Kubernetes 架构详解

这一章详细介绍一下 Kubernetes 架构原理、应用场景以及 Kubernetes 集群基本概念和术语。本章无实操内容，大家了解一下 Kubernetes 原理即可。后面这些原理我们还会简单再提。

Kubernetes 学习常用参考地址：
kubernetes github 地址：
https://github.com/kubernetes/kubernetes

kubernetes 官方站点：
官网地址：https://kubernetes.io/
中文官方站点：https://kubernetes.io/zh/
官方手册：https://kubernetes.io/docs/

张岩峰老师微信，加我微信，邀请你加入 VIP 交流答疑群：
微信号：ZhangYanFeng0429
二维码：

1、Kubernetes 简介

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，Kubernetes 的目标是让部署容器化的应用简单并且高效（powerful）,Kubernetes 提供了应用部

署，规划，更新，维护的一种机制。

传统的应用部署方式是通过插件或脚本来安装应用。这样做的缺点是应用的运行、配置、管理、所有生存周期将与当前操作系统绑定，这样做并不利于应用的升级更新/回滚等操作，当然也可以通过创建虚拟机的方式来实现某些功能，但是虚拟机非常重，并不利于可移植性。

新的方式是通过部署容器方式实现，每个容器之间互相隔离，每个容器有自己的文件系统，容器之间进程不会相互影响，能区分计算资源。相对于虚拟机，容器能快速部署，由于容器与底层设施、机器文件系统解耦的，所以它能在不同云、不同版本操作系统间进行迁移。

容器占用资源少、部署快，每个应用可以被打包成一个容器镜像，每个应用与容器间成一对一关系也使容器有更大优势，使用容器可以在 build 或 release 的阶段，为应用创建容器镜像，因为每个应用不需要与其余的应用堆栈组合，也不依赖于生产环境基础结构，这使得从研发到测试、生产能提供一致环境。类似地，容器比虚拟机轻量、更“透明”，这更便于监控和管理。

Kubernetes 是 Google 开源的一个容器编排引擎，Kubernetes 积累了作为 Google 生产环境运行工作负载 15 年的经验，它支持自动化部署、大规模可伸缩、应用容器化管理。在生产环境中部署一个应用程序时，通常要部署该应用的多个实例以便对应用请求进行负载均衡。

在 Kubernetes 中，我们可以创建多个容器，每个容器里面运行一个应用实例，然后通过内置的负载均衡策略，实现对这一组应用实例的管理、发现、访问，而这些细节都不需要运维人员去进行复杂的手工配置和处理。

2、Kubernetes 功能优点

（1）自动装箱

基于容器对应用运行环境的资源配置要求自动部署应用容器。

（2）自我修复(自愈能力)

当容器失败时，会对容器进行重启。

当所部署的 Node 节点有问题时，会对容器进行重新部署和重新调度。

当容器未通过监控检查时，会关闭此容器直到容器正常运行时，才会对外提供服务。

（3）水平扩展

通过简单的命令、用户 UI 界面或基于 CPU 等资源使用情况，对应用容器进行规模扩大或规模剪裁。

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。

（4）服务发现

用户不需使用额外的服务发现机制，就能够基于 Kubernetes 自身能力实现服务发现和负载均衡。

（5）滚动更新

可以根据应用的变化，对应用容器运行的应用，进行一次性或批量式更新。

（6）版本回退

可以根据应用部署情况，对应用容器运行的应用，进行历史版本即时回退。

（7）密钥和配置管理

在不需要重新构建镜像的情况下，可以部署和更新密钥和应用配置，类似热部署。

（8）存储编排

自动实现存储系统挂载及应用，特别对有状态应用实现数据持久化非常重要存储系统可以来自于本地目录、网络存储（NFS、Ceph 等）、公共云存储服务。

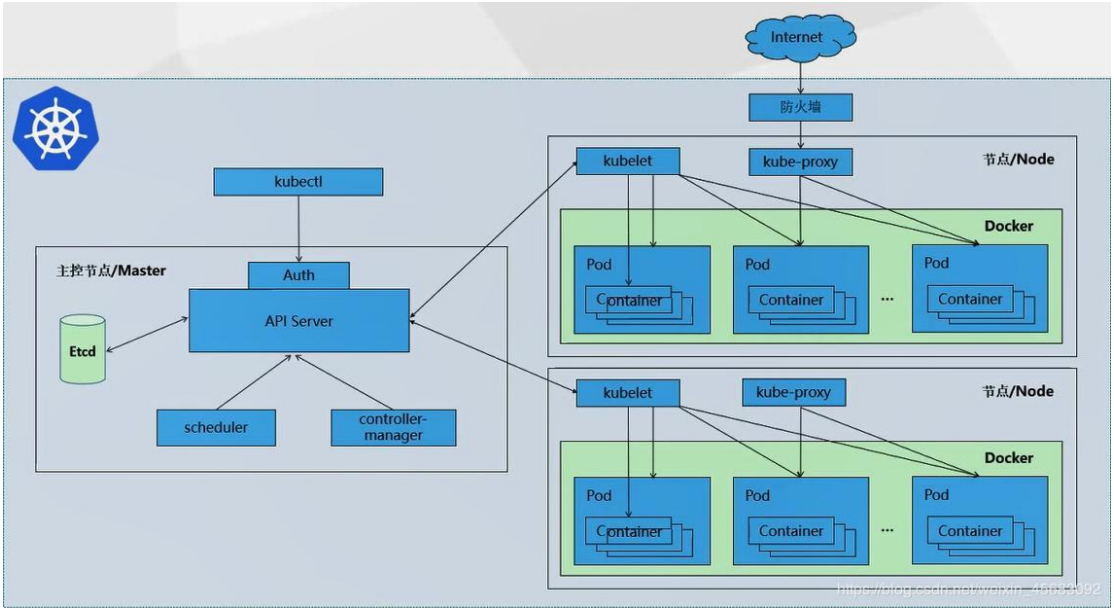
（9）批处理

提供一次性任务，定时任务；满足批量数据处理和分析的场景。

3、Kubernetes 集群架构

k8s 的物理架构是 master/node 模式：

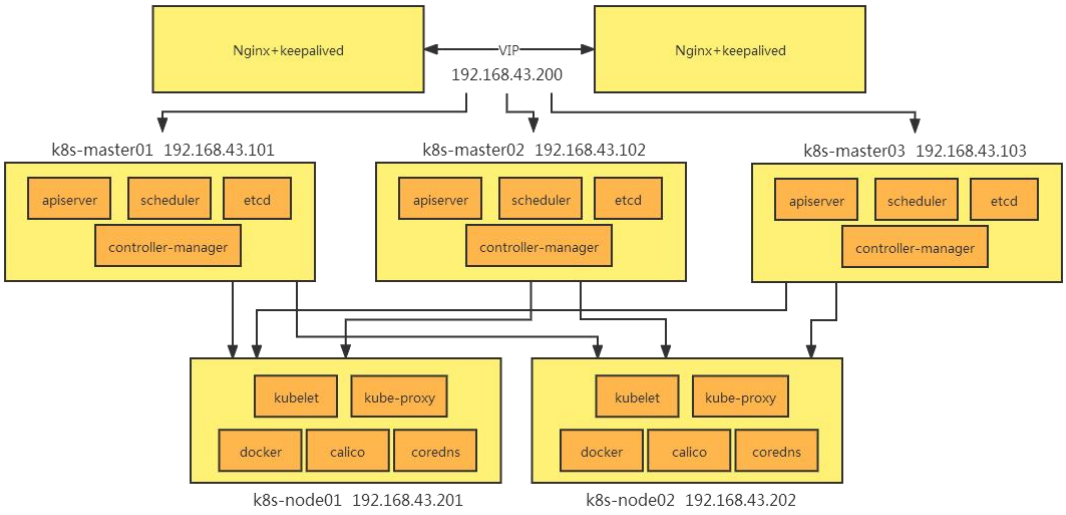
K8S 集群至少需要一个主节点(Master)和多个工作节点(Worker)，单 master 节点架构图如下：



Master 节点是集群的控制节点，负责整个集群的管理和控制，主节点主要

用于暴露 API，调度部署和节点的管理。工作节点主要是运行容器的。

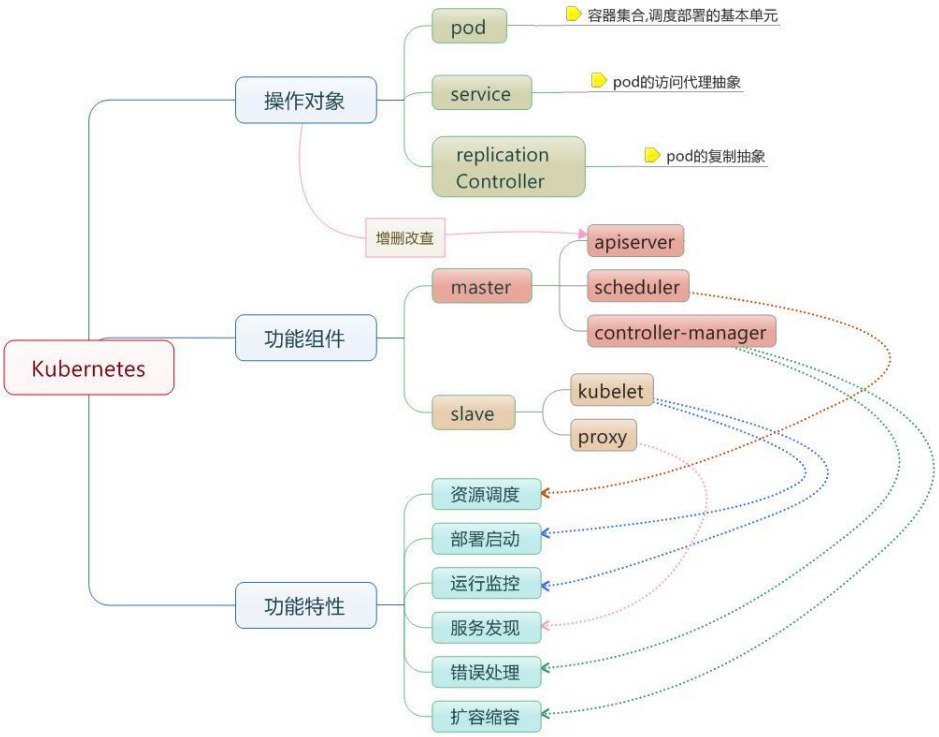
多 master 节点高可用架构图如下：



Kubernetes 组件：

Master：apiserver、scheduler、controller-manager、Etcd

Work：kubelet、kube-proxy、Calico、Coredns、docker



版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。

K8S 组件介绍：

- kubectl：管理 k8s 的命令行工具，可以操作 k8s 中的资源对象。

Matser 节点组件：

- etcd：是一个高可用的键值数据库，存储 k8s 的资源状态信息和网络信息的，etcd 中的数据变更是通过 api server 进行的。

- apiserver：提供 k8s api，是整个系统的对外接口，提供资源操作的唯一入口，供客户端和其它组件调用，提供了 k8s 各类资源对象（pod、deployment、Service 等）的增删改查，是整个系统的数据总线和数据中心，并提供认证、授权、访问控制、API 注册和发现等机制，并将操作对象持久化到 etcd 中。

- scheduler：负责 k8s 集群中 pod 的调度的，scheduler 通过与 apiserver 交互监听到创建 Pod 副本的信息后，它会检索所有符合该 Pod 要求的工作节点列表，开始执行 Pod 调度逻辑。调度成功后将 Pod 绑定到目标节点上。

- controller-manager：作为集群内部的管理控制中心，负责集群内的 Node、Pod 副本、服务端点（Endpoint）、命名空间（Namespace）、服务账号（ServiceAccount）、资源定额（ResourceQuota）的管理，当某个 Node 意外宕机时，Controller Manager 会及时发现并执行自动化修复流程，确保集群始终处于预期的工作状态。

- kubelet：每个 Node 节点上的 kubelet 定期就会调用 API Server 的 REST 接口报告自身状态，API Server 接收这些信息后，将节点状态信息更新到 etcd 中。kubelet 也通过 API Server 监听 Pod 信息，从而对 Node 机器上的 POD 进行管理，如创建、删除、更新 Pod。

- kube-proxy：提供网络代理和负载均衡，是实现 service 的通信与负载均衡机制的重要组件，kube-proxy 负责为 Pod 创建代理服务，从 apiserver 获取所有 service 信息，并根据 service 信息创建代理服务，实现 service 到 Pod 的请求路由和转发，从而实现 K8s 层级的虚拟转发网络，将到 service 的请求转发到后端的 pod 上。

