



Kubernetes Handbook

Jimmy Song

目錄

1. 前言	1.1
2. 概念原理	1.2
2.1 设计理念	1.2.1
2.2 主要概念	1.2.2
2.2.1 Pod	1.2.2.1
2.2.1.1 Pod解析	1.2.2.1.1
2.2.2 Node	1.2.2.2
2.2.3 Namespace	1.2.2.3
2.2.4 Service	1.2.2.4
2.2.5 Volume和Persistent Volume	1.2.2.5
2.2.6 Deployment	1.2.2.6
2.2.7 Secret	1.2.2.7
2.2.8 StatefulSet	1.2.2.8
2.2.9 DaemonSet	1.2.2.9
2.2.10 ServiceAccount	1.2.2.10
2.2.11 ReplicationController和ReplicaSet	1.2.2.11
2.2.12 Job	1.2.2.12
2.2.13 CronJob	1.2.2.13
2.2.14 Ingress	1.2.2.14
2.2.15 ConfigMap	1.2.2.15
2.2.16 Horizontal Pod Autoscaling	1.2.2.16
2.2.17 Label	1.2.2.17
3. 用户指南	1.3
3.1 配置Pod的liveness和readiness探针	1.3.1
3.2 管理集群中的TLS	1.3.2
3.3 使用kubectl	1.3.3
3.4 适用于kubernetes的应用开发部署流程	1.3.4

3.5 配置Pod的Service Account	1.3.5
3.6 访问集群	1.3.6
4. 最佳实践	1.4
4.1 在CentOS上部署kubernetes1.6集群	1.4.1
4.1.1 创建TLS证书和秘钥	1.4.1.1
4.1.2 创建kubeconfig文件	1.4.1.2
4.1.3 创建高可用etcd集群	1.4.1.3
4.1.4 安装kubectl命令行工具	1.4.1.4
4.1.5 部署master节点	1.4.1.5
4.1.6 部署node节点	1.4.1.6
4.1.7 安装kubedns插件	1.4.1.7
4.1.8 安装dashboard插件	1.4.1.8
4.1.9 安装heapster插件	1.4.1.9
4.1.10 安装EFK插件	1.4.1.10
4.2 服务发现与负载均衡	1.4.2
4.2.1 安装Traefik ingress	1.4.2.1
4.2.2 分布式负载测试	1.4.2.2
4.2.3 网络和集群性能测试	1.4.2.3
4.2.4 边缘节点配置	1.4.2.4
4.3 运维管理	1.4.3
4.3.1 服务滚动升级	1.4.3.1
4.3.2 应用日志收集	1.4.3.2
4.3.3 配置最佳实践	1.4.3.3
4.3.4 集群及应用监控	1.4.3.4
4.3.5 使用Jenkins进行持续构建与发布	1.4.3.5
4.3.6 数据持久化问题	1.4.3.6
4.4 存储管理	1.4.4
4.4.1 GlusterFS	1.4.4.1
4.4.1.1 使用GlusterFS做持久化存储	1.4.4.1.1
4.4.1.2 在OpenShift中使用GlusterFS做持久化存储	1.4.4.1.2

5. 领域应用	1.5
5.1 微服务架构	1.5.1
5.1.1 Istio	1.5.1.1
5.1.1.1 安装istio	1.5.1.1.1
5.1.1.2 配置请求的路由规则	1.5.1.1.2
5.1.2 Linkerd	1.5.1.2
5.1.2.1 Linkerd 使用指南	1.5.1.2.1
5.1.3 微服务中的服务发现	1.5.1.3
5.2 大数据	1.5.2
5.2.1 Spark on Kubernetes	1.5.2.1
6. 开发指南	1.6
6.1 开发环境搭建	1.6.1
6.2 单元测试和集成测试	1.6.2
6.3 client-go示例	1.6.3
6.4 社区贡献	1.6.4
7. 附录	1.7
7.1 Docker最佳实践	1.7.1
7.2 问题记录	1.7.2
7.3 使用技巧	1.7.3

build passing

Figure: wercker status

Kubernetes Handbook

Kubernetes 是 Google 基于 **Borg** 开源的容器编排调度引擎，作为 **CNCF** (Cloud Native Computing Foundation) 最重要的组件之一，它的目标不仅仅是一个编排系统，而是提供一个规范，可以让你来描述集群的架构，定义服务的最终状态，它将自动得将系统达到和维持在这个状态。

本书记录了本人从零开始学习和使用 **Kubernetes** 的心路历程，着重于经验分享和总结，同时也会有相关的概念解析，希望能够帮助大家少踩坑，少走弯路。

在写作本书时，安装的所有组件、所用示例和操作等皆基于 **Kubernetes1.6.0** 版本。

文章目录

GitHub 地址：<https://github.com/rootsongjc/kubernetes-handbook>

Gitbook 在线浏览：<https://www.gitbook.com/book/rootsongjc/kubernetes-handbook/>

如何使用本书

在线浏览

访问 [gitbook](#)

注意：文中涉及的配置文件和代码链接在 gitbook 中会无法打开，请下载 github 源码后，在 **MarkDown** 编辑器中打开，点击链接将跳转到你的本地目录，推荐使用 [typora](#)。

本地查看

1. 将代码克隆到本地
2. 安装 gitbook：[Setup and Installation of GitBook](#)

1. 前言

3. 执行 `gitbook serve`
4. 在浏览器中访问<http://localhost:4000>
5. 生成的文档在 `_book` 目录下

生成 **pdf**

[下载Calibre](#)

- **On Mac**

在Mac下安装后，使用该命令创建链接

```
ln -s /Applications/calibre.app/Contents/MacOS/ebook-convert /usr/local/bin
```

在该项目目录下执行以下命令生成 `kubernetes-handbook.pdf` 文档。

```
gitbook pdf . ./kubernetes-handbook.pdf
```

- **On Windows**

需要用到的工具：[calibre](#)，[phantomjs](#)

1. 将上述2个安装，`calibre` 默认安装的路径 `C:\Program Files\Calibre2` 为你解压路径；
2. 并将其目录均加入到系统变量 `path` 中,参考:[目录添加到系统变量 path 中](#)；
3. 在 `cmd` 打开你需要转 `pdf` 的文件夹,输入 `gitbook pdf` 即可；

生成单个章节的**pdf**

使用 `pandoc` 和 `latex` 来生成pdf格式文档。

```
pandoc --latex-engine=xelatex --template=pm-template input.md -o output.pdf
```

如何贡献

提 **issue**

1. 前言

如果你发现文档中的错误，或者有好的建议，不要犹豫，欢迎 [提交issue](#)。

发起 Pull Request

当你发现文章中明确的错误或者逻辑问题，在你自己的 fork 的分支中，创建一个新的 branch，修改错误，push 到你的 branch，然后在 [提交issue](#) 后直接发起 Pull Request。

贡献文档

文档的组织规则

- 如果要创建一个大的主题就在最顶层创建一个目录；
- 全书五大主题，每个主题一个目录，其下不再设二级目录；
- 所有的图片都放在最顶层的 `images` 目录下，原则上文章中用到的图片都保存在本地；
- 所有的文档的文件名使用英文命名，可以包含数字和中划线；
- `etc`、`manifests` 目录专门用来保存配置文件和文档中用到的其他相关文件；

添加文档

1. 在该文章相关主题的目录下创建文档；
2. 在 `SUMMARY.md` 中在相应的章节下添加文章链接；
3. 执行 `gitbook server` 测试是否报错，访问 <http://localhost:4000> 查看该文档是否出现在相应主题的目录下；
4. 提交PR

关于

贡献者列表

[Jimmy Song](#)

for GitBook update 2017-08-07 13:54:27

1. 前言

Kubernetes架构

Kubernetes最初源于谷歌内部的Borg，提供了面向应用的容器集群部署和管理系统。Kubernetes的目标旨在消除编排物理/虚拟计算，网络和存储基础设施的负担，并使应用程序运营商和开发人员完全将重点放在以容器为中心的原语上进行自助运营。Kubernetes也提供稳定、兼容的基础（平台），用于构建定制化的workflows和更高级的自动化任务。Kubernetes具备完善的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建负载均衡器、故障发现和自我修复能力、服务滚动升级和在线扩容、可扩展的资源自动调度机制、多粒度的资源配置管理能力。Kubernetes还提供完善的管理工具，涵盖开发、部署测试、运维监控等各个环节。

Borg简介

Borg是谷歌内部的大规模集群管理系统，负责对谷歌内部很多核心服务的调度和管理。Borg的目的是让用户能够不必操心资源管理的问题，让他们专注于自己的核心业务，并且做到跨多个数据中心的资源利用率最大化。

Borg主要由BorgMaster、Borglet、borgcfg和Scheduler组成，如下图所示

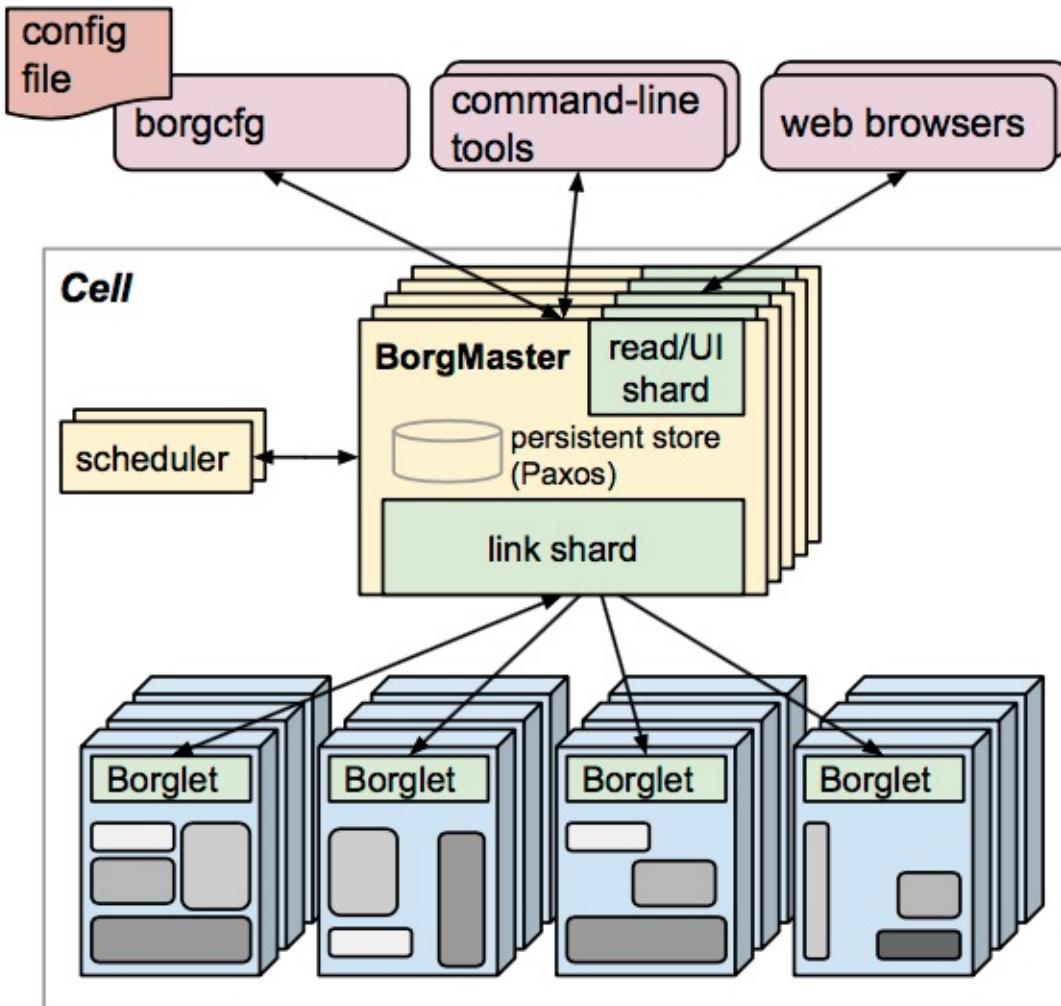


Figure: Borg架构

- BorgMaster是整个集群的大脑，负责维护整个集群的状态，并将数据持久化到 Paxos存储中；
- Scheduler负责任务的调度，根据应用的特点将其调度到具体的机器上去；
- Borglet负责真正运行任务（在容器中）；
- borgcfg是Borg的命令行工具，用于跟Borg系统交互，一般通过一个配置文件来提交任务。

Kubernetes 架构

Kubernetes借鉴了Borg的设计理念，比如Pod、Service、Labels和单Pod单IP等。Kubernetes的整体架构跟Borg非常像，如下图所示

2. 概念原理

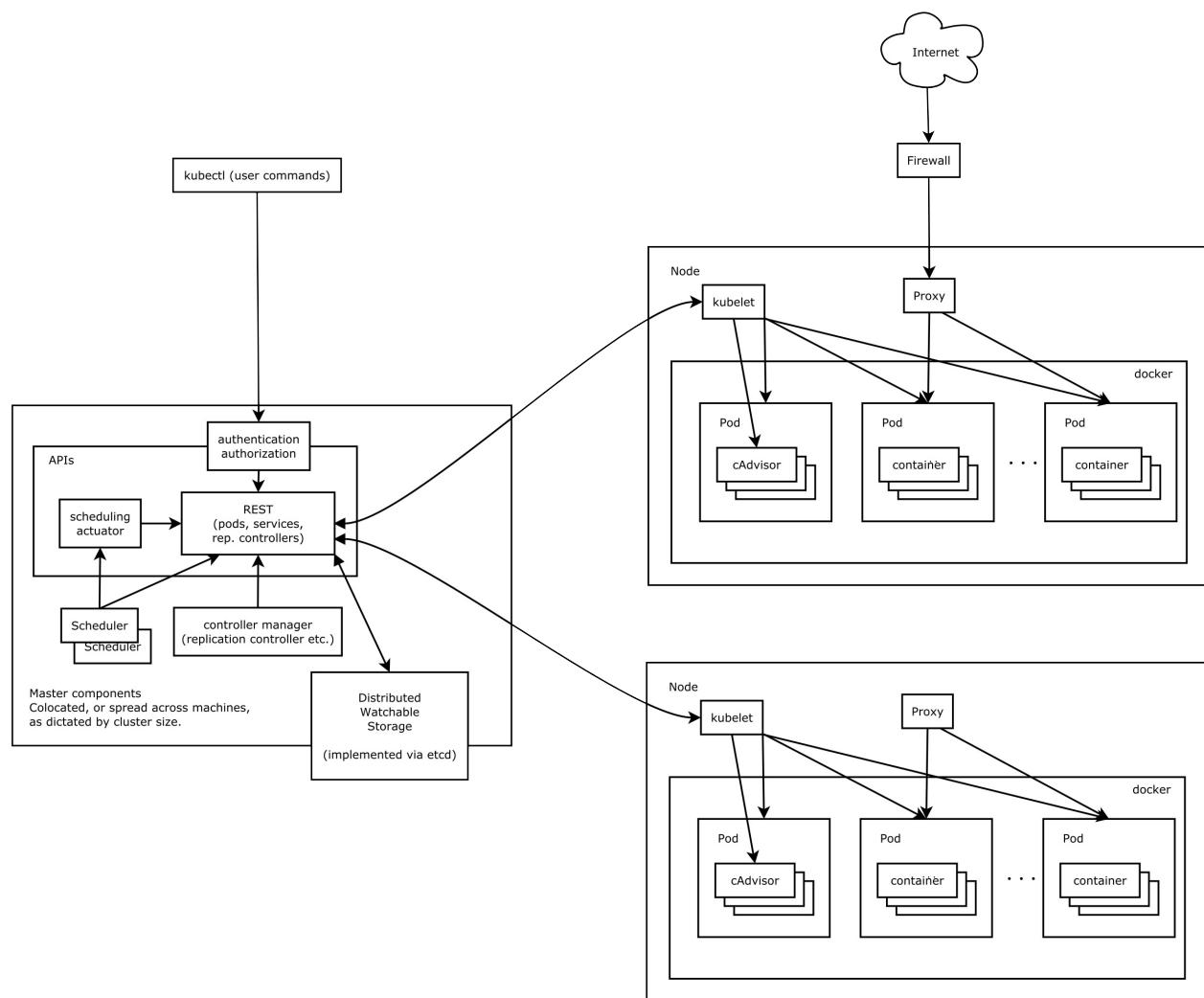


Figure: Kubernetes架构

Kubernetes主要由以下几个核心组件组成：

- etcd保存了整个集群的状态；
- apiserver提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制；
- controller manager负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- scheduler负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上；
- kubelet负责维护容器的生命周期，同时也负责Volume (CVI) 和网络 (CNI) 的管理；
- Container runtime 负责镜像管理以及Pod和容器的真正运行 (CRI) ；
- kube-proxy负责为Service提供cluster内部的服务发现和负载均衡；

除了核心组件，还有一些推荐的Add-ons：

2. 概念原理

- kube-dns 负责为整个集群提供 DNS 服务
- Ingress Controller 为服务提供外网入口
- Heapster 提供资源监控
- Dashboard 提供 GUI
- Federation 提供跨可用区的集群

Kubernetes 架构示意图

整体架构

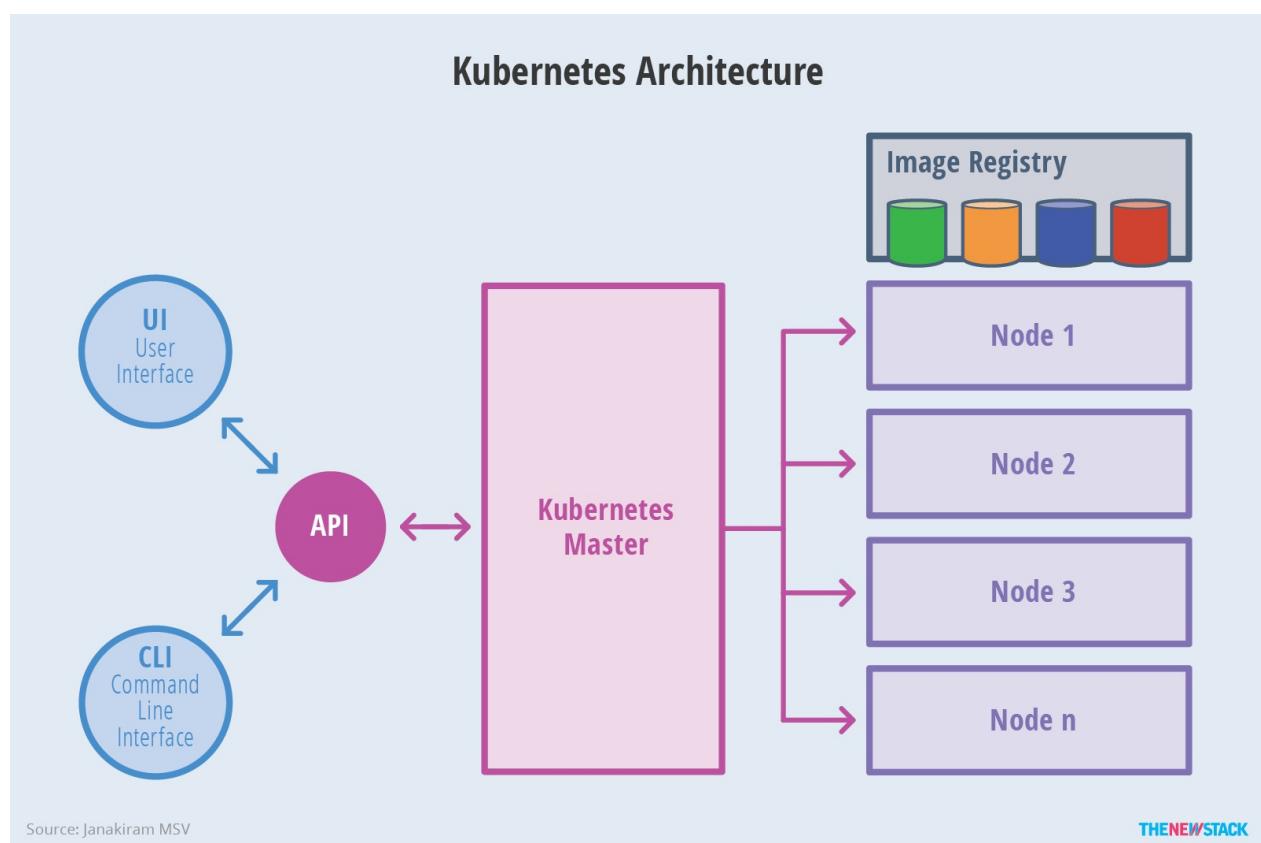


Figure: kubernetes 整体架构示意图

Master 架构

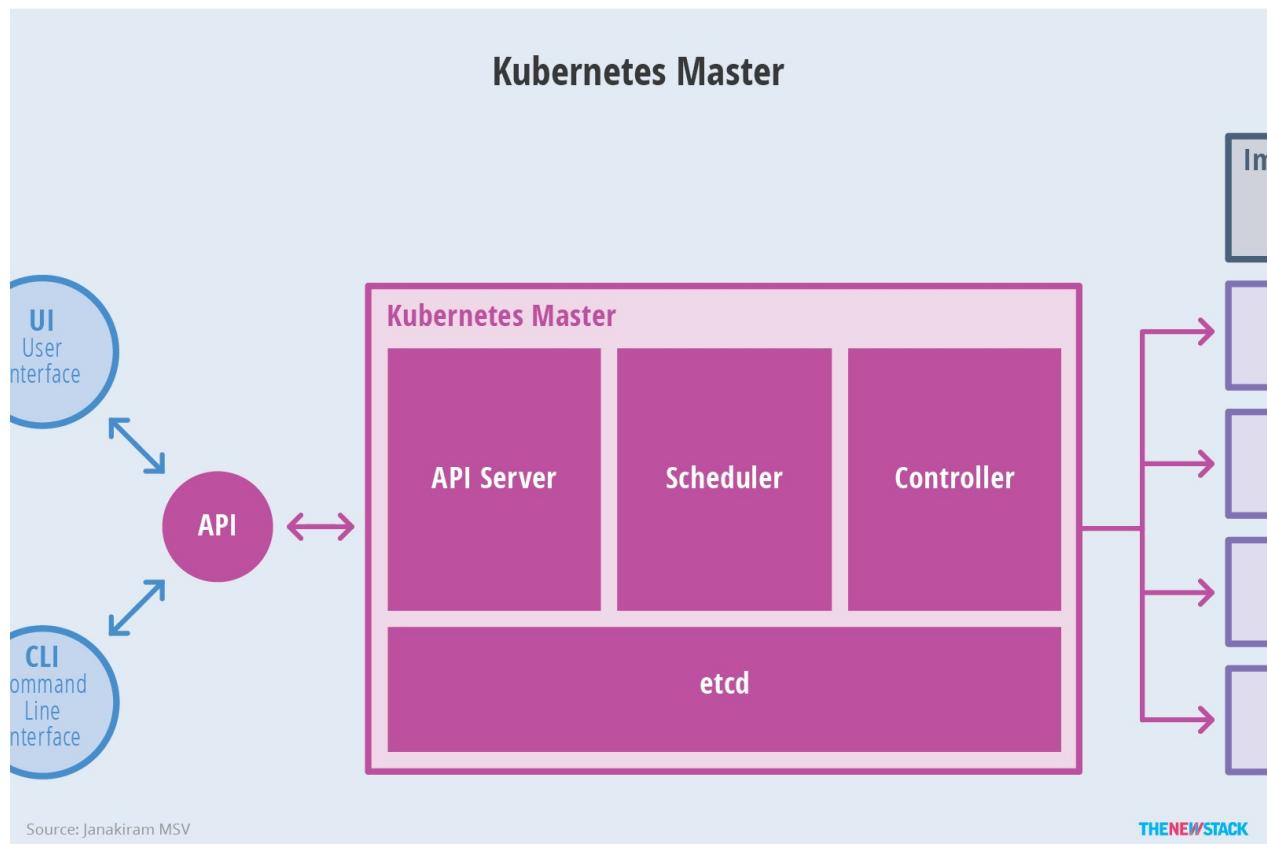


Figure: *Kubernetes master*架构示意图

Node 架构

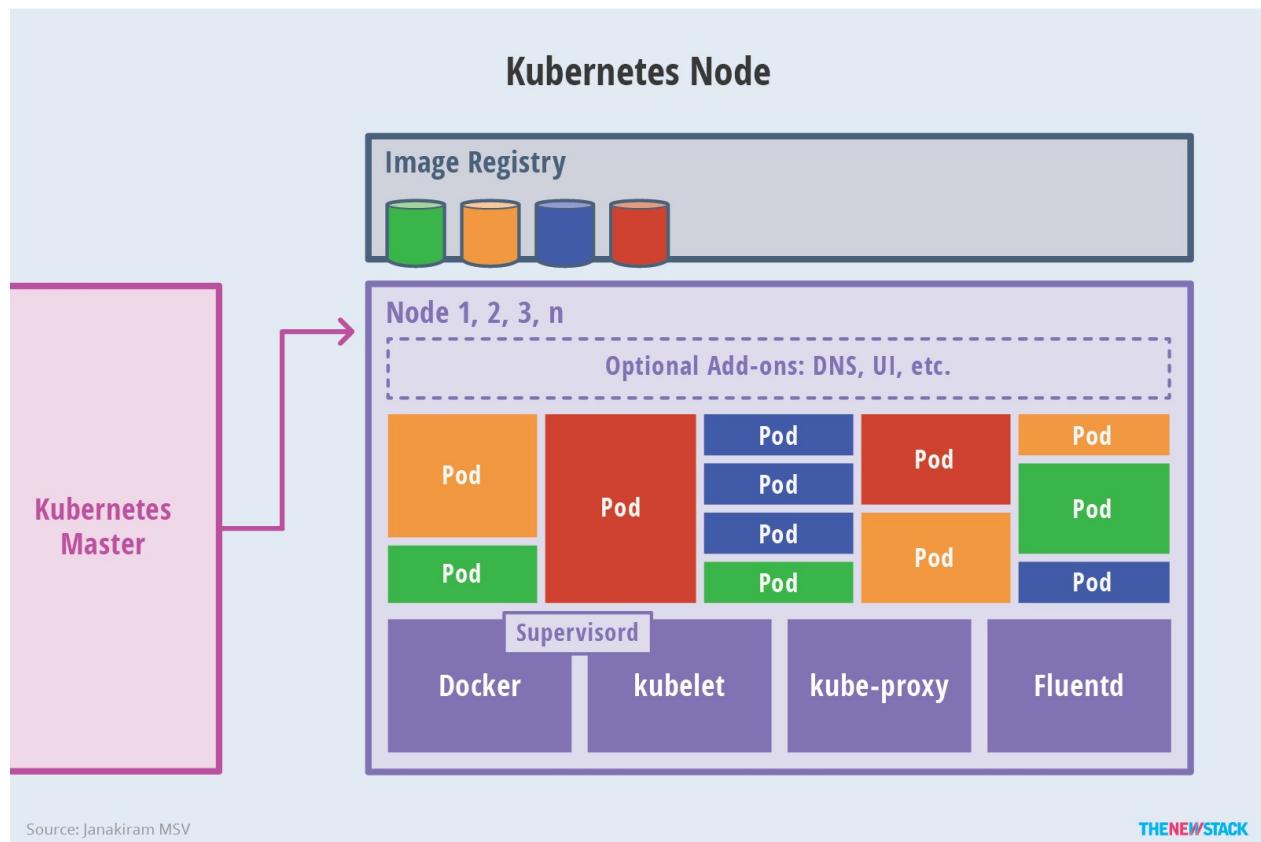


Figure: kubernetes node架构示意图

分层架构

Kubernetes设计理念和功能其实就是一个类似Linux的分层架构，如下图所示

2. 概念原理

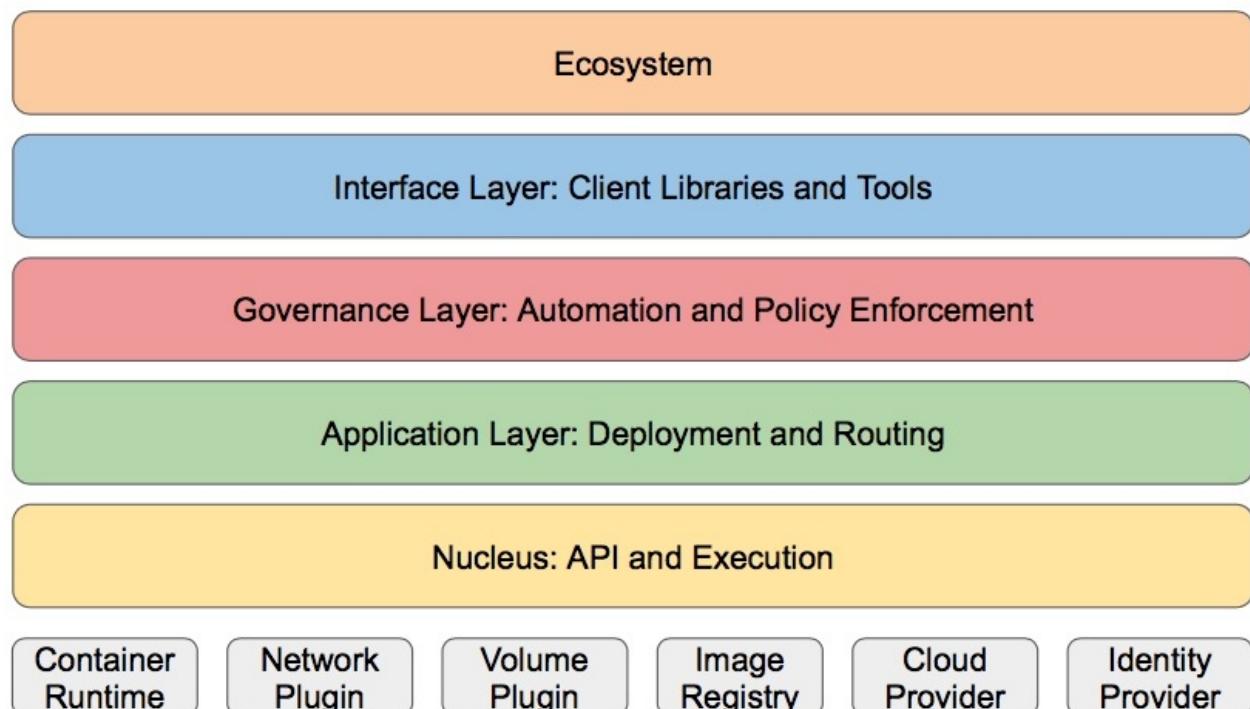


Figure: Kubernetes分层架构示意图

- 核心层：Kubernetes最核心的功能，对外提供API构建高层的应用，对内提供插件式应用执行环境
- 应用层：部署（无状态应用、有状态应用、批处理任务、集群应用等）和路由（服务发现、DNS解析等）
- 管理层：系统度量（如基础设施、容器和网络的度量），自动化（如自动扩展、动态Provision等）以及策略管理（RBAC、Quota、PSP、NetworkPolicy等）
- 接口层：kubectl命令行工具、客户端SDK以及集群联邦
- 生态系统：在接口层之上的庞大容器集群管理调度的生态系统，可以划分为两个范畴
 - Kubernetes外部：日志、监控、配置管理、CI、CD、Workflow、FaaS、OTS应用、ChatOps等
 - Kubernetes内部：CRI、CNI、CVI、镜像仓库、Cloud Provider、集群自身的配置和管理等

关于分层架构，可以关注下Kubernetes社区正在推进的[Kubernetes architectural roadmap](#)和[slide](#)。

参考文档

2. 概念原理

- Kubernetes design and architecture
- <http://queue.acm.org/detail.cfm?id=2898444>
- <http://static.googleusercontent.com/media/research.google.com/zh-CN//pubs/archive/43438.pdf>
- <http://thenewstack.io/kubernetes-an-overview>
- Kbernetes architectual roadmap和slide

for GitBook update 2017-08-07 13:54:27

Kubernetes的设计理念

Kubernetes设计理念与分布式系统

分析和理解Kubernetes的设计理念可以使我们更深入地了解Kubernetes系统，更好地利用它管理分布式部署的云原生应用，另一方面也可以让我们借鉴其在分布式系统设计方面的经验。

分层架构

Kubernetes设计理念和功能其实就是一个类似Linux的分层架构，如下图所示

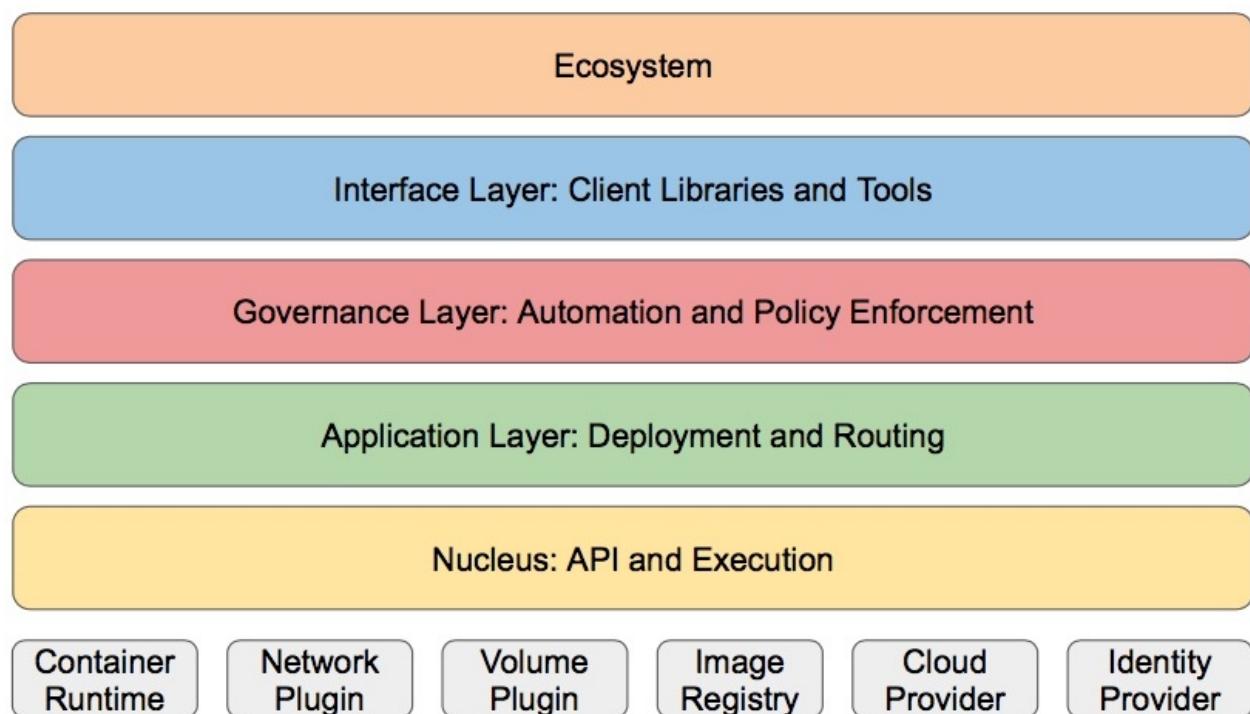


Figure: 分层架构示意图

- 核心层：Kubernetes最核心的功能，对外提供API构建高层的应用，最内提供插件式应用执行环境
- 应用层：部署（无状态应用、有状态应用、批处理任务、集群应用等）和路由（服务发现、DNS解析等）
- 管理层：系统度量（如基础设施、容器和网络的度量），自动化（如自动扩

展、动态 Provision 等) 以及策略管理 (RBAC、Quota、PSP、NetworkPolicy 等)

- 接口层：kubectl 命令行工具、客户端 SDK 以及集群联邦
- 生态系统：在接口层之上的庞大容器集群管理调度的生态系统，可以划分为两个范畴
 - Kubernetes 外部：日志、监控、配置管理、CI、CD、Workflow、FaaS、OTS 应用、ChatOps 等
 - Kubernetes 内部：CRI、CNI、CVI、镜像仓库、Cloud Provider、集群自身的配置和管理等

API 设计原则

对于云计算系统，系统 API 实际上处于系统设计的统领地位，正如本文前面所说，kubernetes 集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的 API 对象，支持对该功能的管理操作，理解掌握的 API，就好比抓住了 kubernetes 系统的牛鼻子。Kubernetes 系统 API 的设计有以下几条原则：

1. 所有 API 应该是声明式的。正如前文所说，声明式操作，相对于命令式操作，对于重复操作的效果是稳定的，这对于容易出现数据丢失或重复的分布式环境来说是很重要的。另外，声明式操作更容易被用户使用，可以使系统向用户隐藏实现的细节，隐藏实现的细节的同时，也就保留了系统未来持续优化的可能性。此外，声明式的 API，同时隐含了所有的 API 对象都是名词性质的，例如 Service、Volume 这些 API 都是名词，这些名词描述了用户所期望得到的一个目标分布式对象。
2. API 对象是彼此互补而且可组合的。这里面实际是鼓励 API 对象尽量实现面向对象设计时的要求，即“高内聚，松耦合”，对业务相关的概念有一个合适的分解，提高分解出来的对象的可重用性。事实上，Kubernetes 这种分布式系统管理平台，也是一种业务系统，只不过它的业务就是调度和管理容器服务。
3. 高层 API 以操作意图为基础设计。如何能够设计好 API，跟如何能用面向对象的方法设计好应用系统有相通的地方，高层设计一定是从业务出发，而不是过早的从技术实现出发。因此，针对 Kubernetes 的高层 API 设计，一定是以 Kubernetes 的业务为基础出发，也就是以系统调度管理容器的操作意图为基础设计。
4. 低层 API 根据高层 API 的控制需要设计。设计实现低层 API 的目的，是为了被高层 API 使用，考虑减少冗余、提高重用性的目的，低层 API 的设计也要以需求为基础，要尽量抵抗受技术实现影响的诱惑。

5. 尽量避免简单封装，不要有在外部**API**无法显式知道的内部隐藏的机制。简单的封装，实际没有提供新的功能，反而增加了对所封装**API**的依赖性。内部隐藏的机制也是非常不利于系统维护的设计方式，例如**PetSet**和**ReplicaSet**，本来就是两种Pod集合，那么**Kubernetes**就用不同**API**对象来定义它们，而不会说只用同一个**ReplicaSet**，内部通过特殊的算法再来区分这个**ReplicaSet**是有状态的还是无状态。
6. **API**操作复杂度与对象数量成正比。这一条主要是从系统性能角度考虑，要保证整个系统随着系统规模的扩大，性能不会迅速变慢到无法使用，那么最低的限定就是**API**的操作复杂度不能超过 $O(N)$ ， N 是对象的数量，否则系统就不具备水平伸缩性了。
7. **API**对象状态不能依赖于网络连接状态。由于众所周知，在分布式环境下，网络连接断开是经常发生的事情，因此要保证**API**对象状态能应对网络的不稳定，**API**对象的状态就不能依赖于网络连接状态。
8. 尽量避免让操作机制依赖于全局状态，因为在分布式系统中要保证全局状态的同步是非常困难的。

控制机制设计原则

- 控制逻辑应该只依赖于当前状态。这是为了保证分布式的稳定可靠，对于经常出现局部错误的分布式系统，如果控制逻辑只依赖当前状态，那么就非常容易将一个暂时出现故障的系统恢复到正常状态，因为你只要将该系统重置到某个稳定状态，就可以自信的知道系统的所有控制逻辑会开始按照正常方式运行。
- 假设任何错误的可能，并做容错处理。在一个分布式系统中出现局部和临时错误是大概率事件。错误可能来自于物理系统故障，外部系统故障也可能来自于系统自身的代码错误，依靠自己实现的代码不出错来保证系统稳定其实也是难以实现的，因此要设计对任何可能错误的容错处理。
- 尽量避免复杂状态机，控制逻辑不要依赖无法监控的内部状态。因为分布式系统各个子系统都是不能严格通过程序内部保持同步的，所以如果两个子系统的控制逻辑如果互相有影响，那么子系统就一定要能互相访问到影响控制逻辑的状态，否则，就等同于系统里存在不确定的控制逻辑。
- 假设任何操作都可能被任何操作对象拒绝，甚至被错误解析。由于分布式的复杂性以及各子系统的相对独立性，不同子系统经常来自不同的开发团队，所以不能奢望任何操作被另一个子系统以正确的方式处理，要保证出现错误的时候，操作级别的错误不会影响到系统稳定性。
- 每个模块都可以在出错后自动恢复。由于分布式系统中无法保证系统各个模块

是始终连接的，因此每个模块要有自我修复的能力，保证不会因为连接不到其他模块而自我崩溃。

- 每个模块都可以在必要时优雅地降级服务。所谓优雅地降级服务，是对系统鲁棒性的要求，即要求在设计实现模块时划分清楚基本功能和高级功能，保证基本功能不会依赖高级功能，这样同时就保证了不会因为高级功能出现故障而导致整个模块崩溃。根据这种理念实现的系统，也更容易快速地增加新的高级功能，以为不必担心引入高级功能影响原有的基本功能。

Kubernetes的核心技术概念和API对象

API对象是Kubernetes集群中的管理操作单元。Kubernetes集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作。例如副本集Replica Set对应的API对象是RS。

每个API对象都有3大类属性：元数据metadata、规范spec和状态status。元数据是用来标识API对象的，每个对象都至少有3个元数据：namespace，name和uid；除此以外还有各种各样的标签labels用来标识和匹配不同的对象，例如用户可以用标签env来标识区分不同的服务部署环境，分别用env=dev、env=testing、env=production来标识开发、测试、生产的不同服务。规范描述了用户期望Kubernetes集群中的分布式系统达到的理想状态（Desired State），例如用户可以通过复制控制器Replication Controller设置期望的Pod副本数为3；status描述了系统实际当前达到的状态（Status），例如系统当前实际的Pod副本数为2；那么复制控制器当前的程序逻辑就是自动启动新的Pod，争取达到副本数为3。

Kubernetes中所有的配置都是通过API对象的spec去设置的，也就是用户通过配置系统的理想状态来改变系统，这是Kubernetes重要设计理念之一，即所有的操作都是声明式（Declarative）的而不是命令式（Imperative）的。声明式操作在分布式系统中的好处是稳定，不怕丢操作或运行多次，例如设置副本数为3的操作运行多次也还是一个结果，而给副本数加1的操作就不是声明式的，运行多次结果就错了。

Pod

Kubernetes有很多技术概念，同时对应很多API对象，最重要的也是最基础的是Pod。Pod是在Kubernetes集群中运行部署应用或服务的最小单元，它是可以支持多容器的。Pod的设计理念是支持多个容器在一个Pod中共享网络地址和文件系

统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。Pod对多容器的支持是K8最基础的设计理念。比如你运行一个操作系统发行版的软件仓库，一个Nginx容器用来发布软件，另一个容器专门用来从源仓库做同步，这两个容器的镜像不太可能是一个团队开发的，但是他们一块儿工作才能提供一个微服务；这种情况下，不同的团队各自开发构建自己的容器镜像，在部署的时候组合成一个微服务对外提供服务。

Pod是Kubernetes集群中所有业务类型的基础，可以看作运行在K8集群中的小机器人，不同类型的业务就需要不同类型的小机器人去执行。目前Kubernetes中的业务主要可以分为长期伺服型（long-running）、批处理型（batch）、节点后台支撑型（node-daemon）和有状态应用型（stateful application）；分别对应的小机器人控制器为Deployment、Job、DaemonSet和PetSet，本文后面会一一介绍。

副本控制器（**Replication Controller**，RC）

RC是Kubernetes集群中最早的保证Pod高可用的API对象。通过监控运行中的Pod来保证集群中运行指定数目的Pod副本。指定的数目可以是多个也可以是1个；少于指定数目，RC就会启动运行新的Pod副本；多于指定数目，RC就会杀死多余的Pod副本。即使在指定数目为1的情况下，通过RC运行Pod也比直接运行Pod更明智，因为RC也可以发挥它高可用的能力，保证永远有1个Pod在运行。RC是Kubernetes较早期的技术概念，只适用于长期伺服型的业务类型，比如控制小机器人提供高可用的Web服务。

副本集（**Replica Set**，RS）

RS是新一代RC，提供同样的高可用能力，区别主要在于RS后来居上，能支持更多种类的匹配模式。副本集对象一般不单独使用，而是作为Deployment的理想状态参数使用。

部署（**Deployment**）

部署表示用户对Kubernetes集群的一次更新操作。部署是一个比RS应用模式更广的API对象，可以是创建一个新的服务，更新一个新的服务，也可以是滚动升级一个服务。滚动升级一个服务，实际是创建一个新的RS，然后逐渐将新RS中副本数

增加到理想状态，将旧RS中的副本数减小到0的复合操作；这样一个复合操作用一个RS是不太好描述的，所以用一个更通用的Deployment来描述。以Kubernetes的发展方向，未来对所有长期伺服型的业务的管理，都会通过Deployment来管理。

服务（Service）

RC、RS和Deployment只是保证了支撑服务的微服务Pod的数量，但是没有解决如何访问这些服务的问题。一个Pod只是一个运行服务的实例，随时可能在一个节点上停止，在另一个节点以一个新的IP启动一个新的Pod，因此不能以确定的IP和端口号提供服务。要稳定地提供服务需要服务发现和负载均衡能力。服务发现完成的工作，是针对客户端访问的服务，找到对应的后端服务实例。在K8集群中，客户端需要访问的服务就是Service对象。每个Service会对应一个集群内部有效的虚拟IP，集群内部通过虚拟IP访问一个服务。在Kubernetes集群中微服务的负载均衡是由Kube-proxy实现的。Kube-proxy是Kubernetes集群内部的负载均衡器。它是一个分布式代理服务器，在Kubernetes的每个节点上都有一个；这一设计体现了它的伸缩性优势，需要访问服务的节点越多，提供负载均衡能力的Kube-proxy就越多，高可用节点也随之增多。与之相比，我们平时在服务器端做个反向代理做负载均衡，还要进一步解决反向代理的负载均衡和高可用问题。

任务（Job）

Job是Kubernetes用来控制批处理型任务的API对象。批处理业务与长期伺服业务的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出了。成功完成的标志根据不同的spec.completions策略而不同：单Pod型任务有一个Pod成功就标志完成；定数成功型任务保证有N个任务全部成功；工作队列型任务根据应用确认的全局成功而标志成功。

后台支撑服务集（DaemonSet）

长期伺服型和批处理型服务的核心在业务应用，可能有些节点运行多个同类业务的Pod，有些节点上又没有这类Pod运行；而后台支撑型服务的核心关注点在Kubernetes集群中的节点（物理机或虚拟机），要保证每个节点上都有一个此类Pod运行。节点可能是所有集群节点也可能是通过nodeSelector选定的一些特定节点。典型的后台支撑型服务包括，存储，日志和监控等在每个节点上支持Kubernetes集群运行的服务。

有状态服务集（PetSet）

Kubernetes在1.3版本里发布了Alpha版的PetSet功能。在云原生应用的体系里，有下面两组近义词；第一组是无状态（stateless）、牲畜（cattle）、无名（nameless）、可丢弃（disposable）；第二组是有状态（stateful）、宠物（pet）、有名（having name）、不可丢弃（non-disposable）。RC和RS主要是控制提供无状态服务的，其所控制的Pod的名字是随机设置的，一个Pod出故障了就被丢弃掉，在另一个地方重启一个新的Pod，名字变了、名字和启动在哪儿都不重要，重要的只是Pod总数；而PetSet是用来控制有状态服务，PetSet中的每个Pod的名字都是事先确定的，不能更改。PetSet中Pod的名字的作用，并不是《千与千寻》的人性原因，而是关联与该Pod对应的状态。

对于RC和RS中的Pod，一般不挂载存储或者挂载共享存储，保存的是所有Pod共享的状态，Pod像牲畜一样没有分别（这似乎也确实意味着失去了人性特征）；对于PetSet中的Pod，每个Pod挂载自己独立的存储，如果一个Pod出现故障，从其他节点启动一个同样名字的Pod，要挂接上原来Pod的存储继续以它的状态提供服务。

适合于PetSet的业务包括数据库服务MySQL和PostgreSQL，集群化管理服务Zookeeper、etcd等有状态服务。PetSet的另一种典型应用场景是作为一种比普通容器更稳定可靠的模拟虚拟机的机制。传统的虚拟机正是一种有状态的宠物，运维人员需要不断地维护它，容器刚开始流行时，我们用容器来模拟虚拟机使用，所有状态都保存在容器里，而这已被证明是非常不安全、不可靠的。使用PetSet，Pod仍然可以通过漂移到不同节点提供高可用，而存储也可以通过外挂的存储来提供高可靠性，PetSet做的只是将确定的Pod与确定的存储关联起来保证状态的连续性。PetSet还只在Alpha阶段，后面的设计如何演变，我们还要继续观察。

集群联邦（Federation）

Kubernetes在1.3版本里发布了beta版的Federation功能。在云计算环境中，服务的作用距离范围从近到远一般可以有：同主机（Host，Node）、跨主机同可用区（Available Zone）、跨可用区同地区（Region）、跨地区同服务商（Cloud Service Provider）、跨云平台。Kubernetes的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足Kubernetes的调度和计算存储连接要求。而联合集群服务就是为提供跨Region跨服务商Kubernetes集群服务而设计的。

每个Kubernetes Federation有自己的分布式存储、API Server和Controller Manager。用户可以通过Federation的API Server注册该Federation的成员 Kubernetes Cluster。当用户通过Federation的API Server创建、更改API对象时，Federation API Server会在自己所有注册的子Kubernetes Cluster都创建一份对应的API对象。在提供业务请求服务时，Kubernetes Federation会先在自己的各个子Cluster之间做负载均衡，而对于发送到某个具体Kubernetes Cluster的业务请求，会依照这个Kubernetes Cluster独立提供服务时一样的调度模式去做Kubernetes Cluster内部的负载均衡。而Cluster之间的负载均衡是通过域名服务的负载均衡来实现的。

所有的设计都尽量不影响Kubernetes Cluster现有的工作机制，这样对于每个子Kubernetes集群来说，并不需要更外层的有一个Kubernetes Federation，也就是意味着所有现有的Kubernetes代码和机制不需要因为Federation功能有任何变化。

存储卷（Volume）

Kubernetes集群中的存储卷跟Docker的存储卷有些类似，只不过Docker的存储卷作用范围为一个容器，而Kubernetes的存储卷的生命周期和作用范围是一个Pod。每个Pod中声明的存储卷由Pod中的所有容器共享。Kubernetes支持非常多的存储卷类型，特别的，支持多种公有云平台的存储，包括AWS，Google和Azure云；支持多种分布式存储包括GlusterFS和Ceph；也支持较容易使用的主机本地目录hostPath和NFS。Kubernetes还支持使用Persistent Volume Claim即PVC这种逻辑存储，使用这种存储，使得存储的使用者可以忽略后台的实际存储技术（例如AWS，Google或GlusterFS和Ceph），而将有关存储实际技术的配置交给存储管理员通过Persistent Volume来配置。

持久存储卷（Persistent Volume，PV）和持久存储卷声明（Persistent Volume Claim，PVC）

PV和PVC使得Kubernetes集群具备了存储的逻辑抽象能力，使得在配置Pod的逻辑里可以忽略对实际后台存储技术的配置，而把这项配置的工作交给PV的配置者，即集群的管理者。存储的PV和PVC的这种关系，跟计算的Node和Pod的关系是非常类似的；PV和Node是资源的提供者，根据集群的基础设施变化而变化，由Kubernetes集群管理员配置；而PVC和Pod是资源的使用者，根据业务服务的需求变化而变化，有Kubernetes集群的使用者即服务的管理员来配置。

节点（Node）

Kubernetes集群中的计算能力由Node提供，最初Node称为服务节点Minion，后来改名为Node。Kubernetes集群中的Node也就等同于Mesos集群中的Slave节点，是所有Pod运行所在的工作主机，可以是物理机也可以是虚拟机。不论是物理机还是虚拟机，工作主机的统一特征是上面要运行kubelet管理节点上运行的容器。

密钥对象（Secret）

Secret是用来保存和传递密码、密钥、认证凭证这些敏感信息的对象。使用Secret的好处是可以避免把敏感信息明文写在配置文件里。在Kubernetes集群中配置和使用服务不可避免的要用到各种敏感信息实现登录、认证等功能，例如访问AWS存储的用户名密码。为了避免将类似的敏感信息明文写在所有需要使用的配置文件中，可以将这些信息存入一个Secret对象，而在配置文件中通过Secret对象引用这些敏感信息。这种方式的好处包括：意图明确，避免重复，减少暴漏机会。

用户帐户（User Account）和服务帐户（Service Account）

顾名思义，用户帐户为人提供账户标识，而服务帐户为计算机进程和Kubernetes集群中运行的Pod提供账户标识。用户帐户和服务帐户的一个区别是作用范围；用户帐户对应的是人的身份，人的身份与服务的namespace无关，所以用户帐户是跨namespace的；而服务帐户对应的是一个运行中程序的身份，与特定namespace是相关的。

名字空间（Namespace）

名字空间为Kubernetes集群提供虚拟的隔离作用，Kubernetes集群初始有两个名字空间，分别是默认名字空间default和系统名字空间kube-system，除此以外，管理员可以创建新的名字空间满足需要。

RBAC访问授权

Kubernetes在1.3版本中发布了alpha版的基于角色的访问控制（Role-based Access Control，RBAC）的授权模式。相对于基于属性的访问控制（Attribute-based Access Control，ABAC），RBAC主要是引入了角色（Role）和角色绑定

(RoleBinding) 的抽象概念。在ABAC中，Kubernetes集群中的访问策略只能跟用户直接关联；而在RBAC中，访问策略可以跟某个角色关联，具体的用户在跟一个或多个角色相关联。显然，RBAC像其他新功能一样，每次引入新功能，都会引入新的API对象，从而引入新的概念抽象，而这一新的概念抽象一定会使集群服务管理和服务更容易扩展和重用。

总结

从Kubernetes的系统架构、技术概念和设计理念，我们可以看到Kubernetes系统最核心的两个设计理念：一个是容错性，一个是易扩展性。容错性实际是保证Kubernetes系统稳定性和安全性的基础，易扩展性是保证Kubernetes对变更友好，可以快速迭代增加新功能的基础。

按照分布式系统一致性算法Paxos发明人计算机科学家Leslie Lamport的理念，一个分布式系统有两类特性：安全性Safety和活性Liveness。安全性保证系统的稳定，保证系统不会崩溃，不会出现业务错误，不会做坏事，是严格约束的；活性使得系统可以提供功能，提高性能，增加易用性，让系统可以在用户“看到的时间内”做些好事，是尽力而为的。Kubernetes系统的设计理念正好与Lamport安全性与活性的理念不谋而合，也正是因为Kubernetes在引入功能和技术的时候，非常好地划分了安全性和活性，才可以让Kubernetes能有这么快版本迭代，快速引入像RBAC、Federation和PetSet这种新功能。

[1] <http://www.infoq.com/cn/articles/kubernetes-and-cloud-native-applications-part01>

for GitBook update 2017-08-07 13:54:27

主要概念

- Pod
- Node
- Namespace
- Service
- Volume和Persistent Volume
- Deployment
- Secret
- StatefulSet
- DaemonSet
- ServiceAccount
- ReplicationController和ReplicaSet
- Job
- CronJob
- SecurityContext
- Resource Quota
- Pod Security Policy
- Horizontal Pod Autoscaling
- Network Policy
- Ingress
- ThirdPartyResources

for GitBook update 2017-08-07 13:54:27

Pod概览

理解Pod

Pod是kubernetes中你可以创建和部署的最小也是最简单单位。一个Pod代表着集群中运行的一个进程。

Pod中封装着应用的容器（有的情况下是好几个容器），存储、独立的网络IP，管理容器如何运行的策略选项。Pod代表着部署的一个单位：kubernetes中应用的一个实例，可能由一个或者多个容器组合在一起共享资源。

Docker是kubernetes中最常用的容器运行时，但是Pod也支持其他容器运行时。

Pods are employed a number of ways in a Kubernetes cluster, including:

在Kubrenetes集群中Pod有如下两种使用方式：

- 一个Pod中运行一个容器。“每个Pod中一个容器”的模式是最常见的用法；在这种使用方式中，你可以把Pod想象成是单个容器的封装，kuberentes管理的是Pod而不是直接管理容器。
- 在一个Pod中同时运行多个容器。一个Pod中也可以同时封装几个需要紧密耦合互相协作的容器，它们之间共享资源。这些在同一个Pod中的容器可以互相协作成为一个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。

Kubernetes Blog 有关于Pod用例的详细信息：查看：

- [The Distributed System Toolkit: Patterns for Composite Containers](#)
- [Container Design Patterns](#)

每个Pod都是应用的一个实例。如果你想平行扩展应用的话（运行多个实例），你应该运行多个Pod，每个Pod都是一个应用实例。在Kubernetes中，这通常被叫称为是replication。

Pod中如何管理多个容器

2.2.1 Pod

Pod中可以同时运行多个进程（作为容器运行）协同工作。同一个Pod中的容器会自动的分配到同一个 node 上。同一个Pod中的容器共享资源、网络环境和依赖，它们总是被同时调度。

注意在一个Pod中同时运行多个容器是一种比较高级的用法。只有当你的容器需要紧密配合协作的时候才考虑用这种模式。例如，你有一个容器作为web服务器运行，需要用到共享的volume，有另一个“sidecar”容器来从远端获取资源更新这些文件，如下图所示：

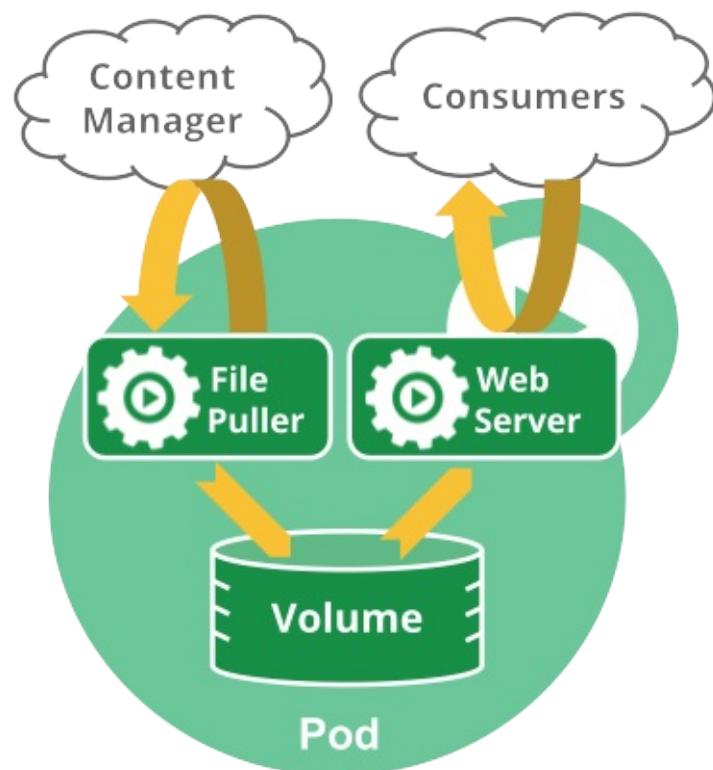


Figure: pod diagram

Pod中可以共享两种资源：网络和存储。

网络

每个Pod都会被分配一个唯一的IP地址。Pod中的所有容器共享网络空间，包括IP地址和端口。Pod内部的容器可以使用 `localhost` 互相通信。Pod中的容器与外界通信时，必须分配共享网络资源（例如使用宿主机的端口映射）。

存储

可以Pod指定多个共享的Volume。Pod中的所有容器都可以访问共享的volume。Volume也可以用来持久化Pod中的存储资源，以防容器重启后文件丢失。

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Kubernetes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 Templates

Pod模版是包含了其他object的Pod定义，例如[Replication Controllers](#)，[Jobs](#)和[DaemonSets](#)。Controller根据Pod模版来创建实际的Pod。

2.2.1 Pod

Pod解析

Pod是kubernetes中可以创建的最小部署单元。

V1 core版本的Pod的配置模板见[Pod template](#)。

什么是Pod？

Pod就像是豌豆荚一样，它由一个或者多个容器组成（例如Docker容器），它们共享容器存储、网络和容器运行配置项。Pod中的容器总是被同时调度，有共同的运行环境。你可以把单个Pod想象成是运行独立应用的“逻辑主机”——其中运行着一个或者多个紧密耦合的应用容器——在有容器之前，这些应用都是运行在几个相同的物理机或者虚拟机上。

尽管kubernetes支持多种容器运行时，但是docker依然是最常用的运行时环境，我们可以使用docker的术语和规则来定义Pod。

Pod中共享的环境包括Linux的namespace，cgroup和其他可能的隔绝环境，这一点跟docker容器一致。在Pod的环境中，每个容器中可能还有更小的子隔离环境。

Pod中的容器共享IP地址和端口号，它们之间可以通过localhost互相发现。它们之间可以通过进程间通信，例如[SystemV](#)信号或者POSIX共享内存。不同Pod之间的容器具有不同的IP地址，不能直接通过IPC通信。

Pod中的容器也有访问共享volume的权限，这些volume会被定义成pod的一部分并挂载到应用容器的文件系统中。

根据docker的结构，Pod中的容器共享namespace和volume，不支持共享PID的namespace。

就像每个应用容器，pod被认为是临时（非持久的）实体。在Pod的生命周期中讨论过，pod被创建后，被分配一个唯一的UID（UID），调度到节点上，并一致维持期望的状态知道被终结（根据重启策略）或者被删除。如果node死掉了，分配到了这个node上的pod，在经过一个超时时间后会被重新调度到其他node节点上。一个给定的pod（如UID定义的）不会被“重新调度”到新的节点上，而是被一个同样的pod取代，如果期望的话甚至可以是相同的名字，但是会有一个新的UID（查看[replication controller](#)获取详情）。（未来，一个更高级别的API将支持pod迁移）。

2.2.1 Pod

Volume跟pod有相同的生命周期（当其UID存在的时候）。当Pod因为某种原因被删除或者被新创建的相同的Pod取代，它相关的东西（例如volume）也会被销毁和再创建一个新的volume。

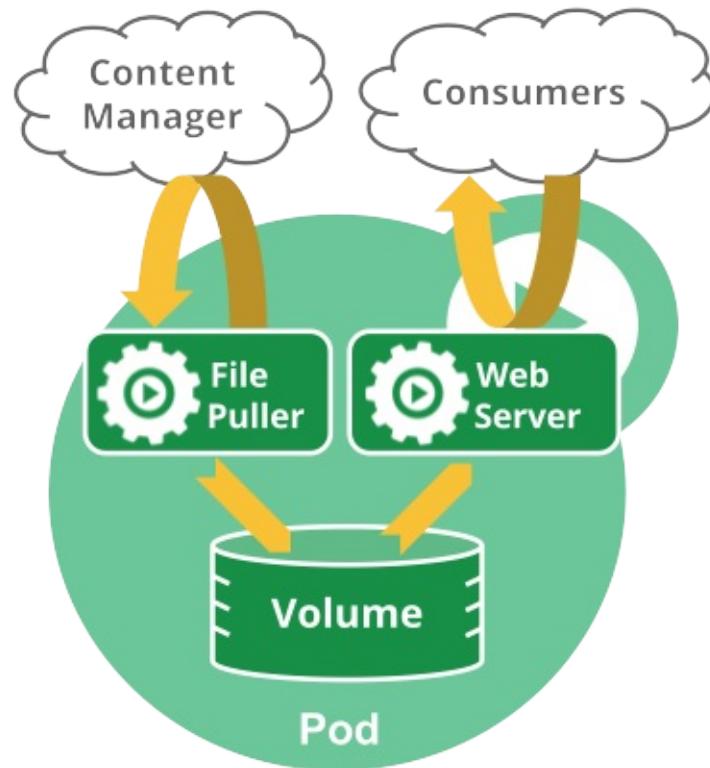


Figure: Pod示意图

A multi-container pod that contains a file puller and a web server that uses a persistent volume for shared storage between the containers.

Pod的动机

管理

Pod是一个服务的多个进程的聚合单位，pod提供这种模型能够简化应用部署管理，通过提供一个更高级别的抽象的方式。Pod作为一个独立的部署单位，支持横向扩展和复制。共生（协同调度），命运共同体（例如被终结），协同复制，资源共享，依赖管理，Pod都会自动的为容器处理这些问题。

资源共享和通信

2.2.1 Pod

Pod中的应用可以共享网络空间（IP地址和端口），因此可以通过 `localhost` 互相发现。因此，pod中的应用必须协调端口占用。每个pod都有一个唯一的IP地址，跟物理机和其他pod都处于一个扁平的网络空间中，它们之间可以直接连通。

Pod中应用容器的`hostname`被被设置成Pod的名字。

Pod中的应用容器可以共享volume。Volume能够保证pod重启时使用的数据不丢失。

Pod的使用

Pod也可以用于垂直应用栈（例如LAMP），这样使用的主要动机是为了支持共同调度和协调管理应用程序，例如：

- content management systems, file and data loaders, local cache managers, etc.
- log and checkpoint backup, compression, rotation, snapshotting, etc.
- data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
- proxies, bridges, and adapters
- controllers, managers, configurators, and updaters

通常单个pod中不会同时运行一个应用的多个实例。

详细说明请看：[The Distributed System ToolKit: Patterns for Composite Containers](#).

其他替代选择

为什么不直接在一个容器中运行多个应用程序呢？

1. 透明。让Pod中的容器对基础设施可见，以便基础设施能够为这些容器提供服务，例如进程管理和资源监控。这可以为用户带来极大的便利。
2. 解耦软件依赖。每个容器都可以进行版本管理，独立的编译和发布。未来kubernetes甚至可能支持单个容器的在线升级。
3. 使用方便。用户不必运行自己的进程管理器，还要担心错误信号传播等。
4. 效率。因为由基础架构提供更多的职责，所以容器可以变得更加轻量级。

为什么不支持容器的亲和性的协同调度？

这种方法可以提供容器的协同定位，能够根据容器的亲和性进行调度，但是无法实现使用pod带来的大部分好处，例如资源共享，IPC，保持状态一致性和简化管理等。

Pod的持久性（或者说缺乏持久性）

Pod在设计支持就不是作为持久化实体的。在调度失败、节点故障、缺少资源或者节点维护的状态下都会死掉会被驱逐。

通常，用户不需要手动直接创建Pod，而是应该使用controller（例如[Deployments](#)），即使时创建单个Pod的情况下。Controller可以提供集群级别的自愈功能、复制和升级管理。

The use of collective APIs as the primary user-facing primitive is relatively common among cluster scheduling systems, including [Borg](#), [Marathon](#), [Aurora](#), and [Tupperware](#).

