



## 大数据、云计算工程师、云原生开发人员、 互联网运维人员必读



精选Kubernetes的硬核知识，详解Kubernetes的实战技术  
随书附赠关键示例程序源代码、配置文件、数据文件，以及  
配套的《实践手册》电子书、系列高清课程视频



Linux、云原生、大数据教程系列

# Kubernetes 快速入门与实战 实践手册

艾叔 编著

2021.09



## 目录

前 言 .....	4
第 1 章 扩展阅读 .....	错误！未定义书签。
第 2 章 实践 .....	5
2.1 实践 1---定制 VMware 虚拟机 .....	5
2.2 实践 2---最小化安装 CentOS 8 .....	13
2.2.1 下载 CentOS 8 镜像文件 .....	13
2.2.2 关联 CentOS 8 镜像文件 .....	14
2.2.3 安装 CentOS 8 .....	14
2.2.4 配置 CentOS 8 .....	19
2.3 实践 3---远程登录和文件传输 .....	24
2.4 实践 4---远程无密码登录 .....	错误！未定义书签。
2.5 实践 5---Docker 的安装与使用 .....	错误！未定义书签。
2.5.1 安装 Docker .....	错误！未定义书签。
2.5.2 Docker 常用操作 .....	错误！未定义书签。
2.6 实践 6---RC/RS 基本操作 .....	24
2.7 实践 7---Deployment 典型使用 .....	32
2.8 实践 8---Service 典型使用 .....	36
2.9 实践 9---Pod 多容器运行实践 .....	41
2.10 实践 10---Kubernetes 中容器的高可用实践 .....	43

# 前言

**Kubernetes** 是由 Google 开源的一个容器编排（Orchestration）系统，它实现了集群中容器的管理、部署、迁移和扩展的自动化<sup>[1]</sup>（<https://kubernetes.io/docs/home/>）。自 2014 年开源以来，Kubernetes 经过多个版本的迭代和完善，已经广泛用于生产环境。Google、Microsoft、Amazon、阿里和腾讯等，都提供云上的 Kubernetes 服务，而阿里自身的核心应用更是全部运行在 Kubernetes 之上。Linux 基金会报告显示，2021 年云原生技术首次超过 Linux 自身，成为最热门开源技术，而 Kubernetes 作为云原生技术的代表，则更是热门中的热门。

因此，对于 IT 从业人员而言，**Kubernetes** 是一个重要的加分项和加薪项，Kubernetes 学的越早，掌握的越好，就越会成为自身的一个优势。然而从学习的角度而言，Kubernetes 却不是那么友好，Kubernetes 涉及的概念新、概念多，而且需要很多的前置知识，例如 Linux、网络、虚拟化、Docker 容器等；而且 **Kubernetes** 是面向整个集群的容器编排，在架构、运行机制和使用上更为复杂；再加上 Kubernetes 是一个底层基础设施，几乎所有的应用都需要进行迁移，这些都增加了 **Kubernetes** 的学习难度。


为此，笔者根据自身在 Kubernetes 上的研发和使用经验，编写了《Kubernetes/K8s 快速入门与实战（LVS+Prometheus+Grafana+GitLab+Jenkins）》一书，该书共分为 8 章，分别是：认识 Kubernetes、快速构建 Kubernetes 集群、Kubernetes 核心对象使用、Kubernetes 容器编排实践、Kubernetes 系统运维与故障处理、Kubernetes 高可用集群构建、Kubernetes 监控与告警（Prometheus+Grafana）、基于 Kubernetes 的 CI/CD 项目综合实践（GitLab+Harbor+Jenkins）。

而本书《Kubernetes/K8s 快速入门与实战---实践手册》就是《Kubernetes/K8s 快速入门与实战（LVS+Prometheus+Grafana+GitLab+Jenkins）》配套的免费电子书，本书既可以作为云原生及相关行业从业者的技术参考书，也可以作为高等学校计算机、云计算和大数据相关专业的教材。

感谢一直以来，关心帮助我成长的家人、老师、领导、同学和朋友们！

时间紧、任务急，书中有很多疏漏、甚至错误之处，如果您在阅读过程中有任何疑问，可以通过下面的方式，联系我们。

- 扫码  添加作者微信，加入学习交流群。

- 扫码关注本书公众号：艾叔编程 。

- 作者邮箱：spark\_aishu@126.com

艾叔  
2022.03 月

# 第 1 章 实践手册

## 1.1 实践 1---定制 VMware 虚拟机

所谓“定制 VMware 虚拟机”，就是利用虚拟机软件 VMware Workstation 来创建一台符合用户要求的虚拟机。这台虚拟机和真实的物理机一样，有 CPU、内存、硬盘和网卡等，其 CPU 的核数、内存的大小、硬盘的大小和网卡的个数可以根据用户的要求定制。

本书采用 VMware Workstation 15.5 Pro。

定制虚拟机的具体操作如下。

1) 点击菜单栏的“File”，在弹出的下拉菜单中点击“新建虚拟机”菜单项，如图 1-1 所示。



图 1-1 新建虚拟机菜单

2) 在弹出的虚拟机配置界面中，选择“自定义”，即定制虚拟机，然后点击“下一步”按钮，如图 1-2 所示。

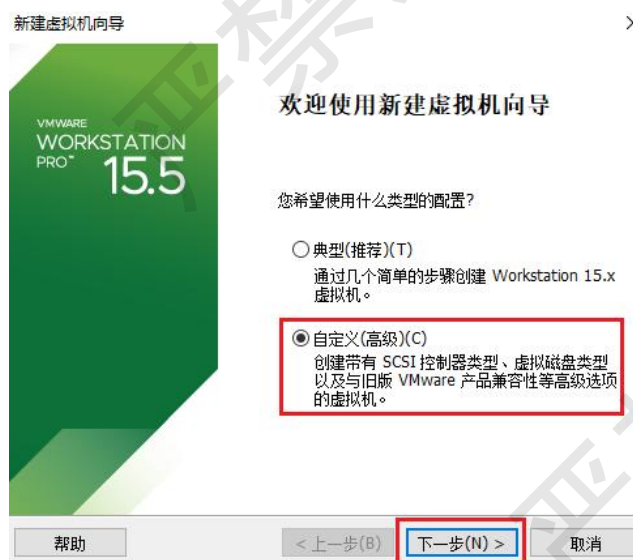


图 1-2 虚拟机配置界面

如果选择“典型”，则虚拟机的各项配置都是固定好的，无法修改；选择“自定义”，则可以根据需要，修改虚拟机硬件配置。

3) 在硬件兼容性下拉菜单中，选择“Workstation 15.x”，如图 1-3 所示。



图 1-3 虚拟机硬件兼容性配置界面

“硬件兼容性”决定虚拟机的两个方面：

1. 兼容性，即哪些 VMware 软件能打开并运行该虚拟机。如果选择“Workstation 15.x”，则将本例创建的虚拟机镜像复制到移动硬盘，插入另外一台安装了 VMware 软件的计算机，如图 1-3 所示，只有 Fusion 11.x 和 Workstation 15.x 能打开并运行移动硬盘中的虚拟机，其他的低版本软件如 Workstation 9.0 将无法使用此虚拟机。因此，“硬件兼容性”越高，能使用它的虚拟机软件就越少，本例的虚拟机“硬件兼容性”为“Workstation 15.x”，因此使用“VMware Workstation 15.x”打开是没有问题的；

2. 硬件限制，“硬件兼容性”越高，硬件限制越少，所创建的虚拟机的硬件性能越高。本例创建的虚拟机，内存的限制为 64GB，这就是说，如果物理机器的内存大于 64GB，例如 128GB，那么，创建的虚拟机内存最大可以到 64GB，但是，如果“硬件兼容性”为 6.5-7x，即使物理机内存有 128GB，所创建的虚拟机也只能支持 32GB 内存。

4) 在安装客户机（Guest）操作系统配置界面中，选择“稍后安装操作系统”，如图 1-4 所示。



图 1-4 Guest 操作系统安装配置界面

宿主机（Host）是指虚拟机所在的主机，本例中的 Host 就是物理机器，在 Host 上安装的系统，称为 Host 系统；客户机（Guest）则指虚拟机本身，在 Guest 上安装的系统，称为 Guest 系统；Host 只有一个，而 Guest 可以有多个；Host 和 Guest 是相对概念，例如，在虚拟机 A 上通过虚拟化软件又虚拟了一台机器 B，虽然 A 和 B 都是虚拟机，但 A 是 B 的 Host，A 是物理主机的 Guest。

5) 在选择客户机操作系统界面中，选择 Linux，在“版本”下拉菜单中选择“Red Hat Enterprise Linux 8 64 位”，如图 1-5 所示。



图 1-5 Guest 操作系统选择界面

6) 设置虚拟机的名字为“centos8”，虚拟机文件存储路径为“E:\vm\02\centos8”，如图 1-6 所示。



E 盘下面的 vm 及它下面的 02 文件夹和 centos8 文件夹，需要手动创建，VMware Workstation 不会自动创建。



图 1-6 Guest 操作系统选择界面

7) 设置虚拟机的处理器个数为 1，核数也为 2，如图 1-7 所示。



图 1-7 虚拟机处理器配置界面

8) 设置虚拟机内存大小为 4096MB，如图 1-8 所示。

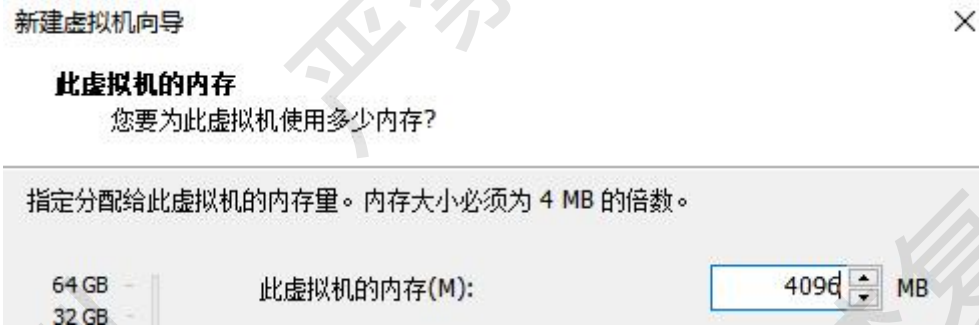


图 1-8 虚拟机内存配置界面

9) 设置 VMware 虚拟网络类型为“使用桥接网络”，如图 2-9 所示。



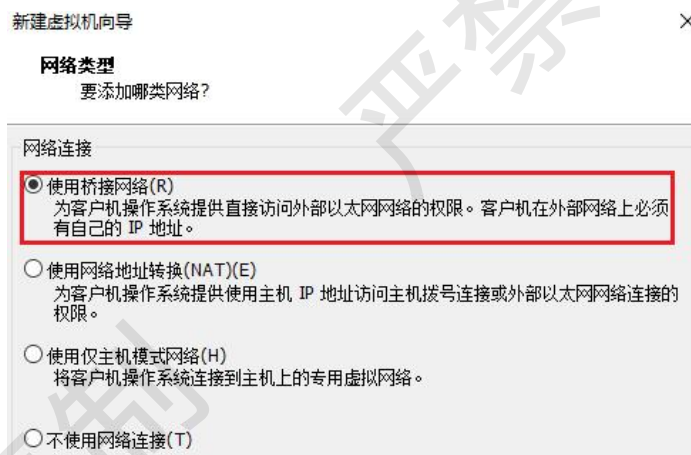


图 1-9 虚拟机网络配置界面

VMware 虚拟网络有 3 种类型，每种类型适应不同的应用场景，说明如下：

1. “桥接网络（Bridged networking）”中，Guest 拥有和 Host 对等的 IP，并且在同一物理网络上，如果 Guest 需要暴露自己的 IP 供物理网络上的其他机器访问，那么可以使用“桥接网络”；
2. “网络地址转换”（NAT）适用于 Guest 共享 Host 的 IP 上网的情形；
3. “仅主机”（Host-only networking）则适合 Guest 之间互联，不需要被 Host 所在物理网络其他机器访问的情形。

**注意：**3 种虚拟网络是虚拟机学习的重难点，此处不再赘述，本手册提供免费高清视频教程链接《[艾叔：零基础 VMware 虚拟机实战入门](#)》，对其原理和操作进行了详细说明，参见附录“学习资源获取方式”。

10) I/O 控制器类型为推荐的“LSI Logic(L)”，如图 1-10 所示。



图 1-10 I/O 控制器配置界面

11) 磁盘类型选择“NVMe(V)”，如图 1-11 所示。

## 新建虚拟机向导

### 选择磁盘类型

您要创建何种磁盘？

虚拟磁盘类型

☐ IDE(I)

☐ SCSI(S)

☐ SATA(A)

☒ NVMe(V) (推荐)

图 1-11 磁盘配置界面

NVMe 是一种接口协议，不是指的接口，NVMe 标准是面向 PCI-E 固态硬盘的，解除了旧标准施放在 SSD 上的各种限制。支持所有常见的操作系统、良好的可拓展性、具有低延迟，低能耗，高性能等优点、自动功耗状态切换和动态能耗管理功能大大降低功耗、解决了不同 PCI-E SSD 之间的驱动适用性问题。

12) 选择“创建新虚拟磁盘”，如图 1-12 所示。

### 选择磁盘

您要使用哪个磁盘？

磁盘

☒ 创建新虚拟磁盘(V)

虚拟磁盘由主机文件系统上的一个或多个文件组成，客户机操作系统会将其视为单个硬盘。虚拟磁盘可在一台主机上或多台主机之间轻松复制或移动。

☐ 使用现有虚拟磁盘(E)

选择此选项可重新使用以前配置的磁盘。

☐ 使用物理磁盘 (适用于高级用户)(P)

选择此选项可为虚拟机提供直接访问本地硬盘的权限。需要具有管理员特权。

图 1-12 磁盘选择界面

13) 磁盘容量设置为“100GB”，并选择将虚拟磁盘存储为单个文件，如图 1-13 所示。

## 新建虚拟机向导

### 指定磁盘容量

磁盘大小为多少？

最大磁盘大小 (GB)(S)

针对 Red Hat Enterprise Linux 8 64 位的建议大小: 20 GB

☐ 立即分配所有磁盘空间(A)。

分配所有容量可以提高性能，但要求所有物理磁盘空间立即可用。如果不立即分配所有空间，虚拟磁盘的空间最初很小，会随着您向其中添加数据而不断变大。

☒ 将虚拟磁盘存储为单个文件(O)

☐ 将虚拟磁盘拆分成多个文件(M)

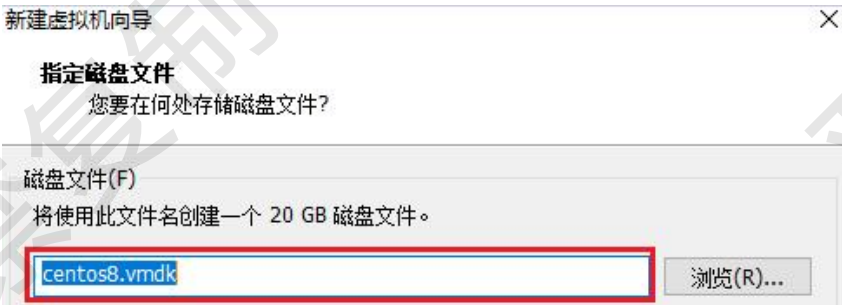
拆分磁盘后，可以更轻松地在计算机之间移动虚拟机，但可能会降低大容量磁盘的性能。

图 1-13 磁盘容量及存储方式配置界面

图 1-13 中“立即分配所有磁盘空间”这个不要勾选，否则，虚拟机会直接生产一个 100GB 大小的空文件，浪费空间；

“将虚拟磁盘存储为单个文件”会将虚拟磁盘存储成一个文件，这样性能相对会高一点。如果选择“将虚拟磁盘拆分为多个文件”则会将虚拟硬盘存储成多个文件，每个文件的大小不超过 2GB。