- Calico：Calico 是一个纯三层的网络插件，calico 的 bgp 模式类似于 flannel 的 host-gw，calico 在 kubernetes 中可提供网络功能和网络策略。

Node 节点组件：

- Coredns：k8s1.11 之前使用的是 kube dns，1.11 之后才有 coredns，coredns 是一个 DNS 服务器，能够为 Kubernetes services 提供 DNS 记录。

- Calico：Calico 是一个纯三层的网络插件，calico 的 bgp 模式类似于

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。

flannel 的 host-gw，calico 在 kubernetes 中可提供网络功能和网络策略。

4、Kubernetes 容器编排工具的优势

（1）灵活部署

kubernetes 支持在多种平台部署，可在私有云、公有云、混合云、openstack、VMware vSphere、VMware Workstation、虚拟机、物理机等环境部署。

（2）安全高效，拥有完善的认证授权机制，自带审计功能

可以对多用户做细化的授权管理（如 rbac 授权），达到相互之间的操作完全隔离，互不影响，而且自身带有审计功能，可以对操作过程进行实时的日志记录，出现问题可以方便排查。

（3）负载均衡

支持四层、七层负载均衡，可用于多种场景。

（4）可扩展性强

拥有强大的集群扩展能力，可以根据业务规模自动增加和缩减主机节点的数量，确保服务可以承受大量并发带来的压力，保证业务稳定运行。

（5）根据节点资源的使用情况对 pod 进行合理的调度

可以按照用户需要调度 pod，例如保证 Pod 只在资源足够的节点上运行，会尝试把同一功能的 pod 分散在不同的节点上，还会尝试平衡不同节点的资源使用率等。

（6）拥有完善的灾备预警方案

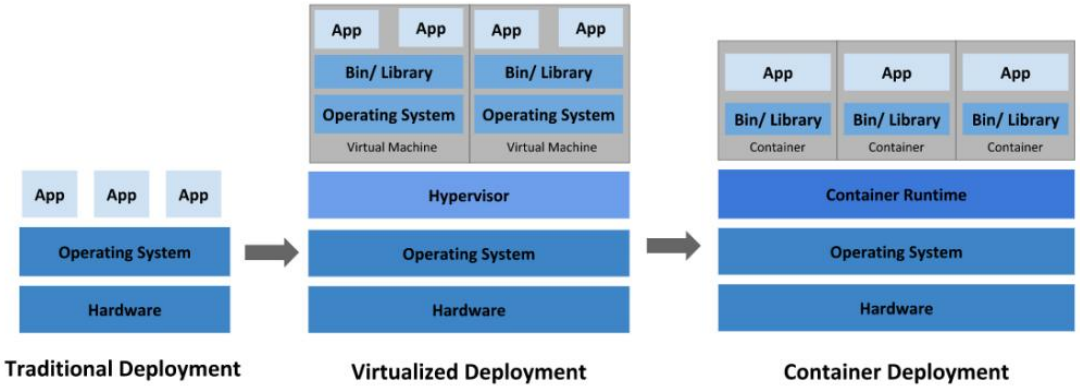
拥有多种灾备解决方案，支持备份和容灾，出现故障可以达到秒级切换，保证线上业务不受影响。

5、容器部署和传统部署对比分析

如下图：“传统部署、虚拟化部署、容器部署业务对比分析图”

Traditional Deployment（传统部署） -> Virtualized Deployment（虚拟化部署） -> Container Deployment（容器部署）

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。



（1）传统部署时代：

早期，应用程序在物理服务器上运行。无法为物理服务器中的应用程序定义资源边界，这会导致资源分配问题。例如，如果在物理服务器上运行多个应用程序，则可能会出现一个应用程序占用大部分资源的情况，结果可能导致其他应用程序的性能下降。一种解决方案是在不同的物理服务器上运行每个应用程序，但是由于资源利用不足而无法扩展，并且许多物理服务器的维护成本也很高。

（2）虚拟化部署时代：

作为解决方案，引入了虚拟化功能，它允许你在单个物理服务器的 CPU 上运行多个虚拟机（VM）。虚拟化功能允许应用程序在 VM 之间隔离，并提供安全级别，因为一个应用程序的信息不能被另一应用程序自由地访问。因为虚拟化可以轻松地添加或更新应用程序、降低硬件成本等，所以虚拟化可以更好地利用物理服务器中的资源，并可以实现更好的可伸缩性。每个 VM 是一台完整的计算机，在虚拟化硬件之上运行所有组件，包括其自己的操作系统。

（3）容器部署时代：

容器类似 VM，但是它们具有轻量级的隔离属性，可以在应用程序之间共享操作系统（OS）。因此，容器被认为是轻量级的。容器与 VM 类似，具有自己的文件系统、CPU、内存、进程空间等。由于它们与基础架构分离，因此可以跨云和 OS 分发进行移植。容器因具有许多优势而变得流行起来。下面列出了容器的一些好处：

- 敏捷应用程序的创建和部署：
  与使用 VM 镜像相比，提高了容器镜像创建的简便性和效率。

- 持续开发、集成和部署：
  通过快速简单的回滚(由于镜像不可变性)，提供可靠且频繁的容器镜像构建和部署。

- 关注开发与运维的分离：
  在构建/发布时而不是在部署时创建应用程序容器镜像，从而将应用程序与

基础架构分离。

- 可观察性：
  不仅可以显示操作系统级别的信息和指标，还可以显示应用程序的运行状况和其他指标信号。

- 跨开发、测试和生产的环境一致性：
  在便携式计算机上与在云中相同地运行。

- 云和操作系统分发的可移植性：
  可在 Ubuntu、RHEL、CoreOS、本地、Google Kubernetes Engine 和其他任何地方运行。

- 以应用程序为中心的管理：
  提高抽象级别，从在虚拟硬件上运行 OS 到在 OS 上运行应用程序。

- 松散耦合、分布式、弹性、解放的微服务：
  应用程序被分解成较小的独立部分，并且可以动态部署和管理 - 而不是在一台大型单机上整体运行。

- 资源隔离：
  可预测的应用程序性能。

- 资源利用：
  高效率和高密度。

6、Kubernetes 的基本概念和术语

Kubernetes 中的大部分概念如 Node、Pod、Replication Controller、Service 等都可以被看作一种资源对象，几乎所有资源对象都可以通过 Kubernetes 提供的 kubectl 工具（或者 API 编程调用）执行增、删、改、查等操作并将其保存在 etcd 中持久化存储。从这个角度来看，Kubernetes 其实是一个高度自动化的资源控制系统，它通过跟踪对比 etcd 库里保存的“资源期望状态”与当前环境中的“实际资源状态”的差异来实现自动控制和自动纠错的高级功能。

在声明一个 Kubernetes 资源对象的时候，需要注意一个关键属性：apiVersion。以下面的 Pod 声明为例，可以看到 Pod 这种资源对象归属于 v1 这个核心 API。

```
apiVersion: v1
kind: Pod
metadata:
  name: myweb
  labels:
```

```
    name: myweb
spec:
  containers:
  - name: myweb
    image: kubeguide/tomcat-app:v1
    ports:
    - containerPort: 8080
```

Kubernetes 平台采用了“核心+外围扩展”的设计思路，在保持平台核心稳定的同时具备持续演进升级的优势。Kubernetes 大部分常见的核心资源对象都归属于 v1 这个核心 API，比如 Node、Pod、Service、Endpoints、Namespace、RC、PersistentVolume 等。在版本迭代过程中，Kubernetes 先后扩展了 extensions/v1beta1、apps/v1beta1、apps/v1beta2 等 API 组，而在 1.9 版本之后引入了 apps/v1 这个正式的扩展 API 组，正式淘汰（deprecated）了 extensions/v1beta1、apps/v1beta1、apps/v1beta2 这三个 API 组。

我们可以采用 YAML 或 JSON 格式声明（定义或创建）一个 Kubernetes 资源对象，每个资源对象都有自己的特定语法格式（可以理解为数据库中一个特定的表），但随着 Kubernetes 版本的持续升级，一些资源对象会不断引入新的属性。为了在不影响当前功能的情况下引入对新特性的支持，我们通常会采用下面两种典型方法。

- 方法一：在设计数据库表的时候，在每个表中都增加一个很长的备注字段，之后扩展的数据以某种格式（如 XML、JSON、简单字符串拼接等）放入备注字段。因为数据库表的结构没有发生变化，所以此时程序的改动范围是最小的，风险也更小，但看起来不太美观。
- 方法二：直接修改数据库表，增加一个或多个新的列，此时程序的改动范围较大，风险更大，但看起来比较美观。

显然，两种方法都不完美。更加优雅的做法是，先采用方法 1 实现这个新特性，经过几个版本的迭代，等新特性变得稳定成熟了以后，可以在后续版本中采用方法 2 升级到正式版。为此，Kubernetes 为每个资源对象都增加了类似数据库表里备注字段的通用属性 Annotations，以实现方法 1 的升级。以 Kubernetes 1.3 版本引入的 Pod 的 Init Container 新特性为例，一开始，Init Container 的定义是在 Annotations 中声明的，如下面代码中粗体部分所示，是不是很不美观？

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
  annotations:
    pod.beta.kubernetes.io/init-containers: '[
```

```
        {
          "name": "init-mydb",
          "image": "busybox",
          "command": [......]
        }
      ]'
spec:
  containers:
......
```

在 Kubernetes 1.8 版本以后，Init container 特性完全成熟，其定义被放入 Pod 的 spec.initContainers 一节，看起来优雅了很多：

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  # These containers are run during pod initialization
  initContainers:
  - name: init-mydb
    image: busybox
    command:
    - xxx
```

在 Kubernetes 1.8 中，资源对象中的很多 Alpha、Beta 版本的 Annotations 被取消，升级成了常规定义方式，在学习 Kubernetes 的过程中需要特别注意。下面介绍 Kubernetes 中重要的资源对象。

6.1、Master

Kubernetes 里的 Master 指的是集群控制节点，在每个 Kubernetes 集群里都需要有一个 Master 来负责整个集群的管理和控制，基本上 Kubernetes 的所有控制命令都发给它，它负责具体的执行过程，我们后面执行的所有命令基本都是在 Master 上运行的。Master 通常会占据一个独立的服务器（高可用部署建议用 3 台服务器），主要原因是它太重要了，是整个集群的“首脑”，如果它宕机或者不可用，那么对集群内容器应用的管理都将失效。

在 Master 上运行着以下关键进程。

- Kubernetes API Server（kube-apiserver）：提供了 HTTP Rest 接口的关键服务进程，是 Kubernetes 里所有资源的增、删、改、查等操作的唯一入口，也是集群控制的入口进程。
- Kubernetes Controller Manager（kube-controller-manager）：

Kubernetes 里所有资源对象的自动化控制中心，可以将其理解为资源对象的“大总管”。

● Kubernetes Scheduler（kube-scheduler）：负责资源调度（Pod 调度）的进程，相当于公交公司的“调度室”。

另外，在 Master 上通常还需要部署 etcd 服务，因为 Kubernetes 里的所有资源对象的数据都被保存在 etcd 中。

6.2、Node

除了 Master，Kubernetes 集群中的其他机器被称为 Node。与 Master 一样，Node 可以是一台物理主机，也可以是一台虚拟机。Node 是 Kubernetes 集群中的工作负载节点，每个 Node 都会被 Master 分配一些工作负载（Docker 容器），当某个 Node 宕机时，其上的工作负载会被 Master 自动转移到其他节点上。

在每个 Node 上都运行着以下关键进程。

● kubelet：负责 Pod 对应的容器的创建、启停等任务，同时与 Master 密切协作，实现集群管理的基本功能。

● kube-proxy：实现 Kubernetes Service 的通信与负载均衡机制的重要组件。

● Docker Engine（docker）：Docker 引擎，负责本机的容器创建和管理工作。

Node 可以在运行期间动态增加到 Kubernetes 集群中，前提是在这个节点上已经正确安装、配置和启动了上述关键进程，在默认情况下 kubelet 会向 Master 注册自己，这也是 Kubernetes 推荐的 Node 管理方式。一旦 Node 被纳入集群管理范围，kubelet 进程就会定时向 Master 汇报自身的情报，例如操作系统、Docker 版本、机器的 CPU 和内存情况，以及当前有哪些 Pod 在运行等，这样 Master 就可以获知每个 Node 的资源使用情况，并实现高效均衡的资源调度策略。而某个 Node 在超过指定时间不上报信息时，会被 Master 判定为“失联”，Node 的状态被标记为不可用（Not Ready），随后 Master 会触发“工作负载大转移”的自动流程。

