

# 目录

Introduction	1.1
<b>A 部署 Kubernetes</b>	<b>1.2</b>
A.1 准备部署 Kubernetes 集群	1.2.1
A.1.1 部署目标	1.2.1.1
A.1.2 系统环境及部署准备	1.2.1.2
A.2 部署 Kubernetes 集群	1.2.2
A.2.1 设定容器运行环境	1.2.2.1
A.2.2 设定 kubernetes 集群节点	1.2.2.2
A.2.3 集群初始化	1.2.2.3
A.2.4 设定 kubectl 的配置文件	1.2.2.4
A.2.5 部署网络插件	1.2.2.5
A.2.6 添加 Node 到集群中	1.2.2.6
A.2.7 获取集群状态信息	1.2.2.7
A.3 从集群中移除节点	1.2.3
A.4 重新生成用于节点加入集群的认证命令	1.2.4
<b>B 部署 GlusterFS 及 Heketi</b>	<b>1.3</b>
B.1 部署 GlusterFS 集群	1.3.1
B.2 部署 Heketi	1.3.2
B.2.1 安装并启动 Heketi 服务器	1.3.2.1
B.2.2 设置 Heketi 系统拓扑	1.3.2.2
<b>第二章 kubernetes 快速入门</b>	<b>1.4</b>
2.1 Kubernetes 的核心对象	1.4.1
2.1.1 Pod 资源对象	1.4.1.1
2.1.2 Controller	1.4.1.2
2.1.3 Service	1.4.1.3
2.1.4 部署应用程序的主体过程	1.4.1.4
2.2 部署 Kubernetes 集群	1.4.2
2.2.1 kubeadm 部署工具	1.4.2.1
2.2.2 集群运行模式	1.4.2.2
2.2.3 准备用于实践操作的集群环境	1.4.2.3
2.2.4 获取集群环境相关的信息	1.4.2.4
2.3 kubectl 使用基础与示例	1.4.3
2.4 命令式容器应用编排	1.4.4
2.4.1 部署应用 (Pod)	1.4.4.1
2.4.2 探查 Pod 及应用详情	1.4.4.2
2.4.3 部署 Service 对象	1.4.4.3

## A.1.1 部署目标

2.4.4 扩容和缩容	1.4.4.4
2.4.5 修改及删除对象	1.4.4.5
<b>第三章 资源管理基础</b>	<b>1.5</b>
3.1 资源对象及API群组	1.5.1
3.1.1 Kubernetes 的资源对象	1.5.1.1
3.1.2 资源及其在API中的组织形式	1.5.1.2
3.1.3 访问kubernetes REST API	1.5.1.3
3.2 对象类资源格式	1.5.2
3.2.1 资源配置清单	1.5.2.1
3.2.2 metadata 嵌套字段	1.5.2.2
3.2.3 spec 和 status 字段	1.5.2.3
3.2.4 资源配置清单格式文档	1.5.2.4
3.2.5 资源对象管理方式	1.5.2.5
3.3 kubectl 命令与资源管理	1.5.3
3.3.1 资源管理操作概述	1.5.3.1
3.3.2 kubectl 的基本用法	1.5.3.2
3.4 管理名称空间资源	1.5.4
3.4.1 查看名称空间及其资源对象	1.5.4.1
3.4.2 管理 Namespace 资源	1.5.4.2
3.5 Pod 资源的基础管理操作	1.5.5
3.5.1 陈述式对象配置管理方式	1.5.5.1
3.5.2 声明式对象配置管理方式	1.5.5.2
<b>第四章 管理 Pod 资源对象</b>	<b>1.6</b>
4.1 容器与 Pod 资源对象	1.6.1
4.2 管理 Pod 对象的容器	1.6.2
4.2.1 镜像及其获取策略	1.6.2.1
4.2.2 暴露端口	1.6.2.2
4.2.3 自定义运行的容器化应用	1.6.2.3
4.2.4 环境变量	1.6.2.4
4.2.5 共享节点的网络名称空间	1.6.2.5
4.2.6 设置 Pod 对象的安全上下文	1.6.2.6
4.3 标签与标签选择器	1.6.3
4.3.1 标签概述	1.6.3.1
4.3.2 管理资源标签	1.6.3.2
4.3.3 标签选择器	1.6.3.3
4.3.4 Pod 节点选择器 nodeSelector	1.6.3.4
4.4 资源注解	1.6.4
4.4.1 查看资源注解	1.6.4.1

## A.1.1 部署目标

4.4.2 管理资源注解	1.6.4.2
<b>4.5 Pod 对象的生命周期</b>	<b>1.6.5</b>
4.5.1 Pod 的相位	1.6.5.1
4.5.2 Pod 的创建过程	1.6.5.2
4.5.3 Pod 生命周期中的重要行为	1.6.5.3
4.5.4 容器的重启策略	1.6.5.4
4.5.5 Pod 的终止过程	1.6.5.5
<b>4.6 Pod 存活性探测</b>	<b>1.6.6</b>
4.6.1 设置 exec 探针	1.6.6.1
4.6.2 设置 HTTP 探针	1.6.6.2
4.6.3 设置 TCP 探针	1.6.6.3
4.6.4 存活性探测行为属性	1.6.6.4
<b>4.7 Pod 就绪性探测</b>	<b>1.6.7</b>
<b>4.8 资源需求及资源限制</b>	<b>1.6.8</b>
4.8.1 资源需求	1.6.8.1
4.8.2 资源限制	1.6.8.2
4.8.3 容器的可见资源	1.6.8.3
4.8.4 Pod 的服务质量类比	1.6.8.4
<b>第五章 Pod 控制器</b>	<b>1.7</b>
<b>5.1 关于 Pod 控制器</b>	<b>1.7.1</b>
5.1.1 Pod 控制器概述	1.7.1.1
5.1.2 控制器与 Pod 对象	1.7.1.2
5.1.3 Pod 模版资源	1.7.1.3
<b>5.2 ReplicaSet 控制器</b>	<b>1.7.2</b>
5.2.1 ReplicaSet 概述	1.7.2.1
5.2.2 创建 ReplicaSet	1.7.2.2
5.2.3 ReplicaSet 管控下的 Pod 对象	1.7.2.3
5.2.4 更新 ReplicaSet 控制器	1.7.2.4
5.2.5 删除 ReplicaSet 控制器资源	1.7.2.5
<b>5.3 Deployment 控制器</b>	<b>1.7.3</b>
5.3.1 创建 Deployment	1.7.3.1
5.3.2 更新策略	1.7.3.2
5.3.3 升级 Deployment	1.7.3.3
5.3.4 金丝雀发布	1.7.3.4
5.3.5 回滚 Deployment 控制器下的应用发布	1.7.3.5
5.3.6 扩容和缩容	1.7.3.6
<b>5.4 DaemonSet 控制器</b>	<b>1.7.4</b>
5.4.1 创建 DaemonSet 资源对象	1.7.4.1

5.4.2 更新DaemonSet对象	1.7.4.2
5.5 Job控制器	1.7.5
5.5.1 创建Job对象	1.7.5.1
5.5.2 并形式Job	1.7.5.2
5.5.3 Job扩容	1.7.5.3
5.5.4 删除Job	1.7.5.4
5.6 CronJob控制器	1.7.6
5.6.1 创建CronJob对象	1.7.6.1
5.6.2 CronJob的控制机制	1.7.6.2
5.7 ReplicationController	1.7.7
5.8 Pod中断预算	1.7.8
5.9 本章小结	1.7.9
第六章 Service 和 Ingress	1.8
6.1 Service资源及其实现模型	1.8.1
6.1.1 Service资源概述	1.8.1.1
6.1.2 虚拟IP和服务代理	1.8.1.2
6.2 Service资源的基础应用	1.8.2
6.2.1 创建Service资源	1.8.2.1
6.2.2 向Service对象请求服务	1.8.2.2
6.2.3 Service会话粘性	1.8.2.3
6.3 服务发现	1.8.3
6.3.1 服务发现概述	1.8.3.1
6.3.2 服务发现方式：环境变量	1.8.3.2
6.3.3 ClusterDNS和服务发现	1.8.3.3
6.3.4 服务发现方式：DNS	1.8.3.4
6.4 服务暴露	1.8.4
6.4.1 Service类型	1.8.4.1
6.4.2 NodePort类型的Service资源	1.8.4.2
6.4.3 LoadBalancer类型的Service资源	1.8.4.3
6.4.4 ExternalName Service	1.8.4.4
6.5 Headless类型的Service资源	1.8.5
6.5.1 创建Headless Service资源	1.8.5.1
6.5.2 Pod资源发现	1.8.5.2
6.6 Ingress资源	1.8.6
6.6.1 Ingress和Ingress Controller	1.8.6.1
6.6.2 创建Ingress资源	1.8.6.2
6.6.3 Ingress资源类型	1.8.6.3
6.6.4 部署Ingress控制器（Nginx）	1.8.6.4

## A.1.1 部署目标

6.7 案例：使用Ingress发布tomcat	1.8.7
6.7.1 准备名称空间	1.8.7.1
6.7.2 部署tomcat实例	1.8.7.2
6.7.3 创建Service资源	1.8.7.3
6.7.4 创建Ingress资源	1.8.7.4
6.7.5 配置TLS Ingress资源	1.8.7.5
6.8 本章小结	1.8.8

### A.1.1 部署目标



build passing

## A.1.1 部署目标

- A.1 准备部署 Kubernetes 集群
  - A.1.1 部署目标
  - A.1.2 系统环境及部署准备
- A.2 部署 Kubernetes 集群
  - A.2.1 设定容器运行环境
  - A.2.2 设定 kubernetes 集群节点
  - A.2.3 集群初始化
  - A.2.4 设定 kubectl 的配置文件
  - A.2.5 部署网络插件
  - A.2.6 添加 Node 到集群中
  - A.2.7 获取集群状态信息
- A.3 从集群中移除节点
- A.4 重新生成用于节点加入集群的认证命令

## A.1 准备部署 Kubernetes 集群

Kubernetes 项目目前仍处于快速迭代阶段，演示过程中使用的配置对于其后续版本可能存在某些变动，因此，版本不同时，对具体特性支持的变动请参考 Kubernetes 的 ChangeLog 或其他相关文档中的说明。

## A.1.1 部署目标

图 A-1 给出了本节要部署的目标集群的基本环境，它拥有一个 Master 主机和三个 Node 主机。各 Node 主机的配置方式基本相同。

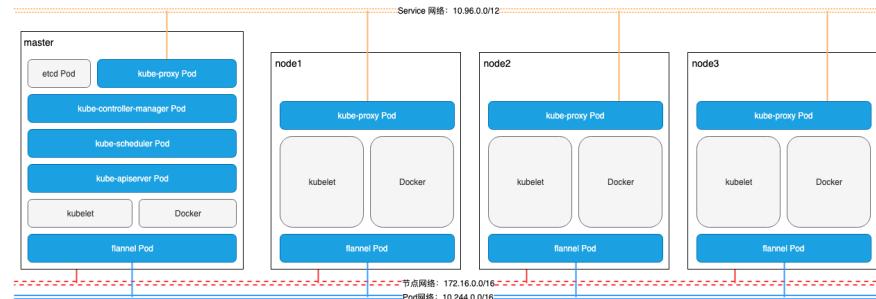


图 1.2.1.1 - Kubernetes 集群部署目标示意图

各个主机上采用的容器运行时环境为 Docker，为 Pod 提供网络功能的 CNI 是 flannel，它运行为托管于 kubernetes 之上的 Pod 对象，另外，基础附件还包括 KubeDNS（或 CoreDNS）用于名称解析或服务发现。

## A.1.2 系统环境及部署准备

如前所述，Kubernetes 当前仍处于快速迭代的周期中，其版本变化频繁、跨版本的特性变化较大，为了帮助读者确认各配置和功能的可用性，本书使用如下基础环境。

### 1. 各相关组件及主机环境

操作系统、容器引擎、etcd 及 kubernetes 的相关版本分别如下：

- OS: CentOS 7.4 x86\_64
- Container runtime: Docker 18.09 ce
- Kubernetes: 1.14.0

各主机角色分配及 IP 地址如表 A-1所示。

IP 地址	主机名	角色
172.16.0.6	master, master.renkeju.mobi	master
172.16.0.7	node01, node01.renkeju.mobi	node
172.16.0.8	node02, node02.renkeju.mobi	node
172.16.0.9	node03, node03.renkeju.mobi	node

### 2. 基础环境设置

Kubernetes 的正确运行依赖于一些基础环境的设定，如各节点时间通过网络时间服务保持同步和主机名称解析等，集群规模较大的实践场景中，主机名称解析通常由 DNS 服务器完成。本测试示例中，时间同步服务直接基于系统的默认配置从互联网的时间服务中获取，主机名称解析则由 hosts 文件进行。

#### (1) 主机名称解析

分布式系统环境中的多主机通信通常基于主机名称进行，这在 IP 地址存在变化的可能性时为主机提供了固定的访问入口，因此一般需要由专用的 DNS 服务负责解决各节点主机名。不过，考虑到此处部署的是测试集群，因此为了降低系统复杂度，这里将采用基于 hosts 的文件进行主机名称解析。编辑 Master 和各 Node 上的 /etc/hosts 文件，确保其内容如下：

```
172.16.0.6 master.renkeju.mobi master
172.16.0.7 node01.renkeju.mobi node01
172.16.0.8 node02.renkeju.mobi node02
172.16.0.9 node03.renkeju.mobi node03
```

#### (2) 主机时间同步

## A.1.1 部署目标

如果各主机可直接访问互联网，则直接启动各主机上的 chronyd 服务即可。否则徐奥使用本地网络中的时间服务器，例如，可以将 Master 配置为 chronyd server，而后其他节点均从 Master 同步时间：

```
**[terminal]
**[path ~]***[delimiter # ]**[command systemctl start chronyd.service]
**[path ~]***[delimiter # ]**[command systemctl enable chronyd.service]
```

## (3) 关闭防火墙服务

各 Node 运行的 kube-proxy 组件均要借助 iptables 或 ipvs 构建 Service 资源对象，该资源对象是 Kubernetes 的核心资源之一。出于简化问题复杂度之需，这里需要事先关闭所有主机之上的 iptables 或 firewalld 服务。

```
**[terminal]
**[path ~]***[delimiter # ]**[command systemctl stop firewalld.service iptab
**[path ~]***[delimiter # ]**[command systemctl disable firewalld.service]
**[path ~]***[delimiter # ]**[command systemctl disable ipatbles.service]
```

## (4) 关闭并禁用 SELinux

若当前启用了 SELinux，则需要临时设置其当前状态为 permissive：

```
**[terminal]
**[path ~]***[delimiter # ]**[command setenforce 0]
```

另外，编辑 `/etc/sysconfig/selinux` 文件，以彻底禁用 SELinux：

```
**[terminal]
**[path ~]***[delimiter # ]**[command sed -i 's@^\\(SELINUX=\\).*@\\1disabled@'
```

## (5) 禁用 Swap 设备（可选步骤）

kubeadm 默认会预先检查当前主机是否禁用了 Swap 设备，并在未禁用时强制终止部署过程。因此，在主机内存资源充裕的条件下，需要分两步完成。首先是关闭当前已启用的所有 Swap 设备：

```
**[terminal]
**[path ~]***[delimiter # ]**[command swapoff -a]
```

而后编辑 `/etc/fstab` 配置文件，注释用于挂载 Swap 设备的所有行。不同系统环境默认启用的 Swap 设备是不尽相同的，请读者根据实际情况完成相应操作。另外，部署时也可以选不禁用 Swap，而是通过后文的 kubeadm init 及 kubeadm join 命令执行时额外使用相关的选项忽略检查错误。

## (6) 启用 ipvs 内核模块（可选步骤）

Kubernetes 1.11 之后的版本默认支持使用 ipvs 代理模式的 Service 资源，但它依赖于 ipvs 相关的内核模块，而这些模块默认不会自动载入。因此，这里选择创建载入内核模块相关的脚本文件 `/etc/sysconfig/modules/ipvs.modules`，设定于系统引导时自动载入的 ipvs 相关的内核模块，以支持使用 ipvs 代理模式的 Service 资源。文件内容如下：

```
#!/bin/bash
ipvs_mods_dir="/usr/lib/modules/$(uname -r)/kernel/net/netfilter/ipvs"
for i in $(ls $ipvs_mods_dir | grep -o "^[^.]*"); do
    /sbin/modinfo -F filename $i &> /dev/null
    if [ $? -eq 0 ]; then
        /sbin/modprobe $i
    fi
done
```

而后修改文件权限，并手动为当前系统环境加载内核模块：

```
**[terminal]
**[path ~]**[delimiter #]**[command chmod +x /etc/sysconfig/modules/ipvs.modules]
**[path ~]**[delimiter #]**[command /etc/sysconfig/modules/ipvs.modules]
```

不过，ipvs 仅负责实现负载均衡相关的任务，它无法完成 kube-proxy 中的包过滤及 SNAT 等功能，这些仍需要由 iptables 实现。另外，对于初学者来说，前期的测试并非必然要用到 ipvs 代理模式。部署时可以省略此步骤。

## A.2 部署 Kubernetes 集群

kubeadm 是用于快速构建 Kubernetes 集群的工具，随着 Kubernetes 的发行版本而提供，使用它构建集群时，大致可分为如下几步：

1. 在 Master 及各 Node 安装 Docker、kubelet 及 kubeadm。并以系统守护进程的方式启动 Docker 和 kubelet 服务。
2. 在 Master 节点上通过 kubeadm init 命令进行集群初始化。
3. 各 Node 通过 kubeadm join 命令加入初始化完成的集群中。
4. 在集群上部署网络附件，如 flannel 或 Calico 等以提供 Service 网络及 Pod 网络。

为了简化部署过程，kubeadm 使用一组固定的目录及文件路径存储相关的配置及数据文件，其中 `/etc/kubernetes` 目录使所有文件或目录的统一存储目录。它使用 `/etc/kubernetes/manifests` 目录存储各静态 Pod 资源的配置清单，用到的文件有 `etcd.yaml`、`kube-apiserver.yaml`、`kube-controller-manager.yaml` 和 `kube-scheduler.yaml` 四个，它们的作用基本能够见文件，如 `kubelet.conf`、`controller-manager.conf`、`scheduler.conf` 和 `admin.conf` 等，它们分别为相关的组件提供接入 API Server 的认证信息等。此外，它还会在 `/etc/kubernetes/pki` 目录中存储若干私钥和证书文件。

## A.2.1 设定容器运行环境

Kubernetes 支持多种容器运行时环境，例如 Docker、RKT 和 Frakti 等，本书将采用其中目前最为流行的、接受程度最为广泛的 Docker，它的常用部署方式有两种，具体如下。

- 由系统发行版的程序包仓库提供，如 CentOS 7 Extras 仓库中的 Docker。
- Docker 官方仓库中的程序包，以 CentOS 7 为例，它通常能够提供较 Extras 仓库中更新版本的程序包，获取地址为 <https://download.docker.com/>。

本文采用的是第二种方式，不过，Kubernetes 认证的 Docker 版本通常忽略低于其最新版本，因此生产环境部署时应该尽可能部署经过认证的版本。考虑到 Kubernetes 版本迭代周期较短，它对 Docker 版本的支持也会快速变化，因此本示例讲直接使用 Docker 仓库中的最新版本。部署时，Docker 需要安装于 Master 及各 Node 主机之上，安装方式相同，其步骤如下：

```
**[terminal]
**[path ~]***[delimiter # ]**[command wget https://download.docker.com/linux,
**[path ~]***[delimiter # ]**[command yum install docker-ce]
```

kubeadm 构建集群的过程需要到 gcr.io 中获取 Docker 镜像，因此必须确保 Docker 主机能够正常访问到此站点，否则，就得配置 Docker 以代理的方式访问 gcr.io，或者配置 kubeadm 从其他 Registry 获取相关的镜像。代理的方法是在 [service] 配置段中添加类似如下格式的配置项：Environment="HTTP\_PROXY=http://IP:PORT" 或 Environment="HTTPS\_PROXY=https://IP:PORT"。

另外，Docker 自 1.13 版起会自动设置 iptables 的 FORWARD 默认策略为 DROP，这可能会影响 Kubernetes 集群依赖的报文转发功能，因此，需要在 docker 服务启动后，重新将 FORWARD 链的默认策略设置为 ACCEPT，方式是修改 /usr/lib/systemd/system/docker.service 文件，在 “ExecStart=/usr/bin/dockerd” 一行之后新增一行如下内容：

```
ExecStartPost=/usr/sbin/iptables -P FORWARD ACCEPT
```

上面各步骤设置完成后即可启动 docker 服务，并将其设置为随系统引导而自动启用，相关命令如下：

```
**[terminal]
**[path ~]***[delimiter # ]**[command systemctl daemon-reload]
**[path ~]***[delimiter # ]**[command systemctl start docker.service]
**[path ~]***[delimiter # ]**[command systemctl enable docker.service]
```

国内访问 DockerHub 下载镜像的速度较缓慢，建议使用国内的镜像对其进行加速，如 <https://registry.docker-cn.com>，另外，中国科技大学也提供了公共可用的镜像加速服务，其 URL 为 <https://docker.mirrors.ustc.edu.cn>，将其定义在 daemon.json 中重启 Docker 即可使用。

## 设置 Docker daemon 配置文件

Docker 默认的 cgroup 驱动程序为“cgroupfs”，kubernetes 推荐的驱动程序是“systemd”。所以我们可以通过修改 `/etc/docker/daemon.json` 来修改 Docker cgroup 驱动程序。

```
cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opt": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ]
}
EOF
```

## CRI-O

本节包含将CRI-O安装为CRI运行时的必要步骤。

使用以下命令在系统上安装CRI-O：

- 先决条件

```
modprobe overlay
modprobe br_netfilter

# Setup required sysctl params, these persist across reboots.
cat > /etc/sysctl.d/99-kubernetes-cri.conf <<EOF
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

sysctl --system
```

```
**[terminal]
**[path ~]**[delimiter #]**[command yum-config-manager --add-repo=https://crio.io/repos/mainline]
**[path ~]**[delimiter #]**[command yum install --nogpgcheck cri-o]
**[path ~]**[delimiter #]**[command systemctl start crio]
```

## A.2.2 设定 kubernetes 集群节点

kubelet 使运行于集群中每个节点之上的 Kubernetes 代理程序，它的核心功能在于通过 API Server 获取调度至自身运行的 Pod 资源的 PodSpec 并依之运行 Pod 对象。事实上，以自托管的方式部署的 Kubernetes 集群，除了 kubelet 和 Docker 之外的所有组件均以 Pod 对象的形式运行。

### 1. 安装 kubelet 及 kubeadm

安装 kubelet 的常用方式包含如下几种。快速迭代期内，Linux 发行商提供的安装包通常版本较低，因此建议采用下列方式的第一种或第二种，本章采用第二种方式。

- Kubernetes 提供的二进制格式的 tar 包。
- Google 的 yum 仓库中提供的 rpm 包，可通过国内的镜像站点获取，例如阿里云镜像站。
- OS 发行商提供的安装包，例如 CentOS 7 Extras 仓库中的 Kubernetes 相关程序包。

对于 rpm 方式的安装来说，kubelet、kubeadm 和 kubectl 等是各自独立的程序包，Master 及各 Node 至少应该安装 kubelet 和 kubeadm，而 kubectl 则只需要安装于客户端主机即可，不过，由于依赖关系它通常也会自动安装。另外，Google 提供的 kubelet rpm 包的 yum 仓库托管于 Google 站点的服务器主机之上，目前访问起来略有不便。幸运的是，目前国内阿里云等镜像 (<https://opsx.alibaba.com/mirror>) 对此项目也有镜像提供。

首先设定用于安装 kubelet、kubeadm 和 kubectl 等组件的 yum 仓库，编辑配置文件 `/etc/yum.repos.d/kubernetes.repo`，内容如下：

```
[kubernetes]
name=Kubernetes
baseurl=https://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-el7-x86_64/
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg https://mirro
```

而后执行如下命令即可安装相关的程序包：

```
**[terminal]
[**[prompt root@master]**[path ~]**[delimiter #]**[command yum install kubelet]
```

### 2. 配置 kubelet

### A.1.1 部署目标

Kubernetes 自 1.8 版本起强制要求关闭系统的交换分区（Swap），否则 kubelet 将无法启动。当然，用户也可以通过将 kubelet 的启动参数 “--fail-swap-on” 设置为 “false” 忽略此限制，尤其使系统上运行有其他重要进程且系统内存资源稍显不足时建议保留交换分区。

编辑 kubelet 的配置文件 `/etc/sysconfig/kubelet`，设置其配置参数如下，以忽略禁止使用 Swap 的限制：

```
KUBELET_EXTRA_ARGS="--fail-swap-on=false"
```

待配置文件的修改完成后，需要设定 kubelet 服务开机自动启动，这也是 kubeadm 的强制要求：

```
**[terminal]
[**[prompt root@master]**[path ~]**[delimiter #]**[command systemctl enable
```

## A.2.3 集群初始化

一旦 Master 和各个 Node 的 Docker 及 kubelet 设置完成后，接着便可以在 Master 节点上执行 `kubeadm init` 命令进行集群初始化。`kubeadm init` 命令支持两种初始化方式，一是通过命令行选项传递关键的参数设定，另一个是基于 yaml 格式的专用配置文件设定更详细的配置参数。下面分别给出了两种实现方式的配置步骤，建议读者采用第二种方式。

### Master 初始化方式一

运行下面的命令，便可完成 Master 的初始化：

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter # ]**[command kubeadm init \
--kubernetes-version=v1.14.0 \
--pod-network-cidr=10.244.0.0/16 \
--service-cidr=10.96.0.0/12 \
--apiserver-advertise-address=0.0.0.0 \
--ignore-preflight-errors=Swap]
```

上面命令的选项及参数设定决定了集群运行环境的众多特性设定，这些设定对于此后在集群中部署运行应用程序至关重要。

- `--kubernetes-version`

正在使用的 Kubernetes 程序组件的版本号，需要与 kubelet 的版本号相同。

- `--pod-network-cidr`

Pod 网络的地址范围，其值为 CIDR 格式的网络地址；使用 flannel 网络插件时，其默认地址为 10.244.0.0/16。

- `--service-cidr`

Service 的网络地址范围，其值为 CIDR 格式的网络地址，默认地址为 10.96.0.0/12。

- `--apiserver-advertise-address`

API server 通告给其他组件的 IP 地址，一般应该为 Master 节点的 IP 地址，0.0.0.0 表示节点上所有可用的地址。

- `--ignore-preflight-errors`

忽略哪些运行时的错误信息，其值为 Swap 时，表示忽略因 Swap 未关闭而导致的错误。

更多的参数请参考 `kubeadm` 的文档，链接地址为  
<https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init/>。

### Master 初始化方式二

### A.1.1 部署目标

kubeadm init 也可通过配置文件加载配置，以定制更丰富的部署选项。以下是符合前述命令设定方式的使用示例，不过，它明确定义了 kubeProxy 的模式为 ipvs，并支持通过修改 imageRepository 的值来修改获取系统镜像时使用的镜像仓库。

```
apiVersion: kubeadm.k8s.io/v1beta1
bootstrapTokens:
- groups:
  - system:bootstrappers:kubeadm:default-node-token
  token: abcdef.0123456789abcdef
  ttl: 24h0m0s
  usages:
  - signing
  - authentication
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: 172.16.0.6
  bindPort: 6443
nodeRegistration:
  criSocket: /var/run/dockershim.sock
  name: master.renkeju.com
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
---
apiServer:
  timeoutForControlPlane: 4m0s
apiVersion: kubeadm.k8s.io/v1beta1
certificatesDir: /etc/kubernetes/pki
clusterName: kubernetes
controlPlaneEndpoint: ""
controllerManager: {}
dns:
  type: CoreDNS
etcd:
  local:
    dataDir: /var/lib/etcd
  imageRepository: reg.bih.cn/gcr
kind: ClusterConfiguration
kubernetesVersion: v1.14.0
networking:
  dnsDomain: cluster.local
  podSubnet: 10.244.0.0/16
  serviceSubnet: 10.96.0.0/12
scheduler: {}
```

将上面的内容保存于配置文件中，如 kubeadm-config.yaml，而后执行相应的命令即可完成 Master 初始化：

## A.1.1 部署目标

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter # ]**[command kubeadm init --c
```

无论使用上述哪种方法，命令的执行过程都会执行众多部署操作并生成相关的信息，如生成配置文件，标示主节点、生成 bootstrap token 及部署核心附加组件 kube-proxy 和 kube-dns 等，尤其是为集群通信安全于 `/etc/kubernetes/pki` 目录中生成的一众密钥和数字证书，并在最后给出了 kubectl 客户端工具的配置文件生成方式以及随后将 Node 加入集群时使用的引导认证令牌（bootstrap token）等，后续需要加入集群的各个 Node 都将使用该引导认证令牌加入集群。不同版本的 kubeadm 其输出结果或许略有不同。本示例特定将需要注意的部分以粗体格式予以标示，它们是后续步骤的重要提示信息。命令执行的结果如下所示：

```
.....
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
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 a
.....
```

```
kubeadm join 172.16.0.6:6443 --token abcdef.0123456789abcdef \
    --discovery-token-ca-cert-hash sha256:14f2e6b032ac349de8304ef7f3ac859618a08
```

根据上述输出信息的提示，完成集群部署还需要执行三类操作，设定 kubectl 的配置文件，部署网络附件以及将各 Node 加入集群。下面就来讲解如何进行这三步操作。

## A.2.4 设定 kubectl 的配置文件

kubectl 是执行 Kubernetes 集群管理的核心工具。默认情况下，kubectl 会从当前用户主目录（保存在环境变量 HOME 中的值）中的隐藏目录 .kube 下名为 config 的配置文件中读取配置信息，包括要接入 Kubernetes 集群、以及用于集群认证的证书或令牌等信息。集群初始化时，kubeadm 会自动生成一个用于此类功能的配置文件 /etc/kubernetes/admin.conf，将它复制为用户的 \$HOME/.kube/config 文件即可直接使用。这里以 Master 节点上的 root 用户为例进行操作，不过，在实践中应该以普通用户的身份进行：

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter #]**[command mkdir -p $HOME/
[**[prompt root@master]**[path ~]]**[delimiter #]**[command cp -i /etc/kubei
```

至此为止，一个 Kubernetes Master 节点已经基本配置完成。接下来即可通过 API Server 来验证其各组件的运行是否正常。kubectl 有着众多子命令，其中 `get componentsstatus` 即能显示出集群组件当前的状态，也可使用其简写格式 `get cs`：

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter #]**[command kubectl get cs]
NAME          STATUS      MESSAGE           ERROR
controller-manager  Healthy    ok
scheduler        Healthy    ok
etcd-0          Healthy    {"health":"true"}
```

若上面命令结果的 STATUS 字段为 “Healthy”，则表示组建处于健康运行状态，否则需要检查其错误所在，必要时可使用 “kubeadm reset” 命令重置之后重新进行集群初始化。

另外，使用 `kubectl get node` 命令能够获取集群节点的相关状态信息，如下命令结果显示了 Master 节点的状态为 “NotReady”（未就绪），这是因为集群中尚未安装网络插件所致，执行完后面的其他步骤后它即自行转为 “Ready”：

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter #]**[command kubectl get nodes]
NAME          STATUS     ROLES   AGE     VERSION
master.renkeju.com  NotReady  master  108s   v1.14.3
```

## A.2.5 部署网络插件

为 Kubernetes 提供 Pod 网络的插件有很多，目前最为流行的是 flannel 和 Calico。相比较来说，flannel 以其简单、易部署、易用性等特性广受用户喜欢。基于 kubeadm 部署时，flannel 同样运行为 Kubernetes 集群的附件，以 Pod 的形式部署运行于每个集群节点上以接受 Kubernetes 集群管理。事实上，也可以直接将 flannel 程序包装并以守护进程的方式运行于集群节点上，即以非托管的方式运行。部署方式既可以是获取其资源配置清单于本地而后部署于集群中，也可以直接在线进行应用部署。部署命令是 `kubectl apply` 或 `kubectl create`，例如，下面的命令将直接使用在线的配置清淡进行 flannel 部署：

```
**[terminal]
[**[prompt root@master]**[path ~]**[delimiter #]**[command kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/218d7d5437c1eac9ecf4d9f660835d34ca861a8/Documentation/kube-flannel.yml]
```

`kubectl` 可根据定义资源对象的清单文件将其提交给 API Server 以管理对象，如使用 `kubectl apply -f /PATH/TO/MANIFEST` 命令即可根据清单设置资源的目标状态。`kubectl` 的具体使用会再后面的章节进行详细的介绍。

配置 flannel 网络插件时，Master 节点上的 Docker 首先会去获取 flannel 的镜像文件，而后根据镜像文件启动相应的 Pod 对象。待其运行完成后再次查看集群中的节点状态可以看出 Master 已经变为“Ready”状态：

```
**[terminal]
[**[prompt root@master]**[path ~]**[delimiter #]**[command kubectl get nodes]
```

NAME	STATUS	ROLES	AGE	VERSION
master.renkeju.com	Ready	master	45h	v1.14.3

`kubectl get pods -n kube-system | grep flannel` 命令的结果显示 Pod 的状态为 Running 时即表示网络插件 flannel 部署完成。

## A.2.6 添加 Node 到集群中

Master 各组建运行正常后即可将各 Node 添加至集群中。配置节点时，需要事先参考前面“设置容器运行环境”和“设定 Kubernetes 集群节点”两节中的配置过程设置好 Node 主机，而后即可再 Node 主机上使用“kubeadm join”命令将其加入集群中。不过，为了系统安全起见，任何一个试图加入到集群中的节点都需要先经由 API Server 完成认证，其认证方法和认证信息在 Master 上运行 `kubernetes init` 命令执行初始化时将于输出结果信息的最后一部分中提供。

例如，类似如下命令即可将 node01.renkeju.com 加入集群中，它使用集群初始化时生成的认证令牌（token）信息进行认证。另外，同样出于为操作系统及其他应用保留交换分区之目的，也可以在 `kubeadm join` 命令上添加 `--ignore-preflight-errors=Swap` 选项：

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter # ]**[command kubeadm join 172.16.1.10 --token 69e21bce3379b13fdb093195722f --discovery-token-ca-cert-hash sha256:79f6a440a71e21bce3379b13fdb093195722f]
```

提供给 API Server 的 bootstrap token 认证完成后，`kubeadm join` 命令会为后续 Master 与 Node 组件间的双向 ssl/tls 认证生成私钥及证书签署请求，并由 Node 在首次加入集群时提交给 Master 端的 CA 进行签署。默认情况下，`kubeadm` 配置 `kube-apiserver` 启用了 bootstrap TLS 功能，并支持证书的自动签署。于是，`kubelet` 及 `kube-proxy` 等组件的相关私钥和证书文件在命令执行结束后便可自动生成，它们默认保存与 `/var/lib/kubelet/pki` 目录中。

在每个节点上重复上述步骤就能够将其加入集群中。所有节点加入完成后，即可使用 `kubectl get nodes` 命令验证集群的节点状态，包括各节点的名称、状态就绪与否、角色（是为节点 Master）、加入集群的时长以及程序的版本等信息：

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter # ]**[command kubectl get nodes]
NAME           STATUS    ROLES   AGE     VERSION
master.renkeju.com   Ready    master   45h    v1.14.3
node01.renkeju.com  Ready    <none>  45h    v1.14.3
node02.renkeju.com  Ready    <none>  45h    v1.14.3
node03.renkeju.com  Ready    <none>  45h    v1.14.3
```

到此为止，使用 `kubeadm` 构建 Kubernetes 集群已经完成。后续若有 Node 需要加入，其方式均可使用此节介绍的方式来进行。

## A.2.7 获取集群状态信息

Kubernetes 集群以及部署的插件提供了多种不同的服务，如此前部署过的 API Server、Kube-dns 等。API 客户端访问集群时需要事先知道 API Server 的通告地址，管理员可使用 `kubectl cluster-info` 命令了解到这些信息：

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter #]**[command kubectl cluster-
Kubernetes master is running at https://172.16.0.6:6443
KubeDNS is running at https://172.16.0.6:6443/api/v1/namespaces/kube-system/sei

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'
```

Kubernetes 集群 Server 端和 Client 端的版本等信息可以使用 `kubectl version` 命令进行查看：

```
**[terminal]
[**[prompt root@master]**[path ~]]**[delimiter #]**[command kubectl version
Client Version: v1.14.3
Server Version: v1.14.0
```

## A.3 从集群中移除节点

运行过程中，若有节点需要从正常运行的集群中移除，则可使用如下步骤来进行。

1. 在 Master 上使用如下命令“排干”（迁移至集群中的其他节点）当前节点之上的 Pod 资源并移除 Node 节点：

```
**[terminal]
**[path ~]**[delimiter #]**[command kubectl drain NODE_ID --delete-local-pv]
**[path ~]**[delimiter #]**[command kubectl delete NODE_ID]
```

2. 而后在要删除的 Node 上执行如下命令重置系统状态便可完成移除操作：

```
**[terminal]
**[path ~]**[delimiter #]**[command kubectl delete node NODE_ID]
```

## A.4 重新生成用于节点加入集群的认证命令

如果忘记了记录 Master 主机的 kubeadm init 命令执行结果中用于让节点加入集群的 kubeadm join 命令及其认证信息，则需要分别通过 kubectl 获取认证令牌及验证 CA 公钥的哈希值。

kubeadm token list 能够获取到集群上存在认证令牌，定位到其中 DESCRIPTION 字段中标示为由 kubeadm init 命令生成的行，其第一字段 TOKEN 中的令牌即为认证令牌。而验证CA公钥的哈希值（discovery-token-ca-cert-hash）的获取命令则略微复杂，其完成格式如下所示：

```
**[terminal]
[**[prompt root@master]**[path ~]**[delimiter #]**[command openssl x509 -pl
```

而后，将上述两个命令生成的结果合成如下格式的 kubeadm join 命令即可用于让 Node 加入集群中，其中的 TOKEN 可替换为上面第一个命令的生成结果，HASH 可替换为第二个命令的生成结果：

```
kubeadm join 172.16.0.6:6443 --token TOKEN --discover-token-ca-cert-hash HASH
```

最后，需要提供读者注意的是，Kubernetes 项目目前处于快速迭代时期，其版本号演进速度较快，因此，读者在测试阶段试默认安装的程序版本与本书使用的极有可能存在着不同，甚至连部署步骤都可能会发生改变。

## B 部署 GlusterFS 及 Heketi

在实践 Kubernetes 的 StatefulSet 及各种需要持久存储数据的组件活功能时，通常会用到 PV 的动态供给功能，这就需要用到支持此类功能的存储系统。在各类支持 PV 动态供给功能的存储系统中，GlusterFS 的设定较为简单，因此本书用到的各类动态存储功能将以此为例进行说明。本章将试图为读者提供一个设置以满足此功能的 GlusterFS 存储系统的简单说明文档，而不是对 GlusterFS 进行全面介绍。

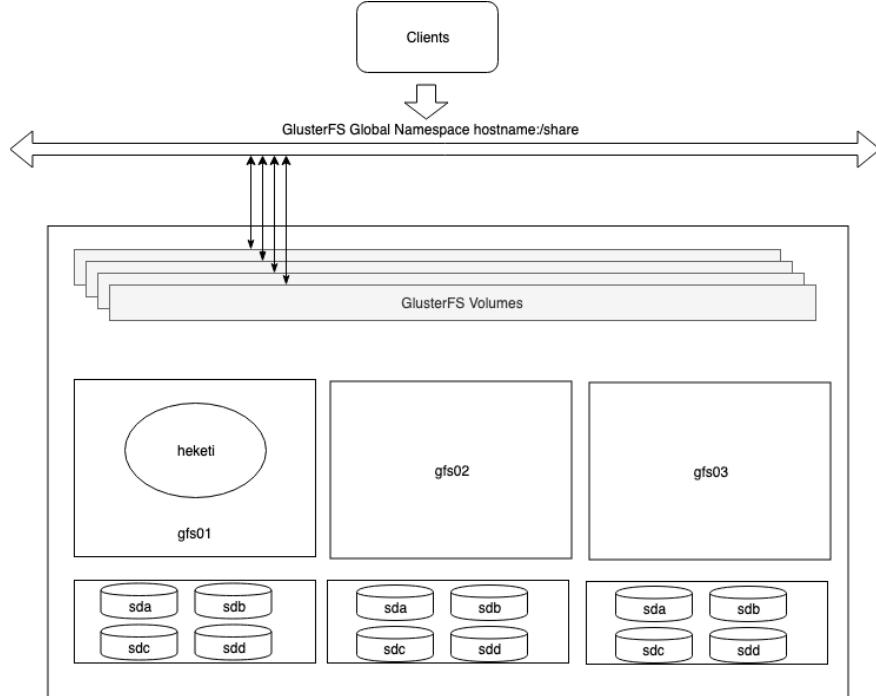


图 1.3 - GlusterFS 架构

在本示例中，`gfs01.renkeju.com`、`gfs02.renkeju.com` 和 `gfs03.renkeju.com` 三个节点组成了 GlusterFS 存储集群（如图 B-1 所示），并将 `gfs01` 节点部署为 `heketi` 服务器。多节点上，`sdb`、`sdc` 和 `sdd` 用于为 GlusterFS 提供存储空间。

## B.1 部署 GlusterFS 集群

首先，分别在三个节点上安装 glusterfs-server 程序包，并启动 glusterfsd 服务，命令如下：

```
**[terminal]
[[path ~]]*[delimiter # ]**[command yum install centos-release-glusterfs]
[[path ~]]*[delimiter # ]**[command yum --enbalerepo=centos-gluster*-test :
[[path ~]]*[delimiter # ]**[command systemctl start glusterd.service]
```

第二步，在任一节点上使用 `glusterfs peer probe` 命令“发现”其他节点，组建 GlusterFS 集群。命令格式为 `peer probe {<HOSTNAME>|<IP-address>}`，例如，在 gfs01 上运行如下命令：

```
**[terminal]
[[prompt root@gfs01]]*[path ~]]*[delimiter # ]**[command gluster peer probe
[[prompt root@gfs01]]*[path ~]]*[delimiter # ]**[command gluster peer probe
```

第三步，通过节点状态命令 `gluster peer status` 确认各节点已经加入同一个可信池中（trusted pool）：

```
**[terminal]
[[prompt root@gfs01]]*[path ~]]*[delimiter # ]**[command gluster peer probe
peer probe: success.
[root@gfs01 ~]# gluster peer status
Number of Peers: 2

Hostname: gfs02.renkeju.com
Uuid: d3621cd7-c621-4a2b-88c2-dc3a18b29d5b
State: Peer in Cluster (Connected)

Hostname: gfs03.renkeju.com
Uuid: e1b8caf5-6f58-4082-b492-9f63ee6d5704
State: Peer in Cluster (Connected)
```

## B.2 部署 Heketi

Heketi 为管理 GlusterFS 存储卷的生命周期提供一个 RESTful 管理接口，借助于 Heketi，像 OpenStack Manila、Kubernetes 和 OpenShift 这样的云服务可以动态调配 Gluster 存储卷。Heketi 能够自动确定整个集群的 brick 位置，并确保将 brick 及其副本放置在不同的故障域中。另外，Heketi 还支持任意数量的 Gluster 存储集群，并支持云服务提供网络文件存储。

有了 Heketi，存储管理员不必再管理或配置 brick、磁盘或可信存储池（trusted pool），Heketi 服务将为管理员管理所有硬件，并使其能够按需分配存储。不过，在 Heketi 中注册的任何磁盘都必须以原始格式提供，而不能是创建过文件系统的磁盘分区。Heketi 架构如下图：



图 1.3.2 - Heketi 架构

本部署示例将 gfs01.renkeju.com 节点用作 Heketi 服务器，因此以下所有步骤均在 gfs01 上执行。

## B.2.1 安装并启动 Heketi 服务器

首先安装 Heketi。Heketi 程序可于 epel 仓库中获取，配置好相关的仓库后即可运行如下安装命令：

```
**[terminal]
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command yum install heketi]
```

第二步，配置 Heketi 用户能够基于 SSH 密钥的认证方式连接至 GlusterFS 集群中的各节点，并拥有相应节点的管理权限：

```
**[terminal]
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command ssh-keygen -f /etc/ssh/ssh_host_rsa_key -N ""]
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command chown heketi:heketi /etc/ssh/ssh_host_rsa_key]
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command for host in gfs01; do ssh-copy-id -i /etc/ssh/ssh_host_rsa_key $host; done]
```

第三步，设置 Heketi 的主配置文件 /etc/heketi/heketi.json，定义服务监听的端口、认证及连接 Gluster 存储集群的方式。一个配置示例如下：

```
{
  "port": "8080",
  "user_auth": false,
  "jwt": {
    "admin": {
      "key": "admin Secret"
    },
    "user": {
      "key": "user Secret"
    }
  },
  "glusterfs": {
    "executor": "ssh",
    "sshexec": {
      "keyfile": "/etc/heketi/heketi_key",
      "user": "root",
      "port": "22",
      "fstab": "/etc/fstab"
    },
    "db": "/var/lib/heketi/heketi.db",
    "loglevel": "debug"
  }
}
```

## A.1.1 部署目标

若要启用连接 Heketi 的认证，则需要将“use\_auth”参数的值设置为“true”，并在“jwt{}”配置段中为各用户设定相应的密码，用户名和密码都可以自定义。“glusterfs{}”配置段用于指定接入 Gluster 存储集群的认证方式及信息。

若启用了认证功能，则于 Kubernetes 集群中配置存储类时需要设置相应的认证信息。

第四步，启动 Heketi 服务：

```
**[terminal]
[[prompt root@gfs01]]*[path ~]*[delimiter #]*[command systemctl enable
[[prompt root@gfs01]]*[path ~]*[delimiter #]*[command systemctl start h
[[prompt root@gfs01]]*[path ~]*[delimiter #]*[command curl http://gfs01:5443/v1/health
[[root@gfs01 ~]# curl http://gfs01:5443/v1/health
Hello from Heketi
```

需要注意的是，将 Gluster 存储集群的功能托管于 Heketi 之后便不能够再于集群中使用命令管理存储卷，以免于 Heketi 数据库中存储的信息不一致。

第五步，向 Heketi 发起访问测试请求，无须认证时，使用 curl 命令既能完成测试：

```
**[terminal]
[[prompt root@gfs01]]*[path ~]*[delimiter #]*[command curl http://gfs01:5443/v1/health
[[root@gfs01 ~]# curl http://gfs01:5443/v1/health
Hello from Heketi
```

若 Heketi 启用了认证功能，则需要使用 heketi-cli 命令进行测试，命令格式如下：

```
**[terminal]
[[prompt root@gfs01]]*[path ~]*[delimiter #]*[command heketi-cli --ser
[[root@gfs01 ~]# heketi-cli --server http://gfs01:5443 status
{
  "status": "ok"
}
```

## B.2.2 设置 Heketi 系统拓扑

拓扑信息用于让 Heketi 确认可使用的节点、磁盘和集群，管理员必须自行确定节点故障域和节点集群。故障域时赋予一组节点的整数值，这组节点共享共同的交换机、电源或其他任何会导致他们同时失效的组件。管理员必须确定哪些节点构成一个集群，Heketi 使用这些信息来确保跨故障域中创建副本，从而提供数据的冗余能力。Heketi 支持多个 Gluster 存储集群，这为管理员提供了创建 SSD、SAS、SATA 或为用户提供特定服务质量的任何其他类型的集群的选项。

命令行客户端 heketi-cli 通过加载预定义的集群拓扑，从而添加节点到集群中，以及将磁盘关联到节点上。要使用 heketi-cli 加载拓扑文件，可通过以下命令完成：

```
**[terminal]
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command export HEKETI_CL]
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command heketi-cli topology]
```

一个适用于当前配置环境的示例配置如下所示 `/etc/heketi/topology_demo.json`，它根据 Gluster 存储集群的实际环境把 gfs01、gfs02 和 gfs03 三个节点定义在同一个集群中并指明各节点上可用于提供存储空间的磁盘设备：

### A.1.1 部署目标

```
{  
    "clusters": [  
        {  
            "nodes": [  
                {  
                    "node": {  
                        "hostnames": {  
                            "manage": [  
                                "172.16.0.10"  
                            ],  
                            "storage": [  
                                "172.16.0.10"  
                            ]  
                        },  
                        "zone": 1  
                    },  
                    "devices": [  
                        "/dev/vdb"  
                    ]  
                },  
                {  
                    "node": {  
                        "hostnames": {  
                            "manage": [  
                                "172.16.0.11"  
                            ],  
                            "storage": [  
                                "172.16.0.11"  
                            ]  
                        },  
                        "zone": 1  
                    },  
                    "devices": [  
                        "/dev/vdb"  
                    ]  
                },  
                {  
                    "node": {  
                        "hostnames": {  
                            "manage": [  
                                "172.16.0.12"  
                            ],  
                            "storage": [  
                                "172.16.0.12"  
                            ]  
                        },  
                        "zone": 1  
                    }  
                }  
            ]  
        }  
    ]  
}
```

### A.1.1 部署目标

```
        "devices": [
            "/dev/vdb"
        ]
    }
]
}
}
```

而后运行如下命令加载拓扑信息，从而完成集群配置。此命令会生成一个集群，并为其添加的各节点生成随机 ID 号：

```
**[terminal]
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command export HEKETI_CL...
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command heketi-cli topolo...
```

而后运行如下命令查看集群的状态信息：

```
**[terminal]
[**[prompt root@gfs01]**[path ~]]**[delimiter # ]**[command heketi-cli cluster...
Cluster id: f97313b9354866c7dc1b9f21a341605c
Nodes:
62db8b2082e97bf5408b258d6a61c2a8
8dbb0b2812f2c05653a7fdcc3c510df0
f6f0e94620ed747b97f13df537423836
Volumes:

Block: true

File: true
```

heketi-cli volume create --size=<size in Gb> [options] 能够创建存储卷，例如，下面的命令测试即用测试即用于创建一个存储卷：

## A.1.1 部署目标

```
**[terminal]
[**[prompt root@gfs01]**[path ~]]**[delimiter #]**[command heketi-cli volume get vol_a7b311867ae269d1e705f8a772e2b253]
Name: vol_a7b311867ae269d1e705f8a772e2b253
Size: 2
Volume Id: a7b311867ae269d1e705f8a772e2b253
Cluster Id: f97313b9354866c7dc1b9f21a341605c
Mount: 172.16.0.10:vol_a7b311867ae269d1e705f8a772e2b253
Mount Options: backup-volfile-servers=172.16.0.11,172.16.0.12
Block: false
Free Size: 0
Reserved Size: 0
Block Hosting Restriction: (none)
Block Volumes: []
Durability Type: replicate
Distributed+Replica: 3
```

而后在要使用远程存储卷的节点上安装 GlusterFS 和 glusterfs-fuse 的程序包，提供 GlusterFS 客户端驱动及对 GlusterFS 文件系统的运行，并基于 GlusterFS 文件系统类型挂载使用确认无误后即可删除测试卷。删除 Heketi 卷的命令为 `heketi-cli volume delete <vol_id>`，如何删除前面创建的存储卷，可使用以下的命令：

```
**[terminal]
[**[prompt root@gfs01]**[path ~]]**[delimiter #]**[command heketi-cli volume delete vol_a7b311867ae269d1e705f8a772e2b253]
Volume a7b311867ae269d1e705f8a772e2b253 deleted
```

至此为止，一个支持动态存储卷配置的 GlusterFS 存储集群即设置完成，用户即可于 Kubernetes 中通过 PVC 请求使用某事先创建完成的 GlusterFS 存储卷，也可把 Heketi 配置为存储类，而后提供 PV 的动态供给功能。

