**Kubernetes**

## 一、基础知识

**1 Kubernetes是Google开源的容器集群管理系统**，其提供应用部署、维护、 扩展机制等功能，利用Kubernetes能方便地管理跨机器运行容器化的应用，是Docker分布式系统的解决方案。k8s里所有的资源都可以用yaml或Json定义。

**2 Master**

Master节点负责整个集群的控制和管理，所有的控制命令都是发给它，上面运行着一组关键进程：

**kube-apiserver：**提供了HTTP REST接口，是k8s所有资源增删改查等操作的唯一入口，也是集群控制的入口。

**kube-controller-manager**：所有资源的自动化控制中心。当集群状态与期望不同时，kcm会努力让集群恢复期望状态，比如：当一个pod死掉，kcm会努力新建一个pod来恢复对应replicas set期望的状态。

**kube-scheduler：**负责Pod的调度。

实际上，Master只是一个名义上的概念，三个关键的服务不一定需要运行在一个节点上。

**3 API Server的原理**

集群中的各个功能模块通过 apiserver将信息存储在Etcd，当需要修改这些信息的时候通过其REST接口来实现。

**4 Controller Manager的原理**

Replication Controller

Node Controller

ResourceQuota Controller

Namespace Controller

ServiceAccount Controller

Token Controller

Service Controller

Endpoint Controller等

这些Controller通过API Server实时监控各个资源的状态，当有资源因为故障导致状态变化，Controller就会尝试将系统由“现有状态”恢复到“期待状态”。

**5 Scheduler的原理**

作用是将apiserver或controller manager创建的Pod调度和绑定到具体的Node上，一旦绑定，就由Node上的kubelet接手Pod的接下来的生命周期管理。

