

目录

介绍

序言	1.1
课程介绍	1.2

Docker 基础

Docker 简介	2.1
镜像和容器的基本操作	2.2
Dockerfile 定制镜像	2.3
私有镜像仓库	2.4
数据共享与持久化	2.5
Docker 的网络模式	2.6

Docker 三架马车

Docker Compose	3.1
Docker Machine	3.2
Docker Swarm	3.3

Docker 实践

图形化管理和监控	4.1
Docker 的多阶段构建	4.2
Dockerfile 最佳实践	4.3

Kubernetes 基础

Kubernetes 初体验	5.1
基本概念与组件	5.2

kubeadm 搭建集群

使用 kubeadm 搭建集群环境	6.1
安装 Dashboard 插件	6.2

深入理解 Pod

YAML 文件	7.1
静态 Pod	7.2
Pod Hook	7.3
Pod 的健康检查	7.4
初始化容器	7.5

常用对象操作:

Replication Controller 与 Replica Set	8.1
Deployment	8.2
HPA	8.3
Job/CronJob	8.4
Service	8.5
ConfigMap	8.6
Secret	8.7
RBAC	8.8
部署Wordpress示例	8.9
DaemonSet 和 StatefulSet	8.10

持久化存储:

PV	9.1
PVC	9.2
StorageClass	9.3

服务发现

kubedns	10.1
ingress 安装配置	10.2
ingress tls 和 path 的使用	10.3

包管理工具 Helm

Helm 的安装使用	11.1
Helm 的基本使用	11.2
Helm 模板之内置函数和Values	11.3

Helm 模板之模板函数与管道	11.4
Helm 模板之控制流程	11.5
Helm 模板之命名模板	11.6
Helm 模板之其他注意事项	11.7
Helm Hooks	11.8

调度器

Kubernetes 调度器介绍	12.1
Kubernetes 亲和性调度	12.2

集群监控

手动安装 Prometheus	13.1
监控 Kubernetes 集群应用	13.2
监控 Kubernetes 集群节点	13.3
监控 Kubernetes 常用资源对象	13.4
Grafana 的安装使用	13.5
AlertManager 的使用	13.6
Prometheus Operator 的安装	13.7
自定义Prometheus Operator 监控项	13.8
Prometheus Operator高级配置	13.9

日志收集

日志收集架构	14.1
搭建 EFK 日志系统	14.2

CI/CD:

动态 Jenkins Slave	15.1
Jenkins Pipeline 部署 Kubernetes 应用	15.2
Jenkins BlueOcean	15.3
Harbor	15.4
Gitlab	15.5
Gitlab CI	15.6
Devops	15.7

从Docker到Kubernetes进阶

从 Docker 入门一步步迁移到 Kubernetes 的进阶课程

在线浏览：<https://www.qikqiak.com/k8s-book>

GitHub地址：<https://github.com/cnch/kubernetes-learning/>

视频课程在线地址：<https://youdianzhishi.com/course/6n8xd6/>

介绍

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，Kubernetes可以帮助你将系统自动地达到和维持在这个状态。Kubernetes作为云原生应用的基石，相当于一个云操作系统，其重要性不言而喻。



从Docker到Kubernetes进阶

之前一直有同学跟我说我 Docker 掌握得还可以，但是不知道怎么使用 Kubernetes，网上的其他关于 Kubernetes 的课程费用又太高，本书就是为你们准备的，当然如果你不了解 Docker，不了解 Kubernetes，都没有关系，我们会从 Docker 入门一步步深入，到 Kubernetes 的进阶使用的。所以大家完全没必要担心。

学完本课程以后，你将会对 Docker 和 Kubernetes 有一个更加深入的认识，我们会讲到：

- Docker 的一些常用方法，当然我们的重点会在 Kubernetes 上面

- 会用 `kubeadm` 来搭建一套 `Kubernetes` 的集群
- 理解 `Kubernetes` 集群的运行原理
- 常用的一些控制器使用方法
- 还有 `Kubernetes` 的一些调度策略
- `Kubernetes` 的运维
- 包管理工具 `Helm` 的使用
- 最后我们会实现基于 `Kubernetes` 的 CI/CD

社区&读者交流

- 博客：[阳明的博客](#)
- 微信群：`k8s` 技术圈，扫描我的微信二维码，[阳明](#)，或直接搜索微信号iEverything后拉您入群，请增加备注(k8s或kubernetes)
- 知乎专栏：[k8s技术圈](#)
- 开发者头条：[k8s技术圈](#)
- 微信公众号：扫描下面的二维码关注微信公众号 `k8s技术圈`



*k8s*公众帐号

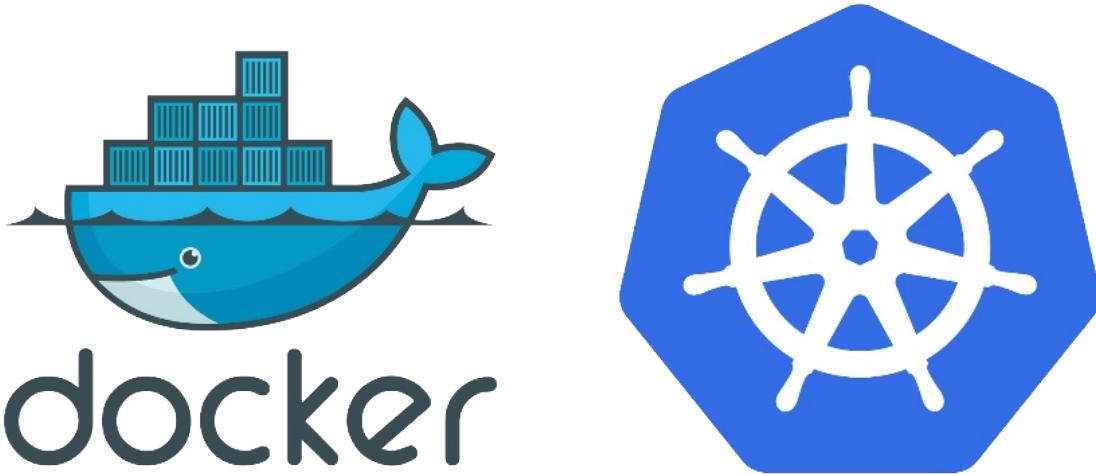
- 优点知识：[优点知识](#)是一个综合的技术学习平台，本书配套的视频教程将会发布在该平台上面，感兴趣的朋友可以扫描下发的二维码关注自己感兴趣的课程。



Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-25 14:57:47

1. 课程介绍

之前一直有同学跟我说我 Docker 掌握得还可以，但是不知道怎么使用 Kubernetes，网上的其他关于 Kubernetes 的课程费用又太高，这节课就是为你们准备的，当然如果你不了解 Docker，不了解 Kubernetes，都没有关系，我们这个课程会从 Docker 入门一步步深入，到 Kubernetes 的进阶使用的。所以大家完全没必要担心。



学完本课程以后，你将会对 Docker 和 Kubernetes 有一个更加深入的认识，我们会讲到：

- Docker 的一些常用方法，当然我们的重点会在 Kubernetes 上面
 - 会用 kubeadm 来搭建一套 Kubernetes 的集群
 - 理解 Kubernetes 集群的运行原理
 - 常用的一些控制器使用方法
 - 还有 Kubernetes 的一些调度策略
 - Kubernetes 的运维
 - 包管理工具 Helm 的使用
 - 最后我们会实现基于 Kubernetes 的 CI/CD
-

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



qrcode

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 12:57:57

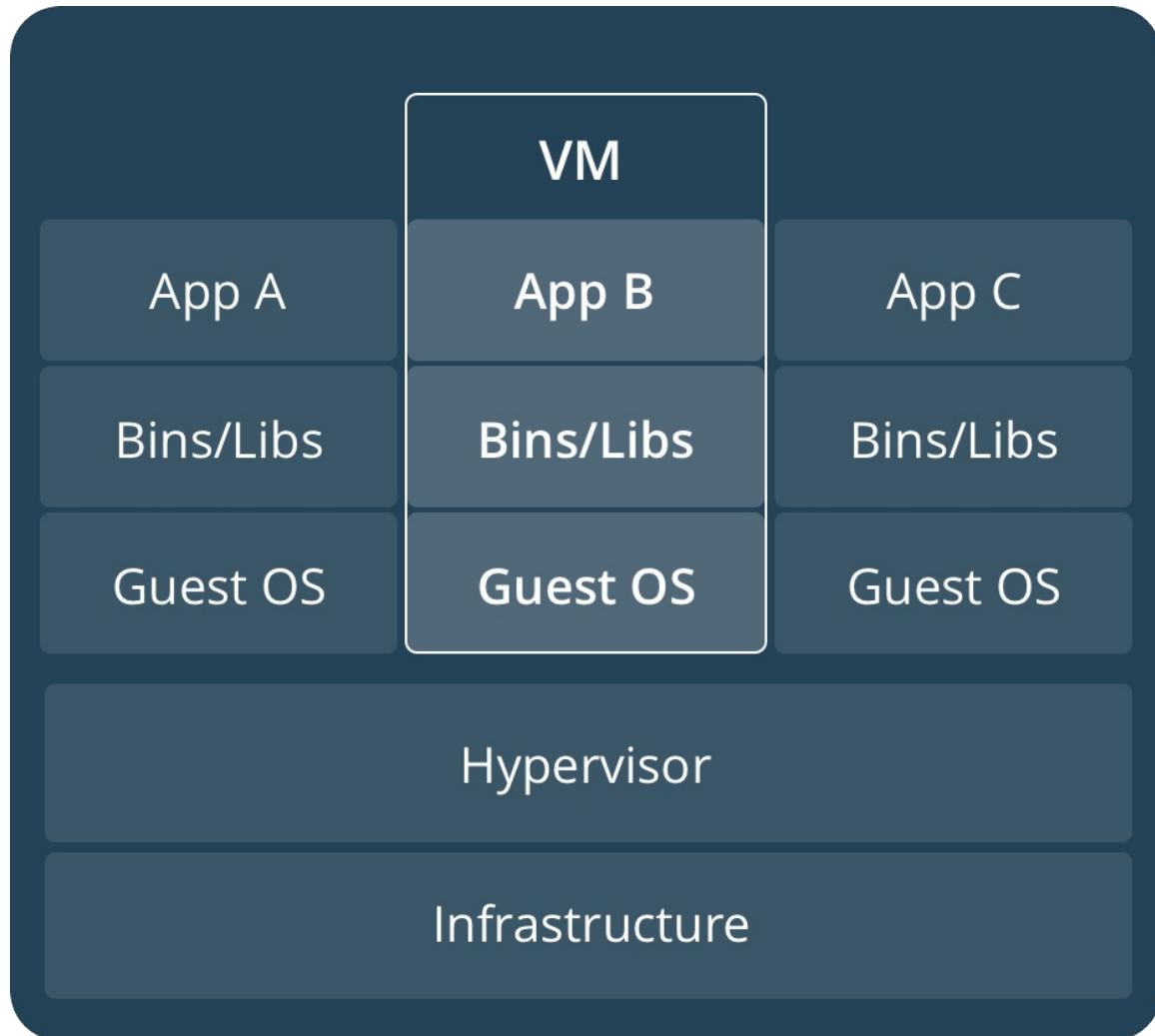
2. Docker 简介

什么是 Docker?

Docker 的英文翻译是“搬运工”的意思，他搬运的东西就是我们常说的集装箱 container，Container 里面装的是任意类型的 App，我们的开发人员可以通过 Docker 将App 变成一种标准化的、可移植的、自管理的组件，我们可以在任何主流的操作系统中开发、调试和运行。

从概念上来看 Docker 和我们传统的虚拟机比较类似，只是更加轻量级，更加方便使，Docker 和虚拟机最主要的区别有以下几点：

- 虚拟化技术依赖的是物理CPU和内存，是硬件级别的；而我们的 Docker 是构建在操作系统层面的，利用操作系统的容器化技术，所以 Docker 同样的可以运行在虚拟机上面。
- 我们知道虚拟机中的系统就是我们常说的操作系统镜像，比较复杂；而 Docker 比较轻量级，我们的可以用 Docker 部署一个独立的 Redis，就类似于在虚拟机当中安装一个 Redis 应用，但是我们用 Docker 部署的应用是完全隔离的。
- 我们都知道传统的虚拟化技术是通过快照来保存状态的；而 Docker 引入了类似于源码管理的机制，将容器的快照历史版本一一记录下来，切换成本非常之低。
- 传统虚拟化技术在构建系统的时候非常复杂；而 Docker 可以通过一个简单的 Dockerfile 文件来构建整个容器，更重要的是 Dockerfile 可以手动编写，这样应用程序开发人员可以通过发布 Dockerfile 来定义应用的环境和依赖，这样对于持续交付非常有利。



为啥要用容器？

应用容器是个啥样子呢，一个做好的应用容器长得就像一个装好了一组特定应用的虚拟机一样，比如我现在想用 Redis，那我就找个装好了 Redis 的容器就可以了，然后运行起来，我就能直接使用了。

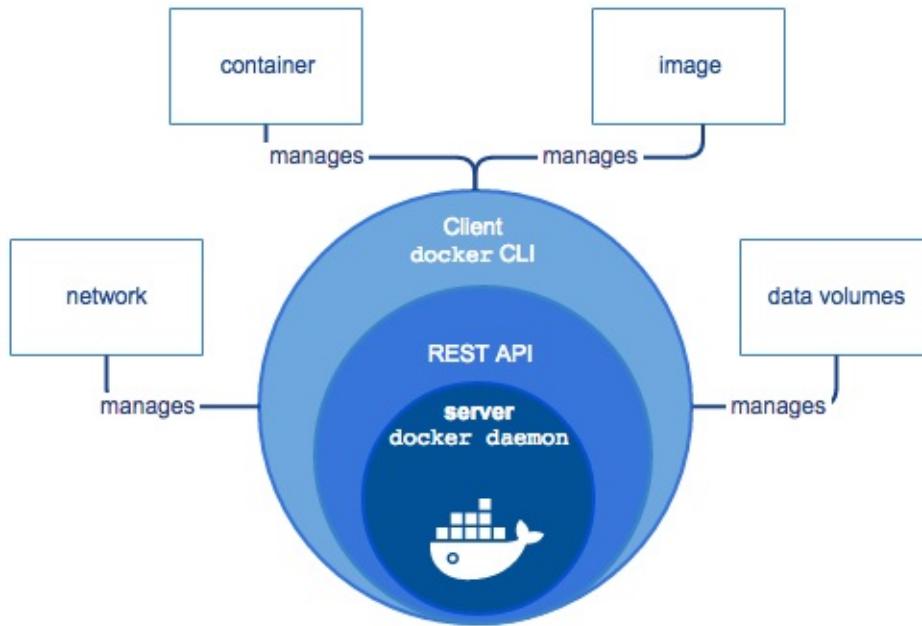
那为什么不能直接安装一个 Redis 呢？肯定是可行的，但是有的时候根据每个人电脑的不同，在安装的时候可能会报出各种各样的错误，万一你的机器中毒了，你的电脑挂了，你所有的服务都需要重新安装。但是有了 Docker 或者说有了容器就不一样了，你就相当于有了一个可以运行起来的虚拟机，只要你能运行容器，Redis 的配置就省了。而且如果你想换个电脑，没问题，很简单，直接把容器“端过来”就可以使用容器里面的服务了。

Docker Engine

Docker Engine 是一个C/S架构的应用程序，主要包含下面几个组件：

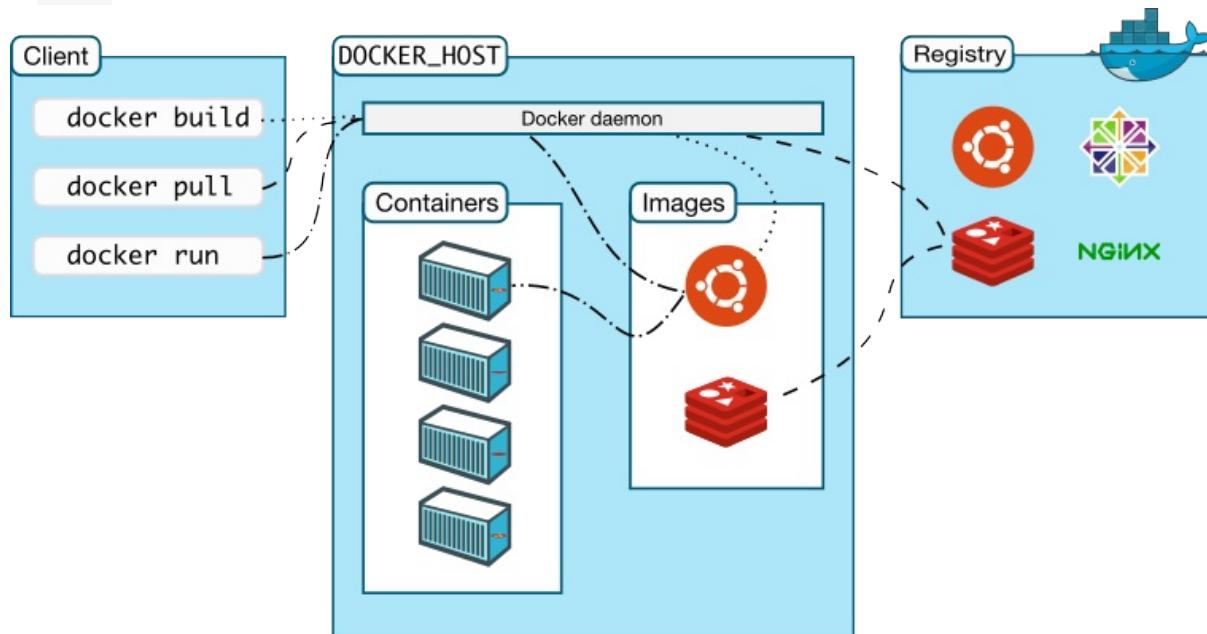
- 常驻后台进程 Dockerd
- 一个用来和 Dockerd 交互的 REST API Server

- 命令行 CLI 接口，通过和 REST API 进行交互（我们经常使用的 docker 命令）



Docker 架构

Docker 使用 C/S (客户端/服务器) 体系的架构，Docker 客户端与 Docker 守护进程通信，Docker 守护进程负责构建，运行和分发 Docker 容器。Docker 客户端和守护进程可以在同一个系统上运行，也可以将 Docker 客户端连接到远程 Docker 守护进程。Docker 客户端和守护进程使用 REST API 通过 UNIX 套接字或网络接口进行通信。



- Docker Daemon: dockerd, 用来监听 Docker API 的请求和管理 Docker 对象，比如镜像、容器、

网络和 Volume。

- Docker Client: docker, docker client 是我们和 Docker 进行交互的最主要的方式方法, 比如我们可以通过 docker run 命令来运行一个容器, 然后我们的这个 client 会把命令发送给上面的 Dockerd, 让他来做真正事情。
- Docker Registry: 用来存储 Docker 镜像的仓库, Docker Hub 是 Docker 官方提供的一个公共仓库, 而且 Docker 默认也是从 Docker Hub 上查找镜像的, 当然你也可以很方便的运行一个私有仓库, 当我们使用 docker pull 或者 docker run 命令时, 就会从我们配置的 Docker 镜像仓库中去拉取镜像, 使用 docker push 命令时, 会将我们构建的镜像推送到对应的镜像仓库中。
- Images: 镜像, 镜像是一个只读模板, 带有创建 Docker 容器的说明, 一般说的, 镜像会基于另外的一些基础镜像并加上一些额外的自定义功能。比如, 你可以构建一个基于 Centos 的镜像, 然后在这个基础镜像上面安装一个 Nginx 服务器, 这样就可以构成一个属于我们自己的镜像了。
- Containers: 容器, 容器是一个镜像的可运行的实例, 可以使用 Docker REST API 或者 CLI 来操作容器, 容器的实质是进程, 但与直接在宿主执行的进程不同, 容器进程运行于属于自己的独立的命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间, 甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里, 使用起来, 就好像是在一个独立于宿主的系统下操作一样。这种特性使得容器封装的应用比直接在宿主运行更加安全。
- 底层技术支持: Namespaces (做隔离) 、CGroups (做资源限制) 、UnionFS (镜像和容器的分层) the-underlying-technology Docker 底层架构分析

安装

直接前往[官方文档](#)选择合适的平台安装即可, 比如我们这里想要在 centos 系统上安装 Docker, 这前往地址<https://docs.docker.com/install/linux/docker-ce/centos/>根据提示安装即可。

安装依赖软件包:

```
$ sudo yum install -y yum-utils device-mapper-persistent-data lvm2
```

添加软件仓库, 我们这里使用稳定版 Docker, 执行下面命令添加 yum 仓库地址:

```
$ sudo yum-config-manager \
--add-repo \
https://download.docker.com/linux/centos/docker-ce.repo
```

然后直接安装即可:

```
$ sudo yum install docker-ce
```

如果要安装指定的版本, 可以使用 yum list 列出可用的版本:

```
$ yum list docker-ce --showduplicates | sort -r
docker-ce.x86_64          18.03.0.ce-1.el7.centos      docker-ce-stable
```

比如这里可以安装18.03.0.ce版本:

```
$ sudo yum install docker-ce-18.03.0.ce
```

要启动 Docker 也非常简单：

```
$ sudo systemctl enable docker  
$ sudo systemctl start docker
```

另外一种安装方式是可以直接下载指定的软件包直接安装即可，前往地址：https://download.docker.com/linux/centos/7/x86_64/stable/Packages/ 找到合适的 .rpm 包下载，然后安装即可：

```
$ sudo yum install /path/to/package.rpm
```

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

3. 镜像和容器的基本操作

这节课给大家讲解 Docker 镜像和容器的一些基本操作方法。

获取镜像

之前我们提到过 Docker 官方提供了一个公共的镜像仓库：Docker Hub，我们就可以从这上面获取镜像，获取镜像的命令：docker pull，格式为：

```
$ docker pull [选项] [Docker Registry 地址[:端口]/]仓库名[:标签]
```

- Docker 镜像仓库地址：地址的格式一般是 <域名/IP>[:端口号]，默认地址是 Docker Hub。
- 仓库名：这里的仓库名是两段式名称，即 <用户名>/<软件名>。对于 Docker Hub，如果不给出用户名，则默认为 library，也就是官方镜像。比如：

```
$ docker pull ubuntu:16.04
16.04: Pulling from library/ubuntu
bf5d46315322: Pull complete
9f13e0ac480c: Pull complete
e8988b5b3097: Pull complete
40af181810e7: Pull complete
e6f7c7e5c03e: Pull complete
Digest: sha256:147913621d9cdea08853f6ba9116c2e27a3ceffecf3b492983ae97c3d643fbbe
Status: Downloaded newer image for ubuntu:16.04
```

上面的命令中没有给出 Docker 镜像仓库地址，因此将会从 Docker Hub 获取镜像。而镜像名称是 ubuntu:16.04，因此将会获取官方镜像 library/ubuntu 仓库中标签为 16.04 的镜像。从下载过程中可以看到我们之前提及的分层存储的概念，镜像是由多层存储所构成。下载也是一层层的去下载，并非单一文件。下载过程中给出了每一层的 ID 的前 12 位。并且下载结束后，给出该镜像完整的 sha256 的摘要，以确保下载一致性。

运行

有了镜像后，我们就能够以这个镜像为基础启动并运行一个容器。以上面的 ubuntu:16.04 为例，如果我们打算启动里面的 bash 并且进行交互式操作的话，可以执行下面的命令。

```
$ docker run -it --rm \
    ubuntu:16.04 \
    /bin/bash

root@e7009c6ce357:/# cat /etc/os-release
NAME="Ubuntu"
VERSION="16.04.4 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.4 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
```

```
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
```

`docker run` 就是运行容器的命令，具体格式我们会在后面的课程中进行详细讲解，我们这里简要的说明一下上面用到的参数。

- `-it`: 这是两个参数，一个是 `-i`: 交互式操作，一个是 `-t` 终端。我们这里打算进入 `bash` 执行一些命令并查看返回结果，因此我们需要交互式终端。
- `--rm`: 这个参数是说容器退出后随之将其删除。默认情况下，为了排障需求，退出的容器并不会立即删除，除非手动 `docker rm`。我们这里只是随便执行个命令，看看结果，不需要排障和保留结果，因此使用 `--rm` 可以避免浪费空间。
- `ubuntu:16.04`: 这是指用 `ubuntu:16.04` 镜像为基础来启动容器。
- `bash`: 放在镜像名后的是命令，这里我们希望有个交互式 Shell，因此用的是 `bash`。

进入容器后，我们可以在 Shell 下操作，执行任何所需的命令。这里，我们执行了 `cat /etc/os-release`，这是 Linux 常用的查看当前系统版本的命令，从返回的结果可以看到容器内是 Ubuntu 16.04.4 LTS 系统。最后我们通过 `exit` 退出了这个容器。

列出镜像

```
$ docker image ls
```

列表包含了仓库名、标签、镜像 ID、创建时间以及所占用的空间。镜像 ID 则是镜像的唯一标识，一个镜像可以对应多个标签。

镜像大小

如果仔细观察，会注意到，这里标识的所占用空间和在 Docker Hub 上看到的镜像大小不同。比如，`ubuntu:16.04` 镜像大小，在这里是 127 MB，但是在 Docker Hub 显示的却是 43 MB。这是因为 Docker Hub 中显示的体积是压缩后的体积。在镜像下载和上传过程中镜像是保持着压缩状态的，因此 Docker Hub 所显示的大小是网络传输中更关心的流量大小。而 `docker image ls` 显示的是镜像下载到本地后，展开的大小，准确说，是展开后的各层所占空间的总和，因为镜像到本地后，查看空间的时候，更关心的是本地磁盘空间占用的大小。

另外一个需要注意的问题是，`docker image ls` 列表中的镜像体积总和并非是所有镜像实际硬盘消耗。由于 Docker 镜像是多层存储结构，并且可以继承、复用，因此不同镜像可能会因为使用相同的基础镜像，从而拥有共同的层。由于 Docker 使用 Union FS，相同的层只需要保存一份即可，因此实际镜像硬盘占用空间很可能要比这个列表镜像大小的总和要小的多。你可以通过以下命令来便捷的查看镜像、容器、数据卷所占用的空间。

```
$ docker system df
```

新建并启动

所需要的命令主要为 `docker run`。例如，下面的命令输出一个“Hello World”，之后终止容器。

```
$ docker run ubuntu:16.04 /bin/echo 'Hello world'
Hello world
```

这跟在本地直接执行 `/bin/echo 'hello world'` 几乎感觉不出任何区别。下面的命令则启动一个 bash 终端，允许用户进行交互。

```
$ docker run -t -i ubuntu:16.04 /bin/bash
root@af8bae53bdd3:/#
```

其中，`-t` 选项让 Docker 分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上，`-i` 则让容器的标准输入保持打开。在交互模式下，用户可以通过所创建的终端来输入命令，例如：

```
root@af8bae53bdd3:/# pwd
/
root@af8bae53bdd3:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
```

当利用 `docker run` 来创建容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载
- 利用镜像创建并启动一个容器
- 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
- 从地址池配置一个 ip 地址给容器
- 执行用户指定的应用程序
- 执行完毕后容器被终止

启动已经终止容器

可以利用 `docker container start` 命令，直接将一个已经终止的容器启动运行。

容器的核心为所执行的应用程序，所需要的资源都是应用程序运行所必需的。除此之外，并没有其它的资源。可以在伪终端中利用 `ps` 或 `top` 来查看进程信息。

```
root@ba267838cc1b:/# ps
 PID TTY      TIME CMD
  1 ?        00:00:00 bash
 11 ?        00:00:00 ps
```

可见，容器中仅运行了指定的 `bash` 应用。这种特点使得 Docker 对资源的利用率极高，是货真价实的轻量级虚拟化。

后台运行

更多的时候，需要让 Docker 在后台运行而不是直接把执行命令的结果输出在当前宿主机下。此时，可以通过添加 `-d` 参数来实现。下面举两个例子来说明一下。

如果不使用 `-d` 参数运行容器。

```
$ docker run ubuntu:16.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
hello world
hello world
hello world
hello world
```

容器会把输出的结果 (STDOUT) 打印到宿主机上面。如果使用了 `-d` 参数运行容器。

```
$ docker run -d ubuntu:16.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
77b2dc01fe0f3f1265df143181e7b9af5e05279a884f4776ee75350ea9d8017a
```

此时容器会在后台运行并不会把输出的结果 (STDOUT) 打印到宿主机上面(输出结果可以用 `docker logs` 查看)。

注：容器是否会长久运行，是和 `docker run` 指定的命令有关，和 `-d` 参数无关。

使用 `-d` 参数启动后会返回一个唯一的 id，也可以通过 `docker container ls` 命令来查看容器信息。

```
$ docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
77b2dc01fe0f ubuntu:16.04 /bin/sh -c 'while tr 2 minutes ago Up 1 minute agitated_wright
要获取容器的输出信息，可以通过 docker container logs 命令。
$ docker container logs [container ID or NAMES]
hello world
hello world
hello world
. . .
```

终止容器

可以使用 `docker container stop` 来终止一个运行中的容器。此外，当 Docker 容器中指定的应用终结时，容器也自动终止。

例如对于上一章节中只启动了一个终端的容器，用户通过 `exit` 命令或 `Ctrl+d` 来退出终端时，所创建的容器立刻终止。终止状态的容器可以用 `docker container ls -a` 命令看到。例如

```
$ docker container ls -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
ba267838cc1b ubuntu:16.04 "/bin/bash" 30 minutes ago Exited (0) About a minute ago
trusting_newton
```

处于终止状态的容器，可以通过 `docker container start` 命令来重新启动。

此外，`docker container restart` 命令会将一个运行态的容器终止，然后再重新启动它。

进入容器

在使用 `-d` 参数时，容器启动后会进入后台。某些时候需要进入容器进行操作：`exec` 命令 `-i -t` 参数。

只用 `-i` 参数时，由于没有分配伪终端，界面没有我们熟悉的 Linux 命令提示符，但命令执行结果仍然可以返回。当 `-i -t` 参数一起使用时，则可以看到我们熟悉的 Linux 命令提示符。

```
$ docker run -dit ubuntu:16.04
69d137adef7a8a689cbbc059e94da5489d3cddd240ff675c640c8d96e84fe1f6

$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
NAMES
69d137adef7a       ubuntu:16.04        "/bin/bash"        18 seconds ago   Up 17 seconds
ds                  zealous_swirles

$ docker exec -i 69d1 bash
ls
bin
boot
dev
. . .

$ docker exec -it 69d1 bash
root@69d137adef7a:/#
```

如果从这个 `stdin` 中 `exit`，不会导致容器的停止。这就是为什么推荐大家使用 `docker exec` 的原因。

更多参数说明请使用 `docker exec --help` 查看。

删除容器

可以使用 `docker container rm` 来删除一个处于终止状态的容器。例如：

```
$ docker container rm  trusting_newton
trusting_newton
```

也可用使用 `docker rm` 容器名来删除，如果要删除一个运行中的容器，可以添加 `-f` 参数。Docker 会发送 `SIGKILL` 信号给容器。

用 `docker container ls -a` (或者 `docker ps -a`) 命令可以查看所有已经创建的包括终止状态的容器，如果数量太多要一个个删除可能会很麻烦，用下面的命令可以清理掉所有处于终止状态的容器。

```
$ docker container prune
```

或者

```
$ docker ps -aq
```

删除本地镜像

如果要删除本地的镜像，可以使用`docker image rm`命令，其格式为：

```
$ docker image rm [选项] <镜像1> [<镜像2> ...]
```

或者

```
$ docker rmi 镜像名
```

或者用 ID、镜像名、摘要删除镜像 其中，<镜像> 可以是 镜像短 ID、镜像长 ID、镜像名 或者 镜像摘要。比如我们有这么一些镜像：

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ZE				
centos	latest	0584b3d2cf6d	3 weeks ago	19
6.5 MB				
redis	alpine	501ad78535f0	3 weeks ago	21
.03 MB				
docker	latest	cf693ec9b5c7	3 weeks ago	10
5.1 MB				
nginx	latest	e43d811ce2f4	5 weeks ago	18
1.5 MB				

我们可以用镜像的完整 ID，也称为 长 ID，来删除镜像。使用脚本的时候可能会用长 ID，但是人工输入就太累了，所以更多的时候是用 短 ID 来删除镜像。`docker image ls` 默认列出的就是短 ID 了，一般取前3个字符以上，只要足够区分子别的镜像就可以了。

比如这里，如果我们要删除redis:alpine镜像，可以执行：

```
$ docker image rm 501
Untagged: redis:alpine
Untagged: redis@sha256:f1ed3708f538b537eb9c2a7dd50dc90a706f7debd7e1196c9264edeeaa521a86d
Deleted: sha256:501ad78535f015d88872e13fa87a828425117e3d28075d0c117932b05bf189b7
Deleted: sha256:96167737e29ca8e9d74982ef2a0dda76ed7b430da55e321c071f0dbff8c2899b
Deleted: sha256:32770d1dcf835f192cafdf6b9263b7b597a1778a403a109e2cc2ee866f74adf23
Deleted: sha256:127227698ad74a5846ff5153475e03439d96d4b1c7f2a449c7a826ef74a2d2fa
Deleted: sha256:1333ecc582459bac54e1437335c0816bc17634e131ea0cc48daa27d32c75eab3
Deleted: sha256:4fc455b921edf9c4aea207c51ab39b10b06540c8b4825ba57b3feed1668fa7c7
```

我们也可以用镜像名，也就是 <仓库名>:<标签>，来删除镜像。

```
$ docker image rm centos
Untagged: centos:latest
Untagged: centos@sha256:b2f9d1c0ff5f87a4743104d099a3d561002ac500db1b9bfa02a783a46e0d366c
Deleted: sha256:0584b3d2cf6d235ee310cf14b54667d889887b838d3f3d3033acd70fc3c48b8a
Deleted: sha256:97ca462ad9eeae25941546209454496e1d66749d53dfa2ee32bf1faabd239d38
```

docker commit定制镜像

镜像是容器的基础，每次执行 `docker run` 的时候都会指定哪个镜像作为容器运行的基础。在之前的例子中，我们所使用的都是来自于 Docker Hub 的镜像。直接使用这些镜像是可以满足一定的需求，而当这些镜像无法直接满足需求时，我们就需要定制这些镜像。接下来的几节就将讲解如何定制镜像。

回顾一下之前我们学到的知识，镜像是多层存储，每一层是在前一层的基础上进行的修改；而容器同样也是多层存储，是在以镜像为基础层，在其基础上加一层作为容器运行时的存储层。

现在让我们以定制一个 Web 服务器为例，来讲解镜像是如何构建的。

```
$ docker run --name webserver -d -p 80:80 nginx
```

这条命令会用 `nginx` 镜像启动一个容器，命名为 `webserver`，并且映射了 80 端口，这样我们可以用浏览器去访问这个 `nginx` 服务器。

如果是在 Linux 本机运行的 Docker，或者如果使用的是 Docker for Mac、Docker for Windows，那么可以直接访问：<http://localhost>；如果使用的是 Docker Toolbox，或者是在虚拟机、云服务器上安装的 Docker，则需要将 `localhost` 换为虚拟机地址或者实际云服务器地址。

直接用浏览器访问的话，我们会看到默认的 Nginx 欢迎页面。

现在，假设我们非常不喜欢这个欢迎页面，我们希望改成欢迎 Docker 的文字，我们可以使用 `docker exec` 命令进入容器，修改其内容。

```
$ docker exec -it webserver bash
root@3729b97e8226:/# echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
root@3729b97e8226:/# exit
```

我们以交互式终端方式进入 `webserver` 容器，并执行了 `bash` 命令，也就是获得一个可操作的 Shell。然后，我们用 `<h1>Hello, Docker!</h1>` 覆盖了 `/usr/share/nginx/html/index.html` 的内容。现在我们再刷新浏览器的话，会发现内容被改变了。

我们修改了容器的文件，也就是改动了容器的存储层。我们可以通过 `docker diff` 命令看到具体的改动。

```
$ docker diff webserver
C /root
A /root/.bash_history
C /run
C /usr
C /usr/share
C /usr/share/nginx
C /usr/share/nginx/html
C /usr/share/nginx/html/index.html
C /var
C /var/cache
C /var/cache/nginx
A /var/cache/nginx/client_temp
A /var/cache/nginx/fastcgi_temp
A /var/cache/nginx/proxy_temp
```

```
A /var/cache/nginx/scgi_temp
A /var/cache/nginx/uwsgi_temp
```

现在我们定制好了变化，我们希望能将其保存下来形成镜像。

要知道，当我们运行一个容器的时候（如果不使用卷的话），我们做的任何文件修改都会被记录于容器存储层里。而 Docker 提供了一个 `docker commit` 命令，可以将容器的存储层保存下来成为镜像。换句话说，就是在原有镜像的基础上，再叠加上容器的存储层，并构成新的镜像。以后我们运行这个新镜像的时候，就会拥有原有容器最后的文件变化。

我们可以用下面的命令将容器保存为镜像：

```
$ docker commit \
--author "海马学院" \
--message "修改了默认首页" \
webserver \
nginx:v2
sha256:07e33465974800ce65751acc279adc6ed2dc5ed4e0838f8b86f0c87aa1795214
```

其中 `--author` 是指定修改的作者，而 `--message` 则是记录本次修改的内容。这点和 git 版本控制相似，不过这里这些信息可以省略留空。

我们可以在 `docker image ls` 中看到这个新定制的镜像：

```
$ docker image ls nginx
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
nginx           v2       07e334659748  9 seconds ago  181.5 MB
nginx           1.11    05a60462f8ba  12 days ago   181.5 MB
nginx           latest   e43d811ce2f4  4 weeks ago   181.5 MB
```

我们还可以用 `docker history` 具体查看镜像内的历史记录，如果比较 `nginx:latest` 的历史记录，我们会发现新增了我们刚刚提交的这一层。

```
$ docker history nginx:v2
IMAGE          CREATED      CREATED BY      SH
ZE             COMMENT
07e334659748  54 seconds ago  nginx -g daemon off;
B               修改了默认网页
e43d811ce2f4  4 weeks ago   /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon
<missing>      4 weeks ago   /bin/sh -c #(nop)  EXPOSE 443/tcp 80/tcp
<missing>      4 weeks ago   /bin/sh -c ln -sf /dev/stdout /var/log/nginx/
B               22
<missing>      4 weeks ago   /bin/sh -c apt-key adv --keyserver hkp://pgp.
.46 MB         58
<missing>      4 weeks ago   /bin/sh -c #(nop)  ENV NGINX_VERSION=1.11.5-1
<missing>      4 weeks ago   /bin/sh -c #(nop)  MAINTAINER NGINX Docker Ma
<missing>      4 weeks ago   /bin/sh -c #(nop)  CMD ["/bin/bash"]
<missing>      4 weeks ago   /bin/sh -c #(nop)  ADD file:23aa4f893e3288698c 12
```



新的镜像定制好后，我们可以来运行这个镜像。

```
$ docker run --name webserv2 -d -p 81:80 nginx:v2
```

这里我们命名为新的服务为 `webserv2`，并且映射到 81 端口。如果是 Docker for Mac/Windows 或 Linux 桌面的话，我们就可以直接访问 <http://localhost:81> 看到结果，其内容应该和之前修改后的 `webserver` 一样。

至此，我们第一次完成了定制镜像，使用的是 `docker commit` 命令，手动操作给旧的镜像添加了新的一层，形成新的镜像，对镜像多层存储应该有了更直观的感觉。

注意：`docker commit` 命令除了学习之外，还有一些特殊的应用场合，比如被入侵后保存现场等。但是，不要使用 `docker commit` 定制镜像，定制镜像应该使用 `Dockerfile` 来完成。如果你想要定制镜像请查看下一小节。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



4. Dockerfile 定制镜像

从前面一节的 `docker commit` 的学习中，我们可以了解到，镜像的定制实际上就是定制每一层所添加的配置、文件等信息，但是命令毕竟只是命令，每次定制都得去重复执行这个命令，而且还不够直观，如果我们可以把每一层修改、安装、构建、操作的命令都写入一个脚本，用这个脚本来构建、定制镜像，那么这些问题不就都可以解决了吗？对的，这个脚本就是我们说的 `Dockerfile`。

介绍

`Dockerfile` 是一个文本文件，其内包含了一条条的指令(Instruction)，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建。

还以之前定制 `nginx` 镜像为例，这次我们使用 `Dockerfile` 来定制。在一个空白目录中，建立一个文本文件，并命名为 `Dockerfile`：

```
$ mkdir mynginx
$ cd mynginx
$ touch Dockerfile
```

其内容为：

```
FROM nginx
RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
```

这个 `Dockerfile` 很简单，一共就两行。涉及到了两条指令，`FROM` 和 `RUN`。

FROM 指定基础镜像

所谓定制镜像，那一定是以一个镜像为基础，在其上进行定制。就像我们之前运行了一个 `nginx` 镜像的容器，再进行修改一样，基础镜像是必须指定的。而 `FROM` 就是指定基础镜像，因此一个 `Dockerfile` 中 `FROM` 是必备的指令，并且必须是第一条指令。

在 [Docker Store](#) 上有非常多的高质量的官方镜像，有可以直接拿来使用的服务类的镜像，如 `nginx`、`redis`、`mongo`、`mysql`、`httpd`、`php`、`tomcat` 等；也有一些方便开发、构建、运行各种语言应用的镜像，如 `node`、`openjdk`、`python`、`ruby`、`golang` 等。可以在其中寻找一个最符合我们最终目标的镜像为基础镜像进行定制。

如果没有找到对应服务的镜像，官方镜像中还提供了一些更为基础的操作系统镜像，如 `ubuntu`、`debian`、`centos`、`fedora`、`alpine` 等，这些操作系统的软件库为我们提供了更广阔的扩展空间。

除了选择现有镜像为基础镜像外，Docker 还存在一个特殊的镜像，名为 `scratch`。这个镜像是虚拟的概念，并不存在，它表示一个空白的镜像。

```
FROM scratch
...
```

如果你以 `scratch` 为镜像的话，意味着你不以任何镜像为基础，接下来所写的指令将作为镜像第一层开始存在。有的同学可能感觉很奇怪，没有任何基础镜像，我怎么去执行我的程序呢，其实对于 Linux 下静态编译的程序来说，并不需要有操作系统提供运行时支持，所需的一切库都已经在可执行文件里了，因此直接 `FROM scratch` 会让镜像体积更加小巧。使用 Go 语言 开发的应用很多会使用这种方式来制作镜像，这也是为什么有人认为 Go 是特别适合容器微服务架构的语言的原因之一。

RUN 执行命令

`RUN` 指令是用来执行命令行命令的。由于命令行的强大能力，`RUN` 指令在定制镜像时是最常用的指令之一。其格式有两种：

- shell 格式：`RUN <命令>`，就像直接在命令行中输入的命令一样。刚才写的 Dockerfile 中的 `RUN` 指令就是这种格式。

```
RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
```

- exec 格式：`RUN ["可执行文件", "参数1", "参数2"]`，这更像是函数调用中的格式。既然 `RUN` 就像 Shell 脚本一样可以执行命令，那么我们是否就可以像 Shell 脚本一样把每个命令对应一个 `RUN` 呢？比如这样：

```
FROM debian:jessie
RUN apt-get update
RUN apt-get install -y gcc libc6-dev make
RUN wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz"
RUN mkdir -p /usr/src/redis
RUN tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1
RUN make -C /usr/src/redis
RUN make -C /usr/src/redis install
```

之前说过，Dockerfile 中每一个指令都会建立一层，`RUN` 也不例外。每一个 `RUN` 的行为，就和刚才我们手工建立镜像的过程一样：新建立一层，在其上执行这些命令，执行结束后，`commit` 这一层的修改，构成新的镜像。

而上面的这种写法，创建了 7 层镜像。这是完全没有意义的，而且很多运行时不需要的东西，都被装进了镜像里，比如编译环境、更新的软件包等等。结果就是产生非常臃肿、非常多层的镜像，不仅仅增加了构建部署的时间，也很容易出错。这是很多初学 Docker 的人常犯的一个错误。

Union FS 是有最大层数限制的，比如 AUFS，曾经是最大不得超过 42 层，现在是不得超过 127 层。

上面的 Dockerfile 正确的写法应该是这样：

```
FROM debian:jessie
RUN buildDeps='gcc libc6-dev make' \
    && apt-get update \
    && apt-get install -y $buildDeps \
    && wget -O redis.tar.gz "http://download.redis.io/releases/redis-3.2.5.tar.gz" \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
```

```
&& rm -rf /var/lib/apt/lists/* \
&& rm redis.tar.gz \
&& rm -r /usr/src/redis \
&& apt-get purge -y --auto-remove $buildDeps
```

首先，之前所有的命令只有一个目的，就是编译、安装 redis 可执行文件。因此没有必要建立很多层，这只是一层的事情。因此，这里没有使用很多个 RUN 对一一对应不同的命令，而是仅仅使用一个 RUN 指令，并使用 `&&` 将各个所需命令串联起来。将之前的 7 层，简化为了 1 层。在撰写 Dockerfile 的时候，要经常提醒自己，这并不是在写 Shell 脚本，而是在定义每一层该如何构建。

并且，这里为了格式化还进行了换行。Dockerfile 支持 Shell 类的行尾添加 `\` 的命令换行方式，以及行首 `#` 进行注释的格式。良好的格式，比如换行、缩进、注释等，会让维护、排障更为容易，这是一个比较好的习惯。

此外，还可以看到这一组命令的最后添加了清理工作的命令，删除了为了编译构建所需要的软件，清理了所有下载、展开的文件，并且还清理了 apt 缓存文件。这是很重要的一步，我们之前说过，镜像是多层存储，每一层的东西并不会在下一层被删除，会一直跟随着镜像。因此镜像构建时，一定要确保每一层只添加真正需要添加的东西，任何无关的东西都应该清理掉。很多人初学 Docker 制作出了很臃肿的镜像的原因之一，就是忘记了每一层构建的最后一定要清理掉无关文件。

构建镜像

好了，让我们再回到之前定制的 nginx 镜像的 Dockerfile 来。现在我们明白了这个 Dockerfile 的内容，那么让我们来构建这个镜像吧。在 Dockerfile 文件所在目录执行：

```
$ docker build -t nginx:v3 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM nginx
--> e43d811ce2f4
Step 2 : RUN echo '<h1>Hello, Docker!</h1>' > /usr/share/nginx/html/index.html
--> Running in 9cdc27646c7b
--> 44aa4490ce2c
Removing intermediate container 9cdc27646c7b
Successfully built 44aa4490ce2c
```

从命令的输出结果中，我们可以清晰的看到镜像的构建过程。在 Step 2 中，如同我们之前所说的那样，RUN 指令启动了一个容器 `9cdc27646c7b`，执行了所要求的命令，并最后提交了这一层 `44aa4490ce2c`，随后删除了所用到的这个容器 `9cdc27646c7b`。这里我们使用了 `docker build` 命令进行镜像构建。其格式为：

```
$ docker build [选项] <上下文路径/URL/->
```

在这里我们指定了最终镜像的名称 `-t nginx:v3`，构建成功后，我们可以像之前运行 `nginx:v2` 那样来运行这个镜像，其结果会和 `nginx:v2` 一样。

镜像构建上下文（Context）

如果注意，会看到 docker build 命令最后有一个 `..`。`..` 表示当前目录，而 Dockerfile 就在当前目录，因此不少初学者以为这个路径是在指定 Dockerfile 所在路径，这么理解其实是不准确的。如果对应上面的命令格式，你可能会发现，这是在指定上下文路径。那么什么是上下文呢？

首先我们要理解 docker build 的工作原理。Docker 在运行时分为 Docker 引擎（也就是服务端守护进程）和客户端工具。Docker 的引擎提供了一组 REST API，被称为 Docker Remote API，而如 docker 命令这样的客户端工具，则是通过这组 API 与 Docker 引擎交互，从而完成各种功能。因此，虽然表面上我们好像是在本机执行各种 docker 功能，但实际上，一切都是使用的远程调用形式在服务端（Docker 引擎）完成。也因为这种 C/S 设计，让我们操作远程服务器的 Docker 引擎变得轻而易举。

当我们进行镜像构建的时候，并非所有定制都会通过 RUN 指令完成，经常会需要将一些本地文件复制进镜像，比如通过 COPY 指令、ADD 指令等。而 docker build 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种客户端/服务端的架构中，如何才能让服务端获得本地文件呢？

这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，docker build 命令得知这个路径后，会将路径下的所有内容打包，然后上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件。如果在 Dockerfile 中这么写：

```
COPY ./package.json /app/
```

这并不是要复制执行 docker build 命令所在的目录下的 package.json，也不是复制 Dockerfile 所在目录下的 package.json，而是复制上下文（context）目录下的 package.json。

因此，COPY 这类指令中的源文件的路径都是相对路径。这也是初学者经常会问的为什么 COPY `./package.json /app` 或者 COPY `/opt/xxxx /app` 无法工作的原因，因为这些路径已经超出了上下文的范围，Docker 引擎无法获得这些位置的文件。如果真的需要那些文件，应该将它们复制到上下文目录中去。

现在就可以理解刚才的命令 `docker build -t nginx:v3 .` 中的这个 `.`，实际上是在指定上下文的目录，docker build 命令会将该目录下的内容打包交给 Docker 引擎以帮助构建镜像。

如果观察 docker build 输出，我们其实已经看到了这个发送上下文的过程：

```
$ docker build -t nginx:v3 .
Sending build context to Docker daemon 2.048 kB
...
```

理解构建上下文对于镜像构建是很重要的，可以避免犯一些不应该的错误。比如有些初学者在发现 COPY `/opt/xxxx /app` 不工作后，于是干脆将 Dockerfile 放到了硬盘根目录去构建，结果发现 docker build 执行后，在发送一个几十 GB 的东西，极为缓慢而且很容易构建失败。那是因为这种做法是在让 docker build 打包整个硬盘，这显然是使用错误。

一般来说，应该会将 Dockerfile 置于一个空目录下，或者项目根目录下。如果该目录下没有所需文件，那么应该把所需文件复制一份过来。如果目录下有些东西确实不希望构建时传给 Docker 引擎，那么可以用 .gitignore 一样的语法写一个 .dockerignore，该文件是用于剔除不需要作为上下文传递给 Docker 引擎的。

那么为什么会有人误以为 . 是指定 Dockerfile 所在目录呢？这是因为在默认情况下，如果不额外指定 Dockerfile 的话，会将上下文目录下的名为 Dockerfile 的文件作为 Dockerfile。

这只是默认行为，实际上 Dockerfile 的文件名并不要求必须为 Dockerfile，而且并不要求必须位于上下文目录中，比如可以用 -f ./Dockerfile.php 参数指定某个文件作为 Dockerfile。

当然，一般大家习惯性的会使用默认的文件名 Dockerfile，以及会将其置于镜像构建上下文目录中。

迁移镜像

Docker 还提供了 docker load 和 docker save 命令，用以将镜像保存为一个 tar 文件，然后传输到另一个位置上，再加载进来。这是在没有 Docker Registry 时的做法，现在已经不推荐，镜像迁移应该直接使用 Docker Registry，无论是直接使用 Docker Hub 还是使用内网私有 Registry 都可以。

使用 docker save 命令可以将镜像保存为归档文件。比如我们希望保存这个 alpine 镜像。

```
$ docker image ls alpine
REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
alpine              latest   baa5d63471ea  5 weeks ago   4.803 MB
```

保存镜像的命令为：

```
$ docker save alpine | gzip > alpine-latest.tar.gz
```

然后我们将 alpine-latest.tar.gz 文件复制到了到了另一个机器上，可以用下面这个命令加载镜像：

```
$ docker load -i alpine-latest.tar.gz
Loaded image: alpine:latest
```

如果我们结合这两个命令以及 ssh 甚至 pv 的话，利用 Linux 强大的管道，我们可以写一个命令完成从一个机器将镜像迁移到另一个机器，并且带进度条的功能：

```
docker save <镜像名> | bzip2 | pv | ssh <用户名>@<主机名> 'cat | docker load'
```

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 12:59:25

5. 私有镜像仓库

这节课给大家讲讲私有镜像仓库的使用。

Docker Hub

目前 Docker 官方维护了一个公共仓库 Docker Hub，大部分需求都可以通过在 Docker Hub 中直接下载镜像来实现。如果你觉得拉取 Docker Hub 的镜像比较慢的话，我们可以配置一个镜像加速器：<http://docker-cn.com/>，当然国内大部分云厂商都提供了相应的加速器，简单配置即可。

注册

你可以在 <https://cloud.docker.com> 免费注册一个 Docker 账号。

登录

通过执行 `docker login` 命令交互式的输入用户名及密码来完成在命令行界面登录 Docker Hub。

注销

你可以通过 `docker logout` 退出登录。 拉取镜像

拉取镜像

你可以通过 `docker search` 命令来查找官方仓库中的镜像，并利用 `docker pull` 命令来将它下载到本地。

例如以 centos 为关键词进行搜索：

```
$ docker search centos
NAME                           DESCRIPTION
centos                         The official build of CentOS.
        465      [OK]
tianon/centos                   CentOS 5 and 6, created using rinse instead
        ...      28
blalor/centos                  Bare-bones base CentOS 6.5 image
        6      [OK]
saltstack/centos-6-minimal    DEPRECATED. Use tutum/centos:6.4 instead.
        6      [OK]
tutum/centos-6.4               DEPRECATED. Use tutum/centos:6.4 instead.
        ...      5      [OK]
```

可以看到返回了很多包含关键字的镜像，其中包括镜像名字、描述、收藏数（表示该镜像的受关注程度）、是否官方创建、是否自动创建。

官方的镜像说明是官方项目组创建和维护的， automated 资源允许用户验证镜像的来源和内容。

根据是否是官方提供，可将镜像资源分为两类。

- 一种是类似 centos 这样的镜像，被称为基础镜像或根镜像。这些基础镜像由 Docker 公司创建、验证、支持、提供。这样的镜像往往使用单个单词作为名字。
- 还有一种类型，比如 tianon/centos 镜像，它是由 Docker 的用户创建并维护的，往往带有用户名前缀。可以通过前缀 `username/` 来指定使用某个用户提供的镜像，比如 tianon 用户。

另外，在查找的时候通过 `--filter=stars=N` 参数可以指定仅显示收藏数量为 N 以上的镜像。下载官方 centos 镜像到本地。

```
$ docker pull centos
Pulling repository centos
0b443ba03958: Download complete
539c0211cd76: Download complete
511136ea3c5a: Download complete
7064731afe90: Download complete
```

推送镜像

用户也可以在登录后通过 `docker push` 命令来将自己的镜像推送到 Docker Hub。以下命令中的 `username` 请替换为你的 Docker 账号用户名。

```
$ docker tag ubuntu:17.10 username/ubuntu:17.10
$ docker image ls

REPOSITORY          TAG      IMAGE ID
ubuntu              17.10   275d79972a
username/ubuntu     17.10   275d79972a
$ docker push username/ubuntu:17.10
$ docker search username

NAME          DESCRIPTION      STARS
username/ubuntu          AUTOMATED
```

私有仓库

有时候使用 Docker Hub 这样的公共仓库可能不方便，用户可以创建一个本地仓库供私人使用。

`docker-registry` 是官方提供的工具，可以用于构建私有的镜像仓库。本文内容基于 `docker-registry` v2.x 版本。你可以通过获取官方 `registry` 镜像来运行。

```
$ docker run -d -p 5000:5000 --restart=always --name registry registry
```

这将使用官方的 `registry` 镜像来启动私有仓库。默认情况下，仓库会被创建在容器的 `/var/lib/registry` 目录下。你可以通过 `-v` 参数来将镜像文件存放在本地的指定路径。例如下面的例子将上传的镜像放到本地的 `/opt/data/registry` 目录。

```
$ docker run -d \
-p 5000:5000 \
-v /opt/data/registry:/var/lib/registry \
registry
```

在私有仓库上传、搜索、下载镜像

创建好私有仓库之后，就可以使用 `docker tag` 来标记一个镜像，然后推送它到仓库。例如私有仓库地址为 `127.0.0.1:5000`。先在本机查看已有的镜像。

```
$ docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED
ubuntu              latest   ba5877dc9bec  6 weeks ago
                     192.7 MB
```

使用 `docker tag` 将 `ubuntu:latest` 这个镜像标记为 `127.0.0.1:5000/ubuntu:latest`。格式为 `docker tag IMAGE[:TAG] [REGISTRY_HOST[:PORT]]/REPOSITORY[:TAG]`

```
$ docker tag ubuntu:latest 127.0.0.1:5000/ubuntu:latest
$ docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED
ubuntu              latest   ba5877dc9bec  6 weeks ago
                     192.7 MB
127.0.0.1:5000/ubuntu:latest    latest   ba5877dc9bec  6 weeks ago
                     192.7 MB
```

使用 `docker push` 上传标记的镜像。

```
$ docker push 127.0.0.1:5000/ubuntu:latest
The push refers to repository [127.0.0.1:5000/ubuntu]
373a30c24545: Pushed
a9148f5200b0: Pushed
cdd3de0940ab: Pushed
fc56279bbb33: Pushed
b38367233d37: Pushed
2aebd096e0e2: Pushed
latest: digest: sha256:fe4277621f10b5026266932ddf760f5a756d2facd505a94d2da12f4f52f71f5a size: 1568
```

用 `curl` 查看仓库中的镜像。

```
$ curl 127.0.0.1:5000/v2/_catalog
{"repositories":["ubuntu"]}
```

这里可以看到 {"repositories":["ubuntu"]}, 表明镜像已经被成功上传了。

先删除已有镜像，再尝试从私有仓库中下载这个镜像。

```
$ docker image rm 127.0.0.1:5000/ubuntu:latest

$ docker pull 127.0.0.1:5000/ubuntu:latest
Pulling repository 127.0.0.1:5000/ubuntu:latest
ba5877dc9bec: Download complete
511136ea3c5a: Download complete
9bad880da3d2: Download complete
25f11f5fb0cb: Download complete
ebc34468f71d: Download complete
2318d26665ef: Download complete

$ docker image ls
REPOSITORY                      TAG      IMAGE ID      CREATED
VIRTUAL SIZE
127.0.0.1:5000/ubuntu:latest    latest   ba5877dc9bec  6 weeks ago
192.7 MB
```

注意事项

如果你不想使用 127.0.0.1:5000 作为仓库地址，比如想让本网段的其他主机也能把镜像推送到私有仓库。你就得把例如 192.168.199.100:5000 这样的内网地址作为私有仓库地址，这时你会发现无法成功推送镜像。

这是因为 Docker 默认不允许非 HTTPS 方式推送镜像。我们可以通过 Docker 的配置选项来取消这个限制。

Ubuntu 14.04, Debian 7 Wheezy

对于使用 upstart 的系统而言，编辑 `/etc/default/docker` 文件，在其中的 `DOCKER_OPTS` 中增加如下内容：

```
DOCKER_OPTS="--registry-mirror=https://registry.docker-cn.com --insecure-registries=192.16
8.199.100:5000"
```

重新启动服务：

```
$ sudo service docker restart
```

Ubuntu 16.04+, Debian 8+, centos 7

对于使用 systemd 的系统，请在 `/etc/docker/daemon.json` 中写入如下内容（如果文件不存在请新建该文件）

```
{
  "registry-mirror": [
```

```
    "https://registry.docker-cn.com"
],
"insecure-registries": [
    "192.168.199.100:5000"
]
}
```

注意：该文件必须符合 json 规范，否则 Docker 将不能启动。

其他

对于 Docker for Windows、Docker for Mac 在设置中编辑 `daemon.json` 增加和上边一样的字符串即可。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

6. 数据共享与持久化

这一节介绍如何在 Docker 内部以及容器之间管理数据，在容器中管理数据主要有两种方式：

- 数据卷 (Data Volumes)
- 挂载主机目录 (Bind mounts)

数据卷

数据卷 是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

- 数据卷 可以在容器之间共享和重用
- 对 数据卷 的修改会立马生效
- 对 数据卷 的更新，不会影响镜像
- 数据卷 默认会一直存在，即使容器被删除

注意：数据卷 的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，能显示看的是挂载的数据卷。

选择 -v 还是 --mount 参数： Docker 新用户应该选择 --mount 参数，经验丰富的 Docker 使用者对 -v 或者 --volume 已经很熟悉了，但是推荐使用 --mount 参数。

创建一个数据卷：

```
$ docker volume create my-vol
```

查看所有的 数据卷：

```
$ docker volume ls
local          my-vol
```

在主机里使用以下命令可以查看指定 数据卷 的信息

```
$ docker volume inspect my-vol
[
  {
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
    "Name": "my-vol",
    "Options": {},
    "Scope": "local"
  }
]
```

启动一个挂载数据卷的容器：在用 `docker run` 命令的时候，使用 `--mount` 标记来将 数据卷 挂载到容器里。在一次 `docker run` 中可以挂载多个 数据卷。下面创建一个名为 `web` 的容器，并加载一个 数据卷 到容器的 `/webapp` 目录。

```
$ docker run -d -P \
--name web \
# -v my-vol:/weapp \
--mount source=my-vol,target=/webapp \
training/webapp \
python app.py
```

查看数据卷的具体信息：在主机里使用以下命令可以查看 `web` 容器的信息

```
$ docker inspect web
...
"Mounts": [
{
  "Type": "volume",
  "Name": "my-vol",
  "Source": "/var/lib/docker/volumes/my-vol/_data",
  "Destination": "/app",
  "Driver": "local",
  "Mode": "",
  "RW": true,
  "Propagation": ""
},
...
]
```

删除数据卷：

```
$ docker volume rm my-vol
```

数据卷 是被设计用来持久化数据的，它的生命周期独立于容器，Docker 不会在容器被删除后自动删除数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的数据卷。如果需要在删除容器的同时移除数据卷。可以在删除容器的时候使用 `docker rm -v` 这个命令。无主的数据卷可能会占据很多空间，要清理请使用以下命令

```
$ docker volume prune
```

挂载主机目录

选择 `-v` 还是 `--mount` 参数： Docker 新用户应该选择 `--mount` 参数，经验丰富的 Docker 使用者对 `-v` 或者 `--volume` 已经很熟悉了，但是推荐使用 `--mount` 参数。

挂载一个主机目录作为数据卷：使用 `--mount` 标记可以指定挂载一个本地主机的目录到容器中去。

```
$ docker run -d -P \
--name web \
# -v /src/webapp:/opt/webapp \
```

```
--mount type=bind,source=/src/webapp,target=/opt/webapp \
training/webapp \
python app.py
```

上面的命令加载主机的 /src/webapp 目录到容器的 /opt/webapp 目录。这个功能在进行测试的时候十分方便，比如用户可以放置一些程序到本地目录中，来查看容器是否正常工作。本地目录的路径必须是绝对路径，以前使用 -v 参数时如果本地目录不存在 Docker 会自动为你创建一个文件夹，现在使用 --mount 参数时如果本地目录不存在，Docker 会报错。

Docker 挂载主机目录的默认权限是 读写，用户也可以通过增加 `readonly` 指定为 只读。

```
$ docker run -d -P \
--name web \
# -v /src/webapp:/opt/webapp:ro \
--mount type=bind,source=/src/webapp,target=/opt/webapp,readonly \
training/webapp \
python app.py
```

加了 `readonly` 之后，就挂载为 只读 了。如果你在容器内 /opt/webapp 目录新建文件，会显示如下错误：

```
/opt/webapp # touch new.txt
touch: new.txt: Read-only file system
```

查看数据卷的具体信息：在主机里使用以下命令可以查看 web 容器的信息

```
$ docker inspect web
...
"Mounts": [
  {
    "Type": "bind",
    "Source": "/src/webapp",
    "Destination": "/opt/webapp",
    "Mode": "",
    "RW": true,
    "Propagation": "rprivate"
  }
],
```

挂载一个本地主机文件作为数据卷：`--mount` 标记也可以从主机挂载单个文件到容器中

```
$ docker run --rm -it \
# -v $HOME/.bash_history:/root/.bash_history \
--mount type=bind,source=$HOME/.bash_history,target=/root/.bash_history \
ubuntu:17.10 \
bash

root@2affd44b4667:/# history
1  ls
2  diskutil list
```

这样就可以记录在容器输入过的命令了。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



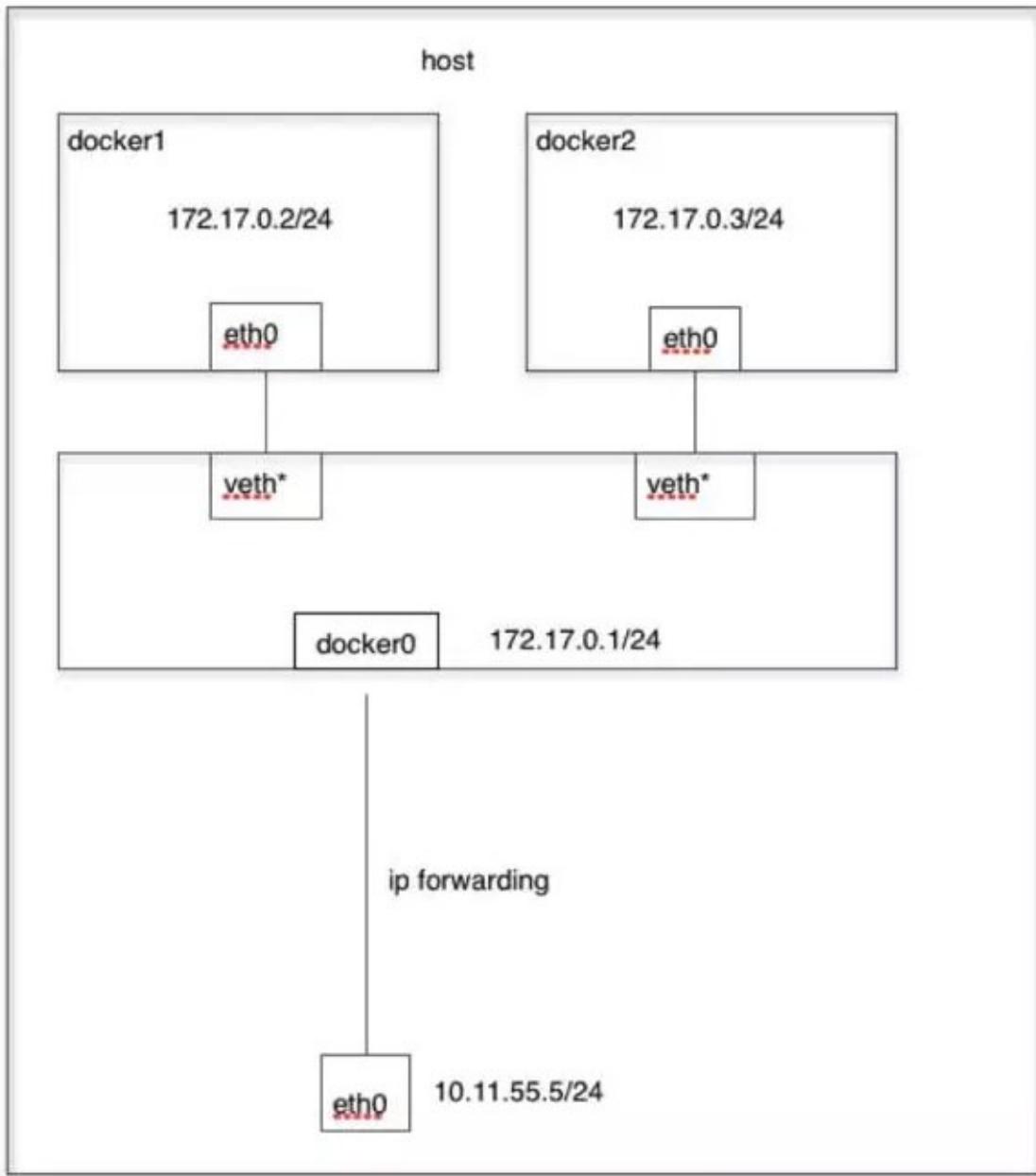
k8s技术圈二维码

7. Docker 的网络模式

Bridge模式

当 Docker 进程启动时，会在主机上创建一个名为 `docker0` 的虚拟网桥，此主机上启动的 Docker 容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。从 `docker0` 子网中分配一个 IP 给容器使用，并设置 `docker0` 的 IP 地址为容器的默认网关。在主机上创建一对虚拟网卡 `veth pair` 设备，Docker 将 `veth pair` 设备的一端放在新创建的容器中，并命名为 `eth0`（容器的网卡），另一端放在主机中，以 `vethxxx` 这样类似的名字命名，并将这个网络设备加入到 `docker0` 网桥中。可以通过 `brctl show` 命令查看。

`bridge` 模式是 `docker` 的默认网络模式，不写 `-net` 参数，就是 `bridge` 模式。使用 `docker run -p` 时，`docker` 实际是在 `iptables` 做了 `DNAT` 规则，实现端口转发功能。可以使用 `iptables -t nat -vnL` 查看。`bridge` 模式如下图所示：



演示：

```
$ docker run -tid --net=bridge --name docker_bri1 \
    ubuntu-base:v3
$ docker run -tid --net=bridge --name docker_bri2 \
    ubuntu-base:v3

$ brctl show
$ docker exec -ti docker_bri1 /bin/bash
$ ifconfig -a
$ route -n
```

如果你之前有 Docker 使用经验，你可能已经习惯了使用 `--link` 参数来使容器互联。

随着 Docker 网络的完善，强烈建议大家将容器加入自定义的 Docker 网络来连接多个容器，而不是使用 `--link` 参数。

下面先创建一个新的 Docker 网络。

```
$ docker network create -d bridge my-net
```

`-d` 参数指定 Docker 网络类型，有 `bridge overlay`。其中 `overlay` 网络类型用于 Swarm mode，在本小节中你可以忽略它。

运行一个容器并连接到新建的 `my-net` 网络

```
$ docker run -it --rm --name busybox1 --network my-net busybox sh
```

打开新的终端，再运行一个容器并加入到 `my-net` 网络

```
$ docker run -it --rm --name busybox2 --network my-net busybox sh
```

再打开一个新的终端查看容器信息

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
b47060aca56b	busybox	"sh"	11 minutes ago	Up 11 minu
tes	busybox	"sh"	16 minutes ago	Up 16 minu
8720575823ec	busybox	"sh"		
tes	busybox1			

下面通过 ping 来证明 `busybox1` 容器和 `busybox2` 容器建立了互联关系。在 `busybox1` 容器输入以下命令

```
/ # ping busybox2
PING busybox2 (172.19.0.3): 56 data bytes
64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.072 ms
64 bytes from 172.19.0.3: seq=1 ttl=64 time=0.118 ms
```

用 ping 来测试连接 `busybox2` 容器，它会解析成 172.19.0.3。同理在 `busybox2` 容器执行 ping `busybox1`，也会成功连接到。

```
/ # ping busybox1
PING busybox1 (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.064 ms
64 bytes from 172.19.0.2: seq=1 ttl=64 time=0.143 ms
```

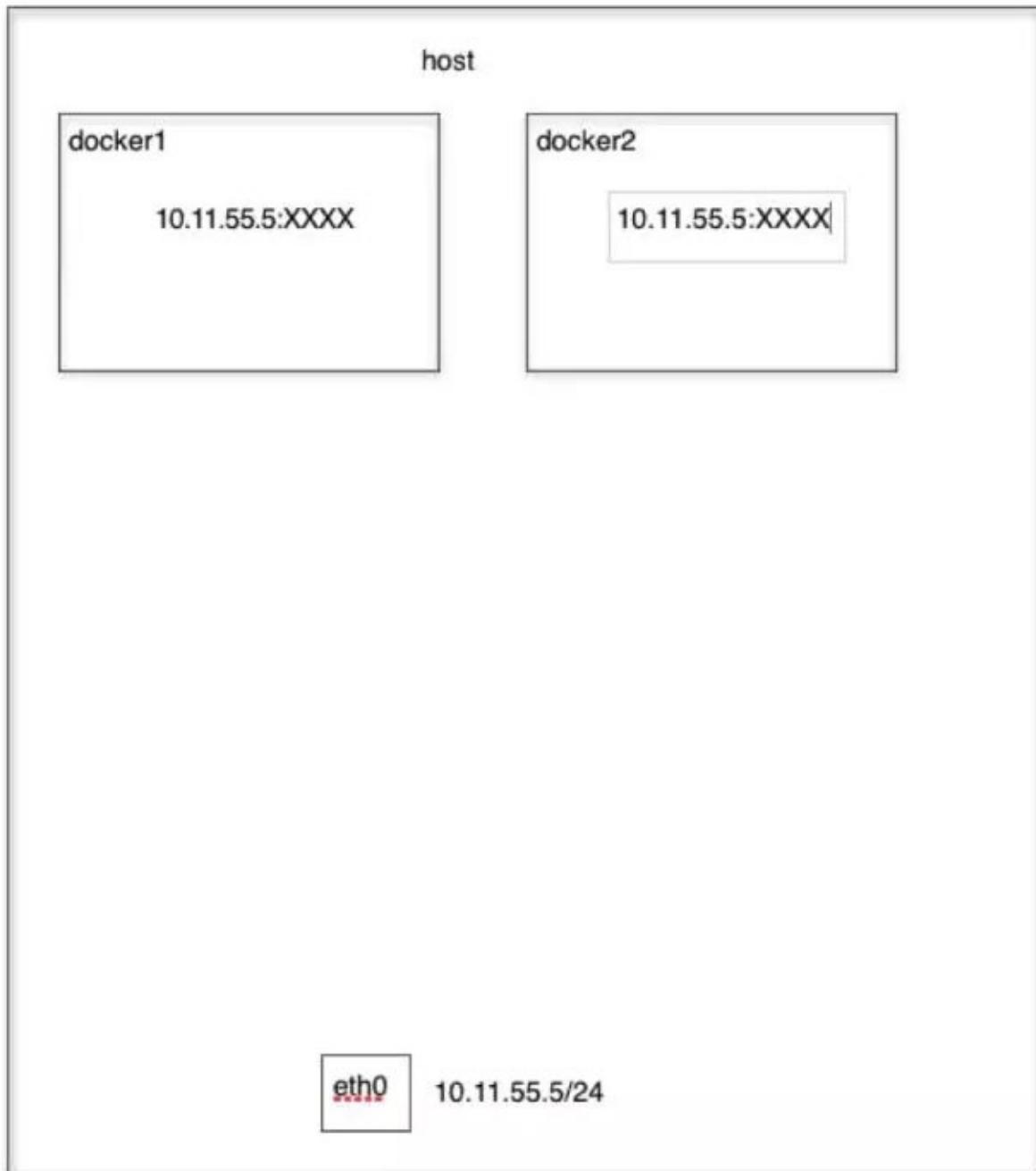
这样，`busybox1` 容器和 `busybox2` 容器建立了互联关系。

如果你有多个容器之间需要互相连接，推荐使用 `Docker Compose`。

Host 模式

如果启动容器的时候使用 host 模式，那么这个容器将不会获得一个独立的 Network Namespace，而是和宿主机共用一个 Network Namespace。容器将不会虚拟出自己的网卡，配置自己的 IP 等，而是使用宿主机的 IP 和端口。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。

Host模式如下图所示：



演示：

```
$ docker run -tid --net=host --name docker_host1 ubuntu-base:v3
$ docker run -tid --net=host --name docker_host2 ubuntu-base:v3

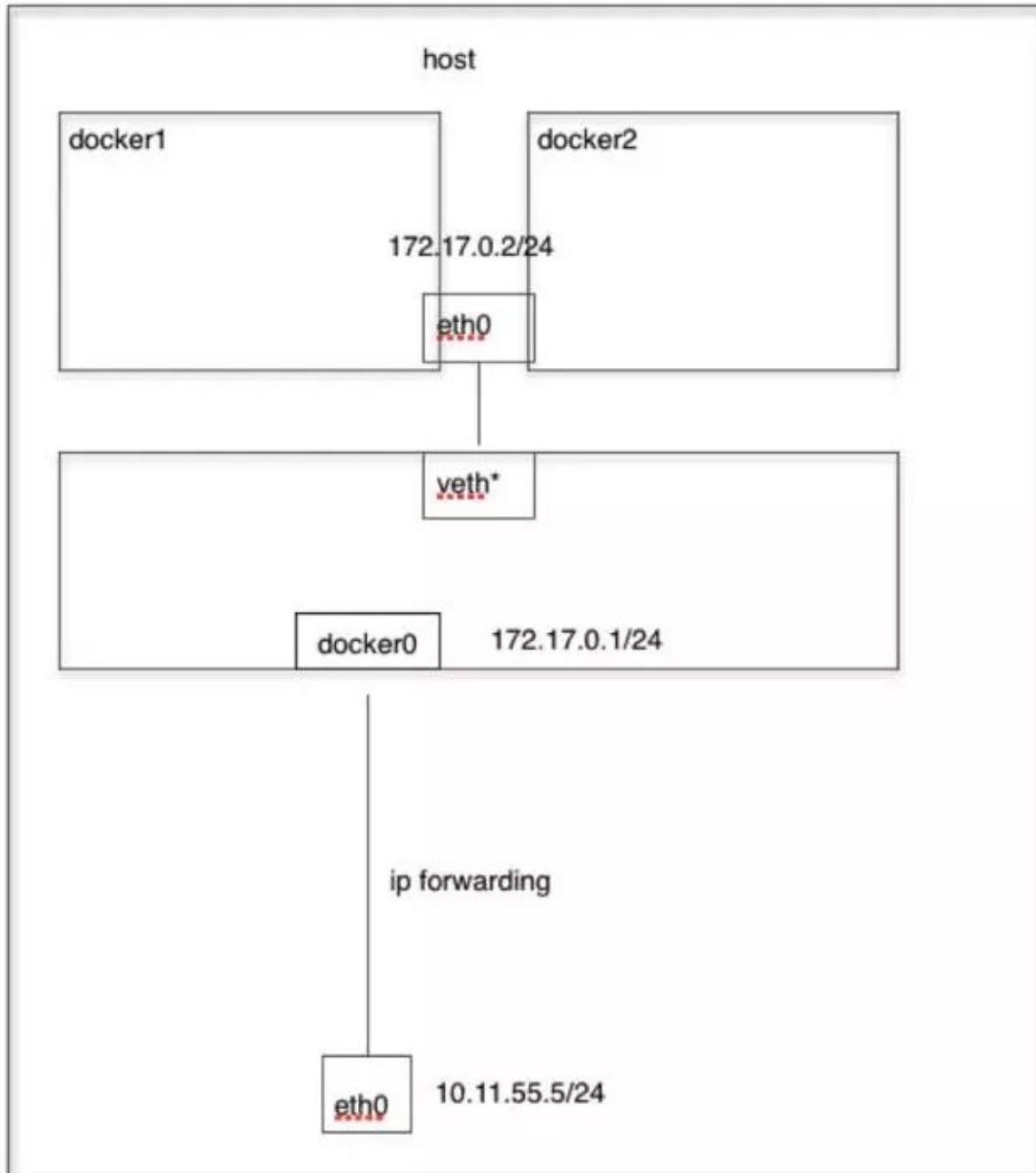
$ docker exec -ti docker_host1 /bin/bash
$ docker exec -ti docker_host1 /bin/bash

$ ifconfig -a
```

```
$ route -n
```

