

# 一文深入理解 Kubernetes

作者：xixie，腾讯 IEG 后台开发工程师

这篇文章，你要翻很久，建议收藏。

Kubernetes，简称 K8s，是用 8 代替 8 个字符“ubernete”而成的缩写。是一个开源的，用于管理云平台中多个主机上的容器化的应用。k8s 作为学习云原生的入门技术，熟练运用 k8s 就相当于打开了云原生的大门。本文通过笔者阅读书籍整理完成，希望能帮助想学习云原生、以及正在学习云原生的童鞋快速掌握核心要点。学习 k8s 和大家学习 linux 差不多，看似复杂，但掌握了日常熟悉的指令和运行机理就能愉快的使用了，本文的重点和难点是服务、kubernetes 机理部分。

## 要点

Kubernetes 采用的是指令式模型，你不必判断出部署的资源的当前状态，然后向它们发送命令来将资源状态切换到你期望的那样。你需要做的就是告诉 **Kubernetes** 你希望的状态，然后 Kubernetes 会采取相关的必要措施来将集群的状态切换到你期望的样子。

- 1：资源：定义了一份资源，意味着将创建一个对象(如 Pod)或新加了某种规则(类似于打补丁，NetworkPolicy);
- 2：每个种类的 资源都对应一个 **控制器**，负责资源的管理；
- 3：pod 可以看成运行单个应用的 虚拟机，但可能被频繁地自动迁移而无需人工介入；

集群管理和部署的最小单位。

- 无状态服务：新的 IP 名和主机地址
- 有状态服务：StatefulSet, 一致的主机名 和 持久化状态

pod 中应用写入磁盘的数据随时 会丢失 【包括运行时，容器重启，会在新的写入层写入】

记住，pod 是随时可能会被重启

4：容器重启原因：

- 进程崩溃了
- 存活探针返回失败
- 节点内存耗尽，进程被 OOM

需要 Pod 级别的存储卷，和 Pod 同生命周期，防止容器重启丢失数据，例如挂载 emptyDir 卷，看容器启动日志。

容器重启以指数时间避退，直到满 5 分钟。特别注意，容器重启并不影响 Pod 数量变化【理论上所有 Pod 都是一样，即使换新的 Pod，容器还是会重启】。

5：控制 Pod 的启动顺序：

- Init 容器：Pod 中可以有任意多个 initContainers，初始化容器用来控制 Pod 与 Pod 之间 启动 的先后顺序；
- 就绪探针：一个应用依赖于另一个应用，若依赖无法工作，则 阻止 当前应用成为服务 端点群内的 pod 访问。
  - Deployment 滚动升级中会应用 就绪探针，避免错误版本的出现。

6：localhost 一般指代节点，而非 pod；

7: 当 Pod 关闭的时候，可能工作节点上 kube-proxy 还没来得及修改 Iptables，还是会将通信转移到关闭的 Pod 上去；

推荐是 Pod 等一段时间再关闭【等多长时间是个问题， 和 TCP 断开连接等 2MSL 再关闭差不多】，直到 kube-proxy 已修改 路由表。

8: 服务目录可以在 kubernetes 中轻松配置和暴露服务；

9: Kubernetes 可以通过单个 JSON 或 YAML 清单部署 一 组资源；

10: Endpoint, 有站点的意思 (URL) ， REST endpoint, 就是一个 http 请求而已。

Endpoint 资源指 Service 下覆盖的 pod 的 ip:端口列表。

## 整理

### 字段

1: 在定义 manifest 时， 常用的一些字段罗列如下：

备注：常用字段，非全部。

[在线文档](#)

### 标注和必知

1: 常见的注解整列

[在线文档](#)

k8s-标注和必知

### 命名空间和资源

1: k8s 中整理的命名空间和常用资源如下：

[在线文档](#)

k8s-命名空间和资源

### 常用指令

1: 梳理常用指令

[在线文档](#)

kubernetes 指令汇总

## 环境

### 集群安装

1: 单节点集群， minikube

2: 多节点集群， 虚拟机 + kubeadm

# k8s 介绍

## 微服务

1: 微服务: 大量的单体应用 被拆成独立的、小的 组件

2: 配置、管理 需要自动化;

3: 监控应用, 变成了监控 kubernets

传统的应用 由 kubernets 自己去监控

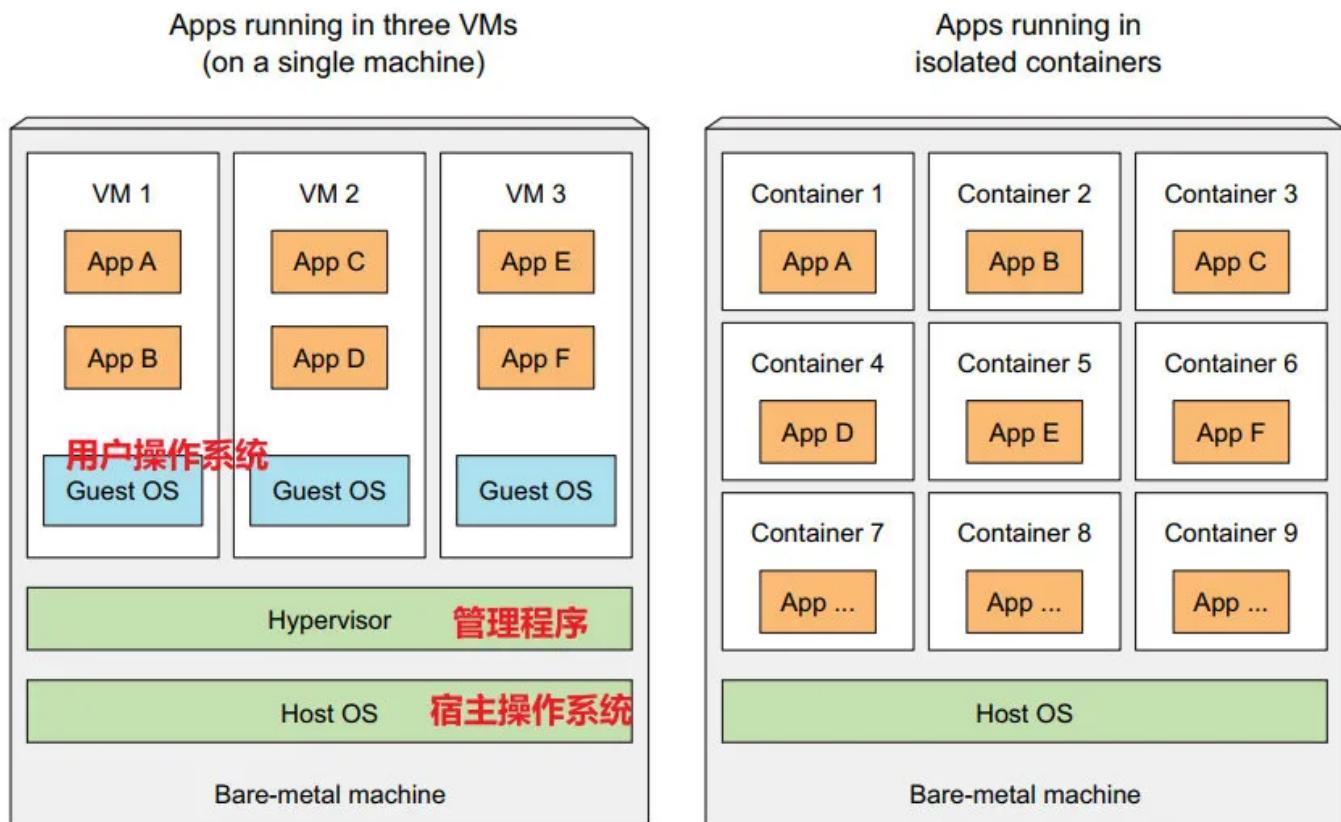
4: 拆成微服务的好处:

- 1: 改动单个服务的 API 成本更小;
- 2: 服务之间可通过 HTTP (同步协议) 、AMQP (异步协议) 通信;
- 3: 新服务可用不同语言开发;

## 虚拟机和容器

1: 虚拟机多出来的三个部分:

- 虚拟化 CPU;
- 用户操作系统;
- 管理程序: 透传虚拟机上应用的操作指令 到 宿主机上的 物理 CPU 来执行;



2: 容器的隔离机制

- Linux 命名空间, 每个进程只能看到自己的系统视图 (文件、进程、网络接口、主机名等)
  - Mount: 挂载卷, 存储;

- PID: process ID, 进程树;
  - Network: 网络接口;
  - Inter-process communication: IPC, 进程间通信;
  - UTS: 本地主机名;
  - User ID: 用户。
- Linux 控制组 (cgroups), 限制进程能使用的资源量 (CPU、内存、网络带宽等)

3: 容器限制了只能使用母机的 Linux 内核;

- X86 上编译的应用容器化后, 不能运行在 ARM 架构的母机上;

## Docker

1: Docker: 打包、分发、运行应用程序的 ==平台==。

运行容器镜像的软件, 类似于 VMware

简化了 Linux 命名空间隔离和 cgroups 之内的系统管理;

- 镜像: 经过 Docker 打包的环境 (包含应用程序的依赖, 配置文件, 运行 app)
- 镜像仓库: 云端存储;
- 容器: 基于 Docker 创建的运行时环境, 是一个运行在 Docker 主机上的进程, 和其他进程隔离且能使用的资源受限;

2: 类似的容器运行时, 还有 rock-it;

## k8s 组成

1: 一个 k8s 分成两类:

- master node (主节点): 主节点上的组件可以组成一个集群, 负责集群的控制和调度
- work node (工作节点) : 工作节点一般是多个, 实际部署应用的节点

2: 组件

- 1: 调度器: 选择资源足够的 node 分配 pod

节点不够, Cluster Autoscaler 横向扩容节点控制 pod 选在哪个节点上 【亲缘性 affinity】 pod 要和哪个 pod 在一块。

- 2: 控制器: 跟踪节点状态, 复制 pod, 持续跟踪 pod, 处理节点失败, 大部分资源都会对应一个控制器  
例如 Endpoint Controller 通知工作节点上的 kube-proxy 修改 iptables。

本质是一个死循环, 监听 API 服务器的状态;

- 3: etcd 分布式数据存储, API 服务器将集群配置存入。比如每次提交一个 yaml 文件, 校验通过后会存入;
- 4: kubelete: 接收 API 服务器通知和 Docker 交互, 控制容器的启动上报 node 的资源总量
- 5: Docker 容器运行时, 拉取镜像、启动容器
- 6: pod:

1: 独立的 IP 和端口空间;

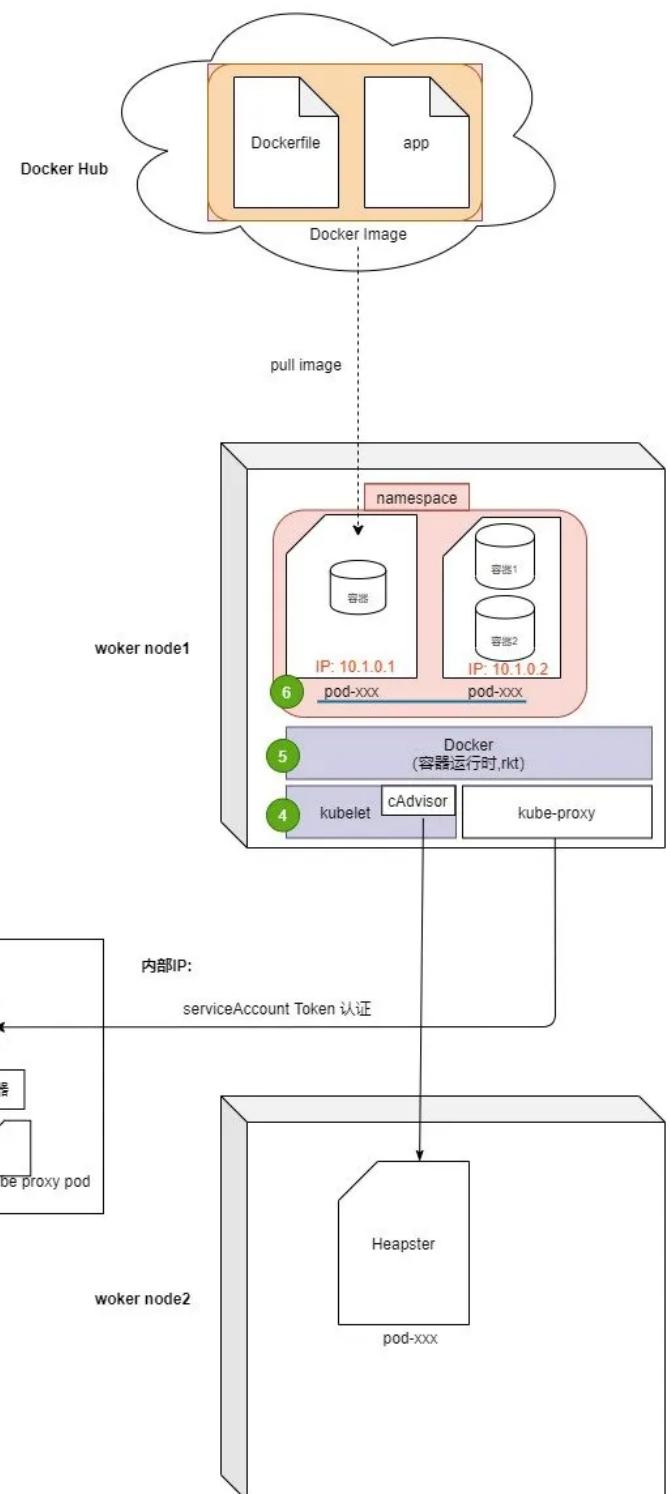
pod.hostNetwork: true, 使用宿主节点的网络接口和端口; containers.ports.hostPort: 仅把容器端口绑定在节点上。

## 2: 独立的进程树，自己的 PID 命名空间

- pod.hostPID: true, 使用宿主节点的 进程空间
- pod.hostIPC: true, 使用宿主节点的 IPC 命名空间

## 3: 自己的 IPC 命名空间，同 pod 可以通过进程间通信 IPC

## 4: 同 pod 两个容器，可以共享存储卷



k8s总体架构图-第 2 页

## 3: k8s 功能

### 1: 自动处理失败容器：重启， 可自动选择新的 工作节点。【自修复】

- 2: 自动调整副本数：根据 CPU 负载、内存消耗、每秒查询 应用程序的其它指标 等； 【==自动扩缩容==】
- 3: 容器 移除或移动， 对外暴露的 静态 IP 不变 （环境变量 or client 通过 DNS 查询 IP） 【==可实现复杂的 集群 leader 选举==】
- 4: 可以限制某些容器镜像 运行 在指定硬件的机器群 【在 HDDS 系列的机器上选择一个】 上：

```
1 | - HDDS;
2 | - SSD;
```

## Docker 和 k8s

### 使用 Docker

#### 安装 Docker

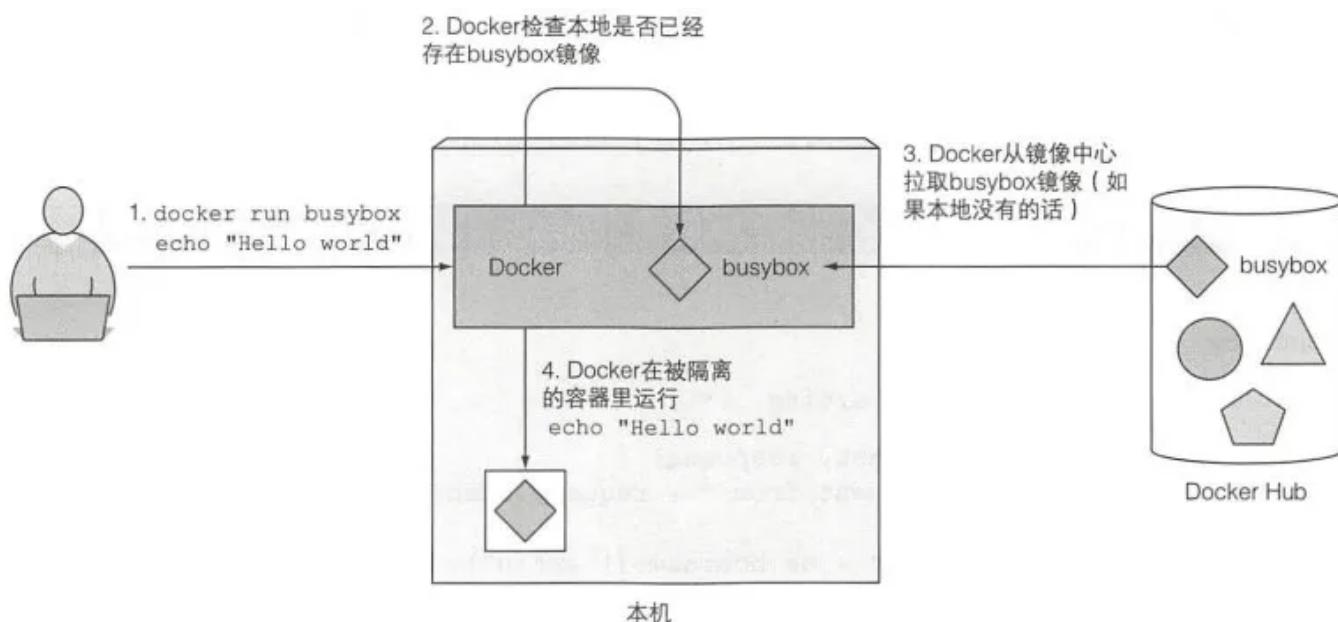
1: 安装：略

2: `docker run <image>` : 现在本地查找镜像，若无，前往 <http://docker.io> 中 pull 镜像。

其它可用镜像：<http://hub.docker.com>

```
1 | # 运行一个镜像，并传入参数
2 | docker run busybox echo "Hello World"
3 | Unable to find image 'busybox:latest' locally
4 | latest: Pulling from library/busybox
5 | b71f96345d44: Pull complete
6 | Digest: sha256:930490f97e5b921535c153e0e7110d251134cc4b72bbb8133c6a5065cc68580d
7 | Status: Downloaded newer image for busybox:latest
8 | Hello World
```

3: ==docker 查看镜像本地是否已存在 -- 下载镜像-- 创建容器 -- 运行 echo 命令--- 进程终止 --- 容器停止运行； ==



## 创建应用

1: 创建一个简单的 `node.js` 应用，输出主机名：

```
1 const http = require('http');
2 const os = require('os');
3
4 console.log("Kubia server starting...");
5
6 var handler = function(request, response) {
7   console.log("Received request from " + request.connection.remoteAddress);
8   response.writeHead(200);
9   response.end("You've hit " + os.hostname() + "\n");
10 }
11
12 var www = http.createServer(handler);
13 www.listen(8080);
```

2: 构建镜像，需要一个 Dockerfile 文件

和 `app.js` 同一目录

```
1 # From 定义了基础镜像，可以是 Ubuntu 等系统，但应尽量遵从精简
2 FROM node:7
3 # 本地文件添加到 镜像的根目录
4 ADD app.js /app.js
5 # 镜像被运行时 需要执行的命令
6 ENTRYPOINT [ "node", "app.js" ]
```

## 构建镜像

1: Dockerfile + `app.js` 就能创建镜像包：

```
1 # 基于当前目录 创建 kubia 镜像
2 $ docker build -t kubia .
3 Sending build context to Docker daemon 3.072kB
4 Step 1/3 : FROM node:7
5 7: Pulling from library/node
6 ad74af05f5a2: Pull complete
7 2b032b8bbe8b: Pull complete
8 a9a5b35f6ead: Pull complete
9 3245b5a1c52c: Pull complete
10 afa075743392: Pull complete
11 9fb9f21641cd: Pull complete
12 3f40ad2666bc: Pull complete
13 49c0ed396b49: Pull complete
14 Digest: sha256:af5c2c6ac8bc3fa372ac031ef60c45a285eeba7bce9ee9ed66dad3a01e29ab8d
15 Status: Downloaded newer image for node:7
16    --> d9aed20b68a4
```

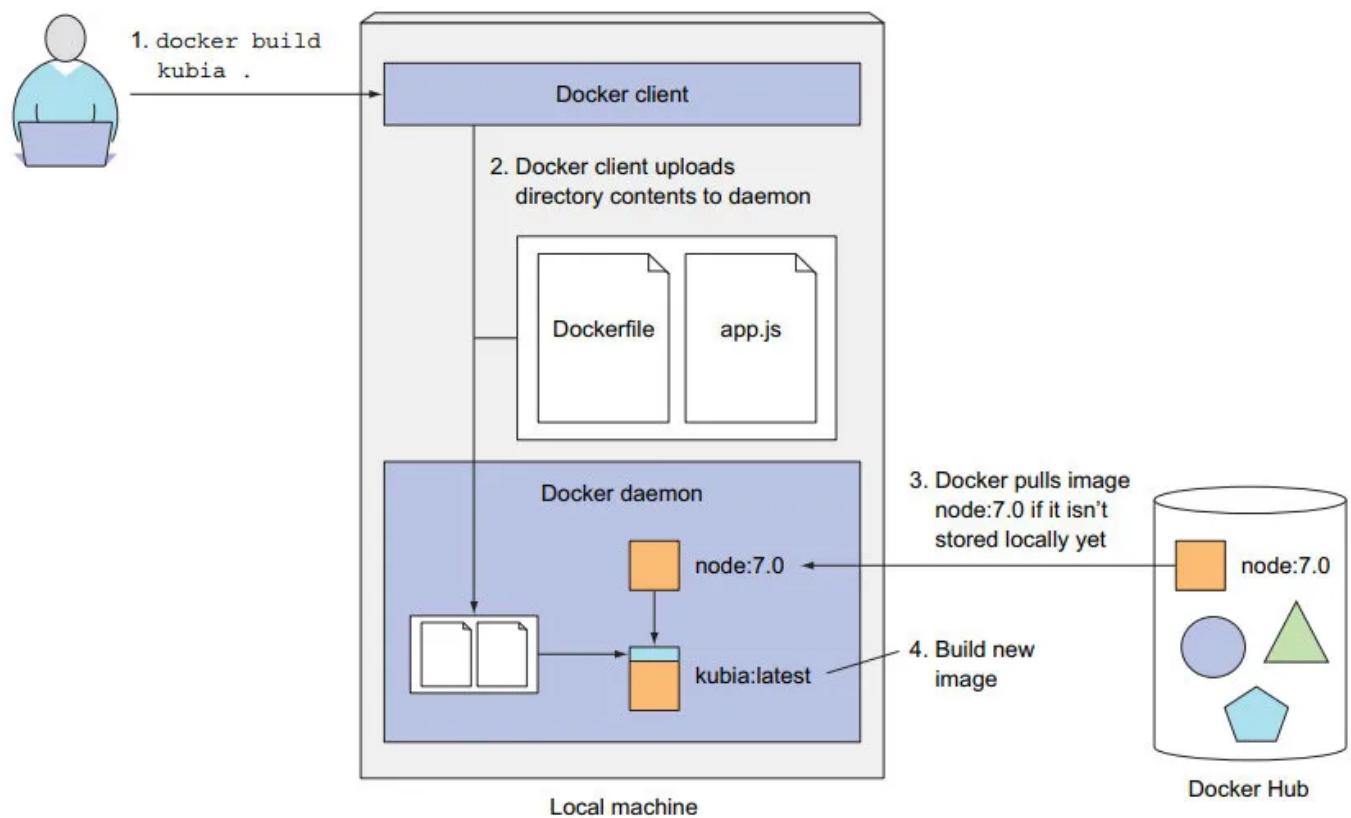
```

17 Step 2/3 : ADD app.js /app.js
18    --> 43461e2e8cef
19 Step 3/3 : ENTRYPOINT [ "node" , "app.js" ]
20    --> Running in 56bc0e5982ce
21 Removing intermediate container 56bc0e5982ce
22    --> 0d2c12c8cc80
23 Successfully built 0d2c12c8cc80
24 Successfully tagged kubia:latest

```

## 2: 镜像构建过程

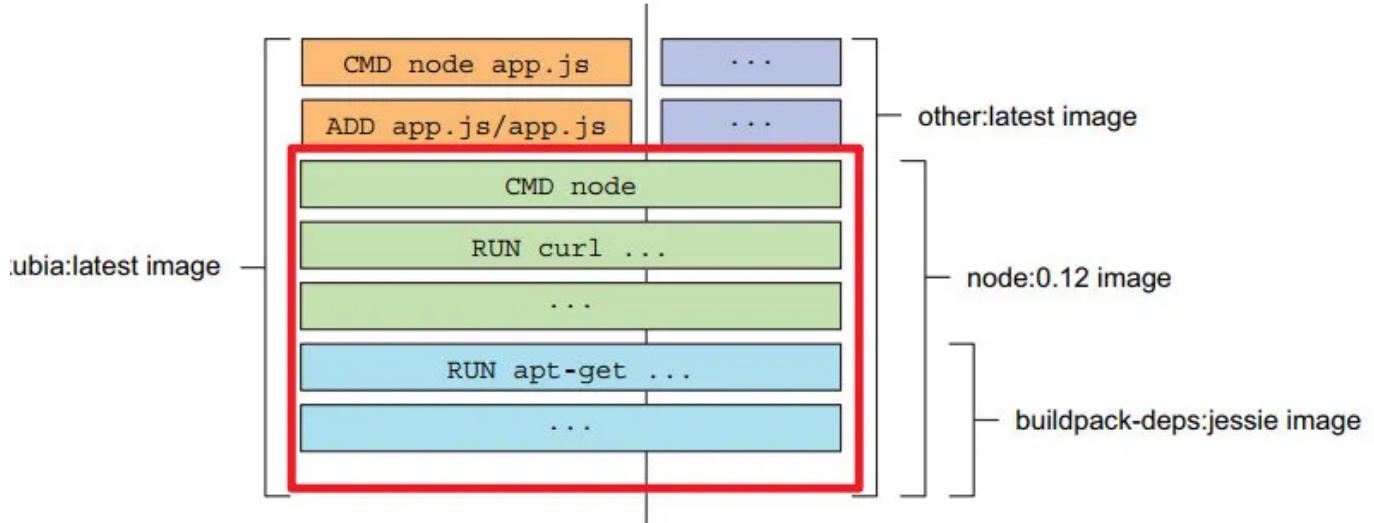
- Docker 客户端：在宿主机上的；
- Docker 守护进程：运行在一个虚拟机内；【可以在远端机器】



## 镜像分层

1: Dockerfile 中的每一行都会形成一个镜像层

最后一层是 `kubia:latest` 镜像



2: 镜像分层的好处: 节省下载,

3: 查看本地新镜像 `docker image`

## 运行镜像

1: 端口映射后, 可用 <http://localhost:8080> 访问;

```

1          # 容器名称      # 本机8080 映射到容器的 8080 端口      # 镜像文件
2 docker run --name kubia-container -p 8080:8080                  -d kubia
3 4099df6236c5d4905a268b213ab986949f6522122454de41f56293ce3508e958 # 容器 ID
4
5
6 # test, 主机名就是 分配的 容器 ID
7 $ curl localhost:8080
8 You've hit 4099df6236c5

```

2: 查看运行中的容器:

```

1 $ docker ps
2 CONTAINER ID   IMAGE     COMMAND      CREATED      STATUS      PORTS      NAMES
3 4099df6236c5   kubia     "node app.js"  40 seconds ago   Up 38 seconds   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   kubia-container
4 2744c31527b9   gcr.io/k8s-minikube/kicbase:v0.0.22   "/usr/local/bin/entr..."   3 days ago   Up 3 days    127.0.0.1:49172->22/tcp, 127.0.0.1:49171->2376/tcp,
5                                         127.0.0.1:49170->5000/tcp, 127.0.0.1:49169->8443/tcp, 127.0.0.1:49168->32443/tcp   minikube
6
7 # 查看容器 详细信息
8 docker inspect kubia-container

```

## 查看容器内部

1: 一个容器可以运行多个进程;

2: Node.js 镜像中包含了 bash shell, 可在容器内运行 shell:

```
1 # kubia-container 容器内 运行 bash 进程
2 docker exec -it kubia-container bash
3
4 # 退出
5 exit
```

- `-i`, 确保标准输入流保持开放。需要在 shell 中输入命令。

- `-t`, 分配一个伪终端(TTY)。

3: 查看容器内的进程;

```
1 $ docker exec -it kubia-container bash
2 root@4099df6236c5:/# ps axuf
3 USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
4 root           12  0.6  0.0  20244  2984 pts/0      Ss  21:44   0:00 bash
5 root           18  0.0  0.0  17496  2020 pts/0      R+  21:44   0:00 \_ ps axuf
6 root            1  0.0  0.6 614432 26320 ?         Ssl  21:41   0:00 node app.js
```

只能在 docker 的守护进程中查看

4: 在母机上查看进程

```
1 user00@ubuntu:~$ ps axuf | grep app.js
2 app.js
3 root      3224898  0.0  0.6 614432 26320 ?         Ssl  14:41   0:00 \_ node app.js
```

容器和主机上的进程 ID 是独立、不同的;

5: 容器是独立的:

- PID Linux 命名空间
- 文件系统都是独立的
- 进程
- 用户
- 主机名
- 网络接口

## 停止和删除容器

1: 停止容器, 会停止容器内的主进程。

容器本身依然存在, 通过 `docker ps -a`

```
1 docker stop kubia-container
2
3 # 打印所有容器，包括 运行中的和已停止的
4 $ docker ps -a
5 CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
6 NAMES
7 4099df6236c5 kubia "node app.js" 6 minutes ago Exited (137) 4 seconds ago
8
9 # 真正的 删除容器
10 docker rm kubia-container
```

## 向仓库推送镜像

1: 仓库 <http://hub.docker.com> 镜像中心；

2: 按照仓库要求，创建额外的 tag

```
1 # 个人 ID
2 docker tag kubia luksa/kubia
3
4 # 查看镜像
5 docker images | head
6
7 # 自己的 ID 登录
8 docker login
9
10 # 推送镜像
11 docker push luksa/kubia
```

3: 可在任意机器上运行 云端 docker 镜像：

```
1 docker run -p 8080:8080 -d luksa/kubia
```

## 使用 kubernetes 集群

### 安装集群的方式

1: 安装集群通常有以下四种方式：

- 本地 单点；
- Google Kubernetes Engine(GKE) 上托管的集群；
- kubeadm 工具安装；
- 亚马逊的 AWS 上安装 kubernetes

## Minikube 启动 k8s 集群

1: 安装略

2: 启动 Minikube 虚拟机

```
1 | minikube start
```

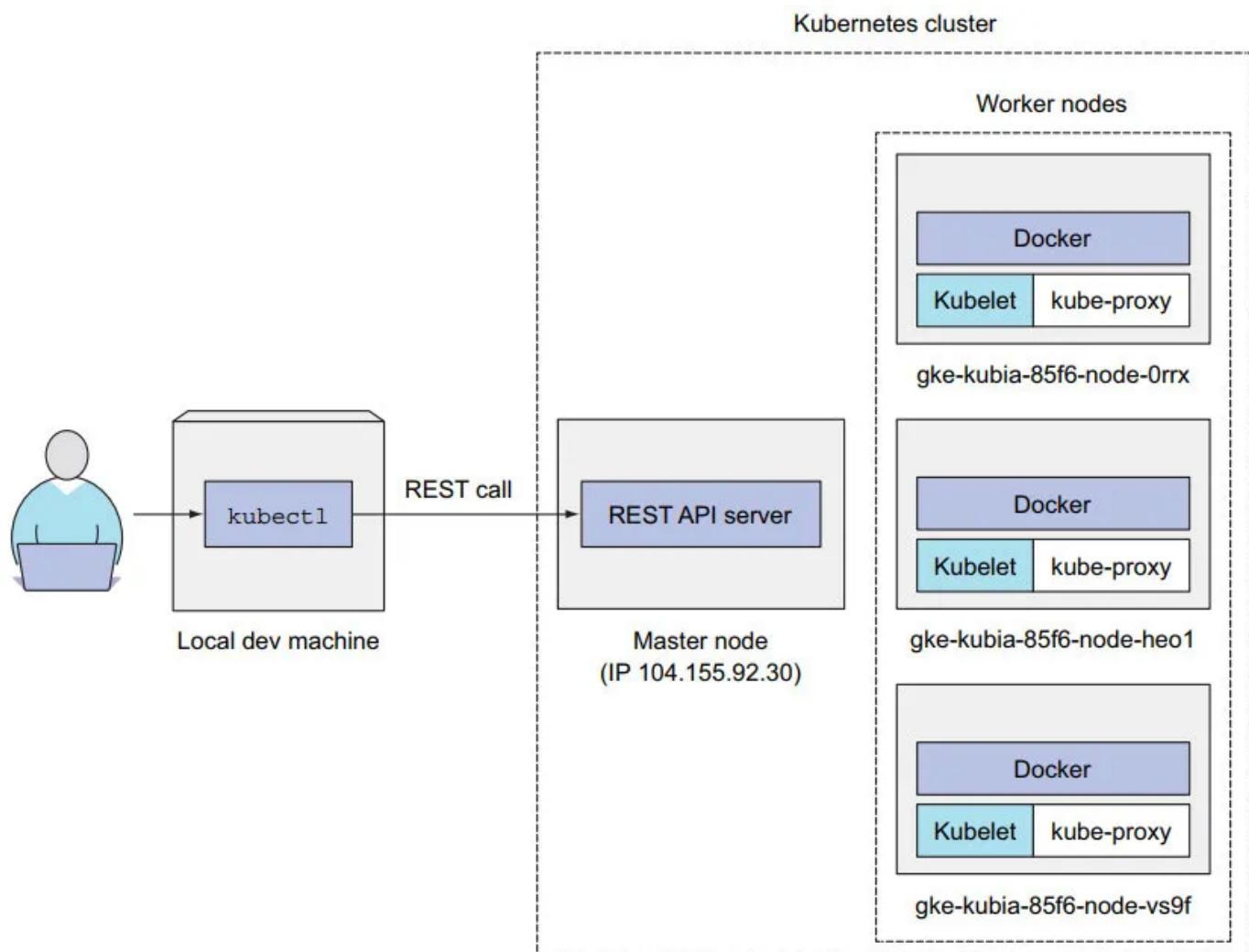
3: 安装 k8s 客户端(kubectl)

## GKE 创建三节点集群

1: 创建 3 个工作节点的示例:

```
1 | gcloud container clusters create kubia --num-nodes 3 --machine-type f1-micro
```

2: 本地发起请求 到 k8s 的 master 节点; master 节点负责 调度 pod, 以下创建了 3 个工作节点;



## 部署 Node.js 应用

```
1 | kubectl run kubia --image=luksa/kubia --port=8080 --generator=run/v1
```

### pod

1: 多个容器运行在一起：

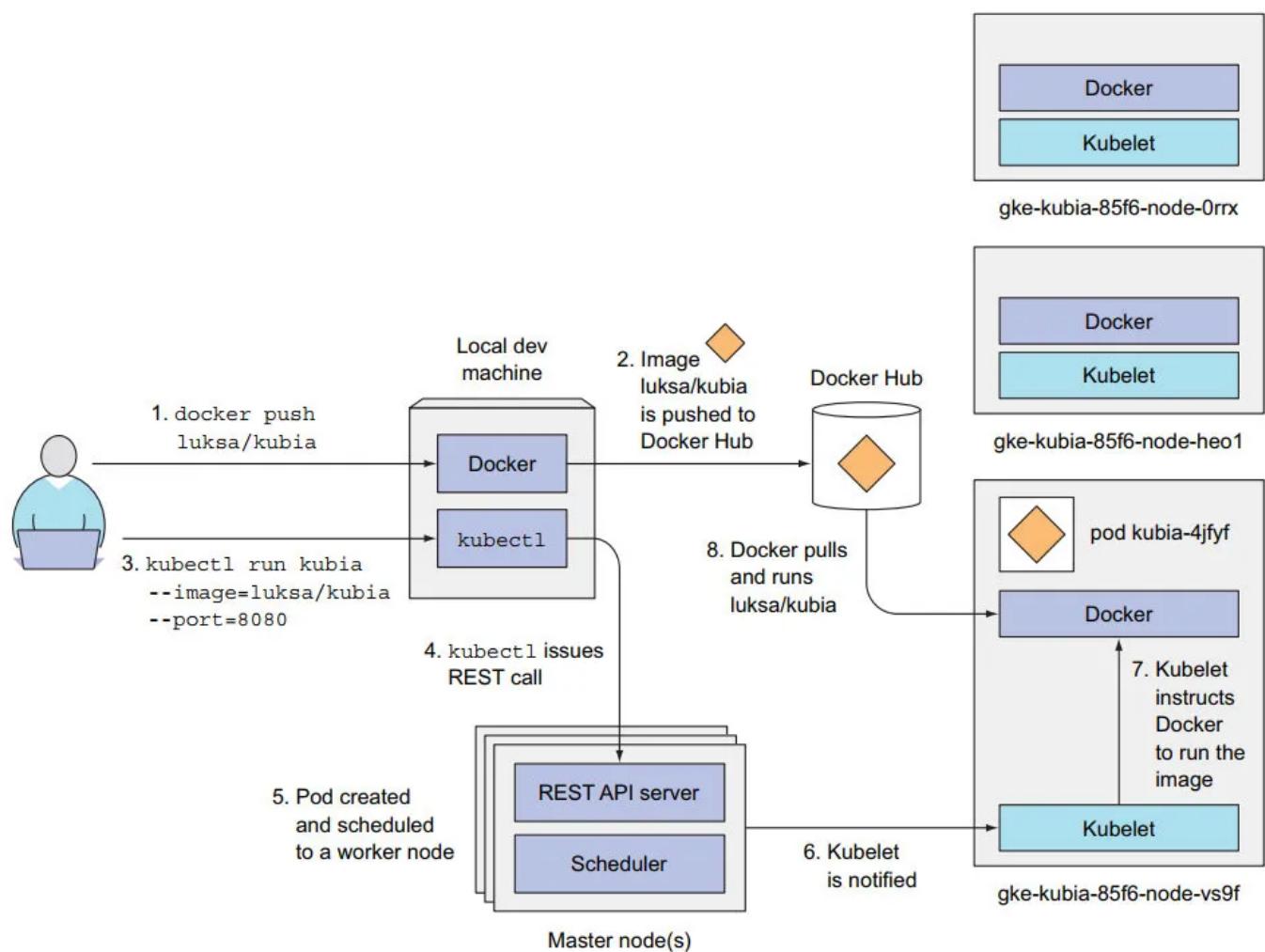
一个 pod 可以包含任意数量的容器；

2: pod 拥有单独的 **私有 IP**、主机名、单独的 Linux 命名空间；

**==Pod 是扩缩容的基本单位； ==**

3: k8s 中运行容器镜像需要经历两个步骤：

- 1: 推送 docker 镜像 到云端 【不同工作节点上的 Docker 能访问到 该镜像】；
- 2: 运行 kubectl , ==创建一个 ReplicationController 对象 【运行指定数量的 pod 副本】； ==
  - 调度器 创建 pod, 并 选择一个 工作节点, 分配给 pod;