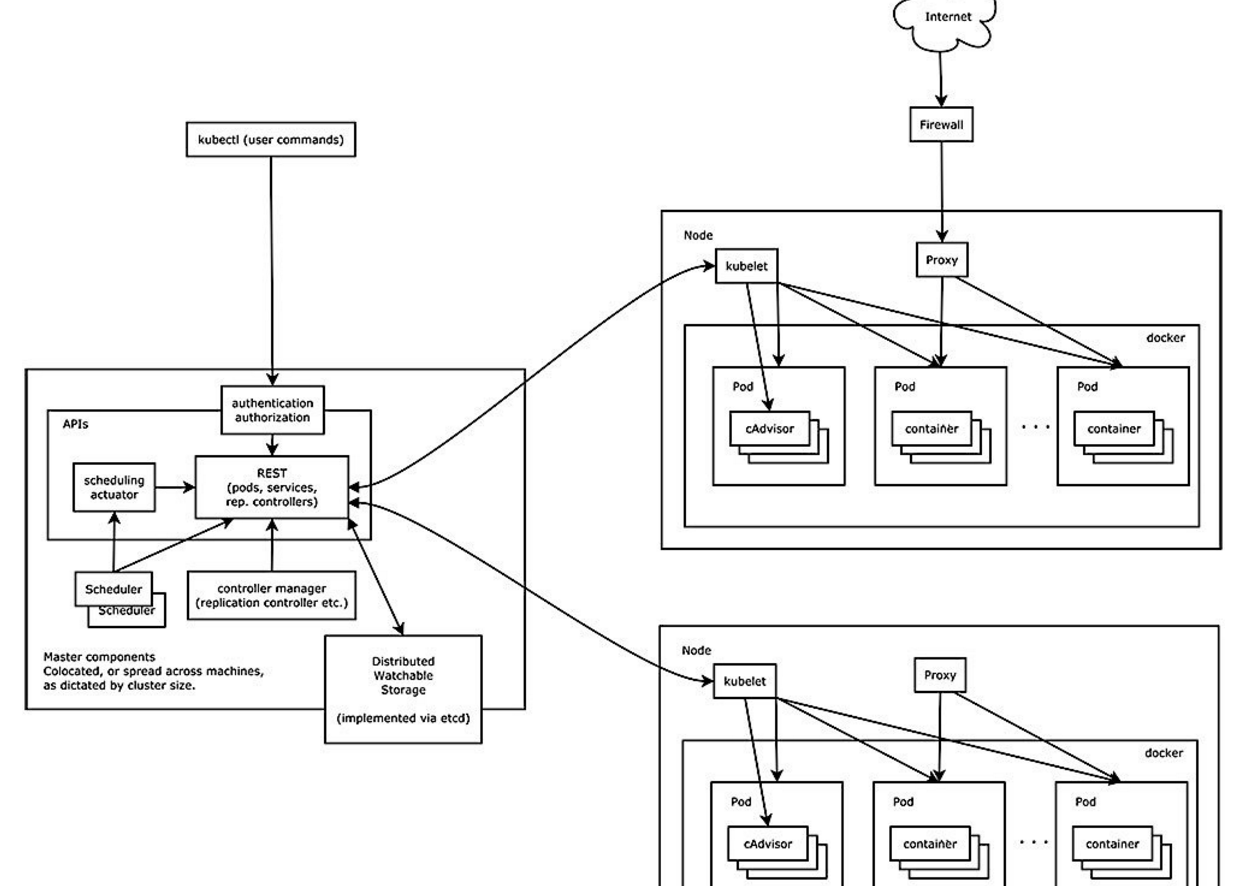
**6 Node**

Node是工作负载节点，运行着Master分配的负载（Pod），但一个Node宕机时，其上的负载会被自动转移到其他Node上。其上运行的关键组件是：

kubelet：负责Pod的生命周期管理，同时与Master密切协作，实现集群管理的基本功能。

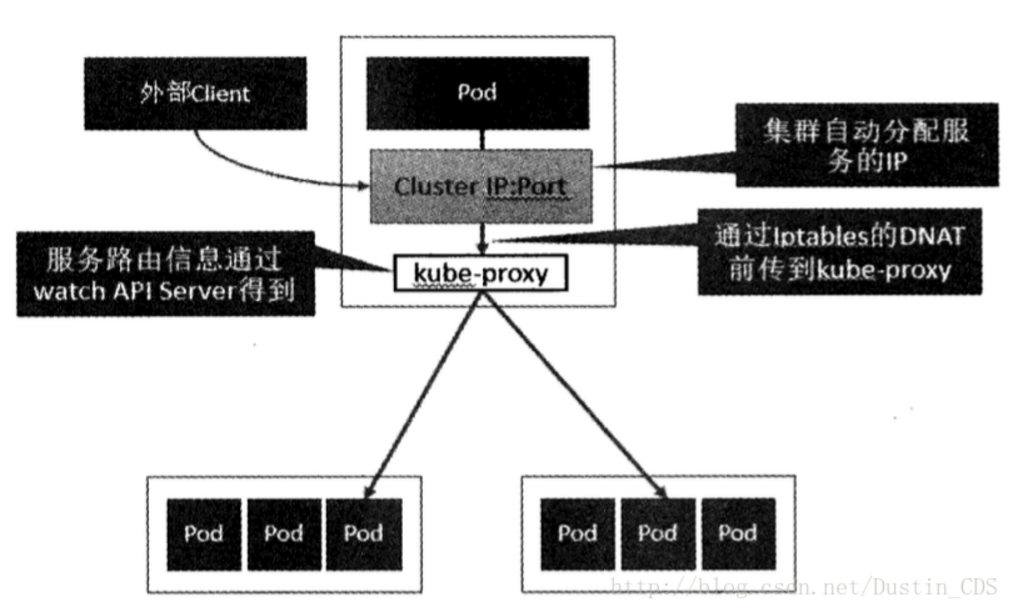
kube-proxy：实现Service的通信与负载均衡机制的重要组件，老版本主要通过设置iptables规则实现，新版1.9基于kube-proxy-lvs 实现。

Docker Engine：Docker引擎，负责Docker的生命周期管理。



**7 kube-proxy的原理**

每个Node上都运行着一个kube-proxy进程，它在本地建立一个SocketServer接收和转发请求，可以看作是Service的透明代理和负载均衡器，负载均衡策略模式是Round Robin。也可以设置会话保持，策略使用的是“ClientIP”，将同一个ClientIP的请求转发同一个Endpoint上。