Container 模式

这个模式指定新创建的容器和已经存在的一个容器共享一个 Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的 IP，而是和一个指定的容器共享 IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过 lo 网卡设备通信。Container模式示意图：



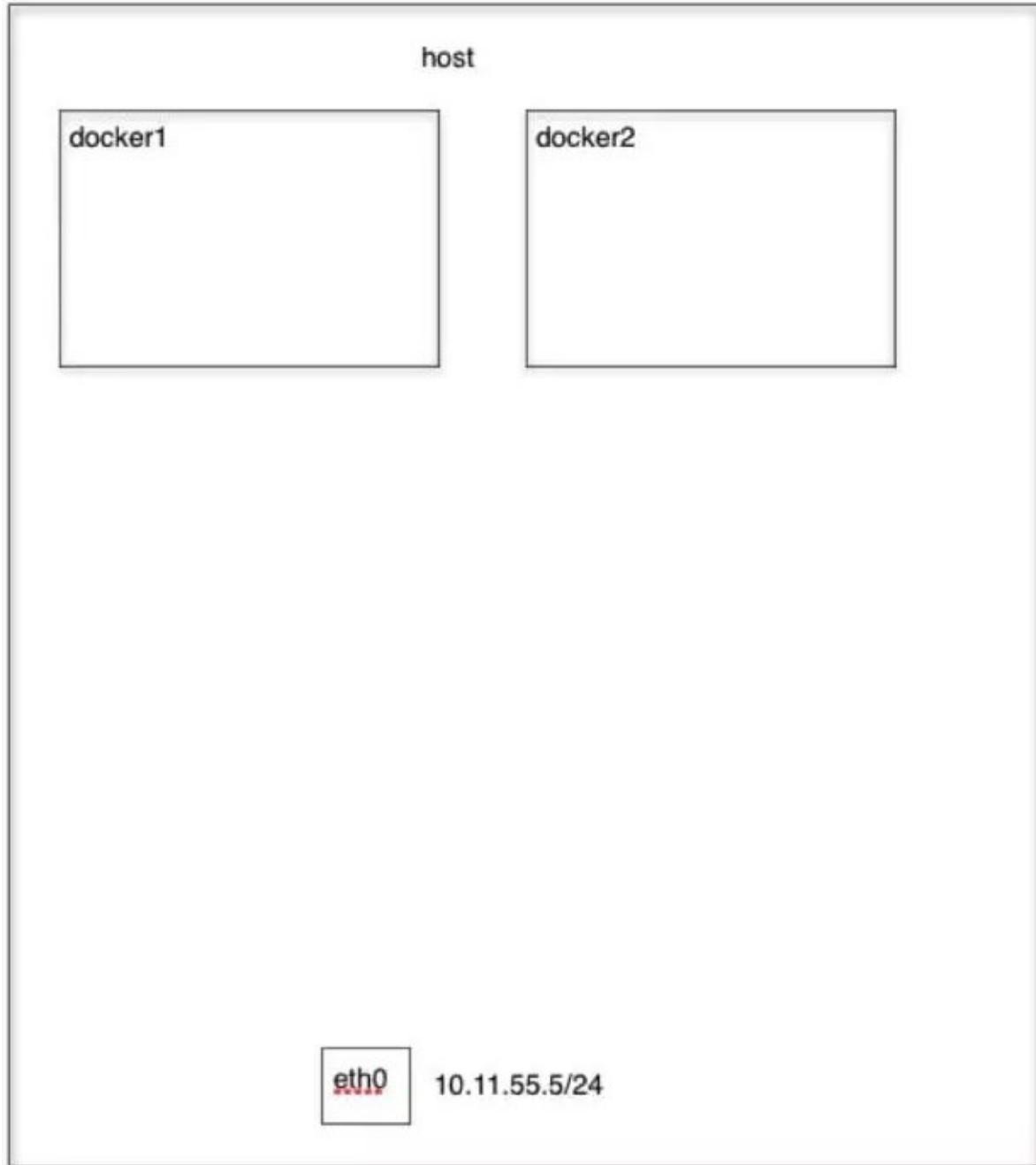
演示：

```
$ docker run -tid --net=container:docker_bri1 \
--name docker_con1 ubuntu-base:v3
```

```
$ docker exec -ti docker_con1 /bin/bash  
$ docker exec -ti docker_bri1 /bin/bash  
  
$ ifconfig -a  
$ route -n
```

None模式

使用 `none` 模式，Docker 容器拥有自己的 Network Namespace，但是，并不为 Docker 容器进行任何网络配置。也就是说，这个 Docker 容器没有网卡、IP、路由等信息。需要我们自己为 Docker 容器添加网卡、配置 IP 等。`None`模式示意图：



演示：

```
$ docker run -tid --net=none --name \
    docker_non1 ubuntu-base:v3

$ docker exec -ti docker_non1 /bin/bash

$ ifconfig -a
$ route -n
```

Docker 的跨主机通信我们这里就先暂时不讲解，我们在后面的 `Kubernetes` 课程当中会用到。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



*k8s技术圈*二维码

8. Docker Compose

介绍

Docker Compose 是 Docker 官方编排（Orchestration）项目之一，负责快速的部署分布式应用。其代码目前在<https://github.com/docker/compose>上开源。Compose 定位是「定义和运行多个 Docker 容器的应用（Defining and running multi-container Docker applications）」，其前身是开源项目 Fig。

前面我们已经学习过使用一个 `Dockerfile` 模板文件，可以很方便的定义一个单独的应用容器。然而，在日常工作中，经常会碰到需要多个容器相互配合来完成某项任务的情况。例如要实现一个 Web 项目，除了 Web 服务容器本身，往往还需要再加上后端的数据库服务容器或者缓存服务容器，甚至还包括负载均衡容器等。Compose 恰好满足了这样的需求。它允许用户通过一个单独的 `docker-compose.yml` 模板文件（YAML 格式）来定义一组相关联的应用容器为一个项目（project）。

Compose 中有两个重要的概念：

- 服务 (service)：一个应用的容器，实际上可以包括若干运行相同镜像的容器实例。
- 项目 (project)：由一组关联的应用容器组成的一个完整业务单元，在 `docker-compose.yml` 文件中定义。

Compose 的默认管理对象是项目，通过子命令对项目中的一组容器进行便捷地生命周期管理。

Compose 项目由 Python 编写，实现上调用了 Docker 服务提供的 API 来对容器进行管理。所以只要所操作的平台支持 Docker API，就可以在其上利用 Compose 来进行编排管理。

安装与卸载

Compose 支持 Linux、macOS、Windows 10 三大平台。Compose 可以通过 Python 的包管理工具 `pip` 进行安装，也可以直接下载编译好的二进制文件使用，甚至能够直接在 Docker 容器中运行。前两种方式是传统方式，适合本地环境下安装使用；最后一种方式则不破坏系统环境，更适合云计算场景。Docker for Mac、Docker for Windows 自带 `docker-compose` 二进制文件，安装 Docker 之后可以直接使用。

```
$ docker-compose --version
docker-compose version 1.17.1, build 6d101fb
```

二进制安装

在 Linux 上的安装十分简单，从官方 GitHub Release 处直接下载编译好的二进制文件即可。例如，在 Linux 64 位系统上直接下载对应的二进制包。

```
$ sudo curl -L https://github.com/docker/compose/releases/download/1.17.1/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

PIP 安装

注：x86_64 架构的 Linux 建议按照上边的方法下载二进制包进行安装，如果您计算机的架构是 ARM (例如，树莓派)，再使用 pip 安装。

这种方式是将 Compose 当作一个 Python 应用来从 pip 源中安装。执行安装命令：

```
$ sudo pip install -U docker-compose
Collecting docker-compose
  Downloading docker-compose-1.17.1.tar.gz (149kB): 149kB downloaded
  ...
Successfully installed docker-compose cached-property requests texttable websocket-client
docker-py dockerpty six enum34 backports.ssl-match-hostname ipaddress
```

bash 补全命令：

```
$ curl -L https://raw.githubusercontent.com/docker/compose/1.8.0/contrib/completion/bash/docker-compose > /etc/bash_completion.d/docker-compose
```

容器中执行

Compose 既然是一个 Python 应用，自然也可以直接用容器来执行它。

```
$ curl -L https://github.com/docker/compose/releases/download/1.8.0/run.sh > /usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
```

实际上，查看下载的 run.sh 脚本内容，如下：

```
set -e

VERSION="1.8.0"
IMAGE="docker/compose:$VERSION"

# Setup options for connecting to docker host if [ -z "$DOCKER_HOST" ]; then
DOCKER_HOST="/var/run/docker.sock"fiif [ -S "$DOCKER_HOST" ]; then
DOCKER_ADDR="-v $DOCKER_HOST:$DOCKER_HOST -e DOCKER_HOST"else
DOCKER_ADDR="-e DOCKER_HOST -e DOCKER_TLS_VERIFY -e DOCKER_CERT_PATH"fi

# Setup volume mounts for compose config and context if [ "$(pwd)" != '/' ]; then
VOLUMES="-v $(pwd):$(pwd)"fiif [ -n "$COMPOSE_FILE" ]; then
compose_dir=$(dirname $COMPOSE_FILE)fi# TODO: also check --file argument if [ -n "$compose_dir" ]; then
VOLUMES="$VOLUMES -v $compose_dir:$compose_dir"fiif [ -n "$HOME" ]; then
VOLUMES="$VOLUMES -v $HOME:$HOME -v $HOME:/root" # mount $HOME in /root to share docker.configfi
# Only allocate tty if we detect one if [ -t 1 ]; then
DOCKER_RUN_OPTIONS="-t"fiif [ -t 0 ]; then
DOCKER_RUN_OPTIONS="$DOCKER_RUN_OPTIONS -i"fi
exec docker run --rm $DOCKER_RUN_OPTIONS $DOCKER_ADDR $COMPOSE_OPTIONS $VOLUMES -w "$(pwd)"
$IMAGE "$@"
```

可以看到，它其实是下载了 `docker/compose` 镜像并运行。

卸载

如果是二进制包方式安装的，删除二进制文件即可。

```
$ sudo rm /usr/local/bin/docker-compose
```

如果是通过 pip 安装的，则执行如下命令即可删除。

```
$ sudo pip uninstall docker-compose
```

使用

下面我们用 Python 来建立一个能够记录页面访问次数的 web 网站。新建文件夹，在该目录中编写 `app.py` 文件

```
import time
import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)

if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True)
```

接着编写 `Dockerfile` 文件，内容为：

```
FROM python:3.6-alpine
ADD . /code
WORKDIR /code
RUN pip install redis flask
CMD ["python", "app.py"]
```

然后是编写 `docker-compose.yml` 文件，这个是 Compose 使用的主模板文件。

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
  redis:
    image: "redis:alpine"
```

运行 compose 项目：

```
$ docker-compose up
```

此时访问本地 5000 端口，每次刷新页面，计数就会加 1。

Compose 命令

对于 Compose 来说，大部分命令的对象既可以是项目本身，也可以指定为项目中的服务或者容器。如果没有特别的说明，命令对象将是项目，这意味着项目中所有的服务都会受到命令影响。

执行 `docker-compose [COMMAND] --help` 或者 `docker-compose help [COMMAND]` 可以查看具体某个命令的使用格式。

`docker-compose` 命令的基本的使用格式是：

```
docker-compose [-f=<arg>...] [options] [COMMAND] [ARGS...]
```

命令选项：

- `-f, --file FILE` 指定使用的 Compose 模板文件，默認為 `docker-compose.yml`，可以多次指定。
- `-p, --project-name NAME` 指定项目名称，默認将使用所在目录名称作为项目名。
- `--x-networking` 使用 Docker 的可拔插网络后端特性
- `--x-network-driver DRIVER` 指定网络后端的驱动，默認為 `bridge`
- `--verbose` 输出更多调试信息。
- `-v, --version` 打印版本并退出。

`build` 格式为 `docker-compose build [options] [SERVICE...]`。构建（重新构建）项目中的服务容器。服务容器一旦构建后，将会带上一个标记名，例如对于 `web` 项目中的一个 `db` 容器，可能是 `web_db`。可以随时在项目目录下运行 `docker-compose build` 来重新构建服务。选项包括：

- `--force-rm` 删除构建过程中的临时容器。
- `--no-cache` 构建镜像过程中不使用 cache（这将加长构建过程）。
- `--pull` 始终尝试通过 pull 来获取更新版本的镜像。

`config:` 验证 Compose 文件格式是否正确，若正确则显示配置，若格式错误显示错误原因。

down: 此命令将会停止 up 命令所启动的容器，并移除网络

exec: 进入指定的容器。

help: 获得一个命令的帮助。

images: 列出 Compose 文件中包含的镜像。

kill: 格式为 docker-compose kill [options] [SERVICE...], 通过发送 SIGKILL 信号来强制停止服务容器。支持通过 -s 参数来指定发送的信号，例如通过如下指令发送 SIGINT 信号。

```
$ docker-compose kill -s SIGINT
```

logs: 格式为 docker-compose logs [options] [SERVICE...], 查看服务容器的输出。默认情况下，docker-compose 将对不同的服务输出使用不同的颜色来区分。可以通过 --no-color 来关闭颜色。该命令在调试问题的时候十分有用。

pause: 格式为 docker-compose pause [SERVICE...], 暂停一个服务容器。

port: 格式为 docker-compose port [options] SERVICE PRIVATE_PORT, 打印某个容器端口所映射的公共端口。选项：

- --protocol=proto 指定端口协议，tcp（默认值）或者 udp。
- --index=index 如果同一服务存在多个容器，指定命令对象容器的序号（默认为 1）。

ps: 格式为 docker-compose ps [options] [SERVICE...], 列出项目中目前的所有容器。选项：

- -q 只打印容器的 ID 信息。

pull: 格式为 docker-compose pull [options] [SERVICE...], 拉取服务依赖的镜像。选项：

- --ignore-pull-failures 忽略拉取镜像过程中的错误。

push: 推送服务依赖的镜像到 Docker 镜像仓库。

restart: 格式为 docker-compose restart [options] [SERVICE...], 重启项目中的服务。选项：

- -t, --timeout TIMEOUT 指定重启前停止容器的超时（默认为 10 秒）。

rm: 格式为 docker-compose rm [options] [SERVICE...], 删除所有（停止状态的）服务容器。推荐先执行 docker-compose stop 命令来停止容器。选项：

- -f, --force 强制直接删除，包括非停止状态的容器。一般尽量不要使用该选项。
- -v 删除容器所挂载的数据卷。

run: 格式为 docker-compose run [options] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...], 在指定服务上执行一个命令。例如：

```
$ docker-compose run ubuntu ping docker.com
```

将会启动一个 ubuntu 服务容器，并执行 ping docker.com 命令。默认情况下，如果存在关联，则所有关联的服务将会自动被启动，除非这些服务已经在运行中。

该命令类似启动容器后运行指定的命令，相关卷、链接等等都将会按照配置自动创建。

给定命令将会覆盖原有的自动运行命令；不会自动创建端口，以避免冲突。

如果不希望自动启动关联的容器，可以使用 `--no-deps` 选项，例如：

```
$ docker-compose run --no-deps web python manage.py shell
```

将不会启动 `web` 容器所关联的其它容器，选项：

- `-d` 后台运行容器。
- `--name NAME` 为容器指定一个名字。
- `--entrypoint CMD` 覆盖默认的容器启动指令。
- `-e KEY=VAL` 设置环境变量值，可多次使用选项来设置多个环境变量。
- `-u, --user=""` 指定运行容器的用户名或者 uid。
- `--no-deps` 不自动启动关联的服务容器。
- `--rm` 运行命令后自动删除容器，`d` 模式下将忽略。
- `-p, --publish=[:]` 映射容器端口到本地主机。
- `--service-ports` 配置服务端口并映射到本地主机。
- `-T` 不分配伪 `tty`，意味着依赖 `tty` 的指令将无法运行。

`scale`：格式为 `docker-compose scale [options] [SERVICE=NUM...]`，设置指定服务运行的容器个数。通过 `service=num` 的参数来设置数量。例如：

```
$ docker-compose scale web=3 db=2
```

将启动 3 个容器运行 `web` 服务，2 个容器运行 `db` 服务。

一般的，当指定数目多于该服务当前实际运行容器，将新创建并启动容器；反之，将停止容器。选项：

- `-t, --timeout TIMEOUT` 停止容器时候的超时（默认为 10 秒）。

`start`：格式为 `docker-compose start [SERVICE...]`，启动已经存在的服务容器。

`stop`：格式为 `docker-compose stop [options] [SERVICE...]`，停止已经处于运行状态的容器，但不删除它。通过 `docker-compose start` 可以再次启动这些容器。选项：

- `-t, --timeout TIMEOUT` 停止容器时候的超时（默认为 10 秒）。

`top`：查看各个服务容器内运行的进程。

`unpause`：格式为 `docker-compose unpause [SERVICE...]`，恢复处于暂停状态中的服务。

`up`：格式为 `docker-compose up [options] [SERVICE...]`，该命令十分强大，它将尝试自动完成包括构建镜像，（重新）创建服务，启动服务，并关联服务相关容器的一系列操作。链接的服务都将会被自动启动，除非已经处于运行状态。可以说，大部分时候都可以直接通过该命令来启动一个项目。

默认情况，`docker-compose up` 启动的容器都在前台，控制台将会同时打印所有容器的输出信息，可以很方便进行调试。当通过 `Ctrl-C` 停止命令时，所有容器将会停止。

如果使用 `docker-compose up -d`，将会在后台启动并运行所有的容器。一般推荐生产环境下使用该选项。

默认情况，如果服务容器已经存在，`docker-compose up` 将会尝试停止容器，然后重新创建（保持使用 `volumes-from` 挂载的卷），以保证新启动的服务匹配 `docker-compose.yml` 文件的最新内容。如果用户不希望容器被停止并重新创建，可以使用 `docker-compose up --no-recreate`。这样将只会启动处于停止状态的容器，而忽略已经运行的服务。

如果用户只想重新部署某个服务，可以使用 `docker-compose up --no-deps -d <SERVICE_NAME>` 来重新创建服务并后台停止旧服务，启动新服务，并不会影响到其所依赖的服务。选项：

- `-d` 在后台运行服务容器。
 - `--no-color` 不使用颜色来区分不同的服务的控制台输出。
 - `--no-deps` 不启动服务所链接的容器。
 - `--force-recreate` 强制重新创建容器，不能与 `--no-recreate` 同时使用。`--no-recreate` 如果容器已经存在了，则不重新创建，不能与 `--force-recreate` 同时使用。`--no-build` 不自动构建缺失的服务镜像。
 - `-t, --timeout TIMEOUT` 停止容器时候的超时（默认为 10 秒）。
-

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



*k8s技术圈*二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 12:59:47

9. Docker Machine

Docker Machine 是 Docker 官方编排（Orchestration）项目之一，负责在多种平台上快速安装 Docker 环境。

Docker Machine 项目基于 Go 语言实现，目前在[Github](#)上进行维护。

Docker Machine 是 Docker 官方提供的一个工具，它可以帮助我们在远程的机器上安装 Docker，或者在虚拟机 host 上直接安装虚拟机并在虚拟机中安装 Docker。我们还可以通过 docker-machine 命令来管理这些虚拟机和 Docker。

本章将介绍 Docker Machine 的安装及使用。

安装

Docker Machine 可以在多种操作系统平台上安装，包括 Linux、macOS，以及 Windows。

macOS、Windows

Docker for Mac、Docker for Windows 自带 docker-machine 二进制包，安装之后即可使用。查看版本信息。

```
$ docker-machine -v
docker-machine version 0.13.0, build 9ba6da9
```

Linux

在 Linux 上的也安装十分简单，从[官方 GitHub Release](#)处直接下载编译好的二进制文件即可。例如，在 Linux 64 位系统上直接下载对应的二进制包。

```
$ sudo curl -L https://github.com/docker/machine/releases/download/v0.13.0/docker-machine-`uname -s`-`uname -m` > /usr/local/bin/docker-machine
$ sudo chmod +x /usr/local/bin/docker-machine
```

完成后，查看版本信息。

```
$ docker-machine -v
docker-machine version 0.13.0, build 9ba6da9
```

使用

Docker Machine 支持多种后端驱动，包括虚拟机、本地主机和云平台等。

创建本地主机实例Virtualbox 驱动

使用 virtualbox 类型的驱动，创建一台 Docker 主机，命名为 test。

```
$ docker-machine create -d virtualbox test
```

你也可以在创建时加上如下参数，来配置主机或者主机上的 Docker。

- --engine-opt dns=114.114.114.114 配置 Docker 的默认 DNS
- --engine-registry-mirror <https://registry.docker-cn.com> 配置 Docker 的仓库镜像
- --virtualbox-memory 2048 配置主机内存
- --virtualbox-cpu-count 2 配置主机 CPU

更多参数请使用 `docker-machine create --driver virtualbox --help` 命令查看。

```
$ docker-machine create -d generic \
--generic-ip-address=123.59.188.19 \
--generic-ssh-user=root \
--generic-ssh-key ~/.ssh/id_rsa \
dev
```

MacOS xhyve 驱动

xhyve 驱动 GitHub: <https://github.com/zchee/docker-machine-driver-xhyve>，`xhyve`是`MacOS`上轻量化的虚拟引擎，使用其创建的 Docker Machine 较 VirtualBox 驱动创建的运行效率要高。

```
$ brew install docker-machine-driver-xhyve
.....
$ docker-machine create \
-d xhyve \
# --xhyve-boot2docker-url ~/.docker/machine/cache/boot2docker.iso \
--engine-opt dns=114.114.114.114 \
--engine-registry-mirror https://registry.docker-cn.com \
--xhyve-memory-size 2048 \
--xhyve-rawdisk \
--xhyve-cpu-count 2 \
xhyve
```

注意：非首次创建时建议加上`--xhyve-boot2docker-url ~/.docker/machine/cache/boot2docker.iso`参数，避免每次创建时都从 GitHub 下载 ISO 镜像。

更多参数请使用 `docker-machine create --driver xhyve --help` 命令查看。

Windows 10

Windows 10 安装 Docker for Windows 之后不能再安装 VirtualBox，也就不能使用 virtualbox 驱动来创建 Docker Machine，我们可以选择使用 hyperv 驱动。

```
$ docker-machine create --driver hyperv vm
```

更多参数请使用 `docker-machine create --driver hyperv --help` 命令查看。

使用介绍

创建好主机之后，查看主机

```
$ docker-machine ls

NAME      ACTIVE     DRIVER      STATE      URL
ERRORS    test       virtualbox   Running   tcp://192.168.99.187:2376
          0.0-ce

SWARM      DOCKER
           v17.1
```

创建主机成功后，可以通过 `env` 命令来让后续操作对象都是目标主机。

```
$ docker-machine env test
```

后续根据提示在命令行输入命令之后就可以操作 `test` 主机。也可以通过 `SSH` 登录到主机。

```
$ docker-machine ssh test

docker@test:~$ docker --version
Docker version 17.10.0-ce, build f4ffd25
```

连接到主机之后你就可以在其上使用 Docker 了。

官方支持驱动

通过 `-d` 选项可以选择支持的驱动类型：

- amazonec2
- azure
- digitalocean
- exoscale
- generic
- google
- hyperv
- none
- openstack
- rackspace
- softlayer
- virtualbox
- vmwarevcloudair
- vmwarefusion
- vmwarevsphere

操作命令

- active 查看活跃的 Docker 主机
- config 输出连接的配置信息
- create 创建一个 Docker 主机
- env 显示连接到某个主机需要的环境变量
- inspect 输出主机更多信息
- ip 获取主机地址
- kill 停止某个主机
- ls 列出所有管理的主机
- provision 重新设置一个已存在的主机
- regenerate-certs 为某个主机重新生成 TLS 认证信息
- restart 重启主机
- rm 删除某台主机
- ssh SSH 到主机上执行命令
- scp 在主机之间复制文件
- mount 挂载主机目录到本地
- start 启动一个主机
- status 查看主机状态
- stop 停止一个主机
- upgrade 更新主机 Docker 版本为最新
- url 获取主机的 URL
- version 输出 docker-machine 版本信息
- help 输出帮助信息

每个命令，又带有不同的参数，可以通过如下命令来查看具体的用法：

```
$ docker-machine COMMAND --help
```

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 12:59:54

10. Docker Swarm

基本概念

`Swarm` 是使用 [SwarmKit](#) 构建的 Docker 引擎内置（原生）的集群管理和编排工具。Docker Swarm 是 Docker 官方三剑客项目之一，提供 Docker 容器集群服务，是 Docker 官方对容器云生态进行支持的核心方案。

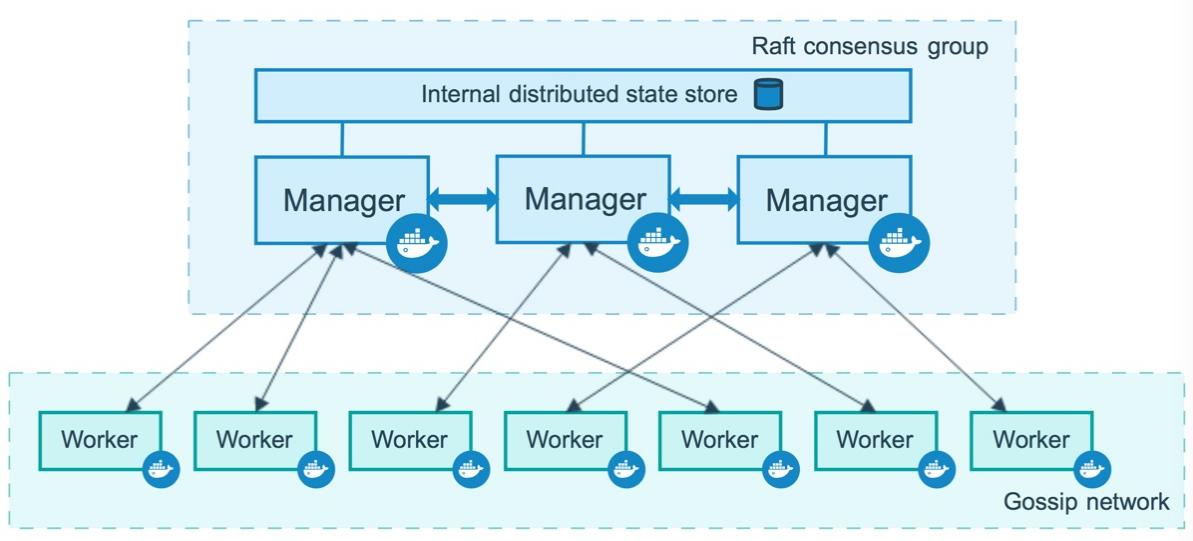
使用它，用户可以将多个 Docker 主机封装为单个大型的虚拟 Docker 主机，快速打造一套容器云平台。Swarm mode 内置 kv 存储功能，提供了众多的新特性，比如：具有容错能力的去中心化设计、内置服务发现、负载均衡、路由网格、动态伸缩、滚动更新、安全传输等。使得 Docker 原生的 Swarm 集群具备与 Mesos 、 Kubernetes 竞争的实力。使用 Swarm 集群之前需要了解以下几个概念。

节点

运行 Docker 的主机可以主动初始化一个 Swarm 集群或者加入一个已存在的 Swarm 集群，这样这个运行 Docker 的主机就成为一个 Swarm 集群的节点 (node)。节点分为 管理 (manager) 节点和工作 (worker) 节点。

管理节点用于 Swarm 集群的管理，`docker swarm` 命令基本只能在管理节点执行（节点退出集群命令 `docker swarm leave` 可以在工作节点执行）。一个 Swarm 集群可以有多个管理节点，但只有一个管理节点可以成为 `leader`，`leader` 通过 raft 协议实现。

工作节点是任务执行节点，管理节点将服务 (service) 下发至工作节点执行。管理节点默认也作为工作节点。你也可以通过配置让服务只运行在管理节点。来自 Docker 官网的这张图片形象的展示了集群中管理节点与工作节点的关系。

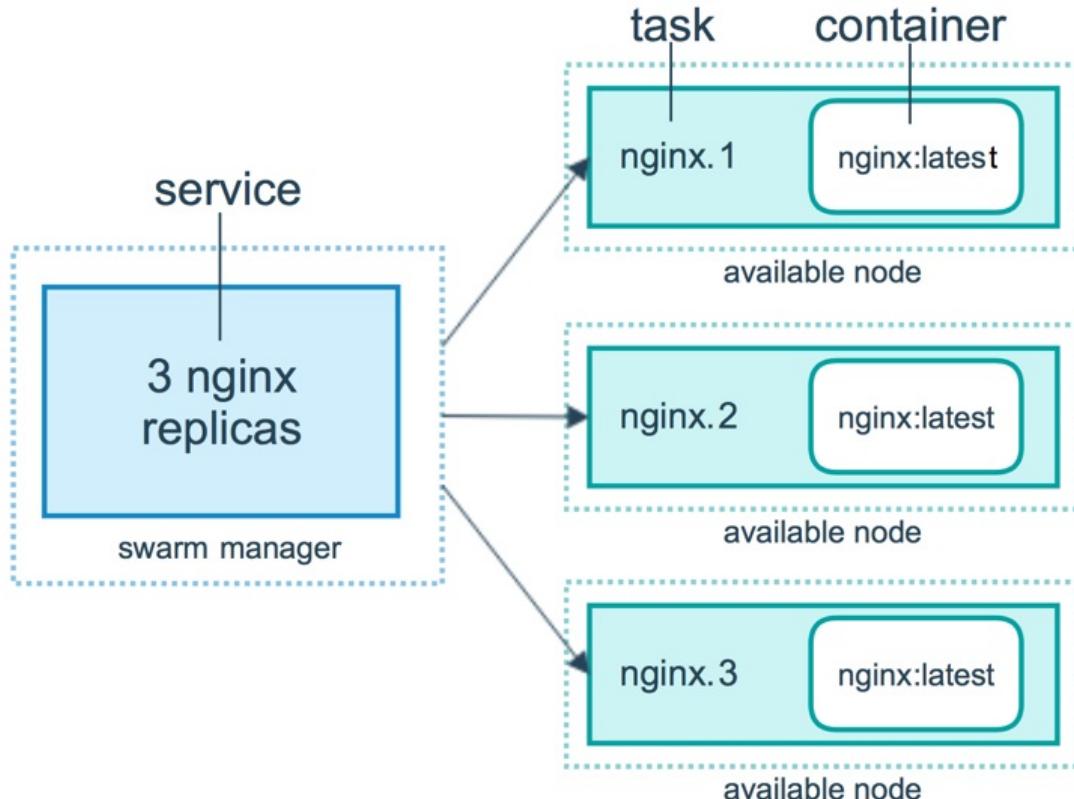


服务和任务

任务（Task）是 Swarm 中的最小的调度单位，目前来说就是一个单一的容器；服务（Services）是指一组任务的集合，服务定义了任务的属性。服务有两种模式：

- replicated services 按照一定规则在各个工作节点上运行指定个数的任务。
- global services 每个工作节点上运行一个任务

两种模式通过 `docker service create` 的 `--mode` 参数指定。来自 Docker 官网的这张图片形象的展示了容器、任务、服务的关系。



初始化集群

我们这里利用上一节的 `docker machine` 来充当集群的主机，首先先创建一个 `manager` 节点，然后在该节点上执行初始化集群命令：

```

~ ~ docker-machine create -d virtualbox manager
Running pre-create checks...
Creating machine...
(manager) Copying /Users/ych/.docker/machine/cache/boot2docker.iso to /Users/ych/.docker/machine/machines/manager/boot2docker.iso...
(manager) Creating VirtualBox VM...
(manager) Creating SSH key...
(manager) Starting the VM...
(manager) Check network to re-create if needed...
(manager) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...

```

执行 `docker swarm init` 命令的节点自动成为管理节点。

增加工作节点

管理节点初始化完成后，然后同样的用 `docker-machine` 创建工作节点，然后将其加入到管理节点之中去即可：

```
~ docker-machine create -d virtualbox worker1
Running pre-create checks...
Creating machine...
(worker1) Copying /Users/ych/.docker/machine/cache/boot2docker.iso to /Users/ych/.docker/machine/machines/worker1/boot2docker.iso...
```

我们可以看到上面的提示信息：This node joined a swarm as a worker., 表明节点已经加入到 swarm 集群之中了。

查看集群

经过上边的两步，我们已经拥有了一个最小的 Swarm 集群，包含一个管理节点和两个工作节点。

管理节点使用 docker node ls 查看集群:

~ docker node ls				
ID	HOSTNAME	STATUS	AVAILABILITY	
MANAGER STATUS	ENGINE VERSION			
3gsjpckj5ag1vvdg44fgzylow	* manager Leader 18.03.1-ce	Ready	Active	
cxmj5lr0vbwo1em9y9oang5m8	worker1 18.03.1-ce	Ready	Active	
ksruuum3uc1c265ywm4kn9a88g	worker2	Ready	Active	

```

18.03.1-ce
- ~ docker service ls
ID          NAME      MODE      REPLICAS      IMAGE
PORTS
- ~ docker service create --replicas 3 -p 80:80 --name nginx nginx:1.13.7-alpine
4k9cbna8ive87p4or9mny9kfs
overall progress: 3 out of 3 tasks
1/3: running  [=====>]
2/3: running  [=====>]
3/3: running  [=====>]
verify: Service converged

- ~ docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER
ERRORS
manager * virtualbox Running tcp://192.168.99.101:2376 v18.03.1-ce
worker1 - virtualbox Running tcp://192.168.99.102:2376 v18.03.1-ce
worker2 - virtualbox Running tcp://192.168.99.103:2376 v18.03.1-ce
- ~ docker service ls
ID          NAME      MODE      REPLICAS      IMAGE
PORTS
4k9cbna8ive8    nginx      replicated  3/3      nginx:1.13
.7-alpine  *:80->80/tcp
- ~ docker service ps nginx
ID          NAME      IMAGE      NODE      DESIRED
STATE CURRENT STATE ERROR      PORTS
r7hmzkqsri8p   nginx.1  nginx:1.13.7-alpine  worker1  Running
          Running about a minute ago
y0xgrfwmjfrj   nginx.2  nginx:1.13.7-alpine  worker2  Running
          Running about a minute ago
j8k7be8xkbg3   nginx.3  nginx:1.13.7-alpine  manager  Running
          Running about a minute ago

```

使用 `docker service logs` 来查看某个服务的日志。

```
- ~ docker service logs nginx
```

使用 `docker service rm` 来从 Swarm 集群移除某个服务:

```
- ~ docker service rm nginx
nginx
```

正如之前使用 `docker-compose.yml` 来一次配置、启动多个容器，在 Swarm 集群中也可以使用 `compose` 文件 (`docker-compose.yml`) 来配置、启动多个服务。

上一节中，我们使用 `docker service create` 一次只能部署一个服务，使用 `docker-compose.yml` 我们可以一次启动多个关联的服务。

我们以在 Swarm 集群中部署 WordPress 为例进行说明：(`docker-compose.yml`)

```

version: "3"

services:
  wordpress:
```

```

image: wordpress
ports:
  - 80:80
networks:
  - overlay
environment:
  WORDPRESS_DB_HOST: db:3306
  WORDPRESS_DB_USER: wordpress
  WORDPRESS_DB_PASSWORD: wordpress
deploy:
  mode: replicated
  replicas: 3

db:
  image: mysql
  networks:
    - overlay
  volumes:
    - db-data:/var/lib/mysql
  environment:
    MYSQL_ROOT_PASSWORD: somewordpress
    MYSQL_DATABASE: wordpress
    MYSQL_USER: wordpress
    MYSQL_PASSWORD: wordpress
  deploy:
    placement:
      constraints: [node.role == manager]

visualizer:
  image: dockersamples/visualizer.stable
  ports:
    - "8080:8080"
  stop_grace_period: 1m30s
  volumes:
    - "/var/run/docker.sock:/var/run/docker.sock"
  deploy:
    placement:
      constraints: [node.role == manager]

volumes:
  db-data:
networks:
  overlay:

```

其中constraints: [node.role == manager]是调度策略，文档地址：<https://docs.docker.com/swarm/scheduler/filter/>

在 Swarm 集群管理节点新建该文件，其中的 visualizer 服务提供一个可视化页面，我们可以从浏览器中很直观的查看集群中各个服务的运行节点。

在 Swarm 集群中使用 docker-compose.yml 我们用 docker stack 命令，下面我们将对该命令进行详细讲解。

部署服务

部署服务使用 docker stack deploy，其中 -c 参数指定 compose 文件名。

```
$ docker stack deploy -c docker-compose.yml wordpress
```

查看服务

```
$ docker stack ls
NAME      SERVICES
wordpress      3
```

移除服务

要移除服务，使用 `docker stack down`：

```
$ docker stack down wordpress
Removing service wordpress_db
Removing service wordpress_visualizer
Removing service wordpress_wordpress
Removing network wordpress_overlay
Removing network wordpress_default
```

该命令不会移除服务所使用的 数据卷，如果你想移除数据卷请使用 `docker volume rm`。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:00:01

11. 图形化管理和监控

下面我们介绍几个可以用图形化的方式来管理 Docker 的工具。

Shipyard: <https://github.com/shipyard/shipyard> (已停止维护)

Portainer

Portainer (基于 Go) 是一个轻量级的管理界面，可让您轻松管理 Docker 主机或 Swarm 集群。

Portainer 的使用意图是简单部署。它包含可以在任何 Docker 引擎上运行的单个容器 (Docker for Linux 和 Docker for Windows)。

Portainer 允许您管理 Docker 容器、image、volume、network 等。它与独立的 Docker 引擎和 Docker Swarm 兼容。

Docker 命令安装：

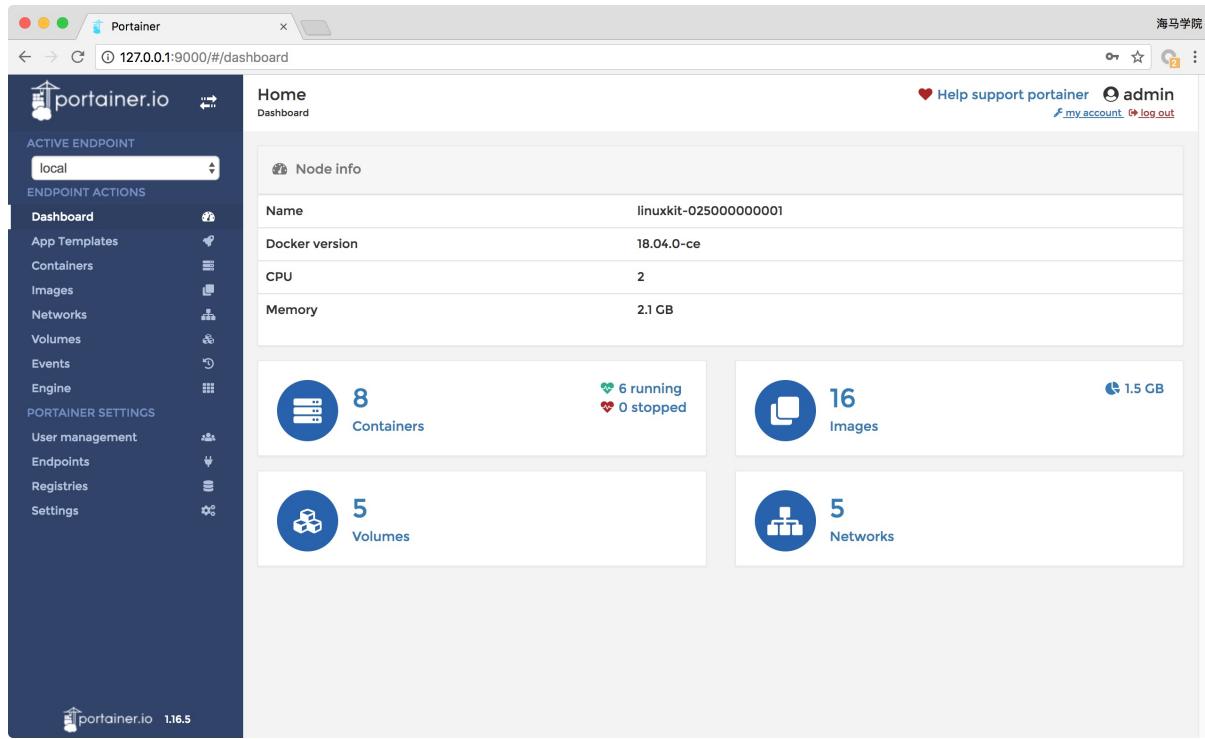
```
$ docker volume create portainer_data
$ docker run -d -p 9000:9000 -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer
```

Swarm 集群部署：

```
$ docker volume create portainer_data
$ docker service create \
--name portainer \
--publish 9000:9000 \
--replicas=1 \
--constraint 'node.role == manager' \
--mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
--mount type=volume,src=portainer_data,dst=/data \
portainer/portainer \
-H unix:///var/run/docker.sock
```

Docker Compose 部署：

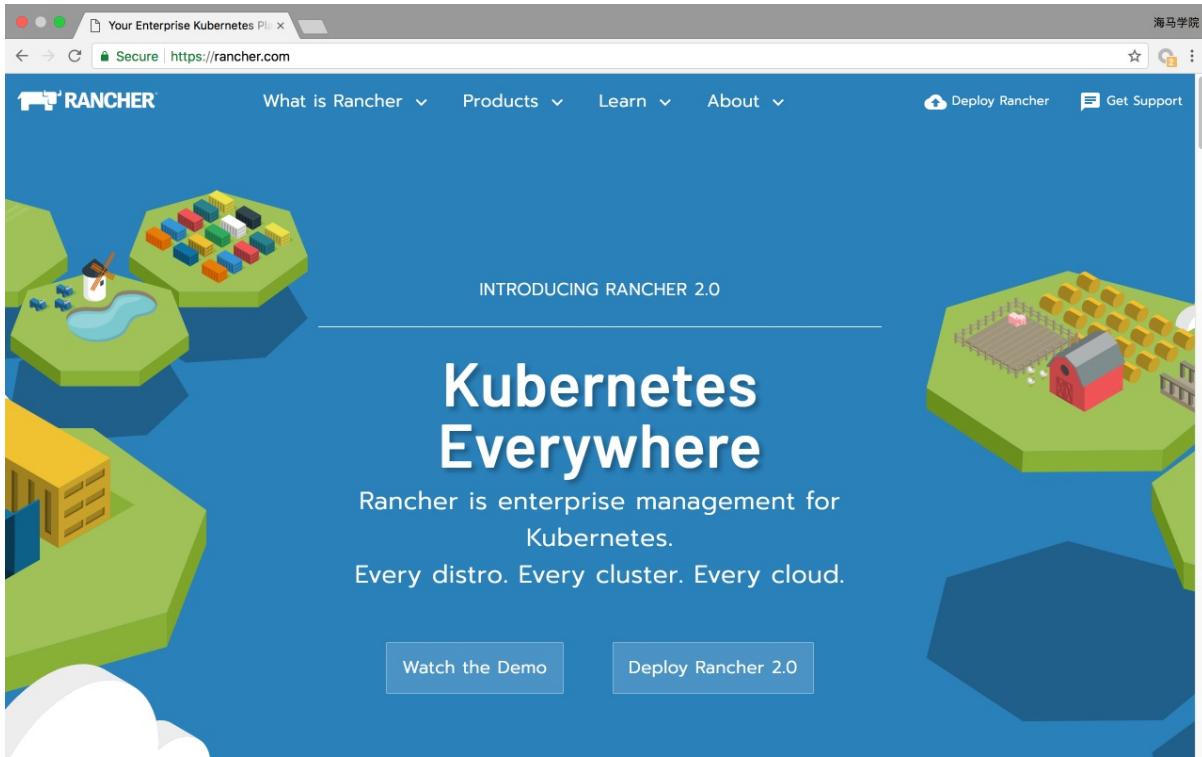
```
version: '2'
services:
  portainer:
    image: portainer/portainer
    command: -H unix:///var/run/docker.sock
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - portainer_data:/data
volumes:
  portainer_data:
```



portainer

Rancher

Rancher 是一个开源的企业级容器管理平台。通过 Rancher，企业不必自己使用一系列的开源软件去从头搭建容器服务平台。Rancher 提供了在生产环境中使用管理 Docker 和 Kubernetes 的全栈化容器部署与管理平台。



在后面学习 kubernetes 的课程的时候会给大家演示，用于我们快速搭建一个可运行 kubernetes 集群环境，非常方便。

cAdvisor

cAdvisor 是 Google 开发的容器监控工具，我们来看看 cAdvisor 有什么能耐。

- 监控 Docker Host cAdvisor 会显示当前 host 的资源使用情况，包括 CPU、内存、网络、文件系统等。
- 监控容器 点击 Docker Containers 链接，显示容器列表。点击某个容器，比如 sysdig，进入该容器的监控页面。

以上就是 cAdvisor 的主要功能，总结起来主要两点：

- 展示 Host 和容器两个层次的监控数据。
- 展示历史变化数据。

由于 cAdvisor 提供的操作界面略显简陋，而且需要在不同页面之间跳转，并且只能监控一个 host，这不免会让人质疑它的实用性。但 cAdvisor 的一个亮点是它可以将监控到的数据导出给第三方工具，由这些工具进一步加工处理。

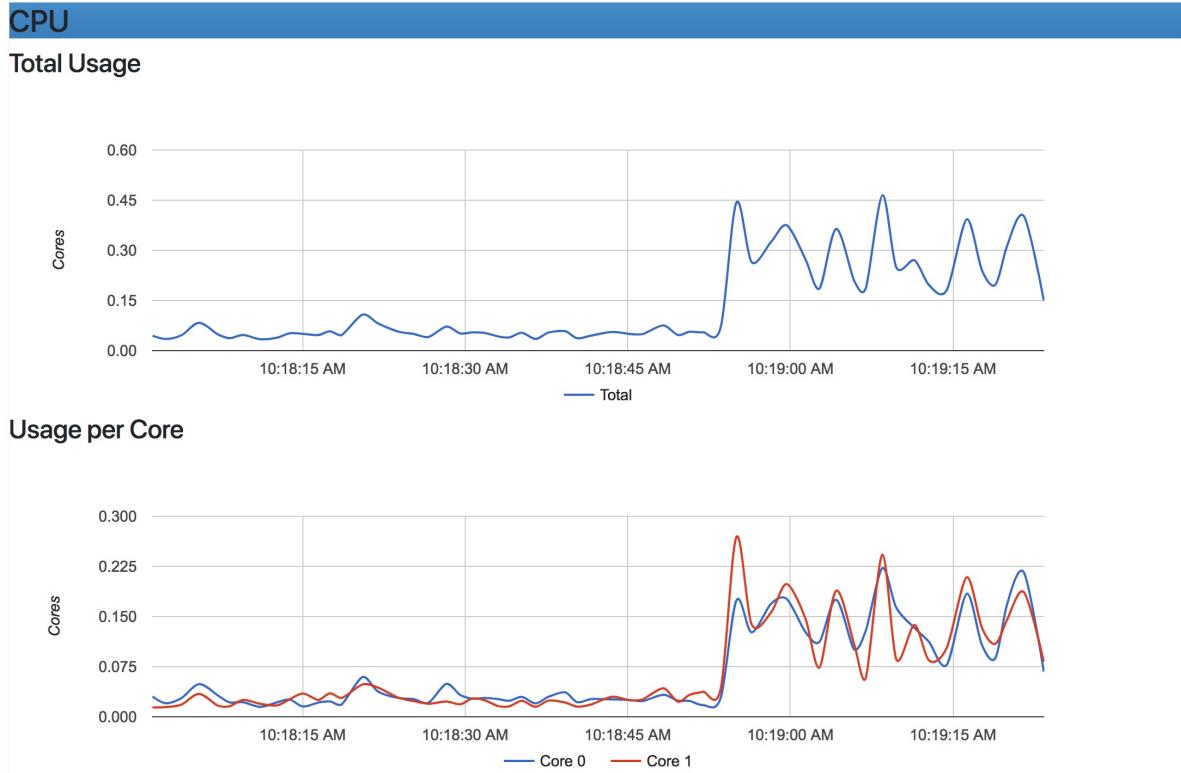
我们可以把 cAdvisor 定位为一个监控数据收集器，收集和导出数据是它的强项，而非展示数据。

cAdvisor 支持很多第三方工具，其中就包括后面我们重点要学习的 Prometheus。

```
$ docker run \
--volume=/:/rootfs:ro \
--volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \
```

```
--volume=/var/lib/docker/:/var/lib/docker:ro \
--volume=/dev/disk/:/dev/disk:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor:latest
```

通过访问地址：<http://127.0.0.1:8080/containers/> 可以查看所有容器信息：



cAdvisor 通过该 REST API 暴露监控数据，格式如下：

```
http://<hostname>:<port>/api/<version>/<request>
```

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:00:11

12. Docker 的多阶段构建

Docker 的口号是 Build,Ship, and Run Any App, Anywhere, 在我们使用 Docker 的大部分时候，的确能感觉到其优越性，但是往往在我们 Build 一个应用的时候，是将我们的源代码也构建进去的，这对于类似于 golang 这样的编译型语言肯定是不行的，因为实际运行的时候我只需要把最终构建的二进制包给你就行，把源码也一起打包在镜像中，需要承担很多风险，即使是脚本语言，在构建的时候也可能需要使用到一些上线的工具，这样无疑也增大了我们的镜像体积。

示例

比如我们现在有一个最简单的 golang 服务，需要构建一个最小的 Docker 镜像，源码如下：

```
package main
import (
    "github.com/gin-gonic/gin"
    "net/http"
)
func main() {
    router := gin.Default()
    router.GET("/ping", func(c *gin.Context) {
        c.String(http.StatusOK, "PONG")
    })
    router.Run(":8080")
}
```

解决方案

我们最终的目的都是将最终的可执行文件放到一个最小的镜像(比如 alpine)中去执行，怎样得到最终的编译好的文件呢？基于 Docker 的指导思想，我们需要在一个标准的容器中编译，比如在一个 Ubuntu 镜像中先安装编译的环境，然后编译，最后也在该容器中执行即可。

但是如果我们要想把编译后的文件放置到 alpine 镜像中执行呢？我们就得通过上面的 Ubuntu 镜像将编译完成的文件通过 volume 挂载到我们的主机上，然后我们再将这个文件挂载到 alpine 镜像中去。

这种解决方案理论上肯定可行的，但是这样的话在构建镜像的时候就得定义两步了，第一步是先用一个通用的镜像编译镜像，第二步是将编译后的文件复制到 alpine 镜像中执行，而且通用镜像编译后的文件在 alpine 镜像中不一定能执行。

定义编译阶段的 Dockerfile：(保存为Dockerfile.build)

```
FROM golang
WORKDIR /go/src/app
ADD . /go/src/app
RUN go get -u -v github.com/kardianos/govendor
RUN govendor sync
RUN GOOS=linux GOARCH=386 go build -v -o /go/src/app/app-server
```

定义 `alpine` 镜像: (保存为Dockerfile.old)

```
FROM alpine:latest
RUN apk add -U tzdata
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
WORKDIR /root/
COPY app-server .
CMD [ "./app-server" ]
```

根据我们的执行步骤, 我们还可以简单定义成一个脚本: (保存为build.sh)

```
#!/bin/sh
echo Building cnych/docker-multi-stage-demo:build

docker build -t cnych/docker-multi-stage-demo:build . -f Dockerfile.build

docker create --name extract cnych/docker-multi-stage-demo:build
docker cp extract:/go/src/app/app-server ./app-server
docker rm -f extract

echo Building cnych/docker-multi-stage-demo:old

docker build --no-cache -t cnych/docker-multi-stage-demo:old . -f Dockerfile.old
rm ./app-server
```

当我们执行完上面的构建脚本后, 就实现了我们的目标。

多阶段构建

有没有一种更加简单的方式来实现上面的镜像构建过程呢? Docker 17.05版本以后, 官方就提供了一个新的特性: `Multi-stage builds` (多阶段构建)。使用多阶段构建, 你可以在一个 `Dockerfile` 中使用多个 `FROM` 语句。每个 `FROM` 指令都可以使用不同的基础镜像, 并表示开始一个新的构建阶段。你可以很方便的将一个阶段的文件复制到另外一个阶段, 在最终的镜像中保留下你需要的内容即可。

我们可以调整前面一节的 `Dockerfile` 来使用多阶段构建: (保存为Dockerfile)

```
FROM golang AS build-env
ADD . /go/src/app
WORKDIR /go/src/app
RUN go get -u -v github.com/kardianos/govendor
RUN govendor sync
RUN GOOS=linux GOARCH=386 go build -v -o /go/src/app/app-server

FROM alpine
RUN apk add -U tzdata
RUN ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
COPY --from=build-env /go/src/app/app-server /usr/local/bin/app-server
EXPOSE 8080
CMD [ "app-server" ]
```

现在我们只需要一个 `Dockerfile` 文件即可, 也不需要拆分构建脚本了, 只需要执行 `build` 命令即可:

```
$ docker build -t cnych/docker-multi-stage-demo:latest .
```

默认情况下，构建阶段是没有命令的，我们可以通过它们的索引来引用它们，第一个 FROM 指令从 0 开始，我们也可以用 AS 指令为阶段命令，比如我们这里的将第一阶段命名为 build-env，然后在其他阶段需要引用的时候使用 --from=build-env 参数即可。

最后我们简单的运行下该容器测试：

```
$ docker run --rm -p 8080:8080 cnych/docker-multi-stage-demo:latest
```

运行成功后，我们可以在浏览器中打开 <http://127.0.0.1:8080/ping> 地址，可以看到PONG返回。

现在我们就把两个镜像的文件最终合并到一个镜像里面了。

文章中涉及到代码可以前往 [github](https://github.com/cnych/docker-multi-stage-demo) 查看：<https://github.com/cnych/docker-multi-stage-demo>

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

13. Dockerfile 最佳实践

Docker 官方关于 Dockerfile 最佳实践原文链接地址：https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Docker 可以通过从 Dockerfile 包含所有命令的文本文件中读取指令自动构建镜像，以便构建给定镜像。

Dockerfiles 使用特定的格式并使用一组特定的指令。您可以在[Dockerfile Reference](#)页面上了解基础知识。如果你是新手写作 Dockerfile，你应该从那里开始。

本文档介绍了由 Docker, Inc. 和 Docker 社区推荐的用于构建高效镜像的最佳实践和方法。要查看许多实践和建议，请查看[Dockerfile for buildpack-deps](#)。

一般准则和建议

容器应该是短暂的

通过 Dockerfile 构建的镜像所启动的容器应该尽可能短暂（生命周期短）。「短暂」意味着可以停止和销毁容器，并且创建一个新容器并部署好所需的设置和配置工作量应该是极小的。我们可以查看下[12 Factor\(12要素\)应用程序方法](#)的进程部分，可以让我们理解这种无状态方式运行容器的动机。

建立上下文

当你发出一个 `docker build` 命令时，当前的工作目录被称为构建上下文。默认情况下，Dockerfile 就位于该路径下，当然您也可以使用`-f`参数来指定不同的位置。无论 Dockerfile 在什么地方，当前目录中的所有文件内容都将作为构建上下文发送到 Docker 守护进程中去。

下面是一个构建上下文的示例，为构建上下文创建一个目录并 `cd` 放入其中。将“hello”写入一个文本文件 `hello`，然后并创建一个 Dockerfile 并运行 `cat`。从构建上下文 `(.)` 中构建图像：

```
mkdir myproject && cd myproject
echo "hello" > hello
echo -e "FROM busybox\nCOPY /hello /RUN cat /hello" > Dockerfile
docker build -t helloapp:v1 .
```

现在移动 Dockerfile 和 `hello` 到不同的目录，并建立了图像的第二个版本（不依赖于缓存中的最后一个版本）。使用 `-f` 指向 Dockerfile 并指定构建上下文的目录：

```
mkdir -p dockerfiles context
mv Dockerfile dockerfiles && mv hello context
docker build --no-cache -t helloapp:v2 -f dockerfiles/Dockerfile context
```

在构建的时候包含不需要的文件会导致更大的构建上下文和更大的镜像大小。这会增加构建时间，拉取和推送镜像的时间以及容器的运行时间大小。要查看您的构建环境有多大，请在构建您的系统时查找这样的消息

```
Dockerfile:  
Sending build context to Docker daemon 187.8MB
```

使用 `.dockerignore` 文件

使用 Dockerfile 构建镜像时最好是将 Dockerfile 放置在一个新建的空目录下。然后将构建镜像所需要的文件添加到该目录中。为了提高构建镜像的效率，你可以在目录下新建一个 `.dockerignore` 文件来指定要忽略的文件和目录。`.dockerignore` 文件的排除模式语法和 Git 的 `.gitignore` 文件相似。

使用多阶段构建

在 Docker 17.05 以上版本中，你可以使用 多阶段构建 来减少所构建镜像的大小。上一节课我们已经重点讲解过了。

避免安装不必要的包

为了降低复杂性、减少依赖、减小文件大小和构建时间，应该避免安装额外的或者不必要的软件包。例如，不要在数据库镜像中包含一个文本编辑器。

一个容器只专注做一件事情

应该保证在一个容器中只运行一个进程。将多个应用解耦到不同容器中，保证了容器的横向扩展和复用。例如一个 web 应用程序可能包含三个独立的容器：web应用、数据库、缓存，每个容器都是独立的镜像，分开运行。但这并不是说一个容器就只跑一个进程，因为有的程序可能会自行产生其他进程，比如 Celery 就可以有很多个工作进程。虽然“每个容器跑一个进程”是一条很好的法则，但这并不是一条硬性的规定。我们主要是希望一个容器只关注意见事情，尽量保持干净和模块化。

如果容器互相依赖，你可以使用[Docker 容器网络](#)来把这些容器连接起来，我们前面已经跟大家讲解过 Docker 的容器网络模式了。

最小化镜像层数

在 Docker 17.05 甚至更早 1.10 之前，尽量减少镜像层数是非常重要的，不过现在的版本已经有了一定的改善了：

- 在 1.10 以后，只有 RUN、COPY 和 ADD 指令会创建层，其他指令会创建临时的中间镜像，但是不会直接增加构建的镜像大小了。
- 上节课我们也讲解到了 17.05 版本以后增加了多阶段构建的支持，允许我们把需要的数据直接复制到最终的镜像中，这就允许我们在中间阶段包含一些工具或者调试信息了，而且不会增加最终的镜像大小。

当然减少 RUN、COPY、ADD 的指令仍然是很有必要的，但是我们也需要在 Dockerfile 可读性（也包括长期的可维护性）和减少层数之间做一个平衡。

对多行参数排序

只要有可能，就将多行参数按字母顺序排序（比如要安装多个包时）。这可以帮助你避免重复包含同一个包，更新包列表时也更容易，也更容易阅读和审查。建议在反斜杠符号 \ 之前添加一个空格，可以增加可读性。下面是来自 buildpack-deps 镜像的例子：

```
RUN apt-get update && apt-get install -y \
    bzr \
    cvs \
    git \
    mercurial \
    subversion
```

构建缓存

在镜像的构建过程中，Docker 根据 Dockerfile 指定的顺序执行每个指令。在执行每条指令之前，Docker 都会在缓存中查找是否已经存在可重用的镜像，如果有就使用现存的镜像，不再重复创建。当然如果你不想在构建过程中使用缓存，你可以在 docker build 命令中使用 --no-cache=true 选项。

如果你想在构建的过程中使用了缓存，那么了解什么时候可以什么时候无法找到匹配的镜像就很重要了，Docker 中缓存遵循的基本规则如下：

- 从一个基础镜像开始（FROM 指令指定），下一条指令将和该基础镜像的所有子镜像进行匹配，检查这些子镜像被创建时使用的指令是否和被检查的指令完全一样。如果不是，则缓存失效。
- 在大多数情况下，只需要简单地对比 Dockerfile 中的指令和子镜像。然而，有些指令需要更多的检查和解释。
- 对于 ADD 和 COPY 指令，镜像中对应文件的内容也会被检查，每个文件都会计算出一个校验值。这些文件的修改时间和最后访问时间不会被纳入校验的范围。在缓存的查找过程中，会将这些校验值和已存在镜像中的文件校验值进行对比。如果文件有任何改变，比如内容和元数据，则缓存失效。
- 除了 ADD 和 COPY 指令，缓存匹配过程不会查看临时容器中的文件来决定缓存是否匹配。例如，当执行完 RUN apt-get -y update 指令后，容器中一些文件被更新，但 Docker 不会检查这些文件。这种情况下，只有指令字符串本身被用来匹配缓存。
- 一旦缓存失效，所有后续的 Dockerfile 指令都将产生新的镜像，缓存不会被使用。

Dockerfile 指令

下面是一些常用的 Dockerfile 指令，我们也分别来总结下，根据上面的建议和下面这些指令的合理使用，可以帮助我们编写高效且易维护的 Dockerfile 文件。

FROM

尽可能使用当前官方仓库作为你构建镜像的基础。推荐使用Alpine镜像，因为它被严格控制并保持最小尺寸（目前小于 5 MB），但它仍然是一个完整的发行版。

LABEL

你可以给镜像添加标签来帮助组织镜像、记录许可信息、辅助自动化构建等。每个标签一行，由 `LABEL` 开头加上一个或多个标签对。

下面的示例展示了各种不同的可能格式。`#` 开头的行是注释内容。

注意：如果你的字符串包含空格，那么它必须被引用或者空格必须被转义。如果您的字符串包含内部引号字符 ("")，则也可以将其转义。

```
# Set one or more individual labels
LABEL com.example.version="0.0.1-beta"
LABEL vendor="ACME Incorporated"
LABEL com.example.release_date="2015-02-12"
LABEL com.example.version.is_production=""
```

一个镜像可以包含多个标签，在 1.10 之前，建议将所有标签合并为一条 `LABEL` 指令，以防止创建额外的层，但是现在这个不再是必须的了，以上内容也可以写成下面这样：

```
# Set multiple labels at once, using line-continuation characters to break long lines
LABEL vendor=ACME\ Incorporated \
      com.example.is_production="" \
      com.example.version="0.0.1-beta" \
      com.example.release_date="2015-02-12"
```

关于标签可以接受的键值对，参考[Understanding object labels](#)。

RUN

为了保持 Dockerfile 文件的可读性，以及可维护性，建议将长的或复杂的 `RUN` 指令用反斜杠 \ 分割成多行。

`RUN` 指令最常见的用法是安装包用的 `apt-get`。因为 `RUN apt-get` 指令会安装包，所以有几个问题需要注意。

- 不要使用 `RUN apt-get upgrade` 或 `dist-upgrade`，如果基础镜像中的某个包过时了，你应该联系它的维护者。如果你确定某个特定的包，比如 `foo`，需要升级，使用 `apt-get install -y foo` 就行，该指令会自动升级 `foo` 包。
- 永远将 `RUN apt-get update` 和 `apt-get install` 组合成一条 `RUN` 声明，例如：

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo
```

将 `apt-get update` 放在一条单独的 `RUN` 声明中会导致缓存问题以及后续的 `apt-get install` 失败。比如，假设你有一个 Dockerfile 文件：

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y curl
```

构建镜像后，所有的层都在 Docker 的缓存中。假设你后来又修改了其中的 `apt-get install` 添加了一个包：

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y curl nginx
```

Docker 发现修改后的 `RUN apt-get update` 指令和之前的完全一样。所以，`apt-get update` 不会执行，而是使用之前的缓存镜像。因为 `apt-get update` 没有运行，后面的 `apt-get install` 可能安装的是过时的 `curl` 和 `nginx` 版本。

使用 `RUN apt-get update && apt-get install -y` 可以确保你的 Dockerfiles 每次安装的都是包的最新的版本，而且这个过程不需要进一步的编码或额外干预。这项技术叫作 `cache busting`（缓存破坏）。你也可以显示指定一个包的版本号来达到 `cache-busting`，这就是所谓的固定版本，例如：

```
RUN apt-get update && apt-get install -y \
    package-bar \
    package-baz \
    package-foo=1.3.*
```

固定版本会迫使构建过程检索特定的版本，而不管缓存中有什么。这项技术也可以减少因所需包中未预料到的变化而导致的失败。

下面是一个 `RUN` 指令的示例模板，展示了所有关于 `apt-get` 的建议。

```
RUN apt-get update && apt-get install -y \
    aufs-tools \
    automake \
    build-essential \
    curl \
    dpkg-sig \
    libcap-dev \
    libsqlite3-dev \
    mercurial \
    reprepro \
    ruby1.9.1 \
    ruby1.9.1-dev \
    s3cmd=1.1.* \
    && rm -rf /var/lib/apt/lists/*
```

其中 `s3cmd` 指令指定了一个版本号 `1.1.*`。如果之前的镜像使用的是更旧的版本，指定新的版本会导致 `apt-get update` 缓存失效并确保安装的是新版本。另外，清理掉 `apt` 缓存 `var/lib/apt/lists` 可以减小镜像大小。因为 `RUN` 指令的开头为 `apt-get update`，包缓存总是在 `apt-get install` 之前刷新。

注意：官方的 Debian 和 Ubuntu 镜像会自动运行 `apt-get clean`，所以不需要显式的调用 `apt-get clean`。

CMD

`CMD` 指令用于执行目标镜像中包含的软件和任何参数。`CMD` 几乎都是以 `CMD ["executable", "param1", "param2"...]` 的形式使用。因此，如果创建镜像的目的是为了部署某个服务(比如 Apache)，你可能会执行类似于 `CMD ["apache2", "-DFOREGROUND"]` 形式的命令。

多数情况下，`CMD` 都需要一个交互式的 shell (`bash`, `Python`, `perl` 等)，例如 `CMD ["perl", "-de0"]`，或者 `CMD ["PHP", "-a"]`。使用这种形式意味着，当你执行类似 `docker run -it python` 时，你会进入一个准备好的 shell 中。

`CMD` 在极少的情况下才会以 `CMD ["param", "param"]` 的形式与 `ENTRYPOINT` 协同使用，除非你和你的镜像使用者都对 `ENTRYPOINT` 的工作方式十分熟悉。

EXPOSE

`EXPOSE` 指令用于指定容器将要监听的端口。因此，你应该为你的应用程序使用常见的端口。

例如，提供 Apache web 服务的镜像应该使用 `EXPOSE 80`，而提供 MongoDB 服务的镜像使用 `EXPOSE 27017`。

对于外部访问，用户可以在执行 `docker run` 时使用一个标志来指示如何将指定的端口映射到所选择的端口。

ENV

为了方便新程序运行，你可以使用 `ENV` 来为容器中安装的程序更新 `PATH` 环境变量。例如使用 `ENV PATH /usr/local/nginx/bin:$PATH` 来确保 `CMD ["nginx"]` 能正确运行。

`ENV` 指令也可用于为你想要容器化的服务提供必要的环境变量，比如 Postgres 需要的 `PGDATA`。最后，`ENV` 也能用于设置常见的版本号，比如下面的示例：

```
ENV PG_MAJOR 9.3
ENV PG_VERSION 9.3.4
RUN curl -SL http://example.com/postgres-$PG_VERSION.tar.xz | tar -xJc /usr/src/postgress
&& ...ENV PATH /usr/local/postgres-$PG_MAJOR/bin:$PATH
```

类似于程序中的常量，这种方法可以让你只需改变 `ENV` 指令来自动的改变容器中的软件版本。

ADD 和 COPY

虽然 `ADD` 和 `COPY` 功能类似，但一般优先使用 `COPY`。因为它比 `ADD` 更透明。`COPY` 只支持简单将本地文件拷贝到容器中，而 `ADD` 有一些并不明显的功能（比如本地 `tar` 提取和远程 URL 支持）。因此，`ADD`的最佳用例是将本地 `tar` 文件自动提取到镜像中，例如 `ADD rootfs.tar.xz`。

如果你的 Dockerfile 有多个步骤需要使用上下文中不同的文件。单独 COPY 每个文件，而不是一次性地 COPY 所有文件，这将保证每个步骤的构建缓存只在特定的文件变化时失效。例如：

```
COPY requirements.txt /tmp/
RUN pip install --requirement /tmp/requirements.txt
COPY . /tmp/
```

如果将 COPY . /tmp/ 放置在 RUN 指令之前，只要 . 目录中任何一个文件变化，都会导致后续指令的缓存失效。

为了让镜像尽量小，最好不要使用 ADD 指令从远程 URL 获取包，而是使用 curl 和 wget。这样你可以在文件提取完之后删掉不再需要的文件来避免在镜像中额外添加一层。比如尽量避免下面的用法：

```
ADD http://example.com/big.tar.xz /usr/src/things/
RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things
RUN make -C /usr/src/things all
```

而是应该使用下面这种方法：

```
RUN mkdir -p /usr/src/things \
&& curl -SL http://example.com/big.tar.xz \
| tar -xJC /usr/src/things \
&& make -C /usr/src/things all
```

上面使用的管道操作，所以没有中间文件需要删除。对于其他不需要 ADD 的自动提取功能的文件或目录，你应该使用 COPY。

ENTRYPOINT

ENTRYPOINT 的最佳用处是设置镜像的主命令，允许将镜像当成命令本身来运行（用 CMD 提供默认选项）。

例如，下面的示例镜像提供了命令行工具 s3cmd：

```
ENTRYPOINT ["s3cmd"]
CMD ["--help"]
```

现在直接运行该镜像创建的容器会显示命令帮助：

```
$ docker run s3cmd
```

或者提供正确的参数来执行某个命令：

```
$ docker run s3cmd ls s3://mybucket
```

这样镜像名可以当成命令行的参考。ENTRYPOINT 指令也可以结合一个辅助脚本使用，和前面命令行风格类似，即使启动工具需要不止一个步骤。

例如，Postgres 官方镜像使用下面的脚本作为 ENTRYPOINT：

```
#!/bin/bash
set -e
if [ "$1" = 'postgres' ]; then
    chown -R postgres "$PGDATA"

    if [ -z "$(ls -A "$PGDATA")" ]; then
        gosu postgres initdb
    fi

    exec gosu postgres "$@"
exec "$@"
```

注意：该脚本使用了 Bash 的内置命令 exec，所以最后运行的进程就是容器的 PID 为 1 的进程。这样，进程就可以接收到任何发送给容器的 Unix 信号了。

该辅助脚本被拷贝到容器，并在容器启动时通过 ENTRYPOINT 执行：

```
COPY ./docker-entrypoint.sh /
ENTRYPOINT ["/docker-entrypoint.sh"]
```

该脚本可以让用户用几种不同的方式和 Postgres 交互。你可以很简单地启动 Postgres：

```
$ docker run postgres
```

也可以执行 Postgres 并传递参数：

```
$ docker run postgres postgres --help
```

最后，你还可以启动另外一个完全不同的工具，比如 Bash：

```
$ docker run --rm -it postgres bash
```

VOLUME

VOLUME 指令用于暴露任何数据库存储文件，配置文件，或容器创建的文件和目录。强烈建议使用 VOLUME 来管理镜像中的可变部分和用户可以改变的部分。

USER

如果某个服务不需要特权执行，建议使用 USER 指令切换到非 root 用户。先在 Dockerfile 中使用类似 RUN groupadd -r postgres && useradd -r -g postgres postgres 的指令创建用户和用户组。

注意：在镜像中，用户和用户组每次被分配的 UID/GID 都是不确定的，下次重新构建镜像时被分配到的 UID/GID 可能会不一样。如果要依赖确定的 UID/GID，你应该显示的指定一个 UID/GID。

你应该避免使用 `sudo`，因为它不可预期的 TTY 和信号转发行为可能造成的问题比它能解决的问题还多。如果你真的需要和 `sudo` 类似的功能（例如，以 `root` 权限初始化某个守护进程，以非 `root` 权限执行它），你可以使用 `gosu`。

最后，为了减少层数和复杂度，避免频繁地使用 `USER` 来回切换用户。

WORKDIR

为了清晰性和可靠性，你应该总是在 `WORKDIR` 中使用绝对路径。另外，你应该使用 `WORKDIR` 来替代类似于 `RUN cd ... && do-something` 的指令，后者难以阅读、排错和维护。

ONBUILD

格式：`ONBUILD <其它指令>`。`ONBUILD` 是一个特殊的指令，它后面跟的是其它指令，比如 `RUN`、`COPY` 等，而这些指令，在当前镜像构建时并不会被执行。只有当以当前镜像为基础镜像，去构建下一级镜像的时候才会被执行。`Dockerfile` 中的其它指令都是为了定制当前镜像而准备的，唯有 `ONBUILD` 是为了帮助别人定制自己而准备的。

假设我们要制作 Node.js 所写的应用的镜像。我们都知道 Node.js 使用 `npm` 进行包管理，所有依赖、配置、启动信息等会放到 `package.json` 文件里。在拿到程序代码后，需要先进行 `npm install` 才可以获得所有需要的依赖。然后就可以通过 `npm start` 来启动应用。因此，一般来说会这样写 `Dockerfile`：

```
FROM node:slim
RUN mkdir /app
WORKDIR /app
COPY ./package.json /app
RUN [ "npm", "install" ]
COPY . /app/
CMD [ "npm", "start" ]
```

把这个 `Dockerfile` 放到 Node.js 项目的根目录，构建好镜像后，就可以直接拿来启动容器运行。但是如果还有第二个 Node.js 项目也差不多呢？好吧，那就再把这个 `Dockerfile` 复制到第二个项目里。那如果有第三个项目呢？再复制么？文件的副本越多，版本控制就越困难，让我们继续看这样的场景维护的问题：

如果第一个 Node.js 项目在开发过程中，发现这个 `Dockerfile` 里存在问题，比如敲错字了、或者需要安装额外的包，然后开发人员修复了这个 `Dockerfile`，再次构建，问题解决。第一个项目没问题了，但是第二个项目呢？虽然最初 `Dockerfile` 是复制、粘贴自第一个项目的，但是并不会因为第一个项目修复了他们的 `Dockerfile`，而第二个项目的 `Dockerfile` 就会被自动修复。

那么我们可不可以做一个基础镜像，然后各个项目使用这个基础镜像呢？这样基础镜像更新，各个项目不用同步 `Dockerfile` 的变化，重新构建后就继承了基础镜像的更新？好吧，可以，让我们看看这样的结果。那么上面的这个 `Dockerfile` 就会变为：

```
FROM node:slim
RUN mkdir /app
WORKDIR /app
CMD [ "npm", "start" ]
```

这里我们把项目相关的构建指令拿出来，放到子项目里去。假设这个基础镜像的名字为 my-node 的话，各个项目内的自己的 Dockerfile 就变为：

```
FROM my-node
COPY ./package.json /app
RUN [ "npm", "install" ]
COPY . /app/
```

基础镜像变化后，各个项目都用这个 Dockerfile 重新构建镜像，会继承基础镜像的更新。

那么，问题解决了么？没有。准确说，只解决了一半。如果这个 Dockerfile 里面有些东西需要调整呢？比如 npm install 都需要加一些参数，那怎么办？这一行 RUN 是不可能放入基础镜像的，因为涉及到了当前项目的 ./package.json，难道又要一个个修改么？所以说，这样制作基础镜像，只解决了原来的 Dockerfile 的前4条指令的变化问题，而后面三条指令的变化则完全没办法处理。

ONBUILD 可以解决这个问题。让我们用 ONBUILD 重新写一下基础镜像的 Dockerfile:

```
FROM node:slim
RUN mkdir /app
WORKDIR /app
ONBUILD COPY ./package.json /app
ONBUILD RUN [ "npm", "install" ]
ONBUILD COPY . /app/
CMD [ "npm", "start" ]
```

这次我们回到原始的 Dockerfile，但是这次将项目相关的指令加上 ONBUILD，这样在构建基础镜像的时候，这三行并不会被执行。然后各个项目的 Dockerfile 就变成了简单地：