## 创建外部访问的服务

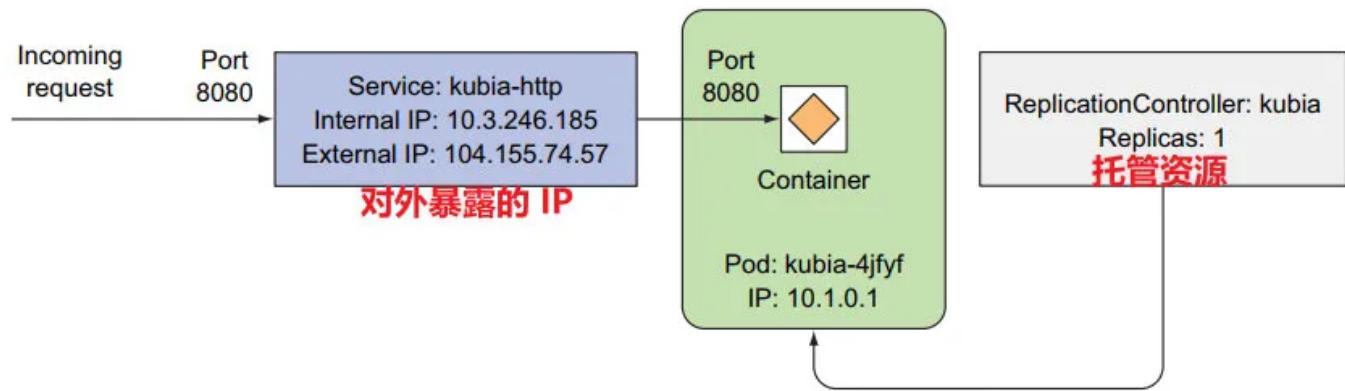
- 1: 常规服务: ClusterIP 服务, 比如 pod, 只能从 集群内部访问;
- 2: 创建 LoadBalancer 类型的服务, 负载均衡, 对外提供 公共 IP 访问 pod;  
内部删减 pod, 移动 pod, 外部 IP 不变, 一个外部 IP 可对应 多个 pod。

```
1 | kubectl expose rc kubia --type=LoadBalancer --name kubia-http  
2 |  
3 | kubectl get svc # 查看服务是否分配 EXTERNAL-IP (外部IP)
```

`minikube service kubia-http` 可查看 IP: 端口

## 系统逻辑部分

- 1: 服务、pod、对象:



## 水平伸缩 pod 节点

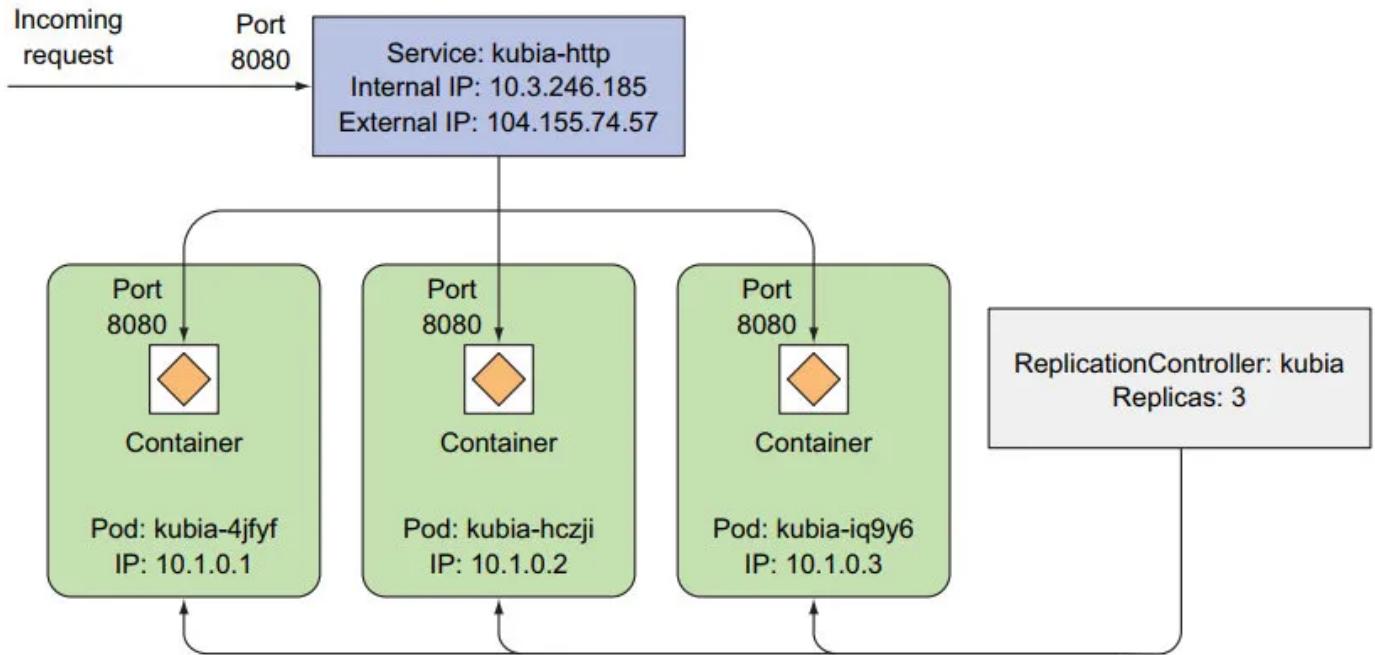
```
1 | kubectl get replicationcontrollers # 获取 ReplicationControllers 状态
```

告诉期望数量即可:

```
1 | kubectl scale rc kubia --replicas=3
```

多次访问, 服务作为负载均衡, 会随机选择一个 pod:

和上面对比, 这里有三个 pod 实例。



## pod 运行在哪个节点上

1: pod 的 IP 和运行的工作节点

image-20210617220307330

```

1 $ kubectl describe pod kubia-jppwd
2 Name:           kubia-jppwd
3 Namespace:      default
4 Priority:       0
5 Node:          minikube/192.168.49.2 # 具体的节点

```

## Dashboard

1: 若是 GKE 集群, 可通过 `kubectl cluster-info | grep dashboard` 获取 dashboard 的 URL。

2: 终端输入 `minikube dashboard` 会自动打开浏览器:

## pod

### Pod

1: 容器被设计为每个容器只运行一个进程，保证轻量和一定的隔离性；

2: 有些容器又有一些紧密的联系，例如常说的 side-car 容器，负责网络代理，日志采集的容器，这些容器最好放一起。这就出现了上层的 pod。

pod 是 k8s 中引入的概念，docker 是可以直接运行容器的。

3: k8s 通过配置 Docker 让一个 pod 内的所有容器 共享 相同的 Linux 命名空间 【有些容器放到一个 pod 的好处】：

- 相同的 network 和 UTS 命名空间；
- 共享相同的主机名和网络接口；pod 中的端口，不能绑定多次；
  - 两个 pod 之间可以实现 两个 IP 相互访问

不管两个 pod 是否在同一节点，可以想无 NAT 的平坦网络之间通信（类似局域网 LAN）

- 相同的 IPC 命名空间下运行；能通过 IPC 进行通信；
- 共享相同的 PID 命名空间

注意：文件系统来自容器镜像，默认容器的文件系统彼此隔离。

4: pod 是逻辑主机，其行为与非容器世界中的物理主机或虚拟机非常相似。此外，运行在同一个 pod 中的进程与运行在同一物理机或虚拟机上的进程相似，只是每个进程都封装在一个容器之中。

5: pod 可以当做独立的机器，非常轻量，可同时有大量的 pod；

6: pod 是扩缩容的基本单位；

7: pod 的定义包含三个部分：

- metadata 包括名称、命名空间、标签和关于该容器的其他信息。
- spec 包含 pod 内容的实际说明，例如 pod 的容器、卷和其他数据。
- status 包含运行中的 pod 的当前信息，例如 pod 所处的条件、每个容器的描述和状态，以及内部 IP 和其他基本信息。

一个简单 pod 包含的三部分：

```
1 apiVersion: v1    # 分组和版本
2 kind: Pod          # 资源类型
3 metadata:
4   name: kubia-manual  # Pod 名称
5 spec:
6   containers:
7     - image: luksa/kubia  # 容器使用的镜像
8       name: kubia
9       ports:
10      - containerPort: 8080 # 应用监听的端口
11        protocol: TCP
```

8：可使用 `kubectl explain` 查看指定资源信息

```
1 | kubectl explain pods
```

9：创建资源：

```
1 | # 所有定义的 资源 manifest 都通过该指令来创建，非常重要
2 | kubectl create -f kubia-manual.yaml
```

10：查看日志：

```
1 | kubectl logs <pod-name>
```

11：若不想通过 Service 与 pod 通信，可通过端口转发：

```
1 | # 将 8888 端口 转发到 该 pod的 8080 端口
2 | kubectl port-forward <pod-name> 8888:8080
3 |
4 | curl localhost:8888
```

端口转发是 `kubectl` 内部实现的。

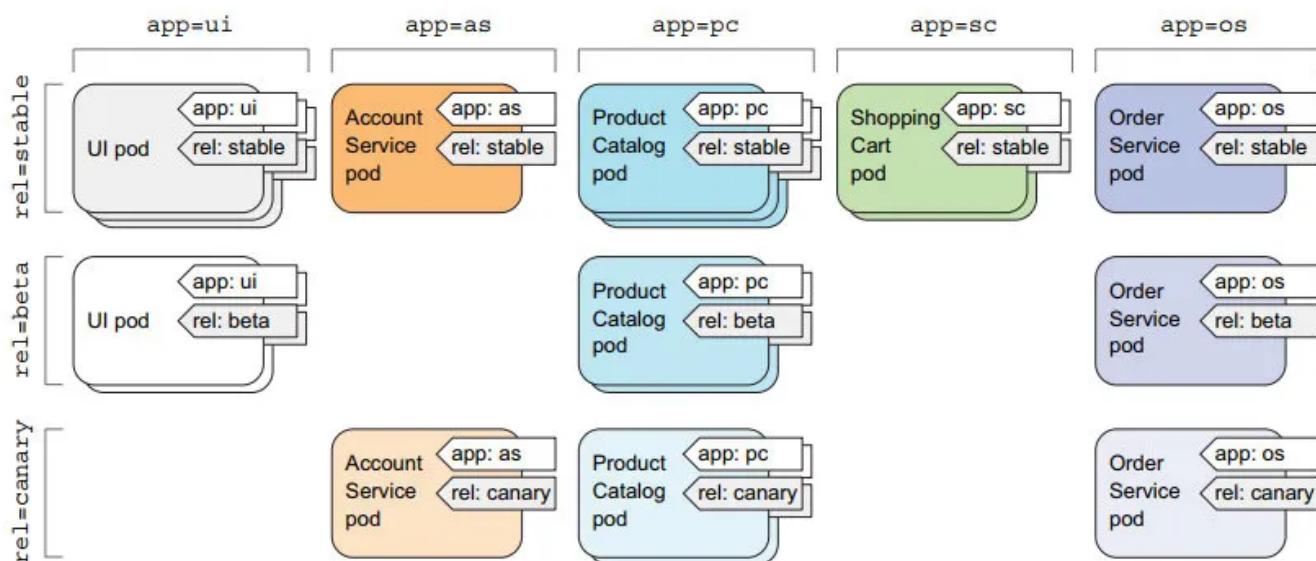
## 标签

1: 可使用 标签组织管理 pod

  | 标签也能组织其他 k8s 资源

2: 例如定义两组标签，可不同维度管理 pod

- `label_key = app`: 按应用分类
- `label_key = rel`: 按版本分类



3: pod 中使用 labels 定义标签：

```
1 kind: Pod
2 metadata:
3   name: kubia-manual-v2
4   labels:
5     # 定义的两组标签
6     creation_method: manual
7   env: prod
```

4: 可通过 `-l` 指定标签选择列出 pod 子集：

```
1 $ kubectl get po -l creation_method=manual
2 NAME           READY   STATUS    RESTARTS   AGE
3 kubia-manual-v2   1/1     Running   0          31s
```

  | 使用多个标签，中间用逗号分隔即可。

## pod 调度到特定节点

- 1: 默认下， pod 调度到哪个 节点是不确定的， 这由调度器决定。
- 2: 有些情况， 需要将 pod 调度到特定的节点上（比如偏计算的 pod 需要调度到 gpu 集群上）
- 3: 给节点打标签

```
1 | kubectl label node <node-name> <label-key>=<label_value>
```

- 4: 通过 `nodeSelector` 调度到含有 `gpu=true` 标签的节点上。

```
1 | apiVersion: v1
2 | kind: Pod
3 | metadata:
4 |   name: kubia-gpu
5 | spec:
6 |   # 调度到含有 gpu=true 的节点上
7 |   nodeSelector:
8 |     gpu: "true"
9 |   containers:
10 |   - image: luksa/kubia
11 |     name: kubia
```

注意：含有该标签的节点可能有多个，届时将选择其中一个。

候选节点最好是一个集合，避免单个节点故障会造成服务不可用。

## 注解

- 1: 注解 (Annotation) 和 标签类似，也是键值对。

有些注解是 k8s 自动添加的。

注解不能超过 256K

## 命名空间

- 1: 命名空间 (namespace, 简称 ns) 可对对象分组。
    - 资源名称只需要在命名空间内保证唯一即可，跨命名空间可以重。
  - 2: 列出某个命名空间下的 pod
- ```
1 | kubectl get po --namespace kube-system
```
- 3: 命名空间，可控制用户在该命名空间的访问权限，限制单个用户可用的 资源数量；
  - 4: 创建命名空间：

```
1 apiVersion: v1
2 kind: Namespace
3 metadata:
4   name: custom-namespace
```

5: 尽管命名空间对对象进行了分组，但 ==并不提供实质上的隔离==，例如不同命名空间的 pod 能否通信取决于网络策略。

6: 删除 pod 有多种方式：

```
1 kubectl delete po <pod-name> # 按名称删除
2 kubectl delete po --all          # 删除当前命名空间的所有 pod, 若 ReplicationController 未删除, 将重新创建 pods
3 kubectl delete po -l <label-key>=<label-val> # 按标签删除
4
5 kubectl delete ns <ns-anme> # 删除整个命名空间
6
7 kubectl delete all --all      # 删除当前命名空间内的所有资源, 包括托管的
     ReplicationController, Service, 但 Secret 会保留
```

## 托管集群

### 保持进程健康

1: 进程异常的几种情形：

- 主进程崩溃->kubelet 将重启容器；
- 内存泄漏，JVM 会一直运行，但会抛出 OutofMemoryErrors，让程序向 k8s 发出信号触发重启；
- 外部检查：应用死循环 or 死锁

### 存活探针

1: 定期检查容器

2: 三种探测机制：

- HTTP Get 向容器发送请求；
- TCP 套接字，与容器建立 TCP 连接；
- Exec 探针，在容器内执行任意指令，查看退出状态码；

3: HTTP 探针，定期发送 http Get 请求；

/heath HTTP 站点不需要认证，否则会一直认为失败，容器无限重启；

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: kubia-liveness
5 spec:
6   containers:
7     # 镜像内有坏掉的应用
```

```
8   - image: luksa/kubia-unhealthy
9     name: kubia
10    # 存活探针
11    livenessProbe:
12      httpGet:
13        path: /
14        port: 8080
```

4: 返回的状态码 137 和 143:

```
1 $ kubectl describe pod kubia-liveness
2       State:          Running
3         Started:      Thu, 17 Jun 2021 11:04:53 -0700
4       Last State:    Terminated
5         Reason:       Error
6       Exit Code:     137    # 有错误码返回
7
8   Warning  Unhealthy  10s    kubelet           Liveness probe failed: HTTP probe
   failed with statuscode: 500
```

5: 探针的附加信息:

查看状态时, 可看到 存活探针信息:

```
1 Liveness:          http-get http://:8080/ delay=0s timeout=1s period=10s #success=1
  #failure=3
```

- `delay=0s`: 容器启动后立即检测;
- `timeout=1s`: 限制容器在 1s 内响应, 否则失败;
- `period=10s`: 每隔 10s 探测一次;
- `failure=3`: 连续三次失败后, 重启容器;

6: 具有初始延迟的 存活探针: 【程序还未 启动稳定】

```
1 # 第一次探针前, 等15s, 防止容器没准备好
2     initialDelaySeconds: 15
```

7: 若无探针, k8s 认为进程还在运行, 容器就是健康的。

8: 探针的注意事项:

- 1: 探针应该轻量, 不能占用太多 cpu 【应计入容器的 CPU 配额】, 一般 1s 内执行完;
- 2: java 程序应该用 http get 探针, 而非启动全新 JVM 获取存活信息的 Exec 探针 (太耗时)
- 3: 无需设置 探针的失败重试次数, k8s 为了确认一次探测的失败, ==默认就会尝试若干次==;

9: 重启容器由 kubelet 执行; 主服务器上的 k8s Control Plane 组件不会参与;

- ==若整个节点崩溃, 则无法重启== 【kubelet 依赖于节点】;
- 若要保证节点挂了, pod 能重启, 应该使用 RC 或 RS;

## ReplicationController

- 1: 用于管理 pod 的多个副本;
- 2: 会自动调整 pod 数量为 指定数量:

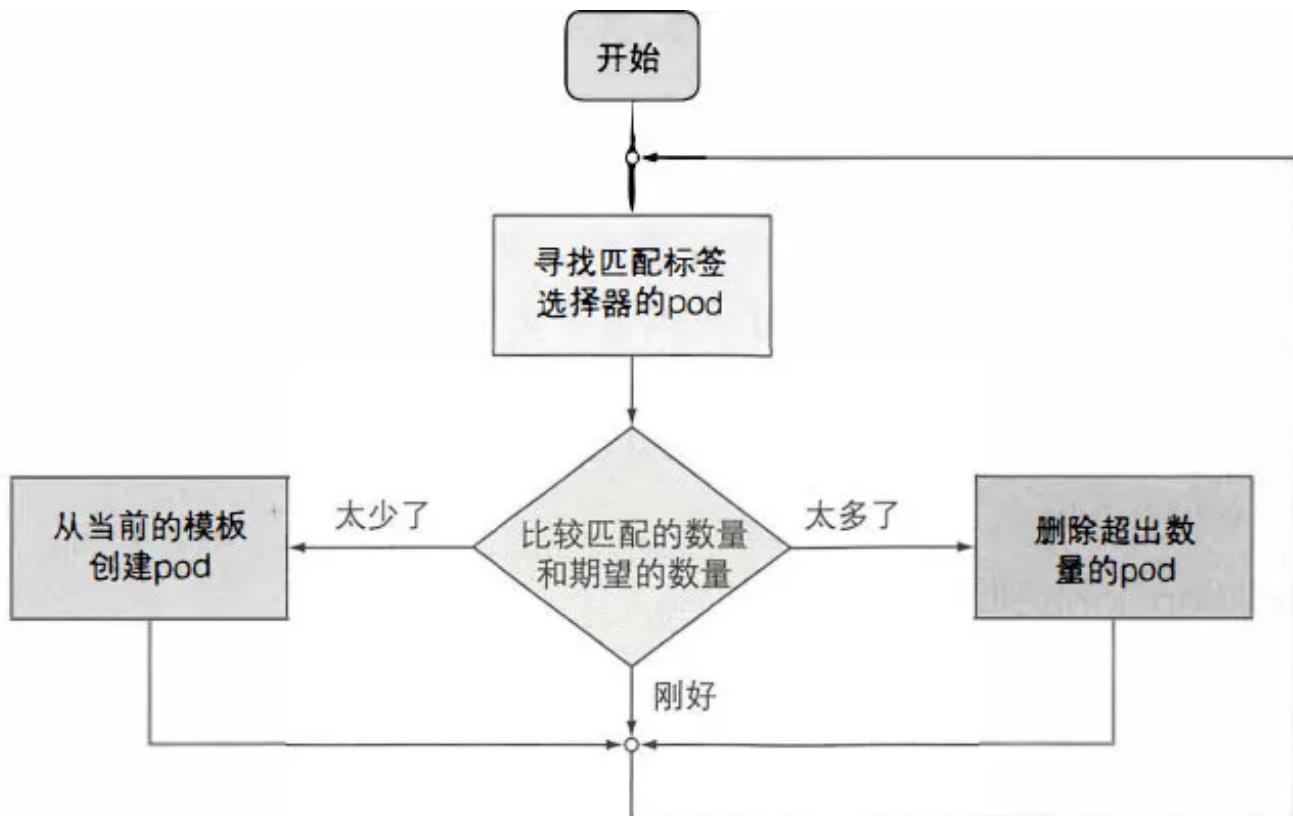
多余副本在以下几种情况下会出现:

- 有人会手动创建相同类型的 pod。
- 有人更改现有的 pod 的 "类型"。
- 有人减少了所需的 pod 的数量, 等等。

- 3: ReplicationController 的功能:

- 确保一个 pod (或多个 pod 副本) 持续运行, 方法是在现有 pod 丢失时启动一个新 pod。
- 集群节点发生故障时, 它将为故障节点上运行的所有 pod (即受 ReplicationController 控制的节点上的那些 pod) 创建替代副本。
- 它能轻松实现 pod 的水平伸缩: 手动和自动都可以

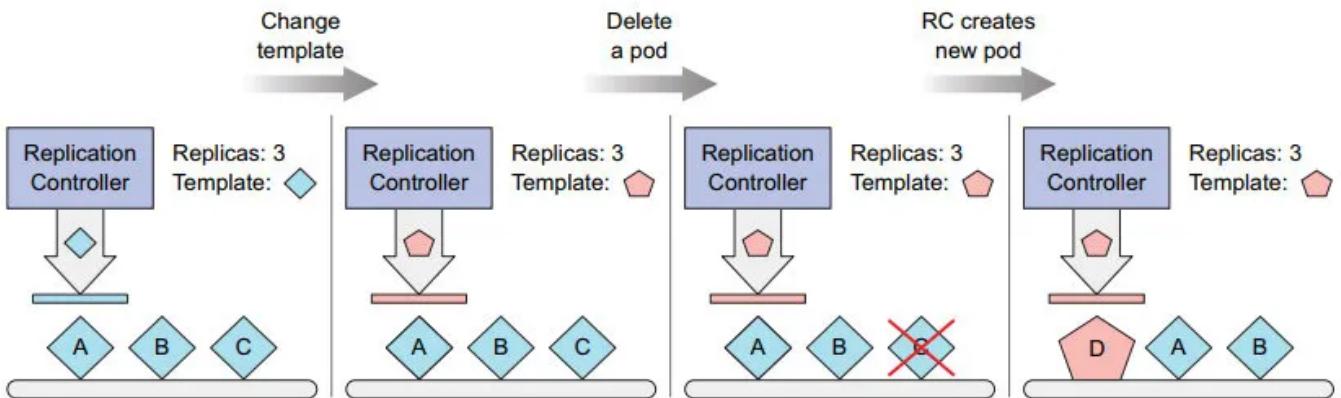
- 4: 根据 pod 是否匹配 标签选择器 来调整:



- 5: 更改标签选择器 和 pod 模板, 对当前的 pod 没有影响;

- 也不关心 容器镜像、环境变量和 其它;
- 只影响 创建新的 pod (新的 曲奇 切模 cookie cutter )

修改 pod 模板:



更改副本个数，就能实现动态扩缩容：

```
1 | kubectl scale rc kubia --replicas=3 # 调整副本数为3
```

6: 创建对象：

上传到 API 服务器，会创建 kubia 的 ReplicationController 【简称 RC】。

- 模板中的 pod 标签必须与 RC 一致，否则会无休止创建容器（达不到期望数量的 pod）
- API 服务会校验 RC 的定义，不会接受错误配置；
- 可以不指定 RC 的选择器，会自动根据 pod 模板中的标签自动设置；

```
1 apiVersion: v1
2 kind: ReplicationController
3 metadata:
4   name: kubia
5 spec:
6   replicas: 3  # pod 副本数量
7
8   # 标签选择器，选择 标签 app = kubia的pod进行管理
9   selector:
10    app: kubia
11
12   # pod 模板
13   template:
14     metadata:
15       labels:
16         app: kubia
17     spec:
18       containers:
19         - name: kubia
20           image: luksa/kubia
21           ports:
22             - containerPort: 8080
```

7: 删除 pod 标签（或者移入另一个 RC 的掌控），脱离 RC 掌控，RC 会自动起一个新的 pod;

- 原 pod 可用于调试，完成后，手动删除该 pod 即可；

- 实际 pod 数量， RC 通过就绪探针；
- 删除 pod， 允许客户端监听到 API 服务器的通知，通过检查实际的 pod 数量采取适当的措施；

```
1 | kubectl delete pod <pod-name>
```

添加 pod 另外的标签， RC 并不 care；

8：通过 pod 的 `metadata.ownerReferences` 可以知道该 pod 属于哪个 RC；

9：节点故障：例如网络断开；

- RC 一段时间后检测到 pod 关闭（旧节点变为 `unknown`），会启动新的 pod 代替原来的 pod；
- 当旧节点恢复时，节点状态变为 `ready, unknown` pod 会被删除；

10：更改 RC 的 标签选择器：

- 原有 pod 都会脱离管控；
- RC 会创建（若无）新的指定数量、指定标签的 pod

**==RC 的标签选择器可以修改，但其他的 控制器对象 不能。==**

11：删除 RC， 默认会删除 RC 管理的 pod

可以使用 选项 `--cascade=false` 保留 pod.

```
1 | kubectl delete rc <rc-name> --cascade=false
```

## 使用 ReplicaSet 替换 ReplicationController

1：ReplicationController 最初是 用于赋值和异常时重新调度节点 的唯一 组件。

2：一般不会直接创建 ReplicaSet， 而是 创建 更高层次的 Deployment 资源时（第 9 章） 自动创建他们。

3：ReplicaSet 功能和 ReplicationController 一样， pod 选择器的 表达能力更强：

- ReplicationController 只允许 `k` 和 `v` 同时 匹配的标签；
- ReplicationController 只能匹配单个 `kv`；
- ReplicaSet 基于 标签名 `k` 匹配；

4：若已经有了 3 个 pod， 不会创建任何新的 pod， 会将 旧 pod 纳入自己的管辖范围；

基础使用 和 RC 一样简单。

```
1 apiVersion: apps/v1
2 kind: ReplicaSet
3 metadata:
4   name: kubia
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: kubia
10  template:
```

```
11     metadata:  
12         labels:  
13             app: kubia  
14     spec:  
15         containers:  
16             - name: kubia  
17                 image: luksa/kubia
```

5: `matchExpressions` 更强大的选择器;

```
1 selector:  
2     # 标签 需要包含的 key 和 val  
3     matchExpressions:  
4         - key: app  
5             operator: In  
6             values:  
7                 - kubia
```

6: 四个有效的运算符 `operator`:

- `In` : Label 的值 必须与其中一个指定的 values 匹配。
- `NotIn` : Label 的值与任何指定的 values 不匹配。
- `Exists` : pod 必须包含一个指定名称的标签（值不重要）。使用此运算符时，不应指定 values 字段。
- `DoesNotExist` : pod 不得包含有指定名称的标签。values 属性不得指定。

7: `matchLabels` 和 `matchExpressions` 可以同时指定，条件是与的关系。

8: 删除 ReplicaSet 也会删除 pod:

```
1 | kubectl delete rs <rs-name>
```

## DaemonSet: 在每个节点上运行一个 pod

1: 需在每个节点上运行日志收集 or 监控: 如 k8s 的 `kube-proxy` 进程

2: 通过 系统初始化脚本 or systemd 守护进程启动;

3: 无期望副本数概念，在 节点选择器下，运行一个 pod;

- 和节点绑定在一起：节点下线，并不会再创建新 pod;
- ==新节点加入【添加 节点 label 后】，匹配节点选择器，自动创建一个新的 pod；==
- 无意中删除了该 pod，会自动创建一个 pod;

4: 从 DaemonSet 的 pod 模板 创建 pod

5: 通过 节点选择器 `nodeSelector` 选中 部分节点创建 pod;

```
1 apiVersion: apps/v1beta2  
2 kind: DaemonSet  
3 metadata:
```

```

4   name: ssd-monitor
5   spec:
6     selector:
7       matchLabels:
8         app: ssd-monitor
9     template:
10    metadata:
11      labels:
12        app: ssd-monitor
13    spec:
14      # 节点选择器, 选择 标签为 disk=ssd 的节点
15      nodeSelector:
16        disk: ssd
17      containers:
18        - name: main
19          image: luksa/ssd-monitor

```

6: 节点可以设置为 不可调度 【通过调度器控制】， 防止 pod 部署到 该节点；

但 DaemonSet 管理的 pod 作为==系统服务， 完全绕过调度器， == 即使 节点是 不可调度的， 仍然可以运行系统服务；

7: 从节点删除 节点标签， DaemonSet 管理的 Pod 也会被删除：

```
1 | kubectl label node <node-name> disk=hdd --overwrite
```

## Job：运行单个任务的 pod

1: Job: 一旦任务完成， 不重启 容器；

两个地方会重启：

- 1: job 异常；
- 2: pod 在执行任务时， 被从节点逐出；

2: 会重启的资源

job 只有在执行失败的时候才会被重启；

被托管的 ReplicaSet 会重启， Job 若未完成， 也会重启。

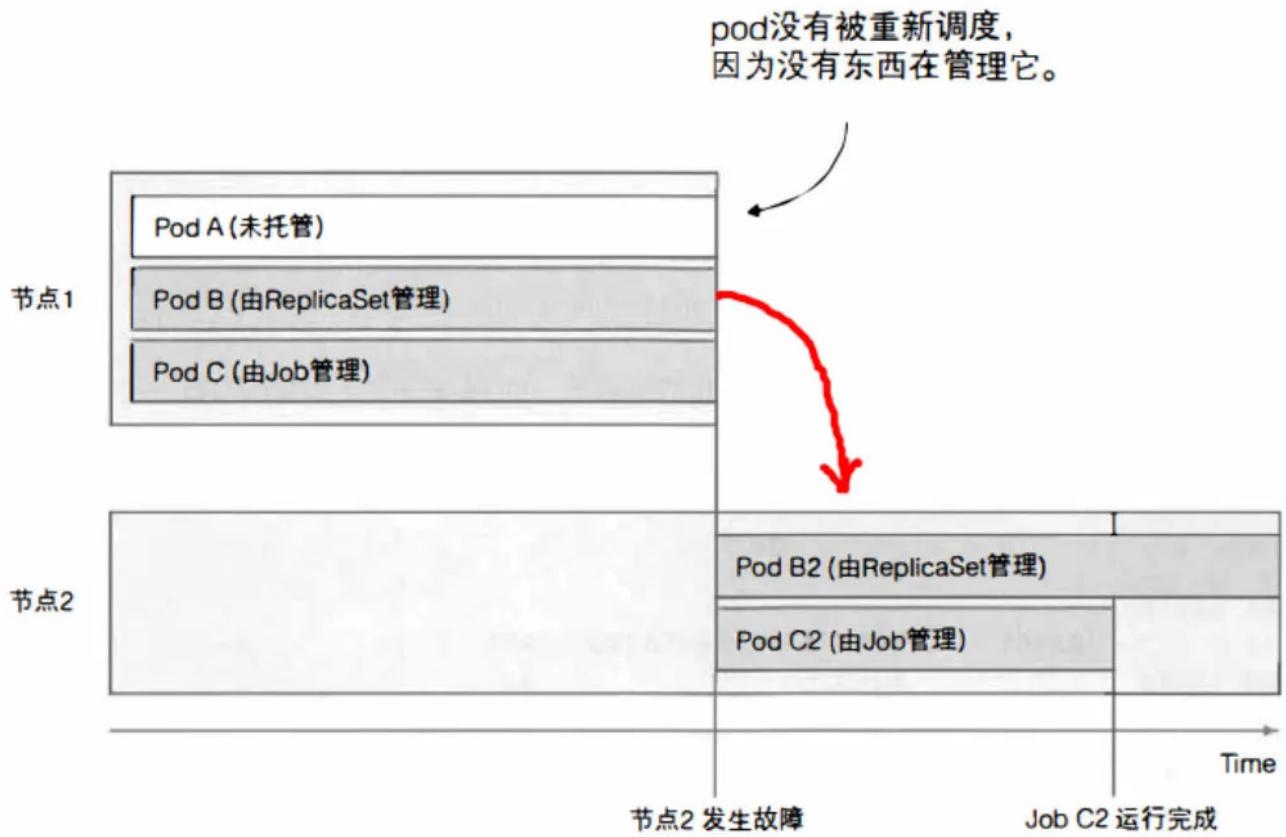


image-20210617182356951

3: job 资源:

`restartPolicy` 配置为 `Onfailure` or `Never`: 完成后不需要一直重启

```

1 apiVersion: batch/v1
2 kind: Job
3 metadata:
4   name: batch-job
5 spec:
6   template:
7     metadata:
8       labels:
9         # Job 未指定 pod 选择器, 默认根据 pod 模板中的标签创建
10        app: batch-job
11   spec:
12     # job 不能使用默认的 Always 作为重启策略
13     restartPolicy: OnFailure
14     containers:
15       - name: main
16         image: luksa/batch-job

```

4: job 中串行运行多个 pod:

```
1 kind: Job
2 metadata:
3   name: multi-completion-batch-job
4 spec:
5   completions: 5 # 顺序执行 5 次
```

5: job 中并行运行多个 pod:

```
1 spec:
2   completions: 5 # 需要完成 5 次
3   parallelism: 2 # 最多同时 2 个并行
```

6: Job 在运行时, 可调整 Job 数量:

```
1 | kubectl scale job <job-name> --replicas 3
```

7: 限制 Job 完成时间和失败重试次数:

- activeDeadlineSeconds: 限制 Job 运行的时间
- spec.backoffLimit: 默认 6, Job 失败前 可重试的次数

## CronJob: 定期执行

1: 时间格式: cron

2: 每隔 15 分钟运行一次:

```
1 apiVersion: batch/v1beta1
2 kind: CronJob
3 metadata:
4   name: batch-job-every-fifteen-minutes
5 spec:
6   # 每15分钟运行一次
7   schedule: "0,15,30,45 * * * *"
8   jobTemplate:
9     spec:
10    template:
11      metadata:
12        labels:
13          app: periodic-batch-job
14        spec:
15          restartPolicy: OnFailure
16          containers:
17            - name: main
18              image: luksa/batch-job
```

3: 时间格式:

- 分钟

- 小时
- 每月中的第几天
- 月
- 星期几

#### 4: 注意事项

- CronJob 根据 `jobTemplate` 创建 job 对象;
- `startingDeadlineSeconds` 超时未执行 视为 Failed;
- Job 能被重复执行，可能会被创建多个；
- Job 应该是串行的，中间不能有遗漏的任务；

## 服务

1: 服务可以说是 k8s 中最复杂的一环。

k8s 集群和普通集群不同的是，

- pod 是临时的，随时会被创建和关闭（动态扩缩容）
- pod 重启，会分配新的 IP 地址；

2: Service 资源：外部访问后端 pod，可提供一个统一可供外部访问的 IP 地址和端口

这里更多的是针对无状态服务，所有 pod 都是对等的，API 服务器只需随机分配一个 pod

3: 应用服务的两种情形：

- 外部集群 可通过 服务 连接 pod
- 内部 pod 之间也可通过服务连接

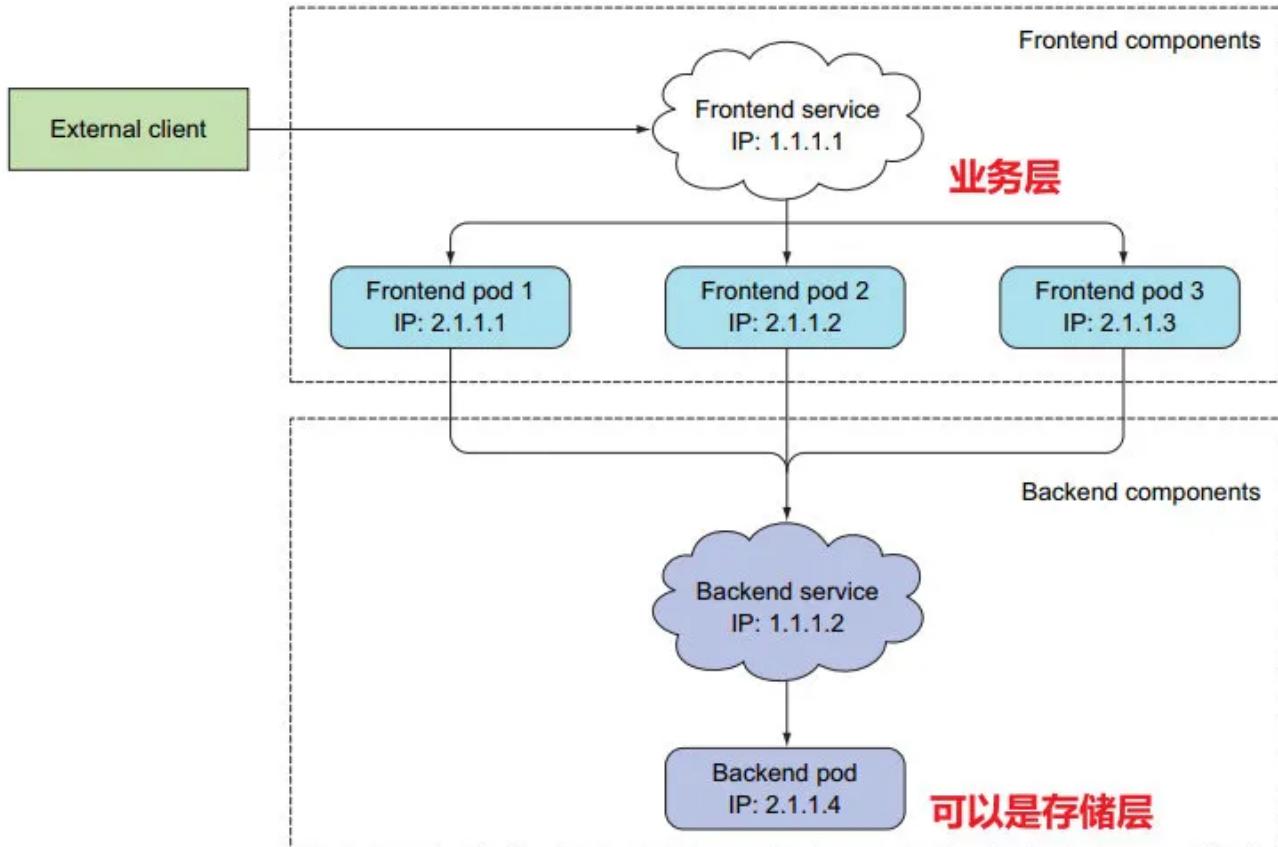
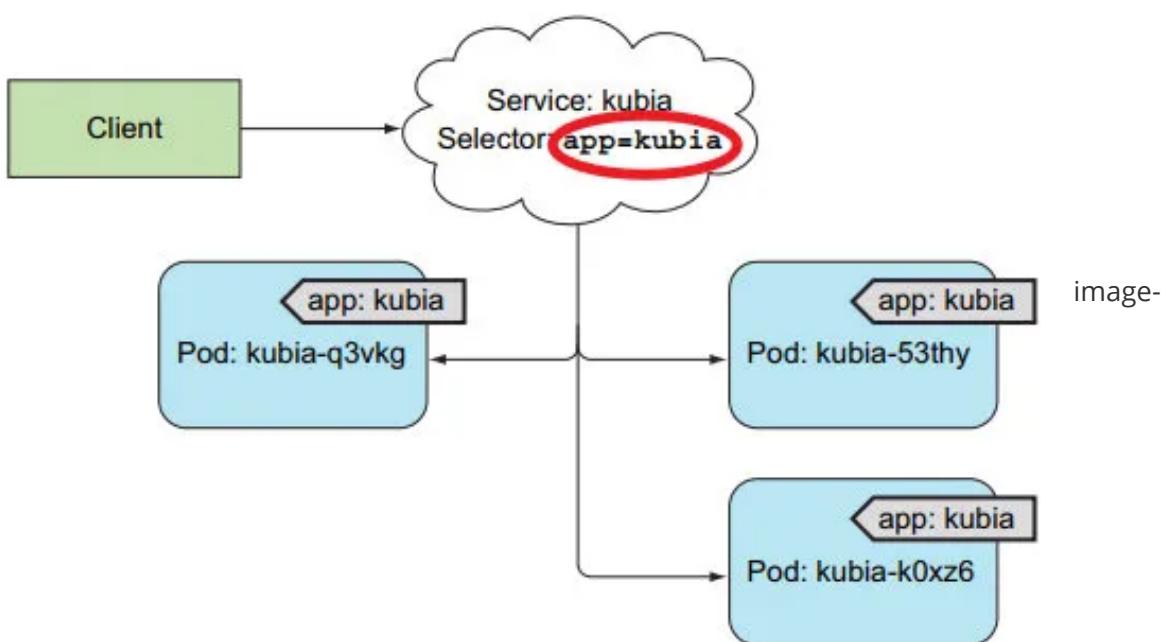


image-20210617141428940

## 连接集群内部的 Service

4: 服可通过标签选择器，选择需要连接的 pod

控制 Service 服务的范围。



20210617141644761

5: 服务可通过 `kubectl expose` 创建，也可通过 yaml 创建。

```

1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: kubia
5 spec:
6   ports:
7     # 服务对外的端口
8     - port: 80
9       # 服务转发到容器的端口
10      targetPort: 8080
11
12    # 连接pod 集合是带有 app: kubia 标签的 pods
13    selector:
14      app: kubia

```

创建 svc 后，会分配一个 **集群 IP**，==该 IP 对外不可用==，仅限于 集群内的 pod 访问 【pod 和 pod 之间也可通过 服务连接】。

```

user00@ubuntu:~$ kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes  ClusterIP  10.96.0.1      <none>        443/TCP      2d21h
kubia      ClusterIP  10.105.237.81  <none>        80/TCP       5s

```

image-20210621205022413

集群内部测试服务，有三种方法向 Service 发送请求：

- 创建一个 pod 应用，并向 集群 IP 发送 requests；
- ssh 登录到节点上，使用 curl
- 登录到其中的一个 pod 运行 curl 指令

6：可用 `kubectl exec` 在远程容器里执行：

```

1 user00@ubuntu:~$ kubectl get po
2 NAME      READY   STATUS    RESTARTS   AGE
3 kubia-jppwd  1/1     Running   0          5m13s
4 kubia-sxkrr  1/1     Running   0          5m13s
5 kubia-xvkg  1/1     Running   0          5m13s
6
7 # -- 表示 Kubectl 执行命令的结束
8 # -s 告诉 kubectl 需要连接不同的 API服务器，而非默认的
9 user00@ubuntu:~$ kubectl exec kubia-jppwd -- curl -s http://10.105.237.81

```

curl 的过程如下，Service 选择 pod 是随机选择一个。

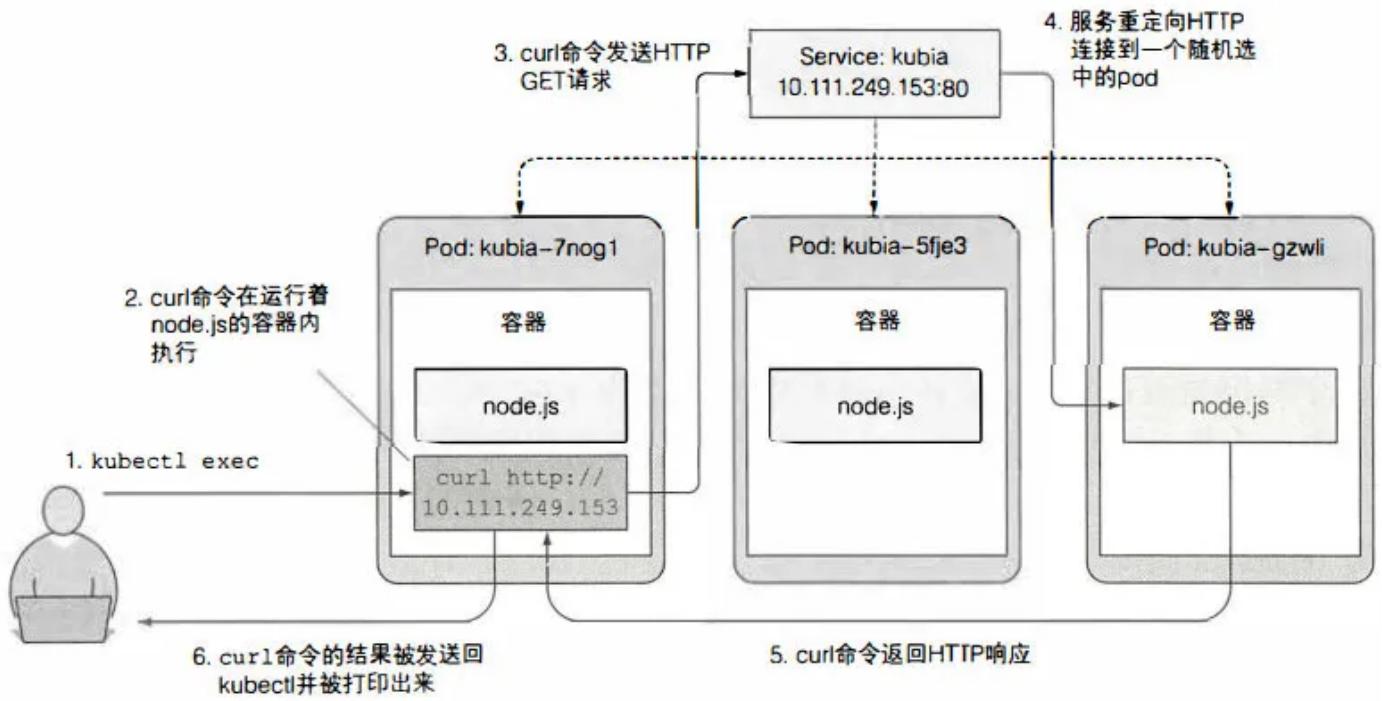


image-20210617144543726

7: Service 可以通过设置 `sessionAffinity: ClientIP` 来让同一个客户端的请求每次指向同一个 pod.

默认值是 None

8: Service 可同时暴露多个端口, 例如 http 请求时, 80 端口映射到 8080, https 请求时, 443 端口映射到 8443

9: 在 Service 的生命周期内, 服务 ip 不变。

- 当 pod 创建时, k8s 会初始环境变量指向现在的 集群 IP

```

1 user00@ubuntu:~$ kubectl exec kubia-jppwd env | grep -in service
2 kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version.
  Use kubectl exec [POD] -- [COMMAND] instead.
3 :KUBERNETES_SERVICE_HOST=10.96.0.1
4 :KUBERNETES_SERVICE_PORT_HTTPS=443
5
6 # 集群IP 和 端口
7 7:KUBIA_SERVICE_HOST=10.105.237.81
8 8:KUBIA_SERVICE_PORT=80
9 12:KUBERNETES_SERVICE_PORT=443

```

对应了两个服务:

|   | \$ kubectl get svc                                                                     |
|---|----------------------------------------------------------------------------------------|
| 1 | <code>NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE</code>     |
| 2 | <code>kubernetes  ClusterIP  10.96.0.1    &lt;none&gt;        443/TCP     2d21h</code> |
| 3 | <code>kubia      ClusterIP  10.105.237.81  &lt;none&gt;        80/TCP      50m</code>  |

10: 每个 pod 默认使用的 dns:

```
1 $kubectl exec kubia-jppwd -- cat /etc/resolv.conf
2 nameserver 10.96.0.10
3 search default.svc.cluster.local svc.cluster.local cluster.local localdomain
4 options ndots:5
```

pod 是否适用 dns, 由 `dnsPolicy` 属性决定。

系统命名空间，通常有个 pod 运行 DNS 服务；

```
1 $ kubectl get po -n kube-system
2   NAME           READY   STATUS    RESTARTS   AGE
3   coredns-74ff55c5b-pvqxv   1/1     Running   0          2d21h
```

11: 在 pod 中，可使用全限定域名 (FQDN) 来访问 Service。

格式如下：

```
1 kubia.default.svc.cluster.local
2
3 # kubia: Service 名称
4 # default: namespace
5 # svc.cluster.local: 所有集群本地服务中使用的可配置集群域后缀
```

若在同命名空间下，`svc.cluster.local` 和 `default` 可省略。

```
1 user00@ubuntu:~$ kubectl exec -it kubia-jppwd bash
2 kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version.
3 Use kubectl exec [POD] -- [COMMAND] instead.
4 root@kubia-jppwd:/# curl http://kubia.default.svc.cluster.local
5 You've hit kubia-sxkrr
6 root@kubia-jppwd:/# curl http://kubia.default
7 You've hit kubia-xvkpg
8 root@kubia-jppwd:/# curl http://kubia
9 You've hit kubia-xvkpg
```

12: 集群 IP 是一个虚拟的 IP，只有配合服务端口才有意义。

```
1 root@kubia-jppwd:/# ping kubia
2 PING kubia.default.svc.cluster.local (10.105.237.81): 56 data bytes
3 ^C--- kubia.default.svc.cluster.local ping statistics ---
4 51 packets transmitted, 0 packets received, 100% packet loss
```

## 连接集群外部的 Service

- 1: 让 pod 连接到集群外部。
- 2: 服务并不是和 pod 直接相连，中间有 Endpoint 资源

```
1 user00@ubuntu:~$ kubectl get svc kubia
2 Selector:           app=kubia # 创建 endpoint 资源, 选择的 pod 标签
3
4 TargetPort:         8080/TCP
5 Endpoints:          172.17.0.3:8080,172.17.0.4:8080,172.17.0.5:8080
```

Service 通过选择器构建 IP 和端口列表，然后存储在 endpoint 资源中

连接时，随机选择其中一个

- 3: 当 Service 无节点选择器时，不会自动创建 Endpoint 资源。

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   # Service 名称, 后面会用
5   name: external-service
6 spec:
7   ports:
8     - port: 80
9
10  # 无 选择器
```

手动指定 Endpoint 注意需要==和 Service 同名称==。

```
1 apiVersion: v1
2 kind: Endpoints
3 metadata:
4   # 和 Service 同名称
5   name: external-service
6 subsets:
7   # Service 重定向的 IP 地址
8     - addresses:
9       - ip: 11.11.11.11
10      - ip: 22.22.22.22
11     ports:
12       - port: 80
```

通过 endpoint 列表，可连向其它地址。

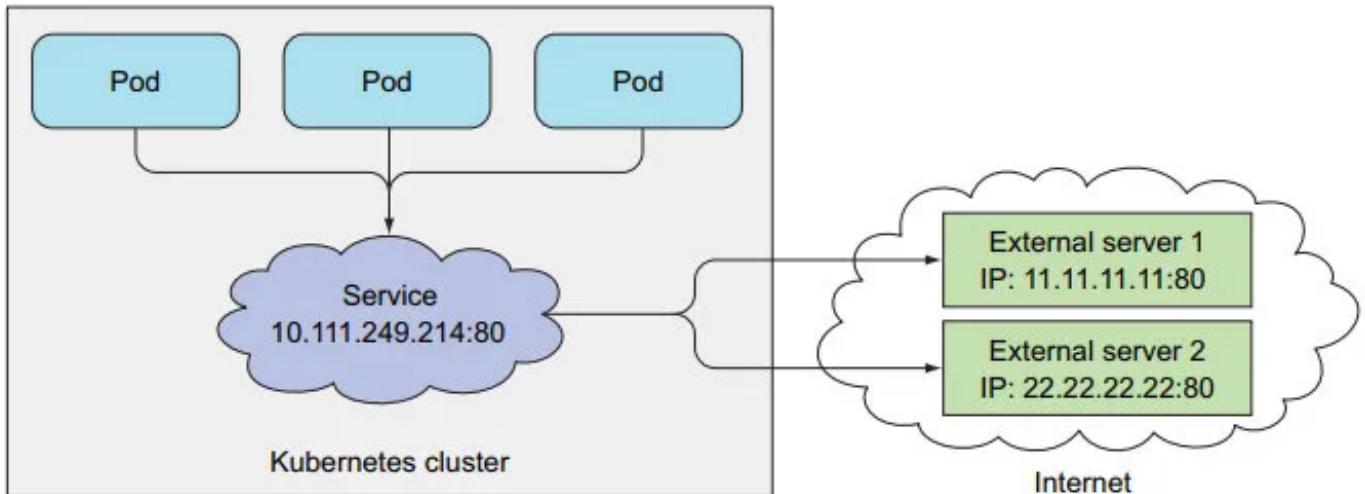


image-20210617154439324

4: 也可创建具有别名的外部服务:

```

1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: external-service
5 spec:
6   # 别名
7   type: ExternalName
8   externalName: api.somecompany.com
9   ports:
10  - port: 80

```

创建该服务后，内部 pod 可通过 `external-service.default.svc.cluster.local` 访问外部域名

## 将服务暴露给外部客户端

1: pod 向外部公开的服务，如 web 服务。

2: 有以下几种方式，使外部可访问服务：

- 服务类型为 NodePort: 每个节点上开放一个端口，访问内部服务 (可被 Service 访问)。
- 服务类型为 LoadBalance: NodePort 的一种扩展，通过负载均衡器访问，将流量重定向到所有节点的 NodePort;
- 创建 Ingress 资源：通过一个 IP 地址公开多个服务 (运行在 HTTP 层)

### NodePort

1: 创建一个 NodePort 类型的 Service。

```

1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: kubia-nodeport
5 spec:

```

```

6 # 服务类型为 NodePort
7 type: NodePort
8 ports:
9   # 集群IP的端口
10  - port: 80
11    # pod 开放的端口
12    targetPort: 8080
13
14  # 不指定端口，将随机选择一个端口
15  nodePort: 30123
16 selector:
17  app: kubia

```

2: 现在可通过以下几种方式访问服务：

```

1 user00@ubuntu:~$ kubectl get svc
2 NAME           TYPE      CLUSTER-IP        EXTERNAL-IP   PORT(S)
3 kubia-nodeport   NodePort  10.104.119.122 <none>       80:30123/TCP
4 5m21s

```

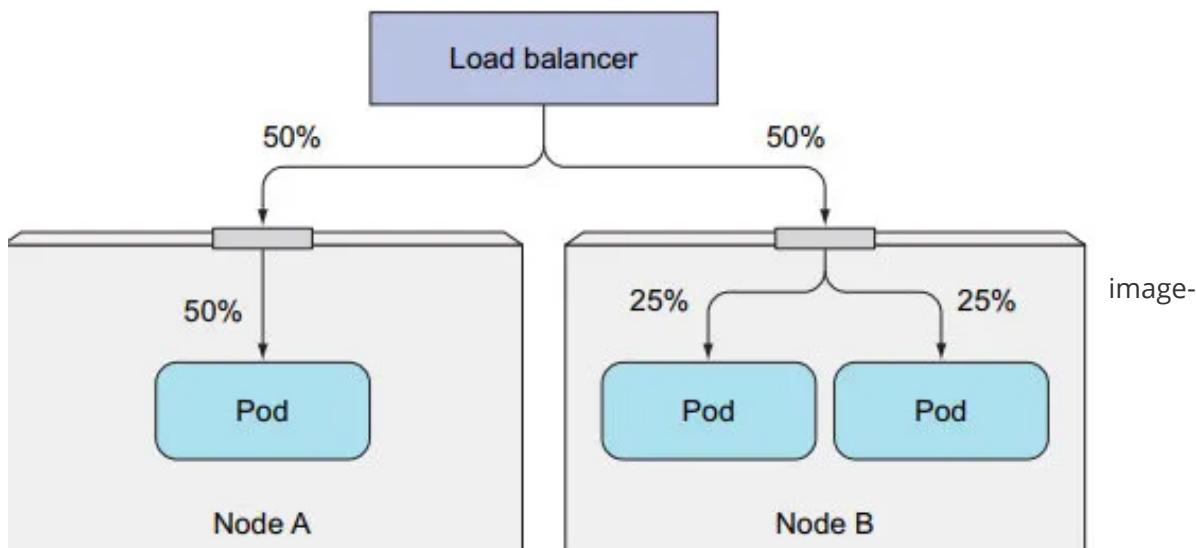
- 1: <集群 IP>:80
- 2: 多节点集群
  - 节点 1IP:30123
  - 节点 2IP:30123

注意：即使连到节点 1 的端口，也可能分配到 节点 2 上执行。

设置 `externalTrafficPolicy:local` 属性，可将 流量路由到当前节点所在的 pod . 若当前节点 pod 不存在，连接挂起。

`externalTrafficPolicy:local` 不利于负载均衡，当节点上的 pod 分散不均时。

有个好处是，转发到本地 pod，不用进行 SNAT（源网路地址转换），这会将 改变 源 IP 记录。



开放 NodePort 端口。

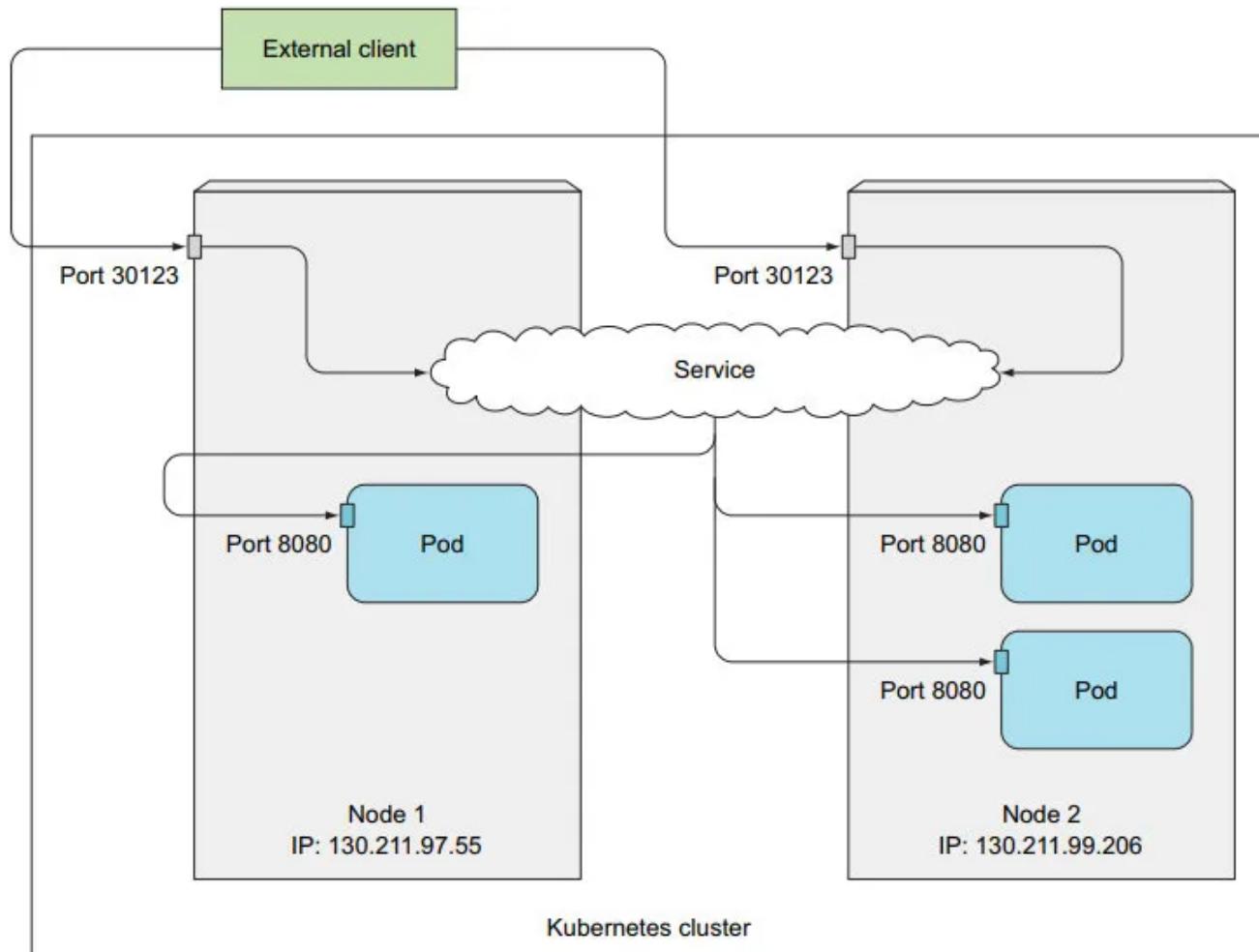


image-20210617162126942

3: 获取节点 IP 【特意观察了下，节点 IP 和机器 IP 不是一个，但在同一网关下】

```
1 user00@ubuntu:~$ kubectl get nodes -o json | grep address
2         "addresses": [
3             "address": "192.168.49.2",
```

可在本地机器上，向 `节点IP:nodePort` 发送请求：

```
1 $ curl http://192.168.49.2:30123
2 You've hit kubia-jppwd
```

minikube 可通过网页访问 `minikube service <service-name>`：

image-20210617164756909

## LoadBalancer

负载均衡器在 NodePort 外面包了一层。

1: 负载均衡器放在节点前面，将请求传播到健康的节点。

负载均衡器拥有自己独一无二的 可公开访问的IP 地址，并将连接重定向到服务。

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: kubia-loadbalancer
5 spec:
6   # 负载均衡器类型
7   type: LoadBalancer
8   ports:
9     - port: 80
10    targetPort: 8080
11   selector:
12     app: kubia
```

若没有指定特定端口，负载均衡器会随机选择一个端口。

```
1 $ kubectl get svc | grep load
2           # <EXTERNAL IP>
3 kubia-loadbalancer  LoadBalancer  10.96.24.178  <pending>      80:30600/TCP
4 11m
```

2：创建服务后，要等待很长时间，才能将外部 IP 地址写入对象。

可通过外部 ip 直接访问：

```
1 curl http://<EXTERNAL IP>
```

3：可以通过浏览器访问，但每次访问都是一个 pod，即使 `Session Affinity` 设为 None

因为浏览器采用 keep-alive 连接，而 curl 每次开新连接。

4：请求连的时候，会连接到负载均衡器的 80 端口，并路由到某个节点上分配的 NodePort 上，随后转发到一个 pod 实例上。

本质还是打开了 NodePort，仍能继续使用 `节点IP:隐式 NodePort 端口` 访问：

```
1 $ curl http://192.168.49.2:30600
2 You've hit kubia-sxkrr
```

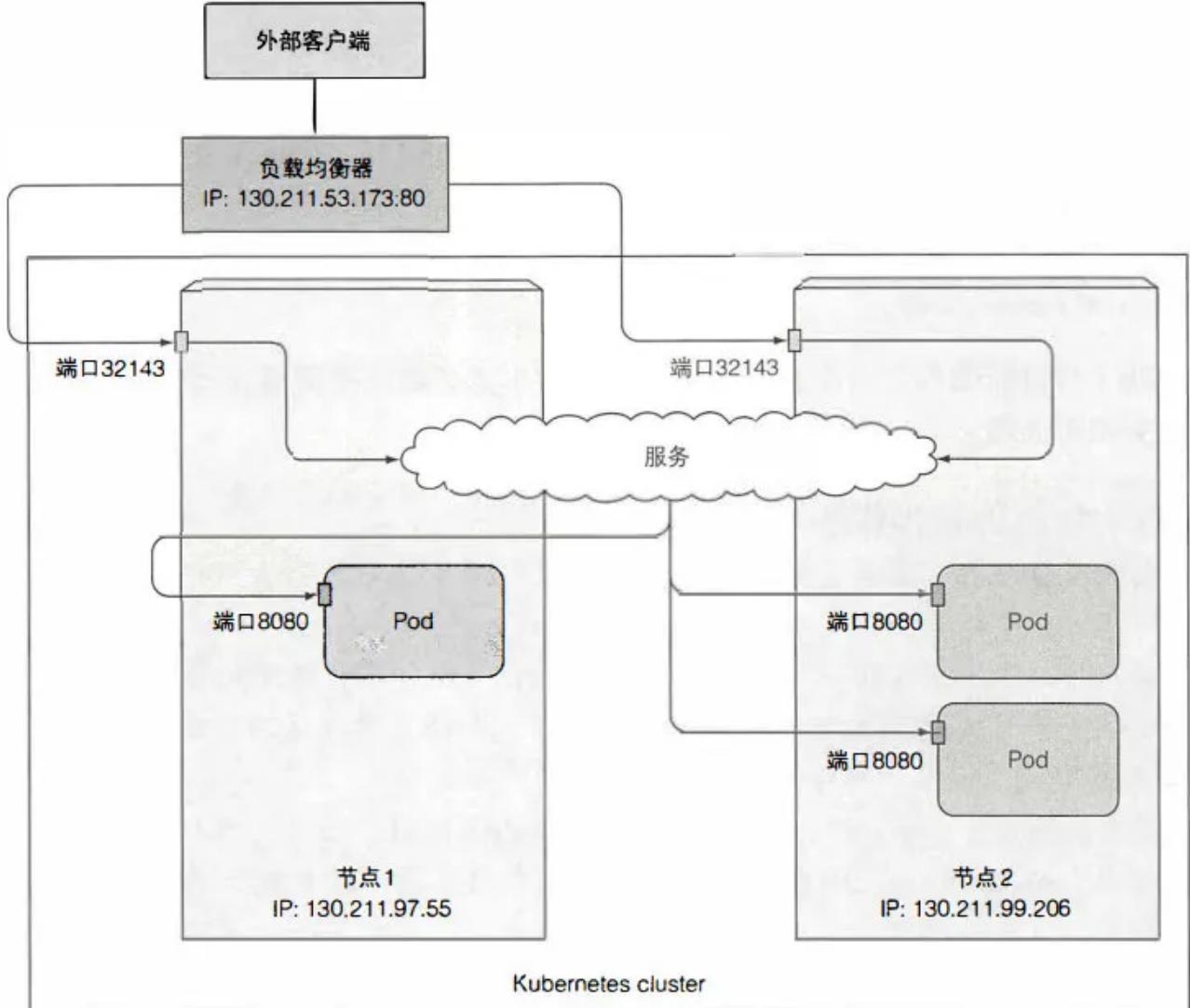


image-20210617170233160

## Ingress

- 1: Ingress 也可对外暴露服务，准入流量控制。
- 2: Ingress 工作在 HTTP 层，通过一个 主机名 + 路径 就能转到不同的服务

而 每个 LoadBalancer 服务需要自己的负载均衡器和独有的 公有 IP 地址。

工作在更高层次的协议层，可以提供更丰富的服务。

ingress 可以绑定 ==多主机、同主机多路径。==

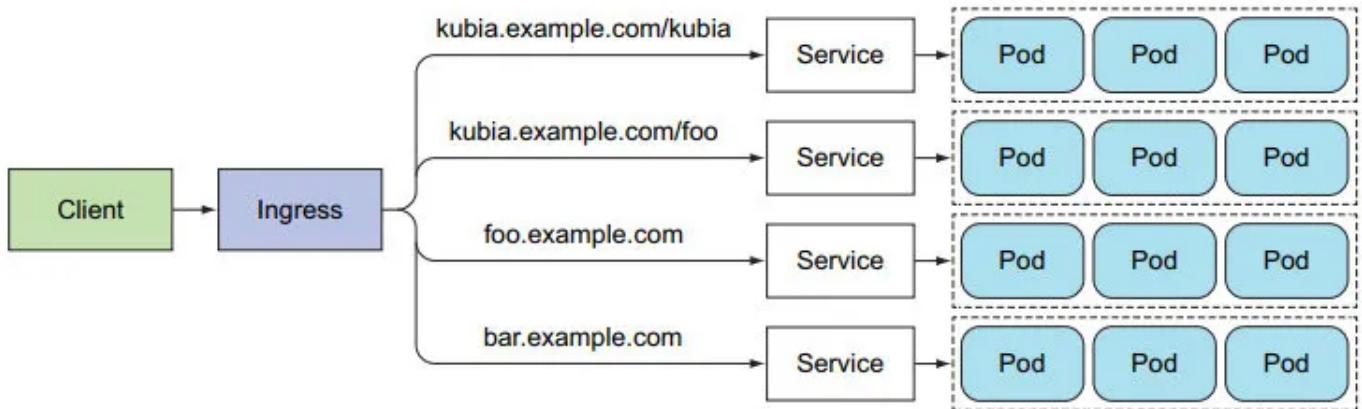


image-20210617171436236

3: 启用 Ingress 资源需要 Ingress 控制器。

通过 `minikube addons list` 确认。

```

1 apiVersion: extensions/v1beta1
2 kind: Ingress
3 metadata:
4   name: kubia
5 spec:
6   rules:
7     - host: kubia.example.com
8       http:
9         paths:
10           - path: /
11             backend:
12               # 将 kubia.example.com/ 请求转发到 nodeport 服务
13               serviceName: kubia-nodeport
14               servicePort: 80

```

4: `kubia.example.com` 访问服务的前提是 域名能正确解析为 ingress 的 IP。

```

1 $ kubectl get ingress
2 NAME      CLASS      HOSTS          ADDRESS      PORTS      AGE
3 kubia     <none>    kubia.example.com  192.168.49.2  80        70s

```

然后再 `/etc/hosts` 加入映射:

```

1 root@ubuntu:~# cat /etc/hosts
2 127.0.0.1      localhost
3 127.0.1.1      ubuntu
4
5 192.168.49.2  kubia.example.com

```

通过地址访问:

```
1 # 必须配置 hosts, 直接通过 ingress IP 访问是不行的, 无法知道访问的是哪个服务
2 $ curl http://kubia.example.com
3 You've hit kubia-xvkg
```

5: ingress 访问流程如下:

- 通过 DNS 查找 `http://kubia.example.com` 对应的 ingress IP
- 向 Ingress 控制器发送请求, 并在头部中包含需要访问的服务 `kubia.example.com`
- 在 Endpoints 中查看该服务对应的 pod 的 IP 表, 选择其中一个 pod 进行处理;

image-20210617172908817

6: 若要采用 https 访问, 需要配置 tls.secret

针对每个主机域名 配置一个。

```
1 name: kubia
2 spec:
3   tls:
4     - hosts:
5       - kubia.example.com
6       secretName: tls-secret
```

## 就绪探针

1: 确认服务启动后, 对外提供服务。

## headless 服务

有时候在集群内部/或者集群外部 需要知道 其它节点的 IP 列表, 创建一个 headless 服务包裹一层, 去查询该服务的 DNS, 会打印 该服务下的 (标签选择器选中的) 所有 pod。

1: `clusterIP:None`, DNS 服务器返回的是 pod 的 IP, 而非集群 IP

2: 创建 headless Service:

该 headless 后端, 包含标签选择器选择的所有 pod

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: kubia-headless
5 spec:
6   # headless 服务
7   clusterIP: None
8   ports:
9     - port: 80
10    targetPort: 8080
11   selector:
12     app: kubia
```

headless 服务无集群 IP

| NAME               | TYPE         | CLUSTER-IP     | EXTERNAL-IP   | PORT(S)      | AGE   |
|--------------------|--------------|----------------|---------------|--------------|-------|
| external-service   | ExternalName | <none>         | www.baidu.com | 80/TCP       | 108m  |
| kubernetes         | ClusterIP    | 10.96.0.1      | <none>        | 443/TCP      | 3d    |
| kubia              | ClusterIP    | 10.105.237.81  | <none>        | 80/TCP       | 3h21m |
| kubia-headless     | ClusterIP    | None           | <none>        | 80/TCP       | 10s   |
| kubia-loadbalancer | LoadBalancer | 10.96.24.178   | <pending>     | 80:30600/TCP | 49m   |
| kubia-nodeport     | NodePort     | 10.104.119.122 | <none>        | 80:30123/TCP | 89m   |

image-20210617174322816

运行临时 pod

```
1 | kubectl run dnsutils --image=tutum/dnsutils --command -- sleep  
infinitypod/dnsutils created
```

在该临时 pod 查看 headless 服务标签选择器选择的 pod

注意，只会展示 就绪的 pod ip

```
1 | $ kubectl exec dnsutils nslookup kubia-headless  
2 | Name: kubia-headless.default.svc.cluster.local  
3 | Address: 172.17.0.5  
4 | Name: kubia-headless.default.svc.cluster.local  
5 | Address: 172.17.0.4  
6 | Name: kubia-headless.default.svc.cluster.local  
7 | Address: 172.17.0.3
```

普通 service 返回的是集群 IP：

```
1 | $ kubectl exec dnsutils nslookup kubia  
2 |  
3 | Name: kubia.default.svc.cluster.local  
4 | Address: 10.105.237.81
```

## 卷

1：卷的作用是将 磁盘挂载到容器。

这个和 linux 将指定目录挂载到盘 很类似。

2：每个 pod 都有独立的文件系统，文件系统来自于 容器镜像。

默认，容器重启后并不能识别 之前容器写入文件系统的内容。

这是因为 新的容器拥有 新的 写入层。

3：pod 中的所有容器都能使用卷，但是需要提前挂载。

4：emptyDir 卷是挂载一个空的目录。

- 卷的装载在容器启动之前执行；

- emptyDir 卷的生命周期和 pod 相同；

5: 可用的卷类型：

- emptyDir —— 用于存储临时数据的简单空目录。
- hostPath —— 用于将目录从工作节点的文件系统挂载到 pod 中。
- gitRepo —— 通过检出 Git 仓库的内容来初始化的卷。
- nfs —— 挂载到 pod 中的 NFS 共享卷。
- gcePersistentDisk (Google 高效能型存储磁盘卷)、awsElastic BlockStore (AmazonWeb 服务弹性块存储卷)、azureDisk (Microsoft Azure 磁盘卷) —— 用于挂载云服务商提供的特定存储类型。
- cinder、ceph、iscsi、flocker、glusterfs、quobyte、rbd、flexVolume、vsphere-Volume、photonPersistentDisk、scaleIO 用于挂载其他类型的网络存储。
- configMap、secret、downwardAPI —— 用于将 Kubernetes 部分资源和集群信息公开给 pod 的特殊类型的卷。
- persistentVolumeClaim —— 一种使用预置或者动态配置的持久存储类型（我们将在本章的最后一节对此展开讨论）。