我们可以执行下述命令查看在集群中有多少个 Node：

```
[root@master ~]# kubectl get nodes
NAME     STATUS   ROLES    AGE   VERSION
master   Ready    master   26m   v1.23.3
node-1   Ready    <none>   22m   v1.23.3
```

然后，通过 kubectl describe node <node_name>查看某个 Node 的详细信息：

```
[root@master ~]# kubectl describe node node-1
Name:               node-1
Roles:              <none>
```

```
    Labels:             beta.kubernetes.io/arch=amd64
                        beta.kubernetes.io/os=linux
                        kubernetes.io/arch=amd64
                        kubernetes.io/hostname=node-1
                        kubernetes.io/os=linux
    Annotations:                        kubeadm.alpha.kubernetes.io/cri-socket:
/var/run/dockershim.sock
                        node.alpha.kubernetes.io/ttl: 0
                        volumes.kubernetes.io/controller-managed-attach-detach:
true
    CreationTimestamp:  Wed, 21 Oct 2020 09:30:04 -0400
    Taints:             <none>
    Unschedulable:      false
    Lease:
      HolderIdentity:  node-1
      AcquireTime:     <unset>
      RenewTime:       Thu, 22 Oct 2020 08:34:24 -0400
    Conditions:
      Type                                          Status     LastHeartbeatTime
LastTransitionTime                Reason                        Message
      ----                                          ------     -----------------
------------------                ------                        -------
      NetworkUnavailable   False   Wed, 21 Oct 2020 09:51:57 -0400   Wed, 21 Oct
2020 09:51:57 -0400   WeaveIsUp                     Weave pod has set this
      MemoryPressure       False   Wed, 21 Oct 2020 10:17:39 -0400   Wed, 21 Oct
2020 09:30:04 -0400   KubeletHasSufficientMemory   kubelet has sufficient memory
available
      DiskPressure         False   Wed, 21 Oct 2020 10:17:39 -0400   Wed, 21 Oct
2020 09:30:04 -0400   KubeletHasNoDiskPressure     kubelet has no disk pressure
      PIDPressure          False   Wed, 21 Oct 2020 10:17:39 -0400   Wed, 21 Oct
2020 09:30:04 -0400   KubeletHasSufficientPID      kubelet has sufficient PID
available
      Ready                True    Wed, 21 Oct 2020 10:17:39 -0400   Wed, 21 Oct
2020 09:52:06 -0400   KubeletReady                 kubelet is posting ready status
    Addresses:
      InternalIP:  192.168.43.102
      Hostname:    node-1
    Capacity:
      cpu:                4
      ephemeral-storage:  17394Mi
      hugepages-1Gi:      0
      hugepages-2Mi:      0
```

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。

```
      memory:             1872772Ki
      pods:               110
    Allocatable:
      cpu:                4
      ephemeral-storage:  16415037823
      hugepages-1Gi:      0
      hugepages-2Mi:      0
      memory:             1770372Ki
      pods:               110
    System Info:
      Machine ID:                 96ecd11ac10d4bc9baa908c01f4fb99f
      System UUID:                564D2EA7-1749-D121-FDF5-D8F1E7A0BD5F
      Boot ID:                    e3956228-2661-409e-8b86-b212f1291d71
      Kernel Version:             3.10.0-1127.el7.x86_64
      OS Image:                   CentOS Linux 7 (Core)
      Operating System:           linux
      Architecture:               amd64
      Container Runtime Version:  docker://1.13.1
      Kubelet Version:            v1.19.3
      Kube-Proxy Version:         v1.19.3
    Non-terminated Pods:          (2 in total)
      Namespace                   Name                CPU Requests  CPU Limits
Memory Requests  Memory Limits  AGE
      ---------                   ----                ------------  ----------
---------------  -------------  ---
      kube-system                 kube-proxy-4scmr    0 (0%)        0 (0%)
0 (0%)           0 (0%)         23h
      kube-system                 weave-net-q7pw6     100m (2%)     0 (0%)
200Mi (11%)      0 (0%)         22h
    Allocated resources:
      (Total limits may be over 100 percent, i.e., overcommitted.)
      Resource           Requests     Limits
      --------           --------     ------
      cpu                100m (2%)    0 (0%)
      memory             200Mi (11%)  0 (0%)
      ephemeral-storage  0 (0%)       0 (0%)
      hugepages-1Gi      0 (0%)       0 (0%)
      hugepages-2Mi      0 (0%)       0 (0%)
    Events:
```

上述命令展示了 Node 的如下关键信息：

- Node 的基本信息：名称、标签、创建时间等。
- Node 当前的运行状态：Node 启动后会做一系列的自检工作，比如磁盘

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。

空间是否不足（DiskPressure）、内存是否不足（MemoryPressure）、网络是否正常（NetworkUnavailable）、PID 资源是否充足（PIDPressure）。在一切正常时设置 Node 为 Ready 状态（Ready=True），该状态表示 Node 处于健康状态，Master 将可以在其上调度新的任务了（如启动 Pod）。

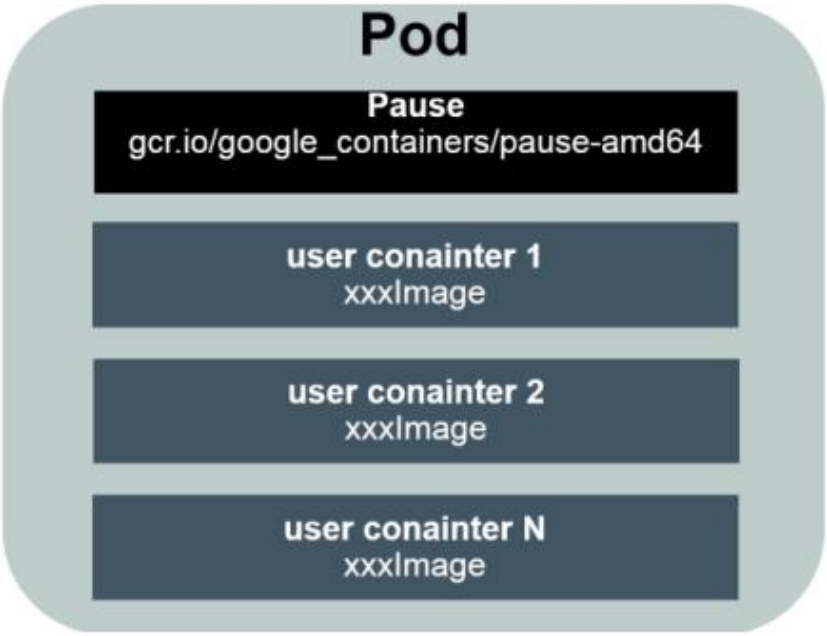
- Node 的主机地址与主机名。
- Node 上的资源数量：描述 Node 可用的系统资源，包括 CPU、内存数量、最大可调度 Pod 数量等。
- Node 可分配的资源量：描述 Node 当前可用于分配的资源量。
- 主机系统信息：包括主机 ID、系统 UUID、Linux kernel 版本号、操作系统类型与版本、Docker 版本号、kubelet 与 kube-proxy 的版本号等。
- 当前运行的 Pod 列表概要信息。
- 已分配的资源使用概要信息，例如资源申请的最低、最大允许使用量占系统总量的百分比。
- Node 相关的 Event 信息。

6.3、Pod

Pod 是 Kubernetes 最重要的基本概念，如下图所示是 Pod 的组成示意图，我们看到每个 Pod 都有一个特殊的被称为“根容器”的 Pause 容器。Pause 容器对应的镜像属于 Kubernetes 平台的一部分，除了 Pause 容器，每个 Pod 还包含一个或多个紧密相关的用户业务容器。

Pause 也叫做初始化容器，主要为 Pod 提供共享网络、Volume 等。

Pod 的组成示意图：



为什么 Kubernetes 会设计出一个全新的 Pod 的概念并且 Pod 有这样特殊的组成结构？

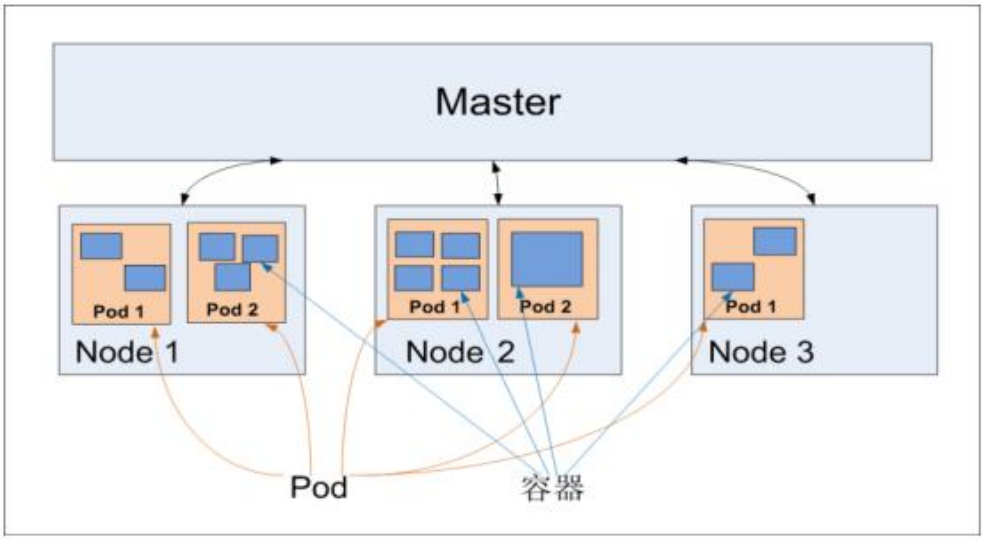
原因之一：在一组容器作为一个单元的情况下，我们难以简单地对“整体”进行判断及有效地行动。比如，一个容器死亡了，此时算是整体死亡么？是 N/M 的死亡率么？引入业务无关并且不易死亡的 Pause 容器作为 Pod 的根容器，以它的状态代表整个容器组的状态，就简单、巧妙地解决了这个难题。

原因之二：Pod 里的多个业务容器共享 Pause 容器的 IP，共享 Pause 容器挂接的 Volume，这样既简化了密切关联的业务容器之间的通信问题，也很好地解决了它们之间的文件共享问题。

Kubernetes 为每个 Pod 都分配了唯一的 IP 地址，称之为 Pod IP，一个 Pod 里的多个容器共享 Pod IP 地址。Kubernetes 要求底层网络支持集群内任意两个 Pod 之间的 TCP/IP 直接通信，这通常采用虚拟二层网络技术来实现，例如 Flannel、Open vSwitch 等，因此我们需要牢记一点：在 Kubernetes 里，一个 Pod 里的容器与另外主机上的 Pod 容器能够直接通信。

Pod 其实有两种类型：普通的 Pod 及静态 Pod（Static Pod）。后者比较特殊，它并没被存放在 Kubernetes 的 etcd 存储里，而是被存放在某个具体的 Node 上的一个具体文件中，并且只在此 Node 上启动、运行。而普通的 Pod 一旦被创建，就会被放入 etcd 中存储，随后会被 Kubernetes Master 调度到某个具体的 Node 上并进行绑定（Binding），随后该 Pod 被对应的 Node 上的 kubelet 进程实例化成一组相关的 Docker 容器并启动。在默认情况下，当 Pod 里的某个容器停止时，Kubernetes 会自动检测到这个问题并且重新启动这个 Pod（重启 Pod 里的所有容器），如果 Pod 所在的 Node 宕机，就会将这个 Node 上的所有 Pod 重新调度到其他节点上。

Pod、容器与 Node 的关系如下图所示：



Kubernetes 里的所有资源对象都可以采用 YAML 或者 JSON 格式的文件来定义或描述，下面是我们在之前的 Hello World 例子里用到的 myweb 这个 Pod 的资源定义文件：

```
apiVersion: v1
```

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。

```
  kind: Pod
  metadata:
    name: myweb
    labels:
      name: myweb
  spec:
    containers:
    - name: myweb
      image: kubeguide/tomcat-app:v1
      ports:
      - containerPort: 8080
      env:
      - name: MYSQL_SERVICE_HOST
        value: 'mysql'
      - name: MYSQL_SERVICE_PORT
        value: '3306'
```