```
FROM my-node
```

是的，只有这么一行。当在各个项目目录中，用这个只有一行的 Dockerfile 构建镜像时，之前基础镜像的那三行 ONBUILD 就会开始执行，成功的将当前项目的代码复制进镜像、并且针对本项目执行 npm install，生成应用镜像。

官方仓库示例

这些官方仓库的 Dockerfile 都是参考典范：<https://github.com/docker-library/docs>

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:00:25

14. Kubernetes 初体验

今天开始正式进入 Kubernetes 的课程学习，Kubernetes 我们已经听过很多了，那么什么是 Kubernetes 呢？

简介

Kubernetes 是 Google 团队发起的一个开源项目，它的目标是管理跨多个主机的容器，用于自动部署、扩展和管理容器化的应用程序，主要实现语言为 Go 语言。Kubernetes 的组件和架构还是相对复杂的，如果我们一上来就给大家讲解这些概念，可能很多同学都消化不了，所以我们先让我们的同学来使用我们的 Kubernetes，去感受下，去体验下面的一些概念和用法，等你对这些基本概念熟悉以后，再来给大家讲解 Kubernetes 的组件和架构应该就更加容易了。

怎样体验呢？当然最好的办法就是我们自己搭建一套集群了，但是如果完完全全都我们手动去搭建的话，第一是太耗时，第二是太复杂，可能我们现在还没有这个实力，没关系，我们可以使用一些工具来辅助我们。

1. katacoda 的课程：[katacoda](#)，可以在网站上帮我们启动一个 minikube 的环境（学习）

Get Started!

Scenarios Completed: 0 of 17 | Progress: 0% | Points: 0

[Create Your Free Account](#)

[Start Scenario](#)

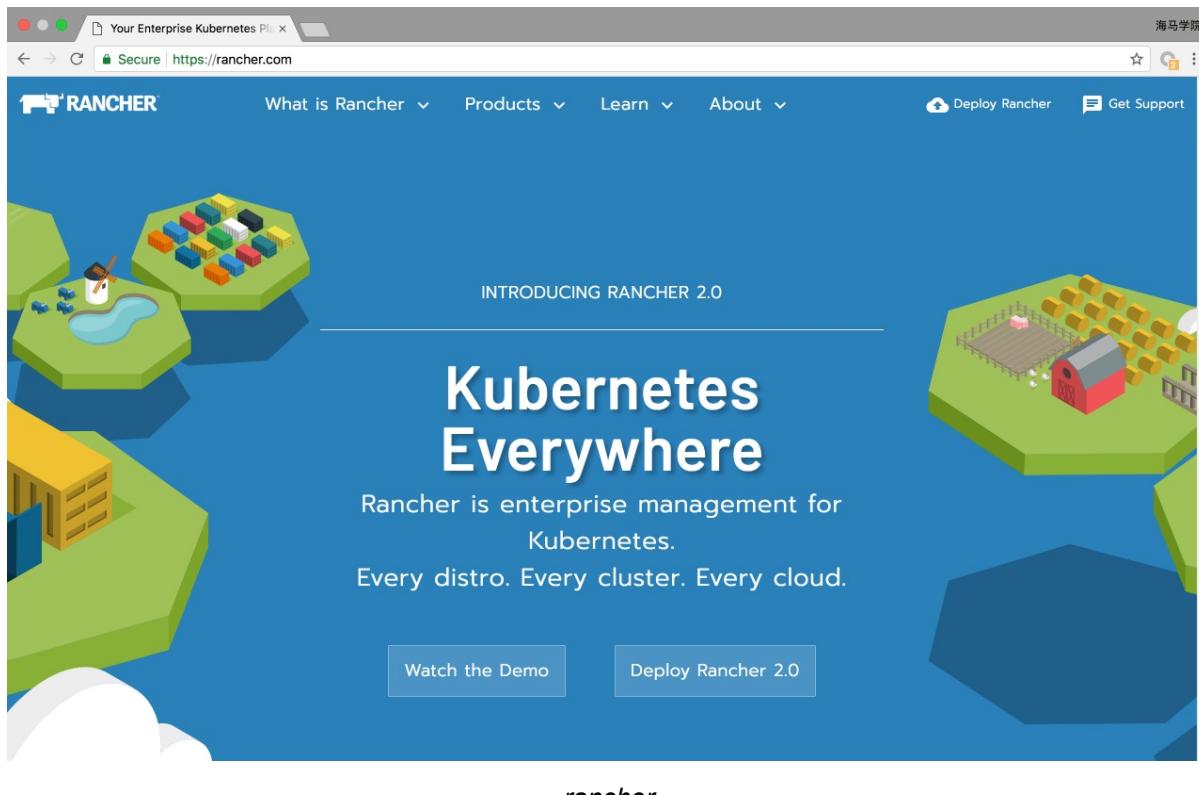
[Start Scenario](#)

Launch A Single Node Cluster
Learn how to launch a Single Node Minikube cluster including DNS and Kube UI

Launch a multi-node cluster using Kubeadm
Bootstrap a Kubernetes cluster using Kubeadm

2. 需要我们自己来搭建的 - [Rancher](#)，我们之前给大家介绍过，如果你网速不好的话安装 Rancher 可能需要花费一点时间，不过这是值得的。（测试）

```
$ docker run -d --restart=unless-stopped -p 80:80 -p 443:443 rancher/rancher:v2.0.0 # 查看日志
$ docker logs -f rancher
```



rancher

3.Docker for MAC/Windows (推荐) /[minikube](#)/ (本地)

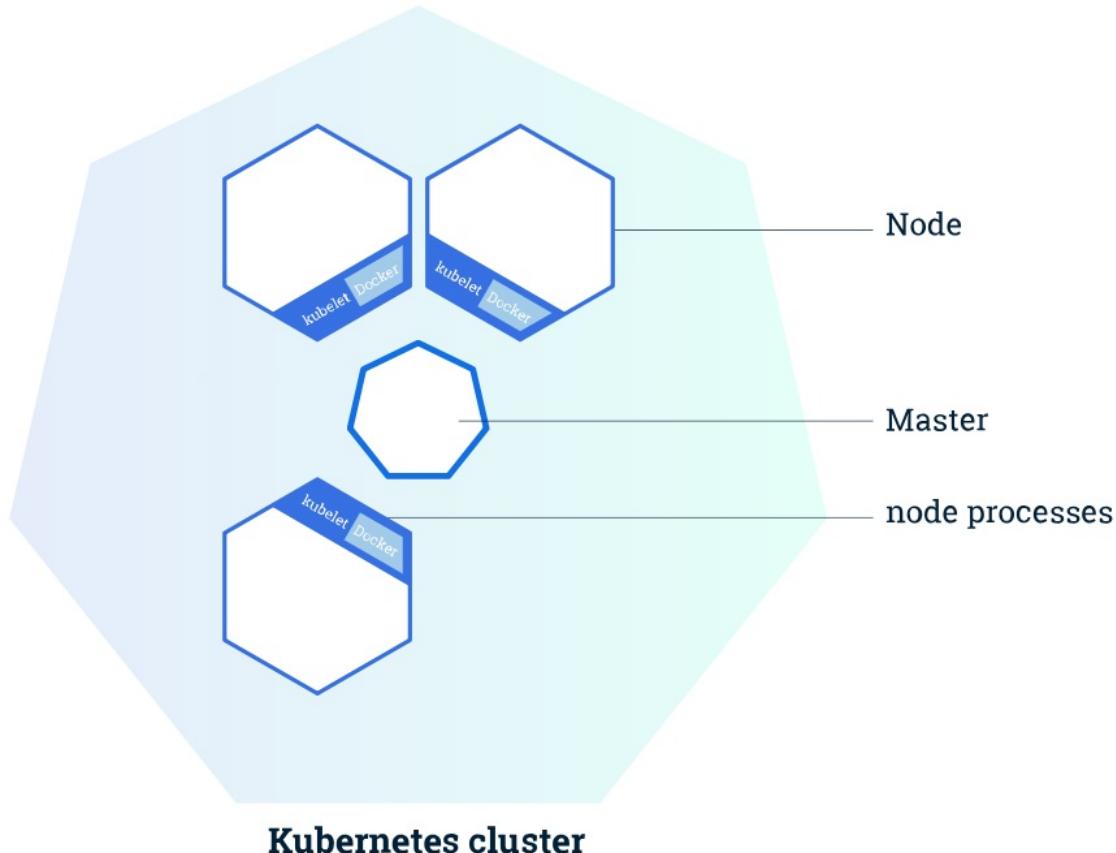
Docker for MAC/Windows 和 minikube 安装之前需要安装[kubectl](#)工具

4.[kubeadm](#) (测试)

5.[二进制纯手动搭建](#) (生产)

集群

集群是一组节点，这些节点可以是物理服务器或者虚拟机，在他上面安装了Kubernetes环境。

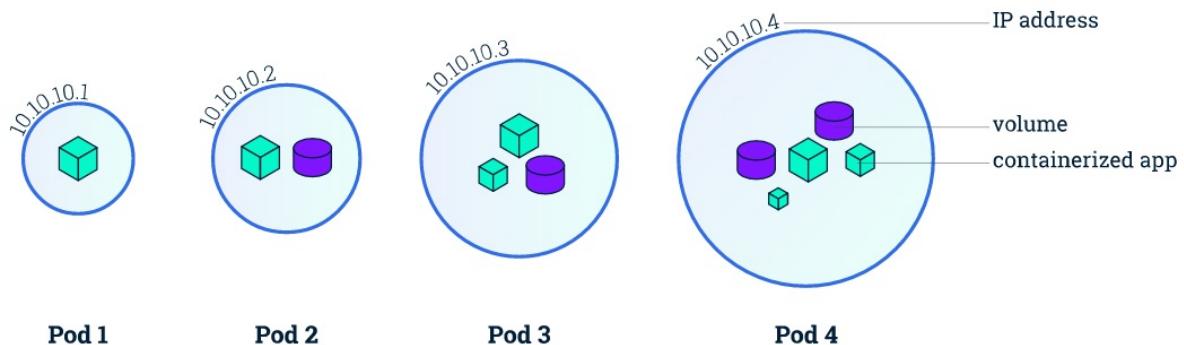


Master 负责管理集群, master 协调集群中的所有活动, 例如调度应用程序、维护应用程序的所需状态、扩展应用程序和滚动更新。节点是 Kubernetes 集群中的工作机器, 可以是物理机或虚拟机。每个工作节点都有一个 kubelet, 它是管理节点并与 Kubernetes Master 节点进行通信的代理。节点上还应具有处理容器操作的容器运行时, 例如 Docker 或 rkt。一个 Kubernetes 工作集群至少有三个节点。Master 管理集群, 而 节点 用于托管正在运行的应用程序。

当您在 Kubernetes 上部署应用程序时, 您可以告诉 master 启动应用程序容器。Master 调度容器在集群的节点上运行。节点使用 Master 公开的 Kubernetes API 与 Master 通信。用户也可以直接使用 Kubernetes 的 API 与集群交互。

Pod

Pod 是一组紧密关联的容器集合, 它们共享 PID、IPC、Network 和 UTS namespace, 是Kubernetes 调度的基本单位。Pod 的设计理念是支持多个容器在一个 Pod 中共享网络和文件系统, 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。



在 Kubernetes 中，所有对象都使用 manifest (yaml或json) 来定义，比如一个简单的 nginx 服务可以定义为 nginx.yaml，它包含一个镜像为 nginx 的容器：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Label

Label 是识别 Kubernetes 对象的标签，以 key/value 的方式附加到对象上（key最长不能超过63字节，value 可以为空，也可以是不超过253字节的字符串）。Label 不提供唯一性，并且实际上经常是很多对象（如Pods）都使用相同的 label 来标志具体的应用。Label 定义好后其他对象可以使用 Label Selector 来选择一组相同 label 的对象（比如Service 用 label 来选择一组 Pod）。Label Selector支持以下几种方式：

- 等式，如app=nginx和env!=production
- 集合，如env in (production, qa)
- 多个label（它们之间是AND关系），如app=nginx,env=test

Namespace

Namespace 是对一组资源和对象的抽象集合，比如可以用来将系统内部的对象划分为不同的项目组或用户组。常见的 pods, services,deployments 等都是属于某一个 namespace 的（默认是default），而 Node, PersistentVolumes 等则不属于任何 namespace。

Deployment

是否手动创建 Pod，如果想要创建同一个容器的多份拷贝，需要一个个分别创建出来么，能否将Pods划到逻辑组里？

Deployment 确保任意时间都有指定数量的 Pod“副本”在运行。如果为某个 Pod 创建了 Deployment 并且指定3个副本，它会创建3个 Pod，并且持续监控它们。如果某个 Pod 不响应，那么 Deployment 会替换它，保持总数为3。

如果之前不响应的 Pod 恢复了，现在就有4个 Pod 了，那么 Deployment 会将其中一个终止保持总数为3。如果在运行中将副本总数改为5，Deployment 会立刻启动2个新 Pod，保证总数为5。

Deployment 还支持回滚和滚动升级。

当创建 Deployment 时，需要指定两个东西：

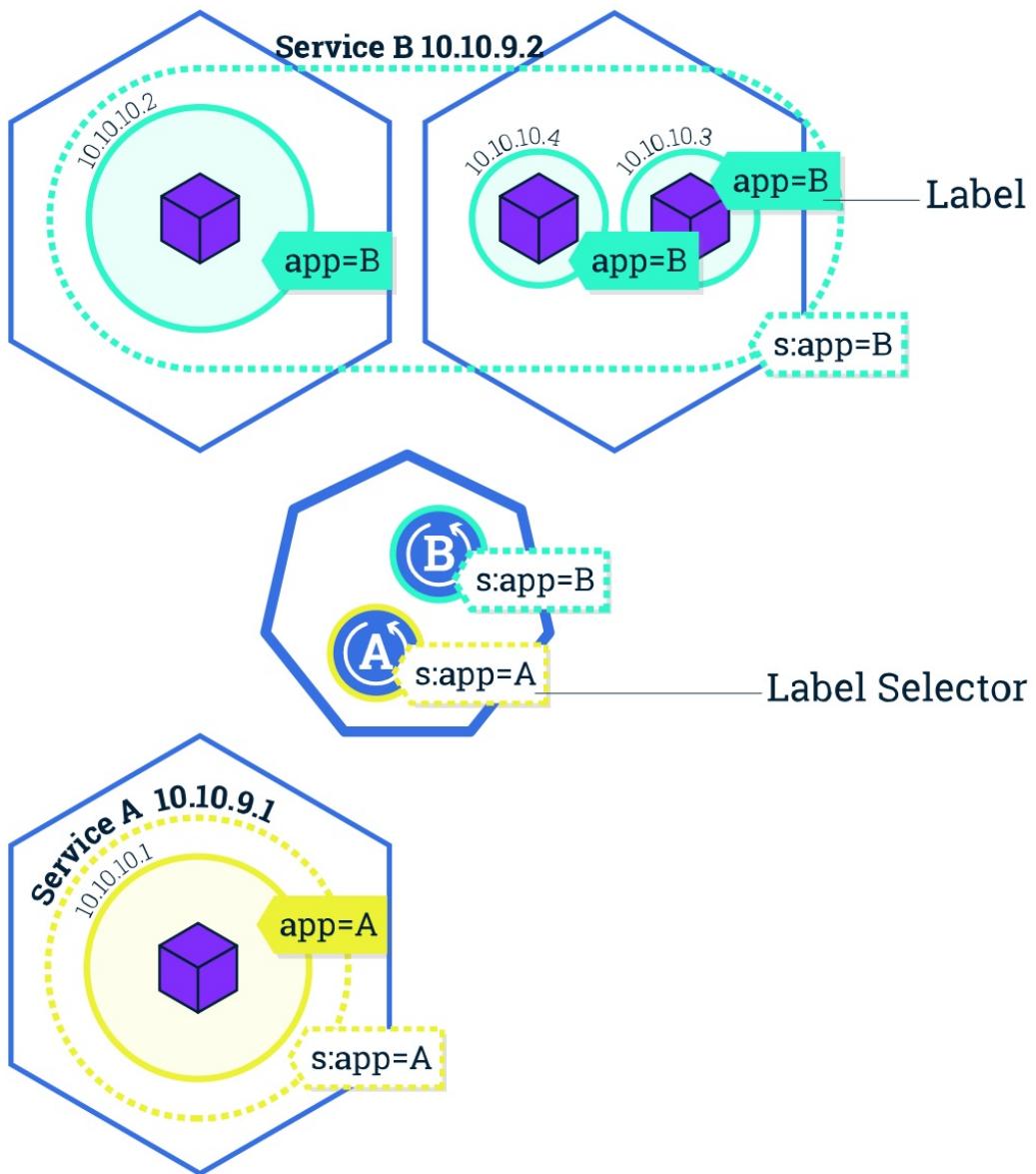
- Pod模板：用来创建 Pod 副本的模板
- Label标签：Deployment 需要监控的 Pod 的标签。

现在已经创建了 Pod 的一些副本，那么在这些副本上如何均衡负载呢？我们需要的是 Service。

Service

Service 是应用服务的抽象，通过 labels 为应用提供负载均衡和服务发现。匹配 labels 的Pod IP 和端口列表组成 endpoints，由 kube-proxy 负责将服务 IP 负载均衡到这些endpoints 上。

每个 Service 都会自动分配一个 cluster IP (仅在集群内部可访问的虚拟地址) 和 DNS 名，其他容器可以通过该地址或 DNS 来访问服务，而不需要了解后端容器的运行。



[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

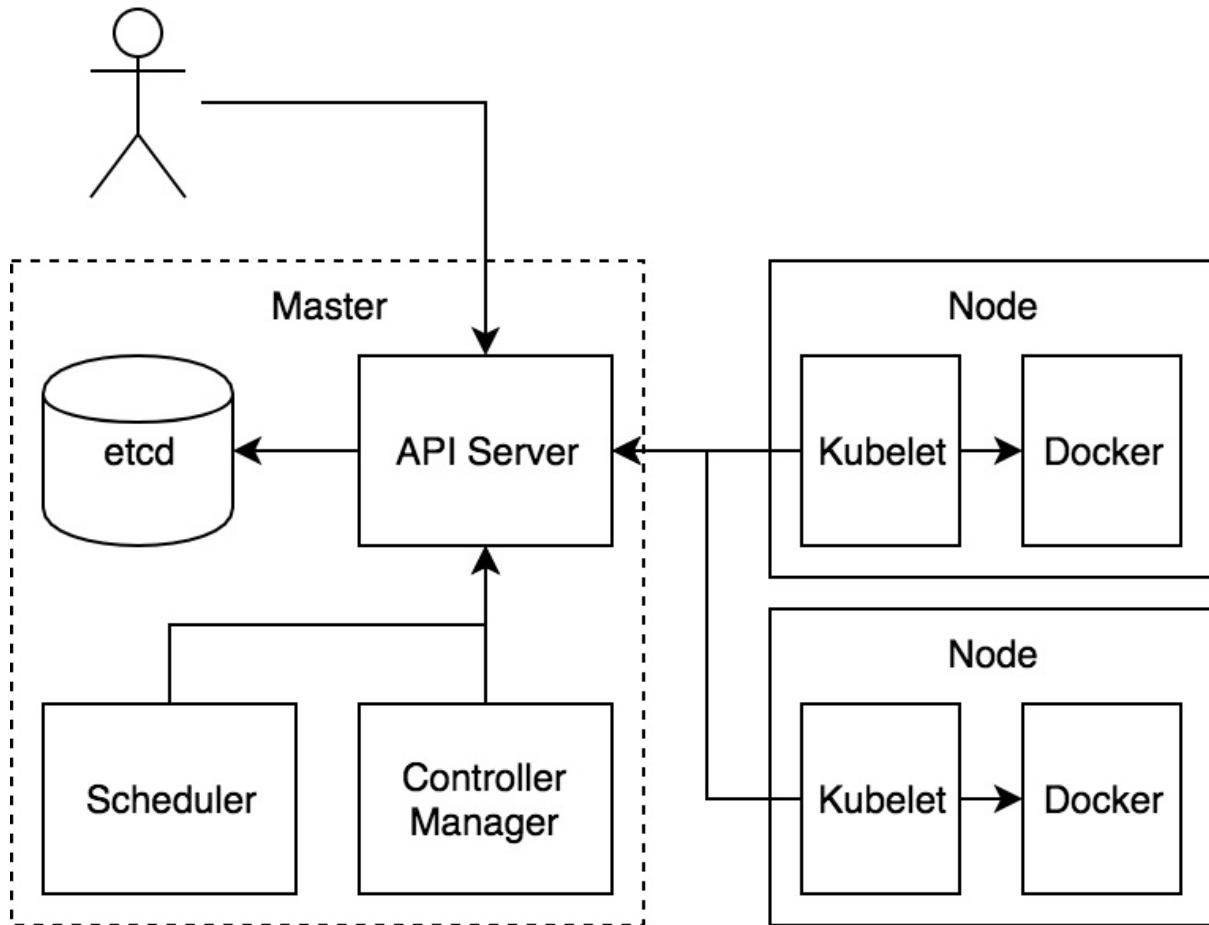
Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:00:30

15. 基本概念与组件

基本概念

Kubernetes 中的绝大部分概念都抽象成 Kubernetes 管理的一种资源对象，下面我们一起复习一下我们上节课遇到的一些资源对象：

- Master: Master 节点是 Kubernetes 集群的控制节点，负责整个集群的管理和控制。Master 节点上包含以下组件：
 - kube-apiserver: 集群控制的入口，提供 HTTP REST 服务
 - kube-controller-manager: Kubernetes 集群中所有资源对象的自动化控制中心
 - kube-scheduler: 负责 Pod 的调度
- Node: Node 节点是 Kubernetes 集群中的工作节点，Node 上的工作负载由 Master 节点分配，工作负载主要是运行容器应用。Node 节点上包含以下组件：
 - kubelet: 负责 Pod 的创建、启动、监控、重启、销毁等工作，同时与 Master 节点协作，实现集群管理的基本功能。
 - kube-proxy: 实现 Kubernetes Service 的通信和负载均衡
 - 运行容器化(Pod)应用
- Pod: Pod 是 Kubernetes 最基本的部署调度单元。每个 Pod 可以由一个或多个业务容器和一个根容器(Pause 容器)组成。一个 Pod 表示某个应用的一个实例
- ReplicaSet: 是 Pod 副本的抽象，用于解决 Pod 的扩容和伸缩
- Deployment: Deployment 表示部署，在内部使用ReplicaSet 来实现。可以通过 Deployment 来生成相应的 ReplicaSet 完成 Pod 副本的创建
- Service: Service 是 Kubernetes 最重要的资源对象。Kubernetes 中的 Service 对象可以对应微服务架构中的微服务。Service 定义了服务的访问入口，服务的调用者通过这个地址访问 Service 后端的 Pod 副本实例。Service 通过 Label Selector 同后端的 Pod 副本建立关系，Deployment 保证后端Pod 副本的数量，也就是保证服务的伸缩性。



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，就是一个数据库；
- 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 (CSI) 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；

当然了除了上面的这些核心组件，还有一些推荐的插件：

- 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

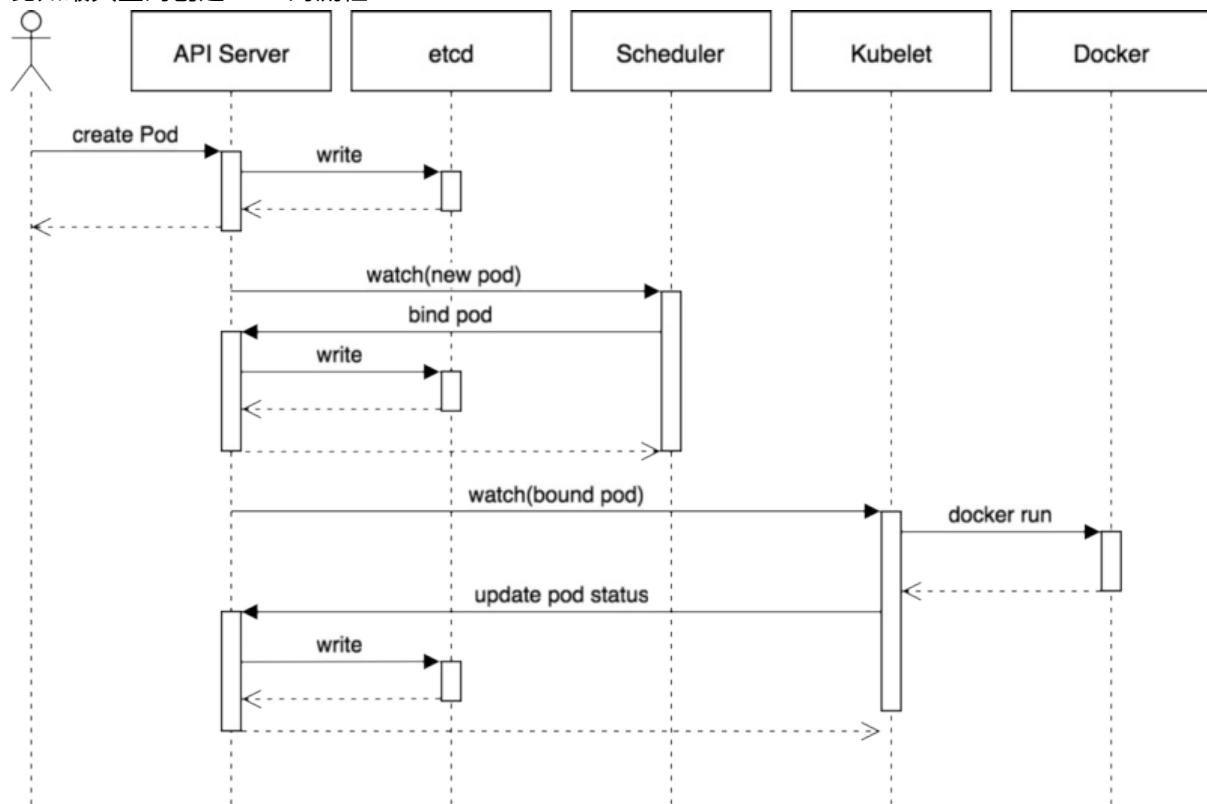
组件通信

Kubernetes 多组件之间的通信原理：

- apiserver 负责 etcd 存储的所有操作，且只有 apiserver 才直接操作 etcd 集群

- apiserver 对内（集群中的其他组件）和对外（用户）提供统一的 REST API，其他组件均通过 apiserver 进行通信
 - controller manager、scheduler、kube-proxy 和 kubelet 等均通过 apiserver watch API 监测资源变化情况，并对资源作相应的操作
 - 所有需要更新资源状态的操作均通过 apiserver 的 REST API 进行
- apiserver 也会直接调用 kubelet API（如 logs, exec, attach 等），默认不校验 kubelet 证书，但可以通过 `--kubelet-certificate-authority` 开启（而 GKE 通过 SSH 隧道保护它们之间的通信）

比如最典型的创建 Pod 的流程：



- 用户通过 REST API 创建一个 Pod
- apiserver 将其写入 etcd
- scheduluer 检测到未绑定 Node 的 Pod，开始调度并更新 Pod 的 Node 绑定
- kubelet 检测到有新的 Pod 调度过来，通过 container runtime 运行该 Pod
- kubelet 通过 container runtime 取到 Pod 状态，并更新到 apiserver 中

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

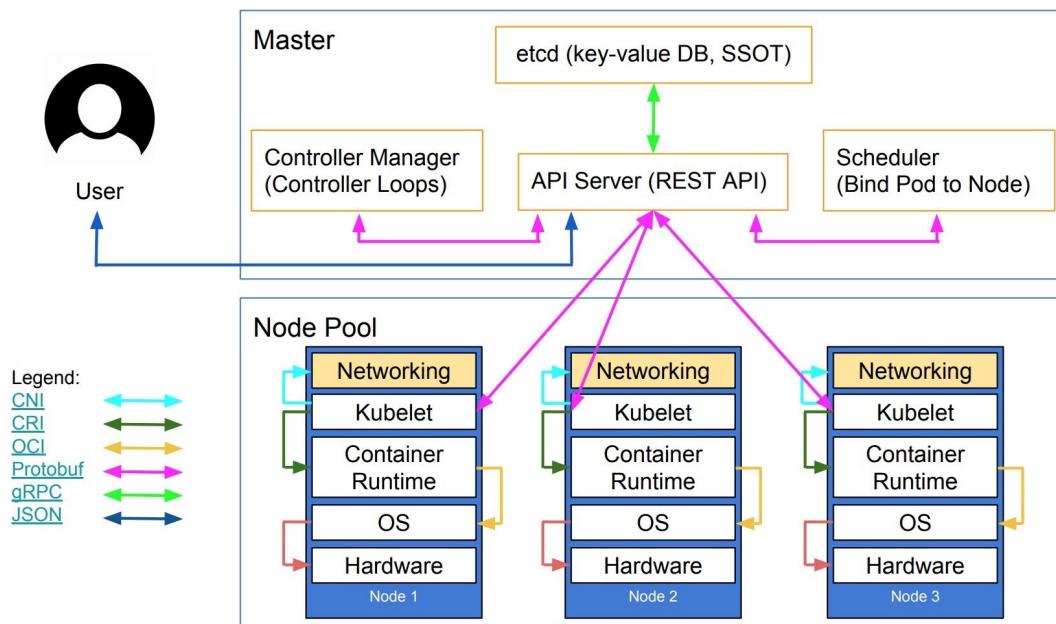
Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:00:35

16. 用 kubeadm 搭建集群环境

架构

上节课我们给大家讲解了 k8s 的基本概念与几个主要的组件，我们在了解了 k8s 的基本概念过后，实际上就可以去正式使用了，但是我们前面的课程都是在 katacoda 上面进行的演示，只提供给我们15分钟左右的使用时间，所以最好的方式还是我们自己来手动搭建一套 k8s 的环境，在搭建环境之前，我们再来看一张更丰富的k8s的架构图。

Kubernetes' high-level component architecture



- 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 等
 - Kubernetes 内部：CRI、CNI、CVI、镜像仓库、Cloud Provider、集群自身的配置和管理等

在更进一步了解了 k8s 集群的架构后，我们就可以来证书的的安装我们的 k8s 集群环境了，我们这里使用的是 kubeadm 工具来进行集群的搭建。

kubeadm 是 Kubernetes 官方提供的用于快速安装 Kubernetes 集群的工具，通过将集群的各个组件进行容器化安装管理，通过 kubeadm 的方式安装集群比二进制的方式安装要方便不少，但是目录 kubeadm 还处于 beta 状态，还不能用于生产环境，[Using kubeadm to Create a Cluster 文档](#)中已经说明 kubeadm 将会很快能够用于生产环境了。对于现阶段想要用于生产环境的，建议还是参考我们前面的文章：[手动搭建高可用的 kubernetes 集群](#)或者[视频教程](#)。

环境

我们这里准备两台 CentOS7 的主机用于安装，后续节点可以根据需要添加即可：

```
$ cat /etc/hosts  
10.151.30.57 master  
10.151.30.62 node01
```

禁用防火墙：

```
$ systemctl stop firewalld  
$ systemctl disable firewalld
```

禁用 SELinux：

```
$ setenforce 0  
$ cat /etc/selinux/config  
SELINUX=disabled
```

创建 /etc/sysctl.d/k8s.conf 文件，添加如下内容：

```
net.bridge.bridge-nf-call-ip6tables = 1  
net.bridge.bridge-nf-call-iptables = 1  
net.ipv4.ip_forward = 1
```

执行如下命令使修改生效：

```
$ modprobe br_netfilter  
$ sysctl -p /etc/sysctl.d/k8s.conf
```

镜像

如果你的节点上面有科学上网的工具，可以忽略这一步，我们需要提前将所需的 gcr.io 上面的镜像下载到节点上面，当然前提条件是你已经成功安装了 docker。master 节点，执行下面的命令：

```
docker pull cnych/kube-apiserver-amd64:v1.10.0  
docker pull cnych/kube-scheduler-amd64:v1.10.0  
docker pull cnych/kube-controller-manager-amd64:v1.10.0  
docker pull cnych/kube-proxy-amd64:v1.10.0  
docker pull cnych/k8s-dns-kube-dns-amd64:1.14.8  
docker pull cnych/k8s-dns-dnsmasq-nanny-amd64:1.14.8
```

```

docker pull cnych/k8s-dns-sidecar-amd64:1.14.8
docker pull cnych/etcd-amd64:3.1.12
docker pull cnych/flannel:v0.10.0-amd64
docker pull cnych/pause-amd64:3.1

docker tag cnych/kube-apiserver-amd64:v1.10.0 k8s.gcr.io/kube-apiserver-amd64:v1.10.0
docker tag cnych/kube-scheduler-amd64:v1.10.0 k8s.gcr.io/kube-scheduler-amd64:v1.10.0
docker tag cnych/kube-controller-manager-amd64:v1.10.0 k8s.gcr.io/kube-controller-manager-
amd64:v1.10.0
docker tag cnych/kube-proxy-amd64:v1.10.0 k8s.gcr.io/kube-proxy-amd64:v1.10.0
docker tag cnych/k8s-dns-kube-dns-amd64:1.14.8 k8s.gcr.io/k8s-dns-kube-dns-amd64:1.14.8
docker tag cnych/k8s-dns-dnsMasq-nanny-amd64:1.14.8 k8s.gcr.io/k8s-dns-dnsMasq-nanny-amd64
:1.14.8
docker tag cnych/k8s-dns-sidecar-amd64:1.14.8 k8s.gcr.io/k8s-dns-sidecar-amd64:1.14.8
docker tag cnych/etcd-amd64:3.1.12 k8s.gcr.io/etcd-amd64:3.1.12
docker tag cnych/flannel:v0.10.0-amd64 quay.io/coreos/flannel:v0.10.0-amd64
docker tag cnych/pause-amd64:3.1 k8s.gcr.io/pause-amd64:3.1

```

可以将上面的命令保存为一个 shell 脚本，然后直接执行即可。这些镜像是在 master 节点上需要使用到的镜像，一定要提前下载下来。其他Node，执行下面的命令：

```

docker pull cnych/kube-proxy-amd64:v1.10.0
docker pull cnych/flannel:v0.10.0-amd64
docker pull cnych/pause-amd64:3.1
docker pull cnych/kubernetes-dashboard-amd64:v1.8.3
docker pull cnych/heapster-influxdb-amd64:v1.3.3
docker pull cnych/heapster-grafana-amd64:v4.4.3
docker pull cnych/heapster-amd64:v1.4.2
docker pull cnych/k8s-dns-kube-dns-amd64:1.14.8
docker pull cnych/k8s-dns-dnsMasq-nanny-amd64:1.14.8
docker pull cnych/k8s-dns-sidecar-amd64:1.14.8

docker tag cnych/flannel:v0.10.0-amd64 quay.io/coreos/flannel:v0.10.0-amd64
docker tag cnych/pause-amd64:3.1 k8s.gcr.io/pause-amd64:3.1
docker tag cnych/kube-proxy-amd64:v1.10.0 k8s.gcr.io/kube-proxy-amd64:v1.10.0

docker tag cnych/k8s-dns-kube-dns-amd64:1.14.8 k8s.gcr.io/k8s-dns-kube-dns-amd64:1.14.8
docker tag cnych/k8s-dns-dnsMasq-nanny-amd64:1.14.8 k8s.gcr.io/k8s-dns-dnsMasq-nanny-amd64
:1.14.8
docker tag cnych/k8s-dns-sidecar-amd64:1.14.8 k8s.gcr.io/k8s-dns-sidecar-amd64:1.14.8

docker tag cnych/kubernetes-dashboard-amd64:v1.8.3 k8s.gcr.io/kubernetes-dashboard-amd64:v
1.8.3
docker tag cnych/heapster-influxdb-amd64:v1.3.3 k8s.gcr.io/heapster-influxdb-amd64:v1.3.3
docker tag cnych/heapster-grafana-amd64:v4.4.3 k8s.gcr.io/heapster-grafana-amd64:v4.4.3
docker tag cnych/heapster-amd64:v1.4.2 k8s.gcr.io/heapster-amd64:v1.4.2

```

上面的这些镜像是在 Node 节点中需要用到的镜像，在 join 节点之前也需要先下载到节点上面。

安装 kubeadm、kubelet、kubectl

在确保 docker 安装完成后，上面的相关环境配置也完成了，对应所需要的镜像(如果可以科学上网可以跳过这一步)也下载完成了，现在我们就可以来安装 kubeadm 了，我们这里是通过指定 yum 源的方式来进行安装的：

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-e17-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
```

当然了，上面的 `yum` 源也是需要科学上网的，如果不能科学上网的话，我们可以使用阿里云的源进行安装：

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=http://mirrors.aliyun.com/kubernetes/yum/repos/kubernetes-e17-x86_64
enabled=1
gpgcheck=0
repo_gpgcheck=0
gpgkey=http://mirrors.aliyun.com/kubernetes/yum/doc/yum-key.gpg
http://mirrors.aliyun.com/kubernetes/yum/doc/rpm-package-key.gpg
EOF
```

目前阿里云的源最新版本已经是1.10.2版本，所以可以直接安装，由于我们上面的相关镜像是关联的1.10版本，所以我们安装的时候需要指定版本。`yum` 源配置完成后，执行安装命令即可：

```
$ yum makecache fast && yum install -y kubelet-1.10.0-0 kubeadm-1.10.0-0 kubectl-1.10.0-0
```

正常情况我们可以都能顺利安装完成上面的文件。

配置 kubelet

安装完成后，我们还需要对 `kubelet` 进行配置，因为用 `yum` 源的方式安装的 `kubelet` 生成的配置文件将参数 `--cgroup-driver` 改成了 `systemd`，而 `docker` 的 `cgroup-driver` 是 `cgroupfs`，这二者必须一致才行，我们可以通过 `docker info` 命令查看：

```
$ docker info |grep Cgroup
Cgroup Driver: cgroupfs
```

修改文件 `kubelet` 的配置文件 `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf`，将其中的 `KUBELET_CGROUP_ARGS` 参数改成 `cgroupfs`：

```
Environment="KUBELET_CGROUP_ARGS=--cgroup-driver=cgroupfs"
```

另外还有一个问题是关于交换分区的，之前我们在手动搭建高可用的 kubernetes 集群一文中已经提到过，Kubernetes 从1.8开始要求关闭系统的 Swap，如果不关闭，默认配置的 kubelet 将无法启动，我们可以通过 kubelet 的启动参数 `--fail-swap-on=false` 更改这个限制，所以我们需要在上面的配置文件中增加一项配置(在 `ExecStart` 之前)：

```
Environment="KUBELET_EXTRA_ARGS=--fail-swap-on=false"
```

当然最好的还是将 swap 给关掉，这样能提高 kubelet 的性能。修改完成后，重新加载我们的配置文件即可：

```
$ systemctl daemon-reload
```

集群安装初始化

到这里我们的准备工作就完成了，接下来我们就可以在 `master` 节点上用 `kubeadm` 命令来初始化我们的集群了：

```
$ kubeadm init --kubernetes-version=v1.10.0 --pod-network-cidr=10.244.0.0/16 --apiserver-advertise-address=10.151.30.57
```

命令非常简单，就是 `kubeadm init`，后面的参数是需要安装的集群版本，因为我们这里选择 `flannel` 作为 Pod 的网络插件，所以需要指定 `--pod-network-cidr=10.244.0.0/16`，然后是 `apiserver` 的通信地址，这里就是我们 `master` 节点的 IP 地址。执行上面的命令，如果出现 `running with swap on is not supported. Please disable swap` 之类的错误，则我们还需要增加一个参数 `--ignore-preflight-errors=Swap` 来忽略 swap 的错误提示信息：

```
$ kubeadm init \
  --kubernetes-version=v1.10.0 \
  --pod-network-cidr=10.244.0.0/16 \
  --apiserver-advertise-address=10.151.30.57 \
  --ignore-preflight-errors=Swap
[init] Using Kubernetes version: v1.10.0
[init] Using Authorization modes: [Node RBAC]
[preflight] Running pre-flight checks.
  [WARNING FileExisting-crictl]: crictl not found in system pathSuggestion: go get github.com/kubernetes-incubator/cri-tools/cmd/crictl
  [preflight] Starting the kubelet service
  [certificates] Generated ca certificate and key.
  [certificates] Generated apiserver certificate and key.
  [certificates] apiserver serving cert is signed for DNS names [ydzs-master1 kubernetes kubernetes.default kubernetes.default.svc kubernetes.default.svc.cluster.local] and IPs [10.9.6.0.1 10.151.30.57]
  [certificates] Generated apiserver-kubelet-client certificate and key.
  [certificates] Generated etcd/ca certificate and key.
  [certificates] Generated etcd/server certificate and key.
  [certificates] etcd/server serving cert is signed for DNS names [localhost] and IPs [127.0.0.1]
  [certificates] Generated etcd/peer certificate and key.
  [certificates] etcd/peer serving cert is signed for DNS names [ydzs-master1] and IPs [10.151.30.57]
```

```
[certificates] Generated etcd/healthcheck-client certificate and key.
[certificates] Generated apiserver-etcd-client certificate and key.
[certificates] Generated sa key and public key.
[certificates] Generated front-proxy-ca certificate and key.
[certificates] Generated front-proxy-client certificate and key.
[certificates] Valid certificates and keys now exist in "/etc/kubernetes/pki"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/admin.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/kubelet.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/controller-manager.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/scheduler.conf"
[controlplane] Wrote Static Pod manifest for component kube-apiserver to "/etc/kubernetes/manifests/kube-apiserver.yaml"
[controlplane] Wrote Static Pod manifest for component kube-controller-manager to "/etc/kubernetes/manifests/kube-controller-manager.yaml"
[controlplane] Wrote Static Pod manifest for component kube-scheduler to "/etc/kubernetes/manifests/kube-scheduler.yaml"
[etcd] Wrote Static Pod manifest for a local etcd instance to "/etc/kubernetes/manifests/etcd.yaml"
[init] Waiting for the kubelet to boot up the control plane as Static Pods from directory "/etc/kubernetes/manifests".
[init] This might take a minute or longer if the control plane images have to be pulled.
[apiclient] All control plane components are healthy after 22.007661 seconds
[uploadconfig] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" Namespace
[markmaster] Will mark node ydzs-master1 as master by adding a label and a taint
[markmaster] Master ydzs-master1 tainted and labelled with key/value: node-role.kubernetes.io/master=""
[bootstraptoken] Using token: 8xomlq.0cdf2pbvjs2gjho3
[bootstraptoken] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for nodes to get long term certificate credentials
[bootstraptoken] Configured RBAC rules to allow the csrapprover controller automatically approve CSRs from a Node Bootstrap Token
[bootstraptoken] Configured RBAC rules to allow certificate rotation for all node client certificates in the cluster
[bootstraptoken] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy
```

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

You should now deploy a pod network to the cluster.

Run "`kubectl apply -f [podnetwork].yaml`" with one of the options listed at:
<https://kubernetes.io/docs/concepts/cluster-administration/addons/>

You can now join any number of machines by running the following on each node as root:

```
kubeadm join 10.151.30.57:6443 --token 8xomlq.0cdf2pbvjs2gjho3 --discovery-token-ca-cert-hash sha256:92802317cb393682c1d1356c15e8b4ec8af2b8e5143ffd04d8be4eafb5fae368
```

要注意将上面的加入集群的命令保存下面，如果忘记保存上面的 token 和 sha256 值的话也不用担心，我们可以使用下面的命令来查找：

```
$ kubeadm token list
kubeadm token list
TOKEN          TTL      EXPIRES           USAGES          D
DESCRIPTION
i5gbaw.os1iow5tdo17rwdu  23h     2018-05-18T01:32:55+08:00  authentication,signing  T
he default bootstrap token generated by 'kubeadm init'.  system:bootstrappers:kubeadm:default-node-token
```

要查看 CA 证书的 sha256 的值的话，我们可以使用 `openssl` 来读取证书获取 sha256 的值：

```
$ openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>
/dev/null | openssl dgst -sha256 -hex | sed 's/^.* //'
e9ca4d9550e698105f1d8fae7ecfd297dd9331ca7d50b5493fa0491b2b4df40c
```

另外还需要注意的是当前版本的 `kubeadm` 支持的 `docker` 版本最大是 17.03，所以要注意下。上面的信息记录了 `kubeadm` 初始化整个集群的过程，生成相关的各种证书、`kubeconfig` 文件、`bootstraptoken` 等等，后边是使用 `kubeadm join` 往集群中添加节点时用到的命令，下面的命令是配置如何使用 `kubectl` 访问集群的方式：

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

最后给出了将节点加入集群的命令：

```
kubeadm join 10.151.30.57:6443 --token 8xom1q.0cdf2pbvjs2gjho3 --discovery-token-ca-cert-h
ash sha256:92802317cb393682c1d1356c15e8b4ec8af2b8e5143ffd04d8be4eafb5fae368
```

我们根据上面的提示配置好 `kubectl` 后，就可以使用 `kubectl` 来查看集群的信息了：

```
$ kubectl get cs
NAME          STATUS    MESSAGE           ERROR
scheduler     Healthy   ok
controller-manager  Healthy   ok
etcd-0        Healthy   {"health": "true"}
$ kubectl get csr
NAME                           AGE      REQUESTOR
CONDITION
node-csr-8qygb8Hjxj-byhbRHawropk81LHNpqZCTePeWoZs3-g  1h      system:bootstrap:8xom1q
Approved,Issued
$ kubectl get nodes
NAME          STATUS    ROLES      AGE      VERSION
ydzs-master1  Ready     master     3h      v1.10.0
```

如果你的集群安装过程中遇到了其他问题，我们可以使用下面的命令来进行重置：

```
$ kubeadm reset
$ ifconfig cni0 down && ip link delete cni0
$ ifconfig flannel.1 down && ip link delete flannel.1
$ rm -rf /var/lib/cni/
```

安装 Pod Network

接下来我们来安装 flannel 网络插件，很简单，和安装普通的 POD 没什么两样：

```
$ wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
$ kubectl apply -f kube-flannel.yml
clusterrole.rbac.authorization.k8s.io "flannel" created
clusterrolebinding.rbac.authorization.k8s.io "flannel" created
serviceaccount "flannel" created
configmap "kube-flannel-cfg" created
daemonset.extensions "kube-flannel-ds" created
```

另外需要注意的是如果你的节点有多个网卡的话，需要在 kube-flannel.yml 中使用 `--iface` 参数指定集群主机内网网卡的名称，否则可能会出现 dns 无法解析。flanneld 启动参数加上 `--iface=<iface-name>`

```
args:
- --ip-masq
- --kube-subnet-mgr
- --iface=eth0
```

安装完成后使用 `kubectl get pods` 命令可以查看到我们集群中的组件运行状态，如果都是Running 状态的话，那么恭喜你，你的 master 节点安装成功了。

```
$ kubectl get pods --all-namespaces
NAMESPACE      NAME          READY   STATUS    RESTARTS   AGE
kube-system    etcd-ydzs-master1   1/1     Running   0          10m
kube-system    kube-apiserver-ydzs-master1   1/1     Running   0          10m
kube-system    kube-controller-manager-ydzs-master1   1/1     Running   0          10m
kube-system    kube-dns-86f4d74b45-f5595   3/3     Running   0          10m
kube-system    kube-flannel-ds-qxjs2   1/1     Running   0          1m
kube-system    kube-proxy-vf5fg    1/1     Running   0          10m
kube-system    kube-scheduler-ydzs-master1   1/1     Running   0          10m
```

添加节点

同样的上面的环境配置、docker 安装、kubeadm、kubelet、kubectl 这些都在 Node(10.151.30.62) 节点安装配置好过后，我们就可以直接在 Node 节点上执行 `kubeadm join` 命令了（上面初始化的时候有），同样加上参数 `--ignore-preflight-errors=Swap`：

```
$ kubeadm join 10.151.30.57:6443 --token 8xomlq.0cdf2pbvjs2gjho3 --discovery-token-ca-cert
-hash sha256:92802317cb393682c1d1356c15e8b4ec8af2b8e5143ffd04d8be4eafb5fae368 --ignore-pre
flight-errors=Swap
[preflight] Running pre-flight checks.
[WARNING Swap]: running with swap on is not supported. Please disable swap
[WARNING FileExisting-crictl]: crictl not found in system path
Suggestion: go get github.com/kubernetes-incubator/cri-tools/cmd/crictl
[discovery] Trying to connect to API Server "10.151.30.57:6443"
[discovery] Created cluster-info discovery client, requesting info from "https://10.151.30
.57:6443"
```

```
[discovery] Requesting info from "https://10.151.30.57:6443" again to validate TLS against
the pinned public key
[discovery] Cluster info signature and contents are valid and TLS certificate validates ag
ainst pinned roots, will use API Server "10.151.30.57:6443"
[discovery] Successfully established connection with API Server "10.151.30.57:6443"

This node has joined the cluster:
* Certificate signing request was sent to master and a response
  was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the master to see this node join the cluster.
```

我们可以看到该节点已经加入到集群中去了，然后我们把 master 节点的 `~/.kube/config` 文件拷贝到当前节点对应的位置即可使用 `kubectl` 命令行工具了。

```
$ kubectl get nodes
NAME           STATUS    ROLES      AGE       VERSION
evjfaxic      Ready     <none>    1h        v1.10.0
ydzs-master1   Ready     master    3h        v1.10.0
```

到这里就算我们的集群部署成功了，接下来就可以根据我们的需要安装一些附加的插件，比如 `Dashboard`、`Heapster`、`Ingress-Controller` 等等，这些插件的安装方法就和我们之前手动安装集群的方式方法一样了，这里就不在重复了，有问题可以在 `github` 上留言讨论。

Pods							
Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)	
● kubernetes-dashboard-6f59cd7699	ydzs-master1	Waiting: ContainerCreate	0	9 seconds	-	-	
✓ heapster-676cc864c6-2vtsf	evjfaxic	Running	0	55 minutes	0.002	34.461 Mi	
✓ monitoring-grafana-69df66f668-l6z2	evjfaxic	Running	0	55 minutes	0.001	11.383 Mi	
✓ monitoring-influxdb-78d4c6f5b6-wj	evjfaxic	Running	0	55 minutes	0.003	34.414 Mi	
✓ kubernetes-dashboard-7d5dcdb6d5	evjfaxic	Running	0	an hour	0	23.816 Mi	
✓ kube-proxy-xjpzb	evjfaxic	Running	0	an hour	0.006	14.176 Mi	
✓ kube-flannel-ds-n8zf9	evjfaxic	Running	0	an hour	0.005	13.555 Mi	
✓ kube-flannel-ds-qxjs2	ydzs-master1	Running	3	3 hours	0.006	13.109 Mi	
✓ etcd-ydzs-master1	ydzs-master1	Running	2	3 hours	0.038	82.633 Mi	
✓ kube-controller-manager-ydzs-mas	ydzs-master1	Running	2	3 hours	0.097	41.227 Mi	

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-04-02 09:54:59

17. 安装 Dashboard 插件

Kubernetes Dashboard 是 k8s 集群的一个 WEB UI 管理工具，代码托管在 github 上，地址：<https://github.com/kubernetes/dashboard>

安装：

直接使用官方的配置文件安装即可：

```
$ wget https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml
```

为了测试方便，我们将 Service 改成 NodePort 类型，然后直接部署新版本的 dashboard 即可。

```
$ kubectl create -f kubernetes-dashboard.yaml
```

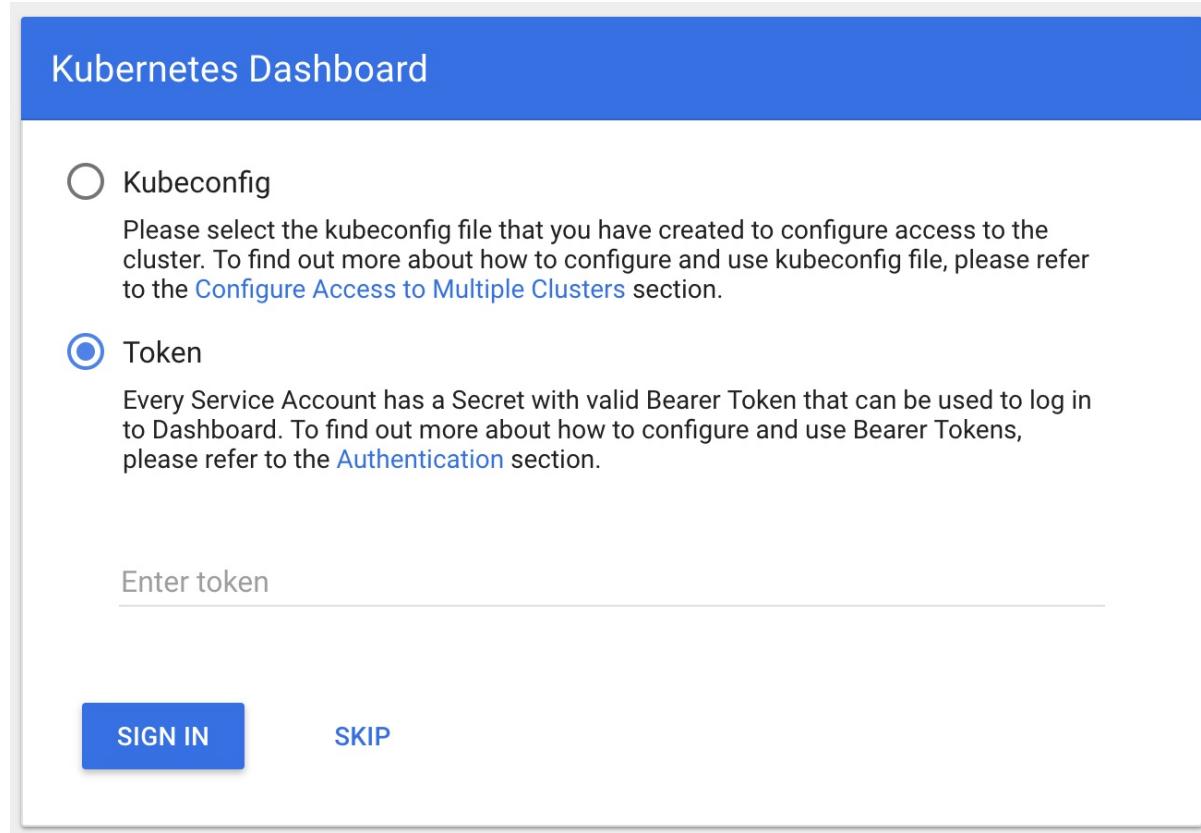
然后我们可以查看 dashboard 的外网访问端口：

```
$ kubectl get svc kubernetes-dashboard -n kube-system
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP     PORT(S)        AGE
haproxy        ClusterIP  10.254.125.90 <none>        8440/TCP,8442/TCP   2d
kubernetes-dashboard  NodePort   10.254.122.185  <none>        443:31694/TCP    10s
```

然后直接访问集群中的任何一个节点 IP 加上上面的 31694 端口即可打开 dashboard 页面了

由于 dashboard 默认是自建的 https 证书，该证书是不受浏览器信任的，所以我们需要强制跳转就可以了。

默认 dashboard 会跳转到登录页面，我们可以看到 dashboard 提供了 Kubeconfig 和 token 两种登录方式，我们可以直接跳过或者使用本地的 Kubeconfig 文件进行登录，可以看到会跳转到如下页面：



这是由于该用户没有对 default 命名空间的访问权限。

身份认证

登录 dashboard 的时候支持 Kubeconfig 和token 两种认证方式，Kubeconfig 中也依赖token 字段，所以生成token 这一步是必不可少的。

生成token

我们创建一个admin用户并授予admin 角色绑定，使用下面的yaml文件创建admin用户并赋予他管理员权限，然后就可以通过token 登陆dashbaord，这种认证方式本质实际上是通过Service Account 的身份认证加上Bearer token请求 API server 的方式实现，参考 [Kubernetes 中的认证](#)。

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: admin
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
subjects:
```

```

- kind: ServiceAccount
  name: admin
  namespace: kube-system

  ...
  apiVersion: v1
  kind: ServiceAccount
  metadata:
    name: admin
    namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile

```

上面的 admin 用户创建完成后我们就可以获取到该用户对应的 token 了，如下命令：

```

$ kubectl get secret -n kube-system|grep admin-token
admin-token-d5jsg          kubernetes.io/service-account-token   3      1d
$ kubectl get secret admin-token-d5jsg -o jsonpath={.data.token} -n kube-system |base64 -d
# 会生成一串很长的base64后的字符串

```

然后在 dashboard 登录页面上直接使用上面得到的 token 字符串即可登录，这样就可以拥有管理员权限操作整个 kubernetes 集群的对象，当然你也可以为你的登录用户新建一个指定操作权限的用户。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:00:45

18. YAML 文件

在前面的课程中，我们在安装 kubernetes 集群的时候使用了一些 YAML 文件来创建相关的资源，但是很多同学对 YAML 文件还是非常陌生。所以我们先来简单看一看 YAML 文件是如何工作的，并使用 YAML 文件来定义一个 kubernetes pod，然后再来定义一个 kubernetes deployment吧。

YAML 基础

它的基本语法规则如下：

- 大小写敏感
- 使用缩进表示层级关系
- 缩进时不允许使用Tab键，只允许使用空格。
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可
- `#` 表示注释，从这个字符一直到行尾，都会被解析器忽略。

在我们的 kubernetes 中，你只需要两种结构类型就行了：

- Lists
- Maps

也就是说，你可能会遇到 Lists 的 Maps 和 Maps 的 Lists，等等。不过不用担心，你只要掌握了这两种结构也就就可以了，其他更加复杂的我们暂不讨论。

Maps

首先我们来看看 Maps，我们都知道 Map 是字典，就是一个 `key:value` 的键值对，Maps 可以让我们更加方便的去书写配置信息，例如：

```
---
apiVersion: v1
kind: Pod
```

第一行的 `---` 是分隔符，是可选的，在单一文件中，可用连续三个连字号 `---` 区分多个文件。这里我们可以看到，我们有两个键：`kind` 和 `apiVersion`，他们对应的值分别是：v1 和 Pod。上面的 YAML 文件转换成 JSON 格式的话，你肯定就容易明白了：

```
{
  "apiVersion": "v1",
  "kind": "pod"
}
```

我们在创建一个相对复杂一点的 YAML 文件，创建一个 KEY 对应的值不是字符串而是一个 Maps：

```
---
apiVersion: v1
```

```

kind: Pod
metadata:
  name: kube100-site
  labels:
    app: web

```

上面的 YAML 文件， metadata 这个 KEY 对应的值就是一个 Maps 了，而且嵌套的 labels 这个 KEY 的值又是一个 Map，你可以根据你自己的情况进行多层嵌套。

上面我们也提到了 YAML 文件的语法规则，YAML 处理器是根据行缩进来知道内容之间的嵌套性的。比如我们上面的 YAML 文件，我用了两个空格作为缩进，空格的数量并不重要，但是你得保持一致，并且至少要求一个空格（什么意思？就是你别一会缩进两个空格，一会缩进4个空格）。

我们可以看到 name 和 labels 是相同级别的缩进，所以 YAML 处理器就知道了他们属于同一个 MAP，而 app 是 labels 的值是因为 app 的缩进更大。

注意：在 YAML 文件中绝对不要使用 tab 键。

同样的，我们可以将上面的 YAML 文件转换成 JSON 文件：

```

{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "kube100-site",
    "labels": {
      "app": "web"
    }
  }
}

```

或许你对上面的 JSON 文件更熟悉，但是你不得不承认 YAML 文件的语义化程度更高吧？

Lists

Lists 就是列表，说白了就是数组，在 YAML 文件中我们可以这样定义：

```

args
- Cat
- Dog
- Fish

```

你可以有任何数量的项在列表中，每个项的定义以破折号 (-) 开头的，与父元素直接可以缩进一个空格。对应的 JSON 格式如下：

```

{
  "args": [ 'Cat', 'Dog', 'Fish' ]
}

```

当然，list 的子项也可以是 Maps，Maps 的子项也可以是 list 如下所示：

```

---
apiVersion: v1
kind: Pod
metadata:
  name: kube100-site
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: flaskapp-demo
      image: jcdemo/flaskapp
      ports:
        - containerPort: 5000

```

比如这个 YAML 文件，我们定义了一个叫 `containers` 的 List 对象，每个子项都由 `name`、`image`、`ports` 组成，每个 `ports` 都有一个 key 为 `containerPort` 的 Map 组成，同样的，我们可以转成如下 JSON 格式文件：

```

{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "name": "kube100-site",
    "labels": {
      "app": "web"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "front-end",
        "image": "nginx",
        "ports": [
          {
            "containerPort": "80"
          }
        ]
      },
      {
        "name": "flaskapp-demo",
        "image": "jcdemo/flaskapp",
        "ports": [
          {
            "containerPort": "5000"
          }
        ]
      }
    ]
  }
}

```

是不是觉得用 JSON 格式的话文件明显比 YAML 文件更复杂了呢？

使用 YAML 创建 Pod

现在我们已经对 YAML 文件有了大概的了解了，我相信你应该没有之前那么懵逼了吧？我们还是来使用 YAML 文件来创建一个 Deployment 吧。

API 说明: <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.10/>

创建 Pod

```
---
apiVersion: v1
kind: Pod
metadata:
  name: kube100-site
  labels:
    app: web
spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: flaskapp-demo
      image: jcdemo/flaskapp
      ports:
        - containerPort: 5000
```

这是我们上面定义的一个普通的 POD 文件，我们先来简单分析下文件内容：

- apiVersion, 这里它的值是 v1, 这个版本号需要根据我们安装的 kubernetes 版本和资源类型进行变化的，记住不是写死的
- kind, 这里我们创建的是一个 Pod, 当然根据你的实际情况，这里资源类型可以是 Deployment、Job、Ingress、Service 等待。
- metadata: 包含了我们定义的 Pod 的一些 meta 信息，比如名称、namespace、标签等等信息。
- spec: 包括一些 containers, storage, volumes, 或者其他 Kubernetes 需要知道的参数，以及诸如是否在容器失败时重新启动容器的属性。你可以在特定 Kubernetes API 找到完整的 Kubernetes Pod 的属性。

让我们来看一个典型的容器的定义：

```
...spec:
  containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
...
```

在这个例子中，这是一个简单的最小定义：一个名字 (front-end) , 基于 nginx 的镜像，以及容器将会监听的一个端口 (80) 。在这些当中，只有名字是非常需要的，你也可以指定一个更加复杂的属性，例如在容器启动时运行的命令，应使用的参数，工作目录，或每次实例化时是否拉取映像的新副本。以下是一些容器可选的设置属性：

- name
- image
- command

- args
- workingDir
- ports
- env
- resources
- volumeMounts
- livenessProbe
- readinessProbe
- lifecycle
- terminationMessagePath
- imagePullPolicy
- securityContext
- stdin
- stdinOnce
- tty

明白了 POD 的定义后，我们将上面创建 POD 的 YAML 文件保存成 pod.yaml，然后使用 `kubectl` 创建 POD：

```
$ kubectl create -f pod.yaml
pod "kube100-site" created
```

然后我们就可以使用我们前面比较熟悉的 `kubectl` 命令来查看 POD 的状态了：

```
$ kubectl get pods
NAME          READY     STATUS    RESTARTS   AGE
kube100-site  2/2      Running   0          1m
```

到这里我们的 POD 就创建成功了，如果你在创建过程中有任何问题，我们同样可以使用前面的 `kubectl describe` 进行排查。我们先删除上面创建的 POD：

```
$ kubectl delete -f pod.yaml
pod "kube100-site" deleted
```

创建 Deployment

现在我们可以来创建一个真正的 Deployment。在上面的例子中，我们只是单纯的创建了一个 POD 实例，但是如果这个 POD 出现了故障的话，我们的服务也就挂掉了，所以 `kubernetes` 提供了一个 Deployment 的概念，可以让 `kubernetes` 去管理一组 POD 的副本，也就是副本集，这样就可以保证一定数量的副本一直可用的，不会因为一个 POD 挂掉导致整个服务挂掉。我们可以这样定义一个 Deployment：

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kube100-site
```

```
spec:
  replicas: 2
```

注意这里的 `apiVersion` 对应的值是 `extensions/v1beta1`，当然 `kind` 要指定为 `Deployment`，因为这就是我们需要的，然后我们可以指定一些 `meta` 信息，比如名字，或者标签之类的。最后，最重要的是 `spec` 配置选项，这里我们定义需要两个副本，当然还有很多可以设置的属性，比如一个 Pod 在没有任何错误变成准备的情况下必须达到的最小秒数。我们可以在[Kubernetes v1beta1 API参考](#)中找到一个完整的 Deployment 可指定的参数列表。现在我们来定义一个完整的 Deployment 的 YAML 文档：

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: kube100-site
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: front-end
          image: nginx
          ports:
            - containerPort: 80
        - name: flaskapp-demo
          image: jcdemo/flaskapp
          ports:
            - containerPort: 5000
```

看起来是不是和我们上面的 `pod.yaml` 很类似啊，注意其中的 `template`，其实就是对 POD 对象的定义。将上面的 YAML 文件保存为 `deployment.yaml`，然后创建 Deployment：

```
$ kubectl create -f deployment.yaml
deployment "kube100-site" created
```

同样的，想要查看它的状态，我们可以检查 Deployment 的列表：

```
$ kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube100-site   2         2         2           2           2m
```

我们可以看到所有的 Pods 都已经正常运行了。

到这里我们就完成了使用 YAML 文件创建 Kubernetes Deployment 的过程，在了解了 YAML 文件的基础后，定义 YAML 文件其实已经很简单了，最主要的是要根据实际情况去定义 YAML 文件，所以查阅 Kubernetes 文档很重要。

可以使用<http://www.yamllint.com/>去检验 YAML 文件的合法性。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:00:52

静态 Pod

我们上节课给大家讲解了 YAML 文件的使用，也手动的创建了一个简单的 Pod，这节课开始我们就来深入的学习下我们的 Pod。在Kubernetes集群中除了我们经常使用到的普通的 Pod 外，还有一种特殊的 Pod，叫做 `Static Pod`，就是我们说的静态 Pod，静态 Pod 有什么特殊的地方呢？

静态 Pod 直接由特定节点上的 `kubelet` 进程来管理，不通过 master 节点上的 `apiserver`。无法与我们常用的控制器 `Deployment` 或者 `DaemonSet` 进行关联，它由 `kubelet` 进程自己来监控，当 pod 崩溃时重启该 pod，`kubectl` 也无法对他们进行健康检查。静态 pod 始终绑定在某一个 `kubelet`，并且始终运行在同一个节点上。`kubelet` 会自动为每一个静态 pod 在 Kubernetes 的 `apiserver` 上创建一个镜像 Pod (Mirror Pod)，因此我们可以在 `apiserver` 中查询到该 pod，但是不能通过 `apiserver` 进行控制（例如不能删除）。

创建静态 Pod 有两种方式：配置文件和 HTTP 两种方式

配置文件

配置文件就是放在特定目录下的标准的 JSON 或 YAML 格式的 pod 定义文件。用 `kubelet --pod-manifest-path=<the directory>` 来启动 `kubelet` 进程，`kubelet` 定期的去扫描这个目录，根据这个目录下出现或消失的 YAML/JSON 文件来创建或删除静态 pod。

比如我们在 node01 这个节点上用静态 pod 的方式来启动一个 nginx 的服务。我们登录到 node01 节点上面，可以通过下面命令找到 `kubelet` 对应的启动配置文件

```
$ systemctl status kubelet
```

配置文件路径为：

```
$ /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

打开这个文件我们可以看到其中有一条如下的环境变量配置：

```
Environment="KUBELET_SYSTEM_PODS_ARGS=--pod-manifest-path=/etc/kubernetes/manifests --allow-privileged=true"
```

所以如果我们通过 `kubeadm` 的方式来安装的集群环境，对应的 `kubelet` 已经配置了我们的静态 Pod 文件的路径，那就是 `/etc/kubernetes/manifests`，所以我们只需要在该目录下面创建一个标准的 Pod 的 JSON 或者 YAML 文件即可：

如果你的 `kubelet` 启动参数中没有配置上面的 `--pod-manifest-path` 参数的话，那么添加上这个参数然后重启 `kubelet` 即可。

```
[root@ node01 ~] $ cat <<EOF >/etc/kubernetes/manifest/static-web.yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
```

```

labels:
  app: static
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
EOF

```

通过 HTTP 创建静态 Pods

kubelet 周期地从 `--manifest-url=` 参数指定的地址下载文件，并且把它翻译成 JSON/YAML 格式的 pod 定义。此后的操作方式与 `--pod-manifest-path=` 相同，kubelet 会不时地重新下载该文件，当文件变化时对应地终止或启动静态 pod。

静态pods的动作行为

kubelet 启动时，由 `--pod-manifest-path=` or `--manifest-url=` 参数指定的目录下定义的所有 pod 都会自动创建，例如，我们示例中的 static-web。（可能要花些时间拉取nginx 镜像，耐心等待...）

```
[root@node01 ~] $ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f6d05272b57e nginx:latest "nginx" 8 minutes ago Up 8 minutes
f4_static-web-fk-node1_default_67e24ed9466ba55986d120c867395f3c_378e5f3c
```

现在我们通过 kubectl 工具可以看到这里创建了一个新的镜像 Pod：

```
[root@node01 ~] $ kubectl get pods
NAME READY STATUS RESTARTS AGE
static-web-my-node01 1/1 Running 0 2m
```

静态 pod 的标签会传递给镜像 Pod，可以用来过滤或筛选。需要注意的是，我们不能通过 API 服务器来删除静态 pod（例如，通过 kubectl 命令），kubelet 不会删除它。

```
[root@node01 ~] $ kubectl delete pod static-web-my-node01
[root@node01 ~] $ kubectl get pods
NAME READY STATUS RESTARTS AGE
static-web-my-node01 1/1 Running 0 12s
```

我们尝试手动终止容器，可以看到 kubelet 很快就会自动重启容器。

```
[root@node01 ~] $ docker ps
CONTAINER ID IMAGE COMMAND CREATED ...
5b920cbaf8b1 nginx:latest "nginx -g 'daemon off'" 2 seconds ago ...
```

静态pods的动态增加和删除

运行中的kubelet周期扫描配置的目录（我们这个例子中就是/etc/kubernetes/manifests）下文件的变化，当这个目录中有文件出现或消失时创建或删除pods。

```
[root@node01 ~] $ mv /etc/kubernetes/manifests/static-web.yaml /tmp
[root@node01 ~] $ sleep 20
[root@node01 ~] $ docker ps
// no nginx container is running
[root@node01 ~] $ mv /tmp/static-web.yaml /etc/kubernetes/manifests
[root@node01 ~] $ sleep 20
[root@node01 ~] $ docker ps
CONTAINER ID        IMAGE             COMMAND           CREATED          ...
e7a62e3427f1        nginx:latest     "nginx -g 'daemon of"  27 seconds ago
```

其实我们用 kubeadm 安装的集群，master 节点上面的几个重要组件都是用静态 Pod 的方式运行的，我们登录到 master 节点上查看 /etc/kubernetes/manifests 目录：

```
[root@master ~]# ls /etc/kubernetes/manifests/
etcd.yaml  kube-apiserver.yaml  kube-controller-manager.yaml  kube-scheduler.yaml
```

现在明白了吧，这种方式也为我们将集群的一些组件容器化提供了可能，因为这些 Pod 都不会受到 apiserver 的控制，不然我们这里 kube-apiserver 怎么自己去控制自己呢？万一不小心把这个 Pod 删掉了呢？所以只能有 kubelet 自己来进行控制，这就是我们所说的静态 Pod。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:00:57

Pod Hook

我们知道 Pod 是 Kubernetes 集群中的最小单元，而 Pod 是有容器组组成的，所以在讨论 Pod 的生命周期的时候我们可以先来讨论下容器的生命周期。

实际上 Kubernetes 为我们的容器提供了生命周期钩子的，就是我们说的 Pod Hook，Pod Hook 是由 kubelet 发起的，当容器中的进程启动前或者容器中的进程终止之前运行，这是包含在容器的生命周期之中。我们可以同时为 Pod 中的所有容器都配置 hook。

Kubernetes 为我们提供了两种钩子函数：

- PostStart: 这个钩子在容器创建后立即执行。但是，并不能保证钩子将在容器 ENTRYPOINT 之前运行，因为没有参数传递给处理程序。主要用于资源部署、环境准备等。不过需要注意的是如果钩子花费太长时间以至于不能运行或者挂起，容器将不能达到 running 状态。
- PreStop: 这个钩子在容器终止之前立即被调用。它是阻塞的，意味着它是同步的，所以它必须在删除容器的调用发出之前完成。主要用于优雅关闭应用程序、通知其他系统等。如果钩子在执行期间挂起，Pod阶段将停留在 running 状态并且永不会达到 failed 状态。

如果 PostStart 或者 PreStop 钩子失败，它会杀死容器。所以我们应该让钩子函数尽可能的轻量。当然有些情况下，长时间运行命令是合理的，比如在停止容器之前预先保存状态。

另外我们有两种方式来实现上面的钩子函数：

- Exec - 用于执行一段特定的命令，不过要注意的是该命令消耗的资源会被计入容器。
- HTTP - 对容器上的特定的端点执行 HTTP 请求。

示例1 环境准备

以下示例中，定义了一个Nginx Pod，其中设置了 PostStart 钩子函数，即在容器创建成功后，写入一句话到 /usr/share/message 文件中。

```
apiVersion: v1
kind: Pod
metadata:
  name: hook-demo1
spec:
  containers:
  - name: hook-demo1
    image: nginx
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr/share/m
essage"]
```

示例2 优雅删除资源对象

当用户请求删除含有 pod 的资源对象时（如Deployment等），K8S 为了让应用程序优雅关闭（即让应用程序完成正在处理的请求后，再关闭软件），K8S提供两种信息通知：

- 默认：K8S 通知 node 执行 `docker stop` 命令，docker 会先向容器中 PID 为1的进程发送系统信号 `SIGTERM`，然后等待容器中的应用程序终止执行，如果等待时间达到设定的超时时间，或者默认超时时间（30s），会继续发送 `SIGKILL` 的系统信号强行 kill 掉进程。
- 使用 pod 生命周期（利用 `PreStop` 回调函数），它执行在发送终止信号之前。

默认所有的优雅退出时间都在30秒内。`kubectl delete` 命令支持 `--grace-period=<seconds>` 选项，这个选项允许用户用他们自己指定的值覆盖默认值。值'0'代表强制删除 pod. 在 kubectl 1.5 及以上的版本里，执行强制删除时必须同时指定 `--force --grace-period=0` 。

强制删除一个 pod 是从集群状态还有 etcd 里立刻删除这个 pod。当 Pod 被强制删除时，api 服务器不会等待来自 Pod 所在节点上的 kubelet 的确认信息：pod 已经被终止。在 API 里 pod 会被立刻删除，在节点上，pods 被设置成立刻终止后，在强行杀掉前还会有一个很小的宽限期。

以下示例中，定义了一个Nginx Pod，其中设置了 `PreStop` 钩子函数，即在容器退出之前，优雅的关闭 Nginx:

```

apiVersion: v1
kind: Pod
metadata:
  name: hook-demo
spec:
  containers:
    - name: hook-demo2
      image: nginx
      lifecycle:
        preStop:
          exec:
            command: ["/usr/sbin/nginx", "-s", "quit"]

---
apiVersion: v1
kind: Pod
metadata:
  name: hook-demo2
  labels:
    app: hook
spec:
  containers:
    - name: hook-demo2
      image: nginx
      ports:
        - name: webport
          containerPort: 80
      volumeMounts:
        - name: message
          mountPath: /usr/share/
      lifecycle:
        preStop:
          exec:
            command: ['/bin/sh', '-c', 'echo Hello from the preStop Handler > /usr/share/message']
  volumes:

```

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

另外 Hook 调用的日志没有暴露个给 Pod 的 event，所以只能通过 describe 命令来获取，如果有错误将可以看到 FailedPostStartHook 或 FailedPreStopHook 这样的 event。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

21. 健康检查

上节课我们和大家一起学习了 Pod 中容器的生命周期的两个钩子函数， PostStart 与 PreStop ，其中 PostStart 是在容器创建后立即执行的，而 PreStop 这个钩子函数则是在容器终止之前执行的。除了上面这两个钩子函数以外，还有一项配置会影响到容器的生命周期的，那就是健康检查的探针。

在 Kubernetes 集群当中，我们可以通过配置 liveness probe （存活探针）和 readiness probe （可读性探针）来影响容器的生存周期。

* kubelet 通过使用 liveness probe 来确定你的应用程序是否正在运行，通俗点将就是是否还活着。一般来说，如果你的程序一旦崩溃了， Kubernetes 就会立刻知道这个程序已经终止了，然后就会重启这个程序。而我们的 liveness probe 的目的就是来捕获到当前应用程序还没有终止，还没有崩溃，如果出现了这些情况，那么就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。

* kubelet 使用 readiness probe 来确定容器是否已经就绪可以接收流量过来了。这个探针通俗点讲就是说是否准备好了，现在可以开始工作了。只有当 Pod 中的容器都处于就绪状态的时候 kubelet 才会认定该 Pod 处于就绪状态，因为一个 Pod 下面可能会有多个容器。当然 Pod 如果处于非就绪状态，那么我们就会将他从我们的工作队列(实际上就是我们后面需要重点学习的 Service)中移除出来，这样我们的流量就不会被路由到这个 Pod 里面来了。

和前面的钩子函数一样的，我们这两个探针的支持两种配置方式：

```
* exec: 执行一段命令
* http: 检测某个 http 请求
* tcpSocket: 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。实际上就是检查端口
```

好，我们先来给大家演示下存活探针的使用方法，首先我们用 exec 执行命令的方式来检测容器的存活，如下：

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

我们这里需要用到一个新的属性：`livenessProbe`，下面通过 `exec` 执行一段命令，其中 `periodSeconds` 属性表示让 `kubelet` 每隔5秒执行一次存活探针，也就是每5秒执行一次上面的 `cat /tmp/healthy` 命令，如果命令执行成功了，将返回0，那么 `kubelet` 就会认为当前这个容器是存活的并且很监控，如果返回的是非0值，那么 `kubelet` 就会把该容器杀掉然后重启它。另外一个属性 `initialDelaySeconds` 表示在第一次执行探针的时候要等待5秒，这样能够确保我们的容器能够有足够的启动时间。大家可以想象下，如果你的第一次执行探针等候的时间太短，是不是很有可能容器还没正常启动起来，所以存活探针很可能始终都是失败的，这样就会无休止的重启下去了，对吧？所以一个合理的 `initialDelaySeconds` 非常重要。

另外我们在容器启动的时候，执行了如下命令：

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

意思是说在容器最开始的30秒内有一个 `/tmp/healthy` 文件，在这30秒内执行 `cat /tmp/healthy` 命令都会返回一个成功的返回码。30秒后，我们删除这个文件，现在执行 `cat /tmp/healthy` 是不是就会失败了，这个时候就会重启容器了。

我们来创建下该 Pod，在30秒内，查看 Pod 的 Event：

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

我们可以观察到容器是正常启动的，在隔一会儿，比如40s后，再查看下 Pod 的 Event，在最下面有一条信息显示 `liveness probe` 失败了，容器被删掉并重新创建。

然后通过 `kubectl get pod liveness-exec` 可以看到 `RESTARTS` 值加1了。

同样的，我们还可以使用 `HTTP GET` 请求来配置我们的存活探针，我们这里使用一个 `liveness` 镜像来验证演示下，

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

同样的，根据 `periodSeconds` 属性我们可以知道 `kubelet` 需要每隔3秒执行一次 `liveness probe`，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，`kubelet` 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，`kubelet` 将杀掉该容器并重启它。。`initialDelaySeconds` 指定 `kubelet` 在该执行第一次探测之前需要等待3秒钟。

通常来说，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。

我们可以来查看下上面的 `healthz` 的实现：

```
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"))
    }
})
```

大概意思就是最开始前10s返回状态码200，10s过后就返回500的 `status_code` 了。所以当容器启动3秒后，`kubelet` 开始执行健康检查。第一次健康监测会成功，因为是在10s之内，但是10秒后，健康检查将失败，因为现在返回的是一个错误的状态码了，所以 `kubelet` 将会杀掉和重启容器。

同样的，我们来创建下该 Pod 测试下效果，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
```

然后我们来通过端口的方式来配置存活探针，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: cnych/goproxy
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
```

```

tcpSocket:
  port: 8080
initialDelaySeconds: 15
periodSeconds: 20

```

我们可以看到，TCP 检查的配置与 HTTP 检查非常相似，只是将 `httpGet` 替换成了 `tcpSocket`。而且我们同时使用了 `readiness probe` 和 `liveness probe` 两种探针。容器启动后5秒后，`kubelet` 将发送第一个 `readiness probe`（可读性探针）。该探针会去连接容器的8080端，如果连接成功，则该 Pod 将被标记为就绪状态。然后 `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，这将尝试去连接到容器的8080端口。如果 `liveness probe` 失败，容器将重新启动。

有的时候，应用程序可能暂时无法对外提供服务，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，您不想杀死应用程序，也不想对外提供服务。那么这个时候我们就可以使用 `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。

从上面的 YAML 文件我们可以看出 `readiness probe` 的配置跟 `liveness probe` 很像，基本上一致的。唯一的不同是使用 `readinessProbe` 而不是 `livenessProbe`。两者如果同时使用的话就可以确保流量不会到达还未准备好的容器，准备好过后，如果应用程序出现了错误，则会重新启动容器。

另外除了上面的 `initialDelaySeconds` 和 `periodSeconds` 属性外，探针还可以配置如下几个参数：