Service的Cluster IP和NodePort等概念都是kube-proxy服务通过Iptables的NAT转换实现，Iptables机制针对的是kube-proxy监听的端口，所以每个Node上都要有kube-proxy。

**8 kubelet原理**

每个Node都会启动一个kubelet，主要作用有：

**（1）Node管理**

注册节点信息；

通过cAdvisor监控容器和节点的资源；

定期向Master（实际上是apiserver）汇报本节点资源消耗情况

**（2）Pod管理**

所以非通过apiserver方式创建的Pod叫Static Pod，这里我们讨论的都是通过apiserver创建的普通Pod。kubelet通过apiserver监听etcd，所有针对Pod的操作都会被监听到，如果其中有涉及到本节点的Pod，则按照要求进行创建、修改、删除等操作。

**（3）容器健康检查**

kubelet通过两类探针检查容器的状态：

LivenessProbe：判断一个容器是否健康，如果不健康则会删除这个容器，并按照restartPolicy看是否重启这个容器。实现的方式有ExecAction（在容器内部执行一个命令）、TCPSocketAction（如果端口可以被访问，则健康）、HttpGetAction（如果返回200则健康）。

ReadinessProbe：用于判断容器是否启动完全。如果返回的是失败，则Endpoint Controller会将这个Pod的Endpoint从Service的Endpoint列表中删除。也就是，不会有请求转发给它。

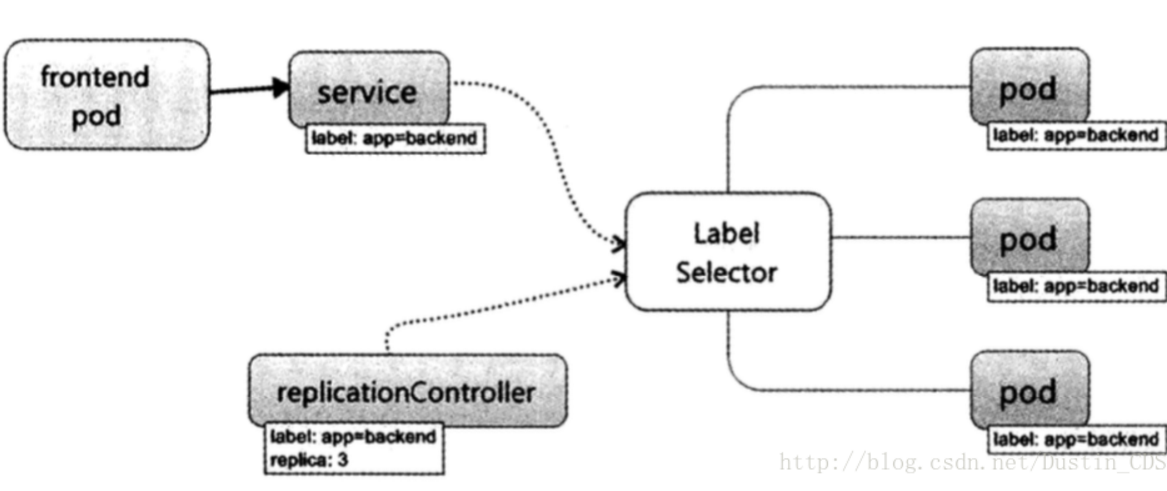
**9 Pod**

Pod是k8s进行资源调度的最小单位，每个Pod中运行着一个或多个密切相关的业务容器，这些业务容器共享这个Pause容器的IP和Volume，我们以这个不易死亡的Pause容器作为Pod的根容器，以它的状态表示整个容器组的状态。一个Pod一旦被创建就会放到Etcd中存储，然后由Master调度到一个Node绑定，由这个Node上的Kubelet进行实例化。

每个Pod会被分配一个单独的Pod IP，Pod IP + ContainerPort 组成了一个Endpoint。

**10 Service**

K8s中一个Service相当于一个微服务的概念，一个Service对应后端多个Pod计算实例，使用LabelSelector将一类Pod都绑定到自己上来。一般还会需要一个Deployment或者RC来帮助这个Service来保证这个Service的服务能力和质量。



