1.4.5 Replication Controller（RC）

之前已经对 Replication Controller（以后简称 RC）的定义和作用做了一些说明，本节对 RC 的概念进行深入描述。

RC 是 Kubernetes 系统中的核心概念之一，简单来说，它其实是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（replicas）。
- 用于筛选目标 Pod 的 Label Selector。
- 当 Pod 的副本数量小于预期数量的时候，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-demo
        image: tomcat
        imagePullPolicy: IfNotPresent
        env:
        - name: GET_HOSTS_FROM
          value: dns
        ports:
        - containerPort: 80
```

当我们定义了一个 RC 并提交到 Kubernetes 集群中以后，Master 节点上的 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，否则系统就会再自动创建一些 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）。

下面我们以 3 个 Node 节点的集群为例，说明 Kubernetes 如何通过 RC 来实现 Pod 副本数量自动控制的机制。假如我们的 RC 里定义 redis-slave 这个 Pod 需要保持 3 个副本，系统将可能在其中的两个 Node 上创建 Pod。图 1.11 描述了在两个 Node 上创建 redis-slave Pod 的情形。

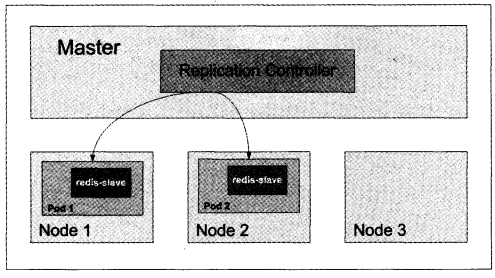

图 1.11　在两个 Node 上创建 redis-slave Pod

假设 Node 2 上的 Pod 2 意外终止，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，以保证整个集群中始终有两个 redis-slave Pod 在运行。

如图 1.12 所示，系统可能选择 Node 3 或者 Node 1 来创建一个新的 Pod。

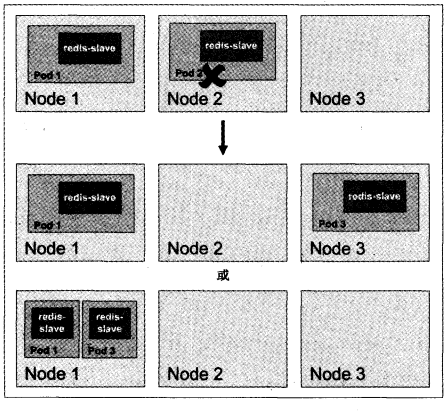

图 1.12　根据 RC 定义创建新的 Pod

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）功能，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 的执行结果如图 1.13 所示。

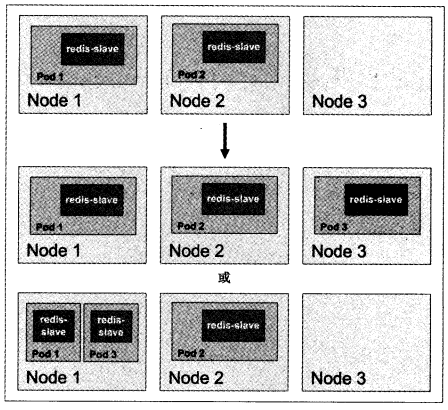

图 1.13　Scaling 的执行结果

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。

当我们的应用升级时，通常会通过 Build 一个新的 Docker 镜像，并用新的镜像版本来替代旧的版本的方式达到目的。在系统升级的过程中，我们希望是平滑的方式，比如当前系统中 10 个对应的旧版本的 Pod，最佳的方式是旧版本的 Pod 每次停止一个，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 都已经是新版本的时候，升级过程完成。通过 RC 的机制，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），具体的操作方法详见第 4 章。

由于 Replication Controller 与 Kubernetes 代码中的模块 Replication Controller 同名，同时这个词也无法准确表达它的本意，所以在 Kubernetes 1.2 的时候，它就升级成了另外一个新的概念——Replica Set，官方解释为“下一代的 RC”，它与 RC 当前存在的唯一区别是：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: extensions/v1beta1
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 的绝大部分命令都同样适用于 Replica Set。此外，当前我们很少单独使用 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。当我们使用 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 这两个重要资源对象逐步替换了之前的 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。

最后我们总结一下关于 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 的创建过程及副本数量的自动控制。
- 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 的扩容或缩容功能。
- 通过改变 RC 里 Pod 模板中的镜像版本，可以实现 Pod 的滚动升级功能。

1.4.6　Deployment

Deployment 是 Kubernetes 1.2 引入的新概念，引入的目的是为了更好地解决 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、它的 YAM 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 副本的创建过程。
- 检查 Deployment 的状态来看部署动作是否完成（Pod 副本的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。

◎ 挂起或者恢复一个 Deployment。

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

