typora-root-url: ./

1、Kubernetes入门

1.1 什么是Kubernetes

Kubernetes是容器集群管理系统，是一个开源的平台，可以实现容器集群的自动化部署、自动扩缩容、维护等功能。 Kubernetes是Google 2014年创建管理的，是Google 10多年大规模容器管理技术Borg的开源版本。

Kubernetes 这个单词的含义？k8s?

Kubernetes 这个单词来自于希腊语，含义是 舵手 或 领航员 。其词根是 governor 和 cybernetic。 K8s 是它的缩写，用 8 字替代了“ubernete”。

通过Kubernetes你可以：

- 快速部署应用
- 快速扩展应用
- 无缝对接新的应用功能
- 节省资源，优化硬件资源的使用

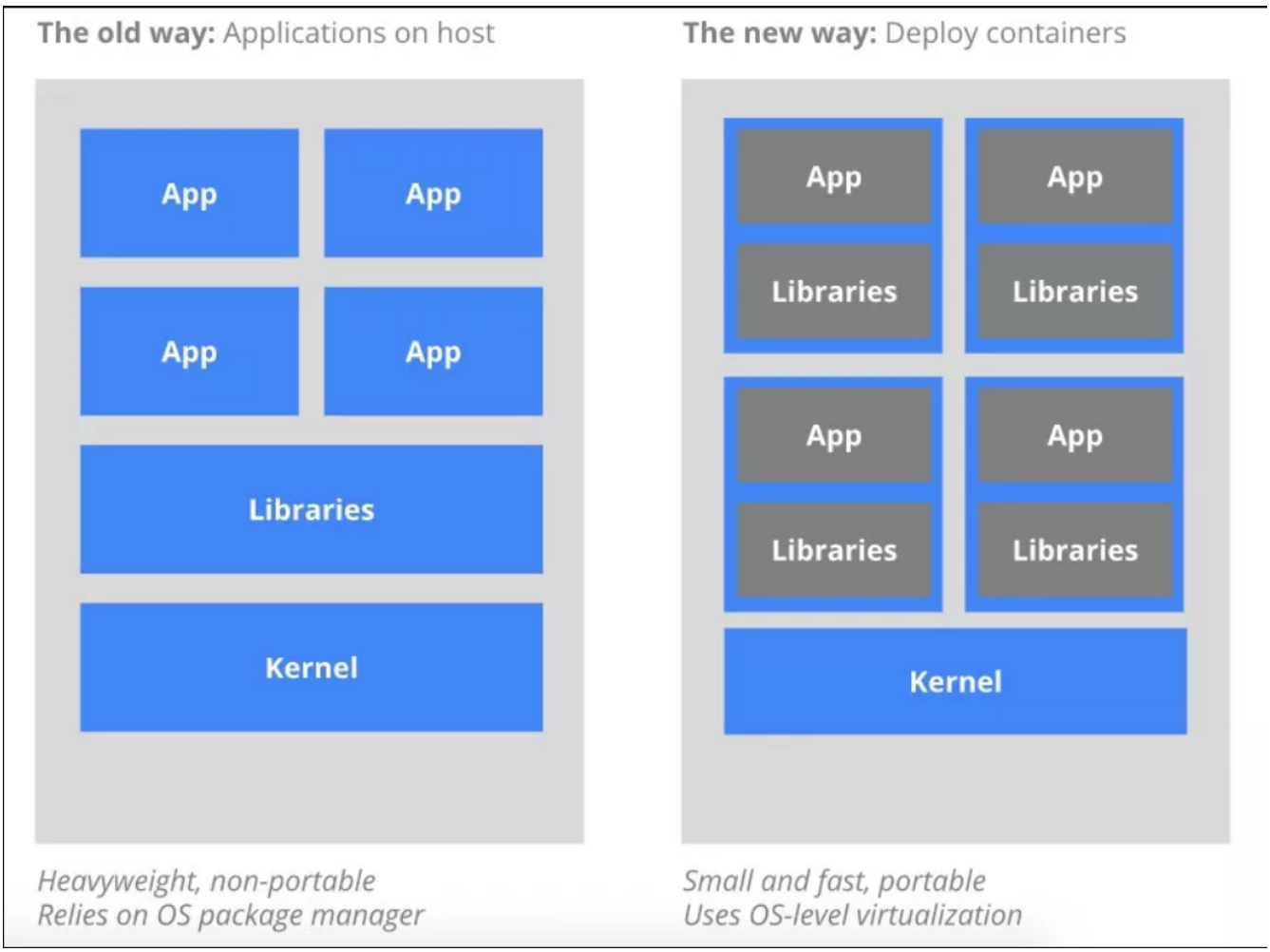
我们的目标是促进完善组件和工具的生态系统，以减轻应用程序在公有云或私有云中运行的负担。

Kubernetes 特点

- 可移植: 支持公有云，私有云，混合云，多重云（multi-cloud）
- 可扩展: 模块化, 插件化, 可挂载, 可组合
- 自动化: 自动部署，自动重启，自动复制，自动伸缩/扩展

1.2 为什么选择容器?

想要知道你为什么要选择使用容器?



程序部署的传统方法是指通过操作系统包管理器在主机上安装程序。这样做的缺点是，容易混淆程序之间以及程序和主机系统之间的可执行文件、配置文件、库、生命周期。为了达到精准展现和精准回撤，你可以搭建一台不可变的虚拟机镜像。但是虚拟机体量往往过于庞大而且不可转移。

容器部署的新的方式是基于操作系统级别的虚拟化，而非硬件虚拟化。容器彼此是隔离的，与宿主机也是隔离的：它们有自己的文件系统，彼此之间不能看到对方的进程，分配到的计算资源都是有限制的。它们比虚拟机更容易搭建。并且由于和基础架构、宿主机文件系统是解耦的，它们可以在不同类型的云上或操作系统上转移。

正因为容器又小又快，每一个容器镜像都可以打包装载一个程序。这种一对一的“程序 – 镜像”联系带给了容器诸多便捷。有了容器，静态容器镜像可以在编译/发布时期创建，而非部署时期。因此，每个应用不必再等待和整个应用栈其它部分进行整合，也不必和产品基础架构环境之间进行妥协。在编译/发布时期生成容器镜像建立了一个持续地把开发转化为产品的环境。相似地，容器远比虚拟机更加透明，尤其在设备监控和管理上。这一点，在容器的进程生命

周期被基础架构管理而非被容器内的进程监督器隐藏掉时，尤为显著。最终，随着每个容器内都装载了单一的程序，管理容器就等于管理或部署整个应用。

容器优势总结：

- 敏捷的应用创建与部署：相比虚拟机镜像，容器镜像的创建更简便、更高效。
- 持续的开发、集成，以及部署：在快速回滚下提供可靠、高频的容器镜像编译和部署（基于镜像的不可变性）。
- 开发与运营的关注点分离：由于容器镜像是在编译/发布期创建的，因此整个过程与基础架构解耦。
- 跨开发、测试、产品阶段的环境稳定性：在笔记本电脑上的运行结果和在云上完全一致。
- 在云平台与 OS 上分发的可转移性：可以在 Ubuntu、RHEL、CoreOS、预置系统、Google 容器引擎，乃至其它各类平台上运行。
- 以应用为核心的管理：从在虚拟硬件上运行系统，到在利用逻辑资源的系统上运行程序，从而提升了系统的抽象层级。
- 松散耦联、分布式、弹性、无拘束的微服务：整个应用被分散为更小、更独立的模块，并且这些模块可以被动态地部署和管理，而不再是存储在大型的单用途机器上的臃肿的单一应用栈。
- 资源隔离：增加程序表现的可预见性。
- 资源利用率：高效且密集。

1.3 为什么是Kubernetes

k8s目前是容器编排的标准，对于标准，是值得持续投入的。k8s也是istio的基础，istio在service mesh领域目前没有竞争，istio是knative的基础，knative在serverless领域目前没有竞争，也许5年之后knative才会普及流行，现在已经可以预见持续投入k8s的技术演进和技术红利 Kubernetes是一个十分强大的容器编排系统。它在全球范围内已广受使用，支持了一些很大型的部署，但同时它也伴随着一些代价。

Kubernetes 能在实体机或虚拟机集群上调度和运行程序容器。而且，Kubernetes 也能让开发者斩断联系着实体机或虚拟机的“锁链”，从以主机为中心的架构跃至以容器为中心的架构。该架构最终提供给开发者诸多内在的优势和便利。Kubernetes 提供给基础架构以真正的以容器为中心的开发环境。

Kubernetes 满足了一系列产品内运行程序的普通需求，诸如：

- 协调辅助进程，协助应用程序整合，维护一对一“程序 – 镜像”模型。
- 挂载存储系统
- 分布式机密信息
- 检查程序状态
- 复制应用实例
- 使用横向荚式自动缩放
- 命名与发现
- 负载均衡
- 滚动更新
- 资源监控
- 访问并读取日志
- 程序调试
- 提供验证与授权

以上兼具平台即服务（PaaS）的简化和基础架构即服务（IaaS）的灵活，并促进了在平台服务提供商之间的迁移。

Kubernetes 是一个什么样的平台?

虽然 Kubernetes 提供了非常多的功能，总会有更多受益于新特性的新场景出现。针对特定应用的工作流程，能被流水线化以加速开发速度。特别的编排起初是可接受的，这往往需要拥有健壮的大规模自动化机制。这也是为什么 Kubernetes 也被设计为一个构建组件和工具的生态系统的平台，使其更容易地部署、缩放、管理应用程序。

标签（label）可以让用户按照自己的喜好组织资源。 注释（annotation）让用户在资源里添加客户信息，以优化工作流程，为管理工具提供一个标示调试状态的简单方法。

此外，Kubernetes 控制面板是由开发者和用户均可使用的同样的 API 构建的。用户可以编写自己的控制器，比如 调度器（scheduler），使用可以被通用的命令行工具识别的他们自己的 API。

这种设计让大量的其它系统也能构建于 Kubernetes 之上。

Kubernetes 不是什么?

Kubernetes 不是传统的、全包容的平台即服务（Paas）系统。它尊重用户的选择，这很重要。

Kubernetes：

- 并不限制支持的程序类型。它并不检测程序的框架 (例如，Wildfly)，也不限制运行时支持的语言集合 (比如，Java、Python、Ruby)，也不仅仅迎合 12 因子应用程序，也不区分 应用 与 服务 。Kubernetes 旨在支持尽可能多种类的工作负载，包括无状态的、有状态的和处理数据的工作负载。如果某程序在容器内运行良好，它在 Kubernetes 上只可能运行地更好。
- 不提供中间件（例如消息总线）、数据处理框架（例如 Spark）、数据库（例如 mysql），也不把集群存储系统（例如 Ceph）作为内置服务。但是以上程序都可以在 Kubernetes 上运行。
- 没有“点击即部署”这类的服务市场存在。
- 不部署源代码，也不编译程序。持续集成 (CI) 工作流程是不同的用户和项目拥有其各自不同的需求和表现的地方。所以，Kubernetes 支持分层 CI 工作流程，却并不监听每层的工作状态。
- 允许用户自行选择日志、监控、预警系统。（ Kubernetes 提供一些集成工具以保证这一概念得到执行）
- 不提供也不管理一套完整的应用程序配置语言/系统（例如 jsonnet）。
- 不提供也不配合任何完整的机器配置、维护、管理、自我修复系统。

Kubernetes 运营在应用程序层面而不是在硬件层面，它提供了一些 PaaS 所通常提供的常见的适用功能，比如部署、伸缩、负载平衡、日志和监控。然而，Kubernetes 并非铁板一块，这些默认的解决方案是可供选择，可自行增加或删除的。

而且， Kubernetes 不只是一个编排系统 。事实上，它满足了编排的需求。 编排 的技术定义是，一个定义好的工作流程的执行：先做 A，再做 B，最后做 C。相反地， Kubernetes 囊括了一系列独立、可组合的控制流程，它们持续驱动当前状态向需求的状态发展。从 A 到 C 的具体过程并不唯一。集中化控制也并不是必须的；这种方式更像是编舞。这将使系统更易用、更高效、更健壮、复用性、扩展性更强。

特别是对于规模较小的团队而言，会因为维护它并且因它的学习曲线陡峭导致大量耗时。

1.4 典型案例

2 Kubernetes基本概念和术语

2.1 Master

k8s的控制节点扮演者整个调度和管理的角色，所以是非常关键的一部分。k8s的master节点主要包含三个部分：

1. **kube-apiserver** 提供了统一的资源操作入口;
2. **kube-scheduler** 是一个资源调度器，它根据特定的调度算法把pod生成到指定的计算节点中;
3. **kube-controller-manager** 也是运行在控制节点上一个很关键的管理控制组件;

kube-scheduler、kube-controller-manager 和 kube-apiserver 三者的功能紧密相关； 同时只能有一个 kube-scheduler、kube-controller-manager 进程处于工作状态，如果运行多个，则需要通过选举产生一个 leader；

2.2 Node

Node是Kubernetes中的工作节点，最开始被称为minion。一个Node可以是VM或物理机。每个Node（节点）具有运行pod的一些必要服务，并由Master组件进行管理，Node节点上的服务包括Docker、kubelet和kube-proxy。有关更多详细信息，请参考架构设计文档中的"Kubernetes Node"部分。

Node Status

节点的状态信息包含：

- Addresses
- ~~Phase~~ (已弃用)
- Condition
- Capacity
- Info

下面详细描述每个部分。

Addresses

这些字段的使用取决于云提供商或裸机配置。

- HostName：可以通过kubelet 中 --hostname-override参数覆盖。
- ExternalIP：可以被集群外部路由到的IP。
- InternalIP：只能在集群内进行路由的节点的IP地址。

Phase

不推荐使用，已弃用。

Condition

conditions字段描述所有Running节点的状态。

Node Condition	Description
OutOfDisk	True：如果节点上没有足够的可用空间来添加新的pod；否则为：False
Ready	True：如果节点是健康的并准备好接收pod；False：如果节点不健康并且不接受pod；Unknown：如果节点控制器在过去40秒内没有收到node的状态报告。
MemoryPressure	True：如果节点存储器上内存过低; 否则为：False。
DiskPressure	True：如果磁盘容量存在压力 - 也就是说磁盘容量低；否则为：False。

node condition被表示为一个JSON对象。例如，下面的响应描述了一个健康的节点。

```
"conditions": [
  {
    "kind": "Ready",
    "status": "True"
  }
]
```

如果Ready condition的Status是“Unknown” 或 “False”，比“pod-eviction-timeout”的时间长，则传递给“ kube-controller-manager”的参数，该节点上的所有Pod都将被节点控制器删除。默认的eviction timeout时间为5分钟。在某些情况下，当节点无法访问时，apiserver将无法与kubelet通信，删除Pod的需求不会传递到kubelet，直到重新与apiserver建立通信，这种情况下，计划删除的Pod会继续在划分的节点上运行。

在Kubernetes 1.5之前的版本中，节点控制器将强制从apiserver中删除这些不可达（上述情况）的pod。但是，在1.5及更高版本中，节点控制器在确认它们已经停止在集群中运行之前，不会强制删除Pod。可以看到这些可能在不可达节点上运行的pod处于"Terminating"或 “Unknown”。如果节点永久退出集群，Kubernetes是无法从底层基础架构辨别出来，则集群管理员需要手动删除节点对象，从Kubernetes删除节点对象会导致运行在上面的所有Pod对象从apiserver中删除，最终将会释放names。

Capacity

描述节点上可用的资源：CPU、内存和可以调度到节点上的最大pod数。

Info

关于节点的一些基础信息，如内核版本、Kubernetes版本（kubelet和kube-proxy版本）、Docker版本（如果有使用）、OS名称等。信息由Kubelet从节点收集。

Management

与 pods 和 services 不同，节点不是由Kubernetes 系统创建，它是由Google Compute Engine等云提供商在外部创建的，或使用物理和虚拟机。这意味着当Kubernetes创建一个节点时，它只是创建一个代表节点的对象，创建后，Kubernetes将检查节点是否有效。例如，如果使用以下内容创建一个节点：

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes将在内部创建一个节点对象，并通过基于metadata.name字段的健康检查来验证节点，如果节点有效，即所有必需的服务会同步运行，则才能在上面运行pod。请注意，Kubernetes将保留无效节点的对象（除非客户端有明确删除它）并且它将继续检查它是否变为有效。

目前，有三个组件与Kubernetes节点接口进行交互：节点控制器（node controller）、kubelet和kubectl。

Node Controller

节点控制器（Node Controller）是管理节点的Kubernetes master组件。

节点控制器在节点的生命周期中具有多个角色。第一个是在注册时将CIDR块分配给节点。

第二个是使节点控制器的内部列表与云提供商的可用机器列表保持最新。当在云环境中运行时，每当节点不健康时，节点控制器将询问云提供程序是否该节点的VM仍然可用，如果不可用，节点控制器会从其节点列表中删除该节点。

第三是监测节点的健康状况。当节点变为不可访问时，节点控制器负责将NodeStatus的NodeReady条件更新为ConditionUnknown，随后从节点中卸载所有pod ，如果节点继续无法访问，（默认超时时间为40 --node-monitor-period秒，开始报告ConditionUnknown，之后为5m开始卸载）。节点控制器按每秒来检查每个节点的状态。

在Kubernetes 1.4中，我们更新了节点控制器的逻辑，以更好地处理大量节点到达主节点的一些问题（例如，主节某些网络问题）。从1.4开始，节点控制器将在决定关于pod卸载的过程中会查看集群中所有节点的状态。

在大多数情况下，节点控制器将逐出速率限制为 --node-eviction-rate（默认为0.1）/秒，这意味着它不会每10秒从多于1个节点驱逐Pod。

当给定可用性的区域中的节点变得不健康时，节点逐出行为发生变化，节点控制器同时检查区域中节点的不健康百分比（NodeReady条件为ConditionUnknown或ConditionFalse）。如果不健康节点的比例为 --unhealthy-zone-threshold（默认为0.55），那么驱逐速度就会降低：如果集群很小（即小于或等于--large-cluster-size-threshold节点 - 默认值为50），则停止驱逐，否则， --secondary-node-eviction-rate（默认为0.01）每秒。这些策略在可用性区域内实现的原因是，一个可用性区域可能会从主分区中被分区，而其他可用区域则保持连接。如果集群没有跨多个云提供商可用性区域，那么只有一个可用区域(整个集群)。

在可用区域之间传播节点的一个主要原因是，当整个区域停止时，工作负载可以转移到健康区域。因此，如果区域中的所有节点都不健康，则节点控制器以正常速率逐出--node-eviction-rate。如所有的区域都是完全不健康的（即群集中没有健康的节点），在这种情况下，节点控制器会假设主连接有一些问题，并停止所有驱逐，直到某些连接恢复。

从Kubernetes 1.6开始，节点控制器还负责驱逐在节点上运行的NoExecutepod。

Self-Registration of Nodes

当kubelet flag --register-node为true（默认值）时，kubelet将向API服务器注册自身。这是大多数发行版使用的首选模式。

对于self-registration，kubelet从以下选项开始：

- `--api-servers` - Location of the apiservers.
- `--kubeconfig` - Path to credentials to authenticate itself to the apiserver.
- `--cloud-provider` - How to talk to a cloud provider to read metadata about itself.
- `--register-node` - Automatically register with the API server.
- `--register-with-taints` - Register the node with the given list of taints (comma separated `<key>=<value>:<effect>`). No-op if `register-node` is false.
- `--node-ip` IP address of the node.
- `--node-labels` - Labels to add when registering the node in the cluster.
- `--node-status-update-frequency` - Specifies how often kubelet posts node status to master.