**11 kube-proxy负载均衡**

运行在每个Node上的kube-proxy其实就是一个智能的软件负载均衡器，它负载将发给Service的请求转发到后端对应的Pod，也就是说它负责会话保持和负责均衡。

**12 Cluster IP**

负载均衡的基础是负载均衡器要维护一个后端Endpoint列表，但是Pod的Endpoint会随着Pod的销毁和重建而改变，k8s使这个问题透明化。一旦Service被创建，就会立刻分配给它一个Cluster IP，在Service的整个生命周期内，这个Cluster IP不会改变。于是，服务发现的问题也解决了：只要用Service Name和Service Cluster IP做一个DNS域名映射就可以了。

**13 DNS**

从Kubernetes 1.3开始，DNS通过使用插件管理系统cluster add-on，成为了一个内建的自启动服务。Kubernetes DNS在Kubernetes集群上调度了一个DNS Pod和Service，并配置kubelet，使其告诉每个容器使用DNS Service的IP来解析DNS名称。

**（1）Service**

集群中定义的每个Service（包括DNS Service它自己）都被分配了一个DNS名称。默认的，Pod的DNS搜索列表中会包含Pod自己的命名空间和集群的默认域，下面我们用示例来解释以下。 假设有一个名为foo的Service，位于命名空间bar中。运行在bar命名空间中的Pod可以通过DNS查找foo关键字来查找到这个服务，而运行在命名空间quux中的Pod可以通过关键字foo.bar来查找到这个服务。

普通（非headless）的Service都被分配了一个DNS记录，该记录的名称格式为my-svc.my-namespace.svc.cluster.local，通过该记录可以解析出服务的集群IP。 Headless（没有集群IP）的Service也被分配了一个DNS记录，名称格式为my-svc.my-namespace.svc.cluster.local。与普通Service不同的是，它会解析出Service选择的Pod的IP列表。

**（2）Pod**

Pod也可以使用DNS服务。pod会被分配一个DNS记录，名称格式为pod-ip-address.my-namespace.pod.cluster.local。 比如，一个pod，它的IP地址为1.2.3.4，命名空间为default，DNS名称为cluster.local，那么它的记录就是：1-2-3-4.default.pod.cluster.local。 当pod被创建时，它的hostname设置在Pod的metadata.name中。

在v1.2版本中，用户可以指定一个Pod注解，pod.beta.kubernetes.io/hostname，用于指定Pod的hostname。这个Pod注解，一旦被指定，就将优先于Pod的名称，成为pod的hostname。比如，一个Pod，其注解为pod.beta.kubernetes.io/hostname: my-pod-name，那么该Pod的hostname会被设置为my-pod-name。 v1.2中还引入了一个beta特性，用户指定Pod注解，pod.beta.kubernetes.io/subdomain，来指定Pod的subdomain。比如，一个Pod，其hostname注解设置为“foo”，subdomain注解为“bar”，命名空间为“my-namespace”，那么它最终的FQDN就是“foo.bar.my-namespace.svc.cluster.local”。 在v1.3版本中，PodSpec有了hostname和subdomain字段，用于指定Pod的hostname和subdomain。它的优先级则高于上面提到的pod.beta.kubernetes.io/hostname和pod.beta.kubernetes.io/subdomain。

**14 外部访问Service的问题**

先明确这样几个IP：

Node IP：Node主机的IP，与它是否属于K8s无关。

Pod IP：是Dokcer Engine通过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。k8s中一个Pod访问另一个Pod就是通过Pod IP。

Cluster IP：仅用于Service对象，属于k8s的内部IP，外界无法直接访问。

**（1）NodePort**

在Service的yaml中定义NodePort，k8s为集群中每个Node都增加对这个端口的监听，使用这种方式往往需要一个独立与k8s之外的负载均衡器作为流量的入口。

