第N章 容器云平台的高可用设计

作者： 光大科技 - 谢楚瑜

高可用（HighAvailability），是设计分布式系统架构所必须要考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。任何系统的中断，都会带来不同程度的损失。所以为了避免损失，我们需要尽可能的提高平台中的可用性。

那么如何为容器云平台实现高可用呢？对于其中的应用又要如何保障其可用性呢？在这一章中您将会学习容器云平台的系统架构，并且自底向上的为其中的每一个部分实现高可用。

N.1 绪论

N.1.1 什么是高可用

高可用（HighAvailability），是设计分布式系统架构所必须要考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。

虽然大部分情况下，我们的系统都能够很好的运行。但绝对不会故障的系统是不存在的。无论代码写的多好，硬件如何的可靠，也无法保证整个系统的稳定。因为故障总是会发生在意想不到的地方。

比如 2015 年 5 月 27 日支付宝在杭州的一个机房光纤被挖断，导致部分用户无法正常使用支付宝，直到当日晚上 7 点 20 分，支付宝才宣布用户服务恢复正常。

再以微信为例，在 2019 年 10 月 29 日微信发生了一场事故：与银联系统对接时发生网络故障，导致支付系统宕机近 30 分钟，使得近两千万笔订单无法正常的完成。

阿里腾讯的技术水平不可谓不强，而支付业务又是其中的重中之重。尽管如此，还是会有一些揪心的时刻。 为了能够尽可能的减少服务的不可用时间，最大降低损失，我们可以通过架构设计的手段为服务提供强的可用性，即高可用架构设计。

N.1.1.1 高可用架构的设计原则

- **冗余**： 单点永远是高可用的最大敌人。任何组件都有可能奔溃，哪怕只是简单的执行`ping`命令。单点组件一旦奔溃，如果没有后备组件的话，那么这一组件就算是彻底不可用了。为了保证组件的故障不会导致整个系统的故障，我们应当为每一个组件都留有一个到 N 个冗余后备。
- **故障转移**： 出了故障后，要是请求还在往故障的节点上发，那么有再多的冗余节点也是毫无意义。所以需要自动的检查故障的产生，并将流量转发到冗余中的可用节点上。而想要及时的实现故障转移，我们就需要有健康检查机制去检查服务的状态。
- **排查异常**： 如果高可用架构设计完美，那么即使发生故障，用户也很可能永远感知不到。但这并不代表就不需要检查并解决故障了，定期的维护可以进一步提高系统的可用性。

N.1.1.2 其他未能提到的机制

对于高可用架构还应当考虑服务分级与降级，灾难恢复与异地多中心等等机制或场景。随着服务规模的不断扩张，保持或提高可用性的成本会以一种夸张的方式上涨。由于篇幅有限，对于很多知识点都难以面面俱到的深入探讨，在实际应用中还是需要更多的参考实际场景来进行规划。

N.1.2 容器云平台架构介绍

通过前一节的介绍我们可以知道，如果想要保证整个云平台的高可用，对于云平台各个部分自然是需要一定程度了解的。接下来让我们看看常见的容器云平台架构：

- **基础设施**： 基础设施指的是位于架构中最底层的物理设备及相应的资源池。虽然基础设施通常都有专门的团队进行维护，或者使用了诸如 AWS、阿里云之类的公有云服务。但是我们仍然需要能够为应用实现对物理机的亲和与反亲和（如多个高 IO 应用避免部署在同一个物理机下的不同虚拟机中），对于部分应用还需要做到分机房部署。

- **容器云平台**： 容器云本质上我们可以将其看作就是一个普通的 Web 应用（通常是无状态的，持久化数据使用集群 etcd），通过将其部署到 Kubernetes 中，并配置好健康检查与副本集来为其实现高可用。

- **私有镜像仓库**： 如果我们的容器云平台是私有云，或者我们的 CI\CD 制品不想上传到公网上，那么我们就会需要一个自己的私有镜像仓库。在容器云平台完成相关设置后，从此以后私有的镜像就可以都从内网中拉取。当有 Pod 不可用，开始漂移时，这个 Pod 可能会飘到一个并没有去过的节点中，这时候就会需要向私有镜像仓库拉取镜像。如果这个过程失败了，那么我们的 Pod 就会长时间处于不可用的状态，进而可能会对业务的可用性造成影响。

- **容器云平台纳管的 Kubernetes 集群**： 这些集群是通过容器云平台进行创建和管理的集群。通常多云架构中会包含这一层，这样通过在多个云厂商下部署集群来提高更多的冗余。当然，这一层并不是必要的。

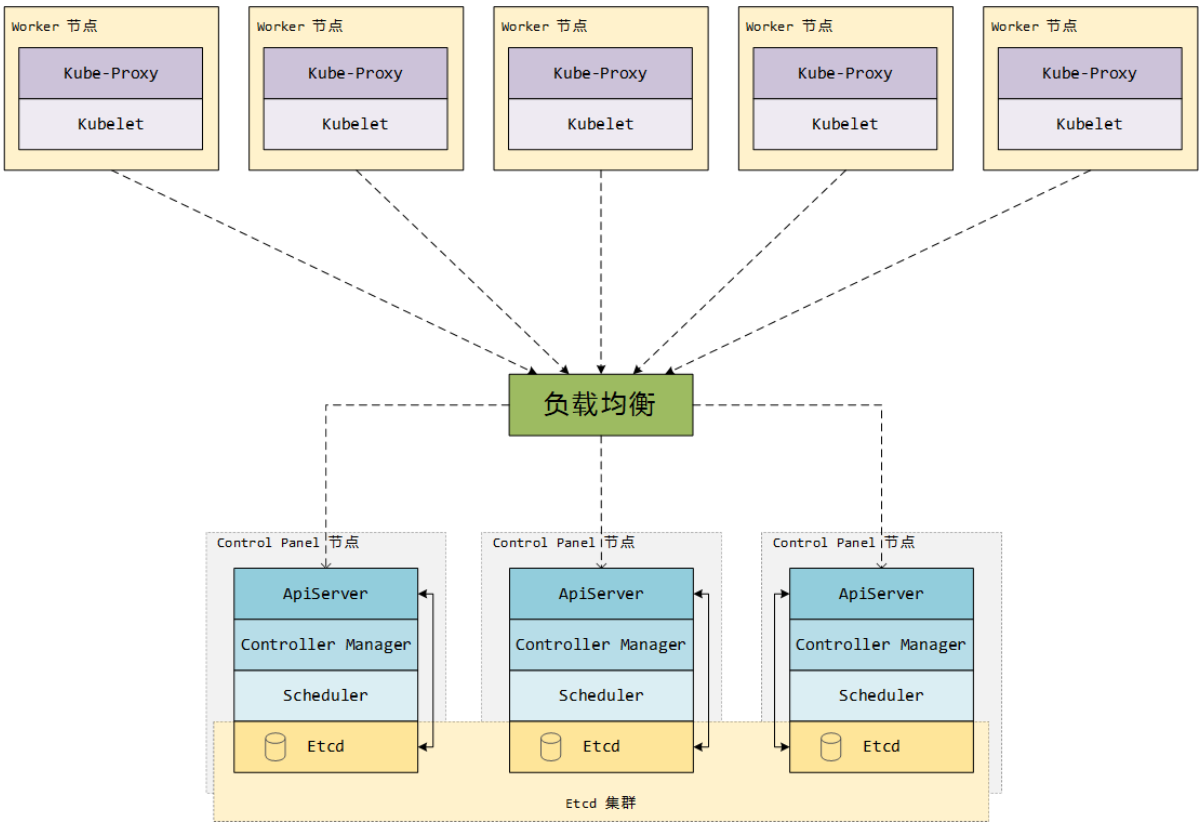
- **容器云托管应用**： 在布置好容器云平台后，我们就可以在平台上部署应用了。这一部分除了容器云平台的用户所部署的应用外，还包含诸如监控、日志收集和网格网络等由容器云管理者所创建的应用。这些应用的可用性和状态则由容器云平台和 Kubernetes 共同维护，而这部分内容会在后面的小节中具体描述。