```

* 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，但是如果`liveness`则必须是 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。

```

这就是 `liveness probe`（存活探针）和 `readiness probe`（可读性探针）的使用方法。在 Pod 的生命周期当中，我们已经学习了容器生命周期中的钩子函数和探针检测，下节课给大家讲解 Pod 层面生命周期的一个阶段：初始化容器。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:01:14

22. 初始化容器

上节课我们学习了容器的健康检查的两个探针： `liveness probe`（存活探针）和 `readiness probe`（可读性探针）的使用方法，我们说在这两个探针是可以影响容器的生命周期的，包括我们之前提到的容器的两个钩子函数 `PostStart` 和 `PreStop`。我们今天要给大家介绍的是 `Init Container`（初始化容器）。

`Init Container` 就是用来做初始化工作的容器，可以是一个或者多个，如果有多个的话，这些容器会按定义的顺序依次执行，只有所有的 `Init Container` 执行完后，主容器才会被启动。我们知道一个 `Pod` 里面的所有容器是共享数据卷和网络命名空间的，所以 `Init Container` 里面产生的数据可以被主容器使用到的。

是不是感觉 `Init Container` 和之前的钩子函数有点类似啊，只是是在容器执行前来做一些工作，是吧？从直观的角度看上去的话，初始化容器的确有点像 `PreStart`，但是钩子函数和我们的 `Init Container` 是处在不同的阶段的，我们可以通过下面的图来了解下：



从上面这张图我们可以直观的看到 `PostStart` 和 `PreStop` 包括 `liveness` 和 `readiness` 是属于主容器的生命周期范围内的，而 `Init Container` 是独立于主容器之外的，当然他们都属于 `Pod` 的生命周期范畴之内的，现在我们应该明白 `Init Container` 和钩子函数之类的区别了吧。

另外我们可以看到上面我们的 Pod 右边还有一个 `infra` 的容器，这是一个什么容器呢？我们可以在集群环境中去查看下人任意一个 Pod 对应的运行的 Docker 容器，我们可以发现每一个 Pod 下面都包含了一个 `pause-amd64` 的镜像，这个就是我们的 `infra` 镜像，我们知道 Pod 下面的所有容器是共享同一个网络命名空间的，这个镜像就是来做这个事情的，所以每一个 Pod 当中都会包含一个这个镜像。

很多同学最开始 Pod 启动不起来就是因为这个 `infra` 镜像没有被拉下来，因为默认该镜像是需要到谷歌服务器上拉取的，所以需要提前拉取到节点上面。

我们说 `Init Container` 主要是来做初始化容器工作的，那么他有哪些应用场景呢？

- 等待其他模块Ready：这个可以用来解决服务之间的依赖问题，比如我们有一个 Web 服务，该服务又依赖于另外一个数据库服务，但是在我们启动这个 Web 服务的时候我们并不能保证依赖的这个数据库服务就已经启动起来了，所以可能会出现一段时间内 Web 服务连接数据库异常。要解决这个问题的话我们就可以在 Web 服务的 Pod 中使用一个 `InitContainer`，在这个初始化容器中去检查数据库是否已经准备好了，准备好了过后初始化容器就结束退出，然后我们的主容器 Web 服务被启动起来，这个时候去连接数据库就不会有问题了。
- 做初始化配置：比如集群里检测所有已经存在的成员节点，为主容器准备好集群的配置信息，这样主容器起来后就能用这个配置信息加入集群。
- 其它场景：如将 pod 注册到一个中央数据库、配置中心等。

我们先来给大家演示下服务依赖的场景下初始化容器的使用方法，如下 Pod 的定义方法

```
apiVersion: v1
kind: Pod
metadata:
  name: init-pod1
  labels:
    app: init
spec:
  containers:
    - name: init-container
      image: busybox
      command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
    - name: init-myservice
      image: busybox
      command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
    - name: init-mydb
      image: busybox
      command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

Service 的对应 YAML 内容：

```
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 6376
```

```

---  

kind: Service  

apiVersion: v1  

metadata:  

  name: mydb  

spec:  

  ports:  

  - protocol: TCP  

    port: 80  

    targetPort: 6377

```

我们可以先创建上面的 Pod，然后查看下 Pod 的状态，然后再创建下面的 Service，对比下前后状态。

我们在 Pod 启动过程中，初始化容器会按顺序在网络和数据卷初始化之后启动。每个容器必须在下一个容器启动之前成功退出。如果由于运行时或失败退出，导致容器启动失败，它会根据 Pod 的 restartPolicy 指定的策略进行重试。然而，如果 Pod 的 restartPolicy 设置为 Always，Init 容器失败时会使用 RestartPolicy 策略。

在所有的初始化容器没有成功之前，Pod 将不会变成 Ready 状态。正在初始化中的 Pod 处于 Pending 状态，但应该会将条件 Initializing 设置为 true。

接下来我们再来尝试创建一个做初始化配置工作的 Pod：

```

apiVersion: v1  

kind: Pod  

metadata:  

  name: init-demo  

spec:  

  containers:  

  - name: nginx  

    image: nginx  

    ports:  

    - containerPort: 80  

    volumeMounts:  

    - name: workdir  

      mountPath: /usr/share/nginx/html  

  initContainers:  

  - name: install  

    image: busybox  

    command:  

    - wget  

    - "-O"  

    - "/work-dir/index.html"  

    - http://www.baidu.com  

    volumeMounts:  

    - name: workdir  

      mountPath: "/work-dir"  

  volumes:  

  - name: workdir  

    emptyDir: {}

```

我们可以看到这里又出现了 `volumes`，`spec.volumes` 指的是 Pod 中的卷，`spec.containers.volumeMounts`，是将指定的卷 mount 到容器指定的位置，相当于 docker 里面的 `-v` 宿主机目录：容器目录，我们前面用到过 `hostPath`，我们这里使用的是 `emptyDir{}`，这个就相当于一个共享卷，是一个临时的目录，生命周期等同于 Pod 的生命周期。

初始化容器执行完，会下载一个 html 文件映射到 `emptyDir{}`，而主容器也是和 `spec.volumes` 里的 `emptyDir{}` 进行映射，所以 `nginx` 容器的 `/usr/share/nginx/html` 目录下会映射 `index.html` 文件。

我们来创建下该 Pod，然后验证 `nginx` 容器是否运行：

```
$ kubectl get pod init-demo
```

输出显示了 `nginx` 容器正在运行：

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	43m

在 `init-demo` 容器里的 `nginx` 容器打开一个 shell：

```
$ kubectl exec -it init-demo -- /bin/bash
```

在 Shell 里，直接查看下 `index.html` 的内容：

```
root@nginx:~# cat /usr/share/nginx/html/index.html
```

如果我们看到有百度相关的信息那么证明我们上面的初始化的工作就完成了。

这就是我们初始化容器的使用方法，到这里我们就把 Pod 的整个生命周期当中的几个主要阶段讲完了，第一个是容器的两个钩子函数：`PostStart` 和 `PreStop`，还有就是容器健康检查的两个探针：`liveness probe` 和 `readiness probe`，以及这节课的 `Init Container`。下节课开始我们来讲解一些常用的控制器和 Pod 的结合。``

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:01:20

使用Replication Controller、Replica Set 管理Pod

前面我们的课程中学习了 Pod 的一些基本使用方法，而且前面我们都是直接来操作的 Pod，假如我们现在有一个 Pod 正在提供线上的服务，我们来想想一下我们可能会遇到的一些场景：

- 某次运营活动非常成功，网站访问量突然暴增
- 运行当前 Pod 的节点发生故障了，Pod 不能正常提供服务了

第一种情况，可能比较好应对，一般活动之前我们会大概计算下会有多大的访问量，提前多启动几个 Pod，活动结束后再把多余的 Pod 杀掉，虽然有点麻烦，但是应该还是能够应对这种情况的。

第二种情况，可能某天夜里收到大量报警说服务挂了，然后起来打开电脑在另外的节点上重新启动一个新的 Pod，问题也很好的解决了。

如果我们都人工的去解决遇到的这些问题，似乎又回到了以前刀耕火种的时代了是吧，如果有一种工具能够来帮助我们管理 Pod 就好了，Pod 不够了自动帮我新增一个，Pod 挂了自动帮我在合适的节点上重新启动一个 Pod，这样是不是遇到上面的问题我们都不需要手动去解决了。

幸运的是，Kubernetes 就为我们提供了这样的资源对象：

- Replication Controller：用来部署、升级 Pod
- Replica Set：下一代的 Replication Controller
- Deployment：可以更加方便的管理 Pod 和 Replica Set

Replication Controller (RC)

Replication Controller 简称 RC，RC 是 Kubernetes 系统中的核心概念之一，简单来说，RC 可以保证在任意时间运行 Pod 的副本数量，能够保证 Pod 总是可用的。如果实际 Pod 数量比指定的多那就结束掉多余的，如果实际数量比指定的少就新启动一些 Pod，当 Pod 失败、被删除或者挂掉后，RC 都会去自动创建新的 Pod 来保证副本数量，所以即使只有一个 Pod，我们也应该使用 RC 来管理我们的 Pod。

我们想想如果现在我们遇到上面的问题的话，可能除了第一个不能做到完全自动化，其余的我们是不是都不用担心了，运行 Pod 的节点挂了，RC 检测到 Pod 失败了，就会去合适的节点重新启动一个 Pod 就行，不需要我们手动去新建一个 Pod 了。如果是第一种情况的话在活动开始之前我们给 Pod 指定10个副本，结束后将副本数量改成2，这样是不是也远比我们手动去启动、手动去关闭要好得多，而且我们后面还会给大家介绍另外一种资源对象 HPA 可以根据资源的使用情况进行自动扩缩容，这样以后遇到这种情况，我们就真的可以安心的去睡觉了。

现在我们来使用 RC 来管理我们前面使用的 Nginx 的 Pod，YAML 文件如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: rc-demo
  labels:
    name: rc
spec:
```

```

replicas: 3
selector:
  name: rc
template:
  metadata:
    labels:
      name: rc
spec:
  containers:
  - name: nginx-demo
    image: nginx
    ports:
    - containerPort: 80

```

上面的 YAML 文件相对于我们之前的 Pod 的格式：

- kind: ReplicationController
- spec.replicas: 指定 Pod 副本数量， 默认为1
- spec.selector: RC 通过该属性来筛选要控制的 Pod
- spec.template: 这里就是我们之前的 Pod 的定义的模块，但是不需要 apiVersion 和 kind 了
- spec.template.metadata.labels: 注意这里的 Pod 的 labels 要和 spec.selector 相同，这样 RC 就可以来控制当前这个 Pod 了。

这个 YAML 文件中的意思就是定义了一个 RC 资源对象，它的名字叫 rc-demo，保证一直会有3个 Pod 运行，Pod 的镜像是 nginx 镜像。

注意 spec.selector 和 spec.template.metadata.labels 这两个字段必须相同，否则会创建失败的，当然我们也可以不写 spec.selector，这样就默认与 Pod 模板中的 metadata.labels 相同了。所以为了避免不必要的错误的话，不写为好。

然后我们来创建上面的 RC 对象(保存为 rc-demo.yaml):

```
$ kubectl create -f rc-demo.yaml
```

查看 RC：

```
$ kubectl get rc
```

查看具体信息：

```
$ kubectl describe rc rc-demo
```

然后我们通过 RC 来修改下 Pod 的副本数量为2:

```
$ kubectl apply -f rc-demo.yaml
```

或者

```
$ kubectl edit rc rc-demo
```

而且我们还可以用 `RC` 来进行滚动升级，比如我们将镜像地址更改为 `nginx:1.7.9`：

```
$ kubectl rolling-update rc-demo --image=nginx:1.7.9
```

但是如果我们的 `Pod` 中多个容器的话，就需要通过修改 `YAML` 文件来进行修改了：

```
$ kubectl rolling-update rc-demo -f rc-demo.yaml
```

如果升级完成后出现了新的问题，想要一键回滚到上一个版本的话，使用 `RC` 只能用同样的方法把镜像地址替换成之前的，然后重新滚动升级。

Replication Set (RS)

`Replication Set` 简称 `RS`，随着 `Kubernetes` 的高速发展，官方已经推荐我们使用 `RS` 和 `Deployment` 来代替 `RC` 了，实际上 `RS` 和 `RC` 的功能基本一致，目前唯一的一个区别就是 `RC` 只支持基于等式的 `selector` (`env=dev` 或 `environment!=qa`)，但 `RS` 还支持基于集合的 `selector` (`version in (v1.0, v2.0)`)，这对复杂的运维管理就非常方便了。

`kubectl` 命令行工具中关于 `RC` 的大部分命令同样适用于我们的 `RS` 资源对象。不过我们也很少会去单独使用 `RS`，它主要被 `Deployment` 这个更加高层的资源对象使用，除非用户需要自定义升级功能或根本不需要升级 `Pod`，在一般情况下，我们推荐使用 `Deployment` 而不直接使用 `Replica Set`。

最后我们总结下关于 `RC / RS` 的一些特性和作用吧：

- 大部分情况下，我们可以通过定义一个 `RC` 实现的 `Pod` 的创建和副本数量的控制
- `RC` 中包含一个完整的 `Pod` 定义模块（不包含 `apiVersion` 和 `kind`）
- `RC` 是通过 `label selector` 机制来实现对 `Pod` 副本的控制的
- 通过改变 `RC` 里面的 `Pod` 副本数量，可以实现 `Pod` 的扩缩容功能
- 通过改变 `RC` 里面的 `Pod` 模板中镜像版本，可以实现 `Pod` 的滚动升级功能（但是不支持一键回滚，需要用相同的方法去修改镜像地址）

好，这节课我们就给大家介绍了使用 `RC` 或者 `RS` 来管理我们的 `Pod`，我们下节课来给大家介绍另外一种更加高级也是现在推荐使用的一个资源对象 `Deployment`。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:01:24

Deployment的使用

前面的课程中我们学习了 Replication Controller 和 Replica Set 两种资源对象， RC 和 RS 的功能基本上是差不多的，唯一的区别就是 RS 支持集合的 selector 。我们也学习到了用 RC / RS 来控制 Pod 副本的数量，也实现了滚动升级 Pod 的功能。现在看上去似乎一切都比较完美的运行着，但是我们上节课最后也提到了现在我们推荐使用 Deployment 这种控制器了，而不是我们之前的 RC 或者 RS ，这是为什么呢？

没有对比就没有伤害对吧，我们来对比下二者之间的功能吧，首先 RC 是 Kubernetes 的一个核心概念，当我们把应用部署到集群之后，需要保证应用能够持续稳定的运行， RC 就是这个保证的关键，主要功能如下：

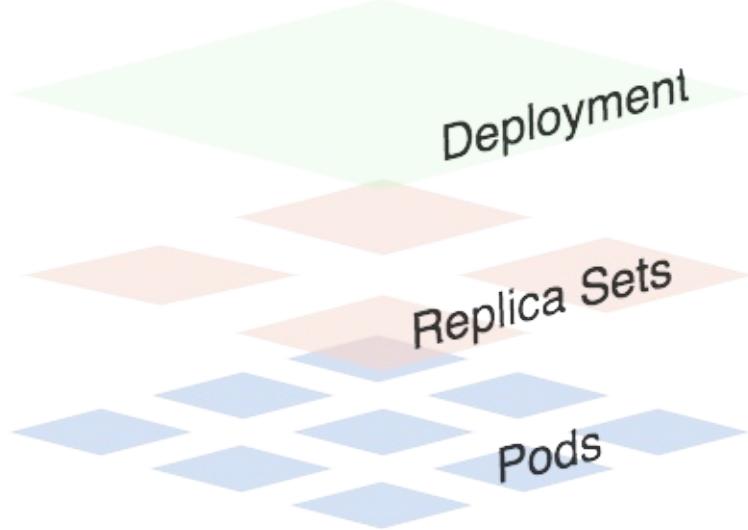
- 确保 Pod 数量：它会确保 Kubernetes 中有指定数量的 Pod 在运行，如果少于指定数量的 Pod ， RC 就会创建新的，反之会删除多余的，保证 Pod 的副本数量不变。
- 确保 Pod 健康：当 Pod 不健康，比如运行出错了，总之无法提供正常服务时， RC 也会杀死不健康的 Pod ，重新创建新的。
- 弹性伸缩：在业务高峰或者低峰的时候，可以用过 RC 来动态的调整 Pod 数量来提供资源的利用率，当然我们也提到过如果使用 HPA 这种资源对象的话可以做到自动伸缩。
- 滚动升级：滚动升级是一种平滑的升级方式，通过逐步替换的策略，保证整体系统的稳定性，这个我们上节课已经给大家演示过了。

Deployment 同样也是 Kubernetes 系统的一个核心概念，主要职责和 RC 一样的都是保证 Pod 的数量和健康，二者大部分功能都是完全一致的，我们可以看成是一个升级版的 RC 控制器，那 Deployment 又具备那些新特性呢？

- RC 的全部功能： Deployment 具备上面描述的 RC 的全部功能
- 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 的时候如果出现问题，可以使用回滚操作回滚到之前的任一版本
- 版本记录：每一次对 Deployment 的操作，都能够保存下来，这也是保证可以回滚到任一版本的基础
- 暂停和启动：对于每一次升级都能够随时暂停和启动

作为对比，我们知道 Deployment 作为新一代的 RC ，不仅在功能上更为丰富了，同时我们也说过现在官方也都是推荐使用 Deployment 来管理 Pod 的，比如一些官方组件 kube-dns 、 kube-proxy 也都是使用的 Deployment 来管理的，所以当大家在使用的使用也最好使用 Deployment 来管理 Pod 。

创建



可以看出一个Deployment拥有多个 Replica Set，而一个Replica Set拥有一个或多个Pod。一个Deployment控制多个rs主要是为了支持回滚机制，每当Deployment操作时，Kubernetes会重新生成一个Replica Set并保留，以后有需要的话就可以回滚至之前的状态。下面创建一个Deployment，它创建了一个Replica Set来启动3个nginx pod，yaml文件如下：

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deploy
  labels:
    k8s-app: nginx-demo
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80

```

将上面内容保存为：nginx-deployment.yaml，执行命令：

```
$ kubectl create -f nginx-deployment.yaml
deployment "nginx-deploy" created
```

然后执行一下命令查看刚刚创建的Deployment：

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

隔一会再次执行上面命令：

```
$ kubectl get deployments
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
nginx-deploy  3        3        3           3          4m
```

我们可以看到Deployment已经创建了1个Replica Set了，执行下面的命令查看rs和pod:

```
$ kubectl get rs
NAME      DESIRED  CURRENT  READY  AGE
nginx-deploy-431080787  3        3        3      6m
```

```
$ kubectl get pod --show-labels
NAME          READY  STATUS  RESTARTS  AGE  LABELS
nginx-deploy-431080787-53z8q  1/1   Running  0          7m  app=nginx, pod-temp
late-hash=431080787
nginx-deploy-431080787-bhhq0  1/1   Running  0          7m  app=nginx, pod-temp
late-hash=431080787
nginx-deploy-431080787-sr44p  1/1   Running  0          7m  app=nginx, pod-temp
late-hash=431080787
```

上面的Deployment的yaml文件中的 `replicas:3` 将会保证我们始终有3个POD在运行。

由于 `Deployment` 和 `RC` 的功能大部分都一样的，我们上节课已经和大家演示了大部分功能了，我们这里重点给大家演示下 `Deployment` 的滚动升级和回滚功能。

滚动升级

现在我们将刚刚保存的yaml文件中的nginx镜像修改为 `nginx:1.13.3`，然后在spec下面添加滚动升级策略：

```
minReadySeconds: 5
strategy:
  # indicate which strategy we want for rolling update
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 1
```

- `minReadySeconds`:
 - Kubernetes在等待设置的时间后才进行升级
 - 如果没有设置该值，Kubernetes会假设该容器启动起来后就提供服务了
 - 如果没有设置该值，在某些极端情况下可能会造成服务正常运行
- `maxSurge`:
 - 升级过程中最多可以比原先设置多出的POD数量
 - 例如：`maxSurge=1, replicas=5`, 则表示Kubernetes会先启动1一个新的Pod后才删掉一个旧的POD，整个升级过程中最多会有5+1个POD。
- `maxUnavailable`:
 - 升级过程中最多有多少个POD处于无法提供服务的状态
 - 当 `maxSurge` 不为0时，该值也不能为0

- 例如：maxUnavailable=1， 则表示Kubernetes整个升级过程中最多会有1个POD处于无法服务的状态。

然后执行命令：

```
$ kubectl apply -f nginx-deployment.yaml
deployment "nginx-deploy" configured
```

然后我们可以使用 `rollout` 命令：

- 查看状态：

```
$ kubectl rollout status deployment/nginx-deploy
Waiting for rollout to finish: 1 out of 3 new replicas have been updated...
deployment "nginx-deploy" successfully rolled out
```

- 暂停升级

```
$ kubectl rollout pause deployment <deployment>
```

- 继续升级

```
$ kubectl rollout resume deployment <deployment>
```

升级结束后，继续查看rs的状态：

```
$ kubectl get rs
NAME           DESIRED   CURRENT   READY   AGE
nginx-deploy-2078889897    0          0          0      47m
nginx-deploy-3297445372    3          3          3      42m
nginx-deploy-431080787    0          0          0      1h
```

根据AGE我们可以看到离我们最近的当前状态是：3， 和我们的yaml文件是一致的， 证明升级成功了。用 `describe` 命令可以查看升级的全部信息：

```
Name:      nginx-deploy
Namespace:  default
CreationTimestamp:  Wed, 18 Oct 2017 16:58:52 +0800
Labels:    k8s-app=nginx-demo
Annotations:  deployment.kubernetes.io/revision=3
  kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1beta1","kind":"Deployment","metadata":{"annotations":{},"labels":{"k8s-app":"nginx-demo"},"name":"nginx-deploy","namespace":"defa..."}
Selector:  app=nginx
Replicas:  3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:  RollingUpdate
MinReadySeconds:  0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels: app=nginx
  Containers:
```

```

nginx:
  Image:    nginx:1.13.3
  Port:    80/TCP
  Environment: <none>
  Mounts:   <none>
  Volumes:  <none>
Conditions:
  Type      Status  Reason
  ----      ----   -----
  Progressing  True   NewReplicaSetAvailable
  Available   True   MinimumReplicasAvailable
OldReplicaSets: <none>
NewReplicaSet:  nginx-deploy-3297445372 (3/3 replicas created)
Events:
  FirstSeen  LastSeen  Count  From            SubObjectPath  Type    Reason          Message
  -----     -----   -----  -----           -----          -----   -----          -----
  50m       50m       1     deployment-controller  nginx-deploy-2078889897 to 1        Normal  ScalingReplicaSet Scaled up replica set
  45m       45m       1     deployment-controller  nginx-deploy-2078889897 to 0        Normal  ScalingReplicaSet Scaled down replica set
  45m       45m       1     deployment-controller  nginx-deploy-3297445372 to 1        Normal  ScalingReplicaSet Scaled up replica set
  39m       39m       1     deployment-controller  nginx-deploy-431080787 to 2        Normal  ScalingReplicaSet Scaled down replica set
  39m       39m       1     deployment-controller  nginx-deploy-3297445372 to 2        Normal  ScalingReplicaSet Scaled up replica set
  38m       38m       1     deployment-controller  nginx-deploy-431080787 to 1        Normal  ScalingReplicaSet Scaled down replica set
  38m       38m       1     deployment-controller  nginx-deploy-3297445372 to 3        Normal  ScalingReplicaSet Scaled up replica set
  38m       38m       1     deployment-controller  nginx-deploy-431080787 to 0        Normal  ScalingReplicaSet Scaled down replica set

```

回滚Deployment

我们已经能够滚动平滑的升级我们的Deployment了，但是如果升级后的POD出了问题该怎么办？我们能够想到的最好最快的方式当然是回退到上一次能够提供正常工作的版本，Deployment就为我们提供了回滚机制。

首先，查看Deployment的升级历史：

```

$ kubectl rollout history deployment nginx-deploy
deployments "nginx-deploy"
REVISION  CHANGE-CAUSE
1  <none>
2  <none>
3  kubectl apply --filename=Desktop/nginx-deployment.yaml --record=true

```

从上面的结果可以看出在执行 Deployment 升级的时候最好带上 record 参数，便于我们查看历史版本信息。

默认情况下，所有通过 `kubectl xxxx --record` 都会被 `kubernetes` 记录到 `etcd` 进行持久化，这无疑会占用资源，最重要的是，时间久了，当你 `kubectl get rs` 时，会有成百上千的垃圾 `rs` 返回给你，那时你可能就眼花缭乱了。

上生产时，我们最好通过设置Deployment的 `.spec.revisionHistoryLimit` 来限制最大保留的 `revision number`，比如15个版本，回滚的时候一般只会回滚到最近的几个版本就足够了。其实 `rollout history` 中记录的 `revision` 都和 `ReplicaSets` 一一对应。如果手动 `delete` 某个 `ReplicaSet`，对应的 `rollout history` 就会被删除，也就是说你无法回滚到这个 `revision` 了。

`rollout history` 和 `ReplicaSet` 的对应关系，可以在 `kubectl describe rs $RSNAME` 返回的 `revision` 字段中得到，这里的 `revision` 就对应着 `rollout history` 返回的 `revision`。

同样我们可以使用下面的命令查看单个 `revision` 的信息：

```
$ kubectl rollout history deployment nginx-deploy --revision=3
deployments "nginx-deploy" with revision #3
Pod Template:
  Labels: app=nginx
  pod-template-hash=3297445372
  Annotations: kubernetes.io/change-cause=kubectl apply --filename=nginx-deployment.yaml
--record=true
  Containers:
    nginx:
      Image: nginx:1.13.3
      Port: 80/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
```

假如现在要直接回退到当前版本的前一个版本：

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

当然也可以用 `revision` 回退到指定的版本：

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

现在可以用命令查看Deployment现在的状态了。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



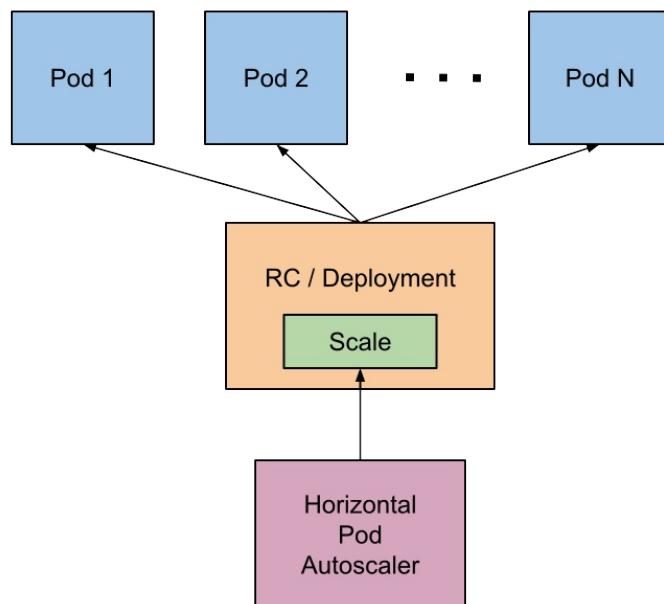
k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-05-04 22:00:00

Pod 自动扩缩容

在前面的课程中，我们提到过通过手工执行 `kubectl scale` 命令和在 Dashboard 上操作可以实现 Pod 的扩缩容，但是这样毕竟需要每次去手工操作一次，而且指不定什么时候业务请求量就很大了，所以如果不能做到自动化的去扩缩容的话，这也是一个很麻烦的事情。如果 Kubernetes 系统能够根据 Pod 当前的负载的变化情况来自动的进行扩缩容就好了，因为这个过程本来就是不固定的，频繁发生的，所以纯手工的方式不是很现实。

幸运的是 Kubernetes 为我们提供了这样一个资源对象：Horizontal Pod Autoscaling（Pod 水平自动伸缩），简称 HPA。HPA 通过监控分析 RC 或者 Deployment 控制的所有 Pod 的负载变化情况来确定是否需要调整 Pod 的副本数量，这是 HPA 最基本的原理。



hpa

HPA 在 Kubernetes 集群中被设计成一个 controller，我们可以简单的通过 `kubectl autoscale` 命令来创建一个 HPA 资源对象，HPA Controller 默认 30s 轮询一次（可通过 `kube-controller-manager` 的标志 `--horizontal-pod-autoscaler-sync-period` 进行设置），查询指定的资源（RC 或者 Deployment）中 Pod 的资源使用率，并且与创建时设定的值和指标做对比，从而实现自动伸缩的功能。

当你创建了 HPA 后， HPA 会从 Heapster 或者用户自定义的 RESTClient 端获取每一个一个 Pod 利用率或原始值的平均值，然后和 HPA 中定义的指标进行对比，同时计算出需要伸缩的具体值并进行相应的操作。目前， HPA 可以从两个地方获取数据：

- Heapster：仅支持 CPU 使用率
- 自定义监控：我们到后面的监控的课程中再给大家讲解这部分的使用方法

我们这节课来给大家介绍从 Heapster 获取监控数据来进行自动扩缩容的方法，所以首先我们得安装 Heapster，前面我们在 kubeadm 搭建集群的课程中，实际上我们已经默认把 Heapster 相关的镜像都已经拉取到节点上了，所以接下来我们只需要部署即可，我们这里使用的是 Heapster 1.4.2 版本的，前往 Heapster 的 github 页面：

<https://github.com/kubernetes/heapster>

我们将该目录下面的 yaml 文件保存到我们的集群上，然后使用 kubectl 命令行工具创建即可，另外创建完成后，如果需要在 Dashboard 当中看到监控图表，我们还需要在 Dashboard 中配置上我们的 heapster-host。

同样的，我们来创建一个 Deployment 管理的 Nginx Pod，然后利用 HPA 来进行自动扩缩容。定义 Deployment 的 YAML 文件如下：(hap-deploy-demo.yaml)

```
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: hpa-nginx-deploy
  labels:
    app: nginx-demo
spec:
  revisionHistoryLimit: 15
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

然后创建 Deployment：

```
$ kubectl create -f hap-deploy-demo.yaml
```

现在我们来创建一个 HPA，可以使用 kubectl autoscale 命令来创建：

```
$ kubectl autoscale deployment hpa-nginx-deploy --cpu-percent=10 --min=1 --max=10
deployment "hpa-nginx-deploy" autoscaled
...
$ kubectl get hpa
NAME      REFERENCE      TARGET      CURRENT      MINPODS      MAXPODS      AGE
```

hpa-nginx-deploy	Deployment/hpa-nginx-deploy	10%	0%	1	10	1
3s						

此命令创建了一个关联资源 hpa-nginx-deploy 的 HPA，最小的 pod 副本数为1，最大为10。HPA 会根据设定的 cpu 使用率（10%）动态的增加或者减少 pod 数量。

当然出来使用 `kubectl autoscale` 命令来创建外，我们依然可以通过创建 YAML 文件的形式来创建 HPA 资源对象。如果我们不知道怎么编写的话，可以查看上面命令行创建的 HPA 的 YAML 文件：

```
$ kubectl get hpa hpa-nginx-deploy -o yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  creationTimestamp: 2017-06-29T08:04:08Z
  name: nginxtest
  namespace: default
  resourceVersion: "951016361"
  selfLink: /apis/autoscaling/v1/namespaces/default/horizontalpodautoscalers/nginxtest
  uid: 86febb63-5ca1-11e7-aaef-5254004e79a3
spec:
  maxReplicas: 5 // 资源最大副本数
  minReplicas: 1 // 资源最小副本数
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment // 需要伸缩的资源类型
    name: nginxtest // 需要伸缩的资源名称
  targetCPUUtilizationPercentage: 50 // 触发伸缩的cpu使用率
status:
  currentCPUUtilizationPercentage: 48 // 当前资源下pod的cpu使用率
  currentReplicas: 1 // 当前的副本数
  desiredReplicas: 2 // 期望的副本数
  lastScaleTime: 2017-07-03T06:32:19Z
```

好，现在我们根据上面的 YAML 文件就可以自己来创建一个基于 YAML 的 HPA 描述文件了。

现在我们来增大负载进行测试，我们来创建一个 busybox，并且循环访问上面创建的服务。

```
$ kubectl run -i --tty load-generator --image=busybox /bin/sh
If you don't see a command prompt, try pressing enter.
/ # while true; do wget -q -O- http://172.16.255.60:4000; done
```

下图可以看到，HPA 已经开始工作。

```
$ kubectl get hpa
NAME      REFERENCE          TARGET     CURRENT   MINPODS   MAXPODS   AGE
hpa-nginx-deploy  Deployment/hpa-nginx-deploy  10%        29%       1         10        27m
```

同时我们查看相关资源 hpa-nginx-deploy 的副本数量，副本数量已经从原来的1变成了3。

```
$ kubectl get deployment hpa-nginx-deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hpa-nginx-deploy  3         3         3           3           4d
```

同时再次查看 HPA，由于副本数量的增加，使用率也保持在了10%左右。

```
$ kubectl get hpa
NAME          REFERENCE          TARGET          CURRENT      MINPODS      MAXPODS      AGE
hpa-nginx-deploy  Deployment/hpa-nginx-deploy  10%           9%           1           10          3
5m
```

同样的这个时候我们来关掉 busybox 来减少负载，然后等待一段时间观察下 HPA 和 Deployment 对象

```
$ kubectl get hpa
NAME          REFERENCE          TARGET          CURRENT      MINPODS      MAXPODS      AGE
hpa-nginx-deploy  Deployment/hpa-nginx-deploy  10%           0%           1           10          4
8m
$ kubectl get deployment hpa-nginx-deploy
NAME          DESIRED      CURRENT      UP-TO-DATE      AVAILABLE      AGE
hpa-nginx-deploy  1           1           1           1           4d
```

可以看到副本数量已经由3变为1。

不过当前的 HPA 只有 CPU 使用率这一个指标，还不是很灵活的，在后面的课程中我们来根据我们自定义的监控来自动对 Pod 进行扩缩容。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:01:38

Job 和 Cronjob 的使用

上节课我们学习了 Pod 自动伸缩的方法，我们使用到了 HPA 这个资源对象，我们在后面的课程中还会和大家接触到 HPA 的。今天我们来给大家介绍另外一类资源对象：Job，我们在日常的工作中经常都会遇到一些需要进行批量数据处理和分析的需求，当然也会有按时间来进行调度的工作，在我们的 Kubernetes 集群中为我们提供了 Job 和 CronJob 两种资源对象来应对我们的这种需求。

Job 负责处理任务，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。而 CronJob 则就是在 Job 上加上了时间调度。

Job

我们用 Job 这个资源对象来创建一个任务，我们定一个 Job 来执行一个倒计时的任务，定义 YAML 文件：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-demo
spec:
  template:
    metadata:
      name: job-demo
    spec:
      restartPolicy: Never
      containers:
        - name: counter
          image: busybox
          command:
            - "bin/sh"
            - "-c"
            - "for i in 9 8 7 6 5 4 3 2 1; do echo $i; done"
```

注意 Job 的 RestartPolicy 仅支持 Never 和 OnFailure 两种，不支持 Always，我们知道 Job 就相当于来执行一个批处理任务，执行完就结束了，如果支持 Always 的话是不是就陷入了死循环了？

然后来创建该 Job，保存为 job-demo.yaml：

```
$ kubectl create -f ./job.yaml
job "job-demo" created
```

然后我们可以查看当前的 Job 资源对象：

```
$ kubectl get jobs
```

注意查看我们的 Pod 的状态，同样我们可以通过 kubectl logs 来查看当前任务的执行结果。

CronJob

CronJob 其实就是在 Job 的基础上加上了时间调度，我们可以：在给定的时间点运行一个任务，也可以周期性地在给定时间点运行。这个实际上和我们 Linux 中的 crontab 就非常类似了。

一个 CronJob 对象其实就对应中 crontab 文件中的一行，它根据配置的时间格式周期性地运行一个 Job，格式和 crontab 也是一样的。

crontab 的格式如下：

```
分时日月星期要运行的命令 第1列分钟0~59 第2列小时0~23) 第3列日1~31 第4列月1~12 第5列星期0~7 (0和7表示星期天) 第6列要运行的命令
```

现在，我们用 CronJob 来管理我们上面的 Job 任务，

```
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: cronjob-demo
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
            - name: hello
              image: busybox
              args:
                - "bin/sh"
                - "-c"
                - "for i in 9 8 7 6 5 4 3 2 1; do echo $i; done"
```

我们这里的 Kind 是 CronJob 了，要注意的是 .spec.schedule 字段是必须填写的，用来指定任务运行的周期，格式就和 crontab 一样，另外一个字段是 .spec.jobTemplate，用来指定需要运行的任务，格式当然和 Job 是一致的。还有一些值得我们关注的字

段 .spec.successfulJobsHistoryLimit 和 .spec.failedJobsHistoryLimit，表示历史限制，是可选的字段。它们指定了可以保留多少完成和失败的 Job，默认没有限制，所有成功和失败的 Job 都会被保留。然而，当运行一个 Cron Job 时，Job 可以很快就堆积很多，所以一般推荐设置这两个字段的值。如果设置限制的值为 0，那么相关类型的 Job 完成后将不会被保留。

接下来我们来创建这个 cronjob

```
$ kubectl create -f cronjob-demo.yaml
cronjob "cronjob-demo" 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
```

```
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

一旦不再需要 Cron Job，简单地可以使用 kubectl 命令删除它：

```
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

这将会终止正在创建的 Job。然而，运行中的 Job 将不会被终止，不会删除 Job 或它们的 Pod。为了清理那些 Job 和 Pod，需要列出该 Cron Job 创建的全部 Job，然后删除它们：

```
$ kubectl get jobs
NAME      DESIRED      SUCCESSFUL      AGE
hello-1201907962  1            1            11m
hello-1202039034  1            1            8m
...
$ kubectl delete jobs hello-1201907962 hello-1202039034 ...
job "hello-1201907962" deleted
job "hello-1202039034" deleted
...
```

一旦 Job 被删除，由 Job 创建的 Pod 也会被删除。注意，所有由名称为“hello”的 Cron Job 创建的 Job 会以前缀字符串“hello-”进行命名。如果想要删除当前 Namespace 中的所有 Job，可以通过命令 kubectl delete jobs --all 立刻删除它们。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



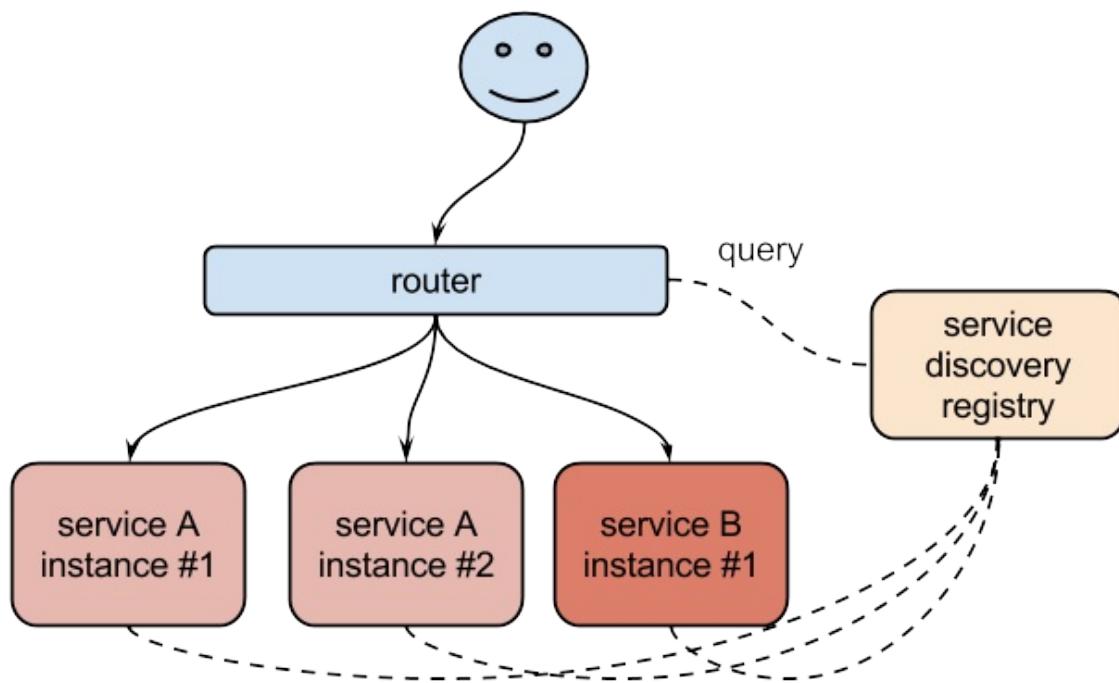
k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:01:42

Service

我们前面的课程中学习了 Pod 的基本用法，我们也了解到 Pod 的生命是有限的，死亡过后不会复活了。我们后面学习到的 RC 和 Deployment 可以用来动态的创建和销毁 Pod。尽管每个 Pod 都有自己的 IP 地址，但是如果 Pod 重新启动了的话那么他的 IP 很有可能也就变化了。这就会带来一个问题：比如我们有一些后端的 Pod 的集合为集群中的其他前端的 Pod 集合提供 API 服务，如果我们在前端的 Pod 中把所有的这些后端的 Pod 的地址都写死，然后去某种方式去访问其中一个 Pod 的服务，这样看上去是可以工作的，对吧？但是如果这个 Pod 挂掉了，然后重新启动起来了，是不是 IP 地址非常有可能就变了，这个时候前端就极大可能访问不到后端的服务了。

遇到这样的问题该怎么解决呢？在没有使用 Kubernetes 之前，我相信可能很多同学都遇到过这样的问题，不一定是 IP 变化的问题，比如我们在部署一个 WEB 服务的时候，前端一般部署一个 Nginx 作为服务的入口，然后 Nginx 后面肯定就是挂载的这个服务的大量后端，很早以前我们可能是去手动更改 Nginx 配置中的 upstream 选项，来动态改变提供服务的数量，到后面出现了一些 服务发现 的工具，比如 Consul、ZooKeeper 还有我们熟悉的 etcd 等工具，有了这些工具过后我们就可以只需要把我们的服务注册到这些服务发现中心去就可以，然后让这些工具动态的去更新 Nginx 的配置就可以了，我们完全不用去手工的操作了，是不是非常方便。



同样的，要解决我们上面遇到的问题是不是实现一个服务发现的工具也可以解决啊？没错的，当我们 Pod 被销毁或者新建过后，我们可以把这个 Pod 的地址注册到这个服务发现中心去就可以，但是这样的话我们的前端的 Pod 结合就不能直接去连接后台的 Pod 集合了是吧，应该连接到一个能够做服务发现的中间件上面，对吧？

没错，Kubernetes 集群就为我们提供了一个对象 - `Service`，`Service` 是一种抽象的对象，它定义了一组 `Pod` 的逻辑集合和一个用于访问它们的策略，其实这个概念和微服务非常类似。一个 `Service` 下面包含的 `Pod` 集合一般是由 `Label Selector` 来决定的。

比如我们上面的例子，假如我们后端运行了3个副本，这些副本都是可以替代的，因为前端并不关心它们使用的是哪一个后端服务。尽管由于各种原因后端的 `Pod` 集合会发送变化，但是前端却不需要知道这些变化，也不需要自己用一个列表来记录这些后端的服务，`Service` 的这种抽象就可以帮我们达到这种解耦的目的。

三种IP

在继续往下学习 `Service` 之前，我们需要先弄明白 Kubernetes 系统中的三种IP这个问题，因为经常有同学混乱。

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

首先，`Node IP` 是 Kubernetes 集群中节点的物理网卡 IP 地址(一般为内网)，所有属于这个网络的服务器之间都可以直接通信，所以 Kubernetes 集群外要想访问 Kubernetes 集群内部的某个节点或者服务，肯定得通过 `Node IP` 进行通信（这个时候一般是通过外网 IP 了）

然后 `Pod IP` 是每个 `Pod` 的 IP 地址，它是 Docker Engine 根据 `docker0` 网桥的 IP 地址段进行分配的（我们这里使用的是 `flannel` 这种网络插件保证所有节点的 `Pod IP` 不会冲突）

最后 `Cluster IP` 是一个虚拟的 IP，仅仅作用于 `Kubernetes Service` 这个对象，由 Kubernetes 自己来进行管理和分配地址，当然我们也无法 ping 这个地址，他没有一个真正的实体对象来响应，他只能结合 `Service Port` 来组成一个可以通信的服务。

定义Service

定义 `Service` 的方式和我们前面定义的各种资源对象的方式类型，例如，假定我们有一组 `Pod` 服务，它们对外暴露了 8080 端口，同时都被打上了 `app=myapp` 这样的标签，那么我们就可以像下面这样来定义一个 `Service` 对象：

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      name: myapp-http
```

然后通过的使用 `kubectl create -f myservice.yaml` 就可以创建一个名为 `myservice` 的 Service 对象，它会将请求代理到使用 TCP 端口为 8080，具有标签 `app=myapp` 的 Pod 上，这个 Service 会被系统分配一个我们上面说的 Cluster IP，该 Service 还会持续的监听 selector 下面的 Pod，会把这些 Pod 信息更新到一个名为 `myservice` 的 Endpoints 对象上去，这个对象就类似于我们上面说的 Pod 集合了。

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。因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，所以对于部署和设计 Service，这种方式会提供更大的灵活性。

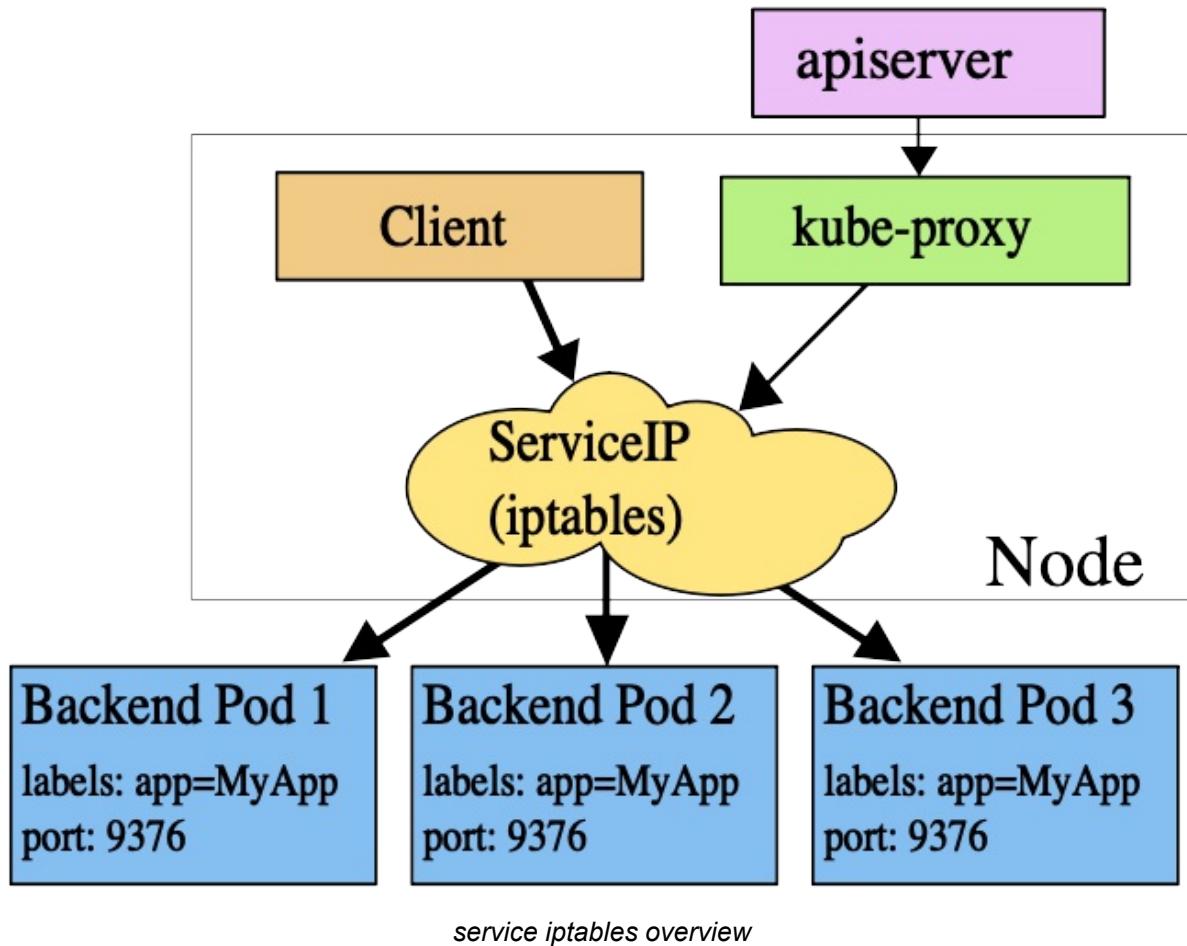
另外 Service 能够支持 TCP 和 UDP 协议，默认是 TCP 协议。

kube-proxy

前面我们讲到过，在 Kubernetes 集群中，每个 Node 会运行一个 `kube-proxy` 进程，负责为 Service 实现一种 VIP（虚拟 IP，就是我们上面说的 `clusterIP`）的代理形式，现在的 Kubernetes 中默认是使用的 `iptables` 这种模式来代理。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，它会添加上 `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 中的某一个上面。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。我们也可以实现基于客户端 IP 的会话亲和性，可以将 `service.spec.sessionAffinity` 的值设置为 "ClientIP"（默认值为 "None"）。

另外需要了解的是如果最开始选择的 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。



Service 类型

我们在定义 Service 的时候可以指定一个自己需要的类型的 Service，如果不指定的话默认是 ClusterIP 类型。

我们可以使用的服务类型如下：

- 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 节点上的 IP 和静态端口（NodePort）暴露服务。NodePort 服务会路由到 ClusterIP 服务，这个 ClusterIP 服务会自动创建。通过请求：，可以从集群的外部访问一个 NodePort 服务。
- LoadBalancer：使用云提供商的负载均衡器，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，这个需要结合具体的云厂商进行操作。
- 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，如果不指定的话会自动生成一个端口。

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

接下来我们来给大家创建一个 NodePort 的服务来访问我们前面的 Nginx 服务：(保存为service-demo.yaml)

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  selector:
    app: myapp
  type: NodePort
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    name: myapp-http
```

创建该 Service：

```
$ kubectl create -f service-demo.yaml
```

然后我们可以查看 Service 对象信息：

```
$ kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP      27d
myservice   NodePort   10.104.57.198  <none>        80:32560/TCP  14h
```

我们可以看到 myservice 的 TYPE 类型已经变成了 NodePort，后面的 PORT(S) 部分也多了一个 32560 的映射端口。

ExternalName

`ExternalName` 是 Service 的特例，它没有 selector，也没有定义任何的端口和 Endpoint。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.local` (后面服务发现的时候我们会再深入讲解) 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, 完全不需要修改调用的代码, 这样就完全解耦了。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号, 在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



*k8s技术圈*二维码

ConfigMap

前面的课程中我们学习了 Service 的使用，Service 是 Kubernetes 系统中非常重要的一个核心概念，我们还会在后面的课程中继续学习 Service 的一些使用方法的。这节课我们来学习另外一个非常重要的资源对象：ConfigMap，我们知道许多应用经常会有从配置文件、命令行参数或者环境变量中读取一些配置信息，这些配置信息我们肯定不会直接写死到应用程序中去的，比如你一个应用连接一个 redis 服务，下一次想更换一个了的，还得重新去修改代码，重新制作一个镜像，这肯定是不可取的，而 ConfigMap 就给我们提供了向容器中注入配置信息的能力，不仅可以用来保存单个属性，也可以用来保存整个配置文件，比如我们可以用来配置一个 redis 服务的访问地址，也可以用来保存整个 redis 的配置文件。

创建

ConfigMap 资源对象使用 key-value 形式的键值对来配置数据，这些数据可以在 Pod 里面使用，ConfigMap 和我们后面要讲到的 Secrets 比较类似，一个比较大的区别是 ConfigMap 可以比较方便的处理一些非敏感的数据，比如密码之类的还是需要使用 Secrets 来进行管理。我们来举个例子说明下 ConfigMap 的使用方法：

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: cm-demo
  namespace: default
data:
  data.1: hello
  data.2: world
  config: |
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

其中配置数据在 data 属性下面进行配置，前两个被用来保存单个属性，后面一个被用来保存一个配置文件。

当然同样的我们可以使用 kubectl create -f xx.yaml 来创建上面的 ConfigMap 对象，但是如果我不知道怎么创建 ConfigMap 的话，不要忘记 kubectl 是我们最好的老师，可以使用 kubectl create configmap -h 来查看关于创建 ConfigMap 的帮助信息，

```
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=confi
g2
```

我们可以看到可以从一个给定的目录来创建一个 ConfigMap 对象，比如我们有一个 `testcm` 的目录，该目录下面包含一些配置文件，`redis` 和 `mysql` 的连接信息，如下：

```
$ ls testcm
redis.conf
mysql.conf

$ cat testcm/redis.conf
host=127.0.0.1
port=6379

$ cat testcm/mysql.conf
host=127.0.0.1
port=3306
```

然后我们可以使用 `from-file` 关键字来创建包含这个目录下面所有配置文件的 ConfigMap：

```
$ kubectl create configmap cm-demo1 --from-file=testcm
configmap "cm-demo1" created
```

其中 `from-file` 参数指定在该目录下面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。

创建完成后，同样我们可以使用如下命令来查看 ConfigMap 列表：

```
$ kubectl get configmap
NAME      DATA      AGE
cm-demo1  2          17s
```

可以看到已经创建了一个 `cm-demo1` 的 ConfigMap 对象，然后可以使用 `describe` 命令查看详细信息：

```
kubectl describe configmap cm-demo1
Name:           cm-demo1
Namespace:      default
Labels:         <none>
Annotations:   <none>

Data
-----
mysql.conf:
-----
host=127.0.0.1
port=3306

redis.conf:
-----
host=127.0.0.1
port=6379

Events:  <none>
```

我们可以看到两个 key 是 testcm 目录下面的文件名称，对应的 value 值的话就是文件内容，这里值得注意的是如果文件里面的配置信息很大的话，`describe` 的时候可能不会显示对应的值，要查看键值的话，可以使用如下命令：

```
$ kubectl get configmap cm-demo1 -o yaml
apiVersion: v1
data:
  mysql.conf: |
    host=127.0.0.1
    port=3306
  redis.conf: |
    host=127.0.0.1
    port=6379
kind: ConfigMap
metadata:
  creationTimestamp: 2018-06-14T16:24:36Z
  name: cm-demo1
  namespace: default
  resourceVersion: "3109975"
  selfLink: /api/v1/namespaces/default/configmaps/cm-demo1
  uid: 6e0f4d82-6fef-11e8-a101-525400db4df7
```

除了通过文件目录进行创建，我们也可以使用指定的文件进行创建 `ConfigMap`，同样的，以上面的配置文件为例，我们创建一个 `redis` 的配置的一个单独 `ConfigMap` 对象：

```
$ kubectl create configmap cm-demo2 --from-file=testcm/redis.conf
configmap "cm-demo2" created
$ kubectl get configmap cm-demo2 -o yaml
apiVersion: v1
data:
  redis.conf: |
    host=127.0.0.1
    port=6379
kind: ConfigMap
metadata:
  creationTimestamp: 2018-06-14T16:34:29Z
  name: cm-demo2
  namespace: default
  resourceVersion: "3110758"
  selfLink: /api/v1/namespaces/default/configmaps/cm-demo2
  uid: cf59675d-6ff0-11e8-a101-525400db4df7
```

我们可以看到一个关联 `redis.conf` 文件配置信息的 `ConfigMap` 对象创建成功了，另外值得注意的是 `--from-file` 这个参数可以使用多次，比如我们这里使用两次分别指定 `redis.conf` 和 `mysql.conf` 文件，就和直接指定整个目录是一样的效果了。

另外，通过帮助文档我们可以看到我们还可以直接使用字符串进行创建，通过 `--from-literal` 参数传递配置信息，同样的，这个参数可以使用多次，格式如下：

```
$ kubectl create configmap cm-demo3 --from-literal=db.host=localhost --from-literal=db.port=3306
configmap "cm-demo3" created
```

```
$ kubectl get configmap cm-demo3 -o yaml
apiVersion: v1
data:
  db.host: localhost
  db.port: "3306"
kind: ConfigMap
metadata:
  creationTimestamp: 2018-06-14T16:43:12Z
  name: cm-demo3
  namespace: default
  resourceVersion: "3111447"
  selfLink: /api/v1/namespaces/default/configmaps/cm-demo3
  uid: 06eeec7e-6ff2-11e8-a101-525400db4df7
```

使用

ConfigMap 创建成功了，那么我们应该怎么在 Pod 中来使用呢？我们说 ConfigMap 这些配置数据可以通过很多种方式在 Pod 里使用，主要有以下几种方式：

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

首先，我们使用 ConfigMap 来填充我们的环境变量：

```
apiVersion: v1
kind: Pod
metadata:
  name: testcm1-pod
spec:
  containers:
    - name: testcm1
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: cm-demo3
              key: db.host
        - name: DB_PORT
          valueFrom:
            configMapKeyRef:
              name: cm-demo3
              key: db.port
      envFrom:
        - configMapRef:
            name: cm-demo1
```

这个Pod运行后会输出如下几行：

```
$ kubectl logs testcm1-pod
...
DB_HOST=localhost
```

```
DB_PORT=3306
mysql.conf=host=127.0.0.1
port=3306
redis.conf=host=127.0.0.1
port=6379
....
```

我们可以看到 `DB_HOST` 和 `DB_PORT` 都已经正常输出了，另外的环境变量是因为我们这里直接把 `cm-demo1` 给注入进来了，所以把他们的整个键值给输出出来了，这也是符合预期的。

另外我们可以使用 `ConfigMap` 来设置命令行参数，`ConfigMap` 也可以被用来设置容器中的命令或者参数值，如下 Pod：

```
apiVersion: v1
kind: Pod
metadata:
  name: testcm2-pod
spec:
  containers:
    - name: testcm2
      image: busybox
      command: [ "/bin/sh", "-c", "echo $(DB_HOST) $(DB_PORT)" ]
      env:
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: cm-demo3
              key: db.host
        - name: DB_PORT
          valueFrom:
            configMapKeyRef:
              name: cm-demo3
              key: db.port
```

运行这个 Pod 后会输出如下信息：

```
$ kubectl logs testcm2-pod
localhost 3306
```

另外一种是非常常见的使用 `ConfigMap` 的方式：通过数据卷使用，在数据卷里面使用 `ConfigMap`，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: testcm3-pod
spec:
  containers:
    - name: testcm3
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/config/redis.conf" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
```

```

volumes:
- name: config-volume
  configMap:
    name: cm-demo2

```

运行这个 Pod 的，查看日志：

```

$ kubectl logs testcm3-pod
host=127.0.0.1
port=6379

```

当然我们也可以在 ConfigMap 值被映射的数据卷里去控制路径，如下 Pod 定义：

```

apiVersion: v1
kind: Pod
metadata:
  name: testcm4-pod
spec:
  containers:
  - name: testcm4
    image: busybox
    command: [ "/bin/sh", "-c", "cat /etc/config/path/to/mysql.conf" ]
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: cm-demo1
      items:
      - key: mysql.conf
        path: path/to/mysql.conf

```

运行这个 Pod 的，查看日志：

```

$ kubectl logs testcm4-pod
host=127.0.0.1
port=3306

```

另外需要注意的是，当 ConfigMap 以数据卷的形式挂载进 Pod 的时，这时更新 ConfigMap（或删掉重建 ConfigMap），Pod 内挂载的配置信息会热更新。这时可以增加一些监测配置文件变更的脚本，然后 reload 对应服务。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:02:30

Secret

上节课我们学习了 ConfigMap 的时候，我们说 ConfigMap 这个资源对象是 Kubernetes 当中非常重要的一个对象，一般情况下 ConfigMap 是用来存储一些非安全的配置信息，如果涉及到一些安全相关的数据的话用 ConfigMap 就非常不妥了，因为 ConfigMap 是名为存储的，我们说这个时候我们就需要用到另外一个资源对象了： Secret ， Secret 用来保存敏感信息，例如密码、OAuth 令牌和 ssh key 等等，将这些信息放在 Secret 中比放在 Pod 的定义中或者 docker 镜像中来说更加安全和灵活。

Secret 有三种类型：

- Opaque: base64 编码格式的 Secret，用来存储密码、密钥等；但数据也可以通过base64 – decode解码得到原始数据，所有加密性很弱。
- kubernetes.io/dockerconfigjson: 用来存储私有docker registry的认证信息。
- kubernetes.io/service-account-token: 用于被 serviceaccount 引用， serviceaccount 创建时 Kubernetes会默认创建对应的secret。Pod如果使用了serviceaccount，对应的secret会自动挂载到Pod目录 /run/secrets/kubernetes.io/serviceaccount 中。

Opaque Secret

Opaque 类型的数据是一个 map 类型，要求 value 是 base64 编码格式，比如我们来创建一个用户名为 admin， 密码为 admin321 的 Secret 对象，首先我们先把这用户名和密码做 base64 编码，

```
$ echo -n "admin" | base64
YWRTaW4=
$ echo -n "admin321" | base64
YWRTaW4zMjE=
```

然后我们就可以利用上面编码过后的数据来编写一个 YAML 文件：(secret-demo.yaml)

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRTaW4=
  password: YWRTaW4zMjE=
```

然后同样的我们就可以使用 kubectl 命令来创建了：

```
$ kubectl create -f secret-demo.yaml
secret "mysecret" created
```

利用 get secret 命令查看：

```
$ kubectl get secret
```

NAME	TYPE	DATA	AGE
default-token-n9w2d	kubernetes.io/service-account-token	3	33d
mysecret	Opaque	2	40s

其中 `default-token-cty7pdefault-token-n9w2d` 为创建集群时默认创建的 secret，被 `serviceaccount/default` 引用。

使用 `describe` 命令，查看详情：

```
$ kubectl describe secret mysecret
Name:         mysecret
Namespace:    default
Labels:       <none>
Annotations: <none>

Type:  Opaque

Data
====
password:  8 bytes
username:  5 bytes
```

我们可以看到利用 `describe` 命令查看到的 `Data` 没有直接显示出来，如果想看到 `Data` 里面的详细信息，同样我们可以输出成 `YAML` 文件进行查看：

```
$ kubectl get secret mysecret -o yaml
apiVersion: v1
data:
  password: YWRtaW4zMjE=
  username: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2018-06-19T15:27:06Z
  name: mysecret
  namespace: default
  resourceVersion: "3694084"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 39c139f5-73d5-11e8-a101-525400db4df7
type: Opaque
```

创建好 `Secret` 对象后，有两种方式来使用它：

- 以环境变量的形式
- 以Volume的形式挂载

环境变量

首先我们来测试下环境变量的方式，同样的，我们来使用一个简单的 `busybox` 镜像来测试下(`secret1-pod.yaml`)

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: secret1-pod
spec:
  containers:
    - name: secret1
      image: busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: password

```

主要上面环境变量中定义的 `secretKeyRef` 关键字，和我们上节课的 `configMapKeyRef` 是不是比较类似，一个是从 `Secret` 对象中获取，一个是从 `ConfigMap` 对象中获取，创建上面的 Pod：

```
$ kubectl create -f secret1-pod.yaml
pod "secret1-pod" created
```

然后我们查看 Pod 的日志输出：

```
$ kubectl logs secret1-pod
...
USERNAME=admin
PASSWORD=admin321
...
```

可以看到有 `USERNAME` 和 `PASSWORD` 两个环境变量输出出来。

Volume 挂载

同样的我们用一个 Pod 来验证下 Volume 挂载，创建一个 Pod 文件：(`secret2-pod.yaml`)

```

apiVersion: v1
kind: Pod
metadata:
  name: secret2-pod
spec:
  containers:
    - name: secret2
      image: busybox
      command: [ "/bin/sh", "-c", "ls /etc/secrets" ]
      volumeMounts:
        - name: secrets
          mountPath: /etc/secrets
  volumes:
    - name: secrets
      secret:

```

```
secretName: mysecret
```

创建 Pod :

```
$ kubectl create -f secret-pod2.yaml
pod "secret2-pod" created
```

然后我们查看输出日志：

```
$ kubectl logs secret2-pod
password
username
```

可以看到 secret 把两个key挂载成了两个对应的文件。当然如果想要挂载到指定的文件上面，是不是也可以使用上一节课的方法：在 secretName 下面添加 items 指定 key 和 path，这个大家可以参考上节课 ConfigMap 中的方法去测试下。

kubernetes.io/dockerconfigjson

除了上面的 Opaque 这种类型外，我们还可以来创建用户 docker registry 认证的 Secret，直接使用 kubectl create 命令创建即可，如下：

```
$ kubectl create secret docker-registry myregistry --docker-server=DOCKER_SERVER --docker-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL
secret "myregistry" created
```

然后查看 Secret 列表：

NAME	TYPE	DATA	AGE
default-token-n9w2d	kubernetes.io/service-account-token	3	33d
myregistry	kubernetes.io/dockerconfigjson	1	15s
mysecret	Opaque	2	34m

注意看上面的 TYPE 类型， myregistry 是不是对应的 kubernetes.io/dockerconfigjson，同样的可以使用 describe 命令来查看详细信息：

```
$ kubectl describe secret myregistry
Name:           myregistry
Namespace:      default
Labels:          <none>
Annotations:    <none>

Type:  kubernetes.io/dockerconfigjson

Data
=====
.dockerconfigjson:  152 bytes
```

同样的可以看到 Data 区域没有直接展示出来，如果想查看的话可以使用 `-o yaml` 来输出展示出来：

```
$ kubectl get secret myregistry -o yaml
apiVersion: v1
data:
  .dockerconfigjson: eyJhdXRocI6eyJET0NLRVJfU0VSvksIjp7InVzZXJuYW1lIjoiRE9DS0VSX1VTRVIiL
  CJwYXNzd29yZCI6IkRPQ0tFU19QQVNTV09SRCIsImVtYWsIjoiRE9DS0VSX0VNQU1MIiwiYXV0aCI6I1JFOURTMFZ
  TWDFWVFJWSTZSRT1EUzBWU1gxQkJVMU5YVDFKRSJ9fx0=
kind: Secret
metadata:
  creationTimestamp: 2018-06-19T16:01:05Z
  name: myregistry
  namespace: default
  resourceVersion: "3696966"
  selfLink: /api/v1/namespaces/default/secrets/myregistry
  uid: f91db707-73d9-11e8-a101-525400db4df7
  type: kubernetes.io/dockerconfigjson
```

可以把上面的 `data.dockerconfigjson` 下面的数据做一个 `base64` 解码，看看里面的数据是怎样的呢？

```
$ echo eyJhdXRocI6eyJET0NLRVJfU0VSvksIjp7InVzZXJuYW1lIjoiRE9DS0VSX1VTRVIiLCJwYXNzd29yZCI
  6IkRPQ0tFU19QQVNTV09SRCIsImVtYWsIjoiRE9DS0VSX0VNQU1MIiwiYXV0aCI6I1JFOURTMFZTWDFWVFJWSTZS
  RT1EUzBWU1gxQkJVMU5YVDFKRSJ9fx0= | base64 -d
  {"auths": {"DOCKER_SERVER": {"username": "DOCKER_USER", "password": "DOCKER_PASSWORD", "email": "DOCKER_EMAIL", "auth": "RE9DS0VSX1VTRVI6RE9DS0VSX1BBU1NXT1JE"}}}
```

如果我们需要拉取私有仓库中的 `docker` 镜像的话就需要使用到上面的 `myregistry` 这个 `Secret`：

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
    - name: foo
      image: 192.168.1.100:5000/test:v1
  imagePullSecrets:
    - name: myregistrykey
```

我们需要拉取私有仓库镜像 `192.168.1.100:5000/test:v1`，我们就需要针对该私有仓库来创建一个如上的 `Secret`，然后在 `Pod` 的 `YAML` 文件中指定 `imagePullSecrets`，我们会在后面的私有仓库搭建的课程中跟大家详细说明的。

kubernetes.io/service-account-token

另外一种 `Secret` 类型就是 `kubernetes.io/service-account-token`，用于被 `serviceaccount` 引用。`serviceaccount` 创建时 `Kubernetes` 会默认创建对应的 `secret`。`Pod` 如果使用了 `serviceaccount`，对应的 `secret` 会自动挂载到 `Pod` 的 `/run/secrets/kubernetes.io/serviceaccount` 目录中。

这里我们使用一个 `nginx` 镜像来验证一下，大家想一想为什么不是呀 `busybox` 镜像来验证？当然也是可以的，但是我们就不能在 `command` 里面来验证了，因为 `token` 是需要 `Pod` 运行起来过后才会被挂载上去的，直接在 `command` 命令中去查看肯定是还没有 `token` 文件的。

```
$ kubectl run secret-pod3 --image nginx:1.7.9
deployment.apps "secret-pod3" created
$ kubectl get pods
NAME                   READY   STATUS    RESTARTS   AGE
secret-pod3-78c8c76db8-7zmqm   1/1     Running   0          13s
...
$ kubectl exec secret-pod3-78c8c76db8-7zmqm ls /run/secrets/kubernetes.io/serviceaccount
ca.crt
namespace
token
$ kubectl exec secret-pod3-78c8c76db8-7zmqm cat /run/secrets/kubernetes.io/serviceaccount/
token
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcw5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZX
R1cy5pb9zZXJ2aWN1YWNgjb3VudC9uYW1lc3BhY2UiOijkZWZhdWx0Iiwia3ViZXJuZXRLcy5pb9zZXJ2aWN1YW
jb3VudC9zZWNyZXQubmFtZSI6ImR1ZmF1bHQtdG9rZW4tbj13MmQilCJrdWJlcw5ldGVzLm1vL3NlcnZpY2VhY2Nv
dW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoizGVmYXVsdcIsImt1YmVybmv0ZXMuaw8vc2VydmljZWfjY291bnQvc2V
ydm1jZS1hY2NvdW50LnVpZCI6IjMzY2FkOWQxLTU5MmYtMTF1OC1hMTAxLTUyNTQwMGR1NGRmNyIsInN1YiI6InN5c
3R1bTpzzXJ2aWN1YWNgjb3VudDpkZWZhdWx0OmR1ZmF1bHQifQ.0FpzPD8W0_fwnMjwpGIOphdVu4K9wUINwpXpBOJA
Q-Tawd0RTbAUHcgYy3sEHSk9uvgnl1FJRQpbQN3yVR_DWSI1Atbmd4dIPxK407ZVdd4UnmC467cNXEBql1sDWLfs5f
03d7D1dw11jFJ_pJw2P65Fjd13reKJvvTQnpu5U0SDcfxj675-Z3z-i003XSa1ZmkFIw2MfYMzf_WpxW0yMFCVkuZ
tBStegA9-NJZedecA_VC0dKcUjDPrDo-CNti3wZqax5WPw950u8RJDMAIS5EcVym7M2_zjGiqHEL3VTvcwXbdFKx
sNX-1VW6nr_KKuMGKOyx-5vgxeb171QQ
```

Secret 与 ConfigMap 对比

最后我们来对比下 `Secret` 和 `ConfigMap` 这两种资源对象的异同点：

相同点：

- key/value 的形式
- 属于某个特定的 namespace
- 可以导出到环境变量
- 可以通过目录/文件形式挂载
- 通过 volume 挂载的配置信息均可热更新

不同点：

- Secret 可以被 ServerAccount 关联
- Secret 可以存储 docker register 的鉴权信息，用在 `ImagePullSecret` 参数中，用于拉取私有仓库的镜像
- Secret 支持 Base64 加密
- Secret 分为 `kubernetes.io/service-account-token`、`kubernetes.io/dockerconfigjson`、`Opaque` 三种类型，而 Configmap 不区分类型

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:02:41

RBAC

前面两节课我们学习了 Kubernetes 中的两个用于配置信息的重要资源对象： ConfigMap 和 Secret，其实到这里我们基本上学习的内容已经覆盖到 Kubernetes 中一些重要的资源对象了，来部署一个应用程序是完全没有问题的了。在我们演示一个完整的示例之前，我们还需要给大家讲解一个重要的概念： RBAC - 基于角色的访问控制。

RBAC 使用 `rbac.authorization.k8s.io` API Group 来实现授权决策，允许管理员通过 Kubernetes API 动态配置策略，要启用 RBAC，需要在 apiserver 中添加参数 `--authorization-mode=RBAC`，如果使用的 `kubeadm` 安装的集群，1.6 版本以上的都默认开启了 RBAC，可以通过查看 Master 节点上 apiserver 的静态 Pod 定义文件：

```
$ cat /etc/kubernetes/manifests/kube-apiserver.yaml
...
- --authorization-mode=Node,RBAC
...
```

如果是二进制的方式搭建的集群，添加这个参数过后，记得要重启 apiserver 服务。

RBAC API 对象

Kubernetes 有一个很基本的特性就是它的所有资源对象都是模型化的 API 对象，允许执行 CRUD(Create、Read、Update、Delete)操作(也就是我们常说的增、删、改、查操作)，比如下面的这些资源：

- Pods
- ConfigMaps
- Deployments
- Nodes
- Secrets
- Namespaces

上面这些资源对象的可能存在的操作有：

- create
- get
- delete
- list
- update
- edit
- watch
- exec

在更上层，这些资源和 API Group 进行关联，比如 Pods 属于 Core API Group，而 Deployments 属于 apps API Group，要在 Kubernetes 中进行 RBAC 的管理，除了上面的这些资源和操作以外，我们还需要另外的一些对象：

- Rule：规则，规则是一组属于不同 API Group 资源上的一组操作的集合
- Role 和 ClusterRole：角色和集群角色，这两个对象都包含上面的 Rules 元素，二者的区别在于，在 Role 中，定义的规则只适用于单个命名空间，也就是和 namespace 关联的，而 ClusterRole 是集群范围内的，因此定义的规则不受命名空间的约束。另外 Role 和 ClusterRole 在 Kubernetes 中都被定义为集群内部的 API 资源，和我们前面学习过的 Pod、ConfigMap 这些类似，都是我们集群的资源对象，所以同样的可以使用我们前面的 kubectl 相关的命令来进行操作
- Subject：主题，对应在集群中尝试操作的对象，集群中定义了3种类型的主题资源：
 - User Account：用户，这是有外部独立服务进行管理的，管理员进行私钥的分配，用户可以使用 KeyStone或者 Goolge 帐号，甚至一个用户名和密码的文件列表也可以。对于用户的管理集群内部没有一个关联的资源对象，所以用户不能通过集群内部的 API 来进行管理
 - Group：组，这是用来关联多个账户的，集群中有一些默认创建的组，比如cluster-admin
 - Service Account：服务帐号，通过 Kubernetes API 来管理的一些用户帐号，和 namespace 进行关联的，适用于集群内部运行的应用程序，需要通过 API 来完成权限认证，所以在集群 内部进行权限操作，我们都需要使用到 ServiceAccount，这也是我们这节课的重点
- RoleBinding 和 ClusterRoleBinding：角色绑定和集群角色绑定，简单来说就是把声明的 Subject 和我们的 Role 进行绑定的过程(给某个用户绑定上操作的权限)，二者的区别也是作用范围的区别：RoleBinding 只会影响到当前 namespace 下面的资源操作权限，而 ClusterRoleBinding 会影响到所有的 namespace。

接下来我们来通过几个示例来演示下 RBAC 的配置方法。

创建一个只能访问某个 namespace 的用户

我们来创建一个 User Account，只能访问 kube-system 这个命名空间：

- username: haimaxy
- group: youdianzhishi

第1步：创建用户凭证

我们前面已经提到过， Kubernetes 没有 User Account 的 API 对象，不过要创建一个用户帐号的话也是挺简单的，利用管理员分配给你的一个私钥就可以创建了，这个我们可以参考[官方文档中的方法](#)，这里我们来使用 OpenSSL 证书来创建一个 User，当然我们也可以使用更简单的 cfssl 工具来创建：

- 给用户 haimaxy 创建一个私钥，命名成：haimaxy.key：

```
$ openssl genrsa -out haimaxy.key 2048
```

- 使用我们刚刚创建的私钥创建一个证书签名请求文件：haimaxy.csr，要注意需要确保在 -subj 参数中指定用户名和组(CN表示用户名，O表示组)：

```
$ openssl req -new -key haimaxy.key -out haimaxy.csr -subj "/CN=haimaxy/O=youdianzhis"
```

- 然后找到我们的 Kubernetes 集群的 CA，我们使用的是 kubeadm 安装的集群，CA 相关证书位于 /etc/kubernetes/pki/ 目录下面，如果你是二进制方式搭建的，你应该在最开始搭建集群的时候就已经指定好了 CA 的目录，我们会利用该目录下面的 ca.crt 和 ca.key 两个文件来批准上面的证书请求
- 生成最终的证书文件，我们这里设置证书的有效期为500天：

```
$ openssl x509 -req -in haimaxy.csr -CA /etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out haimaxy.crt -days 500
```

现在查看我们当前文件夹下面是否生成了一个证书文件：

```
$ ls
haimaxy.csr haimaxy.key haimaxy.crt
```

- 现在我们可以使用刚刚创建的证书文件和私钥文件在集群中创建新的凭证和上下文(Context)：

```
$ kubectl config set-credentials haimaxy --client-certificate=haimaxy.crt --client-key=haimaxy.key
```

我们可以看到一个用户 haimaxy 创建了，然后为这个用户设置新的 Context：

```
$ kubectl config set-context haimaxy-context --cluster=kubernetes --namespace=kube-system --user=haimaxy
```

到这里，我们的用户 haimaxy 就已经创建成功了，现在我们使用当前的这个配置文件来操作 kubectl 命令的时候，应该会出现错误，因为我们还没有为该用户定义任何操作的权限呢：

```
$ kubectl get pods --context=haimaxy-context
Error from server (Forbidden): pods is forbidden: User "haimaxy" cannot list pods in the namespace "default"
```

第2步：创建角色

用户创建完成后，接下来就需要给该用户添加操作权限，我们来定义一个 YAML 文件，创建一个允许用户操作 Deployment、Pod、ReplicaSets 的角色，如下定义：(haimaxy-role.yaml)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: haimaxy-role
  namespace: kube-system
rules:
- apiGroups: [ "", "extensions", "apps" ]
  resources: [ "deployments", "replicasets", "pods" ]
```