手动管理节点

集群管理员可以创建和修改节点对象。

如果管理员希望手动创建节点对象，请设置kubelet flag --register-node=false。

管理员可以修改节点资源（不管--register-node设置如何），修改包括在节点上设置的labels 标签，并将其标记为不可调度的。

节点上的标签可以与pod上的节点选择器一起使用，以控制调度，例如将一个pod限制为只能在节点的子集上运行。

将节点标记为不可调度将防止新的pod被调度到该节点，但不会影响节点上的任何现有的pod，这在节点重新启动之前是有用的。例如，要标记节点不可调度，请运行以下命令：

```
kubectl cordon $NODENAME
```

注意，由daemonSet控制器创建的pod可以绕过Kubernetes调度程序，并且不遵循节点上无法调度的属性。

Node容量

节点的容量(cpu数量和内存数量)是节点对象的一部分。通常，节点在创建节点对象时注册并通知其容量。如果是手动管理节点，则需要在添加节点时设置节点容量。

Kubernetes调度程序可确保节点上的所有pod都有足够的资源。它会检查节点上容器的请求的总和不大于节点容量。

如果要明确保留非pod过程的资源，可以创建一个占位符pod。使用以下模板：

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-reserver
spec:
  containers:
  - name: sleep-forever
    image: gcr.io/google_containers/pause:0.8.0
    resources:
      requests:
        cpu: 100m
        memory: 100Mi
```

将cpu和内存值设置为你想要保留的资源量，将文件放置在manifest目录中(`--config=DIR` flag of kubelet)。在要预留资源的每个kubelet上执行此操作。

2.3 Pod

Pod是Kubernetes创建或部署的最小/最简单的基本单位

一个Pod封装一个应用容器（也可以有多个容器），存储资源、一个独立的网络IP以及管理控制容器运行方式的策略选项。Pod代表部署的一个单位：Kubernetes中单个应用的实例，它可能由单个容器或多个容器共享组成的资源。

> Docker是Kubernetes Pod中最常见的runtime，Pods也支持其他容器runtimes。

Kubernetes中的Pod使用可分两种主要方式：

- Pod中运行一个容器。“one-container-per-Pod”模式是Kubernetes最常见的用法; 在这种情况下，你可以将Pod视为单个封装的容器，但是Kubernetes是直接管理Pod而不是容器。
- Pods中运行多个需要一起工作的容器。Pod可以封装紧密耦合的应用，它们需要由多个容器组成，它们之间能够共享资源，这些容器可以形成一个单一的内部service单位 - 一个容器共享文件，另一个“sidecar”容器来更新

这些文件。Pod将这些容器的存储资源作为一个实体来管理。

关于Pod用法其他信息请参考：

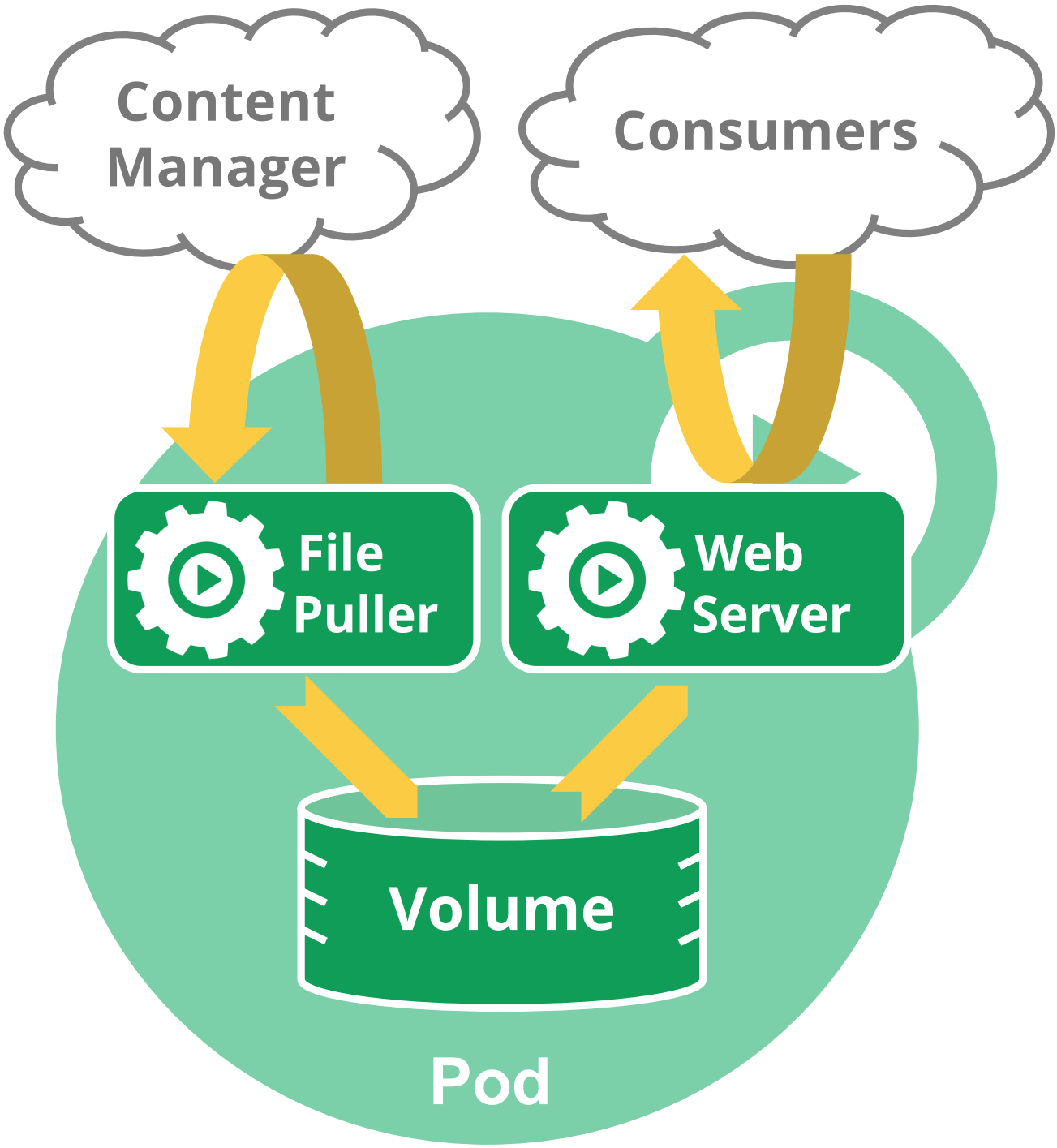
- The Distributed System Toolkit: Patterns for Composite Containers
- Container Design Patterns

每个Pod都是运行应用的单个实例，如果需要水平扩展应用（例如，运行多个实例），则应该使用多个Pods，每个实例一个Pod。在Kubernetes中，这样通常称为Replication。Replication的Pod通常由Controller创建和管理。更多信息，请参考Pods和控制器。

Pods如何管理多个容器

Pods的设计可用于支持多进程的协同工作（作为容器），形成一个cohesive的Service单位。Pod中的容器在集群中Node上被自动分配，容器之间可以共享资源、网络和相互依赖关系，并同时被调度使用。

请注意，在单个Pod中共同管理多个容器是一个相对高级的用法，应该只有在容器紧密耦合的特殊实例中使用此模式。例如，有一个容器被用作WEB服务器，用于共享volume，以及一个单独“sidecar”容器需要从远程获取资源来更新这些文件，如下图所示：



Pods提供两种共享资源：网络和存储。

网络

每个Pod被分配一个独立的IP地址，Pod中的每个容器共享网络命名空间，包括IP地址和网络端口。Pod内的容器可以使用localhost相互通信。当Pod中的容器与Pod 外部通信时，他们必须协调如何使用共享网络资源（如端口）。

存储

Pod可以指定一组共享存储*volumes*。Pod中的所有容器都可以访问共享*volumes*，允许这些容器共享数据。*volumes*还用于Pod中的数据持久化，以防其中一个容器需要重新启动而丢失数据。有关Kubernetes如何在Pod中实现共享存储的更多信息，请参考Volumes。

使用Pod

你很少会直接在kubernetes中创建单个Pod。因为Pod的生命周期是短暂的，用后即焚的实体。当Pod被创建后（不论是由你直接创建还是被其他Controller），都会被Kuberentes调度到集群的Node上。直到Pod的进程终止、被删掉、因为缺少资源而被驱逐、或者Node故障之前这个Pod都会一直保持在那个Node上。

> 注意：重启Pod中的容器跟重启Pod不是一回事。Pod只提供容器的运行环境并保持容器的运行状态，重启容器不会造成Pod重启。

Pod不会自愈。如果Pod运行的Node故障，或者是调度器本身故障，这个Pod就会被删除。同样的，如果Pod所在Node缺少资源或者Pod处于维护状态，Pod也会被驱逐。Kubernetes使用更高级的称为Controller的抽象层，来管理Pod实例。虽然可以直接使用Pod，但是在Kubernetes中通常是使用Controller来管理Pod的。

Pod和Controller

Controller可以创建和管理多个Pod，提供副本管理、滚动升级和集群级别的自愈能力。例如，如果一个Node故障，Controller就能自动将该节点上的Pod调度到其他健康的Node上。

包含一个或者多个Pod的Controller示例：

- Deployment
- StatefulSet
- DaemonSet

通常，Controller会用你提供的Pod Template来创建相应的Pod。

Pod模板

Pod模板是包含了其他对象（如Replication Controllers，Jobs和 DaemonSets）中的pod定义 。Controllers控制器使用Pod模板来创建实际需要的pod。

pod模板类似cookie cutters。“一旦饼干被切掉，饼干和刀将没有关系”。随后对模板的后续更改甚至切换到新模板对已创建的pod并没有任何的影响。

2.4 标签(Label)与标签选择器(Label Selector)

Labels其实就一对 key/value ，被关联到对象上，标签的使用我们倾向于能够标示对象的特殊特点，并且对用户而言是有意义的（就是一眼就看出了这个Pod是尼玛数据库），但是标签对内核系统是没有直接意义的。标签可以用来划分特定组的对象（比如，所有女的），标签可以在创建一个对象的时候直接给与，也可以在后期随时修改，每一个对象可以拥有多个标签，但是，key值必须是唯一的

```
"labels": {
  "key1" : "value1",
  "key2" : "value2"
}
```

我们最终会索引并且反向索引（reverse-index）labels，以获得更高效的查询和监视，把他们用到UI或者CLI中用来排序或者分组等等。我们不想用那些不具有指认效果的label来污染label，特别是那些体积较大和结构型的的数据。不具有指认效果的信息应该使用annotation来记录。

Motivation

Labels可以让用户将他们自己的有组织目的的结构以一种松耦合的方式应用到系统的对象上，且不需要客户端存放这些对应关系（mappings）。

服务部署和批处理管道通常是多维的实体（例如多个分区或者部署，多个发布轨道，多层，每层多微服务）。管理通常需要跨越式的切割操作，这会打破有严格层级展示关系的封装，特别对那些是由基础设施而非用户决定的很死板的层级关系。

示例标签：

- "release" : "stable"， "release" : "canary"
- "environment" : "dev"，"environment" : "qa"，"environment" : "production"
- "tier" : "frontend"，"tier" : "backend"，"tier" : "cache"
- "partition" : "customerA"， "partition" : "customerB"
- "track" : "daily"， "track" : "weekly"

这些只是常用Labels的例子，你可以按自己习惯来定义，需要注意，每个对象的标签key具有唯一性。

语法和字符集

Label其实是一对 key/value。有效的标签键有两个段：可选的前缀和名称，用斜杠（/）分隔，名称段是必需的，最多63个字符，以[a-z0-9A-Z]带有虚线（-）、下划线（_）、点（.）和开头和结尾必须是字母或数字（都是字符串形式）的形式组成。前缀是可选的。如果指定了前缀，那么必须是DNS子域：一系列的DNSlabel通过"."来划分，不超过253个字符，以斜杠（/）结尾。如果前缀被省略了，这个Label的key被假定为对用户私有的。自动化系统组件有（例如kube-scheduler，kube-controller-manager，kube-apiserver，kubectl，或其他第三方自动化），这些添加标签终端用户对象都必须指定一个前缀。Kuberentes.io 前缀是为Kubernetes 内核部分保留的。

有效的标签值最长为63个字符。要么为空，要么使用[a-z0-9A-Z]带有虚线（-）、下划线（_）、点（.）和开头和结尾必须是字母或数字（都是字符串形式）的形式组成。

Labels选择器

与Name和UID 不同，标签不需要有唯一性。一般来说，我们期望许多对象具有相同的标签。

通过标签选择器（Labels Selectors），客户端/用户 能方便辨识出一组对象。标签选择器是kubernetes中核心的组成部分。

API目前支持两种选择器：equality-based（基于平等）和set-based（基于集合）的。标签选择器可以由逗号分隔的多个requirements 组成。在多重需求的情况下，必须满足所有要求，因此逗号分隔符作为AND逻辑运算符。

一个为空的标签选择器（即有0个必须条件的选择器）会选择集合中的每一个对象。

一个null型标签选择器（仅对于可选的选择器字段才可能）不会返回任何对象。

注意：两个控制器的标签选择器不能在命名空间中重叠。

Equality-based requirement 基于相等的要求

基于相等的或者不相等的条件允许用标签的keys和values进行过滤。匹配的对象必须满足所有指定的标签约束，尽管他们可能也有额外的标签。有三种运算符是允许的，"="，"=="和"!="。前两种代表相等性（他们是同义运算符），后一种代表非相等性。例如：

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

第一个选择所有key等于 environment 值为 production 的资源。后一种选择所有key为 tier 值不等于 frontend 的资源，和那些没有key为 tier 的label的资源。要过滤所有处于 production 但不是 frontend 的资源，可以使用逗号操作符，

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

Set-based requirement

Set-based 的标签条件允许用一组value来过滤key。支持三种操作符: in， notin 和 exists(仅针对于key符号) 。例如：

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

第一个例子，选择所有key等于 environment，且value等于 production 或者 qa 的资源。 第二个例子，选择所有key等于 tier 且值是除了 frontend 和 backend 之外的资源，和那些没有标签的key是 tier 的资源。 第三个例子，选择所有有一个标签的key为partition的资源；value是什么不会被检查。 第四个例子，选择所有的没有lable的key名为partition 的资源；value是什么不会被检查。

类似的，逗号操作符相当于一个AND操作符。因而要使用一个 partition 键（不管value是什么），并且 environment 不是 qa 过滤资源可以用 partition,environment notin (qa) 。

Set-based 的选择器是一个相等性的宽泛的形式，因为 environment=production 相当于environment in (production)，与 != and notin 类似。

Set-based的条件可以与Equality-based的条件结合。例如， partition in (customerA,customerB),environment!=qa 。

API

LIST和WATCH过滤

LIST和WATCH操作可以指定标签选择器来过滤使用查询参数返回的对象集。这两个要求都是允许的（在这里给出，它们会出现在URL查询字符串中）：

LIST和WATCH操作，可以使用query参数来指定label选择器来过滤返回对象的集合。两种条件都可以使用：

- Set-based的要求：?labelSelector=environment%3Dproduction,tier%3Dfrontend
- Equality-based的要求：?labelSelector=environment+in+%28production%2Cqa%29%2Ctier+in+%28frontend%29

两个标签选择器样式都可用于通过REST客户端列出或观看资源。例如，apiserver使用kubectl和使用基于平等的人可以写：

两种标签选择器样式，都可以通过REST客户端来list或watch资源。比如使用 kubectl 来针对 apiserver ，并且使用 Equality-based的条件，可以用：

```
$ kubectl get pods -l environment=production,tier=frontend
```

或使用Set-based 要求：

```
$ kubectl get pods -l 'environment in (production),tier in (frontend)'
```

如已经提到的Set-based要求更具表现力。例如，它们可以对value执行OR运算：

```
$ kubectl get pods -l 'environment in (production, qa)'
```

或者通过exists操作符进行否定限制匹配：

```
$ kubectl get pods -l 'environment,environment notin (frontend)'
```

API对象中引用

一些Kubernetes对象，例如services和replicationcontrollers，也使用标签选择器来指定其他资源的集合，如pod。

Service和ReplicationController

一个service针对的pods的集合是用标签选择器来定义的。类似的，一个replicationcontroller管理的pods的群体也是用标签选择器来定义的。

对于这两种对象的Label选择器是用map定义在json或者yaml文件中的，并且只支持Equality-based的条件：

```
"selector": {
    "component" : "redis",
}
```

要么

```
selector:
    component: redis
```

此选择器（分别为json或yaml格式）等同于component=redis或component in (redis)。

支持set-based要求的资源

Job，Deployment，Replica Set，和Daemon Set，支持set-based要求。