14) 设置虚拟硬盘文件名为“centos8.vmdk”，其中后缀 vmdk 是“virtual machine disk”的缩写，如图 1-14 所示。



1-14 硬盘容量及存储方式配置界面

15) 至此，虚拟机定制完毕，定制后的虚拟机信息如图 1-15 所示。



1-15 虚拟机参数界面

16) 点击 Finish 按钮，将出现定制的 centos8 虚拟机操作界面，如图 1-16 所示。



1-16 虚拟机 centos8 界面

接下来对 **centos8** 虚拟机进一步定制，去除一些不必要的硬件，操作步骤说明如下。

- 1) 点击菜单栏的“虚拟机”按钮，在下拉菜单中点击“设置”菜单项，如图 1-17 所示。



1-17 虚拟机设置菜单

- 2) 在弹出的对话框中点击“硬件”标签，在“硬件”选项卡中点击“声卡”选项，然后点击“移除”按钮，删除该硬件，如图 1-18 所示。



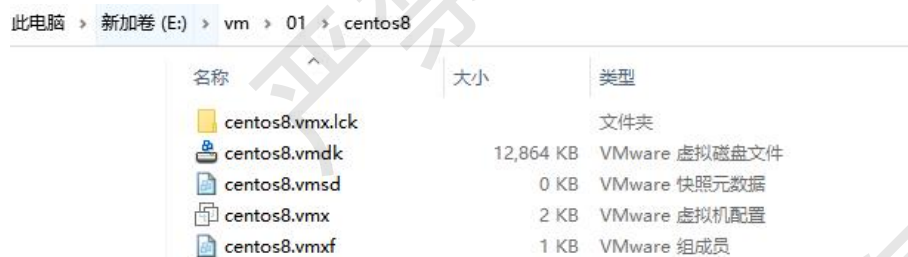
1-18 虚拟机硬件设置界面

3) 按照上面的步骤，删除“打印机”这个不需要的硬件设备，最后点击“确定”按钮结束定制。

虚拟机定制后的相关文件，如图 1-19 所示。重要的虚拟机文件有两个，说明如下：

- `centos8.vmdk` 文件是虚拟机硬盘文件，虚拟机硬盘的所有数据，包括硬盘自身的数据也都保存在这个文件中，目前这个文件只有 12MB 左右，随着往虚拟机硬盘写入数据，该文件会不断增大；
- `centos8.vmx` 是虚拟机配置文件，虚拟机名称，硬件配置等信息都在这个文件中。

虚拟机定制好后，要验证 `centos8.vmdk` 和 `centos8.vmx` 是否存在，如果要备份、迁移虚拟机，则只需要关闭虚拟机，复制或剪切这两个文件即可。



1-19 虚拟机文件

`centos8.vmx.lck` 是虚拟机被打开的锁文件，如果虚拟机在 VMware Workstation 面板中存在，就会有此文件（注意，不需要虚拟机运行，只要虚拟机被打开，出现在面板中，就会有此文件，当然，虚拟机运行时，肯定也会有此文件，因为，虚拟机运行之前肯定要被打开）。后续如果打开虚拟机时，提示该虚拟机 `is locked`，无法打开的话，可以尝试删除锁文件来解决问题。

## 1.2 实践 2---最小化安装 CentOS 8

本节将在 2.1 节所定制的 `centos8` 虚拟机上安装 Linux 发行版 **CentOS 8**，具体步骤如下。

### 1.2.1 下载 CentOS 8 镜像文件

CentOS 8 镜像文件的下载地址是：  
[https://mirrors.aliyun.com/centos-vault/8.2.2004/isos/x86\\_64/CentOS-8.2.2004-x86\\_64-dvd1.iso](https://mirrors.aliyun.com/centos-vault/8.2.2004/isos/x86_64/CentOS-8.2.2004-x86_64-dvd1.iso),



将该文件下载保存到 E:\vm 文件夹下，如图 2-1 所示。



2-1 CentOS 8 镜像文件

## 1.2.2 关联 CentOS 8 镜像文件

镜像文件下载后，在“硬件”选项卡中点击“CD/DVD（SATA）”项，选中虚拟机的光驱设备，如图 2-2 所示。



2-2 虚拟机 centos8 的光驱设备

在“连接”中选择“使用 ISO 映像文件”，然后点击“浏览”按钮，选择之前下载的镜像文件 **CentOS-8.2.2004-x86\_64-dvd1**，如图 2-3 所示，将镜像文件同虚拟机关联起来，相当于将光盘插入计算机。。



2-22 关联 CentOS 8 镜像文件

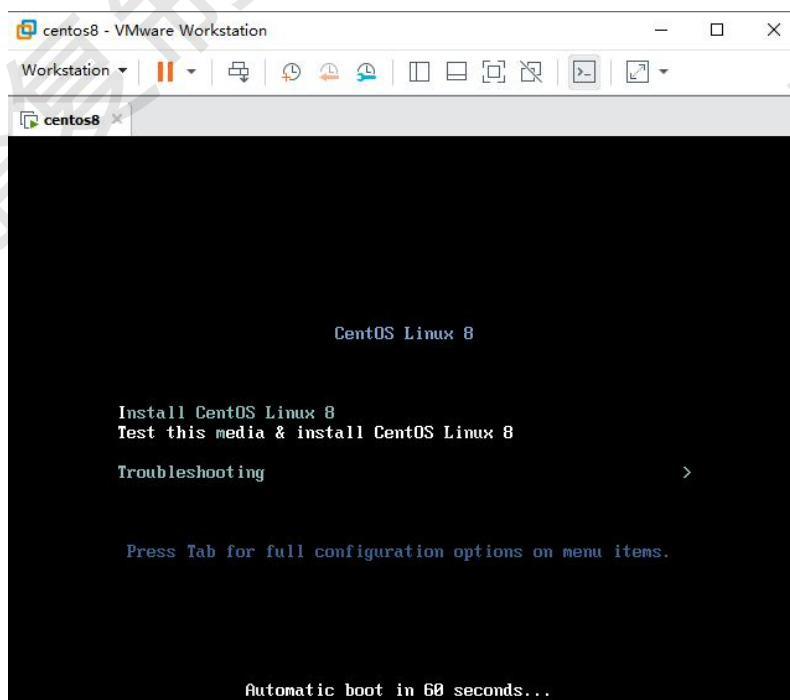
## 1.2.3 安装 CentOS 8

点击工具栏中的“开启”按钮，给虚拟机 **centos8** 上电，如图 2-4 所示。



2-4 虚拟机上电按钮

虚拟机上电后，会从光盘引导，显示界面如图 2-5 所示。



2-5 CentOS 8 安装界面

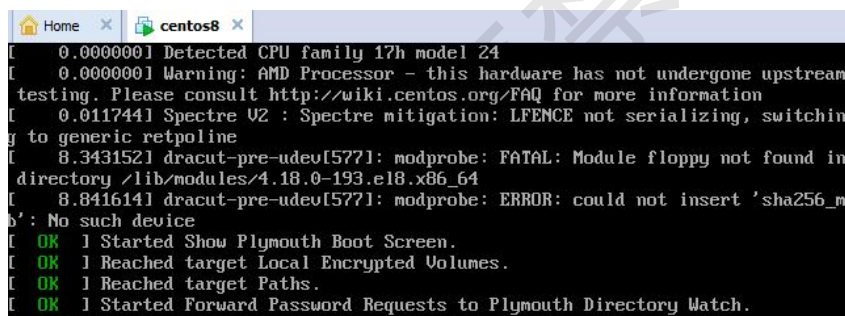
将鼠标移动到 **CentOS 8** 的安装界面并单击左键，鼠标焦点将进入虚拟机，然后按下向上的光标键，选中第一项“Install **CentOS Linux 8**”，如果该项被选中，将显示白色，如图 2-6 所示。



2-6 CentOS 8 安装选择界面

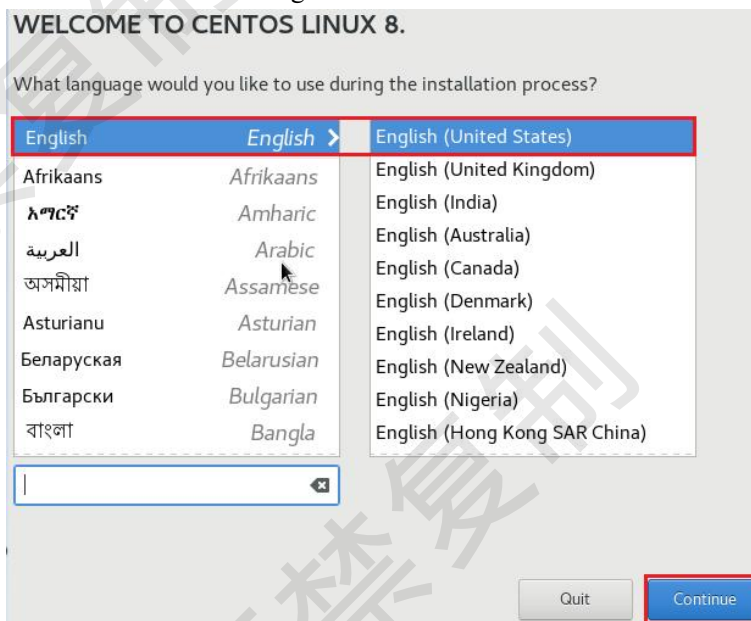
按下 Enter 键后，将开始安装，屏幕上会显示安装过程的提示信息，如图 2-7 所示。





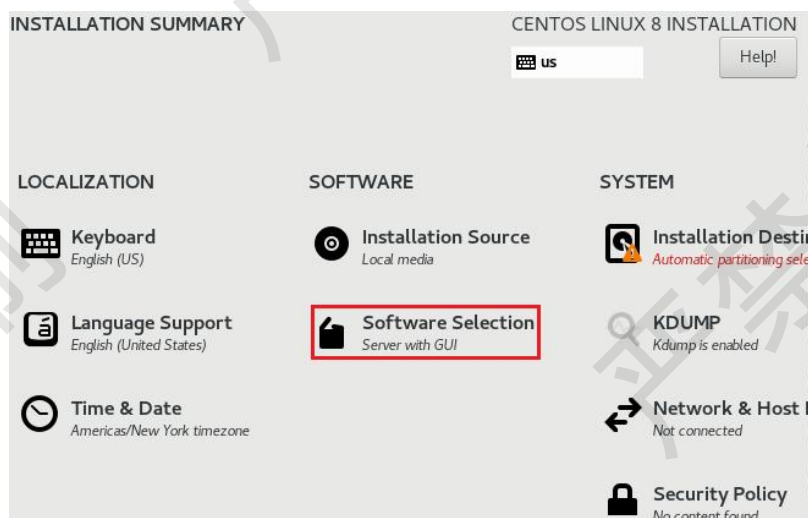
2-7 CentOS 8 安装过程信息

设置安装过程所采用的语言为 English，点击“Continue”按钮，如图 2-8 所示。



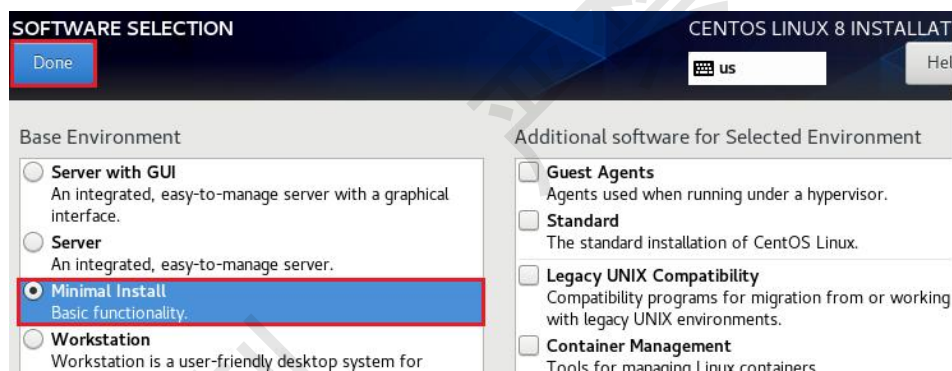
2-8 CentOS 8 安装过程语言选择界面

在安装配置界面中，点击“Software Selection”，如图 2-9 所示。



2-9 CentOS 8 安装配置界面

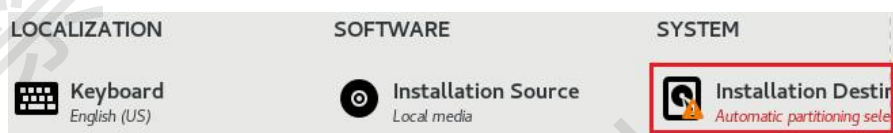
在“SOFTWARE SELECTION”界面中选择“Minimal Install”，然后点击“Done”按钮，如图 2-10 所示。



2-10 CentOS 8 软件配置界面

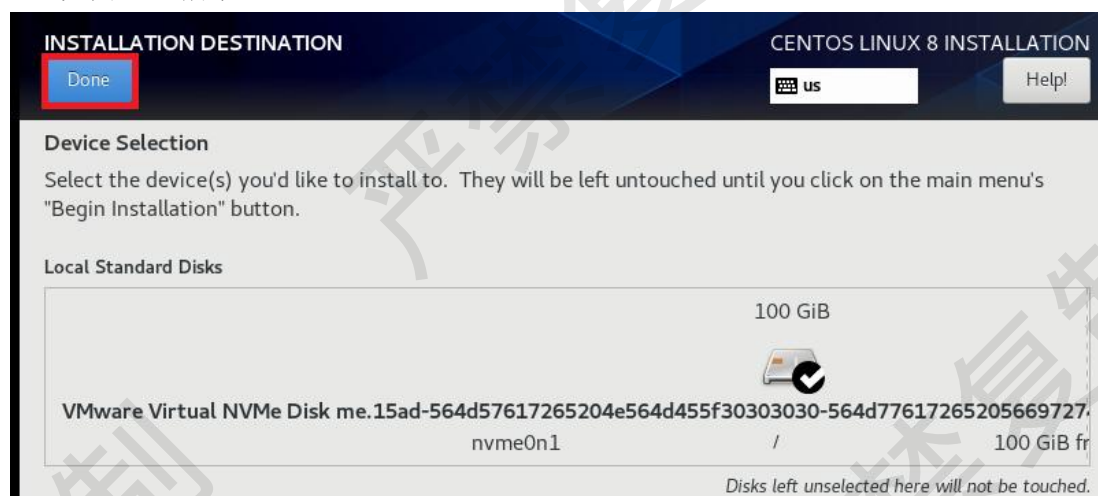
注意：此处安装的 **CentOS 8** 是 **Minimal** 安装，即最小化安装，只有字符界面和最基本的软件。

点击 “Installation Destination”，设置安装对象，如图 2-11 所示。



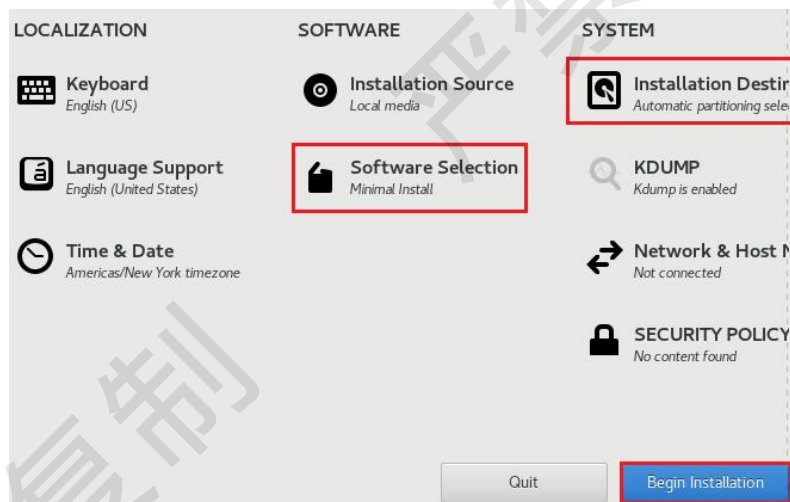
2-11 CentOS 8 安装配置界面

如果虚拟机有多个硬盘，需要选择将 **CentOS 8** 安装到哪个硬盘上。本书中虚拟机只有一个硬盘，所以只有一个安装对象，在安装对象上 **要打勾** 选中即可，然后点击 “Done” 按钮，如图 2-12 所示。



