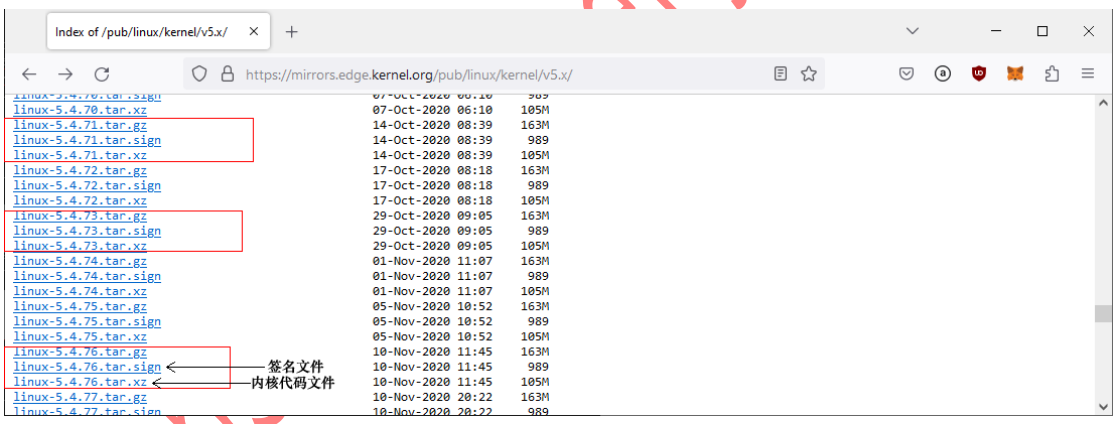


图 10-11 内核代码文件和对应的签名文件

1. 首先下载内核代码文件和签名文件：

```
[root@master1 ~]# wget https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-5.4.76.tar.xz
[root@master1 ~]# wget https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/linux-5.4.76.tar.sign
[root@master1 ~]# unxz linux-5.4.76.tar.xz                                                    #解压缩

[root@master1 ~]# ll linux-5.4.76.tar*
-rw-r--r-- 1 root root 939448320 Nov 10  2020 linux-5.4.76.tar                          #被签名文件
-rw-r--r-- 1 root root       989 Nov 10  2020 linux-5.4.76.tar.sign                     #签名文件
```

2. 验证签名：

```
[root@master1 ~]# gpg2 --verify linux-5.4.76.tar.sign
gpg: assuming signed data in 'linux-5.4.76.tar'
gpg: Signature made Tue 10 Nov 2020 07:38:48 PM CST
gpg:                using RSA key 647F28654894E3BD457199BE38DBBDC86092693E
gpg: Can't check signature: No public key
```

不能验证签名，因为没有公钥。

3. 下载内核维护者 Linus Torvalds 和 Greg Kroah-Hartman 的公钥：

```
[root@master1 ~]# gpg2 --locate-keys torvalds@kernel.org gregkh@kernel.org
gpg: /root/.gnupg/trustdb.gpg: trustdb created
gpg: key 38DBBDC86092693E: public key "Greg Kroah-Hartman <gregkh@kernel.org>" imported
gpg: Total number processed: 1
gpg:               imported: 1
gpg: key 79BE3E4300411886: public key "Linus Torvalds <torvalds@kernel.org>" imported
gpg: Total number processed: 1
gpg:               imported: 1
pub   rsa4096 2011-09-23 [SC]
      647F28654894E3BD457199BE38DBBDC86092693E
uid           [ unknown] Greg Kroah-Hartman <gregkh@kernel.org>
sub   rsa4096 2011-09-23 [E]

pub   rsa2048 2011-09-20 [SC]
      ABAF11C65A2970B130ABE3C479BE3E4300411886
uid           [ unknown] Linus Torvalds <torvalds@kernel.org>
sub   rsa2048 2011-09-20 [E]

[root@master1 ~]# gpg2 --list-keys                                          #列出存在的公钥
/root/.gnupg/pubring.kbx
------------------------
pub   rsa4096 2011-09-23 [SC]
      647F28654894E3BD457199BE38DBBDC86092693E
uid           [ unknown] Greg Kroah-Hartman <gregkh@kernel.org>
sub   rsa4096 2011-09-23 [E]

pub   rsa2048 2011-09-20 [SC]
      ABAF11C65A2970B130ABE3C479BE3E4300411886
uid           [ unknown] Linus Torvalds <torvalds@kernel.org>
sub   rsa2048 2011-09-20 [E]
```

4. 再次验证签名：

```
[root@master1 ~]# gpg2 --verify linux-5.4.76.tar.sign
gpg: assuming signed data in 'linux-5.4.76.tar'
gpg: Signature made Tue 10 Nov 2020 07:38:48 PM CST
gpg:                using RSA key 647F28654894E3BD457199BE38DBBDC86092693E
gpg: Good signature from "Greg Kroah-Hartman <gregkh@kernel.org>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 647F 2865 4894 E3BD 4571  99BE 38DB BDC8 6092 693E
```

使用 Greg Kroah-Hartman 的公钥验证，并警告说公钥没有经过权威机构认证，我们暂且信任它：

```
[root@master1 ~]# gpg2 --tofu-policy good 647F28654894E3BD457199BE38DBBDC86092693E
gpg: Changing TOFU trust policy for binding <key: 647F28654894E3BD457199BE38DBBDC86092693E, user id: Greg Kroah-
Hartman <gregkh@kernel.org>> from good to good.
```

再次验证：

```
[root@master1 ~]# gpg2 --trust-model tofu --verify linux-5.4.76.tar.sign
gpg: assuming signed data in 'linux-5.4.76.tar'
gpg: Signature made Tue 10 Nov 2020 07:38:48 PM CST
gpg:                using RSA key 647F28654894E3BD457199BE38DBBDC86092693E
gpg: Good signature from "Greg Kroah-Hartman <gregkh@kernel.org>" [full]
gpg: gregkh@kernel.org: Verified 1 signatures in the past 25 minutes.  Encrypted
     0 messages.
```

一切 ok 了。

§10.3 数字证书

前面讲过，公/私钥可用于信息的加密和解密、签名和验证签名，只要你把公钥发给对方，且对方愿意相信你并接受你的公钥。现在的问题是对方为何相信你并接受你的公钥？他是我的儿子啊，或者是我老爹啊，或者是我最好的朋友啊，是我的兄弟姐妹啊，……。但如果对方压根不认识你呢？ta 看到你发来的公钥就会惊讶地问“你谁呀？……操!”，得了，他根本不接受你的公钥。也许你会问，我和 ta 没有任何关系，就没必要把公钥发给 ta 啊。你还别急，还真存在这样的情况：你要向陌生人证明自己的身份，比如你的网站，你是多么希望全世界的所有人都来访问你的网站，对吧？但是对方凭什么相信这就是你的网站而不是钓鱼网站呢？

现在的网址一般都是以“https://”开头的，比如百度就是 https://www.baidu.com，京东 https://www.jd.com，中国银行 https://www.boc.cn，华为 https://www.huawei.com，……，就连作者自己的网站也是 https://www.veryopen.org，参见图 10-6。以 https://开头的网站都是加密传输的，所以安全性很高，而 http://开头的网站是明文传输的，现在几乎很少用了，浏览器遇到这样的网站会提示安全警告。

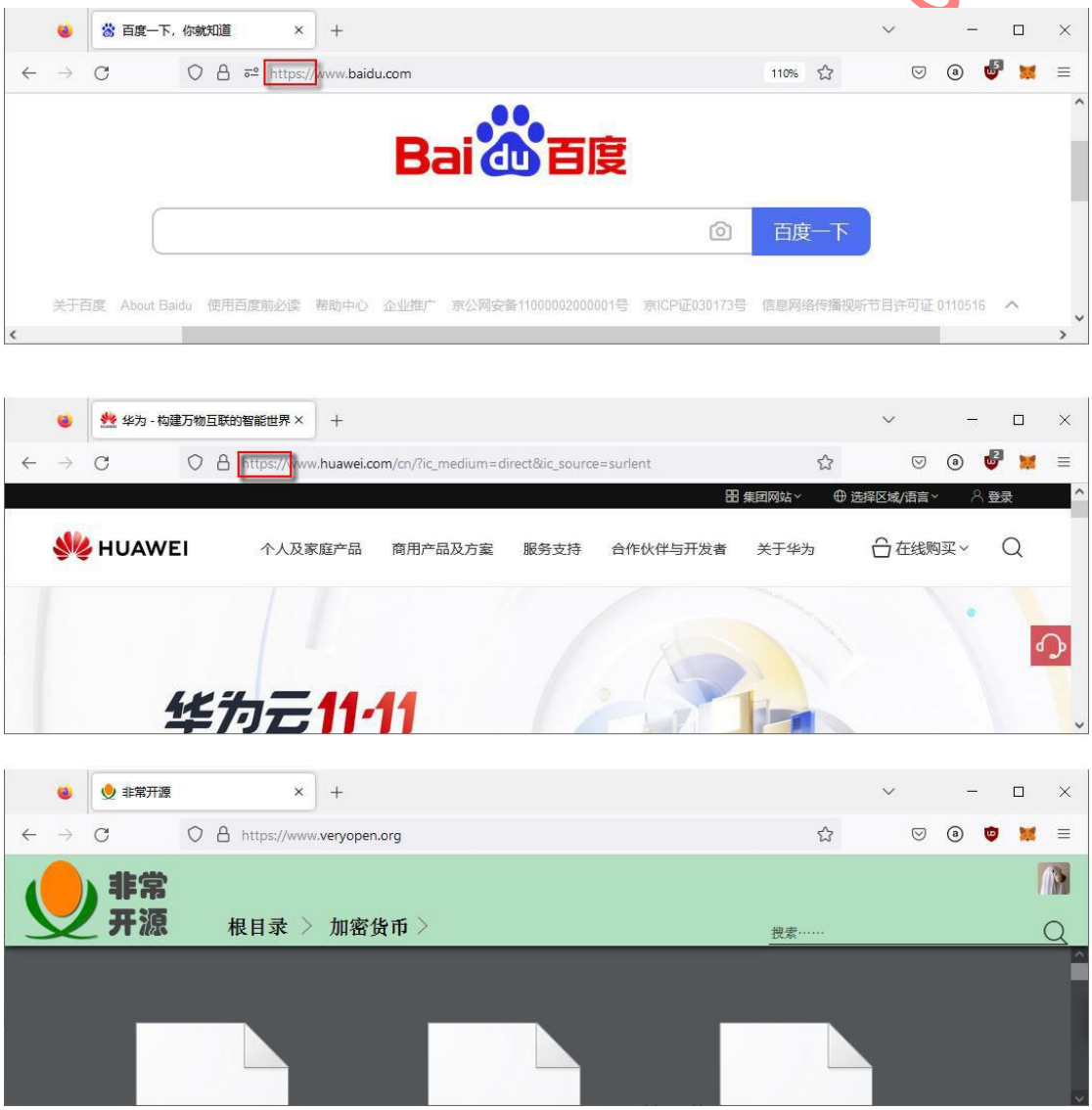


图 10-6 “https://”协议的网站

凡是以“https://”开头的网站都是向对方提供了公钥的，且对方通过权威的第三方认证机构，证明了此公钥确实是你的。这里的“对方”就是各种设备上的浏览器，比如 firefox 浏览器、chrome 浏览器、edge 浏览器等。例如你用 chrome 浏览器打开中国银行的网站，你的浏览器就接受了中国银行的网站服务器发来的公钥并通过第三方权威机

构验证此公钥确实是中国银行的，所以你才会放心地输入银行账号和密码登录到网络银行。只不过接受公钥和验证都是由浏览器自动完成的。好家伙，我只是输入网址并回车，浏览器却在背后为我做了很多事情，真乃幕后英雄耶！“偷偷的进村，打枪的不要！”

其实网站服务器发给浏览器的身份信息，除了公钥，还有网站主人的身份信息，如公司名称、网址、有效期、身份证或机构代码等，这些统称为证书内容，再加上第三方权威认证机构的签名，一起构成了数字证书，示意图参考 10-8。

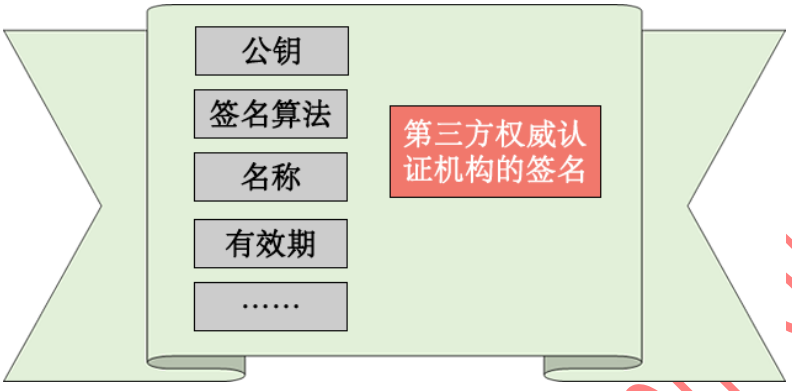
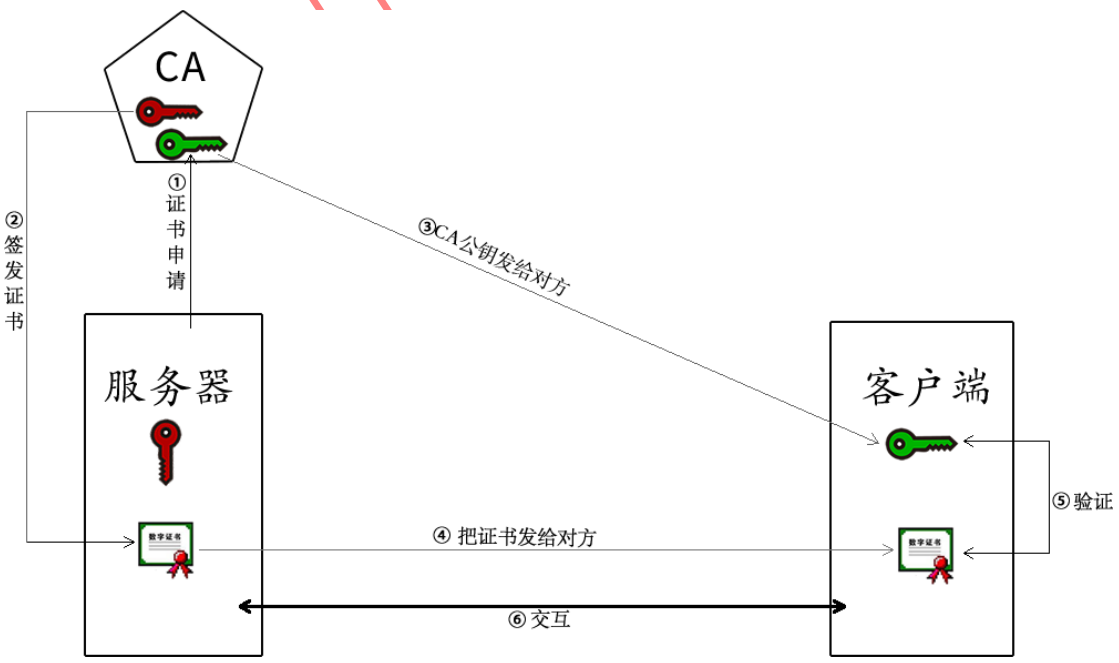


图 10-8 数字证书

第三方权威认证机构（简称 CA）就是指政府的认证部门、或者有信誉的非政府组织、或者是非盈利性的民间组织，这些认证机构同样拥有自己的公/私钥，且把公钥发给了各个浏览器厂家并且也得到了浏览器厂家的接受承认——即浏览器已经保存了认证机构的公钥。

数字证书是用来向你证明我的身份的，我和你构成了交互的双方。在网络世界中，“你和我”的关系就是“服务器和客户端”的关系——先发起通信请求的一方就是客户端，那么另一方就是服务器了。出于安全的考虑，通信双方正式交互之前必须验证对方身份，这种验证有的场合是单向验证（比如浏览器这个客户端要验证网站服务器的证书，而网站服务器不用验证浏览器），有的场合需要双向验证，即客户端要验证服务器，服务器也要验证客户端。用 CA 的公钥验证你的数字证书，你的数字证书就是用 CA 的私钥签发的，所以为了能验证你的数字证书，我必须拥有为你签发证书的那个 CA 的公钥，单向验证示意图请参考 10-9，要注意图中的钥匙图标的方向，方向不同代表不同的私钥和公钥，另外红色钥匙代表私钥，绿色钥匙代表公钥。



www.veryyoen.org

图 10-9 客户端验证服务器的身份

图 10-10 为双向验证示意图。

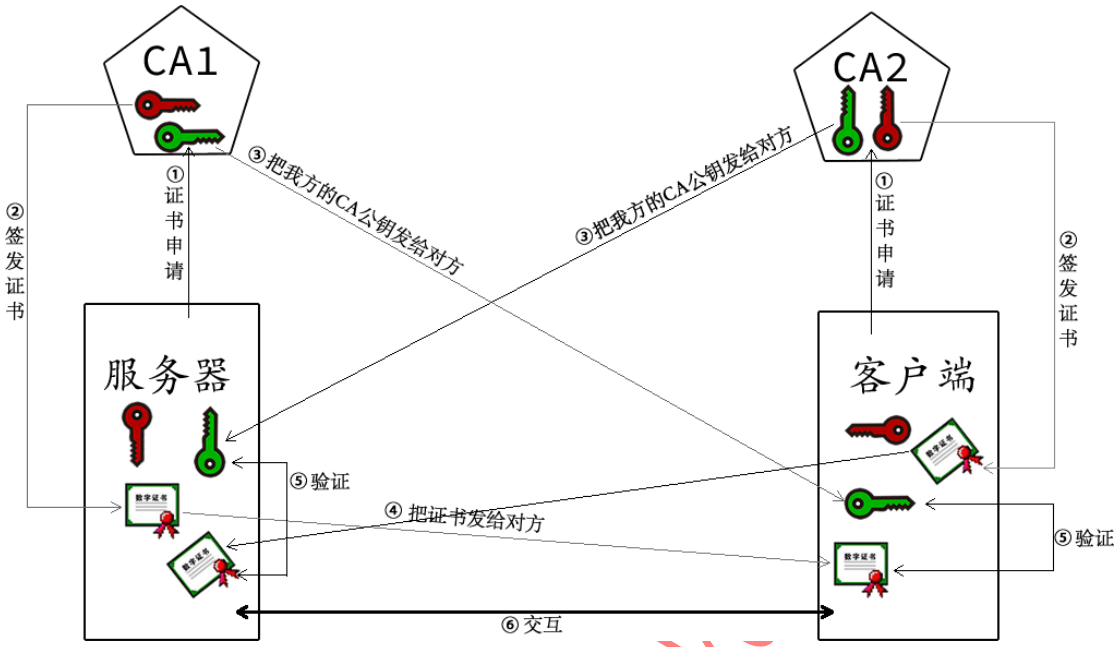


图 10-9 双向验证身份

给双方签发数字证书的权威机构（即 CA）可能相同，也可能不同，世界上这样的权威机构有很多，而且还有开源的。还有一种特殊的情况，就是服务器和客户端拥有相同的私钥和数字证书，这在企业内部的 IT 系统中常见，比如后面讲到的 etcd 集群中各个节点之间的通信就采用同一套私钥和数字证书。

下面以作者本人的网站的证书获取和认证为例讲解整个过程。

1. 我产生自己网站（即 www.veryopen.org）的公/私钥，然后产生证书申请文件（内含公钥、域名和我的身份信息）并发给 DigiCert 公司（可在 www.digicert.com 网站提交）。DigiCert 公司是一家权威的数字证书颁发机构，此机构得到了几乎所有浏览器厂家的认可。

2. DigiCert 公司收到我的证书申请文件之后，核实我的身份（采用各种线上和线下核实手段）。

3. 我支付数字证书费用（按年收费，一年 500 美元不到，有效期越长费用越高）给 DigiCert 公司。我 2007 年在华为上班时，参与过华为网站证书的申请过程，颁发机构是中国政府的某个部门，那时一年费用六千多元，比国外的贵多了。

4. DigiCert 公司产生数字证书并反馈给我。签名过程见图 10-9 所示，最后把证书内容和第三方权威机构的签名发回给我。

https://www.veryo

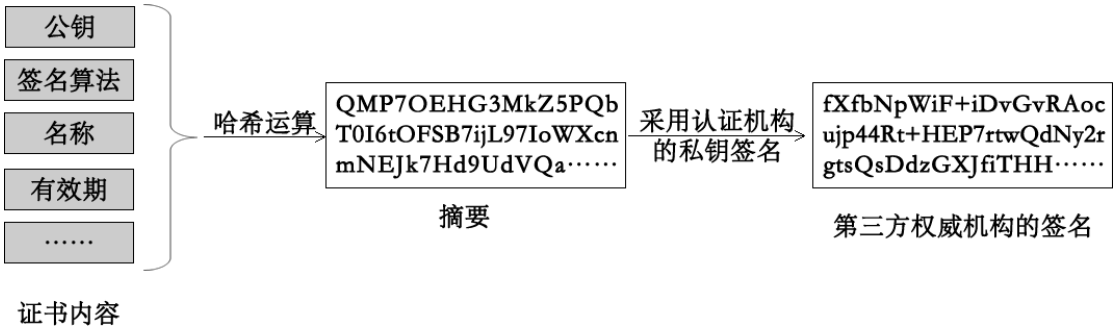


图 10-9 第三方权威认证机构的签名过程

5. 我收到数字证书后导入到网站服务器中。从此等待别人浏览我的网站，然后网站服务器就会把证书发给对方（的浏览器）。

当另一个人打开我的网站时，ta 的浏览器会收到我的网站服务器发来的数字证书，这时浏览器验证数字证书的过程是这样的：用第三方权威认证机构的公钥（比如 DigiCert 的公钥已经保存在浏览器中）验证数字证书中的签名（即第三方权威认证机构签名）——从签名算出摘要（不妨称为“摘要 1”），再对证书内容做同样的哈希运算，算出摘要（称为“摘要 2”），如果摘要 1 等于摘要 2 说明验证成功，否则就是验证失败。如果验证失败，浏览器会发出警告，并拒绝打开网站。如果验证成功，浏览器就用证书里的公钥加密一段随机产生的字符串（不妨称为“字符串 1”），然后把加密后的密文发给网站服务器，网站服务器用私钥解密后得到“字符串 1”，此后双方把“字符串 1”作为对称密码，用于加/解密此后交互的网页内容。因为对称加/解密运算效率要远远高于非对称加/解密运算。

图 10-10 是查看中国银行网站的数字证书。

https://www.veryopen.org

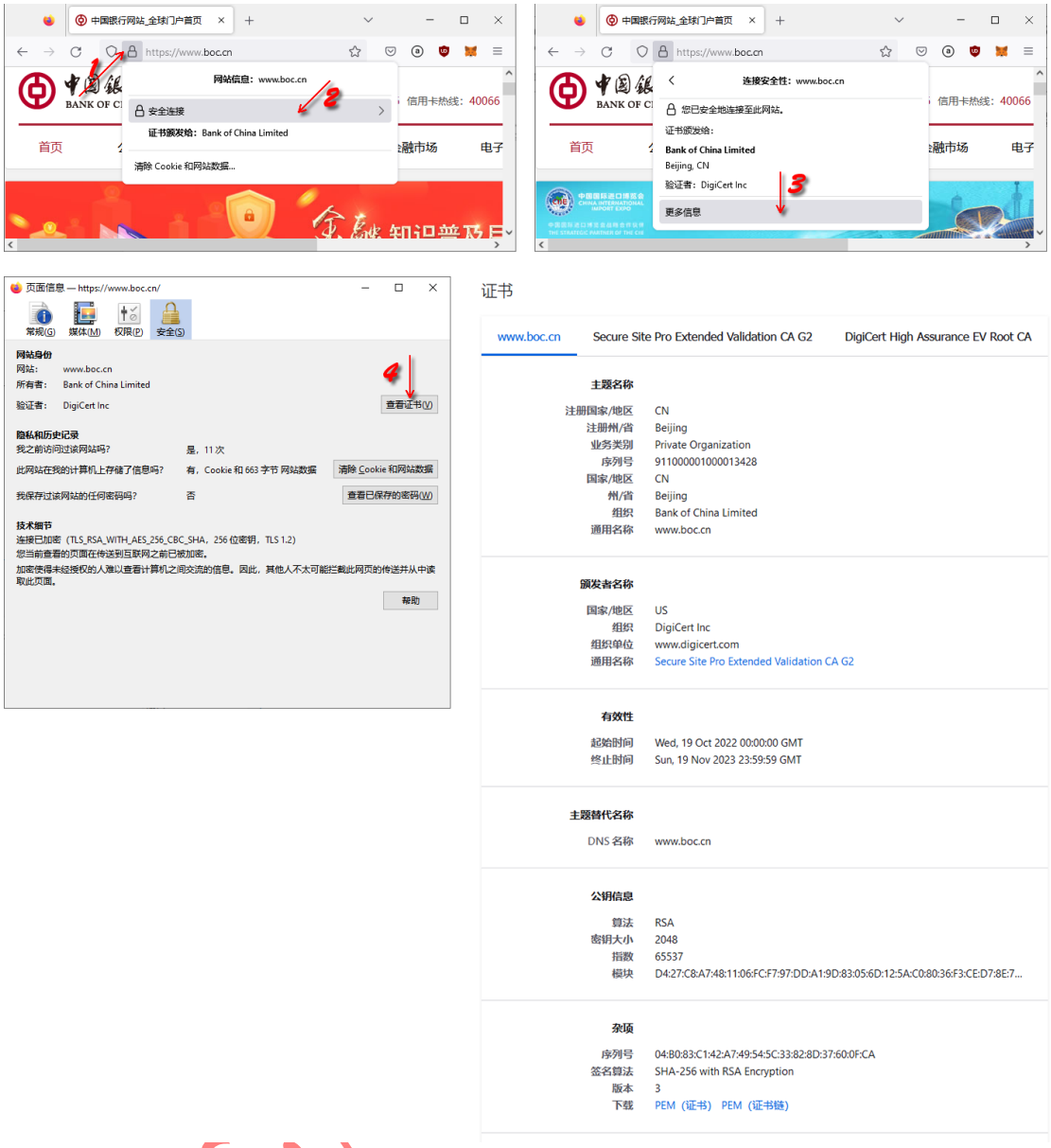


图 10-10 查看中国银行网站的数字证书

能颁发数字证书的权威机构称为 CA（Certificate Authority），也就是营运 PKI 的机构，PKI 是公钥基础设施(Public Key Infrastructure)的英文缩写，由于这个机构的公钥得到了普遍认可（比如浏览器都保存了此公钥），所以它能充当一个公证人的角色，即凡是用它的私钥签过名的公钥都被认为是可信的。被 CA 的私钥签过名的公钥就称为数字证书，以前有一个术语叫 SSL 证书，现在改名为 TLS 证书。用户如要获得数字证书，需要按年支付费用，在中国还不便宜。因此国外有技术员就很不高兴，众所周知“技术大咖一不高兴后果就会很严重”，尼玛不就是一串字符嘛，你每年都收我这么多钱，不讲武德啊，所以技术大咖们就搞出了开源数字证书的 CA，名叫 Let’s Encrypt，用户申请证书不要钱，但证书有效期只有三个月，所以每三个月要刷新一下证书，可去其官网 https://letsencrypt.org/进一步了解如何使用。本书作者个人的网站（https://www.veryopen.org）就是采用这个开源数字证书的，我使用 letsencrypt 证书的切身过程如下：

我们采用 LetsEncrypt 制作证书，并在网站上启用，先要确认 nginx 支持 TLSv1.2 或以上版本（执行 nginx -V 查看）。我用的是 nginx 网站服务，跑在 debian 11 操作系统上。

1. 安装颁发 LetsEncrypt 证书的软件 certbot：

```
root@instance-2:~# apt install certbot -y
```

如果要在 Rocky 9 上安装 certbot 软件，必须先执行命令“dnf install epel-release -y”增加安装源才行。

2．关闭 nginx 网站服务：

```
root@instance-2:~# systemctl stop nginx
```

3．产生证书：

```
root@instance-2:~# certbot certonly --dry-run --nginx -d veryopen.org,www.veryopen.org                    #先测试下
root@instance-2:~# certbot certonly --nginx -d veryopen.org,www.veryopen.org                              #正式颁发证书
```

产生的证书在/etc/letsencrypt/live/www.veryopen.org 目录下，我们需要 fullchain.pem、privkey.pem 两个文件，certbot 会自动修改网站的配置文件/etc/nginx/sites-enabled/veryopenorg.conf 如下：

```
server {
        server_name www.veryopen.org veryopen.org;
        root /var/www/veryopen.org;
        index index.php index.html index.htm;
        error_log /var/log/nginx/wordpress_error.log;
        access_log /var/log/nginx/wordpres_access.log;
        client_max_body_size 100M;
        location / {
                #try_files  $uri $uri/ /index.php?$args;
                try_files  $uri $uri/ /api/v1/api.php?$args;
        }
        location ~ \.php$ {
                include snippets/fastcgi-php.conf;
                fastcgi_pass unix:/run/php/php7.4-fpm.sock;
                fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        }

        listen 443 ssl; # managed by Certbot
        ssl_certificate /etc/letsencrypt/live/www.veryopen.org/fullchain.pem; # managed by Certbot
        ssl_certificate_key /etc/letsencrypt/live/www.veryopen.org/privkey.pem; # managed by Certbot
        include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
        ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot


        add_header Strict-Transport-Security "max-age=31536000" always; # managed by Certbot


        ssl_trusted_certificate /etc/letsencrypt/live/www.veryopen.org/chain.pem; # managed by Certbot
        ssl_stapling on; # managed by Certbot
        ssl_stapling_verify on; # managed by Certbot

}
server {
        listen 80;
        if ($host = www.veryopen.org) {
                return 301 https://$host$request_uri;
        } # managed by Certbot
#这里省略了若干行，实际上这些行是没有用的
}
```

配置文件里有两个 server，最下面那个 server 监听 80 端口（对应 http://协议），当用户浏览 http://www.veryopen.org 时匹配下面的 server，里面有一个判断：只要网址是 www.veryopen.org 就重定向到 https://www.veryopen.org。上面那个 server 监听 443 端口（对应 https://协议），因此不管用户浏览 http://www.veryopen.org 还是 https://www.veryopen.org，最终都会匹配到上面的那个 server。

4．最后重启 nginx 即可：

```
root@instance-2:~# systemctl reload nginx
```

5．以后每过 90 天执行下面的命令更新证书：

```
root@instance-2:~# certbot renew
```

或者做一个定时任务：

```
root@instance-2:~# crontab -e
……
0 4 1 */3 * /usr/bin/certbot renew --force-renewal
```

意思是每隔三个月的 1 号 4:00 点钟运行命令/usr/bin/certbot renew --force-renewal，对所有的 https 证书进行强行更新。

如果是企业内部使用证书，比如 IT 系统的东西流量的数字证书（如 Kubernetes 集群中的通信双方大量采用数字证

书），可以采用自签数字证书——企业内部自己搭建一个 CA 并产生一对 CA 公/私钥，然后用这个 CA 私钥签名其他公钥来产生数字证书，并把 CA 的公钥下发给各个需要验证数字证书的服务端和客户端。

做个类比：数字证书相当于身份证，CA 相当于公安机关，我相信你的数字证书是因为它是 CA 颁发的，同样，我相信你的身份证是因为它是公安机关制作的。

接下来我们实战：生成 www.abc.com 域名的自签数字证书。

1．生成 CA 私钥和 CA 证书：

```
生成 2048 位的 CA 私钥，私钥保存在 ca.key 文件中：
[root@master1 ~]# openssl genrsa -out ca.key 2048

由私钥导出公钥,并自签生成 CA 证书，CA 证书文件为 ca.crt：
[root@master1 ~]# openssl req -new -x509 -days 3650 -key ca.key -subj "/CN=Myself-CA" -out ca.crt
```

CA 证书有效期十年，CA 机构名叫 Myself-CA。有了 ca.key 和 ca.crt，我们就可以给其他人签发数字证书了。

2．为 www.abc.com 生成私钥和证书申请文件：

```
产生私钥文件 abc_com.key：
[root@master1 ~]# openssl genrsa -out abc_com.key 2048

由私钥生成证书申请文件 abc_com.csr：
[root@master1 ~]# openssl req -new -key abc_com.key -subj "/CN=www.abc.com" -out abc_com.csr
```

3．签发数字证书（证书有效期 1 年）：

```
[root@master1 ~]# openssl x509 -req -in abc_com.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out abc_com.crt \
 -days 365 -subj "/CN=www.abc.com"
Certificate request self-signature ok
subject=CN = www.abc.com

[root@master1 ~]# ll ca.* abc_com*
-rw-r--r-- 1 root root  993 Aug 11 10:34 abc_com.crt                    #www.abc.com 域名的证书
-rw-r--r-- 1 root root  895 Aug 11 10:30 abc_com.csr
-rw------- 1 root root 1704 Aug 11 10:26 abc_com.key                    #www.abc.com 域名的私钥
-rw-r--r-- 1 root root 1115 Aug 11 10:20 ca.crt                         #CA 的证书
-rw------- 1 root root 1704 Aug 11 10:12 ca.key                         #CA 的私钥
-rw-r--r-- 1 root root   41 Aug 11 10:34 ca.srl
```

获取到域名数字证书之后，就要修改网站的配置文件，加入证书的配置参数。网站服务器不同，网站的配置文件存放目录也不同，比如 nginx 的网站配置文件放在/etc/nginx/sites-enabled 目录下，而 apache 的网站配置文件统一放在/etc/httpd/conf.d 目录种。

现在如果你足够聪明，你应该会想到这么一个问题：万一某个 CA 出问题（比如私钥被泄露），岂不是由它签发的数字证书都遭殃？我们都知道，私钥被泄露，唯一补救的方法就是重新生成新的私钥，原来的私钥和公钥作废！CA 的公钥作废又会导致什么灾难？不敢想象。

目前人类应对 CA 问题的唯一方法就是尽量扩大 CA 的数量，这样由每个 CA 签发的数字证书就少一些，这样即使某个 CA 出问题，影响的范围也就有限。但 CA 数量多了又会带来另外一个问题，即 CA 公钥管理就很麻烦，比如几十万个 CA 公钥要填充到浏览器中、操作系统中、以及各种形形色色的服务器中和客户端中，不说其他的，光占用的磁盘和内存空间就让人受不了。后来地球人类想到了一个折中的办法：指定几十个 CA 为根 CA，由根 CA 签发更多数量的中间层 CA，虽然根 CA 和中间层 CA 都可以签发用户数字证书，但现实中很少直接采用根 CA 来签发用户证书。中间层 CA 可以有多个层级，这样形成一个链式的证书层级，称为证书链。多层证书链的生成和验证过程和两层（根证书直接生成用户证书）的情况是类似的，只是要多验证几次。比如三层证书链的生成和验证过程如下：

数据签名的过程：证书机构生成自签名根证书 → 根证书签发中间证书 →中间证书签发用户证书 → 用户采用用户证书签名用户数据。

验证签名的过程：用中间证书验证用户证书的签名 → 采用根证书验证中间证书的签名 → 采用用户证书验证数据中的签名。

证书中存在两个唯一名字 DN（Distinguished Name），其中一个 DN 为证书本身的名字，另一个 DN 为签发者的名字，由每一个的证书的签发者名字一直可以追溯到根证书，根证书里的两个 DN 的值相同。

下面我们来实战三层证书链。

1. 签发：

```
生成根 CA 的私钥 rootCA.key:
[root@master1 ~]# openssl genpkey -algorithm RSA -out rootCA.key -quiet

自签发根 CA 证书 rootCA.crt (有效期十年):
[root@master1 ~]# openssl req -new -x509 -days 3650 -key rootCA.key -subj "/C=CN/CN=Root-CA" -out rootCA.crt

生成中间层 CA 的私钥 interCA.key:
[root@master1 ~]# openssl genpkey -algorithm RSA -out interCA.key -quiet

生成证书申请文件 interCA.csr:
[root@master1 ~]# openssl req -new -key interCA.key -subj "/C=CN/ST=Guangdong/CN=Inter-CA" -out interCA.csr
```

后面用到的 openssl ca 命令实现了一个简单的证书机构，会使用一个文本数据库来记录生成的证书，我们需要生成该数据库索引文件：

```
[root@master1 ~]# mkdir db
[root@master1 ~]# touch db/index

[root@master1 ~]# vim interCA.conf
[ ca ]
default_ca = intermediate_ca

[ intermediate_ca ]
dir = .
private_key = $dir/rootCA.key
certificate = $dir/rootCA.crt
new_certs_dir = $dir/
serial = $dir/crt.srl
database = $dir/db/index
default_md = sha256
policy = policy_any
email_in_dn = no

[ policy_any ]
domainComponent = optional
countryName = optional
stateOrProvinceName = optional
localityName = optional
organizationName = optional
organizationalUnitName = optional
commonName = optional
emailAddress = optional

[ ca_ext ]
keyUsage                = critical,keyCertSign,cRLSign
basicConstraints        = critical,CA:true
subjectKeyIdentifier    = hash
authorityKeyIdentifier  = keyid:always
```

和用户证书不同的是，中间证书需要在证书的 basicConstraints 中设置 CA:true 标签，以标明该证书属于证书机构的证书，可以用于签发和验证用户证书。

```
签发中间层证书 interCA.crt:
[root@master1 ~]# openssl ca -config interCA.conf -days 365 -create_serial -in interCA.csr -out interCA.crt \
 -extensions ca_ext -notext
Using configuration from interCA.conf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
countryName           :PRINTABLE:'CN'
stateOrProvinceName   :ASN.1 12:'Guangdong'
commonName            :ASN.1 12:'Inter-CA'
Certificate is to be certified until Aug 21 09:11:04 2024 GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated

查看中间证书的内部信息:
[root@master1 ~]# openssl x509 -in interCA.crt -text -noout
```

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            78:90:95:3a:f3:00:60:1b:16:e2:f8:bc:e0:3d:79:18:69:c2:1e:eb
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = CN, CN = Root-CA ----------#签发者的 DN
        Validity
            Not Before: Aug 22 09:11:04 2023 GMT
            Not After : Aug 21 09:11:04 2024 GMT
        Subject: C = CN, ST = Guangdong, CN = Inter-CA ----------#证书本身的 DN
        Subject Public Key Info:
......
        X509v3 extensions:
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE ---------- #表明本证书为 CA 证书，只有 CA 证书才可签发证书
......

生成 RSA 型私钥 Alice.key:
[root@master1 ~]# openssl genpkey -algorithm RSA -out Alice.key -quiet

生成证书申请文件 Alice.csr:
[root@master1 ~]# openssl req -new -key Alice.key -subj "/CN=Alice" -out Alice.csr

用中间层证书签发用户证书:
[root@master1 ~]# openssl x509 -req -in Alice.csr -CA interCA.crt -CAkey interCA.key -CAcreateserial -out Alice.crt
Certificate request self-signature ok
subject=CN = Alice
```

2．验证：

```
[root@master1 ~]# openssl verify -show_chain -CAfile rootCA.crt -untrusted interCA.crt Alice.crt
Alice.crt: OK
Chain:
depth=0: CN = Alice (untrusted)
depth=1: C = CN, ST = Guangdong, CN = Inter-CA (untrusted)
depth=2: C = CN, CN = Root-CA
```

可以把根证书导入到 Rocky 9 操作系统中，这样以后验证时就不要特别指明根证书了。导入根证书的方法如下：

```
[root@master1 ~]# dnf install ca-certificates -y                                    #这个软件一般都已经安装了

[root@master1 ~]# cp rootCA.crt /etc/pki/ca-trust/source/anchors/

任何时候往/etc/pki/ca-trust/source 中增删了证书，都要更新信任链:
[root@master1 ~]# update-ca-trust

查看刚增加的信任证书:
[root@master1 ~]# trust list --filter=ca-anchors|grep ": Root-CA" -i -A 2 -B 3
pkcs11:id=%35%31%F6%E8%7C%8C%12%85%88%C6%47%11%46%7B%59%92%57%24%B2%35;type=cert
    type: certificate
    label: Root-CA
    trust: anchor
    category: authority
```

在操作系统中加入了根证书后，以后验证就可以这样：

```
[root@master1 ~]# openssl verify -show_chain  -untrusted interCA.crt Alice.crt
Alice.crt: OK
Chain:
depth=0: CN = Alice (untrusted)
depth=1: C = CN, ST = Guangdong, CN = Inter-CA (untrusted)
depth=2: C = CN, CN = Root-CA
```

§10.4 openSSL/GPG

在当今世界，网络威胁手段层出不穷，确保我们的数据安全就显得至关重要，这就是 OpenSSL 和 GPG 等加密工具的用武之地。跨互联网的应用，必须确保“管线”和“端点”安全，“管线”就先计算机网络，而“端点”就是终端设备（如电脑、手机等）。

开源软件 openSSL 是 SSL 和 TLS 协议的具体实现，这两个协议用于网络通讯安全，重在建立安全的跨互联网的通信管道（比如实现 https），打造“管线”安全。openSSL 既是一套开发库，也提供了一系列的命令行工具（如 openssl），在 Linux 上，我们可直接使用 openssl 轻松生成公/私钥、数字证书等。官方网站为 https://www.openssl.org。

而 GPG 全称是 GnuPG，是 OpenGPG 标准协议的具体实现，包含了开发库和若干应用程序（如 windows 上的 Gpg4win、linux 上 gpg 和 gpg2 等）。OpenGPG 协议主要被终端用户用来加密和签名数据资料——文档、邮件等，确保“端点”安全。

§10.4.1 openssl 工具

本节介绍 Linux 上的 openssl 命令行工具的用法。

openssl 命令功能强大，功能大致分成以下几类：

- 生成和管理公钥、私钥以及相关生成参数
- 使用公/私钥加密、解密、签名、验证
- 管理证书（X.509 格式）、认证请求和证书吊销列表
- 计算摘要（hash），支持各种摘要算法
- 使用各种加密方法(cipher)加密、解密
- 测试 SSL 和 TLS 连接
- 对 email 签名或加密（S/MIME）格式
- 生成和验证时间戳

官网 https://www.openssl.org/docs/manmaster 由详细的文档。

默认已经安装，如果没有安装，请使用下面的命令安装它：

```
[root@master1 ~]# dnf install openssl -y
[root@master1 ~]# openssl version -a
OpenSSL 3.0.7 1 Nov 2022 (Library: OpenSSL 3.0.7 1 Nov 2022)
built on: Wed Jul 12 00:00:00 2023 UTC
platform: linux-x86_64
options:  bn(64,64)
compiler: gcc -fPIC -pthread -m64 ……,-DSYSTEM_CIPHERS_FILE="/etc/crypto-policies/back-ends/openssl.config"
OPENSSLDIR: "/etc/pki/tls"
ENGINESDIR: "/usr/lib64/engines-3"
MODULESDIR: "/usr/lib64/ossl-modules"
Seeding source: os-specific
CPUINFO: OPENSSL_ia32cap=0xfffa32034f8bffff:0x9c27ab
```

OPENSSLDIR 指定了 openssl 的工作目录，其中文件/etc/pki/tls/openssl.cnf 包含默认的参数，可以通过环境变量 OPENSSL_CONF 或者 openssl 的命令行参数-config 指定特定的参数文件。openssl 命令的参数非常多，为了简化 openssl 命令行输入，对于常用的参数建议放在参数文件重。

https://www.veryopen.org

一、生成和管理公钥、私钥以及相关生成参数

1. 生成私钥

选择哪种算法？私钥需要多少位？要不要用密码保护私钥？在你正式生成私钥前必须想清楚这三个问题。生成私钥的命令语法如图 10-12 所示。

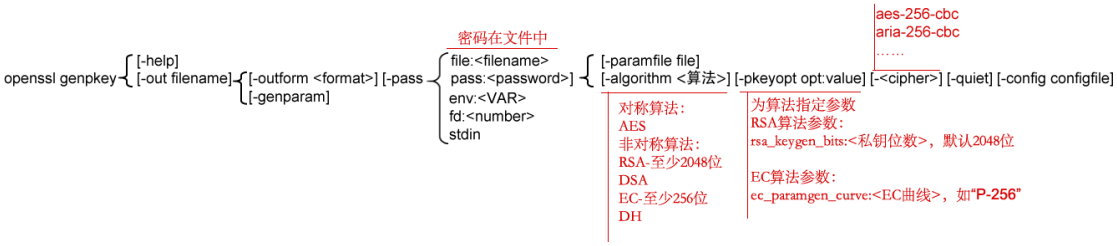


图 10-12 生成私钥的命令语法

一些参数说明如下：

-out filename：生成的私钥保存的文件名，如-out ~/myprivate.pem。

-outform <format>：私钥格式，可取值 DER（二进制格式）、P12（包含 PKCS#12 对象的 DER-编码）、PEM（base64 编码的文本）、SMIME（一 S/MIME 对象），默认是 PEM。

-pass pass:<password>|file:<filename>|env:<VAR>|fd:<number>|stdin：指定用于加密私钥的密码，其中 pass:<password>直接在命令行上指定密码，如-pass pass:A123456；file:<filename>表示文件中第一行就是密码，如-pass file:~/pass.txt；而 env:<VAR>表示环境变量 VAR 的值做密码，如-pass var:MYPASS；stdin 表示密码从键盘上输入，这是默认值。

-algorithm <算法>：指定公钥算法，有这些：RSA、DSA、EC、DH、RSA-PSS、DHX、X25519、X448、ED25519 等等。RSA 诞生最早，目前低于 1024 位的私钥已经不安全了，建议采用 2048 或者 4096 位的，不过位数越多，消耗的计算资源就越大。EC 是基于椭圆曲线的，256 位就非常安全，比特币就是采用 EC 算法生成的 256 位密钥，EC 比 RSA 要好很多，所以建议大家在生产环境中尽量采用 EC。目前基于 EC 算法的椭圆曲线有很多，我们在使用 EC 算法时还需要指明使用什么椭圆曲线，可用命令“openssl ecparam -list_curves”列出全部椭圆曲线，目前支持 secp224r1、secp256k1、secp384r1、secp521r1、prime256v1、brainpoolP256r1、brainpoolP256t1、brainpoolP320r1、brainpoolP320t1、brainpoolP384r1、brainpoolP384t1、brainpoolP512r1、brainpoolP512t1。用法请看后面的例子。

-pkeyopt <opt:value>：为公钥算法指定相应的参数，显然算法不同，参数也不尽相同。对于最流行的算法 RSA，对应的参数有 rsa_keygen_bits:<私钥位数>，比如-pkeyopt rsa_keygen_bits:2048，默认就是 2048 位。-pkeyopt 参数可多次使用，且必须位于“-algorithm <算法>”参数的后面。

-<cipher>：指定用于加密私钥的算法，常用的有 aes256、aria256 等（可有用“openssl list -cipher-algorithms”列出全部支持的加密算法）。

-genparam：产生参数集合，而不是私钥。

-paramfile file：从参数文件生成私钥。

-quiet：不显示过程。

命令 openssl genrsa ...和 openssl gendsa ...也能产生相应的私钥。

看例子：

```
[root@master1 ~]# openssl genpkey -algorithm RSA -quiet                    #生成私钥并显示在屏幕上
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQCSNoTLdlVwjg1f
xqmeE9U32DZp4QqvnJSrhojbI0Z6SHqnlNvpOgu0LjKDEnIXzL7s2D/bvfX5xc59
……
qmCqWBo9B5jHLkiHgO5FUdg=
```

https://www.verypen.org

```
-----END PRIVATE KEY-----

[root@master1 ~]# openssl genpkey -algorithm RSA -quiet -out ./private.pem           #生成私钥保存在 private.pem 文件中
[root@master1 ~]# ll ./private.pem
-rw------- 1 root root 1704 Aug 14 16:59 ./private.pem

生成私钥文件 mykey.pem，并加密保存：
[root@master1 ~]# openssl genpkey -algorithm RSA -quiet -out ./mykey.pem -aes-256-cbc
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:

产生 1024 位的私钥，并加密保存：
[root@master1 ~]# openssl genpkey -algorithm RSA -quiet -out ./mykey.pem -aes-256-cbc -pkeyopt rsa_keygen_bits:1024
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:

[root@master1 ~]# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -out key.pem
[root@master1 ~]# ll key.pem
-rw------- 1 root root 241 Aug 14 17:09 key.pem

产生密钥参数文件 dsap.pem，要求采用 DSA 算法，长度 2048 位：
[root@master1 ~]# openssl genpkey -genparam -algorithm DSA -out dsap.pem -pkeyopt dsa_paramgen_bits:2048

直接从密钥参数文件产生私钥：
[root@master1 ~]# openssl genpkey -paramfile dsap.pem -out dsakey.pem

产生用密码保护的私钥：
[root@master1 ~]# openssl genpkey -algorithm RSA -aria-256-cbc -pass pass:A1b2C3 -out safekey.pem

产生基于 EC 算法的 secp256k1 椭圆曲线的私钥：
[root@master1 ~]# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:secp256k1 -out bitkey1.pem
```

2. 管理私钥和公钥

主要采用“openssl pkey ...”命令，其语法如下：

```
openssl pkey { [-help]
               [-check] [-pubcheck] [-in <filename>] [-inform <format>] [-passin arg] [-pubin] [-out <filename>] [-outform DER|PEM] [-<cipher>] \
                                                                                                                    aes-256-cbc
                                                                                                                    aria-256-cbc
                                                                                                                    ......
                    [-text] [-passout arg] [-pubout] [-noout] [-ec_conv_form arg] [-ec_param_enc arg]
```

一些参数说明如下：

-check：检查公钥和私钥是否是合法的一对。

-pubcheck：检查公钥或私钥中的公钥组件的正确性。

-in <filename>：指定输入文件，可为私钥或公钥文件。

-inform <format>：输入密钥（私钥或公钥）的格式，可取值 DER（二进制格式）、P12（包含 PKCS#12 对象的 DER-编码）、PEM（base64 编码的文本）、ENGINE。

-passin arg：输入钥匙的保护密码，具体介绍请参考上面一条命令的-pass 参数。

-pubin：指定由参数-in 定义的文件是公钥文件，默认是私钥文件。

-out <filename>：指定输出文件名。

-outform DER|PEM：定义输出文件的格式。

-<cipher>：私钥的加密算法，请参考上一条命令的同名参数。

-text：针对 PEM 格式的私钥，以纯文本形式输出各个关键组件。

-passout arg：输出私钥的保护密码，格式请参考上一条命令的-pass 参数。

-pubout：只输出公钥。默认是既输出公钥，也输出私钥。

-noout：不输出编码后的密钥（公钥或者私钥）。

-ec_conv_form arg、-ec_param_enc arg：这两个参数值针对基于椭圆曲线的钥匙。-ec_conv_form 参数指定如何将椭圆曲线上的点转换为八位字节字符串，可取值 compressed（默认值）、uncompressed 和 hybrid。-

ec_param_enc 参数指定椭圆曲线参数的编码方式，可取值 named_curve、explicit。关于“椭圆曲线密钥生成算法”，请自行去网上了解。

下面我们一些举例以加深理解上面的各个参数：

```
[root@master1 ~]# openssl genpkey -algorithm RSA -out private.pem -quiet                                    #生成私钥
[root@master1 ~]# openssl pkey -in private.pem -text                                                        #查看密钥的组成
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQCPyWKplRT3Zmny
……
vVfZt6agEICCEOanP23gdImpCDieBtYxe5CZ7z115LCq4eHRYzPqRBvdmwr3HyDu
BKSIfTwXHDjBMCDORLyo5xI=
-----END PRIVATE KEY-----
Private-Key: (2048 bit, 2 primes)
modulus:                                                                                                    #模数 n
    00:8f:c9:62:a9:95:14:f7:66:69:f2:b7:57:ce:aa:
    42:bc:22:e7:1f:0a:a0:60:00:57:8a:c6:97:4f:fd:
    ……
    c3:99:e5:c0:94:6b:3d:cb:97:a3:be:da:a3:34:8d:
    e6:1f
publicExponent: 65537 (0x10001)                                                                             #公钥因子 e
privateExponent:                                                                                            #私钥因子 d
    14:b9:fb:dd:e4:1e:94:97:0f:b8:ac:13:1d:38:4f:
    ……
    18:31:d2:26:be:34:f1:00:fd:0f:a1:c2:da:fa:38:
    e1
prime1:                                                                                                     #素数因子 p
    00:bc:46:cf:f5:c5:eb:bd:f1:93:5b:ea:c7:14:3a:
    ……
    6f:90:70:66:d7:96:b1:2d:23:35:f6:9b:41:40:ab:
    ed:48:7b:34:1e:73:d7:25:71
prime2:                                                                                                     #素数因子 q
    00:c3:81:c7:4b:82:a1:6f:66:2f:4f:67:d9:e1:25:
    ……
    75:ac:d7:57:a6:09:71:bc:8f
exponent1:                                                                                  #等于 d mod (p - 1)
    00:b7:e1:95:24:1d:dc:83:13:98:6f:32:5a:a2:d7:
    ……
    96:01:af:4a:94:df:60:66:11
exponent2:                                                                                  #等于 d mod (q - 1)
    1f:d6:12:4e:09:87:a0:4c:53:87:ea:b1:60:28:70:
    ……
    cd:2f:83:e6:55:c3:5f:a1
coefficient:                                                                                                #CRT 系数
    7c:a8:ee:66:e3:ba:ba:bb:39:b2:36:c3:42:48:ee:
    ……
    30:20:ce:44:bc:a8:e7:12
```

关于 RSA 密钥中的各个组成部分的详细了解，请上网搜索“RSA 算法原理”。

```
[root@master1 ~]# openssl pkey -in private.pem -out public.pem -pubout                                      #从私钥导出公钥
[root@master1 ~]# openssl pkey -check -in private.pem -noout                                                #检查密钥是否合法
Key is valid
[root@master1 ~]# openssl pkey -pubcheck -in public.pem -pubin                                              #检查公钥是否合法
Key is valid
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAj8liqZUU92Zp8rdXzqpC
vCLnHwqgYABXisaXT/0iB9PGkiSPYbL7s2i/A1tk1CPPS0eEl8+ZekT2Q0l6+G0Z
kyS3txgA64PsMnp+ztIZdqQaRZJ/tm9j5TQWOpGpe03UuWoNtF6x96kAQOqf3w5A
FpdMBUOvYrLHDn3lqAmSqCUGe8EyfItibfsN+0Y3xZyGCQjnuSa4srywVIKY1psv
Zpner9tjX7+7thDxSwsDOLjIVUU5UyKw96sWrXib7RCFAE7zyUrjv7bpuBmVZ8Xj
+o7C7pCO+gJKvG7yv6bNflrb206/VQ4yuuplvh5zpsrDmeXAlGs9y5ejvtqjNI3m
HwIDAQAB
-----END PUBLIC KEY-----

下面两条命令都是给私钥添加保护密码（有什么区别？）：
[root@master1 ~]# openssl pkey -in private.pem -aes256 -out private1.pem
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
[root@master1 ~]# openssl pkey -in private.pem -aes256 -out private2.pem -passout pass:A1b2C3

移除保护密码：
[root@master1 ~]# openssl pkey -in private2.pem -passin pass:A1b2C3 -out mykey.pem

产生 256 位基于椭圆曲线的密钥（比特币就是这种密钥）：
[root@master1 ~]# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:secp256k1 -out bitkey.pem
[root@master1 ~]# openssl pkey -in bitkey.pem -text
-----BEGIN PRIVATE KEY-----
MIGEAgEAMBAGByqGSM49AgEGBSuBBAAKBG0wawIBAQQgs5GduiWqahWgKmQTXShS
KVP+Uq6BfPUUn+G/vP7i+IOhRANCAAQP8soNwJvc/lqp/e/GNHzUaPmjmMco99ku
uwu+sE4eDVAHIPo0Y8rekfQh6joI/X3AZOPHpRB7VfwspKBg/IOS
-----END PRIVATE KEY-----
Private-Key: (256 bit)
priv:
    b3:91:9d:ba:25:aa:6a:15:a0:2a:64:13:5d:28:52:
    29:53:fe:52:ae:81:7c:f5:14:9f:e1:bf:bc:fe:e2:
    f8:83
pub:
```

```
    04:0f:f2:ca:0d:c0:9b:dc:ff:5a:a9:fd:ef:c6:34:
    7c:d4:68:f9:a3:98:c7:28:f7:d9:2e:bb:0b:be:b0:
    4e:1e:0d:50:07:20:fa:34:63:ca:de:91:f4:21:ea:
    3a:08:fd:7d:c0:64:e3:c7:a5:10:7b:55:fc:2c:a4:
    a0:60:fc:83:92
ASN1 OID: secp256k1

修改椭圆曲线参数的编码方式为 explicit:
[root@master1 ~]# openssl pkey -in bitkey.pem -ec_param_enc explicit -out key1.pem
```

二、使用公/私钥加密、解密、签名、验证

主要采用“openssl pkeyutl ...”命令，其使用语法如下：

```
openssl pkeyutl [-help] [-in <file>] [-rawin] [-digest <algorithm>] [-out <file>] [-sigfile <file>] [-inkey
<filename|uri>] [-keyform <format>] [-passin arg] [-peerkey <file>] [-peerform <format>] [-pubin] [-certin] [-sign]
[-verify] [-verifyrecover] [-encrypt] [-decrypt] [-derive] [-kdf <algorithm>] [-kdflen <length>] [-pkeyopt
opt:value] [-pkeyopt_passin opt[:passarg]] [-hexdump] [-asn1parse] [-config configfile]
```

其中参数：

-help：获取在线帮助。

-in <file>：指定输入数据文件，比如待签名的文件。

-rawin：表名输入数据是没有经过哈希计算的原始数据，可以结合参数-digest 指明摘要算法，-rawin 参数值允许结合-sgin 或-verify 参数一起使用，并且要求生成私钥的算法是 Ed25519 或 Ed448。

-digest <algorithm>：指定摘要算法。在使用密钥签名或验证签名之前，需要使用摘要算法对输入数据进行哈希处理。 如果签名算法不需要摘要计算（例如 EdDSA），则可以省略此参数。 如果省略此参数但签名算法又需要摘要时，则将使用默认摘要算法，对于 RSA、DSA 和 ECDSA 等签名算法，SHA-256 是默认的摘要算法，对于 SM2 签名算法，默认的摘要算法是 SM3。 如果参数-digest 存在，则还必须指定 -rawin 参数。支持的摘要算法有 sha1、sha256、sm3、md5、md5-sha1、sha1、rmd160、sha512、shake256 等，具体可用命令“openssl dgst -list”列出全部的摘要算法。

-out <filename>：指定输出文件。

-sigfile <file>：指定签名文件，-verify 参数需要用到签名文件。

-inkey <filename|uri>：密钥文件，默认是私钥文件。

-keyform <format>：密钥格式，可取值 DER|PEM|P12|ENGINE。

-passin arg：输入密钥（由参数-inkey 指定）的保护密码。

-peerkey <file>：对等密钥文件，在做密钥派生（协议）操作时使用。

-peerform <format>：对等密钥的格式，可取值 DER|PEM|P12|ENGINE。

-pubin：由参数-inkey 指定的文件是公钥文件。如果-inkey 指定的文件是私钥文件，且-pubin 参数没省略，那么就先从私钥导出公钥，然后再使用这个公钥。

-certin：由参数-inkey 指定的文件是包含公钥的证书。

-sign：对输入数据（必须是哈希值）签名，并输出已签名结果。签名需要用到私钥。

-verify：根据签名文件验证输入数据（必须是哈希值），验证结果要么成功要么失败

-verifyrecover：验证输入数据（必须是哈希）并输出恢复的数据。

-encrypt：使用公钥加密数据。

-decrypt：使用私钥解密数据。

-derive：使用对等密钥派生共享秘密，只有 EC 密钥算法才支持这个参数。

https://www.yeyoben.com

-kdf：指定密钥导出算法。目前支持的算法有 TLS1-PRF 和 HKDF 等（可用命令“openssl list -kdf-algorithms”列出支持的全部算法）。 注意：通常必须设置附加参数和 KDF 输出长度才能使其工作。

-kdflen <length>：指定 KDF 算法的输出长度。

-pkeyopt ope:value：

-pkeyopt_passin opt[:passarg]：允许从键盘输入或密码源读取公钥选项。如果仅指定 opt，系统将提示用户在键盘上输入密码。

-hexdump：以十六进制格式输出。

-asn1parse：解析 ASN.1 输出数据。当我们签名一个 ASN1 结构时，本参数要与-verifyrecover 结合使用，效果会很好。也可以在线解析：https://lapo.it/asn1js/。

-config configfile：指定配置文件，配置文件中一般保存一些常用的参数。

密钥算法不同，能支持的操作也不同，比如 RAS 算法适用性最广，能支持加密（-encrypt 参数）、解密（-decrypt）、签名（-sign）、验证签名（-verify、-verifyrecover）。而 DSA 算法只支持签名和验证签名、DH、X25519 和 X448 算法只支持派生操作（-derive），EC 算法支持签名、验证签名和派生。ED25519 和 ED448 算法只支持签名和验证。SM2 算法支持加密、解密、签名和验证签名，但是对于签名和验证签名，要求提供唯一的字串（-pkeyopt distid:<string>|hexdistid:<hex_string>）。

另外签名操作的数据有长度限制，密钥的算法不同，待签名数据的长度也不同，对于 RSA 算法，待签名数据的长度不能超过模数 n。对于 ECDSA 和 DSA，数据不应长于字段大小，否则它将被自动截断为字段大小。无论如何，输入数据的长度不得大于支持的最大摘要的长度。所以如果要对一个很大的文件签名，就必须先计算它的摘要，然后对摘要签名。

下面看例子：

1．准备两对公/私钥：

```
[root@master1 ~]# openssl genpkey -algorithm RSA  -out private1.pem -quiet                    #生成私钥 1
[root@master1 ~]# openssl pkey -in private1.pem -out public1.pem -pubout                      #导出公钥 1
[root@master1 ~]# openssl genpkey -algorithm RSA  -out private2.pem -quiet                    #生成私钥 2
[root@master1 ~]# openssl pkey -in private2.pem -out public2.pem -pubout                      #导出公钥 2

[root@master1 ~]# ll private?.pem public?.pem
-rw------- 1 root root 1704 Aug 15 11:26 private1.pem                                        #私钥
-rw------- 1 root root 1704 Aug 15 15:20 private2.pem                                        #私钥
-rw-r--r-- 1 root root  451 Aug 15 11:56 public1.pem                                         #公钥
-rw-r--r-- 1 root root  451 Aug 15 15:20 public2.pem                                         #公钥
```

2．签名与验证签名：

```
[root@master1 ~]# vim file.dat                                                               #输入下面一行内容
这次女足世界杯，姚明没去踢球，真是没有发挥中国乒乓球的应有水平，这就是蔡赞这个羽毛球队长的失策啊，全国人民要求郎平下课。

[root@master1 ~]# openssl pkeyutl -sign -inkey private1.pem -in file.dat -out file.dat.sig
Error: The input data looks too long to be a hash
```

什么鬼？就一行字也太长了？！

```
[root@master1 ~]# openssl pkeyutl -rawin -digest sha256 -sign -inkey private1.pem -in file.dat -out file.dat.sig
[root@master1 ~]# ll file.dat*
-rw-r--r-- 1 root root 181 Aug 15 11:58 file.dat
-rw-r--r-- 1 root root 256 Aug 15 14:23 file.dat.sig

验证签名：
[root@master1 ~]# openssl pkeyutl -verify -in file.dat -sigfile file.dat.sig -inkey public1.pem -pubin
Error: The input data looks too long to be a hash
[root@master1 ~]# openssl pkeyutl -rawin -verify -in file.dat -sigfile file.dat.sig -inkey public1.pem -pubin
Signature Verified Successfully
```

对一个小文件签名和验证签名：

```
[root@master1 ~]# echo "To be or not to be,it's a problem!" > smallfile.txt
[root@master1 ~]# openssl pkeyutl -sign -inkey private1.pem -in smallfile.txt -out small.sig
[root@master1 ~]# openssl pkeyutl -verify -in smallfile.txt -inkey public1.pem -pubin -sigfile small.sig
Signature Verified Successfully
```

https://blog.csdn.net/weixin_43936968

下面稍微改动一下数据文件，再验证签名看看：

```
[root@master1 ~]# echo "." >> smallfile.txt
[root@master1 ~]# openssl pkeyutl -verify -in smallfile.txt -inkey public1.pem -pubin -sigfile small.sig
Signature Verification Failure
```

数据文件修改后，验证签名失败了。大家还可以使用另一个公钥 public2.pem 验证签名，看看能否成功。

从签名文件中恢复原始数据：

```
[root@master1 ~]# rm -rf smallfile.txt
[root@master1 ~]# openssl pkeyutl -verifyrecover -in small.sig -inkey public1.pem -pubin
To be or not to be,it's a problem!
```

3．加密与解密：

```
用公钥加密 file.dat，加密后的文件为 file.dat.enc：
[root@master1 ~]# openssl pkeyutl -encrypt -pubin -inkey public1.pem -in file.dat -out file.dat.enc

用私钥解密-显示在屏幕上：
[root@master1 ~]# openssl pkeyutl -decrypt  -inkey private1.pem -in file.dat.enc
这次女足世界杯，姚明没去踢球，真是没有发挥中国乒乓球的应有水平，这真是蔡赟这个羽毛球队长的失策啊，全国人民要求郎平下课。

用私钥解密-解密到文件中：
[root@master1 ~]# openssl pkeyutl -decrypt  -inkey private1.pem -in file.dat.enc -out 1.dat
```

看看能否加密大文件：

```
[root@master1 ~]# ll /boot/vmlinu*
-rwxr-xr-x. 1 root root 12543240 Jul 16 20:47 /boot/vmlinuz-0-rescue-d32189a53fe84d9396ad83386ac9d1fa
-rwxr-xr-x. 1 root root 12543240 Jun 28 17:53 /boot/vmlinuz-5.14.0-333.el9.x86_64
-rwxr-xr-x  1 root root 13349640 Aug  4 20:46 /boot/vmlinuz-5.14.0-350.el9.x86_64

[root@master1 ~]# cp /boot/vmlinuz-5.14.0-350.el9.x86_64 ./vmlinuz
[root@master1 ~]# openssl pkeyutl -encrypt -pubin -inkey public.pem -in vmlinuz -out vmlinuz.enc
Public Key operation error
805B0A449F7F0000:error:0200006E:rsa routines:ossl_rsa_padding_add_PKCS1_type_2_ex:data too large for key size:crypto/rsa/rsa_pk1.c:133:
```

非对称密钥不能加密大文件！这时只有采用对称密钥了，既“openssl enc ...”命令，请参考“五、使用各种加密方法(cipher)加密、解密”。

4．导出共享对称密钥：

```
生成 EC 私钥，曲线类型有 prime256v1, secp224r1, secp384r1, secp256k1, secp521r1：
[root@master1 ~]# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:secp256k1 -out bitkey1.der -outform DER
[root@master1 ~]# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:secp256k1 -out bitkey2.der -outform DER

导出公钥：
[root@master1 ~]# openssl pkey -in bitkey2.der -inform DER -out bitpub2.der -outform DER -pubout
[root@master1 ~]# openssl pkey -in bitkey1.der -inform DER -out bitpub1.der -outform DER -pubout

[root@master1 ~]# openssl pkey -in bitkey1.der -text
-----BEGIN PRIVATE KEY-----
MIGEAgEAMBAGByqGSM49AgEGBSuBBAAKBG0wawIBAQQgmdT/O9lVQSY4K33vaLNR
RerjI+E1XcguXyxV7BzsGsihRANCAAS3yoDH/CUAThNWW7um7Fb1FYShcCbvjO7K
xYXBsbIJcrJ7Hc9GM3p4H0SY1G3dhRu8WLOxGZP/Jk/Qi4M3MGcO
-----END PRIVATE KEY-----
Private-Key: (256 bit)
priv:
    99:d4:ff:3b:d9:55:41:26:38:2b:7d:ef:68:b3:51:
    45:ea:e3:23:e1:35:5d:c8:2e:5f:2c:55:ec:1c:ec:
    1a:c8
pub:
    04:b7:ca:80:c7:fc:25:00:4e:13:56:c3:bb:a6:ec:
    56:f5:15:84:a1:70:26:ef:8c:ee:ca:c5:85:c1:b1:
    b2:09:72:b2:7b:1d:cf:46:33:7a:78:1f:44:98:d4:
    6d:dd:85:1b:bc:58:b3:b1:19:93:ff:26:4f:d0:8b:
    83:37:30:67:0e
ASN1 OID: secp256k1

导出共享密码-用我的私钥和对方的公钥：
[root@master1 ~]# openssl pkeyutl -derive -inkey bitkey1.der -keyform DER -peerkey bitpub2.der -peerform DER -out secret
[root@master1 ~]# cat secret | od -t x1
0000000 c3 ce 0e fd 52 03 b1 f9 50 1e fb 4f 04 ab 25 28
0000020 11 e1 8e 84 66 14 a0 fb fe 59 cf 2f 60 a7 3b 07
0000040

导出共享密码-用对方的私钥和我的公钥：
```

```
[root@master1 ~]# openssl pkeyutl -derive -inkey bitkey2.der -keyform DER -peerkey bitpub1.der -peerform DER -out secret
[root@master1 ~]# cat secret1 | od -t x1
0000000 c3 ce 0e fd 52 03 b1 f9 50 1e fb 4f 04 ab 25 28
0000020 11 e1 8e 84 66 14 a0 fb fe 59 cf 2f 60 a7 3b 07
0000040
```

可见导出的共享密码相同，共 32 个字节。共享密码通常用于会话的对称加密密钥，非对称加密性能很低，不太适合用于频繁性的大量数据加密，而对称密钥加密计算性能就要高得多。比如浏览器和 Web 服务器之间在验证证书后产生共享的对称加密密钥，用于此后会话的加解密，再比如 ssh 公钥登录之后的会话也是采用对称加密密钥的。

三、管理证书、证书申请和证书吊销列表

openssl 支持的证书格式为 X.509 格式。证书管理包括证书申请、证书颁发、证书验证和证书吊销等。主要用到命令“openssl req ...”和“openssl x509 ...”。

命令“openssh req ...” 用于创建和处理 PKCS#10 格式的证书申请 (CSR)，它还可以创建自签名证书用作根 CA。其使用语法如下：

```
openssl req [-help] [-inform DER|PEM] [-outform DER|PEM] [-in filename] [-passin arg] [-out filename] [-passout arg] [-text] [-pubkey] [-noout] [-verify] [-new] [-newkey arg] [-pkeyopt opt:value] [-noenc] [-key filename|uri] [-keyform DER|PEM|P12|ENGINE] [-keyout filename] [-<digest>] [-config filename] [-section name] [-x509] [-x509v1] [-CA filename|uri] [-CAkey filename|uri] [-days n] [-set_serial n] [-newhdr] [-copy_extensions arg] [-extensions section] [-reqexts section] [-addext ext] [-precert] [-utf8] [-subj arg] [-sigopt nm:v] [-vfyopt nm:v] [-batch] [-verbose] [-nameopt option] [-rand files] [-writerand file]
```

其中参数-help、-inform、-outform、-in、-passin、-out、-passout、-text、-noout、-pkeyopt、-key、-keyform、-keyout、-<digest>、-config 前面已经介绍过。

其他参数介绍如下：

-pubkey：显示证书申请中的公钥。

-verify：检查并验证证书申请。

-new：生成新的证书申请。会提示用户输入相关字段（如城市、机构、名称等），提示的字段及其最大和最小值在配置文件和相应的参数中指定。如果省略了-key 参数，就使用配置文件中指定的信息或通过-newkey 和-pkeyopt 参数的值生成新的私钥，默认情况下为 2048 位长度的 RSA 私钥。openssl 命令的配置文件为 /etc/pki/tls/openssl.cnf。

-newkey arg：生成新的私钥并随后使用它（创建新的证书申请或在使用-x509 的情况下创建新的证书），如果同时提供了参数-key，那么-newke 参数被忽略。参数的值 arg 可为以下几种格式之一：
[rsa:]nbits - 生成大小为 nbits 的 RSA 密钥。 如果省略 nbits，即指定-newkey rsa，则使用配置文件中的 default_bits - 选项指定的默认密钥大小（如果存在），否则密钥长度为 2048。
param:file - 使用参数文件或证书文件生成密钥，算法由参数决定。
algname[:file] - 使用给定的算法 algname 生成密钥。如果给出了参数文件 file，则使用其中指定的参数，其中算法参数必须与 algname 匹配。如果未给出算法参数，则应通过-pkeyopt 指定任何必要的参数。
dsa:filename - 使用文件 filename 中的参数生成 DSA 密钥。
ec:filename - 生成 EC 密钥（可与 ECDSA 或 ECDH 算法一起使用）。
gost2001:filename - 生成 GOST R 34.10-2001 密钥（需要在配置文件中配置 gost 引擎）。 如果仅指定 gost2001，则应通过-pkeyopt 指定参数集。

-noenc：新创建的私钥没有加密保护。参数-newkey 会创建私钥。

-section name：指定要使用的节的名称， 默认是 req 节。配置文件会分若干节，每个节下面就是若干参数，使用某个节也就是使用该节下的参数。

https://www.verylon.org

-x509：生成证书（颁发证书）而不是证书申请。参数-CA 隐含了本参数。如果省略了 -in 参数，则-x509 隐含了-new 参数。如果已经使用-in 参数指定了现有的证书申请，则将其转换为证书，否则会先创建申请。除非使用 -set_serial 参数指定了序列号，否则将使用一个大的随机数作为序列号。除非使用了-copy_extensions 参数，否则不会从任何提供的证书申请文件中拷贝 X.509 扩展。可以在配置文件中指定要添加的 X.509 扩展，或者通过-extensions 和-addext 参数指定。除非给出 -x509v1，否则生成的证书为 X.509 的 3 版。

-x509v1：请求生成 X.509 的 1 版证书，有时为了向前兼容，不得不使用低版证书。本参数隐含这-x509 参数，如果通过配置文件或命令行参数额外提供了 X.509 扩展，则无论如何都会生成 3 版的 X.509 证书。

-CA filename|uri：指定用于签署新证书的“CA”证书并暗示使用 -x509。 “CA”证书的使用者名称作为颁发者名称放置在新证书中，然后使用-CAkey 参数指定的私钥进行签名。

-CAkey filename|uri：指定 CA 私钥。CA 证书中的公钥和 CA 私钥必须是合法的一对公/私钥。

-days n：结合-x509 参数，指定证书的有效期，默认是 30 天。

-set_serial n：颁发自签名证书时使用的序列号，值为十进制或十六进制（附加前缀 0x）。如果省略本参数，将使用一个大的随机数作为序列号。

-newhdr：将单词 NEW 添加到输出请求的 PEM 文件页眉和页脚行。一些软件（Netscape 证书服务器）和一些 CA 需要这个。

-copy_extensions arg：确定在使用 -x509 时应如何处理证书申请中的 X.509 扩展。如果 arg 为 none 或省略此参数，则忽略扩展。如果 arg 是 copy 或 copyall，则申请中的所有扩展都会复制到证书中。此参数的主要用途是允许证书申请者提供某些扩展的名字（例如 subjectAltName）。

-extensions section、-reqexts section：指定配置文件中的节名，允许证书（使用 -x509 时）或证书申请中包含的 X.509 扩展覆盖这些节中的同名参数。 这允许在同一配置文件中使用多个不同的节来指定用于各种目的的申请。

-addext ext：将特定扩展添加到证书（如果使用-x509）或证书申请。该参数的值为“键=值”对的形式，就像它出现在配置文件中一样。该选项可以多次给出。

-precert：在证书中添加有毒扩展，使其成为“预证书”（请参阅 RFC6962）。预证书可以提交到 CT 的日志节点中，以获得签名证书时间戳 (SCT)。 然后，在消除毒害并签署证书之前，可以将这些 SCT 作为扩展嵌入到预证书中。-precert 参数隐含了-new 参数。CT 是 Certificate Transparency 的缩写，用于透明化证书的签发记录，从而提升整体网络环境安全性的一套解决方案。任何人都可以在某个 CT 的日志节点上查询某个证书的签发详情，一旦某个证书提交到日志节点上，那么它就不可被修改了。原理和详细介绍可以参看官网。

-utf8：表名字段值为 UTF8 编码，因此无论是从控制台还是从配置文件中获取的字段值，都必须是有效的 UTF8 字符串。默认是 ASCII 编码。

-subject：打印证书申请的主题（如果使用-x509，则打印出证书主题）。

-subj arg：设置使用者名称或在处理证书请求时取代使用者名称。arg 的格式必须为 /type0=value0/type1=value1/type2=.... ，特殊字符可以通过 \（反斜杠）转义，保留空格。 arg 允许为空，这样相应的 type 不会包含在请求中。如果 arg 为/，那么 RDN 就为空。可以通过在指定集合成员的 AttributeValueAssertions (AVA) 之间放置+字符而不是/来形成多值 RDN，例如 /DC=org/DC=OpenSSL/DC=users/UID=123456+CN=John Doe 表示的 DN 为 UID=123456+CN=John Doe, DC=users,DC=OpenSSL,DC=org。RDN（relative distinguished name）就是相对专有名字，是 DN 的组成部分，即一个 DN 由多个 RDN 组成。

-sigopt nm:v：在签名操作阶段将参数传递给签名算法，这些参数的名称和值是特定于具体算法的。

https://www.verypen.org

-vfyopt nm:v：在验证操作阶段将参数传递给签名算法，这些参数的名称和值是特定于具体算法的。

-batch：非交互模式。

-verbose：打印有关正在执行的操作的额外详细信息。

-rand files：包含随机数据的一个或多个文件，文件中的数据为随机数生成器提供种子。 多个文件之间用:分隔。

-writerand file：退出时将种子数据写入指定文件。 该文件可在后续命令中使用。

看例子：

```
1. 直接产生 RSA 私钥并颁发证书（有效期 30 天）:
[root@master1 ~]# openssl req -x509 -newkey 1024 -noenc -subj /CN=wlm -keyout key.pem -out req.pem
[root@master1 ~]# ll *.pem
-rw------- 1 root root 916 Aug 16 13:49 key.pem                                          #RSA 1024 位私钥
-rw-r--r-- 1 root root 745 Aug 16 13:49 req.pem                                          #RSA 证书

2. 直接产生 EC 私钥，并颁发证书，且私钥受莫玛保护:
[root@master1 ~]# openssl req -x509 -newkey EC -pkeyopt ec_paramgen_curve:secp256k1 -subj /OU=gdpi/CN=Alice \
 -keyout key_ec.pem -out req_ec.pem
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
[root@master1 ~]# ll *.pem
-rw------- 1 root root 399 Aug 16 14:01 key_ec.pem
-rw------- 1 root root 916 Aug 16 13:57 key.pem
-rw-r--r-- 1 root root 603 Aug 16 14:01 req_ec.pem
-rw-r--r-- 1 root root 745 Aug 16 13:57 req.pem

3. 产生 CA 私钥，并自签证书:
[root@master1 ~]# openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:secp256k1 -out ca.key
[root@master1 ~]# openssl req -x509 -key ca.key -subj "/CN=myCA" -days 365 -out ca.crt
[root@master1 ~]# ll ca.*
-rw-r--r-- 1 root root 558 Aug 16 14:27 ca.crt                                           #CA 自签证书
-rw------- 1 root root 237 Aug 16 14:19 ca.key                                           #CA 私钥

4. 完整走一遍：生成私钥、生成证书申请、采用 CA 私钥签发证书:
[root@master1 ~]# openssl genpkey -algorithm RSA -quiet -out test.key                    #生成私钥
[root@master1 ~]# openssl req -new -key test.key -subj "/CN=test" -out test.csr          #生成证书申请
[root@master1 ~]# openssl req -x509 -in test.csr -CA ca.crt -CAkey ca.key -out test.crt -days 100 \
 -copy_extensions copy                                                                   #签发证书，有效期 100 天
[root@master1 ~]# ll test.*
-rw-r--r-- 1 root root  839 Aug 16 14:44 test.crt                                        #证书文件
-rw-r--r-- 1 root root  883 Aug 16 14:38 test.csr                                        #证书申请文件
-rw------- 1 root root 1704 Aug 16 14:37 test.key                                        #私钥文件


[root@master1 ~]# openssl req -in test.csr -text -verify -noout                          #查看证书申请信息
Certificate request self-signature verify OK
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: CN = test
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:c6:41:fe:03:4e:03:5e:53:c0:1f:7c:49:91:04:
                    ......
                    e4:f1
                Exponent: 65537 (0x10001)
        Attributes:
            (none)
            Requested Extensions:
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        c1:ae:0c:f1:02:00:6b:1d:8e:ab:22:ce:e5:a9:58:a6:de:76:
        ......
        7b:e8:31:d9

[root@master1 ~]# openssl x509 -in test.crt -text -noout                                 #查看证书信息
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0f:cd:3c:b1:8d:02:27:f4:b5:0f:4f:e1:08:aa:4d:b8:e5:f5:ea:36
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN = myCA
        Validity
            Not Before: Aug 16 06:44:08 2023 GMT
            Not After : Nov 24 06:44:08 2023 GMT
        Subject: CN = test
        Subject Public Key Info:
```

```
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:c6:41:fe:03:4e:03:5e:53:c0:1f:7c:49:91:04:
                    ......
                    e4:f1
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                AF:E7:22:B4:83:3F:F8:7F:A3:DC:A8:27:A2:F7:73:5F:0A:E0:2D:E8
            X509v3 Authority Key Identifier:
                97:98:34:D1:7A:88:E8:75:27:35:DC:3C:A0:21:29:F7:35:1B:89:A8
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
        30:45:02:20:51:71:3a:55:85:71:f2:69:5c:f8:11:9d:3d:fb:
        ......
        14:30:65:80:3e:b3:4d:c9:73:09:79:39:e3:15:2b:c3:59
```

证书管理命令“openssl x509 ...”的语法如下：

四、计算摘要（hash），支持各种摘要算法

使用命令“openssl dgst ... files”生成文件的摘要。

五、使用各种加密方法(cipher)加密、解密

使用命令“openssl enc ...”对称加密数据。

```
[root@master1 ~]# openssl base64 -in file.bin -out file.b64                    #二进制文件编码成 base64
[root@master1 ~]# openssl base64 -d -in file.b64 -out file.bin                 #base64 到二进制

[root@master1 ~]# openssl enc -aes128 -pbkdf2 -in file.b64 -out file.enc       #加密文件
enter AES-128-CBC encryption password:
Verifying - enter AES-128-CBC encryption password:

[root@master1 ~]# openssl enc -aes128 -pbkdf2 -d -in file.enc -out file.txt -pass pass:A1b2C3   #解
密
```

六、测试 SSL 和 TLS 连接

openssl s_server -www -key key.pem -cert req.pem，启动 web 服务，可通过 https:/ip:4433 访问。

七、对 email 签名或加密（S/MIME）格式

采用命令“openssl smime ...”命令。

https://www.veryopen.org

八、生成和验证时间戳

命令“openssl ts ...”

§10.4.2 gpg2 命令

下面我们在 Linux 做签名的实验：

1. 产生密钥对。这个命令建议在电脑主控台上执行，前三个问题直接会取默认值：ECC（签名和加密）、Curve 25519、0 = 密钥永不过期，然后输入用户（如Abraham II）和邮件地址（如abrahamii@walletoffline.com），确认参数无误后输入私钥保护密码：

```
gpg --full-generate-key
……
pub   ed25519 2025-06-14 [SC]
      0871A5938FE377B7D3CBA8E2AE8D057131094963                                    #指纹
uid                      Abraham II <abrahamii@walletoffline.com>
sub   cv25519 2025-06-14 [E]
```

密钥指纹和公钥对外公布，指纹一般公布在公开网站上，并且以四个字符为一组以便于他人核对，如 0871 A593 8FE3 77B7 D3CB A8E2 AE8D 0571 3109 4963。

2. 管理密钥：

```
gpg -k                                                                        #列出密钥对
……
--------------------------------
pub   ed25519 2025-06-14 [SC]
      0871A5938FE377B7D3CBA8E2AE8D057131094963
uid           [ultimate] Abraham II <abrahamii@walletoffline.com>
sub   cv25519 2025-06-14 [E]

#导出公钥，gpg -a --export <邮件或者指纹> > public_key.asc。如：
gpg -a --export 0871A5938FE377B7D3CBA8E2AE8D057131094963 > public_key.asc

gpg --export-secret-keys --armor <邮件或者指纹> > privatekey.asc                  #导出私钥
gpg --import privatekey.asc   ----------------------------------------------#在其他电脑上导入密钥
gpg --delete-secret-and-public-key <邮件或者指纹>                             #删除密钥对
gpg --import publicKey.asc                                                    #导入公钥
```

3. 签名（假如对 file.txt 文件签名）：

```
gpg --detach-sign -a -o file.txt.sig file.txt                  #输入密码保护密码（如果有）
gpg --verify file.txt.sig file.txt                                #自己验证签名
```

4. 其他人验证签名：

```
gpg --import publicKey.asc
gpg --verify file.txt.sig file.txt
gpg: Signature made Sat Jun 14 15:50:25 2025 CST
gpg:                using EDDSA key 0871A5938FE377B7D3CBA8E2AE8D057131094963
gpg: Good signature from "Abraham II <abrahamii@walletoffline.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 0871 A593 8FE3 77B7 D3CB  A8E2 AE8D 0571 3109 4963
```

图 10-13 就是作者开发的离线加密币钱包项目（https://www.walletoffline.com）的下载页面。

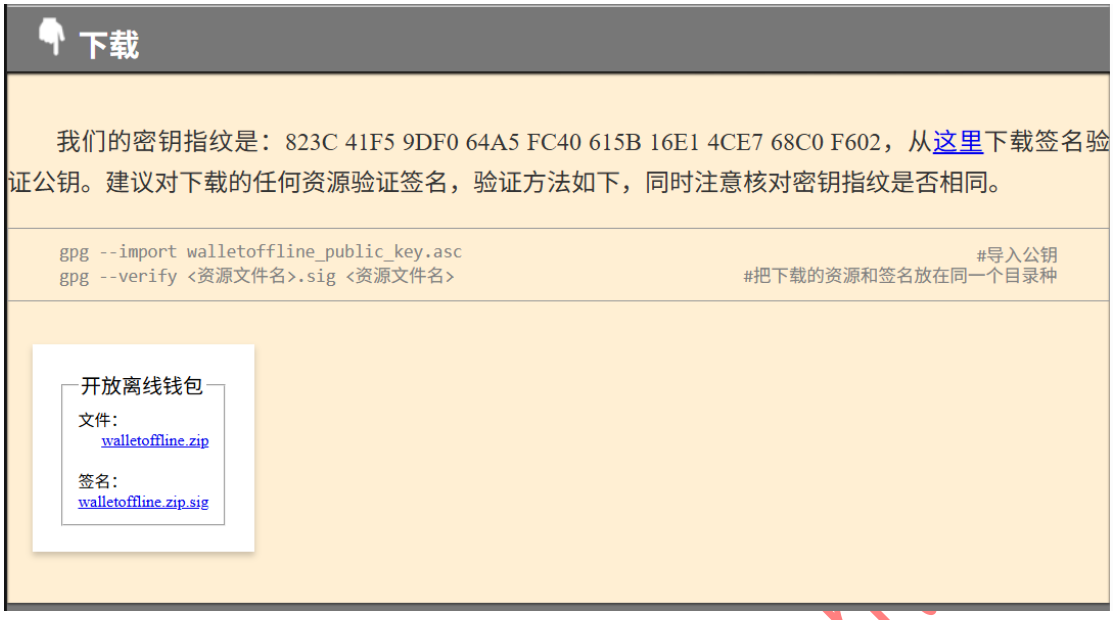


图 10-13 把公钥和密钥指纹公布在下载页面上

https://www.veryopen

第11章 etcd 数据库



etcd 是一个分布式可靠的“键/值”数据库，被其他分布式系统用来存储关键的集群配置信息，它的对标产品不是 redis，而是 zookeeper，redis 用来存储业务数据，而 etcd 用来存储配置数据（集群的元数据）。etcd 比 zookeeper 诞生晚，却是后起之秀。etcd 具有四个最显著的优势：

1. 使用简单：定义明确、面向用户的 API (gRPC)。
2. 安全性高：带有可选客户端证书身份验证的自动 TLS。
3. 访问快速：基准测试达到每秒 10,000 次写。
4. 高可靠性：使用 Raft 分部式共识算法。

etcd 使用 Go 编写，目前被广泛应用于各种分布式系统中，如 kubernetes、ROOK、CoreDNS、M3 等等，还被云原生计算基金会 (CNCF)收录为明星项目。

etcd 引入了 watch 机制，允许监视由客户端指定的某些键，当这些键的值发生改变时会向客户端发送事件。类似“如果有人修改了键 age 的值，请通知我啦！”kubernetes 就充分利用了 etcd 的 watch 机制。

https://www.verypeo

§11.1 安装

1. 安装最新版本的 etcd：

```
把最新版本号保存在 ETCD_RELEASE 变量中：
[root@master1 ~]# ETCD_RELEASE=$(curl -s https://api.github.com/repos/etcd-io/etcd/releases/latest|grep tag_name | cut -d '"' -f 4)
[root@master1 ~]# echo $ETCD_RELEASE
v3.6.5
```

上面第一条命令是从网上获取 etcd 的最新版本号，如果没法访问网站，就手工设置 ETCD_RELEASE 变量的值为 v3.7.2 即可（直接执行命令 ETCD_RELEAS=v3.7.2）。

```
下载最新版本的 etcd:
[root@master1 ~]# wget https://github.com/etcd-io/etcd/releases/download/${ETCD_RELEASE}/etcd-${ETCD_RELEASE}-linux-amd64.tar.gz
[root@master1 ~]# tar -xf etcd-${ETCD_RELEASE}-linux-amd64.tar.gz
[root@master1 ~]# mv etcd-${ETCD_RELEASE}-linux-amd64/etcd* /usr/local/bin
[root@master1 ~]# chown root:root /usr/local/bin/etcd*
```

安装的过程就是下载、解压解包，并把 etcd、etcdctl、etcdutl 三个可执行程序移到/usr/local/bin 目录中。etcd 是服务程序、etcdctl 为客户端命令、etcdutl 为命令行管理工具，是对 etcdctl 功能的补充（比如恢复数据）。由于/usr/local/bin 目录在 PATH 环境变量中，因此以后可以直接输入 etcdctl 命令而不用带路径。

查看三个程序的版本号：

```
[root@master1 ~]# etcd --version
etcd Version: 3.7.2
Git SHA: 68c065e
Go Version: go1.26.8
Go OS/Arch: linux/amd64

[root@master1 ~]# etcdctl version
etcdctl version: 3.7.2
API version: 3.7
[root@master1 ~]# etcdutl version
etcdutl version: 3.7.2
API version: 3.7
```

2. 把启动 etcd 的任务做成 systemd 服务：

```
创建用户 etcd 和组 etcd:
[root@master1 ~]# groupadd --system etcd
[root@master1 ~]# useradd -s /sbin/nologin --system -g etcd etcd

[root@master1 ~]# mkdir -p /var/lib/etcd
[root@master1 ~]# chown -R etcd: /var/lib/etcd
[root@master1 ~]# chmod 0755 /var/lib/etcd/

新建 systemd 服务配置文件:
[root@master1 ~]# vim /etc/systemd/system/etcd.service
[Unit]
Description=etcd key-value store
After=network.target

[Service]
User=etcd  ------------------------------------------------------------------------------------------#用 etcd 用户启动 etcd 服务
Type=notify
Environment=ETCD_DATA_DIR=/var/lib/etcd ------------------------------------------------------------ #etcd 保存数据的目录
Environment=ETCD_NAME=etcd01                                                                          #etcd 的节点名称
ExecStart=/usr/local/bin/etcd
Restart=always
RestartSec=10s
LimitNOFILE=40000

[Install]
WantedBy=multi-user.target

[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl enable etcd.service                                  #激活服务，下次开机自动启动 etcd
Created symlink /etc/systemd/system/multi-user.target.wants/etcd.service → /etc/systemd/system/etcd.service.
```

确保 selinux 已经禁用（用命令 sestatus 查看），然后启动 etcd 服务：

```
[root@master1 ~]# systemctl start etcd.service

[root@master1 ~]# systemctl status etcd
● etcd.service - etcd key-value store
     Loaded: loaded (/etc/systemd/system/etcd.service; enabled; preset: disabled)
     Active: active (running) since Wed 2026-09-30 11:29:04 CST; 5s ago
   Main PID: 6135 (etcd)
      Tasks: 10 (limit: 22916)
     Memory: 12.7M (peak: 13.3M)
        CPU: 65ms
     CGroup: /system.slice/etcd.service
             └─6135 /usr/local/bin/etcd

Aug 07 11:57:15 worker3 etcd[3137599]: {"level":"info","ts":"2023-08- ……
```

如果启动失败，请检查有没有关闭 SELinux 安全模块（参考 2.3 小节的相关部分去禁用 SELinux）。

```
[root@master1 ~]# ss -tunelp | grep etcd
tcp   LISTEN 0      4096       127.0.0.1:2380       0.0.0.0:*    users:(("etcd",pid=3137599,fd=7)) uid:985 ino:3364387 sk:3005 cgroup:/system.slice/etcd.service <->
tcp   LISTEN 0      4096       127.0.0.1:2379       0.0.0.0:*    users:(("etcd",pid=3137599,fd=8)) uid:985 ino:3364390 sk:3006 cgroup:/system.slice/etcd.service <->

[root@master1 ~]# etcdctl member list                                                                 #查询集群中的成员节点
8e9e05c52164694d, started, d32189a53fe84d9396ad83386ac9d1fa, http://localhost:2380, http://localhost:2379, false
```

```
[root@master1 ~]# etcdctl endpoint health                                                    #查询集群健康状况
127.0.0.1:2379 is healthy: successfully committed proposal: took = 10.951847ms

[root@worker3 ~]# etcdctl endpoint status                                                    #查询集群状态
127.0.0.1:2379, 8e9e05c52164694d, 3.7.2, 3.7.0, 25 kB, 16 kB, 34%, 2.1 GB, true, false, 2, 5, 5, , , false
```

如果启动 etcd 失败，请执行命令“journalctl -b --no-pager -u etcd”查看详细的错误日志。

etcd 监听两个端口，2379/tcp 监听客户端请求，2380/tcp 监听集群内其他节点的请求。在生产环境中，一般至少采用三个节点部署 etcd。

3．开启防火墙，让 etcd 服务通过防火墙：

```
[root@master1 ~]# firewall-cmd --add-service={etcd-server,etcd-client} --permanent
success
[root@master1 ~]# firewall-cmd --reload
success
```

4．先简单用用 etcd：

```
[root@master1 ~]# etcdctl put welcome "Hello World"                                  #写入 welcome="Hello World"
OK
[root@master1 ~]# etcdctl put class3/2021123/name "Alice"
OK
[root@master1 ~]# etcdctl put class3/2021123/age 21
OK
[root@master1 ~]# etcdctl get welcome ------------------------------------------------------  #读取 welcome 键的值
welcome
Hello World
[root@master1 ~]# etcdctl get class3/2021123/name
class3/2021123/name
Alice
[root@master1 ~]# etcdctl del welcome ------------------------------------------------------------------------  #删除键 welcome
1
[root@master1 ~]# etcdctl get welcome                                                #读不到了
```

§11.2 增加证书

etcd 集群中存在两种类型的通信，一是与客户端的通信（主要是南北流量），二是与其他节点的通信（主要是东西流量），这些通信允许跨网络（LAN、WAN），所以最好采用证书加密和签名以增强安全性。

1．客户端证书

客户端证书针对客户端与 etcd 服务之间的通信，为南北流量。

(1)．准备目录和配置文件：

```
[root@master1 ~]# mkdir ~/etcd
[root@master1 ~]# cd ~/etcd
[root@master1 etcd]# vim openssl.conf
[req]
req_extensions = v3_req
```

```
distinguished_name = req_distinguished_name

[req_distinguished_name]

[v3_req]
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth, clientAuth
subjectAltName = @alt_names

[alt_names]
DNS.1 = etcd01
DNS.2 = etcd02
DNS.3 = etcd03
IP.1 = 127.0.0.1
IP.2 = 10.10.10.11
IP.3 = 10.10.10.31
IP.4 = 10.10.10.32
```

注意：集群中的节点名称和 Ip 地址都要加入，否则节点没法使用节点证书的。

我们采用 openssl 命令生成证书，涉及一些常用的参数放在 openssl.conf 文件中，省得每次运行 openssl 命令都要输入长长的参数。

(2)．产生 CA 私钥和 CA 证书：

生成 2048 位的 CA 的私钥：

```
[root@master1 etcd]# openssl genrsa -out ca.key 2048
[root@master1 etcd]# ll
total 4
-rw------- 1 root root 1704 Aug  7 15:41 ca.key
```

从 CA 私钥导出公钥并自签，CA 证书有效期 10 年：

```
[root@master1 etcd]# openssl req -x509 -new -noenc -key ca.key -subj "/CN=Myself-CA" -days 3650 -out ca.crt
[root@master1 etcd]# ll
total 8
-rw-r--r-- 1 root root 1107 Aug  7 15:45 ca.crt ---------------------------------------------------------------------- #CA 证书
-rw------- 1 root root 1704 Aug  7 15:41 ca.key                                                                      #CA 私钥
-rw-r--r-- 1 root root  298 Aug  7 16:41 openssl.conf
```

(3)．产生客户端公、私钥和证书：

产生 2048 位的客户端私钥：

```
[root@master1 etcd]# openssl genrsa -out client.key 2048
```

产生证书申请文件 client.csr：

```
[root@master1 etcd]# openssl req -new -key client.key -subj "/CN=etcd-client" -out client.csr -config openssl.conf
```

采用 CA 私钥对客户端公钥签名并产生证书，证书有效期 1 年：

```
[root@master1 etcd]# openssl x509 -req -sha256 -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
 -out client.crt -days 365 -extensions v3_req  -extfile openssl.conf
Certificate request self-signature ok
subject=CN = etcd-client

[root@master1 etcd]# ll
total 24
-rw-r--r-- 1 root root 1107 Aug  7 15:45 ca.crt                                                                      #CA 公钥
-rw------- 1 root root 1704 Aug  7 15:41 ca.key ---------------------------------------------------------------------#CA 私钥
-rw-r--r-- 1 root root   41 Aug  7 15:48 ca.srl
-rw-r--r-- 1 root root  993 Aug  7 15:48 client.crt                                                                 #客户端证书
-rw-r--r-- 1 root root  895 Aug  7 15:46 client.csr---------------------------------------------------------- #证书申请文件
-rw------- 1 root root 1708 Aug  7 15:45 client.key                                                                 #客户端私钥
-rw-r--r-- 1 root root  298 Aug  7 16:41 openssl.conf

[root@master1 etcd]# mkdir -p /etc/etcd/pki
[root@master1 etcd]# cp ca.crt client.crt client.key /etc/etcd/pki/
[root@master1 etcd]# cd
[root@master1 ~]# ll /etc/etcd/pki
total 16
-rw-r--r-- 1 root root 1107 Aug  7 16:10 ca.crt
```

https://www.yuque.com

```
-rw------- 1 root root 1704 Aug  7 16:10 ca.key
-rw-r--r-- 1 root root  993 Aug  7 16:10 client.crt
-rw------- 1 root root 1708 Aug  7 16:10 client.key

[root@master1 ~]# chgrp -R etcd /etc/etcd
[root@master1 ~]# chmod g+r /etc/etcd/pki/client.key
```

(4)．启动 etcd 服务：

```
[root@master1 ~]# systemctl stop etcd                                                    #先关闭 etcd 服务
[root@master1 ~]# etcd --name etcd01 -client-cert-auth --trusted-ca-file=/etc/etcd/pki/ca.crt \
 --cert-file=/etc/etcd/pki/client.crt --key-file=/etc/etcd/pki/client.key \
 --listen-client-urls 'https://10.10.10.11:2379,https://127.0.0.1:2379' \
 --advertise-client-urls https://10.10.10.11:2380
```

参数-client-cert-auth 表明启用客户端证书，下次用户操纵数据库时不许提供证书才行。也可以把这些参数放在配置文件中，然后启动 etcd.service 服务时指定配置文件即可。先按 Ctrl+c 终止 etcd 服务，然后：

```
[root@master1 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd01"
ETCD_DATA_DIR="/var/lib/etcd"

#配置与客户端的通信:
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.11:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.11:2379"
ETCD_CLIENT_CERT_AUTH=true
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置 etcd 集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
#ETCD_INITIAL_CLUSTER_STATE="new"| "existing"
```

ETCD_NAME：定义当前节点名称，集群中的节点名称不能相同，节点名不一定与主机名相同。

ETCD_DATA_DIR：定义 etcd 保存数据的目录。

ETCD_LISTEN_CLIENT_URLS：etcd 服务监听的 ip 和端口，客户端（etcdctl、etcdutil、curl 等）通过此 ip 和端口向 etcd 服务请求数据，值可为用逗号分隔的多个 url，如"http://192.168.3.103:2379,http://127.0.0.1:2379"。如果要在一个端口上监听所有网络接口，请使用 0.0.0.0 为监听 ip 地址，这时 127.0.0.1 以及所有网卡的 ip 都被监听，查看被监听的端口时（ss -tnlp），地址会用*表示。

ETCD_ADVERTISE_CLIENT_URLS：向外通告本机的 url，告诉非本机上的客户端："喂，记住啊，你们以后用这些 url 跟我通信。"

很多初学者对 etcd 的 --listen-client-urls 、--advertise-client-urls 参数（对应 ETCD_LISTEN_CLIENT_URLS、ETCD_ADVERTISE_CLIENT_URLS）不理解，我这里举个例子说明你就很清楚了：Alice 是公司的电话接线员，她的办公桌上有很多部电话机，每个电话机对应不同的电话号码，这些电话号码就用--listen-client-urls 参数登记了（或者说保存在 ETCD_LISTEN_CLIENT_URLS 变量中），Alice 平时就监听这些电话机，但是分布在世界各地的客户不知道 Alice 所在公司的业务电话号码，所以 Alice 的公司还需要向外通告部分电话号码，需要通告的电话号码就用 --advertise-client-urls 登记（或者说保存在 ETCD_ADVERTISE_CLIENT_URLS 变量中）。通告的电话号码必须是监听的电话号码的子集，即 ETCD_ADVERTISE_CLIENT_URLS 变量的值为 ETCD_LISTEN_CLIENT_URLS 的子集，这也很好理解，因为 Alice 监听的电话号码有的是公司内部员工才能打的（类似 127.0.0.1），也有的可能是领导专线号码。

如果设置了 ETCD_LISTEN_CLIENT_URLS，那一定要设置 ETCD_ADVERTISE_CLIENT_URLS。

ETCD_CLIENT_CERT_AUTH：true-启用客户端证书，false-不启用。

ETCD_TRUSTED_CA_FILE：指明 CA 证书文件。

ETCD_CERT_FILE：指明客户端证书文件。

ETCD_KEY_FILE：指明客户端私钥文件。以后客户端每次请求都必须同时提供 CA 证书文件、客户端证书文件和客户端私钥文件。

ETCD_INITIAL_CLUSTER：集群初始化时包含哪些节点，用“节点名=http|https://ip:2380”表示一个节点，多个节点之间用逗号分隔，如：

```
ETCD_INITIAL_CLUSTER="etcd01=http://10.10.10.11:2380,etcd02=http://10.10.10.22:2380,etcd03=http://10.10.10.33:2380"
```

如果启用了节点证书，就要采用 https 协议了，如 etcd01=https://10.10.10.11:2380。

ETCD_INITIAL_CLUSTER_TOKEN：集群名称。同一个集群内的所有节点，此变量的值必须相同。

ETCD_INITIAL_CLUSTER_STATE：初始化集群还是加入已经存在的集群，取值有"new"、"existing"，默认是"new"。在初始化或加入集群之后日常启动 etcd 服务时，本变量被忽略。/var/lib/etcd 目录为空表示要初始化集群，否则表示日常启动 etcd 服务。所以如果要重置一个集群，请执行命令“rm -rf /var/lib/etcd/*”清空目录/var/lib/etcd/即可。

修改服务配置文件，指明环境变量文件：

```
[root@master1 ~]# vim /etc/systemd/system/etcd.service
[Unit]
Description=etcd key-value store
After=network.target

[Service]
User=etcd
Type=notify
EnvironmentFile=/etc/etcd/etcd.conf
ExecStart=/usr/local/bin/etcd
Restart=always
RestartSec=10s
LimitNOFILE=40000

[Install]
WantedBy=multi-user.target

[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl restart etcd
```

如果启动报错，查看具体原因：

```
[root@master1 ~]# journalctl -xeu etcd.service
```

现在 etcd 服务要求客户端必须提供证书才允许访问。如：

```
携带证书访问 etcd 服务——写入 hello="Hello Etcd!":
[root@wmaster1 ~]# etcdctl --cacert=/etc/etcd/pki/ca.crt --cert=/etc/etcd/pki/client.crt \
 --key=/etc/etcd/pki/client.key --endpoints=https://10.10.10.11:2379 put hello "Hello Etcd!"
OK

携带证书访问 etcd 服务——读取键 hello:
[root@master1 ~]# etcdctl --cacert=/etc/etcd/pki/ca.crt --cert=/etc/etcd/pki/client.crt \
 --key=/etc/etcd/pki/client.key --endpoints=https://10.10.10.11:2379 get hello
hello
Hello Etcd!
```

为了简化 etcdctl 命令的证书参数，建议把它们放到环境变量中：

```
[root@master1 ~]# cat <<EOF >>~/.bashrc
export ETCDCTL_ENDPOINTS=https://10.10.10.11:2379,https://10.10.10.31:2379,https://10.10.10.32:2379
export ETCDCTL_CACERT=/etc/etcd/pki/ca.crt
export ETCDCTL_CERT=/etc/etcd/pki/client.crt
export ETCDCTL_KEY=/etc/etcd/pki/client.key
EOF

[root@master1 ~]# . ~/.bashrc                                    #输出文件中定义的环境变量
```

此命令就是在~/.bashrc 文件的末尾增加 export … 这些行。注意：集群中的全部节点的 EndPoint 都要加在 ETCDCTL_ENDPOINTS 环境变量中，本例中只有三台。

这样就可以直接使用类似 etcdctl get …直接操纵数据库了，如：

```
[root@master1 ~]# etcdctl put pass "a12345678!"
OK

[root@master1 ~]# etcdctl get pass
pass
a12345678
```

(5)．在另一台电脑上使用 etcd 服务（比如在 worker1 上操作）：

```
从 master1 上下载客户端：
[root@worker1 ~]# scp master1:/usr/local/bin/etcdctl /usr/local/bin/

从 master1 上下载客户端证书：
[root@worker1 ~]# mkdir -p /etc/etcd/pki
[root@worker1 ~]# scp master1:/etc/etcd/pki/ca.crt /etc/etcd/pki/
[root@worker1 ~]# scp master1:/etc/etcd/pki/client.* /etc/etcd/pki/
[root@worker1 ~]# ll /etc/etcd/pki/
total 12
-rw-r--r-- 1 root root 1107 Aug  7 17:06 ca.crt
-rw-r--r-- 1 root root 1237 Aug  7 17:06 client.crt
-rw------- 1 root root 1708 Aug  7 17:06 client.key

携带证书访问 etcd 服务——写入 hello="Hello Etcd!"：
[root@worker1 ~]# etcdctl --cacert=/etc/etcd/pki/ca.crt --cert=/etc/etcd/pki/client.crt \
 --key=/etc/etcd/pki/client.key --endpoints=https://10.10.10.11:2379 put hello "Hello Etcd!"
OK

携带证书访问 etcd 服务——读取键 hello：
[root@worker1 ~]# etcdctl --cacert=/etc/etcd/pki/ca.crt --cert=/etc/etcd/pki/client.crt \
 --key=/etc/etcd/pki/client.key --endpoints=https://10.10.10.11:2379 get hello
hello
Hello Etcd!
```

建议设置下面的环境变量，这样每次使用 etcdctl 就不用输入长长的参数了：

```
[root@worker1 ~]# export ETCDCTL_ENDPOINTS=https://10.10.10.11:2379
[root@worker1 ~]# export ETCDCTL_CACERT=/etc/etcd/pki/ca.crt
[root@worker1 ~]# export ETCDCTL_CERT=/etc/etcd/pki/client.crt
[root@worker1 ~]# export ETCDCTL_KEY=/etc/etcd/pki/client.key

[root@worker1 ~]# etcdctl put name "Alice"  ---------- #写入 name="Alice"
OK
[root@worker1 ~]# etcdctl get name                    #读键 name
name
Alice
```

这样设置的环境变量是临时的，重新登录就没有了，建议追加到家目录中的.bashrc 文件的末尾：

```
[root@worker1 ~]# cat <<EOF >>~/.bashrc
export ETCDCTL_ENDPOINTS=https://10.10.10.11:2379,https://10.10.10.31:2379,https://10.10.10.32:2379
export ETCDCTL_CACERT=/etc/etcd/pki/ca.crt
export ETCDCTL_CERT=/etc/etcd/pki/client.crt
export ETCDCTL_KEY=/etc/etcd/pki/client.key
EOF
```

然后输出里面的环境变量：

```
[root@worker1 ~]# source ~/.bashrc
```

好了，下次登录，环境变量就自动生效了。

§11.3 创建 etcd 集群

迄今为止，我们在 master1 安装了 etcd 服务，并且启用了客户端证书，这样保证了客户端与 etcd 服务通信的安全，但是可靠性不行，因为只要 master1 故障，etcd 就不可用！而且很可能数据也会丢失!!

假设集群的节点数目为 n，那么最多允许 x 个节点故障，x 与 n 的关系为：$x = \lfloor (n-1)/2 \rfloor$。现在假设 n=3，那么由公式得到 x=1，即允许一个节点故障，又假如 n=9，可计算得到 x=4。一般 n 不要为偶数。

接下来再增加两个节点，再在 worker1 和 worker2 上各跑一个 etcd 服务，这样就组成了一个集群——由 master1、worker1 和 worker2 三台电脑组成的三节点集群。在集群中，每时每刻只有一个节点是领导（Leader），其他节点都是随从（Follower），领导节点处理所有客户端交互和日志复制等。当领导节点故障时，成员节点转换为候选节点（Candidate），候选节点参与投票，通过选举算法选出新的领导节点，没被选中的候选节点又转换为随从节点。

再次查查现在的 etcd 集群中的节点：

```
[root@master1 ~]# etcdctl member list --write-out=table
+------------------+---------+--------+-------------------------+-------------------------+------------+
|        ID        | STATUS  |  NAME  |       PEER ADDRS        |       CLIENT ADDRS      | IS LEARNER |
+------------------+---------+--------+-------------------------+-------------------------+------------+
| 36993bb67a1292d7 | started | etcd01 | https://10.10.10.11:2380 | https://10.10.10.11:2379 |      false |
+------------------+---------+--------+-------------------------+-------------------------+------------+
```

目前集群中只有一个节点。

www.veryopen.org

§11.3.1 启用节点证书

多节点集群，节点之间通信会跨网络，所以通信安全就显得特别重要，我们需要启用节点之间通信的证书。

1. 创建节点证书：

```
[root@master1 ~]# cd etcd
[root@master1 etcd]# ll
total 28
-rw-r--r-- 1 root root 1107 Aug  7 15:45 ca.crt
-rw------- 1 root root 1704 Aug  7 15:41 ca.key
-rw-r--r-- 1 root root   41 Aug  7 17:06 ca.srl
-rw-r--r-- 1 root root 1237 Aug  7 17:06 client.crt
-rw-r--r-- 1 root root 1062 Aug  7 17:06 client.csr
-rw------- 1 root root 1708 Aug  7 15:45 client.key
drwx------ 3 root root   20 Aug  7 17:06 etcd01.etcd
-rw-r--r-- 1 root root  364 Aug  7 17:05 openssl.conf

[root@master1 etcd]# more openssl.conf
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name

[req_distinguished_name]

[v3_req]
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth, clientAuth
subjectAltName = @alt_names

[alt_names]
DNS.1 = etcd01
DNS.2 = etcd02
DNS.3 = etcd03
IP.1 = 127.0.0.1
IP.2 = 10.10.10.11
IP.3 = 10.10.10.31
IP.4 = 10.10.10.32
```

注意：结群中节点的 Ip 地址和节点名称都要加入配置文件中。

```
生成私钥文件 peer.key:
[root@master1 etcd]# openssl genrsa -out peer.key 2048
```

```
生成证书申请文件 peer.csr:
[root@master1 etcd]# openssl req -new -key peer.key -subj "/CN=etcd-peer" -out peer.csr -config openssl.conf

采用 CA 私钥颁发证书，有效期 1 年:
[root@master1 etcd]# openssl x509 -req -sha256 -in peer.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
 -out peer.crt -days 365 -extensions v3_req  -extfile openssl.conf
Certificate request self-signature ok
subject=CN = etcd-peer

[root@master1 etcd]# ll
total 40
-rw-r--r-- 1 root root 1107 Aug  7 15:45 ca.crt
-rw------- 1 root root 1704 Aug  7 15:41 ca.key
-rw-r--r-- 1 root root   41 Aug  8 10:46 ca.srl
-rw-r--r-- 1 root root 1237 Aug  7 17:06 client.crt
-rw-r--r-- 1 root root 1062 Aug  7 17:06 client.csr
-rw------- 1 root root 1708 Aug  7 15:45 client.key
drwx------ 3 root root   20 Aug  7 17:06 etcd01.etcd
-rw-r--r-- 1 root root  364 Aug  7 17:05 openssl.conf
-rw-r--r-- 1 root root 1233 Aug  8 10:46 peer.crt ------------------------------------------------------------------ #节点证书
-rw-r--r-- 1 root root 1058 Aug  8 10:45 peer.csr
-rw------- 1 root root 1704 Aug  8 10:45 peer.key ------------------------------------------------------------------ #节点私钥

[root@master1 etcd]# cp peer.crt peer.key /etc/etcd/pki/
[root@master1 etcd]# chmod g+r /etc/etcd/pki/peer.key
[root@master1 etcd]# chgrp etcd /etc/etcd/pki/peer.*
[root@master1 etcd]# ll /etc/etcd/pki/
total 24
-rw-r--r-- 1 root etcd 1107 Aug  7 17:06 ca.crt
-rw------- 1 root etcd 1704 Aug  7 17:06 ca.key
-rw-r--r-- 1 root etcd 1237 Aug  7 17:06 client.crt
-rw-r----- 1 root etcd 1708 Aug  7 17:06 client.key
-rw-r--r-- 1 root etcd 1233 Aug  8 10:48 peer.crt
-rw-r----- 1 root etcd 1704 Aug  8 10:48 peer.key

[root@master1 etcd]# cd
```

2．修改 etcd 服务的配置文件，加入节点证书：

```
[root@master1 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd01"
ETCD_DATA_DIR="/var/lib/etcd"

#配置客户端与 etcd 服务之间的通信:
ETCD_CLIENT_CERT_AUTH=true
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.11:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.11:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信:
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.11:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.11:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="new"

[root@master1 ~]# systemctl stop etcd
[root@master1 ~]# rm -rf /var/lib/etcd/* ------------------------------------- #删除之前的数据，这样促使 etcd 启动时初始化集群
[root@master1 ~]# chmod 700 /var/lib/etcd
[root@master1 ~]# systemctl start etcd
```

如果启动 etcd 失败，请执行命令“journalctl -xeu etcd.service”查看详细的错误日志，或者查看日通日志 /var/log/messages。

ETCD_PEER_CLIENT_CERT_AUTH：同时开启客户端和节点证书，默认值'false'。

ETCD_LISTEN_PEER_URLS：监听哪些 ip 和端口，从这些 url 可响应集群中其他成员的请求。如果启用了证书，就采用 https 协议，否则采用 http 协议。

ETCD_INITIAL_ADVERTISE_PEER_URLS：需要对外通告的 url，等于告诉其他节点“以后你们通过这些 url 可以找到我!”本变量的值只能是 ETCD_LISTEN_PEER_URLS 变量的子集。

ETCD_LISTEN_PEER_URLS 和 ETCD_INITIAL_ADVERTISE_PEER_URLS 两个变量的理解，可参考 ETCD_LISTEN_CLIENT_URLS 和 ETCD_ADVERTISE_CLIENT_URLS 的理解（在上一小节中）。

ETCD_PEER_TRUSTED_CA_FILE、ETCD_PEER_KEY_FILE、ETCD_KEY_FILE 分别指定 CA 证书文件、节点证书文件和节点私钥文件。集群中的所有节点需要相同的证书和私钥。

2．看看集群中的节点：

```
[root@master1 ~]# etcdctl member list --write-out=table
+------------------+---------+--------+-------------------------+-------------------------+------------+
|        ID        | STATUS  |  NAME  |       PEER ADDRS        |       CLIENT ADDRS      | IS LEARNER |
+------------------+---------+--------+-------------------------+-------------------------+------------+
| 36993bb67a1292d7 | started | etcd01 | https://10.10.10.11:2380 | https://10.10.10.11:2379 |      false |
+------------------+---------+--------+-------------------------+-------------------------+------------+

[root@master1 ~]# etcdctl endpoint status --write-out=table
+-------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
|        ENDPOINT         |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+-------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
| https://10.10.10.11:2379 | 36993bb67a1292d7 |   3.5.9 |   20 kB |      true |      false |         2 |          4 |                  4 |        |
+-------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
```

上面第二条命令因链接 worker1 和 worker2 超时错误，不管他。发现只有一个节点 etcd01，它监听 https://10.10.10.11:2380 以等待其他节点的请求，监听 https://10.10.10.11:2379 以等待客户端的请求。

§11.3.2 加入 worker1 节点

1．准备新节点

创建用户 etcd 和组 etcd：