```
selector:
  matchLabels:
    component: redis
  matchExpressions:
    - {key: tier, operator: In, values: [cache]}
    - {key: environment, operator: NotIn, values: [dev]}
```

matchLabels 是一个{key,value}的映射。一个单独的 {key,value} 相当于 matchExpressions 的一个元素，它的key字段是"key",操作符是 In ，并且value数组value包含"value"。 matchExpressions 是一个pod的选择器条件的list 。有效运算符包含In, NotIn, Exists, 和DoesNotExist。在In和NotIn的情况下，value的组必须不能为空。所有的条件，包含 matchLabels andmatchExpressions 中的，会用AND符号连接，他们必须都被满足以完成匹配。

2.5 Replication Controller(RC)

注意：建议使用Deployment 配置 ReplicaSet （简称RS）方法来控制副本数。

ReplicationController（简称RC）是确保用户定义的Pod副本数保持不变。

ReplicationController 工作原理

在用户定义范围内，如果pod增多，则ReplicationController会终止额外的pod，如果减少，RC会创建新的pod，始终保持在定义范围。例如，RC会在Pod维护（例如内核升级）后在节点上重新创建新Pod。

注：

- ReplicationController会替换由于某些原因而被删除或终止的pod，例如在节点故障或中断节点维护（例如内核升级）的情况下。因此，即使应用只需要一个pod，我们也建议使用ReplicationController。
- RC跨多个Node节点监视多个pod。

示例：

replication.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

下载示例文件然后运行：

```
$ kubectl create -f ./replication.yaml
replicationcontroller "nginx" created
```

检查ReplicationController状态：

```
$ kubectl describe replicationcontrollers/nginx
Name:        nginx
Namespace:   default
```

```
Image(s):    nginx
Selector:    app=nginx
Labels:      app=nginx
Replicas:    3 current / 3 desired
Pods Status: 0 Running / 3 Waiting / 0 Succeeded / 0 Failed
Events:
  FirstSeen       LastSeen     Count    From                        SubobjectPath    Type
   Reason              Message
  ---------       --------     -----    ----                        -------------    ----
   ------              -------
  20s             20s          1        {replication-controller }                    Normal
   SuccessfulCreate    Created pod: nginx-qrm3m
  20s             20s          1        {replication-controller }                    Normal
   SuccessfulCreate    Created pod: nginx-3ntk0
  20s             20s          1        {replication-controller }                    Normal
   SuccessfulCreate    Created pod: nginx-4ok8v
```

创建了三个pod：

```
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
```

列出属于ReplicationController的所有pod：

```
$ pods=$(kubectl get pods --selector=app=nginx --output=jsonpath={.items..metadata.name})
echo $pods
nginx-3ntk0 nginx-4ok8v nginx-qrm3m
```

删除ReplicationController及其Pods

使用kubectl delete命令删除ReplicationController及其所有pod。

当使用REST API或客户端库时，需要明确地执行这些步骤（将副本缩放为0，等待pod删除，然后删除ReplicationController）。

只删除 ReplicationController

在删除ReplicationController时，可以不影响任何pod。

使用kubectl，为kubectl delete指定- cascade = false选项。

使用REST API或go客户端库时，只需删除ReplicationController对象即可。

原始文件被删除后，你可以创建一个新的ReplicationController来替换它。只要旧的和新.spec.selector 相匹配，那么新的将会采用旧的Pod。

ReplicationController隔离pod

可以通过更改标签来从ReplicationController的目标集中删除Pod。

RC常用方式

- Rescheduling（重新规划）

- 扩展
- 滚动更新
- 多版本跟踪
- 使用ReplicationControllers与关联的Services

API对象

Replication controller是Kubernetes REST API中的顶级资源。有关API对象更多详细信息，请参见：ReplicationController API对象。

RC 替代方法

ReplicaSet

ReplicaSet是支持新的set-based选择器要求的下一代ReplicationController 。它主要用作Deployment协调pod创建、删除和更新。请注意，除非需要自定义更新编排或根本不需要更新，否则建议使用Deployment而不是直接使用ReplicaSets。

Deployment（推荐）

Deployment是一个高级的API对象，以类似的方式更新其底层的副本集和它们的Pods kubectl rolling-update。如果您希望使用这种滚动更新功能，建议您进行部署，因为kubectl rolling-update它们是声明式的，服务器端的，并具有其他功能。

Bare Pods

与用户直接创建pod的情况不同，ReplicationController会替换由于某些原因而被删除或终止的pod，例如在节点故障或中断节点维护（例如内核升级）的情况下。因此，即使应用只需要一个pod，我们也建议使用ReplicationController。

2.6 Deployment

Deployment为Pod和Replica Set（升级版的 Replication Controller）提供声明式更新。

你只需要在 Deployment 中描述您想要的目标状态是什么，Deployment controller 就会帮您将 Pod 和ReplicaSet 的实际状态改变到您的目标状态。您可以定义一个全新的 Deployment 来创建 ReplicaSet 或者删除已有的 Deployment 并创建一个新的来替换。

注意：您不该手动管理由 Deployment 创建的 Replica Set，否则您就篡越了 Deployment controller 的职责！下文罗列了 Deployment 对象中已经覆盖了所有的用例。如果未有覆盖您所有需要的用例，请直接在 Kubernetes 的代码库中提 issue。

典型的用例如下：

- 使用Deployment来创建ReplicaSet。ReplicaSet在后台创建pod。检查启动状态，看它是成功还是失败。
- 然后，通过更新Deployment的PodTemplateSpec字段来声明Pod的新状态。这会创建一个新的ReplicaSet，Deployment会按照控制的速率将pod从旧的ReplicaSet移动到新的ReplicaSet中。
- 如果当前状态不稳定，回滚到之前的Deployment revision。每次回滚都会更新Deployment的revision。
- 扩容Deployment以满足更高的负载。
- 暂停Deployment来应用PodTemplateSpec的多个修复，然后恢复上线。
- 根据Deployment 的状态判断上线是否hang住了。

- 清除旧的不必要的 ReplicaSet。

创建 Deployment

下面是一个 Deployment 示例，它创建了一个 ReplicaSet 来启动3个 nginx pod。

下载示例文件并执行命令：

```
$ kubectl create -f https://kubernetes.io/docs/user-guide/nginx-deployment.yaml --record
deployment "nginx-deployment" created
```

将kubectl的 --record 的 flag 设置为 true可以在 annotation 中记录当前命令创建或者升级了该资源。这在未来会很有用，例如，查看在每个 Deployment revision 中执行了哪些命令。

然后立即执行 get 将获得如下结果：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         0         0            0           1s
```

输出结果表明我们希望的repalica数是3（根据deployment中的.spec.replicas配置）当前replica数（.status.replicas）是0, 最新的replica数（.status.updatedReplicas）是0，可用的replica数（.status.availableReplicas）是0。

过几秒后再执行get命令，将获得如下输出：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           18s
```

我们可以看到Deployment已经创建了3个 replica，所有的 replica 都已经是最新的了（包含最新的pod template），可用的（根据Deployment中的.spec.minReadySeconds声明，处于已就绪状态的pod的最少个数）。执行kubectl get rs和kubectl get pods会显示Replica Set（RS）和Pod已创建。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-2035384211   3         3         0       18s
```

您可能会注意到 ReplicaSet 的名字总是<Deployment的名字>-<pod template的hash值>。

```
$ kubectl get pods --show-labels
NAME                                READY     STATUS    RESTARTS   AGE       LABELS
nginx-deployment-2035384211-7ci7o   1/1       Running   0          18s       app=nginx,pod-
template-hash=2035384211
nginx-deployment-2035384211-kzszj   1/1       Running   0          18s       app=nginx,pod-
template-hash=2035384211
nginx-deployment-2035384211-qqcnn   1/1       Running   0          18s       app=nginx,pod-
template-hash=2035384211
```

刚创建的Replica Set将保证总是有3个 nginx 的 pod 存在。

注意： 您必须在 Deployment 中的 selector 指定正确的 pod template label（在该示例中是 app = nginx），不要跟其他的 controller 的 selector 中指定的 pod template label 搞混了（包括 Deployment、Replica Set、Replication Controller 等）。Kubernetes 本身并不会阻止您任意指定 pod template label，但是如果您真的这么做了，这些 controller 之间会相互打架，并可能导致不正确的行为。

Pod-template-hash label

注意：这个 label 不是用户指定的！

注意上面示例输出中的 pod label 里的 pod-template-hash label。当 Deployment 创建或者接管 ReplicaSet 时，Deployment controller 会自动为 Pod 添加 pod-template-hash label。这样做的目的是防止 Deployment 的子 ReplicaSet 的 pod 名字重复。通过将 ReplicaSet 的 PodTemplate 进行哈希散列，使用生成的哈希值作为 label 的值，并添加到 ReplicaSet selector 里、 pod template label 和 ReplicaSet 管理中的 Pod 上。

更新Deployment

注意： Deployment 的 rollout 当且仅当 Deployment 的 pod template（例如.spec.template）中的label更新或者镜像更改时被触发。其他更新，例如扩容Deployment不会触发 rollout。

假如我们现在想要让 nginx pod 使用nginx:1.9.1的镜像来代替原来的nginx:1.7.9的镜像。

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
deployment "nginx-deployment" image updated
```

我们可以使用edit命令来编辑 Deployment，修改 .spec.template.spec.containers[0].image ，将nginx:1.7.9 改写成 nginx:1.9.1。

```
$ kubectl edit deployment/nginx-deployment
deployment "nginx-deployment" edited
```

查看 rollout 的状态，只要执行：

```
$ kubectl rollout status deployment/nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment "nginx-deployment" successfully rolled out
```

Rollout 成功后，get Deployment：

```
$ kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           36s
```

UP-TO-DATE 的 replica 的数目已经达到了配置中要求的数目。

CURRENT 的 replica 数表示 Deployment 管理的 replica 数量，AVAILABLE 的 replica 数是当前可用的replica数量。

我们通过执行kubectl get rs可以看到 Deployment 更新了Pod，通过创建一个新的 ReplicaSet 并扩容了3个 replica，同时将原来的 ReplicaSet 缩容到了0个 replica。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   3         3         0       6s
nginx-deployment-2035384211   0         0         0       36s
```

执行 get pods只会看到当前的新的 pod:

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-1564180365-khku8   1/1       Running   0          14s
nginx-deployment-1564180365-nacti   1/1       Running   0          14s
nginx-deployment-1564180365-z9gth   1/1       Running   0          14s
```

下次更新这些 pod 的时候，只需要更新 Deployment 中的 pod 的 template 即可。

Deployment 可以保证在升级时只有一定数量的 Pod 是 down 的。默认的，它会确保至少有比期望的Pod数量少一个是up状态（最多一个不可用）。

Deployment 同时也可以确保只创建出超过期望数量的一定数量的 Pod。默认的，它会确保最多比期望的Pod数量多一个的 Pod 是 up 的（最多1个 surge ）。

在未来的 Kuberentes 版本中，将从1-1变成25%-25%。

例如，如果您自己看下上面的 Deployment，您会发现，开始创建一个新的 Pod，然后删除一些旧的 Pod 再创建一个新的。当新的Pod创建出来之前不会杀掉旧的Pod。这样能够确保可用的 Pod 数量至少有2个，Pod的总数最多4个。

```
$ kubectl describe deployments
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 12:01:06 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     <none>
NewReplicaSet:      nginx-deployment-1564180365 (3/3 replicas created)
Events:
  FirstSeen LastSeen    Count   From                    SubobjectPath   Type        Reason
            Message
  --------- --------    -----   ----                    -------------   --------    ------
            -------
  36s       36s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-2035384211 to 3
  23s       23s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 1
  23s       23s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 2
  23s       23s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 2
```

```
  21s       21s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 0
  21s       21s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 3
```

我们可以看到当我们刚开始创建这个 Deployment 的时候，创建了一个 ReplicaSet（nginx-deployment-2035384211），并直接扩容到了3个 replica。

当我们更新这个 Deployment 的时候，它会创建一个新的 ReplicaSet（nginx-deployment-1564180365），将它扩容到1个replica，然后缩容原先的 ReplicaSet 到2个 replica，此时满足至少2个 Pod 是可用状态，同一时刻最多有4个 Pod 处于创建的状态。

接着继续使用相同的 rolling update 策略扩容新的 ReplicaSet 和缩容旧的 ReplicaSet。最终，将会在新的 ReplicaSet 中有3个可用的 replica，旧的 ReplicaSet 的 replica 数目变成0。

Rollover（多个rollout并行）

每当 Deployment controller 观测到有新的 deployment 被创建时，如果没有已存在的 ReplicaSet 来创建期望个数的 Pod 的话，就会创建出一个新的 ReplicaSet 来做这件事。已存在的 ReplicaSet 控制 label 与.spec.selector匹配但是 template 跟.spec.template不匹配的 Pod 缩容。最终，新的 ReplicaSet 将会扩容出.spec.replicas指定数目的 Pod，旧的 ReplicaSet 会缩容到0。

如果您更新了一个的已存在并正在进行中的 Deployment，每次更新 Deployment都会创建一个新的 ReplicaSet并扩容它，同时回滚之前扩容的 ReplicaSet ——将它添加到旧的 ReplicaSet 列表中，开始缩容。

例如，假如您创建了一个有5个niginx:1.7.9 replica的 Deployment，但是当还只有3个nginx:1.7.9的 replica 创建出来的时候您就开始更新含有5个nginx:1.9.1 replica 的 Deployment。在这种情况下，Deployment 会立即杀掉已创建的3个nginx:1.7.9的 Pod，并开始创建nginx:1.9.1的 Pod。它不会等到所有的5个nginx:1.7.9的 Pod 都创建完成后才开始改变航道。

Label selector 更新

我们通常不鼓励更新 label selector，我们建议实现规划好您的 selector。

任何情况下，只要您想要执行 label selector 的更新，请一定要谨慎并确认您已经预料到所有可能因此导致的后果。

- 增添 selector 需要同时在 Deployment 的 spec 中更新新的 label，否则将返回校验错误。此更改是不可覆盖的，这意味着新的 selector 不会选择使用旧 selector 创建的 ReplicaSet 和 Pod，从而导致所有旧版本的 ReplicaSet 都被丢弃，并创建新的 ReplicaSet。
- 更新 selector，即更改 selector key 的当前值，将导致跟增添 selector 同样的后果。
- 删除 selector，即删除 Deployment selector 中的已有的 key，不需要对 Pod template label 做任何更改，现有的 ReplicaSet 也不会成为孤儿，但是请注意，删除的 label 仍然存在于现有的 Pod 和 ReplicaSet 中。

回退Deployment

有时候您可能想回退一个 Deployment，例如，当 Deployment 不稳定时，比如一直 crash looping。

默认情况下，kubernetes 会在系统中保存前两次的 Deployment 的 rollout 历史记录，以便您可以随时回退（您可以修改revision history limit来更改保存的revision数）。

注意： 只要 Deployment 的 rollout 被触发就会创建一个 revision。也就是说当且仅当 Deployment 的 Pod template（如.spec.template）被更改，例如更新template 中的 label 和容器镜像时，就会创建出一个新的 revision。

其他的更新，比如扩容 Deployment 不会创建 revision——因此我们可以很方便的手动或者自动扩容。这意味着当您回退到历史 revision 是，直有 Deployment 中的 Pod template 部分才会回退。

假设我们在更新 Deployment 的时候犯了一个拼写错误，将镜像的名字写成了nginx:1.91，而正确的名字应该是nginx:1.9.1：

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.91
deployment "nginx-deployment" image updated
```

Rollout 将会卡住。

```
$ kubectl rollout status deployments nginx-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

按住 Ctrl-C 停止上面的 rollout 状态监控。

您会看到旧的 replica（nginx-deployment-1564180365 和 nginx-deployment-2035384211）和新的 replica（nginx-deployment-3066724191）数目都是2个。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-1564180365   2         2         0       25s
nginx-deployment-2035384211   0         0         0       36s
nginx-deployment-3066724191   2         2         2       6s
```

看下创建 Pod，您会看到有两个新的 ReplicaSet 创建的 Pod 处于 ImagePullBackOff 状态，循环拉取镜像。

```
$ kubectl get pods
NAME                                READY     STATUS             RESTARTS   AGE
nginx-deployment-1564180365-70iae   1/1       Running            0          25s
nginx-deployment-1564180365-jbqqo   1/1       Running            0          25s
nginx-deployment-3066724191-08mng   0/1       ImagePullBackOff   0          6s
nginx-deployment-3066724191-eocby   0/1       ImagePullBackOff   0          6s
```

注意，Deployment controller会自动停止坏的 rollout，并停止扩容新的 ReplicaSet。

```
$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       2 updated | 3 total | 2 available | 2 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     nginx-deployment-1564180365 (2/2 replicas created)
NewReplicaSet:      nginx-deployment-3066724191 (2/2 replicas created)
Events:
  FirstSeen LastSeen    Count   From                    SubobjectPath   Type        Reason
            Message
```

```
  --------- --------    -----   ----                    -------------   --------    ------
            -------
  1m        1m          1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-2035384211 to 3
  22s       22s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 1
  22s       22s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 2
  22s       22s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 2
  21s       21s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 0
  21s       21s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 3
  13s       13s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-3066724191 to 1
  13s       13s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-1564180365 to 2
  13s       13s         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-3066724191 to 2
```

为了修复这个问题，我们需要回退到稳定的 Deployment revision。

检查 Deployment 升级的历史记录

首先，检查下 Deployment 的 revision：

```
$ kubectl rollout history deployment/nginx-deployment
deployments "nginx-deployment":
REVISION    CHANGE-CAUSE
1           kubectl create -f https://kubernetes.io/docs/user-guide/nginx-deployment.yaml--
record
2           kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
3           kubectl set image deployment/nginx-deployment nginx=nginx:1.91
```