单个容器可同时使用不同类型的多个卷

## emptyDir

1: emptyDir: 在 pod 中的多个容器间共享存储：

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: fortune
5  spec:
6    containers:
7      - image: luksa/fortune
8        name: html-generator
9        volumeMounts:
10       - name: html
11         # 挂载的目录
12         mountPath: /var/htdocs
13      - image: nginx:alpine
14        name: web-server
15        volumeMounts:
16       - name: html
17         mountPath: /usr/share/nginx/html
18         # 只读
19         readOnly: true
20     ports:
21       # Nginx 监听80端口
22       - containerPort: 80
23         protocol: TCP
24     volumes:
25       - name: html # 卷名称
26         emptyDir: {}

```

可在内存上创建 emptyDir

```
1 volumes:
2   - name: html # 卷名称
3     # emptyDir: {}
4     emptyDir:
5       medium: Memory
```

## gitRepo

1: gitRepo 卷：gitRepo 卷基本上也是一个 emptyDir 卷，它通过克隆 Git 仓库并在 pod 启动时（但在创建容器之前）检出特定版本来填充数据

创建 pod 时，会 checkout 指定版本。

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: gitrepo-volume-pod
5 spec:
6   containers:
7     - image: nginx:alpine
8       name: web-server
9       volumeMounts:
10      - name: html
11        mountPath: /usr/share/nginx/html
12        readOnly: true
13   ports:
14     - containerPort: 80
15       protocol: TCP
16   volumes:
17     - name: html
18       # 创建一个 gitRepo 卷
19       gitRepo:
20         repository: https://github.com/luksa/kubia-website-example.git
21         revision: master
22         # checkout 到当前目录， 可通过路径 /usr/share/nginx/html 访问
23         directory: .
```

8: 可创建一个 sidecar 容器，实时同步 git，如 Docker Hub 上的 gitsync

## hostPath

1: hostPath 卷，指向节点文件系统上特定的文件或目录。

注意是一些系统级别的 pod（通常由 DaemonSet 管理）需要访问。

2: 多个 pod hostPath 卷中使用相同的路径，可看到相同的文件。

image-20210617125426036

3: hostPath 是第一种持久性存储的卷。【pod 删除后依然还在】

emptyDir 和 gitRepos 随 pod 删除而删除。

4: 若将数据存储到 节点上，则 pod 不能随机调度，需要调度到指定节点才行。

特别是 pod 重启时。

## nfs

1: nfs 服务器可共享路径

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: mongodb-nfs
5 spec:
6   volumes:
7     - name: mongodb-data
8       nfs:
9         # 指定 nfs 服务
10        server: 1.2.3.4
11        # 共享路径
12        path: /some/path
13   containers:
14     - image: mongo
15       name: mongodb
16       volumeMounts:
17         - name: mongodb-data
18           mountPath: /data/db
19       ports:
20         - containerPort: 27017
21           protocol: TCP
```

## PV 和 PVC

1: 将卷这种持久性信息 和 pod 解耦，可避免处理基础设施细节。

2: PersistentVolume (持久卷， 简称 PV)，由集群管理员设置的底层存储。

3: PersistentVolumeClaim (持久卷声明，简称 PVC)，用户声明需要申请的 (存储容量和访问模式)

API 服务器 负责找到满足要求的 持久卷并绑定到 持久卷声明。

持久卷声明可当做 pod 中的一个卷来使用；

image-20210617130652486

4: 创建 PV

```
1 apiVersion: v1
2
3 # 创建持久卷
```

```

4 kind: PersistentVolume
5 metadata:
6   name: mongodb-pv
7 spec:
8   # 持久卷大小
9   capacity:
10    storage: 1Gi
11   # 读写模式
12   accessModes:
13     # 单个客户端挂载是 读写模式
14     - ReadWriteOnce
15     # 多个客户端挂载是 只读模式
16     - ReadOnlyMany
17
18   # 设置策略, PVC 释放后, PV将保留
19 persistentVolumeReclaimPolicy: Retain
20
21   # 在Google的 gce 磁盘上分配
22 gcePersistentDisk:
23   pdName: mongodb
24   fsType: ext4

```

`persistentVolumeReclaimPolicy` 可指定回收策略:

- Retain: 删除 PVC 后, PV 保留;
- Recycle: 删除内容, 可再次被 PVC 绑定
- Delete: 删除底层存储

5: 特别注意的是:

PV: 属于集群层面的资源;

PVC 和 Pod: 属于命名空间内的资源;

PVC 声明:

```

1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: mongodb-pvc
5 spec:
6   resources:
7     # 申请 容量
8     requests:
9       storage: 1Gi
10    # 单个客户端, 支持读写
11    accessModes:
12      - ReadWriteOnce
13    # 动态配置
14    storageClassName: ""

```

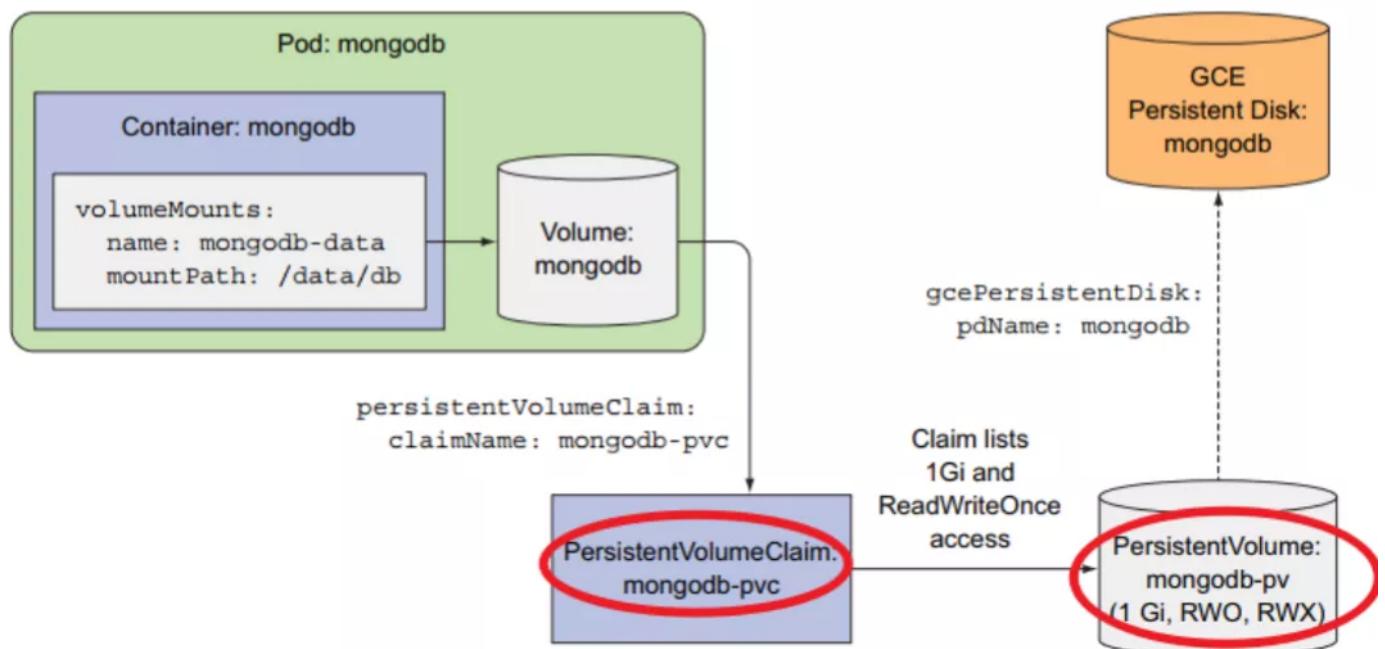
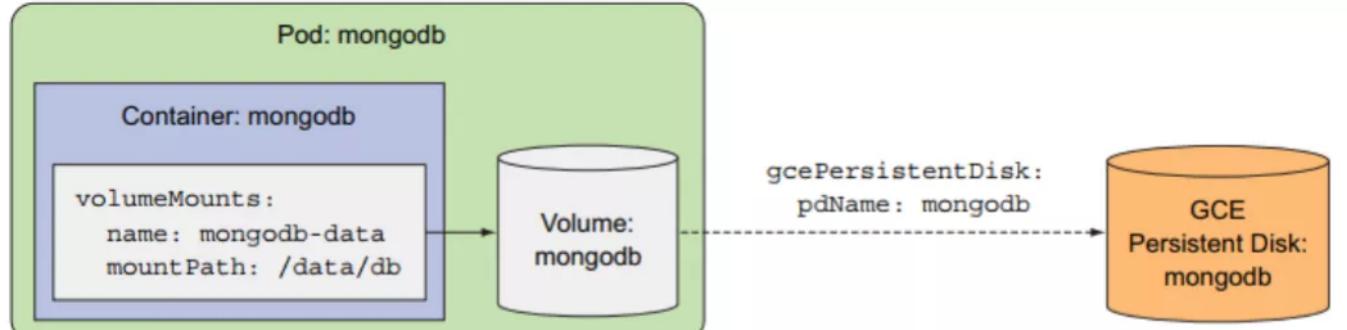
k8s 根据声明的访问权限、容量大小， 寻找满足要求的 PV

PV 和 PVC 的创建相对独立。

6: 使用 PVC，只需要在 volumes 中引用即可。

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: mongodb
5 spec:
6   containers:
7     - image: mongo
8       name: mongodb
9       volumeMounts:
10      - name: mongodb-data
11        mountPath: /data/db
12     ports:
13       - containerPort: 27017
14         protocol: TCP
15   volumes:
16     - name: mongodb-data
17     # 引用 PVC
18     persistentVolumeClaim:
19       claimName: mongodb-pvc
```

7: PV 和 PVC 的解耦，存储这块，方便管理。



## StorageClass

1: 创建存储类 StorageClass，可动态创建 PV。

方便集群管理员管理。

2: 不指定 StorageClass，会使用集群默认的存储类分配。

```

1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: mongodb-pvc2
5 spec:
6   resources:
7     requests:
8       storage: 100Mi
9   accessModes:
10    - ReadWriteOnce
11
12 # 未指定 StorageClass, 则使用默认的 StorageClass 分配

```

3: 手动配置的 PV 和 StorageClass 可同时存在，若不想用 StorageClass 分配时，可将 `StorageClass: ""` 配置为空，则将使用预先配置的 PV 持久卷。

4: 创建一个 StorageClass 对象：

```

1 apiVersion: storage.k8s.io/v1
2 kind: StorageClass
3 metadata:
4   name: fast
5
6 # 创建 PV的预制程序
7 provisioner: k8s.io/minikube-hostpath
8 # 参数
9 parameters:
10  type: pd-ssd

```

在 PVC 中使用 StorageClass:

```

1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: mongodb-pvc
5 spec:
6   # 引用 StorageClass 即可
7   storageClassName: fast
8   resources:
9     requests:
10      storage: 100Mi
11   accessModes:
12     - ReadWriteOnce

```

5: 整体关系如图所示：

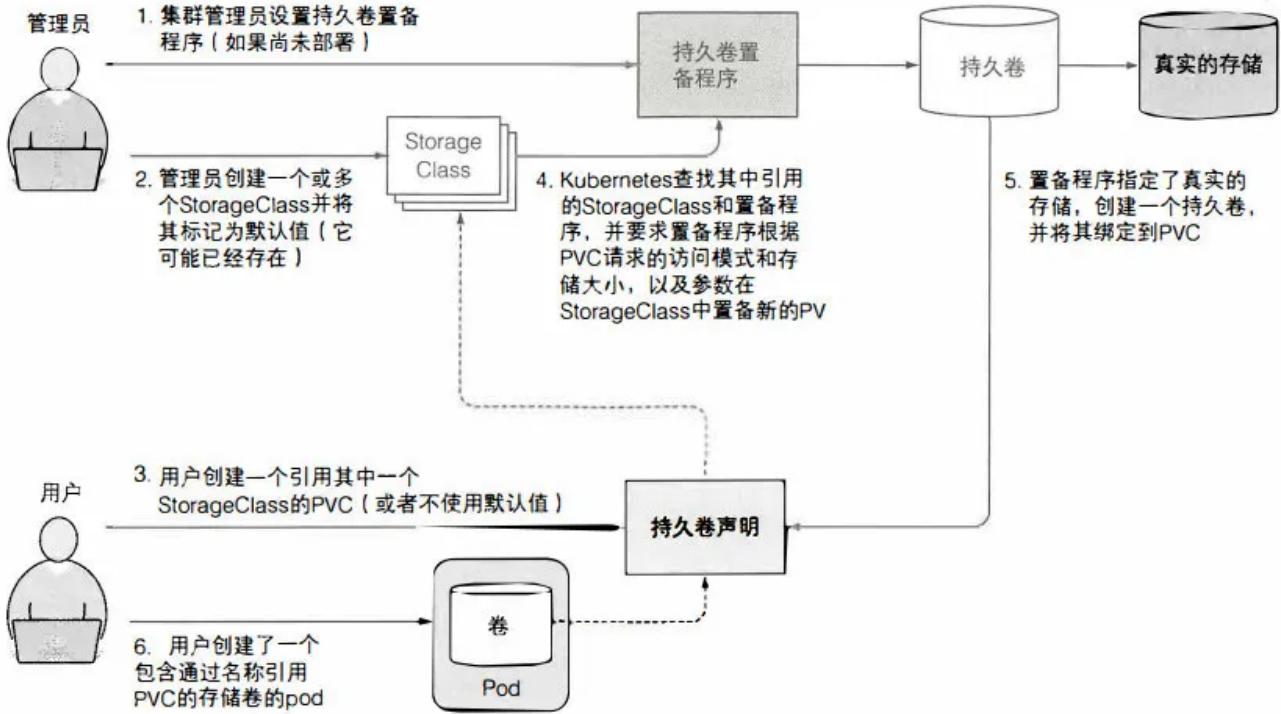


image-20210617140107905

## ConfigMap 和 Secret

### ConfigMap

1: 给应用传递参数，有以下几种方式：

- 命令行参数；

```
1 | docker run <image> <arguments>
```

在容器中启动有两种方式：

-- shell 形式一如 ENTRYPOINT node app.js。

-- exec 形式一如 ENTRYPOINT ["node", "app.js"]

前者是通过 shell 启动的，后者是 进程直接运行。

例如 在 Dockerfile 镜像中传入：

```
1 FROM ubuntu:latest
2
3 RUN apt-get update ; apt-get -y install fortune
4 ADD fortuneloop.sh /bin/fortuneloop.sh
5
6 ENTRYPOINT [ "/bin/fortuneloop.sh" ]
7 # 参数列表
8 CMD [ "10" ]
```

在 pod 中定义容器时，可覆盖 ENTRYPOINT 和 CMD，只需要设置 command 和 args

```

1 kind:pod
2 spec:
3   containers:
4     - image: some/image
5       command: [ "/bin/command" ]
6       args: [ "arg1", "arg2", "arg3" ]
7
8     # 多个参数, 可用下面格式
9     args:
10    - foo
11    - bar
12
13   # 数值需要用引号
14   - "15"

```

- 环境变量；

在容器中设置环境变量如下，可以在 shell 中直接引用该变量 `$INTERVAL`

```

1 kind: Pod
2 metadata:
3   name: fortune-env
4 spec:
5   containers:
6     - image: luksa/fortune:env
7       env:
8         - name: INTERVAL
9           value: "30"
10
11        - name: SECOND_VAR
12          # 引用第一个参数 30test
13          value: "${INTERVAL}test"

```

- 通过特殊类型的卷挂载到容器

卷挂载如下：

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: fortune-env
5 spec:
6   containers:
7     - image: luksa/fortune:env
8       env:
9         - name: INTERVAL
10          value: "30"
11
12        - name: SECOND_VAR

```

```

13      # 引用第一个参数 30test
14      value: "${INTERVAL}test"
15
16      name: html-generator
17
18      volumeMounts:
19          - name: html
20              mountPath: /var/htdocs
21
22      - image: nginx:alpine
23          name: web-server
24
25          volumeMounts:
26              - name: html
27                  mountPath: /usr/share/nginx/html
28                  readOnly: true
29
30          ports:
31              - containerPort: 80
32                  protocol: TCP
33
34          volumes:
35              - name: html
36                  emptyDir: {}

```

- ConfigMap 是 k8s 中存储配置数据的资源。
- 证书和私钥相关的配置数据，使用 Secret 资源存储。

2: ConfigMap 主要是将配置从 pod 中解耦出来，这样生产环境和非生产环境下的 pod 定义文件可以保持不变。

多种环境下，只要保证 ConfigMap 不同即可。

```

1  kubectl create configmap <configmap-name> --from-literal=foo=bar  # kv 结构, key 是
2    foo, value 是bar
3
3  kubectl create configmap <configmap-name> --from-file=customkey=config.conf  # 从文件
   读取 或者 文件夹读取

```

如：

```
$ kubectl create configmap my-config
  --from-file=foo.json
  --from-file=bar=foobar.conf
  --from-file=config-opts/
  --from-literal=some=thing
```

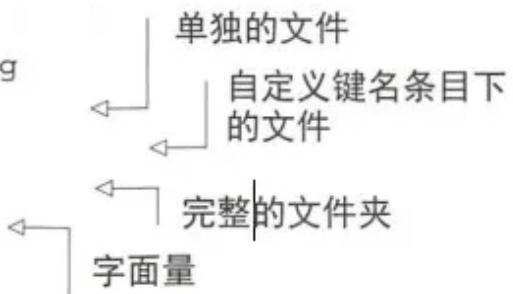


image-20210617103038594

对应关系如下：

image-20210617103121948

3：将 configMap 条目作为环境变量：

```
1 kind: Pod
```

```

2 metadata:
3   name: fortune-env-from-configmap
4 spec:
5   containers:
6     - image: luksa/fortune:env
7       env:
8         - name: INTERVAL # 设置环境变量
9           valueFrom:
10            configMapKeyRef: # 来自于 ConfigMap
11              # 引用 config-map 的名称
12              name: fortune-config
13              # config-map 下的 key
14              key: sleep-interval
15   args: ["${INTERVAL}"] # 作为命令行参数

```

引用不存在的 configMap, 容器会启动失败, 从而一直重启。

4: `envFrom` 字段可暴露所有来自 ConfigMap 的变量, 并可在变量名引入后加入 前缀。

5: 可将 configMap 中的条目挂载到指定目录下 【每个 key 作为 文件存在】

```

1 kind: Pod
2 metadata:
3   name: fortune-configmap-volume
4 spec:
5   containers:
6     - image: luksa/fortune:env
7       env:
8         - name: INTERVAL
9           valueFrom:
10            configMapKeyRef:
11              name: fortune-config
12              key: sleep-interval
13   name: html-generator
14   volumeMounts:
15     - name: html
16       # 将卷中的条目, 挂载至该目录下, 条目的 名称就是该目录下的文件名
17       mountPath: /var/www/html
18   - image: nginx:alpine
19     name: web-server
20     volumeMounts:
21       - name: html
22         mountPath: /usr/share/nginx/html
23         readOnly: true
24       - name: config
25         mountPath: /etc/nginx/conf.d
26         readOnly: true
27       - name: config
28         mountPath: /tmp/whole-fortune-config-volume
29         readOnly: true

```

```
30     ports:
31         - containerPort: 80
32             name: http
33             protocol: TCP
34     volumes:
35         - name: html
36             emptyDir: {}
37         - name: config
38             # configMap 卷
39             configMap:
40                 name: fortune-config
```

6: 注意: configMap 挂载到已存在的文件夹, 会隐藏所有已有的条目。

可以选择挂载部分卷, 避免隐藏整个文件夹。

```
1 spec:
2   containers:
3     - image: nginx:alpine
4       name: web-server
5       volumeMounts:
6         - name: html
7           mountPath: /usr/share/nginx/html
8             # 只挂载 configmap 中的指定条目
9             subPath: myconfig.conf
```

configMap 卷中的文件权限默认设置为 644 (-rw-r-r--),

可通过 `defaultMode` 修改

7: ConfigMap 通过暴露卷, 可以达到配置热更的效果, 无需新建 pod 或 重启容器。

## Secret

1: 和 configMap 一样, secret 也是 key-val 存储。

2: 使用 和 ConfigMap 相同:

- 将 Secret 条目作为环境变量传递给容器
- 将 Secret 条目暴露为卷中的文件

Secret 只会存储在节点的内存中, 永不写入物理存储,

3: 使用场景:

- 采用 ConfigMap 存储非敏感的文本配置数据。
- 采用 Secret 存储天生敏感的数据, 通过键来引用。如果一个配置文件同时包含敏感与非敏感数据, 该文件应该被存储在 Secret 中。

4: Secrets 一般包含三种文件:

- ca.crt
- namespace

- token

```

1 user00@ubuntu:~$ kubectl describe secrets
2 Name:         default-token-c2v26
3 Namespace:    default
4 Labels:       <none>
5 Annotations: kubernetes.io/service-account.name: default
                 kubernetes.io/service-account.uid: 1fe33279-b738-4a3b-a012-
6               5c5a709cdcb8
7
8 Type:  kubernetes.io/service-account-token
9
10 Data
11 ====
12 ca.crt:     1111 bytes
13 namespace:  7 bytes
14 token:
eyJhbGciOiJSUzI1NiIsImtpZCI6IjVWWWRld2FCX0tLbGJFZFdBd3JBcnCxRmxQRnBGMTFaRDZfM2FJWTV
ra0kifQ.

```

Secret 条目的内容会以 Base64 编码

5: 挂载 Secret 卷:

```

1 kind: Pod
2 spec:
3   containers:
4     - image: nginx:alpine
5       name: web-server
6       volumeMounts:
7         - name: certs
8
9           # 挂载证书的路径
10          mountPath: /etc/nginx/certs/
11          readOnly: true
12
13         ports:
14           - containerPort: 80
15           - containerPort: 443
16
17         volumes:
18
19           # 引用 secret 卷
20           - name: certs
21             secret:
22               secretName: fortune-https

```

6: k8s 允许通过环境变量暴露 Secret, 但是不安全。

```
1 env:
2   -name: FOO_SECRET
3   valueFrom:
4     secretKeyRef:
5       # secret 资源名称
6       name: fortune-https
7       # 引用的条目
8       key:foo
```

7: 从 Docker Hub 网站拉取私有 Docker 镜像仓库时，需用 Secret 鉴权

```
1 $ kubectl create secret docker-registry mydockerhubsecret \
2   --docker-username=myusername --docker-password=mypassword \
3   --docker-email=my.email@provider.com
```

8: Pod 中使用 Secret。

```
1 kind: Pod
2 metadata:
3   name: private-pod
4 spec:
5   # 引用 Secret
6   imagePullSecrets:
7     - name: mydockerhubsecret
8   containers:
9     - image: username/private:tag
10    name: main
```

若运行大量 pod，可通过将 Secret 添加到 ServiceAccount，所有 pod 只要使用了 sa，都能自动添加上镜像拉取的 Secret.

## 从应用访问 pod 元数据及其它资源

### Downward API

1: 传递不能提前知道的数据，如 pod 的 IP、主机名

或者在别处预定义的数据，如 pod 标签和注解。

2: Downward API 不像 REST Endpoint 那样需要通过访问的方式获取数据；

通过将 pod 中取得的数据作为环境变量 或 文件的值（Downward API 卷）对外暴露。

注意：标签和注解只能通过卷暴露

3: Downward API 可以给容器传递的元数据有：

- pod 的名称
- pod 的 IP
- pod 所在的命名空间

- pod 运行节点的名称
- pod 运行所归属的服务账户的名称
- 每个容器请求的 CPU 和内存的使用量
- 每个容器可以使用的 CPU 和内存的限制
- pod 的标签
- pod 的注解

4: 使用元数据作为环境变量:

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4      name: downward
5  spec:
6      containers:
7          - name: main
8              image: busybox
9              command: ["sleep", "9999999"]
10             resources:
11                 requests:
12                     cpu: 15m
13                     memory: 100Ki
14                 limits:
15                     cpu: 100m
16                     memory: 4Mi
17
18         # 设置环境变量
19         env:
20             - name: POD_NAME
21                 valueFrom:
22                     fieldRef:
23                         fieldPath: metadata.name # 应用 pod 中的元数据名称字段
24             - name: POD_NAMESPACE
25                 valueFrom:
26                     fieldRef:
27                         fieldPath: metadata.namespace
28             - name: POD_IP
29                 valueFrom:
30                     fieldRef:
31                         fieldPath: status.podIP
32             - name: NODE_NAME
33                 valueFrom:
34                     fieldRef:
35                         fieldPath: spec.nodeName
36             - name: SERVICE_ACCOUNT
37                 valueFrom:
38                     fieldRef:
39                         fieldPath: spec.serviceAccountName
40             - name: CONTAINER_CPU_REQUEST_MILLICORES

```

```

41     valueFrom:
42         # 注意: CPU 和 内存的引用使用 resourceFieldRef, 而非 fieldRef
43         resourceFieldRef:
44             resource: requests.cpu
45             # 基数单位是 1 毫核, 1/1000 核
46             divisor: 1m
47     - name: CONTAINER_MEMORY_LIMIT_KIBIBYTES
48         valueFrom:
49             resourceFieldRef:
50                 resource: limits.memory
51                 divisor: 1Ki

```

对应关系如下:

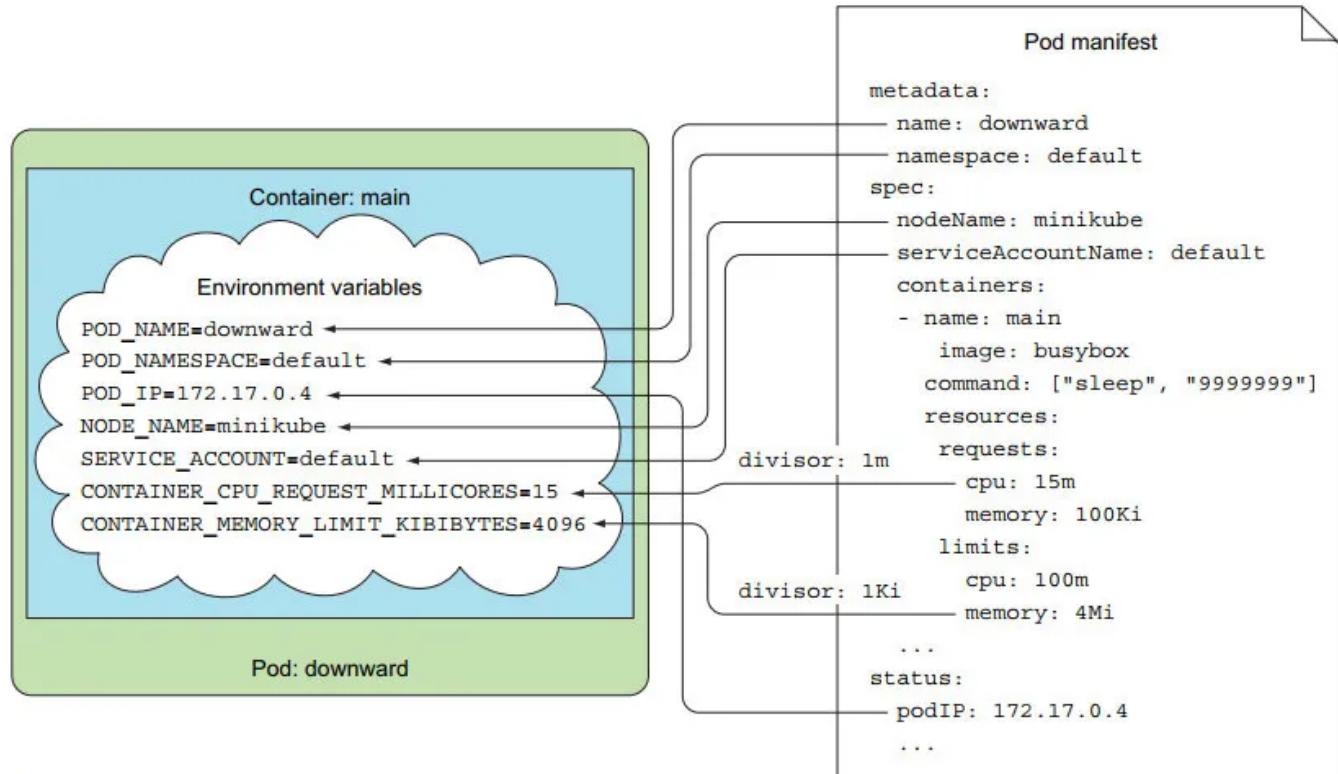


image-20210616211558165

## 5: 挂载 downwardAPI 卷暴露 【推荐】

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: downward
5
6   # 将会在 Downward 卷暴露的元数据
7   labels:
8     foo: bar
9   annotations:
10    key1: value1
11    key2: |
12      multi

```

```
13     line
14     value
15 spec:
16   containers:
17     - name: main
18       image: busybox
19       command: ["sleep", "9999999"]
20     resources:
21       requests:
22         cpu: 15m
23         memory: 100Ki
24       limits:
25         cpu: 100m
26         memory: 4Mi
27     volumeMounts:
28     - name: downward
29       # downwardAPI 挂载的目录
30       mountPath: /etc/downward
31 volumes:
32 - name: downward
33   downwardAPI:
34     items:
35       # pod 的名称会写入文件 /etc/downward/podName
36     - path: "podName"
37       fieldRef:
38         fieldPath: metadata.name
39
40       # 写入文件 /etc/downward/podNamespace
41     - path: "podNamespace"
42       fieldRef:
43         fieldPath: metadata.namespace
44
45       # 写入文件 /etc/downward/labels
46     - path: "labels"
47       fieldRef:
48         fieldPath: metadata.labels
49
50       # 写入文件 /etc/downward/annotations
51     - path: "annotations"
52       fieldRef:
53         fieldPath: metadata.annotations
54
55       # 写入文件 /etc/downward/containerCpuRequestMilliCores
56     - path: "containerCpuRequestMilliCores"
57       resourceFieldRef:
58         # 资源相关的必须指定容器名称, 因为卷是 pod 级别的
59         containerName: main
60         resource: requests.cpu
61         divisor: 1m
```

```

62
63     # 写入文件 /etc/downward/containerMemoryLimitBytes
64     - path: "containerMemoryLimitBytes"
65         resourceFieldRef:
66             containerName: main
67             resource: limits.memory
68             divisor: 1

```

文件列表的对应关系：

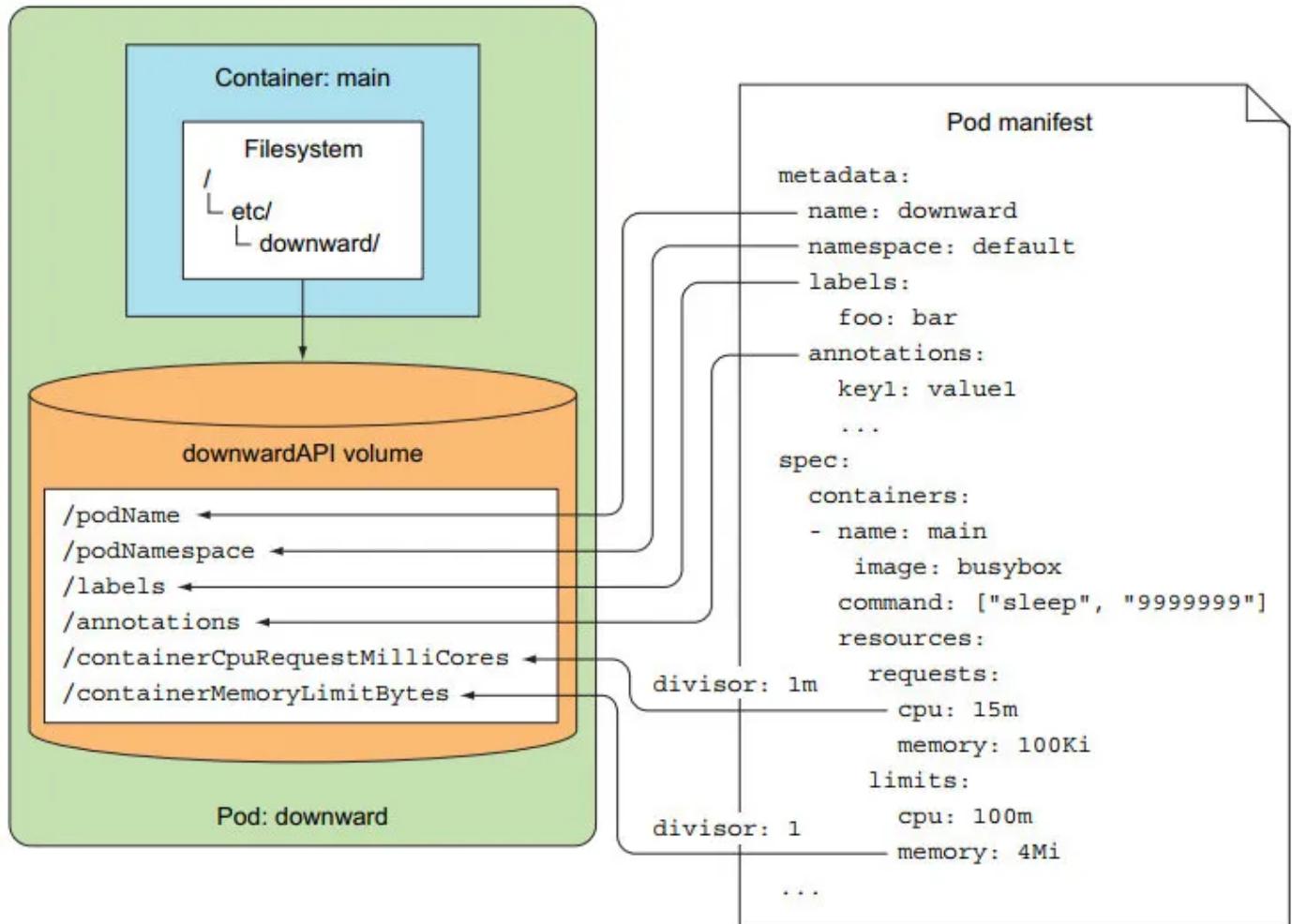


image-20210616212130889

6: 推荐用卷的方式暴露，在运行时修改 注解 或 标签， k8s 会更新相关的文件；

且卷能在 同 pod 的多容器间传递；

但环境变量一旦设置，修改后无法暴露新值。

7: downward API 只能对 pod 内的容器暴露数据，有一定的局限性；

## 与 k8s API 服务器交互

1: k8s API 可获取其它 pod 或集群中其它资源信息；

2: 可以通过 `kube proxy` 启动一个代理服务接收本地的 http 连接并转发至 API 服务器。

同时处理身份认证，所以不需要每次请求都上传认证凭证。它也可以确保我们直接与真实的 API 服务器交互，而不是一个中间人（通过每次验证服务器证书的方式）

● kube proxy 会捎带 API 服务器的 URL、认证凭证等。

代理服务器在本地的 8001 端口接收请求。

```
1 $ curl localhost:8001
2 {
3   "paths": [
4     "/api",
5     "/api/v1",
```

3：可以通过 URL 查看对应资源的 API，例如 Job 资源的 API，分组在 /apis/batch

例如：

image-20210616213620233

4：从 pod 内部访问 API 服务器，需要带上凭证和授权：

image-20210616213916792

● 后续可通过 ServiceAccount 和 RBAC 解决账户和授权的问题。

5：Pod 与 API 服务器的交互如下：

- 应用应该验证 API 服务器的证书是否是证书机构所签发，这个证书是在 ca.crt 文件中。
- 应用应该将它在 token 文件中持有的凭证通过 Authorization 标头来获得 API 服务器的授权。
- 当对 pod 所在命名空间的 API 对象进行 CRUD 操作时，应该使用 namespace 文件来传递命名空间信息到 API 服务器

CRUD 代表创建、读取、修改和删除操作，与之对应的 HTTP 方法分别是 POST、GET、PATCH/PUT 以及 DELETE。

可在 pod 中运行一个 sidecar 容器（代理服务器）；

- 主容器通过端口访问代理容器【同一网络命名空间】
- 代理服务器运行 kubectl-proxy 命令，实现和 API 服务器通信；

image-20210616214346169

代理容器如下，

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: curl-with-ambassador
5 spec:
6   containers:
7     - name: main
8       image: curlimages/curl:7.77.0
9       command: ["sleep", "9999999"]
10    - name: ambassador
11      # 代理容器, 在该容器中, 可通过 curl localhost:8001 访问 API 服务器
12      image: luksa/kubectl-proxy:1.6.2