N.2 Kubernetes集群高可用

在容器这一新秀刚刚发展起来的那段时间里，曾经出现过大量的容器编排系统。其中以`Swarm`、`Mesos`和`Kubernetes`三个项目最受欢迎。在经过多年的竞争与发展后，如今，由 Kubernetes 构建的容器生态体系已经成为了容器领域中的事实标准。

安装一个高可用的 Kubernetes 集群是保障容器云平台高可用的第一步。尽管我们可以通过诸如 Kubeadm、RKE 或 Kops 之类的工具轻松的安装一个 Kubernetes 集群，但是了解一下常见的 Kubernetes 高可用的实现方法才可以帮助我们更好的维护集群。

N.2.1 Kubernetes高可用架构



如图所示，我们将一个高可用的 Kubernetes 集群分为了由三种节点组成的集群。这三种节点分别是`Control Panel`节点、`Worker`节点、`Etcd`节点。

N.2.2 Control Panel节点

ControlPanel 主要负责对集群进行全局决策，他们检测并响应集群事件。其中各组件描述如下：

- *ApiServer*： ApiServer 是 Kubernetes 服务的入口，我们对于 Kubernetes 所做的任何操作都是通过 ApiServer 进行的。为了实现该组件的高可用，需要在在全部 Control Panel 节点上部署 ApiServer，并为这些 Pod 都配置到同一个负载均衡器中。再将其他组件中的 Api 地址指向这个负载均衡器。这样的话，只要还有一个 ApiServer Pod 可以使用，那么请求就可以通过负载均衡发送到 ApiServer 中去处理。

- *Controller Manager*： Controller Manager 当中包含了大量的控制器，这些控制器监听着 ApiServer 中全部资源的状态，并始终维护这些资源到正确的状态。通过这个组件，Kubernetes 实现了其自愈性与声明式创建的功能。同样的，了实现该组件的高可用，需要在在全部 Control Panel 节点上部署 Controller Manager。

- *Schedule*： Schedule 负责在创建工作负载（Workload）时，为工作负载中各个 Pod 的部署提供推荐的节点。如果我们在配置中加入了亲和性、反亲和性、选择器、资源配额等配置的话，Schedule 在调度 Pod 时会优先考虑这些设置，如果不存在符合用户要求的节点，哪怕集群中计算资源足够多，这个 Pod 也不会被部署到集群中。这个组件的高可用也是通过在全部 Control Panel 节点上部署即可。

N.2.3 Etcd节点

Etcd 是 Kubernetes 中的各类数据的存储中心，对于在 ApiServer 中我们可以看到的各种数据都是持久化在 Etcd 中的，一旦 etcd 不可用那么 ApiServer 也会不可用。而维护 Kubernetes 中大部分组件状态所使用的 Controller Manager 组件又依赖于 ApiServer 的可用。所以当 etcd 出现故障

时，虽然各个工作负载可能还是能正常运行，但是其可用性已经完全失去了保障，同时我们也将无法再对这个 Kubernetes 集群进行任何变更。

所以说 Etcd 是 Kubernetes 中至关重要的一个组件。想要为这个组件实现更高的可用性和稳定性的话，对于这个组件的一些特性有必要进行进一步的了解。

- **Raft 选举**：Etcd 的高可用方案类似与 Redis 的哨兵机制。首先我们创建一个 Etcd 集群，这时候每一个 Etcd 节点都是 Follower 节点，而在经历一轮选举后，这个 Etcd 集群中就会产生一个 Leader 节点。这时候我们所有的改动与新增只需要与 Leader 通信，Leader 再将数据同步到其他的 Etcd 节点中。这种同步算法就是Raft算法。如果想要多了解一些 raft 算法的机制，可以看一下 http://thesecretlivesofdata.com/raft/

  在 Etcd 中，数据的同步和 Leader 的选举都需要大多数的节点达成共识，在这里，**大多数**指的是`(n - 1)/2`（向下取整）。由这个算式我们可以发现，偶数个节点并不能为集群提供更多的冗余，所以通常情况下，我们都会部署一个 3 节点的集群，如果对于 Master 的安全性有更高的要求，可以布置 5 节点集群，但是集群大小最好不要大于 7。更大的集群虽然可以提供更高容错能力，但是由于 raft 的同步机制原因，会对写入能力造成很大的影响，所以不建议部署超过 7 个节点的 etcd 集群。

  一个 Raft 集群只能有一个 Leader 存在，如果一旦发生网络分区，Leader 只会在多数派一边被选举出来，而少数派则全部处于 Follower 或 Candidate 状态，所以一个长期运行的集群是不存在脑裂问题的。尽管短期内的多个 Leader 并存的现象是可能存在的，但在 etcd 中也有 `ReadIndex`、`Lease read` 机制来解决这种状态下的一致性语义问题。这样一来，在出现网络分区后，少数派中的节点将不再支持读写，也就是牺牲可用性换取了数据的强一致性。除此之外我们可以通过减少选举周期(`election-timeout`)的时长来尽可能的减少多个 Leader 并存这一状况的产生时间。

  > 虽然我们可以通过`--consistency=s`选项开启对于非一致性读请求的支持，以提高可用性。而且在某些场景下，我们可能就是需要牺牲一致性换取更高的可用性。但在Kubernetes中，弱一致性可能会导致集群中应用的状态出现异常，所以还是用默认设置比较好。