## 第二章 kubernetes 快速入门

Kubernetes 集群将所有节点上的资源都整合到一个大的虚拟资源池里，以代替一个个单独的服务器，而后开放诸如 CPU、内存和 I/O 这些基本资源用于运行其基本单元——Pod 资源对象。Pod 的容器中运行着隔离的任务单元，它们以 Pod 为原子单位，并根据其资源需求从虚拟资源池中为其动态分配资源。若可以将整个集群类比为一台传统的服务器，那么 Kubernetes（Master）就好比是操作系统内核，其主要职责在于抽象资源并调度任务，而 Pod 资源对象就是那些运行于用户空间中的进程。于是，传统意义上的向单节点或集群直接部署、配置应用的模型日渐式微，取而代之的是向 Kubernetes 的 API Server 提交运行 Pod 对象。

API Server 是负责接受并响应客户端提交任务的接口，用户可使用诸如 CLI 工具（如 kubectl）、UI 工具（如 Dashboard）或程序代码（客户端开发库）发起请求，其中，kubectl 是最为常用的交互式命令行工具。快速了解 kubernetes 的办法之一就是部署一个测试集群，并尝试测试使用他的各项基本功能。本章在简单介绍核心资源对象后将尝试使用 kubectl 创建 Deployment 和 Service 资源部署并暴露一个 Web 应用，以便读者快速了解如何在 Kubernetes 系统上运行应用程序的核心任务。

## 2.1 Kubernetes 的核心对象

API Server 提供了 RESTful 风格的编程接口，其管理的资源是 Kubernetes API 中的端点，用于存储某种 API 对象的集合，例如，内置 Pod 资源是包含了所有 Pod 对象的集合。资源对象是用于表现集群状态的实体，常用于描述应于哪个节点进行容器化应用、需要为其配置什么资源以及应用程序的管理策略等，例如，重启、升级及容错机制。另外，一个对象也是一种“意向记录”——一旦创建，Kubernetes 就需要一直确保对象始终存在。Pod、Deployment 和 Service 等都是最常见核心对象。

## 2.1.1 Pod 资源对象

Pod 资源对象是一种集合了一到多个应用容器、存储资源、专用IP及支撑容器运行的其他选项的逻辑组件，如下图所示。换言之，Pod 代表着 Kubernetes 的部署单元及原子运行单元，即一个应用程序的单一运行实例，它通常由共享资源且关系紧密的一个或多个应用容器组成。

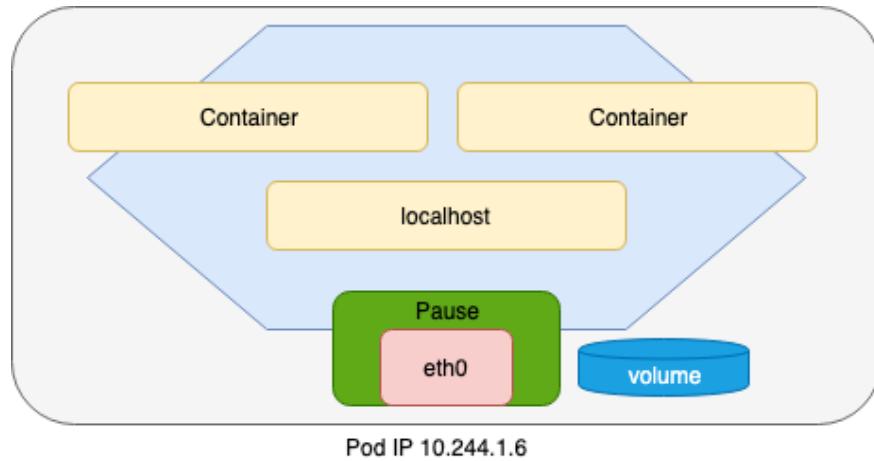


图 1.4.1.1 - Pod 通常由一个到多个共享网络和存储资源的容器组合而成

Kubernetes 的网络模型要求其各 Pod 对象的 IP 地址位于同一网络平面内（同一 IP 网段），各 Pod 之间可使用其 IP 地址直接进行通信，无论他们运行于集群内的哪个工作节点之上，这些 Pod 对象都像是运行于同一局域网中的多个主机。

读者可以将每个 Pod 对象想象成一个逻辑主机，它类似于现实世界中的物理主机或 VM (Virtual Machine)，运行于同一个 Pod 对象中的多个进程也类似于物理机或 VM 上独立运行的进程。不过，Pod 对象中的各进程均运行于彼此隔离的容器中，并于各容器间共享两种关键资源：**网络**和**存储卷**。

- 网络 (networking)：每个 Pod 对象都会被分配一个集群内专用的 IP 地址，也称为 Pod IP，同一 Pod 内部的所有容器共享 Pod 对象的 Network 和 UTS 名称空间，其中包括主机名、IP 地址和端口等。因此，这些容器间的通信就可以基于本地回环接口 lo 进行，而与 Pod 外的其他组件的通信则需要使用 Service 资源对象的 ClusterIP 及其相应的端口完成。
- 存储卷 (volume)：用户可以为 Pod 对象配置一组“存储卷”资源，这些资源可以共享给其内部的所有容器使用，从而完成容器间数据的共享。存储卷还可以确保在容器终止后被重启，甚至是被删除后也能确保数据不会丢失，从而保证了生命周期内的 Pod 对象数据的持久化存储。

一个 Pod 对象代表某个应用程序的一个特定实例，如果需要扩展应用程序，则意味着为此应用程序同时创建多个 Pod 实例，每个实例均代表应用程序的一个运行的“副本” (replica)。这些副本化的 Pod 对象的创建和管理通常由一组称之为“控制器” (Controller) 的对象实现，例如，Deployment 控制器对象。

### A.1.1 部署目标

创建 Pod 时，还可以使用 Pod Preset 对象为 Pod 注入特定的信息，如 ConfigMap、Secret、存储卷、卷挂载和环境变量等。有了 Pod Preset 对象，Pod 模版的创建者就无须为每个模版显示提供所有信息，因此，也就无须事先了解需要配置的每个应用的细节即可完成模版定义。这些内容将在后面的章节中予以介绍。

基于期望的目标状态和各节点的资源可用性，Master 会将 Pod 对象调度至某选定的工作节点运行，工作节点于指向的镜像仓库（image registry）下载镜像，并于本地的容器运行时环境中启动容器。Master 会将整个集群的状态保存与 etcd 中，并通过 API Server 共享给集群的各个组件及客户端。

## 2.1.2 Controller

Kubernetes 集群的设计中，Pod 是有生命周期的对象。用户通过手工创建或由 Controller（控制器）直接创建的 Pod 对象会被“调度器”（Scheduler）调度至集群中的某工作节点运行，待到容器应用进程运行结束之后正常终止，随后就会被删除。另外，节点资源耗尽或故障也会导致 Pod 对象被回收。

但 Pod 对象本身并不具有“自愈”功能，若是因为工作节点甚至是调度器自身导致了运行失败，那么它将会被删除；同样，资源耗尽或节点故障导致的回收操作也会删除相关的 Pod 对象。在设计上，Kubernetes 使用“控制器”实现对一次性的（用后即弃）Pod 对象的管理操作，例如，要确保部署的应用程序的 Pod 副本数量严格反应用于期望的数目，以及基于 Pod 模版来重建 Pod 对象等，从而实现 Pod 对象的扩缩容、滚动更新和自愈能力等。例如，某节点发生故障时，相关的控制器会将此节点上运行的 Pod 对象重新调度到其他节点上进行重建。

控制器本身也是一种资源类型，它有着多种实现，其中与工作负载相关的实现 Replication Controller、Deployment、StatefulSet、DaemonSet 和 Jobs 等，也可统称它们为 Pod 控制器。如下图中的 Deployment 就是这类控制器的代表实现，是目前最常用的管理无状态应用的 Pod 控制器。

Pod 控制器的定义通常由期望的副本数量、Pod 模版和标签选择器（Label Selector）组成。Pod 控制器会根据标签选择器对 Pod 对象的标签进行匹配检查，所有满足选择条件的 Pod 对象都将受控于当前控制器并计入其副本总数，并确保此数目能够精确反映期望的副本数。

需要注意的是，在实际的应用场景中，在接受到请求流量负载显著低于或接近于已有 Pod 副本的整体承载能力时，用户需要手动修改 Pod 控制器中的期望副本数量以实现应用规模的扩容或缩容。不过，若集群中部署了 Heapster 或 Prometheus 一类的资源指标监控附件时，用户还可以使用“HorizontalPodAutoscaler”(HPA) 计算出合适的 Pod 副本数量，并自动修改 Pod 控制器中期望的副本数以实现应用规模的动态伸缩，提高集群资源利用率，如下图所示。

Kubernetes 集群中的每个节点都运行着 cAdvisor 以收集容器及节点 CPU、内存及磁盘资源的利用率指标数据，这些统计数据由 Heapster 聚合后可通过 API Server 访问。HorizontalPodAutoscaler 基于这些统计数据监控容器健康状态并做出扩展决策。

### A.1.1 部署目标

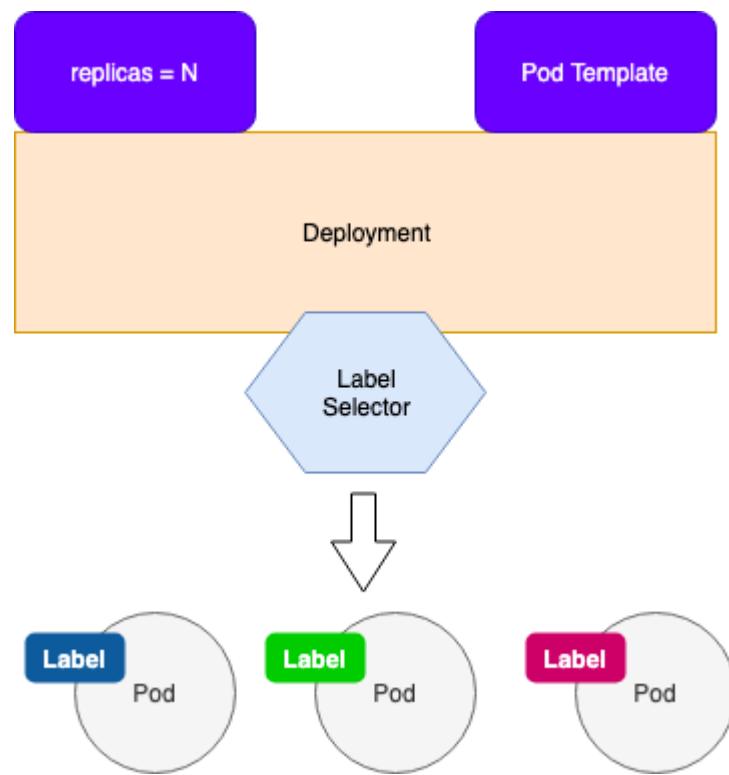


图 1.4.1.2 - Replication Controller

### A.1.1 部署目标

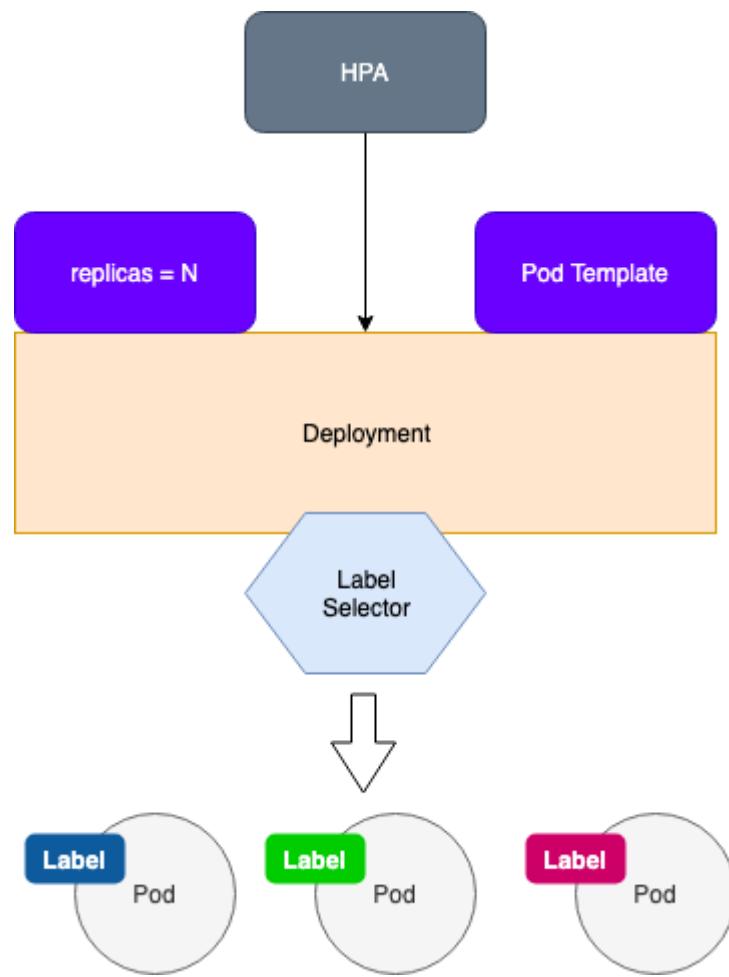


图 1.4.1.2 - Horizontal Pod Autoscaler

## 2.1.3 Service

尽管 Pod 对象可以拥有 IP 地址，但此地址无法确保在 Pod 对象重启或被重建后保持不变，这会为集群中的 Pod 应用间依赖关系的维护带来麻烦：前端 Pod 应用（依赖方）无法基于固定地址持续跟踪后端 Pod 应用（被依赖方）。于是，Service 资源被用于在被访问的 Pod 对象中添加一个有着固定 IP 地址的中间层，客户端向此地址发起访问请求后由相关的 Service 资源调度并代理至后端的 Pod 对象。

换言之，Service 是“微服务”的一种实现，事实上它是一种抽象：通过规则定义出由多个 Pod 对象组合而成的逻辑集合，并附带访问这组 Pod 对象的策略。Service 对象挑选，关联 Pod 对象的方式和 Pod 控制器一样，都是要基于 Label Selector 进行定义，其示意图如下图所示。

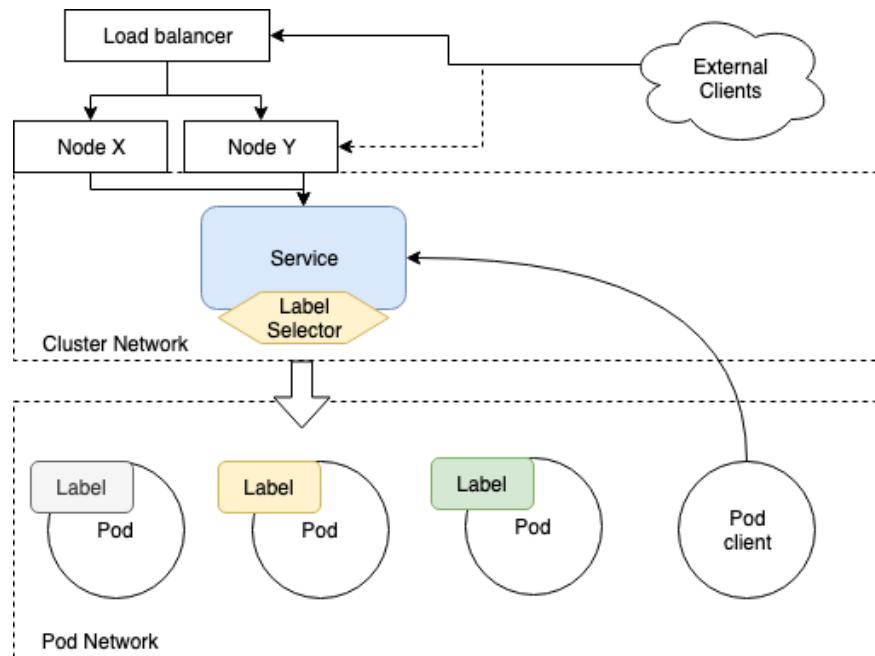


图 1.4.1.3 - Service 对象功能示意图

Service IP 是一种虚拟 IP，也称为 Cluster IP，它专用于集群内通信，通常使用专用的地址段，如 “10.96.0.0/12” 网络，各 Service 对象的 IP 地址在此范围内由系统动态分配。

集群内的 Pod 对象可直接请求此类的 Cluster IP，例如，上图中来自 pod client 的访问请求即可以 Service 的 Cluster IP 作为目标地址，但集群网络属于私有网络地址，它们仅在集群内部可达。将集群外部的访问流量引入集群内部的常用方法是通过节点网络进行，实现方法是通过工作节点的 IP 地址和某端口（NodePort）接入请求并将其代理至相应的 Service 对象的 Cluster IP 上的服务端口，而后由 Service 对象的 Cluster IP 上的服务端口，而后由 Service 对象将请求代理至后端的 Pod 对象的 Pod IP 及应用程序监听的端口。因此，诸如上图中 External Clients

### A.1.1 部署目标

这种来自集群外部的客户端无法直接请求此 Service 提供的服务，而是需要事先经由某一个工作节点（如 Node Y）的 IP 地址进行，这类请求需要两次转发才能到达目标 Pod 对象，因此在通信效率上必然存在负面影响。

事实上，NodePort 会部署于集群中的每一个节点，这就意味着，集群外部的客户端通过任何一个工作节点的 IP 地址来访问定义好的 NodePort 都可以到达相应的 Service 对象。此种场景中，如果存在集群外部的一个负载均衡器，即可将用户请求负载均衡至集群中的部分或者所有节点。这是一种称为“LoadBalancer”类型的 Service，它通常是由 Cloud Provider 自动创建并提供的软件负载均衡器，不过，也可以是由管理员手工配置的诸如 F5 Big-IP 一类的硬件设备。

简单来说，Service 主要由三种常用类型：第一种是仅用与集群内部通信的 ClusterIP 类型；第二种是接入集群外部请求的 NodePort 类型，它工作于每个节点的主机 IP 之上；第三种是 LoadBalacer 类型，它可以把外部请求负载均衡至多个 Node 的主机IP的 NodePort 之上。此三种类型中，每一种都以前一种为基础才能实现，而且第三种类型中的 LoadBalacer 需要协同集群外部的组建才能实现，并且此外部组建并不接受 Kubernetes 的管理。

## 2.1.4 部署应用程序的主体过程

Docker 容器技术使得部署应用程序从传统的安装、配置、启动应用程序的方式转为在容器引擎上基于镜像创建和运行容器，而 Kubernetes 又使得创建和运行容器的操作不必再关注其位置，并在一定程度上赋予了它动态扩缩容及自愈的能力，从而让用户从主机、系统及应用程序的维护工作中解脱出来。

用到某应用程序时，用户只需要向 API Server 请求创建一个 Pod 控制器，由控制器根据镜像等信息向 API Server 请求创建出一定数量的 Pod 对象，并由 Master 之上的调度器指派至选定的工作节点以运行容器化应用。此外，用户一般还需要创建一个具体的 Service 对象以便为这些 Pod 对象建立起一个固定的访问入口，从而使得其客户端能够通过其服务名称或 ClusterIP 进行访问，如下图所示：

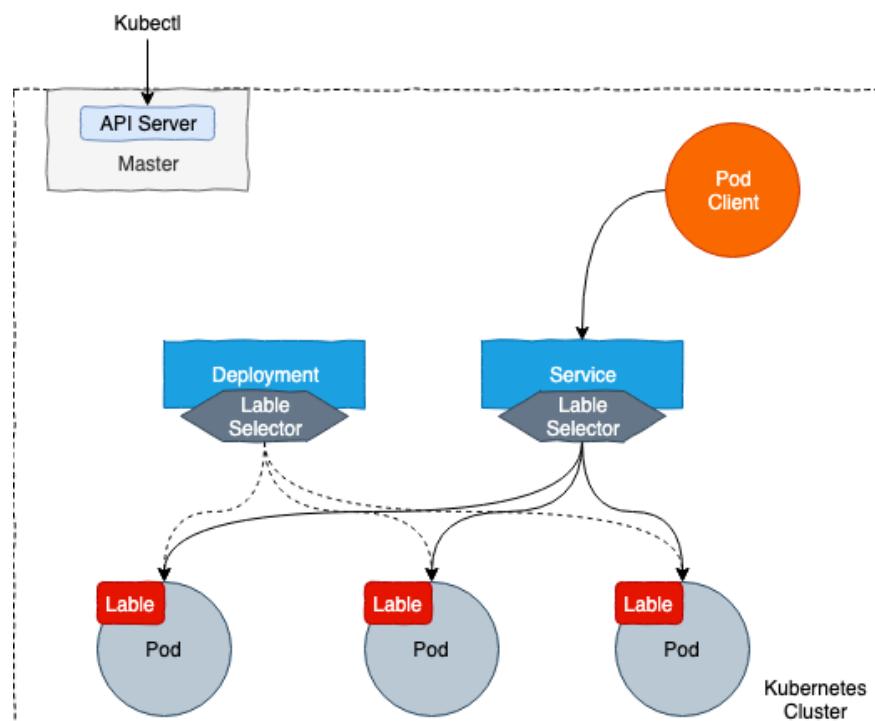


图 1.4.1.4 - 应用程序简单的部署示例

API Server 的常用客户端程序是 Kubernetes 系统自带的命令行工具 kubectl，它通过一众子命令用于实现集群及相关资源对象的管理操作，并支持直接命令式、命令式配置清单及声明式配置清单等三种操作方式，特性丰富且功能强大。而需作为集群附件额外部署的 Dashboard 则提供了基于 Web 界面的图形客户端，它是一个通用目的的管理工具，与 Kubernetes 紧密集成，支持多级别用户授权，能在一定程度上替代 kubectl 的大多数操作。

本章后面的篇幅将介绍在部署完成的 Kubernetes 集群环境众如何快速部署如图所示的示例应用程序，并简单说明如何完成对容器化应用的访问，以及如何进行应用规模的动态伸缩，并借此让读者了解 kubectl 命令的基本功能和用法。

## 2.2 部署 Kubernetes 集群

Kubernetes 系统可运行于多种平台之上，包括虚拟机、裸服务器或 PC 等，例如本地主机或托管的云端虚拟机。若仅用于快速了解或开发的目的，那么读者可直接于单个主机之上部署“伪”分布式的 Kubernetes 集群，将集群的所有组建均部署运行于单台主机上，著名的 minikube 项目可帮助用户快速构建此类环境。如果要学习使用 Kubernetes 集群的完整功能，则应该构建真正的分布式集群环境，将 Master 和 Node 等部署与多台主机之上，主机的具体数量要按实际需求而定。另外，集群部署的方式也有多种选择，简单的可以基于 kubeadm 一类的部署工具运行几条命令即可实现，而复杂的则可以是从零开始手动构建集群环境。

## 2.2.1.kubeadm 部署工具

kubeadm 是 Kubernetes 项目自带的集群构建工具，它负责执行构建一个最小化的可用集群以及将其启动等的基本步骤，简单来讲，kubeadm 是 Kubernetes 集群全生命周期的管理工具，可用于实现集群的部署、升级\降级及拆除，如图所示。不过，在部署操作中，kubeadm 仅关心如何初始化并启动集群，余下的其他操作，例如安装 Kubernetes Dashboard、监控系统、日志系统等必要的附加组建则不在其考虑范围之内，需要管理员按需自行部署。

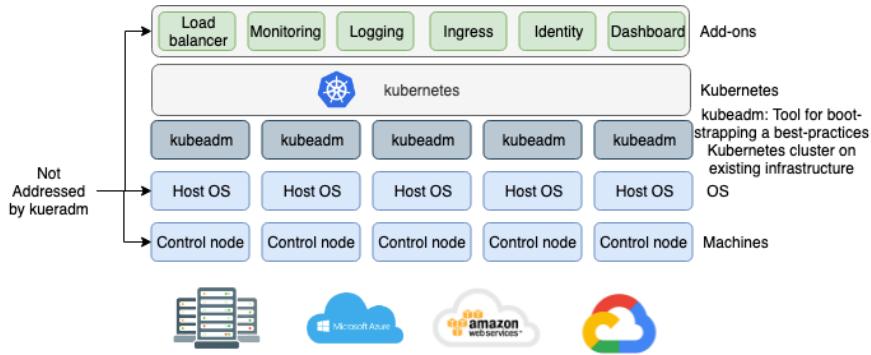


图 1.4.2.1 - kubeadm 功能示意图

kubeadm 集成了 kubeadm init 和 kubeadm join 等工具程序，其中 kubeadm init 用于集群的快速初始化，其核心功能是部署 Master 节点的各个组件，而 kubeadm join 则用于将节点快速加入到指定集群中，它们是创建 Kubernetes 集群最佳实践的“快速路径”。另外，kubeadm token 可于集群构建后管理用于加入集群时使用的认证令牌（token），而 kubeadm reset 命令的功能则是删除集群构建过程中生成的文件以重置回初始状态。

kubeadm 还支持管理初始引导认证令牌（Bootstrap Token），完成待加入的新节点首次联系 API Server 时的身份认证（基于共享密钥）。另外，它们还支持管理集群版本的升级和降级操作。Kubernetes 1.8 版本之前，kubeadm 一直处于 beta 级别，并警告不能用于生产环境。不过，自 1.9 版本开始，其虽仍处于 beta 版本，但已经不再输出警告信息，而随着 1.11 版本发布的 kubeadm 又得到了进一步的增强，它支持动态配置 kubelet，通过增强的 CRI 集成支持动态探测以判定所用的容器引擎，并引入了几个新的命令行工具，包括 kubeadm config print-default、kubeadm config migrate、kubeadm config image pull 和 kubeadm upgrade node config 等。总体来说，使用 kubeadm 部署 Kubernetes 集群具有如下几个方面的优势。

- 简单易用：kubeadm 可完成集群的部署、升级和拆除操作，并且对新手用户非常友好。
- 适用领域广泛：支持将集群部署与裸机、VMware、AWS、Azure、GCE 及更多环境的主机上，且部署过程基本一致。
- 富有弹性：1.11 版本中的 kubeadm 支持阶段式部署，管理员可分为多个独立步骤完成部署操作。

### A.1.1 部署目标

- 生产环境可用：kubeadm 遵循以最佳实践的方式部署 Kubernetes 集群，它强制启用 RBAC，设定 Master 的各组件间以及 API Server 与 kubelet 之间进行认证及安全通信，并锁定了 kubelet API 等。

由此可见，kubeadm 并非一键安装类的解决方案，相反，它有着更宏大的目标，旨在成为一个更大解决方案的一部分，试图为集群创建和运营构建一个声明式的 API 驱动模型，它将集群本身视为不可变组件，而升级操作等同于全新部署或就地更新。目前，使用 kubeadm 部署集群已经成为越来越多的 Kubernetes 工程师的选择。

## 2.2.2 集群运行模式

Kubernetes 集群支持三种运行模式：一是“独立组件”模式，系统各组件直接以守护进程的方式运行于节点之上，各组件之间相互协作构成集群，如图所示，第二种是“静态 Pod 模式”，除 kubelet 和 Docker 之外的其他组件（如 etcd、kube-apiserver、kube-controller、manager 和 kube-scheduler 等）都是以静态 Pod 对象运行于 Master 主机之上的，如图所示；第三种是 Kubernetes 的“自托管”（self-hosted）模式，它类似于第二种方式，将除了 kubelet 和 Docker 之外的其他组件运行为集群之上的 Pod 对象，但不同的是，这些 Pod 对象托管运行在集群自身之上受控于 DaemonSet 类型的控制器，而非静态的 Pod 对象。

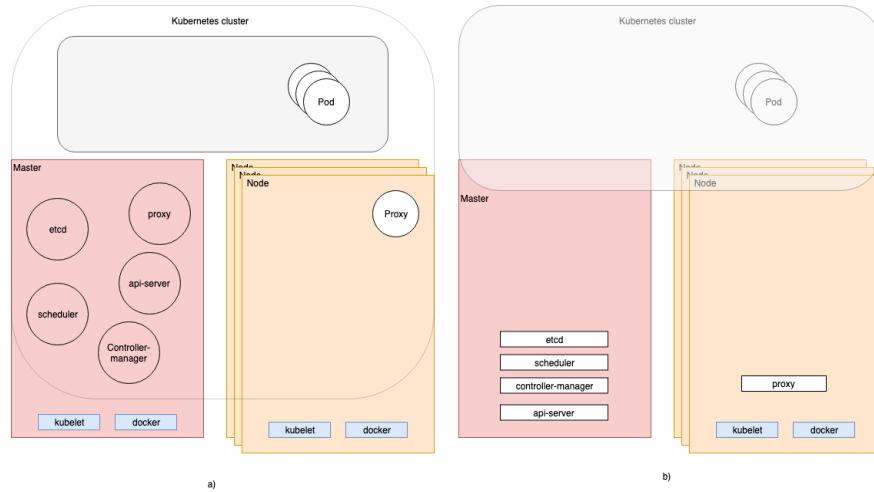


图 1.4.2.2 - kubernetes 集群的运行模式

使用 kubeadm 部署的 Kubernetes 集群可运行为第二种或者第三种模式，默认为静态 Pod 对象模式，需要使用自托管模式时，kubeadm init 命令使用 “--feature-gates=selfHosting” 选项即可。第一种模式集群的构建需要将各组件运行于系统之上的独立守护进程中，其间需要用到的证书及 Token 等认证信息也需要手动生成，过程繁琐且极易出错；若有必要用到，则建议使用 Github 上合用的项目辅助进行，例如，通过 ansible playbook 进行自动部署等。

## 2.2.3 准备用于实践操作的集群环境

本书后面的篇幅中用到的测试集群如图所示，该集群中由一个 Master 主机和是三个 Node 主机组成，它基于 kubernetes 部署，除了 kubelet 和 Docker 之外其他的集群组件都运行于 Pod 对象中。多数情况下，两个或以上的独立运行的 Node 主机即可测试分布式集群的核心功能，因此其数量可按需定义，但两个主机是模拟分布式环境的最低需求。生产实践中，应该至少部署三个协同工作的 Master 节点以确保控制平面的服务可用性，不过，在测试环境中仅部署一个 Master 节点也是常见的选择。

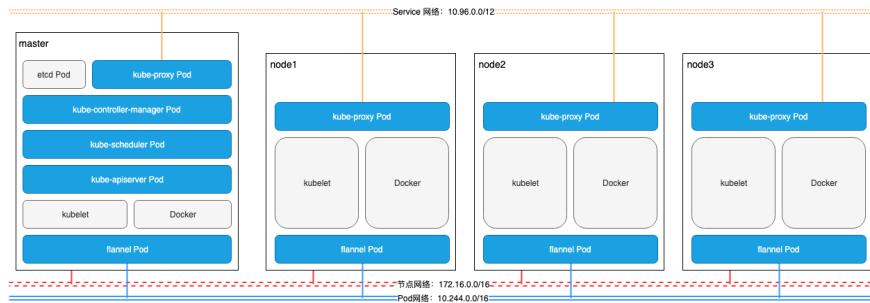


图 1.4.2.3 - Kubernetes 集群部署目标示意图

各 Node 上采用的容器运行时环境为 docker，后续的众多容器的运行任务都将依赖于 Docker Registry 服务，包括 DockerHub、GCR（Google Container Registry）和 Quay 等，甚至是私有的 Registry 服务，本书假设读者对 Docker 容器技术由熟练的使用基础。另外，本部署示例中使用的用于为 Pod 对象提供网络功能的插件是 flannel，其同样以 Pod 对象的形式托管运行于 Kubernetes 系统之上。

具体的部署过程以及本书用到的集群环境请参考 [附录A](#)，本章后续的操作都将依赖于根据其步骤部署完成的集群环境，读者需要根据其中的内容成功搭建出 Kubernetes 测试集群后才能进行后面章节的学习。

## 2.2.4 获取集群环境相关的信息

Kubernetes 系统目前仍然处于快速迭代阶段，版本演进频繁，读者所部署的版本与本书中使用的版本或将有所不同，其功能特性也将存在一定程度的变动。因此，事先查看系统版本，以及对比了解不同版本间的功能特性变动也将是不可或缺的步骤。当然，用户也可选择安装与本书相同的系统版本。下面的命令显示的是当前使用的客户端及服务端程序版本信息。

```
**[terminal]
[**[prompt root@master]**[path ~]**[delimiter #]**[command kubectl version
Client Version: v1.14.3
Server Version: v1.14.0
```

Kubernetes 系统版本变动时的 ChangeLog 可参考 [github.com](https://github.com) 站点上相关版本中的介绍。

Kubernetes 集群以及部署的附件 CoreDNS 等提供了多种不同的服务，客户端访问这些服务时需要事先了解其访问接口，管理员可使用“kubectl cluster-info”命令获取相关的信息。

```
**[terminal]
[**[prompt root@master]**[path ~]**[delimiter #]**[command kubectl cluster-
Kubernetes master is running at https://172.16.0.6:6443
KubeDNS is running at https://172.16.0.6:6443/api/v1/namespaces/kube-system/sei

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'
```

一个功能完整的 Kubernetes 集群应当具备的附加组件还包括 Dashboard、Ingress Controller 和 Heapster（或 Prometheus）等，后续章节中的某些概念将会依赖到这些组件，读者可选择在将要用到时再进行部署。

## 2.3 kubectl 使用基础与示例

Kubernetes API 是管理其各种资源对象的唯一入口，它提供了一个 RESTful 风格的 CRUD(Create、Read、Update 和 Delete)接口用于查询和修改集群状态，并将结果存储于集群状态存储系统 etcd 中。事实上，API Server 也是用于更新 etcd 中资源对象状态的唯一途径，Kubernetes 的其他所有组件和客户端都要通过它来完成查询或修改操作，如图 2-9 所示。从这个角度来讲，他们都算得上是 API Server 的客户端。

任何 RESTful 风格 API 中的核心概念都是“资源”(resource)，它是具有类型、关联数据、同其他资源的关系以及可对其执行的一组操作方法的对象，它与对象式编程语言中的对象实例类似，两者之间的重要区别在于 RESTful API 仅为资源定义了少量的标准方法（对应与标准 HTTP 的 GET、POST、PUT 和 DELETE 方法），而编程语言中的对象实例通常有很多方法。另外，资源可以根据其特性分组，每个组是同一类型资源的集合，它仅包含一种类型的资源，并且各资源间不存在顺序的概念，集合本身也是资源。对应于 Kubernetes 中，Pod、Deployment 和 Service 等都是所谓的资源类型，它们由相应类型的对象集合而成。

API Server 通过认证 (Authentication)、授权 (Authorization) 和准入控制 (Admission Control) 等来管理对资源的访问请求，因此，来自于任何客户端（如 kubectl、kubelet、kube-proxy 等）的访问请求都必须事先完成认证之后方可进行后面的其他操作。API Server 支持多种认证方式，客户端可以使用命令行选项或专用的配置文件（称为 kubeconfig）提供认证信息。相关的内容将在后面的章节中给予详细说明。

kubectl 的核心功能在于通过 API Server 操作 Kubernetes 的各种资源对象，它支持三种操作方式，其中直接命令式 (Imperative commands) 的使用最为简单，是了解 Kubernetes 集群管理的一种有效途径。

## kubectl 命令常用操作示例

为了方便读者快速适应 kubectl 的命令操作，这里给出几个使用示例用于说明其基本使用方法。

### 1. 创建资源对象

直接通过 kubectl 命令及相关的选项创建资源对象的方式即为直接命令式操作，例如下面的命令分别创建了名为 nginx-deploy 的 Deployment 控制器资源对象，以及名为 nginx-svc 的 Service 资源对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl run nginx-deploy --image=nginx:1.12 --
**[delimiter $ ]**[command kubectl expose deployment/nginx --name=nginx-s
```

用户也可以根据资源清单创建资源对象，即命令式对象配置文件，例如，假设存在定义了 Deployment 对象的 nginx-deploy.yaml 文件，和定义了 Service 对象的 nginx-svc.yaml 文件，使用 kubectl create 命令即可进行基于命令式对象配置文件的创建操作：

```
**[terminal]
**[delimiter $ ]**[command kubectl create -f nginx-deploy.yaml -f nginx-
```

甚至还可以将创建交由 kubectl 自行确定，用户只要声明期望的状态，这种方式称为声明式对象配置。例如，假设存在定义了 Deployment 对象的 nginx-deploy.yaml 文件，以及定义了 Service 对象的 nginx-svc.yaml 文件，那么使用 kubectl apply 命令即可实现声明式配置：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f nginx-deploy.yaml -f nginx-s
```

本章后面的章节主要使用第一种资源管理方式，第二种和第三种方式将在后面的章节中展开讲述。

## 2. 查看资源对象

运行着实际负载的 Kubernetes 系统上通常会存在多种资源对象，用户可分类列出感兴趣的资源对象及其相关状态信息，“kubectl get”正是用于完成此类功能的命令。例如，列出系统上所有的 Namespace 资源对象，命令如下：

```
**[terminal]
**[delimiter $ ]**[command kubectl get namespace]
```

用户也可一次查看多个资源类别下的资源对象，例如，列出默认名称空间内的所有 Pod 和 Service 对象，并输出额外信息，可以使用如下形式的 kubectl get 命令：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods,services -o wide]
```

Kubernetes 系统的大部分资源都隶属于某个 Namespace 对象，缺省的名称空间为 default，若需要获取指定 Namespace 对象中的资源对象的信息，则需要使用 -n 或 --namespace 指明其名称。例如，列出 kube-system 名称空间中拥有 k8s-app 标签名称的所有 Pod 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l k8s-app -n kube-system]
```

## 3. 打印资源对象的详细信息

每个资源对象都包含着用户期望的状态（Spec）和现有的实际状态（Status）两种状态信息，“kubectl get -o {yaml|json}”或“kubectl describe”命令都能够打印出指定资源对象的详细描述信息。例如，查看 kube-system 名称空间中拥有标签 component=kube-apiserver 的 Pod 对象的资源配置清单（期望的状态）及当前的状态信息，并输出为 yaml 格式，命令如下：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l component=kube-apiserver -
```

而“kubectl describe”命令还能显示与当前对象相关的其他资源对象，如 Event 或 Controller 等。例如，查看 kube-system 名称空间中拥有标签 component=kube-apiserver 的 Pod 对象的详细描述信息，可以使用下面的命令：

```
**[terminal]
**[delimiter $ ]**[command kubectl describe pods -l component=kube-apiser
```

这两个命令都支持以“TYPE NAME”或“TYPE/NAME”的格式指定具体的资源对象，如“pods kube-apiserver-master.rnkeju.com”或“pods/kube-apiserver-master.rnkeju.com”，以了解特定资源对象的详细属性信息及状态信息。

#### 4. 打印容器中的日志信息

通常一个容器中仅会运行一个进程（及其子进程），此进程作为 PID 为 1 的进程接收并处理管理信息，同时将日志直接输出至终端中，而无须再像传统的多进程系统环境那样将日志保存于文件中，因此容器日志信息的获取一般要到其控制上进行。“kubect logs”命令可打印 Pod 对象内指定容器的日志信息，命令格式为“kubectl logs [-f] [-p] (POD|TYPE/NAME) [-c CONTAINER] [options]”，若 Pod 对象内仅有一个容器，则 -c 选型及容器名为可选。例如，查看名称空间 kube-system 中仅有一个容器的 Pod 对象 kube-apiserver-master.rnkeju.com 的日志：

```
**[terminal]
**[delimiter $ ]**[command kubectl logs kube-apiserver-master.rnkeju.com
```

为上面的命令添加“-f”选项，还能用于持续监控指定容器中的日志输出，其行为类似于使用了 -f 选项的 tail 命令。

#### 5. 在容器中执行命令

容器的隔离属性使得对其内部信息的获取变得不再直观，这一点在用户需要了解容器内进程的运行特性、文件系统上的文件及路径分布等信息时，需要穿透其隔离边界进行。“kubectl exec”命令便是用于在指定的容器内运行其他应用程序的命令，例如，在 kube-system 名称空间中的 Pod 对象 kube-apiserver-masster.rnkeju.com 上的唯一容器中运行 ps 命令：

```
**[terminal]
**[delimiter $ ]**[command kubectl exec kube-apiserver-master.rnkeju.com
```

注意，若 Pod 对象中存在多个容器，则需要以 -c 选项指定容器后再运行。

## 6. 删除资源对象

使命已经完成或存在错误的资源对象可使用“`kubectl delete`”命令予以删除，不过，对于受控于控制器的对象来说，删除之后其控制器可能会重建出类似的对象，例如，`Deployment` 控制器下的 Pod 对象在被删除时就会被重建。例如，删除默认名称空间中名为 `nginx-svc` 的 Service 资源对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl delete services nginx-svc]
```

下面的命令可用于删除 `kube-system` 名称空间中拥有的标签“`k8s-app=kube-proxy`”的所有 Pod 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl delete pods -l app=monitor -n kube-sys
```

若要删除指定名称空间中的所有的某类对象，可以使用“`kubectl delete TYPE -all -n NS`”命令，例如，删除 `kube-public`名称空间中的所有 Pod 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl delete pods --all -n kube-public]
```

另外，有些资源类型（如 Pod），支持优雅删除机制，它们有着默认的删除宽限期，不过，用户可以在命令中使用`--grace-period`选项或`--now`选项来覆盖默认的宽限期。

## 2.4 命令式容器应用编排

本节将使用示例镜像“ikubernetes/myapp:v1”来演示容器应用编排的基本操作：应用部署、访问、查看、服务暴露和扩缩容等。一般来说，Kubernetes之上应用程序的基础管理操作由如下几个部分组成。

1. 通过公用的 Controller 类的资源（如 Deployment 或 ReplicationController）创建并管控 Pod 对象以运行特定的应用程序，如 Nginx 或 Tomcat 等。无状态（stateless）应用的部署和控制通常使用 Deployment 控制器进行，而有状态应用则需要使用 StatefulSet 控制器。
2. 为 Pod 对象创建 Service 对象，以便向客户端提供固定的访问路径，并借助于 CoreDNS 进行服务发现。
3. 随时按需获取各资源对象的简要或详细信息，以了解其运行状态。
4. 如有需要，则手动对支持扩缩容的 Controller 组件进行扩容或缩容；或者，为支持 HPA 的 Controller 组件（如 Deployment 或 ReplicationController）创建 HPA 资源对象以实现 Pod 副本数目的自动伸缩。
5. 滚动更新：当应用程序的镜像出现新版本时，对其执行更新操作；必要时，为 Pod 对象中的容器更新其镜像版本；并根据需要执行回滚操作。

本节中的操作示例仅演示了前三个部分的功能，即应用的部署、服务暴露及相关信息的查看。应用的扩缩容、升级及回滚等操作会再后面的章节中进行详细介绍。

以下操作命令在任何部署了 kubectl 并能正常访问到 Kubernetes 集群的主机上均可执行，包括集群外的主机。复制 master 主机上的 /etc/kubernetes/admin.conf 至相关用户主目录下的 .kube/config 文件即可正常执行，具体方法请参考 kubeadm init 命令结果中的提示。

## 2.4.1 部署应用（Pod）

在 Kubernetes 集群上自主运行的 Pod 对象再非计划内终止后，其生命周期即告以结束，用户需要再次手动创建类似的 Pod 对象才能确保容器中的应用依然可得。

对于 Pod 数量众多的场景，尤其是对微服务来说，用户必将疲于应付此类需求。

Kubernetes 的工作负载（workload）类型的控制器能够自动确保由其管控的 Pod 对象按用户期望的方式运行，因此，Pod 的创建和管理大多都会通过这种类型的控制器来进行，包括 Deployment、ReplicaSet、ReplicationController 等。

### 1. 创建 Deployment 控制器对象

“kubectl run”命令可用于命令行直接创建 Deployment 控制器，并以 --image 选项指定镜像的运行 Pod 中的容器，--dry-run 选项可用于命令的测试运行，但并未真正执行资源对象的创建过程。例如，下面的命令要创建一个名为 myapp 的 Deployment 控制器对象，它使用镜像 ikubernetes/myapp:v1 创建 Pod 对象，但仅在测试运行后即退出：

```
**[terminal]
**[delimiter $ ]**[command kubectl run myapp --image=ikubernetes/myapp:v1 --po
```

镜像 ikubernetes/myapp:v1 中定义的容器主进程为默认监听于 80 端口的 Web 服务程序 Nginx，因此，如下命令使用“--port=80”来指明容器要暴露的端口。而“--replicas=1”选项则指定了目标控制器对象要自动创建的 Pod 对象的副本数量。确认测试命令无误后，可移除“--dry-run”选项后再次执行命令以完成资源对象的创建：

```
**[terminal]
**[delimiter $ ]**[command kubectl run myapp --image=ikubernetes/myapp:v1 --po
```

创建完成后，其运行效果示意图如图 2-10 所示，它在 default 名称空间中创建了一个名为 myapp 的 Deployment 控制器对象，并由它基于指定的镜像文件创建了一个 Pod 对象。

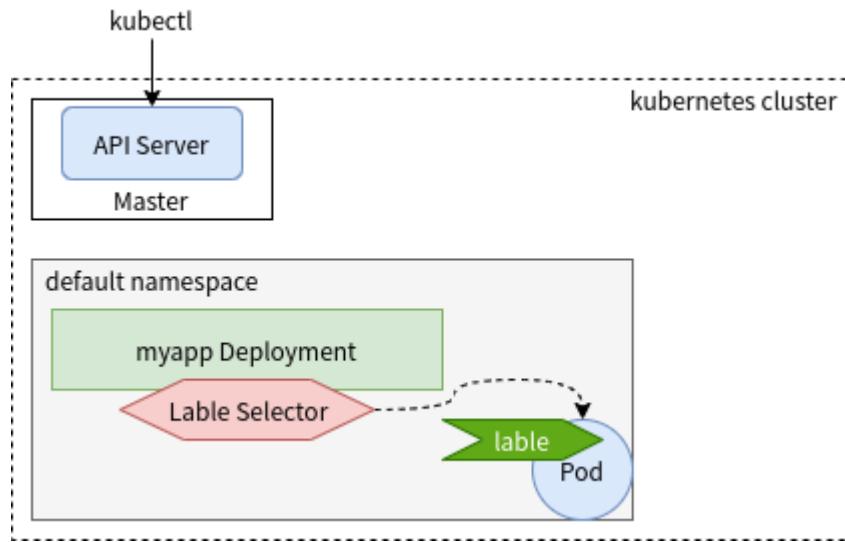


图 1.4.4.1 - Deployment 对象 myapp 及其创建的 Pod 对象

kubectl run 命令其他常用的选项还有如下几个，它们支持用户在创建资源对象时实现更多的控制，具体如下：

- -l, --labels：为 Pod 对象设定自定义标签
- --record：是否将当前的对象创建命令保存至对象的 Annotation 中，布尔型数据，其值可为 true 或 false。
- --save-config：是否将当前对象的配置信息保存至 Annotation 中，布尔型数据，其值可为 true 或 false。
- --restart=Never：创建不受控制器管控的自主式 Pod 对象。

其他可用选项及使用方式可通过“kubectl run --help”命令获取。资源对象创建完成后，通常需要了解其当前状态是否正常，以及是否能够吻合于用户期望的目标状态，相关的操作一般使用 kubectl get、kubectl describe 等命令进行。

## 2. 打印资源的相关信息

kubectl get 命令可用于获取各种资源对象的相关信息，它既能够显示对象类型特有的简要信息，也能够指定出格式为 YAML 或 JSON 的详细信息，或者使用 Go 模板自定义要显示的属性及信息等。例如，下面是查看前面创建的 Deployment 对象的相关运行状态的命令及其输出的结果：

```
**[terminal]
**[delimiter $ ]**[command kubectl get deployments]
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
myapp      1/1     1           1           4m59s
```

上面的命令的执行结果中，各字段的说明具体如下：

1. NAME：资源对象的名称

### A.1.1 部署目标

2. READY：在“/”前的数值表示为当前控制器已有的Pod对象的副本数量，“/”后的数值表示为用户期望由当前控制器管理的Pod对象副本的精确数量。
3. UP-TO-DATE：更新到最新版本定义的Pod对象的副本数量，再控制器的滚动更新模式下，它表示已经完成版本更新的Pod对象的副本数量。
4. AVAILABLE：当前处于可用状态的Pod对象的副本数量，即可正常提供服务的副本数。
5. AGE：Pod的存在时长。

Deployment 资源对象通过 ReplicaSet 控制器实例完成对 Pod 对象的控制，而非直接控制。另外，通过控制器创建的Pod对象都会被自动附加一个标签，其格式为“run=”，例如，上面的命令所创建的Pod，会拥有“run=myapp”标签。后面的章节对此会有更详细的描述。

而此Deployment控制器创建的唯一Pod对象运行正常与否，其被调用至哪个节点运行，当前是否就绪等也是用户在创建完成后应该重点关注的信息。由控制器创建的Pod对象的名称通常是以控制器名称为前缀，以随机字符串为后缀，例如，下面命令输出的结果中的myapp-5c647497bf-9km8t：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -o wide]
NAME           READY   STATUS    RESTARTS   AGE     IP
myapp-5c647497bf-9km8t   1/1     Running   0          8m53s   10.244
nginx-deployment-d86dfb797-m2vpm   1/1     Running   0          18h     10.244
```

上面命令的执行结果中，每一个字段均代表着Pod资源对象一个方面的属性，除了NAME之外的其他字段及功能说明如下：

1. READY：Pod中的容器进程初始化完成并能够正常提供服务即为就绪状态，此字段用于记录处于就绪状态的容器数量。
2. STATUS：Pod的当前状态，其值可能是Pending、Running、Succeeded、Failed 和 Unknown 等其中一种。
3. RESTARTS：Pod 对象可能会因容器进程崩溃、超出资源限额等原因发生故障问题而被重启，此字段记录了它重启的次数。
4. IP：Pod的IP地址，其通常由网络插件自动分配。
5. NODE：创建时，Pod对象会由调度器调度至急群众的某节点上运行，此字段即为节点的相关标示信息。

如果指定的名称空间中存在大量的Pod对象而使得类似如上命令的输出结果存在太多的不相关信息时，则可通过指定选项“-l run=myapp”进行Pod的对象过滤，其仅显示符合此标签选择器的Pod对象。

确认Pod对象已转为“Running”状态之后，即可于集群中的任一节点（或其他Pod对象）直接访问其容器化应用中的服务，如图2-11中节点 NodeX 上客户端程序 Client，或者集群上运行于Pod中的客户端程序。

### A.1.1 部署目标

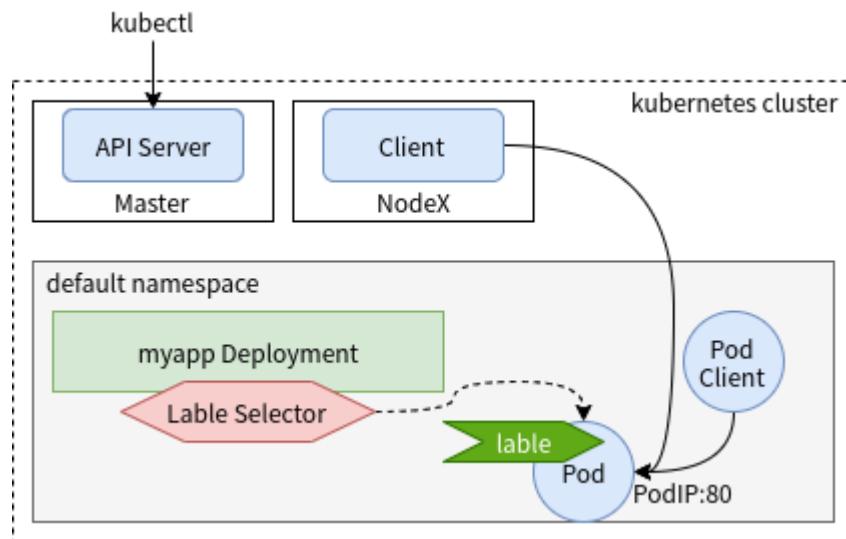


图 1.4.4.1 - 访问 Pod 中容器化应用服务程序

例如，在集群中任一节点上使用curl命令对地址为10.244.3.3的Pod对象myapp-5c647497bf-9km8t的80端口发起服务请求，命令及结果如下所示：