**（2）使用External IP**

运行Hello World应用程序的五个实例。

创建一个暴露外部IP地址的Service对象。

使用Service对象访问正在运行的应用程序。

使用deployment创建暴露的Service对象：

~ kubectl expose deployment hello-world --type=LoadBalancer --name=my-service

显示关于Service的信息：

~ kubectl get services my-service

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE

my-service 10.3.245.137 104.198.205.71 8080/TCP 54s

~ kubectl describe services my-service

Name: my-service

Namespace: default

Labels: run=load-balancer-example

Selector: run=load-balancer-example

Type: LoadBalancer

IP: 10.3.245.137

LoadBalancer Ingress: 104.198.205.71

Port: <unset> 8080/TCP

NodePort: <unset> 32377/TCP

Endpoints: 10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more...

Session Affinity: None

Events:

在此例子中，外部IP地址为104.198.205.71。还要注意Port的值。在这个例子中，端口是8080。在上面的输出中，您可以看到该服务有多个端点：10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more…。这些是运行Hello World应用程序的pod的内部地址。

使用外部IP地址访问Hello World应用程序：

~ curl http://<external-ip>:<port>

Hello Kubernetes!

删除服务

~ kubectl delete services my-service

~ kubectl delete deployment hello-world

**15 Ingress**

通常情况下，service和pod仅可在集群内部网络中通过IP地址访问。所有到达边界路由器的流量或被丢弃或被转发到其他地方。Ingress是授权入站连接到达集群服务的规则集合。你可以给Ingress配置提供外部可访问的URL、负载均衡、SSL、基于名称的虚拟主机等。用户通过POST Ingress资源到API server的方式来请求ingress。 Ingress controller负责实现Ingress，通常使用负载平衡器，它还可以配置边界路由和其他前端，这有助于以HA方式处理流量。

最简化的Ingress配置：

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: test-ingress

spec:

rules:

- http:

paths:

- path: /testpath

backend:

serviceName: test

servicePort: 80

- path: /bar

backend:

serviceName: s2

servicePort: 80

1-4行：跟Kubernetes的其他配置一样，ingress的配置也需要apiVersion，kind和metadata字段。配置文件的详细说明请查看部署应用, 配置容器和 使用resources.

5-7行: Ingress spec 中包含配置一个loadbalancer或proxy server的所有信息。最重要的是，它包含了一个匹配所有入站请求的规则列表。目前ingress只支持http规则。

8-9行：每条http规则包含以下信息：一个host配置项（比如for.bar.com，在这个例子中默认是\*），path列表（比如：/testpath），每个path都关联一个backend(比如test:80)。在loadbalancer将流量转发到backend之前，所有的入站请求都要先匹配host和path。

10-12行：backend是一个service:port的组合。Ingress的流量被转发到它所匹配的backend。

配置TLS证书

你可以通过指定包含TLS私钥和证书的secret来加密Ingress。 目前，Ingress仅支持单个TLS端口443，并假定TLS termination。 如果Ingress中的TLS配置部分指定了不同的主机，则它们将根据通过SNI TLS扩展指定的主机名（假如Ingress controller支持SNI）在多个相同端口上进行复用。 TLS secret中必须包含名为tls.crt和tls.key的密钥，这里面包含了用于TLS的证书和私钥，例如：

（1）创建Secret

apiVersion: v1

data:

tls.crt: base64 encoded cert

tls.key: base64 encoded key

kind: Secret

metadata:

name: testsecret

namespace: default

type: Opaque

（2）创建Ingress：

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: no-rules-map

spec:

tls:

- secretName: testsecret

backend:

serviceName: s1

servicePort: 80

## 二、高深知识

**1 高可用**

Kubernetes服务本身的稳定运行对集群管理至关重要，影响服务稳定的因素一般来说分为两种,一种是服务本身异常或者服务所在机器宕机，另一种是因为网络问题导致的服务不可用。本文将从存储层、管理层、接入层三个方面介绍高可用Kubernetes集群的原理。

**2 Etcd高可用方案**

Kubernetes的存储层使用的是Etcd。Etcd是CoreOS开源的一个高可用强一致性的分布式存储服务，Kubernetes使用Etcd作为数据存储后端，把需要记录的pod、rc、service等资源信息存储在Etcd中。

