再次了解 kubernetes 的基本概念和术语

1、kubernetes 整体概要

在 Kubernetes 中，Service（服务）是分布式集群架构的一个核心，一个 Service 对象拥有如下特征。

- 拥有一个唯一指定的名字（如 mysql-server）。
- 拥有一个虚拟 IP（Cluster IP、Service IP 或 VIP）。
- 能够提供某个远程服务的能力。
- 被印射到了提供这种服务能力的一组容器应用之上。

Service 的服务进程目前都是居于 Socker 通信方式对外提供服务，如 Redis、Memcache、Mysql、Web Service，或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 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 上面。

kubernetes 内建的透明负载均衡、故障恢复机制及 service 创建后 ip 不变化，基于此可以向用户提供稳定的服务。

容器提供了强大的隔离功能，所以有必要把 Service 提供服务的这组进程放到容器中进行隔离。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 间的关联联系，kubernetes 首先给每个 Pod 贴上一个标签（Label），如给运行 mysql 的 Pod 贴上 name=mysql 的标签，然后给相应的 Service 定义标签选择器（Label Selector），如 mysql 的 Service 的标签选择器的选择条件为 name=mysql，意为 Service 下所有包含 name-=mysql 的 Label 的 Pod 上。

说到 Pod，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，这个节点可能是物理机或虚拟机，通常一个节点上上面运行几百个 Pod；其次每个 Pod 运行着一个特殊的被称之为 Pause 的容器，其它的容器被称为业务容器，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效率更为高效。**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到同一个 Pod 中。最后需要注意的是并不是每一个 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 Master 节点运行着集群管理的一组进程 `kube-apiserver`、`kube-controller-manage`、`kube-scheduler`。这些进程实现了整个集群的资源管理、Pod 调度、弹性收缩、安全控制、系统控制和纠错等管理功能，并且全是自动完成的。Node 作为一个集群中的工作节点，运行真正的应用程序，在 Node 上 kubernetes 管理的最小单元是 Pod。Node 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、重启和销毁，以及实现软件模式的均衡负载。**

解决传统的服务扩容和服务升级两个难题里，**kubernetes 集群中，我们只需要为扩容的 service 关联的 Pod 创建一个 `Replication Controller`（简称 RC）**，在这个 RC 定义文件中写好下面几个关键信息：

- 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 实例并全自动实时监控其状态和数量。

pod - pod 位置示意

2、基本概念和术语

kubernetes 中大部分概念如 `Node`、`Pod`、`Replication Controller`、`Service` 等都可以被看作一种资源对象，几乎所有的资源对象都可以通过 kubernetes 提供的 `kubectl` 工具（或 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，它通过跟踪对比保存在 etcd 库里的"资源期望状态"和当前环境中的"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。

2.1 Master

kubernetes 里的 Master 指的是集群控制节点。每个 kubernetes 集群里都需要一个 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，我们后面所有的执行的命令基本上都是在 Master 节点上运行的。Master 节点通常会占据一个独立的服务器或虚拟机，就是它的重要性体现，一个集群的大脑，如果它宕机，那么整个集群将无法响应控制命令。

Master 节点上运行着以下几组关键进程：

- Kubernetes API Server（kube-apiserver）：提供了 HTTP rest 接口的关进服务进程，是 Kubernetes 里所有资源增删改查等操作的唯一入口，也是集群管理的入口进程。
- Kubernetes Controller Manage（kube-controller-manage）：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）：负责资源的调度（Pod 调度）的进程。

Master 还启动了一个 etcd server 进程，因为 kubernetes 里所有的资源对象的数据都是保存在 etcd 中的。

2.2 Node

除了 Master，kubernetes 集群的其它机器也被称为 Node 节点，在较早的版本也被称为 Monion。同样的它也是一台物理机或是虚拟机。Node 节点才是 Kubernetes 集群中工作负载节点，每个 Node 都会被 Master 分配一些工作负载（Docker 容器），当某个 Node 宕机之后，其上的工作负载会被 Master 自动转移到其它节点上面去。

每个 Node 节点运行着以下几个关键进程：

- kubelet：负责 pod 对应的同期创建、启动停止等任务，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 的通信与负载均衡的重要组件。
- Docker-Engine（docker）：docker 引擎，负责本机容器的创建和管理工作。