```
apiVersion: extensions/v1beta1                    apiVersion: v1
kind: Deployment                                  kind: ReplicaSet
metadata:                                         metadata:
  name: nginx-deployment                            name: nginx-repset
```

下面我们通过运行一些例子来一起直观地感受这个新概念。首先创建一个名为 tomcat-deployment.yaml 的 Deployment 描述文件，内容如下：

```
apiVersion: extensions/v1beta1
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：

```
# kubectl create -f tomcat-deployment.yaml
deployment "tomcat-deploy" created
```

运行下述命令查看 Deployment 的信息：

```
# kubectl get deployments
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
tomcat-deploy       1         1         1            1           4m
```

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 Deployment 里定义的 Replica。

- CURRENT：当前 Replica 的值，实际上是 Deployment 所创建的 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，我们看到它的命名跟 Deployment 的名字有关系：

```
# kubectl get rs
NAME                          DESIRED   CURRENT   AGE
tomcat-deploy-1640611518      1         1         1m
```

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 的名字为前缀，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：

```
# kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
tomcat-deploy-1640611518-zhrsc      1/1       Running   0          3m
```

运行 kubectl describe deployments，可以清楚地看到 Deployment 控制的 Pod 的水平扩展过程，此命令的输出比较多，这里不再赘述。

1.4.7　Horizontal Pod Autoscaler（HPA）

在前两节提到过，通过手工执行 kubectl scale 命令，我们可以实现 Pod 扩容或缩容。如果仅仅到此为止，显然不符合谷歌对 Kubernetes 的定位目标——自动化、智能化。在谷歌看来，分布式系统要能够根据当前负载的变化情况自动触发水平扩展或缩容的行为，因为这一过程可能是频繁发生的、不可预料的，所以手动控制的方式是不现实的。

因此，Kubernetes 的 1.0 版本实现后，这帮大牛们就已经在默默研究 Pod 智能扩容的特性了，并在 Kubernetes 1.1 的版本中首次发布这一重量级新特性——Horizontal Pod Autoscaling。随后的 1.2 版本中 HPA 被升级为稳定版本（apiVersion: autoscaling/v1），但同时仍然保留旧版本（apiVersion: extensions/v1beta1），官方的计划是在 1.3 版本里先移除旧版本，并且会在 1.4 版本里彻底移除旧版本的支持。

Horizontal Pod Autoscaling 简称 HPA，意思是 Pod 横向自动扩容，与之前的 RC、Deployment 一样，也属于一种 Kubernetes 资源对象。通过追踪分析 RC 控制的所有目标 Pod 的负载变化情况，来确定是否需要针对性地调整目标 Pod 的副本数，这是 HPA 的实现原理。当前，HPA 可以有以下两种方式作为 Pod 负载的度量指标。