Etcd使用raft算法将一组主机组成集群，raft 集群中的每个节点都可以根据集群运行的情况在三种状态间切换：follower, candidate 与 leader。leader 和 follower 之间保持心跳。如果follower在一段时间内没有收到来自leader的心跳，就会转为candidate，发出新的选主请求。集群初始化的时候内部的节点都是follower节点，之后会有一个节点因为没有收到leader的心跳转为candidate节点，发起选主请求。当这个节点获得了大于一半节点的投票后会转为leader节点。当leader节点服务异常后，其中的某个follower节点因为没有收到leader的心跳转为candidate节点，发起选主请求。只要集群中剩余的正常节点数目大于集群内主机数目的一半，Etcd集群就可以正常对外提供服务。

当集群内部的网络出现故障集群可能会出现“脑裂”问题，这个时候集群会分为一大一小两个集群（奇数节点的集群），较小的集群会处于异常状态，较大的集群可以正常对外提供服务。

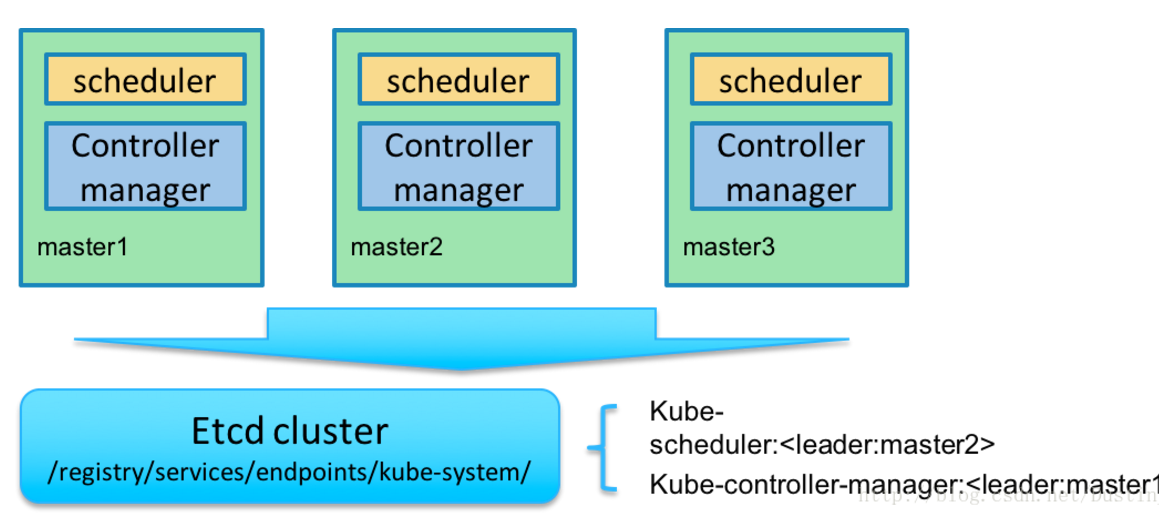
**3 Master高可用方案**

Master上有三个关键的服务：apiserver、controller-manager和scheduler，这三个不一定要运行在一台主机上。

**4 controller-manager和scheduler的选举配置**

Kubernetes的管理层服务包括kube-scheduler和kube-controller-manager。kube-scheduer和kube-controller-manager使用一主多从的高可用方案，在同一时刻只允许一个服务处以具体的任务。Kubernetes中实现了一套简单的选主逻辑，依赖Etcd实现scheduler和controller-manager的选主功能。

如果scheduler和controller-manager在启动的时候设置了leader-elect参数，它们在启动后会先尝试获取leader节点身份，只有在获取leader节点身份后才可以执行具体的业务逻辑。它们分别会在Etcd中创建kube-scheduler和kube-controller-manager的endpoint，endpoint的信息中记录了当前的leader节点信息，以及记录的上次更新时间。leader节点会定期更新endpoint的信息，维护自己的leader身份。每个从节点的服务都会定期检查endpoint的信息，如果endpoint的信息在时间范围内没有更新，它们会尝试更新自己为leader节点。