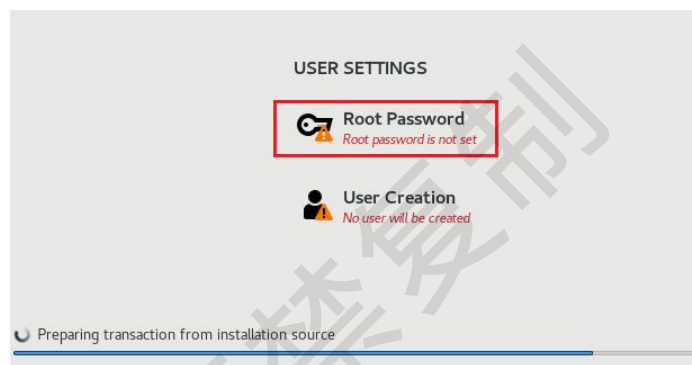
2-12 CentOS 8 安装对象界面

最终配置好的界面如图 2-13 所示，确认无误后，点击 “Begin Installation” 开始安装，如图 2-13 所示。



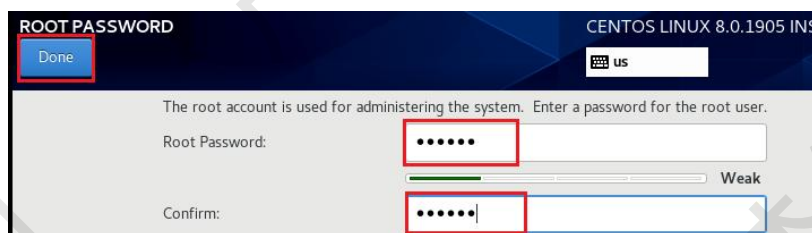
2-13 CentOS 8 安装配置界面

安装进度显示如图 2-14 所示，安装过程中还可以设置 root 密码，点击“Root Password”按钮。



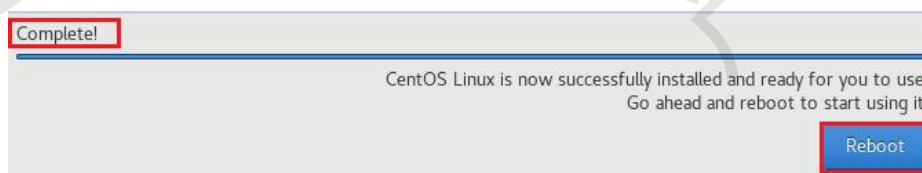
2-14 CentOS 8 安装过程界面

简单起见，此处设置密码为“123456”，点击“Done”按钮即可，如图 2-15 所示。



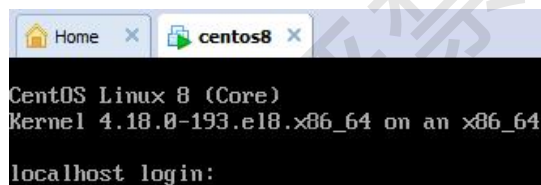
2-15 CentOS 8 安装过程界面

CentOS 8 安装完成后会显示“Complete!”，点击“Reboot”按钮结束安装，重启系统，如图 2-16 所示。



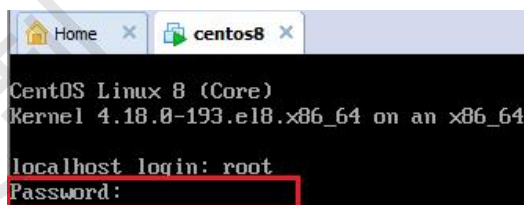
2-16 CentOS 8 安装界面

CentOS 8 启动成功后，显示登陆界面如图 2-17 所示。



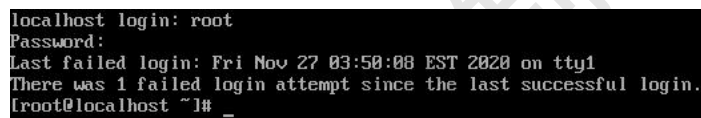
2-17 CentOS 8 登陆界面

登录，在“localhost login:”后输入 root，按下 Enter 键，则会出现“Password”提示输入密码，如图 2-18 所示。



2-18 CentOS 8 密码输入界面

在“Password:”后输入密码“123456”，注意，此时屏幕不会显示用户输入的密码信息，甚至连表示密码的圆点都不会有，这个在 Linux 中称为“没有回显”，我们不用担心，尽管输入，完成后按下 Enter 键即可。如果密码输入正确，则可以成功登陆，如图 2-19 所示，看到登陆提示符“[root@localhost ~]#”，其中#表示当前登录的用户为 root 用户，有关登录提示符后续还会有详细说明。



2-19 CentOS 8 登录成功界面

至此，CentOS 8 的安装和验证都已完成。

## 1.2.4 配置 CentOS 8

### 1. 设置 IP 地址

编辑 IP 地址配置文件，命令如下。

#### (1) 修改网卡配置文件

IP 地址的路径如图 3-6 所示，配置文件名为 ifcfg-ens160，配置文件名和网卡名是相关的，如果网卡名为 A，则配置文件名为 ifcfg-A。

```
[root@localhost ~]# vi /etc/sysconfig/network-scripts/ifcfg-ens160
```

3-6 网卡配置文件路径

ifcfg-ens160 文件内容如图 3-7 所示，其中修改/增加的内容为白色方框部分，BOOTPROTO=static 表示配置静态 IP 地址，如果改成 dhcp 则是配置动态 IP；ONBOOT=yes 表示网卡随系统一同启动；IPADDR=192.168.0.226 是网卡的 IP；NETMASK=255.255.255.0 即网卡 IP 的子网掩码，此处为 24 位的子网掩码。

```

TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=static
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6_INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=ens160
UUID=2173d2fb-e739-4e1a-8f20-84232fcf7ddc
DEVICE=ens160
ONBOOT=yes
IPADDR=192.168.0.226
NETMASK=255.255.255.0

```

3-7 网卡配置文件内容

(2) 使得配置文件生效

1) 运行“**nmcli c reload**”来加载修改后的 **ifcfg-ens33** 文件，如图 3-8 所示。

```
[root@localhost ~]# nmcli c reload
```

3-8 网卡配置文件内容

3) 查看网卡 **ens160** 的内容，可以看到该网卡的 IP 已经设置成了 192.168.0.226，如图 3-9 所示。

```

[root@localhost ~]# 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:c1:16:b0 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.226/24 brd 192.168.0.255 scope global noprefixroute ens160

```

3-9 网卡信息

如果上述 IP 地址设置不成功，请依次检查以下三点。

- 1) **ifcfg-ens160** 中的配置是否正确；
- 2) 使用“**systemctl status NetworkManager**”查看 **NetworkManager** 服务是否启动，如果看到 **running**，则说明服务已经启动，否则使用“**systemctl start NetworkManager**”启动 **NetworkManager** 服务，供 **nmcli** 使用；
- 3) 再次运行“**nmcli c reload**”，如果 IP 地址还未设置上，运行“**nmcli c up ens160**”使能 **ens160** 网卡，并再次查看 IP 地址。

## 2. 设置 centos8 连接互联网

虚拟机连接互联网有两项关键配置：配置网关，它将决定虚拟机的 Package 转发给谁；配置 DNS，它实现了域名同 IP 地址的转换。配置网关和 DNS 的具体步骤如下。

### (1) 配置网关和 DNS

编辑 **ifcfg-ens160** 文件，增加图 3-10 白框所示内容，其中“**GATEWAY=192.168.0.1**”用来配置网关地址为 192.168.0.1；“**DNS1=192.168.0.1**”用来配置第一个 DNS 的地址为 192.168.0.1，DNS 可以配置多个，在 DNS 后面增加序号即可，例如 **DNS2**、**DNS3** 等等。此处的网关和 DNS 均需要替换成读者网络的实际 IP 地址；



```

TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=static
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=ens160
UUID=2173d2fb-e739-4e1a-8f20-84232fcf7ddc
DEVICE=ens160
ONBOOT=yes
IPADDR=192.168.0.226
NETMASK=255.255.255.0
GATEWAY=192.168.0.1
DNS1=192.168.0.1

```

3-10 上网配置项

注配置文件修改后，一定要先运行“**nmcli c reload**”加载配置文件，再运行“**nmcli c up ens160**”使能网卡，使得配置生效。

## (2) 检查配置

运行“**ip route show**”查看网关是否设置正确，如果能看到图 3-11 白色方框的内容，则说明网关设置正确。

```

[root@localhost ~]# ip route show
default via 192.168.0.1 dev ens160 proto static metric 100
192.168.0.0/24 dev ens160 proto kernel scope link src 192.168.0.226 metric 100

```

3-11 路由信息图

## (3) 验证

使用“**ping www.baidu.com**”看外网能否 ping 通，如图 3-12 所示，如果可以，则说明 Linux 下的上网配置是正确的。

```

[root@localhost ~]# ping www.baidu.com
PING www.a.shifen.com (14.215.177.39) 56(84) bytes of data:
64 bytes from 14.215.177.39 (14.215.177.39): icmp_seq=1 ttl=54 time=21.5 ms
64 bytes from 14.215.177.39 (14.215.177.39): icmp_seq=2 ttl=54 time=21.4 ms

```

3-12 ping 外网信息图

如果虚拟机不能上外网，除了检查上网设置外，还要检查：Guest 能否 ping 通网关、Host 能否上外网，路由器中的防火墙是否放开了对虚拟机的限制等。

## 3. 关闭防火墙

关闭当前运行的防火墙，命令如下。

```
[root@localhost ~]# systemctl stop firewalld
```

禁止防火墙自启动，命令如下。

```
[root@localhost ~]# systemctl disable firewalld
```

## 4. 创建普通用户

创建普通用户 user 及其 home 目录，命令如下。

```
[root@localhost ~]# useradd -m user
```

```
[root@localhost ~]# ls /home/
```

user

设置 user 用户的密码为 user，命令如下。

```
[root@localhost ~]# passwd user
```

Changing password for user user.

New password:

BAD PASSWORD: The password is shorter than 8 characters

Retype new password:

passwd: all authentication tokens updated successfully.

## 5. 添加光盘安装源

### (1) 修改装源配置文件

编辑命令如下。

```
[root@localhost yum.repos.d]# vi /etc/yum.repos.d/CentOS-Media.repo
```

按以下步骤修改修改 CentOS-Media.repo。

1) 第 16 行 **baseurl** 修改成 “file:///media/**BaseOS**”，表示 yum 仓库位于本地目录 /media/**BaseOS** 下；

2) 第 18 行修改成 “enabled=1”，表示该配置项（c8-media-**BaseOS**）生效；

3) 第 23 行 **baseurl** 修改成 “file:///media/**AppStream**”，表示 yum 仓库位于本地目录 /media/**AppStream** 下；

4) 第 25 行修改成 “enabled=1”，表示该配置项（c8-media-**AppStream**）生效。

修改后的 CentOS-Media.repo 内容如下。

```
14 [c8-media-BaseOS]
```

```
15 name=CentOS-BaseOS-$releasever - Media
```

```
16 baseurl=file:///media/BaseOS
```

```
17 gpgcheck=1
```

```
18 enabled=1
```

```
19 gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial
```

```
20
```

```
21 [c8-media-AppStream]
```

```
22 name=CentOS-AppStream-$releasever - Media
```

```
23 baseurl=file:///media/AppStream
```

```
24 gpgcheck=1
```

```
25 enabled=1
```

```
26 gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-centosofficial
```

### (2) 挂载光盘

1) 挂载光盘到/media 目录，命令如下。

```
[root@localhost ~]# mount /dev/sr0 /media/
```

mount: /media: /dev/sr0 already mounted on /media.

2) 查看/media 目录，如果能看到下面的内容，则说明挂载成功。光盘目录中有两个子目录 **BaseOS** 和 **AppStream**，分别对应光盘上两个 yum 仓库，这样 yum 就可以使用这两个仓库来安装软件了。

```
[root@localhost ~]# ls /media/
```

**AppStreamBaseOS** EFI images **isolinux**media.repo TRANS.TBL

## 3. 更换下载速度更快安装源



CentOS 8 默认的网络安装源都在 CentOS 官网, 有的时候下载速度很慢, 可以将这些安装源修改成国内的安装源, 这样可以大大加快软件的安装速度, 以添加阿里云的安装源为例, 具体说明如下。

#### (1) 修改 CentOS-AppStream.repo

编辑配置文件命令如下。

```
[root@localhost yum.repos.d]# vi /etc/yum.repos.d/CentOS-AppStream.repo
```

在 CentOS-AppStream.repo 中 name 的值后面加上 Ali 标识, 注释掉 mirrorlist 和原来的 baseurl, 添加新 baseurl (阿里云的仓库路径), 如下所示。

```
13 [AppStream]
14 name=CentOS-$releasever - AppStream - Ali
15
16 #mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=$basearch&repo=AppStream&infra=$infra
17 #baseurl=http://mirror.centos.org/$contentdir/$releasever/AppStream/$basearch/os/
18 baseurl=https://mirrors.aliyun.com/centos/$releasever-stream/AppStream/$basearch/os/
```

baseurl 中的字符串输入, 要一个一个字符地去核对。

#### (2) 修改 CentOS-Base.repo

同样的原理, 修改 CentOS-Base.repo 的配置如下。

```
13 [BaseOS]
14 name=CentOS-$releasever - Base - Ali
15
16 #mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=$basearch&repo=BaseOS&infra=$infra
17 #baseurl=http://mirror.centos.org/$contentdir/$releasever/BaseOS/$basearch/os/
18 baseurl=https://mirrors.aliyun.com/centos/$releasever-stream/BaseOS/$basearch/os/
```

#### (3) 修改 CentOS-Extras.repo

同样的原理, 修改 CentOS-Extras.repo 的配置如下。

```
14 [extras]
15 name=CentOS-$releasever - Extras - Ali
16
17 #mirrorlist=http://mirrorlist.centos.org/?release=$releasever&arch=$basearch&repo=extras&infra=$infra
18 #baseurl=http://mirror.centos.org/$contentdir/$releasever/extras/$basearch/os/
19 baseurl=https://mirrors.aliyun.com/centos/$releasever-stream/extras/$basearch/os/
```

#### (4) 重新缓存 yum 元数据

执行下面的命令, 清空 yum 元数据, 重新缓存 yum 元数据。

```
[root@localhost yum.repos.d]# yum clean all
```

```
[root@localhost yum.repos.d]# yum makecache
```

如果能看到下面的输出, 则说明 yum 安装源配置成功。

```
CentOS-8 - AppStream - Ali      23 kB/s | 4.3 kB      00:00
CentOS-8 - Base                 38 kB/s | 3.9 kB      00:00
CentOS-8 - Extras - Ali         9.9 kB/s | 1.5 kB      00:00
CentOS-BaseOS-8 - Media         82 MB/s | 2.2 MB      00:00
CentOS-AppStream-8 - Media      59 MB/s | 5.7 MB      00:00
```

Metadata cache created.