- **数据持久化**：Etcd 的数据会时刻以日志的形式记录在内存中和硬盘的 wal 文件中。在这个过程中 etcd 对磁盘的延迟会非常敏感，如果有意外的磁盘活动导致`fsync`延迟变长，很可能会导致 etcd 错过心跳包，从而产生请求超时或 Leader 丢失的结果。所以我们需要给 etcd 配置更高的磁盘优先级来避免这种情况的发生。

  在 linux 上，我们可以通过 ionice 来配置磁盘优先级：

  ```
  $ sudo ionice -c2 -n0 -p `pgrep etcd`
  ```

  如果可以的话，给 etcd 挂载 ssd 盘可以显著的提升 etcd 的性能。

- **数据同步**： Etcd 的 Leader 在收到变更时，会将变更发送到各个 Follower 节点。当集群中的大多数节点返回了认同变更的响应后，Leader 才会 commit 这个变更，并将这个变更同步到各个 Follower 中,这种行为就是二阶段提交（Two phase Commit）。如果说这个过程中存在较大的网络延迟，也会极大的影响 Etcd 的写入能力，产生请求超时甚至 Leader 丢失的结果。

通过减少节点间的物理距离，减少网络拥塞，提高设备吞吐能力等方式可以有效的减少网络延迟

- **快照**： Etcd 默认会记录 10000 条更改日志，当超过这个阈值的时候，etcd 将会生成一个快照记录当前资源的状态，并删除当前内存和硬盘中过时的日志记录。如果这个值太高可能会导致 Etcd 占用过多的内存，甚至达到触发 OOM 的程度，可能导致集群中出现 Pod 漂移以及 Control Panel 不可用。如果需要减少 etcd 的内存和磁盘占用，我们可以通过下面的方法修改阈值为 5000：

```
#命令行参数：
$ etcd --snapshot-count = 5000
#环境变量：
$ ETCD_SNAPSHOT_COUNT = 5000
```

> 由于Etcd集群中数据会保持一致，所以当我们发现某一个Etcd节点内存紧张的时候，往往整个Etcd集群的节点内存都以及处于比较紧张的状态了。为Etcd的内存状态专门设置监控和告警是很有必要的。

N.2.4 Worker节点

worker 节点本质上不存在高可用问题，他是 Kubernetes 中工作负载（即 Deployment，DaemonSet,Job 等对象）的载体。我们关心的往往不是 Worker 节点是否可用，而是运行在其中的工作负载是否可用。在 Worker 节点中通常有以下组件：

- *Kubelet*： kubelet 是一个 Agent，用来维护各个节点上的 Pod 与容器间的状态。kubelet 接收一组通过各类机制提供给它的 PodSpecs，确保这些 PodSpecs 中描述的容器处于运行状态且健康，而不会管理非 Kubernetes 创建的容器。
- *KubeProxy*： KubeProxy 则是维护集群网络规则的代理。这个组件会维护节点上的网络规则，这些网络规则允许从集群内部或外部的网络会话与 Pod 进行网络通信。

  KubeProxy 可以配置为`iptables`代理模式、`IPVS`代理模式或`User space`代理模式。其中`User space`模式 与 `iptables`模式 都会依赖于 iptables 来制定网络规则，随着 Iptables 中规则数量的增加，会导致内核逐渐变得非常繁忙，而专为负载均衡设计的 IPVS 则并不存在这个问题，IPVS 几乎可以无限扩容。同时 IPVS 模式也支持更高的网络吞吐量。但是我们需要在全部的节点中安装 IPVS 内核模块后才能正常的使用 IPVS 模式。
- *Container Runtime*: 容器运行时是负责运行容器的软件。 Kubernetes 支持多种容器的运行时: Docker、 containerd、cri-o、 rktlet 以及任何实现 Kubernetes CRI (容器运行环境接口)。

在后面的内容中，我们会重点讲解如何为集群中的应用配置高可用性，使其能够更加灵活的面对诸如 worker 节点不可用、网络故障等种种问题

N.2.5 Etcd是否应该与Control Panel在同一组节点上?

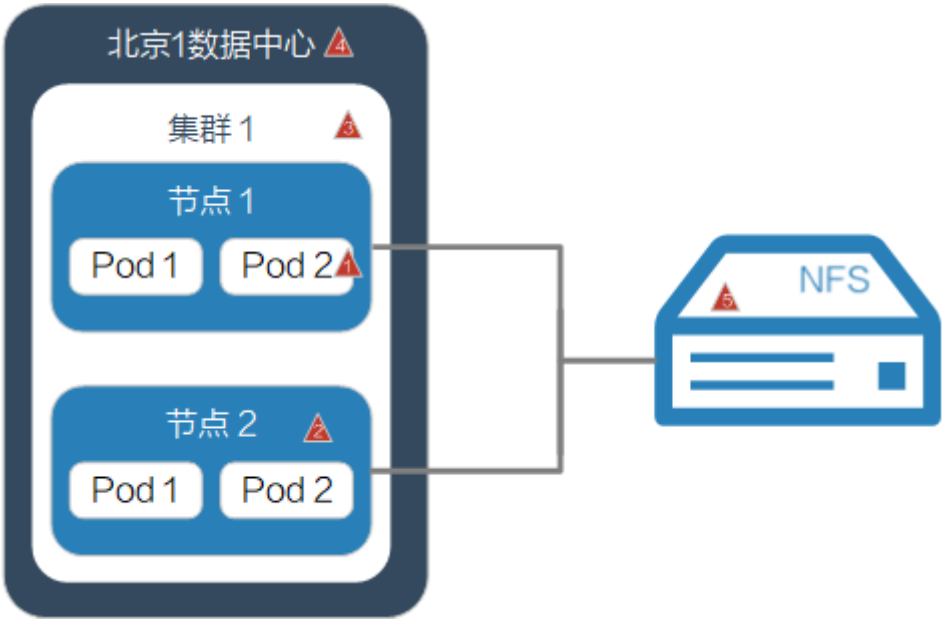
由于用途不同，Etcd 主要使用的是内存资源，而 Control Panel 中的组件则主要是需要运算能力。所以将 Etcd 和 Control 组件放在同一位置中可以优化计算资源的分配。

但在生产环境中，etcd 最好单独部署在满足运行条件的节点上。这样可以提供更多的冗余性和可用性，但会需要更多的计算资源。

N.3 应用高可用

虽然我们已经有了一个高可用的 Kubernetes 集群，但是这并不代表我们部署在上面的工作负载也会自动变得高可用。为了能够让我们部署在上面的应用（包括容器云平台这个应用）能够实现高可用，需要了解阻碍我们实现高可用的故障主要都存在于何处，以及如何为 Kubernetes 中的工作负载实现高可用。