```

由于主容器和 sidecar 容器共享网络命名空间，直接可以访问代理服务器的 8001 端口。

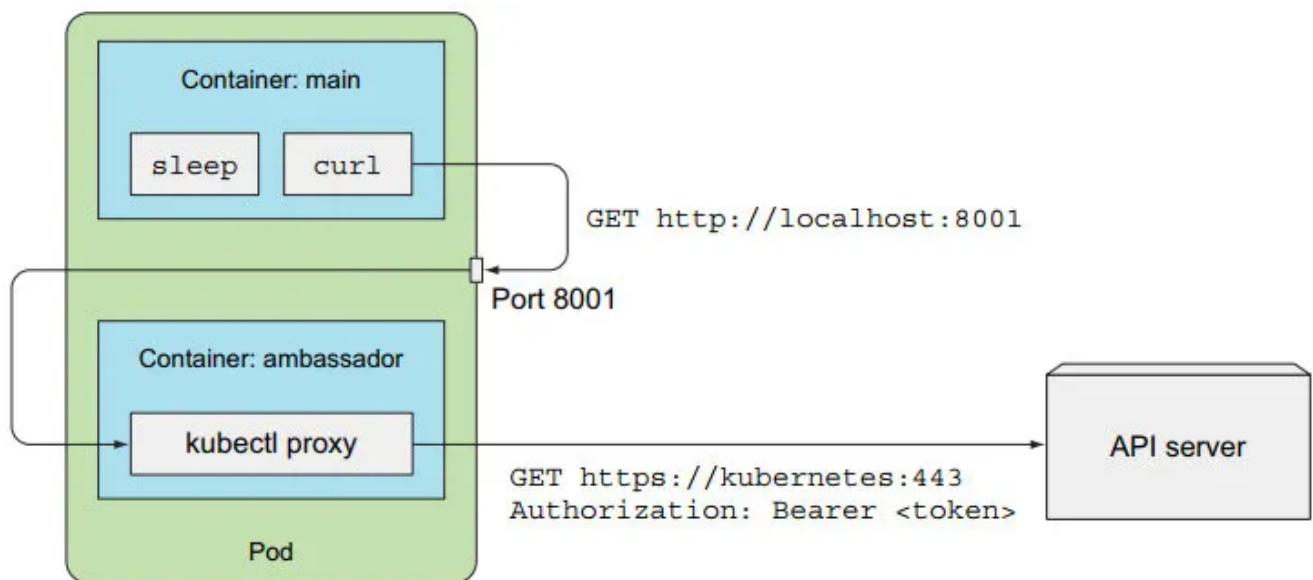
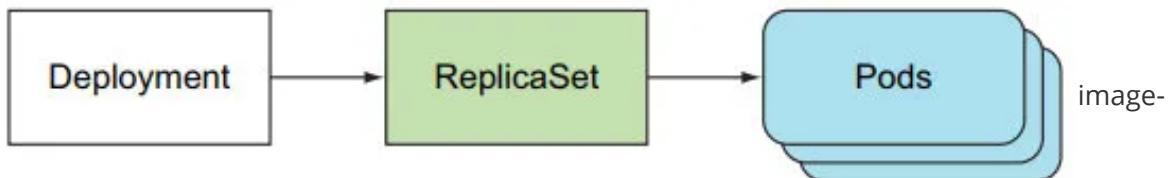


image-20210616214821893

6：也可使用标准的客户端库与 API 服务器交互。

## Deployment

1: Deployment 是基于 ReplicaSet 资源，声明式的升级应用 【滚动更新】



20210616202200622

Deployment 在上层控制期望的状态，更新时，会创建新的 ReplicaSet 资源

2: 通常有两种更新方式：

- 直接删除所有现有的 pod, 然后创建新的 pod。

- 停机更新，短时间不可用

使用 ReplicationController 管理，修改 replicas 副本数即可。

image-20210616201348186

- 也可以先创建新的 pod，并等待它们成功运行之后，再删除旧的 pod。可以先创建所有新的 pod，然后一次性删除所有旧的 pod，或者按顺序创建新的 pod，然后逐渐删除旧的 pod。

- 需要支持两个版本同时对外提供服务

第一种方法：先创建新的，可用后，一次性删除旧的 【蓝绿部署】：

image-20210616201709906

第二种方法：手动执行滚动升级 【旧的副本数在减少，新的副本数在增加。】：

image-20210616201824859

3：创建一个 Deployment 示例：

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4      name: kubia
5  spec:
6      # 目标副本数
7      replicas: 3
8
9      # 新的 pod 模板
10     template:
11         metadata:
12             name: kubia
13             labels:
14                 app: kubia
15         spec:
16             containers:
17                 - image: luksa/kubia:v1
18                     name: nodejs
19
20     # 需要更新的 pod
21     selector:
22         matchLabels:
23             app: kubia

```

4：Deployment 提供两种升级策略：

- 1：滚动更新， RollingUpdate;

升级过程中速度可控：`minReadySeconds` 可控制滚动更新的速度

`minReadySeconds` 要求 pod 至少运行多久才算可用。在新 pod 可用之前，`maxUnavailable` 会卡住更新。

若在 minReadySeconds 时间内，就绪探针失败， 新版本滚动会停止。

pod 未就绪时，新的请求也不会分发到上面。

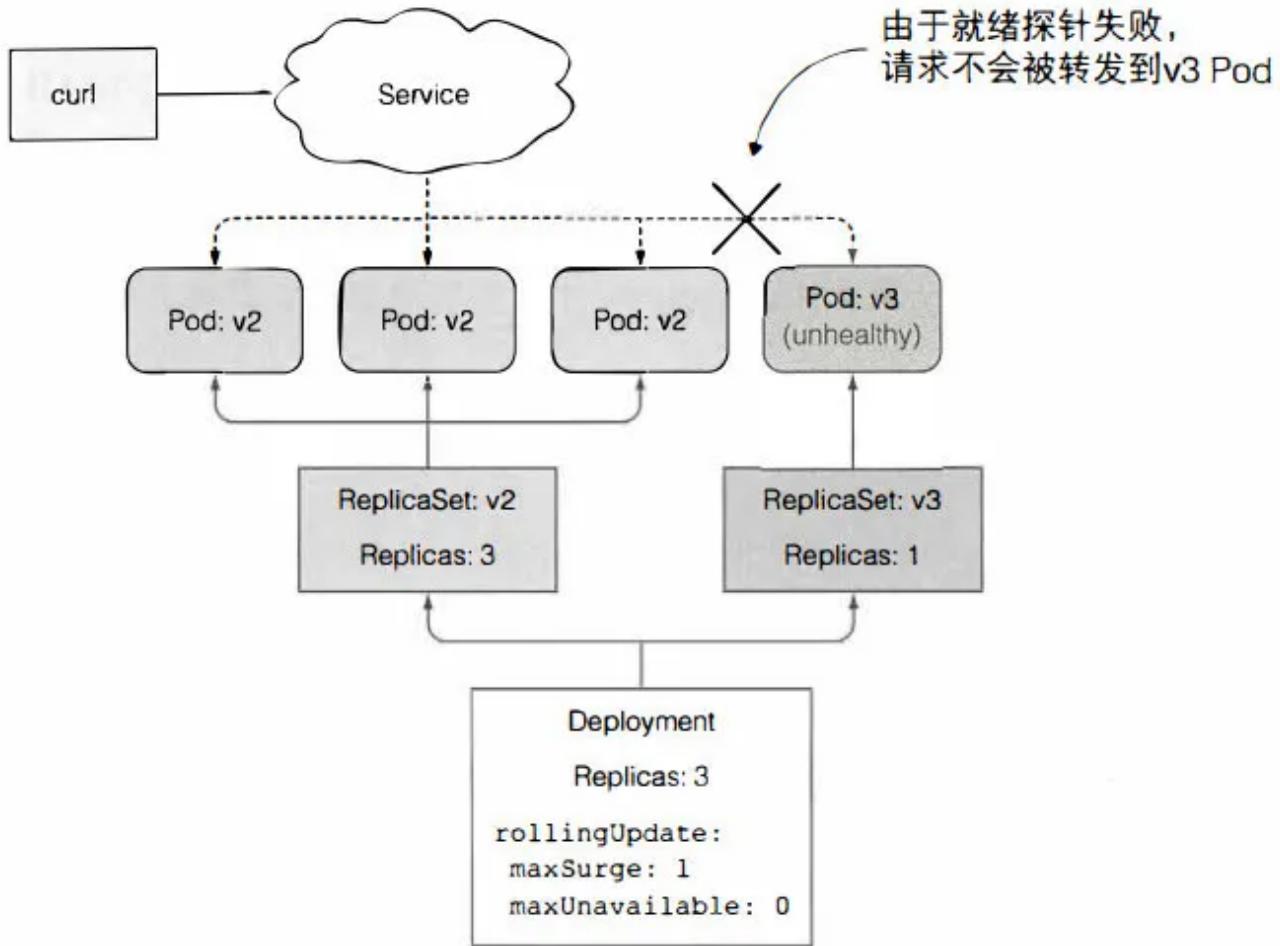


image-20210616204948017

触发更新如下， 直接设置新的镜像版本即可：

```
1 kubectl set image deployment/replicationcontroller/replicaset <资源名> <container-name>=<ID>/<image>:<tag>: 修改 资源下容器里的 镜像，会修改 pod 模板；并 触发 Deployment 的滚动升级
```

- 2: 一次性删除所有旧的，然后创建新的, Recreate

5: Deployment 升级完后，旧的 ReplicaSet 还会保留，方便出错之后回滚。

image-20210616203617762

可以回滚到指定的版本，保留的历史版本数 通过 `revisionHistoryLimit` 属性控制：

image-20210616203753923

6: 控制滚动 更新速度的两个参数：

- `maxSurge`: 某一时刻，最多运行的 pod 数量（包括 新版本下的 pod 和旧版本下的 pod 数总和）， 控制 每次新版本 增加的 pod 数
- `maxUnavailable`: 滚动期间，最大不可用 pod 数量， 控制必须得保留一定量 旧版本的 pod 树；

image-20210616204222968

假设集群目标数量是 3, maxSurge 允许最多 pod 数量达到 4, 同时 maxUnavailable = 0 (任意时刻, 必须至少有 3 个可用)

3 个副本数下的更新过程:

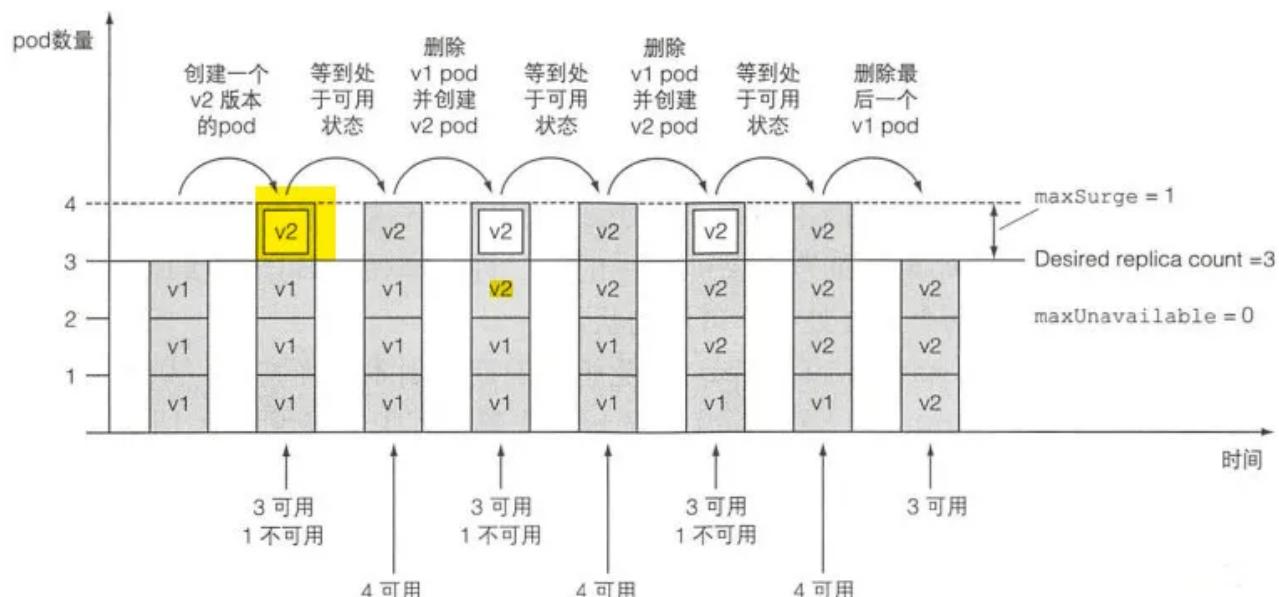


image-20210616204359032

7: 升级过程中, 可以暂停滚动升级 (发布金丝雀版本进行测试), 但在哪个时刻暂停无法控制。

功能验证后, 可恢复滚动升级。

8: 可以通过 `progressDeadlineSeconds` 属性指定滚动升级必须在 多长时间 内 完成, 否则视为失败。

## 部署有状态多副本

1: 有状态服务通常需要考虑:

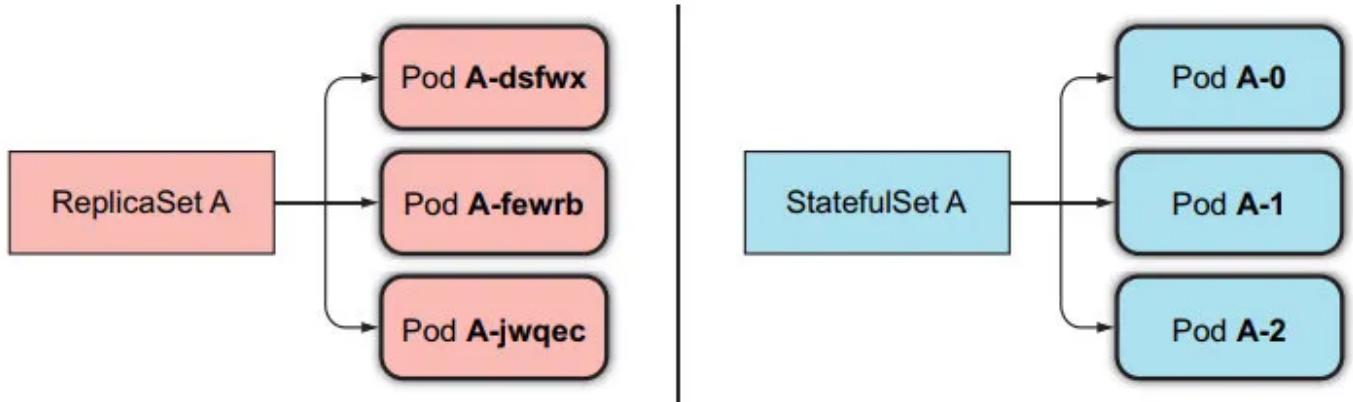
- 有状态服务需要有独立的存储:
- 不变的主机名和 ip 地址

## StatefulSet

1: StatefulSet 可以保证重启一个 pod 实例, 拥有和之前一样的 名称 和 主机名。

2: StatefulSet 可以保证每个 pod 都有稳定的名字和状态;

- RS 分配的主机名都是随机的 (默认平等)
- StatefulSet 分配主机名是按顺序递增的



3: pod 重启后，并不保证在原来的节点上：

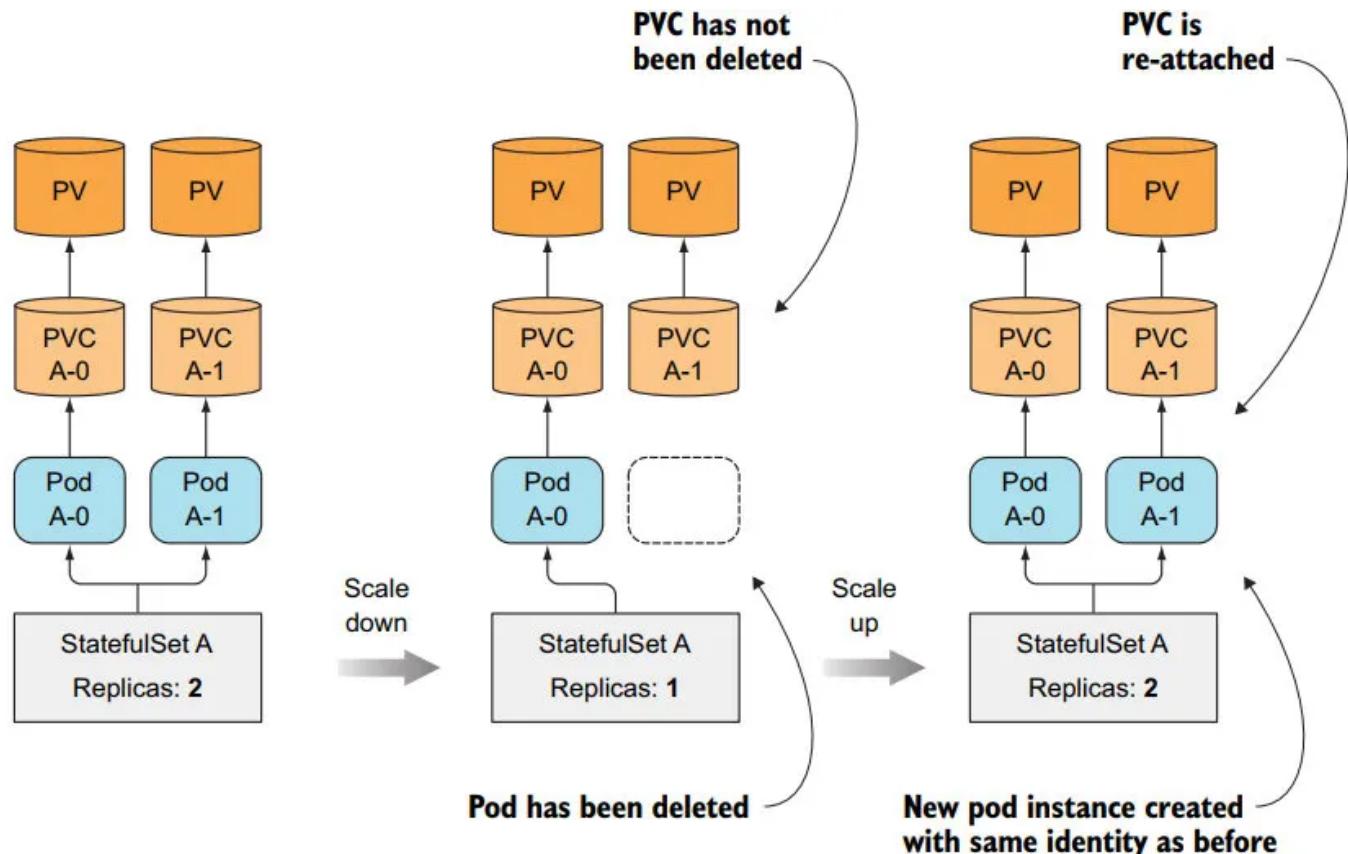
但主机名和 ip 保持不变

4: StatefulSet 缩容时，会优先删除高索引主机名的实例（如下，第一次缩容，Pod A-2 最先被删除。）

5: 为每个 pod 声明单独的 PVC，提供独立的存储

缩容时，只会删除 pod，PVC 默认不会删除（除非手动），当下次重新扩容时，会绑定到之前的 PVC

可以保证扩容出来的 pod 还是写相同的文件。



6: StatefulSet 实例，创建三个对象

- 创建 PV

1 | kind: List

```

2  apiVersion: v1
3  items:
4  - apiVersion: v1
5    kind: PersistentVolume
6    metadata:
7      name: pv-a
8    spec:
9      capacity:
10        storage: 1Mi
11      accessModes:
12        - ReadWriteOnce
13      persistentVolumeReclaimPolicy: Recycle
14      gcePersistentDisk:
15        pdName: pv-a
16        fsType: ext4
17 - apiVersion: v1
18   kind: PersistentVolume
19   metadata:
20     name: pv-b
21   spec:
22     capacity:
23       storage: 1Mi
24     accessModes:
25       - ReadWriteOnce
26     persistentVolumeReclaimPolicy: Recycle
27     gcePersistentDisk:
28       pdName: pv-b
29       fsType: ext4
30 - apiVersion: v1
31   kind: PersistentVolume
32   metadata:
33     name: pv-c
34   spec:
35     capacity:
36       storage: 1Mi
37     accessModes:
38       - ReadWriteOnce
39     persistentVolumeReclaimPolicy: Recycle
40     gcePersistentDisk:
41       pdName: pv-c
42       fsType: ext4

```

- 创建一个控制 Service

创建一个 headless Service, 可以让 pod 之间彼此发现。

通过这个 Service, 每个 pod 都有 独立的 DNS 记录, 可以通过主机名方便的找到它。

```

1  apiVersion: v1
2  kind: Service

```

```

3   metadata:
4     name: kubia
5
6   spec:
7     # StatefulSet 的控制 Service 必须是 headless 模式
8     clusterIP: None
9
10    selector:
11      app: kubia
12    ports:
13      - name: http
14        port: 80

```

可以通过在 pod 中触发一次 SRV DNS 查询，获取其它 pod 列表

有时不需要或不想要负载均衡，以及单独的 Service IP。遇到这种情况，可以通过指定 Cluster IP (`spec.clusterIP`) 的值为 `"None"` 来创建 Headless Service。

你可以使用无头 Service 与其他服务发现机制进行接口，而不必与 Kubernetes 的实现捆绑在一起。

对这无头 Service 并不会分配 Cluster IP，`kube-proxy` 不会处理它们，而且平台也不会为它们进行负载均衡和路由。DNS 如何实现自动配置，依赖于 Service 是否定义了选择算符。

### 带选择算符的服务

对定义了选择算符的无头服务，Endpoint 控制器在 API 中创建了 Endpoints 记录，并且修改 DNS 配置返回 A 记录（IP 地址），通过这个地址直接到达 Service 的后端 Pod 上。

### 无选择算符的服务

对没有定义选择算符的无头服务，Endpoint 控制器不会创建 Endpoints 记录。然而 DNS 系统会查找和配置，无论是：

- 对于 `ExternalName` 类型的服务，查找其 CNAME 记录
- 对所有其他类型的服务，查找与 Service 名称相同的任何 Endpoints 的记录
- StatefulSet 本身

```

1   apiVersion: apps/v1
2   kind: StatefulSet
3
4   metadata:
5     name: kubia
6
7   spec:
8     serviceName: kubia
9
10    # 创建2个副本，pod 上带有 标签 app: kubia
11    replicas: 2
12    selector:
13      matchLabels:
14        app: kubia # has to match .spec.template.metadata.labels

```

```

15     labels:
16       app: kubia
17   spec:
18     containers:
19       - name: kubia
20         image: luksa/kubia-pet
21         ports:
22           - name: http
23             containerPort: 8080
24         volumeMounts:
25           - name: data
26             mountPath: /var/data
27
28 # 创建 pvc 模板, 生成的pvc data-<主机名>
29 volumeClaimTemplates:
30   - metadata:
31     name: data
32     spec:
33       resources:
34         requests:
35           storage: 1Mi
36       accessModes:
37         - ReadWriteOnce

```

7: StatefulSet 对集群同时启动两个 pod 非常敏感，可能存在竞争。

依次启动 是比较安全可靠的：所以后面的 pod 会等待前面的 pod 成为就绪状态后 创建。

8: StatefulSet 修改模板文件后，不会自动触发运行的 pod 更新。

和 ReplicaSet 一样，重启更新。

9: 当 pod 突然失效时 (NotReady) , StatefulSet 不会立刻驱逐. 需要等足够多的时间，或者显示的删除该 pod, 才能触发 重新调度。

简言之， StatefulSet 会避免同时运行两个一样的 pod.

## kubernetes 机理

### 架构

1: k8s 集群分两部分：

- master node (The k8s Control Plane, 控制面板): 存储和管理集群的状态
  - etcd 分布式持久化存储
  - API 服务器
  - 调度器
  - 控制器管理器
- work node
  - Kubelet

- Kubelet 服务代理( kube-proxy)
- 容器运行时(Docker、rkt 或者其他)
- 附加组件
  - KubernetesDNS 服务器：通过 IP 对外暴露 HTTP 服务；
  - 仪表板
  - Ingress 控制器：对客户端 ip 保存，后续多次连接路由到同一个 pod
  - Heapster(容器集群监控)
  - 容器网络接口插件

整体组件如下：

1 | `kubectl get componentstatus` : 显示控制面板各个组件的状态

2: 系统组件只能通过 API 服务器进行通信，相互之间不同通信。

3: etcd 和 API 服务器可以有多个实例 同时并行工作。【分布式集群】

为了保证集群一致性，采用 RAFT 算法。

但 调度器 和 控制器 在某一个时刻，只能有一个 实例起作用，其它实例处于 待命状态。

4: kubelet 是唯一一直作为常规系统运行的组件。

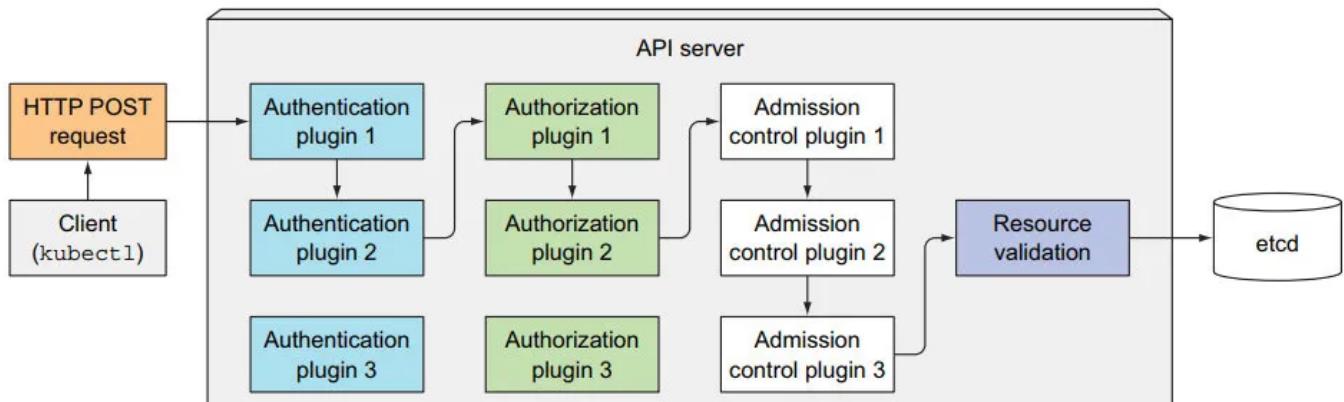
5: **API 服务器**，可查询、修改集群状态的 CRUD (Create、Read、Update、Delete)，并最终存入 etcd .

对请求的 Rest 进行校验。

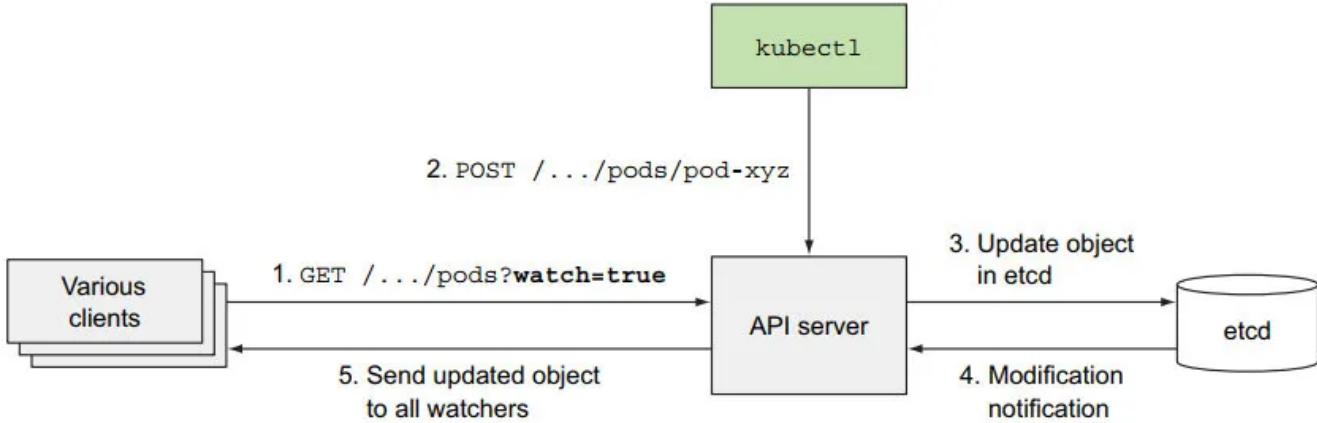
- Authentication plugin: 认证插件，获取用户、用户组 等信息；
- Authorization plugin: 授权插件，是否有权限对指定资源进行 指定的操作；
- Admission Control plugin: 准入插件控制，例如 资源限制 ResourceQuota 等

准入控制插件包括：
 

- AlwaysPullImages: 重写 pod 的 imagePullPolicy 为 Always, 强制每次部署 pod 时拉取镜像。
- ServiceAccount: 未明确定义服务账户的使用默认账户。
- NamespaceLifecycle: 防止在命名空间中创建正在被删除的 pod, 或在不存在的命名空间中创建 pod。
- ResourceQuota: 保证特定命名空间中的 pod 只能使用该命名空间分配数量的资源，如 CPU 和内存。



控制器可通过定期的去拉取 API 服务器信息，监听资源的变化。



## 6: 调度器: 为 pod 选择合适的节点

- 最简单的是随机选择一个；
- 以更优的方式选择一个， 对所有节点按优先级排序，找出最优节点（若分数一致，则循环分配）

筛选哪些节点可用：

- 节点是否能满足 pod 对硬件资源的请求。
- 节点是否耗尽资源（是否报告过内存 / 硬盘压力参数）？
- pod 是否要求被调度到指定节点（通过名字），是否是当前节点？
- 节点是否有和 pod 规格定义里的节点选择器一致的标签（如果定义了的话）？
- 如果 pod 要求绑定指定的主机端口（第 13 章中讨论）那么这个节点上的这个端口是否已经被占用？
- 如果 pod 要求有特定类型的卷，该节点是否能为此 pod 加载此卷，或者说该节点上是否已经有 pod 在使用该卷了？
- pod 是否能够容忍节点的污点，涉及污点和容忍度。
- pod 是否定义了节点、pod 的亲缘性以及非亲缘性规则？如果是，那么调度节点给该 pod 是否会违反规则？

集群中可运行多个调度器而非单个，设置 `schedulerName` 来调度特定的 pod

## 7: 控制器：控制集群服务器的状态朝 API 服务器定义的期望状态收敛。

控制器包括

- Replication 管理器 (ReplicationController 资源的管理器)：监听 pod 的数量
- ReplicaSet、DaemonSet 以及 Job 控制器：部署和维护 pod
- Deployment 控制器：控制滚动更新，每个版本，都会创建一个 ReplicaSet
- StatefulSet 控制器：有状态服务的管理，挂载到相同的 PV，相同的 IP 和主机名
- Node 控制器：管理 Node 资源，监控每个 Node 的健康状态；
- Service 控制器：网络管理相关组件，创建或删除 LoadBalancer 类型服务；
- Endpoints 控制器：从 Service 的 pod 选择器中选出指定的 pod，并将 IP 和端口更新到 Endpoint 资源中；
- Namespace 控制器：创建或删除 Namespace 对象；
- PersistentVolume 控制器：创建一个 PVC 后，由该控制器找到一个合适的 PV 绑定。【存储量大于 PVC 声明的最小 PV】
- 其他

控制器就是活跃的 Kuberetes 组件，去做具体工作部署资源。

控制器通过监听 API 资源，作出相应的调整，如更新、删除已有对象。

控制器之间不会直接通信，每个控制器都会连接到 API 服务器。

8: kubelet: 监控 API 服务器 是否在当前节点 新分配了 pod, 告知 容器运行时(如 Docker) 运行容器。

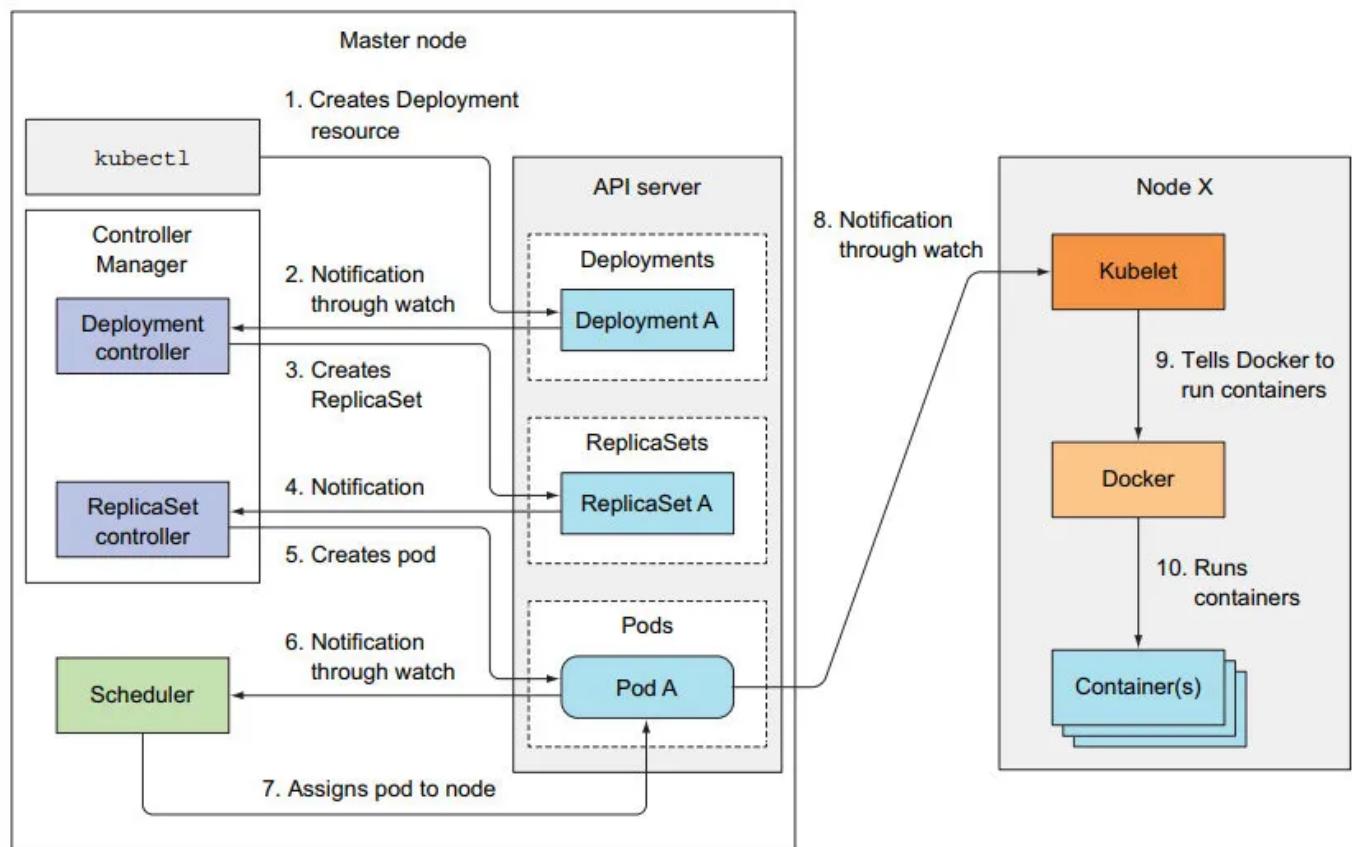
9: kube-proxy: 通过 修改 iptables , 将请求重定向到 服务器。

userspace 代理模式如下, 对每个进来的连接, 代理到一个 pod

现在是 默认的 iptables 模式。

10: 控制器之间是相互协作的, 通过监听 API 服务器来判断 是否要创建 / 删除 资源。

如下 是创建一个 Deployment 资源的事件链:



11: 在每个 pod 中会有一个 基础容器 (处于 pending 状态) , 用于保存 Linux 命名空间, 当容器被重启时, 需要保持和之前的命名空间一样, 这个 pod 就发挥了作用。

## 网络通信

1: 同节点的 pod 之间通信。

基础容器启动前, 会为容器创建一个 虚拟 Ethernet 接口对 (veth pair) :

- 一端在 node 节点的命名空间中: vethXXXX
- 一端在容器网络命名空间中: eth0

只要连接到 同一 网桥, 相互之间就能通信。

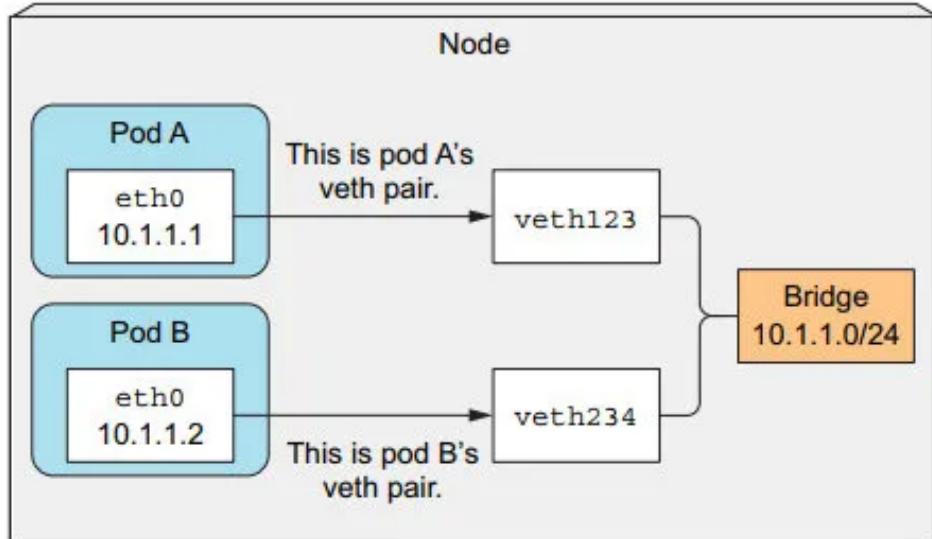


image-20210616165429183

2: 不同节点的 pod 之间通信，两个节点之间需要连接网桥

连接网桥的方式有：

- overlay
- underlay 网络
- 常规的三层路由

跨节点网桥必须使用 非重叠地址段，保证不同的 pod 有不同的 IP

以下节点需要配置路由 或者 连接到相同网关 【中间无路由】，否则会因为 pod IP 是局域私有的，会丢包。

使用 SDN (软件定义网络) 技术，可以忽略底层网络拓扑，所有节点就像连接到同一个网关；

pod 发出的报文会被封装，到达其它节点，会被解封装；

## 服务

1：和 Service 相关的操作 都是由 每个 节点上的 kube-proxy 进行处理的。

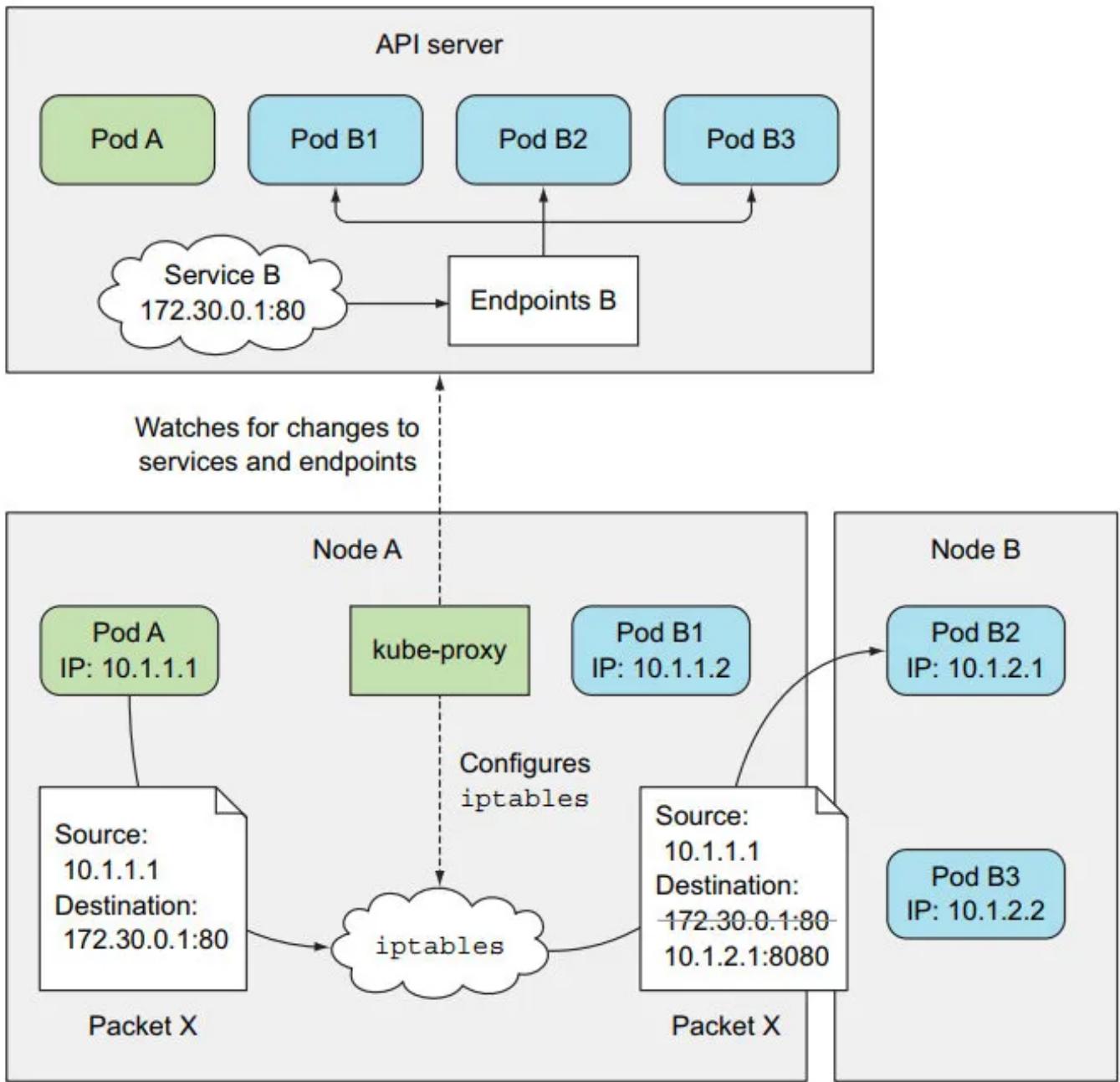
2：每个 service 都有一个稳定的 IP 地址 和端口 对；【针对多端口 Service 有多个 IP: 端口对】

单独的服务 IP 无任何意义，不能 ping

3：创建一个 Service，会发生以下事件链：

- 当创建一个服务时，虚拟 IP 地址会分配给它
- API 服务器 会通知所有 节点上的 kube-proxy, 有个新的服务创建了，修改 iptables, 让服务在字节所在的节点可寻址；【修改目的地址，重定向到其中的一个 pod】
- kube - proxy 还会监听 Endpoint 对象的更改
  - Endpoint 保存了所有 pod 的 ip:端口 对 【每次创建或删除 pod，都会影响 Endpoint】

下图中 pod A 请求 pod B 时，会随机选择一个 pod（假设 pod B2 选中），根据 iptables 规则，目标地址被修改为 pod B2 的 ip:端口



## 高可用集群

- 1: 应用高可用：分布式集群；
- 2: 主节点高可用：部署成集群
  - API server 通过负载均衡器选择，同时并行工作；
  - 控制器和调度器：领导者选举一个运行；

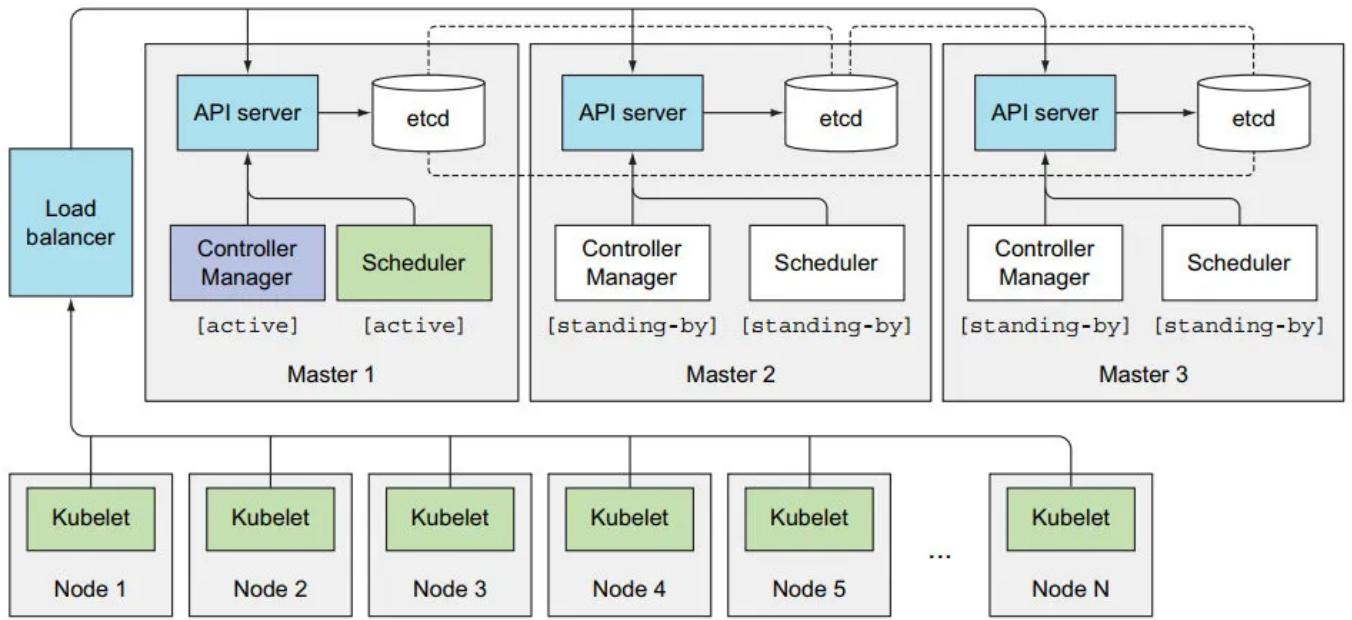


image-20210616172020019

## kubernetes API 服务器的安全防护

### 认证机制

1: pod 与 API 服务器进行通信时，会经过 API 服务器的 **认证插件**

该插件会根据 **证书+token** 或 **HTTP 用户验证** 提取客户端的 **用户名、用户 ID 和组信息**。

2: 连接 API 服务器有两种客户端：

- 真实的用户；
- 运行在 pod 中的应用；

3: 系统内置的组 有特殊含义

- `system:unauthenticated` 组用于所有认证插件都不会认证客户端身份的请求。
- `system:authenticated` 组会自动分配给一个成功通过认证的用户。
- `system:serviceaccounts` 组包含所有在系统中的 ServiceAccount。
- `system:serviceaccounts:<namespace>` 组包含了所有在特定命名空间中的 ServiceAccount。

### ServiceAccount

1: ServiceAccount 也是一种资源(简称 sa), 会为每个命名空间自动创建一个默认的 ServiceAccount

主要用于 客户端身份认证，省去 手动传 token

2: pod 只能使用 同一命名空间的 ServiceAccount

- 可单独使用一个 ServiceAccount
- 也可和同命名空间的 其它 pod 共用 ServiceAccount

3: 可在 pod 中显示指定使用的 ServiceAccount, 否则使用该命名 空间下 默认的 sa.

添加注解 `kubernetes.io/enforce-mountable-secrets= "true"`， 可强制 pod 只允许挂载 ServiceAccount 中的秘钥

4: ServiceAccount 的镜像拉取秘钥， 用于拉取容器镜像的凭证。

注意：所有使用 该 ServiceAccount 的 pod 都会拥有这个秘钥， 而不必每个单独添加。