```
**[terminal]
**[delimiter $ ]**[command curl http://10.244.3.3:80/]
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
```

## 2.4.2 探查 Pod 及应用详情

资源创建或运行过程中偶尔会因故出现异常，此时用户需要充分获取相关状态及配置信息以便确定问题所在。另外，在对资源对象进行创建或修改完成之后，也需要通过其详细的状态来了解操作成功与否。`kubectl`有多个子命令可用于从不同的角度显示对象的状态信息，这些信息有助于用户了解对象的运行状态、属性详情等信息。

1. `kubectl describe`: 显示资源的详情，包括运行状态、事件等信息，但不同资源类型其输出的内容不尽相同。
2. `kubectl logs`: 查看Pod对象中容器输出在控制台的日志信息。在Pod中运行有多个容器时，需要使用“-c”指定容器名称。
3. `kubectl exec`: 在Pod对象某容器内运行指定的程序，其功能类似与“`docker exec`”命令，可用于了解容器各方面的相关信息或执行必需的设定等操作等，其具体功能取决于容器内可用的程序。

### 1. 查看Pod对象的详细描述

下面给出的命令打印了此前由myapp创建的Pod对象的详细状态信息，为了便于后续的多次引用，这里先将其名称保存与变量`POD_NAME`中。命令的执行结果中省略了部分输出：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command POD_NAME=myapp-5c647497bf-9km8t]
**[delimiter $ ]**[command kubectl describe pods $POD_NAME]

Name:           myapp-5c647497bf-9km8t
Namespace:      default
Priority:       0
PriorityClassName: <none>
Node:           kube-node-2.localdomain/192.168.0.183
Start Time:     Wed, 27 Nov 2019 14:38:42 +0800
Labels:         pod-template-hash=5c647497bf
                run=myapp
Annotations:    <none>
Status:         Running
IP:             10.244.3.3
Controlled By: ReplicaSet/myapp-5c647497bf
Containers:
  myapp:
    Container ID:  docker://7b5fd18032fe6dee9f9f7b144e388483c8cd592b6906ac612c
    Image:          ikubernetes/myapp:v1
    Image ID:      docker-pullable://ikubernetes/myapp@sha256:40ccda7b7e2d080t
    Port:          80/TCP
    Host Port:    0/TCP
    State:         Running
    Started:      Wed, 27 Nov 2019 14:38:47 +0800
    Ready:         True
    Restart Count: 0
    Environment:   <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fwfzr (rw)
Conditions:
  Type        Status
  Initialized  True
  Ready        True
  ContainersReady  True
  PodScheduled  True
Volumes:
  default-token-fwfzr:
    Type:           Secret (a volume populated by a Secret)
    SecretName:    default-token-fwfzr
    Optional:      false
    QoS Class:     BestEffort
    Node-Selectors: <none>
    Tolerations:   node.kubernetes.io/not-ready:NoExecute for 300s
                  node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type  Reason  Age   From           Message
  ----  -----  ----  --  -----
  Normal Scheduled  48m   default-scheduler  Successfully assig
```

```

Normal  Pulling   48m  kubelet, kube-node-2.localdomain  Pulling image "ikl
Normal  Pulled    48m  kubelet, kube-node-2.localdomain  Successfully pulled
Normal  Created   48m  kubelet, kube-node-2.localdomain  Created container
Normal  Started   48m  kubelet, kube-node-2.localdomain  Started container

```

不同的需求场景中，用户需要关注不同维度的输出，但一般来说，Events 和 Status 字段会是重点关注的对象，它们分别代表了Pod对象运行过程中的重要信息及当前状态。上面命令执行结果中的不少字段都将可以见名知义，而且部分字段再前面介绍其他命令输出时已经给出，还有一部分会在本书后面的篇幅中给予介绍。

## 2. 查看容器日志

Docker容器一般仅运行单个应用程序，其日志信息将通过标准错误输出等方式直接打印至控制台，“kubectl logs”命令即用于查看这些日志。例如，查看由 Deployment 控制器 myapp 创建的 Pod 对象的控制台日志，命令如下：

```

**[terminal]
**[delimiter $ ]**[command kubectl logs $POD_NAME]
10.244.0.0 - - [27/Nov/2019:07:15:50 +0000] "GET / HTTP/1.1" 200 65 "-" "curl/-

```

如果Pod中运行有多个容器，则需要再查看日志时为其使用“-c”选项指定容器名称。例如，当读者所部署的是KubeDNS附件而非CoreDNS时，kube-system名称空间内的kube-dns相关的Pod中同时运行这kubedns、dnsmasq、sidecar三个容器，如果要查看kubedns容器的日志，需要使用类似如下的命令：

```

**[terminal]
**[delimiter $ ]**[command DNS_POD=$(kubectl get pods -o name -n kube-system|grep kubedns)
**[delimiter $ ]**[command kubectl logs $DNS_POD -c kubedns -n kube-system]

```

需要注意的是，日志查看命令仅能用于打印存在Kubernetes系统之上的Pod中容器的日志，对于已删除的Pod对象，其容器日志信息将无从获取。日志信息是用于辅助用户获取容器中应用程序运行状态的最有效的途径之一，也是非常重要的排错手段，因此通常需要使用集中式的日志服务器统一收集存储与各Pod对象中容器的日志信息。

## 3. 在容器中运行额外的程序

运行着非交互式进程的容器中，默认运行的唯一进程及其子进程启动后，容器即进入独立、隔离的运行状态。对容器内各种详情的了解需要穿透容器边界进入其中运行其他的应用程序来进行，“kubectl exec”可以让用户在Pod的某容器中运行用户所需的任何存在于容器中的程序。在“kubectl logs”获取的信息不够全面时，此命令可

### A.1.1 部署目标

以通过再Pod中运行其他指定的命令（前提是容器中存在此程序）来辅助用户获取更多的信息。一个更便捷的使用接口的方式是直接交互式运行容器中的某个shell程序。例如，直接查看Pod中的容器运行的进程：

```
**[terminal]
**[delimiter $ ]**[command kubectl exec $POD_NAME ps aux]
PID      USER      TIME      COMMAND
 1 root      0:00 nginx: master process nginx -g daemon off;
 6 nginx      0:00 nginx: worker process
 7 root      0:00 ps aux
```

如果Pod对象中运行了多个容器，那么在程序运行时还需要使用“-c”选项指定要于其内部运行程序的容器名称。

若要进入容器的交互式Shell接口，可使用类似如下的命令，斜体部分表示在容器的交互式接口中执行的命令：

```
**[terminal]
**[delimiter $ ]**[command kubectl -it exec $POD_NAME /bin/sh]
/ # hostname
myapp-5c647497bf-9km8t
/ # netstat -tnl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 0.0.0.0:80              0.0.0.0:*              LISTEN
/ #
```

## 2.4.3 部署 Service 对象

简单来说，一个Service对象可视作通过其标签选择器过滤出的一组Pod对象，并能够为此组Pod对象监听的套接字提供端口代理及调度服务。

### 1. 创建 Service 对象

“kubectl expose”命令可用于创建Service对象已将应用程序“暴露”（expose）于网络中。例如，下面的命令即可将 myapp 创建的Pod对象使用“NodePort”类型的服务暴露到集群外面。

```
**[terminal]
**[delimiter $ ]**[command kubectl expose deployments/myapp --type="NodePort"
service/myapp exposed
```

上面的命令中，--type选项用于指定Service的类型，而 --port 则用于指定要暴露的容器端口，目标Service对象的名称为myapp。创建完成后，default 名称空间中的对象及其通信示意图如下所示：

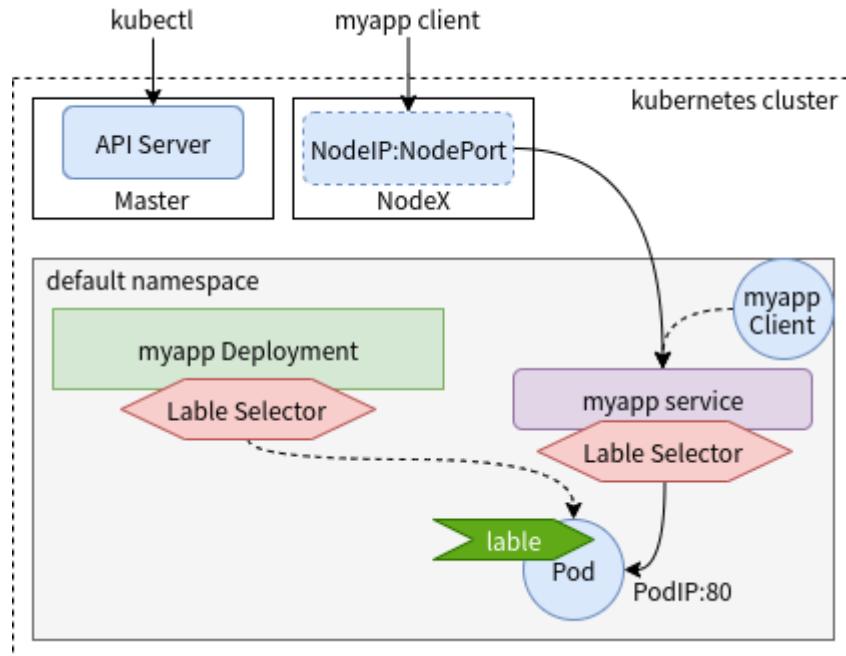


图 1.4.4.3 - Service 对象在 Pod 对象前端添加了一个固定访问层

下面通过运行于同一集群中的Pod对象中的客户端程序发起访问测试，来模拟图2-12中的源自myapp Client Pod对象的访问请求。首先，使用kubectl run 命令创建一个Pod对象，并直接接入其交互式接口，如下命令的-it组合选项即用于交互式打开

### A.1.1 部署目标

并保持其shell命令行接口；而后通过wget命令对此前创建的Service对象的名称发起访问请求，如下命令中的 myapp 即 Service 对象名称，default即其所属的 Namespace 对象的名称：

```
**[terminal]
**[delimiter $ ]**[command kubectl run client --image=busybox --restart=Never .
If you don't see a command prompt, try pressing enter.
/ # wget -O - -q http://myapp.default:80
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
/ #
```

创建时，Service对象名称及其ClientIP会由CoreDNS附件动态添加至明后才能解析库当中，因此，名称解析服务在对象创建后即可直接使用。

类似与列出 Deployment 控制器及 Pod 对象的方式，“kubectl get services” 命令能够列出 Service 对象的相关信息，例如下面的命令显示了 Service 对象 myapp 的简要状态信息：

```
**[terminal]
**[delimiter $ ]**[command kubectl get svc/myapp]
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
myapp    NodePort    10.101.202.124  <none>        80:32204/TCP   20h
```

其中，“PORT(s)”字段表明，集群中各工作节点会捕获发往本地的目标端口为32204的流量，并将其代理至当前Service对象的80端口，于是，集群外部的用户可以使用当前集群中任一节点的此端口来请求Services对象上的服务。CLUSTER-IP字段为当前Service的IP地址，它是一个虚拟IP，并没有配置与集群中的任何主机的任何接口之上，但每个node之上的kube-proxy都会为CLUSTER-IP所在的网络创建用于转发的iptables或ipvs规则。此时，用户可用于集群外部任一浏览器请求集群任一节点的相关端口来进行访问测试。

创建Service对象的另一种方式是使用“kubectl create service”命令，对应于每个类似，它分别有一个专用的子命令，例如“kubectl create service clusterip” 和 “kubectl create service nodeport”等，各命令在使用方式也略有区别。

## 2. 查看 Service 资源对象的描述

“kubectl describe services” 命令用于打印 Service 对象的详细信息，它通常包括 Service 对象的 Cluster IP，关联 Pod 对象时使用的标签选择器及关联到的 Pod 资源的端点等，示例如下：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl describe services]
Name:           myapp
Namespace:      default
Labels:         run=myapp
Annotations:    <none>
Selector:       run=myapp
Type:          NodePort
IP:            10.101.202.124
Port:          <unset>  80/TCP
TargetPort:     80/TCP
NodePort:       <unset>  32204/TCP
Endpoints:     10.244.3.3:80
Session Affinity: None
External Traffic Policy: Cluster
Events:        <none>
```

上面命令的执行结果输出基本上可以做到见名而知义，此处需要特别说明的几个字段具体如下：

1. Selector：当前Service对象使用的标签选择器，用于选择关联Pod对象。
2. Type：即Services的类型，其值可以是ClusterIP、NodePort和LoadBalancer等其中之一。
3. IP：当前Service对象的ClusterIP。
4. Port：暴露的端口，即当前Service用于接受并响应请求的端口。
5. TargetPort：容器中的用于暴露的目标端口，由 Service Port 路由请求至此端口。
6. NodePort：当前 Service 的 NodePort，它是否存在有效值与 Type 字段中的类型相关。
7. EndPoints：后端端点，即被当前Service的Selector挑中的所有Pod的IP及其端口。
8. Session Affinity：是否启用会话粘性。
9. External Traffic Policy：外部流量的调度策略。

## 2.4.4 扩容和缩容

前面示例中创建的 Deployment 对象 myapp 仅创建了一个 Pod 对象，其所能够承载的访问请求数量仅受限与这单个 Pod 对象的服务容量。请求流量上升到接近或超过其容量之前，用户可以通过 kubernetes 的“扩容机制”来扩展 Pod 的副本数量，从而提升其服务容量。

简单来说，所谓的“伸缩”(Scaling)就是指改变特定控制器上 Pod 副本数量的操作，“扩容”(scaling up) 即为增加副本数量，而“缩容”(scaling down) 则意指缩减副本数量。不过，无论是扩容还是缩容，其数量都需要用户明确给出。

Service 对象内建的复杂均衡机制可在其后端副本数不止一个时自动进行流量分发，它还会自动监控关联到的 Pod 的健康状态，以确保仅将请求流量分发至可用的后端 Pod 对象。若某 Deployment 控制器管理包含多个 Pod 实例，则必要时用户还可以为其使用“滚动更新”机制将其容器镜像升级到新的版本或变更那些支持动态修改的 Pod 属性。

使用 kubectl run 命令创建 Deployment 对象时，“--replicas=”选项能够指定由该对象创建或管理的 Pod 对象副本的数量，且其数量支持运行时进行修改，并立即生效。“kubectl scale”命令就是专用于变动控制器应用规模的命令，它支持对 Deployment 资源对象的扩容和缩容操作。例如，如果要将 myapp 的 Pod 副本数量扩展为 3 个，则可以使用如下命令来完成：

```
**[terminal]
**[delimiter $ ]**[command kubectl scale deployments/myapp --replicas=3]
deployment.extensions/myapp scaled
```

而后列出有 myapp 创建的 Pod 副本，确认其扩展操作的完成状态。如下命令显示出其 Pod 副本数量已经扩增至 3 个，其中包括此前的 myapp-5c647497bf-9km8t：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l run=myapp]
NAME          READY   STATUS    RESTARTS   AGE
myapp-5c647497bf-9km8t   1/1     Running   0          23h
myapp-5c647497bf-ksg4f   1/1     Running   0          3m6s
myapp-5c647497bf-ncjrq   1/1     Running   0          3m6s
```

Deployment 对象 myapp 规模扩展完成之后，default 名称空间中的资源对象及其关联关系如图 2-13 所示：

### A.1.1 部署目标

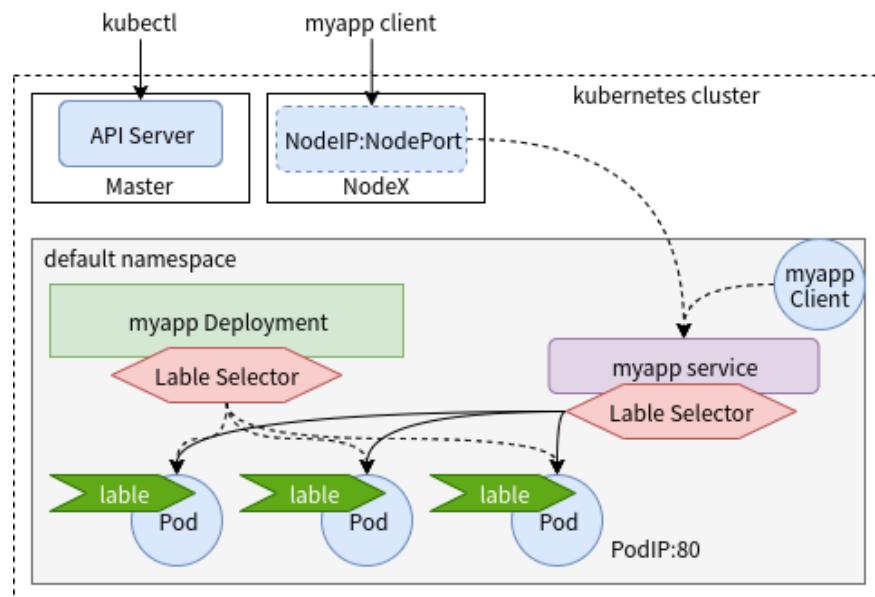


图 1.4.4.4 - Deployment 对象规模扩增完成

而后由“`kubectl describe deployment`”命令打印 Deployment 对象 myapp 的详细信息，了解其应用规模的变动及当前Pod副本的状态等相关信息。从下面的命令结果可以看出，其Pod副本数量的各项指标都已经转换到了新的目标数量，而其事件信息中也有相应的事件显示其扩增操作已成功完成：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl describe deployments/myapp]
Name:           myapp
Namespace:      default
CreationTimestamp:   Wed, 27 Nov 2019 14:38:42 +0800
Labels:          run=myapp
Annotations:    deployment.kubernetes.io/revision: 1
Selector:        run=myapp
Replicas:       3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  run=myapp
  Containers:
    myapp:
      Image:      ikubernetes/myapp:v1
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type        Status  Reason
    ----        -----  -----
    Progressing  True    NewReplicaSetAvailable
    Available   True    MinimumReplicasAvailable
    OldReplicaSets: <none>
    NewReplicaSet:  myapp-5c647497bf (3/3 replicas created)
    Events:     <none>
```

由 myapp 自动创建的Pod资源全部都拥有同一个标签选择器“run=myapp”，因此，前面创建的Service资源对象myapp的后端端点也已经通过标签选择器自动扩展到了这3个Pod对象相关的端点，如下面的命令结果及图2-13所示：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl describe services/myapp]
Name:           myapp
Namespace:      default
Labels:         run=myapp
Annotations:    <none>
Selector:       run=myapp
Type:          NodePort
IP:            10.101.202.124
Port:          <unset>  80/TCP
TargetPort:     80/TCP
NodePort:       <unset>  32204/TCP
Endpoints:     10.244.2.4:80,10.244.3.3:80,172.17.0.2:80
Session Affinity: None
External Traffic Policy: Cluster
Events:        <none>
```

回到此前创建的客户端Pod对象Client的交互式接口，对Service对象myapp反复发起测试请求，即可验证其负载均衡的效果。由如下命令及其结果可以看出，它会将请求调度至后端的各Pod对象进行处理：

```
**[terminal]
**[delimiter $ ]**[command kubectl run client --image=busybox --restart=Never .
If you don't see a command prompt, try pressing enter.
/ # while true; do wget -O - -q http://myapp.default:80/hostname.html; sleep 1;
.....
myapp-5c647497bf-9km8t
myapp-5c647497bf-ncjrq
myapp-5c647497bf-ksg4f
.....
```

应用规模缩容的方式与扩容相似，只不过是将Pod副本的数量调至比原来小的数字即可。例如，将myapp的Pod副本缩减至2个，可以使用如下命令进行：

```
**[terminal]
**[delimiter $ ]**[command kubectl scale deployments/myapp --replicas=2]
deployment.extensions/myapp scaled
```

至此，功能基本完整的容器化应用已在Kubernetes上部署完成，即便是一个略复杂的分层应用也只需要通过合适的镜像以类似的方式就能完成部署。

## 2.4.5 修改及删除对象

成功创建与Kubernetes之上的对象也称为活动对象（live object），其配置信息（live object configuration）由API Server保存于集群状态存储系统etcd中，“kubectl get TYPE NAME -o yaml”命令可以获取到相关的完整信息，而运行“kubectl edit”命令可调用默认编辑器对活动对象的可配置属性进行编辑。例如，修改此前创建的Service对象myapp的类型为ClusterIP，使用“kubectl edit service myapp”命令打开编辑器界面后修改type属性的值为ClusterIP，并删除NodePort属性，然后保存即可。对活动对象的修改将实时生效，但资源对象的有些属性并不支持运行时修改，此种情况下，编辑器将不允许保存退出。

有些命令是kubectl edit命令某一部分功能的二次封装，例如，kubectl scale命令不过是专用于修改资源对象的replicas属性值而已，它也同样直接作用于活动对象。

不再有价值的活动对象可使用“kubectl delete”命令予以删除，需要删除Service对象myapp时，使用如下命令即可完成：

```
**[terminal]
**[delimiter $ ]**[command kubectl delete service myapp]
service "myapp" deleted
```

有时候需要清空某一类型下的所有对象，只需要将上面命令对象的名称换成“--all”选项便能实现。例如，删除默认名称空间中所有的Deployment控制器的命令如下：

```
**[terminal]
**[delimiter $ ]**[command kubectl delete deployment --all]
deployment.extensions "myapp" deleted
deployment.extensions "nginx-deployment" deleted
```

需要注意的是，受控于控制器的Pod对象再删除后会被重建，删除此类对象需要直接删除控制器对象。不过，删除控制器时若不想删除其Pod对象，可在删除命令上使用“--cascade=false”选项。

虽然直接命令式管理相关功能强大且适合用于操纵kubernetes资源对象，但其明显的缺点是缺乏操作行为以及待运行对象的可信源。另外，直接命令式管理资源对象存在较大的局限性，它们在设置资源对象属性方面提供的配置能力相当有限，而且还有不少资源并不支持命令操作创建，例如，用户无法创建带有多个容器的Pod对象，也无法为Pod对象创建存储卷。因此，管理资源对象更有效的方式是基于保存有对象配置信息的配置清单来进行。

## 第三章 资源管理基础

Kubernetes 系统的API Server基于HTTP/HTTPS接受并相应客户端的操作请求，它提供了一种“基于资源”（resource-based）风格的编程接口，将集群的各种组件都抽象称为标准的REST资源，如Node、Namespace和Pod等，并支持通过标准的HTTP方式以JSON为数据序列化方案进行资源管理操作。本章将着重描述 Kubernetes的资源管理方式。

## 3.1 资源对象及API群组

REST是Representational State Transfer的缩写，意为“表征状态转移”，它是一种程序架构风格，基本元素为资源（resource）、表征（representation）和行为（action）。资源即对象，一个资源通常意味着一个附带类型和关联数据、支持的操作方法以及与其它对象的关系的对象，它们是持有状态的事物，即REST中的S（State）。REST组件通过使用“表征”来捕获资源的当前或预期的状态并在组件之间传输该表征从而对资源执行操作。表征是一个字节序列，由数据、描述数据的元数据以及偶尔描述元数据的元数据组成（通常用于验证消息的完整性），表征还有一些其他常用但不太精确的名称，如文档、文件和HTTP消息实体等。表征的数据格式被称为媒体类型（media type），常用的有JSON或XML。API客户端不能直接访问资源，它们需要执行“动作”（action）来改变资源的状态，于是资源的状态从一种形态“转移”（Transfer）为另一种形式。

资源可以分组为集合（collection），每个集合值包含单一类型的资源，并且各资源间是无序的，当然资源可以不属于任何集合，它们称为单体资源。事实上，集合本身也是资源，它可以部署与全局级别，位于API的顶层，也可以包含与某个资源中，表现为“子集合”。集合、资源、子集合及子资源间的关系如图3-1所示。

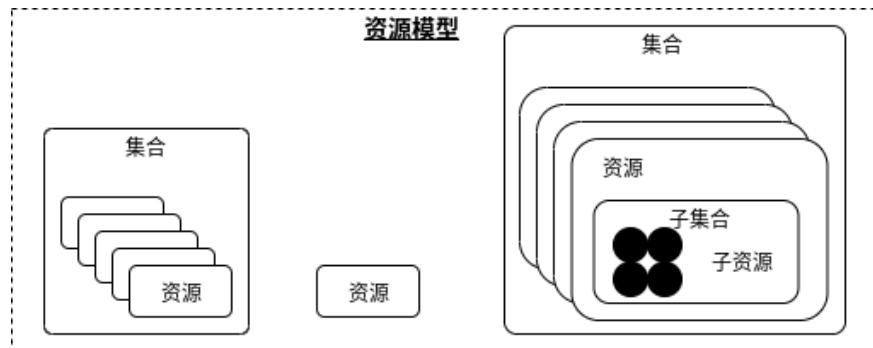


图 1.5.1 - 集合、资源和子资源

Kubernetes 系统将一切事物都抽象为API资源，其遵循REST架构风格组织并管理这些资源及对象，同时还支持通过标准的HTTP方法（POST、PUT、PATCH、DELETE和GET）对资源进行增、删、改、查等管理操作。不过，在Kubernetes系统的语境中，“资源”用于表示“对象”的集合，例如，Pod资源可用于描述所有Pod类型的对象，但本书将不加区别地使用资源、对象和资源对象，并将它们统统理解为资源类型生成的示例——对象。

### 3.1.1 Kubernetes 的资源对象

依据资源的主要功能作为分类标准。Kubernetes的API对象大体可分为工作负载（Workload）、发现和负载均衡（Discovery & LB）、存储和配置（Config & Storage）、集群（Cluster）以及元数据（Metadata）五个类别。它们基本上都是围绕一个核心目的而设计：如何更好的运行和丰富Pod资源，从而为容器化应用提供更灵活、更完善的操作与管理组件，如图3-2所示：

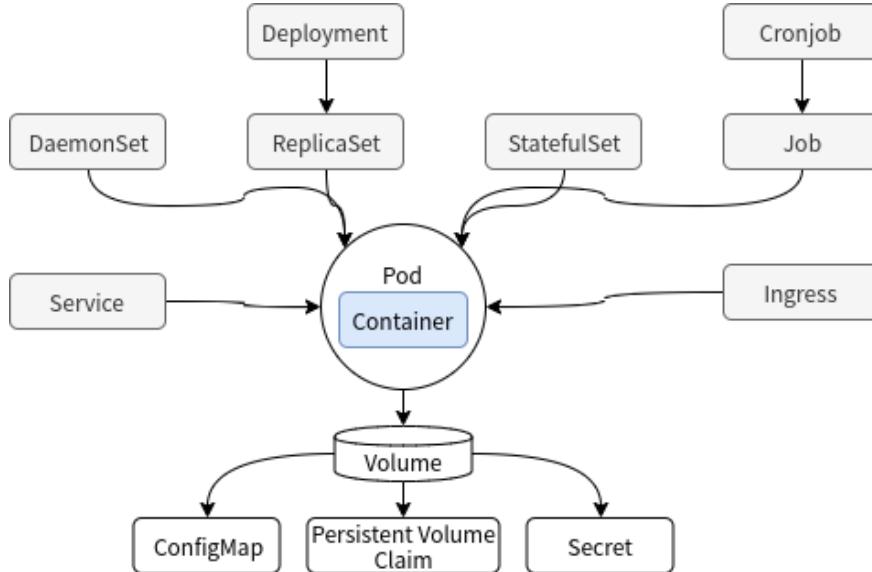


图 1.5.1.1 - Kubernetes 常用资源对象

工作负载型资源用于确保Pod资源对象能够更好地运行容器化应用，具有同一种负载的各Pod对象需要以负载的方式服务于各请求，而各种容器化应用彼此之间需要彼此“发现”以完成工作协同。Pod资源具有生命周期，存储型资源能够为重构的Pod对象提供持久化的数据存储机制，共享同一配置的Pod资源可从配置型资源中统一获取配置改动信息，这些资源作为配置中心为管理容器化应用的配置文件提供了极为便捷的管理机制。集群型资源为管理集群本身的工作特性提供了配置接口，而元数据型资源则用于配置集群内部的其他资源的行为。

#### 1. 工作负载型资源

Pod 是工作负载型资源中的基础资源，它负责运行容器，并为其解决环境的依赖，例如，向容器内注入共享的或持久化的存储卷、配置信息或秘钥数据等。但Pod可能会因为资源超限或节点故障等原因而终止，这些非正常终止的Pod资源需要被重建，不过，这类工作将由工作负载型的控制器来完成，它们通常也称为Pod控制器。

应用程序分为无状态和有状态两种类型，它们对环境的依赖及工作特性有很大的不同，因此分属两种不同类型的Pod控制器来管理，ReplicationController、ReplicaSet和Deployment负责管理无状态应用，StatefulSet用于管控有状态类应用。ReplicationController是上一代的控制器，其功能由ReplicaSet和Deployment负责实现，因此几近于废弃。还有些应用较为独特，它们需要在

集群中的每个节点上运行单个Pod资源，负责收集日志或运行系统服务等任务，这些Pod资源的管理则属于DaemonSet控制器的分内之事。另外，有些容器化应用需要继续运行以为守护进程不间断地提供服务，而有些则因该在正常完成后退出，这些在正常完成后就应该退出的容器化应用则由Job控制器负责管控。下面是各Pod控制器更为详细的说明。

- ReplicationController：用于确保每个Pod副本在任一时刻均能满足目标数量，换言之，用于保证每个容器或容器组总是运行并且可访问；它是上一代的无状态Pod应用控制器，建议读者使用新型控制器Deployment和ReplicaSet来取代它。
- ReplicaSet：新一代ReplicationController，它与ReplicationController的唯一不同之处仅在于支持的标签选择器不同，ReplicationController只支持等值选择器，而ReplicaSet还额外支持基于集合的选择器。
- Deployment：用于管理无状态的持久化应用，例如HTTP服务器；它用于为Pod和ReplicaSet提供声明式更新，是构建在ReplicaSet之上更为高级的控制器。
- StatefulSet：用于管理无状态的持久化应用，如Database服务程序；其与Deployment的不同之处在于StatefulSet会为每个Pod创建一个独有的持久性标识符，并会确保各Pod之间的顺序性。
- DaemonSet：用于确保每个节点都运行了某Pod的一个副本，新增的节点一样会被添加此类Pod；在节点移除时，此类Pod会被回收；DaemonSet常用于运行集群存储守护进程——如glusterd和ceph，还有日志收集进程——如fluentd和logstash，以及监控进程——如Prometheus的Node exporter、collectd、Datadog agent和Ganglia的gmond等。
- Job：用于管理运行完成后即可终止的应用，例如批处理作业任务；换句话说，Job创建一个或多个Pod，并确保其符合目标数量，直到Pod正常结束而中。

## 2. 发现和负责均衡

Pod资源可能会因为任何意外故障而被重建，于是它需要固定的可被“发现”的方式。另外，Pod资源仅在集群内可见，它的客户端也可能是集群内的其他Pod资源，若要开放给外部网络中的用户访问，则需要事先将其暴露到集群外部，并且要为同一种工作负载的访问流量进行负载均衡。Kubernetes使用标准的资源对象来解决此类问题，它们是用于为工作负载添加发现机制及负载均衡功能的Service资源和Endpoint资源，以及通过七次代理实现请求流量负载均衡的Ingress资源。

## 3. 配置和存储

Docker容器分层联合挂载的方式决定了不宜再容器内部存储需要持久化的数据，于是它通过引入挂载外部存储卷的方式来解决此类问题，而Kubernetes为此设计了Volume资源，它支持众多类型的存储设备和存储系统，如GlusterFS、CEPH RBD和Flocker等，另外，新版本的kubernetes还支持通过标准的CSI(Container Storage interface)统一存储接口以及扩展支持更多类型的存储系统。

另外，基于镜像构建容器应用时，其配置信息于镜像制作时焙入，从而为不同的环境定制配置就变得比较困难。Docker使用环境变量等作为解决方案，但这么一来就得于容器启动时将值传入，且无法在运行时修改。ConfigMap资源能够以环境变量或存储卷的方式接入到Pod资源的容器中，并且被多个同类的Pod共享引用，从而实现“一次修改，多处生效”。不过，这种方式不适用于存储敏感数据，如私钥、密码等，那是另一个资源类型Secret的功能。

#### 4. 集群级资源

Pod、Deployment、Service和ConfigMap等资源属于名称空间级别，可由相应的项目管理员所管理。然而，Kubernetes还存在一些集群级别的资源，用于定义集群自身配置信息的对象，它们仅应该有集群管理员进行操作。集群级资源主要包含以下集中类型。

- Namespace：资源对象名称的作用范围，绝大多数对象都隶属与某个名称空间，默认隶属于“default”。
- Node：Kubernetes集群的工作节点，其标识符再当前的集群中必须是唯一的。
- Role：名称空间级别的有规则组成的权限集合，可被RoleBinding引用。
- ClusterRole：Cluster级别的由规则组成的权限集合，可被Rolebinding和ClusterRoleBinding引用。
- RoleBinding：将Role中的许可权限绑定在一个或一组用户之上，它隶属与且仅能作用于一个名称空间；绑定时，可以引用同一名称空间的Role，也可以引用全局名称空间中的ClusterRole。
- ClusterRoleBinding：将ClusterRole中定义的许可权限绑定在一个或一组用户之上；它能够引用全局名称空间中的ClusterRole，并能通过Subject添加相关信息。

#### 5. 元数据型资源

此类资源对象用于为集群内部的其他资源配置其行为或特性，如HorizontalPodAutoscaler资源可用于自动伸缩工作负载类型的资源对象的规模，Pod模板资源可用于为Pod资源的构件预制模板，而LimitRange则可以为名称空间的资源设置其CPU和内存等系统级资源的数量限制等。

一个应用通常需要多个资源的支撑，例如，使用Deployment资源管理应用实例（Pod）、使用ConfigMap资源保存应用配置、使用Service和Ingress资源暴露服务、使用Volume资源提供外部存储等。

本书后面篇幅的主题部分就展开介绍这些资源类型，它们是将容器化应用托管运行于Kubernetes集群的重要工具组件。

### 3.1.2 资源及其在API中的组织形式

在Kubernetes上，资源对象代表了系统中的持久类实体，Kubernetes用这些持久类实体来表达集群的状态，包括容器化的应用程序正运行于哪些节点，每个应用程序有哪些资源可用，以及每个应用程序各自的行为策略，如重启、升级及容错策略等。一个对象可能会包含多个资源，用户可对这些资源执行增、删、改、查等管理操作。Kubernetes通常利用标准的RESTful术语来描述API概念。

- 资源类型（resource type）是指在URL中使用的名称，如Pod、Namespace和Service等，其URL格式为“GROUP/VERSION/RESOURCE”，如app/v1/deployment。
- 所有资源类型都有一个对应的JSON表示格式，称为“种类”（kind）；客户端创建对象必须以JSON提交对象的配置信息。
- 隶属于同一种资源类型的对象组成的列表称为“集合”（collection），如PodList。
- 某种类型的单个实例称为“资源”（resource）或“对象”（object），如名为pod-demo的Pod对象。

kind 代表着资源对象所属的类型，如Namespace、Deployment、Service及Pod等，而这些资源类型大体又可以分为三个类别，具体如下：

- 对象（object）类：对象表示kubernetes系统上的持久化实例，一个对象可能包含多个资源，客户端可用它执行多种操作。Namespace、Deployment、Service及Pod都属于这个类别。
- 列表（list）类：列表通常是指同一类型资源的集合，如PodLists、 NodeList等。
- 简单（Simple）类：常用于在对象上执行某种特殊操作，或者管理非持久化的实体，如/binding或/status等。

Kubernetes绝大多数的API资源类型都是“对象”，它们代表着集群中某个概念的实例。有一小部分的API资源类型为“虚拟”（virtual）类型，它们用于表达一类“操作”（operation）。所有的对象类型都拥有一个独有的名称标识以实现其幂等的创建及获取操作，不过，虚拟型资源无须获取或不依赖与幂等性时也可以不使用专用标识符。

有些资源类型隶属于集群范畴，如Namespace和PersistentVolume，而多数资源类型则受限于名称空间，如Pod、Deployment和Service等。名称空间级别的资源的URL路径中含有其所属空间的名称，这些资源对象在名称空间中被删除时会被一并删除，并且这些资源对象的访问也将受控于其所属的名称空间级别的授权审查。

Kubernetes 将API分割为多个逻辑组合、称为API群组，它们支持单独启用或禁用，并能够再次分解。API Server 支持在不同的群组中使用不同的版本，允许各组以不同的速度演进，而且也支持同一群组同时存在不同版本，如apps/v1、apps/v1beta2和apps/v1beta1，也因此能够在不同的群组中使用同名的资源类型，从而能在稳定版本的群组及新的实验群组中以不同的特性同时使用同一个资源类型。群组化管理的API使得其可以更轻松地进行扩展。当前系统的API Server上的相关信息可通过“kube api-versions”命令获取。命令结果中显示的不少API群组在后续的章节中配置资源清单时会多次用到：

## A.1.1 部署目标

```
**[terminal]
**[prompt root@master]**[path ~]**[delimiter $ ]**[command kubectl api-version
admissionregistration.k8s.io/v1beta1
apiextensions.k8s.io/v1beta1
apiregistration.k8s.io/v1
apiregistration.k8s.io/v1beta1
apps/v1
apps/v1beta1
apps/v1beta2
authentication.k8s.io/v1
authentication.k8s.io/v1beta1
authorization.k8s.io/v1
authorization.k8s.io/v1beta1
autoscaling/v1
autoscaling/v2beta1
autoscaling/v2beta2
batch/v1
batch/v1beta1
certificates.k8s.io/v1beta1
coordination.k8s.io/v1
coordination.k8s.io/v1beta1
devices.kubeedge.io/v1alpha1
events.k8s.io/v1beta1
extensions/v1beta1
networking.k8s.io/v1
networking.k8s.io/v1beta1
node.k8s.io/v1beta1
policy/v1beta1
rbac.authorization.k8s.io/v1
rbac.authorization.k8s.io/v1beta1
scheduling.k8s.io/v1
scheduling.k8s.io/v1beta1
storage.k8s.io/v1
storage.k8s.io/v1beta1
v1
```

kubernetes 的 API 以层级结构组织在一起，每个API群组表现为一个以“/apis”为根路径的REST路径，不过核心群组core有一个专用的简化路径“/api/v1”。目前，常用的API群组可归为如下两类：

- **核心群组 (core group)** : REST 路径为/api/v1，在资源的配置信息 apiVersion 字段中引用时间可以不指定路径，而仅给出版本，如 “apiVersion:v1”。
- **命名的群组 (named group)** : REST 路径为 /api/\$GROUP\_NAME/\$VERSION，例如/apis/apps/v1，它在apiVersion字段中引用的格式为“apiVersion:\$GROUP\_NAME/\$VERSION”，如

### A.1.1 部署目标

“apiVersion:apps/v1”。

总结起来，名称空间级别的每一个资源类型在API中的URL路径表示都可简单抽象为形如： /apis/<group>/<version>/namespaces/<namespace>/<kind-plural> 的路径，如default名称空间中Deployment类型的路径为/apis/apps/v1/namespaces/default/deployments，通过此路径可获取到 default 名称空间中所有的 Deployment 对象的列表。

```
**[terminal]
**[delimiter $ ]**[command kubectl get --raw /apis/apps/v1/namespaces/default/deployments]
{
  "kind": "DeploymentList",
  "apiVersion": "apps/v1",
  "metadata": {
    "selfLink": "/apis/apps/v1/namespaces/default/deployments",
    "resourceVersion": "1244879"
  },
  "items": []
}
```

另外，Kubernetes 还支持用户自定义资源类型，目前常用的方式有三种：一是修改Kubernetes源代码自定义类型；二是创建一个自定义的 API Server，并经其聚合至集群内；三是使用自定义资源（Custom Resource Definition，CRD）。

### 3.1.3 访问kubernetes REST API

以编程的方式访问 Kubernetes REST API 有助于了解其流程化的集群管理机制，一种常用的方式是使用curl作为HTTP客户端直接通过API Server在集群上操作资源对象模拟请求和响应的过程。不过，由 kubeadm 部署 Kubernetes 集群默认仅支持 HTTPS的访问接口，它需要一系列的认证检查，还在用户也可以借助 kubectl proxy 命令在本地主机上为API Server启动一个代理网关，由它支持使用HTTP进行通讯，其工作逻辑如图3-3所示。

例如，于本地 127.0.0.1 的 8080 端口上启动 API Server 的一个代理网关：

```
**[terminal]
**[delimiter $ ]**[command kubectl proxy --port=8080]
Starting to serve on 127.0.0.1:8080
```

而后即可于另一终端使用 curl 一类的客户端工具对此套接字地址发起访问请求，例如，请求 kubernetes 集群上的NamespaceList资源对象，即列出集群上所有的 Namespace对象：

```
**[terminal]
**[delimiter $ ]**[command curl localhost:8080/api/v1/namespaces/]
{
  "kind": "NamespaceList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/",
    "resourceVersion": "1356011"
  },
}
```

或者使用JSON的命令行处理器jq命令对响应的JSON数据流进行内容过滤，例如，下面的命令仅用于显示相关的NamespaceList对象中各成员对象：

```
**[terminal]
**[delimiter $ ]**[command curl -s localhost:8080/api/v1/namespaces/ | jq .items
"default"
" kube-node-lease"
" kube-public"
" kube-system"
```

给出特定的 Namespace 资源对象的名称则能够直接获取相对应的资源信息，以 Kube-system 名称空间为例：

### A.1.1 部署目标

```
**[terminal]
*[delimiter $ ]*[command curl -s localhost:8080/api/v1/namespaces/kube-syste
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "kube-system",
    "selfLink": "/api/v1/namespaces/kube-system",
    "uid": "0af98bf7-0ba2-11ea-9b5d-000c29d3bc46",
    "resourceVersion": "16",
    "creationTimestamp": "2019-11-20T14:28:38Z"
  },
  "spec": {
    "finalizers": [
      "kubernetes"
    ],
    "status": {
      "phase": "Active"
    }
  }
}
```

上述命令相应的结果中展现了 Kubernetes 大多数资源对象的资源配置格式，它是一个 JSON 序列化的数据结构，具有 kind、apiVersion、metadata、spec 和 status 五个一级字段，各字段的意义和功能将在3.2节中重点介绍。

## 3.2 对象类资源格式

Kubernetes API 仅接受及响应 JSON 格式的数据（JSON对象），同时，为了便于使用，它也允许用户提供YAML个数的POST对象，但API Server需要事先自定将其转换为JSON格式后方能提交。API Server接受和返回的所有JSON对象都遵循同一个模式，它们都具有 kind 和 apiVersion 字段，用于标识对象所属的资源类型、API 群组及相关的版本。

进一步地，大多数的对象或列表类型的资源还需要具有是三个嵌套型的字段 metadata、spec 和 status。其中 metadata 字段为资源提供元数据信息，如名称、隶属的空间名称和标签等；spec 则用于定义用户期望的状态，不同的资源类型，其状态的意义也各有不同，例如 Pod 资源最为核心的功能在于运行容器；而status 则记录着活动对象的当前状态信息，它由 kubernetes 系统自行维护，对用户来说为只读字段。

每个资源通常仅接受并返回单一类型的数据，而一种类型可以被多个反映特定用例的资源所接受或返回。例如对于 Pod 类型的资源来说，用户可创建、更新或删除 Pod对象，然而，每个Pod对象的metadata、spec和status字段的值却又是各自独立的对象型数据，它们可被单独操作，求其是 status 对象，是由 kubernetes 系统单独进行自动更新，而不能由用户手动操作它。

### 3.2.1 资源配置清单

3.1 节中曾使用 curl 命令通过代理的方式于 API Server 上请求到了 kube-system 这个 Namespace 活动对象的状态信息，事实上，用户也可以直接使用“kubectl get TYPE/NAME -o yaml”命令获取任何一个对象的 YAML 格式的配置清单，或者使用“kubectl get TYPE/NAME -o json”命令获取 JSON 格式的配置清单。例如，可使用下面的命令获取 kube-system 的状态：

```
**[terminal]
**[delimiter $ ]**[command kubectl get namespace kube-system -o yaml]
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: "2019-11-20T14:28:38Z"
  name: kube-system
  resourceVersion: "16"
  selfLink: /api/v1/namespaces/kube-system
  uid: 0af98bf7-0ba2-11ea-9b5d-000c29d3bc46
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```

除了极少数的资源之外，Kubernetes 系统上的绝大多数资源都是由其使用者所创建的。创建时，需要以与上诉输出结果中类似的方式以 YAML 或 JSON 序列化方案定义资源的相关配置数据，即用户期望的目标状态，而后再由 Kubernetes 的底层组件确保活动对象的运行时状态与用户提供的配置清单中定义的状态无限接近。因此，资源的创建要通过用户提供的资源配置清单来进行，其格式类似与 kubectl get 命令获取到的 YAML 或 JSON 形式的输出结果。不过，status 字段对用户来说为只读字段，它由 Kubernetes 集群自动维护。例如，下面就是一个创建 Namespace 资源时提供的资源配置清单示例，它仅提供了几个必要的字段：

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
spec:
  finalizers:
  - kubernetes
```

将如上所述的配置清单中的内容保存于文件中，使用“kubectl create -f /PATH/TO/FILE”命令即可将其创建到集群中。创建完成后查看其 YAML 或 JSON 格式的输出结果，可以看到 Kubernetes 会补全其大部分的字段，并提供相应的数据。事实上，Kubernetes 的大多数资源都能够以类似的方式进行创建和查看，而且它们几乎都遵循类似的组织结构，下面的命令显示了第二章中使用 kubectl run 命令创建的 Deployment 资源对象 myapp 的状态信息：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl get deployment myapp -o yaml]
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: "2019-11-30T11:31:38Z"
  generation: 1
  labels:
    run: myapp
  name: myapp
  namespace: default
  resourceVersion: "1362322"
  selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/myapp
  uid: f8ffa298-1364-11ea-bba9-000c29d3bc46
spec:
  progressDeadlineSeconds: 600
  replicas: 3
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      run: myapp
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: myapp
    spec:
      containers:
        - image: ikubernetes/myapp:v1
          imagePullPolicy: IfNotPresent
          name: myapp
          ports:
            - containerPort: 80
              protocol: TCP
          resources: {}
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
```

### A.1.1 部署目标

```
terminationGracePeriodSeconds: 30

status:
  availableReplicas: 3
  conditions:
    - lastTransitionTime: "2019-11-30T11:31:49Z"
      lastUpdateTime: "2019-11-30T11:31:49Z"
      message: Deployment has minimum availability.
      reason: MinimumReplicasAvailable
      status: "True"
      type: Available
    - lastTransitionTime: "2019-11-30T11:31:38Z"
      lastUpdateTime: "2019-11-30T11:31:49Z"
      message: ReplicaSet "myapp-5c647497bf" has successfully progressed.
      reason: NewReplicaSetAvailable
      status: "True"
      type: Progressing
  observedGeneration: 1
  readyReplicas: 3
  replicas: 3
  updatedReplicas: 3
```

从命令的结果可以看出，它也遵循 kubernetes API 标准的资源组织格式，由 apiVersion、kind、metadata、spec 和 status 五个核心字段组成，只是 spec 字段中嵌套的内容与 Namespace 资源几乎完全不同。

事实上，对几乎所有的资源来说，apiVerison、kind 和 metadata 字段的功能基本上都是相同的，但 spec 则用于资源的期望状态，而资源之所以存在类型上的不同，也在于它们的嵌套属性存在显著差别，它由用户定义和维护。而 status 字段则用于记录活动对象的当前状态，它要与用户在 spec 中定义的期望状态相同，或者正处于转换为与其相关的过程中。

## 3.2.2 metadata 嵌套字段

metadata 字段用于描述对象的属性信息，其内嵌多个字段用于定义资源的元数据，例如 name 和 labels 等，这些字段大体可分为必要字段和可选字段两大类。名称空间级别的资源的必选字段包括如下三项：

- namespace：指定当前对象隶属的名称空间，默认值为 default。
- name：设定当前对象的名称，在其所属的名称空间的同一类型中必须唯一。
- uid：当前对象的唯一标识符，其唯一性仅发生在特定的时间段和名称空间中：此标识符主要是用于区别拥有同样名字的“已删除”和“重新创建”的同一个名称的对象。

可选字段通常是指由 Kubernetes 系统自行维护的设置，或者存在默认，或者本身允许使用空值等类型的字段，常用的有如下几个：

- labels：设定用于标识当前对象的标签，键值数据，常被用作挑选条件。
- annotations：非标识型键值数据，用来作为挑选条件，用于 labels 的补充。
- resourceVersion：当前对象的内部版本标识符，用于让客户端确定对象变动与否。
- generation：用于标识当前对象目标状态的代别。
- creationTimestamp：当前对象创建日期的时间戳。
- deletionTimestamp：当前对象删除日期的时间戳。

此外，用户通过配置清单创建资源时，通常仅需要给出必选字段，可选字段可按需指定，对于用户未明确定义的嵌套字段，则需要由一系列的 finalizer 组件自动予以填充。而用户需要对资源创建的目标资源对象进行强制校验，或者在修改时需要用到 initializer 组件完成，例如，为每个待创建的 Pod 对象添加一个 Sidecar 容器等。不同的资源类型也会存在一些专有的嵌套字段，例如，ConfigMap 资源还支持使用 clusterName 等。

### 3.2.3 spec 和 status 字段

Kubernetes 用 spec 来描述所期望的对象应该具有的状态，而用 status 字段来记录对象在系统上的当前状态，因此 status 字段仅对活动对象才有意义。这两个字段都属于嵌套类型的字段。在定义资源配置清单时，spec是必须定义的字段，用于描述对象的目标状态，即用户期望对象需要表现出来的特征。status字段则记录了对象的当前状态（或实际状态），此字段值由 Kuberne te 系统负责填充或更新，用户不能手动进行定义。Master 的 controller-manager 通过相应的控制器组件动态管理并确保对象的实际状态匹配用户所期望的状态，他是一种调和（reconciliation）配置系统。

例如，Deployment 是一种用于描述集群中运行的应用的对象，因此，创建 Deployment 类型的对象时，需要为目标 Deployment 对象设定 spec，指定期望需要运行的pod副本数量、使用的标签选择器以及pod模板等。Kubernetes 系统读取待创建的 Deployment 对象的量、使用的标签选择器以及Pod模板等。Kubernetes 系统读取待创建的Deployment对象的spec以及系统上相应的活动对象的当前状态，必要时进行对象更新以确保status字段吻合spec字段中期望的状态。如果这其中任一实例出现了问题（status字段发生了变化），那么Kubernetes系统则需要及时对spec和status字段的差异做出响应，例如，补足缺失的Pod副本数目等。

spec字段嵌套的字段对于不同的对象类型来说各不相同，具体需要参照Kubernetes API参考手册中的说明分别进行获取，核心资源对象的常用配置字段将会再本书后面的章节中进行讲解。

### 3.2.4 资源配置清单格式文档

定义资源配置清单时，监管apiVersion、kind 和 metadata 都有章可循，但 spec 字段对不同的资源来说确是千差万别的，因此用户需要参考Kubernetes API的参考文档来了解各种可用属性字段。好在，Kubernetes 在系统上内建了相关的文档，用户可以使用“kubectl explain”命令直接获取相关的使用帮助，它将根据给出的对象类型或相应的嵌套字段来显示相关的下一级文档。例如，要了解Pod资源的一级字段，可以使用类似如下的命令，命令结果会输出支持使用的各一组字段及其说明：

```
**[terminal]
**[delimiter $ ]**[command 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/api-conventions.md#resource-kind-version

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/api-conventions.md#types-kinds

metadata  <Object>
Standard object's metadata. More info:
https://git.k8s.io/community/contributors/devel/api-conventions.md#metadata

spec  <Object>
Specification of the desired behavior of the pod. More info:
https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-api

status  <Object>
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/api-conventions.md#status
```

### A.1.1 部署目标

需要了解某一字段表示的对象之下的二级对象字段时，只需要指定其一级字段的对象名称即可，三级和四级字段对象等的查看方式以此类推。例如查看Pod资源的Spec对象支持嵌套使用的二级字段，可使用类似如下的命令：

## A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl explain pods.spec]
KIND: Pod
VERSION: v1

RESOURCE: spec <Object>

DESCRIPTION:
  Specification of the desired behavior of the pod. More info:
  https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-api-configuration

  PodSpec is a description of a pod.

FIELDS:
  activeDeadlineSeconds <integer>
    Optional duration in seconds the pod may be active on the node relative to its
    StartTime before the system will actively try to mark it failed and kill
    associated containers. Value must be a positive integer.

  affinity <Object>
    If specified, the pod's scheduling constraints

  automountServiceAccountToken <boolean>
    AutomountServiceAccountToken indicates whether a service account token
    should be automatically mounted.

  containers <[]Object> -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 <Object>
    Specifies the DNS parameters of a pod. Parameters specified here will be
    merged to the generated DNS configuration based on DNSPolicy.

  dnsPolicy <string>
    Set DNS policy for the pod. Defaults to "ClusterFirst". Valid values are
    'ClusterFirstWithHostNet', 'ClusterFirst', 'Default' or 'None'. DNS
    parameters given in DNSConfig will be merged with the policy selected with
    DNSPolicy. To have DNS options set along with hostNetwork, you have to
    specify DNS policy explicitly to 'ClusterFirstWithHostNet'.

  enableServiceLinks <boolean>
    EnableServiceLinks indicates whether information about services should be
    injected into pod's environment variables, matching the syntax of Docker
    links. Optional: Defaults to true.

  hostAliases <[]Object>
```

## A.1.1 部署目标

```
HostAliases is an optional list of hosts and IPs that will be injected into
the pod's hosts file if specified. This is only valid for non-hostNetwork
pods.

hostIPC    <boolean>
Use the host's ipc namespace. Optional: Default to false.

hostNetwork   <boolean>
Host networking requested for this pod. Use the host's network namespace.
If this option is set, the ports that will be used must be specified.
Default to false.

hostPID     <boolean>
Use the host's pid namespace. Optional: Default to false.

hostname    <string>
Specifies the hostname of the Pod If not specified, the pod's hostname will
be set to a system-defined value.

imagePullSecrets  <[]Object>
ImagePullSecrets is an optional list of references to secrets in the same
namespace to use for pulling any of the images used by this PodSpec. If
specified, these secrets will be passed to individual puller
implementations for them to use. For example, in the case of docker, only
DockerConfig type secrets are honored. More info:
https://kubernetes.io/docs/concepts/containers/images#specifying-imagepullsecrets

initContainers  <[]Object>
List of initialization containers belonging to the pod. Init containers are
executed in order prior to containers being started. If any init container
fails, the pod is considered to have failed and is handled according to its
restartPolicy. The name for an init container or normal container must be
unique among all containers. Init containers may not have Lifecycle
actions, Readiness probes, or Liveness probes. The resourceRequirements of
an init container are taken into account during scheduling by finding the
highest request/limit for each resource type, and then using the max of one
of that value or the sum of the normal containers. Limits are applied to init
containers in a similar fashion. Init containers cannot currently be added
or removed. Cannot be updated. More info:
https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

nodeName    <string>
nodeName is a request to schedule this pod onto a specific node. If it is
non-empty, the scheduler simply schedules this pod onto that node, assuming
that it fits resource requirements.

nodeSelector  <map[string]string>
NodeSelector is a selector which must be true for the pod to fit on a node.
```

## A.1.1 部署目标

```
Selector which must match a node's labels for the pod to be scheduled on
that node. More info:
https://kubernetes.io/docs/concepts/configuration/assign-pod-node/

priority <integer>
The priority value. Various system components use this field to find the
priority of the pod. When Priority Admission Controller is enabled, it
prevents users from setting this field. The admission controller populates
this field from PriorityClassName. The higher the value, the higher the
priority.

priorityClassName <string>
If specified, indicates the pod's priority. "system-node-critical" and
"system-cluster-critical" are two special keywords which indicate the
highest priorities with the former being the highest priority. Any other
name must be defined by creating a PriorityClass object with that name. If
not specified, the pod priority will be default or zero if there is no
default.

readinessGates <[]Object>
If specified, all readiness gates will be evaluated for pod readiness. A
pod is ready when all its containers are ready AND all conditions specified
in the readiness gates have status equal to "True" More info:
https://git.k8s.io/enhancements/keps/sig-network/0007-pod-ready%2B%2B.md

restartPolicy <string>
Restart policy for all containers within the pod. One of Always, OnFailure,
Never. Default to Always. More info:
https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy

runtimeClassName <string>
RuntimeClassName refers to a RuntimeClass object in the node.k8s.io group,
which should be used to run this pod. If no RuntimeClass resource matches
the named class, the pod will not be run. If unset or empty, the "legacy"
RuntimeClass will be used, which is an implicit class with an empty
definition that uses the default runtime handler. More info:
https://git.k8s.io/enhancements/keps/sig-node/runtime-class.md This is an
alpha feature and may change in the future.

schedulerName <string>
If specified, the pod will be dispatched by specified scheduler. If not
specified, the pod will be dispatched by default scheduler.

securityContext <Object>
SecurityContext holds pod-level security attributes and common container
settings. Optional: Defaults to empty. See type description for default
values of each field.
```

### A.1.1 部署目标

```
serviceAccount      <string>
DeprecatedServiceAccount is a deprecated alias for ServiceAccountName.
Deprecated: Use serviceAccountName instead.

serviceAccountName    <string>
ServiceAccountName is the name of the ServiceAccount to use to run this
pod. More info:
https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

shareProcessNamespace   <boolean>
Share a single process namespace between all of the containers in a pod.
When this is set containers will be able to view and signal processes from
other containers in the same pod, and the first process in each container
will not be assigned PID 1. HostPID and ShareProcessNamespace cannot both
be set. Optional: Default to false. This field is beta-level and may be
disabled with the PodShareProcessNamespace feature.

subdomain    <string>
If specified, the fully qualified Pod hostname will be
"<hostname>.<subdomain>.<pod namespace>.svc.<cluster domain>". If not
specified, the pod will not have a domainname at all.

terminationGracePeriodSeconds   <integer>
Optional duration in seconds the pod needs to terminate gracefully. May be
decreased in delete request. Value must be non-negative integer. The value
zero indicates delete immediately. If this value is nil, the default grace
period will be used instead. The grace period is the duration in seconds
after the processes running in the pod are sent a termination signal and
the time when the processes are forcibly halted with a kill signal. Set
this value longer than the expected cleanup time for your process. Default
to 30 seconds.

tolerations    <[]Object>
If specified, the pod's tolerations.

volumes     <[]Object>
List of volumes that can be mounted by containers belonging to the pod.
More info: https://kubernetes.io/docs/concepts/storage/volumes/
```

对象的spec字段的文档通常包含RESOURCE、DESCRIPTION和FILEDS几节，其中FILEDS节中给出了可嵌套使用的字段、数据类型及功能描述。例如，上面命令的结果显示在FILEDS中的containers字段的数据类型是一个对象列表`([]Object)`，而且是一个必选字段。任何值为对象类型数据的字段都会嵌套一到多个下一级字段，例如，Pod对象中的每个容器也是对象类型数据，它同样包含嵌套字段，但容器不支持单独创建，而是要包含于Pod对象的上下文中，其详细信息可通过三级字段来获取，命令及其结果示例如下：

## A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl explain pods.spec.containers]
KIND: Pod
VERSION: v1

RESOURCE: containers <[]Object>

DESCRIPTION:
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.

