

第 1 章

Kubernetes 入门

1.1 Kubernetes 是什么

Kubernetes 是什么？

首先，它是一个全新的基于容器技术的分布式架构领先方案。这个方案虽然还很新，但它是谷歌十几年以来大规模应用容器技术的经验积累和升华的一个重要成果。确切地说，Kubernetes 是谷歌严格保密十几年的秘密武器——Borg 的一个开源版本。Borg 是谷歌的一个久负盛名的内部使用的大规模集群管理系统，它基于容器技术，目的是实现资源管理的自动化，以及跨多个数据中心的资源利用率的最大化。十几年来，谷歌一直通过 Borg 系统管理着数量庞大的应用程序集群。由于谷歌员工都签署了保密协议，即便离职也不能泄露 Borg 的内部设计，所以外界一直无法了解关于它的更多信息。直到 2015 年 4 月，传闻许久的 Borg 论文伴随 Kubernetes 的高调宣传被谷歌首次公开，大家才得以了解它的更多内幕。正是由于站在 Borg 这个前辈的肩膀上，吸取了 Borg 过去十年间的经验与教训，所以 Kubernetes 一经开源就一鸣惊人，并迅速称霸了容器技术领域。

其次，如果我们的系统设计遵循了 Kubernetes 的设计思想，那么传统系统架构中那些和业务没有多大关系的底层代码或功能模块，都可以立刻从我们的视线中消失，我们不必再费心于负载均衡器的选型和部署实施问题，不必再考虑引入或自己开发一个复杂的服务治理框架，不必再头疼于服务监控和故障处理模块的开发。总之，使用 Kubernetes 提供的解决方案，我们不仅节省了不少于 30% 的开发成本，同时可以将精力更加集中于业务本身，而且由于 Kubernetes 提供了强大的自动化机制，所以系统后期的运维难度和运维成本大幅度降低。

然后，Kubernetes 是一个开放的开发平台。与 J2EE 不同，它不局限于任何一种语言，没有

限定任何编程接口，所以不论是用 Java、Go、C++还是用 Python 编写的服务，都可以毫无困难地映射为 Kubernetes 的 Service，并通过标准的 TCP 通信协议进行交互。此外，由于 Kubernetes 平台对现有的编程语言、编程框架、中间件没有任何侵入性，因此现有的系统也很容易改造升级并迁移到 Kubernetes 平台上。

最后，Kubernetes 是一个完备的分布式系统支撑平台。Kubernetes 具有完备的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建智能负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制，以及多粒度的资源配额管理能力。同时，Kubernetes 提供了完善的管理工具，这些工具涵盖了包括开发、部署测试、运维监控在内的各个环节。因此，Kubernetes 是一个全新的基于容器技术的分布式架构解决方案，并且是一个一站式的完备的分布式系统开发和支撑平台。

在正式开始本章的 Hello World 之旅之前，我们首先要学习 Kubernetes 的一些基本知识，这样我们才能理解 Kubernetes 提供的解决方案。

在 Kubernetes 中，Service（服务）是分布式集群架构的核心，一个 Service 对象拥有如下关键特征。

- ☉ 拥有一个唯一指定的名字（比如 mysql-server）。
- ☉ 拥有一个虚拟 IP（Cluster IP、Service IP 或 VIP）和端口号。
- ☉ 能够提供某种远程服务能力。
- ☉ 被映射到了提供这种服务能力的一组容器应用上。

Service 的服务进程目前都基于 Socket 通信方式对外提供服务，比如 Redis、Memcache、MySQL、Web Server，或者是实现了某个具体业务的一个特定的 TCP Server 进程。虽然一个 Service 通常由多个相关的服务进程来提供服务，每个服务进程都有一个独立的 Endpoint（IP+Port）访问点，但 Kubernetes 能够让我们通过 Service（虚拟 Cluster IP +Service Port）连接到指定的 Service 上。有了 Kubernetes 内建的透明负载均衡和故障恢复机制，不管后端有多少服务进程，也不管某个服务进程是否会由于发生故障而重新部署到其他机器，都不会影响到我们对服务的正常调用。更重要的是这个 Service 本身一旦创建就不再变化，这意味着，在 Kubernetes 集群中，我们再也不用为了服务的 IP 地址变来变去的问题而头疼了。

容器提供了强大的隔离功能，所以有必要把为 Service 提供服务的这组进程放入容器中进行隔离。为此，Kubernetes 设计了 Pod 对象，将每个服务进程包装到相应的 Pod 中，使其成为 Pod 中运行的一个容器（Container）。为了建立 Service 和 Pod 间的关联关系，Kubernetes 首先给每个 Pod 贴上一个标签（Label），给运行 MySQL 的 Pod 贴上 name=mysql 标签，给运行 PHP 的

Pod 贴上 `name=php` 标签, 然后给相应的 Service 定义标签选择器 (Label Selector), 比如 MySQL Service 的标签选择器的选择条件为 `name=mysql`, 意为该 Service 要作用于所有包含 `name=mysql` Label 的 Pod 上。这样一来, 就巧妙地解决了 Service 与 Pod 的关联问题。

