Kubernetes应用部署模型解析（原理篇）

发表于 2015-06-11 16:29 | 1277次阅读 | 来源 CSDN | 0 条评论 | 作者 龚永生

Kubernetes 容器 Docker 云计算

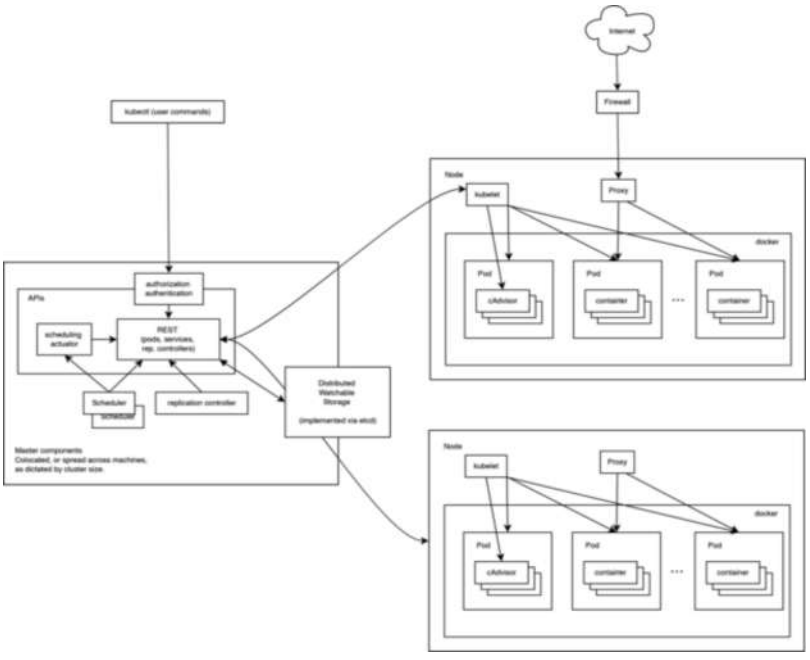
摘要：本系列文章着眼于实际部署，带您快速掌握Kubernetes。本文主要介绍部署之前需要了解的原理和概念，包括Kubernetes的组件结构，各个组件角色的功能，以及Kubernetes的应用模型等。

【编者按】Kubernetes可用来管理Linux容器集群，加速开发和简化运维（即DevOps）。但目前网络上关于Kubernetes的文章介绍性远多于实际使用。本系列文章着眼于实际部署，带您快速掌握Kubernetes。本文为上篇，主要介绍部署之前需要了解的原理和概念，包括Kubernetes的组件结构，各个组件角色的功能，以及Kubernetes的应用模型等。

十多年来Google一直在生产环境中使用容器运行业务，负责管理其容器集群的系统就是Kubernetes的前身Borg。其实现在很多工作在Kubernetes项目上的Google开发者先前就在Borg这个项目上工作。多数Kubernetes的应用部署模型的思想都起源于Borg,了解这些模型是掌握Kubernetes的关键。Kubernetes的API版本目前是v1，本文以代码0.18.2版为基础来介绍它的应用部署模型，最后我们用一个简单的用例来说明部署过程。在部署结束后，阐述了它是如何用Iptables规则来实现各种类型Service的。

Kubernetes架构

Kubernetes集群包括**Kubernetes**代理**(agents)**和**Kubernetes**服务**(master node)**两种角色，代理角色的组件包括**Kube-proxy** 和**Kubelet**，它们同时部署在一个节点上，这个节点也就是代理节点。服务角色的组件包括**kube-apiserver**，**kube-scheduler**，**kube-controller-manager**，它们可以任意布属，它们可以部署在同一个节点上，也可以部署在不同的节点上（目前版本好像不行）。**Kubernetes**集群依赖的第三方组件目前有**etcd**和**docker**两个。前者提供状态存储，二者用来管理容器。集群还可以使用分布式存储给容器提供存储空间。下图显示了目前系统的组成部分：



Kubernetes代理节点

Kubelet和Kube-proxy运行在代理节点上。他们监听服务节点的信息来启动容器和实现Kubernetes网络和其它业务模型，比如Service、Pod等。当然每个代理节点都运行Docker。Docker负责下载容器