A single application container that you want to run within a pod.

FIELDS:
args <[]string>
Arguments to the entrypoint. The docker image's CMD is used if this is not
provided. Variable references ${VAR_NAME} are expanded using the
container's environment. If a variable cannot be resolved, the reference :
the input string will be unchanged. The ${VAR_NAME} syntax can be escaped
with a double $$, ie: $$(${VAR_NAME}). Escaped references will never be
expanded, regardless of whether the variable exists or not. Cannot be
updated. More info:
https://kubernetes.io/docs/tasks/inject-data-application/define-command-and-args/

command <[]string>
Entrypoint array. Not executed within a shell. The docker image's
ENTRYPOINT is used if this is not provided. Variable references ${VAR_NAME}
are expanded using the container's environment. If a variable cannot be
resolved, the reference in the input string will be unchanged. The
${VAR_NAME} syntax can be escaped with a double $$, ie: $$(${VAR_NAME}).
Escaped references will never be expanded, regardless of whether the
variable exists or not. Cannot be updated. More info:
https://kubernetes.io/docs/tasks/inject-data-application/define-command-and-args/

env <[]Object>
List of environment variables to set in the container. Cannot be updated.

envFrom <[]Object>
List of sources to populate environment variables in the container. The
keys defined within a source must be a C_IDENTIFIER. All invalid keys will
be reported as an event when the container is starting. When a key exists
in multiple sources, the value associated with the last source will take
precedence. Values defined by an Env with a duplicate key will take
precedence. Cannot be updated.

image <string>
Docker image name. More info:
```

## A.1.1 部署目标

```
https://kubernetes.io/docs/concepts/containers/images This field is
optional to allow higher level config management to default or override
container images in workload controllers like Deployments and StatefulSets.

imagePullPolicy    <string>
Image pull policy. One of Always, Never, IfNotPresent. Defaults to Always
if :latest tag is specified, or IfNotPresent otherwise. Cannot be updated
More info:
https://kubernetes.io/docs/concepts/containers/images#updating-images

lifecycle    <Object>
Actions that the management system should take in response to container
lifecycle events. Cannot be updated.

livenessProbe    <Object>
Periodic probe of container liveness. Container will be restarted if the
probe fails. Cannot be updated. More info:
https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle#container-liveness-probes

name    <string> -required-
Name of the container specified as a DNS_LABEL. Each container in a pod
must have a unique name (DNS_LABEL). Cannot be updated.

ports    <[]Object>
List of ports to expose from the container. Exposing a port here gives the
system additional information about the network connections a container
uses, but is primarily informational. Not specifying a port here DOES NOT
prevent that port from being exposed. Any port which is listening on the
default "0.0.0.0" address inside a container will be accessible from the
network. Cannot be updated.

readinessProbe    <Object>
Periodic probe of container service readiness. Container will be removed
from service endpoints if the probe fails. Cannot be updated. More info:
https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle#container-readiness-probes

resources    <Object>
Compute Resources required by this container. Cannot be updated. More info:
https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container

securityContext    <Object>
Security options the pod should run with. More info:
https://kubernetes.io/docs/concepts/policy/security-context/ More info:
https://kubernetes.io/docs/tasks/configure-pod-container/security-context,
```

```
stdin    <boolean>
Whether this container should allocate a buffer for stdin in the container
runtime. If this is not set, reads from stdin in the container will always
```

## A.1.1 部署目标

```
result in EOF. Default is false.

stdinOnce    <boolean>
Whether the container runtime should close the stdin channel after it has
been opened by a single attach. When stdin is true the stdin stream will
remain open across multiple attach sessions. If stdinOnce is set to true,
stdin is opened on container start, is empty until the first client
attaches to stdin, and then remains open and accepts data until the client
disconnects, at which time stdin is closed and remains closed until the
container is restarted. If this flag is false, a container processes that
reads from stdin will never receive an EOF. Default is false

terminationMessagePath    <string>
Optional: Path at which the file to which the container's termination
message will be written is mounted into the container's filesystem. Message
written is intended to be brief final status, such as an assertion failure
message. Will be truncated by the node if greater than 4096 bytes. The
total message length across all containers will be limited to 12kb.
Defaults to /dev/termination-log. Cannot be updated.

terminationMessagePolicy    <string>
Indicate how the termination message should be populated. File will use the
contents of terminationMessagePath to populate the container status message
on both success and failure. FallbackToLogsOnError will use the last chunk
of container log output if the termination message file is empty and the
container exited with an error. The log output is limited to 2048 bytes or
80 lines, whichever is smaller. Defaults to File. Cannot be updated.

tty    <boolean>
Whether this container should allocate a TTY for itself, also requires
'stdin' to be true. Default is false.

volumeDevices    <[]Object>
volumeDevices is the list of block devices to be used by the container.
This is a beta feature.

volumeMounts    <[]Object>
Pod volumes to mount into the container's filesystem. Cannot be updated.

workingDir    <string>
Container's working directory. If not specified, the container runtime's
default will be used, which might be configured in the container image.
Cannot be updated.
```

内建文档大大降低了用户手动创建资源配置清单的难度，尝试使用某个资源类型时，explain也的确是用户常用的命令之一。熟悉各常用字段的功能之后，以同类型的现有活动对象的清单为模板可以更快地生成目标资源的配置文件，命令格式为

### A.1.1 部署目标

“kubectl get TYPE NAME -o yaml --export”，其中 --export 选项用于省略输出由系统生成的信息。例如，基于现在的 Deployment 资源对象 myapp 生成配置模板 deploy-demo.yaml 文件，可以使用如下命令：

```
**[terminal]
**[delimiter $ ]**[command kubectl get deployment myapp -o yaml --export > dep...
```

通过资源清单文件管理资源对象较之直接通过命令操作有着诸多优势，具体包括命令行的操作方式仅支持部分资源对象的部分属性，而资源清单支持配置资源的所有属性字段，而且使用配置清单文件还能够进行版本追踪、复审等高级功能的操作。本书后续章节中的大部分资源管理操作都会借助资源配置文件进行。

### 3.2.5 资源对象管理方式

Kubernetes 的 API Server 遵循声明式编程（declarative programming）范式而设计，侧重于构建程序逻辑而无须用户描述其实现流程，用户只需要设定期望的状态，系统即能自行确定需要执行的操作以确保达到用户期望的状态。例如，期望某 Deployment 控制器管理三个Pod资源对象时，而系统观察到的当前数量却是两个，于是系统就会直到需要创建一个新的Pod资源来满足此期望。Kubernetes 的自愈、自治的功能都依赖于其声明式机制。

于此对应的另一种范式称为陈述式编程（imperative programming），代码侧重于通过创建一种告诉计算机如何执行操作的算法来更改程序状态的语句来完成，它与硬件的工作方式密切相关，通常，代码将使用条件语句、循环和类继承等控制结构。为了便于用户使用，kubernetes 的 API Server 也支持陈述式范式，它直接通过命令及其选项完成对象的管理操作，前面用到的 run、expose、delete 和 get 等命令都属于此类，执行时用户需要告诉系统要做什么，例如，使用 run 命令创建一个有着 3 个pod对象副本的Deployment对象，或者通过 delete 命令删除一个名为 myapp 的 Service 对象。

Kubernetes 系统的大部分API对象都有着 spec 和 status 两个字段，其中，spec 用于让用户定义所期望的状态，系统从中读出相关的定义；而 status 则是系统观察并负责写入的当前状态，用户可以从中获取相关的信息。Kubernetes 系统通过控制器监视着系统对象，由其负责让系统当前的状态无线接近用户所期望的状态。

kubectl 的命令由此可以分为三类：陈述式命令（imperative command）、陈述式对象配置（imperative object configuration）和声明式对象配置（declarative object configuration）。第一种方式即此前用到的 run、export、delete 和 get 等命令，它们直接作用于 Kubernetes 系统上的活动对象，简单易用，但不支持代码复用、修改复审及审计日志等功能，这些功能的使用通常要依赖于资源配置文件，这些文件也称为资源清单。在这种模式下，用户可以访问每个对象的完整模式，但用户还需要深入学习 Kubernetes API。

如 3.2.4 节所述，资源清单本质上是一个JSON或YAML格式的文本文件，由资源对象的配置信息组成，支持使用Git等进行版本控制。而用户可以以资源清单为基础，在 kubernetes 系统上以陈述式或声明式进行资源对象管理，如图 3-4 所示：

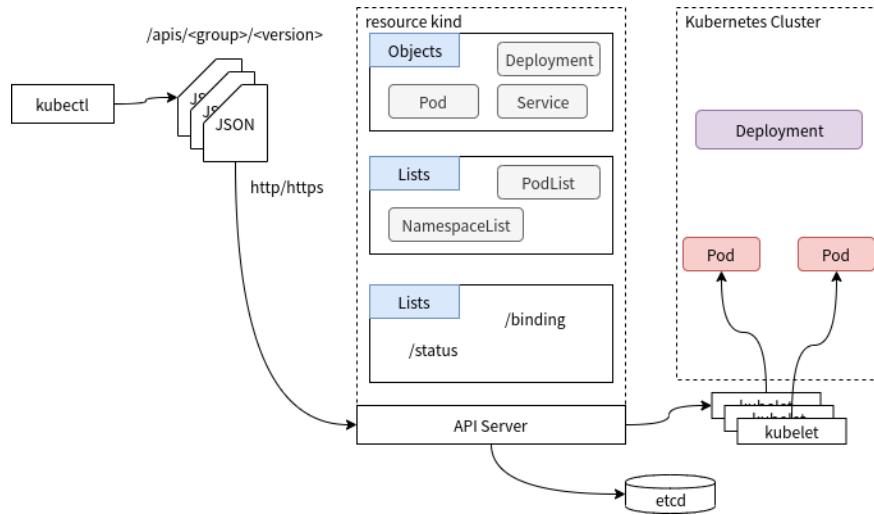


图 1.5.2.5 - 基于资源配置清单管理对象

陈述式管理方式包括包括 create、delete、get 和 replace 等命令，与陈述式命令的不同之处在于，它通过资源配置清单读取需要管理的目标资源对象。陈述式对象配置的管理操作直接作用活动对象，即便仅修改配置清单中的极小一部分内容，使用 replace 命令进行的对象更新也将会导致整个对象被替换。进一步地，混合使用陈述式命令进行清单文件带外修改时，必然会导致用户丢失活动对象的当前状态。

声明式对象配置并不直接指明要进行的对象管理操作，而是提供配置清单文件给 Kubernetes 系统，并委托系统跟踪活动对象的状态变动。资源对象的创建、删除及修改操作全部通过唯一的命令 apply 来完成，并且每次操作时，提供给命令的配置信息都将保存与对象的注解信息（`kubectl.kubernetes.io/last-applied-configuration`）中，并通过对比检查活动对象的当前状态、注解中的配置信息及资源清单中的配置信息三方进行变更合并，从而实现仅修改变动字段的高级补丁机制。

陈述式对象配置相较于声明式对象配置来说，其缺点在于同一目录下的配置文件必须同时进行同一操作，例如，要么都创建，要么都更新等，而且其他用户的更新也必须反映在配置文件中，不然其更新在下一次的更新中将会被覆盖。因此，声明式对象配置是优先推荐给用户使用的管理机制。

然而，对于新手来说，陈述式命令的配置方式最易于上手，对系统有所了解后易于切换为使用陈述式对象配置管理方式。因此，若推荐给高级用户则推荐使用声明式配置，并建议同时使用版本控制系统存储所期望的状态，以及跨对象的历史信息，并启用变更复审机制。另外，推荐使用借助于 `kube-applier` 等一类的项目实现自动化声明式配置，用户将配置推送到 git 仓库中，然后借助此类工具既能将其自动同步于 kubernetes 集群之上。

### 3.3 kubectl 命令与资源管理

API Server 是 kubernetes 集群的网关，用户和管理员以及其他客户端仅能通过此网关接口与集群尽心交互。API 是面向程序员的访问接口，目前可较好地支持 Golang 和 Python 编程语言，当然，终端用户更为常用的是通过命令行工具 kubectl。访问集群之前，各类客户端需要了解集群的位置并拥有访问集群的凭据才能获取访问许可。使用 kubeadm 进行集群初始化时，kubeadm init 自动生成的 /etc/kubernetes/admin.conf 文件是客户端接入当前集群时使用的 kubeconfig 文件，它内建了用于访问 Kubernetes 集群的最高管理权限的用户账号及相关的认证凭据，可由 kubectl 直接使用。

### 3.3.1 资源管理操作概述

资源的管理操作可简单归结为增、删、改、查四种，kubectl 提供了一系列的子命令用于执行此类任务，如 create、delete、patch、apply、replace、edit、get 等，其中有些命令必须基于资源清单来进行，如 apply 和 replace 命令，也有些命令即可基于清单文件进行，也可实时作用于活动资源之上，如 create、get、patch 和 delete 等。

kubectl 命令能够读取任何以 .yaml、.yml 或 .json 为后缀的文件（可称为配置清单或配置文件，后文将不加区别地使用这两个术语）。实践中，用户既可以为每个资源使用专用的清单文件，也可以将多个相关的资源（例如，属于同一个应用或微服务）组织在同一个清单文件中。不过，如果是 YAML 格式的清单文件，多个资源彼此之间要使用“---”符号作为单独的一行进行资源的分割。这样，多个资源就将以清单文件中定义的次序被 create、apply 等子命令调用。

kubectl 的多数子命令支持使用“-f”选项指定使用的清单文件路径或 URL，也可以是存储有清单文件的目录，另外，此选项再同一命令中也可重复使用多次。如果指定的目录路径存在子目录中时，那么可按需同时使用“-R”选项以递归获取子目录中的配置清单。

再者，支持使用标签和注解是 Kubernetes 系统的一大特色，它为资源管理机制增色不少，而且 delete 和 get 等命令能够基于标签挑选目标对象，有些资源甚至必须依赖标签才能正常使用和工作，例如 Service 和 Pod 控制器 Deployment 等资源对象。子命令 label 用于管理资源标签，而管理资源注解的子命令则是 annotate。

就地更新（修改）现有的资源也是一种常见的操作。apply 命令通过比较资源再清单文件中的版本及前一次的版本执行更新操作，它不会对未定义的属性产生额外的作用。edit 命令相当于先使用 get 命令获取资源配置，通过交互式编辑器修改后再自动使用 apply 命令将其应用。patch 命令基于 JSON 补丁、JSON 合并补丁及策略合并补丁对资源进行就地更新操作。

为了利用 apply 命令的优势，用户应该总是使用 apply 命令或 create --save-config 命令创建资源。

### 3.3.2 kubectl 的基本用法

Kubectl 是最常用的客户端工具之一，它提供了基于命令行访问 kubernetes API 的简洁方式，能够满足对 kubernetes 的绝大部分的操作需求。例如，需要创建资源对象时，kubectl会将JSON格式的清单内容以POST方式提交至API Server。本节主要描述 kubectl 的基本功能。

如果要单独部署 kubectl，Kubernetes 也提供了相应的单独发行包，或者适配与各平台的程序管理器的相关程序包，如rpm包或deb包等，用户根据平台类型的不同获取相匹配的版本安装完成即可，操作步骤类似与前面的安装方法，因此这里不在给出其具体过程。

kubectl 是 kubernetes 系统的命令行客户端工具，特性丰富且功能强大，是 Kubernetes 管理员最常用的集群管理工具。其最基本的语法格式为“`kubectl [command] [TYPE] [NAME] [flags]`”，其中各部分的简要说明如下：

1. command：对资源执行相应操作的子命令，如get、create、delete、run等；常用的核心子命令如表3-1所示。
2. TYPE：要操作的资源对象的类型，如 pods、services 等；类型名称区分字符大小写，但支持使用简写格式。
3. NAME：对象名称，区分字符大小写；省略时，则表示指定TYPE的所有资源对象；另外，也可以直接使用“TYPE/NAME”的格式来表示资源对象。
4. flags：命令行选项，如“-s”或“--server”；另外，get等命令再输出时还有一个常用的标志 `-o <format>` 用于指定输出格式，如表3-1所示：

### A.1.1 部署目标

命令	命令类别	功能说明
create	基础命令 (初级)	通过文件或标准输入创建资源
expose		基于rc、service、deployment或pod创建service资源
run		通过创建Deployment在集群中运行指定的镜像
set		设置指定资源的特定属性
get	基础命令 (中级)	显示一个或多个资源
explain		打印资源文档
edit		编辑资源
delete		基于文件名、stdin、资源或名字，以及资源和选择器删除资源
rollout	部署命令	管理资源的滚动更新
rollout-update		对ReplicationController执行滚动升级
scale		伸缩 Deployment、ReplicaSet、RC 或 Job 的规模
autoscale		对 Deployment、ReplicaSet 或 RC 进行自动伸缩
certificate	集群管理命令	配置数字证书资源
cluster-info		打印集群信息
top		打印资源 (CPU/Memory/Storage) 使用率
cordon		将指定node设定为“不可用”(unschedulable) 状态
uncordon		将指定node设定为“可用”(schedulable) 状态
drain		“排干”指定的node的负载以进入“维护”模式
taint		为node声明污点及标准行为
describe	排错及调试命令	显示指定资源或资源组的详细信息
logs		显示一个Pod内某容器的日志
attach		附加终端至一个运行中的容器
exec		在容器中执行指定命令

### A.1.1 部署目标

port-forward		将本地的一个或多个端口转发至指定的Pod
proxy		创建能够访问Kubernetes API Server的代理
cp		在容器间复制文件和目录
auth		打印授权信息
apply	高级命令	基于文件或stdin将配置应用于资源
patch		使用策略合并补丁更新资源字段
replace		基本文件或stdin替换一个资源
convert		为不同的API版本转换配置文件
label	设置命令	更新指定资源的label
annotate		更新资源的annotation
completion		输出指定的shell（如bash）的补全码
version	其他命令	打印 Kubernetes 服务端和客户端的版本信息
api-version		以“group/version”格式打印服务器支持的API版本信息
config		配置 kubeconfig 文件的内容
plugin		运行命令行插件
help		打印任一命令的帮助信息

kubectl 命令还包含了多种不同的输出格式（如表3-2所示），它们为用户提供了非常灵活的自定义输出机制，如输出为YAML或JSON格式等。

输出格式	格式说明
-o wide	显示资源的额外信息
-o name	仅打印资源的名称
-o yaml	YAML格式化输出API对象信息
-o json	JSON格式化输出API对象信息
-o go-template	以自定义的go模板格式化输出API对象信息
-o custom-columns	自定义要输出的字段

此外，kubectl 命令还有许多通用的选项，这个可以使用“kubectl options”命令来获取。下面列举几个比较常用命令。

- -s 或 --server：指定API Server的地址和端口

### A.1.1 部署目标

- `--kubeconfig`：使用的kubeconfig文件路径，默认为 `~/.kube/config`
- `--namespace`：命令执行的目标空间名称

kubectl 的部分在第二章已经多次提到，余下的大多数命令在后续的章节中还会用到，对于它们的使用说明也将在首次用到时进行展开说明。

## 3.4 管理名称空间资源

名称空间（Namespace）是 Kubernetes 集群级别的资源，用于将集群分隔为多个隔离的逻辑分区以配置给不同的用户、租户、环境或项目使用，例如，可以为 development、qa 和 production 应用环境分别创建各自的名称空间。

Kubernetes 的绝大多数资源都隶属于名称空间级别（另一个是全局级别或集群级别），名称空间资源为这类资源名称提供了隔离的作用域，同一名称空间内的同类型资源名称必须是唯一的，但跨名称空间时并无此限制。不过，Kubernetes 还是有一些资源隶属于集群级别的，如 Node、Namespace、PersistentVolume 等资源，它们不属于任何名称空间，因此资源对象的名称必须是全局唯一的。

Kubernetes 的名称空间资源不同于Linux系统的名称空间，它们是各自独立的概念。另外，kubernetes名称空间并不能实现Pod之间的通讯隔离，它仅用于限制资源对象名称的作用域。

### 3.4.1 查看名称空间及其资源对象

Kubernetes 集群默认提供了几个名称空间用于特定的目的，例如，kube-system 主要用于运行系统级资源，而 default 则为那些未指定名称空间的资源操作提供一个默认值，前面章节中的绝大多数资源管理操作都在 default 名称空间中进行。

“kubectl get namespaces”命令则可以查看 namespaces 资源：

```
**[terminal]
**[delimiter $ ]**[command kubectl get namespaces]
NAME      STATUS   AGE
default   Active   10d
kube-node-lease Active   10d
kube-public Active   10d
kube-system Active   10d
```

也可以使用 “kubectl describe namespaces” 命令查看特定名称空间的详细信息，例如：

```
**[terminal]
**[delimiter $ ]**[command kubectl describe namespaces]
Name:      default
Labels:    <none>
Annotations: <none>
Status:    Active

No resource quota.

No resource limits.
```

kubectl 的资源查看命令在多数情况下应该针对特定的名称空间来进行，为其使用“-n”或“--namespace”选项即可，例如，查看 kube-system 下所有的 Pod 资源：

### A.1.1 部署目标

NAME	READY	STATUS	RESTARTS	AGE
coredns-f8d45bc4-fs49m	1/1	Running	1	100d
coredns-f8d45bc4-tplxw	1/1	Running	1	100d
etcd-kubeedge.localdomain	1/1	Running	1	100d
kube-apiserver-kubeedge.localdomain	1/1	Running	1	100d
kube-controller-manager-kubeedge.localdomain	1/1	Running	1	100d
kube-flannel-ds-amd64-67gpx	1/1	Running	1	100d
kube-flannel-ds-amd64-rxnqv	1/1	Running	0	8d
kube-flannel-ds-amd64-txzbz	1/1	Running	0	8d
kube-flannel-ds-amd64-xz572	1/1	Running	0	8d
kube-proxy-2ppcl	1/1	Running	0	8d
kube-proxy-jjrnr5	1/1	Running	0	8d
kube-proxy-l784k	1/1	Running	1	100d
kube-proxy-qfj4g	1/1	Running	0	100d
kube-proxy-sz2xp	1/1	Running	0	8d
kube-scheduler-kubeedge.localdomain	1/1	Running	1	100d

命令结果显示 out kube-system 与 default 名称空间的 Pod 资源对象并不相同，这正是 Namespace 资源的名称隔离功能的体现。有了 Namespace 对象，用户再也不必精心安排资源名称，也不用担心误操作了其他用户的资源。

## 3.4.2 管理 Namespace 资源

Namespace 是 Kubernetes Server API 的标准资源类型之一，如 3.2.1 节中所述，它的配置主要有 kind、apiVersion、metadata 和 spec 等一级字段组成。将 3.2.1 节中的名称空间配置清单保存与配置文件中，使用陈述式对象配置命令 create 或声明式对象配置命令 apply 便能完成创建：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f namespace-example.yaml]
namespace/dev created
```

名称空间资源属性较少（通常只需要指定名称即可），简单起见，kubectl 为其提供了一个封装的专用陈述式命令“kubectl create namespace”。namespace 资源的名称仅能由字母、数字、连接线、下划线等字符组成。例如，下面的命令可用于创建名为 qa 的 Namespace 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl create namespace qa]
namespace/qa created
```

namespace 资源的名称仅能由字母、数字、连接线、下划线等字符组成。

实践中，不建议混用不同类别的管理方式。考虑到声明式对象配置管理机制的强大功能，强烈推荐用户使用 apply 和 patch 等命令进行资源创建及修改一类的管理操作。

使用 kubectl 管理资源时，如果一并提供名称空间选项，就表示此管理操作仅针对指定名称空间进行，而删除 Namespace 资源会级联删除其包含的所有其他资源对象。表 3-3 给出了几个常用的命令格式。

命令格式	功能
kubectl delete TYPE RESOURCE -n NS	删除指定名称空间内的指定资源
kubectl delete TYPE --all -n NS	删除指定名称空间内的指定类型的所 有资源
kubectl delete all -n NS	删除指定名称空间内的所有资源
kubectl delete all --all	删除所有名称空间中的所有资源

需要再次指出的是，kubernetes 对象仅用于资源对象名称的隔离，它自身并不能隔  
绝跨名称空间的 Pod 间通讯，那是网络策略（network policy）资源的功能。

## 3.5 Pod 资源的基础管理操作

Pod 是 Kubernetes API 中的核心资源类型，它可以定义在JSON或YAML格式的资源清单中，由资源管理命令进行陈述式或声明式管理。创建时，用户通过 `create` 或 `apply` 命令将请求提交到 API Server 并将其保存至集群状态存储系统 etcd 中，而后由调度器将其调度至最佳目标节点，并被相应节点的 kubelet 借助于容器引擎创建并启动。这种由用户直接通过API创建的Pod对象也成为自主式Pod。

### 3.5.1 陈述式对象配置管理方式

陈述式对象配置管理机制，是由用户通过配置文件指定要管理的目标资源对象，而后再由用户借助与命令直接指定 Kubernetes 系统要执行的管理操作的管理方式，常用的命令由 create、delete、replace、get 和 describe 等。

## 1. 创建 Pod 资源

Pod 是标准的 Kubernetes API 资源，在配置清单中使用 kind、apiVersion、metadata 和 spec 字段进行定义，status 字段在对象创建后由系统进行维护。Pod 对象的核心功能用在于运行容器化应用，在其spec字段中嵌套的必选字段是 containers，它的值是一个容器对象列表，支持嵌套创建一到多个容器。下面是一个Pod资源清单示例文件，在spec中定义的期望的状态是在 Pod 对象中基于 ikubernetes/myapp:v1 镜像运行一个名为 myapp 的容器：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
    - name: myapp
      image: ikubernetes/myapp:v1
```

把上面的内容保存于配置文件中，使用“kubectl [command] f [path/to/yaml\_file]”命令以陈述式对象配置进行资源对象的创建，下面是相应的命令及响应结果：

```
**[terminal]
**[delimiter $ ]**[command kubectl create -f pod-example.yaml]
pod/pod-exmaple created
```

如果读者熟悉JSON，也可以直接将清单文件定义为JSON格式；YAML格式的清单文件本身也是由API Server事先将其转换为JSON格式而后才进行应用的。

命令返回的信息表示目标 Pod 对象 pod-example 得以创建成功。事实上，create 命令中的 -f 选项也支持使用目录路径或URL，而且目标路径为目录时，还支持使用-R选项进行子目录递归。另外，--record 选项可以将命令本身记录为目标对象的注解信息 kubernetes.io/change-cause，而 --save-config 则能够将提供给命令的资源对象配置信息保存于对象的注解信息 kubectl.kubernetes.io/last-applied-configuration 中，后一个命令的功用与声明式对象配置命令 apply 的功能相近。

## 2. 查看 Pod 状态

get 命令默认显示资源对象最为关键的状态信息，而 describe 等命令则能够打印出 kubernetes 资源对象的详细状态。不过，虽然创建时给出的资源清单文件较为简洁，但“kubectl get”命令既可以使用“-o yaml”或“-o json”选项输出资源对象的配置数据及状态，也能够借助于“--custom-columns”选项自定义要显示的字段：

```
**[terminal]
**[delimiter $ ]**[command kubectl get -f pod-example.yaml]
NAME      READY   STATUS    RESTARTS   AGE
pod-exmaple  1/1     Running   0          10m
**[delimiter $ ]**[command kubectl get -f pod-example.yaml -o custom-columns=NAME:STATUS]
NAME      STATUS
pod-exmaple  Running
```

使用“-o yaml”或“-o json”选项时，get命令能够返回资源对象的元数据、期望的状态及当前状态数据信息，而要打印活动对象的详细信息，则需要 describe 命令，它可根据资源清单、资源名称或卷标等方式过滤输出符合条件的资源对象的信息。命令格式为“kubectl describe (-f FILENAME | TYPE [NAME\_PREFIX] | -l label | TYPE/NAME)”。例如，显示pod-example的详细信息，可使用类如下的命令：

```
**[terminal]
**[delimiter $ ]**[command kubectl describe -f pod-example.yaml]
```

对于Pod资源来说，他能够返回活动对象的元数据、当前状态、容器列表及各容器的详情、存储卷对象列表、QoS列表、事件及相关信息，这些详情对于了解目标资源对象的状态或进行错误排查等操作来说至关重要。

### 3. 更新 Pod 资源

对于活动对象，并非每个属性值都支持修改，例如，Pod 资源对象的 metadata.name 字段就不支持修改，除非删除并重建它。对于那些支持修改的属性，比如，容器的 image 字段，可将其完整的配置清单导出与配置文件中并更新相应的配置数据，而后使用 replace 命令基于陈述式对象配置的管理机制进行资源对象的更新。例如，将前面创建 pod-example 时使用的资源清单中的 image 值修改为 “ikubernetes/myapp:v2”，而后执行更新操作：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods pod-example -o yaml > pod-example.yaml]
**[delimiter $ ]**[command sed -i 's@^(image:@).*@image: ikubernetes/myapp:v2@'
**[delimiter $ ]**[command kubectl replace -f pod-example-update.yaml]
pod/pod-example replaced
```

更新活动对象的配置时，replace 命令要重构整个资源对象，故此它必须基于完整格式化的配置信息才能进行活动对象的完全替换。若要基于此前的配置文件进行替换，就必须使用 --force 选项删除此前的活动对象，而后再进行新建操作，否则命

### A.1.1 部署目标

令会返回错误信息。例如，将前面第一步创建“创建Pod资源”内的配置清单中的镜像修改为“ikubernetes/myapp:v2”后再进行强制替换，命令如下：

```
**[terminal]
**[delimiter $ ]**[command kubectl replace -f pod-example.yaml --force]
pod "pod-example" deleted
pod/pod-example replaced
```

## 4. 删除 Pod 资源

陈述式对象配置管理方式下的删除操作与创建、查看及更新操作类似，为 delete 命令使用 -f 选项指定配置清单即可，例如，删除 pod-example.yaml 文件中定义的 Pod 资源对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl delete -f pod-example.yaml]
pod "pod-example" deleted
```

之后再次打印相关配置清单中定义的资源对象即可验证其删除的结果，例如：

```
**[terminal]
**[delimiter $ ]**[command kubectl get -f pod-example.yaml]
Error from server (NotFound): pods "pod-example" not found
```

## 3.5.2 声明式对象配置管理方式

陈述式对象配置管理机制中，同时指定的多个资源必须进行同一种操作，而且其 replace命令是通过完全替换现有活动对象来进行资源的更新操作，对于生产环境来说，这并非理想的选择。声明式对象配置操作在管理资源对象时将配置信息保存于目标对象的注解中，并通过比较活动对象的当前配置、前一次管理操作时保存于注解中的配置，以及当前命令提供的配置生成更新补丁从而完成活动对象的补丁式更新操作。此类管理操作的常用命令有 apply 和 patch 等。

例如，创建 3.5.1 节中定义的主容器使用“kubernetes/myapp:v1”镜像的Pod资源对象，还可以使用如下命令进行：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f pod-example.yaml]
pod/pod-example created
```

而更新对象的操作，可在直接修改原有资源清单文件后再次对其执行 apply 命令来完成，例如，修改Pod资源配置清单中的镜像文件为“ikubernetes/myapp:v2”后再次执行如上的 apply 命令：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f pod-example.yaml]
pod/pod-example configured
```

命令结果显示资源重新配置完成并且已经生效。事实上，此类操作也完全能够使用 patch 命令直接进行补丁操作。而资源对象的删除操作依然可以使用 apply 命令，但要同时使用 --prune 选项，命令的格式为 kubectl apply -f <directory/> --prune -l <labels>。需要注意的是，此命令异常凶险，因为它将基于标签选择器过滤出所有符合条件的对象，并检查由 -f 指定的目录中是否存在某配置文件已经定义了相应的资源对象，那些不存在相应定义的资源对象将被删除。因此，删除资源对象的操作依然建议使用陈述式对象配置方式的命令“kubectl delete”进行，这样的命令格式操作目标明确且不易出现偏差。

## 第四章 管理 Pod 资源对象

Pod 是 Kubernetes 系统的基础单元，是资源对象模型中可由用户创建或部署的最小组件，也是在 kubernetes 系统上运行容器化应用的资源对象。其他的大多数资源对象都是用于支撑和扩展Pod对象功能的，例如，用于管控Pod运行的 StatefulSet 和 Deployment 等控制器对象，用于暴露Pod应用的 Service 和 Ingress 对象，为 Pod 提供存储的 PersistentVolume 存储资源对象等。这些资源对象大体可分为有限的几个类别，并且可基于资源清单作为资源配置文件进行陈述式或声明式管理。本章将描述这些类别，并详细介绍 Pod 资源的基础应用。

## 4.1 容器与 Pod 资源对象

现代的容器技术被设计用来运行单个进程（包括子进程）时，该进程在容器中PID名称空间中的进程号为1，可直接接受并处理信号，于是，在此类进程终止时，容器即终止退出。若要在容器内运行多个进程，则需要为这些进程提供一个类似与Linux操作系统init进程的管控类进程，以树状结构完成多进程的生命周期管理，例如，崩溃后回收相应的系统资源等。单容器运行多进程时，通常还需要日志进程来管理这些进程的日志，例如，将它们分别保存于不同的目标日志文件等，否则用户就不得不手动来分拣日志信息。因此，绝大多数场景中都应该于一个容器中仅运行一个进程，它将日志信息直接输出至容器的标准输出，支持用户直接使用命令（kubectl logs）进程获取，这也是Docker或Kubernetes使用容器的标准方式。

不过，分别运行与各自容器的进程之间无法实现基于IPC的通信机制，此时，容器间的隔离机制对于依赖于此类通信方式的进程来说却又成为了障碍。Pod 资源抽象正是用来解决此类问题的组件，前文已然多次提到，Pod对象是一组容器的集合，这些容器共享 Network、UTS 及 IPC 名称空间，因此具有相同的域名、主机名和网络接口，并可通过IPC直接通信。为一个POD对象中的各容器提供网络名称空间等共享机制的是底层基础容器pause，图4-1所示为一个有三个容器组成的Pod资源，各容器共享Network、IPC和UTS名称空间，但分别拥有各自的MNT、USR和PID名称空间。需要特别强调的是，一个Pod对象中的多个容器必须运行于同一工作节点之上。

尽管可以将Pod类比为物理机或VM，但一个Pod内通常仅应该运行一个应用，除非多个进程之间具有密切的关系。这也意味着，实践中应该将多个应用分别构建到多个而非单个Pod中，这样也更能符合容器的轻量化设计、运行之目的。

例如，一个有着前端（application server）和后端（database server）的应用，其前、后端应该分别组织与各自的Pod中，而非同一个Pod的不同容器中。这样做的好处在于，多个Pod可被调度至多个不同的主机运行，提高了资源的利用率。另外，Pod也是Kubernetes进行系统规模伸缩的基础单元，分别运行与不同Pod的多个应用可独立按需求进行规模的变动，这就增强了系统架构的灵活性。事实上，前、后端应用的规模需求通常不会相同，而且无状态应用（application server）的规模变动也比有状态应用（database server）容易的多，将它们组织于同一个Pod中时将无法享受这种便利。

不过，这些场景要求必须与同一Pod中同时运行多个容器。此时，这些分布式应用（尤其是微服务架构中的多个服务）必须遵守某些最佳实践机制或基本准则。事实上，Kubernetes 并非期望成为一个管理系统，而是一个支持这些最佳实践的向开发人员或管理人员提供更高级别服务的系统。分布式系统设计通常包含以下几种模型：

- 1) Sidecar pattern（边车模型或跨斗模型）：即为 Pod 的主应用容器提供协同的辅助应用容器，每个应用独立运行，最为典型的代表是将主应用容器中的日志使用 agent 收集至日志服务器中时，可以将 agent 运行为辅助应用容器，即 sidecar。另一个典型的应用是为主应用容器中的 database server 启用本地缓存，如图 4-2 所示：

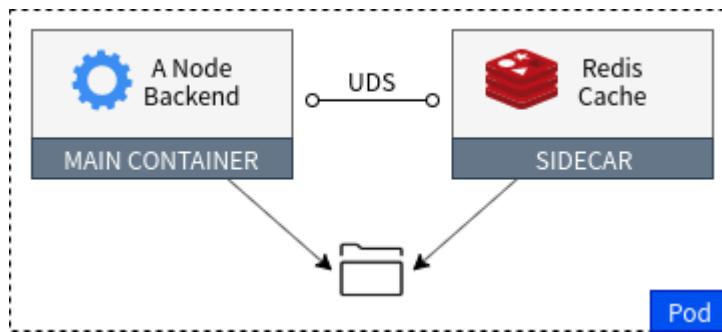


图 1.6.1 - Sidecar pattern

2) Ambassador pattern (大使模型)：即为远程服务创建一个本地代理，代理应用运行与容器中，主容器中的应用通过代理容器访问远程服务，如图4-3所示。一个典型的使用示例是主应用程序中的进程访问“一主多从”模型的远程Redis应用时，可在当前Pod容器中为Redis服务创建一个 Ambassador container，主应用容器中的进程直接通过 localhost 接口访问 Ambassador container即可。即便是Redis主从集群架构发生变动时，也仅需要将 Ambassador container 加以修改即可，主应用容器无须对此做出任何反映。

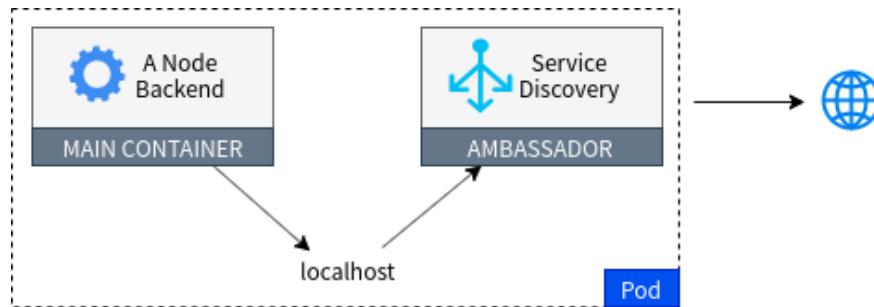


图 1.6.1 - Ambassador pattern

3) Adapter pattern: (适配器模型)：此种模型一般用于将主应容器中的内容进行标准化输出，例如，日志数据或指标数据的输出，这有助于调用者统一接收数据的接口，如果图 4-4 所示。另外，某应用滚动升级后的版本不兼容旧的版本时，其报告信息的格式也存在不兼容的可能性，使用 Adapter container 有助于避免那些调用此报告数据的应用发生错误。

### A.1.1 部署目标

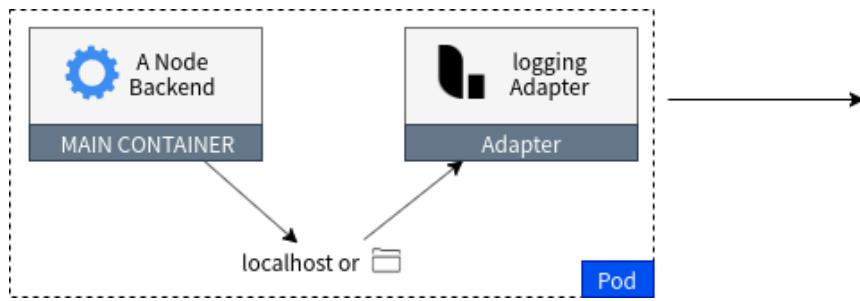


图 1.6.1 - Adapter pattern

Kubernetes 系统的Pod资源对象用于运行单个容器化应用，此应用称为Pod对象的主容器（main container），同时Pod也能够容纳多个容器，不过额外的容器一般工作为 Sidecar 模型，用于辅助主容器完成工作职能。

## 4.2 管理 Pod 对象的容器

一个Pod对象中至少要存在一个容器，因此，`containers` 字段是定义 Pod 时其嵌套字段 Spec 中的必选项，用于为 Pod 指定要创建的容器列表。进行容器配置时，`name`为必选字段，用于指定容器名称，`image`字段是为可选，以方便更高级别的管理类资源（如Deployment）等能覆盖此字段，于是自主式的Pod并不可省略此字段。因此，定义一个容器的基础框架如下：

```
name: CONTAINER_NAME  
image: IMAGE_FILE_NAME
```

此外，定义容器时还有一些其他常用的字段，例如，定义要暴露的端口、改变镜像运行的默认程序、传递环境变量、定义可用的系统资源配置等。

## 4.2.1 镜像及其获取策略

各工作节点负责运行Pod对象，而Pod的核心功用在于运行容器，因此工作节点上必须配置容器运行引擎，如Docker等。启动容器时，容器引擎将首先于本地查找指定的镜像文件，不存在的镜像则需要从指定的镜像仓库（Registry）下载至本地，如图 4-5 所示。

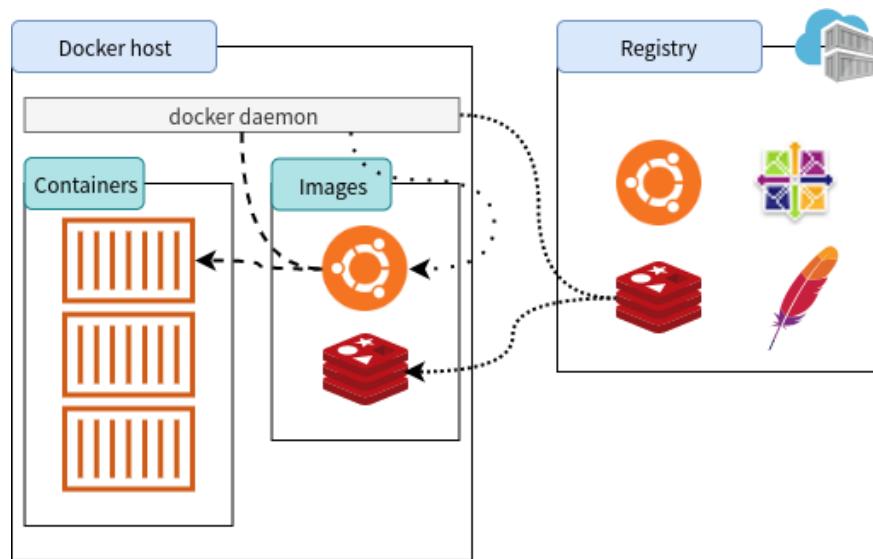


图 1.6.2.1 - Docker 及其 Registry

Kubernetes 系统支持用户自定义镜像文件的获取策略，例如在网络资源较为紧张的情况下可以禁止从仓库中获取镜像文件等。容器的“imagePullPolicy”字段用于为其指定镜像获取策略，它的可用值包括如下几个。

- Always：镜像标签为“latest”或镜像不存在时总是从指定的仓库中获取镜像。
- ifNotPresent：仅当本地镜像缺失时方才从目标仓库下载镜像。
- Never：禁止从仓库下载镜像，仅使用本地镜像。

下面的资源清单中的容器定义了如何使用 nginx:latest 镜像，其获取策略为 Always，这意味着每次启动容器时，他都会到镜像仓库中获取最新版本的镜像文件：

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
    - name: nginx
      image: nginx:latest
      imagePullPolicy: Always

```

### A.1.1 部署目标

对于标签为“latest”的镜像文件，其默认的镜像策略即为“always”，而对于其他标签的镜像，其默认策略则为“IfNotPresent”。需要注意的是，使用私有仓库中的镜像时通常需要由 Registry 服务器完成认证后才能进行。认证过程要么需要在相关节点上交互式执行 docker login 命令来进行，要么就是将认证信息定义为专有的 Secret 资源，并配置 Pod 通过“imagePullSecrets”字段调用此认证信息完成。后面 8.5 节会详细介绍此功能及其实现。

## 4.2.2 暴露端口

Docker 的网络模型中，使用默认网络的容器化应用需要通过NAT机制将其“暴露”（expose）到外部网络中才能被其他节点之上的容器客户端所访问。然而，Kubernetes 系统的网络模型中，各 Pod 的 IP 地址处于同一网络平面，无论是否为容器指定了要暴露的端口，都不会影响集群中其他节点之上的Pod客户端对其进行访问，这就意味着，任何监听在非lo接口上的端口都可以通过Pod网络直接被请求。从这个角度来说，容器端口只是信息性数据，它只是为集群用户提供了一个快速了解相关Pod对象的可访问端口的途径，而且显式指定容器端口，还能为其赋予一个名称以方便调用。

容器的 ports 字段的值是一个列表，由一到多个端口对象组成，它的常用嵌套字段包括如下几个。

- `containerPort <integer>`：必选字段，指定在Pod对象的IP地址上暴露的容器端口，有效范围为(0, 65536)；使用时，应该总是指定容器应用正常监听着的端口。
- `name <string>`：当前端口的名称，必须符合 IANA\_SVC\_NAME 规范且在当前 Pod 内必须是唯一的；此端口名可被 Service 资源调用。
- `protocol`：端口相关的协议，其值仅可为TCP或UDP，默认为TCP。

下面的资源配置清单示例（pod-example-with-port.yaml）中定义的 pod-example 指定了要暴露容器上 TCP 的 80 端口，并将之命令为 http：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
    - name: myapp
      image: ikubernetes/myapp:v1
      ports:
        - name: http
          containerPort: 80
          protocol: TCP
```