说到 Pod, 我们这里先简单介绍其概念。首先, Pod 运行在一个我们称之为节点 (Node) 的环境中, 这个节点既可以是物理机, 也可以是私有云或者公有云中的一个虚拟机, 通常在一个节点上运行几百个 Pod; 其次, 每个 Pod 里运行着一个特殊的被称之为 Pause 的容器, 其他容器则为业务容器, 这些业务容器共享 Pause 容器的网络栈和 Volume 挂载卷, 因此它们之间的通信和数据交换更为高效, 在设计时我们可以充分利用这一特性将一组密切相关的服务进程放入同一个 Pod 中; 最后, 需要注意的是, 并不是每个 Pod 和它里面运行的容器都能“映射”到一个 Service 上, 只有那些提供服务 (无论是对内还是对外) 的一组 Pod 才会被“映射”成一个服务。

在集群管理方面, Kubernetes 将集群中的机器划分为一个 Master 节点和一群工作节点 (Node)。其中, 在 Master 节点上运行着集群管理相关的一组进程 `kube-apiserver`、`kube-controller-manager` 和 `kube-scheduler`, 这些进程实现了整个集群的资源管理、Pod 调度、弹性伸缩、安全控制、系统监控和纠错等管理功能, 并且都是全自动完成的。Node 作为集群中的工作节点, 运行真正的应用程序, 在 Node 上 Kubernetes 管理的最小运行单元是 Pod。Node 上运行着 Kubernetes 的 `kubelet`、`kube-proxy` 服务进程, 这些服务进程负责 Pod 的创建、启动、监控、重启、销毁, 以及实现软件模式的负载均衡器。

最后, 我们再来看看传统的 IT 系统中服务扩容和服务升级这两个难题, 以及 Kubernetes 所提供的全新解决思路。服务的扩容涉及资源分配 (选择哪个节点进行扩容)、实例部署和启动等环节, 在一个复杂的业务系统中, 这两个问题基本上靠人工一步步操作才得以完成, 费时费力又难以保证实施质量。

在 Kubernetes 集群中, 你只需为需要扩容的 Service 关联的 Pod 创建一个 Replication Controller (简称 RC), 则该 Service 的扩容以至于后来的 Service 升级等头疼问题都迎刃而解。在一个 RC 定义文件中包括以下 3 个关键信息。

- ☉ 目标 Pod 的定义。
- ☉ 目标 Pod 需要运行的副本数量 (Replicas)。
- ☉ 要监控的目标 Pod 的标签 (Label)。

在创建好 RC (系统将自动创建好 Pod) 后, Kubernetes 会通过 RC 中定义的 Label 筛选出对应的 Pod 实例并实时监控其状态和数量, 如果实例数量少于定义的副本数量 (Replicas), 则会根据 RC 中定义的 Pod 模板来创建一个新的 Pod, 然后将此 Pod 调度到合适的 Node 上启动运行, 直到 Pod 实例的数量达到预定目标。这个过程完全是自动化的, 无须人工干预。有了 RC,

服务的扩容就变成了一个纯粹的简单数字游戏了，只要修改 RC 中的副本数量即可。后续的 Service 升级也将通过修改 RC 来自动完成。

以将在第 2 章介绍的 PHP+Redis 留言板应用为例，只要为 PHP 留言板程序（frontend）创建一个有 3 个副本的 RC+Service，为 Redis 读写分离集群创建两个 RC：写节点（redis-master）创建一个单副本的 RC+Service，读节点（redis-slaver）创建一个有两个副本的 RC+Service，就可以分分钟完成整个集群的搭建过程了，是不是很简单？

1.2 为什么要用 Kubernetes

使用 Kubernetes 的理由很多，最根本的一个理由就是：IT 从来都是一个由新技术驱动的行业。

Docker 这个新兴的容器化技术当前已经被很多公司所采用，其从单机走向集群已成为必然，而云计算的蓬勃发展正在加速这一进程。Kubernetes 作为当前唯一被业界广泛认可和看好的 Docker 分布式系统解决方案，可以预见，在未来几年内，会有大量的新系统选择它，不管这些系统是运行在企业本地服务器上还是被托管到公有云上。

使用了 Kubernetes 又会收获哪些好处呢？

首先，最直接的感受就是我们可以“轻装上阵”地开发复杂系统了。以前动不动就需要十几个人而且团队里需要不少技术达人一起分工协作才能设计实现和运维的分布式系统，在采用 Kubernetes 解决方案之后，只需一个精悍的小团队就能轻松应对。在这个团队里，一名架构师专注于系统中“服务组件”的提炼，几名开发工程师专注于业务代码的开发，一名系统兼运维工程师负责 Kubernetes 的部署和运维，从此再也不用“996”了，这并不是因为我们少做了什么，而是因为 Kubernetes 已经帮我们做了很多。

其次，使用 Kubernetes 就是在全面拥抱微服务架构。微服务架构的核心是将一个巨大的单体应用分解为很多小的互相连接的微服务，一个微服务背后可能有多个实例副本在支撑，副本的数量可能会随着系统的负荷变化而进行调整，内嵌的负载均衡器在这里发挥了重要作用。微服务架构使得每个服务都可以由专门的开发团队来开发，开发者可以自由选择开发技术，这对于大规模团队来说很有价值，另外每个微服务独立开发、升级、扩展，因此系统具备很高的稳定性和快速迭代进化能力。谷歌、亚马逊、eBay、NetFlix 等众多大型互联网公司都采用了微服务架构，此次谷歌更是将微服务架构的基础设施直接打包到 Kubernetes 解决方案中，让我们有机会直接应用微服务架构解决复杂业务系统的架构问题。