镜像和运行容器。

Kubelet

Kubelet组件管理**Pods**和它们的容器，镜像和卷等信息。

Kube-Proxy

Kube-proxy是一个简单的网络代理和负载均衡器。它具体实现**Service**模型，每个**Service**都会在所有的**Kube-proxy**节点上体现。根据**Service**的**selector**所覆盖的**Pods**, **Kube-proxy**会对这些**Pods**做负载均衡来服务于**Service**的访问者。

Kubernetes服务节点

Kubernetes服务组件形成了**Kubernetes**的控制平面，目前他们运行在单一节点上，但是将来会分开来部署，以支持高可用性。

etcd

所有的持久性状态都保存在etcd中。Etcd同时支持watch，这样组件很容易得到系统状态的变化，从而快速响应和协调工作。

Kubernetes API Server

这个组件提供对API的支持，响应REST操作，验证API模型和更新etcd中的相应对象。

Scheduler

通过访问Kubernetes中/binding API, Scheduler负责Pods在各个节点上的分配。Scheduler是插件式的，Kubernetes将来可以支持用户自定义的scheduler。

Kubernetes Controller Manager Server

Controller Manager Server负责所有其它的功能，比如endpoints控制器负责Endpoints对象的创建，更新。node控制器负责节点的发现，管理和监控。将来可能会把这些控制器拆分并且提供插件式的实现。

Kubernetes模型

Kubernetes的伟大之处就在于它的应用部署模型，主要包括Pod、Replication controller、Label和Service。

Pod

Kubernetes的最小部署单元是Pod而不是容器。作为First class API公民，Pods能被创建，调度和管理。简单地来说，像一个豌豆荚中的豌豆一样，一个Pod中的应用容器同享同一个上下文：

1. PID 名字空间。但是在docker中不支持
2. 网络名字空间，在同一Pod中的多个容器访问同一个IP和端口空间。
3. IPC名字空间，同一个Pod中的应用能够使用SystemV IPC和POSIX消息队列进行通信。
4. UTS名字空间，同一个Pod中的应用共享一个主机名。
5. Pod中的各个容器应用还可以访问Pod级别定义的共享卷。

从生命周期来说，Pod应该是短暂的而不是长久的应用。 Pods被调度到节点，保持在这个节点上直到被销毁。当节点死亡时，分配到这个节点的Pods将会被删掉。将来可能会实现Pod的迁移特性。在实际使用时，我们一般不直接创建Pods, 我们通过replication controller来负责Pods的创建，复制，监控和销毁。一个Pod可以包括多个容器，他们直接往往相互协作完成一个应用功能。

Replication controller

复制控制器确保Pod的一定数量的份数(replica)在运行。如果超过这个数量，控制器会杀死一些，如果少了，控制器会启动一些。控制器也会在节点失效、维护的时候来保证这个数量。所以强烈建议即使我们的份数是1，也要使用复制控制器，而不是直接创建Pod。

在生命周期上讲，复制控制器自己不会终止，但是跨度不会比Service强。Service能够横跨多个复制控制器管理的Pods。而且在一个Service的生命周期内，复制控制器能被删除和创建。Service和客户端程序是不知道复制控制器的存在的。

复制控制器创建的Pods应该是可以互相替换的和语义上相同的，这个对无状态服务特别合适。

Pod是临时性的对象，被创建和销毁，而且不会恢复。复制器动态地创建和销毁Pod。虽然Pod会分配到IP地址，但是这个IP地址都不是持久的。这样就产生了一个疑问：外部如何消费Pod提供的服务

呢？

Service

Service定义了一个Pod的逻辑集合和访问这个集合的策略。集合是通过定义Service时提供的Label选择器完成的。举个例子，我们假定有3个Pod的备份来完成一个图像处理的后端。这些后端备份逻辑上是相同的，前端不关心哪个后端在给它提供服务。虽然组成这个后端的实际Pod可能变化，前端客户端不会意识到这个变化，也不会跟踪后端。Service就是用来实现这种分离的抽象。