Pod is exposed as a primitive in order to facilitate:

- scheduler and controller pluggability
- support for pod-level operations without the need to "proxy" them via controller APIs
- decoupling of pod lifetime from controller lifetime, such as for bootstrapping
- decoupling of controllers and services — the endpoint controller just watches pods
- clean composition of Kubelet-level functionality with cluster-level functionality — Kubelet is effectively the "pod controller"
- high-availability applications, which will expect pods to be replaced in advance of their termination and certainly in advance of deletion, such as in the case of planned evictions, image prefetching, or live pod migration [#3949](#)

[StatefulSet](#) controller（目前还是beta状态）支持有状态的Pod。在1.4版本中被称为[PetSet](#)。在kubernetes之前的版本中创建有状态pod的最佳方式是创建一个replica为1的replication controller。

Pod的终止

因为Pod作为在集群的节点上运行的进程，所以在不再需要的时候能够优雅的终止掉是十分必要的（比起使用发送KILL信号这种暴力的方式）。用户需要能够放松删除请求，并且知道它们何时会被终止，是否被正确的删除。用户想终止程序时发送删除pod的请求，在pod可以被强制删除前会有一个优雅删除的时间，会发送一个TERM请求到每个容器的主进程。一旦超时，将向主进程发送KILL信号并从API server中删除。如果kubelet或者container manager在等待进程终止的过程中重启，在重启后仍然会重试完整的优雅删除阶段。

示例流程如下：

1. 用户发送删除pod的命令，默认优雅删除时期是30秒；
2. 在Pod超过该优雅删除期限后API server就会更新Pod的状态为“dead”；
3. 在客户端命令行上显示的Pod状态为“terminating”；
4. 跟第三步同时，当kubelet发现pod被标记为“terminating”状态时，开始停止pod进程：
 - i. 如果在pod中定义了preStop hook，在停止pod前会被调用。如果在优雅删除期限过期后，preStop hook依然在运行，第二步会再增加2秒的优雅时间；
 - ii. 向Pod中的进程发送TERM信号；
5. 跟第三步同时，该Pod将从该service的端点列表中删除，不再是replication controller的一部分。关闭的慢的pod将继续处理load balancer转发的流量；
6. 过了优雅周期后，将向Pod中依然运行的进程发送SIGKILL信号而杀掉进程。
7. Kubectl会在API server中完成Pod的删除，通过将优雅周期设置为0（立即删除）。Pod在API中消失，并且在客户端也不可见。

删除优雅周期默认是30秒。`kubectl delete` 命令支持 `--grace-period=<seconds>` 选项，允许用户设置自己的优雅周期时间。如果设置为0将强制删除pod。在`kubectl >= 1.5`版本的命令中，你必须同时使用 `--force` 和 `--grace-period=0` 来强制删除pod。

强制删除Pod

Pod的强制删除是通过在集群和etcd中将其定义为删除状态。当执行强制删除命令时，API server不会等待该pod所运行在节点上的kubelet确认，就会立即将该pod从API server中移除，这时就可以创建跟原pod同名的pod了。这时，在节点上的pod会被立即设置为terminating状态，不过在被强制删除之前依然有一小段优雅删除周期。

强制删除对于某些pod具有潜在危险性，请谨慎使用。使用StatefulSet pod的情况下，请参考删除StatefulSet中的pod文章。

Pod中容器的特权模式

从kubernetes1.1版本开始，pod中的容器就可以开启privileged模式，在容器定义文件的 `SecurityContext` 下使用 `privileged flag`。这在使用linux的网络操作和访问设备的能力时是很用的。同时开启的特权模式的Pod中的容器也可以访问到容器外的进程和应用。在不需要修改和重新编译kubelet的情况下就可以使用pod来开发节点的网络和存储插件。

如果master节点运行的是kuberentes1.1.或更高版本，而node节点的版本低于1.1版本，则API server将也可以接受新的特权模式的pod，但是无法启动，pod将处于pending状态。

执行 `kubectl describe pod FooPodName`，可以看到为什么pod处于pending状态。输出的event列表中将显示：
`Error validating pod "FooPodName". "FooPodNamespace" from api, ignoring:
spec.containers[0].securityContext.privileged: forbidden '<*>
(0xc2089d3248)true'`

如果master节点的版本低于1.1，无法创建特权模式的pod。如果你仍然试图去创建的话，你得到如下错误：

```
The Pod "FooPodName" is invalid.  
spec.containers[0].securityContext.privileged: forbidden '<*>  
(0xc20b222db0)true'
```

API Object

Pod是kubernetes REST API中的顶级资源类型。

在kuberentes1.6的V1 core API版本中的Pod的数据结构如下图所示：

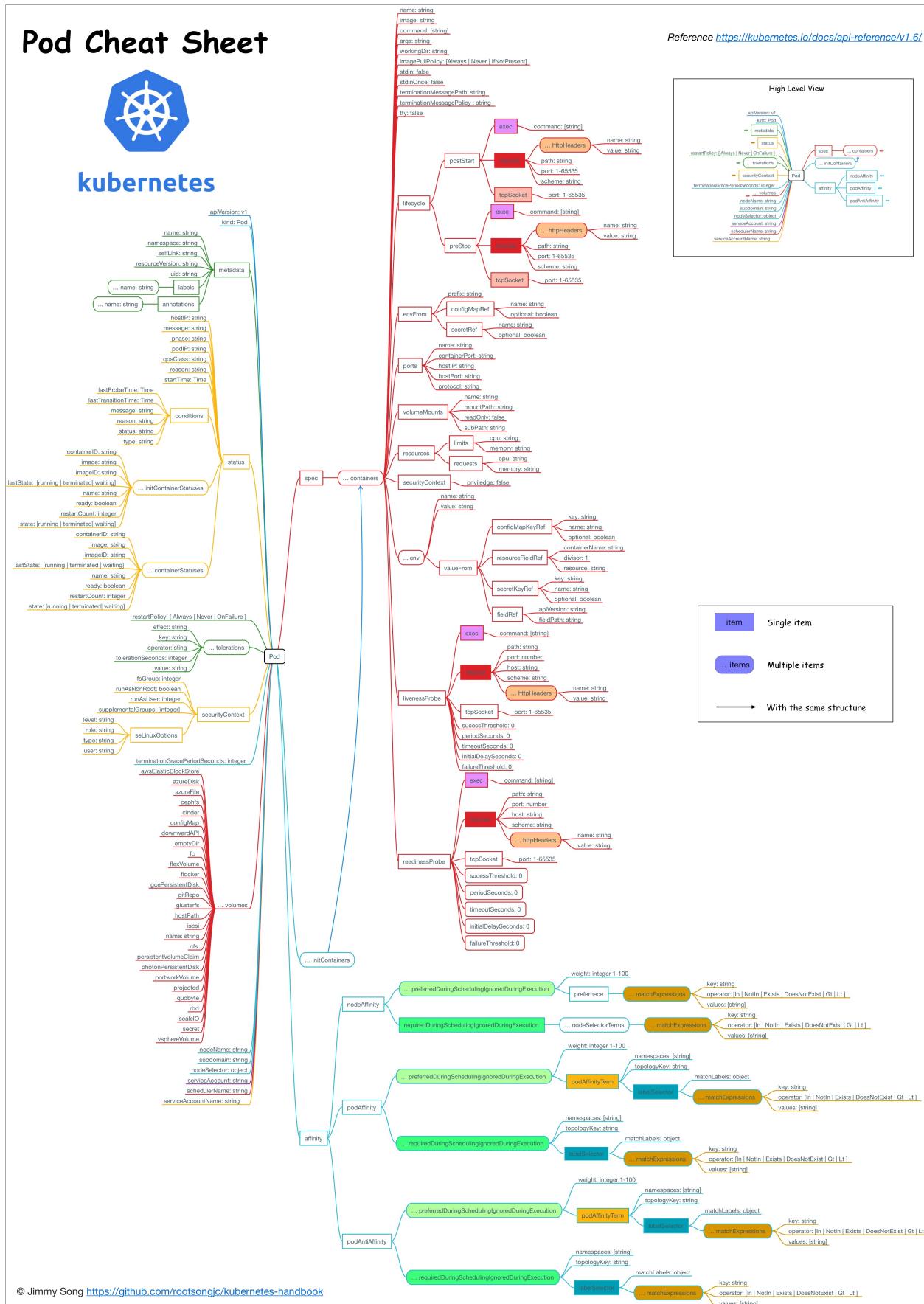


Figure: Pod Cheatsheet

for GitBook update 2017-08-07 13:54:27

Node

Node是kubernetes集群的工作节点，可以是物理机也可以是虚拟机。

Node的状态

Node包括如下状态信息：

- Address
 - HostName：可以被kubelet中的 `-hostname-override` 参数替代。
 - ExternalIP：可以被集群外部路由到的IP地址。
 - InternalIP：集群内部使用的IP，集群外部无法访问。
- Condition
 - OutOfDisk：磁盘空间不足时为 `true`
 - Ready：Node controller 40秒内没有收到node的状态报告为 `Unknown`，健康为 `True`，否则为 `False`。
 - MemoryPressure：当node没有内存压力时为 `True`，否则为 `False`。
 - DiskPressure：当node没有磁盘压力时为 `True`，否则为 `False`。
- Capacity
 - CPU
 - 内存
 - 可运行的最大Pod个数
- Info：节点的一些版本信息，如OS、kubernetes、docker等

Node管理

禁止pod调度到该节点上

```
kubectl cordon <node>
```

驱逐该节点上的所有pod

```
kubectl drain <node>
```

2.2.2 Node

该命令会删除该节点上的所有Pod（DaemonSet除外），在其他node上重新启动它们，通常该节点需要维护时使用该命令。直接使用该命令会自动调用 `kubectl cordon <node>` 命令。当该节点维护完成，启动了kubelet后，再使用 `kubectl uncordon <node>` 即可将该节点添加到kubernetes集群中。

for GitBook update 2017-08-07 13:54:27

Namespace

在一个Kubernetes集群中可以使用namespace创建多个“虚拟集群”，这些namespace之间可以完全隔离，也可以通过某种方式，让一个namespace中的service可以访问到其他的namespace中的服务，我们在[CentOS中部署kubernetes1.6集群](#)的时候就用到了好几个跨越namespace的服务，比如Traefik ingress和 kube-system namespace下的service就可以为整个集群提供服务，这些都需要通过RBAC定义集群级别的角色来实现。

哪些情况下适合使用多个namespace

因为namespace可以提供独立的命名空间，因此可以实现部分的环境隔离。当你的项目和人员众多的时候可以考虑根据项目属性，例如生产、测试、开发划分不同的namespace。

Namespace使用

获取集群中有哪些namespace

```
kubectl get ns
```

集群中默认会有 default 和 kube-system 这两个namespace。

在执行 kubectl 命令时可以使用 -n 指定操作的namespace。

用户的普通应用默认是在 default 下，与集群管理相关的为整个集群提供服务的应用一般部署在 kube-system 的namespace下，例如我们在安装kubernetes集群时部署的 kubedns 、 heapster 、 EFK 等都是在这个namespace下面。

另外，并不是所有的资源对象都会对应 namespace， node 和 persistentVolume 就不属于任何namespace。

2.2.3 Namespace

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，同时还被打上了 "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`

`Service` 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`。

2.2.4 Service

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"`）。

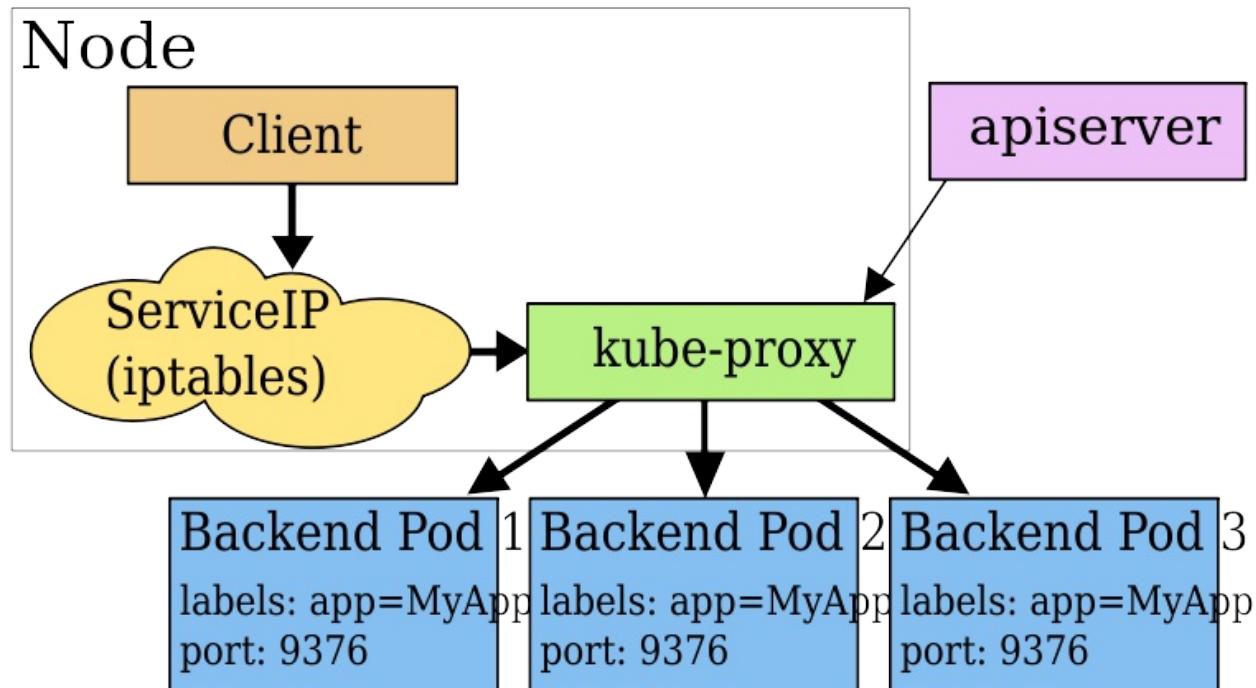


Figure: userspace 代理模式下 Service 概览图

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](#)。

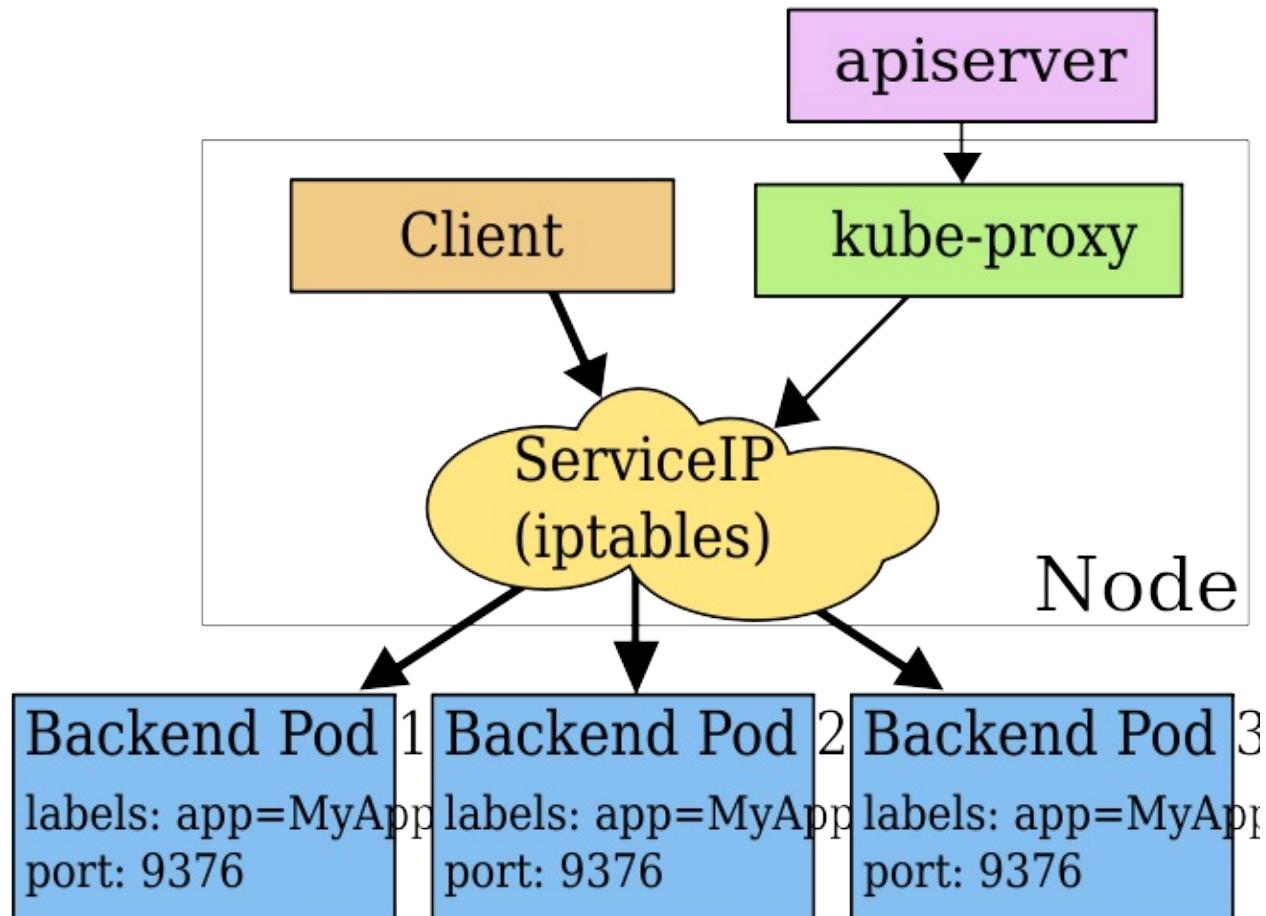


Figure: iptables 代理模式下 Service 概览图

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

2.2.4 Service

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 <NodeIP>:<NodePort>，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均衡器，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）。

2.2.4 Service

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 `<NodeIP>.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

2.2.4 Service

`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: a  
    rn: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。在下面的例子中，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` 会把它指派一个虚拟 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` 会把它指派一个虚拟 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。

API 对象

在 `Kubernetes REST API` 中，`Service` 是 top-level 资源。关于 API 对象的更多细节可以查看：[Service API 对象](#)。

更多信息

2.2.4 Service

阅读 [使用 Service 连接 Frontend 到 Backend](#)。

for GitBook update 2017-08-07 13:54:27

Kubernetes存储卷

我们知道默认情况下容器的数据都是非持久化的，在容器消亡以后数据也跟着丢失，所以Docker提供了Volume机制以便将数据持久化存储。类似的，Kubernetes提供了更强大的Volume机制和丰富的插件，解决了容器数据持久化和容器间共享数据的问题。

Volume

目前，Kubernetes支持以下Volume类型：

- emptyDir
- hostPath
- gcePersistentDisk
- awsElasticBlockStore
- nfs
- iscsi
- flocker
- glusterfs
- rbd
- cephfs
- gitRepo
- secret
- persistentVolumeClaim
- downwardAPI
- azureFileVolume
- vsphereVolume
- flexvolume

注意，这些volume并非全部都是持久化的，比如emptyDir、secret、gitRepo等，这些volume会随着Pod的消亡而消失。

PersistentVolume

对于持久化的Volume，PersistentVolume (PV)和PersistentVolumeClaim (PVC)提供了更方便的管理卷的方法：PV提供网络存储资源，而PVC请求存储资源。这样，设置持久化的工作流包括配置底层文件系统或者云数据卷、创建持久性数据卷、最后创建claim来将pod跟数据卷关联起来。PV和PVC可以将pod和数据卷解耦，pod不需要知道确切的文件系统或者支持它的持久化引擎。

PV

PersistentVolume (PV) 是集群之中的一块网络存储。跟 Node 一样，也是集群的资源。PV 跟 Volume (卷) 类似，不过会有独立于 Pod 的生命周期。比如一个NFS 的PV可以定义为

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /tmp
    server: 172.17.0.2
```

PV的访问模式有三种：

- 第一种，ReadWriteOnce：是最基本的方式，可读可写，但只支持被单个Pod挂载。
- 第二种，ReadOnlyMany：可以以只读的方式被多个Pod挂载。
- 第三种，ReadWriteMany：这种存储可以以读写的方式被多个Pod共享。不是每一种存储都支持这三种方式，像共享方式，目前支持的还比较少，比较常用的是NFS。在PVC绑定PV时通常根据两个条件来绑定，一个是存储的大小，另一个就是访问模式。

StorageClass

2.2.5 Volume和Persistent Volume

上面通过手动的方式创建了一个NFS Volume，这在管理很多Volume的时候不太方便。Kubernetes还提供了[StorageClass](#)来动态创建PV，不仅节省了管理员的时间，还可以封装不同类型的存储供PVC选用。

GCE的例子：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  zone: us-central1-a
```

Ceph RBD的例子：

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/rbd
parameters:
  monitors: 10.16.153.105:6789
  adminId: kube
  adminSecretName: ceph-secret
  adminSecretNamespace: kube-system
  pool: kube
  userId: kube
  userSecretName: ceph-secret-user
```

PVC

PV是存储资源，而PersistentVolumeClaim (PVC) 是对PV的请求。PVC跟Pod类似：Pod消费Node的源，而PVC消费PV资源；Pod能够请求CPU和内存资源，而PVC请求特定大小和访问模式的数据卷。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

PVC可以直接挂载到Pod中：

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
        - mountPath: "/var/www/html"
          name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

emptyDir

如果Pod配置了emptyDir类型Volume，Pod被分配到Node上时候，会创建emptyDir，只要Pod运行在Node上，emptyDir都会存在（容器挂掉不会导致emptyDir丢失数据），但是如果Pod从Node上被删除（Pod被删除，或者Pod发生迁移），emptyDir也会被删除，并且永久丢失。

```
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
```

其他**Volume**说明

hostPath

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

```
- hostPath:
  path: /tmp/data
  name: data
```

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。

```
volumes:  
- name: nfs  
  nfs:  
    # FIXME: use the right hostname  
    server: 10.254.234.223  
    path: "/"
```

FlexVolume

注意要把 volume plugin 放到 `/usr/libexec/kubernetes/kubelet-plugins/volume/exec/<vendor~driver>/<driver>`，plugin 要实现 `init/attach/detach/mount/umount` 等命令（可参考 lvm 的 [示例](#)）。

```
- name: test  
  flexVolume:  
    driver: "kubernetes.io/lvm"  
    fsType: "ext4"  
    options:  
      volumeID: "vol1"  
      size: "1000m"  
      volumegroup: "kube_vg"
```

Deployment

[TOC]

简述

Deployment 为 Pod 和 ReplicaSet 提供了一个声明式定义(declarative)方法，用来替代以前的ReplicationController 来方便的管理应用。典型的应用场景包括：

- 定义Deployment来创建Pod和ReplicaSet
- 滚动升级和回滚应用
- 扩容和缩容
- 暂停和继续Deployment

比如一个简单的nginx应用可以定义为

```
apiVersion: extensions/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
```

扩容：

```
kubectl scale deployment nginx-deployment --replicas 10
```

2.2.6 Deployment

如果集群支持 horizontal pod autoscaling 的话，还可以为Deployment设置自动扩展：

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

更新镜像也比较简单：

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
```

回滚：

```
kubectl rollout undo deployment/nginx-deployment
```

Deployment 结构示意图

参考：<https://kubernetes.io/docs/api-reference/v1.6/#deploymentspec-v1beta1-apps>

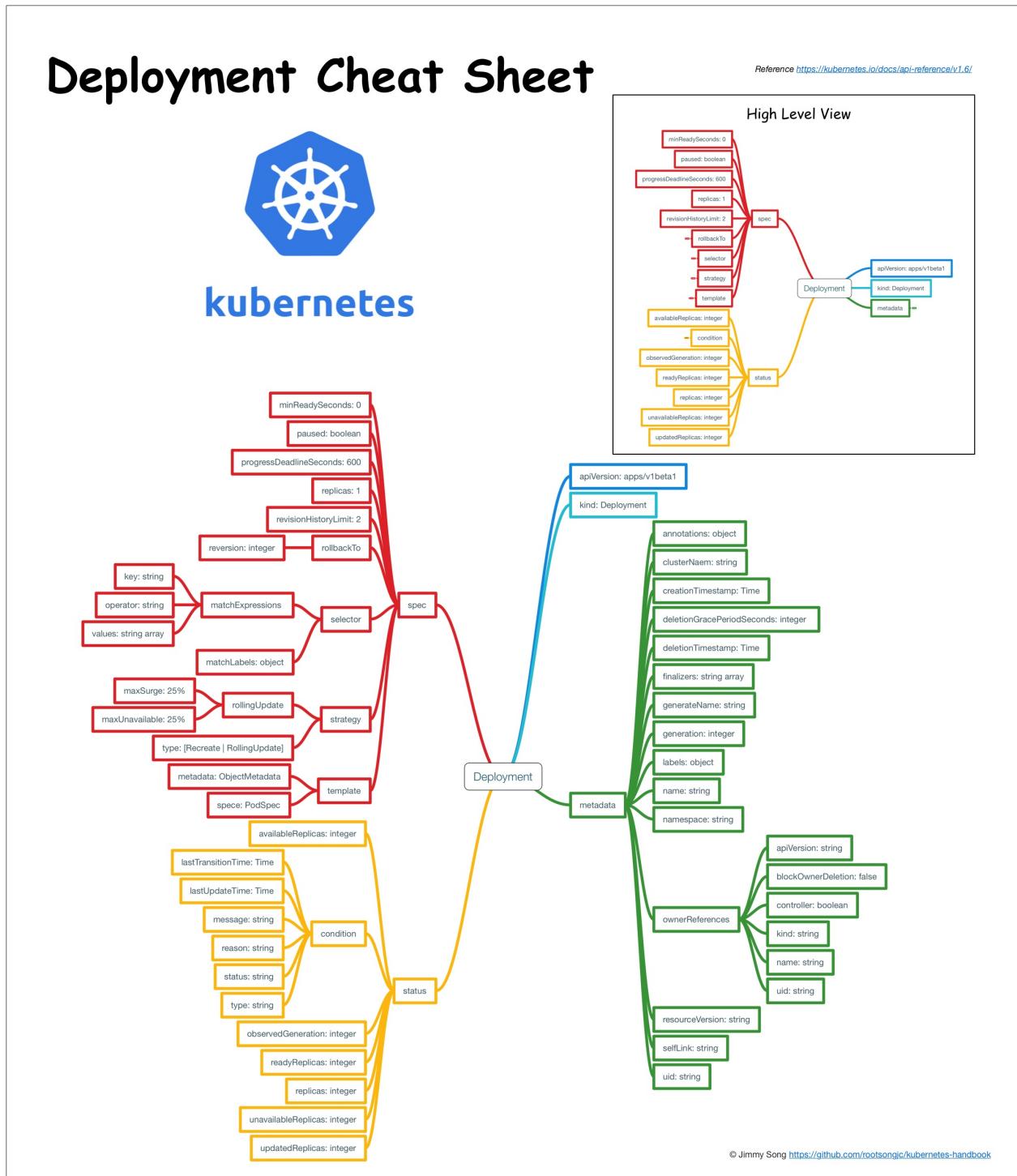


Figure: kubernetes deployment cheatsheet

Deployment 概念详细解析

本文翻译自 kubernetes 官方文

档：<https://kubernetes.io/docs/concepts/workloads/controllers/deployment.md>

根据2017年5月10日的Commit 8481c02 翻译。

Deployment 是什么？

Deployment为Pod和Replica Set（下一代Replication Controller）提供声明式更新。

您只需要在 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 ReplicaSet，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：

2.2.6 Deployment

```
$ 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replica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-kzszej   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。

2.2.6 Deployment

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 is 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 **Kubernetes** 版本中，将从 **1-1** 变成 **25%-25%**。

例如，如果您自己看上面的 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。

2.2.6 Deployment

```
$ 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                    Subobject
  ctPath     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 nginx: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，只有 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...
```

2.2.6 Deployment

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-jbqko 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
```

2.2.6 Deployment

```
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     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 `--record` 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

2.2.6 Deployment

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 revision 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             Subobject
```

2.2.6 Deployment

tPath	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 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.revisionHistoryLimit` 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=1  
5 --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。

2.2.6 Deployment

```
$ 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replica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RepliciaSet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

2.2.6 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 Deployment 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：

2.2.6 Deployment

```
$ 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，也就是说更新完成。
- 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：

2.2.6 Deployment

- 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 kubernetes 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：

2.2.6 Deployment

```
$ 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` , Deployment 的状态可能看起来像这个样子：

2.2.6 Deployment

```
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，kubernetes 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

2.2.6 Deployment

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后就会被认为是有可用状态）。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revision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。

for GitBook update 2017-08-07 13:54:27

Secret

Secret解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中。Secret可以以Volume或者环境变量的方式使用。

Secret有三种类型：

- **Service Account**：用来访问Kubernetes API，由Kubernetes自动创建，并且会自动挂载到Pod的 `/run/secrets/kubernetes.io/serviceaccount` 目录中；
- **Opaque**：base64编码格式的Secret，用来存储密码、密钥等；
- **kubernetes.io/dockerconfigjson**：用来存储私有docker registry的认证信息。

Opaque Secret

Opaque类型的数据是一个map类型，要求value是base64编码格式：

```
$ echo -n "admin" | base64
YWRTaW4=
$ echo -n "1f2d1e2e67df" | base64
MwYyZDFlMmU2N2Rm
```

secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  password: MwYyZDFlMmU2N2Rm
  username: YWRTaW4=
```

接着，就可以创建secret了：`kubectl create -f secrets.yml`。

创建好**secret**之后，有两种方式来使用它：

- 以**Volume**方式
- 以环境变量方式

将**Secret**挂载到**Volume**中

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    name: db
    name: db
spec:
  volumes:
    - name: secrets
      secret:
        secretName: mysecret
  containers:
    - image: gcr.io/my_project_id/pg:v1
      name: db
      volumeMounts:
        - name: secrets
          mountPath: "/etc/secrets"
          readOnly: true
  ports:
    - name: cp
      containerPort: 5432
      hostPort: 5432
```

将**Secret**导出到环境变量中

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wordpress-deployment
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: wordpress
        visualize: "true"
    spec:
      containers:
        - name: "wordpress"
          image: "wordpress"
          ports:
            - containerPort: 80
      env:
        - name: WORDPRESS_DB_USER
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: password
```

kubernetes.io/dockerconfigjson

可以直接用 `kubectl` 命令来创建用于docker registry认证的secret：

2.2.7 Secret

```
$ kubectl create secret docker-registry myregistrykey --docker-server=DOCKER_REGISTRY_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL  
secret "myregistrykey" created.
```

也可以直接读取 `~/.docker/config.json` 的内容来创建：

在创建Pod的时候，通过 `imagePullSecrets` 来引用刚创建的 `myregistrykey`：

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
    - name: foo
      image: janedoe/awesomeapp:v1
  imagePullSecrets:
    - name: myregistrykey
```

Service Account

2.2.7 Secret

Service Account用来访问Kubernetes API，由Kubernetes自动创建，并且会自动挂载到Pod的 /run/secrets/kubernetes.io/serviceaccount 目录中。

```
$ kubectl run nginx --image nginx
deployment "nginx" created
$ kubectl get pods
NAME                  READY     STATUS    RESTARTS   AGE
nginx-3137573019-md1u2   1/1      Running   0          13s
$ kubectl exec nginx-3137573019-md1u2 ls /run/secrets/kubernetes
.io/serviceaccount
ca.crt
namespace
token
```

for GitBook update 2017-08-07 13:54:27

StatefulSet

StatefulSet 作为 Controller 为 Pod 提供唯一的标识。它可以保证部署和 scale 的顺序。

StatefulSet 是为了解决有状态服务的问题（对应 Deployments 和 ReplicaSets 是为无状态服务而设计），其应用场景包括：

- 稳定的持久化存储，即 Pod 重新调度后还是能访问到相同的持久化数据，基于 PVC 来实现
- 稳定的网络标志，即 Pod 重新调度后其 PodName 和 HostName 不变，基于 Headless Service（即没有 Cluster IP 的 Service）来实现
- 有序部署，有序扩展，即 Pod 是有序的，在部署或者扩展的时候要依据定义的顺序依次依次进行（即从 0 到 N-1，在下一个 Pod 运行之前所有之前的 Pod 必须都是 Running 和 Ready 状态），基于 init containers 来实现
- 有序收缩，有序删除（即从 N-1 到 0）

从上面的应用场景可以发现，StatefulSet 由以下几个部分组成：

- 用于定义网络标志（DNS domain）的 Headless Service
- 用于创建 PersistentVolumes 的 volumeClaimTemplates
- 定义具体应用的 StatefulSet

StatefulSet 中每个 Pod 的 DNS 格式为 statefulSetName-{0..N-1}.serviceName.namespace.svc.cluster.local，其中

- serviceName 为 Headless Service 的名字
- 0..N-1 为 Pod 所在的序号，从 0 开始到 N-1
- statefulSetName 为 StatefulSet 的名字
- namespace 为服务所在的 namespace，Headless Service 和 StatefulSet 必须在相同的 namespace
- .cluster.local 为 Cluster Domain

使用 StatefulSet

StatefulSet 适用于有以下某个或多个需求的应用：

- 稳定，唯一的网络标志。
- 稳定，持久化存储。
- 有序，优雅地部署和 scale。
- 有序，优雅地删除和终止。
- 有序，自动的滚动升级。

在上文中，稳定是 Pod（重新）调度中持久性的代名词。如果应用程序不需要任何稳定的标识符、有序部署、删除和 scale，则应该使用提供一组无状态副本的 controller 来部署应用程序，例如 Deployment 或 ReplicaSet 可能更适合您的无状态需求。

限制

- StatefulSet 是 beta 资源，Kubernetes 1.5 以前版本不支持。
- 对于所有的 alpha/beta 的资源，您都可以通过在 apiserver 中设置 `--runtime-config` 选项来禁用。
- 给定 Pod 的存储必须由 PersistentVolume Provisioner 根据请求的 storage class 进行配置，或由管理员预先配置。
- 删除或 scale StatefulSet 将不会删除与 StatefulSet 相关联的 volume。这样做是为了确保数据安全性，这通常比自动清除所有相关 StatefulSet 资源更有价值。
- StatefulSets 目前要求 Headless Service 负责 Pod 的网络身份。您有责任创建此服务。

组件

下面的示例中描述了 StatefulSet 中的组件。

- 一个名为 nginx 的 headless service，用于控制网络域。
- 一个名为 web 的 StatefulSet，它的 Spec 中指定在有 3 个运行 nginx 容器的 Pod。
- volumeClaimTemplates 使用 PersistentVolume Provisioner 提供的 PersistentVolumes 作为稳定存储。

```
apiVersion: v1
kind: Service
```

2.2.8 StatefulSet

```
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
      ports:
      - containerPort: 80
        name: web
      volumeMounts:
      - name: www
        mountPath: /usr/share/nginx/html
volumeClaimTemplates:
- metadata:
    name: www
    annotations:
      volume.beta.kubernetes.io/storage-class: anything
spec:
```

```

accessModes: [ "ReadWriteOnce" ]
resources:
  requests:
    storage: 1Gi

```

Pod 身份

StatefulSet Pod 具有唯一的身份，包括序数，稳定的网络身份和稳定的存储。身份绑定到 Pod 上，不管它（重新）调度到哪个节点上。

序数

对于一个有 N 个副本的 StatefulSet，每个副本都会被指定一个整数序数，在 [0,N) 之间，且唯一。

稳定的网络 ID

StatefulSet 中的每个 Pod 从 StatefulSet 的名称和 Pod 的序数派生其主机名。构造的主机名的模式是 \$(statefulset名称)-\$(序数)。上面的例子将创建三个名为 web-0, web-1, web-2 的 Pod。

StatefulSet 可以使用 [Headless Service](#) 来控制其 Pod 的域。此服务管理的域的格式为：\$(服务名称).\$(namespace).svc.cluster.local，其中“cluster.local”是集群域。

在创建每个Pod时，它将获取一个匹配的 DNS 子域，采用以下形式：\$(pod 名称).\$(管理服务域)，其中管理服务由 StatefulSet 上的 serviceName 字段定义。

以下是 Cluster Domain，服务名称，StatefulSet 名称以及如何影响 StatefulSet 的 Pod 的 DNS 名称的一些示例。

Cluster Domain	Service (ns/name)	StatefulSet (ns/name)	StatefulSet Domain	
cluster.local	default/nginx	default/web	nginx.default.svc.cluster.local	W 1
cluster.local	foo/nginx	foo/web	nginx.foo.svc.cluster.local	W 1
kube.local	foo/nginx	foo/web	nginx.foo.svc.kube.local	W 1

注意 Cluster Domain 将被设置成 `cluster.local` 除非进行了 [其他配置](#)。

稳定存储

Kubernetes 为每个 VolumeClaimTemplate 创建一个 [PersistentVolume](#)。上面的 nginx 的例子中，每个 Pod 将具有一个由 `anything` 存储类创建的 1 GB 存储的 PersistentVolume。当该 Pod（重新）调度到节点上，`volumeMounts` 将挂载与 PersistentVolume Claim 相关联的 PersistentVolume。请注意，与 PersistentVolume Claim 相关联的 PersistentVolume 在产出 Pod 或 StatefulSet 的时候不会被删除。这必须手动完成。

部署和 **Scale** 保证

- 对于有 N 个副本的 StatefulSet，Pod 将按照 {0..N-1} 的顺序被创建和部署。
- 当删除 Pod 的时候，将按照逆序来终结，从 {N-1..0}
- 对 Pod 执行 scale 操作之前，它所有的前任必须处于 Running 和 Ready 状态。
- 在终止 Pod 前，它所有的继任者必须处于完全关闭状态。

不应该将 StatefulSet 的 `pod.Spec.TerminationGracePeriodSeconds` 设置为 0。这样是不安全的且强烈不建议您这样做。进一步解释，请参阅 [强制删除 StatefulSet Pod](#)。

上面的 nginx 示例创建后，3 个 Pod 将按照如下顺序创建 web-0，web-1，web-2。在 web-0 处于 [运行并就绪](#) 状态之前，web-1 将不会被部署，同样当 web-1 处于运行并就绪状态之前 web-2 也不会被部署。如果在 web-1 运行并就绪后，web-2 启动之前，web-0 失败了，web-2 将不会启动，直到 web-0 成果重启并处于运行并就绪状态。

如果用户通过修补 StatefulSet 来 scale 部署的示例，以使 `replicas=1`，则 `web-2` 将首先被终止。在 `web-2` 完全关闭和删除之前，`web-1` 不会被终止。如果 `web-0` 在 `web-2` 终止并且完全关闭之后，但是在 `web-1` 终止之前失败，则 `web-1` 将不会终止，除非 `web-0` 正在运行并准备就绪。

Pod 管理策略

在 Kubernetes 1.7 和之后版本，StatefulSet 允许您放开顺序保证，同时通过 `.spec.podManagementPolicy` 字段保证身份的唯一性。

OrderedReady Pod 管理

StatefulSet 中默认使用的是 `OrderedReady` pod 管理。它实现了 [如上](#) 所述的行为。

并行 Pod 管理

`Parallel` pod 管理告诉 StatefulSet controller 并行的启动和终止 Pod，在启动和终止其他 Pod 之前不会等待 Pod 变成 运行并就绪或完全终止状态。

更新策略

在 kubernetes 1.7 和以上版本中，StatefulSet 的 `.spec.updateStrategy` 字段允许您配置和禁用 StatefulSet 中的容器、label、resource request/limit、annotation 的滚动更新。

删除

`OnDelete` 更新策略实现了遗留（1.6和以前）的行为。当 `spec.updateStrategy` 未指定时，这是默认策略。当 StatefulSet 的 `.spec.updateStrategy.type` 设置为 `OnDelete` 时，StatefulSet 控制器将不会自动更新 StatefulSet 中的 Pod。用户必须手动删除 Pod 以使控制器创建新的 Pod，以反映对 StatefulSet 的 `.spec.template` 进行的修改。

滚动更新

`RollingUpdate` 更新策略在 StatefulSet 中实现 Pod 的自动滚动更新。当 StatefulSet 的 `.spec.updateStrategy.type` 设置为 `RollingUpdate` 时，StatefulSet 控制器将在 StatefulSet 中删除并重新创建每个 Pod。它将以与 Pod 终止相同的顺序进行（从最大的序数到最小的序数），每次更新一个 Pod。在更新其前身之前，它将等待正在更新的 Pod 状态变成正在运行并就绪。

分区

可以通过指定 `.spec.updateStrategy.rollingUpdate.partition` 来对 RollingUpdate 更新策略进行分区。如果指定了分区，则当 StatefulSet 的 `.spec.template` 更新时，具有大于或等于分区序数的所有 Pod 将被更新。具有小于分区的序数的所有 Pod 将不会被更新，即使删除它们也将被重新创建。如果 StatefulSet 的 `.spec.updateStrategy.rollingUpdate.partition` 大于其 `.spec.replicas`，则其 `.spec.template` 的更新将不会传播到 Pod。

在大多数情况下，您不需要使用分区，但如果想要进行分阶段更新，使用金丝雀发布或执行分阶段发布，它们将非常有用。

简单示例

以一个简单的nginx服务[web.yaml](#)为例：

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta1
```

2.2.8 StatefulSet

```
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: gcr.io/google_containers/nginx-slim:0.8
          ports:
            - containerPort: 80
              name: web
      volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
    - metadata:
        name: www
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

```
$ kubectl create -f web.yaml
service "nginx" created
statefulset "web" created

# 查看创建的headless service和statefulset
$ kubectl get service nginx
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
nginx     None          <none>       80/TCP      1m
```

2.2.8 StatefulSet

```
$ kubectl get statefulset web
NAME      DESIRED   CURRENT   AGE
web       2          2          2m

# 根据volumeClaimTemplates自动创建PVC（在GCE中会自动创建kubernetes.io/gce-pd类型的volume）
$ kubectl get pvc
NAME      STATUS    VOLUME
CAPACITY  ACCESSMODES   AGE
www-web-0  Bound     pvc-d064a004-d8d4-11e6-b521-42010a800002
  1Gi      RWO        16s
www-web-1  Bound     pvc-d06a3946-d8d4-11e6-b521-42010a800002
  1Gi      RWO        16s

# 查看创建的Pod，他们都是有序的
$ kubectl get pods -l app=nginx
NAME      READY   STATUS    RESTARTS   AGE
web-0     1/1     Running   0          5m
web-1     1/1     Running   0          4m

# 使用nslookup查看这些Pod的DNS
$ kubectl run -i --tty --image busybox dns-test --restart=Never
--rm /bin/sh
/ # nslookup web-0.nginx
Server:  10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.2.10
/ # nslookup web-1.nginx
Server:  10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.3.12
/ # nslookup web-0.nginx.default.svc.cluster.local
Server:  10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx.default.svc.cluster.local
```

2.2.8 StatefulSet

```
Address 1: 10.244.2.10
```

还可以进行其他的操作

```
# 扩容
$ kubectl scale statefulset web --replicas=5

# 缩容
$ kubectl patch statefulset web -p '{"spec":{"replicas":3}}'

# 镜像更新（目前还不支持直接更新image，需要patch来间接实现）
$ kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/spec/template/spec/containers/0/image", "value": "gcr.io/google_containers/nginx-slim:0.7"}]'

# 删除StatefulSet和Headless Service
$ kubectl delete statefulset web
$ kubectl delete service nginx

# StatefulSet删除后PVC还会保留着，数据不再使用的话也需要删除
$ kubectl delete pvc www-web-0 www-web-1
```

zookeeper

另外一个更能说明StatefulSet强大功能的示例为[zookeeper.yaml](#)。

```
---
apiVersion: v1
kind: Service
metadata:
  name: zk-headless
  labels:
    app: zk-headless
spec:
  ports:
    - port: 2888
      name: server
    - port: 3888
```

2.2.8 StatefulSet

```
    name: leader-election
  clusterIP: None
  selector:
    app: zk
  ---
apiVersion: v1
kind: ConfigMap
metadata:
  name: zk-config
data:
  ensemble: "zk-0;zk-1;zk-2"
  jvm.heap: "2G"
  tick: "2000"
  init: "10"
  sync: "5"
  client.cnxns: "60"
  snap.retain: "3"
  purge.interval: "1"
  ---
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-budget
spec:
  selector:
    matchLabels:
      app: zk
  minAvailable: 2
  ---
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: zk
spec:
  serviceName: zk-headless
  replicas: 3
  template:
    metadata:
      labels:
        app: zk
```

2.2.8 StatefulSet

```
annotations:
  pod.alpha.kubernetes.io/initialized: "true"
  scheduler.alpha.kubernetes.io/affinity: >
    {
      "podAntiAffinity": {
        "requiredDuringSchedulingRequiredDuringExecution
      }: [{{
        "labelSelector": {
          "matchExpressions": [{{
            "key": "app",
            "operator": "In",
            "values": ["zk-headless"]
          }]}
        },
        "topologyKey": "kubernetes.io/hostname"
      }]
    }
  }

spec:
  containers:
    - name: k8szk
      imagePullPolicy: Always
      image: gcr.io/google_samples/k8szk:v1
    resources:
      requests:
        memory: "4Gi"
        cpu: "1"
    ports:
      - containerPort: 2181
        name: client
      - containerPort: 2888
        name: server
      - containerPort: 3888
        name: leader-election
    env:
      - name : ZK_ENSEMBLE
        valueFrom:
          configMapKeyRef:
            name: zk-config
            key: ensemble
```

2.2.8 StatefulSet

```
- name : ZK_HEAP_SIZE
  valueFrom:
    configMapKeyRef:
      name: zk-config
      key: jvm.heap
- name : ZK_TICK_TIME
  valueFrom:
    configMapKeyRef:
      name: zk-config
      key: tick
- name : ZK_INIT_LIMIT
  valueFrom:
    configMapKeyRef:
      name: zk-config
      key: init
- name : ZK_SYNC_LIMIT
  valueFrom:
    configMapKeyRef:
      name: zk-config
      key: tick
- name : ZK_MAX_CLIENT_CNXNS
  valueFrom:
    configMapKeyRef:
      name: zk-config
      key: client.cnxns
- name: ZK_SNAP_RETAIN_COUNT
  valueFrom:
    configMapKeyRef:
      name: zk-config
      key: snap.retain
- name: ZK_PURGE_INTERVAL
  valueFrom:
    configMapKeyRef:
      name: zk-config
      key: purge.interval
- name: ZK_CLIENT_PORT
  value: "2181"
- name: ZK_SERVER_PORT
  value: "2888"
- name: ZK_ELECTION_PORT
```

2.2.8 StatefulSet

```
        value: "3888"
    command:
      - sh
      - -c
      - zkGenConfig.sh && zkServer.sh start-foreground
  readinessProbe:
    exec:
      command:
        - "zkOk.sh"
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    exec:
      command:
        - "zkOk.sh"
    initialDelaySeconds: 15
    timeoutSeconds: 5
  volumeMounts:
    - name: datadir
      mountPath: /var/lib/zookeeper
  securityContext:
    runAsUser: 1000
    fsGroup: 1000
volumeClaimTemplates:
- metadata:
    name: datadir
    annotations:
      volume.alpha.kubernetes.io/storage-class: anything
spec:
  accessModes: [ "ReadWriteOnce" ]
  resources:
    requests:
      storage: 20Gi
```

```
kubectl create -f zookeeper.yaml
```

详细的使用说明见[zookeeper stateful application](#)。

参考

<https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>

for GitBook update 2017-08-07 13:54:27

DaemonSet

DaemonSet保证在每个Node上都运行一个容器副本，常用来部署一些集群的日志、监控或者其他系统管理程序。典型的应用常见包括：

- 日志收集，比如fluentd，logstash等
- 系统监控，比如Prometheus Node Exporter，collectd，New Relic agent，Ganglia gmond等
- 系统程序，比如kube-proxy, kube-dns, glusterd, ceph等

使用Fluentd收集日志的例子：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  template:
    metadata:
      labels:
        app: logging
        id: fluentd
        name: fluentd
    spec:
      containers:
        - name: fluentd-es
          image: gcr.io/google_containers/fluentd-elasticsearch:1.3

      env:
        - name: FLUENTD_ARGS
          value: -qq
      volumeMounts:
        - name: containers
          mountPath: /var/lib/docker/containers
        - name: varlog
          mountPath: /var/log
      volumes:
        - hostPath:
            path: /var/lib/docker/containers
            name: containers
        - hostPath:
            path: /var/log
            name: varlog
```



指定Node节点

DaemonSet会忽略Node的unschedulable状态，有两种方式来指定Pod只运行在指定的Node节点上：

- nodeSelector：只调度到匹配指定label的Node上

2.2.9 DaemonSet

- nodeAffinity：功能更丰富的Node选择器，比如支持集合操作
- podAffinity：调度到满足条件的Pod所在的Node上

nodeSelector示例：

```
spec:  
  nodeSelector:  
    disktype: ssd
```

nodeAffinity示例：

```
metadata:  
  name: with-node-affinity  
  annotations:  
    scheduler.alpha.kubernetes.io/affinity: >  
    {  
      "nodeAffinity": {  
        "requiredDuringSchedulingIgnoredDuringExecution": {  
          "nodeSelectorTerms": [  
            {  
              "matchExpressions": [  
                {  
                  "key": "kubernetes.io/e2e-az-name",  
                  "operator": "In",  
                  "values": ["e2e-az1", "e2e-az2"]  
                }  
              ]  
            }  
          ]  
        }  
      }  
    }  
  another-annotation-key: another-annotation-value
```

podAffinity示例：

```
metadata:  
  name: with-pod-affinity
```

2.2.9 DaemonSet

```
annotations:
  scheduler.alpha.kubernetes.io/affinity: >
    {
      "podAffinity": {
        "requiredDuringSchedulingIgnoredDuringExecution": [
          {
            "labelSelector": {
              "matchExpressions": [
                {
                  "key": "security",
                  "operator": "In",
                  "values": ["S1"]
                }
              ]
            },
            "topologyKey": "failure-domain.beta.kubernetes.io/zone"
          }
        ]
      },
      "podAntiAffinity": {
        "requiredDuringSchedulingIgnoredDuringExecution": [
          {
            "labelSelector": {
              "matchExpressions": [
                {
                  "key": "security",
                  "operator": "In",
                  "values": ["S2"]
                }
              ]
            },
            "topologyKey": "kubernetes.io/hostname"
          }
        ]
      }
    }
spec:
  ...

```

静态Pod

除了DaemonSet，还可以使用静态Pod来在每台机器上运行指定的Pod，这需要kubelet在启动的时候指定manifest目录：

```
kubelet --pod-manifest-path=<the directory>
```

然后将所需要的Pod定义文件放到指定的manifest目录中即可。

注意：静态Pod不能通过API Server来删除，但可以通过删除manifest文件来自动删除对应的Pod。

for GitBook update 2017-08-07 13:54:27

Service Account

Service account是为了方便Pod里面的进程调用Kubernetes API或其他外部服务，它不同于User account：

- User account是为人设计的，而service account则是为了Pod中的进程；
 - User account是跨namespace的，而service account则是仅局限它所在的namespace；
 - 开启ServiceAccount（默认开启）后，每个namespace都会自动创建一个Service account，并会相应的secret挂载到每一个Pod中
 - 默认ServiceAccount为default，自动关联一个访问kubernetes API的Secret
 - 每个Pod在创建后都会自动设置 spec.serviceAccount 为 default（除非指定了其他ServiceAccout）
 - 每个container启动后都会挂载对应的token 和 ca.crt 到 /var/run/secrets/kubernetes.io/serviceaccount/
- 当然了，也可以创建更多的Service Account：

```
$ cat > /tmp/serviceaccount.yaml <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
  namespace: default
EOF
$ kubectl create -f /tmp/serviceaccount.yaml
serviceaccounts/build-robot
```

Service Account为服务提供了一种方便的认知机制，但它不关心授权的问题。可以配合[RBAC](#)来为Service Account鉴权：

- 配置 --authorization-mode=RBAC 和 --runtime-config=rbac.authorization.k8s.io/v1alpha1
- 配置 --authorization-rbac-super-user=admin
- 定义Role、ClusterRole、RoleBinding或ClusterRoleBinding

2.2.10 ServiceAccount

比如

```
# This role allows to read pods in the namespace "default"
kind: Role
apiVersion: rbac.authorization.k8s.io/v1alpha1
metadata:
  namespace: default
  name: pod-reader
rules:
  - apiGroups: [""]
    # The API group "" indicates the core API Group.
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
    nonResourceURLs: []
---
# This role binding allows "default" to read pods in the namespace "default"
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1alpha1
metadata:
  name: read-pods
  namespace: default
subjects:
  - kind: ServiceAccount
    # May be "User", "Group" or "ServiceAccount"
    name: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

for GitBook update 2017-08-07 13:54:27

ReplicationController和ReplicaSet

ReplicationController用来确保容器应用的副本数始终保持在用户定义的副本数，即如果有容器异常退出，会自动创建新的Pod来替代；而如果异常多出来的容器也会自动回收。

在新版本的Kubernetes中建议使用ReplicaSet来取代ReplicationController。

ReplicaSet跟ReplicationController没有本质的不同，只是名字不一样，并且ReplicaSet支持集合式的selector。

虽然ReplicaSet可以独立使用，但一般还是建议使用Deployment来自动管理ReplicaSet，这样就无需担心跟其他机制的不兼容问题（比如ReplicaSet不支持rolling-update但Deployment支持）。

ReplicaSet示例：

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
  # these labels can be applied automatically
  # from the labels in the pod template if not set
  # labels:
  #   app: guestbook
  #   tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
  # selector can be applied automatically
  # from the labels in the pod template if not set,
  # but we are specifying the selector here to
  # demonstrate its usage.
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
```

```
template:
  metadata:
    labels:
      app: guestbook
      tier: frontend
  spec:
    containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
          - name: GET_HOSTS_FROM
            value: dns
            # If your cluster config does not include a dns service, then to
            # instead access environment variables to find service host
            # info, comment out the 'value: dns' line above, and uncomment the
            # line below.
            # value: env
    ports:
      - containerPort: 80
```

for GitBook update 2017-08-07 13:54:27

Job

Job负责批处理任务，即仅执行一次的任务，它保证批处理任务的一个或多个Pod成功结束。

Job Spec格式

- spec.template格式同Pod
- RestartPolicy仅支持Never或OnFailure
- 单个Pod时，默认Pod成功运行后Job即结束
- .spec.completions 标志Job结束需要成功运行的Pod个数，默认为1
- .spec.parallelism 标志并行运行的Pod的个数，默认为1
- spec.activeDeadlineSeconds 标志失败Pod的重试最大时间，超过这个时间不会继续重试

一个简单的例子：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(20
00)"]
      restartPolicy: Never
```

```
$ kubectl create -f ./job.yaml
job "pi" created
$ pods=$(kubectl get pods --selector=job-name=pi --output=jsonpath={.items..metadata.name})
$ kubectl logs $pods
3.141592653589793238462643383279502...
```

Bare Pods

所谓Bare Pods是指直接用PodSpec来创建的Pod（即不在ReplicaSets或者ReplicationController的管理之下的Pods）。这些Pod在Node重启后不会自动重启，但Job则会创建新的Pod继续任务。所以，推荐使用Job来替代Bare Pods，即便是应用只需要一个Pod。

for GitBook update 2017-08-07 13:54:27

CronJob

CronJob即定时任务，就类似于Linux系统的crontab，在指定的时间周期运行指定的任务。在Kubernetes 1.5，使用CronJob需要开启 `batch/v2alpha1 API`，即 `-runtime-config=batch/v2alpha1`。

CronJob Spec

- `.spec.schedule` 指定任务运行周期，格式同Cron
- `.spec.jobTemplate` 指定需要运行的任务，格式同Job
- `.spec.startingDeadlineSeconds` 指定任务开始的截止期限
- `.spec.concurrencyPolicy` 指定任务的并发策略，支持Allow、Forbid和Replace三个选项

```
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure
```

2.2.13 CronJob

```
$ kubectl create -f cronjob.yaml
cronjob "hello" created
```

当然，也可以用 `kubectl run` 来创建一个CronJob：

```
kubectl run hello --schedule="*/1 * * * *" --restart=OnFailure -
--image=busybox -- /bin/sh -c "date; echo Hello from the Kubernetes cluster"
```

```
$ kubectl get cronjob
NAME      SCHEDULE      SUSPEND      ACTIVE      LAST-SCHEDULE
hello     */1 * * * *    False        0           <none>
$ kubectl get jobs
NAME          DESIRED      SUCCESSFUL      AGE
hello-1202039034   1           1            49s
$ pods=$(kubectl get pods --selector=job-name=hello-1202039034 --
-output=jsonpath={.items..metadata.name} -a)
$ kubectl logs $pods
Mon Aug 29 21:34:09 UTC 2016
Hello from the Kubernetes cluster

# 注意，删除cronjob的时候不会自动删除job，这些job可以用kubectl delete job来删除
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

for GitBook update 2017-08-07 13:54:27

Ingress解析

前言

这是kubernetes官方文档中[Ingress Resource](#)的翻译，后面的章节会讲到使用[Traefik](#)来做Ingress controller，文章末尾给出了几个相关链接。

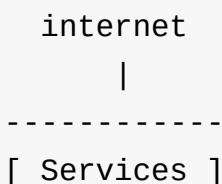
术语

在本篇文章中你将会看到一些在其他地方被交叉使用的术语，为了防止产生歧义，我们首先来澄清下。

- 节点：Kubernetes集群中的一台物理机或者虚拟机。
- 集群：位于Internet防火墙后的节点，这是Kubernetes管理的主要计算资源。
- 边界路由器：为集群强制执行防火墙策略的路由器。这可能是由云提供商或物理硬件管理的网关。
- 集群网络：一组逻辑或物理链接，可根据Kubernetes[网络模型](#)实现群集内的通信。集群网络的实现包括Overlay模型的[flannel](#) 和基于SDN的[OVS](#)。
- 服务：使用标签选择器标识一组pod成为的Kubernetes[服务](#)。除非另有说明，否则服务假定在集群网络内仅可通过虚拟IP访问。

什么是Ingress？

通常情况下，service和pod仅可在集群内部网络中通过IP地址访问。所有到达边界路由器的流量或被丢弃或被转发到其他地方。从概念上讲，可能像下面这样：



Ingress是授权入站连接到达集群服务的规则集合。

```
internet
  |
[ Ingress ]
--|-----|--
[ Services ]
```

你可以给Ingress配置提供外部可访问的URL、负载均衡、SSL、基于名称的虚拟主机等。用户通过POST Ingress资源到API server的方式来请求ingress。Ingress controller负责实现Ingress，通常使用负载平衡器，它还可以配置边界路由和其他前端，这有助于以HA方式处理流量。

先决条件

在使用Ingress resource之前，有必要先了解下面几件事情。Ingress是beta版本的resource，在kubernetes1.1之前还没有。你需要一个 Ingress Controller 来实现 Ingress，单纯的创建一个 Ingress 没有任何意义。

GCE/GKE会在master节点上部署一个ingress controller。你可以在一个pod中部署任意个自定义的ingress controller。你必须正确地annotate每个ingress，比如 运行多个ingress controller 和 关闭glbc.

确定你已经阅读了Ingress controller的beta版本限制。在非GCE/GKE的环境中，你需要在pod中部署一个controller。

Ingress Resource

最简化的Ingress配置：

```

1: apiVersion: extensions/v1beta1
2: kind: Ingress
3: metadata:
4:   name: test-ingress
5: spec:
6:   rules:
7:     - http:
8:       paths:
9:         - path: /testpath
10:        backend:
11:          serviceName: test
12:          servicePort: 80

```

如果你没有配置*Ingress controller*就将其`POST`到*API server*不会有任何用处

配置说明

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

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

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

10-12行：正如[services doc](#)中描述的那样，`backend`是一个`service:port`的组合。`Ingress`的流量被转发到它所匹配的`backend`。

全局参数：为了简单起见，Ingress示例中没有全局参数，请参阅资源完整定义的[api参考](#)。在所有请求都不能跟spec中的path匹配的情况下，请求被发送到Ingress controller的默认后端，可以指定全局缺省backend。

Ingress controllers

为了使Ingress正常工作，集群中必须运行Ingress controller。这与其他类型的控制器不同，其他类型的控制器通常作为 `kube-controller-manager` 二进制文件的一部分运行，在集群启动时自动启动。你需要选择最适合自己的Ingress controller或者自己实现一个。示例和说明可以在[这里](#)找到。

在你开始前

以下文档描述了Ingress资源中公开的一组跨平台功能。理想情况下，所有的Ingress controller都应该符合这个规范，但是我们还没有实现。GCE和nginx控制器的文档分别在[这里](#)和[这里](#)。确保您查看控制器特定的文档，以便您了解每个文档的注意事项。

Ingress类型

单Service Ingress

Kubernetes中已经存在一些概念可以暴露单个service（查看[替代方案](#)），但是你仍然可以通过Ingress来实现，通过指定一个没有rule的默认backend的方式。

ingress.yaml定义文件：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

使用 `kubectl create -f` 命令创建，然后查看ingress：

\$ kubectl get ing	NAME	RULE	BACKEND	ADDRESS
	<code>test-ingress</code>	-	testsingress:80	107.178.254.228

2.2.14 Ingress

107.178.254.228 就是Ingress controller为了实现Ingress而分配的IP地址。 RULE 列表示所有发送给该IP的流量都被转发到了 BACKEND 所列的Kubernetes service上。

简单展开

如前面描述的那样，kubernetes pod中的IP只在集群网络内部可见，我们需要在边界设置一个东西，让它能够接收ingress的流量并将它们转发到正确的端点上。这个东西一般是高可用的loadbalancer。使用Ingress能够允许你将loadbalancer的个数降低到最少，例如，假如你想要创建这样的一个设置：

```
foo.bar.com -> 178.91.123.132 -> / foo      s1:80  
                      / bar      s2:80
```

你需要一个这样的ingress：

```
apiVersion: extensions/v1beta1  
kind: Ingress  
metadata:  
  name: test  
spec:  
  rules:  
    - host: foo.bar.com  
      http:  
        paths:  
          - path: /foo  
            backend:  
              serviceName: s1  
              servicePort: 80  
          - path: /bar  
            backend:  
              serviceName: s2  
              servicePort: 80
```

使用 `kubectl create -f` 创建完ingress后：

```
$ kubectl get ing
NAME      RULE          BACKEND    ADDRESS
test      -
          foo.bar.com
          /foo        s1:80
          /bar        s2:80
```

只要服务 (s1, s2) 存在，Ingress controller就会将提供一个满足该Ingress的特定 loadbalancer实现。这一步完成后，您将在Ingress的最后一列看到loadbalancer的地址。

基于名称的虚拟主机

Name-based的虚拟主机在同一个IP地址下拥有多个主机名。

```
foo.bar.com --|           | -> foo.bar.com s1:80
              | 178.91.123.132 |
bar.foo.com --|           | -> bar.foo.com s2:80
```

下面这个ingress说明基于[Host header](#)的后端loadbalancer的路由请求：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: s1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: s2
          servicePort: 80
```

默认**backend**：一个没有rule的ingress，如前面章节中所示，所有流量都将发送到一个默认backend。你可以用该技巧通知loadbalancer如何找到你网站的404页面，通过制定一些列rule和一个默认backend的方式。如果请求header中的host不能跟ingress中的host匹配，并且/或请求的URL不能与任何一个path匹配，则流量将路由到你的默认backend。

TLS

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

```

apiVersion: v1
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
kind: Secret
metadata:
  name: testsecret
  namespace: default
type: Opaque

```

在Ingress中引用这个secret将通知Ingress controller使用TLS加密从将客户端到loadbalancer的channel：

```

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: no-rules-map
spec:
  tls:
    - secretName: testsecret
  backend:
    serviceName: s1
    servicePort: 80

```

请注意，各种Ingress controller支持的TLS功能之间存在差距。请参阅有关[nginx](#)，[GCE](#)或任何其他平台特定Ingress controller的文档，以了解TLS在你的环境中的工作原理。

Ingress controller启动时附带一些适用于所有Ingress的负载平衡策略设置，例如负载均衡算法，后端权重方案等。更高级的负载平衡概念（例如持久会话，动态权重）尚未在Ingress中公开。你仍然可以通过[service loadbalancer](#)获取这些功能。随着时间的推移，我们计划将适用于跨平台的负载平衡模式加入到Ingress资源中。

还值得注意的是，尽管健康检查不直接通过Ingress公开，但Kubernetes中存在并行概念，例如[准备探查](#)，可以使你达成相同的最终结果。请查看特定控制器的文档，以了解他们如何处理健康检查（[nginx](#)，[GCE](#)）。

更新Ingress

假如你想要向已有的ingress中增加一个新的Host，你可以编辑和更新该ingress：

```
$ kubectl get ing
NAME      RULE          BACKEND    ADDRESS
test      -              178.91.123.132
          foo.bar.com
          /foo           s1:80
$ kubectl edit ing test
```

这会弹出一个包含已有的yaml文件的编辑器，修改它，增加新的Host配置。

```
spec:
rules:
- host: foo.bar.com
  http:
    paths:
    - backend:
        serviceName: s1
        servicePort: 80
        path: /foo
- host: bar.baz.com
  http:
    paths:
    - backend:
        serviceName: s2
        servicePort: 80
        path: /foo
..
```

保存它会更新API server中的资源，这会触发ingress controller重新配置loadbalancer。

```
$ kubectl get ing
NAME      RULE          BACKEND    ADDRESS
test      -              178.91.123.132
          foo.bar.com
          /foo           s1:80
          bar.baz.com
          /foo           s2:80
```

在一个修改过的ingress yaml文件上调用 `kubectl replace -f` 命令一样可以达到同样的效果。

跨可用域故障

在不通云供应商之间，跨故障域的流量传播技术有所不同。有关详细信息，请查看相关Ingress controller的文档。有关在federation集群中部署Ingress的详细信息，请参阅[federation文档](#)。

未来计划

- 多样化的HTTPS/TLS模型支持（如SNI，re-encryption）
- 通过声明来请求IP或者主机名
- 结合L4和L7 Ingress
- 更多的Ingress controller

请跟踪[L7和Ingress的proposal](#)，了解有关资源演进的更多细节，以及[Ingress repository](#)，了解有关各种Ingress controller演进的更多详细信息。

替代方案

你可以通过很多种方式暴露service而不必直接使用ingress：

- 使用[Service.Type=LoadBalancer](#)
- 使用[Service.Type=NodePort](#)
- 使用[Port Proxy](#)
- 部署一个[Service loadbalancer](#) 这允许你在多个service之间共享单个IP，并通过[Service Annotations](#)实现更高级的负载平衡。

参考

[Kubernetes Ingress Resource](#)

[使用NGINX Plus负载均衡Kubernetes服务](#)

[使用NGINX和NGINX Plus的Ingress Controller进行Kubernetes的负载均衡](#)

[Kubernetes : Ingress Controller with Traefik and Let's Encrypt](#)

[Kubernetes : Traefik and Let's Encrypt at scale](#)

[Kubernetes Ingress Controller-Traefik](#)

[Kubernetes 1.2 and simplifying advanced networking with Ingress](#)

for GitBook update 2017-08-07 13:54:27

前言

其实ConfigMap功能在Kubernetes1.2版本的时候就有了，许多应用程序会从配置文件、命令行参数或环境变量中读取配置信息。这些配置信息需要与docker image解耦，你总不能每修改一个配置就重做一个image吧？ConfigMap API给我们提供了向容器中注入配置信息的机制，ConfigMap可以被用来保存单个属性，也可以用来保存整个配置文件或者JSON二进制大对象。

ConfigMap概览

ConfigMap API资源用来保存**key-value pair**配置数据，这个数据可以在**pods**里使用，或者被用来为像**controller**一样的系统组件存储配置数据。虽然ConfigMap跟**Secrets**类似，但是ConfigMap更方便的处理不含敏感信息的字符串。注意：ConfigMaps不是属性配置文件的替代品。ConfigMaps只是作为多个**properties**文件的引用。你可以把它理解为Linux系统中的`/etc`目录，专门用来存储配置文件的目录。下面举个例子，使用ConfigMap配置来创建Kubernetes Volumes，ConfigMap中的每个**data**项都会成为一个新文件。

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

`data`一栏包括了配置数据，ConfigMap可以被用来保存单个属性，也可以用来保存一个配置文件。配置数据可以通过很多种方式在Pods里被使用。ConfigMaps可以被用来：

1. 设置环境变量的值
2. 在容器里设置命令行参数
3. 在数据卷里面创建config文件

用户和系统组件两者都可以在ConfigMap里面存储配置数据。

其实不用看下面的文章，直接从 `kubectl create configmap -h` 的帮助信息中就可以对ConfigMap究竟如何创建略知一二了。

Examples:

```
# Create a new configmap named my-config based on folder bar
kubectl create configmap my-config --from-file=path/to/bar

# Create a new configmap named my-config with specified keys instead of file basenames on disk
kubectl create configmap my-config --from-file=key1=/path/to/bar/file1.txt --from-file=key2=/path/to/bar/file2.txt

# Create a new configmap named my-config with key1=config1 and key2=config2
kubectl create configmap my-config --from-literal=key1=config1 --from-literal=key2=config2
```

创建ConfigMaps

可以使用该命令，用给定值、文件或目录来创建ConfigMap。

```
kubectl create configmap
```

使用目录创建

比如我们已经有个了包含一些配置文件，其中包含了我们想要设置的ConfigMap的值：

2.2.15 ConfigMap

```
$ ls docs/user-guide/configmap/kubectl/  
game.properties  
ui.properties  
  
$ cat docs/user-guide/configmap/kubectl/game.properties  
enemies=aliens  
lives=3  
enemies.cheat=true  
enemies.cheat.level=noGoodRotten  
secret.code.passphrase=UUDDLRLRBABAS  
secret.code.allowed=true  
secret.code.lives=30  
  
$ cat docs/user-guide/configmap/kubectl/ui.properties  
color.good=purple  
color.bad=yellow  
allow.textmode=true  
how.nice.to.look=fairlyNice
```

使用下面的命令可以创建一个包含目录中所有文件的ConfigMap。

```
$ kubectl create configmap game-config --from-file=docs/user-guide/configmap/kubectl
```

`--from-file` 指定在目录下的所有文件都会被用在ConfigMap里面创建一个键值对，键的名字就是文件名，值就是文件的内容。

让我们来看一下这个命令创建的ConfigMap：

2.2.15 ConfigMap

```
$ kubectl describe configmaps game-config
Name:          game-config
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
=====
game.properties:      158 bytes
ui.properties:        83 bytes
```

我们可以看到那两个key是从kubectl指定的目录中的文件名。这些key的内容可能会很大，所以在kubectl describe的输出中，只能够看到键的名字和他们的大小。如果想要看到键的值的话，可以使用 kubectl get :

```
$ kubectl get configmaps game-config -o yaml
```

我们以 yaml 格式输出配置。

```
apiVersion: v1
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"
  selfLink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

使用文件创建

刚才使用目录创建的时候我们 `--from-file` 指定的是一个目录，只要指定为一个文件就可以从单个文件中创建ConfigMap。

```
$ kubectl create configmap game-config-2 --from-file=docs/user-guide/configmap/kubectl/game.properties

$ kubectl get configmaps game-config-2 -o yaml
```

```
apiVersion: v1
data:
  game-special-key: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

`-from-file` 这个参数可以使用多次，你可以使用两次分别指定上个实例中的那两个配置文件，效果就跟指定整个目录是一样的。

使用字面值创建

使用文字值创建，利用 `--from-literal` 参数传递配置信息，该参数可以使用多次，格式如下：

```
$ kubectl create configmap special-config --from-literal=special
.how=very --from-literal=special.type=charm

$ kubectl get configmaps special-config -o yaml
```

```
apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

Pod中使用ConfigMap

使用**ConfigMap**来替代环境变量

ConfigMap可以被用来填入环境变量。看下下面的ConfigMap。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

2.2.15 ConfigMap

我们可以在Pod中这样使用ConfigMap：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
      envFrom:
        - configMapRef:
            name: env-config
  restartPolicy: Never
```

这个Pod运行后会输出如下几行：

```
SPECIAL_LEVEL_KEY=very
SPECIAL_TYPE_KEY=charm
log_level=INFO
```