然后，我们的系统可以随时随地整体“搬迁”到公有云上。Kubernetes 最初的目标就是运

行在谷歌自家的公有云 GCE 中，未来会支持更多的公有云及基于 OpenStack 的私有云。同时，在 Kubernetes 的架构方案中，底层网络的细节完全被屏蔽，基于服务的 Cluster IP 甚至都无须我们改变运行期的配置文件，就能将系统从物理机环境中无缝迁移到公有云中，或者在服务高峰期将部分服务对应的 Pod 副本放入公有云中以提升系统的吞吐量，不仅节省了公司的硬件投入，还大大改善了客户体验。我们所熟知的铁道部的 12306 购票系统，在春节高峰期就租用了阿里云进行分流。

最后，Kubernetes 系统架构具备了超强的横向扩容能力。对于互联网公司来说，用户规模就等价于资产，谁拥有更多的用户，谁就能在竞争中胜出，因此超强的横向扩容能力是互联网业务系统的关键指标之一。不用修改代码，一个 Kubernetes 集群即可从只包含几个 Node 的小集群平滑扩展到拥有上百个 Node 的大规模集群，我们利用 Kubernetes 提供的工具，甚至可以在线完成集群扩容。只要我们的微服务设计得好，结合硬件或者公有云资源的线性增加，系统就能够承受大量用户并发访问所带来的巨大压力。

1.3 从一个简单的例子开始

考虑到本书第 1 版中的 PHP+Redis 留言板的 Hello World 例子对于绝大多数刚接触 Kubernetes 的人来说比较复杂，难以顺利上手和实践，所以我们在此将这个例子替换成一个简单得多的 Java Web 应用，可以让新手快速上手和实践。

此 Java Web 应用的结构比较简单，是一个运行在 Tomcat 里的 Web App，如图 1.1 所示，JSP 页面通过 JDBC 直接访问 MySQL 数据库并展示数据。为了演示和简化的目的，只要程序正确连接到了数据库上，它就会自动完成对应的 Table 的创建与初始化数据的准备工作。所以，当我们通过浏览器访问此应用的时候，就会显示一个表格的页面，数据则来自数据库。

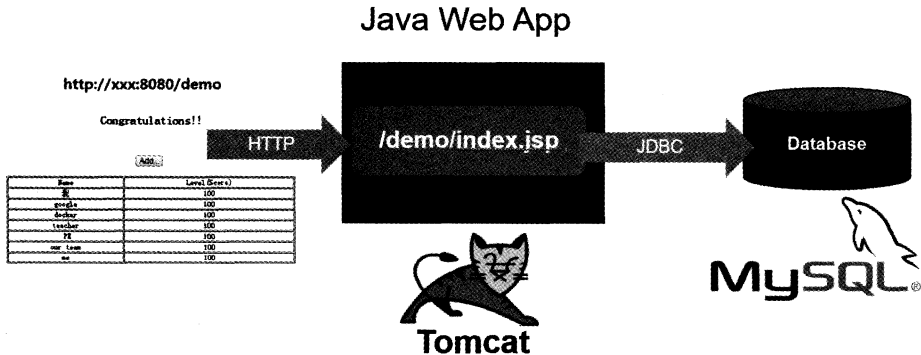


图 1.1 Java Web 应用的架构组成

此应用需要启动两个容器：Web App 容器和 MySQL 容器，并且 Web App 容器需要访问 MySQL 容器。在 Docker 时代，假设我们在一个宿主机上启动了这两个容器，则我们需要把 MySQL 容器的 IP 地址通过环境变量的方式注入 Web App 容器里；同时，需要将 Web App 容器的 8080 端口映射到宿主机的 8080 端口，以便能在外部访问。在本章的这个例子里，我们看看在 Kubernetes 时代是如何完成这个目标的。

1.3.1 环境准备

首先，我们开始准备 Kubernetes 的安装和相关镜像下载，本书建议采用 VirtualBox 或者 VMware Workstation 在本机虚拟一个 64 位的 CentOS 7 虚拟机作为学习环境，虚拟机采用 NAT 的网络模式以便能够连接外网，然后按照以下步骤快速安装 Kubernetes。

(1) 关闭 CentOS 自带的防火墙服务：

```
# systemctl disable firewalld
# systemctl stop firewalld
```

(2) 安装 etcd 和 Kubernetes 软件（会自动安装 Docker 软件）：

```
# yum install -y etcd kubernetes
```

(3) 安装好软件后，修改两个配置文件（其他配置文件使用系统默认的配置参数即可）。

☉ Docker 配置文件为/etc/sysconfig/docker，其中 OPTIONS 的内容设置为：

```
OPTIONS='--selinux-enabled=false --insecure-registry gcr.io'
```

☉ Kubernetes apiserver 配置文件为/etc/kubernetes/apiserver，把--admission_control 参数中的 ServiceAccount 删除。