因为我们创建 Deployment 的时候使用了--recored参数可以记录命令，我们可以很方便的查看每次 revision 的变化。

查看单个revision 的详细信息：

```
$ kubectl rollout history deployment/nginx-deployment --revision=2
deployments "nginx-deployment" revision 2
  Labels:       app=nginx
          pod-template-hash=1159050644
  Annotations:  kubernetes.io/change-cause=kubectl set image deployment/nginx-deployment
nginx=nginx:1.9.1
  Containers:
   nginx:
    Image:      nginx:1.9.1
    Port:       80/TCP
     QoS Tier:
        cpu:      BestEffort
```

```
        memory:   BestEffort
    Environment Variables:      <none>
  No volumes.
```

回退到历史版本

现在，我们可以决定回退当前的 rollout 到之前的版本：

```
$ kubectl rollout undo deployment/nginx-deployment
deployment "nginx-deployment" rolled back
```

也可以使用 --revision参数指定某个历史版本：

```
$ kubectl rollout undo deployment/nginx-deployment --to-revision=2
deployment "nginx-deployment" rolled back
```

与 rollout 相关的命令详细文档见kubectl rollout。

该 Deployment 现在已经回退到了先前的稳定版本。如您所见，Deployment controller产生了一个回退到revison 2的DeploymentRollback的 event。

```
$ kubectl get deployment
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           30m

$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp:  Tue, 15 Mar 2016 14:48:04 -0700
Labels:         app=nginx
Selector:       app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailable
StrategyType:       RollingUpdate
MinReadySeconds:    0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:     <none>
NewReplicaSet:      nginx-deployment-1564180365 (3/3 replicas created)
Events:
  FirstSeen LastSeen    Count   From                    SubobjectPath   Type        Reason
            Message
  --------- --------    -----   ----                    -------------   --------    ------
            -------
  30m       30m         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-2035384211 to 3
  29m       29m         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 1
  29m       29m         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 2
  29m       29m         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 2
```

```
  29m       29m         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-2035384211 to 0
  29m       29m         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-3066724191 to 2
  29m       29m         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-3066724191 to 1
  29m       29m         1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-1564180365 to 2
  2m        2m          1       {deployment-controller }                Normal
ScalingReplicaSet   Scaled down replica set nginx-deployment-3066724191 to 0
  2m        2m          1       {deployment-controller }                Normal
DeploymentRollback  Rolled back deployment "nginx-deployment" to revision 2
  29m       2m          2       {deployment-controller }                Normal
ScalingReplicaSet   Scaled up replica set nginx-deployment-1564180365 to 3
```

清理 Policy

您可以通过设置.spec.revisonHistoryLimit项来指定 deployment 最多保留多少 revision 历史记录。默认的会保留所有的 revision；如果将该项设置为0，Deployment就不允许回退了。

Deployment 扩容

您可以使用以下命令扩容 Deployment：

```
$ kubectl scale deployment nginx-deployment --replicas 10
deployment "nginx-deployment" scaled
```

假设您的集群中启用了horizontal pod autoscaling，您可以给 Deployment 设置一个 autoscaler，基于当前 Pod的 CPU 利用率选择最少和最多的 Pod 数。

```
$ kubectl autoscale deployment nginx-deployment --min=10 --max=15 --cpu-percent=80
deployment "nginx-deployment" autoscaled
```

比例扩容

RollingUpdate Deployment 支持同时运行一个应用的多个版本。或者 autoscaler 扩 容 RollingUpdate Deployment 的时候，正在中途的 rollout（进行中或者已经暂停的），为了降低风险，Deployment controller 将会平衡已存在的活动中的 ReplicaSet（有 Pod 的 ReplicaSet）和新加入的 replica。这被称为比例扩容。

例如，您正在运行中含有10个 replica 的 Deployment。maxSurge=3，maxUnavailable=2。

```
$ kubectl get deploy
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment    10        10        10           10          50s
```

您更新了一个镜像，而在集群内部无法解析。

```
$ kubectl set image deploy/nginx-deployment nginx=nginx:sometag
deployment "nginx-deployment" image updated
```

镜像更新启动了一个包含ReplicaSet nginx-deployment-1989198191的新的rollout，但是它被阻塞了，因为我们上面提到的maxUnavailable。

```
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-1989198191   5         5         0         9s
nginx-deployment-618515232    8         8         8         1m
```

然后发起了一个新的Deployment扩容请求。autoscaler将Deployment的repllica数目增加到了15个。Deployment controller需要判断在哪里增加这5个新的replica。如果我们没有谁用比例扩容，所有的5个replica都会加到一个新的ReplicaSet中。如果使用比例扩容，新添加的replica将传播到所有的ReplicaSet中。大的部分加入replica数最多的ReplicaSet中，小的部分加入到replica数少的ReplciaSet中。0个replica的ReplicaSet不会被扩容。

在我们上面的例子中，3个replica将添加到旧的ReplicaSet中，2个replica将添加到新的ReplicaSet中。rollout进程最终会将所有的replica移动到新的ReplicaSet中，假设新的replica成为健康状态。

```
$ kubectl get deploy
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment    15        18        7            8           7m
$ kubectl get rs
NAME                          DESIRED   CURRENT   READY     AGE
nginx-deployment-1989198191   7         7         0         7m
nginx-deployment-618515232    11        11        11        7m
```

暂停和恢复Deployment

您可以在发出一次或多次更新前暂停一个 Deployment，然后再恢复它。这样您就能多次暂停和恢复 Deployment，在此期间进行一些修复工作，而不会发出不必要的 rollout。

例如使用刚刚创建 Deployment：

```
$ kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     3         3         3            3           1m
[mkargaki@dhcp129-211 kubernetes]$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         1m
```

使用以下命令暂停 Deployment：

```
$ kubectl rollout pause deployment/nginx-deployment
deployment "nginx-deployment" paused
```

然后更新 Deplyment中的镜像：

```
$ kubectl set image deploy/nginx nginx=nginx:1.9.1
deployment "nginx-deployment" image updated
```

注意新的 rollout 启动了：

```
$ kubectl rollout history deploy/nginx
deployments "nginx"
REVISION  CHANGE-CAUSE
1   <none>

$ kubectl get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   3         3         3         2m
```

您可以进行任意多次更新，例如更新使用的资源：

```
$ kubectl set resources deployment nginx -c=nginx --limits=cpu=200m,memory=512Mi
deployment "nginx" resource requirements updated
```

Deployment 暂停前的初始状态将继续它的功能，而不会对 Deployment 的更新产生任何影响，只要 Deployment是暂停的。

最后，恢复这个 Deployment，观察完成更新的 ReplicaSet 已经创建出来了：

```
$ kubectl rollout resume deploy nginx
deployment "nginx" resumed
$ KUBECTL get rs -w
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   2         2         2         2m
nginx-3926361531   2         2         0         6s
nginx-3926361531   2         2         1         18s
nginx-2142116321   1         2         2         2m
nginx-2142116321   1         2         2         2m
nginx-3926361531   3         2         1         18s
nginx-3926361531   3         2         1         18s
nginx-2142116321   1         1         1         2m
nginx-3926361531   3         3         1         18s
nginx-3926361531   3         3         2         19s
nginx-2142116321   0         1         1         2m
nginx-2142116321   0         1         1         2m
nginx-2142116321   0         0         0         2m
nginx-3926361531   3         3         3         20s
^C
$ KUBECTL get rs
NAME               DESIRED   CURRENT   READY     AGE
nginx-2142116321   0         0         0         2m
nginx-3926361531   3         3         3         28s
```

注意： 在恢复 Deployment 之前您无法回退一个已经暂停的 Deployment。

Deployment 状态

Deployment 在生命周期中有多种状态。在创建一个新的 ReplicaSet 的时候它可以是 progressing 状态， complete 状态，或者 fail to progress 状态。

进行中的 Deployment

Kubernetes 将执行过下列任务之一的 Deployment 标记为 progressing 状态：

- Deployment 正在创建新的ReplicaSet过程中。
- Deployment 正在扩容一个已有的 ReplicaSet。
- Deployment 正在缩容一个已有的 ReplicaSet。
- 有新的可用的 pod 出现。

您可以使用kubectl rollout status命令监控 Deployment 的进度。

完成的 Deployment

Kubernetes 将包括以下特性的 Deployment 标记为 complete 状态：

- Deployment 最小可用。最小可用意味着 Deployment 的可用 replica 个数等于或者超过 Deployment 策略中的期望个数。
- 所有与该 Deployment 相关的replica都被更新到了您指定版本，也就说更新完成。
- 该 Deployment 中没有旧的 Pod 存在。

您可以用kubectl rollout status命令查看 Deployment 是否完成。如果 rollout 成功完成，kubectl rollout status将返回一个0值的 Exit Code。

```
$ kubectl rollout status deploy/nginx
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment "nginx" successfully rolled out
$ echo $?
0
```

失败的 Deployment

您的 Deployment 在尝试部署新的 ReplicaSet 的时候可能卡住，用于也不会完成。这可能是因为以下几个因素引起的：

- 无效的引用
- 不可读的 probe failure
- 镜像拉取错误
- 权限不够
- 范围限制
- 程序运行时配置错误

探测这种情况的一种方式是，在您的 Deployment spec 中指定spec.progressDeadlineSeconds。spec.progressDeadlineSeconds 表示 Deployment controller 等待多少秒才能确定（通过 Deployment status）Deployment进程是卡住的。

下面的kubectl命令设置progressDeadlineSeconds 使 controller 在 Deployment 在进度卡住10分钟后报告：

```
$ kubectl patch deployment/nginx-deployment -p '{"spec":{"progressDeadlineSeconds":600}}'
"nginx-deployment" patched
```

当超过截止时间后，Deployment controller 会在 Deployment 的 status.conditions中增加一条 DeploymentCondition，它包括如下属性：

- Type=Progressing

- Status=False
- Reason=ProgressDeadlineExceeded

浏览 Kubernetes API conventions 查看关于status conditions的更多信息。

注意： kubernetes除了报告Reason=ProgressDeadlineExceeded状态信息外不会对卡住的 Deployment 做任何操作。更高层次的协调器可以利用它并采取相应行动，例如，回滚 Deployment 到之前的版本。

注意： 如果您暂停了一个 Deployment，在暂停的这段时间内kubernetnes不会检查您指定的 deadline。您可以在 Deployment 的 rollout 途中安全的暂停它，然后再恢复它，这不会触发超过deadline的状态。

您可能在使用 Deployment 的时候遇到一些短暂的错误，这些可能是由于您设置了太短的 timeout，也有可能是因为各种其他错误导致的短暂错误。例如，假设您使用了无效的引用。当您 Describe Deployment 的时候可能会注意到如下信息：

```
$ kubectl describe deployment nginx-deployment
<...>
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     True    ReplicaSetUpdated
  ReplicaFailure  True    FailedCreate
<...>
```

执行 kubectl get deployment nginx-deployment -o yaml，Deployement 的状态可能看起来像这个样子：

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: Replica set "nginx-deployment-4262182780" is progressing.
    reason: ReplicaSetUpdated
    status: "True"
    type: Progressing
  - lastTransitionTime: 2016-10-04T12:25:42Z
    lastUpdateTime: 2016-10-04T12:25:42Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: 2016-10-04T12:25:39Z
    lastUpdateTime: 2016-10-04T12:25:39Z
    message: 'Error creating: pods "nginx-deployment-4262182780-" is forbidden: exceeded
quota:
      object-counts, requested: pods=1, used: pods=3, limited: pods=2'
    reason: FailedCreate
    status: "True"
    type: ReplicaFailure
  observedGeneration: 3
  replicas: 2
  unavailableReplicas: 2
```

最终，一旦超过 Deployment 进程的 deadline，kuberentes 会更新状态和导致 Progressing 状态的原因：

```
Conditions:
  Type            Status  Reason
  ----            ------  ------
  Available       True    MinimumReplicasAvailable
  Progressing     False   ProgressDeadlineExceeded
  ReplicaFailure  True    FailedCreate
```

您可以通过缩容 Deployment的方式解决配额不足的问题，或者增加您的 namespace 的配额。如果您满足了配额条件后，Deployment controller 就会完成您的 Deployment rollout，您将看到 Deployment 的状态更新为成功状态（Status=True并且Reason=NewReplicaSetAvailable）。

```
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
```

Type=Available、 Status=True 以为这您的Deployment有最小可用性。 最小可用性是在Deployment策略中指定的参数。Type=Progressing 、 Status=True意味着您的Deployment 或者在部署过程中，或者已经成功部署，达到了期望的最少的可用replica数量（查看特定状态的Reason——在我们的例子中Reason=NewReplicaSetAvailable 意味着Deployment已经完成）。

您可以使用kubectl rollout status命令查看Deployment进程是否失败。当Deployment过程超过了deadline，kubectl rollout status将返回非0的exit code。