```
1 apiVersion: v1
2 kind: ServiceAccount
3 metadata:
4   name: my-service-account
5 imagePullSecrets:
6   # 所有使用 该 ServiceAccount 的 pod 都会拥有这个秘钥
7   - name: my-dockerhub-secret
```

5: 在 pod 中使用 serviceAccount

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: curl-custom-sa
5 spec:
6   # 若不显示声明，则使用 默认的 serviceAccount
7   serviceAccountName: foo
8   containers:
9     - name: main
10    image: tutum/curl
11    command: [ "sleep", "9999999" ]
12     - name: ambassador
13    image: luksa/kubectl-proxy:1.6.2
```

ServiceAccount 账户 产生的 Token 在 `/var/run/secrets/kubernetes.io/serviceaccount/` 目录。

未使用 RBAC 授权插件， ==默认的 serviceaccount 和 显示创建的 serviceaccount 都允许 执行任何操作。==

## RBAC

1: RBAC: 基于 角色的权限访问控制插件；

开启 RBAC 后， 未经授权的 serviceAccount 或 默认的 serviceAccount 禁止查看集群状态。

2: RBAC 用来设置一个用户、ServiceAccount 或者一组用户， 控制该角色在特定资源上 能否执行动作的权限， 比如以下动作：

| HTTP 方法  | 单一资源的动词             | 集合的动词            |
|----------|---------------------|------------------|
| GET、HEAD | get (以及 watch 用于监听) | list (以及 watch)  |
| POST     | create              | n/a              |
| PUT      | update              | n/a              |
| PATCH    | patch               | n/a              |
| DELETE   | delete              | deletecollection |

use 动词用于 PodSecurityPolicy 资源。

3: RBAC 授权规则是通过四种资源来进行配置的，它们可以分为两个组

- Role( 角色) 和 ClusterRole (集群角色) ， 它们指定了在资源上可以执行哪些动词。
  - 命名空间 范围内的资源
- RoleBinding (角色绑定) 和 ClusterRoleBinding (集群角色绑定) ， 它们将上述角色绑定到特定的用户、组或 ServiceAccounts 上。
  - 集群级别的 资源

角色定义了可以做什么操作，而绑定定义了谁可以做这些操作

需要注意的是，RoleBinding 也可以引用 不在命名空间中的集群角色。

Role 和 RoleBinding 都在命名空间中，ClusterRole 和 ClusterRoleBinding 不在命名空间中。

4: 启用 RBAC，一旦开启，禁止未授权的 serviceAccount 查看/修改 资源。

`kubectl delete clusterrolebinding permissive-binding`: 重新启用 RBAC

5: Role: 哪些操作可以在 哪些资源上执行。【授权某些资源的操作权限。】

```

1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: Role
3 metadata:
4   # Role 所在的命名空间，若没指定，默认是当前命名空间
5   namespace: foo
6   name: service-reader
7 rules:
8   # service 是核心 apiGroup 资源，无 apiGroup 就是 ""
9   - apiGroups: [""]
10  # 允许执行的操作
11  verbs: ["get", "list"]
12
13  # 该规则和服务相关（复数）
14  resources: ["services"]
```

在本例中，你允许访问所有服务资源，但是也可以通过额外的 `resourceNames` 字段指定服务实例的名称来限制对服务实例的访问。

Role 中的声明，表明能 get 和 list 中的 service 资源。【默认是禁止的】

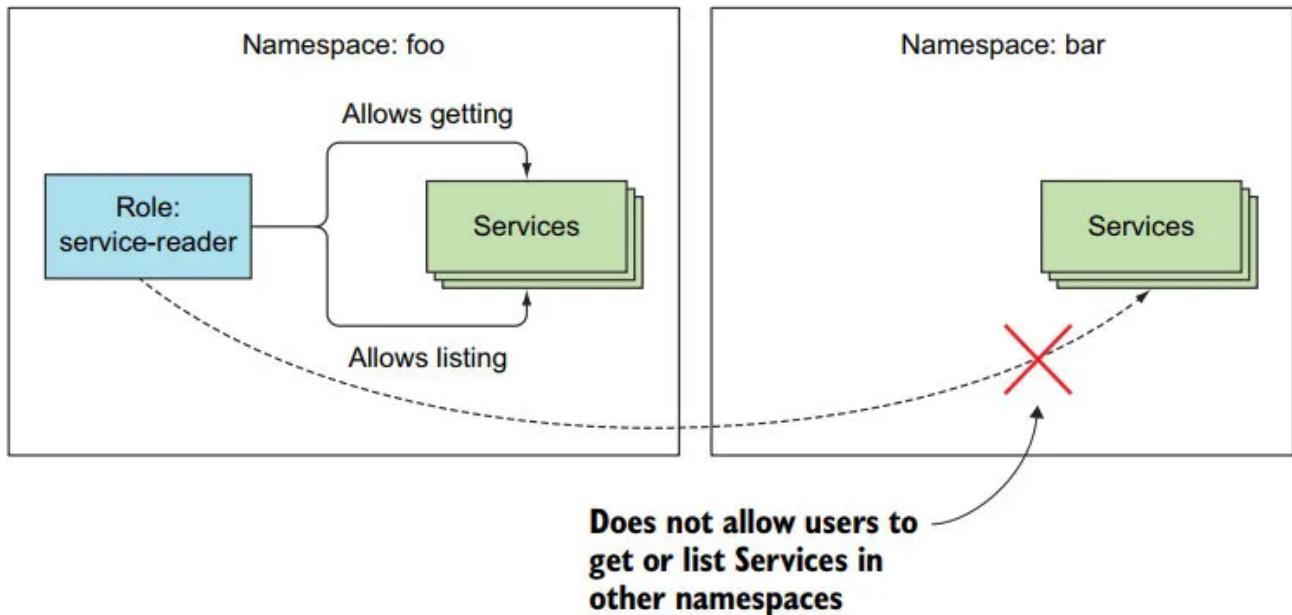


image-20210616143601754

6: rolebinding, 将 Role 中允许的权限 绑定到指定的账户上。

```
1 kubectl create rolebinding <rolebinding-name> --role=<role-name>
  serviceaccount=foo:default -n foo: 创建 rolebinding 资源，绑定到 foo 命名空间中， default 的 ServiceAccount 账号上
```

--user: 绑定到用户

--group: 绑定 Role 到组

RoleBinding 也能绑定到其它命名空间的 serviceAccount。总之，绑定到哪个账户，就赋予哪个账户权限。

如下，RoleBinding 绑定到 bar 命名空间的 sa 账户时，也能查看 services。

7: ClusterRole 和 ClusterRoleBinding 属于集群级别的资源管理

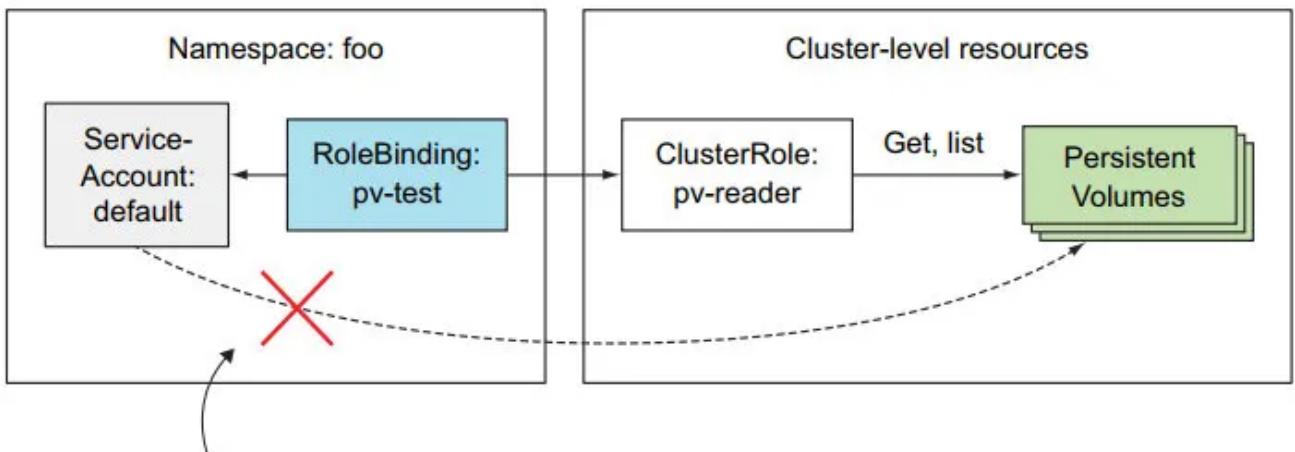
以下两种情况，需要使用集群级别的授权：

- 1: 当允许跨命名空间访问资源时，每次扩展，都需要为新的命名空间添加 Role 和 RoleBinding 【命名空间与命名空间之间会相互绑定】
- 2: 有些资源不在命名空间中: Node、PersistentVolume、Namespace 等
  - 非资源的 URL 路径 (`/healthz`)

默认的 ClusterRole 都以 `system:` 为前缀

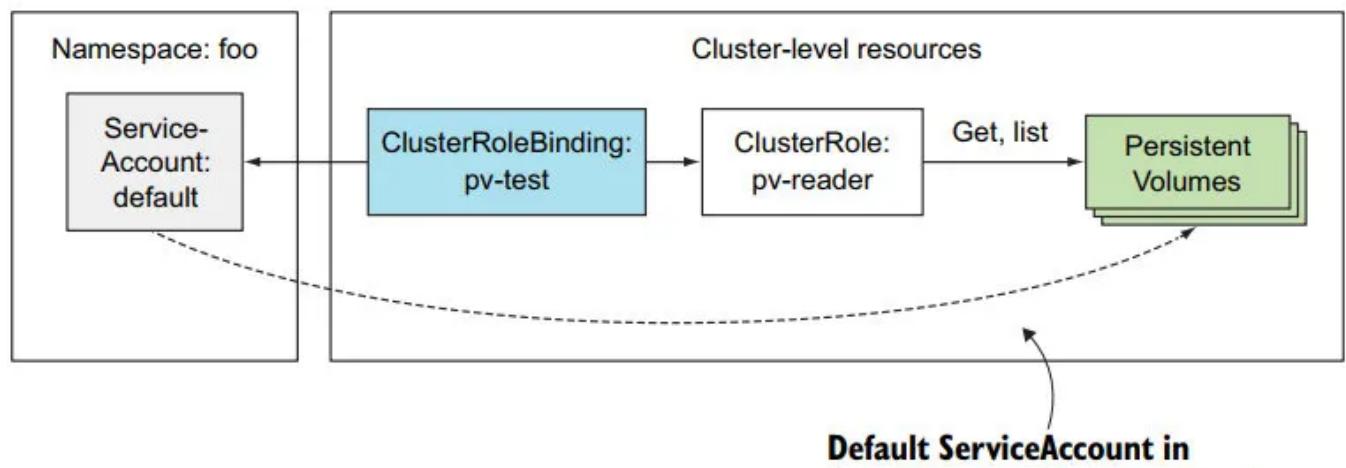
注意：sa 账户可以通过 RoleBinding 绑定到 ClusterRole 上，但是无法访问集群级别的资源。

只能访问指定命名空间中的资源。



**Default ServiceAccount  
is unable to get and list  
PersistentVolumes**

只有 通过 ClusterRoleBinding 才能访问集群级别的资源。



**Default ServiceAccount in  
foo namespace is now allowed  
to get and list PersistentVolumes**

非资源型的 URL 在 ClusterRole 中 使用 的是 URL 路径 而非资源。

通过 ClusterRoleBinding 绑定后，也能访问。

以下是一个 ClusterRole 示例。

特殊的 clusterrole

```
$ kubectl get clusterrole system:discovery -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: system:discovery
  ...
rules:
- nonResourceURLs:
  - /api
  - /api/*
  - /apis
  - /apis/*
  - /healthz
  - /swaggerapi
  - /swaggerapi/*
  - /version
verbs:
- get
```

这条规则指向了非资源型的 URL 而不是资源

对于这些 URL 只有 HTTP GET 方法是被允许的

8: 使用 ClusterRole 授权访问指定命名空间中的资源。

- ClusterRoleBinding --- 绑定 --- ClusterRole: 可以查看所有命名空间、集群中的资源；
- RoleBinding --- 绑定 --- ClusterRole: 只能查看绑定的 RoleBinding 命名空间中的资源

9: Role 和 Binding 的组合，特别注意倒数第二条

## 节点和网络安全

1: pod 可以访问宿主 node 的资源。

### 使用节点的 Linux 命名空间

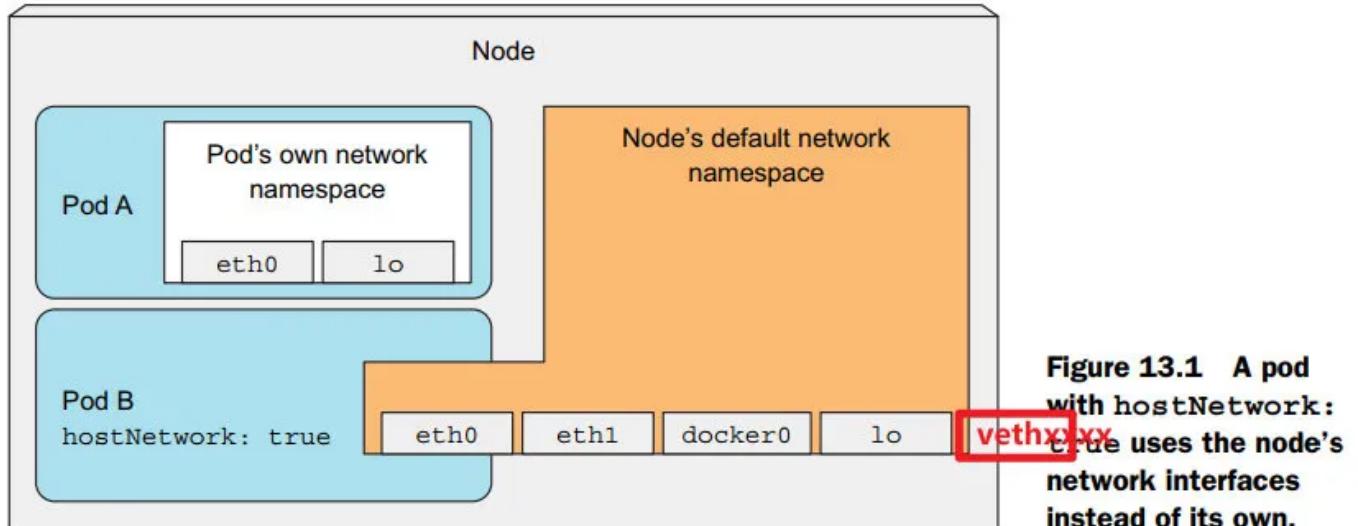
#### 使用节点的网络命名空间和端口

1: 默认每个 pod 拥有自己的 IP 和端口空间。

设置 `pod.spec.hostNetwork:true`, pod 使用节点的网络接口，而无自己的 IP 地址。

端口也会绑定到主节点

2: 如下，pod B 和节点公用 ip: 端口空间，



3: 一般 master 节点上部署的 node 经常会开启 `pod.spec.hostNetwork:true`

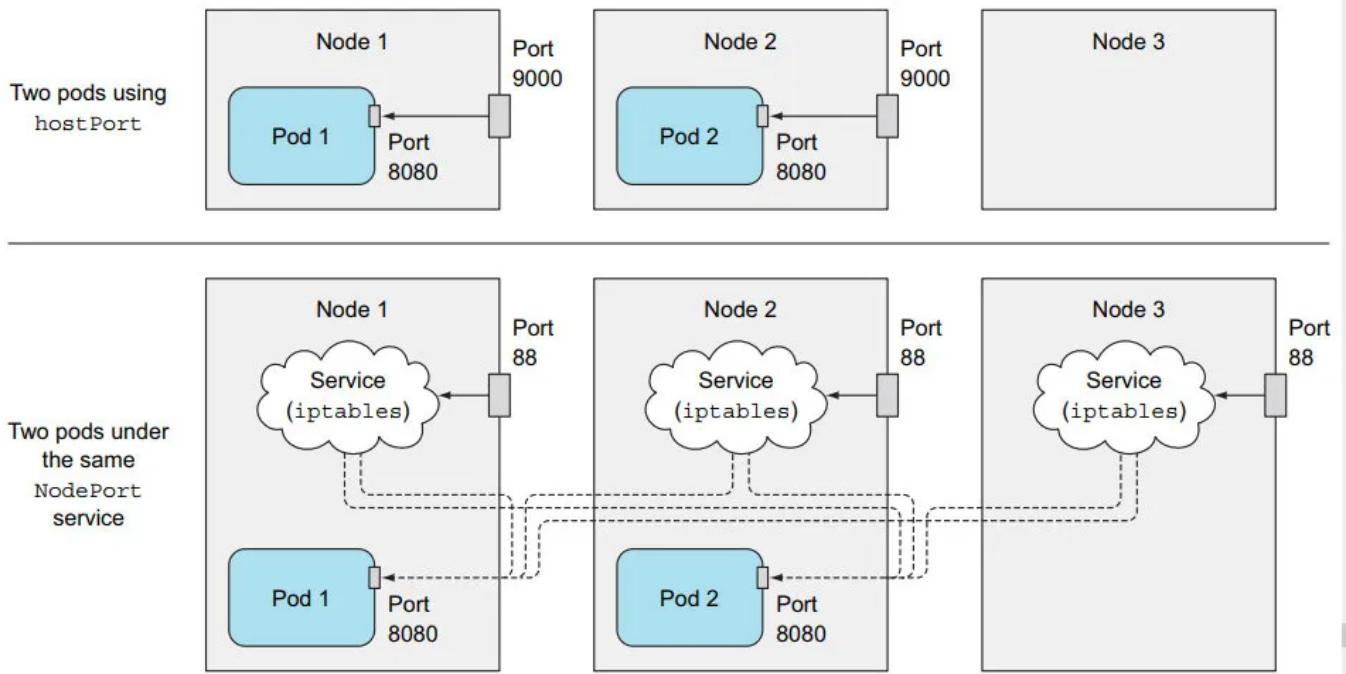
### 仅绑定节点的端口

1: 仅仅绑定节点的端口，而让 pod 继续有自己的网络命名空间。

2: 设置 `pod.spec.hostPort:true`

使用 `hostPort` 和 `NodePort` 有两点不同：

- 到达节点端口的连接
  - `hostPort`: 会直接转发到 pod 对应的端口上；
  - `NodePort`: 随机选择一个 pod
- 作用范围
  - `hostPort`: 仅有运行了 `hostPort` 配置的 pod 才会绑定对应的端口，未运行则不绑定(Node 3)
  - `NodePort`: 集群中的所有节点都会绑定。



3: 若采用 hostPort 公用主机端口，则在一个节点上，一个端口只允许绑定一次。

若在调度的时候，多个 pod 需要绑定到同一端口，则控制器会分散到不同的 节点。

若无足够多的节点，pod 会保持 pending 状态

4: 同样的，设置 `hostPID`: 可共用 Node 的 进程树空间。

`hostIPC`: 可通过 IPC 进行进程间通信

## 配置节点安全上下文

1: 配置节点的安全上下文，可通过 `securityContext` 设置。

配置安全上下文可以允许你做很多事：

- 指定容器中运行进程的用户（用户 ID）。
- 阻止容器使用 root 用户运行（容器的默认运行用户通常在其镜像中指定，所以可能需要阻止容器以 root 用户运行）。
- 使用特权模式运行容器，使其对宿主节点的内核具有完全的访问权限。
- 与以上相反，通过添加或禁用内核功能，配置细粒度的内核访问权限。
  - 修改系统时间
- 设置 SELinux （Security Enhanced Linux， 安全增强型 Linux）边项，加强对容器的限制。
- 阻止进程写入容器的根文件系统
- 同 pod 多容器下，多用户共享存储卷。
  - `fsGroup` 属性，在创建文件时起作用
  - `supplementalGroups` 属性定义了某个用户所关联的额外的用户组。

## PodSecurityPolicy

1: PodSecurityPolicy 是集群级别的资源，限制用户在 pod 中能否使用安全相关的特性。

2: PodSecurityPolicy 可以做的事项：

- 是否允许 pod 使用宿主节点的 PID、IPC、网络命名空间
- pod 允许绑定的宿主节点端口
- 容器运行时允许使用的用户 ID
- 是否允许拥有特权模式容器的 pod
- 允许添加哪些内核功能， 默认添加哪些内核功能， 总是禁用哪些内核功能
- 允许容器使用哪些 SELinux 选项
- 容器是否允许使用可写的根文件系统
- 允许容器在哪些文件系统组下运行
- 允许 pod 使用哪些类型的存储卷

## NetworkPolicy

1: NetworkPolicy 用于限制 pod 与 pod 之间的通信。网络 隔离 组件。

- ingress: 允许访问这些 pod 的源地址；【入向规则，能被 哪些 pod/命名空间下的 pod/IP 段 访问】
- egress: 这些 pod 可以访问的目标地址；【出向规则，该 pod 只能与 哪些 pod 通信】

2: 可选定 pod 的范围：

- 标签选择器 选出的 pod；
- 一个 namespace 中的所有 pod；
- 无类别域间路由 (Classes Inter-Domain Routing, CIDR) 指定的 IP 段；

3: 由于 NetworkPolicy 是网络隔离组件，该命名空间下，pod 无法访问。

```
1 apiVersion: networking.k8s.io/v1
2 kind: NetworkPolicy
3 metadata:
4   name: default-deny
5 spec:
6   # 空的标签选择器 匹配 命名空间内的 所有pod
7   podSelector:
```

4: 即使其他 pod 通过 service 访问，依然会被 NetworkPolicy 隔离。

```
1 apiVersion: networking.k8s.io/v1
2 kind: NetworkPolicy
3 metadata:
4   name: postgres-netpolicy
5 spec:
6   podSelector:
7     # 标签为 app=database 的 pod 设置了访问权限
8   matchLabels:
9     app: database
10  ingress:
```

```

11 - from:
12   # 只对 app=webserver 的 pod 开放了 5432 端口
13 - podSelector:
14   matchLabels:
15     app: webserver
16 ports:
17   - port: 5432

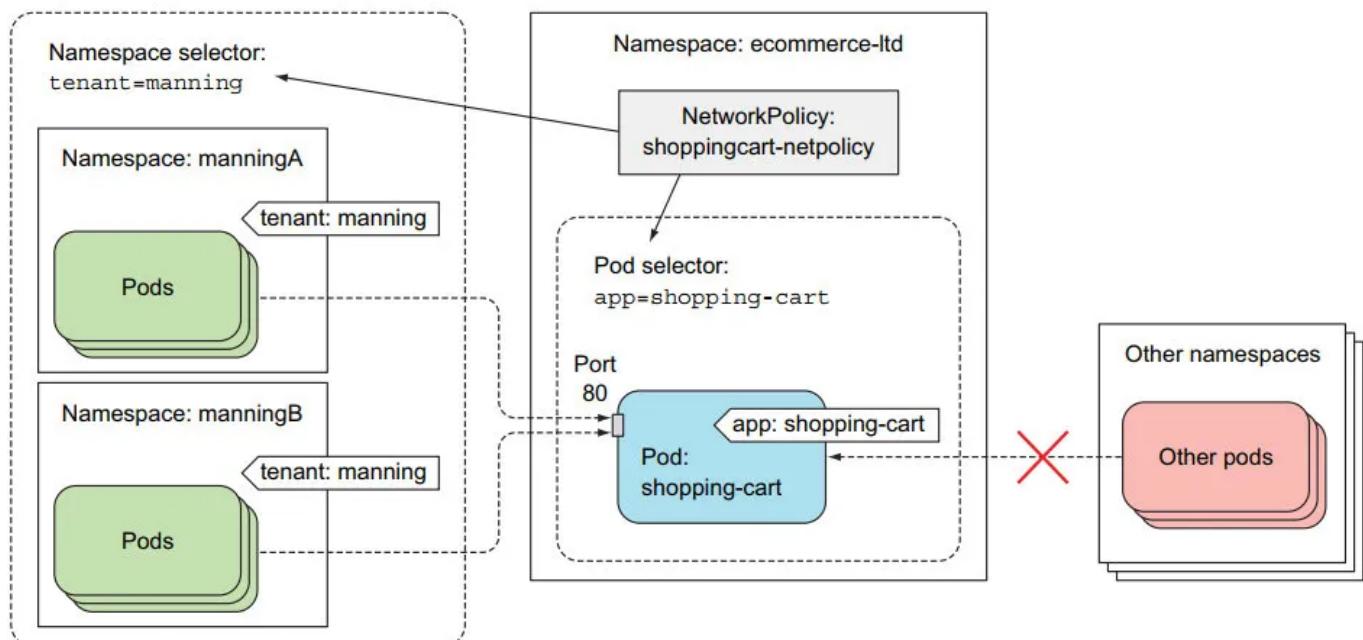
```

5: 在不同的命名空间之间隔离。

```

1 apiVersion: networking.k8s.io/v1
2 kind: NetworkPolicy
3 metadata:
4   name: shoppingcart-netpolicy
5 spec:
6   podSelector:
7     # 限定 pod 的范围
8     matchLabels:
9       app: shopping-cart
10  ingress:
11    - from:
12      # 只对以下命名空间 开放了 80 端口
13      - namespaceSelector:
14        matchLabels:
15          tenant: manning
16      ports:
17        - port: 80

```



6: CIDR 表示法：设置一个 IP 段

```

1 apiVersion: networking.k8s.io/v1
2 kind: NetworkPolicy
3 metadata:
4   name: ipblock-netpolicy
5 spec:
6   podSelector:
7     matchLabels:
8       app: shopping-cart
9   ingress:
10  - from:
11    # 限定 只有 192.168.1.1 ~ 192.168.1.255 的 pod 可以访问
12    - ipBlock:
13      cidr: 192.168.1.0/24

```

7: 出向规则 egress: 指定 pod 只能对外访问哪些 pod。

```

1 apiVersion: networking.k8s.io/v1
2 kind: NetworkPolicy
3 metadata:
4   name: egress-net-policy
5 spec:
6   podSelector:
7     matchLabels:
8       app: webserver
9   egress:
10  - to:
11    # webserver 只能访问 带有 app: database 标签的 pod
12    - podSelector:
13      matchLabels:
14        app: database
15    ports:
16      - port: 5432

```

## 资源管理

1: 配置 pod 资源的 预期使用量和最大使用量，可保证 pod 公平使用 集群内的资源。

### 容器申请资源

1: `requests` 中可申请 CPU 和内存使用量。

- 若不申请 CPU，极端情形，会被挂起

```

1 containers:
2   - image: busybox
3     command: ["dd", "if=/dev/zero", "of=/dev/null"]
4   name: main
5   resources:
6     requests:
7       # 200 毫核, 单核的 1/5, 1200m, 则是 1.2 个核【多核CPU】
8       cpu: 200m
9       memory: 10Mi

```

### top 中 CPU 的使用量率 占所有核的百分比

2: 调度器在调度 pod 时, 判断 pod 是否能调度到该节点的依据是根据 资源的申请量之和, 而非资源的实际使用量。

3: 调度器利用 pod requests 为其选择最佳节点 有两种策略:

- LeastRequestedPriority: pod 调度到 资源使用量少的节点上
- MostRequestedPriority: pod 调度到资源使用量高的节点上 【按节点付费时, 可选择】

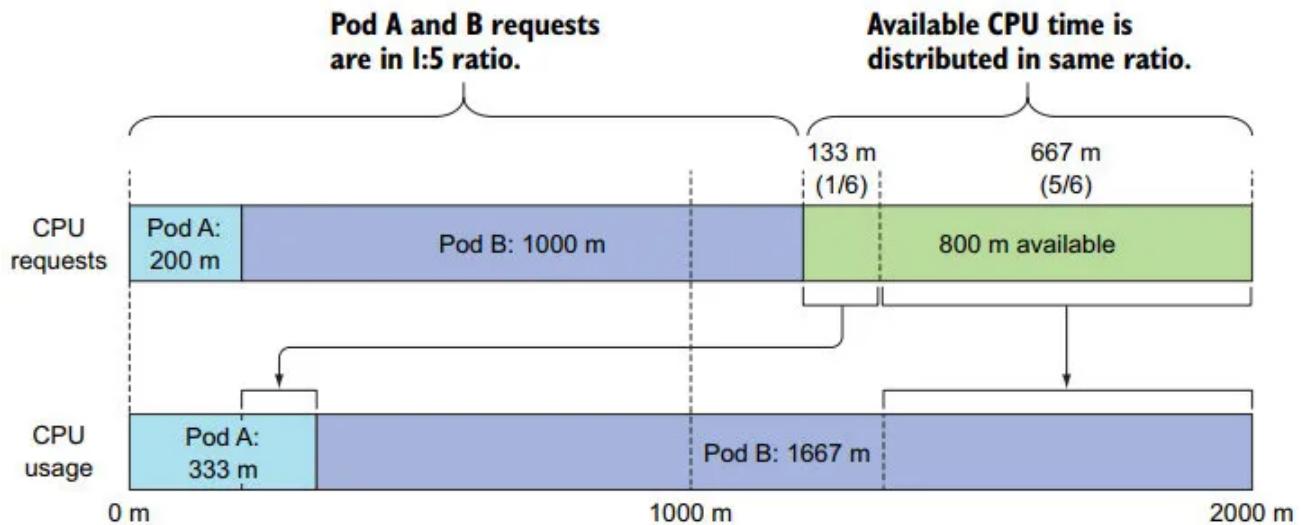
4: 当没有合适的 节点分配 给待调度的 pod 时, pod 状态会一直卡在 Pending 状态。

但此时并未放弃, 一旦有 pod 删除, 调度器将收到通知, 有可能重新 将 pod 部署在上面。

5: CPU requests 会影响时间片的分配。

假设一个节点上运行两个 pod, 一个 pod 请求 200 毫核 CPU, 另一个是 1000 毫核 CPU, 则时间片将按照 1: 5 分配。

- 若一个 pod 空闲, 另一个 pod 跑满, 则另一个 pod 将占用全部 CPU;
- 当空闲的 pod 重新运转, 另一个 pod 的 CPU 会立刻压缩到之前的比例 【动态伸缩, 提高 CPU 使用率】



6: 允许自定义资源, 例如 GPU。

## 限制容器资源

1: 限制容器可以消耗资源的最大量;

特别是内存，不可压缩资源，会影响后来 pod 的内存分配。

2: 未设置 requests, 则将指定与资源 limits 相同的值.

```
1 containers:
2   - image: busybox
3     command: ["dd", "if=/dev/zero", "of=/dev/null"]
4     name: main
5     resources:
6       limits:
7         cpu: 1
8         memory: 20Mi
```

3: 节点中所有 pod 的 limits 总量允许超过 节点总量 100%

4: 若内存超过物理上限，容器会被 OOM killed

若 Pod 的重启策略 `restartPolicy` 设置为 Always 或 OnFailure, 容器会立刻重启。

若 pod 连续重启 `CrashLoopBackOff`, 下次重启时间呈 指数级避退: 10、20、40、80、160 秒，最终收敛到 300s。一旦达到 300s 间隔，后续将以 5 分钟为间隔 无限重启。

5: 注意：在容器内看的内存(`free -g`)始终是节点的内存，而非容器的内存

无论有没有配置 CPU limits，容器内也会看到节点所有的 CPU。将 CPU 限额配置为 1，并不会神奇地只为容器暴露一个核。CPU limits 做的只是限制容器使用的 CPU 时间。

因此如果一个拥有 i 核 CPU 限额的容器运行在 64 核 CPU 上，只能获得 1/64 的全部 CPU 时间。而且即使限额设置为 1 核，容器进程也不会只运行在一个核上，不同时刻，代码还是会在多个核上执行。

一些程序通过查询系统 CPU 核数来决定启动工作线程的数量。同样在开发环境的笔记本电脑上运行良好，但是部署在拥有更多数量 CPU 的节点上，程序将快速启动大量线程，所有线程都会争夺（可能极其）有限的 CPU 时间。同时每个线程通常都需要额外的内存资源，导致应用的内存用量急剧增加。

## pod QoS 等级

1: 当资源 limits 超出节点上限时，可指定哪些 pod 从优先级更高，当资源不足时，首先杀掉的是低优先级的 pod.