(4) 按顺序启动所有的服务：

```
# systemctl start etcd
# systemctl start docker
# systemctl start kube-apiserver
# systemctl start kube-controller-manager
# systemctl start kube-scheduler
# systemctl start kubelet
# systemctl start kube-proxy
```

至此，一个单机版的 Kubernetes 集群环境就安装启动完成了。

接下来，我们可以在这个单机版的 Kubernetes 集群中上手练习了。

注：本书示例中的 Docker 镜像下载地址为 <https://hub.docker.com/u/kubeguide/>。

1.3.2 启动 MySQL 服务

首先为 MySQL 服务创建一个 RC 定义文件：mysql-rc.yaml，下面给出了该文件的完整内容和解释，如图 1.2 所示。

apiVersion: v1	
kind: ReplicationController	副本控制器 RC
metadata:	
name: mysql	RC 的名称，全局唯一
spec:	
replicas: 1	Pod 副本期待数量
selector:	
app: mysql	符合目标的 Pod 拥有此标签
template:	根据此模板创建 Pod 的副本（实例）
metadata:	
labels:	
app: mysql	Pod 副本拥有的标签，对应 RC 的 Selector
spec:	
containers:	Pod 容器的定义部分
- name: mysql	容器的名称
image: mysql	容器对应的 Docker Image
ports:	
- containerPort: 3306	容器暴露的端口号
env:	注入到容器内的环境变量
- name: MYSQL_ROOT_PASSWORD	
value: "123456"	

图 1.2 RC 的定义和解说图

yaml 定义文件中的 kind 属性，用来表明此资源对象的类型，比如这里的值为“ReplicationController”，表示这是一个 RC；spec 一节中是 RC 的相关属性定义，比如 spec.selector 是 RC 的 Pod 标签（Label）选择器，即监控和管理拥有这些标签的 Pod 实例，确保当前集群上始终有且仅有 replicas 个 Pod 实例在运行，这里我们设置 replicas=1 表示只能运行一个 MySQL Pod 实例。当集群中运行的 Pod 数量小于 replicas 时，RC 会根据 spec.template 一节中定义的 Pod 模板来生成一个新的 Pod 实例，spec.template.metadata.labels 指定了该 Pod 的标签，需要特别注意的是：这里的 labels 必须匹配之前的 spec.selector，否则此 RC 每次创建了一个无法匹配 Label 的 Pod，就会不停地尝试创建新的 Pod，最终陷入“只为他人做嫁衣”的悲惨世界中，永无翻身之时。

创建好 redis-master-controller.yaml 文件以后，为了将它发布到 Kubernetes 集群中，我们在 Master 节点执行命令：

```
# kubectl create -f mysql-rc.yaml
replicationcontroller "mysql" created
```

接下来，我们用 `kubectl` 命令查看刚刚创建的 RC：

```
# kubectl get rc
NAME          DESIRED  CURRENT  AGE
mysql         1        1        7m
```

查看 Pod 的创建情况时，可以运行下面的命令：

```
# kubectl get pods
NAME             READY    STATUS    RESTARTS  AGE
mysql-c95jc      1/1     Running   0         9m
```

我们看到一个名为 `mysql-xxxxx` 的 Pod 实例，这是 Kubernetes 根据 `mysql` 这个 RC 的定义自动创建的 Pod。由于 Pod 的调度和创建需要花费一定的时间，比如需要一定的时间来确定调度到哪个节点上，以及下载 Pod 里容器的镜像需要一段时间，所以一开始我们看到 Pod 的状态将显示为 `Pending`。当 Pod 成功创建完成以后，状态最终会被更新为 `Running`。

我们通过 `docker ps` 指令查看正在运行的容器，发现提供 MySQL 服务的 Pod 容器已经创建并正常运行了，此外，你会发现 MySQL Pod 对应的容器还多创建了一个来自谷歌的 `pause` 容器，这就是 Pod 的“根容器”，详见 1.4.3 节的说明。

```
# docker ps | grep mysql
72ca992535b4 mysql
"docker-entrypoint.sh" 12 minutes ago    Up 12 minutes
k8s_mysql.86dc506e_mysql-c95jc_default_511d6705-5051-11e6-a9d8-000c29ed42c1_9f89d0b4
76c1790aad27          gcr.io/google_containers/pause-amd64:3.0
"/pause"              12 minutes ago    Up 12 minutes
k8s_POD.16b20365_mysql-c95jc_default_511d6705-5051-11e6-a9d8-000c29ed42c1_28520aba
```

最后，我们创建一个与之关联的 Kubernetes Service——MySQL 的定义文件（文件名为 `mysql-svc.yaml`），完整的内容和解释如图 1.3 所示。

```
apiVersion: v1
kind: Service          _____ 表明是 Kubernetes Service
metadata:
  name: mysql          _____  Service 的全局唯一名称
spec:
  ports:
    - port: 3306       _____  Service 提供服务的端口号
  selector:            _____  Service 对应的 Pod 拥有这里定义的标签
    app: mysql
```