如果配置不成功，则要首先检查能否连接互联网；其次要重点检查安装源配置文件中的 `baseurl` 配置；最后检查 `baseurl` 能否访问。

## 1.3 实践 3---远程登录和文件传输

本小节介绍远程登录工具 `ssh` 和 `PuTTY`，以及文件传输工具 `scp` 和 `WinSCP` 的使用。其中 `ssh` 和 `scp` 运行在 Linux 下，`PuTTY` 和 `WinSCP` 运行在 Windows 下，它们都是 Linux 系统操作和运维的常用工具。

### 1. Linux 下的远程登录工具 `ssh`

`ssh` 命令是 Linux 下的常用远程登录工具，`ssh` 命令又称 `ssh` 客户端，它使用 `SSH` 协议同 `SSH` 服务器 `sshd` 交互，从而实现从本机（Linux 主机）远程登录到其他 Linux 主机进行操作的功能。因此，在远程登录前，要求被登录的 Linux 主机先启动 `sshd` 服务。

CentOS 8 下可以使用 “`systemctl status sshd`” 查看 `SSH` 服务的状态，如果能看到 “active (running)” 则说明 `sshd` 服务正常工作，否则就要切换到 `root` 用户，运行 “`systemctl start sshd`” 来启动 `sshd` 服务。

`ssh` 命令的使用方法如下，其中 `USER` 为登录的用户身份，也可以不指定登录用户，即 “`ssh IP`”，那么 `ssh` 命令就会以当前用户的身份进行登录；`IP` 是远程主机的 `IP` 地址，也可以使用主机名替代。

```
[user@localhost ~]$ ssh USER@IP
```

`ssh` 命令的使用示例如下，它将以 `root` 用户身份远程登录 `192.168.0.226`。

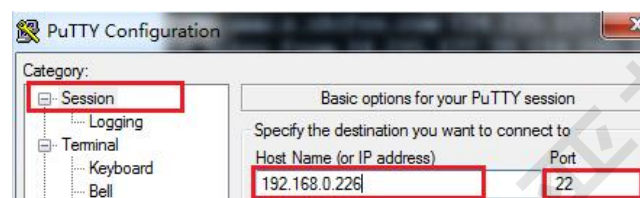
```
[user@localhost ~]$ ssh root@192.168.0.226
```

按下 `Enter` 键后，Linux 会提示输入密码，如果密码正确，就会以 `root` 身份登录到 `192.168.0.226`，此后的操作就和在 `192.168.0.226` 本机上操作完全一样了，如下所示。

```
[root@localhost ~]#
```

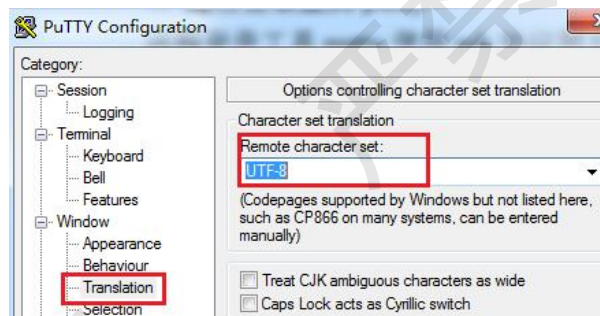
### 2. Windows 下的远程登录工具 `PuTTY`

远程登录工具 `PuTTY` 是一个运行在 Windows 下的终端模拟器，其工作原理请参考 1.2.4 节内容。它使用 `SSH` 协议同 Linux 通信，因此，要求 Linux 上先启动 `SSH` 服务。运行的 `PuTTY` 界面如图 2-39 所示，需要在 “Host Name” 文本框中填入 Linux 主机的 `IP` 或者主机名，例如 `192.168.0.226`，`Port` 文本框中填入 `22`，这是 `SSH` 服务的监听端口。



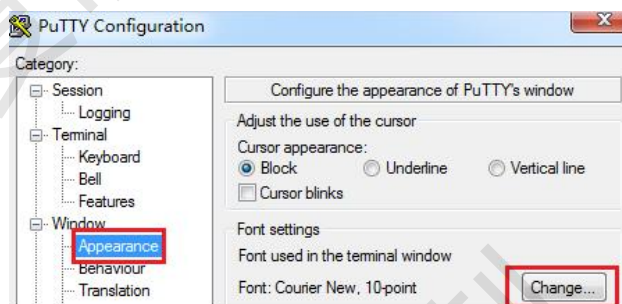
2-39 `ping` 外网信息图

在 `Translation` 设置中，将 “Remote character set” 字符集设置成 `UTF-8`，如图 2-40 所示，如果不设置的话，中文将显示乱码。



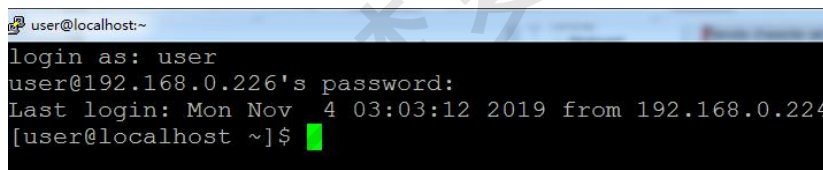
2-40 字符集设置界面

此外，PuTTY 字体大小也是常用设置项，如图 2-41 所示，点击“Change”按钮，在弹出的对话框中选择更大的字体即可。



2-41 字体大小设置界面

上述配置完成后，点击“Open”按钮，就可以看到 Linux 的登录界面，输入用户名和密码，就可以看到登录提示符，如图 2-42 所示。



2-42 PuTTY 操作界面

使用 PuTTY 有很多好处，说明如下。

- 和虚拟终端相比，PuTTY 操作更为方便。可以同时运行多个 PuTTY 连接到 Linux，这样可以同时有多个 Linux 的操作界面，而虚拟终端每次只能显示一个界面，不同界面之间还要用组合键进行切换；
- 使用 PuTTY，Linux 主机可以不需要外接显示器，从而节约成本；
- 使用 PuTTY 可以登录到千里之外的 Linux 服务器，就如同在本地操作一样，而且字符信息占用带宽极少，即使在恶劣的网络环境下也能顺利操作。

在 PuTTY 中，用鼠标选中内容后，就会自动进行 Copy 操作，按下右键，就会执行粘贴操作，这是 PuTTY 中一个非常有用的操作。

### 3. Linux 下的文件传输工具 scp

scp 命令是 Linux 下的常用文件传输工具，它也是基于 SSH 协议实现的安全传输工具，因此，在接收文件的 Linux 主机要先启动 SSH 服务。scp 的典型用例说明如下。

#### (1) 将文件从 Linux 主机 A 传输到主机 B

本示例将 Linux 主机 A 下/etc/profile 文件以 root 用户的身份传输到主机 B(192.168.0.226) 的/tmp 目录下，命令如下。

```
[user@localhost ~]$ scp /etc/profile root@192.168.0.226:/tmp
```

上述命令及参数说明如下。

- **scp** 是远程复制命令；
- **/etc/profile** 是源文件路径；
- **root** 是远程登录的用户身份，即以 **root** 用户的身份登录 **192.168.0.226**，并以该身份将 **profile** 文件复制到 **/tmp**，因此、复制后的 **profile** 文件的 Owner 是 **root**。此处也可以不指定用户身份，直接指定 IP，例如“**scp /etc/profile 192.168.0.226:/tmp**”，则会以当前用户（**user**）的身份远程登录和复制；
- **@** 是地址符号，后面跟主机 **B** 的地址或主机名；
- **192.168.0.226** 是主机 **B** 的地址，也可以用主机名替代；
- **:** 是分隔符，后面跟文件的目的路径，本例中 **profile** 会复制到主机 **B** 的 **/tmp** 目录下。

按下 **Enter** 键后，会提示输入主机 **B** 的 **root** 密码，验证通过后，**profile** 文件会从主机 **A** 复制到主机 **B** 的 **/tmp** 目录，验证命令如下。

```
[user@localhost ~]$ ls -l /tmp/profile
```

```
-rw-r--r--. 1 root root 2078 Nov  4 19:36 /tmp/profile
```

#### (2) 将目录从 Linux 主机 A 传输到主机 B

远程目录的复制命令同远程文件的复制一样，仅多了一个参数 **-r**，示例如下，它将主机 **A** 上的 **/home/user** 目录以 **root** 用户的身份，复制到主机 **B** 的 **/tmp** 目录，命令如下。

```
[user@localhost ~]$ scp -r /home/user/ root@192.168.0.226:/tmp/
```

按下 **Enter** 键后，会提示输入主机 **B** 的 **root** 密码，验证通过后，**user** 目录会从主机 **A** 复制到主机 **B** 的 **/tmp** 目录，验证如下。

```
[user@localhost ~]$ ls -l /tmp/
```

```
drwx-----. 5 root root 244 Nov  4 19:45 user
```

#### 4. 文件传输工具 WinSCP

**WinSCP** 使用 **SCP** 实现 **Windows** 同 **Linux** 之间的文件传输，**SCP** 是“Secure Copy”的缩写，它也是基于 **SSH** 协议实现的，因此，在 **Linux** 上需要先启动 **SSH** 服务。

**WinSCP** 运行后界面如图 2-43 所示，使用步骤说明如下。

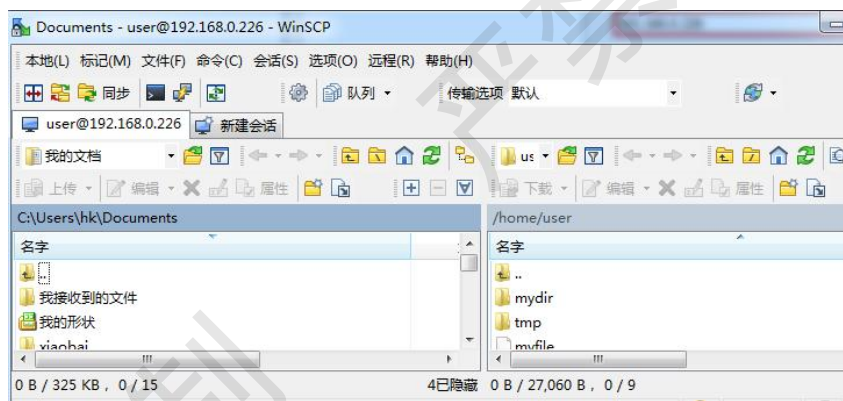
1) 先在“文件协议”下拉菜单中选择 **SCP**，然后在“主机名”文本框中填入 **192.168.0.226**，即 **Linux** 主机的 IP，接下来输入用户名 **user** 和密码；



2-43 WinSCP 配置界面

2) 点击“登录”按钮后，**WinSCP** 会根据刚才填入的配置信息，远程登录 **192.168.0.226**，并显示 **Windows** 和 **Linux** 双方的目录结构，如图 2-44 所示，左侧显示的是 **Windows** 的目录结构，右侧显示的是 **Linux** 的目录结构；

3) 使用工具栏中的按钮，遍历到不同的路径，然后使用拖拽操作，实现 **Windows** 和 **Linux** 之间的文件传输。



2-44 WinSCP 操作界面

## 1.4 实践 4---RC/RS 基本操作

RC/RS 可以使得一个 Pod 或者一组 Pod 按照用户所配置的副本数来运行。例如在 RC/RS 中配置 Pod 的副本数为 3，当集群中该 Pod 的副本数小于 3 时，RC/RS 会启动新的 Pod，使得该 Pod 的副本数等于 3；而当集群中该 Pod 的副本数大于 3 时，RC/RS 会删除掉多余的 Pod，使得 Pod 的副本数等于 3。下面按照先 RC，再 RS 的顺序，来介绍它们的使用。

### 1. RC 操作

使用 YAML 来创建 RC，首先编辑 RC 文件，命令如下。

```
[user@master k8s]$ mkdir rc
[user@master k8s]$ cd rc
[user@master rc]$ vi rc.yml
```

#### (1) 编辑创建 RC 的 YAML 文件

将 RC 的 Object 内容按照 YAML 格式，写入文件 rc.yml，具体内容如下。

```
1 apiVersion: v1
2 kind: ReplicationController
3 metadata:
4   name: myrc
5 spec:
6   replicas: 2
7   selector:
8     app: mynginx-pod
9   template:
10     metadata:
11       labels:
12         app: mynginx-pod
13     spec:
14       containers:
15         - name: nginx-container
16           image: nginx
17           imagePullPolicy: IfNotPresent
```

配置文件说明如下。

- 1) 第 1 行为 Object 的第 1 个成员 **apiVersion**，值是 v1；
- 2) 第 2 行是第二个成员 **kind**，值是 ReplicationController；



3) 第 3~4 行是第三个成员 `metadata` 是一个 Object，它有 1 个成员 `name`，其值是 `myrc`，`myrc` 不能和 `Kubernetes` 中其他 RC 同名；

4) 第 5~17 行是第 4 个成员 `spec` 是一个 Object，它包括 3 个成员；

5) 第 6 行是 `spec` 的第一个成员 `replicas`，用来设置副本数，值是 2；

6) 第 7~8 行是 `spec` 的第二个成员 `selector`，用于匹配符合条件的 Pod，该 `selector` 是 equality-based（基于相等）的，`selector` 中的每一个成员就是一个表达式，`selector` 会使用这些表达式，对 Pod 的 labels（标签）进行相等关系（等于/不等于）运算，如果所有表达式都条件匹配，则将此 Pod 作为为 RC 的管理对象。本例中 `selector` 有一个成员 `app`，其值是 `mynginx-pod`，这就是一个 `app=mynginx-pod` 的表达式，如果 Pod 的标签（Labels）中设置了 `app: mynginx-pod`，则该 Pod 条件匹配，如果该 Pod 没有被其他的 RC/RS 所控制，那么 RC 就会控制该 Pod，并不用管这个 Pod 是不是在该 RC 文件中创建的。

任何一个 Pod 只会被一个 RC/RS 所管理，因此，当 RC 使用 `selector` 匹配时，如果一个 Pod 的 labels 条件匹配，但是该 Pod 在另一个 RC/RS（名字不同）中，那么，RC 是不会将该 Pod 作为管理对象的，这样可以有效避免多个 RC/RS 管理同一个 Pod 所带来的冲突。

7) 第 9~12 行是 `spec` 的第三个成员 `template`，用来描述 Pod 的信息，这部分内容和单纯创建 Pod 时写入 YAML 的内容基本相同，只是没有 `apiVersion` 和 `kind`。要注意的是，此处在 `metadata` 中，要设置一个 labels 的 Object 成员，并在 labels 中增加成员 `app`，值 `mynginx-pod`，用于前面 RC 的 `selector` 匹配。

RC 的 `selector` 不仅可以匹配 RC YAML 文件中所创建的 Pod，也可以匹配 RC YAML 文件外所创建的 Pod，因此，RC 控制和管理的是一组 Pod。

8) 第 14~17 行是 Pod 中的容器信息，其中 `name` 是容器名字，`image` 是容器镜像名称，新增的 `imagePullPolicy` 是容器镜像拉取策略，指定“`imagePullPolicy: IfNotPresent`”表示仅当镜像在本地不存在时才被拉取，也就是说如果 Pod 要 pull 的镜像在本地存在，就会使用本地镜像，这样可以有效避免网络情况不好时 pull 镜像导致的 pull 失败或者长时间的等待。

特别注意：name、image、imagePullPolicy 首字母要对齐。

## (2) 使用命令创建 RC

YAML 编辑好后，我们就可以使用下面的命令来创建 RC。

```
[user@master rc]$ kubectl apply -f rc.yml
```

如果系统输出以下内容，则说明创建成功。

```
replicationcontroller/myrc created
```

## (3) 查看 RC

使用下面的命令，来查看 RC 的信息。

```
[user@master rc]$ kubectl get rc -o wide
```

RC 的信息显示如图 2-47 所示，各列根据其名字和具体的值就可以很清楚地看出其中的含义，在此不再赘述。

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS	IMAGES	SELECTOR
myrc	2	2	2	2m22s	nginx-container	nginx	app=mynginx-pod

2-47 RC 信息图

## 2. RS 操作

使用 YAML 来创建 RS，首先编辑 RS 文件，命令如下。

```
[user@master rc]$ vi rs.yml
```

### (1) 编辑创建 RS 的 YAML 文件

将 RS 的 Object 内容按照 YAML 格式，写入文件 `rs.yml`，具体内容如下。

```
1 apiVersion: apps/v1
2 kind: ReplicaSet
3 metadata:
4   name: myrs
5 spec:
6   replicas: 1
7   selector:
8     matchLabels:
9       app: mynginx-pod
10    matchExpressions:
11      - {key: ver, operator: In, values: [alpha, beta, stable]}
12   template:
13     metadata:
14       labels:
15         app: mynginx-pod
16         ver: beta
17     spec:
18       containers:
19         - name: nginx-container
20           image: nginx
21           imagePullPolicy: IfNotPresent
```

上述配置文件说明如下。

- 1) 第 1 行为 Object 的第 1 个成员 `apiVersion`，值是 `apps/v1`；
- 2) 第 2 行是第二个成员 `kind`，值是 `ReplicaSet`；
- 3) 第 3~4 行是第三个成员 `metadata` 是一个 Object，它有 1 个成员 `name`，其值是 `myrs`，`myrs` 不能和 `Kubernetes` 中的其他 RS 同名；
- 4) 第 5~21 行是第 4 个成员 `spec` 是一个 Object，它包括 3 个成员；
- 5) 第 6 行是 `spec` 的第一个成员 `replicas`，用来设置副本数，值是 1；
- 6) 第 7~11 行是 `spec` 的第二个成员 `selector`，用于匹配符合条件的 Pod，`selector` 提供两类匹配方法，即 `matchLabels` 和 `matchExpressions`。（创建后，不可修改）

其中 `matchLabels` 的使用是基于 equality-based（基于相等）的，其使用方法和 RC 中的 `selector` 是一样的，本例中 `matchLabels` 有一个成员 `app`，其值是 `mynginx-pod`，这就是一个 `app=mynginx-pod` 的表达式，如果 Pod 的 labels 中也设置了 `app: mynginx-pod-rs`，则条件匹配，否则不匹配。如果 `matchLabels` 中所有的表达式（成员）都匹配，则可以说 `matchLabels` 的条件匹配；

`matchExpressions` 的使用是 set-based（基于集合）的。`matchExpressions` 是一个数组，数组中的每个元素就是一个集合运算表达式，`matchExpressions` 会使用该表达式，去和 Pod 的 labels 中的每个值进行集合运算，如果每个表达式都条件匹配，则说明 `matchExpressions` 条件匹配。本例中，`matchExpressions` 只有 1 个元素：`- {key: ver, operator: In, values: [alpha, beta, stable]}`，`matchExpressions` 会查找 Pod 的 labels 是否有 Key 为 `ver` 的成员，如果没有，则条件不匹配；如果有，则看 `ver` 的值中是否是 `alpha, beta, stable` 中的任意一个，如果不是，则条件不匹配，如果是，则该元素的条件匹配。使用 `In` 操作，可以实现 Value 值的或匹配，