```
$ kubectl rollout status deploy/nginx
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
error: deployment "nginx" exceeded its progress deadline
$ echo $?
1
```

操作失败的 Deployment

所有对完成的 Deployment 的操作都适用于失败的 Deployment。您可以对它扩/缩容，回退到历史版本，您甚至可以多次暂停它来应用 Deployment pod template。

清理Policy

您可以设置 Deployment 中的 .spec.revisionHistoryLimit 项来指定保留多少旧的 ReplicaSet。 余下的将在后台被当作垃圾收集。默认的，所有的 revision 历史就都会被保留。在未来的版本中，将会更改为2。

注意： 将该值设置为0，将导致所有的 Deployment 历史记录都会被清除，该 Deployment 就无法再回退了。

用例

金丝雀 Deployment

如果您想要使用 Deployment 对部分用户或服务器发布 release，您可以创建多个 Deployment，每个 Deployment 对应一个 release，参照 managing resources 中对金丝雀模式的描述。

编写 Deployment Spec

在所有的 Kubernetes 配置中，Deployment 也需要apiVersion，kind和metadata这些配置项。配置文件的通用使用说明查看 部署应用，配置容器，和 使用 kubectl 管理资源 文档。

Deployment也需要 .spec section.

Pod Template

.spec.template 是 .spec中唯一要求的字段。

.spec.template 是 pod template. 它跟 Pod有一模一样的schema，除了它是嵌套的并且不需要apiVersion 和 kind 字段。

另外为了划分Pod的范围，Deployment中的pod template必须指定适当的label（不要跟其他controller重复了，参考selector）和适当的重启策略。

.spec.template.spec.restartPolicy 可以设置为 Always , 如果不指定的话这就是默认配置。

Replicas

.spec.replicas 是可以选字段，指定期望的pod数量，默认是1。

Selector

.spec.selector是可选字段，用来指定 label selector ，圈定Deployment管理的pod范围。

如果被指定， .spec.selector 必须匹配 .spec.template.metadata.labels，否则它将被API拒绝。如果 .spec.selector 没有被指定， .spec.selector.matchLabels 默认是 .spec.template.metadata.labels。

在Pod的template跟.spec.template不同或者数量超过了.spec.replicas规定的数量的情况下，Deployment会杀掉label跟selector不同的Pod。

注意： 您不应该再创建其他label跟这个selector匹配的pod，或者通过其他Deployment，或者通过其他Controller，例如ReplicaSet和ReplicationController。否则该Deployment会被把它们当成都是自己创建的。Kubernetes不会阻止您这么做。

如果您有多个controller使用了重复的selector，controller们就会互相打架并导致不正确的行为。

策略

.spec.strategy 指定新的Pod替换旧的Pod的策略。 .spec.strategy.type 可以是"Recreate"或者是"RollingUpdate"。"RollingUpdate"是默认值。

Recreate Deployment

.spec.strategy.type==Recreate时，在创建出新的Pod之前会先杀掉所有已存在的Pod。

Rolling Update Deployment

.spec.strategy.type==RollingUpdate时，Deployment使用rolling update 的方式更新Pod 。您可以指定 maxUnavailable 和 maxSurge 来控制 rolling update 进程。

MAX UNAVAILABLE

.spec.strategy.rollingUpdate.maxUnavailable 是可选配置项，用来指定在升级过程中不可用Pod的最大数量。该值可以是一个绝对值（例如5），也可以是期望Pod数量的百分比（例如10%）。通过计算百分比的绝对值向下取整。如果.spec.strategy.rollingUpdate.maxSurge 为0时，这个值不可以为0。默认值是1。

例如，该值设置成30%，启动rolling update后旧的ReplicatSet将会立即缩容到期望的Pod数量的70%。新的Pod ready后，随着新的ReplicaSet的扩容，旧的ReplicaSet会进一步缩容，确保在升级的所有时刻可以用的Pod数量至少是期望Pod数量的70%。

MAX SURGE

.spec.strategy.rollingUpdate.maxSurge 是可选配置项，用来指定可以超过期望的Pod数量的最大个数。该值可以是一个绝对值（例如5）或者是期望的Pod数量的百分比（例如10%）。当MaxUnavailable为0时该值不可以为0。通过百分比计算的绝对值向上取整。默认值是1。

例如，该值设置成30%，启动rolling update后新的ReplicatSet将会立即扩容，新老Pod的总数不能超过期望的Pod数量的130%。旧的Pod被杀掉后，新的ReplicaSet将继续扩容，旧的ReplicaSet会进一步缩容，确保在升级的所有时刻所有的Pod数量和不会超过期望Pod数量的130%。

Progress Deadline Seconds

.spec.progressDeadlineSeconds 是可选配置项，用来指定在系统报告Deployment的failed progressing——表现为resource的状态中type=Progressing、Status=False、 Reason=ProgressDeadlineExceeded前可以等待的Deployment进行的秒数。Deployment controller会继续重试该Deployment。未来，在实现了自动回滚后，deployment controller在观察到这种状态时就会自动回滚。

如果设置该参数，该值必须大于 .spec.minReadySeconds。

Min Ready Seconds

.spec.minReadySeconds是一个可选配置项，用来指定没有任何容器crash的Pod并被认为是可用状态的最小秒数。默认是0（Pod在ready后就会被认为是可用状态）。进一步了解什么什么后Pod会被认为是ready状态，参阅 Container Probes。

Rollback To

.spec.rollbackTo 是一个可以选配置项，用来配置Deployment回退的配置。设置该参数将触发回退操作，每次回退完成后，该值就会被清除。

Revision

.spec.rollbackTo.revision是一个可选配置项，用来指定回退到的revision。默认是0，意味着回退到历史中最老的revision。

Revision History Limit

Deployment revision history存储在它控制的ReplicaSets中。

.spec.revisionHistoryLimit 是一个可选配置项，用来指定可以保留的旧的ReplicaSet数量。该理想值取决于心Deployment的频率和稳定性。如果该值没有设置的话，默认所有旧的Replicaset或会被保留，将资源存储在etcd中，是用kubectl get rs查看输出。每个Deployment的该配置都保存在ReplicaSet中，然而，一旦您删除的旧的RepelicaSet，您的Deployment就无法再回退到那个revison了。

如果您将该值设置为0，所有具有0个replica的ReplicaSet都会被删除。在这种情况下，新的Deployment rollout无法撤销，因为revision history都被清理掉了。

Paused

.spec.paused是可以可选配置项，boolean值。用来指定暂停和恢复Deployment。Paused和没有paused的Deployment之间的唯一区别就是，所有对paused deployment中的PodTemplateSpec的修改都不会触发新的rollout。Deployment被创建之后默认是非paused。

Deployment 的替代选择

kubectl rolling update

Kubectl rolling update 虽然使用类似的方式更新Pod和ReplicationController。但是我们推荐使用Deployment，因为它是声明式的，客户端侧，具有附加特性，例如即使滚动升级结束后也可以回滚到任何历史版本。

2.7 HPA(Horizontal Pod Autoscaling)

Horizontal Pod Autoscaler根据观察到的CPU利用率自动调整复制控制器，部署或副本集中的pod数量（或者，在beta支持的情况下，根据应用程序提供的其他指标）。

需要在群集中部署度量标准 - 服务器监控，以通过资源度量标准API提供度量标准，因为Horizontal Pod Autoscaler使用此API来收集度量标准。部署它的说明位于metrics-server的GitHub存储库中，如果您开始使用GCE指南，则默认情况下会启用metrics-server监控。

Horizontal Pod Autoscaling (HPA) 可以根据 CPU 使用率或应用自定义 metrics 自动扩 展 Pod 数量（支持 replication controller、deployment 和 replica set ） 。 控制管理器每隔 30s（可以通过 --horizontal-pod-autoscaler-sync-period 修改） 查询 metrics 的资源使用情况 支持三种 metrics 类型 预定义 metrics（比如 Pod 的 CPU） 以利用率的方式计算 自定义的 Pod metrics，以原始值（raw value） 的方式计算 自定义的 object metrics 支持两种 metrics 查询方式：Heapster 和自定义的 REST API 支持多 metrics 注意： 本章是关于 Pod 的自动扩展，而 Node 的自动扩展请参考 Cluster AutoScaler。 在使用 HPA 之前需要 确保已部署好 metrics-server。

HPA 最佳实践 为容器配置 CPU Requests HPA 目标设置恰当，如设置 70% 给容器和应用预留 30% 的余量 保持 Pods 和 Nodes 健康（避免 Pod 频繁重建） 保证用户请求的负载均衡 使用 kubectl top node 和 kubectl top pod 查看资源使用情况

2.8 服务(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?

关于 Service

Kubernetes Service 定义了这样一种抽象：一个 Pod 的逻辑分组，一种可以访问它们的策略 —— 通常称为微服务。这一组 Pod 能够被 Service 访问到，通常是通过 Label Selector（查看下面了解，为什么可能需要没有 selector 的 Service）实现的。

举个例子，考虑一个图片处理 backend，它运行了3个副本。这些副本是可互换的 —— frontend 不需要关心它们调用了哪个 backend 副本。 然而组成这一组 backend 程序的 Pod 实际上可能会发生变化，frontend 客户端不应该也没必要知道，而且也不需要跟踪这一组 backend 的状态。 Service 定义的抽象能够解耦这种关联。

对 Kubernetes 集群中的应用，Kubernetes 提供了简单的 Endpoints API，只要 Service 中的一组 Pod 发生变更，应用程序就会被更新。 对非 Kubernetes 集群中的应用，Kubernetes 提供了基于 VIP 的网桥的方式访问 Service，再由 Service 重定向到 backend Pod。

定义 Service

一个 Service 在 Kubernetes 中是一个 REST 对象，和 Pod 类似。 像所有的 REST 对象一样， Service 定义可以基于 POST 方式，请求 apiserver 创建新的实例。 例如，假定有一组 Pod，它们对外暴露了 9376 端口，同时还被打上 "app=MyApp" 标签。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

上述配置将创建一个名称为 “my-service” 的 Service 对象，它会将请求代理到使用 TCP 端口 9376，并且具有标签 "app=MyApp" 的 Pod 上。 这个 Service 将被指派一个 IP 地址（通常称为 “Cluster IP”），它会被服务的代理使用（见下面）。 该 Service 的 selector 将会持续评估，处理结果将被 POST 到一个名称为 “my-service” 的 Endpoints 对象上。

需要注意的是， Service 能够将一个接收端口映射到任意的 targetPort。 默认情况下，targetPort 将被设置为与 port 字段相同的值。 可能更有趣的是，targetPort 可以是一个字符串，引用了 backend Pod 的一个端口的名称。但是，实际指派给该端口名称的端口号，在每个 backend Pod 中可能并不相同。 对于部署和设计 Service ，这种方式会提供更大的灵活性。 例如，可以在 backend 软件下一个版本中，修改 Pod 暴露的端口，并不会中断客户端的调用。

Kubernetes Service 能够支持 TCP 和 UDP 协议，默认 TCP 协议。

没有 selector 的 Service

Servcie 抽象了该如何访问 Kubernetes Pod，但也能够抽象其它类型的 backend，例如：

- 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 Namespace 中或其它集群中的服务。

- 正在将工作负载转移到 Kubernetes 集群，和运行在 Kubernetes 集群之外的 backend。

在任何这些场景中，都能够定义没有 selector 的 Service：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

由于这个 Service 没有 selector，就不会创建相关的 Endpoints 对象。可以手动将 Service 映射到指定的 Endpoints：

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

注意：Endpoint IP 地址不能是 loopback（127.0.0.0/8）、 link-local（169.254.0.0/16）、或者 link-local 多播（224.0.0.0/24）。

访问没有 selector 的 Service，与有 selector 的 Service 的原理相同。请求将被路由到用户定义的 Endpoint（该示例中为 1.2.3.4:9376）。

ExternalName Service 是 Service 的特例，它没有 selector，也没有定义任何的端口和 Endpoint。 相反地，对于运行在集群外部的服务，它通过返回该外部服务的别名这种方式来提供服务。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

当查询主机 my-service.prod.svc.CLUSTER时，集群的 DNS 服务将返回一个值为 my.database.example.com 的 CNAME 记录。 访问这个服务的工作方式与其它的相同，唯一不同的是重定向发生在 DNS 层，而且不会进行代理或转发。 如果后续决定要将数据库迁移到 Kubernetes 集群中，可以启动对应的 Pod，增加合适的 Selector 或 Endpoint，修改 Service 的 type。

VIP 和 Service 代理

在 Kubernetes 集群中，每个 Node 运行一个 kube-proxy 进程。kube-proxy 负责为 Service 实现了一种 VIP（虚拟 IP）的形式，而不是 ExternalName 的形式。 在 Kubernetes v1.0 版本，代理完全在 userspace。在 Kubernetes v1.1 版本，新增了 iptables 代理，但并不是默认的运行模式。 从 Kubernetes v1.2 起，默认就是 iptables 代理。

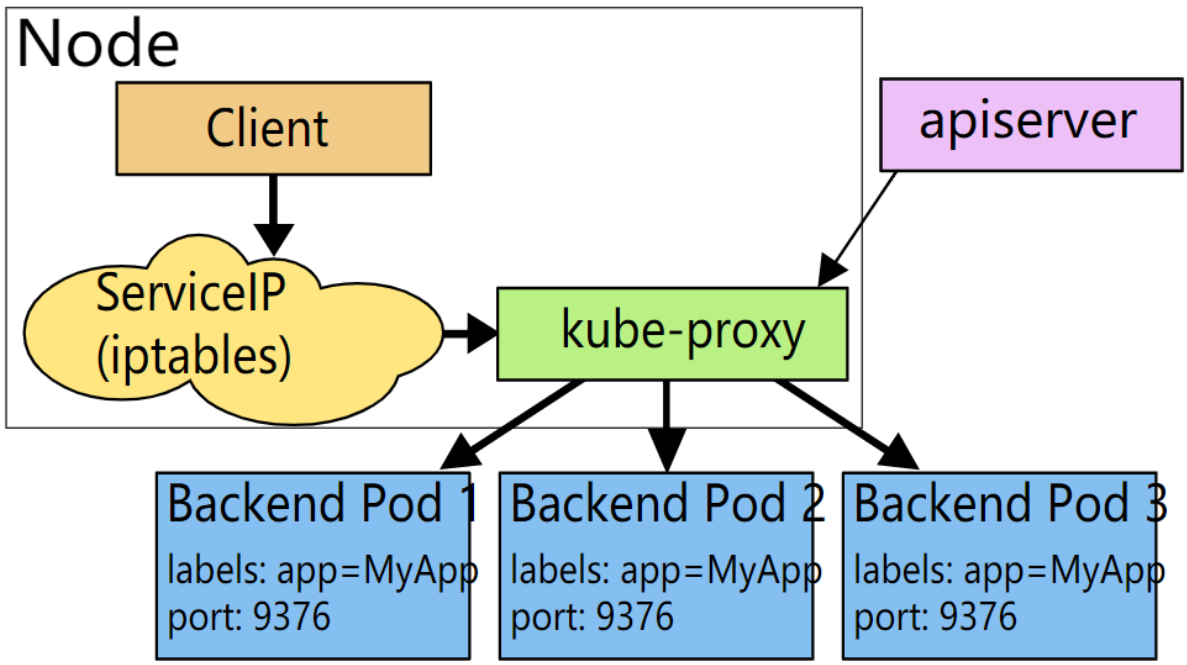
在 Kubernetes v1.0 版本，Service 是 “4层”（TCP/UDP over IP）概念。 在 Kubernetes v1.1 版本，新增了 Ingress API（beta 版），用来表示 “7层”（HTTP）服务。

userspace 代理模式

这种模式，kube-proxy 会监视 Kubernetes master 对 Service 对象和 Endpoints 对象的添加和移除。 对每个 Service，它会在本地 Node 上打开一个端口（随机选择）。 任何连接到“代理端口”的请求，都会被代理到 Service 的 backend Pods 中的某个上面（如 Endpoints 所报告的一样）。 使用哪个 backend Pod，是基于 Service 的 SessionAffinity 来确定的。 最后，它安装 iptables 规则，捕获到达该 Service 的 clusterIP（是虚拟 IP）和 Port 的请求，并重定向到代理端口，代理端口再代理请求到 backend Pod。

网络返回的结果是，任何到达 Service 的 IP:Port 的请求，都会被代理到一个合适的 backend，不需要客户端知道关于 Kubernetes、Service、或 Pod 的任何信息。

默认的策略是，通过 round-robin 算法来选择 backend Pod。 实现基于客户端 IP 的会话亲和性，可以通过设置 service.spec.sessionAffinity 的值为 "ClientIP" （默认值为 "None"）。

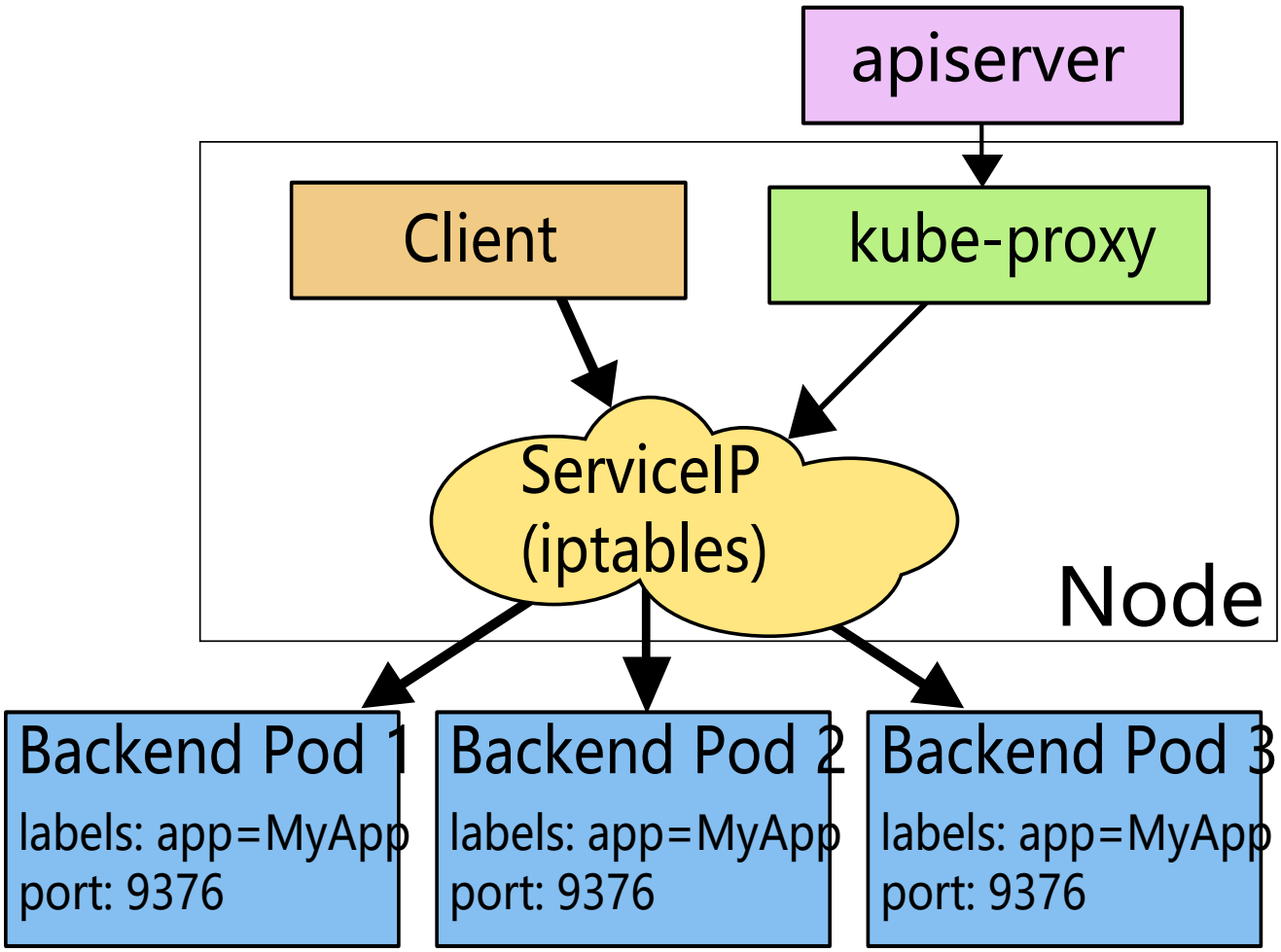


iptables 代理模式

这种模式，kube-proxy 会监视 Kubernetes master 对 Service 对象和 Endpoints 对象的添加和移除。 对每个 Service，它会安装 iptables 规则，从而捕获到达该 Service 的 clusterIP（虚拟 IP）和端口的请求，进而将请求重定向到 Service 的一组 backend 中的某个上面。 对于每个 Endpoints 对象，它也会安装 iptables 规则，这个规则会选择一个 backend Pod。

默认的策略是，随机选择一个 backend。 实现基于客户端 IP 的会话亲和性，可以将 service.spec.sessionAffinity 的值设置为 "ClientIP" （默认值为 "None"）。

和 userspace 代理类似，网络返回的结果是，任何到达 Service 的 IP:Port 的请求，都会被代理到一个合适的 backend，不需要客户端知道关于 Kubernetes、Service、或 Pod 的任何信息。 这应该比 userspace 代理更快、更可靠。然而，不像 userspace 代理，如果初始选择的 Pod 没有响应，iptables 代理能够自动地重试另一个 Pod，所以它需要依赖 readiness probes。



多端口 Service

很多 Service 需要暴露多个端口。对于这种情况，Kubernetes 支持在 Service 对象中定义多个端口。 当使用多个端口时，必须给出所有的端口的名称，这样 Endpoint 就不会产生歧义，例如：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
    selector:
      app: MyApp
    ports:
      - name: http
        protocol: TCP
        port: 80
        targetPort: 9376
      - name: https
        protocol: TCP
```

```
        port: 443
        targetPort: 9377
```

选择自己的 IP 地址

在 Service 创建的请求中，可以通过设置 spec.clusterIP 字段来指定自己的集群 IP 地址。 比如，希望替换一个已经已存在的 DNS 条目，或者遗留系统已经配置了一个固定的 IP 且很难重新配置。 用户选择的 IP 地址必须合法，并且这个 IP 地址在 service-cluster-ip-range CIDR 范围内，这对 API Server 来说是通过一个标识来指定的。 如果 IP 地址不合法，API Server 会返回 HTTP 状态码 422，表示值不合法。

为何不使用 round-robin DNS?