```
[root@worker1 ~]# groupadd --system etcd
[root@worker1 ~]# useradd -s /sbin/nologin --system -g etcd etcd

[root@worker1 ~]# id etcd
uid=991(etcd) gid=991(etcd) groups=991(etcd)
```

准备数据库目录：

```
[root@worker1 ~]# mkdir -p /var/lib/etcd
[root@worker1 ~]# chown -R etcd: /var/lib/etcd
[root@worker1 ~]# chmod 0700 /var/lib/etcd

[root@worker1 ~]# ip a
……
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:3c:34:be brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.31/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe3c:34be/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

把 etcd 书库程序从 master1 上拷贝过来：

```
[root@worker1 ~]# scp master1:/usr/local/bin/* /usr/local/bin/
[root@worker1 ~]# ll /usr/local/bin/
total 52336
-rwxr-xr-x 1 root root 22474752 Aug  8 07:44 etcd
-rwxr-xr-x 1 root root 16998400 Aug  8 07:44 etcdctl
-rwxr-xr-x 1 root root 14118912 Aug  8 07:44 etcdutl
```

从 master1 上把证书文件拷贝过来：

https://www.yeryopen.com

```
[root@worker1 ~]# scp -r master1:/etc/etcd /etc/
[root@worker1 ~]# rm -rf /etc/etcd/etcd01.etcd
[root@worker1 ~]# tree /etc/etcd/
/etc/etcd/
├── etcd.conf
└── pki
    ├── ca.crt
    ├── ca.key
    ├── client.crt
    ├── client.key
    ├── peer.crt
    └── peer.key

1 directory, 7 files

[root@worker1 ~]# chmod g+r /etc/etcd/pki/{peer.key,client.key}
[root@worker1 ~]# chgrp -R etcd /etc/etcd
[root@worker1 ~]# ll /etc/etcd/pki/
total 24
-rw-r--r-- 1 root etcd 1107 Aug  8 16:45 ca.crt
-rw------- 1 root etcd 1704 Aug  8 16:45 ca.key
-rw-r--r-- 1 root etcd 1237 Aug  8 16:45 client.crt
-rw-r----- 1 root etcd 1708 Aug  8 16:45 client.key
-rw-r--r-- 1 root etcd 1233 Aug  8 16:45 peer.crt
-rw-r----- 1 root etcd 1704 Aug  8 16:45 peer.key
```

准备 etcd 服务文件：

```
[root@worker1 ~]# scp master1:/etc/systemd/system/etcd.service /etc/systemd/system/
[root@worker1 ~]# more /etc/systemd/system/etcd.service
[Unit]
Description=etcd key-value store
After=network.target

[Service]
User=etcd
Type=notify
EnvironmentFile=/etc/etcd/etcd.conf
ExecStart=/usr/local/bin/etcd
Restart=always
RestartSec=10s
LimitNOFILE=40000

[Install]
WantedBy=multi-user.target

[root@worker1 ~]# systemctl enable etcd
Created symlink /etc/systemd/system/multi-user.target.wants/etcd.service → /etc/systemd/system/etcd.service.
```

```
[root@worker1 ~]# scp master1:.bashrc .
root@master1's password:
.bashrc                                                                   100%  611   555.7KB/s   00:00
[root@worker1 ~]# . .bashrc
[root@worker1 ~]# env|grep ETCD
ETCDCTL_ENDPOINTS=https://10.10.10.11:2379,https://10.10.10.31:2379,https://10.10.10.32:2379
ETCDCTL_KEY=/etc/etcd/pki/client.key
ETCDCTL_CACERT=/etc/etcd/pki/ca.crt
ETCDCTL_CERT=/etc/etcd/pki/client.crt
```

让 etcd 数据库服务通过防火墙：

```
[root@worker1 ~]# firewall-cmd --add-service={etcd-server,etcd-client} --permanent
success
[root@worker1 ~]# firewall-cmd --reload
success
```

2. 正式把 worker1 加入集群（在 master1 上操作）：

```
[root@master1 ~]# etcdctl member add etcd02 --peer-urls=https://10.10.10.31:2380
Member e8d65284d2109029 added to cluster 8879ffbd85ff9506

ETCD_NAME="etcd02"
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380,etcd02=https://10.10.10.31:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.31:2380"
ETCD_INITIAL_CLUSTER_STATE="existing"
```

如果加错了，请在 master1 采用命令“etcdctl member remove <节点的 ID>”删掉，或者参考 23.11 小节排错。在启动新节点上的 etcd 服务时需要带上上面这条命令提示的四个变量，我们直接修改 master1 和 worker1 上的启动配置文件：

```
[root@master1 ~]# vim /etc/etcd/etcd.conf
```

```
ETCD_NAME="etcd01"
ETCD_DATA_DIR="/var/lib/etcd"

#配置客户端与 etcd 服务之间的通信：
ETCD_CLIENT_CERT_AUTH=true
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.11:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.11:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信：
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.11:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.11:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群：
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380,etcd02=https://10.10.10.31:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="new"

[root@worker1 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd02"
ETCD_DATA_DIR="/var/lib/etcd"

#配置客户端与 etcd 服务之间的通信：
ETCD_CLIENT_CERT_AUTH=true
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.31:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.31:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信：
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.31:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.31:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群：
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380,etcd02=https://10.10.10.31:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="existing"
```

最后重新启动 etcd（在 master1 和 worker1 上操作）：

```
[root@worker1 ~]# systemctl start etcd
[root@worker1 ~]# ss -tnlp | grep etcd
LISTEN 0      4096       127.0.0.1:2379       0.0.0.0:*    users:(("etcd",pid=12933,fd=9))
LISTEN 0      4096     10.10.10.31:2380       0.0.0.0:*    users:(("etcd",pid=12933,fd=7))
LISTEN 0      4096     10.10.10.31:2379       0.0.0.0:*    users:(("etcd",pid=12933,fd=8))

[root@master1 ~]# systemctl restart etcd
[root@master1 ~]# ss -tnlp | grep etcd
LISTEN 0      4096       127.0.0.1:2379       0.0.0.0:*    users:(("etcd",pid=9567,fd=9))
LISTEN 0      4096     10.10.10.11:2380       0.0.0.0:*    users:(("etcd",pid=9567,fd=7))
LISTEN 0      4096     10.10.10.11:2379       0.0.0.0:*    users:(("etcd",pid=9567,fd=8))
```

如果启动 etcd 失败，请执行命令“journalctl -xeu etcd.service”查看详细错误信息。

3．检查集群状态：

```
[root@master1 ~]# etcdctl member list --write-out=table
+------------------+---------+--------+---------------------------+---------------------------+------------+
|        ID        | STATUS  |  NAME  |        PEER ADDRS         |       CLIENT ADDRS        | IS LEARNER |
+------------------+---------+--------+---------------------------+---------------------------+------------+
| 36993bb67a1292d7 | started | etcd01 | https://10.10.10.11:2380 | https://10.10.10.11:2379 |      false |
| e8d65284d2109029 | started | etcd02 | https://10.10.10.31:2380 | https://10.10.10.31:2379 |      false |
+------------------+---------+--------+---------------------------+---------------------------+------------+

[root@master1 ~]#  etcdctl endpoint status --write-out=table
+--------------------------+------------------+---------+--------+-----------+------------+-----------+------------+--------------------+--------+
|         ENDPOINT         |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+--------------------------+------------------+---------+--------+-----------+------------+-----------+------------+--------------------+--------+
| https://10.10.10.11:2379 | 36993bb67a1292d7 |   3.5.9 |   20 kB |      true |      false |         3 |          7 |                  7 |        |
| https://10.10.10.31:2379 | e8d65284d2109029 |   3.5.9 |   20 kB |     false |      false |         3 |          7 |                  7 |        |
+--------------------------+------------------+---------+--------+-----------+------------+-----------+------------+--------------------+--------+
```

etcd01 节点目前为领导，etcd02 为随从。

```
[root@master1 ~]# etcdctl endpoint health --write-out=table
```

```
+--------------------------+--------+-------------+-------+
|         ENDPOINT         | HEALTH |    TOOK     | ERROR |
+--------------------------+--------+-------------+-------+
| https://10.10.10.11:2379 |   true |  7.591043ms |       |
| https://10.10.10.31:2379 |   true | 11.731226ms |       |
+--------------------------+--------+-------------+-------+
```

参考类似的方法，请把 worker2 加入到 etcd 集群，注意：第三个节点名为 etcd03，ip 地址为 10.10.10.32，三个节点上的配置文件/etc/etcd/etcd.conf 如下：

```
[root@master1 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd01"
ETCD_DATA_DIR="/var/lib/etcd"

#配置客户端与 etcd 服务之间的通信:
ETCD_CLIENT_CERT_AUTH=true
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.11:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.11:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信:
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.11:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.11:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380,etcd02=https://10.10.10.31:2380,etcd03=https://10.10.10.32:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="new"

[root@worker1 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd02"
ETCD_DATA_DIR="/var/lib/etcd"

#配置客户端与 etcd 服务之间的通信:
ETCD_CLIENT_CERT_AUTH=true
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.31:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.31:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信:
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.31:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.31:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380,etcd02=https://10.10.10.31:2380,etcd03=https://10.10.10.32:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="existing"

[root@worker2 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd03"
ETCD_DATA_DIR="/var/lib/etcd"

#配置客户端与 etcd 服务之间的通信:
ETCD_CLIENT_CERT_AUTH=true
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.32:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.32:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信:
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.32:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.32:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380,etcd02=https://10.10.10.31:2380,etcd03=https://10.10.10.32:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="existing"
```

如果加入失败，不妨在全部节点上执行：systemctl stop etcd;rm -rf /var/lib/etcd/*;systemctl start etcd，然后重新加入两个节点。

最后看到的集群如下：

```
[root@master1 ~]# etcdctl member list --write-out=table
+------------------+---------+---------+---------------------------+---------------------------+------------+
|        ID        | STATUS  |  NAME   |        PEER ADDRS         |       CLIENT ADDRS        | IS LEARNER |
+------------------+---------+---------+---------------------------+---------------------------+------------+
| 26d84fbfa456caca | started | etcd03 | https://10.10.10.32:2380 | https://10.10.10.32:2379 |      false |
| 36993bb67a1292d7 | started | etcd01 | https://10.10.10.11:2380 | https://10.10.10.11:2379 |      false |
| e8d65284d2109029 | started | etcd02 | https://10.10.10.31:2380 | https://10.10.10.31:2379 |      false |
+------------------+---------+---------+---------------------------+---------------------------+------------+

[root@master1 ~]# etcdctl endpoint status --write-out=table
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
|         ENDPOINT          |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
| https://10.10.10.32:2379 | 26d84fbfa456caca |   3.5.9 |   20 kB |     false |      false |         4 |         16 |                 16 |        |
| https://10.10.10.11:2379 | 36993bb67a1292d7 |   3.5.9 |   20 kB |     false |      false |         4 |         16 |                 16 |        |
| https://10.10.10.31:2379 | e8d65284d2109029 |   3.5.9 |   20 kB |      true |      false |         4 |         16 |                 16 |        |
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
```

发现现在 etcd02 成为领导节点，etcd02 是跑在 worker1 上的实例。由于三个节点的集群允许一个节点故障，我们手工关闭 worker1 上的 etcd 服务试试（模拟领导节点出故障）：

```
[root@worker1 ~]# systemctl stop etcd

[root@master1 ~]# etcdctl endpoint status --write-out=table
{"level":"warn","ts":"2023-08-09T10:34:28.629084+0800","logger":"etcd-client","caller":"v3@v3.5.9/retry_interceptor.go:62","msg":"retrying of unary
invoker failed","target":"etcd-endpoints://0xc000540540/10.10.10.11:2379","attempt":0,"error":"rpc error: code = DeadlineExceeded desc = latest balancer
error: last connection error: connection error: desc = \"transport: Error while dialing dial tcp 10.10.10.31:2379: connect: connection refused\""}
Failed to get the status of endpoint https://10.10.10.31:2379 (context deadline exceeded)
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
|         ENDPOINT          |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
| https://10.10.10.32:2379 | 26d84fbfa456caca |   3.5.9 |   20 kB |     false |      false |         5 |         17 |                 17 |        |
| https://10.10.10.11:2379 | 36993bb67a1292d7 |   3.5.9 |   20 kB |      true |      false |         5 |         17 |                 17 |        |
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
```

看到了没，etcd01 成为领导了。

```
[root@master1 ~]# etcdctl get hello
hello
Hello Etcd!
[root@master1 ~]# etcdctl put name Alice
OK
[root@master1 ~]# etcdctl get name
name
Alice
```

再启动 worker1 上的 etcd 服务：

```
[root@worker1 ~]# systemctl start etcd

[root@master1 ~]# etcdctl endpoint status --write-out=table
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
|         ENDPOINT          |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
| https://10.10.10.32:2379 | 26d84fbfa456caca |   3.5.9 |   20 kB |     false |      false |         5 |         19 |                 19 |        |
| https://10.10.10.11:2379 | 36993bb67a1292d7 |   3.5.9 |   20 kB |      true |      false |         5 |         19 |                 19 |        |
| https://10.10.10.31:2379 | e8d65284d2109029 |   3.5.9 |   20 kB |     false |      false |         5 |         19 |                 19 |        |
+---------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
```

领导没变。

```
[root@master1 ~]# etcdctl endpoint health
https://10.10.10.11:2379 is healthy: successfully committed proposal: took = 8.874755ms
https://10.10.10.32:2379 is healthy: successfully committed proposal: took = 13.719069ms
https://10.10.10.31:2379 is healthy: successfully committed proposal: took = 12.288992ms
```

重启动三台电脑，看看集群会不会自动起来。

如果要从集群中移除节点，请使用命令“etcdctl member remove <节点 ID>”。

§11.4 使用 etcd

§11.4.1 权限管理

主要针对用户、角色和鉴权操作，分别对应命令 etcdctl user ...、etcdctl role ...和 etcdctl auth ...。

1. 用户管理

```
etcdctl user ┬ add <用户名>|<用户名:密码> [--new-user-password|--no-password]
             ├ delete  <用户名>
             ├ get <用户名>
             ├ grant-role <用户名> <角色名>
             ├ list
             ├ passwd <用户名>
             └ revoke-role <用户名> <角色名>
```

刚开始 etcd 集群中没有用户。

```
[root@master1 ~]# etcdctl user list
[root@master1 ~]# etcdctl user add alice                                    #创建用户 alice
Password of alice:
Type password of alice again for confirmation:
User alice created
[root@master1 ~]# etcdctl user list
alice
[root@master1 ~]# etcdctl user add root:654321  ----------------------------------------  #创建超级用户 root，密码是 654321
User root created
[root@master1 ~]# etcdctl user get alice                                    #获取用户 alice 的信息
User: alice
Roles:
```

etcd 中的特殊用户 root 和特殊角色 root，角色 root 的权限没有限制，root 用户默认被赋予了 root 角色。

```
[root@master1 ~]# etcdctl user grant-role root root                         #授予 root 用户 root 角色
Role root is granted to user root

[root@master1 ~]# etcdctl user get root
User: root
Roles: root

[root@master1 ~]# etcdctl user grant-role alice root  ----------------------------------------  #授予 alice 用户 root 角色
Role root is granted to user alice

[root@master1 ~]# etcdctl user revoke-role alice root                       #剥夺 alice 用户 root 角色
Role root is revoked from user alice
```

2. 角色管理

```
etcdctl role ┬ add  <角色名>
             ├ delete <角色名>
             ├ get  <角色名>
             ├ grant-permission <角色名> [--prefix=true] <read|write|readwrite> <key> [endkey]
             ├ list
             └ revoke-premission <角色名> <key> [endkey]
```

权限有三种：read、write 和 readwrite。etcd 保存的记录都是“键=值”（即 key=value）对，所以给角色增加权限的命令“etcdctl role grant-permission …”中的<key>就是“键=值”对中的键，如果不省略[endkey]，表示范围[key,endkey)；如果没省略--prefix=true，表示以 key 为前缀的所有键。

```
[root@master1 ~]# etcdctl role list
```

```
[root@master1 ~]# etcdctl role add myrole ------------------------------------------------------------------------ #新建角色 myrole
Role myrole created
[root@master1 ~]# etcdctl role grant-permission myrole write /foo                    #在键/foo 上赋予 myrole 写的权限
Role myrole updated

在以 Alice/为前缀的所有键上赋予 myrole 读和写的权限:
[root@master1 ~]# etcdctl role grant-permission myrole --prefix=true readwrite Alcie/
Role myrole updated

在大于等于 John/2021 且小于 John/2022 的全部键上赋予 myrole 角色读的权限:
[root@master1 ~]# etcdctl role grant-permission myrole read John/2021 John/2022
Role myrole updated

[root@master1 ~]# etcdctl role list
myrole

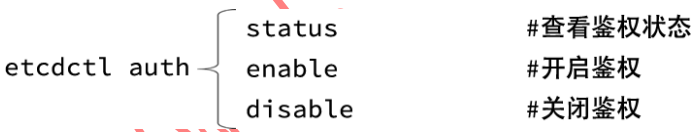
[root@master1 ~]# etcdctl role get myrole ------------------------------------------------------- #查看 myrole 角色拥有的权限
Role myrole
KV Read:
        /foo
        [Alcie/, Alcie0) (prefix Alcie/)
        [John/2021, John/2022) (prefix John/2021)
KV Write:
        [Alcie/, Alcie0) (prefix Alcie/)

在/foo 键上剥夺 myrole 角色的全部权限:
[root@master1 ~]# etcdctl role revoke-permission myrole /foo
Permission of key /foo is revoked from role myrole

#在[John/2021, John/2022)上剥夺 myrole 角色的全部权限:
[root@master1 ~]# etcdctl role revoke-permission myrole John/2021 John/2022
Permission of range [John/2021, John/2022) is revoked from role myrole

[root@master1 ~]# etcdctl role get myrole
Role myrole
KV Read:
        [Alcie/, Alcie0) (prefix Alcie/)
KV Write:
        [Alcie/, Alcie0) (prefix Alcie/)
```

3．开启鉴权



默认是访问 etcd 存储不用鉴权的——谁都可以访问。但在生产环境中，还是建议开启鉴权，然后只有合法用户才能执行允许的操作。开启鉴权前首先要确保 root 用户存在。

```
[root@master1 ~]# etcdctl auth status                                                #检查鉴权状态
Authentication Status: false
AuthRevision: 16

[root@master1 ~]# etcdctl user list ------------------------------------------------------------------------ #确保 root 用户存在
alice
root

[root@master1 ~]# etcdctl auth enable ---------------------------------------------------------------------------------- #开启鉴权
Authentication Enabled

[root@master1 ~]# etcdctl auth status --user=root:654321                    #从此需要用--user 参数指明用户和密码
Authentication Status: true
AuthRevision: 16
```

开启鉴权之后，客户端都要提供合法的用户才能访问，可以在命令行上通过参数--user=<user>:<password>指定，或者分别用两个参数指定，即--user=<user> --password=<password>，但建议定义两个环境变量：ETCDCTL_USER=<user>、ETCDCTL_PASSWORD=<password>，如把定义这两个环境的语句追加到~/.bashrc 文件的末尾：

```
[root@master1 ~]# cat << EOF >> ~/.bashrc
export ETCDCTL_USER=root
export ETCDCTL_PASSWORD=654321
EOF
```

这样下次登陆时，环境变量自动生效。

§11.4.2 内容操作

主要是“增/删/改/查”和监测。

1. 增/删/改/查：

```
[root@master1 ~]# etcdctl put hello "Hello Etcd!"                                    #增加 hello="Hello Etcd!"
OK
[root@master1 ~]# etcdctl get hello  ------------------------------------------------------------------------ #查询 hello 键的值
hello
Hello Etcd!
[root@master1 ~]# etcdctl put hello "Good Morning!" ------------------------------------------------------- #修改 hello 键的值
OK
[root@master1 ~]# etcdctl get hello
hello
Good Morning!
[root@master1 ~]# etcdctl del hello  ------------------------------------------------------------------------ #删除 hello 键
1
[root@master1 ~]# etcdctl get hello
```

实战：把下面的内容保存到 etcd 中：

```
myclass/=我上课的班级
        netclass1/=网络 1 班
                  2021001/=学号
                           name="张三"
                           age=19
                           phone=123456
                  2021002/=学号
                           name="李四"
                           age=20
                           phone=20223456
                           sex=male

        netclass2/=网络 2 班
                  2022005/=学号
                           name="王二"
                           age=22
                           phone=1023456
        netclass3/=网络 3 班
                  2023001/=学号
                           name="Alcie"
                           age=20
                           sex=female
```

操作如下：

```
[root@master1 ~]# etcdctl put myclass/ "我上课的班级"
OK
[root@master1 ~]# etcdctl put myclass/netclass1/ "网络 1 班"
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021001/ "学号"
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021001/name "张三"
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021001/age 19
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021001/phone 123456
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021002/ "学号"
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021002/name "李四"
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021002/age 20
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021002/phone 20223456
OK
[root@master1 ~]# etcdctl put myclass/netclass1/2021002/sex male
OK
[root@master1 ~]# etcdctl put myclass/netclass2/ "网络 2 班"
```

```
OK
[root@master1 ~]# etcdctl put myclass/netclass2/2022005/ "学号"
OK
[root@master1 ~]# etcdctl put myclass/netclass2/2022005/name "王二"
OK
[root@master1 ~]# etcdctl put myclass/netclass2/2022005/age 22
OK
[root@master1 ~]# etcdctl put myclass/netclass2/2022005/phone 1023456
OK
[root@master1 ~]# etcdctl put myclass/netclass3/ "网络 3 班"
OK
[root@master1 ~]# etcdctl put myclass/netclass3/2023001/ "学号"
OK
[root@master1 ~]# etcdctl put myclass/netclass3/2023001/name "Alice"
OK
[root@master1 ~]# etcdctl put myclass/netclass3/2023001/age 20
OK
[root@master1 ~]# etcdctl put myclass/netclass3/2023001/sex female
OK

[root@worker1 ~]# etcdctl get --prefix=true myclass/
myclass/
我上课的班级
myclass/netclass1
网络 1 班
myclass/netclass1/2021001
学号
myclass/netclass1/2021001/age
19
myclass/netclass1/2021001/name
张三
myclass/netclass1/2021001/phone
123456
myclass/netclass1/2021002
学号
myclass/netclass1/2021002/age
20
myclass/netclass1/2021002/name
李四
myclass/netclass1/2021002/phone
20223456
myclass/netclass1/2021002/sex
male
myclass/netclass2
网络 2 班
myclass/netclass2/2022005
学号
myclass/netclass2/2022005/age
22
myclass/netclass2/2022005/name
王二
myclass/netclass2/2022005/phone
1023456
myclass/netclass3
网络 3 班
myclass/netclass3/2023001/
学号
myclass/netclass3/2023001/age
20
myclass/netclass3/2023001/name
Alice
myclass/netclass3/2023001/sex
female

[root@master1 ~]# etcdctl get --prefix=true myclass/netclass2/
myclass/netclass2/2022005
学号
myclass/netclass2/2022005/age
22
myclass/netclass2/2022005/name
王二
myclass/netclass2/2022005/phone
1023456

[root@master1 ~]# etcdctl get myclass/netclass2/2022005/name
myclass/netclass2/2022005/name
王二
```

2. 监测键的值

etcdctl watch --help

```
[root@master1 ~]# etcdctl watch --prefix=true myclass/
```

在其他屏幕上修改：

```
[root@worker1 ~]# etcdctl put myclass/netclass2/2022005/age 21
OK
```

那个检测的屏幕上会显示：

```
[root@master1 ~]# etcdctl watch --prefix=true myclass/
PUT
myclass/netclass2/2022005/age
21
```

当有人改动数据时执行 linux 命令：

```
[root@master1 ~]# etcdctl watch --prefix=true myclass/ -- wall "有人改动数据了！"
```

在其他屏幕上修改：

```
[root@worker1 ~]# etcdctl put myclass/netclass2/2022005/name "特朗普"
OK
```

3．租约管理：

租约用于给“键=值”对设置有效期，过期的数据被自动销毁。

```
etcdctl lease ┬ grant <秒数>                          #创建租约
              ├ revoke <租约的id>                     #删除租约
              ├ keep-alive [--once=true] <租约的id>   #续约
              ├ timetolive <租约的id>                 #查看租约信息
              └ list                                  #列出存在的租约
```

到期的租约被销毁，还没到期的租约允许续约，续约命令中的可选参数--once=true 表示重置一次续约期，即又从最大时间开始倒计时。

可以为 etcd 设定租约。在时间到期后租约失效。键可以绑定到租约上，租约到期时键随着失效。

```
[root@master1 ~]# etcdctl lease grant 100                                    #新建一个租约期 100 秒
lease 102989d8296a3538 granted with TTL(100s)

[root@master1 ~]# etcdctl lease list
found 1 leases
102989d8296a3538

[root@master1 ~]# etcdctl lease timetolive 102989d8296a3538                  #查看租约期剩下的时间
lease 102989d8296a3538 granted with TTL(100s), remaining(65s)

[root@master1 ~]# etcdctl lease keep-alive 102989d8296a3538                  #暂停租约倒计时，直到按 Ctrl+c 后开始倒计时
lease 102989d8296a3538 keepalived with TTL(100)
^C

[root@master1 ~]# etcdctl lease timetolive 102989d8296a3538
lease 102989d8296a3538 granted with TTL(100s), remaining(83s)

[root@master1 ~]# etcdctl lease revoke 102989d8296a3538                      #删除租约
lease 102989d8296a3538 revoked

[root@master1 ~]# etcdctl lease list
found 0 leases

[root@master1 ~]# etcdctl lease grant 200
lease 102989d8296a3540 granted with TTL(200s)

[root@master1 ~]# etcdctl lease list
found 1 leases
102989d8296a3540

[root@master1 ~]# etcdctl put --lease=102989d8296a3540 foo bar               #增加数据“foo=bar”，并绑定租约（有效期）
OK
[root@master1 ~]# etcdctl put get foo
OK

[root@master1 ~]# etcdctl lease timetolive 102989d8296a3540
lease 102989d8296a3540 granted with TTL(200s), remaining(36s)
```

```
[root@master1 ~]# etcdctl lease timetolive 102989d8296a3540
lease 102989d8296a3540 already expired

[root@master1 ~]# etcdctl lease list ------------------------------------------------------------------------------#租约到期就销毁了
found 0 leases

[root@master1 ~]# etcdctl get foo                                                                    #过期的“键值”对不存在了
```

§11.4.3 备份和恢复

1. 备份：

```
[root@master1 ~]# env|grep ETCDCTL
ETCDCTL_USER=root
ETCDCTL_ENDPOINTS=https://10.10.10.11:2379,https://10.10.10.31:2379,https://10.10.10.32:2379
ETCDCTL_KEY=/etc/etcd/pki/client.key
ETCDCTL_CACERT=/etc/etcd/pki/ca.crt
ETCDCTL_PASSWORD=654321
ETCDCTL_CERT=/etc/etcd/pki/client.crt

备份只能采用一个 endpoint，所以先删除 ETCDCTL_ENDPOINTS 变量
[root@master1 ~]# unset ETCDCTL_ENDPOINTS                                  #删除环境变量，因为它与下面的--endpoints 参数冲突

在多节点集群中，备份时必须用参数--endpoints 指明一个节点。备份出来的数据保存当前目录中的 etcd-backup.db 文件内：
[root@master1 ~]# etcdctl --endpoints=https://10.10.10.11:2379 snapshot save ./etcd-backup.db
{"level":"info","ts":"2023-08-10T07:21:37.538247+0800","caller":"snapshot/v3_snapshot.go:65","msg":"created temporary db file","path":"my.db.part"}
{"level":"info","ts":"2023-08-10T07:21:37.596144+0800","logger":"client","caller":"v3@v3.5.9/maintenance.go:212","msg":"opened snapshot stream; downloading"}
{"level":"info","ts":"2023-08-10T07:21:37.596207+0800","caller":"snapshot/v3_snapshot.go:73","msg":"fetching snapshot","endpoint":"https://10.10.10.11:2379"}
{"level":"info","ts":"2023-08-10T07:21:37.599481+0800","logger":"client","caller":"v3@v3.5.9/maintenance.go:220","msg":"completed snapshot read; closing"}
{"level":"info","ts":"2023-08-10T07:21:37.600167+0800","caller":"snapshot/v3_snapshot.go:88","msg":"fetched snapshot","endpoint":"https://10.10.10.11:2379","size":"25
kB","took":"now"}
{"level":"info","ts":"2023-08-10T07:21:37.600245+0800","caller":"snapshot/v3_snapshot.go:97","msg":"saved","path":"etcd-backup.db"}
Snapshot saved at etcd-backup.db

[root@master1 ~]# ll etcd-backup.db
-rw------- 1 root root 24608 Aug  9 09:13 etcd-backup.db

[root@master1 ~]# etcdutl snapshot status etcd-backup.db --write-out=table                          #查看备份状态
+----------+----------+------------+------------+
|   HASH   | REVISION | TOTAL KEYS | TOTAL SIZE |
+----------+----------+------------+------------+
| ff1d47ee |       34 |         45 |      25 kB |
+----------+----------+------------+------------+
```

2. 恢复（要在每个节点上操作）：

恢复其实就是新建集群和集群中的全部节点，所以要取新的集群名、节点名、数据目录。

首先关闭 etcd 集群（全部节点上执行）：

```
[root@allnodes ~]# systemctl stop etcd
```

第一个节点：新的集群名 etcd-cluster.1、新的节点名 etcd01.1、新的数据目录/var/lib/etcd-cluster.1。名字中的“.1”同时表示第 1 次恢复，以后第 2 次恢复时可加后缀“.2”，具体怎么命名其实没有规定，由你自己定。

```
[root@master1 ~]# etcdutl snapshot restore etcd-backup.db \
  --name etcd01.1 \
  --initial-cluster etcd01.1=https://10.10.10.11:2380,etcd02.1=https://10.10.10.31:2380,etcd03.1=https://10.10.10.32:2380 \
  --initial-cluster-token etcd-cluster.1 \
  --initial-advertise-peer-urls https://10.10.10.11:2380 \
  --data-dir /var/lib/etcd-cluster.1
……
g membership information from the backend...
 added member    {"cluster-id": "faf49cc6e967146", "local-member-id": "0", "added-peer-id": "993a4bcbafefb5fa", "added-peer-peer-urls": ["https://10.10.10.32:2380"]}
 added member    {"cluster-id": "faf49cc6e967146", "local-member-id": "0", "added-peer-id": "a1f9d89d066dff31", "added-peer-peer-urls": ["https://10.10.10.11:2380"]}
 added member    {"cluster-id": "faf49cc6e967146", "local-member-id": "0", "added-peer-id": "ab3b3403519a72dc", "added-peer-peer-urls": ["https://10.10.10.31:2380"]}
 restored snapshot       {"path": "etcd-backup.db", "wal-dir": "/var/lib/etcd-cluster.1/member/wal", "data-dir": "/var/lib/etcd-cluster.1", "snap-dir": "/var/lib/etcd

[root@master1 ~]# chown -R etcd: /var/lib/etcd-cluster.1
```

第二个节点：新的节点名 etcd02.1：

```
[root@worker1 ~]# scp master1:etcd-backup.db ./
```

```
[root@worker1 ~]# etcdutl snapshot restore etcd-backup.db \
  --name etcd02.1 \
  --initial-cluster etcd01.1=https://10.10.10.11:2380,etcd02.1=https://10.10.10.31:2380,etcd03.1=https://10.10.10.32:2380 \
  --initial-cluster-token etcd-cluster.1 \
  --initial-advertise-peer-urls https://10.10.10.31:2380 \
  --data-dir /var/lib/etcd-cluster.1
......
{"level":"info","ts":1677744273.1,"caller":"membership/cluster.go:392","msg":"added member","cluster-id":"faf49cc6e967146","local-member-id":"0","added-peer-id":"993a4bcbafefb5fa","added-peer-peer-urls":["https://10.10.10.32:2380"]}
{"level":"info","ts":1677744273.1,"caller":"membership/cluster.go:392","msg":"added member","cluster-id":"faf49cc6e967146","local-member-id":"0","added-peer-id":"a1f9d89d066dff31","added-peer-peer-urls":["https://10.10.10.11:2380"]}
{"level":"info","ts":1677744273.1,"caller":"membership/cluster.go:392","msg":"added member","cluster-id":"faf49cc6e967146","local-member-id":"0","added-peer-id":"ab3b3403519a72dc","added-peer-peer-urls":["https://10.10.10.31:2380"]}
{"level":"info","ts":1677744273.1,"caller":"snapshot/v3_snapshot.go:309","msg":"restored snapshot","path":"etcd-backup.db","wal-dir":"/var/lib/etcd-cluster.1/member/wal","data-dir":"/var/lib/etcd-cluster.1","snap-dir":"/var/lib/etcd-cluster.1/member/snap"}

[root@worker1 ~]# chown -R etcd: /var/lib/etcd-cluster.1
```

第三个节点：新的节点名 etcd03.1：

```
[root@worker2 ~]# scp master1:etcd-backup.db ./
[root@worker2 ~]# etcdutl snapshot restore etcd-backup.db \
  --name etcd03.1 \
  --initial-cluster etcd01.1=https://10.10.10.11:2380,etcd02.1=https://10.10.10.31:2380,etcd03.1=https://10.10.10.32:2380 \
  --initial-cluster-token etcd-cluster.1 \
  --initial-advertise-peer-urls https://10.10.10.32:2380 \
  --data-dir /var/lib/etcd-cluster.1
......
{"level":"info","ts":1677744273.1,"caller":"membership/cluster.go:392","msg":"added member","cluster-id":"faf49cc6e967146","local-member-id":"0","added-peer-id":"993a4bcbafefb5fa","added-peer-peer-urls":["https://10.10.10.32:2380"]}
{"level":"info","ts":1677744273.1,"caller":"membership/cluster.go:392","msg":"added member","cluster-id":"faf49cc6e967146","local-member-id":"0","added-peer-id":"a1f9d89d066dff31","added-peer-peer-urls":["https://10.10.10.11:2380"]}
{"level":"info","ts":1677744273.1,"caller":"membership/cluster.go:392","msg":"added member","cluster-id":"faf49cc6e967146","local-member-id":"0","added-peer-id":"ab3b3403519a72dc","added-peer-peer-urls":["https://10.10.10.31:2380"]}
{"level":"info","ts":1677744273.1,"caller":"snapshot/v3_snapshot.go:309","msg":"restored snapshot","path":"etcd-backup.db","wal-dir":"/var/lib/etcd-cluster.1/member/wal","data-dir":"/var/lib/etcd-cluster.1","snap-dir":"/var/lib/etcd-cluster.1/member/snap"}

[root@worker2 ~]# chown -R etcd: /var/lib/etcd-cluster.1
```

3. 启动新的集群：

```
[root@master1 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd01.1"
ETCD_DATA_DIR="/var/lib/etcd-cluster.1"
ETCD_PEER_CLIENT_CERT_AUTH=true

#配置客户端与 etcd 服务之间的通信：
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.11:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.11:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信：
ETCD_LISTEN_PEER_URLS="https://10.10.10.11:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.11:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群：
ETCD_INITIAL_CLUSTER="etcd01.1=https://10.10.10.11:2380,etcd02.1=https://10.10.10.31:2380,etcd03.1=https://10.10.10.32:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster.1"
ETCD_INITIAL_CLUSTER_STATE="existing"

[root@master1 ~]# systemctl start etcd
```

在第二个节点上操作：

```
[root@worker1 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd02.1"
ETCD_DATA_DIR="/var/lib/etcd-cluster.1"
ETCD_PEER_CLIENT_CERT_AUTH=true

#配置客户端与 etcd 服务之间的通信：
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.31:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.31:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信：
ETCD_LISTEN_PEER_URLS="https://10.10.10.31:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.31:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群：
ETCD_INITIAL_CLUSTER="etcd01.1=https://10.10.10.11:2380,etcd02.1=https://10.10.10.31:2380,etcd03.1=https://10.10.10.32:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster.1"
ETCD_INITIAL_CLUSTER_STATE="existing"

[root@worker1 ~]# systemctl start etcd
```

在第三个节点上操作：

```
[root@worker2 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd03.1"
ETCD_DATA_DIR="/var/lib/etcd-cluster.1"
```

```
ETCD_PEER_CLIENT_CERT_AUTH=true

#配置客户端与 etcd 服务之间的通信:
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.32:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.32:2379"
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信:
ETCD_LISTEN_PEER_URLS="https://10.10.10.32:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.32:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置集群:
ETCD_INITIAL_CLUSTER="etcd01.1=https://10.10.10.11:2380,etcd02.1=https://10.10.10.31:2380,etcd03.1=https://10.10.10.32:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster.1"
ETCD_INITIAL_CLUSTER_STATE="existing"

[root@worker2 ~]# systemctl start etcd
```

4．检查集群状态：

```
[root@master1 ~]# . .bashrc                                                    #输出环境变量
[root@master1 ~]# env | grep ETCDCTL
ETCDCTL_USER=root
ETCDCTL_ENDPOINTS=https://10.10.10.11:2379,https://10.10.10.31:2379,https://10.10.10.32:2379
ETCDCTL_KEY=/etc/etcd/pki/client.key
ETCDCTL_CACERT=/etc/etcd/pki/ca.crt
ETCDCTL_PASSWORD=654321
ETCDCTL_CERT=/etc/etcd/pki/client.crt

[root@master1 ~]# etcdctl endpoint status --write-out=table
+--------------------------+-----------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
|        ENDPOINT          |        ID       | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+--------------------------+-----------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
| https://10.10.10.32:2379 | 993a4bcbafefb5fa |   3.5.9 |   25 kB |     false |      false |         2 |         11 |                 11 |        |
| https://10.10.10.11:2379 | a1f9d89d066dff31 |   3.5.9 |   25 kB |      true |      false |         2 |         12 |                 12 |        |
| https://10.10.10.31:2379 | ab3b3403519a72dc |   3.5.9 |   29 kB |     false |      false |         2 |         13 |                 13 |        |
+--------------------------+-----------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+

[root@master1 ~]# etcdctl endpoint health --write-out=table
+--------------------------+--------+------------+-------+
|          ENDPOINT        | HEALTH |    TOOK    | ERROR |
+--------------------------+--------+------------+-------+
| https://10.10.10.11:2379 |   true |  1.31599ms |       |
| https://10.10.10.31:2379 |   true | 2.064035ms |       |
| https://10.10.10.32:2379 |   true | 1.928158ms |       |
+--------------------------+--------+------------+-------+

[root@master1 ~]# etcdctl get  0 z                                             #查询[0,z)范围内的全部建值对
get
foo
hello/k1
Hello Etcd!
myclass/
我上课的班级
myclass/netclass1
网络 1 班
……
```

等新的集群正常使用后，可以考虑把之前的数据目录删除以腾出磁盘空间。比如执行如下命令：

```
[root@allnodes ~]# rm -rf /var/lib/etcd
```

第12章 Ceph 存储

§12.1 系统设计

§ 12.1.1 ceph 简介

Ceph 是一个 SDS，它提供了块存储、文件存储和对象存储（通过 RESTful 调用）。Ceph 的逻辑架构如下图参考图 12-1。



图 12-1 Ceph 的逻辑架构

Ceph 的物理结构见图 12-2 所示。

https://www.cn.org

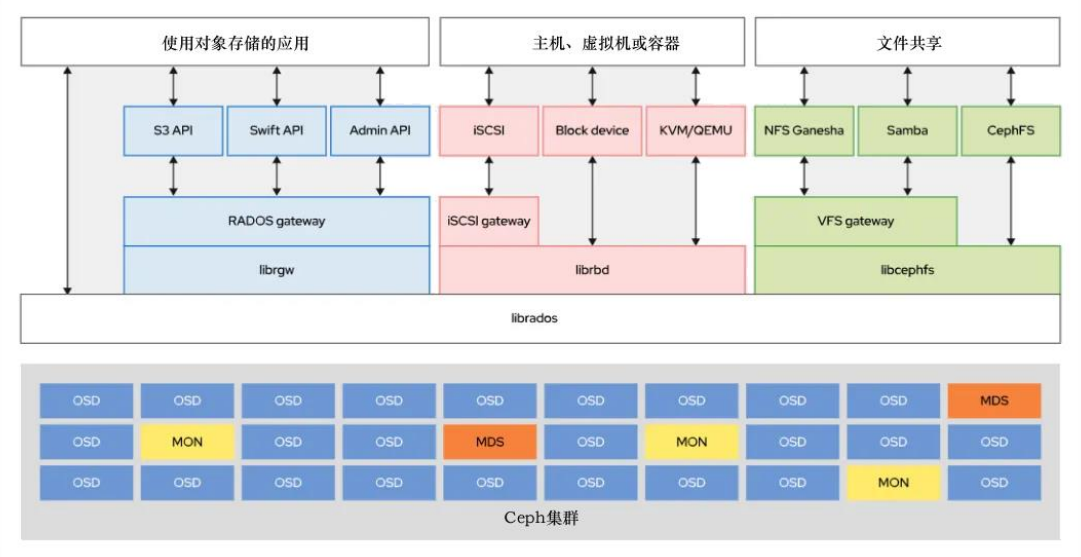


图 12-2 Ceph 集群的物理架构

MDS=MetaData Server（文件服务需要用到）、MON=Monitor、OSD=Object Storage Daemon。

一、 Ceph 存储算法流程图

参考图 12-3 所示。

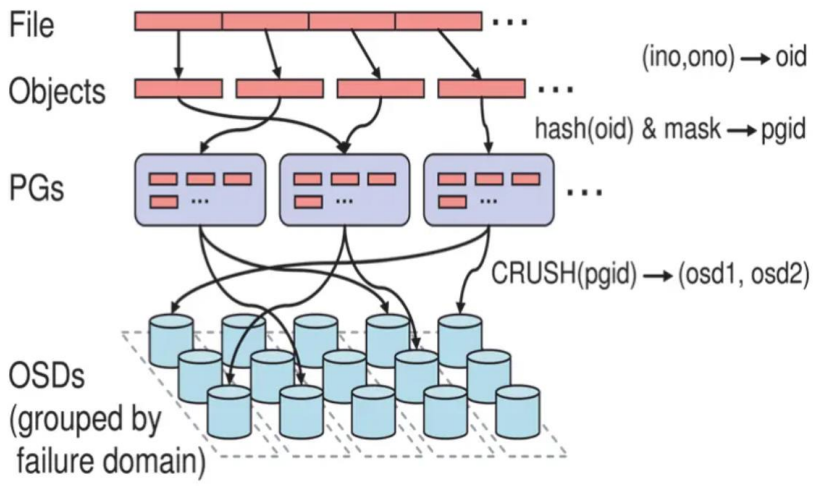


图 12-3 Ceph 存储算法流程

1. File：用户需要读写的文件。File->Object 映射：

   (1). ino (File 的元数据，File 的唯一 id)。
   (2). ono(File 切分产生的某个 object 的序号，默认以 4M 切分一个块大小)。
   (3). oid(object id: ino + ono)。

2. Object 是 RADOS 需要的对象。Ceph 指定一个静态 hash 函数计算 oid 的值，将 oid 映射成一个近似均匀分布的伪随机值，然后和 mask 按位相与，得到 pgid。Object->PG 映射：

(1). hash(oid) & mask-> pgid 。

(2). mask = PG 总数 m(m 为 2 的整数幂)-1 。

3. PG(Placement Group)，用途是对 object 的存储进行组织和位置映射， (类似于 redis cluster 里面的 slot 的概念) 一个 PG 里面会有很多 object。采用 CRUSH 算法，将 pgid 代入其中，然后得到一组 OSD。

PG->OSD 映射：

(1). CRUSH(pgid)->(osd1,osd2,osd3) 。

二、Ceph RBD IO 流程

参考图 12-4。

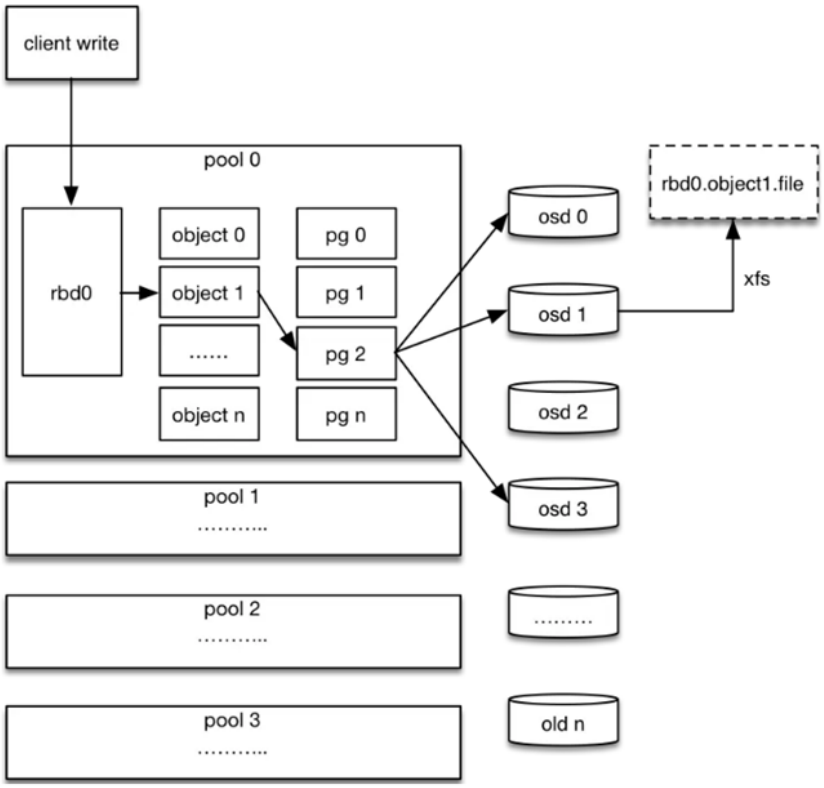


图 12-4 Ceph RBD 的 io 流程

步骤：

1. 客户端创建一个 pool，需要为这个 pool 指定 pg 的数量。
2. 创建 pool/image rbd 设备进行挂载。
3. 用户写入的数据进行切块，每个块的大小默认为 4M，并且每个块都有一个名字，名字就是 object+序号。
4. 将每个 object 通过 pg 进行副本位置的分配。
5. pg 根据 cursh 算法会寻找 3 个 osd，把这个 object 分别保存在这三个 osd 上。

6. osd 上实际是把底层的 disk 进行了格式化操作，一般部署工具会将它格式化为 xfs 文件系统。

7. object 的存储就变成了存储一个文 rbd0.object1.file。

三、Ceph RBD IO 框架图

参考图 12-5。

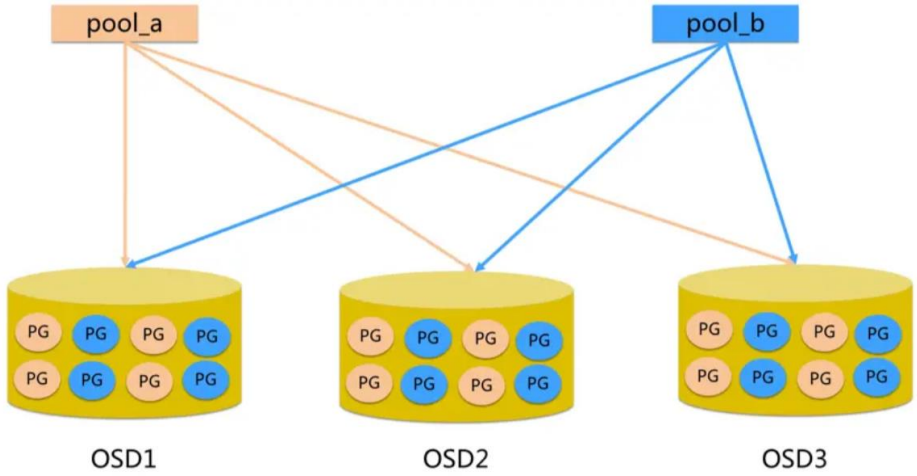


图 12-5 Ceph RBD 的 IO 框架

说明：

1. pool 是 ceph 存储数据时的逻辑分区，它起到 namespace 的作用。
2. 每个 pool 包含一定数量(可配置)的 PG。
3. PG 里的对象被映射到不同的 Object 上。
4. pool 是分布到整个集群的。
5. pool 可以做故障隔离域，可根据不同的用户场景进行隔离。

由此可见，一个 Ceph 集群包含多个 pool（池），每个 pool 包含多个 pg（放置组），每个 pg 被存储在多个 osd 上。存储时，每个文件被分割成多个相同大小的块（object），每个块被映射到某个 pg 中，通过 pg 的多副本存储实现了每个文件的多副本存储，从而消除了单点故障。

§12.1.2 实验架构

做实验时采用五台虚拟机，实验系统架构图如图 12-6 所示。

https://www.ve...

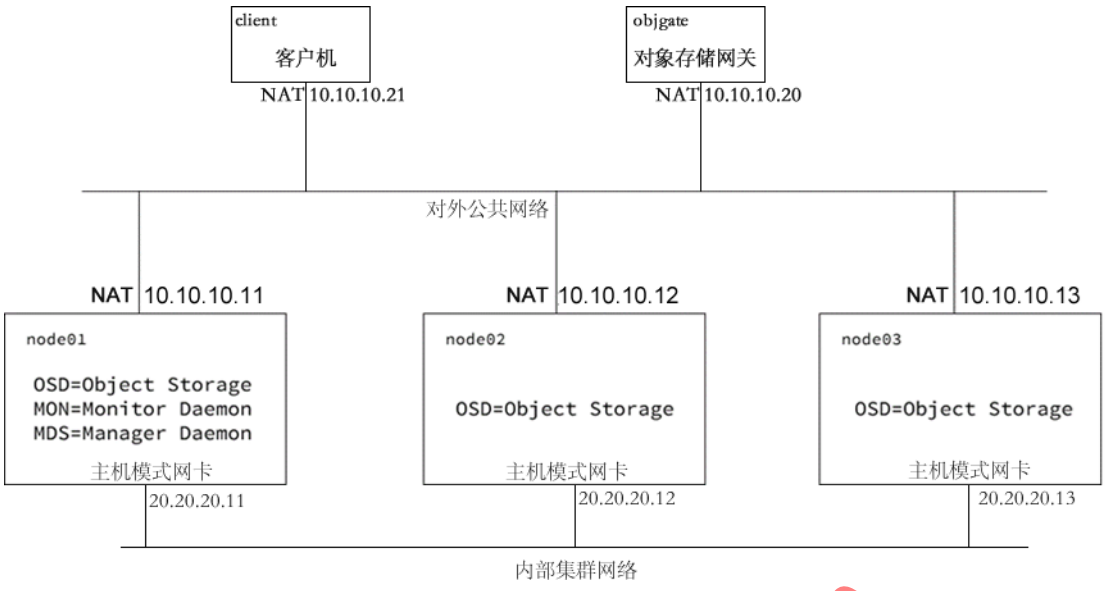


图 12-6 实验框图

注意：也可以使用 node02 或 node03 充当 client 和 objgate 的角色。

1．查看第二块网卡的名称：

```
[root@node01 ~]# nmcli device status
DEVICE  TYPE      STATE         CONNECTION
ens160  ethernet  已连接        ens160
ens192  ethernet  已连接        ens192                                                                    #第二块网卡
lo      loopback  连接（外部）  lo
```

2．三个节点配置第二块网卡，注意网卡名称要根据实际情况替换：

```
[root@node01 ~]# nmcli c del ens192
[root@node01 ~]# nmcli c add type ethernet ifname ens192 con-name ens192
[root@node01 ~]# nmcli c modify ens192 ipv4.addresses 20.20.20.11/24
[root@node01 ~]# nmcli c modify ens192 ipv4.method manual
[root@node01 ~]# nmcli c modify ens192 connection.autoconnect true
[root@node01 ~]# nmcli c up ens192

[root@node02 ~]# nmcli c del ens192
[root@node02 ~]# nmcli c add type ethernet ifname ens192 con-name ens192
[root@node02 ~]# nmcli c modify ens192 ipv4.addresses 20.20.20.12/24
[root@node02 ~]# nmcli c modify ens192 ipv4.method manual
[root@node02 ~]# nmcli c modify ens192 connection.autoconnect true
[root@node02 ~]# nmcli c up ens192

[root@node03 ~]# nmcli c del ens192
[root@node03 ~]# nmcli c add type ethernet ifname ens192 con-name ens192
[root@node03 ~]# nmcli c modify ens192 ipv4.addresses 20.20.20.13/24
[root@node03 ~]# nmcli c modify ens192 ipv4.method manual
[root@node03 ~]# nmcli c modify ens192 connection.autoconnect true
[root@node03 ~]# nmcli c up ens192

[root@node01 ~]# ping 20.20.20.12
PING 20.20.20.12 (20.20.20.12) 56(84) 比特的数据。
64 比特，来自 20.20.20.12: icmp_seq=1 ttl=64 时间=0.471 毫秒
^C
--- 20.20.20.12 ping 统计 ---
已发送 1 个包， 已接收 1 个包，0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.471/0.471/0.471/0.000 ms

[root@node01 ~]# ping 20.20.20.13
PING 20.20.20.13 (20.20.20.13) 56(84) 比特的数据。
64 比特，来自 20.20.20.13: icmp_seq=1 ttl=64 时间=0.504 毫秒
^C
--- 20.20.20.13 ping 统计 ---
已发送 1 个包， 已接收 1 个包，0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.504/0.504/0.504/0.000 ms
```

§12.2 安装 Ceph

基本要求：Python3、Podman/Docker、三个节点时钟已经同步、存在至少一块空硬盘、LVM2 文件系统。

§12.2.1 初始化最小集群

这里采用 cephadm 工具来部署。

1. 搜索 ceph 版本：

```
[root@node01 ~]# dnf search release-ceph
==================================== 名称 匹配: release-ceph =====================================
rocky-release-ceph-pacific.noarch : Ceph Pacific packages from the Rocky Storage SIG repository
rocky-release-ceph-quincy.noarch : Ceph Quincy packages from the Rocky Storage SIG repository
rocky-release-ceph-reef.noarch : Ceph Reef packages from the Rocky Storage SIG repository
```

可见 Rocky 支持的最新三个版本是 pacific、quincy 和 reef，版本的新旧有单词的首字母决定，所以 reef 是最新版。本手册安装 ceph 的最新版本 reef。

2. 安装最新版本：

```
[root@node01 ~]# dnf install rocky-release-ceph-reef.noarch -y
[root@node01 ~]# dnf install cephadm -y
```

会同时安装诸如 podman 若干依赖包。podman 是容器，与 docker 完全兼容。

3. 检查各个软件版本：

```
[root@node01 ~]# python --version
Python 3.9.18

[root@node01 ~]# podman version
Client:       Podman Engine
Version:      4.9.3-dev
API Version:  4.9.3-dev
Go Version:   go1.21.3
Built:        Tue Feb  6 18:42:38 2024
OS/Arch:      linux/amd64

[root@node01 ~]# cephadm version
cephadm version 18.2.2 (531c0d11a1c5d39fbfe6aa8a521f023abf3bf3e2) reef (stable)
```

4. 初始化一个最小的 ceph 集群，其中 10.10.10.11 为第一块网卡的 ip，20.20.20.0/24 为内部集群网络，ceph 的复制、恢复和心跳数据包通过内部集群网络传递。初始化集群的命令会输出很多重要的信息在屏幕上，为了事后查询这些信息，我们同时保存在/tmp/bootstrap.log 文件中（| tee /tmp/bootstrap.log）：

```
[root@node01 ~]# cephadm bootstrap --mon-ip 10.10.10.11 --cluster-network 20.20.20.0/24 | tee /tmp/bootstrap.log
Verifying podman|docker is present...

Verifying lvm2 is present...
Verifying time synchronization is in place...
Unit chronyd.service is enabled and running
Repeating the final host check...
podman (/usr/bin/podman) version 4.9.3 is present
systemctl is present
lvcreate is present
Unit chronyd.service is enabled and running
Host looks OK
Cluster fsid: 6d6b022e-e689-11ee-b2f5-000c2912f7ba
Verifying IP 10.10.10.11 port 3300 ...
Verifying IP 10.10.10.11 port 6789 ...
Mon IP `10.10.10.11` is in CIDR network `10.10.10.0/24`
Mon IP `10.10.10.11` is in CIDR network `10.10.10.0/24`
Pulling container image quay.io/ceph/ceph:v18...

Ceph version: ceph version 18.2.2 (531c0d11a1c5d39fbfe6aa8a521f023abf3bf3e2) reef (stable)
Extracting ceph user uid/gid from container image...
Creating initial keys...
```

```
Creating initial monmap...
Creating mon...
firewalld ready
Enabling firewalld service ceph-mon in current zone...
Waiting for mon to start...
Waiting for mon...
mon is available
Assimilating anything we can from ceph.conf...
Generating new minimal ceph.conf...
Restarting the monitor...
Setting public_network to 10.10.10.0/24 in mon config section----------------------------------------------------

Setting cluster_network to 20.20.20.0/24
Wrote config to /etc/ceph/ceph.conf
Wrote keyring to /etc/ceph/ceph.client.admin.keyring
Creating mgr...
Verifying port 0.0.0.0:9283 ...
Verifying port 0.0.0.0:8765 ...
Verifying port 0.0.0.0:8443 ...
firewalld ready
Enabling firewalld service ceph in current zone...
firewalld ready
Enabling firewalld port 9283/tcp in current zone...
Enabling firewalld port 8765/tcp in current zone...
Enabling firewalld port 8443/tcp in current zone...
Waiting for mgr to start...
Waiting for mgr...
mgr not available, waiting (1/15)...
mgr not available, waiting (2/15)...
mgr not available, waiting (3/15)...
mgr is available
Enabling cephadm module...
Waiting for the mgr to restart...
Waiting for mgr epoch 5...
mgr epoch 5 is available
Setting orchestrator backend to cephadm...
Generating ssh key...
Wrote public SSH key to /etc/ceph/ceph.pub
Adding key to root@localhost authorized_keys...
Adding host node01...
Deploying mon service with default placement...
Deploying mgr service with default placement...
Deploying crash service with default placement...
Deploying ceph-exporter service with default placement...
Deploying prometheus service with default placement...
Deploying grafana service with default placement...
Deploying node-exporter service with default placement...
Deploying alertmanager service with default placement...
Enabling the dashboard module...
Waiting for the mgr to restart...
Waiting for mgr epoch 9...
mgr epoch 9 is available
Generating a dashboard self-signed certificate...
Creating initial admin user...
Fetching dashboard port number...
firewalld ready
Ceph Dashboard is now available at:          ------------------------------------------------------------------#管理页面的 URL

             URL: https://node01:8443/
            User: admin
        Password: s7ydpyj0ls

Enabling client.admin keyring and conf on hosts with "admin" label
Saving cluster configuration to /var/lib/ceph/6d6b022e-e689-11ee-b2f5-000c2912f7ba/config directory
Enabling autotune for osd_memory_target
You can access the Ceph CLI as following in case of multi-cluster or non-default config:

        sudo /usr/sbin/cephadm shell --fsid 6d6b022e-e689-11ee-b2f5-000c2912f7ba -c /etc/ceph/ceph.conf -k
/etc/ceph/ceph.client.admin.keyring

Or, if you are only running a single cluster on this host:

        sudo /usr/sbin/cephadm shell

Please consider enabling telemetry to help improve Ceph:

        ceph telemetry on

For more information see:

        https://docs.ceph.com/en/latest/mgr/telemetry/

Bootstrap complete.
```

红色字体部分告诉我们，可以通过 url（https://<ip 地址或者 node01>:8443）打开 ceph 的管理页面，用户名 admin，密码为 s7ydpyj0ls，首次登录需要更换密码，参考图 12-6。

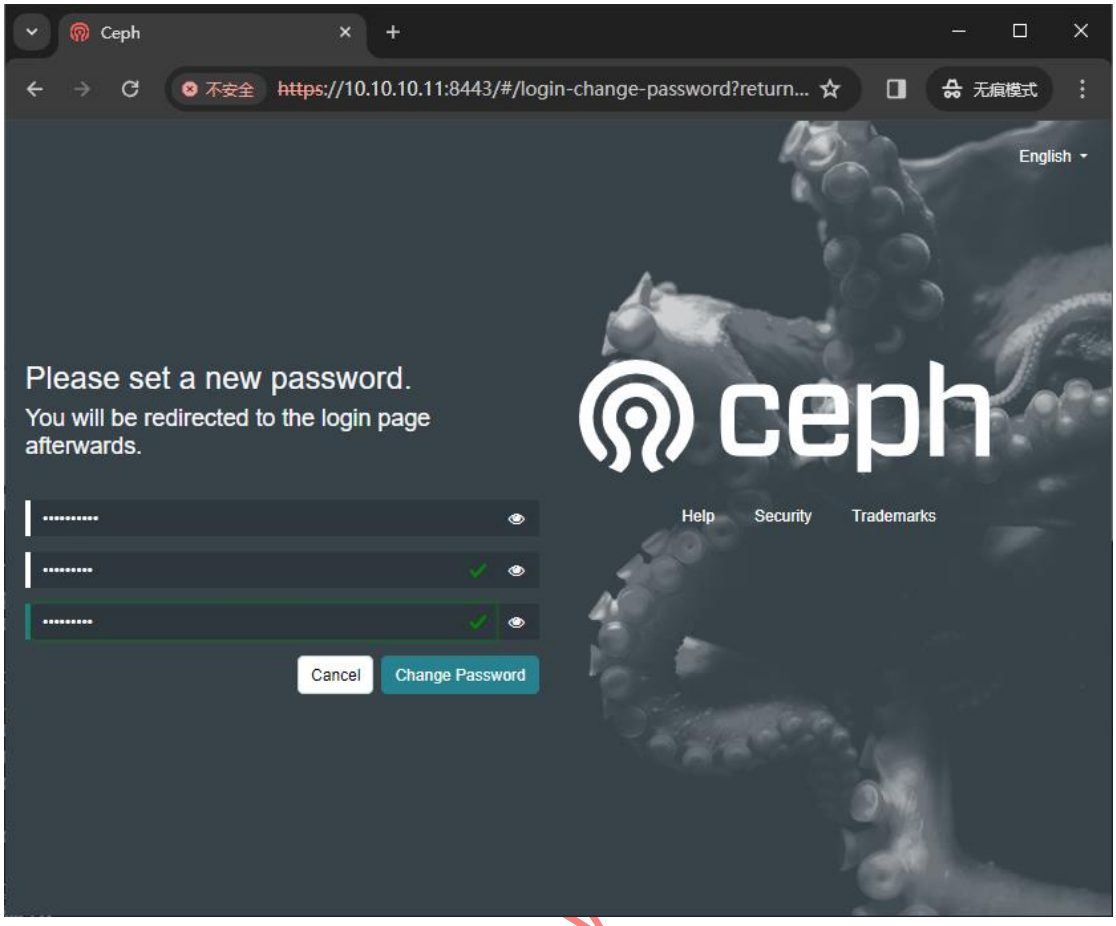


图 12-6 首次登陆后修改密码

如果忘记了密码，可采用如下命令重置密码（比如重置为 A1b2C3789）：

```
[root@node01 ~]# echo "A1b2C3789" > pass.txt
[root@node01 ~]# ceph dashboard ac-user-set-password admin -i pass.txt
```

另外被禁用的用户也是无法登陆的，可采用命令“ceph dashboard ac-user-show admin”查看用户信息，可采用命令“ceph dashboard ac-user-enable admin”激活用户。

另外告诉我们有以下三种命令行方式来管理 ceph 集群。

1. 运行 cephadm shell 命令进入 ceph 的交互模式，如：

```
[root@node01 ~]# cephadm shell
Inferring fsid 6d6b022e-e689-11ee-b2f5-000c2912f7ba
Inferring config /var/lib/ceph/6d6b022e-e689-11ee-b2f5-000c2912f7ba/mon.node01/config
Using ceph image with id 'a27483cc3ea0' and tag 'v18' created on 2024-03-19 16:13:11 +0000 UTC
quay.io/ceph/ceph@sha256:ecca1aa2bb3c4712ff2469b829548965183ccb2c9b78dc73f1fa4caf489caa94
[ceph: root@node01 /]# ceph -s
  cluster:
    id:     f9cd065c-eb18-11ee-a1e4-000c2901cff8
    health: HEALTH_WARN
            OSD count 0 < osd_pool_default_size 3

  services:
    mon: 1 daemons, quorum node01 (age 26m)
    mgr: node01.bvhoxe(active, since 23m)
    osd: 0 osds: 0 up, 0 in

  data:
    pools:   0 pools, 0 pgs
    objects: 0 objects, 0 B
    usage:   0 B used, 0 B / 0 B avail
    pgs:

[ceph: root@node01 /]# exit
[root@node01 ~]#
```

命令后带参数-h 获取特定命令的帮助信息。

2. 添加前缀“cephadm shell -- ”，比如：

```
[root@node01 ~]# cephadm shell -- ceph -v
Inferring fsid 6d6b022e-e689-11ee-b2f5-000c2912f7ba
Inferring config /var/lib/ceph/6d6b022e-e689-11ee-b2f5-000c2912f7ba/mon.node01/config
Using ceph image with id 'a27483cc3ea0' and tag 'v18' created on 2024-03-19 16:13:11 +0000 UTC
quay.io/ceph/ceph@sha256:eccaaa2bb3c4712ff2469b829548965183ccb2c9b78dc73f1fa4caf489caa94
ceph version 18.2.2 (531c0d11a1c5d39fbfe6aa8a521f023abf3bf3e2) reef (stable)
[root@node01 ~]#
```

这样每个命令都要添加长长的前缀，在偶尔执行少量命令时还可以接受。

3. 本地命令模式。不过需要首先在本地安装 ceph-common。如：

```
[root@node01 ~]# cephadm add-repo --release reef
[root@node01 ~]# cephadm install ceph-common
Installing packages ['ceph-common']...
```

或直接采用 dnf install ceph-common 命令安装。以后就可以直接运行 ceph 的各种命令了，如：

```
[root@node01 ~]# ceph -s
  cluster:
    id:     0039db42-d359-11ed-8ab8-000c294bc62c
    health: HELLO_WARN
            OSD count 0 < osd_pool_default_size 3

  services:
    mon: 1 daemons, quorum node01 (age 4m)
    mgr: node01.kaihcq(active, since 4m)
    osd: 0 osds: 0 up, 0 in

  data:
    pools:   0 pools, 0 pgs
    objects: 0 objects, 0 B
    usage:   0 B used, 0 B / 0 B avail
    pgs:

[root@node01 ~]#
```

方法 1 和方法 2 实际上是登录到容器里去执行命令，而方法 3 就在容器的宿主机上执行命令。本实验我们采用方法 1 来操纵 ceph 集群，例如：

```
[root@node01 ~]# cephadm shell
Inferring fsid 6d6b022e-e689-11ee-b2f5-000c2912f7ba
Inferring config /var/lib/ceph/6d6b022e-e689-11ee-b2f5-000c2912f7ba/mon.node01/config
Using ceph image with id 'a27483cc3ea0' and tag 'v18' created on 2024-03-19 16:13:11 +0000 UTC
quay.io/ceph/ceph@sha256:eccaaa2bb3c4712ff2469b829548965183ccb2c9b78dc73f1fa4caf489caa94

查看 ceph 集群状态：
[ceph: root@node01 /]# ceph status
  cluster:
    id:     6d6b022e-e689-11ee-b2f5-000c2912f7ba
    health: HELLO_WARN
            OSD count 0 < osd_pool_default_size 3

  services:
    mon: 1 daemons, quorum node01 (age 44m)
    mgr: node01.bqfkkw(active, since 40m)
    osd: 0 osds: 0 up, 0 in

  data:
    pools:   0 pools, 0 pgs
    objects: 0 objects, 0 B
    usage:   0 B used, 0 B / 0 B avail
    pgs:

查看容器编排服务：
[ceph: root@node01 /]# ceph orch ls
NAME           PORTS        RUNNING  REFRESHED  AGE  PLACEMENT
alertmanager   ?:9093,9094      1/1  9m ago     45m  count:1
ceph-exporter                   1/1  9m ago     45m  *
crash                           1/1  9m ago     45m  *
grafana        ?:3000           1/1  9m ago     45m  count:1
mgr                             1/2  9m ago     45m  count:2
mon                             1/5  9m ago     45m  count:5
node-exporter  ?:9100           1/1  9m ago     45m  *
prometheus     ?:9095           1/1  9m ago     45m  count:1

查看容器：
[ceph: root@node01 /]# ceph orch ps
NAME                 HOST    PORTS            STATUS         REFRESHED  AGE  MEM USE  MEM LIM  VERSION  IMAGE ID      CONTAINER ID
alertmanager.node01  node01  *:9093,9094      running (41m)     9m ago  44m    24.0M        -  0.25.0   c8568f914cd2  345b9aa8f7c0
ceph-exporter.node01 node01                   running (45m)     9m ago  45m    6371k        -  18.2.2   a27483cc3ea0  6b86b67d78f3
crash.node01         node01                   running (45m)     9m ago  45m    6665k        -  18.2.2   a27483cc3ea0  fc6ce0299326
```

```
grafana.node01        node01  *:3000           running (41m)     9m ago  42m    84.0M        -  9.4.7    954c08fa6188  ff703921c6de
mgr.node01.bqfkkw     node01  *:9283,8765,8443  running (46m)     9m ago  46m     474M        -  18.2.2   a27483cc3ea0  e45e7c8f6efc
mon.node01            node01                    running (46m)     9m ago  46m    41.2M    2048M  18.2.2   a27483cc3ea0  fef8aa65d952
node-exporter.node01  node01  *:9100           running (45m)     9m ago  45m    17.0M        -  1.5.0    0da6a335fe13  7eeb83419f8e
prometheus.node01     node01  *:9095           running (41m)     9m ago  41m    39.2M        -  2.43.0   a07b618ecd1d  7a70a6b19a01

从容器中退出：
[ceph: root@node01 /]# exit
exit
[root@node01 ~]#
```

检查防火墙规则：

```
[root@node01 ~]# firewall-cmd --list-all
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: ens160 ens192
  sources:
  services: ceph ceph-mon cockpit dhcpv6-client ntp ssh
  ports: 9283/tcp 8765/tcp 8443/tcp 9100/tcp 9093/tcp 9094/tcp 3000/tcp 9095/tcp
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

检查被监听的端口：

```
[root@node01 ~]# ss -tunlp
Netid  State    Recv-Q   Send-Q     Local Address:Port     Peer Address:Port  Process
udp    UNCONN   0        0              0.0.0.0:123           0.0.0.0:*      users:(("chronyd",pid=863,fd=7))
udp    UNCONN   0        0            127.0.0.1:323           0.0.0.0:*      users:(("chronyd",pid=863,fd=5))
udp    UNCONN   0        0                    *:9094                *:*      users:(("alertmanager",pid=14507,fd=7))
udp    UNCONN   0        0                [::1]:323              [::]:*      users:(("chronyd",pid=863,fd=6))
tcp    LISTEN   0        5          10.10.10.11:7150          0.0.0.0:*      users:(("ceph-mgr",pid=3986,fd=47))
tcp    LISTEN   0        128            0.0.0.0:22            0.0.0.0:*      users:(("sshd",pid=949,fd=3))
tcp    LISTEN   0        512        10.10.10.11:6789          0.0.0.0:*      users:(("ceph-mon",pid=3729,fd=28))
tcp    LISTEN   0        128            0.0.0.0:9926          0.0.0.0:*      users:(("ceph-exporter",pid=10687,fd=22))
tcp    LISTEN   0        512        10.10.10.11:3300          0.0.0.0:*      users:(("ceph-mon",pid=3729,fd=27))
tcp    LISTEN   0        512            0.0.0.0:6801          0.0.0.0:*      users:(("ceph-mgr",pid=3986,fd=28))
tcp    LISTEN   0        512            0.0.0.0:6800          0.0.0.0:*      users:(("ceph-mgr",pid=3986,fd=27))
tcp    LISTEN   0        5          10.10.10.11:8765          0.0.0.0:*      users:(("ceph-mgr",pid=3986,fd=50))
tcp    LISTEN   0        4096                 *:9093                *:*      users:(("alertmanager",pid=14507,fd=8))
tcp    LISTEN   0        4096                 *:9095                *:*      users:(("prometheus",pid=12247,fd=7))
tcp    LISTEN   0        4096                 *:9094                *:*      users:(("alertmanager",pid=14507,fd=3))
tcp    LISTEN   0        4096                 *:9100                *:*      users:(("node_exporter",pid=11208,fd=3))
tcp    LISTEN   0        5                    *:8443                *:*      users:(("ceph-mgr",pid=3986,fd=45))
tcp    LISTEN   0        128              [::]:22               [::]:*      users:(("sshd",pid=949,fd=4))
tcp    LISTEN   0        5                    *:9283                *:*      users:(("ceph-mgr",pid=3986,fd=38))
tcp    LISTEN   0        4096                 *:3000                *:*      users:(("grafana",pid=14787,fd=13))
```

建议给 node01 创建快照“before add host”，以便后续操作错误可随时回滚到此。

§12.2.2 添加主机

加入集群的主机，需要满足：python 3、Systemd、podman 或者 docker、与第一个节点时钟同步、lvm2。

1. 分别在 node02 和 node03 上安装 podman：

```
[root@node02 ~]# dnf install podman -y
[root@node03 ~]# dnf install podman -y
```

2. 把集群公钥从 node01 拷贝到其他节点：

```
[root@node01 ~]# ssh-copy-id -f -i /etc/ceph/ceph.pub root@node02
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/etc/ceph/ceph.pub"
The authenticity of host 'node02 (10.10.10.12)' can't be established.
ED25519 key fingerprint is SHA256:EKcuxlM/bk7YDmlV2V/nHjiMuaJ5AYS2X4Cb2xIuPf8.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
root@node02's password:

Number of key(s) added: 1
```

```
Now try logging into the machine, with:   "ssh 'root@node02'"
and check to make sure that only the key(s) you wanted were added.

[root@node01 ~]# ssh-copy-id -f -i /etc/ceph/ceph.pub root@node03
```

3．把主机 node02 和 node03 加入集群：

```
[root@node01 ~]# cephadm shell
[ceph: root@node01 /]# ceph orch host add node02 10.10.10.12 --labels _admin
Added host 'node02' with addr '10.10.10.12'
```

加入主机时若带标签_admin，那么会把第一个节点上的 ceph.conf 和 client.admin 的键文件（放在/etc/ceph 目录下）拷贝到 node02。大概过一分钟，就能在 node02 执行 ceph 命令了。也可以在加入主机之后再增加添标签（采用命令 ceph orch host label add <主机名> <标签>）。

继续加入 node03：

```
[ceph: root@node01 /]# ceph orch host add node03 10.10.10.13
Added host 'node03' with addr '10.10.10.13'
```

列出集群中的主机：

```
[ceph: root@node01 /]# ceph orch host ls
HOST    ADDR          LABELS  STATUS
node01  10.10.10.11  _admin
node02  10.10.10.12  _admin
node03  10.10.10.13
3 hosts in cluster
```

4．把 mon 数量调整为 3 个：

```
[ceph: root@node01 /]# ceph orch ls
NAME           PORTS        RUNNING  REFRESHED  AGE  PLACEMENT
alertmanager   ?:9093,9094      1/1  4m ago     80m  count:1
ceph-exporter                   1/2  4m ago     80m  *
crash                           1/2  4m ago     80m  *
grafana        ?:3000           1/1  4m ago     80m  count:1
mgr                             1/2  4m ago     80m  count:2
mon                             1/5  4m ago     80m  count:5
node-exporter  ?:9100           1/2  4m ago     80m  *
prometheus     ?:9095           1/1  4m ago     80m  count:1
```

发现 mon 服务默认需要 5 个，现在只部署了 3 个节点，我们没有这么多的节点。需要调整为 3 个 mon，执行下面的命令进行调整：

```
[ceph: root@node01 /]# ceph orch apply mon node01,node02,node03
Scheduled mon update...
```

当然我们也可以指定 mon 的数量（采用命令 ceph orch apply mon 3）。

5．浏览管理页面

此时在页面上看到三个节点的状况（打开 https://10.10.10.11:8443/），登录用户名 admin，密码见集群初始化的输出，第一次登陆需要修改密码。参考图 12-7。

https://www.

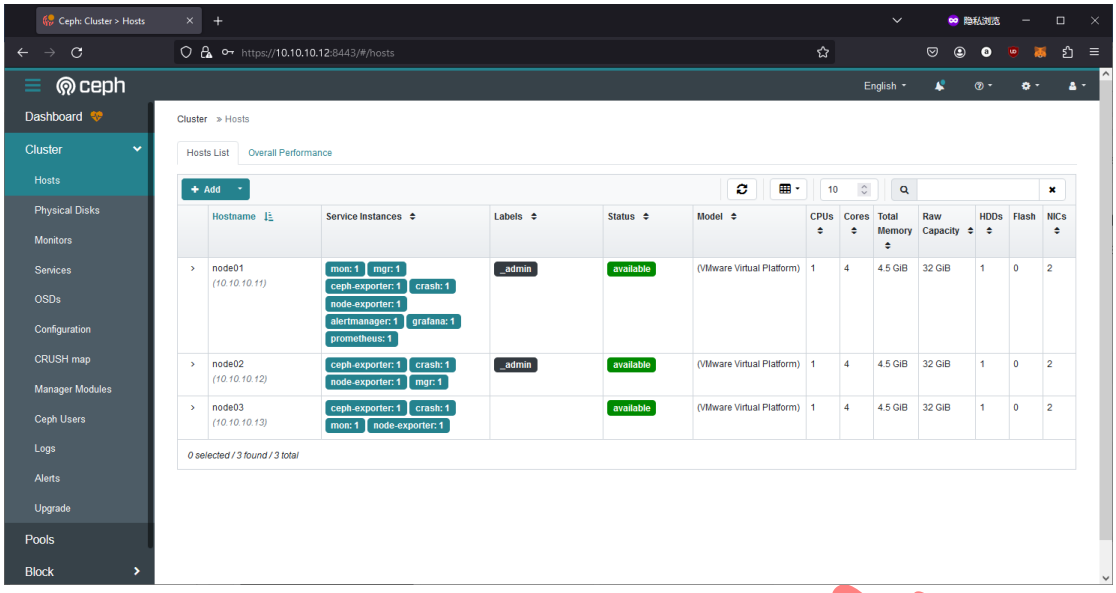


图 12-7 节点上的服务

添加节点后会自动部署 mon 和 mgr 服务。如果在某个节点上暂时没法发现相应的服务，那就等几分钟后再刷新页面看看，因为安装这些服务软件是要耗费一些时间的。

命令 ceph orch host -h 获取操纵主机的帮助信息。

如果可行，现在再给三个节点新建一个快照，快照取名“before_add_osd”。

§ 12.2.3 添加 OSD

OSD 主机要求有一块还没有分区的硬盘，硬盘至少 5GB。关闭三个节点，然后各增加一块 16GB 的磁盘。

1. 列出磁盘及各个磁盘上的分区：

```
[root@node01 ~]# lsblk
NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
sda           8:0    0   32G  0 disk
├─sda1        8:1    0    1G  0 part /boot
└─sda2        8:2    0   31G  0 part
  ├─cs-root 253:0    0 27.8G  0 lvm  /var/lib/containers/storage/overlay
  │                                 /
  └─cs-swap 253:1    0  3.2G  0 lvm  [SWAP]
sdb           8:16   0   16G  0 disk
sr0          11:0    1  9.1G  0 rom

[root@node01 ~]# ceph orch device ls
HOST    PATH      TYPE  DEVICE ID                                             SIZE  AVAILABLE  REFRESHED  REJECT
REASONS
node01  /dev/sdb  hdd   ATA_VMware_Virtual_SATA_Hard_Drive_0200000000001  8192M  Yes        4m ago
node02  /dev/sdb  hdd   ATA_VMware_Virtual_SATA_Hard_Drive_0200000000001  8192M  Yes        4m ago
node03  /dev/sdb  hdd   ATA_VMware_Virtual_SATA_Hard_Drive_0200000000001  8192M  Yes        6s ago
```

第二块磁盘的名字是 sdb，对应的设备文件名为/dev/sdb。AVAILABLE 列为 yes 表明节点上的磁盘/dev/sdb 是可用的。上面第二条命令需要等几分钟才能看到三个节点上的磁盘。

2. 确保两块网卡都配置了 ip 并且已经启用：

```
[root@allnodes ~]# ip a
……
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:0c:29:12:f7:ba brd ff:ff:ff:ff:ff:ff
    altname enp3s0
```

https://www.veryopen.

```
    inet 10.10.10.11/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe12:f7ba/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:0c:29:12:f7:c4 brd ff:ff:ff:ff:ff:ff
    altname enp11s0
    inet 20.20.20.11/24 brd 20.20.20.255 scope global noprefixroute ens192
       valid_lft forever preferred_lft forever
    inet6 fe80::bf21:29d8:ce0e:b56/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

3．把 node01 作为 OSD 加入集群：

```
[root@node01 ~]# cephadm shell
[ceph: root@node01 /]# ceph orch daemon add osd node01:/dev/sdb
Created osd(s) 0 on host 'node01'

[ceph: root@node01 /]# ceph -s
......
  services:
    mon: 3 daemons, quorum node01,node02,node03 (age 22m)
    mgr: node01.kaihcq(active, since 68m), standbys: node02.cnmuxc
    osd: 1 osds: 1 up (since 63s), 1 in (since 77s)
......
```

把剩下的节点都当成 OSD 节点：

```
[ceph: root@node01 /]# ceph orch apply osd --all-available-devices
Scheduled osd.all-available-devices update...
```

4．再检查下还有没有可用的磁盘：

```
[root@node01 ~]# ceph orch device ls
HOST    PATH      TYPE  DEVICE ID                                             SIZE  AVAILABLE  REFRESHED  REJECT REASONS
node01  /dev/sdb  hdd   ATA_VMware_Virtual_SATA_Hard_Drive_02000000000001  8192M  No         4s ago     Has a FileSystem, Insufficient space (<10 extents) on vgs, LVM
detected
node02  /dev/sdb  hdd   ATA_VMware_Virtual_SATA_Hard_Drive_02000000000001  8192M  No         20m ago    Has a FileSystem, Insufficient space (<10 extents) on vgs, LVM
detected
node03  /dev/sdb  hdd   ATA_VMware_Virtual_SATA_Hard_Drive_02000000000001  8192M  No         20m ago    Has a FileSystem, Insufficient space (<10 extents) on vgs, LVM
detected
```

表明三个节点上再也没有可用的磁盘了。如果以后又增加了磁盘，可以继续加入集群。

过一会儿的页面参见图 12-8 所示。

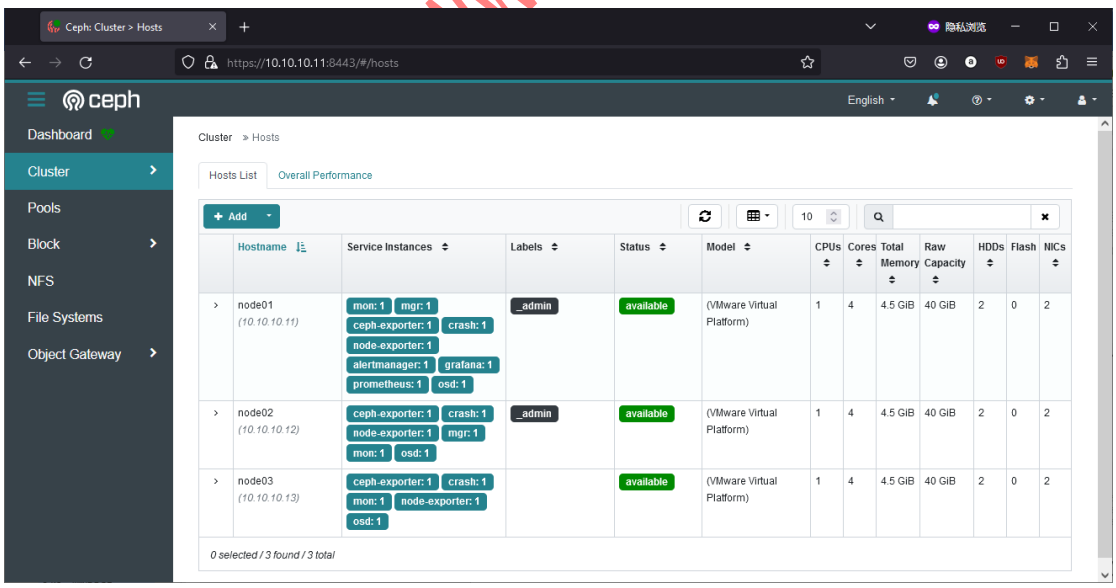


图 12-8 加入 OSD 后各个节点上的服务

发现每个节点上都运行了一个 osd 服务和一个 mon 服务。

查看容器：

```
[root@node01 ~]# ceph orch ps
NAME                 HOST    PORTS             STATUS         REFRESHED  AGE  MEM USE  MEM LIM  VERSION  IMAGE ID      CONTAINER ID
alertmanager.node01  node01  *:9093,9094       running (8m)     7m ago   2h    15.1M        -  0.25.0   c8568f914cd2  844b6e70c33c
ceph-exporter.node01 node01                    running (8m)     7m ago   2h    6941k        -  18.2.2   a27483cc3ea0  597736b12220
```

```
ceph-exporter.node02  node02                  running (7m)     7m ago   2h    6521k        -  18.2.2   a27483cc3ea0  b603350a60cd
ceph-exporter.node03  node03                  running (7m)     6m ago   2h    6471k        -  18.2.2   a27483cc3ea0  67ec6d91382b
crash.node01          node01                  running (8m)     7m ago   2h    6665k        -  18.2.2   a27483cc3ea0  a847421f40a6
crash.node02          node02                  running (7m)     7m ago   2h    24.7M        -  18.2.2   a27483cc3ea0  da0f85d6adf5
crash.node03          node03                  running (7m)     6m ago   2h    6669k        -  18.2.2   a27483cc3ea0  52ced8b859d2
grafana.node01        node01  *:3000          running (8m)     7m ago   2h    73.8M        -  9.4.7    954c08fa6188  659c62c9d21e
mgr.node01.pndpcg     node01  *:9283,8765,8443  running (8m)     7m ago   2h     457M        -  18.2.2   a27483cc3ea0  3e53c5f679a7
mgr.node02.cugzwf     node02  *:8443,9283,8765  running (7m)     7m ago   2h     285M        -  18.2.2   a27483cc3ea0  768b22bb9c3b
mon.node01            node01                  running (8m)     7m ago   2h    27.8M    2048M  18.2.2   a27483cc3ea0  6dbb293346fb
mon.node02            node02                  running (7m)     7m ago   2h    20.1M    2048M  18.2.2   a27483cc3ea0  bd41c0b1acba
mon.node03            node03                  running (7m)     6m ago   2h    27.0M    2048M  18.2.2   a27483cc3ea0  1429bf8bf750
node-exporter.node01  node01  *:9100          running (8m)     7m ago   2h    5322k        -  1.5.0    0da6a335fe13  7c73b922839d
node-exporter.node02  node02  *:9100          running (7m)     7m ago   2h    5460k        -  1.5.0    0da6a335fe13  e351c07a7f67
node-exporter.node03  node03  *:9100          running (7m)     6m ago   2h    7612k        -  1.5.0    0da6a335fe13  b018a7ff140b
osd.0                 node01                  running (8m)     7m ago  34m    48.7M    4096M  18.2.2   a27483cc3ea0  5b0059770afa
osd.1                 node02                  running (7m)     7m ago  24m    21.7M    4096M  18.2.2   a27483cc3ea0  ab8ba07bdbbb
osd.2                 node03                  running (7m)     6m ago  24m    50.9M    4096M  18.2.2   a27483cc3ea0  8b5e832a32fb
prometheus.node01     node01  *:9095          running (8m)     7m ago   2h    34.1M        -  2.43.0   a07b618ecd1d  7733dc7b59ef
```

其中容器名 osd.0、osd.1、osd.2 就是三个 osd 服务，后缀的那个数字代表第几号 osd 服务。

查看 osd 树：

```
[root@node01 ~]# ceph osd tree
ID  CLASS  WEIGHT   TYPE NAME         STATUS  REWEIGHT  PRI-AFF
-1         0.02339  root default
-3         0.00780      host node01
 0    hdd  0.00780          osd.0         up   1.00000  1.00000
-5         0.00780      host node02
 1    hdd  0.00780          osd.1         up   1.00000  1.00000
-7         0.00780      host node03
 2    hdd  0.00780          osd.2         up   1.00000  1.00000
```

可用命令 ceph orch osd rm 0 删除 0 号 osd。

最后检查一下集群状态：

```
[root@node01 ~]# ceph -s
  cluster:
    id:     9c891b16-e71c-11ee-b8b5-000c2912f7ba
    health: HEALTH_OK

  services:
    mon: 3 daemons, quorum node01,node02,node03 (age 8m)
    mgr: node01.pndpcg(active, since 9m), standbys: node02.cugzwf
    osd: 3 osds: 3 up (since 8m), 3 in (since 10m)

  data:
    pools:   1 pools, 1 pgs
    objects: 2 objects, 577 KiB
    usage:   81 MiB used, 24 GiB / 24 GiB avail
    pgs:     1 active+clean
```

ceph osd -h 可获取二级子命令 osd 的全部三级子命令和帮助信息。

https://w

§12.3 使用 ceph 存储

建议从 node01 的第一个快照克隆一台虚拟机来做客户机，当然也可以直接拿 node02 或 node03 做客户机。我们这里单独克隆一台客户机，取名 client，只有一块 NAT 网卡，ip 为 10.10.10.21，参考图 12-6 所示。

§12.3.1 使用 Ceph 块设备

就是把分布式存储空间映射成客户机上的一块磁盘，然后再对此磁盘分区、格式化、挂载并使用。

操作时要注意命令行的提示符，不要在错误的电脑上操作。

1．客户机上安装相应软件：

```
[root@client ~]# dnf -y install rocky-release-ceph-reef epel-release
[root@client ~]# dnf -y install ceph-common
```

2．从 node01 拷贝有关文件给客户机：

```
[root@node01 ~]# scp /etc/ceph/ceph.conf 10.10.10.21:/etc/ceph/
[root@node01 ~]# scp /etc/ceph/ceph.client.admin.keyring 10.10.10.21:/etc/ceph/
[root@client ~]# chown ceph. /etc/ceph/ceph.*
```

3．创建磁盘并挂载

(1)．创建一个池，池的名字为 rbd1，内有 32 个 pg：

```
[root@client ~]# ceph osd pool create rbd1 32
pool 'rbd1' created
```

(2)．启用归置组（Placement Groups）自动缩放模式（其实默认已经开启）：

```
[root@client ~]# ceph osd pool set rbd1 pg_autoscale_mode on
set pool 2 pg_autoscale_mode to on
```

(3)．初始化池

```
[root@client ~]# rbd pool init rbd1
[root@client ~]# ceph osd pool autoscale-status
POOL                SIZE  TARGET SIZE  RATE  RAW CAPACITY   RATIO  TARGET RATIO  EFFECTIVE RATIO  BIAS  PG_NUM  NEW PG_NUM  AUTOSCALE  BULK
.mgr              452.0k                3.0        45044M  0.0000                                  1.0       1              on         False
rbd1              340.1M                3.0        45044M  0.0227                                  1.0      32              on         False
```

上面第一条命令如果长时间没完成，可以肯定 ceph 集群有问题，在 node01 上执行命令 ceph -s 查看集群状态，正常情况会看到 health: HEALTH_OK，否则检查第二块网卡有没有启动（ip a 查看），能否 ping 20.20.20.12，如果第二块网卡没启动，请查看虚拟机网卡的“设备状态”，参考图 12-9 操作。

https://www.yeryon.org

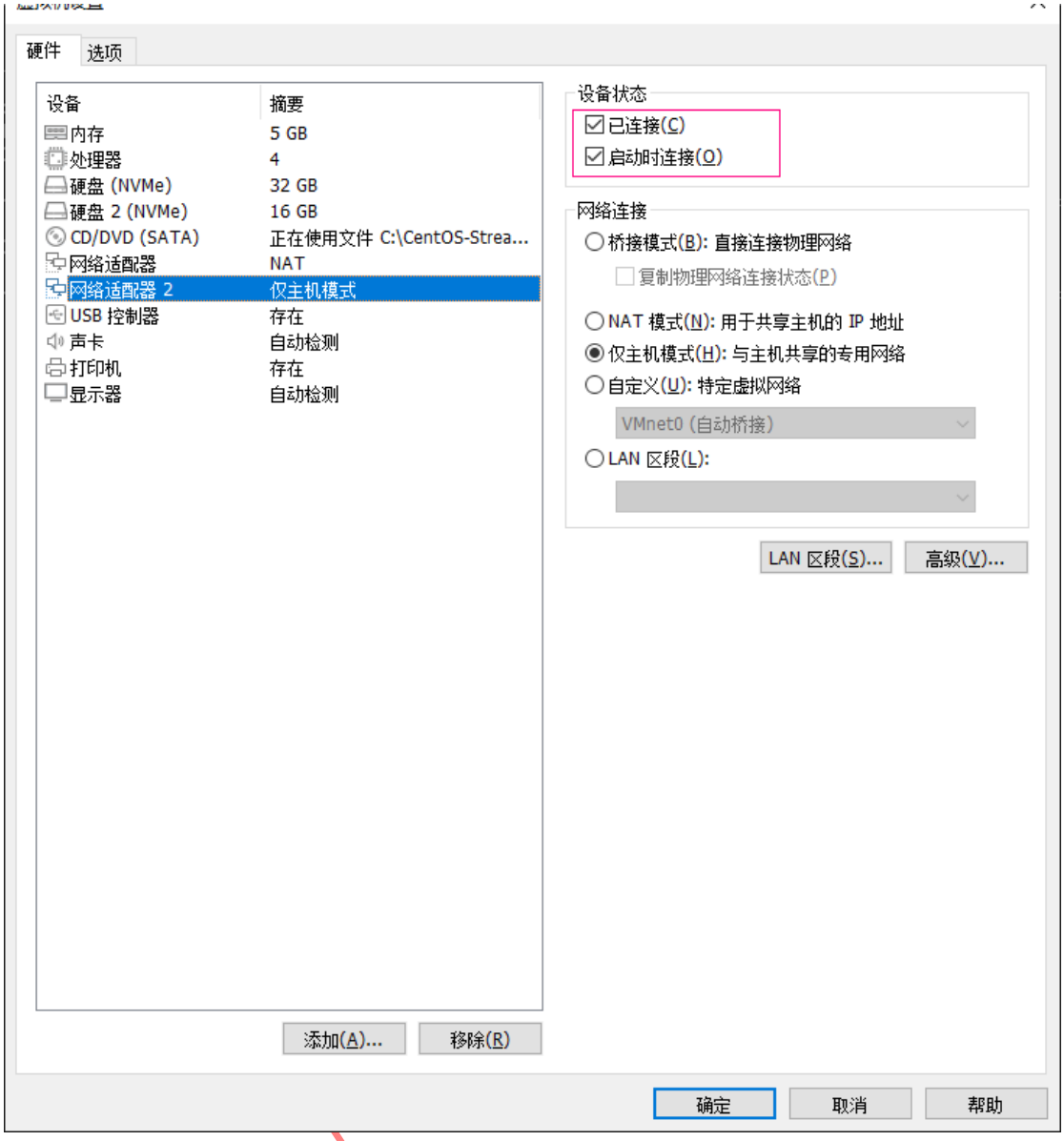


图 12-9 虚拟机网卡的状态

还有就是查看 ceph 的日志文件/var/log/ceph/cephadm.log。

(4). 在池 rbd1 中创建磁盘块设备（4GB 容量），名称为 rbd101：

```
[root@client ~]# rbd create --size 4G --pool rbd1 rbd101
[root@client ~]# rbd ls --pool rbd1 -l
NAME    SIZE   PARENT  FMT  PROT  LOCK
rbd101  4 GiB            2
```

(5). 把 rbd101 映射成/dev/rdb0：

```
[root@client ~]# rbd map --pool rbd1 rbd101
/dev/rbd0
[root@client ~]# rbd showmapped
id  pool  namespace  image   snap  device
0   rbd1             rbd101  -     /dev/rbd0
```

现在我们有一个新磁盘，设备文件名是/dev/rdb0，接下来就可以对它进行分区或者直接格式化。

(6). 格式化磁盘为 xfs 文件系统：

```
[root@client ~]# mkfs.xfs /dev/rbd0
meta-data=/dev/rbd0              isize=512    agcount=8, agsize=131072 blks
```

```
         =                       sectsz=512   attr=2, projid32bit=1
         =                       crc=1        finobt=1, sparse=1, rmapbt=0
         =                       reflink=1    bigtime=0 inobtcount=0
data     =                       bsize=4096   blocks=1048576, imaxpct=25
         =                       sunit=16     swidth=16 blks
naming   =version 2              bsize=4096   ascii-ci=0, ftype=1
log      =internal log           bsize=4096   blocks=2560, version=2
         =                       sectsz=512   sunit=16 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
Discarding blocks...Done.
```

(7). 挂载磁盘到/mnt 目录（或者任何一个空目录）上：

```
[root@client ~]# mount /dev/rbd0 /mnt
[root@client ~]# lsblk

NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda           8:0    0   20G  0 disk
├─sda1        8:1    0    1G  0 part /boot
└─sda2        8:2    0   19G  0 part
  ├─cs-root 253:0    0   17G  0 lvm  /
  └─cs-swap 253:1    0    2G  0 lvm  [SWAP]
sr0          11:0    1  852M  0 rom
rbd0        252:0    0    4G  0 disk /mnt

[root@client ~]# df -T
文件系统            类型         1K-块     已用      可用 已用% 挂载点
devtmpfs            devtmpfs      4096        0      4096    0% /dev
tmpfs               tmpfs      2370336        0   2370336    0% /dev/shm
tmpfs               tmpfs       948136     9268    938868    1% /run
/dev/mapper/cs-root xfs       29077504  2362808  26714696    9% /
/dev/sda1           xfs         983040   287920    695120   30% /boot
tmpfs               tmpfs       474064        0    474064    0% /run/user/0
/dev/rbd0           xfs        4128768    62096   4066672    2% /mnt
```

此后进入/mnt 目录等于进入了硬盘中，往/mnt 目录拷贝文件等于就是把文件存储在 ceph 构建的分布式存储上。如把/boot 目录下的大文件拷贝到/mnt 目录：

```
[root@client ~]# cp /boot/init* /mnt/
```

如果想让电脑开机时自动挂载到目录/opt/ceph123 上，可以这样做：

```
[root@client ~]# mkdir /opt/ceph123
[root@client ~]# echo '/dev/rbd0 /opt/ceph123 xfs defaults,noatime,_netdev 0 0' >> /etc/fstab

[root@client ~]# echo "rbd1/rbd101 id=admin,keyring=/etc/ceph/ceph.client.admin.keyring" >> /etc/ceph/rbdmap
[root@client ~]# systemctl enable rbdmap.service
```

然后重启电脑看看是不是自动挂载了。

如果要删除块设备，请参考下面的命令：

```
[root@client ~]# umount /mnt                                                                    #卸载
[root@client ~]# rbd unmap /dev/rbd/rbd1/rbd101 ------------------------------------------------------------------------ #取消块设备映射
[root@client ~]# rbd rm rbd101 --pool rbd1                                                      #删除块设备
[root@client ~]# ceph osd pool delete rbd1 rbd1 --yes-i-really-really-mean-it                   #删除 OSD 池
Error EPERM: pool deletion is disabled; you must first set the mon_allow_pool_delete config ...
```

报错，不能删除，需要把 mon_allow_pool_delete 参数设为 true 才行：

```
[root@node01 ~]# ceph --admin-daemon  /var/run/ceph/0039db42-d359-11ed-8ab8-000c294bc62c/ceph-mon.node01.asok \
 config show|grep mon_allow_pool_delete
“mon_allow_pool_delete”: “false”,

[root@node01 ~]# ceph tell mon.* injectargs --mon_allow_pool_delete=true
```

然后就可以删除 OSD 池了。

§12.3.2 使用 Ceph 文件系统

就是把分布式存储上的目录挂载到客户机上。Ceph 文件系统需要用到 MDS（元数据服务器），文件名、文件大小、权限、主人等统称为文件的员数据，元数据通过 MDS 单独存放在员数据池中。而文件的数据存放在数据池中，哭护短读/写文件的具体数据时直接访问数据池。参考图 12-10 所示。

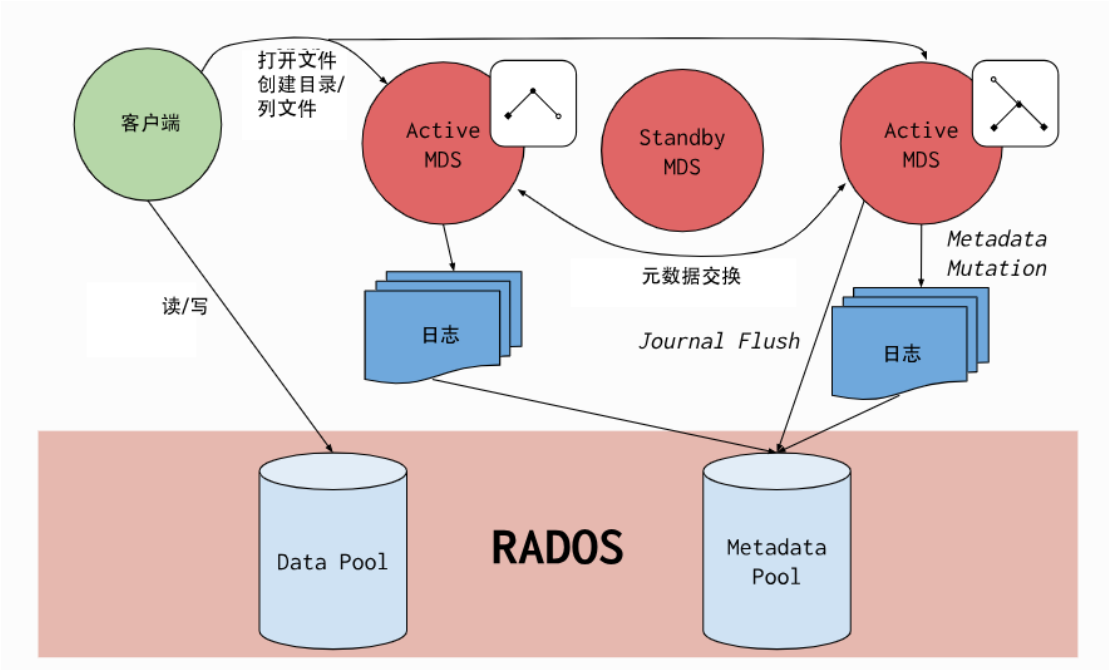


图 12-10 访问文件系统

1. 创建 ceph 文件系统，一个 Ceph 默认智能创建一个文件系统：

```
[root@node01 ~]# cephadm shell
[ceph: root@node01 /]# ceph fs volume create cephfs1 --placement=3
[ceph: root@node01 /]# ceph fs volume ls
[
    {
        "name": "cephfs1"
    }
]
[ceph: root@node01 /]# ceph fs status cephfs1
cephfs1 - 0 clients
RANK  STATE            MDS               ACTIVITY     DNS    INOS   DIRS   CAPS
 0    active  cephfs1.node01.ovgvtk  Reqs:    0 /s    10     13     12      0
       POOL           TYPE     USED  AVAIL
cephfs.cephfs1.meta  metadata  96.0k  7765M
cephfs.cephfs1.data    data       0   7765M
     STANDBY MDS
cephfs1.node03.iijkte
cephfs1.node02.kowmls
MDS version: ceph version 17.2.5 (98318ae89f1a893a6ded3a640405cdbb33e08757) quincy (stable)
```

参数--placement 指明使用几个 MDS 服务，MDS 就是文件系统的元数据服务。除了指明元数据服务的个数，还可以指定元数据服务泡在哪些节点上，更详细的说明请参考 https://docs.ceph.com/en/latest/cephadm/services/#orchestrator-cli-placement-spec。

2. 在客户机上挂载 Ceph 文件系统：

```
(1). 客户机上安装相应软件:
[root@client ~]# dnf -y install cephadm
[root@client ~]# cephadm install ceph-fuse

(2). 获取配置文件:
[root@client ~]# scp node01:/etc/ceph/ceph.conf /etc/ceph/
[root@client ~]# scp node01:/etc/ceph/ceph.client.admin.keyring /etc/ceph/
[root@client ~]# chown ceph. /etc/ceph/ceph.*
```

```
(3). 挂载 ceph 文件系统:
[root@client ~]# mkdir /opt/mycephfs
[root@client ~]# ceph-fuse /opt/mycephfs
2024-03-21T16:14:46.828+0800 7f558dcda480 -1 init, newargv = 0x5607eace72a0 newargc=13
ceph-fuse[2024-03-21T16:14:46.828+0800 7f558dcda480 -1 init, args.argv = 0x5607eace09f0 args.argc=4
3517]: starting ceph client
ceph-fuse[3517]: starting fuse

[root@client ~]# df -T
文件系统             类型              1K-块     已用      可用 已用% 挂载点
devtmpfs             devtmpfs         977432        0    977432    0% /dev
tmpfs                tmpfs            996008        0    996008    0% /dev/shm
tmpfs                tmpfs            996008     9188    986820    1% /run
tmpfs                tmpfs            996008        0    996008    0% /sys/fs/cgroup
/dev/mapper/rl-root xfs             29128812 2810308 26318504   10% /
/dev/nvme0n1p1       xfs             1038336  210976   827360   21% /boot
tmpfs                tmpfs            199200        0    199200    0% /run/user/0
ceph-fuse            fuse.ceph-fuse  7946240        0  7946240    0% /opt/mycephfs
```

如果让开机时自动挂载 ceph 文件系统，请在/etc/fstab 文件中加入下面一行：

```
none    /opt/mycephfs  fuse.ceph ceph.id=admin,_netdev,defaults  0 0
```

这行的完整语法如下：

```
none  挂载目录  fuse.ceph ceph.id={user-ID}[,ceph.conf={path/to/conf.conf}],_netdev,defaults  0 0
```

注意：其中 user-id 只是用户名（比如 admin），而不是类似 client.*（比如 client.admin），请查看 /etc/ceph/ceph.client.admin.keyring 文件。

接下来访问挂载目录就是访问 ceph 构建的分布式存储了，比如往此目录拷贝文件就是把文件保存在 Ceph 集群中了。

要卸载 Ceph 文件系统，请参考下面的命令：

```
[root@client ~]# umount /opt/mycephfs

[root@node01 ~]# ceph config set mon mon_allow_pool_delete true
[root@node01 ~]# ceph fs volume rm cephfs1 --yes-i-really-mean-it
```

§12.3.3 使用对象存储

Ceph 对象存储完全兼容亚马逊的 S3。

随便拿一个节点做对象存储的网关，不过建议还是单独用一台电脑做网关，取名为 objgate，ip 地址为设为 10.10.10.20，请参考图 12-6。

1. 安装有关软件：

```
[root@objgate ~]# dnf -y install rocky-release-ceph-reef epel-release                #添加 ceph 的安装源
[root@objgate ~]# dnf -y install ceph-radosgw                                        #安装对象网关
```

2. 在 Ceph 集群中配置网关：

```
[root@node01 ~]# vim /etc/ceph/ceph.conf                                             #在文件末尾增加下面的内容
# client.rgw.(Node Name)
[client.rgw.www]
# 网关的 ip
host = 10.10.10.20
# 网关的主机名
rgw dns name = objgate
keyring = /var/lib/ceph/radosgw/ceph-rgw.www/keyring
log file = /var/log/ceph/radosgw.gateway.log

[root@node01 ~]# echo "10.10.10.20 objgate" >> /etc/hosts ------------------------------------------------ #文件末尾增加一行
[root@node01 ~]# scp /etc/ceph/ceph.conf objgate:/etc/ceph/
[root@node01 ~]# scp /etc/ceph/ceph.client.admin.keyring objgate:/etc/ceph/
```

https://www.vryopen.org

3．在网关上配置：

```
[root@objgate ~]# mkdir -p /var/lib/ceph/radosgw/ceph-rgw.www
[root@objgate ~]# ceph auth get-or-create client.rgw.www osd 'allow rwx' mon 'allow rw' -o /var/lib/ceph/radosgw/ceph-rgw.www/keyring
[root@objgate ~]# chown ceph. /etc/ceph/ceph.*
[root@objgate ~]# chown -R ceph. /var/lib/ceph/radosgw

[root@objgate ~]# systemctl --now enable ceph-radosgw@rgw.www

[root@objgate ~]# firewall-cmd --add-port=7480/tcp
[root@objgate ~]# firewall-cmd --runtime-to-permanent
```

4．测试：

```
[root@node01 ~]# curl objgate:7480
<?xml version="1.0" encoding="UTF-8"?><ListAllMyBucketsResult xmlns="http://s3.amazonaws.com/doc/2006-03-
01/"><Owner><ID>anonymous</ID><DisplayName></DisplayName></Owner><Buckets></Buckets></ListAllMyBucketsResult>
```

5．在对象网关（objgate）上创建 S3 兼容的用户：

```
[root@objgate ~]# radosgw-admin user create --uid=cephuser1 --display-name="Ceph User" --email=admin@veryopen.org
{
    "user_id": "cephuser1",
    "display_name": "Ceph User",
    "email": "admin@veryopen.org",
    "suspended": 0,
    "max_buckets": 1000,
    "subusers": [],
    "keys": [
        {
            "user": "cephuser1",
            "access_key": "GMG3T0SNNWUZICFHILF4",
            "secret_key": "bCpG8tBiG1ymYxWfcrqxod4xpL5DCBTBOj1cFHq3"
        }
    ],
    "swift_keys": [],
    "caps": [],
    "op_mask": "read, write, delete",
    "default_placement": "",
    "default_storage_class": "",
    "placement_tags": [],
    "bucket_quota": {
        "enabled": false,
        "check_on_raw": false,
        "max_size": -1,
        "max_size_kb": 0,
        "max_objects": -1
    },
    "user_quota": {
        "enabled": false,
        "check_on_raw": false,
        "max_size": -1,
        "max_size_kb": 0,
        "max_objects": -1
    },
    "temp_url_keys": [],
    "type": "rgw",
    "mfa_ids": []
}

[root@objgate ~]# radosgw-admin user list ---------- #列出用户
[
    "cephuser1"
]

[root@objgate ~]# radosgw-admin user info --uid=cephuser1                    #显示用户的信息
{
    "user_id": "cephuser1",
    "display_name": "Ceph User",
    "email": "admin@veryopen.org",
    "suspended": 0,
    "max_buckets": 1000,
    "subusers": [],
    "keys": [
        {
            "user": "cephuser1",
            "access_key": "GMG3T0SNNWUZICFHILF4",
            "secret_key": "bCpG8tBiG1ymYxWfcrqxod4xpL5DCBTBOj1cFHq3"
        }
    ],
.....
```

6．在另一台电脑（如 client）上测试能否使用对象存储：

```
[root@client ~]# dnf -y install python3-pip
[root@client ~]# pip3 install boto3
[root@client ~]# vim s3_test.py                                    #编辑一个 python 程序
import sys
import boto3
from botocore.config import Config
```

```
# 使用在第 5 步中新建 用户的 Id 和密码:
session = boto3.session.Session(
    aws_access_key_id = 'GMG3T0SNNWUZICFHILF4',
    aws_secret_access_key = 'bCpG8tBiG1ymYxWfcrqxod4xpL5DCBTBOj1cFHq3'
)

# 对象网关的 URL:
s3client = session.client(
    's3',
    endpoint_url = 'http://10.10.10.20:7480',
    config = Config()
)

# 创建桶 [my-new-bucket]
bucket = s3client.create_bucket(Bucket = 'my-new-bucket')

# 列出 Buckets
print(s3client.list_buckets())

# 删除 [my-new-bucket]
s3client.delete_bucket(Bucket = 'my-new-bucket')


[root@client ~]# python s3_test.py
{'ResponseMetadata': {'RequestId': 'tx0000084499fe06a1a29fb-0065fe85c6-196e0-default', 'HostId': '',
'HTTPStatusCode': 200, 'HTTPHeaders': {'transfer-encoding': 'chunked', 'x-amz-request-id':
'tx0000084499fe06a1a29fb-0065fe85c6-196e0-default', 'content-type': 'application/xml', 'date': 'Sat, 23 Mar 2024
07:33:26 GMT', 'connection': 'Keep-Alive'}, 'RetryAttempts': 0}, 'Buckets': [{'Name': 'my-new-bucket',
'CreationDate': datetime.datetime(2024, 3, 23, 7, 33, 26, 763000, tzinfo=tzutc())}], 'Owner': {'DisplayName': 'Ceph
User', 'ID': 'cephuser1'}}
```

本测试完成这些任务：创建桶 my-new-bucket，然后列出存在的桶，最后删除桶 my-new-bucket。

也可以在管理页面管理对象存储，参见图 12-11。

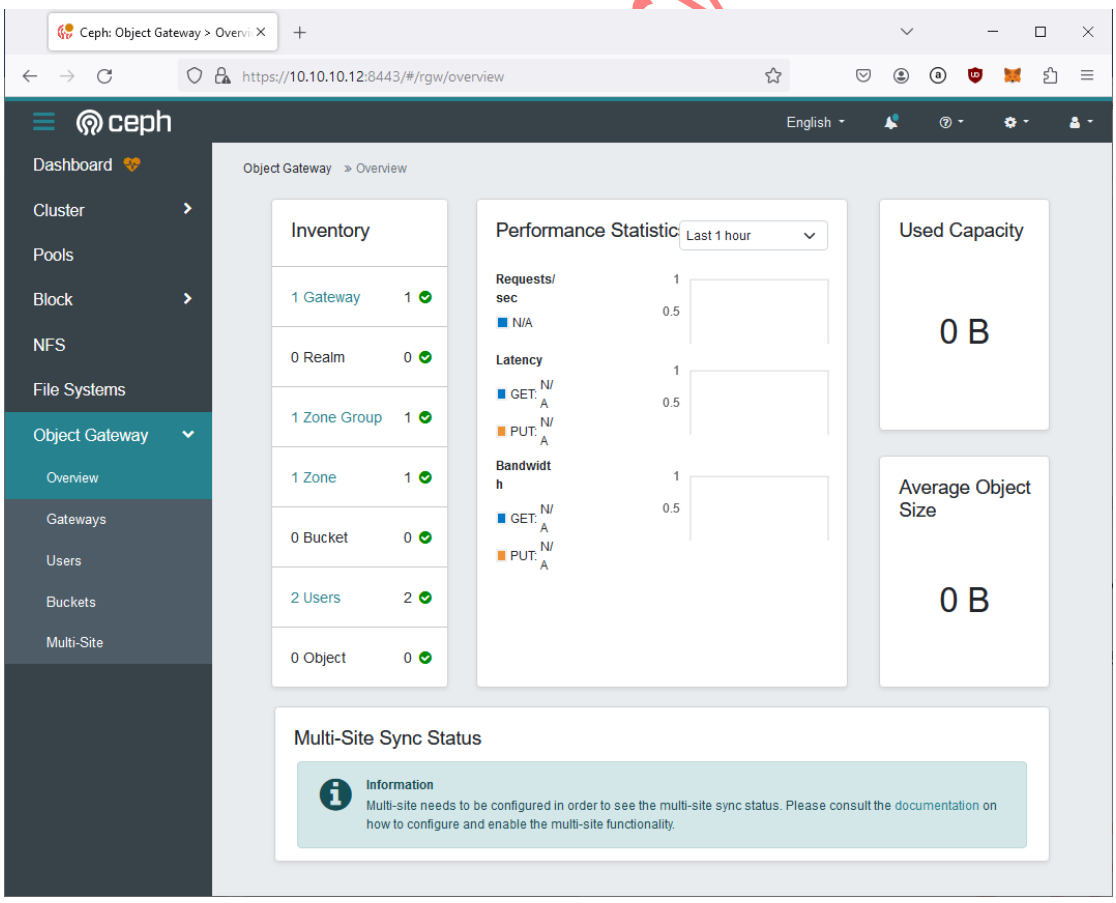


图 12-11 对象存储的管理页面

参考图 12-12 创建一个 bucket。

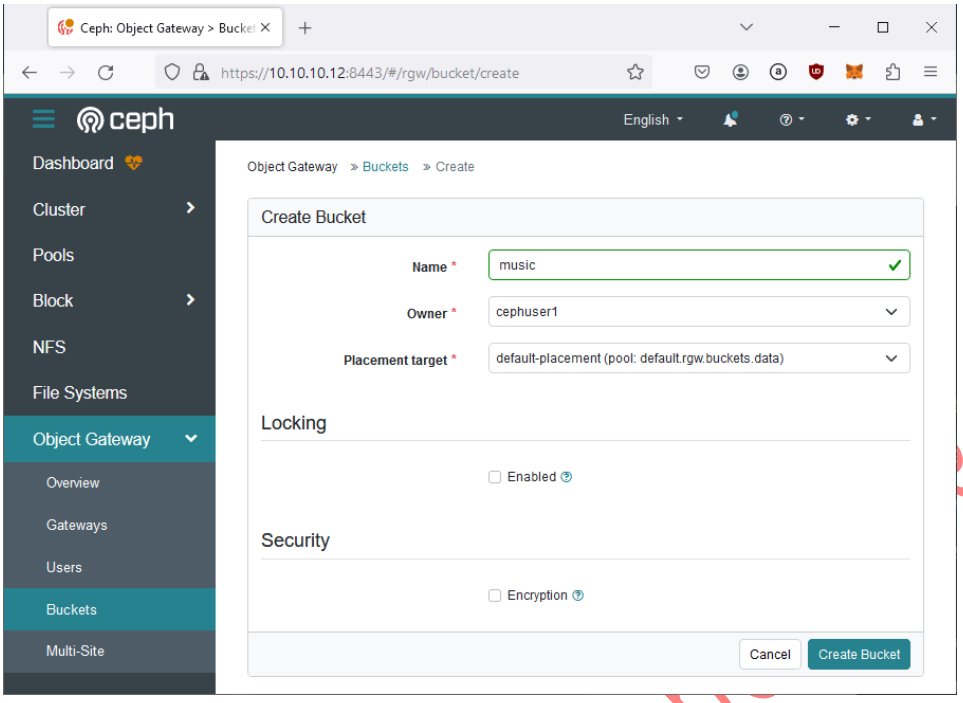


图 12-12 创建 bucket

https://www.veryop

§12.4 管理 Ceph

可采用命令方式管理，也可以登录管理页面进行可视化管理。

§12.4.1 管理命令

一切从命令“ceph -h”开始。

§12.4.2 管理页面

一切从浏览“https://node01:8443”开始。

参考 https://access.redhat.com/documentation/zh-cn/red_hat_ceph_storage/7/html/dashboard_guide/