Node 节点可以在运行期间动态增加到 Kubernetes 集群中，前提是这个节点已经正确安装、配置和启动的上述关键进程。在默认情况下 kubenet 会向 Master 注册自己，这也是 kubernetes 推荐的管理方式。一旦 Node 被纳入集群管理范围，kubelet 进程就会定时向 Master 节点汇报自身的情况，包括操作系统、Docker 版本、机器 cpu 和内存情况及有哪些 pod 在运行。这样 Master 可以获知 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 超过时间不进行上报信息时,master 将会将其标记为失联（Not Ready），随后会触发资源转移的自动流程。

2.3 Pod

Pod 是 Kubernets 最重要也是最基础的概念，其由下图示意组成：

pod 的组成与容器的关系.png

需求推动发展，下面解释为何会有 pod 的设计和其如此特殊的结构。

- 一组容器作为单元的情况下，难以去对"整体"进行简单判断并采取有效的行动。如一个容器挂了，是算整体服务挂了，还是算 N/M 的死亡率。引入业务无关并且不易挂掉的 `Pause` 容器作为 Pod 的根容器，以它的状态作为整体容器组的状态，就简单巧妙的解决了这个问题。
- Pod 多个业务容器共享 Pause 容器的 IP，共享 Pause 容器挂接的 Volume 卷。这样既简化了密切关联业务容器之间的通信问题，还很好地解决了它们之间文件共享的问题。

k8s 为每个 Pod 都都分配了唯一的 ip，称之为 Pod IP，一个 Pod 里的多个容器共享 Pod IP 地址。**k8s 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，如 Flannel、Openswith 等，牢记在 k8s 中，一个 Pod 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 及静态的 Pod（static Pod），后者比较特殊，并不存放在 k8s 的 etcd 里面存储，而是存放在某一个具体的 Node 上的一个文件中，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 k8s Master 调度到某个具体的 Node 上面进行绑定（Binding），随后该 Pod 被对应的 Node 上的 kubblet 进程实例化成一组的 Docker 容器并启动起来。默认情况下，当 Pod 里的某个容器停止时，k8s 会自动检测这个问题并重启启动这个 Pod（Pod 里面的所有容器），如果 Pod 所在的 Node 宕机，则会将这个 Node 上的所有 Pod 重新调度到其它节点上。Pod、容器与 Node 的关系如下：

pod、node 和容器之间的关系.png

k8s 里的所有的资源对象都可以采用 yaml 或 JSON 格式的文件定义或描述，下面是某个 Pod 文件的资源定义文件。

```
apiSerdion: v1

kind: Pod

matadata:

  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；matadata 的 name 属性为 Pod 的名字，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。Pod 里面所包含的容器组的定义则在 spec 中声明，这里定义一个名字为 myweb，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）。Pod 的 IP 加上这里的容器端口（containerPort）就组成了一个新的概念 Endpoint（端口），它代表 Pod 的一个服务进程对外的通信地址。一个 Pod 也存在着具有多个 Endpoint 的情况，比如 tomcat 定义 pod 时候，可以暴露管理端口和服务端口两个 Endpoint。

我们所熟悉的 Docker Volume 在 k8s 中也有相应的概念 - Pod Volume，后者有一些幻化，比如可以用分布式系统 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中的。

最后是 k8s 中 Event 的概念，Event 是一个事件的记录，记录了时间的最早产生时间、最后重现时间、重复次数、发起者、类型及导致此次事件的原因等总舵信息。Event 通常会关联到某个具体的资源对象上，是排查故障的重要信息参考信息。Node 描述信息包含 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 name` 来查看它的信息，用来定位问题的原因。

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，当前可以设置的限额的计算资源有 CPU 和 Memory 两种，其中 CPU 的资源单位为 CPU（core）核数，一个绝对值。

一个 cpu 的配额对绝大多数容器去是一个相当大的资源配额，所以在 k8s 中，**通常以千分之一的 cpu 配额为最小单位，用 `m` 来表示，即一个容器的限额为 100-300m，即占用 0.1-0.3 个 cpu 限额。**由于它是绝对值，所以无论在 1 个 cpu 的机器还是多个 cpu 的机器，这个 100m 的配额都是一样的。类似的 Memory 配额也是一个绝对值，单位是内存字节数。

在 k8s 中，一个计算资源进行配额限定需要设定下面两个参数：

- Request：该资源的最小申请量，系统必须满足要求。
- Limits：该资源允许最大的使用量，不能被突破，当容器视图使用超出这个量的资源时，可能会被 k8s 杀死并重启。

通常会把 Request 设置为一个比较小的数值，符合同期平时的工作负载情况下的资源要求，而把 Limit 设置为峰值负载情况下资源占用的最大量。如下面的设定：