图 1.3 Service 的定义和解说图

其中，`metadata.name` 是 Service 的服务名（`ServiceName`）；`port` 属性则定义了 Service 的虚端口；`spec.selector` 确定了哪些 Pod 副本（实例）对应到本服务。类似地，我们通过 `kubectl create` 命令创建 Service 对象。

运行 `kubecttl` 命令，创建 `service`:

```
# kubecttl create -f mysql-svc.yaml
service "mysql" created
```

运行 `kubecttl` 命令，可以查看到刚刚创建的 `service`:

```
# kubecttl get svc
NAME          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
mysql         169.169.253.143 <none>           3306/TCP         48s
```

注意到 MySQL 服务被分配了一个值为 169.169.253.143 的 Cluster IP 地址，这是一个虚地址，随后，Kubernetes 集群中其他新创建的 Pod 就可以通过 Service 的 Cluster IP+端口号 3306 来连接和访问它了。

在通常情况下，Cluster IP 是在 Service 创建后由 Kubernetes 系统自动分配的，其他 Pod 无法预先知道某个 Service 的 Cluster IP 地址，因此需要一个服务发现机制来找到这个服务。为此，最初的时候，Kubernetes 巧妙地使用了 Linux 环境变量 (Environment Variable) 来解决这个问题，后面会详细说明其机制。现在我们只需知道，根据 Service 的唯一名字，容器可以从环境变量中获取到 Service 对应的 Cluster IP 地址和端口，从而发起 TCP/IP 连接请求了。

1.3.3 启动 Tomcat 应用

上面我们定义和启动了 MySQL 服务，接下来我们采用同样的步骤，完成 Tomcat 应用的启动过程。首先，创建对应的 RC 文件 `myweb-rc.yaml`，内容如下：

```
kind: ReplicationController
metadata:
  name: myweb
spec:
  replicas: 5
  selector:
    app: myweb
  template:
    metadata:
      labels:
        app: 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'
```

注意到上面 RC 对应的 Tomcat 容器里引用了 MYSQL_SERVICE_HOST=mysql 这个环境变量，而“mysql”恰好是我们之前定义的 MySQL 服务的服务名，运行下面的命令，完成 RC 的创建和验证工作：

```
#kubectl create -f myweb-rc.yaml
replicationcontroller "myweb" created

# kubectl get pods
NAME          READY    STATUS    RESTARTS   AGE
mysql-c95jc   1/1      Running   0           2h
myweb-g9pmm   1/1      Running   0           3s
```

最后，创建对应的 Service。以下是完整的 yaml 定义文件（myweb-svc.yaml）：

```
apiVersion: v1
kind: Service
metadata:
  name: myweb
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 30001
  selector:
    app: myweb
```

注意 type=NodePort 和 nodePort=30001 的两个属性，表明此 Service 开启了 NodePort 方式的外网访问模式，在 Kubernetes 集群之外，比如在本机的浏览器里，可以通过 30001 这个端口访问 myweb（对应到 8080 的虚端口上）。

运行 kubectl create 命令进行创建：

```
# kubectl create -f myweb-svc.yaml
You have exposed your service on an external port on all nodes in your
cluster. If you want to expose this service to the external internet, you may
need to set up firewall rules for the service port(s) (tcp:30001) to serve traffic.
See http://releases.k8s.io/release-1.3/docs/user-guide/services-firewalls.md
for more details.
service "myweb" created
```

我们看到上面有提示信息，意思是需要把 30001 这个端口在防火墙上打开，以便外部的访问能穿过防火墙。

运行 kubectl 命令，查看创建的 Service：

```
# kubectl get services
NAME          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
mysql         169.169.253.143 <none>        3306/TCP        44m
myweb         169.169.149.215 <nodes>       8080/TCP        4m
kubernetes    169.169.0.1    <none>        443/TCP        16d
```

至此，我们的第 1 个 Kubernetes 例子搭建完成了，在下一节中我们验证结果。

1.3.4 通过浏览器访问网页

经过上面的几个步骤，我们终于成功实现了 Kubernetes 上第 1 个例子的部署搭建工作。现在一起来见证成果吧，在你的笔记本上打开浏览器，输入 `http://虚拟机 IP:30001/demo/`。

比如虚机 IP 为 192.168.18.131（可以通过 `#ip a` 命令进行查询），在浏览器里输入地址 `http://192.168.18.131: 30001/demo/`后，看到了如图 1.4 所示的网页界面，那么恭喜你，之前的努力没有白费，顺利闯关成功！

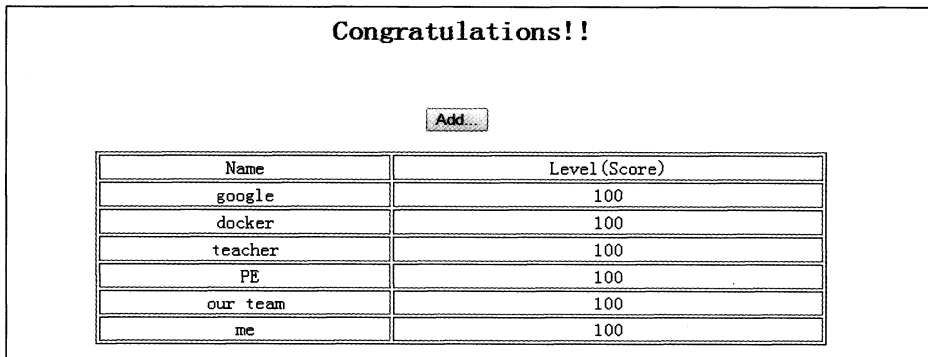
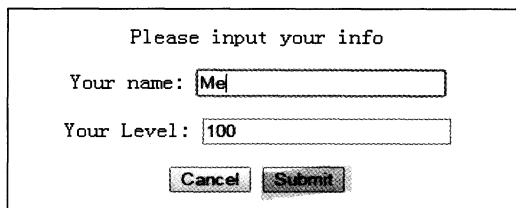