相对于 `matchLabels` 功能更强大，更灵活。本例中 Pod 的 labels 中设置了 “ver: beta” 条件匹配，又因为 `matchExpressions` 只有 1 个元素，因此，`matchExpressions` 的条件是匹配的。

`matchExpressions` 中的集合运算符包括：In、NotIn、Exists 和 DoesNotExist。使用 In 和 NotIn 时，values 不能为空。

RS 和 RC 最大的不同，就在于 selector，RS 增加了基于集合的条件匹配，更加灵活。

如果该 Pod 的 `matchLabels` 和 `matchExpressions` 都条件匹配，并且该 Pod 没有被其他的 RC/RS 所控制，那么 RS 就会控制该 Pod，并不用管这个 Pod 是不是在该 RS 文件中创建的。

7) 第 12~21 行是 spec 的第三个成员 template，用来描述 Pod 的信息，这部分内容和我们单纯创建 Pod 时写入 YAML 的内容基本相同，只是没有 `apiVersion` 和 `kind`。要注意的是，此处处在 metadata 中要设置一个 labels 的 Object 成员，并在 labels 中增加成员 app，值 mynginx-pod，用于 selector 的 `matchLabels` 匹配，增加成员 ver，值 beta，用于 selector 的 `matchExpressions` 匹配，如果 selector 匹配不上，则该 RS 创建会报错（创建后，此处内容在 rs 中也不可修改，但在 Pod 中可以修改）；

特别注意：name、image、imagePullPolicy 首字母要对齐。

8) 第 21 行设置容器拉取策略为 “imagePullPolicy: IfNotPresent”，这样当本地有对应镜像时，将会使用本地镜像，可以有效降低镜像拉取的时间和失败概率。

问题 1：既然 spec.template 会有 Pod 的创建信息，RC/RS 直接创建/管理该 Pod 即可，为何 RC/RS 中还要有 selector 呢？

答：RC/RS 在创建时，会用该 selector 去匹配 Kubernetes 中已有的 Pod，如果符合条件的 Pod 少于 RC/RS 设置的副本数（replicas），则会根据 spec.template 来创建新的 Pod 来创建新的 Pod 副本；反之，则会删除多余的 Pod。

因此，如果没有 selector，那么 Kubernetes 中已有的无组织的 Pod（不属于任何一个 RC/RS），就管理不起来，因为没有匹配的条件了（只能无条件地匹配 spec.template 的 Pod）。

问题 2：既然 RS 中有了 selector，可以匹配 Kubernetes 中所有的 Pod，为何又要在 spec.template 中指定 Pod，而且要求 Pod 一定要匹配 selector 的条件呢？

答：如果 RS 要创建新的 Pod，那么由于 RS 的 selector 可能会匹配多个 Pod（外部无组织的 Pod），那以哪个为准，来创建新的 Pod 呢，RS 就不清楚了。因此，只能是以 spec.template 来创建新的 Pod，既然这样的话，spec.template 中 Pod 的信息就必须匹配 RS 的 selector，否则，Pod 创建后，RS 就无法匹配到它了。

## （2）使用命令创建 RS

YAML 编辑好后，使用下面的命令来创建 RS。

```
[user@master rc]$ kubectl apply -f rs.yml
```

如果系统打印如下信息，则说明创建成功。

```
replicaset.apps/myrs created
```

## （3）查看 RS

使用下面的命令来查看 RS 的信息。

```
[user@master rc]$ kubectl get rs -o wide
```

RS 的信息显示如图 2-48 所示，其中第二列 DESIRED 表示配置的 Pod 副本个数；第三列 CURRENT 表示当前已经启动的 Pod 副本个数；第四列 READY 表示已启动，并正常工作的 Pod 副本的个数。其余各列根据其名字和具体的值就可以很清楚地看出其中的含义，在

此不再赘述。

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS	IMAGES	SELECTOR
myrs	1	1	1	2m7s	nginx-container	nginx	app=mynginx-pod,ver in (alpha,beta,stable)

2-48 RS 信息图

#### (4) 删除 RS

RS 删除的命令如下

```
[user@master rc]$ kubectl delete rs myrs
```

上述命令在删除 RS 的同时,也会 Terminate RS 所管理的 Pod。如果我们不希望删除 Pod,则可以加上 “--cascade='orphan'” 选项,具体命令如下。

```
[user@master rc]$ kubectl delete rs myrs --cascade='orphan'
```

如果要显示地删除 RS 所管理的 Pod,则可以加上 “--cascade='background'” 选项

RC 的删除命令和原理同上。

### 3. 使用 RS 实现 Pod 扩展

使用 RS 可以很方便地实现 Pod 动态扩展,例如上例中,Pod 的副本只有 1 个,我们可以通过修改创建 RS 的 YAML 文件中的 replicas,很方便地实现 Pod 的动态扩展,具体步骤说明如下。

(1) 查看 Pod 信息,命令如下。

```
[user@master rc]$ kubectl get pod -o wide
```

Pod 信息显示如图 2-49 所示。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
myrs-6bmn6	1/1	Running	0	7m59s	192.168.2.204	node03

2-49 Pod 副本信息图

(2) 修改 rs.yml 中的 replicas 值为 2,如下所示。

```
1 apiVersion: apps/v1
2 kind: ReplicaSet
3 metadata:
4   name: myrs
5 spec:
6   replicas: 2
```

(3) 重新应用修改后的 rs.yml,命令如下。

```
[user@master rc]$ kubectl apply -f rs.yml
```

(4) 查看 Pod 信息,命令如下。

```
[user@master rc]$ kubectl get pod
```

如图 2-50 所示,可以看到一个新的 Pod 正在创建,Pod 的个数等于设置的 replicas 数。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
myrs-6bmn6	1/1	Running	0	10m	192.168.2.204	node03
myrs-98vv1	0/1	ContainerCreating	0	5s	<none>	node01

2-50 Pod 创建状态图

使用类似的方法,可以实现 RC 所控制的 Pod 的扩展。

### 5. RC 和 RS 的使用

RS 是 RC 的下一代控制器,和 RC 相比,RS 在 selector 上既提供了 equality-based (基于相等) matchLabels 匹配,又提供了 set-based (基于集合) matchExpressions,更加灵活。因此在 RC 和 RS 的使用上,我们推荐使用 RS。实际使用中,RS 并不独立使用,而是主要包含在 Deployment 之中,用于 Pod 的创建、删除和更新。根据 Kubernetes 的官方建议,我

们应使用 Deployment，通过 Deployment 来创建和管理 RS。

## 1.5 实践 5---Deployment 典型使用

Deployment 是比 RC/RS 更高级的抽象，它可以很方便地创建 RS 和 Pod，实现集群规模的动态伸缩，同时还有 rollout（回滚）和 pause（暂停）等新功能。本节以示例的形式来说明 Deployment 的典型使用。

### 1. 创建 Deployment

使用 YAML 来创建 Deployment，首先编辑 Deployment 文件，命令如下。

```
[user@master k8s]$ mkdir deploy
[user@master k8s]$ cd deploy/
[user@master deploy]$ vi deployment.yml
```

#### (1) 编辑创建 RS 的 YAML 文件

把 Deployment 的 Object 内容按照 YAML 格式，写入文件 deployment.yml，该 Deployment 将创建 3 个 Pod 副本，具体内容如下。

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: mydeployment
5   labels:
6     app: nginx
7
8 spec:
9   replicas: 3
10  selector:
11    matchLabels:
12      app: nginx
13    matchExpressions:
14      - {key: ver, operator: In, values: [alpha, beta, stable]}
15
16  template:
17    metadata:
18      labels:
19        app: nginx
20        ver: beta
21    spec:
22      containers:
23        - name: nginx
24          image: nginx:latest
25          imagePullPolicy: IfNotPresent
```

配置文件说明如下。

- 1) 第 1 行为 Object 的第一个成员 `apiVersion`，值是 `apps/v1`；
- 2) 第 2 行为 Object 的第二个成员 `kind`，值是 `Deployment`；
- 3) 第 4~6 行为 Object 的第三个成员 `metadata`，`metadata` 是一个 Object，它有一个成员 `name`，其值是 `mydeployment`，`mydeployment` 不能和 `Kubernetes` 中的其他 `Deployment` 同名；

4) 其余第 8~25 行内容同 2.6 节 rs.yml 中 5~21 行, 具体说明见 2.6 节 rs.yml 的相关说明, 在此不再赘述。

## (2) 使用命令创建 Deployment

YAML 编辑好后, 使用下面的命令来创建 Deployment。

```
[user@master deploy]$ kubectl apply -f deployment.yml
```

如果系统输出以下内容, 则说明创建成功。

```
deployment.apps/mydeployment created
```

## (3) 查看 Deployment

查看 Deployment 的信息, 命令如下。

```
[user@master deploy]$ kubectl get deployment -o wide
```

Deployment 信息如图 2-51 所示。

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS	IMAGES	SELECTOR
mydeployment	3/3	3	3	40s	nginx	nginx:latest	app=nginx,ver in (alpha,beta,stable)

2-51 Deployment 信息图

Deployment 创建的同时, 会创建对应的 RS, 如图 2-52 所示。

NAME	DESIRED	CURRENT	READY	AGE	CONTAINERS	IMAGES	SELECTOR
mydeployment-88bbb46f	3	3	3	2m10s	nginx	nginx:latest	app=nginx,pod-template-hash=88bbb46f,ver in (alpha,beta,stable)

2-52 RS 信息图

RS 中的 Pod 信息如图 2-53 所示, 可以看到 3 个 Pod, 分别运行在 node01 和 node03 上。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
mydeployment-88bbb46f-4d5q9	1/1	Running	0	3m10s	192.168.2.131	node01
mydeployment-88bbb46f-6sxxv	1/1	Running	0	3m10s	192.168.2.209	node03
mydeployment-88bbb46f-ft4jx	1/1	Running	0	3m10s	192.168.2.193	node03

2-53 Pod 信息图

## (4) 删除 Deployment

Deployment 删除命令如下所示, 该命令会自动删除 Deployment 对应的 RS 和 Pod。

```
[user@master deploy]$ kubectl delete deploy mydeployment
```

## 2. Deployment 滚动更新和暂停

Deployment 的滚动更新, 是指创建 Deployment 之后, 如果修改 Deployment 中 Pod 的设置, 例如修改 Pod 容器的镜像等, 则该 Deployment 会自动启动一个新的 Pod, 待该 Pod 可用后, 删除一个老的 Pod, 循环执行, 直至新的 Pod 全部替换老的 Pod。

下面介绍 Deployment 实现滚动更新的具体示例, 该示例首先创建一个 Deployment, 它的 Pod 有 1 个容器, 容器镜像是 nginx, 副本数是 3; 然后修改 Pod 的容器镜像为 busybox, 副本数不变, 最后使用 Deployment 实现 Pod 的滚动更新。具体步骤如下。

### (1) 使用命令创建 Deployment

首先准备 Deployment 的 YAML 文件 dep-nginx.yml, 直接复制 deployment.yml 即可, 命令如下。

```
[user@master deploy]$ cp deployment.yml dep-nginx.yml
```

使用下面的命令创建 Deployment。Kubernetes 会为每次操作编一个版本号, 其中 --record=true 用来记录每次操作所使用的命令, 便于用户了解每个版本所做的操作。

```
[user@master deploy]$ kubectl apply -f dep-nginx.yml --record=true
```

### (2) 编辑新的 Deployment 文件

准备一个新的 Deployment 的 YAML 文件 dep-busybox.yml, 修改其中 Pod 容器的镜像为 busybox, 具体步骤如下。

#### 1) 准备 dep-busybox.yml, 命令如下。

```
[user@master deploy]$ cp dep-nginx.yml dep-busybox.yml
```

2) 编辑 `dep-busybox.yml`, 命令如下。

```
[user@master deploy]$ vi dep-busybox.yml
```

修改 `dep-busybox.yml` 的内容如下。

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: mydeployment
5   labels:
6     app: nginx
7
8 spec:
9   replicas: 3
10  selector:
11    matchLabels:
12      app: nginx
13    matchExpressions:
14      - {key: ver, operator: In, values: [alpha, beta, stable]}
15
16  template:
17    metadata:
18      labels:
19        app: nginx
20        ver: beta
21    spec:
22      containers:
23        - name: nginx
24          image: busybox:latest
25          imagePullPolicy: IfNotPresent
26          command:
27            - /bin/sh
28          stdin: true
29          tty: true
```

上述配置和 `dep-nginx.yml` 相比, 修改了第 24 行, 将 `image` 由 `nginx` 修改成 `busybox`; 增加第 26 行 `command`, 用来配置容器启动时运行的命令为 `/bin/sh`, 因为 `busybox` 没有指定默认启动程序; 增加了第 28 行 `stdin:true`, 即打开容器的 `stdin`, 从而接受输入; 增加第 29 行 `tty:true`, 为容器分配一个伪终端, 并将其绑定在容器的 `stdin` 上, 第 28 和第 29 行都是 `sh` 程序运行时必须要的。

### (3) 更新 Deployment

运行下面的命令来更新 Deployment。

```
[user@master deploy]$ kubectl apply -f dep-busybox.yml --record=true
```

此时 **Kubernetes** 会按照 `dep-busybox.yml` 中的配置, 启动一个新的 Pod, 如图 2-54 所示, 该 Pod 此时的状态为 `ContainerCreating`, Pod 中容器的镜像为 `busybox`。



```
[user@master deploy]$ kubectl get pod -o wide
```

NAME	READY	STATUS	RESTARTS
mydeployment-7c88d9b568-b29cp	0/1	ContainerCreating	0
mydeployment-88bbb46f-2kbcm	1/1	Running	0
mydeployment-88bbb46f-6r9fv	1/1	Running	0
mydeployment-88bbb46f-82cxf	1/1	Running	0

2-54 Pod 信息图

暂停滚动更新的进程，命令如下。

```
[user@master deploy]$ kubectl rollout pause deploy mydeployment
```

此时，Kubernetes 会停止创建新的 Pod 和删除旧的 Pod，但正在创建的 Pod 和正在删除的 Pod 不会暂停，它们会继续工作，直至 Pod 创建好或删除掉。如果要恢复滚动更新，则可以使用下面的命令。

```
[user@master deploy]$ kubectl rollout resume deploy mydeployment
```

此时，Kubernetes 会继续启动滚动更新，当一个新的 Pod 准备好后，就会删除一个老的 Pod，如图 2-55 所示，从而实现新 Pod 对老 Pod 的替换。

NAME	READY	STATUS
mydeployment-7c88d9b568-b29cp	1/1	Running
mydeployment-7c88d9b568-dx7x2	1/1	Running
mydeployment-7c88d9b568-r18r7	1/1	Running
mydeployment-88bbb46f-2kbcm	0/1	Terminating

2-55 Pod 状态图

Kubernetes 会重复上面的替换，直至所有的老 Pod 被替换掉，如图 2-56 所示。

NAME	READY	STATUS
mydeployment-7c88d9b568-b29cp	1/1	Running
mydeployment-7c88d9b568-dx7x2	1/1	Running
mydeployment-7c88d9b568-r18r7	1/1	Running

2-56 新 Pod 运行图

查看每个 Pod 的信息，检查其容器镜像是否已经替换成了 busybox，命令如下。

```
[user@master deploy]$ kubectl describe pod mydeployment-7c88d9b568-b29cp | grep busybox
```

如果系统输出以下信息，则说明更新成功。

```
Image:          busybox:latest
```

Deployment 的滚动更新是自动完成的。

而 RS/RC 则是手动更新，必须手动删除 Pod，才新建一个 Pod（按照新的配置）。

### 3. Deployment 版本回滚

如果更新后的版本有问题，需要退回到前面的某个版本，则可以使用 Deployment 的版本回滚功能，具体操作步骤如下。

1) 查看 Deployment 的各个版本，命令如下。

```
[user@master deploy]$ kubectl rollout history deploy mydeployment
```

上述命令后，可以看到每个版本所对应的操作。

REVISION	CHANGE-CAUSE
1	<none>
2	kubectl apply --filename=dep-nginx.yml --record=true
3	kubectl apply --filename=dep-busybox.yml --record=true