然而，Pod 对象的IP地址仅在当前集群内可达，它们无法直接接收来自集群外部客户端的请求流量，尽管它们的服务可达性不受工作节点边界的约束，但依然受制于集群边界。一个简单的解决方案是通过其所在的工作节点的IP地址和端口将其暴露到集群外部，如图 4-6 所示。

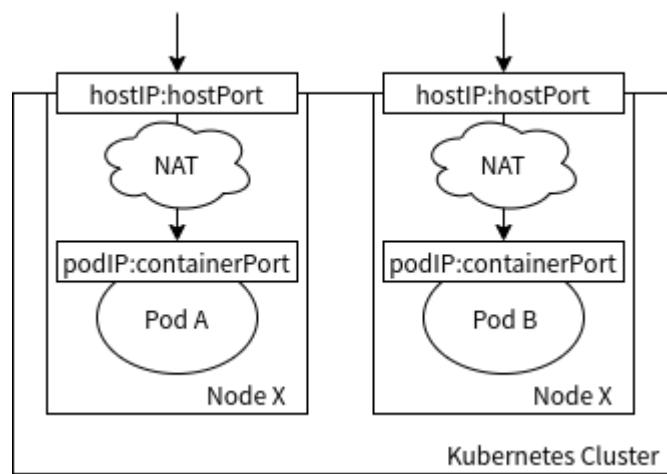


图 1.6.2.2 - 通过 hostIP 和 hostPort 暴露容器服务

- `hostPort <integer>`：主机端口，它将接收到的请求通过NAT机制转发至由 `containerPort` 字段指定的容器接口。
- `hostIP <string>`：主机端口要绑定的主机IP，默认为 0.0.0.0，即主机之上所有可用的IP地址；考虑到托管的Pod对象是由调度器调度运行的，工作节点的IP地址难以明确指定，因此此字段通常使用默认值。

需要注意的是，`hostPort` 和 `NodePort` 类型的 Service 对象暴露端口的方式不同，`NodePort` 是通过所有节点暴露容器服务，而 `hostPort` 则是经由 Pod 对象所在的节点的IP地址来进行。

## 4.2.3 自定义运行的容器化应用

由 Docker 镜像启动容器时运行的应用程序在相应的 Dockerfile 中由 ENTRYPOINT 指令进行定义，传递给程序的参数则通过 CMD 指令指定，ENTRYPOINT 指令不存在时，CMD 可用于同时指定程序及其参数。例如，在某工作节点上运行下面的命令获取 ikubernetes/myapp:v1 镜像中定义的 CMD 和 ENTRYPOINT，命令如下：

```
**[terminal]
**[delimiter $ ]**[command docker inspect ikubernetes/myapp:v1 -f {{.Config.Cmd}}
[nginx -g daemon off;]

**[delimiter $ ]**[command docker inspect ikubernetes/myapp:v1 -f {{.Config.Entrypoint}}
[]
```

容器的 command 字段能够指定不同于镜像默认运行的应用程序，并且可以同时使用 args 字段进行参数传递，它们将覆盖镜像中定义的程序及参数，并以无参数方式运行应用程序。例如下面的资源清单文件将镜像 ikubernetes/myapp:v1 的默认应用程序修改为了 “/bin/sh”，传递应用的参数修改为了 “-c while true; do sleep 30; done”：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-custom-command
spec:
  containers:
    - name: myapp
      image: alpine:latest
      command: ["/bin/sh"]
      args: ["-c", "while true; do sleep 30; done"]
```

自定义args，也是向容器中的应用程序传递配置信息的常用方式之一，对于非云原生（cloud native）的应用程序，这几乎也是最简单的配置方式。另一个常用的方式是使用环境变量。

## 4.2.4 环境变量

非容器化的传统管理方式中，复杂应用程序的配置信息多数由配置文件进行指定，用户可借助于简单的文本编辑器完成配置管理。然而，对于容器隔离出的环境中的应用程序，用户就不得不穿透容器边界在容器内进行配置编辑并进行重载，这种方式复杂且低效。于是，有环境变量在容器启动时传递配置信息就成为一种备受青睐的方式。

这种方式依赖于应用程序支持通过环境变量进行配置的能力，否则，用户在制作 Docker 镜像时需要通过 entrypoint 脚本完成环境变量到程序配置文件的同步。

向 Pod 对象中的容器环境变量传递数据的方法有两种：env 和 envFrom，这里重点介绍第一种方式，第二种方式将在介绍 ConfigMap 和 Secret 资源时进行说明。

通过环境变量配置容器化应用时，需要在容器配置段中嵌套使用 env 字段，他的值是一个由环境变量构成的列表。环境变量通常由 name 和 value 字段构成。

- `name <string>`：环境变量的名称，必选字段。
- `value <string>`：传递给环境变量的值，通过`$(VAR_NAME)`引用，逃逸格式为`$$$(VAR_NAME)"`，默认值为空。

下面配置清单中定义的Pod对象为其容器 filebeat 传递了两个环境变量，`REDIS_HOST` 定义了 filebeat 收集的日志信息要发往的 Redis 主机地址，`LOG_LEVEL` 则定义了 filebeat 的日志级别：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-env
spec:
  containers:
    - name: filebeat
      image: ikubernetes/filebeat:5.6.5-alpine
      env:
        - name: REDIS_HOST
          value: db.ilinux.io:6379
        - name: LOG_LEVEL
          value: info
```

这些环境变量可直接注入容器的 shell 环境中，无论它们是否真正被用到，使用 `printenv` 一类的命令都能在容器中获取到所有环境变量的列表。

## 4.2.5 共享节点的网络名称空间

同一个 Pod 对象的各容器均运行于一个独立的、隔离的 Network 名称空间中，共享同一个网络协议栈及相关的网络设备，如图 4-7a 所示。也有一些特殊的 Pod 对象需要运行于所在节点的名称空间中，执行系统级的管理任务，例如查看和操作节点的网站资源甚至是网络设备等，如图4-7b所示。

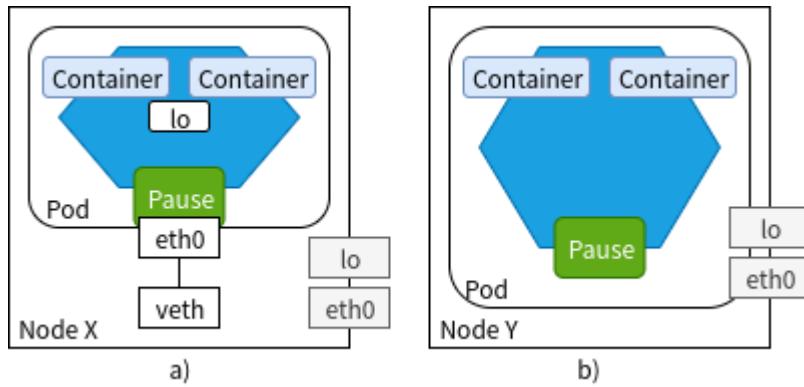


图 1.6.2.5 - Pod 对象的网络名称空间

通常，以Kubeadm 部署的 Kubernetes 集群中的 kube-apiserver、kube-controller-manager、kube-scheduler，以及 kube-proxy 和 kube-flannel 等通常都是第二种类型的Pod对象。事实上，仅需要设置 spec.hostNetwork 的属性为 true 即可创建共享节点网络名称空间的 Pod 对象，如下面的配置清单所示：

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-use-hostnetwork
spec:
  containers:
    - name: myapp
      image: ikubernetes/myapp:v1
      hostNetwork: true

```

将上面的配置清单保存与配置文件中，如 pod-use-hostnetwork.yaml，将其创建与集群上，并查看其网络接口的相关属性信息以验证它是否能共享使用工作节点的网络名称空间：

## A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f pod-use-hostnetwork.yaml]
pod/pod-use-hostnetwork created
**[delimiter $ ]**[command kubectl exec -it pod-use-hostnetwork -- sh]
/ # ifconfig
cni0      Link encap:Ethernet  HWaddr 72:C6:CF:37:34:8C
          inet addr:10.244.3.1  Bcast:0.0.0.0  Mask:255.255.255.0
             inet6 addr: fe80::70c6:ffff:fe37:348c/64 Scope:Link
               UP BROADCAST MULTICAST  MTU:1500  Metric:1
               RX packets:276 errors:0 dropped:0 overruns:0 frame:0
               TX packets:286 errors:0 dropped:0 overruns:0 carrier:0
               collisions:0 txqueuelen:1000
               RX bytes:19983 (19.5 KiB)  TX bytes:35428 (34.5 KiB)

docker0   Link encap:Ethernet  HWaddr 02:42:5C:E0:08:C5
          inet addr:172.17.0.1  Bcast:172.17.255.255  Mask:255.255.0.0
             UP BROADCAST MULTICAST  MTU:1500  Metric:1
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

ens192    Link encap:Ethernet  HWaddr 00:0C:29:AD:A8:D2
          inet addr:192.168.0.183  Bcast:192.168.0.255  Mask:255.255.255.0
             inet6 addr: fe80::20c:29ff:fead:a8d2/64 Scope:Link
               UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
               RX packets:3235233 errors:0 dropped:1 overruns:0 frame:0
               TX packets:2402001 errors:0 dropped:0 overruns:0 carrier:0
               collisions:0 txqueuelen:1000
               RX bytes:1594050628 (1.4 GiB)  TX bytes:237476297 (226.4 MiB)

flannel.1 Link encap:Ethernet  HWaddr 36:4D:70:4C:69:47
          inet addr:10.244.3.0  Bcast:0.0.0.0  Mask:255.255.255.255
             inet6 addr: fe80::344d:70ff:fe4c:6947/64 Scope:Link
               UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
               RX packets:237 errors:0 dropped:0 overruns:0 frame:0
               TX packets:232 errors:0 dropped:8 overruns:0 carrier:0
               collisions:0 txqueuelen:0
               RX bytes:28098 (27.4 KiB)  TX bytes:17586 (17.1 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
             inet6 addr: ::1/128 Scope:Host
               inet6 addr: ::1/128 Scope:Host
               UP LOOPBACK RUNNING  MTU:65536  Metric:1
               RX packets:1338 errors:0 dropped:0 overruns:0 frame:0
               TX packets:1338 errors:0 dropped:0 overruns:0 carrier:0
```

### A.1.1 部署目标

```
collisions:0 txqueuelen:1000
RX bytes:81611 (79.6 KiB) TX bytes:81611 (79.6 KiB)
```

如上述命令的结果显示所示，它打印出的工作节点的网络设备及其相关的接口信息。这就意味着，Pod 对象中运行的容器化应用也将监听与其所在的工作节点的IP 地址之上，这可以通过直接向 kube-node-2.localdomain 节点发起请求来验证：

```
**[terminal]
**[delimiter $ ]**[command curl kube-node-2.localdomain]
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
```

另外，在 Pod 对象中时还可以分别使用 spec.hostPID 和 spec.hostIPC 来共享工作节点的PID和IPC名称空间。

## 4.2.6 设置 Pod 对象的安全上下文

Pod 对象的安全上下文用于设定 Pod 或容器的权限和访问控制功能，其支持设置的常用属性包括以下几个方面：

- 基于用户ID（UID）和组ID（GID）控制访问对象（如文件）时的权限。
- 以特权或非特权的方式运行。
- 通过Linux Capabilities为其提供部分特征。
- 基于 Seccomp 过滤进程的系统调用。
- 基于 SELinux 的安全标签。
- 是否能够进行权限升级。

Pod 对象的安全上下文定义在 spec.securityContext 字段中，而容器的安全上下文则定义在 spec.containers[].securityContext 字段中，且二者可嵌套使用的字段还有所不同。下面的配置清单示例为 busybox 容器定义了安全上下文，它以 uid 为 1000 的非特权用户运行容器，并禁止权限升级：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-securitycontext
spec:
  containers:
    - name: busybox
      image: busybox
      command: ["/bin/sh", "-c", "sleep 86400"]
      securityContext:
        runAsNonRoot: true
        runAsUser: 1000
        allowPrivilegeEscalation: false
```

将上面的配置清单保存与配置文件（如 pod-with-securitycontext.yaml 文件）中，而后创建于集群中即可验证容器进程的运行者身份：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f pod-with-securitycontext.yaml]
**[delimiter $ ]**[command exec pod-with-securitycontext -- ps aux]
PID   USER     TIME   COMMAND
 1  1000      0:00  sleep 86400
 6  1000      0:00  ps aux
```

另外，可设置的安全上下文属性还有 fsGroup、seLinuxOptions、supplementalGroups、sysctls、capabilities 和 privileged 等，且 Pod 和容器各自支持的字段也有所不同，感兴趣的读者可按需对各属性进行测试。

## 4.3 标签与标签选择器

实践中，随着同类资源对象的数量越来越多，分类管理也变得越来越有必要：基于简单且直接的标准将资源对象划分为多个比较小的分组，无论是对开发人员还是对系统工程师来说，都能提升管理效率，这也正是 Kubernetes 标签（Label）的核心功能之一。对于附带标签的资源对象，可使用标签选择器（Label Selector）挑选出符合过滤条件的资源以完成所需要的操作，如关联、查看和删除等。

### 4.3.1 标签概述

标签是 Kubernetes 极具特色的功能之一，它能够附加于 Kubernetes 的任何资源对象之上。简单来说，标签就是“键值”类型的数据，它们可在资源创建时直接指定，也可随时按需添加于活动对象之中，而后即可由标签选择器进行匹配检查从而完成资源挑选。一个对象可拥有不止一个标签，而同一个标签也可以被添加至多个资源之上。

实践中，可以为资源附加多个不同维度的标签以实现灵活的资源分组管理功能，例如，版本标签、环境标签、分层架构标签等，用于交叉标示同一个资源所属的不同版本、环境及架构层级等，如图 4-8 所示。下面是较为常用的标签。

- 版本标签：“release”: “stable”, “release”: “canary”, “release”: “beta”
- 环境标签：“envionmrnt”: “dev”, “environment”: “qa”, “environment”: “production”
- 应用标签：“app”: “ui”, “app”: “as”, “app”: “pc”, “app”: “sc”
- 架构层级标签：“tier”: “frontend”, “tier”: “backend”, “tier”: “cache”
- 分区标签：“partition”: “customerA”, “partition”: “customerB”
- 品控级别标签：“track”: “daily”, “track”: “weekly”

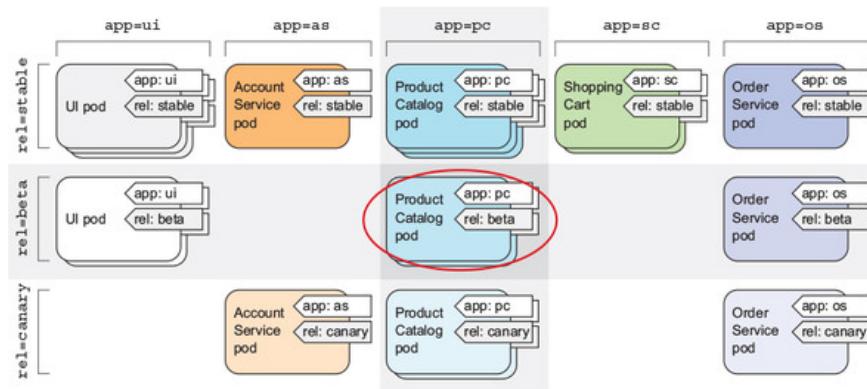


图 1.6.3.1 - 多维度标签使用示例

标签中的键名称通常由键前缀和键名组成，其中键前缀可选，其格式形如“KEYPREFIX/KEY\_NAME”。键名至多只能使用63个字符，可使用字母、数字、连接号 (-)、下划线 (\_)、点号 (.) 等字符，并且只能以字母或数字开头。键前缀必须为DNS子域名格式，且不能超过 253 个字符。省略键前缀时，键将被视为用户的私有数据，不过由 Kubernetes 系统组件或第三方组件自动为用户资源添加的键必须使用键前缀，而“kubernetes.io”前缀则预留给 Kubernetes 的核心组件使用。

标签中的键值必须不能多于 63 个字符，它要么为空，要么是以字母或数字开头及结尾，且中间仅使用字母、数字、连接号 (-)、下划线 (\_) 或点号 (.) 等字符的数据。

实践中，建议键名及键值能做到“见名知义”，且尽可能保持简单。

## 4.3.2 管理资源标签

创建资源时，可直接在其 metadata 中嵌套使用 “labels” 字段以定义要附加的标签项。例如，下面的 Pod 资源清单文件示例 pod-with-labels.yaml 中使用了两个标签 env=qa 和 tier=frontend：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-labels
  labels:
    env: qa
    tier: frontend
spec:
  containers:
  - name: myapp
    image: ikubernetes/myapp:v1
```

基于此资源清单创建出定义的Pod对象之后，即可在“kubectl get pods”命令中使用“--show-labels”选项，以额外显示对象的标签信息：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f pod-with-labels.yaml]
pod/pod-with-labels created
**[delimiter $ ]**[command kubectl get pods --show-labels]
NAME          READY   STATUS    RESTARTS   AGE   LABELS
pod-example   1/1     Running   1          2d4h  <none>
pod-with-labels   1/1     Running   0          32s   env=qa,tier=fror
```

标签较多时，在“kubectl get pods”命令上使用“-L key1, key2, ...”选项可指定显示有着特定键的标签信息。例如，仅显示各 pods 之上的以 env 和 tier 为键名的标签：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -L env,tier]
NAME          READY   STATUS    RESTARTS   AGE   ENV   TIER
pod-example   1/1     Running   1          2d4h
pod-with-labels   1/1     Running   0          4m3s   qa     frontend
```

“kubectl label”命令可以直接管理活动对象的标签，以按需进行添加或修改等操作。例如，为 pod-example 添加 env=production 标签：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl label pods/pod-example env=production]
pod/pod-example labeled
```

不过，对于已经附带了指定键名的标签，使用“kubectl label”为其设定新的键值时需要为命令同时使用“--overwrite”命令以强制覆盖原有的键值。例如，将 pod-with-labels 的 env 的值修改为“testing”：

```
**[terminal]
**[delimiter $ ]**[command kubectl label pods/pod-with-labels env=testing --ovr
pod/pod-with-labels labeled
```

用户若期望某标签之下的资源集合执行某类操作，例如，查看或删除等，则需要先使用“标签选择器”挑选出满足条件的资源对象。

### 4.3.3 标签选择器

标签选择器用于表达标签的查询条件或选择标准，Kubernetes API 目前支持两个选择器：基于等值关系（equality-based）以及基于集合关系（set-based）。例如，`env=production` 和 `env!=qa` 是基于等值关系的选择器，而 `tier in (frontend, backend)` 则是基于集合关系的选择器。另外，使用标签选择器时还将遵循以下逻辑。

1) 同时指定的多个选择器之间的逻辑关系为“与”操作。2) 使用空值的标签选择器意味着每个资源对象都将被选中。3) 空的标签选择器将无法选出任何资源。

基于等值关系的标签选择器的可用操作符有“`=`”“`==`”和“`!=`”三种，其中前两个意义相同，都表示“等值”关系，最后一个表示“不等”关系。“`kubectl get`”命令的“`-l`”选项能够指定使用标签选择器，例如，显示键名 `env` 的值不为 `qa` 的所有 Pod 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l "env!=qa" -L env]
NAME          READY   STATUS    RESTARTS   AGE   ENV
pod-example   1/1     Running   1          2d19h  production
pod-with-labels 1/1     Running   0          15h    testing
```

再例如，显示标签键名 `env` 的值不为 `qa`，且标签键名 `tier` 的值为 `frontend` 的所有 Pod 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l "env!=qa,tier=frontend" -L env,1
NAME          READY   STATUS    RESTARTS   AGE   ENV      TIER
pod-with-labels 1/1     Running   0          15h    testing   frontend
```

基于集合关系的标签选择器支持 `in`、`notin` 和 `exists` 三种操作符，它们的使用格式及意义具体如下。

- `KEY in (VALUE1, VALUE2, ...)`：指定的键名的值存在于给定的列表中即满足条件。
- `KEY notin (VALUE1, VALUE2, ...)`：指定的键名的值不存在于给定的列表中即满足条件。
- `KEY`：所有存在此键名标签的资源。
- `!KEY`：所有不存在此键名标签的资源。

例如，显示标签键名 `env` 的值为 `production` 或 `env` 的所有 Pod 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l "env in (production,dev)" -L env
NAME          READY   STATUS    RESTARTS   AGE   ENV
pod-example   1/1     Running   1          2d19h  production
```

### A.1.1 部署目标

再如，列出标签键名 env 的值为 production 或 dev，且不存在键名为 tier 的标签的所有 Pod 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l 'env in (production,dev),!tier'
NAME      READY   STATUS    RESTARTS   AGE     ENV        TIER
pod-example  1/1     Running   1          2d19h   production
```

为了避免shell解释器解析惊叹号（!），必须要为此类表达式使用单引号。

此外，Kubernetes 的诸多资源对象必须以标签选择器的方式关联到Pod资源对象，例如 Service、Deployment 和 ReplicaSet 类型的资源等，它们在 spec 字段中嵌套使用嵌套的“selector”字段，通过“matchLabels”来指定标签选择器，有的甚至还支持使用“matchExpressions”构造复杂的标签选择机制。

- matchLabels：通过直接给定键值对来指定标签选择器。
- matchExpressions：基于表达式指定的标签选择器列表，每个选择器都形如“{key:KEY\_NAME, operator: OPERATOR, values: [VALUE1, VALUE2, ...]}”，选择器列表间为“逻辑与”关系；使用 In 或 NotIn 操作符时，其 values 不强制要求为非空的字符串列表，而使用 Exists 或 DoesNotExist 时，其 values 必须为空。

下面所示的资源清单片段是一个示例，它同时定义了两类标签选择器：

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: Exists, values:}
```

标签赋予了 Kubernetes 灵活操作资源对象的能力，它也是 Service 和 Deployment 等核心资源类型得以实现的基本前提。

## 4.3.4 Pod 节点选择器 nodeSelector

Pod 节点选择器是标签及标签选择器的一种应用，它能够让 Pod 对象基于集群中工作节点的标签来挑选倾向运行的目标节点。

Kubernetes 的 kube-scheduler 守护进程负责在各工作节点中基于系统资源的可用性等标签挑选一个来运行创建的 Pod 对象，默认的调度器是 default-scheduler。Kubernetes 可将所有工作节点上的各系统资源抽象成资源池统一分配使用，因此用户无须关心 Pod 对象的具体运行位置也能良好工作。不过，事情总有例外，比如仅有部分节点拥有 Pod 对象依赖到的特殊硬件设备的情况，如 GPU 和 SSD 等。即便如此，用户也不应该静态指定 Pod 对象的运行位置，而是让 scheduler 基于标签和标签选择器为 Pod 用户挑选匹配的工作节点。

Pod 对象的 spec.nodeSelector 可用于定义节点标签选择器，用户事先为特定部分的 Node 资源对象设定好标签，而后配置 Pod 对象通过节点标签选择器进行匹配检测，从而完成节点亲和性调度。

为 Node 资源对象附加标签的方法同 Pod 资源，使用 “kubectl label nodes/NODE” 命令即可。例如，可为 kube-node-1.localdomain 和 kube-node-3.localdomain 节点设置 “disktype=ssd” 标签以标识其拥有 SSD 设备：

```
**[terminal]
**[delimiter $ ]**[command kubectl label nodes kube-node-1.localdomain disktype=ssd --overwrite
node/kube-node-1.localdomain labeled
**[delimiter $ ]**[command kubectl label nodes kube-node-3.localdomain disktype=ssd --overwrite
node/kube-node-3.localdomain labeled
```

查看具有键名 SSD 的标签的 Node 资源：

```
**[terminal]
**[delimiter $ ]**[command kubectl get nodes -l 'disktype:ssd' -L disktype]
NAME          STATUS   ROLES   AGE    VERSION   DISKTYPE
kube-node-1.localdomain  Ready    <none>  11d   v1.14.3   ssd
kube-node-3.localdomain  Ready    <none>  11d   v1.14.2   ssd
```

如果某 Pod 资源需要调度至这些具有 SSD 设备的节点之上，那么只需要为其使用 spec.nodeSelector 标签选择器即可，例如下面的资源清单文件 pod-with-nodeselector.yaml 示例：

### A.1.1 部署目标

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-nodeselector
  labels:
    env: testing
spec:
  containers:
  - name: myapp
    image: ikubernetes/myapp:v1
  nodeSelector:
    disktype: ssd
```

将如上资源清单中定义的 Pod 资源创建于集群中，通过检查其运行的节点即可判定调度效果

另外，手动测试和查看节点标签的读者或许已经注意到了，集群中的每个节点默认已经附带了多个标签，如 kubernetes.io/hostname、beta.kubernetes.io/os 和 beta.kubernetes.io/arch 等，这些标签也可以直接由 nodeSelector 使用，尤其是希望将 Pod 调度至某特定节点时，可以使用 kubernetes.io/hostname 直接绑定至相应的主机即可。不过，这种绑定至特定主机的需求还有一种更为简单的实现方式，即使用 spec.nodeName 字段直接指定目标节点。

## 4.4 资源注解

除了标签（label）之外，Pod 与其他各种资源还能使用资源注解（annotation）。与标签类似，注解也是“键值”类型的数据，不过它不能用于标签及挑选 Kubernetes 对象，仅可用于为资源提供“元数据”信息。另外，注解中的元数据不受字符数量的限制，它可大可小，可以为结构化或非结构化形式，也支持使用在标签中禁止使用的其他字段。

资源注解可由用户手动添加，也可由工具程序自动附加并使用它们。在 Kubernetes 的新版本中（alpha 或 beta 阶段）为某资源引入新字段时，常以注解的方式提供，以避免其增删等变动对用户带来困扰，一旦确定支持使用它们，这些新增字段就将再引入到资源中并淘汰相关的注解，另外，为资源添加注解也可以让其他用户快速了解资源的相关信息，例如其创建者的身份等。以下为常用的场景案例。

- 由声明式配置层（如 apply 命令）管理的字段：将这些字段定义的注解有助于识别由服务器或客户端设定的默认值、系统自动生成的字段以及由自动伸缩系统生成的字段。
- 构建、发行或镜像相关的信息，例如，时间戳、发行ID、Git分支、PR号码、镜像哈希及仓库地址等。
- 指向日志、监控、分析或审计仓库的指针。
- 由客户端库或工具程序生成的用于调试目的的信息：如名称、版本、构建信息等。
- 用户或工具程序的来源地信息，例如，来自其他生态系统组件的相关对象的 url。
- 轻量化滚动升级工具的元数据，如 config 及 checkpoints。
- 相关人员的电话号码等联系信息，或者指向类似信息的可寻址的目录条目，如网站站点。

## 4.4.1 查看资源注解

“kubectl get -o yaml” 和 “kubectl describe” 命令均能显示资源的注解信息。例如下面的命令显示的 pod-example 的注解信息：

```
**[terminal]
**[delimiter $ ]**[command kubectl describe pods pod-example]
Name:          pod-example
Namespace:     default
Priority:      0
PriorityClassName: <none>
Node:          kube-node-1.localdomain/192.168.0.134
Start Time:    Sun, 01 Dec 2019 21:34:46 +0800
Labels:        env=production
Annotations:   kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":'
Status:        Running
IP:           10.244.2.6
... 省略 ...
```

pod-example 此前由声明式配置命令 apply 创建，因此它在注解中保存了如上的相关信息以便在下次资源变动时进行版本对比。

## 4.4.2 管理资源注解

annotations 可在资源创建时使用 “metadata.annotations” 字段指定，也可随时按需在活动的资源上使用 “kubectl annotate” 命令行进行附加。例如，为 pod-example 重新进行注解：

```
**[terminal]
**[delimiter $ ]**[command kubectl annotate pods pod-example ilinux.io/created-by: cluster admin
pod/pod-example annotated
```

查看生成的注释信息：

```
**[terminal]
**[delimiter $ ]**[command kubectl describe pods pod-example | grep "Annotation"]
Annotations:           ilinux.io/created-by: cluster admin
```

如果需要在资源创建时的清单中指定，那么使用类似如下的方式即可：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  annotations:
    ilinux.io/created-by: cluster admin
spec:
  ... 省略 ...
```

## 4.5 Pod 对象的生命周期

Pod 对象自从其创建开始至其终止退出的时间范围称为其生命周期。在这段时间中，Pod 会处于多种不同的状态，并执行一些操作；其中，创建主容器（main container）为必需的操作，其他可选的操作还包括运行初始化容器（init container）、容器启动后钩子（post start hook）、容器的存活性探测（liveness probe）、就绪性探测（readiness probe）以及容器终止前钩子（post stop hook）等，这些操作是否执行则取决于 pod 的定义，如图 4-9 所示。

## 4.5.1 Pod 的相位

无论是类似前面几节中的由用户手动创建，还是通过 Deployment 等控制器创建，Pod 对象总是应该处于其生命进程中以下几个相位（phase）之一。

- Pending：API Server 创建了Pod资源对象并存入 etcd 中，但它尚未被调度完成，或者仍处于从仓库下载镜像的过程中。
- Running：Pod 已经被调度至某节点，并且所有容器都已经被 kubelet 创建完成。
- Succeeded：Pod 中的所有容器都已经成功终止并且不会被重启。
- Failed：所有容器都已经终止，但至少有一个容器终止失败，即容器返回了非0 值的退出状态或已经被系统终止。
- Unknown：API Server 无法正常获取到 Pod 对象的状态信息，通常是由于其无法与所在工作节点的 kubelet 通讯所致。



图 1.6.5.1 - Pod 的生命周期

Pod 的相位是在其生命周期中的宏观描述，并非对容器或 Pod 对象的综合汇总，而且相位的数量和含义被严格界定，它仅包含上面列举的相位值。

## 4.5.2 Pod的创建过程

Pod 是 Kubernetes 的基础单元，理解它的创建过程对于了解系统运行大有裨益。

图 4-10 描述了一个 Pod 资源对象的典型创建过程。

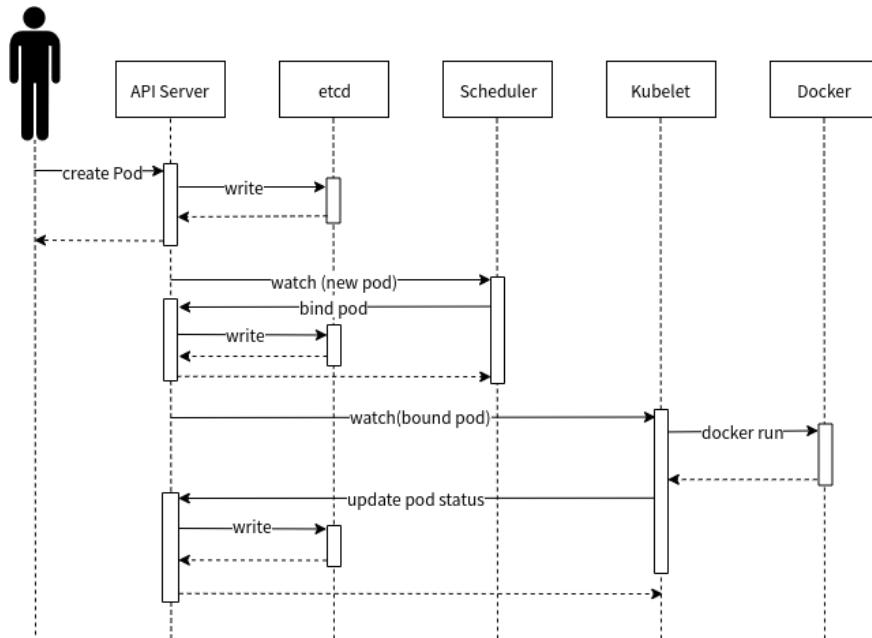


图 1.6.5.2 - Pod 资源对象创建过程

- 1) 用户通过 kubectl 或其他 API 客户端提交 Pod Spec 给 API Server。
- 2) API Server 尝试着将 Pod 对象的相关信息存入 etcd 中，待写入操作执行完成，API Server 即会返回确认信息至客户端。
- 3) API Server 开发反映 etcd 中的状态变化。
- 4) 所有的 Kubernetes 组件均使用“watch”机制来跟踪检查 API Server 上的相关的变动。
- 5) kube-scheduler（调度器）通过其“watcher”觉察到 API Server 创建了新的 Pod 对象但尚未绑定至任何工作节点。
- 6) kube-scheduler 为 Pod 对象挑选一个工作节点并将结果信息更新至 API Server。
- 7) 调度结果信息由 API Server 更新至 etcd 存储系统，而且 API Server 也开始反映此 Pod 的调度结果。
- 8) Pod 被调度到目标工作节点上的 kubelet 尝试在当前节点上调用 Docker 启动容器，并将容器的结果状态回送值 API Server。
- 9) API Server 将 Pod 状态信息存入 etcd 系统中。

### A.1.1 部署目标

- 10) 在 etcd 确认写入操作完成后，API Server 将确认信息发送至相关的 kubelet，事件将通过它被接受。

## 4.5.3 Pod 生命周期中的重要行为

除了创建应用容器（主容器及其辅助容器）之外，用户还可以为 Pod 对象定义其生命周期中的多种行文，如初始化容器、存活性探测及就绪性探测等。

### 1. 初始化容器

初始化容器（init container）即应用程序的主容器启动之前要运行的容器，常用于为主容器执行一些预置操作，它们具有两种典型特征。

- 1) 初始化容器必须运行完成直至结束，若某初始化容器运行失败，那么 Kubernetes 需要重启它直到成功完成。
- 2) 每个初始化容器都必须按定义的顺序串行运行。

有不少场景都需要在应用容器启动之前进行部分初始化操作，例如，等待其他关联自检服务可用、基于环境变量或配置模板为应用程序生成配置文件、从配置中心获取配置等。初始化容器的典型应用需求具体包含如下几种：

- 1) 用于运行特定的工具程序，出于安全等方面的原因，这些程序不适于包含再主容器镜像中。
- 2) 提供主容器镜像中不具备的工具程序或自定义代码。
- 3) 为容器镜像的构建和部署人员提供了分离、独立工作的途径，使得他们不必协同起来制作单个镜像文件。
- 4) 初始化容器和主容器处于不同的文件系统视图中，因此可以分别安全地使用敏感数据，例如 Secrets 资源。
- 5) 初始化容器要先于应用容器串行启动并运行完成，因此可用于延后应用容器的启动直至其依赖的条件得到满足。

Pod 资源的 “spec.initContainers” 字段以列表的形式定义可用的初始容器，其嵌套可用字段类似与 “spec.Containers”。下面的资源清单仅是一个初始化容器的使用示例，读者可自行创建并观察初始化容器的相关状态：

```

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
    - name: myapp-container
      image: ikubernetes/myapp:v1
  initContainers:
    - name: init-something
      image: busybox
      command: ['sh', '-c', 'sleep 10']

```

## 2. 生命周期钩子函数

生命周期钩子函数（lifecycle hook）是编程语言（如 Angular）中常用的生命周期管理的组件，它实现了程序运行周期中的关键时刻的可见性，并赋予用户为此采取某种行动的能力。类似地，容器生命周期钩子使它能够感知自身生命周期管理中的事件，并在相应的时刻到来时运行由用户指定的处理程序代码。Kubernetes 为容器提供了两种生命周期钩子。

- **postStart**: 于容器创建完成之后立即运行的钩子处理器（handler），不过 Kubernetes 无法确保它一定会于容器中的 ENTRYPPOINT 之前运行。
- **preStop**: 于容器终止操作之前立即运行的钩子处理器，它以同步的方式调用，因此在其完成之前会阻塞删除容器的操作的调用。

钩子处理器的实现方式有“Exec”和“HTTP”两种，前一种在钩子时间触发时直接在当前容器中运行由用户定义的命令，后一种则是在当前容器中向某URL发起HTTP请求。

`postStart` 和 `preStop` 处理器定义在容器的 `spec.lifecycle` 嵌套字段中，其使用方法如下面的资源清单所示，读者可自行创建相关的 Pod 资源对象，并验证其执行结果：

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
    - name: lifecycle-demo-container
      image: ikubernetes/myapp:v1
      lifecycle:
        postStart:
          exec:
            command: ["/bin/sh", "-c", "echo 'lifecycle hooks handlers' > /usr/
```

## 3. 容器探测

容器探测（container probe）是 Pod 对象生命周期中的一项重要的日常任务，它是 kubelet 对容器周期性执行的健康状态诊断，诊断操作由于容器的处理器（handler）进行定义。Kubernetes 支持三种处理器用于 Pod 探测。

- **ExecAction**: 在容器中执行一个命令，并根据其返回的状态码进行诊断的操作称为 Exec 探测，状态码为 0 表示成功，否则即为不健康的状态。
- **TCPSocketAction**: 通过与容器的某 TCP 端口尝试建立连接进行诊断，端口能够成功打开即为正常，否则为不健康状态。
- **HTTPGetAction**: 通过向容器IP地址的某指定端口的指定path发起 HTTP GET 请求进行诊断，响应码为 2xx 或 3xx 时即为成功，否则为失败。

### A.1.1 部署目标

任何一种探测方式都可能存在三种结果：“Success”（成功）、“Failure”（失败）或“Unknown”（未知），只有第一种结果表示成功通过检测。

Kubelet 可在活动容器上执行两种类型的检测：存活性检测（livenessProbe）和就绪性检测（readinessProbe）。

- 存活性检测：用于判断容器是否处于“运行”（Running）状态；一旦此类检测未通过，kubelet 将杀死容器并根据其 restartPolicy 决定是否将其重启；未定义存活性检测的容器的默认状态为“Success”。
- 就绪性检测：用于判断容器是否准备就绪并可对外提供服务；未通过检测的容器意味着尚未准备就绪，端点控制器（如 Service 对象）会将其IP从所有匹配到此 Pod 对象的 Service 对象的端点列表中移除；检测通过之后，会再次将其 IP添加至端点列表中。

存活性检测和就绪性检测相关的话题在后文的章节中还会有进一步的介绍。

## 4.5.4 容器的重启策略

容器进程发生崩溃或容器申请超出限制的资源等原因都可能会导致 Pod 对象的终止，此时是否应该重建该 Pod 对象则取决于其重启策略（restartPolicy）属性的定义。

- 1) Always：但凡 Pod 对象终止就将其重启，此为默认设定。
- 2) OnFailure：仅在 Pod 对象出现错误时方才将其重启。
- 3) Never：从不重启。

需要注意的是，restartPolicy 适用于 Pod 对象中的所有容器，而且它仅用于控制在同一节点上重新启动 Pod 对象的相关容器。首次需要重启的容器，将在其需要时立即进行重启，随后再次需要重启的操作将由 kubelet 延迟一段时间后进行，且反复的重启操作的延迟时长依次为 10秒、20秒、40秒、80秒、160秒和300秒，300秒是最大延迟时长。事实上，一旦绑定到一个节点，Pod 对象将永远不会被重新绑定到另一个节点，它要么被重启，要么终止，直到节点发生故障或被删除。

## 4.5.5 Pod 的终止过程

Pod 对象代表了在 kubernetes 集群节点上运行的进程，它可能曾用于处理生产数据或向用户提供服务等，于是，当 Pod 本身不再具有存在的价值时，如何将其优雅地终止就显得尤为重要了，而用户也需要能够在正常提交删除后可以获知其何时开始终止并最终完成。操作中，当用户提交删除请求之后，系统就会进行强制删除操作的宽限期倒计时，并将 TERM 信息发送给 Pod 对象的每个容器中的主进程。宽限期倒计时结束后，这些进程将收到强制终止的 KILL 信号，Pod 对象随即也将由 API Server 删除。如果在等待进程终止的过程中，kubelet 或容器管理器发生了重启，那么终止操作会重新获得一个满额的删除宽限期并重新执行删除操作。

如图 4-11 所示，一个典型的 Pod 对象终止流程具体如下：

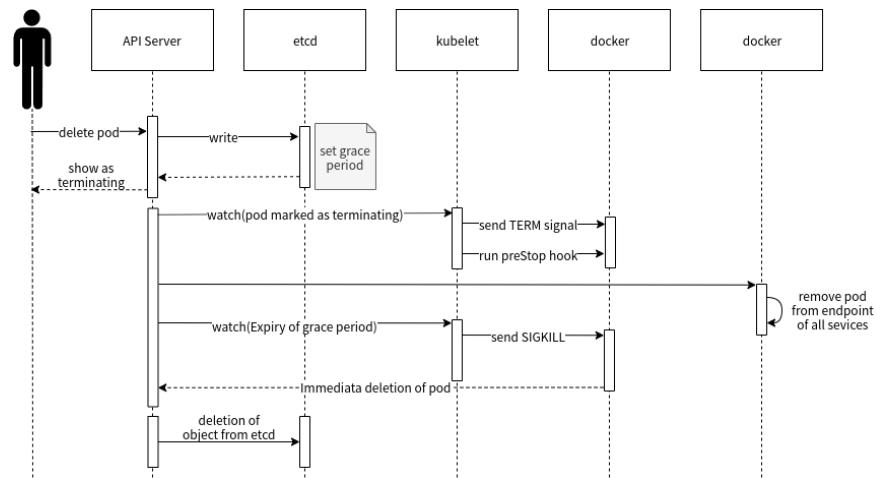


图 4-11: Pod 的终止过程

- 1) 用户发送删除 Pod 对象的命令。
- 2) API 服务器中的 Pod 对象会随着时间的推移而更新，在宽限期内（默认为30秒），Pod 被视为“dead”。
- 3) 将 Pod 标记为“Terminating”状态。
- 4) （与第3步同时运行）kubelet 在监控到 Pod 对象转为“Terminating”状态的同时启动 Pod 关闭程序。
- 5) （与第3步同时运行）端点控制器监控到 Pod 对象的关闭行为时将其从所有匹配到此端点的 Service 资源的端点列表中移除。
- 6) 如果当前当前 Pod 对象定义了 preStop 钩子处理器，则在其标记为“terminating”后即会以同步的方式启动执行；如若宽限期结束后，preStop 仍未执行结束，则第2步会被重新执行并额外获取一个时长为2秒的小宽限期。
- 7) Pod 对象中的容器进程收到 TERM 信号。
- 8) 宽限期结束后，若存在任何一个仍在运行的进程，那么 Pod 对象即会收到 SIGKILL 信号。

### A.1.1 部署目标

9) kubelet 请求 API Server 将此 Pod 资源的宽限期设置为0从而完成删除操作，它变得对用户不再可见。

默认情况下，所有删除操作的宽限期都是30秒，不过，`kubectl delete` 命令可以使用“`--grace-period=`”选项自定义其时长，若使用0值则表示直接强制删除指定的资源，不过，此时需要同时为命令使用“`--force`”选项。

## 4.6 Pod 存活性探测

有不少应用程序长时间持续运行后会逐渐转为不可用状态，并且仅能通过重启操作恢复，Kubernetes 的容器存活性检测机制可发现诸如此类的问题，并依据探测结果结合重启策略触发后续的行文。存活性探测是隶属于容器级别的配置，kubelet 可基于它判断何时需要重启一个容器。

Pod spec 为容器列表中的相应容器定义其专用的探针（存活性探测机制）即可启用存活性探测。目前，Kubernetes 的容器支持存活性探测的方法包含以下三种：ExecAction、TCPSocketAction 和 HTTPGetAction。

## 4.6.1 设置 exec 探针

exec 类型的探针通过在目标容器中执行由用户自定义的命令来判断容器的健康状态，若命令状态返回值为0则表示“成功”通过检测，其他值均为“失败”状态。

“spec.containers.livenessProbe.exec” 字段用于定义此类检测，它只有一个可用属性“command”，用于指定要执行的命令。下面是定义在资源清单文件 liveness-exec.yaml 中的示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec
  labels:
    test: liveness-exec
spec:
  containers:
    - name: liveness-exec-demo
      image: busybox
      args: ["/bin/sh", "-c", "touch /tmp/healthy; sleep 60; rm -rf /tmp/healthy"]
      livenessProbe:
        exec:
          command: ["test", "-e", "/tmp/healthy"]
```

上面的清单文件中定义了一个 Pod 对象，基于 busybox 镜像启动一个运行 `touch /tmp/healthy; sleep 60; rm -rf /tmp/healthy; sleep 600` 命令的容器，此命令在容器启动时创建 `/tmp/healthy` 文件，并于 60 秒之后将其删除。存活性运行探针“`test -e /tmp/healthy`”命令检查 `/tmp/healthy` 文件的存在性，若文件存在则返回状态码 0，表示成功通过测试。

首先，执行类似如下的命令，创建 Pod 对象 liveness-exec：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f liveness-exec.yaml]
pod/liveness-exec created
```

在60秒之内使用 “kubectl describe pods/liveness-exec” 查看其详细信息，其存活性探测不会出现错误。而超过60秒之后，再次运行“kubectl describe pods/liveness-exec” 查看其详细信息可以发现，存活性探测出现了规章，并且间隔更长一段时间之后再查看甚至还可以看到容器重启的相关信息。

### A.1.1 部署目标

Events:					
Type	Reason	Age	From	Message	FirstTimestamp
Normal	Scheduled	5m24s	default-scheduler		2019-12-06T12:05:36.000+08:00
Normal	Pulling	90s (x3 over 5m23s)	kubelet, kube-node-2.localdomain	Plugin "busybox" has been pulled	2019-12-06T12:05:36.000+08:00
Normal	Pulled	90s (x3 over 5m22s)	kubelet, kube-node-2.localdomain	Plugin "busybox" has been pulled	2019-12-06T12:05:36.000+08:00
Normal	Created	90s (x3 over 5m22s)	kubelet, kube-node-2.localdomain	Container "liveness-exec-demo" has been created	2019-12-06T12:05:36.000+08:00
Normal	Started	90s (x3 over 5m22s)	kubelet, kube-node-2.localdomain	Container "liveness-exec-demo" has started	2019-12-06T12:05:36.000+08:00
Warning	Unhealthy	1s (x9 over 4m21s)	kubelet, kube-node-2.localdomain	Liveness probe failed	2019-12-06T12:05:36.000+08:00
Normal	Killing	1s (x3 over 4m1s)	kubelet, kube-node-2.localdomain	Container "liveness-exec-demo" has been killed	2019-12-06T12:05:36.000+08:00

另外，输出信息的“Containers”一段中还清晰的显示了容器健康状态检测及状态变化的相关信息：容器当前处于“Running”状态，但前一次是为“Terminated”，原因是退出码为 137 的错误信息，它表示进程是被外部信号所终止的，。137 事实上是由两部分数字之和生成的：128+signum，其中 signum 是导致进程终止的信号的数字标识，9表示SIGKILL，这意味着进程被强制终止的：

Containers:	
liveness-exec-demo:	
Container ID:	docker://ed245e553dd6a98f6e19a001420196a54886593ad1804752385
Image:	busybox
Image ID:	docker-pullable://busybox@sha256:24fd20af232ca4ab5efbf1aeae
Port:	<none>
Host Port:	<none>
Args:	
/bin/sh	
-c	
touch /tmp/healthy; sleep 60; rm -rf /tmp/healthy; sleep 600	
State:	Running
Started:	Fri, 06 Dec 2019 12:05:36 +0800
Last State:	Terminated
Reason:	Error
Exit Code:	137
Started:	Fri, 06 Dec 2019 12:03:36 +0800
Finished:	Fri, 06 Dec 2019 12:05:36 +0800
Ready:	True
Restart Count:	4
Liveness:	exec [test -e /tmp/healthy] delay=0s timeout=1s period=10s

待容器重启完成后再次查看，容器已经处于正常运行状态，直到文件再次被删除，存活探测失败而重启。从下面的命令显示可以看出，liveness-exec 在 13 分钟内已然重启了 6 次：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods liveness-exec]
NAME        READY   STATUS    RESTARTS   AGE
liveness-exec   1/1     Running   6          13m
```

需要特别说明的是，exec 指定的命令运行于容器中，会消耗容器的可用资源配置，另外，考虑到探针操作的效率本身等因素，探针操作的命令应该尽可能简单和轻量。

## 4.6.2 设置 HTTP 探针

基于 HTTP 的探测（HTTPGetAction）向目标容器发起一个HTTP请求，根据其相应码进行结果判定，响应码形如 2xx 或 3xx 时表示检测通过。

“spec.containers.liveness.Probe.httpGet” 字段用于定义此类检测，它的可用配置字段包括如下几个。

- host <string>：请求的主机地址，默认为 Pod IP；也可以在 httpHeaders 中使用 “Host:” 来定义
- httpHeaders <[]Object>：自定义的请求报文首部。
- path <string>：请求的 HTTP 资源路径，即 URL path。
- scheme : 建立连接使用的协议，仅可为 HTTP 或 HTTPS，默认为 HTTP。

下面是一个定义再资源清单文件 liveness-http.yaml 中的示例，它通过 lifecycle 中 postStart hook 创建了一个专用于 httpGet 测试的页面文件 healthz:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness-http-demo
    image: nginx:1.12-alpine
    ports:
    - name: http
      containerPort: 80
  lifecycle:
    postStart:
      exec:
        command: ["/bin/sh", "-c", "echo Healthy > /usr/share/nginx/html/healthz"]
  livebessProbe:
    httpGet:
      path: /healthz
      port: http
      scheme: HTTP
```

上述清单文件中定义了 httpGet 测试中，请求的资源路径为 “/healthz”，地址默认为 Pod IP，端口使用了容器中定义的端口名称http，这也是明确为容器指明要暴露的端口的用途之一。首先创建此 Pod 对象：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f liveness-http.yaml]
pod/liveness-http created
```

### A.1.1 部署目标

而后查看其健康状态检测相关信息，健康状态检测正常时，容器也将正常运行：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl describe pods liveness-http]
Name:           liveness-http
Namespace:      default
Priority:       0
PriorityClassName: <none>
Node:           kube-node-3.localdomain/192.168.0.116
Start Time:     Fri, 06 Dec 2019 12:50:51 +0800
Labels:         test=liveness
Annotations:    kubectl.kubernetes.io/last-applied-configuration:
                  {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"labels":{"test":"liveness"},"name":"liveness-http","namespace":"default","uid":"b621747212dc32244818d389579dadc265332d27d6f850dea"},}
Status:         Running
IP:             10.244.4.10
Containers:
  liveness-http-demo:
    Container ID:   docker://b621747212dc32244818d389579dadc265332d27d6f850dea
    Image:          nginx:1.12-alpine
    Image ID:       docker-pullable://nginx@sha256:3a7edf11b0448f171df8f4acac88
    Port:           80/TCP
    Host Port:     0/TCP
    State:          Running
    Started:       Fri, 06 Dec 2019 12:50:53 +0800
    Ready:          True
    Restart Count:  0
    Liveness:       http-get http://:http/healthz delay=0s timeout=1s period=10s
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fwfzr (rw)
Conditions:
  Type        Status
  Initialized  True
  Ready        True
  ContainersReady  True
  PodScheduled  True
Volumes:
  default-token-fwfzr:
    Type:           Secret (a volume populated by a Secret)
    SecretName:    default-token-fwfzr
    Optional:      false
    QoS Class:     BestEffort
    Node-Selectors: <none>
    Tolerations:   node.kubernetes.io/not-ready:NoExecute for 300s
                  node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type  Reason  Age   From           Message
  ----  -----  ----  --  -----
  Normal Scheduled  89s   default-scheduler   Successfully assigned liveness-http to kube-node-3.localdomain
  Normal  Pulling   88s   kubelet, kube-node-3.localdomain  Pulling image "nginx:1.12-alpine"
```

### A.1.1 部署目标

```
Normal  Pulled     87s   kubelet, kube-node-3.localdomain  Successfully pulled "nginx" from "nginx-74d5574f6c-5q7t5"
Normal  Created    87s   kubelet, kube-node-3.localdomain  Created container nginx
Normal  Started    87s   kubelet, kube-node-3.localdomain  Started container nginx
```