Kind 为 Pod 表明这是一个 Pod 的定义，metadata 里的 name 属性为 Pod 的名称，在 metadata 里还能定义资源对象的标签，这里声明 myweb 拥有一个 name=myweb 的标签。在 Pod 里所包含的容器组的定义则在 spec 一节中声明，这里定义了一个名为 myweb、对应镜像为 kubeguide/tomcat-app:v1 的容器，该容器注入了名为 MYSQL_SERVICE_HOST='mysql'和 MYSQL_SERVICE_PORT='3306'的环境变量（env 关键字），并且在 8080 端口（containerPort）启动容器进程。Pod 的 IP 加上这里的容器端口（containerPort），组成了一个新的概念——Endpoint，它代表此 Pod 里的一个服务进程的对外通信地址。一个 Pod 也存在具有多个 Endpoint 的情况，比如当我们把 Tomcat 定义为一个 Pod 时，可以对外暴露管理端口与服务端口这两个 Endpoint。

我们所熟悉的 Docker Volume 在 Kubernetes 里也有对应的概念——Pod Volume，后者有一些扩展，比如可以用分布式文件系统 GlusterFS 实现后端存储功能；Pod Volume 是被定义在 Pod 上，然后被各个容器挂载到自己的文件系统中的。

这里顺便提一下 Kubernetes 的 Event 概念。Event 是一个事件的记录，记录了事件的最早产生时间、最后重现时间、重复次数、发起者、类型，以及导致此事件的原因等众多信息。Event 通常会被关联到某个具体的资源对象上，是排查故障的重要参考信息，之前我们看到 Node 的描述信息包括了 Event，而 Pod 同样有 Event 记录，当我们发现某个 Pod 迟迟无法创建时，可以用 kubectl describe pod xxxx 来查看它的描述信息，以定位问题的成因，比如下面这个 Event 记录信息表明 Pod 里的一个容器被探针检测为失败一次：

```
  Events:
   FirstSeen LastSeenCount From SubobjectPath Type Reason
  Message
```

```
  ---------  -------------  ----  -------------  --------------
  -------
   10h  12m  32  {kubelet k8s-node-1}  spec.containers{kube2sky}
  Warning  Unhealthy  Liveness  probe  failed:  Get
http://172.17.1.2:8080/healthz:  net/http:  request  canceled
(Client.Timeout exceeded while awaiting headers)
```

每个 Pod 都可以对其能使用的服务器上的计算资源设置限额，当前可以设置限额的计算资源有 CPU 与 Memory 两种，其中 CPU 的资源单位为 CPU（Core）的数量，是一个绝对值而非相对值。

对于绝大多数容器来说，一个 CPU 的资源配额相当大，所以在 Kubernetes 里通常以千分之一的 CPU 配额为最小单位，用 m 来表示。通常一个容器的 CPU 配额被定义为 100～300m，即占用 0.1～0.3 个 CPU。由于 CPU 配额是一个绝对值，所以无论在拥有一个 Core 的机器上，还是在拥有 48 个 Core 的机器上，100m 这个配额所代表的 CPU 的使用量都是一样的。与 CPU 配额类似，Memory 配额也是一个绝对值，它的单位是内存字节数。

在 Kubernetes 里，一个计算资源进行配额限定时需要设定以下两个参数。

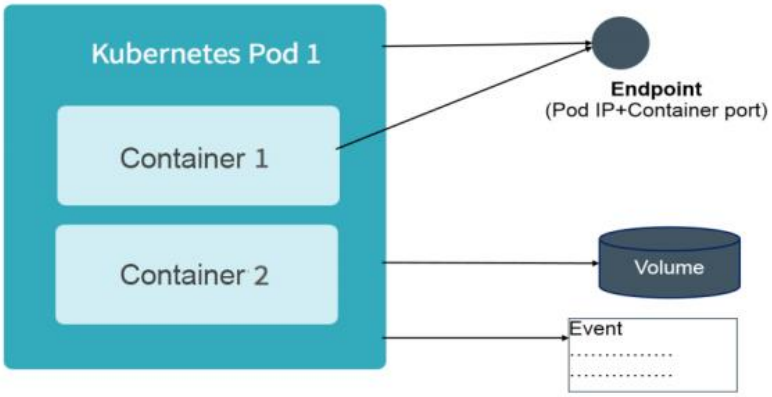
- Requests：该资源的最小申请量，系统必须满足要求。
- Limits：该资源最大允许使用的量，不能被突破，当容器试图使用超过这个量的资源时，可能会被 Kubernetes“杀掉”并重启。

通常，我们会把 Requests 设置为一个较小的数值，符合容器平时的工作负载情况下的资源需求，而把 Limit 设置为峰值负载情况下资源占用的最大量。下面这段定义表明 MySQL 容器申请最少 0.25 个 CPU 及 64MiB 内存，在运行过程中 MySQL 容器所能使用的资源配额为 0.5 个 CPU 及 128MiB 内存：

```
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

最后给出 Pod 及 Pod 周边对象的示意图作为总结，如下图所示，后面部分还会涉及这张图里的对象和概念，以进一步加强理解。Pod 及周边对象：

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。



6.4、Label

Label（标签）是 Kubernetes 系统中另外一个核心概念。一个 Label 是一个 key=value 的键值对，其中 key 与 value 由用户自己指定。Label 可以被附加到各种资源对象上，例如 Node、Pod、Service、RC 等，一个资源对象可以定义任意数量的 Label，同一个 Label 也可以被添加到任意数量的资源对象上。Label 通常在资源对象定义时确定，也可以在对象创建后动态添加或者删除。

我们可以通过给指定的资源对象捆绑一个或多个不同的 Label 来实现多维度的资源分组管理功能，以便灵活、方便地进行资源分配、调度、配置、部署等管理工作。例如，部署不同版本的应用到不同的环境中；监控和分析应用（日志记录、监控、告警）等。

一些常用的 Label 示例如下：

- 版本标签
  "release":"stable"
  "release":"canary"
- 环境标签
  "environment":"dev"
  "environment":"qa"
  "environment":"production"
- 架构标签
  "tier":"frontend"
  "tier":"backend"
  "tier":"middleware"
- 分区标签
  "partition":"customerA"
  "partition":"customerB"
- 质量管控标签
  "track":"daily"
  "track":"weekly"

Label 相当于我们熟悉的“标签”。给某个资源对象定义一个 Label，就相当于给它打了一个标签，随后可以通过 Label Selector（标签选择器）查询和筛选拥有某些 Label 的资源对象，Kubernetes 通过这种方式实现了类似 SQL 的简单又通用的对象查询机制。

Label Selector 可以被类比为 SQL 语句中的 where 查询条件，例如，name=redis-slave 这个 Label Selector 作用于 Pod 时，可以被类比为 select * from pod where pods name = "redis-slave"这样的语句。当前有两种 Label Selector 表达式：基于等式的（Equality-based）和基于集合的（Set-based），前者采用等式类表达式匹配标签，下面是一些具体的例子：

- name = redis-slave：匹配所有具有标签 name=redis-slave 的资源对象。
- env != production：匹配所有不具有标签 env=production 的资源对象，比如 env=test 就是满足此条件的标签之一。

后者则使用集合操作类表达式匹配标签，下面是一些具体的例子。

- name in（redis-master, redis-slave）：匹配所有具有标签 name=redis-master 或者 name=redis-slave 的资源对象。
- name not in（php-frontend）：匹配所有不具有标签 name=php-frontend 的资源对象。

可以通过多个 Label Selector 表达式的组合实现复杂的条件选择，多个表达式之间用“,”进行分隔即可，几个条件之间是“AND”的关系，即同时满足多个条件，比如下面的例子：

```
name=redis-slave,env!=production
name notin (php-frontend),env!=production
```

以 myweb Pod 为例，Label 被定义在其 metadata 中：

```
apiVersion: v1
kind: Pod
metadata:
  name: myweb
  labels:
    app: myweb
```

管理对象 RC 和 Service 则通过 Selector 字段设置需要关联 Pod 的 Label：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myweb
spec:
```

```
    replicas: 1
    selector:
      app: myweb
    template:
    ......

apiVersion: v1
kind: Service
metadata:
  name: myweb
spec:
  selector:
    app: myweb
  ports:
  - port: 8080
```

其他管理对象如 Deployment、ReplicaSet、DaemonSet 和 Job 则可以在 Selector 中使用基于集合的筛选条件定义，例如：

```
  selector:
    matchLabels:
      app: myweb
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
      - {key: environment, operator: NotIn, values: [dev]}
```

matchLabels 用于定义一组 Label，与直接写在 Selector 中的作用相同；matchExpressions 用于定义一组基于集合的筛选条件，可用的条件运算符包括 In、NotIn、Exists 和 DoesNotExist。

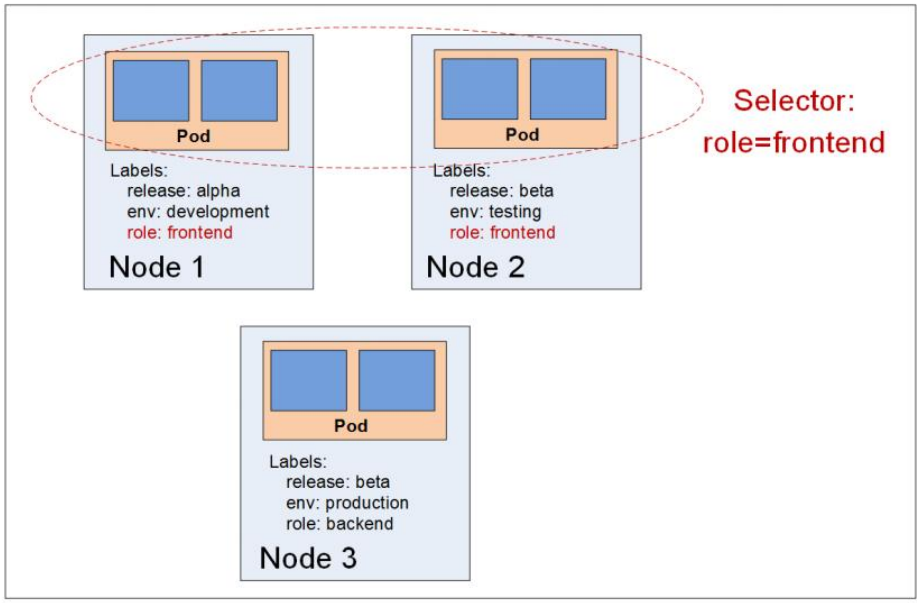
如果同时设置了 matchLabels 和 matchExpressions，则两组条件为 AND 关系，即需要同时满足所有条件才能完成 Selector 的筛选。

Label Selector 在 Kubernetes 中的重要使用场景如下。

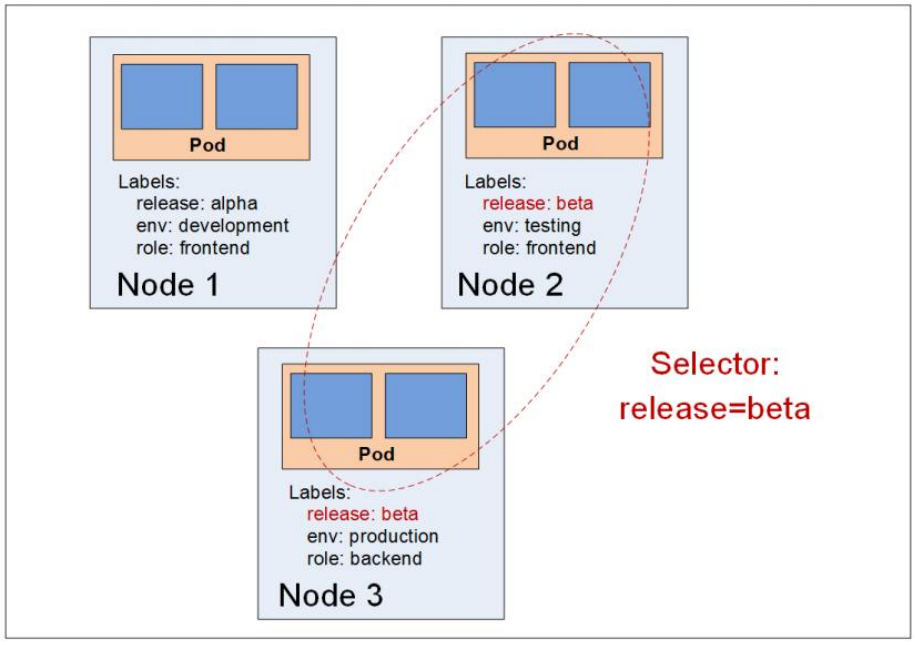
- kube-controller 进程通过在资源对象 RC 上定义的 Label Selector 来筛选要监控的 Pod 副本数量，使 Pod 副本数量始终符合预期设定的全自动控制流程。
- kube-proxy 进程通过 Service 的 Label Selector 来选择对应的 Pod，自动建立每个 Service 到对应 Pod 的请求转发路由表，从而实现 Service 的智能负载均衡机制。
- 通过对某些 Node 定义特定的 Label，并且在 Pod 定义文件中使用 NodeSelector 这种标签调度策略，kube-scheduler 进程可以实现 Pod 定向调度的特性。