2) 选择回滚到版本 2，命令如下，其中--to-revision=2 用来指定要回滚的版本为 2，可以用其他的版本号来替换 1，从而回滚到对应的版本。

```
[user@master deploy]$ kubectl rollout undo deploy mydeployment --to-revision=2
```

上述命令会输出回滚的中间结果，如图 2-57 所示。

NAME	READY	STATUS
mydeployment-7c88d9b568-b29cp	1/1	Running
mydeployment-7c88d9b568-dx7x2	1/1	Running
mydeployment-7c88d9b568-r18r7	1/1	Running
mydeployment-88bbb46f-24zl2	0/1	ContainerCreating

2-57 Deployment 回滚状态图

回滚结束后的 Pod 信息如图 2-58 所示。

NAME	READY	STATUS	RESTARTS	AGE
mydeployment-7c88d9b568-b29cp	1/1	Terminating	0	30m
mydeployment-88bbb46f-24zl2	1/1	Running	0	12m
mydeployment-88bbb46f-5qljw	1/1	Running	0	50s
mydeployment-88bbb46f-hqsgw	1/1	Running	0	12m

2-58 回滚后的 Pod 信息图

查看每个 Pod 中的容器镜像是否为 **nginx**，命令如下。

```
[user@master deploy]$ kubectl describe pod mydeployment-88bbb46f-24zl2 | grep nginx
```

上述命令执行后，如果系统输出以下信息，就说明回滚成功。

```
Image:      nginx:latest
```

## 1.6 实践 6---Service 典型使用

Service 解决了用户访问 Pod 服务的两个问题：首先 Service 为用户提供了一种固定的方式（通常是一个固定的虚拟 IP）来访问 Pod 服务，用户只需要访问该虚拟 IP，就可以访问到对应 Pod 所提供的服务，而不需要关心服务到底是由 Pod 的哪个副本提供的，这个 Pod 副本的 IP 是多少，即使 Pod 副本的 IP 发生了变化，用户通过 Service 的虚拟 IP，也总能访问到 Pod 所提供的服务；其次 Service 可以使得 **Kubernetes** 以外的节点能够访问 Pod 服务。因此，本节将介绍 Service 解决上述 Pod 服务的两个问题的典型示例，具体说明如下。

### 1. 示例一：基于 Service 的虚拟 IP 来访问 Pod 服务

#### (1) 准备新的虚拟机节点 node02

由于后续实验环境中需要 2 个 Node，而目前只有 1 个 Node 节点 node01，因此，需要增加一个新的虚拟机节点 node02（IP 地址为 192.168.0.228），该节点由 node01 复制而来；修改虚拟机名称、主机名、IP 地址；运行“**kubeadm reset**”；最后运行“**kubeadm join XX**”将 node02 加入到 Kubernetes 集群中，使得最后 Kubernetes 集群的 Node 信息如下所示。

```
[user@master deploy]$ kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
master	Ready	control-plane,master	45h	v1.20.1
node01	Ready	<none>	44h	v1.20.1
node02	Ready	<none>	12s	v1.20.1

注意：每个节点的 hosts 文件中要增加 node02 的主机名-IP 映射项。

#### (2) 在没有 Service 的情况下访问 Pod 服务

1) 使用 Deployment 来创建一个 Pod，该 Pod 使用 **nginx** 提供 Web 服务，Pod 副本数为 1，其 YAML 文件为 **dep-nginx.yml**，操作步骤如下。

```
[user@master k8s]$ mkdir svc
```

```
[user@master k8s]$ cd svc/
```

添加 **dep-nginx.yml** 的内容如下。

- 1 **apiVersion**: apps/v1
- 2 **kind**: Deployment



```

3 metadata:
4   name: dep-nginx
5   labels:
6     app: nginx
7
8 spec:
9   replicas: 1
10  selector:
11    matchLabels:
12      app: nginx
13    matchExpressions:
14      - {key: ver, operator: In, values: [alpha, beta, stable]}
15
16  template:
17    metadata:
18      labels:
19        app: nginx
20        ver: beta
21    spec:
22      containers:
23        - name: nginx
24          image: nginx:latest
25          imagePullPolicy: IfNotPresent

```

上述配置文件的内容可以参考 6.3.4 节中的说明，此处不再赘述。

2) 创建 Deployment，命令如下。

```
[user@master svc]$ kubectl apply -f dep-nginx.yml
```

查看 Pod 信息，命令如下。

```
[user@master svc]$ kubectl get pod -o wide
```

如图 2-59 所示，Pod 在 node02 上运行，Pod 的 IP 地址为 192.168.2.65。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
dep-nginx-54d5b4d898-jj85r	1/1	Running	0	88s	192.168.2.65	node02

2-59 Pod 信息图

3) 在 master 节点上，测试 master 节点是否可以 ping 通该 Pod，命令如下。结果显示，master 节点是可以 ping 通该 Pod 的。

```
[user@master svc]$ ping 192.168.2.65
```

```
PING 192.168.2.65 (192.168.2.65) 56(84) bytes of data.
```

```
64 bytes from 192.168.2.65: icmp_seq=1 ttl=63 time=0.592 ms
```

4) 在 master 使用 curl 命令获取 Pod 的主页信息，以此模拟访问 Pod 所提供的 web 服务，命令如下。

```
[user@master svc]$ curl 192.168.2.65
```

如果系统打印下面的 Welcome 信息，则说明访问该 Pod 的 Web 服务成功。

```
<title>Welcome to nginx!</title>
```

5) 删除该 Pod，模拟 Pod 节点不可用，命令如下。

```
[user@master svc]$ kubectl delete pod dep-nginx-54d5b4d898-jj85r
```

6) 观察新 Pod, Pod 删除后, Deployment 会启动一个新的 Pod, 如图 2-60 所示, 新 Pod 所在节点为 node02, 新 Pod 的 IP 为 192.168.2.66, 和原来的 192.168.2.65 不同。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
dep-nginx-54d5b4d898-8vgjx	1/1	Running	0	20s	192.168.2.66	node02

2-60 新 Pod 信息图

### 7) 访问 Pod 服务

由于 Pod 所在节点发生了改变, 此时访问 Pod 服务就要使用新节点的 IP 192.168.2.66 了, 命令如下。

```
[user@master deploy]$ curl 192.168.2.66
```

因此, 在不使用 Service 之前, Pod 服务是和 Pod 的 IP 紧密绑定的, 一旦发生变化, 就要先获取新的 Pod IP, 然后使用该 IP 去访问 Pod 所提供的 Web 服务。这种使用方式, 就好比访问 baidu 网站, 先要手动获取一次 baidu 的 IP, 然后才能根据该 IP 去访问网站, 这个对于用户来说是不可接受的。

### (3) 使用 Service 访问 Pod 服务

针对上述问题, Kubernetes 提供了基于 Service 的解决方案, 可以创建一个 Service, 该 Service 一端连接着 Pod, 另外一端则对外提供一个虚拟 IP, 不管后端的 Pod 如何变化, 用户始终只需要通过该虚拟 IP, 就能访问到 Pod 所提供的服务, 这中间的转接就由 Service 来完成, 具体示例的操作步骤说明如下。

#### 1) 编辑 Service 的 YAML 文件 `svc-nginx.yml`, 命令如下。

```
[user@master svc]$ vi svc-nginx.yml
```

#### 2) 在 `svc-nginx.yml` 中编辑以下内容。

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: svc-nginx
5
6 spec:
7   ports:
8     - name: http
9       port: 80
10      targetPort: 80
11      protocol: TCP
12   selector:
13     app: nginx
```

上述配置文件说明如下。

1. 第 1 行为 Service Object 的第一个成员 `apiVersion`, 值是 `v1`;
2. 第 2 行为 Service Object 的第二个成员 `kind`, 值是 `Service`;
3. 第 3~4 行为 Service Object 的第三个成员 `metadata`, 它是一个 Object, 只有 1 个成员 `name`, 其值为 `svc-nginx`;
4. 第 6~13 行为 Service Object 的第四个成员 `spec`, 它有两个成员, 第一个是 `ports`, 第二个是 `selector`, 具体说明如下。
  - 第一个成员 `ports` 是一个数组, 数组中有 1 个元素, 该元素有 4 个成员, 第一个成员是 `name`, 用于该元素的命名; 第二个成员 `port`, 其值为 80, 表示该 Service 对外的端口为 80; 第三个成员 `targetPort`, 表示 Service 对内连接 Pod 的端口, 即

Pod 的端口为 80；第四个成员 protocol 表示该 Service 所使用的底层协议，本例为 TCP（这个可以不用写，默认就是 TCP）；

- 第二个成员 selector，用来匹配该 Service 所对应的 Pod，该 selector 也是基于相等判断的，即哪个 Pod 的 labels 中设置了 app: nginx，则该 Pod 条件匹配。基于 selector，Service 实现了 Pod 的逻辑分组。

3）运行下面的命令来创建 Service。

```
[user@master svc]$ kubectl apply -f svc-nginx.yml
```

4）使用下面的命令查看 Service。

```
[user@master svc]$ kubectl get svc -o wide
```

系统会打印刚创建的 Service svc-nginx，如图 2-61 所示，其中第 3 列 CLUSTER-IP，就是该 Service 对外的虚拟 IP，本例中为 10.104.60.206。

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	46h	<none>
svc-nginx	ClusterIP	10.98.60.206	<none>	80/TCP	11s	app=nginx

2-61 Service 信息图

5）使用该虚拟 IP 10.98.60.206 访问该 Pod 所提供的服务，命令如下。

```
[user@master svc]$ curl 10.98.60.206
```

6）删除 Pod，模拟 Pod 不可用，命令如下。

```
[user@master svc]$ kubectl delete pod dep-nginx-54d5b4d898-8vgjx
```

7）观察新 Pod，Pod 删除后，Deployment 会启动一个新的 Pod，如图 6-38 所示，新 Pod 所在节点为 node02，新 Pod 的 IP 为 192.168.2.67，和原来的 192.168.2.66 不同。

8）在 master、node01 或 node02 任意一个节点上使用虚拟 IP 10.98.60.206 访问新 Pod 的服务，命令如下。

```
[user@master svc]$ curl 10.98.60.206
```

有了 Service，我们再也不用像前面那样，在访问 Pod 服务时，先查询 Pod 副本的 IP，才能去访问 Pod 服务。而是直接使用 Service 的虚拟 IP 就可以直接访问，而不用管提供服务的 Pod 副本到底在哪个节点上，即便 Pod 的 IP 发生改变也没有关系。

---

即使 Pod 因为被删除或者所在节点宕机，导致 Pod 被重新创建，重新分配 IP。我们依然能够通过 Service 的虚拟 IP，来访问 Pod 所提供的服务。

---

## 2. 示例二：基于 Service 实现从 Kubernetes 外部访问 Pod 服务

示例一的 Service 提供的虚拟 IP 只能供 Kubernetes 内部访问，即 Kubernetes 集群的节点或者 Kubernetes 中 Pod 的容器可以访问该虚拟 IP。对于 Kubernetes 以外的节点，即使和 Kubernetes 在同一个网络之上，也是无法访问该虚拟 IP 的，这样就导致 Pod 提供的服务，无法被 Kubernetes 以外的节点所访问。

使用 Service 可以解决上述问题，Service 提供了 NodePort 机制，它可以将 Kubernetes 集群节点上的端口映射到 Service 的端口，这样，只要外部节点能够访问 Kubernetes 上的节点，它就可以通过端口映射，来访问 Pod 服务了，具体步骤说明如下。

（1）编辑 Service 的 YAML 文件

1）该文件的名称为 svc-nginx-nodeport.yml，编辑命令如下。

```
[user@master svc]$ vi svc-nginx-nodeport.yml
```

2）在 svc-nginx-nodeport.yml 中输入以下内容。

```
1 apiVersion: v1
2 kind: Service
3 metadata:
```

```

4  name: svc-nginx
5
6  spec:
7  type: NodePort
8  ports:
9  - name: http
10    port: 80
11    targetPort: 80
12    protocol: TCP
13    nodePort: 30001
14  selector:
15    app: nginx

```

`svc-nginx-nodeport.yml` 和 `svc-nginx.yml` 相比，增加了第 7 行 “`type: NodePort`” 用来指定该 Service 的类型是 `NodePort`，如果不指定 `type`，默认是 `ClusterIP`，即 `svc-nginx.yml` 中的 Service 类型；第 13 行 “`nodePort: 30001`” 用来指定节点上的端口，该端口的范围是 30000-32767 (<http://docs.kubernetes.org.cn/703.html>)，当外部节点访问 `Kubernetes` 节点的 30001 端口时，`svc-nginx` Service 会将该请求转发到 Service 的 80 端口 (`port`)，就好像内部节点访问 Service 的 80 端口一样，继而再转发到 Pod 的 80 端口 (`targetPort`)。配置文件的其余重复内容在此不再赘述。

3) 创建 Service，命令如下。

```
[user@master svc]$ kubectl apply -f svc-nginx-nodeport.yml
```

4) 查看 Service 信息，命令如下。

```
[user@master svc]$ kubectl get svc -o wide
```

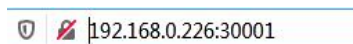
我们可以看到 `svc-nginx` Service 的信息如下，其中 80:30001/TCP 表示端口映射信息。

svc-nginx	NodePort	10.98.60.206	<none>	80:30001/TCP	6h52m	app=nginx

5) 在 `Kubernetes` 内部使用虚拟 IP 10.98.60.206 访问 Pod 的 Web 服务，命令如下。

```
[user@master svc]$ curl 10.98.60.206
```

6) 在 `Kubernetes` 外部的 Host 主机上使用浏览器来访问 Pod 的 Web 服务，如图 2-62 所示。



2-62 浏览器地址栏

输入 `Kubernetes` 中任何一个节点的外部 IP，例如 `node01` 的 192.168.0.227，或者 `node02` 的 192.168.0.228，都可以访问到该 Pod 的 Web 服务。

如果浏览器返回图 2-63 所示页面，说明 Pod 服务访问成功。

## Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to [nginx.org](http://nginx.org).  
Commercial support is available at [nginx.com](http://nginx.com).

Thank you for using nginx.