```
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

即表明 mysql 容器在申请最少 0.25 个 cpu 及 64Mib 的内存，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 的内存。

pod 及周边对象.png

2.4 Label（标签）

Label 是 k8s 系统中另一个核心的概念。一个 Label 是一个 key=value 的键值对，其中 key 和 value 由用户自己指定。Label 对象可以附加到各种资源对象上面，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资源对象上去，Label 通常在资源对象定义时确定，也可以通过对象创建之后动态添加或删除。

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、配置和布署等管理工作。一些常用的 label 如下：

- 版本控制："release": "stable", "release": "canary"
- 环境标签："environment": "dev", "environment": "qa", "environment": "production"
- 架构标签："tier": "frontend", "release": "backend", "release": "middkeware"
- 分区标签："partition": "customerA"
- 质量管控标签："track": "daily", "track": "weekly"

给某个资源对象定义一个 Label，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，k8s 通过这种方式实现了类是 SQL 的简单通用的对象查询方式。

当前有两种 `Label Selector` 的表达式：基于等式的（Equality-based）和基于集合（Set-based）。

基于等式的表达式匹配标签实例：

- name=redis-slave：匹配所有具有标签 name=redis-slave 的资源对象。
- env!=production： 匹配不具有标签 name=production 的资源对象。

基于集合方式的表达式匹配标签实例：

- name in (redis-slave, redis-master)：匹配所有具有标签 name=redis-slave 或 name=redis-master 的资源对象。
- name not in (php-frontend)：匹配不具有标签 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“,”分隔即可，几个条件是 and 的关系，即同时满足多个条件，如：

- name=redis-slave, env!=production
- name not in (php-frontend), env!=production

Label Selector 在 k8s 中重要使用场景有下面几处：

- kube-controller-manage 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。
- kube-proxy 进程通过 Service 的 Label Selector 来选择对于那个的 Pod，自动建立起 Service 对应每个 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 Pod 定义文件中使用 NodeSelector 这种标签调度策略，kube-schedule 进程可以实现 Pod"定向调度"的特性。

总结：使用 Label 可以给资源对象创建多组标签，Label 和 Label Selector 共同构成了 k8s 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。

2.5 Replication Controller（RC）

前面部分对 Replication Controller（简称 RC）的定义和作用做了一些说明，本节对 RC 的概念再进行深入研究下。

RC 是 k8s 系统中的核心概念之一，简单来说，它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 的定义包含如下几个部分：

- 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 模版（templates）。

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）在整个集群中始终只有一个副本。

```
apiVersion: v1

kind: ReplicationController

metadata:

  name: frontend

spec:

  replicas: 1

  selector:

    tier: frontend

  templete:

    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 k8s 集群中以后，Master 节点上的 Controller Manager 组件就得到了通知，定期巡检系统中当前存活目标的 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 副本在进行，系统就会停掉一些 Pod，否则系统会自动创建一些 Pod。可以说，通过 RC，k8s 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环境中完成许多的运维工作（主机监控、应用监控、故障恢复）。

需要注意的有几点：**删除 RC 并不会影响已经通过该 RC 创建好的 Pod。为了删除所有的 Pod，可以设置 replicas 为 0，然后更新 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代替旧的版本方式达到目的。在系统升级的过程中，我们希望是平滑的方式，如系统中 10 个对应旧版本的 Pod，最佳的方式就是就旧版本的 Pod 每次停止一个，同时创建一个新版本的 Pod，在整个升级的过程中，此消彼长，而运行中的 Pod 始终是 10 个，几分钟后，当所有的 Pod 都是新版本之后，升级过程完成。通过 RC 的机制，k8s 很容易就实现这种高级实用的特性，被称为**“滚动升级”（Rolling Update）**。

由于 Replication Controller 与 k8s 代码中的模块 Replication Controller 同名，同时这个词又无法准确表达它的本意，所以在 k8s1.2 的时候，它就升级成了另外一个新的概念 - Replica Set，官网解释为“下一代的 RC”。与当前 RC 区别在于：Replica Sets 支持基于集合的 Label Selector（Set-based selector），而当前 RC 只支持基于等式的 Label Selector（equality-based selector）。

当前我们很少使用 Replica Set，它主要被 Deployment 这个更高层的资源对象使用，从而形成一整套 Pod 的创建、删除、更新的编排机制。当我们使用 Deployment 时，无需关系它是如何创建和维护 Replica Set，都是自动完成的。

Replica Set 与 Deployment 这两个重要的资源对象逐步替换了之前 RC 的作用，是 k8s1.3 Pod 自动扩容（伸缩）这个告警功能实现的基础。

最后**总结 RC（Replica Set）的的一些特性与作用：**

- 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，可以实现 Pod 的弹性收缩。
- RC 里包括完整的 Pod 定义模版。
- RC 通过 Label Selector 机制实现对 Pod 的自动控制。
- **通过改变 RC 里 Pod 模版的镜像版本，可以实现 Pod 的滚动升级功能。**

2.6 Deployment

Deployment 是 k8s 1.2 之后引入的新概念，引入的目的就是为了更好的解决 Pod 的编排问题。为此，Deployment 在内部使用了 Replica Set 来实现目的，可以看作是 RC 的一次升级。

Deployment 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 的状态来看布署动作是否完成（Pod 副本的数量是否达到预期值）。
- 更新 Deployment 以创建新的 Pod（如镜像升级）。
- 如果当前 Deployment 不稳定，则回滚到早先的 Deployment 版本。
- 挂起或恢复一个 Deployment。

下面是一个实例，文件名为 tomcat-deployment.yaml：