看一个复杂的例子：假设为 Pod 定义了 3 个 Label：release、env 和 role，

不同的 Pod 定义了不同的 Label 值，如下图所示，如果设置“role=frontend”的 Label Selector，则会选取到 Node1 和 Node2 上的 Pod。



如果设置“release=beta”的 Label Selector，则会选取到 Node2 和 Node3 上的 Pod，如下图所示。



总之，使用 Label 可以给对象创建多组标签，Label 和 Label Selector 共同构成了 Kubernetes 系统中核心的应用模型，使得被管理对象能够被精细地分组管理，同时实现了整个集群的高可用性。

6.5、Replication Controller

RC 是 Kubernetes 系统中的核心概念之一，简单来说，它其实定义了一个期

望的场景，即声明某种 Pod 的副本数量在任意时刻都符合某个预期值，所以 RC 的定义包括如下几个部分：

- Pod 期待的副本数量。
- 用于筛选目标 Pod 的 Label Selector。
- 当 Pod 的副本数量小于预期数量时，用于创建新 Pod 的 Pod 模板（template）。

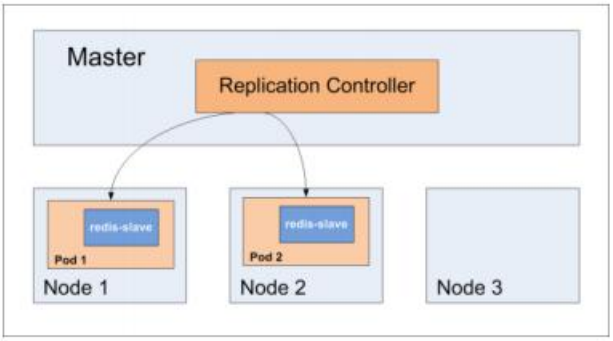
下面是一个完整的 RC 定义的例子，即确保拥有 tier=frontend 标签的这个 Pod（运行 Tomcat 容器）在整个 Kubernetes 集群中始终只有一个副本：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    tier: frontend
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
        - name: tomcat-damo
          image: tomcat
          imagePullPolicy: IfNotPresent
          env:
          - name: GENT_HOSTS_FROM
            value: dns
          ports:
          - containerPort: 80
```

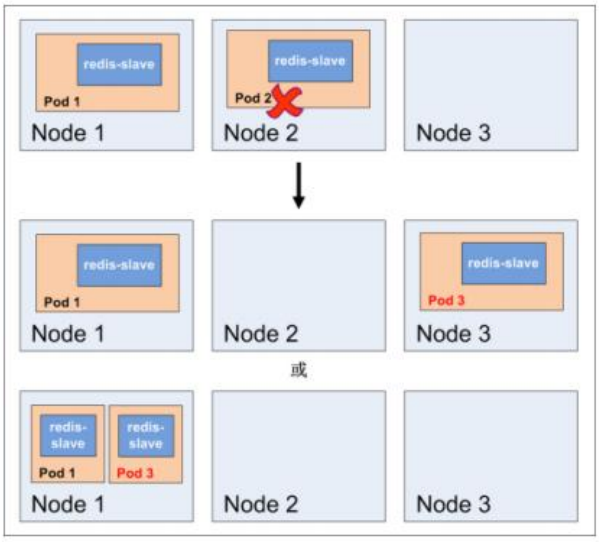
在我们定义了一个 RC 并将其提交到 Kubernetes 集群中后，Master 上的 Controller Manager 组件就得到通知，定期巡检系统中当前存活的目标 Pod，并确保目标 Pod 实例的数量刚好等于此 RC 的期望值，如果有过多的 Pod 副本在运行，系统就会停掉一些 Pod，否则系统会再自动创建一些 Pod。可以说，通过 RC，Kubernetes 实现了用户应用集群的高可用性，并且大大减少了系统管理员在传统 IT 环境中需要完成的许多手工运维工作（如主机监控脚本、应用监控脚本、故障恢复脚本等）。

下面以有 3 个 Node 的集群为例，说明 Kubernetes 如何通过 RC 来实现 Pod 副本数量自动控制的机制。假如在我们的 RC 里定义 redis-slave 这个 Pod 需要

保持两个副本，系统将可能在其中的两个 Node 上创建 Pod。下图描述了在两个 Node 上创建 redis-slave Pod 的情形。



假设 Node 2 上的 Pod 2 意外终止，则根据 RC 定义的 replicas 数量 2，Kubernetes 将会自动创建并启动一个新的 Pod，以保证在整个集群中始终有两个 redis-slave Pod 运行。如下图所示，系统可能选择 Node 3 或者 Node 1 来创建一个新的 Pod。

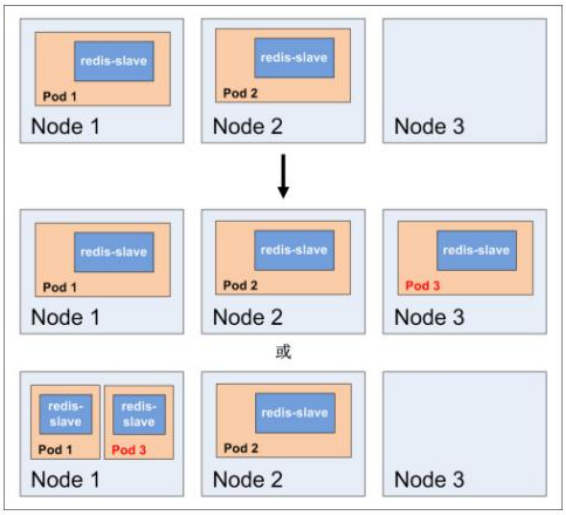


此外，在运行时，我们可以通过修改 RC 的副本数量，来实现 Pod 的动态缩放（Scaling），这可以通过执行 kubectl scale 命令来一键完成：

```
$ kubectl scale rc redis-slave --replicas=3
scaled
```

Scaling 的执行结果如下图所示。

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。



需要注意的是，删除 RC 并不会影响通过该 RC 已创建好的 Pod。为了删除所有 Pod，可以设置 replicas 的值为 0，然后更新该 RC。另外，kubectl 提供了 stop 和 delete 命令来一次性删除 RC 和 RC 控制的全部 Pod。

应用升级时，通常会使用一个新的容器镜像版本替代旧版本。我们希望系统平滑升级，比如在当前系统中有 10 个对应的旧版本的 Pod，则最佳的系统升级方式是旧版本的 Pod 每停止一个，就同时创建一个新版本的 Pod，在整个升级过程中此消彼长，而运行中的 Pod 数量始终是 10 个，几分钟以后，当所有的 Pod 都已经是新版本时，系统升级完成。通过 RC 机制，Kubernetes 很容易就实现了这种高级实用的特性，被称为“滚动升级”（Rolling Update）。

Replication Controller 由于与 Kubernetes 代码中的模块 Replication Controller 同名，同时“Replication Controller”无法准确表达它的本意，所以在 Kubernetes 1.2 中，升级为另外一个新概念—Replica Set，官方解释其为“下一代的 RC”。Replica Set 与 RC 当前的唯一区别是，Replica Sets 支持基于集合的 Label selector（Set-based selector），而 RC 只支持基于等式的 Label Selector（equality-based selector），这使得 Replica Set 的功能更强。下面是等价于之前 RC 例子的 Replica Set 的定义（省去了 Pod 模板部分的内容）：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
```

```
      ......
```

kubectl 命令行工具适用于 RC 的绝大部分命令同样适用于 Replica Set。此外，我们当前很少单独使用 Replica Set，它主要被 Deployment 这个更高层的资源对象所使用，从而形成一整套 Pod 创建、删除、更新的编排机制。我们在使用 Deployment 时，无须关心它是如何创建和维护 Replica Set 的，这一切都是自动发生的。

Replica Set 与 Deployment 这两个重要的资源对象逐步替代了之前 RC 的作用，是 Kubernetes 1.3 里 Pod 自动扩容（伸缩）这个告警功能实现的基础，也将继续在 Kubernetes 未来的版本中发挥重要的作用。

最后总结一下 RC（Replica Set）的一些特性与作用。

- 在大多数情况下，我们通过定义一个 RC 实现 Pod 的创建及副本数量的自动控制。
- 在 RC 里包括完整的 Pod 定义模板。
- RC 通过 Label Selector 机制实现对 Pod 副本的自动控制。
- 通过改变 RC 里的 Pod 副本数量，可以实现 Pod 的扩容或缩容。
- 通过改变 RC 里 Pod 模板中的镜像版本，可以实现 Pod 的滚动升级。

6.6、Deployment

Deployment 是 Kubernetes 在 1.2 版本中引入的新概念，用于更好地解决 Pod 的编排问题。为此，Deployment 在内部使用了 Replica Set 来实现目的，无论从 Deployment 的作用与目的、YAML 定义，还是从它的具体命令行操作来看，我们都可以把它看作 RC 的一次升级，两者的相似度超过 90%。

Deployment 相对于 RC 的一个最大升级是我们可以随时知道当前 Pod“部署”的进度。实际上由于一个 Pod 的创建、调度、绑定节点及在目标 Node 上启动对应的容器这一完整过程需要一定的时间，所以我们期待系统启动 N 个 Pod 副本的目标状态，实际上是一个连续变化的“部署过程”导致的最终状态。

Deployment 的典型使用场景有以下几个。

- 创建一个 Deployment 对象来生成对应的 Replica Set 并完成 Pod 副本的创建。
- 检查 Deployment 的状态来看部署动作是否完成（Pod 副本数量是否达到预期的值）。
- 更新 Deployment 以创建新的 Pod（比如镜像升级）。
- 如果当前 Deployment 不稳定，则回滚到一个早先的 Deployment 版本。
- 暂停 Deployment 以便于一次性修改多个 PodTemplateSpec 的配置项，之后再恢复 Deployment，进行新的发布。
- 扩展 Deployment 以应对高负载。
- 查看 Deployment 的状态，以此作为发布是否成功的指标。
- 清理不再需要的旧版本 ReplicaSets。

除了 API 声明与 Kind 类型等有所区别，Deployment 的定义与 Replica Set 的定义很类似：

```
apiVersion: apps/v1                    apiVersion: v1
kind: Deployment                       kind: ReplicaSet
metadata:                              metadata:
  name: nginx-deployment                 name: nginx-repset
```

下面通过运行一些例子来直观地感受 Deploymen 的概念。创建一个名为 tomcat-deployment.yaml 的 Deployment 描述文件，内容如下：

```
[root@master-node ~]# vi tomcat-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
      - name: tomcat-demo
        image: tomcat
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

运行下述命令创建 Deployment：

```
[root@master ~]# kubectl create -f tomcat-deployment.yaml
deployment.apps/frontend created
```

运行下述命令查看 Deployment 的信息：

```
[root@master ~]# kubectl get deployments
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
frontend   1/1     1            1           10m
```

对上述输出中涉及的数量解释如下。

- DESIRED：Pod 副本数量的期望值，即在 Deployment 里定义的 Replica。
- CURRENT：当前 Replica 的值，实际上是 Deployment 创建的 Replica Set 里的 Replica 值，这个值不断增加，直到达到 DESIRED 为止，表明整个部署过程完成。
- UP-TO-DATE：最新版本的 Pod 的副本数量，用于指示在滚动升级的过程中，有多少个 Pod 副本已经成功升级。
- AVAILABLE：当前集群中可用的 Pod 副本数量，即集群中当前存活的 Pod 数量。

运行下述命令查看对应的 Replica Set，我们看到它的命名与 Deployment 的名称有关系：