用**ConfigMap**设置命令行参数

ConfigMap也可以被使用来设置容器中的命令或者参数值。它使用的是Kubernetes的\$(VAR_NAME)替换语法。我们看下下面这个ConfigMap。

2.2.15 ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

为了将ConfigMap中的值注入到命令行的参数里面，我们还要像前面那个例子一样使用环境变量替换语法 `$(VAR_NAME)`。（其实这个东西就是给Docker容器设置环境变量，以前我创建镜像的时候经常这么玩，通过`docker run`的时候指定`-e`参数修改镜像里的环境变量，然后`docker`的`CMD`命令再利用该`$(VAR_NAME)`通过`sed`来修改配置文件或者作为命令行启动参数。）

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

2.2.15 ConfigMap

运行这个Pod后会输出：

```
very charm
```

通过数据卷插件使用**ConfigMap**

ConfigMap也可以在数据卷里面被使用。还是这个ConfigMap。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

在数据卷里面使用这个ConfigMap，有不同的选项。最基本的就是将文件填入数据卷，在这个文件中，键就是文件名，键值就是文件内容：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "cat /etc/config/special.how"]
  volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
  restartPolicy: Never
```

2.2.15 ConfigMap

运行这个Pod的输出是 very 。

我们也可以在ConfigMap值被映射的数据卷里控制路径。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "cat /etc/config/path/to/special
-key" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
          - key: special.how
            path: path/to/special-key
  restartPolicy: Never
```

运行这个Pod后的结果是 very 。

for GitBook update 2017-08-07 13:54:27

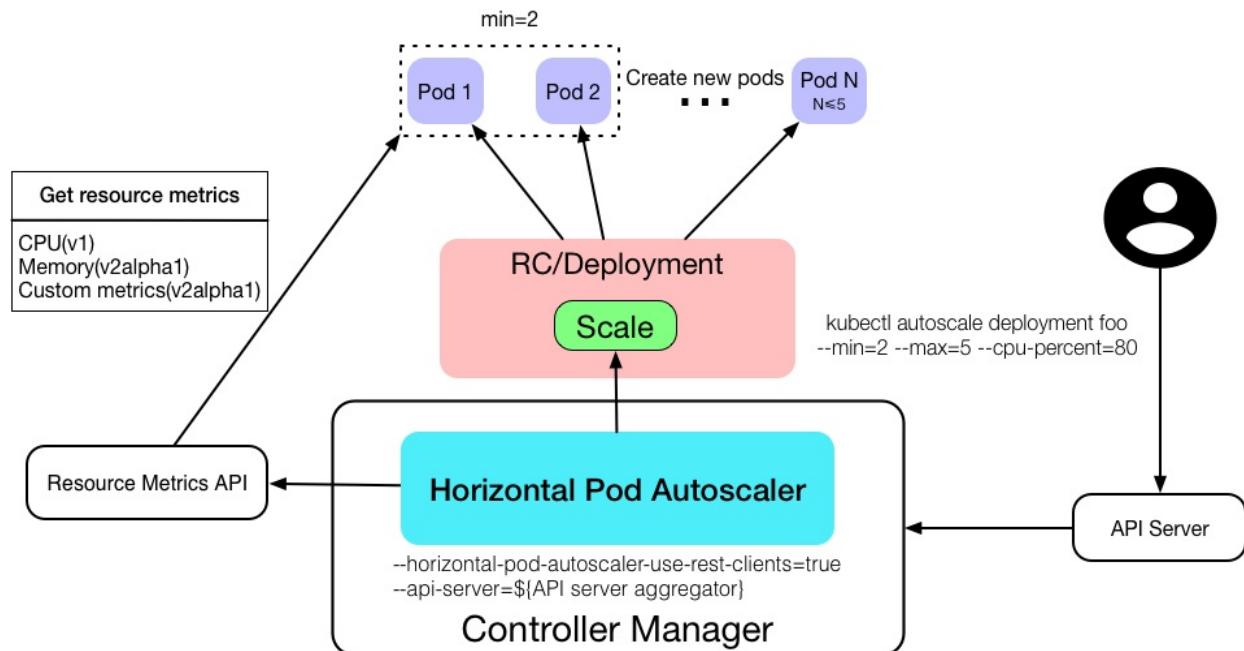
Horizontal Pod Autoscaling

应用的资源使用率通常都有高峰和低谷的时候，如何削峰填谷，提高集群的整体资源利用率，让service中的Pod个数自动调整呢？这就有赖于Horizontal Pod Autoscaling了，顾名思义，使Pod水平自动缩放。这个Object（跟Pod、Deployment一样都是API resource）也是最能体现kubernetes之于传统运维价值的地方，不再需要手动扩容了，终于实现自动化了，还可以自定义指标，没准未来还可以通过人工智能自动进化呢！

Horizontal Pod Autoscaling仅适用于Deployment和ReplicationController（ReplicaSet已经被ReplicationController取代），在V1版本中仅支持根据Pod的CPU利用率扩所容，在v1alpha版本中，支持根据内存和用户自定义的metric扩缩容。

如果你不想看下面的文章可以直接看下面的示例图，组件交互、组件的配置、命令示例，都画在图上了。

Horizontal Pod Autoscaling由API server和controller共同实现。



Source <https://github.com/rootsongjc/kubernetes-handbook>

Figure: horizontal-pod-autoscaler

Metrics 支持

在不同版本得API中，HPA autoscale时可以根据以下指标来判断：

- autoscaling/v1
 - CPU
- autoscaling/v2alpha1
 - 内存
 - 自定义metrics
 - kubernetes1.6起支持自定义metrics，但是必须在kube-controller-manager中配置如下两项：
 - --horizontal-pod-autoscaler-use-rest-clients=true
 - --api-server 指向[kube-aggregator](#)，也可以使用heapster来实现，通过在启动heapster的时候指定 --api-server=true。
 - 查看[kubernetes metrics](#)
 - 多种metrics组合
 - HPA会根据每个metric的值计算出scale的值，并将最大的那个指作为扩容的最终结果。

使用kubectl管理

Horizontal Pod Autoscaling作为API resource也可以像Pod、Deployment一样使用kubectl命令管理，使用方法跟它们一样，资源名称为 `hpa`。

```
kubectl create hpa
kubectl get hpa
kubectl describe hpa
kubectl delete hpa
```

有一点不同的是，可以直接使用 `kubectl autoscale` 直接通过命令行的方式创建Horizontal Pod Autoscaler。

用法如下：

```
kubectl autoscale (-f FILENAME | TYPE NAME | TYPE/NAME) [--min=M  
INPODS] --max=MAXPODS  
[--cpu-percent=CPU] [flags] [options]
```

举个例子：

```
kubectl autoscale deployment foo --min=2 --max=5 --cpu-percent=8  
0
```

为Deployment foo创建一个autoscaler，当Pod的CPU利用率达到80%的时候，RC的replica数在2到5之间。该命令的详细使用文档见<https://kubernetes.io/docs/user-guide/kubectl/v1.6/#autoscale>。

注意：如果为ReplicationController创建HPA的话，无法使用rolling update，但是对于Deployment来说是可以的，因为Deployment在执行rolling update的时候会自动创建新的ReplicationController。

什么是 Horizontal Pod Autoscaling？

利用Horizontal Pod Autoscaling，kubernetes能够根据监测到的CPU利用率（或者在alpha版本中支持的应用提供的metric）自动的扩容replication controller，deployment和replica set。

Horizontal Pod Autoscaler作为kubernetes API resource和controller的实现。Resource确定controller的行为。Controller会根据监测到用户指定的目标的CPU利用率周期性得调整replication controller或deployment的replica数量。

Horizontal Pod Autoscaler 如何工作？

Horizontal Pod Autoscaler由一个控制循环实现，循环周期由controller manager中的`--horizontal-pod-autoscaler-sync-period`标志指定（默认是30秒）。

在每个周期内，controller manager会查询HorizontalPodAutoscaler中定义的metric的资源利用率。Controller manager从resource metric API（每个pod的resource metric）或者自定义metric API（所有的metric）中获取metric。

- 每个 Pod 的 resource metric (例如 CPU) , controller 通过 resource metric API 获取 HorizontalPodAutoscaler 中定义的每个 Pod 中的 metric。然后，如果设置了目标利用率，controller 计算利用的值与每个 Pod 的容器里的 resource request 值的百分比。如果设置了目标原始值，将直接使用该原始 metric 值。然后 controller 计算所有目标 Pod 的利用率或原始值 (取决于所指定的目标类型) 的平均值，产生一个用于缩放所需 replica 数量的比率。请注意，如果某些 Pod 的容器没有设置相关的 resource request，则不会定义 Pod 的 CPU 利用率，并且 Aucoscaler 也不会对该 metric 采取任何操作。有关自动缩放算法如何工作的更多细节，请参阅 [自动缩放算法设计文档](#)。
- 对于每个 Pod 自定义的 metric，controller 功能类似于每个 Pod 的 resource metric，只是它使用原始值而不是利用率值。
- 对于 object metric，获取单个度量 (描述有问题的对象)，并与目标值进行比较，以产生如上所述的比率。

HorizontalPodAutoscaler 控制器可以以两种不同的方式获取 metric：直接的 Heapster 访问和 REST 客户端访问。

当使用直接的 Heapster 访问时，HorizontalPodAutoscaler 直接通过 API 服务器的服务代理子资源查询 Heapster。需要在集群上部署 Heapster 并在 kube-system namespace 中运行。

有关 REST 客户端访问的详细信息，请参阅 [支持自定义度量](#)。

Autoscaler 访问相应的 replication controller，deployment 或 replica set 来缩放子资源。

Scale 是一个允许您动态设置副本数并检查其当前状态的接口。

有关缩放子资源的更多细节可以在 [这里](#) 找到。

API Object

Horizontal Pod Autoscaler 是 kubernetes 的 `autoscaling` API 组中的 API 资源。当前的稳定版本中，只支持 CPU 自动扩缩容，可以在 `autoscaling/v1` API 版本中找到。

在 alpha 版本中支持根据内存和自定义 metric 扩缩容，可以在 `autoscaling/v2alpha1` 中找到。`autoscaling/v2alpha1` 中引入的新字段在 `autoscaling/v1` 中是做为 annotation 而保存的。

关于该 API 对象的更多信息，请参阅 [HorizontalPodAutoscaler Object](#)。

在 kubectl 中支持 Horizontal Pod Autoscaling

Horizontal Pod Autoscaler 和其他的所有 API 资源一样，通过 `kubectl` 以标准的方式支持。

我们可以使用 `kubectl create` 命令创建一个新的 autoscaler。

我们可以使用 `kubectl get hpa` 列出所有的 autoscaler，使用 `kubectl describe hpa` 获取其详细信息。

最后我们可以使用 `kubectl delete hpa` 删除 autoscaler。

另外，可以使用 `kubectl autoscale` 命令，很轻易的就可以创建一个 Horizontal Pod Autoscaler。

例如，执行 `kubectl autoscale rc foo --min=2 --max=5 --cpu-percent=80` 命令将为 replication controller `foo` 创建一个 autoscaler，目标的 CPU 利用率是 80%，replica 的数量介于 2 和 5 之间。

关于 `kubectl autoscale` 的更多信息请参阅 [这里](#)。

滚动更新期间的自动扩缩容

目前在 Kubernetes 中，可以通过直接管理 replication controller 或使用 deployment 对象来执行 滚动更新，该 deployment 对象为您管理基础 replication controller。

Horizontal Pod Autoscaler 仅支持后一种方法：Horizontal Pod Autoscaler 被绑定到 deployment 对象，它设置 deployment 对象的大小，deployment 负责设置底层 replication controller 的大小。

Horizontal Pod Autoscaler 不能使用直接操作 replication controller 进行滚动更新，即不能将 Horizontal Pod Autoscaler 绑定到 replication controller，并进行滚动更新（例如使用 `kubectl rolling-update`）。

这不行的原因是，当滚动更新创建一个新的 replication controller 时，Horizontal Pod Autoscaler 将不会绑定到新的 replication controller 上。

支持多个 metric

Kubernetes 1.6 中增加了支持基于多个 metric 的扩缩容。您可以使用 `autoscaling/v2alpha1` API 版本来为 Horizontal Pod Autoscaler 指定多个 metric。然后 Horizontal Pod Autoscaler controller 将权衡每一个 metric，并根据该 metric 提议一个新的 scale。在所有提议里最大的那个 scale 将作为最终的 scale。

支持自定义 metric

注意： Kubernetes 1.2 根据特定于应用程序的 metric，通过使用特殊注释的方式，增加了对缩放的 alpha 支持。

在 Kubernetes 1.6 中删除了对这些注释的支持，有利于 `autoscaling/v2alpha1` API。虽然旧的收集自定义 metric 的旧方法仍然可用，但是这些 metric 将不可供 Horizontal Pod Autoscaler 使用，并且用于指定要缩放的自定义 metric 的以前的注释也不再受 Horizontal Pod Autoscaler 认可。

Kubernetes 1.6 增加了在 Horizontal Pod Autoscaler 中使用自定义 metric 的支持。

您可以为 `autoscaling/v2alpha1` API 中使用的 Horizontal Pod Autoscaler 添加自定义 metric。

Kubernetes 然后查询新的自定义 metric API 来获取相应自定义 metric 的值。

前提条件

为了在 Horizontal Pod Autoscaler 中使用自定义 metric，您必须在您集群的 controller manager 中将 `--horizontal-pod-autoscaler-use-rest-clients` 标志设置为 true。然后，您必须通过将 controller manager 的目标 API server 设置为 API server aggregator（使用 `--apiserver` 标志），配置您的 controller manager 通过 API server aggregator 与 API server 通信。Resource metric API 和自定义 metric API 也必须向 API server aggregator 注册，并且必须由集群上运行的 API server 提供。

您可以使用 Heapster 实现 resource metric API，方法是将 `--api-server` 标志设置为 true 并运行 Heapster。单独的组件必须提供自定义 metric API（有关自定义 metric API 的更多信息，可从 k8s.io/metrics repository 获得）。

进一步阅读

- 设计文档：[Horizontal Pod Autoscaling](#)
- kubectl autoscale 命令：[kubectl autoscale](#)
- 使用例子：[Horizontal Pod Autoscaler](#)

参考

HPA设计文

档：<https://github.com/kubernetes/community/blob/master/contributors/design-proposals/horizontal-pod-autoscaler.md>

HPA说明：<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>

HPA详解：<https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>

kubectl autoscale命令详细使用说明：<https://kubernetes.io/docs/user-guide/kubectl/v1.6/#autoscale>

自定义metrics开发：<https://github.com/kubernetes/metrics>

for GitBook update 2017-08-07 13:54:27

Label

Label是附着到object上（例如Pod）的键值对。可以在创建object的时候指定，也可以在object创建后随时指定。Labels的值对系统本身并没有什么含义，只是对用户才有意义。

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

Kubernetes最终将对labels最终索引和反向索引来优化查询和watch，在UI和命令行中会对它们排序。不要在label中使用大型、非标识的结构化数据，记录这样的数据应该用annotation。

动机

Label能够将组织架构映射到系统架构上（就像是康威定律），这样能够更便于微服务的管理，你可以给object打上如下类型的label：

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

语法和字符集

Label key的组成：

- 不得超过63个字符
- 可以使用前缀，使用/分隔，前缀必须是DNS子域，不得超过253个字符，系统中的自动化组件创建的label必须指定前缀， kubernetes.io/ 由kubernetes

保留

- 起始必须是字母（大小写都可以）或数字，中间可以有连字符、下划线和点

Label value的组成：

- 不得超过63个字符
- 起始必须是字母（大小写都可以）或数字，中间可以有连字符、下划线和点

Label selector

Label不是唯一的，很多object可能有相同的label。

通过label selector，客户端／用户可以指定一个object集合，通过label selector对object的集合进行操作。

Label selector有两种类型：

- *equality-based*：可以使用 = 、 == 、 != 操作符，可以使用逗号分隔多个表达式
- *set-based*：可以使用 in 、notin 、 ! 操作符，另外还可以没有操作符，直接写出某个label的key，表示过滤有某个key的object而不管该key的value是什么值，！表示没有该label的object

示例

```
$ kubectl get pods -l environment=production,tier=frontend
$ kubectl get pods -l 'environment in (production),tier in (front
tend)'
$ kubectl get pods -l 'environment in (production, qa)'
$ kubectl get pods -l 'environment,environment notin (frontend)'
```

在API object中设置label selector

在 service 、 replicationcontroller 等object中有对pod的label selector，使用方法只能使用等于操作，例如：

2.2.17 Label

```
selector:  
  component: redis
```

在 Job 、 Deployment 、 ReplicaSet 和 DaemonSet 这些object中，支持set-based的过滤，例如：

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

如Service通过label selector将同一类型的pod作为一个服务expose出来。

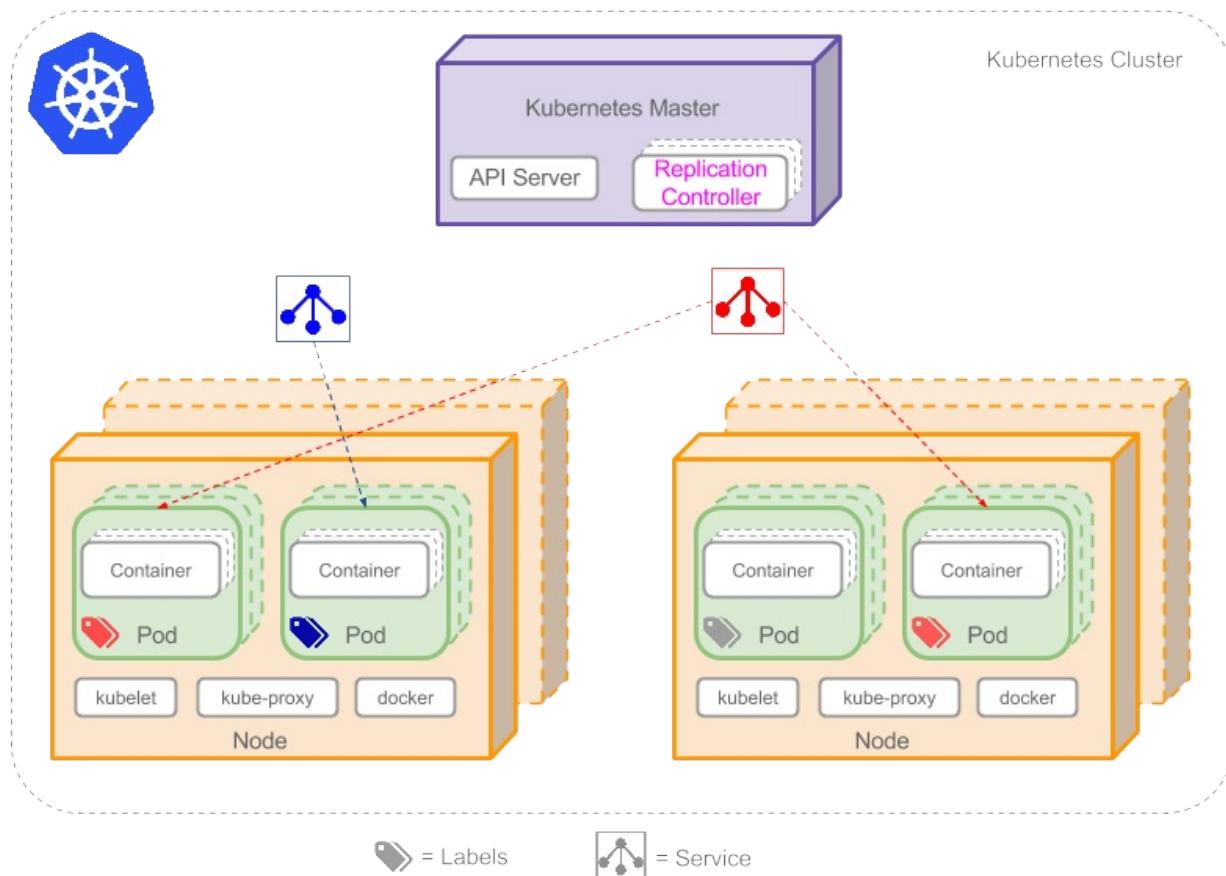


Figure: label示意图

另外在node affinity和pod affinity中的label selector的语法又有些许不同，示例如下：

```
affinity:  
  nodeAffinity:  
    requiredDuringSchedulingIgnoredDuringExecution:  
      nodeSelectorTerms:  
        - matchExpressions:  
            - key: kubernetes.io/e2e-az-name  
              operator: In  
              values:  
                - e2e-az1  
                - e2e-az2  
    preferredDuringSchedulingIgnoredDuringExecution:  
      - weight: 1  
        preference:  
          matchExpressions:  
            - key: another-node-label-key  
              operator: In  
              values:  
                - another-node-label-value
```

详见[node selection](#)。

for GitBook update 2017-08-07 13:54:27

用户指南

该章节主要记录kubernetes使用过程中的一些配置技巧和操作。

- 配置Pod的liveness和readiness探针
- 管理集群中的TLS

for GitBook update 2017-08-07 13:54:27

配置Pod的liveness和readiness探针

当你使用kubernetes的时候，有没有遇到过Pod在启动后一会就挂掉然后又重新启动这样的恶性循环？你有没有想过kubernetes是如何检测pod是否还存活？虽然容器已经启动，但是kubernetes如何知道容器的进程是否准备好对外提供服务了呢？让我们通过kubernetes官网的这篇文章[Configure Liveness and Readiness Probes](#)，来一探究竟。

本文将向展示如何配置容器的存活和可读性探针。

Kubelet使用liveness probe（存活探针）来确定何时重启容器。例如，当应用程序处于运行状态但无法做进一步操作，liveness探针将捕获到deadlock，重启处于该状态下的容器，使应用程序在存在bug的情况下依然能够继续运行下去（谁的程序还没几个bug呢）。

Kubelet使用readiness probe（就绪探针）来确定容器是否已经就绪可以接受流量。只有当Pod中的容器都处于就绪状态时kubelet才会认定该Pod处于就绪状态。该信号的作用是控制哪些Pod应该作为service的后端。如果Pod处于非就绪状态，那么它们将会被从service的load balancer中移除。

定义 liveness 命令

许多长时间运行的应用程序最终会转换到broken状态，除非重新启动，否则无法恢复。Kubernetes提供了liveness probe来检测和补救这种情况。

在本次练习将基于 `gcr.io/google_containers/busybox` 镜像创建运行一个容器的Pod。以下是Pod的配置文件 `exec-liveness.yaml`：

```

apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
    - name: liveness
      args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
      image: gcr.io/google_containers/busybox
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
        initialDelaySeconds: 5
        periodSeconds: 5

```

该配置文件给Pod配置了一个容器。`periodSeconds` 规定kubelet要每隔5秒执行一次liveness probe。`initialDelaySeconds` 告诉kubelet在第一次执行probe之前要的等待5秒钟。探针检测命令是在容器中执行 `cat /tmp/healthy` 命令。如果命令执行成功，将返回0，kubelet就会认为该容器是活着的并且很健康。如果返回非0值，kubelet就会杀掉这个容器并重启它。

容器启动时，执行该命令：

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"
```

在容器生命的最初30秒内有一个 `/tmp/healthy` 文件，在这30秒内 `cat /tmp/healthy` 命令会返回一个成功的返回码。30秒后，`cat /tmp/healthy` 将返回失败的返回码。

3.1 配置Pod的liveness和readiness探针

创建Pod：

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/exec-liveness.yaml
```

在30秒内，查看Pod的event：

```
kubectl describe pod liveness-exec
```

结果显示没有失败的liveness probe：

FirstSeen	LastSeen	Count	From	SubobjectPath
Type	Reason	Message		
24s	24s	1	{default-scheduler }	No
ormal	Scheduled		Successfully assigned liveness-exec to worker0	
23s	23s	1	{kubelet worker0} spec.containers{liveness}	
	Normal		Pulling	spec.containers{liveness}/busybox"
23s	23s	1	{kubelet worker0} spec.containers{liveness}	
	Normal		Pulled	Successfully pulled image "gcr.io/google_conta
				iners/busybox"
23s	23s	1	{kubelet worker0} spec.containers{liven	
	Normal		Created	Created container with docker id 8
				6849c15382e; Security:[seccomp=unconfined]
23s	23s	1	{kubelet worker0} spec.containers{liven	
	Normal		Started	Started container with docker id 8
				6849c15382e

启动35秒后，再次查看pod的event：

```
kubectl describe pod liveness-exec
```

在最下面有一条信息显示liveness probe失败，容器被删掉并重新创建。

3.1 配置Pod的liveness和readiness探针

FirstSeen Type	LastSeen Reason	Count	From Message	SubobjectPath
37s ormal	37s cheduled	1	{default-scheduler } Successfully assigned liveness-exec to worker0	No
36s	36s Normal	1	{kubelet worker0} spec.containers{liveness} Pulling pulling image "gcr.io/google_containers/busybox"	
36s	36s Normal	1	{kubelet worker0} spec.containers{liveness} Pulled Successfully pulled image "gcr.io/google_containers/busybox"	
36s	36s Normal	1	{kubelet worker0} spec.containers{liveness} Created Created container with docker id 8 6849c15382e; Security:[seccomp=unconfined]	
36s	36s Normal	1	{kubelet worker0} spec.containers{liveness} Started Started container with docker id 8 6849c15382e	
2s	2s Warning	1	{kubelet worker0} spec.containers{liveness} Unhealthy Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory	

再等30秒，确认容器已经重启：

```
kubectl get pod liveness-exec
```

从输出结果来 RESTARTS 值加1了。

NAME	READY	STATUS	RESTARTS	AGE
liveness-exec	1/1	Running	1	1m

定义一个liveness HTTP请求

3.1 配置Pod的liveness和readiness探针

我们还可以使用HTTP GET请求作为liveness probe。下面是一个基于 gcr.io/google_containers/liveness 镜像运行了一个容器的Pod的例子 http-liveness.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    args:
    - /server
    image: gcr.io/google_containers/liveness
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
    initialDelaySeconds: 3
    periodSeconds: 3
```

该配置文件只定义了一个容器，livenessProbe 指定kubeblete需要每隔3秒执行一次liveness probe。initialDelaySeconds 指定kubeblet在该执行第一次探测之前需要等待3秒钟。该探针将向容器中的server的8080端口发送一个HTTP GET请求。如果server的 /healthz 路径的handler返回一个成功的返回码，kubeblet就会认定该容器是活着的并且很健康。如果返回失败的返回码，kubeblet将杀掉该容器并重启它。

任何大于200小于400的返回码都会认定是成功的返回码。其他返回码都会被认为是失败的返回码。

查看该server的源码：[server.go](#).

3.1 配置Pod的liveness和readiness探针

最开始的10秒该容器是活着的，`/healthz` handler返回200的状态码。这之后将返回500的返回码。

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

容器启动3秒后，kubelet开始执行健康检查。第一次健康监测会成功，但是10秒后，健康检查将失败，kubelet将杀掉和重启容器。

创建一个Pod来测试一下HTTP liveness检测：

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/http-liveness.yaml
```

After 10 seconds, view Pod events to verify that liveness probes have failed and the Container has been restarted:

10秒后，查看Pod的event，确认liveness probe失败并重启了容器。

```
kubectl describe pod liveness-http
```

定义TCP liveness探针

第三种liveness probe使用TCP Socket。使用此配置，kubelet将尝试在指定端口上打开容器的套接字。如果可以建立连接，容器被认为是健康的，如果不能就认为是失败的。

```

apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: gcr.io/google_containers/goproxy:0.1
    ports:
    - containerPort: 8080
      readinessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10
      livenessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 15
        periodSeconds: 20

```

如您所见，TCP检查的配置与HTTP检查非常相似。此示例同时使用了readiness和liveness probe。容器启动后5秒钟，kubelet将发送第一个readiness probe。这将尝试连接到端口8080上的goproxy容器。如果探测成功，则该pod将被标记为就绪。Kubelet将每隔10秒钟执行一次该检查。

除了readiness probe之外，该配置还包括liveness probe。容器启动15秒后，kubelet将运行第一个liveness probe。就像readiness probe一样，这将尝试连接到goproxy容器上的8080端口。如果liveness probe失败，容器将重新启动。

使用命名的端口

可以使用命名的ContainerPort作为HTTP或TCP liveness检查：

```

ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port

```

定义readiness探针

有时，应用程序暂时无法对外部流量提供服务。例如，应用程序可能需要在启动期间加载大量数据或配置文件。在这种情况下，你不想杀死应用程序，但你也不想发送请求。Kubernetes提供了readiness probe来检测和减轻这些情况。Pod中的容器可以报告自己还没有准备，不能处理Kubernetes服务发送过来的流量。

Readiness probe的配置跟liveness probe很像。唯一的不同是使用 `readinessProbe` 而不是 `livenessProbe`。

```

readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5

```

Readiness probe的HTTP和TCP的探测器配置跟liveness probe一样。

Readiness和liveness probe可以并行用于同一容器。使用两者可以确保流量无法到达未准备好的容器，并且容器在失败时重新启动。

配置Probe

Probe中有很多精确和详细的配置，通过它们你能准确的控制liveness和readiness检查：

- `initialDelaySeconds` : 容器启动后第一次执行探测是需要等待多少秒。
- `periodSeconds` : 执行探测的频率。默认是10秒，最小1秒。
- `timeoutSeconds` : 探测超时时间。默认1秒，最小1秒。
- `successThreshold` : 探测失败后，最少连续探测成功多少次才被认定为成功。默认是1。对于liveness必须是1。最小值是1。
- `failureThreshold` : 探测成功后，最少连续探测失败多少次才被认定为失败。默认是3。最小值是1。

HTTP probe中可以给 `httpGet` 设置其他配置项：

- `host` : 连接的主机名，默认连接到pod的IP。你可能想在http header中设置"Host"而不是使用IP。
- `scheme` : 连接使用的schema，默认HTTP。
- `path` : 访问的HTTP server的path。
- `httpHeaders` : 自定义请求的header。HTTP运行重复的header。
- `port` : 访问的容器的端口名字或者端口号。端口号必须介于1和65525之间。

对于HTTP探测器，kubelet向指定的路径和端口发送HTTP请求以执行检查。

Kubelet将probe发送到容器的IP地址，除非地址被 `httpGet` 中的可选 `host` 字段覆盖。在大多数情况下，你不想设置主机字段。有一种情况下你可以设置它。假设容器在127.0.0.1上侦听，并且Pod的 `hostNetwork` 字段为true。然后，在 `httpGet` 下的 `host` 应该设置为127.0.0.1。如果你的pod依赖于虚拟主机，这可能是更常见的情况，你不应该是用 `host`，而是应该在 `httpHeaders` 中设置 `Host` 头。

参考

- 关于 Container Probes 的更多信息

管理集群中的TLS

在本书的最佳实践部分，我们在CentOS上部署了kubernetes集群，其中最开始有重要的一步就是创建TLS认证的，查看[创建TLS证书和秘钥](#)。很多人在进行到这一步时都会遇到各种各样千奇百怪的问题，这一步是创建集群的基础，我们有必要详细了解一下背后的流程和原理。

概览

每个Kubernetes集群都有一个集群根证书颁发机构（CA）。集群中的组件通常使用CA来验证API server的证书，由API服务器验证kubelet客户端证书等。为了支持这一点，CA证书包被分发到集群中的每个节点，并作为一个secret附加分发到默认service account上。或者，你的workload可以使用此CA建立信任。你的应用程序可以使用类似于[ACME草案](#)的协议，使用 `certificates.k8s.io` API请求证书签名。

集群中的TLS信任

让Pod中运行的应用程序信任集群根CA通常需要一些额外的应用程序配置。您将需要将CA证书包添加到TLS客户端或服务器信任的CA证书列表中。例如，您可以使用golang TLS配置通过解析证书链并将解析的证书添加到 `tls.Config` 结构中的 `Certificates` 字段中，CA证书捆绑包将使用默认服务账户自动加载到pod中，路径为 `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`。如果您没有使用默认服务账户，请请求集群管理员构建包含您有权访问使用的证书包的configmap。

请求认证

以下部分演示如何为通过DNS访问的Kubernetes服务创建TLS证书。

步骤0. 下载安装SSL

下载cfssl工具：<https://pkg.cfssl.org/>.

步骤1. 创建证书签名请求

通过运行以下命令生成私钥和证书签名请求（或CSR）：

```
$ cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "172.168.0.24",
    "10.0.34.2"
  ],
  "CN": "my-pod.my-namespace.pod.cluster.local",
  "key": {
    "algo": "ecdsa",
    "size": 256
  }
}
EOF
```

`172.168.0.24` 是service的cluster IP，`my-svc.my-namespace.svc.cluster.local` 是service的DNS名称，`10.0.34.2` 是Pod的IP，`my-pod.my-namespace.pod.cluster.local` 是pode的DNS名称，你可以看到以下输出：

```
2017/03/21 06:48:17 [INFO] generate received request
2017/03/21 06:48:17 [INFO] received CSR
2017/03/21 06:48:17 [INFO] generating key: ecdsa-256
2017/03/21 06:48:17 [INFO] encoded CSR
```

此命令生成两个文件；它生成包含PEM编码的[pkcs #10](#)认证请求的 `server.csr`，以及包含仍然要创建的证书的PEM编码密钥的 `server-key.pem`。

步骤2. 创建证书签名请求对象以发送到Kubernetes API

使用以下命令创建CSR yaml文件，并发送到API server：

```
$ cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: my-svc.my-namespace
spec:
  groups:
  - system:authenticated
  request: $(cat server.csr | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - server auth
EOF
```

请注意，在步骤1中创建的 `server.csr` 文件是base64编码并存储在 `.spec.request` 字段中。我们还要求提供“数字签名”，“密钥加密”和“服务器身份验证”密钥用途的证书。我们[这里](#)支持列出的所有关键用途和扩展的关键用途，以便您可以使用相同的API请求客户端证书和其他证书。

在API server中可以看到这些CSR处于pending状态。执行下面的命令你将可以看到：

```
$ kubectl describe csr my-svc.my-namespace
Name:           my-svc.my-namespace
Labels:         <none>
Annotations:   <none>
CreationTimestamp: Tue, 21 Mar 2017 07:03:51 -0700
Requesting User: yourname@example.com
Status:         Pending
Subject:
  Common Name:  my-svc.my-namespace.svc.cluster.local
  Serial Number:
Subject Alternative Names:
  DNS Names:   my-svc.my-namespace.svc.cluster.local
  IP Addresses: 172.168.0.24
                           10.0.34.2
Events: <none>
```

步骤3. 获取证书签名请求

批准证书签名请求是通过自动批准过程完成的，或由集群管理员一次完成。有关这方面涉及的更多信息，请参见下文。

步骤4. 下载签名并使用

一旦CSR被签署并获得批准，您应该看到以下内容：

```
$ kubectl get csr
NAME          AGE     REQUESTOR      CONDITION
N
my-svc.my-namespace   10m    yourname@example.com  Approved
,Issued
```

你可以通过运行以下命令下载颁发的证书并将其保存到 `server.crt` 文件：

```
$ kubectl get csr my-svc.my-namespace -o jsonpath='{.status.certificate}' \
| base64 -d > server.crt
```

现在你可以用 `server.crt` 和 `server-key.pem` 来做为keypair来启动HTTPS server。

给集群管理员的一个建议

本教程假设将signer设置为服务证书API。Kubernetes controller manager提供了一个signer的默认实现。要启用它，请将 `--cluster-signature-cert-file` 和 `--cluster-signing-key-file` 参数传递给controller manager，并配置具有证书颁发机构的密钥对的路径。

使用**kubectl**

Kubernetes提供的kubectl命令是与集群交互最直接的方式，v1.6版本的kubectl命令参考图如下：

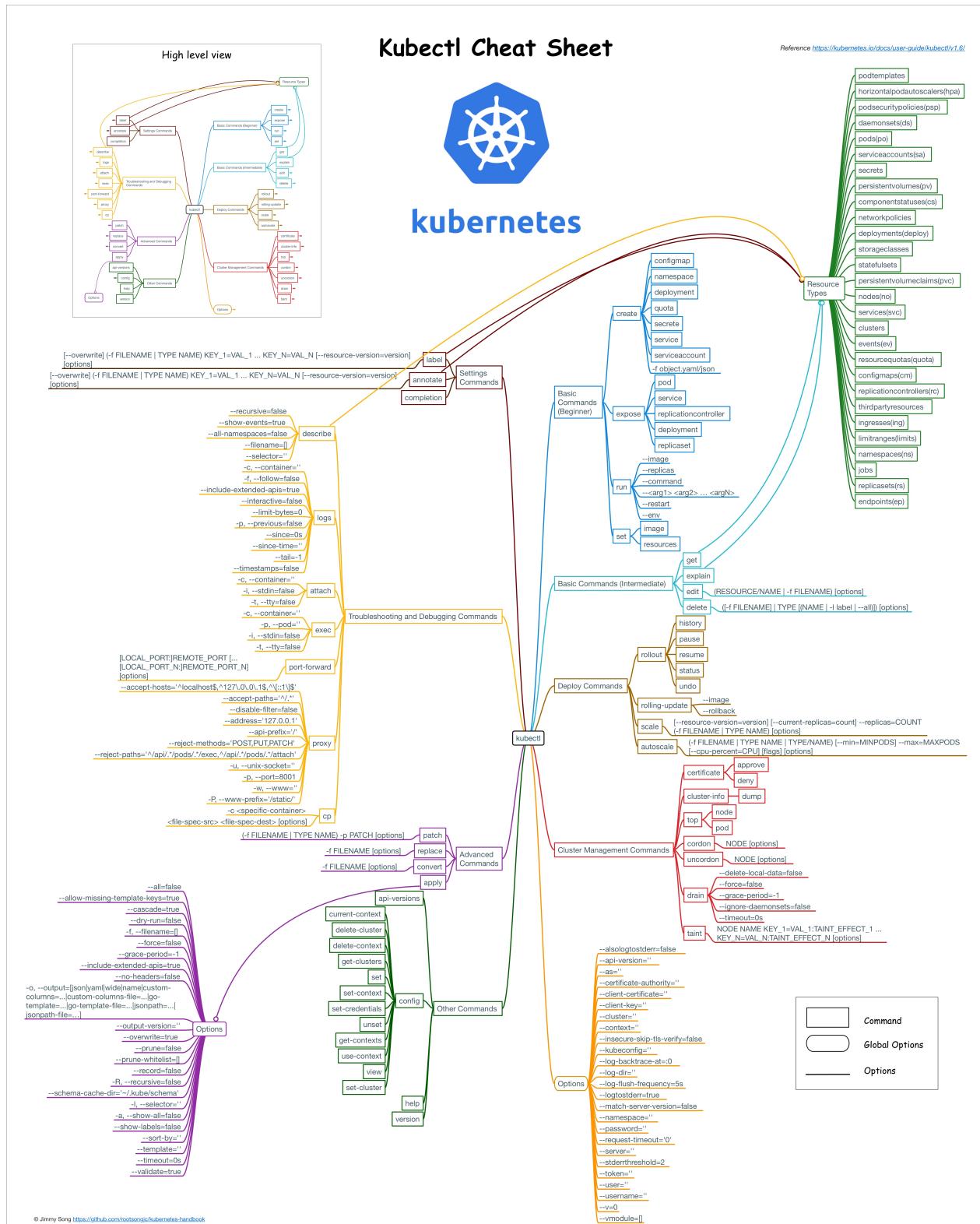


Figure: kubectl cheatsheet

Kubectl的子命令主要分为8个类别：

- #### • 基础命令（初学者都会使用的）

- 基础命令（中级）
- 部署命令
- 集群管理命令
- 故障排查和调试命令
- 高级命令
- 设置命令
- 其他命令

熟悉这些命令有助于大家来操作和管理kubernetes集群。

kube-shell

开源项目[kube-shell](#)可以为kubectl提供自动的命令提示和补全，使用起来特别方便，推荐给大家。

Kube-shell有以下特性：

- 命令提示，给出命令的使用说明
- 自动补全，列出可选命令并通过tab键自动补全，支持模糊搜索
- 高亮
- 使用tab键可以列出可选的对象
- vim模式

Mac下安装

```
pip install kube-shell --user -U
```

3.3 使用kubectl

```
|kube-shell> kubectl top node
NAME          CPU(cores)   CPU%     MEMORY(bytes)  MEMORY%
172.20.0.114  734m        1%      34540Mi       26%
172.20.0.115  308m        0%      9334Mi       7%
172.20.0.113  894m        2%      50624Mi      39%
|kube-shell> kubectl describe
  persistentvolumeclaim
  endpoints
  horizontalpodautoscaler
  job
  storageclass
  secret
  serviceaccount
  configmap
  deployment
  deployment
  statefulset
  poddisruptionbudget
  pod
  replicationcontroller
  resourcequota
  persistentvolume
Cluster: kubernetes Namespace: default User: admin [F4] In-line help: ON [F10] Exit
```

for GitBook

update 2017-08-07 13:54:27

适用于kubernetes的应用开发部署流程

为了讲解详细流程，我特意写了两个示例程序放在GitHub中，模拟监控流程：

- [k8s-app-monitor-test](#)：生成模拟的监控数据，发送http请求，获取json返回值
- [K8s-app-monitor-agent](#)：获取监控数据并绘图，访问浏览器获取图表

API文档见[k8s-app-monitor-test](#)中的 `api.html` 文件，该文档在API blueprint中定义，使用[aglio](#)生成，打开后如图所示：

The screenshot shows the API documentation for the 'Metrics' section of the 'Kubernetes app monitoring test'. The left sidebar has sections for 'Overview' and 'Metrics'. Under 'Metrics', there are dropdowns for 'Resource Group' (set to 'Metrics') and buttons for 'List All Metric' and 'Get the specific application...'. Below this is the URL <http://localhost:3000/>.

The main content area has a title 'Metrics' and a sub-section 'Resource Group'. It includes a 'METRICS COLLECTION' section with a 'GET /metrics' endpoint. The description says: 'The metric collection represents the status of the application.' It shows an example URI: `GET http://localhost:3000//metrics`, a 'Response 200' status, and a 'Show' button.

Below this is a 'GET SPECIFIC APPLICATION METRIC' section with a 'GET /metrics/{appname}' endpoint. The description says: 'Get the specific application's metric.' It shows an example URI: `GET http://localhost:3000//metrics/"Gateway_quota_request"`, a 'URI Parameters' table with 'appname' as a required string parameter, and 'Response 200' and 'Response 404' status buttons. There is also a 'Hide' button and 'Show' buttons for the parameters and responses.

At the bottom of the page is a footer: 'Generated by [aglio](#) on 18 Jul 2017'.

Figure: API

关于服务发现

K8s-app-monitor-agent 服务需要访问 k8s-app-monitor-test 服务，这就涉及到服务发现的问题，我们在代码中直接写死了要访问的服务的内网DNS地址（kubedns中的地址，即 k8s-app-monitor-test.default.svc.cluster.local ）。

我们知道Kubernetes在启动Pod的时候为容器注入环境变量，这些环境变量在所有的 namespace 中共享（环境变量是不断追加的，新启动的Pod中将拥有老的Pod中所有的环境变量，而老的Pod中的环境变量不变）。但是既然使用这些环境变量就已经可以访问到对应的service，那么获取应用的地址信息，究竟是使用变量呢？还是直接使用DNS解析来发现？

答案是使用DNS，详细说明见[Kubernetes中的服务发现与Docker容器间的环境变量传递源码探究](#)。

打包镜像

因为我使用wercker自动构建，构建完成后自动打包成docker镜像并上传到docker hub中（需要现在docker hub中创建repo）。

构建流程见：<https://app.wercker.com/jimmysong/k8s-app-monitor-agent/>

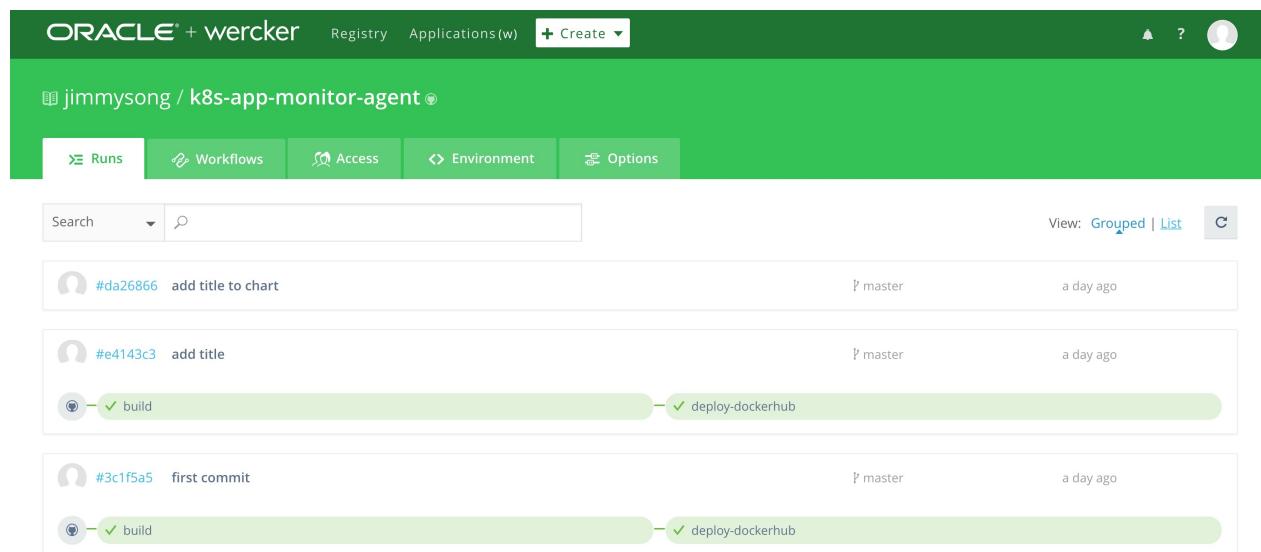


Figure: wercker

生成了如下两个docker镜像：

- `jimmysong/k8s-app-monitor-test:latest`
- `jimmysong/k8s-app-monitor-agent:latest`

启动服务

所有的kubernetes应用启动所用的yaml配置文件都保存在那两个GitHub仓库的 `manifest.yaml` 文件中。

分别在两个GitHub目录下执行 `kubectl create -f manifest.yaml` 即可启动服务。

外部访问

服务启动后需要更新ingress配置，在`ingress.yaml`文件中增加以下几行：

```
- host: k8s-app-monitor-agent.jimmysong.io
  http:
    paths:
      - path: /
        backend:
          serviceName: k8s-app-monitor-agent
          servicePort: 8080
```

保存后，然后执行 `kubectl replace -f ingress.yaml` 即可刷新ingress。

修改本机的 `/etc/hosts` 文件，在其中加入以下一行：

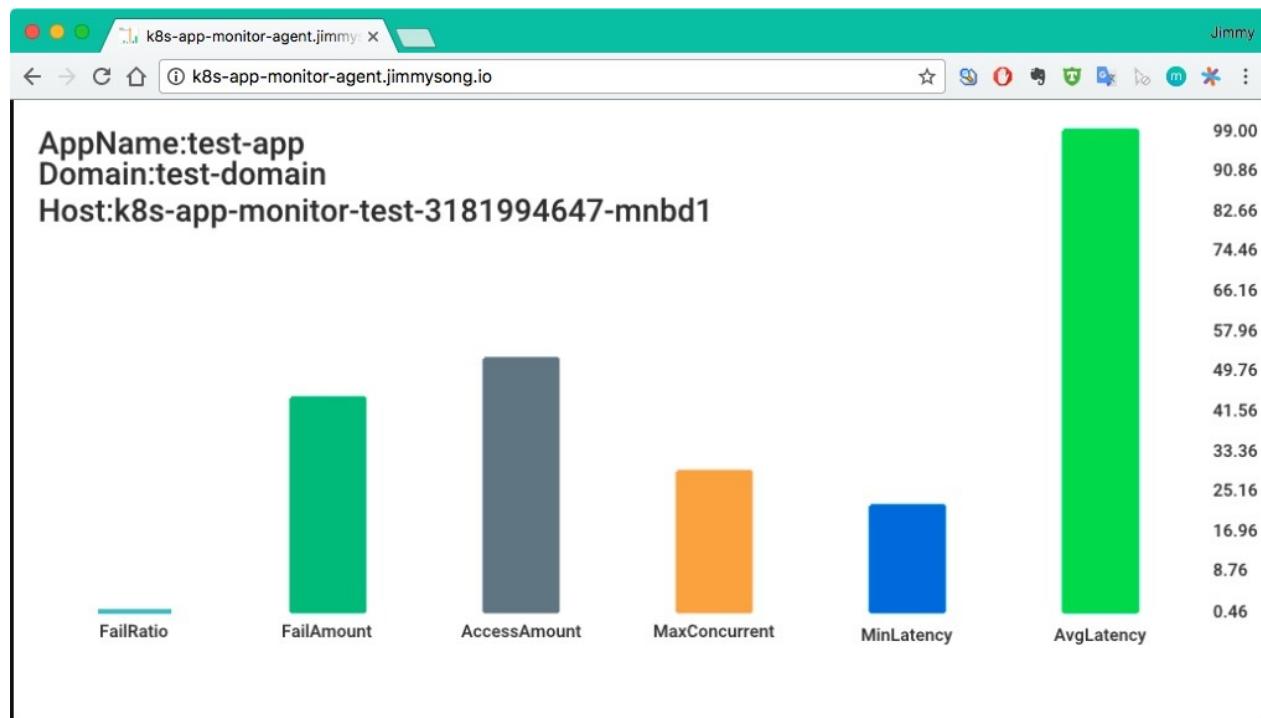
```
172.20.0.119 k8s-app-monitor-agent.jimmysong.io
```

当然你也可以加入到DNS中，为了简单起见我使用hosts。

详见[边缘节点配置](#)。

在浏览器中访问<http://k8s-app-monitor-agent.jimmysong.io>

3.4 适用于kubernetes的应用开发部署流程



for GitBook

update 2017-08-07 13:54:27

配置 Pod 的 Service Account

Service account 为 Pod 中的进程提供身份信息。

本文是关于 *Service Account* 的用户指南，管理指南另见 *Service Account* 的集群管理指南。

注意：本文档描述的关于 *Service Account* 的行为只有当您按照 *Kubernetes* 项目建议的方式搭建起集群的情况下才有效。您的集群管理员可能在您的集群中有自定义配置，这种情况下该文档可能并不适用。

当您（真人用户）访问集群（例如使用 `kubectl` 命令）时，`apiserver` 会将您认证为一个特定的 `User Account`（目前通常是 `admin`，除非您的系统管理员自定义了集群配置）。Pod 容器中的进程也可以与 `apiserver` 联系。当它们在联系 `apiserver` 的时候，它们会被认证为一个特定的 `Service Account`（例如 `default`）。

使用默认的 Service Account 访问 API server

当您创建 pod 的时候，如果您没有指定一个 service account，系统会自动得在与该pod 相同的 namespace 下为其指派一个 `default service account`。如果您获取刚创建的 pod 的原始 json 或 yaml 信息（例如使用 `kubectl get pods/podename -o yaml` 命令），您将看到 `spec.serviceAccountName` 字段已经被设置为 `automatically set`。

您可以在 pod 中使用自动挂载的 service account 凭证来访问 API，如 [Accessing the Cluster](#) 中所描述。

Service account 是否能够取得访问 API 的许可取决于您使用的 [授权插件和策略](#)。

在 1.6 以上版本中，您可以选择取消为 service account 自动挂载 API 凭证，只需在 service account 中设置 `automountServiceAccountToken: false`：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
...
```

在 1.6 以上版本中，您也可以选择只取消单个 pod 的 API 凭证自动挂载：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
  automountServiceAccountToken: false
...
```

如果在 pod 和 service account 中同时设置了 `automountServiceAccountToken`，pod 设置中的优先级更高。

使用多个Service Account

每个 namespace 中都有一个默认的叫做 `default` 的 service account 资源。

您可以使用以下命令列出 namespace 下的所有 serviceAccount 资源。

```
$ kubectl get serviceAccounts
NAME      SECRETS   AGE
default    1          1d
```

您可以像这样创建一个 ServiceAccount 对象：

```
$ cat > /tmp/serviceaccount.yaml <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
$ kubectl create -f /tmp/serviceaccount.yaml
serviceaccount "build-robot" created
```

如果您看到如下的 service account 对象的完整输出信息：

```
$ kubectl get serviceaccounts/build-robot -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  selfLink: /api/v1/namespaces/default/serviceaccounts/build-robot
  uid: 721ab723-13bc-11e5-aec2-42010af0021e
secrets:
- name: build-robot-token-bvbk5
```

然后您将看到有一个 token 已经被自动创建，并被 service account 引用。

您可以使用授权插件来 [设置 service account 的权限](#)。

设置非默认的 service account，只需要在 pod 的 `spec.serviceAccountName` 字段中将 `name` 设置为您想要用的 service account 名字即可。

在 pod 创建之初 service account 就必须已经存在，否则创建将被拒绝。

您不能更新已创建的 pod 的 service account。

您可以清理 service account，如下所示：

```
$ kubectl delete serviceaccount/build-robot
```

手动创建 service account 的 API token

假设我们已经有了一个如上文提到的名为 "build-robot" 的 service account，我们手动创建一个新的 secret。

```
$ cat > /tmp/build-robot-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
$ kubectl create -f /tmp/build-robot-secret.yaml
secret "build-robot-secret" created
```

现在您可以确认下新创建的 secret 取代了 "build-robot" 这个 service account 原来的 API token。

所有已不存在的 service account 的 token 将被 token controller 清理掉。

```
$ kubectl describe secrets/build-robot-secret
Name:      build-robot-secret
Namespace:  default
Labels:     <none>
Annotations: kubernetes.io/service-account.name=build-robot, kubernetes.io/service-account.uid=870ef2a5-35cf-11e5-8d06-005056b45392

Type:      kubernetes.io/service-account-token

Data
=====
ca.crt: 1220 bytes
token: ...
namespace: 7 bytes
```

注意该内容中的 token 被省略了。

为 service account 添加 ImagePullSecret

首先，创建一个 imagePullSecret，详见[这里](#)。

然后，确认已创建。如：

```
$ kubectl get secrets myregistrykey
NAME          TYPE
myregistrykey  kubernetes.io/.dockerconfigjson
                DATA   AGE
                  1      1d
```

然后，修改 namespace 中的默认 service account 使用该 secret 作为 imagePullSecret。

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}'
```

Vi 交互过程中需要手动编辑：

```

$ kubectl get serviceaccounts default -o yaml > ./sa.yaml
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
$ vi sa.yaml
[editor session not shown]
[delete line with key "resourceVersion"]
[add lines with "imagePullSecret:"]
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
$ kubectl replace serviceaccount default -f ./sa.yaml
serviceaccounts/default

```

现在，所有当前 namespace 中新创建的 pod 的 spec 中都会增加如下内容：

```

spec:
  imagePullSecrets:
    - name: myregistrykey

```

for GitBook update 2017-08-07 13:54:27

访问集群

第一次使用 **kubectl** 访问

如果您是第一次访问 Kubernetes API 的话，我们建议您使用 Kubernetes 命令行工具：`kubectl`。

为了访问集群，您需要知道集群的地址，并且需要有访问它的凭证。通常，如果您完成了[入门指南](#)那么这些将会自动设置，或者其他人为您部署的集群提供并给您凭证和集群地址。

使用下面的命令检查 `kubectl` 已知的集群的地址和凭证：

```
$ kubectl config view
```

关于 `kubectl` 命令使用的更多 [示例](#) 和完整文档可以在这里找到：[kubectl 手册](#)

直接访问 REST API

Kubectl 处理对 `apiserver` 的定位和认证。如果您想直接访问 REST API，可以使用像 `curl`、`wget` 或浏览器这样的 `http` 客户端，有以下几种方式来定位和认证：

- 以 `proxy` 模式运行 `kubectl`。
 - 推荐方法。
 - 使用已保存的 `apiserver` 位置信息。
 - 使用自签名证书验证 `apiserver` 的身份。没有 MITM（中间人攻击）的可能。
 - 认证到 `apiserver`。
 - 将来，可能会做智能的客户端负载均衡和故障转移。
- 直接向 `http` 客户端提供位置和凭据。
 - 替代方法。
 - 适用于通过使用代理而混淆的某些类型的客户端代码。
 - 需要将根证书导入浏览器以防止 MITM。

使用 **kubectl proxy**

以下命令作为反向代理的模式运行 `kubectl proxy`。它处理对 `apiserver` 的定位并进行认证。

像这样运行：

```
$ kubectl proxy --port=8080 &
```

查看关于 [kubectl proxy](#) 的更多细节。

然后您可以使用 `curl`、`wget` 或者浏览器来访问 API，如下所示：

```
$ curl http://localhost:8080/api/
{
  "versions": [
    "v1"
  ]
}
```

不使用 `kubectl proxy` (1.3.x 以前版本)

通过将认证 token 直接传递给 `apiserver` 的方式，可以避免使用 `kubectl proxy`，如下所示：

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d " "
  | tr -d "\n")
$ TOKEN=$(kubectl config view | grep token | cut -f 2 -d ":" | t
r -d "\n")
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --
insecure
{
  "versions": [
    "v1"
  ]
}
```

不使用 `kubectl proxy` (1.3.x 以后版本)

在 Kubernetes 1.3 或更高版本中，`kubectl config view` 不再显示 token。使用 `kubectl describe secret ...` 获取 default service account 的 token，如下所示：

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d "\n")
$ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default | cut -f1 -d ' ') | grep -E '^token' | cut -f2 -d ':' | tr -d '\t')
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

以上示例使用 `--insecure` 标志。这使得它容易受到 MITM 攻击。当 `kubectl` 访问集群时，它使用存储的根证书和客户端证书来访问服务器。（这些安装在 `~/.kube` 目录中）。由于集群证书通常是自签名的，因此可能需要特殊配置才能让您的 http 客户端使用根证书。

对于某些群集，`apiserver` 可能不需要身份验证；可以选择在本地主机上服务，或者使用防火墙保护。对此还没有一个标准。[配置对API的访问](#) 描述了群集管理员如何配置此操作。这种方法可能与未来的高可用性支持相冲突。

编程访问 API

Kubernetes 支持 [Go](#) 和 [Python](#) 客户端库。

Go 客户端

- 要获取该库，请运行以下命令：`go get k8s.io/client-go/<version number>/kubernetes` 请参阅 <https://github.com/kubernetes/client-go> 以查看支持哪些版本。
- 使用 `client-go` 客户端编程。请注意，`client-go` 定义了自己的 API 对象，因此如果需要，请从 `client-go` 而不是从主存储库导入 API 定义，例如导入 `k8s.io/client-go/1.4/pkg/api/v1` 是正确的。

Go 客户端可以使用与 `kubectl` 命令行工具相同的 `kubeconfig` 文件 来定位和验证 `apiserver`。参考官方 [示例](#) 和 [client-go 示例](#)。

如果应用程序在群集中以 Pod 的形式部署，请参考 [下一节](#)。

Python 客户端

要使用 `Python client`，请运行以下命令：`pip install kubernetes`。查看 [Python 客户端库页面](#) 获取更多的安装选择。

Python 客户端可以使用与 `kubectl` 命令行工具相同的 `kubeconfig` 文件 来定位和验证 `apiserver`。参考该 [示例](#)。

其他语言

还有更多的 [客户端库](#) 可以用来访问 API。有关其他库的验证方式，请参阅文档。

在 Pod 中访问 API

在 Pod 中访问 API 时，定位和认证到 API server 的方式有所不同。在 Pod 中找到 `apiserver` 地址的推荐方法是使用 Kubernetes DNS 名称，将它解析为服务 IP，后者又将被路由到 `apiserver`。

向 `apiserver` 认证的推荐方法是使用 `service account` 凭据。通过 `kube-system`，pod 与 `service account` 相关联，并且将该 `service account` 的凭据（token）放入该 pod 中每个容器的文件系统树中，位于

`/var/run/secrets/kubernetes.io/serviceaccount/token`。

如果可用，证书包将位于每个容器的文件系统树的

`/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` 位置，并用于验证 `apiserver` 的服务证书。

最后，用于 `namespace API` 操作的默认 `namespace` 放在每个容器中的

`/var/run/secrets/kubernetes.io/serviceaccount/namespace` 中。

在 `pod` 中，连接到 `API` 的推荐方法是：

- 将 `kubectl proxy` 作为 `pod` 中的一个容器来运行，或作为在容器内运行的后台进程。它将 `Kubernetes API` 代理到 `pod` 的本地主机接口，以便其他任何 `pod` 中的容器内的进程都可以访问它。请参阅 [在 pod 中使用 kubectl proxy 的示例](#)。
- 使用 `Go` 客户端库，并使用 `rest.InClusterConfig()` 和 `kubernetes.NewForConfig()` 函数创建一个客户端。

他们处理对 `apiserver` 的定位和认证。[示例](#)

在以上的几种情况下，都需要使用 `pod` 的凭据与 `apiserver` 进行安全通信。

访问集群中运行的 `service`

上一节是关于连接到 `kubernetes API server`。这一节是关于连接到 `kubernetes` 集群中运行的 `service`。在 `Kubernetes` 中，`node`、`pod` 和 `services` 都有它们自己的 IP。很多情况下，集群中 `node` 的 IP、`Pod` 的 IP、`service` 的 IP 都是不可路由的，因此在集群外面的机器就无法访问到它们，例如从您自己的笔记本电脑。

连接的方式

您可以选择以下几种方式从集群外部连接到 `node`、`pod` 和 `service`：

- 通过 `public IP` 访问 `service`。
 - 使用 `NodePort` 和 `LoadBalancer` 类型的 `service`，以使 `service` 能够在集群外部被访问到。请查看 `service` 和 `kubectl expose` 文档。
 - 根据您的群集环境，这可能会将服务暴露给您的公司网络，或者可能会将其暴露在互联网上。想想暴露的服务是否安全。它是否自己进行身份验证？
 - 将 `pod` 放在服务后面。要从一组副本（例如为了调试）访问一个特定的

pod，请在 pod 上放置一个唯一的 label，并创建一个选择该 label 的新服务。

- 在大多数情况下，应用程序开发人员不需要通过 node IP 直接访问节点。
- 通过 Proxy 规则访问 service、node、pod。
 - 在访问远程服务之前，请执行 apiserver 认证和授权。如果服务不够安全，无法暴露给互联网，或者为了访问节点 IP 上的端口或进行调试，请使用这种方式。
 - 代理可能会导致某些 Web 应用程序出现问题。
 - 仅适用于 HTTP/HTTPS。
 - [见此描述](#)。
- 在集群内访问 node 和 pod。
 - 运行一个 pod，然后使用 `kubectl exec` 命令连接到 shell。从该 shell 中连接到其他 node、pod 和 service。
 - 有些集群可能允许 ssh 到集群上的某个节点。从那个节点您可以访问到集群中的服务。这是一个非标准的方法，它可能将在某些集群上奏效，而在某些集群不行。这些节点上可能安装了浏览器和其他工具也可能没有。集群 DNS 可能无法正常工作。

访问内置服务

通常集群内会有几个在 kube-system 中启动的服务。使用 `kubectl cluster-info` 命令获取该列表：

```
$ kubectl cluster-info

Kubernetes master is running at https://104.197.5.247
elasticsearch-logging is running at https://104.197.5.247/api/
v1/namespaces/kube-system/services/elasticsearch-logging/proxy
kibana-logging is running at https://104.197.5.247/api/v1/name
spaces/kube-system/services/kibana-logging/proxy
kube-dns is running at https://104.197.5.247/api/v1/namespaces/
/kube-system/services/kube-dns/proxy
grafana is running at https://104.197.5.247/api/v1/namespaces/
kube-system/services/monitoring-grafana/proxy
heapster is running at https://104.197.5.247/api/v1/namespaces/
/kube-system/services/monitoring-heapster/proxy
```

这显示了访问每个服务的代理 URL。

例如，此集群启用了集群级日志记录（使用 Elasticsearch），如果传入合适的凭据，可以在该地址 `https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/` 访问到，或通过 kubectl 代理，例如：`http://localhost:8080/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/`。

（有关如何传递凭据和使用 kubectl 代理，请 [参阅上文](#)）

手动构建 apiserver 代理 URL

如上所述，您可以使用 `kubectl cluster-info` 命令来检索服务的代理 URL。要创建包含服务端点、后缀和参数的代理 URL，您只需附加到服务的代理 URL：

```
http:// kubernetes_master_address /api/v1/namespaces/ namespace_name/services/ service_name[:port_name] /proxy
```

如果您没有指定 port 的名字，那么您不必在 URL 里指定 port_name。

示例

- 要想访问 Elasticsearch 的服务端点 `_search?q=user:kimchy`，您需要使用：`http://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_search?q=user:kimchy`
- 要想访问 Elasticsearch 的集群健康信息 `_cluster/health?pretty=true`，您需要使用：`https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_cluster/health?pretty=true`

```
{
  "cluster_name" : "kubernetes_logging",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 5,
  "active_shards" : 5,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 5
}
```

使用 web 浏览器来访问集群中运行的服务

您可以将 `apiserver` 代理网址放在浏览器的地址栏中。然而：

- Web 浏览器通常不能传递 `token`，因此您可能需要使用基本（密码）认证。
`Apiserver` 可以配置为接受基本认证，但您的集群可能未配置为接受基本认证。
- 某些网络应用程序可能无法正常工作，特别是那些在不知道代理路径前缀的情况下构造 URL 的客户端 JavaScript。

请求重定向

重定向功能已被弃用和删除。请改用代理（见下文）。

多种代理

在使用 `kubernetes` 的时候您可能会遇到许多种不同的代理：

1. `kubectl` 代理：

- 在用户桌面或 `pod` 中运行
- 从 `localhost` 地址到 `Kubernetes apiserver` 的代理
- 客户端到代理使用 `HTTP`
- `apiserver` 的代理使用 `HTTPS`
- 定位 `apiserver`

- 添加身份验证 header

2. apiserver 代理：

- 将一个堡垒机作为 apiserver
- 将群集之外的用户连接到群集IP，否则可能无法访问
- 在 apiserver 进程中运行
- 客户端到代理使用 HTTPS（或 http，如果 apiserver 如此配置）
- 根据代理目标的可用信息由代理选择使用 HTTP 或 HTTPS
- 可用于访问 node、pod 或 service
- 用于访问 service 时进行负载均衡

3. kube 代理：

- 在每个节点上运行
- 代理 UDP 和 TCP
- 不支持 HTTP
- 提供负载均衡
- 只是用来访问 service

4. apiserver 前面的代理/负载均衡器：

- 存在和实现因群集而异（例如 nginx）
- 位于所有客户端和一个或多个 apiserver 之间
- 作为负载均衡器，如果有多个 apiserver

5. 外部服务的云负载均衡器：

- 由一些云提供商提供（例如 AWS ELB，Google Cloud Load Balancer）
- 当 Kubernetes service 类型为 LoadBalancer 时，会自动创建
- 仅使用 UDP/TCP
- 实施方式因云提供商而异

最佳实践

本章节从零开始创建我们自己的kubernetes集群，并在该集群的基础上，配置服务发现、负载均衡和日志收集等功能，使我们的集群能够成为一个真正线上可用、功能完整的集群。

第一部分 [在CentOS上部署kubernetes1.6集群](#) 中介绍了如何通过二进制文件在 CentOS物理机上快速部署一个kubernetes集群，而[安装EFK插件](#)是官方提供的一种日志收集方案，不一定适用于我们的业务，在此仅是介绍，并没有在实际生产中应用，后面的运维管理部分有详细的[应用日志收集方案](#)介绍。

for GitBook update 2017-08-07 13:54:27

在CentOS上部署kubernetes1.6集群

说明：本安装文档在[opsnull](#)文档的基础上修改、整理而成。

本系列文档介绍使用二进制部署 kubernetes 集群的所有步骤，而不是使用 `kubeadm` 等自动化方式来部署集群，同时开启了集群的TLS安全认证；

在部署的过程中，将详细列出各组件的启动参数，给出配置文件，详解它们的含义和可能遇到的问题。

部署完成后，你将理解系统各组件的交互原理，进而能快速解决实际问题。

所以本文档主要适合于那些有一定 kubernetes 基础，想通过一步步部署的方式来学习和了解系统配置、运行原理的人。

注：本文档中不包括[docker](#)和私有镜像仓库的安装。

提供所有的配置文件

集群安装时所有组件用到的配置文件，包含在以下目录中：

- **etc** : service的环境变量配置文件
- **manifest** : kubernetes应用的yaml文件
- **systemd** : systemd serivce配置文件

集群详情

- Kubernetes 1.6.0
- Docker 1.12.5 (使用yum安装)
- Etcd 3.1.5
- Flanneld 0.7 vxlan 网络
- TLS 认证通信 (所有组件，如 etcd、kubernetes master 和 node)
- RBAC 授权
- kublet TLS BootStrapping
- kubedns、dashboard、heapster(influxdb、grafana)、EFK(elasticsearch、fluentd、kibana) 集群插件

- 私有docker镜像仓库[harbor](#)（请自行部署，harbor提供离线安装包，直接使用[docker-compose](#)启动即可）

环境说明

在下面的步骤中，我们将在三台CentOS系统的物理机上部署具有三个节点的kubernetes1.6.0集群。

角色分配如下：

Master : 172.20.0.113

Node : 172.20.0.113、172.20.0.114、172.20.0.115

注意：172.20.0.113这台主机master和node复用。所有生成证书、执行[kubectl](#)命令的操作都在这台节点上执行。一旦node加入到kubernetes集群之后就不再需要再登陆node节点了。

安装前的准备

1. 在node节点上安装[docker1.12.5](#)
2. 关闭所有节点的SELinux
3. 准备[harbor](#)私有镜像仓库

步骤介绍

- 1 创建 TLS 证书和秘钥
- 2 创建[kubeconfig](#) 文件
- 3 创建高可用etcd集群
- 4 安装[kubectl](#)命令行工具
- 5 部署高可用master集群
- 6 部署node节点
- 7 安装[kubedns](#)插件
- 8 安装[dashboard](#)插件
- 9 安装[heapster](#)插件
- 10 安装[EFK](#)插件

提醒

1. 由于启用了 TLS 双向认证、RBAC 授权等严格的安全机制，建议从头开始部署，而不要从中间开始，否则可能会认证、授权等失败！
2. 本文档将随着各组件的更新而更新，有任何问题欢迎提 issue！

关于

[Jimmy Song](#)

for GitBook update 2017-08-07 13:54:27

创建**TLS**证书和秘钥

`kubernetes` 系统的各组件需要使用 `TLS` 证书对通信进行加密，本文档使用 `CloudFlare` 的 `PKI` 工具集 `cfssl` 来生成 `Certificate Authority (CA)` 和其它证书；

生成的 **CA** 证书和秘钥文件如下：

- `ca-key.pem`
- `ca.pem`
- `kubernetes-key.pem`
- `kubernetes.pem`
- `kube-proxy.pem`
- `kube-proxy-key.pem`
- `admin.pem`
- `admin-key.pem`

使用证书的组件如下：

- `etcd`：使用 `ca.pem`、`kubernetes-key.pem`、`kubernetes.pem`；
- `kube-apiserver`：使用 `ca.pem`、`kubernetes-key.pem`、`kubernetes.pem`；
- `kubelet`：使用 `ca.pem`；
- `kube-proxy`：使用 `ca.pem`、`kube-proxy-key.pem`、`kube-proxy.pem`；
- `kubectl`：使用 `ca.pem`、`admin-key.pem`、`admin.pem`；

`kube-controller` 、 `kube-scheduler` 当前需要和 `kube-apiserver` 部署在同一台机器上且使用非安全端口通信，故不需要证书。

安装 **CFSSL**

方式一：直接使用二进制源码包安装

```
$ wget https://pkg.cfssl.org/R1.2/cfssl_linux-amd64  
$ chmod +x cfssl_linux-amd64  
$ sudo mv cfssl_linux-amd64 /root/local/bin/cfssl  
  
$ wget https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64  
$ chmod +x cfssljson_linux-amd64  
$ sudo mv cfssljson_linux-amd64 /root/local/bin/cfssljson  
  
$ wget https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64  
$ chmod +x cfssl-certinfo_linux-amd64  
$ sudo mv cfssl-certinfo_linux-amd64 /root/local/bin/cfssl-certinfo  
  
$ export PATH=/root/local/bin:$PATH
```

方式二：使用go命令安装

我们的系统中安装了Go1.7.5，使用以下命令安装更快捷：

```
$go get -u github.com/cloudflare/cfssl/cmd/...  
$echo $GOPATH  
/usr/local  
$ls /usr/local/bin/cfssl*  
cfssl cfssl-bundle cfssl-certinfo cfssljson cfssl-newkey cfssl-scan
```

在 `$GOPATH/bin` 目录下得到以`cfssl`开头的几个命令。

注意：以下文章中出现的`cat`的文件名如果不存在需要手工创建。

创建 CA (Certificate Authority)

创建 CA 配置文件

```

$ mkdir /root/ssl
$ cd /root/ssl
$ cfssl print-defaults config > config.json
$ cfssl print-defaults csr > csr.json
$ cat ca-config.json
{
  "signing": {
    "default": {
      "expiry": "87600h"
    },
    "profiles": {
      "kubernetes": {
        "usages": [
          "signing",
          "key encipherment",
          "server auth",
          "client auth"
        ],
        "expiry": "87600h"
      }
    }
  }
}

```

字段说明

- `ca-config.json` : 可以定义多个 `profiles`，分别指定不同的过期时间、使用场景等参数；后续在签名证书时使用某个 `profile`；
- `signing` : 表示该证书可用于签名其它证书；生成的 `ca.pem` 证书中 `CA=TRUE`；
- `server auth` : 表示client可以用该 CA 对server提供的证书进行验证；
- `client auth` : 表示server可以用该CA对client提供的证书进行验证；

创建**CA**证书签名请求

```
$ cat ca-csr.json
{
  "CN": "kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "k8s",
      "OU": "System"
    }
  ]
}
```

- "CN" : Common Name , kube-apiserver 从证书中提取该字段作为请求的用户名 (User Name)；浏览器使用该字段验证网站是否合法；
- "O" : Organization , kube-apiserver 从证书中提取该字段作为请求用户所属的组 (Group)；

生成 **CA** 证书和私钥

```
$ cfssl gencert -initca ca-csr.json | cfssljson -bare ca
$ ls ca*
ca-config.json  ca.csr  ca-csr.json  ca-key.pem  ca.pem
```

创建 **kubernetes** 证书

创建 **kubernetes** 证书签名请求

```
$ cat kubernetes-csr.json
{
    "CN": "kubernetes",
    "hosts": [
        "127.0.0.1",
        "172.20.0.112",
        "172.20.0.113",
        "172.20.0.114",
        "172.20.0.115",
        "10.254.0.1",
        "kubernetes",
        "kubernetes.default",
        "kubernetes.default.svc",
        "kubernetes.default.svc.cluster",
        "kubernetes.default.svc.cluster.local"
    ],
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
        {
            "C": "CN",
            "ST": "BeiJing",
            "L": "BeiJing",
            "O": "k8s",
            "OU": "System"
        }
    ]
}
```

- 如果 hosts 字段不为空则需要指定授权使用该证书的 IP 或域名列表，由于该证书后续被 etcd 集群和 kubernetes master 集群使用，所以上面分别指定了 etcd 集群、kubernetes master 集群的主机 IP 和 kubernetes 服务的服务 IP（一般是 kube-apiserver 指定的 service-cluster-ip-range 网段的第一个IP，如 10.254.0.1）。

生成 **kubernetes** 证书和私钥

```
$ cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -profile=kubernetes kubernetes-csr.json | cfssljson -bare kubernetes
$ ls kubernetes*
kubernetes.csr  kubernetes-csr.json  kubernetes-key.pem  kubernetes.pem
```

或者直接在命令行上指定相关参数：

```
$ echo '{"CN":"kubernetes","hosts":[],"key":{"algo":"rsa","size":2048}}' | cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -profile=kubernetes -hostname="127.0.0.1,172.20.0.112,172.20.0.113,172.20.0.114,172.20.0.115,kubernetes,kubernetes.default" - | cfssljson -bare kubernetes
```

创建 admin 证书

创建 admin 证书签名请求

```
$ cat admin-csr.json
{
  "CN": "admin",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "system:masters",
      "OU": "System"
    }
  ]
}
```

- 后续 `kube-apiserver` 使用 `RBAC` 对客户端(如 `kubelet`、`kube-proxy`、`Pod`)请求进行授权；
- `kube-apiserver` 预定义了一些 `RBAC` 使用的 `RoleBindings`，如 `cluster-admin` 将 `Group system:masters` 与 `Role cluster-admin` 绑定，该 `Role` 授予了调用 `kube-apiserver` 的所有 `API`的权限；
- `OU` 指定该证书的 `Group` 为 `system:masters`，`kubelet` 使用该证书访问 `kube-apiserver` 时，由于证书被 `CA` 签名，所以认证通过，同时由于证书用户组为经过预授权的 `system:masters`，所以被授予访问所有 `API` 的权限；

生成 `admin` 证书和私钥

```
$ cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -profile=kubernetes admin-csr.json | cfssljson -bare admin
$ ls admin*
admin.csr  admin-csr.json  admin-key.pem  admin.pem
```

创建 `kube-proxy` 证书

创建 kube-proxy 证书签名请求

```
$ cat kube-proxy-csr.json
{
  "CN": "system:kube-proxy",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "k8s",
      "OU": "System"
    }
  ]
}
```

- CN 指定该证书的 User 为 system:kube-proxy ；
- kube-apiserver 预定义的 RoleBinding cluster-admin 将 User system:kube-proxy 与 Role system:node-proxier 绑定，该 Role 授予了调用 kube-apiserver Proxy 相关 API 的权限；

生成 kube-proxy 客户端证书和私钥

```
$ cfssl gencert -ca=ca.pem -ca-key=ca-key.pem -config=ca-config.json -profile=kubernetes kube-proxy-csr.json | cfssljson -bare kube-proxy
$ ls kube-proxy*
kube-proxy.csr  kube-proxy-csr.json  kube-proxy-key.pem  kube-proxy.pem
```

校验证书

以 kubernetes 证书为例

使用 **openssl** 命令

```
$ openssl x509 -noout -text -in kubernetes.pem
...
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=CN, ST=BeiJing, L=BeiJing, O=k8s, OU=System, CN=Kubernetes
    Validity
        Not Before: Apr 5 05:36:00 2017 GMT
        Not After : Apr 5 05:36:00 2018 GMT
        Subject: C=CN, ST=BeiJing, L=BeiJing, O=k8s, OU=System, CN=kubernetes
    ...
    X509v3 extensions:
        X509v3 Key Usage: critical
            Digital Signature, Key Encipherment
        X509v3 Extended Key Usage:
            TLS Web Server Authentication, TLS Web Client Authentication
        X509v3 Basic Constraints: critical
            CA:FALSE
        X509v3 Subject Key Identifier:
            DD:52:04:43:10:13:A9:29:24:17:3A:0E:D7:14:DB:36:F8:6C:E0:E0
        X509v3 Authority Key Identifier:
            keyid:44:04:3B:60:BD:69:78:14:68:AF:A0:41:13:F6:17:07:13:63:58:CD

        X509v3 Subject Alternative Name:
            DNS:kubernetes, DNS:kubernetes.default, DNS:kubernetes.default.svc, DNS:kubernetes.default.svc.cluster, DNS:kubernetes.default.svc.cluster.local, IP Address:127.0.0.1, IP Address:172.20.0.112, IP Address:172.20.0.113, IP Address:172.20.0.114, IP Address:172.20.0.115, IP Address:10.254.0.1
    ...