一个不时出现的问题是，为什么我们都使用 VIP 的方式，而不使用标准的 round-robin DNS，有如下几个原因：

- 长久以来，DNS 库都没能认真对待 DNS TTL、缓存域名查询结果
- 很多应用只查询一次 DNS 并缓存了结果
    - 就算应用和库能够正确查询解析，每个客户端反复重解析造成的负载也是非常难以管理的

我们尽力阻止用户做那些对他们没有好处的事情，如果很多人都来问这个问题，我们可能会选择实现它。

服务发现

Kubernetes 支持2种基本的服务发现模式 —— 环境变量和 DNS。

环境变量

当 Pod 运行在 Node 上，kubelet 会为每个活跃的 Service 添加一组环境变量。 它同时支持 Docker links兼容 变量（查看 makeLinkVariables）、简单的 {SVCNAME}*SERVICE_HOST 和{SVCNAME}*SERVICE_PORT 变量，这里 Service 的名称需大写，横线被转换成下划线。

举个例子，一个名称为 "redis-master" 的 Service 暴露了 TCP 端口 6379，同时给它分配了 Cluster IP 地址 10.0.0.11，这个 Service 生成了如下环境变量：

```
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
```

这意味着需要有顺序的要求 —— Pod 想要访问的任何 Service 必须在 Pod 自己之前被创建，否则这些环境变量就不会被赋值。DNS 并没有这个限制。

DNS

一个可选（尽管强烈推荐）集群插件 是 DNS 服务器。 DNS 服务器监视着创建新 Service 的 Kubernetes API，从而为每一个 Service 创建一组 DNS 记录。 如果整个集群的 DNS 一直被启用，那么所有的 Pod 应该能够自动对 Service 进行名称解析。

例如，有一个名称为 "my-service" 的 Service，它在 Kubernetes 集群中名为 "my-ns" 的 Namespace 中，为 "my-service.my-ns" 创建了一条 DNS 记录。 在名称为 "my-ns" 的 Namespace 中的 Pod 应该能够简单地通过名称查询找到 "my-service"。 在另一个 Namespace 中的 Pod 必须限定名称为 "my-service.my-ns"。 这些名称查询的结果是 Cluster IP。

Kubernetes 也支持对端口名称的 DNS SRV（Service）记录。 如果名称为 "my-service.my-ns" 的 Service 有一个名为 "http" 的 TCP 端口，可以对 "*http*.tcp.my-service.my-ns" 执行 DNS SRV 查询，得到 "http" 的端口号。

Kubernetes DNS 服务器是唯一的一种能够访问 ExternalName 类型的 Service 的方式。 更多信息可以查看DNS Pod 和 Service。

Headless Service

有时不需要或不想要负载均衡，以及单独的 Service IP。 遇到这种情况，可以通过指定 Cluster IP（spec.clusterIP）的值为 "None" 来创建 Headless Service。

这个选项允许开发人员自由寻找他们自己的方式，从而降低与 Kubernetes 系统的耦合性。 应用仍然可以使用一种自注册的模式和适配器，对其它需要发现机制的系统能够很容易地基于这个 API 来构建。

对这类 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。 DNS 如何实现自动配置，依赖于 Service 是否定义了 selector。

配置 Selector

对定义了 selector 的 Headless Service，Endpoint 控制器在 API 中创建了 Endpoints 记录，并且修改 DNS 配置返回 A 记录（地址），通过这个地址直接到达 Service 的后端 Pod上。

不配置 Selector

对没有定义 selector 的 Headless Service，Endpoint 控制器不会创建 Endpoints 记录。 然而 DNS 系统会查找和配置，无论是：

- ExternalName 类型 Service 的 CNAME 记录
    - 记录：与 Service 共享一个名称的任何 Endpoints，以及所有其它类型

发布服务 —— 服务类型

对一些应用（如 Frontend）的某些部分，可能希望通过外部（Kubernetes 集群外部）IP 地址暴露 Service。

Kubernetes ServiceTypes 允许指定一个需要的类型的 Service，默认是 ClusterIP 类型。

Type 的取值以及行为如下：

- ClusterIP：通过集群的内部 IP 暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的 ServiceType。
- NodePort：通过每个 Node 上的 IP 和静态端口（NodePort）暴露服务。NodePort 服务会路由到 ClusterIP 服务，这个 ClusterIP 服务会自动创建。通过请求 :，可以从集群的外部访问一个 NodePort 服务。
- LoadBalancer：使用云提供商的负载局衡器，可以向外部暴露服务。外部的负载均衡器可以路由到 NodePort 服务和 ClusterIP 服务。
- ExternalName：通过返回 CNAME 和它的值，可以将服务映射到 externalName 字段的内容（例如，foo.bar.example.com）。 没有任何类型代理被创建，这只有 Kubernetes 1.7 或更高版本的 kube-dns 才支持。

NodePort 类型

如果设置 type 的值为 "NodePort"，Kubernetes master 将从给定的配置范围内（默认：30000-32767）分配端口，每个 Node 将从该端口（每个 Node 上的同一端口）代理到 Service。该端口将通过 Service 的 spec.ports[*].nodePort 字段被指定。

如果需要指定的端口号，可以配置 nodePort 的值，系统将分配这个端口，否则调用 API 将会失败（比如，需要关心端口冲突的可能性）。

这可以让开发人员自由地安装他们自己的负载均衡器，并配置 Kubernetes 不能完全支持的环境参数，或者直接暴露一个或多个 Node 的 IP 地址。

需要注意的是，Service 将能够通过 :spec.ports[*].nodePort 和 spec.clusterIp:spec.ports[*].port 而对外可见。

LoadBalancer 类型

使用支持外部负载均衡器的云提供商的服务，设置 type 的值为 "LoadBalancer"，将为 Service 提供负载均衡器。 负载均衡器是异步创建的，关于被提供的负载均衡器的信息将会通过 Service 的 status.loadBalancer 字段被发布出去。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
      nodePort: 30061
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
      - ip: 146.148.47.155
```

来自外部负载均衡器的流量将直接打到 backend Pod 上，不过实际它们是如何工作的，这要依赖于云提供商。 在这些情况下，将根据用户设置的 loadBalancerIP 来创建负载均衡器。 某些云提供商允许设置 loadBalancerIP。如果没有设置 loadBalancerIP，将会给负载均衡器指派一个临时 IP。 如果设置了 loadBalancerIP，但云提供商并不支持这种特性，那么设置的 loadBalancerIP 值将会被忽略掉。

AWS 内部负载均衡器

在混合云环境中，有时从虚拟私有云（VPC）环境中的服务路由流量是非常有必要的。 可以通过在 Service 中增加 annotation 来实现，如下所示：

```
[...]
metadata:
    name: my-service
    annotations:
        service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0
[...]
```

在水平分割的 DNS 环境中，需要两个 Service 来将外部和内部的流量路由到 Endpoint 上。

AWS SSL 支持

对运行在 AWS 上部分支持 SSL 的集群，从 1.3 版本开始，可以为 LoadBalancer 类型的 Service 增加两个 annotation：

```
    metadata:
      name: my-service
      annotations:
        service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:us-east-
1:123456789012:certificate/12345678-1234-1234-1234-123456789012
```

第一个 annotation 指定了使用的证书。它可以是第三方发行商发行的证书，这个证书或者被上传到 IAM，或者由 AWS 的证书管理器创建。

```
    metadata:
      name: my-service
      annotations:
         service.beta.kubernetes.io/aws-load-balancer-backend-protocol:
(https|http|ssl|tcp)
```

第二个 annotation 指定了 Pod 使用的协议。 对于 HTTPS 和 SSL，ELB 将期望该 Pod 基于加密的连接来认证自身。

HTTP 和 HTTPS 将选择7层代理：ELB 将中断与用户的连接，当转发请求时，会解析 Header 信息并添加上用户的 IP 地址（Pod 将只能在连接的另一端看到该 IP 地址）。

TCP 和 SSL 将选择4层代理：ELB 将转发流量，并不修改 Header 信息。

外部 IP

如果外部的 IP 路由到集群中一个或多个 Node 上，Kubernetes Service 会被暴露给这些 externalIPs。 通过外部 IP（作为目的 IP 地址）进入到集群，打到 Service 的端口上的流量，将会被路由到 Service 的 Endpoint 上。 externalIPs 不会被 Kubernetes 管理，它属于集群管理员的职责范畴。

根据 Service 的规定，externalIPs 可以同任意的 ServiceType 来一起指定。 在上面的例子中，my-service 可以在 80.11.12.10:80（外部 IP:端口）上被客户端访问。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
```

```
    app: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
  externalIPs:
    - 80.11.12.10
```

不足之处

为 VIP 使用 userspace 代理，将只适合小型到中型规模的集群，不能够扩展到上千 Service 的大型集群。 查看 最初设计方案 获取更多细节。

使用 userspace 代理，隐藏了访问 Service 的数据包的源 IP 地址。 这使得一些类型的防火墙无法起作用。 iptables 代理不会隐藏 Kubernetes 集群内部的 IP 地址，但却要求客户端请求必须通过一个负载均衡器或 Node 端口。

Type 字段支持嵌套功能 —— 每一层需要添加到上一层里面。 不会严格要求所有云提供商（例如，GCE 就没必要为了使一个 LoadBalancer 能工作而分配一个 NodePort，但是 AWS 需要 ），但当前 API 是强制要求的。

未来工作

未来我们能预见到，代理策略可能会变得比简单的 round-robin 均衡策略有更多细微的差别，比如 master 选举或分片。 我们也能想到，某些 Service 将具有 “真正” 的负载均衡器，这种情况下 VIP 将简化数据包的传输。

我们打算为 L7（HTTP）Service 改进我们对它的支持。

我们打算为 Service 实现更加灵活的请求进入模式，这些 Service 包含当前 ClusterIP、NodePort 和 LoadBalancer 模式，或者更多。

VIP 的那些骇人听闻的细节

对很多想使用 Service 的人来说，前面的信息应该足够了。 然而，有很多内部原理性的内容，还是值去理解的。

避免冲突

Kubernetes 最主要的哲学之一，是用户不应该暴露那些能够导致他们操作失败、但又不是他们的过错的场景。 这种场景下，让我们来看一下网络端口 —— 用户不应该必须选择一个端口号，而且该端口还有可能与其他用户的冲突。 这就是说，在彼此隔离状态下仍然会出现失败。

为了使用户能够为他们的 Service 选择一个端口号，我们必须确保不能有2个 Service 发生冲突。 我们可以通过为每个 Service 分配它们自己的 IP 地址来实现。

为了保证每个 Service 被分配到一个唯一的 IP，需要一个内部的分配器能够原子地更新 etcd 中的一个全局分配映射表，这个更新操作要先于创建每一个 Service。 为了使 Service能够获取到 IP，这个映射表对象必须在注册中心存在，否则创建 Service 将会失败，指示一个 IP 不能被分配。 一个后台 Controller 的职责是创建映射表（从 Kubernetes 的旧版本迁移过来，旧版本中是通过在内存中加锁的方式实现），并检查由于管理员干预和清除任意 IP 造成的不合理分配，这些 IP 被分配了但当前没有 Service 使用它们。

IP 和 VIP

不像 Pod 的 IP 地址，它实际路由到一个固定的目的地，Service 的 IP 实际上不能通过单个主机来进行应答。 相反，我们使用 iptables（Linux 中的数据包处理逻辑）来定义一个虚拟IP地址（VIP），它可以根据需要透明地进行重定向。 当客户端连接到 VIP 时，它们的流量会自动地传输到一个合适的 Endpoint。 环境变量和 DNS，实际上会根据 Service 的 VIP 和端口来进行填充。

Userspace

作为一个例子，考虑前面提到的图片处理应用程序。 当创建 backend Service 时，Kubernetes master 会给它指派一个虚拟 IP 地址，比如 10.0.0.1。 假设 Service 的端口是 1234，该 Service 会被集群中所有的 kube-proxy 实例观察到。 当代理看到一个新的 Service， 它会打开一个新的端口，建立一个从该 VIP 重定向到新端口的 iptables，并开始接收请求连接。

当一个客户端连接到一个 VIP，iptables 规则开始起作用，它会重定向该数据包到 Service代理 的端口。 Service代理选择一个 backend，并将客户端的流量代理到 backend 上。

这意味着 Service 的所有者能够选择任何他们想使用的端口，而不存在冲突的风险。 客户端可以简单地连接到一个 IP 和端口，而不需要知道实际访问了哪些 Pod。

Iptables

再次考虑前面提到的图片处理应用程序。 当创建 backend Service 时，Kubernetes master 会给它指派一个虚拟 IP 地址，比如 10.0.0.1。 假设 Service 的端口是 1234，该 Service会被集群中所有的 kube-proxy 实例观察到。 当代理看到一个新的 Service， 它会安装一系列的 iptables 规则，从 VIP 重定向到 per-Service 规则。 该 per-Service 规则连接到 per-Endpoint 规则，该 per-Endpoint 规则会重定向（目标 NAT）到 backend。

当一个客户端连接到一个 VIP，iptables 规则开始起作用。一个 backend 会被选择（或者根据会话亲和性，或者随机），数据包被重定向到这个 backend。 不像 userspace 代理，数据包从来不拷贝到用户空间，kube-proxy 不是必须为该 VIP 工作而运行，并且客户端 IP 是不可更改的。 当流量打到 Node 的端口上，或通过负载均衡器，会执行相同的基本流程，但是在那些案例中客户端 IP 是可以更改的。

2.9 存储卷(Volume)

默认情况下容器中的磁盘文件是非持久化的，对于运行在容器中的应用来说面临两个问题，第一：当容器挂掉kubelet将重启启动它时，文件将会丢失；第二：当Pod中同时运行多个容器，容器之间需要共享文件时。Kubernetes的Volume解决了这两个问题。

背景

在Docker中也有一个docker Volume的概念，Docker的Volume只是磁盘中的一个目录，生命周期不受管理。当然Docker现在也提供Volume将数据持久化存储，但支持功能比较少（例如，对于Docker 1.7，每个容器只允许挂载一个Volume，并且不能将参数传递给Volume）。

另一方面，Kubernetes Volume具有明确的生命周期 - 与pod相同。因此，Volume的生命周期比Pod中运行的任何容器要持久，在容器重新启动时能可以保留数据，当然，当Pod被删除不存在时，Volume也将消失。注意，Kubernetes支持许多类型的Volume，Pod可以同时使用任意类型/数量的Volume。

内部实现中，一个Volume只是一个目录，目录中可能有一些数据，pod的容器可以访问这些数据。至于这个目录是如何产生的、支持它的介质、其中的数据内容是什么，这些都由使用的特定Volume类型来决定。

要使用Volume，pod需要指定Volume的类型和内容（`spec.volumes`字段），和映射到容器的位置（`spec.containers.volumeMounts`字段）。

Volume 类型

Kubernetes支持Volume类型有：

- emptyDir
- hostPath
- gcePersistentDisk
- awsElasticBlockStore
- nfs
- iscsi
- fc (fibre channel)
- flocker
- glusterfs
- rbd
- cephfs
- gitRepo
- secret
- persistentVolumeClaim
- downwardAPI
- projected
- azureFileVolume
- azureDisk
- vsphereVolume
- Quobyte
- PortworxVolume
- ScaleIO
- StorageOS
- local

emptyDir

使用emptyDir，当Pod分配到Node上时，将会创建emptyDir，并且只要Node上的Pod一直运行，Volume就会一直存。当Pod（不管任何原因）从Node上被删除时，emptyDir也同时会删除，存储的数据也将永久删除。注：删除容器不影响emptyDir。

示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
```

```
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

hostPath

hostPath允许挂载Node上的文件系统到Pod里面去。如果Pod需要使用Node上的文件，可以使用hostPath。

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
```

gcePersistentDisk

gcePersistentDisk可以挂载GCE上的永久磁盘到容器，需要Kubernetes运行在GCE的VM中。与emptyDir不同，Pod删除时，gcePersistentDisk被删除，但Persistent Disk 的内容任然存在。这就意味着gcePersistentDisk能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间“切换”。

提示：使用gcePersistentDisk，必须用gcloud或使用GCE API或UI 创建PD

创建PD

使用GCE PD与pod之前，需要创建它

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
```

```
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    # This GCE PD must already exist.
    gcePersistentDisk:
      pdName: my-data-disk
      fsType: ext4
```

awsElasticBlockStore

awsElasticBlockStore可以挂载AWS上的EBS盘到容器，需要Kubernetes运行在AWS的EC2上。与emptyDir Pod被删除情况不同，Volume仅被卸载，内容将被保留。这就意味着awsElasticBlockStore能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间“切换”。

提示：必须使用aws ec2 create-volumeAWS API 创建EBS Volume，然后才能使用。

创建EBS Volume

在使用EBS Volume与pod之前，需要创建它。

```
aws ec2 create-volume --availability-zone eu-west-1a --size 10 --volume-type gp2
```

AWS EBS配置示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

NFS

NFS 是Network File System的缩写，即网络文件系统。Kubernetes中通过简单地配置就可以挂载NFS到Pod中，而NFS中的数据是可以永久保存的，同时NFS支持同时写操作。Pod被删除时，Volume被卸载，内容被保留。这就意味着NFS能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间相互传递。

详细信息，请参阅NFS示例。

iSCSI

iscsi允许将现有的iscsi磁盘挂载到我们的pod中，和emptyDir不同的是，删除Pod时会被删除，但Volume只是被卸载，内容被保留，这就意味着iscsi能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间“切换”。

详细信息，请参阅iSCSI示例。

flocker

Flocker是一个开源的容器集群数据卷管理器。它提供各种存储后端支持的数据卷的管理和编排。

详细信息，请参阅Flocker示例。

glusterfs

glusterfs，允许将Glusterfs（一个开源网络文件系统）Volume安装到pod中。不同于emptyDir，Pod被删除时，Volume只是被卸载，内容被保留。味着glusterfs能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间“切换”。

详细信息，请参阅GlusterFS示例。

RBD

RBD允许Rados Block Device格式的磁盘挂载到Pod中，同样的，当pod被删除的时候，rbd也仅仅是被卸载，内容保留，rbd能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间“切换”。

详细信息，请参阅RBD示例。

cephfs

cephfs Volume可以将已经存在的CephFS Volume挂载到pod中，与emptyDir特点不同，pod被删除的时，cephfs仅被被卸载，内容保留。cephfs能够允许我们提前对数据进行处理，而且这些数据可以在Pod之间“切换”。

提示：可以使用自己的Ceph服务器运行导出，然后在使用cephfs。

详细信息，请参阅CephFS示例。

gitRepo

gitRepo volume将git代码下拉到指定的容器路径中。

示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: server
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /mypath
      name: git-volume
```