```
[root@master ~]# kubectl get rs
NAME                  DESIRED   CURRENT   READY   AGE
frontend-7d7c57fc94   1         1         0       18m
```

运行下述命令查看创建的 Pod，我们发现 Pod 的命名以 Deployment 对应的 Replica Set 的名称为前缀，这种命名很清晰地表明了一个 Replica Set 创建了哪些 Pod，对于 Pod 滚动升级这种复杂的过程来说，很容易排查错误：

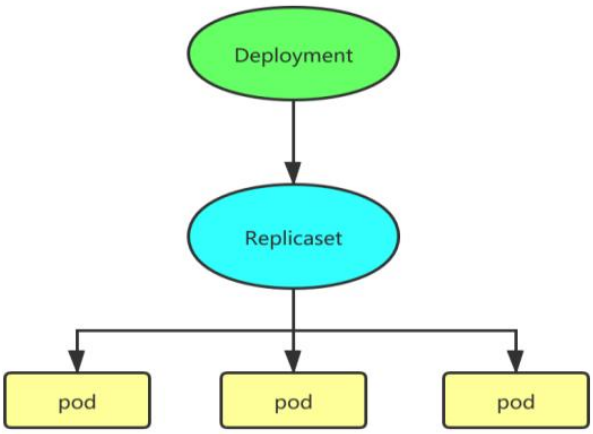
```
[root@master ~]# kubectl get pod
NAME                  READY   STATUS              RESTARTS   AGE
frontend-7d7c57fc94-vrzfc   0/1   ContainerCreating   0        11m
```

运行 kubectl describe deployments，可以清楚地看到 Deployment 控制的 Pod 的水平扩展过程，这里不再赘述。

Pod 的管理对象，除了 RC 和 Deployment，还包括 ReplicaSet、DaemonSet、StatefulSet、Job 等，分别用于不同的应用场景中。

总结：

Deployment 管理 Replicaset 和 Pod 的副本控制器，Deployment 可以管理多个 Replicaset，是比 Replicaset 更高级的控制器，也即是说在创建 Deployment 的时候，会自动创建 Replicaset，由 Replicaset 再创建 Pod，Deployment 能对 Pod 扩容、缩容、滚动更新和回滚、维持 Pod 数量。



6.7、StatefulSet

在 Kubernetes 系统中，Pod 的管理对象 RC、Deployment、DaemonSet 和 Job 都面向无状态的服务。但现实中有很多服务是有状态的，特别是一些复杂的中间件集群，例如 MySQL 集群、MongoDB 集群、Kaka 集群、ZooKeeper 集群等，这些应用集群有 4 个共同点。

（1）每个节点都有固定的身份 ID，通过这个 ID，集群中的成员可以相互发现并通信。

（2）集群的规模是比较固定的，集群规模不能随意变动。

（3）集群中的每个节点都是有状态的，通常会持久化数据到永久存储中。

（4）如果磁盘损坏，则集群里的某个节点无法正常运行，集群功能受损。

如果通过 RC 或 Deployment 控制 Pod 副本数量来实现上述有状态的集群，就会发现第 1 点是无法满足的，因为 Pod 的名称是随机产生的，Pod 的 IP 地址也是在运行期才确定且可能有变动的，我们事先无法为每个 Pod 都确定唯一不变的 ID。另外，为了能够在其他节点上恢复某个失败的节点，这种集群中的 Pod 需要挂接某种共享存储，为了解决这个问题，Kubernetes 从 1.4 版本开始引入了 PetSet 这个新的资源对象，并且在 1.5 版本时更名为 StatefulSet，StatefulSet 从本质上来说，可以看作 Deployment/RC 的一个特殊变种，它有如下特性：

- StatefulSet 里的每个 Pod 都有稳定、唯一的网络标识，可以用来发现集群内的其他成员。假设 StatefulSet 的名称为 kafka，那么第 1 个 Pod 叫 kafka-0，第 2 个叫 kafka-1，以此类推。
- StatefulSet 控制的 Pod 副本的启停顺序是受控的，操作第 n 个 Pod 时，前 n-1 个 Pod 已经是运行且准备好的状态。
- StatefulSet 里的 Pod 采用稳定的持久化存储卷，通过 PV 或 PVC 来实现，删除 Pod 时默认不会删除与 StatefulSet 相关的存储卷（为了保证数据的安全）。

StatefulSet 除了要与 PV 卷捆绑使用以存储 Pod 的状态数据，还要与 Headless Service 配合使用，即在每个 StatefulSet 定义中都要声明它属于哪

个 Headless Service。Headless Service 与普通 Service 的关键区别在于，它没有 Cluster IP，如果解析 Headless Service 的 DNS 域名，则返回的是该 Service 对应的全部 Pod 的 Endpoint 列表。StatefulSet 在 Headless Service 的基础上又为 StatefulSet 控制的每个 Pod 实例都创建了一个 DNS 域名，这个域名的格式为：

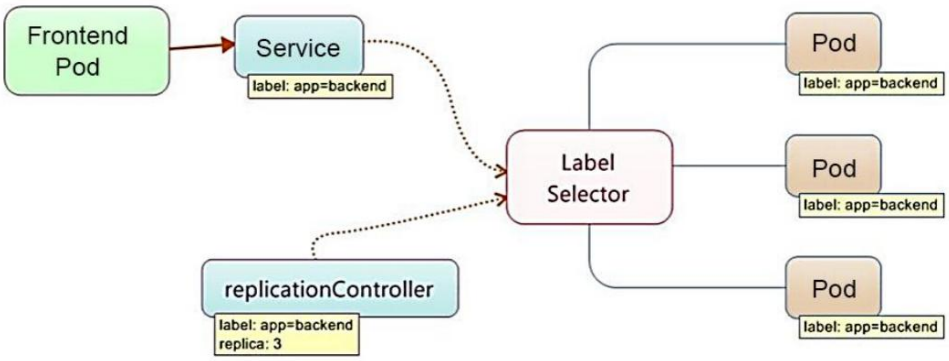
```
$(podname).$(headless service name)
```

比如一个 3 节点的 Kafka 的 StatefulSet 集群对应的 Headless Service 的名称为 kafka，StatefulSet 的名称为 kafka，则 StatefulSet 里的 3 个 Pod 的 DNS 名称分别为 kafka-0.kafka、kafka-1.kafka、kafka-3.kafka，这些 DNS 名称可以直接在集群的配置文件中固定下来。

6.8、Service

6.8.1、概述

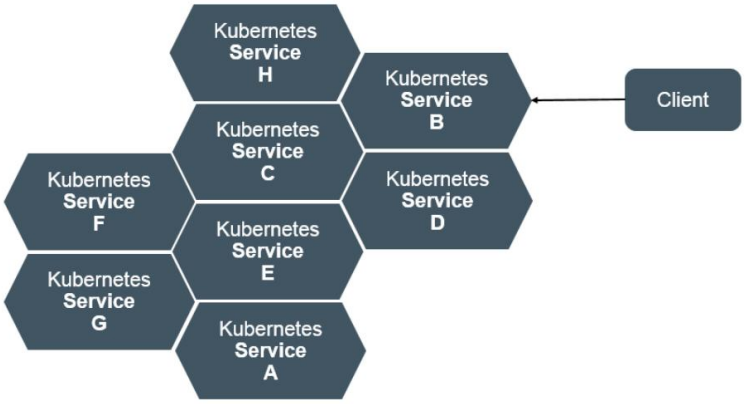
Service 服务也是 Kubernetes 里的核心资源对象之一，Kubernetes 里的每个 Service 其实就是微服务架构中的一个微服务，之前讲解 Pod、RC 等资源对象其实都是为讲解 Kubernetes Service 做铺垫的。下图显示了 Pod、RC 与 Service 的逻辑关系。



从上图中可以看到，Kubernetes 的 Service 定义了一个服务的访问入口地址，前端的应用（Pod）通过这个入口地址访问其背后的一组由 Pod 副本组成的集群实例，Service 与其后端 Pod 副本集群之间则是通过 Label Selector 来实现无缝对接的。RC 的作用实际上是保证 Service 的服务能力和服务质量始终符合预期标准。

通过分析、识别并建模系统中的所有服务为微服务—Kubernetes Service，我们的系统最终由多个提供不同业务能力而又彼此独立的微服务单元组成的，服务之间通过 TCP/IP 进行通信，从而形成了强大而又灵活的弹性网格，拥有强大的分布式能力、弹性扩展能力、容错能力，程序架构也变得简单和直观许多，如下图所示。

Kubernetes 提供的微服务网格架构图：



既然每个 Pod 都会被分配一个单独的 IP 地址，而且每个 Pod 都提供了一个独立的 Endpoint（Pod IP+ContainerPort）以被客户端访问，现在多个 Pod 副本组成了一个集群来提供服务，那么客户端如何来访问它们呢？一般的做法是部署一个负载均衡器（软件或硬件），为这组 Pod 开启一个对外的服务端口如 8000 端口，并且将这些 Pod 的 Endpoint 列表加入 8000 端口的转发列表，客户端就可以通过负载均衡器的对外 IP 地址+服务端口来访问此服务。客户端的请求最后会被转发到哪个 Pod，由负载均衡器的算法所决定。

Kubernetes 也遵循上述常规做法，运行在每个 Node 上的 kube-proxy 进程其实就是一个智能的软件负载均衡器，负责把对 Service 的请求转发到后端的某个 Pod 实例上，并在内部实现服务的负载均衡与会话保持机制。但 Kubernetes 发明了一种很巧妙又影响深远的设计：Service 没有共用一个负载均衡器的 IP 地址，每个 Service 都被分配了一个全局唯一的虚拟 IP 地址，这个虚拟 IP 被称为 Cluster IP。这样一来，每个服务就变成了具备唯一 IP 地址的通信节点，服务调用就变成了最基础的 TCP 网络通信问题。

我们知道，Pod 的 Endpoint 地址会随着 Pod 的销毁和重新创建而发生改变，因为新 Pod 的 IP 地址与之前旧 Pod 的不同。而 Service 一旦被创建，Kubernetes 就会自动为它分配一个可用的 Cluster IP，而且在 Service 的整个生命周期内，它的 Cluster IP 不会发生改变。于是，服务发现这个棘手的问题在 Kubernetes 的架构里也得以轻松解决：只要用 Service 的 Name 与 Service 的 Cluster IP 地址做一个 DNS 域名映射即可完美解决问题。现在想想，这真是一个很棒的设计。

说了这么久，下面动手创建一个 Service 来加深对它的理解。创建一个名为 tomcat-service.yaml 的定义文件，内容如下：

```
[root@master ~]# vi tomcat-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
  - port: 8080
  selector:
```

```
    tier: frontend
```

上述内容定义了一个名为tomcat-service的Service,它的服务端口为8080，拥有“tier = frontend”这个Label的所有Pod实例都属于它，运行下面的命令进行创建：

```
[root@master ~]# kubectl create -f tomcat-service.yaml
service/tomcat-service created
```

我们之前在tomcat-deployment.yaml里定义的Tomcat的Pod刚好拥有这个标签，所以刚才创建的tomcat-service已经对应一个Pod实例，运行下面的命令可以查看tomcatservice的Endpoint列表,其中10.44.0.1是Pod的IP地址，端口8080是Container暴露的端口：

```
[root@master ~]# kubectl get endpoints
NAME             ENDPOINTS             AGE
kubernetes       192.168.43.101:6443   2d17h
tomcat-service   10.44.0.1:8080        2m38s
```

你可能有疑问：“说好的Service的Cluster IP呢？怎么没有看到？”运行下面的命令即可看到tomct-service被分配的Cluster IP及更多的信息：

```
[root@master ~]# kubectl get svc tomcat-service -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2020-10-24T06:23:13Z"
  name: tomcat-service
  namespace: default
  resourceVersion: "18470"
  selfLink: /api/v1/namespaces/default/services/tomcat-service
  uid: eb989c22-94a6-4fe7-936c-5781e929b9ef
spec:
  clusterIP: 10.96.247.187
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    tier: frontend
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

在spec.ports的定义中，targetPort属性用来确定提供该服务的容器所暴露（EXPOSE）的端口号，即具体业务进程在容器内的targetPort上提供TCP/IP接入；port属性则定义了Service的虚端口。前面定义Tomcat服务时没有指定

targetPort，则默认 targetPort 与 port 相同。

接下来看看 Service 的多端口问题。

很多服务都存在多个端口的问题，通常一个端口提供业务服务，另外一个端口提供管理服务，比如 Mycat、Codis 等常见中间件。Kubernetes Service 支持多个 Endpoint，在存在多个 Endpoint 的情况下，要求每个 Endpoint 都定义一个名称来区分。下面是 Tomcat 多端口的 Service 定义样例：

```
[root@master ~]# vi tomcat-service-multiple-ports.yaml
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
  - port: 8080
    name: service-port
  - port: 8005
    name: shutdown-port
  selector:
    tier: frontend
```

多端口为什么需要给每个端口都命名呢？这就涉及 Kubernetes 的服务发现机制了，在下一节来给大家进行讲解。

6.8.2、Kubernetes 的服务发现机制

任何分布式系统都会涉及“服务发现”这个基础问题，大部分分布式系统都通过提供特定的 API 接口来实现服务发现功能，但这样做会导致平台的侵入性比较强，也增加了开发、测试的难度。Kubernetes 则采用了直观朴素的思路去解决这个棘手的问题。

首先，每个 Kubernetes 中的 Service 都有唯一的 Cluster IP 及唯一的名称，而名称是由开发者自己定义的，部署时也没必要改变，所以完全可以被固定在配置中。接下来的问题就是如何通过 Service 的名称找到对应的 Cluster IP。