```
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"] # 也可以使用["*"]
```

其中 Pod 属于 core 这个 API Group，在 YAML 中用空字符就可以，而 Deployment 属于 apps 这个 API Group，ReplicaSets 属于 extensions 这个 API Group(我怎么知道的？[点这里查文档](#))，所以 rules 下面的 apiGroups 就综合了这几个资源的 API Group：["", "extensions", "apps"]，其中 verbs 就是我们上面提到的可以对这些资源对象执行的操作，我们这里需要所有的操作方法，所以我们也可以使用[*]来代替。

然后创建这个 Role：

```
$ kubectl create -f haimaxy-role.yaml
```

注意这里我们没有使用上面的 haimaxy-context 这个上下文了，因为木有权限啦

第3步：创建角色权限绑定

Role 创建完成了，但是很明显现在我们这个 Role 和我们的用户 haimaxy 还没有任何关系，对吧？这里我就需要创建一个 RoleBinding 对象，在 kube-system 这个命名空间下面将上面的 haimaxy-role 角色和用户 haimaxy 进行绑定：(haimaxy-rolebinding.yaml)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: haimaxy-rolebinding
  namespace: kube-system
subjects:
- kind: User
  name: haimaxy
  apiGroup: ""
roleRef:
  kind: Role
  name: haimaxy-role
  apiGroup: ""
```

上面的 YAML 文件中我们看到了 subjects 关键字，这里就是我们上面提到的用来尝试操作集群的对象，这里对应上面的 User 帐号 haimaxy，使用 kubectl 创建上面的资源对象：

```
$ kubectl create -f haimaxy-rolebinding.yaml
```

第4步. 测试

现在我们应该可以上面的 haimaxy-context 上下文来操作集群了：

```
$ kubectl get pods --context=haimaxy-context
```

```
....
```

我们可以看到我们使用 kubectl 的使用并没有指定 namespace 了，这是因为我们已经为该用户分配了权限了，如果我们在后面加上一个 -n default 试看看呢？

```
$ kubectl --context=haimaxy-context get pods --namespace=default
Error from server (Forbidden): pods is forbidden: User "haimaxy" cannot list pods in the namespace "default"
```

是符合我们预期的吧？因为该用户并没有 default 这个命名空间的操作权限

创建一个只能访问某个 namespace 的ServiceAccount

上面我们创建了一个只能访问某个命名空间下面的普通用户，我们前面也提到过 subjects 下面还有一类类型的主题资源：ServiceAccount，现在我们来创建一个集群内部的用户只能操作 kube-system 这个命名空间下面的 pods 和 deployments，首先来创建一个 ServiceAccount 对象：

```
$ kubectl create sa haimaxy-sa -n kube-system
```

当然我们也可以定义成 YAML 文件的形式来创建。

然后新建一个 Role 对象：(haimaxy-sa-role.yaml)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: haimaxy-sa-role
  namespace: kube-system
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

可以看到我们这里定义的角色没有创建、删除、更新 Pod 的权限，待会我们可以重点测试一下，创建该 Role 对象：

```
$ kubectl create -f haimaxy-sa-role.yaml
```

然后创建一个 RoleBinding 对象，将上面的 haimaxy-sa 和角色 haimaxy-sa-role 进行绑定：(haimaxy-sa-rolebinding.yaml)

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: haimaxy-sa-rolebinding
  namespace: kube-system
subjects:
- kind: ServiceAccount
  name: haimaxy-sa
  namespace: kube-system
roleRef:
  kind: Role
```

```
name: haimaxy-sa-role
apiGroup: rbac.authorization.k8s.io
```

添加这个资源对象：

```
$ kubectl create -f haimaxy-sa-rolebinding.yaml
```

然后我们怎么去验证这个 ServiceAccount 呢？我们前面的课程中是不是提到过一个 ServiceAccount 会生成一个 Secret 对象和它进行映射，这个 Secret 里面包含一个 token，我们可以利用这个 token 去登录 Dashboard，然后我们就可以在 Dashboard 中来验证我们的功能是否符合预期了：

```
$ kubectl get secret -n kube-system | grep haimay-sa
haimay-sa-token-nxgqx           kubernetes.io/service-account-token  3          47m
$ kubectl get secret haimay-sa-token-nxgqx -o jsonpath={.data.token} -n kube-system | base64 -d
# 会生成一串很长的base64后的字符串
```

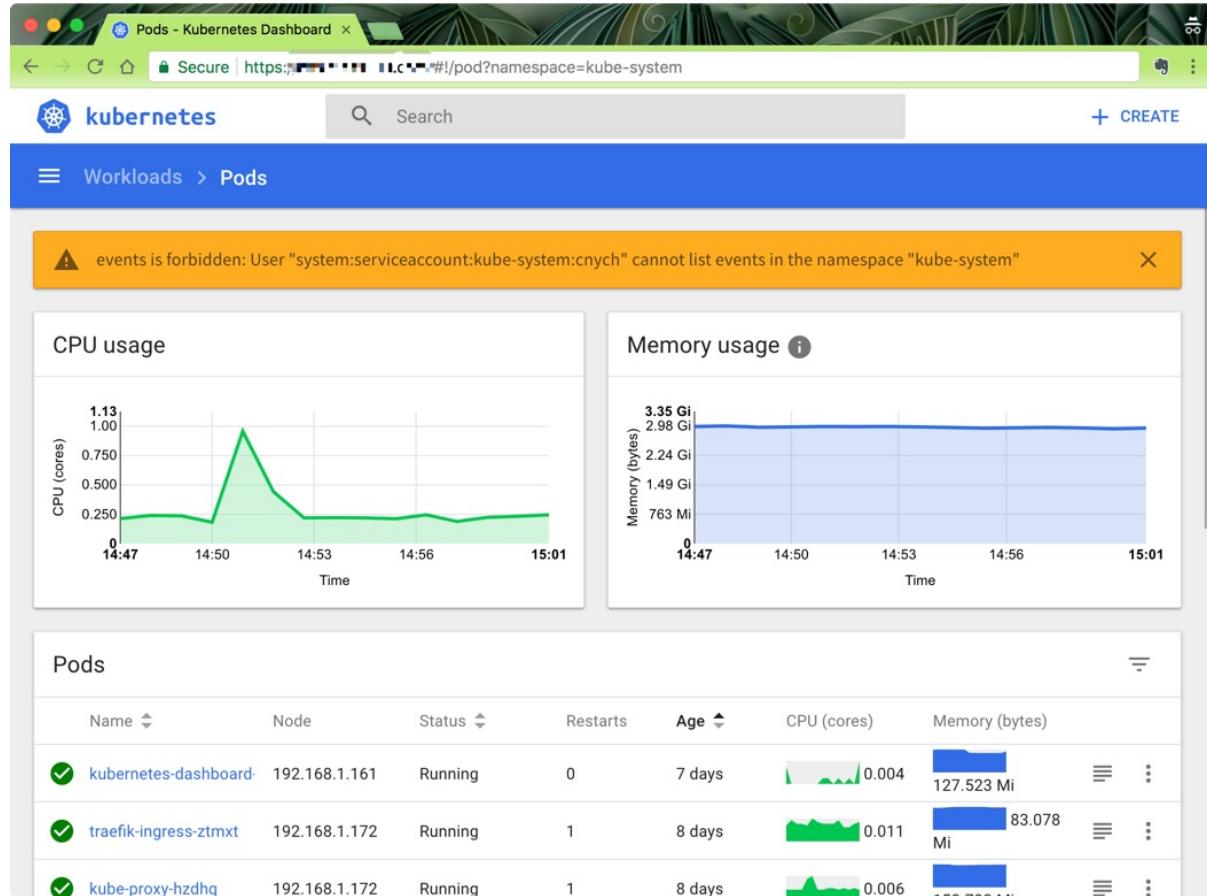
使用这里的 token 去 Dashboard 页面进行登录：

The screenshot shows the Kubernetes Dashboard's Overview page. At the top, there is a navigation bar with tabs for Overview, Nodes, Services, and Events. Below the navigation bar, the title is "Overview". A prominent yellow banner at the top displays two error messages from the Kubernetes API server:

- configmaps is forbidden:** User "system:serviceaccount:kube-system:kubernetes-dashboard" cannot list configmaps in the namespace "default"
- persistentvolumeclaims is forbidden:** User "system:serviceaccount:kube-system:kubernetes-dashboard" cannot list persistentvolumeclaims in the namespace "default"

Below the banner, there is a link to "SHOW 11 MORE" errors and a "DISMISS ALL" button. The main content area below the banner contains the message "There is nothing to display here". At the bottom of the content area, there is a note: "You can [deploy a containerized app](#), select other namespace or [take the Dashboard Tour](#) to learn more."

我们可以看到上面的提示信息，这是因为我们登录进来后默认跳转到 default 命名空间，我们切换到 kube-system 命名空间下面就可以了：



我们可以看到可以访问pod列表了，但是也会有一些其他额外的提示：events is forbidden: User "system:serviceaccount:kube-system:haimaxy-sa" cannot list events in the namespace "kube-system"，这是因为当前登录用只被授权了访问 pod 和 deployment 的权限，同样的，访问下 deployment看看可以了吗？

同样的，你可以根据自己的需求来对访问用户的权限进行限制，可以自己通过 Role 定义更加细粒度的权限，也可以使用系统内置的一些权限……

创建一个可以访问所有 namespace 的ServiceAccount

刚刚我们创建的 `haimaxy-sa` 这个 ServiceAccount 和一个 Role 角色进行绑定的，如果我们现在创建一个新的 ServiceAccount，需要他操作的权限作用于所有的 namespace，这个时候我们就需要使用到 ClusterRole 和 ClusterRoleBinding 这两种资源对象了。同样，首先新建一个 ServiceAccount 对象：`(haimaxy-sa2.yaml)`

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: haimaxy-sa2
  namespace: kube-system
```

创建：

```
$ kubectl create -f haimaxy-sa2.yaml
```

然后创建一个 ClusterRoleBinding 对象(haimaxy-clusterrolebinding.yaml)：

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: haimaxy-sa2-clusterrolebinding
subjects:
- kind: ServiceAccount
  name: haimaxy-sa2
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

从上面我们可以看到我们没有为这个资源对象声明 namespace，因为这是一个 ClusterRoleBinding 资源对象，是作用于整个集群的，我们也没有单独新建一个 ClusterRole 对象，而是使用的 cluster-admin 这个对象，这是 Kubernetes 集群内置的 ClusterRole 对象，我们可以使用 kubectl get clusterrole 和 kubectl get clusterrolebinding 查看系统内置的一些集群角色和集群角色绑定，这里我们使用的 cluster-admin 这个集群角色是拥有最高权限的集群角色，所以一般需要谨慎使用该集群角色。

创建上面集群角色绑定资源对象，创建完成后同样使用 ServiceAccount 对应的 token 去登录 Dashboard 验证下：

```
$ kubectl create -f haimaxy-clusterrolebinding.yaml
$ kubectl get secret -n kube-system | grep haimay-sa2
haimay-sa2-token-nxgqx           kubernetes.io/service-account-token   3          47
m
$ kubectl get secret haimay-sa2-token-nxgqx -o jsonpath='{.data.token}' -n kube-system | base
64 -d
# 会生成一串很长的base64后的字符串
```

我们在最开始接触到 RBAC 认证的时候，可能不太熟悉，特别是不知道应该怎么去编写 rules 规则，大家可以去分析系统自带的 clusterrole、clusterrolebinding 这些资源对象的编写方法，怎么分析？还是利用 kubectl 的 get、describe、-o yaml 这些操作，所以 kubectl 最基本的用户一定要掌握好。

Rbac 只是 Kubernetes 中安全认证的一种方式，当然也是现在最重要的一种方式，后面我们再和大家一起聊一聊 Kubernetes 中安全设计。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:02:46

部署 Wordpress 示例

前面的课程中我们基本上了解了 Kubernetes 当中常见的一些对象资源，这节课我们就来利用前面学习的知识点来部署一个实际的应用 - 将 Wordpress 应用部署到我们的集群当中，我们前面是不是已经用 docker-compose 的方式部署过了，我们可以了解到要部署一个 Wordpress 应用主要涉及到两个镜像： wordpress 和 mysql ， wordpress 是应用的核心程序， mysql 是用于数据存储的。

现在我们来看看如何来部署我们的这个 wordpress 应用

一个Pod

我们知道一个 Pod 中可以包含多个容器，那么很明显我们这里是不是就可以将 wordpress 部署成一个独立的 Pod ？我们将我们的应用都部署到 blog 这个命名空间下面，所以先创建一个命名空间：

```
$ kubectl create namespace blog
namespace "blog" created
```

然后来编写 YAML 文件： (wordpress-pod.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: wordpress
  namespace: blog
spec:
  containers:
    - name: wordpress
      image: wordpress
      ports:
        - containerPort: 80
          name: wdport
      env:
        - name: WORDPRESS_DB_HOST
          value: localhost: 3306
        - name: WORDPRESS_DB_USER
          value: wordpress
        - name: WORDPRESS_DB_PASSWORD
          value: wordpress
    - name: mysql
      image: mysql:5.7
      imagePullPolicy: IfNotPresent
      ports:
        - containerPort: 3306
          name: dbport
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: rootPassW0rd
        - name: MYSQL_DATABASE
          value: wordpress
        - name: MYSQL_USER
          value: wordpress
        - name: MYSQL_PASSWORD
```

```

    value: wordpress
  volumeMounts:
    - name: db
      mountPath: /var/lib/mysql
  volumes:
    - name: db
      hostPath:
        path: /var/lib/mysql

```

要注意这里针对 mysql 这个容器我们做了一个数据卷的挂载，这是为了能够将 mysql 的数据能够持久化到节点上，这样下次 mysql 容器重启过后数据不至于丢失。然后创建上面的 Pod：

```
$ kubectl create -f wrodpess-pod.yaml
pod "wordpress" created
```

接下来就是等待拉取镜像，启动容器，同样我们可以使用 describe 指令查看详细信息：

```
$ kubectl describe pod wordpress -n blog
```

大家可以看看我们现在这种单一 Pod 的方式有什么缺点？假如我们现在需要部署3个 Wordpress 的副本，该怎么办？是不是我们只需要在上面的 YAML 文件中加上 replicas: 3 这个属性就可以了啊？但是有个什么问题呢？是不是不仅仅是 Wordpress 这个容器会被部署成3份，连我们的 MySQL 数据库也会被部署成3份了呢？MySQL 数据库单纯的部署成3份他们能联合起来使用吗？不能，如果真的这么简单的话就不需要各种数据库集群解决方案了，所以我们这里部署3个 Pod 实例，实际上他们互相之间是独立的，因为数据不想通，明白吧？所以该怎么办？拆分呗，把 wordpress 和 mysql 这两个容器部署成独立的 Pod 是不是就可以了。

另外一个问题是我们的 wordpress 容器需要去连接 mysql 数据库吧，现在我们这里放在一起能保证 mysql 先启动起来吗？貌似没有特别的办法，前面学习的 InitContainer 也是针对 Pod 来的，所以无论如何，我们都需要将他们进行拆分。

两个Pod

现在来把上面的一个 Pod 拆分成两个 Pod，我们前面也反复强调过要使用 Deployment 来管理我们的 Pod，上面只是为了单纯给大家说明怎么来把前面的 Docker 环境下的 wordpress 转换成 Kubernetes 环境下面的 Pod，有了上面的 Pod 模板，我们现在来转换成 Deployment 很容易了吧。

第一步，创建一个 MySQL 的 Deployment 对象：(wordpress-db.yaml)

```

---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mysql_deploy
  namespace: blog
  labels:
    app: mysql
spec:

```

```

template:
  metadata:
    labels:
      app: mysql
  spec:
    containers:
      - name: mysql
        image: mysql:5.7
        imagePullPolicy: IfNotPresent
        ports:
          - containerPort: 3306
            name: dbport
        env:
          - name: MYSQL_ROOT_PASSWORD
            value: rootPassW0rd
          - name: MYSQL_DATABASE
            value: wordpress
          - name: MYSQL_USER
            value: wordpress
          - name: MYSQL_PASSWORD
            value: wordpress
        volumeMounts:
          - name: db
            mountPath: /var/lib/mysql
    volumes:
      - name: db
        hostPath:
          path: /var/lib/mysql

```

如果我们只创建上面的 Deployment 这个对象，那么我们应该怎样让后面的 Wordpress 来访问呢？貌似没办法是吧，之前在一个 Pod 里面还可以使用 localhost 来进行访问，现在分开了该怎样访问呢？还记得前面的 Service 吗？没错，使用 Service 就可以了，所以我们在上面的 `wordpress-db.yaml` 文件中添加上 Service 的信息：

```

---
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: blog
spec:
  selector:
    app: mysql
  ports:
    - name: mysqlport
      protocol: TCP
      port: 3306
      targetPort: dbport

```

然后创建上面的 `wordpress-db.yaml` 文件：

```

$ kubectl create -f wordpress-db.yaml
service "mysql" created
deployment.apps "mysql-deploy" created

```

然后我们查看 Service 的详细情况：

```
$ kubectl describe svc mysql -n blog
Name:           mysql
Namespace:      blog
Labels:          <none>
Annotations:    <none>
Selector:       app=mysql
Type:           ClusterIP
IP:             10.98.27.19
Port:           mysqlport  3306/TCP
TargetPort:     dbport/TCP
Endpoints:      10.244.2.213:3306
Session Affinity: None
Events:         <none>
```

可以看到 Endpoints 部分匹配到了一个 Pod，生成了一个 clusterIP : 10.98.27.19，现在我们是不是就可以通过这个 clusterIP 加上定义的3306端口就可以正常访问我们这个 mysql 服务了。

第二步. 创建 Wordpress 服务，将上面的 wordpress 的 Pod 转换成 Deployment 对象：

(wordpress.yaml)

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: wordpress-deploy
  namespace: blog
  labels:
    app: wordpress
spec:
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
        - name: wordpress
          image: wordpress
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
              name: wdport
          env:
            - name: WORDPRESS_DB_HOST
              value: 10.98.27.19:3306
            - name: WORDPRESS_DB_USER
              value: wordpress
            - name: WORDPRESS_DB_PASSWORD
              value: wordpress
```

注意这里的环境变量 WORDPRESS_DB_HOST 的值将之前的 localhost 地址更改成了上面 mysql 服务的 clusterIP 地址了，然后创建上面的 Deployment 对象：

```
$ kubectl create -f wordpress.yaml
deployment.apps "wordpress-deploy" created
```

创建完成后，我们可以看看我们创建的 Pod 的状态：

```
$ kubectl get pods -n blog
NAME                      READY   STATUS    RESTARTS   AGE
mysql-deploy-86bdcc7484-fv2dj   1/1     Running   0          19m
wordpress-deploy-784cf6dd4-d9f52  1/1     Running   0          23s
```

可以看到都已经是 Running 状态了，然后我们需要怎么来验证呢？是不是去访问下我们的 wordpress 服务就可以了，要访问，我们就需要建立一个能让外网用户访问的 Service，前面我们学到过是不是 NodePort 类型的 Service 就可以？所以在上面的 wordpress.yaml 文件中添加上 Service 的信息：

```
---
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  namespace: blog
spec:
  type: NodePort
  selector:
    app: wordpress
  ports:
  - name: wordpressport
    protocol: TCP
    port: 80
    targetPort: wdport
```

注意要添加属性 type: NodePort，然后重新更新 wordpress.yaml 文件：

```
$ kubectl apply -f wordpress.yaml
service "wordpress" created
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
deployment.apps "wordpress-deploy" configured
```

创建完成后，查看下 svc：

```
$ kubectl get svc -n blog
NAME      TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
mysql    ClusterIP  10.98.27.19 <none>        3306/TCP    25m
wordpress  NodePort   10.101.7.69 <none>        80:32255/TCP 1m
```



Welcome

Welcome to the famous five-minute WordPress installation process! Just fill in the information below and you'll be on your way to using the most extendable and powerful personal publishing platform in the world.

Information needed

Please provide the following information. Don't worry, you can always change these settings later.

Site Title

Username
Usernames can have only alphanumeric characters, spaces, underscores, hyphens, periods, and the @ symbol.

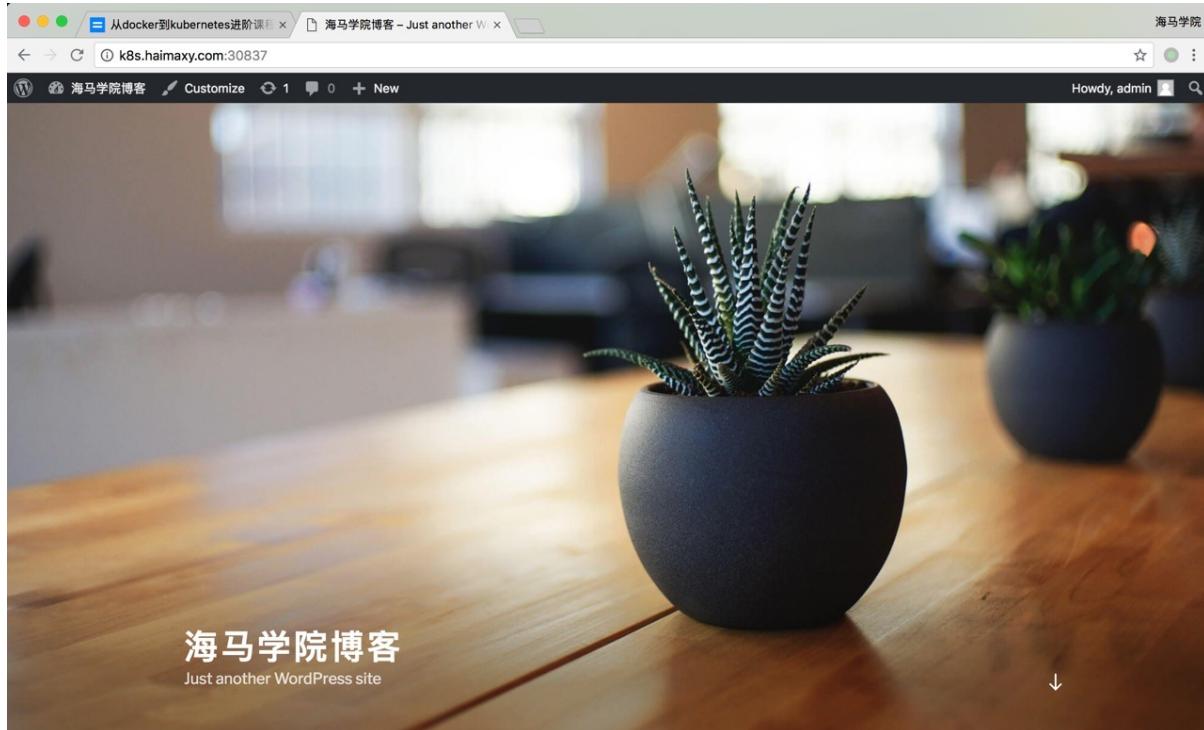
Password *#9V)6VDWn&T*L3Vke ()  Hide
Strong

Important: You will need this password to log in. Please store it in a secure location.

Your Email
Double-check your email address before continuing.

Search Engine Visibility Discourage search engines from indexing this site
It is up to search engines to honor this request.

然后根据页面提示，填上对应的信息，点击“安装”即可，最终安装成功后，我们就可以看到熟悉的首页界面了：



提高稳定性

现在 wordpress 应用已经部署成功了，那么就万事大吉了吗？如果我们的网站访问量突然变大了怎么办，如果我们要更新我们的镜像该怎么办？如果我们的 mysql 服务挂掉了怎么办？

所以要保证我们的网站能够非常稳定的提供服务，我们做得还不够，我们可以通过做些什么事情来提高网站的稳定性呢？

第一. 增加健康检测，我们前面说过 liveness probe 和 readiness probe 是提高应用稳定性非常重要的方法：

```
livenessProbe:
  tcpSocket:
    port: 80
  initialDelaySeconds: 3
  periodSeconds: 3
readinessProbe:
  tcpSocket:
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 10
```

增加上面两个探针，每10s检测一次应用是否可读，每3s检测一次应用是否存活

第二. 增加 HPA，让我们的应用能够自动应对流量高峰期：

```
$ kubectl autoscale deployment wordpress-deploy --cpu-percent=10 --min=1 --max=10 -n blog
deployment "wordpress-deploy" autoscaled
```

我们用 `kubectl autoscale` 命令为我们的 `wordpress-deploy` 创建一个 HPA 对象，最小的 pod 副本数为1，最大为10，HPA 会根据设定的 cpu 使用率（10%）动态的增加或者减少 pod 数量。当然最好我们也为 Pod 声明一些资源限制：

```
resources:
  limits:
    cpu: 200m
    memory: 200Mi
  requests:
    cpu: 100m
    memory: 100Mi
```

更新 Deployment 后，我们可以来测试下上面的 HPA 是否会生效：

```
$ kubectl run -i --tty load-generator --image=busybox /bin/sh
If you don't see a command prompt, try pressing enter.
/ # while true; do wget -q -O- http://10.244.1.62:80; done
```

观察 Deployment 的副本数是否有变化

```
$ kubectl get deployment wordpress-deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
wordpress-deploy   3         3         3           3          4d
```

第三. 增加滚动更新策略，这样可以保证我们在更新应用的时候服务不会被中断：

```
replicas: 2
revisionHistoryLimit: 10
minReadySeconds: 5
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 1
```

第四. 我们知道如果 mysql 服务被重新创建了的话，它的 `clusterIP` 非常有可能就变化了，所以上面我们环境变量中的 `WORDPRESS_DB_HOST` 的值就会有问题，就会导致访问不了数据库服务了，这个地方我们可以直接使用 Service 的名称来代替 host，这样即使 `clusterIP` 变化了，也不会有任何影响，这个我们会在后面的服务发现的章节和大家深入讲解的：

```
env:
- name: WORDPRESS_DB_HOST
  value: mysql:3306
```

第五. 我们在部署 wordpress 服务的时候，mysql 服务以前启动起来了吗？如果没有启动起来是不是我们也没办法连接数据库了啊？该怎么办，是不是在启动 wordpress 应用之前应该去检查一下 mysql 服务，如果服务正常的话我们就开始部署应用了，这是不是就是 `InitContainer` 的用法：

```

initContainers:
- name: init-db
  image: busybox
  command: ['sh', '-c', 'until nslookup mysql; do echo waiting for mysql service; sleep 2; done;']

```

直到 mysql 服务创建完成后， initContainer 才结束，结束完成后我们才开始下面的部署。

最后，我们把部署的应用整合到一个 YAML 文件中来：(wordpress-all.yaml)

```

---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mysql-deploy
  namespace: blog
  labels:
    app: mysql
spec:
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:5.7
          ports:
            - containerPort: 3306
              name: dbport
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: rootPassw0rd
            - name: MYSQL_DATABASE
              value: wordpress
            - name: MYSQL_USER
              value: wordpress
            - name: MYSQL_PASSWORD
              value: wordpress
          volumeMounts:
            - name: db
              mountPath: /var/lib/mysql
      volumes:
        - name: db
          hostPath:
            path: /var/lib/mysql

---
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: blog
spec:
  selector:
    app: mysql
  ports:
    - name: mysqlport

```

```
protocol: TCP
port: 3306
targetPort: dbport

---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: wordpress-deploy
  namespace: blog
  labels:
    app: wordpress
spec:
  revisionHistoryLimit: 10
  minReadySeconds: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      initContainers:
        - name: init-db
          image: busybox
          command: ['sh', '-c', 'until nslookup mysql; do echo waiting for mysql service; sleep 2; done;']
      containers:
        - name: wordpress
          image: wordpress
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
              name: wdport
          env:
            - name: WORDPRESS_DB_HOST
              value: mysql:3306
            - name: WORDPRESS_DB_USER
              value: wordpress
            - name: WORDPRESS_DB_PASSWORD
              value: wordpress
          livenessProbe:
            tcpSocket:
              port: 80
            initialDelaySeconds: 3
            periodSeconds: 3
          readinessProbe:
            tcpSocket:
              port: 80
            initialDelaySeconds: 5
            periodSeconds: 10
      resources:
        limits:
          cpu: 200m
          memory: 200Mi
        requests:
```

```
cpu: 100m
memory: 100Mi

---
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  namespace: blog
spec:
  selector:
    app: wordpress
  type: NodePort
  ports:
  - name: wordpressport
    protocol: TCP
    port: 80
    nodePort: 32255
    targetPort: wdport
```

我们这里主要是针对的 `wordpress` 来做的提高稳定性的方法，如何对 `mysql` 提高一些稳定性呢？大家下去可以试一试，我们接下来会和大家讲解 `mysql` 这类有状态的应用在 `Kubernetes` 当中的使用方法。

最后，我们来把前面我们部署的相关服务全部删掉，重新通过上面的 `YAML` 文件来创建：

```
$ kubectl create -f wordpress-all.yaml
deployment.apps "mysql-deploy" created
service "mysql" created
deployment.apps "wordpress-deploy" created
service "wordpress" created
```

看看最后能不能得到我们的最终成果呢？

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:02:52

DaemonSet 与 StatefulSet 的使用

前面我们的课程中学习了大部分资源对象的使用方法，上节课我们通过一个 WordPress 的示例把我们前面的内容做了一个总结。今天我们来给大家讲解另外一个 Pod 控制器的使用方法，我们前面主要讲解的是 Deployment 这种对象资源的使用，接下来我们要讲解的是在特定场合下使用的控制器： DaemonSet 与 StatefulSet。

DaemonSet 的使用

通过该控制器的名称我们可以看出它的用法：Daemon，就是用来部署守护进程的， DaemonSet 用于在每个 Kubernetes 节点中将守护进程的副本作为后台进程运行，说白了就是在每个节点部署一个 Pod 副本，当节点加入到 Kubernetes 集群中，Pod 会被调度到该节点上运行，当节点从集群只能被移除后，该节点上的这个 Pod 也会被移除，当然，如果我们删除 DaemonSet，所有和这个对象相关的 Pods 都会被删除。

在哪种情况下我们会需要用到这种业务场景呢？其实这种场景还是比较普通的，比如：

- 集群存储守护程序，如 glusterd、ceph 要部署在每个节点上以提供持久性存储；
- 节点监视守护进程，如 Prometheus 监控集群，可以在每个节点上运行一个 node-exporter 进程来收集监控节点的信息；
- 日志收集守护程序，如 fluentd 或 logstash，在每个节点上运行以收集容器的日志

这里需要特别说明的一个就是关于 DaemonSet 运行的 Pod 的调度问题，正常情况下，Pod 运行在哪个节点上是由 Kubernetes 的调度器策略来决定的，然而，由 DaemonSet 控制器创建的 Pod 实际上提前已经确定了在哪个节点上了（Pod 创建时指定了 .spec.nodeName），所以：

- DaemonSet 并不关心一个节点的 unschedulable 字段，这个我们在后面的调度章节和大家讲解的。
- DaemonSet 可以创建 Pod，即使调度器还没有启动，这点非常重要。

下面我们直接使用一个示例来演示下，在每个节点上部署一个 Nginx Pod：(nginx-ds.yaml)

```

kind: DaemonSet
apiVersion: extensions/v1beta1
metadata:
  name: nginx-ds
  labels:
    k8s-app: nginx
spec:
  template:
    metadata:
      labels:
        k8s-app: nginx
    spec:
      containers:
        - image: nginx:1.7.9
          name: nginx
          ports:

```

```
- name: http
  containerPort: 80
```

然后直接创建即可：

```
$ kubectl create -f nginx-ds.yaml
```

然后我们可以观察下 Pod 是否被分布到了每个节点上：

```
$ kubectl get nodes
$ kubectl get pods -o wide
```

StatefulSet 的使用

在学习 StatefulSet 这种控制器之前，我们就得先弄明白一个概念：什么是有状态服务？什么是无状态服务？

- 无状态服务（Stateless Service）：该服务运行的实例不会在本地存储需要持久化的数据，并且多个实例对于同一个请求响应的结果是完全一致的，比如前面我们讲解的 WordPress 实例，我们是不是可以同时启动多个实例，但是我们访问任意一个实例得到的结果都是一样的吧？因为他唯一需要持久化的数据是存储在 MySQL 数据库中的，所以我们可以将 WordPress 这个应用是无状态服务，但是 MySQL 数据库就不是了，因为他需要把数据持久化到本地。
- 有状态服务（Stateful Service）：就和上面的概念是对立的了，该服务运行的实例需要在本地存储持久化数据，比如上面的 MySQL 数据库，你现在运行在节点A，那么他的数据就存储在节点A上面的，如果这个时候你把该服务迁移到节点B去的话，那么就没有之前的数据了，因为他需要去对应的数据目录里面恢复数据，而此时没有任何数据。

现在大家对有状态和无状态有一定的认识了吧，比如我们常见的 WEB 应用，是通过 session 来保持用户的登录状态的，如果我们将 session 持久化到节点上，那么该应用就是一个有状态的服务了，因为我现在登录进来把你我的 session 持久化到节点A上了，下次我登录的时候可能会将请求路由到节点B上去了，但是节点B上根本就没有我当前的 session 数据，就会被认为是未登录状态了，这样就导致我前后两次请求得到的结果不一致了。所以一般为了横向扩展，我们都会把这类 WEB 应用改成无状态的服务，怎么改？将 session 数据存入一个公共的地方，比如 redis 里面，是不是就可以了，对于一些客户端请求 API 的情况，我们就不使用 session 来保持用户状态，改用 token 也是可以的。

无状态服务利用我们前面的 Deployment 或者 RC 都可以很好的控制，对应有状态服务，需要考虑的细节就要多很多了，容器化应用程序最困难的任务之一，就是设计有状态分布式组件的部署体系结构。由于无状态组件可能没有预定义的启动顺序、集群要求、点对点 TCP 连接、唯一的网络标识符、正常的启动和终止要求等，因此可以很容易地进行容器化。诸如数据库，大数据分析系统，分布式 key/value 存储和 message brokers 可能有复杂的分布式体系结构，都可能会用到上述功能。为此，Kubernetes 引入了 StatefulSet 资源来支持这种复杂的需求。

StatefulSet 类似于 ReplicaSet，但是它可以处理 Pod 的启动顺序，为保留每个 Pod 的状态设置唯一标识，同时具有以下功能：

- 稳定的、唯一的网络标识符
- 稳定的、持久化的存储
- 有序的、优雅的部署和缩放
- 有序的、优雅的删除和终止
- 有序的、自动滚动更新

创建StatefulSet

接下来我们来给大家演示下 StatefulSet 对象的使用方法，在开始之前，我们先准备两个1G的存储卷(PV)，在后面的课程中我们也会和大家详细讲解 PV 和 PVC 的使用方法的，这里我们先不深究：(pv001.yaml)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv001
  labels:
    release: stable
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: /tmp/data
```

另外一个只需要把 name 改成 pv002即可，然后创建：

```
$ kubectl create -f pv001.yaml && kubectl create -f pv002.yaml
persistentvolume "pv001" created
persistentvolume "pv002" created
$ kubectl get pv
kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM     STORAGECLASS
REASON    AGE
pv001     1Gi        RWO          Recycle          Available
           12s
pv002     1Gi        RWO          Recycle          Available
           11s
```

可以看到成功创建了两个 PV对象，状态是：Available。

然后我们使用 StatefulSet 来创建一个 Nginx 的 Pod，对于这种类型的资源，我们一般是通过创建一个 Headless Service 类型的服务来暴露服务，将 clusterIP 设置为 None 就是一个无头的服务：
(statefulset-demo.yaml)

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
```

```

ports:
- port: 80
  name: web
clusterIP: None
selector:
  app: nginx
  role: stateful

...
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
        role: stateful
    spec:
      containers:
        - name: nginx
          image: cnych/nginx-slim:0.8
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www
              mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
    - metadata:
        name: www
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi

```

注意上面的 YAML 文件中和 `volumeMounts` 进行关联的是一个新的属性：`volumeClaimTemplates`，该属性会自动声明一个 pvc 对象和 pv 进行管理：

然后这里我们开启两个终端窗口。在第一个终端中，使用 `kubectl get` 来查看 StatefulSet 的 Pods 的创建情况。

```
$ kubectl get pods -w -l role=stateful
```

在另一个终端中，使用 `kubectl create` 来创建定义在 `statefulset-demo.yaml` 中的 Headless Service 和 StatefulSet。

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

检查 Pod 的顺序索引

对于一个拥有 N 个副本的 StatefulSet，Pod 被部署时是按照 {0..N-1} 的序号顺序创建的。在第一个终端中我们可以看到如下的一些信息：

```
$ kubectl get pods -w -l role=stateful
NAME      READY   STATUS    RESTARTS   AGE
web-0     0/1     Pending   0          0s
web-0     0/1     Pending   0          0s
web-0     0/1     ContainerCreating   0          0s
web-0     1/1     Running   0          19s
web-1     0/1     Pending   0          0s
web-1     0/1     Pending   0          0s
web-1     0/1     ContainerCreating   0          0s
web-1     1/1     Running   0          18s
```

请注意在 web-0 Pod 处于 Running 和 Ready 状态后 web-1 Pod 才会被启动。

如同 StatefulSets 概念中所提到的，StatefulSet 中的 Pod 拥有一个具有稳定的、独一无二的身份标志。这个标志基于 StatefulSet 控制器分配给每个 Pod 的唯一顺序索引。Pod 的名称的形式为 <statefulset name>-<ordinal index>。web StatefulSet 拥有两个副本，所以它创建了两个 Pod：web-0 和 web-1。

上面的命令创建了两个 Pod，每个都运行了一个 NGINX web 服务器。获取 nginx Service 和 web StatefulSet 来验证是否成功的创建了它们。

```
$ kubectl get service nginx
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx    None          <none>        80/TCP    12s
$ kubectl get statefulset web
NAME      DESIRED   CURRENT   AGE
web      2          1          20s
```

使用稳定的网络身份标识

每个 Pod 都拥有一个基于其顺序索引的稳定的主机名。使用 kubectl exec 在每个 Pod 中执行 hostname。

```
$ for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
web-0
web-1
```

然后我们使用 kubectl run 运行一个提供 nslookup 命令的容器。通过对 Pod 的主机名执行 nslookup，你可以检查他们在集群内部的 DNS 地址。

```
$ 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: 10.244.1.6

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: 10.244.2.6
```

headless service 的 CNAME 指向 SRV 记录（记录每个 Running 和 Ready 状态的 Pod）。SRV 记录指向一个包含 Pod IP 地址的记录表项。

然后我们再来看下删除 StatefulSet 下面的 Pod：

在一个终端中查看 StatefulSet 的 Pod：

```
$ kubectl get pod -w -l role=stateful
```

在另一个终端中使用 kubectl delete 删除 StatefulSet 中所有的 Pod。

```
$ kubectl delete pod -l role=stateful
pod "web-0" deleted
pod "web-1" deleted
```

等待 StatefulSet 重启它们，并且两个 Pod 都变成 Running 和 Ready 状态。

```
$ kubectl get pod -w -l app=nginx
NAME      READY   STATUS            RESTARTS   AGE
web-0     0/1     ContainerCreating   0          0s
web-0     1/1     Running    0          2s
web-1     0/1     Pending    0          0s
web-1     0/1     Pending    0          0s
web-1     0/1     ContainerCreating   0          0s
web-1     1/1     Running    0          34s
```

然后再次使用 kubectl exec 和 kubectl run 查看 Pod 的主机名和集群内部的 DNS 表项。

```
$ for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
web-0
web-1
$ 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: 10.244.1.7

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: 10.244.2.8
```

我们可以看到Pod 的序号、主机名、SRV 条目和记录名称没有改变，但和 Pod 相关联的 IP 地址可能会发生改变。所以说这就是为什么不要在其他应用中使用 StatefulSet 中的 Pod 的 IP 地址进行连接，这点很重要。一般情况下我们直接通过 SRV 记录连接就行：web-0.nginx、web-1.nginx，因为他们是稳定的，并且当你的 Pod 的状态变为 Running 和 Ready 时，你的应用就能够发现它们的地址。

同样我们可以查看 PV、PVC的最终绑定情况：

```
$ kubectl get pv
NAME      CAPACITY   ACCESS MODES  RECLAIM POLICY  STATUS    CLAIM
CLASS     REASON     AGE
pv001     1Gi        RWO          Recycle        Bound     default/www-web-0
                    1h
pv002     1Gi        RWO          Recycle        Bound     default/www-web-1
                    1h
$ kubectl get pvc
NAME      STATUS    VOLUME   CAPACITY  ACCESS MODES  STORAGECLASS  AGE
www-web-0  Bound    pv001    1Gi       RWO          22m
www-web-1  Bound    pv002    1Gi       RWO          22m
```

我们会在下节课开始和大家讲解存储卷的使用的，所以这里我们先暂时不展开讲解了，避免大家糊涂。

当然 StatefulSet 还拥有其他特性，在实际的项目中，我们还是很少回去直接通过 StatefulSet 来部署我们的有状态服务的，除非你自己能够完全能够 hold 住，对于一些特定的服务，我们可能会使用更加高级的 Operator 来部署，比如 etcd-operator、prometheus-operator 等等，这些应用都能够很好的来管理有状态的服务，而不是单纯的使用一个 StatefulSet 来部署一个 Pod就行，因为对于有状态的应用最重要的还是数据恢复、故障转移等等。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:03:02

PV 的使用

前面我们和大家一起学习了一些基本的资源对象的使用方法，前面我们也和大家讲到了有状态的应用和对数据有持久化的应用，我们有通过 hostPath 或者 emptyDir 的方式来持久化我们的数据，但是显然我们还需要更加可靠的存储来保存应用的持久化数据，这样容器在重建后，依然可以使用之前的数据。但是显然存储资源和 CPU 资源以及内存资源有很大不同，为了屏蔽底层的技术实现细节，让用户更加方便的使用，Kubernetes 便引入了 PV 和 PVC 两个重要的资源对象来实现对存储的管理。这也是我们这节课需要和大家讲解的核心：PV 和 PVC。

概念

PV 的全称是：PersistentVolume（持久化卷），是对底层的共享存储的一种抽象，PV 由管理员进行创建和配置，它和具体的底层的共享存储技术的实现方式有关，比如 Ceph、GlusterFS、NFS 等，都是通过插件机制完成与共享存储的对接。

PVC 的全称是：PersistentVolumeClaim（持久化卷声明），PVC 是用户存储的一种声明，PVC 和 Pod 比较类似，Pod 消耗的是节点，PVC 消耗的是 PV 资源，Pod 可以请求 CPU 和内存，而 PVC 可以请求特定的存储空间和访问模式。对于真正使用存储的用户不需要关心底层的存储实现细节，只需要直接使用 PVC 即可。

但是通过 PVC 请求到一定的存储空间也很有可能不足以满足应用对于存储设备的各种需求，而且不同的应用程序对于存储性能的要求可能也不尽相同，比如读写速度、并发性能等，为了解决这一问题，Kubernetes 又为我们引入了一个新的资源对象：StorageClass，通过 StorageClass 的定义，管理员可以将存储资源定义为某种类型的资源，比如快速存储、慢速存储等，用户根据 StorageClass 的描述就可以非常直观的知道各种存储资源的具体特性了，这样就可以根据应用的特性去申请合适的存储资源了。

NFS

我们这里为了演示方便，决定使用相对简单的 NFS 这种存储资源，接下来我们在节点10.151.30.57上来安装 NFS 服务，数据目录：/data/k8s/

1. 关闭防火墙

```
$ systemctl stop firewalld.service
$ systemctl disable firewalld.service
```

2. 安装配置 nfs

```
$ yum -y install nfs-utils rpcbind
```

共享目录设置权限：

```
$ chmod 755 /data/k8s/
```

配置 nfs, nfs 的默认配置文件在 /etc/exports 文件下, 在该文件中添加下面的配置信息:

```
$ vi /etc/exports
/data/k8s * (rw,sync,no_root_squash)
```

配置说明:

- /data/k8s: 是共享的数据目录
- * : 表示任何人都有权限连接, 当然也可以是一个网段, 一个 IP, 也可以是域名
- rw: 读写的权限
- sync: 表示文件同时写入硬盘和内存
- no_root_squash: 当登录 NFS 主机使用共享目录的使用者是 root 时, 其权限将被转换成为匿名使用者, 通常它的 UID 与 GID, 都会变成 nobody 身份

当然 nfs 的配置还有很多, 感兴趣的同学可以在网上去查找一下。

1. 启动服务 nfs 需要向 rpc 注册, rpc 一旦重启了, 注册的文件都会丢失, 向他注册的服务都需要重启

注意启动顺序, 先启动 rpcbind

```
$ systemctl start rpcbind.service
$ systemctl enable rpcbind
$ systemctl status rpcbind
● rpcbind.service - RPC bind service
  Loaded: loaded (/usr/lib/systemd/system/rpcbind.service; disabled; vendor preset: enabled)
  Active: active (running) since Tue 2018-07-10 20:57:29 CST; 1min 54s ago
    Process: 17696 ExecStart=/sbin/rpcbind -w $RPCBIND_ARGS (code=exited, status=0/SUCCESS)
   Main PID: 17697 (rpcbind)
     Tasks: 1
    Memory: 1.1M
   CGroup: /system.slice/rpcbind.service
           └─17697 /sbin/rpcbind -w

Jul 10 20:57:29 master systemd[1]: Starting RPC bind service...
Jul 10 20:57:29 master systemd[1]: Started RPC bind service.
```

看到上面的 Started 证明启动成功了。

然后启动 nfs 服务:

```
$ systemctl start nfs.service
$ systemctl enable nfs
$ systemctl status nfs
● nfs-server.service - NFS server and services
  Loaded: loaded (/usr/lib/systemd/system/nfs-server.service; enabled; vendor preset: disabled)
  Drop-In: /run/systemd/generator/nfs-server.service.d
            └─order-with-mounts.conf
  Active: active (exited) since Tue 2018-07-10 21:35:37 CST; 14s ago
    Main PID: 32067 (code=exited, status=0/SUCCESS)
   CGroup: /system.slice/nfs-server.service
```

```
Jul 10 21:35:37 master systemd[1]: Starting NFS server and services...
Jul 10 21:35:37 master systemd[1]: Started NFS server and services.
```

同样看到 Started 则证明 NFS Server 启动成功了。

另外我们还可以通过下面的命令确认下：

```
$ rpcinfo -p | grep nfs
 100003    3  tcp   2049  nfs
 100003    4  tcp   2049  nfs
 100227    3  tcp   2049  nfs_acl
 100003    3  udp   2049  nfs
 100003    4  udp   2049  nfs
 100227    3  udp   2049  nfs_acl
```

查看具体目录挂载权限：

```
$ cat /var/lib/nfs/etab
/data/k8s *(rw,sync,wdelay,hide,nocrossmnt,secure,no_root_squash,no_all_squash,no_subtree_check,secure_locks,acl,no_pnfs,anonuid=65534,anongid=65534,sec=sys,secure,no_root_squash,no_all_squash)
```

到这里我们就把 nfs server 给安装成功了，接下来我们在节点10.151.30.62上来安装 nfs 的客户端来验证下 nfs

1. 安装 nfs 当前也需要先关闭防火墙：

```
$ systemctl stop firewalld.service
$ systemctl disable firewalld.service
```

然后安装 nfs

```
$ yum -y install nfs-utils rpcbind
```

安装完成后，和上面的方法一样，先启动 rpc、然后启动 nfs：

```
$ systemctl start rpcbind.service
$ systemctl enable rpcbind.service
$ systemctl start nfs.service
$ systemctl enable nfs.service
```

1. 挂载数据目录 客户端启动完成后，我们在客户端来挂载下 nfs 测试下：

首先检查下 nfs 是否有共享目录：

```
$ showmount -e 10.151.30.57
Export list for 10.151.30.57:
/data/k8s *
```

然后我们在客户端上新建目录：

```
$ mkdir -p /root/course/kubeadm/data
```

将 nfs 共享目录挂载到上面的目录：

```
$ mount -t nfs 10.151.30.57:/data/k8s /root/course/kubeadm/data
```

挂载成功后，在客户端上面的目录中新建一个文件，然后我们观察下 nfs 服务端的共享目录下面是否也会出现该文件：

```
$ touch /root/course/kubeadm/data/test.txt
```

然后在 nfs 服务端查看：

```
$ ls -ls /data/k8s/
total 4
4 -rw-r--r--. 1 root root 4 Jul 10 21:50 test.txt
```

如果上面出现了 test.txt 的文件，那么证明我们的 nfs 挂载成功了。

PV

有了上面的 NFS 共享存储，下面我们就来使用 PV 和 PVC 了。PV 作为存储资源，主要包括存储能力、访问模式、存储类型、回收策略等关键信息，下面我们来新建一个 PV 对象，使用 nfs 类型的后端存储，1G 的存储空间，访问模式为 ReadWriteOnce，回收策略为 Recycle，对应的 YAML 文件如下：(pv1-demo.yaml)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /data/k8s
    server: 10.151.30.57
```

Kubernetes 支持的 PV 类型有很多，比如常见的 Ceph、GlusterFs、NFS，甚至 HostPath 也可以，不过 HostPath 我们之前也说过仅仅可以用于单机测试，更多的支持类型可以前往 [Kubernetes PV 官方文档](#) 进行查看，因为每种存储类型都有各自的特点，所以我们在使用的时候可以去查看相应的文档来设置对应的参数。

然后同样的，直接使用 kubectl 创建即可：

```
$ kubectl create -f pv1-demo.yaml
persistentvolume "pv1" created
$ kubectl get pv
NAME      CAPACITY   ACCESS MODES  RECLAIM POLICY  STATUS        CLAIM           STORA
GECLASS   REASON     AGE
pv1       1Gi        RWO          Recycle        Available
                    12s
```

我们可以看到 pv1 已经创建成功了，状态是 Available，表示 pv1 就绪，可以被 PVC 申请。我们来分别对上面的属性进行一些解读。

Capacity (存储能力)

一般来说，一个 PV 对象都要指定一个存储能力，通过 PV 的 capacity 属性来设置的，目前只支持存储空间的设置，就是我们这里的 storage=1Gi，不过未来可能会加入 IOPS、吞吐量等指标的配置。

AccessModes (访问模式)

AccessModes 是用来对 PV 进行访问模式的设置，用于描述用户应用对存储资源的访问权限，访问权限包括下面几种方式：

- ReadWriteOnce (RWO)：读写权限，但是只能被单个节点挂载
- ReadOnlyMany (ROX)：只读权限，可以被多个节点挂载
- ReadWriteMany (RWX)：读写权限，可以被多个节点挂载

注意：一些 PV 可能支持多种访问模式，但是在挂载的时候只能使用一种访问模式，多种访问模式是不会生效的。

下图是一些常用的 Volume 插件支持的访问模式：

Volume Plugin	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
FC	✓	✓	-
FlexVolume	✓	✓	-
Flocker	✓	-	-
GCEPersistentDisk	✓	✓	-
Glusterfs	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
Quobyte	✓	✓	✓
NFS	✓	✓	✓
RBD	✓	✓	-
VsphereVolume	✓	-	- (works when pods are collocated)
PortworxVolume	✓	-	✓
ScaleIO	✓	✓	-
StorageOS	✓	-	-

persistentVolumeReclaimPolicy (回收策略)

我这里指定的 PV 的回收策略为 Recycle，目前 PV 支持的策略有三种：

- Retain (保留) - 保留数据，需要管理员手工清理数据
- Recycle (回收) - 清除 PV 中的数据，效果相当于执行 `rm -rf /thevolume/*`
- Delete (删除) - 与 PV 相连的后端存储完成 volume 的删除操作，当然这常见于云服务商的存储服务，比如 ASW EBS。

不过需要注意的是，目前只有 NFS 和 HostPath 两种类型支持回收策略。当然一般来说还是设置为 Retain 这种策略保险一点。

状态

一个 PV 的生命周期中，可能会处于4中不同的阶段：

- Available (可用) : 表示可用状态, 还未被任何 PVC 绑定
- Bound (已绑定) : 表示 PVC 已经被 PVC 绑定
- Released (已释放) : PVC 被删除, 但是资源还未被集群重新声明
- Failed (失败) : 表示该 PV 的自动回收失败

这就是 PV 的声明方法, 下节课我们来和大家一起学习下 PVC 的使用方法。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号, 在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

PVC 的使用

上节课我们学习了 PV 的使用，但是在我们真正使用的时候是使用的 PVC，就类似于我们的服务是通过 Pod 来运行的，而不是 Node，只是 Pod 跑在 Node 上而已，所以这节课我们就来给大家讲解下 PVC 的使用方法。

准备工作

在使用 PVC 之前，我们还得把其他节点上的 nfs 客户端给安装上，比如我们这里：

```
$ kubectl get nodes
NAME      STATUS    ROLES      AGE       VERSION
master    Ready     master     61d      v1.10.0
node01   Ready     <none>    61d      v1.10.0
node03   Ready     <none>    41d      v1.10.0
```

我们需要在所有节点安装 nfs 客户端程序，安装方法和上节课的安装方法一样的。必须在所有节点都安装 nfs 客户端，否则可能会导致 PV 挂载不上的问题

新建 PVC

同样的，我们来新建一个数据卷声明，我们来请求 1Gi 的存储容量，访问模式也是 ReadWriteOnce，YAML 文件如下：(pvc-nfs.yaml)

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-nfs
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

我们可以看到我们这里的声明方法几乎和新建 PV 是一样的，在新建 PVC 之前，我们可以看下之前创建的 PV 的状态：

```
kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS        CLAIM
GECLASS   REASON     AGE
pv-nfs    1Gi        RWO          Recycle        Available
```

我们可以看到当前 pv-nfs 是在 Available 的一个状态，所以这个时候我们的 PVC 可以和这个 PV 进行绑定：

```
$ kubectl create -f pvc-nfs.yaml
persistentvolumeclaim "pvc-nfs" created
$ kubectl get pvc
NAME      STATUS    VOLUME   CAPACITY  ACCESS MODES  STORAGECLASS  AGE
pvc-nfs   Bound     pv-nfs   1Gi       RWO          default        12s
```

我们可以看到 pvc-nfs 创建成功了，状态是 Bound 状态了，这个时候我们再看下 PV 的状态呢：

```
$ kubectl get pv
NAME      CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS    CLAIM           STORAGE
CLASS     REASON    AGE
pv-nfs    1Gi       RWO          Recycle        Bound     default/pvc-nfs
                    23m
```

同样我们可以看到 PV 也是 Bound 状态了，对应的声明是 default/pvc-nfs，就是 default 命名空间下面的 pvc-nfs，证明我们刚刚新建的 pvc-nfs 和我们的 pv-nfs 绑定成功了。

有的同学可能会觉得很奇怪，我们并没有在 pvc-nfs 中指定关于 pv 的什么标志，它们之间是怎么就关联起来了的呢？其实这是系统自动帮我们去匹配的，他会根据我们的声明要求去查找处于 Available 状态的 PV，如果没有找到的话那么我们的 PVC 就会一直处于 Pending 状态，找到了的话当然就会把当前的 PVC 和目标 PV 进行绑定，这个时候状态就会变成 Bound 状态了。比如我们新建一个 PVC，如下：(pvc2-nfs.yaml)

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc2-nfs
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
  selector:
    matchLabels:
      app: nfs
```

我们这里声明一个 PV 资源的请求，邀请访问模式是 ReadWriteOnce，存储容量是 2Gi，最后我们还要求匹配具有标签 app=nfs 的 PV，这样要求的 PV 有吗？我们先查看下当前系统的所有 PV：

```
$ kubectl get pv
NAME      CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS    CLAIM           STORAGE
CLASS     REASON    AGE
pv-nfs    1Gi       RWO          Recycle        Bound     default/pvc-nfs
                    43m
pv001    1Gi       RWO          Recycle        Bound     default/www-web-0
                    13d
pv002    1Gi       RWO          Recycle        Bound     default/www-web-1
                    13d
```

都是 Bound 状态，并没有 Available 状态的 PV，所以我们可以想象到我们上面新建的 PVC 是没办法选择到合适的 PV 的，我们创建一下看看：

```
$ kubectl create -f pvc2-nfs.yaml
persistentvolumeclaim "pvc2-nfs" created
$ kubectl get pvc
NAME      STATUS    VOLUME   CAPACITY  ACCESS MODES  STORAGECLASS  AGE
pvc-nfs   Bound     pv-nfs    1Gi       RWO          -           23m
pvc2-nfs  Pending   -         -          -            -           14s
```

很显然是 Pending 状态，因为并没有合适的 PV 给你使用，现在我们来新建一个 PV，让上面的 PVC 有合适的 PV 使用：(pv2-nfs.yaml)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv2-nfs
  labels:
    app: nfs
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    server: 10.151.30.57
    path: /data/k8s
```

我们这里新建一个名为 pv2-nfs 的 PV，具有标签 app=nfs，容量也是 2Gi，访问模式是 ReadWraiteOnce，看上去这一切都很适合上面的 PVC，新建试一试：

```
$ kubectl create -f pv2-nfs.yaml
persistentvolume "pv2-nfs" created
$ kubectl get pv
NAME      CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS     CLAIM
CLASS     REASON    AGE
pv-nfs    1Gi       RWO          Recycle        Bound      default/pvc-nfs
                    51m
pv2-nfs   2Gi       RWO          Recycle        Bound      default/pvc2-nfs
                    12s
```

创建完 pv2-nfs 后，是不是很快就发现该 PV 是 Bound 状态了，对应的 PVC 是 default/pvc2-nfs，证明上面的 pvc2-nfs 终于找到合适的 PV 进行绑定上了：

```
$ kubectl get pvc
kubectl get pvc
NAME      STATUS    VOLUME   CAPACITY  ACCESS MODES  STORAGECLASS  AGE
pvc-nfs   Bound     pv-nfs    1Gi       RWO          -           30m
pvc2-nfs  Bound     pv2-nfs   2Gi       RWO          -           7m
```

成功了，对吧！有的同学可能又会说了，我们的 pv2-nfs 声明的容量是 2Gi，如果我 pvc2-nfs 这里声明的容量是 1Gi 的话呢？还能正常绑定吗？如果可以正常绑定的话，那剩下的 1Gi 容量还能使用吗？其实我也不清楚，怎么办？我们去实际测试下就知道了吧，先删除上面的 pvc2-nfs，然后我们把该 PVC 里面的容量改成 1Gi，再新建试一试呢：

```
$ kubectl delete pvc pvc2-nfs
persistentvolumeclaim "pvc2-nfs" deleted
$ cat pvc2-nfs.yaml
...
resources:
  requests:
    storage: 1Gi
...
$ kubectl create -f pvc2-nfs.yaml
persistentvolumeclaim "pvc2-nfs" created
$ kubectl get pvc
NAME      STATUS      VOLUME      CAPACITY      ACCESS MODES      STORAGECLASS      AGE
pvc2-nfs   Bound       pv2-nfs    2Gi          RWO

```

我们可以看到上面的 PVC 依然可以正常的绑定，仔细看 CAPACITY 这一列的数据：2Gi，也就是说我们声明的 1Gi 是没什么用的，我 PV 是 2Gi，你这里声明 1Gi 是不行的，你必须得使用 2Gi。

如果我们这里容量声明是 3Gi 呢？还可以正常绑定吗？大家可以思考一下，如果声明的容量大于了 PV 里面的容量的话，是没办法进行绑定的，大家可以下去自己测试一下。

使用 PVC

上面我们已经知道怎么创建 PV 和 PVC 了，现在我们就来使用下我们的 PVC，这里我们同样使用之前的 nginx 的镜像来测试下：(nfs-pvc-deploy.yaml)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nfs-pvc
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nfs-pvc
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
              name: web
      volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumes:
    - name: www
      persistentVolumeClaim:
        claimName: pvc2-nfs
---
apiVersion: v1
kind: Service
```

```

metadata:
  name: nfs-pvc
  labels:
    app: nfs-pvc
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: web
  selector:
    app: nfs-pvc

```

我们这里使用 nginx 镜像，将容器的 /usr/share/nginx/html 目录通过 volume 挂载到名为 pvc2-nfs 的 PVC 上面，然后创建一个 NodePort 类型的 Service 来暴露服务：

```

$ kubectl create -f nfs-pvc-deploy.yaml
deployment.extensions "nfs-pvc" created
service "nfs-pvc" created
$ kubectl get pods
kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
...
nfs-pvc-57c9945bd9-5r4r6          1/1     Running   0          19s
nfs-pvc-57c9945bd9-gz6p9          1/1     Running   0          19s
nfs-pvc-57c9945bd9-x6mvc          1/1     Running   0          19s
...
$ kubectl get svc
kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
...
nfs-pvc   NodePort   10.98.246.155   <none>        80:30769/TCP   1m
...

```

然后我们就可以通过任意节点的 IP:30769 端口来访问我们这里的 Nginx 服务了，但是这个时候我们来访问会出现 403，这是为什么？我们再去看看 nfs 共享数据目录下面有没有数据呢？



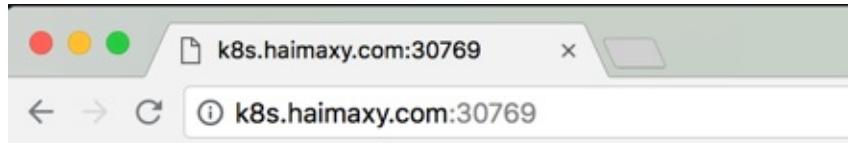
nginx 403

```
$ ls /data/k8s
```

我们发现并没有任何数据，这是因为我们把容器目录 /user/share/nginx/html 和挂载到了 pvc2-nfs 这个 PVC 上面，这个 PVC 就是对应着我们上面的 nfs 的共享数据目录的，该目录下面还没有任何数据，所以我们访问就出现了 403，现在我们在 /data/k8s 这个目录下面新建一个 index.html 的文件：

```
$ echo "<h1>Hello Kubernetes~</h1>" >> /data/k8s/index.html
$ ls /data/k8s/
index.html
```

我们可以看到共享数据目录中已经有一个 index.html 的文件了，由于我们挂载了 pvc2-nfs 到上面的 nginx 容器中去，是不是这个时候容器目录/usr/share/nginx/html下面也有index.html这个文件了啊？所以这个时候我们再来访问下服务，任一节点IP:30769：



Hello Kubernetes~

nginx 200

现在是不是正常了啊，但是我们可以看到我们容器中的数据是直接放到共享数据目录根目录下面的，如果以后我们又有一个新的 nginx 容器也做了数据目录的挂载，是不是就会有冲突了啊，所以这个时候就不太好区分了，这个时候我们可以在 Pod 中使用一个新的属性：subPath，该属性可以来解决这个问题，我们只需要更改上面的 Pod 的 YAML 文件即可：

```
...
volumeMounts:
- name: www
  subPath: nginxpvc-test
  mountPath: /usr/share/nginx/html
...

```

更改完 YAML 文件后，我们重新更新即可：

```
$ kubectl apply -f nfs-pvc-deploy.yaml
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
deployment.extensions "nfs-pvc" configured
Warning: kubectl apply should be used on resource created by either kubectl create --save-config or kubectl apply
service "nfs-pvc" configured
```

更新完后，我们再去看看 nfs 的数据共享目录：

```
$ ls /data/k8s/
index.html  nginxpvc-test
$ ls /data/k8s/nginxpvc-test/
```

我们可以预想到现在我们访问上面的服务，是不是又会得到403的结果啊，因为nginxpvc-test目录下面还没有任何文件呢，我们把根目录下面的 index.html 文件移动到 nginxpvc-test 目录下面去是不是又可以访问了：

```
$ mv /data/k8s/index.html /data/k8s/nginxpvc-test/
```

现在快去验证下吧，看看能不能得到正确结果。

到这里我们就算完整的使用了一次 PVC 了，现在我们再来验证下我们的数据是否会丢失，怎么验证？首先我们把上面的 Deployment 删除掉，这样是不是他下面管理的3个 Pod 也会被一起删除掉啊：

```
$ kubectl delete deployment nfs-pvc
deployment.extensions "nfs-pvc" deleted
```

Deployment 被删除掉了，但是 nfs 的数据共享目录下面的数据呢？

```
$ ls /data/k8s/nginxpvc-test/
index.html
```

还在吧？当然了如果不在了，我们用他就没有任何意义了吧，现在我们再来重新创建上面的 Deployment，看看访问服务还能得到上面的正常输出结果吗：

```
$ kubectl create -f nfs-pvc-deploy.yaml
deployment.extensions "nfs-pvc" created
Error from server (AlreadyExists): error when creating "nfs-pvc-deploy.yaml": services "nfs-pvc" already exists
```

可以看到 nfs-pvc 这个 Deployment 创建成功了，由于 Service 我们之前没有删除掉，所以这里提示已经存在，我们忽略就可以了，现在同样我们用任一节点 IP:30769 来访问我们这里的 service，是不是依然可以在页面上看到Hello Kubernetes~这里的输出信息啊，这证明我们的数据持久化是成功的吧！

注意事项

上面我们演示了数据持久化，如果这个时候我们把 PV 给删除了，上面持久化的数据还会存在吗？如果是删除的 PVC 呢？在实际使用的工程中，是很有可能出现这种情况的吧？下面我们来实际验证下。

我们先删除上面使用的 PV：

```
$ kubectl delete pv pv2-nfs
persistentvolume "pv2-nfs" deleted
```

然后再看看之前创建的 PVC 还会存在吗：

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
...						
pvc2-nfs	Bound	pv2-nfs	2Gi	RWO		1h

是不是觉得很奇怪，pvc2-nfs 仍然是 Bound 的状态，也就意味着我们还可以正常使用这个 PVC，但是如果我们有一个新的 Pod 来使用这个 PVC 会是怎样的情况呢？大家下去自己验证下

如有 Pod 正在使用此 pvc2-nfs 这个 PVC 的话，那么新建的 Pod 则仍可使用，如无 Pod 使用，则创建 Pod 挂载此 PVC 时会出现失败。大家自己去验证下吧

现在我们在恢复到最开始的状态，把 PV 和 PVC 添加回来，如果现在我们把使用 pvc2-nfs 关联的 Pod 都删除，然后再删除该 PVC 的话，那么我们的持久化数据还存在吗？

```
$ kubectl delete -f nfs-pvc-deploy.yaml
deployment.extensions "nfs-pvc" deleted
service "nfs-pvc" deleted
$ kubectl get pvc
NAME      STATUS      VOLUME      CAPACITY      ACCESS MODES      STORAGECLASS      AGE
...
pvc2-nfs   Bound       pv2-nfs    2Gi          RWO
...
$ kubectl delete pvc pvc2-nfs
persistentvolumeclaim "pvc2-nfs" deleted
$ kubectl get pv
NAME      CAPACITY      ACCESS MODES      RECLAIM POLICY      STATUS      CLAIM      STORAGECLASS      AGE
ECLASS     REASON      AGE
...
pv2-nfs   2Gi          RWO          Recycle          Released   default/pvc2-nfs
...
$ ls /data/k8s/
```

我们可以看到 pv2-nfs 这个 PV 的状态已经变成了 Released 状态了，这个状态是不是表示 PVC 已经被释放了，现在可以被重新绑定了，由于我们设置的 PV 的回收策略是 Recycle，所以我们可以很明显的发现 nfs 的共享数据目录下面已经没有了数据了，这是因为我们把 PVC 给删除掉了，然后回收了数据。

不过大家要注意，并不是所有的存储后端的表现结果都是这样的，我们这里使用的是 nfs，其他存储后端肯能会有不一样的结果。

大家在使用 PV 和 PVC 的时候一定要注意这些细节，不然一不小心就把数据搞丢了。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-04-24 15:32:07

StorageClass

前面的课程中我们学习了 PV 和 PVC 的使用方法，但是前面的 PV 都是静态的，什么意思？就是我要使用的一个 PVC 的话就必须手动去创建一个 PV，我们也说过这种方式在很大程度上并不能满足我们的需求，比如我们有一个应用需要对存储的并发度要求比较高，而另外一个应用对读写速度又要求比较高，特别是对于 StatefulSet 类型的应用简单的来使用静态的 PV 就很不合适了，这种情况下我们就需要用到动态 PV，也就是我们今天要讲解的 StorageClass。

创建

要使用 StorageClass，我们就得安装对应的自动配置程序，比如我们这里存储后端使用的是 nfs，那么我们就需要使用到一个 nfs-client 的自动配置程序，我们也叫它 Provisioner，这个程序使用我们已经配置好的 nfs 服务器，来自动创建持久卷，也就是自动帮我们创建 PV。

- 自动创建的 PV 以 \${namespace}-\${pvcName}-\${pvName} 这样的命名格式创建在 NFS 服务器上的共享数据目录中
- 而当这个 PV 被回收后会以 archived-\${namespace}-\${pvcName}-\${pvName} 这样的命名格式存在 NFS 服务器上。

当然在部署 nfs-client 之前，我们需要先成功安装上 nfs 服务器，前面的课程中我们已经过了，服务地址是 10.151.30.57，共享数据目录是 /data/k8s/，然后接下来我们部署 nfs-client 即可，我们也可以直接参考 nfs-client 的文档：<https://github.com/kubernetes-incubator/external-storage/tree/master/nfs-client>，进行安装即可。

第一步：配置 Deployment，将里面的对应的参数替换成我们自己的 nfs 配置（nfs-client.yaml）

```

kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: nfs-client-provisioner
spec:
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
  spec:
    serviceAccountName: nfs-client-provisioner
    containers:
      - name: nfs-client-provisioner
        image: quay.io/external_storage/nfs-client-provisioner:latest
        volumeMounts:
          - name: nfs-client-root
            mountPath: /persistentvolumes
    env:
      - name: PROVISIONER_NAME
        value: fuseim.pri/ifs
      - name: NFS_SERVER

```

```

    value: 10.151.30.57
  - name: NFS_PATH
    value: /data/k8s
  volumes:
  - name: nfs-client-root
    nfs:
      server: 10.151.30.57
      path: /data/k8s

```

第二步：将环境变量 NFS_SERVER 和 NFS_PATH 替换，当然也包括下面的 nfs 配置，我们可以看到我们这里使用了一个名为 nfs-client-provisioner 的 serviceAccount，所以我们也需要创建一个 sa，然后绑定上对应的权限：(nfs-client-sa.yaml)

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: nfs-client-provisioner

---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-client-provisioner-runner
rules:
- apiGroups: [""]
  resources: ["persistentvolumes"]
  verbs: ["get", "list", "watch", "create", "delete"]
- apiGroups: [""]
  resources: ["persistentvolumeclaims"]
  verbs: ["get", "list", "watch", "update"]
- apiGroups: ["storage.k8s.io"]
  resources: ["storageclasses"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["events"]
  verbs: ["list", "watch", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["endpoints"]
  verbs: ["create", "delete", "get", "list", "watch", "patch", "update"]

---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: run-nfs-client-provisioner
subjects:
- kind: ServiceAccount
  name: nfs-client-provisioner
  namespace: default
roleRef:
  kind: ClusterRole
  name: nfs-client-provisioner-runner
  apiGroup: rbac.authorization.k8s.io

```

我们这里新建的一个名为 nfs-client-provisioner 的 ServiceAccount，然后绑定了一个名为 nfs-client-provisioner-runner 的 ClusterRole，而该 ClusterRole 声明了一些权限，其中就包括对 persistentvolumes 的增、删、改、查等权限，所以我们可以利用该 ServiceAccount 来自动创建 PV。

第三步：nfs-client 的 Deployment 声明完成后，我们就可以来创建一个 StorageClass 对象了：(nfs-client-class.yaml)

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: course-nfs-storage
provisioner: fuseim.pri/ifs # or choose another name, must match deployment's env PROVISIONER_NAME'
```

我们声明了一个名为 course-nfs-storage 的 StorageClass 对象，注意下面的 provisioner 对应的值一定要和上面的 Deployment 下面的 PROVISIONER_NAME 这个环境变量的值一样。

现在我们来创建这些资源对象吧：

```
$ kubectl create -f nfs-client.yaml
$ kubectl create -f nfs-client-sa.yaml
$ kubectl create -f nfs-client-class.yaml
```

创建完成后查看下资源状态：

NAME	READY	STATUS	RESTARTS	A
...				
nfs-client-provisioner-7648b664bc-7f9pk	1/1	Running	0	7
...				
NAME	PROVISIONER	AGE		
course-nfs-storage	fuseim.pri/ifs	11s		

新建

上面把 StorageClass 资源对象创建成功了，接下来我们来通过一个示例测试下动态 PV，首先创建一个 PVC 对象：(test-pvc.yaml)

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-pvc
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
```

```
storage: 1Mi
```

我们这里声明了一个 PVC 对象，采用 `ReadWriteMany` 的访问模式，请求 `1Mi` 的空间，但是我们可以看到上面的 PVC 文件我们没有标识出任何和 `StorageClass` 相关联的信息，那么如果我们现在直接创建这个 PVC 对象能够自动绑定上合适的 PV 对象吗？显然是不能的(前提是没有任何合适的 PV)，我们这里有两种方法可以来利用上面我们创建的 `StorageClass` 对象来自动帮我们创建一个合适的 PV：

- 第一种方法：在这个 PVC 对象中添加一个声明 `StorageClass` 对象的标识，这里我们可以利用一个 `annotations` 属性来标识，如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-pvc
  annotations:
    volume.beta.kubernetes.io/storage-class: "course-nfs-storage"
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
```

- 第二种方法：我们可以设置这个 `course-nfs-storage` 的 `StorageClass` 为 Kubernetes 的默认存储后端，我们可以用 `kubectl patch` 命令来更新：

```
$ kubectl patch storageclass course-nfs-storage -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

上面这两种方法都是可以的，当然为了不影响系统的默认行为，我们这里还是采用第一种方法，直接创建即可：

```
$ kubectl create -f test-pvc.yaml
persistentvolumeclaim "test-pvc" created
$ kubectl get pvc
NAME      STATUS      VOLUME          CAPACITY   ACCESS MODES
STORAGECLASS      AGE
...
test-pvc      Bound      pvc-73b5ffd2-8b4b-11e8-b585-525400db4df7   1Mi      RWX
course-nfs-storage      2m
...
```

我们可以看到一个名为 `test-pvc` 的 PVC 对象创建成功了，状态已经是 `Bound` 了，是不是也产生了一个对应的 VOLUME 对象，最重要的一栏是 `STORAGECLASS`，现在是不是也有值了，就是我们刚刚创建的 `StorageClass` 对象 `course-nfs-storage`。

然后查看下 PV 对象呢：

```
$ kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STAT
US       CLAIM      STORAGECLASS      REASON      AGE
...
```

```

...
pvc-73b5ffd2-8b4b-11e8-b585-525400db4df7  1Mi          RWX        Delete      Bound
d       default/test-pvc   course-nfs-storage           8m
...

```

可以看到是不是自动生成了一个关联的 PV 对象，访问模式是 RWX，回收策略是 Delete，这个 PV 对象并不是我们手动创建的吧，这是通过我们上面的 StorageClass 对象自动创建的。这就是 StorageClass 的创建方法。

测试

接下来我们还是用一个简单的示例来测试下我们上面用 StorageClass 方式声明的 PVC 对象吧：(test-pod.yaml)

```

kind: Pod
apiVersion: v1
metadata:
  name: test-pod
spec:
  containers:
  - name: test-pod
    image: busybox
    imagePullPolicy: IfNotPresent
    command:
    - "/bin/sh"
    args:
    - "-c"
    - "touch /mnt/SUCCESS && exit 0 || exit 1"
  volumeMounts:
  - name: nfs-pvc
    mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
  - name: nfs-pvc
    persistentVolumeClaim:
      claimName: test-pvc

```

上面这个 Pod 非常简单，就是用一个 busybox 容器，在 /mnt 目录下面新建一个 SUCCESS 的文件，然后把 /mnt 目录挂载到上面我们新建的 test-pvc 这个资源对象上面了，要验证很简单，只需要去查看下我们 nfs 服务器上面的共享数据目录下面是否有 SUCCESS 这个文件即可：

```
$ kubectl create -f test-pod.yaml
pod "test-pod" created
```

然后我们可以在 nfs 服务器的共享数据目录下面查看下数据：

```
$ ls /data/k8s/
default-test-pvc-pvc-73b5ffd2-8b4b-11e8-b585-525400db4df7
```

我们可以看到下面有名字很长的文件夹，这个文件夹的命名方式是不是和我们上面的规则：\${namespace}-\${pvcName}-\${pvName}是一样的，再看下这个文件夹下面是否有其他文件：

```
$ ls /data/k8s/default-test-pvc-pvc-73b5ffd2-8b4b-11e8-b585-525400db4df7
SUCCESS
```

我们看到下面有一个 SUCCESS 的文件，是不是就证明我们上面的验证是成功的啊。

另外我们可以看到我们这里是手动创建的一个 PVC 对象，在实际工作中，使用 StorageClass 更多的是 StatefulSet 类型的服务，StatefulSet 类型的服务我们也可以通过一个 volumeClaimTemplates 属性来直接使用 StorageClass，如下：(test-statefulset-nfs.yaml)

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: nfs-web
spec:
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nfs-web
  spec:
    terminationGracePeriodSeconds: 10
    containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80
            name: web
        volumeMounts:
          - name: www
            mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
    - metadata:
        name: www
        annotations:
          volume.beta.kubernetes.io/storage-class: course-nfs-storage
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

实际上 volumeClaimTemplates 下面就是一个 PVC 对象的模板，就类似于我们这里 StatefulSet 下面的 template，实际上就是一个 Pod 的模板，我们不单独创建成 PVC 对象，而用这种模板就可以动态的去创建了对象了，这种方式在 StatefulSet 类型的服务下面使用得非常多。

直接创建上面的对象：

```
$ kubectl create -f test-statefulset-nfs.yaml
statefulset.apps "nfs-web" created
$ kubectl get pods
NAME           READY   STATUS    RESTARTS
AGE
...
nfs-web-0      1/1     Running   0
```

1m nfs-web-1	1/1	Running	0
1m nfs-web-2	1/1	Running	0
33s ...			

创建完成后可以看到上面的3个 Pod 已经运行成功，然后查看下 PVC 对象：

NAME	STATUS	VOLUME	CAPACITY	ACCESS MOD
ES	STORAGECLASS	AGE		
www-nfs-web-0	Bound	pvc-cc36b3ce-8b50-11e8-b585-525400db4df7	1Gi	RWO
	course-nfs-storage	2m		
www-nfs-web-1	Bound	pvc-d38285f9-8b50-11e8-b585-525400db4df7	1Gi	RWO
	course-nfs-storage	2m		
www-nfs-web-2	Bound	pvc-e348250b-8b50-11e8-b585-525400db4df7	1Gi	RWO
	course-nfs-storage	1m		

我们可以看到是不是也生成了3个 PVC 对象，名称由模板名称 name 加上 Pod 的名称组合而成，这3个 PVC 对象也都是绑定状态了，很显然我们查看 PV 也可以看到对应的3个 PV 对象：

NAME	CLAIM	CAPACITY	ACCESS MODES	RECLAIM POLICY	STAT
US	STORAGECLASS	REASON	AGE		
		1d			
pvc-cc36b3ce-8b50-11e8-b585-525400db4df7	default/www-nfs-web-0	course-nfs-storage	RWO	Delete	Boun
d			4m		
pvc-d38285f9-8b50-11e8-b585-525400db4df7	default/www-nfs-web-1	course-nfs-storage	RWO	Delete	Boun
d			4m		
pvc-e348250b-8b50-11e8-b585-525400db4df7	default/www-nfs-web-2	course-nfs-storage	RWO	Delete	Boun
d			4m		

查看 nfs 服务器上面的共享数据目录：

```
$ ls /data/k8s/
...
default-www-nfs-web-0-pvc-cc36b3ce-8b50-11e8-b585-525400db4df7
default-www-nfs-web-1-pvc-d38285f9-8b50-11e8-b585-525400db4df7
default-www-nfs-web-2-pvc-e348250b-8b50-11e8-b585-525400db4df7
...
```

是不是也有对应的3个数据目录，这就是我们的 StorageClass 的使用方法，对于 StorageClass 多用于 StatefulSet 类型的服务，在后面的课程中我们还学不断的接触到。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号， 在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:03:19

内部服务发现

前面我们给大家讲解了 Service 的用法，我们可以通过 Service 生成的 ClusterIP(VIP)来访问 Pod 提供的服务，但是在使用的时候还有一个问题：我们怎么知道某个应用的 VIP 呢？比如我们有两个应用，一个是 api 应用，一个是 db 应用，两个应用都是通过 Deployment 进行管理的，并且都通过 Service 暴露出了端口提供服务。api 需要连接到 db 这个应用，我们只知道 db 应用的名称和 db 对应的 Service 的名称，但是并不知道它的 VIP 地址，我们前面的 Service 课程中是不是学习到我们通过 ClusterIP 就可以访问到后面的 Pod 服务，如果我们知道了 VIP 的地址是不是就行了？

apiserver

我们知道可以从 apiserver 中直接查询获取到对应 service 的后端 Endpoints 信息，所以最简单的办法是从 apiserver 中直接查询，如果偶尔一个特殊的应用，我们通过 apiserver 去查询到 Service 后面的 Endpoints 直接使用是没问题的，但是如果每个应用都在启动的时候去查询依赖的服务，这不但增加了应用的复杂度，这也导致了我们的应用需要依赖 Kubernetes 了，耦合度太高了，不具有通用性。

环境变量

为了解决上面的问题，在之前的版本中，Kubernetes 采用了环境变量的方法，每个 Pod 启动的时候，会通过环境变量设置所有服务的 IP 和 port 信息，这样 Pod 中的应用可以通过读取环境变量来获取依赖服务的地址信息，这种方法使用起来相对简单，但是有一个很大的问题就是依赖的服务必须在 Pod 启动之前就存在，不然不会被注入到环境变量中的。比如我们首先创建一个 Nginx 服务：(test-nginx.yaml)

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx-deploy
  labels:
    k8s-app: nginx-demo
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:

```

```

    name: nginx-service
  spec:
    ports:
      - port: 5000
        targetPort: 80
    selector:
      app: nginx

```

创建上面的服务：

```

$ kubectl create -f test-nginx.yaml
deployment.apps "nginx-deploy" created
service "nginx-service" created
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
...
nginx-deploy-75675f5897-47h4t       1/1     Running   0          53s
nginx-deploy-75675f5897-mmm8w       1/1     Running   0          53s
...
$ kubectl get svc
NAME         TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
...
nginx-service   ClusterIP   10.107.225.42   <none>           5000/TCP      1m
...