N.3.1 Kubernetes中的潜在故障点



在这张图中我们罗列了常见的 5 个潜在的故障点。通过认识这些故障点，可以帮助我们通过冗余和反亲和性的方式为应用带来更高的可用性。

N.3.1.1 容器或者Pod故障

意外的异常、部署配置不正确、压力过大等情况可能会导致一个 Pod 中的容器处于不可用的状态。为了能让应用不受到某些 Pod 的异常带来的影响，需要部署足够多的 Pod，并分配在足够多的 Worker 节点上。

N.3.1.2 Worker节点故障

Worker 节点通常都是运行在物理机上的虚拟机，其可用性可能会受到其宿主机的硬件故障，网络故障等故障的影响。为了避免单个 Worker 节点故障带来的不可用，应该要创建更多的 Worker 节点。

不过尽管拥有了多个 Worker 节点，但是这些节点可能都是同一个物理机中的虚拟机。如果这一台物理机出现了故障的话，带来的结果可能依然是全军覆没。为了避免这种情况的出现，应该与基础设施方面沟通，让 worker 节点均匀分布在更多的物理机、机柜中。

N.3.1.3 集群故障

Kubernetes 集群的高可用可以参考第二节，Kubernetes 高可用架构。

> 值得注意的是，在集群中的Control Panel或者etcd发生故障的时候，各个Worker节点中的Pod可能都还是正常工作的。但是这个时候如果重新启动的worker节点，那么会导致这一节点上的Pod全部不可用。因为Controller Manager已经无法工作了。

N.3.1.4 数据中心故障

机房故障会导致区域内所有的物理机和 NFS 存储不可用。其成因可能包括电源、网络、散热或者洪水地震等非人为因素。为了避免单个机房故障所带来的不可用，我们可以使用两地三中心的方式去部署集群。

N.3.1.5 存储故障

数据安全永远是重中之重的重中之重，其可用性应该是各个部分中最优先的。存储的可用主要受到散热（ssd）、断电（长时间或者突然断电可能导致 ssd 数据丢失）、网络、震动（hdd）、磁盘寿命等因素的影响。我们可以通过使用合适的双机热备，冷备的方式为存储提供高可用性。

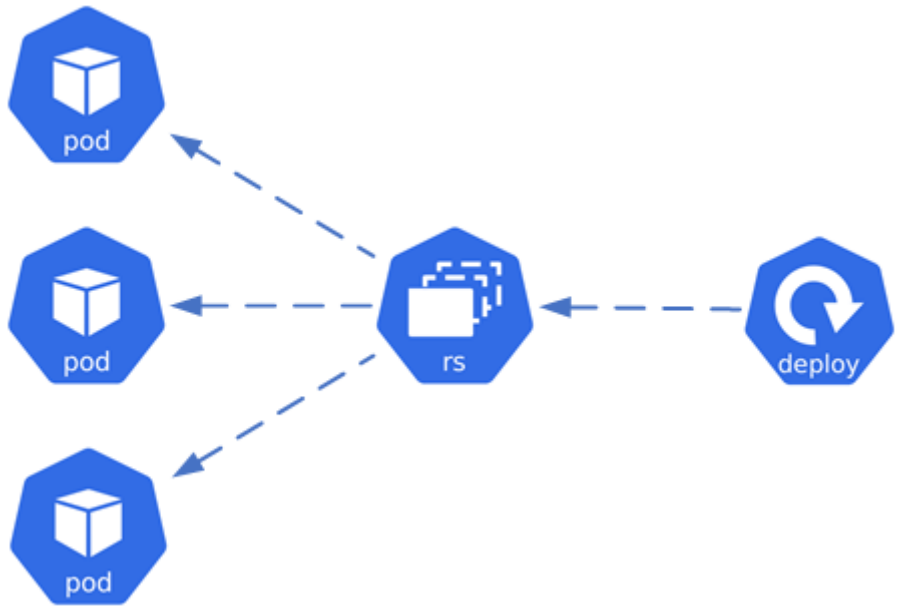
N.3.2 Kubernetes如何保障应用高可用

接下来我们从 Kubernetes 中提供的功能下手，了解如何通过 Kubernetes 集群创建一个在集群中高可用的应用。

N.3.2.1 ReplicaSet

ReplicaSet（副本集）是 Kubernets 中的一个非常重要的概念。ReplicaSet 是 Deployment 对象的子集，其功能主要是维持某一个 Pod 的副本数量，通过这个功能我们可以实现 Pod 的水平扩展/收缩与滚动更新。对于无状态的组件而已，ReplicaSet 是一种非常合适的载体：我们只需要 Service 为多个 Pod 提供负载均衡就可以轻松实现高可用。

但在通常情况下我们并不会直接的使用这个对象，而是通过使用 Deployment 对象的方式来实现 ReplicaSet 的效果。其中 Pod，ReplicaSet 和 Deployment 之间的关系如下图所示：



N.3.2.2 弹性伸缩（Horizontal Pod Autoscaler）

大部分工作负载的资源使用率都会随着业务需求的增长而不断增加。当 cpu 资源不足的时候，工作负载中 Pod 的响应时间会不断变长直至超时；而当内存不足的时候 Pod 可能会出现一些异常，甚至被系统杀死。为了应对业务扩张的情况，我们可以为这个工作负载设置弹性伸缩。当这个工作负载的 Deployment、ReplicaSet 或者 StatefulSet（状态集）的某项指标达到我们所设定的极限时，对应的 Controller 就会增加其中 Pod 数量，反过来，指标低于我们设置的极限时，Pod 的数量就会减少。

通常情况下，我们只能通过 Pod 的 cpu 使用率为其设置弹性伸缩。但是我们也可以自行为弹性伸缩配置更多的指标：比如内存的使用，磁盘 io 的使用。如果某些业务之间的流量压力存在相关性，也可以设置为 A 业务的工作负载监听 B 业务工作负载的压力以自动扩展。

如果在一定时间内业务需求上下波动较多，可能会导致过多的扩展与收缩，这种行为被称为抖动(thrashing)。为了消除抖动，我们可以为弹性伸缩设置冷却时间，和各类伸缩策略来实现更加稳定、高效的伸缩方式，为工作负载带来更好的并发能力。

N.3.2.3 滚动更新

我们希望应用总是可用，又希望应用能始终维护到最新的版本。对于这种情况，我们就需要滚动更新功能。滚动更新允许通过用新的 Pod 实例增量更新 Pod 实例。如果我们是通过 Service 的方式访问的该 Deployment，那么通过滚动更新就能做到无感知的更新服务。