接下来借助于“kubectl exec”命令删除经由 postStart hook 创建的测试页面 healthz：

```
**[terminal]
**[delimiter $ ]**[command kubectl exec liveness-http rm /usr/share/nginx/html/index.html]
```

而后再次使用“kubectl describe pods liveness-http”查看其详细的状态信息，事件输出中的信息可以表明探测测试失败，容器被杀掉后进行了重新创建：

Events:					
Type	Reason	Age	From	Message	Last Seen At
---	---	---	---	---	---
Normal	Scheduled	5m27s	default-scheduler	Scheduled pod to node kube-node-3.localdomain	2023-01-12T10:27:45Z
Normal	Pulling	5m26s	kubelet, kube-node-3.localdomain	Pulling image "nginx:latest"	2023-01-12T10:26:45Z
Normal	Pulled	5m25s	kubelet, kube-node-3.localdomain	Successfully pulled "nginx" from "nginx-74d5574f6c-5q7t5"	2023-01-12T10:25:45Z
Warning	Unhealthy	3s (x3 over 23s)	kubelet, kube-node-3.localdomain	Liveness probe failed: http://127.0.0.1:80/liveness	2023-01-12T10:25:45Z
Normal	Killing	3s	kubelet, kube-node-3.localdomain	Container will be killed	2023-01-12T10:25:45Z
Normal	Pulled	3s	kubelet, kube-node-3.localdomain	Successfully pulled "nginx" from "nginx-74d5574f6c-5q7t5"	2023-01-12T10:25:45Z
Normal	Created	2s (x2 over 5m25s)	kubelet, kube-node-3.localdomain	Created container nginx	2023-01-12T10:25:45Z
Normal	Started	2s (x2 over 5m25s)	kubelet, kube-node-3.localdomain	Started container nginx	2023-01-12T10:25:45Z

一般来说，HTTP 类型的探测操作应该针对专用的 URL 路径进行，例如，前面示例中特别为其准备的“/healthz”。另外，此 URL 路径对应的 Web 资源应该以轻量化的方式在内部对应用程序的各关键组件进行全面检测以确保它们可正常向客户端提供完整的服务。

需要注意的是，这种检测方式仅对分层架构中的当前一层有效，例如，它能检测应用程序工作正常与否的状态，但重启操作却无法解决其后端服务（如数据库或缓存服务）导致的故障。此时，容器可能会被一次次的重启，直到后端服务恢复正常为止。其它两种检测方式也存在类似的问题。

## 4.6.3 设置 TCP 探针

基于 TCP 的存活性探测（TCP Socket Action）用于向容器特定端口发起 TCP 请求并尝试建立连接进行结果判定，连接建立成功即为通过检测。相比较来说，它比基于 HTTP 的探测更高效、更节约资源，但精确度略低，毕竟连接建立成功未必意味着页面资源可用。“spec.containers.livenessProbe.tcpSocket” 字段用于定义此类检测，它主要包含以下两个可用的属性。

- 1) host <string> : 请求连接的目标IP地址，默认为 Pod IP
- 2) port <string> : 请求连接的目标端口，必选字段

下面是一个定义在资源清单文件 liveness-tcp.yaml 中的示例，它向 Pod IP 的 80/tcp 端口发起连接请求，并根据连接建立的状态判定测试结果：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-tcp
spec:
  containers:
  - name: liveness-tcp-demo
    image: nginx:1.12-alpine
    ports:
    - name: http
      containerPort: 80
  livenessProbe:
    tcpSocket:
      port: http
```

这里不再给出其具体的创建与测试过程，有兴趣的读者可自行进行测试。

## 4.6.4 存活性探测行为属性

使用 kubectl describe 命令查看配置了存活性探测的 Pod 对象的详细信息时，其相关容器中会输出类似如下的一行内容。

```
Liveness: http-get http://:http/healthz delay=0s timeout=1s period=10s #s
```

它给出了探测方式及其额外的配置属性 delay、timeout、period、success 和 failure 及其各自的相关属性值。用户没有明确定义这些属性字段时，它们会使用各自的默认值，例如上面显示出的设定。这些属性信息可通过“spec.containers.livenessProbe”的如下属性字段来给出。

- `initialDelaySeconds <integer>`：存活性探测延迟时长，即容器启动多久之后再开始第一次探测操作，显示为 `delay` 属性；默认为 0 秒，即容器启动后立刻便开始进行探测。
- `timeoutSeconds <integer>`：存活性探测的超时时长，显示为 `timeout` 属性，默认为 1s，最小值也为 1s。
- `periodSeconds <integer>`：存活性探测的频度，显示为 `period` 属性，默认为 10s，最小值为 1s；过高的频率会对 Pod 对象带来较大的额外开销，而过低的频率又会使得对错误的反映不及时。
- `successThreshold <integer>`：处于失败状态时，探测操作至少连续多少次的成功才被认为是通过检测，显示为 `#success` 属性，默认值为 1，最小值也为 1。
- `failureThreshold <integer>`：处于成功状态时，探测操作至少连续多少次的失败才被视为是检测不通过，显示为 `#failure` 属性，默认值为 3，最小值为 1。

例如，这里可将 4.6.1 节中清单文件中定义的探测示例重新定义为如下所示的内容：

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-exec
  labels:
    test: liveness-exec
spec:
  containers:
    - name: liveness-exec-demo
      image: busybox
      args: ["/bin/sh", "-c", "touch /tmp/healthy; sleep 60; rm -rf /tmp/healthy"]
      livenessProbe:
        exec:
          command: ["test", "-e", "/tmp/healthy"]
        initialDelaySeconds: 5
        timeoutSeconds: 2
        periodSeconds: 5
```

### A.1.1 部署目标

根据修改的清单文件再次创建 Pod 对象并进行效果测试，可以从输出的详细信息中看出 liveness 已经更新到自定义的属性，其内容如下所示。具体过程这里不再给出，请感兴趣的读者自行测试：

```
liveness: exec [test -e /tmp/healthy] delay=5s timeout=2s period=5s #successif
```

## 4.7 Pod 就绪性探测

Pod 对象启动后，容器应用通常需要一段时间才能完成其初始化过程，例如加载配置或数据，甚至有些程序需要运行某类预热过程，若在此阶段完成之前即接入客户端的请求，势必会因为等待太久而影响用户体验。因此，应该避免于 Pod 对象启动后立即让其处理客户端请求，而是等待容器初始化工作执行完成并转为“就绪”状态，尤其是存在其他提供相同服务的 Pod 对象的场景更是如此。

与存活性探测机制类似，就绪性探测使用来判断容器就绪与否的周期性（默认周期为10秒钟）操作，它用于探测容器是否已经初始化完成并可服务于客户端请求，探测操作返回“success”状态时，即为传递容器已经“就绪”的信号。

与存活性探测机制相同，就绪性探测也支持 Exec、HTTP GET 和 TCP Socket 三种探测方式，且各自的定义机制也都相同。但与存活性探测触发的操作不同的是，探测失败时，就绪性探测不会杀死或重启容器以保证其健康性，而是通知其尚未就绪，并触发依赖于其就绪状态的操作（例如，从 Service 对象中移除此 Pod 对象）以确保不会由客户端请求接入此 Pod 对象。不过，即便是在运行过程中，Pod 就绪性探测依然有其价值所在，例如 Pod A 依赖到的 Pod B 因网络故障等原因而不可用时，Pod A 上的服务应该转为未就绪状态，以免无法向客户端提供完整的响应。

将容器定义中的 livenessProbe 字段替换为 readinessProbe 即可定义出就绪性探测的配置，一个简单的示例如下面的配置清单（readiness-exec.yaml）所示，它会在 Pod 对象创建完成 5 秒钟后使用 test -e /tmp/ready 命令来探测容器的就绪性，命令执行成功即为就绪，探测周期为5秒钟：

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-exec
  labels:
    test: readiness-exec
spec:
  containers:
    - name: readiness-demo
      image: busybox
      args: ["/bin/sh", "-c", "while true; do rm -f /tmp/ready; sleep 30; touch /tmp/ready; done"]
      readinessProbe:
        exec:
          command: ["test", "-e", "/tmp/ready"]
        initialDelaySeconds: 5
        periodSeconds: 5
```

首先，使用“kubectl create”命令将资源配置清单定义的资源创建到集群中：

```
**[terminal]
**[delimiter $ ]**[command kubectl create -f readiness-exec.yaml]
pod/readiness-exec created
```

### A.1.1 部署目标

接着，运行“kubectl get -w”命令监视其资源变动信息，由如下命令结果可知，尽管 Pod 对象处于“Running”状态，但直到就绪探测命令执行成功后，Pod 资源才转为“就绪”：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l test=readiness-exec -w]
NAME        READY   STATUS    RESTARTS   AGE
readiness-exec   0/1     Running   0          25s
readiness-exec   1/1     Running   0          33s
```

另外，还可以从 Pod 对象的详细信息中得到类似如下的表示其已经处于就绪状态的信息片段：

```
Ready:           True
Restart Count:  0
Readiness:      exec [test -e /tmp/ready] delay=5s timeout=1s period=5s #success=1
```

这里需要特别提醒读者的是，未定义就绪性探测的 Pod 对象在 Pod 进入“Running”状态后将立即就绪，在容器需要时间进行初始化的场景中，在应用真正就绪之前必然无法正常响应客户端请求，因此，生产实践中，必须为关键性 Pod 资源中的容器定义就绪性探测机制。其探测机制的定义请参考 4.6 节中的定义。

## 4.8 资源需求及资源限制

在 Kubernetes 上，可由容器或 Pod 请求或消费的“计算资源”是指 CPU 和内存 (RAM)，这也是目前仅有的受支持的两种类型。相比较来说，CPU 属于可压缩 (compressible) 型资源，即资源额度可按需收缩，而内存（当前）则是不可压缩资源，对其执行收缩操作可能会导致某种程度的问题。

目前来说，资源隔离尚且属于容器级别，CPU 和内存资源的配置需要在 Pod 中的容器上进行，每种资源均可由 “requests” 属性定义其请求的确保可用值，即容器运行可能用不到这些额度的资源，但用到时必须要确保有如此多的资源可用，而 “limits” 属性则用于限制资源可用的最大值，即硬限制，如图 4-12 所示。不过，为了表述方便，人们通常仍然把资源配置称作 Pod 资源的请求和限制，只不过它是指 Pod 内所有容器上某种类型资源的请求和限制的总和。

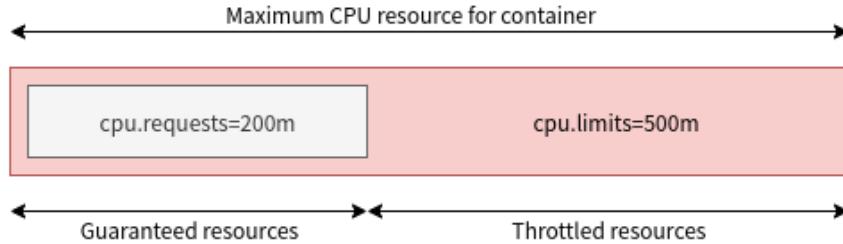


图 1.6.8 - 容器资源需求及资源限制示意图

在 Kubernetes 系统上，1各单位的CPU相当于虚拟机上的1颗虚拟CPU (vCPU) 或物理机上的一个超线程 (Hyperthread, 或称为一个逻辑 CPU)，它支持分数计量方式，一个核心 (1 core) 相当于 1000 个微核心 (millicores)，因此 500 相当于 0.5 个核心，即二分之一核心。内存的计量方式与日常使用方式相同，默认单位是字节，也可以使用 E、P、T、G、M 和 K 作为单位后缀，或 Ei、Pi、Ti、Gi、Mi 和 Ki 形式的单位后缀。

## 4.8.1 资源需求

下面的是示例中，自主式 Pod 要求为 stress 容器确保 128Mi 的内存及五分之一个 CPU 核心 (200m) 资源可用，它运行 stress-ng 镜像启动一个进程 (-m 1) 进行内存性能压力测试，满载测试时它也会尽可能多的占用 CPU 资源，另外再启动一个专用的 CPU 压力测试进程 (-c 1)。stress-ng 是一个多功能系统压力测试工具，master/worker 模型，Master 为主进程，负责生成和控制子进程，worker 是负责执行各类特定测试的子进程，例如测试 CPU 的子进程，以及测试 RAM 的子进程等：

```
apiVersion: v1
kind: Pod
metadata:
  name: stress-pod
spec:
  containers:
  - name: stress
    image: ikubernetes/stress-ng
    command: ["/usr/bin/stress-ng", "-m 1", "-c 1", "-metrics-brief"]
    resources:
      requests:
        memory: "128Mi"
        cpu: "200m"
```

上面的配置清单中，其请求使用的 CPU 资源大小为 200m，这意味着一个 CPU 核心足以确保其以期望的最快方式运行。另外，配置清单中期望使用的内存大小为 128Mi，不过其运行时未必会真的用到这么多。考虑到内存为非压缩性资源，其超出指定的大小再运行时存在被 OOM killer 杀死的可能性，于是请求值也应该就是其理想中使用的内存空间上限。

接下来创建并运行此 Pod 对其资源限制效果尽心检查。需要特别说明的是，笔者当前使用的系统环境中，每个节点的可用 CPU 核心数均为 8，物理内存空间为 16GB：

```
**[terminal]
**[delimiter $ ]**[command kubectl create -f pod-resources-test.yaml]
pod/stress-pod created
```

而后在 Pod 资源的容器内运行 top 命令观察其 CPU 及内存资源的占用状态，如下所示，其中 {stress-ng-vm} 是执行内存压测的子进程，它默认使用 256m 的内存空间，{stress-ng-cpu} 是执行 CPU 压测的专用子进程：

```
**[terminal]
**[delimiter $ ]**[command kubectl exec stress-pod -- top]
Mem: 1731528K used, 6277504K free, 28504K shrd, 2104K buff, 929196K cached
CPU: 55% usr 7% sys 0% nic 37% idle 0% io 0% irq 0% sirq
Load average: 1.42 0.50 0.21 5/390 18
 PID  PPID USER      STAT  VSZ %VSZ CPU %CPU COMMAND
   8    7 root      R    262m  3%   2 13% {stress-ng-vm} /usr/bin/stress-n
   6    1 root      R    6884  0%   1 13% {stress-ng-cpu} /usr/bin/stress-n
   1    0 root      S    6244  0%   3  0% /usr/bin/stress-ng -m 1 -c 1 --me
....
```

top 命令的输出结果显示，每个测试进程的CPU占用率为 23%（实际为 12.5%），{stress-ng-vm}的内存占用量为 262m (VSZ)，此两项资源占用量都远超其请求的用量，原因是stress-ng会在可用的范围内尽可能多的占用相关资源。两个测试线程分布与两个CPU核心资源充裕，虽然容器的内存用量远超 128M，但它依然可运行。一旦资源紧张时，节点仅保证容器有五分之一各CPU核心可用，对于有着8个核心的节点来说，它的占用率为 2.5%，于是每个进程为 1.25%，多占用的资源会被压缩。内存为非可压缩性资源，所以此Pod在内存资源紧张时可能会因为 OOM 被杀死 (killed)。

对于压缩型的资源CPU来说，未定义其请求用量以确保其最小的可用资源时，他可能会被其他的Pod资源压缩至极低的水平，甚至会达到Pod不能被调度运行的境地。而对与非压缩性资源来说，内存资源在任何原因导致的紧缺情况下都有可能导致相关的进程被杀死。因此，在 Kubernetes 系统上运行关键型业务相关的Pod时必须使用requests属性为容器定义资源的确保可用量。

集群中的每个节点都拥有定量的CPU和内存资源，调度 Pod 时，仅那些被请求资源的余量可容纳当前被调度的 Pod 的请求量的节点才可作为目标节点。也就是说，Kubernetes 的调度器会根据容器的 requests 属性中定义的资源需求量来判定哪些节点可接受运行相关的 Pod 资源，而对于一个节点的资源来说，每运行一个 Pod 对象，其 requests 中定义的请求量都要被预留，直到被所有 Pod 对象瓜分完毕为止。

## 4.8.2 资源限制

容器的资源需求仅能达到为其保证可用的最少资源量的目的，它并不会限制容器的可用资源上限，因此对因应用本身存在 Bug 等多种原因而导致的系统资源被长时间占用的情况则无计可施，这就需要通过 limits 属性为容器定义资源的最大可用量。资源分配时，可压缩型资源 CPU 的控制阀门可自由调节，容器进程无法获得超过其 CPU 配额的可用时间。不过，如果进程申请分配超出其 limits 属性定义的硬限制的内存资源时，它将被 OOM Killer 杀死，不过，随后可能会被其控制进程所重启，例如，容器进程的 Pod 对象会被杀死并重启（重启策略为 Always 或 OnFailure 时），或者时容器进程的子进程被其父进程所重启。

下面的配置清单文件（memleak-pod.yaml）中定义了如何使用 saadali/simmemleak 镜像运行一个 Pod 对象，它模拟内存泄露操作不断地申请使用内存资源，直到超出 limits 属性中 memory 字段设定的值而导致“OOMKilled”为止：

```
apiVersion: v1
kind: Pod
metadata:
  name: memleak-pod
  labels:
    app: memleak
spec:
  containers:
    - name: simmemleak
      image: saadali/simmemleak
      resources:
        requests:
          memory: "64Mi"
          cpu: "1"
        limits:
          memory: "64Mi"
          cpu: "1"
```

下面测试其运行效果，首先将配置清单中定义的资源使用下面的命令创建于集群中：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f memleak-pod.yaml]
pod/memleak-pod created
```

pod 资源的默认重启策略为 Always，于是在 memleak 因内存资源达到硬限制而被终止后会立即重启，因为用户很难观察到其因 OOM 而被杀死的相关信息。不过，多次重复的因为内存资源消耗尽而重启会出发 Kubernetes 系统的重启延迟机制，即每次重启的时间间隔会不断地拉长。于是，用户看到的 Pod 资源的相关状态通常为“CrashLoopBackOff”：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f memleak-pod.yaml]
NAME      READY   STATUS          RESTARTS   AGE
memleak-pod  0/1    CrashLoopBackOff  5          4m11s
```

Pod 资源首次的重启将在 crash 后立即完成，若随后再次 crash，那么其重启操作会延迟10秒进行，随后的延迟时长会逐渐增加，依次为20秒，40秒，80秒，160秒和300秒，随后的延迟将固定在5分钟的时长之上而不再增加，直到其不再 crash 或者 delete 为止。describe 命令可以显示其状态的详细信息，其部分内容如下所示：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl describe pods memleak-pod]

Name:           memleak-pod
Namespace:      default
Priority:       0
PriorityClassName: <none>
Node:           kube-node-1.localdomain/192.168.0.134
Start Time:     Sat, 07 Dec 2019 10:56:17 +0800
Labels:         app=memleak
Annotations:    kubectl.kubernetes.io/last-applied-configuration:
                  {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"labels":{"app":"memleak"},"name":"memleak-pod","namespace":"default","uid":"d4f3a2c5-1a2e-4a2b-8a2d-0a2e2a2f2a2f"}, "resourceVersion": "1000000000000000000", "selfLink": "/apis/.../namespaces/default/pods/memleak-pod"}, "status":{"podIP": "10.244.2.9", "phase": "Running"}, "version": "v1"}, kubectl.kubernetes.io/pod-updated-at: "2019-12-07T11:02:02Z"
Status:         Running
IP:            10.244.2.9

Containers:
  simmemleak:
    Container ID:  docker://465caf91fd04e71f16e413401f5b3ebb43eae6c2942e8a400
    Image:          saadali/simmemleak
    Image ID:       docker-pullable://saadali/simmemleak@sha256:5cf58299a7698bc...
    Port:          <none>
    Host Port:     <none>
    State:         Waiting
      Reason:      CrashLoopBackOff
    Last State:    Terminated
      Reason:      OOMKilled
      Exit Code:   137
    Started:       Sat, 07 Dec 2019 11:02:01 +0800
    Finished:      Sat, 07 Dec 2019 11:02:02 +0800
    Ready:         False
    Restart Count: 6
    Limits:
      cpu:        1
      memory:    64Mi
    Requests:
      cpu:        1
      memory:    64Mi
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fwfzr (rw)
Conditions:
  Type        Status
  Initialized  True
  Ready        False
  ContainersReady  False
  PodScheduled  True

Volumes:
  default-token-fwfzr:
    Type:           Secret (a volume populated by a Secret)
    SecretName:    default-token-fwfzr
```

### A.1.1 部署目标

```
Optional:    false
QoS Class:   Guaranteed
Node-Selectors: <none>
Tolerations:  node.kubernetes.io/not-ready:NoExecute for 300s
               node.kubernetes.io/unreachable:NoExecute for 300s

Events:
Type  Reason  Age           From
----  -----  --           --
Normal  Scheduled  8m20s      default-scheduler
Normal  Pulling   7m32s (x4 over 8m19s)  kubelet, kube-node-1.localdomain
Normal  Pulled    7m32s (x4 over 8m17s)  kubelet, kube-node-1.localdomain
Normal  Created   7m31s (x4 over 8m17s)  kubelet, kube-node-1.localdomain
Normal  Started   7m31s (x4 over 8m17s)  kubelet, kube-node-1.localdomain
Warning BackOff   3m12s (x28 over 8m14s)  kubelet, kube-node-1.localdomain
```

如上述命令结果所显示的，OOMKilled 表示容器因内存耗尽而被终止，因此，为 limits 属性中的 memory 设置一个合理值至关重要。与 requests 不同的是，limits 并不影响 Pod 的调度结果，也就是说，一个节点上的所有 Pod 对象的 limits 数量之和可以大于节点所拥有的资源量，即支持资源的过载使用（overcommitted）。不过，这么一来一旦资源耗尽，尤其是内存资源耗尽，则必然会有容器因 OOMKilled 而终止。

另外需要说明的是，Kubernetes 仅会确保 Pod 能够获得它们请求（requests）的 CPU 时间额度，它们能否获得额外（throttled）的 CPU 时间，则取决于其他正在运行的作业对 CPU 资源的占用情况。例如，对于总数为 1000m 的 CPU 资源来说，容器A请求使用 200m，容器B请求使用 500m，在不超出它们各自的最大限额的前提下，余下的300m在双方都需要时会以 2:5(200m:500m) 的方式进行分配。

## 4.8.3 容器的可见资源

细心的读者可能已经发现了这一点：于容器中运行的 top 等命令观察资源可用量信息时，即便定义了 requests 和 limits 属性，虽然其可用资源受限于此两个文明属性的定义，但容器中可见的资源量依然是节点级别的可用总量。例如，为前面定义的 stress-pod 添加如下 limits 属性定义：

```
limits:
  memory: "512Mi"
  cpu: "400m"
```

重新创建 stress-pod 对象，并于其容器内分别列出容器可见的内核和CPU资源总量，命令及结果如下所示：

```
**[terminal]
**[delimiter $ ]**[command kubectl exec stress-pod -- cat /proc/meminfo | grep
MemTotal:      8009032 kB
**[delimiter $ ]**[command kubectl exec stress-pod -- cat /proc/cpuinfo | grep
4
```

命令结果中显示其可用内存资源总量为8009032KB（8GB），CPU核心数为8个，这是节点级的资源数量，而非由容器的 limits 所定义的 512Mi 和 400m。其实，这种结果不仅仅使得其查看命令的显示结果看起来有些奇怪，而且对有些容器应用的配置也会带来不小的负面影响。

较为经典的是在 Pod 中运行 Java 应用程序时，若未使用 “-Xmx” 选项指定 JVM 的堆内存可用总量，它默认会设置为主机内存总量的一个空间比例（如30%），这会导致容器中的应用程序申请内存资源时将会达到上限而转为 OOMKilled。另外，即便使用了“-Xmx”选项设置其堆内存上限，但它对于非堆内存的可用空间不会产生任何限制作用，结果是仍然存在达到容器内存资源上限的可能性。

另一个颇具代表性的场景是于 Pod 中运行的 Nginx 应用，在配置参数 worker\_processes 的值为 “auto” 时，主进程会创建于 Pod 中能够访问到的 CPU 核心数相同数量的 worker 进程。若 Pod 的实际可用 CPU 核心远低于主机级别的数量时，那么这种设置在较大的并发访问负荷下会导致严重的资源竞争，并将带来更多的内存资源消耗。一个较为妥当的解决方案是使用 Downward API 将 limits 定义的资源量暴露给容器，这一点将在后面的章节中予以介绍。

## 4.8.4 Pod 的服务质量类比

前面曾提到过，Kubernetes 允许节点资源对 limits 的过载使用，这意味着节点无法同时满足其上的所有 Pod 对象以资源满载的方式运行。于是，在内存资源紧缺时，应该以何种次序先后终止哪些 Pod 对象？Kubernetes 无法自行对此作出决策，它需要借助于 Pod 对象的优先级完成判定。根据 Pod 对象的 requests 和 limits 属性，Kubernetes 将 Pod 对象归类到 BestEffort、Burstable 和 Guaranteed 三个服务质量（Quality of Service, QoS）类别下，具体说明如下。

- **Guaranteed**：每个容器都为 CPU 资源设置了具有相同值的 requests 和 limits 属性，以及每个容器都为内存资源设置了具有相同值的 requests 和 limits 属性的 Pod 资源会自动归属于此类别，这类 Pod 资源具有最高优先级。
- **Burstable**：至少有一个容器设置了 CPU 或内存资源的 requests 属性，但不满足 Guaranteed 类别要求的 Pod 资源将自动归属于此类别，它们具有中等优先级。
- **BestEffort**：未为任何一个容器设置 requests 或 limits 属性的 Pod 资源将自动归属于此类别，它们的优先级为最低级别。

内存资源紧缺时，BestEffort 类别的容器将首当其冲地被终止，因为系统不为其提供任何级别的资源保证，但换来的好处是，它们能够在可用时做到尽可能多的占用资源。若已然不存在任何 BestEffort 类别的容器，则接下来是有着中等优先级的 Burstable 类别的 Pod 被终止。Guaranteed 类别的容器拥有最高优先级，它们不会被杀死，除非其内存资源需求超限，或者 OOM 时没有其他更低优先级的 Pod 资源存在。

每个运行状态容器都有其 OOM 得分，得分越高越会被优先杀死。OOM 得分主要根据两个纬度进行计算：由 QoS 类别继承而来的默认分值和容器的可用内存资源比例。同等类别的 Pod 资源的默认分值相同，下面的代码片段取自 pkg/kubelet/qos/policy.go 源码文件，它们定义的各种类别的 Pod 资源的 OOM 调节（Adjust）分值，即默认分值。其中，Guaranteed 类别的 Pod 资源的 Adjust 分值为 -998 分，而 BestEffort 分别时默认分值为 1000，Burstable 类别的 Pod 资源的 Adjust 分值则经由相应的算法计算得出：

```
const {
    PodinfraOOMAdj          int = -998
    KubeletOOMScoreAdj      int = -999
    DockerOOMScoreAdj       int = -999
    KubeProxyOOMScoreAdj    int = -999
    guaranteedOOMScoreAdj   int = -998
    besteffortOOMScoreAdj   int = 1000
}
```

因此，同等级别优先级的 Pod 资源在 OOM 时，与自身的 requests 属性相比，其内存占用比例最大的 Pod 对象将被首先杀死。例如，图 4-13 中的同属于 Burstable 类别的 Pod A 将先于 Pod B 被杀死，虽然其内存用量小，但与自身的 requests 值相比，它的占用比例 95% 要大于 Pod B 的 80%。

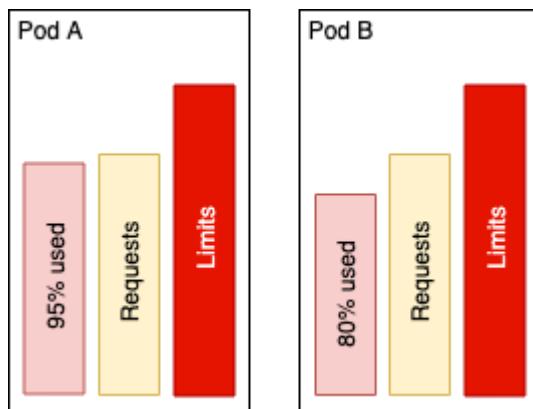


图 1.6.8.4 - 资源需求和资源限额及OOM

需要特别说明的是，OOM 是内存耗尽时的处理机制，它们与可压缩型资源 CPU 无关，因此 CPU 资源的需求无法得到保证时，Pod 仅仅是暂时获取不到相应的资源而已。

## 第五章 Pod 控制器

自主式 Pod 对象由调度器绑定至工作节点后即由相应节点上的 kubelet 负责监控其容器的存活性，容器主进程崩溃后，kubelet 能够自动重启相应的容器。不过，kubelet 对非主进程崩溃类的容器错误却无从感知，这依赖于用户为 Pod 资源对象自定义的存活性探测（liveness probe）机制，以便 kubelet 能够探知到此类故障。然而，在 Pod 对象遭到意外删除，或者工作节点自身发生故障时，又该如何处理呢？

kubelet 是 kubernetes 集群节点代理程序，它在每个工作节点上都运行着一个实例。因而，集群中的某工作节点发生故障时，其 kubelet 也必将不再可用，于是，节点上的 Pod 资源的健康状态将无从得到保证，也无法再由 kubelet 重启。此种场景重的 Pod 存活性一般由工作节点之外的 Pod 控制器来保证。事实上，遭意外删除的 Pod 资源的恢复也依赖于其控制器。

Pod 控制器由 master 的 kube-controller-manager 组件提供，常见的此类控制器有 Replication Controller、ReplicaSet、Deployment、DaemonSet、StatefulSet、Job 和 CronJob 等，它们分别以不同的方式管理 Pod 资源对象。实践中，对 Pod 对象的管理通常都是由某种控制器的特定对象来实现的，包括其创建、删除即重新调度等操作。本章将逐一讲解常用的 Pod 控制器资源。

## 5.1 关于 Pod 控制器

我们可以把 API Server 类比成一个存储对象的数据库系统，它向客户端提供API，并负责存储由用户创建的各种资源对象，至于各对象的当前状态如何才能符合用户期望的状态，则需要交由另一类称为控制器的组件来负责完成。Kubernetes 提供了众多的控制器来管理各种类型的资源，如 Node Lifecycle Controller、Namespace Controller、Service Controller 和 Deployment Controller 等，它们的功用几乎可以做到见名知义。创建完成后，每一个控制器对象都可以通过内部的和解循环（reconciliation loop），不间断地监控着由其负责的所有资源并确保其处于或不断地逼近用户定义的目标状态。

尽管能够由 kubelet 为其提供自愈能力，但在节点宕机时，自主式 Pod 对象的重建式自愈机制则需要由 Pod 控制器对象负责提供，并且由它来负责实现生命周期中的各类自动管理行为，如创建及删除等。

## 5.1.1 Pod 控制器概述

master 的各组件中，API Server 仅负责将资源存储于 etcd 中，并将其变动通知给各相关的客户端程序，如 kubelet、kube-scheduler、kube-proxy 和 kube-controller-manager 等，kube-scheduler 监控到处于未绑定状态的 Pod 对象出现时遂启动调度器为其挑选适配的工作节点，然而，Kubernetes 的核心功能之一还在于要确保各资源对象的当前状态（status）已匹配用户期望的状态（spec），使当前状态不断地向期望状态“和解”（reconciliation）来完成容器应用管理，而这些则是 kube-controller-manager 的任务。kube-controller-manager 是一个独立的单体守护进程，然而它包含了众多功能不同的控制器类型分别用于各类和解任务，如图 5-1 所示。

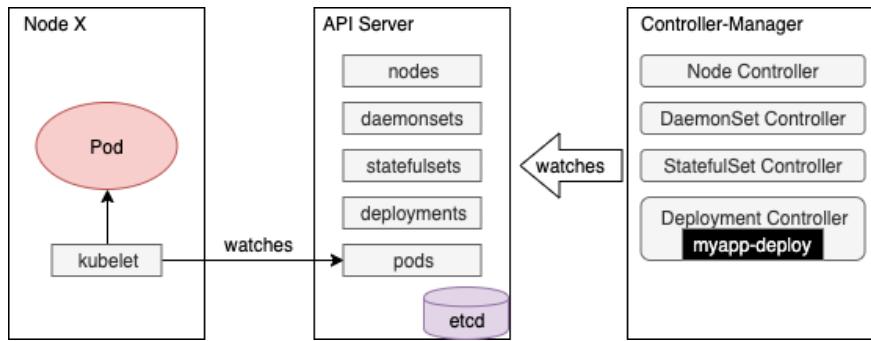


图 1.7.1.1 - kube-controller-manager 及其控制器

kubernetes 可用的控制器有 attachdetach、bootstrapsigner、clusterrole-aggregation、cronjob、csrapproving、csrcleaner、daemonset、deployment、disruption、endpoint、garbagecollector、horizontalpodautoscaling、job、namespace、node、persistentvolume-expander、podgc、pvc-protection、replicaset、replication-controller、resourcequota、route、service、serviceaccount、serviceaccount-token、statefulset、tokencleaner 和 ttl 等数十种。

创建为具体的控制器对象之后，每个控制器均通过 API Server 提供的接口持续监控相关资源对象的当前状态，并在因故障、更新或其他原因导致系统状态发生变化时，尝试让资源的当前状态向期望状态迁移和逼近。简单来说，每个控制器对象运行一个和解循环负责状态和解，并将目标资源对象的当前状态写入到其 status 字段中。控制器的“和解”循环如图 5-2 所示。

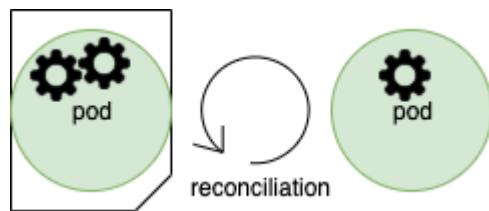


图 1.7.1.1 - 控制器的“和解”循环

List-Watch 是 Kubernetes 实现的核心机制之一，在资源对象的状态发生变动时，由 API Server 负责写入 etcd 并通过水平触发（level-triggered）机制主动通知给相关的客户端程序以确保其不会错过任何一个事件。控制器通过 API Server 的 Watch 接口实时监控目标资源对象的变动并执行和解操作，但并不会与其他控制器进行任何交互，设置彼此之间根本就意识不到对方的存在。

工作负载（workload）一类的控制器资源类型包括 ReplicationController、ReplicaSet、Deployment、DaemonSet、StatefulSet、Job 和 CronJob 等，它们分别代表了一种类型的 Pod 控制器资源，各类型的功用在 3.1.1 节中已经给出过说明。本章后面的篇幅主要介绍各控制器的特性及其应用，不过 StatefulSet 控制器依赖于存储卷资源，因此它将单独在存储卷之后的章节中给予介绍。

## 5.1.2 控制器与 Pod 对象

Pod 控制器资源通过持续性地监控集群中运行着的 Pod 资源对象来确保受其管控的资源严格符合用户期望的状态，例如资源副本的数量要精确符合期望等。通常，一个 Pod 控制器资源至少应该包含三个基本的组成部分。

- 标签选择器：匹配并关联 Pod 资源对象，并据此完成受其管控的 Pod 资源计数。
- 期望的副本数：期望在集群中精确运行着的 Pod 资源的对象数量。
- Pod 模版：用于新建 Pod 资源对象的 Pod 模版资源。

DaemonSet 用于确保集群中的每个工作节点或符合条件的每个节点上都运行着一个 Pod 副本，而不是精确的数量值，因此不具有上面组成部分中的第二项。

例如，一个如图 5-3 所示的 Deployment 控制器资源使用的标签选择器为“role=be-eshop”，它期望相关的 Pod 资源副本数量精确为 3 个，少于此数量的缺失部分将由控制器通过 Pod 模版予以创建，而多出的副本也将由控制器负责终止及删除。

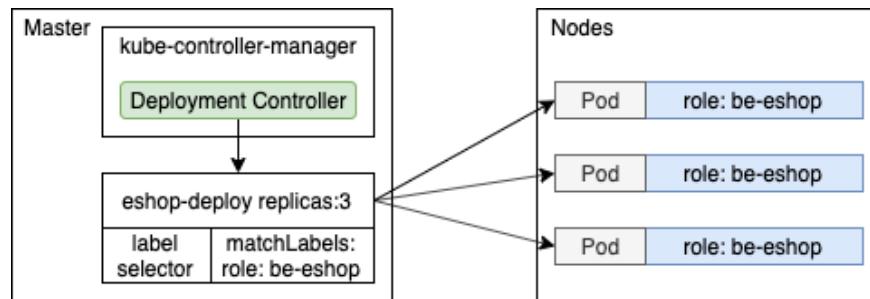


图 1.7.1.2 - Deployment 控制器示例

### 5.1.3 Pod 模版资源

PodTemplate 是 Kubernetes API 的常用资源类型，常用于为控制器指定自动创建 Pod 资源对象时所需的配置信息。因为要内嵌于控制器中使用，所以 Pod 模版的配置信息中不需要 apiServer 和 kind 字段，但除此之外的其他内容与定义自主式 Pod 对象所支持的字段几乎完全相同，这包括 metadata 和 spec 及其内嵌的其他各个字段。Pod 控制器类资源的 spec 字段通常都要内嵌 replicas、selector 和 template 字段，其中 template 即为 Pod 模版的定义。下面是一个定义在 ReplicaSet 资源中的模版资源示例：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 2
  selector:
    matchLabels:
      app: rs-demo
  template:
    metadata:
      labels:
        app: rs-demo
    spec:
      containers:
        - name: myapp
          image: ikubernetes/myapp:v1
        ports:
          - name: http
            containerPort: 80
```

如上示例中，spec.template 字段在定义时仅给出了 metadata 和 spec 两个字段，它的使用方法与自主式 Pod 资源完全相同。后面讲到控制器的章节时会反复用到 Pod 模版资源。

## 5.2 ReplicaSet 控制器

Kubernetes 较早期的版本中仅有 ReplicationController 一种类型的 Pod 控制器，后来的版本中陆续引入了更多的控制器实现，这其中就包括用来取代 ReplicationController 的新一代实现 ReplicaSet。事实上，除了额外支持基本集合（set-based）的标签选择器，以及它的滚动更新（Rolling-Update）机制要基于更高级的控制器 Deployment 实现之外，目前的 ReplicaSet 的其余功能基本上与 ReplicationController 相同。考虑到 Kubernetes 强烈推荐使用 ReplicaSet 控制器，且表示 ReplicationController 不久后即将废弃，这里就重点介绍 ReplicaSet 控制器。

## 5.2.1 ReplicaSet 概述

ReplicaSet（简称 RS）是Pod控制器类型的一种实现，用于确保由其管控的 Pod 对象副本数在任意时间都能精确满足期望的数量。如图 5-4 所示，ReplicaSet 控制器资源启动后会查找集群中匹配其标签选择器的 Pod 资源对象，当前活动对象的数量与期望的数量不吻合时，多则删除，少则通过 Pod 模版创建以补足，等 Pod 资源副本数量符合期望值后即进入下一轮和解循环。

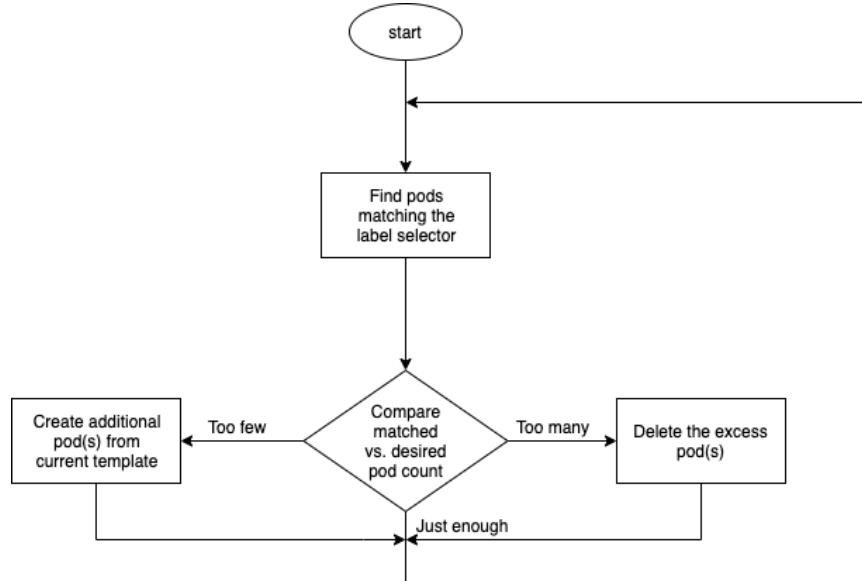


图 1.7.2.1 - ReplicaSet 的控制循环

ReplicaSet 的副本数量、标签选择器甚至是 Pod 模版都可以随时按需进行修改，不过仅改动期望的副本数量会对现存的 Pod 副本产生直接影响。修改标签选择器可能会使得现有的 Pod 副本的标签变得不再匹配，此时 ReplicaSet 控制器要做的不过是不再计入它们而已。另外，创建完成后，ReplicaSet 也不会再关注 Pod 对象中的实际内容，因此 Pod 模版的改动也只会对后来新建的 Pod 副本产生影响。

相比较与手动创建和管理Pod资源来说，ReplicaSet能够实现以下功能：

- 确保 Pod 资源对象的数量精确反映期望值：ReplicaSet 需要确保由其控制运行的 Pod 副本数量精确吻合配置定义的期望值，否则就会自动补充所缺或终止所余。
- 确保 Pod 健康运行：探测到由其管控的 Pod 对象因其所在的工作节点故障而不可用时，自动请求由调度器于其他工作节点创建缺失的 Pod 脚本。
- 弹性伸缩：业务规模因各种原因时常存在明显波动，在波峰或波谷期间，可以通过 ReplicaSet 控制器动态调整相关 Pod 资源对象数量。此外，在必要时还可以通过HPA(HroizontalPodAutoscaler)控制器实现 Pod 资源规模的自动伸缩。

## 5.2.2 创建 ReplicaSet

类似于 Pod 资源，创建 ReplicaSet 控制器对象同样可以使用 YAML 或 JSON 格式的清单文件定义其配置，而后使用相关的创建命令来完成资源创建。前面 5.1.3 节中给出的示例清单就是一个简单的 ReplicaSet 的定义。它也由 kind、apiVersion、metadata、spec 和 status 这五个一级字段组成，其中 status 为只读字段，因此需要在清单文件中配置的仅为前 4 个字段。它的 spec 字段一般嵌套使用以下几个属性字段。

- `replicas <integer>`：期望的 Pod 对象副本数。
- `selector <Object>`：当前控制器匹配 Pod 对象副本的标签选择器，支持 `matchLabels` 和 `matchExpressions` 两种匹配机制。
- `template <Object>`：用于补足 Pod 副本数量时使用的 Pod 模版资源。
- `minReadySeconds <integer>`：新建的 Pod 对象，在启动后的多长时间内如果其容器未发生崩溃等异常情况即被视为“就绪”；默认为0秒，表示一旦就绪性探测成功，即被视作可用。

将 5.1.3 节中的示例保存与资源清单文件中，例如 `rs-example.yaml`，而后即可使用如下命令将其创建：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f rs-example.yaml]
replicaset.apps/rs-example created
```

集群中当前没有标签为 “app: rs-demo” 的 Pod 资源存在，因此 `rs-example` 需要按照 `replicas` 字段定义创建它们，名称以其所属的控制器名称为前缀。这两个 Pod 资源目前都处于 `ContainerCreating` 状态，即处于容器创建过程中，待创建过程完成后，其状态即转为 `Running`，Pod 也将转变为 “READY”：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l app=rs-demo]
NAME          READY   STATUS        RESTARTS   AGE
rs-example-97snc  0/1    ContainerCreating   0          2s
rs-example-vrz2n  0/1    ContainerCreating   0          2s
```

接下来可以使用 “`kubectl get replicaset`” 命令查看 ReplicaSet 控制器资源的相关状态。下面的命令结果显示出它已经根据清单中配置的 Pod 模版创建了 2 个 Pod 资源，不过这时他们尚未创建完成，因此仍为“READY”：

```
**[terminal]
**[delimiter $ ]**[command kubectl get replicaset rs-example -o wide]
NAME      DESIRED   CURRENT   READY   AGE   CONTAINERS   IMAGES
rs-example  2         2         2       4s    myapp       ikubernetes/myapp:\
```

经由控制器创建与用户自主创建的 Pod 对象的功能并无二致，但其自动和解的功能能在很大程度上能为用户省去不少的管理精力，这也是使用 Kubernetes 系统之上的应用程序变得拥有自愈能力的主要保障。

## 5.2.3 ReplicaSet 管控下的 Pod 对象

5.2.2 节中创建的 rc-example 通过标签选择器将拥有 “app=rs-demo” 标签的 Pod 资源收归与麾下，并确保其数量精确符合所期望的数目，使用标签选择器显示出的 Pod 资源列表也能验证这一点。然而，实际中存在着不少可能导致 Pod 对象数目与期望值不符合的可能性，如 Pod 对象的意外删除、Pod 对象标签的变动（已有的 Pod 资源变得不匹配控制器的标签选择器，或者外部的 Pod 资源标签变得匹配到了控制器的标签选择器）、控制器的标签选择器变动，甚至是工作节点故障等。ReplicaSet 控制器的和解循环过程能够实时监控到这类异常，并及时启动和解操作。

### 1. 缺少 Pod 副本

任何原因导致的相关 Pod 对象丢失，都会由 ReplicaSet 控制器自动补足。例如，手动删除上面列出的一个 Pod 对象，命令如下：

```
**[terminal]
**[delimiter $ ]**[command kubectl delete pods rs-example-cwvpn]
pod "rs-example-cwvpn" deleted
```

再次列出相关的 Pod 资源，可以看到 rs-example 控制器启动了删除多余 Pod 的操作，pod-example 正处于终止过程中：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l app=rs-demo]
NAME        READY   STATUS            RESTARTS   AGE
rs-example-cwvpn   1/1    Terminating      0          13m
rs-example-s47z9   1/1    Running           0          99s
rs-example-tc7d8   0/1    ContainerCreating  0          2s
```

另外，强制修改隶属于控制器 rs-example 的某 Pod 资源（匹配于标签控制器）的标签，会导致它不再被控制器作为副本计数，这也将触发控制器的 Pod 对象副本缺失补足机制。例如，将 rs-example-tc7d8 的标签 app 的值重置为空：

```
**[terminal]
**[delimiter $ ]**[command kubectl label pods rs-example-tc7d8 app= --overwrite
pod/rs-example-tc7d8 labeled
```

列出 rs-example 相关的 Pod 对象的信息，发现 rs-example-tc7d8 已经消失不见了，并且正在创建新的对象副本。

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l app=rs-demo]
NAME        READY   STATUS            RESTARTS   AGE
rs-example-42cw7  0/1    ContainerCreating   0          1s
rs-example-s47z9  1/1    Running           0          12m
```

由此可见，修改 Pod 资源的标签即可将其从控制器的管控之下移除，当然，修改后的标签如果又能被其他控制器资源的标签选择器命中，则此时它又成了隶属于另一控制器的副本。如果修改其标签后的 Pod 对象不再隶属于任何控制器，那么它就将成为自主式 Pod，与此前手动直接创建的 Pod 对象的特征相同，即误删除或所在工作节点故障都会造成其永久性的消失。

## 2. 多处 Pod 副本

一旦被标签选择器匹配到的 Pod 资源数量因任何原因超出期望值，多余的部分都将被控制器自动删除。例如，为 pod-example 手动为其添加 “app: rs-demo” 标签：

```
**[terminal]
**[delimiter $ ]**[command kubectl label pods pod-example app=rs-demo]
pod/pod-example labeled
```

再次列出相关的 Pod 资源，可以看到 rs-example 控制器启动了删除多余 Pod 的操作，rs-example-42cw7 正处于终止过程中：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pod -l app=rs-demo]
NAME        READY   STATUS            RESTARTS   AGE
pod-example  1/1    Running           1          6d23h
rs-example-42cw7  0/1    Terminating   0          10m
rs-example-s47z9  1/1    Running           0          23m
```

这就意味着，任何自主式的或本隶属于其他控制器的 Pod 资源其标签变动的结果一旦匹配到了其他的副本数足额的控制器，就会导致这类资源被删除。

## 查看 Pod 资源变动的相关事件

“kubectl describe replicaset” 命令可打印出 ReplicaSet 控制器的详细状态。从下面命令结果中 Events 一段也可以看出，rs-example 执行了 Pod 资源的创建和删除操作，为的就是确保其数量的精确性。

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl describe replicaset/rs-example]
Name:          rs-example
Namespace:     default
Selector:      app=rs-demo
Labels:        <none>
Annotations:   kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"apps/v1","kind":"ReplicaSet","metadata":{"annotations":{},"labels":{"app": "rs-demo"}, "name": "rs-example", "namespace": "default"}, "spec": {"replicas": 2, "selector": {"matchLabels": {"app": "rs-demo"}}, "template": {"labels": {"app": "rs-demo"}, "spec": {"containers": [{"image": "ikubernetes/myapp:v1", "name": "myapp", "port": 80, "hostPort": 0}], "volumes": []}}}, "status": {"replicas": 2, "currentReplicas": 2, "desiredReplicas": 2, "conditions": [{"status": "True", "type": "Available"}, {"status": "True", "type": "Ready"}, {"status": "True", "type": "PodScheduled"}]}, "events": [{"type": "Normal", "reason": "SuccessfulCreate", "age": "42m", "from": "replicaset-controller", "message": "Created pod: rs-example-645d5"}, {"type": "Normal", "reason": "SuccessfulCreate", "age": "42m", "from": "replicaset-controller", "message": "Created pod: rs-example-645d5"}, {"type": "Normal", "reason": "SuccessfulCreate", "age": "30m", "from": "replicaset-controller", "message": "Created pod: rs-example-645d5"}, {"type": "Normal", "reason": "SuccessfulCreate", "age": "28m", "from": "replicaset-controller", "message": "Created pod: rs-example-645d5"}]}
```

事实上，ReplicaSet 控制器能对 Pod 对象数目的异常及时作出响应，是因为它向 API Server 注册监听（watch）了相关资源及其列表的变动信息，于是 API Server 会在变动发生时立即通知给相关的监听客户端。

而因节点自身故障而导致的 Pod 对象丢失，ReplicaSet 控制器一样会使用补足资源的方式进行处理，这里不再详细说明其过程。有兴趣的读者可通过直接关掉类似上面 Pod 对象运行所在的某一节点来检验其处理过程。

## 5.2.4 更新 ReplicaSet 控制器

ReplicaSet 控制器的核心组成部分是标签选择器、副本数量及 Pod 模板，但更新操作一般是围绕 replicas 和 template 两个字段值进行的，毕竟改变标签选择器的需求几乎不存在。改动 Pod 模板的定义对已经创建完成的活动对象无效，但在用户逐个手动关闭其旧版本的 Pod 资源后就能以新代旧，实现控制器下应用版本的滚动升级。另外，修改副本的数量也就意味着应用规模的扩展（提升期望的副本数量）或收缩（降低期望的副本数量）。这两种操作也是系统运维人员日常维护工作的重要组成部分。

### 1. 更新 Pod 模板：升级应用

ReplicaSet 控制器的 Pod 模板可随时按需修改，但它仅影响这之后由其新建的 Pod 对象，对已有的副本不会产生作用。大多数情况下，用户需要改变的通常是模板中的容器镜像文件及其相关的配置以实现应用的版本升级。下面的示例清单文件片段 (rs-example-v2.yaml) 中的内容与之前版本 (rs-example.yaml) 的唯一不同之处也仅在于镜像文件的改动：

```
containers:
- name: nginx
  image: ikubernetes/myapp:v2
  ports:
- name: http
  containerPort: 80
```

对新版本的清单文件执行 “kubectl apply” 或 “kubectl replace” 命令即可完成 rs-example 控制器资源的修改操作：

```
**[terminal]
**[delimiter $ ]**[command kubectl replace -f rs-example-v2.yaml]
replicaset.apps/rs-example replaced
```