图 1.4 通过浏览器访问 Tomcat 应用

如果看不到这个网页，那么可能有几个原因：比如防火墙的问题，无法访问 30001 端口，或者因为你是通过代理上网的，浏览器错把虚拟机的 IP 地址当成远程地址了。可以在虚拟机上直接运行 `curl localhost:30001` 来验证此端口是否能被访问，如果还是不能访问，那么这肯定不是机器的问题……

接下来可以尝试单击“Add...”按钮添加一条记录并提交，如图 1.5 所示，提交以后，数据就被写入 MySQL 数据库中了。

至此，我们终于完成了 Kubernetes 上的 Tomcat 例子，这个例子并不是很复杂。我们也看到，相对于传统的分布式应用的部署方式，在 Kubernetes 之上我们仅仅通过一些很容易理解的配置文件和相关的简单命令就完成了对整个集群的部署，这让我们惊诧于 Kubernetes 的创新和强大。



Please input your info

Your name:

Your Level:

图 1.5 在留言板网页添加新的留言

下一节，我们将开始对 Kubernetes 中的基本概念和术语进行全面学习，在这之前，读者可以继续研究下这个例子里的一些拓展内容，如下所述。

- 研究 RC、Service 等文件的格式。
- 熟悉 kubectl 的子命令。
- 手工停止某个 Service 对应的容器进程，然后观察有什么现象发生。
- 修改 RC 文件，改变副本数量，重新发布，观察结果。

1.4 Kubernetes 基本概念和术语

Kubernetes 中的大部分概念如 Node、Pod、Replication Controller、Service 等都可以看作一种“资源对象”，几乎所有的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。

1.4.1 Master

Kubernetes 里的 Master 指的是集群控制节点，每个 Kubernetes 集群里需要有一个 Master 节点来负责整个集群的管理和控制，基本上 Kubernetes 所有的控制命令都是发给它，它来负责具体的执行过程，我们后面所有执行的命令基本都是在 Master 节点上运行的。Master 节点通常会占据一个独立的 X86 服务器（或者一个虚拟机），一个主要的原因是它太重要了，它是整个集群的“首脑”，如果它宕机或者不可用，那么我们所有的控制命令都将失效。

Master 节点上运行着以下一组关键进程。

- 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”。

其实 Master 节点上往往还启动了一个 etcd Server 进程, 因为 Kubernetes 里的所有资源对象的数据全部是保存在 etcd 中的。

1.4.2 Node

除了 Master, Kubernetes 集群中的其他机器被称为 Node 节点, 在较早的版本中也被称为 Minion。与 Master 一样, Node 节点可以是一台物理主机, 也可以是一台虚拟机。Node 节点才是 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去。

每个 Node 节点上都运行着以下一组关键进程。

- ◎ **kubelet**: 负责 Pod 对应的容器的创建、启停等任务, 同时与 Master 节点密切协作, 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 节点可以在运行期间动态增加到 Kubernetes 集群中, 前提是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 节点汇报自身的情报, 例如操作系统、Docker 版本、机器的 CPU 和内存情况, 以及之前有哪些 Pod 在运行等, 这样 Master 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 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集群中有多少个 Node:

```
# kubectl get nodes
NAME                                STATUS    AGE
kubernetes-minion1                 Ready     2d
```

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

```
$ kubectl describe node kubernetes-minion1

Name:      k8s-node-1
```

Kubernetes 权威指南：从 Docker 到 Kubernetes 实践全接触（第 2 版）

```
Labels:      beta.kubernetes.io/arch=amd64
            beta.kubernetes.io/os=linux
            kubernetes.io/hostname=k8s-node-1
Taints:      <none>
CreationTimestamp:  Wed, 06 Jul 2016 11:46:41 +0800
Phase:
Conditions:
  Type          Status LastHeartbeatTime          xxxx
  OutOfDisk      False Sat, 09 Jul 2016 08:17:39 +0800  Wed  ....
  MemoryPressure False Sat, 09 Jul 2016 08:17:39 +0800  Wed  .....
  Ready          True  Sat, 09 Jul 2016 08:17:39 +0800  Wed  .....
Addresses:    192.168.18.131,192.168.18.131
Capacity:
  alpha.kubernetes.io/nvidia-gpu: 0
  cpu:                             4
  memory:                         1868692Ki
  pods:                           110
Allocatable:
  alpha.kubernetes.io/nvidia-gpu: 0
  cpu:                             4
  memory:                         1868692Ki
  pods:                           110
System Info:
  Machine ID:      6e4e2af2afeb42b9aac47d866aa56ca0
  System UUID:     564D63D3-9664-3393-A3DC-9CD424ED42C1
  Boot ID:         b0c34f9f-76ab-478e-9771-bd4fe6e98880
  Kernel Version:  3.10.0-327.22.2.el7.x86_64
  OS Image:        CentOS Linux 7 (Core)
  Operating System: linux
  Architecture:    amd64
  Container Runtime Version: docker://1.11.2
  Kubelet Version:  v1.3.0
  Kube-Proxy Version: v1.3.0
ExternalID:      k8s-node-1
Non-terminated Pods: (1 in total)
  Namespace      Name      CPU Requests  CPU Limits Memory xxx
  -----
  kube-system    kube-dns-v11-wxdhf    310m (7%)  310m (7%)  170Mi (9%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted. More info:
  CPU Requests      CPU Limits Memory Requests      Memory Limits
  -----
      310m (7%)      310m (7%)  170Mi (9%)      170Mi (9%)
No events.
```