滚动更新也有阻止故障扩散的作用。我们可以通过修改 rollingUpdate 的 maxSurge 指定一次滚动更新中最多创建多少个新 Pod，修改 maxUnavailable 控制更新时最多可以删除多少个旧 Pod。这两个值都可以设置为百分比，其具体数据需要依靠实际情况去做判断。假设我们有一个 Deployment 的下一个版本出现了导致不可用的异常，或者写了一个不存在镜像 tag。那么在更新时，旧的可用 Pod 仍然会保留一定的数量（只要 maxUnavailable 不是写的 100%或者大于 Deployment 的 replicas 设置），同时滚动更新将会停止，服务始终保持可用状态。

这就要求我们做好关于这个 ReplicaSet 的健康检查，不能简单的依靠其 Running 状态来判断其是否可用。比如像是一个启动时间比较长的服务，可能容器已经时 running 了，但是由于其服务还未启动，所以无法正常使用这个服务。

N.3.2.4 健康检查

在很多时候，服务可能不可用了，但是进程仍然存在与机器中。或者服务可用，但是响应时间超出我们的限制。又或者接口返回异常之类的其他我们认为服务以及不可用的场景。这些情况下我们都需要进行故障转移。

所以我们需要一种检查服务是否健康的组件，这个组件可能需要预设一个目标操作，响应时间，响应周期，正常响应内容等。然后这个组件就会周期性的检查服务是否可用，并为故障转移服务提供相应的结果。在 Kubernetes 中，为健康状态的检查准备以下几种探针：

存活检测探针（Liveness Probes）： 存活检测探针用于检查容器中的应用是否处于存活与运行状态。通过设置，Kubernets 可以定期的执行一些命令或者发送 TCP/HTTP/HTTPS 请求来判断容器的响应是否符合预期。如果存活检测探针发现了与预期不符的结果，Kubernetes 会通过重启容器来尝试恢复整个 Pod 的功能。

就绪检测探针（Readiness Probes）： 就绪检测探针在设置上与存活检测探针类似。但是这一个应用是用来判断容器中的应用是否准备好了接收流量。有一些容器的启动流程需要初始化，或者

重新启动进程以应用新的配置，当就绪准备探针判定与预期不符后，Kubernetes 会暂停向这个 Pod 发送流量。这样 Pod 可以继续进行其初始化或重启任务，而不会杀死这个容器。

启动检测探针（Startup Probes）： 启动检测探针设置上也与前两者相似。其目的是为了保护启动时间过于漫长的容器。通过为这个探针设置较长的检查间隔和检查次数可以保护容器在存活检测达到最大失败尝试次数时不重启容器，而是在经历了启动检测探针尝试次数*启动检测探针检查间隔的时间之后才会重启应用。通过这一点，我们不用将存活检测探针设置过大的值去等待应用启动，而是为启动检测探针设置一个值来保护容器的启动过程。

通过灵活的配置这几种探针，我们可以让应用及时的发现问题，并决定是否重启。这样由 Kubernetes 管理应用的可用性与健康状态后，可以减少很多额外的运维工作。

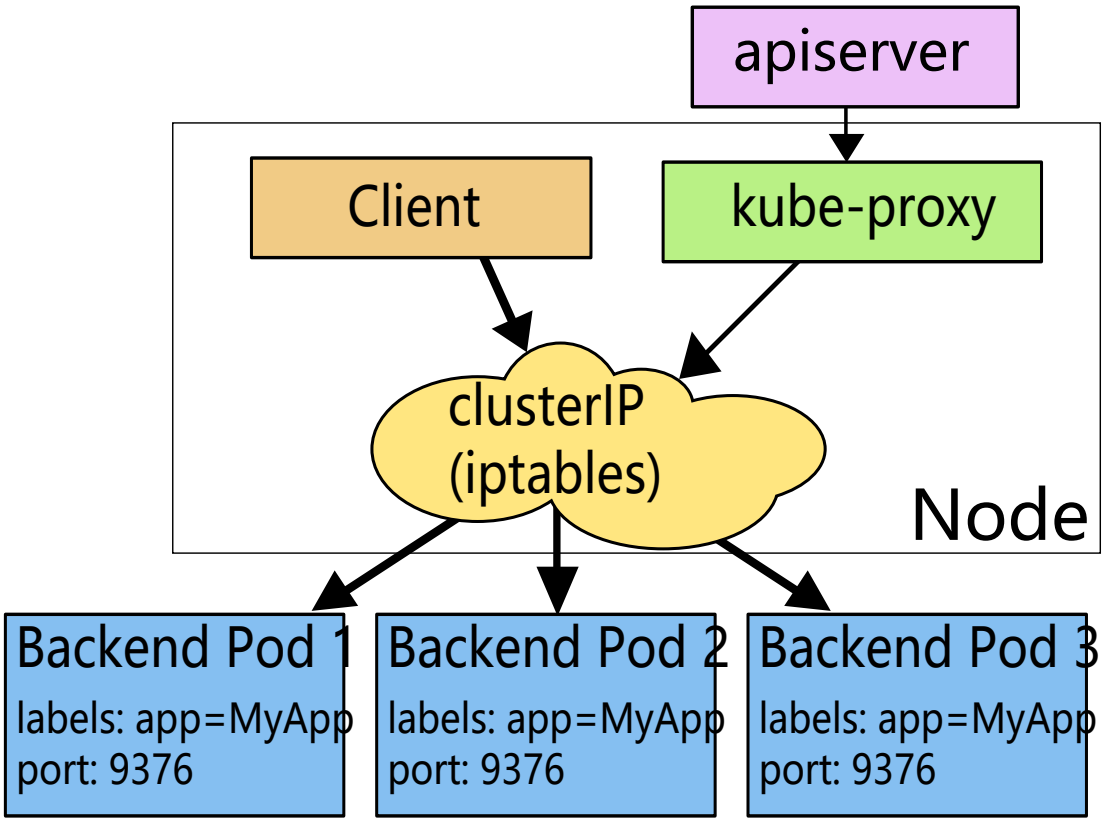
N.3.2.5 Service

在通过前面介绍，我们知道了通常一个应用会有多个相同的 Pod 同时存在以保证其可用性。这导致了一个问题：如果一组 Pod（称为"后端"）为群集内的其他 Pod（称为"前端"）提供服务，那么前端如何找出并监听连接的 IP 地址，以便前端可以随时像后端发送请求呢?