§12.4.3 监控 Ceph

采用 cephadm 部署 Ceph 集群，默认使用 Grafana、Prometheus 来监控 Ceph 集群，Ceph、Prometheus 和 Grafana 之间的关系见图 12-13 所示。

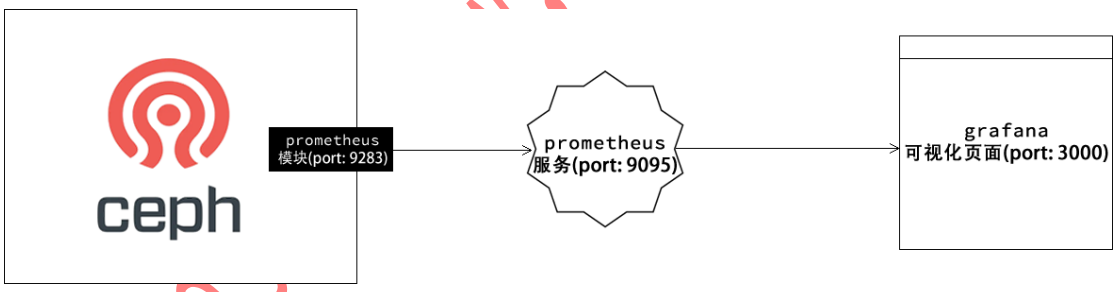


图 12-13 Ceph，Ceph、Prometheus 和 Grafana 的关系

浏览 http://10.10.10.11:9283/查看 Prometheus 模块收集的 ceph 状态参数，浏览 http://10.10.10.11:9095 查看 Prometheus 服务收集的各种数据，参见图 12-14 所示。

https://www.veryopen.org

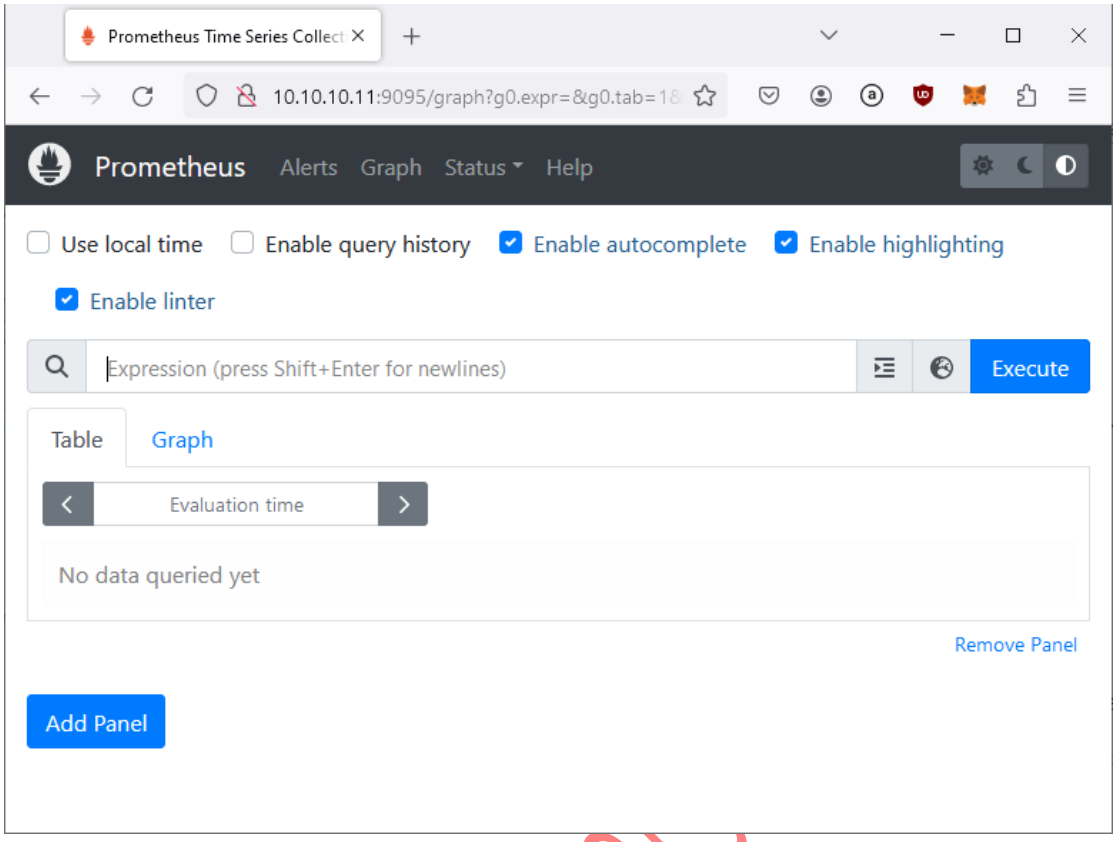


图 12-14 Prometheus 服务收集的数据

点击图中的地球图标选择参数，然后点击“Execute”按钮，可看到参数的值。浏览 https://10.10.10.11:3000 打开 Grafana 可视化页面，参考图 12-15 所示。

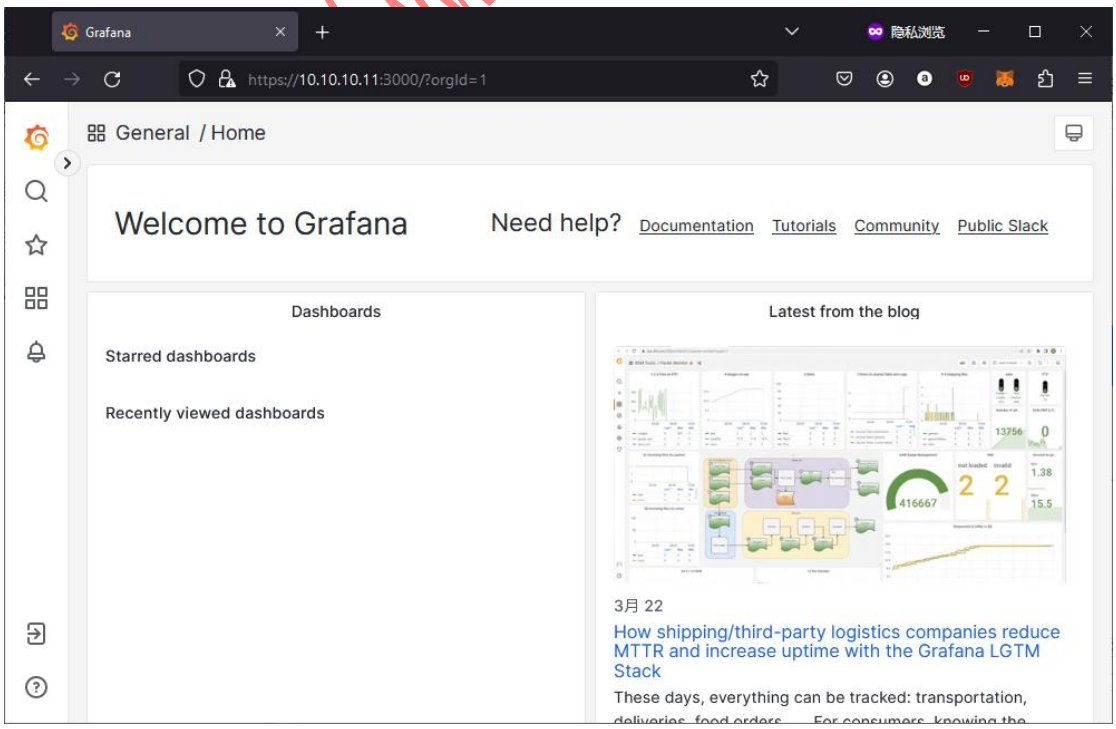


图 12-15 Grafana 页面

点击左侧的“ ”图标，然后点击 ceph-dashboard 目录下的某个项目，比如“Ceph Cluster - Advanced”，打开如图 12-16 所示的页面。

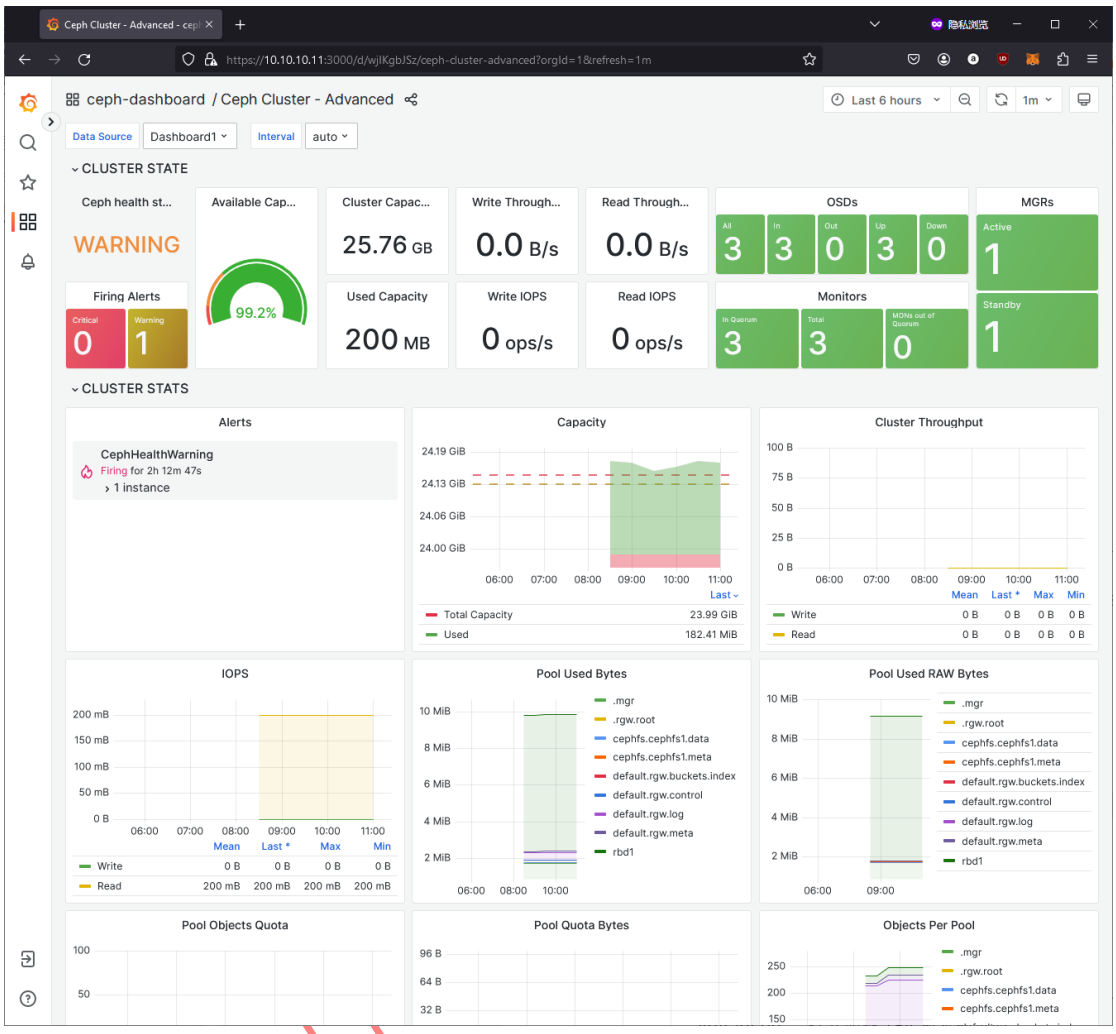


图 12-16 监控 Ceph 集群

点击上图左侧的“ ”图标展开侧边栏，然后点击 Sign in 打开登录页面，参见图 12-17 所示。

https://

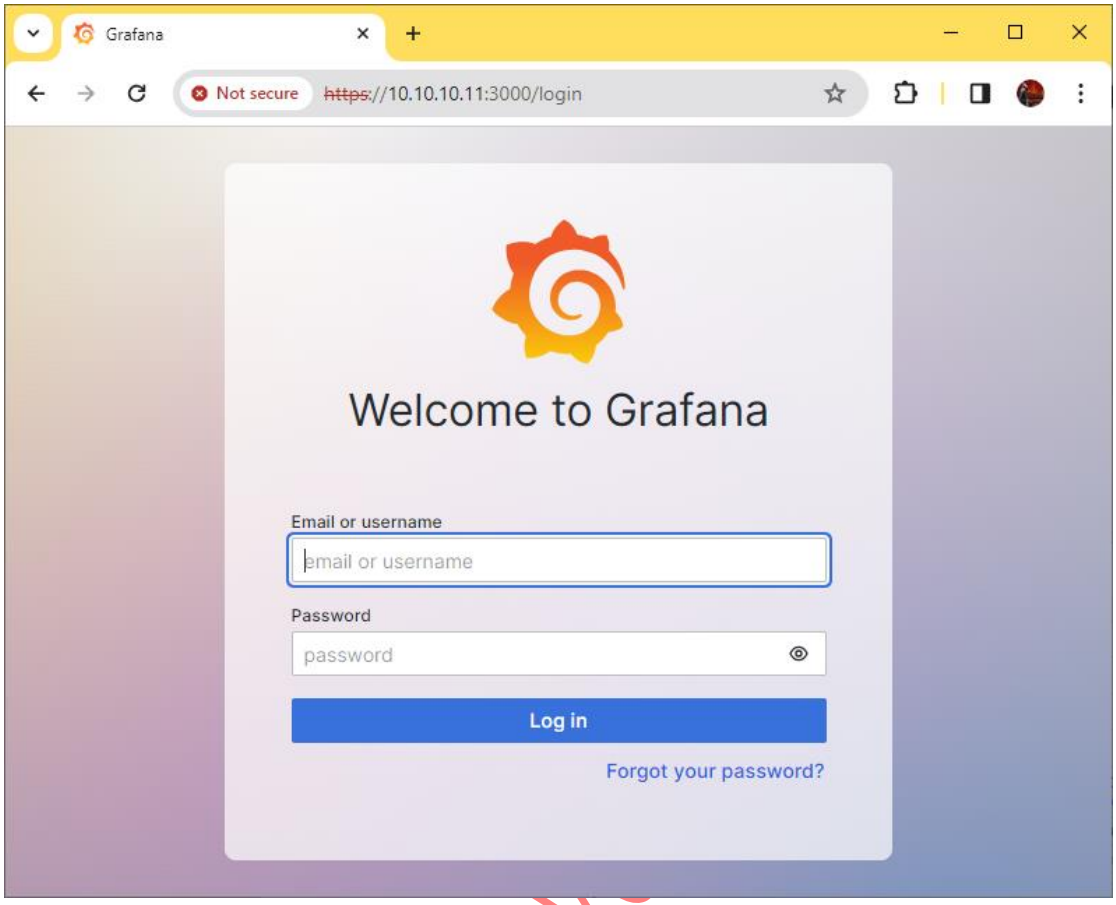


图 12-17 登录 Grafana 页面

用户名为 admin，但是不知道密码。下面我们来重设密码。

1. 新建文件 grafana.yml，定义密码为 A1b2C3：

```
[root@node01 ~]# vim grafana.yml
service_type: grafana
spec:
  initial_admin_password: A1b2C3
```

2. 将 grafana.yml 文件挂载到容器的一个目录中：

```
[root@node01 ~]# cephadm shell --mount grafana.yml:/var/lib/ceph/grafana.yml
Inferring fsid 9c891b16-e71c-11ee-b8b5-000c2912f7ba
Inferring config /var/lib/ceph/9c891b16-e71c-11ee-b8b5-000c2912f7ba/mon.node01/config
Using ceph image with id 'a27483cc3ea0' and tag 'v18' created on 2024-03-19 16:13:11 +0000 UTC
quay.io/ceph/ceph@sha256:798f1b1e71ca1bbf76c687d8bcf5cd3e88640f044513ae55a0fb571502ae641f
[ceph: root@node01 /]#
```

3. 使用 orch 命令应用规格：

```
[ceph: root@node01 /]# ceph orch apply -i /var/lib/ceph/grafana.yml
Scheduled grafana update...
```

4. 重新部署 grafana 服务：

```
[ceph: root@node01 /]# ceph orch redeploy grafana
Scheduled to redeploy grafana.node01 on host 'node01'
[ceph: root@node01 /]# exit
exit
[root@node01 ~]#
```

好了，现在刷新图 12-17 所示的登录页面，输入用户名 admin，密码 A1b2C3 就可以登录了，登录后发现左侧边栏多了不少菜单，见图 12-18 所示。

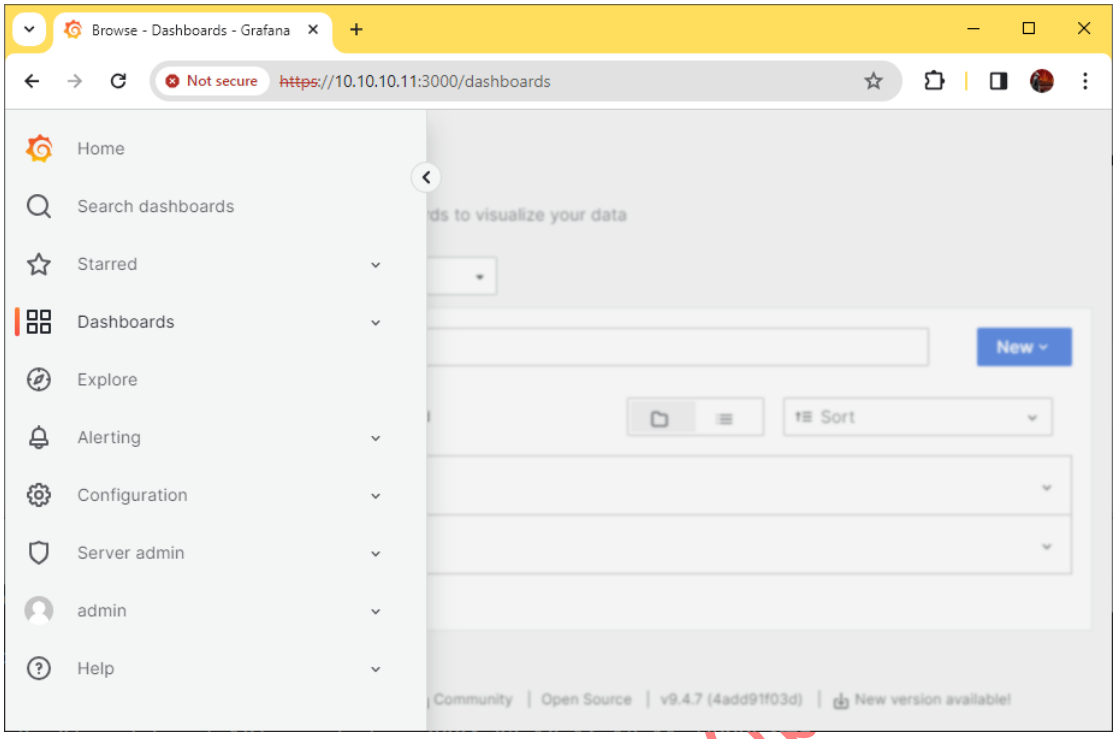


图 12-18 登录后的可视化页面

下面来汉化它，店家左侧边栏“admin”》“Profile”，然后在 Language 下面选择“中文（简体）”，最后点击“Save”按钮保存即可。

https://www.verycd

------- 下篇 -------

正式实战 kubernetes 了。

https://www.veryopen.org

第13章 Kubernetes 简介



https://www.veryopen.

§13.1 体系结构

k8s 是负责容器调度的“操作系统”，正如 OpenStack 是负责虚拟机调度的“操作系统”，以及 Linux 和 windows 是负责进程调度的操作系统一样。

“容器调度”就是根据用户的要求在合适的节点上启动容器，并提供用户访问容器的方式。启动一个容器的大致步骤是：用户通过 RESTful API 调用向 k8s 请求启动容器，然后 k8s 寻找合适的节点并在此节点上启动容器，最后按预先的配置参数构建网络通路（如负载均衡器），这样外部的用户就可以访问容器了。

“寻找合适的节点”真正体现了“调度”二字的含义，即采用何种策略选择节点，比如可能是这样的策略：满足资源（内存、磁盘、cpu、网络、安全性等）要求的最空闲的那个节点，或者满足资源需要并且最繁忙的那个节点。前者尽量让所有节点承担平均的负载，体现公平性，而后者尽量让已经启动的节点满负荷运转，而后根据需要次第启动更多的节点，做到绿色计算从而节省电力能源。

为了履行好“调度”这个中心任务，k8s 需要管理资源（登记、分配、回收等）、管理节点（假如、退出、健康检查等）、与用户交互等。容器调度包括启动、停止、迁移、创建和销毁容器等。在 kubernetes 中，资源统称为资源对象。

k8s 集群中的电脑（称为节点）允许承担两种角色中的一种或两种，即 control-plane 角色或者 worker 角色，前者控制整个集群，后者用于跑 pod，pod 为容器的分组。因此一台电脑就可构造最小的集群，最新版（1.31）的 k8s 集群中最多存在 5000 个工作节点，每个工作节点最多跑 110 个 pod，总的 pod 上限为 150000 个，总容器数不能超过 300000 个(参考 https://kubernetes.io/docs/setup/best-practices/cluster-large/)。集群、节点、pod、容器和网络名字空间的关系参考示意图 13-1。

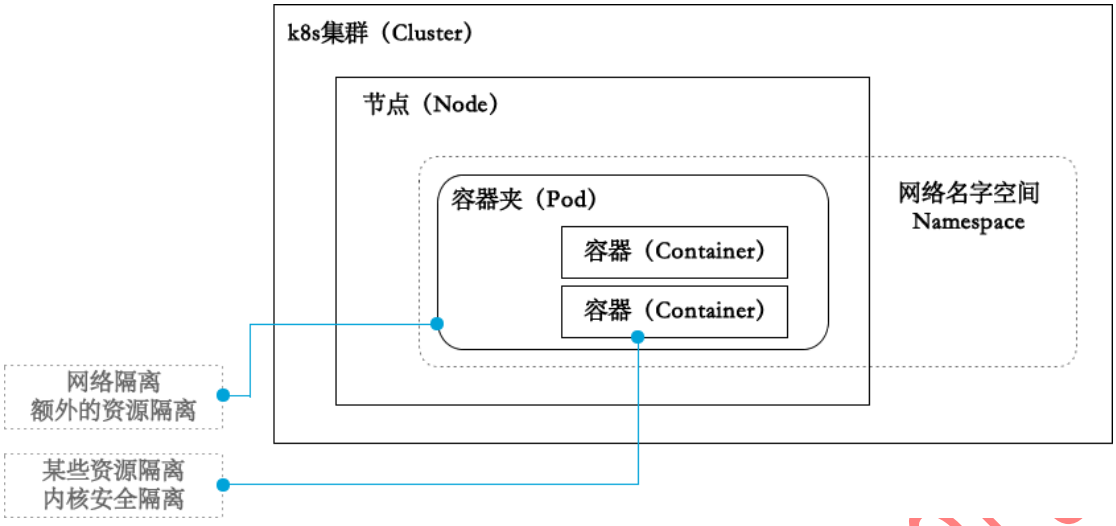


图 13-1 Cluster、Node 和 Pod 的关系

一个 k8s 集群中至少存在一个节点充当 control-plane 角色，至少存在一个节点充当 worker 节点，充当 control-plane 角色的节点称为控制面节点，而充当 worker 角色的节点称为工作节点。控制面节点上跑 kubernetes 的几个核心组件，相当于 k8s 集群的调度指挥中心，而工作节点上跑各种应用程序的 pod，相当于工厂的车间。

即一个 k8s 集群包含多个节点（包含控制面节点和工作节点），每个节点上运行多个 pod，每个 pod 包含多个容器，每个容器中跑若干应用，而 pod 是 kubernetes 调度的基本单位。最小的 k8s 集群就是一个节点（课采用 minikube 快速搭建只有一个节点的集群）。

Pod 可理解为 K8s 集群上的虚拟机，一个 pod 包含多个容器。一般地，一个 pod 跑一个应用，一个容器跑一个服务。我们都知道，一个应用往往包含多个服务，比如企业的门户网站这个应用，至少包含两个服务——web 服务和数据库服务，因此一个 pod 中往往包含多个容器。每个 pod 一般在独立的网络名字空间内，pod 内的容器共享网络协议栈，就像这些容器中的服务运行在同一台电脑上一样，直接通过 127.0.0.1 也能互相通信，因此同一个 pod 内各个容器中的服务监听的端口不能相同。k8s 管理的对象就是 pod，就像 linux 操作系统管理的对象是进程，OpenStack 管理的对象是虚拟机那样。

在生产环境中，往往采用三个或以上节点专司 control-plane 角色，以冗余换取高可用性，只要不同时坏，集群控制面就是好的。其他大量的节点都充当 worker 角色，工作节点通过负载均衡器访问集群控制面。同时可单独部署 ectd 集群，ectd 是 noSQL 型的键/值对数据库，用来存储 k8s 集群数据。图 13-2 是高可用性的 k8s 集群拓扑图（etcd 集成到控制面节点上）。

https://www.verypen.com

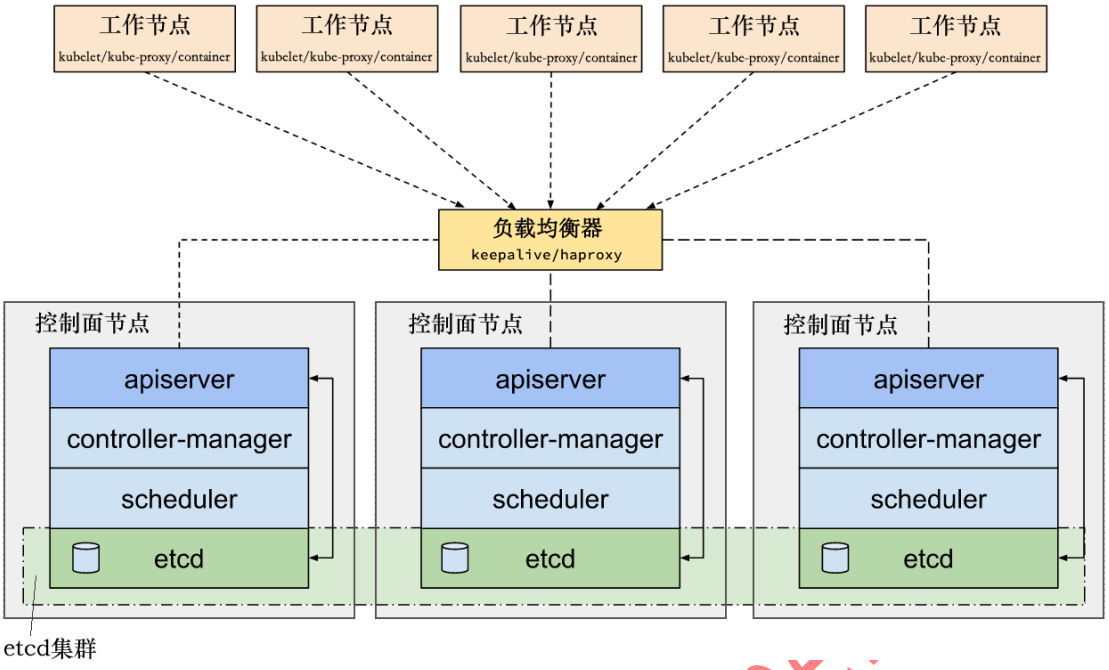


图 13-2 高可用性 k8s 集群（集成 etcd）

图 13-3 是另一中常用的 k8s 高可用性集群拓扑图，etcd 集群独立开来。

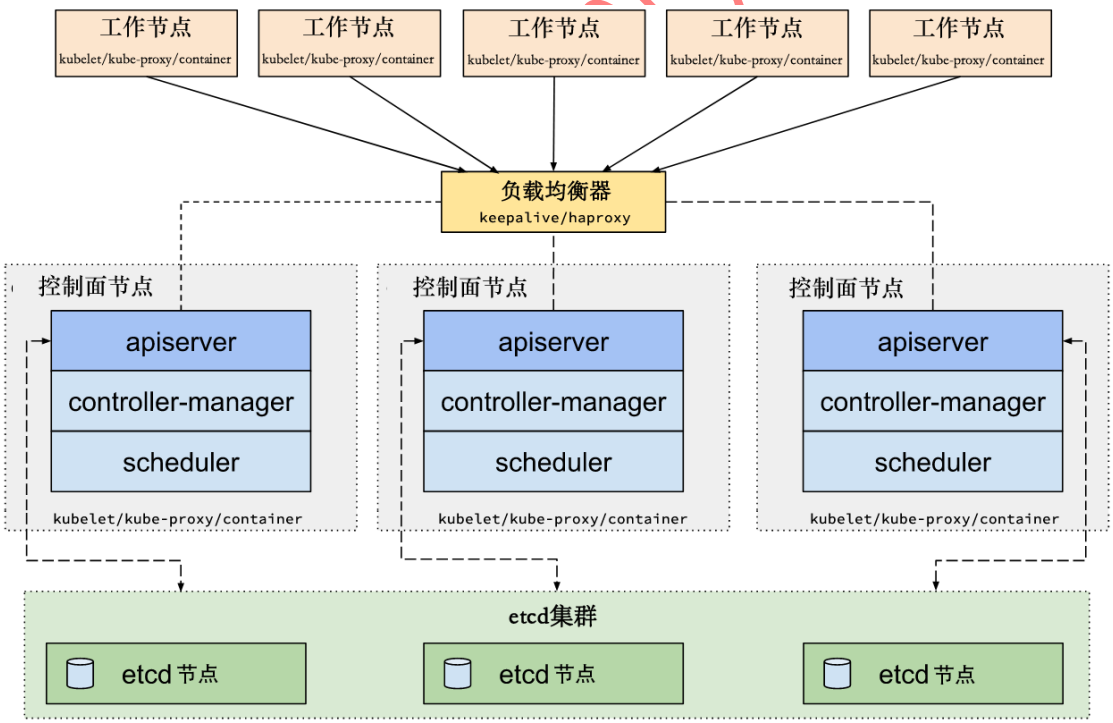


图 13-3 高可用性 k8s 集群（独立 etcd）

在生产环境中，究竟采用哪种高可用性架构与很多因素有关，如果预算充足，可以采用独立 etcd 的架构，这样就多购买三台电脑。如果控制面节点配置不高，再跑 etcd 很卡的话，最好也把 etcd 独立开来。

§13.2 K8s 中的组件

图 13-4 中的黑色背景框就是 kubernetes 的组件，其他的是第三方软件。在 k8s 集群中，除了 kubernetes 组件外，一些第三方软件必不可少，比如数据库（如 etcd）、容器运行时（如 containerd）、网络插件（如 Flannel）等。

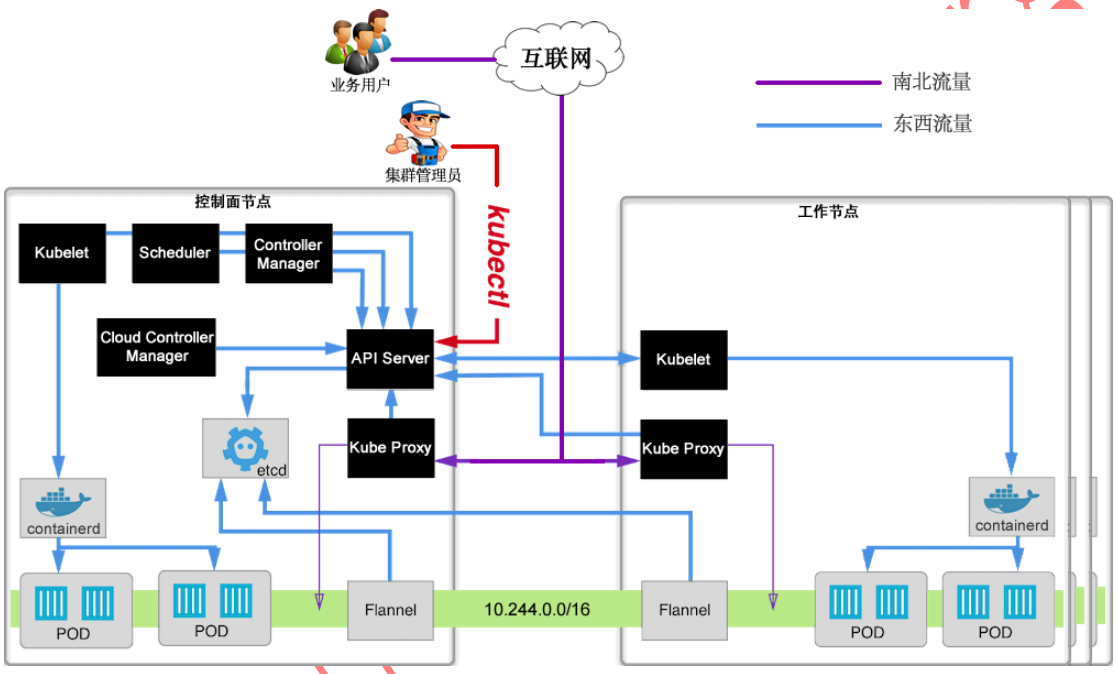


图 13-4 kubernetes 组件

在生产环境中，南北流量和东西流量一般走不同的物理网卡。其中 API Server、Controller Manager、Scheduller、Kube Proxy、Cloud Controller Manager 和 etcd 分别运行在 kube-system 名字空间内各自的 pod 中。

下面各节介绍 kubernetes 组件。

§13.2.1 API Server

对应服务进程 kube-apiserver，此进程对外暴露 RESTful API，调用这些 api 完成各种资源对象的增/删/改/查操作和 watch 事件中继（把 etcd 的事件转发给其他组件）任务。API Server 成为集群内各个组件之间数据交互和

通信的中心枢纽，充当集群的数据总线和数据中心，参考图 13-4 所示。

§13.2.2 Controller Manager

对应的服务进程为 Kube-controller-manager，算是 kubernetes 的大脑，使用事件驱动机制，通过监听资源变化来触发各种操作——etcd 把被监视键的修改事件推送到 API Server，然后 API Server 再把事件转发给 Controller Manager，最后分派给具体控制器以完成某项任务。Controller Manager 管理一系列的控制器（内置的控制器或者用户自定义的控制器），一个控制器至少追踪一种类型的 Kubernetes 资源对象，这些资源对象有一个代表期望状态的字段，该资源的控制器确保其当前状态时时接近期望状态，就像空调总是确保室温接近设定的温度那样。

控制器管理器是通过定期重复执行下面三个步骤来完成控制任务的：

(1) 从 API Server 读取资源对象的期望状态和当前状态。
(2) 比较两者的差异，然后运行控制器操纵现实中的资源对象，将资源对象的真实状态调整为资源对象的 Spec 中定义的期望状态。
(3) 调整成功后，将结果状态写回到在 API Server 上的目标资源对象的 status 字段中。

由 Controller Manager 管理的控制器见表 13-1。

表 13-1 控制器

控制器名称	说明
bootstrap-signer-controller	采用共享令牌完成 ConfigMap 资源对象的签名，ConfigMap 资源对象在 kube-public 名字空间中。本控制器默认不启用。
certificatesigningrequest-approving-controller	检查 klusterlet 发起的 CSR 请求是否可以自动签发。
certificatesigningrequest-cleaner-controller	证书签名请求清理控制器会定期删除这些 CSR：(1) 已批准、已拒绝、未颁发证书或既未批准也未拒绝但已过时；(2) 已批准但证书已过期。
certificatesigningrequest-signing-controller	在 Kubernetes 控制器管理器的启动和运行过程中，提供证书的签署、批准、清理和颁发等相关功能，确保集群中的组件具有有效的证书，保证了集群的安全性和稳定性。
cloud-node-lifecycle-controller	
clusterrole-aggregation-controller	
cronjob-controller	负责处理周期性的任务。
daemonset-controller	确保全部节点上运行一个 Pod 的副本。当有节点加入集群时，会自动为它们新增一个 Pod，当有节点从集群移除时，这些 Pod 也会被回收。删除 DaemonSet 将会删除它创建的所有 Pod
deployment-controller	根据 Deployment 中描述的期望状态，本控制器采用一定的策略将 ReplicaSet 与 Pod 更新到管理员预期的状态。
disruption-controller	监视 Pod 和 PDB（即 PodDisruptionBudget 对象），当监听到 pod/pdb 的 Add/Del/Update 事件后，将对应的 pdb 对象放到限速队列中等待工作节点处理。限速队列的特点是：需要周期性遍历执行，执行完毕将会再次执行，执行失败将延时再次执行。
endpoints-controller	监听 Service 和对应的 Pod 副本的变化，当监测到 Service 被删除，则删除和该 Service 同名的 Endpoint 对象。如果监测到新的 Service 被创建或者修改，则根据该 Service 获得相关的 Pod 列表，然后创建或者更新 Service 对应的 Endpoint 对象。
endpointslice-controller	
endpointslice-mirroring-controller	
ephemeral-volume-controller	
garbage-collector-controller	清理垃圾资源，回收没有属主的资源。
horizontal-pod-autoscaler-controller	根据 CPU 使用率或内存使用率等指标自动扩展部署中的 pod 副本数量。要使用 HPA，您需要在 k8s 集群中定义水平 pod 自动缩放器资源，指定您要扩展的部署以及要用于触发扩展的指标以及可以拥有的最小和最大副本数。
job-controller	管理一次性的批量处理任务。
namespace-controller	监视监控名字空间的状态，在其失效的情况下，对其进行处理。控制器定时通过 API Server 读取名字空间信息，如果某个名字空间被标记为“优雅删除”（通过设置删除周期），则将该 namespace 的状态设置为“terminating”并保存

https://www.veryopen.org

	到 etcd 中，同时删除该名字空间下的 serviceAccount、RC、pod、secret、PV、listRange、resourceQuota 和 event 等资源对象 。
node-ipam-controller	
node-lifecycle-controller	
node-route-controller	
persistentvolume-attach-detach-controller	
persistentvolume-binder-controller	
persistentvolume-expander-controller	
persistentvolume-protection-controller	
persistentvolumeclaim-protection-controller	
pod-garbage-collector-controller	回收节点上不再运行的 Pod，从而释放资源。
replicaset-controller	保证每一个应用都有特定数量的 Pod 副本在运行，如果数量太多，会删除几个，如果数量太少，将创建几个。
replicationcontroller-controller	
resourcequota-controller	资源配额管理器确保资源配额得到有效执行。
root-ca-certificate-publisher-controller	
service-lb-controller	
serviceaccount-controller	管理名字空间内的 ServiceAccount，并确保每个活跃的名字空间中都存在名为 default 的 ServiceAccount。
serviceaccount-token-controller	每当创建服务帐户（Service Account）时，令牌控制器就创建 API 令牌并添加到 Secret 资源对象中，此外，令牌控制器还监视 Secret 资源对象，当向服务帐户添加或删除 Secret 资源对象时添加或删除令牌。
statefulset-controller	管理一组 Pod 集合的部署和扩/缩容，并为这些 Pod 提供持久化存储和持久化标识符。
token-cleaner-controller	自动删除过期的令牌。本控制器默认不启用。
ttl-after-finished-controller	
ttl-controller	限制资源对象执行完后存活的时间，目前只处理 Job 资源对象。

Kube-controller-manager 服务程序通过命令参数“--controllers <控制器列表>”来指定需要启动的控制器，如“--controllers *”表示全部默认启动的控制器；“--controllers *,bootstrap-signer-controller”表示所有默认启动的控制器和 bootstrap-signer-controller；“--controllers *,-ttl-controller”表示除了 ttl-controller 控制器外全部默认启动的控制器，控制器加前缀-表示不要启动。

https://www.veryyoga.org

§ 13.2.3 Scheduler

对应的服务进程为 kube-scheduler，完成 pod 调度任务。调度决策考虑的因素包括：单个 Pod 及 Pod 集合的资源需求、软硬件及策略约束、亲和性及反亲和性规范、数据位置、工作负载间的干扰及最后时限。

§ 13.2.4 Kubelet

本组件运行在各个节点上，向 API Server 注册节点的状态信息——节点健康状况、节点名称、节点地址信息、操作

系统版本、Docker 版本、kubelet 版本等。根据 pod 的配置文件管理本节点上的 Pod，确保这些容器处于正常的运行状态，保证资源配额不被突破，同时收集日志和监控数据。kubelet 不会管理不是由 Kubernetes 创建的容器，比如用户手工启动的 docker 容器会被 kubelet 忽略。

§ 13.2.5 Kube Proxy

对应服务进程为 Kube-proxy。一个简单的网络代理、服务发现和负载均衡器，主要实现 service 功能，即实现集群内的客户端 pod 访问 service，或者集群外的主机通过 NodePort 等方式访问 Service。在 k8s 中，提供相同服务的一组 pod 可以抽象成一个 service，并分配一个 Cluster IP，通过访问这个 cluster ip 来间接访问 pod。Service 相当于一个负载均衡器（通过 iptables 实现负载均衡任务）。

§ 13.2.6 Cloud Controller Manager

对应服务进程 cloud-controller-manager，允许你将你的集群连接到云提供商的 API 之上， 并将与该云平台交互的组件与你的集群交互的组件分离开来。如果不与第三方云平台连接，本组件可以不安装。目前主流公共云提供商如亚马逊、阿里云、谷歌云等都推出了 k8s 服务，我们可以把线下的 k8s 集群与部分租用的 K8s 服务整合在一起。

https://www.veryopen.org

第14章 安装基础服务

---=== 全部虚拟机恢复快照 basic ===---

§14.1 设置防火墙规则

K8s 采用 iptables，不适用 firewalld，所以为了避免冲突，建议关闭 firewalld 防火墙：

```
[root@allnodes ~]# systemctl disable --now firewalld
```

如果开启了 firewalld，那么 Ping 不通其他节点上的 pod 的 ip 地址，但是 k8s 官方又说解决了防火墙冲突。

1. master1 上放行这些端口：2379-2380/tcp（etcd 服务）、6443/tcp(kube-apiserver 服务)、8472/udp（flannel 网络插件）、10250/tcp（kubelet API）、10257/tcp（kube-controller-manager）、10259/tcp（kube-scheduler）：

```
[root@master1 ~]# firewall-cmd --add-port={2379,2380,6443,10250,10256,10259}/tcp --add-port=8472/udp
[root@master1 ~]# firewall-cmd --runtime-to-permanent
[root@master1 ~]# firewall-cmd --list-all
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: ens160 ens192
  sources:
  services: cockpit dhcpv6-client ntp ssh
  ports: 8472/udp 2379/tcp 2380/tcp 6443/tcp 10250/tcp 10257/tcp 10259/tcp
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

2. 所有工作节点上允许这些端口进入：8472/udp（flannel 网络插件 vxlan 隧道）、10250/tcp（kubelet API）、10256/tcp（kube-proxy API）、30000-32767/tcp（对外暴露的应用端口）：

```
[root@worker? ~]# firewall-cmd --add-port=8472/udp --add-port=10250/tcp --add-port=10256/tcp \
--add-port=30000-32767/tcp --permanent
[root@worker? ~]# firewall-cmd --add-masquerade --permanent                                    #
开启混杂模式
[root@worker? ~]# firewall-cmd --reload
```

之所以要开启混杂模式，那是因为 coredns-xxxxxx pod 需要使用。

```
[root@worker? ~]# firewall-cmd --get-active-zones                                              #查看现
在使用哪个区
public
  interfaces: ens160
```

表明正在使用的区是 public，且网卡 ens160 绑定到此区。

```
[root@worker? ~]# firewall-cmd --info-zone=public
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: ens160 ens192
```

https://www.yuque.com/docker-yopen

```
  sources:
  services: cockpit dhcpv6-client ssh
  ports: 8472/udp 10250/tcp 10256/tcp 30000-32767/tcp
  protocols:
  forward: yes
  masquerade: yes
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

当然也可以线尝试放开全部端口，就像下面的命令那样：

```
firewall-cmd --add-port=1-65535/tcp --add-port=1-65535/udp --permanent
firewall-cmd --add-masquerade --permanent
firewall-cmd --reload
```

§14.2 时间同步服务

master1 一方面与互联网上的时间服务器对时，另一方面又为其他节点提供对时服务，其他节点只与 master1 对时，参考图 14-1。Kubernetes 集群要求所有节点的时间必须一致才行。

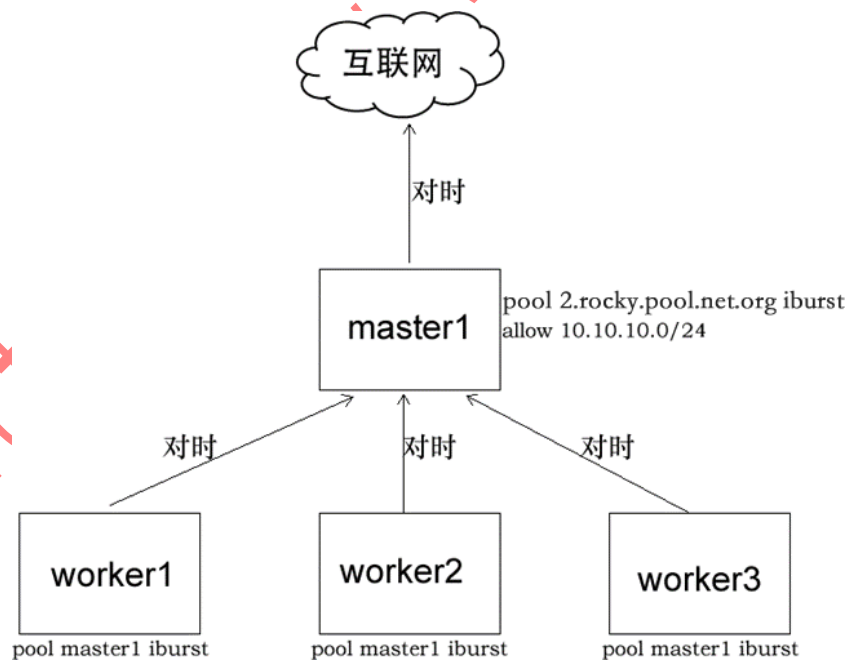


图 14-1 对时示意图

1. 安装和配置

```
[root@master1 ~]# vim /etc/chrony.conf
```

```
……
pool 2.Rocky.pool.ntp.org iburst                                                    #与网上时钟服务器对时
……
allow 10.10.10.0/24                                                                 #允许局域网内的电脑与本机对时
……

[root@master1 ~]# systemctl restart chronyd                                         #重启 chronyd 服务
[root@master1 ~]# systemctl enable chronyd
[root@master1 ~]# systemctl status chronyd
● chronyd.service - NTP client/server
   Loaded: loaded (/usr/lib/systemd/system/chronyd.service; enabled; vendor preset: enabled)
   Active: active (running) since Fri 2022-04-08 03:10:53 EDT; 6min ago
     Docs: man:chronyd(8)
……
```

上面倒数第二条命令是激活时钟服务，只有被激活的服务下次开机时才会自动启动。

查看 123 端口是否在监听（123 是 udp 端口，所以 ss 命令需要带-u 参数）：

```
[root@master1 ~]# ss -unlp
State  Recv-Q  Send-Q   Local Address:Port   Peer Address:Port Process
UNCONN 0       0            0.0.0.0:123         0.0.0.0:*     users:(("chronyd",pid=1393,fd=7))
UNCONN 0       0          127.0.0.1:323         0.0.0.0:*     users:(("chronyd",pid=1393,fd=5))
UNCONN 0       0              [::1]:323            [::]:*     users:(("chronyd",pid=1393,fd=6))
```

如果没有“0.0.0.0:123”，请修改/etc/chrony.conf 文件，把“#allow 10.10.10.0/24”中“#”字符删除，然后再重启 chronyd 服务。

在工作节点上执行下面的命令：

```
[root@worker? ~]# vim /etc/chrony.conf
……
pool master1 iburst                                                                 #与 master1 对时
……

[root@worker? ~]# systemctl restart chronyd
[root@worker? ~]# systemctl enable chronyd
[root@worker? ~]# chronyc sources
MS Name/IP address         Stratum Poll Reach LastRx Last sample
===============================================================================
^? master1                       0   6     0     -     +0ns[   +0ns] +/-    0ns
```

~~2. 设置防火墙规则，允许其他节点同步时间，即允许 ntp 服务能进入控制节点：~~

```
~~[root@master1 ~]# firewall-cmd --add-service=ntp --permanent~~
~~[root@master1 ~]# firewall-cmd --reload~~                                        ~~#规则立即生效~~
~~[root@master1 ~]# firewall-cmd --info-zone=public~~                              ~~#查看防火墙规则~~
~~public (active)~~
~~……~~
~~  interfaces: ens160 ens192~~
~~  sources:~~
~~  services: cockpit dhcpv6-client ntp ssh~~
~~  ports:~~
~~  protocols:~~
~~……~~
```

3. 全部节点立即对时：

```
[root@allnodes ~]# chronyc makestep
[root@allnodes ~]# timedatectl status                                               #检查系统时间同步状态
               Local time: Tue 2023-08-01 07:51:35 CST
           Universal time: Mon 2023-07-31 23:51:35 UTC
                 RTC time: Mon 2023-07-31 23:51:35
                Time zone: Asia/Shanghai (CST, +0800)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: no
[root@allnodes ~]# date                                                             #检查三个节点系统时间是否一样
```

过一会儿就会出现已经同步状态，如果总是出现“System clock synchronized: no”，干脆重启电脑试试。

§14.3 安装容器运行时

CRI 就是 Container Runtime Interface 的首字母缩写，即容器运行时接口，它是 Kubernetes 与容器运行时交互的接口。容器运行时类似虚拟机软件（如 VMWare Workstation），有了容器运行时才能跑容器，正如有了虚拟机软件，才能用虚拟机一样。k8s 从 1.24 版本开始不再支持 docker 容器运行时了，改用 containerd 或 CRI-O。我们这里采用 containerd。

Kubernetes 和 Containerd 之间的互动参考图 14-1 和 14-2。

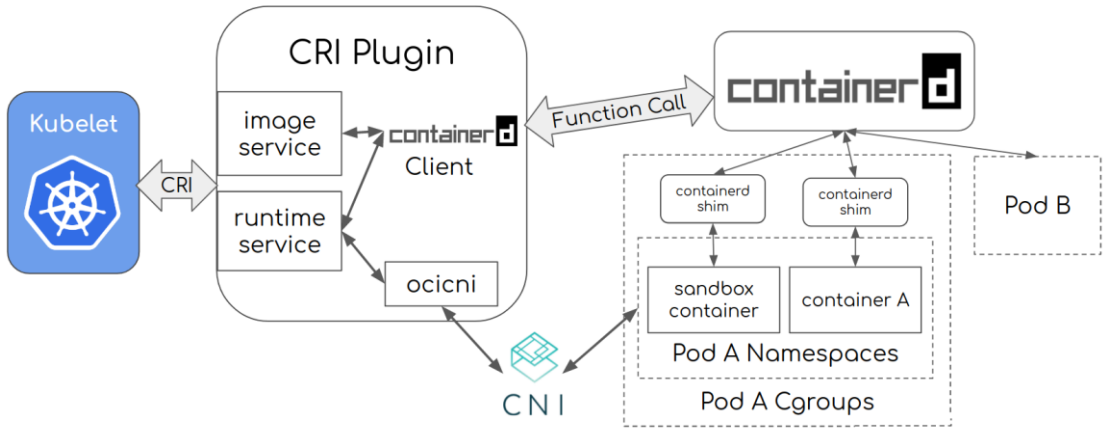


图 14-1 kubernetes 与 Containerd 的关系

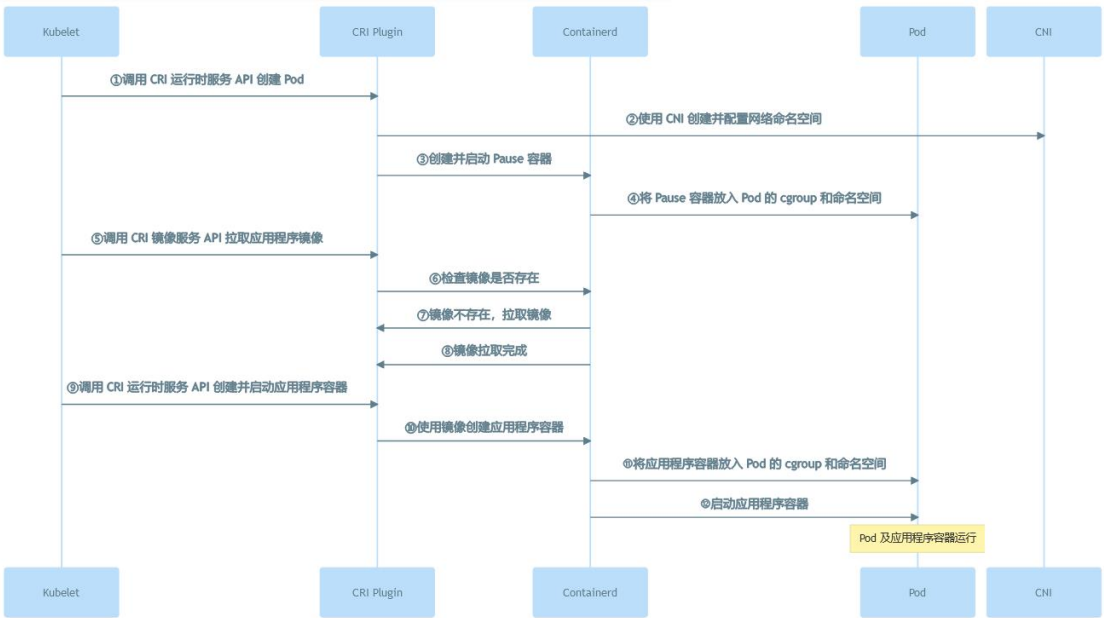


图 14-2 创建单容器 pod 的过程

在所有节点上操作。

1. containerd 不在 rocky 的标准安装源里面，需要先加入 docker 安装源：

```
[root@allnodes ~]# dnf config-manager --add-repo https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
[root@allnodes ~]# dnf makecache -y                                                    #更新本地缓存
[root@allnodes ~]# dnf upgrade -y                                                      #给操作系统打上最新的补丁
[root@allnodes ~]# shutdown -r now
```

~~上面第一条命令可能会失败，请多尝试几次，如果实在不行，就手工添加安装源：~~

```
~~[root@allnodes ~]# vim /etc/yum.repos.d/docker-ce.repo~~
~~[docker-ce-stable]~~
~~name=Docker CE Stable - $basearch~~
~~baseurl=https://download.docker.com/linux/rocky/$releasever/$basearch/stable~~
~~enabled=1~~
~~gpgcheck=1~~
~~gpgkey=https://download.docker.com/linux/rocky/gpg~~
```

2. 安装 containerd：

```
[root@allnodes ~]# dnf remove podman containers-common* docker-ce -y                   #先卸载这些软件
[root@allnodes ~]# dnf install containerd.io -y
```

3. 采用国内容器镜像加速

```
[root@allnodes ~]# cp /etc/containerd/config.toml /etc/containerd/config.toml.bak
[root@allnodes ~]# containerd config default > /etc/containerd/config.toml
[root@allnodes ~]# vim /etc/containerd/config.toml
......
[plugins."io.containerd.cri.v1.runtime".containerd.runtimes.runc.options]
  SystemdCgroup = true
......
    [plugins.'io.containerd.cri.v1.images'.registry]
      config_path = '/etc/containerd/certs.d'
......
   [plugins.'io.containerd.cri.v1.images'.pinned_images]
      sandbox = 'registry.aliyuncs.com/google_containers/pause:3.10.2'
......

[root@allnodes ~]# mkdir -p /etc/containerd/certs.d/{docker.io,registry.k8s.io,quay.io,ghcr.io}

[root@allnodes ~]# vim /etc/containerd/certs.d/docker.io/hosts.toml
server = "https://registry-1.docker.io"

[host."http://10.8.20.8:5000"]
  capabilities = ["pull", "resolve"]
  skip_verify = true

[root@allnodes ~]# vim /etc/containerd/certs.d/registry.k8s.io/hosts.toml
server = "https://registry.k8s.io"

[host."http://10.8.20.8:5001"]
  capabilities = ["pull", "resolve"]
  skip_verify = true

[root@allnodes ~]# vim /etc/containerd/certs.d/quay.io/hosts.toml
server = "https://quay.io"

[host."http://10.8.20.8:5002"]
  capabilities = ["pull", "resolve"]
  skip_verify = true

[root@allnodes ~]# vim /etc/containerd/certs.d/ghcr.io/hosts.toml
server = "https://ghcr.io"

[host."http://10.8.20.8:5003"]
  capabilities = ["pull", "resolve"]
  skip_verify = true

[root@allnodes ~]# systemctl restart containerd
```

配置号了之后，不管是手工拉取还是 yaml 文件中的镜像，都不需要特意加上 10.8.20.8:500?/前缀。

如果没法访问校内镜像服务器，那么：

```
[root@allnodes ~]# tee /etc/containerd/certs.d/docker.io/hosts.toml > /dev/null <<'EOF'
server = "https://registry-1.docker.io"

[host."https://docker.1ms.run"]
```

```
  capabilities = ["pull", "resolve"]

[host."https://docker.m.daocloud.io"]
  capabilities = ["pull", "resolve"]
EOF

[root@allnodes ~]# tee /etc/containerd/certs.d/registry.k8s.io/hosts.toml > /dev/null <<'EOF'
server = "https://registry.k8s.io"

[host."https://k8s.1ms.run"]
  capabilities = ["pull", "resolve"]

[host."https://registry.aliyuncs.com/google_containers"]
  capabilities = ["pull", "resolve"]
EOF

sudo tee /etc/containerd/certs.d/ghcr.io/hosts.toml > /dev/null <<'EOF'
server = "https://ghcr.io"

[host."https://ghcr.1ms.run"]
  capabilities = ["pull", "resolve"]
EOF

[root@allnodes ~]# tee /etc/containerd/certs.d/quay.io/hosts.toml > /dev/null <<'EOF'
server = "https://quay.io"

[host."https://quay.1ms.run"]
  capabilities = ["pull", "resolve"]
EOF
```

镜像加速存在三种方式：通过网络代理使用官方镜像库、采用国内镜像库、搭建私有镜像库。

对于 docker.io、gcr.io 和 quay.ip 等国外镜像库，我们替换为诸如阿里云的国内加速库。首先通过自己的手机号码登录 https://www.aliyun.com，然后点击右上角的“控制台”，在鼠标指到左上角的“三”主菜单，然后点击“容器”下面的“容器镜像服务 ACR”，再点击“镜像工具”下面的“镜像加速器”，最后复制图 14-3 中的红框内的网址，待用。

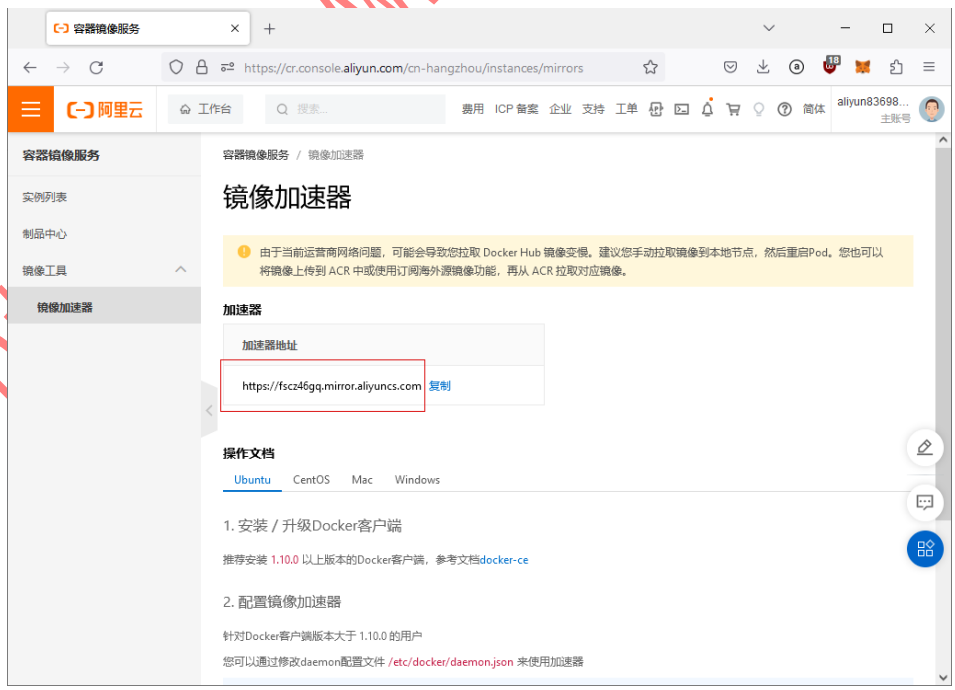


图 14-3 获取阿里云的镜像加速器

每个账号有限制流量，每 6 个小时只能下载 200 个，所以如果一个班的同学都在作实验，多注册几个账号。另一个账

号是 https://no3hfwj9.mirror.aliyuncs.com。

Containerd 的著配置文件 config.toml：

```
[root@allnodes ~]# mv /etc/containerd/config.toml{,.orig}                                    #备份原来的配置文件
[root@allnodes ~]# containerd config default > /etc/containerd/config.toml                   #导出默认的配置参数
[root@allnodes ~]# vim /etc/containerd/config.toml
......
    sandbox = "registry.aliyuncs.com/google_containers/pause:3.10.2"                         #指定国内镜像库
......
            SystemdCgroup = true                                                             #采用systemdcgroup控制组
......
    [plugins."io.containerd.grpc.v1.cri".registry]
      config_path = "/etc/containerd/certs.d"                                                #国内加速镜像库配置目录
.....
```

在 vim 中搜索关键字 sandbox、SystemdCgroup、config_path，定位到目标行后修改参数的值，不过强烈建议保留原先的行（在行首增加“#”注释掉），额外增加一行，比如针对 sandbox 这个参数，改为：

```
#    sandbox = "registry.k8s.io/pause:3.10.2"
    sandbox = "registry.aliyuncs.com/google_containers/pause:3.10.2"
```

配置国内加速镜像库：

```
[root@allnodes ~]# mkdir -p \
/etc/containerd/certs.d/{docker.io,gcr.io,quay.io,registry.k8s.io,_default,10.8.20.8:5000}
[root@allnodes ~]# vim /etc/containerd/certs.d/docker.io/hosts.toml
server = "https://docker.io"

[host."https://docker.m.daocloud.io"]
  capabilities = ["pull", "resolve"]

[host."https://dockerproxy.com/"]
  capabilities = ["pull", "resolve"]

[host."https://docker.rainbond.cc/"]
  capabilities = ["pull", "resolve"]

[host."https://fscz46gq.mirror.aliyuncs.com"]
  capabilities = ["pull", "resolve"]


[root@allnodes ~]# vim /etc/containerd/certs.d/gcr.io/hosts.toml
server = "https://gcr.io"

[host."https://gcr.nju.edu.cn"]
  capabilities = ["pull", "resolve"]

[root@allnodes ~]# vim /etc/containerd/certs.d/quay.io/hosts.toml
server = "https://quay.io"

[host."https://quay.mirrors.ustc.edu.cn"]
  capabilities = ["pull", "resolve"]

[host."https://quay.nju.edu.cn"]
  capabilities = ["pull", "resolve"]

[root@allnodes ~]# vim /etc/containerd/certs.d/registry.k8s.io/hosts.toml
server = "https://registry.k8s.io"

[host."https://k8s.m.daocloud.io"]
  capabilities = ["pull", "resolve"]

[host."https://registry.aliyuncs.com"]
  capabilities = ["pull", "resolve"]

[root@allnodes ~]# vim /etc/containerd/certs.d/10.8.20.8:5000/hosts.toml
server = "http://10.8.20.8:5000"

[host."http://10.8.20.8:5000"]
  capabilities = ["pull", "resolve"]

[root@master1 ~]# vim /etc/containerd/certs.d/_default/hosts.toml
[host."https://docker.rainbond.cc"]
  capabilities = ["pull","resolve"]
```

要注意缩进格式。server 定义名字空间的默认 URL，host 定义端点，如果没有 host，就访问默认 URL，如果有多个 Host，则按从上至下的顺序访问这些镜像库，直到镜下拉成功为止。capabilities 参数定义允许的操作："pull"-拉取镜像、"push"-推送镜像、"resolve"-产生镜像的唯一哈希值，除了私有镜像库，一般不允许推送镜像。

比如下拉 docker.io/library/busybox:latest 镜像时，就会采用

/etc/containerd/certs.d/docker.io/hosts.toml 配置文件，转而去下拉 docker.m.daocloud.i/library/busybox:latest，如果下拉失败，继续尝试下拉 dockerproxy.com/library/busybox:latest。下拉的镜像的 tag 依然是 docker.io/library/busybox:latest。国内的镜像库不稳定，如果以后遇到下来镜像失败的情况，请更新上面的配置文件。

重启并激活 containerd 服务：

```
[root@allnodes ~]# systemctl restart containerd
[root@allnodes ~]# systemctl enable containerd

[root@allnodes ~]# systemctl status containerd
● containerd.service - containerd container runtime
   Loaded: loaded (/usr/lib/systemd/system/containerd.service; enabled; vendor preset: disabled)
   Active: active (running) since Fri 2022-11-18 16:29:45 CST; 9s ago
     Docs: https://containerd.io
  Process: 3753 ExecStartPre=/sbin/modprobe overlay (code=exited, status=0/SUCCESS)
 Main PID: 3756 (containerd)
    Tasks: 10
   Memory: 17.1M
   CGroup: /system.slice/containerd.service
           └─3756 /usr/bin/containerd

Nov 18 16:29:45 worker1 containerd[3756]: time="2022-11-18T16:29:45.143966868+08:00" level=info msg="Start subscribing containerd event"
Nov 18 16:29:45 worker1 containerd[3756]: time="2022-11-18T16:29:45.144160648+08:00" level=info msg=serving... address=/run/containerd/containerd.sock.ttrpc
Nov 18 16:29:45 worker1 containerd[3756]: time="2022-11-18T16:29:45.144213935+08:00" level=info msg=serving... address=/run/containerd/containerd.sock
Nov 18 16:29:45 worker1 containerd[3756]: time="2022-11-18T16:29:45.144276000+08:00" level=info msg="containerd successfully booted in 0.015744s"
……
```

测试：

```
[root@allnodes ~]# ctr -n k8s.io images pull --hosts-dir /etc/containerd/certs.d --plain-http \
 docker.io/library/busybox:latest
docker.io/library/busybox:latest                  saved
└──index (c5e894c6b246)                             complete       |++++++++++++++++++++++++++++++++++++++|
   └──manifest (1cfa4e2b09e1)                       complete       |++++++++++++++++++++++++++++++++++++++|
      ├──config (c6348fa86ba0)                      complete       |++++++++++++++++++++++++++++++++++++++|
      └──layer (b05093807bb0)                       extracted      |++++++++++++++++++++++++++++++++++++++|
application/vnd.oci.image.index.v1+json sha256:c5e894c6b24654792f64f609b27ba2fe8058036731bdf183a74f8a9a0ff9d3ef
Completed pull from OCI Registry (docker.io/library/busybox:latest)     elapsed: 0.3 s  total:  2.1 Mi  (7.0 MiB/s)
```

如果依次拉取失败，请尝试多次拉取。

§14.4 总结

1. 检查交换区是否被禁用：

```
[root@allnodes ~]# free
               total        used        free      shared  buff/cache   available
Mem:         3980604      441084     3604712        9192      118824     3539520
Swap:              0           0           0
```

如果没禁用，请参考“§2.3 基本配置”去禁用交换区。

2. 查看 SELinux 是否被禁用：

```
[root@allnodes ~]# sestatus
SELinux status:                 disabled
```

如果没禁用，请参考“§2.3 基本配置”去禁用 SELinux。

3. 有没有额外加载两个内核模块：

```
[root@allnodes ~]# lsmod|grep -e overlay -e br_netfilter
br_netfilter           36864  0
bridge                405504  1 br_netfilter
overlay               200704  0
```

https://www.ver

如果没加载，请参考“§2.3 基本配置”去加载这两个模块。

4．检查三个内核参数是否开启：

```
[root@allnodes ~]# sysctl -a | grep -e bridge-nf-call-ip -e 'ip_forward ='
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
```

如果没有开启，请参考“§2.3 基本配置”去开启。

5．检查时区是否是中国上海：

```
[root@allnodes ~]# timedatectl
               Local time: Fri 2023-08-18 08:53:13 CST
           Universal time: Fri 2023-08-18 00:53:13 UTC
                 RTC time: Fri 2023-08-18 00:53:13
                Time zone: Asia/Shanghai (CST, +0800)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: no
```

如果不是，请参考“§2.3 基本配置”去修改时区。

6．检查主机名有没有解析：

```
[root@allnodes ~]# more /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
10.10.10.11     master1
10.10.10.31     worker1
10.10.10.32     worker2
10.10.10.33     worker3
```

7．检查 firewalld 防火墙是否被禁用：

```
[root@allnodes ~]# systemctl status firewalld
○ firewalld.service - firewalld - dynamic firewall daemon
     Loaded: loaded (/usr/lib/systemd/system/firewalld.service; disabled; preset: enabled)
     Active: inactive (dead)
       Docs: man:firewalld(1)
……
```

如果没禁用，请运行命令“systemctl disable --now firewalld”禁用它。

关闭全部虚拟机，创建快照“before kubemdm”，快照备注为“第 15 章前”。

第15章 部署 k8s 集群

图 15-1 是我们最终要实现的 k8s 集群。

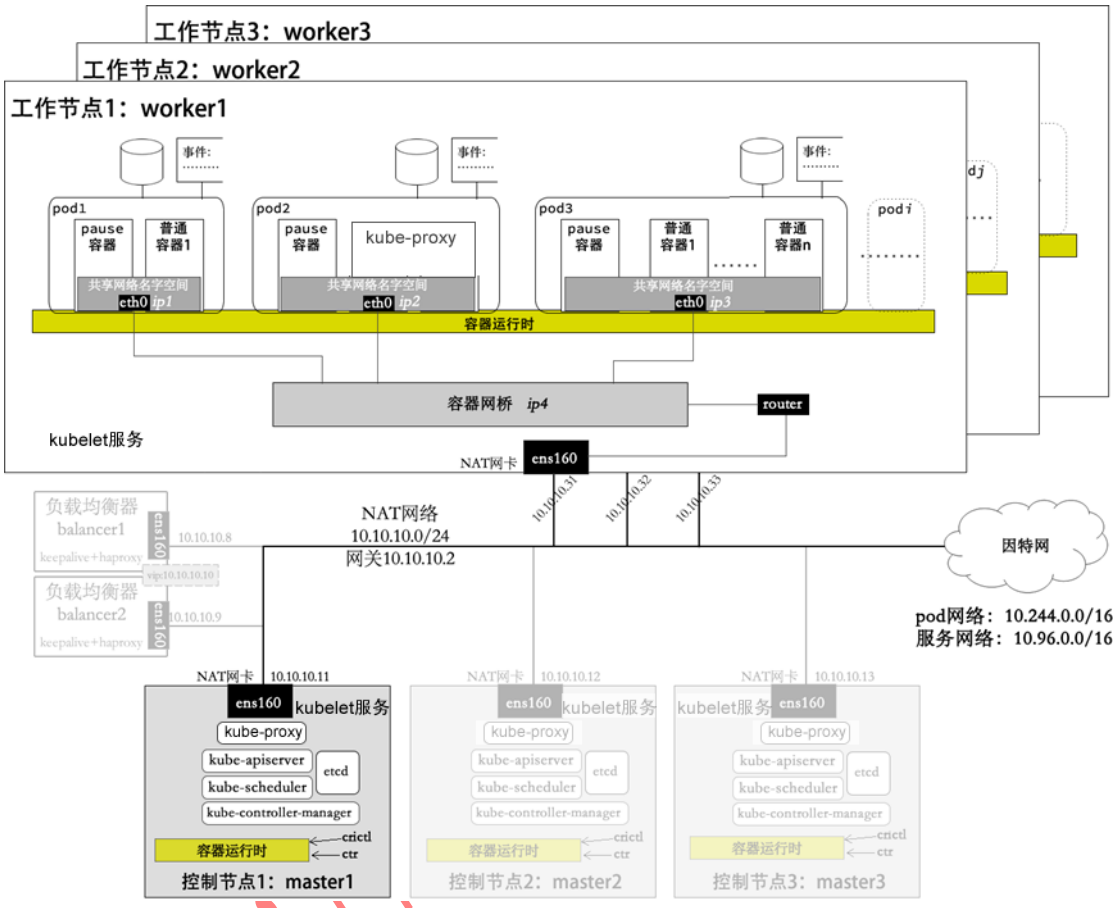


图 15-1 k8s 集群

每台虚拟机都有一块 NAT 模式的网卡。在生产环境中，一般采用多块网卡来分流不通类型的网络流量。

我们分阶段实战，第一阶段：采用一台控制节点+三台工作节点；第二阶段：增加两台控制节点+一台负载均衡器；第三阶段：再增加一个工作节点+部署若干应用，然后人为造成某些节点故障，监测应用的可用性。

可以用下面三种方式中的任意一种来部署 k8s 集群：

1. 采用 Kubeadm 或 minikube 工具部署。部署快且简单，在开发环境中经常采用，我们下面先采用这个方法。

2. 二进制文件部署。部署过程比较复杂，涉及自签证书、配置文档和大量的软件安装。学习环境中经常采用，让学员掌握部署的详细过程，了解各种细节，这样运维时就能快速排错。中大型企业的生产环境也经常采用本部署方法，是南北各路 kubernetes 高手的最爱，他们偏执地总想掌控一切，当然运维过程中，解决问题也是快、恨、准。我们在授课过程中，如果时间充裕，不妨也来尝试本部署方法，能否成功，就看各位造化了（具体参考 21 章）。

3. 直接使用第三方提供的 k8s 服务（如亚马逊的 EKS、谷歌的 GKS 等）。在中小企的生产环境中普遍采用这个方法，当然大型企业也可以采用，不过大型企业往往都有自己的数据中心，他们更愿意把 k8s 部署在自己的机房。

大致的步骤就是：

准备环境→安装 containerd、kubectl、kubeadm 和 kubelet 软件包→创建 k8s 集群→把工作节点加入集群→验证。

§15.1 安装 k8s 程序

所有节点上操作。

1. 增加新版 Kubernetes 官方安装源：

```
[root@allnodes ~]# vim /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes-new/core/stable/v1.36/rpm/
enabled=1
gpgcheck=1
gpgkey=https://mirrors.aliyun.com/kubernetes-new/core/stable/v1.36/rpm/repodata/repomd.xml.key
exclude=kubeadm kubelet kubectl cri-tools kubernetes-cni

[root@allnodes ~]# dnf makecache -y
```

“exclude=kubeadm kubelet kubectl ……”表示升级系统时不升级 kubeadm kubelet kubectl ……。如果要安装 1.29 版本之前的 kubernetes，安装源配置请参考 https://developer.aliyun.com/mirror/kubernetes/?spm=a2c6h.25603864.0.0.5124368fWxxCWH 。

2. 查看 kubeadm 的版本：

```
[root@master1 ~]# dnf list  kubeadm.x86_64 --showduplicates --disableexcludes=kubernetes
上次元数据过期检查：0:01:03 前，执行于 2026 年 08 月 22 日 星期六 18 时 28 分 23 秒。
可安装的软件包
kubeadm.x86_64                          1.36.0-150500.1.1                       kubernetes
kubeadm.x86_64                          1.36.1-150500.1.1                       kubernetes
kubeadm.x86_64                          1.36.2-150500.2.1                       kubernetes
kubeadm.x86_64                          1.36.3-150500.1.1                       kubernetes
kubeadm.x86_64                          1.36.4-150500.1.1                       kubernetes
kubeadm.x86_64                          1.36.5-150500.1.1                       kubernetes
```

可见最新版本为 1.36.5，也可以浏览文件 https://kubernetes.io/releases/查看最新版本号。生产换种中一般不主张采用最新版，采用次新版本比较稳妥，因为最新版本还不够稳定。kubernetes 的版本格式 x.y.z，其中 x 为主版本号、y 为次版本号、z 为补丁次数。

3. 安装（工作节点上可以不装 kubectl）：

```
[root@allnodes ~]# dnf install kubeadm-1.36.4 kubelet-1.36.4 kubectl-1.36.4 --disableexcludes=kubernetes -y
```

也可以安装最新的版本，即执行命令 dnf install kubeadm kubectl kubelet -y 即可。

4. 配置 kubelet 使用 containerd 容器运行时：

```
[root@allnodes ~]# vim /etc/sysconfig/kubelet                                    #确保存在下面一行
KUBELET_EXTRA_ARGS="--cgroup-driver=systemd --container-runtime-endpoint=unix:///run/containerd/containerd.sock"
```

https://www.ve...nen.org

5. 命令 crictl 的配置文件（如何与 containerd 交互）：

```
# 下载 crictl v1.36.0
[root@allnodes ~]# wget https://github.com/kubernetes-sigs/cri-tools/releases/download/v1.36.0/crictl-v1.36.0-linux-amd64.tar.gz

# 解压并将二进制文件移动到系统路径
[root@allnodes ~]# tar -xvf crictl-v1.36.0-linux-amd64.tar.gz
[root@allnodes ~]# mv crictl /usr/local/bin/

# 配置：
[root@allnodes ~]# vim /etc/crictl.yaml
runtime-endpoint: unix:///run/containerd/containerd.sock
image-endpoint: unix:///run/containerd/containerd.sock
timeout: 120
debug: false
```

“timeout: 120”表示在 120 秒内不能下载镜像就报错，“/run/containerd/containerd.sock”为 containerd 的网络套接字。

测试：

```
[root@allnodes ~]# crictl pull docker.io/library/busybox:latest
```

如果是被，请多运行几次。

6. kubectl 命令自动补全。在使用 kubectl 命令的时候，经常需要输入各种参数，如果有自动补全，可以节省很多时间，让工作更有效率！

```
[root@master1 ~]# dnf install bash-completion -y
[root@master1 ~]# kubectl completion bash > /etc/bash_completion.d/kubectl
[root@master1 ~]# source /usr/share/bash-completion/bash_completion
[root@master1 ~]# source <(kubectl completion bash)
```

上面第二条命令在/etc/bash_completion 目录下创建文件 kubectl，这样以后任何用户登录进来都可以使用 kubectl 命令自动补全参数。上面最后一条命令表示现在就可以使用此功能。好了，你现在输入 kubectl cl 在按 Tab 键试试会不会补全，或者输入 kubectl 然后连续按 Tab 键看看会发生什么情况。

https://www.verydoc.org

§15.2 初始化控制面

只在 master1 节点上操作。

1. 列出初始化时需要的容器镜像：

```
[root@master1 ~]# kubeadm config images list --image-repository=registry.aliyuncs.com/google_containers \
  --kubernetes-version=v1.36.4
registry.aliyuncs.com/google_containers/kube-apiserver:v1.36.4
registry.aliyuncs.com/google_containers/kube-controller-manager:v1.36.4
registry.aliyuncs.com/google_containers/kube-scheduler:v1.36.4
registry.aliyuncs.com/google_containers/kube-proxy:v1.36.4
registry.aliyuncs.com/google_containers/coredns:v1.14.2
registry.aliyuncs.com/google_containers/pause:3.10.2
registry.aliyuncs.com/google_containers/etcd:3.6.8-0
```

如果省略“--kubernetes-version=v1.36.4”，就表示使用各个容器镜像的最新版本，不过在国内，下载版本号

文件 https://cdn.dl.k8s.io/release/stable-1.txt 一般会超时而失败，所以你也可以事先在浏览器中浏览这个文件或者执行“curl https://cdn.dl.k8s.io/release/stable-1.txt”命令获取最新版本号，然后增加命令行参数--kubernetes-version=<最新版本号>。

2．首先下载初始化时需要的容器镜像：

```
[root@master1 ~]# kubeadm config images pull --kubernetes-version=v1.36.4
[config/images] Pulled registry.k8s.io/kube-apiserver:v1.36.4
[config/images] Pulled registry.k8s.io/kube-controller-manager:v1.36.4
[config/images] Pulled registry.k8s.io/kube-scheduler:v1.36.4
[config/images] Pulled registry.k8s.io/kube-proxy:v1.36.4
[config/images] Pulled registry.k8s.io/coredns/coredns:v1.14.2
[config/images] Pulled registry.k8s.io/pause:3.10.2
[config/images] Pulled registry.k8s.io/etcd:3.6.8-0
```

如果不单独下载这些镜像，在初始化集群时也会自动下载，但是由于国内的网络环境，下载镜像很可能会因超时而导致初始化集群失败，因此建议大家提前下载好。如果一次下载失败，可以尝试多次下载，直到需要的容器镜像全部下载成功为止。如果下拉最新版，那么请省略--kubernetes-version=v1.36.4 参数。如果不用本地库，就加上--image-repository=registry.aliyuncs.com/google_containers 参数。

如果存在某个镜像总是下拉失败，可以使用命令单独下拉它，比如单独下拉 registry.aliyuncs.com/google_containers/etcd:3.6.8-0 镜像的命令：ctr -n k8s.io images pull registry.aliyuncs.com/google_containers/etcd:3.6.8-0。

如果一直下拉失败，请参考“§23.3 初始化时下拉镜像失败”处理。

3．显示已经下拉的容器镜像：

```
[root@master1 ~]# crictl images ls
IMAGE                                TAG                 IMAGE ID            SIZE
docker.io/library/busybox            latest              c6348fa86ba0f       2.23MB
registry.k8s.io/coredns/coredns      v1.14.2             38667dd9be96c       23.3MB
registry.k8s.io/etcd                 3.6.8-0             ee85eb1f0edd2       22.9MB
registry.k8s.io/kube-apiserver       v1.36.4             b0f70fa6ec47e       29.8MB
registry.k8s.io/kube-controller-manager   v1.36.4        c27613cea70f9       26.5MB
registry.k8s.io/kube-proxy           v1.36.4             a5280d36e1aa3       30.1MB
registry.k8s.io/kube-scheduler       v1.36.4             11ce0d8bbc73b       20.5MB
registry.k8s.io/pause                3.10.2              4a83b15d3ecfe       318kB
```

或者采用下面的命令：

```
[root@master1 ~]# ctr -n k8s.io image ls -q | grep -v sha256
registry.k8s.io/coredns/coredns:v1.14.2
registry.k8s.io/etcd:3.6.8-0
registry.k8s.io/kube-apiserver:v1.36.4
registry.k8s.io/kube-controller-manager:v1.36.4
registry.k8s.io/kube-proxy:v1.36.4
registry.k8s.io/kube-scheduler:v1.36.4
registry.k8s.io/pause:3.10.2
```

ctr 命令需要用参数-n k8s.io 指明名字空间。

4．查看容器镜像中的根文件系统（可以不查看）：

```
[root@master1 ~]# mkdir ~/pod                                                    #创建目录

把容器镜像中的根文件系统挂载到~/pod 目录：
[root@master1 ~]# ctr -n k8s.io image mount registry.k8s.io/etcd:3.6.8-0 ~/pod/
sha256:b6844fe8237e06364826ecf1934f1b577d8f0955e4acd87f1a37f6099614f2cb
./pod/
[root@master1 ~]# cd ~/pod
[root@master1 ~]# ll
total 0
drwxr-xr-x 1 root  root  16 May 16 22:13 bin
drwxr-xr-x 2 root  root   6 Apr  2 19:55 boot
drwxr-xr-x 2 root  root   6 Apr  2 19:55 dev
drwxr-xr-x 1 root  root  17 Jan  1  2000 etc
```

https://www.veryopen.org

```
drwxr-xr-x 1 65532 65532 21 Jan  1  1970 home
drwxr-xr-x 2 root  root   6 Apr  2 19:55 lib
drwxr-xr-x 2 root  root   6 Apr  2 19:55 proc
drwx------ 1 root  root   6 Jan  1  1970 root
drwxr-xr-x 2 root  root   6 Apr  2 19:55 run
drwxr-xr-x 2 root  root   6 Apr  2 19:55 sbin
drwxr-xr-x 2 root  root   6 Apr  2 19:55 sys
drwxrwxrwt 1 root  root   6 Aug  4  2017 tmp
drwxr-xr-x 1 root  root  19 May 16 22:13 usr
drwxr-xr-x 1 root  root  17 Jan  1  1970 var

[root@master1 pod]# more etc/os-release                                    #查看发行版本
PRETTY_NAME="Distroless"
NAME="Debian GNU/Linux"
ID="debian"
VERSION_ID="12"
VERSION="Debian GNU/Linux 12 (bookworm)"                                   #通过定制 debian 12 而来的
……

显示全部的可执行命令：
[root@master1 pod]# find bin sbin usr/{bin,sbin,local/bin}
bin
sbin
usr/bin
usr/sbin
usr/local/bin
usr/local/bin/etcd
usr/local/bin/etcdutl
usr/local/bin/etcdctl

[root@master1 pod]# ldd usr/local/bin/etcd                                 #查看 etcd 程序的编译方式
        not a dynamic executable                                           #静态编译
```

下面卸载容器的根文件系统：

```
[root@master1 pod]# cd
[root@master1 ~]# ctr -n k8s.io image unmount ~/pod
/root/pod
```

可见容器中的 linux 命令是很少的，另外容器中的程序一般都是静态编译，所以运行时不需要额外的运行库。如果你想往容器中增加程序，挂载时需要带参数--rw，然后就可往挂载的目录中拷贝程序。比如挂载的命令如下：

```
ctr -n k8s.io image mount --rw registry.k8s.io/etcd:3.6.8-0 ~/pod
```

5. 修改初始化 k8s 集群的默认配置参数：

导出默认的初始化参数到文件 kubeadm-config.yaml 中：

```
[root@master1 ~]# kubeadm config print init-defaults --kubeconfig ClusterConfiguration > ~/kubeadm-config.yaml
```

修改初始化参数（只需要改动红色字体部分）：

```
[root@master1 ~]# vim ~/kubeadm-config.yaml
apiVersion: kubeadm.k8s.io/v1beta4
bootstrapTokens:
- groups:
  - system:bootstrappers:kubeadm:default-node-token
  token: abcdef.0123456789abcdef
  ttl: 24h0m0s
  usages:
  - signing
  - authentication
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: 10.10.10.11  ----------#监听本机第一块网卡的 ip 地址
  bindPort: 6443
nodeRegistration:
  criSocket: unix:///var/run/containerd/containerd.sock
  imagePullPolicy: IfNotPresent
  name: master1  ----------#本机的机器名字
  taints: null
---
apiServer:
  timeoutForControlPlane: 4m0s
apiVersion: kubeadm.k8s.io/v1beta4
certificatesDir: /etc/kubernetes/pki
clusterName: k8scluster  ----------#集群名字
controllerManager: {}
dns: {}
etcd:
```

```
  local:
    dataDir: /var/lib/etcd
imageRepository: registry.aliyuncs.com/google_containers                    #国内的镜像库
kind: ClusterConfiguration
kubernetesVersion: 1.36.4
networking:
  dnsDomain: cluster.local
  serviceSubnet: 10.96.0.0/16                                               #服务网
  podSubnet: 10.244.0.0/16                                                  #pod 网络
proxy: {}
```

pod 网 10.244.0.0/16 给所有的 pod 分配 ip 地址，首先划分成子网 10.244.*.0/24，每个节点使用一个子网，比如 master1 使用 10.244.0.0/24，worker1 使用 10.244.1.0/24，worker2 使用 10.244.2.0/24，……。各个节点上的 pod 的 Ip 就落在本节点子网内。所以 10.244.0.0/16 最多只能划分成 255 个子网，也就决定了 k8s 集群中的节点数不能超过 255 个，如果超过，请把 pod 网的网络部分改小，如改为 10.244.0.0/8。

6．搭建集群前的准备工作。

采用命令 kubeadm 初始化 k8s 集群，会执行很多“阶段”，一旦某个“阶段”出错，整个初始化过程就算失败，所以为了保证一次性通过，我们尽量做好前期的准备工作。

6.1）检查环境是否满足要求

```
[root@allnodes ~]# systemctl enable kubelet                                 #激活 kubelet 服务

[root@master1 ~]# kubeadm init phase preflight --config kubeadm-config.yaml
[preflight] Running pre-flight checks
        [WARNING SystemVerification]: kernel release 5.14.0-626.el9.x86_64 is unsupported. Supported LTS versions from the 5.x series are
5.4, 5.10 and 5.15. Any 6.x version is also supported. For cgroups v2 support, the recommended version is 5.10 or newer
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
```

警告 linux 操作系统的内核版本不支持，在实验环境下可以忽略它。如果报了错误，请排错。~~上面有一个警告说防火墙已经启动，请确保 6443/tcp 和 10250/tcp 端口允许通过防火墙。运行命令 firewall-cmd --list-all 查看防火墙规则中有没有放行这两个端口，如果没放行，请执行下面的命令放行：~~

```
[root@master1 ~]# firewall-cmd --add-port={6443,10250}/tcp --permanent
success
[root@master1 ~]# firewall-cmd --reload
```

~~不过建议关闭防火墙。~~

采用 kubeadm 初始 K8s 控制面会经历多个阶段，可用下面命令列出这些阶段：

```
[root@master1 ~]# kubeadm init phase -h
Use this command to invoke single phase of the init workflow

Usage:
  kubeadm init phase [command]

Available Commands:
  addon               阶段 12：安装一些额外的必要的插件（如 CoreDNS、kube-proxy）。
  bootstrap-token     阶段 10：生成用于将节点加入集群的 bootstrap 令牌。
  certs               阶段 2：生成各种证书和私钥。
  control-plane       阶段 5：产生控制面所需的全部静态 Pod 清单文件。控制面运行在 pod 中。
  etcd                阶段 4：为本地 etcd 生成静态 Pod 清单文件。etcd 运行在 pod 中。
  kubeconfig          阶段 3：产生各种配置文件，这些配置文件用于搭建控制面。
  kubelet-finalize    阶段 11：更新“/etc/kubernetes/kubelet.conf”以指向可轮换的 kubelet 客户端证书和密钥。
  kubelet-start       阶段 6：配置并启动 kubelet。
  mark-control-plane  阶段 9：通过添加标签将节点（比如 master1）标记为控制平面，添加的标签是[node-role.kubernetes.io/control-
plane node.kubernetes.io/exclude-from-external-load-balancers]。
  preflight           阶段 1：检查环境是否合适。
  show-join-command   阶段 13：打印工作节点加入集群的命令。本阶段不能单独执行。
  upload-certs        阶段 8：把证书保存到 kubeadm-certs。本步骤会跳过，请参考--upload-certs 参数。
  upload-config       阶段 7：将“kubeadm-config”这个 ConfigMap 中的配置存储在“kube-system”命名空间中。
  wait-control-plane  阶段 14：等待控制面板启动完成。
……
Use "kubeadm init phase [command] --help" for more information about a command.
```

大部分“阶段”允许单独执行，比如“kubeadm init phase preflight”执行完第一阶段就退出。也可以采用参数--skip-phases 跳过某些阶段，如“kubeadm init --skip-phases preflight,certs”跳过“阶段 1”和“阶段 2”。不过全部阶段必须都要被成功执行，比如首次初始化时执行到“阶段 5”时失败而退出，然后我们排错后

再次初始化时就不能跳过“阶段 5”，只能跳过“阶段 1”到“阶段 4”，但是有一些“阶段”不能重复执行。所以如果初始化是失败，建议还是执行命令“kubeadm reset -f”重置已经执行过的全部“阶段”，然后再来重新初始化。

6.2） 完整模拟一次初始化全过程：

```
[root@master1 ~]# kubeadm init --dry-run --config ~/kubeadm-config.yaml
…… 大量信息，从中可以了解初始化过程中的很多内部机密😉 ……
Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 10.10.10.11:6443 --token abcdef.0123456789abcdef \
        --discovery-token-ca-cert-hash sha256:0c1d2c4924bf6a4e1418ddaf1df0c3a761008535d2d11d7ed81da49246d6aca8
```

如果输出结果的最后是告诉你如何加入工作节点，那么恭喜你，初始化过程模拟成功，接下来你就可以放心大胆地正式开始初始化了。模拟产生的文件在/etc/kubernetes/tmp 目录下，模拟完了可以把此目录删除。

---=== 建议关闭虚拟机创建快照 before kubeadm init ===---

7. 初始化 k8s 控制面（正式初始 K8s 集群）：

```
[root@master1 ~]# kubeadm init --config ~/kubeadm-config.yaml
[init] Using Kubernetes version: v1.31.2
[preflight] Running pre-flight checks                                                                    #执行“阶段 1”
        [WARNING Firewalld]: firewalld is active, please ensure ports [6443 10250] are open or your cluster may not function correctly
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"                                                #执行“阶段 2”
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default kubernetes.default.svc
kubernetes.default.svc.cluster.local master1] and IPs [10.96.0.1 10.10.10.11]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [localhost master1] and IPs [10.10.10.11 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [localhost master1] and IPs [10.10.10.11 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"                                                   #执行“阶段 3”
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"                        #执行“阶段 4”
[control-plane] Using manifest folder "/etc/kubernetes/manifests"                                        #执行“阶段 5”
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"  #执行“阶段 6”
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Starting the kubelet
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from directory "/etc/kubernetes/manifests". This
can take up to 4m0s
[apiclient] All control plane components are healthy after 6.001915 seconds
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" Namespace  #执行“阶段 7”
[kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the configuration for the kubelets in the cluster
[upload-certs] Skipping phase. Please see --upload-certs                                                 #执行“阶段 8”
[mark-control-plane] Marking the node master1 as control-plane by adding the labels: [node-role.kubernetes.io/control-plane
node.kubernetes.io/exclude-from-external-load-balancers]                                                 #执行“阶段 9”
[mark-control-plane] Marking the node master1 as control-plane by adding the taints [node-role.kubernetes.io/control-plane:NoSchedule]
[bootstrap-token] Using token: abcdef.0123456789abcdef                                                   #执行“阶段 10”
[bootstrap-token] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for nodes to get long term certificate
credentials
[bootstrap-token] Configured RBAC rules to allow the csrapprover controller automatically approve CSRs from a Node Bootstrap Token
[bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client certificates in the cluster
[bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[kubelet-finalize] Updating "/etc/kubernetes/kubelet.conf" to point to a rotatable kubelet client certificate and key  #执行“阶段 11”
[addons] Applied essential addon: CoreDNS                                                                #执行“阶段 12”
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!                                              #恭喜，初始化成功！

To start using your cluster, you need to run the following as a regular user:                            #普通用户如何操纵 k8s 集群

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:                                                    #root 用户如何操纵 k8s 集群
```

```
  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.  ------------------------------------------------------------------#在集群中如何部署 pod 网络
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:                                  #执行“阶段 13”

kubeadm join 10.10.10.11:6443 --token abcdef.0123456789abcdef \
        --discovery-token-ca-cert-hash sha256:ad5a49bb1bb4f439e2716e164eca2859da36147548654779838 8c6a71694a442
```

从输出结果我们能了解到以下一些重要信息：

(1) 大部分行的行首[…]指示正在执行的“阶段”，如[kubeconfig]表示正在执行 kubeconfig 阶段，也就是“阶段 3”。如果初始化失败，我们就知道执行到哪个阶段失败了。

(2) 作为 Linux 操作系统的普通用户，接下来如何去操纵 k8s 集群？请参考下面的“7. 设置工作环境”。

(3) 作为 Linux 操作系统的超级用户 root，接下来如何去操纵 k8s 集群？只要定义一个环境变量 KUBECONFIG 就可以了，具体操作请参考下面的“7. 设置工作环境”。

(4) 如何在集群中部署 pod 网络？只要执行“kubectl apply -f <podnetwork>.yaml”命令即可，其中<podnetwork>需用具体的网络插件名替换。但是在国内很不幸，需要采用其他方法才行，具体请参考“§13.4 安装网络插件”小节。

(5) 告诉我们接下来如何把工作节点加入到 K8s 集群中。就是在工作节点上执行一个命令，即上面黄色字体部分的那个命令。建议把最后两行保存到文件中，以后往集群中增加工作节点时需要用到它，当然可以随时执行“kubeadm token create --print-join-command”查看加入节点的命令。

初始化集群也可以直接采用命令行参数，而不用配置文件，比如命令：kubeadm init --image-repository registry.aliyuncs.com/google_containers --kubernetes-version v1.36.4 --pod-network-cidr 10.244.0.0/16 --service-cidr 10.96.0.0/16 --apiserver-advertise-address 10.10.10.11 --node-name master1 --v 5，不过没法定义集群名称。

如果初始化失败，重点检查 containerd、kubelet 服务和它们的配置文件/etc/containerd/config.toml（参考 14.3 小节）、/etc/sysconfig/kubelet（参考 15.1 小节），如：systemctl status containerd、systemctl status kubelet。

如果需要再次初始化，请先执行命令 kubeadm reset -f 重置。

8. 设置工作环境：

如果是 Linux 的 root 用户，定义一个环境变量 KUBECONFIG 即可：

```
[root@master1 ~]# export KUBECONFIG=/etc/kubernetes/admin.conf
[root@master1 ~]# echo 'export KUBECONFIG=/etc/kubernetes/admin.conf' >> ~/.bashrc
```

上面第二条命令把定义环境变量的命令追加到文件~/.bashrc 文件的末尾，这样下次登录时就自动设置好了环境变量。我们以后都是以 root 用户来操纵 k8s 集群，所以下面的三条命令可以略过。

如果是 linux 的普通用户（比如 alice），则登录后执行下面三条命令即可：

```
[alice@master1 ~]$ mkdir -p $HOME/.kube
[alice@master1 ~]$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
[alice@master1 ~]$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

~~9. 设置防火墙规则：~~

```
[root@master1 ~]# firewall-cmd \
--add-service={etcd-server,etcd-client,kube-apiserver,kube-control-plane,kube-control-plane-secure,kubelet,\
kubelet-readonly,http,https}
[root@master1 ~]# firewall-cmd --runtime-to-permanent

[root@master1 ~]# firewall-cmd --list-all
public (active)
```

https://www.veryopen.org

```
  target: default
  icmp-block-inversion: no
  interfaces: ens160
  sources:
  services: cockpit dhcpv6-client etcd-client etcd-server http https kube-apiserver kube-control-plane kube-
control-plane-secure kubelet kubelet-readonly ntp ssh
  ports: 6443/tcp 10250/tcp
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

§15.3 了解集群

到此只有一个节点的集群初始化完了，接下来我们来看看这个集群“长”得咋样？

1．查看集群的状态：

```
[root@master1 ~]# kubectl cluster-info
Kubernetes control plane is running at https://10.10.10.11:6443
CoreDNS is running at https://10.10.10.11:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

如果要查看集群更详细的状态信息，请执行命令“kubectl cluster-info dump”。

2．查看集群中的节点：

```
[root@master1 ~]# kubectl get nodes
NAME      STATUS     ROLES           AGE   VERSION
master1   NotReady   control-plane   11m   v1.34.0
```

目前集群中只有一个节点 master1，节点角色 control-plane（控制面），节点的状态为 NotReady，因为还没有部署 pod 网络，等以后部署完网络，就变为 Ready 了。

3．查看已有的名字空间：

```
[root@master1 ~]# kubectl get namespaces
NAME              STATUS   AGE
default           Active   100m                                  #默认名字空间，操纵资源时如果没有指定名字空间，就使用 default
kube-node-lease   Active   100m        #用于与各个节点关联的租约对象。节点租约允许 kubelet 发送心跳，因此控制面能够检测到节点故障
kube-public       Active   100m                                  #公共名字空间，所有客户端均可见
kube-system       Active   100m                                  #kubernetes 系统创建的对象就在这里
```

初始命令空间存在四个。名字空间（Namespace） 提供一种机制，将同一集群中的资源划分为相互隔离的组，同一名字空间内的资源名称必须唯一，但跨名字空间时没有这个要求。执行命令“kubectl api-resources”可查看 kubernetes 支持的全部资源。

4．查看集群中的 pod：

```
[root@master1 ~]# kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE    IP             NODE      NOMINATED NODE   READINESS GATES
kube-system   coredns-66f779496c-lmwcz          0/1     Pending   0          148m   <none>         <none>    <none>           <none>
kube-system   coredns-66f779496c-m869q          0/1     Pending   0          148m   <none>         <none>    <none>           <none>
kube-system   etcd-master1                      1/1     Running   2          148m   10.10.10.11    master1   <none>           <none>
kube-system   kube-apiserver-master1            1/1     Running   2          148m   10.10.10.11    master1   <none>           <none>
kube-system   kube-controller-manager-master1   1/1     Running   2          148m   10.10.10.11    master1   <none>           <none>
kube-system   kube-proxy-kkjzk                  1/1     Running   0          148m   10.10.10.11    master1   <none>           <none>
kube-system   kube-scheduler-master1            1/1     Running   2          148m   10.10.10.11    master1   <none>           <none>
```

发现只有 kube-system 名字空间内存在资源。这几个 pod 是初始化集群时创建的静态 pod，每个 pod 跑了一个服务，即：coredns、etcd、kube-apiserver、kube-controller-manager、kube-proxy 和 kube-

https://www.verylopen.org

scheduler，控制面中的节点都要跑这几个 pod。几个静态 pod（etcd-master、kube-apiserver、kube-controller-manager-master1 和 kube-scheduler-master1 ）的定义文件在 /etc/kubernetes/manifests 目录下。

对于异常的 pod，可用类似这样的命令 kubectl describe pod kube-proxy-h9dcx -n kube-system 查看具体的错误信息，然后针对性的排错，或者参考 § 23.10 小节。

当然我们也可以采用下面的命令查看 Pod 的定义（比如查看 kube-proxy-kkjzk）：

```
[root@master1 ~]# kubectl get pod kube-proxy-dsm98 -n kube-system -o yaml
kind: Pod
metadata:
  creationTimestamp: "2023-08-23T07:13:58Z"
  generateName: kube-proxy-
  labels:
    controller-revision-hash: 67485c9d66
    k8s-app: kube-proxy
    pod-template-generation: "1"
  name: kube-proxy-dsm98
  namespace: kube-system
……
```

先不管 pod 是如何定义的，等以后学过 k8s 中各种资源的定义之后，回过头来看这些输出，你就很快明白了。

另外发现两个跑 coredns 服务的 Pod 还没准备好，且都是挂起状态（Pending），等以后部署完了网络插件后就正常了。

5. 进一步查看某个具体的 pod：

```
[root@master1 ~]# kubectl describe pod kube-apiserver-master1 -n kube-system
Name:                 kube-apiserver-master1
Namespace:            kube-system
Priority:             2000001000
Priority Class Name:  system-node-critical
Node:                 master1/10.10.10.11
Start Time:           Mon, 21 Aug 2023 09:22:32 +0800
Labels:               component=kube-apiserver
                      tier=control-plane
Annotations:          kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 10.10.10.11:6443
                      kubernetes.io/config.hash: af78bd27329fdcf9e91a97ac512f05ce
                      kubernetes.io/config.mirror: af78bd27329fdcf9e91a97ac512f05ce
                      kubernetes.io/config.seen: 2023-08-21T09:22:32.001820324+08:00
                      kubernetes.io/config.source: file
Status:               Running
SeccompProfile:       RuntimeDefault
IP:                   10.10.10.11
IPs:
  IP:           10.10.10.11
Controlled By:  Node/master1
Containers:     ----------------------------------------------------------------------------------------------------------------------------------------#pod 中的容器
  kube-apiserver:
    Container ID:  containerd://03d9584dc7d1e76308f0eb6cfdf12164a3bfea017314b84e5089099c51ce229b
    Image:         registry.aliyuncs.com/google_containers/kube-apiserver:v1.28.2
    Image ID:      registry.aliyuncs.com/google_containers/kube-apiserver@sha256:42ebad4ff7b72c7aa6988a6d4674f9391a80f2e0ba4bf20d705a1f844ba0a5c3
    Port:          <none>
    Host Port:     <none>
    Command:
      kube-apiserver   ----------------------------------------------------------------------------------------------------------------------------------#容器内启动的程序
      --advertise-address=10.10.10.11
      --allow-privileged=true
      --authorization-mode=Node,RBAC
      --client-ca-file=/etc/kubernetes/pki/ca.crt
      --enable-admission-plugins=NodeRestriction
      --enable-bootstrap-token-auth=true
      --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
      --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
      --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
      --etcd-servers=https://127.0.0.1:2379
      --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
      --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
      --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
      --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
      --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
      --requestheader-allowed-names=front-proxy-client
      --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt
      --requestheader-extra-headers-prefix=X-Remote-Extra-
      --requestheader-group-headers=X-Remote-Group
      --requestheader-username-headers=X-Remote-User
      --secure-port=6443
      --service-account-issuer=https://kubernetes.default.svc.cluster.local
      --service-account-key-file=/etc/kubernetes/pki/sa.pub
      --service-account-signing-key-file=/etc/kubernetes/pki/sa.key
      --service-cluster-ip-range=10.96.0.0/16
      --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
      --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
    State:          Running
      Started:      Mon, 21 Aug 2023 09:22:27 +0800
    Ready:          True
    Restart Count:  2
    Requests:
      cpu:        250m
    Liveness:     http-get https://10.10.10.11:6443/livez delay=10s timeout=15s period=10s #success=1 #failure=8
    Readiness:    http-get https://10.10.10.11:6443/readyz delay=0s timeout=15s period=1s #success=1 #failure=3
    Startup:      http-get https://10.10.10.11:6443/livez delay=10s timeout=15s period=10s #success=1 #failure=24
    Environment:  <none>
    Mounts:
      /etc/kubernetes/pki from k8s-certs (ro)
      /etc/pki from etc-pki (ro)
      /etc/ssl/certs from ca-certs (ro)
```

```
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  ca-certs:
    Type:          HostPath (bare host directory volume)
    Path:          /etc/ssl/certs
    HostPathType:  DirectoryOrCreate
  etc-pki:
    Type:          HostPath (bare host directory volume)
    Path:          /etc/pki
    HostPathType:  DirectoryOrCreate
  k8s-certs:
    Type:          HostPath (bare host directory volume)
    Path:          /etc/kubernetes/pki
    HostPathType:  DirectoryOrCreate
QoS Class:         Burstable
Node-Selectors:    <none>
Tolerations:       :NoExecute op=Exists
Events:            <none>
```

可见这个 pod 中只有一个容器 kube-apiserver。

6. 查看 pod 里的容器：

```
[root@master1 ~]# kubectl get pod -o \
 "custom-columns=PodName:.metadata.name,Containers:.spec.containers[*].name,Image:.spec.containers[*].image" \
 --all-namespaces
PodName                          Containers                Image
coredns-66f779496c-5mxc9         coredns                   registry.aliyuncs.com/google_containers/coredns:v1.10.1
coredns-66f779496c-jb2q5         coredns                   registry.aliyuncs.com/google_containers/coredns:v1.10.1
etcd-master1                     etcd                      registry.aliyuncs.com/google_containers/etcd:3.5.9-0
kube-apiserver-master1           kube-apiserver            registry.aliyuncs.com/google_containers/kube-apiserver:v1.28.2
kube-controller-manager-master1  kube-controller-manager   registry.aliyuncs.com/google_containers/kube-controller-manager:v1.28.2
kube-proxy-xsbz8                 kube-proxy                registry.aliyuncs.com/google_containers/kube-proxy:v1.28.2
kube-scheduler-master1           kube-scheduler            registry.aliyuncs.com/google_containers/kube-scheduler:v1.28.2
```

7. 进入一个 pod 内（比如 kube-proxy-kkjzk，参考前面“4. 查看集群中的 pod”查看到的 pod 名称）的容器：

```
查看 pod 内的容器：
[root@master1 ~]# kubectl get pods kube-proxy-kkjzk -n kube-system -o jsonpath={.spec.containers[*].name}
Kube-proxy

进入 pod 内的容器 kube-proxy 中：
[root@master1 ~]# kubectl exec -n kube-system kube-proxy-kkjzk -c kube-proxy -it -- sh
#
# exit

如果 Pod 只有一个容器，也可以这样进入：
[root@master1 ~]# kubectl exec -n kube-system kube-proxy-kkjzk -t -i -- sh
# exit
```

由于容器中缺少 linux 系统中的大部分常用命令，进去后几乎没法操作。

8.查看被监听的端口：

```
[root@master1 ~]# ss -utnlp
Netid     State      Recv-Q     Send-Q         Local Address:Port         Peer Address:Port    Process
udp       UNCONN     0          0                  0.0.0.0:123               0.0.0.0:*        users:(("chronyd",pid=718,fd=7))
udp       UNCONN     0          0              127.0.0.1:323                 0.0.0.0:*        users:(("chronyd",pid=718,fd=5))
udp       UNCONN     0          0                  [::1]:323                    [::]:*        users:(("chronyd",pid=718,fd=6))
tcp       LISTEN     0          4096           127.0.0.1:10257               0.0.0.0:*        users:(("kube-controller",pid=3052,fd=3))
tcp       LISTEN     0          4096           127.0.0.1:10259               0.0.0.0:*        users:(("kube-scheduler",pid=3084,fd=3))
tcp       LISTEN     0          4096           127.0.0.1:10249               0.0.0.0:*        users:(("kube-proxy",pid=3263,fd=11))
tcp       LISTEN     0          4096           127.0.0.1:10248               0.0.0.0:*        users:(("kubelet",pid=3171,fd=20))
tcp       LISTEN     0          4096           127.0.0.1:2381                0.0.0.0:*        users:(("etcd",pid=3072,fd=14))
tcp       LISTEN     0          4096           127.0.0.1:2379                0.0.0.0:*        users:(("etcd",pid=3072,fd=9))
tcp       LISTEN     0          128              0.0.0.0:22                  0.0.0.0:*        users:(("sshd",pid=844,fd=3))
tcp       LISTEN     0          4096           127.0.0.1:44907               0.0.0.0:*        users:(("containerd",pid=848,fd=14))
tcp       LISTEN     0          4096         10.10.10.11:2379                0.0.0.0:*        users:(("etcd",pid=3072,fd=8))
tcp       LISTEN     0          4096         10.10.10.11:2380                0.0.0.0:*        users:(("etcd",pid=3072,fd=7))
tcp       LISTEN     0          4096                 *:6443                        *:*        users:(("kube-apiserver",pid=3073,fd=3))
tcp       LISTEN     0          4096                 *:10250                       *:*        users:(("kubelet",pid=3171,fd=26))
tcp       LISTEN     0          128                [::]:22                     [::]:*        users:(("sshd",pid=844,fd=4))
tcp       LISTEN     0          4096                 *:10256                       *:*        users:(("kube-proxy",pid=3263,fd=8))
```

要存在这些端口：123（chronyd 时钟同步）、2379/2380（etcd 数据库）、6443（kube-apiserver 服务）、10257（kube-controller 服务）、10259（kube-scheduler 服务）、10249/10256（kube-proxy 服务）、10248/10250（kubelet 服务）。

9. 查看各种证书和静态 pod 的配置文件：

```
[root@master1 ~]# tree /etc/kubernetes/
/etc/kubernetes/
├── admin.conf                                                  #kubernetes-admin 用户访问 k8s 集群的配置文件（内嵌客户端证书和私钥）
├── controller-manager.conf                                     #控制器管理器（kube-controller-manager）用到的配置文件（内嵌客户端证书和私钥）
├── kubelet.conf                                                #kubelet 服务的启动配置文件
├── manifests
│   ├── etcd.yaml                                               #etcd 服务的静态 pod 清单文件
│   ├── kube-apiserver.yaml                                     #kube-apiserver 的静态 pod 清单文件
```

```
|   ├── kube-controller-manager.yaml ------------------
```

加入集群成功后可在控制节点上用命令 kubectl get nodes 查看。

```
[root@master1 ~]# kubectl get nodes -o wide
master1   NotReady   control-plane   47h     v1.34.0   10.10.10.11   <none>        Rocky Linux 9   5.14.0-626.el9.x86_64   containerd://1.7.28
worker1   NotReady   <none>          44m     v1.34.0   10.10.10.31   <none>        Rocky Linux 9   5.14.0-626.el9.x86_64   containerd://1.7.28

[root@master1 ~]# kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME                                READY   STATUS    RESTARTS      AGE     IP             NODE      NOMINATED NODE   READINESS
GATES
kube-system   coredns-855c4dd65d-sxb9d            0/1     Pending   0             2d16h   <none>         <none>    <none>           <none>
kube-system   coredns-855c4dd65d-zcpb2            0/1     Pending   0             2d16h   <none>         <none>    <none>           <none>
kube-system   etcd-master1                        1/1     Running   1 (21m ago)   2d16h   10.10.10.11    master1   <none>           <none>
kube-system   kube-apiserver-master1              1/1     Running   1 (21m ago)   2d16h   10.10.10.11    master1   <none>           <none>
kube-system   kube-controller-manager-master1     1/1     Running   1 (21m ago)   2d16h   10.10.10.11    master1   <none>           <none>
kube-system   kube-proxy-cfp2s                    1/1     Running   0             4m1s    10.10.10.31    worker1   <none>           <none>
kube-system   kube-proxy-nvrtm                    1/1     Running   1 (21m ago)   2d16h   10.10.10.11    master1   <none>           <none>
kube-system   kube-scheduler-master1              1/1     Running   1 (21m ago)   2d16h   10.10.10.11    master1   <none>           <none>
```

两个节点都是 NotReady 状态，同时存在两个 pod 的状态是 Pending，这都是没有安装网络插件导致的。另外一个工作节点的角色为<none>，可执行下面的命令修改角色。

设置 worker1 为工作节点角色（打上 node-role.kubernetes.io/worker=worker 标签，其中“node-role.kubernetes.io/worker”为标签名，等号右边的“worker”是标签的值）：

```
[root@master1 ~]# kubectl label node worker1 node-role.kubernetes.io/worker=worker
node/worker1 labeled
```

如果打错了标签，请使用命令“kubectl label node worker1 node-role.kubernetes.io/worker-”删除它，即在标签名后面增加“-”就是删除标签。

```
[root@master1 ~]# kubectl get nodes
NAME      STATUS     ROLES           AGE   VERSION
master1   NotReady   control-plane   18h   v1.34.0
worker1   NotReady   worker          18h   v1.34.0
```

好了，现在集群中存在两个节点，其中 master1 为控制面节点，而 worker1 为工作节点。可参照类似方法继续加入 worker2 和 worker3。

最后集群的节点和 pod 如下：

```
[root@master1 ~]# kubectl get node,pod -A -o wide
NAME           STATUS     ROLES           AGE     VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE          KERNEL-VERSION          CONTAINER-RUNTIME
node/master1   NotReady   control-plane   47h     v1.34.0   10.10.10.11   <none>        Rocky Linux 9   5.14.0-626.el9.x86_64   containerd://1.7.28
node/worker1   NotReady   worker          46m     v1.34.0   10.10.10.31   <none>        Rocky Linux 9   5.14.0-626.el9.x86_64   containerd://1.7.28
node/worker2   NotReady   worker          5m15s   v1.34.0   10.10.10.32   <none>        Rocky Linux 9   5.14.0-626.el9.x86_64   containerd://1.7.28
node/worker3   NotReady   worker          4m58s   v1.34.0   10.10.10.33   <none>        Rocky Linux 9   5.14.0-626.el9.x86_64   containerd://1.7.28

NAMESPACE     NAME                                    READY   STATUS    RESTARTS      AGE     IP             NODE      NOMINATED NODE   READINESS GATES
kube-system   pod/coredns-7cc97dffdd-2zs7r            0/1     Pending   0             47h     <none>         <none>    <none>           <none>
kube-system   pod/coredns-7cc97dffdd-r7j4l            0/1     Pending   0             47h     <none>         <none>    <none>           <none>
kube-system   pod/etcd-master1                        1/1     Running   2 (75m ago)   47h     10.10.10.11    master1   <none>           <none>
kube-system   pod/kube-apiserver-master1              1/1     Running   2 (75m ago)   47h     10.10.10.11    master1   <none>           <none>
kube-system   pod/kube-controller-manager-master1     1/1     Running   2 (75m ago)   47h     10.10.10.11    master1   <none>           <none>
kube-system   pod/kube-proxy-68vp4                    1/1     Running   2 (75m ago)   47h     10.10.10.11    master1   <none>           <none>
kube-system   pod/kube-proxy-j9lrk                    1/1     Running   0             5m15s   10.10.10.32    worker2   <none>           <none>
kube-system   pod/kube-proxy-q4k8p                    1/1     Running   0             46m     10.10.10.31    worker1   <none>           <none>
kube-system   pod/kube-proxy-xtg8j                    1/1     Running   0             4m58s   10.10.10.33    worker3   <none>           <none>
kube-system   pod/kube-scheduler-master1              1/1     Running   2 (75m ago)   47h     10.10.10.11    master1   <none>           <none>
```

表明每个工作节点上都要运行一个特别的 pod，kube-proxy 服务就运行在此 pod 中（实际在一个容器里面），如果十分钟后这个 pod 的状态不是 Running，请先排错（可使用命令“kubectl describe pod --namespace=kube-system <pod 名>”查看失败的原因），大部分原因是下拉镜像失败，请检查工作节点上的 DNS 是不是 8.8.8.8 和 9.9.9.9，如果不是，请参考§2.3 小节修改 DNS。或者参考§23.3 小节从 master1 上导出 kube-proxy:v1.34.0，然后再导入到 worker1 上。

~~防火墙放行 10250/tcp 和 10256/tcp 端口：~~