```

我们可以看到两个 Pod 和一个名为 nginx-service 的服务创建成功了，该 Service 监听的端口是 5000，同时它会把流量转发给它代理的所有 Pod（我们这里就是拥有 app: nginx 标签的两个 Pod）。

现在我们再来创建一个普通的 Pod，观察下该 Pod 中的环境变量是否包含上面的 nginx-service 的服务信息：(test-pod.yaml)

```

apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
    - name: test-service-pod
      image: busybox
      command: ["/bin/sh", "-c", "env"]

```

然后创建该测试的 Pod：

```

$ kubectl create -f test-pod.yaml
pod "test-pod" created

```

等 Pod 创建完成后，我们查看日志信息：

```

$ kubectl logs test-pod
...
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_SERVICE_PORT=443
HOSTNAME=test-pod

```

```

HOME=/root
NGINX_SERVICE_PORT_5000_TCP_ADDR=10.107.225.42
NGINX_SERVICE_PORT_5000_TCP_PORT=5000
NGINX_SERVICE_PORT_5000_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
NGINX_SERVICE_SERVICE_HOST=10.107.225.42
NGINX_SERVICE_PORT_5000_TCP=tcp://10.107.225.42:5000
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
NGINX_SERVICE_SERVICE_PORT=5000
NGINX_SERVICE_PORT=tcp://10.107.225.42:5000
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
...

```

我们可以看到打印了很多环境变量处理，其中就包括我们刚刚创建的 nginx-service 这个服务，有 HOST、PORT、PROTO、ADDR 等，也包括其他已经存在的 Service 的环境变量，现在如果我们需要在这个 Pod 里面访问 nginx-service 的服务，我们是不是可以直接通过 NGINX_SERVICE_SERVICE_HOST 和 NGINX_SERVICE_SERVICE_PORT 就可以了，但是我们也知道如果这个 Pod 启动起来的时候如果 nginx-service 服务还没启动起来，在环境变量中我们是无法获取到这些信息的，当然我们可以通过 initContainer 之类的方法来确保 nginx-service 启动后再启动 Pod，但是这种方法毕竟增加了 Pod 启动的复杂性，所以这不是最优的方法。

KubeDNS

由于上面环境变量这种方式的局限性，我们需要一种更加智能的方案，其实我们可以自己想学一种比较理想的方案：那就是可以直接使用 Service 的名称，因为 Service 的名称不会变化，我们不需要去关心分配的 ClusterIP 的地址，因为这个地址并不是固定不变的，所以如果我们直接使用 Service 的名字，然后对应的 ClusterIP 地址的转换能够自动完成就很好了。我们知道名字和 IP 直接的转换是不是和我们平时访问的网站非常类似啊？他们之间的转换功能通过 DNS 就可以解决了，同样的，Kubernetes 也提供了 DNS 的方案来解决上面的服务发现的问题。

kubedns 介绍

DNS 服务不是一个独立的系统服务，而是作为一种 addon 插件而存在，也就是说不是 Kubernetes 集群必须安装的，当然我们强烈推荐安装，可以将这个插件看成是一种运行在 Kubernetes 集群上的一个特殊的应用，现在比较推荐的两个插件：kube-dns 和 CoreDNS。我们在前面使用 kubeadm 搭建集群的时候直接安装的 kube-dns 插件，如果不记得了可以回头去看一看。当然如果我们想使用 CoreDNS 的话也很方便，只需要执行下面的命令即可：

```
$ kubeadm init --feature-gates=CoreDNS=true
```

Kubernetes DNS pod 中包括 3 个容器，可以通过 kubectl 工具查看：

```
$ kubectl get pods -n kube-system
```

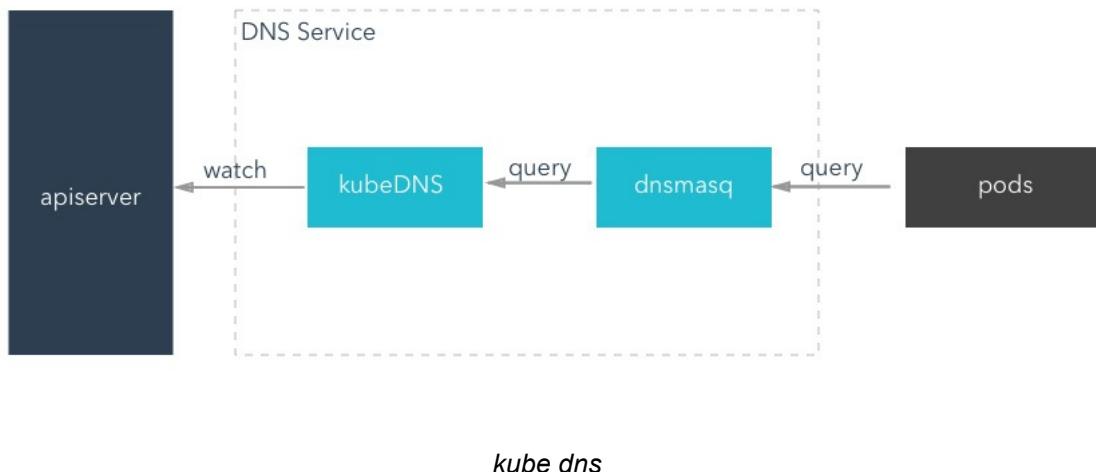
NAME	READY	STATUS	RESTARTS	AGE
...				
kube-dns-5868f69869-zp5kz	3/3	Running	0	19d
...				

READY 一栏可以看到是 3/3，用如下命令可以很清楚的看到 kube-dns 包含的3个容器：

```
$ kubectl describe pod kube-dns-5868f69869-zp5kz -n kube-system
```

kube-dns、dnsmasq-nanny、sidecar 这3个容器分别实现了什么功能？

- **kubedns:** kubedns 基于 SkyDNS 库，通过 apiserver 监听 Service 和 Endpoints 的变更事件同时也同步到本地 Cache，实现了一个实时的 Kubernetes 集群内 Service 和 Pod 的 DNS 服务发现
- **dnsmasq:** dnsmasq 容器则实现了 DNS 的缓存功能(在内存中预留一块默认大小为 1G 的地方，保存当前最常用的 DNS 查询记录，如果缓存中没有要查找的记录，它会到 kubedns 中查询，并把结果缓存起来)，通过监听 ConfigMap 来动态生成配置
- **sidecar:** sidecar 容器实现了可配置的 DNS 探测，并采集对应的监控指标暴露出来供 prometheus 使用



对 Pod 的影响

DNS Pod 具有静态 IP 并作为 Kubernetes 服务暴露出来。该静态 IP 被分配后，kubelet 会将使用 `--cluster-dns = <dns-service-ip>` 参数配置的 DNS 传递给每个容器。DNS 名称也需要域名，本地域可以使用参数 `--cluster-domain = <default-local-domain>` 在 kubelet 中配置。

我们说 dnsmasq 容器通过监听 ConfigMap 来动态生成配置，可以自定义存根域和上下游域名服务器。

例如，下面的 ConfigMap 建立了一个 DNS 配置，它具有一个单独的存根域和两个上游域名服务器：

```
apiVersion: v1
```

```

kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {"acme.local": ["1.2.3.4"]}
  upstreamNameservers: |
    ["8.8.8.8", "8.8.4.4"]

```

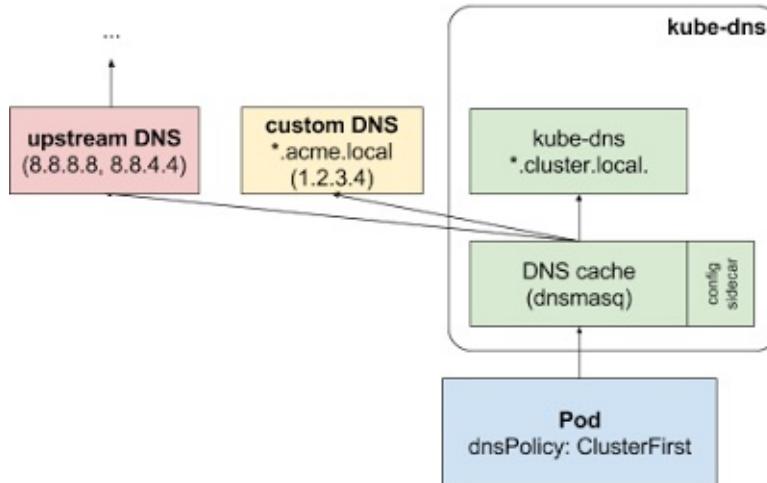
按如上说明，具有.acme.local后缀的 DNS 请求被转发到 DNS 1.2.3.4。Google 公共 DNS 服务器为上游查询提供服务。下表描述了具有特定域名的查询如何映射到它们的目标 DNS 服务器：

域名	响应查询的服务器
kubernetes.default.svc.cluster.local	kube-dns
foo.acme.local	自定义 DNS (1.2.3.4)
widget.com	上游 DNS (8.8.8.8, 8.8.4.4, 其中之一)

另外我们还可以为每个 Pod 设置 DNS 策略。当前 Kubernetes 支持两种 Pod 特定的 DNS 策略：“Default” 和 “ClusterFirst”。可以通过 dnsPolicy 标志来指定这些策略。

注意：Default 不是默认的 DNS 策略。如果没有显式地指定 dnsPolicy，将会使用 ClusterFirst

- 如果 dnsPolicy 被设置为 “Default”，则名字解析配置会继承自 Pod 运行所在的节点。自定义上游域名服务器和存根域不能够与这个策略一起使用
- 如果 dnsPolicy 被设置为 “ClusterFirst”，这就要依赖于是否配置了存根域和上游 DNS 服务器
 - 未进行自定义配置：没有匹配上配置的集群域名后缀的任何请求，例如 “www.kubernetes.io”，将会上游到继承自节点的上游域名服务器。
 - 进行自定义配置：如果配置了存根域和上游 DNS 服务器（类似于前面示例 配置的内容），DNS 查询将基于下面的流程对请求进行路由：
 - 查询首先被发送到 kube-dns 中的 DNS 缓存层。
 - 从缓存层，检查请求的后缀，并根据下面的情况转发到对应的 DNS 上：
 - 具有集群后缀的名字（例如 “.cluster.local”）：请求被发送到 kubedns。
 - 具有存根域后缀的名字（例如 “.acme.local”）：请求被发送到配置的自定义 DNS 解析器（例如：监听在 1.2.3.4）。
 - 未能匹配上后缀的名字（例如 “widget.com”）：请求被转发到上游 DNS（例如：Google 公共 DNS 服务器，8.8.8.8 和 8.8.4.4）。



域名格式

我们前面说了如果我们建立的 Service 如果支持域名形式进行解析，就可以解决我们的服务发现的功能，那么利用 kubedns 可以将 Service 生成怎样的 DNS 记录呢？

- 普通的 Service：会生成 servicename.namespace.svc.cluster.local 的域名，会解析到 Service 对应的 ClusterIP 上，在 Pod 之间的调用可以简写成 servicename.namespace，如果处于同一个命名空间下面，甚至可以只写成 servicename 即可访问
- Headless Service：无头服务，就是把 clusterIP 设置为 None 的，会被解析为指定 Pod 的 IP 列表，同样还可以通过 podname.servicename.namespace.svc.cluster.local 访问到具体的某一个 Pod。

CoreDNS 实现的功能和 KubeDNS 是一致的，不过 CoreDNS 的所有功能都集成在了同一个容器中，在最新版的 1.11.0 版本中官方已经推荐使用 CoreDNS 了，大家也可以安装 CoreDNS 来代替 KubeDNS，其他使用方法都是一致的：<https://coredns.io/>

测试

现在我们来使用一个简单 Pod 来测试下 Service 的域名访问：

```
$ kubectl run --rm -i --tty test-dns --image=busybox /bin/sh
If you don't see a command prompt, try pressing enter.
/ # cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
/ #
```

我们进入到 Pod 中，查看/etc/resolv.conf中的内容，可以看到 nameserver 的地址10.96.0.10，该 IP 地址即是在安装 kubedns 插件的时候集群分配的一个固定的静态 IP 地址，我们可以通过下面的命令进行查看：

```
$ kubectl get svc kube-dns -n kube-system
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
```

kube-dns	ClusterIP	10.96.0.10	<none>	53/UDP, 53/TCP	62d
----------	-----------	------------	--------	----------------	-----

也就是说我们这个 Pod 现在默认的 nameserver 就是 kubedns 的地址，现在我们来访问下前面我们创建的 nginx-service 服务：

```
/ # wget -q -O- nginx-service.default.svc.cluster.local
```

可以看到上面我们使用 wget 命令去访问 nginx-service 服务的域名的时候被 hang 住了，没有得到期望的结果，这是因为上面我们建立 Service 的时候暴露的端口是 5000：

```
/ # wget -q -O- nginx-service.default.svc.cluster.local:5000
<!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>
```

加上 5000 端口，就正常访问到服务，再试一试访问：nginx-service.default.svc、nginx-service.default、nginx-service，不出意外这些域名都可以正常访问到期望的结果。

到这里我们是不是就实现了在集群内部通过 Service 的域名形式进行互相通信了，大家下去试着看看访问不同 namespace 下面的服务呢？下节课我们来给大家讲解使用 ingress 来实现集群外部的服务发现功能。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:03:41

外部服务发现之 ingress

上节课我们学习了在 Kubernetes 集群内部使用 kube-dns 实现服务发现的功能，那么我们部署在 Kubernetes 集群中的应用如何暴露给外部的用户使用呢？我们知道前面我们使用 NodePort 和 LoadBlancer 类型的 Service 可以实现把应用暴露给外部用户使用，除此之外，Kubernetes 还为我们提供了一个非常重要的资源对象可以用来暴露服务给外部用户，那就是 ingress。对于小规模的应用我们使用 NodePort 或许能够满足我们的需求，但是当你的应用越来越多的时候，你就会发现对于 NodePort 的管理就非常麻烦了，这个时候使用 ingress 就非常方便了，可以避免管理大量的 Port。

介绍

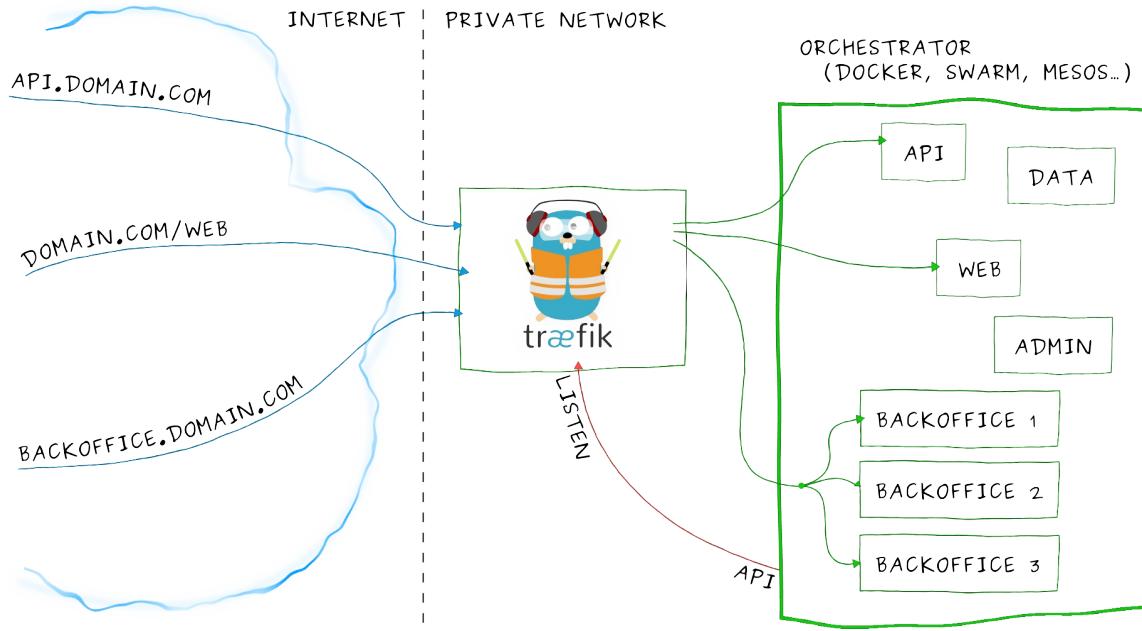
Ingress 其实就是从 kubernetes 集群外部访问集群的一个入口，将外部的请求转发到集群内不同的 Service 上，其实就相当于 nginx、haproxy 等负载均衡代理服务器，有的同学可能觉得我们直接使用 nginx 就实现了，但是只使用 nginx 这种方式有很大缺陷，每次有新服务加入的时候怎么改 Nginx 配置？不可能让我们去手动更改或者滚动更新前端的 Nginx Pod 吧？那我们再加上一个服务发现的工具比如 consul 如何？貌似是可以，对吧？而且在之前单独使用 docker 的时候，这种方式已经使用得很普遍了，Ingress 实际上就是这样实现的，只是服务发现的功能自己实现了，不需要使用第三方的服务了，然后再加上一个域名规则定义，路由信息的刷新需要一个靠 Ingress controller 来提供。

Ingress controller 可以理解为一个监听器，通过不断地与 kube-apiserver 打交道，实时的感知后端 service、pod 的变化，当得到这些变化信息后，Ingress controller 再结合 Ingress 的配置，更新反向代理负载均衡器，达到服务发现的作用。其实这点和服务发现工具 consul consul-template 非常类似。

现在可以供大家使用的 Ingress controller 有很多，比如 [traefik](#)、[nginx-controller](#)、[Kubernetes Ingress Controller for Kong](#)、[HAProxy Ingress controller](#)，当然你也可以自己实现一个 Ingress Controller，现在普遍用得较多的是 traefik 和 nginx-controller，traefik 的性能较 nginx-controller 差，但是配置使用要简单许多，我们这里会以更简单的 traefik 为例给大家介绍 ingress 的使用。

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。



要使用 traefik，我们同样需要部署 traefik 的 Pod，由于我们演示的集群中只有 master 节点有外网网卡，所以我们这里只有 master 这一个边缘节点，我们将 traefik 部署到该节点上即可。

首先，为安全起见我们这里使用 RBAC 安全认证方式：(rbac.yaml):

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: traefik-ingress-controller
  namespace: kube-system
---

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: traefik-ingress-controller
rules:
  - apiGroups:
    - ""
      resources:
        - services
        - endpoints
        - secrets
      verbs:
        - get
        - list
        - watch
  - apiGroups:
    - extensions
      resources:
        - ingresses
      verbs:
        - get
        - list
        - watch
---

kind: ClusterRoleBinding
```

```

apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: traefik-ingress-controller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: traefik-ingress-controller
subjects:
- kind: ServiceAccount
  name: traefik-ingress-controller
  namespace: kube-system

```

直接在集群中创建即可：

```

$ kubectl create -f rbac.yaml
serviceaccount "traefik-ingress-controller" created
clusterrole.rbac.authorization.k8s.io "traefik-ingress-controller" created
clusterrolebinding.rbac.authorization.k8s.io "traefik-ingress-controller" created

```

然后使用 Deployment 来管理 Pod，直接使用官方的 traefik 镜像部署即可（traefik.yaml）

```

---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: traefik-ingress-controller
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: traefik-ingress-lb
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      serviceAccountName: traefik-ingress-controller
      terminationGracePeriodSeconds: 60
      tolerations:
      - operator: "Exists"
      nodeSelector:
        kubernetes.io/hostname: master
      containers:
      - image: traefik
        name: traefik-ingress-lb
        ports:
        - name: http
          containerPort: 80
        - name: admin
          containerPort: 8080
        args:
        - --api
        - --kubernetes

```

```

    - --logLevel=INFO
    ---

kind: Service
apiVersion: v1
metadata:
  name: traefik-ingress-service
  namespace: kube-system
spec:
  selector:
    k8s-app: traefik-ingress-lb
  ports:
    - protocol: TCP
      port: 80
      name: web
    - protocol: TCP
      port: 8080
      name: admin
  type: NodePort

```

直接创建上面的资源对象即可：

```
$ kubectl create -f traefik.yaml
deployment.extensions "traefik-ingress-controller" created
service "traefik-ingress-service" created
```

要注意上面 yaml 文件：

```

tolerations:
- operator: "Exists"
nodeSelector:
  kubernetes.io/hostname: master

```

由于我们这里的特殊性，只有 master 节点有外网访问权限，所以我们使用 nodeSelector 标签将 traefik 的固定调度到 master 这个节点上，那么上面的tolerations是干什么的呢？这个是因为我们集群使用的 kubeadm 安装的，master 节点默认是不能被普通应用调度的，要被调度的话就需要添加这里的 tolerations 属性，当然如果你的集群和我们的不太一样，直接去掉这里的调度策略就行。

nodeSelector 和 tolerations 都属于 Pod 的调度策略，在后面的课程中会为大家讲解。

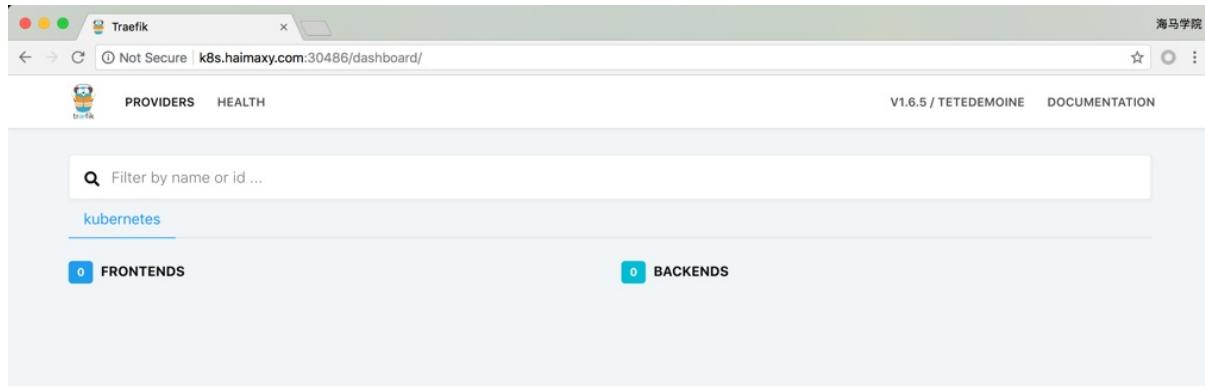
traefik 还提供了一个 web ui 工具，就是上面的 8080 端口对应的服务，为了能够访问到该服务，我们这里将服务设置成的 NodePort：

```

$ kubectl get pods -n kube-system -l k8s-app=traefik-ingress-lb -o wide
NAME                               READY   STATUS    RESTARTS   AGE     IP           NODE
traefik-ingress-controller-57c4f787d9-bfhnl   1/1     Running   0          8m     10.244.0.18   master
$ kubectl get svc -n kube-system
NAME        TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
...
traefik-ingress-service   NodePort    10.102.183.112   <none>         80:30539/TCP,8080:304
86/TCP      8m
...

```

现在在浏览器中输入 master_node_ip:30486 就可以访问到 traefik 的 dashboard 了：



Ingress 对象

现在我们是通过 NodePort 来访问 traefik 的 Dashboard 的，那怎样通过 ingress 来访问呢？首先，需要创建一个 ingress 对象：(ingress.yaml)

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-web-ui
  namespace: kube-system
  annotations:
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: traefik.haimaxy.com
    http:
      paths:
      - backend:
          serviceName: traefik-ingress-service
          servicePort: 8080
```

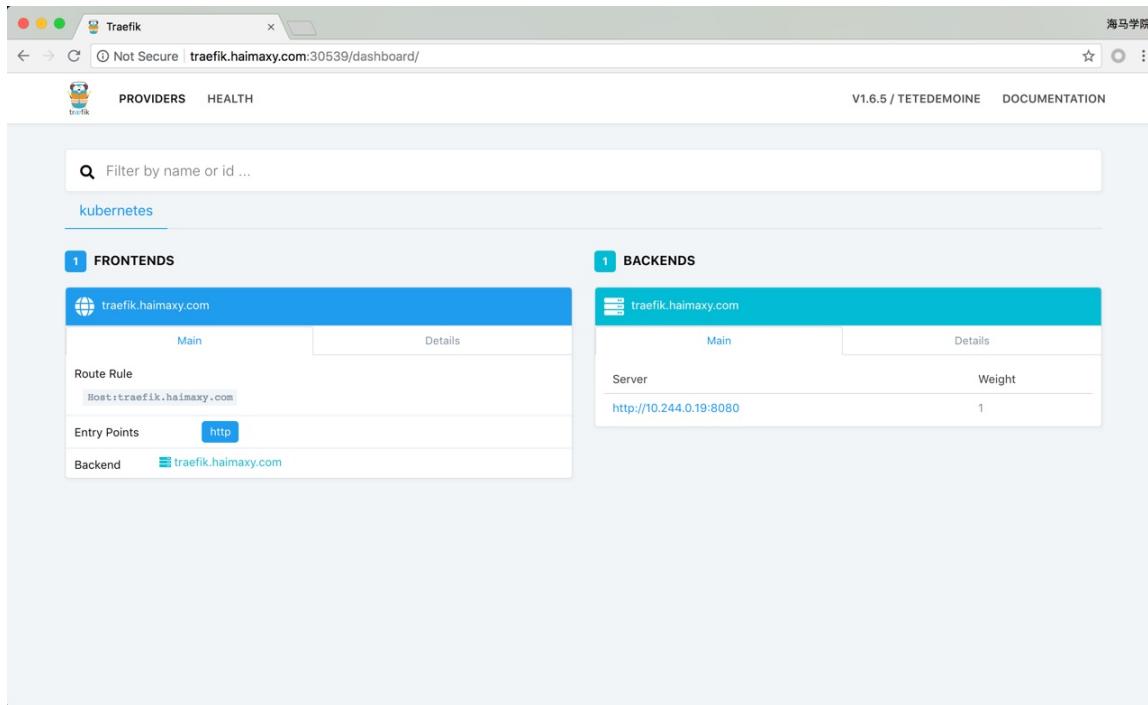
然后为 traefik dashboard 创建对应的 ingress 对象：

```
$ kubectl create -f ingress.yaml
ingress.extensions "traefik-web-ui" created
```

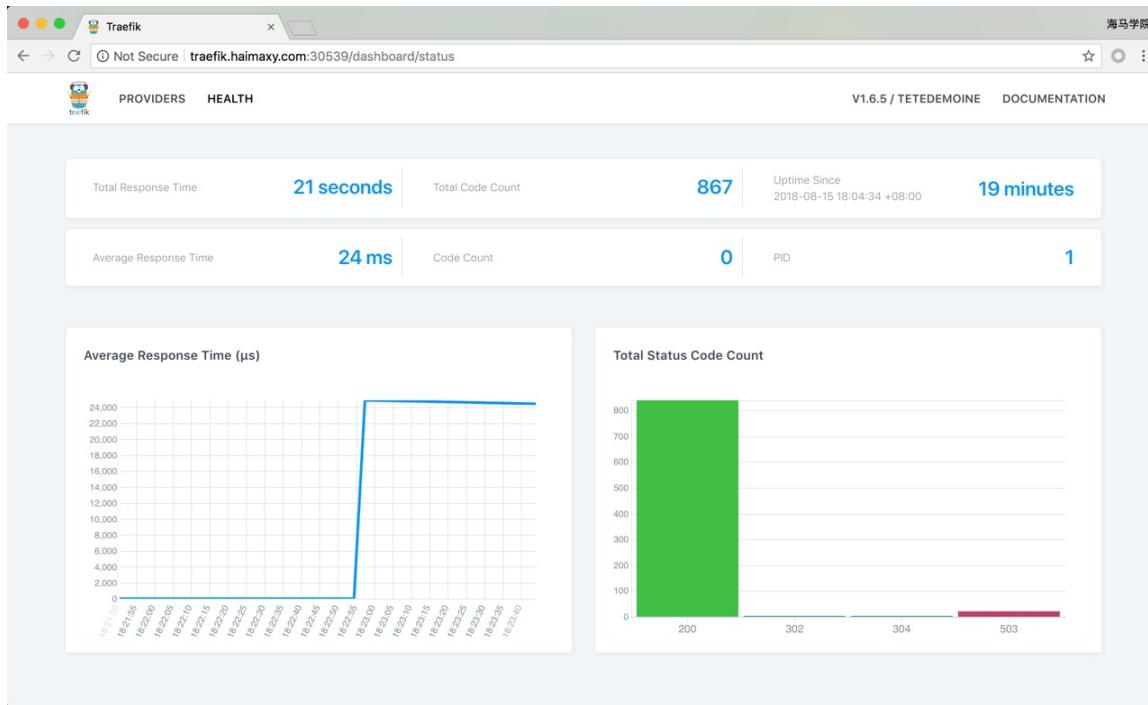
要注意上面的 ingress 对象的规则，特别是 rules 区域，我们这里是要为 traefik 的 dashboard 建立一个 ingress 对象，所以这里的 serviceName 对应的是上面我们创建的 traefik-ingress-service，端口也要注意对应 8080 端口，为了避免端口更改，这里的 servicePort 的值也可以替换成上面定义的 port 的名字：admin

创建完成后，我们应该怎么来测试呢？

- 第一步，在本地的/etc/hosts里面添加上 traefik.haimaxy.com 与 master 节点外网 IP 的映射关系
- 第二步，在浏览器中访问：<http://traefik.haimaxy.com> 我们会发现并没有得到我们期望的 dashboard 界面，这是因为我们上面部署 traefik 的时候使用的是 NodePort 这种 Service 对象，所以我们只能通过上面的 30539 端口访问到我们的目标对象：<http://traefik.haimaxy.com:30539>



加上端口后我们发现可以访问到 dashboard 了，而且在 dashboard 当中多了一条记录，正是上面我们创建的 ingress 对象的数据，我们还可以切换到 HEALTH 界面中，可以查看当前 traefik 代理的服务的整体的健康状态



- 第三步，上面我们可以通过自定义域名加上端口可以访问我们的服务了，但是我们平时服务别人的服务是不是都是直接用的域名啊，http 或者 https 的，几乎很少有在域名后面加上端口访问的吧？为什么？太麻烦啊，端口也记不住，要解决这个问题，怎么办，我们只需要把我们上面的 traefik 的核心应用的端口隐射到 master 节点上的 80 端口，是不是就可以了，因为 http 默认就是访问 80 端口，但是我们在 Service 里面是添加的一个 NodePort 类型的服务，没办法映射 80 端口，怎么办？这里就可以直接在 Pod 中指定一个 hostPort 即可，更改上面的 traefik.yaml 文件中的容器端口：

```

containers:
- image: traefik
  name: traefik-ingress-lb
  ports:
- name: http
  containerPort: 80
  hostPort: 80
- name: admin
  containerPort: 8080

```

添加以后 hostPort: 80 , 然后更新应用:

```
$ kubectl apply -f traefik.yaml
```

更新完成后，这个时候我们在浏览器中直接使用域名方法测试下：

FRONTENDS		BACKENDS	
Host: traefik.haimaxy.com	Main	Server	Main
Route Rule: Host:traefik.haimaxy.com		http://10.244.0.19:8080	
Entry Points: http		Weight: 1	
Backend: traefik.haimaxy.com			

- 第四步，正常来说，我们如果有自己的域名，我们可以将我们的域名添加一条 DNS 记录，解析到 master 的外网 IP 上面，这样任何人都可以通过域名来访问我的暴露的服务了。

如果你有多个边缘节点的话，可以在每个边缘节点上部署一个 ingress-controller 服务，然后在边缘节点前面挂一个负载均衡器，比如 nginx，将所有的边缘节点均作为这个负载均衡器的后端，这样就可以实现 ingress-controller 的高可用和负载均衡了。

到这里我们就通过 ingress 对象对外成功暴露了一个服务，下节课我们再来详细了解 traefik 的更多用法。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-16 09:51:34

ingress tls

上节课给大家展示了 traefik 的安装使用以及简单的 ingress 的配置方法，这节课我们来学习一下 ingress tls 以及 path 路径在 ingress 对象中的使用方法。

TLS 认证

在现在大部分场景下面我们都会使用 https 来访问我们的服务，这节课我们将使用一个自签名的证书，当然你有在一些正规机构购买的 CA 证书是最好的，这样任何人访问你的服务的时候都是受浏览器信任的证书。使用下面的 openssl 命令生成 CA 证书：

```
$ openssl req -newkey rsa:2048 -nodes -keyout tls.key -x509 -days 365 -out tls.crt
```

现在我们有了证书，我们可以使用 kubectl 创建一个 secret 对象来存储上面的证书：

```
$ kubectl create secret generic traefik-cert --from-file=tls.crt --from-file=tls.key -n kube-system
```

配置 Traefik

前面我们使用的是 Traefik 的默认配置，现在我们来配置 Traefik，让其支持 https：

```
defaultEntryPoints = ["http", "https"]

[entryPoints]
  [entryPoints.http]
    address = ":80"
    [entryPoints.http.redirect]
      entryPoint = "https"
  [entryPoints.https]
    address = ":443"
    [entryPoints.https.tls]
      [[entryPoints.https.tls.certificates]]
        CertFile = "/ssl/tls.crt"
        KeyFile = "/ssl/tls.key"
```

上面的配置文件中我们配置了 http 和 https 两个入口，并且配置了将 http 服务强制跳转到 https 服务，这样我们所有通过 traefik 进来的服务都是 https 的，要访问 https 服务，当然就得配置对应的证书了，可以看到我们指定了 CertFile 和 KeyFile 两个文件，由于 traefik pod 中并没有这两个证书，所以我们要想办法将上面生成的证书挂载到 Pod 中去，是不是前面我们讲解过 secret 对象可以通过 volume 形式挂载到 Pod 中？至于上面的 traefik.toml 这个文件我们要怎么让 traefik pod 能够访问到呢？还记得我们前面讲过的 ConfigMap 吗？我们是不是可以将上面的 traefik.toml 配置文件通过一个 ConfigMap 对象挂载到 traefik pod 中去：

```
$ kubectl create configmap traefik-conf --from-file=traefik.toml -n kube-system
```

现在就可以更改下上节课的 traefik pod 的 yaml 文件了：

```

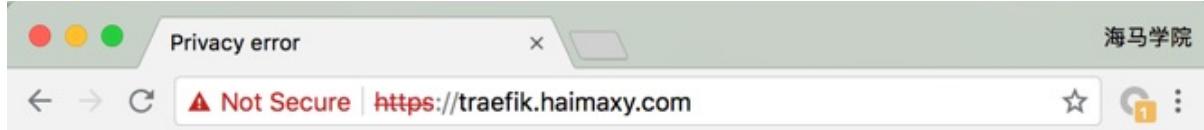
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: traefik-ingress-controller
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: traefik-ingress-lb
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      serviceAccountName: traefik-ingress-controller
      terminationGracePeriodSeconds: 60
      volumes:
        - name: ssl
          secret:
            secretName: traefik-cert
        - name: config
          configMap:
            name: traefik-conf
      tolerations:
        - operator: "Exists"
      nodeSelector:
        kubernetes.io/hostname: master
      containers:
        - image: traefik
          name: traefik-ingress-lb
          volumeMounts:
            - mountPath: "/ssl"
              name: "ssl"
            - mountPath: "/config"
              name: "config"
          ports:
            - name: http
              containerPort: 80
              hostPort: 80
            - name: https
              containerPort: 443
              hostPort: 443
            - name: admin
              containerPort: 8080
          args:
            - --configfile=/config/traefik.toml
            - --api
            - --kubernetes
            - --logLevel=INFO

```

和之前的比较，我们增加了 443 的端口配置，以及启动参数中通过 configfile 指定了 traefik.toml 配置文件，这个配置文件是通过 volume 挂载进来的。然后更新下 traefik pod:

```
$ kubectl apply -f traefik.yaml
$ kubectl logs -f traefik-ingress-controller-7dcfd9c6df-v58k7 -n kube-system
time="2018-08-26T11:26:44Z" level=info msg="Server configuration reloaded on :80"
time="2018-08-26T11:26:44Z" level=info msg="Server configuration reloaded on :443"
time="2018-08-26T11:26:44Z" level=info msg="Server configuration reloaded on :8080"
```

更新完成后我们查看 traefik pod 的日志，如果出现类似于上面的一些日志信息，证明更新成功了。现在我们去访问 traefik 的 dashboard 会跳转到 https 的地址，并会提示证书相关的报警信息，这是因为我们的证书是我们自建的，并不受浏览器信任，如果你是正规机构购买的证书并不会出现改报警信息，你应该可以看到我们常见的绿色标志：



Your connection is not private

Attackers might be trying to steal your information from **traefik.haimaxy.com** (for example, passwords, messages, or credit cards). [Learn more](#)
NET::ERR_CERT_COMMON_NAME_INVALID

Automatically send some [system information and page content](#) to Google to help detect dangerous apps and sites. [Privacy policy](#)

ADVANCED

BACK TO SAFETY

点击下面的高级，我们可以强制让其跳转，这样我们就可以正常访问 traefik 的 dashboard 了。

配置 ingress

其实上面的 TLS 认证方式已经成功了，接下来我们通过一个实例来说明下 ingress 中 path 的用法，这里我们部署了3个简单的 web 服务，通过一个环境变量来标识当前运行的是哪个服务：

(backend.yaml)

```
kind: Deployment
apiVersion: extensions/v1beta1
```

```
metadata:
  name: svc1
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: svc1
  spec:
    containers:
      - name: svc1
        image: cnych/example-web-service
        env:
          - name: APP_SVC
            value: svc1
        ports:
          - containerPort: 8080
            protocol: TCP
    ---  
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: svc2
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: svc2
  spec:
    containers:
      - name: svc2
        image: cnych/example-web-service
        env:
          - name: APP_SVC
            value: svc2
        ports:
          - containerPort: 8080
            protocol: TCP
    ---  
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: svc3
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: svc3
  spec:
    containers:
      - name: svc3
        image: cnych/example-web-service
        env:
          - name: APP_SVC
            value: svc3
        ports:
```

```
- containerPort: 8080
  protocol: TCP

---
kind: Service
apiVersion: v1
metadata:
  labels:
    app: svc1
  name: svc1
spec:
  type: ClusterIP
  ports:
  - port: 8080
    name: http
  selector:
    app: svc1

---
kind: Service
apiVersion: v1
metadata:
  labels:
    app: svc2
  name: svc2
spec:
  type: ClusterIP
  ports:
  - port: 8080
    name: http
  selector:
    app: svc2

---
kind: Service
apiVersion: v1
metadata:
  labels:
    app: svc3
  name: svc3
spec:
  type: ClusterIP
  ports:
  - port: 8080
    name: http
  selector:
    app: svc3
```

可以看到上面我们定义了3个 Deployment，分别对应3个 Service：

```
$ kubectl create -f backend.yaml
deployment.extensions "svc1" created
deployment.extensions "svc2" created
deployment.extensions "svc3" created
service "svc1" created
service "svc2" created
service "svc3" created
```

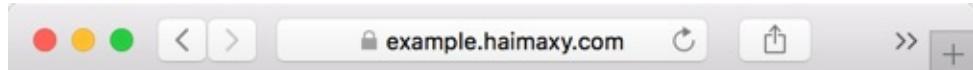
然后我们创建一个 ingress 对象来访问上面的3个服务：(example-ingress.yaml)

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: example-web-app
  annotations:
    kubernetes.io/ingress.class: "traefik"
spec:
  rules:
  - host: example.haimaxy.com
    http:
      paths:
      - path: /s1
        backend:
          serviceName: svc1
          servicePort: 8080
      - path: /s2
        backend:
          serviceName: svc2
          servicePort: 8080
      - path: /
        backend:
          serviceName: svc3
          servicePort: 8080
```

注意我们这里定义的 ingress 对象和之前有一个不同的地方是我们增加了 path 路径的定义，不指定的话默认是 '/'，创建该 ingress 对象：

```
$ kubectl create -f test-ingress.yaml
ingress.extensions "example-web-app" created
$ kubectl get ingress
NAME           HOSTS           ADDRESS   PORTS   AGE
example-web-app   example.haimaxy.com       80        1m
$ kubectl describe ingress example-web-app
Name:           example-web-app
Namespace:      default
Address:
Default backend: default-http-backend:80 (<none>)
Rules:
  Host            Path  Backends
  ----            ---   -----
  example.haimaxy.com
                  /s1    svc1:8080 (<none>)
                  /s2    svc2:8080 (<none>)
                  /      svc3:8080 (<none>)
Annotations:
  kubernetes.io/ingress.class: traefik
Events:
```

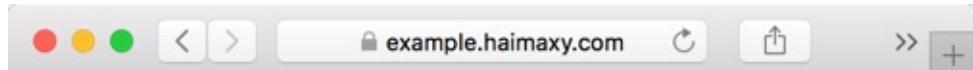
现在我们可以在本地 hosts 里面给域名 example.haimaxy.com 添加对应的 hosts 解析，然后就可以在浏览器中访问，可以看到默认也会跳转到 https 的页面：



Hi, I'm the svc3 service!

Hostname: svc3-6c9fd658fb-5vrjm

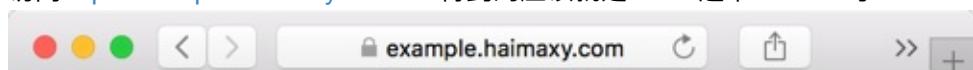
我们可以看到访问上面的域名得到的结果是 svc3 service！这是因为上面在 ingress 中我们为域名的跟路径匹配的是 svc3 这个 service，同样的，我们访问<http://example.haimaxy.com/s1> 得到的应该就是 svc1 这个 service 了：



Hi, I'm the svc1 service!

Hostname: svc1-678fbcd7f8-nkx4d

访问<http://example.haimaxy.com/s2> 得到的应该就是 svc2 这个 service 了：



Hi, I'm the svc2 service!

Hostname: svc2-6bddb4948c-zw92f

这里我们需要注意的是根路径 / 必须在 ingress 对象中声明的时候必须放在最后面，不然就被 / 匹配到拦截到了，大家可以尝试下把 / 这个 path 放在最上面，然后访问下 s1 和 s2 这两个 path，看看得到的结果是怎样的？

有的同学可能有这样的需求，就是不同的 ingress 对象是供不同的域名进行使用的，然后不同的域名的证书还不相同，这样我们想使用上面 traefik 给大家提供的统一的 https 证书就不行了，这个时候我们就可以单独为当前的服务提供单独的证书就可以，同样用证书文件创建一个 secret 对象，然后在 ingress 对象中声明一个 tls 对象即可，比如上面的 example.haimaxy.com 我们可以单独指定一个证书文件：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: example-web-app
  annotations:
    kubernetes.io/ingress.class: "traefik"
spec:
  tls:
    - secretName: traefik-cert
  rules:
    - host:
      ...
      ...
```

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 kubernetes 讨论群里面共同学习。



*k8s技术圈*二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:04:03

42. Helm安装使用

`Helm` 这个东西其实早有耳闻，但是一直没有用在生产环境，而且现在对这货的评价也是褒贬不一。正好最近需要再次部署一套测试环境，对于单体服务，部署一套测试环境我相信还是非常快的，但是对于微服务架构的应用，要部署一套新的环境，就有点折磨人了，微服务越多、你就会越绝望的。虽然我们线上和测试环境已经都迁移到了 `kubernetes` 环境，但是每个微服务也得维护一套 `yaml` 文件，而且每个环境下的配置文件也不太一样，部署一套新的环境成本是真的很高。如果我们能使用类似于 `yum` 的工具来安装我们的应用的话是不是就很爽歪歪了啊？`Helm` 就相当于 `kubernetes` 环境下的 `yum` 包管理工具。

用途

做为 `Kubernetes` 的一个包管理工具，`Helm` 具有如下功能：

- 创建新的 chart
- chart 打包成 tgz 格式
- 上传 chart 到 chart 仓库或从仓库中下载 chart
- 在 `Kubernetes` 集群中安装或卸载 chart
- 管理用 `Helm` 安装的 chart 的发布周期

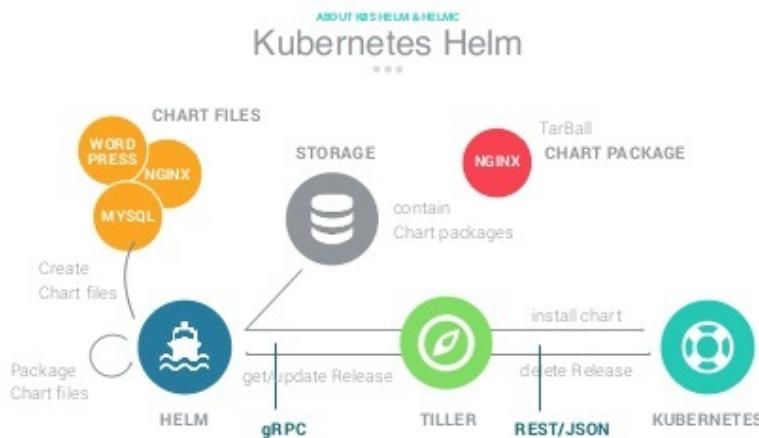
重要概念

`Helm` 有三个重要概念：

- chart：包含了创建 `Kubernetes` 的一个应用实例的必要信息
- config：包含了应用发布配置信息
- release：是一个 chart 及其配置的一个运行实例

Helm组件

Helm 有以下两个组成部分：



`Helm Client` 是用户命令行工具，其主要负责如下：

- 本地 chart 开发
- 仓库管理
- 与 Tiller sever 交互
- 发送预安装的 chart
- 查询 release 信息
- 要求升级或卸载已存在的 release

`Tiller Server` 是一个部署在 Kubernetes 集群内部的 server，其与 Helm client、Kubernetes API server 进行交互。Tiller server 主要负责如下：

- 监听来自 Helm client 的请求
- 通过 chart 及其配置构建一次发布
- 安装 chart 到 Kubernetes 集群，并跟踪随后的发布
- 通过与 Kubernetes 交互升级或卸载 chart
- 简单的说，client 管理 charts，而 server 管理发布 release

安装

我们可以在[Helm Realese](#)页面下载二进制文件，这里下载的v2.10.0版本，解压后将可执行文件 `helm` 拷贝到 `/usr/local/bin` 目录下即可，这样 `Helm` 客户端就在这台机器上安装完成了。

现在我们可以使用 `Helm` 命令查看版本了，会提示无法连接到服务端 `Tiller`：

```
$ helm version
Client: &version.Version{SemVer:"v2.10.0", GitCommit:"9ad53aac42165a5fadcc6c87be0dea6b115f9
3090", GitTreeState:"clean"}
Error: could not find tiller
```

要安装 Helm 的服务端程序，我们需要使用到 kubectl 工具，所以先确保 kubectl 工具能够正常的访问 kubernetes 集群的 apiserver 哦。

然后我们在命令行中执行初始化操作：

```
$ helm init
```

由于 Helm 默认会去 gcr.io 拉取镜像，所以如果你当前执行的机器没有配置科学上网的话可以实现下面的命令代替：

```
$ helm init --upgrade --tiller-image cnych/tiller:v2.10.0
$HELM_HOME has been configured at /root/.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see: https://docs.helm.sh/using_helm/#securing-your-helm-installation
Happy Helming!
```

如果一直卡住或者报 google api 之类的错误，可以使用下面的命令进行初始化：

```
$ helm init --upgrade --tiller-image cnych/tiller:v2.10.0 --stable-repo-url https://cnych.github.io/kube-charts-mirror/
```

这个命令会把默认的 google 的仓库地址替换成我同步的一个镜像地址。

如果在安装过程中遇到了一些其他问题，比如初始化的时候出现了如下错误：

```
E0125 14:03:19.093131 56246 portforward.go:331] an error occurred forwarding 55943 -> 44
134: error forwarding port 44134 to pod d01941068c9dfea1c9e46127578994d1cf8bc34c971ff109dc
6faa4c05043a6e, uid : unable to do port forwarding: socat not found.
2018/01/25 14:03:19 (0xc420476210) (0xc4203ae1e0) Stream removed, broadcasting: 3
2018/01/25 14:03:19 (0xc4203ae1e0) (3) Writing data frame
2018/01/25 14:03:19 (0xc420476210) (0xc4200c3900) Create stream
2018/01/25 14:03:19 (0xc420476210) (0xc4200c3900) Stream added, broadcasting: 5
Error: cannot connect to Tiller
```

解决方案：在节点上安装 socat 可以解决

```
$ sudo yum install -y socat
```

Helm 服务端正常安装完成后，Tiller 默认被部署在 kubernetes 集群的 kube-system 命名空间下：

```
$ kubectl get pod -n kube-system -l app=helm
NAME                  READY     STATUS    RESTARTS   AGE
tiller-deploy-86b844d8c6-44fpq   1/1      Running   0          7m
```

此时，我们查看 Helm 版本就都正常了：

```
$ helm version
Client: &version.Version{SemVer:"v2.10.0", GitCommit:"9ad53aac42165a5fadcc6c87be0dea6b115f9
3090", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.10.0", GitCommit:"9ad53aac42165a5fadcc6c87be0dea6b115f9
3090", GitTreeState:"clean"}
```

另外一个值得注意的问题是 RBAC，我们的 kubernetes 集群是1.10.0版本的，默认开启了 RBAC 访问控制，所以我们需要为 Tiller 创建一个 ServiceAccount，让他拥有执行的权限，详细内容可以查看 Helm 文档中的[Role-based Access Control](#)。创建 rbac.yaml 文件：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
```

然后使用 kubectl 创建：

```
$ kubectl create -f rbac-config.yaml
serviceaccount "tiller" created
clusterrolebinding.rbac.authorization.k8s.io "tiller" created
```

创建了 tiller 的 ServiceAccount 后还没完，因为我们的 Tiller 之前已经就部署成功了，而且是没有指定 ServiceAccount 的，所以我们需要给 Tiller 打上一个 ServiceAccount 的补丁：

```
$ kubectl patch deploy --namespace kube-system tiller-deploy -p '{"spec":{"template":{"spe
c":{"serviceAccount":"tiller"}}}}'
```

上面这一步非常重要，不然后面在使用 Helm 的过程中可能出现 Error: no available release name found 的错误信息。

至此，Helm 客户端和服务端都配置完成了，接下来我们看看如何使用吧。

使用

我们现在尝试创建一个 Chart:

```
$ helm create hello-helm
Creating hello-helm
$ tree hello-helm
hello-helm
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── ingress.yaml
│   ├── NOTES.txt
│   └── service.yaml
└── values.yaml

2 directories, 7 files
```

我们通过查看 `templates` 目录下面的 `deployment.yaml` 文件可以看出默认创建的 Chart 是一个 nginx 服务，具体的每个文件是干什么用的，我们可以前往 [Helm 官方文档](#) 进行查看，后面会和大家详细讲解的。比如这里我们来安装 1.7.9 这个版本的 nginx，则我们更改 `values.yaml` 文件下面的 `image tag` 即可，将默认的 `stable` 更改为 1.7.9，为了测试方便，我们把 Service 的类型也改成 `NodePort`

```
...
image:
  repository: nginx
  tag: 1.7.9
  pullPolicy: IfNotPresent

nameOverride: ""
fullnameOverride: ""

service:
  type: NodePort
  port: 80
...
```

现在我们来尝试安装下这个 Chart :

```
$ helm install ./hello-helm
NAME: iced-ferret
LAST DEPLOYED: Thu Aug 30 23:39:45 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME                  TYPE      CLUSTER-IP      EXTERNAL-IP  PORT(S)   AGE
iced-ferret-hello-helm  ClusterIP  10.100.118.77  <none>        80/TCP   0s

==> v1beta2/Deployment
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
iced-ferret-hello-helm  1         0         0           0          0s

==> v1/Pod(related)
```

```

NAME                                READY  STATUS    RESTARTS  AGE
iced-ferret-hello-helm-58cb69d5bb-s9f2m  0/1    Pending   0          0s

NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l "app=hello-helm,release=iced-ferret" -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl port-forward $POD_NAME 8080:80

$ kubectl get pods -l app=hello-helm
NAME                                READY  STATUS    RESTARTS  AGE
iced-ferret-hello-helm-58cb69d5bb-s9f2m  1/1    Running   0          2m
$ kubectl get svc -l app=hello-helm
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
iced-ferret-hello-helm   NodePort   10.104.127.141  <none>        80:31236/TCP   3m

```

等到 Pod 创建完成后，我们可以根据创建的 Service 的 NodePort 来访问该服务了，然后在浏览器中打开 <http://k8s.haimaxy.com:31236> 就可以正常的访问我们刚刚部署的 nginx 应用了。



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

查看 release :

```

$ helm list
NAME      REVISION  UPDATED           STATUS      CHART
PP VERSION  NAMESPACE
winning-zebra  1       Thu Aug 30 23:50:29 2018  DEPLOYED  hello-helm-0.1.0  1
.0          default

```

打包 chart :

```

$ helm package hello-helm
Successfully packaged chart and saved it to: /root/course/kubeadm/helm/hello-helm-0.1.0.tgz

```

然后我们就可以将打包的 `tgz` 文件分发到任意的服务器上，通过 `helm fetch` 就可以获取到该 Chart 了。

删除 `release`：

```
$ helm delete winning-zebra
release "winning-zebra" deleted
```

然后我们看到 `kubernetes` 集群上的该 `nginx` 服务也被删除了。

```
$ kubectl get pods -l app=hello-helm
No resources found.
```

还有更多关于 `Helm` 的使用命令，我们可以前往[官方文档](#)查看。下节课我们再来和大家详细讲解。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



*k8s技术圈*二维码

43. Helm 的基本使用

上节课我们成功安装了 Helm 的客户端以及服务端 Tiller Server，我们也自己尝试创建了我们的第一个 Helm Chart 包，这节课就来和大家一起学习下 Helm 中的一些常用的操作方法。

仓库

Helm 的 Repo 仓库和 Docker Registry 比较类似，Chart 库可以用来存储和共享打包 Chart 的位置，我们在安装了 Helm 后，默认的仓库地址是 google 的一个地址，这对于我们不能科学上网的同学就比较苦恼了，没办法访问到官方提供的 Chart 仓库，可以用 helm repo list 来查看当前的仓库配置：

```
$ helm repo list
NAME      URL
stable    https://kubernetes-charts.storage.googleapis.com/
local     http://127.0.0.1:8879/charts
```

我们可以看到除了一个默认的 stable 的仓库配置外，还有一个 local 的本地仓库，这是我们本地测试的一个仓库地址。其实要创建一个 Chart 仓库也是非常简单的，Chart 仓库其实就是一个带有 index.yaml 索引文件和任意个打包的 Chart 的 HTTP 服务器而已，比如我们想要分享一个 Chart 包的时候，将我们本地的 Chart 包上传到该服务器上面，别人就可以使用了，所以其实我们自己托管一个 Chart 仓库也是非常简单的，比如阿里云的 OSS、Github Pages，甚至自己创建的一个简单服务器都可以。

为了解决科学上网的问题，我这里建了一个 Github Pages 仓库，每天会自动和官方的仓库进行同步，地址是：<https://github.com/cnyc/helm-charts-mirror>，这样我们就可以将我们的 Helm 默认仓库地址更改成我们自己的仓库地址了：

```
$ helm repo remove stable
"stable" has been removed from your repositories
$ helm repo add stable https://cnyc.github.io/kube-charts-mirror/
"stable" has been added to your repositories
$ helm repo list
NAME      URL
stable    https://cnyc.github.io/kube-charts-mirror/
local     http://127.0.0.1:8879/charts
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. * Happy Helming!*
```

仓库添加完成后，可以使用 update 命令进行仓库更新。当然如果要我们自己来创建一个 web 服务器来服务 Helm Chart 的话，只需要实现下面几个功能点就可以提供服务了：

- 将索引和 Chart 置于服务器目录中
- 确保索引文件 index.yaml 可以在没有认证要求的情况下访问
- 确保 yaml 文件的正确内容类型 (text/yaml 或 text/x-yaml)

如果你的 web 服务提供了上面几个功能，那么也就可以当做 Helm Chart 仓库来使用了。

查找 chart

Helm 将 Charts 包安装到 Kubernetes 集群中，一个安装实例就是一个新的 Release，要找到新的 Chart，我们可以通过搜索命令完成。

记住，如果不能科学上网，将默认的 stable 的仓库地址更换成上面我们创建的地址

直接运行 `helm search` 命令可以查看有哪些 Charts 是可用的：

```
$ helm search
NAME          CHART VERSION   APP VERSION
DESCRIPTION

...
stable/minio      1.6.3          RELEASE.2018-08-25T01-56-38Z
Minio is a high performance distributed object storage se...
stable/mission-control 0.4.2          3.1.2
A Helm chart for JFrog Mission Control
stable/mongodb     4.2.2          4.0.2
NoSQL document-oriented database that stores JSON-like do...
stable/mongodb-replicaset 3.5.6          3.6
NoSQL document-oriented database that stores JSON-like do...
...
stable/zetcd       0.1.9          0.0.3
CoreOS zetcd Helm chart for Kubernetes
...
```

如果没有使用过滤条件，`helm search` 显示所有可用的 charts。可以通过使用过滤条件进行搜索来缩小搜索的结果范围：

```
$ helm search mysql
NAME          CHART VERSION   APP VERSION   DESCRIPTION
...
stable/mysql      0.10.1         5.7.14        Fast, reliable, scalab
le, and easy to use open-source rel...
stable/mysqldump    0.1.0          5.7.21        A Helm chart to help b
ackup MySQL databases using mysqldump
stable/prometheus-mysql-exporter 0.1.0         v0.10.0       A Helm chart for prome
theus
stable/mariadb     4.4.0          10.1.35       Fast, reliable, scalab
le, and easy to use open-source rel...
...
```

可以看到明显少了很多 charts 了，同样的，我们可以使用 `inspect` 命令来查看一个 chart 的详细信息：

```
$ helm inspect stable/mysql
appVersion: 5.7.14
description: Fast, reliable, scalable, and easy to use open-source relational database
system.
engine: gotpl
home: https://www.mysql.com/
icon: https://www.mysql.com/common/logos/logo-mysql-170x115.png
keywords:
```

```

- mysql
- database
- sql
maintainers:
- email: o.with@sportradar.com
  name: olemarkus
- email: viglesias@google.com
  name: viglesiasce
name: mysql
sources:
- https://github.com/kubernetes/charts
- https://github.com/docker-library/mysql
version: 0.10.1

---
## mysql image version
## ref: https://hub.docker.com/r/library/mysql/tags/
##
image: "mysql"
imageTag: "5.7.14"
...

```

使用 `inspect` 命令可以查看到该 chart 里面所有描述信息，包括运行方式、配置信息等等。

通过 `helm search` 命令可以找到我们想要的 chart 包，找到后就可以通过 `helm install` 命令来进行安装了。

安装 chart

要安装新的软件包，直接使用 `helm install` 命令即可。最简单的情况下，它只需要一个 chart 的名称参数：

```

$ helm install stable/mysql
NAME: mewing-squid
LAST DEPLOYED: Tue Sep 4 23:31:23 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/PersistentVolumeClaim
NAME           STATUS      VOLUME   CAPACITY   ACCESS MODES  STORAGECLASS   AGE
mewing-squid-mysql   Pending    1s

==> v1/Service
NAME          TYPE      CLUSTER-IP      EXTERNAL-IP  PORT(S)   AGE
mewing-squid-mysql  ClusterIP  10.108.197.48  <none>     3306/TCP  1s

==> v1beta1/Deployment
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
mewing-squid-mysql  1        0        0          0          1s

==> v1/Pod(related)
NAME          READY  STATUS      RESTARTS  AGE
mewing-squid-mysql-69f587bdf9-z7glv  0/1    Pending    0          0s

==> v1/Secret

```

```

NAME          TYPE    DATA  AGE
mewing-squid-mysql  Opaque  2      1s

==> v1/ConfigMap
NAME          DATA  AGE
mewing-squid-mysql-test  1      1s

NOTES:
MySQL can be accessed via port 3306 on the following DNS name from within your cluster:
mewing-squid-mysql.default.svc.cluster.local

To get your root password run:

  MYSQL_ROOT_PASSWORD=$(kubectl get secret --namespace default mewing-squid-mysql -o jsonpath='{.data.mysql-root-password}' | base64 --decode; echo)

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

  kubectl run -i --tty ubuntu --image=ubuntu:16.04 --restart=Never -- bash -il

2. Install the mysql client:

  $ apt-get update && apt-get install mysql-client -y

3. Connect using the mysql cli, then provide your password:
  $ mysql -h mewing-squid-mysql -p

To connect to your database directly from outside the K8s cluster:

  MYSQL_HOST=127.0.0.1
  MYSQL_PORT=3306

  # Execute the following command to route the connection:
  kubectl port-forward svc/mewing-squid-mysql 3306

  mysql -h ${MYSQL_HOST} -P${MYSQL_PORT} -u root -p${MYSQL_ROOT_PASSWORD}

```

现在 mysql chart 已经安装上了，安装 chart 会创建一个新 release 对象。上面的 release 被命名为 hmewing-squid。如果你想使用你自己的 release 名称，只需使用 `--name` 参数指定即可，比如：

```
$ helm install stable/mysql --name mydb
```

在安装过程中，helm 客户端将打印有关创建哪些资源的有用信息，release 的状态以及其他有用的配置信息，比如这里的有访问 mysql 服务的方法、获取 root 用户的密码以及连接 mysql 的方法等信息。

值得注意的是 Helm 并不会一直等到所有资源都运行才退出。因为很多 charts 需要的镜像资源非常大，所以可能需要很长时间才能安装到集群中去。

要跟踪 release 状态或重新读取配置信息，可以使用 helm status 查看：

```
$ helm status mewing-squid
LAST DEPLOYED: Tue Sep 4 23:31:23 2018
NAMESPACE: default
STATUS: DEPLOYED
```

```
RESOURCES:
```

```
...
```

可以看到当前 release 的状态是 `DEPLOYED`，下面还有一些安装的时候出现的信息。

自定义 chart

上面的安装方式是使用 chart 的默认配置选项。但是在很多时候，我们都需要自定义 chart 以满足自身的需求，要自定义 chart，我们就需要知道我们使用的 chart 支持的可配置选项才行。

要查看 chart 上可配置的选项，使用 `helm inspect values` 命令即可，比如我们这里查看上面的 mysql 的配置选项：

```
$ helm inspect values stable/mysql
## mysql image version
## ref: https://hub.docker.com/r/library/mysql/tags/
##
image: "mysql"
imageTag: "5.7.14"

## Specify password for root user
##
## Default: random 10 character string
# mysqlRootPassword: testing

## Create a database user
##
# mysqlUser:
## Default: random 10 character string
# mysqlPassword:

## Allow unauthenticated access, uncomment to enable
##
# mysqlAllowEmptyPassword: true

## Create a database
##
# mysqlDatabase:

## Specify an imagePullPolicy (Required)
## It's recommended to change this to 'Always' if the image tag is 'latest'
## ref: http://kubernetes.io/docs/user-guide/images/#updating-images
##
imagePullPolicy: IfNotPresent

extraVolumes: |
  # - name: extras
  #   emptyDir: {}

extraVolumeMounts: |
  # - name: extras
  #   mountPath: /usr/share/extras
  #   readOnly: true

extraInitContainers: |
```

```

# - name: do-something
#   image: busybox
#   command: ['do', 'something']

# Optionally specify an array of imagePullSecrets.
# Secrets must be manually created in the namespace.
# ref: https://kubernetes.io/docs/concepts/containers/images/#specifying-imagepullsecrets-on-a-pod
# imagePullSecrets:
# - name: myRegistryKeySecretName

## Node selector
## ref: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#nodeselector
nodeSelector: {}

livenessProbe:
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  successThreshold: 1
  failureThreshold: 3

readinessProbe:
  initialDelaySeconds: 5
  periodSeconds: 10
  timeoutSeconds: 1
  successThreshold: 1
  failureThreshold: 3

## Persist data to a persistent volume
persistence:
  enabled: true
  ## database data Persistent Volume Storage Class
  ## If defined, storageClassName: <storageClass>
  ## If set to "-", storageClassName: "", which disables dynamic provisioning
  ## If undefined (the default) or set to null, no storageClassName spec is
  ## set, choosing the default provisioner. (gp2 on AWS, standard on
  ## GKE, AWS & OpenStack)
  ##
  # storageClass: "-"
  accessMode: ReadWriteOnce
  size: 8Gi
  annotations: {}

## Configure resource requests and limits
## ref: http://kubernetes.io/docs/user-guide/compute-resources/
##
resources:
  requests:
    memory: 256Mi
    cpu: 100m

# Custom mysql configuration files used to override default mysql settings
configurationFiles: {}
# mysql.cnf: |-
#   [mysqld]
#   skip-name-resolve
#   ssl-ca=/ssl/ca.pem
#   ssl-cert=/ssl/server-cert.pem
#   ssl-key=/ssl/server-key.pem

```

```

# Custom mysql init SQL files used to initialize the database
initializationFiles: {}
# first-db.sql: |-
#   CREATE DATABASE IF NOT EXISTS first DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_general_ci;
# second-db.sql: |-
#   CREATE DATABASE IF NOT EXISTS second DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_general_ci;

metrics:
  enabled: false
  image: prom/mysqld-exporter
  imageTag: v0.10.0
  imagePullPolicy: IfNotPresent
  resources: {}
  annotations: {}
    # prometheus.io/scrape: "true"
    # prometheus.io/port: "9104"
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
readinessProbe:
  initialDelaySeconds: 5
  timeoutSeconds: 1

## Configure the service
## ref: http://kubernetes.io/docs/user-guide/services/
service:
  ## Specify a service type
  ## ref: https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services--service-types
  type: ClusterIP
  port: 3306
  # nodePort: 32000

ssl:
  enabled: false
  secret: mysql-ssl-certs
  certificates:
    - name: mysql-ssl-certs
      ca: |-
        -----BEGIN CERTIFICATE-----
        ...
        -----END CERTIFICATE-----
      cert: |-
        -----BEGIN CERTIFICATE-----
        ...
        -----END CERTIFICATE-----
      key: |-
        -----BEGIN RSA PRIVATE KEY-----
        ...
        -----END RSA PRIVATE KEY-----

## Populates the 'TZ' system timezone environment variable
## ref: https://dev.mysql.com/doc/refman/5.7/en/time-zone-support.html
##
## Default: nil (mysql will use image's default timezone, normally UTC)
## Example: 'Australia/Sydney'
# timezone:

```

```
# To be added to the database server pod(s)
podAnnotations: {}
```

然后，我们可以直接在 YAML 格式的文件中来覆盖上面的任何配置，在安装的时候直接使用该配置文件即可：(config.yaml)

```
mysqlUser: haimaxyUser
mysqlDatabase: haimaxyDB
service:
  type: NodePort
```

我们这里通过 config.yaml 文件定义了 mysqlUser 和 mysqlDatabase，并且把 service 的类型更改为了 NodePort，然后现在我们来安装的时候直接指定该 yaml 文件：

```
$ helm install -f config.yaml stable/mysql --name mydb
NAME: mydb
LAST DEPLOYED: Wed Sep 5 00:09:44 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Secret
NAME      TYPE    DATA  AGE
mydb-mysql   Opaque  2     1s

==> v1/ConfigMap
NAME          DATA  AGE
mydb-mysql-test  1     1s

==> v1/PersistentVolumeClaim
NAME      STATUS  VOLUME  CAPACITY  ACCESS MODES  STORAGECLASS  AGE
mydb-mysql   Pending  1s

==> v1/Service
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP  PORT(S)      AGE
mydb-mysql  NodePort  10.96.150.198  <none>        3306:32604/TCP  0s

==> v1beta1/Deployment
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
mydb-mysql  1        1        1           0          0s

==> v1/Pod(related)
NAME          READY  STATUS    RESTARTS  AGE
mydb-mysql-dfc999888-hbw5d  0/1    Pending   0          0s
...
```

我们可以看到当前 release 的名字已经变成 mydb 了。然后可以查看下 mydb 关联的 Service 是否变成 NodePort 类型的了：

```
$ kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP  PORT(S)      AGE
kubernetes  ClusterIP  10.96.0.1    <none>        443/TCP      110d
mewning-squid-mysql  ClusterIP  10.108.197.48  <none>        3306/TCP      46m
mydb-mysql   NodePort  10.96.150.198  <none>        3306:32604/TCP  8m
```

看到服务 mydb-mysql 变成了 NodePort 类型的，之前默认创建的 mewing-squid-mysql 是 ClusterIP 类型的，证明上面我们通过 YAML 文件来覆盖 values 是成功的。

接下来我们查看下 Pod 的状况：

```
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
mewing-squid-mysql-69f587bdf9-z7glv   0/1     Pending   0          49m
mydb-mysql-dfc999888-hbw5d           0/1     Pending   0          11m
```

比较奇怪的是之前默认创建的和现在的 mydb 的 release 创建的 Pod 都是 Pending 状态，直接使用 describe 命令查看下：

```
$ kubectl describe pod mydb-mysql-dfc999888-hbw5d
Name:           mydb-mysql-dfc999888-hbw5d
Namespace:      default
Node:           <none>
Labels:         app=mysqld
                pod-template-hash=897555444
...
Events:
Type    Reason        Age           From            Message
----    ----        --            --             --
Warning FailedScheduling 2m (x37 over 12m)  default-scheduler  pod has unbound PersistentVolumeClaims (repeated 2 times)
```

我们可以发现两个 Pod 处于 Pending 状态的原因都是 PVC 没有被绑定上，所以这里我们可以通过 storageclass 或者手动创建一个合适的 PV 对象来解决这个问题。

另外为了说明 helm 更新的用法，我们这里来直接禁用掉数据持久化，可以在上面的 config.yaml 文件中设置：

```
persistence:
  enabled: false
```

另外一种方法就是在安装过程中使用 `--set` 来覆盖对应的 value 值，比如禁用数据持久化，我们这里可以这样来覆盖：

```
$ helm install stable/mysql --set persistence.enabled=false --name mydb
```

升级

我们这里将数据持久化禁用掉来对上面的 mydb 进行升级：

```
$ echo config.yaml
mysqlUser: haimaxyUser
mysqlDatabase: haimaxyDB
service:
```

```

    type: NodePort
  persistence:
    enabled: false
$ helm upgrade -f config.yaml mydb stable/mysql
helm upgrade -f config.yaml mydb stable/mysql
Release "mydb" has been upgraded. Happy Helming!
LAST DEPLOYED: Wed Sep  5 00:38:33 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
  ...

```

可以看到已经变成 DEPLOYED 状态了，现在我们再去看看 Pod 的状态呢：

```

$ kubectl get pods
NAME                               READY   STATUS        RESTARTS   AGE
mewing-squid-mysql-69f587bdf9-z7glv   0/1     Pending       0          1h
mydb-mysql-6ffc84bbf6-lcn4d         0/1     PodInitializing   0          49s
...

```

我们看到 mydb 关联的 Pod 已经变成了 PodInitializing 的状态，已经不是 Pending 状态了，同样的，使用 describe 命令查看：

```

$ kubectl describe pod mydb-mysql-6ffc84bbf6-lcn4d
Name:           mydb-mysql-6ffc84bbf6-lcn4d
Namespace:      default
Node:          node02/10.151.30.63
Start Time:    Wed, 05 Sep 2018 00:38:33 +0800
Labels:         app=mydb-mysql
                pod-template-hash=2997406692
Annotations:   <none>
Status:        Pending
...
Events:
  Type  Reason          Age   From            Message
  ----  ----          --   --              --
  Normal  SuccessfulMountVolume  58s  kubelet, node02  MountVolume.SetUp succeeded for volume "data"
  Normal  SuccessfulMountVolume  58s  kubelet, node02  MountVolume.SetUp succeeded for volume "default-token-n9w2d"
  Normal  Scheduled          57s  default-scheduler  Successfully assigned mydb-mysql-6ffc84bbf6-lcn4d to node02
  Normal  Pulling            57s  kubelet, node02  pulling image "busybox:1.25.0"
  Normal  Pulled             45s  kubelet, node02  Successfully pulled image "busybox:1.25.0"
  Normal  Created            44s  kubelet, node02  Created container
  Normal  Started            44s  kubelet, node02  Started container
  Normal  Pulling            41s  kubelet, node02  pulling image "mysql:5.7.14"

```

我们可以看到现在没有任何关于 PVC 的错误信息了，这是因为我们刚刚更新的版本中就是禁用掉了的数据持久化的，证明 helm upgrade 和 --values 是生效了的。现在我们使用 helm ls 命令查看先当前的 release：

```
$ helm ls
```

NAME REVISION SION NAMESPACE	REVISION 1 default	UPDATED Tue Sep 4 23:31:23 2018	STATUS DEPLOYED	CHART mysql-0.10.1	APP VE 5.7.14
mydb	2	Wed Sep 5 00:38:33 2018	DEPLOYED	mysql-0.10.1	5.7.14

可以看到 mydb 这个 release 的 REVISION 已经变成2了，这是因为 release 的版本是递增的，每次安装、升级或者回滚，版本号都会加1，第一个版本号始终为1，同样我们可以使用 helm history 命令查看 release 的历史版本：

```
$ helm history mydb
REVISION UPDATED STATUS CHART DESCRIPTION
1 Wed Sep 5 00:09:44 2018 SUPERSEDED mysql-0.10.1 Install complete
2 Wed Sep 5 00:38:33 2018 DEPLOYED mysql-0.10.1 Upgrade complete
```

当然如果我们要回滚到某一个版本的话，使用 helm rollback 命令即可，比如我们将 mydb 回滚到上一个版本：

```
$ $ helm rollback mydb 1
```

删除

上节课我们就学习了要删除一个 release 直接使用 helm delete 命令就 OK:

```
$ helm delete mewing-squid
release "mewing-squid" deleted
```

这将从集群中删除该 release，但是这并不代表就完全删除了，我们还可以通过 --deleted 参数来显示被删除掉 release:

```
$ helm list --deleted
NAME REVISION UPDATED STATUS CHART APP VER
SION NAMESPACE
mewing-squid 1 Tue Sep 4 23:31:23 2018 DELETED mysql-0.10.1 5.7.14
$ helm list --all
NAME REVISION UPDATED STATUS CHART APP VE
RSION NAMESPACE
mewing-squid 1 Tue Sep 4 23:31:23 2018 DELETED mysql-0.10.1 5.7.14
mydb 2 Wed Sep 5 00:38:33 2018 DEPLOYED mysql-0.10.1 5.7.14
default default
```

helm list --all 则会显示所有的 release，包括已经被删除的

由于 Helm 保留已删除 release 的记录，因此不能重新使用 release 名称。（如果确实需要重新使用此 release 名称，则可以使用此 --replace 参数，但它只会重用现有 release 并替换其资源。）这点是不是和 docker container 的管理比较类似

请注意，因为 release 以这种方式保存，所以可以回滚已删除的资源并重新激活它。

如果要彻底删除 release，则需要加上 --purge 参数：

