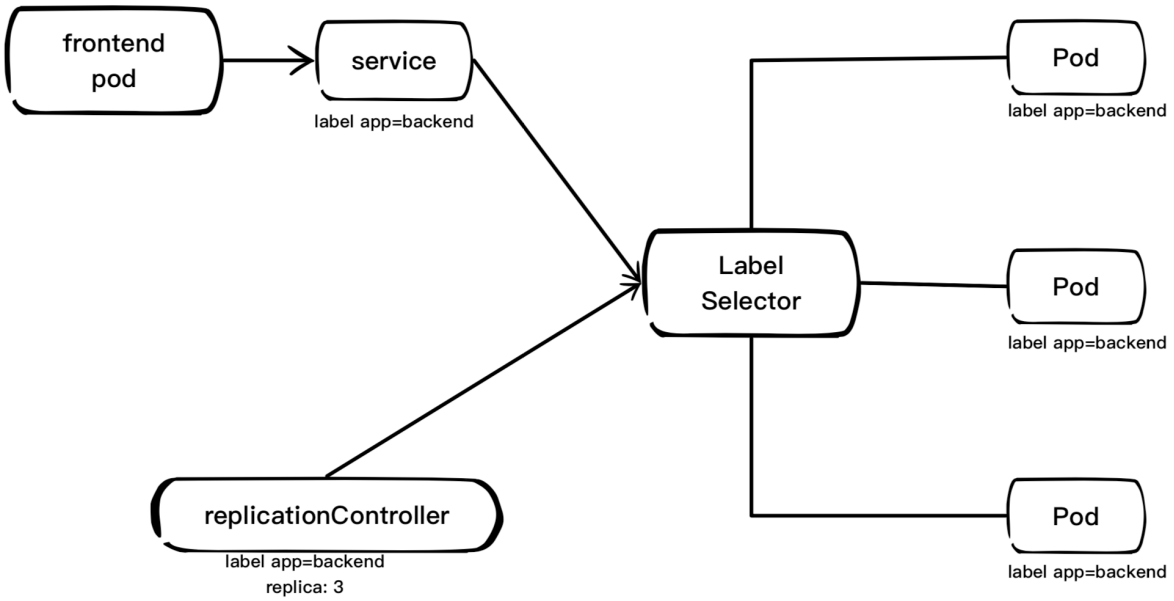
Service

Kubernetes Pod 是有生命周期的，它们可以被创建，也可以被销毁，然而一旦被销毁生命就永远结束。 通过 ReplicationController 能够动态地创建和销毁 Pod（例如，需要进行扩缩容，或者执行 滚动升级）。 每个 Pod 都会获取它自己的 IP 地址，即使这些 IP 地址不总是稳定可依赖的。 这会导致一个问题：在 Kubernetes 集群中，如果一组 Pod（称为 backend）为其它 Pod （称为 frontend）提供服务，那么那些 frontend 该如何发现，并连接到这组 Pod 中的哪些 backend 呢？

Service 也是 Kubernetes 里的就核心的资源对象之一，Kubernertes 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”。