```
~~[root@allnodes ~]# firewall-cmd --permanent --add-port={10250,10256}/tcp~~
~~[root@allnodes ~]# firewall-cmd --reload~~
```

https://www.verydoc.org

§15.5 安装网络插件

强烈建议给虚拟机生成快照（可取名 before install CNI），因为这一步很可能会失败从而需要重做多次。安装过程中会下载几个容器镜像，如果你的电脑不能访问国外网站或者网速超慢，就会导致安装网络插件失败。

网路插件用于 Pod 组网，所有节点上的 pod 处于一个网络平面之内，这些 pod 是互通的。kubernetes 支持很多网络插件，目前最常用的包括 calico、flannel、Cilium 等，其中 flannel 网络插件使用了虚拟网络技术来实现容器之间的通信，它支持多种网络后端，如 VXLAN、UDP 和 Host-GW；而 calico 是一种基于 BGP 的网络插件，它使用路由表来路由容器之间的流量，支持多种网络拓扑结构，并提供了安全性和网络策略功能；Cilium 是一种基于 eBPF（Extended Berkeley Packet Filter）技术的网络插件，它使用 Linux 内核的动态插件来提供网络功能，如路由、负载均衡、安全性和网络策略等。在 github 上 Cilium 人气最旺，其次是 flannel。在一个集群中，我们只需要安装一个网络插件即可，本书的实战中，我们介绍 flannel 和 calico 两种插件的安装方法。

下面介绍安装 calico 和 flannel 网络插件的方法，在具体实验时只需选择一种即可（本书是选择“方法一”）。

方法一： 安装 flannel 网络插件

1. 准备容器镜像：

flannel 跑在容器中，因此安装此插件时需要下载容器镜像。由于默认的容器镜像要从 docker.io 库中拉下来，但是国内访问这个网站很慢，按照官方的方法安装网络插件，很可能会因为下拉镜像超时而导致失败，为此我们有两种的折中的方法来准备容器镜像，你可选择任何一种。

方法 1：手工下拉容器镜像。

```
下载最新版的网络插件部署配置文件（如果一次失败，请多尝试几次，直到下载成功）：
[root@master1 ~]# wget https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml
……
kube-flannel.yml        100%[=====================================================>]   4.24K  --.-KB/s    in 0s

2024-11-11 09:58:31 (23.4 MB/s) - ‘kube-flannel.yml’ saved [4345/4345]


查看需要下拉哪些容器镜像：
[root@master1 ~]# grep "image: " kube-flannel.yml
        image: ghcr.io/flannel-io/flannel:v0.28.10
        image: ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1-flannel3
        image: ghcr.io/flannel-io/flannel:v0.28.10
```

由此可见，需要下载两个容器镜像，它们分别是：ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1-flannel3 和 ghcr.io/flannel-io/flannel:v0.28.10。

手工下载 ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1-flannel3 容器镜像：

```
[root@master1 ~]# ctr -n k8s.io images pull --platform linux/amd64 ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1-flannel3
ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1-flannel3:                    resolved       |++++++++++++++++++++++++++++++++++++++|
index-sha256:5d4fb9f90389a33b397fba4c8f371454c21aa146696aec46481214892e66c1b8:    done           |++++++++++++++++++++++++++++++++++++++|
manifest-sha256:6e700d30d2c9316b4f54f9d0b1423a2d15290bf082f5d1fbd87f9f25173a374c: done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:af6afc41c64c2dfd64e8963de64c5f5e8ade534cd516efbe12a4c40cd432b842:    done           |++++++++++++++++++++++++++++++++++++++|
config-sha256:962fd97b50f9c6693b3e1ff4786c5643eef5eefafca5f43d3240ae8b87cbac2e:   done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:32b772fa507186eddade1aa8a0f01f5ceacba1fa94a5bb968eb355ac417baca3:    done           |++++++++++++++++++++++++++++++++++++++|
elapsed: 1.8 s                                                            total:  4.6 Mi (2.6 MiB/s)
unpacking linux/amd64 sha256:5d4fb9f90389a33b397fba4c8f371454c21aa146696aec46481214892e66c1b8...
done: 604.605049ms
```

本地镜像必须放在 k8s.io 名字空间内。另外同一个容器镜像支持多种硬件平台，我们现在使用 64 位的 x86 CPU 和 Linux 操作系统，所以参数--platform linux/amd64 指明只下载适合我们这次需要的容器镜像。如果你在 ARM 硬

https://www.yeyopen.org

件（如树莓派）或苹果电脑上做实验，请指明相应的平台参数。

手工下载 ghcr.io/flannel-io/flannel:v0.28.10 容器镜像：

```
root@master1:~# ctr -n k8s.io images pull --platform linux/amd64 ghcr.io/flannel-io/flannel:v0.28.10
ghcr.io/flannel-io/flannel:v0.28.10:                                      resolved       |++++++++++++++++++++++++++++++++++++++|
index-sha256:8bdea8aed559e45f78f52b8f782852c66a791cb04e2b938d5c5f80b97fcdea33:    done           |++++++++++++++++++++++++++++++++++++++|
manifest-sha256:35c519c04fcc6d09672363578db3032a4f557d23ff0ed7cad42728974434f6e1: done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:689ce0ce68b1760d49aceabc4d127777735b6bd981fd0670062b3e946e29a440:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:362a3f4c5c20d1fa186c15d22746d8e541812015f910097dc1c3e556055d6360:    done           |++++++++++++++++++++++++++++++++++++++|
……
layer-sha256:68bbdc6197a01211816b6cf44e77f44a3ae988d785c98a2ebf227679120a16d4:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:4f4fb700ef54461cfa02571ae0db9a0dc1e0cdb5577484a6d75e68dc38e8acc1:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:b56a5545cb8cfee722dbb696c84e3bf51bd45f2463b84731fd1bf08b4620fad5:    done           |++++++++++++++++++++++++++++++++++++++|
elapsed: 2.5 s                                                            total:  27.2 M (10.8 MiB/s)
unpacking linux/amd64 sha256:8bdea8aed559e45f78f52b8f782852c66a791cb04e2b938d5c5f80b97fcdea33...
done: 2.026079227s
```

查看本地的镜像：

```
[root@master1 ~]# crictl images ls
IMAGE                                     TAG                 IMAGE ID            SIZE
ghcr.io/flannel-io/flannel-cni-plugin     v1.9.1-flannel3     f12bfaa8145d8       5.17MB
ghcr.io/flannel-io/flannel                v0.28.10            7b3a80c3d0347       35.6MB
registry.k8s.io/coredns/coredns           v1.14.2             38667dd9be96c       23.3MB
registry.k8s.io/etcd                      3.6.8-0             ee85eb1f0edd2       22.9MB
registry.k8s.io/pause                     3.10.2              4a83b15d3ecfe       320kB
registry.k8s.io/kube-apiserver            v1.36.4             b0f70fa6ec47e       29.8MB
registry.k8s.io/kube-controller-manager   v1.36.4             c27613cea70f9       26.5MB
registry.k8s.io/kube-proxy                v1.36.4             a5280d36e1aa3       30.1MB
registry.k8s.io/kube-scheduler            v1.36.4             11ce0d8bbc73b       20.5MB
```

导出本地的容器镜像到文件：

```
root@master1:~# ctr -n k8s.io images export --platform linux/amd64 flannel-cni-plugin-1.9.1.tar \
 ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1-flannel3
root@master1:~# ctr -n k8s.io images export --platform linux/amd64 flannel-0.28.10.tar \
 ghcr.io/flannel-io/flannel:v0.28.10
```

最后把导出的容器镜像文件 flannel-cni-plugin-1.9.1.tar 和 flannel-0.28.10.tar 拷贝到其他节点：

```
[root@master1 ~]# scp flannel-* worker1:.
[root@master1 ~]# scp flannel-* worker2:.
[root@master1 ~]# scp flannel-* worker3:.
```

方法 2：如果根本没法访问 dcker.io 网站，那只好采用一台能访问外网的电脑去下拉容器镜像了，具体参考§23.1 小节。

2．在所有工作节点上导入镜像：

```
导入两个镜像到 k8s.io 名字空间：
[root@worker? ~]# ctr -n k8s.io images import --platform linux/amd64 flannel-cni-plugin-1.9.1.tar
unpacking ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1 (sha256:bf4b62b131666d040f35a327d906ee5a3418280b68a88d9b9c7e828057210443)...done

[root@worker? ~]# ctr -n k8s.io images import --platform linux/amd64 flannel-0.28.10.tar
unpacking ghcr.io/flannel-io/flannel:v0.28.10 (sha256:5f83f1243057458e2724915739 4e3859cf31cc075354af150d497f2ebc8b54db)...done

检查一下本地的镜像库：
[root@worker? ~]# crictl images ls
IMAGE                                              TAG                 IMAGE ID            SIZE
ghcr.io/flannel-io/flannel-cni-plugin              v1.9.1-flannel1     bb28ded63816e       4.92MB
ghcr.io/flannel-io/flannel                         v0.28.10            e83704a177312       34.1MB
registry.aliyuncs.com/google_containers/kube-proxy v1.36.4             df0860106674d       26MB
registry.aliyuncs.com/google_containers/pause      3.10.2              cd073f4c5f6a8       320kB
```

3．修改网络插件配置（只在 master1 节点上操作，会自动部署到所有工作节点上）：

```
# 确保配置文件中存在红色字体部分：
[root@master1 ~]# vim kube-flannel.yml
……
  net-conf.json: |
    {
      "Network": "10.244.0.0/16"----------------------------------------------------------------#要与集群初始化时的 pod 网相同
      "Backend": {
        "Type": "vxlan"
      }
    }
……
      containers:
```

```
      - name: kube-flannel
        image: ghcr.io/flannel/flannel:v0.28.10
        command:
        - /opt/bin/flanneld
        args:
        - --ip-masq
        - --kube-subnet-mgr
……
```

ens160 网卡就是配置了 10.10.10.11 ip 的网卡，如果电脑只有一块网卡，那就没得选，这时配置文件中的“- --iface=ens160”可以省略。在生产环境中，节点一般都有多张网卡，甚至是多网卡捆绑，这时最好将东西流量和南北流量分开，pod 网显然属于东西流量。还可进一步把存储流量从东西流量中独立出来，存储流量一般走万兆网卡，甚至是多张万兆网卡捆绑。

4．构建 pod 网络（会在所有节点上创建若干 pod 和容器，从而在节点之间组建 Pod 网络）：

```
[root@master1 ~]# kubectl create -f kube-flannel.yml
namespace/kube-flannel created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds created
```

创建了 kube-flannel 名字空间，相关的 pod 就在此名字空间内。

大概过十几秒钟（如果临时下载容器镜像，时间会更长）后查看结果：

```
[root@master1 ~]# kubectl get pods -o wide -A
[root@master1 ~]# kubectl get pods -o wide -A
NAMESPACE      NAME                                READY   STATUS    RESTARTS      AGE    IP             NODE      NOMINATED NODE   READINESS GATES
kube-flannel   kube-flannel-ds-8kmv6               1/1     Running   0             9m52s  10.10.10.33    worker3   <none>           <none>
kube-flannel   kube-flannel-ds-92w86               1/1     Running   0             9m52s  10.10.10.11    master1   <none>           <none>
kube-flannel   kube-flannel-ds-cwgpk               1/1     Running   0             9m52s  10.10.10.31    worker1   <none>           <none>
kube-flannel   kube-flannel-ds-gmcws               1/1     Running   0             9m52s  10.10.10.32    worker2   <none>           <none>
kube-system    coredns-66f779496c-ltrkq            1/1     Running   0             144m   10.244.3.3     worker3   <none>           <none>
kube-system    coredns-66f779496c-w7xgl            1/1     Running   0             144m   10.244.3.2     worker3   <none>           <none>
kube-system    etcd-master1                        1/1     Running   2 (23m ago)   144m   10.10.10.11    master1   <none>           <none>
kube-system    kube-apiserver-master1              1/1     Running   2 (23m ago)   144m   10.10.10.11    master1   <none>           <none>
kube-system    kube-controller-manager-master1     1/1     Running   2 (23m ago)   144m   10.10.10.11    master1   <none>           <none>
kube-system    kube-proxy-j77qt                    1/1     Running   1 (23m ago)   124m   10.10.10.31    worker1   <none>           <none>
kube-system    kube-proxy-lcb24                    1/1     Running   1 (23m ago)   123m   10.10.10.33    worker3   <none>           <none>
kube-system    kube-proxy-vk2tv                    1/1     Running   2 (23m ago)   144m   10.10.10.11    master1   <none>           <none>
kube-system    kube-proxy-vzn4n                    1/1     Running   1 (23m ago)   123m   10.10.10.32    worker2   <none>           <none>
kube-system    kube-scheduler-master1              1/1     Running   2 (23m ago)   144m   10.10.10.11    master1   <none>           <none>
```

即每个节点上运行了一个 pod，pod 中跑 flanneld 进程，此进程通过 flannel.1 虚拟网卡建立 vxlan 隧道，类似 vpn，请参照“§8.2.4 一个简单的 VPN”理解原理。另外发现 worker3 上跑了两个 coredns pod，这不好，删除一个，让集群自动在另一个节点上启动 coredns pod：

```
[root@master1 ~]# kubectl delete pod coredns-66f779496c-w7xgl -n kube-system
pod "coredns-66f779496c-w7xgl" deleted
[root@master1 ~]# kubectl get pods -o wide -A
NAMESPACE      NAME                                READY   STATUS    RESTARTS      AGE    IP             NODE      NOMINATED NODE   READINESS GATES
kube-flannel   kube-flannel-ds-8kmv6               1/1     Running   0             14m    10.10.10.33    worker3   <none>           <none>
kube-flannel   kube-flannel-ds-92w86               1/1     Running   0             14m    10.10.10.11    master1   <none>           <none>
kube-flannel   kube-flannel-ds-cwgpk               1/1     Running   0             14m    10.10.10.31    worker1   <none>           <none>
kube-flannel   kube-flannel-ds-gmcws               1/1     Running   0             14m    10.10.10.32    worker2   <none>           <none>
kube-system    coredns-66f779496c-59nlb            1/1     Running   0             95s    10.244.1.2     worker1   <none>           <none>
kube-system    coredns-66f779496c-ltrkq            1/1     Running   0             148m   10.244.3.3     worker3   <none>           <none>
kube-system    etcd-master1                        1/1     Running   2 (27m ago)   148m   10.10.10.11    master1   <none>           <none>
kube-system    kube-apiserver-master1              1/1     Running   2 (27m ago)   148m   10.10.10.11    master1   <none>           <none>
kube-system    kube-controller-manager-master1     1/1     Running   2 (27m ago)   148m   10.10.10.11    master1   <none>           <none>
kube-system    kube-proxy-j77qt                    1/1     Running   1 (27m ago)   128m   10.10.10.31    worker1   <none>           <none>
kube-system    kube-proxy-lcb24                    1/1     Running   1 (27m ago)   127m   10.10.10.33    worker3   <none>           <none>
kube-system    kube-proxy-vk2tv                    1/1     Running   2 (27m ago)   148m   10.10.10.11    master1   <none>           <none>
kube-system    kube-proxy-vzn4n                    1/1     Running   1 (27m ago)   127m   10.10.10.32    worker2   <none>           <none>
kube-system    kube-scheduler-master1              1/1     Running   2 (27m ago)   148m   10.10.10.11    master1   <none>           <none>
```

READY 列全是 1/1，STATUS 列全是 Running 才表明一切正常，而且两个 coredns 要跑在不同的节点上，如果发现跑在同一个节点上，请删除其中一个 pod，这样集群会自动在群体节点上启动另一个 pod。如果发现一些 pod 不正常，采用命令 kubectl describe pod <pod-name> -n <namespace>查看详细的错误信息，类似：

```
[root@master1 ~]# kubectl describe pod kube-flannel-ds-4pvg9 -n kube-flannel
Name:                 kube-flannel-ds-4pvg9
Namespace:            kube-flannel
Priority:             2000001000
Priority Class Name:  system-node-critical
Service Account:      flannel
Node:                 worker3/10.10.10.33
Start Time:           Mon, 30 Oct 2023 09:54:51 +0800
Labels:               app=flannel
                      controller-revision-hash=6fbbc5c975
                      pod-template-generation=1
                      tier=node
Annotations:          <none>
Status:               Pending
IP:                   10.10.10.33
……
```

```
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  2m43s  default-scheduler  Successfully assigned kube-flannel/kube-flannel-ds-4pvg9 to worker3
  Normal  Pulling    2m44s  kubelet            Pulling image "docker.io/flannel/flannel-cni-plugin:v1.5.1-flannel2 "
  Normal  Pulled     72s    kubelet            Successfully pulled image "docker.io/flannel/flannel-cni-plugin:v1.5.1-flannel2" in 1m31.151s (1m31.151s including waiting)
  Normal  Created    72s    kubelet            Created container install-cni-plugin
  Normal  Started    72s    kubelet            Started container install-cni-plugin
  Normal  Pulling    72s    kubelet            Pulling image "docker.io/flannel/flannel:v0.26.1 "
```

最后一行表明正在下拉容器镜像。主要：四个节点上的防火墙都要关闭（采用命令 systemctl --disable --now firewalld）。

如果因下拉容器镜像失败而导致 pod 运行异常，请手工导入此镜像，然后删除出问题的 pod，k8s 会自动重建 pod。删除 pod 的命令为 kubectl delete pod <pod 名称> -n <pod 所在的名字空间>，如 kubectl delete pod kube-flannel-ds-cwgpk -n kube-flannel 就是删除 kube-flannel 名字空间内的 kube-flannel-ds-cwgpk。

5. 检查创建的各种网元：

(1) master1 节点：

```
[root@master1 ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:0c:29:4e:3e:75 brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.11/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe4e:3e75/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 22:5e:89:da:d9:df brd ff:ff:ff:ff:ff:ff
    inet 10.244.0.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::205e:89ff:feda:d9df/64 scope link
       valid_lft forever preferred_lft forever
```

发现创建了 xvlan 类型的网卡 flannel.1。再查看其详细参数：

```
[root@master1 ~]# ip -d link show flannel.1
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/ether 22:5e:89:da:d9:df brd ff:ff:ff:ff:ff:ff promiscuity 0  allmulti 0 minmtu 68 maxmtu 65535
    vxlan id 1 local 10.10.10.11 dev ens160 srcport 0 0 dstport 8472 nolearning ttl auto ageing 300 udpcsum ……
    inet 10.244.0.0/32 scope global flannel.1
    ……
```

虚拟网卡命名 flannel.<vni>，<vni>为 Vxlan Network Identifier 首字母的缩写，表示 Vxlan 网络标识符，默认为 1。从上面黄色字体部分可以看出：vxlan 的 id 为 1，经本地 ens160 网卡（ip 地址为 10.10.10.11）建立 vxlan 隧道，目的端口为 8472/udp，nolearning 标记禁用源地址学习，这意味着不使用多播，而是使用对等点的静态 L3 条目进行单播。所以节点上的防火墙都要放行 8472/udp 端口，否则跨节点的 pod 之间不能通信。

由 flanneld 添加的永久 ARP 条目：

```
[root@master1 ~]# ip neigh show dev flannel.1
10.244.2.0 lladdr 52:af:5d:46:0c:85 PERMANENT
10.244.3.0 lladdr 1a:22:49:13:b1:5e PERMANENT
10.244.1.0 lladdr ea:a5:51:e5:20:29 PERMANENT
```

其他节点上的 flannel.1 网卡的 ip 和 MAC 地址。

由 flanneld 维护的静态 fdb 数据库：

```
[root@master1 ~]# bridge fdb show flannel.1
01:00:5e:00:00:01 dev ens160 self permanent
33:33:00:00:00:01 dev ens160 self permanent
33:33:ff:4e:3e:75 dev ens160 self permanent
ea:a5:51:e5:20:29 dev flannel.1 dst 10.10.10.31 self permanent
1a:22:49:13:b1:5e dev flannel.1 dst 10.10.10.33 self permanent
52:af:5d:46:0c:85 dev flannel.1 dst 10.10.10.32 self permanent
```

其他节点上的 flannel.1 网卡的 MAC 以及节点 Ip。

查看路由表：

```
[root@master1 ~]# ip r
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.11 metric 100
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink
10.244.3.0/24 via 10.244.3.0 dev flannel.1 onlink
```

（2）worker1 节点：

```
[root@worker1 ~]# ip a
……
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:28:72:40 brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.31/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe28:7240/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether ea:a5:51:e5:20:29 brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::e8a5:51ff:fee5:2029/64 scope link
       valid_lft forever preferred_lft forever
4: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether ca:91:84:5b:cd:0f brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.1/24 brd 10.244.1.255 scope global cni0
       valid_lft forever preferred_lft forever
    inet6 fe80::c891:84ff:fe5b:cd0f/64 scope link
       valid_lft forever preferred_lft forever
5: veth1b2b8c5a@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP group default
    link/ether 7a:00:64:19:2e:3e brd ff:ff:ff:ff:ff:ff link-netns cni-1e173d74-874e-af05-1b28-a4544f2a28c0
    inet6 fe80::7800:64ff:fe19:2e3e/64 scope link
       valid_lft forever preferred_lft forever

[root@worker1 ~]# ip -d a show flannel.1
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether ea:a5:51:e5:20:29 brd ff:ff:ff:ff:ff:ff promiscuity 0  allmulti 0 minmtu 68 maxmtu 65535
    vxlan id 1 local 10.10.10.31 dev ens160 srcport 0 0 dstport 8472 nolearning ttl auto ageing 300 udpcsum ……
    inet 10.244.1.0/32 scope global flannel.1
    ……

当工作节点上跑了应用 pod 时，还会创建 cni0 网桥：
[root@worker1 ~]# ip a show cni0
4: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether ca:91:84:5b:cd:0f brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.1/24 brd 10.244.1.255 scope global cni0
       valid_lft forever preferred_lft forever
    inet6 fe80::c891:84ff:fe5b:cd0f/64 scope link
       valid_lft forever preferred_lft forever

列出 cni0 网桥上的网络接口：
[root@worker1 ~]# ip link show master cni0
5: veth1b2b8c5a@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT group default
    link/ether 7a:00:64:19:2e:3e brd ff:ff:ff:ff:ff:ff link-netns cni-1e173d74-874e-af05-1b28-a4544f2a28c0
```

在工作节点上，每一个使用了单独网路名字空间的 pod 都通过一对 veth 设备接到网桥上，上面最后那条命令的输出红色字体部分，即 cni-1e173d74-874e-af05-1b28-a4544f2a28c0 为网络名字空间，它通过 veth1b2b8c5a 网元连接到网桥上。虚拟网卡 flannel.1 为 overlay 网络的设备，由 flanneld 进程进行 vxlan 报文的处理（封包和解包），不同节点之间的 pod 数据流量都经 flannel.1 设备以隧道的形式发送到对端。关于 veth 设备，请参考“§8.2.2 veth”。

另外一个要注意的地方是：目前只有跑了 coredns 这个 pod 的节点才有 cni0 网桥，现在暂时没有 cni0 网桥的节点等以后跑了其他应用 pod 后，会自动创建 cni0 网桥。

进入网络名字空间 cni-1e173d74-874e-af05-1b28-a4544f2a28c0 查看网卡配置：

```
[root@worker1 ~]# ip netns exec cni-1e173d74-874e-af05-1b28-a4544f2a28c0 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
```

```
2: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 56:73:2e:e6:3b:d7 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.1.2/24 brd 10.244.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::5473:2eff:fee6:3bd7/64 scope link
       valid_lft forever preferred_lft forever
```

可见这个网络名字空间就是跑了 coredns 的那个 pod，2 号网卡 eth0 与宿主机上的 5 号网卡是一对 veth 设备。

```
[root@worker1 ~]# ip r
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.31 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 dev cni0 proto kernel scope link src 10.244.1.1
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink
10.244.3.0/24 via 10.244.3.0 dev flannel.1 onlink

[root@worker1 ~]# ip neigh show dev flannel.1
10.244.0.0 lladdr 22:5e:89:da:d9:df PERMANENT
10.244.3.0 lladdr 1a:22:49:13:b1:5e PERMANENT
10.244.2.0 lladdr 52:af:5d:46:0c:85 PERMANENT
```

(3) worker2 节点：

```
[root@worker2 ~]# ip a
……
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:2c:b3:5f brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.32/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe2c:b35f/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 52:af:5d:46:0c:85 brd ff:ff:ff:ff:ff:ff
    inet 10.244.2.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::50af:5dff:fe46:c85/64 scope link
       valid_lft forever preferred_lft forever

[root@worker2 ~]# ip r
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.32 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.3.0/24 via 10.244.3.0 dev flannel.1 onlink
```

目前 worker2 没有跑应用 pod，所以暂时还没有 cni0 网桥。

(3) worker3 节点：

```
[root@worker3 ~]# ip a
……
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:23:dd:49 brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.33/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe23:dd49/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 1a:22:49:13:b1:5e brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::1822:49ff:fe13:b15e/64 scope link
       valid_lft forever preferred_lft forever
4: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether e2:fc:c1:e5:3e:5e brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.1/24 brd 10.244.3.255 scope global cni0
       valid_lft forever preferred_lft forever
    inet6 fe80::e0fc:c1ff:fee5:3e5e/64 scope link
       valid_lft forever preferred_lft forever
5: veth4c73adfa@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP group default
    link/ether 8e:be:7b:3e:36:17 brd ff:ff:ff:ff:ff:ff link-netns cni-06727bdb-235b-1c8a-bd5a-c8508c337aa5
    inet6 fe80::8cbe:7bff:fe3e:3617/64 scope link
       valid_lft forever preferred_lft forever

[root@worker3 ~]# ip r
```

```
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.33 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink
10.244.3.0/24 dev cni0 proto kernel scope link src 10.244.3.1
```

~~最后所有节点上的防火墙放行 8472/udp 端口：~~

```
~~[root@allnodes ~]# firewall-cmd --permanent --add-port=8472/udp~~
~~success~~
~~[root@allnodes ~]# firewall-cmd --reload~~
~~success~~
```

每一个节点分配 10.244.0.0/16 的一个子网 10.244.*.0/24，比如 master1 的 pod 子网为 10.244.0.0/24、worker1 的 pod 子网为 10.244.1.0/24、worker2 的 pod 子网为 10.244.2.0/24、worker3 的 pod 子网为 10.244.3.0/24。每个节点的网桥 cni0 作为本 pod 子网的网关，请参考图 15-12。

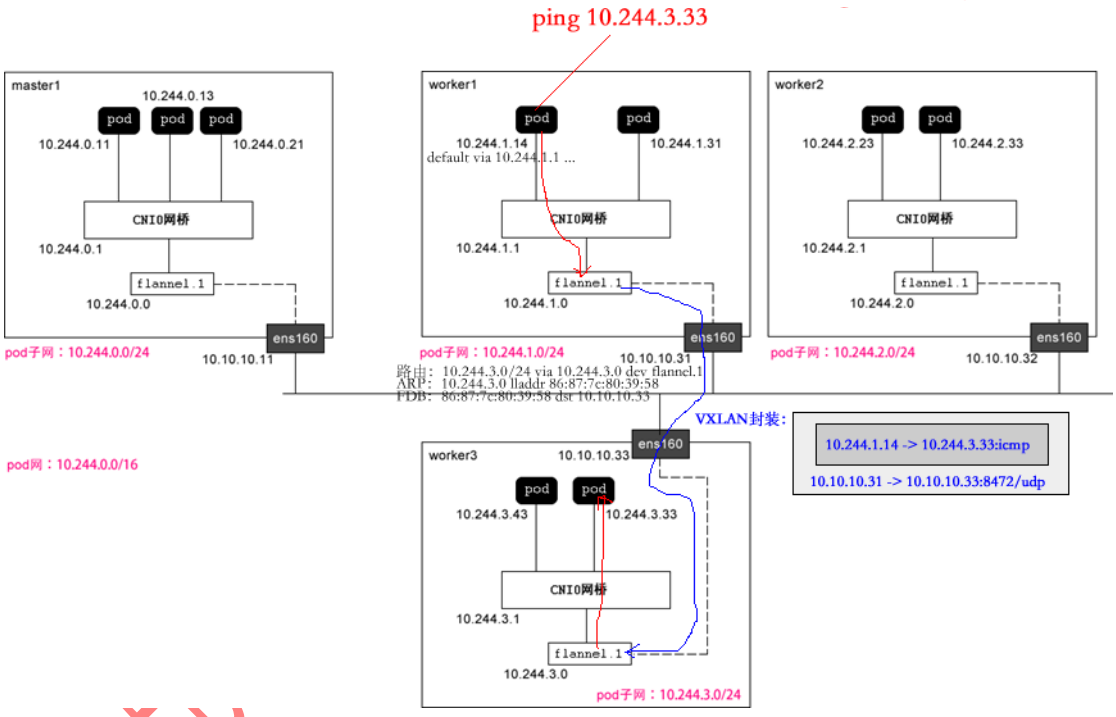


图 15-12 各个节点的 pod 子网

从上图中我们看看在 10.244.1.14 的 pod 中 ping 10.244.3.33，其过程是：根据 pod 的默认路由 default via 10.244.1.1，把 icmp 包发给 CNIO 网桥，CNIO 网桥查看 worker1 的路由表“10.244.3.0/24 via 10.244.3.0 dev flannel.1”，把 icmp 包发给本机的 flannel.1 网卡，flannel.1 做 VXLAN 封装：查询 ARP 表得到“10.244.3.0 lladdr 86:87:7c:80:39:58”，再查询 FDB 表得到“86:87:7c:80:39:58 dst 10.10.10.33”，所以 VXLAN 的外部为 10.10.10.31-> 10.10.10.33:8472/udp。

方法二： 安装 calico 网络插件

安装 calico 网络插件超级简单，只需执行命令“kubectl create -f https://docs.tigera.io/calico/latest/manifests/calico.yaml”即可，但是由于国内网络原因，这条命令很可能会因网络超时而失败，所以稳妥的办法就是先手工把需要的容器镜像拉下来。

在 master1 节点上安装 calico 网络插件，以便给集群组建 pod 网络。

1．下载最新版的 calico 配置文件：

```
[root@master1 ~]# wget https://docs.tigera.io/calico/latest/manifests/calico.yaml
……
Location: https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/calico.yaml [following]
--2024-11-11 14:44:00--  https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/calico.yaml
……
```

浏览最新版的配置文件 https://docs.tigera.io/calico/latest/manifests/calico.yaml，最终会重定向到当前的版本 https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/calico.yaml 。由于国内网络环境，上面的命令可能需要执行多次才会成功一次，或者把网址拷贝到浏览器中，然后把结果粘贴拷贝到 linux 这边来。

2．查看 calico 用到的容器镜像：

```
[root@master1 ~]# grep "image: " calico.yaml
          image: docker.io/calico/cni:v3.26.1
          image: docker.io/calico/cni:v3.26.1
          image: docker.io/calico/node:v3.26.1
          image: docker.io/calico/node:v3.26.1
          image: docker.io/calico/kube-controllers:v3.26.1
```

重复了两个，实际上只用到三个容器镜像：docker.io/calico/cni:v3.26.1 、docker.io/calico/node:v3.26.1 和 docker.io/calico/kube-controllers:v3.26.1。下面先手工拉下这三个镜像。

3．手工下拉容器镜像：

```
[root@master1 ~]# ctr -n k8s.io images pull --platform linux/amd64 docker.io/calico/cni:v3.26.1
docker.io/calico/cni:master:                                      resolved       |++++++++++++++++++++++++++++++++++++++|
index-sha256:a6ecebc865e5900a4558157f6894061f93ee9bb226a1750996a01b2a1110bad8:    done           |++++++++++++++++++++++++++++++++++++++|
……
layer-sha256:77db5f946c95a3ac5d69548ccbb751e6934bd29574ebf80f5574c8e8a4943cf7:    done           |++++++++++++++++++++++++++++++++++++++|
elapsed: 85.7s                                                    total:  88.4 M (1.0 MiB/s)
unpacking linux/amd64 sha256:a6ecebc865e5900a4558157f6894061f93ee9bb226a1750996a01b2a1110bad8...
done: 2.761645797s

[root@master1 ~]# ctr -n k8s.io images pull --platform linux/amd64 docker.io/calico/node:v3.26.1
docker.io/calico/node:master:                                     resolved       |++++++++++++++++++++++++++++++++++++++|
index-sha256:27542b2f2527a82dd596d3383d17a5fecb344578a779c7ff7e4cb150335c2494:    done           |++++++++++++++++++++++++++++++++++++++|
……
config-sha256:c2d64ceaba9c19895304dbe6ead7e885703c0f1296d23b966641118cdfcda289:   done           |++++++++++++++++++++++++++++++++++++++|
elapsed: 110.3s                                                   total:  104.0  (965.6 KiB/s)
unpacking linux/amd64 sha256:27542b2f2527a82dd596d3383d17a5fecb344578a779c7ff7e4cb150335c2494...
done: 4.797612196s

[root@master1 ~]# ctr -n k8s.io images pull --platform linux/amd64 docker.io/calico/kube-controllers:v3.26.1
docker.io/calico/kube-controllers:master:                         resolved       |++++++++++++++++++++++++++++++++++++++|
index-sha256:8177650970580d0ac18dcd7801c835a30a7fb70fde6ef80e873dfd883ec81574:    done           |++++++++++++++++++++++++++++++++++++++|
……
layer-sha256:7a999637f383c77e80b970454938c1aa545932d92789748fb40736f132d8a2ef:    done           |++++++++++++++++++++++++++++++++++++++|
elapsed: 58.7s                                                    total:  31.0 M (541.2 KiB/s)
unpacking linux/amd64 sha256:8177650970580d0ac18dcd7801c835a30a7fb70fde6ef80e873dfd883ec81574...
done: 1.012426431s
```

4．把镜像导出到其他的工作节点上：

```
[root@master1 ~]# ctr -n k8s.io images export --platform linux/amd64 calico_cni-3.26.1.tar \
 docker.io/calico/cni:v3.26.1
[root@master1 ~]# ctr -n k8s.io images export --platform linux/amd64 calico_node-3.26.1.tar \
 docker.io/calico/node:v3.26.1
[root@master1 ~]# ctr -n k8s.io images export --platform linux/amd64 calico_kube-controllers-3.26.1.tar \
 docker.io/calico/kube-controllers:v3.26.1
[root@master1 ~]# ll *.tar
-rw-r--r-- 1 root root 93393408 Aug 24 15:36 calico_cni-3.26.1.tar
-rw-r--r-- 1 root root 32820224 Aug 24 15:40 calico_kube-controllers-3.26.1.tar
-rw-r--r-- 1 root root 86601216 Aug 24 15:37 calico_node-3.26.1.tar

[root@master1 ~]# scp *.tar worker?:.                              #上传到其他所有的工作节点上
```

然后在工作节点上导入：

```
[root@worker? ~]# ctr -n k8s.io images import calico_cni-3.26.1.tar
unpacking docker.io/calico/cni:v3.26.1
(sha256:3be3c67ddba17004c292eafec98cc49368ac273b40b27c8a6621be4471d348d6)...done
[root@worker? ~]# ctr -n k8s.io images import calico_kube-controllers-3.26.1.tar
unpacking docker.io/calico/kube-controllers:v3.26.1
(sha256:01ce29ea8f2b34b6cef904f526baed98db4c0581102f194e36f2cd97943f77aa)...done
[root@worker? ~]# ctr -n k8s.io images import calico_node-3.26.1.tar
unpacking docker.io/calico/node:v3.26.1
```

```
(sha256:8e34517775f319917a0be516ed3a373dbfca650d1ee8e72158087c24356f47fb)...done
```

导入时如果报类似这样的错误 ctr: content digest sha256:22e583d86a570b5a084ed1f27909589de6849977d622cce5d8cfe79df0c5b855: not found，那么导入的命令带上参数--platform linux/amd64 试试。

5. 安装 calico 网络插件：

```
[root@master1 ~]# kubectl create -f calico.yaml
poddisruptionbudget.policy/calico-kube-controllers created
serviceaccount/calico-kube-controllers created
serviceaccount/calico-node created
serviceaccount/calico-cni-plugin created
configmap/calico-config created
customresourcedefinition.apiextensions.k8s.io/bgpconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgpfilters.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/blockaffinities.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/caliconodestatuses.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamblocks.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamconfigs.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamhandles.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipreservations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/kubecontrollersconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networksets.crd.projectcalico.org created
clusterrole.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrole.rbac.authorization.k8s.io/calico-node created
clusterrole.rbac.authorization.k8s.io/calico-cni-plugin created
clusterrolebinding.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrolebinding.rbac.authorization.k8s.io/calico-node created
clusterrolebinding.rbac.authorization.k8s.io/calico-cni-plugin created
daemonset.apps/calico-node created
deployment.apps/calico-kube-controllers created
```

6. 检查新增的网元：

```
[root@master1 ~]# ip a
……
4: tunl0@NONE: <NOARP,UP,LOWER_UP> mtu 1480 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
    inet 10.244.137.64/32 scope global tunl0
       valid_lft forever preferred_lft forever

[root@worker1 ~]# ip a
……
4: tunl0@NONE: <NOARP,UP,LOWER_UP> mtu 1480 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
    inet 10.244.235.128/32 scope global tunl0
       valid_lft forever preferred_lft forever
7: caliaff4b53fb2d@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1480 qdisc noqueue state UP group default qlen 1000
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns cni-82b8baea-c284-1021-6ed5-28d8c7cbea90
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
8: cali0e4a4616cca@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1480 qdisc noqueue state UP group default qlen 1000
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns cni-1664c2da-326f-4a85-bdb4-07df351624a7
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
9: calib782f67fc63@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1480 qdisc noqueue state UP group default qlen 1000
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns cni-6737ef84-70d5-24e7-f15d-f084d708f966
    inet6 fe80::ecee:eeff:feee:eeee/64 scope link
       valid_lft forever preferred_lft forever
```

最后查看集群中的节点状态：

```
[root@master1 ~]# kubectl get nodes -o wide
NAME      STATUS   ROLES           AGE   VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE          KERNEL-VERSION          CONTAINER-RUNTIME
master1   Ready    control-plane   25h   v1.28.2   10.10.10.11   <none>        Rocky Linux 9   5.14.0-350.el9.x86_64   containerd://1.6.22
worker1   Ready    worker          25h   v1.28.2   10.10.10.31   <none>        Rocky Linux 9   5.14.0-350.el9.x86_64   containerd://1.6.22
worker2   Ready    worker          25h   v1.28.2   10.10.10.32   <none>        Rocky Linux 9   5.14.0-350.el9.x86_64   containerd://1.6.22
```

全部准备就绪！

§15.6 操纵 etcd

把容器中的命令 etcdctl 拷贝到宿主机上的/usr/bin 目录中：

```
[root@master1 ~]# find / -name etcdctl
/run/containerd/io.containerd.runtime.v2.task/k8s.io/e4bf4a96a8bee5b0452c0778e393af64ce164a1b9f208f52c1ad2cf1816ecf03/rootfs/usr/local/bin/etcdctl
/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/18/fs/usr/local/bin/etcdctl

[root@master1 ~]# cp /var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/18/fs/usr/local/bin/etcdctl /usr/bin/

[root@master1 ~]# ss -tnlp | grep etcd
LISTEN 0      4096       127.0.0.1:2379       0.0.0.0:*    users:(("etcd",pid=1480,fd=9))
LISTEN 0      4096       127.0.0.1:2381       0.0.0.0:*    users:(("etcd",pid=1480,fd=14))
LISTEN 0      4096     10.10.10.11:2379       0.0.0.0:*    users:(("etcd",pid=1480,fd=8))
LISTEN 0      4096     10.10.10.11:2380       0.0.0.0:*    users:(("etcd",pid=1480,fd=7))

[root@master1 ~]# ll /etc/kubernetes/pki/etcd/
总用量 32
-rw-r--r-- 1 root root 1094  9 月  2 11:54 ca.crt
-rw------- 1 root root 1675  9 月  2 11:54 ca.key
-rw-r--r-- 1 root root 1159  9 月  2 11:54 healthcheck-client.crt
-rw------- 1 root root 1679  9 月  2 11:54 healthcheck-client.key
-rw-r--r-- 1 root root 1196  9 月  2 11:54 peer.crt
-rw------- 1 root root 1675  9 月  2 11:54 peer.key
-rw-r--r-- 1 root root 1196  9 月  2 11:54 server.crt
-rw------- 1 root root 1679  9 月  2 11:54 server.key
```

/etc/kubernetes/pki/etcd/ca.crt 签发了三套证书：healthcheck-client.*、peer.*和 server.*，使用哪一套都行，这里我们使用 healthcheck-client.*证书。

```
[root@master1 ~]# vim ~/.bashrc                                                                    #在文件末尾追加下面几行
export ETCDCTL_ENDPOINTS=https://127.0.0.1:2379 ----------------------------------------#也可以使用 https://10.10.10.11:2379
export ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt
export ETCDCTL_CERT=/etc/kubernetes/pki/etcd/healthcheck-client.crt
export ETCDCTL_KEY=/etc/kubernetes/pki/etcd/healthcheck-client.key

[root@master1 ~]# . .bashrc ------------------------------------------------------------------------#执行.bashrc 并输出里面定义的环境变量
[root@master1 ~]# env | grep ETCDCTL
ETCDCTL_ENDPOINTS=https://127.0.0.1:2379
ETCDCTL_KEY=/etc/kubernetes/pki/etcd/healthcheck-client.key
ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt
ETCDCTL_CERT=/etc/kubernetes/pki/etcd/healthcheck-client.crt

[root@master1 ~]# etcdctl member list -w table
+------------------+---------+---------+---------------------------+---------------------------+------------+
|        ID        | STATUS  |  NAME   |        PEER ADDRS         |       CLIENT ADDRS        | IS LEARNER |
+------------------+---------+---------+---------------------------+---------------------------+------------+
| 36993bb67a1292d7 | started | master1 | https://10.10.10.11:2380 | https://10.10.10.11:2379 |      false |
+------------------+---------+---------+---------------------------+---------------------------+------------+

[root@master1 ~]# etcdctl endpoint status -w table
+------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
|        ENDPOINT        |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
| https://127.0.0.1:2379 | 36993bb67a1292d7 |   3.5.9 |  2.2 MB |      true |      false |         5 |      35059 |              35059 |        |
+------------------------+------------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
```

k8s 把数据保存在“/registry/”为前缀的键中：

```
[root@master1 ~]# etcdctl get / --prefix=true --keys-only | more
/registry/apiregistration.k8s.io/apiservices/v1.

/registry/apiregistration.k8s.io/apiservices/v1.admissionregistration.k8s.io

/registry/apiregistration.k8s.io/apiservices/v1.apiextensions.k8s.io

/registry/apiregistration.k8s.io/apiservices/v1.apps

/registry/apiregistration.k8s.io/apiservices/v1.authentication.k8s.io

/registry/apiregistration.k8s.io/apiservices/v1.authorization.k8s.io

/registry/apiregistration.k8s.io/apiservices/v1.autoscaling

/registry/apiregistration.k8s.io/apiservices/v1.batch

/registry/apiregistration.k8s.io/apiservices/v1.certificates.k8s.io
```

```
/registry/apiregistration.k8s.io/apiservices/v1.coordination.k8s.io

.....
```

etcd 的键组成类似磁盘目录，K8s 的键一级目录都是 registry，二级目录为：

```
[root@master1 ~]# etcdctl get /registry --prefix=true --keys-only | grep -v '^$' | awk -F'/' '{print $3}' | sort | uniq
apiextensions.k8s.io
apiregistration.k8s.io
clusterrolebindings
clusterroles
configmaps
csinodes
deployments
endpointslices
events
flowschemas
jobs
leases
masterleases
minions
namespaces
pods
priorityclasses
prioritylevelconfigurations
ranges
replicasets
rolebindings
roles
serviceaccounts
services
validatingwebhookconfigurations
```

k8s 把 pods 参数保存在/registry/pods 为前缀的键中：

```
[root@master1 ~]# etcdctl get /registry/pods --prefix=true --keys-only | more
/registry/pods/default/busybox

/registry/pods/default/nginx-www-54b468b46-4rnsw

/registry/pods/default/nginx-www-54b468b46-bbz58

/registry/pods/default/nginx-www-54b468b46-swvtk

/registry/pods/kube-flannel/kube-flannel-ds-8kmv6

/registry/pods/kube-flannel/kube-flannel-ds-92w86

/registry/pods/kube-flannel/kube-flannel-ds-cwgpk

/registry/pods/kube-flannel/kube-flannel-ds-gmcws

/registry/pods/kube-system/coredns-66f779496c-59nlb

/registry/pods/kube-system/coredns-66f779496c-ltrkq

/registry/pods/kube-system/coredns-66f779496c-pffrx

/registry/pods/kube-system/etcd-master1

/registry/pods/kube-system/kube-apiserver-master1

/registry/pods/kube-system/kube-controller-manager-master1

/registry/pods/kube-system/kube-proxy-j77qt

/registry/pods/kube-system/kube-proxy-lcb24

/registry/pods/kube-system/kube-proxy-vk2tv

/registry/pods/kube-system/kube-proxy-vzn4n

/registry/pods/kube-system/kube-scheduler-master1
```

备份 etcd：

```
[root@master1 ~]# etcdctl snapshot save "etcd_$(date +'%Y-%m-%d_%H:%M').db"
{"level":"info","ts":"2024-11-14T11:16:51.007416+0800","caller":"snapshot/v3_snapshot.go:65","msg":"created temporary db file","path":"etcd_2024-11-14_11:16.db.part"}
{"level":"info","ts":"2024-11-14T11:16:51.032972+0800","logger":"client","caller":"v3@v3.5.15/maintenance.go:212","msg":"opened snapshot stream; downloading"}
{"level":"info","ts":"2024-11-14T11:16:51.033152+0800","caller":"snapshot/v3_snapshot.go:73","msg":"fetching snapshot","endpoint":"https://127.0.0.1:2379"}
{"level":"info","ts":"2024-11-14T11:16:51.118833+0800","logger":"client","caller":"v3@v3.5.15/maintenance.go:220","msg":"completed snapshot read; closing"}
{"level":"info","ts":"2024-11-14T11:16:51.125287+0800","caller":"snapshot/v3_snapshot.go:88","msg":"fetched snapshot","endpoint":"https://127.0.0.1:2379","size":"2.1 MB","took":"now"}
{"level":"info","ts":"2024-11-14T11:16:51.125461+0800","caller":"snapshot/v3_snapshot.go:97","msg":"saved","path":"etcd_2026-10-21_09:07.db"}
Snapshot saved at etcd_2026-10-21_09:07.db
```

备份出来的文件为 etcd_2026-10-21_09:07.db。

恢复 etcd 时，要 etcd 集群的全部节点上操作，必须先停止 etcd 和 kube-apiserver，把 etcd 和 kube-

apiserver 的配置文件移走，两个 pod 就停止了：

```
[root@master1 ~]# mv /etc/kubernetes/manifests/kube-apiserver.yaml /tmp/
[root@master1 ~]# mv /etc/kubernetes/manifests/etcd.yaml /tmp/

[root@master1 manifests]# ss -utnlp
Netid   State    Recv-Q   Send-Q     Local Address:Port       Peer Address:Port   Process
udp     UNCONN   0        0              0.0.0.0:123             0.0.0.0:*       users:(("chronyd",pid=644,fd=7))
udp     UNCONN   0        0              0.0.0.0:8472            0.0.0.0:*
udp     UNCONN   0        0            127.0.0.1:323             0.0.0.0:*       users:(("chronyd",pid=644,fd=5))
udp     UNCONN   0        0                [::1]:323                [::]:*       users:(("chronyd",pid=644,fd=6))
tcp     LISTEN   0        4096         127.0.0.1:33969           0.0.0.0:*       users:(("containerd",pid=740,fd=14))
tcp     LISTEN   0        128            0.0.0.0:22              0.0.0.0:*       users:(("sshd",pid=730,fd=3))
tcp     LISTEN   0        4096         127.0.0.1:10257           0.0.0.0:*       users:(("kube-controller",pid=25749,fd=3))
tcp     LISTEN   0        4096         127.0.0.1:10259           0.0.0.0:*       users:(("kube-scheduler",pid=25742,fd=3))
tcp     LISTEN   0        4096         127.0.0.1:10248           0.0.0.0:*       users:(("kubelet",pid=803,fd=16))
tcp     LISTEN   0        4096         127.0.0.1:10249           0.0.0.0:*       users:(("kube-proxy",pid=1687,fd=16))
tcp     LISTEN   0        4096        10.10.10.11:2379           0.0.0.0:*       users:(("etcd",pid=27187,fd=8))
tcp     LISTEN   0        4096        10.10.10.11:2380           0.0.0.0:*       users:(("etcd",pid=27187,fd=7))
tcp     LISTEN   0        128              [::]:22                 [::]:*       users:(("sshd",pid=730,fd=4))
tcp     LISTEN   0        4096                *:10256                 *:*       users:(("kube-proxy",pid=1687,fd=12))
tcp     LISTEN   0        4096                *:10250                 *:*       users:(("kubelet",pid=803,fd=13))
```

恢复 etcd 数据：

```
[root@master1 ~]# rm -rf /var/lib/etcd/*                                                    #先删除老的数据
[root@master1 ~]# etcdutl snapshot restore etcd_2026-10-21_09\:07.db --data-dir=/var/lib/etcd    #回复数据
```

启动 etcd 和 kube-apiserver：

```
[root@master1 ~]# cp -v /tmp/etcd.yaml /tmp/kube-apiserver.yaml /etc/kubernetes/manifests/
```

过一会儿查看 Pod 状态：

```
[root@master1 ~]# kubectl get pods -A
NAMESPACE              NAME                                READY   STATUS    RESTARTS       AGE
kube-flannel           kube-flannel-ds-f7m99               1/1     Running   12 (33m ago)   20d
kube-flannel           kube-flannel-ds-km8gt               1/1     Running   13 (33m ago)   20d
kube-flannel           kube-flannel-ds-srpwm               1/1     Running   9 (33m ago)    20d
kube-flannel           kube-flannel-ds-zsfz2               1/1     Running   8 (33m ago)    20d
kube-system            coredns-66f779496c-7cft7            1/1     Running   7 (35m ago)    21d
kube-system            coredns-66f779496c-hd6qr            1/1     Running   2 (34m ago)    6d23h
kube-system            etcd-master1                        1/1     Running   9 (35m ago)    21d
kube-system            kube-apiserver-master1              1/1     Running   9 (35m ago)    21d
kube-system            kube-controller-manager-master1     1/1     Running   9 (35m ago)    21d
kube-system            kube-proxy-5chkf                    1/1     Running   8 (34m ago)    21d
kube-system            kube-proxy-6hphz                    1/1     Running   9 (35m ago)    21d
kube-system            kube-proxy-gks9l                    1/1     Running   9 (34m ago)    21d
kube-system            kube-proxy-tgqx9                    1/1     Running   8 (33m ago)    21d
kube-system            kube-scheduler-master1              1/1     Running   9 (35m ago)    21d
```

https://www.oen.org

§15.7 集群中的证书

要搞清楚集群中各个数字证书的作用，首先要搞清楚集群中存在哪些通信的双方（“服务器/客户端”对），还有就是集群中通信双方都要验证对方身份——双向验证。集群中的通信双方和相应的证书请参考图 15-5 所示。

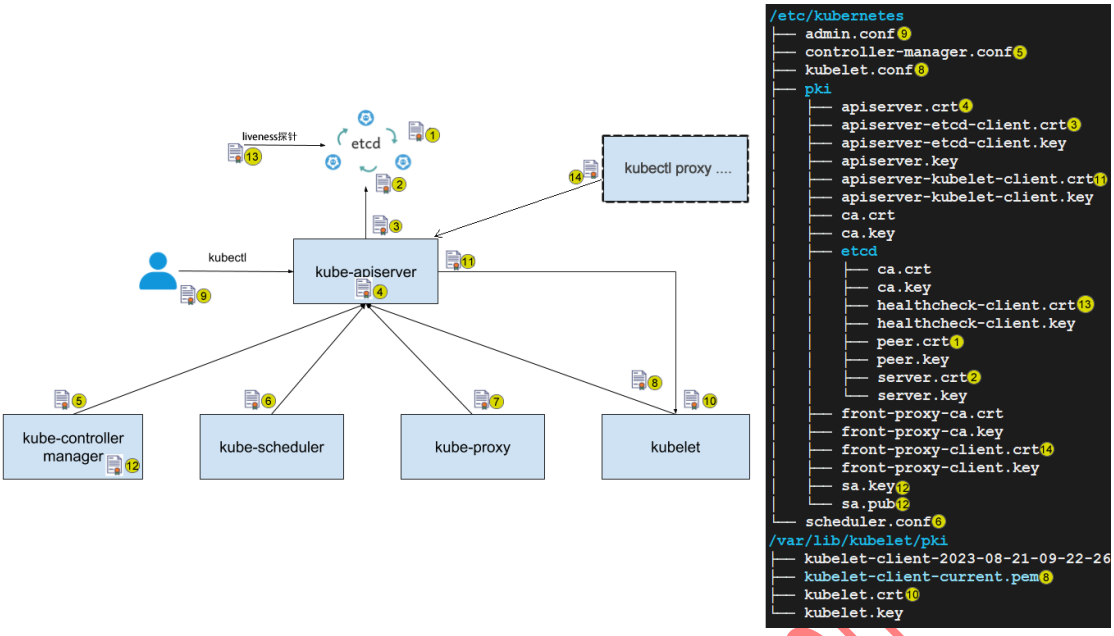


图 15-5 通信双方和证书

图中的箭头方向代表服务器，箭尾方向代表客户端，箭头线两端都有证书图标表示需要双向验证身份（如④→⑤、③→②，比如 server.crt 用来向 kube-apiserver 证明 etcd 数据库服务），否则就是单向验证（如⑨→）。*.crt 为证书文件，*.key 为私钥文件，而*.conf 文件中嵌入了证书和对应的私钥。⑦号不是证书而是令牌（JWT token），这个令牌由 kube-controller-manager 创建并采用 sa.key 私钥签名，然后被挂载到 kube-proxy 进程所在的 pod 上，以后 kube-proxy 访问 kube-apiserver 时需要携带此令牌，kube-apiserver 再使用 sa.pub 公钥验证令牌，这样就完成了身份认证。k8s 集群上的应用（跑在 pod 中）都是携带令牌访问 kube-apiserver。令牌认证大致过程请参考图 15-6。对服务账号的令牌进行认证的例子请参考“§17.1 busybox”小节内的相关内容。

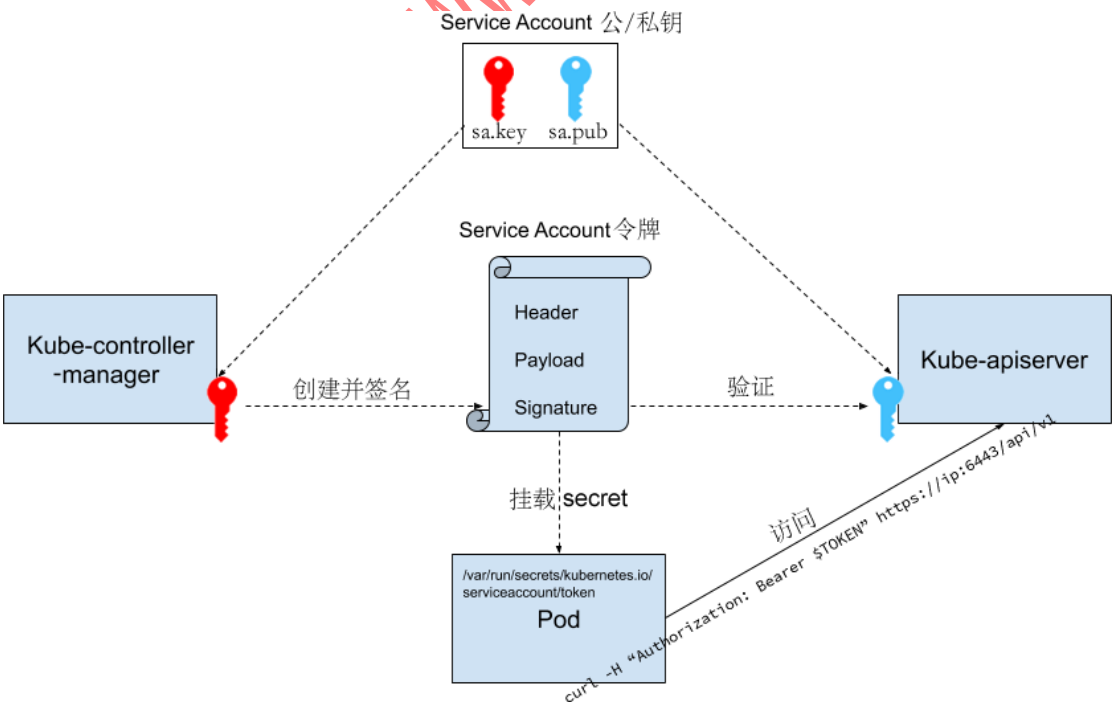


图 15-6 令牌认证过程

一共有三套证书，它们分别是：

1. 由 ca.*给 kubernetes 内部组件签发的证书：admin.conf、controller-manager.conf、kubelet.conf、scheduler.conf、apiserver.crt、apiserver-kubelet-client.crt、kubelet.crt 和 kubelet-client-current.pem。
2. 由 etcd/ca.*给 etcd 签发的证书：apiserver-etcd-client.crt、healthcheck-client.crt、peer.crt、server.crt。
3. 由 font-proxy-ca.*给前端聚合层（或称代理层）签发的证书：front-proxy-client.crt

当然如果嫌麻烦，只采用一个 CA 签发所有证书也是可以的。

查看证书的签发机构和有效期等：

```
[root@master1 ~]# openssl x509 -in /etc/kubernetes/pki/ca.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 2370136742661932141 (0x20e46acf5777446d)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = kubernetes
        Validity
            Not Before: Sep  2 03:49:26 2023 GMT
            Not After : Aug 30 03:54:26 2033 GMT
        Subject: CN = kubernetes
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:c1:c4:98:2c:f9:0e:f3:36:36:70:46:a1:5f:40:
                    ......
                    59:b0:e9:c4:99:fb:e2:75:c8:aa:37:bd:d0:82:31:
                    df:9f
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment, Certificate Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                67:32:D8:84:E2:5C:2F:4B:36:0C:C6:EE:61:2D:1C:A8:5C:B0:C7:BE
            X509v3 Subject Alternative Name:
                DNS:kubernetes
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        48:9b:68:3c:8b:3d:72:cd:e4:e3:2b:f6:25:dd:d7:2d:8e:8f:
        ......
        78:10:40:34
```

由此可知，ca.crt 是自己颁发给自己的证书，即颁发者和证书拥有者都是 kubernetes，有效期十年。

```
[root@master1 ~]# openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 4672818730440541673 (0x40d92fe0afb345e9)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = kubernetes
        Validity
            Not Before: Sep  2 03:49:26 2023 GMT
            Not After : Sep  1 03:54:27 2024 GMT
        Subject: CN = kube-apiserver
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:98:48:15:7a:37:7e:be:ee:33:04:55:15:b6:41:
                    ......
                    ec:61
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Authority Key Identifier:
                67:32:D8:84:E2:5C:2F:4B:36:0C:C6:EE:61:2D:1C:A8:5C:B0:C7:BE
            X509v3 Subject Alternative Name:
                DNS:kubernetes, DNS:kubernetes.default, DNS:kubernetes.default.svc, DNS:kubernetes.default.svc.cluster.local, DNS:master1, IP Address:10.96.0.1, IP
Address:10.10.10.11
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        9a:2b:00:2c:45:53:68:83:cd:05:77:74:2d:22:6d:e9:fd:a0:
        ......
        c3:fc:a8:f8
```

这是由 kubernetes 颁发给 kube-apiserver 的证书，有效期一年。

14 号证书是前端代理层专用的证书，只要执行命令“kubectl proxy [...]”就启动了一个前端代理，它默认监听 localhost:8001，并与 kube-apiserver 建立了 TLS 连接（连接参数如 ip 和端口等在 /etc/kubernetes/admin.conf 或者~/.kube/config 中），这时 kubectl proxy 作为 kube-apiserver 的客户端需要用到 front-proxy-client.crt 证书。此后用户访问 http://localhost:8001 来间接访问 kube-apiserver。下面做一个简单实验：

```
[root@master1 ~]# kubectl proxy
Starting to serve on 127.0.0.1:8001
```

换个登录屏幕，执行命令：

```
[root@master1 ~]# curl localhost:8001/api                                                    #访问前端代理
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.10.10.11:6443"
    }
  ]
}

[root@master1 ~]#kubectl get pods --all-namespaces                                           #直接访问 kube-apiserver
NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE
kube-system   coredns-66f779496c-764xl          0/1     Pending   0          18h
kube-system   coredns-66f779496c-vtnp8          0/1     Pending   0          18h
kube-system   etcd-master1                      1/1     Running   6          18h
kube-system   kube-apiserver-master1            1/1     Running   6          18h
kube-system   kube-controller-manager-master1   1/1     Running   6          18h
kube-system   kube-proxy-dsm98                  1/1     Running   0          18h
kube-system   kube-scheduler-master1            1/1     Running   6          18h

[root@master1 ~]# kubectl get pods --all-namespaces -s localhost:8001                        #访问前端代理
NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE
kube-system   coredns-66f779496c-764xl          0/1     Pending   0          18h
kube-system   coredns-66f779496c-vtnp8          0/1     Pending   0          18h
kube-system   etcd-master1                      1/1     Running   6          18h
kube-system   kube-apiserver-master1            1/1     Running   6          18h
kube-system   kube-controller-manager-master1   1/1     Running   6          18h
kube-system   kube-proxy-dsm98                  1/1     Running   0          18h
kube-system   kube-scheduler-master1            1/1     Running   6          18h

[root@master1 ~]# curl http://localhost:8001/api/v1/namespaces/kube-system/pods              #访问前端代理
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "21984"
  },
  "items": [
......

[root@master1 ~]# curl localhost:8001/api/v1/pods                                            #访问前端代理。返回全部的 pod
```

关于“kubectl proxy”命令的详细用法，请执行“kubectl proxy -h”获取在线帮助。

集群中的证书默认有效期一年，需要在到期前执行下面的命令更新证书：

```
[root@master1 ~]# kubeadm certs renew all
```

§15.8 检查集群状态

检查集群中的 pod 状态：

```
[root@master1 ~]# kubectl get pods --all-namespaces -o wide
NAMESPACE      NAME                                READY   STATUS    RESTARTS      AGE    IP            NODE      NOMINATED NODE   READINESS GATES
kube-flannel   kube-flannel-ds-8kmv6               1/1     Running   0             51m    10.10.10.33   worker3   <none>           <none>
kube-flannel   kube-flannel-ds-92w86               1/1     Running   0             51m    10.10.10.11   master1   <none>           <none>
kube-flannel   kube-flannel-ds-cwgpk               1/1     Running   0             51m    10.10.10.31   worker1   <none>           <none>
kube-flannel   kube-flannel-ds-gmcws               1/1     Running   0             51m    10.10.10.32   worker2   <none>           <none>
kube-system    coredns-66f779496c-59nlb            1/1     Running   0             39m    10.244.1.2    worker1   <none>           <none>
kube-system    coredns-66f779496c-ltrkq            1/1     Running   0             3h5m   10.244.3.3    worker3   <none>           <none>
kube-system    etcd-master1                        1/1     Running   2 (64m ago)   3h6m   10.10.10.11   master1   <none>           <none>
kube-system    kube-apiserver-master1              1/1     Running   2 (64m ago)   3h6m   10.10.10.11   master1   <none>           <none>
kube-system    kube-controller-manager-master1     1/1     Running   2 (64m ago)   3h6m   10.10.10.11   master1   <none>           <none>
kube-system    kube-proxy-j77qt                    1/1     Running   1 (64m ago)   165m   10.10.10.31   worker1   <none>           <none>
kube-system    kube-proxy-lcb24                    1/1     Running   1 (64m ago)   165m   10.10.10.33   worker3   <none>           <none>
kube-system    kube-proxy-vk2tv                    1/1     Running   2 (64m ago)   3h5m   10.10.10.11   master1   <none>           <none>
kube-system    kube-proxy-vzn4n                    1/1     Running   1 (64m ago)   165m   10.10.10.32   worker2   <none>           <none>
kube-system    kube-scheduler-master1              1/1     Running   2 (64m ago)   3h6m   10.10.10.11   master1   <none>           <none>
```

查看一个具体 Pod 的详细信息：

```
[root@master1 ~]# kubectl describe pod coredns-66f779496c-59nlb --namespace kube-system
Name:                 coredns-66f779496c-59nlb
Namespace:            kube-system
Priority:             2000000000
Priority Class Name:  system-cluster-critical
Service Account:      coredns
Node:                 worker1/10.10.10.31
Start Time:           Sat, 02 Sep 2023 14:21:39 +0800
Labels:               k8s-app=kube-dns
                      pod-template-hash=66f779496c
Annotations:          <none>
Status:               Running
IP:                   10.244.1.2
IPs:
  IP:           10.244.1.2
Controlled By:  ReplicaSet/coredns-66f779496c
Containers:
  coredns:
    Container ID:  containerd://dfaa018b1bf707adda6ef155a64c50f01163ed9033198ba5f7a825a0fe214fb6
    Image:         registry.aliyuncs.com/google_containers/coredns:v1.10.1
    Image ID:
registry.aliyuncs.com/google_containers/coredns@sha256:90d3eeb2e2108a14fe2ecbef1bc1b5607834335d99c842a377f338aade9da028
    Ports:         53/UDP, 53/TCP, 9153/TCP
    Host Ports:    0/UDP, 0/TCP, 0/TCP
    Args:
      -conf
      /etc/coredns/Corefile
    State:          Running
      Started:      Sat, 02 Sep 2023 14:23:00 +0800
    Ready:          True
    Restart Count:  0
    Limits:
      memory:  170Mi
    Requests:
      cpu:        100m
      memory:     70Mi
    Liveness:     http-get http://:8080/health delay=60s timeout=5s period=10s #success=1 #failure=5
    Readiness:    http-get http://:8181/ready delay=0s timeout=1s period=10s #success=1 #failure=3
    Environment:  <none>
    Mounts:
      /etc/coredns from config-volume (ro)
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-nnqjm (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  config-volume:
    Type:      ConfigMap (a volume populated by a ConfigMap)
    Name:      coredns
    Optional:  false
  kube-api-access-nnqjm:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   Burstable
Node-Selectors:              kubernetes.io/os=linux
Tolerations:                 CriticalAddonsOnly op=Exists
                             node-role.kubernetes.io/control-plane:NoSchedule
                             node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  40m   default-scheduler  Successfully assigned kube-system/coredns-66f779496c-59nlb to worker1
```

```
  Normal  Pulling    40m   kubelet            Pulling image "registry.aliyuncs.com/google_containers/coredns:v1.10.1"
  Normal  Pulled     39m   kubelet            Successfully pulled image
"registry.aliyuncs.com/google_containers/coredns:v1.10.1" in 1m20.34s (1m20.34s including waiting)
  Normal  Created    39m   kubelet            Created container coredns
  Normal  Started    39m   kubelet            Started container coredns
```

检查节点状态：

```
[root@master1 ~]# kubectl get nodes -o wide
NAME      STATUS   ROLES           AGE     VERSION   INTERNAL-IP    EXTERNAL-IP   OS-IMAGE         KERNEL-VERSION          CONTAINER-RUNTIME
master1   Ready    control-plane   5d22h   v1.31.2   10.10.10.11    <none>        Rocky Linux 9   5.14.0-522.el9.x86_64   containerd://1.7.22
worker1   Ready    worker          21h     v1.31.2   10.10.10.31    <none>        Rocky Linux 9   5.14.0-522.el9.x86_64   containerd://1.7.22
worker2   Ready    worker          21h     v1.31.2   10.10.10.32    <none>        Rocky Linux 9   5.14.0-522.el9.x86_64   containerd://1.7.22
worker3   Ready    worker          21h     v1.31.2   10.10.10.33    <none>        Rocky Linux 9   5.14.0-522.el9.x86_64   containerd://1.7.22
```

再查看路由表：

```
[root@master1 ~]# ip r
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.11 metric 100
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink
10.244.3.0/24 via 10.244.3.0 dev flannel.1 onlink
```

查看本地镜像：

```
[root@master1 ~]# crictl images ls
IMAGE                                                 TAG                 IMAGE ID            SIZE
docker.io/flannel/flannel-cni-plugin                  v1.5.1-flannel2     962fd97b50f9c       4.84MB
docker.io/flannel/flannel                             v0.26.1             4c92107ad4cfe       32.2MB
registry.aliyuncs.com/google_containers/coredns       v1.11.3             c69fa2e9cbf5f       18.6MB
registry.aliyuncs.com/google_containers/etcd          3.6.4-0             2e96e5913fc06       56.9MB
registry.aliyuncs.com/google_containers/kube-apiserver            v1.34.0             9499c9960544e       28MB
registry.aliyuncs.com/google_containers/kube-controller-manager   v1.34.0             0486b6c53a1b5       26.1MB
registry.aliyuncs.com/google_containers/kube-proxy                v1.34.0             505d571f5fd56       30.2MB
registry.aliyuncs.com/google_containers/kube-scheduler            v1.34.0             847c7bc1a5418       20.1MB
registry.aliyuncs.com/google_containers/pause         3.10.1              873ed75102791       320kB
```

如果发现存在其他版本的 k8s 组件，可用命令“crictl rmi <镜像 ID>”删除，以释放更多的磁盘空间，不过不删除也不会影响集群。

检查工作节点：

```
[root@worker1 ~]# ip r
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.31 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 dev cni0 proto kernel scope link src 10.244.1.1
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink
10.244.3.0/24 via 10.244.3.0 dev flannel.1 onlink

[root@worker2 ~]# ip r
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.32 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.3.0/24 via 10.244.3.0 dev flannel.1 onlink

[root@worker3 ~]# ip r
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.33 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink
10.244.3.0/24 dev cni0 proto kernel scope link src 10.244.3.1

[root@master1 ~]# kubectl get node,pod -A -o wide
NAME           STATUS   ROLES           AGE   VERSION   INTERNAL-IP    EXTERNAL-IP   OS-IMAGE                      KERNEL-VERSION                  CONTAINER-RUNTIME
node/master1   Ready    control-plane   25h   v1.36.4   10.10.10.11    <none>        Rocky Linux 9.8 (Blue Onyx)   5.14.0-687.56.1.el9_8.x86_64 (amd64)   containerd://2.4.1
node/worker1   Ready    worker          24h   v1.36.4   10.10.10.31    <none>        Rocky Linux 9.8 (Blue Onyx)   5.14.0-687.56.1.el9_8.x86_64 (amd64)   containerd://2.4.1
node/worker2   Ready    worker          23h   v1.36.4   10.10.10.32    <none>        Rocky Linux 9.8 (Blue Onyx)   5.14.0-687.56.1.el9_8.x86_64 (amd64)   containerd://2.4.1
node/worker3   Ready    worker          23h   v1.36.4   10.10.10.33    <none>        Rocky Linux 9.8 (Blue Onyx)   5.14.0-687.56.1.el9_8.x86_64 (amd64)   containerd://2.4.1

NAMESPACE      NAME                                       READY   STATUS    RESTARTS      AGE   IP              NODE      NOMINATED NODE   READINESS GATES
kube-flannel   pod/kube-flannel-ds-ftf5f                  1/1     Running   2 (108m ago)  22h   10.10.10.32     worker2   <none>           <none>
kube-flannel   pod/kube-flannel-ds-fz9ch                  1/1     Running   2 (108m ago)  22h   10.10.10.11     master1   <none>           <none>
kube-flannel   pod/kube-flannel-ds-fzzdm                  1/1     Running   2 (108m ago)  22h   10.10.10.31     worker1   <none>           <none>
kube-flannel   pod/kube-flannel-ds-zrs8s                  1/1     Running   2 (108m ago)  22h   10.10.10.33     worker3   <none>           <none>
kube-system    pod/coredns-6b5f954497-lz94x               1/1     Running   2 (108m ago)  22h   10.244.3.4      worker3   <none>           <none>
kube-system    pod/coredns-6b5f954497-slgcg               1/1     Running   2 (108m ago)  25h   10.244.2.5      worker2   <none>           <none>
kube-system    pod/etcd-master1                           1/1     Running   4 (108m ago)  25h   10.10.10.11     master1   <none>           <none>
kube-system    pod/kube-apiserver-master1                 1/1     Running   4 (108m ago)  25h   10.10.10.11     master1   <none>           <none>
kube-system    pod/kube-controller-manager-master1        1/1     Running   4 (108m ago)  25h   10.10.10.11     master1   <none>           <none>
kube-system    pod/kube-proxy-6sqw5                       1/1     Running   2 (108m ago)  24h   10.10.10.31     worker1   <none>           <none>
kube-system    pod/kube-proxy-cp5zd                       1/1     Running   2 (108m ago)  25h   10.10.10.11     master1   <none>           <none>
kube-system    pod/kube-proxy-slqrv                       1/1     Running   2 (108m ago)  23h   10.10.10.33     worker3   <none>           <none>
kube-system    pod/kube-proxy-vm72g                       1/1     Running   2 (108m ago)  23h   10.10.10.32     worker2   <none>           <none>
kube-system    pod/kube-scheduler-master1                 1/1     Running   4 (108m ago)  25h   10.10.10.11     master1   <none>           <none>
```

说明一切 OK 了。但是如果出现下面的状态，就有问题。

```
[root@master1 ~]# kubectl get pods --all-namespaces
NAMESPACE     NAME                                       READY   STATUS              RESTARTS       AGE
kube-system   calico-kube-controllers-7549b9dcd8-vqpfl   0/1     Terminating         0              3m28s
kube-system   calico-node-dr74t                          0/1     Running             1 (29s ago)    2m3s
kube-system   calico-node-fdm89                          0/1     Running             0              2m3s
```

```
kube-system   coredns-c676cc86f-4dxn9                  0/1     ContainerCreating   0              5h39m
kube-system   coredns-c676cc86f-j2bng                  0/1     ContainerCreating   0              5h39m
kube-system   etcd-master1                             1/1     Running             0              5h39m
kube-system   kube-apiserver-master1                   1/1     Running             0              5h39m
kube-system   kube-controller-manager-master1          1/1     Running             1 (63s ago)    5h39m
kube-system   kube-proxy-9tgk4                         1/1     Running             0              5h39m
kube-system   kube-proxy-b6m2s                         1/1     Running             1 (27m ago)    31m
kube-system   kube-scheduler-master1                   1/1     Running             1 (63s ago)    5h39m
```

对于运行异常的 pod，排错过程请参考“§23.10 Pod 异常”。

https://www.veryopen.org

第16章 K8s 管理基础

§16.1 k8s 中的网络

§16.1.1 节点网

建议多网卡，分开南北向和东西向流量

§16.1.2 pod 网络

节点网、服务网和 pod 网的关系如图 16-1 所示。

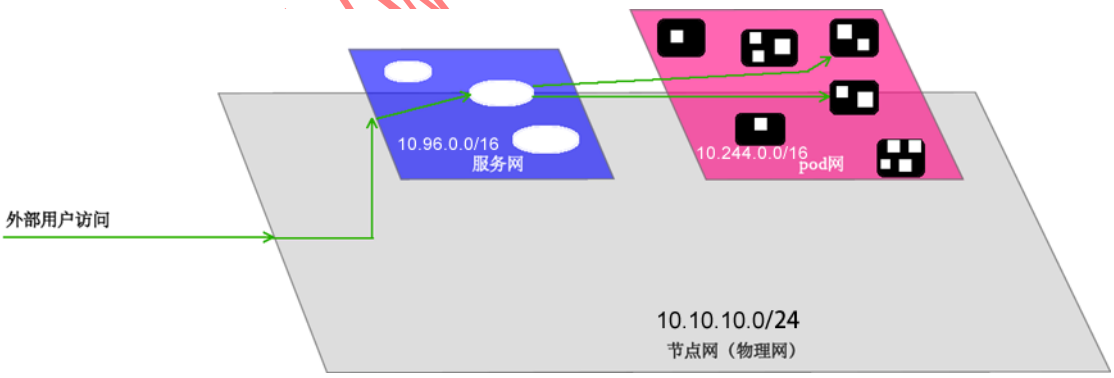


图 16-1 k8s 集群网络

所有的 pod 其网络是相通的，pod 内的容器通过 localhost 连接。pod 之间通过集群网络通讯（非 NAT）。通过 Service API 暴露 pod 中的应用以便外面能访问。Ingress 提供了专门用于暴露 HTTP 应用程序、网站和 RESTful API 的额外功能（相比 Service API）。对于集群内的服务（如数据库）也可以用过 Service API 发布。节点上代理（如 kubelet、系统守护进程等）能与本节点上的 pod 通信。

每一个节点分配 10.244.0.0/16 的一个子网 10.244.*.0/24，比如 master1 的 pod 子网为 10.244.0.0/24、worker1 的 pod 子网为 10.244.1.0/24、worker2 的 pod 子网为 10.244.2.0/24、worker3 的 pod 子网为 10.244.3.0/24。每个节点的网桥 cni0 作为本 pod 子网的网关，请参考图 16-2。

www.veryopen.org

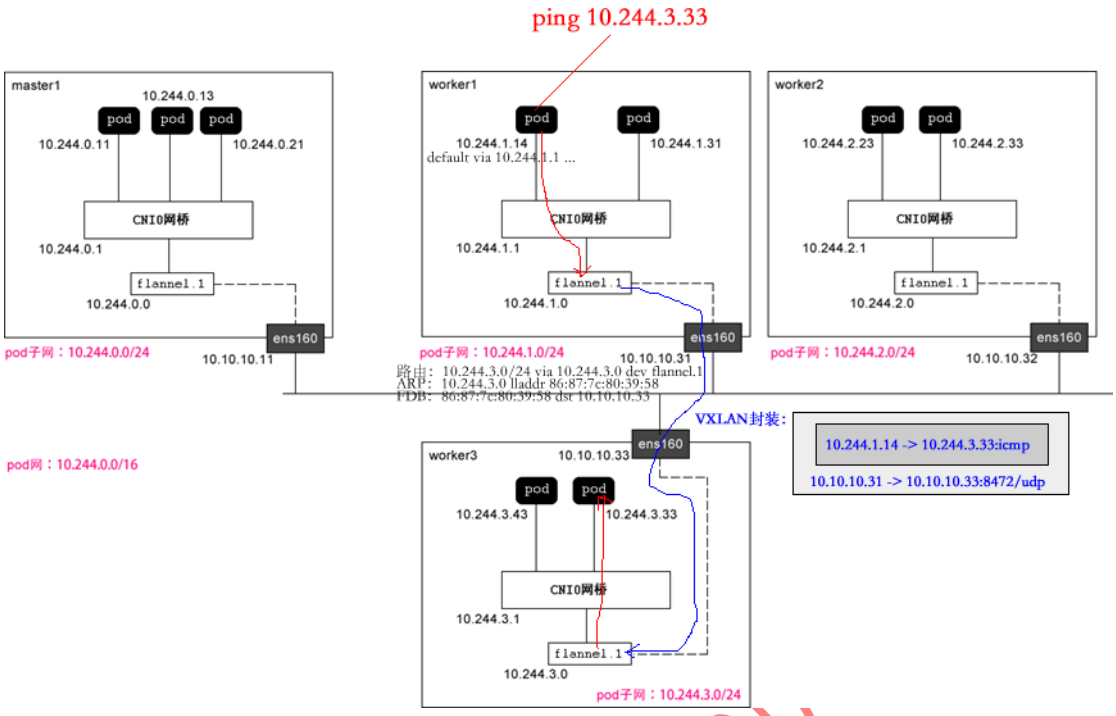


图 16-2 各个节点的 pod 子网

§ 16.1.3 网络插件 CNI

§ 16.1.4 南北网络

https://www.veryo

§16.2 k8s 中的资源对象

§16.2.1 资源对象介绍

k8s 主要的任务是管理集群中的各种资源，包括对资源的登记、分配、回收等，就像计算机操作系统主要是管理进程、内存、cpu、磁盘等资源那样。本小节我们来了解 K8s 集群中的各种资源对象，对于部分重要的资源类型作进一步介绍。官方的资源介绍页面为 https://kubernetes.io/zh-cn/docs/reference/kubernetes-api/。

下面的命令显示当前集群中的全部资源对象。

```
[root@master1 ~]# kubectl api-resources -o wide --sort-by=name
NAME                              SHORTNAMES   APIVERSION                             NAMESPACED   KIND                             VERBS                                                        CATEGORIES
apiservices                                    apiregistration.k8s.io/v1              false        APIService                       create,delete,deletecollection,get,list,patch,update,watch   api-extensions
bindings                                       v1                                     true         Binding                          create
certificatesigningrequests        csr          certificates.k8s.io/v1                 false        CertificateSigningRequest        create,delete,deletecollection,get,list,patch,update,watch
clusterrolebindings                            rbac.authorization.k8s.io/v1           false        ClusterRoleBinding               create,delete,deletecollection,get,list,patch,update,watch
clusterroles                                   rbac.authorization.k8s.io/v1           false        ClusterRole                      create,delete,deletecollection,get,list,patch,update,watch
componentstatuses                 cs           v1                                     false        ComponentStatus                  get,list
configmaps                        cm           v1                                     true         ConfigMap                        create,delete,deletecollection,get,list,patch,update,watch
controllerrevisions                            apps/v1                                true         ControllerRevision               create,delete,deletecollection,get,list,patch,update,watch
cronjobs                          cj           batch/v1                               true         CronJob                          create,delete,deletecollection,get,list,patch,update,watch   all
csidrivers                                     storage.k8s.io/v1                      false        CSIDriver                        create,delete,deletecollection,get,list,patch,update,watch
csinodes                                       storage.k8s.io/v1                      false        CSINode                          create,delete,deletecollection,get,list,patch,update,watch
csistoragecapacities                           storage.k8s.io/v1                      true         CSIStorageCapacity               create,delete,deletecollection,get,list,patch,update,watch
customresourcedefinitions         crd,crds     apiextensions.k8s.io/v1                false        CustomResourceDefinition         create,delete,deletecollection,get,list,patch,update,watch   api-extensions
daemonsets                        ds           apps/v1                                true         DaemonSet                        create,delete,deletecollection,get,list,patch,update,watch   all
deployments                       deploy       apps/v1                                true         Deployment                       create,delete,deletecollection,get,list,patch,update,watch   all
endpoints                         ep           v1                                     true         Endpoints                        create,delete,deletecollection,get,list,patch,update,watch
endpointslices                                 discovery.k8s.io/v1                    true         EndpointSlice                    create,delete,deletecollection,get,list,patch,update,watch
events                            ev           v1                                     true         Event                            create,delete,deletecollection,get,list,patch,update,watch
events                            ev           events.k8s.io/v1                       true         Event                            create,delete,deletecollection,get,list,patch,update,watch
flowschemas                                    flowcontrol.apiserver.k8s.io/v1        false        FlowSchema                       create,delete,deletecollection,get,list,patch,update,watch
horizontalpodautoscalers          hpa          autoscaling/v2                         true         HorizontalPodAutoscaler          create,delete,deletecollection,get,list,patch,update,watch   all
ingressclasses                                 networking.k8s.io/v1                   false        IngressClass                     create,delete,deletecollection,get,list,patch,update,watch
ingresses                         ing          networking.k8s.io/v1                   true         Ingress                          create,delete,deletecollection,get,list,patch,update,watch
jobs                                           batch/v1                               true         Job                              create,delete,deletecollection,get,list,patch,update,watch   all
leases                                         coordination.k8s.io/v1                 true         Lease                            create,delete,deletecollection,get,list,patch,update,watch
limitranges                       limits       v1                                     true         LimitRange                       create,delete,deletecollection,get,list,patch,update,watch
localsubjectaccessreviews                      authorization.k8s.io/v1                true         LocalSubjectAccessReview         create
mutatingwebhookconfigurations                  admissionregistration.k8s.io/v1        false        MutatingWebhookConfiguration     create,delete,deletecollection,get,list,patch,update,watch   api-extensions
namespaces                        ns           v1                                     false        Namespace                        create,delete,get,list,patch,update,watch
networkpolicies                   netpol       networking.k8s.io/v1                   true         NetworkPolicy                    create,delete,deletecollection,get,list,patch,update,watch
nodes                             no           v1                                     false        Node                             create,delete,deletecollection,get,list,patch,update,watch
persistentvolumeclaims            pvc          v1                                     true         PersistentVolumeClaim            create,delete,deletecollection,get,list,patch,update,watch
persistentvolumes                 pv           v1                                     false        PersistentVolume                 create,delete,deletecollection,get,list,patch,update,watch
poddisruptionbudgets              pdb          policy/v1                              true         PodDisruptionBudget              create,delete,deletecollection,get,list,patch,update,watch
pods                              po           v1                                     true         Pod                              create,delete,deletecollection,get,list,patch,update,watch   all
podtemplates                                   v1                                     true         PodTemplate                      create,delete,deletecollection,get,list,patch,update,watch
priorityclasses                   pc           scheduling.k8s.io/v1                   false        PriorityClass                    create,delete,deletecollection,get,list,patch,update,watch
prioritylevelconfigurations                    flowcontrol.apiserver.k8s.io/v1        false        PriorityLevelConfiguration       create,delete,deletecollection,get,list,patch,update,watch
replicasets                       rs           apps/v1                                true         ReplicaSet                       create,delete,deletecollection,get,list,patch,update,watch   all
replicationcontrollers            rc           v1                                     true         ReplicationController            create,delete,deletecollection,get,list,patch,update,watch   all
resourcequotas                    quota        v1                                     true         ResourceQuota                    create,delete,deletecollection,get,list,patch,update,watch
rolebindings                                   rbac.authorization.k8s.io/v1           true         RoleBinding                      create,delete,deletecollection,get,list,patch,update,watch
roles                                          rbac.authorization.k8s.io/v1           true         Role                             create,delete,deletecollection,get,list,patch,update,watch
runtimeclasses                                 node.k8s.io/v1                         false        RuntimeClass                     create,delete,deletecollection,get,list,patch,update,watch
secrets                                        v1                                     true         Secret                           create,delete,deletecollection,get,list,patch,update,watch
selfsubjectaccessreviews                       authorization.k8s.io/v1                false        SelfSubjectAccessReview          create
selfsubjectreviews                             authentication.k8s.io/v1               false        SelfSubjectReview                create
selfsubjectrulesreviews                        authorization.k8s.io/v1                false        SelfSubjectRulesReview           create
serviceaccounts                   sa           v1                                     true         ServiceAccount                   create,delete,deletecollection,get,list,patch,update,watch
services                          svc          v1                                     true         Service                          create,delete,deletecollection,get,list,patch,update,watch   all
statefulsets                      sts          apps/v1                                true         StatefulSet                      create,delete,deletecollection,get,list,patch,update,watch   all
storageclasses                    sc           storage.k8s.io/v1                      false        StorageClass                     create,delete,deletecollection,get,list,patch,update,watch
subjectaccessreviews                           authorization.k8s.io/v1                false        SubjectAccessReview              create
tokenreviews                                   authentication.k8s.io/v1               false        TokenReview                      create
validatingadmissionpolicies                    admissionregistration.k8s.io/v1        false        ValidatingAdmissionPolicy        create,delete,deletecollection,get,list,patch,update,watch   api-extensions
validatingadmissionpolicybindings              admissionregistration.k8s.io/v1        false        ValidatingAdmissionPolicyBinding create,delete,deletecollection,get,list,patch,update,watch   api-extensions
validatingwebhookconfigurations                admissionregistration.k8s.io/v1        false        ValidatingWebhookConfiguration   create,delete,deletecollection,get,list,patch,update,watch   api-extensions
volumeattachments                              storage.k8s.io/v1                      false        VolumeAttachment                 create,delete,deletecollection,get,list,patch,update,watch
```

上面第 1 列 NAME 为资源对象的名称，第 2 列 SHORTNAMES 为简称（有的资源没有简称）。

第 3 列 APIVERSION 为资源对象的版本，格式为“[GROUP/]VERSION”，即属于哪一组的第几个版本，这意味着同一个组允许存在多个版本，如稳定版、阿尔法版（Alpha）和贝塔版（Beta），如果省略[GROUP/]，则表示核心组。其中阿尔法版和贝塔版为开发测试版，建议不要用于生产环境。内建的开发测试版默认被禁用，除非在 kube-apiserver 的配置文件中明确启用或者通过命令行参数启用，如--runtime-config=batch/v2alpha3。APIVERSION 组成 RESTful 的 URL，即/apis/$APIVERSION，比如/apis/**apps/v1**、/apis/**batch/v1**、/apis/**apps/v1**/daemonsets 等，不过核心组的 RESTful 的 URL 可简化为/apps/v1。请看下面的例子：

```
[root@master1 ~]# kubectl get --raw "/apis" | jq                                    #查看全部组的所有版本
{
  "kind": "APIGroupList",
  "apiVersion": "v1",
  "groups": [
    {
      "name": "apiregistration.k8s.io",
      "versions": [
        {
          "groupVersion": "apiregistration.k8s.io/v1",
          "version": "v1"
        }
      ],
      "preferredVersion": {
        "groupVersion": "apiregistration.k8s.io/v1",
        "version": "v1"
      }
    },
    {
      "name": "apps",
      "versions": [
        {
          "groupVersion": "apps/v1",
```

```
          "version": "v1"
        }
      ],
      "preferredVersion": {
        "groupVersion": "apps/v1",
        "version": "v1"
      }
    },
    ……
}

[root@master1 ~]# kubectl get --raw "/apis/apps" | jq                                    #查看 apps 组的所有版本
{
  "kind": "APIGroup",
  "apiVersion": "v1",
  "name": "apps",
  "versions": [
    {
      "groupVersion": "apps/v1",
      "version": "v1"
    }
  ],
  "preferredVersion": {
    "groupVersion": "apps/v1",
    "version": "v1"
  }
}

[root@master1 ~]# kubectl get --raw "/apis/apps/v1/daemonsets" | jq                      #查看核心组中的 daemonsets 资源
{
  "kind": "DaemonSetList",
  "apiVersion": "apps/v1",
  "metadata": {
    "resourceVersion": "143886"
  },
  "items": [
    {
      "metadata": {
        "name": "kube-flannel-ds",
        "namespace": "kube-flannel",
        "uid": "4f13b5c3-c6ec-4333-a19f-b40a96fcfefb",
        "resourceVersion": "114800",
        "generation": 1,
        "creationTimestamp": "2024-11-14T01:40:07Z",
        "labels": {
          "app": "flannel",
          "k8s-app": "flannel",
          "tier": "node"
        },
        "annotations": {
          "deprecated.daemonset.template.generation": "1"
        },
    ……
}
```

其中 preferredVersion 为推荐的版本，稳定版的格式 v*x*，阿尔法版的格式 v*x*alpha*y*，贝塔版格式 v*x*beta*y*，其中*x*为主版本号，*y*为次版本号，二者均为正整数，如 v1、v1alpha2、v2beta3。一个贝塔版的生命期最多 9 个月或者次版本号不能超过 3。

```
[root@master1 ~]# kubectl api-resources --api-group=''                                   #查看核心组的资源类型
NAME                     SHORTNAMES   APIVERSION   NAMESPACED   KIND
bindings                              v1           true         Binding
componentstatuses        cs           v1           false        ComponentStatus
configmaps               cm           v1           true         ConfigMap
endpoints                ep           v1           true         Endpoints
events                   ev           v1           true         Event
limitranges              limits       v1           true         LimitRange
namespaces               ns           v1           false        Namespace
nodes                    no           v1           false        Node
persistentvolumeclaims   pvc          v1           true         PersistentVolumeClaim
persistentvolumes        pv           v1           false        PersistentVolume
pods                     po           v1           true         Pod
podtemplates                          v1           true         PodTemplate
replicationcontrollers   rc           v1           true         ReplicationController
resourcequotas           quota        v1           true         ResourceQuota
secrets                               v1           true         Secret
serviceaccounts          sa           v1           true         ServiceAccount
services                 svc          v1           true         Service
```

可用命令“kubectl api-versions”获取集群支持的所有 API 版本。表 16-1 列举 k8s 支持的资源组。

表 16-1 资源组

组	说明
''	一组核心 API，包含最核心的资源对象，如 pods、podtemplates、services、serviceaccounts、secrets、persistentvolumes、persistentvolumeclaims、endpoints、configmaps 等。在资源对象的定义中设为 apiVersion: v1

admissionregistration.k8s.io	
apiextensions.k8s.io	
apiregistration.k8s.io	
apps	一组常用的 API，包含的资源主要与应用的发布、部署有关，如 deployments、replicasets、statefulsets、daemonsets。在资源对象的定义中设为 apiVersion: apps/v1
authentication.k8s.io	包含身份认证的资源对象，如 tokenreviews
authorization.k8s.io	
autoscaling	包含高性能计算有关的资源对象，如 horizontalpodautoscalers
batch	包含批处理和作业相关的资源对象，如 cronjobs、jobs 等
certificates.k8s.io	包含集群数字证书有关的资源对象，如 certificatesigningrequests
coordination.k8s.io	
discovery.k8s.io	与端点切片相关的资源对象，如 endpointslices。
events.k8s.io	包含事件资源对象 ，如 events
flowcontrol.apiserver.k8s.io	
networking.k8s.io	包含网络安全策略和出口的资源对象，如 ingresses、ingressclasses、networkpolicies。在资源对象的定义中设为 apiVersion: networking.k8s.io/v1
node.k8s.io	
policy	关于 pod 安全策略相关的资源对象，如 poddisruptionbudgets
rbac.authorization.k8s.io	包括访问控制权限有关的资源对象，如 roles、rolebindings、clusterroles、clusterrolebindings
scheduling.k8s.io	与调度有关的资源对象，如 priorityclasses。在资源对象的定义中设为 apiVersion: scheduling.k8s.io/v1
storage.k8s.io	与附加到 Pod 的数据存储相关的资源对象，如 volumeattachments、storageclasses、csinodes

通过 API 组、资源类型、名字空间（对于划分名字空间的资源）和名称可以唯一定位一个资源对象。

第 4 列 NAMESPACED 为是否全局标志，false 表示全局的，true 表示资源只在名字空间内部使用，下面的命令查看所有全局资源类型：

```
[root@master1 ~]# kubectl api-resources --namespaced=false
NAME                              SHORTNAMES   APIVERSION                             NAMESPACED   KIND
componentstatuses                 cs           v1                                     false        ComponentStatus
namespaces                        ns           v1                                     false        Namespace
nodes                             no           v1                                     false        Node
persistentvolumes                 pv           v1                                     false        PersistentVolume
mutatingwebhookconfigurations                  admissionregistration.k8s.io/v1        false        MutatingWebhookConfiguration
validatingadmissionpolicies                    admissionregistration.k8s.io/v1        false        ValidatingAdmissionPolicy
validatingadmissionpolicybindings              admissionregistration.k8s.io/v1        false        ValidatingAdmissionPolicyBinding
validatingwebhookconfigurations                admissionregistration.k8s.io/v1        false        ValidatingWebhookConfiguration
customresourcedefinitions         crd,crds     apiextensions.k8s.io/v1                false        CustomResourceDefinition
apiservices                                    apiregistration.k8s.io/v1              false        APIService
selfsubjectreviews                             authentication.k8s.io/v1               false        SelfSubjectReview
tokenreviews                                   authentication.k8s.io/v1               false        TokenReview
selfsubjectaccessreviews                       authorization.k8s.io/v1                false        SelfSubjectAccessReview
selfsubjectrulesreviews                        authorization.k8s.io/v1                false        SelfSubjectRulesReview
subjectaccessreviews                           authorization.k8s.io/v1                false        SubjectAccessReview
certificatesigningrequests        csr          certificates.k8s.io/v1                 false        CertificateSigningRequest
flowschemas                                    flowcontrol.apiserver.k8s.io/v1        false        FlowSchema
prioritylevelconfigurations                    flowcontrol.apiserver.k8s.io/v1        false        PriorityLevelConfiguration
ingressclasses                                 networking.k8s.io/v1                   false        IngressClass
runtimeclasses                                 node.k8s.io/v1                         false        RuntimeClass
clusterrolebindings                            rbac.authorization.k8s.io/v1           false        ClusterRoleBinding
clusterroles                                   rbac.authorization.k8s.io/v1           false        ClusterRole
priorityclasses                   pc           scheduling.k8s.io/v1                   false        PriorityClass
csidrivers                                     storage.k8s.io/v1                      false        CSIDriver
csinodes                                       storage.k8s.io/v1                      false        CSINode
storageclasses                    sc           storage.k8s.io/v1                      false        StorageClass
volumeattachments                              storage.k8s.io/v1                      false        VolumeAttachment
```

第 5 列 KIND 为资源类型，我们采用 yaml 语言定义资源时需要指定 KIND 和 APIVERSION（可参考§16.2.2 小节）。按照工作分类，资源可分为下面五大类：

(1) 工作负载类资源：指在集群上管理和运行容器的资源，包括 Container、CronJob、DaemonSet、Deployment、DeploymentList、Job、Pod、ReplicaSet、ReplicationController、StatefulSet。
(2) 负载发现与均衡类资源：这类资源用于把集群中的负载暴露出去，包括 Endpoints、EndpointSlice、ClusterCIDR、Ingress、IngressClass、Service。
(3) 配置和存储类资源：用于将初始化数据注入到应用程序中以及让容器把数据保存到外部，包括 ConfigMap、CSIDriver、CSINode、Secret、PersistentVolume、PersistentVolumeClaim、StorageClass、Volume、VolumeAttachment。

www.verypen.org

(4) 集群类资源：定义集群本身的配置方式，这些资源通常仅由集群操作员使用，包括 APIService、Binding、CertificateSigningRequest 、 ClusterRole 、 ClusterRoleBinding 、 ComponentStatus 、FlowSchema、IPAddress、Lease、LocalSubjectAccessReview、Namespace、NetworkPolicy、Node、PersistentVolume、PriorityLevelConfiguration、ResourceQuota、Role、RoleBinding、RuntimeClass、SelfSubjectAccessReview、SelfSubjectReview、SelfSubjectRulesReview、ServiceAccount、StorageVersion、SubjectAccessReview、TokenRequest、TokenReview。

(5) 元数据类资源：用于配置集群内其他资源行为，包括 ClusterTrustBundle、ControllerRevision、CustomResourceDefinition 、 Event 、 HorizontalPodAutoscaler 、 LimitRange 、MutatingWebhookConfiguration 、 PodSchedulingContext 、 PodTemplate 、PodDisruptionBudget 、 PriorityClass 、 ResourceClaim 、 ResourceClaimTemplate 、ResourceClass 、 ValidatingWebhookConfiguration 、 ValidatingAdmissionPolicy 、ValidatingAdmissionPolicyBinding。

对于每一大类中的一些主要资源，我们在下一小节介绍。

第 6 列 VERBS 是该资源允许的操作，这是基于角色的访问控制（RBAC），操作主要是“增/删/改/查”：

(1) “增”：create-新建资源，对应 RESTful 的 POST 或者 kubectl 命令的 create 子命令。

(2) “删”：delete-删除单个资源、deletecollection-删除多个资源，对应 RESTful 的 DELETE 或者 kubectl 命令的 delete 子命令，具体删除一个还是多个资源由选择条件决定，如“kubectl delete pod mypod1”删除一个 pod，而“kubectl delete pods -l db=mysql”删除标记了 db=mysql 的全部 Pod。

(3) “改”：update-修改资源，对应 RESTful 的 PUT 或者 kubectl 命令的 edit 子命令，先获取资源对象，然后修改后提交整个资源对象，apiserver 首先检查修改后的资源对象与原先的资源对象版本是否一致，如果一致就以一个新版本存储修改后的资源，否则报告冲突而拒绝修改。patch-更新资源对象的某些字段，对应 RESTful 的 PATCH 或者 kubectl 命令的 patch 子命令，提交后，apiserver 不比较版本号而直接以新版本存储修改后的资源对象。

(4) “查”：get-获取资源对象（元数据和数据）、list-获取资源对象列表（元数据）、watch-实时监视资源对象的变化，对应 RESTful 的 GET 或者 kubectl 命令的 get 子命令（watch 对应参数--watch=true）。开启 watch 权限必先开启 get 和 list 权限。大家仔细体会下面 RESTful api 调用的区别：

```
GET /apis/apps/v1/namespaces/{namespace}/deployments/{name}                    #get 权限
GET /apis/apps/v1/namespaces/{namespace}/deployments                           #list 权限
GET /apis/apps/v1/deployments                                                  #list 权限
GET /apis/apps/v1/deployments?watch=true                                       #watch 权限
GET /apis/apps/v1/watch/namespaces/{namespace}/deployments?watch=true          #watch 权限
```

上面第一行返回具体部署{name}这个资源对象，第二行以列表方式返回特定名字空间内的所有部署，第三行以列表方式返回全部部署，第四行实时返回被修改的部署资源，第五行实时返回特定名字空间内被改动的部署。

此外针对某些资源类型，还增加了三种操作：回滚版本、更新副本数量、更新状态。

以后随着安装的插件越来越多或人为创建，还会有更多的资源，比如部署 Gateway API 后，会增加 gatewayclasses、gateways、httproutes 和 referencegrants 四种资源。

§16.2.2 定义资源对象

k8s 支持采用 json 或 yaml 语言定义资源对象，官方推荐使用 yaml，不过 yaml 和 json 代码之间很容易相互转换。

https://www.verypan.org

定义某种类型的资源，涉及构建 yaml 语言的清单文件，文件中只包含一系列的“键： 值”对，其中的值要么是 Map 要么是 List，各行通过缩进缩出确定包含关系，这个有点类似 Python 语言的风格，建议每一次缩进两个空格。

可用命令“kubectl explain <资源名>”获取资源的帮助信息——定义资源对象的各个字段说明。例如下面的命令是 pods 资源对象的顶格字段说明：

```
[root@master1 ~]# kubectl explain pods
KIND:       Pod
VERSION:    v1

DESCRIPTION:
    Pod is a collection of containers that can run on a host. This resource is
    created by clients and scheduled onto hosts.

FIELDS:
  apiVersion    <string>
    APIVersion defines the versioned schema of this representation of an object.
    Servers should convert recognized schemas to the latest internal value, and
    may reject unrecognized values. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources

  kind  <string>
    Kind is a string value representing the REST resource this object
    represents. Servers may infer this from the endpoint the client submits
    requests to. Cannot be updated. In CamelCase. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds

  metadata      <ObjectMeta>
    Standard object's metadata. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata

  spec  <PodSpec>
    Specification of the desired behavior of the pod. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

  status        <PodStatus>
    Most recently observed status of the pod. This data may not be up to date.
    Populated by the system. Read-only. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
```

即 pods 资源对象的顶格字段只有五个：apiVersion、kind、metadata、sepc 和 status，其中前两个字段的值为字符串类型，其他三个为对象类型，比如下面的命令进一步获取 spec 的帮助：

```
[root@master1 ~]# kubectl explain pods.spec
KIND:       Pod
VERSION:    v1

FIELD: spec <PodSpec>


DESCRIPTION:
    Specification of the desired behavior of the pod. More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
    PodSpec is a description of a pod.