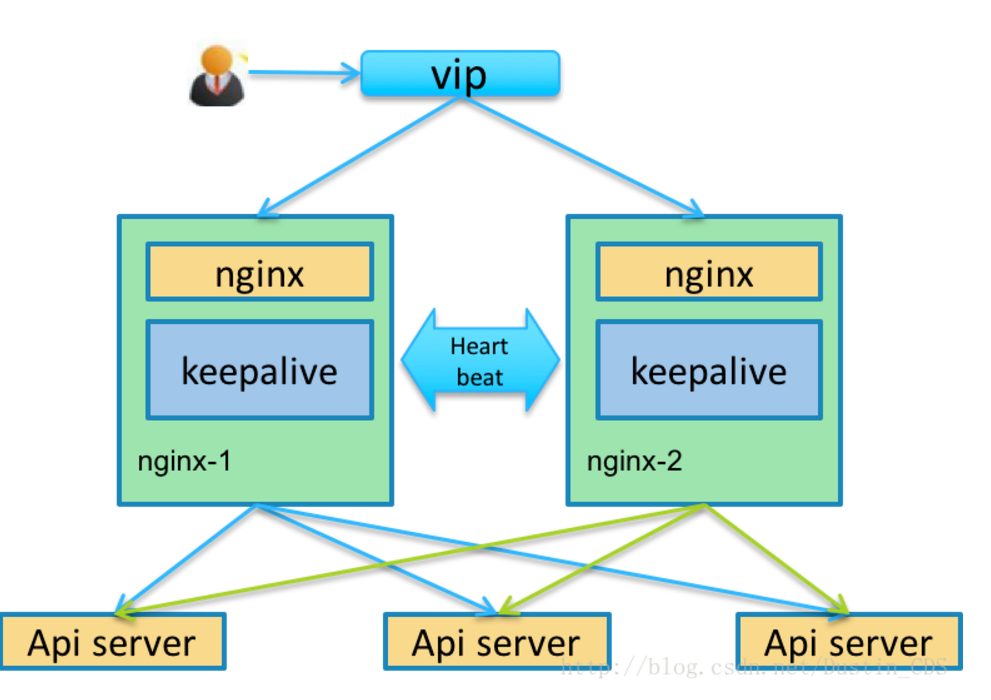
scheduler服务以及controller-manager服务之间不会进行通信，利用Etcd的强一致性，能够保证在分布式高并发情况下leader节点的全局唯一性。整体方案如下图所示：

当集群中的leader节点服务异常后，其它节点的服务会尝试更新自身为leader节点，当有多个节点同时更新endpoint时，由Etcd保证只有一个服务的更新请求能够成功。通过这种机制sheduler和controller-manager可以保证在leader节点宕机后其它的节点可以顺利选主，保证服务故障后快速恢复。当集群中的网络出现故障时对服务的选主影响不是很大，因为scheduler和controller-manager是依赖Etcd进行选主的，在网络故障后，可以和Etcd通信的主机依然可以按照之前的逻辑进行选主，就算集群被切分，Etcd也可以保证同一时刻只有一个节点的服务处于leader状态。

**5 apiserver的高可用**

Kubernetes的接入层服务主要是kube-apiserver。apiserver本身是无状态的服务，它的主要任务职责是把资源数据存储到Etcd中，后续具体的业务逻辑是由scheduler和controller-manager执行的。所以可以同时起多个apiserver服务，使用nginx把客户端的流量转发到不同的后端apiserver上实现接入层的高可用。具体的实现如下图所示：