- CPUUtilizationPercentage。
- 应用程序自定义的度量指标，比如服务在每秒内的相应的请求数（TPS 或 QPS）。

CPUUtilizationPercentage 是一个算术平均值，即目标 Pod 所有副本自身的 CPU 利用率的平均值。一个 Pod 自身的 CPU 利用率是该 Pod 当前 CPU 的使用量除以它的 Pod Request 的值，比如我们定义一个 Pod 的 Pod Request 为 0.4，而当前 Pod 的 CPU 使用量为 0.2，则它的 CPU 使用率为 50%，如此一来，我们就可以就算出来一个 RC 控制的所有 Pod 副本的 CPU 利用率的算术平均值了。如果某一时刻 CPUUtilizationPercentage 的值超过 80%，则意味着当前的 Pod 副本数很可能不足以支撑接下来更多的请求，需要进行动态扩容，而当请求高峰时段过去后，Pod 的 CPU 利用率又会降下来，此时对应的 Pod 副本数应该自动减少到一个合理的水平。

CPUUtilizationPercentage 计算过程中使用到的 Pod 的 CPU 使用量通常是 1 分钟内的平均值，目前通过查询 Heapster 扩展组件来得到这个值，所以需要安装部署 Heapster，这样一来便增加了系统的复杂度和实施 HPA 特性的复杂度，因此，未来的计划是 Kubernetes 自身实现一个基础性能数据采集模块，从而更好地支持 HPA 和其他需要用到基础性能数据的功能模块。此外，我们也看到，如果目标 Pod 没有定义 Pod Request 的值，则无法使用 CPUUtilizationPercentage 来实现 Pod 横向自动扩容的能力。除了使用 CPUUtilizationPercentage，Kubernetes 从 1.2 版本开始，尝试支持应用程序自定义的度量指标，目前仍然为实验特性，不建议在生产环境中使用。

下面是 HPA 定义的一个具体例子：

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 90
```

根据上面的定义，我们可以知道这个 HPA 控制的目标对象为一个名叫 php-apache 的 Deployment 里的 Pod 副本，当这些 Pod 副本的 CPUUtilizationPercentage 的值超过 90%时会触发自动动态扩容行为，扩容或缩容时必须满足的一个约束条件是 Pod 的副本数要介于 1 与 10 之间。

除了可以通过直接定义 yaml 文件并且调用 kubectrl create 的命令来创建一个 HPA 资源对象的方式，我们还能通过下面的简单命令行直接创建等价的 HPA 对象：

```
# kubectl autoscale deployment php-apache --cpu-percent=90 --min=1 --max=10
```

第 2 章将会给出一个完整的 HPA 例子来说明其用法和功能。

1.4.8　Service（服务）

1. 概述

Service 也是 Kubernetes 里的最核心的资源对象之一，Kubernetes 里的每个 Service 其实就是我们经常提起的微服务架构中的一个“微服务”，之前我们所说的 Pod、RC 等资源对象其实都是为这节所说的“服务”——Kubernetes Service 做“嫁衣”的。图 1.14 显示了 Pod、RC 与 Service 的逻辑关系。

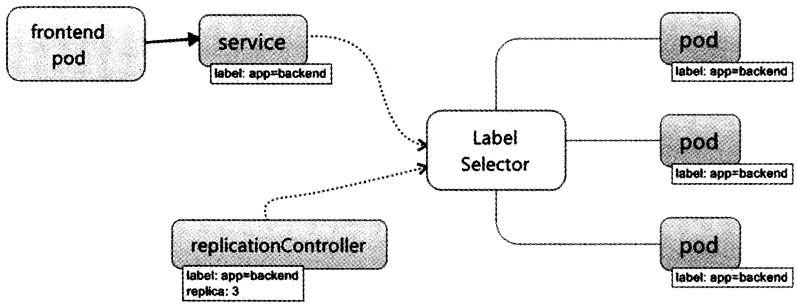

图 1.14　Pod、RC 与 Service 的关系

从图 1.14 中我们看到，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。而 RC 的作用实际上是保证 Service 的服务能力和服务质量始终处于预期的标准。

通过分析、识别并建模系统中的所有服务为微服务——Kubernetes Service，最终我们的系统由多个提供不同业务能力而又彼此独立的微服务单元所组成，服务之间通过 TCP/IP 进行通信，从而形成了我们强大而又灵活的弹性网格，拥有了强大的分布式能力、弹性扩展能力、容错能力，与此同时，我们的程序架构也变得简单和直观许多，如图 1.15 所示。

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 端口的转发列表中，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，而客户端的请求最后会被转发到哪个 Pod，则由负载均衡器的算法所决定。

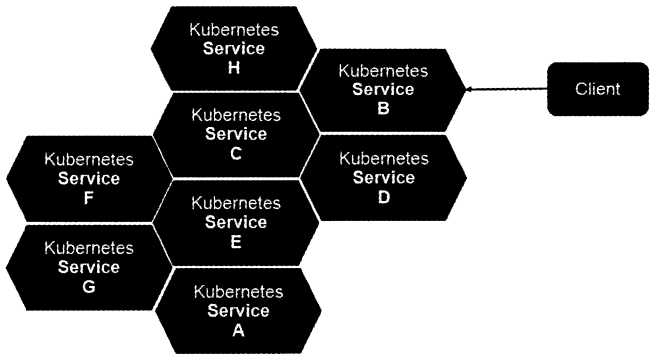

图 1.15　Kubernetes 所提供的微服务网格架构

Kubernetes 也遵循了上述常规做法，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，它负责把对 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 不是共用一个负载均衡器的 IP 地址，而是每个 Service 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，下面我们动手创建一个 Service，来加深对它的理解。首先我们创建一个名为 tomcat-service.yaml 的定义文件，内容如下：