不过，控制器 rs-example 管控的现存 Pod 对象使用的仍然是原来版本中定义的镜像：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l app=rs-demo -o custom-columns=Name:pod-name,Image:image]
Name          Image
pod-example   ikubernetes/myapp:v1
rs-example-s47z9  ikubernetes/myapp:v1
```

此时，手动删除控制器现有的 Pod 对象（或修改与其匹配的控制器标签选择器的标签），并由控制器基于新的 Pod 模板自动创建出足额的 Pod 副本，即可完成一次应用的升级。新旧更替的过程支持如下两类操作方式。

### A.1.1 部署目标

- 一次性删除控制器相关的所有 Pod 副本或更改相关的标签：剧烈更替，可能会导致 Pod 中的应用短时间不可访问（如图 5-5 所示）；生产实践中，这种做法不可取。

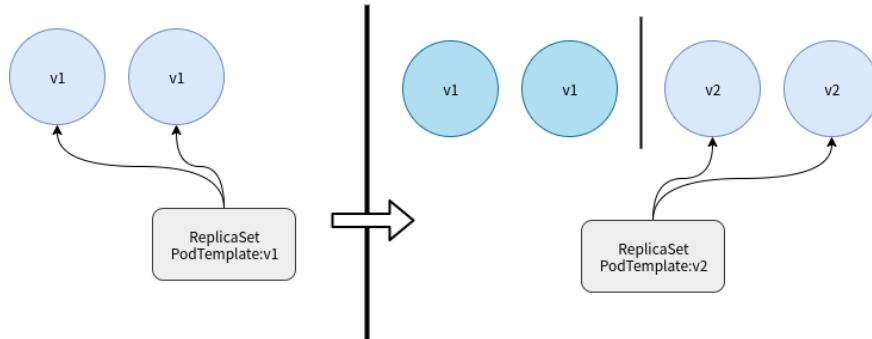


图 1.7.2.4 - 直接更替所有 Pod 资源

- 分批次删除旧有的 Pod 副本或更改其标签（待控制器补足后再删除另一批）：滚动更替，更替期间新旧版本共存（如图 5-6 所示）。

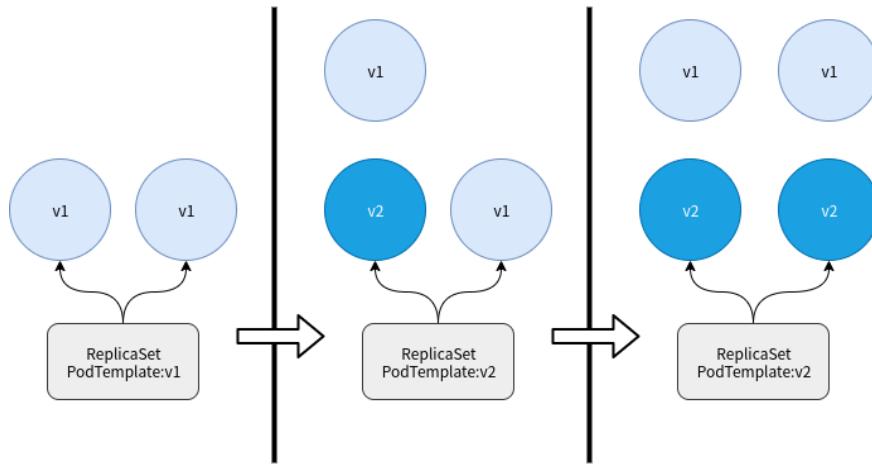


图 1.7.2.4 - 滚动更替 Pod 资源

例如，这里采用第一种方式进行操作，一次性删除 rs-example 相关的所有 Pod 副本：

```
**[terminal]
**[delimiter $ ]**[command kubectl delete pods -l app=rs-demo]
pod "pod-example" deleted
pod "rs-example-s47z9" deleted
```

再次列出 rc-example 控制器相关的 Pod 及其容器镜像版本时可以发现，使用新版本镜像的 Pod 已经创建完成：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l app=rs-demo -o custom-columns=Name:metadata.name,Image:image --selector app=rs-demo]
  Name           Image
rs-example-rmmtp  ikubernetes/myapp:v2
rs-example-smmbv  ikubernetes/myapp:v2
```

必要时，用户还可以将 Pod 模板改回就的版本进行应用的“降级”或“回滚”，它的操作过程与上述过程基本类似。事实上，修改 Pod 模板时，不仅仅能替换镜像文件的版本，甚至还可以将其替换成其他正在运行着的、完全不同应用的镜像，只不过此类需求并不多见。若同时改动的还有 Pod 模板中的其他字段，那么在新旧更替的过程中，它们也将随之被应用。

以上操作只为说明应用部署的方式，实际使用时还需要更为完善的机制。即便是仅执行了一到多次删除操作，手动执行更替操作也并非一项轻松的任务，幸运的是，更高级别的 Pod 控制器 Deployment 能够自动实现更完善的滚动更新和回滚，并为用户提供自定义更新策略的接口。而且，经过精心组织的更新操作还可以实现诸如蓝绿部署（Blue/Green Deployment）、金丝雀部署（Canary Deployment）和灰度部署等，这些内容将在后面章节中详细展开说明。

## 2. 扩容和缩容

改动 ReplicaSet 控制器对象配置中期望的 Pod 副本数量（replicas 字段）会由控制器实时做出响应，从而实现应用规模的水平伸缩。replicas 的修改及应用方式同 Pod 模板，不过，kubectl 还提供了一个专用的子命令 scale 用于实现应用规模的伸缩，它支持从资源清单文件中获取新的目标副本数量，也可以直接在命令行通过“--replicas”选项进行读取，例如将 rs-example 控制器的 Pod 副本数量提升至5个：

```
**[terminal]
**[delimiter $ ]**[command kubectl scale replicaset rs-example --replicas=5]
replicaset.extensions/rs-example scaled
```

由下面显示的 rs-example 资源的状态可以看出，将其 Pod 资源副本数量扩展至5个的操作已经成功完成：

```
**[terminal]
**[delimiter $ ]**[command kubectl get replicaset rs-example]
  NAME      DESIRED   CURRENT   READY   AGE
rs-example  5          5          5     20h
```

收缩规模的方式与扩展相同，只需要明确指定目标副本数量即可。例如：

## A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl scale replicaset rs-example --replicas=3]
replicaset.extensions/rs-example scaled
```

另外，kubectl scale 命令还支持现有 Pod 副本数量符合指定的值时才执行扩展操作，这仅需要为命令使用“--current-replicas”选项即可。例如，下面的命令表示如果 rs-example 目前的 Pod 副本数量为 2，就将其扩展至 4 个：

```
**[terminal]
**[delimiter $ ]**[command kubectl scale replicaset rs-example --current-replicas=2 --replicas=4]
error: Expected replicas to be 2, was 3
```

但由于 rs-example 控制器现存的副本数量是 3 个，因此上面的扩展操作未执行并返回了错误提示。

如果 ReplicaSet 控制器管控的是有状态的应用，例如主从架构的 Redis 集群，那么上述这些升级、降级、扩展和收缩的操作都需要精心编排和参与才能进行，不过，这也在一定程度上降低了 Kubernetes 容器编排的价值和意义。好在，它提供了 StatefulSet 资源来应对这种需求，因此，ReplicaSet 通常仅用于管理无状态的应用，如 HTTP 服务程序等。

## 5.2.5 删除 ReplicaSet 控制器资源

使用 `kubectl delete` 命令删除 ReplicaSet 对象时默认会一并删除其管控的各 Pod 对象。有时，考虑到这些 Pod 资源未必由其创建，或者即便由其创建也并非其自身的组成部分，故而可以为命令使用 “`--cascade=false`” 选项，取消级联，删除相关的 Pod 对象，这在 Pod 资源后续可能会再次用到时尤为有用。例如，删除 rs 控制器 rs-example：

```
**[terminal]
**[delimiter $ ]**[command kubectl replicsets rs-example --cascade=false]
replicaset.extensions "rs-example" deleted
```

删除操作完成后，此前由 rs-example 控制器管控的各 Pod 对象仍处于活动状态，但他们变成了自主式 Pod 资源，用户需要自行组织和维护好它们。

后续讲到的各 Pod 控制器的删除方式都与 ReplicaSet 类似，这里就不再分别进行说明了。

尽管 ReplicaSet 控制器功能强大，但在实践中，它却并非是用户直接使用的控制器，而是要由比其更高一级抽象的 Deployment 控制器对象来调用。

## 5.3 Deployment 控制器

Deployment（简写为 deploy）是 Kubernetes 控制器的又一种实现，它构建于 ReplicaSet 控制器之上，可为 Pod 和 ReplicaSet 资源提供声明式更新。相比较而言，Pod 和 ReplicaSet 是较低级别的资源，它们很少被直接使用。Deployment、ReplicaSet 和 Pod 的关系如图 5-7 所示：

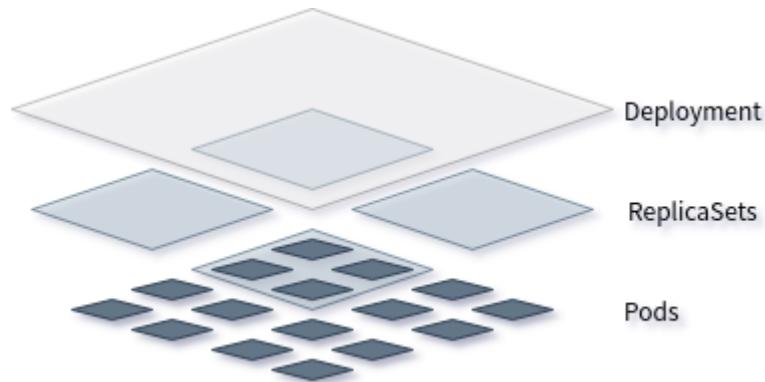


图 1.7.3 - Deployment、ReplicaSets 和 Pods

Deployment 控制器资源的主要职责同样是为了保证 Pod 资源的健康运行，其大部分功能均可通过调用 ReplicaSet 控制器来实现，同时还增添了部分特性。

- 事件和状态查看：必要时可以查看 Deployment 对象升级的详细进度和状态。
- 回滚：升级操作完成后发现问题时，支持使用回滚机制将应用返回到前一个或由用户指定的历史记录中的版本上。
- 版本记录：对 Deployment 对象的每一次操作都予以保存，以供后续可能执行的回滚操作使用。
- 暂停和启动：对于每一次升级，都能随时暂停和启动。
- 多种自动更新方案：一是 Recreate，及重建更新机制，全面停止、删除旧有的 Pod 后用新版本替代；另一个是 RollingUpdate，即滚动升级机制，逐步替换旧有的 Pod 至新的版本。

## 5.3.1 创建 Deployment

Deployment 是标准 Kubernetes API 资源，它构建于 ReplicaSet 资源之上，于是其 spec 字段中嵌套使用的字段包含了 ReplicaSet 控制器支持 replicas、selector、template 和 minReadySeconds，它也正是利用这些信息完成了其二级资源 ReplicaSet 对象的创建。下面是一个 Deployment 控制器资源的配置清单示例：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: ikubernetes/myapp:v1
          ports:
            - name: http
              containerPort: 80
```

上面的内容显示出，除了控制器类型和名称之外，它与前面 ReplicaSet 控制器示例中的内容几乎没有什么不同。下面在集群中创建以了解它的工作方式：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f myapp-deploy.yaml --record]
deployment.apps/myapp-deploy created
```

“kubectl get deployment” 命令可以列出创建的 Deployment 对象 myapp-deploy 及其相关的信息。下面显示的字段中，UP-TO-DATE 表示已经达到期望状态的 Pod 副本数量，AVAILABLE 则表示当前处于可用状态的应用程序的数量：

```
**[terminal]
**[delimiter $ ]**[command kubectl get deployment]
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
myapp-deploy  3/3     3           3          4m15s
```

Deployment 控制器会自动创建相关的 ReplicaSet 控制器资源，并以 “[DEPLOYMENT-NAME]-[POD-TEMPLATE-HASH-VALUE]” 格式为其命令，其中的 hash 值由 Deployment 控制器自动生成。由 Deployment 创建的 ReplicaSet 对

### A.1.1 部署目标

象会自动使用相同的标签选择器，因此，可使用类似如下命令查看其相关的信息：

```
**[terminal]
**[delimiter $ ]**[command kubectl get replicaset -l app=myapp]
NAME          DESIRED   CURRENT   READY   AGE
myapp-deploy-5fdb5f69f    3         3        3      9m48s
```

相关的 Pod 对象的信息可以用相似的命令进行获取。下面的命令结果中，Pod 对象的名称遵循 ReplicaSet 控制器的命令格式，它以 ReplicaSet 控制器的名称为前缀，后跟 5 位随机字符：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l app=myapp]
NAME          READY   STATUS    RESTARTS   AGE
myapp-deploy-5fdb5f69f-df4qn  1/1     Running   0          12m
myapp-deploy-5fdb5f69f-gnbcz  1/1     Running   0          12m
myapp-deploy-5fdb5f69f-p2s6b  1/1     Running   0          12m
```

由此验证了 Deployment 借助于 ReplicaSet 管理 Pod 资源的机制，于是可以得知，其大部分管理操作与 ReplicaSet 相同。不过，Deployment 也有 ReplicaSet 所不具有的部分高级功能，这其中最著名的当属其自动滚动更新的机制。

## 5.3.2 更新策略

如前所述，ReplicaSet 控制器的应用更新需要手动分成多步并以特定的次序进行，过程繁琐且容易出错，而Deployment却只需要由用户指定在Pod模板中要改动的内容，例如容器镜像文件的版本，余下的步骤可交由其自动完成。同样，更新应用程序的规模也需要修改期望的副本数量，余下的事情交给Deployment控制器即可。

Deployment控制器的详细信息中包含了其更新策略的相关配置信息，如 myapp-deploy 控制器资源 “kubectl describe” 命令中输出的 `StrategyType`、`RollingUpdateStrategy` 字段等。

```
**[terminal]
**[delimiter $ ]**[command kubectl describe deployments myapp-deploy]
Name:           myapp-deploy
Namespace:      default
CreationTimestamp:   Fri, 11 Sep 2020 16:22:37 +0800
Labels:          <none>
Annotations:    deployment.kubernetes.io/revision: 1
Selector:        app=myapp
Replicas:       3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=myapp
  Containers:
    myapp:
      Image:      ikubernetes/myapp:v1
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type        Status  Reason
    ----        ----   -----
    Available   True    MinimumReplicasAvailable
    Progressing True    NewReplicaSetAvailable
  OldReplicaSets: <none>
  NewReplicaSet:  myapp-deploy-5cbd66595b (3/3 replicas created)
Events:
  Type        Reason          Age      From            Message
  ----        ----   -----   ----
  Normal  ScalingReplicaSet  2m26s  deployment-controller  Scaled up replica set
```

Deployment 控制器支持两种更新策略：滚动更新（rolling update）和重新创建（recreate）。默认为滚动更新。重新创建更新类似于前文中ReplicaSet的第一种更新方式，即首先删除现有的Pod对象，而后由控制器基于新模板重新创建出新版本资源对象。通常，只应该在应用的新旧版本不兼容（如依赖的后端数据库的 schema不同且无法兼容）时才会使用recreate策略，因为它会导致应用替换期间暂时不可用，好处在于它不存在中间状态，用户访问到的要么是应用的新版本，要么是旧版本。

滚动更新是默认的更新策略，它在删除一部分就版本Pod资源的同时，补充创建一部分新版本的Pod对象进行应用升级，其优势是升级期间，容器中应用提供的服务不会中断，但要求应用程序能够应对新旧版本同时工作的情形，例如新旧版本兼容同一个数据库方案等。不过，更新操作期间，不同客户端得到的响应内容可能会来自不同版本的应用。

Deployment控制器的滚动更新操作并非在同一个ReplicaSet控制器对象下删除并创建Pod资源，而是将它们分置于两个不同的控制器之下：旧控制器的Pod对象不断减少的同时，新控制器的Pod对象数量不断增加，直到旧控制器的Pod对象，而新控制的副本数量变得完全符合期望值为止，如图5-8所示。

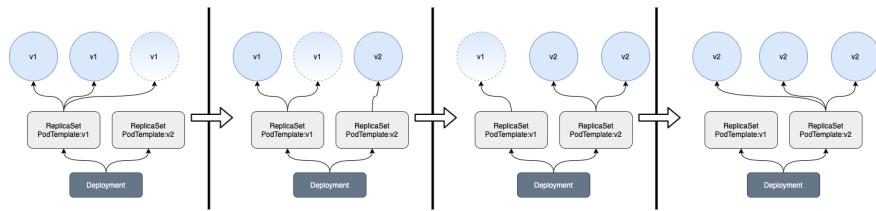


图 1.7.3.2 - Deployment 的滚动更新

滚动更新时，应用升级期间还要确保可用的Pod对象数量不低于某阈值以确保可以持续处理客户端的服务请求，变动的方式和Pod对象的数量分为将通过 `spec.strategy.rollingUpdate.maxSurge` 和 `spec.strategy.rollingUpdate.maxUnavailable` 两个属性协同进行定义，它们的功用如图5-9所示：

- `maxSurge`：指定升级期间存在的总Pod对象数量最多可超出期望值的个数，其值可以是0或正整数，也可以是一个期望值的百分比；例如，如果期望值为3，当前的属性值为1，则表示Pod对象的总数不能超过4个。
- `maxUnavailable`：升级期间正常可用的Pod副本数（包括旧版本）最多不能地域期望值的个数，其值可以是0或正整数，也可以是一个期望值的百分比；默认值为1，该值意味着如果期望值是3，则升级期间至少要有两个Pod对象处于正常提供服务的状态。

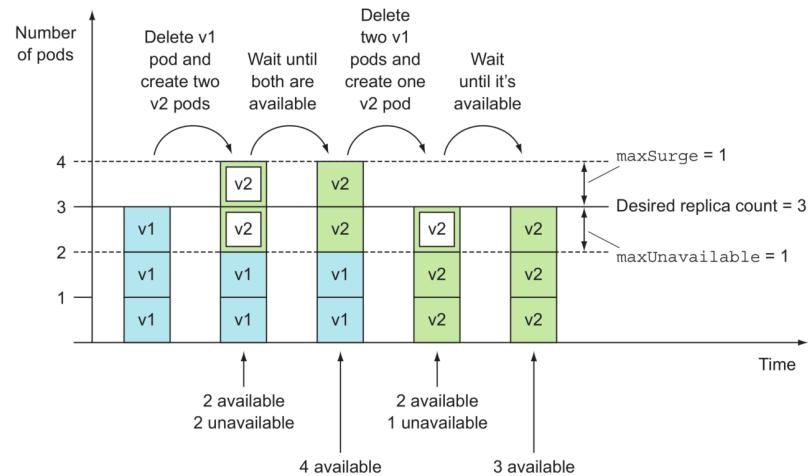


图 1.7.3.2 - `maxSurge` 和 `maxUnavailable` 的作用方式

`maxSurge` 和 `maxUnavailable` 属性的值不可同时为 0，否则 Pod 对象的副本数量在符合用户期望的数量后无法做出合理变动以进行滚动更新操作。

配置时，用户还可以使用 Deployment 控制器的 `spec.minReadySeconds` 属性来控制应用升级的速度。新旧更替过程中，新创建的 Pod 对象一旦成功响应就绪探测即被视作可用，而后即可立即开始下一轮的替换操作。而 `spec.minReadySeconds` 能够定义在新的 Pod 对象创建后至少要等待多久才将其视作就绪，在此期间，更新操作会被阻塞。因此，它可以用来让 Kubernetes 在每次创建出 Pod 资源后都要登上一段时长后再开始下一轮的更替，这个时间长度的理想值是等到 Pod 对象中的应用已经可以接受并处理请求流量。事实上，一个精心设计的等待时长和就绪性试探能让 Kubernetes 系统规避一部分因程序 Bug 而导致的升级故障。

Deployment 控制器也支持用户保留其滚动更新历史中的旧 ReplicaSet 对象版本，如图 5-10 所示，这赋予了控制器进行应用回滚的能力：用户可按需回滚到指定的历史版本。控制器可保存的历史版本数量有“`spec.revisionHistoryLimit`”属性进行定义。当然，也只有保存于 revision 历史中的 ReplicaSet 版本可用于回滚，因此，用户要习惯性地在更新操作时指定保留旧版本。

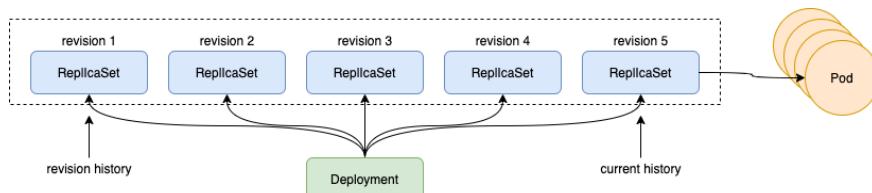


图 1.7.3.2 - `Deployment` 的版本历史记录

为了保存版本升级的历史，需要在创建 Deployment 对象时于命令中使用“`--record`”选项。

### A.1.1 部署目标

尽管滚动更新以节约系统资源著称，但它也存在一些劣势。直接改动现有环境，会使系统引入不确定性风险，而且升级过程中出现问题后，执行回滚操作也会较为缓慢。有鉴于此，金丝雀部署可能使较为理想的实现方式，当然，如果不考虑系统资源的可用性，那么传统的蓝绿部署也是不错的选择。

### 5.3.3 升级Deployment

修改Pod模版相关的配置参数便能够完成Deployment控制器资源的更新。由于是声明式配置，因此对Deployment控制器资源的修改尤其适合使用apply和patch命令来进行；当然，如果仅是修改容器镜像，“set image”命令更为易用。

接下来通过更新此前创建的Deployment控制器deploy-example来了解更新操作过程的执行细节，为了使得升级过程更易于观察，这里使用“kubectl patch”命令为其spec.minReadySeconds字段定一个等待时长，例如5s：

```
**[terminal]
**[delimiter $ ]**[command kubectl patch deployment myapp-deploy -p '{"spec": +
deployment.apps/myapp-deploy patched' +
```

patch命令的补丁形式为JSON格式，以-p指定选项，上面命令中的{"spec": {"minReadySeconds": 5}}表示设置spec.minReadySeconds属性的值。若要改变myapp-deploy中myapp容器的镜像，也可以使用patch命令，如{"spec": {"containers": [{"name": "myapp", "image": "ikubernetes/myapp:v2"}]}，不过，修改容器镜像有更为简单的专用命令“set image”。

修改Deployment控制器的minReadySeconds、replicas和strategy等字段的值并不会触发Pod资源的更新操作，因为他们不属于模板的内嵌字段，对现存的Pod对象不产生任何影响。

接着，使用“ikubernetes/myapp:v2”镜像文件修改Pod模板中的myapp容器，启动Deployment控制器的滚动更新过程：

```
**[terminal]
**[delimiter $ ]**[command kubectl set image deployments myapp-deploy myapp=iku
deployment.apps/myapp-deploy image updated' +
```

“kubectl rollout status”命令可用于打印滚动更新过程中的状态信息：

```
**[terminal]
**[delimiter $ ]**[command kubectl rollout status deployments myapp-deploy]
Waiting for deployment "myapp-deploy" rollout to finish: 3 out of 4 new replicas
Waiting for deployment "myapp-deploy" rollout to finish: 3 out of 4 new replicas
Waiting for deployment "myapp-deploy" rollout to finish: 3 out of 4 new replicas
Waiting for deployment "myapp-deploy" rollout to finish: 1 old replicas are per
Waiting for deployment "myapp-deploy" rollout to finish: 1 old replicas are per
Waiting for deployment "myapp-deploy" rollout to finish: 1 old replicas are per
Waiting for deployment "myapp-deploy" rollout to finish: 1 old replicas are per
Waiting for deployment "myapp-deploy" rollout to finish: 3 of 4 updated replicas
deployment "myapp-deploy" successfully rolled out
```

### A.1.1 部署目标

另外，还可以使用“kubectl get deployments --watch”命令监控其更新过程中Pod对象的变动过程：

```
**[terminal]
**[delimiter $ ]**[command kubectl get replicaset -l app=myapp]
NAME          DESIRED   CURRENT   READY   AGE
myapp-deploy-5cbd66595b   2         2         2      45h
myapp-deploy-6685c8c7fc   3         3         1      5m10s
```

myapp-deploy控制器管控的Pod资源对象也将随之更新为以新版本ReplicaSet名称“myapp-deploy-6685c8c7fc”为前缀的Pod副本，命令结果如下所示：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l app=myapp]
NAME          READY   STATUS    RESTARTS   AGE
myapp-deploy-6685c8c7fc-b7c7n   1/1     Running   0          119s
myapp-deploy-6685c8c7fc-ck8l4   1/1     Running   0          2m15s
myapp-deploy-6685c8c7fc-dxv5g   1/1     Running   0          2m7s
myapp-deploy-6685c8c7fc-h4f7t   1/1     Running   0          2m15s
```

由于已经处于READY状态，因此上面命令列出的任一Pod资源均可正常向用户提供相关服务，例如，在集群内任一能使用kubectl的节点访问myapp-deploy-6685c8c7fc-dxv5g中的Web服务，命令如下：

```
**[terminal]
**[delimiter $ ]**[command curl $(kubectl get pods myapp-deploy-6685c8c7fc-b7c7n -o yaml | grep ip | awk '{print $2}')]
```

Hello MyApp | Version: v2 | <a href="hostname.html">Pod Name</a>

## 5.3.4 金丝雀发布

Deployment控制器还支持自定义控制更新过程中的滚动节奏，如“暂停”（pause）或“继续”（resume）更新操作，尤其是借助于前文讲到的maxSurge和maxUnavailable属性还能实现更为精巧的过程控制。比如，待第一批新的Pod资源创建完成后立即暂停更新过程，此时，仅存在一小部分新版本的应用，主体部分还是旧的版本。然后，在根据用户特征精心筛选出小部分用户的请求路由至新版本的Pod应用，并持续观察其能否稳定地按期望的方式运行。确定没有问题后再继续完成余下Pod资源的滚动更新，否则立即回滚更新操作。这便是所谓的金丝雀发布（Canary Release），如图5-11所示。

### 扩展知识：矿井中的金丝雀

17世纪，英国矿井工人发现，金丝雀对瓦斯这种气体十分敏感。空气中哪怕有极其微量的瓦斯气体，金丝雀

直接发布新应用版本的在线发布形式中，金丝雀发布是一种较为妥当的方式。不过，这里只涉及其部署操作的相关步骤，发布方式则通常依赖于具体的环境设置。接下来说明如何在Kubernetes上使用Deployment控制器实现金丝雀部署。

为了尽可能地降低对现有系统及其容量的影响，金丝雀发布过程通常建议采用“先添加、再删除，且可用Pod资源对象总数不低于期望值”的方式进行。首次添加的Pod对象数量取决于其接入的第一批请求的规则及单个Pod的承载能力，视具体需求而定，为了能够更简单的说明问题，接下来采用首批添加1个Pod资源的方式。将Deployment控制器的maxSurge属性的值设置为1，并将maxUnavailable属性的值设置为0：

```
**[terminal]
**[delimiter $ ]**[command kubectl patch deployment myapp-deploy -p '{"spec": +
deployment.apps/myapp-deploy patched']
```

接下来，启动myapp-deploy控制器的更新过程，在修改相应容器的镜像版本后立即暂停更新进度，它会在启动第一批新版本Pod对象的创建操作之后转为暂停状态。需要注意的是，这里之所以能够在第一批更新启动后就暂停，有赖此前为maxReadySeconds属性设置的时长，因此用户要在更新命令后的此时长指定的时间范围内启动暂停操作，其执行过程如图5-12所示。当然，对kubectl命令来说，也可以直接以“&&”符号在Shell中连接两个命令：

```
**[terminal]
**[delimiter $ ]**[command kubectl set image deployment myapp-deploy myapp=ikut
kubectl rollout pause deployments myapp-deploy]
deployment.apps/myapp-deploy image updated
deployment.apps/myapp-deploy paused
```

### A.1.1 部署目标

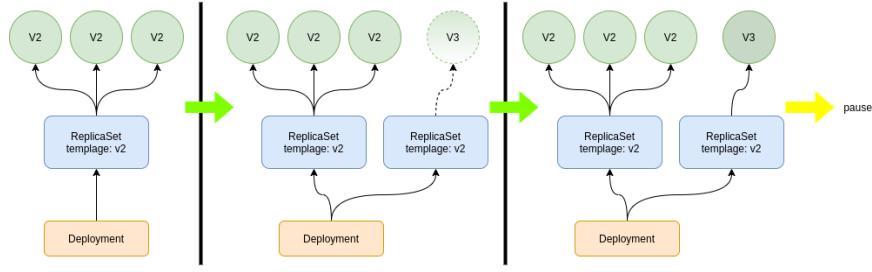


图 1.7.3.4 - 暂停Deployment滚动更新

通过其状态查看命令可以看到，在创建完一个新版本的Pod资源后滚动更新操作“暂停”：

```
**[terminal]
**[delimiter $ ]**[command kubectl rollout status deployment myapp-deploy]
Waiting for deployment "myapp-deploy" rollout to finish: 1 out of 3 new replicas
are available...
```

相关的Pod列表也可能显示旧版本的ReplicaSet的所有Pod副本仍在正常运行，新版本的ReplicaSet也包含一个Pod副本，但最多不超过期望值1个，myapp-deploy原有的期望值为3，因此总数不超过4个。此时，通过Service或Ingress资源及相关路由策略等设定，即可将一部分用户的流量引入到这些Pod之上进行发布验证。运行一段时间后，如果确认没有问题，即可使用“kubectl rollout resume”命令继续此前的滚动更新过程：

```
**[terminal]
**[delimiter $ ]**[command kubectl rollout resume deployments myapp-deploy]
deployment.apps/myapp-deploy resumed
```

“kubectl rollout status”命令监控到滚动更新过程完成后，即可通过myapp-deploy控制器及其ReplicaSet和Pod对象的相关信息来了解其结果状态。

然而，如果“金丝雀”遇险甚至遭遇不幸，那么回滚操作便成了接下来的当紧任务。

## 5.3.5 回滚Deployment控制器下的应用发布

若因各种原因的导致滚动更新无法正常进行，如镜像文件获取失败、“金丝雀”遇险等，则应该将应用回滚到之前的版本，或者回滚到由用户指定的历史记录中版本。Deployment控制器的回滚操作可使用“kubectl rollout undo”命令完成，例如，下面的命令可将myapp-deploy回滚至之前的版本：

```
**[terminal]
**[delimiter $ ]**[command kubectl rollout undo deployments myapp-deploy]
deployment.apps/myapp-deploy rolled back
```

等待回滚完成后，验证myapp-deploy的ReplicaSet控制器对象是否已恢复到指定的历史版本以确保其回滚正常完成。在“kubectl rollout undo”命令上使用“--to-revision”选项指定revision号码即可回滚到历史特定版本，例如，假设myapp-deploy包含如下的revision历史记录：

```
**[terminal]
**[delimiter $ ]**[command kubectl rollout history deployment myapp-deploy]
deployment.apps/myapp-deploy
REVISION  CHANGE-CAUSE
3          <none>
5          <none>
6          <none>
```

若要回滚到号码为3的revision记录，则使用如下命令即可完成：

```
**[terminal]
**[delimiter $ ]**[command kubectl rollout undo deployment myapp-deploy --to-re
deployment.apps/myapp-deploy rolled back
```

回滚操作中，其revision记录中的信息会发生变动，回滚操作会被当作一次滚动更新追加进历史记录中，而被回滚的条目则会被删除。需要注意的是，如果此前的滚动更新过程处于“暂停”状态，那么回滚操作就需要先将Pod模板改回到之前的版本，然后“继续”更新，否则，其将一直处于暂停状态而无法回滚。

## 5.3.6 扩容和缩容

通过修改 `.spec.replicas` 即可修改Deployment控制器中Pod资源的副本数量，它将实时作用于控制器并直接生效。Deployment控制器是声明式配置，`replicas`属性的值可直接修改资源配置文件，然后使用“`kubectl apply`”进行应用，也可以使用“`kubectl edit`”对其进行实时修改。而前一种方式能将修改结果予以长期保留。

另外，“`kubectl scale`”是专用于扩展某些控制器类型的应用规模的命令，包括 Deployment和Job等。而Deployment通过ReplicaSet控制其Pod资源，因此扩容的方式是相同的，除了命令直接作用的资源对象有所不同之外，这里不再对其进行展开说明。

## 5.4 DaemonSet 控制器

DaemonSet是Pod控制器的又一种实现，用于在集群中的全部节点上同时运行一份指定的Pod资源副本，后续新加入集群的工作节点也会自动创建一个相关的Pod对象，当从集群移除节点时，此类Pod对象也将被自动回收而无须重建。管理员也可以使用节点选择器及节点标签指定仅在部分具有特定特征的节点上运行指定的Pod对象。

DaemonSet是一种特殊的控制器，它有特定的应用场景，通常运行那些执行系统级操作任务的应用，其应用场景具体如下。

- 运行集群存储的守护进程，如在各个节点上运行 glusterd 或 ceph。
- 在各个节点上运行日志收集守护进程，如fluentd或logstash。
- 在各个节点上运行监控系统的代理守护进程，如Prometheus Node Exporter、collectd、Datadog agent、New Relic agent或Ganglia gmond 等。

当然，既然是需要运行与集群内的每个节点或部分节点，于是很多场景中也可以把应用直接运行为工作节点上的系统级守护进程，不过，这样一来就失去了运用Kubernetes管理所带来的便捷性。另外，也只有必须将Pod对象运行于固定的几个节点并且需要先于其他Pod启动时，才有必要使用DaemonSet控制器，否则就应该使用Deployment控制器。

## 5.4.1 创建DaemonSet资源对象

DaemonSet控制器的spec字段中嵌套使用的字段同样主要包括了前面讲到的Pod控制器资源支持的 selector、template和minReadySeconds，并且功能和用法基本相同，但他不支持使用replicas，毕竟DaemonSet并不是基于期望的副本数来控制Pod资源数量，而是基于节点数量，但是template是必选字段。

下面的资源清单文件（filebeat-ds.yaml）示例中定义了一个名为filebeat-ds的DaemonSet控制器，它将在每个节点上运行一个filebeat进程以收集容器相关的日志数据：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: filebeat-ds
  labels:
    app: filebeat
spec:
  selector:
    matchLabels:
      app: filebeat
  template:
    metadata:
      labels:
        app: filebeat
      name: filebeat
    spec:
      containers:
        - name: filebeat
          image: ikubernetes/filebeat:5.6.5-alpine
          env:
            - name: REDIS_HOST
              value: db.ilinux.io:6379
            - name: LOG_LEVEL
              value: info
```

通过清单文件创建DaemonSet资源的命令与其他资源的创建并无不同：

```
**[terminal]
**[delimiter $ ]**[command kubectl apply -f filebeat-ds.yaml]
daemonset.apps "filebeat-ds" created
```

自 Kubernetes 1.8 版本起，DaemonSet 也必须使用 selector 来匹配 Pod 模板中指定的标签，而且它也支持matchLabels和matchExpressions两种标签选择器。

### A.1.1 部署目标

与其他资源对象相同，用户也可以使用“kubectl describe”命令查看DaemonSet对象的详细信息。下面命令的结果信息中，Node-Selector字段的值为空，表示它需要运行于集群中的每个节点之上。而当前集群的节点数量为5，因此，其期望的Pod副本数（Desired Number of Nodes Scheduled）为5，而当前也已经成功创建了5个相关的Pod对象：

### A.1.1 部署目标

### A.1.1 部署目标

```
Path:          /proc
HostPathType:
sys:
Type:          HostPath (bare host directory volume)
Path:          /sys
HostPathType:
Events:        <none>
```

根据DaemonSet资源本身的意义，prometheus-stack-prometheus-node-exporter控制器成功创建的5个Pod对象应该分别运行于集群中的每个节点之上，这一点可以通过如下命令进行验证：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -n kube-prometheus -l app=prometheus-node-exporter
NAME                  NODE
prometheus-stack-prometheus-node-exporter-76zbp   node3
prometheus-stack-prometheus-node-exporter-djvdn    node5
prometheus-stack-prometheus-node-exporter-qrbfp   node1
prometheus-stack-prometheus-node-exporter-zdg6z    node4
prometheus-stack-prometheus-node-exporter-zx2gz   node2
```

集群中的部分工作节点偶尔也需要将Pod对象以单一实例形式运行的情况，例如对于拥有特殊硬件的节点来说，可能会需要为其运行特定的监控代理程序（agent）程序，等等。其实现方式与前面讲到的Pod资源的节点绑定机制类似，只需要在Pod模板的spec字段中嵌套使用nodeSelector字段，并确保其值定义的标签选择器于部分特定工作节点的标签匹配即可。

## 5.4.2 更新DaemonSet对象

DaemonSet 自 Kubernets 1.6 版本起也开始支持更新机制，相关配置定义在 `spec.update-Strategy` 嵌套字段中。目前，它支持 RollingUpdate（滚动更新）和OnDelete（删除时更新）两种更新策略，滚动更新为默认更新的更新策略，工作逻辑类似于Deployment控制，不过仅支持使用`maxUnavailable`属性定义最大不可用Pod资源副本数（默认值为1），而删除时更新的方式则是在删除相应节点的Pod资源后重建并更新为新版本。

例如，将此前创建的filebeat-ds中的Pod模板中的容器镜像升级为“ikubernetes/test/filebeat:5.6.6-alpine”，使用“kubectl set image”命令即可实现：

```
**[terminal]
**[delimiter $ ]**[command kubectl set image daemonsets filebeat-ds filebeat=ikubernetes/test/filebeat:5.6.6-alpine
daemonset.apps/filebeat-ds image updated]
```

从下面命令的返回结果可以看出，filebeat-ds控制器Pod模板中的容器镜像文件已经完成更新，对滚动更新策略来说，它会自动触发更新操作。用户也可以通过filebeat-ds控制器的详细信息中的Events字段等来了解滚动更新的操作过程。从下面的命令结果可以看出，默认的滚动更新策略是一次删除一个工作节点上的Pod资源，待其新版本Pod资源重建完成后再开始操作另一个工作节点上的Pod资源：

## A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl describe daemonsets.apps filebeat-ds]
Name:           filebeat-ds
Selector:       app=filebeat
Node-Selector: <none>
Labels:         app=filebeat
Annotations:   deprecated.daemonset.template.generation: 2
Desired Number of Nodes Scheduled: 5
Current Number of Nodes Scheduled: 5
Number of Nodes Scheduled with Up-to-date Pods: 5
Number of Nodes Scheduled with Available Pods: 5
Number of Nodes Misscheduled: 0
Pods Status:   5 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=filebeat
  Containers:
    filebeat:
      Image:      ikubernetes/filebeat:5.6.6-alpine
      Port:       <none>
      Host Port: <none>
      Environment:
        REDIS_HOST: db.ilinux.io:6379
        LOG_LEVEL:  info
      Mounts:     <none>
      Volumes:    <none>
  Events:
    Type  Reason          Age   From            Message
    ----  -----          ----  --              -----
    Normal SuccessfulCreate 42m   daemonset-controller  Created pod: filebeat-0
    Normal SuccessfulCreate 42m   daemonset-controller  Created pod: filebeat-1
    Normal SuccessfulCreate 42m   daemonset-controller  Created pod: filebeat-2
    Normal SuccessfulCreate 42m   daemonset-controller  Created pod: filebeat-3
    Normal SuccessfulCreate 42m   daemonset-controller  Created pod: filebeat-4
    Normal SuccessfulDelete 40m   daemonset-controller  Deleted pod: filebeat-0
    Normal SuccessfulCreate 40m   daemonset-controller  Created pod: filebeat-0
    Normal SuccessfulDelete 40m   daemonset-controller  Deleted pod: filebeat-1
    Normal SuccessfulCreate 40m   daemonset-controller  Created pod: filebeat-1
    Normal SuccessfulDelete 39m   daemonset-controller  Deleted pod: filebeat-2
    Normal SuccessfulCreate 39m   daemonset-controller  Created pod: filebeat-2
    Normal SuccessfulDelete 39m   daemonset-controller  Deleted pod: filebeat-3
    Normal SuccessfulCreate 39m   daemonset-controller  Created pod: filebeat-3
    Normal SuccessfulDelete 39m   daemonset-controller  Deleted pod: filebeat-4
    Normal SuccessfulCreate 39m   daemonset-controller  (combined from similar
```

DaemonSet 控制器的滚动更新机制也可以借助于minReadySeconds字段控制滚动节奏，必要时可以执行暂停和继续操作，因此它也能够设计为金丝雀发布机制。另外，故障的更新操作也可以进行回滚，包括回滚至revision历史记录中的任何一个

### A.1.1 部署目标

指定的版本。鉴于篇幅，这里不再给出其详细过程，感兴趣的读者可参考 Deployment 控制器的步骤测试其实现。

## 5.5 Job控制器

与Deployment及DaemonSet控制器管理的守护进程类的服务应用不同的是，Job控制器用于调配Pod对象运行一次性任务，容器中的进程在正常运行结束后不会对其进行重启，而是将Pod对象置于“Completed”（完成）状态。若容器中的进程因错误而终止，则需要依据配置确定重启与否，未运行完成的Pod对象因其所在的节点故障而意外终止后会被重新调度。Job控制器的Pod对象的状态转换状态如图5-13所示。

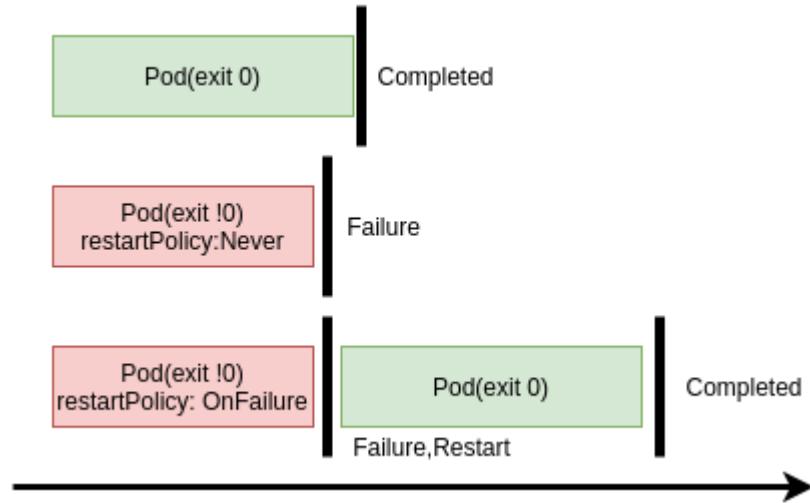


图 1.7.5 - Job 管理下Pod资源的运行方式

实践中，有的作业任务可能需要运行不止一次，用户可以配置他们以串行或并行的方式运行。总结起来，这种类型的Job控制器对象有两种，具体如下。

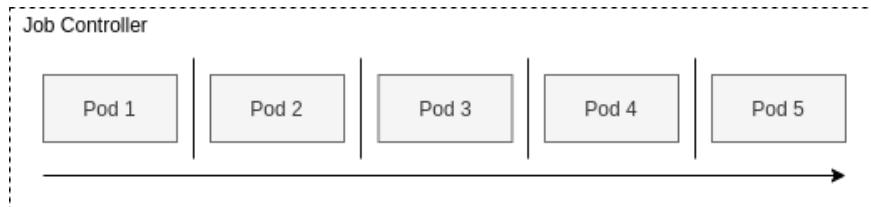


图 1.7.5 - 串行式多任务

- 单工作队列（work queue）的串行Job：即以多个一次性的作业方式串行执行多次作业，直至满足期望的次数，如图5-14所示；这次Job也可以理解为并行度为1的作业执行方式，在某个时刻仅存在一个Pod资源对象。

### A.1.1 部署目标

- 多工作队列的并行形式Job：这种方式可以设置工作队列数，即作业数，每个队列仅负责运行一个作业，如图 5-15a 所示；也可以用有限的工作队列运行较多的作业，即工作队列少于总作业数，相当于运行多个串行作业队列。如图5-15b所示，工作队列即为同时可运行的Pod资源数。

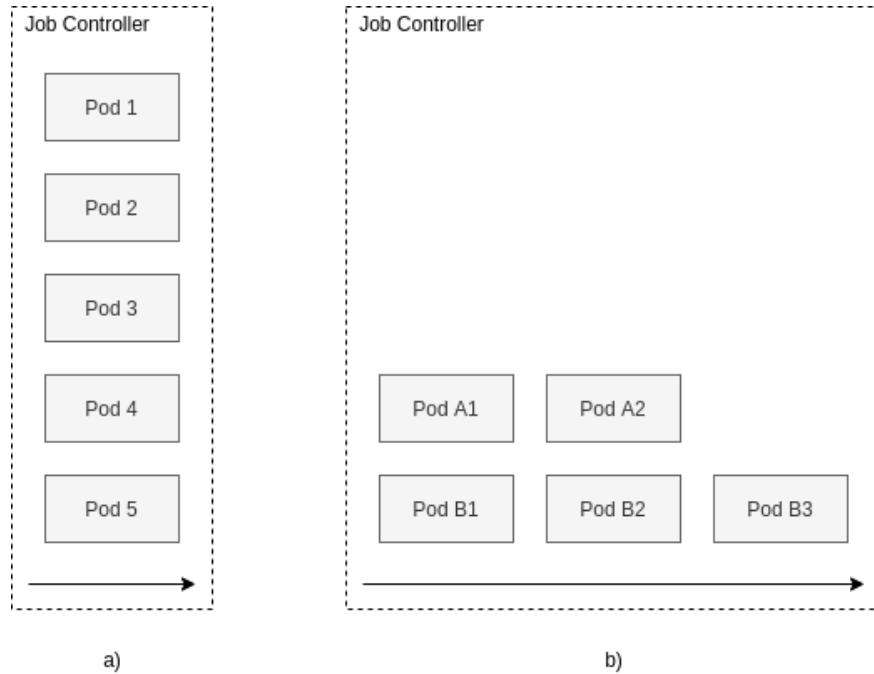


图 1.7.5 - 多队列并行式多任务

Job 控制器常用于管理那些运行一段时间便可“完成”的任务，例如计算或备份操作。

## 5.5.1 创建Job对象

Job 控制器的spec字段内嵌的必要字段仅为template，它的使用方式与Deployment等控制器并无不同。Job会为其Pod对象自动添加“Job-name=JOB\_NAME”和“controller-uid=UID”标签，并使用标签选择器完成对controller-uid标签的关联。需要注意的是，Job位于API群组“batch/v1”之内。下面的资源清单文件（job-example.yaml）中定义了一个Job控制器：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
  template:
    spec:
      containers:
        - name: myjob
          image: alpine
          command:
            - /bin/sh
            - -c
            - sleep 120
  restartPolicy: Never
```

Pod 模板中的 spec.restartPolicy 默认为“Always”，这对Job控制器来说并不适用，因此必须在Pod模板中显式设定restartPolicy属性的值为“Never”或“OnFailure”。

使用“kubectl create”或“kubectl apply”命令完成创建后即可查看相关的任务状态，COMPLETIONS字段右侧数字表示期望并运行的Pod资源数量，而左侧则表示成功完成的Job数：

```
**[terminal]
**[delimiter $ ]**[command kubectl get jobs.batch job-example]
NAME      COMPLETIONS   DURATION   AGE
job-example  0/1           58s       58s
```

相关的Pod资源能够以Job控制器名称为标签进行匹配：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l job-name=job-example]
NAME      READY   STATUS    RESTARTS   AGE
job-example-wfw8s  1/1     Running   0          87s
```

其详细信息中可显示所使用的标签选择器及配置的Pod资源的标签，具体如下：

## A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl describe jobs job-example]
Name:          job-example
Namespace:     default
Selector:      controller-uid=b2e2d178-b9c1-4511-b4f8-9792e2ff0bbd
Labels:        controller-uid=b2e2d178-b9c1-4511-b4f8-9792e2ff0bbd
               job-name=job-example
Annotations:   <none>
Parallelism:  1
Completions:   1
Start Time:    Mon, 14 Sep 2020 10:45:44 +0800
Completed At:  Mon, 14 Sep 2020 10:48:05 +0800
Duration:     2m21s
Pods Statuses: 0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=b2e2d178-b9c1-4511-b4f8-9792e2ff0bbd
            job-name=job-example
  Containers:
    myjob:
      Image:      alpine
      Port:       <none>
      Host Port: <none>
      Command:
        /bin/sh
        -c
        sleep 120
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Events:
    Type  Reason           Age    From            Message
    ----  -----           ----  --  -----
    Normal SuccessfulCreate 4m21s  job-controller  Created pod: job-example-wfw8s
    Normal  Completed       2m    job-controller  Job completed

```

两分钟后，等待sleep命令执行完成并成功退出后，Pod资源即转换为Completed状态，再使用“kubectl get pods”命令查看：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l job-name=job-example]
NAME          READY  STATUS    RESTARTS  AGE
job-example-wfw8s  0/1    Completed   0          7m
```

此时，如果使用“kubectl get jobs”显示job-example的相关信息，那么其COMPLETIONS字段左侧的数字就不再为“0”。

## 5.5.2 并形式Job

将并行度属性 `.spec.parallelism` 的值设置为1，并设置总任务数 `.spec.completions` 属性便能够让Job控制器以串行方式运行多任务。下面是一个串行运行5次任务的Job控制器示例：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-multi
spec:
  completions: 5
  templates:
    spec:
      containers:
        - name: myjob
          image: alpine
          command:
            - /bin/sh
            - -c
            - sleep 120
  restartPolicy: OnFailure
```

在创建之后或创建之前，可以在另一个终端启动Pod资源的列出命令“`kubectl get pods -l job-name=job-multi --watch`”来监控其执行过程。

`.spec.parallelism` 能够定义作业执行的并行度，将其设置为2或者以上的值即可实现并行多队列作业运行。同时，如果 `.spec.completions` 使用的默认值1，则表示并行度即作业总数，如图5-15a所示：如果将 `.spec.completions` 属性值设置为大于 `.spec.parallelism` 的属性值，则表示使用多队列串行任务作业模式，如图5-15b所示。例如，某Job控制器配置中的spec字段嵌套了如下属性，表示以2个队列并行的方式，总共进行5次作业：

```
spec:
  completions: 5
  parallelism: 2
```

### 5.5.3 Job扩容

Job控制器的`.spec.parallelism`，此属性值支持运行时调整从而改变其队列总数，实现扩容和缩容。Job控制器并不支持“`kubectl scale --replicas`”命令，我们可以使用“`kubectl patch jobs.batch job-multi -p '{"spec": {"parallelism": 2}}'`”命令，例如在其运行过程中（未完成之前）将job-multi的并行度扩展为两路：

```
**[terminal]
**[delimiter $ ]**[command kubectl patch jobs.batch job-multi -p '{"spec": {"parallelism": 2}}']
job.batch/job-multi patched
```

执行命令后可以看到，其同时运行的Pod对象副本数量立即扩展到了两个：

```
**[terminal]
**[delimiter $ ]**[command kubectl get pods -l job-name=job-multi]
NAME          READY   STATUS        RESTARTS   AGE
job-multi-lmh52  1/1    Running      0          76s
job-multi-qr4bv  0/1    ContainerCreating  0          3s
```

根据工作节点及其资源可用量，适度提高Job的并行度，能够大大提升其完成效率，缩短运行时间。

## 5.5.4 删除Job

Job控制器待其Pod资源运行完成后，将不再占用系统资源。用户可按需保留或使用资源删除命令将其删除。不过，如果某Job控制器的容器应用总是无法正常结束运行，而其restartPolicy又定位了重启，则它可能会一直处于不停地重启和错误的循环当中。所幸的是，Job控制器提供了两个属性用于抑制这种情况的发生，具体如下：

- `.spec.activeDeadlineSeconds <integer>`：Job的deadline，用于为其指定最大活动时间长度，超出此时长的作业将被终止。
- `.spec.backoffLimit <integer>`：将作业标记为失败状态之前的重试次数，默认值为6。

例如，下面的配置片段表示其失败重试的次数为5，并且如果超出100秒的时间仍未运行完成，那么其将被终止：