可给 pod 分配三种 QoS 等级：

- BestEffort (优先级最低)
  - 没有设置任何 requests 和 limits 的 pod;
  - 无任何资源保证;
  - 最坏情况下，分不到 CPU 时间片，同时为其它 pod 释放内存时，第一批被杀死;
  - 由于无 limits，内存充足时，可使用任意多的内存;
- Burstable

- 容器的 requests 和 limits 不等
- 只定义了 requests 的 pod
- 部分容器 requests 和 limits 相等，部分不等

Burstable 获得它们所申请的等额资源，并可以使用额外的资源（不超过 limits）。

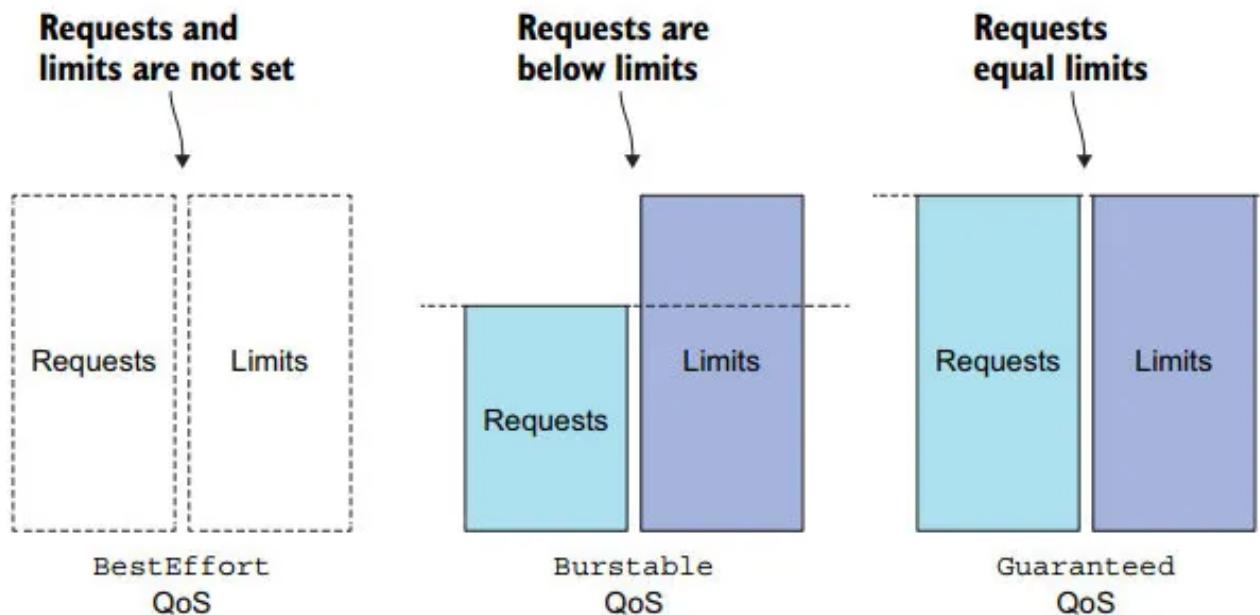
- Guaranteed (优先级最高)

- requests 和 limits 相等的 pod
- CPU 和内存都要设置 requests 和 limits
- 每个容器都要设置 资源量 【注意是每个都要设置】
- 每个容器的每种资源的 requests 和 limits 必须相等

这些 pod 的容器可以使用它所申请的等额资源，但是无法消耗更多的资源（因为它们的 limits 和 requests 相等）。

若没显示设置 requests，则默认和 limits 相同。所以设置了 limits 的 pod，QoS 就是 Guaranteed

三种等级的分类：



单容器 pod 的 QoS 等级

| CPU requests vs. limits | 内存的 requests vs. limits | 容器的 QoS 等级        |
|-------------------------|-------------------------|-------------------|
| 未设置                     | 未设置                     | <b>BestEffort</b> |
| 未设置                     | Requests < Limits       | Burstable         |
| 未设置                     | Requests = Limits       | Burstable         |
| Requests < Limits       | 未设置                     | Burstable         |
| Requests < Limits       | Requests < Limits       | Burstable         |
| Requests < Limits       | Requests = Limits       | Burstable         |
| Requests = Limits       | Requests = Limits       | <b>Guaranteed</b> |

对千多容器 pod, 如果所有的容器的 QoS 等级相同，那么这个等级就是 pod 的 QoS 等级。如果至少有一个容器的 QoS 等级与其他不同，无论这个容器是什么等级，这个 pod 的 QoS 等级都是 Burstable 等级。

| 容器 1 的 QoS 等级 | 容器 2 的 QoS 等级 | pod 的 QoS 容器 |
|---------------|---------------|--------------|
| BestEffort    | BestEffort    | BestEffort   |
| BestEffort    | Burstable     | Burstable    |
| BestEffort    | Guaranteed    | Burstable    |
| Burstable     | Burstable     | Burstable    |
| Burstable     | Guaranteed    | Burstable    |
| Guaranteed    | Guaranteed    | Guaranteed   |

2: 内存不足时, BestEffort 等级的 pod 首先被杀掉。

其次是 Burstable

最后是 Guaranteed, 只有系统进程需要内存时, 才会被杀掉。

对于 QoS 等级相同的 pod, 最先被杀掉的是 实际内存占内存申请量比例更高的 pod.

如下图, pod B 虽然 requests 比 pod C 少, 但使用率高达 90%, 所以先杀掉的是 pod B.

## LimitRange

1: LimitRange 给命名空间中的 pod 设置默认的 requests 和 limits

LimitRange 资源中的 limit 应用于同一个命名空间中每个独立的 pod、容器, 或者其他类型的对象。

当为显示指定 资源 requests 时, 设置默认值。

它并不会限制这个命名空间中所有 pod 可用资源的总量, 总量是通过 ResourceQuota 对象指定的,

2: LimitRange 资源被 LimitRanger 准入插件控制。

LimitRange 示例如下:

```

1  apiVersion: v1
2  kind: LimitRange
3  metadata:
4    name: example
5  spec:
6    limits:
7      # 整个 pod 的资源限制
8      - type: Pod
9        min:
10       cpu: 50m
11       memory: 5Mi
12     max:
13       cpu: 1
14       memory: 1Gi
15
16      # 单个容器的资源限制
17      - type: Container
18        defaultRequest:
19          cpu: 100m

```

```

20      memory: 10Mi
21
22      # limits 默认值
23      default:
24          cpu: 200m
25          memory: 100Mi
26      min:
27          cpu: 50m
28          memory: 5Mi
29      max:
30          cpu: 1
31          memory: 1Gi
32      # 每种资源 requests / limits 的最大比值
33      maxLimitRequestRatio:
34          cpu: 4
35          memory: 10
36
37      # PVC 限制
38      - type: PersistentVolumeClaim
39          min:
40              storage: 1Gi
41          max:
42              storage: 10Gi

```

3: LimitRange 中配置的 limits 只能应用于单独的 pod 或容器。用户仍然可以创建大量的 pod 吃掉集群所有可用资源。

## ResourceQuota

1: 限制命名空间中的 可用资源总量。

LimitRange 是针对单个的实体

ResourceQuota 是针对命名空间下的总量

2: 对象个数配额目前可以为以下对象配置：  
 • pod  
 • ReplicationController  
 • Secret  
 • ConfigMap  
 • Persistent Volume Claim  
 • Service（通用），以及两种特定类型的 Service，比如 LoadBalancer Service (services.loadbalancers) 和 NodePort Service (services.nodeports)

3: 可通过以下四种类型 控制 Quota 作用的范围

- BestEffort : Quota 是否应用于 BestEffort QoS 等级的 pod

注意：BestEffort 只能限制 pod 的个数。不能限制 CPU/内存的 requests 和 limits

下面的都能限制：

- NotBestEffort : Quota 是否应用于 Burstable 和 Guaranteed QoS 等级的 pod
- Termination: 设置了 activeDeadlineSeconds 的 pod (pod 被标记为 Failed 到真正停止前还能运行的事件)
- NotTerminating: 未设置 activeDeadlineSeconds 的 pod

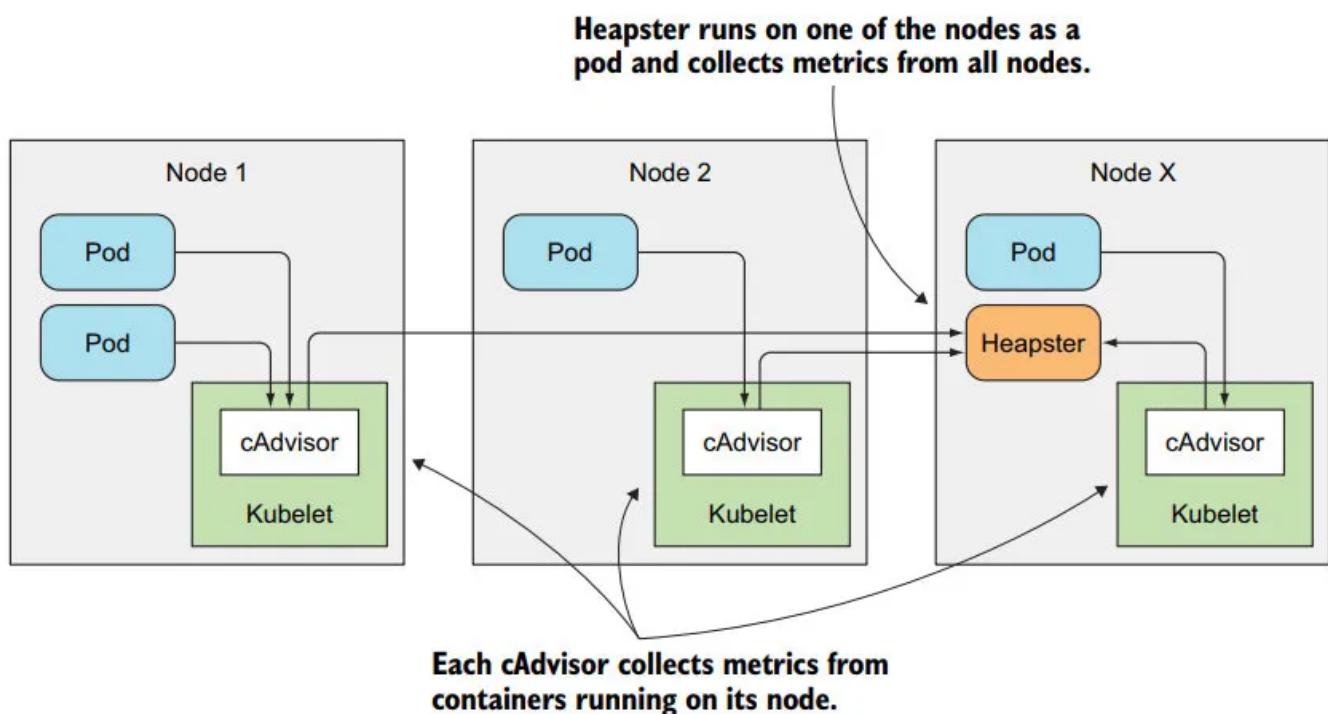
```

1 apiVersion: v1
2 kind: ResourceQuota
3 metadata:
4   name: besteffort-notterminating-pods
5 spec:
6   # 这个 Quota 值作用于 拥有 BestEffort QoS, 以及未设置 最大存活时间的 pod
7   scopes:
8     - BestEffort
9     - NotTerminating
10
11   # 这种的pod 只允许存在 4 个
12   hard:
13     pods: 4

```

## 监控 pod 资源使用量

- 1: 集群中，每个 Node 的 Kubelet 中 cAdvisor 代理负责收集本节点数据；  
 最终汇总到 Heapster 上（也运行在一个 pod）



- 2: 存储历史监控数据，可用 InfluxDB

可视化套件使用 Grafana

# 自动横向伸缩

1: 根据 CPU 使用率或其它度量指标，自动横向扩缩容。

- pod 的横向扩缩容；
- node 的横向扩缩容；

## pod 的横向伸缩

1: 横向 pod 自动伸缩是指由控制器管理的 **pod 副本数量** 的自动伸缩。它由 Horizontal 控制器执行，我们通过创建一个**HorizontalpodAutoscaler (HPA)**资源来启用和配置 Horizontal 控制器。

该控制器周期性检查 **pod 度量**，计算满足 HPA 资源所配置的目标数值所需的副本数量，进而调整目标资源（如 Deployment、ReplicaSet、ReplicationController、StatefulSet 等）的 replicas 字段。

AutoScale 能避免 抖动情况下的 自动扩缩容。

2: Pod 的度量数据 通过 kubelet 上的 cAdvisor agent 采集，并汇总到 Heapster.

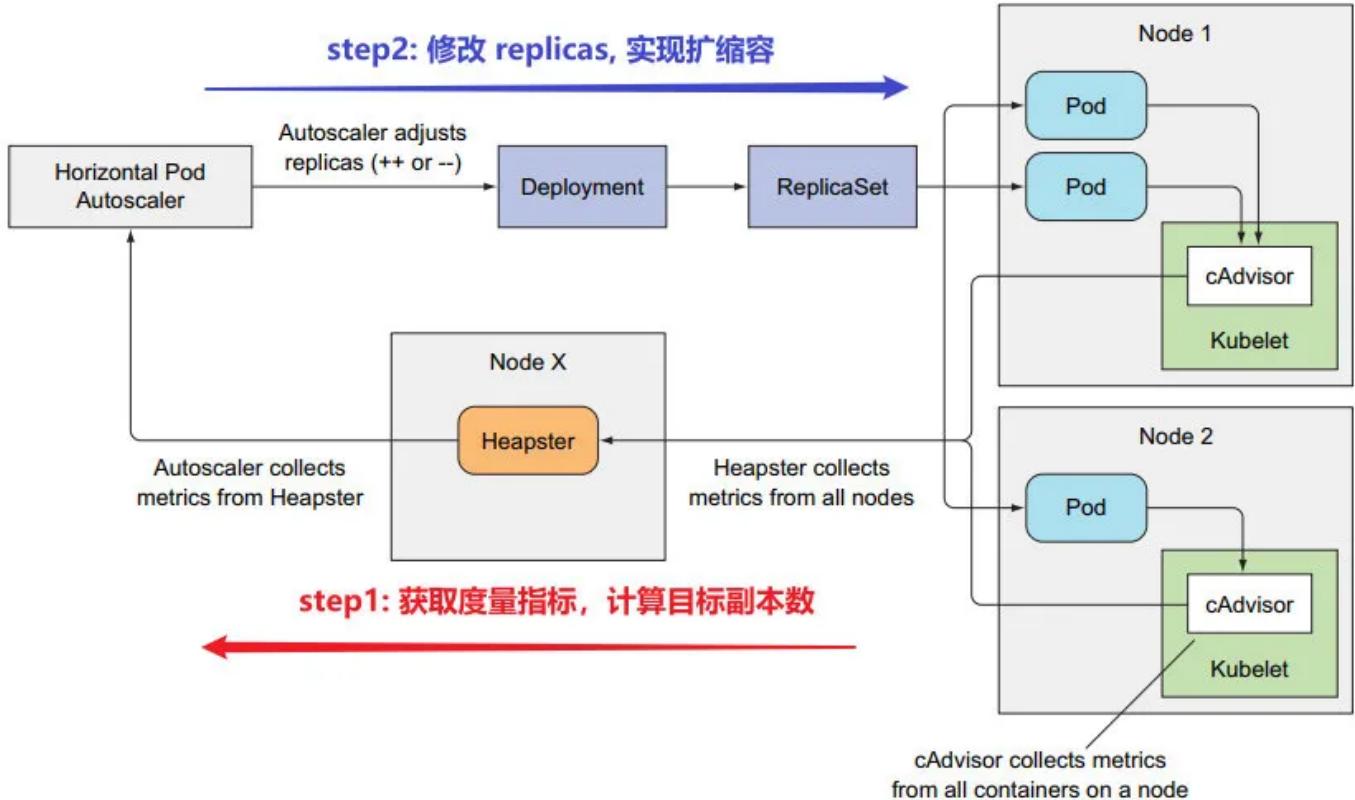
流程如下：

3: HPA 获得 度量值后（可以有多个度量），取每个度量下 计算的副本数 **最大值**作为 最终 pod 数量  
度量的目标值 是一个平均值。

例如，同时按照 CPU 使用率和 QPS 指标计算。

- |   |                                            |
|---|--------------------------------------------|
| 1 | 计算公式：                                      |
| 2 | 目标 Replicas = (所有Pod 度量值总和) / 目标度量值 # 向上取整 |

4: HPA 控制器 通过 Scale 子资源 修改 部署(Deployment、ReplicaSet、ReplicationController、StatefulSet) 的 replicas 字段，由部署相关的 控制器 实现 pod 的增减。



5: 拿 CPU 使用率举例，已经使用了 60% 的 CPU 是相对于 pod 请求量的 CPU 【并非节点 CPU 的 60% 或者资源上限 Quota 的 60%】

pod 请求量的基准可在 pod 模板的 requests 或者 LimitRange 间接设置

- 容器的 CPU 使用率是它实际的 CPU 使用除以它的 CPU 请求。
- 请求量是 ==最低标准，相当于有下限，没上限。==

例如：

```

1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    name: kubia
5  spec:
6    replicas: 3
7    template:
8      metadata:
9        name: kubia
10       labels:
11         app: kubia
12    spec:
13      containers:
14        - image: luksa/kubia:v1
15          name: nodejs
16          resources:
17            requests:
18              # 100 毫核, 1/10 的CPU
19              cpu: 100m

```

6: 创建一个 HPA 对象，并指向该 Deployment。

```
1 kubectl autoscale deployment <deployment-name> --cpu-percent=30 --min=1 --max=5: 对指定 deployment 自动伸缩 pod，使 CPU 使用量达到 30%，最少 1 个 pod，最多 5 个
```

使用 yaml 文件定义：

注意：HPA 的目标是 Deployment，在副本数量 replicas 更新后，Deployment 会给每个应用版本创建一个新的 ReplicaSet。

7: 基于内存的自动伸缩比基于 CPU 的更复杂，因为无法强制 Pod 释放内存，除非杀死并重启应用。

k8s 1.8 开始支持基于内存的自动伸缩

8: HPA 不支持 `minReplicas: 0`，即不允许缩容到 0 个副本，总得保持一个空载(idling)。

## 扩缩容速度

1: HPA 单次扩容操作，至多使副本数翻倍，如果副本数只有 1 或 2，则最多扩容到 4 个副本。

两次扩容之间也有时间限制，只有当 3 分钟内没有任何伸缩操作，才会继续触发扩容。

缩容频率更低，需要 5 分钟。

## Resource 度量类型

1: 基于 **Pod** 的度量类型，例如 pod 的 QPS，运行在 Pod 中消息队列的消息数量。

```
...  
spec:  
  metrics:  
    - type: Pods  
      resource:  
        metricName: qps  
        targetAverageValue: 100  
...  
  
          定义一个 pod 度量  
          度量的名称  
          所有被涵盖 pod 内的  
          目标平均值
```

注意，此时会从所有 Pod 中取度量值后，看平均值与目标度量对比。

2: 基于 Object 的度量类型，间接基于另一个集群对象，例如 Ingress 来伸缩 pod

```

...
spec:
  metrics:
    - type: Object
      resource:
        metricName: latencyMillis
        target:
          ...
          Autoscaler 应该使该度量尽量接近这个值
          targetValue: 20
      scaleTargetRef:
        apiVersion: extensions/v1beta1
        kind: Ingress
        name: frontend
        targetValue: 20
        scaleTargetRef:
          apiVersion: extensions/v1beta1
          kind: Deployment
          name: kubia

```

使用某个特定对象的度量  
度量的名称  
autoscaler 需要从中获取度量的特定对象  
autoscaler 将要管理的可伸缩资源

image-20210615171232588

注意，HPA 只会从这个特定的对象中获取单个的度量数据。【并非基于 Pod 的从所有对象中获取】

3: 一个好的度量类型，应该是增加副本数时，可以使度量平均值下降。例如 CPU、QPS

内存占用并非好的度量类型。

## 自动配置资源请求

1: 如果新创建的 pod 的容器没有明确设置 CPU 与内存请求，该特性即会代为设置。

这一特性由一个叫作**InitialResources**的准入控制(AdmissionControl)插件提供。当一个没有资源请求的 pod 被创建时，该插件会根据 pod 容器的历史资源使用数据（随容器镜像、tag 而变）来设置资源请求。

如果一个容器总是内存不足，下次创建一个包含该容器镜像的 pod 的时候，它的内存资源请求就会被自动调高了

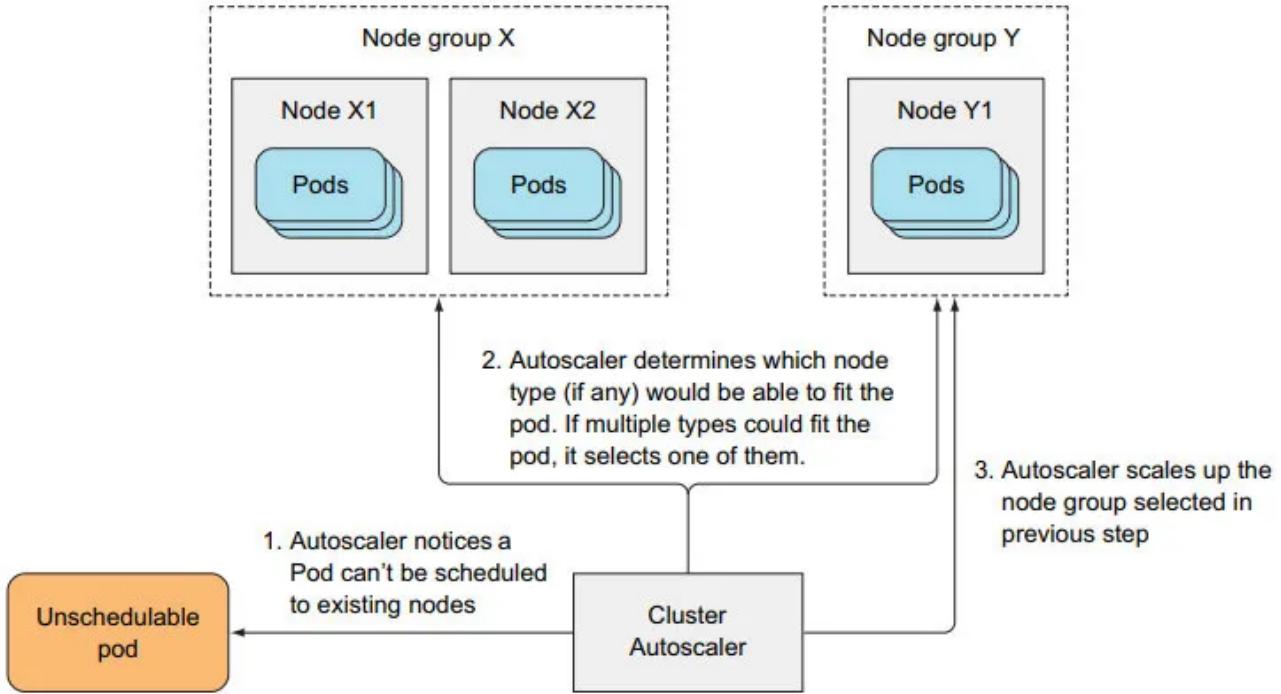
## 集群节点的横向伸缩

1: 集群节点的横向伸缩，解决所有节点都满了，放不下 Pod。

2: 当在云服务厂商上运行集群时，Cluster Autoscaler 负责请求/释放节点。

- 节点资源不足，申请节点：
  - 会先检查新节点有没有可能容纳这个 pod，若无法容纳，则不用启动该 node
  - 当有不同规格的节点类型时，会挑选一个最合适的节点（最差是随机选择一个）
- 节点长时间使用率底下，下线节点；

下图显示，当没有节点分配 pod 时，Cluster Autoscaler 会申请一个新的节点。



3: 归还节点：Cluster Autoscale 通过监控所有节点请求的 CPU 与 内存实现。

若某个 node 上，所有 pod 请求的 CPU、内存均达不到 50%，该节点认定为不再需要。

以下几种情形，节点不会被归还：【如果回收，会导致服务中断】

- 1: 有系统 pod 在运行 (DaemonSet 部署的服务)；
- 2: 非托管 pod
- 3: 本地存储的 pod

只有当节点上运行的所有 pod 能被重新调度到其它节点时，Cluster Autoscale 才会触发缩容。

缩容时，该节点首先会被标记为不可调度 【拒绝 pod 重新调度回来】，然后该节点上的 pod 会疏散到其它节点。

4: 节点下线时，可以通过 `podDisruptionBudget` 资源的指定下线等操作时需要保持的最少 pod 数量 【逐步迁移】，避免服务受影响。

```

1 | # 标签为 app=kubia 的 pod, 至少要保证3个在运行。
2 | kubectl create pd kubia-pdb --selector=app=kubia --min-available=3

```

## 高级调度

### 污点和容忍度

1: Pod 可通过 `nodeSelector` 和 节点亲缘性 选择在哪些节点上部署；

2: 从 node 侧，也能 限制 哪些 pod 可以部署在上面；

**污点 (Taints)**：节点添加污点，拒绝 pod 在该节点上部署；

污点在 master-node 上用的比较多，可以限制普通 pod 部署，只有系统 pod 才能部署在上面。

除非 pod 能容忍(Toleration)这个污点，否则不能部署在该 node.

某些硬件只能运行特殊的 pod，可采用这种形式

3: 通过 kubeadm 部署的集群上的主节点，查看其污点：

污点的格式

```
1 <key>=<value>:<effect>
2
3 # 若value为空
4 <key>:<effect>
```

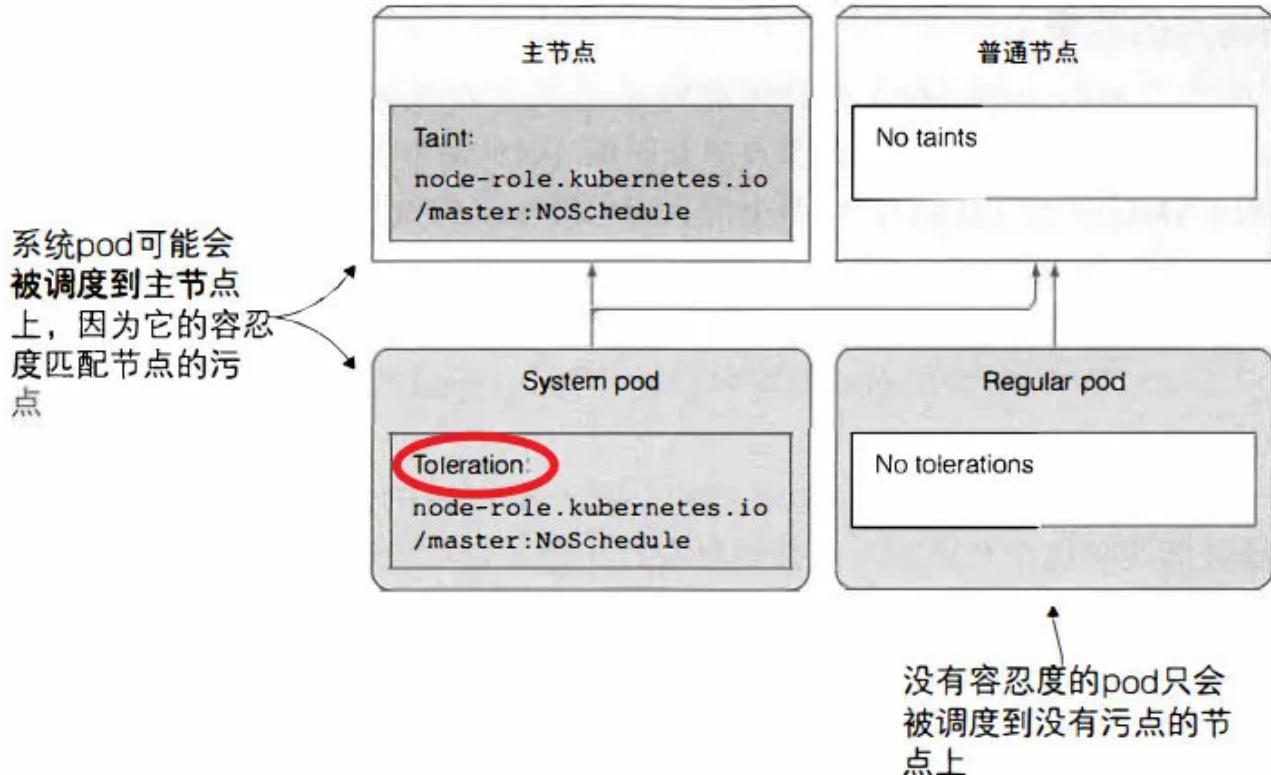
每个污点的效果(effect)包含三种：

- NoSchedule 表示如果 pod 没有容忍这些污点，pod 则不能被调度到包含这些污点的节点上。
- PreferNoSchedule 是 NoSchedule 的一个宽松的版本，表示尽量阻止 pod 被调度到这个节点上，但是如果其他节点可以调度，pod 依然会被调度到这个节点上。
- NoExecute 不同于 NoSchedule 以及 PreferNoSchedule，后两者只在调度期间起作用，而 NoExecute 会影响正在节点上运行着的 pod。如果在一个节点上添加了 NoExecute 污点，那些在该节点上运行着的 pod，如果没有容忍这个 NoExecute 污点，将会从这个节点去除。

4: 查看已有的污点：

如下，

- System pod 可以同时部署在主节点和普通节点；
- 但未添加容忍度(默认)的 pod 只能部署在普通节点；



5: 节点上添加自定义污点：

```
1 | kubectl taint node node1.k8s node-type=production:NoSchedule: 在节点上添加 key = node-type, value = production 的污点
```

6: 给 pod 添加容忍度示例:

```
1 | apiVersion: extensions/v1beta1
2 | kind: Deployment
3 | metadata:
4 |   name: prod
5 | spec:
6 |   replicas: 5
7 |   template:
8 |     metadata:
9 |       labels:
10 |         app: prod
11 |     spec:
12 |       containers:
13 |         - args:
14 |             - sleep
15 |             - "99999"
16 |           image: busybox
17 |           name: main
18 |       tolerations:
19 |         # 和 paint 配对使用
20 |         - key: node-type
21 |           # 使用 Equal 或 Exists
22 |           operator: Equal
23 |           value: production
24 |           effect: NoSchedule
```

7: 配置 node 不可用后, pod 最长等待时间:

可用于网络抖动的情形, 若 超时后, pod 将被调度到其它 node.