最早时 Kubernetes 采用了 Linux 环境变量解决这个问题，即每个 Service 都生成一些对应的 Linux 环境变量（ENV），并在每个 Pod 的容器启动时自动注入这些环境变量。以下是 tomcat-service 产生的环境变量条目：

```
TOMCAT_SERVICE_SERVICE_HOST=169.169.41.218
TOMCAT_SERVICE_SERVICE_PORT_SERVICE_PORT=8080
TOMCAT_SERVICE_SERVICE_PORT_SHUTDOWN_PORT=8005
TOMCAT_SERVICE_SERVICE_PORT=8080
TOMCAT_SERVICE_PORT_8005_TCP_PORT=8005
TOMCAT_SERVICE_PORT=tcp://169.169.41.218:8080
TOMCAT_SERVICE_PORT_8080_TCP_ADDR=169.169.41.218
```

```
TOMCAT_SERVICE_PORT_8080_TCP=tcp://169.169.41.218:8080
TOMCAT_SERVICE_PORT_8080_TCP_PROTO=tcp
TOMCAT_SERVICE_PORT_8080_TCP_PORT=8080
TOMCAT_SERVICE_PORT_8005_TCP=tcp://169.169.41.218:8005
TOMCAT_SERVICE_PORT_8005_TCP_ADDR=169.169.41.218
TOMCAT_SERVICE_PORT_8005_TCP_PROTO=tcp
```

在上述环境变量中，比较重要的是前 3 个环境变量。可以看到，每个 Service 的 IP 地址及端口都有标准的命名规范，遵循这个命名规范，就可以通过代码访问系统环境变量来得到所需的信息，实现服务调用。

考虑到通过环境变量获取 Service 地址的方式仍然不太方便、 不够直观，后来 Kubernetes 通过 Add-On 增值包引入了 DNS 系统，把服务名作为 DNS 域名，这样程序就可以直接使用服务名来建立通信连接了。目前，Kubernetes 上的大部分应用都已经采用了 DNS 这种新兴的服务发现机制，后面会讲解如何部署 DNS 系统。

6.8.3、外部系统访问 Service 的问题

为了更深入地理解和掌握 Kubernetes，我们需要弄明白 Kubernetes 里的 3 种 IP，这 3 种 IP 分别如下。

- Node IP：Node 的 IP 地址。
- Pod IP：Pod 的 IP 地址。
- Cluster IP：Service 的 IP 地址。

首先，Node IP 是 Kubernetes 集群中每个节点的物理网卡的 IP 地址，是一个真实存在的物理网络，所有属于这个网络的服务器都能通过这个网络直接通信，不管其中是否有部分节点不属于这个 Kubernetes 集群。这也表明在 Kubernetes 集群之外的节点访问 Kubernetes 集群之内的某个节点或者 TCP/IP 服务时，都必须通过 Node IP 通信。

其次，Pod IP 是每个 Pod 的 IP 地址，它是 Docker Engine 根据 docker0 网桥的 IP 地址段进行分配的，通常是一个虚拟的二层网络，前面说过，Kubernetes 要求位于不同 Node 上的 Pod 都能够彼此直接通信，所以 Kubernetes 里一个 Pod 里的容器访问另外一个 Pod 里的容器时，就是通过 Pod IP 所在的虚拟二层网络进行通信的，而真实的 TCP/IP 流量是通过 Node IP 所在的物理网卡流出的。

最后说说 Service 的 Cluster IP，它也是一种虚拟的 IP，但更像一个“伪造”的 IP 网络，原因有以下几点。

- Cluster IP 仅仅作用于 Kubernetes Service 这个对象，并由 Kubernetes 管理和分配 IP 地址（来源于 Cluster IP 地址池）。
- Cluster IP 无法被 Ping，因为没有一个“实体网络对象”来响应。
- Cluster IP 只能结合 Service Port 组成一个具体的通信端口，单独的 Cluster IP 不具备 TCP/IP 通信的基础，并且它们属于 Kubernetes 集群这样一

个封闭的空间，集群外的节点如果要访问这个通信端口，则需要做一些额外的工作。

● 在 Kubernetes 集群内，Node IP 网、Pod IP 网与 Cluster IP 网之间的通信，采用的是 Kubernetes 自己设计的一种编程方式的特殊路由规则，与我们熟知的 IP 路由有很大的不同。

根据上面的分析和总结，我们基本明白了：Service 的 Cluster IP 属于 Kubernetes 集群内部的地址，无法在集群外部直接使用这个地址。那么矛盾来了：实际上在我们开发的业务系统中肯定多少有一部分服务是要提供给 Kubernetes 集群外部的应用或者用户来使用的，典型的例子就是 Web 端的服务模块，比如上面的 tomcat-service，那么用户怎么访问它？

采用 NodePort 是解决上述问题的最直接、有效的常见做法。以 tomcat-service 为例，在 Service 的定义里做如下扩展即可（见代码中的粗体部分）：

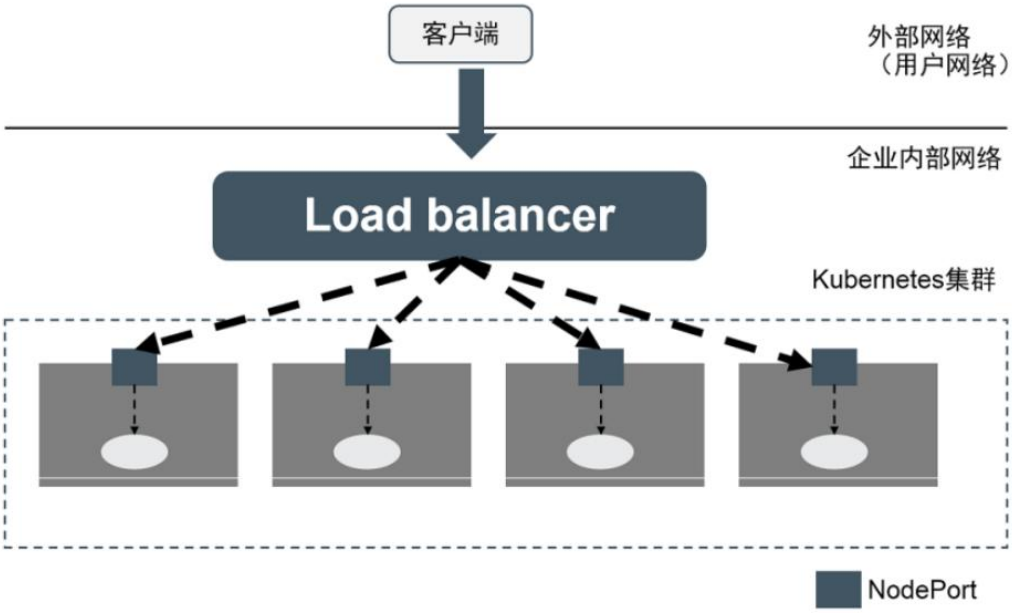
```
[root@master ~]# vi tomcat-service-nodeport.yaml
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
 type: NodePort
 ports:
  - port: 8080
    nodePort: 31002
 selector:
    tier: frontend
```

其中，nodePort:31002 这个属性表明手动指定 tomcat-service 的 NodePort 为 31002，否则 Kubernetes 会自动分配一个可用的端口。接下来在浏览器里访问 http://<nodePort IP>:31002/，就可以看到 Tomcat 的欢迎界面了。

NodePort 的实现方式是在 Kubernetes 集群里的每个 Node 上都为需要外部访问的 Service 开启一个对应的 TCP 监听端口，外部系统只要用任意一个 Node 的 IP 地址+具体的 NodePort 端口号即可访问此服务，在任意 Node 上运行 netstat 命令，就可以看到有 NodePort 端口被监听：

```
[root@node-1 ~]# netstat -tlunp|grep 31002
tcp    0   0 0.0.0.0:31002   0.0.0.0:*   LISTEN      10528/kube-proxy
```

但 NodePort 还没有完全解决外部访问 Service 的所有问题，比如负载均衡问题。假如在我们的集群中有 10 个 Node，则此时最好有一个负载均衡器，外部的请求只需访问此负载均衡器的 IP 地址，由负载均衡器负责转发流量到后面某个 Node 的 NodePort 上，如下图所示。



上图中的 Load balancer 组件独立于 Kubernetes 集群之外，通常是一个硬件的负载均衡器，或者是以软件方式实现的，例如 HAProxy 或者 Nginx。对于每个 Service，我们通常需要配置一个对应的 Load balancer 实例来转发流量到后端的 Node 上，这的确增加了工作量及出错的概率。于是 Kubernetes 提供了自动化的解决方案，如果我们的集群运行在谷歌的公有云 GCE 上，那么只要把 Service 的 type=NodePort 改为 type=LoadBalancer，Kubernetes 就会自动创建一个对应的 Load balancer 实例并返回它的 IP 地址供外部客户端使用。其他公有云提供商只要实现了支持此特性的驱动，则也可以达到上述目的。此外，裸机上的类似机制（Bare Metal Service Load Balancers）也在被开发。

6.9、Job

批处理任务通常并行（或者串行）启动多个计算进程去处理一批工作项（work item），在处理完成后，整个批处理任务结束。从 1.2 版本开始，Kubernetes 支持批处理类型的应用，我们可以通过 Kubernetes Job 这种新的资源对象定义并启动一个批处理任务 Job。与 RC、Deployment、ReplicaSet、DaemonSet 类似，Job 也控制一组 Pod 容器。从这个角度来看，Job 也是一种特殊的 Pod 副本自动控制器，同时 Job 控制 Pod 副本与 RC 等控制器的工作机制有以下重要差别。

（1）Job 所控制的 Pod 副本是短暂运行的，可以将其视为一组 Docker 容器，其中的每个 Docker 容器都仅仅运行一次。当 Job 控制的所有 Pod 副本都运行结束时，对应的 Job 也就结束了。Job 在实现方式上与 RC 等副本控制器不同，Job 生成的 Pod 副本是不能自动重启的，对应 Pod 副本的 RestartPoliy 都被设置为 Never。因此，当对应的 Pod 副本都执行完成时，相应的 Job 也就完成了控制使命，即 Job 生成的 Pod 在 Kubernetes 中是短暂存在的。Kubernetes 在 1.5 版本之后又提供了类似 crontab 的定时任务——CronJob，解决了某些批处理任务需要定时反复执行的问题。

（2）Job 所控制的 Pod 副本的工作模式能够多实例并行计算，以 TensorFlow 框架为例，可以将一个机器学习的计算任务分布到 10 台机器上，在每台机器上都运行一个 worker 执行计算任务，这很适合通过 Job 生成 10 个 Pod 副本同时启动运算。

6.10、Volume

Volume（存储卷）是 Pod 中能够被多个容器访问的共享目录。Kubernetes 的 Volume 概念、用途和目的与 Docker 的 Volume 比较类似，但两者不能等价。首先，Kubernetes 中的 Volume 被定义在 Pod 上，然后被一个 Pod 里的多个容器挂载到具体的文件目录下；其次，Kubernetes 中的 Volume 与 Pod 的生命周期相同，但与容器的生命周期不相关，当容器终止或者重启时，Volume 中的数据也不会丢失。最后，Kubernetes 支持多种类型的 Volume，例如 GlusterFS、Ceph 等先进的分布式文件系统。

Volume 的使用也比较简单，在大多数情况下，我们先在 Pod 上声明一个 Volume，然后在容器里引用该 Volume 并挂载（Mount）到容器里的某个目录上。举例来说，我们要给之前的 Tomcat Pod 增加一个名为 datavol 的 Volume，并且挂载到容器的/mydata-data 目录上，则只要对 Pod 的定义文件做如下修正即可（注意代码中的粗体部分）：

```
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      volumes:
        - name: datavol
          emptyDir: {}
      containers:
      - name: tomcat-demo
        image: tomcat
        volumeMounts:
          - mountPath: /mydata-data
          name: datavol
          imagePullPolicy: IfNotPresent
```

除了可以让一个 Pod 里的多个容器共享文件、让容器的数据写到宿主机的磁盘上或者写文件到网络存储中，Kubernetes 的 Volume 还扩展出了一种非常有实用价值的功能，即容器配置文件集中化定义与管理，这是通过 ConfigMap 这种新的资源对象来实现的，后面会详细说明。