```

- 确认 `Issuer` 字段的内容和 `ca-csr.json` 一致；
- 确认 `Subject` 字段的内容和 `kubernetes-csr.json` 一致；
- 确认 `X509v3 Subject Alternative Name` 字段的内容和 `kubernetes-`

- csr.json 一致；
- 确认 X509v3 Key Usage、Extended Key Usage 字段的内容和 config.json 中 kubernetes profile 一致；

使用 cfssl-certinfo 命令

```
$ cfssl-certinfo -cert kubernetes.pem
...
{
  "subject": {
    "common_name": "kubernetes",
    "country": "CN",
    "organization": "k8s",
    "organizational_unit": "System",
    "locality": "BeiJing",
    "province": "BeiJing",
    "names": [
      "CN",
      "BeiJing",
      "BeiJing",
      "k8s",
      "System",
      "kubernetes"
    ]
  },
  "issuer": {
    "common_name": "Kubernetes",
    "country": "CN",
    "organization": "k8s",
    "organizational_unit": "System",
    "locality": "BeiJing",
    "province": "BeiJing",
    "names": [
      "CN",
      "BeiJing",
      "BeiJing",
      "k8s",
      "System",
      "Kubernetes"
    ]
  }
}
```

```
        ],
    },
    "serial_number": "17436049287242326347315197163229289570712902
2309",
    "sans": [
        "kubernetes",
        "kubernetes.default",
        "kubernetes.default.svc",
        "kubernetes.default.svc.cluster",
        "kubernetes.default.svc.cluster.local",
        "127.0.0.1",
        "10.64.3.7",
        "10.254.0.1"
    ],
    "not_before": "2017-04-05T05:36:00Z",
    "not_after": "2018-04-05T05:36:00Z",
    "sigalg": "SHA256WithRSA",
    ...

```

分发证书

将生成的证书和秘钥文件（后缀名为 `.pem`）拷贝到所有机器的
`/etc/kubernetes/ssl` 目录下备用；

```
$ sudo mkdir -p /etc/kubernetes/ssl
$ sudo cp *.pem /etc/kubernetes/ssl
```

参考

- [Generate self-signed certificates](#)
- [Setting up a Certificate Authority and Creating TLS Certificates](#)
- [Client Certificates V/s Server Certificates](#)
- [数字证书及 CA 的扫盲介绍](#)

创建 kubeconfig 文件

注意：请先参考[安装kubectl命令行工具](#)，先在 master 节点上安装 kubectl 然后再进行下面的操作。

`kubelet`、`kube-proxy` 等 Node 机器上的进程与 Master 机器的 `kube-apiserver` 进程通信时需要认证和授权；

kubernetes 1.4 开始支持由 `kube-apiserver` 为客户端生成 TLS 证书的 [TLS Bootstrapping](#) 功能，这样就不需要为每个客户端生成证书了；该功能当前仅支持为 `kubelet` 生成证书；

因为我的master节点和node节点复用，所有在这一步其实已经安装了kubectl。参考[安装kubectl命令行工具](#)。

以下操作只需要在master节点上执行，生成的 `*.kubeconfig` 文件可以直接拷贝到node节点的 `/etc/kubernetes` 目录下。

创建 TLS Bootstrapping Token

Token auth file

Token可以是任意的包涵128 bit的字符串，可以使用安全的随机数发生器生成。

```
export BOOTSTRAP_TOKEN=$(head -c 16 /dev/urandom | od -An -t x |
  tr -d ' ')
cat > token.csv <<EOF
${BOOTSTRAP_TOKEN},kubelet-bootstrap,10001,"system:kubelet-boots
trap"
EOF
```

后三行是一句，直接复制上面的脚本运行即可。

BOOTSTRAP_TOKEN 将被写入到 `kube-apiserver` 使用的 `token.csv` 文件和 `kubelet` 使用的 `bootstrap.kubeconfig` 文件，如果后续重新生成了 `BOOTSTRAP_TOKEN`，则需要：

1. 更新 token.csv 文件，分发到所有机器 (master 和 node) 的 /etc/kubernetes/ 目录下，分发到node节点上非必需；
2. 重新生成 bootstrap.kubeconfig 文件，分发到所有 node 机器的 /etc/kubernetes/ 目录下；
3. 重启 kube-apiserver 和 kubelet 进程；
4. 重新 approve kubelet 的 csr 请求；

```
$cp token.csv /etc/kubernetes/
```

创建 kubelet bootstrapping kubeconfig 文件

```
$ cd /etc/kubernetes
$ export KUBE_APISERVER="https://172.20.0.113:6443"
$ # 设置集群参数
$ kubectl config set-cluster kubernetes \
--certificate-authority=/etc/kubernetes/ssl/ca.pem \
--embed-certs=true \
--server=${KUBE_APISERVER} \
--kubeconfig=bootstrap.kubeconfig
$ # 设置客户端认证参数
$ kubectl config set-credentials kubelet-bootstrap \
--token=${BOOTSTRAP_TOKEN} \
--kubeconfig=bootstrap.kubeconfig
$ # 设置上下文参数
$ kubectl config set-context default \
--cluster=kubernetes \
--user=kubelet-bootstrap \
--kubeconfig=bootstrap.kubeconfig
$ # 设置默认上下文
$ kubectl config use-context default --kubeconfig=bootstrap.kubeconfig
```

- `--embed-certs` 为 `true` 时表示将 `certificate-authority` 证书写入到生成的 `bootstrap.kubeconfig` 文件中；
- 设置客户端认证参数时没有指定秘钥和证书，后续由 `kube-apiserver` 自动生成；

创建 **kube-proxy** **kubeconfig** 文件

```
$ export KUBE_APISERVER="https://172.20.0.113:6443"
$ # 设置集群参数
$ kubectl config set-cluster kubernetes \
--certificate-authority=/etc/kubernetes/ssl/ca.pem \
--embed-certs=true \
--server=${KUBE_APISERVER} \
--kubeconfig=kube-proxy.kubeconfig
$ # 设置客户端认证参数
$ kubectl config set-credentials kube-proxy \
--client-certificate=/etc/kubernetes/ssl/kube-proxy.pem \
--client-key=/etc/kubernetes/ssl/kube-proxy-key.pem \
--embed-certs=true \
--kubeconfig=kube-proxy.kubeconfig
$ # 设置上下文参数
$ kubectl config set-context default \
--cluster=kubernetes \
--user=kube-proxy \
--kubeconfig=kube-proxy.kubeconfig
$ # 设置默认上下文
$ kubectl config use-context default --kubeconfig=kube-proxy.kubeconfig
```

- 设置集群参数和客户端认证参数时 `--embed-certs` 都为 `true`，这会将 `certificate-authority`、`client-certificate` 和 `client-key` 指向的证书文件内容写入到生成的 `kube-proxy.kubeconfig` 文件中；
- `kube-proxy.pem` 证书中 CN 为 `system:kube-proxy`，`kube-apiserver` 预定义的 `RoleBinding cluster-admin` 将 `User system:kube-proxy` 与 `Role system:node-proxier` 绑定，该 `Role` 授予了调用 `kube-apiserver Proxy` 相关 API 的权限；

分发 **kubeconfig** 文件

将两个 `kubeconfig` 文件分发到所有 Node 机器的 `/etc/kubernetes/` 目录

4.1.2 创建kubeconfig文件

```
$ cp bootstrap.kubeconfig kube-proxy.kubeconfig /etc/kubernetes/
```

for GitBook update 2017-08-07 13:54:27

创建高可用 etcd 集群

kubernetes 系统使用 etcd 存储所有数据，本文档介绍部署一个三节点高可用 etcd 集群的步骤，这三个节点复用 kubernetes master 机器，分别命名为 sz-pg-oam-docker-test-001.tendcloud.com 、 sz-pg-oam-docker-test-002.tendcloud.com 、 sz-pg-oam-docker-test-003.tendcloud.com :

- sz-pg-oam-docker-test-001.tendcloud.com : 172.20.0.113
- sz-pg-oam-docker-test-002.tendcloud.com : 172.20.0.114
- sz-pg-oam-docker-test-003.tendcloud.com : 172.20.0.115

TLS 认证文件

需要为 etcd 集群创建加密通信的 TLS 证书，这里复用以前创建的 kubernetes 证书

```
$ cp ca.pem kubernetes-key.pem kubernetes.pem /etc/kubernetes/ssl
```

- kubernetes 证书的 hosts 字段列表中包含上面三台机器的 IP，否则后续证书校验会失败；

下载二进制文件

到 <https://github.com/coreos/etcd/releases> 页面下载最新版本的二进制文件

```
$ https://github.com/coreos/etcd/releases/download/v3.1.5/etcd-v3.1.5-linux-amd64.tar.gz
$ tar -xvf etcd-v3.1.5-linux-amd64.tar.gz
$ sudo mv etcd-v3.1.5-linux-amd64/etcd* /usr/local/bin
```

创建 etcd 的 systemd unit 文件

4.1.3 创建高可用etcd集群

注意替换IP地址为自己的etcd集群的主机IP。

```
[Unit]
Description=Etcd Server
After=network.target
After=network-online.target
Wants=network-online.target
Documentation=https://github.com/coreos

[Service]
Type=notify
WorkingDirectory=/var/lib/etcd/
EnvironmentFile=-/etc/etcd/etcd.conf
ExecStart=/usr/local/bin/etcd \
--name ${ETCD_NAME} \
--cert-file=/etc/kubernetes/ssl/kubernetes.pem \
--key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
--peer-cert-file=/etc/kubernetes/ssl/kubernetes.pem \
--peer-key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
--trusted-ca-file=/etc/kubernetes/ssl/ca.pem \
--peer-trusted-ca-file=/etc/kubernetes/ssl/ca.pem \
--initial-advertise-peer-urls ${ETCD_INITIAL_ADVERTISE_PEER_URLS} \
--listen-peer-urls ${ETCD_LISTEN_PEER_URLS} \
--listen-client-urls ${ETCD_LISTEN_CLIENT_URLS},http://127.0.0.1:2379 \
--advertise-client-urls ${ETCD_ADVERTISE_CLIENT_URLS} \
--initial-cluster-token ${ETCD_INITIAL_CLUSTER_TOKEN} \
--initial-cluster infra1=https://172.20.0.113:2380,infra2=http://172.20.0.114:2380,infra3=https://172.20.0.115:2380 \
--initial-cluster-state new \
--data-dir=${ETCD_DATA_DIR}
Restart=on-failure
RestartSec=5
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

- 指定 `etcd` 的工作目录为 `/var/lib/etcd`，数据目录为 `/var/lib/etcd`，需在启动服务前创建这两个目录；
- 为了保证通信安全，需要指定 `etcd` 的公私钥(`cert-file`和`key-file`)、`Peers` 通信的公私钥和 `CA` 证书(`peer-cert-file`、`peer-key-file`、`peer-trusted-ca-file`)、客户端的`CA`证书（`trusted-ca-file`）；
- 创建 `kubernetes.pem` 证书时使用的 `kubernetes-csr.json` 文件的 `hosts` 字段包含所有 `etcd` 节点的IP，否则证书校验会出错；
- `--initial-cluster-state` 值为 `new` 时，`--name` 的参数值必须位于 `--initial-cluster` 列表中；

完整 `unit` 文件见：[etcd.service](#)

环境变量配置文件 `/etc/etcd/etcd.conf`。

```
# [member]
ETCD_NAME=infra1
ETCD_DATA_DIR="/var/lib/etcd"
ETCD_LISTEN_PEER_URLS="https://172.20.0.113:2380"
ETCD_LISTEN_CLIENT_URLS="https://172.20.0.113:2379"

#[cluster]
ETCD_INITIAL_ADVERTISE_PEER_URLS="https://172.20.0.113:2380"
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"
ETCD_ADVERTISE_CLIENT_URLS="https://172.20.0.113:2379"
```

这是172.20.0.113节点的配置，其他两个etcd节点只要将上面的IP地址改成相应节点的IP地址即可。`ETCD_NAME`换成对应节点的infra1/2/3。

启动 `etcd` 服务

```
$ sudo mv etcd.service /etc/systemd/system/
$ sudo systemctl daemon-reload
$ sudo systemctl enable etcd
$ sudo systemctl start etcd
$ systemctl status etcd
```

在所有的 kubernetes master 节点重复上面的步骤，直到所有机器的 etcd 服务都已启动。

验证服务

在任一 kubernetes master 机器上执行如下命令：

```
$ etcdctl \
--ca-file=/etc/kubernetes/ssl/ca.pem \
--cert-file=/etc/kubernetes/ssl/kubernetes.pem \
--key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
cluster-health
2017-04-11 15:17:09.082250 I | warning: ignoring ServerName for
user-provided CA for backwards compatibility is deprecated
2017-04-11 15:17:09.083681 I | warning: ignoring ServerName for
user-provided CA for backwards compatibility is deprecated
member 9a2ec640d25672e5 is healthy: got healthy result from http
s://172.20.0.115:2379
member bc6f27ae3be34308 is healthy: got healthy result from http
s://172.20.0.114:2379
member e5c92ea26c4edba0 is healthy: got healthy result from http
s://172.20.0.113:2379
cluster is healthy
```

结果最后一行为 `cluster is healthy` 时表示集群服务正常。

for GitBook update 2017-08-07 13:54:27

安装kubectl命令行工具

本文档介绍下载和配置 kubernetes 集群命令行工具 kubelet 的步骤。

下载 kubectl

```
$ wget https://dl.k8s.io/v1.6.0/kubernetes-client-linux-amd64.tar.gz
$ tar -xvf kubernetes-client-linux-amd64.tar.gz
$ cp kubernetes/client/bin/kube* /usr/bin/
$ chmod a+x /usr/bin/kube*
```

创建 kubectl kubeconfig 文件

```
$ export KUBE_APISERVER="https://172.20.0.113:6443"
$ # 设置集群参数
$ kubectl config set-cluster kubernetes \
  --certificate-authority=/etc/kubernetes/ssl/ca.pem \
  --embed-certs=true \
  --server=${KUBE_APISERVER}
$ # 设置客户端认证参数
$ kubectl config set-credentials admin \
  --client-certificate=/etc/kubernetes/ssl/admin.pem \
  --embed-certs=true \
  --client-key=/etc/kubernetes/ssl/admin-key.pem
$ # 设置上下文参数
$ kubectl config set-context kubernetes \
  --cluster=kubernetes \
  --user=admin
$ # 设置默认上下文
$ kubectl config use-context kubernetes
```

- `admin.pem` 证书 OU 字段值为 `system:masters`，`kube-apiserver` 预定义的 `RoleBinding cluster-admin` 将 Group `system:masters` 与 Role

- cluster-admin 绑定，该 Role 授予了调用 kube-apiserver 相关 API 的权限；
- 生成的 kubeconfig 被保存到 ~/.kube/config 文件；

for GitBook update 2017-08-07 13:54:27

部署master节点

kubernetes master 节点包含的组件：

- kube-apiserver
- kube-scheduler
- kube-controller-manager

目前这三个组件需要部署在同一台机器上。

- `kube-scheduler`、`kube-controller-manager` 和 `kube-apiserver` 三者功能紧密相关；
- 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；

~~本文档记录部署一个三个节点的高可用 kubernetes master 集群步骤。（后续创建一个 load balancer 来代理访问 kube-apiserver 的请求）~~

暂时未实现master节点的高可用。

TLS 证书文件

pem和token.csv证书文件我们在[TLS证书和秘钥](#)这一步中已经创建过了。我们再检查一下。

```
$ ls /etc/kubernetes/ssl
admin-key.pem  admin.pem  ca-key.pem  ca.pem  kube-proxy-key.pem
kube-proxy.pem  kubernetes-key.pem  kubernetes.pem
```

下载最新版本的二进制文件

有两种下载方式

方式一

从[github release 页面](#) 下载发布版 tarball，解压后再执行下载脚本

4.1.5 部署master节点

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/v1.6.0/kubernetes.tar.gz  
$ tar -xzvf kubernetes.tar.gz  
...  
$ cd kubernetes  
$ ./cluster/get-kube-binaries.sh  
...
```

方式二

从 [CHANGELOG 页面](#) 下载 client 或 server tarball 文件

server 的 tarball `kubernetes-server-linux-amd64.tar.gz` 已经包含了 client (`kubectl`) 二进制文件，所以不用单独下载 `kubernetes-client-linux-amd64.tar.gz` 文件；

```
$ # wget https://dl.k8s.io/v1.6.0/kubernetes-client-linux-amd64.tar.gz  
$ wget https://dl.k8s.io/v1.6.0/kubernetes-server-linux-amd64.tar.gz  
$ tar -xzvf kubernetes-server-linux-amd64.tar.gz  
...  
$ cd kubernetes  
$ tar -xzvf kubernetes-src.tar.gz
```

将二进制文件拷贝到指定路径

```
$ cp -r server/bin/{kube-apiserver,kube-controller-manager,kube-scheduler,kubectl,kube-proxy,kubelet} /usr/local/bin/
```

配置和启动 **kube-apiserver**

创建 **kube-apiserver** 的 **service** 配置文件

service 配置文件 `/usr/lib/systemd/system/kube-apiserver.service` 内容：

```
[Unit]
Description=Kubernetes API Service
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=network.target
After=etcd.service

[Service]
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/apiserver
ExecStart=/usr/local/bin/kube-apiserver \
    $KUBE_LOGTOSTDERR \
    $KUBE_LOG_LEVEL \
    $KUBE_ETCD_SERVERS \
    $KUBE_API_ADDRESS \
    $KUBE_API_PORT \
    $KUBELET_PORT \
    $KUBE_ALLOW_PRIV \
    $KUBE_SERVICE_ADDRESSES \
    $KUBE_ADMISSION_CONTROL \
    $KUBE_API_ARGS
Restart=on-failure
Type=notify
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

/etc/kubernetes/config 文件的内容为：

4.1.5 部署master节点

```
###  
# kubernetes system config  
#  
# The following values are used to configure various aspects of  
all  
# kubernetes services, including  
#  
#   kube-apiserver.service  
#   kube-controller-manager.service  
#   kube-scheduler.service  
#   kubelet.service  
#   kube-proxy.service  
# logging to stderr means we get it in the systemd journal  
KUBE_LOGTOSTDERR="--logtostderr=true"  
  
# journal message level, 0 is debug  
KUBE_LOG_LEVEL="--v=0"  
  
# Should this cluster be allowed to run privileged docker contai  
ners  
KUBE_ALLOW_PRIV="--allow-privileged=true"  
  
# How the controller-manager, scheduler, and proxy find the apis  
erver  
#KUBE_MASTER="--master=http://sz-pg-oam-docker-test-001.tendclou  
d.com:8080"  
KUBE_MASTER="--master=http://172.20.0.113:8080"
```

该配置文件同时被kube-apiserver、kube-controller-manager、kube-scheduler、kubelet、kube-proxy使用。

apiserver配置文件 /etc/kubernetes/apiserver 内容为：

```
###  
## kubernetes system config  
##  
## The following values are used to configure the kube-apiserver  
##  
#
```

4.1.5 部署master节点

```
## The address on the local server to listen to.
#KUBE_API_ADDRESS="--insecure-bind-address=sz-pg-oam-docker-test
-001.tendcloud.com"
KUBE_API_ADDRESS="--advertise-address=172.20.0.113 --bind-addres
s=172.20.0.113 --insecure-bind-address=172.20.0.113"
#
## The port on the local server to listen on.
#KUBE_API_PORT="--port=8080"
#
## Port minions listen on
#KUBELET_PORT="--kubelet-port=10250"
#
## Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd-servers=https://172.20.0.113:2379,http
s://172.20.0.114:2379,https://172.20.0.115:2379"
#
## Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

#
## default admission control policies
KUBE_ADMISSION_CONTROL="--admission-control=ServiceAccount,Nam
eSpaceLifecycle,NamespaceExists,LimitRanger,ResourceQuota"
#
## Add your own!
KUBE_API_ARGS="--authorization-mode=RBAC --runtime-config=rbac.a
uthorization.k8s.io/v1beta1 --kubelet-https=true --experimental-
bootstrap-token-auth --token-auth-file=/etc/kubernetes/token.csv
--service-node-port-range=30000-32767 --tls-cert-file=/etc/kube
rnetes/ssl/kubernetes.pem --tls-private-key-file=/etc/kubernetes
/ssl/kubernetes-key.pem --client-ca-file=/etc/kubernetes/ssl/ca.
pem --service-account-key-file=/etc/kubernetes/ssl/ca-key.pem --
etcd-cafile=/etc/kubernetes/ssl/ca.pem --etcd-certfile=/etc/kube
rnetes/ssl/kubernetes.pem --etcd-keyfile=/etc/kubernetes/ssl/kub
ernetes-key.pem --enable-swagger-ui=true --apiserver-count=3 --a
udit-log-maxage=30 --audit-log-maxbackup=3 --audit-log-maxsize=1
00 --audit-log-path=/var/lib/audit.log --event-ttl=1h"
```

- `--authorization-mode=RBAC` 指定在安全端口使用 RBAC 授权模式，拒绝

未通过授权的请求：

- kube-scheduler、kube-controller-manager 一般和 kube-apiserver 部署在同一台机器上，它们使用非安全端口和 kube-apiserver 通信；
- kubelet、kube-proxy、kubectl 部署在其它 Node 节点上，如果通过安全端口访问 kube-apiserver，则必须先通过 TLS 证书认证，再通过 RBAC 授权；
- kube-proxy、kubectl 通过在使用的证书里指定相关的 User、Group 来达到通过 RBAC 授权的目的；
- 如果使用了 kubelet TLS Bootstrap 机制，则不能再指定 `--kubelet-certificate-authority`、`--kubelet-client-certificate` 和 `--kubelet-client-key` 选项，否则后续 kube-apiserver 校验 kubelet 证书时出现 “x509: certificate signed by unknown authority” 错误；
- `--admission-control` 值必须包含 `ServiceAccount`；
- `--bind-address` 不能为 `127.0.0.1`；
- `runtime-config` 配置为 `rbac.authorization.k8s.io/v1beta1`，表示运行时的 apiVersion；
- `--service-cluster-ip-range` 指定 Service Cluster IP 地址段，该地址段不能路由可达；
- 缺省情况下 kubernetes 对象保存在 etcd `/registry` 路径下，可以通过 `--etcd-prefix` 参数进行调整；

完整 unit 见 [kube-apiserver.service](#)

启动 **kube-apiserver**

```
$ systemctl daemon-reload  
$ systemctl enable kube-apiserver  
$ systemctl start kube-apiserver  
$ systemctl status kube-apiserver
```

配置和启动 **kube-controller-manager**

创建 **kube-controller-manager** 的 service 配置文件

文件路径 `/usr/lib/systemd/system/kube-controller-manager.service`

4.1.5 部署master节点

```
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/controller-manager
ExecStart=/usr/local/bin/kube-controller-manager \
    $KUBE_LOGTOSTDERR \
    $KUBE_LOG_LEVEL \
    $KUBE_MASTER \
    $KUBE_CONTROLLER_MANAGER_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件 /etc/kubernetes/controller-manager 。

```
###
# The following values are used to configure the kubernetes controller-manager

# defaults from config and apiserver should be adequate

# Add your own!
KUBE_CONTROLLER_MANAGER_ARGS="--address=127.0.0.1 --service-cluster-ip-range=10.254.0.0/16 --cluster-name=kubernetes --cluster-signing-cert-file=/etc/kubernetes/ssl/ca.pem --cluster-signing-key-file=/etc/kubernetes/ssl/ca-key.pem --service-account-private-key-file=/etc/kubernetes/ssl/ca-key.pem --root-ca-file=/etc/kubernetes/ssl/ca.pem --leader-elect=true"
```

- `--service-cluster-ip-range` 参数指定 Cluster 中 Service 的CIDR范围，该网络在各 Node 间必须路由不可达，必须和 kube-apiserver 中的参数一致；
- `--cluster-signing-*` 指定的证书和私钥文件用来签名为 TLS BootStrap 创建的证书和私钥；
- `--root-ca-file` 用来对 kube-apiserver 证书进行校验，指定该参数后，才

会在**Pod** 容器的 **ServiceAccount** 中放置该 **CA** 证书文件；

- `--address` 值必须为 `127.0.0.1`，因为当前 `kube-apiserver` 期望 `scheduler` 和 `controller-manager` 在同一台机器，否则：

```
$ kubectl get componentstatuses
NAME                  STATUS      MESSAGE
ERROR
scheduler            Unhealthy   Get http://127.0.0.1:1025
1/healthz: dial tcp 127.0.0.1:10251: getsockopt: connection
refused
controller-manager   Healthy    ok

Healthy              {"health": "true"}
etcd-0               Healthy    {"health": "true"}

Healthy              {"health": "true"}
etcd-1
```

如果有组件report unhealthy请参考：<https://github.com/kubernetes-incubator/bootkube/issues/64>

完整 unit 见 [kube-controller-manager.service](#)

启动 **kube-controller-manager**

```
$ systemctl daemon-reload
$ systemctl enable kube-controller-manager
$ systemctl start kube-controller-manager
```

配置和启动 **kube-scheduler**

创建 **kube-scheduler** 的 **service** 配置文件

文件路径 `/usr/lib/systemd/system/kube-scheduler.service`。

```
[Unit]
Description=Kubernetes Scheduler Plugin
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/scheduler
ExecStart=/usr/local/bin/kube-scheduler \
           $KUBE_LOGTOSTDERR \
           $KUBE_LOG_LEVEL \
           $KUBE_MASTER \
           $KUBE_SCHEDULER_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件 `/etc/kubernetes/scheduler`。

```
###  
# kubernetes scheduler config  
  
# default config should be adequate  
  
# Add your own!  
KUBE_SCHEDULER_ARGS="--leader-elect=true --address=127.0.0.1"
```

- `--address` 值必须为 `127.0.0.1`，因为当前 `kube-apiserver` 期望 `scheduler` 和 `controller-manager` 在同一台机器；

完整 unit 见 [kube-scheduler.service](#)

启动 **kube-scheduler**

```
$ systemctl daemon-reload  
$ systemctl enable kube-scheduler  
$ systemctl start kube-scheduler
```

验证 **master** 节点功能

```
$ kubectl get componentstatuses  
NAME           STATUS  MESSAGE           ERROR  
scheduler      Healthy  ok  
controller-manager  Healthy  ok  
etcd-0          Healthy  {"health": "true"}  
etcd-1          Healthy  {"health": "true"}  
etcd-2          Healthy  {"health": "true"}
```

部署node节点

kubernetes node 节点包含如下组件：

- Flanneld：参考我之前写的文章[Kubernetes基于Flannel的网络配置](#)，之前没有配置TLS，现在需要在service配置文件中增加TLS配置。
- Docker1.12.5：docker的安装很简单，这里也不说了。
- kubelet
- kube-proxy

下面着重讲 kubelet 和 kube-proxy 的安装，同时还要将之前安装的flannel集成TLS验证。

注意：每台 node 上都需要安装 flannel，master 节点上可以不必安装。

目录和文件

我们再检查一下三个节点上，经过前几步操作生成的配置文件。

```
$ ls /etc/kubernetes/ssl
admin-key.pem  admin.pem  ca-key.pem  ca.pem  kube-proxy-key.pem
kube-proxy.pem  kubernetes-key.pem  kubernetes.pem
$ ls /etc/kubernetes/
apiserver  bootstrap.kubeconfig  config  controller-manager  kubelet
kube-proxy.kubeconfig  proxy  scheduler  ssl  token.csv
```

配置Flanneld

参考我之前写的文章[Kubernetes基于Flannel的网络配置](#)，之前没有配置TLS，现在需要在service配置文件中增加TLS配置。

直接使用yum安装flanneld即可。

```
yum install -y flannel
```

4.1.6 部署node节点

service配置文件 /usr/lib/systemd/system/flanneld.service 。

```
[Unit]
Description=Flanneld overlay address etcd agent
After=network.target
After=network-online.target
Wants=network-online.target
After=etcd.service
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
EnvironmentFile=-/etc/sysconfig/docker-network
ExecStart=/usr/bin/flanneld-start \
    -etcd-endpoints=${ETCD_ENDPOINTS} \
    -etcd-prefix=${ETCD_PREFIX} \
    $FLANNEL_OPTIONS
ExecStartPost=/usr/libexec/flannel/mk-docker-opts.sh -k DOCKER_N
ETWORK_OPTIONS -d /run/flannel/docker
Restart=on-failure

[Install]
WantedBy=multi-user.target
RequiredBy=docker.service
```

/etc/sysconfig/flanneld 配置文件。

```
# Flanneld configuration options

# etcd url location. Point this to the server where etcd runs
ETCD_ENDPOINTS="https://172.20.0.113:2379,https://172.20.0.114:2
379,https://172.20.0.115:2379"

# etcd config key. This is the configuration key that flannel q
ueries
# For address range assignment
ETCD_PREFIX="/kube-centos/network"

# Any additional options that you want to pass
FLANNEL_OPTIONS="-etcd-cafile=/etc/kubernetes/ssl/ca.pem -etcd-c
ertfile=/etc/kubernetes/ssl/kubernetes.pem -etcd-keyfile=/etc/ku
bernetes/ssl/kubernetes-key.pem"
```

在FLANNEL_OPTIONS中增加TLS的配置。

在**etcd**中创建网络配置

执行下面的命令为**docker**分配IP地址段。

```
etcdctl --endpoints=https://172.20.0.113:2379,https://172.20.0.1
14:2379,https://172.20.0.115:2379 \
--ca-file=/etc/kubernetes/ssl/ca.pem \
--cert-file=/etc/kubernetes/ssl/kubernetes.pem \
--key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
mkdir /kube-centos/network
etcdctl --endpoints=https://172.20.0.113:2379,https://172.20.0.1
14:2379,https://172.20.0.115:2379 \
--ca-file=/etc/kubernetes/ssl/ca.pem \
--cert-file=/etc/kubernetes/ssl/kubernetes.pem \
--key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
mk /kube-centos/network/config '{"Network": "172.30.0.0/16", "SubnetLen": 24, "Backend": {"Type": "vxlan"} }'
```

如果你要使用 `host-gw` 模式，可以直接将vxlan改成 host-gw 即可。

配置**Docker**

Flannel的[文档](#)中有写**Docker Integration**：

Docker daemon accepts `--bip` argument to configure the subnet of the docker0 bridge. It also accepts `--mtu` to set the MTU for docker0 and veth devices that it will be creating. Since flannel writes out the acquired subnet and MTU values into a file, the script starting Docker can source in the values and pass them to Docker daemon:

```
source /run/flannel/subnet.env
docker daemon --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} &
```

Systemd users can use `EnvironmentFile` directive in the .service file to pull in `/run/flannel/subnet.env`

如果你不是使用yum安装的flanneld，那么需要下载flannel github release中的tar包，解压后会获得一个**mk-docker-opts.sh**文件。

这个文件是用来 `Generate Docker daemon options based on flannel env file`。

执行 `./mk-docker-opts.sh -i` 将会生成如下两个文件环境变量文件。

`/run/flannel/subnet.env`

```
FLANNEL_NETWORK=172.30.0.0/16
FLANNEL_SUBNET=172.30.46.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
```

`/run/docker_opts.env`

```
DOCKER_OPT_BIP="--bip=172.30.46.1/24"
DOCKER_OPT_IPMASQ="--ip-masq=true"
DOCKER_OPT_MTU="--mtu=1450"
```

设置**docker0**网桥的IP地址

4.1.6 部署node节点

```
source /run/flannel/subnet.env  
ifconfig docker0 $FLANNEL_SUBNET
```

这样docker0和flannel网桥会在同一个子网中，如

```
6: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc n  
oqueue state DOWN  
    link/ether 02:42:da:bf:83:a2 brd ff:ff:ff:ff:ff:ff  
    inet 172.30.38.1/24 brd 172.30.38.255 scope global docker0  
        valid_lft forever preferred_lft forever  
7: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc n  
oqueue state UNKNOWN  
    link/ether 9a:29:46:61:03:44 brd ff:ff:ff:ff:ff:ff  
    inet 172.30.38.0/32 scope global flannel.1  
        valid_lft forever preferred_lft forever
```

同时在 docker 的配置文件 [docker.service](#) 中增加环境变量配置：

```
EnvironmentFile=-/run/flannel/docker  
EnvironmentFile=-/run/docker_opts.env  
EnvironmentFile=-/run/flannel/subnet.env
```

防止主机重启后 docker 自动重启时加载不到该上述环境变量。

启动**docker**

重启了docker后还要重启kubelet，这时又遇到问题，kubelet启动失败。报错：

```
Mar 31 16:44:41 sz-pg-oam-docker-test-002.tendcloud.com kubelet[  
81047]: error: failed to run Kubelet: failed to create kubelet:  
misconfiguration: kubelet cgroup driver: "cgroupfs" is different  
from docker cgroup driver: "systemd"
```

这是kubelet与docker的**cgroup driver**不一致导致的，kubelet启动的时候有个 `-cgroup-driver` 参数可以指定为"cgrounfs"或者"systemd"。

4.1.6 部署node节点

```
--cgroup-driver string                                Driver
that the kubelet uses to manipulate cgroups on the host. Possible values: 'cgroupfs', 'systemd' (default "cgroupfs")
```

启动flannel

```
systemctl daemon-reload
systemctl start flanneld
systemctl status flanneld
```

现在查询etcd中的内容可以看到：

```
$etcdctl --endpoints=${ETCD_ENDPOINTS} \
--ca-file=/etc/kubernetes/ssl/ca.pem \
--cert-file=/etc/kubernetes/ssl/kubernetes.pem \
--key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
ls /kube-centos/network/subnets
/kube-centos/network/subnets/172.30.14.0-24
/kube-centos/network/subnets/172.30.38.0-24
/kube-centos/network/subnets/172.30.46.0-24
$etcdctl --endpoints=${ETCD_ENDPOINTS} \
--ca-file=/etc/kubernetes/ssl/ca.pem \
--cert-file=/etc/kubernetes/ssl/kubernetes.pem \
--key-file=/etc/kubernetes/ssl/kubernetes-key.pem \
get /kube-centos/network/config
{ "Network": "172.30.0.0/16", "SubnetLen": 24, "Backend": { "Type": "vxlan" } }
$etcdctl get /kube-centos/network/subnets/172.30.14.0-24
{"PublicIP":"172.20.0.114","BackendType":"vxlan","BackendData":{ "VtepMAC":"56:27:7d:1c:08:22"}}
$etcdctl get /kube-centos/network/subnets/172.30.38.0-24
{"PublicIP":"172.20.0.115","BackendType":"vxlan","BackendData":{ "VtepMAC":"12:82:83:59:cf:b8"}}
$etcdctl get /kube-centos/network/subnets/172.30.46.0-24
{"PublicIP":"172.20.0.113","BackendType":"vxlan","BackendData":{ "VtepMAC":"e6:b2:fd:f6:66:96"}}
```

安装和配置 **kubelet**

kubelet 启动时向 **kube-apiserver** 发送 TLS bootstrapping 请求，需要先将 bootstrap token 文件中的 **kubelet-bootstrap** 用户赋予 **system:node-bootstrapper** cluster 角色(role)，然后 **kubelet** 才能有权限创建认证请求(certificate signing requests)：

```
$ cd /etc/kubernetes
$ kubectl create clusterrolebinding kubelet-bootstrap \
--clusterrole=system:node-bootstrapper \
--user=kubelet-bootstrap
```

- `--user=kubelet-bootstrap` 是在 `/etc/kubernetes/token.csv` 文件中指定的用户名，同时也写入了 `/etc/kubernetes/bootstrap.kubeconfig` 文件；

下载最新的 **kubelet** 和 **kube-proxy** 二进制文件

```
$ wget https://dl.k8s.io/v1.6.0/kubernetes-server-linux-amd64.t
r.gz
$ tar -xzvf kubernetes-server-linux-amd64.tar.gz
$ cd kubernetes
$ tar -xzvf kubernetes-src.tar.gz
$ cp -r ./server/bin/{kube-proxy,kubelet} /usr/local/bin/
```

创建 **kubelet** 的**service**配置文件

文件位置 `/usr/lib/systemd/system/kubelet.service`。

```
[Unit]
Description=Kubernetes Kubelet Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=docker.service
Requires=docker.service

[Service]
WorkingDirectory=/var/lib/kubelet
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/kubelet
ExecStart=/usr/local/bin/kubelet \
           $KUBE_LOGTOSTDERR \
           $KUBE_LOG_LEVEL \
           $KUBELET_API_SERVER \
           $KUBELET_ADDRESS \
           $KUBELET_PORT \
           $KUBELET_HOSTNAME \
           $KUBE_ALLOW_PRIV \
           $KUBELET_POD_INFRA_CONTAINER \
           $KUBELET_ARGS
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

kubelet的配置文件 `/etc/kubernetes/kubelet`。其中的IP地址更改为你的每台 node 节点的IP地址。

注意：`/var/lib/kubelet` 需要手动创建。

```

###  

## kubernetes kubelet (minion) config  

#  

## The address for the info server to serve on (set to 0.0.0.0 or "" for all interfaces)  

KUBELET_ADDRESS="--address=172.20.0.113"  

#  

## The port for the info server to serve on  

#KUBELET_PORT="--port=10250"  

#  

## You may leave this blank to use the actual hostname  

KUBELET_HOSTNAME="--hostname-override=172.20.0.113"  

#  

## location of the api-server  

KUBELET_API_SERVER="--api-servers=http://172.20.0.113:8080"  

#  

## pod infrastructure container  

KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image=sz-pg-oam-docker-hub-001.tendcloud.com/library/pod-infrastructure:rhel7"  

#  

## Add your own!  

KUBELET_ARGS="--cgroup-driver=systemd --cluster-dns=10.254.0.2 --experimental-bootstrap-kubeconfig=/etc/kubernetes/bootstrap.kubeconfig --kubeconfig=/etc/kubernetes/kubelet.kubeconfig --require-kubeconfig --cert-dir=/etc/kubernetes/ssl --cluster-domain=cluster.local. --hairpin-mode promiscuous-bridge --serialize-image-pulls=false"

```

- `--address` 不能设置为 `127.0.0.1`，否则后续 Pods 访问 kubelet 的 API 接口时会失败，因为 Pods 访问的 `127.0.0.1` 指向自己而不是 kubelet；
- 如果设置了 `--hostname-override` 选项，则 `kube-proxy` 也需要设置该选项，否则会出现找不到 Node 的情况；
- `--experimental-bootstrap-kubeconfig` 指向 bootstrap kubeconfig 文件，kubelet 使用该文件中的用户名和 token 向 kube-apiserver 发送 TLS Bootstrapping 请求；
- 管理员通过了 CSR 请求后，kubelet 自动在 `--cert-dir` 目录创建证书和私

钥文件(`kubelet-client.crt` 和 `kubelet-client.key`), 然后写入 `--kubeconfig` 文件;

- 建议在 `--kubeconfig` 配置文件中指定 `kube-apiserver` 地址, 如果未指定 `--api-servers` 选项, 则必须指定 `--require-kubeconfig` 选项后才从配置文件中读取 `kube-apiserver` 的地址, 否则 `kubelet` 启动后将找不到 `kube-apiserver`(日志中提示未找到 API Server) , `kubectl get nodes` 不会返回对应的 Node 信息;
- `--cluster-dns` 指定 `kubedns` 的 Service IP(可以先分配, 后续创建 `kubedns` 服务时指定该 IP), `--cluster-domain` 指定域名后缀, 这两个参数同时指定后才会生效;
- `--kubeconfig=/etc/kubernetes/kubelet.kubeconfig` 中指定的 `kubelet.kubeconfig` 文件在第一次启动 `kubelet` 之前并不存在, 请看下文, 当通过 CSR 请求后会自动生成 `kubelet.kubeconfig` 文件, 如果你的节点上已经生成了 `~/.kube/config` 文件, 你可以将该文件拷贝到该路径下, 并重命名为 `kubelet.kubeconfig` , 所有 node 节点可以共用同一个 `kubelet.kubeconfig` 文件, 这样新添加的节点就不需要再创建 CSR 请求就能自动添加到 `kubernetes` 集群中。同样, 在任意能够访问到 `kubernetes` 集群的主机上使用 `kubectl --kubeconfig` 命令操作集群时, 只要使用 `~/.kube/config` 文件就可以通过权限认证, 因为这里面已经有认证信息并认为你是 `admin` 用户, 对集群拥有所有权限。

完整 unit 见 [kubelet.service](#)

启动 `kublet`

```
$ systemctl daemon-reload
$ systemctl enable kubelet
$ systemctl start kubelet
$ systemctl status kubelet
```

通过 `kublet` 的 TLS 证书请求

`kubelet` 首次启动时向 `kube-apiserver` 发送证书签名请求, 必须通过后 `kubernetes` 系统才会将该 Node 加入到集群。

查看未授权的 CSR 请求

4.1.6 部署node节点

```
$ kubectl get csr
NAME      AGE      REQUESTOR      CONDITION
csr-2b308  4m      kubelet-bootstrap  Pending
$ kubectl get nodes
No resources found.
```

通过 CSR 请求

```
$ kubectl certificate approve csr-2b308
certificatesigningrequest "csr-2b308" approved
$ kubectl get nodes
NAME      STATUS     AGE      VERSION
10.64.3.7  Ready     49m     v1.6.1
```

自动生成了 kubelet kubeconfig 文件和公私钥

```
$ ls -l /etc/kubernetes/kubelet.kubeconfig
-rw----- 1 root root 2284 Apr  7 02:07 /etc/kubernetes/kubelet
.kubeconfig
$ ls -l /etc/kubernetes/ssl/kubelet*
-rw-r--r-- 1 root root 1046 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet-client.crt
-rw----- 1 root root  227 Apr  7 02:04 /etc/kubernetes/ssl/kub
elet-client.key
-rw-r--r-- 1 root root 1103 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet.crt
-rw----- 1 root root 1675 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet.key
```

注意：假如你更新kubernetes的证书，只要没有更新 token.csv ，当重启kubelet后，该node就会自动加入到kubernetes集群中，而不会重新发送 certificaterequest ，也不需要在master节点上执行 kubectl certificate approve 操作。前提是不要删除node节点上的 /etc/kubernetes/ssl/kubelet* 和 /etc/kubernetes/kubelet.kubeconfig 文件。否则kubelet启动时会提示找不到证书而失败。

配置 kube-proxy

创建 **kube-proxy** 的**service**配置文件

文件路径 `/usr/lib/systemd/system/kube-proxy.service`。

```
[Unit]
Description=Kubernetes Kube-Proxy Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=network.target

[Service]
EnvironmentFile=-/etc/kubernetes/config
EnvironmentFile=-/etc/kubernetes/proxy
ExecStart=/usr/local/bin/kube-proxy \
    $KUBE_LOGTOSTDERR \
    $KUBE_LOG_LEVEL \
    $KUBE_MASTER \
    $KUBE_PROXY_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

kube-proxy配置文件 `/etc/kubernetes/proxy`。

```
###  
# kubernetes proxy config  
  
# default config should be adequate  
  
# Add your own!  
KUBE_PROXY_ARGS="--bind-address=172.20.0.113 --hostname-override  
=172.20.0.113 --kubeconfig=/etc/kubernetes/kube-proxy.kubeconfig  
--cluster-cidr=10.254.0.0/16"
```

- `--hostname-override` 参数值必须与 `kubelet` 的值一致，否则 `kube-proxy`

启动后会找不到该 Node，从而不会创建任何 iptables 规则；

- kube-proxy 根据 `--cluster-cidr` 判断集群内部和外部流量，指定 `--cluster-cidr` 或 `--masquerade-all` 选项后 kube-proxy 才会对访问 Service IP 的请求做 SNAT；
- `--kubeconfig` 指定的配置文件嵌入了 kube-apiserver 的地址、用户名、证书、秘钥等请求和认证信息；
- 预定义的 RoleBinding `cluster-admin` 将 User `system:kube-proxy` 与 Role `system:node-proxier` 绑定，该 Role 授予了调用 kube-apiserver Proxy 相关 API 的权限；

完整 unit 见 [kube-proxy.service](#)

启动 kube-proxy

```
$ systemctl daemon-reload
$ systemctl enable kube-proxy
$ systemctl start kube-proxy
$ systemctl status kube-proxy
```

验证测试

我们创建一个nginx的service试一下集群是否可用。

```
$ kubectl run nginx --replicas=2 --labels="run=load-balancer-example" --image=sz-pg-oam-docker-hub-001.tendcloud.com/library/nginx:1.9 --port=80
deployment "nginx" created
$ kubectl expose deployment nginx --type=NodePort --name=example-service
service "example-service" exposed
$ kubectl describe svc example-service
Name:           example-service
Namespace:      default
Labels:         run=load-balancer-example
Annotations:    <none>
Selector:       run=load-balancer-example
Type:          NodePort
```

4.1.6 部署node节点

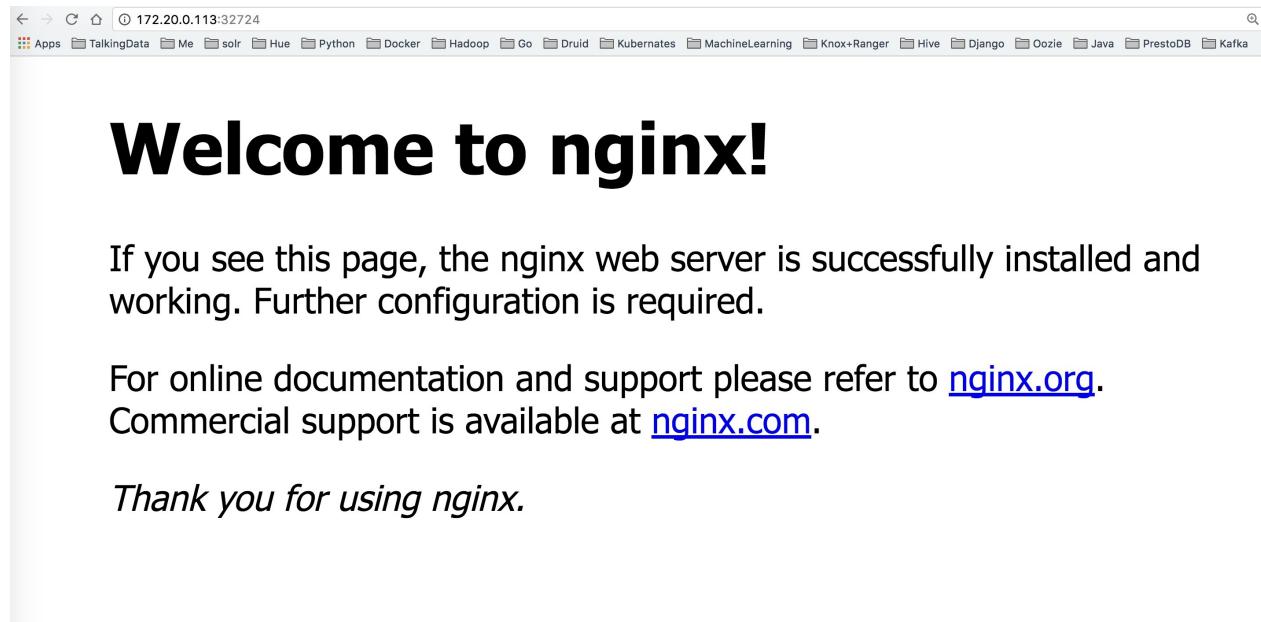
```
IP:          10.254.62.207
Port:        <unset>    80/TCP
NodePort:    <unset>    32724/TCP
Endpoints:   172.30.60.2:80,172.30.94.2:80
Session Affinity: None
Events:      <none>
$ curl "10.254.62.207:80"
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

提示：上面的测试示例中使用的nginx是我的私有镜像仓库中的镜像 sz-pg-oam-docker-hub-001.tendcloud.com/library/nginx:1.9，大家在测试过程中请换成自己的nginx镜像地址。

访问 172.20.0.113:32724 或 172.20.0.114:32724 或者 172.20.0.115:32724 都可以得到nginx的页面。



for GitBook update 2017-08-07 13:54:27

安装kubedns插件

官方的yaml文件目录：`kubernetes/cluster/addons/dns`。

该插件直接使用kubernetes部署，官方的配置文件中包含以下镜像：

```
gcr.io/google_containers/k8s-dns-dnsmasq-nanny-amd64:1.14.1  
gcr.io/google_containers/k8s-dns-kube-dns-amd64:1.14.1  
gcr.io/google_containers/k8s-dns-sidecar-amd64:1.14.1
```

我clone了上述镜像，上传到我的私有镜像仓库：

```
sz-pg-oam-docker-hub-001.tendcloud.com/library/k8s-dns-dnsmasq-nanny-amd64:1.14.1  
sz-pg-oam-docker-hub-001.tendcloud.com/library/k8s-dns-kube-dns-amd64:1.14.1  
sz-pg-oam-docker-hub-001.tendcloud.com/library/k8s-dns-sidecar-amd64:1.14.1
```

同时上传了一份到时速云备份：

```
index.tenxcloud.com/jimmy/k8s-dns-dnsmasq-nanny-amd64:1.14.1  
index.tenxcloud.com/jimmy/k8s-dns-kube-dns-amd64:1.14.1  
index.tenxcloud.com/jimmy/k8s-dns-sidecar-amd64:1.14.1
```

以下yaml配置文件中使用的是私有镜像仓库中的镜像。

```
kubedns-cm.yaml  
kubedns-sa.yaml  
kubedns-controller.yaml  
kubedns-svc.yaml
```

已经修改好的 yaml 文件见：[dns](#)

系统预定义的 **RoleBinding**

预定义的 RoleBinding `system:kube-dns` 将 `kube-system` 命名空间的 `kube-dns` ServiceAccount 与 `system:kube-dns` Role 绑定，该 Role 具有访问 kube-apiserver DNS 相关 API 的权限；

```
$ kubectl get clusterrolebindings system:kube-dns -o yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: 2017-04-11T11:20:42Z
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-dns
  resourceVersion: "58"
  selfLink: /apis/rbac.authorization.k8s.io/v1beta1/clusterrolebindingssystem%3Akube-dns
  uid: e61f4d92-1ea8-11e7-8cd7-f4e9d49f8ed0
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:kube-dns
subjects:
- kind: ServiceAccount
  name: kube-dns
  namespace: kube-system
```

`kubedns-controller.yaml` 中定义的 Pods 时使用了 `kubedns-sa.yaml` 文件定义的 `kube-dns` ServiceAccount，所以具有访问 kube-apiserver DNS 相关 API 的权限。

配置 **kube-dns ServiceAccount**

无需修改。

配置 **kube-dns** 服务

```
$ diff kubedns-svc.yaml.base kubedns-svc.yaml
30c30
<   clusterIP: __PILLAR__DNS__SERVER__
---
>   clusterIP: 10.254.0.2
```

- spec.clusterIP = 10.254.0.2，即明确指定了 kube-dns Service IP，这个 IP 需要和 kubelet 的 --cluster-dns 参数值一致；

配置 **kube-dns Deployment**

```
$ diff kubedns-controller.yaml.base kubedns-controller.yaml
58c58
<           image: gcr.io/google_containers/k8s-dns-kube-dns-amd64
:1.14.1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
k8s-dns-kube-dns-amd64:v1.14.1
88c88
<           - --domain=__PILLAR__DNS__DOMAIN__.
---
>           - --domain=cluster.local.
92c92
<           __PILLAR__FEDERATIONS__DOMAIN__MAP__
---
>           #__PILLAR__FEDERATIONS__DOMAIN__MAP__
110c110
<           image: gcr.io/google_containers/k8s-dns-dnsmasq-nanny-
amd64:1.14.1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
k8s-dns-dnsmasq-nanny-amd64:v1.14.1
129c129
<           - --server=/__PILLAR__DNS__DOMAIN__/127.0.0.1#10053
---
```

4.1.7 安装kubedns插件

```
>      - --server=/cluster.local./127.0.0.1#10053
148c148
<      image: gcr.io/google_containers/k8s-dns-sidecar-amd64:
1.14.1
---
>      image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
k8s-dns-sidecar-amd64:v1.14.1
161,162c161,162
<      - --probe=kubedns,127.0.0.1:10053,kubernetes.default.s
vc.__PILLAR__DNS__DOMAIN__,5,A
<      - --probe=dnsMasq,127.0.0.1:53,kubernetes.default.svc.
__PILLAR__DNS__DOMAIN__,5,A
---
>      - --probe=kubedns,127.0.0.1:10053,kubernetes.default.s
vc.cluster.local.,5,A
>      - --probe=dnsMasq,127.0.0.1:53,kubernetes.default.svc.
cluster.local.,5,A
```

- 使用系统已经做了 RoleBinding 的 kube-dns ServiceAccount，该账户具有访问 kube-apiserver DNS 相关 API 的权限；

执行所有定义文件

```
$ pwd
/root/kubedns
$ ls *.yaml
kubedns-cm.yaml  kubedns-controller.yaml  kubedns-sa.yaml  kubed
ns-svc.yaml
$ kubectl create -f .
```

检查 **kubedns** 功能

新建一个 Deployment

```
$ cat my-nginx.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: sz-pg-oam-docker-hub-001.tendcloud.com/library/nginx:1.9
      ports:
        - containerPort: 80
$ kubectl create -f my-nginx.yaml
```

Export 该 Deployment, 生成 my-nginx 服务

```
$ kubectl expose deploy my-nginx
$ kubectl get services --all-namespaces |grep my-nginx
default     my-nginx     10.254.179.239   <none>       80/TCP
                           42m
```

创建另一个 Pod，查看 /etc/resolv.conf 是否包含 kubelet 配置的 --cluster-dns 和 --cluster-domain，是否能够将服务 my-nginx 解析到 Cluster IP 10.254.179.239。

```
$ kubectl create -f nginx-pod.yaml
$ kubectl exec nginx -i -t -- /bin/bash
root@nginx:/# cat /etc/resolv.conf
nameserver 10.254.0.2
search default.svc.cluster.local. svc.cluster.local. cluster.loc
al. tendcloud.com
options ndots:5

root@nginx:/# ping my-nginx
PING my-nginx.default.svc.cluster.local (10.254.179.239): 56 dat
a bytes
76 bytes from 119.147.223.109: Destination Net Unreachable
^C--- my-nginx.default.svc.cluster.local ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss

root@nginx:/# ping kubernetes
PING kubernetes.default.svc.cluster.local (10.254.0.1): 56 data
bytes
^C--- kubernetes.default.svc.cluster.local ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss

root@nginx:/# ping kube-dns.kube-system.svc.cluster.local
PING kube-dns.kube-system.svc.cluster.local (10.254.0.2): 56 dat
a bytes
^C--- kube-dns.kube-system.svc.cluster.local ping statistics ---
6 packets transmitted, 0 packets received, 100% packet loss
```

从结果来看，service名称可以正常解析。

for GitBook update 2017-08-07 13:54:27

安装**dashboard**插件

官方文件目录：`kubernetes/cluster/addons/dashboard`

我们使用的文件

```
$ ls *.yaml
dashboard-controller.yaml  dashboard-service.yaml  dashboard-rbac
.yaml
```

已经修改好的 yaml 文件见：[dashboard](#)

由于 `kube-apiserver` 启用了 `RBAC` 授权，而官方源码目录的 `dashboard-controller.yaml` 没有定义授权的 `ServiceAccount`，所以后续访问 `kube-apiserver` 的 API 时会被拒绝，web 中提示：

```
Forbidden (403)

User "system:serviceaccount:kube-system:default" cannot list job
s.batch in the namespace "default". (get jobs.batch)
```

增加了一个 `dashboard-rbac.yaml` 文件，定义一个名为 `dashboard` 的 `ServiceAccount`，然后将它和 Cluster Role view 绑定。

配置**dashboard-service**

```
$ diff dashboard-service.yaml.orig dashboard-service.yaml
10a11
>   type: NodePort
```

- 指定端口类型为 `NodePort`，这样外界可以通过地址 `nodeIP:nodePort` 访问 `dashboard`；

配置**dashboard-controller**

```
$ diff dashboard-controller.yaml.orig dashboard-controller.yaml
23c23
<           image: gcr.io/google_containers/kubernetes-dashboard-a
md64:v1.6.0
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
kubernetes-dashboard-amd64:v1.6.0
```

执行所有定义文件

```
$ pwd
/root/kubernetes/cluster/addons/dashboard
$ ls *.yaml
dashboard-controller.yaml  dashboard-service.yaml
$ kubectl create -f .
service "kubernetes-dashboard" created
deployment "kubernetes-dashboard" created
```

检查执行结果

查看分配的 NodePort

```
$ kubectl get services kubernetes-dashboard -n kube-system
NAME                  CLUSTER-IP      EXTERNAL-IP    PORT(S)
AGE
kubernetes-dashboard   10.254.224.130  <nodes>        80:30312/T
CP      25s
```

- NodePort 30312映射到 dashboard pod 80端口；

检查 controller

```
$ kubectl get deployment kubernetes-dashboard -n kube-system
NAME                  DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
E   AGE
kubernetes-dashboard   1         1         1           1
3m
$ kubectl get pods -n kube-system | grep dashboard
kubernetes-dashboard-1339745653-pmn6z   1/1       Running   0
4m
```

访问dashboard

有以下三种方式：

- kubernetes-dashboard 服务暴露了 NodePort，可以使用 `http://NodeIP:nodePort` 地址访问 dashboard；
- 通过 kube-apiserver 访问 dashboard（https 6443 端口和 http 8080 端口方式）；
- 通过 kubectl proxy 访问 dashboard：

通过 kubectl proxy 访问 dashboard

启动代理

```
$ kubectl proxy --address='172.20.0.113' --port=8086 --accept-hosts='^*$'
Starting to serve on 172.20.0.113:8086
```

- 需要指定 `--accept-hosts` 选项，否则浏览器访问 dashboard 页面时提示“Unauthorized”；

浏览器访问 URL：`http://172.20.0.113:8086/ui` 自动跳转到：`http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard/#/workload?namespace=default`

通过 kube-apiserver 访问 dashboard

4.1.8 安装dashboard插件

获取集群服务地址列表

```
$ kubectl cluster-info  
Kubernetes master is running at https://172.20.0.113:6443  
KubeDNS is running at https://172.20.0.113:6443/api/v1/proxy/nam  
espaces/kube-system/services/kube-dns  
kubernetes-dashboard is running at https://172.20.0.113:6443/api  
/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
```

浏览器访问

URL : `https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-
system/services/kubernetes-dashboard` (浏览器会提示证书验证，因为通过
加密通道，以改方式访问的话，需要提前导入证书到你的计算机中)。这是我当时
在这遇到的坑：[通过 kube-apiserver 访问dashboard，提示User
"system:anonymous" cannot proxy services in the namespace "kube-system".
#5](#)，已经解决。

导入证书

将生成的admin.pem证书转换格式

```
openssl pkcs12 -export -in admin.pem -out admin.p12 -inkey admi  
n-key.pem
```

将生成的 `admin.p12` 证书导入的你的电脑，导出的时候记住你设置的密码，导入
的时候还要用到。

如果你不想使用**https**的话，可以直接访问insecure port 8080端
口：`http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-
system/services/kubernetes-dashboard`

4.1.8 安装dashboard插件

The screenshot shows the Kubernetes dashboard interface. At the top, it says "Admin > Namespaces > kube-system". On the left, there's a sidebar with "Namespaces" selected. The main area has two tabs: "Details" and "Events".