```
  volumes:
  - name: git-volume
    gitRepo:
      repository: "git@somewhere:me/my-git-repository.git"
      revision: "22f1d8406d464b0c0874075539c1f2e96c253775"
```

secret

secret volume用于将敏感信息（如密码）传递给pod。可以将secrets存储在Kubernetes API中，使用的时候以文件的形式挂载到pod中，而不用连接api。 secret volume由tmpfs（RAM支持的文件系统）支持。

详细了解secret

persistentVolumeClaim

persistentVolumeClaim用来挂载持久化磁盘的。PersistentVolumes是用户在不知道特定云环境的细节的情况下，实现持久化存储（如GCE PersistentDisk或iSCSI卷）的一种方式。

更多详细信息，请参阅PersistentVolumes示例。

downwardAPI

通过环境变量的方式告诉容器Pod的信息

更多详细信息，请参见downwardAPI卷示例。

projected

Projected volume将多个Volume源映射到同一个目录

目前，可以支持以下类型的卷源：

- secret
- downwardAPI
- configMap

所有卷源都要求与pod在同一命名空间中。更详细信息，请参阅all-in-one volume design document。

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
```

```
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - downwardAPI:
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
          items:
            - key: password
              path: my-group/my-password
              mode: 511
```

FlexVolume

alpha功能

更多细节在这里

AzureFileVolume

AzureFileVolume用于将Microsoft Azure文件卷（SMB 2.1和3.0）挂载到Pod中。

更多细节在这里

AzureDiskVolume

Azure是微软提供的公有云服务，如果使用Azure上面的虚拟机来作为Kubernetes集群使用时，那么可以通过AzureDisk这种类型的卷插件来挂载Azure提供的数据磁盘。

更多细节在这里

vsphereVolume

需要条件：配置了vSphere Cloud Provider的Kubernetes。有关cloudprovider配置，请参阅vSphere入门指南。

vsphereVolume用于将vSphere VMDK Volume挂载到Pod中。卸载卷后，内容将被保留。它同时支持VMFS和VSAN数据存储。

重要提示：使用POD之前，必须使用以下方法创建VMDK。

创建一个VMDK卷

- 使用vmkfstools创建。先将ssh接入ESX，然后使用以下命令创建vmdk

```
vmkfstools -c 2G /vmfs/volumes/DatastoreName/volumes/myDisk.vmdk
```

- 使用vmware-vdiskmanager创建

```
shell vmware-vdiskmanager -c -t 0 -s 40GB -a lsilogic myDisk.vmdk
```

示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-vmdk
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-vmdk
      name: test-volume
  volumes:
  - name: test-volume
    # This VMDK volume must already exist.
    vsphereVolume:
      volumePath: "[DatastoreName] volumes/myDisk"
      fsType: ext4
```

更多例子在这里。

Quobyte

在kubernetes中使用Quobyte存储，需要提前部署Quobyte软件，要求必须是1.3以及更高版本，并且在kubernetes管理的节点上面部署Quobyte客户端。

详细信息，请参阅这里。

PortworxVolume

Portworx能把你的服务器容量进行蓄积（pool），将你的服务器或者云实例变成一个聚合的高可用的计算和存储节点。

PortworxVolume可以通过Kubernetes动态创建，也可以在Kubernetes pod中预先配置和引用。示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-portworx-volume-pod
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /mnt
      name: pxvol
  volumes:
  - name: pxvol
    # This Portworx volume must already exist.
    portworxVolume:
      volumeID: "pxvol"
      fsType: "<fs-type>"
```

更多细节和例子可以在这里找到

ScaleIO

ScaleIO是一种基于软件的存储平台（虚拟SAN），可以使用现有硬件来创建可扩展共享块网络存储的集群。ScaleIO卷插件允许部署的pod访问现有的ScaleIO卷（或者可以为持久卷声明动态配置新卷，请参阅 Scaleio Persistent Volumes）。

示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-0
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: pod-0
```

```
    volumeMounts:
    - mountPath: /test-pd
      name: vol-0
  volumes:
  - name: vol-0
    scaleIO:
      gateway: https://localhost:443/api
      system: scaleio
      volumeName: vol-0
      secretRef:
        name: sio-secret
      fsType: xfs
```

详细信息，请参阅ScaleIO示例。

StorageOS

StorageOS是一家英国的初创公司，给无状态容器提供简单的自动块存储、状态来运行数据库和其他需要企业级存储功能，但避免随之而来的复杂性、刚性以及成本。

核心：是StorageOS向容器提供块存储，可通过文件系统访问。

StorageOS容器需要64位Linux，没有额外的依赖关系，提供免费开发许可证。

安装说明，请参阅StorageOS文档

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: redis
    role: master
  name: test-storageos-redis
spec:
  containers:
    - name: master
      image: kubernetes/redis:v1
      env:
        - name: MASTER
          value: "true"
      ports:
        - containerPort: 6379
      volumeMounts:
        - mountPath: /redis-master-data
          name: redis-data
  volumes:
    - name: redis-data
      storageos:
        # The `redis-vol01` volume must already exist within StorageOS in the `default`
namespace.
        volumeName: redis-vol01
        fsType: ext4
```

有关动态配置和持久卷声明的更多信息，请参阅StorageOS示例。

Local

目前处于 Kubernetes 1.7中的 alpha 级别。

Local 是Kubernetes集群中每个节点的本地存储（如磁盘，分区或目录），在Kubernetes1.7中kubelet可以支持对kube-reserved和system-reserved指定本地存储资源。

通过上面的这个新特性可以看出来，Local Storage同HostPath的区别在于对Pod的调度上，使用Local Storage可以由Kubernetes自动的对Pod进行调度，而是用HostPath只能人工手动调度Pod，因为Kubernetes已经知道了每个节点上kube-reserved和system-reserved设置的本地存储限制。

示例：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
  annotations:
        "volume.alpha.kubernetes.io/node-affinity": '{
            "requiredDuringSchedulingIgnoredDuringExecution": {
                "nodeSelectorTerms": [
                    { "matchExpressions": [
                        { "key": "kubernetes.io/hostname",
                          "operator": "In",
                          "values": ["example-node"]
                        }
                    ]}
                ]}
              }'
spec:
    capacity:
      storage: 100Gi
    accessModes:
    - ReadWriteOnce
    persistentVolumeReclaimPolicy: Delete
    storageClassName: local-storage
    local:
      path: /mnt/disks/ssd1
```

请注意，本地PersistentVolume需要手动清理和删除。

有关local卷类型的详细信息，请参阅 Local Persistent Storage user guide

Using subPath

有时，可以在一个pod中，将同一个卷共享，使其有多个用处。volumeMounts.subPath特性可以用来指定卷中的一个子目录，而不是直接使用卷的根目录。

以下是使用单个共享卷的LAMP堆栈（Linux Apache Mysql PHP）的pod的示例。HTML内容映射到其html文件夹，数据库将存储在mysql文件夹中：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-lamp-site
spec:
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-lamp-site-data
```

Resources

emptyDir Volume的存储介质（Disk，SSD等）取决于kubelet根目录（如/var/lib/kubelet）所处文件系统的存储介质。不限制emptyDir或hostPath Volume使用的空间大小，不对容器或Pod的资源隔离。

2.10 命名空间(NameSpace)

Kubernetes可以使用Namespaces（命名空间）创建多个虚拟集群。

何时使用多个Namespaces

当团队或项目中具有许多用户时，可以考虑使用Namespace来区分，a如果是少量用户集群，可以不需要考虑使用Namespace，如果需要它们提供特殊性质时，可以开始使用Namespace。

Namespace为名称提供了一个范围。资源的Names在Namespace中具有唯一性。

Namespace是一种将集群资源划分为多个用途(通过 resource quota)的方法。

在未来的Kubernetes版本中，默认情况下，相同Namespace中的对象将具有相同的访问控制策略。

对于稍微不同的资源没必要使用多个Namespace来划分，例如同意软件的不同版本，可以使用labels(标签)来区分同一Namespace中的资源。

使用 Namespaces

Namespace的创建、删除和查看。

创建

```
(1) 命令行直接创建
$ kubectl create namespace new-namespace

(2) 通过文件创建
$ cat my-namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: new-namespace

$ kubectl create -f ./my-namespace.yaml
```

注意：命名空间名称满足正则表达式a-z0-9?,最大长度为63位

删除

```
$ kubectl delete namespaces new-namespace
```

注意：

1. 删除一个namespace会自动删除所有属于该namespace的资源。
2. default和kube-system命名空间不可删除。
3. PersistentVolumes是不属于任何namespace的，但PersistentVolumeClaim是属于某个特定namespace的。
4. Events是否属于namespace取决于产生events的对象。

查看 Namespaces

使用以下命令列出群集中的当前的Namespace：

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    1d
kube-system   Active    1d
```

Kubernetes从两个初始的Namespace开始：

- default
- kube-system 由Kubernetes系统创建的对象的Namespace

Setting the namespace for a request

要临时设置Request的Namespace，请使用--namespace 标志。

例如：

```
$ kubectl --namespace=<insert-namespace-name-here> run nginx --image=nginx
$ kubectl --namespace=<insert-namespace-name-here> get pods
```

Setting the namespace preference

可以使用kubectl命令创建的Namespace可以永久保存在context中。

```
$ kubectl config set-context $(kubectl config current-context) --namespace=<insert-
namespace-name-here>
# Validate it
$ kubectl config view | grep namespace:
```

所有对象都在Namespace中?

大多数Kubernetes资源（例如pod、services、replication controllers或其他）都在某些Namespace中，但Namespace资源本身并不在Namespace中。而低级别资源（如Node和persistentVolumes）不在任何Namespace中。Events是一个例外：它们可能有也可能没有Namespace，具体取决于Events的对象。

2.11 注解(Anotation)

Kubernetes Annotations

可以使用Kubernetes Annotations将任何非标识metadata附加到对象。客户端（如工具和库）可以检索此metadata。

将metadata附加到对象

可以使用Labels或Annotations将元数据附加到Kubernetes对象。标签可用于选择对象并查找满足某些条件的对象集合。相比之下，Annotations不用于标识和选择对象。Annotations中的元数据可以是small 或large，structured 或unstructured，并且可以包括标签不允许使用的字符。

Annotations就如标签一样，也是由key/value组成：

```
"annotations": {
  "key1" : "value1",
  "key2" : "value2"
}
```

以下是在Annotations中记录信息的一些例子：

- 构建、发布的镜像信息，如时间戳，发行ID，git分支，PR编号，镜像hashes和注Registry地址。
- 一些日志记录、监视、分析或audit repositories。
- 一些工具信息：例如，名称、版本和构建信息。
- 用户或工具/系统来源信息，例如来自其他生态系统组件对象的URL。
- 负责人电话/座机，或一些信息目录。

注意：Annotations不会被Kubernetes直接使用，其主要目的是方便用户阅读查找。

2.12 小结

Kubernetes对象是Kubernetes系统中的持久实体。Kubernetes使用这些实体来表示集群的状态。具体来说，他们可以描述：

- 容器化应用正在运行(以及在哪些节点上)
- 这些应用可用的资源
- 关于这些应用如何运行的策略，如重新策略，升级和容错

Kubernetes对象是“record of intent”，一旦创建了对象，Kubernetes系统会确保对象存在。通过创建对象，可以有效地告诉Kubernetes系统你希望集群的工作负载是什么样的。

要使用Kubernetes对象（无论是创建，修改还是删除），都需要使用Kubernetes API。例如，当使用kubectl命令管理工具时，CLI会为提供Kubernetes API调用。你也可以直接在自己的程序中使用Kubernetes API，Kubernetes提供一个golang客户端库 （其他语言库正在开发中-如Python）。

对象（Object）规范和状态

每个Kubernetes对象都包含两个嵌套对象字段，用于管理Object的配置：Object Spec和Object Status。Spec描述了对象所需的状态 - 希望Object具有的特性，Status描述了对象的实际状态，并由Kubernetes系统提供和更新。

例如，通过Kubernetes Deployment 来表示在集群上运行的应用的对象。创建Deployment时，可以设置Deployment Spec，来指定要运行应用的三个副本。Kubernetes系统将读取Deployment Spec，并启动你想要的三个应用实例 - 来更新状态以符合之前设置的Spec。如果这些实例中有任何一个失败（状态更改），Kuberentes系统将响应Spec和当前状态之间差异来调整，这种情况下，将会开始替代实例。

有关object spec、status和metadata更多信息，请参考“Kubernetes API Conventions”。

描述Kubernetes对象

在Kubernetes中创建对象时，必须提供描述其所需Status的对象Spec，以及关于对象（如name）的一些基本信息。当使用Kubernetes API创建对象（直接或通过kubectl）时，该API请求必须将该信息作为JSON包含在请求body中。通常，可以将信息提供给kubectl .yaml文件，在进行API请求时，kubectl将信息转换为JSON。

以下示例是一个.yaml文件，显示Kubernetes Deployment所需的字段和对象Spec：

nginx-deployment.yaml

```
apiVersion: apps/v1beta1 kind: Deployment metadata: name: nginx-deployment spec: replicas: 3 template: metadata: labels: app: nginx spec: containers: - name: nginx image: nginx:1.7.9 ports: - containerPort: 80
```

使用上述.yaml文件创建Deployment，是通过在kubectl中使用kubectl create命令来实现。将该.yaml文件作为参数传递。如下例子：

```
$ kubectl create -f docs/user-guide/nginx-deployment.yaml --record
```

其输出与此类似:

```
deployment "nginx-deployment" created
```

必填字段

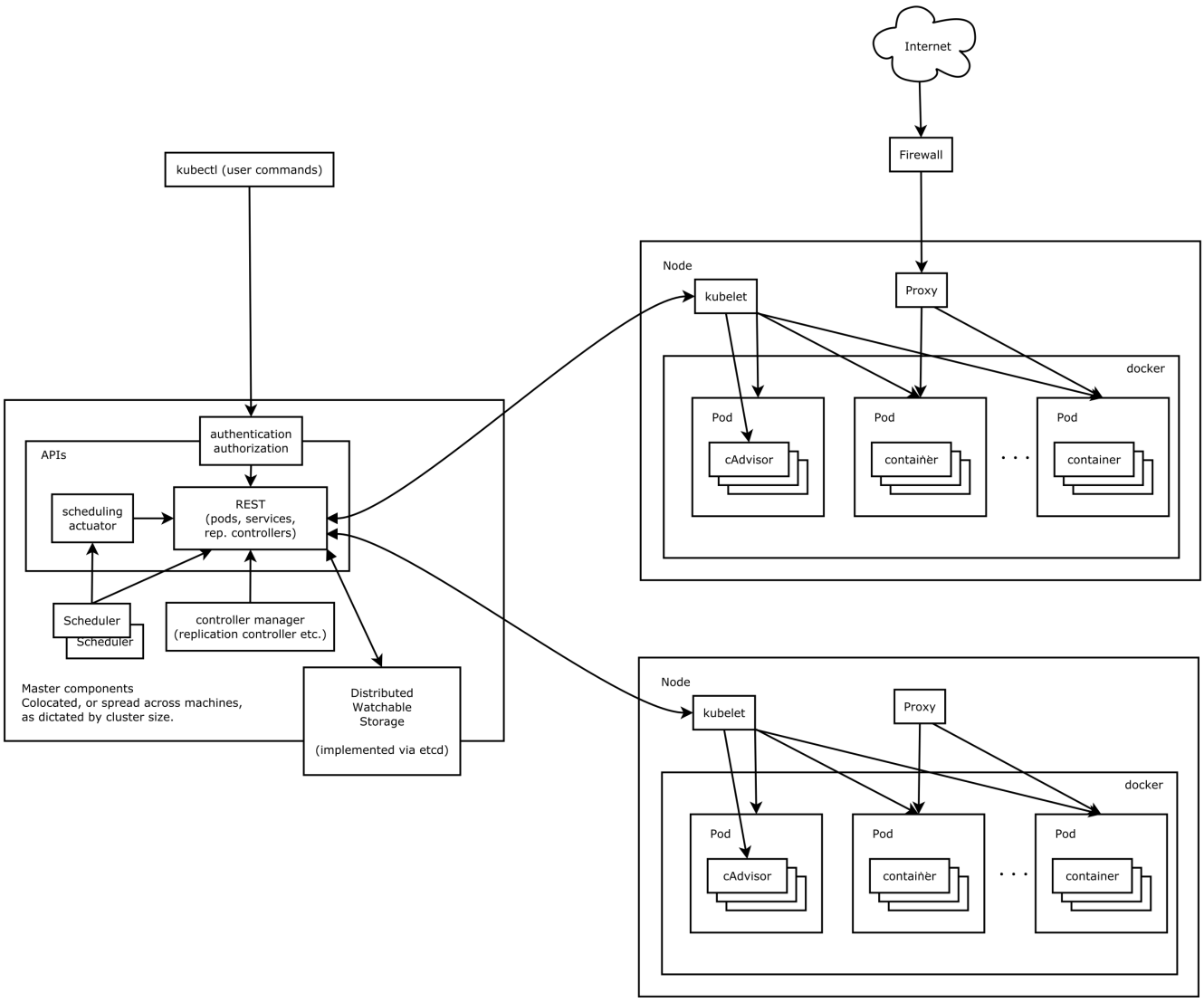
对于要创建的Kubernetes对象的yaml文件，需要为以下字段设置值：

- apiVersion - 创建对象的Kubernetes API 版本
- kind - 要创建什么样的对象?
- metadata- 具有唯一标示对象的数据，包括 name（字符串）、UID和Namespace（可选项）

还需要提供对象Spec字段，对象Spec的精确格式（对于每个Kubernetes 对象都是不同的），以及容器内嵌套的特定于该对象的字段。Kubernetes API reference可以查找所有可创建Kubernetes对象的Spec格式。

3 Kubernetes集群环境搭建

3.1 系统整体架构



架构图参考链接：https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.2/docs/design/architecture.png

3.2 环境规划

3.3 各Node 安装Docker

3.4 kubeadm初始化Kubernetes集群

3.5 部署Master组件

3.6 部署Node组件

3.7 创建Node节点kubeconfig配置文件

3.8 Flannel网络工作原理

overlay网络简介

覆盖网络就是应用层网络，它是面向应用层的，不考虑或很少考虑网络层，物理层的问题。

详细说来，覆盖网络是指建立在另一个网络上的网络。该网络中的结点可以看作通过虚拟或逻辑链路而连接起来的。虽然在底层有很多条物理链路，但是这些虚拟或逻辑链路都与路径一一对应。例如：许多P2P网络就是覆盖网络，因为它运行在互连网的上层。覆盖网络允许对没有IP地址标识的目的主机路由信息，例如：Freenet 和DHT（分布式哈希表）可以路由信息到一个存储特定文件的结点，而这个结点的IP地址事先并不知道。

覆盖网络被认为是一条用来改善互连网路由的途径，让二层网络在三层网络中传递，既解决了二层的缺点，又解决了三层的不灵活！

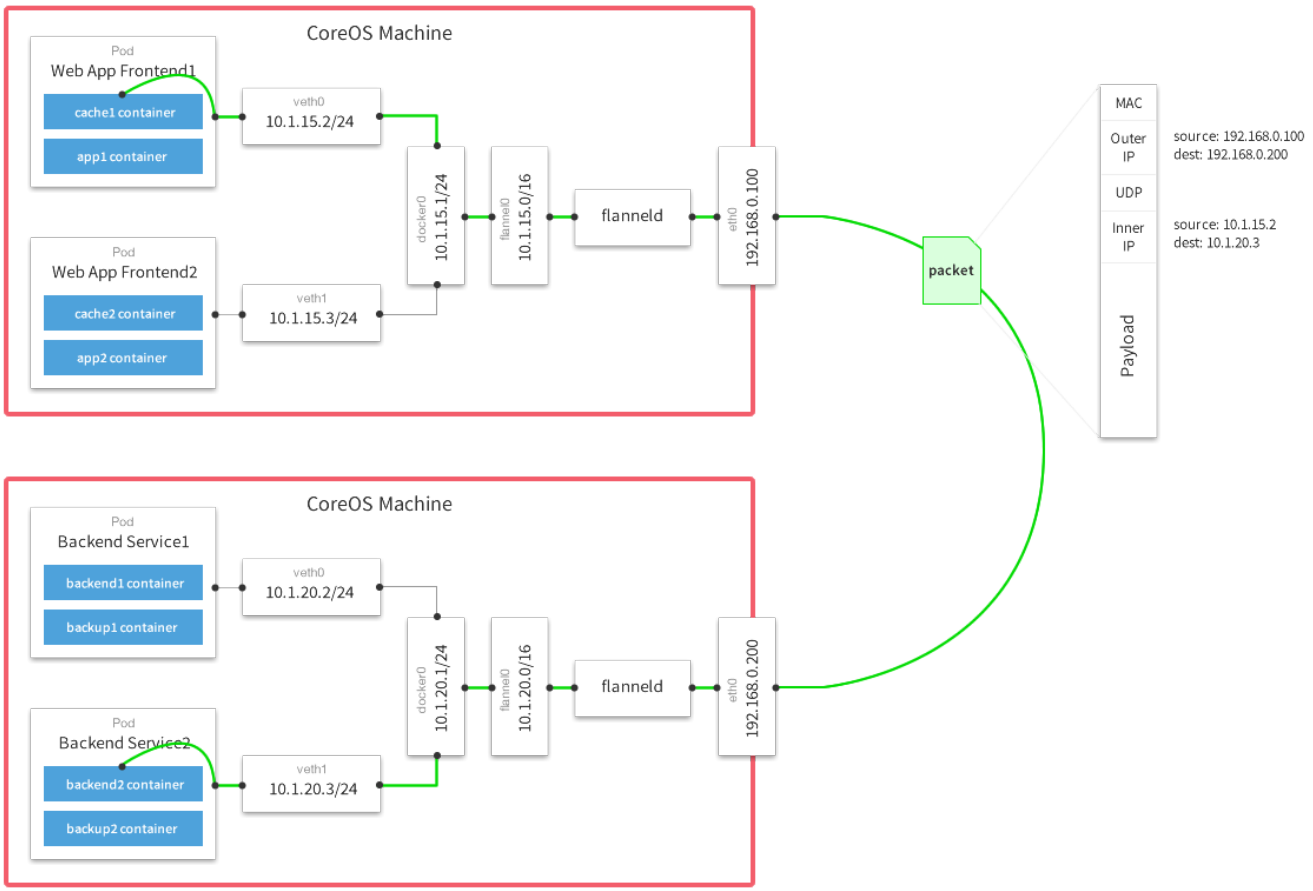
Flannel的工作原理

flannel是CoreOS提供用于解决Dokcer集群跨主机通讯的覆盖网络工具。它的主要思路是：预先留出一个网段，每个主机使用其中一部分，然后每个容器被分配不同的ip；让所有的容器认为大家在同一个直连的网络，底层通过`UDP/VxLAN`等进行报文的封装和转发。

flannel项目地址：https://github.com/coreos/flannel

flannel架构介绍

flannel默认使用8285端口作为`UDP`封装报文的端口，VxLan使用8472端口。



那么一条网络报文是怎么从一个容器发送到另外一个容器的呢?

1. 容器直接使用目标容器的ip访问，默认通过容器内部的eth0发送出去。
2. 报文通过`veth pair`被发送到`vethXXX`。
3. `vethXXX`是直接连接到虚拟交换机`docker0`的，报文通过虚拟`bridge docker0`发送出去。
4. 查找路由表，外部容器ip的报文都会转发到`flannel0`虚拟网卡，这是一个`P2P`的虚拟网卡，然后报文就被转发到监听在另一端的`flanneld`。
5. `flanneld`通过`etcd`维护了各个节点之间的路由表，把原来的报文`UDP`封装一层，通过配置的`iface`发送出去。
6. 报文通过主机之间的网络找到目标主机。
7. 报文继续往上，到传输层，交给监听在8285端口的`flanneld`程序处理。
8. 数据被解包，然后发送给`flannel0`虚拟网卡。
9. 查找路由表，发现对应容器的报文要交给`docker0`。
10. `docker0`找到连到自己的容器，把报文发送过去。

3.9 Flannel容器集群网络部署

参考：https://blog.csdn.net/bbwangj/article/details/81092728

https://www.jianshu.com/p/165a256fb1da

4 kubernetes实践

4.1 运行一个测试示例.NET CORE

4.2 代码剖析

4.3 中间件-配置

4.4 编译打docker镜像

4.5 运行k8s命令部署服务

4.6 演示网站

5 kubectl 命令行管理工具

5.1 kubectl用法概述

kubectl用于运行Kubernetes集群命令的管理工具。

可以在 https://github.com/kubernetes/kubernetes 找到更多的信息。

安装和设置kubectl

使用Kubernetes命令行工具kubectl在Kubernetes上部署和管理应用程序。使用kubectl，可以检查集群资源; 创建，删除和更新组件。

以下是安装kubectl的几种方法。

通过curl安装kubectl二进制文件

Linux

下载最新版本的命令：

```
 curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-
release/release/stable.txt)/bin/linux/amd64/kubectl
```

要下载特定版本，请使用特定版本替换$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)命令的一部分。

例如，要在Linux上下载v1.14.0版本，请键入：

```
 curl -LO https://storage.googleapis.com/kubernetes-
release/release/v1.14.0/bin/linux/amd64/kubectl
```

使kubectl二进制可执行。

```
 chmod +x ./kubectl
```

将二进制文件移动到PATH中。

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

Windows

从此链接下载最新版本v1.14.0 。

如果curl已安装，请使用以下命令：

```
curl -LO https://storage.googleapis.com/kubernetes-
release/release/v1.14.0/bin/windows/amd64/kubectl.exe
```

要了解最新的稳定版本，请查看 https://storage.googleapis.com/kubernetes-release/release/stable.txt

将二进制文件添加到PATH中。

配置kubectl

为了使kubectl找到并访问Kubernetes集群，需要一个kubeconfig文件，当你使用kube-up.sh创建集群或成功部署Minikube集群时，该文件将自动创建。有关创建集群的更多信息，请参阅入门指南。如果你需要访问未创建的群集，请参阅共享群集访问文档。默认情况下，kubectl配置位于~/.kube/config。

检查kubectl配置

通过获取集群状态来检查kubectl是否正确配置：

```
$ kubectl cluster-info
```

如果看到一个URL响应，kubectl被正确配置为访问您的集群。

如果看到类似于以下内容的消息，则kubectl未正确配置：

```
The connection to the server <server-name:port> was refused - did you specify the right
host or port?
```

启用shell自动完成

kubectl包括支持自动完成，可以节省大量打字！

完成脚本本身是由kubectl生成的，所以你通常只需要从你的配置文件中调用它。

这里提供常见的例子。有关详细信息，请咨询kubectl completion -h。

在Linux上，使用bash

要将kubectl自动完成添加到当前shell，请运行source <(kubectl completion bash)。

要将kubectl自动完成添加到你的配置文件中，因此将在以后的shell中自动加载运行：

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

Kubernetes kubectl 命令表

> kubectl命令列表

- kubectl run（创建容器镜像）
- kubectl expose（将资源暴露为新的 Service）
- kubectl annotate（更新资源的Annotations信息）
- kubectl autoscale（Pod水平自动伸缩）
- kubectl convert（转换配置文件为不同的API版本）

- kubectl create（创建一个集群资源对象
- kubectl create clusterrole（创建ClusterRole）
- kubectl create clusterrolebinding（为特定的ClusterRole创建ClusterRoleBinding）
- kubectl create configmap（创建configmap）
- kubectl create deployment（创建deployment）
- kubectl create namespace（创建namespace）
- kubectl create poddisruptionbudget（创建poddisruptionbudget）
- kubectl create quota（创建resourcequota）
- kubectl create role（创建role）
- kubectl create rolebinding（为特定Role或ClusterRole创建RoleBinding）

- kubectl create service（使用指定的子命令创建 Service服务）
- kubectl create service clusterip
- kubectl create service externalname
- kubectl create service loadbalancer
- kubectl create service nodeport
- kubectl create serviceaccount

- kubectl create secret（使用指定的子命令创建 secret）
- kubectl create secret tls
- kubectl create secret generic
- kubectl create secret docker-registry

- kubectl delete（删除资源对象）
- kubectl edit（编辑服务器上定义的资源对象）
- kubectl get（获取资源信息）
- kubectl label（更新资源对象的label）
- kubectl patch（使用patch更新资源对象字段）
- kubectl replace（替换资源对象）
- kubectl rolling-update（使用RC进行滚动更新）
- kubectl scale（扩缩Pod数量）

- kubectl rollout（对资源对象进行管理）
- kubectl rollout history（查看历史版本）
- kubectl rollout pause（标记资源对象为暂停状态）
- kubectl rollout resume（恢复已暂停资源）
- kubectl rollout status（查看资源状态）
- kubectl rollout undo（回滚版本）

- kubectl set（配置应用资源）
- kubectl set resources（指定Pod的计算资源需求）
- kubectl set selector（设置资源对象selector）

- kubectl set image（更新已有资源对象中的容器镜像）
- kubectl set subject（更新RoleBinding / ClusterRoleBinding中User、Group 或 ServiceAccount）

参考：https://www.kubernetes.org.cn/doc-45

5.2 kubectl命令详解

语法

在管理工具界面使用kubectl语法运行如下命令：

```
kubectl [command] [TYPE] [NAME] [flags]
```

其中command，TYPE，NAME，和flags都是：* command：指定要在一个或多个资源执行的操作，例如操作create，get，describe，delete。* TYPE：指定资源类型Resource types。Resource types会区分大小写，也可以指定单数，复数或缩写的形式。例如，以下命令将输出相同的结果：

shell $ kubectl get pod pod1 $ kubectl get pods pod1 $ kubectl get po pod1 * NAME：指定Resource的Name。Name区分大小写，如果省略Name，则显示所有资源的详细信息，例如：$ kubectl get pods。

当在多个资源上执行操作时，可以通过type和name 指定每个资源，或者指定一个或多个file:通过type和name指定的资源:如果它们都是相同的type，就可以对资源进行分组TYPE1 name1 name2 name<#>

- 例：$ kubectl get pod example-pod1 example-pod2 *单独指定多种资源type：TYPE1/name1 TYPE1/name2 TYPE2/name3 TYPE<#>/name<#>
- 例：$ kubectl get pod/example-pod1 replicationcontroller/example-rc1 *使用一个或多个file来指定资源：-f file1 -f file2 -f file<#> 使用YAML而不是JSON，因为YAML往往更容易掌握也对用户更友好，特别是对于配置文件。
- 例：$ kubectl get pod -f ./pod.yaml * flags：指定可选flags。例如，你可以使用-s或--server flag来指定Kubernetes API Server的地址和端口。

提示：命令行指定的flags将覆盖默认值和任何相应的环境变量。

> 如果需要更多相关帮助，只需从终端命令窗口运行 kubectl help

Operations

下表包括了所有kubectl操作简短描述和通用语法：

Operation	Syntax	Description
annotate	kubectl annotate (-f FILENAME | TYPE NAME | TYPE/NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--overwrite] [--all] [--resource-version=version] [flags]	为一个或多个资源添加注释
api-versions	kubectl api-versions [flags]	列出支持的API版本。
apply	kubectl apply -f FILENAME [flags]	对文件或stdin的资源进行配置更改。
attach	kubectl attach POD -c CONTAINER [-i] [-t] [flags]	连接到一个运行的容器，既可以查看output stream，也可以与容器(stdin)进行交互。
autoscale	kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min=MINPODS] --max=MAXPODS [--cpu-percent=CPU] [flags]	自动扩容/缩容由replication controller管理的一组pod。
cluster-info	kubectl cluster-info [flags]	显示有关集群中master和services的终端信息。
config	kubectl config SUBCOMMAND [flags]	修改kubeconfig文件。有关详细信息，请参阅各个子命令。
create	kubectl create -f FILENAME [flags]	从file或stdin创建一个或多个资源。
delete	kubectl delete (-f FILENAME | TYPE [NAME | /NAME | -l label | --all]) [flags]	从file，stdin或指定label选择器，names，resource选择器或resources中删除resources。
describe	kubectl describe (-f FILENAME | TYPE [NAME_PREFIX | /NAME | -l label]) [flags]	显示一个或多个resources的详细状态。
edit	kubectl edit (-f FILENAME | TYPE NAME | TYPE/NAME) [flags]	使用默认编辑器编辑和更新服务器上一个或多个定义的资源。

Operation	Syntax	Description
exec	kubectl exec POD [-c CONTAINER] [-i] [-t] [flags] [--COMMAND [args...]]	对pod中的容器执行命令。
explain	kubectl explain [--include-extended-apis=true] [--recursive=false] [flags]	获取各种资源的文档。例如pod，node，services等
expose	kubectl expose (-f FILENAME | TYPE NAME | TYPE/NAME) [--port=port] [--protocol=TCP|UDP] [--target-port=number-or-name] [--name=name] [----external-ip=external-ip-of-service] [--type=type] [flags]	将 replication controller，service或pod作为一个新的Kubernetes service显示。
get	kubectl get (-f FILENAME | TYPE [NAME | /NAME | -l label]) [--watch] [--sort-by=FIELD] [[-o | --output]=OUTPUT_FORMAT] [flags]	列出一个或多个资源。
label	kubectl label (-f FILENAME | TYPE NAME | TYPE/NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--overwrite] [--all] [--resource-version=version] [flags]	添加或更新一个或多个资源的flags。
logs	kubectl logs POD [-c CONTAINER] [--follow] [flags]	在pod中打印容器的日志。
patch	kubectl patch (-f FILENAME | TYPE NAME | TYPE/NAME) --patch PATCH [flags]	使用strategic merge 补丁程序更新资源的一个或多个字段。
port-forward	kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT [...[LOCAL_PORT_N:]REMOTE_PORT_N] [flags]	将一个或多个本地端口转发到pod。
proxy	kubectl proxy [--port=PORT] [--www=static-dir] [--www-prefix=prefix] [--api-prefix=prefix] [flags]	在Kubernetes API服务器运行代理。
replace	kubectl replace -f FILENAME	从file或stdin替换资源。
rolling-update	kubectl rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] --image=NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC) [flags]	通过逐步替换指定的replication controller及其pod来执行滚动更新。
run	kubectl run NAME --image=image [--env="key=value"] [--port=port] [--replicas=replicas] [--dry-run=bool] [--overrides=inline-json] [flags]	在集群上运行指定的镜像。
scale	kubectl scale (-f FILENAME | TYPE NAME | TYPE/NAME) --replicas=COUNT [--resource-version=version] [--current-replicas=count] [flags]	更新指定replication controller的大小。
stop	kubectl stop	已弃用：请参阅kubectl delete。

Operation	Syntax	Description
version	kubectl version [--client] [flags]	显示客户端和服务器上运行的Kubernetes版本。

提示：有关更多命令信息，请参阅kubectl参考文档。

Resource types

下表列出了所有支持的资源类型及其缩写：

Resource type	Abbreviated alias
apiservices	
certificatesigningrequests	csr
clusters	
clusterrolebindings	
clusterroles	
componentstatuses	cs
configmaps	cm
controllerrevisions	
cronjobs	
customresourcedefinition	crd
daemonsets	ds
deployments	deploy
endpoints	ep
events	ev
horizontalpodautoscalers	hpa
ingresses	ing
jobs	
limitranges	limits
namespaces	ns
networkpolicies	netpol
nodes	no
persistentvolumeclaims	pvc
persistentvolumes	pv
poddisruptionbudget	pdb
podpreset	
pods	po
podsecuritypolicies	psp
podtemplates	

Resource type	Abbreviated alias
replicasets	rs
replicationcontrollers	rc
resourcequotas	quota
rolebindings	
roles	
secrets	
serviceaccounts	sa
services	svc
statefulsets	
storageclasses	

输出选项 Output options

使用以下部分来了解如何格式化或对某些命令的输出进行排序。关于哪些命令支持什么输出选项，请查阅kubectl参考文档。

格式化输出 Formatting output

所有kubectl命令输出的默认格式是可读的纯文本格式。要以特定的格式向终端窗口输出详细信息，可以将-o或-output flags 添加到支持的kubectl命令中。

语法：

```
kubectl [command] [TYPE] [NAME] -o=<output_format>
```

根据kubectl操作，支持以下输出格式：

输出格式	描述
-o=custom-columns=	使用逗号分隔的custom columns列表打印一个表。
-o=custom-columns-file=	使用文件中的custom columns模板打印表。
-o=json	输出JSON格式的API对象。
-o=jsonpath=	