接入层的高可用分为两个部分，一个部分是多活的apiserver服务，另一个部分是一主一备的nginx服务。



**6 Keepalived简介**

Keepalived软件起初是专为LVS负载均衡软件设计的，用来管理并监控LVS集群系统中各个服务节点的状态，后来又加入了可以实现高可用的VRRP功能。因此，Keepalived除了能够管理LVS软件外，还可以作为其他服务（例如：Nginx、Haproxy、MySQL等）的高可用解决方案软件。Keepalived软件主要是通过VRRP协议实现高可用功能的。VRRP是Virtual Router RedundancyProtocol(虚拟路由器冗余协议）的缩写，VRRP出现的目的就是为了解决静态路由单点故障问题的，它能够保证当个别节点宕机时，整个网络可以不间断地运行。所以，Keepalived 一方面具有配置管理LVS的功能，同时还具有对LVS下面节点进行健康检查的功能，另一方面也可实现系统网络服务的高可用功能。

**7 故障切换转移原理**

Keepalived高可用服务对之间的故障切换转移，是通过 VRRP (Virtual Router Redundancy Protocol ,虚拟路由器冗余协议）来实现的。在 Keepalived服务正常工作时，主 Master节点会不断地向备节点发送（多播的方式）心跳消息，用以告诉备Backup节点自己还活看，当主 Master节点发生故障时，就无法发送心跳消息，备节点也就因此无法继续检测到来自主 Master节点的心跳了，于是调用自身的接管程序，接管主Master节点的 IP资源及服务。而当主 Master节点恢复时，备Backup节点又会释放主节点故障时自身接管的IP资源及服务，恢复到原来的备用角色。

**8 docker默认容器网络**

在默认情况下会看到三个网络，它们是Docker Deamon进程创建的。它们实际上分别对应了Docker过去的三种『网络模式』，可以使用docker network ls来查看：

master@ubuntu:~$ sudo docker network ls

NETWORK ID NAME DRIVER SCOPE

18d934794c74 bridge bridge local

f7a7b763f013 host host local

697354257ae3 none null local

这 3 个网络包含在 Docker 实现中。运行一个容器时，可以使用 the –net标志指定您希望在哪个网络上运行该容器。您仍然可以使用这 3 个网络。

bridge 网络表示所有 Docker 安装中都存在的 docker0 网络。除非使用 docker run –net=选项另行指定，否则 Docker 守护进程默认情况下会将容器连接到此网络。在主机上使用 ifconfig命令，可以看到此网桥是主机的网络堆栈的一部分。

none 网络在一个特定于容器的网络堆栈上添加了一个容器。该容器缺少网络接口。

host 网络在主机网络堆栈上添加一个容器。您可以发现，容器中的网络配置与主机相同。

**9 跨主机通信的方案**

和host共享network namespace

这种接入模式下，不会为容器创建网络协议栈，即容器没有独立于host的network namespace，但是容器的其他namespace（如IPC、PID、Mount等）还是和host的namespace独立的。容器中的进程处于host的网络环境中，与host共用L2-L4的网络资源。该方式的优点是，容器能够直接使用host的网络资源与外界进行通信，没有额外的开销（如NAT），缺点是网络的隔离性差，容器和host所使用的端口号经常会发生冲突。

和host共享物理网卡

2与1的区别在于，容器和host共享物理网卡，但容器拥有独立于host的network namespace，容器有自己的MAC地址、IP地址、端口号。这种接入方式主要使用SR-IOV技术，每个容器被分配一个VF，直接通过PCIe网卡与外界通信，优点是旁路了host kernel不占任何计算资源，而且IO速度较快，缺点是VF数量有限且对容器迁移的支持不足。

Behind the POD

这种方式是Google在Kubernetes中的设计中提出来的。Kubernetes中，POD是指一个可以被创建、销毁、调度、管理的最小的部署单元，一个POD有一个基础容器以及一个或一组应用容器，基础容器对应一个独立的network namespace并拥有一个其它POD可见的IP地址（以IP A.B.C.D指代），应用容器间则共享基础容器的network namespace（包括MAC、IP以及端口号等），还可以共享基础容器的其它的namespace（如IPC、PID、Mount等）。POD作为一个整体连接在host的vbridge/vswitch上，使用IP地址A.B.C.D与其它POD进行通信，不同host中的POD处于不同的subnet中，同一host中的不同POD处于同一subnet中。这种方式的优点是一些业务上密切相关的容器可以共享POD的全部资源（它们一般不会产生资源上的冲突），而这些容器间的通信高效便利。