Details Tab:

- Name: kube-system
- Creation time: 2017-04-11T11:20
- Status: Active

Events Tab:

Message	Source	Sub-object	Count	First seen	Last seen
Killing container with id docker://c857a23eb359fb5f46c080b0404f41731d7be47d9729eb1e5ae9d8b2be123d5f:Need to kill Pod	kubelet 172.20.0.114	spec.containers(kubernetes-dashboard)	1	2017-04-12T06:51 UTC	2017-04-12T06:51 UTC
Deleted pod: kubernetes-dashboard-1752429380-7vk0t	replicaset-controller	-	1	2017-04-12T06:51 UTC	2017-04-12T06:51 UTC
Successfully assigned kubernetes-dashboard-3966630548-61b48 to 172.20.0.113	default-scheduler	-	1	2017-04-12T06:56 UTC	2017-04-12T06:56 UTC
Container image "sz-pg-pam-docker-hub-001.tendcloud.com/library/kubernetes-dashboard-amd64:v1.6.0" already present on machine	kubelet 172.20.0.113	spec.containers(kubernetes-dashboard)	1	2017-04-12T06:57 UTC	2017-04-12T06:57 UTC
Created container with id c36e4cc8e1108805a34affd872449442a3bf628466b8ea2da3c99caf1d7cec23	kubelet 172.20.0.113	spec.containers(kubernetes-dashboard)	1	2017-04-12T06:57 UTC	2017-04-12T06:57 UTC
Started container with id c36e4cc8e1108805a34affd872449442a3bf628466b8ea2da3c99caf1d7cec23	kubelet 172.20.0.113	spec.containers(kubernetes-dashboard)	1	2017-04-12T06:57 UTC	2017-04-12T06:57 UTC
⚠ Error creating pods "kubernetes-dashboard-3966630548-": Is forbidden: service account kubernetes-dashboard was replicated from controller	replicaset-controller	-	4	2017-04-12T06:56 UTC	2017-04-12T06:56 UTC

Figure: kubernetes-dashboard

由于缺少 Heapster 插件，当前 dashboard 不能展示 Pod、Nodes 的 CPU、内存等 metric 图形。

for GitBook

update 2017-08-07 13:54:27

安装**heapster**插件

到 [heapster release](#) 页面 下载最新版本的 heapster。

```
$ wget https://github.com/kubernetes/heapster/archive/v1.3.0.zip  
$ unzip v1.3.0.zip  
$ mv v1.3.0.zip heapster-1.3.0
```

文件目录： heapster-1.3.0/deploy/kube-config/influxdb

```
$ cd heapster-1.3.0/deploy/kube-config/influxdb  
$ ls *.yaml  
grafana-deployment.yaml  grafana-service.yaml  heapster-deployment.yaml  
heapster-service.yaml  influxdb-deployment.yaml  influxdb-service.yaml  
heapster-rbac.yaml
```

我们自己创建了heapster的rbac配置 `heapster-rbac.yaml`。

已经修改好的 yaml 文件见：[heapster](#)

配置 **grafana-deployment**

```
$ diff grafana-deployment.yaml.orig grafana-deployment.yaml
16c16
<           image: gcr.io/google_containers/heapster-grafana-amd64
:v4.0.2
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
heapster-grafana-amd64:v4.0.2
40,41c40,41
<           # value: /api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/
<           value: /
---
>           value: /api/v1/proxy/namespaces/kube-system/services
/monitoring-grafana/
>           #value: /
```

- 如果后续使用 kube-apiserver 或者 kubectl proxy 访问 grafana dashboard，则必须将 `GF_SERVER_ROOT_URL` 设置为 `/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/`，否则后续访问grafana时访问时提示找不到 `http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/api/dashboards/home` 页面；

配置 heapster-deployment

```
$ diff heapster-deployment.yaml.orig heapster-deployment.yaml
16c16
<           image: gcr.io/google_containers/heapster-amd64:v1.3.0-
beta.1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
heapster-amd64:v1.3.0-beta.1
```

配置 influxdb-deployment

influxdb 官方建议使用命令行或 HTTP API 接口来查询数据库，从 v1.1.0 版本开始默认关闭 admin UI，将在后续版本中移除 admin UI 插件。

开启镜像中 admin UI 的办法如下：先导出镜像中的 influxdb 配置文件，开启 admin 插件后，再将配置文件内容写入 ConfigMap，最后挂载到镜像中，达到覆盖原始配置的目的：

注意：manifests 目录已经提供了修改后的 ConfigMap 定义文件

```
$ # 导出镜像中的 influxdb 配置文件
$ docker run --rm --entrypoint 'cat' -ti lvanneo/heapster-influxdb-amd64:v1.1.1 /etc/config.toml >config.toml.orig
$ cp config.toml.orig config.toml
$ # 修改：启用 admin 接口
$ vim config.toml
$ diff config.toml.orig config.toml
35c35
<   enabled = false
---
>   enabled = true
$ # 将修改后的配置写入到 ConfigMap 对象中
$ kubectl create configmap influxdb-config --from-file=config.toml -n kube-system
configmap "influxdb-config" created
$ # 将 ConfigMap 中的配置文件挂载到 Pod 中，达到覆盖原始配置的目的
$ diff influxdb-deployment.yaml.orig influxdb-deployment.yaml
16c16
<           image: gcr.io/google_containers/heapster-influxdb-amd64:v1.1.1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/heapster-influxdb-amd64:v1.1.1
19a20,21
>           - mountPath: /etc/
>             name: influxdb-config
22a25,27
>           - name: influxdb-config
>             configMap:
>               name: influxdb-config
```

配置 monitoring-influxdb Service

```
$ diff influxdb-service.yaml.orig influxdb-service.yaml
12a13
>   type: NodePort
15a17,20
>     name: http
>     - port: 8083
>       targetPort: 8083
>     name: admin
```

- 定义端口类型为 NodePort，额外增加了 admin 端口映射，用于后续浏览器访问 influxdb 的 admin UI 界面；

执行所有定义文件

```
$ pwd
/root/heapster-1.3.0/deploy/kube-config/influxdb
$ ls *.yaml
grafana-service.yaml      heapster-rbac.yaml      influxdb-cm.yaml
influxdb-service.yaml
grafana-deployment.yaml  heapster-deployment.yaml  heapster-service.yaml
influxdb-deployment.yaml
$ kubectl create -f .
deployment "monitoring-grafana" created
service "monitoring-grafana" created
deployment "heapster" created
serviceaccount "heapster" created
clusterrolebinding "heapster" created
service "heapster" created
configmap "influxdb-config" created
deployment "monitoring-influxdb" created
service "monitoring-influxdb" created
```

检查执行结果

检查 Deployment

4.1.9 安装heapster插件

```
$ kubectl get deployments -n kube-system | grep -E 'heapster|monitoring'
heapster              1           1           1           1
  2m
monitoring-grafana   1           1           1           1
  2m
monitoring-influxdb  1           1           1           1
  2m
```

检查 Pods

```
$ kubectl get pods -n kube-system | grep -E 'heapster|monitoring'
heapster-110704576-gpg8v          1/1       Running    0
  2m
monitoring-grafana-2861879979-9z89f 1/1       Running    0
  2m
monitoring-influxdb-1411048194-lzrpc 1/1       Running    0
  2m
```

检查 kubernets dashboard 界面，看是显示各 Nodes、Pods 的 CPU、内存、负载等利用率曲线图；

4.1.9 安装heapster插件

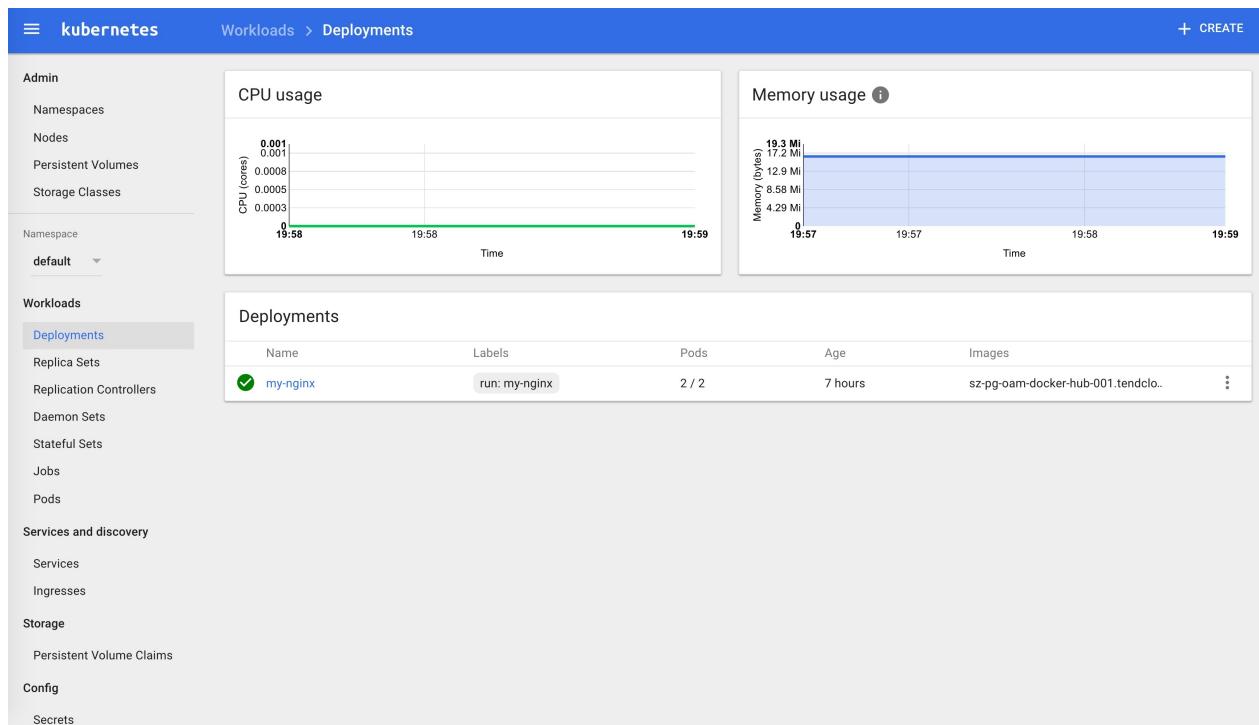


Figure: dashboard-heapster

访问 grafana

1. 通过 kube-apiserver 访问：

获取 monitoring-grafana 服务 URL

4.1.9 安装heapster插件

```
$ kubectl cluster-info
Kubernetes master is running at https://172.20.0.113:6443
Heapster is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/heapster
KubeDNS is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
monitoring-grafana is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
monitoring-influxdb is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

浏览器访问 URL :

```
http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
```

2. 通过 kubectl proxy 访问：

创建代理

```
$ kubectl proxy --address='172.20.0.113' --port=8086 --accept-hosts='^*$'
Starting to serve on 172.20.0.113:8086
```

浏览器访问

```
URL : http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
```

4.1.9 安装heapster插件

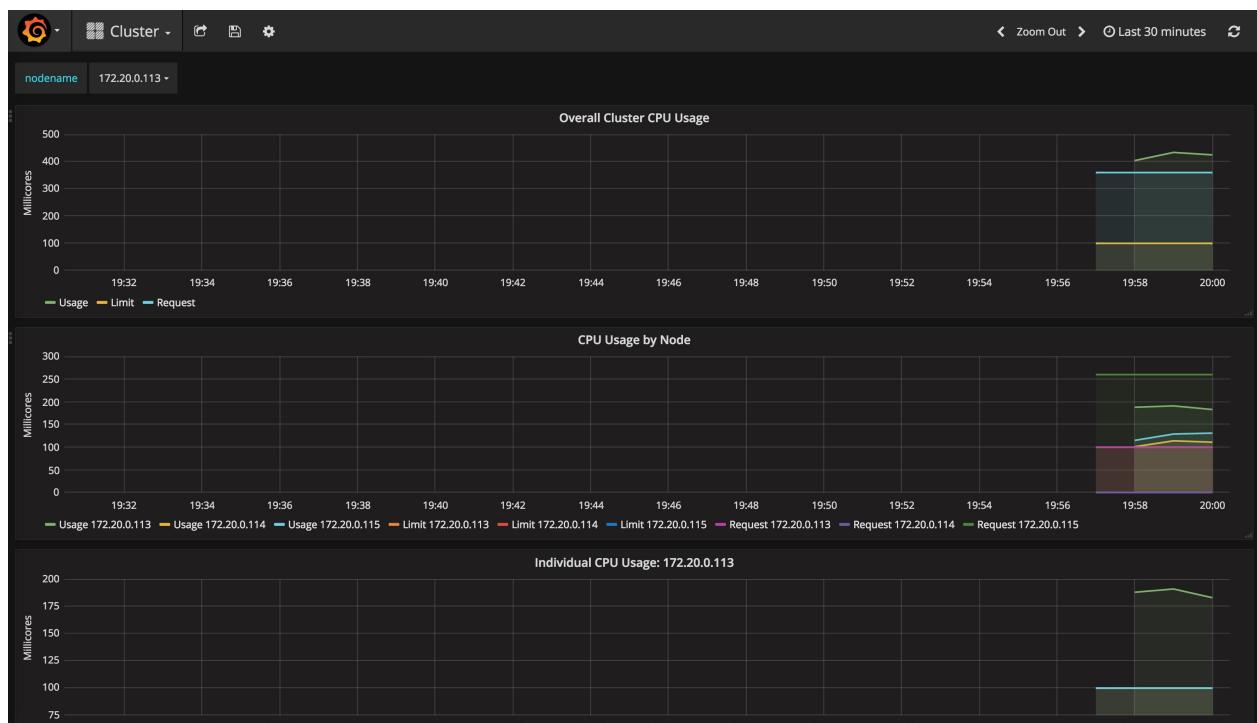


Figure: grafana

访问 influxdb admin UI

获取 influxdb http 8086 映射的 NodePort

```
$ kubectl get svc -n kube-system|grep influxdb
monitoring-influxdb      10.254.22.46      <nodes>          8086:32299/
TCP, 8083:30269/TCP    9m
```

通过 kube-apiserver 的非安全端口访问 influxdb 的 admin UI 界面：

```
http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-
system/services/monitoring-influxdb:8083/
```

在页面的“Connection Settings”的 Host 中输入 node IP，Port 中输入 8086 映射的 nodePort 如上面的 32299，点击“Save”即可（我的集群中的地址是 172.20.0.113:32299）：

4.1.9 安装heapster插件

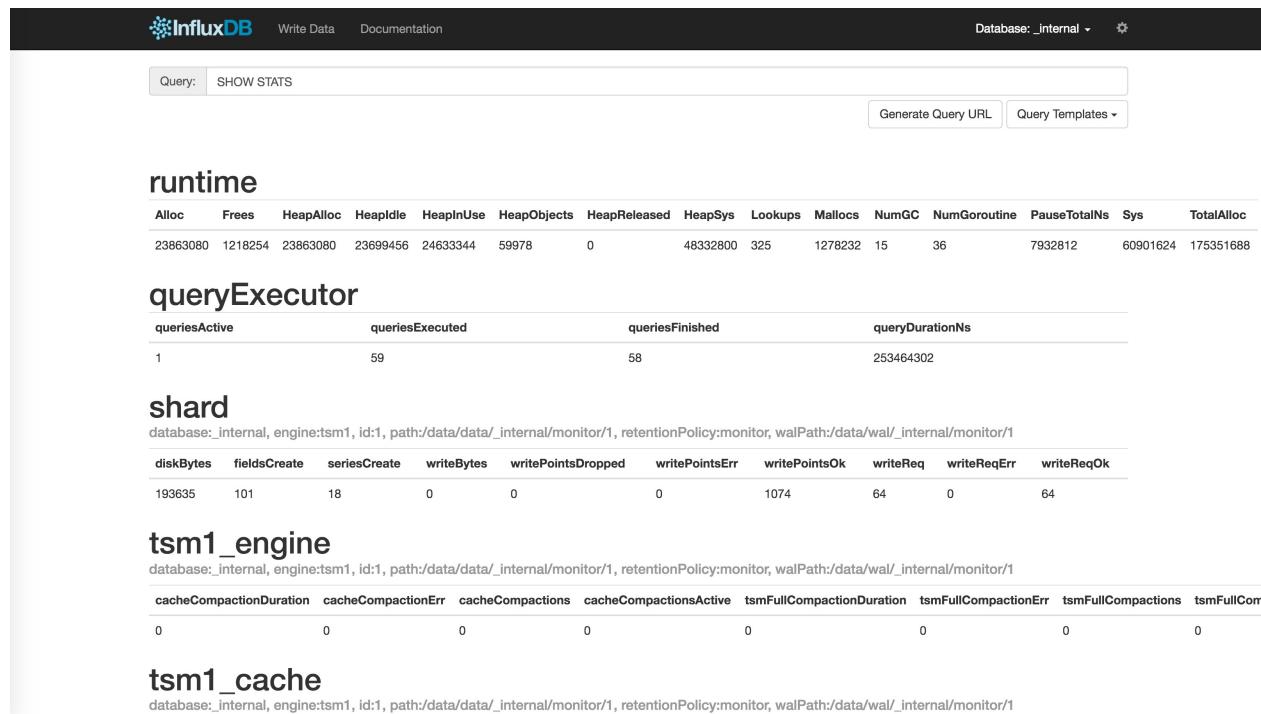


Figure: kubernetes-influxdb-heapster

for GitBook

update 2017-08-07 13:54:27

安装 EFK 插件

我们通过在每台 node 上部署一个以 DaemonSet 方式运行的 fluentd 来收集每台 node 上的日志。Fluentd 将 docker 日志目

录 `/var/lib/docker/containers` 和 `/var/log` 目录挂载到 Pod 中，然后 Pod 会在 node 节点的 `/var/log/pods` 目录中创建新的目录，可以区别不同的容器日志输出，该目录下有一个日志文件链接到 `/var/lib/docker/contianers` 目录下的容器日志输出。

官方文件目录：`cluster/addons/fluentd-elasticsearch`

```
$ ls *.yaml
es-controller.yaml  es-service.yaml  fluentd-es-ds.yaml  kibana-
controller.yaml  kibana-service.yaml  efk-rbac.yaml
```

同样 EFK 服务也需要一个 `efk-rbac.yaml` 文件，配置 serviceaccount 为 `efk`。

已经修改好的 yaml 文件见：[EFK](#)

配置 `es-controller.yaml`

```
$ diff es-controller.yaml.orig es-controller.yaml
24c24
<      - image: gcr.io/google_containers/elasticsearch:v2.4.1-2
---
>      - image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
elasticsearch:v2.4.1-2
```

配置 `es-service.yaml`

无需配置；

配置 `fluentd-es-ds.yaml`

```
$ diff fluentd-es-ds.yaml.orig fluentd-es-ds.yaml
26c26
<           image: gcr.io/google_containers/fluentd-elasticsearch:
1.22
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
fluentd-elasticsearch:1.22
```

配置 kibana-controller.yaml

```
$ diff kibana-controller.yaml.orig kibana-controller.yaml
22c22
<           image: gcr.io/google_containers/kibana:v4.6.1-1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
kibana:v4.6.1-1
```

给 Node 设置标签

定义 DaemonSet fluentd-es-v1.22 时设置了 nodeSelector
beta.kubernetes.io/fluentd-ds-ready=true，所以需要在期望运行 fluentd 的 Node 上设置该标签；

```
$ kubectl get nodes
NAME      STATUS     AGE      VERSION
172.20.0.113   Ready     1d      v1.6.0

$ kubectl label nodes 172.20.0.113 beta.kubernetes.io/fluentd-ds
-ready=true
node "172.20.0.113" labeled
```

给其他两台node打上同样的标签。

执行定义文件

```
$ kubectl create -f .
serviceaccount "efk" created
clusterrolebinding "efk" created
replicationcontroller "elasticsearch-logging-v1" created
service "elasticsearch-logging" created
daemonset "fluentd-es-v1.22" created
deployment "kibana-logging" created
service "kibana-logging" created
```

检查执行结果

```
$ kubectl get deployment -n kube-system|grep kibana
kibana-logging           1           1           1           1
                           2m

$ kubectl get pods -n kube-system|grep -E 'elasticsearch|fluentd|kibana'
elasticsearch-logging-v1-mlstp          1/1       Running   0
                                         1m
elasticsearch-logging-v1-nfbbf          1/1       Running   0
                                         1m
fluentd-es-v1.22-31sm0                1/1       Running   0
                                         1m
fluentd-es-v1.22-bpgqs               1/1       Running   0
                                         1m
fluentd-es-v1.22-qmn7h                1/1       Running   0
                                         1m
kibana-logging-1432287342-0gdng      1/1       Running   0
                                         1m

$ kubectl get service -n kube-system|grep -E 'elasticsearch|kibana'
elasticsearch-logging    10.254.77.62    <none>        9200/TCP
                           2m
kibana-logging          10.254.8.113     <none>        5601/TCP
                           2m
```

kibana Pod 第一次启动时会用较长时间(**10-20分钟**)来优化和 Cache 状态页面，可以 tailf 该 Pod 的日志观察进度：

```
$ kubectl logs kibana-logging-1432287342-0gdng -n kube-system -f
ELASTICSEARCH_URL=http://elasticsearch-logging:9200
server.basePath: /api/v1/proxy/namespaces/kube-system/services/k
ibana-logging
{"type":"log","@timestamp":"2017-04-12T13:08:06Z","tags":["info",
"optimize"],"pid":7,"message":"Optimizing and caching bundles fo
r kibana and statusPage. This may take a few minutes"}
{"type":"log","@timestamp":"2017-04-12T13:18:17Z","tags":["info",
"optimize"],"pid":7,"message":"Optimization of bundles for kiban
a and statusPage complete in 610.40 seconds"}
{"type":"log","@timestamp":"2017-04-12T13:18:17Z","tags":["statu
s","plugin:kibana@1.0.0","info"],"pid":7,"state":"green","messag
e":"Status changed from uninitialized to green - Ready","prevSta
te":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:18Z","tags":["statu
s","plugin:elasticsearch@1.0.0","info"],"pid":7,"state":"yellow",
"message":"Status changed from uninitialized to yellow - Waiting
 for Elasticsearch","prevState":"uninitialized","prevMsg":"unini
tialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:kbn_vislib_vis_types@1.0.0","info"],"pid":7,"state":"
green","message":"Status changed from uninitialized to green - R
eady","prevState":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:markdown_vis@1.0.0","info"],"pid":7,"state":"green",
"message":"Status changed from uninitialized to green - Ready","p
revState":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:metric_vis@1.0.0","info"],"pid":7,"state":"green","me
ssage":"Status changed from uninitialized to green - Ready","pre
vState":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:spyModes@1.0.0","info"],"pid":7,"state":"green","mess
age":"Status changed from uninitialized to green - Ready","prevS
tate":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:statusPage@1.0.0","info"],"pid":7,"state":"green","me
```

```
ssage": "Status changed from uninitialized to green - Ready", "prevState": "uninitialized", "prevMsg": "uninitialized"}  
{"type": "log", "@timestamp": "2017-04-12T13:18:19Z", "tags": ["status", "plugin:table_vis@1.0.0", "info"], "pid": 7, "state": "green", "message": "Status changed from uninitialized to green - Ready", "prevState": "uninitialized", "prevMsg": "uninitialized"}  
{"type": "log", "@timestamp": "2017-04-12T13:18:19Z", "tags": ["listing", "info"], "pid": 7, "message": "Server running at http://0.0.0.0:5601"}  
{"type": "log", "@timestamp": "2017-04-12T13:18:24Z", "tags": ["status", "plugin:elasticsearch@1.0.0", "info"], "pid": 7, "state": "yellow", "message": "Status changed from yellow to yellow - No existing Kibana index found", "prevState": "yellow", "prevMsg": "Waiting for Elasticsearch"}  
{"type": "log", "@timestamp": "2017-04-12T13:18:29Z", "tags": ["status", "plugin:elasticsearch@1.0.0", "info"], "pid": 7, "state": "green", "message": "Status changed from yellow to green - Kibana index ready", "prevState": "yellow", "prevMsg": "No existing Kibana index found"}
```

访问 kibana

1. 通过 kube-apiserver 访问：

获取 monitoring-grafana 服务 URL

```
$ kubectl cluster-info
Kubernetes master is running at https://172.20.0.113:6443
Elasticsearch is running at https://172.20.0.113:6443/api/v
1/proxy/namespaces/kube-system/services/elasticsearch-logging
Heapster is running at https://172.20.0.113:6443/api/v1/pro
xy/namespaces/kube-system/services/heapster
Kibana is running at https://172.20.0.113:6443/api/v1/proxy
/namespaces/kube-system/services/kibana-logging
KubeDNS is running at https://172.20.0.113:6443/api/v1/prox
y/namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at https://172.20.0.113:644
3/api/v1/proxy/namespaces/kube-system/services/kubernetes-da
shboard
monitoring-grafana is running at https://172.20.0.113:6443/
api/v1/proxy/namespaces/kube-system/services/monitoring-graf
ana
monitoring-influxdb is running at https://172.20.0.113:6443
/api/v1/proxy/namespaces/kube-system/services/monitoring-inf
luxdb
```

浏览器访问 URL :

```
https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-
system/services/kibana-logging/app/kibana
```

2. 通过 kubectl proxy 访问：

创建代理

```
$ kubectl proxy --address='172.20.0.113' --port=8086 --acce
pt-hosts='^*$'
Starting to serve on 172.20.0.113:8086
```

浏览器访问

```
URL : http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-
system/services/kibana-logging
```

4.1.10 安装EFK插件

在 Settings -> Indices 页面创建一个 index (相当于 mysql 中的一个 database) , 选中 Index contains time-based events , 使用默认的 logstash-* pattern , 点击 Create ;

可能遇到的问题

如果你在这里发现Create按钮是灰色的无法点击，且Time-filed name中没有选项， fluentd要读取 /var/log/containers/ 目录下的log日志，这些日志是从 /var/lib/docker/containers/\${CONTAINER_ID}/\${CONTAINER_ID}-json.log 链接过来的，查看你的docker配置， –log-dirver 需要设置为json-file格式，默认的可能是journald，参考[docker logging](#))。

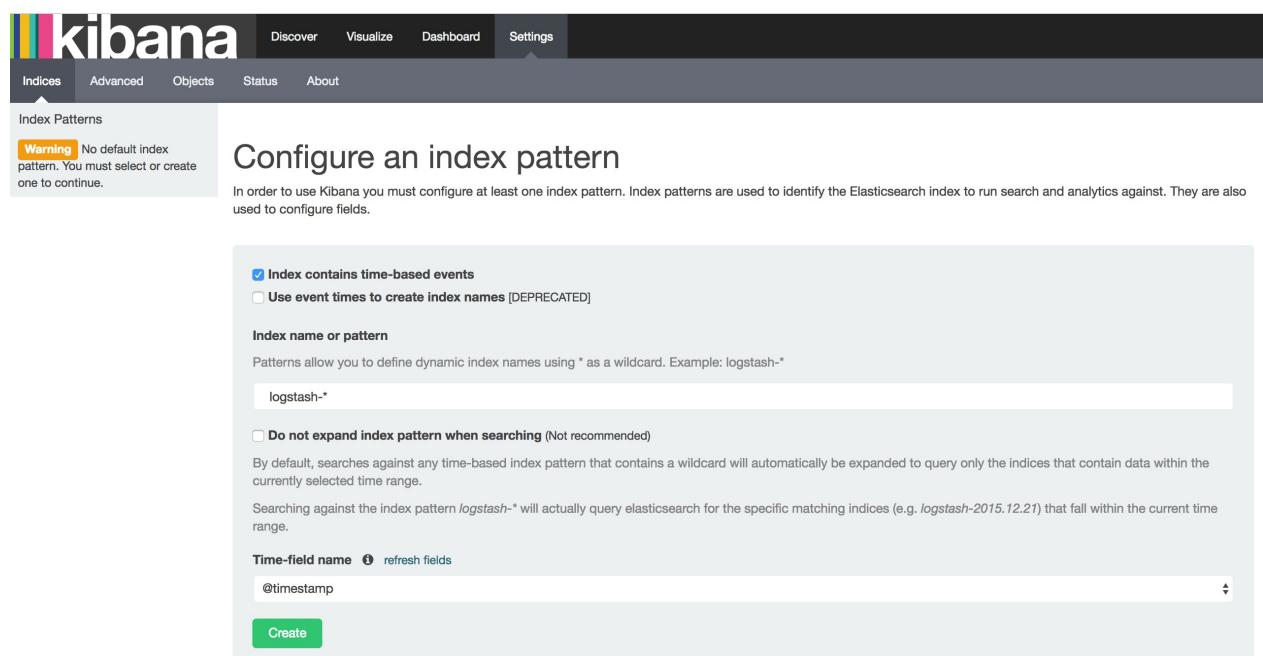
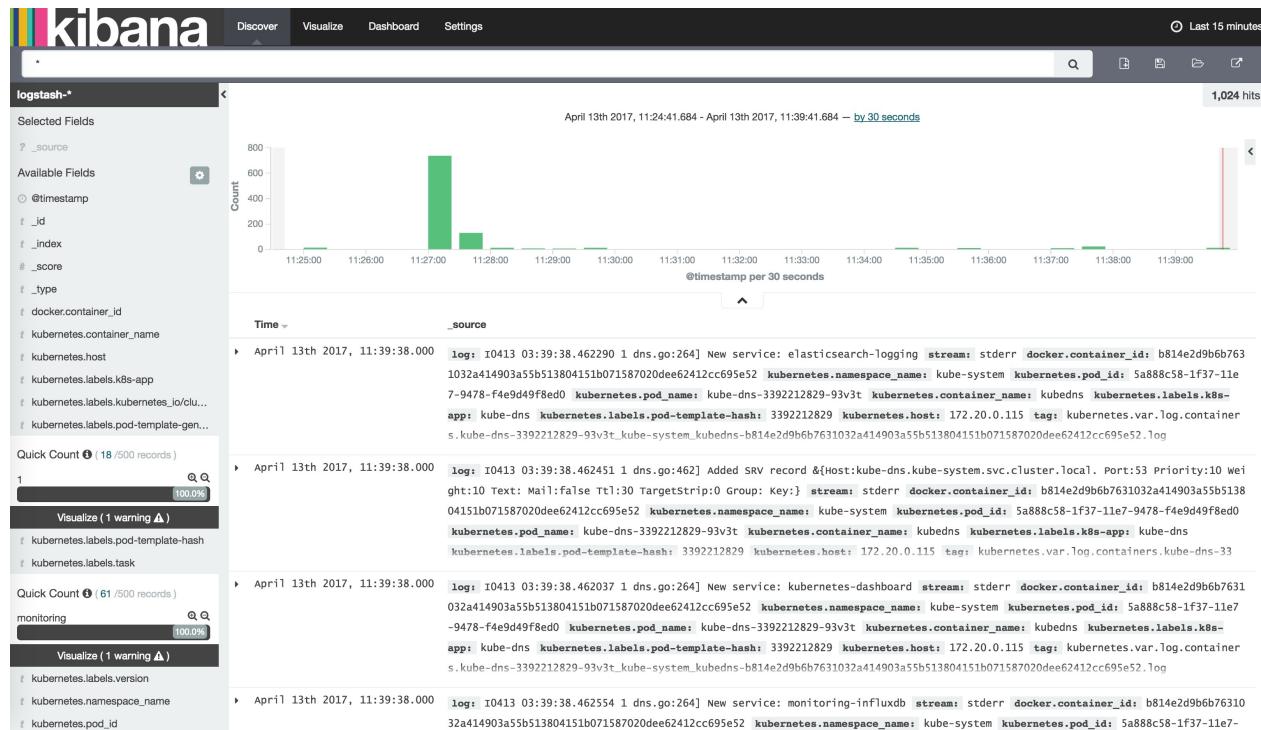


Figure: es-setting

创建Index后，可以在 Discover 下看到 ElasticSearch logging 中汇聚的日志；

4.1.10 安装EFK插件



服务发现与负载均衡

Kubernetes在设计之初就充分考虑了针对容器的服务发现与负载均衡机制，提供了Service资源，并通过kube-proxy配合cloud provider来适应不同的应用场景。随着kubernetes用户的激增，用户场景的不断丰富，又产生了一些新的负载均衡机制。目前，kubernetes中的负载均衡大致可以分为以下几种机制，每种机制都有其特定的应用场景：

- Service：直接用Service提供cluster内部的负载均衡，并借助cloud provider提供的LB提供外部访问
- Ingress Controller：还是用Service提供cluster内部的负载均衡，但是通过自定义LB提供外部访问
- Service Load Balancer：把load balancer直接跑在容器中，实现Bare Metal的Service Load Balancer
- Custom Load Balancer：自定义负载均衡，并替代kube-proxy，一般在物理部署Kubernetes时使用，方便接入公司已有的外部服务

Service

Service是对一组提供相同功能的Pods的抽象，并为它们提供一个统一的入口。借助Service，应用可以方便的实现服务发现与负载均衡，并实现应用的零宕机升级。Service通过标签来选取服务后端，一般配合Replication Controller或者Deployment来保证后端容器的正常运行。

Service有三种类型：

- ClusterIP：默认类型，自动分配一个仅cluster内部可以访问的虚拟IP
- NodePort：在ClusterIP基础上为Service在每台机器上绑定一个端口，这样就可以通过 `<NodeIP>:NodePort` 来访问该服务
- LoadBalancer：在NodePort的基础上，借助cloud provider创建一个外部的负载均衡器，并将请求转发到 `<NodeIP>:NodePort`

另外，也可以将已有的服务以Service的形式加入到Kubernetes集群中来，只需要在创建Service的时候不指定Label selector，而是在Service创建好后手动为其添加endpoint。

Ingress Controller

Service虽然解决了服务发现和负载均衡的问题，但它在使用上还是有一些限制，比如

- 只支持4层负载均衡，没有7层功能
- 对外访问的时候，NodePort类型需要在外部搭建额外的负载均衡，而LoadBalancer要求kubernetes必须跑在支持的cloud provider上面

Ingress就是为了解决这些限制而引入的新资源，主要用来将服务暴露到cluster外面，并且可以自定义服务的访问策略。比如想要通过负载均衡器实现不同子域名到不同服务的访问：

```
foo.bar.com --|           | -> foo.bar.com s1:80
              | 178.91.123.132 |
bar.foo.com --|           | -> bar.foo.com s2:80
```

可以这样来定义Ingress：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - backend:
              serviceName: s1
              servicePort: 80
    - host: bar.foo.com
      http:
        paths:
          - backend:
              serviceName: s2
              servicePort: 80
```

注意：Ingress本身并不会自动创建负载均衡器，cluster中需要运行一个ingress controller来根据Ingress的定义来管理负载均衡器。目前社区提供了nginx和gce的参考实现。

Traefik提供了易用的Ingress Controller，使用方法见<https://docs.traefik.io/user-guide/kubernetes/>。

Service Load Balancer

在Ingress出现以前，Service Load Balancer是推荐的解决Service局限性的方式。Service Load Balancer将haproxy跑在容器中，并监控service和endpoint的变化，通过容器IP对外提供4层和7层负载均衡服务。

社区提供的Service Load Balancer支持四种负载均衡协议：TCP、HTTP、HTTPS和SSL TERMINATION，并支持ACL访问控制。

Custom Load Balancer

虽然Kubernetes提供了丰富的负载均衡机制，但在实际使用的时候，还是会碰到一些复杂的场景是它不能支持的，比如：

- 接入已有的负载均衡设备
- 多租户网络情况下，容器网络和主机网络是隔离的，这样 kube-proxy 就不能正常工作

这个时候就可以自定义组件，并代替kube-proxy来做负载均衡。基本的思路是监控kubernetes中service和endpoints的变化，并根据这些变化来配置负载均衡器。比如weave flux、nginx plus、kube2haproxy等。

Endpoints

有几种情况下需要用到没有selector的service。

- 使用kubernetes集群外部的数据库时
- service中用到了其他namespace或kubernetes集群中的service
- 在kubernetes的工作负载与集群外的后端之间互相迁移

可以这样定义一个没有selector的service。

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

定义一个Endpoints来对应该service。

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

访问没有selector的service跟访问有selector的service时没有任何区别。

使用kubernetes时有一个很常见的需求，就是当数据库部署在kubernetes集群之外的时候，集群内的service如何访问数据库呢？当然你可以直接使用数据库的IP地址和端口号来直接访问，有没有什么优雅的方式呢？你需要用到 `ExternalName Service`。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
  ports:
    - port: 12345
```

这个例子中，在kubernetes集群内访问 my-service 实际上会重定向到 my.database.example.com:12345 这个地址。

参考资料

- <https://kubernetes.io/docs/concepts/services-networking/service/>
- <http://kubernetes.io/docs/user-guide/ingress/>
- <https://github.com/kubernetes/contrib/tree/master/service-loadbalancer>
- <https://www.nginx.com/blog/load-balancing-kubernetes-services-nginx-plus/>
- <https://github.com/weaveworks/flux>
- <https://github.com/AdoHe/kube2haproxy>

for GitBook update 2017-08-07 13:54:27

安装traefik ingress

Ingress简介

如果你还不了解，ingress是什么，可以先看下我翻译的Kubernetes官网上ingress的介绍[Kubernetes Ingress解析](#)。

理解Ingress

简单的说，ingress就是从kubernetes集群外访问集群的入口，将用户的URL请求转发到不同的service上。Ingress相当于nginx、apache等负载均衡方向代理服务器，其中还包括规则定义，即URL的路由信息，路由信息得的刷新由[Ingress controller](#)来提供。

理解Ingress Controller

Ingress Controller 实质上可以理解为是个监视器，Ingress Controller 通过不断地跟kubernetes API 打交道，实时的感知后端 service、pod 等变化，比如新增和减少 pod，service 增加与减少等；当得到这些变化信息后，Ingress Controller 再结合下文的 Ingress 生成配置，然后更新反向代理负载均衡器，并刷新其配置，达到服务发现的作用。

部署Traefik

介绍traefik

[Traefik](#)是一款开源的反向代理与负载均衡工具。它最大的优点是能够与常见的微服务系统直接整合，可以实现自动化动态配置。目前支持Docker, Swarm, Mesos/Marathon, Mesos, Kubernetes, Consul, Etcd, Zookeeper, BoltDB, Rest API 等等后端模型。

以下配置文件可以在[kubernetes-handbook](#)GitHub仓库中的[manifests/traefik-ingress/](#)目录下找到。

创建ingress-rbac.yaml

将用于service account验证。

4.2.1 安装Traefik ingress

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: ingress
  namespace: kube-system

---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: ingress
subjects:
- kind: ServiceAccount
  name: ingress
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

创建名为 **traefik-ingress** 的**ingress**，文件名**traefik.yaml**

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ingress
spec:
  rules:
    - host: traefik.nginx.io
      http:
        paths:
          - path: /
            backend:
              serviceName: my-nginx
              servicePort: 80
    - host: traefik.frontend.io
      http:
        paths:
          - path: /
            backend:
              serviceName: frontend
              servicePort: 80
```

这其中的 `backend` 中要配置 default namespace 中启动的 `service` 名字。`path` 就是 URL 地址后的路径，如 `traefik.frontend.io/path`，`service` 将会接受 `path` 这个路径，`host` 最好使用 `service-name.file1.file2.domain-name` 这种类似主机名称的命名方式，方便区分服务。

根据你自己环境中部署的 `service` 的名字和端口自行修改，有新 `service` 增加时，修改该文件后可以使用 `kubectl replace -f traefik.yaml` 来更新。

我们现在集群中已经有两个 `service` 了，一个是 `nginx`，另一个是官方的 `guestbook` 例子。

创建 Deployment

4.2.1 安装Traefik ingress

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: traefik-ingress-lb
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      terminationGracePeriodSeconds: 60
      hostNetwork: true
      restartPolicy: Always
      serviceAccountName: ingress
      containers:
        - image: traefik
          name: traefik-ingress-lb
          resources:
            limits:
              cpu: 200m
              memory: 30Mi
            requests:
              cpu: 100m
              memory: 20Mi
          ports:
            - name: http
              containerPort: 80
              hostPort: 80
            - name: admin
              containerPort: 8580
              hostPort: 8580
          args:
            - --web
            - --web.address=:8580
            - --kubernetes
```

4.2.1 安装Traefik ingress

注意我们这里用的是Deploy类型，没有限定该pod运行在哪个主机上。Traefik的端口是8580。

Traefik UI

```
apiVersion: v1
kind: Service
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  selector:
    k8s-app: traefik-ingress-lb
  ports:
    - name: web
      port: 80
      targetPort: 8580
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  rules:
    - host: traefik-ui.local
      http:
        paths:
          - path: /
            backend:
              serviceName: traefik-web-ui
              servicePort: web
```

配置完成后就可以启动traefik ingress了。

```
kubectl create -f .
```

我查看到traefik的pod在 172.20.0.115 这台节点上启动了。

4.2.1 安装Traefik ingress

访问该地址 `http://172.20.0.115:8580/` 将可以看到dashboard。

The screenshot shows the Kubernetes dashboard interface with the Traefik configuration. The top navigation bar includes icons for Providers and Health, and links for v1.2.3 / morbier, Documentation, and traefik.io. A sidebar on the left lists 'kubernetes' and other clusters. The main content area displays four sections for different domains:

- traefik-ui.local/**: Shows a table of routes and rules. One rule is defined: PathPrefix: /, Host: traefik-ui.local. Below the table are buttons for http, Backend:traefik-ui.local/, PassHostHeader, and Priority:1.
- traefik.frontend.io/**: Shows a table of routes and rules. One rule is defined: PathPrefix: /, Host: traefik.frontend.io. Below the table are buttons for http, Backend:traefik.frontend.io/, PassHostHeader, and Priority:1.
- traefik.nginx.io/**: Shows a table of routes and rules. One rule is defined: PathPrefix: /, Host: traefik.nginx.io. Below the table are buttons for http, Backend:traefik.nginx.io/, PassHostHeader, and Priority:1.
- traefik.nginx.io/**: Shows a table of servers and their URLs. Three servers are listed: frontend-1289468719-6l4v7 (http://172.30.60.11:80), frontend-1289468719-sfkvb (http://172.30.71.5:80), and frontend-1289468719-vg4zz (http://172.30.94.9:80). A 'Load Balancer: wrr' button is present.

Figure: kubernetes-dashboard

左侧黄色部分部分列出的是所有的rule，右侧绿色部分是所有的backend。

测试

在集群的任意一个节点上执行。假如现在我要访问nginx的"/"路径。

4.2.1 安装Traefik ingress

```
$ curl -H Host:traefik.nginx.io http://172.20.0.115/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

如果你需要在kubernetes集群以外访问就需要设置DNS，或者修改本机的hosts文件。

在其中加入：

```
172.20.0.115 traefik.nginx.io
172.20.0.115 traefik.frontend.io
```

所有访问这些地址的流量都会发送给172.20.0.115这台主机，就是我们启动traefik的主机。

4.2.1 安装Traefik ingress

Traefik会解析http请求header里的Host参数将流量转发给Ingress配置里的相应service。

修改hosts后就可以在kubernetes集群外访问以上两个service，如下图：



Figure: traefik-nginx

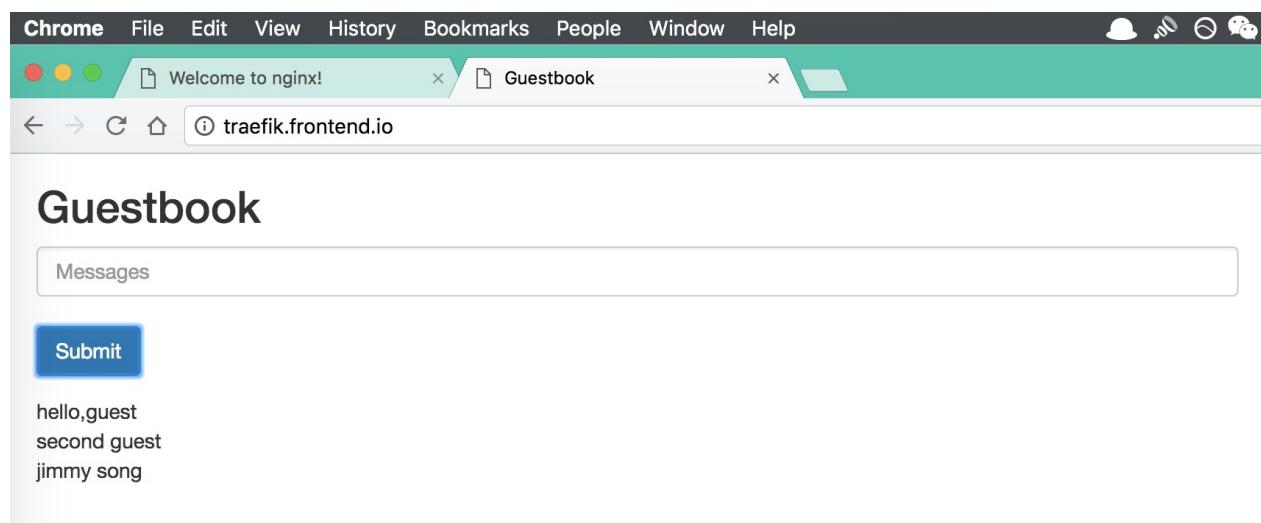


Figure: traefik-guestbook

参考

[Traefik-kubernetes 初试](#)

[Traefik简介](#)

[Guestbook example](#)

for GitBook update 2017-08-07 13:54:27

分布式负载测试

该教程描述如何在Kubernetes中进行分布式负载均衡测试，包括一个web应用、docker镜像和Kubernetes controllers/services。关于分布式负载测试的更多资料请查看[Distributed Load Testing Using Kubernetes](#)。

准备

不需要GCE及其他组件，你只需要有一个kubernetes集群即可。

如果你还没有kubernetes集群，可以参考[kubernetes-handbook](#)部署一个。

部署Web应用

本文中使用的镜像、kubernetes应用的yaml配置来自我的另一个项目，请参考：<https://github.com/rootsongjc/distributed-load-testing-using-kubernetes>

sample-webapp 目录下包含一个简单的web测试应用。我们将其构建为docker镜像，在kubernetes中运行。你可以自己构建，也可以直接用这个我构建好的镜像 `index.tenxcloud.com/jimmy/k8s-sample-webapp:latest`。

在kubernetes上部署sample-webapp。

```
$ git clone https://github.com/rootsongjc/distributed-load-testing-using-kubernetes.git
$ cd kubernetes-config
$ kubectl create -f sample-webapp-controller.yaml
$ kubectl create -f kubectl create -f sample-webapp-service.yaml
```

部署Locust的Controller和Service

locust-master 和 locust-work 使用同样的docker镜像，修改controller中 `spec.template.spec.containers.env` 字段中的value为你 `sample-webapp` service的名字。

```
- name: TARGET_HOST
  value: http://sample-webapp:8000
```

创建**Controller Docker**镜像（可选）

`locust-master` 和 `locust-work` controller使用的都是 `locust-tasks` docker 镜像。你可以直接下载 `gcr.io/cloud-solutions-images/locust-tasks`，也可以自己编译。自己编译大概要花几分钟时间，镜像大小为820M。

```
$ docker build -t index.tenxcloud.com/jimmy/locust-tasks:latest
.
$ docker push index.tenxcloud.com/jimmy/locust-tasks:latest
```

注意：我使用的是时速云的镜像仓库。

每个controller的yaml的 `spec.template.spec.containers.image` 字段指定的是我的镜像：

```
image: index.tenxcloud.com/jimmy/locust-tasks:latest
```

部署**locust-master**

```
$ kubectl create -f locust-master-controller.yaml
$ kubectl create -f locust-master-service.yaml
```

部署**locust-worker**

Now deploy `locust-worker-controller` :

```
$ kubectl create -f locust-worker-controller.yaml
```

你可以很轻易的给work扩容，通过命令行方式：

```
```ba sh $ kubectl scale --replicas=20 replicationcontrollers locust-worker
```

当然你也可以通过WebUI : Dashboard - Workloads - Replication Controllers - \*\*ServiceName\*\* - Scale来扩容。

![dashboard-scale](../images/dashbaord-scale.jpg)

### 配置Traefik

参考[kubernetes的traefik ingress安装](<http://rootsongjc.github.io/blogs/traefik-ingress-installation/>)，在`ingress.yaml`中加入如下配置：

```
```Yaml
- host: traefik.locust.io
  http:
    paths:
      - path: /
        backend:
          serviceName: locust-master
          servicePort: 8089
```

然后执行 `kubectl replace -f ingress.yaml` 即可更新traefik。

通过Traefik的dashboard就可以看到刚增加的 `traefik.locust.io` 节点。

4.2.2 分布式负载测试

The screenshot shows the Traefik dashboard interface with three main sections:

- traefik.guestbook.io/**:
 - Route: / Rule: PathPrefix:/
 - Route: traefik.guestbook.io Rule: Host:traefik.guestbook.io
- Load Balancer: wrr**
- Server** table:

Server	URL	Weight
frontend-1289468719-4565s	http://172.30.94.9:80	1
frontend-1289468719-rf8rw	http://172.30.71.3:80	1
frontend-1289468719-s4m6p	http://172.30.94.10:80	1

traefik.locust.io/:

- Route: / Rule: PathPrefix:/
- Route: traefik.locust.io Rule: Host:traefik.locust.io

Load Balancer: wrr

Server table:

Server	URL	Weight
locust-master-p8vpn	http://172.30.94.3:8089	1

traefik.nginx.io/:

- Route: / Rule: PathPrefix:/
- Route: traefik.nginx.io Rule: Host:traefik.nginx.io

Load Balancer: wrr

Server table:

Server	URL	Weight
my-nginx-2096504489-6l1zs	http://172.30.71.11:80	1
my-nginx-2096504489-9jh91	http://172.30.71.10:80	1
my-nginx-2096504489-9s58t	http://172.30.94.8:80	1
my-nginx-2096504489-gt621	http://172.30.71.12:80	1
my-nginx-2096504489-mp4gt	http://172.30.94.11:80	1

Figure: *traefik-dashboard-locust*

执行测试

打开 `http://traefik.locust.io` 页面，点击 `Edit` 输入伪造的用户数和用户每秒发送的请求个数，点击 `Start Swarming` 就可以开始测试了。

4.2.2 分布式负载测试

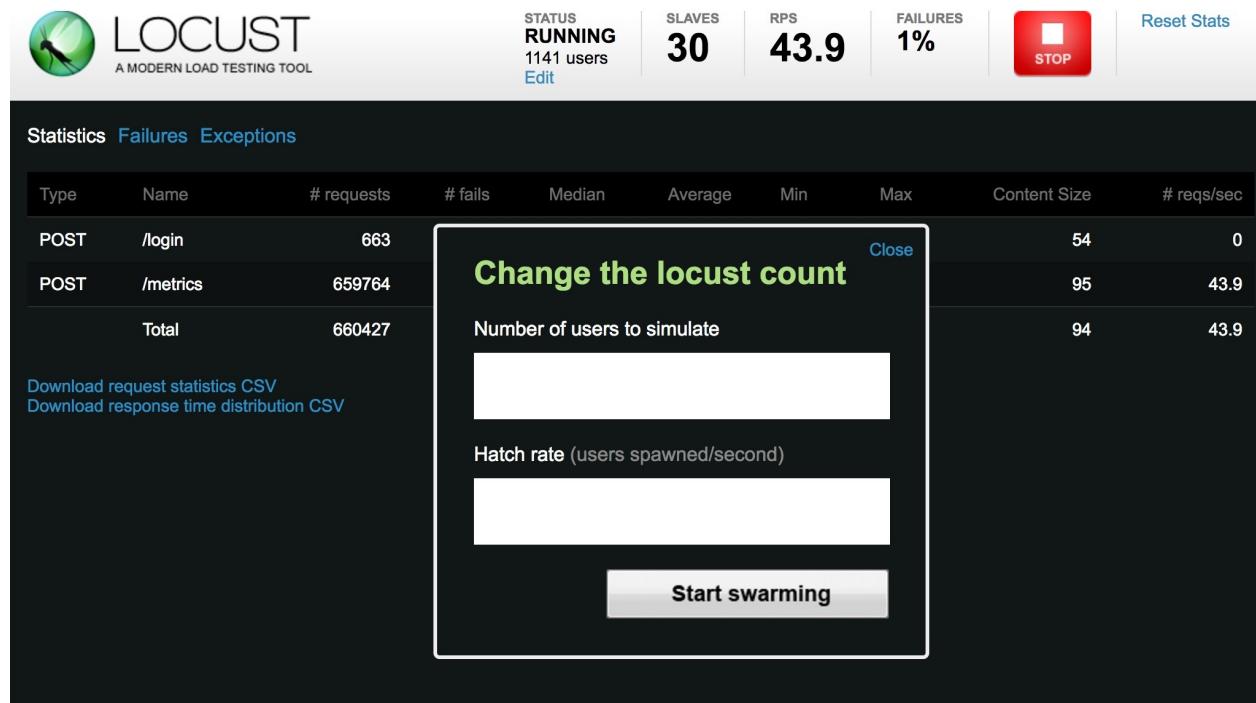


Figure: locust-start-swarming

在测试过程中调整 `sample-webapp` 的 pod 个数（默认设置了 1 个 pod），观察 pod 的负载变化情况。

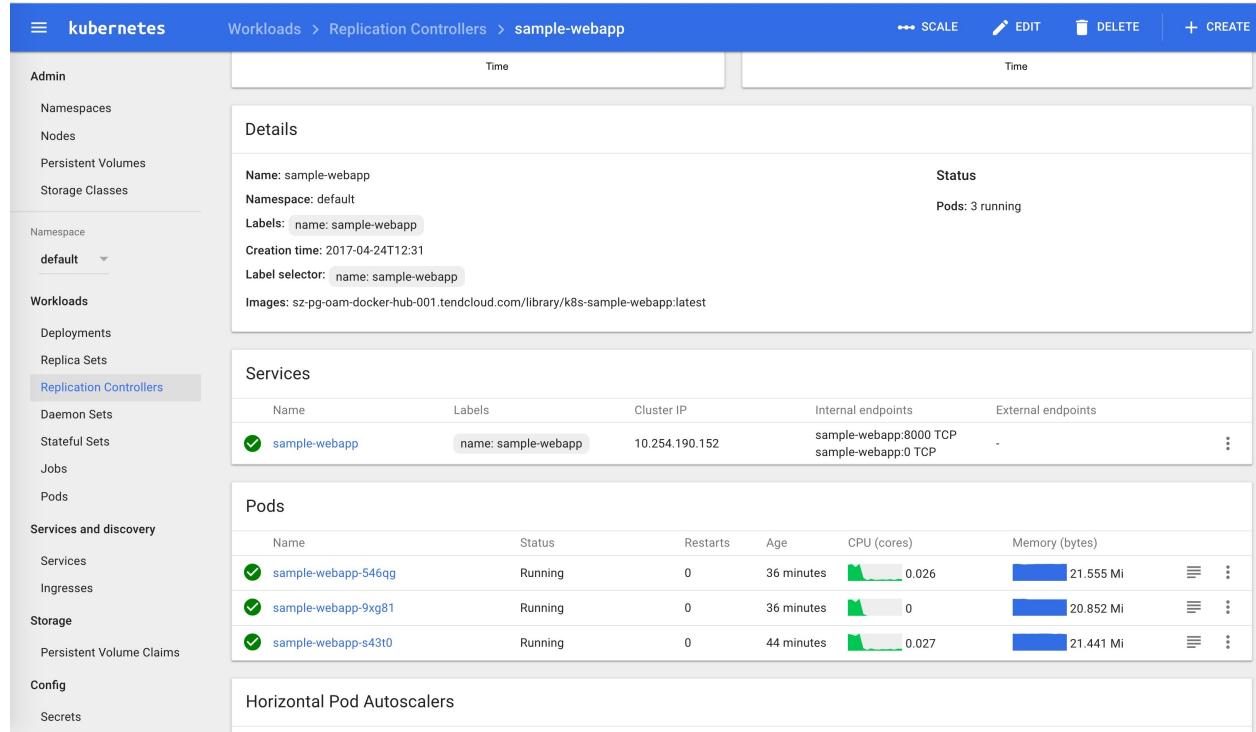


Figure: sample-webapp-rc

4.2.2 分布式负载测试

从一段时间的观察中可以看到负载被平均分配给了3个pod。

在locust的页面中可以实时观察也可以下载测试结果。

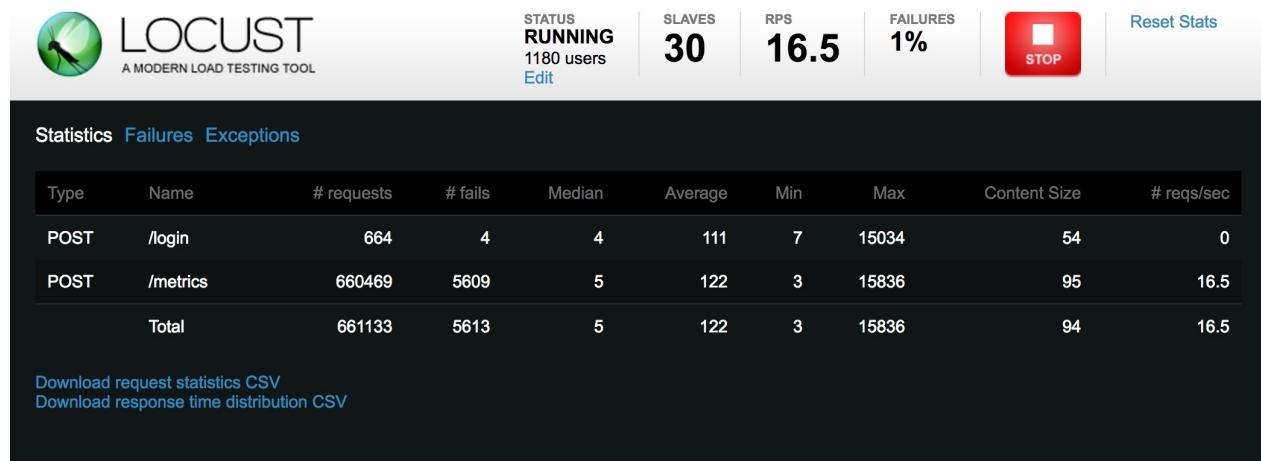


Figure: locust-dashboard

for GitBook update 2017-08-07 13:54:27

Kubernetes 网络和集群性能测试

准备

测试环境

在以下几种环境下进行测试：

- Kubernetes 集群 node 节点上通过 Cluster IP 方式访问
- Kubernetes 集群 内部通过 service 访问
- Kubernetes 集群 外部通过 traefik ingress 暴露的地址访问

测试地址

Cluster IP: 10.254.149.31

Service Port : 8000

Ingress Host : traefik.sample-webapp.io

测试工具

- [Locust](#)：一个简单易用的用户负载测试工具，用来测试 web 或其他系统能够同时处理的并发用户数。
- curl
- [kubemark](#)
- 测试程序：[sample-webapp](#)，源码见 Github [kubernetes](#) 的分布式负载测试

测试说明

通过向 sample-webapp 发送 curl 请求获取响应时间，直接 curl 后的结果为：

```
$ curl "http://10.254.149.31:8000/"
Welcome to the "Distributed Load Testing Using Kubernetes" sample web app
```

网络延迟测试

场景一、Kubernetes 集群 node 节点上通过 Cluster IP 访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://10.254.149.31:8000/"
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.000	0.003	0.003
2	0.000	0.002	0.002
3	0.000	0.002	0.002
4	0.000	0.002	0.002
5	0.000	0.002	0.002
6	0.000	0.002	0.002
7	0.000	0.002	0.002
8	0.000	0.002	0.002
9	0.000	0.002	0.002
10	0.000	0.002	0.002

平均响应时间：2ms

时间指标说明

单位：秒

time_connect：建立到服务器的 TCP 连接所用的时间

time_starttransfer：在发出请求之后，Web 服务器返回数据的第一个字节所用的时间

time_total：完成请求所用的时间

场景二、Kubernetes 集群内部通过 service 访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://sample-webapp:8000/"
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.004	0.006	0.006
2	0.004	0.006	0.006
3	0.004	0.006	0.006
4	0.004	0.006	0.006
5	0.004	0.006	0.006
6	0.004	0.006	0.006
7	0.004	0.006	0.006
8	0.004	0.006	0.006
9	0.004	0.006	0.006
10	0.004	0.006	0.006

平均响应时间：**6ms**

场景三、在公网上通过**traefik ingress**访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://traefik.sample-webapp.io" >>result
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.043	0.085	0.085
2	0.052	0.093	0.093
3	0.043	0.082	0.082
4	0.051	0.093	0.093
5	0.068	0.188	0.188
6	0.049	0.089	0.089
7	0.051	0.113	0.113
8	0.055	0.120	0.120
9	0.065	0.126	0.127
10	0.050	0.111	0.111

平均响应时间：**110ms**

测试结果

在这三种场景下的响应时间测试结果如下：

- Kubernetes 集群node节点上通过Cluster IP方式访问：2ms
- Kubernetes 集群内部通过service访问：6ms
- Kubernetes 集群外部通过traefik ingress暴露的地址访问：110ms

注意：执行测试的node节点/Pod与service所在的pod的距离（是否在同一台主机上），对前两个场景可能会有一定影响。

网络性能测试

网络使用flannel的vxlan模式。

使用iperf进行测试。

服务端命令：

```
iperf -s -p 12345 -i 1 -M
```

客户端命令：

```
iperf -c ${server-ip} -p 12345 -i 1 -t 10 -w 20K
```

场景一、主机之间

[ID]	Interval	Transfer	Bandwidth
[3]	0.0- 1.0 sec	598 MBytes	5.02 Gbits/sec
[3]	1.0- 2.0 sec	637 MBytes	5.35 Gbits/sec
[3]	2.0- 3.0 sec	664 MBytes	5.57 Gbits/sec
[3]	3.0- 4.0 sec	657 MBytes	5.51 Gbits/sec
[3]	4.0- 5.0 sec	641 MBytes	5.38 Gbits/sec
[3]	5.0- 6.0 sec	639 MBytes	5.36 Gbits/sec
[3]	6.0- 7.0 sec	628 MBytes	5.26 Gbits/sec
[3]	7.0- 8.0 sec	649 MBytes	5.44 Gbits/sec
[3]	8.0- 9.0 sec	638 MBytes	5.35 Gbits/sec
[3]	9.0-10.0 sec	652 MBytes	5.47 Gbits/sec
[3]	0.0-10.0 sec	6.25 GBytes	5.37 Gbits/sec

场景二、不同主机的Pod之间(使用flannel的vxlan模式)

[ID]	Interval	Transfer	Bandwidth
[3]	0.0- 1.0 sec	372 MBytes	3.12 Gbits/sec
[3]	1.0- 2.0 sec	345 MBytes	2.89 Gbits/sec
[3]	2.0- 3.0 sec	361 MBytes	3.03 Gbits/sec
[3]	3.0- 4.0 sec	397 MBytes	3.33 Gbits/sec
[3]	4.0- 5.0 sec	405 MBytes	3.40 Gbits/sec
[3]	5.0- 6.0 sec	410 MBytes	3.44 Gbits/sec
[3]	6.0- 7.0 sec	404 MBytes	3.39 Gbits/sec
[3]	7.0- 8.0 sec	408 MBytes	3.42 Gbits/sec
[3]	8.0- 9.0 sec	451 MBytes	3.78 Gbits/sec
[3]	9.0-10.0 sec	387 MBytes	3.25 Gbits/sec
[3]	0.0-10.0 sec	3.85 GBytes	3.30 Gbits/sec

场景三、Node与非同主机的Pod之间（使用flannel的vxlan模式）

[ID]	Interval	Transfer	Bandwidth
[3]	0.0- 1.0 sec	372 MBytes	3.12 Gbits/sec
[3]	1.0- 2.0 sec	420 MBytes	3.53 Gbits/sec
[3]	2.0- 3.0 sec	434 MBytes	3.64 Gbits/sec
[3]	3.0- 4.0 sec	409 MBytes	3.43 Gbits/sec
[3]	4.0- 5.0 sec	382 MBytes	3.21 Gbits/sec
[3]	5.0- 6.0 sec	408 MBytes	3.42 Gbits/sec
[3]	6.0- 7.0 sec	403 MBytes	3.38 Gbits/sec
[3]	7.0- 8.0 sec	423 MBytes	3.55 Gbits/sec
[3]	8.0- 9.0 sec	376 MBytes	3.15 Gbits/sec
[3]	9.0-10.0 sec	451 MBytes	3.78 Gbits/sec
[3]	0.0-10.0 sec	3.98 GBytes	3.42 Gbits/sec

场景四、不同主机的Pod之间（使用flannel的host-gw模式）

[ID]	Interval	Transfer	Bandwidth
[5]	0.0- 1.0 sec	530 MBytes	4.45 Gbits/sec
[5]	1.0- 2.0 sec	576 MBytes	4.84 Gbits/sec
[5]	2.0- 3.0 sec	631 MBytes	5.29 Gbits/sec
[5]	3.0- 4.0 sec	580 MBytes	4.87 Gbits/sec
[5]	4.0- 5.0 sec	627 MBytes	5.26 Gbits/sec
[5]	5.0- 6.0 sec	578 MBytes	4.85 Gbits/sec
[5]	6.0- 7.0 sec	584 MBytes	4.90 Gbits/sec
[5]	7.0- 8.0 sec	571 MBytes	4.79 Gbits/sec
[5]	8.0- 9.0 sec	564 MBytes	4.73 Gbits/sec
[5]	9.0-10.0 sec	572 MBytes	4.80 Gbits/sec
[5]	0.0-10.0 sec	5.68 GBytes	4.88 Gbits/sec

场景五、Node与非同主机的Pod之间（使用flannel的host-gw模式）

[ID]	Interval	Transfer	Bandwidth
[3]	0.0- 1.0 sec	570 MBytes	4.78 Gbits/sec
[3]	1.0- 2.0 sec	552 MBytes	4.63 Gbits/sec
[3]	2.0- 3.0 sec	598 MBytes	5.02 Gbits/sec
[3]	3.0- 4.0 sec	580 MBytes	4.87 Gbits/sec
[3]	4.0- 5.0 sec	590 MBytes	4.95 Gbits/sec
[3]	5.0- 6.0 sec	594 MBytes	4.98 Gbits/sec
[3]	6.0- 7.0 sec	598 MBytes	5.02 Gbits/sec
[3]	7.0- 8.0 sec	606 MBytes	5.08 Gbits/sec
[3]	8.0- 9.0 sec	596 MBytes	5.00 Gbits/sec
[3]	9.0-10.0 sec	604 MBytes	5.07 Gbits/sec
[3]	0.0-10.0 sec	5.75 GBytes	4.94 Gbits/sec

网络性能对比综述

使用 Flannel 的 **vxlan** 模式实现每个 pod 一个 IP 的方式，会比宿主机直接互联的网络性能损耗 30%~40%，符合网上流传的测试结论。而 flannel 的 host-gw 模式比起宿主机互连的网络性能损耗大约是 10%。

Vxlan 会有一个封包解包的过程，所以会对网络性能造成较大的损耗，而 host-gw 模式是直接使用路由信息，网络损耗小，关于 host-gw 的架构请访问 [Flannel host-gw architecture](#)。

Kubernetes 的性能测试

参考 [Kubernetes 集群性能测试](#) 中的步骤，对 kubernetes 的性能进行测试。

我的集群版本是 Kubernetes 1.6.0，首先克隆代码，将 kubernetes 目录复制到 `$GOPATH/src/k8s.io/` 下然后执行：

```

$ ./hack/generate-bindata.sh
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes
Generated bindata file : test/e2e/generated/bindata.go has 13498
test/e2e/generated/bindata.go lines of lovely automated artifacts
No changes in generated bindata file: pkg/generated/bindata.go
/usr/local/src/k8s.io/kubernetes
$ make WHAT="test/e2e/e2e.test"
...
+++ [0425 17:01:34] Generating bindata:
    test/e2e/generated/gobindata_util.go
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes/test/e2e/generated
/usr/local/src/k8s.io/kubernetes/test/e2e/generated
+++ [0425 17:01:34] Building go targets for linux/amd64:
    test/e2e/e2e.test
$ make ginkgo
+++ [0425 17:05:57] Building the toolchain targets:
    k8s.io/kubernetes/hack/cmd/teststale
    k8s.io/kubernetes/vendor/github.com/jteeuwen/go-bindata/go-bindata
+++ [0425 17:05:57] Generating bindata:
    test/e2e/generated/gobindata_util.go
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes/test/e2e/generated
/usr/local/src/k8s.io/kubernetes/test/e2e/generated
+++ [0425 17:05:58] Building go targets for linux/amd64:
    vendor/github.com/onsi/ginkgo/ginkgo

$ export KUBERNETES_PROVIDER=local
$ export KUBECTL_PATH=/usr/bin/kubectl
$ go run hack/e2e.go -v -test --test_args="--host=http://172.20
.0.113:8080 --ginkgo.focus=\[Feature:Performance\]" >>log.txt

```

测试结果

```
Apr 25 18:27:31.461: INFO: API calls latencies: {
```

4.2.3 网络和集群性能测试

```
"apicalls": [
    {
        "resource": "pods",
        "verb": "POST",
        "latency": {
            "Perc50": 2148000,
            "Perc90": 13772000,
            "Perc99": 14436000,
            "Perc100": 0
        }
    },
    {
        "resource": "services",
        "verb": "DELETE",
        "latency": {
            "Perc50": 9843000,
            "Perc90": 11226000,
            "Perc99": 12391000,
            "Perc100": 0
        }
    },
    ...
]

Apr 25 18:27:31.461: INFO: [Result:Performance] {
    "version": "v1",
    "dataItems": [
        {
            "data": {
                "Perc50": 2.148,
                "Perc90": 13.772,
                "Perc99": 14.436
            },
            "unit": "ms",
            "labels": {
                "Resource": "pods",
                "Verb": "POST"
            }
        },
        ...
    ]
}

2.857: INFO: Running AfterSuite actions on all node
Apr 26 10:35:32.857: INFO: Running AfterSuite actions on node 1
```

```
Ran 2 of 606 Specs in 268.371 seconds
SUCCESS! -- 2 Passed | 0 Failed | 0 Pending | 604 Skipped PASS

Ginkgo ran 1 suite in 4m28.667870101s
Test Suite Passed
```

从kubemark输出的日志中可以看到**API calls latencies**和**Performance**。

日志里显示，创建**90**个**pod**用时**40**秒以内，平均创建每个**pod**耗时**0.44**秒。

不同**type**的资源类型**API**请求耗时分布

Resource	Verb	50%	90%	99%
services	DELETE	8.472ms	9.841ms	38.226ms
endpoints	PUT	1.641ms	3.161ms	30.715ms
endpoints	GET	931μs	10.412ms	27.97ms
nodes	PATCH	4.245ms	11.117ms	18.63ms
pods	PUT	2.193ms	2.619ms	17.285ms

从 `log.txt` 日志中还可以看到更多详细请求的测试指标。

4.2.3 网络和集群性能测试

The screenshot shows the Kubernetes dashboard interface. On the left is a sidebar with navigation links: Admin, Namespaces, Nodes, Persistent Volumes, Storage Classes, Namespace (e2e-tests-load-30-nodepods-1-1), Workloads (Deployments, Replica Sets, **Replication Controllers**, Daemon Sets, Stateful Sets, Jobs, Pods), Services and discovery (Services, Ingresses, Storage, Persistent Volume Claims), Config, and Secrets. The main content area is titled "Replication Controllers" and displays a table with 13 rows. The columns are Name, Labels, Pods, Age, and Images. The table shows 12 entries under "load-small" and 1 entry under "load-medium". All pods are in a healthy state (5/5). The images used are gcr.io/google_containers/serve_hostname:v1.4. The table includes navigation controls at the top right: 1 - 10 of 13, <, >, and >>.