下图显示了pod、RC与Service的逻辑关系

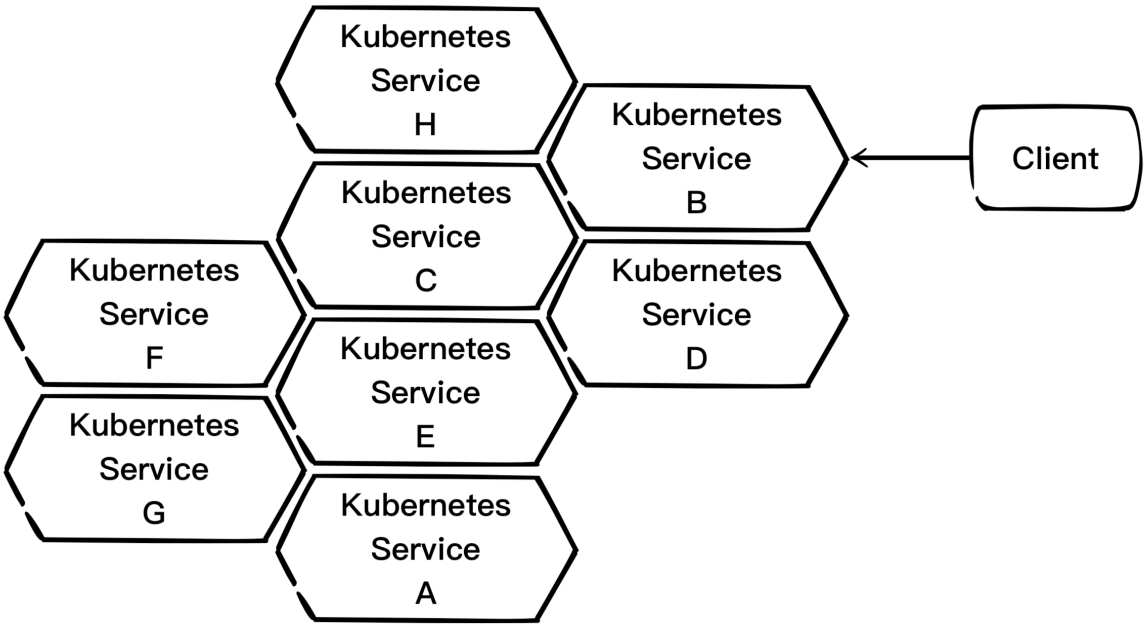


Pod、RC 与 Service 的关系

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。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，从而形成了强大而又灵活的弹性网络，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，程序架构也变得简单和直观许多：



Kubernetes 所提供的微服务网络架构

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。

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。

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。
下面创建一个 Service，来加深对它的理解。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：

```
[root@vlnx251101 ~]# kubectl create -f tomcat-service.yaml
service "tomcat-service" created
```

注意到之前的 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 已经对应到一个 Pod 实例，运行下面的命令可以查看 tomcat-service 的 Endpoint 列表，其中 172.30.88.3 的 Pod 的 IP 地址，端口 8080 是 Container 暴露的端口：

```
[root@vlnx251101 ~]# kubectl get endpoints
NAME                ENDPOINTS                                      AGE
kubernetes          192.168.251.101:6443                           1d
tomcat-service      172.30.88.3:8080                               30s
```

这里查看的实际是 Pod 的真是 IP 地址，并不是 Cluster IP，运行下面的命令查看 tomcat service 信息以及 Cluster IP 及更多的信息：

```
[root@vlnx251101 ~]# kubectl get svc tomcat-service -o
yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2018-03-11T08:35:06Z
  name: tomcat-service
  namespace: default
  resourceVersion: "81238"
  selfLink: /api/v1/namespaces/default/services/tomcat-
service
  uid: 19fcffde-2507-11e8-b74e-000c29526d85
spec:
  clusterIP: 10.254.120.216
  ports:
  - port: 8080
```

```
    protocol: TCP
    targetPort: 8080
  selector:
    tier: frontend
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

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 等常见的中间件。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？这涉及 Kubernetes 的服务发现机制了。

Kubernetes 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 及唯一的名字，而名字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？

最早时 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，以下是 tomcat-service 产生的环境变量条目：(下面的变量是Deployemt中创建的，不过这里需要先启动上方的两个 Port 的 Service 在启动 Deployment，这是因为如果先启动 Deployment 就获取不到下面的最新变量，启动完成后 exec 进入这个容器内执行 env 会看到下面类似的结果，先启动 Deployment 不会看到下面的类似的结果)

```
TOMCAT_SERVICE_SERVICE_HOST=10.254.120.216
TOMCAT_SERVICE_SERVICE_PORT_SERVICE_PORT=8080
TOMCAT_SERVICE_SERVICE_PORT_SHUTDOWN_PORT=8005
TOMCAT_SERVICE_SERVICE_PORT=8080
TOMCAT_SERVICE_PORT=tcp://10.254.120.216:8080
TOMCAT_SERVICE_PORT_8080_TCP_ADDR=10.254.120.216
TOMCAT_SERVICE_PORT_8080_TCP=tcp://10.254.120.216:8080
TOMCAT_SERVICE_PORT_8080_TCP_PROTO=tcp
TOMCAT_SERVICE_PORT_8080_TCP_PORT=8080
TOMCAT_SERVICE_PORT_8005_TCP_ADDR=10.254.120.216
TOMCAT_SERVICE_PORT_8005_TCP=tcp://10.254.120.216:8005
TOMCAT_SERVICE_PORT_8005_TCP_PROTO=tcp
```