```
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：

```
#kubectl create -f tomcat-server.yaml
service "tomcat-service" created
```

注意到我们之前在 tomcat-deployment.yaml 里定义的 Tomcat 的 Pod 刚好拥有这个标签，所以我们刚才创建的 tomcat-service 已经对应到了一个 Pod 实例，运行下面的命令可以查看 tomcat-service 的 Endpoint 列表，其中 172.17.1.3 是 Pod 的 IP 地址，端口 8080 是 Container 暴露的端口：

```
# kubectl get endpoints
NAME            ENDPOINTS           AGE
kubernetes      192.168.18.131:6443   15d
tomcat-service   172.17.1.3:8080      1m
```

你可能有疑问："说好的 Service 的 Cluster IP 呢？怎么没有看到？"我们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：

```
# kubectl get svc tomcat-service -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2016-07-21T17:05:52Z
  name: tomcat-service
  namespace: default
  resourceVersion: "23964"
  selfLink: /api/v1/namespaces/default/services/tomcat-service
  uid: 61987d3c-4f65-11e6-a9d8-000c29ed42c1
spec:
  clusterIP: 169.169.65.227
  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；而 port 属性则定义了 Service 的虚端口。前面我们定义 Tomcat 服务的时候，没有指定 targetPort，则默认 targetPort 与 port 相同。

接下来，我们来看看 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 定义一个名字来区分。下面是 Tomcat 多端口的 Service 定义样例：

```
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命名呢？这就涉及 Kubernetes 的服务发现机制了，我们接下来进行讲解。

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，大部分分布式系统通过提供特定的 API 接口来实现服务发现的功能，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困难。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 都有一个唯一的 Cluster IP 以及唯一的名字，而名字是由开发者自己定义的，部署的时候也没必要改变，所以完全可以固定在配置中。接下来的问题就是如何通过 Service 的名字找到对应的 Cluster IP？