2-63 Nginx Web 页面图

总之，使用 Service 的 `NodePort` 机制，可以实现外部节点访问 Pod 服务，而且只要外部

节点能够访问到 **Kubernetes** 集群的任意一个节点即可，可用性也很高。但是，由于端口映射是全局的，稍显不灵活，例如要发布多个 Web 服务，那么这些 Web 服务的端口就会要映射到不同的端口，无法都使用平时默认的 80 端口，这就会造成用户使用习惯的不同。后续，本书会介绍 Service 的另一种类型 Ingress，来解决上述问题。

## 1.7 实践 7---Pod 多容器运行实践

Pod 是 **Kubernetes** 中的最小运行单元，也是微服务的载体，很多时候，一个服务需要多个容器相互协作。因此，在 Pod 中运行多个容器是很常见的应用场景。本节介绍在 Pod 中运行多个容器的具体示例，它将在 Pod 中启动两个容器，分别是 **nginx** 和 **busybox**，把它们放在一个 Pod 中，注意是为了便于演示，具体步骤说明如下。

### 1. 编写 Deployment

1) 编辑 Deployment 的 YAML 文件，文件名为 **dep-pod-multi-container.yml**，命令如下。

```
[user@master k8s]$ cd deploy/
```

```
[user@master deploy]$ vi dep-pod-multi-container.yml
```

2) 在 **dep-pod-multi-container.yml** 增加以下内容。

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: multicontainer
5
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10      app: multicontainer
11
12 template:
13   metadata:
14     labels:
15       app: multicontainer
16   spec:
17     containers:
18       - name: myfrontend
19         image: nginx:latest
20         imagePullPolicy: IfNotPresent
21       - name: busybox
22         image: busybox:latest
23         imagePullPolicy: IfNotPresent
24     command:
25       - /bin/sh
26     tty: true
27     stdin: true
```

关键配置说明。

- 第 6 行~27 行是 Pod 的配置信息，其中第 7 行设置了 Pod 的副本数为 1，第 8~10

行为 Pod 容器的 selector 设置：

- 第 12~27 行是 Pod 容器的配置信息，第 13~15 行是 Pod 容器的 metadata，其中设置了 labels 信息 `app:multicontainer`，该信息必须要和前面 Pod 容器的 selector 相匹配；
- 第 17 行 `containers` 是一个数组，实现了 Pod 对多个容器的支持，该数组的第一个元素内容对应第 18~20 行，该元素的容器名为 `myfrontend`，镜像名为 `nginx:latest`，镜像 pull 策略为 `IfNotPresent`；数组的第二个元素对应第 20~27 行，该元素的容器名为 `busybox`，镜像名为 `busybox:latest`，镜像 pull 策略为 `IfNotPresent`，`command` 用来设置容器启动时运行的程序及参数，也是一个数组，本例只有一个元素，即 `/bin/sh, tty:true` 表示为该容器分配一个终端，并将其绑定在容器的 stdin 上，`stdin:true` 表示打开容器的 stdin，从而接受输入。

## 2. 创建 Deployment

1) 运行下面的命令创建 Deployment。

```
[user@masterdeploy]$ kubectl apply -f dep-pod-multi-container.yml
```

2) 可以查看 Pod 信息，命令如下。

```
[user@masterdeploy]$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
<code>multicontainer-5895455dfd-nv4hb</code>	2/2	Running	0	2m24s

3) 由上述命令的输出可知，新 Pod 的名称为 `multicontainer-5895455dfd-nv4hb`，以此查看该 Pod 中的容器信息，命令如下。

```
[user@masterdeploy]$ kubectl describe pod multicontainer-5895455dfd-nv4hb
```

如果系统打印 `myfrontend` 和 `busybox` 两个容器的名称，以及它们的 State 都是 `running`，则说明 Pod 中的容器正常运行。

## 3. 查看 Pod 中的容器信息

1) 到 Pod 所在的节点来查看容器的信息，首先获取 Pod 所在节点，命令如下。

```
[user@masterdeploy]$ kubectl get pod -o wide
```

上述命令输出结果如下，可知 Pod 在 `node02` 上运行，如图 6-47 所示。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
<code>dep-nginx-sel-node-599c7cb967-55c97</code>	1/1	Running	0	62m	192.168.2.147	node01
<code>multicontainer-5895455dfd-nv4hb</code>	2/2	Running	0	6m35s	192.168.2.96	node02

2) 在 `node02` 上查看容器信息，首先查看第一个容器 `myfrontend`，命令如下。

```
[user@node02 ~]$ docker ps -a | grep myfrontend
```

如果系统打印如下信息，则说明 Pod 中该容器正常运行，其中容器 ID 为 `a962a423d095`。

```
a962a423d095    ae2feff98a0c    "/docker-entrypoint..."    16 minutes ago    Up 16 minutes
k8s_myfrontend_multicontainer-5895455dfd-nv4hb_default_6225bff7-d534-44ba-a69f-603e52a6bb99_0
```

3) 使用同样的方法查看第二个容器 `busybox` 的信息，命令如下。

```
[user@node02 ~]$ docker ps -a | grep busybox
```

如果系统打印如下信息，则说明 Pod 中该容器正常运行，其中容器 ID 为 `621ca1ff6e0b`。

```
963d12c1f370    219ee5171f80    "/bin/sh"    17 minutes ago    Up 17 minutes
k8s_busybox_multicontainer-5895455dfd-nv4hb_default_6225bff7-d534-44ba-a69f-603e52a6bb99_0
```

4) 登录容器 `busybox`，命令如下。

```
[user@node02 ~]$ docker exec -it 963d12c1f370 /bin/sh
/#
```

5) 在 `busybox` 中查看网络，命令如下。

```
/# ip a
```



上述命令会打印 **busybox** 容器中的网卡信息,如下所示,其中网卡名称是 **eth0@if1.....**,这个网卡也是 Pod 的网卡,其 IP 就是 Pod 所分配的 IP。Pod 中所有的容器使用的是同一块网卡和同一个 IP,容器之间的网络通信通过 **localhost+端口**即可。

```
4: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN>mtu 1440 qdiscnoqueue
    link/ether 6a:98:29:3a:78:46 brdff:ff:ff:ff:ff:ff
inet 192.168.2.154/32 scope global eth0
valid_lft forever preferred_lft forever
```

6) 在 Pod 中使用 “**kubect exec**” 来直接运行 **busybox** 容器的命令, 命令如下。

```
[user@masterdeploy]$kubectl execbusybox_multicontainer-5895455dfd-nv4hb -c busyboxip a
```

上述命令的参数说明如下。

- **busybox\_multicontainer-5895455dfd-nv4hb** 是 Pod 的名字;
- “**-c busybox**” 用来指定运行命令的容器是 **busybox**;
- 后面跟的 “**ip a**” 用来指定在容器中运行的命令。

## 1.8 实践 8---Kubernetes 中容器的高可用实践

导致容器不可用的因素有很多,例如容器自身的不可用、Pod 不可用或 Node 不可用等,Kubernetes 针对这些不可用因素,分别给出了相应的解决方案,以此实现容器的高可用,具体说明如下。

### 1. 因素一: 容器自身不可用

容器中进程的崩溃、容器运行中的异常等因素都可能导致容器自身不可用。这种情况下,容器所在的 Pod 是正常工作的,它会重新启动容器,以确保其可用性,具体示例操作步骤如下。

#### (1) 创建 Pod

1) 进入 **/home/user/k8s/pod** 目录, 创建 Pod 命令如下。

```
[user@master pod]$ kubectl apply -f pod.yml
```

2) 查看 Pod, 命令如下。

```
[user@master pod]$ kubectl get pod
```

如果系统输出如下内容, 则说明 Pod 创建成功, 且 Pod 位于 **node02** 节点上。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
mynginx	1/1	Running	0	60s	192.168.2.84	node02

#### (2) 模拟容器自身不可用

1) 登录 **node02**, 查看容器 **nginx** 信息, 命令如下。

```
[user@node02 ~]$ docker ps -a | grep nginx
```

系统会输出容器的 ID 为 **7d6b54a61fd0**, 如下所示。

```
7d6b54a61fd0      nginx
```

2) 停止该容器, 模拟容器自身不可用, 命令如下。

```
[user@node02 ~]$ docker stop 7d6b54a61fd0
```

```
7d6b54a61fd0
```

#### (3) 查看 Pod 信息

1) 连续查看 Pod 信息, 命令如下, 可以看到 Pod 的状态先变成了 **Completed**, 后续又变成了 **Running**。

```
[user@master pod]$ kubectl get pod -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
mynginx	0/1	Completed	0	6m26s	192.168.2.84	node02

```
[user@master pod]$ kubectl get pod -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
Mynginx	1/1	Running	1	6m28s	192.168.2.84	node02

2) 查看 Pod 事件信息, 命令如下。

```
[user@master pod]$ kubectl describe pod mynginx | grep Normal
```

如图 2-64 所示, Kubernetes 在 4m33s 时重新启动容器。

```
Normal    Scheduled    10m                default-scheduler    Successfully assigned default/mynginx to node02
Normal    Pulled       10m                kubelet              Successfully pulled image "nginx" in 17.044255547s
Normal    Pulling      4m51s (x2 over 10m) kubelet              Pulling image "nginx"
Normal    Created      4m33s (x2 over 10m) kubelet              Created container nginx-container
Normal    Started      4m33s (x2 over 10m) kubelet              Started container nginx-container
Normal    Pulled       4m33s              kubelet              Successfully pulled image "nginx" in 17.94344866s
```

2-64 Pod 事件图

3) 在 node02 上查看容器 nginx 的 ID, 命令如下。

```
[user@node02 ~]$ docker ps -a | grep nginx
```

上述命令输出容器 nginx 的新 ID 为 28b966cbe09c, 说明这是一个新启动的容器。

```
28b966cbe09c      nginx
```

因此, 对于 Pod 中的容器来说, Pod 会检查容器的状态, 并且在容器不可用的情况下, 重新启动容器, 以确保其可用性。

使用命令创建 Pod 时, 如果加入 `--restart=OnFailure` 或 `--restart=Never`, 则 Pod 内部容器不可用时, Kubernetes 将不会在该 Pod 内部重新启动对应的容器。

## 2. 因素二: Pod 不可用

Pod 不可用是导致其内部容器不可用的另一大因素, 此处的 Pod 不可用是指 Pod 自身的不可用, 例如 Pod 被删除等, 而不是指由 Pod 所在节点的不可用。在这种情况下, 该 Pod 内部容器的可用性要分两种情况讨论, 说明如下。

第一种, 该 Pod 是直接创建的, 也就是说直接创建的就是 Pod object, 而不是通过 RC/RS 或者 Deployment 创建的 Pod。这种情况下, 如果 Pod 不可用了, Kubernetes 不会有任何动作;

第二种, 该 Pod 是由 RC/RS 或者 Deployment 创建的, 那么 Kubernetes 会重新启动 Pod, 下面以 RS 创建的 Pod 为例进行说明。

### (1) 通过 RS 创建 Pod

1) 进入 RS 的 YAML 文件所在目录, 命令如下。

```
[user@master pod]$ cd ~/k8s/rc/
```

2) 通过 RS 创建 Pod, 命令如下。

```
[user@master rc]$ kubectl apply -f rs.yml
```

3) 查看 Pod 信息, 命令如下。

```
[user@master rc]$ kubectl get pod -o wide
```

系统输出 Pod 信息, 如下所示, 该 Pod 有 2 个副本, 分别位于 node01 和 node02 上。

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
myrs-ndbqm	1/1	Running	0	5s	192.168.2.86	node02
myrs-x5bld	1/1	Running	0	5s	192.168.2.87	node02

### (2) 模拟 Pod 不可用

1) 删除 Pod 来模拟 Pod 不可用, 命令如下。

```
[user@master rc]$ kubectl delete pod myrs-ndbqm
```

2) 查看 Pod 信息, 命令如下。

```
[user@master ~]$ kubectl get pod
```

可以看到，Pod myrs-ndbqm 正被删除，而新的 Pod myrs-x5bld 已经运行。

NAME	READY	STATUS	RESTARTS	AGE
myrs-mpdhn	1/1	Running	0	24s
myrs-ndbqm	0/1	Terminating	0	2m19s
myrs-x5bld	1/1	Running	0	2m19s

### 3. 因素三：Node 不可用

Node 的不可用将导致该 Node 上的 Pod 不可用，继而导致这些 Pod 中的容器不可用。当 Kubernetes 检测到 Node 不可用时，会给该 Node 加上一个 Taint（污点），采用默认设置创建的 Pod 将会在 300 秒后被驱离（<https://kubernetes.io/zh/docs/concepts/scheduling-eviction/taint-and-toleration/>），然后在其他可用的 Node 节点上启动新的 Pod，从而实现 Pod 以及内部容器的可用性，具体示例说明如下。

#### （1）准备示例环境

本例的环境需要 node01 和 node02 两个节点，确认命令如下。

```
[user@master rc]$ kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
Master	Ready	control-plane,master	2d8h	v1.20.1
node01	Ready	<none>	2d7h	v1.20.1
node02	Ready	<none>	10h	v1.20.1

#### （2）创建 Pod

1）删除前面创建的 RS，命令如下。

```
[user@master rc]$ kubectl delete rs myrs
```

2）创建 Deployment，命令如下。

```
[user@master rc]$ cd ~/k8s/deploy/
```

```
[user@master deploy]$ kubectl apply -f dep-nginx.yml
```

此处 dep-nginx.yml 就是 6.3.4 节中的 dep-nginx.yml 文件。

#### （3）模拟 Node 不可用

1）查看 Pod 信息，命令如下，可知 Pod 所在节点为 node01 和 node02。

```
[user@master deploy]$ kubectl get pod -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
mydeployment-54d5b4d898-8qxnt	1/1	Running	0	63s	192.168.2.88	node02
mydeployment-54d5b4d898-l5w55	1/1	Running	0	63s	192.168.2.137	node01

2）关闭虚拟机 node01

按下 VMware workstation 中的 node01 的“挂起”按钮，挂起该虚拟机。

3）获取 Node 信息，直到 node01 变成 NotReady，命令如下。

```
[user@master deploy]$ kubectl get node
```

NAME	STATUS	ROLES	AGE	VERSION
Master	Ready	control-plane,master	2d8h	v1.20.1
node01	NotReady	<none>	2d7h	v1.20.1
node02	Ready	<none>	10h	v1.20.1

4）查看 node1 的“污点”信息，命令如下。

```
[user@master deploy]$ kubectl describe node node01 | grep Taint
```

系统将输出以下信息，其中 node.kubernetes.io/unreachable 是“污点”信息的 key，NoExecute 是“污点”信息的 effect，这些信息是 Kubernetes 检测到 node01 不可用后给 node01 加上去的。

```
Taints:                node.kubernetes.io/unreachable:NoExecute
```

#### (4) 查看 Pod 信息

1) 查看 node01 上的 Pod 的配置信息，命令如下。

```
[user@master deploy]$ kubectl edit pod mydeployment-54d5b4d898-l5w55
```

2) 在系统输出信息中，找到 Pod 默认的“容忍点”配置，如下所示。

```
tolerations:
- effect: NoExecute
  key: node.kubernetes.io/not-ready
  operator: Exists
  tolerationSeconds: 300
- effect: NoExecute
  key: node.kubernetes.io/unreachable
  operator: Exists
  tolerationSeconds: 300
```

由上述配置可知，Pod 的“容忍点”在 key 为 node.kubernetes.io/unreachable 时的 tolerationSeconds 时间为 300s，也就是说 300s 后，会驱离该 Node 上的 Pod。

3) 查看 Pod 信息，可以看到 node01 上的 Pod 正在 Terminating，一个新的 Pod mydeployment-54d5b4d898-pjlgm 正在 running，这正是 Kubernetes 驱离 Pod 后新启动的 Pod。

```
[user@master deploy]$ kubectl get pod
```

NAME	READY	STATUS	RESTARTS	AGE
mydeployment-54d5b4d898-8qxnt	1/1	Running	0	6m52s
mydeployment-54d5b4d898-pjlgm	1/1	Running	0	49s
mydeployment-54d5b4d898-l5w55	1/1	Terminating	0	6m52s

#### (5) 总结

上述测试在 v 1.20.1 版本上顺利实现了 Pod 的驱离，但是，实际测试中不同的 Kubernetes 版本在这个功能上并不稳定（包括 v 1.20.1），Kubernetes 检测到 Node 不可用（NotReady）后，并不会驱离该 Node 上的 Pod，这些 Pod 一直存在于这个 NotReady 的节点上，且 Pod 的状态为 running，而事实上该节点已经不可用了，这个问题在 Kubernetes 的其他一些版本中也存在，这个问题已经成为了 Kubernetes 开发中的一个 issue，具体可以参考 <https://github.com/kubernetes/kubernetes/issues/55713>。

以 v 1.20.1 版本为例，如果在实际使用中出现上述问题，可以通过在 Deployment 的 YAML 中手动写入“容忍度”（tolerations），覆盖默认的“容忍度”来解决这个问题，具体 YAML 内容如下，其中 tolerationSeconds 表示容忍时间，即该 Pod 所在的 Node 出现对应“污点”后，再等待 tolerationSeconds 秒才驱离该 Pod。

```
containers:
- name: nginx
  image: nginx:latest
  imagePullPolicy: IfNotPresent
tolerations:
- effect: NoExecute
```

```
key: node.kubernetes.io/unreachable
operator: Exists
tolerationSeconds: 10
- effect: NoExecute
  key: node.kubernetes.io/not-ready
  operator: Exists
  tolerationSeconds: 10
```

#### 4. 总结

总之，从容器自身的可用性、Pod 可用性以及 Node 可用性的解决方案来考虑，并结合 Kubernetes 的官方建议，在实现容器的高可用时应注意：1. 通过 Deployment 来创建 Pod；2. 指定 Pod 的副本数大于 1；3. 在 Deployment 中手动写入“容忍度”。

## 大数据、云计算工程师、云原生开发人员、 互联网运维人员必读



精选Kubernetes的硬核知识，详解Kubernetes的实战技术  
随书附赠关键示例程序源代码、配置文件、数据文件，以及  
配套的《实践手册》电子书、系列高清课程视频



## 一本书讲透Kubernetes三大核心知识板块

51CTO学院严选讲师

奇虎360云计算联合实验室原技术负责人

○ 艾叔 倾力打造 ○

# 本书知识架构





## 本书附赠资源

关键示例程序源代码 | 配置文件 | 数据文件

gitlab	文件夹	2022/9/16 16...
harbor	文件夹	2022/9/16 16...
jenkins	文件夹	2022/9/16 16...
k8s	文件夹	2022/9/16 16...
keepalived	文件夹	2022/9/16 16...
prometheus	文件夹	2022/9/16 16...
spaceinv	文件夹	2022/9/16 16...
ca	文件夹	2022/9/16 16...
debug	文件夹	2022/9/16 15...
deploy	文件夹	2022/9/16 15...
ha	文件夹	2022/9/16 16...
hpa	文件夹	2022/9/16 15...
ingress	文件夹	2022/9/16 15...
k8dash	文件夹	2022/9/16 15...
pod	文件夹	2022/9/16 15...
pv	文件夹	2022/9/16 15...
rc	文件夹	2022/9/16 15...
svc	文件夹	2022/9/16 15...