Name	Labels	Pods	Age	Images
load-medium-1	name: load-medium-1	30 / 30	44 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-1	name: load-small-1	5 / 5	43 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-10	name: load-small-10	5 / 5	42 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-11	name: load-small-11	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-12	name: load-small-12	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-2	name: load-small-2	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-3	name: load-small-3	5 / 5	38 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-4	name: load-small-4	5 / 5	39 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-5	name: load-small-5	5 / 5	41 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-6	name: load-small-6	5 / 5	38 seconds	gcr.io/google_containers/serve_hostname:v1.4

Figure: kubernetes-dashboard

注意事项

测试过程中需要用到docker镜像存储在GCE中，需要翻墙下载，我没看到哪里配置这个镜像的地址。该镜像副本已上传时速云：

用到的镜像有如下两个：

- gcr.io/google_containers/pause-amd64:3.0
- gcr.io/google_containers/serve_hostname:v1.4

时速云镜像地址：

- index.tenxcloud.com/jimmy/pause-amd64:3.0
- index.tenxcloud.com/jimmy/serve_hostname:v1.4

将镜像pull到本地后重新打tag。

Locust 测试

请求统计

4.2.3 网络和集群性能测试

Method	Name	# requests	# failures	Median response time	Average response time	Min response time
POST	/login	5070	78	59000	80551	11218
POST	/metrics	5114232	85879	63000	82280	29518
None	Total	5119302	85957	63000	82279	11218

响应时间分布

Name	# requests	50%	66%	75%	80%	90%	95%
POST /login	5070	59000	125000	140000	148000	160000	166000
POST /metrics	5114993	63000	127000	142000	149000	160000	166000
None Total	5120063	63000	127000	142000	149000	160000	166000

以上两个表格都是瞬时值。请求失败率在2%左右。

Sample-webapp起了48个pod。

Locust模拟10万用户，每秒增长100个。

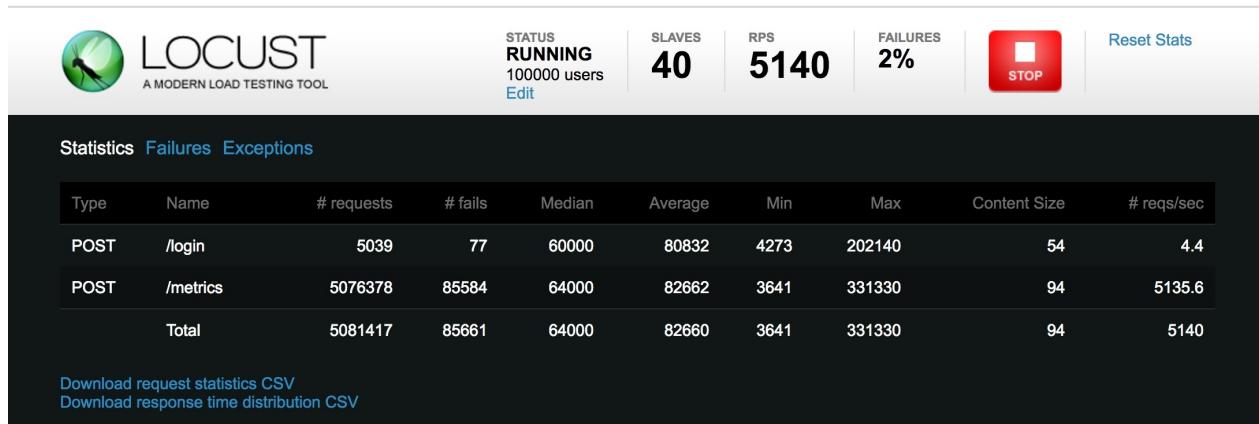


Figure: locust-test

关于Locust的使用请参考Github：<https://github.com/rootsongjc/distributed-load-testing-using-kubernetes>

参考

[基于 Python 的性能测试工具 locust \(与 LR 的简单对比\)](#)

[Locust docs](#)

[python 用户负载测试工具 : locust](#)

[Kubernetes 集群性能测试](#)

[CoreOS 是如何将 Kubernetes 的性能提高 10 倍的](#)

[Kubernetes 1.3 的性能和弹性 —— 2000 节点，60,0000 Pod 的集群](#)

[运用 Kubernetes 进行分布式负载测试](#)

[Kubemark User Guide](#)

[Flannel host-gw architecture](#)

for GitBook update 2017-08-07 13:54:27

边缘节点配置

前言

为了配置kubernetes中的traefik ingress的高可用，对于kubernetes集群以外只暴露一个访问入口，需要使用keepalived排除单点问题。本文参考了[kube-keepalived-vip](#)，但并没有使用容器方式安装，而是直接在node节点上安装。

定义

首先解释下什么叫边缘节点（Edge Node），所谓的边缘节点即集群内部用来向集群外暴露服务能力的节点，集群外部的服务通过该节点来调用集群内部的服务，边缘节点是集群内外交流的一个Endpoint。

边缘节点要考虑两个问题

- 边缘节点的高可用，不能有单点故障，否则整个kubernetes集群将不可用
- 对外的一致暴露端口，即只能有一个外网访问IP和端口

架构

为了满足边缘节点的以上需求，我们使用[keepalived](#)来实现。

在Kubernetes中添加了service的同时，在DNS中增加一个记录，这条记录需要跟ingress中的 host 字段相同，IP地址即VIP的地址，本示例中是 172.20.0.119，这样集群外部就可以通过service的DNS名称来访问服务了。

选择Kubernetes的三个node作为边缘节点，并安装keepalived，下图展示了边缘节点的配置，同时展示了向Kubernetes中添加服务的过程。

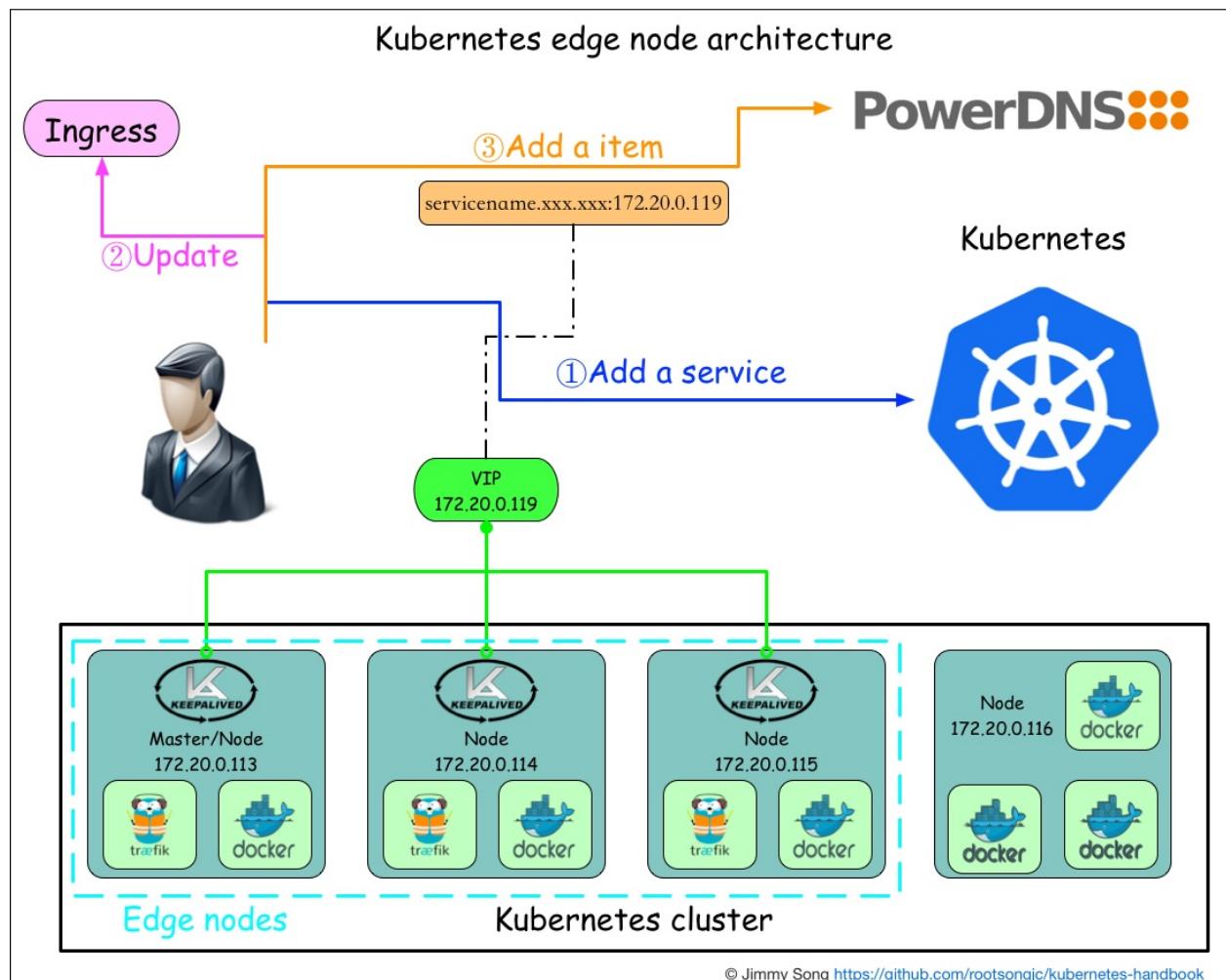


Figure: 边缘节点架构

准备

复用kubernetes测试集群的三台主机。

172.20.0.113

172.20.0.114

172.20.0.115

安装

使用keepalived管理VIP，VIP是使用IPVS创建的，IPVS已经成为linux内核的模块，不需要安装

LVS的工作原理请参考：<http://www.cnblogs.com/codebean/archive/2011/07/25/2116043.html>

不使用镜像方式安装了，直接手动安装，指定三个节点为边缘节点（Edge node）。

因为我们的测试集群一共只有三个node，所有要在三个node上都要安装keepalived和ipvsadm。

```
yum install keepalived ipvsadm
```

配置说明

需要对原先的traefik ingress进行改造，从以Deployment方式启动改成DaemonSet。还需要指定一个与node在同一网段的IP地址作为VIP，我们指定成172.20.0.119，配置keepalived前需要先保证这个IP没有被分配。。

- Traefik以DaemonSet的方式启动
- 通过nodeSelector选择边缘节点
- 通过hostPort暴露端口
- 当前VIP漂移到了172.20.0.115上
- Traefik根据访问的host和path配置，将流量转发到相应的service上

配置keepalived

参考基于keepalived 实现VIP转移，lvs，nginx的高可用，配置keepalived。

keepalived的官方配置文档见：<http://keepalived.org/pdf/UserGuide.pdf>

配置文件 /etc/keepalived/keepalived.conf 文件内容如下：

```
! Configuration File for keepalived

global_defs {
    notification_email {
        root@localhost
    }
    notification_email_from kaadmin@localhost
```

4.2.4 边缘节点配置

```
smtp_server 127.0.0.1
smtp_connect_timeout 30
router_id LVS_DEVEL
}

vrrp_instance VI_1 {
    state MASTER
    interface eth0
    virtual_router_id 51
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        172.20.0.119
    }
}

virtual_server 172.20.0.119 80{
    delay_loop 6
    lb_algo loadbalance
    lb_kind DR
    nat_mask 255.255.255.0
    persistence_timeout 0
    protocol TCP

    real_server 172.20.0.113 80{
        weight 1
        TCP_CHECK {
            connect_timeout 3
        }
    }
    real_server 172.20.0.114 80{
        weight 1
        TCP_CHECK {
            connect_timeout 3
        }
    }
}
```

4.2.4 边缘节点配置

```
real_server 172.20.0.115 80{
    weight 1
    TCP_CHECK {
        connect_timeout 3
    }
}
```

Realserver 的IP和端口即traefik供外网访问的IP和端口。

将以上配置分别拷贝到另外两台node的 /etc/keepalived 目录下。

我们使用转发效率最高的 lb_kind DR 直接路由方式转发，使用TCP_CHECK来检测real_server的health。

启动**keepalived**

```
systemctl start keepalived
```

三台node都启动了keepalived后，观察eth0的IP，会在三台node的某一台上发现一个VIP是172.20.0.119。

```
$ ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether f4:e9:d4:9f:6b:a0 brd ff:ff:ff:ff:ff:ff
    inet 172.20.0.115/17 brd 172.20.127.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet 172.20.0.119/32 scope global eth0
        valid_lft forever preferred_lft forever
```

关掉拥有这个VIP主机上的keepalived，观察VIP是否漂移到了另外两台主机的其中之一上。

改造Traefik

4.2.4 边缘节点配置

在这之前我们启动的traefik使用的是deployment，只启动了一个pod，无法保证高可用（即需要将pod固定在某一台主机上，这样才能对外提供一个唯一的访问地址），现在使用了keepalived就可以通过VIP来访问traefik，同时启动多个traefik的pod保证高可用。

配置文件 `traefik.yaml` 内容如下：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: traefik-ingress-lb
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      terminationGracePeriodSeconds: 60
      hostNetwork: true
      restartPolicy: Always
      serviceAccountName: ingress
      containers:
        - image: traefik
          name: traefik-ingress-lb
          resources:
            limits:
              cpu: 200m
              memory: 30Mi
            requests:
              cpu: 100m
              memory: 20Mi
          ports:
            - name: http
              containerPort: 80
              hostPort: 80
            - name: admin
```

4.2.4 边缘节点配置

```
        containerPort: 8580
        hostPort: 8580
      args:
        - --web
        - --web.address=:8580
        - --kubernetes
      nodeSelector:
        edgenode: "true"
```

注意，我们使用了 `nodeSelector` 选择边缘节点来调度traefik-ingress-lb运行在它上面，所有你需要使用：

```
kubectl label nodes 172.20.0.113 edgenode=true
kubectl label nodes 172.20.0.114 edgenode=true
kubectl label nodes 172.20.0.115 edgenode=true
```

给三个node打标签。

查看DaemonSet的启动情况：

```
$ kubectl -n kube-system get ds
  NAME           DESIRED   CURRENT   READY   UP-TO-DATE   AGE
  AVAILABLE     NODE-SELECTOR
  traefik-ingress-lb   3         3         3       3           2h
  3             edgenode=true
```

现在就可以在外网通过172.20.0.119:80来访问到traefik ingress了。

参考

[kube-keepalived-vip](#)

<http://www.keepalived.org/>

[keepalived工作原理与配置说明](#)

[LVS简介及使用](#)

[基于keepalived 实现VIP转移，lvs，nginx的高可用](#)

4.2.4 边缘节点配置

for GitBook update 2017-08-07 13:54:27

运维管理

for GitBook update 2017-08-07 13:54:27

服务滚动升级

当有镜像发布新版本，新版本服务上线时如何实现服务的滚动和平滑升级？

如果你使用**ReplicationController**创建的pod可以使用 `kubectl rollingupdate` 命令滚动升级，如果使用的是**Deployment**创建的Pod可以直接修改yaml文件后执行 `kubectl apply` 即可。

Deployment已经内置了RollingUpdate strategy，因此不用再调用 `kubectl rollingupdate` 命令，升级的过程是先创建新版的pod将流量导入到新pod上后销毁原来的旧的pod。

Rolling Update适用于 Deployment 、 Replication Controller ，官方推荐使用Deployment而不再使用Replication Controller。

使用ReplicationController时的滚动升级请参考官网说

明：<https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/>

ReplicationController与Deployment的关系

ReplicationController和Deployment的RollingUpdate命令有些不同，但是实现的机制是一样的，关于这两个kind的关系我引用了[ReplicationController与Deployment的区别](#)中的部分内容如下，详细区别请查看原文。

ReplicationController

Replication Controller为Kubernetes的一个核心内容，应用托管到Kubernetes之后，需要保证应用能够持续的运行，Replication Controller就是这个保证的key，主要的功能如下：

- 确保pod数量：它会确保Kubernetes中有指定数量的Pod在运行。如果少于指定数量的pod，Replication Controller会创建新的，反之则会删除掉多余的以保证Pod数量不变。
- 确保pod健康：当pod不健康，运行出错或者无法提供服务时，Replication Controller也会杀死不健康的pod，重新创建新的。

- 弹性伸缩：在业务高峰或者低峰期的时候，可以通过Replication Controller动态的调整pod的数量来提高资源的利用率。同时，配置相应的监控功能（Horizontal Pod Autoscaler），会定时自动从监控平台获取Replication Controller关联pod的整体资源使用情况，做到自动伸缩。
- 滚动升级：滚动升级为一种平滑的升级方式，通过逐步替换的策略，保证整体系统的稳定，在初始化升级的时候就可以及时发现和解决问题，避免问题不断扩大。

Deployment

Deployment同样为Kubernetes的一个核心内容，主要职责同样是为了保证pod的数量和健康，90%的功能与Replication Controller完全一样，可以看做新一代的Replication Controller。但是，它又具备了Replication Controller之外的新特性：

- Replication Controller全部功能：Deployment继承了上面描述的Replication Controller全部功能。
- 事件和状态查看：可以查看Deployment的升级详细进度和状态。
- 回滚：当升级pod镜像或者相关参数的时候发现问题，可以使用回滚操作回滚到上一个稳定的版本或者指定的版本。
- 版本记录：每一次对Deployment的操作，都能保存下来，给予后续可能的回滚使用。
- 暂停和启动：对于每一次升级，都能够随时暂停和启动。
- 多种升级方案：Recreate：删除所有已存在的pod,重新创建新的；
RollingUpdate：滚动升级，逐步替换的策略，同时滚动升级时，支持更多的附加参数，例如设置最大不可用pod数量，最小升级间隔时间等等。

创建测试镜像

我们来创建一个特别简单的web服务，当你访问网页时，将输出一句版本信息。通过区分这句版本信息输出我们就可以断定升级是否完成。

所有配置和代码见[manifests/test/rolling-update-test](#)目录。

Web服务的代码[main.go](#)

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func sayhello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "This is version 1.") //这个写入到w的是输出到客户
    端的
}

func main() {
    http.HandleFunc("/", sayhello) //设置访问的路由
    log.Println("This is version 1.")
    err := http.ListenAndServe(":9090", nil) //设置监听的端口
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

创建Dockerfile

```
FROM alpine:3.5
MAINTAINER Jimmy Song<rootsongjc@gmail.com>
ADD hellov2 /
ENTRYPOINT ["/hellov2"]
```

注意修改添加的文件的名称。

创建Makefile

修改镜像仓库的地址为你自己的私有镜像仓库地址。

修改 Makefile 中的 TAG 为新的版本号。

```
all: build push clean
.PHONY: build push clean

TAG = v1

# Build for linux amd64
build:
    GOOS=linux GOARCH=amd64 go build -o hello${TAG} main.go
    docker build -t sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:${TAG} .

# Push to tenxcloud
push:
    docker push sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:${TAG}

# Clean
clean:
    rm -f hello${TAG}
```

编译

```
make all
```

分别修改main.go中的输出语句、Dockerfile中的文件名称和Makefile中的TAG，创建两个版本的镜像。

测试

我们使用Deployment部署服务来测试。

配置文件 rolling-update-test.yaml :

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: rolling-update-test
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: rolling-update-test
    spec:
      containers:
        - name: rolling-update-test
          image: sz-pg-oam-docker-hub-001.tendcloud.com/library/he
llo:v1
      ports:
        - containerPort: 9090
    ---
apiVersion: v1
kind: Service
metadata:
  name: rolling-update-test
  labels:
    app: rolling-update-test
spec:
  ports:
    - port: 9090
      protocol: TCP
      name: http
  selector:
    app: rolling-update-test
```

部署**service**

```
kubectl create -f rolling-update-test.yaml
```

修改**traefik ingress**配置

4.3.1 服务滚动升级

在 `ingress.yaml` 文件中增加新service的配置。

```
- host: rolling-update-test.traefik.io
  http:
    paths:
      - path: /
        backend:
          serviceName: rolling-update-test
          servicePort: 9090
```

修改本地的host配置，增加一条配置：

```
172.20.0.119 rolling-update-test.traefik.io
```

注意：172.20.0.119是我们之前使用keepalived创建的VIP。

打开浏览器访问<http://rolling-update-test.traefik.io>将会看到以下输出：

```
This is version 1.
```

滚动升级

只需要将 `rolling-update-test.yaml` 文件中的 `image` 改成新版本的镜像名，然后执行：

```
kubectl apply -f rolling-update-test.yaml
```

也可以参考[Kubernetes Deployment Concept](#)中的方法，直接设置新的镜像。

```
kubectl set image deployment/rolling-update-test rolling-update-test=sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:v2
```

或者使用 `kubectl edit deployment/rolling-update-test` 修改镜像名称后保存。

使用以下命令查看升级进度：

```
kubectl rollout status deployment/rolling-update-test
```

升级完成后在浏览器中刷新<http://rolling-update-test.traefik.io>将会看到以下输出：

```
This is version 2.
```

说明滚动升级成功。

使用**ReplicationController**创建的Pod如何RollingUpdate

以上讲解使用**Deployment**创建的Pod的RollingUpdate方式，那么如果使用传统的**ReplicationController**创建的Pod如何Update呢？

举个例子：

```
$ kubectl -n spark-cluster rolling-update zeppelin-controller --image sz-pg-oam-docker-hub-001.tendcloud.com/library/zeppelin:0.7.1
Created zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b
Scaling up zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b from 0 to 1, scaling down zeppelin-controller from 1 to 0 (keep 1 pods available, don't exceed 2 pods)
Scaling zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b up to 1
Scaling zeppelin-controller down to 0
Update succeeded. Deleting old controller: zeppelin-controller
Renaming zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b to zeppelin-controller
replicationcontroller "zeppelin-controller" rolling updated
```

只需要指定新的镜像即可，当然你可以配置RollingUpdate的策略。

参考

[Rolling update机制解析](#)

[Running a Stateless Application Using a Deployment](#)

[Simple Rolling Update](#)

[使用kubernetes的deployment进行RollingUpdate](#)

for GitBook update 2017-08-07 13:54:27

应用日志收集

前言

在进行日志收集的过程中，我们首先想到的是使用Logstash，因为它是ELK stack中的重要成员，但是在测试过程中发现，Logstash是基于JDK的，在没有产生日志的情况下单纯启动Logstash就大概要消耗**500M**内存，在每个Pod中都启动一个日志收集组件的情况下，使用logstash有点浪费系统资源，经人推荐我们选择使用**Filebeat**替代，经测试单独启动Filebeat容器大约会消耗**12M**内存，比起logstash相当轻量级。

方案选择

Kubernetes官方提供了EFK的日志收集解决方案，但是这种方案并不适合所有的业务场景，它本身就有一些局限性，例如：

- 所有日志都必须是out前台输出，真实业务场景中无法保证所有日志都在前台输出
- 只能有一个日志输出文件，而真实业务场景中往往有多个日志输出文件
- Fluentd并不是常用的日志收集工具，我们更习惯用logstash，现使用filebeat替代
- 我们已经有自己的ELK集群且有专人维护，没有必要再在kubernetes上做一个日志收集服务

基于以上几个原因，我们决定使用自己的ELK集群。

Kubernetes集群中的日志收集解决方案

4.3.2 应用日志收集

编号	方案	优点	缺点
1	每个app的镜像中都集成日志收集组件	部署方便， kubernetes 的yaml文件无须特别配置，可以为每个app自定义日志收集配置	强耦合，不方便应用和日志收集组件升级和维护且会导致镜像过大
2	单独创建一个日志收集组件跟app的容器一起运行在同一个pod中	低耦合，扩展性强，方便维护和升级	需要对 kubernetes 的yaml文件进行单独配置，略显繁琐
3	将所有的Pod的日志都挂载到宿主机上，每台主机上单独起一个日志收集Pod	完全解耦，性能最高，管理起来最方便	需要统一日志收集规则，目录和输出方式

综合以上优缺点，我们选择使用方案二。

该方案在扩展性、个性化、部署和后期维护方面都能做到均衡，因此选择该方案。

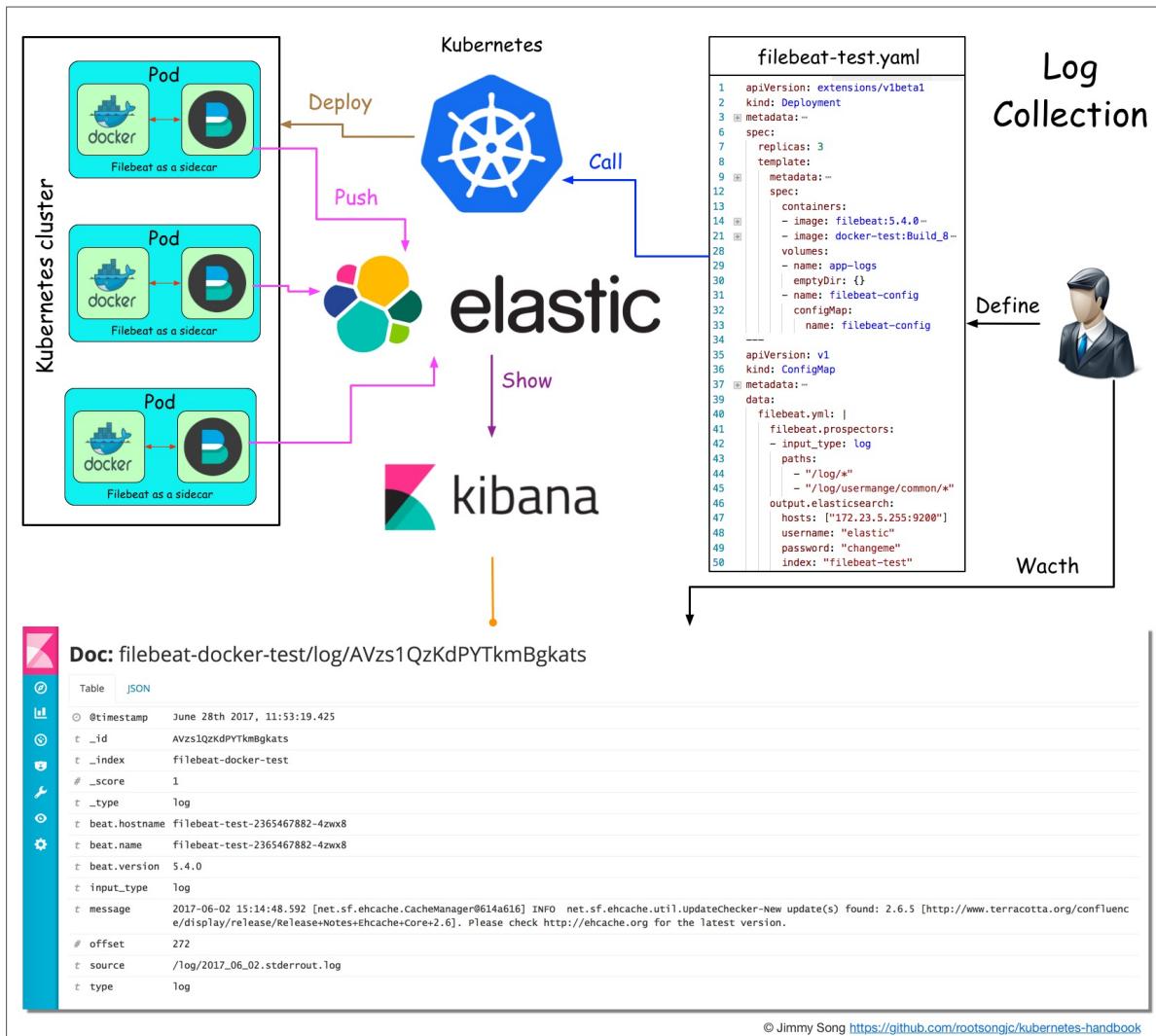


Figure: logstash日志收集架构图

我们创建了自己的filebeat镜像。创建过程和使用方式见
<https://github.com/rootsongjc/docker-images>

镜像地址：`index.tenxcloud.com/jimmy/filebeat:5.4.0`

测试

我们部署一个应用filebeat来收集日志的功能测试。

创建应用yaml文件 `filebeat-test.yaml`。

```

apiVersion: extensions/v1beta1
kind: Deployment

```

```
metadata:
  name: filebeat-test
  namespace: default
spec:
  replicas: 3
  template:
    metadata:
      labels:
        k8s-app: filebeat-test
    spec:
      containers:
        - image: sz-pg-oam-docker-hub-001.tendcloud.com/library/filebeat:5.4.0
          name: filebeat
          volumeMounts:
            - name: app-logs
              mountPath: /log
            - name: filebeat-config
              mountPath: /etc/filebeat/
        - image: sz-pg-oam-docker-hub-001.tendcloud.com/library/analytics-docker-test:Build_8
          name : app
          ports:
            - containerPort: 80
          volumeMounts:
            - name: app-logs
              mountPath: /usr/local/TalkingData/logs
          volumes:
            - name: app-logs
              emptyDir: {}
            - name: filebeat-config
              configMap:
                name: filebeat-config
---
apiVersion: v1
kind: Service
metadata:
  name: filebeat-test
  labels:
    app: filebeat-test
```

```
spec:  
  ports:  
    - port: 80  
      protocol: TCP  
      name: http  
  selector:  
    run: filebeat-test  
---  
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: filebeat-config  
data:  
  filebeat.yml: |  
    filebeat.prospectors:  
      - input_type: log  
        paths:  
          - "/log/*"  
          - "/log/usermange/common/*"  
    output.elasticsearch:  
      hosts: ["172.23.5.255:9200"]  
      username: "elastic"  
      password: "changeme"  
      index: "filebeat-docker-test"
```

说明

该文件中包含了配置文件filebeat的配置文件的ConfigMap，因此不需要再定义环境变量。

当然你也可以不同ConfigMap，通过传统的传递环境变量的方式来配置filebeat。

例如对filebeat的容器进行如下配置：

```

  containers:
    - image: sz-pg-oam-docker-hub-001.tendcloud.com/library/filebeat:5.4.0
      name: filebeat
      volumeMounts:
        - name: app-logs
          mountPath: /log
      env:
        - name: PATHS
          value: "/log/*"
        - name: ES_SERVER
          value: 172.23.5.255:9200
        - name: INDEX
          value: logstash-docker
        - name: INPUT_TYPE
          value: log

```

目前使用这种方式会有个问题，及时 `PATHS` 只能传递单个目录，如果想传递多个目录需要修改`filebeat`镜像的 `docker-entrypoint.sh` 脚本，对该环境变量进行解析增加`filebeat.yml`文件中的`PATHS`列表。

推荐使用`ConfigMap`，这样`filebeat`的配置就能够更灵活。

注意事项

- 将`app`的 `/usr/local/TalkingData/logs` 目录挂载到`filebeat`的 `/log` 目录下。
- 该文件可以在 `manifests/test/filebeat-test.yaml` 找到。
- 我使用了自己的私有镜像仓库，测试时请换成自己的应用镜像。
- `Filebeat`的环境变量的值配置请参考<https://github.com/rootsongjc/docker-images>

创建应用

部署Deployment

```
kubectl create -f filebeat-test.yaml
```

4.3.2 应用日志收集

查看 `http://172.23.5.255:9200/_cat/indices` 将可以看到列表有这样的 indices：

```
green open filebeat-docker-test 7xPEwEbUQRirk8oDX36gA
A 5 1    2151      0    1.6mb 841.8kb
```

访问Kibana的web页面，查看 `filebeat-2017.05.17` 的索引，可以看到filebeat收集到了app日志。

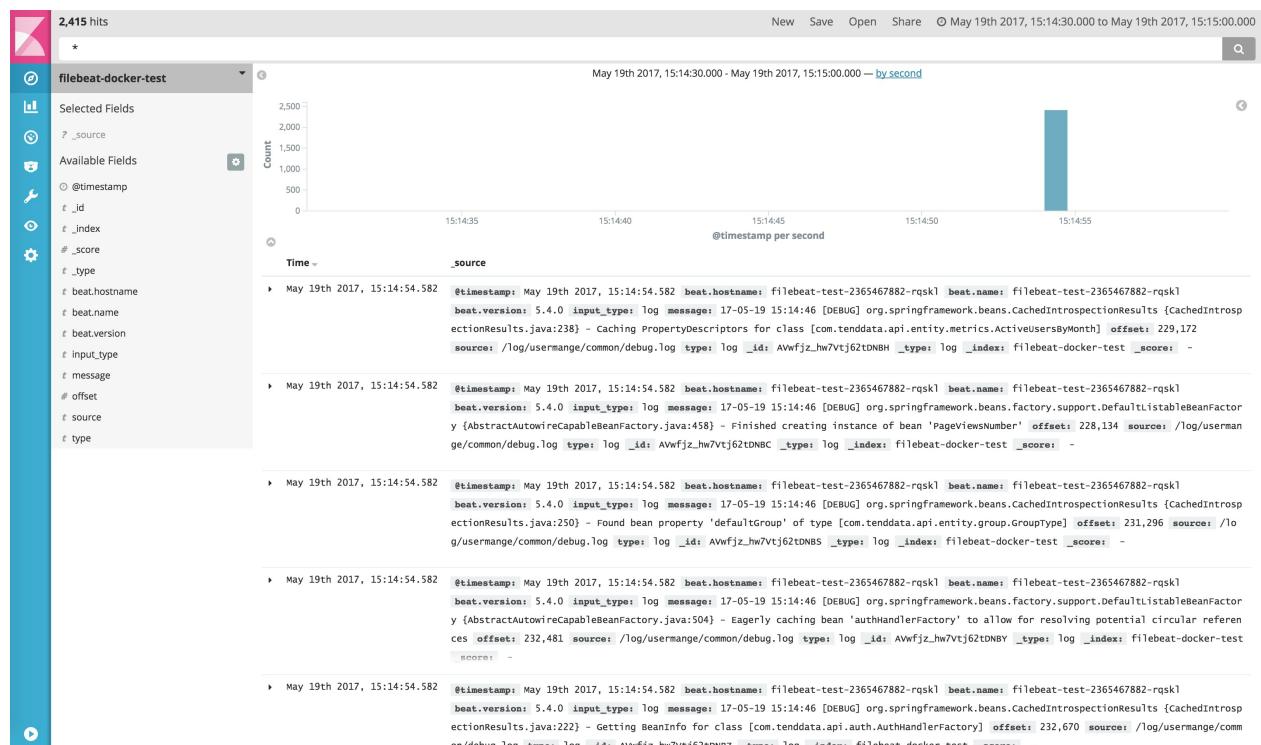


Figure: Kibana 页面

点开没个日志条目，可以看到以下详细字段：

4.3.2 应用日志收集



Doc: filebeat-docker-test/log/AVzs1QzKdPYTkmbgkats

Table JSON

○ @timestamp June 28th 2017, 11:53:19.425
t _id AVzs1QzKdPYTkmbgkats
t _index filebeat-docker-test
_score 1
t _type log
t beat.hostname filebeat-test-2365467882-4zwx8
t beat.name filebeat-test-2365467882-4zwx8
t beat.version 5.4.0
t input_type log
t message 2017-06-02 15:14:48.592 [net.sf.ehcache.CacheManager@614a616] INFO net.sf.ehcache.util.UpdateChecker-New update(s) found: 2.6.5 [http://www.terracotta.org/confluence/display/release/Release+Notes+ehcache+Core+2.6]. Please check http://ehcache.org for the latest version.
offset 272
t source /log/2017_06_02.stderrout.log
t type log

Figure: filebeat 收集的日志详细信息

- `_index` 值即我们在 YAML 文件的 `configMap` 中配置的 `index` 值
- `beat.hostname` 和 `beat.name` 即 pod 的名称
- `source` 表示 filebeat 容器中的日志目录

我们可以通过人为得使 `index = service name`，这样就可以方便的收集和查看每个 service 的日志。

for GitBook update 2017-08-07 13:54:27

配置最佳实践

本文档旨在汇总和强调用户指南、快速开始文档和示例中的最佳实践。该文档会很活跃并持续更新中。如果你觉得很有用的最佳实践但是本文档中没有包含，欢迎给我们提Pull Request。

通用配置建议

- 定义配置文件的时候，指定最新的稳定API版本（目前是V1）。
- 在配置文件push到集群之前应该保存在版本控制系统中。这样当需要的时候能够快速回滚，必要的时候也可以快速的创建集群。
- 使用YAML格式而不是JSON格式的配置文件。在大多数场景下它们都可以作为数据交换格式，但是YAML格式比起JSON更易读和配置。
- 尽量将相关的对象放在同一个配置文件里。这样比分成多个文件更容易管理。
参考[guestbook-all-in-one.yaml](#)文件中的配置（注意，尽管你可以在使用 `kubectl` 命令时指定配置文件目录，你也可以在配置文件目录下执行 `kubectl create` ——查看下面的详细信息）。
- 为了简化和最小化配置，也为了防止错误发生，不要指定不必要的默认配置。例如，省略掉 `ReplicationController` 的 `selector` 和 `label`，如果你希望它们跟 `podTemplate` 中的 `label` 一样的话，因为那些配置默认是 `podTemplate` 的 `label` 产生的。更多信息请查看 [guestbook app](#) 的yaml文件和 [examples](#)。
- 将资源对象的描述放在一个annotation中可以更好的内省。

裸的Pods vs Replication Controllers和Jobs

- 如果有其他方式替代“裸的” pod（如没有绑定到[replication controller](#)上的 pod），那么就使用其他选择。在node节点出现故障时，裸奔的pod不会被重新调度。Replication Controller总是会重新创建pod，除了明确指定了 `restartPolicy: Never` 的场景。Job 也许是比較合适的选择。

Services

- 通常最好在创建相关的[replication controllers](#)之前先创建[service](#)，你也可以在

创建Replication Controller的时候不指定replica数量（默认是1），创建service后，在通过Replication Controller来扩容。这样可以在扩容很多个replica之前先确认pod是正常的。

- 除非十分必要的条件下（如运行一个node daemon），不要使用hostPort（用来指定暴露在主机上的端口号）。当你给Pod绑定了一个hostPort，该pod可被调度到的主机的受限了，因为端口冲突。如果是为了调试目的来通过端口访问的话，你可以使用[kubectl proxy and apiserver proxy](#) 或者[kubectl port-forward](#)。你可使用Service来对外暴露服务。如果你确实需要将pod的端口暴露到主机上，考虑使用NodePort service。
- 跟hostPort一样的原因，避免使用hostNetwork。
- 如果你不需要kube-proxy的负载均衡的话，可以考虑使用使用[headless services](#)。

使用Label

- 定义labels来指定应用或Deployment的semantic attributes。例如，不是将label附加到一组pod来显式表示某些服务（例如，service:myservice），或者显式地表示管理pod的replication controller（例如，controller:mycontroller），附加label应该是标示语义属性的标签，例如{app:myapp,tier:frontend,phase:test,deployment:v3}。这将允许您选择适合上下文的对象组——例如，所有的“tier:frontend”pod的服务或app是“myapp”的所有“测试”阶段组件。有关此方法的示例，请参阅[guestbook](#)应用程序。

可以通过简单地从其service的选择器中省略特定于发行版本的标签，而不是更新服务的选择器来完全匹配replication controller的选择器，来实现跨越多个部署的服务，例如滚动更新。

- 为了滚动升级的方便，在Replication Controller的名字中包含版本信息，例如作为名字的后缀。设置一个version标签页是很有用的。滚动更新创建一个新的controller而不是修改现有的controller。因此，version含混不清的controller名字就可能带来问题。查看[Rolling Update Replication Controller](#)文档获取更多关于滚动升级命令的信息。

注意Deployment对象不再管理replication controller的版本名。Deployment中描述了对象的期望状态，如果对spec的更改被应用了话，Deployment controller会以控制的速率来更改实际状态到期望状态。

(Deployment 目前是 `extensions API Group` 的一部分)。

- 利用 `label` 做调试。因为 Kubernetes replication controller 和 service 使用 `label` 来匹配 `pods`，这允许你通过移除 pod 中的 `label` 的方式将其从一个 controller 或者 service 中移除，原来的 controller 会创建一个新的 pod 来取代移除的 pod。这是一个很有用的方式，帮你在一个隔离的环境中调试之前的“活着的” pod。查看 `kubectl label` 命令。

容器镜像

- 默认容器镜像拉取策略是 `IfNotPresent`，当本地已存在该镜像的时候 `Kubelet` 不会再从镜像仓库拉取。如果你希望总是从镜像仓库中拉取镜像的话，在 yaml 文件中指定镜像拉取策略为 `Always` (`imagePullPolicy: Always`) 或者指定镜像的 tag 为 `:latest`。

如果你没有将镜像标签指定为 `:latest`，例如指定为 `myimage:v1`，当该标签的镜像进行了更新，`kubelet` 也不会拉取该镜像。你可以在每次镜像更新后都生成一个新的 tag (例如 `myimage:v2`)，在配置文件中明确指定该版本。

注意：在生产环境下部署容器应该尽量避免使用 `:latest` 标签，因为这样很难追溯到底运行的是哪个版本的容器和回滚。

使用 `kubectl`

- 尽量使用 `kubectl create -f <directory>`。`kubectl` 会自动查找该目录下的所有后缀名为 `.yaml`、`.yml` 和 `.json` 文件并将它们传递给 `create` 命令。
- 使用 `kubectl delete` 而不是 `stop`。`Delete` 是 `stop` 的超集，`stop` 已经被弃用。
- 使用 `kubectl bulk` 操作 (通过文件或者 `label`) 来 `get` 和 `delete`。查看 `label selectors` 和 `using labels effectively`。
- 使用 `kubectl run` 和 `expose` 命令快速创建只有单个容器的 Deployment。查看 `quick start guide` 中的示例。

参考

Configuration Best Practices

for GitBook update 2017-08-07 13:54:27

集群及应用监控

在前面的[安装heapster插件](#)章节，我们已经谈到Kubernetes本身提供了监控插件作为集群和容器监控的选择，但是在实际使用中，因为种种原因，再考虑到跟我们自身的监控系统集成，我们准备重新造轮子。

针对kubernetes集群和应用的监控，相较于传统的虚拟机和物理机的监控有很多不同，因此对于传统监控需要有很多改造的地方，需要关注以下三个方面：

- Kubernetes集群本身的监控，主要是kubernetes的各个组件
- kubernetes集群中Pod的监控，Pod的CPU、内存、网络、磁盘等监控
- 集群内部应用的监控，针对应用本身的监控

Kubernetes集群中的监控

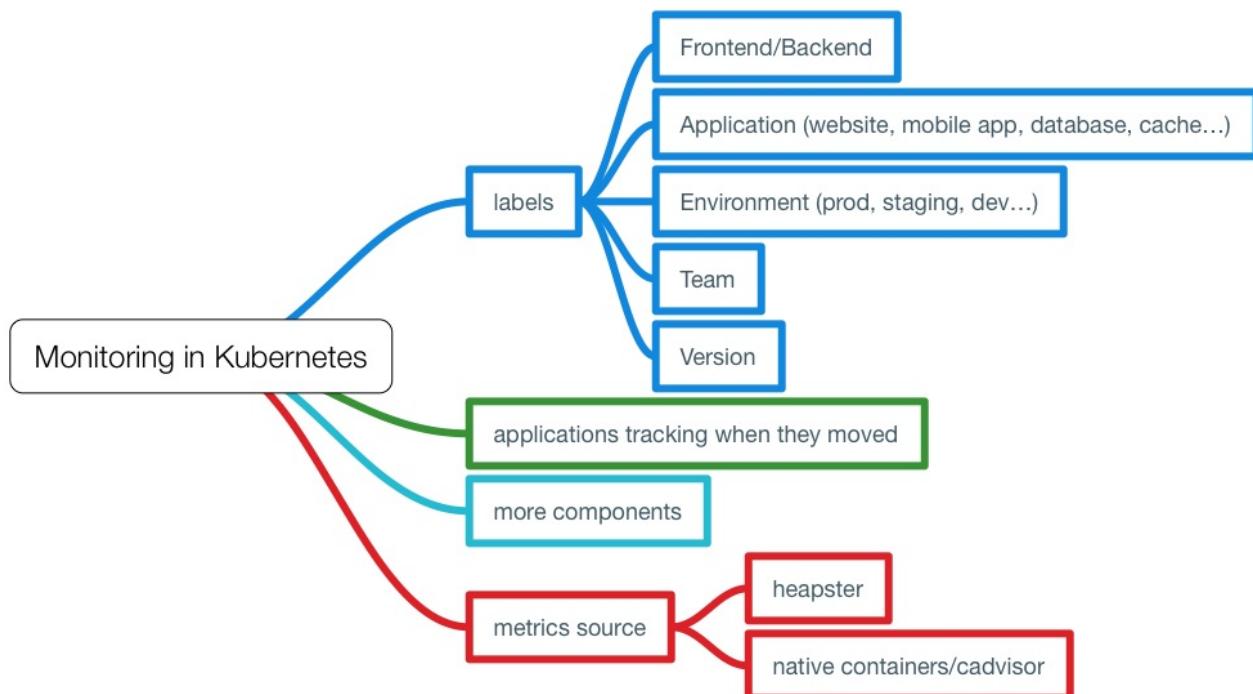


Figure: Kubernetes集群中的监控

跟物理机器和虚拟机的监控不同，在kubernetes集群中的监控复杂度更高一些，因为多了一个虚拟化层，当然这个跟直接监控docker容器又不一样，kubernetes在docker之上又抽象了一层service的概念。

在kubernetes中的监控需要考虑到这几个方面：

- 应该给Pod打上哪些label，这些label将成为监控的metrics。
- 当应用的Pod漂移了之后怎么办？因为要考虑到Pod的生命周期比虚拟机和物理机短的多，如何持续监控应用的状态？
- 更多的监控项，kubernetes本身、容器、应用等。
- 监控指标的来源，是通过heapster收集后汇聚还是直接从每台主机的docker上取？

容器的命名规则

首先我们需要清楚使用cAdvisor收集的数据的格式和字段信息。

当我们通过cAdvisor获取到了容器的信息后，例如访问 `${NODE_IP}:4194/api/v1.3/docker` 获取的json结果中的某个容器包含如下字段：

```

    "labels": {
        "annotation.io.kubernetes.container.hash": "f47f0602"
    },
    "annotation.io.kubernetes.container.ports": "[{\\"containerPort\\":80,\\"protocol\\":\\"TCP\\"}]",
    "annotation.io.kubernetes.container.restartCount": "0",
    "annotation.io.kubernetes.container.terminationMessagePath": "/dev/termination-log",
    "annotation.io.kubernetes.container.terminationMessagePolicy": "File",
    "annotation.io.kubernetes.pod.terminationGracePeriod": "30",
    "io.kubernetes.container.logpath": "/var/log/pods/d8a2e995-3617-11e7-a4b0-ecf4bbe5d414/php-redis_0.log",
    "io.kubernetes.container.name": "php-redis",
    "io.kubernetes.docker.type": "container",
    "io.kubernetes.pod.name": "frontend-2337258262-771lz"
},
"io.kubernetes.pod.namespace": "default",
"io.kubernetes.pod.uid": "d8a2e995-3617-11e7-a4b0-ecf4bbe5d414",
"io.kubernetes.sandbox.id": "843a0f018c0cef2a5451434713ea3f409f0debc2101d2264227e814ca0745677"
},

```

这些信息其实都是kubernetes创建容器时给docker container打的 Labels，使用 docker inspect \$conainer_name 命令同样可以看到上述信息。

你是否想过这些label跟容器的名字有什么关系？当你在node节点上执行 docker ps 看到的容器名字又对应哪个应用的Pod呢？

在kubernetes代码中pkg/kubelet/dockertools/docker.go中的BuildDockerName方法定义了容器的名称规范。

这段容器名称定义代码如下：

```
// Creates a name which can be reversed to identify both full pod name and container name.
```

```

// This function returns stable name, unique name and a unique i
d.
// Although rand.Uint32() is not really unique, but it's enough
for us because error will
// only occur when instances of the same container in the same p
od have the same UID. The
// chance is really slim.
func BuildDockerName(dockerName KubeletContainerName, container
*v1.Container) (string, string, string) {
    containerName := dockerName.ContainerName + "." + strconv.Fo
rmatUint(kubecontainer.HashContainerLegacy(container), 16)
    stableName := fmt.Sprintf("%s_%s_%s_%s",
        containerNamePrefix,
        containerName,
        dockerName.PodFullName,
        dockerName.PodUID)
    UID := fmt.Sprintf("%08x", rand.Uint32())
    return stableName, fmt.Sprintf("%s_%s", stableName, UID), UI
D
}

// Unpacks a container name, returning the pod full name and con
tainer name we would have used to
// construct the docker name. If we are unable to parse the name
, an error is returned.
func ParseDockerName(name string) (dockerName *KubeletContainerN
ame, hash uint64, err error) {
    // For some reason docker appears to be appending '/' to nam
es.
    // If it's there, strip it.
    name = strings.TrimPrefix(name, "/")
    parts := strings.Split(name, "_")
    if len(parts) == 0 || parts[0] != containerNamePrefix {
        err = fmt.Errorf("failed to parse Docker container name
%q into parts", name)
        return nil, 0, err
    }
    if len(parts) < 6 {
        // We have at least 5 fields. We may have more in the f
uture.

```

```

    // Anything with less fields than this is not something
we can
    // manage.
    glog.Warningf("found a container with the %q prefix, but
too few fields (%d): %q", containerNamePrefix, len(parts), name
)
    err = fmt.Errorf("Docker container name %q has less part
s than expected %v", name, parts)
    return nil, 0, err
}

nameParts := strings.Split(parts[1], ".")
containerName := nameParts[0]
if len(nameParts) > 1 {
    hash, err = strconv.ParseUint(nameParts[1], 16, 32)
    if err != nil {
        glog.Warningf("invalid container hash %q in containe
r %q", nameParts[1], name)
    }
}

podFullName := parts[2] + "_" + parts[3]
podUID := types.UID(parts[4])

return &KubeletContainerName{podFullName, podUID, containerN
ame}, hash, nil
}

```

我们可以看到容器名称中包含如下几个字段，中间用下划线隔开，至少有6个字
段，未来可能添加更多字段。

下面的是四个基本字段。

```
containerNamePrefix_containerName_PodFullName_PodUID
```

所有kubernetes启动的容器的containerNamePrefix都是k8s。

Kubernetes启动的docker容器的容器名称规范，下面以官方示例guestbook为例，
Deployment名为frontend中启动的名为php-redis的docker容器的副本数为3。

Deployment frontend的配置如下：

```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: bj-xg-oam-docker-hub-001.tendcloud.com/library/gb
-frontend:v4
          resources:
            requests:
              cpu: 100m
              memory: 100Mi
          env:
            - name: GET_HOSTS_FROM
              value: dns
          ports:
            - containerPort: 80

```

我们选取三个实例中的一个运行php-redis的docker容器。

```
k8s_php-redis_frontend-2337258262-154p7_default_d8a2e2dd-3617-11
e7-a4b0-ecf4bbe5d414_0
```

- containerNamePrefix : k8s
- containerName : php-redis
- podFullName : frontend-2337258262-154p7
- computeHash : 154p7
- deploymentName : frontend
- replicaSetName : frontend-2337258262

- namespace : default
- podUID : d8a2e2dd-3617-11e7-a4b0-ecf4bbe5d414

kubernetes容器命名规则解析，见下图所示。

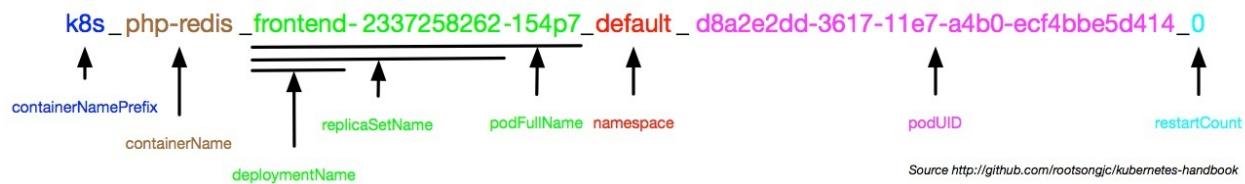


Figure: kubernetes的容器命名规则示意图

使用Heapster进行集群监控

Heapster是kubernetes官方提供的监控方案，我们在前面的章节中已经讲解了如何部署和使用heapster，见[安装Heapster插件](#)。

但是Grafana显示的指标只根据Namespace和Pod两层来分类，实在有些单薄，我们希望通过应用的label增加service这一层分类。架构图如下：

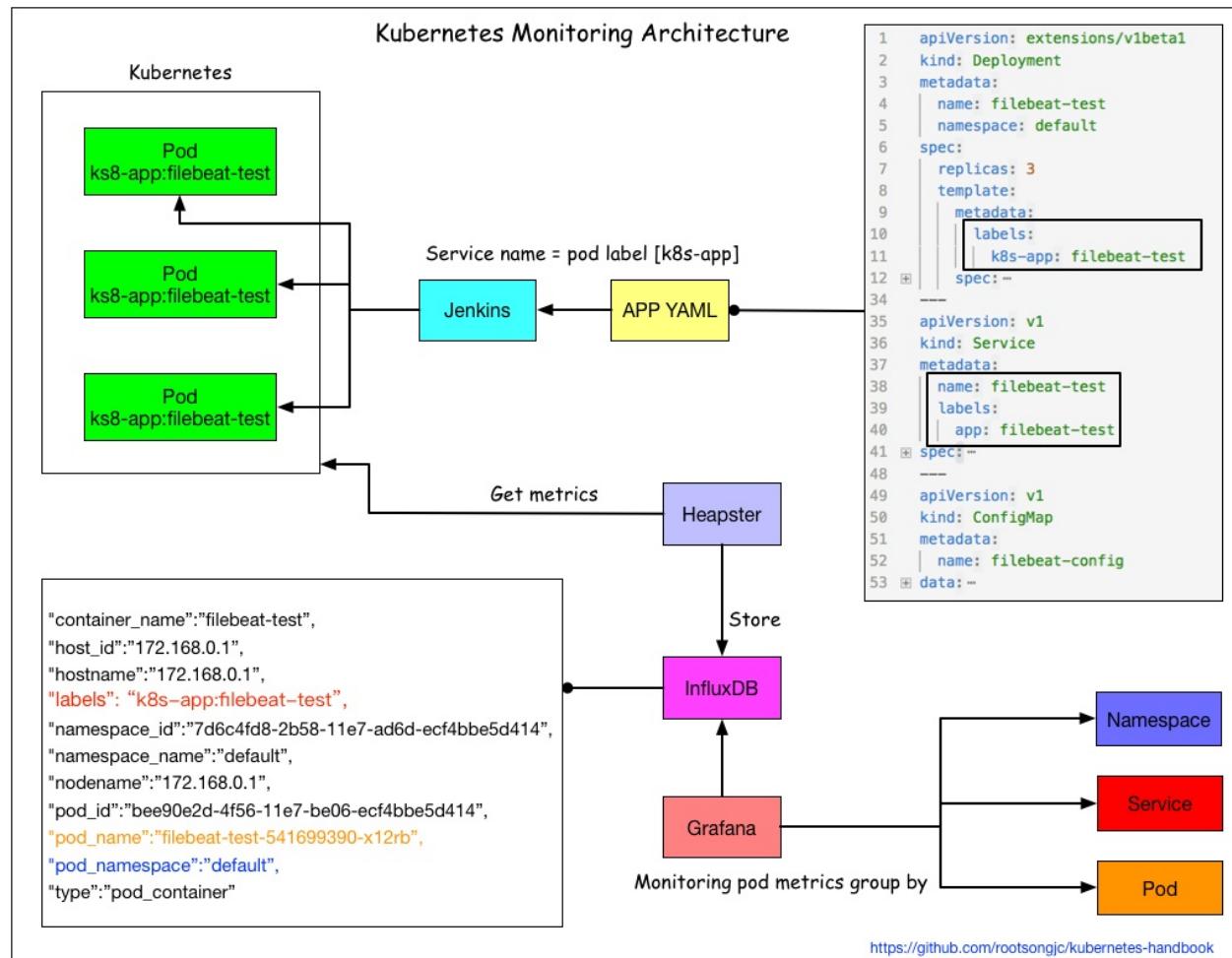


Figure: Heapster架构图（改进版）

在不改变原有架构的基础上，通过应用的label来区分不同应用的pod。

应用监控

Kubernetes中应用的监控架构如图：

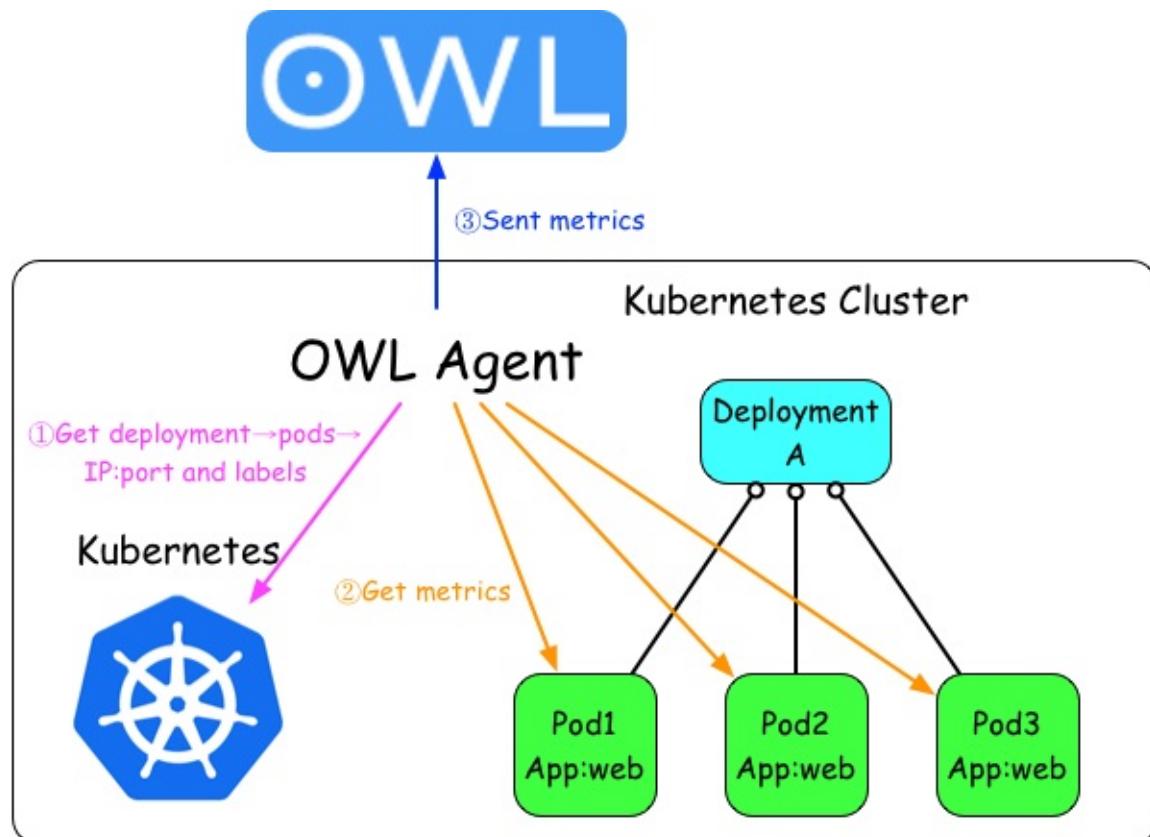


Figure: 应用监控架构图

这种方式有以下几个要点：

- 访问kubernetes API获取应用Pod的IP和端口
- Pod labels作为监控metric的tag
- 直接访问应用的Pod的IP和端口获取应用监控数据
- metrics发送到OWL中存储和展示

应用拓扑状态图

对于复杂的应用编排和依赖关系，我们希望能够有清晰的图标一览应用状态和拓扑关系，因此我们用到了Weaveworks开源的scope。

安装scope

我们在kubernetes集群上使用standalone方式安装，详情参考[Installing Weave Scope](#)。

使用[scope.yaml](#)文件安装scope，该服务安装在 kube-system namespace下。

```
$ kubectl apply -f scope.yaml
```

创建一个新的Ingress： kube-system.yaml ，配置如下：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ingress
  namespace: kube-system
spec:
  rules:
    - host: scope.weave.io
      http:
        paths:
          - path: /
            backend:
              serviceName: weave-scope-app
              servicePort: 80
```

执行 `kubectl apply -f kube-system.yaml` 后在你的主机上的 `/etc/hosts` 文件中添加一条记录：

```
172.20.0.119 scope.weave.io
```

在浏览器中访问 `scope.weave.io` 就可以访问到scope了，详见[边缘节点配置](#)。



Figure: 应用拓扑图

如上图所示，scope可以监控kubernetes集群中的一系列资源的状态、资源使用情况、应用拓扑、scale、还可以直接通过浏览器进入容器内部调试等。

参考

[Monitoring in the Kubernetes Era](#)

for GitBook update 2017-08-07 13:54:27

使用Jenkins进行持续集成与发布

我们基于Jenkins的CI/CD流程如下所示。

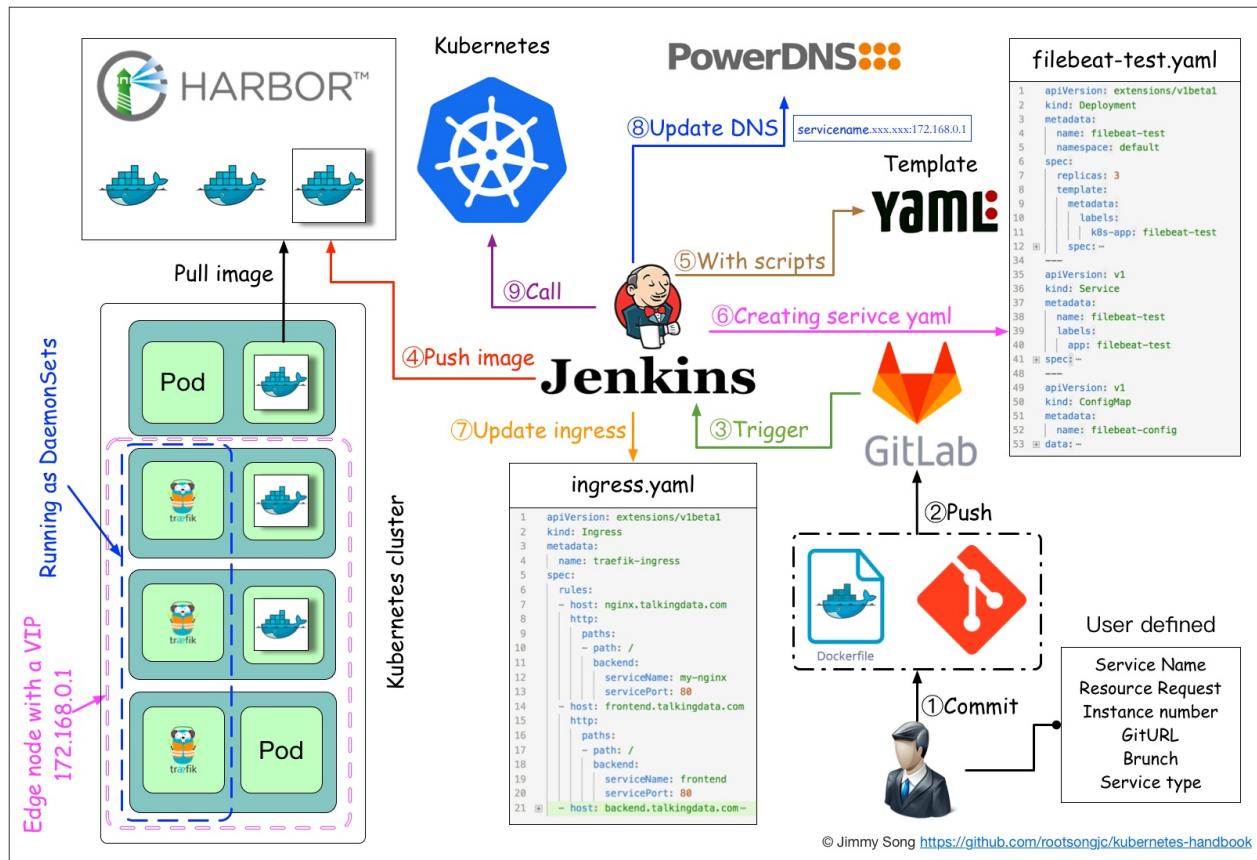


Figure: 基于Jenkins的持续集成与发布

流程说明

应用构建和发布流程说明。

1. 用户向Gitlab提交代码，代码中必须包含 Dockerfile
 2. 将代码提交到远程仓库
 3. 用户在发布应用时需要填写git仓库地址和分支、服务类型、服务名称、资源数量、实例个数，确定后触发Jenkins自动构建
 4. Jenkins的CI流水线自动编译代码并打包成docker镜像推送到Harbor镜像仓库
 5. Jenkins的CI流水线中包括了自定义脚本，根据我们已准备好的kubernetes的YAML模板，将其中的变量替换成用户输入的选项

6. 生成应用的kubernetes YAML配置文件
7. 更新Ingress的配置，根据新部署的应用的名称，在ingress的配置文件中增加一条路由信息
8. 更新PowerDNS，向其中插入一条DNS记录，IP地址是边缘节点的IP地址。关于边缘节点，请查看[边缘节点配置](#)
9. Jenkins调用kubernetes的API，部署应用

for GitBook update 2017-08-07 13:54:27

数据落盘问题的由来

这本质上是数据持久化问题，对于有些应用依赖持久化数据，比如应用自身产生的日志需要持久化存储的情况，需要保证容器里的数据不丢失，在Pod挂掉后，其他应用依然可以访问到这些数据，因此我们需要将数据持久化存储起来。

数据落盘问题解决方案

下面以一个应用的日志收集为例，该日志需要持久化收集到ElasticSearch集群中，如果不考虑数据丢失的情形，可以直接使用前面提到的[应用日志收集](#)一节中的方法，但考虑到Pod挂掉时logstash（或filebeat）并没有收集完该pod内日志的情形，我们想到了如下这种解决方案，示意图如下：

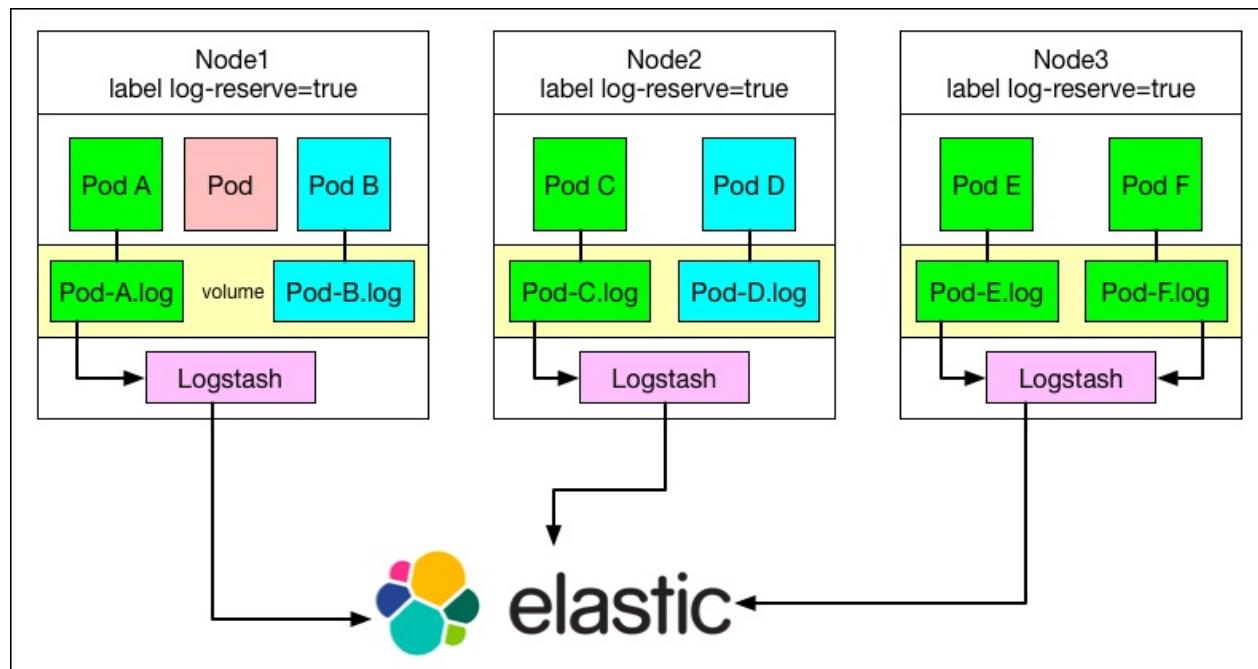


Figure: 日志持久化收集解决方案示意图

1. 首先需要给数据落盘的应用划分node，即这些应用只调用到若干台主机上
2. 给这若干台主机增加label
3. 使用 `deamonset` 方式在这若干台主机上启动logstash的Pod（使用 `nodeSelector` 来限定在这几台主机上，我们在边缘节点启动的 `traefik` 也是这种模式）

4. 将应用的数据通过volume挂载到宿主机上
5. Logstash（或者filebeat）收集宿主机上的数据，数据持久化不会丢失

Side-effect

1. 首先kubernetes本身就提供了数据持久化的解决方案statefulset，不过需要用到公有云的存储货其他分布式存储，这一点在我们的私有云环境里被否定了。
2. 需要管理主机的label，增加运维复杂度，但是具体问题具体对待
3. 必须保证应用启动顺序，需要先启动logstash
4. 为主机打label使用nodeSelector的方式限制了资源调度的范围

for GitBook update 2017-08-07 13:54:27

存储管理

for GitBook update 2017-08-07 13:54:27

GlusterFS

GlusterFS是Scale-Out存储解决方案Gluster的核心，它是一个开源的分布式文件系统，具有强大的横向扩展能力，通过扩展能够支持数PB存储容量和处理数千客户端。GlusterFS借助TCP/IP或InfiniBand RDMA网络将物理分布的存储资源聚集在一起，使用单一全局命名空间来管理数据。GlusterFS基于可堆叠的用户空间设计，可为各种不同的数据负载提供优异的性能。

for GitBook update 2017-08-07 13:54:27

使用glusterfs做持久化存储

我们复用kubernetes的三台主机做glusterfs存储。

以下步骤参考自：<https://www.xf80.com/2017/04/21/kubernetes-glusterfs/>

安装glusterfs

我们直接在物理机上使用yum安装，如果你选择在kubernetes上安装，请参考：<https://github.com/gluster/gluster-kubernetes/blob/master/docs/setup-guide.md>