在 Kubernetes 中，最小的容器单位是 Pod。Pod 中的容器共享网络、存储等资源。每一个 Pod 都有一个独立的 ip，而 Service 就是一种针对 Pod 的服务发现。在使用 Service 后，我们可以通过全限定域名（FQDN）访问服务：`Service_name.default.svc.cluster.local`，其中 Service_name 表示服务名称，default 表示服务在其中定义的名称空间，而 svc.cluster.local 是在所有集群本地服务名称中使用的可配置集群域的后缀。

通过设置一组选择器，Kubernetes 会找出符合选择器要求且通过了就绪检测探针的 Pod 集合，并将其 Pod ip 交给 Service。Service 会根据其配置的不同，为 Pod 的 ip 提供不同的 endpoint。Service 分为下面四种：

- **ClusterIP**： Service 默认会使用 Kubernetes 分配的一个 Vip（虚拟 IP）去代理选择器匹配到的 Pod ip 集合。在默认情况下，对这个 Service 的请求会被随机分配到一个 Pod 中。这是因为在 Kubernetes v1.2 以后，kube-proxy 的 iptables 模式成为了默认设置。



通过 iptables 这种方式处理流量的系统开销更加低，同时由于无需在用户空间和内核空间之间切换，这种方法也可能更可靠。

- **Headless**： 如果在创建 Service 的时候将 ClusterIP 设置为 None，就可以创建 Headless 模式的 Service。

  对于 headless Service 并不会分配 Cluster IP，kube-proxy 不会处理它们，而且平台也不会为它们进行负载均衡和路由。在访问这个 Service 的域名的时候，会直接返回对应 Pod ip 的集合。如果客户端没办法解析这个集合的话，它可能会只会拿到第一个 Pod 的 IP 地址。

Service 也可以不设置选择器，用来实现以下场景：

- 希望在生产环境中使用外部的数据库集群，但测试环境使用自己的数据库。
- 希望服务指向另一个 命名空间 中或其它集群中的服务。
- 我们想要正在将工作负载迁移到 Kubernetes。 在评估该方法时，我们仅在 Kubernetes 中运行一部分后端。

Service 连带着 Kubernetes 网络都是 Kubernetes 中的一个重要且复杂的概念，如果需要更多介绍可以看官方中文文档 https://Kubernetes.io/zh/docs/concepts/Services-networking/Service/#定义-Service

N.3.2.6 Ingress

有了 Service 我们就可以在集群内部轻松的建立起 Pods 之间的连接，但是在集群之外是无法访问这些 Service 的，为了能够从集群外访问集群内的服务，我们需要用 Ingress 将 Service 暴露出来。

不过，虽然 Ingress 对象在 Kubernetes 中是默认存在的，但是只是通过新增一个 Ingress 对象是无法实现 Ingress 的效果的，我们需要安装一个 Ingress Controller，让 Ingress Controller 去监听 Ingress 的变化，并实现 Ingress 声明的业务。

常见的 Ingress Controller 有：F5 Networks、HAProxy Ingress、Istio Controller 和 Nginx Controller 等。在一个集群中可以同时运行一种到多种 Ingress Controller，但是往往我们只需要一个 Nginx Controller 就可以满足常规的业务需求了。

通常来说，Ingress Controller 的可用性会影响着对外服务的可用性。如前端应用，web Service 这些需要对外暴露服务的应用。为了保证 Ingress Controller 的高可用，通常我们会使用 DaemonSet 来确保 Ingress 服务运行在每一个节点中。这样一来，只要还有可用的 Worker 节点，就会存在可用的 Ingress Controller 为其提供服务。

Ingress Controller 通常会使用主机网络，并占用 80 与 443 这两个端口。这意味着 Ingress 的更新需要先删除旧 Pod 再创建新 Pod，不过这也意味着我们可以通过访问节点 ip+端口号的方式直接访问 Ingress Controller。这样的话，我们只需要为 Worker 节点配置一个 Vip，就可以轻松的从外界访问到集群内了。

N.3.2.7 StatefulSet

StatefulSet（状态集）是一种类似于 Deployment 工作负载对象。与 Deployment 不同的是，StatefulSet 会为每一个 Pod 都维护一个独立的身份信息。尽管在同一个 StatefulSet 中的 Pod 是基于同一个配置创建的，但是这些 Pod 之间不能互相替换："无论怎么调度，每个 Pod 都有一个永久不变的 ID。"

StatefulSets 非常适合用来解决需要满足以下一个或多个需求的应用程序：

- 稳定的、唯一的网络标识符。
- 稳定的、持久的存储。
- 有序的、优雅的部署和缩放。
- 有序的、自动的滚动更新。

在上面，稳定意味着 Pod 调度或重调度的整个过程是有持久性的。常见的例子就是各类分布式应用,在后面的章节中,我们会通过 StatefulSet 来部署一个 Redis 应用。

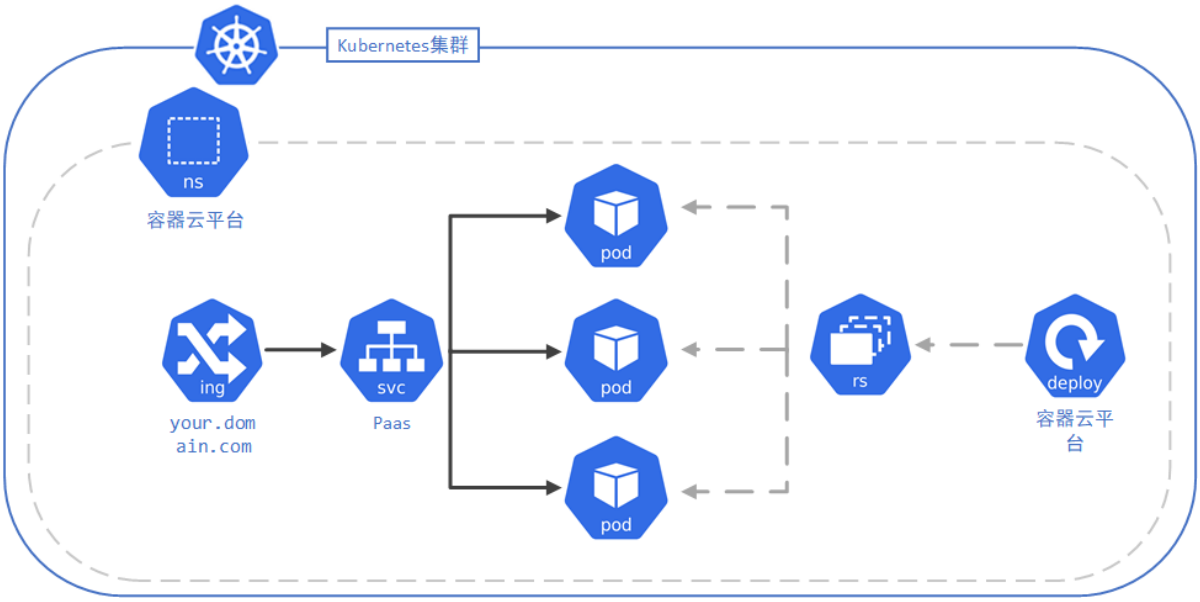
> 如果应用程序不需要任何稳定的标识符或有序的部署、删除或伸缩，则应该使用由一组无状态的副本控制器提供的工作负载来部署应用程序，比如Deployment、DaemonSet等。