FIELDS:
  activeDeadlineSeconds <integer>
    Optional duration in seconds the pod may be active on the node relative to
    StartTime before the system will actively try to mark it failed and kill
    associated containers. Value must be a positive integer.

  affinity      <Affinity>
    If specified, the pod's scheduling constraints

  automountServiceAccountToken  <boolean>
    AutomountServiceAccountToken indicates whether a service account token
    should be automatically mounted.

  containers    <[]Container> -required-
    List of containers belonging to the pod. Containers cannot currently be
    added or removed. There must be at least one container in a Pod. Cannot be
    updated.

  dnsConfig     <PodDNSConfig>
    Specifies the DNS parameters of a pod. Parameters specified here will be
    merged to the generated DNS configuration based on DNSPolicy.
  ……
```

其中标注了“-required-”的字段不能省略。本例中 spec 对象中的 containers 的值为列表（或称为数组），列表中的元素为 Container 对象。可以进一步获取 containers 的帮助，如：

```
[root@master1 ~]# kubectl explain pods.spec.containers
KIND:       Pod
VERSION:    v1
```

```
FIELD: containers <[]Container>

DESCRIPTION:
    List of containers belonging to the pod. Containers cannot currently be
    ……

FIELDS:
  args  <[]string>
    Arguments to the entrypoint. The container image's CMD is used if this is
    ……

  command  <[]string>
    Entrypoint array. Not ……
  ……
  livenessProbe  <Probe>
  ……
```

还可以继续运行命令“kubectl explain pods.spec.containers.livenessProbe”获得 livenessProbe 字段的帮助。命令“kubectl explain pods --recursive”列出 pods 资源对象的全部字段。

作为一个例子，下面我们定义一个简单的 pod 资源对象，代码如下：

```
[root@master1 ~]# vim simple-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-busybox
spec:
  containers:
    - image: busybox
      command:
        - sleep
        - "1d"
      imagePullPolicy: IfNotPresent
      name: container-busybox
  restartPolicy: Always
```

运行命令“kubectl explain pods.spec.restartPolicy”可获得字段 restartPolicy 的帮助。

例子中的 pod 取名 pod-busybox，里面运行一个名叫 container-busybox 的容器，此容器的镜像为 busybox

然后创建这个资源对象，让 pod 跑起来：

```
[root@master1 ~]# kubectl create -f simple-pod.yaml
[root@master1 ~]# kubectl get pods -o wide -A
NAMESPACE      NAME                                READY   STATUS             RESTARTS      AGE     IP              NODE      NOMINATED NODE   READINESS GATES
default        pod-busybox                         0/1     ImagePullBackOff   0             62s     10.244.3.5      worker3   <none>           <none>
kube-flannel   kube-flannel-ds-hdmg7               1/1     Running            3 (29m ago)   6d7h    10.10.10.11     master1   <none>           <none>
kube-flannel   kube-flannel-ds-jhsck               1/1     Running            2 (29m ago)   6d7h    10.10.10.32     worker2   <none>           <none>
kube-flannel   kube-flannel-ds-ljbxz               1/1     Running            2 (29m ago)   6d7h    10.10.10.31     worker1   <none>           <none>
kube-flannel   kube-flannel-ds-lrm2k               1/1     Running            2 (29m ago)   6d7h    10.10.10.33     worker3   <none>           <none>
kube-system    coredns-855c4dd65d-djs6k            1/1     Running            2 (29m ago)   6d6h    10.244.2.4      worker2   <none>           <none>
kube-system    coredns-855c4dd65d-lf6rv            1/1     Running            1 (29m ago)   5d1h    10.244.1.3      worker1   <none>           <none>
kube-system    etcd-master1                        1/1     Running            4 (29m ago)   12d     10.10.10.11     master1   <none>           <none>
kube-system    kube-apiserver-master1              1/1     Running            4 (29m ago)   12d     10.10.10.11     master1   <none>           <none>
kube-system    kube-controller-manager-master1     1/1     Running            4 (29m ago)   12d     10.10.10.11     master1   <none>           <none>
kube-system    kube-proxy-5zhfs                    1/1     Running            3 (29m ago)   6d23h   10.10.10.33     worker3   <none>           <none>
kube-system    kube-proxy-j4xgw                    1/1     Running            4 (29m ago)   12d     10.10.10.11     master1   <none>           <none>
kube-system    kube-proxy-kvqx2                    1/1     Running            3 (29m ago)   6d23h   10.10.10.31     worker1   <none>           <none>
kube-system    kube-proxy-sbh44                    1/1     Running            3 (29m ago)   6d23h   10.10.10.32     worker2   <none>           <none>
kube-system    kube-scheduler-master1              1/1     Running            4 (29m ago)   12d     10.10.10.11     master1   <none>           <none>
```

发现 pod 调度失败，因为没法下拉 docker.io/library/busybox 镜像，我们手工把 busybox 导入到 worker3 上，那么 pod 马上就运行正常了。

然后进入 busybox 容器中：

```
[root@master1 ~]# kubectl exec pod-busybox -it -- sh
/ #
/ # exit
```

下面介绍一些常用的资源对象的定义。

一、工作负载类资源

1. Container

描述容器资源对象的字段比较多，因为容器是 k8s 集群的核心，不过我们一般不直接创建容器，而是通过创建 Pod 来定义容器，或者更进一步，通过 Deployment 来创建 Pod。

表 16-1 容器资源对象的字段

字段（类型）	说明
name（string）	容器名字。同一个 Pod 中的容器名字不能相同。如： name: woordpress_1
镜像：	
image（string）	容器镜像的名称。如果省略，则由父层确定，允许带上版本号。如： image: busybox
imagePullPolicy（string）	镜像下拉策略。可取值 Always（总是下拉）、Never（不下拉）、IfNotPresent（如果本地没有就下拉）。如果镜像名带了:latest 标记，则默认是 Always，否则默认为 IfNotPresent。
入口点程序：	
command（[]string）	启动容器时执行的若干命令，如果省略，则使用容器镜像的 ENTRYPOINT 的值，可引用容器的环境$(VAR_NAME)。注意这些命令不在 shell 中执行。如： command: ['printenv']
args（[]string）	启动容器时执行的命令的参数，如果省略，则使用镜像的 CMD 参数定义的值，可引用容器的环境$(VAR_NAME)。如： args: - -n - 99
workingDir（string）	容器的工作目录。如： workingDir: /var/tmp
端口：	
ports（[]ContainerPort）	容器暴露的端口号，ContainerPort 的结构为：containerPort（int32）定义端口，hostIP（string）定义 Ip,hostPort（int32）定义宿主机的端口，name（string）端口别名（同一个 pod 内名字不能相同），protocol（string）协议（可取值 UDP、TCP 或 SCTP，默认是 TCP）。如： ports: - containerPort: 80 #容器内监听的端口 hostPort: 8081 #映射到宿主机的端口，即 8081->80 - containerPort: 3344 #容器内监听的另一个端口 protocol: UDP #端口协议为 UDP
环境变量：	
env（[]EnvVar）	定义容器的环境变量。EnvVar 的结构为： **name（string）** - 变量名 **value（string）** - 变量的值，$(VAR_NAME)引用容器中先前定义的环境变量，而$$(VAR_NAME)产生值“$(VAR_NAME)”。 **valueFrom（EnvVarSource）** - 变量的值取自哪里，value 不空时必须省略本字段。EnvVarSource 的结构为： **configMapKeyRef（ConfigMapKeySelector）** - 从 ConfigMap 中选择一个键。ConfigMapKeySelector 的结构为： **key（string）** - 键名。 **name（string）** -被引用的 ConfigMap 的名称。 **optional（boolean）** - ConfigMap 或其键名是否要事先定义。 **fieldRef（ObjectFieldSelector）** - 从 pod 中选择一个字段，支持 metadata.name、metadata.namespace、metadata.labels['\<KEY>']、metadata.annotations['\<KEY>']、spec.nodeName、spec.serviceAccountName、status.hostIP、status.podIP、status.podIPs。 **resourceFieldRef（ResourceFieldSelector）** - 从容器中选择一种计算资源，目前支持 limits.cpu、limits.memory、limits.ephemeral-storage、requests.cpu、requests.memory 和 requests.ephemeral-storage。 **secretKeyRef（SecretKeySelector）** - 从 pod 所在的名字空间内选择 secret 中的一个键。SecretKeySelector 的结构为： **key（string）** - 键名。 **name（string）** - 被引用的 Secret 的名称。 **optional（boolean）** - Secret 或其键名是否要事先定义。 例子： env: - name: SHELL value: /bin/bash - name: PASSWORD valueFrom: secretKeyRef: name: mySecret key: PASS optional: true
envFrom（[]EnvFromSource）	导入环境变量的数据源列表。数据源中的键必须是 C_IDENTIFIER（以字母或下划线开头，并可能进一步包含字、母数字和下划线）。如果同一个键在多个数据源中，那么最后的那个起作用。字段 env 定义的键优先。EnvFromSource 的结构如下： **configMapRef（ConfigMapEnvSource）** - 从指定的 ConfigMap 中选择键，ConfigMap 的 data 字段的内容（键值对）表示为环境变量。ConfigMapEnvSource 的结构如下： **name（string）** - 被引用的 ConfigMap 的名称。 **optional（boolean）** - ConfigMap 或其键名是否要事先定义。 **prefix（string）** - 附加到 ConfigMap 中每个键名之前的标识符（前缀），必须是 C_IDENTIFIER。本字段可选。 **secretRef（SecretEnvSource）** - 选择一个 Secret 来填充环境变量。目标 Secret 的 data 字段的内容（键值对）表示为环境变量。SecretEnvSource 的结构如下： **name（string）** - 被引用的 Secret 的名称。 **optional（boolean）** - Secret 是否要事先定义。

https://www.verypen.org

	例如： envFrom: - configMapRef: name: myCnfMap optional: false prefix: WP_ - secretRef: name: secret_mysql
存储卷：	
volumeMounts ([]VolumeMount)	定义若干要挂载到容器文件系统中的 Pod 卷。VolumeMount 的结构如下： **mountPath (string)** - 容器内卷的挂载路径，不能包含':'字符。 **name (string)** - 卷名（必须与 Pod 中的 volumes 字段的名称匹配）。 **mountPropagation (string)** - 定义挂载如何从主机传播到容器，及如何反向传播。可取值 None（不传播，这是默认值）、HostToContainer（宿主机上的后续挂载传播到容器）、Bidirectional（宿主机和容器的后续挂载都能传播到对方）。比如容器把宿主机上的目录/opt/dir1 挂载到/mnt，之后宿主机又在/opt/dir1/data 目录上挂载某个文件系统 X，如果 mountPropagation 字段的值为 None，那么在容器中就看不到 X 文件系统上的内容（/mnt/data 为空目录），如果为 HostToContainer，容器中能看到 X 文件系统上的文件（/mnt/data 目录不空）。 **readOnly (boolean)** - true（只读方式挂载）、false（可读写，这是默认值）。 **subPath (string)** - 指定卷中的某个目录路径作为挂载的根，默认为""（即卷的根就是挂载的根）。 **subPathExpr (string)** - 与 subPath 字段类似，不过值中允许引用环境变量，如/data/$(POD_NAME)。subPath 和 subPathExpr 字段不能同时出现。 一个文件系统 X 上往往包含很多目录及多层子目录，通过 subPath 或者 subPathExpr 字段允许只挂载 X 中的某个子目录。 例如下面的例子把 data_vol1 卷上的 mariadb_dir 目录挂载到容器的/var/lib/mariadb 目录上： volumeMounts: - mountPath: /var/lib/mariadb name: data_vol1 subpath: mariadb_dir
volumeDevices ([]VolumeDevice)	定义若干容器使用的块设备。VolumeDevice 的结构如下： **devicePath (string)** - 块设备将被映射到的容器内的路径。 **name (string)** - 块设备名（必须与 Pod 中的 persistentVolumeClaim 的名称匹配）。
计算资源：	
resources （ResourceRequirements）	容器所需的计算资源。容器的计算资源指 CPU 和内存，cpu 的单位是“cpu 核”，1 个 cpu 核等于 1000 毫核，如果小于 1 个核，必须采用毫核单位，比如 500m（相当于 0.5 个核）。内存的最小计量单位是字节，但可以添加后缀 E、P、T、G、M、k，依次是 1000 倍数，即 1M=1000k=1000*1000，或者 Ei、Pi、Ti、Gi、Mi、Ki，依次是倍数 1024，即 1Gi=1024Mi=1024*1024Ki=1024*1024*1024。ResourceRequirements 的结构如下： ● claims ([]ResourceClaim) - 列出容器使用的资源名称，这些资源名称必须在 pod 的 spec.resourceClaims 字段中定义。ResourceClaim 的结构如下： o name (string) - 资源名称。 ● limits (map[string]Quantity) - 定义计算资源量上限。 ● requests (map[string]Quantity) - 定义计算资源量的下限。下限不能超过上限，如果省略了本字段，则下限就是上限。 例如： resources: requests: memory: "64Mi" cpu: "250m" limits: memory: "128Mi" cpu: "500m"
resizePolicy ([]ContainerResizePolicy)	定义容器的计算资源调整策略。ContainerResizePolicy 的结构如下： ● resourceName (string) - 调整策略针对的资源名称，支持 cpu、memory。 ● restartPolicy (string) - 重启策略，会在调整指定资源大小时使用该策略。如果未指定，则默认为 NotRequired。
生命周期：	
lifecycle (Lifecycle)	集群为响应容器生命周期事件应采取的行动。对于 postStart 和 preStop 生命周期处理程序，容器的管理会阻塞，直到操作完成， 除非容器进程失败，在这种情况下处理程序被中止。postStart 和 preStop 称为钩子或回调函数。Lifecycle 的结构如下： ● postStart (LifecycleHandler)：创建容器后立即调用 postStart 处理程序，如果 postStart 失败，则容器将根据其重新启动策略终止并重新启动。LifecycleHandler 的结构如下： o exec（execAction）：要执行的操作。execAction 的结构如下： ▪ command （[]string）：若干要执行的命令，这些命令的工作目录为根目录，且通过 exec 执行而不单独启动一个 Shell 来执行。 o httpGet（HTTPGetAction）：定义要发起的 http 请求。HTTPGetAction 的结构如下： ▪ port （IntOrString）：请求的端口或者名称。 ▪ host （string）：请求的 ip，默认是 pod 的 ip 地址。 ▪ httpHeaders （[]HTTPHeader）：自定义头部。HTTPHeader 的结构体如下： ▫ name (string)：头部字段名称。 ▫ value (string)：头部字段的值。 o path（string）：访问路径。 o scheme（string）：取值 HTTP 或 HTTPS，默认为 HTTP。 ● preStop (LifecycleHandler)：preStop 在容器因 API 请求或管理事件（如存活态探针/启动探针失败、抢占、资源争用等）而终止之前立即调用。 如果容器崩溃或退出，则不会调用处理程序。Pod 的终止宽限期倒计时在 preStop 钩子执行之前开始。无论处理程序的结果如何，容器最终都会在 Pod 的终止宽限期内终止（除非被终结器延迟）。 例如： lifecycle: postStart: httpGet: path: /index.html port: 80 scheme: HTTPS
terminationMessagePath (string)	宿主机上的一个文件路径，被挂载到容器中，容器终止消息被写入到该文件中。pod 内所有容器的总消息长度将限制为 12 KB。默认为 /dev/termination-log。

https://www.vc.open.org

terminationMessagePolicy (string)	定义如何填充终止消息，可取值 File（使用 terminateMessagePath 的内容来填充成功和失败的容器状态消息）、FallbackToLogsOnError（如果 terminateMessagePath 定义的文件为空将使用容器日志输出的最后一块）。
livenessProbe (Probe)	存活探针。如果探针失败，容器将重新启动。探针是 kubelet 对容器执行的健康检查——通过在容器内部执行命令或者向容器发起请求。检查的结果有三种可能：Success（成功）、Failure（失败）、Unknow（未知）。Probe 的结构如下： ● exec（execAction）：参考前面 lifecycle 字段的介绍。 ● httpGet（HTTPGetAction）：参考前面 lifecycle 字段的介绍。 ● tcpSocket（TCPSocketAction）：对容器的 IP 地址上的指定端口执行 TCP 检查，如果端口打开，则诊断被认为是成功的。TCPSocketAction 的结构如下： o port（int32）：gRPC 服务的端口号。值必须在 1 到 65535 之间。 o service（string）：要放置在 gRPC 运行状况检查请求中的服务的名称。如果未指定，则默认行为由 gRPC 定义。 ● grpc（GRPCAction）：采用 gRPC 协议做监测。GRPCAction 的结构如下： o port （IntOrString）：请求的端口或者名称。端口名称在 ports 字段种定义。 ● initialDelaySeconds (int32)：容器启动后启动存活态探针之前的秒数。 ● terminationGracePeriodSeconds （int64）：Pod 需要在探针失败时体面终止所需的时间长度（以秒为单位），也称为宽限期。 ● periodSeconds (int32)：探针的执行周期，大于等于 1 秒，默认为 10 秒。 ● timeoutSeconds (int32)：探针超时秒数，大于等于 1 秒，默认为 1 秒。也就是探针一次执行的最长时间。 ● failureThreshold (int32)：探针最小的连续失败次数，超出此阈值则认为探测失败。大于等于 1，默认为 3。 ● successThreshold (int32)：探针最小的连续成功次数，超过此阈值才会被视为探测成功。值大于等于 1，默认为 1，存活探针和启动探针必须为 1。 例如： `name: ngx-liveness` `image: nginx:latest` `command:` `  - /bin/sh` `  - -c` `  - /usr/sbin/nginx; sleep 60; rm -rf /run/nginx.pid` `livenessProbe:` `  exec:` `  command: [ "/bin/sh", "-c", "test", "-e", "/run/nginx.pid" ]` `  periodSeconds: 30` `  timeoutSeconds: 2` `restartPolicy: Always`
readinessProbe (Probe)	服务就绪探针。如果探针失败，容器将被从服务端点中删除。
startupProbe (Probe)	启动探针监测是否已成功初始化，此探针成功完成之前不会执行其他探针。如果这个探针失败，容器会重新启动，就像存活态探针失败一样。
restartPolicy (string)	定义容器的重启策略。对于 init 容器，本字段的值只为"Always"，对于非 init 容器或者省略本字段的容器，重启策略由 Pod 的重启策略和容器类型来确定。本字段的其他两个值为 onFailure、Never。init 容器现在发展到 Sidecar（边车）容器。
安全上下文：	
securityContext (SecurityContext)	定义了容器应该运行的安全选项。SecurityContext 的字段将覆盖 PodSecurityContext 的同名字段。SecurityContext 的结构如下： ● runAsUser (int64)：运行容器主进程（由 command 字段或容器镜像的 ENTRYPOINT 定义的程序）的用户 ID 号。当 spec.os.name 为 "windows" 时不能设置本字段。 ● runAsNonRoot (boolean)：指示容器是否以非 root 用户身份运行，如果为 true，kubelet 将在运行时验证镜像，以确保它不会以 UID 0（root）身份运行。 如果省略本字段或为 false，则不会执行此类验证。 ● runAsGroup (int64)：运行容器主进程（由 command 字段或容器镜像的 ENTRYPOINT 定义的程序）的组 ID 号。当 spec.os.name 为 "windows" 时不能设置本字段。 ● readOnlyRootFilesystem (boolean)：根文件系统是否只读，默认为 false。注意，spec.os.name 为 "windows" 时不能设置该字段。 ● procMount (string)：默认情况下，为了防止潜在的安全问题，容器运行时会屏蔽容器内 /proc 文件系统的某些文件。然而有时需要访问 /proc 的这些文件，特别是在使用嵌套容器时。可取值 Default（保持标准的容器运行时行为）、Unmasked（不屏蔽/proc 文件系统的文件）。 ● privileged (boolean)：以特权模式运行容器。特权容器中的进程本质上等同于主机上的 root。默认为 false。注意，spec.os.name 为 "windows" 时不能设置此字段。 ● allowPrivilegeEscalation (boolean)：控制进程是否可以获得比其父进程更多的权限。此布尔值直接控制是否在容器进程上设置 no_new_privs 标志。本字段在容器处于以下状态时始终为 true：以特权身份运行、具有 CAP_SYS_ADMIN。请注意，当 spec.os.name 为 "windows" 时，不能设置此字段。 ● capabilities (Capabilities)：运行容器时添加或放弃的权能（Capabilities）。默认为容器运行时所授予的权能集合。 注意，spec.os.name 为 "windows" 时不能设置此字段。Capabilities 的结构如下： o add ([]string)：添加权能。在 Linux 中，权能是对 root 权限的进步一步细分，具体参考表 11-2。 o drop ([]string)：放弃权能。 ● seccompProfile (SeccompProfile)：使用的 seccomp 选项。如果在 Pod 和容器级别都提供了 seccomp 选项，则容器级别的选项会覆盖 Pod 级别的选项。注意，spec.os.name 为 "windows" 时不能设置本字段。SeccompProfile 的结构如下： o type (string)：指示应用哪种 seccomp 配置文件，支持的值有：Localhost（宿主机上配置文件）、RuntimeDefault（使用容器运行时的默认配置文件）、Unconfined（不应用任何配置文件）。 o localhostProfile (string)：定义宿主机上的配置文件，此文件必须位于 kubelet 所配置的 seccomp 配置文件目录下或子目录下。仅当 type 为 "Localhost" 时才配置本项。 ● seLinuxOptions (SELinuxOptions)：定义容器的 SELinux 上下文，如果省略字段，那么将为每个容器分配一个随机的 SELinux 上下文。spec.os.name 为 "windows" 时不能设置此字段。SELinux (Security-Enhanced Linux) 是一种强制访问控制机制——最小权限为原则。需要在宿主机上开启 SELinux。SELinuxOptions 的结构如下： o level （string）：SELinux 级别标签。 o role (string)：SELinux 角色标签。 o type （string）：SELinux 类型标签。 o user （string）：SELinux 用户标签。 ● windowsOptions （WindowsSecurityContextOptions）：针对 windows 容器的安全设置，spec.os.name 为 "linux" 时不能设置本字段。WindowsSecurityContextOptions 的结构如下： o gmsaCredentialSpecName (string)：指定 GMSA 凭证规范的名称。 o gmsaCredentialSpec (string)：指定 GMSA 准入钩子（Webhoot）的位置，此处开始的规范代码中嵌入了由 gmsaCredentialSpecName 字段定义的名字。 o hostProcess (boolean)：是否应作为 "主机进程" 容器运行。如果 hostProcess 为 true，则 hostNetwork 也必须为 true。

https://www.yeyopen.org

	○ runAsUserName （string）：运行容器入口点进程的用户。入口点程序由 command 字段或者镜像的 ENTRYPOINT 参数指定。
调式：	
stdin （boolean）	是否为容器的 stdin 分配缓冲区，如果未设置或设置成 false，那么从容器中的 stdin 读取不到数据（返回 EOF）。默认为 false。stdin 为标准输入文件句柄，默认为键盘。
stdinOnce （boolean）	为 true 时,只能有一个交互会话，并且关闭交互会话后，stdin 也将关闭，直到容器重启。默认为 false。
tty （boolean）	是否分配一个 TTY，如果为 true 则同时必须设置 stdin 字段为 true（此时类似 docker 命令的-it 参数）。默认为 false。

表 16-2 权能表

权能名称	**说明**
SETPCAP	能修改进程的权能
MKNOD	能使用命令 mknod 创建设备文件
AUDIT_WRITE	修改内核的审计日志
CHOWN	修改文件的主人和组
NET_RAW	能使用 RAW 和 PACKET 套接字
DAC_OVERRIDE	绕过文件读、写和执行权限检查。
FOWNER	忽略文件属主 ID 必须和进程用户 ID 相匹配的限制
FSETID	修改文件时不要清除 set-user-ID 和 set-group-ID 权限位
KILL	发送信号不在检查权限
SETGID	允许改变进程的组 ID
SETUID	允许改变进程的用户 ID
NET_BIND_SERVICE	允许绑定到小于 1024 的端口
SYS_CHROOT	允许使用 chroot()系统调用
SETFCAP	能修改文件的权能
上面的权能默认是开启的，而下面的权能默认是关闭的	
SYS_MODULE	能加载和卸载内核模块
SYS_RAWIO	能执行 I/O 端口操作（即调用 iopl() 和 ioperm()函数）
SYS_PACCT	能调用函数 acct()打开或关闭进程审计
SYS_ADMIN	允许执行若干系统管理任务，如加载或卸载文件系统、设置磁盘配额等
SYS_NICE	允许改变进程的优先级（通过调用函数 nice()、setpriority()）
SYS_RESOURCE	忽略资源限制
SYS_TIME	设置系统时间（调用函数 settimeofday()、stime()、adjtimex()）和机器时间
SYS_TTY_CONFIG	允许配置 TTY 设备（调用函数 ioctl()）
AUDIT_CONTROL	启用或禁用内核审计，修改审计过滤规则，获取审计状态和过滤规则
MAC_OVERRIDE	覆盖强制访问控制 (Mandatory Access Control (MAC))
MAC_ADMIN	能配置和改变 MAC 的权限
SYSLOG	能调用 syslog()系统函数
DAC_READ_SEARCH	绕过文件读权限检查，绕过目录读和执行权限检查
LINUX_IMMUTABLE	设置 i-node 的 FS_APPEND_FL 和 FS_IMMUTABLE_FL 标志位
NET_BROADCAST	进行套接字广播并侦听多播
IPC_LOCK	调用 mlock()、mlockall()、mmap()、shmctl()函数所主内存
IPC_OWNER	绕过操纵 System V IPC 对象的权限检查
SYS_PTRACE	调用 ptrace()函数跟踪进程
SYS_BOOT	使用 reboot()和 kexec_load()，重新启动并加载新内核
LEASE	对任意文件建立租约（请参阅 fcntl()函数）
WAKE_ALARM	出发某些事件以唤醒系统

2. Pod 和 PodTemplate

Pod 是可以在工作节点上运行的容器的集合。此资源由客户端创建并调度到工作节点上，也可以通过创建 Deployment、PodTemplate 或 PodTemplateList 来间接创建 pod。描述 pod 的字段参见标 16-2。

表 16-3 pod 资源对象的字段

字段（类型）	说明
apiVersion (string)	值的格式为[GROUP/]VERSION。对于 pod，目前的值固定为 v1
kind (string)	对于 pod，值固定为 Pod
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
spec (PodSpec)	定义 pod 的行为。关于 PodSpec 对象请参考表 16-5
status (PodStatus)	pod 的当前状态。状态信息只能读，具体请参考官网，https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/pod-v1/

表 16-4 标准元数据对象（ObjectMeta）的字段

字段（类型）	说明
name (string)	定义资源对象的名字，在同一个名字空间内必须唯一
generateName (string)	自动生成名字时添加的前缀，本字段与 name 字段互斥
namespace (string)	指定资源存在哪个命名空间内，默认为 default。对于全局性资源，本字段为空
labels ([]string)	定义若干用于组织和分类的标签。
annotations ([]string)	存储和检索任意元数据，用户自定义的一些数据可以保存在这里。
其他一些用于管理的系统字段和只读字段，这里不在赘述。请自行参考官网介绍，https://kubernetes.io/docs/reference/kubernetes-api/common-definitions/object-meta/	

表 16-5 PodSpec 对象的字段

字段（类型）	说明
容器：	
containers ([]Container)	属于 Pod 的容器列表。关于容器资源对象，请参考“1. Container”。
initContainers ([]Container)	属于 Pod 的 Init 容器列表。Init 容器在容器启动之前按顺序执行，如果任何一个 Init 容器发生故障，则认为该 Pod 失败，并根据其 restartPolicy 处理后事。
ephemeralContainers ([]EphemeralContainer)	在此 Pod 中运行的临时容器列表。临时容器可以在现有的 Pod 中运行，以执行用户发起的操作，例如调试。此列表在创建 Pod 时不能指定，也不能通过更新 Pod 规约来修改。要将临时容器添加到现有 Pod，请使用 Pod 的 ephemeralcontainers 子资源。
imagePullSecrets ([]LocalObjectReference)	同一名字空间中 Secret 的引用的列表，用于拉取本 Pod 规范中使用的任何镜像（即从容器镜像库下拉镜像时可能需要的用户和密码），此字段可选。LocalObjectReference 的结构如下： ● name (string)：被引用的名称。 ●
enableServiceLinks (boolean)	是否将有关服务的信息注入到 Pod 的环境变量中，服务连接的语法与 Docker links 的语法相匹配。默认为 true。
os (PodOS)	指定 Pod 中容器的操作系统。如果设置了此属性，则某些 Pod 和容器的字段会受到限制。PodOS 的结构如下： ● name (string)：操作系统名字，可取值 linux 或 windows。如果取值 linux，那么不能设置 securityContext.windowsOptions。如果本字段取值 windows，那么下面的不要设置下面的字段： ✧ spec.hostPID ✧ spec.hostIPC ✧ spec.hostUsers ✧ spec.securityContext.seLinuxOptions ✧ spec.securityContext.seccompProfile ✧ spec.securityContext.fsGroup ✧ spec.securityContext.fsGroupChangePolicy ✧ spec.securityContext.sysctls ✧ spec.shareProcessNamespace ✧ spec.securityContext.runAsUser ✧ spec.securityContext.runAsGroup ✧ spec.securityContext.supplementalGroups ✧ spec.containers[*].securityContext.seLinuxOptions ✧ spec.containers[*].securityContext.seccompProfile ✧ spec.containers[*].securityContext.capabilities ✧ spec.containers[*].securityContext.readOnlyRootFilesystem ✧ spec.containers[*].securityContext.privileged ✧ spec.containers[*].securityContext.allowPrivilegeEscalation ✧ spec.containers[*].securityContext.procMount ✧ spec.containers[*].securityContext.runAsUser

https://www.veryopen.org

	✧ spec.containers[*].securityContext.runAsGroup
存储卷：	
volumes（[]Volume）	能让 pod 内容器挂载的存储卷列表。Volume 的结构参考表 16-6。
调度：	
nodeSelector (map[string]string)	选择在哪个工作节点上运行 pod。nodeSelector 是一个选择算符，这些算符必须取值为 true 才能认为 Pod 适合在节点上运行。 选择算符必须与节点的标签匹配，以便在该节点上调度 Pod。
nodeName（string）	在指定的工作节点上运行 pod，如果满足资源要求的话。
affinity（Affinity）	定义调度约束。Affinity 的结构如下： ● nodeAffinity（NodeAffinity）：定义节点亲和性调度规则。NodeAffinity 的结构如下： ○ preferredDuringSchedulingIgnoredDuringExecution （[]PreferredSchedulingTerm）：选择若干带有权重的工作节点集，类似[{权重 1 节点集 1}，{权重 2 节点集 2},...]，然后算出权重最大的那个工作节点，并把本 Pod 调度到那个节点上。一个工作节点可能出现在多个节点集中，比如在节点集 2 和 5 中，此时权重 2 加上权重 5 就是该节点的总权重。如果没有一个节点满足条件，则忽略本条规则，继续完成调度过程，因此本字段称为软亲和性。PreferredSchedulingTerm 的结构如下： ▪ preference（NodeSelectorTerm）：选择一组节点。null 值或空值的不会匹配任何节点。多个条件按逻辑与组合。NodeSelectorTerm 的结构如下： ▫ matchExpressions （[]NodeSelectorRequirement）：请参考表 11-5 中的 ephemeral 字段。 ▫ matchFields （[]NodeSelectorRequirement）：若干“键：值”对，包含这些键值对的节点将被选中。 ▪ weight（int32）：设定 preference 字段选中的节点集的权重，取值范围在 1 到 100 之间。 ○ requiredDuringSchedulingIgnoredDuringExecution （NodeSelectors）：如果在调度时不满足该字段指定的亲和性要求，则暂停调度 Pod 直到存在满足条件的节点为止，因此本字段称为硬亲和性。另外，如果在 Pod 执行期间不再满足此字段指定的亲和性要求（例如：由于更新），系统可能会将 Pod 从其节点中逐出。NodeSelectors 的结构如下： ▪ nodeSelectorTerms（[]NodeSelectorTerm）：节点选择条件列表。这些条件按逻辑或操作组合。NodeSelectorTerm 的结构参考上面的介绍。 ● podAffinity（PodAffinity）：描述 Pod 亲和性调度规则（例如，将本 Pod 与其他一些 Pod 放在同一节点、区域等）。PodAffinity 的结构如下： ○ preferredDuringSchedulingIgnoredDuringExecution（[]WeightedPodAffinityTerm）：与上面的 preferredDuringSchedulingIgnoredDuringExecution 字段理解起来差不多，不过选择节点集的方法不同。WeightedPodAffinityTerm 的结构如下： ▪ podAffinityTerm（PodAffinityTerm）：在一些特定的名字空间中选择一组 pod，这组 pod 所在的工作节点的集合不妨设为 X，凡是 X 中包含了标签的值等于 topologyKey 字段的值的工作节点组成节点集 n。PodAffinityTerm 的结构如下： ▫ topologyKey（string）：定义目标工作节点应包含的标签的值。本字段不允许为空。 ▫ labelSelector （LabelSelector）：通过标签查询选择一组 pod。 ▫ namespaceSelector （LabelSelector）：通过标签查询选择一组名字空间。空的选择算符（{}）匹配所有名字空间。 ▫ namespaces （[]string）：列出若干名字空间。最终的名字空间为本字段所列举的名字空间和 namespaceSelector 所选定的名字空间的“并”集。本字段的值为 null 或空，以及 namespaceSelector 字段的值为 null 均表示“本 Pod 的名字空间”。 ▪ weight（int32）：指定节点集 n 的权重（即权重 n），取值范围在 1 到 100 之间。 ○ requiredDuringSchedulingIgnoredDuringExecution（[]PodAffinityTerm）：在一些特定的名字空间中选择一组 Pod（当本列表中有多个元素时，每个 podAffinityTerm 对应的节点集做“交”运算，即必须满足所有条件），当前 Pod 应该与该 Pod 集合位于同一位置（亲和性）。PodAffinityTerm 的结构如下，各个字段的意义同上。 ▪ topologyKey（string）。 ▪ labelSelector （LabelSelector）。 ▪ namespaceSelector （LabelSelector）。 ▪ namespaces （[]string）。 ● podAntiAffinity（PodAntiAffinity）：描述 Pod 反亲和性调度规则（例如，避免将此 Pod 与其他一些 Pod 放在相同的节点、区域等）。PodAntiAffinity 的结构如下： ○ preferredDuringSchedulingIgnoredDuringExecution（[]WeightedPodAffinityTerm）：请参考上面的同名字段。 ○ requiredDuringSchedulingIgnoredDuringExecution（[]PodAffinityTerm）：请参考上面的同名字段。
tolerations（[]Toleration）	定义 Pod 的容忍度。在 pod 和 node 资源对象上都可以设置污点“键:值”对，利用污点可以避开调度本 pod 到某些节点上。Toleration 的结构如下： ● key（string）：污点字段的键名。如果为空，则匹配所有的污点键，且 operator 字段必须为 Exists。 ● operator（string）：表示 key 与 value 字段之间的关系。有效的取值是 Exists 或 Equal，默认为 Equal。Exists 相当于 value 为某种通配符，因此 Pod 可以容忍特定类别的所有污点。 ● value（string）：污点值。如果 operator 字段为 Exists，则此 value 值应为空。 ● effect（string）：指定污点匹配成功后的行为。空值意味著所有的行为，允许的值为 NoSchedule（不调度）、PreferNoSchedule（尽量不调度） 和 NoExecute（不调度并且还要驱逐节点上已有的 pod）之一。 ● tolerationSeconds（int64）：容忍污点的时间长度（以秒为单位）。默认情况下，此字段未被设置，这意味著会一直能够容忍对应污点（不会发生驱逐操作）。零值和负值会被系统当做 0 值处理（立即触发驱逐事件）。注意：effect 字段取值 NoExecute 时，本字段才有效。
schedulerName（string）	指定调度器，如果省略本字段，则使用默认调度器来调度本 Pod。
runtimeClassName（string）	引用 node.k8s.io 组中的一个 RuntimeClass 资源对象，该资源对象被用来运行本 Pod。 如果指定的 RuntimeClass 不存在，则 Pod 将不会运行。如果本字段未设置或为空，将使用默认的 RuntimeClass。通过本字段可以实现在同一个集群中采用多个容器运行时。
priorityClassName（string）	定义 Pod 的优先级。"system-node-critical" 和 "system-cluster-critical" 是两个特殊关键字， 分别表示两个最高优先级（前者更高）。任何其他名称都必须事先通过创建 PriorityClass 对象来定义。如果省略本字段，则优先级为默认值。如果没有默认值，则为零。
priority（int32）	以数字表示的优先级，值越大优先级越高。准入控制器会基于 priorityClassName 字段的值来填充本字段。
preemptionPolicy（string）	定义抢占优先级较低的 Pod 的策略，可取值 "Never"（不抢占）、"PreemptLowerPriority" （抢占。这是默认值）。
topologySpreadConstraints ([]TopologySpreadConstraint)	定义一组 pod 如何分布在一组拓扑域中。本字段的列表中的条目是“与”的关系。TopologySpreadConstraint 的结构如下： ● topologyKey（string）：指定一个节点标签的键名。标签中包含此键名且键值相同的所有节点被认为在同一个拓扑（topology）中。我们习惯将每个〈键，值〉视为一个 "桶（Bucket）"，并尽量保持每个桶中运行数量均衡的

https://www.kevynen.org

	Pod。我们定义域（domain）为拓扑的特例，此外，我们定义一个候选域为其节点与 nodeAffinityPolicy 和 nodeTaintsPolicy 字段的要求匹配的域。例如，如果 topologyKey 是 "kubernetes.io/hostname"，则每个节点都是该拓扑的域，而如果 topologyKey 是 "topology.kubernetes.io/zone"，则每个区（zone）都是该拓扑的一个域。 ● whenUnsatisfiable (string)：当不满足分布约束时，如何调度本 pod。可取值 DoNotSchedule（不调度。这是默认值）、ScheduleAnyway（调度到能降低偏差的候选域上）。 ● labelSelector (LabelSelector)：选择一组 pod。 ● matchLabelKeys ([]string)：指定一组标签键，并在本 pod 标签中查找这些键的值（忽略本 pod 标签中不存在的键），找出来的键值对与 labelSelector 进行逻辑与运算，从而确定一组 pod。要求 matchLabelKeys 和 labelSelector 中禁止存在相同的键，未设置 labelSelector 时无法设置 matchLabelKeys。 ● minDomains (int32)：候选域的最少数量。当候选域个数小于 minDomains 时，Pod 拓扑分布特性会将 "全局最小值" 视为 0。当候选域的数量等于或大于 minDomains 时，本字段的值对调度没有影响。因此，当候选域的数量少于 minDomains 时，调度程序不会将超过 maxSkew 个 Pods 调度到这些域。 如果字段值为 nil，minDomains 被视为 1。字段的有效值为大于 0 的整数。当字段值不为 nil 时，whenUnsatisfiable 必须为 DoNotSchedule。 ● maxSkew (int32)：允许 Pod 分布不均衡的程度——偏差度，即各个候选域中匹配的 Pod 数量允许的最大差值。为了计算各个候选域的差值，必须找出全局最小值（也就是 pod 数量最少的那个候选域），如果候选域的数量小于 minDomains 字段的值，则全局最小值视为零。值为大于 0 的整数，默认是 1。例如在一个包含三个可用区的集群中，maxSkew 设置为 1，具有相同 labelSelector 的 Pod 分布为 zone1(PP)、zone2（PP）、zone3（P），此时全局最小值为 1，新的 pod 只能调度到 zone3，即 zone1(PP)、zone2（PP）、zone3（PP），如果调度到 zone1 或者 zone2，偏差就是 2，这违反了 maxSkew 限制。当 whenUnsatisfiable=ScheduleAnyway 时，此字段被用来给满足此约束的拓扑域更高的优先级。 ● nodeAffinityPolicy (string)：在计算 Pod 拓扑分布偏差时将如何处理 Pod 的 nodeAffinity/nodeSelector，可取值 Honor（只有与 nodeAffinity/nodeSelector 匹配的节点才会包括到计算中。这是默认值）、Ignore（忽略 nodeAffinity/nodeSelector，即所有节点均包括到计算中）。 ● nodeTaintsPolicy (string)：在计算 Pod 拓扑分布偏差时将如何处理节点污点，可取值 Honor（包括不带污点的节点以及本 Pod 具有容忍度且带有污点的节点）、Ignore（无视节点污点，即包括所有节点。这是默认值）。
overhead (map[string]Quantity)	表示与用指定 RuntimeClass 运行 Pod 相关的资源开销。 该字段将由 RuntimeClass 准入控制器在准入时自动填充。
生命周期：	
restartPolicy (string)	Pod 内所有容器的重启策略，可取值 Always、OnFailure 或 Never。默认为 Always。在某些特殊场合，有些值不能用。
terminationGracePeriodSeconds (int64)	表示 Pod 需要体面终止的所需的时长（以秒为单位），即宽限期。值为非负整数。默认为 30 秒。
activeDeadlineSeconds (int64)	尝试将此 Pod 标记为已失败并杀死相关容器之前，Pod 可能在节点上活跃的时长。时长计算基于 startTime 计算（以秒为单位）。取值必须是正整数。
readinessGate ([]PodReadinessGate)	评估所有就绪门控（Readiness Gate）以确定 Pod 就绪状况，当所有容器都已就绪，并且就绪门控中指定的所有条件都为 "true" 时，Pod 被视为就绪。PodReadinessGate 的结构如下： ● conditionType (string)：门控中的条件。
主机名和名称解析：	
hostname (string)	指定 Pod 的主机名。如果此字段未指定，则 Pod 的主机名将设置为系统定义的值。
setHostnameAsFQDN (boolean)	如果为 true，则 Pod 的主机名将配置为 Pod 的 FQDN，而不是叶名称（默认值）。
subdomain (string)	如果设置了此字段，则完全限定的 Pod 主机名将是 <hostname>.<subdomain>.<Pod 名字空间>.svc.<集群域名>。如果未设置此字段，则该 Pod 将没有域名。
hostAliases ([]HostAlias)	是一个可选的列表属性，包含要被注入到 Pod 的 hosts 文件中的主机和 IP 地址。这仅对非 hostNetwork Pod 有效。HostAlias 的结构如下： ● hostnames ([]string)：指定 IP 地址对应的主机名。 ● ip (string)：ip 地址。
dnsConfig (PodDNSConfig)	指定 Pod 的 DNS，这些 DNS 将被合并到基于 dnsPolicy 生成的 DNS 配置中。PodDNSConfig 的结构如下： ● nameservers ([]string)：若干 DNS 服务器的 Ip。 ● options ([]PodDNSConfigOption)：一些 DNS 解析器的参数。PodDNSConfigOption 的结构如下： 　○ name (string)：参数名。 　○ value (string)：参数值。 ● searches ([]string)：用于主机名查找的 DNS 搜索域列表。
dnsPolicy (string)	为 Pod 设置 DNS 策略，有效值为 "ClusterFirstWithHostNet"、"ClusterFirst"（默认值）、"Default" 或 "None"。要针对 hostNetwork 的 Pod 设置 DNS 策略，必须设置为 "ClusterFirstWithHostNet"。
主机名字空间：	
hostNetwork (boolean)	是否使用宿主机的网络名字空间，默认为 false。
hostPID (boolean)	使用宿主机的 PID 名字空间。可选：默认为 false。
hostIPC (boolean)	使用宿主机的 IPC 名字空间。可选：默认为 false。
shareProcessNamespace (boolean)	在 Pod 中的所有容器之间共享单个进程名字空间。设置了此字段之后，容器将能够查看来自同一 Pod 中其他容器的进程并发出信号， 并且每个容器中的第一个进程不会被分配 PID 1。hostPID 和 shareProcessNamespace 不能同时设置。 可选：默认为 false。
账号服务：	
serviceAccountName (string)	运行此 Pod 的服务账号的名称。如果省略，则使用名字空间内默认的服务账号，如 default。
automountServiceAccountToken (boolean)	是否自动挂载服务帐户令牌。默认值为 true。这时要不要挂载由服务账号的同名字段决定。
安全上下文：	
securityContext (PodSecurityContext)	定义 Pod 级别的安全属性和常见的容器级别的安全属性，默认为空。PodSecurityContext 的结构中容器级别的安全属性请参考表 11-1 中的同名字段，pod 级别的安全字段如下： ● supplementalGroups ([]int64)：包含将应用到每个容器中运行的第一个进程的组列表。当 spec.os.name 为 windows 时，不能设置本字段。 ● fsGroup (int64)：应用到 Pod 中所有容器的特殊补充组。某些卷类型允许 kubelet 改变其所有权（为 pod 拥有）：GID 就是 fsGroup，设置 setgid 位（新建文件归 fsGroup 所有），权限位将与 rw-rw---- 进行按位或操作。注意，spec.os.name 为 "windows" 时不能设置此字段。 ● fsGroupChangePolicy (string)：组改变的策略，可取值"OnRootMismatch"（当卷的根目录的权限和所有权与卷的预期权限不匹配时才更改） 和 "Always"（这是默认值）。 ● seccompProfile (SeccompProfile)：定义 Pod 中的容器使用的 seccomp 选项。注意，spec.os.name 为 "windows" 时不能设置此字段。SeccompProfile 的结构如下：

https://www.wenkuoo.org

	○ type（string）：使用哪种 seccomp 配置文件。可取值 localhost（宿主机上的配置文件）、RuntimeDefault（容器运行时默认配置文件）、Unconfined（不使用配置文件）。 ○ localhostProfile（string）：指明宿主机上的配置文件（相对于 kubelet 的 seccom 配置目录而言），本字段必须结合 type=localhost 才行。 ● sysctls（[]Sysctl）：包含用于 Pod 的名字空间的 sysctl 列表。具有不受（容器运行时）支持的 sysctl 的 Pod 可能无法启动。注意，spec.os.name 为 "windows" 时不能设置此字段。Sysctl 的结构如下： ○ name（string）：参数名。 ○ value（string）：参数值。

表 16-6 Volume 的字段

字段（类型）	说明
name（string）	卷的名称。通过容器的 volumeMounts.name 字段等于本字段来引用卷。
暴露的持久卷：	
persistentVolumeClaim（PersistentVolumeClaimVolumeSource）	引用同一名字空间中的 PVC（PersistentVolumeClaim）。PersistentVolumeClaimVolumeSource 的结构如下： ● ClamName（string）：同一名字空间中的 PVC 的名称。 ● readOnly（boolean）：以只读方式挂载卷。默认为 false。
投射卷：	
configMap（ConfigMapVolumeSource）	指定一个 configMap，从这个 configMap 的 data 字段产生一些文件并放在卷中，“键:值”对中键作为文件名（如果省略了下面介绍的 items 字段），值作为文件内容。ConfigMapVolumeSource 的结构如下： ● name（string）：被引用 configMap 资源的名称。 ● optional（boolean）：被引用的 configMap 或其中的键是否需要事先定义。 ● defaultMode（int32）：新建文件的默认权限，取值在 0000-0777（八进制）之间。如果省略本字段，默认为 0644。 ● items（[]KeyToPath）：把键名映射到目录路径（相对路径，且路径中不能出现“..”）。KeyToPath 的结构如下： ○ key（string）：键名 ○ path（string）：相对路径，不能出现“..”。 ○ mode（int32）：定义文件权限，可选字段，取值范围 0000-0777（八进制）。如果省略，则使用 defaultMode 键的值。
secret（SecretVolumeSource）	与 configMap 字段类似，指定一个 Secret，从这个 Secret 的 data 字段产生一些文件并放在卷中。SecretVolumeSource 的结构如下： ● secretName（string）：被引用的处于同一个名字空间内的 Secret 资源的名称。 ● optional（boolean）：被引用的 Secret 或其中的键是否需要事先定义。 ● defaultMode（int32）：新建文件的默认权限，取值在 0000-0777（八进制）之间。如果省略本字段，默认为 0644。 ● items（[]KeyToPath）：参考上面的同名字段的介绍。
downwardAPI（DownwardAPIVolumeSource）	引用一个 DownwordAPI，更具此 DownwardAPI 在卷中新建一些文件。DownwardAPIVolumeSource 的结构体如下： ● defaultMode（int32）：新建文件的默认权限，取值在 0000-0777（八进制）之间。如果省略本字段，默认为 0644。 ● items（[]DownwardAPIVolumeFile）：把键名映射到目录路径（相对路径，且路径中不能出现“..”）。DownwardAPIVolumeFile 的结构如下： ○ path（string）：要创建的文件的相对路径名称。不得使用绝对路径，也不得包含 “..” 路径。必须用 UTF-8 进行编码。 ○ fieldRef（ObjectFieldSelector）：选择 Pod 规范文本中的字段：仅支持注解、标签、名称和名字空间。ObjectFieldSelector 的结构如下： ○ fieldPath（string）：要选择的字段的路径。 ○ apiVersion（string）：pod 资源的版本，默认为 "v1"。 ○ mode（int32）：定义文件权限，可选字段，取值范围 0000-0777（八进制）。如果省略，则使用 defaultMode 键的值。 ○ resourceFieldRef（ResourceFieldSelector）：选择容器的计算资源：目前仅支持资源限制与请求（limits.cpu、limits.memory、requests.cpu 和 requests.memory）。
projected（ProjectedVolumeSource）	供 Secret、ConfigMap 和 Downward API 等资源使用的投射。ProjectedVolumeSource 的结构如下： ● defaultMode（int32）：新建文件的默认权限，取值在 0000-0777（八进制）之间。如果省略本字段，默认为 0644。 ● sources（[]VolumeProjection）：定义系列存储卷的投射。VolumeProjection 的结构如下： ○ configMap（ConfigMapProjection）：将 configMap 投射到卷中。把 configMap 中的 data 字段的内容以文件方式保存在投射卷中，“键”作为文件名（如果省略了 items 字段），“值”作为文件内容。ConfigMapProjection 的结构如下： ○ name（string）：被引用的资源的名称。 ○ optional（boolean）：被引用的 configMap 或其中的键是否需要事先定义。 ○ items（[]KeyToPath）：参考上面的同名字段的介绍。 ○ downwardAPI（DownwardAPIProjection）：参考上面的同名字段的介绍。 ○ secret（SecretProjection）：参考上面的同名字段的介绍。 ○ serviceAccountToken（ServiceAccountTokenProjection）：定义一个投射的服务账号令牌卷。这种投射可用于将服务账号令牌注入到 Pod 运行时文件系统，供访问 API（Kubernetes API Server 或其他）使用。ServiceAccountTokenProjection 的结构如下： ▪ path（string）：令牌投射目标文件相对于挂载点的路径。 ▪ audience（string）： ▪ expirationSeconds（int64）：以秒为单位的服务账号令牌的有效期。至少 10 分钟，默认时 1 个小时。
本地/临时目录：	
emptyDir（EmptyDirVolumeSource）	定义时间跨度与 Pod 生命周期相同的临时空目录。EmptyDirVolumeSource 的结构如下： ● medium（string）：指定临时目录使用存储介质的类型，默认为 ""（使用宿主机的存储介质），取值 Memory 表示内存介质。 ● sizeLimit（Quantity）：定义临时目录的存储上限，默认值位 nil（没有限制）。内存介质的最大使用量为 sizeLimit 与 Pod 中所有容器内存限制总和这两个值之间的最小值。

https://www.verypen.org

hostPath (HostPathVolumeSource)	宿主机上预先存在的文件或目录，它们被挂载到容器中。这种卷通常用于系统代理或允许查看宿主机的其他特权操作。大多数容器不需要这种卷。HostPathVolumeSource 的结构如下： ● path (string)：宿主机上的路径，如果该路径是一个符号链接，则它将沿着链接指向真实路径。 ● type (string)：HostPath 卷的类型，默认为 ""（挂载 hostPath 卷之前不会做类型检查）。其他的类型还有：DirectoryOrCreate（如果由 path 字段定义的路径不存在就创建空目录，权限 0755，uid 和 gid 与 kubelet 进程相同）、Directory（路径必须为事先存在的目录）、FileOrCreate（路径不存在就创建空文件，权限 0644，uid 和 gid 与 kubelet 进程相同。空文件的父目录要求事先存储，否则启动 pod 失败）、File（路径必须为事先存在的文件）、Socket（路径必须为预先存在 UNIX 套接字）、CharDevice（路径必须为事先存在的字符设备文件）、BlockDevice（路径必须为事先存在的块设备文件）。 如： volumes: - name: mydir hostPath: # 确保文件所在目录成功创建。 path: /var/local/aaa type: DirectoryOrCreate - name: myfile hostPath: path: /var/local/aaa/1.txt type: FileOrCreate
持久卷：	
awsElasticBlockStore (AWSElasticBlockStoreVolumeSource)	表示 AWS（亚马逊云） 上的持久化磁盘资源。挂载到一个容器之前 AWS EBS 磁盘必须存在，且该磁盘还必须与 kubelet 位于相同的 AWS 区域中，同时要求 AWS EBS 磁盘只能以“ReadWriteOnce”模式挂载。AWS EBS（亚马逊提供的弹性块存储）的挂载模式有仨：ReadWriteOnce（同一时刻最多只能被一个节点以读/写方式挂载）、ReadOnlyMany（允许被多个节点以只读方式挂载）、ReadWriteMany（允许被多个节点以读/写方式挂载）。AWSElasticBlockStoreVolumeSource 的结构如下： ● volumeID (string)：AWS EBS 中持久磁盘资源的唯一 ID。 ● fsType (string)：文件系统类型，默认为 ext4。要确保宿主机支持指定的文件系统类型，比如 linux 上支持 ext4、xfs、nfs 等。 ● partition (int32)：指定被挂载卷中的分区号。如果省略，则默认为按卷名称进行挂载，例如：对于卷 /dev/sda1， 将分区指定为 1，类似地，/dev/sda 的卷分区为 0。 ● readOnly (boolean)：是否强制为只读方式。默认为 false。
azureDisk (AzureDiskVolumeSource)	指定 Azure 数据盘（微软云提供的块存储）。AzureDiskVolumeSource 的结构如下： ● diskName (string)：数据盘的名称。 ● diskURI (string)：数据盘的 URI。 ● cachingMode (string)：宿主机缓存（Host Caching）模式：None、Read Only、Read Write。 ● fsType (string)：挂载的文件系统类型。必须是主机操作系统所支持的文件系统类型之一。 例如“ext4”、“xfs”、“ntfs”。如果未指定，则隐式推断为 “ext4”。 ● kind (string)：数据盘类型，默认值为 Shared（每个存储帐户多个 Blob 磁盘），其他值还有 Dedicated（每个存储帐户单个 Blob 磁盘）、Managed（由 azure 托管的数据盘）。 ● readOnly (boolean)：强制设置卷挂载中的 readOnly 属性，默认为 false。
azureFile (AzureFileVolumeSource)	Azure 上的文件服务。AzureFileVolumeSource 的结构如下： ● secretName (string)：包含 Azure 存储账号名称和主键的 Secret 的名称。 ● shareName (string)：Azure 共享名称。 ● readOnly (boolean)：强制设置卷挂载中的 readOnly 属性，默认为 false。
cephfs (CephFSVolumeSource)	挂载 Ceph 文件系统。CephFSVolumeSource 的结构如下： ● monitors ([]string)：Ceph 监测器的集合。 ● path (string)：用作挂载的根，而不是挂载完整的 Ceph 树，默认为 “/”。 ● readOnly (boolean)：默认为 false（读/写）。将强制设置卷挂载中的 readOnly 属性。 ● secretFile (string)：指定 User 对应的密钥环的路径，默认为 /etc/ceph/user.secret。 ● secretRef (LocalObjectReference)：引用一个 Secret 资源，次资源用于认证用户身份。 ● user (string)：指定 rados 用户名，默认为 admin。 关于 Ceph 的文件系统存储，请参考相关资料。
cinder (CinderVolumeSource)	挂载 Cinder 卷。CinderVolumeSource 的结构如下： ● volumeID (string)：用于标识 Cinder 中的卷。 ● fsType (string)：文件系统类型，默认为 ext4。要确保宿主机支持指定的文件系统类型，比如 linux 上支持 ext4、xfs、nfs 等。 ● readOnly (boolean)：默认为 false（读/写）。将强制设置卷挂载中的 readOnly 属性。 ● secretRef (LocalObjectReference)：引用 Secret 对象，内含的参数用于连接到 OpenStack。
ephemeral (EphemeralVolumeSource)	与 pod 同寿命的临时卷。EphemeralVolumeSource 的结构如下： ● volumeClaimTemplate (PersistentVolumeClaimTemplate)：将用于创建一个独立的 PVC 来提供临时卷，此 PVC 将随同 pod 一起消亡。PersistentVolumeClaimTemplate 的结构如下： o spec (PersistentVolumeClaimSpec)：定义 PersistentVolumeClaim 的规范代码。整个规范的内容将被原封不动地复制到从此模板创建的 PVC 中。PersistentVolumeClaimSpec 的结构如下： ▪ accessModes ([]string)：指定若干访问模式，卷的访问模式有 ReadWriteOnce、ReadOnlyMany、ReadWriteMany、ReadWriteOncePod（单个 Pod 以读写方式挂载）。 ▪ selector (LabelSelector)：标签选择器。LabelSelector 的结构如下： ▫ matchExpressions ([]LabelSelectorRequirement)：若干标签条件表达式，这些表达是按逻辑“与”计算。LabelSelectorRequirement 的结构如下： ★ key (string)：标签的键名。 ★ operator (string)：条件运算符，有效的运算符包括 In、NotIn、Exists 和 DoesNotExist。 ★ values ([]string)：表示标签的字符串列表。如果运算符为 In 或 NotIn，则 values 数组必须为非空。如果运算符是 Exists 或 DoesNotExist，则 values 数组必须为空。 ▫ matchLabels (map[string]string)：若干 key: value 键值对。一个键值对相当于 matchExpressions 的一个元素，其键字段为 key，运算符为 In，values 数组仅包含 value。所表达的需求最终要按逻辑与的关系组合。 ▪ resources (ResourceRequirements)：指定卷须拥有的最小计算资源。 ResourceRequirements 的结构如下： ▫ claims ([]ResourceClaim)：资源名称。ResourceClaim 只有一个字段 name (string)，其值必须与 pod.spec.resourceClaims 中的一个条目的名称匹配。 ▫ limits (map[string]Quantity)：计算资源的配额。详解请参考容器资源的同名字段。 ▫ requests (map[string]Quantity)：最小计算资源量。详解请参考容器资源的同名字段。 ▪ volumeName (string)：引用本 Claim 所对应的 PersistentVolume。

https://www.liwenwei.com

	▪ storageClassName (string)：本 Claim 所要求的 StorageClass 名称。 ▪ volumeMode (string)：指定本 Claim 要求的卷类型。默认是 Filesystem。 o metadata (ObjectMeta)：允许包含一些标签和注解，在创建 PVC 时，这些数据会被复制到 PVC 中。ObjectMeta 的结构介绍请参考表 11-3。
fc (FCVolumeSource)	表示挂载到宿主机（运行了 kubelet 的主机）上随后暴露给 Pod 的一个 Fibre Channel 资源。FC 的挂载模式只有 ReadWriteOnce。FCVolumeSource 的结构如下： ● fsType (string)：参考上面介绍的 Cinder 卷的同步字段。 ● lun (int32)：FC 目标 lun 编号。 ● readOnly (boolean)：参考上面介绍的 Cinder 卷的同步字段 ● targetWWNs ([]string)：FC 目标全球名称（WWN）。 ● wwids ([]string)：FC 卷全球识别号（wwids）。必须设置 wwids 或 targetWWNs 及 lun 的组合，但不能同时设置两者。
flexVolume (FlexVolumeSource)	表示使用基于 exec 的插件实现的通用卷资源，非常灵活，类似 CNI。FlexVolumeSource 的结构如下： ● driver (string)：卷用到的驱动名称。 ● fsType (string)：文件系统类型，必须是宿主机操作系统所支持的文件系统类型之一。默认的文件系统取决于 flexVolume 脚本。 ● options (map[string]string)：定义一些额外的命令选项（如果有）。 ● readOnly (boolean)：请参考上面的同名字段。 ● secretRef (LocalObjectReference)：对包含敏感信息的 Secret 对象的引用，该 Secret 会被传递到插件脚本。如果未指定 Secret 对象，则此字段可以为空。如果 Secret 对象包含多个 Secret，则所有 Secret 被传递到插件脚本。
flocker (FlockerVolumeSource)	定义 Flocker 卷。FlockerVolumeSource 的结构如下： ● datasetName (string)：数据集的名称，此数据集错位元数据被存储。 ● datasetUUID (string)：数据集的 UUID。 注意：datasetName 与 datasetUUID 是互斥的，只能设置一个。
gcePersistentDisk (GCEPersistentDiskVolumeSource)	表示挂载到宿主机上随后暴露给 Pod 的一个 GCE Disk 资源（谷歌云存储）。BCE 磁盘只支持 ReadWriteOnce 或 ReadOnlyMany 挂载模式。GCEPersistentDiskVolumeSource 的结构如下： ● pdName (string)：标识 GCE 中的磁盘的唯一名称。 ● fsType (string)：请参考上面的同名字段。 ● partition (int32)：请参考上面的同名字段。 ● readOnly (boolean)：请参考上面的同名字段。
glusterfs （GlusterfsVolumeSource）	关联到宿主机并暴露给 Pod 的 Glusterfs 卷。GlusterFS 是开源网络文件系统（https://www.gluster.org/）。GlusterfsVolumeSource 的结构如下： ● endpoints (string)：端点的名称，它详细给出了 Glusterfs 拓扑结构。 ● path (string)：Glusterfs 卷的路径。 ● readOnly (boolean)：请参考上面的同名字段。
iscsi (ISCSIVolumeSource)	挂载到宿主机上随后暴露给 Pod 的一个 iSCSI Disk 资源。只支持 ReadWriteOne 挂载模式。ISCSIVolumeSource 的结构如下： ● iqn (string)：iqn 为 iSCSI Qualified Name 的缩写，也就是 iSCSI 目标的限定名称。 ● lun (int32)：iSCSI 目标逻辑单元号。 ● targetPortal (string)：是 iSCSI 目标门户。如果不是默认端口（通常是 TCP 端口 860 和 3260），则 Portal 为 IP 或 ip_addr:port。 ● chapAuthDiscovery (boolean)：是否支持 iSCSI Discovery CHAP 身份认证。 ● fsType (string)：请参考上面的同名字段。 ● initiatorName (string)：是自定义的 iSCSI 发起程序（Initiator）名称。 ● iscsiInterface (string)：使用 iSCSI 的接口名称，默认为 default(tcp)。 ● portals ([]string)：是 iSCSI 目标门户列表（iSCSI Target Portal List）。如果不是默认端口（通常是 TCP 端口 860 和 3260），则 Portal 为 IP 或 ip_addr:port。 ● readOnly (boolean)：请参考上面的同名字段。 ● secretRef (LocalObjectReference)：是 iSCSI 目标（Target）和发起程序(Initiator)身份认证所用的 CHAP Secret。
nfs (NFSVolumeSource)	表示在主机上挂载的 NFS，其生命周期与 Pod 相同。NFSVolumeSource 的结构如下： ● path (string)：由 NFS 服务器导出的路径。 ● server (string)：NFS 服务器的主机名或 IP 地址。 ● readOnly (boolean)：使用只读权限挂载 NFS，默认为 false。
photonPersistentDisk (PhotonPersistentDiskVolumeSource)	表示宿主机上挂载的 PhotonController 持久磁盘。PhotonPersistentDiskVolumeSource 的结构如下： ● pdID (string)：Photon Controller 持久磁盘的 ID。 ● fsType (string)：请参考上面的同名字段。
portworxVolume (PortworxVolumeSource)	表示宿主机上挂载的 portworx 卷。PortworxVolumeSource 的结构如下： ● volumeID (string)：唯一标识 Portworx 卷的 ID。 ● fsType (string)：请参考上面的同名字段。 ● readOnly (boolean)：请参考上面的同名字段。
quobyte (QuobyteVolumeSource)	表示宿主机上挂载的 Quobyte 卷，寿命同 pod。QuobyteVolumeSource 的结构如下： ● registry (string)：代表一个或多个 Quobyte Registry 服务，格式为 host:port[,host:port,...]。 ● volume (string)：一个已经创建的 Quobyte 卷的名称。 ● group (string)：指定将卷访问映射到的组，默认是不映射到组。 ● readOnly (boolean)：请参考上面的同名字段。 ● tenant (string)：值由插件设置。 ● user (string)：将卷访问映射到的用户，默认为 serivceaccount 用户。
rbd (RBDVolumeSource)	表示宿主机上挂载的 RDB 设备（Rados Block Device），也就是 Ceph 的块设备，寿命同 pod。RBDVolumeSource 的结构如下： ● image (string)：为 rados 镜像名称。 ● monitors ([]string)：是 Ceph 监测的集合。 ● fsType (string)：请参考上面的同名字段。 ● keyring (string)：是 RBDUser 密钥环的路径。默认为 /etc/ceph/keyring。 ● pool (string)：是 rados 池名称。 ● readOnly (boolean)：请参考上面的同名字段。 ● secretRef (LocalObjectReference)：指定一个 Secret 的名称，此 Secret 用于 RBDUser 的身份认证。 ● user (string)：是 rados 用户名。默认为 admin。 关于这参数的进一步了解，请参考 Ceph 的块设备。
scaleIO (ScaleIOVolumeSource)	表示宿主机上挂载的 ScaleIO 持久卷。ScaleIO 由 EMC 公司开发的软件定义存储产品。ScaleIOVolumeSource 的结构如下： ● gateway (string)：是 ScaleIO API 网关的主机地址。 ● secretRef (LocalObjectReference)：引用包含 ScaleIO 用户等敏感信息的 Secret。 ● system (string)：是存储系统的名称，与 ScaleIO 中的配置相同。 ● .fsType (string)：要挂载的文件系统类型。必须是主机操作系统所支持的文件系统类型之一。例如“ext4”、“xfs”、“ntfs”。默认为 “xfs”。

https://www.yongen.org

	● protectionDomain (string)：是 ScaleIO 保护域（ScaleIO Protection Domain）的名称。 ● readOnly (boolean)：请参考上面的同名字段。 ● sslEnabled (boolean)：是否启用与网关的 SSL 通信，默认为 false。 ● storageMode (string)：指示卷所用的存储应是 ThickProvisioned 还是 ThinProvisioned。默认为 ThinProvisioned。 ● storagePool (string)：与保护域关联的 ScaleIO Storage Pool。 ● volumeName (string)：在 ScaleIO 系统中已经创建的一个卷的名称。
storageos (StorageOSVolumeSource)	表示宿主机上挂载的 StorageOS 卷。StorageOSVolumeSource 的结构如下： ● fsType (string)：要挂载的文件系统类型。必须是主机操作系统所支持的文件系统类型之一。例如“ext4”、“xfs”、“ntfs”。默认为 “ext4”。 ● readOnly (boolean)：请参考上面的同名字段。 ● secretRef (LocalObjectReference)：指定用于获取 StorageOS API 凭据的 Secret。如果未指定，则将尝试使用默认值。 ● volumeName (string)：StorageOS 卷名称，卷名称在同一个名字空间内是必须唯一。 ● volumeNamespace (string)：在 StorageOS 内定义卷的作用域。
vsphereVolume (VsphereVirtualDiskVolumeSource)	表示宿主机上挂载的 vSphere 卷。VsphereVirtualDiskVolumeSource 的结构如下： ● volumePath (string)：标识 vSphere 卷 vmdk 的路径。 ● fsType (string)：请参考上面的同名字段。 ● storagePolicyID (string)：与 StoragePolicyName 字段关联的基于存储策略管理（SPBM）的配置文件 ID。 ● toragePolicyName (string)：基于存储策略管理（SPBM）的配置文件名称。

表 16-7 PodTemplate 资源对象的字段

字段（类型）	说明
apiVersion (string)	值的格式为[GROUP/]VERSION。对于 PodTemplate，目前的值固定为 v1
kind (string)	对于 PodTemplate，值固定为 PodTemplate
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
template (PodTemplateSpec)	定义将基于此 Pod 模板所创建的 Pod。PodTemplateSpec 结构的介绍请参考表 16-8。

表 16-8 PodTemplateSpec 资源对象的字段

字段（类型）	说明
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
spec (PodSpec)	定义 pod，具体参考表 11-5

表 16-9 PodTemplateList 资源对象的字段

字段（类型）	说明
apiVersion	值固定为 v1
kind	值固定为 PodTemplateList
metadata (ListMeta)	标准的列表元数据
items ([]PodTemplate)	PodTemplate 对象列表。PodTemplate 结构介绍请参考表 11-8

https://www.veryopen.org

3. ReplicaSet

ReplicaSet 确保在任何时刻都在运行指定数量的 Pod 副本，此资源介绍请参考表 16-9。

表 16-9 ReplicaSet 资源对象的字段

字段（类型）	说明
apiVersion	值固定为 apps/v1

kind	值固定为 ReplicaSet
metadata (ObjectMeta)	标准的对象元数据
spec (ReplicaSetSpec)	定义 ReplicaSet 预期行为。ReplicaSetSpec 的结构如下： ● selector (LabelSelector)：确定一组 pod 的标签选择算符。 ● template (PodTemplateSpec)：创建 pod 对象的模板，关于 PodTemplateSpec 请参考表 11-6。 ● replicas (int32)：Pod 副本的数量。 ● minReadySeconds (int32)：新建的 Pod 在没有任何容器崩溃的情况下就绪并被系统视为可用的最短秒数。 默认为 0（Pod 就绪后即被视为可用）。
status (ReplicaSetStatus)	ReplicaSet 状态

4. Deployment

Deployment 意为部署，是针对应用而言，即一个 Deployment 就是部署一个应用，显然包含若干 Pod，而 Pod 又包含若干容器。表 16-10 列出了 Deployment 的全部字段及说明。

表 16-10 Deployment 资源对象的字段

字段（类型）	说明
apiVersion (string)	值的格式为[GROUP/]VERSION。对于 Deployment，目前的值固定为 apps/v1
kind (string)	对于 Deployment，值固定为 Deployment
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
spec (DeploymentSpec)	定义 Deployment 的行为。关于 DeploymentSpec 对象请参考表 16-11
status (DeploymentStatus)	Deployment 的当前状态。状态信息只能读。

表 16-11 DeploymentSpec 资源对象的字段

字段（类型）	说明
selector (LabelSelector)	用来确定一组 Pod 的标签选择算符。
template (PodTemplateSpec)	描述将要创建的 Pod。template.spec.restartPolicy 唯一被允许的值是 Always。PodTemplateSpec 结构请参考表 11-7
replicas (int32)	预期 Pod 的数量，默认为 1。同一个 pod 运行多个副本能提高可用性。
minReadySeconds (int32)	pod 就绪的最少秒数。表明 Pod 内的容器都已正常运行。但有时因业务逻辑的需要，必须延长某些 pod 的就绪时间，尽管实际上 pod 已经就绪，但是集群还是要等到之后的某个时间点才反馈就绪状态。
strategy (DeploymentStrategy)	将现有 Pod 替换为新 Pod 时所用的部署策略。DeploymentStrategy 的结构如下： ● type (string)：替换模式，可取值 Recreate 或 RollingUpdate，默认为 RollingUpdate。 ● rollingUpdate (RollingUpdateDeployment)：滚动更新这些配置参数。仅当 type = RollingUpdate 时才需要本字段。RollingUpdateDeployment 的结构如下： o maxSurge (IntOrString)：超出预期的 Pod 数量之后可以调度的最大 Pod 数量。该值可以是一个绝对数（例如：5）或一个预期 Pod 的百分比（例如：10%）。如果 MaxUnavailable 为 0，则此字段不能为 0。通过向上取整计算得出一个百分比绝对数。默认为 25%。例如：当此值设为 30% 时，如果滚动更新启动，则可以立即对 ReplicaSet 扩容，从而使得新旧 Pod 总数不超过预期 Pod 数量的 130%。一旦旧 Pod 被杀死，则可以再次对新的 ReplicaSet 扩容，确保更新期间任何时间运行的 Pod 总数最多为预期 Pod 数量的 130%。IntOrString 是可以保存 int32 或字符串的一个类型。 o maxUnavailable (IntOrString)：更新期间允许不可用的最大 Pod 数量。该值可以是一个绝对数（例如：5）或一个预期 Pod 的百分比（例如：10%）。通过向下取整计算得出一个百分比绝对数。 如果 MaxSurge 为 0，则此字段不能为 0。默认为 25%。 例如：当此字段设为 30%，则在滚动更新启动时 ReplicaSet 可以立即缩容为预期 Pod 数量的 70%。 一旦新的 Pod 就绪，ReplicaSet 可以再次缩容，接下来对新的 ReplicaSet 扩容， 确保更新期间任何时间可用的 Pod 总数至少是预期 Pod 数量的 70%。
revisionHistoryLimit (int32)	保留允许回滚的旧 ReplicaSet 的数量，默认为 10
progressDeadlineSeconds (int32)	一个 Deployment 在被视为失败之前继续推进的最长时间（以秒为单位）。在超时之前，deployment 控制器会继续处理失败的 Deployment，最终 Deployment 的状态中将显示部署超时的原因。请注意，在部署暂停期间不会计时。 默认为 600 秒。
status (DeploymentStatus)	Deployment 的状态。关于 DeploymentStatus 请参考 https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/deployment-v1/

https://www.verypen.org

5. StatefulSet

StatefulSet 意为有状态的集合，适用于有状态的应用，比如数据库服务，它本质上为 Deployment 的变体。StatefulSet 管理的若干 pod 有固定的名称、固定的存储和固定的启停顺序等，对应无头服务（不分配集群 ip 因而没有负载均衡功能），DNS 直接解析到后端 pod 的 ip。StatefulSet 对象的字段介绍请参考表 16-12。

表 16-12 StatefulSet 资源对象的字段

字段（类型）	说明
apiVersion（string）	目前的值固定为 apps/v1
kind（string）	值固定为 StatefulSet
metadata（ObjectMeta）	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
spec（StatefulSetSpec）	定义 StatefulSet 的行为。关于 StatefulSetSpec 对象请参考表 16-13
status（StatefulSetStatus）	StatefulSet 的当前状态。

表 16-13 StatefulSetSpec 资源对象的字段

字段（类型）	说明
serviceName（string）	服务的名称
selector（LabelSelector）	通过标签选择算符选择一组 pod。选择算符必须与 Pod 模板中的 label 匹配。
template（PodTemplateSpec）	描述 Pod 的对象，检测到副本不足时将创建新的 Pod，经由 StatefulSet 创建的每个 Pod 都将满足这个模板，但与 StatefulSet 的其余 Pod 相比，每个 Pod 具有唯一的标识。每个 Pod 将以<statefulsetname>-<podindex>格式命名，podindex 从 ordinals 字段定义的值开始，例如，名为"web"且索引号为"3"的 StatefulSet 中的 Pod 将被命名为"web-3"。 template.spec.restartPolicy 唯一被允许的值是 Always。PodTemplateSpec 的介绍请参考表 11-7。
replicas（int32）	Pod 的副本数量，默认为 1。
updateStrategy （StatefulSetUpdateStrategy）	如果修改了 template 字段，那么用何种策略更新 StatefulSet 中的 Pod。 StatefulSetUpdateStrategy 的结构如下： ● type（string）：策略类型，可取值 RollingUpdate（自动滚动更新。这是默认值）、OnDelete（手动删除 pod，然后会自动新建 pod）。“自动滚动更新”即自动地删除并重建每一个 Pod，处理顺序是：①从序号最大的 Pod 开始，逐个删除和更新每一个 Pod，直到序号最小的 Pod 为止，②当正在更新的 Pod 达到就绪状态之后，才继续更新前序 Pod。 ● rollingUpdate（RollingUpdateStatefulSetStrategy）：当 type 字段为 RollingUpdate 时，使用本字段传递参数。RollingUpdateStatefulSetStrategy 的结构如下： ○ maxUnavailable（IntOrString）：更新期间不可用的 Pod 个数上限。取值可以是绝对数量（例如：5）或所需 Pod 的百分比（例如：10%），不能为 0，默认为 1。总的可用数量由 replicas 字段定义。 ○ partition（int32）：表示 StatefulSet 应该被分区进行更新时的序号。在滚动更新期间，序号在[partition, replicas）内的所有 Pod 都会被更新，而序号在[0, partition)内的所有 Pod 保持不变（即使手工删除了 Pod，k8s 也会使用前一个版本的 template 新建 pod）。显然，如果本字段的值大于等于 replicas，那么所有 pod 都不会更新，默认值为 0（所有 pod 都会更新）。本字段有助于进行金丝雀部署。
podManagementPolicy（string）	pod 管理策略——如何增/减 pod 数量。可取值 OrderedReady（扩容时按升序增加 pod，且每个 Pod 就绪后再创建下一个 pod；收缩时按降序删除多余的 pod）、Parallel（新增或删除 pod 时并行进行，不用等待 Pod 就绪）。
revisionHistoryLimit（int32）	修订版本个数，默认是 10。
volumeClaimTemplates （[]PersistentVolumeClaim）	允许 Pod 使用的 PVC 列表。此列表中的每个 PVC 至少必须在模板的某个容器中存在匹配的（按 name 匹配）volumeMount。此列表中的 PVC 优先于模板中具有相同名称的所有卷。
minReadySeconds（int32）	新建 pod 从就绪到可用的最少秒数，默认为 0（Pod 准备就绪就被视为可用）。
persistentVolumeClaimRetentionPolicy （StatefulSetPersistentVolumeClaimRetentionPolicy）	指定新建的 PVC 的生命期限。默认情况下，所有 PVC 都根据需要创建并被保留到手动删除。此属性需要启用 StatefulSetAutoDeletePVC 特性门控。 StatefulSetPersistentVolumeClaimRetentionPolicy 的结构如下： ● whenDeleted（string）：当 StatefulSet 被删除时如何对待 PVC，可取值 Retain（保留。这是默认值）、Delete（删除）。 ● whenScaled（string）：当 StatefulSet 被收缩时，如何处理 PVC。可取值 Retain（保留。这是默认值）、Delete（被删除的 pod 所关联的 PVC 也要删除）。
ordinals（StatefulSetOrdinals）	定义 StatefulSet 中的 pod 的起始索引编号。StatefulSetOrdinals 的结构如下： ● start（int32）：第一个 pod 副本的索引号，默认为 0。如果设置了本字段，副本索引范围为[ordials.start, ordinals.start + replicas)。如果不设置，副本索引范围为 [0, replicas)。

https://www.yuboen.org

二、服务类资源

1. Service

Service 是软件服务（例如 mysql）的命名抽象，向外暴露一组 pod 提供的网络服务。Service 包含代理要侦听的本地端口（例如 3306）和一个选择算符，选择算符用来确定哪些 Pod 将响应通过代理发送的请求。

表 16-14 Service 资源对象的字段

字段（类型）	说明
apiVersion (string)	目前的值固定为 v1
kind (string)	值固定为 Service
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
spec (ServiceSpec)	定义 Service 的行为。ServiceSpec 对象的结构如下： ● type (string)：指定 Service 的暴露类型，可取值 ExternalName、ClusterIP（默认值）、NodePort 或 LoadBalancer。 “ClusterIP” 为端点分配一个集群内部 IP 地址用于负载均衡到 endpoints，这些 endpoints 由 selector 字段确定，如果未设置 selector，则需要手动构造 Endpoints 或 EndpointSlice 的资源对象来确定。如果 clusterIP 字段为“None”，则不分配虚拟 IP，并且这些 endpoints 作为一组端点而不是虚拟 IP 直接对外发布。 “NodePort” 建立在 ClusterIP 之上，并在每个节点上分配一个端口，该端口路由到与 clusterIP 相同的 Endpoints。 “LoadBalancer” 基于 NodePort 构建并创建一个外部负载均衡器（如果当前云支持），该负载均衡器路由到与 clusterIP 相同的 Endpoints。 “ExternalName” 将此 Service 取一个别名 ExternalName。其他几个字段不适用于 ExternalName 类型的 Service。 ● selector (map[string]string)：选择一组 pod，然后把服务流量路由到这些 Pod。类型为 ExternalName 的服务，则忽略本字段。如果使用多个标签，可这样： selector: matchLabels: env: dev app: nginx ● ports ([]ServicePort)：若干暴露的端口。ServicePort 的结构如下： o port (int32)：对外暴露的端口。 o targetPort (IntOrString)：pod 上的端口或名称。如果省略本字段，Pod 端口就是 port 字段的值。端口必须在 1 到 65535 的范围内。名称必须符合 IANA_SVC_NAME。对于 clusterIP 为 None 的服务，此字段将被忽略。 o protocol (string)：网络协议。支持 TCP、UDP 和 SCTP，默认为 TCP。 o name (string)：Service 中此端口的名称。 o nodePort (int32)：当类型为 NodePort 或 LoadBalancer 时，Service 公开在宿主机上的端口， 通常由系统分配。 o appProtocol (string)：此端口的应用协议，遵循标准的 Kubernetes 标签语法。 ● ipFamilies ([]string)：定义本服务的 Ip 协议栈，可取值 IPV4 和 IPV6，本字段一般会自动设置。本字段仅适用于 ClusterIP、NodePort 和 LoadBalancer 类型的服务。 ● ipFamilyPolicy (string)：使用双栈还是单栈，可取值 SingleStack（单栈，即要么 IPV4 要么 IPV6）、PreferDualStack（双栈集群上的两个 IP 协议或单栈集群上的单个 IP 协议）、RequireDualStack（同时启用 IPV4 和 IPV6 地址。要求集群是双栈配置）。ipFamilies 和 clusterIPs 字段取决于此字段的值。 更新服务设置类型为 ExternalName 时，此字段将被擦除。 ● clusterIP (string)：指定服务的 IP 地址，通常是随机分配的。如果手动指定，必须在范围内（根据系统配置）且没有被占用，此 ip 将被分配给服务，否则创建服务失败。本字段可取值 None、“”或有效 ip。本字段不支持 ExternalName 类型的服务，如果创建 Serivce 是指定了本字段，则创建失败，对已有服务更新类型为 ExternalName 时，本字段会被移除。 ● clusterIPs ([]string)：与 clusterIP 做类似理解，不过可定义至多两个 ip——单栈一个 Ip，双栈两个 ip（要与 ipFamilies 的值相对应）。如果没省略本字段，那么 clusterIPs[0] 要等于 clusterIP。 ● externalIPs ([]string)：一些外部的 ip，服务后端的工作节点会接受来自这些 ip 的流量，例如集群外部的负载均衡器。 ● sessionAffinity (string)：启用基于客户端 IP 的会话亲和性，可取值 ClientIP 或 None，默认为 None。 ● loadBalancerClass (string)：针对类型为 LoadBalancer 的 Service，指定本 Service 所属的负载均衡器实现的类。如果未设置此字段，则使用默认负载均衡器实现，默认负载均衡器现在通常通过云提供商集成完成。 ● externalName (string)：定义一个名字并作为本服务的别名由服务发现机制返回，名字中的字母必须小写。本字段只针对 ExternalName 型服务有效。 ● externalTrafficPolicy (string)：来自集群外部的流量如何分发给内部工作节点。外部流量定义为：数据包的源 ip 为集群外部的 ip 而目的 Ip 为 Service 的 NodePort、ExternalIP 或 LoadBalancerIP。可取值 Local（每个工作节点直接把流量传到本地 endpoint）、Cluster（均衡路由到所有 endpoint。这是默认值）。 ● internalTrafficPolicy (string)：来自集群内部流量的分发策略。内部流量定义为：数据包的目的 Ip 为 Service 的 clusterIP。取值为 Local 或 Cluster，意义同上。 ● healthCheckNodePort (int32)：指定 Service 的健康检查节点端口。仅适用于 type 为 LoadBalancer 且 externalTrafficPolicy 设置为 Local 的情况。 ● publishNotReadyAddresses (boolean)：是否忽略提供了 endpoint 的 pod 的就绪/未就绪状态，默认值 false。如果设置为 true，那么未就绪的 pod 也能被 DNS 解析。 ● sessionAffinityConfig (SessionAffinityConfig)：设置会话亲和性。SessionAffinityConfig 的结构如下： o clientIP (ClientIPConfig)：基于客户端 IP 的会话亲和性。ClientIPConfig 的结构如下：

https://www.veryyoen.org

	▪ timeoutSeconds（int32）：指定 ClientIP 类型会话的维系时间秒数。如果 ServiceAffinity 字段的值为"ClientIP"，则本字段的值必须 >0 && <=86400（1 天）。默认值为 10800（3 小时）。 ● allocateLoadBalancerNodePorts（boolean）：针对 LoadBalancer 类型的 Service，是否为 Service 分配 NodePort。本字段默认为 true。如果集群负载均衡器不依赖 NodePort，则可设置此字段为 false。如果调用者（通过指定一个值）请求特定的 NodePort，则无论此字段如何，Service 都会接受这些请求。
status（ServiceStatus）	Service 的当前状态。状态信息只能读。具体请参考 https://kubernetes.io/docs/reference/kubernetes-api/service-resources/service-v1

Endpoints 、 EndpointSlice 、 Ingress 、 IngressClass 服务类资源请参考 https://kubernetes.io/docs/reference/kubernetes-api/service-resources/。

三、配置类资源

1. ConfigMap

ConfigMap 包含供 Pod 使用的配置数据。

表 16-15 ConfigMap 资源对象的字段

字段（类型）	说明
apiVersion（string）	目前的值固定为 v1
kind（string）	值固定为 ConfigMap
metadata（ObjectMeta）	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
binaryData（map[string][]byte）	包含二进制数据。字段的值可以包含不在 UTF-8 范围中的字节序列，键不得与 data 字段中的键重叠。
data（map[string]string）	包含配置数据，如果值包含非 UTF-8 字节序列，则必须使用 binaryData 字段。
immutable（boolean）	是否可以修改 ConfigMap 中存储的数据（dat 和 binaryData 字段），true 不能修改，其他值允许修改，默认值 nil。

ConfigMapList 是包含 ConfigMap 对象列表的资源，其字段介绍参考表 16-16。

表 16-16 ConfigMapList 资源对象的字段

字段（类型）	说明
apiVersion（string）	目前的值固定为 v1
kind（string）	值固定为 ConfigMapList
metadata（ListMeta）	标准的列表元数据
items（[]ConfigMap）	ConfigMap 的列表

2. Secret

Secret 对象包含一些敏感数据，其字段说明版靠表 16-17。

表 16-17 Secret 资源对象的字段

字段（类型）	说明

https://www.veryopen.org

apiVersion (string)	目前的值固定为 v1
kind (string)	值固定为 Secret
metadata (ObjectMeta)	标准的对象元数据。
data (map[string][]byte)	若干敏感数据
immutable (boolean)	是否可以修改 Secret 中存储的数据（data 字段），true 不能修改，其他值允许修改，默认值 nil。
stringData (map[string]string)	指定字符串格式的非二进制敏感数据。 为了方便起见，它作为只写输入字段提供，写入时会将所有键和值合并到 data 字段，且覆盖任何现有的值。从 API 读取时绝不会输出 stringData 字段。
type (string)	敏感数据类型。支持的类型有： ✧ Opaque -用户定义的任意数据，这是默认类型。 ✧ kubernetes.io/service-account-token -服务账号令牌 ✧ kubernetes.io/dockercfg - ~/.dockercfg 文件的序列化形式 ✧ kubernetes.io/dockerconfigjson - ~/.docker/config.json 文件的序列化形式 ✧ kubernetes.io/basic-auth - 用于基本身份认证的凭据。 ✧ kubernetes.io/ssh-auth - 用于 SSH 身份认证的凭据. ✧ kubernetes.io/tls - 用于 TLS 客户端或者服务器端的数据。 ✧ bootstrap.kubernetes.io/token - 启动引导令牌数据。

SecretList 是 Secret 的列表。

表 16-18 SecretList 资源对象的字段

字段（类型）	说明
apiVersion (string)	目前的值固定为 v1
kind (string)	值固定为 SecretList
metadata (ListMeta)	标准的列表元数据
items ([]Secret)	Secret 的列表

§16.3 RBAC 机制

§16.3.1 基础

k8s 的 rbac 机制参考图 16-3 所示。

https://www.veryopen.org

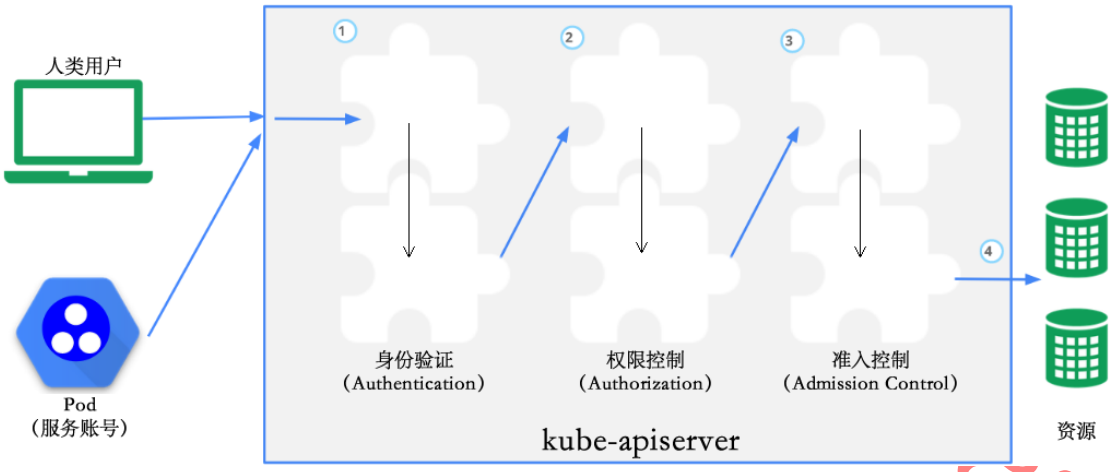


图 16-3 k8s 的 rbac 机制

“身份验证”就是验证身份，确定能否进入系统，“权限控制”就是确定能否对资源执行操作，“准入控制”由各种安全插件实现的进一步检查机制。

RBAC 就是基于角色的访问控制，即在某些资源上赋予谁什么角色，而角色就是权限的集合。Kubernetes 采用 Role、ClusterRole、RoleBinding 和 ClusterRoleBinding 资源对象实现 RBAC，Role 和 ClusterRole 代表一组相关权限的集合，Role 是局限在名字空间内的，而 ClusterRole 是全局性的。如果你希望在名字空间内定义角色，应该使用 Role，如果你希望定义集群范围的角色，请使用 ClusterRole，不过一个用户允许同时被赋予 Role 和 ClusterRole 类型的角色。RoleBinding 和 ClusterRoleBinding 用来给用户赋权，即把角色和用户绑定，RoleBinding 只在名字空间内授权，ClusterRoleBinding 在集群范围内授权。K8s 中的用户被称为主体（Subject），主体可以是组、用户或者服务账号（Service Account）。

只有服务账号由 kubernetes 管理，用户和组由外部管理——创建、编辑、删除等由第三方系统管理，而 k8s 负责权限控制。用户名可以是普通的用户名，像 "alice"，或者是邮件风格的用户名，如 "bob@example.com"，或者是以字符串形式表达的数字 ID，但是 system:不能出现用户名或组名的最前面，因为 system:是系统保留的，比如 system:john 就是不合法的用户名。

每个名字空间内至少有一个服务账号，默认为 default。如果创建 Pod 时没有具体指明服务账号，那么会自动分配 default，可用命令“kubectl get pods/<podname> -o yaml | serviceAccountName”查看 pod 的服务账号。pod 内部的应用使用自动挂载的服务账号凭据（比如令牌）来访问 Kubernetes API。服务账号被身份认证后，用户名的全称为 system:serviceaccount:<名字空间>:<服务账号>，并被分配到用户组 system:serviceaccounts 和 system:serviceaccounts:<名字空间>。

因为用户和组不由 kubernetes 管理，所以在申请证书时，用 CN 指定用户名，O 指定用户归属的若干组，如 /CN=alice/O=gdpi/O=system:master 表示用户 alice 加入了两个组 gdpi 和 system:masters。具体的例子请参考后面的实战二。

Role 资源对象的字段见表 16-19。

表 16-19 Role 资源对象的字段

字段（类型）	说明
apiVersion (string)	目前的值固定为 rbac.authorization.k8s.io/v1
kind (string)	值固定为 Role
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
rules ([]PolicyRule)	若干权限，即对 apiGroups 里的 resources 进行 verbs 操作。PolicyRule 的结构如下： ● apiGroups ([]string)：若干资源组（参见表 11-1）。“” 表示核心 API 组，“*” 表示所有资源组。

https://www.verypen.org

	● resources（[]string）：资源列表。“*” 表示所有资源。如 resources:［"persistentvolumes"］。资源名称请参考§16.2 小节。 ● verbs（[]string）：允许执行的操作。“*”表示资源允许的所有操作。如 verbs:［"get", "list", "watch", "create", "update", "patch"］。关于“操作”请参考§16.2 小节。 ● resourceNames（[]string）：资源白名单，只能“操作”白名当中的资源。默认是没有白名单。 ● nonResourceURLs（[]string）：若干非资源端点，“*”可允许出现在 URL 的末尾。如： rules: - nonResourceURLs: ["/healthz", "/healthz/*"] verbs: ["get", "post"] 表示可对非资源端点 /healthz 和其子路径上发起 GET 和 POST 请求

ClusterRole 资源对象的字段见表 16-20。

表 16-20 ClusterRole 资源对象的字段

字段（类型）	说明
apiVersion（string）	目前的值固定为 rbac.authorization.k8s.io/v1
kind（string）	值固定为 ClusterRole
metadata（ObjectMeta）	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4。里面的字段 namespace 没有意义，即使设置了也被忽略。
aggregationRule（AggregationRule）	如何定位并聚合其它 ClusterRole 到本 ClusterRole 中。AggregationRule 的结构如下： ● clusterRoleSelectors（[]LabelSelector）：通过标签选择算符查找其他 ClusterRole，并把匹配成功的 ClusterRole 中的规则添加到本 ClusterRole 中。关于 LabelSelector 请参考表 16-6 中的 ephemeral 字段。
rules（[]PolicyRule）	参考表 16-19 中的同名字段。

RoleBinding 此象的字段见表 16-21。

表 16-21 RoleBinding 资源对象的字段

字段（类型）	说明
apiVersion（string）	目前的值固定为 rbac.authorization.k8s.io/v1
kind（string）	值固定为 RoleBinding
metadata（ObjectMeta）	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
roleRef（RoleRef）	引用当前命名空间中的 Role 或全局命名空间中的 ClusterRole。RoleRef 的结构如下： ● apiGroup（string）：被引用角色对象归属的组。 ● kind（string）：被引用角色对象的类型。 ● name（string）：被引用角色对象的名字。
subjects（[]Subject）	若干主体（用户、组或服务账号）。Subject 的结构如下： ● apiGroup（string）：被引用主体对象归属的组。对于 ServiceAccount 主体默认为 ""，对于 User 和 Group 主体，默认为 "rbac.authorization.k8s.io"。 ● kind（string）：被引用主体对象的类型。可取值 User、Group 和 ServiceAccount。 ● name（string）：被引用主体对象的名字。 ● namespace（string）：被引用主体对象所在的名字空间。

ClusterRoleBinding 对象的字段见表 16-22。

表 16-22 ClusterRoleBinding 资源对象的字段

字段（类型）	说明
apiVersion（string）	目前的值固定为 rbac.authorization.k8s.io/v1
kind（string）	值固定为 ClusterRoleBinding
metadata（ObjectMeta）	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4。
roleRef（RoleRef）	引用全局命名空间中的 ClusterRole。RoleRef 的结构如下： ● apiGroup（string）：被引用角色对象归属的组。 ● kind（string）：被引用角色对象的类型。 ● name（string）：被引用角色对象的名字。
subjects（[]Subject）	请参考表 16-21 中的同名字段。

ServiceAccount 对象的字段见表 16-23。

表 16-23 ServiceAccount 资源对象的字段

字段（类型）	说明

https://www.yeyopen.org

apiVersion (string)	目前的值固定为 v1
kind (string)	值固定为 ServiceAccount
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4。要在这里定义服务账号的名字。
automountServiceAccountToken (boolean)	以此服务帐户运行的 Pod 是否自动挂载令牌， 可以在 Pod 定义中覆盖本字段。
imagePullSecrets ([]LocalObjectReference)	引用同一名字空间内若干 Secret 对象，用来下拉 pod（以此服务账号运行的 Pod）中的任何镜像。 LocalObjectReference 的结构如下： ● name (string)：Secret 对象的名称。
secrets ([]ObjectReference)	引用同一名字空间内若干 Secret 对象。ObjectReference 的结构如下： ● apiVersion (string)：v1。 ● kind (string)：Secret。 ● name (string)：被引用 Secret 的名字。 ● namespace (string)：被引用的 Secret 所在的名字空间。 ● resourceVersion (string)：被引用的 Secret 的版本（如果有）。 ● uid (string)：被引用的 Secret 的 uuid（如果有）。

一些例子参考图 16-4。

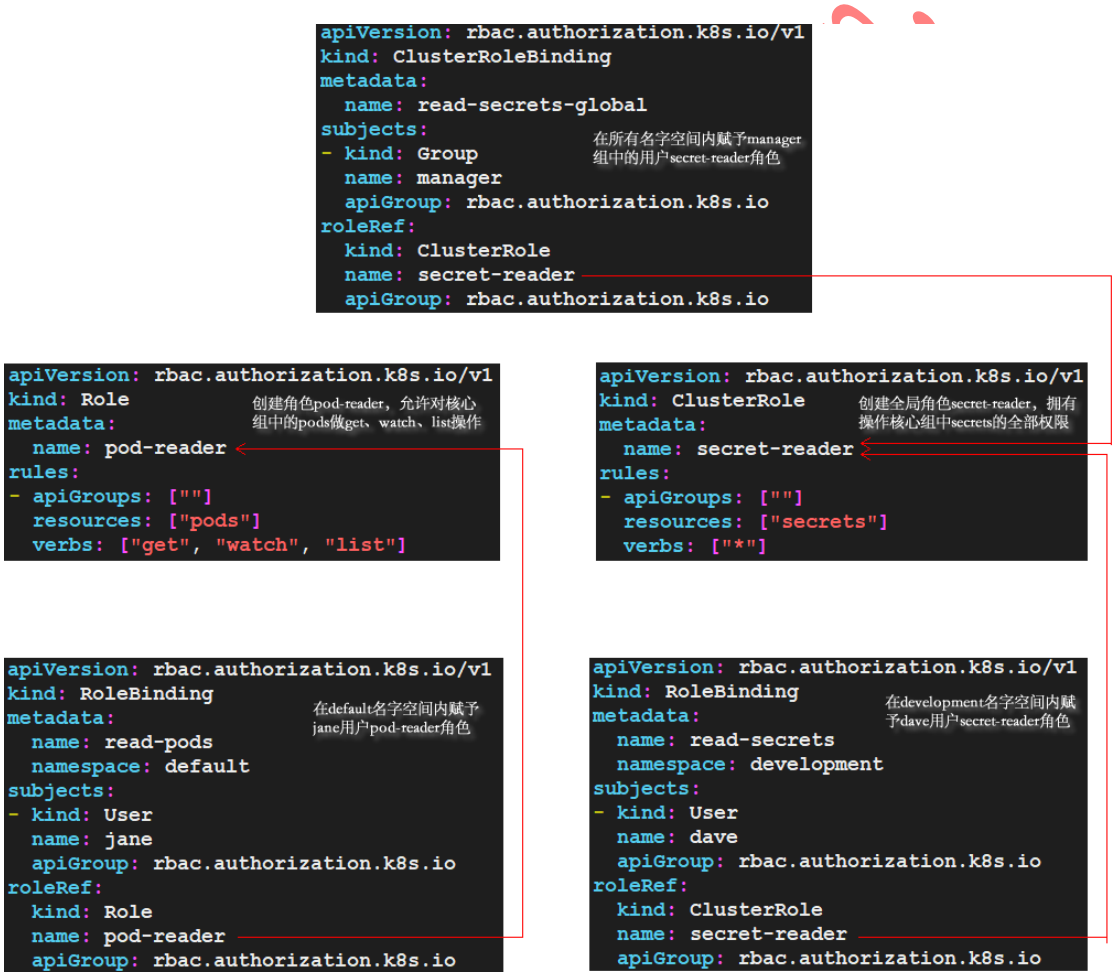


图 16-4 RBAC 的例子

把上面的代码写入文件 rbac.yaml 中，然后创建资源对象：

```
[root@master1 ~]# kubectl create namespace development                    #创建名字空间 development
[root@master1 ~]# kubectl create -f rbac.yaml
```

如果上面第 2 条命令报错“已经存在某类资源”，请先采用命令 kubectl delete -f rbac.yaml 删除再闯将。

我们再看看集群中存在哪些默认的 ClusterRole：

```
[root@master1 ~]# kubectl get clusterrole -A
NAME                                                                   CREATED AT
admin                                                                  2024-11-08T07:25:26Z                  #名字空间内的管理员角色
cluster-admin                                                          2024-11-08T07:25:26Z                  #集群管理员角色
edit                                                                   2024-11-08T07:25:26Z                  #名字空间内的编辑角色
view                                                                   2024-11-08T07:25:26Z                  #名字空间内的浏览角色
kubeadm:get-nodes                                                      2024-11-08T07:25:26Z
system:aggregate-to-admin                                              2024-11-08T07:25:26Z
system:aggregate-to-edit                                               2024-11-08T07:25:26Z
system:aggregate-to-view                                               2024-11-08T07:25:26Z
system:auth-delegator                                                  2024-11-08T07:25:26Z
system:basic-user                                                      2024-11-08T07:25:26Z
system:certificates.k8s.io:certificatesigningrequests:nodeclient       2024-11-08T07:25:26Z
system:certificates.k8s.io:certificatesigningrequests:selfnodeclient   2024-11-08T07:25:26Z
system:certificates.k8s.io:kube-apiserver-client-approver              2024-11-08T07:25:26Z
system:certificates.k8s.io:kube-apiserver-client-kubelet-approver      2024-11-08T07:25:26Z
system:certificates.k8s.io:kubelet-serving-approver                    2024-11-08T07:25:26Z
system:certificates.k8s.io:legacy-unknown-approver                     2024-11-08T07:25:26Z
system:controller:attachdetach-controller                              2024-11-08T07:25:26Z
system:controller:certificate-controller                               2024-11-08T07:25:26Z
……          <大量控制器角色，以“system:controller:”为前缀>
system:controller:ttl-controller                                       2024-11-08T07:25:26Z
system:coredns                                                         2024-11-08T07:25:26Z
system:discovery                                                       2024-11-08T07:25:26Z
system:heapster                                                        2024-11-08T07:25:26Z
system:kube-aggregator                                                 2024-11-08T07:25:26Z
system:kube-controller-manager                                         2024-11-08T07:25:26Z
system:kube-dns                                                        2024-11-08T07:25:26Z
system:kube-scheduler                                                  2024-11-08T07:25:26Z
system:kubelet-api-admin                                               2024-11-08T07:25:26Z
system:monitoring                                                      2024-11-08T07:25:26Z
system:node                                                            2024-11-08T07:25:26Z
system:node-bootstrapper                                               2024-11-08T07:25:26Z
system:node-problem-detector                                           2024-11-08T07:25:26Z
system:node-proxier                                                    2024-11-08T07:25:26Z
system:persistent-volume-provisioner                                   2024-11-08T07:25:26Z
system:public-info-viewer                                              2024-11-08T07:25:26Z
system:service-account-issuer-discovery                                2024-11-08T07:25:26Z
system:volume-scheduler                                                2024-11-08T07:25:26Z
```

面对用户的角色有：

cluster-admin：当在 ClusterRoleBinding 中使用时，授权对集群中以及所有名字空间中的全部资源进行完全控制。当在 RoleBinding 中使用时，授权控制此 RoleBinding 对象所在名字空间中的所有资源，包括名字空间本身。

admin：采用 RoleBinding 在名字空间内授予管理员角色——对名字空间中的大多数资源的读/写权限， 包括创建角色和角色绑定的能力。

edit：采用 RoleBinding 授权，允许对名字空间的大多数对象进行读/写操作。

view：浏览名字空间内的大多数对象，但不允许查看角色或角色绑定。

凡是以 system:为前缀的角色都是系统角色，只授予服务账号。可用命令“kubectl get clusterrole <角色名>”查看角色包含的权限。

下面再看看默认的 ClusterRoleBinding：

```
[root@master1 ~]# kubectl get clusterrolebinding
NAME                                                ROLE                                                                          AGE
cluster-admin                                       ClusterRole/cluster-admin                                                     65d
kubeadm:get-nodes                                   ClusterRole/kubeadm:get-nodes                                                 65d
kubeadm:kubelet-bootstrap                           ClusterRole/system:node-bootstrapper                                          65d
kubeadm:node-autoapprove-bootstrap                  ClusterRole/system:certificates.k8s.io:certificatesigningrequests:nodeclient  65d
kubeadm:node-autoapprove-certificate-rotation       ClusterRole/system:certificates.k8s.io:certificatesigningrequests:selfnodeclient  65d
kubeadm:node-proxier                                ClusterRole/system:node-proxier                                               65d
system:basic-user                                   ClusterRole/system:basic-user                                                 65d
system:controller:attachdetach-controller           ClusterRole/system:controller:attachdetach-controller                         65d
system:controller:certificate-controller            ClusterRole/system:controller:certificate-controller                          65d
system:controller:clusterrole-aggregation-controller ClusterRole/system:controller:clusterrole-aggregation-controller            65d
……  <大量针对控制器的授权，均以“system:controller:”为前缀>
system:controller:ttl-controller                    ClusterRole/system:controller:ttl-controller                                  65d
system:coredns                                      ClusterRole/system:coredns                                                    65d
system:discovery                                    ClusterRole/system:discovery                                                  65d
system:kube-controller-manager                      ClusterRole/system:kube-controller-manager                                    65d
system:kube-dns                                     ClusterRole/system:kube-dns                                                   65d
system:kube-scheduler                               ClusterRole/system:kube-scheduler                                             65d
system:monitoring                                   ClusterRole/system:monitoring                                                 65d
system:node                                         ClusterRole/system:node                                                       65d
system:node-proxier                                 ClusterRole/system:node-proxier                                               65d
system:public-info-viewer                           ClusterRole/system:public-info-viewer                                         65d
system:service-account-issuer-discovery             ClusterRole/system:service-account-issuer-discovery                           65d
system:volume-scheduler                             ClusterRole/system:volume-scheduler                                           65d
```

可见名为 cluster-admin 的 ClusterRoleBinding 对象给 XXX 授予了名为 cluster-admin 的 ClusterRole 角色，到底给谁授予了次角色呢，还要细看此 ClusterRoleBinding 的定义：

```
[root@master1 ~]# kubectl get clusterrolebinding cluster-admin -o yaml
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2024-11-08T07:25:26Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: cluster-admin
  resourceVersion: "175"
  uid: 524c6e38-7f4f-4d4f-9e32-f065280cce4a
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:masters
```

即授予了组 system:masters，而集群管理员 Kubernetes-admin 就属于此组。

集群中内建的 Role 和 RoleBinding 并不多，大家可采用类似方法了解它们。

§16.3.2 RBAC 实战

实战一、集群管理员的授权

集群管理员叫什么？属于哪个组？有什么权限？这是本实战要回答的问题。

1. 查看管理员名称和归属的组：

```
[root@master1 ~]# vim /etc/kubernetes/admin.conf
……
users:
- name: kubernetes-admin                                                                                    #管理员名称
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURWdtZ0F3SUJBZ0lJTm……        #base64 编码的证书
    client-key-data: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSJQkFBS0NBUUVBdnIyYmxRQ2……        #base64 编码的私钥
```

可见管理员为 kubernetes-admin。下面反 base64 编码证书后查看管理员属于哪个组：

```
[root@master1 ~]# more /etc/kubernetes/admin.conf
……
  user:
    client-certificate-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURLVENDQWhHZ0F3SUJBZ0lJUGZDWkwzbzVIbFF3RFFZSktvWklodmNOQVFFTEJRQXdGVEVUTUJ
FR0ExVUUKQXhNS2EzVmlaWEp1WlhSbGN6QWVGdzB5TlRFd01qa3dNekF5TXpkYUZ3MHlOakV3TWprd016QTNNemRhTUR3eApIekFkQmdOVkJBb1RGbX
QxWW1WaFpHMDZZMngxYzNSbGNpMWhaRzFwYm5NeEdUQVhCZ05WQkFNVEVHdDFZbVZ5CmJtVjBaWE10WVdSdGFXNHdnZ0VpTUEwR0NTcUdTSWIzRFFFQ
kFRVUFBNElCRHdBd2dnRUtBb0lCQVFDOEM3V0kKV1V6ZHRpOXg3a2JmRTRGSStIZHVnKy9mZkZCaTJFVVZpb2R3SCtoWW42NmxFQzNHYUVVcHBnbjV6
THF2TVRNcgpoemhhNytJT1IrYlBsT2pGUFhkM29tSjlrenNaN0tPYmlEOGJXd1N4eVRrTmh4c0U4bzMyU0lHamo3SUx6blQ3CjNPOVRWRWZmbUlkejg
1R3p2QXJiRGQxQmVCNDBnSWxGR0psaXBMbW52aElXV3drbG1YRS83ZVYyMjNSSnpzYy8KdkpDVnJXL2pvcG5WQm01TDluYUh1TmdITnMvT1h0cFVGTk
92VW1uYzQ0aGQ0bmV3VGREU1UrTHlaOHdYWkxKYgoyN2llWE4vdEwzbWZZbUErQ05ISDdLWTVaa3ZDcTlQNyszOVhLYTNCaWVlYlZjRU5haXZzTmpLW
HhTdkYzS1MrCmhVSXo0VldzY0V5b2dyN1RBZ01CQUFHalZqQlVNQTRHQTFVZER3RUIvd1FFQXdJRm9EQVRCZ05WSFNVRUREQUsKQmdnckJnRUZCUWNE
QWpBTUJnTlZIUk1CQWY4RUFqQUFNQjhHQTFVZEl3UVlNQmFBRks0NmNmTzhGb2NIZHM0aQpWWmFPcTkxbjlsOTFNQTBHQ1NxR1NJYjNEUUVCQ3dVQUE
0SUJBUUFGRUdqOHZGcm5JREptWnVYUGkzejFDWlRXCkJhQkszKzVIaHpEN04vbTlyTUkyVGd3bTJpQ3lwbkhDWjZJN2d0Q1FXV3FjbVNwUmNGN3dRN2
9lUjVvNlVFVUkKRFdyVmViV2kwZzU5RVZuM25YRzhJbHZkZTZRKzlYczZZRGFYdm5yYmFKSXVGK1RVQktJZkNQV2NBbENRR01BNQpaSGVIa09RZXA3d
Ux3QTdNdmM0WEVFTkFJU0FXRnlyVUY2a3BnRG1TT2lWRTlLNnFNcysrbXo5VytMeHNUbk95CjFRenBpRzNjSGxWdUdrZUZYY2x5R0ttRm50dWtJQXhT
RnV0SUdUc01xM1FhdXNwT0JHdE11MXBmMmVYYTZaRXQKRS9iTHlHMmZWZ1loZlNVYm0wb1NsZHdHcU9Rb29zNHZXczVKVm14L2IzMTNyaUdtQVNUM3N
xbFgrbVEzCi0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K
……

[root@master1 ~]# cat << EOF | base64 -d > /tmp/admin.crt
> LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS……DQVRFLS0tLS0K                                    #把上面整个 base64 编码的证书拷贝到这里
> EOF

[root@master1 ~]# openssl x509 -in /tmp/admin.crt -noout -text                                                #查看证书内容
Certificate:
    Data:
……
        Subject: O=kubeadm:cluster-admins, CN=kubernetes-admin                    #管理员属于组 kubeadm:cluster-admins
……
```

www.veryopen.org

可见管理员 kubernetes-admin 属于 kubeadm:cluster-admins 组。

2. 管理员肯定通过 ClusterRoleBinding 被赋予了全局角色（即 ClusterRole），查看系统中的 ClusterRoleBinding：

```
[root@master1 ~]# kubectl get clusterrolebinding
NAME                                           ROLE                                                    AGE
cluster-admin                                  ClusterRole/cluster-admin                               12d
flannel                                        ClusterRole/flannel                                     7d
kubeadm:cluster-admins                         ClusterRole/cluster-admin                               12d
kubeadm:get-nodes                              ClusterRole/kubeadm:get-nodes                           12d
……
```

从名字看，应该是 kubeadm:cluster-admins 执行了赋权操作，进一步查看其详情：

```
[root@master1 ~]# kubectl get clusterrolebindings kubeadm:cluster-admins -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  creationTimestamp: "2024-11-08T07:25:27Z"
  name: kubeadm:cluster-admins
  resourceVersion: "239"
  uid: a78ec92c-d456-4325-92eb-e44a26cca9b6
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: kubeadm:cluster-admins
```

看到了吗？即把 cluster-admin 全局角色赋予了 kubeadm:cluster-admins 组。下面看看 cluster-admin 角色包含了什么权限：

```
[root@master1 ~]# kubectl get clusterrole cluster-admin -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2024-11-08T07:25:26Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: cluster-admin
  resourceVersion: "111"
  uid: d58ae576-60b8-44e5-ada8-fc8ecd8fe039
rules:
- apiGroups:
  - '*'
  resources:
  - '*'
  verbs:
  - '*'
- nonResourceURLs:
  - '*'
  verbs:
  - '*'
```

发现对所有资源开放了所有权限！

最终得出以下结论：

K8s 集群的管理员 kubernetes-admin，它属于组 kubeadm:cluster-admins，而这个组被赋予了 cluster-admin 角色，而 cluster-admin 角色对所有资源开放了所有权限，所以管理员的权限没有限制。管理员通过证书访问集群。

实战二、名字空间管理员

任务：新建一个名字空间 gdpi，并让外部用户 alice 成为此名字空间的管理员。

1. 为 alice 产生私钥和证书：

```
[root@master1 ~]# openssl genrsa -out alice.key 2048                                    #创建 2048 位二进制的私钥
[root@master1 ~]# openssl req -new -key alice.key -out alice.csr -subj "/CN=alice/O=gdpi"  #产生证书申请文件 alice.csr
[root@master1 ~]# openssl x509 -req -in alice.csr -CA /etc/kubernetes/pki/ca.crt -CAkey \
 /etc/kubernetes/pki/ca.key -CAcreateserial -out alice.crt -days 365                    #颁发一年期有效的证书 alice.crt
Certificate request self-signature ok
subject=CN = alice, O = gdpi
```

alice 用户（CN=alice）归属于组 gdpi（O=gdpi）。

2. 创建名字空间 gdpi、角色并给 alice 用户授权：

```
[root@master1 ~]# vim alice.yaml                                                        #定义资源对象
#1. 定义名字空间 gdpi:
apiVersion: v1
kind: Namespace
metadata:
  name: gdpi

---
#2. 定义角色 gdpi-admin:
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: gdpi
  name: gdpi-admin
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]

---
#3. 在 gdpi 名字空间中把 gdpi-admin 角色赋予 alice:
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: gdpi-admin-binding
  namespace: gdpi
subjects:
- kind: User
  name: alice
roleRef:
  kind: Role
  name: gdpi-admin

[root@master1 ~]# kubectl create -f alice.yaml                                          #生成资源对象
namespace/gdpi created
role.rbac.authorization.k8s.io/gdpi-admin created
rolebinding.rbac.authorization.k8s.io/gdpi-admin-binding created

[root@master1 ~]# kubectl get rolebinding gdpi-admin-binding -o yaml -n gdpi            #查看赋权
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: "2024-11-29T02:50:31Z"
  name: gdpi-admin-binding
  namespace: gdpi
  resourceVersion: "960291"
  uid: ac6a671d-056e-436c-93bc-68ba3a6004a6
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: gdpi-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: alice
```

其实也可以直接给 alice 授予内建的全局 admin 角色即可。

3. 为 kubectl 命令配置 alice 用户：

```
[root@master1 ~]# kubectl config set-credentials alice --client-certificate alice.crt --client-key alice.key
User "alice" set.
[root@master1 ~]# kubectl config set-context alice --cluster k8scluster --namespace gdpi --user alice
Context "alice" created.
```

alice 的配置内容被加到/etc/kubernetes/admin.conf 文件中，见下：

```
[root@master1 ~]# vim /etc/kubernetes/admin.conf
……
contexts:
- context:
    cluster: k8scluster
    namespace: gdpi
```

```
    user: alice
  name: alice
- context:
    cluster: k8scluster
    user: kubernetes-admin
  name: kubernetes-admin@k8scluster
current-context: kubernetes-admin@k8scluster
kind: Config
preferences: {}
users:
- name: alice
  user:
    client-certificate: /root/alice.crt
    client-key: /root/alice.key
- name: kubernetes-admin
  user:
……
```

检查存在哪些被授权用户：

```
[root@master1 ~]# kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
          alice                         k8scluster   alice              gdpi
*         kubernetes-admin@k8scluster   k8scluster   kubernetes-admin
```

表明当前用户为 kubernetes-admin。

4. 切换当前用户为 alice：

```
[root@master1 ~]# kubectl config use-context alice
Switched to context "alice".
```

当然不切换也可以，但 kubectl 命令要增加参数--context，如“kubectl --context alice ...”。

下面验证 alice 能否在 gdpi 名字空间内启动一个 pod：

```
[root@master1 ~]# vim alice-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: alice-pod-busybox
  namespace: gdpi
spec:
  containers:
    - image: busybox
      command:
        - sleep
        - "1d"
      imagePullPolicy: IfNotPresent
      name: container-busybox

[root@master1 ~]# kubectl create -f alice-pod.yaml
pod/alice-pod-busybox created

[root@master1 ~]# kubectl get pod -o wide
NAME                READY   STATUS    RESTARTS   AGE   IP            NODE      NOMINATED NODE   READINESS GATES
alice-pod-busybox   1/1     Running   0          29s   10.244.2.12   worker2   <none>           <none>
```

由于 alice 没有列举所有名字空间内的 pod 的权限，需要 kubernetes-admin@k8scluster 用户才行：

```
[root@master1 ~]# kubectl get pods -n gdpi -o wide --context kubernetes-admin@k8scluster -A
NAMESPACE     NAME                                READY   STATUS    RESTARTS      AGE    IP              NODE      NOMINATED NODE   READINESS GATES
default       pod-busybox                         1/1     Running   3 (38m ago)   2d5h   10.244.4.11     worker3   <none>           <none>
gdpi          alice-pod-busybox                   1/1     Running   0             7s     10.244.1.8      worker1   <none>           <none>
kube-flannel  kube-flannel-ds-2q99l               1/1     Running   8 (38m ago)   13d    10.10.10.33     worker3   <none>           <none>
kube-flannel  kube-flannel-ds-dqzrz               1/1     Running   8 (38m ago)   13d    10.10.10.32     worker2   <none>           <none>
kube-flannel  kube-flannel-ds-f9tpw               1/1     Running   8 (38m ago)   13d    10.10.10.11     master1   <none>           <none>
kube-flannel  kube-flannel-ds-x7zx7               1/1     Running   8 (38m ago)   13d    10.10.10.31     worker1   <none>           <none>
kube-system   coredns-7cc97dffdd-fkwfc            1/1     Running   5 (38m ago)   7d5h   10.244.1.7      worker1   <none>           <none>
kube-system   coredns-7cc97dffdd-g6tjm            1/1     Running   8 (38m ago)   16d    10.244.2.11     worker2   <none>           <none>
kube-system   etcd-master1                        1/1     Running   11 (38m ago)  16d    10.10.10.11     master1   <none>           <none>
kube-system   kube-apiserver-master1              1/1     Running   11 (38m ago)  16d    10.10.10.11     master1   <none>           <none>
kube-system   kube-controller-manager-master1     1/1     Running   11 (38m ago)  16d    10.10.10.11     master1   <none>           <none>
kube-system   kube-proxy-8jh2w                    1/1     Running   9 (38m ago)   14d    10.10.10.33     worker3   <none>           <none>
kube-system   kube-proxy-bpfrx                    1/1     Running   9 (38m ago)   14d    10.10.10.31     worker1   <none>           <none>
kube-system   kube-proxy-dpvxw                    1/1     Running   9 (38m ago)   14d    10.10.10.32     worker2   <none>           <none>
kube-system   kube-proxy-mxz2r                    1/1     Running   11 (38m ago)  16d    10.10.10.11     master1   <none>           <none>
kube-system   kube-scheduler-master1              1/1     Running   11 (38m ago)  16d    10.10.10.11     master1   <none>           <none>
```

到此实战完成，但是 alice 没有 gdpi 之外的任何权限，所以不能操作全局的资源，如列举 pod。当然我们也可采用命令 kubectl config use-context kubernetes-admin@k8scluster 切换到管理员。

为了方便能在其他电脑上使用 alice 用户来管理 k8s 集群，最好把其私钥和证书嵌入到 /etc/kubernetes/admin.conf 文件中，参考下面的方法。

```
[root@master1 ~]# cat alice.key | base64 --wrap=0 > 1                    #私钥转为 base64 编码并保存在 1 文件中
```

```
[root@master1 ~]# cat alice.crt | base64 --wrap=0 > 2                    #证书转为 base64 编码并保存在 2 文件中

[root@master1 ~]# vim /etc/kubernetes/admin.conf
......
users:
- name: alice
  user:
    client-certificate-data: <把文件 2 的内容放在这里>
    client-key-data: <把文件 1 的内容放在这里>
......
```

注意：千万不要修改 kubernetes-admin 用户，是修改 alice 用户。

查看用户配置：

```
[root@master1 ~]# kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://10.10.10.11:6443
  name: k8scluster                                                                                    #集群名称
contexts:
- context:
    cluster: k8scluster
    namespace: gdpi
    user: alice
  name: alice
- context:
    cluster: k8scluster
    user: kubernetes-admin
  name: kubernetes-admin@k8scluster
current-context: kubernetes-admin@k8scluster
kind: Config
preferences: {}
users:
- name: alice
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
- name: kubernetes-admin
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
```

§16.4 k8s 中的存储

§16.4.1 基础

k8s 支持各种各样的存储（包括第三方产品），如 Ceph、NFS、iSCSI 等等，存储的最终使用者是 pod 中的容器，为此，kubernetes 引入了若干与存储相关的资源对象，包括 Volume、PersistentVolumeClaim（简称为 PVC）、PersistentVolume（简称为 PV）、StorageClass 和 VolumeAttachment。这些资源对象的关系参考图 16-5 所示。

https://www.

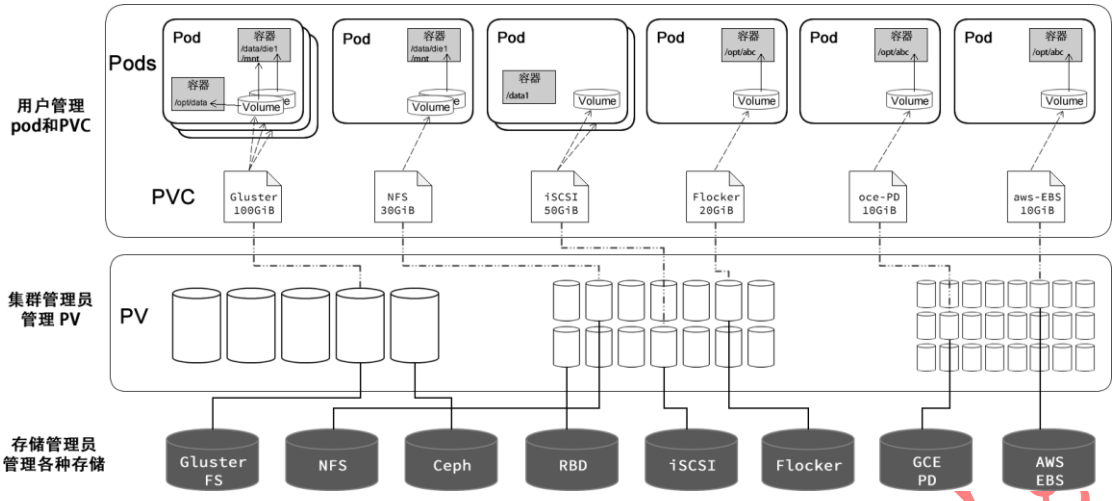


图 16-5 静态 PV

存储管理员负责给工作节点提供存储，并确保能在节点上能使用这些存储资源。k8s 集群管理员负责配置相应的 PVP（Persistent Volume Provisioner）和创建 PV。用户创建 PVC 和 Pod，并通过资源配置把 PVC 挂载到 Pod 上，最终由 pod 内的容器按需再挂载到容器中的目录上。用户创建 PVC 时需要指明捆绑到哪个 PV，所以 PVC 会消耗 PV，PVC 是名字空间内的资源，而 PV 是集群范围内的资源。相互之间的引用参考如 16-6。

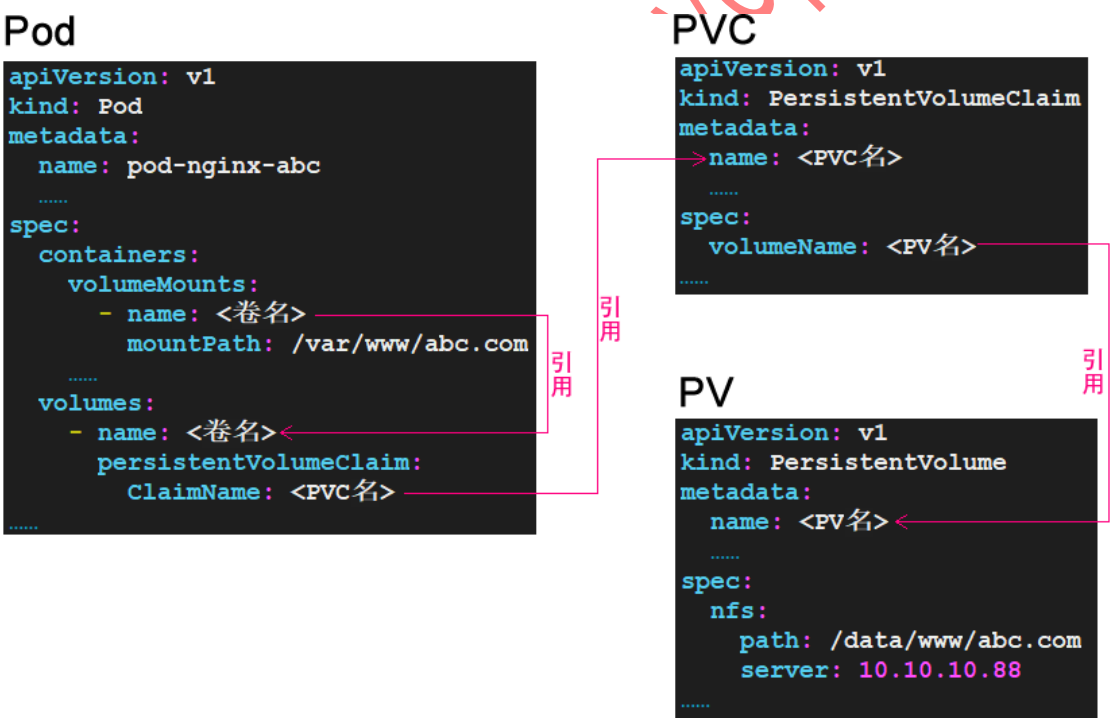


图 16-6 Pod、容器、PVC 和 PV 之间的引用

由管理员事先创建的 PV 称为静态 PV，但在稍具规模的集群中，pod 和容器特别多，而且时时变动，因此它们使用的存储也会经常发生变化，而且数量特别多。集群管理员一般事先不知道需要哪些 PV 以及数量，所以很难做到事先创建全部需要的 PV，为此 k8s 引入了抽象资源类 StorageClass，以后就可以根据 StorageClass 自动按需创建 PV，这样的 PV 称为动态 PV。图 16-7 综合了静态和动态 PV 的使用流程。

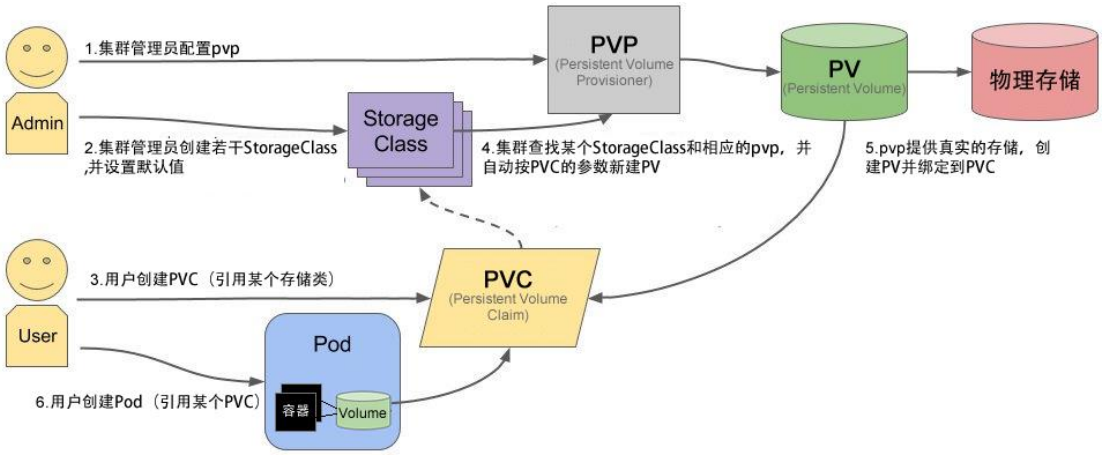


图 16-7 静态和动态 PV 使用流程

PV 资源对象的字段定义参考表 16-24。

表 16-24 PersistentVolume 资源对象的字段

字段（类型）	说明
apiVersion（string）	目前的值固定为 v1
kind（string）	值固定为 PersistentVolume
metadata（ObjectMeta）	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
spec（PersistentVolumeSpec）	定义 PersistentVolume 的行为。关于 PersistentVolumeSpec 对象请参考表 16-25
status（PersistentVolumeStatus）	PersistentVolume 的当前状态。状态信息只能读

表 16-25 PersistentVolumeSpec 对象的字段

字段（类型）	说明
accessModes（[]string）	指定若干访问模式，卷的访问模式有 ReadWriteOnce（同一时刻只允许单节点读/写）、ReadOnlyMany（同一时刻允许多个节点只读）、ReadWriteMany（同一时刻允许多个节点都和写）、ReadWriteOncePod（单个 Pod 以读写方式挂载）。
capacity（map[string]Quantity）	持久卷的资源和容量，如 capacity：10Gi
claimRef（ObjectReference）	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
mountOptions（[]string）	挂载选项的列表，例如［"ro"，"soft"］。 针对此字段无合法性检查——如果某选项无效，则只是挂载会失败。
nodeAffinity（VolumeNodeAffinity）	可以从哪些节点访问此卷的约束限制。此字段会影响调度使用此卷的 Pod。VolumeNodeAffinity 的结构如下： ● required（NodeSelector） - 必须满足的硬性节点约束。NodeSelector 请参考 PodSpec.affinity.nodeAffinity.o requiredDuringSchedulingIgnoredDuringExecution。
persistentVolumeReclaimPolicy（string）	PV 与 PVC 松绑之后，回收 PV 的策略，可取值 Retain（对于静态 PV，这是默认值）、Delete（对于动态 PV，这是默认值）。释放的静态 PV 不会删除，但是也不能与其他 PVC 捆绑，因为上面还有之前 PVC 保存的数据，除非管理员手工操作：删除 PV；清除对应的底层存储上的数据；删除底层存储。
storageClassName（string）	指定本 PV 归属的 StorageClass，空值表示 pv 不属于任何 StorageClass。
volumeMode（string）	指定卷类型，可取值 Filesystem（文件系统。这是默认值）、Block（块设备）。
hostPath（HostPathVolumeSource）	参考 Volume.hostPath。
awsElasticBlockStore、azureDisk、azureFile、cephfs、cinder、fc、flexVolume、flocker、gcePersistentDisk、glusterfs、iscsi、nfs、photonPersistentDisk、portworxVolume、quobyte、rbd、scaleIO、storageos、vsphereVolume 字段请参考表 16-6。	

PVC 的字段说明请参考表 16-26。

表 16-26 PersistenVolumeClaim 资源对象的字段

字段（类型）	说明

https://www.veryopen.org

apiVersion (string)	目前的值固定为 v1
kind (string)	值固定为 PersistentVolumeClaim
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 16-4
spec (PersistentVolumeClaimSpec)	定义 PersistentVolumeClaim 的行为。关于 PersistentVolumeClaimSpecd 的结构如下： ● accessModes ([]string)：参考表 16-25 中的同名字段。 ● selector (LabelSelector)：选择要绑定的 PV，选择结果集中不能多于一个 PV。详情请参考 Volume.ephemeral.volumeClaimTemplate.spec.selector（表 16-6）。 ● resources (ResourceRequirements)：最小计算资源，具体参考表 16-6 的同名字段。 ● volumeName (string)：欲绑定的 PV 的名称。本字段与 selector 字段互斥。 ● storageClassName (string)：引用的 StorageClass 名称，对于动态 PV，本字段不能空。 ● volumeMode (string)：指定卷类型，可取值 Filesystem（文件系统。这是默认值）、Block（块设备）。
status (PersistentVolumeStatus)	PersistentVolume 的当前状态。状态信息只能读

StorageClass 资源对象的字段定义参考表 16-27。

表 16-27 StorageClass 资源对象的字段

字段（类型）	说明
apiVersion (string)	目前的值固定为 storage.k8s.io/v1
kind (string)	值固定为 StorageClass
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 11-3
provisioner (string)	引用 PVP，即 PVP 的名字。
allowVolumeExpansion (boolean)	是否允许卷扩充。
mountOptions ([]string)	请参考表 16-25 中的同名字段
parameters (map[string]string)	给 PVP 提供参数，用于创建本 StorageClass 的 PV。
reclaimPolicy (string)	指定动态 PV 的回收策略，默认为 Delete。详情请参考 persistentVolume.persistentVolumeReclaimPolicy 字段。
volumeBindingMode (string)	如何绑定 PersistentVolumeClaim，未设置本字段时，将使用 VolumeBindingImmediate。 只有启用了 VolumeScheduling 功能特性的服务器才能使用此字段。

StorageClassList 为存储类的集合，其字段定义参考表 16-28。

表 16-28 StorageClassList 资源对象的字段

字段（类型）	说明
apiVersion (string)	目前的值固定为 storage.k8s.io/v1
kind (string)	值固定为 StorageClassList
items ([]StorageClass)	StorageClass 的列表

https://www.vcryopen.org

§16.4.2 NFS 实战

现在我们来实战，采用 NFS 做存储后端。

首先需要准备 NFS 服务器，我们不妨就在 master1 上配置 NFS 服务器。

一. 准备 NFS 服务器

在 master1 操作操作：