上述命令展示了 Node 的如下关键信息。

- ④ Node 基本信息：名称、标签、创建时间等。
- ④ Node 当前的运行状态，Node 启动以后会做一系列的自检工作，比如磁盘是否满了，如果满了就标注 `OutOfDisk=True`，否则继续检查内存是否不足（`MemoryPressure=True`），最后一切正常，就切换为 Ready 状态（`Ready=True`），这种情况表示 Node 处于健康状态，可以在其上创建新的 Pod。
- ④ Node 的主机地址与主机名。
- ④ Node 上的资源总量：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，注意到目前 Kubernetes 已经实验性地支持 GPU 资源分配了（`alpha.kubernetes.io/nvidia-gpu=0`）。
- ④ Node 可分配资源量：描述 Node 当前可用于分配的资源量。
- ④ 主机系统信息：包括主机的唯一标识 UUID、Linux kernel 版本号、操作系统类型与版本、Kubernetes 版本号、kubelet 与 kube-proxy 的版本号等。
- ④ 当前正在运行的 Pod 列表概要信息。
- ④ 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。
- ④ Node 相关的 Event 信息。

1.4.3 Pod

Pod 是 Kubernetes 的最重要也最基本的概念，如图 1.6 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。

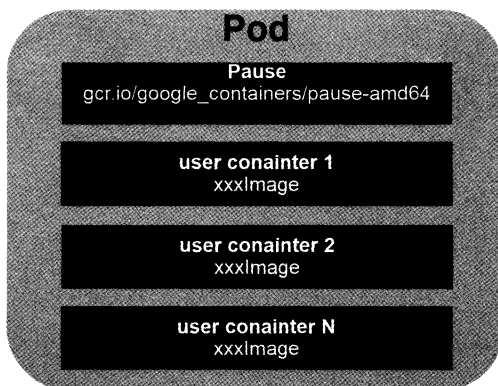


图 1.6 Pod 的组成与容器的关系

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，我们难以对“整体”简单地进行判断及有效地进行行动。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、Openvswitch 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，它并不存放在 Kubernetes 的 etcd 存储里，而是存放在某个具体的 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，就会被放入到 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 容器并启动起来。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，则会将这个 Node 上的所有 Pod 重新调度到其他节点上。Pod、容器与 Node 的关系如图 1.7 所示。

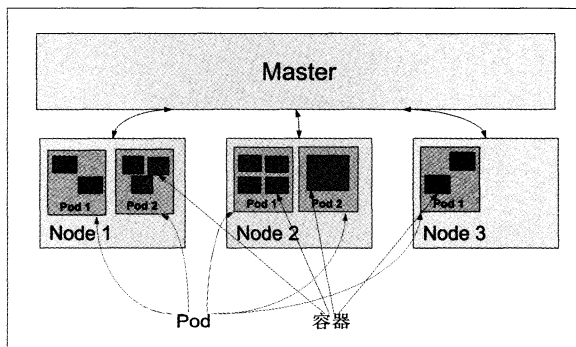


图 1.7 Pod、容器与 Node 的关系

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 Hello World 例子里用到的 myweb 这个 Pod 的资源定义文件：

```
apiVersion: v1
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 的名字，metadata 里还能定义资源对象的标签 (Label)，这里声明 myweb 拥有一个 name=myweb 的标签 (Label)。Pod 里所包含的容器组的定义则在 spec 一节中声明，这里定义了一个名字为 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) 上启动容器进程。Pod 的 IP 加上这里的容器端口 (containerPort)，就组成了一个新的概念——Endpoint，它代表着此 Pod 里的一个服务进程的对外通信地址。一个 Pod 也存在着具有多个 Endpoint 的情况，比如当我们把 Tomcat 定义为一个 Pod 的时候，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 是定义在 Pod 之上，然后被各个容器挂载到自己的文件系统里的。

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，用来定位问题的原因，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    LastSeen    Count   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。

一个 CPU 的配额对于绝大多数容器来说是相当大的一个资源配额了，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 Kill 并重启。

通常我们会把 Request 设置为一个比较小的数值，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。比如下面这段定义，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"
```

本节最后，笔者给出 Pod 及 Pod 周边对象的示意图作为总结，如图 1.8 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。