```
# 先安装 gluster 源
$ yum install centos-release-gluster -y

# 安装 glusterfs 组件
$ yum install -y glusterfs glusterfs-server glusterfs-fuse glust
erfs-rdma glusterfs-geo-replication glusterfs-devel

## 创建 glusterfs 目录
$ mkdir /opt/glusterd

## 修改 glusterd 目录
$ sed -i 's/var\/lib/opt/g' /etc/glusterfs/glusterd.vol

# 启动 glusterfs
$ systemctl start glusterd.service

# 设置开机启动
$ systemctl enable glusterd.service

# 查看状态
$ systemctl status glusterd.service
```

配置glusterfs

4.4.1 GlusterFS

```
# 配置 hosts

$ vi /etc/hosts
172.20.0.113  sz-pg-oam-docker-test-001.tendcloud.com
172.20.0.114  sz-pg-oam-docker-test-002.tendcloud.com
172.20.0.115  sz-pg-oam-docker-test-003.tendcloud.com
```

```
# 开放端口
$ iptables -I INPUT -p tcp --dport 24007 -j ACCEPT

# 创建存储目录
$ mkdir /opt/gfs_data
```

```
# 添加节点到 集群
# 执行操作的本机不需要probe 本机
[root@sz-pg-oam-docker-test-001 ~]#
gluster peer probe sz-pg-oam-docker-test-002.tendcloud.com
gluster peer probe sz-pg-oam-docker-test-003.tendcloud.com

# 查看集群状态
$ gluster peer status
Number of Peers: 2

Hostname: sz-pg-oam-docker-test-002.tendcloud.com
Uuid: f25546cc-2011-457d-ba24-342554b51317
State: Peer in Cluster (Connected)

Hostname: sz-pg-oam-docker-test-003.tendcloud.com
Uuid: 42b6cad1-aa01-46d0-bbba-f7ec6821d66d
State: Peer in Cluster (Connected)
```

配置 volume

GlusterFS中的volume的模式有很多中，包括以下几种：

- 分布卷（默认模式）：即DHT, 也叫 分布卷: 将文件已hash算法随机分布到一

台服务器节点中存储。

- 复制模式：即AFR，创建volume时带 replica x 数量：将文件复制到 replica x 个节点中。
- 条带模式：即Striped，创建volume时带 stripe x 数量：将文件切割成数据块，分别存储到 stripe x 个节点中（类似raid 0）。
- 分布式条带模式：最少需要4台服务器才能创建。创建volume时 stripe 2 server = 4 个节点：是DHT与Striped的组合型。
- 分布式复制模式：最少需要4台服务器才能创建。创建volume时 replica 2 server = 4 个节点：是DHT与AFR的组合型。
- 条带复制卷模式：最少需要4台服务器才能创建。创建volume时 stripe 2 replica 2 server = 4 个节点：是 Striped 与 AFR 的组合型。
- 三种模式混合：至少需要8台服务器才能创建。stripe 2 replica 2，每4个节点组成一个组。

这几种模式的示例图参考：[CentOS7安装GlusterFS](#)。

因为我们只有三台主机，在此我们使用默认的分布卷模式。请勿在生产环境上使用该模式，容易导致数据丢失。

```
# 创建分布卷
$ gluster volume create k8s-volume transport tcp sz-pg-oam-docker-test-001.tendcloud.com:/opt/gfs_data sz-pg-oam-docker-test-002.tendcloud.com:/opt/gfs_data sz-pg-oam-docker-test-003.tendcloud.com:/opt/gfs_data force

# 查看volume状态
$ gluster volume info
Volume Name: k8s-volume
Type: Distribute
Volume ID: 9a3b0710-4565-4eb7-abae-1d5c8ed625ac
Status: Created
Snapshot Count: 0
Number of Bricks: 3
Transport-type: tcp
Bricks:
Brick1: sz-pg-oam-docker-test-001.tendcloud.com:/opt/gfs_data
Brick2: sz-pg-oam-docker-test-002.tendcloud.com:/opt/gfs_data
Brick3: sz-pg-oam-docker-test-003.tendcloud.com:/opt/gfs_data
Options Reconfigured:
transport.address-family: inet
nfs.disable: on

# 启动 分布卷
$ gluster volume start k8s-volume
```

Glusterfs调优

```
# 开启 指定 volume 的配额
$ gluster volume quota k8s-volume enable

# 限制 指定 volume 的配额
$ gluster volume quota k8s-volume limit-usage / 1TB

# 设置 cache 大小, 默认32MB
$ gluster volume set k8s-volume performance.cache-size 4GB

# 设置 io 线程, 太大会导致进程崩溃
$ gluster volume set k8s-volume performance.io-thread-count 16

# 设置 网络检测时间, 默认42s
$ gluster volume set k8s-volume network.ping-timeout 10

# 设置 写缓冲区的大小, 默认1M
$ gluster volume set k8s-volume performance.write-behind-window-size 1024MB
```

Kubernetes中配置glusterfs

官方的文档

见：<https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/glusterfs>

以下用到的所有yaml和json配置文件可以在[glusterfs](#)中找到。注意替换其中私有镜像地址为自己的镜像地址。

kubernetes安装客户端

```
# 在所有 k8s node 中安装 glusterfs 客户端

$ yum install -y glusterfs glusterfs-fuse

# 配置 hosts

$ vi /etc/hosts

172.20.0.113    sz-pg-oam-docker-test-001.tendcloud.com
172.20.0.114    sz-pg-oam-docker-test-002.tendcloud.com
172.20.0.115    sz-pg-oam-docker-test-003.tendcloud.com
```

因为我们glusterfs跟kubernetes集群复用主机，因此这一步可以省去。

配置 endpoints

4.4.1 GlusterFS

```
$ curl -o https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/volumes/glusterfs/glusterfs-endpoints.json

# 修改 endpoints.json , 配置 glusters 集群节点ip
# 每一个 addresses 为一个 ip 组

{
  "addresses": [
    {
      "ip": "172.22.0.113"
    }
  ],
  "ports": [
    {
      "port": 1990
    }
  ]
},

# 导入 glusterfs-endpoints.json

$ kubectl apply -f glusterfs-endpoints.json

# 查看 endpoints 信息
$ kubectl get ep
```

配置 **service**

```
$ curl -o https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/volumes/glusterfs/glusterfs-service.json

# service.json 里面查找的是 endpoints 的名称与端口，端口默认配置为 1，我改成了1990

# 导入 glusterfs-service.json
$ kubectl apply -f glusterfs-service.json

# 查看 service 信息
$ kubectl get svc
```

创建测试 pod

```
$ curl -o https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/volumes/glusterfs/glusterfs-pod.json

# 编辑 glusterfs-pod.json
# 修改 volumes 下的 path 为上面创建的 volume 名称

"path": "k8s-volume"

# 导入 glusterfs-pod.json
$ kubectl apply -f glusterfs-pod.json

# 查看 pods 状态
$ kubectl get pods
NAME                      READY   STATUS    RESTARTS
AGE
glusterfs                1/1     Running   0
1m

# 查看 pods 所在 node
$ kubectl describe pods/glusterfs

# 登陆 node 物理机，使用 df 可查看挂载目录
$ df -h
172.20.0.113:k8s-volume 1073741824          0 1073741824  0% 172.
20.0.113:k8s-volume  1.0T      0  1.0T    0% /var/lib/kubelet/pods
/3de9fc69-30b7-11e7-bfbd-8af1e3a7c5bd/volumes/kubernetes.io~glus
terfs/glusterfsvol
```

配置 PersistentVolume

PersistentVolume (PV) 和 PersistentVolumeClaim (PVC) 是 Kubernetes 提供的两种 API 资源，用于抽象存储细节。管理员关注于如何通过 PV 提供存储功能而无需关注用户如何使用，同样的用户只需要挂载 PVC 到容器中而不需要关注存储卷采用何种技术实现。

PVC 和 PV 的关系跟 pod 和 node 关系类似，前者消耗后者的资源。PVC 可以向 PV 申请指定大小的存储资源并设置访问模式。

PV属性

- storage容量
- 读写属性：分别为ReadWriteOnce：单个节点读写； ReadOnlyMany：多节点只读； ReadWriteMany：多节点读写

```
$ cat glusterfs-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: gluster-dev-volume
spec:
  capacity:
    storage: 8Gi
  accessModes:
    - ReadWriteMany
  glusterfs:
    endpoints: "glusterfs-cluster"
    path: "k8s-volume"
    readOnly: false

# 导入PV
$ kubectl apply -f glusterfs-pv.yaml

# 查看 pv
$ kubectl get pv
NAME          CAPACITY   ACCESSMODES   RECLAIMPOLICY   ST
ATUS          CLAIM      STORAGECLASS  REASON        AGE
gluster-dev-volume   8Gi        RWX           Retain       Av
ailable                  3s
```

PVC属性

- 访问属性与PV相同
- 容量：向PV申请的容量 \leq PV总容量

配置PVC

4.4.1 GlusterFS

```
$ cat glusterfs-pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: glusterfs-nginx
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 8Gi

# 导入 pvc
$ kubectl apply -f glusterfs-pvc.yaml

# 查看 pvc

$ kubectl get pv
NAME           STATUS    VOLUME          CAPACITY   ACCE
SSMODES        STORAGECLASS AGE
glusterfs-nginx Bound    gluster-dev-volume 8Gi       RWX
                           4s
```

创建 nginx deployment 挂载 volume

```
$ vi nginx-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-dm
spec:
  replicas: 2
  template:
    metadata:
      labels:
        name: nginx
  spec:
    containers:
```

4.4.1 GlusterFS

```
- name: nginx
  image: nginx:alpine
  imagePullPolicy: IfNotPresent
  ports:
    - containerPort: 80
  volumeMounts:
    - name: gluster-dev-volume
      mountPath: "/usr/share/nginx/html"
  volumes:
    - name: gluster-dev-volume
      persistentVolumeClaim:
        claimName: glusterfs-nginx

# 导入 deployment
$ kubectl apply -f nginx-deployment.yaml

# 查看 deployment
$ kubectl get pods |grep nginx-dm
nginx-dm-3698525684-g0mvt      1/1      Running   0          6
s
nginx-dm-3698525684-hbzq1      1/1      Running   0          6
s

# 查看 挂载
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- df -h|grep k8s-volume
172.20.0.113:k8s-volume       1.0T      0  1.0T  0% /usr/share
/nginx/html

# 创建文件 测试
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- touch /usr/share
/nginx/html/index.html

$ kubectl exec -it nginx-dm-3698525684-g0mvt -- ls -lt /usr/shar
e/nginx/html/index.html
-rw-r--r-- 1 root root 0 May  4 11:36 /usr/share/nginx/html/inde
x.html

# 验证 glusterfs
# 因为我们使用分布卷，所以可以看到某个节点中有文件
```

4.4.1 GlusterFS

```
[root@sz-pg-oam-docker-test-001 ~] ls /opt/gfs_data/  
[root@sz-pg-oam-docker-test-002 ~] ls /opt/gfs_data/  
index.html  
[root@sz-pg-oam-docker-test-003 ~] ls /opt/gfs_data/
```

参考

[在kubernetes中安装glusterfs](#)

[CentOS 7 安装 GlusterFS](#)

[GlusterFS with kubernetes](#)

for GitBook update 2017-08-07 13:54:27

Storage for Containers using Gluster – Part II

概述

本文由Daniel Messer（Technical Marketing Manager Storage @RedHat）和Keith Tenzer（Solutions Architect @RedHat）共同撰写。

- [Storage for Containers Overview – Part I](#)
- [Storage for Containers using Gluster – Part II](#)
- [Storage for Containers using Container Native Storage – Part III](#)
- [Storage for Containers using Ceph – Part IV](#)
- [Storage for Containers using NetApp ONTAP NAS – Part V](#)
- [Storage for Containers using NetApp SolidFire – Part VI](#)

Gluster作为Container-Ready Storage(CRS)

在本文中，我们将介绍容器存储的首选以及如何部署它。Kubernetes和OpenShift支持GlusterFS已经有一段时间了。GlusterFS的适用性很好，可用于所有的部署场景：裸机、虚拟机、内部部署和公共云。在容器中运行GlusterFS的新特性将在本系列后面讨论。

GlusterFS是一个分布式文件系统，内置了原生协议（GlusterFS）和各种其他协议（NFS，SMB，...）。为了与OpenShift集成，节点将通过FUSE使用原生协议，将GlusterFS卷挂载到节点本身上，然后将它们绑定到目标容器中。OpenShift / Kubernetes具有实现请求、释放和挂载、卸载GlusterFS卷的原生程序。

CRS概述

在存储方面，根据OpenShift / Kubernetes的要求，还有一个额外的组件管理集群，称为“heketi”。这实际上是一个用于GlusterFS的REST API，它还提供CLI版本。在以下步骤中，我们将在3个GlusterFS节点中部署heketi，使用它来部署GlusterFS

存储池，将其连接到OpenShift，并使用它来通过PersistentVolumeClaims为容器配置存储。我们将总共部署4台虚拟机。一个用于OpenShift（实验室设置），另一个用于GlusterFS。

注意：您的系统应至少需要有四核CPU，16GB RAM和20 GB可用磁盘空间。

部署OpenShift

首先你需要先部署OpenShift。最有效率的方式是直接在虚拟机中部署一个All-in-One环境，部署指南见 [the “OpenShift Enterprise 3.4 all-in-one Lab Environment” article.](#)。

确保你的OpenShift虚拟机可以解析外部域名。编辑 `/etc/dnsmasq.conf` 文件，增加下面的Google DNS：

```
server=8.8.8.8
```

重启：

```
# systemctl restart dnsmasq
# ping -c1 google.com
```

部署Gluster

GlusterFS至少需要有以下配置的3台虚拟机：

- RHEL 7.3
- 2 CPUs
- 2 GB内存
- 30 GB磁盘存储给操作系统
- 10 GB磁盘存储给GlusterFS bricks

修改`/etc/hosts`文件，定义三台虚拟机的主机名。

例如（主机名可以根据你自己的环境自由调整）

4.4.1 GlusterFS

```
# cat /etc/hosts
127.0.0.1      localhost localhost.localdomain localhost4 localhost4.localdomain4
::1            localhost localhost.localdomain localhost6 localhost6.localdomain6
172.16.99.144  ocp-master.lab ocp-master
172.16.128.7   crs-node1.lab crs-node1
172.16.128.8   crs-node2.lab crs-node2
172.16.128.9   crs-node3.lab crs-node3
```

在**3台GlusterFS**虚拟机上都执行以下步骤：

```
# subscription-manager repos --disable="*"
# subscription-manager repos --enable=rhel-7-server-rpms
```

如果你已经订阅了**GlusterFS**那么可以直接使用，开启 `rh-gluster-3-for-rhel-7-server-rpms` 的yum源。

如果你没有的话，那么可以通过**EPEL**使用非官方支持的**GlusterFS**的社区源。

```
# yum -y install http://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
# rpm --import http://dl.fedoraproject.org/pub/epel/RPM-GPG-KEY-EPEL-7
```

在 `/etc/yum.repos.d/` 目录下创建 `glusterfs-3.10.repo` 文件：

```
[glusterfs-3.10]
name=glusterfs-3.10
description="GlusterFS 3.10 Community Version"
baseurl=https://buildlogs.centos.org/centos/7/storage/x86_64/gluster-3.10/
gpgcheck=0
enabled=1
```

验证源已经被激活。

4.4.1 GlusterFS

```
# yum repolist
```

现在可以开始安装GlusterFS了。

```
# yum -y install glusterfs-server
```

需要为GlusterFS peers打开几个基本TCP端口，以便与OpenShift进行通信并提供存储：

```
# firewall-cmd --add-port=24007-24008/tcp --add-port=49152-49664/tcp --add-port=2222/tcp  
# firewall-cmd --runtime-to-permanent
```

现在我们可以启动GlusterFS的daemon进程了：

```
# systemctl enable glusterd  
# systemctl start glusterd
```

完成。GlusterFS已经启动并正在运行。其他配置将通过heketi完成。

在**GlusterFS**的一台虚拟机上安装**heketi**

```
[root@crs-node1 ~]# yum -y install heketi heketi-client
```

更新**EPEL**

如果你没有Red Hat Gluster Storage订阅的话，你可以从EPEL中获取heketi。在撰写本文时，2016年10月那时候还是3.0.0-1.el7版本，它不适用于OpenShift 3.4。你将需要更新到更新的版本：

```
[root@crs-node1 ~]# yum -y install wget
[root@crs-node1 ~]# wget https://github.com/heketi/heketi/releases/download/v4.0.0/heketi-v4.0.0.linux.amd64.tar.gz
[root@crs-node1 ~]# tar -xzf heketi-v4.0.0.linux.amd64.tar.gz
[root@crs-node1 ~]# systemctl stop heketi
[root@crs-node1 ~]# cp heketi/heketi* /usr/bin/
[root@crs-node1 ~]# chown heketi:heketi /usr/bin/heketi*
```

在 `/etc/systemd/system/heketi.service` 中创建v4版本的heketi二进制文件的更新语法文件：

```
[Unit]
Description=Heketi Server

[Service]
Type=simple
WorkingDirectory=/var/lib/heketi
EnvironmentFile=-/etc/heketi/heketi.json
User=heketi
ExecStart=/usr/bin/heketi --config=/etc/heketi/heketi.json
Restart=on-failure
StandardOutput=syslog
StandardError=syslog

[Install]
WantedBy=multi-user.target
```

```
[root@crs-node1 ~]# systemctl daemon-reload
[root@crs-node1 ~]# systemctl start heketi
```

Heketi使用SSH来配置GlusterFS的所有节点。创建SSH密钥对，将公钥拷贝到所有3个节点上（包括你登陆的第一个节点）：

```
[root@crs-node1 ~]# ssh-keygen -f /etc/heketi/heketi_key -t rsa
-N ''
[root@crs-node1 ~]# ssh-copy-id -i /etc/heketi/heketi_key.pub root@crs-node1.lab
[root@crs-node1 ~]# ssh-copy-id -i /etc/heketi/heketi_key.pub root@crs-node2.lab
[root@crs-node1 ~]# ssh-copy-id -i /etc/heketi/heketi_key.pub root@crs-node3.lab
[root@crs-node1 ~]# chown heketi:heketi /etc/heketi/heketi_key*
```

剩下唯一要做的事情就是配置heketi来使用SSH。编辑 `/etc/heketi/heketi.json` 文件使它看起来像下面这个样子（改变的部分突出显示下划线）：

```
{
    "_port_comment": "Heketi Server Port Number",
    "port": "8080",
    "_use_auth": "Enable JWT authorization. Please enable for deployment",
    "use_auth": false,
    "_jwt": "Private keys for access",
    "jwt": {
        "_admin": "Admin has access to all APIs",
        "admin": {
            "key": "My Secret"
        },
        "_user": "User only has access to /volumes endpoint",
        "user": {
            "key": "My Secret"
        }
    },
    "_glusterfs_comment": "GlusterFS Configuration",
    "glusterfs": {
        "_executor_comment": [
            "Execute plugin. Possible choices: mock, ssh",
            "mock: This setting is used for testing and development .",
            " It will not send commands to any node."
        ]
    }
}
```

```

        "ssh: This setting will notify Heketi to ssh to the nodes.",
        " It will need the values in sshexec to be configured."
    ,
        "kubernetes: Communicate with GlusterFS containers over",
    ,
        " Kubernetes exec api."
],
"executor":"ssh",
"_sshexec_comment":"SSH username and private key file info
rmation",
"sshexec":{
    "keyfile":"/etc/heketi/heketi_key",
    "user":"root",
    "port":"22",
    "fstab":"/etc/fstab"
},
"_kubeexec_comment":"Kubernetes configuration",
"kubeexec":{
    "host":"https://kubernetes.host:8443",
    "cert":"/path/to/crt.file",
    "insecure":false,
    "user":"kubernetes username",
    "password":"password for kubernetes user",
    "namespace":"OpenShift project or Kubernetes namespace"
},
    "fstab":"Optional: Specify fstab file on node. Default
is /etc/fstab"
},
"_db_comment":"Database file name",
"db":"/var/lib/heketi/heketi.db",
"_loglevel_comment": [
    "Set log level. Choices are:",
    " none, critical, error, warning, info, debug",
    "Default is warning"
],
"loglevel":"debug"
}
}

```

4.4.1 GlusterFS

完成。heketi将监听8080端口，我们来确认下防火墙规则允许它监听该端口：

```
# firewall-cmd --add-port=8080/tcp  
# firewall-cmd --runtime-to-permanent
```

重启heketi：

```
# systemctl enable heketi  
# systemctl restart heketi
```

测试它是否在运行：

```
# curl http://crs-node1.lab:8080/hello  
Hello from Heketi
```

很好。heketi上场的时候到了。我们将使用它来配置我们的GlusterFS存储池。该软件已经在我们所有的虚拟机上运行，但并未被配置。要将其改造为满足我们需求的存储系统，需要在拓扑文件中描述我们所需的GlusterFS存储池，如下所示：

```
# vi topology.json  
{  
  "clusters": [  
    {  
      "nodes": [  
        {  
          "node": {  
            "hostnames": {  
              "manage": [  
                "crs-node1.lab"  
              ],  
              "storage": [  
                "172.16.128.7"  
              ]  
            },  
            "zone": 1  
          },  
          "devices": [  
            {  
              "path": "/dev/nvme0n1",  
              "format": "ext4",  
              "size": 100,  
              "role": "data"  
            }  
          ]  
        }  
      ]  
    }  
  ]  
}
```

```
        "/dev/sdb"
    ],
},
{
  "node": {
    "hostnames": {
      "manage": [
        "crs-node2.lab"
      ],
      "storage": [
        "172.16.128.8"
      ]
    },
    "zone": 1
  },
  "devices": [
    "/dev/sdb"
  ]
},
{
  "node": {
    "hostnames": {
      "manage": [
        "crs-node3.lab"
      ],
      "storage": [
        "172.16.128.9"
      ]
    },
    "zone": 1
  },
  "devices": [
    "/dev/sdb"
  ]
}
]
```

该文件格式比较简单，基本上是告诉heketi要创建一个3节点的集群，其中每个节点包含的配置有FQDN，IP地址以及至少一个将用作GlusterFS块的备用块设备。

现在将该文件发送给heketi：

```
# export HEKETI_CLI_SERVER=http://crs-node1.lab:8080
# heketi-cli topology load --json=topology.json
Creating cluster ... ID: 78cdb57aa362f5284bc95b2549bc7e7d
  Creating node crs-node1.lab ... ID: ffd7671c0083d88aeda9fd1cb40
b339b
    Adding device /dev/sdb ... OK
  Creating node crs-node2.lab ... ID: 8220975c0a4479792e684584153
050a9
    Adding device /dev/sdb ... OK
  Creating node crs-node3.lab ... ID: b94f14c4dbd8850f6ac589ac3b3
9cc8e
    Adding device /dev/sdb ... OK
```

现在heketi已经配置了3个节点的GlusterFS存储池。很简单！你现在可以看到3个虚拟机都已经成功构成了GlusterFS中的可信存储池（Trusted Storage Pool）。

```
[root@crs-node1 ~]# gluster peer status
Number of Peers: 2

Hostname: crs-node2.lab
Uuid: 93b34946-9571-46a8-983c-c9f128557c0e
State: Peer in Cluster (Connected)
Other names:
crs-node2.lab

Hostname: 172.16.128.9
Uuid: e3c1f9b0-be97-42e5-beda-f70fc05f47ea
State: Peer in Cluster (Connected)
```

现在回到OpenShift！

将Gluster与OpenShift集成

为了集成OpenShift，需要两样东西：一个动态的Kubernetes Storage Provisioner和一个StorageClass。Provisioner在OpenShift中开箱即用。实际上关键的是如何将存储挂载到容器上。StorageClass是OpenShift中的用户可以用来实现的PersistentVolumeClaims的实体，它反过来能够触发一个Provisioner实现实际的配置，并将结果表示为Kubernetes PersistentVolume（PV）。

就像OpenShift中的其他组件一样，StorageClass也简单的用YAML文件定义：

```
# cat crs-storageclass.yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: container-ready-storage
  annotations:
    storageclass.beta.kubernetes.io/is-default-class: "true"
  provisioner: kubernetes.io/glusterfs
  parameters:
    resturl: "http://crs-node1.lab:8080"
    restauthenabled: "false"
```

我们的provisioner是kubernetes.io/glusterfs，将它指向我们的heketi实例。我们将类命名为“container-ready-storage”，同时使其成为所有没有显示指定StorageClass的PersistentVolumeClaim的默认StorageClass。

为你的GlusterFS池创建StorageClass：

```
# oc create -f crs-storageclass.yaml
```

在OpenShift中使用Gluster

我们来看下如何在OpenShift中使用GlusterFS。首先在OpenShift虚拟机中创建一个测试项目。

```
# oc new-project crs-storage --display-name="Container-Ready Storage"
```

这会向Kubernetes/OpenShift发出storage请求，请求一个 PersistentVolumeClaim（PVC）。这是一个简单的对象，它描述最少需要多少容量和应该提供哪种访问模式（非共享，共享，只读）。它通常是应用程序模板的一部分，但我们只需创建一个独立的PVC：

```
# cat crs-claim.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-crs-storage
  namespace: crs-storage
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

发送该请求：

```
# oc create -f crs-claim.yaml
```

观察在OpenShift中，PVC正在以动态创建volume的方式实现：

```
# oc get pvc
NAME          STATUS    VOLUME
CAPACITY     ACCESSMODES   AGE
my-crs-storage Bound    pvc-41ad5adb-107c-11e7-afae-000c2949c
ce7  1Gi        RWO       58s
```

太棒了！你现在可以在OpenShift中使用存储容量，而不需要直接与存储系统进行任何交互。我们来看看创建的volume：

```
# oc get pvc/pvc-41ad5adb-107c-11e7-afae-000c2949cce7
Name:          pvc-41ad5adb-107c-11e7-afae-000c2949cce7
Labels:
StorageClass:  container-ready-storage
Status:        Bound
Claim:         crs-storage/my-crs-storage
Reclaim Policy: Delete
Access Modes:   RWO
Capacity:      1Gi
Message:
Source:
  Type:       Glusterfs (a Glusterfs mount on the host that s
  hares a pod's lifetime)
  EndpointsName:  gluster-dynamic-my-crs-storage
  Path:        vol_85e444ee3bc154de084976a9aef16025
  ReadOnly:     false
```

What happened in the background was that when the PVC reached the system, our default StorageClass reached out to the GlusterFS Provisioner with the volume specs from the PVC. The provisioner in turn communicates with our heketi instance which facilitates the creation of the GlusterFS volume, which we can trace in it's log messages:

该volume是根据PVC中的定义特别创建的。在PVC中，我们没有明确指定要使用哪个StorageClass，因为heketi的GlusterFS StorageClass已经被定义为系统范围的默认值。

在后台发生的情况是，当PVC到达系统时，默认的StorageClass请求具有该PVC中volume声明规格的GlusterFS Provisioner。Provisioner又与我们的heketi实例通信，这有助于创建GlusterFS volume，我们可以在其日志消息中追踪：

```
[root@crs-node1 ~]# journalctl -l -u heketi.service
...
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [heketi] DEBUG 2017/03/24 11:25:52 /src/github.com/heketi/heketi/apps/glusterfs/volume_entry.go:298: Volume to be created on cluster e
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [heketi] INFO 2017/03/24 11:25:52 Creating brick 9e791b1daa12af783c9195941fe63103
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [heketi] INFO 2017/03/24 11:25:52 Creating brick 3e06af2f855bef521a95ada91680d14b
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [heketi] INFO 2017/03/24 11:25:52 Creating brick e4daa240f1359071e3f7ea22618cfbab
...
Mar 24 11:25:52 crs-node1.lab heketi[2598]: [sshexec] INFO 2017/03/24 11:25:52 Creating volume vol_85e444ee3bc154de084976a9aef16025 replica 3
...
Mar 24 11:25:53 crs-node1.lab heketi[2598]: Result: volume create: vol_85e444ee3bc154de084976a9aef16025: success: please start the volume to access data
...
Mar 24 11:25:55 crs-node1.lab heketi[2598]: Result: volume start : vol_85e444ee3bc154de084976a9aef16025: success
...
Mar 24 11:25:55 crs-node1.lab heketi[2598]: [asynchttp] INFO 2017/03/24 11:25:55 Completed job c3d6c4f9fc74796f4a5262647dc790fe in 3.176522702s
...
```

成功！ 大约用了3秒钟，GlusterFS池就配置完成了，并配置了一个volume。默认值是replica 3，这意味着数据将被复制到3个不同节点的3个块上（用GlusterFS作为后端存储）。该过程是通过Heketi在OpenShift进行编排的。

你也可以从GlusterFS的角度看到有关volume的信息：

```
[root@crs-node1 ~]# gluster volume list
vol_85e444ee3bc154de084976a9aef16025
[root@crs-node1 ~]# gluster volume info vol_85e444ee3bc154de0849
76a9aef16025

Volume Name: vol_85e444ee3bc154de084976a9aef16025
Type: Replicate
Volume ID: a32168c8-858e-472a-b145-08c20192082b
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: 172.16.128.8:/var/lib/heketi/mounts/vg_147b43f6f6903be8b
23209903b7172ae/brick_9e791b1daa12af783c9195941fe63103/brick
Brick2: 172.16.128.9:/var/lib/heketi/mounts/vg_72c0f520b0c57d807
be21e9c90312f85/brick_3e06af2f855bef521a95ada91680d14b/brick
Brick3: 172.16.128.7:/var/lib/heketi/mounts/vg_67314f879686de975
f9b8936ae43c5c5/brick_e4daa240f1359071e3f7ea22618cfbab/brick
Options Reconfigured:
transport.address-family: inet
nfs.disable: on
```

请注意，GlusterFS中的卷名称如何对应于OpenShift中Kubernetes Persistent Volume的“路径”。

或者，你也可以使用OpenShift UI来配置存储，这样可以很方便地在系统中的所有已知的StorageClasses中进行选择：

4.4.1 GlusterFS

The screenshot shows the 'Create Storage' form. At the top, it says 'Container-Ready Storage > Storage > Create Storage'. The form has several sections:

- * Storage Classes:** A dropdown menu showing 'container-ready-storage' selected.
- Type | Zone:** A dropdown menu showing 'No Storage Class' selected.
- * Name:** An input field containing 'my-crs-storage'.
- * Access Mode:** A dropdown menu showing 'Single User (RWO)' selected.
- * Size:** An input field showing '1' followed by a 'GiB' unit indicator.
- Use label selectors to request storage**: A link.
- Create** and **Cancel** buttons at the bottom.

Figure: Screen Shot 2017-03-23 at 21.50.34

The screenshot shows the 'Storage' list page. At the top, it says 'Container-Ready Storage'. The main area displays a table of Persistent Volume Claims:

Name	Status	Capacity	Access Modes	Age
my-crs-storage	Bound to volume pvc-eb686629-1079-11e7-a0a0-000c2949cc07	1 GiB	RWO (Read-Write-Once)	a few seconds

Figure: Screen Shot 2017-03-24 at 11.09.34.png

让我们做点更有趣的事情，在OpenShift中运行工作负载。

在仍运行着crs-storage项目的OpenShift虚拟机中执行：

```
# oc get templates -n openshift
```

你应该可以看到一个应用程序和数据库模板列表，这个列表将方便你更轻松的使用OpenShift来部署你的应用程序项目。

我们将使用MySQL来演示如何在OpenShift上部署具有持久化和弹性存储的有状态应用程序。Mysql-persistent模板包含一个用于MySQL数据库目录的1G空间的PVC。为了演示目的，可以直接使用默认值。

4.4.1 GlusterFS

```
# oc process mysql-persistent -n openshift | oc create -f -
```

等待部署完成。你可以通过UI或者命令行观察部署进度：

```
# oc get pods
NAME          READY   STATUS    RESTARTS   AGE
mysql-1-h4afb 1/1     Running   0          2m
```

好了。我们已经使用这个模板创建了一个service，secrets、PVC和pod。我们来使用它（你的pod名字将跟我的不同）：

```
# oc rsh mysql-1-h4afb
```

你已经成功的将它挂载到MySQL的pod上。我们连接一下数据库试试：

```
sh-4.2$ mysql -u $MYSQL_USER -p$MYSQL_PASSWORD -h $HOSTNAME $MYSQL_DATABASE
```

这点很方便，所有重要的配置，如MySQL凭据，数据库名称等都是pod模板中的环境变量的一部分，因此可以在pod中作为shell的环境变量。我们来创建一些数据：

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| sampledb       |
+-----+
2 rows in set (0.02 sec)

mysql> \u sampledb
Database changed
mysql> CREATE TABLE IF NOT EXISTS equipment (
->     equip_id int(5) NOT NULL AUTO_INCREMENT,
->     type varchar(50) DEFAULT NULL,
->     install_date DATE DEFAULT NULL,
```

4.4.1 GlusterFS

```
->      color varchar(20) DEFAULT NULL,  
->      working bool DEFAULT NULL,  
->      location varchar(250) DEFAULT NULL,  
->      PRIMARY KEY(equip_id)  
->      );  
Query OK, 0 rows affected (0.13 sec)  
  
mysql> INSERT INTO equipment (type, install_date, color, working  
, location)  
-> VALUES  
-> ("Slide", Now(), "blue", 1, "Southwest Corner");  
Query OK, 1 row affected, 1 warning (0.01 sec)  
  
mysql> SELECT * FROM equipment;  
+-----+-----+-----+-----+-----+-----+  
| equip_id | type   | install_date | color  | working | location  
|          |  
+-----+-----+-----+-----+-----+-----+  
|          |  
|       1 | Slide  | 2017-03-24 | blue   |       1 | Southwest  
Corner |  
+-----+-----+-----+-----+-----+-----+  
|          |  
1 row in set (0.00 sec)
```

很好，数据库运行正常。

你想看下数据存储在哪里吗？很简单！查看刚使用模板创建的mysql volume：

4.4.1 GlusterFS

```
# oc get pvc/mysql
NAME      STATUS      VOLUME                                     C
APACITY   ACCESSMODES   AGE
mysql     Bound       pvc-a678b583-1082-11e7-afae-000c2949cce7  1
Gi        RWO         11m

# oc describe pv/pvc-a678b583-1082-11e7-afae-000c2949cce7
Name:          pvc-a678b583-1082-11e7-afae-000c2949cce7
Labels:
StorageClass:  container-ready-storage
Status:        Bound
Claim:         crs-storage/mysql
Reclaim Policy: Delete
Access Modes:  RWO
Capacity:      1Gi
Message:
Source:
  Type:           Glusterfs (a Glusterfs mount on the host that s
  hares a pod's lifetime)
  EndpointsName:  gluster-dynamic-mysql
  Path:           vol_6299fc74eee513119dafd43f8a438db1
  ReadOnly:       false
```

GlusterFS的volume名字是vol_6299fc74eee513119dafd43f8a438db1。回到你的GlusterFS虚拟机中，输入：

```
# gluster volume info vol_6299fc74eee513119dafd43f8a438db

Volume Name: vol_6299fc74eee513119dafd43f8a438db1
Type: Replicate
Volume ID: 4115918f-28f7-4d4a-b3f5-4b9afe5b391f
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: 172.16.128.7:/var/lib/heketi/mounts/vg_67314f879686de975
f9b8936ae43c5c5/brick_f264a47aa32be5d595f83477572becf8/brick
Brick2: 172.16.128.8:/var/lib/heketi/mounts/vg_147b43f6f6903be8b
23209903b7172ae/brick_f5731fe7175cbe6e6567e013c2591343/brick
Brick3: 172.16.128.9:/var/lib/heketi/mounts/vg_72c0f520b0c57d807
be21e9c90312f85/brick_ac6add804a6a467cd81cd1404841bbf1/brick
Options Reconfigured:
transport.address-family: inet
nfs.disable: on
```

你可以看到数据是如何被复制到3个GlusterFS块的。我们从中挑一个（最好挑选你刚登陆的那台虚拟机并查看目录）：

```
# ll /var/lib/heketi/mounts/vg_67314f879686de975f9b8936ae43c5c5/
brick_f264a47aa32be5d595f83477572becf8/brick
total 180300
-rw-r----. 2 1000070000 2001      56 Mar 24 12:11 auto.cnf
-rw-----. 2 1000070000 2001     1676 Mar 24 12:11 ca-key.pem
-rw-r--r--. 2 1000070000 2001     1075 Mar 24 12:11 ca.pem
-rw-r--r--. 2 1000070000 2001     1079 Mar 24 12:12 client-cert.
pem
-rw-----. 2 1000070000 2001     1680 Mar 24 12:12 client-key.p
em
-rw-r----. 2 1000070000 2001     352 Mar 24 12:12 ib_buffer_po
ol
-rw-r----. 2 1000070000 2001 12582912 Mar 24 12:20 ibdata1
-rw-r----. 2 1000070000 2001 79691776 Mar 24 12:20 ib_logfile0
-rw-r----. 2 1000070000 2001 79691776 Mar 24 12:11 ib_logfile1
-rw-r----. 2 1000070000 2001 12582912 Mar 24 12:12 ibtmp1
drwxr-s---. 2 1000070000 2001     8192 Mar 24 12:12 mysql
-rw-r----. 2 1000070000 2001      2 Mar 24 12:12 mysql-1-h4af
b.pid
drwxr-s---. 2 1000070000 2001     8192 Mar 24 12:12 performance_
schema
-rw-----. 2 1000070000 2001     1676 Mar 24 12:12 private_key.
pem
-rw-r--r--. 2 1000070000 2001     452 Mar 24 12:12 public_key.p
em
drwxr-s---. 2 1000070000 2001      62 Mar 24 12:20 sampledb
-rw-r--r--. 2 1000070000 2001     1079 Mar 24 12:11 server-cert.
pem
-rw-----. 2 1000070000 2001     1676 Mar 24 12:11 server-key.p
em
drwxr-s---. 2 1000070000 2001     8192 Mar 24 12:12 sys
```

你可以在这里看到MySQL数据库目录。它使用GlusterFS作为后端存储，并作为绑定挂载给MySQL容器使用。如果你检查OpenShift VM上的mount表，你将会看到GlusterFS的mount。

总结

在这里我们是在OpenShift之外创建了一个简单但功能强大的GlusterFS存储池。该池可以独立于应用程序扩展和收缩。该池的整个生命周期由一个简单的称为heketi的前端管理，你只需要在部署增长时进行手动干预。对于日常配置操作，使用它的API与OpenShifts动态配置器交互，无需开发人员直接与基础架构团队进行交互。

这就是我们如何将存储带入DevOps世界 - 无痛苦，并在OpenShift PaaS系统的开发人员工具中直接提供。

GlusterFS和OpenShift可跨越所有环境：裸机，虚拟机，私有和公共云（Azure，Google Cloud，AWS ...），确保应用程序可移植性，并避免云供应商锁定。

祝你愉快在容器中使用GlusterFS！

(c) 2017 Keith Tenzer

原文链接：<https://keithtenzer.com/2017/03/24/storage-for-containers-using-gluster-part-ii/>

for GitBook update 2017-08-07 13:54:27

领域应用

for GitBook update 2017-08-07 13:54:27

微服务架构

- [Istio](#)

for GitBook update 2017-08-07 13:54:27

Istio 简介

前言

Istio是由Google、IBM和Lyft开源的微服务管理、保护和监控框架。Istio为希腊语，意思是“起航”。

简介

使用istio可以很简单的创建具有负载均衡、服务间认证、监控等功能的服务网络，而不需要对服务的代码进行任何修改。你只需要在部署环境中，例如Kubernetes的pod里注入一个特别的sidecar proxy来增加对istio的支持，用来截获微服务之间的网络流量。

目前版本的istio只支持kubernetes，未来计划支持其他其他环境。

另外，Istio的前身是IBM开源的Amalgam8，追本溯源，我们来看下它的特性。

Amalgam8

Amalgam8的网站上说，它是一个**Content-based Routing Fabric for Polyglot Microservices**，简单、强大且开源。

Amalgam8是一款基于内容和版本的路由布局，用于集成多语言异构体微服务。其control plane API可用于动态编程规则，用于在正在运行的应用程序中跨微服务进行路由和操作请求。

以内容/版本感知方式路由请求的能力简化了DevOps任务，如金丝雀和红/黑发布，A/B Test和系统地测试弹性微服务。

可以使用Amalgam8平台与受欢迎的容器运行时（如Docker，Kubernetes，Marathon / Mesos）或其他云计算提供商（如IBM Bluemix，Google Cloud Platform或Amazon AWS）。

特性

使用Istio的进行微服务管理有如下特性：

- 流量管理：控制服务间的流量和API调用流，使调用更可靠，增强不同环境下的网络鲁棒性。
- 可观测性：了解服务之间的依赖关系和它们之间的性质和流量，提供快速识别定位问题的能力。
- 策略实施：通过配置mesh而不是以改变代码的方式来控制服务之间的访问策略。
- 服务识别和安全：提供在mesh里的服务可识别性和安全性保护。

未来将支持多种平台，不论是kubernetes、Mesos、还是云。同时可以集成已有的ACL、日志、监控、配额、审计等。

架构

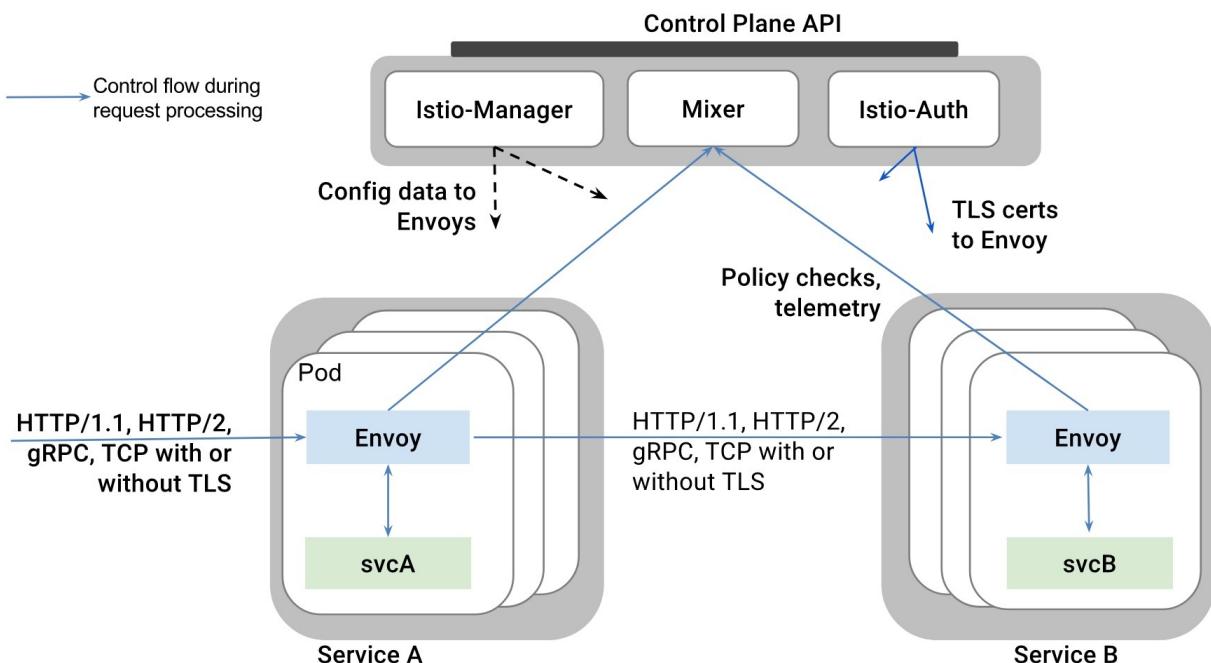


Figure: Istio架构图

Istio架构分为控制层和数据层。

- 数据层：由一组智能代理（Envoy）作为sidecar部署，协调和控制所有microservices之间的网络通信。
- 控制层：负责管理和配置代理路由流量，以及在运行时执行的政策。

Envoy

Istio使用Envoy代理的扩展版本，该代理是以C++开发的高性能代理，用于调解service mesh中所有服务的所有入站和出站流量。Istio利用了Envoy的许多内置功能，例如动态服务发现，负载平衡，TLS终止，HTTP/2&gRPC代理，断路器，运行状况检查，基于百分比的流量拆分分阶段上线，故障注入和丰富指标。

Envoy在kubernetes中作为pod的sidecar来部署。这允许Istio将大量关于流量行为的信号作为属性提取出来，这些属性又可以在Mixer中用于执行策略决策，并发送给监控系统以提供有关整个mesh的行为的信息。Sidecar代理模型还允许你将Istio功能添加到现有部署中，无需重新构建或重写代码。更多信息参见[设计目标](#)。

Mixer

Mixer负责在service mesh上执行访问控制和使用策略，并收集Envoy代理和其他服务的遥测数据。代理提取请求级属性，发送到mixer进行评估。有关此属性提取和策略评估的更多信息，请参见[Mixer配置](#)。混音器包括一个灵活的插件模型，使其能够与各种主机环境和基础架构后端进行接口，从这些细节中抽象出Envoy代理和Istio管理的服务。

Istio Manager

Istio-Manager用作用户和Istio之间的接口，收集和验证配置，并将其传播到各种Istio组件。它从Mixer和Envoy中抽取环境特定的实现细节，为他们提供独立于底层平台的用户服务的抽象表示。此外，流量管理规则（即通用4层规则和七层HTTP/gRPC路由规则）可以在运行时通过Istio-Manager进行编程。

Istio-auth

Istio-Auth提供强大的服务间和最终用户认证，使用相互TLS，内置身份和凭据管理。它可用于升级service mesh中的未加密流量，并为运营商提供基于服务身份而不是网络控制的策略的能力。Istio的未来版本将增加细粒度的访问控制和审计，以使用各种访问控制机制（包括属性和基于角色的访问控制以及授权hook）来控制和监控访问你服务、API或资源的人员。

参考

[Istio开源平台发布，Google、IBM和Lyft分别承担什么角色？](#)

[Istio：用于微服务的服务啮合层](#)

[Istio Overview](#)

for GitBook update 2017-08-07 13:54:27

安装istio

本文根据官网的文档整理而成，步骤包括安装 istio 0.1.5 并创建一个bookinfo 的微服务来测试istio的功能。

文中使用的yaml文件可以在[kubernetes-handbook](#)的 manifests/istio 目录中找到，所有的镜像都换成了我的私有镜像仓库地址，请根据官网的镜像自行修改。

安装环境

- CentOS 7.3.1611
- Docker 1.12.6
- Kubernetes 1.6.0

安装

1. 下载安装包

下载地址：<https://github.com/istio/istio/releases>

下载Linux版本的当前最新版安装包

```
 wget https://github.com/istio/istio/releases/download/0.1.5/isti  
o-0.1.5-linux.tar.gz
```

2. 解压

解压后，得到的目录结构如下：

```
 .  
 └── bin  
     └── istioctl  
 └── install  
     └── kubernetes  
         └── addons  
             └── grafana.yaml
```

```
|   |   └── prometheus.yaml
|   |   └── servicegraph.yaml
|   |   └── zipkin.yaml
|   └── istio-auth.yaml
|   └── istio-rbac-alpha.yaml
|   └── istio-rbac-beta.yaml
|   └── istio.yaml
|   └── README.md
|   └── templates
|       ├── istio-auth
|       |   ├── istio-auth-with-cluster-ca.yaml
|       |   ├── istio-cluster-ca.yaml
|       |   ├── istio-egress-auth.yaml
|       |   ├── istio-ingress-auth.yaml
|       |   └── istio-namespace-ca.yaml
|       ├── istio-egress.yaml
|       ├── istio-ingress.yaml
|       ├── istio-manager.yaml
|       └── istio-mixer.yaml
|
└── istio.VERSION
└── LICENSE
└── samples
    ├── apps
    |   ├── bookinfo
    |   |   ├── bookinfo.yaml
    |   |   ├── cleanup.sh
    |   |   ├── destination-ratings-test-delay.yaml
    |   |   ├── loadbalancing-policy-reviews.yaml
    |   |   ├── mixer-rule-additional-telemetry.yaml
    |   |   ├── mixer-rule-empty-rule.yaml
    |   |   ├── mixer-rule-ratings-denial.yaml
    |   |   ├── mixer-rule-ratings-ratelimit.yaml
    |   |   ├── README.md
    |   |   ├── route-rule-all-v1.yaml
    |   |   ├── route-rule-delay.yaml
    |   |   ├── route-rule-reviews-50-v3.yaml
    |   |   ├── route-rule-reviews-test-v2.yaml
    |   |   ├── route-rule-reviews-v2-v3.yaml
    |   |   └── route-rule-reviews-v3.yaml
    |
    └── httpbin
```

5.1.1 Istio

```
|   |   └── httpbin.yaml
|   |   └── README.md
|   └── sleep
|       ├── README.md
|       └── sleep.yaml
└── README.md
```

11 directories, 41 files

从文件列表中可以看到，安装包中包括了kubernetes的yaml文件，示例应用和安装模板。

3. 安装istioctl

将 `./bin/istioctl` 拷贝到你的 `$PATH` 目录下。

4. 检查RBAC

因为我们安装的kubernetes版本是1.6.0默认支持RBAC，这一步可以跳过。如果你使用的其他版本的kubernetes，请参考[官方文档](#)操作。

执行以下命令，正确的输出是这样的：

```
$ kubectl api-versions | grep rbac
rbac.authorization.k8s.io/v1alpha1
rbac.authorization.k8s.io/v1beta1
```

5. 创建角色绑定

```
$ kubectl create -f install/kubernetes/istio-rbac-beta.yaml
clusterrole "istio-manager" created
clusterrole "istio-ca" created
clusterrole "istio-sidecar" created
clusterrolebinding "istio-manager-admin-role-binding" created
clusterrolebinding "istio-ca-role-binding" created
clusterrolebinding "istio-ingress-admin-role-binding" created
clusterrolebinding "istio-sidecar-role-binding" created
```

5.1.1 Istio

注意：官网的安装包中的该文件中存在RoleBinding错误，应该是集群级别的 clusterrolebinding ，而release里的代码只是普通的 rolebinding ，查看该Issue [Istio manager cannot list of create k8s TPR when RBAC enabled #327](#)。

6. 安装istio核心组件

用到的镜像有：

```
docker.io/istio/mixer:0.1.5  
docker.io/istio/manager:0.1.5  
docker.io/istio/proxy_debug:0.1.5
```

我们暂时不开启 [Istio Auth](#)。

本文中用到的所有yaml文件中的 type: LoadBalancer 去掉，使用默认的 ClusterIP，然后配置Traefik ingress，就可以在集群外部访问。请参考[安装 Traefik ingress](#)。

```
kubectl apply -f install/kubernetes/istio.yaml
```

7. 安装监控插件

用到的镜像有：

```
docker.io/istio/grafana:0.1.5  
quay.io/coreos/prometheus:v1.1.1  
gcr.io/istio-testing/servicegraph:latest  
docker.io/openzipkin/zipkin:latest
```

为了方便下载，其中两个镜像我备份到了时速云：

```
index.tenxcloud.com/jimmy/prometheus:v1.1.1  
index.tenxcloud.com/jimmy/servicegraph:latest
```

安装插件

5.1.1 Istio

```
kubectl apply -f install/kubernetes addons/prometheus.yaml  
kubectl apply -f install/kubernetes addons/grafana.yaml  
kubectl apply -f install/kubernetes addons/servicegraph.yaml  
kubectl apply -f install/kubernetes addons/zipkin.yaml
```

在traefik ingress中增加增加以上几个服务的配置，同时增加istio-ingress配置。

```
- host: grafana.istio.io
  http:
    paths:
      - path: /
        backend:
          serviceName: grafana
          servicePort: 3000
- host: servicegraph.istio.io
  http:
    paths:
      - path: /
        backend:
          serviceName: servicegraph
          servicePort: 8088
- host: prometheus.istio.io
  http:
    paths:
      - path: /
        backend:
          serviceName: prometheus
          servicePort: 9090
- host: zipkin.istio.io
  http:
    paths:
      - path: /
        backend:
          serviceName: zipkin
          servicePort: 9411
- host: ingress.istio.io
  http:
    paths:
      - path: /
        backend:
          serviceName: istio-ingress
          servicePort: 80
```

测试

5.1.1 Istio

我们使用Istio提供的测试应用bookinfo微服务来进行测试。

该微服务用到的镜像有：

```
istio/examples-bookinfo-details-v1  
istio/examples-bookinfo-ratings-v1  
istio/examples-bookinfo-reviews-v1  
istio/examples-bookinfo-reviews-v2  
istio/examples-bookinfo-reviews-v3  
istio/examples-bookinfo-productpage-v1
```

该应用架构图如下：

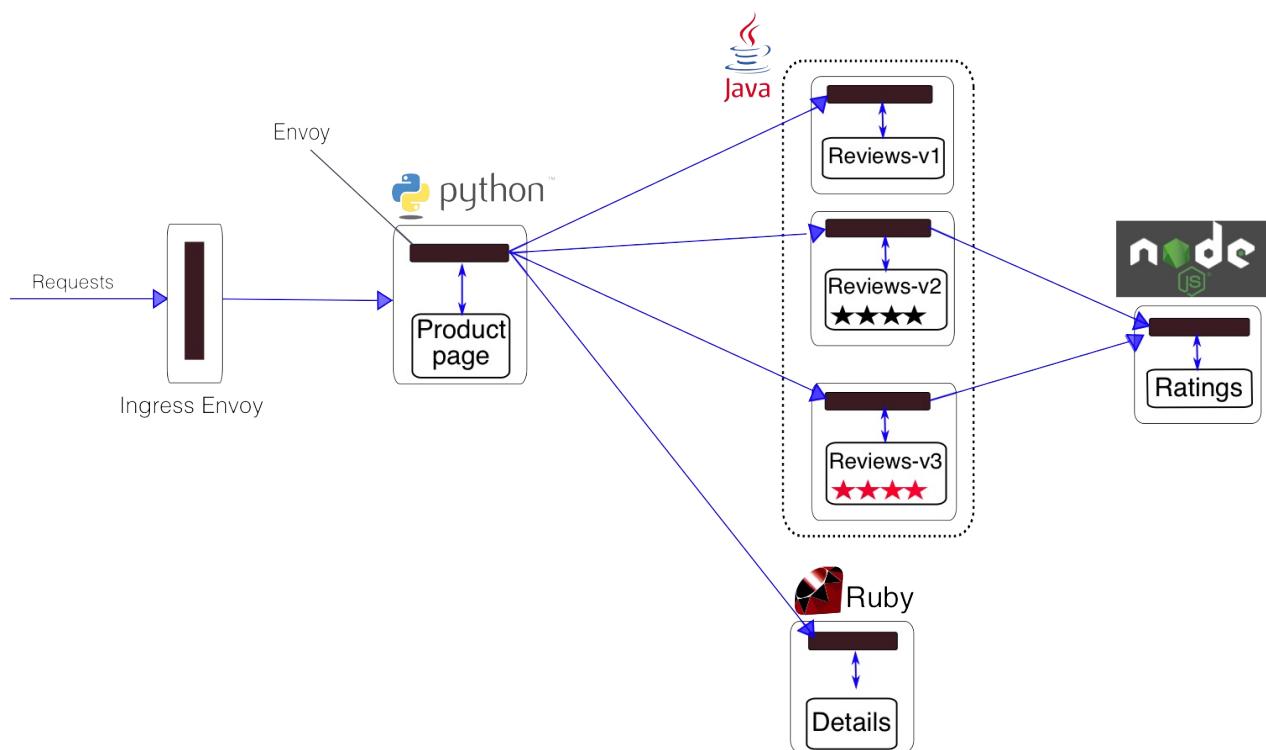


Figure: BookInfo Sample 应用架构图

部署应用

```
kubectl create -f <(istioctl kube-inject -f samples/apps/bookinfo/bookinfo.yaml)
```

5.1.1 Istio

Istio kube-inject 命令会在 bookinfo.yaml 文件中增加Envoy sidecar信息。参考：<https://istio.io/docs/reference/commands/istioctl.html#istioctl-kube-inject>

在本机的 /etc/hosts 下增加VIP节点和 ingress.istio.io 的对应信息。具体步骤参考：[边缘节点配置](#)

在浏览器中访问<http://ingress.istio.io/productpage>

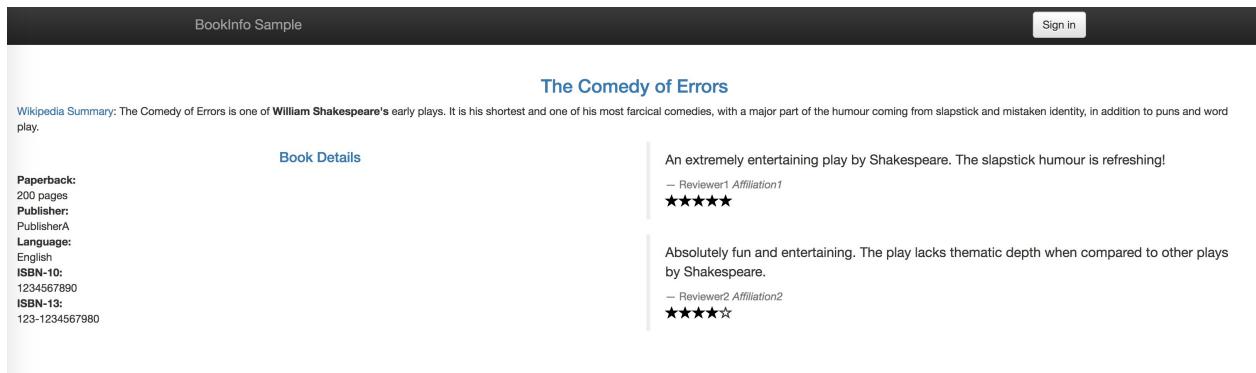


Figure: BookInfo Sample 页面

多次刷新页面，你会发现有的页面上的评论里有星级打分，有的页面就没有，这是因为我们部署了三个版本的应用，有的应用里包含了评分，有的没有。Istio根据默认策略随机将流量分配到三个版本的应用上。

查看部署的bookinfo应用中的 productpage-v1 service和deployment，查看 productpage-v1 的pod的详细json信息可以看到这样的结构：

```
$ kubectl get productpage-v1-944450470-bd530 -o json
```

见[productpage-v1-istio.json](#)文件。从详细输出中可以看到这个Pod中实际有两个容器，这里面包括了 initContainer，作为istio植入到kubernetes deployment中的sidecar。

```
"initContainers": [
  {
    "args": [
      "-p",
      "15001",
```

```
        "-u",
        "1337"
    ],
    "image": "docker.io/istio/init:0.1",
    "imagePullPolicy": "Always",
    "name": "init",
    "resources": {},
    "securityContext": {
        "capabilities": {
            "add": [
                "NET_ADMIN"
            ]
        }
    },
    "terminationMessagePath": "/dev/termination-log"
,
    "terminationMessagePolicy": "File",
    "volumeMounts": [
        {
            "mountPath": "/var/run/secrets/kubernetes
.s.io/serviceaccount",
            "name": "default-token-319f0",
            "readOnly": true
        }
    ]
},
{
    "args": [
        "-c",
        "sysctl -w kernel.core_pattern=/tmp/core.%e.
%p.%t \u0026\u0026 ulimit -c unlimited"
    ],
    "command": [
        "/bin/sh"
    ],
    "image": "alpine",
    "imagePullPolicy": "Always",
    "name": "enable-core-dump",
    "resources": {},
    "securityContext": {
```

```
        "privileged": true
    },
    "terminationMessagePath": "/dev/termination-log"