```
[root@master1 ~]# dnf install nfs-utils -y
[root@master1 ~]# mkdir -p /opt/data/{pv01,pv02}
[root@master1 ~]# echo '/opt/data *(rw,no_root_squash,sync)' >> /etc/exports
```

```
[root@master1 ~]# exportfs -r && exportfs
/opt/data       <world>

[root@master1 ~]# systemctl --now enable rpcbind nfs-server
[root@master1 ~]# showmount -e 10.10.10.11
Export list for 10.10.10.11:
/opt/data *
```

~~如果防火墙没关闭，就设置如下防火墙策略：~~

```
~~[root@master1 ~]# firewall-cmd --permanent --add-service={nfs,rpc-bind,mountd}~~
~~[root@master1 ~]# firewall-cmd --reload~~
```

在所有工作节点上操作：

```
[root@worker? ~]# dnf install nfs-utils -y
[root@worker? ~]# showmount -e 10.10.10.11
Export list for 10.10.10.11:
/opt/data *

测试能否挂载 NFS 文件系统：
[root@worker? ~]# mount -t nfs master1:/opt/data/pv01 /mnt
[root@worker? ~]# df -h /mnt
master1:/opt/data/pv01 nfs4     33487872 3804160 29683712   12% /mnt

[root@worker? ~]# umount /mnt
```

虽然 NFS 服务器上输出/opt/data 目录，但是允许客户端挂载其中的任何子目录。

二. 静态 PV 实验

静态 PV 需要管理员手工创建。

1. 首先创建 pv 资源对象：

```
[root@master1 ~]# vim nfs-static-pv01.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv01
  labels:
    pv: nfs-pv01
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /opt/data/pv01
    server: 10.10.10.11
```

请参考表 16-24 理解各个键、值。

```
[root@master1 ~]# kubectl config get-contexts                          #确保当前用户为管理员
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
          alice                         k8scluster   alice              gdpi
*         kubernetes-admin@k8scluster   k8scluster   kubernetes-admin
```

如果当前用户不是管理员，请使用命令“kubectl config use-context kubernetes-admin@k8scluster”切换成管理员。

创建 pv：

```
[root@master1 ~]# kubectl apply -f nfs-static-pv01.yaml
persistentvolume/nfs-pv01 created

[root@master1 ~]# kubectl get pv -o wide
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE     VOLUMEMODE
nfs-pv01   1Gi        RWO            Retain           Available                                   6m17s   Filesystem
```

2. 创建 pvc 资源对象：

```
[root@master1 ~]# vim nfs-static-pvc01.yaml
apiVersion: v1
```

```
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc01
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  selector:                                                                                                    #选择 pv 资源对象
    matchLabels:
      pv: nfs-pv01

[root@master1 ~]# kubectl apply -f nfs-static-pvc01.yaml
persistentvolumeclaim/nfs-pvc01 created

[root@master1 ~]# kubectl get pvc -o wide
NAME        STATUS   VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS   AGE    VOLUMEMODE
nfs-pvc01   Bound    nfs-pv01   1Gi        RWO                           9m5s   Filesystem
```

可见 nfs-pvc01 绑定到了 nfs-pv01。

接下来我们创建一个 pod，让它使用上面创建的 pvc：

```
[root@master1 ~]# vim nfs-static-pod01.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nfs-pod01
spec:
  containers:
    - name: frontend
      image: nginx
      imagePullPolicy: IfNotPresent
      volumeMounts:
      - mountPath: "/usr/share/nginx/html"
        name: nfs-static
  volumes:
    - name: nfs-static
      persistentVolumeClaim:
        claimName: nfs-pvc01

[root@master1 ~]# kubectl create -f nfs-static-pod01.yaml
pod/nfs-pod01 created

[root@master1 ~]# kubectl get pod -o wide
NAME        READY   STATUS    RESTARTS   AGE     IP           NODE      NOMINATED NODE   READINESS GATES
nfs-pod01   1/1     Running   0          9m22s   10.244.1.5   worker1   <none>           <none>
```

如果 nfs-pod01 的状态不是 Running，请使用命令 kubectl describe pod nfs-pod01 查看错误信息，一般是容器镜像下拉失败，可以考虑在对应的节点上手工下拉 nginx 镜像（命令 ctr -n k8s.io images pull docker.rainbond.cc/library/nginx:latest, ctr -n k8s.io images tag docker.rainbond.cc/library/nginx:latest docker.io/library/nginx:latest），或者从文件中导入（导入命令 ctr -n k8s.io images import nginx_latest.tar）。然后再过十几秒中，nfs-pod01 就正常了。

进入 pod 中看看：

```
[root@master1 ~]# kubectl exec -it nfs-pod01 -- /bin/bash
root@nfs-pod01:/# df -h /usr/share/nginx/html
Filesystem                Size  Used Avail Use% Mounted on
10.10.10.11:/opt/data/pv01   32G  3.7G   29G  12% /usr/share/nginx/html

root@nfs-pod01:/# echo "Hello PV!" > /usr/share/nginx/html/index.html
```

可见静态 PV 被挂载到了容器的/usr/share/nginx/html 目录，所以也可以直接在 master1 的/opt/data/pv01 目录中修改页面文件。在任何一个节点上浏览页面（注意 10.244.1.5 要用 pod 真实的 ip 替换）：

```
[root@worker1 ~]# curl 10.244.1.5
Hello PV!
```

删除 pod：

```
[root@master1 ~]# kubectl delete -f nfs-static-pod01.yaml
pod "nfs-pod01" deleted
[root@master1 ~]# kubectl get pvc
NAME        STATUS   VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
nfs-pvc01   Bound    nfs-pv01   1Gi        RWO                           16m
[root@master1 ~]# kubectl get pv
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM               STORAGECLASS   REASON   AGE
nfs-pv01   1Gi        RWO            Retain           Bound    default/nfs-pvc01
```

虽然删除了 pod，但是 pvc 还是与 pv 绑定在一起。继续删除 pvc：

```
[root@master1 ~]# kubectl delete -f nfs-static-pvc01.yaml
persistentvolumeclaim "nfs-pvc01" deleted
[root@master1 ~]# kubectl get pv
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM              STORAGECLASS   REASON   AGE
nfs-pv01   1Gi        RWO            Retain           Released   default/nfs-pvc01                           18m
[root@master1 ~]# more /opt/data/pv01/index.html
Hello PV!
[root@master1 ~]# kubectl delete -f nfs-static-pv01.yaml
persistentvolume "nfs-pv01" deleted
```

现在 pv 处于被释放状态，继续删除 pv 之后，发现 nfs 上的文件 index.html 还在，因为 pv 的 persistentVolumeReclaimPolicy 的字段默认为 retain（参考表 16-25）。

pod、pvc、pv 和 nfs 服务器的链接参考图 16-8。



图 16-8 pod、pvc、静态 pv 和 nfs 服务器的关系

三. 动态 PV 实验

静态的 pv 需要 k8s 集群管理员手工创建，这在大规模应用场景中不太合适，取而代之的是利用 StorageClass 存储类实现按需动态创建 PV。

需要一个提供 nfs 卷的 PVP（Persistent Volume Provisioner），nfs-client-provisioner 已经不再支持了，现在可用 nfs-subdir-external-provisioner 或者 nfs-ganesha-server-and-external-provisioner，本实战采用前者。

1. 把 nfs-subdir-external-provisioner 项目克隆下来：

```
[root@master1 ~]# dnf install git -y
[root@master1 ~]# git clone https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner/
[root@master1 ~]# cd nfs-subdir-external-provisioner-master/
```

国内可能无法访问 github.com 网站，不妨尝试在 windows 上的浏览器中打开网址 https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner，下载 zip 文件，然后在 Linux 上使用命令“unzip nfs-subdir-external-provisioner-master.zip”解压缩（如果 unzip 命令不能存在，请安装它:dnf install unzip -y）。

2. 创建服务账号 nfs-client-provisioner 并授权（PVP 创建 PV 需要一定的权限）：

```
[root@master1 ~]# cd ~/nfs-subdir-external-provisioner-master
[root@master1 nfs-subdir-external-provisioner-master]# more deploy/rbac.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nfs-client-provisioner
  # replace with namespace where provisioner is deployed
  namespace: default
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-client-provisioner-runner
rules:
  - apiGroups: [""]
    resources: ["nodes"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["create", "update", "patch"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: run-nfs-client-provisioner
subjects:
  - kind: ServiceAccount
    name: nfs-client-provisioner
    # replace with namespace where provisioner is deployed
    namespace: default
roleRef:
  kind: ClusterRole
  name: nfs-client-provisioner-runner
  apiGroup: rbac.authorization.k8s.io
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-nfs-client-provisioner
  # replace with namespace where provisioner is deployed
  namespace: default
rules:
  - apiGroups: [""]
    resources: ["endpoints"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: leader-locking-nfs-client-provisioner
  # replace with namespace where provisioner is deployeo
    namespace: default
roleRef:
  kind: Role
  name: leader-locking-nfs-client-provisioner
  apiGroup: rbac.authorization.k8s.io
```

可见定义了服务账号 nfs-client-provisioner，并授予全局角色 nfs-client-provisioner-runner 和局部角色 leader-locking-nfs-client-provisioner。下面创建这些资源对象：

```
[root@master1 nfs-subdir-external-provisioner-master]# kubectl create -f deploy/rbac.yaml
serviceaccount/nfs-client-provisioner created
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-runner created
```

```
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-provisioner created
role.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
```

3. 创建 StorageClass 存储类，PVC 通过此类委托 PVP 创建 PV：

```
[root@master1 nfs-subdir-external-provisioner-master]# more deploy/class.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-client
provisioner: k8s-sigs.io/nfs-subdir-external-provisioner
parameters:
  archiveOnDelete: "false"

[root@master1 nfs-subdir-external-provisioner-master]# kubectl create -f deploy/class.yaml
storageclass.storage.k8s.io/nfs-client created
```

4. 创建 Deployment，其中包含一个 Pod，Pod 中有一个容器 nfs-client-provisioner。pod 使用服务账号 nfs-client-provisioner 访问 kube-apiserver。

```
[root@master1 nfs-subdir-external-provisioner-master]# vim deploy/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
  # replace with namespace where provisioner is deployed
  namespace: default
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image: registry.k8s.io/sig-storage/nfs-subdir-external-provisioner:v4.0.2
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: k8s-sigs.io/nfs-subdir-external-provisioner
            - name: NFS_SERVER
              value: 10.10.10.11
            - name: NFS_PATH
              value: /opt/data/pv02
      volumes:
        - name: nfs-client-root
          nfs:
            server: 10.10.10.11
            path: /opt/data/pv02

[root@master1 nfs-subdir-external-provisioner-master]# kubectl create -f deploy/deployment.yaml
deployment.apps/nfs-client-provisioner created

[root@master1 nfs-subdir-external-provisioner-master]# kubectl get pods
NAME                                     READY   STATUS    RESTARTS      AGE
nfs-client-provisioner-67487d8678-lvpl4   1/1     Running   3 (13h ago)   41h
```

一定要等到 pod 正常运行（即上面最后一条命令的状态为 Running），才能继续往下操作。因为要下拉容器镜像 registry.k8s.io/sig-storage/nfs-subdir-external-provisioner_v4.0.2.tar，在国内需要较长时间甚至根本下拉不了，这时可在能上外网的电脑上手工下来并导出：

```
ctr -n k8s.io images pull registry.k8s.io/sig-storage/nfs-subdir-external-provisioner:v4.0.2
ctr -n k8s.io images export nfs-subdir-external-provisioner_v4.0.2.tar registry.k8s.io/sig-storage/nfs-subdir-external-provisioner:v4.0.2
```

然后把 nfs-subdir-external-provisioner_v4.0.2.tar 导入到本机：

```
ctr -n k8s.io images import nfs-subdir-external-provisioner_v4.0.2.tar
```

5. 创建 PVC 和 Pod：

deploy/test-claim.yaml 和 deploy/test-pod.yaml 分别提供了 PersistentVolumeClaim 和 pod 案例，我们稍作修改：

```
[root@master1 nfs-subdir-external-provisioner-master]# vim deploy/test-claim.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-claim
spec:
  storageClassName: nfs-client
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 10Mi

[root@master1 nfs-subdir-external-provisioner-master]# kubectl create -f deploy/test-claim.yaml
persistentvolumeclaim/test-claim created

[root@master1 nfs-subdir-external-provisioner-master]# kubectl get pvc
NAME         STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
test-claim   Bound    pvc-1afb2541-a191-4dc4-a870-e8f632986602   10Mi       RWX            nfs-client     <unset>                 103s

[root@master1 nfs-subdir-external-provisioner-master]# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
pvc-1afb2541-a191-4dc4-a870-e8f632986602   10Mi       RWX            Delete           Bound    default/test-claim   nfs-client     <unset>                 94s
```

可见自动创建了 PV，名字就是 pvc-1afb2541-a191-4dc4-a870-e8f632986602，并且已经与 test-claim 这个 PVC 绑定在一起了。

下面我们来创建 pod（此 pod 会使用 PVC “test-claim”）：

```
[root@master1 nfs-subdir-external-provisioner-master]# vim deploy/test-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: test-pod
    image: busybox:stable
    command:
      - "sleep"
      - "5h"
    volumeMounts:
      - name: nfs-pvc
        mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
    - name: nfs-pvc
      persistentVolumeClaim:
        claimName: test-claim

[root@master1 nfs-subdir-external-provisioner-master]# kubectl create -f deploy/test-pod.yaml
pod/test-pod created
```

6. 验证：

```
[root@master1 ~]# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                STORAGECLASS   REASON   AGE
pvc-6f62489f-6e35-4186-ba47-40686c39211e   10Mi       RWX            Delete           Bound    default/test-claim   nfs-client             56m

[root@master1 ~]# kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
nfs-client-provisioner-67487d8678-lvpl4   1/1     Running   0          53m
test-pod                                 1/1     Running   0          5m57s

[root@master1 ~]# kubectl exec -it test-pod -- /bin/sh
/ # df -h /mnt
Filesystem                                                                                    Size   Used Available Use% Mounted on
10.10.10.11:/opt/data/pv02/default-test-claim-pvc-6f62489f-6e35-4186-ba47-40686c39211e   31.9G  3.9G 28.1G     12% /mnt

/ # echo "hello dynamic PV!" >/mnt/hello.txt
/ # exit

[root@master1 ~]# more /opt/data/pv02/default-test-claim-pvc-6f62489f-6e35-4186-ba47-40686c39211e/hello.txt
hello dynamic PV!
```

Pod、PVC、StorageClass 和 PVP 的关系参见图 16-9。

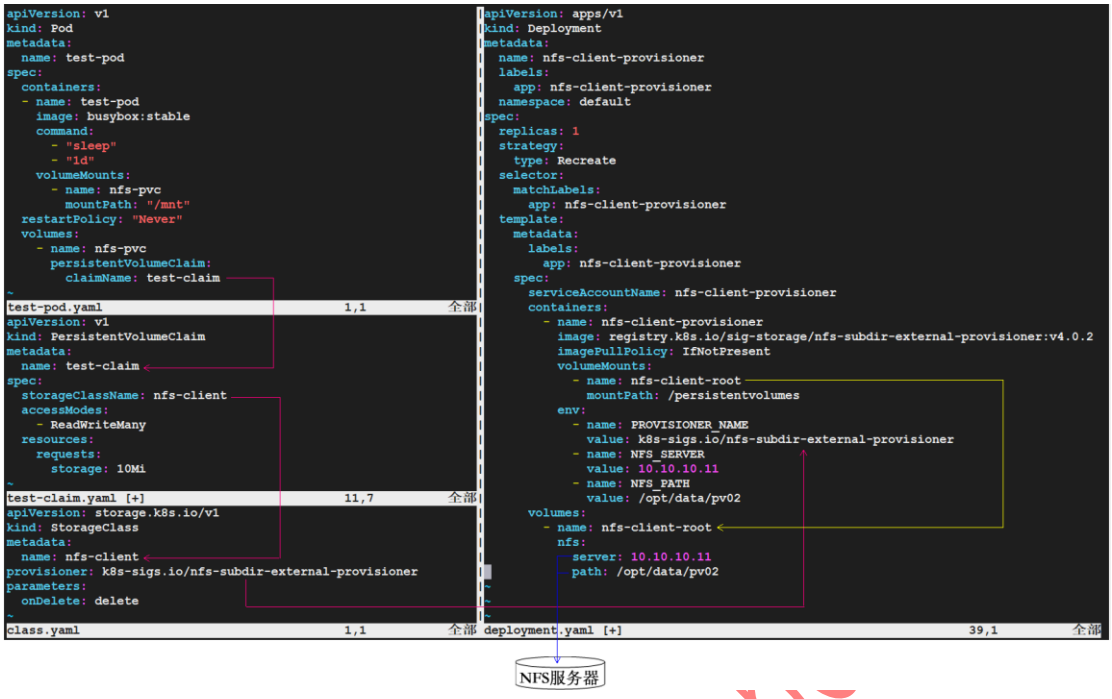


图 16-9 Pod、PVC、StorageClass 和 PVP 的关系

§16.5 管理 k8s 集群

§ 16.5.1 kubectl/ctr 命令

主要采用命令 kubectl、ctr 和 crictl 等。

一、kubectl 命令

详细介绍请参考 https://kubernetes.io/zh-cn/docs/reference/kubectl/。

此命令用来管理 k8s 集群上的资源，作为一个命令行工具，功能非常强大。它的使用语法如图 16-10 所示。

https://www.veryo



图 16-10 kubectl 用法

1．创建资源对象

语法 1：kubectl create -f yaml 或 json 格式的文件

语法 2：

```
kubectl create
  namespace|ns NAME                                                                          #新建名字空间
  service|svc
    clusterip NAME [--tcp=<port>:<targetPort>]                                               #新建ClusterIP服务
    externalname NAME --external-name EXTERNAL-NAME                                          #新建ExternalName服务
    loadbalancer NAME [--tcp=port:targetPort]                                                #新建LoadBalancer服务
    nodeport NAME [--tcp=port:targetPort]                                                    #新建NodePort服务
  deployment|deploy NAME --image=image -- [COMMAND] [args...]                               #新建DeployMent(部署)
  role NAME --verb=verb --resource=resource.group/subresource [--resource-name=resourcename] #新建单规则局部角色
  rolebinding NAME --clusterrole=NAME|--role=NAME [--user=username] [--group=groupname]
    [--serviceaccount=namespace:serviceaccountname]                                          #新建授权
  clusterrole NAME --verb=verb --resource=resource.group [--resource-name=resourcename]     #新建全局角色
  clusterrolebinding NAME --clusterrole=NAME [--user=username] [--group=groupname]
    [--serviceaccount=namespace:serviceaccountname]                                          #新建授权
  serviceaccount|sa NAME                                                                     #新建服务账号
  configmap|cm NAME [--from-file=[key=]source] [--from-literal=key1=value1]                  #新建ConfigMap(配置)
  cronjob|cj NAME --image=image --schedule='0/5 * * * ?' -- [COMMAND] [args...] [flags]      #新建定时任务
  ingress|ing NAME --rule=host/path=service:port[,tls[=secret]]                              #新建Ingress
  job NAME --image=image [--from=cronjob/name] -- [COMMAND] [args...]                        #新建后台任务
  quota|resourcequota NAME [--hard=key1=value1,key2=value2] [--scopes=Scope1,Scope2]         #新建资源配额
  secret
    generic NAME [--type=string] [--from-file=[key=]source] [--from-literal=key1=value1]     #一般密码对象
    tls NAME --cert=path/to/cert/file --key=path/to/key/file                                 #TLS密码对象
    docker-registry NAME --docker-username=user --docker-password=password --docker-email=email
      [--docker-server=string] [--from-file=[key=]source]                                    #Docker库密码对象
  token SERVICE_ACCOUNT_NAME [options]                                                       #新建令牌
  priorityclass|pc NAME --value=VALUE --global-default=BOOL                                  #新建优先类
  poddisruptionbudget|pdb NAME --selector=SELECTOR --min-available=N                         #新建Pod中断预算
```

资源对象建议定义在 yaml 格式的文件中，然后采用语法 1 来创建。语法 2 是针对不同的资源对象而言的，资源不通，语法也各异。

下面针对语法 2 举一些实际的例子。

新建名字空间 myspace：

```
[root@master1 ~]# kubectl create namespace myspace
namespace/myspace created
```

也可一采用命令 Kubectl create ns myspace，ns 是 namespace 的简称。

新建 ClusterIP 服务 my-cs（服务网上的 5678/tcp 端口映射到 pod 网上的 8080/tcp 端口）：

```
[root@master1 ~]# kubectl create service clusterip my-cs --tcp=5678:8080
deployment.apps/my-dep created
```

新建 DeployMent(部署)：

```
新建 sle，一个 pod 里只有一个容器，容器启动时运行 sleep 1d 命令：
[root@master1 ~]# kubectl create deployment sle --image=busybox -- sleep 1d
deployment.apps/sle created

新建 www-dep(三个 pod，容器里运行 web 服务)：
[root@master1 ~]# kubectl create deployment www-dep --image=nginx --replicas=3
service/www-cs created

新建 bb-www-dep，pod 里跑两个容器：
[root@master1 ~]# kubectl create deployment bb-www-dep --image=busybox:latest --image=nginx
deployment.apps/bb-www-dep created
```

新建 foo 角色，要求对 pods、pods/status 具有 get、list、watch 权限：

```
[root@master1 ~]# kubectl create role foo --verb=get,list,watch --resource=pods,pods/status
role.rbac.authorization.k8s.io/foo created
```

把全局角色 admin 授予用户 alice、john 和组 sales：

```
[root@master1 ~]# kubectl create rolebinding admin --clusterrole=admin --user=alice --user=john --group=sales
rolebinding.rbac.authorization.k8s.io/admin created
```

新建 ConfigMap(配置)：

```
包含两对键值 name=erpuser、pass=a123456：
[root@master1 ~]# kubectl create configmap my-config --from-literal=name=erpuser --from-literal=pass=a123456
configmap/my-config created

键值对来自文件/tmp/secret.configmap：
[root@master1 ~]# kubectl create configmap mysecret --from-file=/tmp/secret.configmap
configmap/mysecret created
```

--from-file 可多次重复，从而指明键值对来自多个文件。

新建一个定时任务，要求周日 4:0 在容器里执行 date 命令：

```
[root@master1 ~]# kubectl create cronjob my-job --image=busybox --schedule="0 4 * * 0" -- date
cronjob/my-job created
```

时间的格式为“分钟　小时　日期　月份　星期”，其中字段“分钟”、“小时”、“日期”、“月份”和“星期”允许出现的字符有数字、-、*、/和,。分钟范围是 0~59，小时范围是 0~23，日期范围是 1~31，月份范围是 1~12，星期范围是 0~6（0 代表星期日），也可以采用简写：mon、tue、wed、thu、fri、sat 和 sun。“/”符号右边是数字，表示每隔多少，比如“小时”字段为*/3，表示每隔 3 小时。例如“0 */6 * 5,8,11 1-5”表示在 5、8 和 11 月的工作日，每隔 6 个小时的整点。

新建密码对象：

```
一般密码对象：
[root@master1 ~]# kubectl create secret generic my-secret --from-file=path/to/bar
[root@master1 ~]# kubectl create secret generic die8 --from-literal=key1=supersecret --from-literal=key2=topsecret

TLS 密码对象，需要提供私钥和证书文件：
[root@master1 ~]# kubectl create secret tls tls-secret --cert=path/to/tls.crt --key=path/to/tls.key

用于向 docker 库 push 镜像的密码对象：
[root@master1 ~]# kubectl create secret docker-registry my-secret --docker-server=DOCKER_REGISTRY_SERVER \
 --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
```

建议对 kubectl 最常见的用法定义别名，例如：

```
[root@master1 ~]# vim ~/.bashrc                                          #在末尾增加下面的行
alias k=kubectl
alias kgp='kubectl get pods -o wide'
alias kd='kubectl describe'
alias kex='kubectl exec -i -t'

[root@master1 ~]# . ~/.bashrc
```

例如，以后就可以执行“kgp -A”显示全部 pod，执行“kex pod-nginx -- sh”进入 pod-nginx 了。

2．删除资源对象

语法：

```
kubectl delete ([-f FILENAME] | [-k DIRECTORY] | TYPE [(NAME | -l label | --all)])
```

可通过配置文件名、stdin、资源名称或 label 选择器来删除资源。

二、crictl/ctr 命令

针对 containerd 容器运行时，不再有 docker 命令，取而代之的是 crictl 和 ctr，ctr 是社区版，它默认存在两个名字空间：k8s.io 和 default，而 crictl 只有 k8s.io 名字空间，也没有指定名字空间的命令行参数。ctr 有参数-n 来指明名字空间，但是 ctr 不会使用 containerd 的配置文件/etc/containerd/config.toml，所以它下拉镜像时还是走官方源，这回导致失败，因此国内还是建议使用 crictl 命令。

1. crictl 命令

管理镜像：crictl pull|images|inspecti|imagefsinfo|rmi ……

```
下拉容器镜像 (crictl pull NAME[:TAG|@DIGEST]):
[root@master1 ~]# crictl pull busybox                                    #下拉最新版，等同于 busybox:latest
cricImage is up to date for sha256:517b897a6a8312ce202a85c8a517d820b0fc5b6f5d14ec2a3267906f75680403


[root@master1 ~]# crictl pull docker.io/flannel/flannel:v0.26.1                                  #下拉特定版本的镜像

列举本地的镜像 (crictl images|image|img):
[root@master1 ~]# crictl images                                                                  #列举已经下拉的镜像
IMAGE                                                TAG                 IMAGE ID            SIZE
docker.io/flannel/flannel                            v0.26.1             4c92107ad4cfe       32.2MB
docker.io/library/busybox                            latest              517b897a6a831       2.17MB
```

管理容器：

crictl start|run|stats|stop|update|rm|ps|create|event|inspect|logs|exec|attach|checkpoint ……

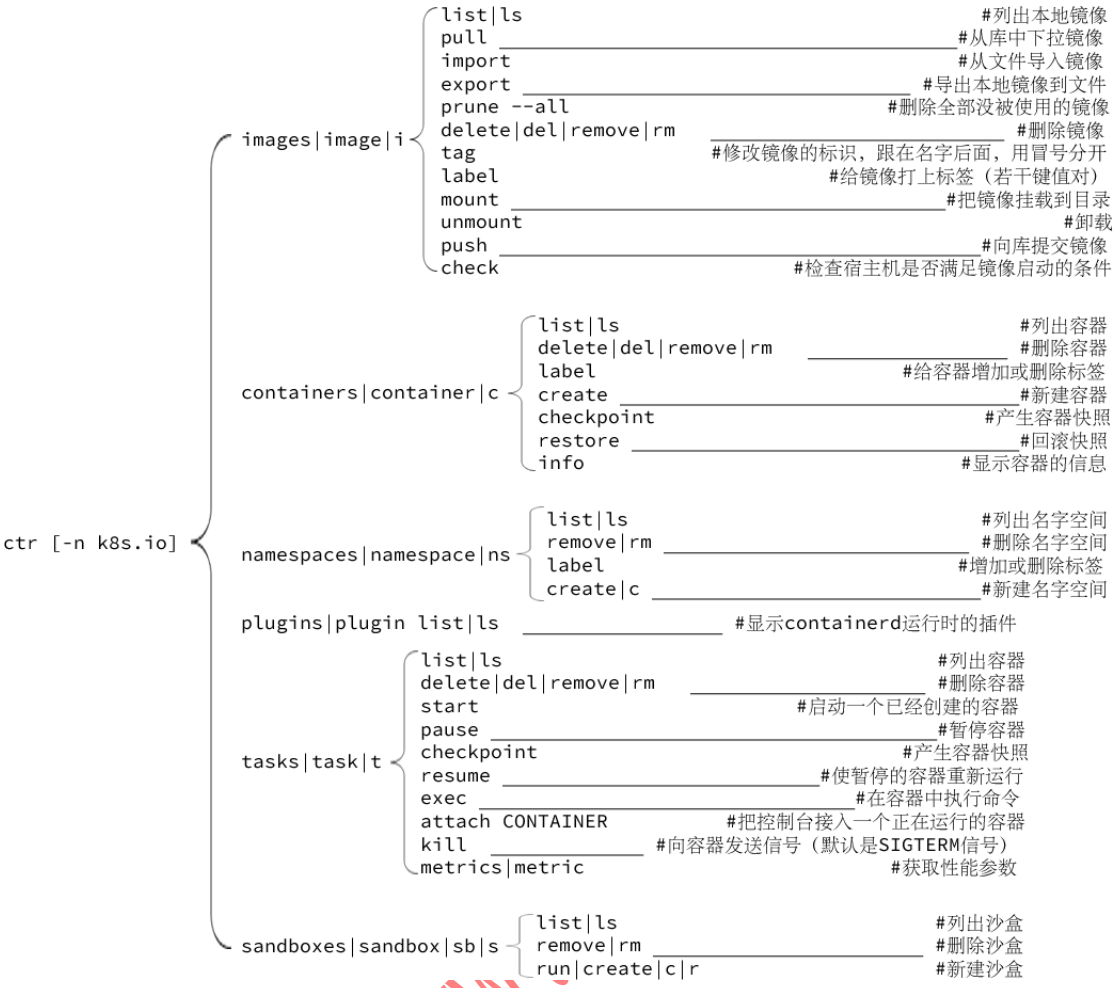
管理 Pod：crictl runp|stopp|rmp|statsp|pods|inspectp|metricsp|port-forward ……

管理其他：crictl config|info|update-runtime-config|version|help ……

2. ctr 命令

语法如下：

https://www.[illegible]en.org



从语法可知 ctr 的功能还是很强的。下面我们通过实例进一步立即 ctr 命令。

1. 管理名字空间

使用 ctr namespaces … 或者 ctr namespace … 或者 ctr ns … 都可以。

```
[root@master1 ~]# ctr ns create sales                                          #新建名字空间 sales
[root@master1 ~]# ctr namespace create netclass name='网络 2 班'          #新建 netclass 名字空间，附加标签 name='网络 2 班'
[root@master1 ~]# ctr ns ls                                                    #显示已经存在的名字空间
NAME    LABELS
default
k8s.io
netclass name=网络 2 班
sales
[root@master1 ~]# ctr ns label sales demo="Sales Department"                   #给 sales 增加标签
[root@master1 ~]# ctr ns label netclass name=""                                #产出标签 name
[root@master1 ~]# ctr ns ls
NAME     LABELS
default
k8s.io
netclass
sales    demo=Sales Department
```

kubernetes 使用 k8s.io 名字空间中的容器镜像，所以 ctr 命令下拉的镜像必须放在 k8s.io 名字空间中，k8s 才用得上。

2. 管理容器镜像

使用 ctr images … 或者 ctr image … 或者 ctr i … 都可以。

```
# 拉镜像:
[root@worker1 ~]# crictl pull docker.io/calico/kube-controllers:v3.24.5

# 很多镜像库在国外，很难拉下来，这时可尝试采用下面的命令:
```

```
[root@master1 ~]# ctr -n k8s.io images pull docker.io/calico/kube-controllers:v3.24.5

# 查看 pod 中的容器（其中 nginx-www-64ffdc9bdd-vjl4q 是 pod 名）:
[root@master1 ~]# kubectl get pod nginx-www-64ffdc9bdd-vjl4q -o jsonpath="{.spec['containers','initContainers'][*].name}"

# 执行容器中的命令（其中 nginx-www-64ffdc9bdd-vjl4q 是 pod 名，nginx 是容器名）:
[root@master1 ~]#  kubectl exec nginx-www-64ffdc9bdd-vjl4q -c nginx -i -t -- ip a
default via 10.244.1.1 dev eth0
10.244.0.0/16 via 10.244.1.1 dev eth0
10.244.1.0/24 dev eth0 scope link  src 10.244.1.20
```

§16.5.2 helm 命令

使用 dnf 命令在 Rocky 操作系统上安装软件，往往一个软件会依赖一些其他软件，dnf 命令会自动安装所有的依赖软件，软件被打包成 rpm 格式并存放在软件库中，安装时自动从软件库下载软件包，然后解包释放里面的文件到磁盘中。

helm 命令就类似 dnf，不过是在 k8s 集群中管理应用——安装、升级、回滚、卸载、打包等。heml 把资源对象的清单文件打包成*.tgz 并存放在 Chart 仓库中。注意：Chart 仓库中不存放容器镜像，只存放了资源对象的清单文件。示意图参考图 16-11。

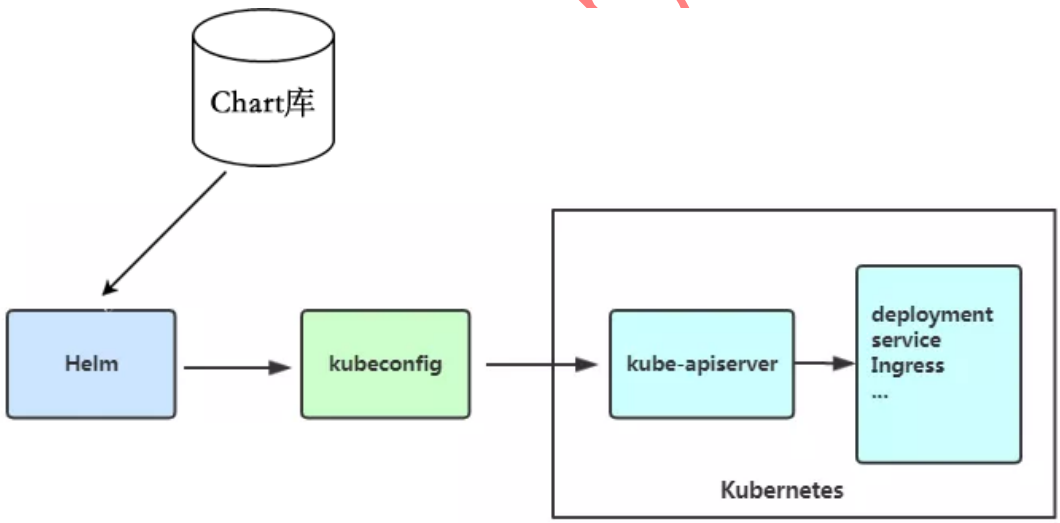


图 16-11 Helm 原理图

比如从 Chart 库拉下的部署 MariaDB 用到的资源清单文件被打包成 mariadb-7.3.14.tgz，解包后的目录如下：

```
[root@master1 ~]# tree mariadb
mariadb
├── Chart.yaml
├── files
│   └── docker-entrypoint-initdb.d
│       └── README.md
├── OWNERS
├── README.md
├── templates
│   ├── _helpers.tpl
│   ├── initialization-configmap.yaml
│   ├── master-configmap.yaml
│   ├── master-pdb.yaml
│   ├── master-statefulset.yaml
│   ├── master-svc.yaml
│   ├── NOTES.txt
│   ├── rolebinding.yaml
│   ├── role.yaml
```

ryopen.org

```
│   ├── secrets.yaml
│   ├── serviceaccount.yaml
│   ├── servicemonitor.yaml
│   ├── slave-configmap.yaml
│   ├── slave-pdb.yaml
│   ├── slave-statefulset.yaml
│   ├── slave-svc.yaml
│   ├── test-runner.yaml
│   └── tests.yaml
├── values-production.yaml
├── values.schema.json
└── values.yaml
```

一、安装 helm 并配置仓库

在 https://github.com/helm/helm/releases 上找到最新版本的下载链接，然后下载并解压：

```
[root@master1 ~]# wget https://get.helm.sh/helm-v4.0.0-linux-amd64.tar.gz
[root@master1 ~]# tar -xf helm-v4.0.0-linux-amd64.tar.gz
[root@master1 ~]# mv linux-amd64/helm /usr/local/bin/
```

Tab 补全：

```
[root@master1 ~]# helm completion bash > /etc/bash_completion.d/helm
[root@master1 ~]# source <(helm completion bash)
```

就是一个静态编译的程序 helm，解压后把它拷贝到/usr/local/bin 目录中即可。

helm 官方 chart 仓库，即 Artifact Hub，官网为 https://artifacthub.io/，可在这里搜索基于 k8s 集群的应用并查看安装的方法，参考图 16-12 和图 16-13 操作。

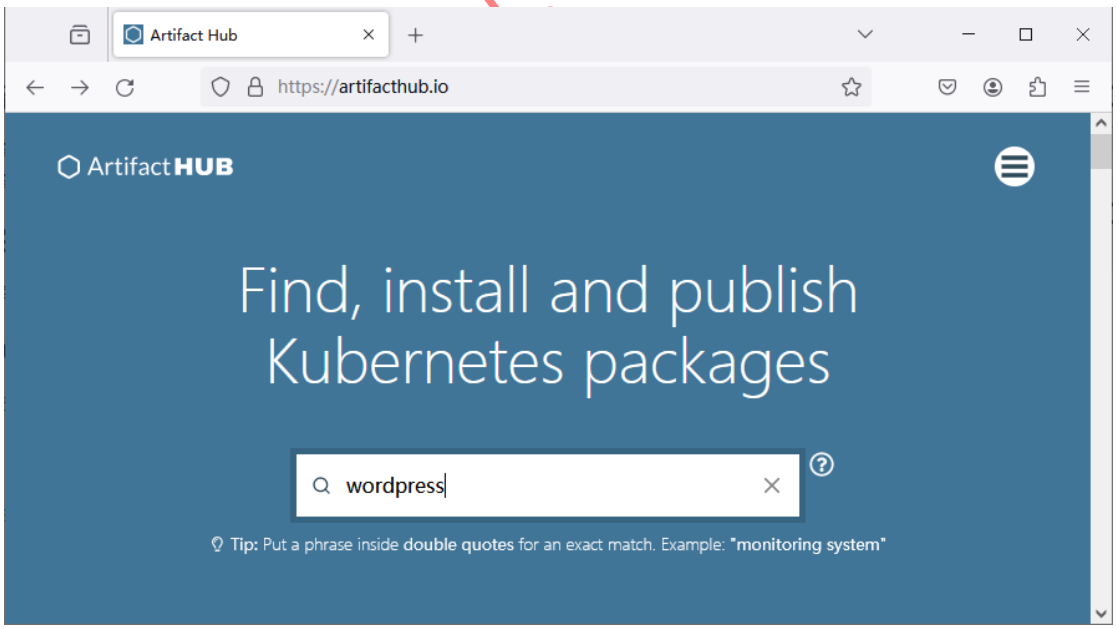


图 16-12 在 helm 库中搜索

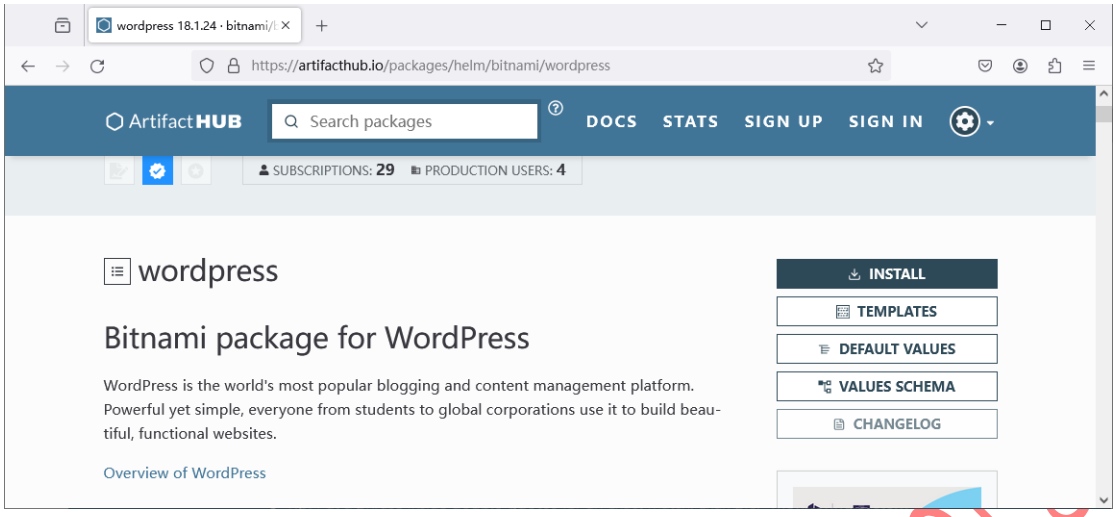


图 16-13 搜索结果

点击上图中的“INSTALL”查看安装的方法参考图 16-14。

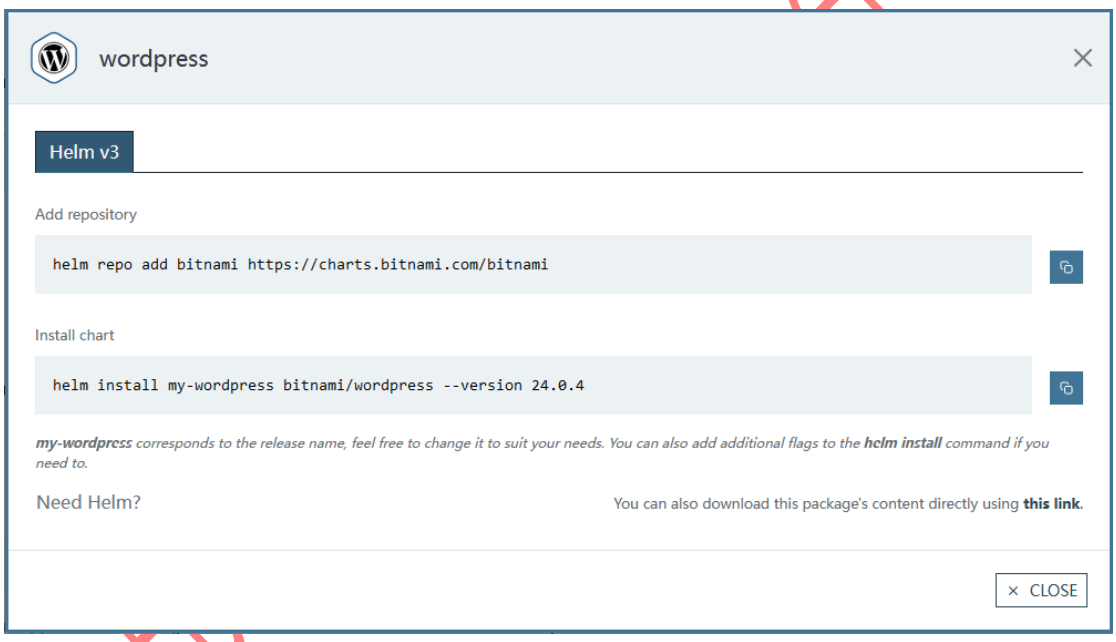


图 16-14 安装方法

告诉我们安装分为两步：首先加入库 bitnami，然后再安装。bitnami 是最流行的官方库，下面先配置好需要的库。

配置 Chart 库（库配置文件为~/.config/helm/repositories.yaml）：

```
helm 官方 Char 库（不过国内访问不稳定）：
[root@master1 ~]# helm repo add bitnami https://charts.bitnami.com/bitnami

国内最好的库：
[root@master1 ~]# helm repo add stable http://mirror.azure.cn/kubernetes/charts

阿里库的 chart 版本往往落后很多：
[root@master1 ~]# helm repo add aliyun https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts

增加本地的私有库：
[root@master1 ~]# helm repo add gdpi http://10.8.20.8:8080

更新本地缓存：
[root@master1 ~]# helm repo update
```

最后一条命令如果报错，不妨多试几遍。

设置环境变量 KUBECONFIG：

```
[root@master1 ~]# echo "export KUBECONFIG=/etc/kubernetes/admin.conf" >> ~/.bashrc
[root@master1 ~]# . ~/.bashrc
[root@master1 ~]# helm repo list                                                    #显示存在哪些库
NAME    URL
bitnami https://charts.bitnami.com/bitnami
stable  http://mirror.azure.cn/kubernetes/charts
aliyun  https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts
```

如果配置文件为~/.kube/config，KUBECONFIG 变量可以省略。

helm 默认的缓存目录 $HOME/.cache/helm，配置目录 $HOME/.config/helm，数据目录 $HOME/.local/share/helm。

二、helm 的基本用法

1. 搜索

语法：

helm search hub <关键字> [--max-col-width uint -o table|json|yaml]
helm search repo <关键字> [-l --devel -o table|json|yaml]

hub 就是在 helm 官方 Artifact Hub 中或自己搭建的 hub 中搜索，repo 在添加到本地的仓库中搜索。参数--max-col-width uint 定义表的列宽，默认为 50，-o talbe|json|yaml 定义输出格式，默认为 table，--devel 搜索开发版，-l 参数列出每个库中的全部版本，否则只列出最新版。

```
[root@master1 ~]# helm search hub wordpress                                         #在 Hub 中搜索 wordpress
URL                                             CHART VERSION   APP VERSION             DESCRIPTION
https://artifacthub.io/packages/helm/wordpress-...      1.0.2           1.0.0           A Helm chart for deploying Wordpress+Mariadb st...
https://artifacthub.io/packages/helm/kube-wordp...      0.1.0           1.1             this is my wordpress package
https://artifacthub.io/packages/helm/bitnami/wo...      18.1.24         6.4.2           WordPress is the world's most popular blogging ...
https://artifacthub.io/packages/helm/shubham-wo...      0.1.0           1.16.0          A Helm chart for Kubernetes
……
```

为了看到完整的 URL，可指定更大的列宽：

```
[root@master1 ~]# helm search hub wordpress --max-col-width 128
URL                                                                             CHART VERSION   APP VERSION     DESCRIPTION
https://artifacthub.io/packages/helm/wordpress-mariadb/wordpress                1.0.2           1.0.0           A Helm chart for deploying Wordpress+Mariadb stack on kubernetes.
https://artifacthub.io/packages/helm/kube-wordpress/wordpress                   0.1.0           1.1             this is my wordpress package
……
```

在仓库中搜索前，建议常更新本地的索引缓存。

```
[root@master1 ~]# helm repo update                                                  #更新本地的索引缓存
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "aliyun" chart repository
...Successfully got an update from the "bitnami" chart repository
Update Complete. ⎈Happy Helming!⎈

[root@master1 ~]# helm search repo wordpress                                        #搜索 wordpress 关键字
NAME                    CHART VERSION   APP VERSION     DESCRIPTION
aliyun/wordpress        0.8.8           4.9.4           Web publishing platform for building blogs and ...
bitnami/wordpress       18.1.24         6.4.2           WordPress is the world's most popular blogging ...
bitnami/wordpress-intel 2.1.31          6.1.1           DEPRECATED WordPress for Intel is the most popu...

[root@master1 ~]# helm show chart gdpi/wordpress                                    #查看 Chart 的详细说明
```

2. 部署应用

语法：

http://www.yeyopen.org

helm install [部署名字] [目标 **chart**] [参数]

“部署名称”随便取名，但只能包含小写字母、数字和-且开头和结尾不能出现“-”，长度小于 53，卸载时需要用到这个部署名称。常用的参数有--atomic（如果失败则回滚。命令会一直等待 pod 正常启动后才返回）、--dry-run[=server]（模拟部署过程，如果没省略“=server”，那么模拟过程需要连接 k8s 集群）、--force（强制更新已经部署的应用）、--labels <name1=value1[,name2=value2,…]>（给部署添加标签）、--repo <仓库的 URL>（从指定的仓库安装）、-version <版本>（指定版本）、-n <名字空间>（部署到指定的名字空间中）。

有六种方式指定目标 chart：

(1).通过 chart 引用，如：helm install mymaria bitnami/mariadb
(2).通过 chart 包，如：helm install mynginx ./nginx-1.2.3.tgz
(3).通过未打包 chart 目录的路径，如：helm install mynginx ./nginx
(4).通过 URL 绝对路径，如：helm install mynginx example.com/charts/ngin…
(5).通过 chart 引用和仓库 url，如：helm install --repo example.com/charts/ mynginx nginx
(6).通过 OCI 注册中心，如：helm install mynginx oci://example.com/charts/nginx

举例：

从 bitnami 仓库部署 wordpress 应用，版本为 20.0.0，部署名字为 my-wordpress。

由于 wordpress 需要用到动态 PV，所以请先确保完成了§16.4.2 小节的“二、动态 PV 实验”中的 1、2、3、4 步，这几步创建了一个名叫 nfs-client 的 StorageClass 资源对象（可用命令 kubectl get storageclass 检查）。

然后再执行下面的命令：

```
[root@master1 ~]# helm pull bitnami/wordpress --version 20.0.0                    #下拉 wordpress 的 Chart 包
[root@master1 ~]# tar -xf wordpress-24.0.0.tgz
[root@master1 ~]# vim wordpress/values.yaml
……
  defaultStorageClass: "nfs-client"
……
```

省略参数--version 20.0.0 就是下拉最新版本，但在国内，最新版没法安装。

```
[root@master1 ~]# helm install my-wordpress ./wordpress                            #部署 wordpress
NAME: my-wordpress
LAST DEPLOYED: Fri Nov 29 14:37:12 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: wordpress
CHART VERSION: 24.0.0
APP VERSION: 6.6.2

** Please be patient while the chart is being deployed **

Your WordPress site can be accessed through the following DNS name from within your cluster:

    my-wordpress.default.svc.cluster.local (port 80)

To access your WordPress site from outside the cluster follow the steps below:

1. Get the WordPress URL by running these commands:

  NOTE: It may take a few minutes for the LoadBalancer IP to be available.
        Watch the status with: 'kubectl get svc --namespace default -w my-wordpress'

   export SERVICE_IP=$(kubectl get svc --namespace default my-wordpress --template "{{ range
(index .status.loadBalancer.ingress 0) }}{{ . }}{{ end }}")
   echo "WordPress URL: http://$SERVICE_IP/"
   echo "WordPress Admin URL: http://$SERVICE_IP/admin"

2. Open a browser and access WordPress using the obtained URL.

3. Login with the following credentials below to see your blog:

  echo Username: user
  echo Password: $(kubectl get secret --namespace default my-wordpress -o jsonpath="{.data.wordpress-password}" |
base64 -d)
```

www.yeryoben.org

```
WARNING: There are "resources" sections in the chart not set. Using "resourcesPreset" is not recommended for
production. For production installations, please set the following values according to your workload needs:
  - resources
+info https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
```

3．列出已经部署的应用

语法：

helm list [参数]

常用的参数有：-A（全部名字空间内的部署）、-d（按部署时间先后排序，越早部署的越排在前面）、-f <正则表达式>（过滤结果）、-m <整数>（显示的最大部署个数）、-r（反排序）、-l <name1=value1[,name2=value2,…]>（标签选择器）、-n <名字空间>（只显示指定名字空间内的部署）。

```
[root@master1 ~]# helm list -d -r
NAME          NAMESPACE       REVISION        UPDATED                                 STATUS          CHART                   APP VERSION
my-wordpress  default         1               2023-12-16 11:12:07.03528701 +0800 CST  deployed        wordpress-18.1.27       6.4.2
redis-cloud   default         1               2023-12-16 11:07:01.233614839 +0800 CST deployed        redis-18.5.0            7.2.3
```

4．查看部署的状态

语法：

heml status [-n <名字空间>]

例如：

```
[root@master1 ~]# helm status my-wordpress
```

5．卸载部署

语法：

helm uninstall|un|delete|del <部署名称> [参数]

常用的参数有--dry-run（模拟卸载过程，并不真正卸载部署）、-n <名字空间>。在生产环境中先采用--dry-run参数比较明智，确认没有误删后再正式卸载部署。

```
[root@master1 ~]# helm uninstall my-wordpress
release "my-wordpress" uninstalled
```

6．下拉 Chart

语法：

helm pull [chart URL | repo/chartname] [参数]

常用的参数有-d <dir1>（下载到 dir1 目录。默认是.）、--untar true（解压解包）、--version <版本>（下拉特定的版本）、--untardir <dir2>（解压解包到 dir1/dir2 目录）。

```
[root@master1 ~]# helm pull bitnami/wordpress                                   #下拉到当前目录下
[root@master1 ~]# helm pull bitnami/wordpress -d /tmp --untar true              #下拉并解压提到/tmp
```

https://www.veryopen.org

7．把 Chart 推送到远程仓库：

语法：

helm push [chart] [remote]

例如推送到自己搭建的仓库：

```
[root@master1 ~]# helm push bookinfo.tgz my-repo
```

8．新建 Chart 骨架：

在当前目录下创建一个 chart 目录结构和必须的文件。语法：

helm create chartname

例如：

```
[root@master1 ~]# helm create erp
Creating erp
[root@master1 ~]# tree erp
erp
├── .helmignore                                                    # 定义 Helm package 打包时要忽略的文件
├── charts/                                                        # 存放本 chart 的依赖的其他 chat
├── Chart.yaml                                                     # chart 的说明文件
├── templates/                                                     #模板文件目录。资源清单文件都放在这里
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── serviceaccount.yaml
│   ├── service.yaml
│   └── tests/                                                     #测试文件目录
│       └── test-connection.yaml
└── values.yaml                                                    #定义模板的默认值
```

把资源对象的清单文件放在 templates 目录下，然后采用 helm install myerp ./erp 试安装。

9．把目录打包成 Chart：

语法：

helm package [chart_path] --app-version string --version string

参数--app-version string 设置 appVersion 的值，参数--version string 指定版本号。例如：

```
[root@master1 ~]# helm package erp --version 1.0
Successfully packaged chart and saved it to: /root/erp-1.0.tgz
```

§16.5.3 K8s Dashboard

需要在 master1 放行端口或者干脆关闭防火墙。

https://www.

官方公布的安装方法极其简单，只需两条命令即可：

```
helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/
helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard --create-namespace --namespace kubernetes-dashboard
```

但是国内没法访问官方的镜像库，原因你懂的，所以在中国，这两条命令没有用。既然政府为我们创造了困难，那我们只有迎难而上了。

首先浏览 https://github.com/kubernetes/dashboard/releases/latest 找到最新版本，截止到 2025 年 11 月是 7.14.0，此页面还告诉我们安装 dashboard 需要用到四个容器镜像：

docker.io/kubernetesui/dashboard-api:1.14.0
docker.io/kubernetesui/dashboard-auth:1.4.0
docker.io/kubernetesui/dashboard-metrics-scraper:1.2.2
docker.io/kubernetesui/dashboard-web:1.7.0

另外还要用到 docker.io/library/kong:3.9 镜像，一共五个容器镜像。

需要手工下拉这些镜像。作为一个应用，Dashboard 跑在工作节点上，所以需要在所有工作节点上下拉镜像即可。下面我们还是在 master1 上下拉，然后导出到工作节点。

手工下来容器镜像：

```
[root@master1 ~]# ctr -n k8s.io images pull docker.rainbond.cc/kubernetesui/dashboard-api:1.14.0
[root@master1 ~]# ctr -n k8s.io images pull docker.rainbond.cc/kubernetesui/dashboard-auth:1.4.0
[root@master1 ~]# ctr -n k8s.io images pull docker.rainbond.cc/kubernetesui/dashboard-metrics-scraper:1.2.2
[root@master1 ~]# ctr -n k8s.io images pull docker.rainbond.cc/kubernetesui/dashboard-web:1.7.0
[root@master1 ~]# ctr -n k8s.io images pull docker.rainbond.cc/library/kong:3.9
```

修改镜像标签（因为 Charts 文件中采用类似 docker.io/*标签）：

```
[root@master1 ~]# ctr -n k8s.io i tag docker.rainbond.cc/kubernetesui/dashboard-api:1.14.0 \
 docker.io/kubernetesui/dashboard-api:1.14.0
[root@master1 ~]# ctr -n k8s.io i tag docker.rainbond.cc/kubernetesui/dashboard-auth:1.4.0 \
 docker.io/kubernetesui/dashboard-auth:1.4.0
[root@master1 ~]# ctr -n k8s.io i tag docker.rainbond.cc/kubernetesui/dashboard-metrics-scraper:1.2.2 \
 docker.io/kubernetesui/dashboard-metrics-scraper:1.2.2
[root@master1 ~]# ctr -n k8s.io i tag docker.rainbond.cc/kubernetesui/dashboard-web:1.7.0 \
 docker.io/kubernetesui/dashboard-web:1.7.0
[root@master1 ~]# ctr -n k8s.io i tag docker.rainbond.cc/library/kong:3.9 docker.io/library/kong:3.9

[root@master1 ~]# crictl images | grep docker.io
IMAGE                                                TAG                 IMAGE ID            SIZE
docker.io/kubernetesui/dashboard-api                 1.14.0              a0607af4fcd8a       16.5MB
docker.io/kubernetesui/dashboard-auth                1.4.0               dd54374d0ab14       14.5MB
docker.io/kubernetesui/dashboard-metrics-scraper     1.2.2               d9cbc9f4053ca       13MB
docker.io/kubernetesui/dashboard-web                 1.7.0               59f642f485d26       62.5MB
docker.io/library/kong                               3.9                 3a975970da2f5       120MB
ghcr.io/flannel-io/flannel-cni-plugin                v1.8.0-flannel1     bb28ded63816e       4.92MB
ghcr.io/flannel-io/flannel                           v0.27.4             e83704a177312       34.1MB
```

导出到文件：

```
[root@master1 ~]# ctr -n k8s.io images export dashboard-api.tar docker.io/kubernetesui/dashboard-api:1.14.0
[root@master1 ~]# ctr -n k8s.io images export dashboard-auth.tar docker.io/kubernetesui/dashboard-auth:1.4.0
[root@master1 ~]# ctr -n k8s.io images export dashboard-metrics-scraper.tar \
 docker.io/kubernetesui/dashboard-metrics-scraper:1.2.2
[root@master1 ~]# ctr -n k8s.io images export dashboard-web.tar docker.io/kubernetesui/dashboard-web:1.7.0
[root@master1 ~]# ctr -n k8s.io images export kong.tar docker.io/library/kong:3.9
```

传递到 worker1、worker2、worker3：

```
[root@master1 ~]# scp dashboard-* worker?:.
[root@master1 ~]# scp kong.tar  worker?:.
```

在工作节点上导入：

```
[root@worker? ~]# ctr -n k8s.io images import dashboard-api.tar
```

www.yeryopen.org

```
[root@worker? ~]# ctr -n k8s.io images import dashboard-auth.tar
[root@worker? ~]# ctr -n k8s.io images import dashboard-metrics-scraper.tar
[root@worker? ~]# ctr -n k8s.io images import dashboard-web.tar
[root@worker? ~]# ctr -n k8s.io images import kong.tar
```

这样就在工作节点上准备好了必需的容器镜像了。下面我们来正式部署 kubernetes-dashboard。

首先下载部署 Dashboard 的 Chart 文件：

```
[root@master1 ~]# wget https://github.com/kubernetes/dashboard/releases/download/kubernetes-dashboard-
7.14.0/kubernetes-dashboard-7.14.0.tgz
```

如果下载失败，请多尝试几次，如果实在不行的话，请把命令种的这个网址拷贝到浏览器中打开并下载 kubernetes-dashboard-7.14.0.tgz。

部署 dashboard（先参考 16.5.2 小节安装 helm）：

```
[root@master1 ~]# helm upgrade --install kubernetes-dashboard ./kubernetes-dashboard-7.14.0.tgz \
  --create-namespace --namespace kubernetes-dashboard
Release "kubernetes-dashboard" does not exist. Installing it now.
NAME: kubernetes-dashboard
LAST DEPLOYED: Sat Nov 22 11:13:44 2025
NAMESPACE: kubernetes-dashboard
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
**********************************************************************************************
*** PLEASE BE PATIENT: Kubernetes Dashboard may need a few minutes to get up and become ready ***
**********************************************************************************************

Congratulations! You have just installed Kubernetes Dashboard in your cluster.

To access Dashboard run:
  kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443

NOTE: In case port-forward command does not work, make sure that kong service name is correct.
      Check the services in Kubernetes Dashboard namespace using:
        kubectl -n kubernetes-dashboard get svc

Dashboard will be available at:
  https://localhost:8443
```

命令的输出告诉我们如何访问 Dashboard 页面，即先运行命令“kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443”把 SVC 的 443 端口映射到节点的 8443，然后用浏览器打开网站 https://localhost:8443 即可访问。

这样过几秒钟后检查 pod 的状态，一般就正常了：

```
[root@master1 ~]# kubectl get pods -o wide -n kubernetes-dashboard
NAME                                                READY   STATUS    RESTARTS   AGE   IP           NODE      NOMINATED NODE
READINESS GATES
kubernetes-dashboard-api-f7985cb95-gcfjn            1/1     Running   0          35m   10.244.1.4   worker1   <none>           <none>
kubernetes-dashboard-auth-7c486d897c-nqwkz          1/1     Running   0          73m   10.244.3.2   worker3   <none>           <none>
kubernetes-dashboard-kong-57d45c4f69-pb68l          1/1     Running   0          38m   10.244.3.3   worker3   <none>           <none>
kubernetes-dashboard-metrics-scraper-df869c886-pfxrp   1/1     Running   0          73m   10.244.1.2   worker1   <none>           <none>
kubernetes-dashboard-web-6ccf8d967-kmw6g            1/1     Running   0          73m   10.244.2.3   worker2   <none>           <none>
```

如果自动下拉镜像失败导致 pod 异常，请参考上面加了中划线部分手工下拉镜像，或者参考§23.10 排错。

再查看 SVC:

```
[root@master1 ~]# kubectl get svc -n kubernetes-dashboard
NAME                                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes-dashboard-api            ClusterIP   10.96.235.59     <none>        8000/TCP   87m
kubernetes-dashboard-auth           ClusterIP   10.96.147.238    <none>        8000/TCP   87m
kubernetes-dashboard-kong-proxy     ClusterIP   10.96.223.77     <none>        443/TCP    87m
kubernetes-dashboard-metrics-scraper   ClusterIP   10.96.81.121     <none>        8000/TCP   87m
kubernetes-dashboard-web            ClusterIP   10.96.214.39     <none>        8000/TCP   87m
```

2. 把 SVC “kubernetes-dashboard-kong-proxy”的类型 ClusterIP 改为 NotePort，这样就可以通过任何节点的 Ip 访问页面了：

```
[root@master1 ~]# kubectl patch svc kubernetes-dashboard-kong-proxy -p '{"spec":{"type":"NodePort"}}' \
  -n kubernetes-dashboard
service/kubernetes-dashboard patched
```

或者采用“kubectl -n kubernetes-dashboard edit service kubernetes-dashboard-kong-proxy”全屏查找并修改 type 的值为 NodePort。

查看分配的端口：

```
[root@master1 ~]# kubectl -n kubernetes-dashboard get svc
NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)         AGE
kubernetes-dashboard-api            ClusterIP   10.96.235.59    <none>        8000/TCP        125m
kubernetes-dashboard-auth           ClusterIP   10.96.147.238   <none>        8000/TCP        125m
kubernetes-dashboard-kong-proxy     NodePort    10.96.223.77    <none>        443:32445/TCP   125m
kubernetes-dashboard-metrics-scraper ClusterIP  10.96.81.121    <none>        8000/TCP        125m
kubernetes-dashboard-web            ClusterIP   10.96.214.39    <none>        8000/TCP        125m
```

现在在其他电脑上的浏览器中打开 https://<任何节点的 ip>:32445，会提示存在风险，不管它，继续访问，会出现图 16-15 所示的登录页面，如果出错，请检查两个 coredns…的 pod 是否运行在不同的节点上（如果运行在同一个节点上，请删除一个 pod，让系统自动调度一次）。

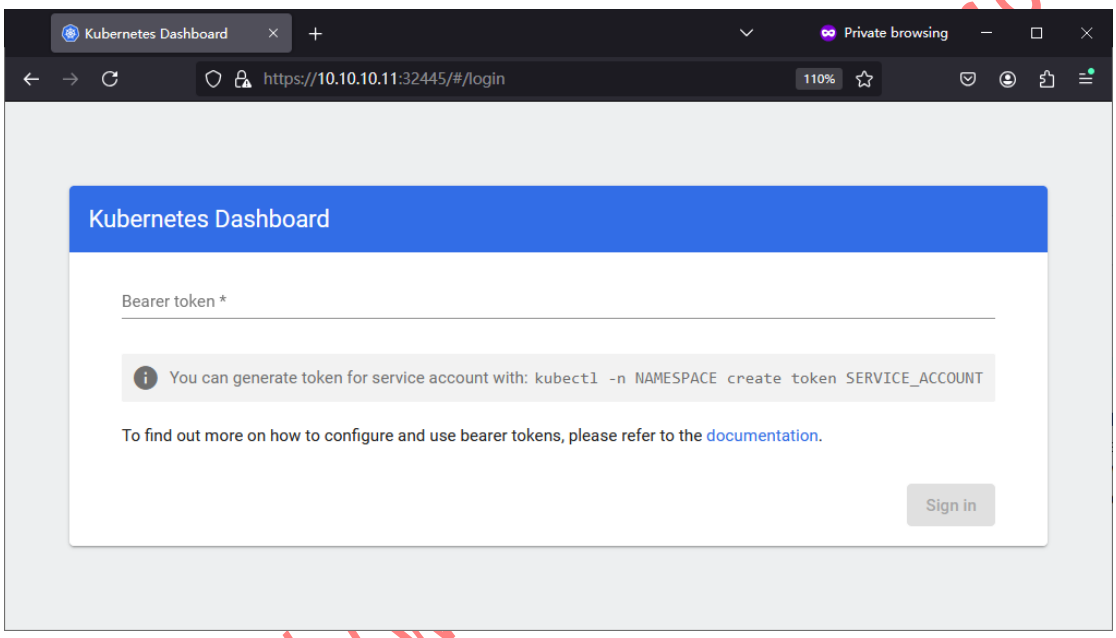


图 16-15 dashboard 的登陆页面

dashboard 的最新版本只能采用令牌登陆。

3. 创建服务账号并授权：

```
创建服务账号 dashboard-admin:
[root@master1 ~]# kubectl create serviceaccount dashboard-admin -n kubernetes-dashboard
serviceaccount/dashboard-admin created

赋予 dashboard-admin 用户全局管理员角色 cluster-admin:
[root@master1 ~]# kubectl create clusterrolebinding dashboard-cluster-admin --clusterrole=cluster-admin \
 --serviceaccount=kubernetes-dashboard:dashboard-admin
clusterrolebinding.rbac.authorization.k8s.io/dashboard-cluster-admin created

为 dashboard-admin 用户创建类型为服务账号的 Secret:
[root@master1 ~]# vim secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: dashboard-admin-secret
  namespace: kubernetes-dashboard
  annotations:
    kubernetes.io/service-account.name: dashboard-admin
type: kubernetes.io/service-account-token

[root@master1 ~]# kubectl create -f secret.yaml
secret/dashboard-admin-secret created
```

4. 获取令牌（token）：

```
[root@master1 ~]# kubectl get secret -n kubernetes-dashboard dashboard-admin-secret -o jsonpath={.data.token} | base64 -d
eyJhbGciOiJSUzI1……uhway84f3A[root@master1 ~]#
```

获取到的 token 就是 eyJhbGciOiJSUzI1……uhway84f3A，把它拷贝粘贴到图 16-15 所示的登陆界面中即可登录。以后每次登陆都要使用 token，建议拷贝出来做好备份，否则每次都得运行上面活得令牌。

~~如果想采用第二种登录方法，即 Kubeconfig 配置文件，那么继续做下一步：~~

~~4. 配置 Kubeconfig 文件：~~

```
[root@master1 ~]# DASH_TOKEN=$(kubectl get secret -n kubernetes-dashboard dashboard-admin-secret -o jsonpath={.data.token}|base64 -d)
[root@master1 ~]# kubectl cluster-info dump|grep cluster-name                                                    #查看集群名称
                            "--cluster-name=k8scluster",

[root@master1 ~]# kubectl config set-cluster k8scluster --server=10.10.10.11:6443 --kubeconfig=/root/dashboard-admin.conf
Cluster "k8scluster" set.

[root@master1 ~]# kubectl config set-credentials dashboard-admin --token=$DASH_TOKEN \
--kubeconfig=/root/dashboard-admin.conf
User "dashboard-admin" set.

[root@master1 ~]# kubectl config set-context dashboard-admin@k8scluster --cluster=k8scluster \
--user=dashboard-admin --kubeconfig=/root/dashboard-admin.conf
Context "dashboard-admin@k8scluster" created.

[root@master1 ~]# kubectl config use-context dashboard-admin@k8scluster --kubeconfig=~/dashboard-admin.conf
Switched to context "dashboard-admin@k8scluster".
```

~~最后把/root/dashboard-admin.conf 文件拷贝到浏览器所在的电脑上，然后就可以采用此文件登录了。~~

登录后的成功登陆后的页面见图 16-16 所示。

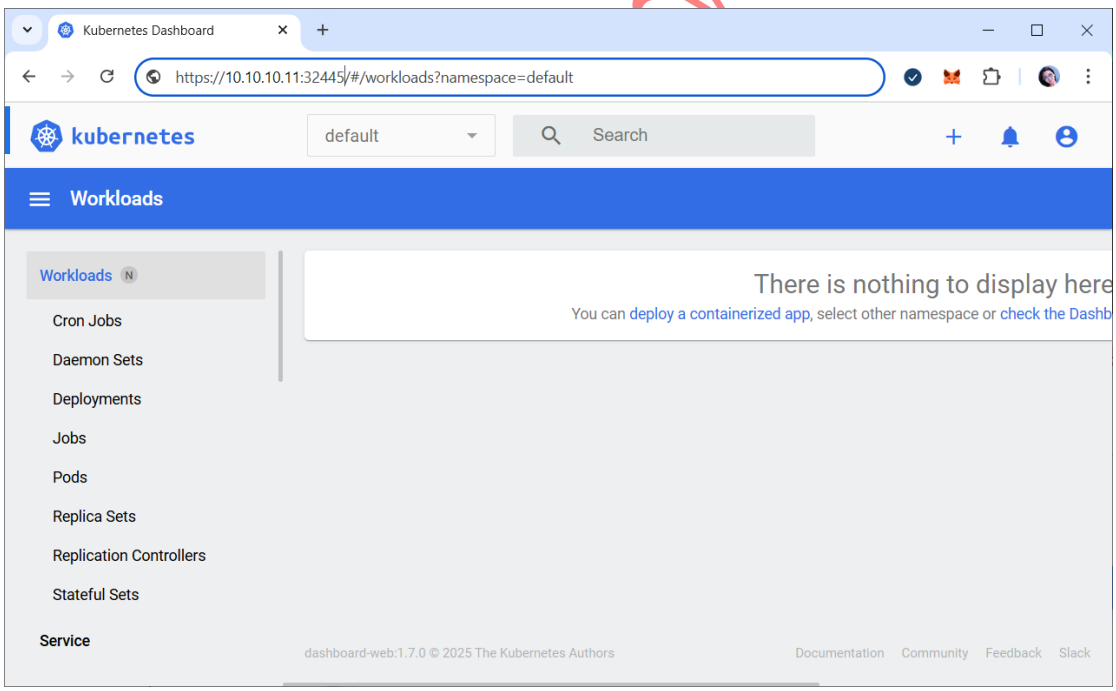


图 16-16 dashboard 登录后的页面

可以汉化，左侧滚到底部，点击 Settings，右边滚到底部，语言里面选择“Chinese Simplified”。

可以采用命令“helm uninstall kubernetes-dashboard -n kubernetes-dashboard”卸载图形面板。

§16.5.4 K8S 集群运维

一、Prometheus-Stack 介绍

kube-prometheus-stack 是 Kubernetes 社区中常用的监控解决方案，翻译成中文就是“普罗米修斯全家桶”，它整合了 Prometheus、Grafana、Alertmanager 和其他相关软件，为 Kubernetes 集群提供全面的监控和告警功能。Prometheus 用 go 语言编写，完全开源，官网 https://prometheus.io/。中文学习资料 https://www.prometheus.wang/。

Prometheus 的架构参考图 16-17。

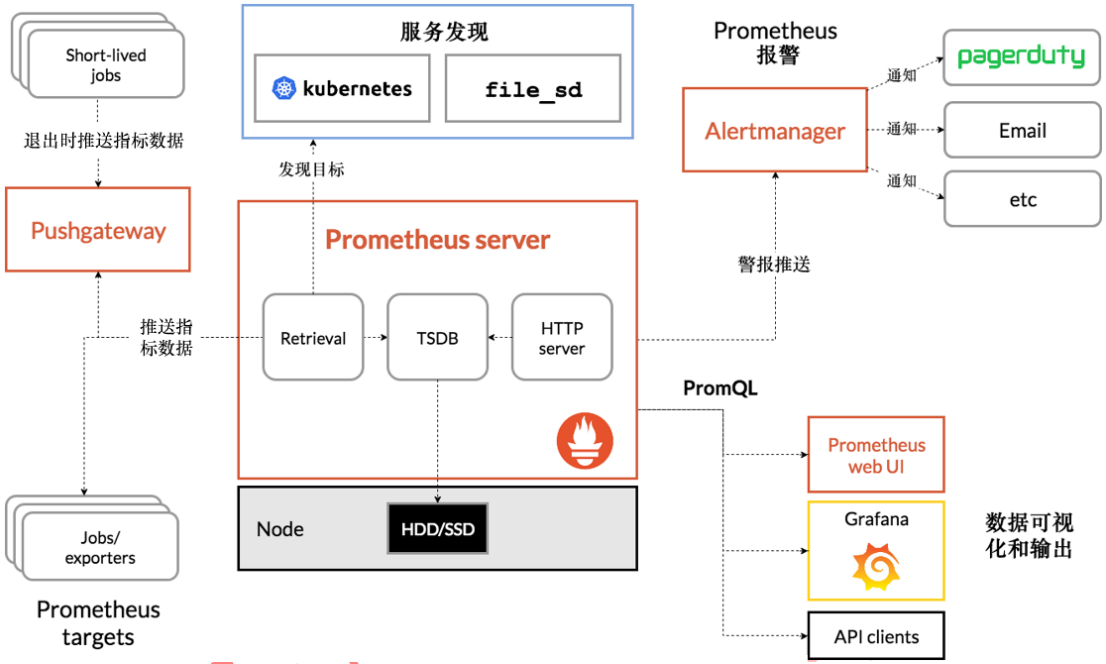


图 16-17 用 Prometheus 构建的运维架构

kube-prometheus-stack 主要由以下几个组件构成：

Prometheus：用于收集、存储和处理集群的各种监控数据（时序数据）。
TSDB：时间序列数据库，OpenTSDB 时序数据库就是这样一种可以满足这种需求的时序数据库。
PromQL： Prometheus 以 PromQL 的形式定义了一种丰富的查询语言，用于从时序数据库中查询数据。
Pushgateway：可用于支持短期工作。
Grafana：提供一个可视化界面，方便用户查看和分析 Prometheus 中的数据。
Alertmanager：根据 Prometheus 中的规则生成告警，并通过邮件、Slack 等方式通知用户。
Node Exporter：采集宿主机的指标数据。
其他组件：如 Blackbox Exporter 等，用于收集节点和服务的监控数据。

二、部署 Prometheus-Stack

1．安装

如果你的电脑能畅通访问外网，那么旧使用方法 1，否则请使用方法 2。

方法 1：

```
[root@master1 ~]# helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
[root@master1 ~]# helm repo update
[root@master1 ~]# helm install prometheus-community/kube-prometheus-stack --generate-name
```

方法 1 结束线

方法 2：

从 https://github.com/prometheus-operator/kube-prometheus 网站上下载 kube-prometheus-0.18.0.zip，然后执行：

```
[root@master1 ~]# dnf install unzip -y                                    #安装解压缩命令
[root@master1 ~]# unzip kube-prometheus-0.18.0.zip                        #解压缩
```

也可以使用命令“git clone https://github.com/prometheus-operator/kube-prometheus.git kube-prometheus-main”克隆代码到 kube-prometheus-main 目录。

kube-prometheus-main 目录下的 README.md 和子目录 docs 中有一些重要文档，建议先好好看看。

kube-prometheus-0.18.0 需要用到如下的容器镜像：

```
docker.io/grafana/grafana:13.0.2
ghcr.io/jimmidyson/configmap-reload:v0.15.0
quay.io/brancz/kube-rbac-proxy:v0.22.0
quay.io/prometheus-operator/prometheus-config-reloader:v0.78.2
quay.io/prometheus-operator/prometheus-operator:v0.92.0
quay.io/prometheus/alertmanager:v0.33.0
quay.io/prometheus/blackbox-exporter:v0.28.0
quay.io/prometheus/node-exporter:v1.11.1
quay.io/prometheus/prometheus:v3.12.0
registry.k8s.io/kube-state-metrics/kube-state-metrics:v2.19.0
registry.k8s.io/prometheus-adapter/prometheus-adapter:v0.12.0
```

master1 上使用 configmap-reload、kube-rbac-proxy、node-exporter 三个镜像，工作节点需要使用全部镜像。如果不能从官方镜像库下拉，请使用其他方法下拉并导入到节点上（比如我提供的 prometheus_images.tar.xz，已经放在各位电脑的桌面上，请上传到 master1 中，然后使用命令 tar -xf prometheus_images.tar.xz 解压解包后产生目录 prometheus，把此目录传递到工作节点【使用命令 scp -r prometheus worker?:.】，然后在工作节点上导入所有镜像，master1 只导入 configmap-reload、kube-rbac-proxy、node-exporter 即可。导入命令 ctr -n k8s.io images import prometheus/文件名，例如 ctr -n k8s.io images import prometheus/alertmanager.tar）。

部署 Prometheus 服务器：

```
[root@master1 ~]# cd kube-prometheus-main
[root@master1 kube-prometheus-main]# kubectl apply --server-side -f manifests/setup --force-conflicts
customresourcedefinition.apiextensions.k8s.io/alertmanagerconfigs.monitoring.coreos.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/alertmanagers.monitoring.coreos.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/podmonitors.monitoring.coreos.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/probes.monitoring.coreos.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/prometheuses.monitoring.coreos.com serverside-applied
```

https://www.veryopen.org

```
customresourcedefinition.apiextensions.k8s.io/prometheusagents.monitoring.coreos.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/prometheusrules.monitoring.coreos.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/scrapeconfigs.monitoring.coreos.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/servicemonitors.monitoring.coreos.com serverside-applied
customresourcedefinition.apiextensions.k8s.io/thanosrulers.monitoring.coreos.com serverside-applied
namespace/monitoring serverside-applied
```

等待较长时间后检查资源就绪状况：

```
[root@master1 kube-prometheus-main]# kubectl wait --for condition=Established --all CustomResourceDefinition \
  --namespace=monitoring
customresourcedefinition.apiextensions.k8s.io/alertmanagerconfigs.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/alertmanagers.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/ingressclassparameterses.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/kongclusterplugins.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/kongconsumergroups.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/kongconsumers.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/kongingresses.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/konglicenses.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/kongplugins.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/kongupstreampolicies.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/kongvaults.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/podmonitors.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/probes.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/prometheusagents.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/prometheuses.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/prometheusrules.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/scrapeconfigs.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/servicemonitors.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/tcpingresses.configuration.konghq.com condition met
customresourcedefinition.apiextensions.k8s.io/thanosrulers.monitoring.coreos.com condition met
customresourcedefinition.apiextensions.k8s.io/udpingresses.configuration.konghq.com condition met
```

Prometheus 的全部资源在 monitoring 名字空间内。注意：要等全部资源就绪后（condition met）再往下操作。

创建其他资源对象：

```
[root@master1 kube-prometheus-main]# kubectl apply -f manifests/
alertmanager.monitoring.coreos.com/main created
networkpolicy.networking.k8s.io/alertmanager-main created
poddisruptionbudget.policy/alertmanager-main created
prometheusrule.monitoring.coreos.com/alertmanager-main-rules created
secret/alertmanager-main created
service/alertmanager-main created
serviceaccount/alertmanager-main created
servicemonitor.monitoring.coreos.com/alertmanager-main created
clusterrole.rbac.authorization.k8s.io/blackbox-exporter created
clusterrolebinding.rbac.authorization.k8s.io/blackbox-exporter created
configmap/blackbox-exporter-configuration created
deployment.apps/blackbox-exporter created
networkpolicy.networking.k8s.io/blackbox-exporter created
service/blackbox-exporter created
serviceaccount/blackbox-exporter created
servicemonitor.monitoring.coreos.com/blackbox-exporter created
secret/grafana-config created
……
service/prometheus-adapter created
serviceaccount/prometheus-adapter created
servicemonitor.monitoring.coreos.com/prometheus-adapter created
clusterrole.rbac.authorization.k8s.io/prometheus-operator created
clusterrolebinding.rbac.authorization.k8s.io/prometheus-operator created
deployment.apps/prometheus-operator created
networkpolicy.networking.k8s.io/prometheus-operator created
prometheusrule.monitoring.coreos.com/prometheus-operator-rules created
service/prometheus-operator created
serviceaccount/prometheus-operator created
servicemonitor.monitoring.coreos.com/prometheus-operator created
```

卸载可采用命令 kubectl delete --ignore-not-found=true -f manifests/ -f manifests/setup。

方法 2 结束线

2．验证安装是否成功

查看全部资源是否就绪：

```
[root@master1 ~]# kubectl get all -n monitoring
NAME                                   READY   STATUS              RESTARTS   AGE
pod/blackbox-exporter-d7779b7d4-nsm9m  0/3     ContainerCreating   0          19s
pod/grafana-fdccb4564-75wt6            0/1     ContainerCreating   0          18s
```

```
pod/kube-state-metrics-74f55cf6d9-drkml    0/3     ContainerCreating   0          18s
pod/node-exporter-dschc                    0/2     ContainerCreating   0          18s
pod/node-exporter-gjl7d                    0/2     ContainerCreating   0          18s
pod/node-exporter-nfjq2                    0/2     ContainerCreating   0          18s
pod/node-exporter-qlfpv                    2/2     Running             0          18s
pod/prometheus-adapter-784f566c54-7cndm    0/1     ContainerCreating   0          16s
pod/prometheus-adapter-784f566c54-tjtmt    0/1     ContainerCreating   0          16s
pod/prometheus-operator-6ddf9d5c64-jb54r   0/2     ContainerCreating   0          16s

NAME                            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)             AGE
service/alertmanager-main       ClusterIP   10.96.99.21     <none>        9093/TCP,8080/TCP   19s
service/blackbox-exporter       ClusterIP   10.96.30.4      <none>        9115/TCP,19115/TCP  19s
service/grafana                 ClusterIP   10.96.112.193   <none>        3000/TCP            18s
service/kube-state-metrics      ClusterIP   None            <none>        8443/TCP,9443/TCP   18s
service/node-exporter           ClusterIP   None            <none>        9100/TCP            18s
service/prometheus-adapter      ClusterIP   10.96.205.24    <none>        443/TCP             16s
service/prometheus-k8s          ClusterIP   10.96.15.234    <none>        9090/TCP,8080/TCP   17s
service/prometheus-operator     ClusterIP   None            <none>        8443/TCP            16s

NAME                             DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/node-exporter     4         4         1       4            1           kubernetes.io/os=linux   18s

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/blackbox-exporter     0/1     1            0           19s
deployment.apps/grafana               0/1     1            0           18s
deployment.apps/kube-state-metrics    0/1     1            0           18s
deployment.apps/prometheus-adapter    0/2     2            0           17s
deployment.apps/prometheus-operator   0/1     1            0           16s

NAME                                             DESIRED   CURRENT   READY   AGE
replicaset.apps/blackbox-exporter-d7779b7d4      1         1         0       19s
replicaset.apps/grafana-fdccb4564                1         1         0       18s
replicaset.apps/kube-state-metrics-74f55cf6d9    1         1         0       18s
replicaset.apps/prometheus-adapter-784f566c54    2         2         0       17s
replicaset.apps/prometheus-operator-6ddf9d5c64   1         1         0       16s
```

发现局部资源（在 monitoring 名字空间内）类型有 Pod、Deployment、Service、Daemonset 和 Replicaset，显然大部分资源还没准备好（READY 列等于 1 才是就绪）。

过几分钟再次检查资源是否就绪：

```
[root@master1 ~]# kubectl get all -n monitoring
NAME                                       READY   STATUS             RESTARTS   AGE
pod/alertmanager-main-0                    2/2     Running            0          54m
pod/alertmanager-main-1                    2/2     Running            0          54m
pod/alertmanager-main-2                    2/2     Running            0          54m
pod/blackbox-exporter-d7779b7d4-nsm9m      3/3     Running            0          57m
pod/grafana-fdccb4564-75wt6                1/1     Running            0          57m
pod/kube-state-metrics-74f55cf6d9-drkml    2/3     ImagePullBackOff   0          57m
pod/node-exporter-dschc                    2/2     Running            0          57m
pod/node-exporter-gjl7d                    2/2     Running            0          57m
pod/node-exporter-nfjq2                    2/2     Running            0          57m
pod/node-exporter-qlfpv                    2/2     Running            0          57m
pod/prometheus-adapter-784f566c54-7cndm    0/1     ImagePullBackOff   0          56m
pod/prometheus-adapter-784f566c54-tjtmt    0/1     ImagePullBackOff   0          56m
pod/prometheus-k8s-0                       2/2     Running            0          54m
pod/prometheus-k8s-1                       2/2     Running            0          54m
pod/prometheus-operator-6ddf9d5c64-jb54r   2/2     Running            0          56m

NAME                            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                        AGE
service/alertmanager-main       ClusterIP   10.96.99.21     <none>        9093/TCP,8080/TCP              57m
service/alertmanager-operated   ClusterIP   None            <none>        9093/TCP,9094/TCP,9094/UDP     54m
service/blackbox-exporter       ClusterIP   10.96.30.4      <none>        9115/TCP,19115/TCP             57m
service/grafana                 NodePort    10.96.112.193   <none>        3000:30080/TCP                 57m
service/kube-state-metrics      ClusterIP   None            <none>        8443/TCP,9443/TCP              57m
service/node-exporter           ClusterIP   None            <none>        9100/TCP                       57m
service/prometheus-adapter      ClusterIP   10.96.205.24    <none>        443/TCP                        56m
service/prometheus-k8s          NodePort    10.96.15.234    <none>        9090:30081/TCP,8080:32472/TCP  57m
service/prometheus-operated     ClusterIP   None            <none>        9090/TCP                       54m
service/prometheus-operator     ClusterIP   None            <none>        8443/TCP                       56m

NAME                             DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/node-exporter     4         4         4       4            4           kubernetes.io/os=linux   57m

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/blackbox-exporter     1/1     1            1           57m
deployment.apps/grafana               1/1     1            1           57m
deployment.apps/kube-state-metrics    0/1     1            0           57m
deployment.apps/prometheus-adapter    0/2     2            0           57m
deployment.apps/prometheus-operator   1/1     1            1           56m

NAME                                             DESIRED   CURRENT   READY   AGE
replicaset.apps/blackbox-exporter-d7779b7d4      1         1         1       57m
replicaset.apps/grafana-fdccb4564                1         1         1       57m
replicaset.apps/kube-state-metrics-74f55cf6d9    1         1         0       57m
replicaset.apps/prometheus-adapter-784f566c54    2         2         0       57m
replicaset.apps/prometheus-operator-6ddf9d5c64   1         1         1       56m
```

```
NAME                                  READY   AGE
statefulset.apps/alertmanager-main    3/3     54m


statefulset.apps/prometheus-k8s       2/2     54m
```

黄色字体部分的资源还没准备好。首先查看异常 pod 的事件：

```
[root@master1 ~]# kubectl describe pod prometheus-adapter-784f566c54-7cndm -n monitoring
……
  Warning  Failed   60m (x3 over 63m)     kubelet  Failed to pull image "registry.k8s.io/prometheus-adapter/prometheus-adapter:v0.12.0"…
……
```

即下拉镜像 registry.k8s.io/prometheus-adapter/prometheus-adapter:v0.12.0 失败，请回到 § 14.3 小节检查文件/etc/containerd/certs.d/registry.k8s.io/hosts.toml 中配置的国内源是否可用。

再次检查各种资源是否就绪：

```
[root@master1 ~]# kubectl get all -n monitoring
NAME                                       READY   STATUS    RESTARTS   AGE
pod/alertmanager-main-0                    2/2     Running   0          96m
pod/alertmanager-main-1                    2/2     Running   0          96m
pod/alertmanager-main-2                    2/2     Running   0          96m
pod/blackbox-exporter-d7779b7d4-nsm9m      3/3     Running   0          99m
pod/grafana-fdccb4564-75wt6                1/1     Running   0          99m
pod/kube-state-metrics-74f55cf6d9-drkml    3/3     Running   0          99m
pod/node-exporter-dschc                    2/2     Running   0          99m
pod/node-exporter-gjl7d                    2/2     Running   0          99m
pod/node-exporter-nfjq2                    2/2     Running   0          99m
pod/node-exporter-qlfpv                    2/2     Running   0          99m
pod/prometheus-adapter-784f566c54-7qdqj    1/1     Running   0          13m
pod/prometheus-adapter-784f566c54-zrztv    1/1     Running   0          3m47s
pod/prometheus-k8s-0                       2/2     Running   0          96m
pod/prometheus-k8s-1                       2/2     Running   0          96m
pod/prometheus-operator-6ddf9d5c64-jb54r   2/2     Running   0          99m

NAME                            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
service/alertmanager-main       ClusterIP   10.96.99.21     <none>        9093/TCP,8080/TCP            99m
service/alertmanager-operated   ClusterIP   None            <none>        9093/TCP,9094/TCP,9094/UDP   96m
service/blackbox-exporter       ClusterIP   10.96.30.4      <none>        9115/TCP,19115/TCP           99m
service/grafana                 NodePort    10.96.112.193   <none>        3000:30080/TCP               99m
service/kube-state-metrics      ClusterIP   None            <none>        8443/TCP,9443/TCP            99m
service/node-exporter           ClusterIP   None            <none>        9100/TCP                     99m
service/prometheus-adapter      ClusterIP   10.96.205.24    <none>        443/TCP                      99m
service/prometheus-k8s          NodePort    10.96.15.234    <none>        9090:30081/TCP,8080:32472/TCP   99m
service/prometheus-operated     ClusterIP   None            <none>        9090/TCP                     96m
service/prometheus-operator     ClusterIP   None            <none>        8443/TCP                     99m

NAME                             DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/node-exporter     4         4         4       4            4           kubernetes.io/os=linux   99m

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/blackbox-exporter     1/1     1            1           99m
deployment.apps/grafana               1/1     1            1           99m
deployment.apps/kube-state-metrics    1/1     1            1           99m
deployment.apps/prometheus-adapter    2/2     2            2           99m
deployment.apps/prometheus-operator   1/1     1            1           99m

NAME                                             DESIRED   CURRENT   READY   AGE
replicaset.apps/blackbox-exporter-d7779b7d4      1         1         1       99m
replicaset.apps/grafana-fdccb4564                1         1         1       99m
replicaset.apps/kube-state-metrics-74f55cf6d9    1         1         1       99m
replicaset.apps/prometheus-adapter-784f566c54    2         2         2       99m
replicaset.apps/prometheus-operator-6ddf9d5c64   1         1         1       99m

NAME                                  READY   AGE
statefulset.apps/alertmanager-main    3/3     96m
statefulset.apps/prometheus-k8s       2/2     96m
```

一切都 ok 了。

3. 修改服务资源的类型

如上安装完成所有 service 都是 ClusterIP 类型，所有资源只能集群内部相互访问。

Grafana 页面

下面修改 grafana 服务类型为 NodePort 类型、端口为 30080，这样就能从外面访问：

```
[root@master1 ~]# kubectl patch svc grafana -p '{"spec":{"type":"NodePort"}}' -n monitoring
service/grafana patched
[root@master1 ~]# kubectl patch svc grafana --type='json' -n monitoring \
  -p='[{"op":"replace","path":"/spec/ports/0/nodePort","value":30080}]'
service/ grafana patched
```

当然也可以采用命令 kubectl -n monitoring edit service grafana 全文修改（实际上使用 vim 编辑器）。

访问测试 http://10.10.10.11:30080/ 默认用户名和密码都是 admin。初始登陆会提示修改密码。登陆后看到的页面建图 16-18 所示。

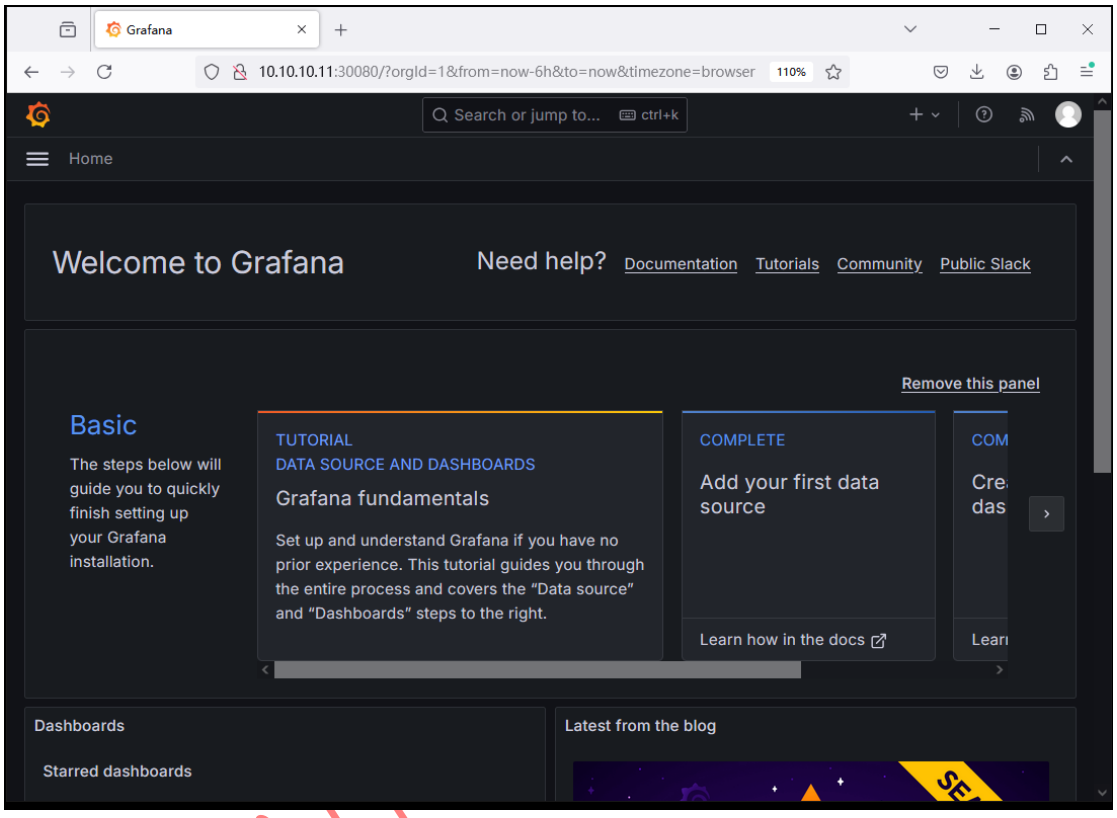


图 16-18 Prometheus 的可视化页面

点击右上角的头像，然后点击 Profile，最后在 language 下选择“中文（简体）”汉化界面。点击右侧的“仪表板”，再选择 Default 下的 Kubernetes/API server，看看 API Server 组件的状态，参见图 16-19。

https://

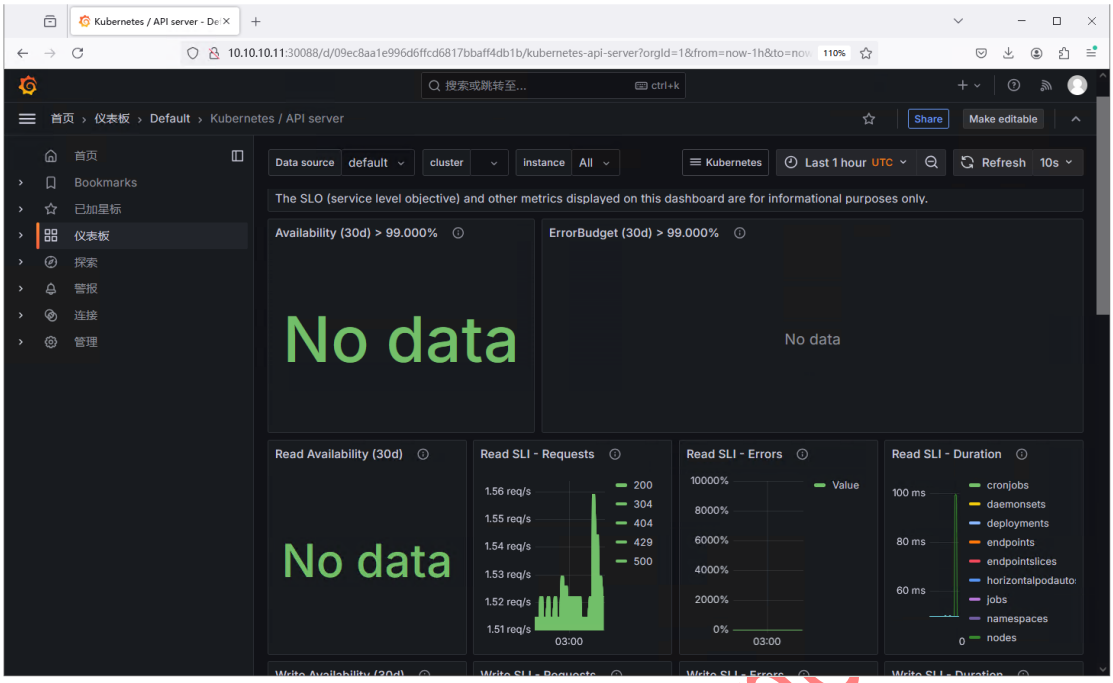


图 16-19 API Server 状态

类似的，再选择 Default 下的 Node Exporter/Nodes 查看各个节点计算资源（cpu、内存、磁盘等）的使用情况，参考图 16-20。

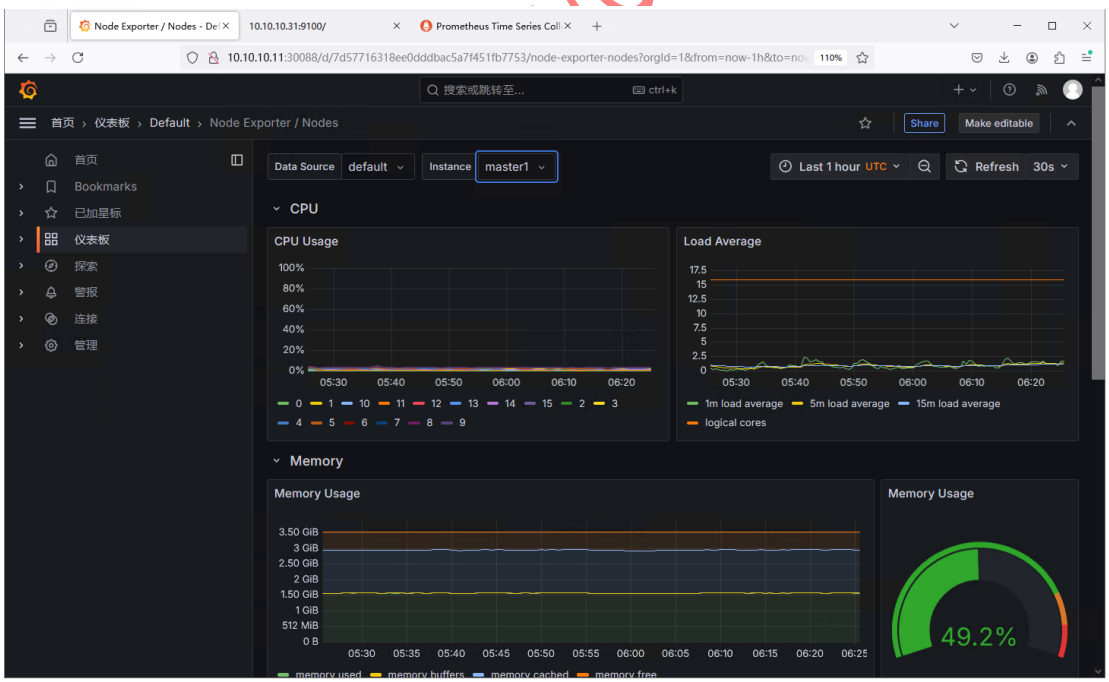


图 16-20 master1 计算资源使用情况

也可以在节点机上直接浏览 https://localhost:9100 查看。

Prometheus 页面

修改 prometheus-k8s 服务的类型为 NodePort，添加物理节点端口 nodePort=30081：

```
[root@master1 ~]# kubectl patch svc prometheus-k8s --type='json' -n monitoring \
-p='[{"op":"replace","path":"/spec/type","value":"NodePort"},{"op":"replace","path":"/spec/ports/0/nodePort","value":30081}]'
service/prometheus-k8s patched
```

访问测试 http://10.10.10.11:30081/，打开的页面参见图 16-21 所示。

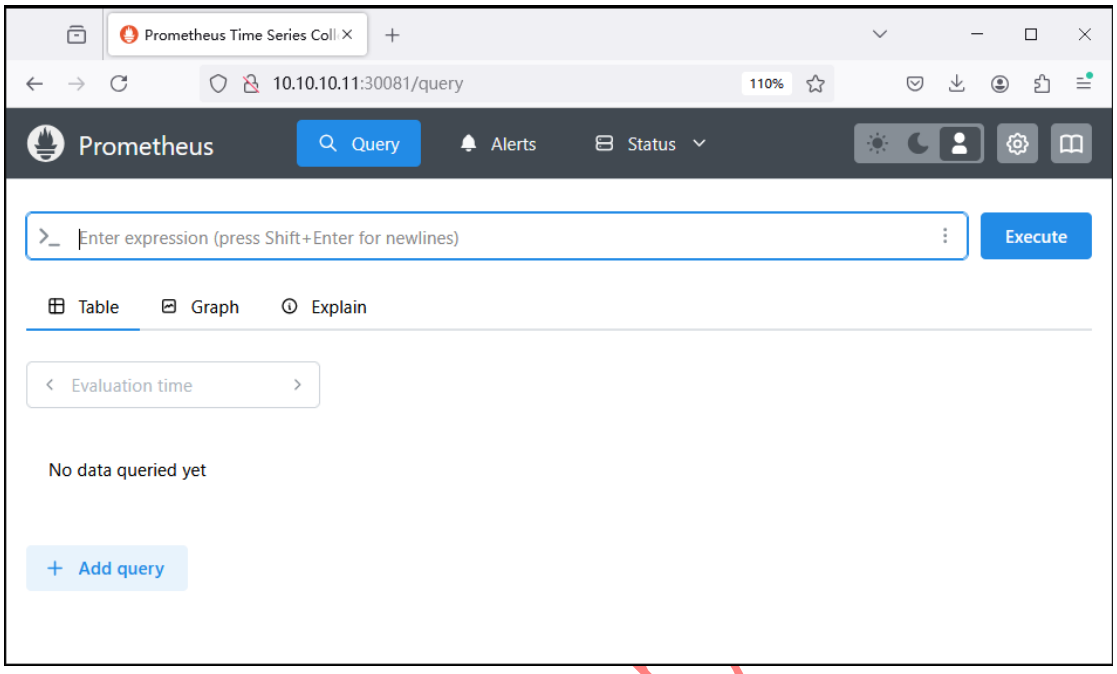


图 16-21 Prometheus 主页

这是采用 PromQL 语言查询时序数据库的界面，比如输入 up 查询各个节点上的服务，参考图 16-22 所示。

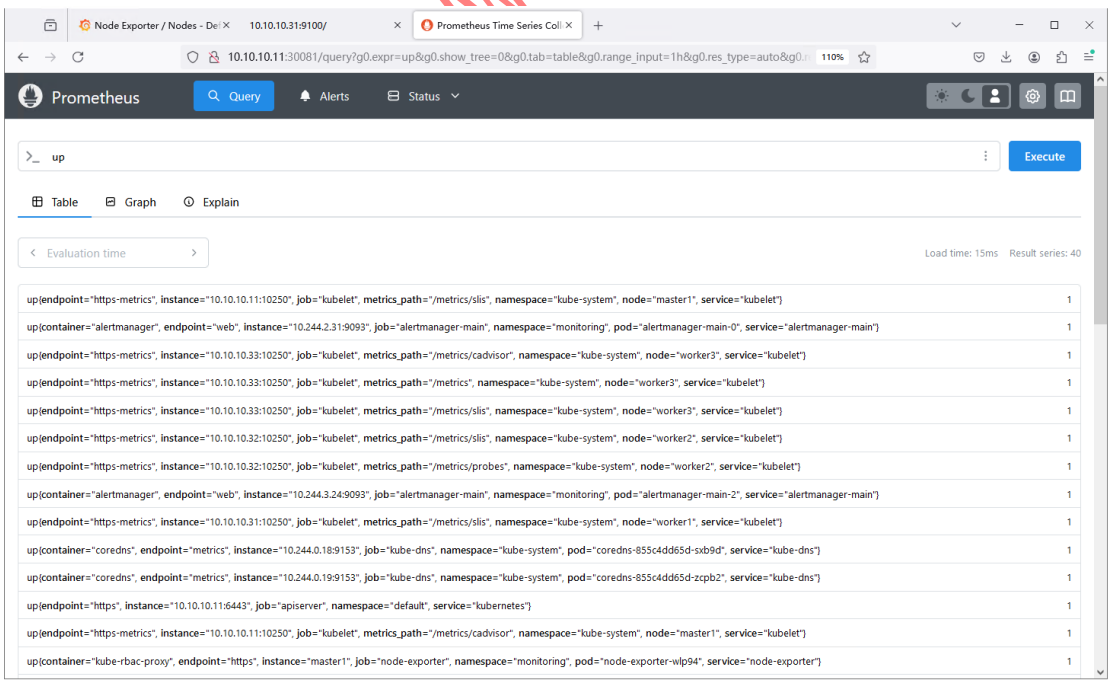


图 16-22 使用 PromQL 语言查询时许数据库

AlertManager 页面

修改 alertmanager-main 服务的类型为 NodePort，添加物理节点端口 nodePort=30082：

```
[root@master1 ~]# kubectl patch svc alertmanager-main --type='json' -n monitoring \
-p='[{"op":"replace","path":"/spec/type","value":"NodePort"},{"op":"replace","path":"/spec/ports/0/nodePort","value":30082}]'
service/alertmanager-main patched
```

访问测试 http://10.10.10.11:30082/，打开的页面参见图 16-23 所示。

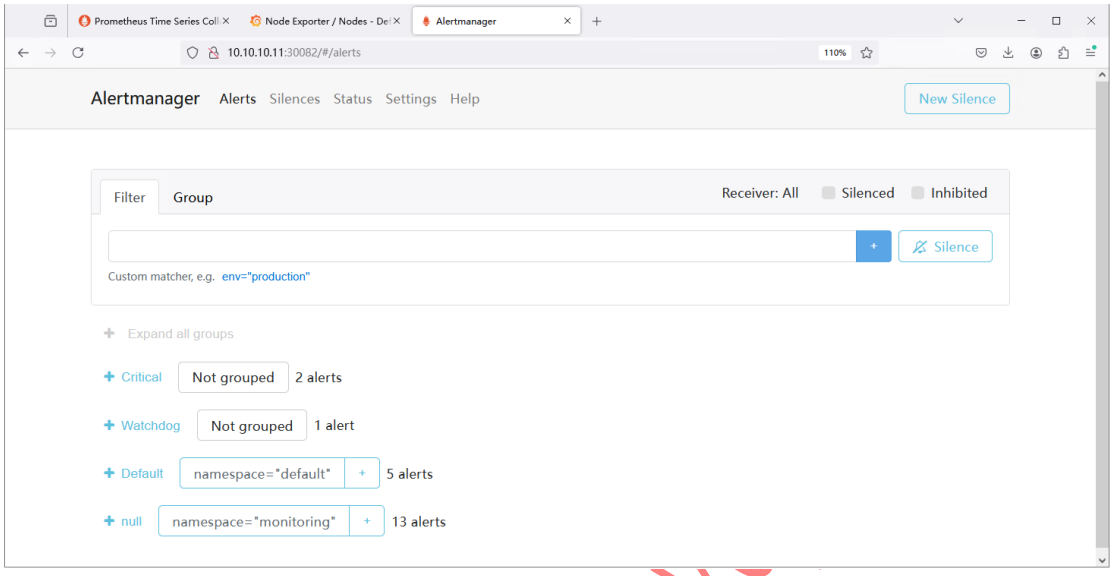


图 16-23 AlertManager 页面

当然也可以临时性做端口转发，采用命令“[root@master1 ~]# kubectl port-forward svc/alertmanager-main --address 0.0.0.0 30082:9093 -n monitoring”，这样只能通过 http://10.10.10.11:30082 访问页面。

整合 Prometheus 和 AlertManager

三、使用 Prometheus-Stack

1. 使用 PromQL 查询监控数据

Prometheus UI 是 Prometheus 内置的一个可视化管理界面，通过 Prometheus UI 用户能够轻松了解 Prometheus 当前的配置、监控任务运行状态等。 通过 Graph 面板，用户还能直接使用 PromQL 实时查询监控数据。访问测试 http://10.10.10.11:30081/进入 Prometheus UI，具体参见图 16-21 所示。

切换到 Graph 面板，用户可以使用 PromQL 表达式查询特定监控指标的监控数据。如图 16-23 所示，使用关键字 node_load1 查询 Prometheus 采集到的主机负载的样本数据，这些样本数据按照时间先后顺序展示，形成了主机负载随时间变化的趋势图表。

https://www.very

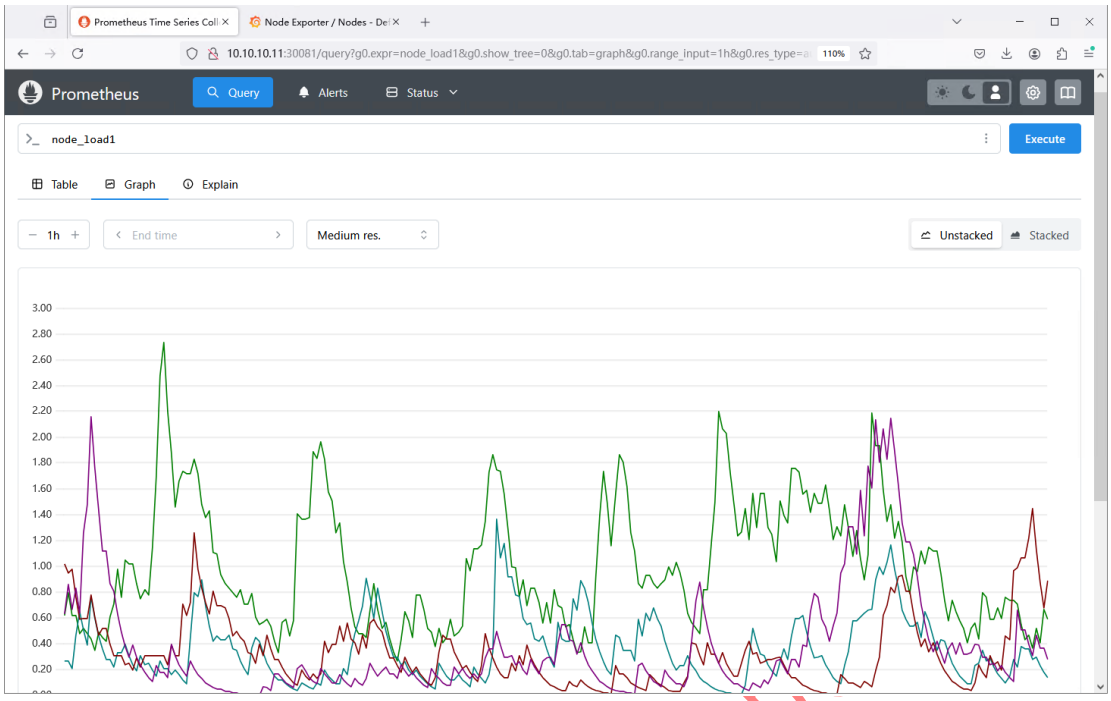


图 16-22 节点负载状况

2. 使用 Grafana

https://www.veryop

第17章 部署应用

如果说 k8s 集群是平台，是手段，那么在上面跑的应用就是目的，我们耗费这么大的精力和时间把集群搭起来了，目的就能方面在上面部署应用，正如修高速路是为了上面跑车那样。本章开始以实战的方式部署应用，让大家明白在 k8s 上部署应用是多么地容易——打个响指就搞定了！

首先创建一个名字空间 myspace，以后就在此名字空间内部署应用。

```
[root@master1 ~]# kubectl create namespace myspace
namespace/myspace created
```

§17.1 busybox

我们部署第一个应用就是 busybox。busybox 是一个很好的调试工具，可以测试和解决 k8s 集群的很多问题。busybox 号称是瑞士军刀，包含了很多常用的命令，比如 ping、wget、telnet 等日常运维需要用到的这些常用工具。虽然可以直接采用命令“kubectl create deployment mybusybox --image=busybox --namespace=myspace -- sleep 5h”部署（打个响指即可），但这次我们采用清单文件来部署，了解“响指”背后的机理最重要，谁让我们还是学生呢，对吧。

1. 新建清单文件——在 pod 中跑 busybox：

```
[root@master1 ~]# vim busybox.yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox-pod
  namespace: myspace
spec:
  containers:
  - image: busybox                                                    #pod 中的容器镜像
    command:                                                          #创建 Pod 后运行命令 sleep 1d，否则马上就销毁了 pod
    - sleep
    - "5h"
    imagePullPolicy: IfNotPresent                                     #如果本地存在镜像就不下载
    name: busybox-container
  restartPolicy: Always
```

本清单文件定义了一个 Pod 资源，几个重要的参数：(1)资源类型为 Pod，(2)Pod 的名字叫 busybox-pod，(3)在 myspace 名字空间内，(4)Pod 里的容器名为 busybox-container，容器镜像为 busybox，(5)启动容器后运行命令“sleep 5h”，这个命令睡眠 5 小时后退出，然后容器就会停止，在五小时内我们可以做很多测试，如果你觉得 5 小时不合适，可以演长或缩短，比如 2d（2 天）、30m（30 分钟）。关于 Pod 资源定义的详细介绍请参考“§16.2.1 定义资源对象”中的“2. Pod 和 PodTemplate”。

2. 创建资源——让 Pod 跑起来：