对于Service，我们还可以定义Endpoint，Endpoint把Service和Pod动态地连接起来。

Service Cluster IP和 kuber proxy

每个代理节点都运行了一个kube-proxy进程。这个进程从服务进程那边拿到Service和Endpoint对象的变化。 对每一个Service, 它在本地打开一个端口。 到这个端口的任意连接都会代理到后端Pod集合中的一个Pod IP和端口。在创建了服务后，服务Endpoint模型会体现后端Pod的 IP和端口列表，kube-proxy就是从这个endpoint维护的列表中选择服务后端的。另外Service对象的sessionAffinity属性也会帮助kube-proxy来选择哪个具体的后端。缺省情况下，后端Pod的选择是随机的。可以设置service.spec.sessionAffinity 成"ClientIP"来指定同一个ClientIP的流量代理到同一个后端。在实现上，kube-proxy会用IPtables规则把访问Service的Cluster IP和端口的流量重定向到这个本地端口。下面的部分会讲什么是service的Cluster IP。

注意：在0.18以前的版本中Cluster IP叫PortalNet IP。

内部使用者的服务发现

Kubernetes在一个集群内创建的对象或者在代理集群节点上发出访问的客户端我们称之为内部使用者。要把服务暴露给内部使用者，Kubernetes支持两种方式：环境变量和DNS。

环境变量

当kubelet在某个节点上启动一个Pod时，它会给这个Pod的容器为当前运行的Service设置一系列环境变量，这样Pod就可以访问这些Service了。一般地情况是{SVCNAME}_SERVICE_HOSTh和{SVCNAME}_SERVICE_PORT变量, 其中{SVCNAME}是Service名字变成大写，中划线变成下划线。比如Service "redis-master"，它的端口是 TCP 6379，分配到的Cluster IP地址是 10.0.0.11，kubelet可能会产生下面的变量给新创建的Pod容器：

```
REDIS_MASTER_SERVICE_HOST= 10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR= 10.0.0.11
```

注意，只有在某个Service后创建的Pod才会有这个Service的环境变量。

DNS

一个可选的Kubernetes附件（强烈建议用户使用）是DNS服务。它跟踪集群中Service对象，为每个Service对象创建DNS记录。这样所有的Pod就可以通过DNS访问服务了。

比如说我们在Kubernetes 名字空间"my-ns"中有个叫my-service的服务，DNS服务会创建一条"my-service.my-ns"的DNS记录。同在这个命名空间的Pod就可以通过"my-service"来得到这个Service分配到的Cluster IP，在其它命名空间的Pod则可以用全限定名"my-service.my-ns"来获得这个Service的地址。

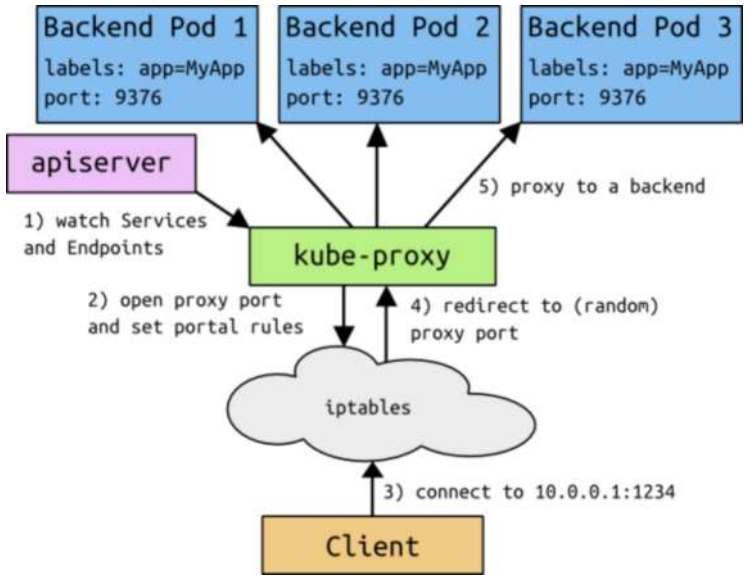
Pod IP and Service Cluster IP

Pod IP 地址是实际存在于某个网卡(可以是虚拟设备)上的，但Service Cluster IP就不一样了，没有网络设备为这个地址负责。它是由kube-proxy使用Iptables规则重新定向到其本地端口，再均衡到后端Pod的。我们前面说的Service环境变量和DNS都使用Service的Cluster IP和端口。