```
spec:  
  backoffLimit: 5  
  activeDeadlineSeconds: 100
```

## 5.6 CronJob控制器

CronJob 控制器用于管理 Job 控制器资源的运行时间。Job 控制器定义的作业任务在其控制器资源创建之后便会立即执行，但 CronJob 可以以类似于 Linux 操作系统的周期性任务作业计划（crontab）的方式控制其运行的时间点及重复运行的方式，具体如下：

- 在未来某时间点运行作业一次。
- 在指定的时间点重复运行作业。

CronJob 对象支持使用的时间格式类似于 Crontab，略有不同的是，CronJob 控制器在指定的时间点，“?”和“\*”的意义相同，都表示任何可用的有效值。

## 5.6.1 创建CronJob对象

CronJob控制器的spec字段可以嵌套使用以下字段。

- `jobTemplate <Object>` : Job控制器模板，用于CronJob控制器生成Job对象；必选字段。
- `schedule <string>` : Cron格式的作业调度运行时间点；必选字段。
- `concurrencyPolicy <string>` : 并行执行策略，可用值有“Allow”（允许）、“Forbid”（禁止）和“Replace”（替换），用于定义前一次作业运行尚未完成时是否以及如何运行后一次作业。
- `failedJobHistoryLimit <integer>` : 为失败的任务执行保留的历史记录数，默認為1。
- `successfulJobsHistoryLimit <integer>` : 为成功的任务执行保留的历史记录数，默認為3。
- `startingDeadlineSeconds <integer>` : 因各种原因缺乏执行作业的时间点所导致的启动作业错误的超时时长，会被记入错误历史记录。
- `suspend <boolean>` : 是否挂起后续的任务执行，默認為false，对运行中的作业不会产生影响。

下面是一个定义在资源清单文件（cronjob-example.yaml）中的CronJob资源对象示例，它每隔2分钟运行一次由jobTemplate定义的简单任务：

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob-example
  labels:
    app: mycronjob
spec:
  schedule: "*/2 * * * *"
  jobTemplate:
    metadata:
      labels:
        app: mycronjob-jobs
    spec:
      parallelism: 2
      template:
        spec:
          containers:
            - name: myjob
              image: alpine
              command:
                - /bin/sh
                - -c
                - date; echo Hello from the kubernetes cluster; sleep 10
      restartPolicy: OnFailure
```

### A.1.1 部署目标

运行资源创建命令创建上述CronJob资源对象，而后再通过资源对象的相关信息了解运行命令。下面创建结果中的SCHEDULE是指其调度的时间点，SUSPEND表示后续任务是否处于挂起状态，即暂停任务的调度和运行，ACTIVE表示活动状态的Job对象的数量，而LAST SCHEDULE则表示上次调度运行至此刻的时长：

```
**[terminal]
**[delimiter $ ]**[command kubectl get cronjobs.batch cronjob-example]
NAME          SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob-example  */2 * * * *  False       0        <none>    23s
```

自 Kubernetes 1.8 起，CronJob资源所在的API资源组从 batch/v2alpha1 移至 batch/v1beta1 中，并且查看其资源格式时也要使用 --api-version 选项指定其所在的资源组，即“kubectl explain cronjob --api-version='batch/v1beta1'”。

## 5.6.2 CronJob的控制机制

CronJob控制器时一个更高级别的资源，它以Job控制器资源为其管控对象，并借助它管理Pod资源对象。因此，要使用类似如下命令来查看某CronJob控制器创建的Job资源对象，其中的标签“mycronjob-jobs”是在创建cronjob-example时为其指定。不过，只有相关的Job对象被调度执行时，此命令才能将其正常列出。可列出的Job对象的数量取决于CronJob资源的`.spec.successfulJobsHistoryLimit`的属性值，默认为3。

```
**[terminal]
**[delimiter $ ]**[command kubectl get jobs -l app=mycronjob-jobs]
NAME          COMPLETIONS   DURATION   AGE
cronjob-example-1600067520  2/1 of 2    26s        4m36s
cronjob-example-1600067640  2/1 of 2    21s        2m36s
cronjob-example-1600067760  2/1 of 2    21s        36s
```

如果作业重复执行时指定的时间点较近，而作业执行时长（普遍或偶尔）跨国过了两次执行的时间长度，则会出现两个Job对象同时存在的情形。这些Job对象可能会存在无法或不能同时运行的情况，这个时候就要通过`.spec.concurrentPolicy`属性控制作业并存的机制，其默认值为“Allow”，即允许前后Job，甚至属于同一个CronJob的更多Job同时运行。其他两个可用值中，“Forbid”用于禁用前后两个Job同时运行，如果前一个尚未结束，后一个则不予启动（跳过）。“Replace”用于让后一个Job取代前一个，即终止前一个并启动后一个。

## 5.7 ReplicationController

ReplicationController（简称rc或RC）是Kubernetes较早实现的Pod控制器，用于确保Pod资源的不间断运行。不过，Kubernetes后来设计了ReplicaSet及更高一级的控制器Deployment来取代ReplicationController，并表示在后来的版本中可能会废弃RC。因此，这里不再对ReplicationController做过多的介绍。事实上，它的使用方式与ReplicaSet相同，一旦用到时，绝大多数操作都可以迁移使用，感兴趣的读者可以自行测试。

## 5.8 Pod中断预算

尽管Deployment或ReplicaSet一类的控制器能够确保相应的Pod对象的副本数量不断逼近期望的数量，但它却无法保证某一时刻一定会存在指定数量或比例的Pod对象，然而这种需求在某些强调服务可用性的场景中却是必备的。于是，Kubernetes自1.4版本起开始引入Pod中断预算（PodDisruptionBudget，简称PDB）类型的资源，用于为那些资源（Voluntary）中断做好预算方案（Budget），限制可自愿中断的最大Pod副本数或确保最少可用的Pod副本数，以确保服务的高可用性。

Pod对象会一直存在，除非有意将其销毁，或者出现了不可避免的硬件或系统软件错误。**非自愿中断**是指那些由不可控外界因素导致的Pod中断退出操作，例如，硬件或系统内核故障、网络故障以及节点资源不足导致Pod对象被驱逐等；而那些由用户特地执行的管理操作导致的Pod中断则称为“**自愿中断**”，例如排空节点、人为删除Pod对象、由更新操作触发的Pod对象重建等。部署在Kubernetes的每个应用程序都可以创建一个对应的PDB对象以限制自愿中断时最大可以中断的副本数或者最少应该保持可用的副本数，从而保证应用自身的高可用性。

PDB资源可以用来保护由控制器管理的应用，此时几乎必然意味着PDB使用等同于相关控制器对象的标签选择器以精确关联至目标Pod对象，支持的控制器类型包括Deployment、ReplicaSet和StatefulSet等。同时，PDB对象也可以用来保护那些纯粹是由定制的标签选择器自由选择的Pod对象。

定义PDB资源时，其spec字段主要嵌套使用以下三个字段。

- `selector <Object>`：当前PDB对象使用的标签选择器，一般是与相关Pod控制器使用同一个选择器。
- `minAvailable <string>`：Pod自愿中断的场景中，至少要保证可用的Pod对象数量或比例，要阻止任何Pod对象发生自愿中断，可将其设置为100%。
- `maxUnavailable <string>`：Pod自愿中断的场景中，最多可转换为不可用状态的Pod对象数量或比例，0值意味着不允许Pod对象进行自愿中断；此字段与minAvailable互斥。

下面的示例定义了一个PDB对象，它对5.3.1节中由Deployment控制器myapp-deploy创建的Pod对象设置了Pod中断预算，要求其最少可用的Pod对象数量为2个：

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: myapp-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: myapp
```

PDB资源对象创建完成后，在它的简要信息输出中也标明了最少可用的Pod对象个数，以及允许中断的Pod对象个数：

### A.1.1 部署目标

```
**[terminal]
**[delimiter $ ]**[command kubectl get pdb]
NAME      MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
myapp-pdb    2                  N/A                1           12s
```

接下来可通过手动删除myapp-deploy控制器下的所有Pod对象模拟自愿中断过程，并监控各Pod对象被终止的过程来验证PDB资源对象的控制功效。

## 5.9 本章小结

本章主要讲解了Kubernetes的Pod控制器，它们是“工作负载”类资源的核心组成部分，是基于Kubernetes运行应用的最重要的资源类型之一，具体如下。

- 工作负载类型的控制器根据业务需求管控Pod资源的生命周期。
- ReplicaSet可以确保守护进程型的Pod资源始终具有精确的、处于运行状态的副本数量，并支持Pod规模的伸缩机制；它是新一代的ReplicationController控制器，不过应该直接使用ReplicaSet，而是要使用Deployment。
- Deployment是构建在ReplicaSet上更加抽象的工作负载型控制器，支持多种更新策略及发布机制。
- Job控制器能够控制相应的作业任务得以正常完成并退出，支持并行式多任务。
- CronJob控制器用于控制周期性作业任务，其功能类似于Linux操作系统上的Crontab。
- PodDisruptionBudget资源对象为Kubernetes系统上的容器化应用提供了高可用能力。

## 第六章 Service 和 Ingress

运行于Pod中的部分容器化应用是向客户端提供服务的守护进程，例如，nginx、tomcat和etcd等，它们受控于控制器资源对象，存在生命周期，在自愿或非自愿中断后只能被重构的新Pod对象所取代，属于非可再生类的组建。于是，在动态、弹性的管理模型下，Service资源用于为此类Pod对象提供一个固定、统一的访问接口及负载均衡的能力，并支持借助新一代DNS系统的服务发现功能，解决客户端发现并访问容器化应用的难题。

然而，Service及Pod对象的IP地址都仅在Kubernetes集群内可达，它们无法接入集群外部的访问流量。解决此类问题的办法中，除了在单一节点上做端口暴露（hostPort）及让Pod资源共享使用工作节点的网络名称空间（hostNetwork）之外，更推荐用户使用的是NodePort或Loadbalancer类型的Service资源，或者是有七层负载均衡能力的Ingress资源。

## 6.1 Service资源及其实现模型

Service是kubernetes的核心资源之一，通常可看作微服务的一种实现。事实上它是一种抽象：通过规则定义出由多个Pod对象组合而成的逻辑集合，以及访问这组Pod的策略。Service关联Pod资源的规则要借助于标签选择器来完成，这一点类似于第五章讲到的Pod控制器。

## 6.1.1 Service资源概述

由Deployment等控制器管理的Pod对象中断后会由新建的资源对象所取代，而扩容后的应用则会带来Pod对象群体的变动，随之变化的还有Pod的IP地址访问接口等，这也是编排系统之上的应用程序必然要面临的问题。例如，当图6-1中的Nginx Pod作为客户端访问tomcat Pod中的应用时，IP的变动或应用规模的缩减会导致客户端访问错误，而Pod规模的扩容又会使得客户端无法有效的使用新增的Pod对象，从而影响达成规模扩展之目的。为此，Kubernetes特地设计了Service资源来解决此类问题。

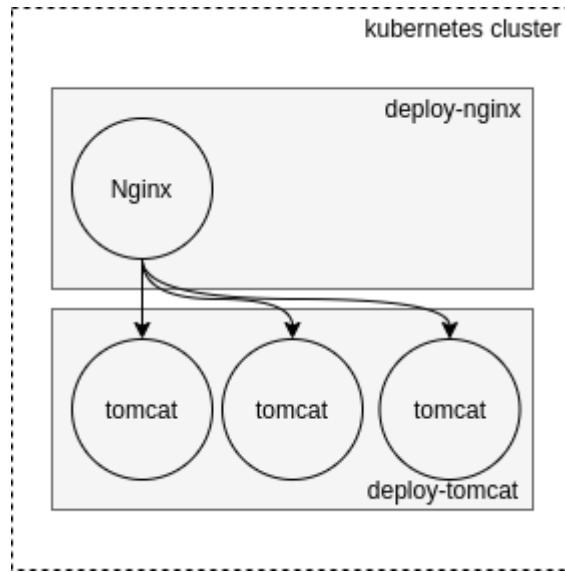


图 1.8.1.1 - Pod及其客户端示例

Service资源基于标签选择器将一组Pod定义成一个逻辑组合，并通过自己的IP地址和端口调度代理请求至组内的Pod对象之上，如图6-2所示，它向客户端隐藏了真实的、处理用户请求的Pod资源，使得客户端的请求看上去就像是由Service直接处理并进行响应的一样。

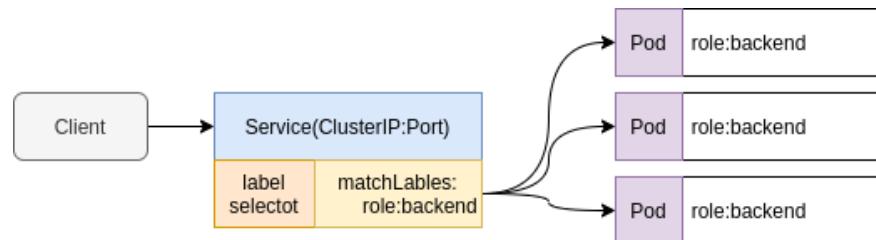


图 1.8.1.1 - Kubernetes Service 资源模型示意图

### A.1.1 部署目标

Service对象的IP地址也称为Cluster IP，它位于为Kubernetes集群配置指定专用IP地址的范围之内，而且是一种虚拟IP地址，它在Service对象创建后即保持不变，并且能够被同一集群中的Pod资源所访问。Service端口用于接收客户端请求并将其转发至后端的Pod中应用的相应端口上，因此，这种代理机制也称为“端口代理”（port proxy）或四层代理，它工作于TCP/IP协议栈的传输层。

通过其标签选择器匹配到的后端Pod资源不止一个时，Service资源能够以负载均衡的方式进行流量调度，实现了请求流量的分发机制。Service与Pod对象之间的关联关系通过标签选择器以松耦合的方式建立，它可以先于Pod对象创建而不会发生错误，于是，创建Service与Pod资源的任务可由不同的用户分别完成，例如，服务架构的设计和创建由运维工程师进行，而填充其实现的Pod资源的任务则可交由开发者进行。Service、控制器与Pod之间的关系如图6-3所示。

Service资源会通过API Server持续监视着（watch）标签选择器匹配到的后端Pod对象，并实时跟踪各对象的变动，例如，IP地址变动、对象增加或减少等。不过，需要特别说明的是，Service并不直接链接至Pod对象，它们之间还有一个中间层——Endpoint资源对象，它是一个由IP地址和端口组成的列表，这些IP地址和端口则来自于由Service的标签选择器匹配到的Pod资源。这也是很多场景中会使用“Service的后端端点”（Endpoint）这一术语的原因。默认情况下，创建Service资源对象时，其关联的Endpoints对象会自动创建。

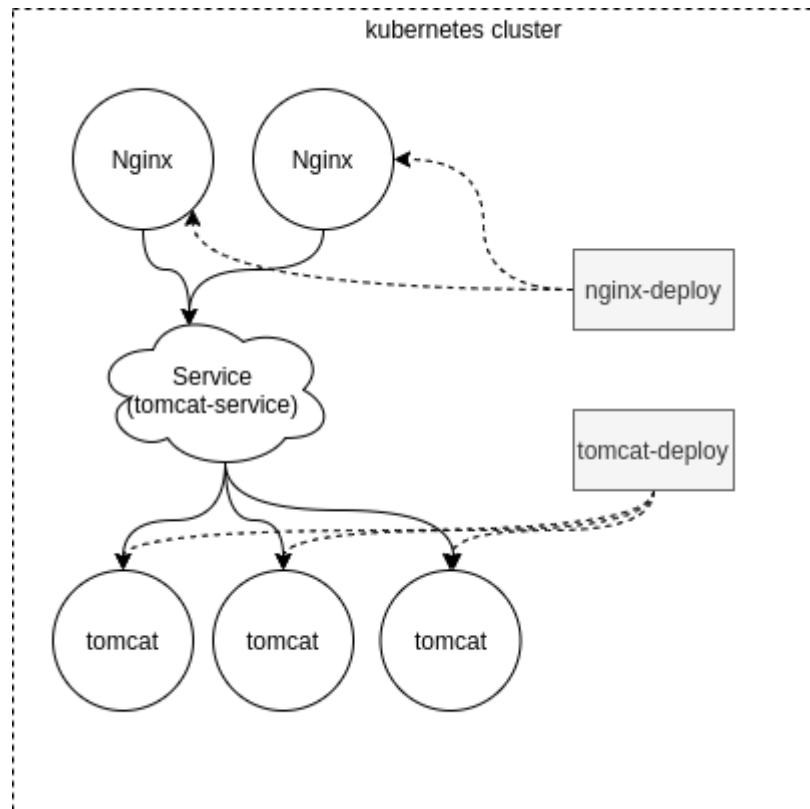


图 1.8.1.1 - Service、控制器与Pod

## 6.1.2 虚拟IP和服务代理

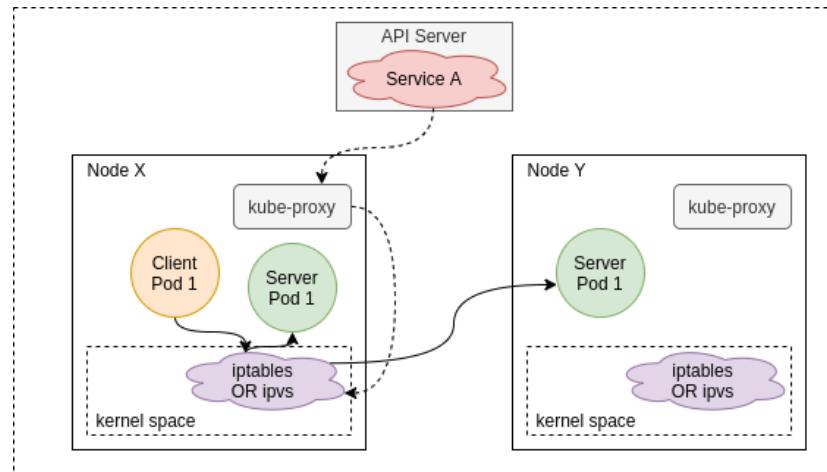
简单来讲，一个Service对象就是工作节点上的一些iptables或ipvs规则，用于将到达Service对象IP地址的流量调度转发至相应的Endpoint对象指向的IP地址和端口之上。工作于每个工作节点的kube-proxy组件通过API Server持续监控着各Service及其关联的Pod对象，并将其创建或变动实时反映至当前工作节点上相应的iptables或ipvs规则上。客户端、Service及其Pod对象的关系如图6-4所示。

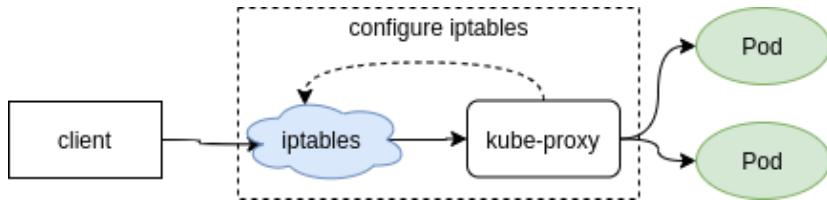
Netfilter是Linux内核中用于管理网络报文的框架，它具有网络地址转换（NAT）、报文改动和报文过滤等防火墙功能，用户借助于用户空间的iptables等工具可按需自由定制规则使用其各项功能。ipvs是借助于Netfilter实现的网络请求报文调度框架，支持rr、wrr、lc、wlc、sh、sed和nq等十余种调度算法，用户空间的命令行工具是ipvsadm，用于管理工作于ipvs之上的调度规则。

Service IP事实上是用于生成iptables或ipvs规则时使用的IP地址，它仅用于实现Kubernetes集群网络的内部通讯，并且仅能够将规则中定义的转发服务的请求作为目标地址予以响应，这也是它被称为虚拟IP的原因之一。kube-proxy将请求代理至相关端点的方式有三种：userspace（用户空间）、iptables和ipvs。

### 1. userspace代理模型

此处的userspace是指Linux操作系统的用户空间。这种模型中，kube-proxy负责跟踪API Server上的Service和Endpoint对象的变动（创建或删除），并根据此调整Service资源的定义。对于每个Service对象，它会随机打开一个本地端口（运行于用户空间的kube-proxy进程负责监听），任何到达此代理端口的请求都将被代理至的当前Service资源后端的各Pod对象上，至于会挑中哪个Pod对象则取决于的当前Service资源的调度方式，默认的调度算法是轮询（round-robin），其工作逻辑如图6-5所示。另外，此类的Service对象还会创建iptables规则以捕获任何到达ClusterIP的端口的流量。在Kubernetes1.1版本之前，userspace是默认的代理模型。





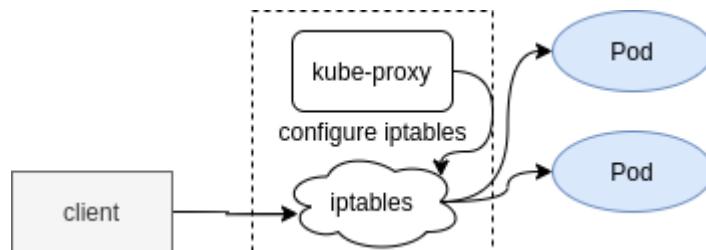
这种代理模型中，请求流量到达内核空间后经由套接字送往用户空间kube-proxy，而后再有它送回内核空间，并调度至后端Pod。这种方式中，请求在内核空间和用户空间来回转发必然会导致效率不高。

## 2. iptables代理模型

同前一种代理模型类似，iptables代理模型中，kube-proxy负责跟踪API Server上Service和Endpoint对象的变动（创建或删除），并据此做出Service资源定义的变动。同时，对于每个Service，它都会创建iptables规则直接捕获到达ClusterIP和Port的流量，并将其重定向至当前Service后端，如图6-6所示。对于每个Endpoint对象，Service资源会为其创建iptables规则并关联至挑选的后端Pod资源，默认算法是随即调度（random）。iptables代理模式由Kubernetes1.1版本引入，并自1.2版本开始成为默认的类型。

在创建Service资源时，集群中每个节点上的kube-proxy都会收到通知并将其定义为当前节点上的iptables规则，用于转发工作接口接收到的与此Service资源的ClusterIP和端口的相关流量。客户端发来的请求被相关的iptables规则进行调度和目标地址转换（DNAT）后再转发至集群内的Pod对象上。

相对于用户空间模型来说，iptables模型无须将流量在用户空间和内核空间来回切换，因而更加高效和可靠。不过，其缺点是iptables代理模型不会在被挑中的后端Pod资源无响应时自动进行重定向，而userspace模型则可以。

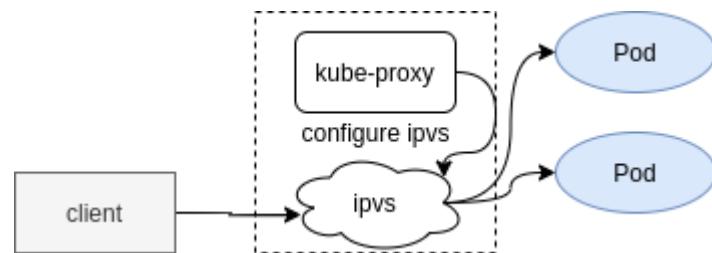


## 3. ipvs代理模型

Kubernetes自1.9-alpha版本引入了ipvs代理模型，且自1.11版本起成为默认设置。此中模型中，kube-proxy跟踪API Server上Service和Endpoint对象的变动，据此来调用netlink接口创建ipvs规则，并确保与API Server中的变动保持同步，如图6-7所示。它与iptables规则的不同之处仅在于其请求流量的调度功能由ipvs实现，余下的其他功能仍由iptables完成。

类似于iptables模型，ipvs构建于netfilter的钩子函数之上，但它使用hash作为底层数据结构并工作于内核空间，因此具有流量转发速度快、规则同步性能好的特性。另外，ipvs支持众多调度算法，例如rr、lc、dh、sh、sed和nq等。

### A.1.1 部署目标



## 6.2 Service资源的基础应用

Service资源本身并不提供任何服务，真正处理并响应客户端请求的是后端的Pod资源，这些Pod资源通常由第5章中介绍的各类控制器对象所创建和管理，因此Service资源通常要与控制器资源（最为常见的控制器之一是Deployment）协同使用以完成应用的创建和对外发布。

## 6.2.1 创建Service资源

创建Service对象的常用方法有两种：一是直接使用“kubectl expose”命令，这在前面第三章中已经介绍过其使用方式；另一个的使用资源配置文件，它与此前使用资源清单配置其他资源的方法类似。定义Service资源对象时，spec的两个较为常用的内嵌字段分别是为selector和ports，分别用于定义使用的标签选择器和要暴露的端口。下面的配置清单是一个Service资源示例：

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
spec:
  selector:
    app: myapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Service资源myapp-svc通过标签选择器关联至标签为“app=myapp”的各Pod对象，它会自动创建名为myapp-svc的Endpoints资源对象，并自动配置一个ClusterIP，暴露的端口由port字段进行指定，后端各Pod对象的端口则由targetPort给出，也可以使用同port字段的默认值。myapp-svc创建完成后，使用下面的命令既能获取相关信息输出以了解资源的状态：

```
**[terminal]
**[delimiter $ ]**[command kubectl get svc myapp-svc]
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
myapp-svc  ClusterIP  10.233.16.105  <none>          80/TCP      28s
```

上面命令中的结果显示，myapp-svc的类型为默认的ClusterIP，其使用的地址自动配置为10.107.208.93。此类型的Service对象仅能通过此IP地址接受来自于集群内部的客户端Pod中的请求。若集群上存在标签为“app=myapp”的Pod资源，则他们会被关联和创建，作为此Service对象的后端Endpoint对象，并负责接受相应的请求流量。类似下面的命令可用于获取Endpoint资源的端点列表，于其相关的端点是由第五章中的Deployment控制器创建的Pod对象的套接字信息组成：

```
**[terminal]
**[delimiter $ ]**[command kubectl get endpoints myapp-svc]
NAME      ENDPOINTS
myapp-svc  10.233.105.43:80,10.233.70.42:80,10.233.90.99:80      AGE
                                                               8m29s
```

也可以不为Service资源指定spec.selector属性值，其关联的Pod资源可由用户手动创建Endpoints资源进行定义。

## 6.2.2 向Service对象请求服务

Service资源的默认类型为ClusterIP，它仅能接收来自集群中的Pod对象中的客户端程序的访问请求。下面创建一个专用的Pod对象，利用其交互式接口完成访问测试。为了简单起见，这里选择直接创建一个临时使用的Pod对象作为交互使用的客户端进行，它使用CirrOS镜像，默认的命令提示符为“/#”：

```
**[terminal]
**[delimiter $ ]**[command kubectl run cirros-$RANDOM --rm -it --image=cirros .
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in
If you don't see a command prompt, try pressing enter.
/ #
```

CirrOS是设计用来进行云计算环境测试的Linux微型发行版，它拥有HTTP客户端工具curl等。

而后，在容器的交互式接口中使用curl命令对myapp-svc服务的ClusterIP(10.233.16.105)和Port(80/tcp)发起访问请求测试：

```
**[terminal]
**[delimiter / # ]**[command curl http://10.233.16.105:80]
Hello MyApp | Version: v1 | <a href="hostname.html">Pod Name</a>
```

myapp容器中的“/hostname.html”页面能够输出当前容器的主机名，可反复向myapp-svc的此URL路径发起多次请求已验证其调度的效果：

```
**[terminal]
**[delimiter / # ]**[command for loop in 1 2 3 4; do curl http://10.233.16.105
myapp-deploy-5cbd66595b-fjzdm
myapp-deploy-5cbd66595b-2lhds
myapp-deploy-5cbd66595b-mzbn6
myapp-deploy-5cbd66595b-fjzdm
```

当前Kubernetes集群的Service代理模式为iptables，他默认使用随机调度至与其关联的某个后端Pod资源上。命令取样次数越大，其调度效果也越来越接近于算法的目标效果。

## 6.2.3 Service会话粘性

Service资源还支持Session affinity（粘性会话）机制，它能够将来自同一个客户端的请求始终转发至同一个后端的Pod对象，这意味着它会影响调度算法的流量分发功能，进而降低其负载均衡的效果。因此，当客户端访问Pod中的应用程序时，如果有基于客户端身份保存某些私有信息，并基于这些私有信息追踪用户的活动等一类的需求时，那么应该启用session affinity机制。

Session affinity的效果仅会在一定时间期限内生效，默认值为10800秒，超出此时长之后，客户端的再次访问会被调度算法重新调度。另外，Service资源的Session affinity仅能基于客户端IP地址识别客户端身份，它会把经由同一NAT服务器进行源地址转换的所有客户端识别为同一个客户端，调度颗粒度粗糙且效果不佳，因此，实践中并不推荐使用此种方法实现粘性会话。此节仅用于读者介绍其功能及实现。

Service资源通过`.spec.sessionAffinity`和`.spec.sessionAffinityConfig`两个字段配置粘性会话。`.spec.sessionAffinity`字段用于定义要使用的粘性会话的类型，它仅支持使用“None”和“ClientIP”两种属性值。

- None：不使用sessionAffinity，默认值。
- ClientIP：基于客户端IP地址识别客户端身份，把来自同一源IP地址的请求始终调度至同一个Pod对象。

在启用粘性会话机制时，`.spec.sessionAffinityConfig`用于配置其会话保持的时长，它是一个嵌套字段，使用格式如下所示，其可用的时长范围为“1~86400”，默认为10800秒：

```
spec:
  sessionAffinity: ClientIP
  sessionAffinityConfig:
    ClientIP:
      timeoutSeconds: <integer>
```

例如，基于默认的10800秒的超时时长，使用下面的命令修改此前的myapp-svc使用Session affinity机制：

```
**[terminal]
**[delimiter $ ]**[command kubectl patch services myapp-svc -p '{"spec": {"sessionAffinity": "ClientIP", "sessionAffinityConfig": {"timeoutSeconds": 10800}}}' --type=JSONpath service/myapp-svc patched]
```

而后再次于交互式客户端内测试其访问效果即可验证其会话粘性效果。

```
**[terminal]
**[delimiter / #]**[command for loop in 1 2 3 4; do curl http://10.233.16.105:31111/app; done]
myapp-deploy-5cbd66595b-2lhd
myapp-deploy-5cbd66595b-2lhd
myapp-deploy-5cbd66595b-2lhd
myapp-deploy-5cbd66595b-2lhd
```

### A.1.1 部署目标

测试完成后，为了保证本章后续的其他使用效果测试不受其影响，建议将其关闭。  
当然，用户也可以使用“kubectl edit”命令直接编辑活动Service对象的配置清单。

## 6.3 服务发现

微服务意味着存在更多的独立服务，但它们并非独立的个体，而是存在着复杂的依赖关系且彼此之间通常需要进行非常频繁地交互和通信的群体。然而，建立通信之前，服务和服务之间该如何获知彼此的地址呢？在Kubernetes系统上，Service为Pod中的服务类应用提供了一个稳定的访问入口，但Pod客户端中的应用如何得知某个特定Service资源的IP和端口呢？这个时候就需要引入服务发现（Service Discovery）的机制。

### 6.3.1 服务发现概述

简单来说，服务发现就是服务或者应用之间互相定位的过程。不过，服务发现并非新概念，传统的单位应用架构时代也会用到，只不过单体应用的动态性不强，更新和重新发布的频度较低，通常以月甚至年计，基本上不会进行自动伸缩，因此服务发现的概念无须显性强调。在传统的单体应用网络位置发生变化时，由IT运维人员手动更新一下相关的配置文件基本上就能解决问题。但在微服务应用场景中，应用被拆分成众多的小服务，它们按需创建且变动频繁，配置信息基本无法事先写入配置文件中并及时跟踪和反映动态变化，因此服务发现的重要性便随之凸显。

服务发现机制的基本实现，一般是事先部署好一个网络位置较为稳定的服务注册中心（也称为服务总线），服务提供者（服务端）向注册中心注册自己的位置信息，并在变动后及时予以更新，相应地，服务消费者则周期性地从注册中心获取服务提供者的最新位置信息从而“发现”要访问的目标服务资源。复杂的服务发现机制还能够让服务提供者提供其描述信息、状态信息及资源使用信息等，以供消费者事先更为复杂的服务选择逻辑。

实践中，根据服务发现过程的实现方式，服务发现还可分为两种类型：客户端发现和服务端发现。

- 客户端发现：由客户端到服务注册中心发现其依赖到的服务的相关信息，因此，它需要内置特定的服务发现程序和发现逻辑。
- 服务端发现：这种方式需要额外用一个称为中央路由器或服务均衡器的组件；服务消费者将请求发往中央路由器或者负载均衡器，由它们负责查询服务注册中心获取服务提供者的位置信息，并将服务消费者的请求转发给服务提供者。

由此可见，服务注册中心是服务发现得以落地的核心组件。事实上，DNS可以算是最为原始的服务发现系统之一，不过在服务动态性很强的场景中，DNS记录的传播速度可能会跟不上服务的变更速度，因此它并不适用于微服务环境。另外，传统实践中，常见的服务注册中心是Zookeeper和etcd等分布式键值存储系统，不过，它们只能提供基本的数据存储功能，距离实现完整的服务发现机制还有大量的二次开发任务需要完成。另外，它们更注重数据的一致性，这与有着更高的服务可用性的微服务发现场景中的需求不太相符。

Netflix的Eureka是目前较为流行的服务发现系统之一，它是专门开发用来实现服务发现的系统，以可用性目的为先，可以在多种故障期间保持服务发现和服务注册功能可用，其设计原则遵从“存在少量的错误数据，总比完全不可用要好”。另一个同级别的实现是Consul，它是由HashiCorp供能提供的商业产品，不过该公司还提供了一个开源基础版本。它于服务发现的基础功能之外还提供了多数据中心的部署能力等一众出色的特性。

尽管传统的DNS系统不适合于微服务环境中的服务发现，但SkyDNS项目（后来称kubedns）却是一个有趣的实现，它结合了古老的DNS技术和时髦的Go语言、Raft算法，并构建于etcd存储系统之上，为Kubernetes系统实现了一种服务发现机制。Service资源为Kubernetes提供了一个较为稳定的抽象层，这有点类似于服务端发现的方式，于是也就不存在DNS服务的时间窗口的问题。

Kubernetes自1.3版本开始，其用于服务发现的DNS更新为了KubeDNS，而类似的另一个基于较新的DNS的服务发现项目是由CNCF（Cloud Native Computing Foundation）孵化的CoreDNS，它基于Go语言开发，通过串接一组实现DNS功能

### A.1.1 部署目标

的插件链进行工作。自Kubernetes1.11版本起，CoreDNS取代kubeDNS称为默认的DNS附件。不过，Kubernetes依然支持使用环境变量进行服务发现。

## 6.3.2 服务发现方式：环境变量

创建Pod资源时，kubelet会将其所属名称空间内的每个活动的Service对象以一系列环境变量的形式注入其中。它支持使用Kubernetes Service环境变量以及与Docker的links兼容的变量。

### 1. kubernetes Service 环境变量

Kubernetes为每个Service资源生成包括以下形式的环境变量在内的一系列环境变量，在同一名称空间中创建的Pod对象都会自动拥有这些变量。

- {SVCNAME}\_SERVICE\_HOST
- {SVCNAME}\_SERVICE\_PORT

如果SVCNAME中使用了连接线，那么Kubernetes会在定义为环境变量时将其转换为下划线。

### 2. Docker Link 形式的环境变量

Docker使用`--link`选项实现容器连接时所设置的环境变量形式，具体使用方式请参考Docker的相关文档。在创建Pod对象时，Kubernetes也会将与此形式兼容的一系列环境变量注入Pod对象中。

例如，在Service资源myapp-svc创建后创建的Pod对象中查看可用的环境变量，其中以MYAPP\_SVC\_SERVICE开头的表示Kubernetes Service环境变量，名称中不包含“SERVICE”字符串的环境变量为Docker Link形式的环境变量：

```
**[terminal]
**[delimiter / # ]**[command env | grep -i myapp]
MYAPP_SVC_PORT_80_TCP_ADDR=10.98.57.156
MYAPP_SVC_PORT_80_TCP_PORT=80
HOSTNAME=myapp-deploy-5cbd66595b-2lhd
MYAPP_SVC_PORT_80_TCP_PROTO=tcp
MYAPP_SVC_PORT_80_TCP=tcp://10.98.57.156:80
MYAPP_SVC_SERVICE_HOST=10.98.57.156
MYAPP_SVC_SERVICE_PORT=80
MYAPP_SVC_PORT=tcp://10.98.57.156:80
```

基于环境变量的服务发现其功能简单、易用，但存在一定的局限，例如，仅有那些与创建Pod对象在同一名称空间中且事先存在的Service对象的信息才会以环境变量的形式注入，那些处于非同一名称空间，或者是在Pod资源创建之后才创建的Service对象的相关环境变量则不会被添加。幸而，基于DNS的发现机制并不存在此类限制。

### 6.3.3 ClusterDNS和服务发现

Kubernetes 系统之上用于名称解析和服务发现的ClusterDNS是集群的核心附件之一，集群中创建的每个Service对象，都会尤其自动生成相关的资源记录。默认情况下，集群内各Pod资源会自动配置其作为名称解析服务器，并在其DNS搜索中包含它所属名称空间的域名后缀。

无论是使用kubeDNS还是CoreDNS，它们提供的基于DNS的服务发现解决方案都会负责解析以下资源记录（Resource Record）类型以实现服务发现。

1. 拥有ClusterIP的Service资源，需要具有以下类型的资源记录。

- A记录： <service>.<ns>.svc.<zone>. <ttl> IN A <cluster-ip>
- SRV记录： \_<port>.\_<proto>.<service>.<ns>.svc.<zone>. <ttl> IN SRV  
    <weight> <priorty> <port-number> <service>.<ns>.svc.<zone>
- PTR记录： <d>.<c>.<b>.<a>.in-addr.arpa. <ttl> IN PTR <service>.  
    <ns>, svc, <zone>

2. Headless 类型的Service资源，需要具有以下类型的资源记录。

- A记录： <service>.<ns>.svc.<zone>. <ttl> IN A <endpoint-ip>
- SRV记录： \_<port>.\_<proto>.<service>.<ns>.svc.<zone>. <ttl> IN SRV  
    <weight> <priorty> <port-number> <hostname>.<service>.<ns>.svc.<zone>
- PTR记录： <d>.<c>.<b>.<a>.in-addr.arpa. <ttl> IN PTR <hostname>.  
    <service>.<ns>.svc.<zone>

3. ExternalName 类型的Service资源，需要具有CNAME类型的资源记录。

- CNAME记录： <service>.<ns>.svc.<zone>. <ttl> IN CNAME <extname>

名称解析和服务发现是Kubernetes系统许多功能得以实现的基础服务，它通常是集群安装完成后应该立即部署的附加组件。使用kubeadm初始化一个集群时，它甚至会自动进行部署。

## 6.3.4 服务发现方式：DNS

创建Service资源对象时，ClusterDNS会为它自动创建资源记录用于名称解析和服务注册，于是，Pod资源可直接使用标准的DNS名称来访问这些Service资源。每个Service对象相关的DNS记录包含如下两个：

- {SVCNAME}.{NAMESPACE}.{CLUSTER\_DOMAIN}
- {SVCNAME}.{NAMESPACE}.svc.{CLUSTER\_DOMAIN}

另外，在前面第2章的部署参数中，“--cluster-dns”指定了集群DNS服务的工作地址，“--cluster-domain”定义了集群使用的本地域名，因此，系统初始化时默认会将“cluster.local”和主机所在的域“ilinux.io”作为DNS的本地域使用，这些信息会在Pod创建时以DNS配置的相关信息注入它的/etc/resolv.conf配置文件中。例如，在此前创建的用于交互式Pod资源的客户端中查看其配置，命令如下：

```
**[terminal]
**[delimiter / #]**[command cat /etc/resolv.conf]
nameserver 169.254.25.10
search default.svc.cluster.local svc.cluster.local cluster.local ilinux.io
options ndots:5
```

上述search参数中指定的DNS各搜索域，是以次序指定的几个域名后缀，具体如下所示：

- {NAMESPACE}.svc.{CLUSTER\_DOMAIN}：如default.svc.cluster.local
- svc.{CLUSTER\_DOMAIN}：如svc.cluster.local
- {CLUSTER\_DOMAIN}：如cluster.local
- {WORK\_NODE\_DOMAIN}：如ilinux.io

例如，在此之前创建的用于交互式Po客户端中尝试请求解析myapp-svc的相关DNS记录：

```
**[terminal]
**[delimiter / #]**[command nslookup myapp-svc.default]
nslookup: can't resolve '(null)': Name does not resolve

Name:      myapp-svc.default
Address 1: 10.233.16.105 myapp-svc.default.svc.cluster.local
```

解析时，“myapp-svc”服务名称的搜索次序依次是default.svc.cluster.local、svc.cluster.local、cluster.local和ilinux.io，因此基于DNS的服务发现不受Service资源所在的名称空间和创建时间的限制。上面的解析结果也正是默认的default名称空间中创建的myapp-svc服务的IP地址。

## 6.4 服务暴露

Service的IP地址尽在集群内可达，然而，总有有些服务需要暴露到外部网络中接受各类客户端的访问，例如分层架构应用中的前端Web应用程序等。此时，就需要在集群的边缘为其添加一层转发机制，以实现将外部请求流量接入到集群的Service资源之上，这种操作也称为发布服务到外部网络中。

## 6.4.1 Service类型

Kubernetes的Service共有四种类型：ClusterIP、NodePort、LoadBalancer和ExternalName。

- ClusterIP：通过集群内部IP地址暴露服务，此地址仅在集群内部可达，而无法被集群外部的客户端访问，如图6-8所示。此为默认的Service类型。
- NodePort：这种类型建立在ClusterIP类型之上，其在每个节点的IP地址的某静态端口（NodePort）暴露服务，因此，它依然会为Service分配集群IP地址，并将此作为NodePort的路由目标。简单来说，NodePort类型就是在工作节点的IP地址上选择一个端口用于将集群外部的用户请求转发至目标Service的ClusterIP和Port，因此，这种类型的Service即可如ClusterIP一样受到集群内部客户端Pod的访问，也会受到集群外部客户端通过套接字进行的请求。

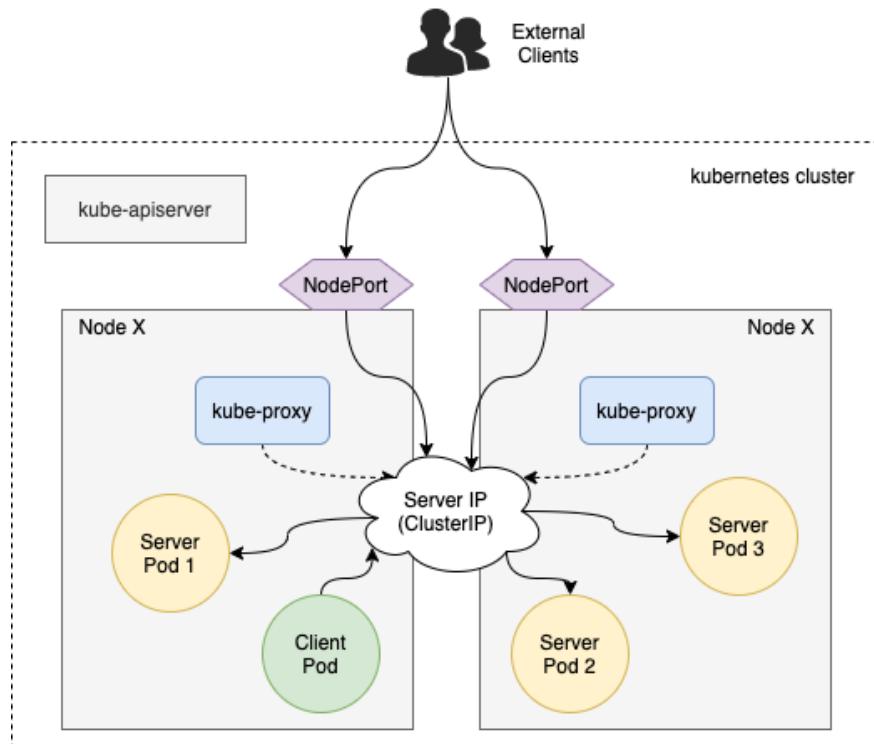


图 1.8.4.1 - NodePort Service 类型

- LoadBalancer：这种类型构建在NodePort类型之上，其通过cloud provider提供的负载均衡器将服务暴露到集群外部，因此，LoadBalancer一样具有NodePort和ClusterIP。简而言之，一个LoadBalancer类型的Service会指向关联至Kubernetes集群外部的、切实存在的某个负载均衡设备，该设备通过工作节点之上的NodePort向集群内部发送请求流量，如图6-9所示。例如Amazon云计算环境中的ELB实例即为此类的负载均衡设备。此类型的优势在于，它能够把来自集群外部客户端的请求调度至所有节点（或部分节点）的NodePort之上，而不是依赖于客户端自行决定连接至哪个节点，从而避免了因客户端指定的节点故障而导致的服务不可用。

### A.1.1 部署目标

- ExternalName：其通过将Service映射至由externalName字段的内容指定的主机名来暴露服务，此主机名需要被DNS服务解析至CNAME类型的记录。换言之，此类型并非定义由Kubernetes集群提供的服务，而是把集群外部的某服务以DNS CNAME记录的方式映射到集群内，从而让集群内的Pod资源能够访问外部的Service的一种实现方式，如图6-10所示。因此，这种类型的Service没有ClusterIP和NodePort，也没有标签选择器用于选择Pod资源，因此也不会有Endpoints存在。

前面章节中创建的 myapp-svc 即为默认的ClusterIP类型Service资源，它仅能接受来自于集群中的Pod对象中的客户端程序的访问请求。如若需要将Service资源发布至网络外部，应该将其配置为NodePort或LoadBalancer类型，而若要把外部的服务发布于集群内供Pod对象使用，则需要定义一个ExternalName类型的Service资源。如若使用kube-dns，那么这种类型的实现将依赖于1.7及其以上版本的Kubernetes版本。

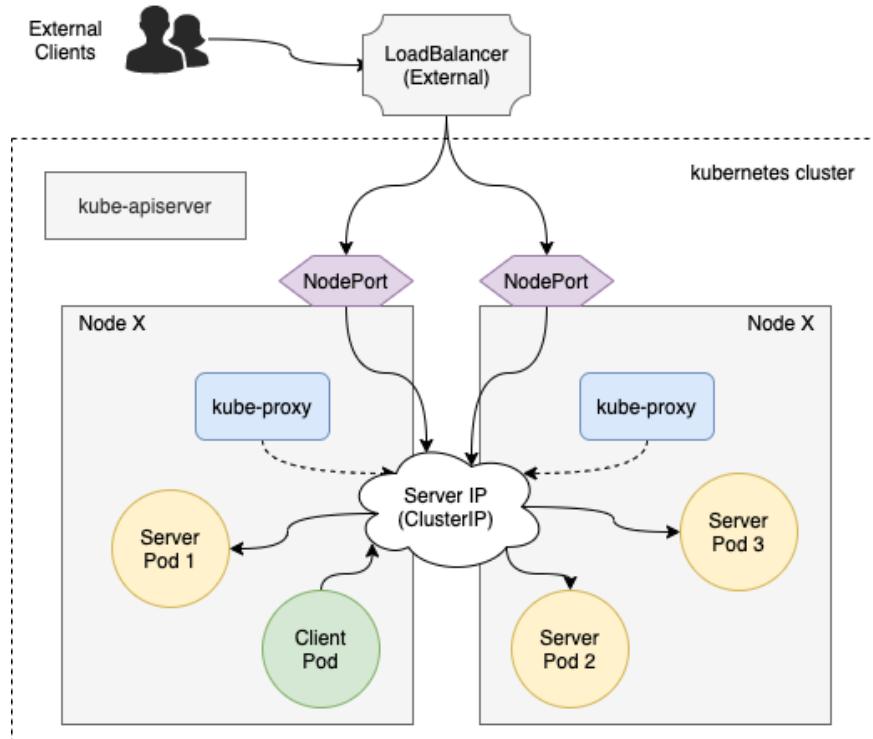


图 1.8.4.1 - LoadBalancer 类型的 Service

### A.1.1 部署目标

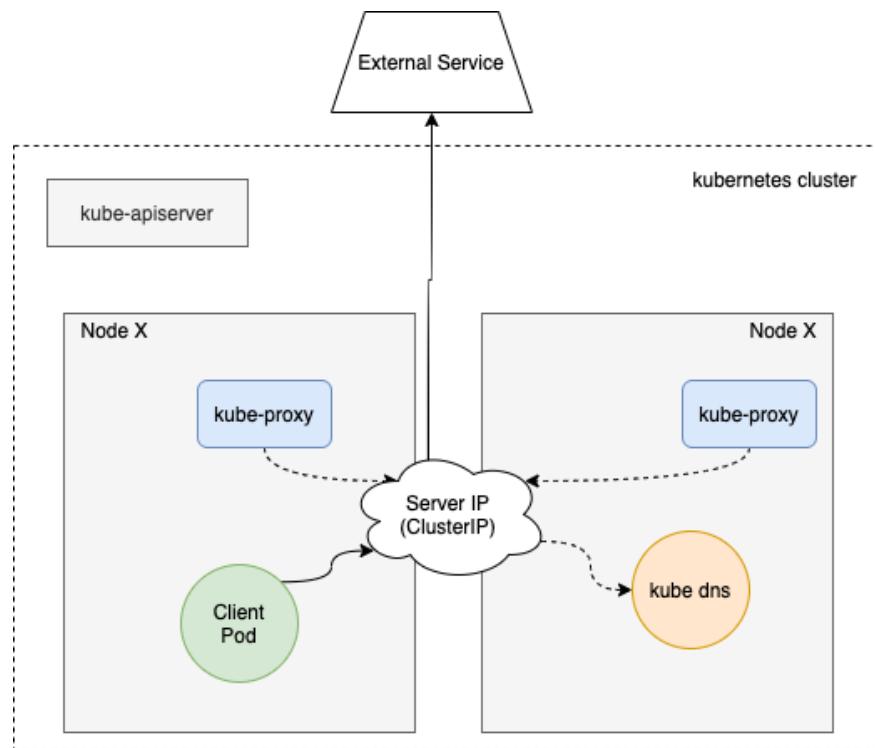


图 1.8.4.1 - `ExternalName`类型的Service

## 6.4.2 NodePort类型的Service资源

NodePort即节点Port，通常在安装部署Kubernetes集群系统时会预留一个端口范围用于NodePort，默认为30000~32767之间的端口。与ClusterIP类型的可省略`.spec.type`属性有所不同的是，定义NodePort类型的Service资源时，需要通过此属性明确指定其类型名称。例如，下面配置清单中定义的Service资源对象`myapp-svc-nodeport`，它使用了NodePort类型，且人为指定其节点端口为32223：

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc-nodeport
spec:
  type: NodePort
  selector:
    app: myapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 32223
```

### 6.4.3 LoadBalancer类型的Service资源

## 6.4.4 ExternalName Service

## 6.5 Headless类型的Service资源

### 6.5.1 创建Headless Service资源

## 6.5.2 Pod资源发现

## 6.6 Ingress资源

## 6.6.1 Ingress和Ingress Controller

## 6.6.2 创建Ingress资源

### 6.6.3 Ingress资源类型

## 6.6.4 部署Ingress控制器（Nginx）

## 6.7 案例：使用Ingress发布tomcat

### 6.7.1 准备名称空间

A.1.1 部署目标

## 6.7.2 部署tomcat实例

### 6.7.3 创建Service资源

## 6.7.4 创建Ingress资源

## 6.7.5 配置TLS Ingress资源

## 6.8 本章小结