```
$ helm delete mewing-squid --purge  
release "mewing-squid" deleted
```

我们这里只是讲到了 Helm 的一些常用的方法，更多用法我们将在后面的课程中和大家接触到。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

44. Helm 模板之内置函数和Values

上节课和大家一起学习了 Helm 的一些常用操作方法，这节课来和大家一起定义一个 chart 包，了解 Helm 中模板的一些使用方法。

定义 chart

Helm 的 github 上面有一个比较[完整的文档](#)，建议大家好好阅读下该文档，这里我们来一起创建一个 chart 包。

一个 chart 包就是一个文件夹的集合，文件夹名称就是 chart 包的名称，比如创建一个 mychart 的 chart 包：

```
$ helm create mychart
Creating mychart
$ tree mychart/
mychart/
├── charts
├── Chart.yaml
└── templates
    ├── deployment.yaml
    ├── _helpers.tpl
    ├── ingress.yaml
    ├── NOTES.txt
    └── service.yaml
└── values.yaml

2 directories, 7 files
```

chart 包的目录上节课我们就已经学习过了，这里我们再来仔细看看 templates 目录下面的文件：

- NOTES.txt: chart 的“帮助文本”。这会在用户运行 helm install 时显示给用户。
- deployment.yaml: 创建 Kubernetes deployment 的基本 manifest
- service.yaml: 为 deployment 创建 service 的基本 manifest
- ingress.yaml: 创建 ingress 对象的资源清单文件
- _helpers.tpl: 放置模板助手的地方，可以在整个 chart 中重复使用

这里我们明白每一个文件是干嘛的就行，然后我们把 templates 目录下面所有文件全部删除掉，这里我们自己来创建模板文件：

```
$ rm -rf mychart/templates/*.*
```

创建模板

这里我们来创建一个非常简单的模板 ConfigMap，在 templates 目录下面新建一个 configmap.yaml 文件：

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"

```

实际上现在我们就有一个可安装的 chart 包了，通过 `helm install` 命令来进行安装：

```

$ helm install ./mychart/
NAME: ringed-lynx
LAST DEPLOYED: Fri Sep 7 22:59:22 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
--> v1/ConfigMap
NAME          DATA   AGE
mychart-configmap  1      0s

```

在上面的输出中，我们可以看到我们的 ConfigMap 资源对象已经创建了。然后使用如下命令我们可以看到实际的模板被渲染过后的资源文件：

```

$ helm get manifest ringed-lynx

---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"

```

现在我们看到上面的 ConfigMap 文件是不是正是我们前面在模板文件中设计的，现在我们删除当前的 `release`：

```

$ helm delete ringed-lynx
release "ringed-lynx" deleted

```

添加一个简单的模板

我们可以看到上面我们定义的 ConfigMap 的名字是固定的，但往往这并不是一种很好的做法，我们可以通过插入 `release` 的名称来生成资源的名称，比如这里 ConfigMap 的名称我们希望是：`ringed-lynx-configmap`，这就需要用到 Chart 的模板定义方法了。

Helm Chart 模板使用的是 [Go 语言模板](#)编写而成，并添加了 [Sprig 库](#)中的50多个附件模板函数以及一些其他[特殊的函](#)。

需要注意的是 kubernetes 资源对象的 labels 和 name 定义被限制 63个字符，所以需要注意名称的定义。

现在我们来重新定义上面的 configmap.yaml 文件：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
```

我们将名称替换成了 {{ .Release.Name }}-configmap，其中包含在 {{ 和 }} 之中的就是模板指令，{{ .Release.Name }} 将 release 的名称注入到模板中来，这样最终生成的 ConfigMap 名称就是以 release 的名称开头的了。这里的 Release 模板对象属于 Helm 内置的一种对象，还有其他很多内置的对象，稍后我们将接触到。

现在我们来重新安装我们的 Chart 包，注意观察 ConfigMap 资源对象的名称：

```
$ helm install ./mychart
helm install ./mychart/
NAME: quoting-zebra
LAST DEPLOYED: Fri Sep 7 23:20:12 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME          DATA   AGE
quoting-zebra-configmap  1      0s
```

可以看到现在生成的名称变成了quoting-zebra-configmap，证明已经生效了，当然我们也可以使用命令 helm get manifest quoting-zebra 查看最终生成的清单文件的样子。

调试

我们用模板来生成资源文件的清单，但是如果我们要想调试就非常不方便了，不可能我们每次都去部署一个 release 实例来校验模板是否正确，所幸的是 Helm 为我们提供了 --dry-run --debug 这个可选参数，在执行 helm install 的时候带上这两个参数就可以把对应的 values 值和生成的最终的资源清单文件打印出来，而不会真正的去部署一个 release 实例，比如我们来调试上面创建的 chart 包：

```
$ helm install . --dry-run --debug ./mychart
[debug] Created tunnel using local port: '35286'

[debug] SERVER: "127.0.0.1:35286"

[debug] Original chart version: ""
[debug] CHART PATH: /root/course/kubeadm/helm/mychart

NAME: wrapping-bunny
REVISION: 1
```

```

RELEASED: Fri Sep 7 23:23:09 2018
CHART: mychart-0.1.0
USER-SUPPLIED VALUES:
{}

COMPUTED VALUES:
...
HOOKS:
MANIFEST:

---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: wrapping-bunny-configmap
data:
  myvalue: "Hello World"

```

现在我们使用 `--dry-run` 就可以很容易地测试代码了，不需要每次都去安装一个 release 实例了，但是要注意的是这不能确保 Kubernetes 本身就一定会接受生成的模板，在调试完成后，还是需要去安装一个实际的 release 实例来进行验证的。

内置对象

刚刚我们使用 `{{.Release.Name}}` 将 release 的名称插入到模板中。这里的 Release 就是 Helm 的内置对象，下面是一些常用的内置对象，在需要的时候直接使用就可以：

- **Release**: 这个对象描述了 release 本身。它里面有几个对象：
 - `Release.Name`: release 名称
 - `Release.Time`: release 的时间
 - `Release.Namespace`: release 的 namespace (如果清单未覆盖)
 - `Release.Service`: release 服务的名称 (始终是 Tiller) 。
 - `Release.Revision`: 此 release 的修订版本号，从1开始累加。
 - `Release.IsUpgrade`: 如果当前操作是升级或回滚，则将其设置为 true。
 - `Release.IsInstall`: 如果当前操作是安装，则设置为 true。
- **Values**: 从 `values.yaml` 文件和用户提供的文件传入模板的值。默认情况下，`Values` 是空的。
- **Chart**: `Chart.yaml` 文件的内容。所有的 Chart 对象都将从该文件中获取。chart 指南中[Charts Guide](#)列出了可用字段，可以前往查看。
- **Files**: 这提供对 chart 中所有非特殊文件的访问。虽然无法使用它来访问模板，但可以使用它来访问 chart 中的其他文件。请参阅 "访问文件" 部分。
 - `Files.Get` 是一个按名称获取文件的函数 (`.Files.Get config.ini`)
 - `Files.GetBytes` 是将文件内容作为字节数组而不是字符串获取的函数。这对于像图片这样的东西很有用。
- **Capabilities**: 这提供了关于 Kubernetes 集群支持的功能的信息。
 - `Capabilities.APIVersions` 是一组版本信息。

- Capabilities.APIVersions.Has \$version 指示是否在群集上启用版本 (batch/v1) 。
- Capabilities.KubeVersion 提供了查找 Kubernetes 版本的方法。它具有以下值： Major, Minor, GitVersion, GitCommit, GitTreeState, BuildDate, GoVersion, Compiler, 和 Platform。
- Capabilities.TillerVersion 提供了查找 Tiller 版本的方法。它具有以下值： SemVer, GitCommit, 和 GitTreeState。
- Template：包含有关正在执行的当前模板的信息
- Name：到当前模板的文件路径（例如 mychart/templates/mytemplate.yaml）
- BasePath：当前 chart 模板目录的路径（例如 mychart/templates）。

上面这些值可用于任何顶级模板，要注意内置值始终以大写字母开头。这也符合 Go 的命名约定。当你创建自己的名字时，你可以自由地使用适合你的团队的惯例。

values 文件

上面的内置对象中有一个对象就是 Values，该对象提供对传入 chart 的值的访问，Values 对象的值有4个来源：

- chart 包中的 values.yaml 文件
- 父 chart 包的 values.yaml 文件
- 通过 helm install 或者 helm upgrade 的 -f 或者 --values 参数传入的自定义的 yaml 文件(上节课我们已经学习过)
- 通过 --set 参数传入的值

chart 的 values.yaml 提供的值可以被用户提供的 values 文件覆盖，而该文件同样可以被 --set 提供的参数所覆盖。

这里我们来重新编辑 mychart/values.yaml 文件，将默认的值全部清空，添加一个新的数据：
(values.yaml)

```
course: k8s
```

然后我们在上面的 templates/configmap.yaml 模板文件中就可以使用这个值了：(configmap.yaml)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  course: {{ .Values.course }}
```

可以看到最后一行我们是通过 {{ .Values.course }} 来获取 course 的值的。现在我们用 debug 模式来查看下我们的模板会被如何渲染：

```
$ helm install --dry-run --debug ./mychart
```

```

helm install --dry-run --debug .
[debug] Created tunnel using local port: '33509'

[debug] SERVER: "127.0.0.1:33509"

[debug] Original chart version: ""
[debug] CHART PATH: /root/course/kubeadm/helm/mychart

NAME: nasal-anaconda
REVISION: 1
RELEASED: Sun Sep 9 17:37:52 2018
CHART: mychart-0.1.0
USER-SUPPLIED VALUES:
{}

COMPUTED VALUES:
course: k8s

HOOKS:
MANIFEST:

---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nasal-anaconda-configmap
data:
  myvalue: "Hello World"
  course: k8s

```

我们可以看到 ConfigMap 中 course 的值被渲染成了 k8s，这是因为在默认的 values.yaml 文件中该参数值为 k8s，同样的我们可以通过 --set 参数来轻松的覆盖 course 的值：

```

$ helm install --dry-run --debug --set course=python ./mychart
[debug] Created tunnel using local port: '44571'

.....
---

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: named-scorpion-configmap
data:
  myvalue: "Hello World"
  course: python

```

由于 --set 比默认 values.yaml 文件具有更高的优先级，所以我们的模板生成为 course: python。

values 文件也可以包含更多结构化内容，例如，我们在 values.yaml 文件中可以创建 course 部分，然后在其中添加几个键：

```

course:
  k8s: devops

```

```
python: django
```

现在我们稍微修改模板：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  k8s: {{ .Values.course.k8s }}
  python: {{ .Values.course.python }}
```

同样可以使用 debug 模式查看渲染结果：

```
$ helm install --dry-run --debug ./mychart
[debug] Created tunnel using local port: '33801'
...
---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: exhaling-turtle-configmap
data:
  myvalue: "Hello World"
  k8s: devops
  python: django
```

可以看到模板中的参数已经被 values.yaml 文件中的值给替换掉了。虽然以这种方式构建数据是可以的，但我们还是建议保持 value 树浅一些，平一些，这样维护起来要简单一点。

到这里，我们已经看到了几个内置对象的使用方法，并用它们将信息注入到了模板之中。下节课我们来看看模板引擎中的其他用法，比如函数、管道、控制结构等等的用法。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:04:21

45. Helm 模板之模板函数与管道

上节课我们学习了如何将信息渲染到模板之中，但是这些信息都是直接传入模板引擎中进行渲染的，有的时候我们想要转换一下这些数据才进行渲染，这就需要使用到 Go 模板语言中的一些其他用法。

模板函数

比如我们需要从 `.Values` 中读取的值变成字符串的时候就可以通过调用 `quote` 模板函数来实现：
(`templates/configmap.yaml`)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  k8s: {{ quote .Values.course.k8s }}
  python: {{ .Values.course.python }}
```

模板函数遵循调用的语法为：`functionName arg1 arg2...`。在上面的模板文件中，`quote .Values.course.k8s` 调用 `quote` 函数并将后面的值作为一个参数传递给它。最终被渲染为：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '39405'
...
#
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: masked-saola-configmap
data:
  myvalue: "Hello World"
  k8s: "devops"
  python: django
```

我们可以看到 `.Values.course.k8s` 被渲染成了字符串 `devops`。上节课我们也提到过 Helm 是一种 Go 模板语言，拥有超过60多种可用的内置函数，一部分是由[Go 模板语言](#)本身定义的，其他大部分都是[Sprig 模板库](#)提供的一部分，我们可以前往这两个文档中查看这些函数的用法。

比如我们这里使用的 `quote` 函数就是 `Sprig` 模板库 提供的一种字符串函数，用途就是用双引号将字符串括起来，如果需要双引号 `"`，则需要添加 `\` 来进行转义，而 `squote` 函数的用途则是用双引号将字符串括起来，而不会对内容进行转义。

所以在我们遇到一些需求的时候，首先要想到的是去查看下上面的两个模板文档中是否提供了对应的模板函数，这些模板函数可以很好的解决我们的需求。

管道

模板语言除了提供了丰富的内置函数之外，其另一个强大的功能就是管道的概念。和 UNIX 中一样，管道我们通常称为 `Pipeline`，是一个链在一起的一系列模板命令的工具，以紧凑地表达一系列转换。简单来说，管道是可以按顺序完成一系列事情的一种方法。比如我们用管道来重写上面的 ConfigMap 模板：(templates/configmap.yaml)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  k8s: {{ .Values.course.k8s | quote }}
  python: {{ .Values.course.python }}
```

这里我们直接调用 `quote` 函数，而是调换了一个顺序，使用一个管道符 `|` 将前面的参数发送给后面的模板函数：`{{ .Values.course.k8s | quote }}`，使用管道我们可以将几个功能顺序的连接在一起，比如我们希望上面的 ConfigMap 模板中的 `k8s` 的 value 值被渲染后是大写的字符串，则我们就可以使用管道来修改：(templates/configmap.yaml)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  k8s: {{ .Values.course.k8s | upper | quote }}
  python: {{ .Values.course.python }}
```

这里我们在管道中增加了一个 `upper` 函数，该函数同样是 [Sprig 模板库](#) 提供的，表示将字符串每一个字母都变成大写。然后我们用 `debug` 模式来查看下上面的模板最终会被渲染成什么样子：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '46651'
.....
-----
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: maudlin-labradoodle-configmap
data:
  myvalue: "Hello World"
  k8s: "DEVOPS"
  python: django
```

我们可以看到之前我们的 `devops` 已经被渲染成了 `"DEVOPS"` 了，要注意的是使用管道操作的时候，前面的操作结果会作为参数传递给后面的模板函数，比如我们这里希望将上面模板中的 `python` 的值渲染为重复出现3次的字符串，则我们就可以使用到 [Sprig 模板库](#) 提供的 `repeat` 函数，不过该函数需要传入一个参数 `repeat COUNT STRING` 表示重复的次数：(templates/configmap.yaml)

```
apiVersion: v1
```

```

kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  k8s: {{ .Values.course.k8s | upper | quote }}
  python: {{ .Values.course.python | quote | repeat 3 }}

```

该 `repeat` 函数会将给定的字符串重复3次返回，所以我们将得到这个输出：

```

helm install --dry-run --debug .
[debug] Created tunnel using local port: '39712'

.....
Error: YAML parse error on mychart/templates/configmap.yaml: error converting YAML to JSON
: yaml: line 7: did not find expected key

---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: piquant-butterfly-configmap
data:
  myvalue: "Hello World"
  k8s: "DEVOPS"
  python: "django""django""django"

```

我们可以看到上面的输出中 `python` 对应的值变成了3个相同的字符串，这显然是不符合我们预期的，我们的预期是形成一个字符串，而现在是3个字符串了，而且上面还有错误信息，根据管道处理的顺序，我们将 `quote` 函数放到 `repeat` 函数后面去是不是就可以解决这个问题了：

(templates/configmap.yaml)

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  k8s: {{ .Values.course.k8s | upper | quote }}
  python: {{ .Values.course.python | repeat 3 | quote }}

```

现在是不是就是先重复3次 `.Values.course.python` 的值，然后调用 `quote` 函数：

```

helm install --dry-run --debug .
[debug] Created tunnel using local port: '33837'

.....
---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap

```

```

metadata:
  name: braided-manatee-configmap
data:
  myvalue: "Hello World"
  k8s: "DEVOPS"
  python: "djangodjangodjango"

```

现在是不是就正常了，也得到了我们的预期结果，所以我们在使用管道操作的时候一定要注意是按照从前到后一步一步顺序处理的。

default 函数

另外一个我们会经常使用的一个函数是 `default` 函数：`default DEFAULT_VALUE GIVEN_VALUE`。该函数允许我们在模板内部指定默认值，以防止该值被忽略掉了。比如我们来修改上面的 ConfigMap 的模板：（`templates/configmap.yaml`）

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: {{ .Values.hello | default "Hello World" | quote }}
  k8s: {{ .Values.course.k8s | upper | quote }}
  python: {{ .Values.course.python | repeat 5 | quote }}

```

由于我们的 `values.yaml` 文件中只定义了 `course` 结构的信息，并没有定义 `hello` 的值，所以如果没有设置默认值的话是得不到 `{{ .Values.hello }}` 的值的，这里我们为该值定义了一个默认值：`Hello World`，所以现在如果在 `values.yaml` 文件中没有定义这个值，则我们也可以得到默认值：

```

$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '42670'

.....
```
Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
 name: orbiting-hog-configmap
data:
 myvalue: "Hello World"
 k8s: "DEVOPS"
 python: "djangodjangodjangodjango"
```

```

我们可以看到 `myvalue` 值被渲染成了 `Hello World`，证明我们的默认值生效了。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:04:47

46. Helm 模板之控制流程

模板函数和管道 是通过转换信息并将其插入到 YAML 文件中的强大方法。但有时候需要添加一些比插入字符串更复杂一些的模板逻辑。这就需要使用到模板语言中提供的控制结构了。

控制流程为我们提供了控制模板生成流程的一种能力，Helm 的模板语言提供了以下几种流程控制：

- `if/else` 条件块
- `with` 指定范围
- `range` 循环块

除此之外，它还提供了一些声明和使用命名模板段的操作：

- `define` 在模板中声明一个新的命名模板
- `template` 导入一个命名模板
- `block` 声明了一种特殊的可填写的模板区域

关于 `命名模板` 的相关知识点，我们会在后面的课程中和大家接触到，这里我们暂时和大家介绍 `if/else`、`with`、`range` 这3中控制流程的用法。

if/else 条件

`if/else` 块是用于在模板中有条件地包含文本块的方法，条件块的基本结构如下：

```
{{ if PIPELINE }}
  # Do something
{{ else if OTHER PIPELINE }}
  # Do something else
{{ else }}
  # Default case
{{ end }}
```

当然要使用条件块就得判断条件是否为真，如果值为下面的几种情况，则管道的结果为 `false`：

- 一个布尔类型的 `假`
- 一个数字 `零`
- 一个 `空` 的字符串
- 一个 `nil` (`空`或 `null`)
- 一个空的集合 (`map`、`slice`、`tuple`、`dict`、`array`)

除了上面的这些情况外，其他所有条件都为 `真`。

同样还是以上面的 ConfigMap 模板文件为例，添加一个简单的条件判断，如果 `python` 被设置为 `django`，则添加一个 `web: true : (tempaltes/configmap.yaml)`

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```

name: {{ .Release.Name }}-configmap
data:
  myvalue: {{ .Values.hello | default "Hello World" | quote }}
  k8s: {{ .Values.course.k8s | upper | quote }}
  python: {{ .Values.course.python | repeat 3 | quote }}
{{ if eq .Values.course.python "django" }}web: true{{ end }}

```

在上面的模板文件中我们增加了一个条件语句判断 {{ if eq .Values.course.python "django" }}web: true{{ end }}，其中运算符 eq 是判断是否相等的操作，除此之外，还有 ne 、 lt 、 gt 、 and 、 or 等运算符都是 Helm 模板已经实现了的，直接使用即可。这里我们 {{ .Values.course.python }} 的值在 values.yaml 文件中默认被设置为了 django，所以正常来说下面的条件语句判断为真，所以模板文件最终被渲染后会有 web: true 这样的的一个条目：

```

$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '40143'

      ...
      ...

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fallacious-prawn-configmap
data:
  myvalue: "Hello World"
  k8s: "DEVOPS"
  python: "djangodjangodjangodjangodjango"
  web: true

```

可以看到上面模板被渲染后出现了 web: true 的条目，如果我们在安装的时候覆盖下 python 的值呢，比如我们改成 ai:

```

helm install --dry-run --debug --set course.python=ai .
[debug] Created tunnel using local port: '42802'

      ...
      ...

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: dull-mite-configmap
data:
  myvalue: "Hello World"
  k8s: "DEVOPS"
  python: "aiaiai"

```

根据我们模板文件中的定义，如果 {{ .Values.course.python }} 的值为 django 的话就会新增 web: true 这样的一个条目，但是现在我们是不是通过参数 --set 将值设置为了 ai，所以这里条件判断为假，正常来说就不应该出现这个条目了，上面我们通过 debug 模式查看最终被渲染的值也没有出现这个条目，证明条件判断是正确的。

空格控制

上面我们的条件判断语句是在一整行中的，如果平时经常写代码的同学可能非常不习惯了，我们一般会将其格式化为更容易阅读的形式，比如：

```
{{ if eq .Values.course.python "django" }}
  web: true
{{ end }}
```

这样的话看上去比之前要清晰很多了，但是我们通过模板引擎来渲染一下，会得到如下结果：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '44537'

      ...
      ...

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: bald-narwhal-configmap
data:
  myvalue: "Hello World"
  k8s: "DEVOPS"
  python: "djangodjangodjango"

  web: true
```

我们可以看到渲染出来会有多余的空行，这是因为当模板引擎运行时，它将一些值渲染过后，之前的指令被删除，但它之前所占的位置完全按原样保留剩余的空白了，所以就出现了多余的空行。`YAML` 文件中的空格是非常严格的，所以对于空格的管理非常重要，一不小心就会导致你的`YAML`文件格式错误。

我们可以通过使用在模板标识`{{`后面添加破折号和空格`{}- 来表示将空白左移，而在}} 前面添加一个空格和破折号-}} 表示应该删除右边的空格，另外需要注意的是换行符也是空格！`

使用这个语法，我们来修改我们上面的模板文件去掉多余的空格：`(templates/configmap.yaml)`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: {{ .Values.hello | default "Hello World" | quote }}
  k8s: {{ .Values.course.k8s | upper | quote }}
  python: {{ .Values.course.python | repeat 3 | quote }}
{{- if eq .Values.course.python "django" }}
  web: true
{{- end }}
```

现在我们来查看上面模板渲染过后的样子：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '34702'

.....
---

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mangy-olm-configmap
data:
  myvalue: "Hello World"
  k8s: "DEVOPS"
  python: "djangodjangodjango"
  web: true
```

现在是不是没有多余的空格了，另外我们需要谨慎使用 `-}}`，比如上面模板文件中：

```
python: {{ .Values.course.python | repeat 3 | quote }}
{{- if eq .Values.course.python "django" -}}
web: true
{{- end }}
```

如果我们在 `if` 条件后面增加 `-}}`，这会渲染成：

```
python: "djangodjangodjango"web: true
```

因为 `-}}` 它删除了双方的换行符，显然这是不正确的。

有关模板中空格控制的详细信息，请参阅官方 Go 模板文档[Official Go template documentation](#)

使用 `with` 修改范围

接下来我们来看下 `with` 关键词的使用，它用来控制变量作用域。还记得之前我们的 `{{ .Release.xxx }}` 或者 `{{ .Values.xxx }}` 吗？其中的 `.` 就是表示对当前范围的引用，`.Values` 就是告诉模板在当前范围内查找 `Values` 对象的值。而 `with` 语句就可以来控制变量的作用域范围，其语法和一个简单的 `if` 语句比较类似：

```
{{ with PIPELINE }}
  # restricted scope
{{ end }}
```

`with` 语句可以允许将当前范围 `.` 设置为特定的对象，比如我们前面一直使用的 `.Values.course`，我们可以使用 `with` 来将 `.` 范围指向 `.Values.course`：(templates/configmap.yaml)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
```

```

data:
  myvalue: {{ .Values.hello | default "Hello World" | quote }}
{{- with .Values.course -}}
  k8s: {{ .k8s | upper | quote }}
  python: {{ .python | repeat 3 | quote }}
{{- if eq .python "django" -}}
  web: true
{{- end -}}
{{- end -}}

```

可以看到上面我们增加了一个 `{{- with .Values.course -}}` 的一个块，这样的话我们就可以在当前的块里面直接引用 `.python` 和 `.k8s` 了，而不需要进行限定了，这是因为该 `with` 声明将 `.` 指向了 `.Values.course`，在 `{- end -}` 后 `.` 就会复原其之前的作用范围了，我们可以使用模板引擎来渲染上面的模板查看是否符合预期结果。

不过需要注意的是在 `with` 声明的范围内，此时将无法从父范围访问到其他对象了，比如下面的模板渲染的时候将会报错，因为显然 `.Release` 根本就不在当前的 `.` 范围内，当然如果我们最后两行交换下位置就正常了，因为 `{- end -}` 之后范围就被重置了：

```

{{- with .Values.course -}}
  k8s: {{ .k8s | upper | quote }}
  python: {{ .python | repeat 3 | quote }}
  release: {{ .Release.Name }}
{{- end -}}

```

range 循环

如果大家对编程语言熟悉的话，几乎所有的编程语言都支持类似于 `for`、`foreach` 或者类似功能的循环机制，在 Helm 模板语言中，是使用 `range` 关键字来进行循环操作。

我们在 `values.yaml` 文件中添加上一个课程列表：

```

course:
  k8s: devops
  python: django
  courselist:
    - k8s
    - python
    - search
    - golang

```

现在我们有一个课程列表，修改 ConfigMap 模板文件来循环打印出该列表：

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: {{ .Values.hello | default "Hello World" | quote }}
{{- with .Values.course -}}
  k8s: {{ .k8s | upper | quote }}

```

```

python: {{ .python | repeat 3 | quote }}
{{- if eq .python "django" -}}
web: true
{{- end -}}
{{- end -}}
courselist:
{{- range .Values.courselist -}}
- {{ . | title | quote }}
{{- end -}}

```

可以看到最下面我们使用了一个 `range` 函数，该函数将会遍历 `{{ .Values.courselist }}` 列表，循环内部我们使用的是一个 `.`，这是因为当前的作用域就在当前循环内，这个 `.` 从列表的第一个元素一直遍历到最后一个元素，然后在遍历过程中使用了 `title` 和 `quote` 这两个函数，前面这个函数是将字符串首字母变成大写，后面就是加上双引号变成字符串，所以按照上面这个模板被渲染过后的结果为：

```

$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '34626'

.....
-- 
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: dining-terrier-configmap
data:
  myvalue: "Hello World"
  k8s: "DEVOPS"
  python: "djangodjangodjango"
  web: true
  courselist:
    - "K8s"
    - "Python"
    - "Search"
    - "Golang"

```

我们可以看到 `courselist` 按照我们的要求循环出来了。除了 `list` 或者 `tuple`，`range` 还可以用于遍历具有键和值的集合（如`map` 或 `dict`），这个就需要用到变量的概念了。

变量

前面我们已经学习了函数、管理以及控制流程的使用方法，我们知道编程语言中还有一个很重要的概念叫：变量，在 Helm 模板中，使用变量的场合不是特别多，但是在合适的时候使用变量可以很好的解决我们的问题。如下面的模板：

```

{{- with .Values.course -}}
k8s: {{ .k8s | upper | quote }}
python: {{ .python | repeat 3 | quote }}
release: {{ .Release.Name }}
{{- end -}}

```

我们在 `with` 语句块内添加了一个 `.Release.Name` 对象，但这个模板是错误的，编译的时候会失败，这是因为 `.Release.Name` 不在该 `with` 语句块限制的作用范围之内，我们可以将该对象赋值给一个变量可以来解决这个问题：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  {{- $releaseName := .Release.Name -}}
  {{- with .Values.course -}}
    k8s: {{ .k8s | upper | quote }}
    python: {{ .python | repeat 3 | quote }}
    release: {{ $releaseName }}
  {{- end -}}
```

我们可以看到我们在 `with` 语句上面增加了一句 `{{- $releaseName := .Release.Name -}}`，其中 `$releaseName` 就是后面的对象的一个引用变量，它的形式就是 `$name`，赋值操作使用 `:=`，这样 `with` 语句块内部的 `$releaseName` 变量仍然指向的是 `.Release.Name`，同样，我们 DEBUG 下查看结果：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '45474'

.....
---

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nosy-seagull-configmap
data:
  k8s: "DEVOPS"
  python: "djangodjangodjango"
  release: nosy-seagull
```

可以看到已经正常了，另外变量在 `range` 循环中也非常有用，我们可以在循环中用变量来同时捕获索引的值：

```
courselist:
{{- range $index, $course := .Values.courselist -}}
- {{ $index }}: {{ $course | title | quote }}
{{- end -}}
```

例如上面的这个列表，我们在 `range` 循环中使用 `$index` 和 `$course` 两个变量来接收后面列表循环的索引和对应的值，最终可以得到如下结果：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '38876'

.....
```

```
---  
# Source: mychart/templates/configmap.yaml  
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: vetoed-anaconda-configmap  
data:  
  courselist:  
    - 0: "K8s"  
    - 1: "Python"  
    - 2: "Search"  
    - 3: "Golang"
```

我们可以看到 `courselist` 下面将索引和对应的值都打印出来了，实际上具有键和值的数据结构我们都可以使用 `range` 来循环获得二者的值，比如我们可以对 `.Values.course` 这个字典来进行循环：

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: {{ .Release.Name }}-configmap  
data:  
{{- range $key, $value := .Values.course }}  
{{ $key }}: {{ $value | quote }}  
{{- end }}
```

直接使用 `range` 循环，用变量 `$key`、`$value` 来接收字段 `.Values.course` 的键和值。这就是变量在 Helm 模板中的使用方法。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:04:53

47. Helm模板之命名模板

前面我们学习了一些 Helm 模板中的一些常用使用方法，但是我们都是操作的一个模板文件，在实际的应用中，很多都是相对比较复杂的，往往会超过一个模板，如果有多个应用模板，我们应该如何进行处理呢？这就需要用到新的概念：命名模板。

命名模板我们也可以称为子模板，是限定在一个文件内部的模板，然后给一个名称。在使用命名模板的时候有一个需要特别注意的是：模板名称是全局的，如果我们声明了两个相同名称的模板，最后加载的一个模板会覆盖掉另外的模板，由于子 chart 中的模板也是和顶层的模板一起编译的，所以在命名的时候一定要注意，不要重名了。

为了避免重名，有个通用的约定就是为每个定义的模板添加上 chart 名称： `{{define "mychart.labels"}}`，`define` 关键字就是用来声明命名模板的，加上 chart 名称就可以避免不同 chart 间的模板出现冲突的情况。

声明和使用命名模板

使用 `define` 关键字就可以允许我们在模板文件内部创建一个命名模板，它的语法格式如下：

```
 {{ define "ChartName.TplName" }}
# 模板内容区域
{{ end }}
```

比如，现在我们可以定义一个模板来封装一个 label 标签：

```
 {{- define "mychart.labels" }}
  labels:
    from: helm
    date: {{ now | htmlDate }}
{{- end }}
```

然后我们可以将该模板嵌入到现有的 ConfigMap 中，然后使用 `template` 关键字在需要的地方包含进来即可：

```
 {{- define "mychart.labels" }}
  labels:
    from: helm
    date: {{ now | htmlDate }}
{{- end }}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "mychart.labels" }}
data:
  {{- range $key, $value := .Values.course }}
    {{ $key }}: {{ $value | quote }}
  {{- end }}
```

我们这个模板文件被渲染过后的结果如下所示：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '42058'
.....
---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: ardent-bunny-configmap
  labels:
    from: helm
    date: 2018-09-22
data:
  k8s: "devops"
  python: "django"
```

我们可以看到 `define` 区域定义的命名模板被嵌入到了 `template` 所在的区域，但是如果我们将命名模板全都写入到一个模板文件中的话无疑也会增大模板的复杂性。

还记得我们在创建 `chart` 包的时候，`templates` 目录下面默认会生成一个 `_helpers.tpl` 文件吗？我们前面也提到过 `templates` 目录下面除了 `NOTES.txt` 文件和以下划线 `_` 开头命令的文件之外，都会被当做 `kubernetes` 的资源清单文件，而这个下划线开头的文件不会被当做资源清单外，还可以被其他 `chart` 模板中调用，这个就是 `Helm` 中的 `partials` 文件，所以其实我们完全就可以将命名模板定义在这些 `partials` 文件中，默认就是 `_helpers.tpl` 文件了。

现在我们将上面定义的命名模板移动到 `templates/_helpers.tpl` 文件中去：

```
{{-- 生成基本的 labels 标签 --}}
{{- define "mychart.labels" -}}
  labels:
    from: helm
    date: {{ now | htmlDate }}
{{- end -}}
```

一般情况下面，我们都会在命名模板头部加一个简单的文档块，用 `/**/` 包裹起来，用来描述我们这个命名模板的用途的。

现在我们讲命名模板从模板文件 `templates/configmap.yaml` 中移除，当然还是需要保留 `template` 来嵌入命名模板内容，名称还是之前的 `mychart.labels`，这是因为模板名称是全局的，所以我们可以直接获取到。我们再用 `DEBUG` 模式来调试下是否符合预期？

模板范围

上面我们定义的命名模板中，没有使用任何对象，只是使用了一个简单的函数，如果我们在里面来使用 `chart` 对象相关信息呢：

```
{{-- 生成基本的 labels 标签 --}}
{{- define "mychart.labels" -}}
```

```

{{- define "mychart.labels" -}}
labels:
  from: helm
  date: {{ now | htmlDate }}
  chart: {{ .Chart.Name }}
  version: {{ .Chart.Version }}
{{- end -}}

```

如果这样的直接进行渲染测试的话，是不会得到我们的预期结果的：

```

$ $ helm install --dry-run --debug .
[debug] Created tunnel using local port: '42058'

.....
---

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: peeking-zorse-configmap
  labels:
    from: helm
    date: 2018-09-22
    chart:
    version:
data:
  k8s: "devops"
  python: "django"

```

chart 的名称和版本都没有正确被渲染，这是因为他们不在我们定义的模板范围内，当命名模板被渲染时，它会接收由 template 调用时传入的作用域，而我们这里并没有传入对应的作用域，因此模板中我们无法调用到 .Chart 对象，要解决也非常简单，我们只需要在 template 后面加上作用域范围即可：

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
{{- template "mychart.labels" . -}}
data:
{{- range $key, $value := .Values.course -}}
{{ $key }}: {{ $value | quote }}
{{- end -}}

```

我们在 template 末尾传递了 .，表示当前的最顶层的作用范围，如果我们想要在命名模板中使用 .Values 范围内的数据，当然也是可以的，现在我们再来渲染下我们的模板：

```

$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '32768'

.....
---

```

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: oldfashioned-mule-configmap
  labels:
    from: helm
    date: 2018-09-22
    chart: mychart
    version: 0.1.0
data:
  k8s: "devops"
  python: "django"
```

我们可以看到 chart 的名称和版本号都已经被正常渲染出来了。

include 函数

假如现在我们将上面的定义的 labels 单独提取出来放置到 _helpers.tpl 文件中：

```
{{/* 生成基本的 labels 标签 */}}
{{- define "mychart.labels" -}}
  from: helm
  date: {{ now | htmlDate }}
  chart: {{ .Chart.Name }}
  version: {{ .Chart.Version }}
{{- end -}}
```

现在我们将该命名模板插入到 configmap 模板文件的 labels 部分和 data 部分：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  labels:
    {{- template "mychart.labels" . -}}
data:
  {{- range $key, $value := .Values.course -}}
    {{ $key }}: {{ $value | quote }}
  {{- end -}}
  {{- template "mychart.labels" . -}}
```

然后同样的查看下渲染的结果：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '42652'

      ...
Error: YAML parse error on mychart/templates/configmap.yaml: error converting YAML to JSON
: yaml: line 9: mapping values are not allowed in this context

---
# Source: mychart/templates/configmap.yaml
```

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: altered-wombat-configmap
  labels:
    from: helm
    date: 2018-09-22
    chart: mychart
    version: 0.1.0
  data:
    k8s: "devops"
    python: "django"
    from: helm
    date: 2018-09-22
    chart: mychart
    version: 0.1.0

```

我们可以看到渲染结果是有问题的，不是一个正常的 YAML 文件格式，这是因为 `template` 只是表示一个嵌入动作而已，不是一个函数，所以原本命名模板中是怎样的格式就是怎样的格式被嵌入进来了，比如我们可以在命名模板中给内容区域都空了两个空格，再来查看下渲染的结构：

```

---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mortal-cricket-configmap
  labels:
    from: helm
    date: 2018-09-22
    chart: mychart
    version: 0.1.0
  data:
    k8s: "devops"
    python: "django"
    from: helm
    date: 2018-09-22
    chart: mychart
    version: 0.1.0

```

我们可以看到 `data` 区域里面的内容是渲染正确的，但是上面 `labels` 区域是不正常的，因为命名模板里面的内容是属于 `labels` 标签的，是不符合我们的预期的，但是我们又不可能再去把命名模板里面的内容再增加两个空格，因为这样的话 `data` 里面的格式又不符合预期了。

为了解决这个问题，Helm 提供了另外一个方案来代替 `template`，那就是使用 `include` 函数，在需要控制空格的地方使用 `indent` 管道函数来自己控制，比如上面的例子我们替换成 `include` 函数：

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  labels:
{{- include "mychart.labels" . | indent 4 }}
  data:
{{- range $key, $value := .Values.course }}

```

```
  {{ $key }}: {{ $value | quote }}
{{- end }}
{{- include "mychart.labels" . | indent 2 }}
```

在 labels 区域我们需要4个空格，所以在管道函数 `indent` 中，传入参数4就可以，而在 data 区域我们只需要2个空格，所以我们传入参数2即可以，现在我们来渲染下我们这个模板看看是否符合预期呢：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '38481'

      .
      .

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: torpid-bobcat-configmap
  labels:
    from: helm
    date: 2018-09-22
    chart: mychart
    version: 0.1.0
data:
  k8s: "devops"
  python: "django"
  from: helm
  date: 2018-09-22
  chart: mychart
  version: 0.1.0
```

可以看到是符合我们的预期，所以在 Helm 模板中我们使用 `include` 函数要比 `template` 更好，可以更好地处理 YAML 文件输出格式。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:04:57

48. Helm模板之其他注意事项

上节课我们学习了命名模板的使用，命名模板是 Helm 模板中非常重要的一个功能，在我们实际开发 Helm Chart 包的时候非常有用，到这里我们基本上就把 Helm 模板中经常使用到的一些知识点和大家介绍完了。但是仍然还是有一些在开发中值得我们注意的一些知识点，比如 NOTES.txt 文件的使用、子 Chart 的使用、全局值的使用，这节课我们就来和大家一起了解下这些知识点。

NOTES.txt 文件

我们前面在使用 helm install 命令的时候，Helm 都会为我们打印出一大堆介绍信息，这样当别的用户在使用我们的 chart 包的时候就可以根据这些注释信息快速了解我们的 chart 包的使用方法，这些信息就是编写在 NOTES.txt 文件之中的，这个文件是纯文本的，但是它和其他模板一样，具有所有可用的普通模板函数和对象。

现在我们在前面的示例中 templates 目录下面创建一个 NOTES.txt 文件：

```
Thank you for installing {{ .Chart.Name }}.

Your release is named {{ .Release.Name }}.

To learn more about the release, try:

$ helm status {{ .Release.Name }}
$ helm get {{ .Release.Name }}
```

我们可以看到我们在 NOTES.txt 文件中也使用 Chart 和 Release 对象，现在我们在 mychart 包根目录下面执行安装命令查看是否能够得到上面的注释信息：

```
$ helm install .
Error: release nomadic-deer failed: ConfigMap in version "v1" cannot be handled as a ConfigMap: v1.ConfigMap: Data: ReadString: expects " or n, but found [, error found in #10 byte of ...|rselist":[{"0":"K8s"|..., bigger context ...|:{app:"mychart", chart:"mychart", courselist":[{"0":"K8s"}, {"1":"Python"}, {"2":"Search"}, {"3":"Go|...
```

我们可以看到出现了上面的错误信息，但是如果我们要去执行 debug 命令来调试的话是没有任何问题的，是可以正常渲染的，但是为什么在正式安装的时候确出现了问题呢？这是因为我们在 debug 调试阶段只是检验模板是否可以正常渲染，并没有去检查对应的 kubernetes 资源对象对 yaml 文件的格式要求，所以我们说 debug 测试通过，并不代表 chart 就真正的是可用状态，比如我们这里是一个 ConfigMap 的资源对象，ConfigMap 对 data 区域的内容是有严格要求的，比如我们这里出现了下面这样的内容：

```
web: true
courseList:
- 0: "K8s"
- 1: "Python"
- 2: "Search"
- 3: "Golang"
```

这的确是一个合法的 yaml 文件的格式，但是对 ConfigMap 就不合法了，需要把 true 变成字符串，下面的字典数组变成一个多个行的字符串，这样就是一个合法的 ConfigMap 了：(templates/config.yaml)

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  labels:
{{- include "mychart.labels" . | indent 4}}
data:
  app: mychart
  myvalue: {{ .Values.hello | default "Hello World" | quote }}
  {{- $releaseName := .Release.Name --}}
  {{- with .Values.course --}}
  k8s: {{ .k8s | upper | quote }}
  python: {{ .python | repeat 5 | quote }}
  release: {{ $releaseName }}
  {{- if eq .python "django" --}}
  web: "true"
  {{- end --}}
  {{- end --}}
  courselist: |
    {{- range $index, $course := .Values.courselist --}}
      {{ $course | title | quote }}
    {{- end --}}
    {{- range $key, $val := .Values.course --}}
      {{ $key }}: {{ $val | upper | quote }}
    {{- end --}}
{{- include "mychart.labels" . | indent 2 --}}

```

现在我们再来执行安装命令：

```

$ helm install .
NAME: nosy-pig
LAST DEPLOYED: Sat Sep 29 19:26:16 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
--> v1/ConfigMap
NAME          DATA  AGE
nosy-pig-configmap  11    0s

NOTES:
Thank you for installing mychart.

Your release is named nosy-pig.

To learn more about the release, try:

$ helm status nosy-pig
$ helm get nosy-pig

```

现在已经安装成功了，而且下面的注释部分也被渲染出来了，我们可以看到 NOTES.txt 里面使用到的模板对象都被正确渲染了。

为我们创建的 chart 包提供一个清晰的 NOTES.txt 文件是非常有必要的，可以为用户提供有关如何使用新安装 chart 的详细信息，这是一种非常友好的方式方法。

子 chart 包

我们到目前为止都只用了一个 chart，但是 chart 也可以有 子 chart 的依赖关系，它们也有自己的值和模板，在学习字 chart 之前，我们需要了解几点关于子 chart 的说明：

- 子 chart 是独立的，所以子 chart 不能明确依赖于其父 chart
- 子 chart 无法访问其父 chart 的值
- 父 chart 可以覆盖子 chart 的值
- Helm 中有全局值的概念，可以被所有的 chart 访问

创建子 chart

现在我们就来创建一个子 chart，还记得我们在创建 mychart 包的时候，在根目录下面有一个空文件夹 charts 目录吗？这就是我们的子 chart 所在的目录，在该目录下面添加一个新的 chart：

```
$ cd mychart/charts
$ helm create mysubchart
Creating mysubchart
$ rm -rf mysubchart/templates/*
$ tree ..
.
├── charts
│   └── mysubchart
│       ├── charts
│       ├── Chart.yaml
│       ├── templates
│       └── values.yaml
└── Chart.yaml
└── templates
    ├── configmap.yaml
    ├── _helpers.tpl
    └── NOTES.txt
    └── values.yaml

5 directories, 7 files
```

同样的，我们将子 chart 模板中的文件全部删除了，接下来，我们为子 chart 创建一个简单的模板和 values 文件了。

```
$ cat > mysubchart/values.yaml <<EOF
in: mysub
EOF
$ cat > mysubchart/templates/configmap.yaml <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap2
data:
  in: {{ .Values.in }}
```

EOF

我们上面已经提到过每个子 chart 都是独立的 chart，所以我们可以单独给 mysubchart 进行测试：

```
$ helm install --dry-run --debug ./mysubchart
[debug] Created tunnel using local port: '33568'

.....
---

# Source: mysubchart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: washed-indri-configmap2
data:
  in: mysub
```

我们可以看到正常渲染出了结果。

值覆盖

现在 mysubchart 这个子 chart 就属于 mychart 这个父 chart 了，由于 mychart 是父级，所以我们可以在 mychart 的 values.yaml 文件中直接配置子 chart 中的值，比如我们可以在 mychart/values.yaml 文件中添加上子 chart 的值：

```
course:
  k8s: devops
  python: django
courselist:
- k8s
- python
- search
- golang

mysubchart:
  in: parent
```

注意最后两行，mysubchart 部分内的任何指令都会传递到 mysubchart 这个子 chart 中去的，现在我们在 mychart 根目录中执行调试命令，可以查看到子 chart 也被一起渲染了：

```
$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '44798'

.....
---

# Source: mychart/charts/mysubchart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: ideal-ostrich-configmap2
data:
```

```

in: parent
---
# Source: mychart/templates/configmap.yaml
.....

```

我们可以看到子 chart 中的值已经被顶层的值给覆盖了。但是在某些场景下面我们还是希望某些值在所有模板中都可以使用，这就需要用到全局 chart 值了。

全局值

全局值可以从任何 chart 或者子 chart 中进行访问使用，values 对象中有一个保留的属性是 `Values.global`，就可以被用来设置全局值，比如我们在父 chart 的 values.yaml 文件中添加一个全局值：

```

course:
  k8s: devops
  python: django
courselist:
- k8s
- python
- search
- golang

mysubchart:
  in: parent

global:
  allin: helm

```

我们在 values.yaml 文件中添加了一个 global 的属性，这样的话无论在父 chart 中还是在子 chart 中我们都可以通过 `{{ .Values.global.allin }}` 来访问这个全局值了。比如我们在 mychart/templates/configmap.yaml 和 mychart/charts/mysubchart/templates/configmap.yaml 文件的数据区域下面都添加上如下内容：

```

...
data:
  allin: {{ .Values.global.allin }}
...

```

现在我们在 mychart 根目录下面执行 debug 调试模式：

```

$ helm install --dry-run --debug .
[debug] Created tunnel using local port: '32775'
.....
MANIFEST:

---
# Source: mychart/charts/mysubchart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap

```

```
metadata:
  name: wistful-swan-dog-configmap2
data:
  allin: helm
  in: parent
---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: wistful-swan-dog-configmap
  ...
data:
  allin: helm
  ...
  ...
```

我们可以看到两个模板中都输出了 `allin: helm` 这样的值，全局变量对于传递这样的信息非常有用，不过也要注意我们不能滥用全局值。

另外值得注意的是我们在学习命名模板的时候就提到过父 chart 和子 chart 可以共享模板。任何 chart 中的任何定义块都可用于其他 chart，所以我们在给命名模板定义名称的时候添加了 chart 名称这样的前缀，避免冲突。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:05:03

49. Helm Hooks

和 Kubernetes 里面的容器一样，Helm 也提供了 [Hook](#) 的机制，允许 chart 开发人员在 release 的生命周期中的某些节点来进行干预，比如我们可以利用 Hooks 来做下面的这些事情：

- 在加载任何其他 chart 之前，在安装过程中加载 ConfigMap 或 Secret
- 在安装新 chart 之前执行作业以备份数据库，然后在升级后执行第二个作业以恢复数据
- 在删除 release 之前运行作业，以便在删除 release 之前优雅地停止服务

值得注意的是 Hooks 和普通模板一样工作，但是它们具有特殊的注释，可以使 Helm 以不同的方式使用它们。

Hook 在资源清单中的 metadata 部分用 annotations 的方式进行声明：

```
apiVersion: ...
kind: ....
metadata:
  annotations:
    "helm.sh/hook": "pre-install"
# ...
```

接下来我们就来和大家介绍下 Helm Hooks 的一些基本使用方法。

Hooks

在 Helm 中定义了如下一些可供我们使用的 Hooks：

- 预安装 `pre-install`：在模板渲染后，kubernetes 创建任何资源之前执行
- 安装后 `post-install`：在所有 kubernetes 资源安装到集群后执行
- 预删除 `pre-delete`：在从 kubernetes 删除任何资源之前执行删除请求
- 删除后 `post-delete`：删除所有 release 的资源后执行
- 升级前 `pre-upgrade`：在模板渲染后，但在任何资源升级之前执行
- 升级后 `post-upgrade`：在所有资源升级后执行
- 预回滚 `pre-rollback`：在模板渲染后，在任何资源回滚之前执行
- 回滚后 `post-rollback`：在修改所有资源后执行回滚请求
- `crd-install`：在运行其他检查之前添加 CRD 资源，只能用于 chart 中其他的资源清单定义的 CRD 资源。

生命周期

Hooks 允许开发人员在 release 的生命周期中的一些关键节点执行一些钩子函数，我们正常安装一个 chart 包的时候的生命周期如下所示：

- 用户运行 `helm install foo`
- chart 被加载到服务端 Tiller Server 中

- 经过一些验证， Tiller Server 渲染 foo 模板
- Tiller 将产生的资源加载到 kubernetes 中去
- Tiller 将 release 名称和其他数据返回给 Helm 客户端
- Helm 客户端退出

如果开发人员在 install 的生命周期中定义了两个 hook: `pre-install` 和 `post-install`，那么我们安装一个 chart 包的生命周期就会多一些步骤了：

- 用户运行 `helm install foo`
- chart 被加载到服务端 Tiller Server 中
- 经过一些验证， Tiller Server 渲染 foo 模板
- Tiller 将 hook 资源加载到 kubernetes 中，准备执行 `pre-install` hook
- Tiller 会根据权重对 hook 进行排序（默认分配权重0，权重相同的 hook 按升序排序）
- Tiller 然后加载最低权重的 hook
- Tiller 等待，直到 hook 准备就绪
- Tiller 将产生的资源加载到 kubernetes 中
- Tiller 执行 `post-install` hook
- Tiller 等待，直到 hook 准备就绪
- Tiller 将 release 名称和其他数据返回给客户端
- Helm 客户端退出

等待 hook 准备就绪，这是一个阻塞的操作，如果 hook 中声明的是一个 Job 资源，那么 Tiller 将等待 Job 成功完成，如果失败，则发布失败，在这个期间，Helm 客户端是处于暂停状态的。

对于所有其他类型，只要 kubernetes 将资源标记为加载（添加或更新），资源就被视为就绪状态，当一个 hook 声明了很多资源时，这些资源是被串行执行的。

另外需要注意的是 hook 创建的资源不会作为 release 的一部分进行跟踪和管理，一旦 Tiller Server 验证了 hook 已经达到了就绪状态，它就不会去管它了。

所以，如果我们在 hook 中创建了资源，那么不能依赖 `helm delete` 去删除资源，因为 hook 创建的资源已经不受控制了，要销毁这些资源，需要在 `pre-delete` 或者 `post-delete` 这两个 hook 函数中去执行相关操作，或者将 `helm.sh/hook-delete-policy` 这个 annotation 添加到 hook 模板文件中。

写一个 hook

上面我们也说了 hook 和普通模板一样，也可以使用普通的模板函数和常用的一些对象，比如 `Values`、`Chart`、`Release` 等等，唯一和普通模板不太一样的地方就是在资源清单文件中的 `metadata` 部分会有一些特殊的注释 annotation。

例如，现在我们来创建一个 hook，在前面的示例 templates 目录中添加一个 `post-install-job.yaml` 的文件，表示安装后执行的一个 hook：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Release.Name }}-post-install-job
  labels:
```

```

release: {{ .Release.Name }}
chart: {{ .Chart.Name }}
version: {{ .Chart.Version }}
annotations:
# 注意，如果没有下面的这个注释的话，当前的这个Job就会被当成release的一部分
"helm.sh/hook": post-install
"helm.sh/hook-weight": "-5"
"helm.sh/hook-delete-policy": hook-succeeded
spec:
template:
metadata:
name: {{ .Release.Name }}
labels:
release: {{ .Release.Name }}
chart: {{ .Chart.Name }}
version: {{ .Chart.Version }}
spec:
restartPolicy: Never
containers:
- name: post-install-job
image: alpine
command: ["/bin/sleep", "{{ default "10" .Values.sleepTime }}"]

```

上面的 Job 资源中我们添加一个 annotations，要注意的是，如果我们没有添加下面这行注释的话，这个资源就会被当成是 release 的一部分资源：

```

annotations:
"helm.sh/hook": post-install

```

当然一个资源中我们也可以同时部署多个 hook，比如我们还可以添加一个 post-upgrade 的钩子：

```

annotations:
"helm.sh/hook": post-install,post-upgrade

```

另外值得注意的是我们为 hook 定义了一个权重，这有助于建立一个确定性的执行顺序，权重可以是正数也可以是负数，但是必须是字符串才行。

```

annotations:
"helm.sh/hook-weight": "-5"

```

最后还添加了一个删除 hook 资源的策略：

```

annotations:
"helm.sh/hook-delete-policy": hook-succeeded

```

删除资源的策略可供选择的注释值：

- hook-succeeded：表示 Tiller 在 hook 成功执行后删除 hook 资源
- hook-failed：表示如果 hook 在执行期间失败了，Tiller 应该删除 hook 资源
- before-hook-creation：表示在删除新的 hook 之前应该删除以前的 hook

当 helm 的 release 更新时，有可能 hook 资源已经存在于群集中。默认情况下，helm 会尝试创建资源，并抛出错误"… already exists"。

我们可以选择 "helm.sh/hook-delete-policy": "before-hook-creation"，取代 "helm.sh/hook-delete-policy": "hook-succeeded,hook-failed" 因为：

例如为了手动调试，将错误的 hook 作业资源保存在 kubernetes 中是很方便的。出于某种原因，可能有必要将成功的 hook 资源保留在 kubernetes 中。同时，在 helm release 升级之前进行手动资源删除是不可取的。 "helm.sh/hook-delete-policy": "before-hook-creation" 在 hook 中的注释，如果在新的 hook 启动前有一个 hook 的话，会使 Tiller 将以前的release 中的 hook 删除，而这个 hook 同时它可能正在被其他一个策略使用。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

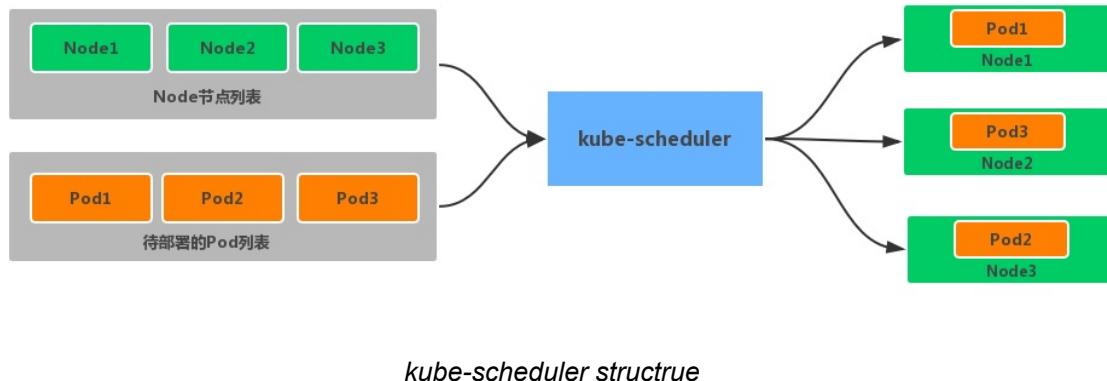
50. Kubernetes 调度器介绍

`kube-scheduler` 是 `kubernetes` 系统的核心组件之一，主要负责整个集群资源的调度功能，根据特定的调度算法和策略，将 Pod 调度到最优的工作节点上面去，从而更加合理、更加充分的利用集群的资源，这也是我们选择使用 `kubernetes` 一个非常重要的理由。如果一门新的技术不能帮助企业节约成本、提供效率，我相信是很难推进的。

调度流程

默认情况下，`kube-scheduler` 提供的默认调度器能够满足我们绝大多数的要求，我们前面和大家接触的示例也基本上用的默认的策略，都可以保证我们的 Pod 可以被分配到资源充足的节点上运行。但是在实际的线上项目中，可能我们自己会比 `kubernetes` 更加了解我们自己的应用，比如我们希望一个 Pod 只能运行在特定的几个节点上，或者这几个节点只能用来运行特定类型的应用，这就需要我们的调度器能够可控。

`kube-scheduler` 是 `kubernetes` 的调度器，它的主要作用就是根据特定的调度算法和调度策略将 Pod 调度到合适的 Node 节点上去，是一个独立的二进制程序，启动之后会一直监听 API Server，获取到 `PodSpec.nodeName` 为空的 Pod，对每个 Pod 都会创建一个 binding。



kube-scheduler structure

这个过程在我看来好像比较简单，但在实际的生产环境中，需要考虑的问题就有很多了：

- 如何保证全部的节点调度的公平性？要知道并不是说有节点资源配置都是一样的
- 如何保证每个节点都能被分配资源？
- 集群资源如何能够被高效利用？
- 集群资源如何才能被最大化使用？
- 如何保证 Pod 调度的性能和效率？
- 用户是否可以根据自己的实际需求定制自己的调度策略？

考虑到实际环境中的各种复杂情况，`kubernetes` 的调度器采用插件化的形式实现，可以方便用户进行定制或者二次开发，我们可以自定义一个调度器并以插件形式和 `kubernetes` 进行集成。

kubernetes 调度器的源码位于 `kubernetes/pkg/scheduler` 中，大体的代码目录结构如下所示：(不同的版本目录结构可能不太一样)

```
kubernetes/pkg/scheduler
-- scheduler.go          //调度相关的具体实现
|--- algorithm
|   |--- predicates    //节点筛选策略
|   |--- priorities     //节点打分策略
|--- algorithmprovider
|   |--- defaults       //定义默认的调度器
```

其中 `Scheduler` 创建和运行的核心程序，对应的代码在 `pkg/scheduler/scheduler.go`，如果要看 `kube-scheduler` 的入口程序，对应的代码在 `cmd/kube-scheduler/scheduler.go`。

调度主要分为以下几个部分：

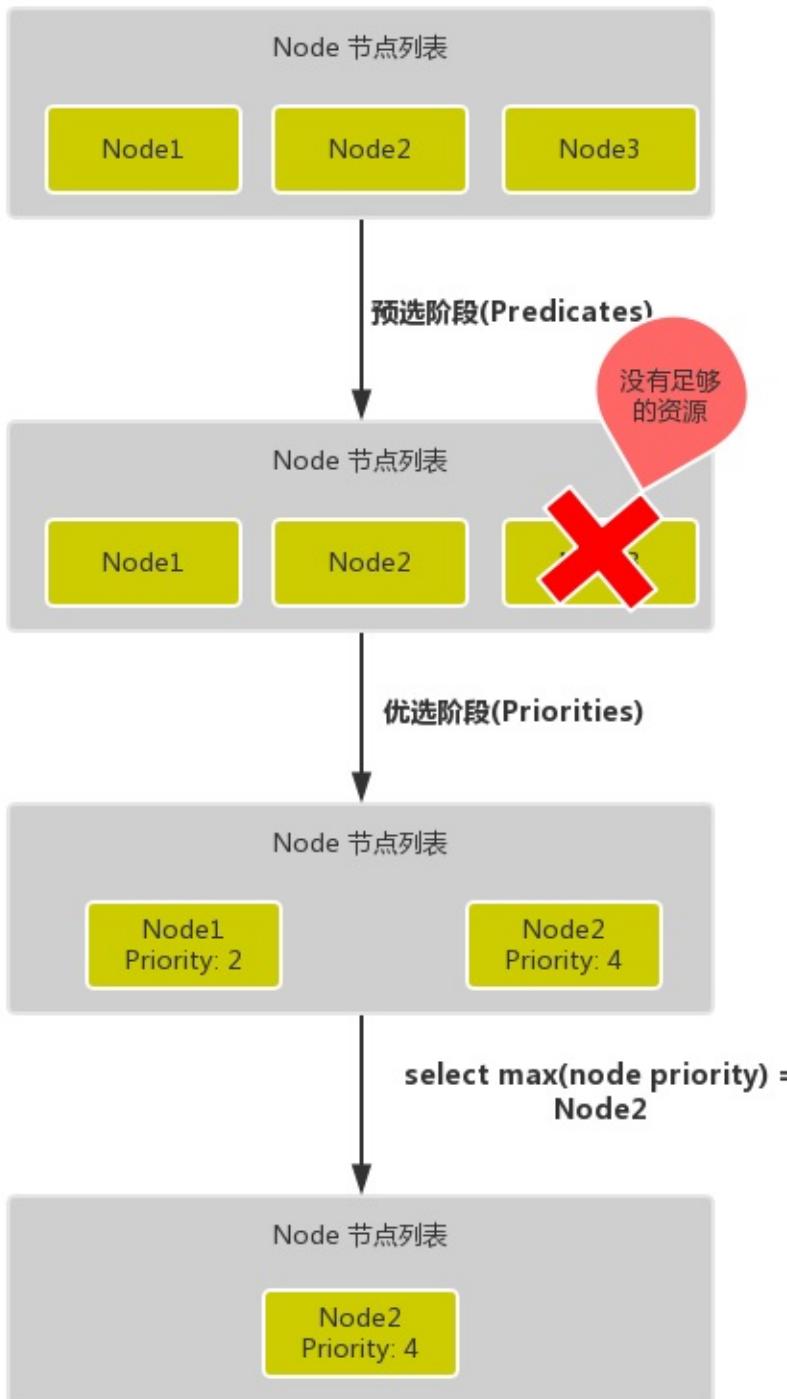
- 首先是预选过程，过滤掉不满足条件的节点，这个过程称为 `Predicates`
- 然后是优选过程，对通过的节点按照优先级排序，称之为 `Priorities`
- 最后从中选择优先级最高的节点，如果中间任何一步骤有错误，就直接返回错误

`Predicates` 阶段首先遍历全部节点，过滤掉不满足条件的节点，属于强制性规则，这一阶段输出的所有满足要求的 Node 将被记录并作为第二阶段的输入，如果所有的节点都不满足条件，那么 Pod 将会一直处于 Pending 状态，直到有节点满足条件，在这期间调度器会不断的重试。

所以我们在部署应用的时候，如果发现有 Pod 一直处于 Pending 状态，那么就是没有满足调度条件的节点，这个时候可以去检查下节点资源是否可用。

`Priorities` 阶段即再次对节点进行筛选，如果有多个节点都满足条件的话，那么系统会按照节点的优先级(priorites)大小对节点进行排序，最后选择优先级最高的节点来部署 Pod 应用。

下面是调度过程的简单示意图：



更详细的流程是这样的：

- 首先，客户端通过 API Server 的 REST API 或者 kubectl 工具创建 Pod 资源
- API Server 收到用户请求后，存储相关数据到 etcd 数据库中
- 调度器监听 API Server 查看为调度(bind)的 Pod 列表，循环遍历地为每个 Pod 尝试分配节点，这个分配过程就是我们上面提到的两个阶段：

- 预选阶段(Predicates)，过滤节点，调度器用一组规则过滤掉不符合要求的 Node 节点，比如 Pod 设置了资源的 request，那么可用资源比 Pod 需要的资源少的主机显然就会被过滤掉
- 优选阶段(Priorities)，为节点的优先级打分，将上一阶段过滤出来的 Node 列表进行打分，调度器会考虑一些整体的优化策略，比如把 Deployment 控制的多个 Pod 副本分布到不同的主机上，使用最低负载的主机等等策略
- 经过上面的阶段过滤后选择打分最高的 Node 节点和 Pod 进行 binding 操作，然后将结果存储到 etcd 中
- 最后被选择出来的 Node 节点对应的 kubelet 去执行创建 Pod 的相关操作

其中 Predicates 过滤有一系列的算法可以使用，我们这里简单列举几个：

- PodFitsResources：节点上剩余的资源是否大于 Pod 请求的资源
- PodFitsHost：如果 Pod 指定了 NodeName，检查节点名称是否和 NodeName 匹配
- PodFitsHostPorts：节点上已经使用的 port 是否和 Pod 申请的 port 冲突
- PodSelectorMatches：过滤掉和 Pod 指定的 label 不匹配的节点
- NoDiskConflict：已经 mount 的 volume 和 Pod 指定的 volume 不冲突，除非它们都是只读的
- CheckNodeDiskPressure：检查节点磁盘空间是否符合要求
- CheckNodeMemoryPressure：检查节点内存是否够用

除了这些过滤算法之外，还有一些其他的算法，更多更详细的我们可以查看源码文件：
k8s.io/kubernetes/pkg/scheduler/algorithm/predicates/predicates.go。

而 Priorities 优先级是由一系列键值对组成的，键是该优先级的名称，值是它的权重值，同样，我们这里给大家列举几个具有代表性的选项：

- LeastRequestedPriority：通过计算 CPU 和内存的使用率来决定权重，使用率越低权重越高，当然正常肯定也是资源是使用率越低权重越高，能给别的 Pod 运行的可能性就越大
- SelectorSpreadPriority：为了更好的高可用，对同属于一个 Deployment 或者 RC 下面的多个 Pod 副本，尽量调度到多个不同的节点上，当一个 Pod 被调度的时候，会先去查找该 Pod 对应的 controller，然后查看该 controller 下面的已存在的 Pod，运行 Pod 越少的节点权重越高
- ImageLocalityPriority：就是如果在某个节点上已经有要使用的镜像节点了，镜像总大小值越大，权重就越高
- NodeAffinityPriority：这个就是根据节点的亲和性来计算一个权重值，后面我们会详细讲解亲和性的使用方法

除了这些策略之外，还有很多其他的策略，同样我们可以查看源码文件：

k8s.io/kubernetes/pkg/scheduler/algorithm/priorities/ 了解更多信息。每一个优先级函数会返回一个0-10的分数，分数越高表示节点越优，同时每一个函数也会对应一个表示权重的值。最终主机的得分用以下公式计算得出：

```
finalScoreNode = (weight1 * priorityFunc1) + (weight2 * priorityFunc2) + ... + (weightn * priorityFuncn)
```

自定义调度

上面就是 kube-scheduler 默认调度的基本流程，除了使用默认的调度器之外，我们也可以自定义调度策略。

调度器扩展

`kube-scheduler` 在启动的时候可以通过 `--policy-config-file` 参数来指定调度策略文件，我们可以根据我们自己的需要来组装 `Predicates` 和 `Priority` 函数。选择不同的过滤函数和优先级函数、控制优先级函数的权重、调整过滤函数的顺序都会影响调度过程。

下面是官方的 Policy 文件示例：

```
{
  "kind": "Policy",
  "apiVersion": "v1",
  "predicates": [
    {"name": "PodFitsHostPorts"},
    {"name": "PodFitsResources"},
    {"name": "NoDiskConflict"},
    {"name": "NoVolumeZoneConflict"},
    {"name": "MatchNodeSelector"},
    {"name": "HostName"}
  ],
  "priorities": [
    {"name": "LeastRequestedPriority", "weight": 1},
    {"name": "BalancedResourceAllocation", "weight": 1},
    {"name": "ServiceSpreadingPriority", "weight": 1},
    {"name": "EqualPriority", "weight": 1}
  ]
}
```

多调度器

如果默认的调度器不满足要求，还可以部署自定义的调度器。并且，在整个集群中还可以同时运行多个调度器实例，通过 `podSpec.schedulerName` 来选择使用哪一个调度器（默认使用内置的调度器）。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  schedulerName: my-scheduler # 选择使用自定义调度器 my-scheduler
  containers:
    - name: nginx
      image: nginx:1.10
```

要开发我们自己的调度器也是比较容易的，比如我们这里的 `my-scheduler`:

- 首先需要通过指定的 API 获取节点和 Pod
- 然后选择 `phase=Pending` 和 `schedulerName=my-scheduler` 的 pod
- 计算每个 Pod 需要放置的位置之后，调度程序将创建一个 `Binding` 对象

- 然后根据我们自定义的调度器的算法计算出最适合的目标节点

优先级调度

与前面所讲的调度优选策略中的优先级 (Priorities) 不同，前面所讲的优先级指的是节点优先级，而我们这里所说的优先级 pod priority 指的是 Pod 的优先级，高优先级的 Pod 会优先被调度，或者在资源不足时牺牲低优先级的 Pod，以便于重要的 Pod 能够得到资源部署。

要定义 Pod 优先级，就需要先定义 PriorityClass 对象，该对象没有 Namespace 的限制：

```
apiVersion: v1
kind: PriorityClass
metadata:
  name: high-priority
  value: 1000000
  globalDefault: false
  description: "This priority class should be used for XYZ service pods only."
```

其中：

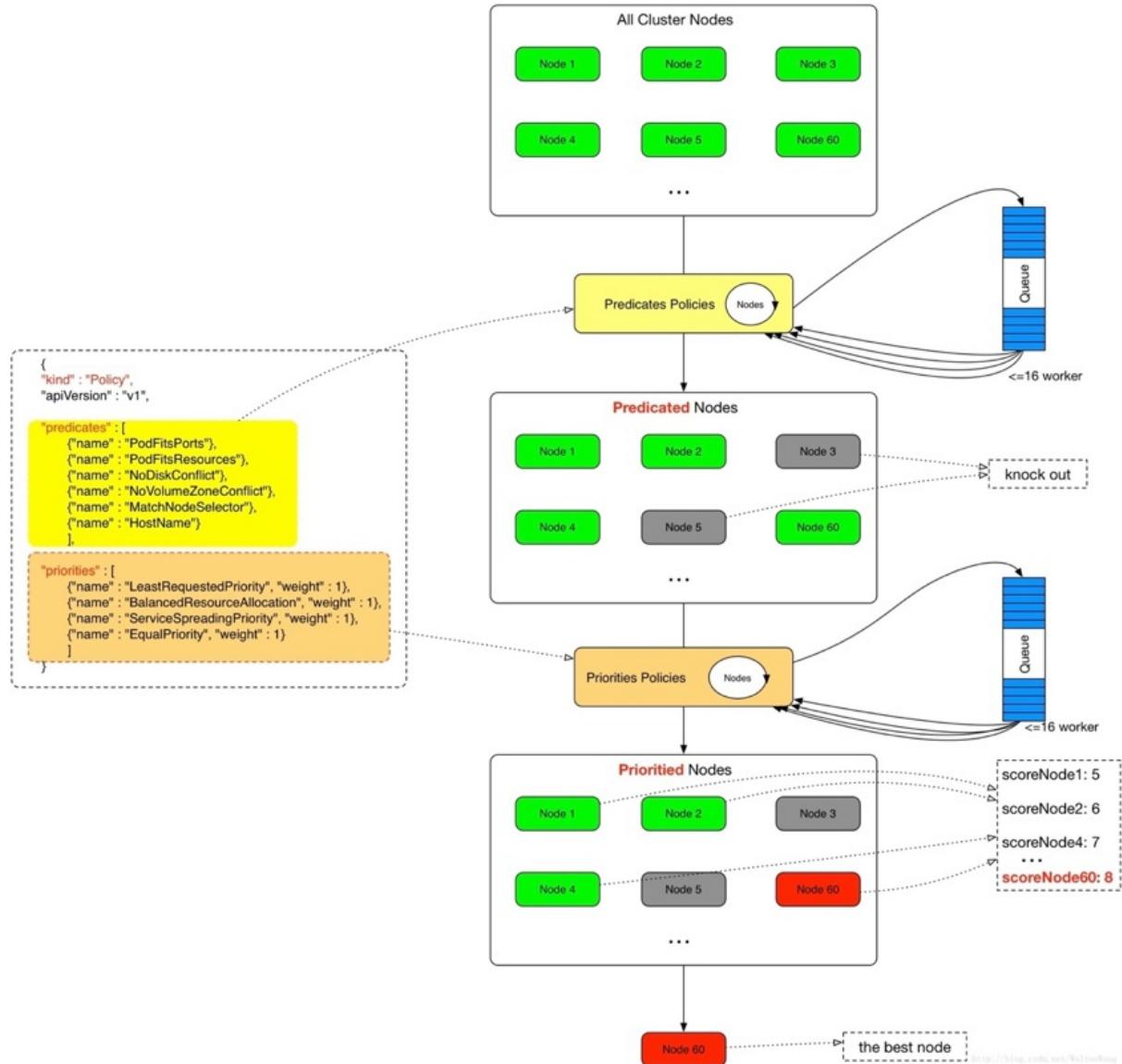
- `value` 为 32 位整数的优先级，该值越大，优先级越高
- `globalDefault` 用于未配置 `PriorityClassName` 的 Pod，整个集群中应该只有一个 `PriorityClass` 将其设置为 `true`

然后通过在 Pod 的 `spec.priorityClassName` 中指定已定义的 `PriorityClass` 名称即可：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
    priorityClassName: high-priority
```

另外一个值得注意的是当节点没有足够的资源供调度器调度 Pod，导致 Pod 处于 pending 时，抢占 (preemption) 逻辑就会被触发。`Preemption` 会尝试从一个节点删除低优先级的 Pod，从而释放资源使高优先级的 Pod 得到节点资源进行部署。

现在我们通过下面的图再去回顾下 kubernetes 的调度过程是不是就清晰很多了：



下节课我们再和大家讲解关于 Pod 调度的一些具体使用方法。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:05:17

51. Kubernetes 亲和性调度

一般情况下我们部署的 Pod 是通过集群的自动调度策略来选择节点的，默认情况下调度器考虑的是资源足够，并且负载尽量平均，但是有的时候我们需要能够更加细粒度的去控制 Pod 的调度，比如我们内部的一些服务 gitlab 之类的也是跑在 Kubernetes 集群上的，我们就不希望对外的一些服务和内部的服务跑在同一个节点上了，害怕内部服务对外部的服务产生影响；但是有的时候我们的服务之间交流比较频繁，又希望能够将这两个服务的 Pod 调度到同一个的节点上。这就需要用到 Kubernetes 里面的一个概念：亲和性和反亲和性。

亲和性有分成节点亲和性(`nodeAffinity`)和 Pod 亲和性(`podAffinity`)。

nodeSelector

在了解亲和性之前，我们先来了解一个非常常用的调度方式：`nodeSelector`。我们知道 `label` 是 `kubernetes` 中一个非常重要的概念，用户可以非常灵活的利用 `label` 来管理集群中的资源，比如最常见的一个就是 `service` 通过匹配 `label` 去匹配 Pod 资源，而 Pod 的调度也可以根据节点的 `label` 来进行调度。

我们可以通过下面的命令查看我们的 node 的 `label`：

```
$ kubectl get nodes --show-labels
NAME      STATUS    ROLES     AGE       VERSION   LABELS
master    Ready     master    147d      v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=master,node-role.kubernetes.io/master=
node02    Ready     <none>    67d       v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,course=k8s,kubernetes.io/hostname=node02
node03    Ready     <none>    127d      v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,jnlp=haimaxy,kubernetes.io/hostname=node03
```

现在我们先给节点node02增加一个 `com=youdianzhishi` 的标签，命令如下：

```
$ kubectl label nodes node02 com=youdianzhishi
node "node02" labeled
```

我们可以通过上面的 `--show-labels` 参数可以查看上述标签是否生效。当 node 被打上了相关标签后，在调度的时候就可以使用这些标签了，只需要在 Pod 的 `spec` 字段中添加 `nodeSelector` 字段，里面是我们需要被调度的节点的 `label` 即可。比如，下面的 Pod 我们要强制调度到 node02 这个节点上去，我们就可以使用 `nodeSelector` 来表示了：(`node-selector-demo.yaml`)

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: busybox-pod
    name: test-busybox
spec:
  containers:
```

```

- command:
  - sleep
  - "3600"
  image: busybox
  imagePullPolicy: Always
  name: test-busybox
  nodeSelector:
    com: youdianzhishi

```

然后我们可以通过 `describe` 命令查看调度结果：

```

$ kubectl create -f node-selector-demo.yaml
pod "test-busybox" created
$ kubectl describe pod test-busybox
Name:           test-busybox
Namespace:      default
Node:          node02/10.151.30.63
...
QoS Class:     BestEffort
Node-Selectors: com=youdianzhishi
Tolerations:   node.kubernetes.io/not-ready:NoExecute for 300s
               node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type      Reason          Age    From            Message
  ----      ----          ----   ----            -----
  Normal   SuccessfulMountVolume 55s   kubelet, node02  MountVolume.SetUp succeeded for volume "default-token-n9w2d"
  Normal   Scheduled         54s   default-scheduler  Successfully assigned test-busybox to node02
  Normal   Pulling          54s   kubelet, node02  pulling image "busybox"
  Normal   Pulled           40s   kubelet, node02  Successfully pulled image "busybox"
  Normal   Created          40s   kubelet, node02  Created container
  Normal   Started          40s   kubelet, node02  Started container

```

我们可以看到 `Events` 下面的信息，我们的 Pod 通过默认的 `default-scheduler` 调度器被绑定到了 `node02` 节点。不过需要注意的是 `nodeSelector` 属于强制性的，如果我们的目标节点没有可用的资源，我们的 Pod 就会一直处于 `Pending` 状态，这就是 `nodeSelector` 的用法。

通过上面的例子我们可以感受到 `nodeSelector` 的方式比较直观，但是还不够灵活，控制粒度偏大，接下来我们再和大家了解下更加灵活的方式：节点亲和性(`nodeAffinity`)。

亲和性和反亲和性调度

上节课我们了解了 `kubernetes` 调度器的一个调度流程，我们知道默认的调度器在使用的时候，经过了 `predicates` 和 `priorities` 两个阶段，但是在实际的生产环境中，往往我们需要根据自己的一些实际需求来控制 pod 的调度，这就需要用到 `nodeAffinity`(节点亲和性)、`podAffinity`(pod 亲和性) 以及 `podAntiAffinity`(pod 反亲和性)。

亲和性调度可以分成软策略和硬策略两种方式：

- 软策略 就是如果你没有满足调度要求的节点的话，pod 就会忽略这条规则，继续完成调度过程，说白了就是满足条件最好了，没有的话也无所谓了的策略

- 硬策略 就比较强硬了，如果没有满足条件的节点的话，就不断重试直到满足条件为止，简单说就是你必须满足我的要求，不然我就不干的策略。

对于亲和性和反亲和性都有这两种规则可以设置：

`preferredDuringSchedulingIgnoredDuringExecution` 和 `requiredDuringSchedulingIgnoredDuringExecution`，前面的就是软策略，后面的就是硬策略。

这命名不觉得有点反人类吗？有点无语.....

nodeAffinity

节点亲和性主要是用来控制 pod 要部署在哪些主机上，以及不能部署在哪些主机上的。它可以进行一些简单的逻辑组合了，不只是简单的相等匹配。

比如现在我们用一个 Deployment 来管理3个 pod 副本，现在我们来控制下这些 pod 的调度，如下例子：（node-affinity-demo.yaml）

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: affinity
  labels:
    app: affinity
spec:
  replicas: 3
  revisionHistoryLimit: 15
  template:
    metadata:
      labels:
        app: affinity
        role: test
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
          name: nginxweb
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution: # 硬策略
            nodeSelectorTerms:
            - matchExpressions:
              - key: kubernetes.io/hostname
                operator: NotIn
                values:
                - node03
          preferredDuringSchedulingIgnoredDuringExecution: # 软策略
          - weight: 1
            preference:
              matchExpressions:
              - key: com
                operator: In
                values:
                - youdianzhishi
```