```
[root@master1 ~]# kubectl create -f busybox.yaml
pod/busybox-pod created
```

虽然命令很快返回，但并不表示 busybox 就已经部署完成并启动了，如果临时下载容器镜像，需要一些时间，如果网速慢，时间就更长了，还有可能因下载镜像超时导致部署失败。

3. 检查部署情况：

```
[root@master1 ~]# kubectl get pods -o wide -n myspace
NAME          READY   STATUS    RESTARTS   AGE   IP           NODE      NOMINATED NODE   READINESS GATES
busybox-pod   1/1     Running   0          36s   10.244.2.2   worker2   <none>           <none>
```

默认部署在 myspace 名字空间内，上面的命令只显示 default 名字空间内的 Pod，如果要显示所有名字空间内的 Pod，请带上参数--all-namespaces 或-A。发现 busybox 这个 pod 运行在 worker2 工作节点上，它的 pod 网络 Ip 为 10.244.2.2。如果部署时间较长，不断执行上面的命令查看部署状态就显得很麻烦，这时可用下面的命令动态监控：

```
[root@master1 ~]# kubectl get pods -o wide --watch -n myspace
NAME          READY   STATUS    RESTARTS   AGE     IP            NODE      NOMINATED NODE   READINESS GATES
busybox-pod   1/1     Running   0          2m34s   10.244.3.14   worker3   <none>           <none> >
```

可按 ctrl+c 中断上面的命令。

4．进一步查看 busybox-pod 这个 pod：

```
[root@master1 ~]# kubectl describe pod busybox-pod -n myspace
NAME          READY   STATUS    RESTARTS   AGE     IP            NODE      NOMINATED NODE   READINESS GATES
busybox-pod   1/1     Running   0          2m34s   10.244.3.14   worker3   <none>           <none>
^C[root@master1 ~]kubectl describe pod busybox-pod -n myspacece
Name:             busybox-pod
Namespace:        myspace
Priority:         0
Service Account:  default
Node:             worker3/10.10.10.33
Start Time:       Fri, 06 Dec 2024 17:19:39 +0800
Labels:           <none>
Annotations:      <none>
Status:           Running
IP:               10.244.3.14
IPs:
  IP:  10.244.3.14
Containers:
  busybox-c:
    Container ID:  containerd://e29b56c84b3f991bc374bc50b96f233a2277a0d6938e2859379238a5d5e6275c
    Image:         busybox
    Image ID:      docker.io/library/busybox@sha256:db142d433cdde11f10ae479dbf92f3b13d693fd1c91053da9979728cceb1dc68
    Port:          <none>
    Host Port:     <none>
    Command:
      sleep
      1d
    State:          Running
      Started:      Fri, 06 Dec 2024 17:19:40 +0800
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-tvw9m (ro)
Conditions:
  Type                        Status
  PodReadyToStartContainers   True
  Initialized                 True
  Ready                       True
  ContainersReady             True
  PodScheduled                True
Volumes:
  kube-api-access-tvw9m:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  3m35s  default-scheduler  Successfully assigned myspace/busybox-pod to worker3
  Normal  Pulled     3m34s  kubelet            Container image "busybox" already present on machine
  Normal  Created    3m34s  kubelet            Created container busybox-c
  Normal  Started    3m34s  kubelet            Started container busybox-c
```

如果发现下载镜像超时报错，只有采用离线镜像了，具体参考§23.2 手工下拉容器镜像 docker.io/library/busybox，然后删除 busybox 这个 pod，这样马上就会自动重建 Pod。或者干脆删掉资源（kubectl delete -f busybox.yaml）并重建（kubectl create -f busybox.yaml）。

查看完整的配置文件：

```
[root@master1 ~]# kubectl get pod busybox-pod -n myspace -o yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2024-12-06T09:19:39Z"
  name: busybox-pod
  namespace: myspace
  resourceVersion: "1702732"
  uid: 1c78cc69-5bbe-4c34-a07d-e53842e724ad
spec:
  containers:
  - command:
```

```
    - sleep
    - 1d
    image: busybox
    imagePullPolicy: IfNotPresent
    name: busybox-c
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount                    #容器内的挂载点
      name: kube-api-access-t68gc                                                 #挂载名称
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  nodeName: worker1
  preemptionPolicy: PreemptLowerPriority
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default                                                         #Service Acocount 账号
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: kube-api-access-t68gc
    projected:
      defaultMode: 420
      sources:
      - serviceAccountToken:
          expirationSeconds: 3607
          path: token                                                             #容器中挂载点内的 token 文件
      - configMap:
          items:
          - key: ca.crt
            path: ca.crt                                                          #容器中挂载点内的 CA 证书
          name: kube-root-ca.crt
      - downwardAPI:
          items:
          - fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
            path: namespace                                  #容器中挂载点内的 namespace 文件（定义了本 pod 所在的名字空间）
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2024-12-06T09:19:41Z"
    status: "True"
    type: PodReadyToStartContainers
  - lastProbeTime: null
    lastTransitionTime: "2024-12-06T09:19:39Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2024-12-06T09:19:41Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2024-12-06T09:19:41Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2024-12-06T09:19:39Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: containerd://e29b56c84b3f991bc374bc50b96f233a2277a0d6938e2859379238a5d5e6275c
    image: docker.io/library/busybox:latest
    imageID: docker.io/library/busybox@sha256:db142d433cdde11f10ae479dbf92f3b13d693fd1c91053da9979728cceb1dc68
    lastState: {}
    name: busybox-c
    ready: true
    restartCount: 0
    started: true
    state:
      running:
        startedAt: "2024-12-06T09:19:40Z"
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-tvw9m
      readOnly: true
      recursiveReadOnly: Disabled
  hostIP: 10.10.10.33
```

```
  hostIPs:
  - ip: 10.10.10.33
  phase: Running
  podIP: 10.244.3.14
  podIPs:
  - ip: 10.244.3.14
  qosClass: BestEffort
  startTime: "2024-12-06T09:19:39Z"
```

工作节点上的目录 /var/lib/kubelet/pods/c10e1d10-5601-4434-8346-06dea9dc5918/volumes/kubernetes.io~projected/kube-api-access-t68gc 会挂载到 Pod 中容器内的 /var/run/secrets/kubernetes.io/serviceaccount 目录上，此目录内有三个文件：ca.crt（就是签发 kube-apiserver 服务器证书的 CA 证书）、token（由 kube-controller-manager 产生并签名）和 namespace。当 Pod 访问 kube-apiserver 服务时，用此三个文件来做双向认证：Pod 使用 ca.crt 验证 kube-apiserver 的服务器证书，kube-apiserver 验证 Pod 的 token，请参考图 13-6。

宿主机上的目录 /var/lib/kubelet/pods/c10e1d10-5601-4434-8346-06dea9dc5918/volumes/kubernetes.io~projected/kube-api-access-t68gc 组成是这样的，c10e1d10-5601-4434-8346-06dea9dc5918 为 pod 的 id 号，kube-api-access-t68gc 为挂载名称，大家参考上面这条命令的输出，自行查验。

5. 登录 busybox 这个 pod 中：

```
[root@master1 ~]# kubectl get pods -o wide -n myspace
NAME          READY   STATUS    RESTARTS   AGE    IP             NODE      NOMINATED NODE   READINESS GATES
busybox-pod   1/1     Running   0          8m7s   10.244.3.14   worker3   <none>           <none>>


[root@master1 ~]# kubectl exec -it busybox-pod -n myspace -- sh
/ # uname -a
Linux busybox 5.14.0-333.el9.x86_64 #1 SMP PREEMPT_DYNAMIC Wed Jun 28 09:47:27 UTC 2023 x86_64 GNU/Linux
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
    link/ether 5e:85:d8:96:8f:af brd ff:ff:ff:ff:ff:ff
    inet 10.244.2.2/24 brd 10.244.2.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::5c85:d8ff:fe96:8faf/64 scope link
       valid_lft forever preferred_lft forever
/ # ip r
default via 10.244.2.1 dev eth0
10.244.0.0/16 via 10.244.2.1 dev eth0
10.244.2.0/24 dev eth0 scope link  src 10.244.2.2
/ # cd /var/run/secrets/kubernetes.io/serviceaccount
/var/run/secrets/kubernetes.io/serviceaccount # ls
ca.crt     namespace  token
/ # ping 9.9.9.9
PING 9.9.9.9 (9.9.9.9): 56 data bytes
64 bytes from 9.9.9.9: seq=0 ttl=127 time=72.286 ms
64 bytes from 9.9.9.9: seq=1 ttl=127 time=77.899 ms
64 bytes from 9.9.9.9: seq=2 ttl=127 time=83.564 ms
^C
--- 9.9.9.9 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 72.286/77.916/83.564 ms
/ # ping www.veryopen.org
ping: bad address 'www.veryopen.org'
```

如果在 Pod 中 ping 不通外面的域名，如 www.veryopen.org，问题处理参考§23.7 小节。也可以登陆到 pod 中指定的容器中（当 pod 中存在多个容器时只能这样登陆）。先列出 pod 中的容器，然后登陆容器：

```
[root@master1 ~]# kubectl get pod -o \
 "custom-columns=PodName:.metadata.name,Containers:.spec.containers[*].name,Image:.spec.containers[*].image" -n \
  myspace
PodName                   Containers          Image
busybox-pod               busybox-container   busybox
[root@master1 ~]# kubectl exec -n myspace busybox-pod -c busybox-container -it -- sh
```

§17.2 web 应用

接下来我们再来部署 web 引用，并对外暴露端口，以便外部能访问页面。

§17.2.1 部署 web 服务

下面我们在集群上部署一个 web 应用，采用 nginx 网站服务器。

1. 部署 web 应用：

```
[root@master1 ~]# kubectl create deployment nginx-www --image=nginx:alpine --replicas=2 -n myspace
deployment.apps/nginx-www created
```

部署名称 nginx-www，容器镜像 nginx:alpine，2 个 pod 做冗余实现高可用性。可用下面的命令输出上面命令所创建的资源清单文件（保存在 deploy123.yaml 文件中）：

```
[root@master1 ~]# kubectl get deployments nginx-www -n myspace -o yaml > deploy123.yaml
```

定义 Deployment 资源的字段请参考“§16.2 定义资源对象”小节中的“一、工作负载类资源”下的“4. Deployment”。

2. 过若干秒后查看部署情况：

```
[root@master1 ~]# kubectl get all -o wide -n myspace --show-labels
NAME                              READY   STATUS    RESTARTS      AGE    IP             NODE      NOMINATED NODE   READINESS GATES   LABELS
pod/busybox-pod                   1/1     Running   2 (20h ago)   2d20h  10.244.3.14    worker3   <none>           <none>            <none>
pod/nginx-www-689dc948d8-bdp88    1/1     Running   0             4h33m  10.244.2.28    worker2   <none>           <none>            app=nginx-www,pod-template-hash=689dc948d8
pod/nginx-www-689dc948d8-wnkth    1/1     Running   0             4h33m  10.244.1.21    worker1   <none>           <none>            app=nginx-www,pod-template-hash=689dc948d8

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE    CONTAINERS   IMAGES         SELECTOR        LABELS
deployment.apps/nginx-www    2/2     2            2           4h33m  nginx        nginx:alpine   app=nginx-www   app=nginx-www

NAME                                    DESIRED   CURRENT   READY   AGE    CONTAINERS   IMAGES         SELECTOR                                          LABELS
replicaset.apps/nginx-www-689dc948d8    2         2         2       4h33m  nginx        nginx:alpine   app=nginx-www,pod-template-hash=689dc948d8        app=nginx-www,pod-
template-hash=689dc948d8
```

发现第 1 步中的命令创建了三类资源：Pod、Deployment 和 Replicaset，ReplicaSet 确保在任何给定的时刻都在运行指定数量的 Pod 副本。Deployment 通过选择器（SELECTOR）定义的标签 app=nginx-www 来选定 pod，而以 nginx-www-689dc948d8-*命令的两个 Pod 都带了标签 app=nginx-www。Replicaset 也是通过一眼的选择器来选择 pod。

查看 nginx-www-54b468b46-fqlxp 这个 pod 内的容器：

```
[root@master1 ~]# kubectl get pod nginx-www-54b468b46-fqlxp -n myspace \
  -o jsonpath="{.spec['containers','initContainers'][*].name}"
nginx
```

正常，即每个 nginx-www pod 运行在不通的工作节点上。

```
[root@master1 ~]# kubectl describe deployment nginx-www -n myspace
Name:                   nginx-www
Namespace:              default
CreationTimestamp:      Tue, 18 Jul 2023 16:22:35 +0800
Labels:                 app=nginx-www                                          #本 Deployment 被贴上的标签
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=nginx-www                                          #选择带有 app=nginx-www 标签的 pod
Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx-www
  Containers:
```

https://www.veryopen.org

```
   nginx:
    Image:        nginx:alpine
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-www-75bd7dddf6 (2/2 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  5m8s  deployment-controller  Scaled up replica set nginx-www-75bd7dddf6 to 2

[root@master1 ~]# kubectl get pods -n myspace --show-labels                                          #显
示 Pod 的标签
NAME                             READY   STATUS        RESTARTS  AGE     LABELS
……
nginx-www-689dc948d8-75ldt       1/1     Running       0         40m     app=nginx-www,pod-template-hash=689dc948d8
nginx-www-689dc948d8-r9s4j       1/1     Running       0         40m     app=nginx-www,pod-template-hash=689dc948d8
```

如果不能访问国外网站或者速度很慢导致部署失败，建议执行下面的命令手工下来镜像：

```
[root@worker1 ~]# ctr -n k8s.io images pull --platform linux/amd64 docker.io/library/nginx:alpine
[root@worker1 ~]# ctr -n k8s.io images export --platform linux/amd64 nginx_alpine.tar docker.io/library/nginx:alpine
[root@worker1 ~]# scp nginx_alpine.tar worker2:./
[root@worker1 ~]# scp nginx_alpine.tar worker3:./

[root@worker2|worker3 ~]# ctr -n k8s.io images import --platform linux/amd64 nginx_alpine.tar
```

然后删除以 nginx-www 开头的两个 Pod，k8s 集群会自动重建 Pod。

§17.2.2 访问网页

```
[root@master1 ~]# kubectl get pods -l app=nginx-www -o wide -n myspace
NAME                         READY   STATUS    RESTARTS   AGE   IP           NODE      NOMINATED NODE   READINESS GATES
nginx-www-54b468b46-482dr    1/1     Running   0          81s   10.244.2.4   worker2   <none>           <none>
nginx-www-54b468b46-jq8mp    1/1     Running   0          81s   10.244.1.4   worker1   <none>           <none>
```

标签 app=nginx-www 就是上面那个 deployment 定义的标签。k8s 集群中，常采用标签来选择目标资源。

采用 curl 命令访问页面（在工作节点上访问自己 Pod 上的网站）：

```
[root@worker1 ~]# curl 10.244.1.4
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>

[root@worker2 ~]# curl 10.244.2.14
<!DOCTYPE html>
<html>
```

```
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

可见，从两个 pod 返回的页面内容是一样的。

也可以进入到容器中浏览页面(其中 pod 名称 nginx-www-75bd7dddf6-gtjlj 需要替换成实际的 pod 名字)：

```
[root@master1 ~]# kubectl exec nginx-www-75bd7dddf6-gtjlj -n myspace -- curl http://localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   615  100   615    0     0  1406k      0 --:--:-- --:--:-- --:--:--  600k
```

现在只能在节点上访问各自 pod 中的 web，或者进入 pod 中访问 http://localhost，不能从外部访问。

https://www.org

§17.2.3 向外暴露应用

对外暴露应用需要用到 Service 资源（ip 和端口位于服务网上，请参考图 17-1），Service 的类型有四种，分别是 ClusterIP（默认类型）、ExternalName、NodePort 和 LoadBalancer，其中后两种可把应用暴露到集群的外面，即外面的用户通过任一节点 ip:<30000-32727>或者负载均衡器的虚拟 ip 来访问集群内部的应用。ClusterIP 只是在服务网中创建一个 Service 对象，并分配一个服务网上的 ip，因此只能在 k8s 集群内部访问应用。而 ExternalName 把外部的应用集成到 k8s 集群中来。它们的关系参见图 17-1 所示。

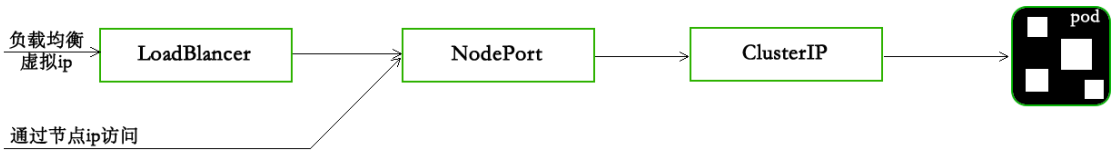


图 17-1 LoadBlancer、NodePort 和 ClusterIP 的关系

Service 实际上就是 pod 的逻辑组合，通过标签选择算符圈定一组 pod，这组 pod 通过 Service 对外呈现统一的访问入口，南北流量经由此入口后被分流到各个 pod。例如在图 17-1 中，Service 通过 app=A 标签选择一组 pod（即凡是带有 app:A 标签的 pod）组成服务的后端，用户通过 1.1.1.1 统一入口 ip 地址访问各个 pod，参考图 17-2。

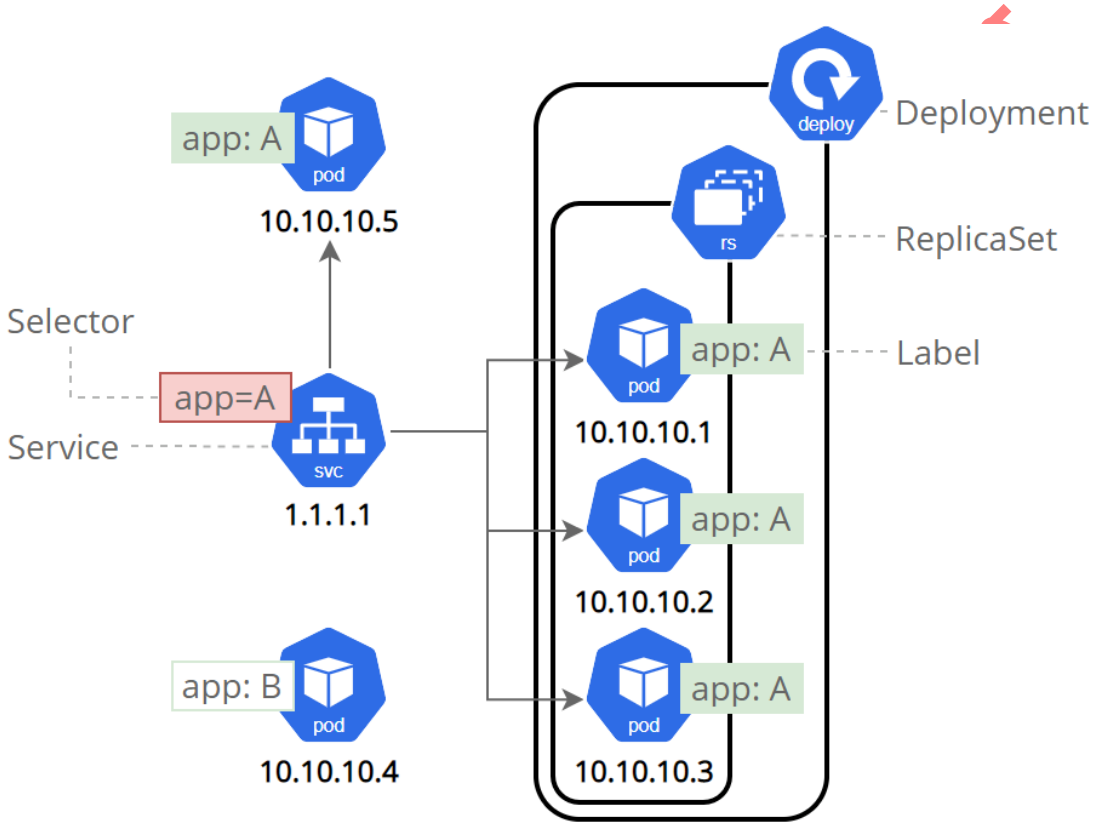


图 17-2 Service 通过标签算符选择一组 pod

一、通过 nodePort 类型的服务暴露

1. 在 myspace 名字空间内定义一个 NodePort 类型的 Service 资源：

```
[root@master1 ~]# vim webapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
  namespace: myspace
spec:
  type: NodePort
  ports:
  - protocol: TCP
    port: 8080                    #Service 网络中的端口
    targetPort: 80                #pod 中的 nginx 监听的端口
    nodePort: 30088               #工作节点上的端口
  selector:
```

```
    app: nginx-www                                    #选择 app=nginx-www 标签的 pod 作为服务的后端
```

访问的路径是：节点网 ip:30088 → 服务网 ip:8080 → pod 网 ip:80，其中节点网 ip 可为任意一个节点的 ip 地址。关于节点网、服务网和 pod 网的关系请参考图 16-1。关于 Service 资源对象的定义，请参考“§16.2.2 定义资源对象”小节中的“二、服务类资源”。

2. 创建 Service 资源：

```
[root@master1 ~]# kubectl create -f webapp-service.yaml
service/web-service created

[root@master1 ~]# kubectl get service -o wide -n myspace
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE     SELECTOR
web-service   NodePort    10.96.8.43    <none>        8080:30088/TCP   6m17s   app=nginx-www
```

刚创建的服务 web-service 为 NodePort 类型，分配了服务网 ip 10.96.8.43，服务网上的端口 8080 被映射到节点的 30088 端口。

查看 myspace 名字空间内的全部资源：

```
[root@master1 ~]# kubectl get all -o wide -n myspace --show-labels
NAME                            READY   STATUS    RESTARTS      AGE     IP            NODE      NOMINATED NODE   READINESS GATES   LABELS
pod/busybox-pod                 1/1     Running   2 (18h ago)   2d18h   10.244.3.14   worker3   <none>           <none>            <none>
pod/nginx-www-689dc948d8-bdp88  1/1     Running   0             149m    10.244.2.28   worker2   <none>           <none>            app=nginx-www,pod-template-hash=689dc948d8
pod/nginx-www-689dc948d8-wnkth  1/1     Running   0             149m    10.244.1.21   worker1   <none>           <none>            app=nginx-www,pod-template-hash=689dc948d8

NAME                  TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE   SELECTOR        LABELS
service/web-service   NodePort   10.96.149.155   <none>        8080:30088/TCP   14m   app=nginx-www   <none>

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES         SELECTOR        LABELS
deployment.apps/nginx-www   2/2     2            2           149m  nginx        nginx:alpine   app=nginx-www   app=nginx-www

NAME                                  DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES         SELECTOR                                          LABELS
replicaset.apps/nginx-www-689dc948d8  2         2         2       149m  nginx        nginx:alpine   app=nginx-www,pod-template-hash=689dc948d8        app=nginx-www,pod-
template-hash=689dc948d8
```

服务通过 app=nginx-www 标签选择器来确定“后端”的 pod，即凡是带有 app=nginx-www 标签的 pod 都被选中。

3. 访问网站。

可在任何节点上采用命令“curl 10.10.10.31:30088”或者 “curl 10.96.8.43:8088”获取页面，或者直接用浏览器打开 http://<任何节点的 ip>:30088，参考图 17-3 所示。

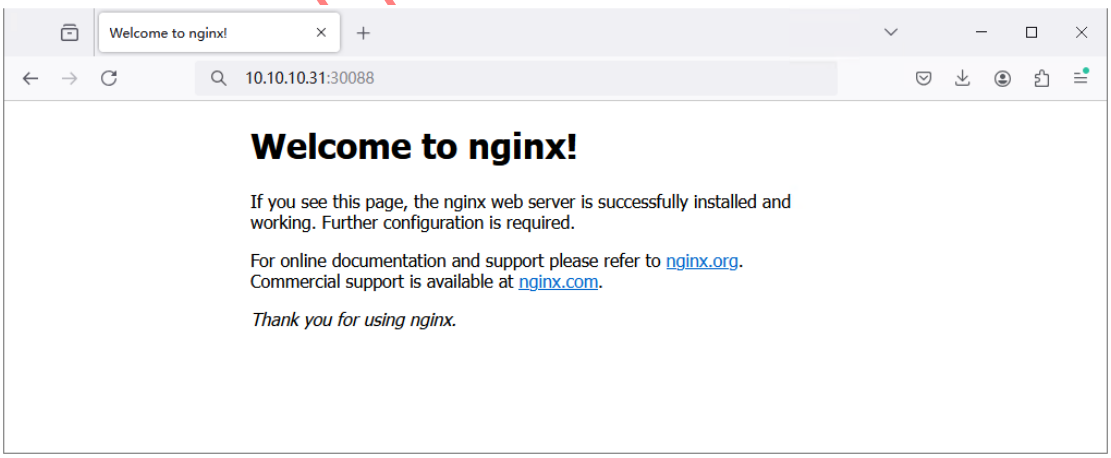


图 17-3 浏览 Pod 中的页面

```
[root@master1 ~]# kubectl describe svc web-service -n myspace
Name:                     web-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
```

```
Selector:                 app=nginx-www
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.96.8.43
IPs:                      10.96.8.43
Port:                     <unset>  8080/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30088/TCP
Endpoints:                10.244.1.13:80,10.244.2.14:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>

[root@master1 ~]# kubectl get endpoints -n myspace
NAME          ENDPOINTS                         AGE
web-service   10.244.1.21:80,10.244.2.28:80   148m
```

上面的命令显示 web-service 服务的后端的 URL 为 10.244.1.21:80、10.244.2.28:80，刚好就是两个跑了网站服务的 pod 的 ip，且网站监听 80 端口。

直接通过节点 ip 访问集群中的应用可不是一个好办法，因为节点可能会出故障，虽然 pod 会继续在另一个节点上跑，但是节点的 Ip 变了。在生产环境中，把应用暴露在 k8s 集群之外，以前常用 Ingress 资源对象来导航南北流量（客户端访问应用服务），但是 Ingress 存在很多问题，所以 SIG-NETWORK 组织开发了 Ingress 的完全替代品 Gateway API，官网 https://gateway-api.sigs.k8s.io/。截止到 2024 年 12 月 5 日，gateway api 的版本为 1.2.1。安装 gatewai api 后会增加四种资源对象：GatewayClass、Gateway、HTTPRoute 和 ReferenceGrant。可采用下面的命令部署它：

```
kubectl apply -f https://github.com/kubernetes-sigs/gateway-api/releases/download/v1.2.0/standard-install.yaml
```

由于本 web 应用跑在两个 Pod 中，而 pod 又跑在不同的工作节点上，尽管我们可以通过任何节点 Ip 访问页面，但是最终还是由 Service 把流量均衡到各个 pod。

4. 修改页面内容：

有两种方法，一是登陆到 Pod 的容器中，修改页面文件，第二种方法是直接在宿主机上修改，下面分别介绍这两种方法。

方法 1：进入容器修改

先找出 pod：

```
[root@master1 ~] kubectl -n myspace get pods -o wide
NAME                     READY   STATUS    RESTARTS      AGE     IP            NODE      NOMINATED NODE   READINESS GATES
mybusybox-5fc45f4b74-q9sqc   1/1     Running   1 (30m ago)   5d17h   10.244.3.11   worker3   <none>           <none>
nginx-www-54b468b46-68mjq    1/1     Running   1 (31m ago)   5d17h   10.244.1.9    worker1   <none>           <none>
nginx-www-54b468b46-ht8vv    1/1     Running   1 (31m ago)   5d17h   10.244.2.8    worker2   <none>           <none>
```

然后进入 Pod，查看页面文件名和所在的目录：

```
[root@master1 ~]# kubectl -n myspace exec nginx-www-54b468b46-68mjq -it -- sh
/ # more /etc/nginx/conf.d/default.conf
……
    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
……
```

修改页面：

```
/ # echo '<h1 lang="zh">1. 震中羞涩,不知市井繁华</h1>' > /usr/share/nginx/html/index.html
```

再进入另一个 pod 中做同样的修改。

https://www.verylinux.org

方法 2：在宿主机上修改然后拷贝过去

在 master1 上编辑 index.html：

```
[root@master1 ~]# vim ~/index.html
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>李贺《雁门太守行》</title>
</head>
<body>
    <h1>1. 报君黄金台上意</h1>
</body>
</html>
```

然后拷贝到其中的一个 Pod（注意：pod 名称要用实际名字替换）：

```
[root@master1 ~]# kubectl cp ~/index.html myspace/nginx-www-54b468b46-68mjq:/usr/share/nginx/html/
```

再修改 index.html：

```
[root@master1 ~]# vim ~/index.html
<!DOCTYPE html>
<html lang="zh">
<head>
    <meta charset="UTF-8">
    <title>李贺《雁门太守行》</title>
</head>
<body>
    <h1>2. 提携玉龙为君死</h1>
</body>
</html>
```

然后再拷贝到另外一个 Pod（注意：pod 名称要用实际名字替换）：

```
[root@master1 ~]# kubectl cp ~/index.html myspace/nginx-www-54b468b46-ht8vv:/usr/share/nginx/html/
```

~~由于 Pod 内的 nginx 存在缓存，所以直接修改节点上的页面文件后，不能马上在浏览器中看到结果。这时可以对部署进行缩放（控制 Pod 的副本数量）：~~

```
[root@master1 ~]# kubectl scale --replicas=0 deployment/nginx-www -n myspace                    #终止全部的 pod
[root@master1 ~]# kubectl scale --replicas=2 deployment/nginx-www -n myspace                    #再次启动 2 个 Pod
```

~~此后再浏览，应该能看到修改后的页面内容，~~并且多次按 Ctrl+F5 刷新页面，应该能看到不通的页面内容。

二、通过 LoadBalancer 类型的服务暴露

采用 nodePort 类型的服务暴露集群中的应用有一个缺点，就是访问时需要使用节点的 ip，但集群中的节点变动较频繁，因此会导致 ip 也会变动，这样可能导致用户没法访问页面了。最好的办法是创建 LoadBalancer 类型的服务，但是需要负载均衡器的支持才行，这样就可以使用负载均衡器的虚拟 ip 来访问页面了。下面我们采用 MetalLB 负载均衡器来实现它。首先参考“§16.5.3 Helm”安装并配置好 helm。

MetalLB 的官方网站 https://metallb.io/ 。从 https://github.com/metallb/metallb/releases/latest 查看最新版，目前为 0.15.2

MetalLB 有两种工作模式，分别是 Layer2 模式和 BGP 模式，任选一种模式进行配置即可，我们这里选择 Layer2 模式。

MetalLB 主要完成两个任务：

1）地址分配：当创建 LoadBalancer 的服务时，MetalLB 会为其分配 IP 地址。这个 IP 地址是从预先配置的 IP 地址池中获取的。同样，当 Service 删除后，已分配的 IP 地址会重新回到地址池中。
2）对外广播：分配了 IP 地址之后，需要让集群外的网络知道这个地址的存在。MetalLB 使用了标准路由协议实现：ARP、NDP 或者 BGP。在 Layer 2 模式下，使用 ARP（ipv4）/NDP（ipv6） 协议；而在 BPG 模式下，自然是

使用 BGP 协议。

MetalLB 分别使用两个组件来实现上述两个功能：

Controller：实现地址分配，以 Deployment 方式运行，专门监听 Service 的变更，以及分配/回收 IP 地址。
Speaker：实现地址对外广播，以 DaemonSet 方式运行，对外广播 Service 的 IP 地址。

具体的工作流如下：

Controller 负责监听 Service 变化并分配或回收 IP，当 Service 配置为 LoadBalancer 模式时，从 IP 池内分配相应的 IP 地址给 Service 并对该 IP 的生命周期进行管理。

创建 Service 时（或者从非 LoadBalancer 类型修改为 LoadBalancer 类型）时从 IP 池中选择一个 IP 并分配给它，删除 Service （或者从 LoadBalancer 类型修改为非 LoadBalancer 类型）时回收该 IP 到 IP 池内。

Speaker 则会依据选择的协议进行相应的广播或应答，实现 IP 地址的通信响应。当业务流量通过 TCP/UDP 协议到达指定的 Node 时，由 Node 上面运行的 Kube-Proxy 组件对流量进行处理，并分发到对应服务的 Pod 上面。

如果是 Layer2 模式 Speaker 就会响应 ARP（ipv4）/NDP（ipv6）请求；
如果是 BGP 模式 Speaker 则发送 BGP 广播，将路由规则同步给 peer。

大致的工作流程如下：

Layer 2 中的 Speaker 工作负载是 DeamonSet 类型，在每个节点上都调度一个 Pod，首先，几个 Pod 会先进行选举，选举出 Leader。由 Leader Pod 获取所有 LoadBalancer 类型的 Service，并将已分配的 IP 地址绑定到当前宿主机的网卡上。同时该 Leader 会响应对 ExternalIP 的 ARP（ipv4）/NDP（ipv6） 请求，因此从局域网层面来看，speaker 所在的机器是有多个 IP 地址的，当然其中也包含 ExternalIP。

因此与 ExternalIP 相关的所有流量都会流向该节点。在该节点上，kube-proxy 将接收到的流量传播到对应 Service 的后端 Pod。也就是说，所有 LoadBalancer 类型的 Service 的 IP 同一时间都是绑定在同一台节点的网卡上。

下面我们来安装 MetalLB。

1. 检查 kube-proxy 的模式：

```
[root@master1 ~]# curl localhost:10249/proxyMode
iptables[root@master1 ~]#
```

可见这里用的是 iptables 模式。如果是 IPVS 模式，请做如下修改：

```
[root@master1 ~]# kubectl edit configmap -n kube-system kube-proxy
……
    ipvs:
      excludeCIDRs: null
      minSyncPeriod: 0s
      scheduler: ""
      strictARP: true
……
```

2. 部署 metallb:

如果方法 1 不行就选择方法 2。

https://www.veryopen.org

方法 1：

```
[root@master1 ~]# helm repo add metallb https://metallb.github.io/metallb
[root@master1 ~]# helm upgrade --install -n metallb-system metallb bitnami/metallb
```

方法 1 结束

方法 2：

首先把 metallb_controller 和 metallb_speaker 两个容器镜像导入到各个节点上（master1 可不导入 metallb_controller）：

```
[root@master1 ~]# ctr -n k8s.io images import metallb_speaker_v0.15.2.tar

[root@worker? ~]# ctr -n k8s.io images import metallb_speaker_v0.15.2.tar
[root@worker? ~]# ctr -n k8s.io images import metallb_controller_v0.15.2.tar
```

然后正式部署 metallb：

```
[root@master1 ~]# wget https://raw.githubusercontent.com/metallb/metallb/v0.15.2/config/manifests/metallb-native.yaml
[root@master1 ~]# kubectl create -f metallb-native.yaml
namespace/metallb-system created
Warning: unrecognized format "int32"
customresourcedefinition.apiextensions.k8s.io/bfdprofiles.metallb.io created
customresourcedefinition.apiextensions.k8s.io/bgpadvertisements.metallb.io created
customresourcedefinition.apiextensions.k8s.io/bgppeers.metallb.io created
customresourcedefinition.apiextensions.k8s.io/communities.metallb.io created
Warning: unrecognized format "int64"
customresourcedefinition.apiextensions.k8s.io/ipaddresspools.metallb.io created
customresourcedefinition.apiextensions.k8s.io/l2advertisements.metallb.io created
customresourcedefinition.apiextensions.k8s.io/servicebgpstatuses.metallb.io created
customresourcedefinition.apiextensions.k8s.io/servicel2statuses.metallb.io created
serviceaccount/controller created
serviceaccount/speaker created
role.rbac.authorization.k8s.io/controller created
role.rbac.authorization.k8s.io/pod-lister created
clusterrole.rbac.authorization.k8s.io/metallb-system:controller created
clusterrole.rbac.authorization.k8s.io/metallb-system:speaker created
rolebinding.rbac.authorization.k8s.io/controller created
rolebinding.rbac.authorization.k8s.io/pod-lister created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:controller created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:speaker created
configmap/metallb-excludel2 created
secret/metallb-webhook-cert created
service/metallb-webhook-service created
deployment.apps/controller created
daemonset.apps/speaker created
validatingwebhookconfiguration.admissionregistration.k8s.io/metallb-webhook-configuration created
```

创建了很多资源，且这些资源都在 metalb-system 名字空间内。

方法 2 结束

查看 metalb-system 名字空间中的全部资源：

```
[root@master1 ~]# kubectl get all -n metallb-system -o wide --show-labels
NAME                              READY   STATUS    RESTARTS   AGE   IP             NODE      NOMINATED NODE   READINESS GATES   LABELS
pod/controller-8694df9d9b-xkxxn   1/1     Running   0          61s   10.244.1.25    worker1   <none>           <none>            app=metallb,component=controller,pod-template-hash=8694df9d9b
pod/speaker-2s5mt                 1/1     Running   0          61s   10.10.10.31    worker1   <none>           <none>            app=metallb,component=speaker,controller-revision-hash=7ff6d8c7b4,pod-template-generation=1
pod/speaker-f67lz                 1/1     Running   0          61s   10.10.10.33    worker3   <none>           <none>            app=metallb,component=speaker,controller-revision-hash=7ff6d8c7b4,pod-template-generation=1
pod/speaker-i87q7                 1/1     Running   0          61s   10.10.10.32    worker2   <none>           <none>            app=metallb,component=speaker,controller-revision-hash=7ff6d8c7b4,pod-template-generation=1
pod/speaker-jg7lm                 0/1     Running   0          61s   10.10.10.11    master1   <none>           <none>            app=metallb,component=speaker,controller-revision-hash=7ff6d8c7b4,pod-template-generation=1

NAME                         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE   SELECTOR                LABELS
service/metallb-webhook-service   ClusterIP   10.96.162.226   <none>        443/TCP   61s   component=controller   <none>

NAME                        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE   CONTAINERS   IMAGES                          SELECTOR                  LABELS
daemonset.apps/speaker   4         4         3       4            3           kubernetes.io/os=linux   61s   speaker      quay.io/metallb/speaker:v0.14.8   app=metallb,component=speaker   app=metallb,component=speaker

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES                          SELECTOR                  LABELS
deployment.apps/controller   1/1     1            1           61s   controller   quay.io/metallb/controller:v0.14.8   app=metallb,component=controller   app=metallb,component=controller

NAME                                    DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES                          SELECTOR                  LABELS
replicaset.apps/controller-8694df9d9b   1         1         1       61s   controller   quay.io/metallb/controller:v0.14.8   app=metallb,component=controller,pod-template-hash=8694df9d9b   app=metallb,component=controller,pod-template-hash=8694df9d9b
```

可见服务 metallb-webhook-service 的后端为 pod/controller-8694df9d9b-xkxxn（使用标签选择器来选定 pod，参考上面黄色字体部分）。

在确保全部的 Pod 状态都是 Running 后再往下做，否则请参考“§23.10 pod 异常”小节排错。

3.配置：

```
[root@master1 ~]# vim lb-ip-pools.yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: metallb-system
spec:
  addresses:
    - 10.10.10.100-10.10.10.127          ---------- #分配负载均衡器的虚拟 Ip 的地址池
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: empty
  namespace: metallb-system
spec:
  ipAddressPools:
  - first-pool

[root@master1 ~]# kubectl apply -f lb-ip-pools.yaml
ipaddresspool.metallb.io/first-pool created
l2advertisement.metallb.io/empty created

[root@master1 ~]# kubectl get ipaddresspool  -n metallb-system
NAME         AUTO ASSIGN   AVOID BUGGY IPS   ADDRESSES
first-pool   true          false             ["10.10.10.100-10.10.10.127"]
```

2. 创建 LoadBalancer 服务：

```
[root@master1 ~]# vim webapp-service-lb.yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service-lb
  namespace: myspace
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 8081                                                          #Service 网络中的端口
    targetPort: 80   ---------------------------------------------------#pod 中的 nginx 监听的端口
  selector:
    app: nginx-www

[root@master1 ~]# kubectl create -f webapp-service-lb.yaml
service/web-service-lb created

[root@master1 ~]# kubectl get svc -n myspace
NAME             TYPE           CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
kubernetes       ClusterIP      10.96.0.1     <none>         443/TCP          119d
web-service-lb   LoadBalancer   10.96.1.25    10.10.10.100   8081:31827/TCP   18s
```

看到了吗？自动分配了虚拟 ip 地址 10.10.10.100，并且服务网上的 8081 端口映射到节点的 31827 端口，然后在浏览器中浏览 http://10.10.10.100:8081 、 http://< 节点 ip>:31827 或者采用命令“ curl 10.10.10.100:8081”即可。如果不能访问，请检查 web-service-lb 是不是和网站服务的 pod 在同一个名字空间，另外检查四台宿主机上的防火墙是否关闭（可使用命令 systemctl --now disable firewalld 关闭）。

进一步查看服务 web-service-lb 的详细信息：

```
[root@master1 ~]# kubectl describe svc web-service-lb -n myspace
Name:                     web-service-lb
Namespace:                myspace
Labels:                   <none>
Annotations:              metallb.universe.tf/ip-allocated-from-pool: first-pool
Selector:                 app=nginx-www
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.96.248.15
IPs:                      10.96.248.15
LoadBalancer Ingress:     10.10.10.100 (VIP)
Port:                     <unset>  8081/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30327/TCP
Endpoints:                10.244.2.4:80,10.244.1.4:80
Session Affinity:         None
External Traffic Policy:  Cluster
Internal Traffic Policy:  Cluster
Events:
  Type    Reason        Age   From                Message
```

```
  ----    ------          ----  ----                -------
  Normal  IPAllocated     85s   metallb-controller  Assigned IP ["10.10.10.101"]
  Normal  nodeAssigned    85s   metallb-speaker     announcing from node "worker3" with protocol "layer2"
```

§17.3 数据库

接下来我们来部署一个有状态应用——mariadb 数据库，状态数据被保存在宿主机的某个目录中（比如/opt/data/mysql）。

1. 定义持久卷（Persistent Volume）资源对象：

```
[root@master1 ~]# mkdir mariadb && cd mariadb
[root@master1 mariadb]# vim pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mariadb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce                                                    #互斥写
  persistentVolumeReclaimPolicy: Retain                                #PV 的回收策略
  hostPath:                                                            # hostPath 服务配置
    path: "/opt/data/mysql"                                            #  宿主机路径
```

关于 PV 资源的定义，请参考“§16.4 k8s 中的存储”小节。Pv 属于全局资源，所以不能放入名字空间内。

2. 定义 PVC 资源对象：

```
[root@master1 mariadb]# vim pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mariadb-pvc
  namespace: myspace
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  volumeName: mariadb-pv
```

关于 PVC 资源的定义，请参考“§16.4 k8s 中的存储”小节。

3. 定义 Secret 资源对象：

```
[root@master1 mariadb]# vim secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mariadb-secret
  namespace: myspace
type: Opaque
data:
  mariadb-root-password: QTFiMkMz                                  #echo -n "A1b2C3" | base64
```

即 mariadb 的 root 用户的密码为 A1b2C3，采用 base64 编码方式保存在清单文件中。

4．定义有状态集（StatefulSet）资源对象：

```
[root@master1 mariadb]# vim statefuleset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mariadb-sts
  namespace: myspace
spec:
  serviceName: "mariadb-service"
  selector:
    matchLabels:
      app: mariadb
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
      - name: mariadb
        image: mariadb
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 3306
          name: mariadb-port
        env:
        # Using Secrets
        - name: MARIADB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mariadb-secret
              key: mariadb-root-password
        volumeMounts:
        - name: pvclaim
          mountPath: /var/lib/mysql/
      volumes:
      - name: pvclaim
        persistentVolumeClaim:
          claimName: mariadb-pvc
```

关于 StatefulSet 资源对象的定义，请参考“§16.2.1 定义资源对象”中的“5. StatefulSet”。

5．定义服务(Service)资源对象：

```
[root@master1 mariadb]# vim service.yaml
apiVersion: v1
kind: Service
metadata:
  name: mariadb-service
  namespace: myspace
  labels:
    app: mariadb
spec:
  ports:
  - port: 3306
    name: mariadb-port
  selector:
    app: mariadb
```

这些资源对象之间的关系见图 17-4 所示。

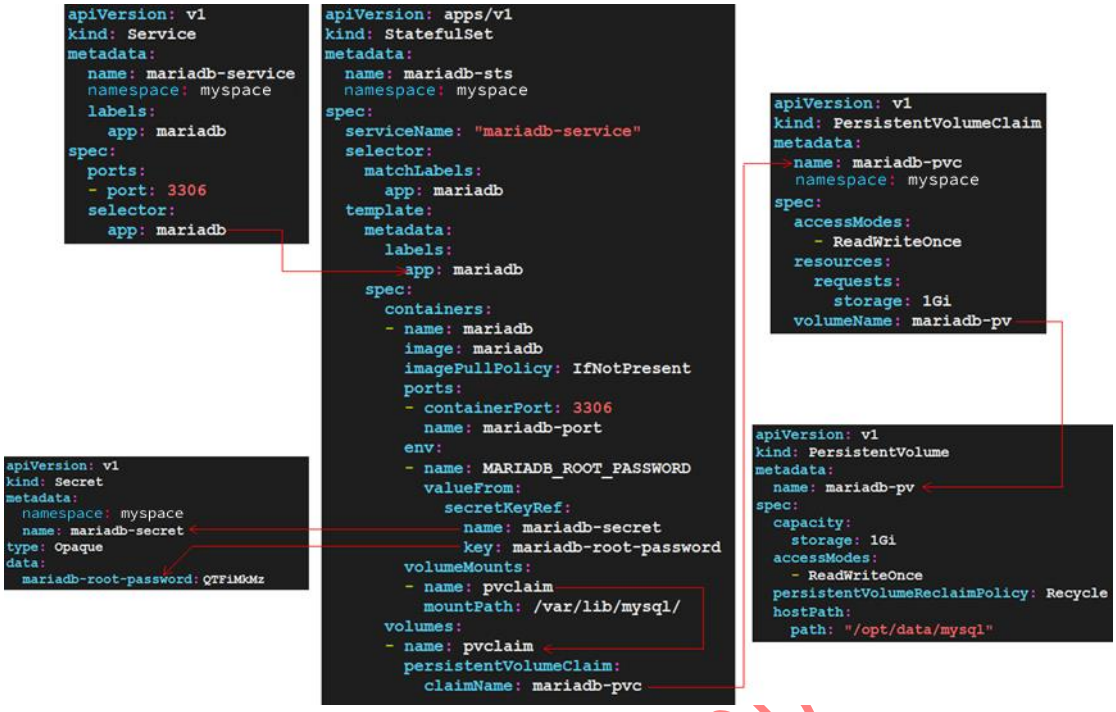


图 17-4 各个资源对象间的关系

6. 创建所有资源对象：

```
[root@master1 mariadb]# cd ../
[root@master1 ~]# kubectl create -f mariadb/
persistentvolumeclaim/mariadb-pvc created
secret/mariadb-secret created
service/mariadb-service created
statefulset.apps/mariadb-sts created
persistentvolume/mariadb-pv
```

如果只部分资源创建成功，请排错后先执行命令“kubectl delete -f mariadb/”删除已经存在的资源后再来创建全部资源对象。

不过为了排错方便，建议一个个资源单独创建，这样就容易发现哪个资源创建失败。创建资源的顺序为：Secret→PersistentVolume→ PersistentVolumeClaim→StatefulSet→Service。

下面检查各种资源是否就绪：

```
[root@master1 ~]# kubectl get all -n myspace -o wide
NAME                          READY   STATUS    RESTARTS      AGE     IP             NODE      NOMINATED NODE   READINESS GATES
pod/busybox-pod               1/1     Running   3 (51m ago)   11d     10.244.2.68    worker2   <none>           <none>
pod/mariadb-sts-0             1/1     Running   0             6m52s   10.244.1.67    worker1   <none>           <none>
pod/nginx-www-6547745c89-2jqlf   1/1     Running   1 (51m ago)   6d2h    10.244.4.64    worker3   <none>           <none>
pod/nginx-www-6547745c89-g2s7c   1/1     Running   1 (51m ago)   6d2h    10.244.1.61    worker1   <none>           <none>

NAME                      TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)          AGE     SELECTOR
service/mariadb-service   ClusterIP      10.96.207.191   <none>         3306/TCP         6m43s   app=mariadb
service/web-service       NodePort       10.96.211.87    <none>         8080:30088/TCP   6d23h   app=nginx-www
service/web-service-lb    LoadBalancer   10.96.110.147   10.10.10.100   8081:31859/TCP   6d1h    app=nginx-www

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES         SELECTOR
deployment.apps/mybusybox         1/1     1            1           11d   busybox      busybox        app=mybusybox
deployment.apps/nginx-www         2/2     2            2           7d    nginx        nginx:alpine   app=nginx-www

NAME                                    DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES         SELECTOR
replicaset.apps/mybusybox-768657c9c7    1         1         1       11d   busybox      busybox        app=mybusybox,pod-template-hash=768657c9c7
replicaset.apps/nginx-www-6547745c89    2         2         2       7d    nginx        nginx:alpine   app=nginx-www,pod-template-hash=6547745c89

NAME                             READY   AGE     CONTAINERS   IMAGES
statefulset.apps/mariadb-sts     1/1     6m52s   mariadb      mariadb

[root@master1 ~]# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                 STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
mariadb-pv                                 1Gi        RWO            Retain           Bound    myspace/mariadb-pvc                  <unset>                          21h
pvc-1afb2541-a191-4dc4-a870-e8f632986602   10Mi       RWX            Delete           Bound    default/test-claim    nfs-client     <unset>                          19d
[root@master1 ~]# kubectl get pvc -n myspace
```

```
NAME          STATUS   VOLUME       CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
mariadb-pvc   Bound    mariadb-pv   1Gi        RWO                           <unset>                 21h
[root@master1 ~]# kubectl get secret -n myspace
NAME             TYPE     DATA   AGE
mariadb-secret   Opaque   1      21h
```

注意黄色字体部分，不能出现错误。

查看数据库所在容器的主机名：

```
[root@master1 ~]# kubectl -n myspace exec mariadb-sts-0 -- bash -c "hostname -f"
mariadb-sts-0.mariadb-service.default.svc.cluster.local
```

另外 mariadb-service 服务为所属的 Pod 创建了一个域名：mariadb-sts-0.mariadb-service.default.svc.cluster.local，可在其他 pod 中 ping 通此域名：

```
[root@master1 ~]# kubectl get pod -n myspace
NAME                        READY   STATUS    RESTARTS      AGE
busybox-pod                 1/1     Running   3 (86m ago)   11d
mariadb-sts-0               1/1     Running   0             7m46s
mybusybox-768657c9c7-gqxfq  1/1     Running   3 (86m ago)   11d
nginx-www-6547745c89-fxfb9  1/1     Running   2 (86m ago)   7d
nginx-www-6547745c89-mgnhf  1/1     Running   2 (86m ago)   7d

[root@master1 ~]# kubectl exec -it busybox-pod -n myspace -- sh
/ # ping mariadb-sts-0.mariadb-service.default.svc.cluster.local
PING mariadb-sts-0.mariadb-service.default.svc.cluster.local (10.244.1.67): 56 data bytes
64 bytes from 10.244.1.67: seq=0 ttl=62 time=0.281 ms
64 bytes from 10.244.1.67: seq=1 ttl=62 time=0.821 ms
```

按 ctrl+c 中断 ping 命令，然后输入 exit 退出 pod。

7. 使用数据库：

```
[root@master1 ~]# kubectl -n myspace exec -it mariadb-sts-0 -- mariadb -uroot -pA1b2C3
......
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> CREATE DATABASE mytest0;
Query OK, 1 row affected (0.000 sec)

MariaDB [(none)]> use mytest0;
Database changed
MariaDB [mytest0]> CREATE TABLE t(age int);
Query OK, 0 rows affected (0.002 sec)

MariaDB [mytest0]> INSERT INTO t VALUES(18),(20);
Query OK, 2 rows affected (0.001 sec)
Records: 2  Duplicates: 0  Warnings: 0

MariaDB [mytest0]> SELECT * FROM t;
+-------+
| age   |
+-------+
|    18 |
|    20 |
+-------+
2 rows in set (0.000 sec)

MariaDB [mytest0]> quit
```

也可以通过服务网 ip 登录数据库：

```
[root@master1 ~]# kubectl get svc -o wide -n myspace
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE    SELECTOR
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP          2d9h   <none>
mariadb-service   ClusterIP   10.96.58.204     <none>        3306/TCP         58s    app=mariadb
web-service       NodePort    10.96.201.207    <none>        8080:30080/TCP   2d6h   app=nginx-www

[root@master1 ~]# dnf install mysql -y                                          #安装数据库客户端

[root@master1 ~]# mysql -h 10.96.58.204 -uroot -pA1b2C3
......
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

服务网的 ip 是 ping 不通的。如果上面的命令失败，请重点检查服务的标签选择器是否匹配了 pod 的标签，本例子的标签就是 app=mariadb。采用命令“kubectl get all -n myspace -o wide --show-labels”显示各种资源的标签。

§17.4 wordpress

流行极广的开源网站定制项目 wordpress 要用到数据库 Mariadb 或 MySQL 以及 php，在 k8s 集群上部署 wordpress 极其简单，采用 helm 一条命令（helm install myblog bitnami/wordpress）搞定。为了了解部署的全过程，这里我们手工一步一步部署。

1．创建有关目录，并进入：

```
[root@master1 ~]# mkdir -p wordpress/{db,wp}
[root@master1 ~]# cd wordpress
```

2．创建名字空间 wordpress。

```
[root@master1 wordpress]# vim namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: wordpress
  labels:
    name: wordpress

[root@master1 wordpress]# kubectl create -f namespace.yaml
namespace/wordpress created

[root@master1 wordpress]# cd ~/wordpress/db
```

§17.4.1 部署数据库 MariaDB

参考图 17-5，在目录~/wordpress/db 中新建 mariadb-service.yaml、mariadb-secret.yaml、mariadb-statefulset.yaml、mariadb-pvc.yaml 和 mariadb-pv.yaml 五个资源清单文件。

https://www.very

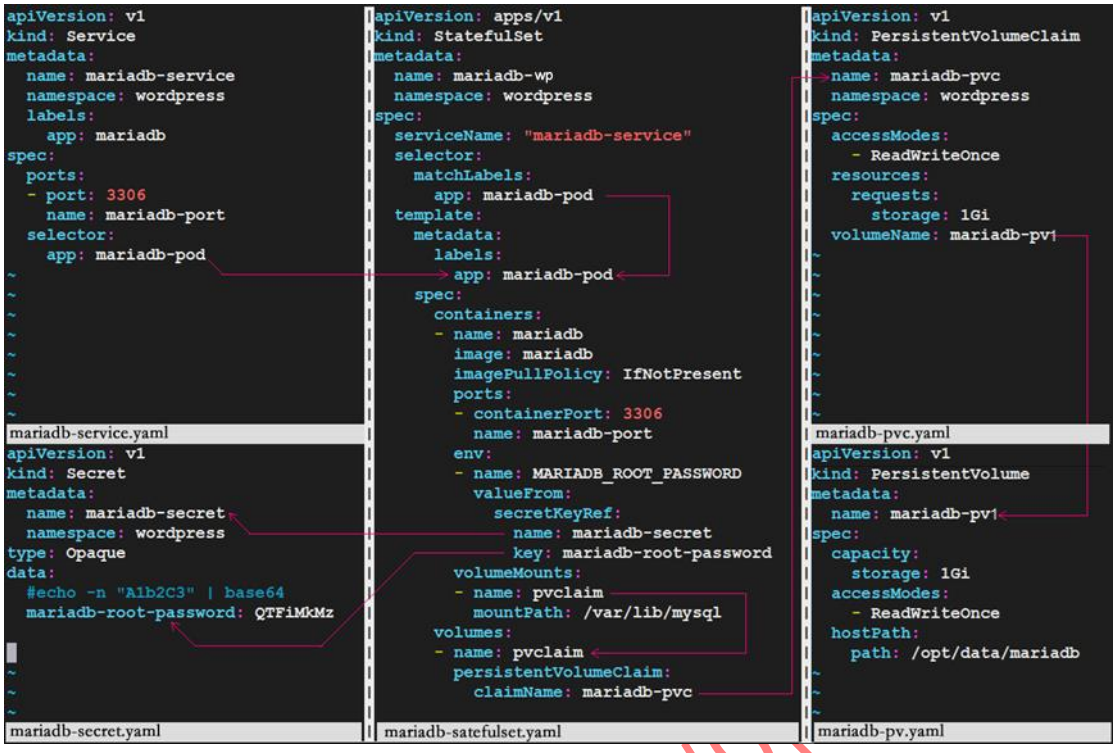


图 17-5 为 wordpress 部署 MariaDB 数据库

注意 mariadb-secret.yaml 文件中的密码 QTFiMkMz 里的“k”是小写字母。

然后创建这些资源对象（确保进入了~/wordpress/db 目录）：

```
[root@master1 ~]# cd ~/wordpress/db
[root@master1 db]# kubectl create -f ./
persistentvolume/mariadb-pv created
persistentvolumeclaim/mariadb-pvc created
secret/mariadb-secret created
service/mariadb-service created
statefulset.apps/mariadb-wp created
```

```
[root@master1 ~]# kubectl get all -n wordpress -o wide                    #查看 wordpress 名字空间内的全部资源对象
NAME                READY   STATUS    RESTARTS   AGE   IP            NODE      NOMINATED NODE   READINESS GATES
pod/mariadb-wp-0    1/1     Running   0          4s    10.244.1.41   worker1   <none>           <none>

NAME                        TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE   SELECTOR
service/mariadb-service     ClusterIP   10.96.118.23   <none>        3306/TCP   18m   app=mariadb-pod

NAME                                READY   AGE   CONTAINERS   IMAGES
statefulset.apps/mariadb-wp         1/1     4s    mariadb      mariadb

[root@master1 ~]# kubectl get events -n wordpress                         #查看有无发生错误事件
LAST SEEN   TYPE      REASON              OBJECT                      MESSAGE
20m         Normal    Scheduled           pod/mariadb-wp-0            Successfully assigned wordpress/mariadb-wp-0 to worker1
20m         Normal    Pulling             pod/mariadb-wp-0            Pulling image "mariadb"
16m         Normal    Pulled              pod/mariadb-wp-0            Successfully pulled image "mariadb" in 3m8.798s (3m8.798s including
waiting). Image size: 122469609 bytes.
4m58s       Normal    Pulled              pod/mariadb-wp-0            Container image "mariadb" already present on machine
74s         Normal    Scheduled           pod/mariadb-wp-0            Successfully assigned wordpress/mariadb-wp-0 to worker1
73s         Normal    Pulled              pod/mariadb-wp-0            Container image "mariadb" already present on machine
73s         Normal    Created             pod/mariadb-wp-0            Created container mariadb
72s         Normal    Started             pod/mariadb-wp-0            Started container mariadb
20m         Normal    SuccessfulCreate    statefulset/mariadb-wp      create Pod mariadb-wp-0 in StatefulSet mariadb-wp successful
```

可忽略历史错误信息。

```
[root@master1 ~]# kubectl exec mariadb-wp-0 -n wordpress -- hostname -f          #查看数据库的主机名
mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local
```

记住这个主机名，后面部署 web 服务时要用到它。

测试数据库是否部署成功：

```
[root@master1 ~]# kubectl get svc -n wordpress -o wide
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE   SELECTOR
mariadb-service   ClusterIP   10.96.118.101   <none>        3306/TCP   45m   app=mariadb-pod

[root@master1 ~]# mysql -h 10.96.118.101 -uroot -pA1b2C3
……
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

如果没有 mysql 命令，请安装它（dnf install mysql -y）。

§17.4.2 部署 WEB 服务

1. 编写各种资源清单文件

在~/wordpress/wp 目录中创建下面的文件：wordpress-deployment.yaml、wordpress-pvc.yaml、wordpress-pv.yaml 和 wordpress-service.yaml，内容如图 17-6。

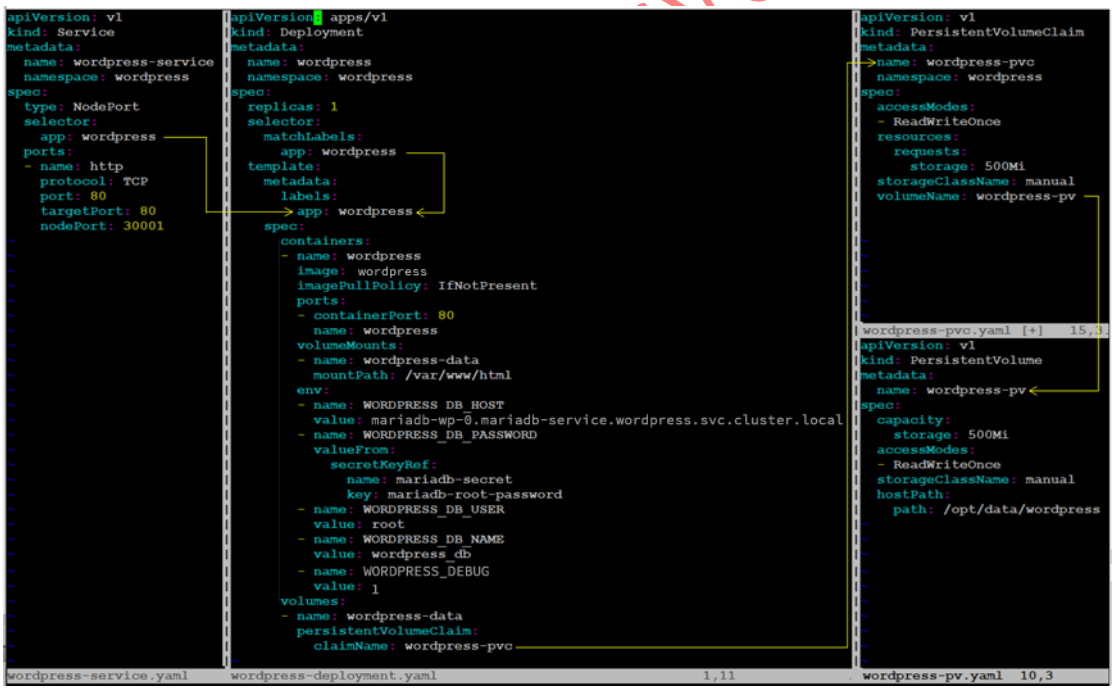


图 17-6 wordpress 网站资源文件

数据库所在的容器的主机名为 mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local（可执行命令“kubectl exec mariadb-wp-0 -n wordpress -- hostname -f”获取这个主机名），也可以采用数据库的服务名mariadb-service，文件wordpress-deployment.yaml中通过环境变量WORDPRESS_DB_HOST 指定。环境变量 WORDPRESS_DB_PASSWORD 定义数据库用户 root 的密码，此密码在 wordpress/db/mariadb-secret.yaml 清单文件中定义了。当网站正常运行之后，建议删除调试环境变量 WORDPRESS_DEBUG。

注意：文件 wordpress-deployment.yaml 中的容器镜像 wordpress 要从官网下拉，但是教室里没法访问外网，所以还得手工导入（从一台能访问官网的电脑上下拉并导出到文件 wordpress_latest.tar 中，然后采用命令

“ctr -n k8s.io images import wordpress_latest.tar --platform linux/amd64”导入到全部工作节点上）。

~~2. 准备好容器镜像~~

~~首先在 worker1 上手工下拉 wordpress 镜像，然后导出并传到 worker2 和 worker3 上再分别导入：~~

```
[root@worker1 ~]# ctr -n k8s.io images pull --platform linux/amd64 docker.io/library/wordpress:latest
[root@worker1 ~]# ctr -n k8s.io images export --platform linux/amd64 wordpress_latest.tar \
docker.io/library/wordpress:latest
[root@worker1 ~]# scp wordpress_latest.tar worker2:.
[root@worker1 ~]# scp wordpress_latest.tar worker3:.

[root@worker2 ~]# ctr -n k8s.io images import --platform linux/amd64 wordpress_latest.tar
[root@worker3 ~]# ctr -n k8s.io images import --platform linux/amd64 wordpress_latest.tar
```

3. 创建各个资源对象：

```
[root@master1 wp]# kubectl create -f wordpress-pv.yaml
persistentvolume/wordpress-pv created
[root@master1 wp]# kubectl create -f wordpress-pvc.yaml
persistentvolumeclaim/wordpress-pvc created
[root@master1 wp]# kubectl create -f wordpress-deployment.yaml
deployment.apps/wordpress created
[root@master1 wp]# kubectl create -f wordpress-service.yaml
service/wordpress-service created

[root@master1 wp]# kubectl get pods -o wide -n wordpress
NAME                         READY   STATUS    RESTARTS        AGE     IP            NODE      NOMINATED NODE   READINESS GATES
mariadb-wp-0                 1/1     Running   1 (3h55m ago)   4h27m   10.244.3.8    worker3   <none>           <none>
wordpress-6cf4d8cd89-2q4hr   1/1     Running   0               9m33s   10.244.2.26   worker2   <none>           <none>

[root@master1 wp]# kubectl get svc -n wordpress
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
mariadb-service     ClusterIP   10.96.200.48    <none>        3306/TCP       4h28m
wordpress-service   NodePort    10.96.74.226    <none>        80:30007/TCP   10m
```

由于最新的 wordpress 有一个 bug，需要手工创建数据库：

```
[root@master1 wp]# mysql -h 10.96.200.48 -uroot -pA1b2C3
mysql> CREATE DATABASE wordpress_db DEFAULT CHARACTER SET utf8mb4;
mysql> exit
```

最后检查所有资源是否就绪：

```
[root@master1 ~]# kubectl get all -n wordpress -o wide --show-labels
NAME                             READY   STATUS    RESTARTS      AGE   IP            NODE      NOMINATED NODE   READINESS GATES   LABELS
pod/mariadb-wp-0                 1/1     Running   2 (89m ago)   28h   10.244.2.86   worker2   <none>           <none>            app=mariadb-
pod,apps.kubernetes.io/pod-index=0,controller-revision-hash=mariadb-wp-55fbcf6748,statefulset.kubernetes.io/pod-name=mariadb-wp-0
pod/wordpress-65cb68bcf5-tnmd4   1/1     Running   0             56m   10.244.1.95   worker1   <none>           <none>            app=wordpress,pod-
template-hash=65cb68bcf5

NAME                        TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE     SELECTOR          LABELS
service/mariadb-service     ClusterIP   10.96.54.218   <none>        3306/TCP       6h45m   app=mariadb-pod   app=mariadb
service/wordpress-service   NodePort    10.96.107.63   <none>        80:30001/TCP   51m     app=wordpress     <none>

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES      SELECTOR        LABELS
deployment.apps/wordpress   1/1     1            1           56m   wordpress    wordpress   app=wordpress   <none>

NAME                                   DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES      SELECTOR                                                  LABELS
replicaset.apps/wordpress-65cb68bcf5   1         1         1       56m   wordpress    wordpress   app=wordpress,pod-template-hash=65cb68bcf5
app=wordpress,pod-template-hash=65cb68bcf5

NAME                           READY   AGE   CONTAINERS   IMAGES    LABELS
statefulset.apps/mariadb-wp    1/1     28h   mariadb      mariadb   <none>
```

注意 pod 的标签和服务的标签选择器是否匹配。

之后可浏览 http://<任何工作节点的 ip>:30001 访问网站。参考图 17-7。

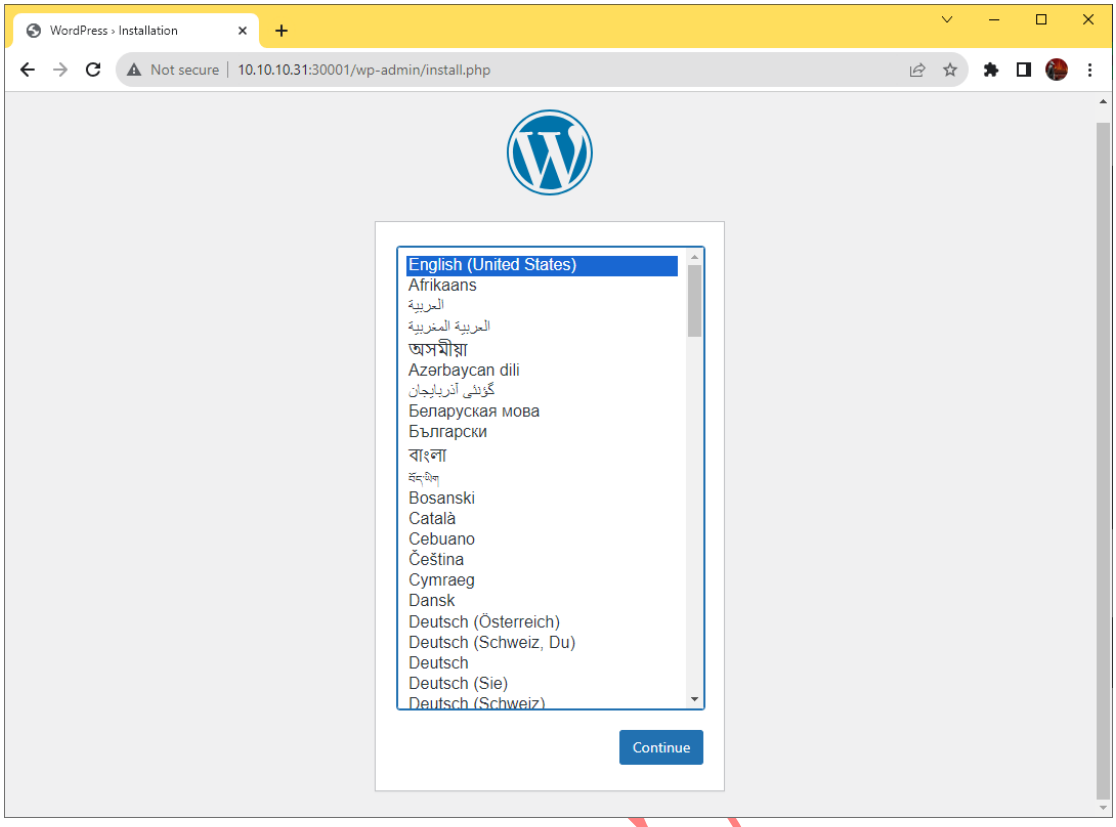


图 17-7 首次登录 wordpress

此后可参考“§3.1.6 容器组合”小节中的“课堂实战练习：5 分钟部署 wordpress”部分进行后续操作。

如果页面反馈错误信息“Error establishing a database connection”表明没法登陆数据库，请参考“§23.12 无法访问 wordpress 页面”排错。

§17.5 网络连通性

我们能在 busybox 所在 pod 内 ping 通外网，参考上面的命令 ping 9.9.9.9，那么 Pod 连通外网的路径到底是什么？本节我们一起来窥探究竟。加如 busybox 跑在 worker1 上，那么就在 worker1 上操作。

1. 有哪些网络名字空间：

```
[root@worker1 ~]# lsns -t net
        NS TYPE NPROCS   PID USER     NETNSID NSFS                                COMMAND
4026531840 net     227     1 root  unassigned                                     /usr/lib/systemd/systemd --switched-root --system --deserialize 31
4026532771 net       2  2080 65535         0 /run/netns/cni-b99101ef-e3b1-3f3e-ab30-34451880efa5 /pause
4026532841 net       2  2208 65535         1 /run/netns/cni-bd2e342c-e5f7-abaa-9618-f5b53d8a47a6 /pause
4026532904 net       2  2378 65535         2 /run/netns/cni-456274ee-a936-d3f3-d209-f87e6fae39ea /pause
```

每个 pod 都跑在各自的网络名字空间内，且每个 pod 都有一个 pause 容器，对应宿主机上的/pause 进程。上面的命令输出了四个网络名字空间，最后一行的进程号为 2378，应该就是 busybox 对应的网络名字空间。

2. 查看 2378 进程号使用的网络名字空间名称：

```
[root@worker1 ~]# ip netns identify 2378
cni-456274ee-a936-d3f3-d209-f87e6fae39ea
```

https://www.ver

3. 查看网络名字空间内的网卡参数（当然也可以登录到 pod 中查看）：

```
[root@worker1 ~]# ip netns exec cni-456274ee-a936-d3f3-d209-f87e6fae39ea ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether b2:86:a6:1f:ba:66 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.1.7/24 brd 10.244.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::b086:a6ff:fe1f:ba66/64 scope link
       valid_lft forever preferred_lft forever
```

发现 2 号接口 eth0@if8，表示 eth0 网卡连接的对端为 8 号网卡，配置了 ip 地址 10.244.1.7。继续查看路由表：

```
[root@worker1 ~]# ip netns exec cni-456274ee-a936-d3f3-d209-f87e6fae39ea ip r
default via 10.244.1.1 dev eth0
10.244.0.0/16 via 10.244.1.1 dev eth0
10.244.1.0/24 dev eth0 proto kernel scope link src 10.244.1.7
```

可见默认路由网关就是 10.244.1.1，且 10.244.0.0/16 网段也是经默认网关到达。

4. 查看宿主机上的网卡配置：

```
[root@worker1 ~]# ip a
……
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 00:50:56:23:79:02 brd ff:ff:ff:ff:ff:ff
    altname enp3s0
    inet 10.10.10.31/24 brd 10.10.10.255 scope global noprefixroute ens160
       valid_lft forever preferred_lft forever
    inet6 fe80::250:56ff:fe23:7902/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
……
4: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 6a:35:43:d1:0b:d9 brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::6835:43ff:fed1:bd9/64 scope link
       valid_lft forever preferred_lft forever
5: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether ca:48:a1:7a:36:62 brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.1/24 brd 10.244.1.255 scope global cni0
       valid_lft forever preferred_lft forever
    inet6 fe80::c848:a1ff:fe7a:3662/64 scope link
       valid_lft forever preferred_lft forever
……
8: veth7b0d3a85@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP group default
    link/ether e6:cb:06:60:2a:65 brd ff:ff:ff:ff:ff:ff link-netns cni-456274ee-a936-d3f3-d209-f87e6fae39ea
    inet6 fe80::e4cb:6ff:fe60:2a65/64 scope link
       valid_lft forever preferred_lft forever

[root@worker1 ~]# ip link show master cni0                                    #查看 cni0 桥上的网络接口
6: vethc27f5660@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT group default
    link/ether 7e:79:fe:01:8a:83 brd ff:ff:ff:ff:ff:ff link-netns cni-b99101ef-e3b1-3f3e-ab30-34451800efa5
7: vethefaad6db@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT group default
    link/ether d6:64:ac:16:71:3a brd ff:ff:ff:ff:ff:ff link-netns cni-bd2e342c-e5f7-abaa-9618-f5b53d8a47a6
8: veth7b0d3a85@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT group default
    link/ether e6:cb:06:60:2a:65 brd ff:ff:ff:ff:ff:ff link-netns cni-456274ee-a936-d3f3-d209-f87e6fae39ea
```

注意 8 号网卡 veth7b0d3a85@if2，名字为 veth7b0d3a85，对端为 cni-456274ee-a936-d3f3-d209-f87e6fae39ea 网络名字空间内的 2 号网卡（就是 busybox 那个 Pod 所在的网络名字空间），8 号网卡接入 cni0 网桥，cni0 网桥配置了 ip 地址 10.244.1.1，这个 ip 就是上面第 3 步中发现的默认网关。

5. 查看宿主机上的路由表：

```
[root@worker1 ~]# ip r
default via 10.10.10.2 dev ens160 proto static metric 100
10.10.10.0/24 dev ens160 proto kernel scope link src 10.10.10.31 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 dev cni0 proto kernel scope link src 10.244.1.1
10.244.2.0/24 via 10.244.2.0 dev flannel.1 onlink
```

默认网关 10.10.10.2 就是 VMWare NAT 网的网关，是去外网的。worker1 节点的 pod 网段为 10.244.1.0/24，所以 worker1 上的所有 Pod 的 ip 都落在此网段内。worker2 节点的 Pod 网段为 10.244.2.0/24。

本小节讨论 pod 连接外网，因此与默认路由有关，以后讨论各个节点之间的 pod 通讯时再来看其他路由。

最后总结 pod 的外出网络拓扑图请参考图 17-8。

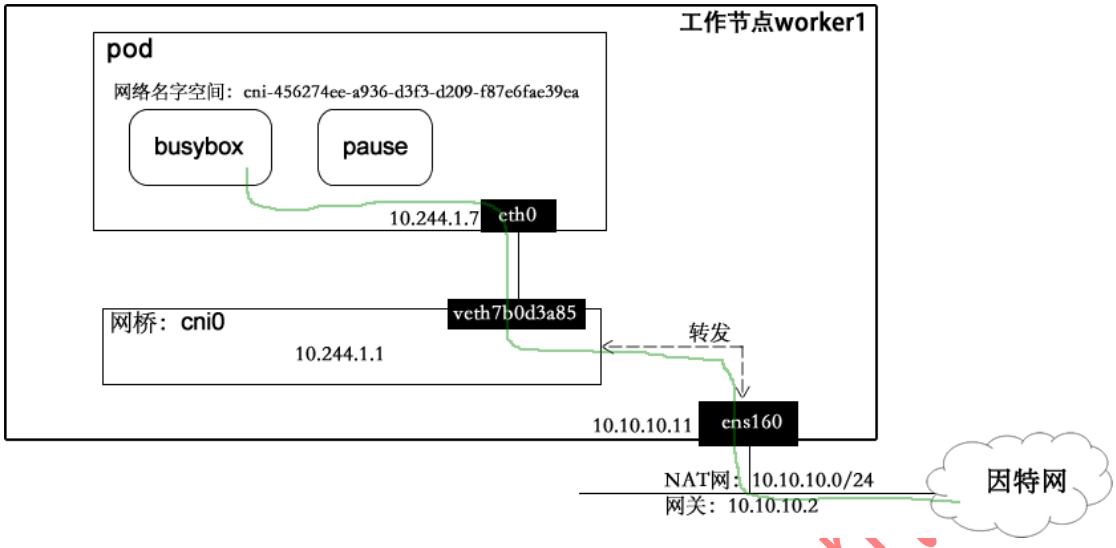


图 17-8 pod 外出网络

```
[root@worker1 ~]# ip  a show cni0
5: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether f2:33:fa:21:8a:36 brd ff:ff:ff:ff:ff:ff
    inet 10.244.1.1/24 brd 10.244.1.255 scope global cni0
    ......

[root@worker2 ~]# ip a show cni0
5: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether 22:97:39:68:a7:91 brd ff:ff:ff:ff:ff:ff
    inet 10.244.2.1/24 brd 10.244.2.255 scope global cni0
    ......
```

由于在每个工作节点上各运行了一个 pod，且 pod 的地址分别是 10.244.1.8（worker1 上）和 10.244.2.2(worker2 上)，两个工作节点上各创建了一个网桥 cni0，网桥配置的 Ip 分别是 10.244.1.1/24 和 10.244.2.1/24，由此可见每个节点单独使用一个子网（从 10.244.0.0/16 拆分子网 10.244.*.0/24）。pod 网络拓扑见图 17-9。

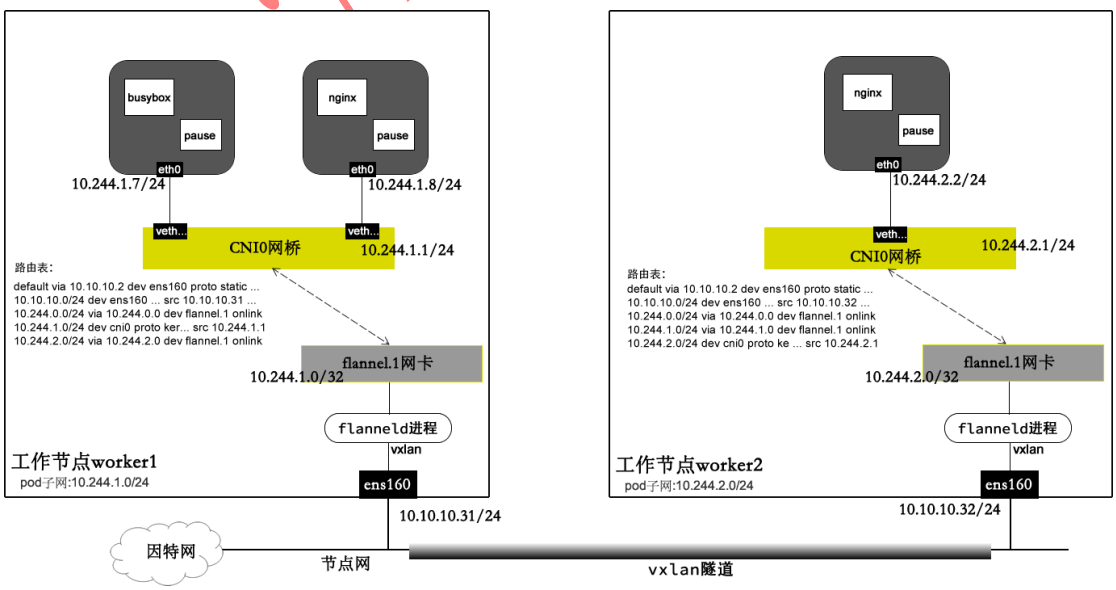


图 17-9 pod 网络

全部 pod 之间的网络通路是这样的：同一个 pod 内的容器通过 localhost 互通，同一节点上的 Pod 通过 cni0 网桥互通，不同节点上的 pod 通过各个节点上的 flannel.1 虚拟网卡经 flanneld 进程建立 vxlan 隧道互通。下面通过实战验证：

1. 验证 worker1 上的两个 pod 之间的通信：

```
进入 busybox 不断 Ping 另一个 pod 的 ip:
[root@master1 ~]# kubectl exec busybox -it -- sh
/ # ping 10.244.1.8
PING 10.244.1.8 (10.244.1.8): 56 data bytes
64 bytes from 10.244.1.8: seq=0 ttl=62 time=0.394 ms
64 bytes from 10.244.1.8: seq=1 ttl=62 time=0.913 ms
64 bytes from 10.244.1.8: seq=2 ttl=62 time=0.953 ms
……

查看 cni0 网桥上的接口（每个 pod 都接到了 cni0 上）:
[root@worker1 ~]# ip link show master cni0
6: veth7718cf15@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT group default
    link/ether be:9f:59:42:73:eb brd ff:ff:ff:ff:ff:ff link-netns cni-e73831c6-6b9c-a5bb-2583-d470ca32400d
7: vethf9faf6f6@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP mode DEFAULT group default
    link/ether b2:c7:7a:d4:9c:69 brd ff:ff:ff:ff:ff:ff link-netns cni-f6afaace-0e8d-6762-7a40-623f15f5f5ae

[root@worker1 ~]# tcpdump -ni vethf9faf6f6 -p icmp
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on vethf9faf6f6, link-type EN10MB (Ethernet), snapshot length 262144 bytes
16:26:06.466730 IP 10.244.1.7 > 10.244.1.8: ICMP echo request, id 49, seq 56, length 64
16:26:06.466763 IP 10.244.1.8 > 10.244.1.7: ICMP echo reply, id 49, seq 56, length 64
16:26:07.477053 IP 10.244.1.7 > 10.244.1.8: ICMP echo request, id 49, seq 57, length 64
^C
[root@worker1 ~]# tcpdump -ni veth7718cf15 -p icmp
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on veth7718cf15, link-type EN10MB (Ethernet), snapshot length 262144 bytes
16:26:23.638280 IP 10.244.1.7 > 10.244.1.8: ICMP echo request, id 49, seq 73, length 64
16:26:23.638303 IP 10.244.1.8 > 10.244.1.7: ICMP echo reply, id 49, seq 73, length 64
^C

在 worker1 对 cni0 抓包（没有抓到）:
[root@worker1 ~]# tcpdump -ni cni0 -p icmp
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on cni0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C
```

证明同一个节点上的 Pod 之间通信直接经由 cni0 网桥完成。

2. 验证不同节点上的 pod 之间的通讯：

```
[root@master1 ~]# kubectl exec busybox -it -- sh
/ # ping 10.244.2.2
PING 10.244.2.2 (10.244.2.2): 56 data bytes
64 bytes from 10.244.2.2: seq=0 ttl=64 time=0.079 ms
64 bytes from 10.244.2.2: seq=1 ttl=64 time=0.113 ms
64 bytes from 10.244.2.2: seq=2 ttl=64 time=0.096 ms
……
```

下面分别在 worker1 和 worker2 抓包 cni0、flannel.1 和 ens160：

```
[root@worker1 ~]# tcpdump -ni cni0 -p icmp
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on cni0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
16:35:20.786386 IP 10.244.1.7 > 10.244.2.2: ICMP echo request, id 56, seq 36, length 64
16:35:20.787730 IP 10.244.2.2 > 10.244.1.7: ICMP echo reply, id 56, seq 36, length 64
16:35:21.796433 IP 10.244.1.7 > 10.244.2.2: ICMP echo request, id 56, seq 37, length 64
^C

[root@worker1 ~]# tcpdump -ni flannel.1 -p icmp
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on flannel.1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
16:40:15.846793 IP 10.244.1.7 > 10.244.2.2: ICMP echo request, id 56, seq 328, length 64
16:40:15.847380 IP 10.244.2.2 > 10.244.1.7: ICMP echo reply, id 56, seq 328, length 64
16:40:16.857801 IP 10.244.1.7 > 10.244.2.2: ICMP echo request, id 56, seq 329, length 64
^C

[root@worker1 ~]# tcpdump -ni ens160 -p udp                                    #抓到叠加网络包了
dropped privs to tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on ens160, link-type EN10MB (Ethernet), snapshot length 262144 bytes
16:40:51.196520 IP 10.10.10.31.38223 > 10.10.10.32.otv: OTV, flags [I] (0x08), overlay 0, instance 1
IP 10.244.1.7 > 10.244.2.2: ICMP echo request, id 56, seq 363, length 64
16:40:51.196838 IP 10.10.10.32.35974 > 10.10.10.31.otv: OTV, flags [I] (0x08), overlay 0, instance 1
IP 10.244.2.2 > 10.244.1.7: ICMP echo reply, id 56, seq 363, length 64
^C
```

同样可以在 worker2 上抓取 ens160、flannel.1 和 cni0 的数据包，这里不在演示了。

§17.6 验收

K8s 集群搭建好了 60 分，busybox 应用 5 分，web 应用 10 分，数据库 10 分，wordpress 应用 15 分。

1. 检查集群：

```
[root@master1 ~]# kubectl get pod -A
NAMESPACE              NAME                                         READY   STATUS    RESTARTS          AGE
......
kube-flannel           kube-flannel-ds-8ztck                        1/1     Running   25 (2m40s ago)    48d
kube-flannel           kube-flannel-ds-fggjg                        1/1     Running   33 (2m40s ago)    48d
kube-flannel           kube-flannel-ds-nscq4                        1/1     Running   24 (2m57s ago)    48d
kube-flannel           kube-flannel-ds-wx98w                        1/1     Running   24 (3m1s ago)     48d
kube-system            coredns-7cc97dffdd-r7j4l                     1/1     Running   24 (3m6s ago)     50d
kube-system            coredns-7cc97dffdd-vzd8w                     1/1     Running   24 (2m57s ago)    48d
kube-system            etcd-master1                                 1/1     Running   22 (3m8s ago)     50d
kube-system            kube-apiserver-master1                       1/1     Running   22 (3m8s ago)     50d
kube-system            kube-controller-manager-master1              1/1     Running   30 (3m8s ago)     50d
kube-system            kube-proxy-68vp4                             1/1     Running   28 (3m8s ago)     50d
kube-system            kube-proxy-j9lrk                             1/1     Running   26 (3m1s ago)     48d
kube-system            kube-proxy-q4k8p                             1/1     Running   26 (3m6s ago)     48d
kube-system            kube-proxy-xtg8j                             1/1     Running   26 (2m57s ago)    48d
kube-system            kube-scheduler-master1                       1/1     Running   29 (3m8s ago)     50d
kubernetes-dashboard   kubernetes-dashboard-api-569bb548f-8qwsz     1/1     Running   11 (3m1s ago)     22d
kubernetes-dashboard   kubernetes-dashboard-auth-766c45c666-tqd6t   1/1     Running   11 (2m57s ago)    22d
kubernetes-dashboard   kubernetes-dashboard-kong-9849c64bd-5f9mb    1/1     Running   11 (3m6s ago)     22d
kubernetes-dashboard   kubernetes-dashboard-metrics-scraper-7685fd8b77-ws9l2   1/1   Running   11 (3m6s ago)   22d
kubernetes-dashboard   kubernetes-dashboard-web-5c9f966b98-5rxgk    1/1     Running   11 (3m1s ago)     22d
......
```

2. 检查应用（busybox、mariadb、www）：

```
[root@master1 ~]# kubectl get all -n myspace -o wide
NAME                                READY   STATUS    RESTARTS      AGE     IP             NODE      NOMINATED NODE   READINESS
GATES
pod/busybox-pod                     1/1     Running   7 (13m ago)   18d     10.244.2.115   worker2   <none>           <none>
pod/mariadb-sts-0                   1/1     Running   4 (13m ago)   6d23h   10.244.1.96    worker1   <none>           <none>
pod/nginx-www-6547745c89-2jqlf      1/1     Running   5 (13m ago)   13d     10.244.4.100   worker3   <none>           <none>
pod/nginx-www-6547745c89-g2s7c      1/1     Running   5 (13m ago)   13d     10.244.1.98    worker1   <none>           <none>

NAME                       TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)          AGE     SELECTOR
service/mariadb-service    ClusterIP      10.96.207.191   <none>         3306/TCP         6d23h   app=mariadb
service/web-service        NodePort       10.96.211.87    <none>         8080:30088/TCP   13d     app=nginx-www
service/web-service-lb     LoadBalancer   10.96.110.147   10.10.10.100   8081:31859/TCP   13d     app=nginx-www

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES          SELECTOR
deployment.apps/mybusybox   1/1     1            1           18d   busybox      busybox         app=mybusybox
deployment.apps/nginx-www   2/2     2            2           14d   nginx        nginx:alpine    app=nginx-www

NAME                                   DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES          SELECTOR
replicaset.apps/mybusybox-768657c9c7   1         1         1       18d   busybox      busybox         app=mybusybox,pod-
template-hash=768657c9c7
replicaset.apps/nginx-www-6547745c89   2         2         2       14d   nginx        nginx:alpine    app=nginx-www,pod-
template-hash=6547745c89

NAME                             READY   AGE     CONTAINERS   IMAGES
statefulset.apps/mariadb-sts     1/1     6d23h   mariadb      mariadb

[root@master1 ~]# curl 10.10.10.100:8081
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
```

```
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

3. 检查应用（wordpress）：

```
[root@master1 ~]# kubectl get all -n wordpress -o wide
NAME                             READY   STATUS    RESTARTS      AGE     IP             NODE      NOMINATED NODE   READINESS GATES
pod/mariadb-wp-0                 1/1     Running   1 (18m ago)   4d18h   10.244.4.97    worker3   <none>           <none>
pod/wordpress-785bc9cc8d-gbw7m   1/1     Running   1 (18m ago)   4d17h   10.244.2.113   worker2   <none>           <none>

NAME                        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE     SELECTOR
service/mariadb-service     ClusterIP   10.96.107.217   <none>        3306/TCP       4d18h   app=mariadb-pod
service/wordpress-service   NodePort    10.96.108.116   <none>        80:30001/TCP   4d18h   app=wordpress

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS   IMAGES      SELECTOR
deployment.apps/wordpress   1/1     1            1           4d17h   wordpress    wordpress   app=wordpress

NAME                                   DESIRED   CURRENT   READY   AGE     CONTAINERS   IMAGES      SELECTOR
replicaset.apps/wordpress-785bc9cc8d   1         1         1       4d17h   wordpress    wordpress   app=wordpress,pod-template-
hash=785bc9cc8d

NAME                           READY   AGE     CONTAINERS   IMAGES
statefulset.apps/mariadb-wp    1/1     4d18h   mariadb      mariadb
```

浏览器中能打开网址 http://<任何一个节点的 ip>:30001，能看到一篇文章，文章内容不限。

4. 问询几个问题。

§17.7 部署 Gemma4

环境假设：K8s 集群已部署完成，GPU 节点为 8x GPU 服务器，模型已下载在节点本地 models/gemma-4-31B-it-uncensored-heretic 目录，本次演示仅使用其中 4 块 GPU。

§17.7.1 环境检查

1. 确认 GPU 节点已被 K8s 正确识别

```
kubectl get nodes -o json | jq '.items[].status.capacity."nvidia.com/gpu"'
```

应该看到“8”。如果返回 null 或者查不到这个字段，说明 NVIDIA Device Plugin（或 GPU Operator）没有正确部署，需要先确认：

```
kubectl get pods -n kube-system | grep nvidia
# 或者 GPU Operator 独立命名空间
kubectl get pods -n gpu-operator
```

https://www.veryopen

确认 nvidia-device-plugin-daemonset 之类的 Pod 处于 Running 状态，这是后续一切 GPU 调度的前提。

2. 确认模型文件路径

在 GPU 节点上确认路径（假设完整路径是 /data/models/gemma-4-31B-it-uncensored-heretic，按你实际路径替换）：

```
ls models/gemma-4-31B-it-uncensored-heretic
```

应该能看到 config.json、tokenizer.json、模型权重分片文件（.safetensors）等。

3. 创建命名空间：

```
kubectl create namespace llm-demo
```

4. 给 GPU 节点打标签，确保 Pod 稳定调度到它上面：

```
kubectl label node worker1 gpu-node=true
```

§17.7.2 部署 vLLM 推理服务

1. 编写 Deployment

```
[root@master1 ~]# vim vllm-gemma4.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vllm-gemma4
  namespace: llm-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vllm-gemma4
  template:
    metadata:
      labels:
        app: vllm-gemma4
    spec:
      nodeSelector:
        gpu-node: "true"
      containers:
      - name: vllm
        image: vllm/vllm-openai:latest
        args:
          - "--model=/models/gemma-4-31b-it"
          - "--served-model-name=gemma-4-31b"
          - "--tensor-parallel-size=4" ------------------------------------ #把模型权重按张量并行切分到 4 块 GPU 上协同计算
          - "--gpu-memory-utilization=0.90"
          - "--max-model-len=8192"
          - "--dtype=bfloat16"
          - "--host=0.0.0.0"
          - "--port=8000"
        ports:
        - containerPort: 8000
        resources:
          limits:
            nvidia.com/gpu: 4 ------------------------------------------------------------------------------#限制只使用 4 块 GPU 显卡
          requests:
            nvidia.com/gpu: 4
        volumeMounts:
        - name: model-storage
          mountPath: /models/gemma-4-31b-it
        - name: shm
          mountPath: /dev/shm
      volumes:
      - name: model-storage
        hostPath:
          path: /data/models/gemma-4-31B-it-uncensored-heretic
          type: Directory
      - name: shm  ------------------------------------------------------------- #vLLM 底层用 NCCL 做多卡通信（通过共享内存）
        emptyDir:
```

```
          medium: Memory
          sizeLimit: 8Gi
---
apiVersion: v1
kind: Service
metadata:
  name: vllm-gemma4-svc
  namespace: llm-demo
spec:
  selector:
    app: vllm-gemma4
  ports:
  - port: 8000
    targetPort: 8000
  type: ClusterIP

[root@master1 ~]# kubectl apply -f vllm-gemma4.yaml
```

2. 验证部署

```
[root@master1 ~]# kubectl get pods -n llm-demo -w
```

首次启动会比较慢（加载 31B 参数模型 + 初始化多卡 NCCL 通信），观察日志：

```
[root@master1 ~]# kubectl logs -n llm-demo -f deploy/vllm-gemma4
```

看到类似 Uvicorn running on http://0.0.0.0:8000 说明服务已经就绪。

3. 验证 GPU 分配（现场演示的好素材）

```
[root@master1 ~]# kubectl exec -n llm-demo deploy/vllm-gemma4 -- nvidia-smi
```

能直接看到容器内 nvidia-smi 只列出 4 块 GPU（而不是宿主机全部 8 块），这是 K8s 设备插件机制在起作用——容器看到的 GPU 视图已经被限定在分配给它的那部分。

4. 用 port-forward 快速测试 API

```
[root@master1 ~]# kubectl port-forward -n llm-demo svc/vllm-gemma4-svc 8000:8000
```

另开终端：

```
[root@master1 ~]# curl http://localhost:8000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gemma-4-31b",
    "messages": [{"role": "user", "content": "用一句话介绍一下 Kubernetes"}]
  }'
```

能收到正常的 JSON 格式回复，说明 vLLM 服务已经跑通。

§17.7.3 部署 Open WebUI

1. 编写 Deployment

```
[root@master1 ~]# vim open-webui.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: open-webui
  namespace: llm-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: open-webui
  template:
    metadata:
      labels:
        app: open-webui
    spec:
      containers:
      - name: open-webui
        image: ghcr.io/open-webui/open-webui:main
        env:
        - name: OPENAI_API_BASE_URL
```

https://www.veryopen.org

```
          value: "http://vllm-gemma4-svc.llm-demo.svc.cluster.local:8000/v1"
        - name: OPENAI_API_KEY
          value: "not-needed-for-vllm"
        - name: WEBUI_AUTH
          value: "false"
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: webui-data
          mountPath: /app/backend/data
      volumes:
      - name: webui-data
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: open-webui-svc
  namespace: llm-demo
spec:
  selector:
    app: open-webui
  ports:
  - port: 80
    targetPort: 8080
  type: NodePort


[root@master1 ~]# kubectl apply -f open-webui.yaml
```

2. 访问验证

```
[root@master1 ~]# kubectl get svc -n llm-demo open-webui-svc
```

拿到 NodePort 之后，浏览器访问：http://<GPU 服务器 IP>:<NodePort>

首次进入需要在设置里确认模型列表能读取到 gemma-4-31b（因为 Open WebUI 会自动从 vLLM 的 /v1/models 接口拉取可用模型列表），选中后就能开始对话。

§17.7.4 清理

```
[root@master1 ~]# kubectl delete -f open-webui.yaml
[root@master1 ~]# kubectl delete -f vllm-gemma4.yaml
[root@master1 ~]# kubectl delete namespace llm-demo
```

§17.8 Wasm 应用

§17.8.1 环境准备

1. 安装 Rust 及 Wasm 编译目标：

```
[root@master1 ~]# dnf install gcc -y
[root@master1 ~]# curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y --default-toolchain \
 stable
[root@master1 ~]# source "$HOME/.cargo/env"

[root@master1 ~]# rustup target add wasm32-wasip2
```

2. 安装 Spin CLI（由 WebAssembly 官方发布的构建 serverless 应用的开源工具）：

```
[root@master1 ~]# curl -fsSL https://developer.fermyon.com/downloads/install.sh | bash
[root@master1 ~]# install -m 0755 ./spin /usr/local/bin/spin
```

https://www.veryopen.

```
[root@master1 ~]# spin --version
```

§17.8.2 编写 Hello World 服务

1. 用模板创建项目

```
[root@master1 ~]# spin new -t http-rust hello-wasm --accept-defaults
[root@master1 ~]# cd hello-wasm
```

上面第一条命令会从 github.com 下载模板，一次失败，请尝试多次，直到成功。

会生成如下的目录结构：

```
hello-wasm/
├── Cargo.toml
├── spin.toml       # Spin 应用清单，描述路由/组件
└── src/
    └── lib.rs       # 业务代码
```

2. 查看/修改 src/lib.rs

模板默认生成的代码已经是一个可用的 HTTP handler，改成我们自己的问候语：

```
use spin_sdk::http::{IntoResponse, Request, Response};
use spin_sdk::http_service;

/// A simple Spin HTTP component.
#[http_service]
async fn handle_hello_wasm(req: Request) -> anyhow::Result<impl IntoResponse> {
    println!("Handling request to {:?}", req.headers().get("spin-full-url"));
    Ok(Response::builder()
        .status(200)
        .header("content-type", "text/plain")
        .body("Hello World from Rust + WebAssembly on Kubernetes!\ns".to_string()))
}
```

3. 查看 spin.toml：

```
spin_manifest_version = 2

[application]
name = "hello-wasm"
version = "0.1.0"
authors = ["root"]
description = ""

[[trigger.http]]
route = "/..."
component = "hello-wasm"

[component.hello-wasm]
source = "target/wasm32-wasip2/release/hello_wasm.wasm"
allowed_outbound_hosts = []
[component.hello-wasm.build]
command = "cargo build --target wasm32-wasip2 --release"
watch = ["src/**/*.rs", "Cargo.toml"]
```

这份清单描述了：这个组件监听哪个路由、编译产物路径、构建命令——spin.toml + 编译出的 .wasm 文件，就是后面要打包成 OCI 镜像的全部内容。

§17.8.3 本地编译与验证

1. 编译

```
[root@master1 hello-wasm]# spin build
……
   Compiling hello-wasm v0.1.0 (/root/hello-wasm)
    Finished `release` profile [optimized] target(s) in 24.69s
Finished building all Spin components
```

2. 本地运行验证

```
[root@master1 hello-wasm]# spin up --listen 0.0.0.0:3000
```

另开一个终端测试：

```
curl http://localhost:3000
```

应该看到：

```
Hello World from Rust + WebAssembly on Kubernetes!
```

验证通过后 Ctrl+C 停掉本地测试进程，准备进入打包流程。

4. 打包成 OCI 镜像并推送到本地仓库

这一步是 Wasm 和普通容器打通的关键——Spin 会把 spin.toml 清单和编译出的 .wasm 文件，包装成一个符合 OCI 镜像规范的 Artifact，可以直接推送到你们之前搭建的私有仓库（比如 10.8.20.8:5000）。

4.1 推送到本地镜像仓库：

```
[root@master1 hello-wasm]# spin registry push --build --insecure 10.8.20.8:5000/wasm/hello-wasm:v0.1.0
```

- --build：推送前自动先跑一次 spin build，确保推的是最新编译产物。
- --insecure：因为你的本地仓库是 HTTP 明文（之前配置 insecure-registries 那一套），Spin CLI 默认只信任 HTTPS，这里要显式声明允许非安全连接，和 Docker/Podman 里 insecure-registries/insecure = true 是同一个道理。
- 10.8.20.8 替换成真实的跑了镜像仓库的那台电脑的 ip。

4.2 验证镜像已经进入仓库

```
curl -s http://10.8.20.8:5000/v2/wasm/hello-wasm/tags/list
```

应该能看到：

```
{"name":"wasm/hello-wasm","tags":["v0.1.0"]}
```

注意：这个镜像和你们之前处理的 nginx/busybox 那些镜像结构完全不同——它不含 Linux 文件系统层，只包含 .wasm 字节码和清单，体积通常只有几百 KB 到几 MB，这也是 Wasm 相比传统容器镜像“轻”的直观体现。

5. K8s 节点安装 containerd-shim-spin

5.1 下载 shim 二进制

```
在每一个 K8s 节点上执行（教学环境可以先在一个节点上跑通，再同步到其他节点）：
SHIM_VERSION="v0.17.0"
wget https://github.com/spinkube/containerd-shim-spin/releases/download/${SHIM_VERSION}/containerd-shim-spin-v2

chmod +x containerd-shim-spin-v2
sudo mv containerd-shim-spin-v2 /usr/local/bin/
```

5.2 在 containerd 配置里注册这个 runtime

编辑 /etc/containerd/config.toml，在 CRI 插件的 runtimes 段里加一个新条目：

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.spin]
  runtime_type = "io.containerd.spin.v2"
```

重启 containerd 使其生效：

```
sudo systemctl restart containerd
```

5.3 验证 containerd 已识别这个运行时

```
sudo crictl info | grep -A5 spin
```

https://www.veryopen.org

能看到刚才配置的 spin runtime 条目，说明 containerd 层面已经就绪。

6. 创建 RuntimeClass

告诉 K8s 调度器："有一种 Pod 要用 spin 这个运行时处理，而不是普通的 runc"：

```
cat > runtimeclass-spin.yaml << 'EOF'
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: wasmtime-spin
handler: spin
EOF

kubectl apply -f runtimeclass-spin.yaml
kubectl get runtimeclass
```

7. 部署到 K8s

7.1 编写 Deployment + Service

```
cat > hello-wasm-deploy.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-wasm
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello-wasm
  template:
    metadata:
      labels:
        app: hello-wasm
    spec:
      runtimeClassName: wasmtime-spin
      containers:
      - name: hello-wasm
        image: 10.8.20.8:5000/wasm/hello-wasm:v0.1.0
        command: ["/"]
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: hello-wasm-svc
spec:
  selector:
    app: hello-wasm
  ports:
  - port: 80
    targetPort: 80
  type: NodePort
EOF

kubectl apply -f hello-wasm-deploy.yaml
```

几个和普通容器 YAML 不一样的地方：

- runtimeClassName: wasmtime-spin：唯一的关键差异点，其他字段（image、ports、replicas）和写一个普通 Nginx Deployment 没有任何区别——这正好呼应了之前讲的"K8s 调度 Wasm 的操作体感和调度普通容器几乎一样，只是底层执行引擎换了"
- image 字段指向的正是你在第四节推送的那个 OCI Wasm 镜像，K8s/containerd 完全不需要知道"这其实是个 Wasm 模块"——对调度器来说，这就是一个普通的镜像拉取+运行请求，差异被 shim 这一层完全屏蔽掉了

7.2 验证部署

```
kubectl get pods -o wide
kubectl get svc hello-wasm-svc
```

观察 Pod 启动速度——正常你会发现 Wasm Pod 从 ContainerCreating 到 Running 的速度比同样场景下的普通容器明显更快。

7.3 访问测试

```
NODE_PORT=$(kubectl get svc hello-wasm-svc -o jsonpath='{.spec.ports[0].nodePort}')
curl http://<任意节点IP>:${NODE_PORT}
```

应该看到：

```
Hello World from Rust + WebAssembly on Kubernetes!
```

https://www.veryopen.org

第18章 集群伸/缩

§18.1 伸/缩工作节点

§18.2 伸/缩控制面节点

§18.2.1 增加负载均衡器

§18.2.2 增加控制面节点

https://www.veryopen.org

第19章 Service Meshes

我们使用 kubernetes 官方开发的 Gateway API 来替换 Istio 的 istio-ingressgateway，所以只需最小化安装 Istio 即可。

Istio，Microservice、Sidecar

§19.1 Istio

节点内存不能少于 3Gi。

1. 下载 Istio

从 https://github.com/istio/istio/releases/latest 下载适合自己硬件平台的最新版，比如截至到 2023 年 12 越 30 日，针对 x86 电脑的 linux 版本的文件为 istio-1.20.1-linux-amd64.tar.gz。

```
[root@master1 ~]# wget https://github.com/istio/istio/releases/download/1.20.1/istio-1.20.1-linux-amd64.tar.gz
[root@master1 ~]# tar -xf istio-1.20.1-linux-amd64.tar.gz
```

或者直接采用下面的命令下载并解压：

```
[root@master1 ~]# curl -L https://istio.io/downloadIstio | sh -
```

然后执行命令：

```
[root@master1 ~]# mv istio-1.20.1 /usr/local/
[root@master1 ~]# echo 'export PATH=$PATH:/usr/local/istio-1.20.1/bin' >> ~/.bashrc
[root@master1 ~]# . ~/.bashrc
```

由于国内访问 github.com 网站不稳定，请多次尝试下载命令。

2. 安装 Istio

如果防火墙开启，请加装如下的 Linux 内核模块。

```
[root@allnodes ~]# cat >/etc/modules-load.d/istio.conf <<EOF
br_netfilter
ip6table_mangle
ip6table_nat
ip6table_raw
iptable_mangle
iptable_nat
iptable_raw
xt_REDIRECT
xt_connmark
xt_conntrack
xt_mark
xt_owner
xt_tcpudp
xt_multiport
bridge
ip6_tables
ip_tables
nf_conntrack
nf_conntrack_ipv4
nf_conntrack_ipv6
nf_nat
nf_nat_ipv4
nf_nat_ipv6
nf_nat_redirect
x_tables
```

https://www.verypen.org

```
EOF

[root@allnodes ~]# shutdown -r now
```

安装 minimal 功能集（不需要 istio-ingressgateway，我们以后采用 Gateway API）：

```
[root@master1 ~]# istioctl install --set profile=minimal -y
✔ Istio core installed
✔ Istiod installed
✔ Installation complete
Made this installation the default for injection and validation.
```

如果出问题，建议多次执行上面的命令。参数“--set profile=<功能集>”安装特定的功能集合，目前系统提供了这些“功能集”：default、demo、minimal、remote、preview、ambient，它们的区别见表 19-1。直接运行命令“istio install -y”表示安装默认的功能集。

表 19-1 istio 功能集

	default	demo	minimal	remote	empty	preview	ambient
核心组件							
istio-egressgateway		√					
istio-ingressgateway	√	√				√	
istiod	√	√	√			√	√
CNI							√
Ztunnel							√

default 适合生产部署，demo 不适合做性能测试，minimal 只安装了控制平面组件，empty 适合作为自定义配置的基本配置文件。

验证安装：

```
[root@master1 ~]# istioctl verify-install
1 Istio control planes detected, checking --revision "default" only
✔ ServiceAccount: istio-reader-service-account.istio-system checked successfully
✔ CustomResourceDefinition: wasmplugins.extensions.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: destinationrules.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: envoyfilters.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: gateways.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: proxyconfigs.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: serviceentries.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: sidecars.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: virtualservices.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: workloadentries.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: workloadgroups.networking.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: authorizationpolicies.security.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: peerauthentications.security.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: requestauthentications.security.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: telemetries.telemetry.istio.io.istio-system checked successfully
✔ CustomResourceDefinition: istiooperators.install.istio.io.istio-system checked successfully
✔ HorizontalPodAutoscaler: istiod.istio-system checked successfully
✔ ClusterRole: istiod-clusterrole-istio-system.istio-system checked successfully
✔ ClusterRole: istiod-gateway-controller-istio-system.istio-system checked successfully
✔ ClusterRoleBinding: istiod-clusterrole-istio-system.istio-system checked successfully
✔ ClusterRoleBinding: istiod-gateway-controller-istio-system.istio-system checked successfully
✔ ConfigMap: istio.istio-system checked successfully
✔ Deployment: istiod.istio-system checked successfully
✔ ConfigMap: istio-sidecar-injector.istio-system checked successfully
✔ MutatingWebhookConfiguration: istio-sidecar-injector.istio-system checked successfully
✔ PodDisruptionBudget: istiod.istio-system checked successfully
✔ ClusterRole: istio-reader-clusterrole-istio-system.istio-system checked successfully
✔ ClusterRoleBinding: istio-reader-clusterrole-istio-system.istio-system checked successfully
✔ Role: istiod.istio-system checked successfully
✔ RoleBinding: istiod.istio-system checked successfully
✔ Service: istiod.istio-system checked successfully
✔ ServiceAccount: istiod.istio-system checked successfully
✔ ValidatingWebhookConfiguration: istio-validator-istio-system.istio-system checked successfully
Checked 15 custom resource definitions
Checked 1 Istio Deployments
✔ Istio is installed and verified successfully
```

查看 Istio 创建的资源对象：

```
[root@master1 ~]# kubectl get all -o wide -n istio-system
NAME                    READY   STATUS    RESTARTS   AGE   IP           NODE      NOMINATED NODE   READINESS GATES
```

verypen.org

```
pod/istiod-6c7b79d8cc-mrrpd   1/1     Running   0          90s   10.244.3.11   worker3   <none>           <none>

NAME             TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)                                 AGE   SELECTOR
service/istiod   ClusterIP   10.96.24.37   <none>        15010/TCP,15012/TCP,443/TCP,15014/TCP   90s   app=istiod,istio=pilot

NAME                     READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES                          SELECTOR
deployment.apps/istiod   1/1     1            1           90s   discovery    docker.io/istio/pilot:1.20.1   istio=pilot

NAME                                DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES                          SELECTOR
replicaset.apps/istiod-6c7b79d8cc   1         1         1       90s   discovery    docker.io/istio/pilot:1.20.1   istio=pilot,pod-template-hash=6c7b79d8cc

NAME                                         REFERENCE           TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/istiod   Deployment/istiod   <unknown>/80%   1         5         1          90s
```

给默认的命名空间 **default** 添加标签，指示 **Istio** 在部署应用的时候，自动注入 **Envoy** 边车代理：

```
[root@master1 ~]# kubectl label namespace default istio-injection=enabled
namespace/default labeled
```

如果你的应用部署在其他名字空间，请做同样的处理。

如果不再使用 **Istio**，请执行下面的命令卸载：

```
[root@master1 ~]# kubectl delete -f /usr/local/istio-1.20.1/samples/addons
[root@master1 ~]# istioctl uninstall -y --purge
[root@master1 ~]# kubectl delete namespace istio-system
[root@master1 ~]# kubectl label namespace default istio-injection-
[root@master1 ~]# sed -i '/istio-1.20.1/d' ~/.bashrc
[root@master1 ~]# rm -rf /usr/local/istio-1.20.1
```

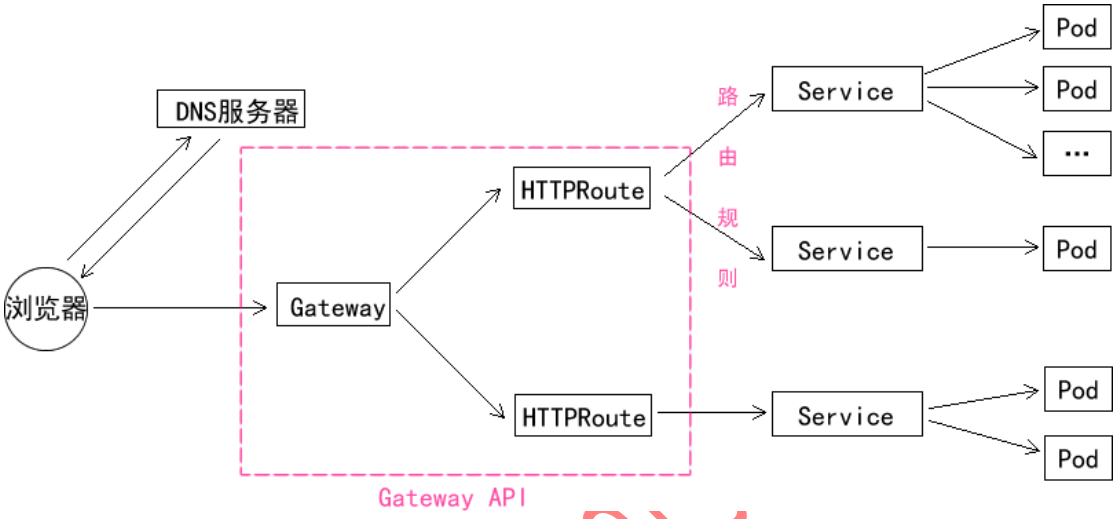
如果需要，可安装一些其他插件（比如 **Kiali** 仪表板、 以及 **Prometheus**、**Grafana**、**Jaeger** 等）：

```
[root@master1 addons]# kubectl apply -f /usr/local/istio-1.20.1/samples/addons
[root@master1 addons]# kubectl rollout status deployment/kiali -n istio-system                                  #查
看上线状态
Waiting for deployment "kiali" rollout to finish: 0 of 1 updated replicas are available...
deployment "kiali" successfully rolled out
```

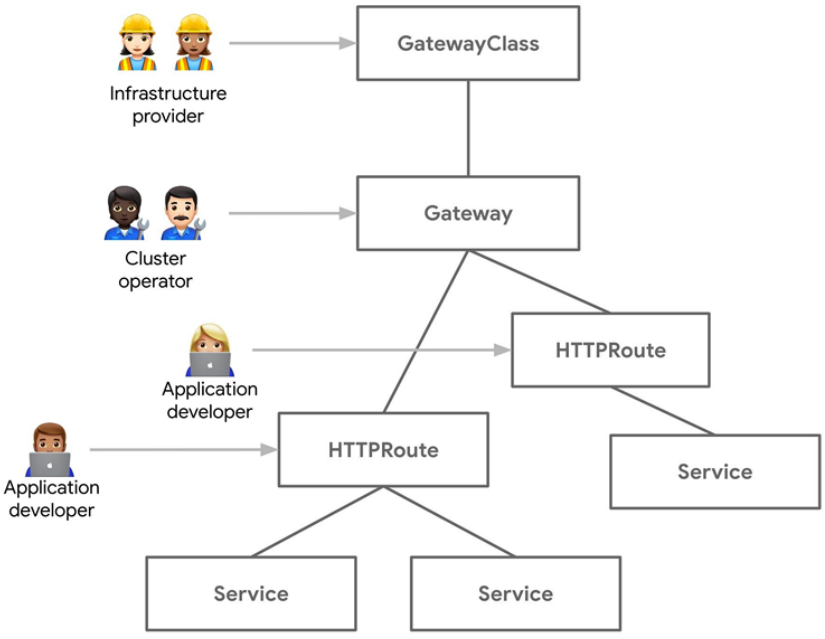
https://www.veryc...