N.4 容器云平台高可用设计

在前面几节中，我们学习了如何为 Kubernetes 集群实现高可用，以及在 Kubernetes 中如何创建高可用应用。现在让我们结合这些知识，为容器云平台设计高可用方案。而这些方案也适用于种种运行在容器云平台中的应用，对于不同的应用，只要稍加改动就是很好的高可用方案了。

N.4.1 容器云平台高可用（无状态组件高可用）

容器云平台本质上可以视为一个无状态的组件，平台中的数据存储在 Kubernetes 的 Etcd 中。



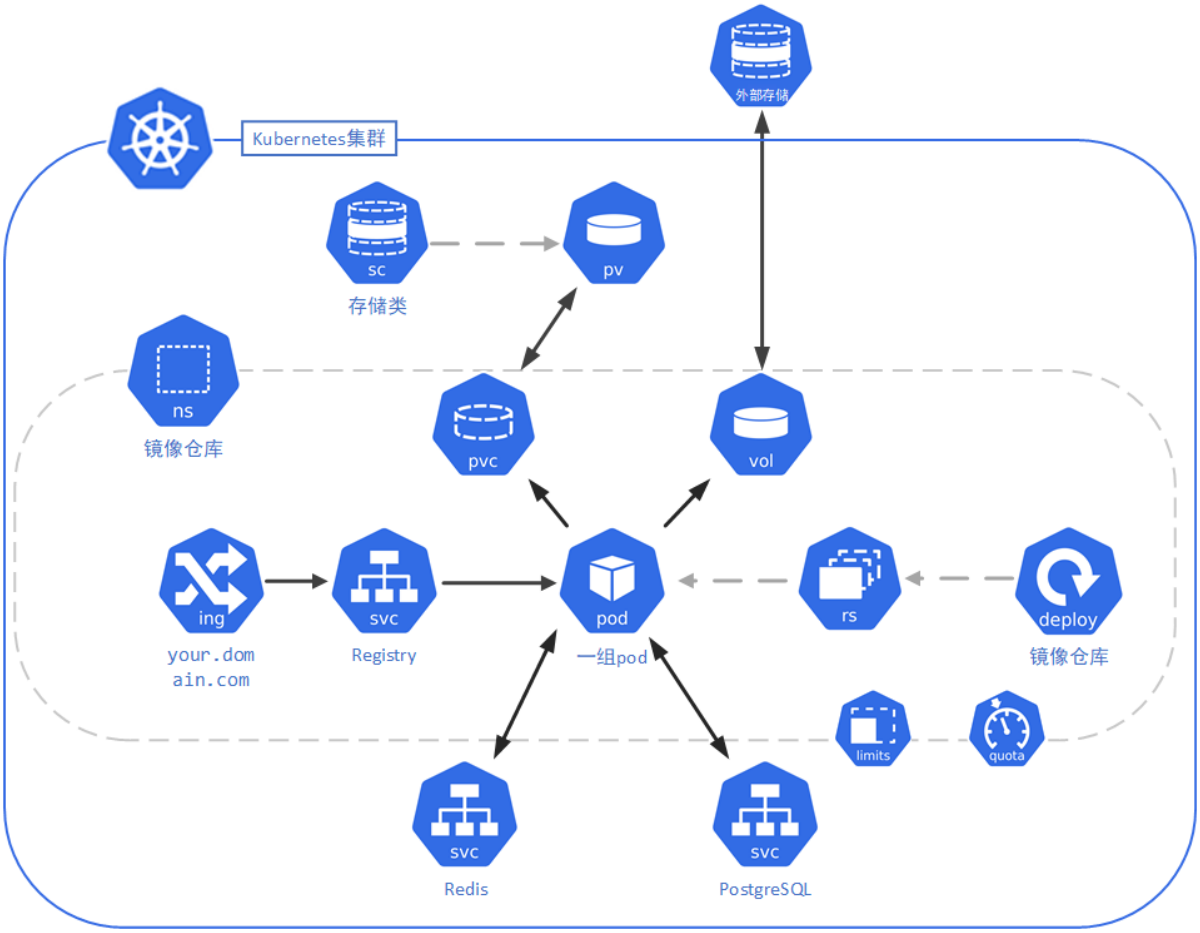
我们将容器云平台的 Pod 通过 Deployment 的方式部署在 Kubernetes 集群中，并将其 Replicas 设置为一个不小于 1 的数。在通过一个选择器，将这几个 Pod 暴露在同一个 Service 中。这样一来，当我们的任意一个 Pod 如果不再健康了，Controller Manager 就会删除这个 Pod，并部署一个新的 Pod。

对于外界而言，由于这些 Pod 是通过 Service 的形式暴露出来的，当有 Pod 没有通过就绪状态检查指针（Readiness Probes）的时候，就无法在通过 Service 访问到这个 Pod，流量会走向其他的 Pod 中。整个过程中，用户并不会察觉到引用的不可用，对于他们来说，只需要访问同一个域名，请求就总是会发送到可用的 Pod 中。其容错能力为`N - 1`（N 为 Pod 数量）。