```
apiVersion:  extension/v1beta1

kind: Deployment

matadata:

  name: frontend

spec:

  replica: 1

  selector:

    matchLabels:
```

```
    tier: frontend
  matchExpressions:
    - {key: tier, operator: In, values: [frontend]}
template:
  matadata:
    labels:
      app: app-demo
      tier: frontend
    spec:
      containers:
      - name: tomcat-demo
        image: tomcat
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

命令 `kubectl create -f tomcat-deployment.yaml`，命令以后再尝试。

2.7 Horizontal Pod Autoscaler（HPA）

之前提到过，通过手工执行 `kubectl scale` 的命令，我们可以实现 Pod 扩容或缩容，但谷歌对 k8s 的定位是自动化、智能化，即分布式系统要能根据当前负载情况变化自动触发水平扩展或缩容的行为，因为这过程是频繁发生、不可预料的。

k8s 1.1 版本中首次发布这一特性：Horizontal Pod Autoscaling，随后在 1
2 版本中 HPA 被升级为稳定版本（apiVersion: autoscaling/v1），但同时仍保留旧版本（apiVersion: extension/v1beta1），官方计划是 1.3 移除旧版本，1.4 彻底移除旧版本的支持。

Horizontal Pod Autoscaling（HPA），意味 Pod 的横向自动扩容，与之前 RC、Deployment 一样，也是 k8s 资源对象的一种。通过分析 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，即 HPA 的实现原理。当前有下两种方式作为 Pod 负载的度量指标。

- CPU Utilization Percentage。
- 应用程序自定义度量指标，如服务每秒内相应的请求数（TPS 或 QPS）。

CPU Utilization Percentage 是一个算术平均值，即目标 Pod 所有副本的 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 Resquest 的值。如我们定义一个 Pod 的 Pod Resquest 为 0.4，而当前 Pod 的 CPU 使用量是 0.2，即它的 CPU 使用率是 50%，如此我们就可以算出一个 RC 所控制的所有 Pod 副本的 CPU 利用率的算术平均值了。某一时刻，CPU Utilization Percentage 的值超过 80%，则意味着当前 Pod 副本数量可能不支持接下来所有的请求，需要进行动态扩容，而请求高峰时段过去后，Pod 利用率又会降下来，对应 Pod 副本也会减少数量。

CPU Utilization Percentage 计算过程中使用到的 Pod 的 COU 的量通常是 1 分钟的平均值，目前通过查询 Heapster 扩展组件来得到这个值，所以需要安装布署 Heapster，这样便增加了系统的复杂度和实施 HPA 特性的复杂度。未来的计划是实现一个自身性能数据采集的模块。

```
apiVersion: autoscaling/v1

kind: HorizontalPodAutoscaler

matadata:

  name: php-apache

  namespace: default

spec:

  maxReplica: 10

  minReplica: 1

  scaleTargetRef:

    kind: Deployment

    name: php-apache

  targetCPUUtilizationPercentage: 90
```

2.8 Service（服务）

2.8.1 概述

Service 同样也是 k8s 核心资源对象之一，它即是我们常提起微服务构架的一个“微服务”，之前我们说的 Pod、RC 等资源对象其实都是为这节所说的 Service（服务）做准备的。

Pod、RC 与 Service 的关系.png

k8s 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由 Pod 副本组成的集群实例，Service 和其后端 Pod 副本集群之间则是通过 Label Selector 来实现无缝对接的；RC 的作用实际是保证 Service 的服务能力与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，同时程序架构也变得灵活很多。