## 节点亲缘性

1: 亲缘性(node affinity): 允许你通知 Kubernetes 将 pod 只调度到某个几点子集上面。

通过给 节点 打标签, 然后 在 pod 中使用 `pod.spec.affinity.nodeaffinity` 强制选择(require) 或者 推荐选择(prefer) 节点。

一个典型的 gke 标准节点, 常常会打以下三种标签:

- `failure-domain.beta.kubernetes.io/region` 表示该节点所在的地理地域。
- `failure-domain.beta.kubernetes.io/zone` 表示该节点所在的可用性区域 (availability zone)。
- `kubernetes.io/hostname` 很显然是该节点的主机名。

已知的, 租户常常会选择 机型、区、地理位置 来确保服务的可用和高可用, 通过打标签, 相当于划分了机群。

## 2: 节点亲缘性比 nodeSelector 表达能力更强

- `requiredDuringScheduling...`: 调度的时候, 强制要求。若找不到合适的 node, 则 pod 无法部署
- `preferredDuringScheduling...`: 调度的时候, 推荐
  - 通常和 权重 weight 搭配使用, 用于控制优先级;
- `...IgnoredDuringExecution`: 忽略已经在 node 上运行的 pod, 否则会把不符合条件的 pod 踢出

```
1 apiVersion: extensions/v1beta1
2 kind: Deployment
3 metadata:
4   name: frontend
5 spec:
6   replicas: 5
7   template:
8     metadata:
9       labels:
10      app: frontend
11
12      spec:
13        affinity:
14          podAffinity:
15            # 强制要求, 忽略已经执行的pod
16            requiredDuringSchedulingIgnoredDuringExecution:
17              # 选择节点的范围
18              - topologyKey: rack
19                labelSelector:
20                  matchLabels:
21                    app: backend
22
23        containers:
24          - name: main
25            image: busybox
26            args:
27              - sleep
28              - "99999"
```

## 3: 节点亲缘性为 required 的, 只会在符合条件的 node 中部署:

4: 亲缘性为 preferred , 会优先选择都符合的, 其次是一个符合的 (按 weight 排列) , 最后是都不符合的 【一般会有一个 pod 部署在上面】

因为调度器还会使用其他优先级函数: `Selector SpreadPriority` 函数. 确保了属于同一个 ReplicaSet 或者 Service 的 pod, 将分散部署在不同节点上, 以避免单个节点失效导致这个服务也宕机。

```
1 spec:
2   affinity:
3     nodeAffinity:
4       preferredDuringSchedulingIgnoredDuringExecution:
5         # 配置第一个权重
```

```

6      - weight: 80
7          preference:
8              matchExpressions:
9                  - key: availability-zone
10                 operator: In
11                 values:
12                     - zone1
13             # 配置第2个的权重
14             - weight: 20
15                 preference:
16                     matchExpressions:
17                         - key: share-type
18                             operator: In
19                             values:
20                             - dedicated

```

## pod 间亲缘性

1: pod 间亲缘性 可以控制 多个 pod 部署在同一节点、同一机架、同一数据中心。

通常是 联系比较紧密的 pod 需要部署在一块，降低延时等。

```

1 kind: Deployment
2 # ....
3 spec:
4     replicas: 5
5     template:
6         #....
7         spec:
8             affinity:
9                 podAffinity:
10                # 强制限定
11                requiredDuringSchedulingIgnoredDuringExecution:
12                    # 节点选择的范围，含有 label-key=kubernetes.io/hostname 的节点集合
13                    - topologyKey: kubernetes.io/hostname
14
15                # 依赖的 pod 的标签
16                labelSelector:
17                    matchLabels:
18                        app: backend

```

2: 若被依赖的 pod 被删除了，重新调度时，还是会调用到原来的节点（调度器根据被依赖度，有一套反向打分机制）

## pod 间非亲缘性

1: 有些 pod 部署在一起会影响彼此的性能，需要分开部署

```
1 spec:
2   affinity:
3     # 非亲缘性
4     podAntiAffinity:
5       requiredDuringSchedulingIgnoredDuringExecution:
6         # 在指定的节点集中, 决定 pod 不能被调度的范围
7         - topologyKey: kubernetes.io/hostname
8           # pod 标签
9           labelSelector:
10             matchLabels:
11               app: frontend
```

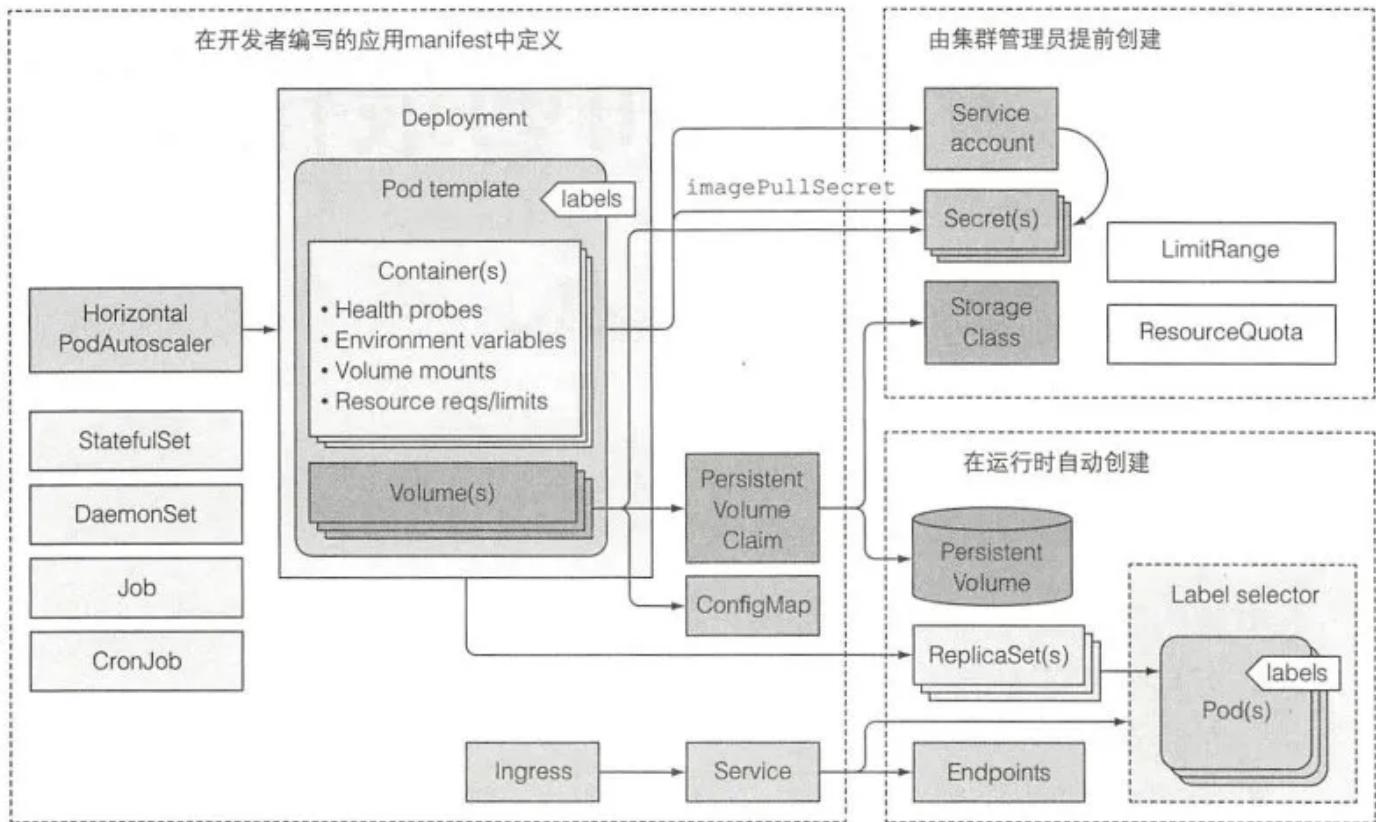
preferredDuringSchedulingIgnoredDuringExecution: 可指定软性要求

## 开发利用

1: 典型应用中使用的 k8s 组件如下:

- Manifest
  - 引用两种 Secret
  - 拉取私有镜像
  - pod 中的运行进程 访问其他 pod 或 k8s API 服务器
  - CronJobs
  - DaemonSet: 集群管理员 在每个 pod 上运行的系统服务
  - HorizontalpodAutoscaler: 水平 pod 扩容器
  - Deployment、StatefulSet 对象
  - Jobs
  - pod 模板
  - 每个容器包含 存活探针 和 就绪探针
- 集群外访问服务
  - LoadBalancer
  - NodePort
  - Ingress 资源
- 存储
  - pod 中以 configmap 卷挂载
  - ConfigMap: 初始化环境变量
  - emptyDir 卷
  - gitRepo 卷
  - PVC 卷
  - StorageClass 资源
- 资源管理

- LimitRange
- ResourceQuota
- 运行时创建的对象
  - 端点控制器(Endpoint controller) 创建的 Endpoint 对象;
  - Deployment Controller 创建的 ReplicaSet 对象
  - ReplicaSet 创建的 pod 对象



## pod 生命周期

1: 一个 pod 可以比作只运行单个应用的虚拟机。

pod 随时会重启，主机名和 ip 会变化（除非使用 StatefulSet）

容器重启，会有新的写入层，磁盘数据会丢失。应该使用 pod 级的存储卷，和 pod 同生命周期

2: 容器重启，并不会影响 ReplicaSet 重新调度 pod，RS 只关注 pod 的数量是否符合预期。

3: 以固定顺序启动 pod

- 一个 pod 可有任意数量的 initContainer 容器
- init 容器结束后，才会启动主容器
- 有依赖启动顺序的应用，最好是添加 Readiness 探针，特别是 Deployment，避免在滚动升级时，出现错误版本

4: 生命周期的钩子

- Post-start: 启动后钩子，容器启动后，和主进程并行执行；【钩子运行失败或返回非零错误码，会杀死主容器】
- Pre-Stop: 停止前钩子，容器停止前执行；执行后给容器发送 SIGTERM 【不管执行成功与否，都会使容器终

止】

容器崩溃不会执行

## 5: Pod 的关闭

Pod 的关闭是通过 API 服务器删除 pod 对象来触发的。

6: Kubelet 关闭 pod 时，会给每个容器一定的时间期限进行优雅的终止，这个时间叫做 **终止宽限期** (Termination Grace Period) 。

pod 的 `spec.terminationGracePeriod` 参数，默认 30

- \1. 执行停止前钩子（如果配置了的话），然后等待它执行完毕
- \2. 向容器的主进程发送 SIGTERM 信号
- \3. 等待容器优雅地关闭或者等待终止宽限期超时
- \4. 如果容器主进程没有优雅地关闭，使用 SIGKILL 信号强制终止进程

7: 若是有状态的 Pod 关闭，关闭前，需要将数据迁移到其它存活的 pod。

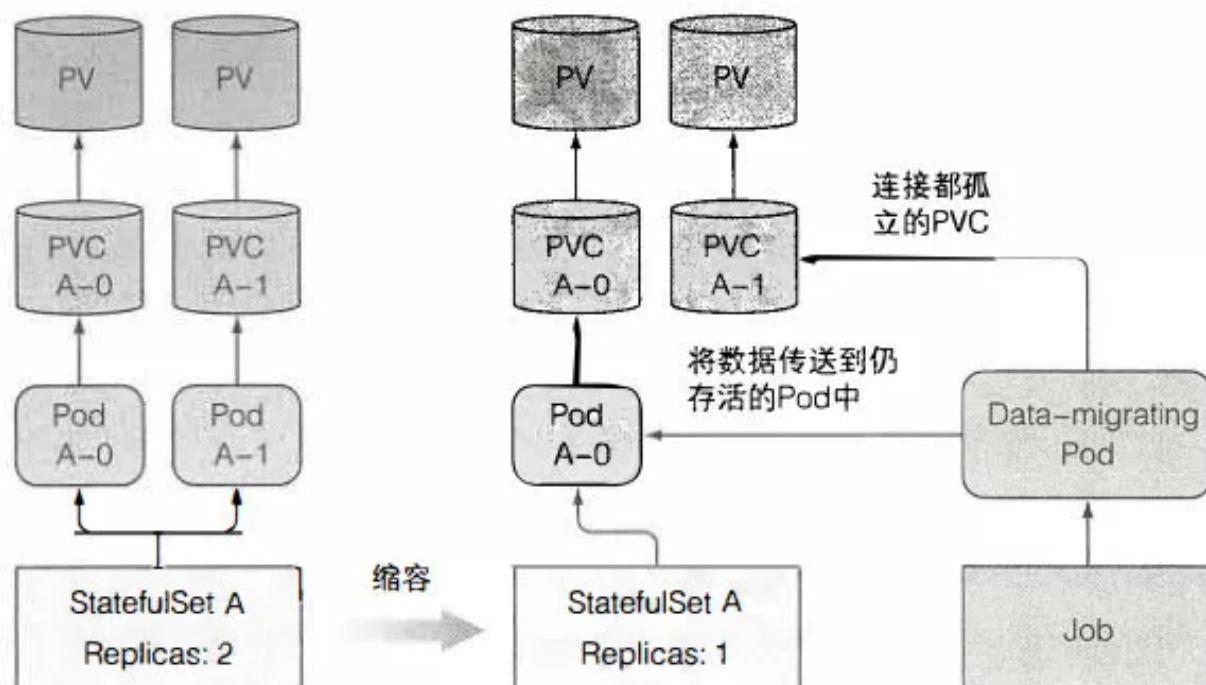
不建议在收到关闭信号的时候，触发数据迁移：

- 容器终止不一定代表整个 Pod 终止了（会有其它容器）
- 无法保证迁移流程在进程被杀死前执行完毕；（宽限期不够 或 关闭过程中 pod 发生故障）

若 pod 重启是不会触发迁移流程的。

推荐做法：

- 应用终止前，创建 Job 资源，运行一个单独的 pod 迁移数据；【前提是创建 Job 不会出故障】
- 创建一个 CronJob，周期性的检测是否有孤立数据
- 若使用 StatefulSet，缩容，会使 PVC 处于孤立状态，数据搁浅。【若扩容永远不发生的时候】，可在终止前拉起一个数据迁移的 pod



## 妥善处理 client 请求

1: pod 需要设置就绪探针, 确保服务可用, 才将 pod 加入可对外服务的集合中。

若不设置就绪探针, 则默认 pod 启动后, 服务立马可用。

2: k8s API 在收到关闭 Pod 时, 要做两件事:

- 修改 etcd 的状态,
- 删除通知给观察者 Kubelet 和 端点控制器(Endpoint Controller)

如下图所示, 这里的问题是 容器停止的事件 和 kube-proxy 修改 iptables 的前后是不确定的。

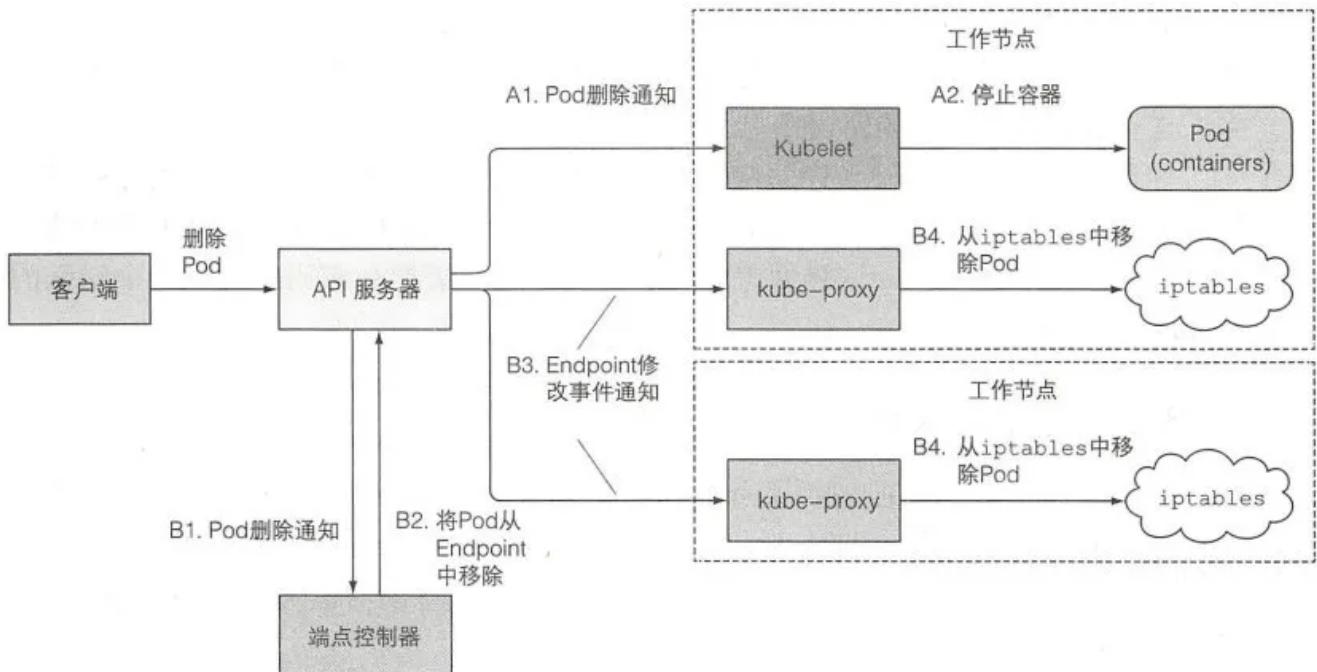


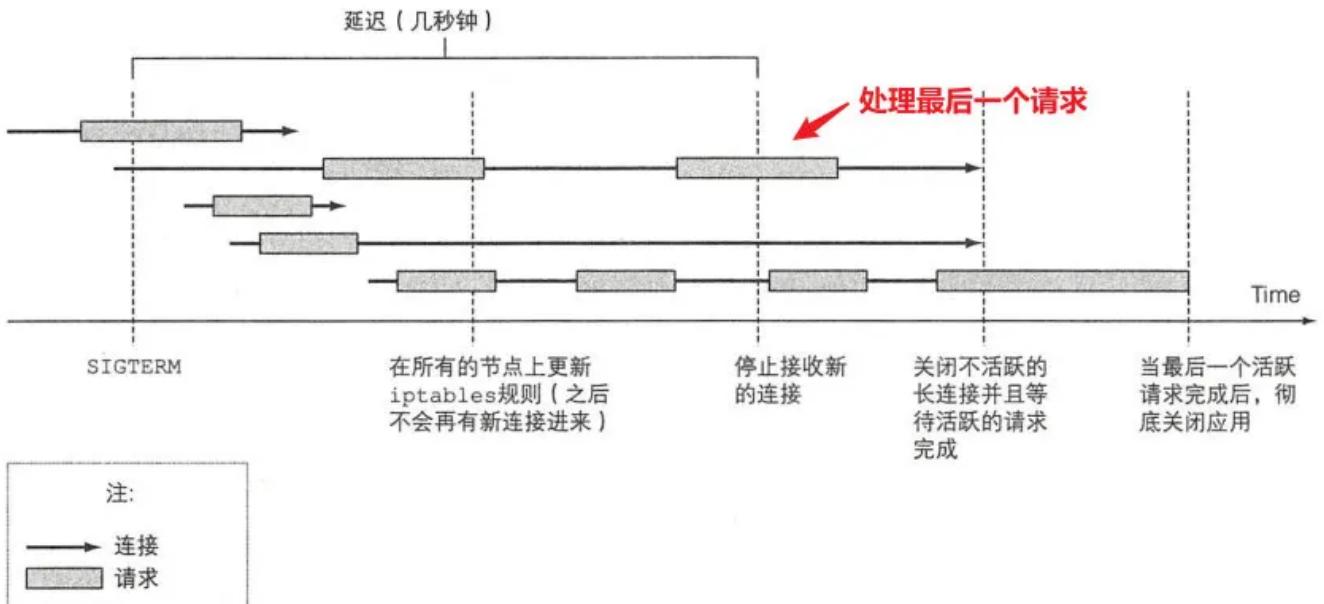
image-20210615141054178

删除 pod 的时序如下:

分两种情形:

- 1: 容器先停止, iptables 后被修改, 这时候 API 服务器会接着分配请求 【不合理】
- 2: iptables 先修改, 容器后停止 【合理】

对于第一种情形, 一般是 将容器 延后停止, 尽量满足第二种情形。停止时间需要取一个合理的值, 若时间太长, 会导致容器无法正常关闭, 太短可能无法处理 request。



## 应用在 k8s 中合理管理

1：打包镜像时：包含最小工具集即可，避免更新版本时间过长。

2：合理给镜像打标签

- 若一直是 latest 镜像，则 imagePullPolicy 需要设置为 Always，会有两个问题
  - 1：无法回退到旧镜像；
  - 2：每次创建新 pod，都要去检查镜像是否被修改了；

3：使用多维度标签

标签可以包含如下的内容 · 资源所属的应用（或者微服务）的名称 · 应用层级（前端、后端，等等） · 运行环境（开发、测试、预发布、生产，等等） · 版本号 · 发布类型（稳定版、金丝雀、蓝绿开发中的绿色或者蓝色，等等） · 租户（如果你在每个租户中运行不同的 pod 而不是使用命名空间） · 分片（带分片的系统）

分组管理资源

4：添加注解

- 包含作者；
- 应用必须的依赖；

5：更完善的进程终止信息

- 将终止消息写入 `/dev/termination-log` 【可通过 `terminationMessagePath` 修改】，当 `kubectl describe pod` 可看到终止日志
  - 若未设置，容器终止的最后几行日志会当做终止消息
- 将日志打印标准中断，`kubectl logs` 可见
- 或者写到 pod 中【挂载 pod 级别的卷】，通过 `kubectl cp` 可拷贝出来；

6：集中式日志记录，由 ElasticSearch、Logstash 和 Kibana 组成的 ELK 栈。【可通过 Helm 部署】

- FluentD 通过 DaemonSet 部署 Pod 收集日志
- Kibana 可可视化 ElasticSearch 的 web 工具，也是单独作为 Pod 运行

# k8s 应用扩展

## 自定义 API 对象

1: 开发者需要向 k8s API 提交 CustomResourceDefinitions (CRD) 对象，即可提交 Json 或 YAML 清单的方式创建新的资源类型。

2: 例如创建一个 静态网站，自动拉取 Git， 创建 pod，并通过 Service 对外公开。

3: 先创建 CRD 对象，让 k8s API 服务器识别该类型。

```
1 apiVersion: apiextensions.k8s.io/v1beta1
2 kind: CustomResourceDefinition
3 metadata:
4   # 资源名称
5   name: websites.extensions.example.com
6 spec:
7   # Website 这种资源属于哪种命名空间
8   scope: Namespaced
9
10  # 定义API 集群和所属版本
11  group: extensions.example.com
12  version: v1
13  names:
14    # 缩短资源的名称
15    kind: Website
16    singular: website
17    # 最后 url 的链接
18    # http://localhost:8001/apis/extensions.example.com/v1/websites?watch=true
19    plural: websites
```

4: 提交自定义资源请求：

```
1 kind: Website
2 metadata:
3   name: kubia
4 spec:
5   gitRepo: https://github.com/luksa/kubia-website-example.git
```

5: create 上述两种资源后，可用 `kubectl get` 查看资源实例

```
1 | kubectl get websites
```

6: 自定义控制器，通过 HTTP 监听 API 服务器 Add/Delete 接口，创建 Deployment 资源和 Service 资源。

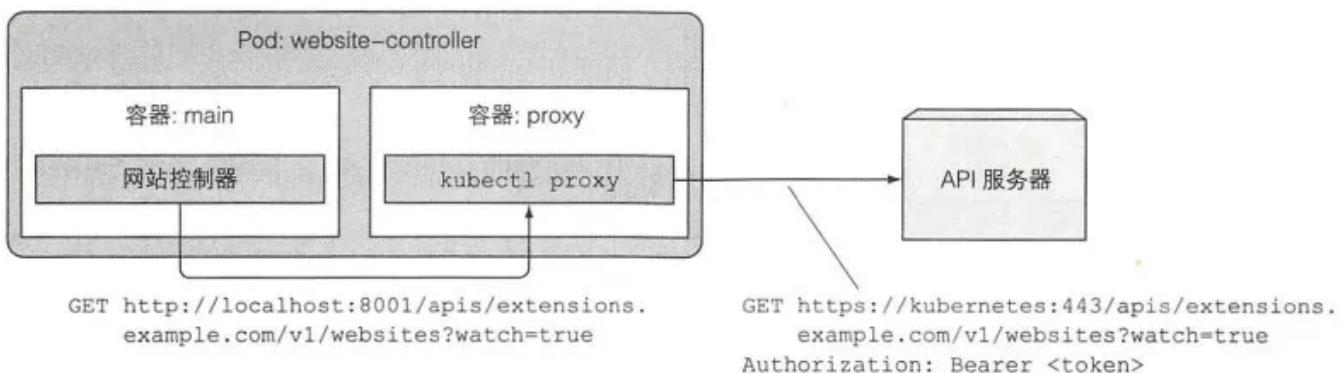
7: 控制器部署成一个 pod

```

1 apiVersion: apps/v1beta1
2 kind: Deployment
3 metadata:
4   name: website-controller
5 spec:
6   replicas: 1
7   template:
8     metadata:
9       name: website-controller
10      labels:
11        app: website-controller
12    spec:
13      serviceAccountName: website-controller
14      containers:
15        - name: main
16          image: luksa/website-controller
17        - name: proxy
18          image: luksa/kubectl-proxy:1.6.2

```

自定义控制器先和 proxy 通信，然后由 proxy 连接 k8s API.



运行这个控制器 pod, 和 API 服务器进行通信，需要创建特殊的 ServiceAccount,

若启用了角色访问控制(RBAC), 则需要 clusterrolebinding 到 `clusterrole=cluster-admin`

控制器运行的核心逻辑如下：

```

1 func main() {
2     log.Println("website-controller started.")
3     for {
4         resp, err :=
5             http.Get("http://localhost:8001/apis/extensions.example.com/v1/websites?
6             watch=true")
7         if err != nil {
8             panic(err)
9         }
10        defer resp.Body.Close()

```

```

10     decoder := json.NewDecoder(resp.Body)
11     for {
12         var event v1.WebsiteWatchEvent
13         if err := decoder.Decode(&event); err == io.EOF {
14             break
15         } else if err != nil {
16             log.Fatal(err)
17         }
18
19         log.Printf("Received watch event: %s: %s: %s\n", event.Type,
20         event.Object.Metadata.Name, event.Object.Spec.GitRepo)
21
22         if event.Type == "ADDED" {
23             createWebsite(event.Object)
24         } else if event.Type == "DELETED" {
25             deleteWebsite(event.Object)
26         }
27     }
28
29 }
30
31 func createWebsite(website v1.Website) {
32     createResource(website, "api/v1", "services", "service-template.json")
33     createResource(website, "apis/extensions/v1beta1", "deployments", "deployment-
34     template.json")
35 }
36
37 func deleteWebsite(website v1.Website) {
38     deleteResource(website, "api/v1", "services", getName(website));
39     deleteResource(website, "apis/extensions/v1beta1", "deployments",
40     getName(website));
41 }
42
43 func createResource(webserver v1.Website, apiGroup string, kind string, filename
44     string) {
45     log.Printf("Creating %s with name %s in namespace %s", kind,
46     getName(webserver), webserver.Metadata.Namespace)
47     templateBytes, err := ioutil.ReadFile(filename)
48     if err != nil {
49         log.Fatal(err)
50     }
51     template := strings.Replace(string(templateBytes), "[NAME]",
52     getName(webserver), -1)
53     template = strings.Replace(template, "[GIT-REPO]", webserver.Spec.GitRepo, -1)
54 }
```

```

50     resp, err := http.Post(fmt.Sprintf("http://localhost:8001/%s/namespaces/%s/%s/", apiGroup,
51     webserver.Metadata.Namespace, kind), "application/json",
52     strings.NewReader(template))
53     if err != nil {
54         log.Fatal(err)
55     }
56     log.Println("response Status:", resp.Status)
57 }
58
59 func deleteResource(webserver v1.Website, apiGroup string, kind string, name
60 string) {
61     log.Printf("Deleting %s with name %s in namespace %s", kind, name,
62     webserver.Metadata.Namespace)
63     req, err := http.NewRequest(http.MethodDelete,
64     fmt.Sprintf("http://localhost:8001/%s/namespaces/%s/%s/%s", apiGroup,
65     webserver.Metadata.Namespace, kind, name), nil)
66     if err != nil {
67         log.Fatal(err)
68     }
69     resp, err := http.DefaultClient.Do(req)
70     if err != nil {
71         log.Fatal(err)
72     }
73     log.Println("response Status:", resp.Status)
74 }
75 }
```

8: 通过 模板 创建所需要的 部署:

请求创建的 `deployment-template.json` pod 模板如下:

- 创建 nginx 容器，提供服务
- 创建 git-sync 拉取仓库，并通过 emptyDir 进行容器间共享

```

1 {
2     "apiVersion": "extensions/v1beta1",
3     "kind": "Deployment",
4     "metadata": {
5         "name": "[NAME]",
6         "labels": {
7             "webserver": "[NAME]"
8         }
9     },
```

```
10 "spec": {
11     "replicas": 1,
12     "template": {
13         "metadata": {
14             "name": "[NAME]",
15             "labels": {
16                 "webserver": "[NAME]"
17             }
18         },
19         "spec": {
20             "containers": [
21                 {
22                     "image": "nginx:alpine",
23                     "name": "main",
24                     "volumeMounts": [
25                         {
26                             "name": "html",
27                             "mountPath": "/usr/share/nginx/html",
28                             "readOnly": true
29                         }
30                     ],
31                     "ports": [
32                         {
33                             "containerPort": 80,
34                             "protocol": "TCP"
35                         }
36                     ]
37                 },
38                 {
39                     "image": "openweb/git-sync",
40                     "name": "git-sync",
41                     "env": [
42                         {
43                             "name": "GIT_SYNC_REPO",
44                             "value": "[GIT-REPO]"
45                         },
46                         {
47                             "name": "GIT_SYNC_DEST",
48                             "value": "/gitrepo"
49                         },
50                         {
51                             "name": "GIT_SYNC_BRANCH",
52                             "value": "master"
53                         },
54                         {
55                             "name": "GIT_SYNC_REV",
56                             "value": "FETCH_HEAD"
57                         },
58                         {
59                         }
60                     ]
61                 }
62             ]
63         }
64     }
65 }
```

```

59                 "name": "GIT_SYNC_WAIT",
60                 "value": "10"
61             }
62         ],
63         "volumeMounts": [
64             {
65                 "name": "html",
66                 "mountPath": "/gitrepo"
67             }
68         ]
69     }
70 ],
71 "volumes": [
72     {
73         "name": "html",
74         "emptyDir": {
75             "medium": ""
76         }
77     }
78 ]
79 }
80 }
81 }
82 }
```

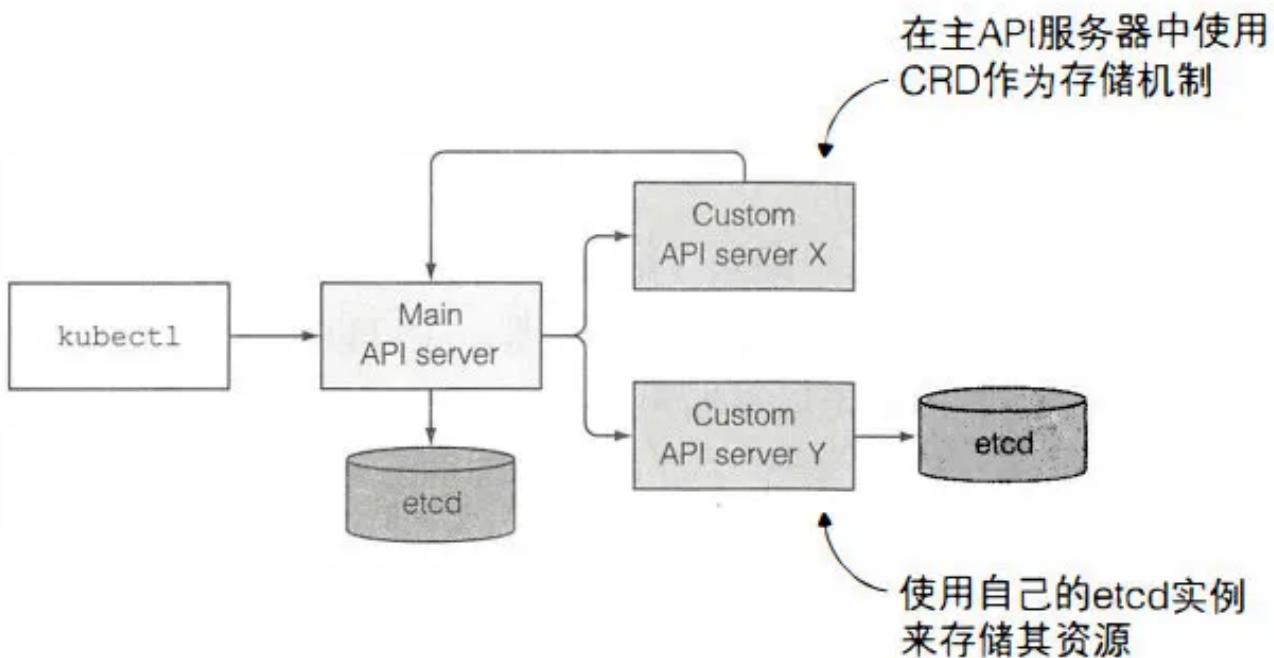
service 模板 `service-template.json` 如下

```

1 {
2     "apiVersion": "v1",
3     "kind": "Service",
4     "metadata": {
5         "labels": {
6             "webserver": "[NAME]"
7         },
8         "name": "[NAME]"
9     },
10    "spec": {
11        "type": "NodePort",
12        "ports": [
13            {
14                "port": 80,
15                "protocol": "TCP",
16                "targetPort": 80
17            }
18        ],
19        "selector": {
20            "webserver": "[NAME]"
21        }
22    }
```

## 自定义 API 服务器

1: k8s 1.7 后, 可自定义 API 服务器 集成到 主 API 服务器上



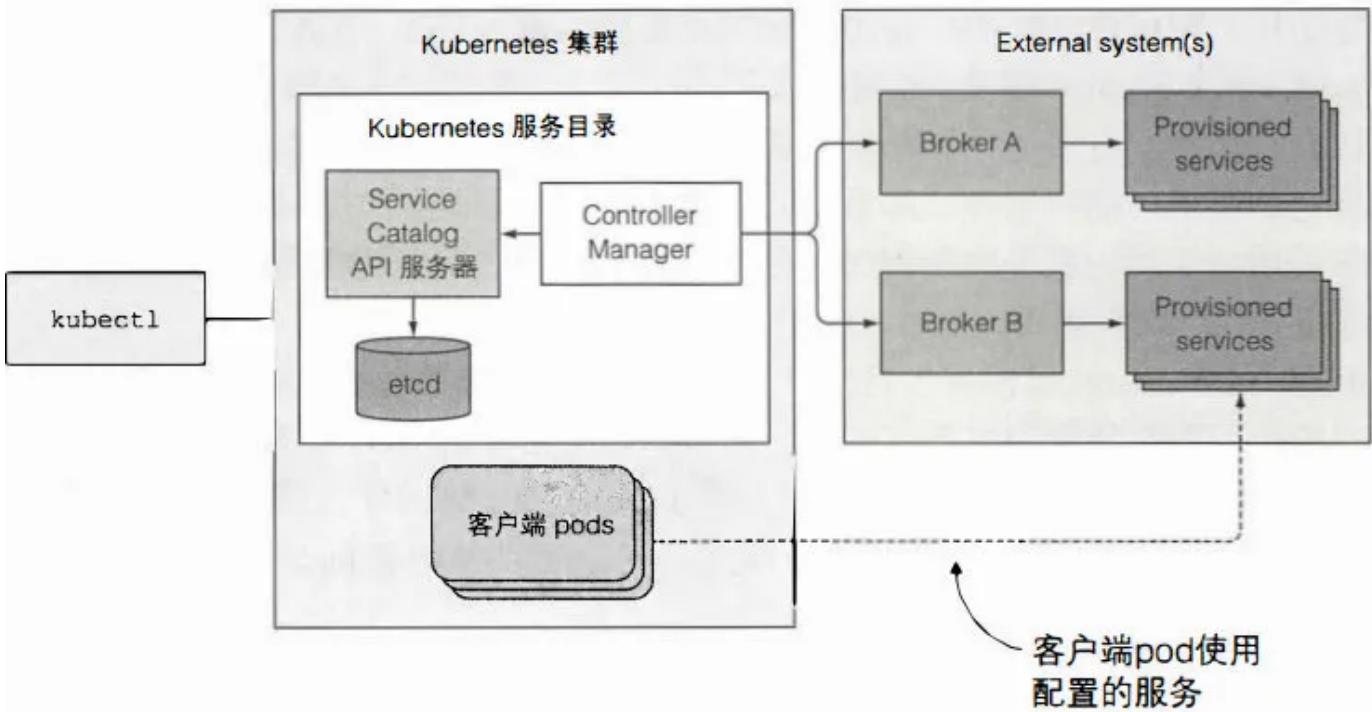
## 使用 k8s 服务目录扩展

1: 服务目录可以快速创建服务实例, 而无需处理一个个的 Pod、service、configMap 和其它资源。

2: 服务目录使用四种通用 API 资源:

- 一个 ClusterServiceBroker, 描述一个可以提供服务的 (外部) 系统
  - 集群管理员 为每个 服务代理创建 Broker 资源
- 一个 ClusterServiceClass, 描述一个可供应的服务类型
  - k8s 从服务代理获取到的 服务列表
- 一个 ServiceInstance, 已配置服务的一个实例
  - 用户调配服务时, 创建实例
- 一个 ServiceBinding, 表示 一 组客户端(pod) 和 ServiceInstance 之间的绑定。
  - 绑定到具体的 pod (`instanceRef` 引入具体的实例), 并注入一个 Secret

3: 服务目录是作为一种外部系统提供服务, 向 k8s 集群中注册代理, 暴露服务。



## PaaS

1: platform-as-a-Service, 平台即服务

2: 红帽(Red Hat) 的 OpenShift

提供多用户环境

- OpenShift PaaS 服务
  - Minishift, 与 Minikube 等效
  - <https://manage.openshift.com>
- 3:
  - Deis Workflow 【微软】
    - <https://deis.com/workflow>
  - Helm : 部署现有应用的标准方式, 包管理器, 类似于 yum, apt, homebrew
    - 寻找现有的图表: <https://github.com/kubernetes/charts>
    - helm install --name my-database stable/mysql: 自动部署所需的 Deployment、Service、Secret 和 PersistentVolumeClaim
    - OpenVPN: 最有用的图表, 通过 VPN 和访问服务来输入 pod 网络, 类似本地计算机是集群中的一个容器, 对开发应用程序并在本地运行时很有用
    - helm CLI 客户端
    - Tiller

## 致谢

感谢读者阅读, 欢迎指正错误, 谢谢。

## 参考及扩展

- Luksa M. Kubernetes in action[M]. Shelter Island: Manning Publications, 2018.
- <https://skyao.io/>
- <https://jimmysong.io/>
- 搜索镜像: <https://hub.docker.com/>
- 社区兴趣小组: <https://github.com/kubernetes/community/blob/master/sig-list.md>
- Swagger 创建 访问 API 服务器的客户端 : <https://swagger.io/>
- 控制器相关 源代码: <https://github.com/kubernetes/kubernetes/tree/master/pkg/controller>
- k8s 中领导者选举的例子: <https://github.com/kubernetes-retired/contrib/blob/master/election/>
- k8s 集群管理员指南: <http://kubernetes.io/docs/admin>
- minikube 文档: <https://minikube.sigs.k8s.io/docs/commands/ip/>
- minikube mount 将本地文件系统挂载到 Minikube Vm 中, 然后通过一个 hostPath 卷 挂载到容器
  - <https://github.com/kuberentes/minikube/tree/master/docs>
- kube-applier 工具: 可以定时从 版本控制中检出资源, 提交更改
- Ksonnet + jsonnet: 自定义高级片段, 快速转换成完整的 Deployment 配置文件
  - <https://github.com/ksonnet/ksonnet-lib>
- Fabric8 持续集成方案: <http://fabric8.io>
  - Google Cloud 在线实验室: <https://github.com/GoogleCloudPlatform/continuous-deployment-on-kubernetes>
- k8s 最新的 代码仓库: <http://github.com/kubernetes>
- OpenShift PaaS 服务
  - Minishift, 与 Minikube 等效
  - <https://manage.openshift.com>
- Deis Workflow 【微软】
  - <https://deis.com/workflow>
- Helm : 部署现有应用的标准方式, 包管理器, 类似于 yum,apt,homebrew
  - 寻找现有的图表: <https://github.com/kubernetes/charts>
  - helm install --name my-database stable/mysql: 自动部署所需的 Deployment、Service、Secret 和 PersistentVolumeClaim
  - OpenVPN: 最有用的图表, 通过 VPN 和访问服务来输入 pod 网络, 类似本地计算机是集群中的一个容器, 对开发应用程序并在本地运行时很有用
  - helm CLI 客户端
  - Tiller
- kubernetes 中文文档: <https://kubernetes.io/zh/docs/home/>