Kubernetes 提供了非常丰富的 Volume 类型，下面逐一进行说明。

- 1、emptyDir

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。

一个 emptyDir Volume 是在 Pod 分配到 Node 时创建的。从它的名称就可以看出，它的初始内容为空，并且无须指定宿主机上对应的目录文件，因为这是 Kubernetes 自动分配的一个目录，当 Pod 从 Node 上移除时，emptyDir 中的数据也会被永久删除。

emptyDir 的一些用途如下：

○ 临时空间，例如用于某些应用程序运行时所需的临时目录，且无须永久保留。

○ 长时间任务的中间过程 CheckPoint 的临时保存目录。

○ 一个容器需要从另一个容器中获取数据的目录（多容器共享目录）。

目前，用户无法控制 emptyDir 使用的介质种类。如果 kubelet 的配置是使用硬盘，那么所有 emptyDir 都将被创建在该硬盘上。Pod 在将来可以设置 emptyDir 是位于硬盘、固态硬盘上还是基于内存的 tmpfs 上，上面的例子便采用了 emptyDir 类的 Volume。

● 2、hostPath

hostPath 为在 Pod 上挂载宿主机上的文件或目录，它通常可以用于以下几方面：

○ 容器应用程序生成的日志文件需要永久保存时，可以使用宿主机的高速文件系统进行存储。

○ 需要访问宿主机上 Docker 引擎内部数据结构的容器应用时，可以通过定义 hostPath 为宿主机/var/lib/docker 目录，使容器内部应用可以直接访问 Docker 的文件系统。

在使用这种类型的 Volume 时，需要注意以下几点：

○ 在不同的 Node 上具有相同配置的 Pod，可能会因为宿主机上的目录和文件不同而导致对 Volume 上目录和文件的访问结果不一致。

○ 如果使用了资源配额管理，则 Kubernetes 无法将 hostPath 在宿主机上使用的资源纳入管理。

在下面的例子中使用宿主机的/data 目录定义了一个 hostPath 类型的 Volume：

```
volumes:
- name: "persistent-storage"
  hostPath:
    path: "/data"
```

● 3、gcePersistenDisk

使用这种类型的 Volume 表示使用谷歌公有云提供的永久磁盘（Persistent Disk，PD）存放 Volume 的数据，它与 emptyDir 不同，PD 上的内容会被永久保存，当 Pod 被删除时，PD 只是被卸载（Unmount），但不会被删除。需要注意的是，你需要先创建一个 PD，才能使用 gcePersistentDisk。

使用 gcePersistentDisk 时有以下一些限制条件：

版权声明，本文档全部内容及版权归“张岩峰”老师所有，只可用于自己学习使用，禁止私自传阅，违者依法追责。

○ Node（运行 kubelet 的节点）需要是 GCE 虚拟机。
○ 这些虚拟机需要与 PD 存在于相同的 GCE 项目和 Zone 中。

通过 gcloud 命令即可创建一个 PD：

```
# gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

定义 gcePersistentDisk 类型的 Volume 的示例如下：

```
volumes:
- name: test-volume
  # This GCE PD must already exist.
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

● 4、awsElasticBlockStore

与 GCE 类似，该类型的 Volume 使用亚马逊公有云提供的 EBS Volume 存储数据，需要先创建一个 EBS Volume 才能使用 awsElasticBlockStore。

使用 awsElasticBlockStore 的一些限制条件如下：

○ Node（运行 kubelet 的节点）需要是 AWS EC2 实例。
○ 这些 AWS EC2 实例需要与 EBS Volume 存在于相同的 region 和 availability-zone 中。
○ EBS 只支持单个 EC2 实例挂载一个 Volume。

通过 aws ec2 create-volume 命令可以创建一个 EBS Volume：

定义 awsElasticBlockStore 类型的 Volume 的示例如下：

```
volumes:
- name: test-volume
  # This AWS EBS volume must already exist.
  awsElasticBlockStore:
    volumeID: aws://<availability-zone>/<volume-id>
    fsType: ext4
```

● 5、NFS

使用 NFS 网络文件系统提供的共享目录存储数据时，我们需要在系统中部署一个 NFS Server。定义 NFS 类型的 Volume 的示例如下：

```
volumes:
- name: nfs
  nfs:
    # 改为你的 NFS 服务器地址
    server: nfs-server.localhost
    path: "/"
```

- 6、其他类型的 Volume
  - iscsi：使用 iSCSI 存储设备上的目录挂载到 Pod 中。
  - flocker：使用 Flocker 管理存储卷。
  - glusterfs：使用开源 GlusterFS 网络文件系统的目录挂载到 Pod 中。
  - rbd：使用 Ceph 块设备共享存储（Rados Block Device）挂载到 Pod 中。
  - gitRepo:通过挂载一个空目录,并从 Git 库 clone 一个 git repository 以供 Pod 使用。
  - secret：一个 Secret Volume 用于为 Pod 提供加密的信息，你可以将定义在 Kubernetes 中的 Secret 直接挂载为文件让 Pod 访问。Secret Volume 是通过 TMFS（内存文件系统）实现的，这种类型的 Volume 总是不会被持久化的。

6.11、Persistent Volume

之前提到的 Volume 是被定义在 Pod 上的，属于计算资源的一部分，而实际上，网络存储是相对独立于计算资源而存在的一种实体资源。比如在使用虚拟机的情况下，我们通常会先定义一个网络存储，然后从中划出一个“网盘”并挂接到虚拟机上。Persistent Volume（PV）和与之相关联的 Persistent Volume Claim（PVC）也起到了类似的作用。

PV 可以被理解成 Kubernetes 集群中的某个网络存储对应的一块存储，它与 Volume 类似，但有以下区别：

- PV 只能是网络存储，不属于任何 Node，但可以在每个 Node 上访问。
- PV 并不是被定义在 Pod 上的，而是独立于 Pod 之外定义的。
- PV 目前支持的类型包括：gcePersistentDisk、AWSElasticBlockStore、AzureFile、AzureDisk、FC（Fibre Channel）、Flocker、NFS、iSCSI、RBD（Rados Block Device）、CephFS、Cinder、GlusterFS、VsphereVolume、Quobyte Volumes、VMware Photon、Portworx Volumes、ScaleIO Volumes 和 HostPath（仅供单机测试）

下面给出了 NFS 类型的 PV 的一个 YAML 定义文件，声明了需要 5Gi 的存储空间：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /somepath
```

```
        server: 172.17.0.2
```

比较重要的是 PV 的 accessModes 属性，目前有以下类型。

- ReadWriteOnce：读写权限，并且只能被单个 Node 挂载。
- ReadOnlyMany：只读权限，允许被多个 Node 挂载。
- ReadWriteMany：读写权限，允许被多个 Node 挂载。

如果某个 Pod 想申请某种类型的 PV，则首先需要定义一个 PersistentVolumeClaim 对象：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

然后，在 Pod 的 Volume 定义中引用上述 PVC 即可：

```
volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim
```

最后说说 PV 的状态。PV 是有状态的对象，它的状态有以下几种：

- Available：空闲状态。
- Bound：已经绑定到某个 PVC 上。
- Released：对应的 PVC 已经被删除，但资源还没有被集群收回。
- Failed：PV 自动回收失败。

6.12、Namespace

Namespace（命名空间）是 Kubernetes 系统中的另一个非常重要的概念，Namespace 在很多情况下用于实现多租户的资源隔离。Namespace 通过将集群内部的资源对象“分配”到不同的 Namespace 中，形成逻辑上分组的不同项目、小组或用户组，便于不同的分组在共享使用整个集群的资源的同时还能被分别管理。

Kubernetes 集群在启动后会创建一个名为 default 的 Namespace，通过 kubectl 可以查看：

```
[root@master ~]# kubectl get namespaces
NAME              STATUS   AGE
default           Active   4d22h
```

接下来，如果不特别指明 Namespace，则用户创建的 Pod、RC、Service 都将被系统创建到这个默认的名为 default 的 Namespace 中。

```
#创建 yaml 文件
    [root@master ~]# vi namespace-1.yaml
    apiVersion: v1
    kind: Namespace
    metadata:
      name: development
#创建：
    [root@master ~]# kubectl create -f namespace-1.yaml
    namespace/development created
#查询：
    [root@master ~]# kubectl get namespaces
    NAME              STATUS   AGE
    development       Active   12s
```

一旦创建了 Namespace，我们在创建资源对象时就可以指定这个资源对象属于哪个 Namespace。比如在下面的例子中定义了一个名为 busybox 的 Pod，并将其放入 development 这个 Namespace 里：

```
#创建 yaml 文件
    [root@master ~]# vi namespace-2.yaml
    apiVersion: v1
    kind: Pod
    metadata:
      name: busybox
      namespace: development
    spec:
      containers:
      - image: busybox
        command:
          - sleep
          - "3600"
        name: busybox
#创建：
    [root@master ~]# kubectl create -f namespace-2.yaml
    pod/busybox created
```

此时使用 kubectl get 命令查看，将无法显示：

```
    [root@master ~]# kubectl get pods
    No resources found in default namespace.
```

这是因为如果不加参数，则 kubectl get 命令将仅显示属于 default 命名空

间的资源对象。

可以在 kubectl 命令中加入--namespace 参数来查看某个命名空间中的对象：

```
[root@master ~]# kubectl get pods --namespace=development
NAME      READY   STATUS    RESTARTS   AGE
busybox   1/1     Running   0          23s
```

当给每个租户创建一个 Namespace 来实现多租户的资源隔离时，还能结合 Kubernetes 的资源配额管理，限定不同租户能占用的资源，例如 CPU 使用量、内存使用量等。关于资源配额管理的问题，在后面的章节中会详细介绍。

6.13、Annotation

Annotation（注解）与 Label 类似，也使用 key/value 键值对的形式进行定义。不同的是 Label 具有严格的命名规则，它定义的是 Kubernetes 对象的元数据（Metadata），并且用于 Label Selector。Annotation 则是用户任意定义的附加信息，以便于外部工具查找。在很多时候，Kubernetes 的模块自身会通过 Annotation 标记资源对象的一些特殊信息。

通常来说，用 Annotation 来记录的信息如下所示：

- build 信息、release 信息、Docker 镜像信息等，例如时间戳、release id 号、PR 号、镜像 Hash 值、Docker Registry 地址等。
- 日志库、监控库、分析库等资源库的地址信息。
- 程序调试工具信息，例如工具名称、版本号等。
- 团队的联系信息，例如电话号码、负责人名称、网址等。

6.14、ConfigMap

为了能够准确和深刻理解 Kubernetes ConfigMap 的功能和价值，我们需要从 Docker 说起。我们知道，Docker 通过将程序、依赖库、数据及配置文件“打包固化”到一个不变的镜像文件中的做法，解决了应用的部署的难题，但这同时带来了棘手的问题，即配置文件中的参数在运行期如何修改的问题。我们不可能在启动 Docker 容器后再修改容器里的配置文件，然后用新的配置文件重启容器里的用户主进程。为了解决这个问题，Docker 提供了两种方式：

- 在运行时通过容器的环境变量来传递参数；
- 通过 Docker Volume 将容器外的配置文件映射到容器内。

这两种方式都有其优势和缺点，在大多数情况下，后一种方式更合适我们的系统，因为大多数应用通常从一个或多个配置文件中读取参数。但这种方式也有明显的缺陷：我们必须在目标主机上先创建好对应的配置文件，然后才能映射到容器里。

上述缺陷在分布式情况下变得更为严重，因为无论采用哪种方式，写入（修改）多台服务器上的某个指定文件，并确保这些文件保持一致，都是一个很难完

成的目标。此外，在大多数情况下，我们都希望能集中管理系统的配置参数，而不是管理一堆配置文件。针对上述问题，Kubernetes 给出了一个很巧妙的设计实现，如下所述。

首先，把所有的配置项都当作 key-value 字符串，当然 value 可以来自某个文本文件，比如配置项 password=123456、user=root、host=192.168.8.4 用于表示连接 FTP 服务器的配置参数。这些配置项可以作为 Map 表中的一个项，整个 Map 的数据可以被持久化存储在 Kubernetes 的 Etcd 数据库中，然后提供 API 以方便 Kubernetes 相关组件或客户应用 CRUD 操作这些数据，上述专门用来保存配置参数的 Map 就是 Kubernetes ConfigMap 资源对象。

接下来，Kubernetes 提供了一种内建机制，将存储在 etcd 中的 ConfigMap 通过 Volume 映射的方式变成目标 Pod 内的配置文件，不管目标 Pod 调度到哪台服务器上，都会完成自动映射。进一步地，如果 ConfigMap 中的 key-value 数据被修改，则映射到 Pod 中的“配置文件”也会随之自动更新。于是，Kubernetes ConfigMap 就成了分布式系统中最为简单（使用方法简单，但背后实现比较复杂）且对应用无侵入的配置中心。ConfigMap 配置集中化的一种简单方案如下图所示：