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 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，一般的做法都是布署一个负载局衡器（软件或硬件），为这组 Pod 开启一个对外的服务端口 8000，并将这些 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+服务器端口来访问此服务，而客户端的请求最后会被转发到那个 Pod，则由负载均衡器的算法决定。

k8s 也遵循了上述做法，运行在每个 Node 上的 kube-proxy 进程其实就是只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 上面，并在内部实现服务的负载均衡和会话保持机制。但 k8s 发明了一种巧妙影响深远的设计：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 IP 地址又被称为 Cluster IP。这样的话，虽然 Pod 的 Endpoint 会随着 Pod 的创建和销毁而发生改变，但是 Cluster IP 会伴随 Service 的整个生命周期，只要通过 Service 的 Name 和 Cluster IP 进行 DNS 域名映射，Pod 的服务发现通过访问 Service 即可解决。

2.8.2 k8s 的服务发现机制

任何分布式系统都会有服务发现的基础问题，大部分的分布式系统通过提供特定的 API 接口，但这增加了系统的可侵入性。

最早 k8s 采用 Linux 环境变量去解决这个问题，即每个 Service 生成一些对应的 Linux 变量（ENV），并在每个 Pod 容器在启动时，自动注入这些环境变量。后来 k8s **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，这样程序就可以直接使用服务名来建立通信连接。**

2.8.3 外部系统访问 Service 的问题

为了更加深入的理解和掌握 k8s，我们要弄明白 k8s 的三个 IP 的关键问题：

- Node IP：Node 节点的 IP 地址。
- Pod IP：Pod 的 IP 地址。
- Cluster IP：Service 的 IP 地址。

Node IP 是 k8s 集群中每个节点物理网卡的 IP 地址，这是一个真是存在的物理网络，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 k8s 集群。这也**表明了 k8s 集群之外的节点访问的 k8s 集群内的某个节点或者 TCP/IP 服务的时候，必须通过 Node IP 进行通信。**

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，k8s 要求 Pod 里的容器访问另一个 Pod 的容器时，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。这个参考 docker 通信相关的细节。

最后看 **Service 的 Cluster IP，它也是一个虚拟的 IP。**更像一个假的 IP：

- Cluster IP 仅仅作用域 k8s Service 这个对象，并由 k8s 管理和分配 IP 地址（来源 Cluster IP 地址池）。
- Cluster IP 无法被 ping，因为没有一个"网络实体对象"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 k8s 集群这样一个封闭的空间，集群之外节点若想访问这个通信端口，需要做额外的操作。
- **在 k8s 集群之内，Node IP 网、Pod IP 网与 Cluster IP 网之间的通信，采用的是 k8s 自己设计的一种编程方式的特殊的路由规则，和我们熟知的 IP 路由有很大的不同。**

那么基于上述，**我们知道：Service 的 Cluster IP 属于 k8s 集群内部的地址，无法在集群外部直接使用这个地址。**那么矛盾在实际中就是开发的业务肯定是一部分提供给 k8s 集群外部的应用或用户使用的，典型的就是 web，那用户如何访问。

采用 NodePort 是解决上述最直接、最常用、最有效的方法。

```
apiVersion: v1

kind: Service

metadata:

  name: tomcat-service

spec:

  type: NodePort

  ports:

  - port: 8080

    nodePort: 31002

  selector:

    tier: frontend
```

nodePort: 31002 这个属性表名我们手动指定 tomcat-service 的 NodePort 为 31002，否则 k8s 就会自动分配一个可用的端口，这个端口我们可以直接在浏览器中访问：`http://<nodePort IP>:31002`。

NodePort 的实现方式就是在 k8s 集群里每个 Node 上为外部访问的 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 地址和具体的 NodePort 端口号就能访问这个服务。在任意的 Node 上运行 netstat 命令，我们就能看到 NodePort 端口被监听。

但 NodePort 并没有完全解决 Service 的所有问题，如负载均衡的问题。假如我们的集群中 10 个 Node，则此时最好有一个均衡负载器，外部的请求只需要访问此负载均衡器的 IP 地址，则负载均衡器负责转发流量到后面某个 Node 的 NodePort 上面。

Load Balance 组件独立于 k8s 集群之外，通常是一个硬件的负载均衡器或是软件方式实现的，如 HAProxy 或 Nginx。对每个 Service，我们通常配置一个对应的 Load Balance 的实例来转发流量到后端的 Node 上，这增加了工作量即出错的概率。

未完、待续.....

2.9 Volume（存储卷）