```
TOMCAT_SERVICE_PORT_8005_TCP_PORT=8005
TOMCAT_SHA1=d1555c86ec84824da6785aa875fc6f28298c51cd
TOMCAT_VERSION=8.5.28
TOMCAT_NATIVE_LIBDIR=/usr/local/tomcat/native-jni-lib
TOMCAT_MAJOR=8
```

上述环境变量中，比较重要的是前 3 条环境变量，可以看到，每个 Service 的 IP 地址及端口都是标准的命名规范的，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。

外部系统访问 Service 问题

为了增加深入的理解掌握 Kubernetes，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之间通信，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。

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 能够彼此之间通信，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。

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，与自身熟知的 IP 路由有很大的不同。

根据上面的分析和总结，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，在 Service 的定义里做如下扩展即可：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
```

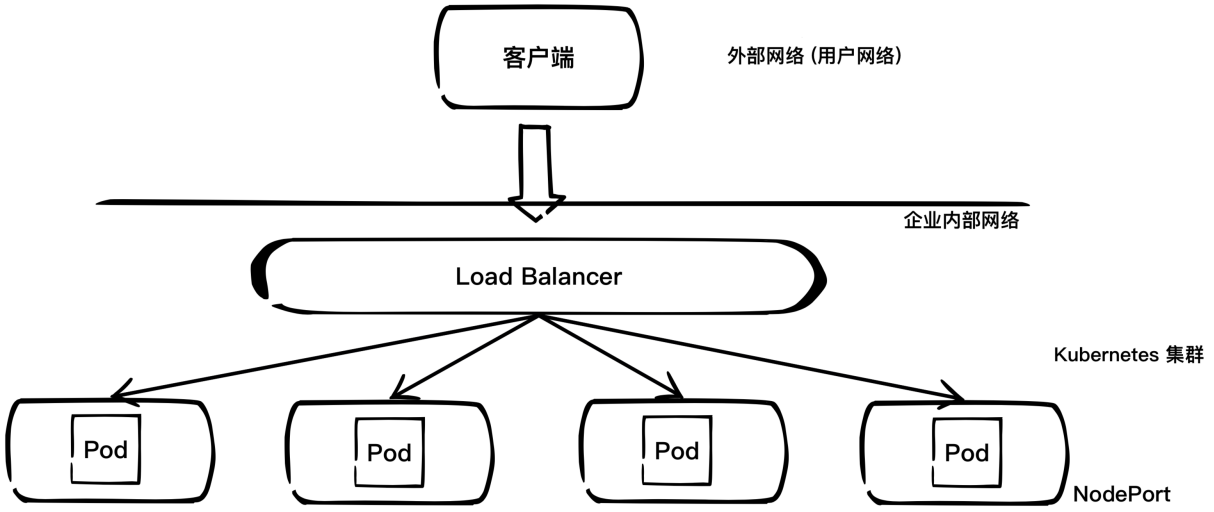
```
spec:
  type: NodePort
  ports:
  - port: 8080
    nodePort: 31002
  selector:
    tier: frontend
```

其实，nodePort: 31002 这个属性表名手动指定 tomcat-service 的 NodePort 为 31002，否则 Kubernetes 会自动分配一个可用的端口。接下来，可以再浏览器输入 http://youAddr:31002/ 来访问这个 Tomcat 了。

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 地址加具体的 NodePort 端口号即可访问此服务，在任意 Node 上运行 netstat 命令，可以看到有 NodePort 端口被监听：

```
[root@vlnx251101 ~]# netstat -tunpl | grep 31002
tcp6       0      0 :::31002                :::*                    LISTEN      1864/kube-proxy
```

但 NodePort 还没完全解决外部访问 Service 的所有问题，比如负载均衡问题，假如集群中有 10 个 Node，则此时最好有一个负载均衡，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。见下图：



NodePort 与 Load Balancer

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，通产需要配置一个对应的 Load Balancer 示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 提供自动化的解决方案，如果集群运行在谷歌的 GCE 公有云上，那么只要把 Service 的 type=NodePort 改为 type=LoadBalancer，此时 Kubernetes 会自动创建一个对应的 Load Balancer 示例被返回它的 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。