最早的时候 Kubernetes 采用了 Linux 环境变量的方式解决这个问题，即每个 Service 生成一些对应的 Linux 环境变量（ENV），并在每个 Pod 的容器在启动时，自动注入这些环境变量，以以下是 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
```

上述环境变量中，比较重要的是前 3 条环境变量，我们可以看到，每个 Service 的 IP 地址及端口都是有标准的命名规范的，遵循这个命名规范，就可以通过代码访问系统环境变量的方式得到所需的信息，实现服务调用。

考虑到环境变量的方式获取 Service 的 IP 与端口的方式仍然不太方便，不够直观，后来 Kubernetes 通过 Add-On 增值包的方式引入了 DNS 系统，把服务名作为 DNS 域名，这样一来，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 这些新兴的服务发现机制，后面的章节中我们会讲述如何部署这套 DNS 系统。

3. 外部系统访问 Service 的问题

为了更加深入地理解和掌握 Kubernetes，我们需要弄明白 Kubernetes 里的“三种 IP”这个关键问题，这三种 IP 分别如下。

- Node IP：Node 节点的 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，这是一个真实存在的物理网络，所有属于这个网络的服务器之间都能通过这个网络直接通信，不管它们中是否有部分节点不属于这个 Kubernetes 集群。这也表明了 Kubernetes 集群之外的节点访问 Kubernetes 集群之内的某个节点或者 TCP/IP 服务的时候，必须要通过 Node IP 进行通信。

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，前面我们说过，Kubernetes 要求位于不同 Node 上的 Pod 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，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 流量则是通过 Node IP 所在的物理网卡流出的。

最后，我们说说 Service 的 Cluster IP，它也是一个虚拟的 IP，但更像是一个“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，集群之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 集群之内，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的路由规则，与我们所熟知的 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、最有效、最常用的做法。具体做法如下，以 tomcat-service 为例，我们在 Service 的定义里做如下扩展即可（黑体字部分）：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
 type: NodePort
 ports:
  - port: 8080
    nodePort: 31002
 selector:
    tier: frontend
```

其中，nodePort:31002 这个属性表明我们手动指定 tomcat-service 的 NodePort 为 31002，否则 Kubernetes 会自动分配一个可用的端口。接下来，我们在浏览器里访问 http://<nodePort IP>:31002/，就可以看到 Tomat 的欢迎界面了，如图 1.16 所示。