    ,
    "terminationMessagePolicy": "File",
    "volumeMounts": [
        {
            "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount",
            "name": "default-token-3l9f0",
            "readOnly": true
        }
    ]
},
],
```

监控

不断刷新productpage页面，将可以在以下几个监控中看到如下界面。

Grafana 页面

<http://grafana.istio.io>

5.1.1 Istio

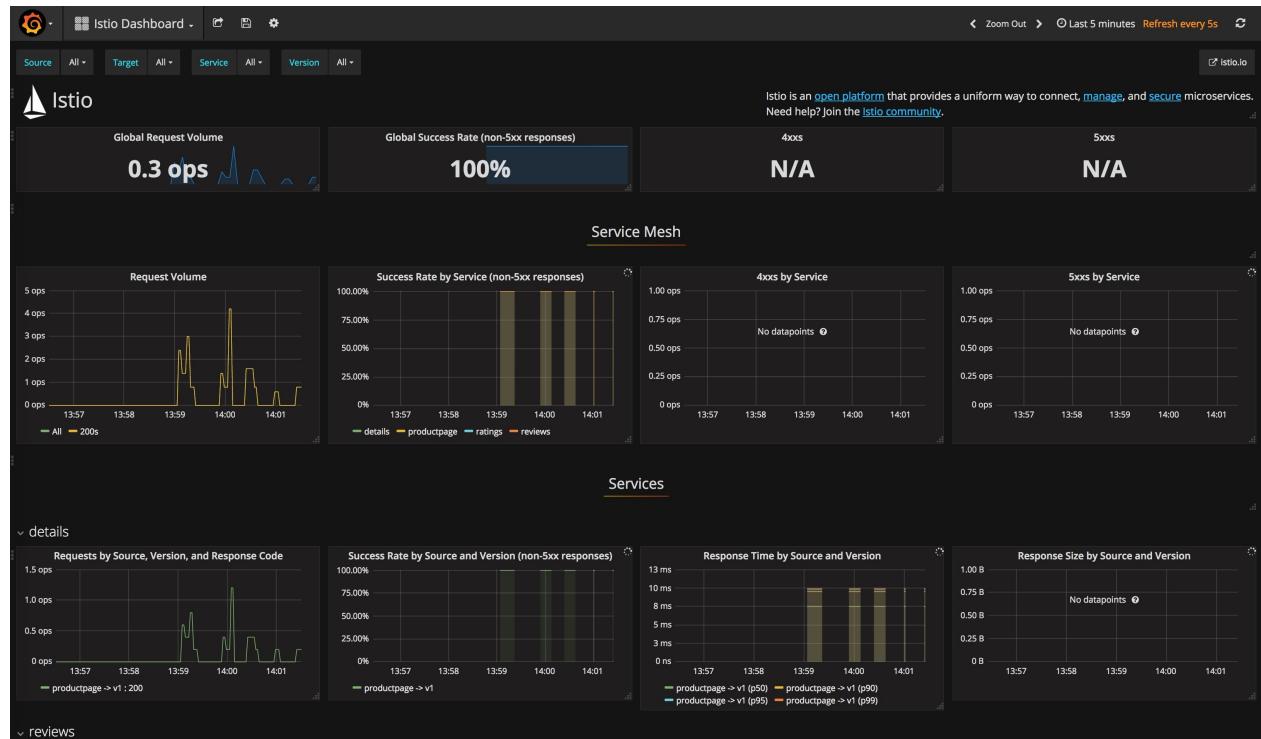


Figure: Istio Grafana界面

Prometheus 页面

<http://prometheus.istio.io>

5.1.1 Istio

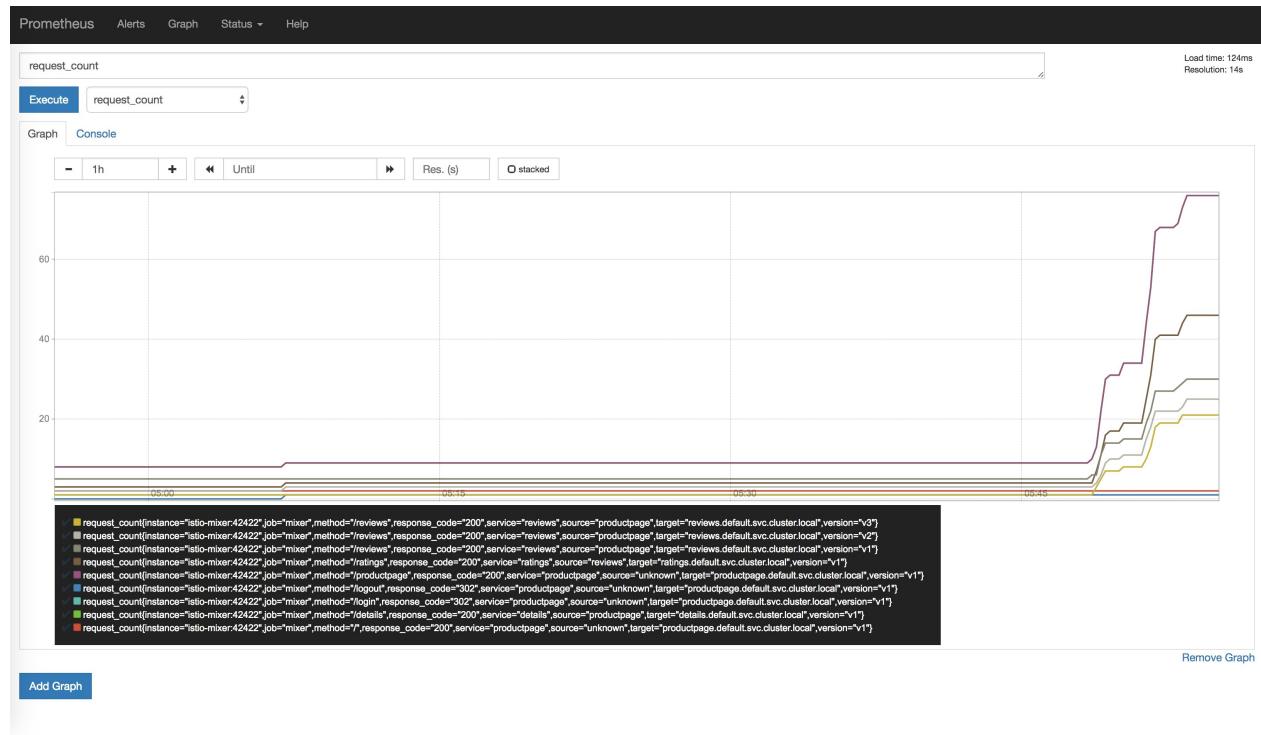


Figure: Prometheus 页面

Zipkin 页面

<http://zipkin.istio.io>

5.1.1 Istio

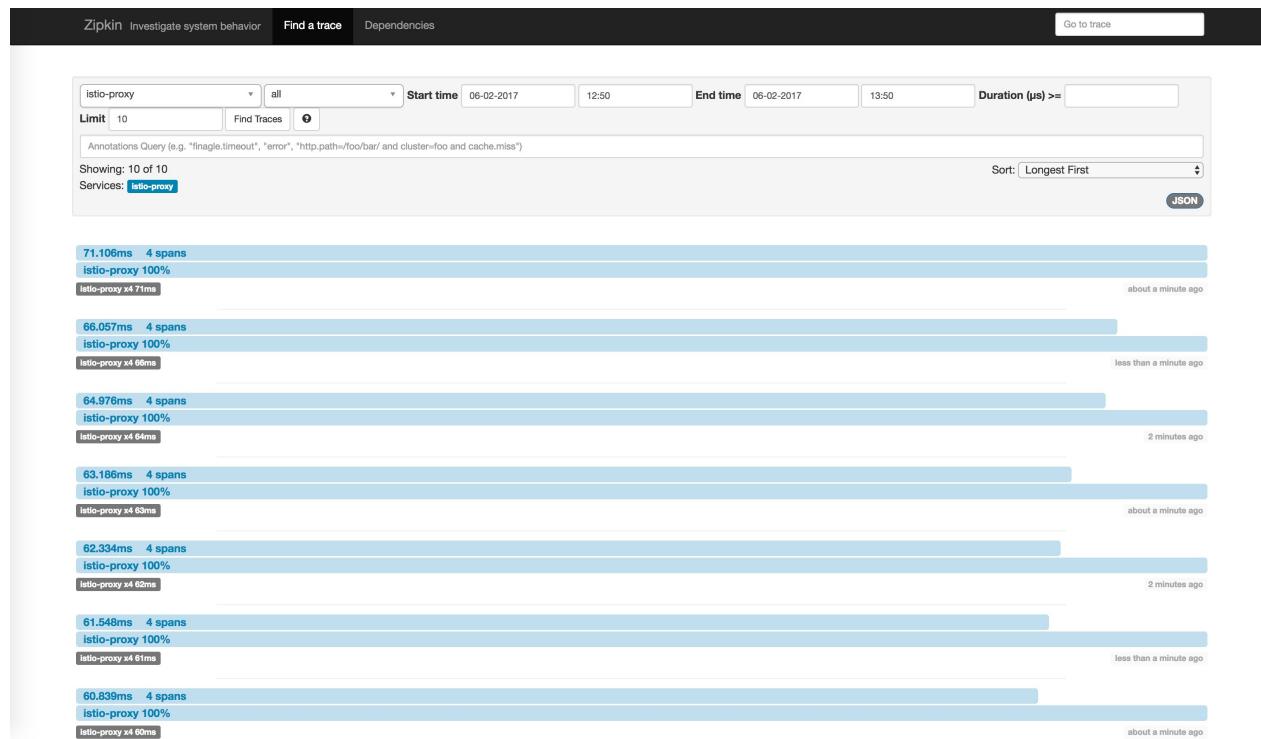


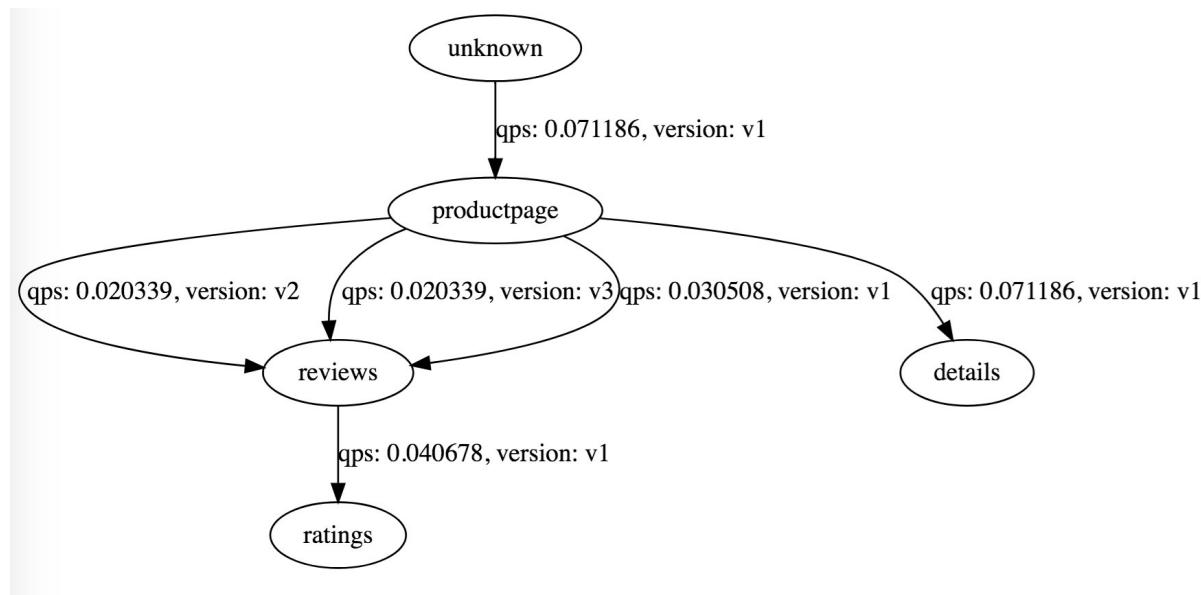
Figure: Zipkin 页面

ServiceGraph 页面

<http://servicegraph.istio.io/dotviz>

可以用来查看服务间的依赖关系。

访问<http://servicegraph.istio.io/graph>可以获得json格式的返回结果。

*Figure: ServiceGraph 页面*

更进一步

BookInfo示例中有三个版本的 `reviews`，可以使用istio来配置路由请求，将流量分摊到不同版本的应用上。参考[Configuring Request Routing](#)。

还有一些更高级的功能，我们后续将进一步探索。

参考

[Installing Istio](#)

[BookInfo sample](#)

for GitBook update 2017-08-07 13:54:27

配置请求的路由规则

在上一节[安装Istio](#)中我们创建[BookInfo](#)的示例，熟悉了Istio的基本功能，现在我们再来看一下Istio的高级特性——配置请求的路由规则。

使用Istio我们可以根据权重和**HTTP headers**来动态配置请求路由。

基于内容的路由

因为BookInfo示例部署了3个版本的评论微服务，我们需要设置一个默认路由。否则，当你多次访问应用程序时，会注意到有时输出包含星级，有时候又没有。这是因为没有明确的默认版本集，Istio将以随机方式将请求路由到服务的所有可用版本。

注意：假定您尚未设置任何路由。如果您已经为示例创建了冲突的路由规则，则需要在以下命令中使用replace而不是create。

下面这个例子能够根据网站的不同登陆用户，将流量划分到服务的不同版本和实例。跟[kubernetes](#)中的应用一样，所有的路由规则都是通过声明式的yaml配置。关于 reviews:v1 和 reviews:v2 的唯一区别是，v1没有调用评分服务，productpage页面上不会显示评分星标。

1. 将微服务的默认版本设置成v1。

```
istioctl create -f samples/apps/bookinfo/route-rule-all-v1.yaml
```

使用以下命令查看定义的路由规则。

```
istioctl get route-rules -o yaml
```

```
type: route-rule
name: details-default
namespace: default
spec:
```

5.1.1 Istio

```
destination: details.default.svc.cluster.local
precedence: 1
route:
- tags:
  version: v1
---
type: route-rule
name: productpage-default
namespace: default
spec:
destination: productpage.default.svc.cluster.local
precedence: 1
route:
- tags:
  version: v1
---
type: route-rule
name: reviews-default
namespace: default
spec:
destination: reviews.default.svc.cluster.local
precedence: 1
route:
- tags:
  version: v1
---
type: route-rule
name: ratings-default
namespace: default
spec:
destination: ratings.default.svc.cluster.local
precedence: 1
route:
- tags:
  version: v1
---
```

由于对代理的规则传播是异步的，因此在尝试访问应用程序之前，需要等待几秒钟才能将规则传播到所有pod。

2. 在浏览器中打开BookInfo URL ([http://\\$GATEWAY_URL/productpage](http://$GATEWAY_URL/productpage)，我们在上一节中使用的是 <http://ingress.istio.io/productpage>) 您应该会看到 BookInfo应用程序的产品页面显示。注意，产品页面上没有评分星，因为 `reviews:v1` 不访问评级服务。

3. 将特定用户路由到 `reviews:v2`。

为测试用户 `jason` 启用评分服务，将 `productpage` 的流量路由到 `reviews:v2` 实例上。

```
istioctl create -f samples/apps/bookinfo/route-rule-reviews-test-v2.yaml
```

确认规则生效：

```
istioctl get route-rule reviews-test-v2
```

```
destination: reviews.default.svc.cluster.local
match:
  httpHeaders:
    cookie:
      regex: ^(.*?;)?(user=jason)(;.*)?$
precedence: 2
route:
- tags:
  version: v2
```

4. 使用 `jason` 用户登陆 `productpage` 页面。

你可以看到每个刷新页面时，页面上都有一个1到5颗星的评级。如果你使用其他用户登陆的话，将因继续使用 `reviews:v1` 而看不到星级评分。

内部实现

在这个例子中，一开始使用 `istio` 将 100% 的流量发送到 BookInfo 服务的 `reviews:v1` 的实例上。然后又根据请求的 `header`（例如用户的 `cookie`）将流量选择性的发送到 `reviews:v2` 实例上。

5.1.1 Istio

验证了v2实例的功能后，就可以将全部用户的流量发送到v2实例上，或者逐步的迁移流量，如10%、20%直到100%。

如果你看了[故障注入](#)这一节，你会发现v2版本中有个bug，而在v3版本中修复了，你想将流量迁移到 reviews:v1 迁移到 reviews:v3 版本上，只需要运行如下命令：

1. 将50%的流量从 reviews:v1 转移到 reviews:v3 上。

```
istioctl replace -f samples/apps/bookinfo/route-rule-reviews-50-v3.yaml
```

注意这次使用的是 replace 命令，而不是 create ，因为该rule已经在前面创建过了。

2. 登出jason用户，或者删除测试规则，可以看到新的规则已经生效。

删除测试规则。

```
istioctl delete route-rule reviews-test-v2  
istioctl delete route-rule ratings-test-delay
```

现在的规则就是刷新 productpage 页面，50%的概率看到红色星标的评论，50%的概率看不到星标。

注意：因为使用Envoy sidecar的实现，你需要刷新页面很多次才能看到接近规则配置的概率分布，你可以将v3的概率修改为90%，这样刷新页面时，看到红色星标的概率更高。

3. 当v3版本的微服务稳定以后，就可以将100%的流量分摊到 reviews:v3 上了。

```
istioctl replace -f samples/apps/bookinfo/route-rule-reviews-v3.yaml
```

现在不论你使用什么用户登陆 productpage 页面，你都可以看到带红色星标评分的评论了。

5.1.1 Istio

Linkerd 简介

Linkerd是一个用于云原生应用的开源、可扩展的service mesh（一般翻译成服务网格，还有一种说法叫“服务啮合层”，见[Istio：用于微服务的服务啮合层](#)）。

Linkerd是什么

Linkerd的出现是为了解决像twitter、google这类超大规模生产系统的复杂性问题。Linkerd不是通过控制服务之间的通信机制来解决这个问题，而是通过在服务实例之上添加一个抽象层来解决的。

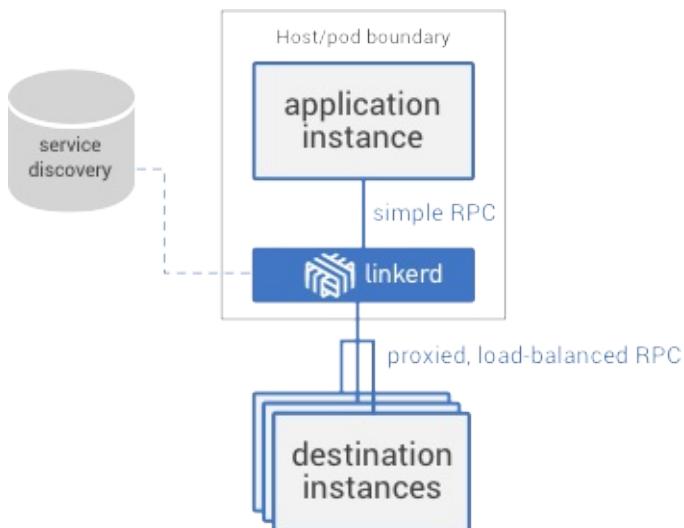


Figure: source <https://linkerd.io>

Linkerd负责跨服务通信中最困难、易出错的部分，包括延迟感知、负载平衡、连接池、TLS、仪表盘、请求路由等——这些都会影响应用程序伸缩性、性能和弹性。

如何运行

Linkerd作为独立代理运行，无需特定的语言和库支持。应用程序通常会在已知位置运行linkerd实例，然后通过这些实例代理服务调用——即不是直接连接到目标服务，服务连接到它们对应的linkerd实例，并将它们视为目标服务。

在该层上，linkerd应用路由规则，与现有服务发现机制通信，对目标实例做负载均衡——与此同时调整通信并报告指标。

通过延迟调用linkerd的机制，应用程序代码与以下内容解耦：

- 生产拓扑
- 服务发现机制
- 负载均衡和连接管理逻辑

应用程序也将从一致的全局流量控制系统中受益。这对于多语言应用程序尤其重要，因为通过库来实现这种一致性是非常困难的。

Linkerd实例可以作为sidecar（既为每个应用实体或每个主机部署一个实例）来运行。由于linkerd实例是无状态和独立的，因此它们可以轻松适应现有的部署拓扑。它们可以与各种配置的应用程序代码一起部署，并且基本不需要去协调它们。

参考

[Buoyant发布服务网格Linkerd的1.0版本](#)

[Linkerd documentation](#)

[Istio：用于微服务的服务啮合层](#)

for GitBook update 2017-08-07 13:54:27

Linkerd 使用指南

前言

Linkerd 作为一款 service mesh 与 kubernetes 结合后主要有以下几种用法：

1. 作为服务网关，可以监控 kubernetes 中的服务和实例
2. 使用 TLS 加密服务
3. 通过流量转移到持续交付
4. 开发测试环境（Eat your own dog food）、Ingress 和边缘路由
5. 给微服务做 staging
6. 分布式 tracing
7. 作为 Ingress controller
8. 使用 gRPC 更方便

以下我们着重讲解在 kubernetes 中如何使用 linkerd 作为 kubernetes 的 Ingress controller，并作为边缘节点代替 [Traefik](#) 的功能，详见 [边缘节点的配置](#)。

准备

安装测试时需要用到的镜像有：

```
buoyantio/helloworld:0.1.4
buoyantio/jenkins-plus:2.60.1
buoyantio/kubectl:v1.4.0
buoyantio/linkerd:1.1.2
buoyantio/namerd:1.1.2
buoyantio/nginx:1.10.2
linkerd/namerctl:0.8.6
openzipkin/zipkin:1.20
tutum/dnsutils:latest
```

这些镜像可以直接通过 Docker Hub 获取，我将它们下载下来并上传到了自己的私有镜像仓库 `sz-pg-oam-docker-hub-001.tendcloud.com` 中，下文中用到的镜像皆来自我的私有镜像仓库，yaml 配置见 [linkerd](#) 目录，并在使用时将配置中的镜

像地址修改为自己的。

部署

首先需要先创建 RBAC，因为使用 namerd 和 ingress 时需要用到。

```
$ kubectl create -f linkerd-rbac-beta.yml
```

Linkerd 提供了 Jenkins 示例，在部署的时候使用以下命令：

```
$ kubectl create -f jenkins-rbac-beta.yml  
$ kubectl create -f jenkins.yml
```

访问 <http://jenkins.jimmysong.io>

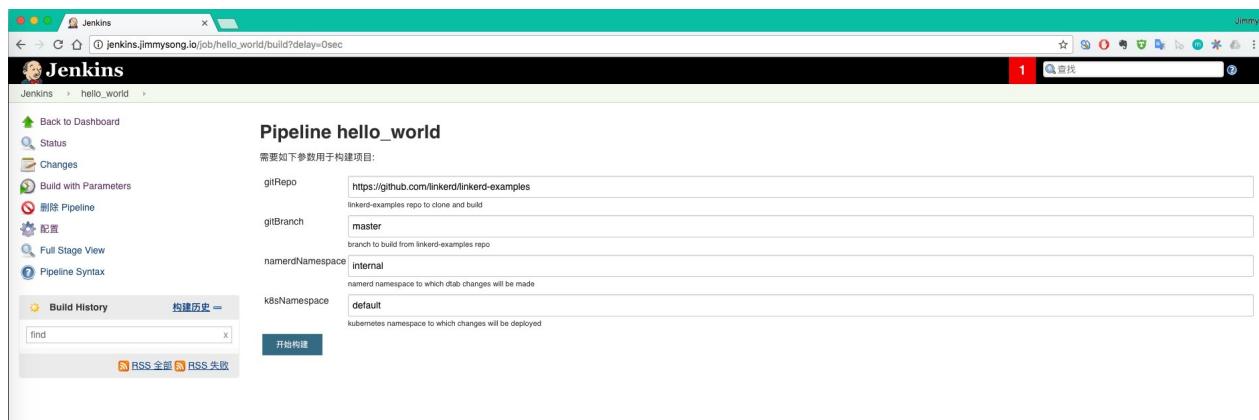


Figure: Jenkins pipeline

5.1.2 Linkerd

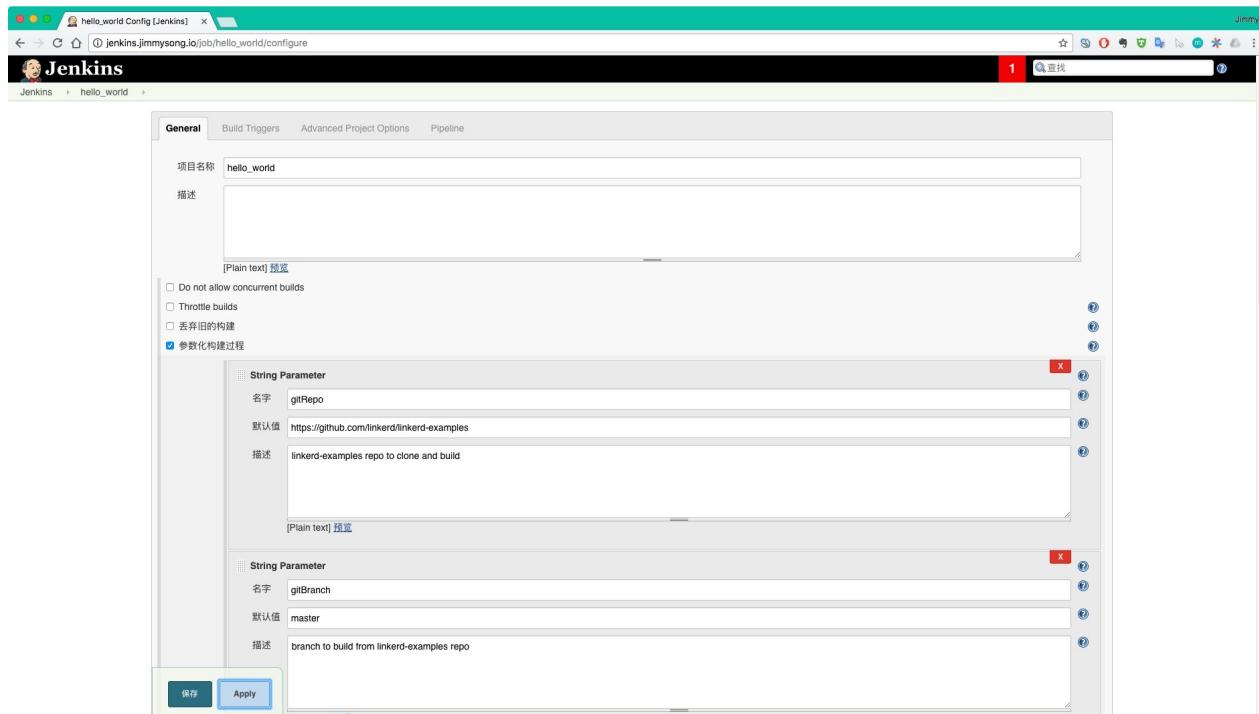


Figure: Jenkins config

注意：要访问 Jenkins 需要在 Ingress 中增加配置，下文会提到。

在 kubernetes 中使用 Jenkins 的时候需要注意 Pipeline 中的配置：

```
def currentVersion = getCurrentVersion()
def newVersion = getNextVersion(currentVersion)
def frontendIp = kubectl("get svc 15d -o jsonpath=\"{.status
.loadBalancer.ingress[0].*}\\"").trim()
def originalDst = getDst(getDtab())
```

`frontendIP` 的地址要配置成 service 的 Cluster IP，因为我们没有用到 LoadBalancer。

需要安装 namerd，namerd 负责 dtab 信息的存储，当然也可以存储在 etcd、consul 中。dtab 保存的是路由规则信息，支持递归解析，详见 [dtab](#)。

流量切换主要是通过 [dtab](#) 来实现的，通过在 HTTP 请求的 header 中增加 `15d-dtab` 和 `Host` 信息可以对流量分离到 kubernetes 中的不同 service 上。

遇到的问题

Failed with the following error(s) Error signal dtab is already marked as being deployed!

因为该 dtab entry 已经存在，需要删除后再运行。

访问 <http://namerd.jimmysong.io>

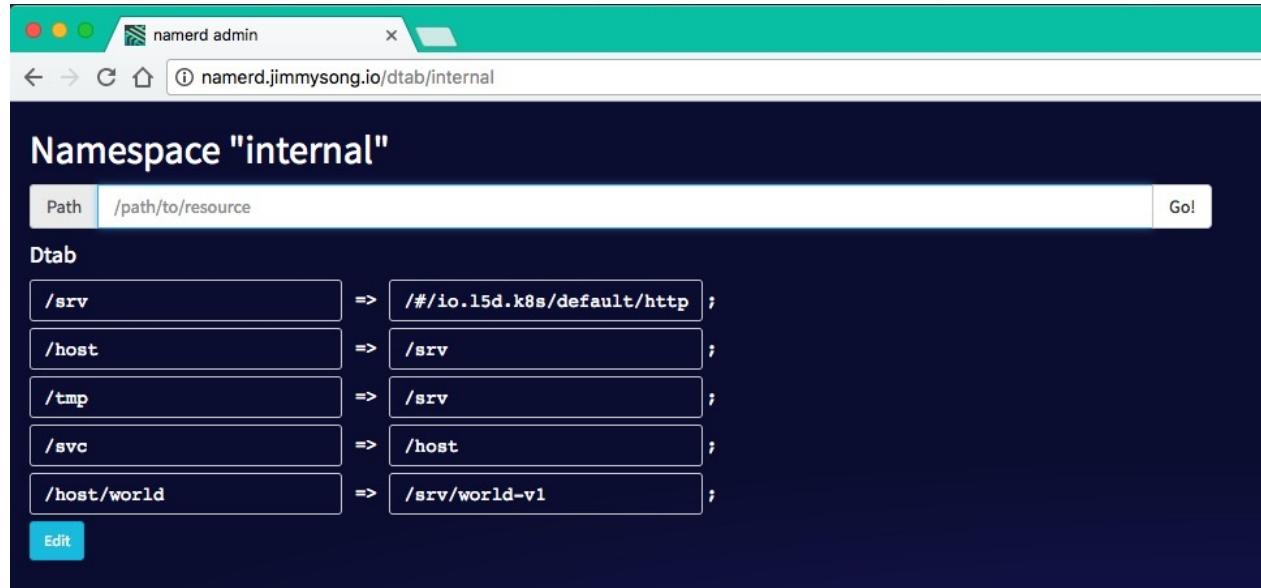


Figure: namerd

dtab 保存在 namerd 中，该页面中的更改不会生效，需要使用命令行来操作。

使用 [namerctl](#) 来操作。

```
$ namerctl --base-url http://namerd-backend.jimmysong.io dtab update internal file
```

注意：update 时需要将更新文本先写入文件中。

部署 Linkerd

直接使用 yaml 文件部署，注意修改镜像仓库地址。

5.1.2 Linkerd

```
# 创建 namerd
$ kubectl create -f namerd.yaml
# 创建 ingress
$ kubectl create -f linkerd-ingress.yaml
# 创建测试服务 hello-world
$ kubectl create -f hello-world.yaml
# 创建 API 服务
$ kubectl create -f api.yaml
# 创建测试服务 world-v2
$ kubectl create -f world-v2.yaml
```

为了在本地调试 linkerd，我们将 linkerd 的 service 加入到 ingress 中，详见 [边缘节点配置](#)。

在 Ingress 中增加如下内容：

5.1.2 Linkerd

```
- host: linkerd.jimmysong.io
  http:
    paths:
      - path: /
        backend:
          serviceName: 15d
          servicePort: 9990
- host: linkerd-viz.jimmysong.io
  http:
    paths:
      - path: /
        backend:
          serviceName: linkerd-viz
          servicePort: 80
- host: 15d.jimmysong.io
  http:
    paths:
      - path: /
        backend:
          serviceName: 15d
          servicePort: 4141
- host: jenkins.jimmysong.io
  http:
    paths:
      - path: /
        backend:
          serviceName: jenkins
          servicePort: 80
```

在本地 /etc/hosts 中添加如下内容：

```
172.20.0.119 linkerd.jimmysong.io
172.20.0.119 linkerd-viz.jimmysong.io
172.20.0.119 15d.jimmysong.io
```

测试路由功能

使用 curl 简单测试。

单条测试

```
$ curl -s -H "Host: www.hello.world" 172.20.0.120:4141
Hello (172.30.60.14) world (172.30.71.19)!!%
```

请注意请求返回的结果，表示访问的是 `world-v1 service`。

```
$ for i in $(seq 0 10000);do echo $i;curl -s -H "Host: www.hello
.world" 172.20.0.120:4141;done
```

使用 `ab test`。

```
$ ab -c 4 -n 10000 -H "Host: www.hello.world" http://172.20.0.1
20:4141/
This is ApacheBench, Version 2.3 <$Revision: 1757674 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeust
ech.net/
Licensed to The Apache Software Foundation, http://www.apache.or
g/

Benchmarking 172.20.0.120 (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests

Server Software:
Server Hostname:      172.20.0.120
Server Port:          4141
```

```

Document Path: /
Document Length: 43 bytes

Concurrency Level: 4
Time taken for tests: 262.505 seconds
Complete requests: 10000
Failed requests: 0
Total transferred: 2210000 bytes
HTML transferred: 430000 bytes
Requests per second: 38.09 [/sec] (mean)
Time per request: 105.002 [ms] (mean)
Time per request: 26.250 [ms] (mean, across all concurrent
requests)
Transfer rate: 8.22 [Kbytes/sec] received

```

Connection Times (ms)

	min	mean	[+/-sd]	median	max
Connect:	36	51	91.1	39	2122
Processing:	39	54	29.3	46	585
Waiting:	39	52	20.3	46	362
Total:	76	105	96.3	88	2216

Percentage of the requests served within a certain time (ms)

50%	88
66%	93
75%	99
80%	103
90%	119
95%	146
98%	253
99%	397
100%	2216 (longest request)

监控 kubernets 中的服务与实例

访问 <http://linkerd.jimmysong.io> 查看流量情况

Outcoming

5.1.2 Linkerd



Figure: linkerd 监控

Incoming

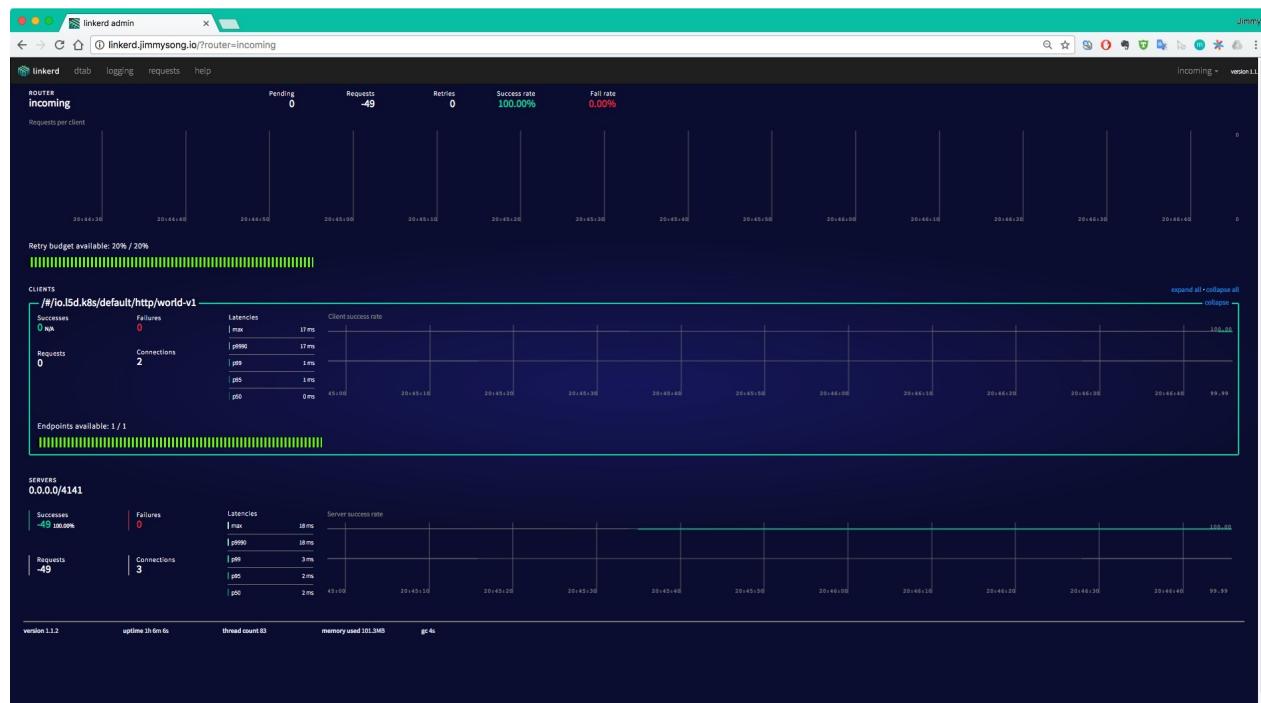


Figure: linkerd 监控

5.1.2 Linkerd

访问 <http://linkerd-viz.jimmysong.io> 查看应用 metric 监控



Figure: linkerd 性能监控

测试路由

测试在 http header 中增加 dtab 规则。

```
$ curl -H "Host: www.hello.world" -H "15d-dtab:/host/world => /s  
rv/world-v2;" 172.20.0.120:4141  
Hello (172.30.60.14) earth (172.30.94.40)!!
```

请注意调用返回的结果，表示调用的是 world-v2 的 service。

另外再对比 ab test 的结果与 linker-viz 页面上的结果，可以看到结果一致。

但是我们可能不想把该功能暴露给所有人，所以可以在前端部署一个 nginx 来过滤 header 中的 15d-dtab 打头的字段，并通过设置 cookie 的方式来替代 header 里的 15d-dtab 字段。

```
$ http_proxy=http://172.20.0.120:4141 curl -s http://hello  
Hello (172.30.60.14) world (172.30.71.19)!!
```

将 Linkerd 作为 Ingress controller

将 Linkerd 作为 kubernetes ingress controller 的方式跟将 Trafik 作为 ingress controller 的过程完全一样，可以直接参考 [边缘节点配置](#)。

架构如下图所示。

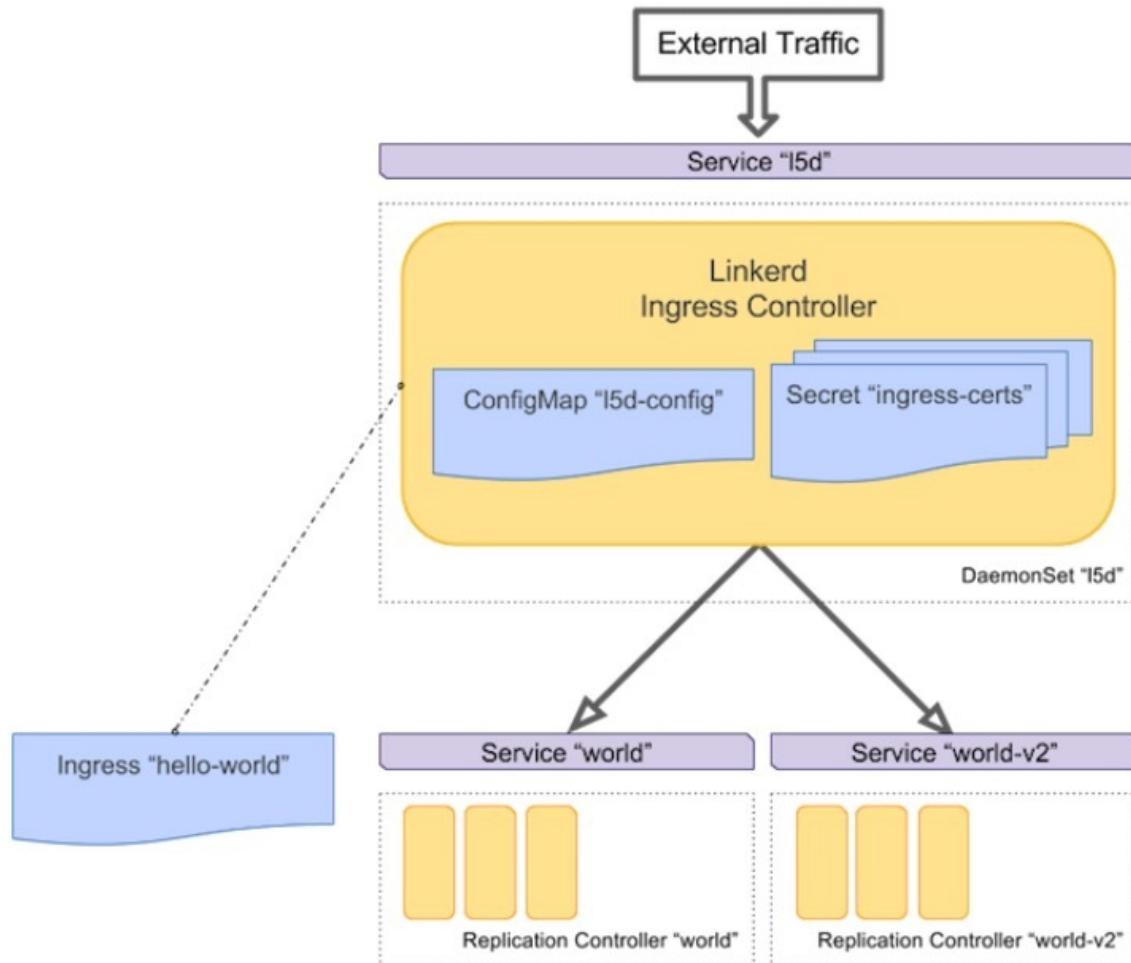


Figure: Linkerd ingress controller

(图片来自 *A Service Mesh for Kubernetes - Buoyant.io*)

当然可以绕过 kubernetes ingress controller 直接使用 linkerd 作为边界路由，通过 dtab 和 linkerd 前面的 nginx 来路由流量。

参考

<https://github.com/linkerd/linkerd-examples/>

A Service Mesh for Kubernetes

dtab

for GitBook update 2017-08-07 13:54:27

5.1.2 Linkerd

微服务中的服务发现

在单体架构时，因为服务不会经常和动态迁移，所有服务地址可以直接在配置文件中配置，所以也不会有服务发现的问题。但是对于微服务来说，应用的拆分，服务之间的解耦，和服务动态扩展带来的服务迁移，服务发现就成了微服务中的一个关键问题。

服务发现分为客户端服务发现和服务端服务发现两种，架构如下图所示。

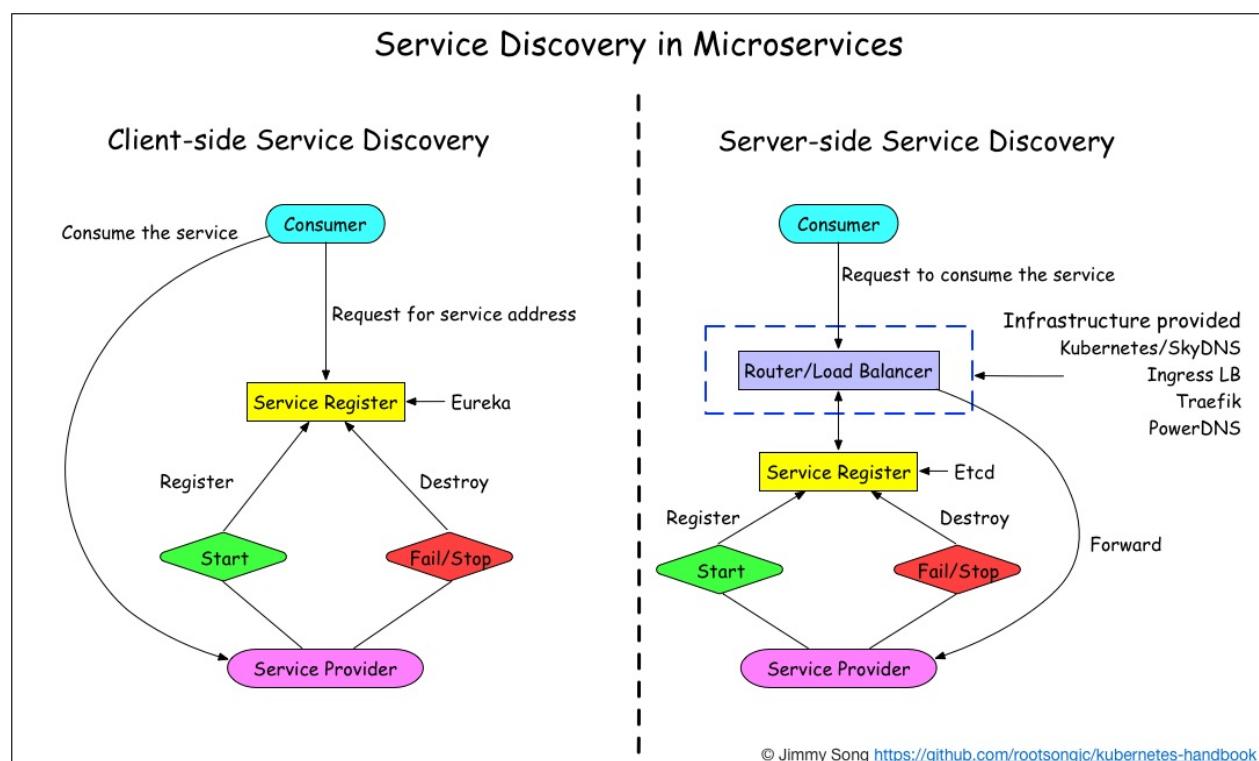


Figure: 微服务中的服务发现

这两种架构都各有利弊，我们拿客户端服务发现软件Eureka和服务端服务发现架构Kubernetes/SkyDNS+Ingress LB+Traefik+PowerDNS为例说明。

服务发现方案	Pros	Cons
Eureka	使用简单，适用于java语言开发的项目，比服务端服务发现少一次网络跳转	对非Java语言的支持不够好，Consumer需要内置特定的服务发现客户端和发现逻辑
Kubernetes	Consumer无需关注服务发现具体细节，只需知道服务的DNS域名即可	需要基础设施支撑，多了一次网络跳转，可能有性能损失

参考

谈服务发现的背景、架构以及落地方案

for GitBook update 2017-08-07 13:54:27

大数据

Kubernetes community中已经有了一个[Big data SIG](#)，大家可以通过这个SIG了解kubernetes结合大数据的应用。

其实在Swarm、Mesos、kubernetes这三种流行的容器编排调度架构中，Mesos对于大数据应用支持是最好的，spark原生就是运行在mesos上的，当然也可以容器化运行在kubernetes上。

[Spark on Kubernetes](#)

for GitBook update 2017-08-07 13:54:27

Spark on Kubernetes

时速云上提供的镜像docker pull
index.tenxcloud.com/google_containers/spark:1.5.2_v1都下载不下来。
因此我自己编译的spark的镜像。

编译好后上传到了时速云镜像仓库

```
index.tenxcloud.com/jimmy/spark:1.5.2_v1  
index.tenxcloud.com/jimmy/zeppelin:0.7.1
```

代码和使用文档见Github地址：<https://github.com/rootsongjc/spark-on-kubernetes>

在Kubernetes上启动spark

创建名为spark-cluster的namespace，所有操作都在该namespace中进行。

所有yaml文件都在 manifests 目录下。

```
$ kubectl create -f manifests/
```

将会启动一个拥有三个worker的spark集群和zeppelin。

同时在该namespace中增加ingress配置，将spark的UI和zeppelin页面都暴露出来，可以在集群外部访问。

该ingress后端使用traefik。

访问spark

通过上面对ingress的配置暴露服务，需要修改本机的/etc/hosts文件，增加以下配置，使其能够解析到上述service。

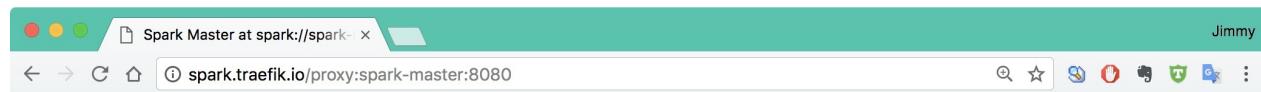
5.2.1 Spark on Kubernetes

```
172.20.0.119 zeppelin.traefik.io  
172.20.0.119 spark.traefik.io
```

172.20.0.119是我设置的VIP地址，VIP的设置和traefik的配置请查看[kubernetes-handbook](#)。

spark ui

访问<http://spark.traefik.io>



1.5.2 Spark Master at spark://spark-master:7077

- **URL:** spark://spark-master:7077
- **REST URL:** spark://spark-master:6066 (cluster mode)
- **Alive Workers:** 3
- **Cores in use:** 120 Total, 0 Used
- **Memory in use:** 373.9 GB Total, 0.0 B Used
- **Applications:** 0 Running, 0 Completed
- **Drivers:** 0 Running, 0 Completed
- **Status:** ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20170509141349-172.30.60.18-36687	172.30.60.18:36687	ALIVE	40 (0 Used)	124.6 GB (0.0 B Used)
worker-20170509141407-172.30.71.12-37758	172.30.71.12:37758	ALIVE	40 (0 Used)	124.6 GB (0.0 B Used)
worker-20170509141410-172.30.94.15-33332	172.30.94.15:33332	ALIVE	40 (0 Used)	124.6 GB (0.0 B Used)

Running Applications

Application ID Name Cores Memory per Node Submitted Time User State Duration

Completed Applications

Application ID Name Cores Memory per Node Submitted Time User State Duration

Figure: spark master ui

zeppelin ui

访问<http://zeppelin.traefik.io>

5.2.1 Spark on Kubernetes

The screenshot shows the Zeppelin web interface running on a Mac OS X browser. The title bar says "zeppelin.traefik.io/#/" and the user is "Jimmy". The main content area features a large blue and white graphic of a zeppelin airship. The text "Welcome to Zeppelin!" is prominently displayed. Below it, a subtext reads: "Zeppelin is web-based notebook that enables interactive data analytics. You can make beautiful data-driven, interactive, collaborative document with SQL, code and even more!". On the left, there's a sidebar with sections for "Notebook" (containing "Import note" and "Create new note" buttons, and a "Filter" input field), "Help" (with a link to "Zeppelin documentation"), and "Community" (links to "Mailing list", "Issues tracking", and "Github").

for GitBook

update 2017-08-07 13:54:27

开发指南

for GitBook update 2017-08-07 13:54:27

配置Kubernetes开发环境

我们将在Mac上使用docker环境编译kubernetes。

安装依赖

```
brew install gnu-tar
```

Docker环境，至少需要给容器分配4G内存，在低于3G内存的时候可能会编译失败。

执行编译

切换目录到kubernetes源码的根目录下执行：

```
./build/run.sh make 可以在docker中执行跨平台编译出二进制文件。
```

需要用的的docker镜像：

```
gcr.io/google_containers/kube-cross:v1.7.5-2
```

我将该镜像备份到时速云上了，可供大家使用：

```
index.tenxcloud.com/jimmy/kube-cross:v1.7.5-2
```

该镜像基于Ubuntu构建，大小2.15G，编译环境中包含以下软件：

- Go1.7.5
- etcd
- protobuf
- g++
- 其他golang依赖包

在我自己的电脑上的整个编译过程大概要半个小时。

6.1 开发环境搭建

编译完成的二进制文件在 `/_output/local/go/bin/` 目录下。

for GitBook update 2017-08-07 13:54:27

Kubernetes 测试

单元测试

单元测试仅依赖于源代码，是测试代码逻辑是否符合预期的最简单方法。

运行所有的单元测试

```
make test
```

仅测试指定的**package**

```
# 单个package  
make test WHAT=./pkg/api  
# 多个packages  
make test WHAT=./pkg/{api,kubelet}
```

或者，也可以直接用 `go test`

```
go test -v k8s.io/kubernetes/pkg/kubelet
```

仅测试指定**package**的某个测试**case**

```
# Runs TestValidatePod in pkg/api/validation with the verbose flag set  
make test WHAT=./pkg/api/validation KUBE_GOFLAGS="-v" KUBE_TEST_ARGS=' -run ^TestValidatePod$'  
  
# Runs tests that match the regex ValidatePod|ValidateConfigMap in pkg/api/validation  
make test WHAT=./pkg/api/validation KUBE_GOFLAGS="-v" KUBE_TEST_ARGS="-run ValidatePod\|ValidateConfigMap$"
```

或者直接用 `go test`

```
go test -v k8s.io/kubernetes/pkg/api/validation -run ^TestValida  
tePod$
```

并行测试

并行测试是root out flakes的一种有效方法：

```
# Have 2 workers run all tests 5 times each (10 total iterations  
).  
make test PARALLEL=2 ITERATION=5
```

生成测试报告

```
make test KUBE_COVER=y
```

Benchmark 测试

```
go test ./pkg/apiserver -benchmem -run=XXX -bench=BenchmarkWatch
```

集成测试

Kubernetes集成测试需要安装etcd（只要按照即可，不需要启动），比如

```
hack/install-etcd.sh # Installs in ./third_party/etcd  
echo export PATH="\$PATH:$(pwd)/third_party/etcd" >> ~/.profile  
# Add to PATH
```

集成测试会在需要的时候自动启动etcd和kubernetes服务，并运行[test/integration](#)里面的测试。

运行所有集成测试

```
make test-integration # Run all integration tests.
```

指定集成测试用例

```
# Run integration test TestPodUpdateActiveDeadlineSeconds with the verbose flag set.
make test-integration KUBE_GOFLAGS="-v" KUBE_TEST_ARGS="-run ^TestPodUpdateActiveDeadlineSeconds$"
```

End to end (e2e) 测试

End to end (e2e) 测试模拟用户行为操作Kubernetes，用来保证Kubernetes服务或集群的行为完全符合设计预期。

在开启e2e测试之前，需要先编译测试文件，并设置KUBERNETES_PROVIDER（默认为gce）：

```
make WHAT='test/e2e/e2e.test'
make ginkgo
export KUBERNETES_PROVIDER=local
```

启动cluster，测试，最后停止cluster

```
# build Kubernetes, up a cluster, run tests, and tear everything down
go run hack/e2e.go -- -v --build --up --test --down
```

仅测试指定的用例

```
go run hack/e2e.go -v -test --test_args='--ginkgo.focus=Kubectl\sclient\[k8s\.io\]\sKubectl\srolling\update\sshould\ssupport\srolling\update\sto\ssame\simage\s\[Conformance\]$'
```

略过测试用例

```
go run hack/e2e.go -- -v --test --test_args="--ginkgo.skip=Pods.*env"
```

并行测试

```
# Run tests in parallel, skip any that must be run serially
GINKGO_PARALLEL=y go run hack/e2e.go --v --test --test_args="--ginkgo.skip=\[Serial\]"

# Run tests in parallel, skip any that must be run serially and
# keep the test namespace if test failed
GINKGO_PARALLEL=y go run hack/e2e.go --v --test --test_args="--ginkgo.skip=\[Serial\] --delete-namespace-on-failure=false"
```

清理测试

```
go run hack/e2e.go -- -v --down
```

有用的 **-ctl**

```
# -ctl can be used to quickly call kubectl against your e2e cluster. Useful for
# cleaning up after a failed test or viewing logs. Use -v to avoid suppressing
# kubectl output.
go run hack/e2e.go -- -v -ctl='get events'
go run hack/e2e.go -- -v -ctl='delete pod foobar'
```

Federation e2e 测试

```

export FEDERATION=true
export E2E_ZONES="us-central1-a us-central1-b us-central1-f"
# or export FEDERATION_PUSH_REPO_BASE="quay.io/colin_hom"
export FEDERATION_PUSH_REPO_BASE="gcr.io/${GCE_PROJECT_NAME}"

# build container images
KUBE_RELEASE_RUN_TESTS=n KUBE_FASTBUILD=true go run hack/e2e.go
-- -v -build

# push the federation container images
build/push-federation-images.sh

# Deploy federation control plane
go run hack/e2e.go -- -v --up

# Finally, run the tests
go run hack/e2e.go -- -v --test --test_args="--ginkgo.focus=\[Feature:Federation\]"

# Don't forget to teardown everything down
go run hack/e2e.go -- -v --down

```

可以用 `cluster/log-dump.sh <directory>` 方便的下载相关日志，帮助排查测试中碰到的问题。

Node e2e 测试

Node e2e 仅测试 Kubelet 的相关功能，可以在本地或者集群中测试

```

export KUBERNETES_PROVIDER=local
make test-e2e-node FOCUS="InitContainer"
make test_e2e_node TEST_ARGS="--experimental-cgroups-per-qos=true"

```

补充说明

借助kubectl的模版可以方便获取想要的数据，比如查询某个container的镜像的方法为

```
kubectl get pods nginx-4263166205-ggst4 -o template '--template={{if (exists . "status" "containerStatuses")}}{{range .status.containerStatuses}}{{if eq .name "nginx"}}{{.image}}{{end}}{{end}}{{end}}
```

参考文档

- [Kubernetes testing](#)
- [End-to-End Testing](#)
- [Node e2e test](#)
- [How to write e2e test](#)
- [Coding Conventions](#)

client-go示例

访问kubernetes集群有几下几种方式：

方式	特点	支持者
Kubernetes dashboard	直接通过Web UI进行操作，简单直接，可定制化程度低	官方支持
kubectl	命令行操作，功能最全，但是比较复杂，适合对其进行进一步的分装，定制功能，版本适配最好	官方支持
client-go	从kubernetes的代码中抽离出来的客户端包，简单易用，但需要小心区分kubernetes的API版本	官方支持
client-python	python客户端，kubernetes-incubator	官方支持
Java client	fabric8中的一部分，kubernetes的java客户端	redhat

下面，我们基于[client-go](https://github.com/rootsongjc/kubernetes-client-go-sample)，对Deployment升级镜像的步骤进行了定制，通过命令行传递一个Deployment的名字、应用容器名和新image名字的方式来升级。代码和使用方式见 <https://github.com/rootsongjc/kubernetes-client-go-sample>。

kubernetes-client-go-sample

代码如下：

```
package main

import (
    "flag"
    "fmt"
    "os"
    "path/filepath"

    "k8s.io/apimachinery/pkg/api/errors"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/tools/clientcmd"
```

```

)

func main() {
    var kubeconfig *string
    if home := homeDir(); home != "" {
        kubeconfig = flag.String("kubeconfig", filepath.Join(home,
            ".kube", "config"), "(optional) absolute path to the kubeconfig file")
    } else {
        kubeconfig = flag.String("kubeconfig", "", "absolute path to the kubeconfig file")
    }
    deploymentName := flag.String("deployment", "", "deployment name")
    imageName := flag.String("image", "", "new image name")
    appName := flag.String("app", "app", "application name")

    flag.Parse()
    if *deploymentName == "" {
        fmt.Println("You must specify the deployment name.")
        os.Exit(0)
    }
    if *imageName == "" {
        fmt.Println("You must specify the new image name.")
        os.Exit(0)
    }
    // use the current context in kubeconfig
    config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
    if err != nil {
        panic(err.Error())
    }

    // create the clientset
    clientset, err := kubernetes.NewForConfig(config)
    if err != nil {
        panic(err.Error())
    }
    deployment, err := clientset.AppsV1beta1().Deployments("default").Get(*deploymentName, metav1.GetOptions{})
}

```

```

    if err != nil {
        panic(err.Error())
    }
    if errors.NotFound(err) {
        fmt.Printf("Deployment not found\n")
    } else if statusError, isStatus := err.(*errors.StatusError);
    ; isStatus {
        fmt.Printf("Error getting deployment%v\n", statusError.E
rrStatus.Message)
    } else if err != nil {
        panic(err.Error())
    } else {
        fmt.Printf("Found deployment\n")
        name := deployment.GetName()
        fmt.Println("name ->", name)
        containers := &deployment.Spec.Template.Spec.Containers
        found := false
        for i := range *containers {
            c := *containers
            if c[i].Name == *appName {
                found = true
                fmt.Println("Old image ->", c[i].Image)
                fmt.Println("New image ->", *imageName)
                c[i].Image = *imageName
            }
        }
        if found == false {
            fmt.Println("The application container not exist in
the deployment pods.")
            os.Exit(0)
        }
        _, err := clientset.AppsV1beta1().Deployments("default")
        .Update(deployment)
        if err != nil {
            panic(err.Error())
        }
    }
}

func homeDir() string {

```

6.3 client-go示例

```
if h := os.Getenv("HOME"); h != "" {  
    return h  
}  
return os.Getenv("USERPROFILE") // windows  
}
```

我们使用 `kubeconfig` 文件认证连接**kubernetes**集群，该文件默认的位置是 `$HOME/.kube/config`。

该代码编译后可以直接在**kubernetes**集群之外，任何一个可以连接到API server的机器上运行。

编译运行

```
$ go get github.com/rootsongjc/kubernetes-client-go-sample
$ cd $GOPATH/src/github.com/rootsongjc/kubernetes-client-go-sample
$ make
$ ./update-deployment-image -h
Usage of ./update-deployment-image:
-alsologtostderr
    log to standard error as well as files
-app string
    application name (default "app")
-deployment string
    deployment name
-image string
    new image name
-kubeconfig string
    (optional) absolute path to the kubeconfig file (default
"/Users/jimmy/.kube/config")
-log_backtrace_at value
    when logging hits line file:N, emit a stack trace
-log_dir string
    If non-empty, write log files in this directory
-logtostderr
    log to standard error instead of files
-stderrthreshold value
    logs at or above this threshold go to stderr
-v value
    log level for V logs
-vmodule value
    comma-separated list of pattern=N settings for file-filtered
logging
```

使用不存在的image更新

```
$ ./update-deployment-image -deployment filebeat-test -image sz  
-pg-oam-docker-hub-001.tendcloud.com/library/analytics-docker-te  
st:Build_9  
Found deployment  
name -> filebeat-test  
Old image -> sz-pg-oam-docker-hub-001.tendcloud.com/library/anal  
ytics-docker-test:Build_8  
New image -> sz-pg-oam-docker-hub-001.tendcloud.com/library/anal  
ytics-docker-test:Build_9
```

查看Deployment的event。

```
$ kubectl describe deployment filebeat-test
Name:           filebeat-test
Namespace:      default
CreationTimestamp:   Fri, 19 May 2017 15:12:28 +0800
Labels:         k8s-app=filebeat-test
Selector:       k8s-app=filebeat-test
Replicas:      2 updated | 3 total | 2 available | 2 unavailable
StrategyType:  RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
Conditions:
  Type    Status  Reason
  ----  -----
  Available  True   MinimumReplicasAvailable
  Progressing  True   ReplicaSetUpdated
OldReplicaSets:  filebeat-test-2365467882 (2/2 replicas created)
NewReplicaSet:   filebeat-test-2470325483 (2/2 replicas created)
Events:
FirstSeen     LastSeen     Count   From               SubObject
tPath        Type        Reason  Message
-----  -----  -----  -----
2h          1m          3      {deployment-controller }      N
ormal       ScalingReplicaSet Scaled down replica set filebe
at-test-2365467882 to 2
1m          1m          1      {deployment-controller }      N
ormal       ScalingReplicaSet Scaled up replica set filebeat
-test-2470325483 to 1
1m          1m          1      {deployment-controller }      N
ormal       ScalingReplicaSet Scaled up replica set filebeat
-test-2470325483 to 2
```

可以看到老的ReplicaSet从3个replica减少到了2个，有2个使用新配置的replica不可用，目前可用的replica是2个。

这是因为我们指定的镜像不存在，查看Deployment的pod的状态。

```
$ kubectl get pods -l k8s-app=filebeat-test
NAME                      READY   STATUS    RE
STARTS      AGE
filebeat-test-2365467882-4zwx8   2/2     Running   0
33d
filebeat-test-2365467882-rqskl   2/2     Running   0
33d
filebeat-test-2470325483-6vjbw   1/2     ImagePullBackOff  0
4m
filebeat-test-2470325483-gc14k   1/2     ImagePullBackOff  0
4m
```

我们可以看到有两个pod正在拉取image。

还原为原先的镜像

将image设置为原来的镜像。

```
$ ./update-deployment-image -deployment filebeat-test -image sz-
pg-oam-docker-hub-001.tendcloud.com/library/analytics-docker-tes
t:Build_8
Found deployment
name -> filebeat-test
Old image -> sz-pg-oam-docker-hub-001.tendcloud.com/library/anal
ytics-docker-test:Build_9
New image -> sz-pg-oam-docker-hub-001.tendcloud.com/library/anal
ytics-docker-test:Build_8
```

现在再查看Deployment的状态。

```
$ kubectl describe deployment filebeat-test
Name:           filebeat-test
Namespace:      default
CreationTimestamp:   Fri, 19 May 2017 15:12:28 +0800
Labels:          k8s-app=filebeat-test
Selector:        k8s-app=filebeat-test
Replicas:       3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
Conditions:
  Type    Status  Reason
  ----  -----
  Available  True   MinimumReplicasAvailable
  Progressing  True   NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet:  filebeat-test-2365467882 (3/3 replicas created)
)
Events:
  FirstSeen  LastSeen  Count  From                    SubObject
  tPath      Type      ReasonMessage
  -----  -----  -----  -----
  2h         8m       3      {deployment-controller } N
  ormal      ScalingReplicaSet Scaled down replica set filebe
at-test-2365467882 to 2
  8m         8m       1      {deployment-controller } N
  ormal      ScalingReplicaSet Scaled up replica set filebeat
-test-2470325483 to 1
  8m         8m       1      {deployment-controller } N
  ormal      ScalingReplicaSet Scaled up replica set filebeat
-test-2470325483 to 2
  2h         1m       3      {deployment-controller } N
  ormal      ScalingReplicaSet Scaled up replica set filebeat
-test-2365467882 to 3
  1m         1m       1      {deployment-controller } N
  ormal      ScalingReplicaSet Scaled down replica set filebe
at-test-2470325483 to 0
```

6.3 client-go示例

可以看到available的replica个数恢复成3了。

其实在使用该命令的过程中，通过kubernetes dashboard的页面上查看Deployment的状态更直观，更加方便故障排查。



Figure: 使用kubernetes dashboard进行故障排查

这也是dashboard最大的优势，简单、直接、高效。

for GitBook update 2017-08-07 13:54:27

Kubernetes社区贡献

- [Contributing guidelines](#)
- [Kubernetes Developer Guide](#)
- [Special Interest Groups](#)
- [Feature Tracking and Backlog](#)
- [Community Expectations](#)

for GitBook update 2017-08-07 13:54:27

附录

参考文档以及一些实用的资源链接。

- [Kubernetes documentation](#)
- [Awesome Kubernetes](#)
- [Kubernetes the hard way](#)
- [Kubernetes Bootcamp](#)
- [Design patterns for container-based distributed systems](#)

for GitBook update 2017-08-07 13:54:27

Docker最佳实践

本文档旨在实验Docker1.13新特性和帮助大家了解docker集群的管理和使用。

环境配置

[Docker1.13环境配置](#)

[docker源码编译](#)

网络管理

网络配置和管理是容器使用中的的一个重点和难点，对比我们之前使用的docker版本是1.11.1，docker1.13中网络模式跟之前的变动比较大，我们会花大力气讲解。

[如何创建docker network](#)

[Rancher网络探讨和扁平网络实现](#)

[swarm mode的路由网络](#)

[docker扁平化网络插件Shrike（基于docker1.11）](#)

存储管理

[Docker存储插件](#)

- [infinit](#) 被docker公司收购的法国团队开发
- [convoy](#) rancher开发的docker volume plugin
- [torus](#) 已废弃
- [flocker](#) ClusterHQ开发

日志管理

Docker提供了一系列[log drivers](#)，如fluentd、journald、syslog等。

需要配置docker engine的启动参数。

[docker logging driver](#)

创建应用

[官方文档：Docker swarm sample app overview](#)

[基于docker1.13手把手教你创建swarm app](#)

[swarm集群应用管理](#)

[使用docker-compose创建应用](#)

集群管理

我们使用docker内置的swarm来管理docker集群。

[swarm mode介绍](#)

我们推荐使用开源的docker集群管理配置方案：

- [Crane](#)：由数人云开源的基于swarmkit的容器管理软件，可以作为docker和go语言开发的一个不错入门项目
- [Rancher](#):Rancher是一个企业级的容器管理平台，可以使用Kubernetes、swarm和rancher自研的cattle来管理集群。

[Crane的部署和使用](#)

[Rancher的部署和使用](#)

资源限制

[内存资源限制](#)

[CPU资源限制](#)

[IO资源限制](#)

服务发现

下面罗列一些常见的服务发现工具。

Etcd:服务发现/全局分布式键值对存储。这是CoreOS的创建者提供的工具，面向容器和宿主机提供服务发现和全局配置存储功能。它在每个宿主机有基于http协议的API和命令行的客户端。<https://github.com/docker/etcd>

- **Cousul**：服务发现/全局分布式键值对存储。这个服务发现平台有很多高级的特性，使得它能够脱颖而出，例如：配置健康检查、ACL功能、HAProxy配置等等。
- **Zookeeper**：诞生于Hadoop生态系统里的组件，Apache的开源项目。服务发现/全局分布式键值对存储。这个工具较上面两个都比较老，提供一个更加成熟的平台和一些新特性。
- **Crypt**：加密etcd条目的项目。Crypt允许组建通过采用公钥加密的方式来保护它们的信息。需要读取数据的组件或被分配密钥，而其他组件则不能读取数据。
- **Confd**：观测键值对存储变更和新值的触发器重新配置服务。Confd项目旨在基于服务发现的变化，而动态重新配置任意应用程序。该系统包含了一个工具来检测节点中的变化、一个模版系统能够重新加载受影响的应用。
- **Vulcand**：vulcand在组件中作为负载均衡使用。它使用etcd作为后端，并基于检测变更来调整它的配置。
- **Marathon**：虽然marathon主要是调度器，它也实现了一个基本的重新加载HAProxy的功能，当发现变更时，它来协调可用的服务。
- **Frontrunner**：这个项目嵌入在marathon中对HAProxy的更新提供一个稳定的解决方案。
- **Synapse**：由Airbnb出品的，Ruby语言开发，这个项目引入嵌入式的HAProxy组件，它能够发送流量给各个组件。<http://bruth.github.io/synapse/docs/>
- **Nerve**：它被用来与synapse结合在一起为各个组件提供健康检查，如果组件不可用，nerve将更新synapse并将该组件移除出去。

插件开发

[插件开发示例-sshfs](#)

[我的docker插件开发文章](#)

[Docker17.03-CE插件开发举例](#)

[网络插件](#)

- [Contiv](#) 思科出的Docker网络插件，趟坑全记录，目前还无法上生产，1.0正式版还没出，密切关注中。
- [Calico](#) 产品化做的不错，已经有人用在生产上了。

存储插件

业界使用案例

[京东从OpenStack切换到Kubernetes的经验之谈](#)

[美团点评容器平台介绍](#)

[阿里超大规模docker化之路](#)

[TalkingData-容器技术在大数据场景下的应用Yarn on Docker](#)

[乐视云基于Kubernetes的PaaS平台建设](#)

资源编排

建议使用[kuberentes](#)，虽然比较复杂，但是专业的工具做专业的事情，将编排这么重要的生产特性绑定到[docker](#)上的风险还是很大的，我已经转投到[kubernetes](#)怀抱了，那么你呢？

[我的kubernetes探险之旅](#)

相关资源

[容器技术工具与资源](#)

[容器技术2016年总结](#)

关于

Author: [Jimmy Song](#)

rootsongjc@gmail.com

更多关于**Docker**、**MicroServices**、**Big Data**、**DevOps**、**Deep Learning**的内容
请关注[Jimmy Song's Blog](#)，将不定期更新。

for GitBook update 2017-08-07 13:54:27

问题记录

安装、使用kubernetes的过程中遇到的所有问题的记录。

推荐直接在Kubernetes的GitHub上[提issue](#)，在此记录所提交的issue。

1. Failed to start ContainerManager failed to initialise top level QOS containers #43856

重启kubelet时报错，目前的解决方法是：

1. 在docker.service配置中增加的 `--exec-opt native.cgroupdriver=systemd` 配置。
2. 手动删除slice（貌似不管用）
3. 重启主机，这招最管用☺

```
for i in $(systemctl list-unit-files --no-legend --no-pager -l | grep --color=never -o *.slice | grep kubepod); do systemctl stop $i; done
```

上面的几种方法在该bug修复前只有重启主机管用，该bug已于2017年4月27日修复，merge到了master分支，见

<https://github.com/kubernetes/kubernetes/pull/44940>

2. High Availability of Kube-apiserver #19816

API server的HA如何实现？或者说这个master节点上的服务 `api-server`、`scheduler`、`controller` 如何实现HA？目前的解决方案是什么？

目前的解决方案是api-server是无状态的可以启动多个，然后在前端再加一个nginx或者ha-proxy。而scheduler和controller都是直接用容器的方式启动的。

3.Kubelet启动时 Failed to start ContainerManager systemd version does not support ability to start a slice as transient unit

CentOS系统版本7.2.1511

kubelet启动时报错systemd版本不支持start a slice as transient unit。

尝试升级CentOS版本到7.3，看看是否可以修复该问题。

与[kubeadm init waiting for the control plane to become ready on CentOS 7.2 with kubeadm 1.6.1 #228](#)类似。

另外有一个使用systemd管理kubelet的[proposal](#)。

4.kube-proxy报错 kube-proxy[2241]: E0502 15:55:13.889842 2241 conntrack.go:42] conntrack returned error: error looking for path of conntrack: exec: "conntrack": executable file not found in \$PATH

导致的现象

kubedns启动成功，运行正常，但是service之间无法解析，kubernetes中的DNS解析异常

解决方法

CentOS中安装 conntrack-tools 包后重启kubernetes集群即可。

5. Pod stuck in terminating if it has a privileged container but has been scheduled to a node which doesn't allow privilege issue#42568

当pod被调度到无法权限不足的node上时，pod一直处于pending状态，且无法删除pod，删除时一直处于terminating状态。

kubelet中的报错信息

```
Error validating pod kube-keepalived-vip-1p62d_default(5d79ccc0-3173-11e7-bfbd-8af1e3a7c5bd) from api, ignoring: spec.containers[0].securityContext.privileged: Forbidden: disallowed by cluster policy
```

6.PVC中对Storage的容量设置不生效

使用[glusterfs做持久化存储](#)文档中我们构建了PV和PVC，当时给 `glusterfs-nginx` 的PVC设置了8G的存储限额，`nginx-dm` 这个Deployment使用了该PVC，进入该Deployment中的Pod执行测试：

```
dd if=/dev/zero of=test bs=1G count=10
```

```
root@nginx-dm-3698525684-g0mvt:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/docker-8:20-2513719-68fc56b9d12f454a80ef69105f2dc6a42ae8ba8132d51764dbf2710b7da6962e  10G  228M  9.8G  3% /
tmpfs          63G    0   63G   0% /dev
tmpfs          63G    0   63G   0% /sys/fs/cgroup
/dev/sdb4       2.0T  28G  2.0T  2% /etc/hosts
shm            64M    0   64M   0% /dev/shm
172.20.0.113:k8s-volume  1.0T    0  1.0T   0% /usr/share/nginx/html
tmpfs          63G   12K  63G   1% /run/secrets/kubernetes.io
/serviceaccount
root@nginx-dm-3698525684-g0mvt:~# dd if=/dev/zero of=test bs=1G count=11
dd: error writing 'test': No space left on device
10+0 records in
9+0 records out
10486870016 bytes (10 GB) copied, 14.6692 s, 715 MB/s
root@nginx-dm-3698525684-g0mvt:~#
```

Figure: pvc-storage-limit

从截图中可以看到创建了9个size为1G的block后无法继续创建了，已经超出了8G的限额。

参考

[Persistent Volume](#)

[Resource Design Proposals](#)

for GitBook update 2017-08-07 13:54:27

1. 在容器中获取 Pod 的 IP

通过环境变量来实现，该环境变量直接引用 resource 的状态字段，示例如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: world-v2
spec:
  replicas: 3
  selector:
    app: world-v2
  template:
    metadata:
      labels:
        app: world-v2
    spec:
      containers:
        - name: service
          image: test
          env:
            - name: POD_IP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
      ports:
        - name: service
          containerPort: 777
```

容器中可以直接使用 `POD_IP` 环境变量获取容器的 IP。

for GitBook update 2017-08-07 13:54:27