就拿上面我们提到的图像处理程序为例。当我们的Service被创建时，Kubernetes给它分配一个地址10.0.0.1。这个地址从我们启动API的service-cluster-ip-range参数(旧版本为portal_net参数)指定的地址池中分配，比如--service-cluster-ip-range=10.0.0.0/16。假设这个Service的端口是1234。集群内的所有kube-proxy都会注意到这个Service。当proxy发现一个新的service后，它会在本地节点打开一个任意端口，建相应的iptables规则，重定向服务的IP和port到这个新建的端口，开始接受到达这个服务的连接。

当一个客户端访问这个service时，这些iptable规则就开始起作用，客户端的流量被重定向到kube-

proxy为这个service打开的端口上，kube-proxy随机选择一个后端pod来服务客户。这个流程如下图所示：



根据Kubernetes的网络模型，使用Service Cluster IP和Port访问Service的客户端可以坐落在任意代理节点上。外部要访问Service，我们就需要给Service外部访问IP。

外部访问Service

Service对象在Cluster IP range池中分配到的IP只能在内部访问，如果服务作为一个应用程序内部的层次，还是很合适的。如果这个Service作为前端服务，准备为集群外的客户提供业务，我们就需要给这个服务提供公共IP了。

外部访问者是访问集群代理节点的访问者。为这些访问者提供服务，我们可以在定义Service时指定其spec.publicIPs，一般情况下publicIP 是代理节点的物理IP地址。和先前的Cluster IP range上分配到的虚拟的IP一样，kube-proxy同样会为这些publicIP提供Iptables 重定向规则，把流量转发到后端的Pod上。有了publicIP，我们就可以使用load balancer等常用的互联网技术来组织外部对服务的访问了。

spec.publicIPs在新的版本中标记为过时了，代替它的是spec.type=NodePort，这个类型的service，系统会给它在集群的各个代理节点上分配一个节点级别的端口，能访问到代理节点的客户端都能访问这个端口，从而访问到服务。

Label和Label selector

Label标签在**Kubernetes**模型中占着非常重要的作用。Label表现为key/value对，附加到Kubernetes管理的对象上，典型的就是Pods。它们定义了这些对象的识别属性，用来组织和选择这些对象。Label可以在对象创建时附加在对象上，也可以对象存在时通过API管理对象的Label。

在定义了对象的Label后，其它模型可以用Label 选择器（selector)来定义其作用的对象。

Label选择器有两种，分别是**Equality-based**和**Set-based**。

比如如下**Equality-based**选择器样例：

```
environment = production
tier != frontend
environment = production, tier != frontend
```

对于上面的选择器，第一条匹配**Label**具有**environment key**且等于**production**的对象，第二条匹配具有**tier key**，但是值不等于**frontend**的对象。由于**kubernetes**使用**AND**逻辑，第三条匹配**production**但不是**frontend**的对象。

Set-based选择器样例：

```
environment in (production, qa)
tier notin (frontend, backend)
partition
```

第一条选择具有**environment key**，而且值是**production**或者**qa**的**label**附加的对象。第二条选择具有**tier key**，但是其值不是**frontend**和**backend**。第三条选则具有**partition key**的对象，不对**value**进行校验。

replication controller复制控制器和**Service**都用**label**和**label selctor**来动态地配备作用对象。复制控制器在定义的时候就指定了其要创建**Pod**的**Label**和自己要匹配这个**Pod**的**selector**，**API**服务器应该校验这个定义。我们可以动态地修改**replication controller**创建的**Pod**的**Label**用于调式，数据恢复等。一旦某个**Pod**由于**Label**改变从**replication controller**移出来后，**replication controller**会马上启动一个新的**Pod**来确保复制池子中的份数。对于**Service**，**Label selector**可以用来选择一个**Service**的后端**Pods**。

下篇：Kubernetes应用部署模型解析（部署篇）（责编/周建丁）

作者简介：龚永生，九州云架构师。多年Linux系统开发，J2EE产品和云计算相关技术研发经验。目前活跃在OpenStack社区的各个项目上，主要技术方向是虚拟网络项目Neutron，是Neutron项目早期的主要贡献者之一。

已有0条评论