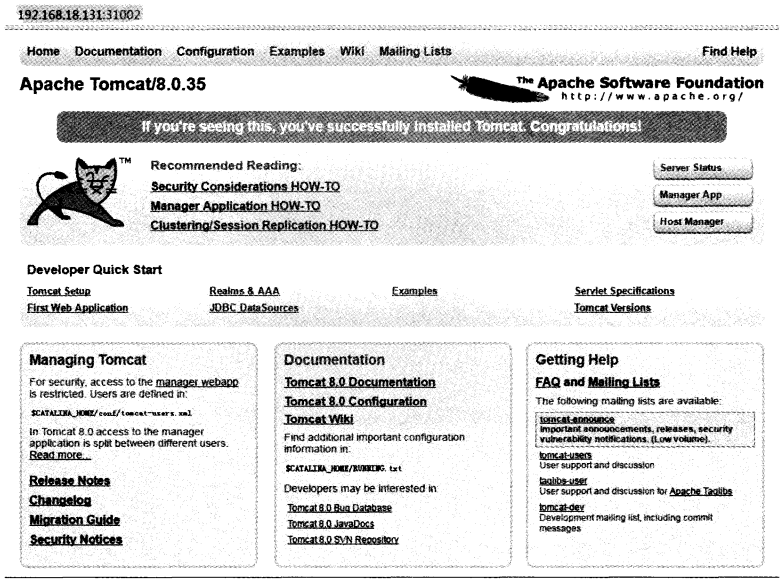

图 1.16　通过 NodePort 访问 Service

NodePort 的实现方式是在 Kubernetes 集群里的每个 Node 上为需要外部访问的 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，我们就可以看到有 NodePort 端口被监听：

```
# netstat -tlp | grep 31002
tcp6  0   0 [::]:31002        [::]:*              LISTEN      1125/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。如图 1.17 所示。

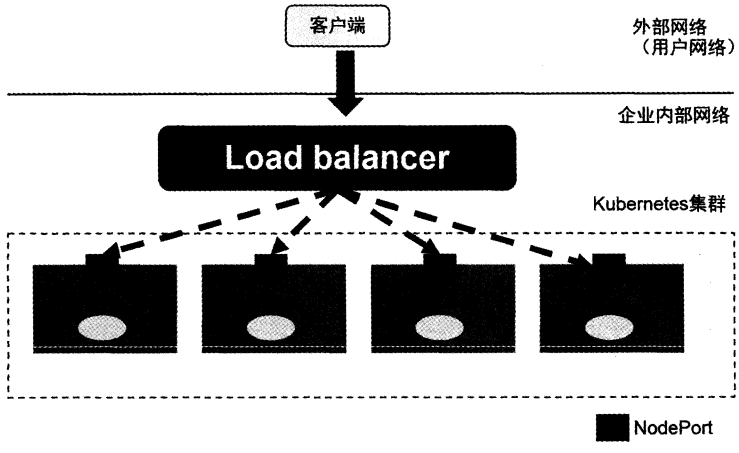

图 1.17　NodePort 与 Load balancer

图 1.17 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 GCE 公有云上，那么只要我们把 Service 的 type= NodePort 改为 type= LoadBalancer，此时 Kubernetes 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）也正在被开发。

1.4.9　Volume（存储卷）

Volume 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 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 Mount 到容器里的某个目录上。举例来说，我们要给之前的 Tomcat Pod 增加一个名字为 dataVol 的 Volume，并且 Mount 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（注意黑体字部分）：

```
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 这个新的资源对象来实现的，后面我们会详细说明。

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

1. emptyDir

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. gcePersistentDisk

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，你需要先创建一个永久磁盘（PD），才能使用 gcePersistentDisk。

使用 gcePersistentDisk 有以下一些限制条件。

◎ 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 mount 一个 volume。

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

```
aws ec2 create-volume --availability-zone eu-west-1a --size 10 --volume-type gp2
```

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 来管理存储卷。
- glusterfs：使用开源 GlusterFS 网络文件系统的目录挂载到 Pod 中。
- rbd：使用 Linux 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，所以这种类型的 volume 总是不会持久化的。

1.4.10　Persistent Volume

之前我们提到的 Volume 是定义在 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”并挂接到虚机上。Persistent Volume（简称 PV）和与之相关联的 Persistent Volume Claim（简称 PVC）也起到了类似的作用。

PV 可以理解成 Kubernetes 集群中的某个网络存储中对应的一块存储，它与 Volume 很类似，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 并不是定义在 Pod 上的，而是独立于 Pod 之外定义。
- PV 目前只有几种类型：GCE Persistent Disks、NFS、RBD、iSCSCI、AWS ElasticBlockStore、GlusterFS 等。

下面给出了 NFS 类型 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 想申请某种条件的 PV，则首先需要定义一个 PersistentVolumeClaim（PVC）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，我们说说 PV 的状态，PV 是有状态的对象，它有以下几种状态。

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

1.4.11 Namespace（命名空间）

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 可以查看到：

```
$ kubectl get namespaces
```

```
NAME          LABELS            STATUS
default       <none>            Active
```

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。

Namespace 的定义很简单。如下所示的 yaml 定义了名为 development 的 Namespace。

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

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，我们定义了一个名为 busybox 的 Pod，放入 development 这个 Namespace 里：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: development
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    name: busybox
```

此时，使用 kubectl get 命令查看将无法显示：

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
```

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：

```
# kubectl get pods --namespace=development
NAME          READY     STATUS    RESTARTS   AGE
busybox       1/1       Running   0          1m
```

当我们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。

1.4.12 Annotation（注解）

Annotation 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。而 Annotation 则是用户任意定义的“附加”信息，以便于外部工具进行查找，很多时候，Kubernetes 的模块自身会通过 Annotation 的方式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。

- 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。

1.4.13 小结

上述这些组件是 Kubernetes 系统的核心组件，它们共同构成了 Kubernetes 系统的框架和计算模型。通过对它们进行灵活组合，用户就可以快速、方便地对容器集群进行配置、创建和管理。除了本章所介绍的核心组件，在 Kubernetes 系统中还有许多辅助配置的资源对象，例如 LimitRange、ResourceQuota。另外，一些系统内部使用的对象 Binding、Event 等请参考 Kubernetes 的 API 文档。

在第 2 章中，我们将开始深入实践并全面掌握 Kubernetes 的各种使用技巧。