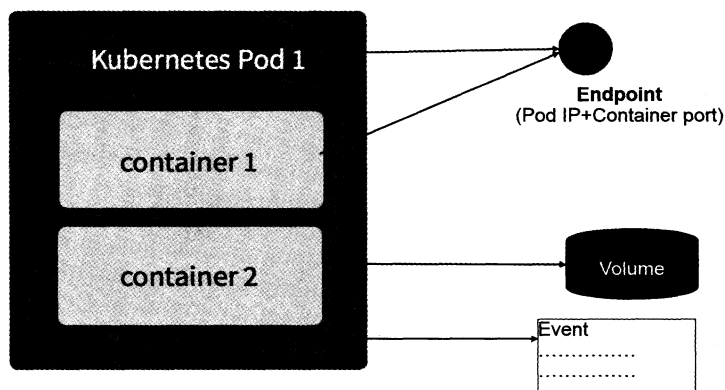


图 1.8 Pod 及周边对象

1.4.4 Label（标签）

Label 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去，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，以便于灵活、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；或者监控和分析应用（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 pod's name = 'redis-slave'` 这样的语句。当前有两种 Label Selector 的表达式：基于等式的（Equality-based）和基于集合的（Set-based），前者采用“等式类”的表达式匹配标签，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。

而后者则使用集合操作的表达式匹配标签，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
```

Label Selector 在 Kubernetes 中的重要使用场景有以下几处。

- ⊙ kube-controller 进程通过资源对象 RC 上定义的 Label Selector 来筛选要监控的 Pod 副本的数量，从而实现 Pod 副本的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，自动建立起每个 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。

在前面的留言板例子中，我们只使用了一个 `name=XXX` 的 Label Selector。让我们看一个更复杂的例子。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，如图 1.9 所示，如果我们设置了“`role=frontend`”的 Label Selector，则会选取到 Node 1 和 Node 2 上的 Pod。

而设置“`release=beta`”的 Label Selector，则会选取到 Node 2 和 Node 3 上的 Pod，如图 1.10 所示。

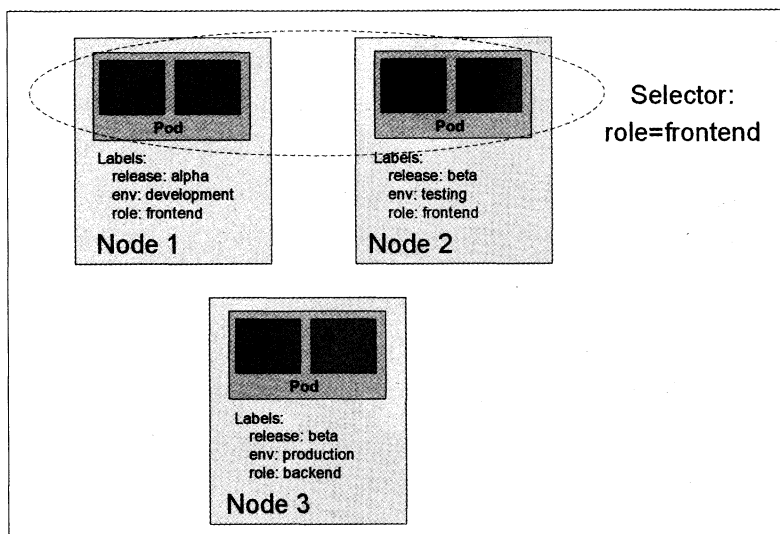


图 1.9 Label Selector 的作用范围 1

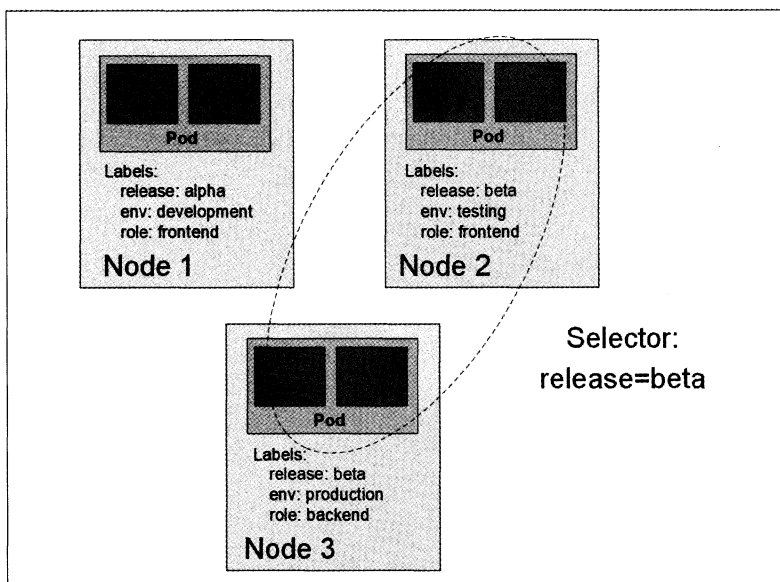


图 1.10 Label Selector 的作用范围 2

总结：使用 Label 可以给对象创建多组标签，Label 和 Label Selector 共同构成了 Kubernetes 系统中最核心的应用模型，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。