N.4.2 镜像仓高可用（有状态组件高可用）

在这一节中,我们以'Harbor'为例,详细讲解一下镜像仓高可用。镜像仓的高可用方案与容器云平台不同在于，镜像仓中存在多个有状态组件，而且还需要一个网络存储。

N.4.2.1 镜像仓设计



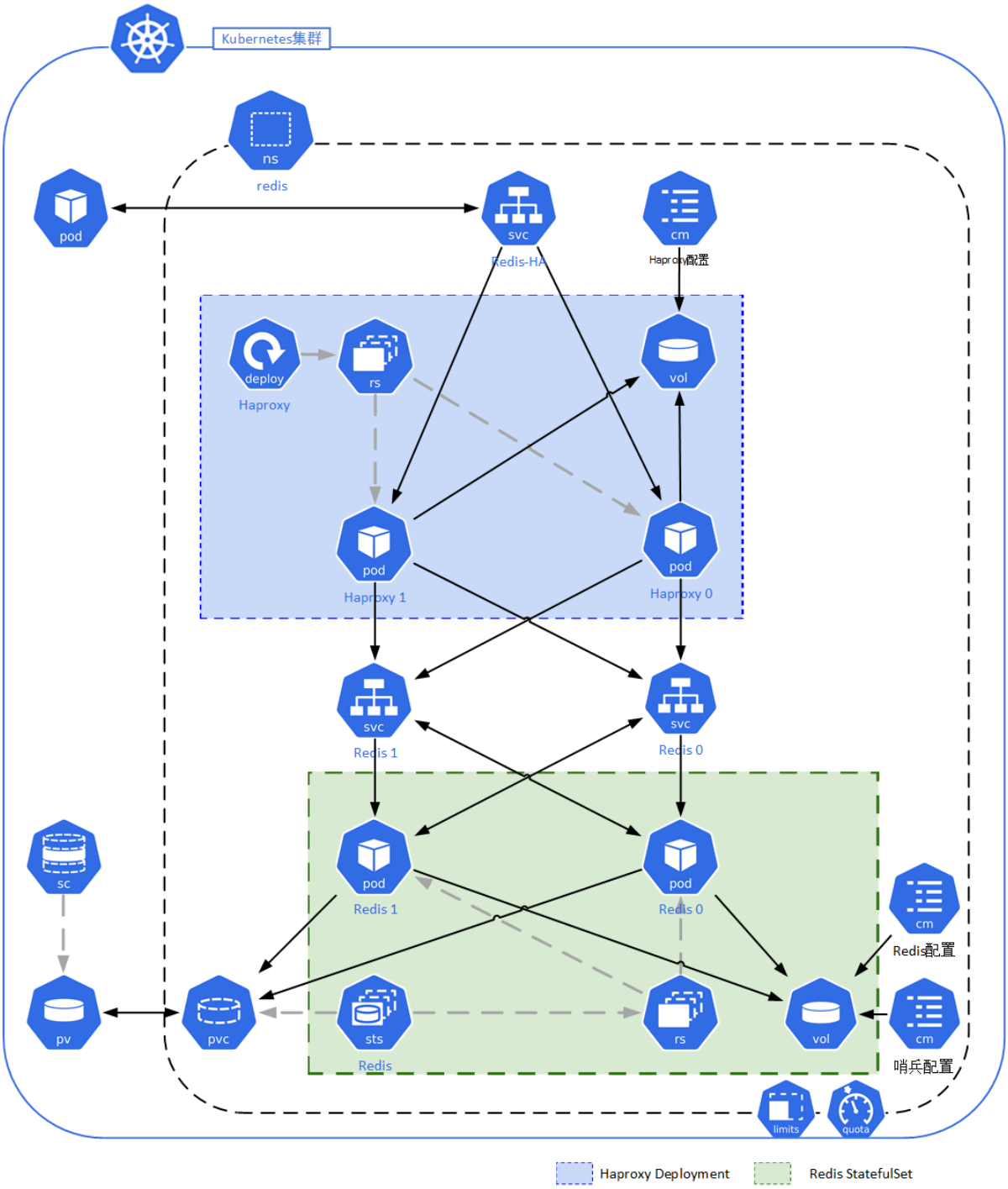
作为一个镜像仓库，持久化镜像文件自然是其中最基本的功能。为了不让镜像文件随着容器的关闭而被删除且多个仓库的镜像都是一致的，这时候需要挂载外部存储到仓库中，并且这个外部存储需要支持多机读写。外部存储的选用视环境而定，我们可以选择挂载 NFS、Nas、oss 和种种云存储。选好了外部存储后，我们只用把同一个外部存储通过 Storage Class 或者其他配置方式挂载多个镜像仓库的 Pod 上，就可以实现数据一致性与数据持久化。

> 需要注意的是，镜像仓库、数据库这些应用可能会产生较大的网络开销，IO开销，Cpu开销与内存开销。为了防止这类应用对其他应用产生影响，更也为了避免其他应用抢占系统资源，我们需要为该组件中的各个Pod设定好配额（limits），如果有必要的话，还需要在相关节点所在的node上划上污点与标签，以避免我们将资源开销大的Pod部署在该node上。

当业务规模较大或者机房分布较广时，单个镜像仓的设计很可能就不再适用了。视情况而定，可能需要在每一个机房中都部署一个时刻与主镜像仓同步的镜像仓副本。也可能需要为多地开发团队都部署一个可以读写的主镜像仓，并通过 push 事件保持一致。在此处我们只讨论单个镜像仓的高可用设计。

N.4.2.2 Redis高可用设计

镜像扫描也是企业级镜像仓库必不可少的的功能之一。但由于镜像数量往往非常的多，不论是 jfrog 的 xray 还是 coreos 的 clair 都会将需要扫描的镜像层压入一个队列中。这里我们以 Redis 为例，讲解如何为非关系型数据库实现高可用。



很多数据库为了实现高可用与数据一致性，都会给出相关同步方案和主从机制。Redis 官方给出的方式就是哨兵机制：在每一个 redis 进程所在的机器上再部署一个 sentinel 进程，这个进程会不断的检查集群中其他 redis 的健康状态，并在当前集群中的 Master 不可用时，从剩余的 Slave 中选举新的 Master 节点。

我们都知道，Deployment 中的 Pod 一旦重启，其 Pod 名称和 ip 都会发生改变。而在哨兵机制中，各个 Pod 之间始终需要互相通信，进而就需要为每一个 Pod 配置服务发现。然而要为每一个 Pod 配置 Service，我们需要用到其 Pod name。所以我们需要有一组稳定的标识符（Label）来帮助我们发现 Redis Pod。

这时候我们就可以用 StatefulSet 来部署包含有 redis 和 sentinel 的 Pod。这样一来，当我们的 Pod 出现故障而需要重启的时候，重启后这个 Pod 会保持此前的 Pod name 和 Pod ip。这样我们就可以通过为每一个 Pod 配置 Service 来为 sentinel 进程提供彼此之间的通信。

在 Redis 集群中，只有 Master 节点可以写入数据，而其他 Slave 只能读取数据。如果有写入请求被发送到 Slave 上，就会发生异常。为了保证能够通过一个地址始终访问到可用的 Master 节点，我们可以在 redis 上面套一层高可用的 Haproxy（无状态），并在其中配置健康检查。这样一来我们只需要通过 haproxy 的 4 层负载就可以总是访问到可用的 Redis Master 了。

所有的 Pod，不管是 Haproxy 还是 Redis，在启动时，都需要配置相关集群信息。而这部分信息往往会因为需要配置的集群规模，命名规则等原因产生变化。但我们不可能每次部署 Redis 集群时都为了这些配置文件重新打包镜像，所以我们可以通过 Kubernetes 将 configmap 以 volume 的形式挂载在容器中。这样，容器在读取文件时，就会读到我们在 configmap 中的配置文件了。

N.5 总结

通过这一章，我们初步认识了容器云平台以及其中应用的高可用架构设计。我们可以看到，不论是有状态的无状态的，还是平台或者应用的高可用架构，本质上其实都是通过增加冗余与故障转移来实现的。而实际上，不同的应用对于可用性的要求不同，甚至会为了性能牺牲可用性，这些都是在这一章中所无法顾及的，希望大家能够从业务场景出发，深刻理解需求后，再为应用设计架构。