§19.2 Gateway API

https://kubernetes.io/docs/concepts/services-networking/gateway/



Gateway API 涉及的资源对象有：GatewayClass、Gateway、HTTPRoute、 BackendTLSPolicy、GRPCRoute、ReferenceGrant、TCPRoute、TLSRoute、UDPRoute，它们之间的关系见图 17-1 所示。



下面讲解几个最常用的资源对象。

§19.2.1 GatewayClass

https://zhuanlan.zhihu.com/p/609314412

https://gateway-api.sigs.k8s.io/api-types/gatewayclass/

GatewayClass 可作为 Gateway 的模板，对 GatewayClass 的修改不会影响之前基于它创建的 Gateway。GatewayClass 的字段介绍请参考表 17-1。

表 17-1 GatewayClass 资源对象的字段

字段（类型）	说明
apiVersion (string)	目前的值固定为 gateway.networking.k8s.io/v1alpha2
kind (string)	值固定为 GatewayClass
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 11-3
spec (GatewayClassSpec)	定义 GatewayClass 的行为。GatewayClassSpec 对象的字段如下： ● **controllerName** (String)：控制器的名字，必须包含域名和路径，如“example.com/bar”，而“example.com”是不合法的。 ● parametersRef (ParametersReference)：引用配置资源对象，如 ConfigMap 类资源。ParametersReference 对象的字段如下： ○ **group** (String)：被引用资源对象的 GROUP。 ○ **kind** (String)：被引用资源对象的 kind。 ○ **name** (String)：被引用资源对象的名字。 ○ namespace (String)：被引用资源对象的所在的名字空间。 ● description (String)：说明。
status (GatewayClassStatus)	GatewayClass 的当前状态。状态信息只能读。

§19.2.2 Gateway

Gateway 资源对象用于绑定若干监听器到一组 ip 地址，其字段定义参考表 17-2。

表 17-2 Gateway 资源对象的字段

字段（类型）	说明
apiVersion (string)	目前的值固定为 gateway.networking.k8s.io/v1alpha2
kind (string)	值固定为 Gateway
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 11-3
spec (GatewaySpec)	定义 Gateway 的行为。GatewaySpec 对象的字段如下： ● **gatewayClassName** (String)：GatewayClass 资源对象的名字。 ● **listeners** ([]Listener)：若干监听器，每个监听器包含协议、端口和主机名，这三者的组合必须唯一。Listener 对象的字段如下： ○ **name** (String)：监听器名称，比如 veryopen.org、www.veryopen.org 等。名字中不能出现“/”字符。在一个 Gateway 中，监听器名称必须唯一。 ○ hostname (String)：主机名，允许最左侧出现*，不能是 ip 地址，格式为“*|标签[.标签[.标签[…]]]”，“标签”中只能出现小写字母、数字或-，且-不能出现开头和结尾。如*.moodisk.com。本字段可选。 ○ **port** (int32)：监听的端口。 ○ **protocol** (String)：协议，可取值 HTTP、HTTPS、TDS、TCP 或 UDP。 ○ tls (GatewayTLSConfig)：可选字段，TLS 协议对应的配置。GatewayTLSConfig 对象的字段如下： ▪ mode (String)：定义 TLS 行为，可取值 **Passthrough** - 直通模式。网关不会终止 TLS 会话，这意味着除了 TLS 协议的 ClientHello 消息外，网关无法破译 TLS 流。请注意，只有 TLSRoute 支持 SSL 直通模式；**Terminate** - 终止模式。即客户端的 TLS 会话终止在网关处——客户端与网关之间建立 TLS 会话。默认值为 Terminate。 ▪ certificateRefs ([]SecretObjectReference)：引用若干 kubernetes 资源对象（如 Secret），这些对象包含了 TLS 证书和私钥，用于网关与各个监听器中对应的主机之间建立 TLS 会话。当 mode 字段的值为 Terminate 时，本字段至少包含一个元素，否则为可选字段。SecretObjectReference 对象的字段如下： ▫ group (String)：资源对象的 GROUP，省略本字段或者值为空字符串""则表示核心组。比如"gateway.networking.k8s.io"。 ▫ kind (String)：资源对象的 Kind，"Secret"。

https://www.veryopen.org

	▫ name (String)：资源对象的名字。 ▫ namespace (String)：资源对象所在的名字空间。 ▪ options：键/值对列表，为扩展 TLS 指定额外的参数，例如配置最低 TLS 版本或支持的密码套件。 ○ allowedRoutes (AllowedRoutes)：可以附加到侦听器的路由类型，以及这些路由资源可能存在的受信任命名空间。AllowedRoutes 对象的字段如下： ▪ namespaces (RouteNamespaces)：从哪一个名字空间中选择路由。RouteNamespaces 对象的字段如下： ▫ from (String)：受信任的名字空间，此名字空间内的路由被附加到 Gateway。可取值"All"（所有名字空间）、"Same"（与 Gateway 同一个名字空间。这是默认值）、"Selector"（由 slector 字段选中的名字空间）。 ▫ selector (LabelSelector)：标签选择器（选择名字空间）。 ▪ kinds ([]RouteGroupKind)：指定 Route 资源对象的 GROUP 和 Kind。RouteGroupKind 对象的字段如下： ▫ group (String)：Route 资源对象的 GROUP。 ▫ **kind** (String)：Route 资源对象的 kind。 ● addresses ([]GatewayAddress)：若干外部负载平衡器或其他网络中的 IP 地址或主机名称。GatewayAddress 对象的字段如下： ○ type (String)：地址类型，可取值"Hostname"（主机名称）、"IPAddress"（ip 地址）。 ○ **value** (String)：ip 或主机名称。 ● infrastructure (GatewayInfrastructure)：定义有关此网关实例的基础架构级别属性。GatewayInfrastructure 对象的字段如下： ○ labels：若干标签（用“键：值对”定义）。 ○ annotations：若干注释（用“键：值对”定义）。
status (GatewayStatus)	Gateway 的当前状态。状态信息只能读。

加黑字段为必需字段。但如果父字段是可选字段，这时只有提供了父字段的情况下，加黑的字段才是必须的，比如 value 字段。

例子：

```
[root@master1 ~]# vim gateway1.yaml
apiVersion: gateway.networking.k8s.io/v1beta2
kind: Gateway
metadata:
  name: prod-web
spec:
  gatewayClassName: acme-lb
  listeners:
  - protocol: HTTP
    port: 80
    name: prod-web-gw
```

§ 19.2.3 HTTPRoute

HTTPRoute 资源对象提供了一种路由 HTTP 请求的方法。路由策略包括按主机名、路径、标头或查询参数匹配。筛选器可用于指定额外的处理步骤。一个完整的 URL 见图 17-1。

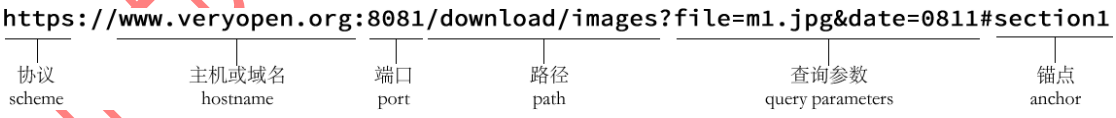


图 17-1 完整的 URL

HTTPRoute 定义请参考表 17-3。

表 17-3 HTTPRoute 资源对象的字段

字段（类型）	说明
apiVersion (string)	目前的值固定为 gateway.networking.k8s.io/v1alpha2
kind (string)	值固定为 HTTPRoute
metadata (ObjectMeta)	描述资源对象的元数据。关于 ObjectMeta 对象请参考表 11-3
spec (HTTPRouteSpec)	定义 HTTPRoute 的行为。HTTPRouteSpec 对象的字段如下： ● parentRefs ([]ParentReference)：若干父资源对象（目前只支持 Gateway，将来可能支持 HTTPRoute）。ParentReference 对象的字段如下： ○ group (String)：父资源对象的 GROUP，如"gateway.networking.k8s.io"。

	○ kind (String)：父资源对象的 kind，核心组支持的两个 kind 为 Gateway 和 Service。 ○ namespace (String)：父资源对象所在的名字 空间。 ○ **name** (String)：父资源对象的名字。 ○ sectioname (String)：父资源对象中的节名。对于 Gateway，节名为监听器名称，对于 Service，节名为端口名称。 ● hostnames ([]String)：指定若干主机名，用于匹配 HTTP 请求中 Host 头部来选择路由。不允许采用 ip 地址，但允许最左侧出现*，如*.veryopen.org。 ● rules ([]HTTPRouteRule)：若干路由策略，即对匹配（matches）成功的 HTTP 请求做修改（filters）后路由到相应的后端。HTTPRouteRule 对象的字段如下： ○ matches ([]HTTPRouteMatch)：定义一系列的匹配器，用来匹配到达的 HTTP 请求。这些匹配器是独立的，即只要任何一个匹配器匹配成功，就表名本条规则匹配成功。HTTPRouteMatch 对象的字段如下： ▪ path (HTTPPathMatch)：匹配 HTTP 请求的路径，默认为“/”。HTTPPathMatch 对象的字段如下： ▫ type (String)：定义匹配的方法，可取值"Exact"（完整匹配，字母大小写敏感）、"PathPrefix"（前缀匹配，以“/”做分隔符，字母大小写敏感）、"RegularExpression"（正则表达式匹配，字母大小写敏感）。 ▫ value (String)：路径。如果 type 的值不是"RegularExpression"，那么 path 的值必须以“/”开头，并且不能连续出现“/”字符。 ▪ headers ([]HTTPHeaderMatch)：若干 HTTP 请求的头部匹配器，这些匹配器是“与”关系。HTTPHeaderMatch 对象的字段如下： ▫ type (String)：定义匹配的方法，可取值"Exact"（完整匹配，字母大小写敏感）、"RegularExpression"（正则表达式匹配，字母大小写敏感）。 ▫ **name** (String)：HTTP 头部参数名，如"Authorization"、"Set-Cookie"。 ▫ **value** (String)：HTTP 头部参数的值。 ▪ queryParams ([]HTTPQueryParamMatch)：若干 HTTP 查询参数匹配器，这些匹配器是“与”关系。HTTPQueryParamMatch 对象的字段如下： ▫ type (String)：定义匹配的方法，可取值"Exact"（完整匹配，字母大小写敏感）、"RegularExpression"（正则表达式匹配，字母大小写敏感）。 ▫ **name** (String)：查询参数的名字。 ▫ **value** (String)：查询参数的值。 ▪ method (String)：HTTP 请求的方法匹配器。可取值"CONNECT"、"DELETE"、"GET"、"HEAD"、"OPTIONS"、"PATCH"、"POST"、"PUT"、"TRACE"。 ○ filters ([]HTTPRouteFilter)：定义若干过滤器，当本条规则匹配成功时，在 HTTP 请求上应用这些过滤器。HTTPRouteFilter 对象的字段如下： ▪ type (String)：过滤器的类型，可取值"ExtensionRef"（自定义过滤器）、"RequestHeaderModifier"（在转发前，增加或删除请求的头部中的参数）、"RequestMirror"（镜像请求到不通的后端）、"RequestRedirect"（重定向请求）、"ResponseHeaderModifier"（把响应返回给客户端前，增加或删除头部参数）、"URLRewrite"（在转发时修改请求）。 ▪ requestHeaderModifier (HTTPHeaderFilter)：用于修改请求 HTTP 头部的过滤器。HTTPHeaderFilter 对象的字段如下： ▫ set ([]HTTPHeader)：修改头部参数。HTTPHeader 对象的字段如下： ★ **name** (string)：头部参数的名称。 ★ **value** (String)：头部参数的值。 ▫ add ([]HTTPHeader)：增加头部参数。 ▫ remove ([]string)：删除若干头部中的参数。 ▪ responseHeaderModifier (HTTPHeaderFilter)：用于修改响应 HTTP 头部的过滤器。HTTPHeaderFilter 对象参见上面的字段介绍。 ▪ requestMirror (HTTPRequestMirrorFilter)：对请求进行镜像的过滤器。HTTPRequestMirrorFilter 对象的字段如下： ▫ backendRef (BackendObjectReference)：引用资源对象，请求被镜像到这些资源对象。BackendObjectReference 对象请参考 HTTPRoute.spec.CommonRouteSpec.parentRefs。 ▪ requestRedirect (HTTPRequestRedirectFilter)：对请求进行重定向的过滤器。HTTPRequestRedirectFilter 对象的字段如下： ▫ scheme (String)：响应头部中的 Location 参数的 scheme 部分，如果省略，就采用请求头部的 scheme。Scheme 就是 Location 中“://”左边部分，比如 https。 ▫ hostname (String)：响应头部中的 Location 参数的主机部分，如果省略，就采用请求头部的主机。主机就是 Location 中“://”与“:”或“/”之间的部分，参见图 17-1。 ▫ path (HTTPPathModifier)：修改请求中的路径。HTTPPathModifier 对象的字段如下： ★ type (String)：指定修改路径的方式。可取值"ReplaceFullPath"（更换整条路径）、"ReplacePrefixMatch"（修改前缀部分）。 ★ replaceFullPath (String)：当 type 为"ReplaceFullPath"时，定义的新路径。 ★ replacePrefixMatch (String)：当 type 为"ReplacePrefixMatch"时，定义的前缀。例如对于请求路径/foo/bar，前缀匹配/foo，replacePrefixMatch 为/xyz，那么路径被修改为/xyz/bar。 ▫ port (unint32)：修改 HTTP 响应的端口。 ▫ statusCode (int)：响应的状态码。重定向状态码为 3xx，其中 301 和 308 是永久重定向，302、303 和 307 是临时重定向。 ▪ urlRewrite (HTTPURLRewriteFilter)：转发时修改请求。HTTPURLRewriteFilter 对象的字段如下： ▫ hostname (String)：新的主机名。 ▫ path (HTTPPathModifier)：新的路径。HTTPPathModifier 对象请参考 HTTPRoute.spec.rules.filters.requestRedirect.path。 ▪ extensionRef (LocalObjectReference)：特定的扩展过滤器（引用另一个资源对象）。LocalObjectReference 对象的字段如下： ▫ **group** (String)：资源对象的 GROUP，省略本字段或者值为空字符串""则表示核心组。比如"gateway.networking.k8s.io"。 ▫ **kind** (String)：资源对象的 Kind，比如"HTTPRoute"、"Service"。 ▫ **name** (String)：资源对象的名字。 ○ backendRefs ([]HTTPBackendRef)：指定若干后端，匹配成功的请求被发送到这些后端。HTTPBackendRef 对象的字段如下： ▪ 字段 group、kind、name、namespace 和 port 可参考 HTTProute.spec.CommonRouteSpec.parentRefs 字段。 ▪ weight (int32)：本后端的权重。最终请求被转发到本后端的概率是“weight/全部后端的权重之和”。 ▪ filters ([]HTTPRouteFilter)：定义若干过滤器，被转发到后端的请求需要引用这些过滤器。HTTPRouteFilter 对象的介绍请的参考 HTTPRoute.spec.rules.filters 字段。 ○ timeouts (HTTPRouteTimeouts)：为请求设置超时。HTTPRouteTimeouts 对象的字段如下： ▪ request (String)：网关响应客户端的最长时间，比如 10s 表示响应客户端不能超过 10 秒。 ▪ backendRequest (String)：后端响应网关的最长时间。backendRequest 只是 request 的一部分，所以 request 的值必须大于 backendRequest 的值。

https://www.veryopen.org

status（HTTPRouteStatus）	HTTPRoute 的当前状态。状态信息只能读。

例子：

```
[root@master1 ~]# vim httproute1.yaml
apiVersion: gateway.networking.k8s.io/v1beta2
kind: HTTPRoute
metadata:
  name: foo
spec:
  parentRefs:
  - name: prod-web                                                    #上面的例子定义的
Gateway
  rules:
  - backendRefs:
    - name: foo-svc
      port: 8080
```

§19.3 实战

https://gateway-api.sigs.k8s.io/guides/http-routing/

我们采用 Istio 的控制面+MetalLB+Gateway API 对外暴露 Wordpress 应用。

1. 安装 Gateway API：

```
[root@master1 ~]# kubectl apply -f https://github.com/kubernetes-sigs/gateway-api/releases/download/v1.0.0/standard-install.yaml
customresourcedefinition.apiextensions.k8s.io/gatewayclasses.gateway.networking.k8s.io created
customresourcedefinition.apiextensions.k8s.io/gateways.gateway.networking.k8s.io created
customresourcedefinition.apiextensions.k8s.io/httproutes.gateway.networking.k8s.io created
customresourcedefinition.apiextensions.k8s.io/referencegrants.gateway.networking.k8s.io created
```

2. 参考“§17.2.3 向外暴露应用”的“二、通过 LoadBalancer 类型的服务暴露”部署 MetalLB。

3. 参考“§19.1 Istio”安装 Istio。

4. 参考“§17.4 wordpress”部署应用，不过文件 wordpress-service.yaml 中的字段 type 的值改为 ClusterIP 或者干脆删除这个字段，因为默认就是 ClusterIP。

5. 部署 Gateway API：

```
[root@master1 ~]# kubectl create namespace istio-gateway
namespace/istio-gateway created

给 istio-gateway 打上一个标签 istio-injection=enabled，这样 Istio 能注入相应的资源：
[root@master1 ~]# kubectl label namespace istio-gateway istio-injection=enabled
namespace/istio-gateway labeled

[root@master1 ~]# cd ~/wordpress
[root@master1 wordpress]# vim gateway.yaml
apiVersion: gateway.networking.k8s.io/v1beta1
kind: Gateway
metadata:
  name: gateway1
  namespace: istio-gateway
spec:
  gatewayClassName: istio # 这里指定使用 istio gatewayclass
  listeners:
  - name: default
    hostname: "*.abc.com"
    port: 80
    protocol: HTTP
    allowedRoutes:
      namespaces:
```

https://www.veryopen.org

```
        from: All

[root@master1 wordpress]# kubectl create -f gateway.yaml
gateway.gateway.networking.k8s.io/gateway1 created

[root@master1 wordpress]# kubectl get pod -n istio-gateway -o wide
NAME                          READY   STATUS    RESTARTS   AGE   IP             NODE      NOMINATED NODE   READINESS GATES
gateway1-istio-859566498c-5lwrl   1/1     Running   0          38m   10.244.2.18   worker2   <none>           <none>
```

可用命令“kubectl get gatewayclass”查看 Istio 的 gatewayclass，名字就是 istio。

```
[root@master1 wordpress]# kubectl create -f httproute.yaml
apiVersion: gateway.networking.k8s.io/v1beta1
kind: HTTPRoute
metadata:
  name: http
  namespace: wordpress
spec:
  hostnames: ["www.abc.com"]
  parentRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: gateway1
    namespace: istio-gateway
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - group: ""
      kind: Service
      name: wordpress-service
      port: 80
      namespace: wordpress
      weight: 1

[root@master1 wordpress]# kubectl create -f httproute.yaml
httproute.gateway.networking.k8s.io/http created

[root@master1 wordpress]# kubectl get svc -n istio-gateway -o wide
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)                      AGE   SELECTOR
gateway1-istio   LoadBalancer   10.96.38.202   10.10.10.101   15021:30361/TCP,80:31906/TCP   22m   istio.io/gateway-name=gateway1
```

修改客户机上 hosts 文件内容，假如下面一行：

```
10.10.10.101           www.abc.com
```

然后浏览 http://www.abc.com。

https://www.ve

第20章 Serverless and Event-Driven Applications

knative

https://www.veryopen.org

第21章 二进制方式安装 K8S

---=== 全部虚拟机恢复快照 before kubeadm ======-

以二进制文件方式安装 Kubernetes 安全高可用集群

https://github.com/kubernetes/kubernetes/releases

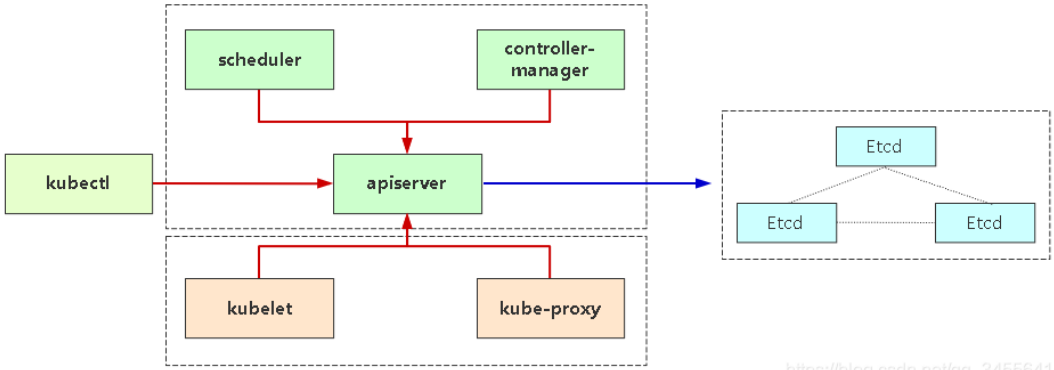
http://zhangshiyu.com/post/33049.html

目前生产环境部署 Kubernetes 主要有两种方式：

1. kubeadm 方式：提供 kubeadm init 和 kubeadm join，用于快速部署 Kubernetes 集群，kubeadm 安装的 k8s 集群，所有的 k8s 组件都是以 pod 形式运行。

2. 二进制方式：从 github.com 下载发行版的二进制包，手动部署每个组件，组成 kubernetes 集群。

Kubeadm 降低了部署成本，但屏蔽了很多部署细节，遇到问题很难排查。如果想更容易掌控，推荐使用二进制方式，虽然手动部署麻烦点，但可以学到很多工作原理，更有利于后期运维。本章我们就采用二进制方式部署 k8s 集群。



https://blog.csdn.net/qq_34556414

Kubernetes 和 etcd 使用同一套 CA 根证书

§21.1 部署 etcd

在 master1 上安装 etcd，请参考“§10.1 安装”小节。

下面给 etcd 增加客户端证书，以后 kube-apiserver 访问 etcd 时要用到，而 etcd 集群中各个节点的证书等以后

增加节点时再来创建。

```
[root@master1 ~]# mkdir -p /etc/kubernetes/pki/etcd
[root@master1 ~]# cd /etc/kubernetes/pki/etcd

生成 RSA 算法 2048 位的 CA 私钥和自签证书:
[root@master1 pki]# openssl genpkey -algorithm RSA -out ca.key -quiet
[root@master1 pki]# openssl req -x509 -key ca.key -subj "/CN=etcd-ca" -days 365 -out ca.crt
[root@master1 pki]# ll ca.*
-rw-r--r-- 1 root root 1099 Aug 18 10:24 ca.crt
-rw------- 1 root root 1704 Aug 18 10:24 ca.key

新建 openssl 配置文件，指明 etcd 集群的节点主机 ip 地址 10.10.10.11、10.10.10.12、10.10.10.13，分别对应主机 master1、master2、
master3，这次先在 master1 运行节点 etcd01:
[root@master1 pki]# vim openssl_etcd.conf
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name

[req_distinguished_name]

[v3_req]
basicConstraints = CA:FALSE
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth, clientAuth
subjectAltName = @alt_names

[alt_names]
DNS.1 = localhost
DNS.2 = master1
IP.1 = 127.0.0.1
IP.2 = 10.10.10.11
IP.3 = 10.10.10.12
IP.4 = 10.10.10.13

生成 RSA 的 etcd 服务器用到的私钥 server.key:
[root@master1 pki]# openssl genpkey -algorithm RSA -quiet -out server.key

生成证书申请文件 server.csr:
[root@master1 pki]# openssl req -new -key server.key -subj "/CN=master1" -out server.csr -config openssl_etcd.conf

签发证书 etcd.crt，证书有效期 180 天:
[root@master1 pki]# openssl req -x509 -in server.csr -CA ca.crt -CAkey ca.key -out server.crt -days 180 \
 -copy_extensions copy

[root@master1 pki]# ll
total 24
-rw-r--r-- 1 root root 1099 Aug 18 10:24 ca.crt                                                    #CA
自签证书
-rw------- 1 root root 1704 Aug 18 10:24 ca.key
#CA 私钥
-rw-r--r-- 1 root root 1224 Aug 18 14:09 server.crt                                                #etcd 服务的证
书
-rw-r--r-- 1 root root 1041 Aug 18 14:09 server.csr
-rw------- 1 root root 1704 Aug 18 14:09 server.key                                                #etcd 服务的私
钥
-rw-r--r-- 1 root root  365 Aug 18 09:33 openssl_etcd.conf

查看证书:
[root@master1 pki]# openssl x509 -in server.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            73:f4:8b:0f:d8:f5:88:08:93:59:4e:df:d2:34:dc:f2:42:f8:9e:c6
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = etcd-ca
        Validity
            Not Before: Aug 18 02:25:47 2023 GMT
            Not After : Feb 14 02:25:47 2024 GMT
        Subject: CN = master1
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:9a:47:09:f8:d4:a1:56:9c:8a:67:c4:70:83:81:
                    ......
                    ef:83:2c:89:4f:d0:f7:65:b3:c2:fc:30:b4:76:1c:
                    0a:bd
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Subject Alternative Name:
                IP Address:127.0.0.1, IP Address:10.10.10.11, IP Address:10.10.10.12, IP Address:10.10.10.13
            X509v3 Subject Key Identifier:
                38:98:BB:2C:62:EF:FE:67:DC:15:2D:91:B1:D4:51:27:58:63:23:2C
            X509v3 Authority Key Identifier:
                D9:F2:91:FB:D7:7F:33:48:3D:AB:44:3B:FA:4A:42:DB:CD:42:79:2D
```

```
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        1d:75:44:e2:5e:eb:cb:84:12:b7:25:d8:75:22:58:b0:1d:44:
        ......
        58:b0:53:c4

[root@master1 ~]# chown etcd /etc/kubernetes/pki/etcd/server.*
[root@master1 ~]# mkdir /etc/etcd/
[root@master1 ~]# vim /etc/etcd/etcd.conf
ETCD_NAME="etcd01"
ETCD_DATA_DIR="/var/lib/etcd"

#配置与客户端的通信:
ETCD_LISTEN_CLIENT_URLS="https://0.0.0.0:2379"
ETCD_ADVERTISE_CLIENT_URLS="http://0.0.0.0:2379"
ETCD_CLIENT_CERT_AUTH=true
ETCD_TRUSTED_CA_FILE="/etc/kubernetes/pki/etcd/ca.crt"
ETCD_CERT_FILE="/etc/kubernetes/pki/etcd/server.crt"
ETCD_KEY_FILE="/etc/kubernetes/pki/etcd/server.key"

配置 etcd 集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
#ETCD_INITIAL_CLUSTER_STATE="new"| "existing"

[root@master1 ~]# vim /etc/systemd/system/etcd.service
[Unit]
Description=etcd key-value store
After=network.target

[Service]
User=etcd
Type=notify
EnvironmentFile=/etc/etcd/etcd.conf
ExecStart=/usr/local/bin/etcd
Restart=always
RestartSec=10s
LimitNOFILE=40000

[Install]
WantedBy=multi-user.target

[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl restart etcd

[root@master1 ~]# ss -tnulp
Netid  State   Recv-Q  Send-Q     Local Address:Port      Peer Address:Port  Process
udp    UNCONN  0       0               0.0.0.0:123            0.0.0.0:*      users:(("chronyd",pid=746,fd=7))
udp    UNCONN  0       0             127.0.0.1:323            0.0.0.0:*      users:(("chronyd",pid=746,fd=5))
udp    UNCONN  0       0                 [::1]:323               [::]:*      users:(("chronyd",pid=746,fd=6))
tcp    LISTEN  0       128             0.0.0.0:22             0.0.0.0:*      users:(("sshd",pid=840,fd=3))
tcp    LISTEN  0       4096          127.0.0.1:35785          0.0.0.0:*      users:(("containerd",pid=847,fd=12))
tcp    LISTEN  0       4096          127.0.0.1:2380           0.0.0.0:*      users:(("etcd",pid=2058,fd=7))
tcp    LISTEN  0       4096                *:2379                 *:*      users:(("etcd",pid=2058,fd=8))
tcp    LISTEN  0       128               [::]:22                [::]:*      users:(("sshd",pid=840,fd=4))

[root@master1 ~]# cat <<EOF >>.bashrc
export ETCDCTL_ENDPOINTS=https://10.10.10.11:2379
export ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt
export ETCDCTL_CERT=/etc/kubernetes/pki/etcd/server.crt
export ETCDCTL_KEY=/etc/kubernetes/pki/etcd/server.key
EOF

[root@master1 ~]# . .bashrc

[root@master1 ~]# env|grep ETCDCTL
ETCDCTL_ENDPOINTS=https://10.10.10.11:2379
ETCDCTL_KEY=/etc/kubernetes/pki/etcd/server.key
ETCDCTL_CACERT=/etc/kubernetes/pki/etcd/ca.crt
ETCDCTL_CERT=/etc/kubernetes/pki/etcd/server.crt

[root@master1 ~]# etcdctl put hello "Hello Etcd!"
OK
[root@master1 ~]# etcdctl get hello
hello
Hello Etcd!

检查 etcd 集群的状态:
[root@master1 ~]# etcdctl member list --write-out=table
+------------------+---------+--------+-----------------------+---------------------+------------+
|        ID        | STATUS  |  NAME  |      PEER ADDRS       |     CLIENT ADDRS    | IS LEARNER |
+------------------+---------+--------+-----------------------+---------------------+------------+
| 8e9e05c52164694d | started | etcd01 | http://localhost:2380 | http://0.0.0.0:2379 |      false |
+------------------+---------+--------+-----------------------+---------------------+------------+

[root@master1 ~]# etcdctl endpoint status --write-out=table
+--------------------------+------------------+---------+--------+-----------+------------+-----------+------------+--------------------+--------+
|         ENDPOINT         |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+--------------------------+------------------+---------+--------+-----------+------------+-----------+------------+--------------------+--------+
| https://10.10.10.11:2379 | 8e9e05c52164694d |   3.5.9 |   29 kB |      true |      false |        42 |         86 |                 86 |        |
```

§21.2 部署 kubernetes

§21.2.1 准备二进制程序

下载 kubernetes 的二进制压缩包，打开网站 https://github.com/kubernetes/kubernetes，点击右侧的 Release 下面的 latest，然后点击 CHANGELOG，再点击 Server Binaries，最后下载 kubernetes-server-linux-amd64.tar.gz

1．准备控制面程序：

```
[root@master1 ~]# wget https://dl.k8s.io/v1.28.2/kubernetes-server-linux-amd64.tar.gz
[root@master1 ~]# tar -xf kubernetes-server-linux-amd64.tar.gz
[root@master1 ~]# cp -v kubernetes/server/bin/kube{let,ctl,-apiserver,-controller-manager,-scheduler,-proxy}
/usr/local/bin/
'kubernetes/server/bin/kubelet' -> '/usr/local/bin/kubelet'
'kubernetes/server/bin/kubectl' -> '/usr/local/bin/kubectl'
'kubernetes/server/bin/kube-apiserver' -> '/usr/local/bin/kube-apiserver'
'kubernetes/server/bin/kube-controller-manager' -> '/usr/local/bin/kube-controller-manager'
'kubernetes/server/bin/kube-scheduler' -> '/usr/local/bin/kube-scheduler'
'kubernetes/server/bin/kube-proxy' -> '/usr/local/bin/kube-proxy'

[root@master1 bin]# kubelet --version
Kubernetes v1.28.2
```

2．下载工作节点程序：

```
[root@master1 ~]# wget https://dl.k8s.io/v1.28.2/kubernetes-node-linux-amd64.tar.gz
```

3．下载客户端工具：

```
[root@master1 ~]# wget https://dl.k8s.io/v1.28.2/kubernetes-client-linux-amd64.tar.gz
```

§21.2.2 部署 kube-apiserver

1．产生 kube-apiserver 服务器证书

```
[root@master1 ~]# cd ~/pki
[root@master1 pki]# vim openssl_k8s.conf
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name

[req_distinguished_name]

[v3_req]
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth, clientAuth
subjectAltName = @alt_names

[alt_names]
DNS.1 = kubernetes
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
```

https://www.chen.org

```
DNS.4 = kubernetes.default.svc.cluster.local
DNS.5 = master1
DNS.6 = master2
DNS.7 = master3
IP.1 = 127.0.0.1
IP.2 = 10.10.10.10
IP.3 = 10.10.10.11
IP.4 = 10.10.10.12
IP.5 = 10.10.10.13

[root@master1 pki]# openssl genpkey -algorithm RSA -quiet -out kube-apiserver.key
[root@master1 pki]# openssl req -new -key kube-apiserver.key -subj "/CN=kubernetes" -out kube-apiserver.csr \
 -config openssl_k8s.conf
[root@master1 pki]# openssl req -x509 -in kube-apiserver.csr -CA ca.crt -CAkey ca.key -out kube-apiserver.crt \
 -days 365 -copy_extensions copy
[root@master1 pki]# ll kube*
-rw-r--r-- 1 root root 1460 Aug 18 14:37 kube-apiserver.crt
-rw-r--r-- 1 root root 1273 Aug 18 14:36 kube-apiserver.csr
-rw------- 1 root root 1704 Aug 18 11:11 kube-apiserver.key

[root@master1 pki]# cp kube-apiserver.crt kube-apiserver.key /etc/kubernetes/pki/
```

```
[root@master1 ~]# vim /etc/systemd/system/kube-apiserver.service
[Unit]
Description=Kubernetes API Server
After=network.target

[Service]
EnvironmentFile=/etc/kubernetes/apiserver.conf
ExecStart=/usr/local/bin/kube-apiserver $KUBE_API_ARGS
Restart=always
RestartSec=10s
LimitNOFILE=40000

[Install]
WantedBy=multi-user.target

[root@master1 ~]# vim /etc/kubernetes/apiserver.conf
KUBE_API_ARGS="--tls-cert-file=/etc/kubernetes/pki/kube-apiserver.crt \
--tls-private-key-file=/etc/kubernetes/pki/kube-apiserver.key \
--client-ca-file=/etc/kubernetes/pki/ca.crt \
--service-cluster-ip-range=10.96.0.0/16 \                                    #服务网络，不能于节点的 ip 地址重
合
--bind-address=10.10.10.11 \                                                 #监听本机
的哪块网卡
--service-account-signing-key-file=/etc/kubernetes/pki/kube-apiserver.key \
--service-account-key-file=/etc/kubernetes/pki/ca.key \                      #对发行的 ID 令牌进行签名的
CA 私钥
--service-account-issuer=api \
--external-hostname=master1 \
--allow-privileged=true \                                                    #是否允许容器以特权模式运行，默认值为
false
--etcd-servers=https://10.10.10.11:2379 \                                    #etcd 集群节点 ip，多个 Ip 之间用逗号分开
--etcd-cafile=/etc/kubernetes/pki/ca.crt \
--etcd-certfile=/etc/kubernetes/pki/etcd_client.crt \
--etcd-keyfile=/etc/kubernetes/pki/etcd_client.key \
--v=2"

[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl enable kube-apiserver
[root@master1 ~]# systemctl start kube-apiserver
[root@master1 ~]# systemctl status kube-apiserver
```

如果启动失败，请用命令“journalctl -b --no-pager -u kube-apiserver”查看详细错误信息。

部署 kube-controller-manager 服务

再创建客户端证书，kube-controller-manager、kube-scheduler、kubelet 和 kube-proxy 服务作为客户端连接 kube-apiserver 服务时，需要使用客户端证书进行访问。

```
[root@master1 pki]# openssl genpkey -algorithm RSA -quiet -out k8s-client.key
[root@master1 pki]# openssl req -new -key k8s-client.key -subj "/CN=admin" -out k8s-client.csr
[root@master1 pki]# openssl x509 -req -in k8s-client.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out k8s-
client.crt -days 365
Certificate request self-signature ok
subject=CN = admin
[root@master1 pki]# cp k8s-client.key k8s-client.crt /etc/kubernetes/pki/

[root@master1 ~]# vim /etc/kubernetes/kubeclient.yaml
apiVersion: v1
kind: Config
clusters:
```

```
- name: default                                                                    #kubernetes 集
群名字
  cluster:
    server: https://10.10.10.11:6443
    certificate-authority: /etc/kubernetes/pki/ca.crt
users:
- name: admin
  user:
    client-certificate: /etc/kubernetes/pki/k8s-client.crt
    client-key: /etc/kubernetes/pki/k8s-client.key
contexts:
- context:
    cluster: default
    user: admin
  name: default
current-context: default

[root@master1 ~]# vim /etc/systemd/system/kube-controller-manager.service
[Unit]
Description=Kubernetes Controller Manager
After=network.target

[Service]
EnvironmentFile=/etc/kubernetes/controller-manager.conf
ExecStart=/usr/local/bin/kube-controller-manager $KUBE_CONTROLLER_MANAGER_ARGS
Restart=always
RestartSec=10s
LimitNOFILE=40000

[Install]
WantedBy=multi-user.target

[root@master1 ~]# vim /etc/kubernetes/controller-manager.conf
KUBE_CONTROLLER_MANAGER_ARGS="--kubeconfig=/etc/kubernetes/kubeclient.yaml \                    #与 API Server 连接的
相关配置
--leader-elect=true \
--bind-address=10.10.10.11 \                                                                    #启
用选举机制
--service-cluster-ip-range=10.96.0.0/16 \                                                       #Service 网
络
--allocate-node-cidrs=true \
--cluster-cidr=10.244.0.0/16 \
#Pod 网络
--cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt \                        #PEM 编码的 X509 CA 证书。用于颁发集群范
围的证书
--cluster-signing-key-file=/etc/kubernetes/pki/ca.key \                           #PEM 编码的 RSA 私钥。用于颁发集群范围
的证书
--root-ca-file=/etc/kubernetes/pki/ca.crt \
--service-account-private-key-file=/etc/kubernetes/pki/kube-apiserver.key \
--v=2"
```

--service-account-private-key-file：为 ServiceAccount 自动颁发 token 使用的私钥文件，例如 /etc/kubernetes/pki/kube-apiserver.key。

--service-cluster-ip-range：Service 虚拟 IP 地址范围，以 CIDR 格式表示，例如 10.96.0.0/16，必须与 kube-apiserver 服务的同名参数一样。

```
[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl enable kube-controller-manager.service
[root@master1 ~]# systemctl start kube-controller-manager.service
```

如果启动失败，请用命令“journalctl -b --no-pager -u kube-controller-manager.service”查看详细错误信息。

部署 kube-scheduler 服务。

```
[root@master1 ~]# vim /etc/systemd/system/kube-scheduler.service
[Unit]
Description=Kubernetes Scheduler
After=network.target

[Service]
EnvironmentFile=/etc/kubernetes/scheduler.conf
ExecStart=/usr/local/bin/kube-scheduler $KUBE_SCHEDULER_ARGS
Restart=always
RestartSec=10s
LimitNOFILE=40000

[Install]
WantedBy=multi-user.target
```

```
[root@master1 ~]# vim /etc/kubernetes/scheduler.conf
KUBE_SCHEDULER_ARGS="--kubeconfig=/etc/kubernetes/kubeclient.yaml \
--leader-elect=true \
--bind-address=10.10.10.11 \
--v=2"

[root@master1 ~]# systemctl daemon-reload
[root@master1 ~]# systemctl enable kube-scheduler
Created symlink /etc/systemd/system/multi-user.target.wants/kube-scheduler.service → /etc/systemd/system/kube-
scheduler.service.
[root@master1 ~]# systemctl start kube-scheduler
[root@master1 ~]# systemctl status kube-scheduler
● kube-scheduler.service - Kubernetes Scheduler
     Loaded: loaded (/etc/systemd/system/kube-scheduler.service; enabled; preset: disabled)
     Active: active (running) since Sat 2023-08-19 08:40:58 CST; 5s ago
   Main PID: 1866 (kube-scheduler)
      Tasks: 9 (limit: 22914)
     Memory: 60.7M
        CPU: 700ms
     CGroup: /system.slice/kube-scheduler.service
             └─1866 /usr/local/bin/kube-scheduler --kubeconfig=/etc/kubernetes/kubeclient.yaml --leader-elect=true
--bind-address=10.10.10.11 --v=2
```

如果启动失败，请用命令“journalctl -b --no-pager -u kube-scheduler.service”查看详细错误信息。

```
[root@master1 ~]# ss -tunlp
Netid  State   Recv-Q  Send-Q     Local Address:Port      Peer Address:Port  Process
udp    UNCONN  0       0              0.0.0.0:123            0.0.0.0:*      users:(("chronyd",pid=750,fd=7))
udp    UNCONN  0       0            127.0.0.1:323            0.0.0.0:*      users:(("chronyd",pid=750,fd=5))
udp    UNCONN  0       0                [::1]:323               [::]:*      users:(("chronyd",pid=750,fd=6))
tcp    LISTEN  0       4096       10.10.10.11:10259          0.0.0.0:*      users:(("kube-scheduler",pid=1866,fd=3))
tcp    LISTEN  0       4096       10.10.10.11:10257          0.0.0.0:*      users:(("kube-controller",pid=1885,fd=3))
tcp    LISTEN  0       4096         127.0.0.1:35095          0.0.0.0:*      users:(("containerd",pid=798,fd=14))
tcp    LISTEN  0       4096         127.0.0.1:2380           0.0.0.0:*      users:(("etcd",pid=786,fd=7))
tcp    LISTEN  0       4096       10.10.10.11:6443           0.0.0.0:*      users:(("kube-apiserver",pid=1375,fd=3))
tcp    LISTEN  0       128            0.0.0.0:22             0.0.0.0:*      users:(("sshd",pid=794,fd=3))
tcp    LISTEN  0       128               [::]:22                [::]:*      users:(("sshd",pid=794,fd=4))
tcp    LISTEN  0       4096                 *:2379                 *:*      users:(("etcd",pid=786,fd=8))
```

https://www.very

第22章 搭建本地源

在 10.8.20.8 上安装 Rocky Linux 9，主机名为 mirrors，禁用 SELinux。

§22.1 操作系统安装源

在生产环境，如果节点比较多，建议搭建 Rocky Linux 9 操作系统的本地安装源，这样既能节省出口流量又能快速安装软件包。

1. 准备一个大磁盘（不少于 128GB）

```
[root@mirrors ~]# parted --script /dev/sdb 'mkpart primary 0% 100%'
[root@mirrors ~]# mkfs.xfs /dev/sdb1
[root@mirrors ~]# blkid /dev/sdb1
/dev/sdb1: UUID="d4582e11-7554-4956-83d9-489a7a4513d7" TYPE="xfs" PARTUUID="3fca9eaf-01"
[root@mirrors ~]# echo "UUID=d4582e11-7554-4956-83d9-489a7a4513d7 /var/www/html xfs defaults 1 1" >> /etc/fstab
[root@mirrors ~]# systemctl daemon-reload
[root@mirrors ~]# mount -a
[root@mirrors ~]# mkdir -p /var/www/html/rockylinux/9/x86_64/os
```

2. 安装必要的软件

```
[root@mirrors ~]# dnf -y install httpd dnf-plugins-core createrepo_c
[root@mirrors ~]# systemctl --now enable httpd
```

3. 同步 rpm 软件包

查看哪些库被激活：

```
[root@mirrors ~]# mkdir -p /etc/yum.repos.d/rocky9-sync
[root@mirrors ~]# vim /etc/yum.repos.d/rocky9-sync/rocky9.repo
[baseos]
name=Rocky Linux 9 - BaseOS
baseurl=https://mirrors.nju.edu.cn/rocky/9/BaseOS/x86_64/os/
enabled=1
gpgcheck=0

[appstream]
name=Rocky Linux 9 - AppStream
baseurl=https://mirrors.nju.edu.cn/rocky/9/AppStream/x86_64/os/
enabled=1
gpgcheck=0

[extras]
name=Rocky Linux 9 - Extras
baseurl=https://mirrors.nju.edu.cn/rocky/9/extras/x86_64/os/
enabled=1
gpgcheck=0

[crb]
name=Rocky Linux 9 - CRB
baseurl=https://mirrors.nju.edu.cn/rocky/9/CRB/x86_64/os/
enabled=1
gpgcheck=0

[resilientstorage]
name=Rocky Linux 9 - ResilientStorage
baseurl=https://mirrors.nju.edu.cn/rocky/9/ResilientStorage/x86_64/os/
enabled=1
gpgcheck=0

[rt]
name=Rocky Linux 9 - RT
baseurl=https://mirrors.nju.edu.cn/rocky/9/RT/x86_64/os/
enabled=1
gpgcheck=0
```

对于这些已激活的库可以同步到本地：

```
[root@mirrors ~]# dnf reposync --download-path /var/www/html/rockylinux/9/x86_64/os/ --repo baseos \
  --config /etc/yum.repos.d/rocky9-sync/rocky9.repo --newest-only --delete

[root@mirrors ~]# dnf reposync --download-path /var/www/html/rockylinux/9/x86_64/os/ --repo appstream \
  --config /etc/yum.repos.d/rocky9-sync/rocky9.repo --newest-only --delete

[root@mirrors ~]# dnf reposync --download-path /var/www/html/rockylinux/9/x86_64/os/ --repo extras \
  --config /etc/yum.repos.d/rocky9-sync/rocky9.repo --newest-only --delete

[root@mirrors ~]# dnf reposync --download-path /var/www/html/rockylinux/9/x86_64/os/ --repo crb \
  --config /etc/yum.repos.d/rocky9-sync/rocky9.repo --newest-only --delete

[root@mirrors ~]# dnf reposync --download-path /var/www/html/rockylinux/9/x86_64/os/ --repo rt \
  --config /etc/yum.repos.d/rocky9-sync/rocky9.repo --newest-only --delete

[root@mirrors ~]# dnf reposync --download-path /var/www/html/rockylinux/9/x86_64/os/ --repo resilientstorage \
  --config /etc/yum.repos.d/rocky9-sync/rocky9.repo --newest-only --delete
```

4．每天同步最新的软件包

```
[root@mirrors ~]# vim /etc/cron.daily/rocky-sync
#!/bin/bash

VER='9'
ARCH='x86_64'
REPOS=(baseos appstream extras crb rt resilientstorage)

# 上游官方源地址（用于单独拉取 modules.yaml）
UPSTREAM_BASE="https://mirrors.nju.edu.cn/rocky/${VER}"

declare -A REPO_PATH_MAP=(
[baseos]="BaseOS"
[appstream]="AppStream"
[extras]="extras"
[crb]="CRB"
[rt]="RT"
[resilientstorage]="ResilientStorage"
)

for REPO in ${REPOS[@]}
do
        LOCAL_DIR="/var/www/html/rockylinux/${VER}/${ARCH}/os/${REPO}"

        dnf reposync --download-path /var/www/html/rockylinux/${VER}/${ARCH}/os/ \
                --repo ${REPO} --config /etc/yum.repos.d/rocky9-sync/rocky9.repo \
                --newest-only --delete

        createrepo_c --update ${LOCAL_DIR}

  # 只有模块化仓库才需要处理 modules.yaml（baseos/extras/crb/rt/resilientstorage 通常没有模块化内容，
  # 但为了保险，统一尝试拉取，如果上游没有这个文件会自然失败，不影响其他仓库同步）
  UPSTREAM_PATH="${REPO_PATH_MAP[$REPO]}"
  MODULES_URL="${UPSTREAM_BASE}/${UPSTREAM_PATH}/${ARCH}/os/repodata/"

  # 从上游 repomd.xml 里解析出 modules.yaml.xz 的实际文件名（带 hash 前缀，每次同步可能变化）
  MODULES_URL_PATH=$(curl -s "${MODULES_URL}repomd.xml" | grep -oP '(?<=href=")[^"]*modules\.yaml\.(gz|xz)' | head
-1)

  if [ -n "$MODULES_URL_PATH" ]; then
          EXT="${MODULES_URL_PATH##*.}"
          curl -s -o /tmp/${REPO}-modules.yaml.${EXT}
"${UPSTREAM_BASE}/${UPSTREAM_PATH}/${ARCH}/os/${MODULES_URL_PATH}"

          if [ "$EXT" = "gz" ]; then
                  gunzip -f /tmp/${REPO}-modules.yaml.${EXT}
          else
                  xz -d -f /tmp/${REPO}-modules.yaml.${EXT}
          fi

          modifyrepo_c --mdtype=modules /tmp/${REPO}-modules.yaml ${LOCAL_DIR}/repodata/
          rm -f /tmp/${REPO}-modules.yaml
          echo "[$REPO] modules.yaml injected successfully"
  else
          echo "[$REPO] no modules.yaml found upstream, skipping"
  fi
done

[root@mirrors ~]# chmod 755 /etc/cron.daily/update-repo
```

5．配置 SELinux

```
如果 SELinux 没有禁用，就执行下面的命令：
[root@mirrors ~]# restorecon -vR /var/www/html

[root@mirrors ~]# systemctl restart httpd
```

7．配置防火墙

```
[root@mirrors ~]# firewall-cmd --add-service=http
[root@mirrors ~]# firewall-cmd --runtime-to-permanent
```

8．其他电脑上配置安装源

```
[root@othernodes ~]# vim /etc/yum.repos.d/local-rocky.repo
[baseos]
name=Rocky Linux $releasever - BaseOS
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/baseos/
gpgcheck=0
enabled=1

[appstream]
name=Rocky Linux $releasever - AppStream
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/appstream/
gpgcheck=0
enabled=1

[extras]
name=Rocky Linux $releasever - Extras
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/extras/
gpgcheck=0
enabled=1

[crb]
name=Rocky Linux $releasever - CRB
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/crb/
gpgcheck=0
enabled=1

[resilientstorage]
name=Rocky Linux $releasever - ResilientStorage
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/resilientstorage/
gpgcheck=0
enabled=1

[rt]
name=Rocky Linux $releasever - RT
baseurl=http://10.8.20.8/rockylinux/$releasever/$basearch/os/rt/
gpgcheck=0
enabled=1

[root@othernodes ~]# dnf clean all
[root@othernodes ~]# dnf makecache
[root@othernodes ~]# dnf repolist
repo id                       repo name
appstream                     Rocky Linux 9 - AppStream
baseos                        Rocky Linux 9 - BaseOS
extras                        Rocky Linux 9 - Extras packages
```

§22.2 k8s/containerd 源

```
# k8s 源:
[root@mirrors ~]# mkdir -p /var/www/html/kubernetes
[root@mirrors ~]# vim /etc/yum.repos.d/kubernetes-sync.repo
[kubernetes]
name=Kubernetes v1.36
baseurl=https://pkgs.k8s.io/core:/stable:/v1.36/rpm/
enabled=1
gpgcheck=0

[root@mirrors ~]# dnf reposync --download-path /var/www/html --repoid=kubernetes --download-metadata \
  --config /etc/yum.repos.d/kubernetes-sync.repo --newest-only --delete
[root@mirrors ~]# createrepo_c /var/www/html/kubernetes

# containerd 源:
[root@mirrors ~]# vim /etc/yum.repos.d/docker-ce-sync.repo
[docker-ce-stable]
name=Docker CE Stable - EL9
baseurl=https://mirrors.tuna.tsinghua.edu.cn/docker-ce/linux/rocky/9/x86_64/stable/
enabled=1
gpgcheck=0

[root@mirrors ~]# mkdir -p /var/www/html/docker-ce
[root@mirrors ~]# dnf reposync --download-path /var/www/html --repoid=docker-ce-stable --download-metadata \
  --config /etc/yum.repos.d/docker-ce-sync.repo --newest-only --delete
[root@mirrors ~]# createrepo_c /var/www/html/docker-ce-stable


# 每天自动同步:
```

```
[root@mirrors ~]# vim /etc/cron.daily/k8s-sync
#!/bin/bash

/usr/bin/dnf reposync --download-path /var/www/html --repoid=kubernetes --download-metadata --config
/etc/yum.repos.d/kubernetes-sync.repo --newest-only --delete
/usr/bin/createrepo_c /var/www/html/kubernetes

/usr/bin/dnf reposync --download-path /var/www/html --repoid=docker-ce-stable --download-metadata --config
/etc/yum.repos.d/docker-ce-sync.repo --newest-only --delete
/usr/bin/createrepo_c /var/www/html/docker-ce-stable
```

客户端配置源：

```
[root@mirrors ~]# vim /etc/yum.repos.d/k8s-containerd.repo
[docker-ce]
name=Docker CE - containerd
baseurl=http://10.8.20.8/docker-ce-stable/
enabled=1
gpgcheck=0

[kubernetes]
name=Kubernetes v1.36
baseurl=http://10.8.20.8/kubernetes/
enabled=1
gpgcheck=0
```

§22.3 容器镜像库

Kubernetes 1.36 不再使用 docker 容器运行时，一般使用 containerd，不过它们的容器镜像是完全兼容的。首先参考“§3.1.1 安装”安装好 docker，然后执行下面的操作（直接采用 registry 容器镜像）。在生产环境中建议采用 harbor 或者 Nexus Repository 搭建私有镜像库。

常用的镜像库有 docker.io、registry.k8s.io、quay.io 和 ghcr.io，常见的应用镜像（nginx、mysql、wordpress 等）一般放在 docker.io 中，带有 library/前缀；而 K8s 的核心组件镜像在 registry.k8s.io 中，没有前缀；quay.io 存放部分 CNI 插件、Prometheus 等常用 quay 镜像，通常带发布方前缀（如 prometheus/node-exporter、jetstack/cert-manager）。下面我们针对这三个镜像库建立本地源，方便学生做实验。

1. 构建私有镜像环境（假如电脑的 Ip 为 10.8.20.8）：

```
[root@mirrors ~]# mkdir -p /var/www/html/registry/{dockerio,k8sio,quayio,ghcrio}

[root@mirrors ~]# vim /var/www/html/registry/docker-compose.yml
services:
  registry-dockerio:
    image: registry:3.1.1
    container_name: registry-dockerio
    restart: always
    ports:
      - "5000:5000"
    volumes:
      - /var/www/html/registry/dockerio:/var/lib/registry
    environment:
      REGISTRY_STORAGE_DELETE_ENABLED: true

  registry-k8sio:
    image: registry:3.1.1
    container_name: registry-k8sio
    restart: always
    ports:
      - "5001:5000"
    volumes:
      - /var/www/html/registry/k8sio:/var/lib/registry
    environment:
      REGISTRY_STORAGE_DELETE_ENABLED: true

  registry-quayio:
    image: registry:3.1.1
```

https://www.veryopen.cn

```
    container_name: registry-quayio
    restart: always
    ports:
      - "5002:5000"
    volumes:
      - /var/www/html/registry/quayio:/var/lib/registry
    environment:
      REGISTRY_STORAGE_DELETE_ENABLED: true

  registry-ghcrio:
    image: registry:3.1.1
    container_name: registry-ghcrio
    restart: always
    ports:
      - "5003:5000"
    volumes:
      - /var/www/html/registry/ghcrio:/var/lib/registry
    environment:
      REGISTRY_STORAGE_DELETE_ENABLED: true


[root@mirrors ~]# docker pull docker.io/registry:3.1.1
[root@mirrors ~]# docker compose -f /var/www/html/registry/docker-compose.yml up -d

[root@mirrors ~]# docker run -d -p 5000:5000 --restart=always -v /var/www/html/registry/dockerio:/var/lib/registry\
 --name registry-dockerio registry:3.1.1

[root@mirrors ~]# docker compose -f /var/www/html/registry/docker-compose.yml ps
CONTAINER ID   IMAGE            COMMAND                  CREATED         STATUS         PORTS                                                  NAMES
9c6f08c2452e   registry:3.1.1   "/entrypoint.sh /etc…"   3 minutes ago   Up 3 minutes   0.0.0.0:5002->5000/tcp, [::]:5002->5000/tcp            registry-quayio
61dcc3d7b27a   registry:3.1.1   "/entrypoint.sh /etc…"   3 minutes ago   Up 3 minutes   0.0.0.0:5001->5000/tcp, [::]:5001->5000/tcp            registry-k8sio
3468ed3ecc9e   registry:3.1.1   "/entrypoint.sh /etc…"   3 minutes ago   Up 3 minutes   0.0.0.0:5000->5000/tcp, [::]:5000->5000/tcp            registry-dockerio
d7b2ed3ec6fc   registry:3.1.1   "/entrypoint.sh /etc…"   3 minutes ago   Up 3 minutes   0.0.0.0:5003->5000/tcp, [::]:5003->5000/tcp            registry-ghcrio

[root@mirrors ~]# firewall-cmd --add-port=5000-5003/tcp --permanent
[root@mirrors ~]# firewall-cmd --reload
```

下面验证查看私有镜像库：

```
[root@mirrors ~]# curl -XGET http://10.8.20.8:5000/v2/_catalog | jq
{"repositories":[]}
[root@mirrors ~]# curl -XGET http://10.8.20.8:5001/v2/_catalog | jq
{"repositories":[]}
[root@mirrors ~]# curl -XGET http://10.8.20.8:5002/v2/_catalog | jq
{"repositories":[]}
[root@mirrors ~]# curl -XGET http://10.8.20.8:5003/v2/_catalog | jq
{"repositories":[]}
```

2. 配置 Docker Registry：

```
编辑/etc/docker/daemon.json 文件，添加以下内容，以允许 Docker 客户端通过非 https 协议连接到自建的镜像库：
[root@mirrors ~]# vim /etc/docker/daemon.json
{
    "insecure-registries": ["10.8.20.8:5000"]
}

重启 docker 服务：
[root@mirrors ~]# systemctl restart docker

测试能否链接到本地镜像库：
[root@mirrors ~]# docker login 10.8.20.8:5000
Username:
Error: Non-null Username Required
```

3. 往私有镜像库中推送镜像

```
# 3.1 往 10.8.20.8:5000 仓库中推送镜像：
[root@mirrors ~]# docker pull nginx:latest                                                    #从公共镜像库下拉镜像
[root@mirrors ~]# docker tag nginx:latest localhost:5000/library/nginx:latest
[root@mirrors ~]# docker push localhost:5000/library/nginx:latest
# 或从其他电脑 docker save nginx:latest -o nginx.tar，然后拷贝道本地，再 docker load -i nginx.tar，最后打标后推送道本地库中

[root@mirrors ~]# ll /var/www/html/registry/dockerio/docker/registry/v2/repositories/library/
total 0
drwxr-xr-x 5 root root 55 Dec 19 14:31 nginx

[root@mirrors ~]# docker pull busybox:latest
[root@mirrors ~]# docker tag busybox:latest localhost:5000/library/busybox:latest
[root@mirrors ~]# docker push localhost:5000/library/busybox:latest

[root@mirrors ~]# docker pull mariadb:latest
[root@mirrors ~]# docker tag mariadb:latest localhost:5000/library/mariadb:latest
[root@mirrors ~]# docker push localhost:5000/library/mariadb:latest

[root@mirrors ~]# docker pull wordpress:latest
[root@mirrors ~]# docker tag wordpress:latest localhost:5000/library/wordpress:latest
[root@mirrors ~]# docker push localhost:5000/library/wordpress:latest

[root@mirrors ~]# docker pull grafana/grafana:13.0.2
[root@mirrors ~]# docker tag grafana/grafana:13.0.2 localhost:5000/library/grafana/grafana:13.0.2
[root@mirrors ~]# docker push localhost:5000/library/grafana/grafana:13.0.2
```

```
[root@mirrors ~]# docker pull hello-world
[root@mirrors ~]# docker tag hello-world:latest localhost:5000/library/hello-world:latest
[root@mirrors ~]# docker push localhost:5000/library/hello-world:latest

[root@mirrors ~]# docker pull debian
[root@mirrors ~]# docker tag debian:latest localhost:5000/library/debian:latest
[root@mirrors ~]# docker push localhost:5000/library/debian:latest

[root@mirrors ~]# curl -XGET http://10.8.20.8:5000/v2/_catalog                    #查看库中的镜像(或用浏览器打开)
{"repositories":["library/busybox","library/grafana/grafana","library/hello-
world","library/mariadb","library/nginx","library/wordpress"]}
```

如果上面这样推送，客户端使用时有问题。那就可以按照下面的方法做：

```
//在其他电脑上操作（先安装 containerd.io，并且启动 containerd 服务）:
root@instance-2:~# ctr -n k8s.io images pull --platform linux/amd64 docker.io/library/busybox:latest
root@instance-2:~# ctr -n k8s.io images export busybox_latest.tar docker.io/library/busybox:latest
//把文件 busybox_latest.tar 传递过来:
[root@mirrors ~]# ctr -n k8s.io images import busybox_latest.tar
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 docker.io/library/busybox:latest \
 localhost:5000/library/busybox:latest
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5000/library/busybox:latest
```

```
# 3.2 往 10.8.20.8:5001 仓库中推送镜像:
//在其他电脑上操作:
[root@others ~]# kubeadm config images list --kubernetes-version=v1.36.4
registry.k8s.io/kube-apiserver:v1.36.4
registry.k8s.io/kube-controller-manager:v1.36.4
registry.k8s.io/kube-scheduler:v1.36.4
registry.k8s.io/kube-proxy:v1.36.4
registry.k8s.io/coredns/coredns:v1.14.2
registry.k8s.io/pause:3.10.2
registry.k8s.io/etcd:3.6.8-0

[root@others ~]# kubeadm config images pull --kubernetes-version=v1.36.4 \
  --image-repository=registry.aliyuncs.com/google_containers
[root@others ~]# ctr -n k8s.io image ls -q | grep -v sha256
registry.aliyuncs.com/google_containers/coredns:v1.14.2
registry.aliyuncs.com/google_containers/etcd:3.6.8-0
registry.aliyuncs.com/google_containers/kube-apiserver:v1.36.4
registry.aliyuncs.com/google_containers/kube-controller-manager:v1.36.4
registry.aliyuncs.com/google_containers/kube-proxy:v1.36.4
registry.aliyuncs.com/google_containers/kube-scheduler:v1.36.4
registry.aliyuncs.com/google_containers/pause:3.10.2

[root@others ~]# ctr -n k8s.io images export coredns_v1.14.2.tar registry.aliyuncs.com/google_containers/coredns:v1.14.2
[root@others ~]# ctr -n k8s.io images export etcd_3.6.8-0.tar registry.aliyuncs.com/google_containers/etcd:3.6.8-0
[root@others ~]# ctr -n k8s.io images export kube-apiserver_v1.36.4.tar registry.aliyuncs.com/google_containers/kube-apiserver:v1.36.4
[root@others ~]# ctr -n k8s.io images export kube-controller-manager_v1.36.4.tar \
 registry.aliyuncs.com/google_containers/kube-controller-manager:v1.36.4
[root@others ~]# ctr -n k8s.io images export kube-proxy_v1.36.4.tar registry.aliyuncs.com/google_containers/kube-proxy:v1.36.4
[root@others ~]# ctr -n k8s.io images export kube-scheduler_v1.36.4.tar registry.aliyuncs.com/google_containers/kube-scheduler:v1.36.4
[root@others ~]# ctr -n k8s.io images export pause_3.10.2.tar registry.aliyuncs.com/google_containers/pause:3.10.2

//在 mirrors 上操作（先把到出的 tar 文件转递道 mirrors）:
[root@mirrors ~]# ctr -n k8s.io images import coredns_v1.14.2.tar
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 registry.aliyuncs.com/google_containers/coredns:v1.14.2 localhost:5001/coredns:v1.14.2
[root@mirrors ~]# ctr -n k8s.io images tag localhost:5001/coredns:v1.14.2 localhost:5001/coredns/coredns:v1.14.2
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5001/coredns/coredns:v1.14.2

[root@mirrors ~]# ctr -n k8s.io images import etcd_3.6.8-0.tar
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 registry.aliyuncs.com/google_containers/etcd:3.6.8-0 localhost:5001/etcd:3.6.8-0
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5001/etcd:3.6.8-0

[root@mirrors ~]# ctr -n k8s.io images import kube-apiserver_v1.36.4.tar
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 registry.aliyuncs.com/google_containers/kube-apiserver:v1.36.4 localhost:5001/kube-apiserver:v1.36.4
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5001/kube-apiserver:v1.36.4

[root@mirrors ~]# ctr -n k8s.io images import kube-controller-manager_v1.36.4.tar
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 registry.aliyuncs.com/google_containers/kube-controller-manager:v1.36.4 \
 localhost:5001/kube-controller-manager:v1.36.4
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5001/kube-controller-manager:v1.36.4

[root@mirrors ~]# ctr -n k8s.io images import kube-proxy_v1.36.4.tar
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 registry.aliyuncs.com/google_containers/kube-proxy:v1.36.4 localhost:5001/kube-proxy:v1.36.4
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5001/kube-proxy:v1.36.4

[root@mirrors ~]# ctr -n k8s.io images import kube-scheduler_v1.36.4.tar
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
```

```
 registry.aliyuncs.com/google_containers/kube-scheduler:v1.36.4 localhost:5001/kube-scheduler:v1.36.4
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5001/kube-scheduler:v1.36.4

[root@mirrors ~]# ctr -n k8s.io images import pause_3.10.2.tar
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 registry.aliyuncs.com/google_containers/pause:3.10.2 localhost:5001/pause:3.10.2
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5001/pause:3.10.2

[root@mirrors ~]# curl -XGET http://10.8.20.8:5001/v2/_catalog
{"repositories":["coredns","etcd","kube-apiserver","kube-controller-manager","kube-proxy","kube-
scheduler","pause"]}

[root@mirrors ~]# ctr -n k8s.io images pull --platform linux/amd64 \
 registry.k8s.io/kube-state-metrics/kube-state-metrics:v2.19.0
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 registry.k8s.io/kube-state-metrics/kube-state-metrics:v2.19.0 \
 localhost:5001/kube-state-metrics/kube-state-metrics:v2.19.0
[root@mirrors ~]# ctr -n k8s.io images push --plain-http \
 localhost:5001/kube-state-metrics/kube-state-metrics:v2.19.0

[root@mirrors ~]# ctr -n k8s.io images pull --platform linux/amd64 \
 registry.k8s.io/prometheus-adapter/prometheus-adapter:v0.12.0
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 registry.k8s.io/prometheus-adapter/prometheus-adapter:v0.12.0 \
 localhost:5001/prometheus-adapter/prometheus-adapter:v0.12.0
[root@mirrors ~]# ctr -n k8s.io images push --plain-http \
 localhost:5001/prometheus-adapter/prometheus-adapter:v0.12.0

# 3.3 往 10.8.20.8:5002 中推送镜像:
[root@mirrors ~]# ctr -n k8s.io images pull --platform linux/amd64 quay.io/prometheus/alertmanager:v0.33.0
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 quay.io/prometheus/alertmanager:v0.33.0 \
 localhost:5002/prometheus/alertmanager:v0.33.0
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5002/prometheus/alertmanager:v0.33.0

[root@mirrors ~]# ctr -n k8s.io images pull --platform linux/amd64 quay.io/prometheus/blackbox-exporter:v0.28.0
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 quay.io/prometheus/blackbox-exporter:v0.28.0\
 localhost:5002/prometheus/blackbox-exporter:v0.28.0
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5002/prometheus/blackbox-exporter:v0.28.0

[root@mirrors ~]# ctr -n k8s.io images pull --platform linux/amd64 quay.io/prometheus/node-exporter:v1.11.1
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 quay.io/prometheus/node-exporter:v1.11.1 \
 localhost:5002/prometheus/node-exporter:v1.11.1
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5002/prometheus/node-exporter:v1.11.1

[root@mirrors ~]# ctr -n k8s.io images pull --platform linux/amd64 quay.io/prometheus/prometheus:v3.12.0
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 quay.io/prometheus/prometheus:v3.12.0 \
 localhost:5002/prometheus/prometheus:v3.12.0
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5002/prometheus/prometheus:v3.12.0

[root@mirrors ~]# ctr -n k8s.io images pull --platform linux/amd64 quay.io/brancz/kube-rbac-proxy:v0.22.0
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 quay.io/brancz/kube-rbac-proxy:v0.22.0 \
 localhost:5002/brancz/kube-rbac-proxy:v0.22.0
[root@mirrors ~]# ctr -n k8s.io images push --plain-http localhost:5002/brancz/kube-rbac-proxy:v0.22.0

[root@mirrors ~]# ctr -n k8s.io images pull --platform \
 linux/amd64 quay.io/prometheus-operator/prometheus-config-reloader:v0.78.2
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 quay.io/prometheus-operator/prometheus-config-reloader:v0.78.2 \
 localhost:5002/prometheus-operator/prometheus-config-reloader:v0.78.2
[root@mirrors ~]# ctr -n k8s.io images push --plain-http \
 localhost:5002/prometheus-operator/prometheus-config-reloader:v0.78.2

[root@mirrors ~]# ctr -n k8s.io images pull --platform linux/amd64 \
 quay.io/prometheus-operator/prometheus-operator:v0.92.0
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
 quay.io/prometheus-operator/prometheus-operator:v0.92.0 \
 localhost:5002/prometheus-operator/prometheus-operator:v0.92.0
[root@mirrors ~]# ctr -n k8s.io images push --plain-http \
 localhost:5002/prometheus-operator/prometheus-operator:v0.92.0



# 3.4 往 10.8.20.8:5003 中推送镜像:
[root@mirrors ~]# docker pull ghcr.io/jimmidyson/configmap-reload:v0.15.0
[root@mirrors ~]# docker tag ghcr.io/jimmidyson/configmap-reload:v0.15.0 \
 localhost:5003/jimmidyson/configmap-reload:v0.15.0
[root@mirrors ~]# docker push localhost:5003/jimmidyson/configmap-reload:v0.15.0

# 推入 flannel 网络插件:
[root@mirrors ~]# ctr -n k8s.io images import flannel_v0.28.10.tar
[root@mirrors ~]# ctr -n k8s.io images import flannel-cni-plugin_v1.9.1-flannel3.tar
[root@mirrors ~]# ctr -n k8s.io image ls -q | grep -v sha256
ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1-flannel3
ghcr.io/flannel-io/flannel:v0.28.10
……

[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 ghcr.io/flannel-io/flannel:v0.28.10 \
  localhost:5003/flannel-io/flannel:v0.28.10
[root@mirrors ~]# ctr -n k8s.io images convert --platform linux/amd64 \
ghcr.io/flannel-io/flannel-cni-plugin:v1.9.1-flannel3 localhost:5003/flannel-io/flannel-cni-plugin:v1.9.1-flannel3
```

```
[root@mirrors ~]#  ctr -n k8s.io images push --plain-http localhost:5003/flannel-io/flannel:v0.28.10
[root@mirrors ~]#  ctr -n k8s.io images push --plain-http localhost:5003/flannel-io/flannel-cni-plugin:v1.9.1-
flannel3

[root@mirrors ~]# curl -XGET http://10.8.20.8:5003/v2/_catalog | jq
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   102  100   102    0     0  36783      0 --:--:-- --:--:-- --:--:-- 51000
{
  "repositories": [
    "flannel-io/flannel",
    "flannel-io/flannel-cni-plugin",
    "jimmidyson/configmap-reload"
  ]
}
```

删除本地仓库中的镜像的方法（比如 busybox）：

```
docker compose -f /var/www/html/registry/docker-compose.yml down
rm -rf /var/www/html/registry/dockerio/docker/registry/v2/repositories/busybox/
docker compose -f /var/www/html/registry/docker-compose.yml up -d
```

4. 配置客户端

4.1 针对 docker：

```
[root@client ~]# vim /etc/docker/daemon.json
{
    "registry-mirrors": [
        "https://docker.mirrors.tuna.tsinghua.edu.cn",
        "https://docker.mirrors.ustc.edu.cn",
        "https://docker.mirrors.nju.edu.cn",
        "https://docker.xuanyuan.me",
        "https://docker.m.daocloud.io",
        "http://10.8.20.8:5000"
    ],
    "insecure-registries":[
        "10.8.20.8:5000",
        "10.8.20.8:5001",
        "10.8.20.8:5002",
        "10.8.20.8:5003"
    ]
}

[root@client ~]# systemctl daemon-reload
[root@client ~]# systemctl restart docker
[root@client ~]# docker pull 10.8.20.8:5000/library/busybox:latest
```

docker 只能重定向 docker.io，不支持其他域名，比如 docker pull nginx 就会转向到本地镜像仓库 http://10.8.20.8:5000 去拉去，但是 docker pull registry.k8s.io/kube-apiserver:v1.36.4 就继续走官方库 registry.k8s.io，所以为了从本地其他仓库拉取 kube-apiserver:v1.36.4，就必须 docker pull 10.8.20.8:5001/kube-apiserver:v1.36.4。

4.2 针对 podman：

```
[root@client ~]# vim /etc/containers/registries.conf
unqualified-search-registries = ["docker.io"]

[[registry]]
location = "docker.io"

[[registry.mirror]]
location = "10.8.20.8:5000"
insecure = true

[[registry]]
location = "registry.k8s.io"

[[registry.mirror]]
location = "10.8.20.8:5001"
insecure = true

[[registry]]
location = "quay.io"

[[registry.mirror]]
location = "10.8.20.8:5002"
insecure = true

[[registry]]
location = "ghcr.io"

[[registry.mirror]]
```

```
location = "10.8.20.8:5003"
insecure = true

[root@client ~]# podman pull nginx
[root@client ~]# podman pull registry.k8s.io/pause:3.10.2
```

4.3 针对 containerd：

```
[root@client ~]# vim /etc/containerd/config.toml
......
    [plugins.'io.containerd.cri.v1.images'.registry]
      config_path = '/etc/containerd/certs.d'
......

[root@client ~]# mkdir -p /etc/containerd/certs.d/{docker.io,registry.k8s.io,quay.io,ghcr.io}

[root@client ~]# vim /etc/containerd/certs.d/docker.io/hosts.toml
server = "https://registry-1.docker.io"

[host."http://10.8.20.8:5000"]
  capabilities = ["pull", "resolve"]
  skip_verify = true

[root@client ~]# vim /etc/containerd/certs.d/registry.k8s.io/hosts.toml
server = "https://registry.k8s.io"

[host."http://10.8.20.8:5001"]
  capabilities = ["pull", "resolve"]
  skip_verify = true

[root@client ~]# vim /etc/containerd/certs.d/quay.io/hosts.toml
server = "https://quay.io"

[host."http://10.8.20.8:5002"]
  capabilities = ["pull", "resolve"]
  skip_verify = true

[root@client ~]# vim /etc/containerd/certs.d/ghcr.io/hosts.toml
server = "https://ghcr.io"

[host."http://10.8.20.8:5003"]
  capabilities = ["pull", "resolve"]
  skip_verify = true

[root@client ~]# systemctl restart containerd
```

配置号了之后，不管是手工拉取还是 yaml 文件中的镜像，都不需要特意加上 10.8.20.8:500?/前缀。

5. 试试成功否：

```
[root@master1 ~]# kubectl run nginx-web --image=nginx
或者
[root@master1 ~]# kubectl run nginx-web --image=10.8.20.8:5000/nginx
```

§22.4 Helm Chart 库

以前 harbor 也有 Chart 库功能，后来官方建议不要使用。这里采用 chartmuseum 来构建本地私有 Chart 库。

请先参考 § 16.5.2 小节安装 helm 工具。

一、构建 Chart 私有库

1. 构建 Chart 库

```
[root@mirrors ~]# mkdir -p /var/www/html/charts
[root@mirrors ~]# docker run -d -p 8080:8080 -e STORAGE=local -e STORAGE_LOCAL_ROOTDIR=/charts -v \
 /var/www/html/charts:/charts bitnami/chartmuseum

[root@mirrors ~]# docker ps
```

https://www.

```
CONTAINER ID   IMAGE                COMMAND                  CREATED          STATUS          PORTS                                       NAMES
6ba79357fb5b   bitnami/chartmuseum  "/opt/bitnami/script…"   58 minutes ago   Up 58 minutes   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   magical_shaw
61a843c66ca6   registry             "/entrypoint.sh /etc…"   23 hours ago     Up 23 hours     0.0.0.0:5000->5000/tcp, :::5000->5000/tcp   condescending_greider

[root@mirrors ~]# docker update --restart=always 6ba79357fb5b

[root@mirrors html]# firewall-cmd --add-port=8080/tcp --permanent
[root@mirrors html]# firewall-cmd --reload
```

2. 安装 cm-push 插件

```
[root@mirrors ~]# helm plugin install https://github.com/chartmuseum/helm-push
Downloading and installing helm-push v0.10.4 ...
https://github.com/chartmuseum/helm-push/releases/download/v0.10.4/helm-push_0.10.4_linux_amd64.tar.gz
Installed plugin: cm-push
```

3. 把 Chart 推送到私有 chart 库中

```
[root@mirrors ~]# helm repo add private http://127.0.01:8080

[root@mirrors ~]# helm pull bitnami/mariadb
[root@mirrors ~]# helm pull bitnami/nginx
[root@mirrors ~]# ll *.tgz
-rw-r--r-- 1 root root     48199 Dec 21 09:35 mariadb-14.1.4.tgz
-rw-r--r-- 1 root root     40432 Dec 23 09:44 nginx-15.5.1.tgz

[root@mirrors ~]# helm cm-push mariadb-14.1.4.tgz private
Pushing mariadb-14.1.4.tgz to private...
Done.
[root@mirrors ~]# helm cm-push nginx-15.5.1.tgz private
Pushing nginx-15.5.1.tgz to private...
Done.

[root@mirrors ~]# helm create cm-demo
Creating cm-demo
[root@mirrors ~]# tree cm-demo
cm-demo
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── serviceaccount.yaml
│   ├── service.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml

3 directories, 10 files

[root@mirrors ~]# helm package cm-demo
Successfully packaged chart and saved it to: /root/cm-demo-0.1.0.tgz

[root@mirrors ~]# helm cm-push cm-demo-0.1.0.tgz private
Pushing cm-demo-0.1.0.tgz to private...
Done.
或者采用下面的命令推送:
[root@mirrors ~]# curl --data-binary "@cm-demo-0.1.0.tgz" http://localhost:8080/api/charts
{"saved":true}
```

4. 删除私有库中的 Chart

```
[root@mirrors ~]# helm search repo private
NAME                    CHART VERSION   APP VERSION     DESCRIPTION
private/cm-demo         0.1.0           1.16.0          A Helm chart for Kubernetes
private/mariadb         14.1.4          11.1.3          MariaDB is an open source, community-developed ...
private/nginx           15.5.1          1.25.3          NGINX Open Source is a web server that can be a...
private/wordpress       18.1.29         6.4.2           WordPress is the world's most popular blogging ...

[root@mirrors ~]# curl  -X DELETE http://localhost:8080/api/charts/cm-demo/0.1.0
{"deleted":true}
```

删除 chart 的语法：curl -X DELETE http://localhost:8080/api/charts/名字/版本号

二、使用 Chart 私有库

1. 增加私有库

```
[root@master1 ~]# helm repo add gdpi http://10.8.20.8:8080
"gdpi" has been added to your repositories
[root@master1 ~]# helm repo list
NAME    URL
aliyun  https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts
bitnami https://charts.bitnami.com/bitnami
gdpi    http://10.8.20.8:8080
```

2. 安装

```
[root@master1 ~]# helm search repo gdpi
NAME            CHART VERSION   APP VERSION     DESCRIPTION
gdpi/mariadb    14.1.4          11.1.3          MariaDB is an open source, community-developed ...
gdpi/nginx      15.5.1          1.25.3          NGINX Open Source is a web server that can be a...
gdpi/wordpress  18.1.29         6.4.2           WordPress is the world's most popular blogging ...

[root@master1 ~]# helm install nginx-gdpi gdpi/nginx
[root@master1 ~]# helm list
NAME          NAMESPACE       REVISION        UPDATED                                 STATUS          CHART                   APP
VERSION
metrics       default         2               2023-12-18 16:14:07.333785895 +0800 CST deployed        metrics-server-6.6.5    0.6.4
nginx-gdpi    default         1               2023-12-23 11:14:17.940288826 +0800 CST deployed        nginx-15.5.1            1.25.3

[root@master1 ~]# kubectl get pods
[root@master1 ~]# kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS   AGE     IP           NODE    NOMINATED NODE READINESS GATES
nginx-gdpi-6b546f674-n5w9t  1/1     Running   0          2m17s   10.244.2.7  worker2 <none>         <none>
```

https://www.veryopc

第23章 排错

§23.1 docker 容器网络不通

原因是 docker 使用 iptables 防火墙，而 rocky 9 操作系统使用 firewalld 防火墙，如果 firewalld 服务被激活，那么就和 iptables 产生冲突，不过最新本的 docker 已经解决了 iptables 和 firewalld 兼容问题：在 firewalld 中创建 docker 区域，docker0 及其他由 docker 服务创建的网元都被加到 docker 区域，并且设置默认策略为 accept，这样经由 docker0 网桥的流量都默认通过 firewalld。

但是存在一种错乱现象，那就是 docker0 可能加入了 firewalld 的默认区域 public 中，这样 public 中的规则就对 docker0 流量起作用了，从而导致容器网络不通。

解决办法：

一、如果是防火墙冲突问题。

查看防火墙默认区域和 docker 区域中的规则：

```
[root@master1 ~]# firewall-cmd  --list-all
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: docker0 ens160
  sources:
  services: cockpit dhcpv6-client ssh
  ......

[root@master1 ~]# firewall-cmd  --zone=docker --list-all
docker (active)
  target: ACCEPT
  icmp-block-inversion: no
  interfaces: br-08965bac343d br-3610bfd19722
  sources:
  services:
  ......
```

发现 docker0 在 public 而不是 docker 区域中，把 docker0 移到 docker 中即可：

```
[root@master1 ~]# firewall-cmd --permanent --zone=public --remove-interface=docker0
success
[root@master1 ~]# firewall-cmd --permanent --zone=docker --change-interface=docker0
success
[root@master1 ~]# firewall-cmd --reload
success
```

然后重启 docker 服务：

```
[root@master1 ~]# rm -rf /var/lib/docker                                    #试试 rm /var/lib/docker/network/
[root@master1 ~]# killall -9 containerd
[root@master1 ~]# systemctl restart docker
```

然后再检查容器的网络是否通。

二、如果还不行，请继续往下。

1. 最好采用 busybox 容器来排错，此容器包含各种网络排错工作，包括 ip、ping 等：

```
[root@master1 ~]# docker pull busybox
[root@master1 ~]# docker run -it busybox
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

```
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
14: eth0@if15: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # ip r
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link  src 172.17.0.2
/ # ping 172.17.0.1
PING 172.17.0.1 (172.17.0.1): 56 data bytes
```

容器的网关为 172.17.0.1，发现网关 ping 不通。

2．检查宿主机上的 docker0 网桥：

```
[root@master1 ~]# ip a show docker0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:3c:a7:a9:42 brd ff:ff:ff:ff:ff:ff
       valid_lft forever preferred_lft forever
    inet6 fe80::42:3cff:fea7:a942/64 scope link
       valid_lft forever preferred_lft forever

[root@master1 ~]# nmcli d show docker0
GENERAL.DEVICE:                         docker0
GENERAL.TYPE:                           bridge
GENERAL.HWADDR:                         02:42:3C:A7:A9:42
GENERAL.MTU:                            1500
GENERAL.STATE:                          100 (连接（外部))
GENERAL.CONNECTION:                     --
GENERAL.CON-PATH:                       --
IP4.ADDRESS[1]:                         --
IP4.GATEWAY:                            --
IP6.GATEWAY:                            --
```

发现网桥没有配置 ip，本来应该配有 172.17.0.1 的，因为容器内部的默认网关就是 172.17.0.1。

3．临时手工配置 Ip：

```
[root@master1 ~]# ip addr add 172.17.0.1/16 dev docker0
```

再试试容器的网络，应该是通外网的。不过重启宿主机之后，docker0 又没有 Ip 了。还得采用下面的永久解决之道。

4．永久解决：

```
[root@master1 ~]# nmcli connection add type bridge ifname docker0 con-name docker0  ip4 172.17.0.1/16 autoconnect true
```

然后在容器中 ping 外网，应该是通的。如果容器里没有 ping 命令，采用下面的命令把 ping 命令从宿主机上拷贝进去：

```
[root@master1 ~]# which ping
/usr/sbin/ping

[root@master1 ~]# ll /usr/sbin/ping
lrwxrwxrwx. 1 root root 11  6月  8 19:23 /usr/sbin/ping -> ../bin/ping

[root@master1 ~]# ll /bin/ping
-rwxr-xr-x. 1 root root 78336  6月  8 19:23 /bin/ping

[root@master1 ~]# docker ps
CONTAINER ID   IMAGE           COMMAND       CREATED         STATUS         PORTS     NAMES
57b409368898   debian:latest   "/bin/bash"   7 seconds ago   Up 6 seconds             bold_cray

[root@master1 ~]# docker cp /bin/ping df4831fd71b7:/bin/                    #把 ping 命令拷贝到容器中
```

§23.2 Docker 官方镜像访问失败

在搜索或下拉 docker 官方镜像时较长时间没反应，最后报错：

```
Error response from daemon: Get "https://index.docker.io/v1/search?q=nginx&n=25": dial tcp 128.242.245.189:443: connect: connection refused.
```

说明没法访问 docker 的官方镜像源，解决的办法有几个：1 是更改为国内的镜像源；2 是搭建私有镜像库，下面分别介绍。

§ 23.2.1 更改为国内镜像源

§23.3 初始化时下拉镜像失败

1．从一台能访问外网的电脑上手工下拉镜像：

```
[root@localhost ~] ctr -n k8s.io images pull registry.aliyuncs.com/google_containers/coredns:v1.12.1
[root@localhost ~] ctr -n k8s.io images pull registry.aliyuncs.com/google_containers/etcd:3.6.4-0
[root@localhost ~] ctr -n k8s.io images pull registry.aliyuncs.com/google_containers/kube-apiserver:v1.34.0
[root@localhost ~] ctr -n k8s.io images pull registry.aliyuncs.com/google_containers/kube-controller-manager:v1.34.0
[root@localhost ~] ctr -n k8s.io images pull registry.aliyuncs.com/google_containers/kube-proxy:v1.34.0
[root@localhost ~] ctr -n k8s.io images pull registry.aliyuncs.com/google_containers/kube-scheduler:v1.34.0
[root@localhost ~] ctr -n k8s.io images pull registry.aliyuncs.com/google_containers/pause:3.10.1
```

2．把镜像导出到文件：

```
[root@localhost ~]# ctr -n k8s.io images export coredns.tar registry.aliyuncs.com/google_containers/coredns:v1.12.1
[root@localhost ~]# ctr -n k8s.io images export etcd.tar registry.aliyuncs.com/google_containers/etcd:3.6.4-0
[root@localhost ~]# ctr -n k8s.io images export kube-apiserver.tar registry.aliyuncs.com/google_containers/kube-apiserver:v1.34.0
[root@localhost ~]# ctr -n k8s.io images export kube-controller-manager.tar registry.aliyuncs.com/google_containers/kube-controller-manager:v1.34.0
[root@localhost ~]# ctr -n k8s.io images export kube-proxy.tar registry.aliyuncs.com/google_containers/kube-proxy:v1.34.0
[root@localhost ~]# ctr -n k8s.io images export kube-scheduler.tar registry.aliyuncs.com/google_containers/kube-scheduler:v1.34.0
[root@localhost ~]# ctr -n k8s.io images export pause.tar registry.aliyuncs.com/google_containers/pause:3.10.1
```

3．把导出的文件传递到 master1 上，然后导入镜像：

```
[root@master1 ~] ctr -n k8s.io images import coredns.tar
[root@master1 ~] ctr -n k8s.io images import etcd.tar
[root@master1 ~] ctr -n k8s.io images import kube-apiserver.tar
[root@master1 ~] ctr -n k8s.io images import kube-controller-manager.tar
[root@master1 ~] ctr -n k8s.io images import kube-proxy.tar
[root@master1 ~] ctr -n k8s.io images import kube-scheduler.tar
[root@master1 ~] ctr -n k8s.io images import pause.tar

[root@master1 ~] ctr -n k8s.io images ls -q | grep -v sha256
registry.aliyuncs.com/google_containers/coredns:v1.12.1
registry.aliyuncs.com/google_containers/etcd:3.6.4-0
registry.aliyuncs.com/google_containers/kube-apiserver:v1.34.0
registry.aliyuncs.com/google_containers/kube-controller-manager:v1.34.0
registry.aliyuncs.com/google_containers/kube-proxy:v1.34.0
registry.aliyuncs.com/google_containers/kube-scheduler:v1.34.0
registry.aliyuncs.com/google_containers/pause:3.10.1

[root@master1 ~]# crictl images ls
IMAGE                                                 TAG                 IMAGE ID            SIZE
registry.aliyuncs.com/google_containers/coredns       v1.12.1             52546a367cc9e       22.4MB
registry.aliyuncs.com/google_containers/etcd          3.6.4-0             5f1f5298c888d       74.3MB
registry.aliyuncs.com/google_containers/kube-apiserver v1.34.0            90550c43ad2bc       27.1MB
registry.aliyuncs.com/google_containers/kube-controller-manager v1.34.0   a0af72f2ec6d6       22.8MB
registry.aliyuncs.com/google_containers/kube-proxy    v1.34.0             df0860106674d       26MB
registry.aliyuncs.com/google_containers/kube-scheduler v1.34.0            46169d968e920       17.4MB
registry.aliyuncs.com/google_containers/pause         3.10.1              cd073f4c5f6a8       320kB
```

https://www.veryopen.org

§23.4 手工拉下容器镜像

当安装 flannel 网络插件时，不能直接从官网下拉容器镜像，需要从一台能上外网的机器上手工下拉，参考如下命令。

1. 先找出 flannel 插件需要哪些容器镜像：

```
[root@master1 ~]# wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

[root@master1 ~]# grep image kube-flannel.yml
        image: docker.io/flannel/flannel-cni-plugin:v1.1.2
       #image: docker.io/rancher/mirrored-flannelcni-flannel-cni-plugin:v1.1.2
        image: docker.io/flannel/flannel:v0.22.0
       #image: docker.io/rancher/mirrored-flannelcni-flannel:v0.22.0
        image: docker.io/flannel/flannel:v0.22.0
       #image: docker.io/rancher/mirrored-flannelcni-flannel:v0.22.0
```

即需要两个插件，上面红色部分所示。

然后在一台能上外网的电脑上手工下拉镜像（先要安装 containerd）：

```
下拉镜像 flannel-cni-plugin:v1.5.1-flannel2:
root@instance-2:~# ctr -n k8s.io images pull --platform linux/amd64 docker.io/flannel/flannel-cni-plugin:v1.5.1-flannel2
docker.io/flannel/flannel-cni-plugin:v1.5.1-flannel2:                             resolved       |++++++++++++++++++++++++++++++++++++++|
index-sha256:5d4fb9f90389a33b397fba4c8f371454c21aa146696aec46481214892e66c1b8:    done           |++++++++++++++++++++++++++++++++++++++|
manifest-sha256:6e700d30d2c9316b4f54f9d0b1423a2d15290bf082f5d1fbd87f9f25173a374c: done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:af6afc41c64c2dfd64e8963de64c5f5e8ade534cd516efbe12a4c40cd432b842:    done           |++++++++++++++++++++++++++++++++++++++|
config-sha256:962fd97b50f9c6693b3e1ff4786c5643eef5eefafca5f43d3240ae8b87cbac2e:   done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:32b772fa507186eddade1aa8a0f01f5ceacba1fa94a5bb968eb355ac417baca3:    done           |++++++++++++++++++++++++++++++++++++++|
elapsed: 1.8 s                                                                    total:  4.6 Mi (2.6 MiB/s)
unpacking linux/amd64 sha256:5d4fb9f90389a33b397fba4c8f371454c21aa146696aec46481214892e66c1b8...
done: 604.605049ms

导出镜像 flannel-cni-plugin:v1.5.1-flannel2 为 flannel-cni-plugin-1.5.1.tar:
root@instance-2:~# ctr -n k8s.io images export flannel-cni-plugin-1.5.1.tar \
  docker.io/flannel/flannel-cni-plugin:v1.5.1-flannel2

下拉镜像 flannel:v0.22.0:
root@instance-1:~# ctr -n k8s.io images pull  docker.io/flannel/flannel:v0.26.1
docker.io/flannel/flannel:v0.26.1:                                                resolved       |++++++++++++++++++++++++++++++++++++++|
index-sha256:8bdea8aed559e45f78f52b8f782852c66a791cb04e2b938d5c5f80b97fcdea33:    done           |++++++++++++++++++++++++++++++++++++++|
manifest-sha256:35c519c04fcc6d09672363578db3032a4f557d23ff0ed7cad42728974434f6e1: done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:689ce0ce68b1760d49aceabc4d127777735b6bd981fd0670062b3e946e29a440:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:362a3f4c5c20d1fa186c15d22746d8e541812015f910097dc1c3e556055d6360:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:79a208cb81f33e3b09f05468b0648b59fc83ba6487cf2e928ec0a21cb73b4d78:    done           |++++++++++++++++++++++++++++++++++++++|
config-sha256:4c92107ad4cfea9f8d78ca46791c2ade35996da235f3f813091a86b517fef65a:   done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:13de5e2ee01f50e0c9f03c104ebc3defc05c61d33f687c4f3fd3440f950351c0:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:1ded544df1326e2a2234676cca54321fb326c75d92ee3ba84eb6396b534ab3fa:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:32b772fa507186eddade1aa8a0f01f5ceacba1fa94a5bb968eb355ac417baca3:    exists         |++++++++++++++++++++++++++++++++++++++|
layer-sha256:821570fb802bb34612a4506baae99c8e2bd3904d7044f6eb9f3dc5ff4c610282:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:68bbdc6197a01211816b6cf44e77f44a3ae988d785c98a2ebf227679120a16d4:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:4f4fb700ef54461cfa02571ae0db9a0dc1e0cdb5577484a6d75e68dc38e8acc1:    done           |++++++++++++++++++++++++++++++++++++++|
layer-sha256:b56a5545cb8cfee722dbb696c84e3bf51bd45f2463b84731fd1bf08b4620fad5:    done           |++++++++++++++++++++++++++++++++++++++|
elapsed: 2.5 s                                                                    total:  27.2 M (10.8 MiB/s)
unpacking linux/amd64 sha256:8bdea8aed559e45f78f52b8f782852c66a791cb04e2b938d5c5f80b97fcdea33...
done: 2.026079227s


导出镜像 flannel:v0.26.1 为 flannel-0.26.1.tar:
root@instance-2:~# ctr -n k8s.io images export flannel-0.26.1.tar docker.io/flannel/flannel:v0.26.1
```

最后把导出的容器镜像文件 flannel-cni-plugin-1.5.1.tar 和 flannel-0.26.1.tar 下载到全部的节点上，并导入文件中的镜像（ctr -n k8s.io images import flannel-0.26.1.tar）。

§23.5 在 pod 中无法 ping 通外部的域名

pod 默认的 DNS 策略采用 ClusterFirst（由部署文件中的 spec.dnsPolicy 定义），此策略表示与配置的集群域后缀（cluster.local）不匹配的任何 DNS 查询（例如 "www.veryopen.org"） 都将转发到从节点继承的上游 DNS 服务器。因此 DNS 查询先走集群内的 CoreDNS（默认两个工作节点上有类似 coredns-xxxxx 的 pod），如果查不到再走公共的 DNS 服务器。

首先确保两个工作节点上各存在一个 coredns：

```
[root@master1 ~]# kubectl get pods -l k8s-app=kube-dns --all-namespaces -o wide
NAMESPACE     NAME                       READY   STATUS    RESTARTS   AGE     IP           NODE      NOMINATED NODE   READINESS GATES
kube-system   coredns-66f779496c-59nlb   1/1     Running   0          83m     10.244.1.2   worker1   <none>           <none>
kube-system   coredns-66f779496c-ltrkq   1/1     Running   0          3h50m   10.244.3.3   worker3   <none>           <none>
```

如果某个工作节点上缺少且另一个节点上多了一个，那么就把多余的那个 pod 删掉（kubectl delete pod ...），这样就会自动在没有 coredns 的那个节点上启动一个 pod 了。如果问题还没有解决，请继续往下排错。

看看 coredns-xxxxx 的日志：

```
[root@master1 ~]# kubectl logs --namespace=kube-system  coredns-66f779496c-59nlb
.:53
[INFO] plugin/reload: Running configuration SHA512 =
591cf328cccc12bc490481273e738df59329c62c0b729d94e8b61db9961c2fa5f046dd37f1cf888b953814040d180f52594972691cd6ff41be96639138a43908
CoreDNS-1.10.1
linux/amd64, go1.20, 055b2c3
[root@master1 ~]# kubectl logs --namespace=kube-system  coredns-66f779496c-ltrkq
.:53
[INFO] plugin/reload: Running configuration SHA512 =
591cf328cccc12bc490481273e738df59329c62c0b729d94e8b61db9961c2fa5f046dd37f1cf888b953814040d180f52594972691cd6ff41be96639138a43908
CoreDNS-1.10.1
linux/amd64, go1.20, 055b2c3
```

没有错误日志。dns 对应的服务：

```
[root@master1 ~]# kubectl get service -n kube-system | grep kube-dns
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   3h54m
```

一个 service 对应 pod 网上的若干 endpoints：

```
[root@master1 ~]# kubectl get endpoints kube-dns -n kube-system
NAME       ENDPOINTS                                                  AGE
kube-dns   10.244.1.2:53,10.244.3.3:53,10.244.1.2:53 + 3 more...   3h55m
```

其他 pod 里的容器中定义的 dns 就是 k8s 集群中的 dns 服务的 ip，即 10.96.0.10。所以检查的思路就是：检查容器中的/etc/resolv.conf 文件，在容器中能否 ping 通 10.96.0.10 和其他的 pod ip 地址。

```
[root@master1 ~]# kubectl exec -it busybox -- sh
/ # cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10                                                                                                                #没问
题
options ndots:5
/ # ping 10.96.0.10
#ping 不通
PING 10.96.0.10 (10.96.0.10): 56 data bytes
^C
--- 10.96.0.10 ping statistics ---
4 packets transmitted, 0 packets received, 100% packet loss
/ # ping 10.244.3.3
#ping 不通
PING 10.244.3.3 (10.244.3.3): 56 data bytes
^C
--- 10.244.3.3 ping statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
```

可能还是节点上的防火墙问题，不妨先关闭防火墙试试。

§23.6 kubeadm init 报错

1. 运行 kubeadm init 时报错

[ERROR CRI]: container runtime is not running: output: E0606 12:41:12.605990 1307975 remote_runtime.go:925] "Status f service runtime.v1alpha2.RuntimeService"

表明容器运行时服务没有启动。

处理：rm -rf /etc/containerd/config.toml，systemctl restart containerd

2. 运行 kubeadm init 时报错

[ERROR ImagePull]: failed to pull image registry.k8s.io/kube-apiserver……

表明下拉镜像失败。

处理： kubeadm init --image-repository=registry.aliyuncs.com/google_containers，指明采用阿里云的镜像库。

3. 运行 kubeadm init 时显示下面的内容等待很久，并最后报错

……

[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from directory "/etc/kubernetes/manifests". This can take up to 4m0s

[apiclient] All control plane components are healthy after 40 seconds

原因是 kubelet 服务异常，kubelet 服务依赖 containerd 服务，kubelet 通过/etc/sysconfig/kubelet 配置文件联系 containerd 服务。所以要检查 containerd 服务是否正常启动，其次检查/etc/sysconfig/kubelet 文件配置是否正常。还可以查看/var/log/messages 日志文件中记录的详细错误信息（在文件末尾）。

处理：没有明确告诉 kubelet 使用哪个 container，修改文件/etc/sysconfig/kubelet，内容如下：

```
1. 检查 containerd 服务是否正常启动：
[root@master1 ~]# systemctl status containerd

2. 查看配置文件（注意是在一行上）：
[root@master1 ~]# vim /etc/sysconfig/kubelet
KUBELET_EXTRA_ARGS="--cgroup-driver=systemd --container-runtime-endpoint=unix:///run/containerd/containerd.sock --
```

https://www.veryopen.org

```
pod-infra-container-image=registry.aliyuncs.com/google_containers/pause:3.10.1"

3. 查看日志文件:
[root@master1 ~]# tail -n 30 /var/log/messages

4. 重启 kubelet 服务:
[root@master1 ~]# systemcl restart kubelet containerd
```

§23.7 手工导入镜像时报错

```
[root@worker2 ~]# ctr -n k8s.io images import nginx_alpine.tar
ctr: content digest sha256:26000523de52034cf92b9013a86d0fb55e738aa863ee980ed7904bc816b80e7c: not found
```

从网上拉镜像和导出时加上--all-platforms 参数，导入时不加--all-platforms 参数。

§23.8 部署应用时报错：下载镜像超时

```
[root@master1 ~]# kubectl describe pod busybox
......
Failed to pull image "busybox:latest": rpc error: code = Unknown desc = failed to pull and unpack image
"docker.io/library/busybox:latest"
......
```

在能访问外网的电脑上（比如谷歌云上的虚拟机）安装 containerd，然后采用下面的命令下载镜像 docker.io/library/busybox:latest 并导出来。

```
ctr -n k8s.io images pull docker.io/library/busybox:latest                                  #下载镜像
ctr -n k8s.io images export busybox_latest.tar docker.io/library/busybox:latest             #导出
```

最后把导出的镜像 busybox_latest.tar 传回国内。

§23.9 pod 之间网络不通

https://www.cnblogs.com/janeysj/p/17312312.html

https://www.verylopen.org

§23.10 pod 异常

1. 确保节点内存足够：master1 4G 或以上、worker? 3G 或以上，可用命令 free 查看内存。
2. 根据§14.4 小节，对阶段进行检查。
3. 确保节点的 DNS 为 8.8.8.8 和 9.9.9.9：

```
[root@allnodes ~]# more /etc/resolv.conf
# Generated by NetworkManager
nameserver 8.8.8.8
nameserver 9.9.9.9
```

如果不是，请修改：

```
[root@allnodes ~]# nmcli connection modify ens160 ipv4.dns "8.8.8.8 9.9.9.9"
[root@allnodes ~]# nmcli connection up ens160
```

4. 查看 Pod 的日志：

```
[root@master1 ~]# kubectl logs <pod 名称> -n <pod 所在的名字空间>
```

如：

```
[root@master1 ~]# kubectl logs coredns-855c4dd65d-djs6k -n kube-system
```

5. 查看 Pod 中的事件：

```
[root@master1 ~]# kubectl describe pod <pod 名称> -n <pod 所在的名字空间>
```

大部分问题是不能下载镜像（从国外下载镜像被“墙”或者超级慢），那么请再次检查所有节点的容器运行时的配置（）参考§14.3 小节），或者干脆可手工下载镜像并导入镜像（具体方法参考§23.3 小节）。

6. 删除异常 Pod，一会儿 k8s 集群会自动重建此 pod：

```
[root@master1 ~]# kubectl delete pod <pod 名称> -n <pod 所在的名字空间>
```

§23.11 重置 etcd 集群

往集群中加入节点错误的处理办法之一是重置集群，然后重新加入节点。

在 master1 上：

```
[root@master1 ~]# systemctl stop  etcd
[root@master1 ~]# rm -rf /var/lib/etcd/*
```

在 workder1 上：

```
[root@worker1 ~]# systemctl stop etcd
[root@worker1 ~]# rm -rf /var/lib/etcd/*
```

在 master1 上：

https://www.verycpen.org

```
[root@master1 ~]# svim /etc/etcd/etcd.conf
ETCD_NAME="etcd01"
ETCD_DATA_DIR="/var/lib/etcd"

#配置与客户端的通信:
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.11:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.11:2379"
ETCD_CLIENT_CERT_AUTH=true
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信:
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.11:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.11:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置 etcd 集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="new"

[root@master1 ~]# systemctl start etcd
[root@master1 ~]# etcdctl member add etcd02 --peer-urls=https://10.10.10.31:2380
```

在 workder1 上：

```
[root@worker1 ~]# svim /etc/etcd/etcd.conf
ETCD_NAME="etcd02"
ETCD_DATA_DIR="/var/lib/etcd"

#配置与客户端的通信:
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.31:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.31:2379"
ETCD_CLIENT_CERT_AUTH=true
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信:
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.31:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.31:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置 etcd 集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380,etcd02=https://10.10.10.31:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="existing"

[root@worker1 ~]# systemctl start etcd
```

在 master1 上：

```
[root@master1 ~]# svim /etc/etcd/etcd.conf
ETCD_NAME="etcd01"
ETCD_DATA_DIR="/var/lib/etcd"

#配置与客户端的通信:
ETCD_LISTEN_CLIENT_URLS="https://10.10.10.11:2379,https://127.0.0.1:2379"
ETCD_ADVERTISE_CLIENT_URLS="https://10.10.10.11:2379"
ETCD_CLIENT_CERT_AUTH=true
ETCD_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_CERT_FILE="/etc/etcd/pki/client.crt"
ETCD_KEY_FILE="/etc/etcd/pki/client.key"

#配置节点之间的通信:
ETCD_PEER_CLIENT_CERT_AUTH=true
ETCD_LISTEN_PEER_URLS="https://10.10.10.11:2380"
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://10.10.10.11:2380,etcd02=https://10.10.10.31:2380"
ETCD_PEER_TRUSTED_CA_FILE="/etc/etcd/pki/ca.crt"
ETCD_PEER_KEY_FILE="/etc/etcd/pki/peer.key"
ETCD_PEER_CERT_FILE="/etc/etcd/pki/peer.crt"

#配置 etcd 集群:
ETCD_INITIAL_CLUSTER="etcd01=https://10.10.10.11:2380"
```

```
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_INITIAL_CLUSTER_STATE="new"

[root@master1 ~]# systemctl restart etcd
```

§23.12 无法访问 wordpress 页面

当浏览 http://<任意节点的 ip>:30001 时报错“Error establishing a database connection”，表明连接数据库出错。

1．查看错误日志：

```
[root@master1 ~]# kubectl logs -l app=wordpress -n wordpress
```

2．检查能否手工登陆数据库。

```
[root@master1 ~]# kubectl get svc -n wordpress -o wide
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE   SELECTOR
mariadb-service   ClusterIP   10.96.118.101   <none>        3306/TCP   45m   app=mariadb-pod

[root@master1 ~]# mysql -h 10.96.118.101 -uroot -pA1b2C3 wordpress_db
……
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

如果不能，请回到 17.4.1 小节去检查数据库是否已经部署好。如果反馈数据库不存在，请手工创建数据库 wordpress_db:

```
[root@master1 ~]# mysql -h 10.96.118.101 -uroot -pA1b2C3
mysql> CREATE DATABASE wordpress_db DEFAULT CHARACTER SET utf8mb4;
mysql> exit
```

3．检查链接数据库的参数是否正确

检查~/wordpress/wp/wordpress-deployment.yaml 文件中的几个环境变量的值对不对，这些环境变量是 WORDPRESS_DB_HOST、WORDPRESS_DB_PASSWORD、WORDPRESS_DB_USER、WORDPRESS_DB_NAME。正确的值应为：

```
[root@master1 ~]# more ~/wordpress/wp/wordpress-deployment.yaml
……
        env:
        - name: WORDPRESS_DB_HOST
          value: mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mariadb-secret
              key: mariadb-root-password
        - name: WORDPRESS_DB_USER
          value: root
        - name: WORDPRESS_DB_NAME
          value: wordpress_db
```

再检查~/wordpress/db/mariadb-secret.yaml 文件，其内容应为：

```
[root@master1 ~]# more ~/wordpress/db/mariadb-secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mariadb-secret
  namespace: wordpress
type: Opaque
data:
```

```
  mariadb-root-password: QTFiMkMz
```

把 A1b2C3 编码为 Base64 格式：

```
[root@master1 ~]# echo -n "A1b2C3" | base64
QTFiMkMz
```

查看运行数据库的容器的主机名：

```
[root@master1 ~]# kubectl exec mariadb-wp-0 -n wordpress -- hostname -f
mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local
```

这个主机名被解析到 mariadb-wp-0 这个 pod 的 ip 地址，本例是 10.244.2.95，见下面命令输出中的黄色字体部分。也尝试把 mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local 替换成数据库服务名“mariadb-service”试试。

直接进入网站所在的容器中检查环境变量：

```
[root@master1 ~]# kubectl get pod -n wordpress -o wide
NAME                         READY   STATUS    RESTARTS       AGE     IP             NODE      NOMINATED NODE   READINESS
GATES
mariadb-wp-0                 1/1     Running   2 (3m4s ago)   6d22h   10.244.2.114   worker2   <none>           <none>
wordpress-785bc9cc8d-7fdc9   1/1     Running   2 (3m7s ago)   6d22h   10.244.1.119   worker1   <none>           <none>

[root@master1 ~]# kubectl exec wordpress-785bc9cc8d-7fdc9 -n wordpress -it -- bash
root@wordpress-785bc9cc8d-7fdc9:/var/www/html# env|grep WORDPRESS
WORDPRESS_SERVICE_SERVICE_PORT=80
WORDPRESS_SERVICE_PORT=tcp://10.96.189.53:80
WORDPRESS_SERVICE_SERVICE_PORT_HTTP=80
WORDPRESS_SERVICE_PORT_80_TCP_PROTO=tcp
WORDPRESS_DB_HOST=mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local
WORDPRESS_SERVICE_PORT_80_TCP_PORT=80
WORDPRESS_SERVICE_SERVICE_HOST=10.96.189.53
WORDPRESS_DB_PASSWORD=A1b2C3
WORDPRESS_SERVICE_PORT_80_TCP=tcp://10.96.189.53:80
WORDPRESS_SERVICE_PORT_80_TCP_ADDR=10.96.189.53
WORDPRESS_DB_USER=root
WORDPRESS_DB_NAME=wordpress_db

root@wordpress-785bc9cc8d-7fdc9:/var/www/html# exit
```

确保上面深红色的环境变量没错。

4. 检查部署（deployment）、服务（service）的标签选择器与 pod 是否一致：

```
[root@master1 ~]# kubectl get all -n wordpress -o wide --show-labels
NAME                         READY   STATUS    RESTARTS       AGE     IP             NODE      NOMINATED NODE   READINESS GATES   LABELS
pod/mariadb-wp-0             1/1     Running   3 (49m ago)    5d21h   10.244.2.95    worker2   <none>           <none>            app=mariadb-
pod,apps.kubernetes.io/pod-index=0,controller-revision-hash=mariadb-sts-55fbcf6748,statefulset.kubernetes.io/pod-name=mariadb-sts-0
pod/wordpress-65cb68bcf5-tnmd4   1/1     Running   1 (49m ago)    4d18h   10.244.1.100   worker1   <none>           <none>            app=wordpress,pod-
template-hash=65cb68bcf5

NAME                        TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE     SELECTOR          LABELS
service/mariadb-service     ClusterIP   10.96.54.218   <none>        3306/TCP       5d      app=mariadb-pod   app=mariadb
service/wordpress-service   NodePort    10.96.107.63   <none>        80:30001/TCP   4d18h   app=wordpress     <none>

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS   IMAGES      SELECTOR          LABELS
deployment.apps/wordpress   1/1     1            1           4d18h   wordpress    wordpress   app=wordpress     <none>
……
```

5. 尝试 ping 通 mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local：

```
[root@master1 ~]# kubectl exec -it busybox-pod -n myspace -- sh
/ # ping mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local
PING mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local (10.244.2.95): 56 data bytes
64 bytes from 10.244.2.95: seq=0 ttl=62 time=0.281 ms
64 bytes from 10.244.2.95: seq=1 ttl=62 time=0.821 ms
```

按 ctrl+c 中断 ping 命令，然后输入 exit 退出 pod。在网站所在的容器中 ping 数据库数据名：

```
由于容器里没有 ping 命令，需从宿主机拷贝过去：
[root@master1 ~]# kubectl cp /usr/bin/ping wordpress/wordpress-785bc9cc8d-7fdc9:/usr/sbin/
[root@master1 ~]# kubectl cp /lib64/libc.so.6 wordpress/wordpress-785bc9cc8d-7fdc9:/lib64/

进入容器再 ping：
[root@master1 ~]# kubectl exec wordpress-785bc9cc8d-7fdc9 -n wordpress -it -- bash
root@wordpress-785bc9cc8d-7fdc9:/var/www/html# ping mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local
PING mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local (10.244.2.114) 56(84) bytes of data.
64 bytes from mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local (10.244.2.114): icmp_seq=1 ttl=62 time=0.878 ms
64 bytes from mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local (10.244.2.114): icmp_seq=2 ttl=62 time=0.476 ms
```

```
64 bytes from mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local (10.244.2.114): icmp_seq=3 ttl=62 time=0.499 ms
^C

root@wordpress-785bc9cc8d-7fdc9:/var/www/html# exit
```

如果 ping 不通，说明 coreDNS 存在问题，进一步查看相应 pod 的状态：

```
[root@master1 ~]# kubectl get pods -n kube-system -o wide                                                    #查看 k8s 组件的状态
NAMESPACE     NAME                              READY   STATUS    RESTARTS      AGE    IP              NODE      NOMINATED NODE   READINESS GATES
kube-system   coredns-66f779496c-59nlb          1/1     Running   0             39m    10.244.1.2      worker1   <none>           <none>
kube-system   coredns-66f779496c-ltrkq          1/1     Running   0             3h5m   10.244.3.3      worker3   <none>           <none>
kube-system   etcd-master1                      1/1     Running   2 (64m ago)   3h6m   10.10.10.11     master1   <none>           <none>
kube-system   kube-apiserver-master1            1/1     Running   2 (64m ago)   3h6m   10.10.10.11     master1   <none>           <none>
kube-system   kube-controller-manager-master1   1/1     Running   2 (64m ago)   3h6m   10.10.10.11     master1   <none>           <none>
kube-system   kube-proxy-j77qt                  1/1     Running   1 (64m ago)   165m   10.10.10.31     worker1   <none>           <none>
kube-system   kube-proxy-lcb24                  1/1     Running   1 (64m ago)   165m   10.10.10.33     worker3   <none>           <none>
kube-system   kube-proxy-vk2tv                  1/1     Running   2 (64m ago)   3h5m   10.10.10.11     master1   <none>           <none>
kube-system   kube-proxy-vzn4n                  1/1     Running   1 (64m ago)   165m   10.10.10.32     worker2   <none>           <none>
kube-system   kube-scheduler-master1            1/1     Running   2 (64m ago)   3h6m   10.10.10.11     master1   <none>           <none>

[root@master1 ~]# kubectl get pods -n kube-flannel -o wide                                                   #查看网络插件的状态
NAMESPACE      NAME                    READY   STATUS    RESTARTS   AGE   IP              NODE      NOMINATED NODE   READINESS GATES
kube-flannel   kube-flannel-ds-8kmv6   1/1     Running   0          51m   10.10.10.33     worker3   <none>           <none>
kube-flannel   kube-flannel-ds-92w86   1/1     Running   0          51m   10.10.10.11     master1   <none>           <none>
kube-flannel   kube-flannel-ds-cwgpk   1/1     Running   0          51m   10.10.10.31     worker1   <none>           <none>
kube-flannel   kube-flannel-ds-gmcws   1/1     Running   0          51m   10.10.10.32     worker2   <none>           <none>
```

如果正常，请暂时用 mariadb-wp-0 这个 pod 的 ip（本例子为 10.244.2.95）替换 ~/wordpress/wp/wordpress-deployment.yaml 文件中的 mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local，然后使用命令 kubectl apply -f ~/wordpress/wp/wordpress-deployment.yaml 重建资源。最后试试能否打开网页。直接使用 ip 地址是临时性的解决办法，因为重启节点之后，跑数据库的 pod 的 ip 可能会改变。

6. 检查 wordpress 配置文件：

```
[root@master1 ~]# kubectl get pods -n wordpress -o wide
NAME                         READY   STATUS    RESTARTS   AGE   IP              NODE      NOMINATED NODE   READINESS GATES
mariadb-wp-0                 1/1     Running   0          24m   10.244.2.98     worker2   <none>           <none>
wordpress-785bc9cc8d-7fdc9   1/1     Running   0          21m   10.244.1.105    worker1   <none>           <none>
```

表明网站所在的 pod 调度在 worker1 上，下面在 worker1 上操作：

```
[root@worker1 ~]# vim /opt/data/wordpress/wp-config.php                                                     #检查下面几行代码
......
define( 'DB_NAME', 'wordpress_db' );
......
define( 'DB_USER', 'root' );
......
define( 'DB_PASSWORD', 'A1b2C3' );
......
define( 'DB_HOST', 'mariadb-wp-0.mariadb-service.wordpress.svc.cluster.local' );
......
```

语句默认是从 WORDPRESS_DB_NAME、WORDPRESS_DB_USER、WORDPRESS_DB_PASSWORD 和 WORDPRESS_DB_HOST 环境变量中获取登陆数据库的参数的。我们可以直接修改语句中连接数据库的参数，然后再次刷新浏览网站试试。

7. 实在没办法，就删除资源、数据库数据，然后重建全部资源对象：

```
[root@master1 ~]# kubectl get pods -n wordpress -o wide
NAME                         READY   STATUS    RESTARTS   AGE   IP              NODE      NOMINATED NODE   READINESS GATES
mariadb-wp-0                 1/1     Running   0          24m   10.244.2.98     worker2   <none>           <none>
wordpress-785bc9cc8d-7fdc9   1/1     Running   0          21m   10.244.1.105    worker1   <none>           <none>
```

查到跑数据库的 pod 被调度在 worker2 节点上，跑网站的 pod 被调度在 worker1 上。

下面删除所有资源对象：

```
[root@master1 ~]# kubectl delete -f ~/wordpress/wp/
[root@master1 ~]# kubectl delete -f ~/wordpress/db/
```

再删除 worker2（跑数据库 pod 的节点）上的数据库数据：

```
[root@worker2 ~]# rm -rf /opt/data/mariadb/*
```

删除 wordpress 页面文件（在跑网站 pod 的那个节点上操作，这里是 worker1）：

```
[root@worker1 ~]# rm -rf /opt/data/wordpress/*
```

重建全部资源对象：

```
[root@master1 ~]# kubectl create -f ~/wordpress/db/
[root@master1 ~]# kubectl create -f ~/wordpress/wp/
```

创建数据库 wordpress_db：

```
[root@master1 ~]# kubectl get service -n wordpress -o wide
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE   SELECTOR
mariadb-service     ClusterIP   10.96.50.169    <none>        3306/TCP       29m   app=mariadb-pod
wordpress-service   NodePort    10.96.189.53    <none>        80:30001/TCP   26m   app=wordpress

[root@master1 ~]# mysql -h 10.96.50.169 -uroot -pA1b2C3
mysql> CREATE DATABASE wordpress_db DEFAULT CHARACTER SET utf8mb4;
mysql> exit
```

最后再去浏览 http://<任意节点的 ip>:30001。

https://www.veryopen.org