## 《实践手册》电子书



Linux、云原生、大数据教程系列

### Kubernetes 快速入门与实战

### 实践手册

#### 目录

前言	3
第 1 章 扩展阅读	错误！未定义书签。
第 2 章 实践	4
2.1 实践 1—定制 VMware 虚拟机	4
2.2 实践 2—最小化安装 CentOS 8	12
2.2.1 下载 CentOS 8 镜像文件	12
2.2.2 关联 CentOS 8 镜像文件	13
2.2.3 安装 CentOS 8	13
2.2.4 配置 CentOS 8	18
2.3 实践 3—远程登录和文件传输	23
2.4 实践 4—远程无密码登录	错误！未定义书签。
2.5 实践 5—Docker 的安装与使用	错误！未定义书签。
2.5.1 安装 Docker	错误！未定义书签。
	23
	31
	35
	40
	42

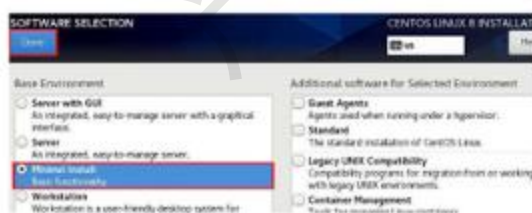


图 2-10 CentOS 8 软件配置界面

注意：此处安装的 CentOS 8 是 Minimal 安装，即最小化安装，只有字符界面和最基本的软件。

点击“Installation Destination”，设置安装对象，如图 2-11 所示。



图 2-11 CentOS 8 安装配置界面

如果虚拟机有多个硬盘，需要选择将 CentOS 8 安装到哪个硬盘上。本书中虚拟机只有一个硬盘，所以只有一个安装对象，在安装对象上要打勾选中即可，然后点击“Done”按钮，如图 2-12 所示。



## 前言

<b>第 1 章 认识 Kubernetes</b> .....	1
1.1 Kubernetes 概述 .....	1
1.1.1 Kubernetes 的定义和背景 .....	1
1.1.2 Kubernetes 与 Docker .....	3
1.1.3 Kubernetes 与云原生 .....	4
1.2 Kubernetes 核心概念 .....	5
1.2.1 resource——Kubernetes 的组成 元素 .....	5
1.2.2 Kubernetes object——定义 Kubernetes 运行状态 .....	10
1.2.3 Pod——实现 Kubernetes 中容器的 逻辑组合 .....	11
1.2.4 RC/RS——控制 Pod 副本个数 .....	12
1.2.5 Deployment——在 Kubernetes 中 部署应用 .....	13
1.2.6 Service——以统一的方式对外 提供服务 .....	13
1.2.7 其他核心概念 .....	14
1.3 Kubernetes 系统架构 .....	15
1.3.1 Control Plane .....	15
1.3.2 Node .....	17
1.3.3 Addons .....	18
1.3.4 kubectl .....	18
1.4 高效学习 Kubernetes .....	19
1.4.1 Kubernetes 快速学习路线图 .....	19
1.4.2 利用本书资源高效学习 Kubernetes (重点必读) .....	19
1.4.3 本书所使用的软件和版本 .....	22
<b>第 2 章 快速构建 Kubernetes 集群</b> .....	23
2.1 Kubernetes 集群规划 .....	23
2.2 准备 Kubernetes 集群节点 .....	23
2.2.1 定制 VMware 虚拟机 (实践 1) .....	
2.2.2 最小化安装 CentOS 8 (实践 2) .....	
2.2.3 远程登录与文件传输 (实践 3) .....	
2.2.4 ssh 远程无密码登录 .....	
2.2.5 Docker 安装与使用 .....	
2.3 kubeadm 安装与系统配置 .....	
2.4 快速构建 Control Plane .....	
2.5 为 Kubernetes 增加 Node 节点 .....	
<b>第 3 章 Kubernetes 核心对象使用</b> .....	
3.1 使用 YAML 创建 Kubernetes resource .....	
3.2 Pod 典型使用 .....	
3.3 RC/RS 基本操作 (实践 4) .....	
3.4 Deployment 典型使用 (实践 5) .....	
3.5 Service 典型使用 (实践 6) .....	
<b>第 4 章 Kubernetes 容器编排实践</b> .....	
4.1 Pod 容器调度 .....	
4.2 Pod 多容器运行 (实践 7) .....	
4.3 Pod 容器数据持久化存储 (PersistentVolume) .....	
4.3.1 安装 NFS .....	
4.3.2 创建 pv 和 pvc .....	
4.3.3 创建 Deployment 使用持久化存	
4.4 Ingress 实现统一访问 Pod 容器 服务 .....	
4.4.1 创建购物网站的 Deployment .....	
4.4.2 创建购物网站的 Service .....	
4.4.3 创建购书网站的 Deployment .....	
4.4.4 创建购书网站的 Service .....	
4.4.5 创建 ingress controller .....	
4.4.6 创建 Ingress .....	



(作者简介)



文艾(艾叔): 解放军理工大学-奇虎360云计算联合实验室原技术负责人, 系统分析师, 51CTO学院严选讲师; 具有多年Linux下的开发、运维和教学经验, 对Linux下的Docker、Kubernetes、Hadoop和Spark等系统有深入研究和丰富的实践经验; 带领团队完成了华为、中兴和奇虎360等公司的多个校企合作Linux相关项目; 指导零基础本科生参加国家级科技创新竞赛和编程大赛, 共获得全国特等奖1次, 一等奖2次, 二等奖2次; 通过“艾叔编程”公众号和网易云课堂开设了一系列Linux相关的免费课程, 已帮助8万多名学习者入门编程并深受好评。



## 前言

<b>第1章 认识 Kubernetes</b> .....	1	2.2.1 定制 VMware 虚拟机 (实践 1)	
1.1 Kubernetes 概述 .....	1	2.2.2 最小化安装 CentOS 8 (实践 2)	
1.1.1 Kubernetes 的定义和背景 .....	1	2.2.3 远程登录与文件传输 (实践 3)	
1.1.2 Kubernetes 与 Docker .....	3	2.2.4 ssh 远程无密码登录 .....	
1.1.3 Kubernetes 与云原生 .....	4	2.2.5 Docker 安装与使用 .....	
1.2 Kubernetes 核心概念 .....	5	2.3 kubectl 安装与系统配置 .....	
1.2.1 resource——Kubernetes 的组成		2.4 快速构建 Control Plane .....	
元素 .....	5	2.5 为 Kubernetes 增加 Node 节点 .....	
1.2.2 Kubernetes object——定义		<b>第3章 Kubernetes 核心对象使用</b> .....	
Kubernetes 运行状态 .....	10	3.1 使用 YAML 创建 Kubernetes	
1.2.3 Pod——实现 Kubernetes 中容器的		resource .....	
逻辑组合 .....	11	3.2 Pod 典型使用 .....	
1.2.4 RC/RS——控制 Pod 副本个数 .....	12	3.3 RC/RS 基本操作 (实践 4) .....	
1.2.5 Deployment——在 Kubernetes 中		3.4 Deployment 典型使用 (实践 5) .....	
部署应用 .....	13	3.5 Service 典型使用 (实践 6) .....	
1.2.6 Service——以统一的方式对外		<b>第4章 Kubernetes 容器编排实践</b> .....	
提供服务 .....	13	4.1 Pod 容器调度 .....	
1.2.7 其他核心概念 .....	14	4.2 Pod 多容器运行 (实践 7) .....	
1.3 Kubernetes 系统架构 .....	15	4.3 Pod 容器数据持久化存储	
1.3.1 Control Plane .....	15	(PersistentVolume) .....	
1.3.2 Node .....	17	4.3.1 安装 NFS .....	
1.3.3 Addons .....	18	4.3.2 创建 pv 和 pvc .....	
1.3.4 kubectl .....	18	4.3.3 创建 Deployment 使用持久化存储 .....	
1.4 高效学习 Kubernetes .....	19	4.4 Ingress 实现统一访问 Pod 容器	
1.4.1 Kubernetes 快速学习路线图 .....	19	服务 .....	
1.4.2 利用本书资源高效学习 Kubernetes		4.4.1 创建购物网站的 Deployment .....	
(重点必读) .....	19	4.4.2 创建购物网站的 Service .....	
1.4.3 本书所使用的软件和版本 .....	22	4.4.3 创建购书网站的 Deployment .....	
<b>第2章 快速构建 Kubernetes 集群</b> .....	23	4.4.4 创建购书网站的 Service .....	
2.1 Kubernetes 集群规划 .....	23	4.4.5 创建 ingress controller .....	
2.2 准备 Kubernetes 集群节点 .....	23	4.4.6 创建 Ingress .....	

4.4.7 按路径统一访问 Pod 容器的服务	70	6.3 构建基于 Keepalived 的 Kubernetes 高可用集群	110
4.5 Pod 容器自动伸缩 (HPA)	71	6.3.1 配置 Keepalived	110
4.5.1 编写 HPA YAML 文件	71	6.3.2 构建 Control Plane	111
4.5.2 创建监控对象和 HPA	73	6.3.3 构建 Node 节点	114
4.5.3 HPA 伸缩算法	74	6.3.4 Kubernetes 高可用性测试	115
4.5.4 HPA 自动伸缩测试	74	第 7 章 Kubernetes 监控与告警 (Prometheus+Grafana)	119
第 5 章 Kubernetes 系统运维与故障 处理	78	7.1 Kubernetes 系统组件指标 (Metrics)	119
5.1 Pod 容器的高可用实践 (实践 8)	78	7.2 Prometheus 监控 Kubernetes	125
5.2 Kubernetes 节点性能数据采集	78	7.2.1 Prometheus 架构和核心概念	125
5.3 使用 k8dash 快速监控 Kubernetes	80	7.2.2 Prometheus 快速部署 (kube-prometheus)	127
5.4 Kubernetes 系统运维常用操作	83	7.2.3 Prometheus 监控机制与配置	131
5.4.1 增加 kubectl 节点	83	7.2.4 Prometheus 监控 Kubernetes 核心组件	140
5.4.2 停止 Kubernetes 组件 Pod 中的 容器	84	7.2.5 Prometheus 监控 Kubernetes 指定对象 (Exporter)	145
5.4.3 重置 Kubernetes 集群节点	85	7.3 Grafana 展示 Kubernetes 监控 数据	152
5.4.4 查看和设置 Kubernetes 组件的启动 参数	85	7.3.1 Grafana 快速访问	152
5.4.5 运行 Pod 容器命令	88	7.3.2 Grafana 展示 Prometheus 数据源 数据 (Kubernetes)	154
5.4.6 查看 Pod 容器网卡名	89	7.3.3 Grafana 展示其他数据源的数据	160
5.4.7 复制文件到 Pod 容器	89	7.3.4 Grafana 配置的持久化存储	164
5.4.8 查看指定进程监听的端口	90	7.4 Kubernetes 监控告警	171
5.5 查看 Kubernetes 日志	90	7.4.1 Prometheus 告警机制	171
5.5.1 系统日志	90	7.4.2 查看 Prometheus 告警	171
5.5.2 Kubernetes 组件日志	91	7.4.3 Prometheus 告警规则 (Rule)	175
5.5.3 Pod 启动信息和容器日志	91	7.4.4 配置 Prometheus 告警发送邮件	182
5.6 Kubernetes 故障处理	92	7.4.5 Grafana 告警配置与邮件通知	188
5.6.1 处理故障 Pod	92	第 8 章 基于 Kubernetes 的 CI/CD 项目综合实践 (GitLab+ Harbor+Jenkins)	198
5.6.2 容器故障调试	94	8.1 CI/CD 核心概念与基础	198
第 6 章 构建 Kubernetes 高可用集群	96	8.2 太空入侵者游戏 CI/CD 方案 设计	199
6.1 Kubernetes 高可用集群的架构与 规划	96	8.2.1 系统架构与集群规划	199
6.2 构建高可用负载均衡器 (Keepalived+LVS)	98		
6.2.1 构建 LB 节点	99		
6.2.2 构建 RS 节点	102		
6.2.3 构建 Client 节点	105		
6.2.4 测试 LB + HA	106		

8.2.2 CI/CD 开发流程.....	200	8.4 构建基于 Kubernetes 的太空入侵者 游戏生产环境 .....	226
8.3 构建太空入侵者游戏开发与测试 环境.....	201	8.5 实现太空入侵者游戏 CI/CD.....	226
8.3.1 构建承载和测试节点——devt 虚拟机 .....	201	8.5.1 Webhook 实现 git 提交触发 .....	226
8.3.2 构建开发节点——spaceinv 容器 ..	202	8.5.2 自动构建镜像和测试 (Jenkins+GitLab+Harbor) .....	233
8.3.3 构建代码管理仓库——GitLab.....	205	8.5.3 Jenkins 自动部署容器化应用到 Kubernetes.....	242
8.3.4 构建容器镜像仓库——Harbor.....	217	8.5.4 CI/CD 综合测试 .....	245
8.3.5 构建持续集成工具——Jenkins .....	223		



## 7.2 Prometheus 监控 Kubernetes

Prometheus 是一个开源的监控和告警系统，它会收集和存储监控对象（包括但不限于 Kubernetes）的指标数据，打上时间戳构成时序数据，并在达到条件时触发告警。

Kubernetes 和 Prometheus 配合非常紧密，首先 Kubernetes 核心组件的指标数据是 Prometheus 格式，可直接接入 Prometheus；其次 Prometheus 不仅能监控 Kubernetes 核心组件的运行情况，还能监控 Kubernetes 所管理的 Pod 和容器运行情况，并能有效告警；此外，Prometheus 还支持通过服务发现来找到监控对象，这种方式相对静态配置监控对象的方式来说更加灵活，更适用于 Kubernetes 的应用场景。

CNCF 对 Prometheus 非常重视，Prometheus 自 2016 年加入 CNCF 后，于 2018 年就成为继 Kubernetes 之后，第二个从 CNCF 毕业的项目。

### 7.2.1 Prometheus 架构和核心概念

本节介绍 Prometheus 的架构和核心概念，它们将为后续深入学习 Prometheus，掌握 Prometheus 的使用打下基础。

#### 1. Prometheus 架构

Prometheus 架构的核心组件由图 7-2（本图引自：Architecture [EB/OL].[2022-3-31]. <https://prometheus.io/docs/introduction/overview/>）的蓝色模块所示，包括：Prometheus server、Alertmanager、Prometheus web UI 和 Pushgateway，具体说明如下。

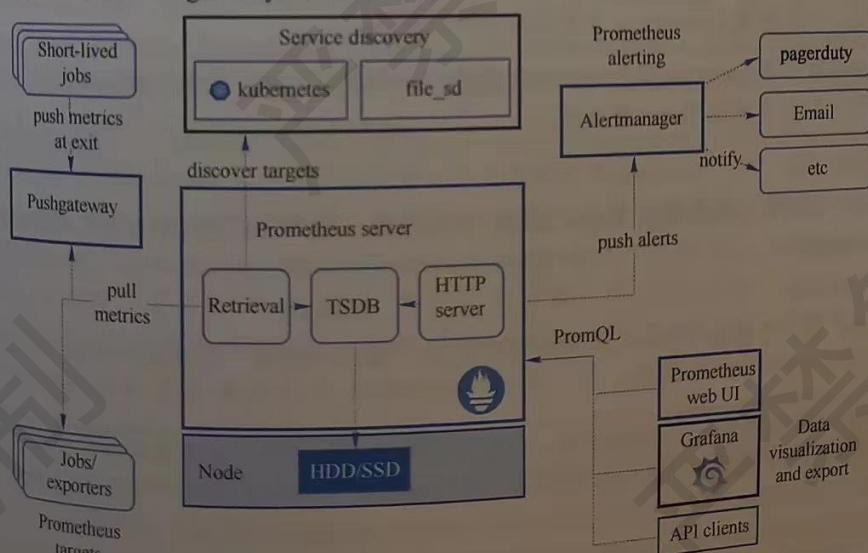


图 7-2 Prometheus 监控架构图

(1) Prometheus server  
负责监控对象的发现、监控指标数据的采集和存储、告警信息的推送、并提供 HTTP 服务供数据可视化组件查询。