上面这个 pod 首先是要求不能运行在 node03 这个节点上，如果有某个节点满足 `com=youdianzhishi` 的话就优先调度到这个节点上。

下面是我们测试的节点列表信息：

```
$ kubectl get nodes --show-labels
NAME      STATUS    ROLES     AGE       VERSION   LABELS
master    Ready     master    154d      v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=master,node-role.kubernetes.io/master=
node02    Ready     <none>   74d       v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,com=youdianzhishi,course=k8s,kubernetes.io/hostname=node02
node03    Ready     <none>   134d      v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,jnlp=haimaxy,kubernetes.io/hostname=node03
```

可以看到 node02 节点有 `com=youdianzhishi` 这样的 label，按要求会优先调度到这个节点来的，现在我们来创建这个 pod，然后使用 `descirbe` 命令查看具体的调度情况是否满足我们的要求。

```
$ kubectl create -f node-affinity-demo.yaml
deployment.apps "affinity" created
$ kubectl get pods -l app=affinity -o wide
NAME            READY   STATUS    RESTARTS   AGE   IP          NODE
affinity-7b4c946854-5gf1n  1/1    Running   0          47s  10.244.4.214  node02
affinity-7b4c946854-18b47  1/1    Running   0          47s  10.244.4.215  node02
affinity-7b4c946854-r86p5  1/1    Running   0          47s  10.244.4.213  node02
```

从结果可以看出 pod 都被部署到了 node02，其他节点上没有部署 pod，这里的匹配逻辑是 label 的值在某个列表中，现在 Kubernetes 提供的操作符有下面的几种：

- In: label 的值在某个列表中
- NotIn: label 的值不在某个列表中
- Gt: label 的值大于某个值
- Lt: label 的值小于某个值
- Exists: 某个 label 存在
- DoesNotExist: 某个 label 不存在

如果 `nodeSelectorTerms` 下面有多个选项的话，满足任何一个条件就可以了；如果 `matchExpressions` 有多个选项的话，则必须同时满足这些条件才能正常调度 POD。

podAffinity

pod 亲和性主要解决 pod 可以和哪些 pod 部署在同一个拓扑域中的问题（其中拓扑域用主机标签实现，可以是单个主机，也可以是多个主机组成的 cluster、zone 等等），而 pod 反亲和性主要是解决 pod 不能和哪些 pod 部署在同一个拓扑域中的问题，它们都是处理的 pod 与 pod 之间的关系，比如一个 pod 在一个节点上了，那么我这个也得在这个节点，或者你这个 pod 在节点上了，那么我就不想和你待在同一个节点上。

由于我们这里只有一个集群，并没有区域或者机房的概念，所以我们这里直接使用主机名来作为拓扑域，把 pod 创建在同一个主机上面。

```
$ kubectl get nodes --show-labels
NAME      STATUS    ROLES     AGE      VERSION   LABELS
master    Ready     master    154d    v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=master,node-role.kubernetes.io/master=
node02    Ready     <none>    74d     v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,com=youdianzhishi,course=k8s,kubernetes.io/hostname=node02
node03    Ready     <none>    134d    v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,jnlp=haimaxy,kubernetes.io/hostname=node03
```

同样，还是针对上面的资源对象，我们来测试下 pod 的亲和性：(pod-affinity-demo.yaml)

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: affinity
  labels:
    app: affinity
spec:
  replicas: 3
  revisionHistoryLimit: 15
  template:
    metadata:
      labels:
        app: affinity
        role: test
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
              name: nginxweb
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution: # 硬策略
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - busybox-pod
          topologyKey: kubernetes.io/hostname
```

上面这个例子中的 pod 需要调度到某个指定的主机上，至少有一个节点上运行了这样的 pod：这个 pod 有一个 app=busybox-pod 的 label。

我们查看有标签 app=busybox-pod 的 pod 列表：

```
$ kubectl get pods -o wide -l app=busybox-pod
NAME      READY    STATUS    RESTARTS   AGE      IP           NODE
test-busybox  1/1     Running   164       7d      10.244.4.205  node02
```

我们看到这个 pod 运行在了 node02 的节点上面，所以按照上面的亲和性来说，上面我们部署的3个 pod 副本也应该运行在 node02 节点上：

```
$ kubectl get pods -o wide -l app=affinity
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE
affinity-564f9d7db9-lzzvq  1/1    Running   0          3m     10.244.4.216 node02
affinity-564f9d7db9-p79cq  1/1    Running   0          3m     10.244.4.217 node02
affinity-564f9d7db9-spfzs  1/1    Running   0          3m     10.244.4.218 node02
```

如果我们把上面的 test-busybox 和 affinity 这个 Deployment 都删除，然后重新创建 affinity 这个资源，看看能不能正常调度呢：

```
$ kubectl delete -f node-selector-demo.yaml
pod "test-busybox" deleted
$ kubectl delete -f pod-affinity-demo.yaml
deployment.apps "affinity" deleted
$ kubectl create -f pod-affinity-demo.yaml
deployment.apps "affinity" created
$ kubectl get pods -o wide -l app=affinity
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE
affinity-564f9d7db9-fbc8w  0/1    Pending   0          2m     <none>      <none>
affinity-564f9d7db9-n8gcf  0/1    Pending   0          2m     <none>      <none>
affinity-564f9d7db9-qc7x6  0/1    Pending   0          2m     <none>      <none>
```

我们可以看到处于 Pending 状态了，这是因为现在没有一个节点上面拥有 busybox-pod 这个 label 的 pod，而上面我们的调度使用的是硬策略，所以就没办法进行调度了，大家可以去尝试下重新将 test-busybox 这个 pod 调度到 node03 这个节点上，看看上面的 affinity 的3个副本会不会也被调度到 node03 这个节点上去？

我们这个地方使用的是 `kubernetes.io/hostname` 这个拓扑域，意思就是我们当前调度的 pod 要和目标的 pod 处于同一个主机上面，因为要处于同一个拓扑域下面，为了说明这个问题，我们把拓扑域改成 `beta.kubernetes.io/os`，同样的我们当前调度的 pod 要和目标的 pod 处于同一个拓扑域中，目标的 pod 是不是拥有 `beta.kubernetes.io/os=linux` 的标签，而我们这里3个节点都有这样的标签，这也就意味着我们3个节点都在同一个拓扑域中，所以我们这里的 pod 可能会被调度到任何一个节点：

```
$ kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE
affinity-7d86749984-g1khz  1/1    Running   0          3m     10.24
4.2.16       node03
affinity-7d86749984-h4fb9  1/1    Running   0          3m     10.24
4.4.219       node02
affinity-7d86749984-tj7k2  1/1    Running   0          3m     10.24
4.2.14       node03
```

podAntiAffinity

这就是 pod 亲和性的用法，而 pod 反亲和性则是反着来的，比如一个节点上运行了某个 pod，那么我们的 pod 则希望被调度到其他节点上去，同样我们把上面的 podAffinity 直接改成 podAntiAffinity，(`pod-antiaffinity-demo.yaml`)

```
apiVersion: apps/v1beta1
kind: Deployment
```

```

metadata:
  name: affinity
  labels:
    app: affinity
spec:
  replicas: 3
  revisionHistoryLimit: 15
  template:
    metadata:
      labels:
        app: affinity
        role: test
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
              name: nginxweb
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution: # 硬策略
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - busybox-pod
          topologyKey: kubernetes.io/hostname

```

这里的意思就是如果一个节点上面有一个 `app=busybox-pod` 这样的 pod 的话，那么我们的 pod 就别调度到这个节点上面来，上面我们把 `app=busybox-pod` 这个 pod 固定到了 node03 这个节点上面来，所以正常来说我们这里的 pod 不会出现在 node03 节点上：

```

$ kubectl create -f pod-antiaffinity-demo.yaml
deployment.apps "affinity" created
$ kubectl get pods -o wide
NAME                               READY   STATUS    RESTARTS   AGE     IP
NODE
affinity-bcbd8854f-br8z8           1/1     Running   0          5s      10.24
4.4.222   node02
affinity-bcbd8854f-cdffh          1/1     Running   0          5s      10.24
4.4.223   node02
affinity-bcbd8854f-htb52          1/1     Running   0          5s      10.24
4.4.224   node02
test-busybox                        1/1     Running   0          23m    10.24
4.2.10    node03

```

这就是 pod 反亲和性的用法。

污点 (taints) 与容忍 (tolerations)

对于 nodeAffinity 无论是硬策略还是软策略方式，都是调度 pod 到预期节点上，而 Taints 恰好与之相反，如果一个节点标记为 Taints，除非 pod 也被标识为可以容忍污点节点，否则该 Taints 节点不会被调度 pod。

比如用户希望把 Master 节点保留给 Kubernetes 系统组件使用，或者把一组具有特殊资源预留给某些 pod，则污点就很有用了，pod 不会再被调度到 taint 标记过的节点。我们使用 kubeadm 搭建的集群默认就给 master 节点添加了一个污点标记，所以我们看到我们平时的 pod 都没有被调度到 master 上去：

```
$ kubectl describe node master
Name:           master
Roles:          master
Labels:         beta.kubernetes.io/arch=amd64
                beta.kubernetes.io/os=linux
                kubernetes.io/hostname=master
                node-role.kubernetes.io/master=
...
Taints:        node-role.kubernetes.io/master:NoSchedule
Unschedulable: false
...
```

我们可以使用上面的命令查看 master 节点的信息，其中有一条关于 Taints 的信息：node-role.kubernetes.io/master:NoSchedule，就表示给 master 节点打了一个污点的标记，其中影响的参数是 NoSchedule，表示 pod 不会被调度到标记为 taints 的节点，除了 NoSchedule 外，还有另外两个选项：

- PreferNoSchedule：NoSchedule 的软策略版本，表示尽量不调度到污点节点上去
- NoExecute：该选项意味着一旦 Taint 生效，如该节点内正在运行的 pod 没有对应 Tolerate 设置，会直接被逐出

污点 taint 标记节点的命令如下：

```
$ kubectl taint nodes node02 test=node02:NoSchedule
node "node02" tainted
```

上面的命名将 node02 节点标记为了污点，影响策略是 NoSchedule，只会影响新的 pod 调度，如果仍然希望某个 pod 调度到 taint 节点上，则必须在 Spec 中做出 Toleration 定义，才能调度到该节点，比如现在我们想要将一个 pod 调度到 master 节点：(taint-demo.yaml)

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: taint
  labels:
    app: taint
spec:
  replicas: 3
  revisionHistoryLimit: 10
  template:
    metadata:
      labels:
```

```

    app: taint
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - name: http
      containerPort: 80
  tolerations:
  - key: "node-role.kubernetes.io/master"
    operator: "Exists"
    effect: "NoSchedule"

```

由于 master 节点被标记为了污点节点，所以我们这里要想 pod 能够调度到 master 节点去，就需要增加容忍的声明：

```

tolerations:
- key: "node-role.kubernetes.io/master"
  operator: "Exists"
  effect: "NoSchedule"

```

然后创建上面的资源，查看结果：

```

$ kubectl create -f taint-demo.yaml
deployment.apps "taint" created
$ kubectl get pods -o wide
NAME                               READY   STATUS    RESTARTS   AGE
taint-845d8bb4fb-57mhm            1/1     Running   0          1m
  10.244.4.247   node02
taint-845d8bb4fb-bbtmp            1/1     Running   0          1m
  10.244.0.33    master
taint-845d8bb4fb-zb78x            1/1     Running   0          1m
  10.244.4.246   node02

```

我们可以看到有一个 pod 副本被调度到了 master 节点，这就是容忍的使用方法。

对于 tolerations 属性的写法，其中的 key、value、effect 与 Node 的 Taint 设置需保持一致，还有以下几点说明：

- 1. 如果 operator 的值是 Exists，则 value 属性可省略
- 1. 如果 operator 的值是 Equal，则表示其 key 与 value 之间的关系是 equal(等于)
- 1. 如果不指定 operator 属性，则默认值为 Equal

另外，还有两个特殊值：

- 1. 空的 key 如果再配合 Exists 就能匹配所有的 key 与 value，也是能容忍所有 node 的所有

Taints

- 1. 空的 effect 匹配所有的 effect

最后，如果我们要取消节点的污点标记，可以使用下面的命令：

```
$ kubectl taint nodes node02 test-node "node02" untainted
```

这就是污点和容忍的使用方法。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



*k8s技术圈*二维码

52. 在 Kubernets 中手动安装 Prometheus

从今天开始我们就和大家一起来学习 Kubernetes 中监控系统的搭建，我们知道监控是保证系统运行必不可少的功能，特别是对于 Kubernetes 这种比较庞大的系统来说，监控报警更是不可或缺，我们需要时刻了解系统的各种运行指标，也需要时刻了解我们的 Pod 的各种指标，更需要在出现问题的时候有报警信息通知到我们。

在早期的版本中 Kubernetes 提供了 heapster、influxDB、grafana 的组合来监控系统，所以我们可以看到 Dashboard 中看到 heapster 提供的一些图表信息，在后续的版本中会陆续移除掉 heapster，现在更加流行的监控工具是 [prometheus](#)，prometheus 是 Google 内部监控报警系统的开源版本，是 Google SRE 思想在其内部不断完善的产物，它的存在是为了更快和高效的发现问题，快速的接入速度，简单灵活的配置都很好的解决了这一切，而且是已经毕业的 CNCF 项目。

这里推荐一本书了解 Goolge 运维的秘密：《SRE: Google运维解密》

简介

Prometheus 最初是 SoundCloud 构建的开源系统监控和报警工具，是一个独立的开源项目，于 2016 年加入了 CNCF 基金会，作为继 Kubernetes 之后的第二个托管项目。

特征

Prometheus 相比于其他传统监控工具主要有以下几个特点：

- 具有由 metric 名称和键/值对标识的时间序列数据的多维数据模型
- 有一个灵活的查询语言
- 不依赖分布式存储，只和本地磁盘有关
- 通过 HTTP 的服务拉取时间序列数据
- 也支持推送的方式来添加时间序列数据
- 还支持通过服务发现或静态配置发现目标
- 多种图形和仪表板支持

组件

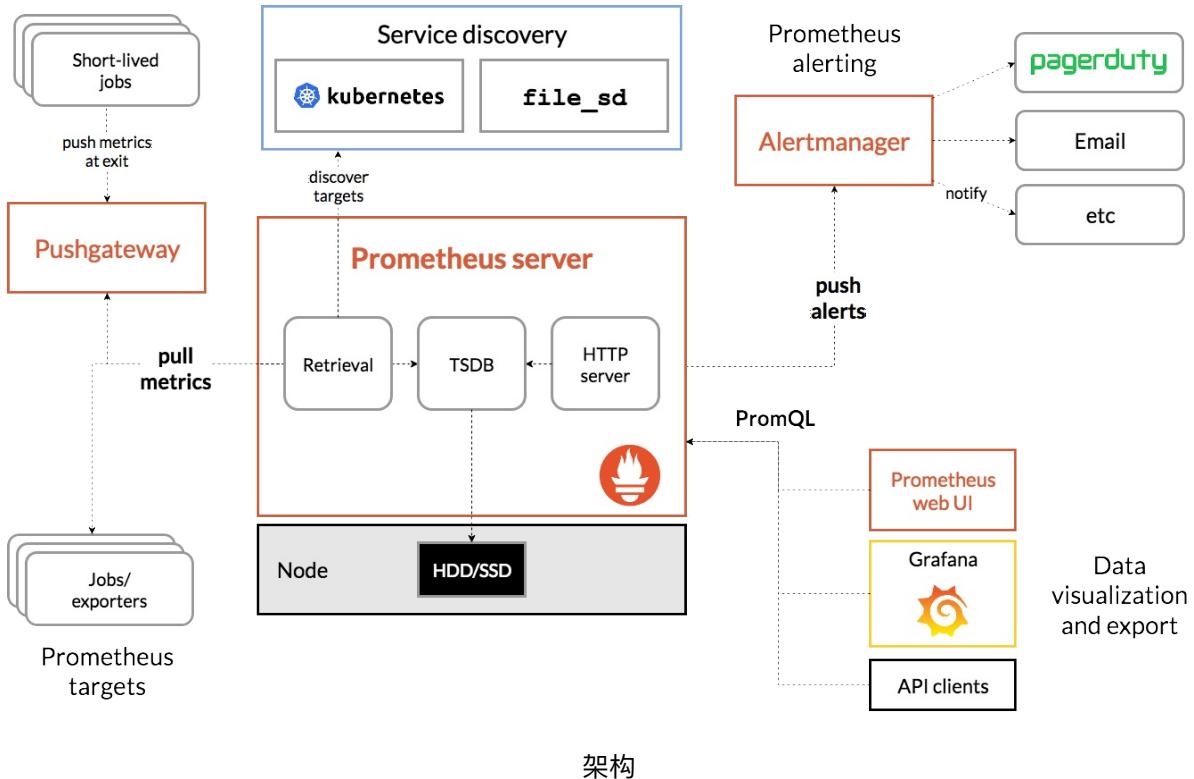
Prometheus 由多个组件组成，但是其中许多组件是可选的：

- Prometheus Server：用于抓取指标、存储时间序列数据
- exporter：暴露指标让任务来抓
- pushgateway：push 的方式将指标数据推送到该网关
- alertmanager：处理报警的报警组件
- adhoc：用于数据查询

大多数 Prometheus 组件都是用 Go 编写的，因此很容易构建和部署为静态的二进制文件。

架构

下图是 Prometheus 官方提供的架构及其一些相关的生态系统组件：



整体流程比较简单，Prometheus 直接接收或者通过中间的 Pushgateway 网关被动获取指标数据，在本地存储所有的获取的指标数据，并对这些数据进行一些规则整理，用来生成一些聚合数据或者报警信息，Grafana 或者其他工具用来可视化这些数据。

安装

由于 Prometheus 是 Golang 编写的程序，所以要安装的话也非常简单，只需要将二进制文件下载下来直接执行即可，前往地址：<https://prometheus.io/download> 下载我们对应的版本即可。

Prometheus 是通过一个 YAML 配置文件来进行启动的，如果我们使用二进制的方式来启动的话，可以使用下面的命令：

```
$ ./prometheus --config.file=prometheus.yml
```

其中 prometheus.yml 文件的基本配置如下：

```
global:
  scrape_interval:      15s
  evaluation_interval: 15s

rule_files:
  # - "first.rules"
  # - "second.rules"
```

```

scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['localhost:9090']

```

上面这个配置文件中包含了3个模块：global、rule_files 和 scrape_configs。

其中 global 模块控制 Prometheus Server 的全局配置：

- scrape_interval：表示 prometheus 抓取指标数据的频率，默认是15s，我们可以覆盖这个值
- evaluation_interval：用来控制评估规则的频率，prometheus 使用规则产生新的时间序列数据或者产生警报

rule_files 模块制定了规则所在的位置，prometheus 可以根据这个配置加载规则，用于生成新的时间序列数据或者报警信息，当前我们没有配置任何规则。

scrape_configs 用于控制 prometheus 监控哪些资源。由于 prometheus 通过 HTTP 的方式来暴露的它本身的监控数据，prometheus 也能够监控本身的健康情况。在默认的配置里有一个单独的 job，叫做prometheus，它采集 prometheus 服务本身的时间序列数据。这个 job 包含了一个单独的、静态配置的目标：监听 localhost 上的9090端口。prometheus 默认会通过目标的 /metrics 路径采集 metrics。所以，默认的 job 通过 URL: `http://localhost:9090/metrics` 采集 metrics。收集到的时间序列包含 prometheus 服务本身的状态和性能。如果我们还有其他的资源需要监控的话，直接配置在该模块下面就可以了。

由于我们这里是要跑在 Kubernetes 系统中，所以我们直接用 Docker 镜像的方式运行即可。

为了方便管理，我们将所有的资源对象都安装在 `kube-ops` 的 namespace 下面，没有的话需要提前安装。

为了能够方便的管理配置文件，我们这里将 `prometheus.yml` 文件用 ConfigMap 的形式进行管理：
(`prometheus-cm.yaml`)

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: kube-ops
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
      scrape_timeout: 15s
    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
          - targets: ['localhost:9090']

```

我们这里暂时只配置了对 prometheus 的监控，然后创建该资源对象：

```
$ kubectl create -f prometheus-cm.yaml
configmap "prometheus-config" created
```

配置文件创建完成了，以后如果我们有新的资源需要被监控，我们只需要将上面的 ConfigMap 对象更新即可。现在我们来创建 prometheus 的 Pod 资源：(prometheus-deploy.yaml)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: prometheus
  namespace: kube-ops
  labels:
    app: prometheus
spec:
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      serviceAccountName: prometheus
      containers:
        - image: prom/prometheus:v2.4.3
          name: prometheus
          command:
            - "/bin/prometheus"
          args:
            - "--config.file=/etc/prometheus/prometheus.yml"
            - "--storage.tsdb.path=/prometheus"
            - "--storage.tsdb.retention=24h"
            - "--web.enable admin-api" # 控制对admin HTTP API的访问，其中包括删除时间序列等功能
            - "--web.enable lifecycle" # 支持热更新，直接执行localhost:9090/-/reload立即生效
          ports:
            - containerPort: 9090
              protocol: TCP
              name: http
          volumeMounts:
            - mountPath: "/prometheus"
              subPath: prometheus
              name: data
            - mountPath: "/etc/prometheus"
              name: config-volume
          resources:
            requests:
              cpu: 100m
              memory: 512Mi
            limits:
              cpu: 100m
              memory: 512Mi
      securityContext:
        runAsUser: 0
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: prometheus
        - configMap:
            name: prometheus-config
            name: config-volume
```

我们在启动程序的时候，除了指定了 `prometheus.yml` 文件之外，还通过参数 `storage.tsdb.path` 指定了 TSDB 数据的存储路径、通过 `storage.tsdb.retention` 设置了保留多长时间的数据，还有下面的 `web.enable-admin-api` 参数可以用来开启对 admin api 的访问权限，参数 `web.enable-lifecycle` 非常重要，用来开启支持热更新的，有了这个参数之后，`prometheus.yml` 配置文件只要更新了，通过执行 `localhost:9090/-/reload` 就会立即生效，所以一定要加上这个参数。

我们这里将 `prometheus.yml` 文件对应的 ConfigMap 对象通过 volume 的形式挂载进了 Pod，这样 ConfigMap 更新后，对应的 Pod 里面的文件也会热更新的，然后我们再执行上面的 `reload` 请求，Prometheus 配置就生效了，除此之外，为了将时间序列数据进行持久化，我们将数据目录和一个 pvc 对象进行了绑定，所以我们需要提前创建好这个 pvc 对象：(`prometheus-volume.yaml`)

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: prometheus
spec:
  capacity:
    storage: 10Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    server: 10.151.30.57
    path: /data/k8s

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: prometheus
  namespace: kube-ops
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi

```

我们这里简单的通过 NFS 作为存储后端创建一个 pv、pvc 对象：

```
$ kubectl create -f prometheus-volume.yaml
```

除了上面的注意事项外，我们这里还需要配置 rbac 认证，因为我们需要在 `prometheus` 中去访问 Kubernetes 的相关信息，所以我们这里管理了一个名为 `prometheus` 的 serviceAccount 对象：(`prometheus-rbac.yaml`)

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: kube-ops
---
apiVersion: rbac.authorization.k8s.io/v1

```

```

kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  - services
  - endpoints
  - pods
  - nodes/proxy
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - configmaps
  - nodes/metrics
  verbs:
  - get
- nonResourceURLs:
  - /metrics
  verbs:
  - get
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: kube-ops

```

由于我们要获取的资源信息，在每一个 namespace 下面都有可能存在，所以我们这里使用的是 ClusterRole 的资源对象，值得一提的是我们这里的权限规则声明中有一个 `nonResourceURLs` 的属性，是用来对非资源型 metrics 进行操作的权限声明，这个在以前我们很少遇到过，然后直接创建上面的资源对象即可：

```

$ kubectl create -f prometheus-rbac.yaml
serviceaccount "prometheus" created
clusterrole.rbac.authorization.k8s.io "prometheus" created
clusterrolebinding.rbac.authorization.k8s.io "prometheus" created

```

还有一个要注意的地方是我们这里必须要添加一个 `securityContext` 的属性，将其中的 `runAsUser` 设置为0，这是因为现在的 prometheus 运行过程中使用的用户是 `nobody`，否则会出现下面的 `permission denied` 之类的权限错误：

```
level=error ts=2018-10-22T14:34:58.632016274Z caller=main.go:617 err="opening storage fail
ed: lock DB directory: open /data/lock: permission denied"
```

现在我们就可以添加 promethues 的资源对象了：

```
$ kubectl create -f prometheus-deploy.yaml
deployment.extensions "prometheus" created
$ kubectl get pods -n kube-ops
NAME           READY   STATUS    RESTARTS   AGE
prometheus-6dd775cbff-zb691   1/1     Running   0          20m
$ kubectl logs -f prometheus-6dd775cbff-zb691 -n kube-ops
...
level=info ts=2018-10-22T14:44:40.535385503Z caller=main.go:523 msg="Server is ready to receive web requests."
```

Pod 创建成功后，为了能够在外部访问到 prometheus 的 webui 服务，我们还需要创建一个 Service 对象：(prometheus-svc.yaml)

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus
  namespace: kube-ops
  labels:
    app: prometheus
spec:
  selector:
    app: prometheus
  type: NodePort
  ports:
    - name: web
      port: 9090
      targetPort: http
```

为了方便测试，我们这里创建一个 `NodePort` 类型的服务，当然我们可以创建一个 `Ingress` 对象，通过域名来进行访问：

```
$ kubectl create -f prometheus-svc.yaml
service "prometheus" created
$ kubectl get svc -n kube-ops
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AG
E
prometheus   NodePort   10.111.118.104   <none>        9090:30987/TCP      24
S
```

然后我们就可以通过<http://任意节点IP:30987>访问 prometheus 的 webui 服务了。

The screenshot shows the Prometheus web interface with the following details:

- Header:** Prometheus Time Series Collector | Not Secure | k8s.haimaxy.com:30987/graph
- Top Bar:** Prometheus, Alerts, Graph, Status, Help
- Message Bar:** Warning! Detected 28797.66 seconds time difference between your browser and the server. Prometheus relies on accurate time and time drift might cause unexpected query results.
- Query Input:** Expression (press Shift+Enter for newlines)
- Buttons:** Execute, - insert metric at cursor -
- Graph Selection:** Graph (selected), Console
- Data Table:** Element, Value
no data
- Buttons:** Remove Graph, Add Graph

prometheus webui

为了数据的一致性，prometheus 所有的数据都是使用的 UTC 时间，所以我们默认打开的 dashboard 中有这样一个警告，我们需要在查询的时候指定我们当前的时间才可以。然后我们可以查看当前监控系统中的一些监控目标：

The screenshot shows the Prometheus Targets page with the following details:

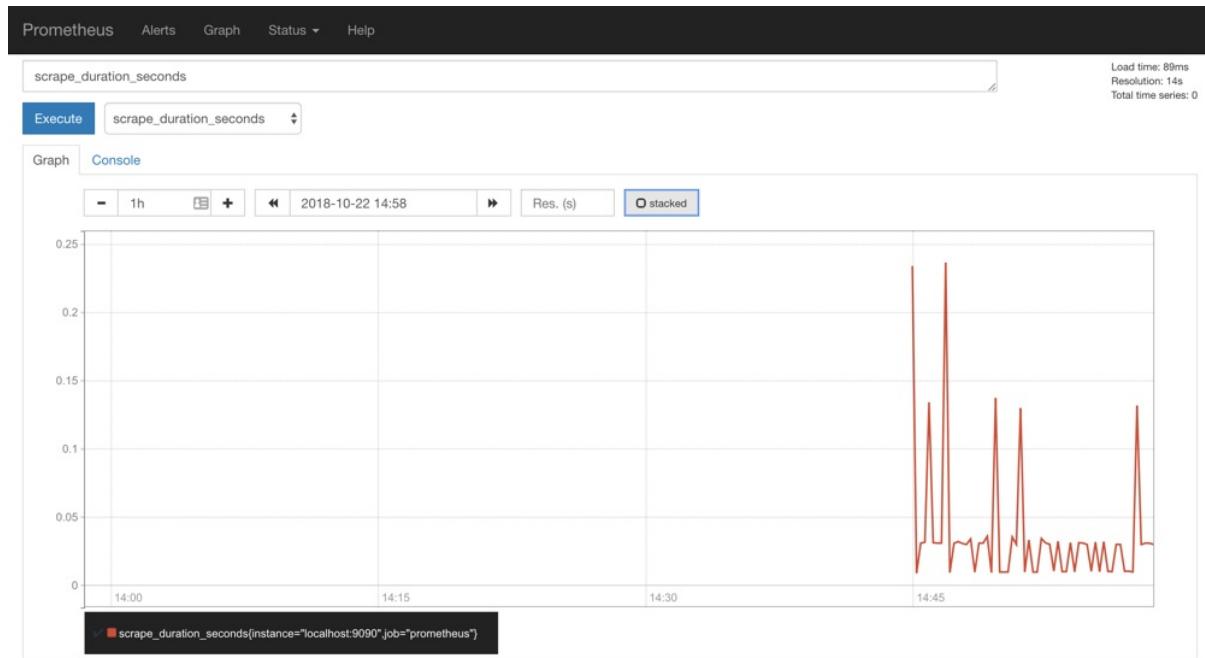
- Header:** Prometheus Time Series Collector | Not Secure | k8s.haimaxy.com:30987/targets
- Top Bar:** Prometheus, Alerts, Graph, Status, Help
- Section:** Targets
- Buttons:** All (selected), Unhealthy
- Table Headers:** Endpoint, State, Labels, Last Scrape, Error
- Table Data:**

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	1.505s ago	
- Context Menu:** Runtime & Build Information, Command-Line Flags, Configuration, Rules, Targets, Service Discovery

由于我们现在还没有配置任何的报警信息，所以 Alerts 菜单下面现在没有任何数据，隔一会儿，我们可以去 Graph 菜单下面查看我们抓取的 prometheus 本身的一些监控数据了，其中 - insert metrics at cursor - 下面就是我们搜集到的一些监控数据指标：

The screenshot shows the Prometheus Query Editor interface. On the left, there are tabs for 'Execute', 'Graph' (which is selected), 'Element', and 'Add Graph'. The 'Graph' tab has a sub-menu with the option 'no data'. The main area is a text input field labeled 'Expression (press Shift+Enter for newlines)'. A dropdown menu is open, listing various Prometheus metrics. The metric 'go_gc_duration_seconds_sum' is highlighted with a blue background, indicating it is selected. Other metrics listed include: go_gc_duration_seconds, go_gc_duration_seconds_count, go_goroutines, go_info, go_memstats_alloc_bytes, go_memstats_alloc_bytes_total, go_memstats_buck_hash_sys_bytes, go_memstats_frees_total, go_memstats_gc_cpu_fraction, go_memstats_gc_sys_bytes, go_memstats_heap_alloc_bytes, go_memstats_heap_idle_bytes, go_memstats_heap_inuse_bytes, go_memstats_heap_objects, go_memstats_heap_released_bytes, go_memstats_heap_sys_bytes, go_memstats_last_gc_time_seconds, go_memstats_lookups_total, go_memstats_mallocs_total, go_memstats_mcache_inuse_bytes, go_memstats_mcache_sys_bytes, and go_memstats_mspan_inuse_bytes.

比如我们这里就选择 `scrape_duration_seconds` 这个指标，然后点击 Execute，如果这个时候没有查询到任何数据，我们可以切换到 Graph 这个 tab 下面重新选择下时间，选择到当前的时间点，重新执行，就可以看到类似于下面的图表数据了：



除了简单的直接使用采集到的一些监控指标数据之外，这个时候也可以使用强大的 PromQL 工具，PromQL 其实就是 prometheus 便于数据聚合展示开发的一套 ad hoc 查询语言的，你想要查什么找对应函数取你的数据好了。

下节课我们再来和大家学习怎样使用 Prometheus 来监控 Kubernetes 系统的组件以及常用的资源对象的监控。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:05:30

53. 监控 Kubernetes 集群应用

上一节我们和大家介绍了 Prometheus 的数据指标是通过一个公开的 HTTP(S) 数据接口获取到的，我们不需要单独安装监控的 agent，只需要暴露一个 metrics 接口，Prometheus 就会定期去拉取数据；对于一些普通的 HTTP 服务，我们完全可以直接重用这个服务，添加一个 /metrics 接口暴露给 Prometheus；而且获取到的指标数据格式是非常易懂的，不需要太高的学习成本。

现在很多服务从一开始就内置了一个 /metrics 接口，比如 Kubernetes 的各个组件、istio 服务网格都直接提供了数据指标接口。有一些服务即使没有原生集成该接口，也完全可以使用一些 exporter 来获取到指标数据，比如 mysql_exporter、node_exporter，这些 exporter 就有点类似于传统监控服务中的 agent，作为一直服务存在，用来收集目标服务的指标数据然后直接暴露给 Prometheus。

普通应用监控

前面我们已经和大家学习了 ingress 的使用，我们采用的是 Traefik 作为我们的 ingress-controller，是我们 Kubernetes 集群内部服务和外部用户之间的桥梁。Traefik 本身内置了一个 /metrics 的接口，但是需要我们在参数中配置开启：

```
[metrics]
[metrics.prometheus]
entryPoint = "traefik"
buckets = [0.1, 0.3, 1.2, 5.0]
```

之前的版本中是通过 --web 和 --web.metrics.prometheus 两个参数进行开启的，要注意查看对应版本的文档。

我们需要在 `traefik.toml` 的配置文件中添加上面的配置信息，然后更新 ConfigMap 和 Pod 资源对象即可，Traefik Pod 运行后，我们可以看到我们的服务 IP：

```
$ kubectl get svc -n kube-system
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
...
traefik-ingress-service   NodePort    10.101.33.56    <none>        80:31692/TCP,8080:321
15/TCP      63d
```

然后我们可以使用 `curl` 检查是否开启了 Prometheus 指标数据接口，或者通过 NodePort 访问也可以：

```
$ curl 10.101.33.56:8080/metrics
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0.000121036
go_gc_duration_seconds{quantile="0.25"} 0.000210328
go_gc_duration_seconds{quantile="0.5"} 0.000279974
go_gc_duration_seconds{quantile="0.75"} 0.000420738
go_gc_duration_seconds{quantile="1"} 0.001191494
```

```
go_gc_duration_seconds_sum 0.004353914
go_gc_duration_seconds_count 12
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 63
....
```

从这里可以看到 Traefik 的监控数据接口已经开启成功了，然后我们就可以将这个 `/metrics` 接口配置到 `prometheus.yml` 中去了，直接加到默认的 `prometheus` 这个 job 下面：(`prome-cm.yaml`)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: kube-ops
data:
  prometheus.yml: |
    global:
      scrape_interval: 30s
      scrape_timeout: 30s

    scrape_configs:
    - job_name: 'prometheus'
      static_configs:
      - targets: ['localhost:9090']

    - job_name: 'traefik'
      static_configs:
      - targets: ['traefik-ingress-service.kube-system.svc.cluster.local:8080']
```

当然，我们这里只是一个很简单的配置，`scrape_configs` 下面可以支持很多参数，例如：

- `basic_auth` 和 `bearer_token`: 比如我们提供的 `/metrics` 接口需要 basic 认证的时候，通过传统的用户名/密码或者在请求的header中添加对应的 token 都可以支持
- `kubernetes_sd_configs` 或 `consul_sd_configs`: 可以用来自动发现一些应用的监控数据

由于我们这里 Traefik 对应的 servicename 是 `traefik-ingress-service`，并且在 `kube-system` 这个 namespace 下面，所以我们这里的 `targets` 的路径配置则需要使用 FQDN 的形式：`traefik-ingress-service.kube-system.svc.cluster.local`，当然如果你的 Traefik 和 Prometheus 都部署在同一个命名空间的话，则直接填 `servicename:serviceport` 即可。然后我们重新更新这个 ConfigMap 资源对象：

```
$ kubectl delete -f pome-cm.yaml
configmap "prometheus-config" deleted
$ kubectl create -f pome-cm.yaml
configmap "prometheus-config" created
```

现在 Prometheus 的配置文件内容已经更改了，隔一会儿被挂载到 Pod 中的 `prometheus.yml` 文件也会更新，由于我们之前的 Prometheus 启动参数中添加了 `--web.enable-lifecycle` 参数，所以现在我们只需要执行一个 `reload` 命令即可让配置生效：

\$ kubectl get svc -n kube-ops	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
--------------------------------	------	------	------------	-------------	---------	-----

```
prometheus  NodePort  10.102.74.90  <none>        9090:30358/TCP  3d
$ curl -X POST "http://10.102.74.90:9090/-/reload"
```

由于 ConfigMap 通过 Volume 的形式挂载到 Pod 中去的热更新需要一定的间隔时间才会生效，所以需要稍微等一小会儿。

reload 这个 url 是一个 POST 请求，所以这里我们通过 service 的 CLUSTER-IP:PORT 就可以访问到这个重载的接口，这个时候我们再去看 Prometheus 的 Dashboard 中查看采集的目标数据：

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	23.283s ago	

Endpoint	State	Labels	Last Scrape	Error
http://traefik-ingress-service.kube-system.svc.cluster.local:8080/metrics	UP	instance="traefik-ingress-service.kube-system.svc.cluster.local:8080"	28.444s ago	

可以看到我们刚刚添加的 traefik 这个任务已经出现了，然后同样的我们可以切换到 Graph 下面去，我们可以找到一些 Traefik 的指标数据，至于这些指标数据代表什么意义，一般情况下，我们可以去查看对应的 /metrics 接口，里面一般情况下都会有对应的注释。

到这里我们就在 Prometheus 上配置了第一个 Kubernetes 应用。

使用 exporter 监控应用

上面我们也说过有一些应用可能没有自带 /metrics 接口供 Prometheus 使用，在这种情况下，我们就需要利用 exporter 服务来为 Prometheus 提供指标数据了。Prometheus 官方为许多应用就提供了对应的 exporter 应用，也有许多第三方的实现，我们可以前往官方网站进行查看：[exporters](#)

比如我们这里通过一个redis-exporter的服务来监控 redis 服务，对于这类应用，我们一般会以 sidecar 的形式和主应用部署在同一个 Pod 中，比如我们这里来部署一个 redis 应用，并用 redis-exporter 的方式来采集监控数据供 Prometheus 使用，如下资源清单文件：(prome-redis.yaml)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: redis
  namespace: kube-ops
spec:
  template:
```

```

metadata:
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/port: "9121"
  labels:
    app: redis
spec:
  containers:
    - name: redis
      image: redis:4
      resources:
        requests:
          cpu: 100m
          memory: 100Mi
      ports:
        - containerPort: 6379
    - name: redis-exporter
      image: oliver006/redis_exporter:latest
      resources:
        requests:
          cpu: 100m
          memory: 100Mi
      ports:
        - containerPort: 9121
  ---
kind: Service
apiVersion: v1
metadata:
  name: redis
  namespace: kube-ops
spec:
  selector:
    app: redis
  ports:
    - name: redis
      port: 6379
      targetPort: 6379
    - name: prom
      port: 9121
      targetPort: 9121

```

可以看到上面我们在 redis 这个 Pod 中包含了两个容器，一个就是 redis 本身的主应用，另外一个容器就是 redis_exporter。现在直接创建上面的应用：

```
$ kubectl create -f prome-redis.yaml
deployment.extensions "redis" created
service "redis" created
```

创建完成后，我们可以看到 redis 的 Pod 里面包含有两个容器：

```
$ kubectl get pods -n kube-ops
NAME                  READY   STATUS    RESTARTS   AGE
prometheus-8566cd9699-gt9wh   1/1    Running   0          3d
redis-544b6c8c54-8xd2g     2/2    Running   0          3m
$ kubectl get svc -n kube-ops
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
E
```

prometheus	NodePort	10.102.74.90	<none>	9090:30358/TCP	3d
redis	ClusterIP	10.104.131.44	<none>	6379/TCP,9121/TCP	5m

我们可以通过 9121 端口来校验是否能够采集到数据：

```
$ curl 10.104.131.44:9121/metrics
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 0
...
# HELP redis_used_cpu_user_children used_cpu_user_childrenmetric
# TYPE redis_used_cpu_user_children gauge
redis_used_cpu_user_children{addr="redis://localhost:6379",alias=""} 0
```

同样的，现在我们只需要更新 Prometheus 的配置文件：

```
- job_name: 'redis'
  static_configs:
    - targets: ['redis:9121']
```

由于我们这里的 redis 服务和 Prometheus 处于同一个 namespace，所以我们直接使用 servicename 即可。

配置文件更新后，重新加载：

```
$ kubectl delete -f prome-cm.yaml
configmap "prometheus-config" deleted
$ kubectl create -f prome-cm.yaml
configmap "prometheus-config" created
# 隔一会儿执行reload操作
$ curl -X POST "http://10.102.74.90:9090/-/reload"
```

这个时候我们再去看 Prometheus 的 Dashboard 中查看采集的目标数据：

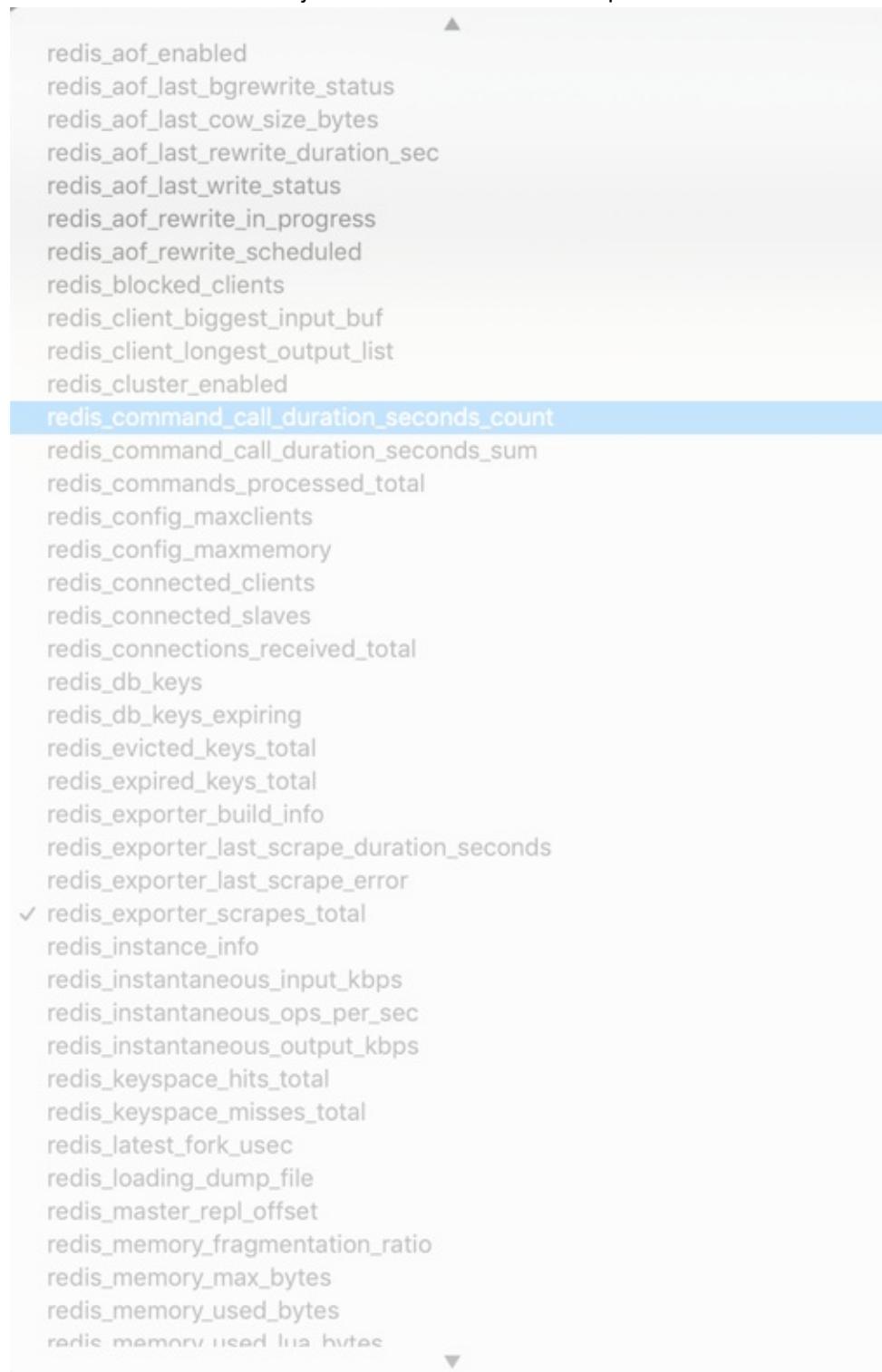
The screenshot shows the Prometheus Dashboard's Targets page. At the top, there are tabs for Prometheus, Alerts, Graph, Status, and Help. Below the tabs, the title "Targets" is displayed. Under "Targets", there are three sections: "prometheus (1/1 up)", "redis (1/1 up)", and "traefik (1/1 up)". Each section has a "show less" link. Each section contains a table with columns: Endpoint, State, Labels, Last Scrape, and Error. The "prometheus" section shows one endpoint: http://localhost:9090/metrics, which is UP and was scraped 29.423s ago. The "redis" section shows one endpoint: http://redis:9121/metrics, which is UP and was scraped 966ms ago. The "traefik" section shows one endpoint: http://traefik-ingress-service.kube-system.svc.cluster.local:8080/metrics, which is UP and was scraped 4.585s ago.

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	29.423s ago	

Endpoint	State	Labels	Last Scrape	Error
http://redis:9121/metrics	UP	instance="redis:9121"	966ms ago	

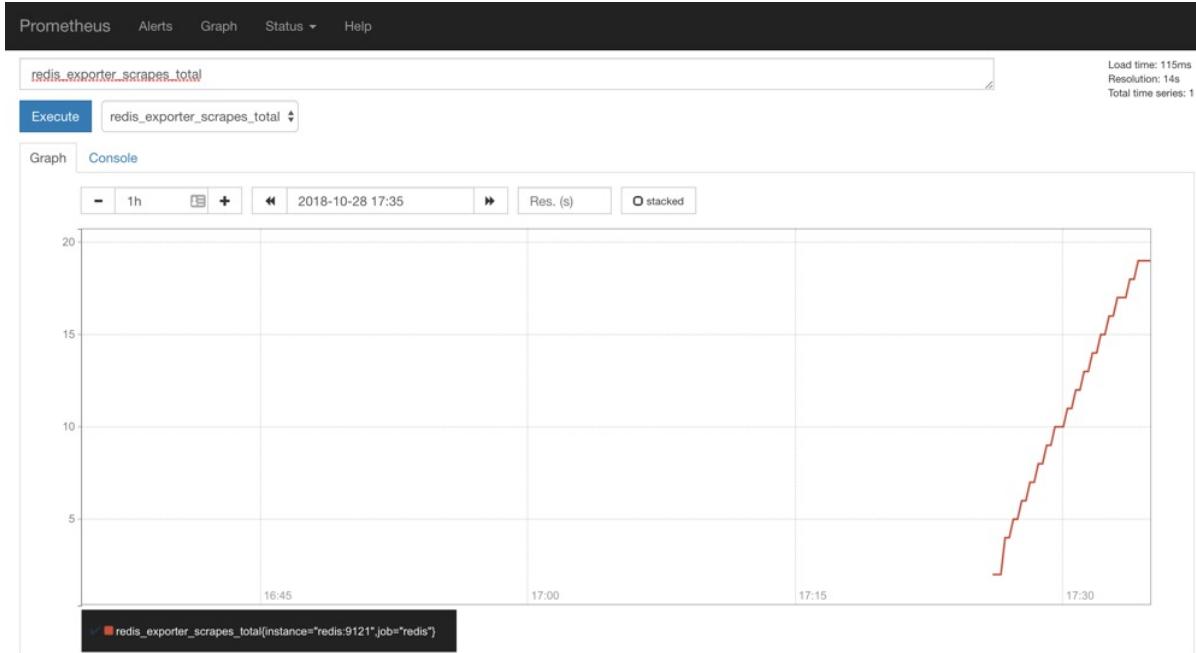
Endpoint	State	Labels	Last Scrape	Error
http://traefik-ingress-service.kube-system.svc.cluster.local:8080/metrics	UP	instance="traefik-ingress-service.kube-system.svc.cluster.local:8080"	4.585s ago	

可以看到配置的 redis 这个 job 已经生效了。切换到 Graph 下面可以看到很多关于 redis 的指标数据：



```
redis_aof_enabled
redis_aof_last_bgrewrite_status
redis_aof_last_cow_size_bytes
redis_aof_last_rewrite_duration_sec
redis_aof_last_write_status
redis_aof_rewrite_in_progress
redis_aof_rewrite_scheduled
redis_blocked_clients
redis_client_biggest_input_buf
redis_client_longest_output_list
redis_cluster_enabled
redis_command_call_duration_seconds_count
redis_command_call_duration_seconds_sum
redis_commands_processed_total
redis_config_maxclients
redis_config_maxmemory
redis_connected_clients
redis_connected_slaves
redis_connections_received_total
redis_db_keys
redis_db_keys_expiring
redis_evicted_keys_total
redis_expired_keys_total
redis_exporter_build_info
redis_exporter_last_scrape_duration_seconds
redis_exporter_last_scrape_error
✓ redis_exporter_scrapes_total
redis_instance_info
redis_instantaneous_input_kbps
redis_instantaneous_ops_per_sec
redis_instantaneous_output_kbps
redis_keyspace_hits_total
redis_keyspace_misses_total
redis_latest_fork_usec
redis_loading_dump_file
redis_master_repl_offset
redis_memory_fragmentation_ratio
redis_memory_max_bytes
redis_memory_used_bytes
redis memory used lru bytes
```

我们选择任意一个指标，比如 `redis_exporter_scrapes_total`，然后点击执行就可以看到对应的数据图表了：



注意，如果时间有问题，我们需要手动在 Graph 下面调整下时间

除了监控群集中部署的服务之外，我们下节课再和大家学习怎样监视 Kubernetes 群集本身。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 `k8s技术圈`)关注我们的微信公众帐号，在微信公众帐号中回复 `加群` 即可加入到我们的 `kubernetes` 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:05:35

54. 监控 Kubernetes 集群节点

上节课我们和大家学习了怎样用 Prometheus 来监控 Kubernetes 集群中的应用，但是对于 Kubernetes 集群本身的监控也是非常重要的，我们需要时时刻刻了解集群的运行状态。

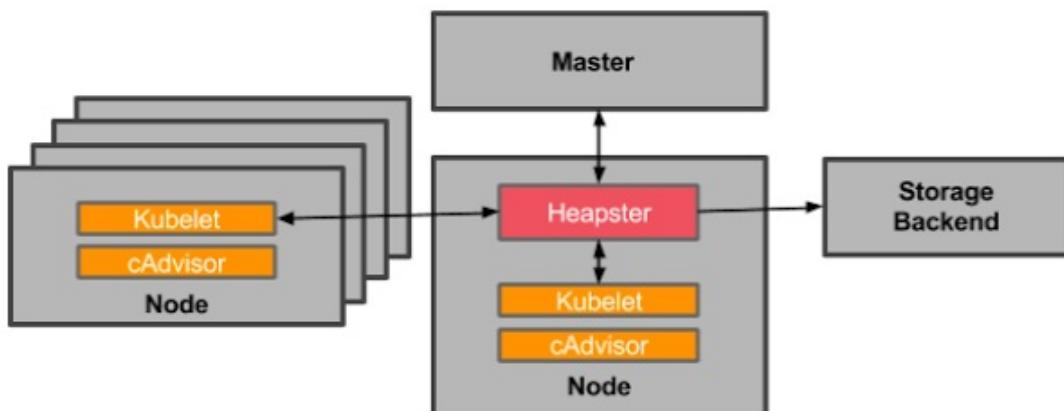
对于集群的监控一般我们需要考虑以下几个方面：

- Kubernetes 节点的监控：比如节点的 cpu、load、disk、memory 等指标
- 内部系统组件的状态：比如 kube-scheduler、kube-controller-manager、kubedns/coredns 等组件的详细运行状态
- 编排级的 metrics：比如 Deployment 的状态、资源请求、调度和 API 延迟等数据指标

监控方案

Kubernetes 集群的监控方案目前主要有以下几种方案：

- Heapster：Heapster 是一个集群范围的监控和数据聚合工具，以 Pod 的形式运行在集群中。



除了 Kubelet/cAdvisor 之外，我们还可以向 Heapster 添加其他指标源数据，比如 kube-state-metrics，我们会在下面和大家讲解的

需要注意的是 Heapster 已经被废弃了，后续版本中会使用 metrics-server 替代。

- cAdvisor：cAdvisor是 Google 开源的容器资源监控和性能分析工具，它是专门为容器而生，本身也支持 Docker 容器，在 Kubernetes 中，我们不需要单独去安装，cAdvisor 作为 kubelet 内置的一部分程序可以直接使用。
- Kube-state-metrics：kube-state-metrics通过监听 API Server 生成有关资源对象的状态指标，比如 Deployment、Node、Pod，需要注意的是 kube-state-metrics 只是简单提供一个 metrics 数据，并不会存储这些指标数据，所以我们可以使用 Prometheus 来抓取这些数据然后存储。
- metrics-server：metrics-server 也是一个集群范围内的资源数据聚合工具，是 Heapster 的替代品，同样的，metrics-server 也只是显示数据，并不提供数据存储服务。

不过 kube-state-metrics 和 metrics-server 之间还是有很大不同的，二者的主要区别如下：

- kube-state-metrics 主要关注的是业务相关的一些元数据，比如 Deployment、Pod、副本状态等

- metrics-server 主要关注的是[资源度量 API](#) 的实现，比如 CPU、文件描述符、内存、请求延时等指标。

监控集群节点

现在我们就来开始我们集群的监控工作，首先来监控我们集群的节点，要监控节点其实我们已经有很多非常成熟的方案了，比如 Nagios、zabbix，甚至我们自己来收集数据也可以，我们这里通过 Prometheus 来采集节点的监控指标数据，可以通过[node_exporter](#)来获取，顾名思义，node_exporter 就是抓取用于采集服务器节点的各种运行指标，目前 node_exporter 支持几乎所有常见的监控点，比如 conntrack, cpu, diskstats, filesystem, loadavg, meminfo, netstat 等，详细的监控点列表可以参考其[Github repo](#)。

我们可以通过 DaemonSet 控制器来部署该服务，这样每一个节点都会自动运行一个这样的 Pod，如果我们从集群中删除或者添加节点后，也会进行自动扩展。

在部署 node-exporter 的时候有一些细节需要注意，如下资源清单文件：(prome-node-exporter.yaml)

```

apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: node-exporter
  namespace: kube-ops
  labels:
    name: node-exporter
spec:
  template:
    metadata:
      labels:
        name: node-exporter
    spec:
      hostPID: true
      hostIPC: true
      hostNetwork: true
      containers:
        - name: node-exporter
          image: prom/node-exporter:v0.16.0
          ports:
            - containerPort: 9100
          resources:
            requests:
              cpu: 0.15
          securityContext:
            privileged: true
        args:
          - --path.procfs
          - /host/proc
          - --path.sysfs
          - /host/sys
          - --collector.filesystem.ignored-mount-points
            -- "/^(sys|proc|dev|host|etc)(\$|/)"'
      volumeMounts:
        - name: dev
          mountPath: /host/dev
        - name: proc
          mountPath: /host/proc

```

```

- name: sys
  mountPath: /host/sys
- name: rootfs
  mountPath: /rootfs
tolerations:
- key: "node-role.kubernetes.io/master"
  operator: "Exists"
  effect: "NoSchedule"
volumes:
- name: proc
  hostPath:
    path: /proc
- name: dev
  hostPath:
    path: /dev
- name: sys
  hostPath:
    path: /sys
- name: rootfs
  hostPath:
    path: /

```

由于我们要获取到的数据是主机的监控指标数据，而我们的 node-exporter 是运行在容器中的，所以我们在 Pod 中需要配置一些 Pod 的安全策略，这里我们就添加了 `hostPID: true`、`hostIPC: true`、`hostNetwork: true` 3个策略，用来使用主机的 PID namespace、IPC namespace 以及主机网络，这些 namespace 就是用于容器隔离的关键技术，要注意这里的 namespace 和集群中的 namespace 是两个完全不相同的概念。

另外我们还将主机的 `/dev`、`/proc`、`/sys` 这些目录挂载到容器中，这些因为我们采集的很多节点数据都是通过这些文件夹下面的文件来获取到的，比如我们在使用 `top` 命令可以查看当前 cpu 使用情况，数据就来源于文件 `/proc/stat`，使用 `free` 命令可以查看当前内存使用情况，其数据来源是来自 `/proc/meminfo` 文件。

另外由于我们集群使用的是 kubeadm 搭建的，所以如果希望 master 节点也一起被监控，则需要添加相应的容忍，对于污点和容忍还不是很熟悉的同学可以在前面的章节中回顾下。

然后直接创建上面的资源对象即可：

```

$ kubectl create -f prome-node-exporter.yaml
daemonset.extensions "node-exporter" created
$ kubectl get pods -n kube-ops -o wide
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE
node-exporter-jfwfv   1/1     Running   0          30m    10.151.30.63   node
02
node-exporter-kr8rt   1/1     Running   0          30m    10.151.30.64   node
03
node-exporter-whb7n   1/1     Running   0          20m    10.151.30.57   mast
er
prometheus-8566cd9699-gt9wh  1/1     Running   0          4d     10.244.4.39    node
02
redis-544b6c8c54-8xd2g   2/2     Running   0          23h    10.244.2.87    node
03

```

部署完成后，我们可以看到在3个节点上都运行了一个 Pod，有的同学可能会说我们这里不需要创建一个 Service 吗？我们应该怎样去获取 /metrics 数据呢？我们上面是不是指定了 hostNetwork=true，所以在每个节点上就会绑定一个端口 9100，我们可以通过这个端口去获取到监控指标数据：

```
$ curl 127.0.0.1:9100/metrics
...
node_filesystem_device_error{device="shm",fstype="tmpfs",mountpoint="/rootfs/var/lib/docker/containers/aefe8b1b63c3aa5f27766053ec817415faf8f6f417bb210d266fef0c2da64674/shm"} 1
node_filesystem_device_error{device="shm",fstype="tmpfs",mountpoint="/rootfs/var/lib/docker/containers/c8652ca72230496038a07e4fe4ee47046abb5f88d9d2440f0c8a923d5f3e133c/shm"} 1
node_filesystem_device_error{device="tmpfs",fstype="tmpfs",mountpoint="/dev"} 0
node_filesystem_device_error{device="tmpfs",fstype="tmpfs",mountpoint="/dev/shm"} 0
...
```

当然如果你觉得上面的手动安装方式比较麻烦，我们也可以使用 Helm 的方式来安装：

```
$ helm install --name node-exporter stable/prometheus-node-exporter --namespace kube-ops
```

服务发现

由于我们这里3个节点上面都运行了 node-exporter 程序，如果我们通过一个 Service 来将数据收集到一起用静态配置的方式配置到 Prometheus 去中，就只会显示一条数据，我们得自己在指标数据中去过滤每个节点的数据，那么有没有一种方式可以让 Prometheus 去自动发现我们节点的 node-exporter 程序，并且按节点进行分组呢？是有的，就是我们前面和大家提到过的服务发现。

在 Kubernetes 下，Promethues 通过与 Kubernetes API 集成，目前主要支持5中服务发现模式，分别是：Node、Service、Pod、Endpoints、Ingress。

我们通过 kubectl 命令可以很方便的获取到当前集群中的所有节点信息：

```
$ kubectl get nodes
NAME     STATUS   ROLES      AGE      VERSION
master   Ready    master     165d    v1.10.0
node02   Ready    <none>    85d     v1.10.0
node03   Ready    <none>    145d    v1.10.0
```

但是要让 Prometheus 也能够获取到当前集群中的所有节点信息的话，我们就需要利用 Node 的服务发现模式，同样的，在 prometheus.yml 文件中配置如下的 job 任务即可：

```
- job_name: 'kubernetes-nodes'
  kubernetes_sd_configs:
  - role: node
```

通过指定 kubernetes_sd_configs 的模式为 node，Prometheus 就会自动从 Kubernetes 中发现所有的 node 节点并作为当前 job 监控的目标实例，发现的节点 /metrics 接口是默认的 kubelet 的 HTTP 接口。

prometheus 的 ConfigMap 更新完成后，同样的我们执行 reload 操作，让配置生效：

```
$ kubectl delete -f prome-cm.yaml
configmap "prometheus-config" deleted
$ kubectl create -f prome-cm.yaml
configmap "prometheus-config" created
# 隔一会儿再执行下面的 reload 操作
$ kubectl get svc -n kube-ops
NAME           TYPE      CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
prometheus    NodePort   10.102.74.90   <none>        9090:30358/TCP
5d
.....
$ curl -X POST "http://10.102.74.90:9090/-/reload"
```

配置生效后，我们再去 prometheus 的 dashboard 中查看 Targets 是否能够正常抓取数据，访问任意节点IP:30358：

The screenshot shows the Prometheus Targets page. At the top, there are tabs for All (selected) and Unhealthy. Below is a table for the 'kubernetes-nodes (0/3 up)' section:

Endpoint	State	Labels	Last Scrape	Error
http://10.151.30.57:10250/metrics	DOWN	instance="master"	11.22s ago	Get http://10.151.30.57:10250/metrics: net/http: HTTP/1.x transport connection broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
http://10.151.30.63:10250/metrics	DOWN	instance="node02"	26.377s ago	Get http://10.151.30.63:10250/metrics: net/http: HTTP/1.x transport connection broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
http://10.151.30.64:10250/metrics	DOWN	instance="node03"	16.794s ago	Get http://10.151.30.64:10250/metrics: net/http: HTTP/1.x transport connection broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02"

Below is a table for the 'prometheus (1/1 up)' section:

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	28.653s ago	

Finally, there is a table for the 'redis (1/1 up)' section:

Endpoint	State	Labels	Last Scrape	Error
http://redis:9121/metrics	UP	instance="redis:9121"	196ms ago	

prometheus nodes target

我们可以看到上面的 `kubernetes-nodes` 这个 job 任务已经自动发现了我们3个 node 节点，但是在获取数据的时候失败了，出现了类似于下面的错误信息：

```
Get http://10.151.30.57:10250/metrics: net/http: HTTP/1.x transport connection broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
```

这是因为 prometheus 去发现 Node 模式的服务的时候，访问的端口默认是10250，而现在该端口下面已经没有了 `/metrics` 指标数据了，现在 kubelet 只读的数据接口统一通过10255端口进行暴露了，所以我们应该去替换掉这里的端口，但是我们是要替换成10255端口吗？不是的，因为我们是要去配置上面通过 `node-exporter` 抓取到的节点指标数据，而我们上面是不是指定了 `hostNetwork=true`，所以在每个节点上就会绑定一个端口9100，所以我们应该将这里的10250替换成9100，但是应该怎样替换呢？

这里我们就需要使用到 Prometheus 提供的 `relabel_configs` 中的 `replace` 能力了，`relabel` 可以在 Prometheus 采集数据之前，通过 Target 实例的 Metadata 信息，动态重新写入 Label 的值。除此之外，我们还能根据 Target 实例的 Metadata 信息选择是否采集或者忽略该 Target 实例。比如我们这里就可以去匹配 `__address__` 这个 Label 标签，然后替换掉其中的端口：

```
- job_name: 'kubernetes-nodes'
  kubernetes_sd_configs:
    - role: node
      relabel_configs:
        - source_labels: [__address__]
          regex: '(.*):10250'
          replacement: '${1}:9100'
          target_label: __address__
          action: replace
```

这里就是一个正则表达式，去匹配 `__address__`，然后将 host 部分保留下来，port 替换成了9100，现在我们重新更新配置文件，执行 `reload` 操作，然后再去看 Prometheus 的 Dashboard 的 Targets 路径下面 `kubernetes-nodes` 这个 job 任务是否正常了：

Endpoint	State	Labels	Last Scrape	Error
http://10.151.30.57:9100/metrics	UP	instance="master"	20.181s ago	
http://10.151.30.63:9100/metrics	UP	instance="node02"	9.251s ago	
http://10.151.30.64:9100/metrics	UP	instance="node03"	15.882s ago	

Endpoint	State	Labels	Last Scrape	Error
http://localhost:9090/metrics	UP	instance="localhost:9090"	29.468s ago	

prometheus nodes target2

我们可以看到现在已经正常了，但是还有一个问题就是我们采集的指标数据 Label 标签就只有一个节点的 `hostname`，这对于我们在进行监控分组分类查询的时候带来了很多不方便的地方，要是我们能够将集群中 Node 节点的 Label 标签也能获取到就很好了。

这里我们可以通过 `labelmap` 这个属性来将 Kubernetes 的 Label 标签添加为 Prometheus 的指标标签：

```
- job_name: 'kubernetes-nodes'
  kubernetes_sd_configs:
    - role: node
      relabel_configs:
        - source_labels: [__address__]
          regex: '(.*):10250'
          replacement: '${1}:9100'
          target_label: __address__
          action: replace
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)
```

添加了一个 action 为 `labelmap`，正则表达式是 `__meta_kubernetes_node_label_(.+)` 的配置，这里的意思就是表达式中匹配的数据也添加到指标数据的 Label 标签中去。

对于 `kubernetes_sd_configs` 下面可用的标签如下：可用元标签：

- `__meta_kubernetes_node_name`: 节点对象的名称
- `__meta_kubernetes_node_label`: 节点对象中的每个标签
- `__meta_kubernetes_node_annotation`: 来自节点对象的每个注释
- `__meta_kubernetes_node_address`: 每个节点地址类型的第一个地址（如果存在） *

关于 `kubernetes_sd_configs` 更多信息可以查看官方文档：[kubernetes_sd_config](#)

另外由于 `kubelet` 也自带了一些监控指标数据，就上面我们提到的10255端口，所以我们这里也把 `kubelet` 的监控任务也一并配置上：

```
- job_name: 'kubernetes-nodes'
  kubernetes_sd_configs:
    - role: node
      relabel_configs:
        - source_labels: [__address__]
          regex: '(.*):10250'
          replacement: '${1}:9100'
          target_label: __address__
          action: replace
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)

- job_name: 'kubernetes-kubelet'
  kubernetes_sd_configs:
    - role: node
      relabel_configs:
        - source_labels: [__address__]
          regex: '(.*):10250'
          replacement: '${1}:10255'
          target_label: __address__
          action: replace
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)
```

现在我们再去更新下配置文件，执行 `reload` 操作，让配置生效，然后访问 Prometheus 的 Dashboard 查看 Targets 路径：

kubernetes-kubelet (3/3 up) show less

Endpoint	State	Labels	Last Scrape	Error
http://10.151.30.57:10255/metrics	UP	beta_kubernetes.io_arch="amd64" beta_kubernetes.io_os="linux" instance="master" kubernetes.io_hostname="node01" kubernetes.io_role_kubernetes.io_master=""	3.667s ago	
http://10.151.30.63:10255/metrics	UP	beta_kubernetes.io_arch="amd64" beta_kubernetes.io_os="linux" com="youdianzhishi" course="k8s" instance="node02" kubernetes.io_hostname="node02"	2.765s ago	
http://10.151.30.64:10255/metrics	UP	beta_kubernetes.io_arch="amd64" beta_kubernetes.io_os="linux" instance="node03" jnlp="haiimaxy" kubernetes.io_hostname="node03"	16.704s ago	

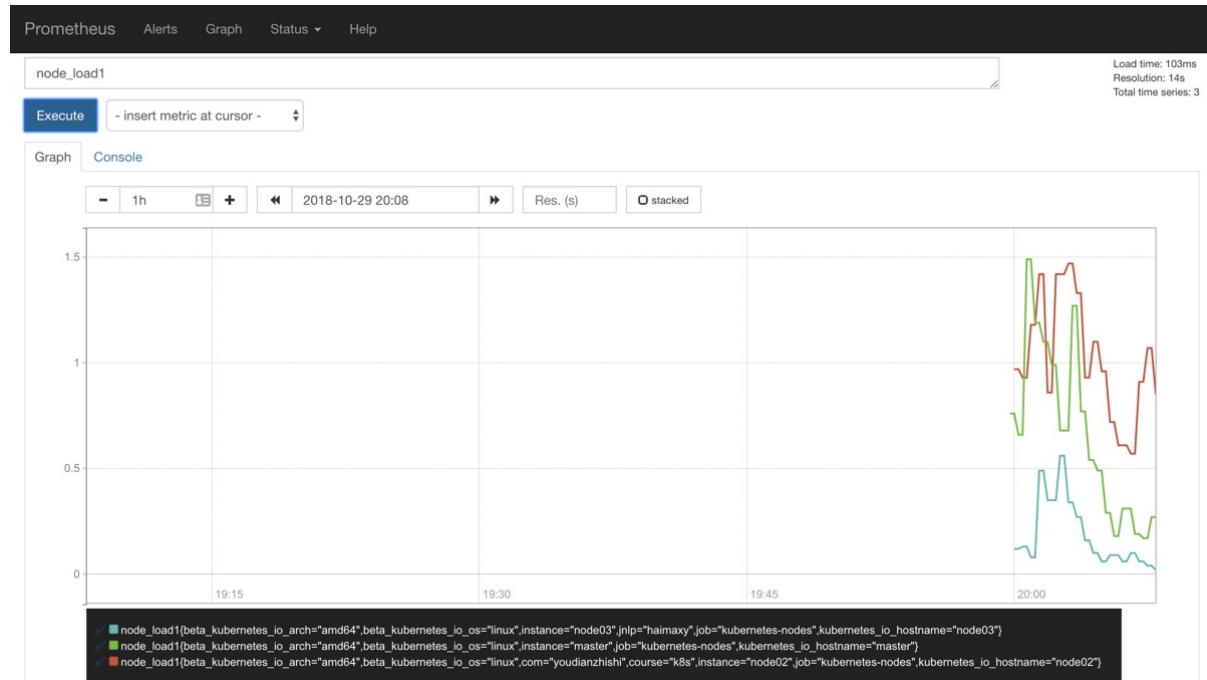
kubernetes-nodes (3/3 up) show less

Endpoint	State	Labels	Last Scrape	Error
http://10.151.30.57:9100/metrics	UP	beta_kubernetes.io_arch="amd64" beta_kubernetes.io_os="linux" instances="master" kubernetes.io_hostname="node01" node_role_kubernetes.io_master=""	13.499s ago	
http://10.151.30.63:9100/metrics	UP	beta_kubernetes.io_arch="amd64" beta_kubernetes.io_os="linux" com="youdianzhishi" course="k8s" instance="node02" kubernetes.io_hostname="node02"	27.383s ago	
http://10.151.30.64:9100/metrics	UP	beta_kubernetes.io_arch="amd64" beta_kubernetes.io_os="linux" instance="node03" jnlp="haiimaxy" kubernetes.io_hostname="node03"	26.297s ago	

prometheus node targets

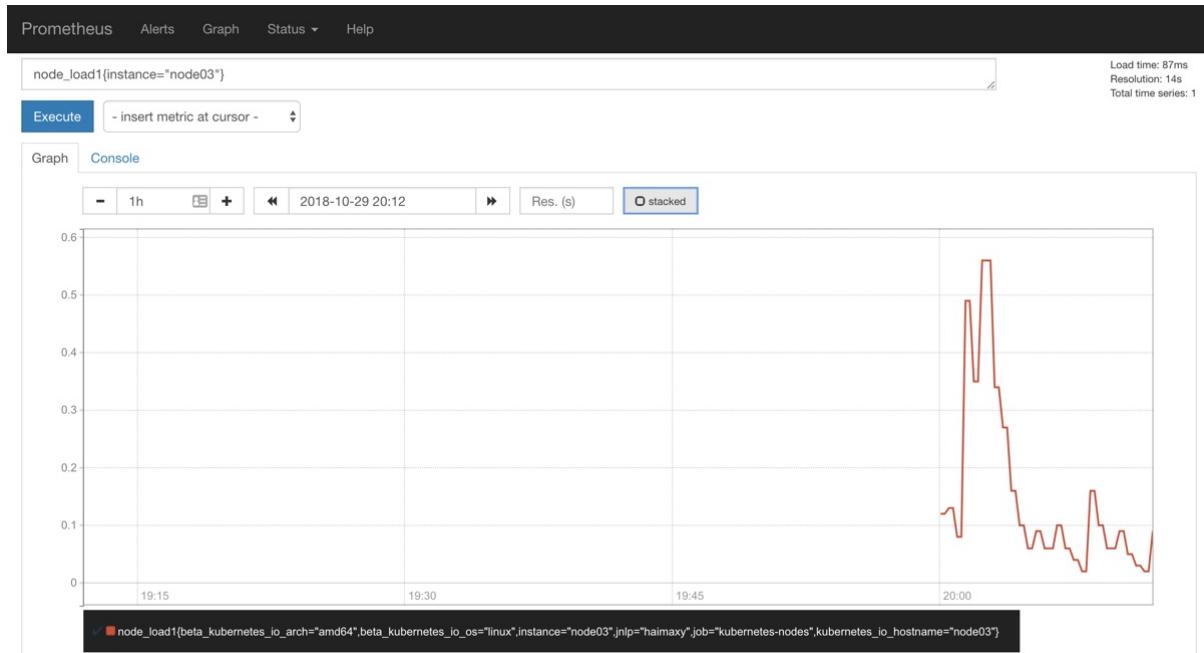
现在可以看到我们上面添加的 `kubernetes-kubelet` 和 `kubernetes-nodes` 这两个 job 任务都已经配置成功了，而且二者的 Labels 标签都和集群的 node 节点标签保持一致了。

现在我们就可以切换到 Graph 路径下面查看采集的一些指标数据了，比如查询 `node_load1` 指标：

*prometheus nodes graph1*

我们可以看到将3个 node 节点对应的 `node_load1` 指标数据都查询出来了，同样的，我们还可以使用 PromQL 语句来进行更复杂的一些聚合查询操作，还可以根据我们的 Labels 标签对指标数据进行聚合，比如我们这里只查询 `node03` 节点的数据，可以使用表达式 `node_load1{instance="node03"}` 来进

行查询：



prometheus nodes graph2

到这里我们就把 Kubernetes 集群节点的使用 Prometheus 监控起来了，下节课我们再来和大家学习怎样监控 Pod 或者 Service 之类的资源对象。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-04-24 15:32:07

55. 监控 Kubernetes 常用资源对象

上节课我们学习了怎样用 Prometheus 来自动发现 Kubernetes 集群的节点，用到了 Prometheus 针对 Kubernetes 的服务发现机制 `kubernetes_sd_configs` 的使用，这节课我们来和大家一起了解下怎样在 Prometheus 中来自动监控 Kubernetes 中的一些常用资源对象。

前面我们和大家介绍过了在 Prometheus 中用静态的方式来监控 Kubernetes 集群中的普通应用，但是如果针对集群中众多的资源对象都采用静态的方式来进行配置的话显然是不现实的，所以同样我们需要使用到 Prometheus 提供的其他类型的服务发现机制。

容器监控

说到容器监控我们自然会想到 `cAdvisor`，我们前面也说过 `cAdvisor` 已经内置在了 `kubelet` 组件之中，所以我们不需要单独去安装，`cAdvisor` 的数据路径为 `/api/v1/nodes/<node>/proxy/metrics`，同样我们这里使用 `node` 的服务发现模式，因为每一个节点下面都有 `kubelet`，自然都有 `cAdvisor` 采集到的数据指标，配置如下：

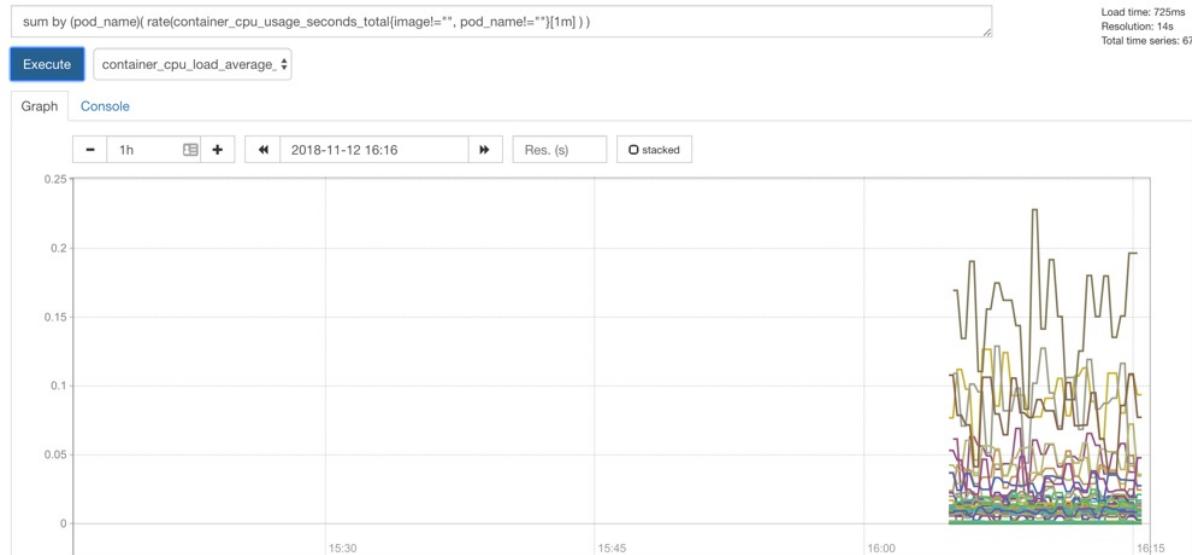
```
- job_name: 'kubernetes-cadvisor'
  kubernetes_sd_configs:
  - role: node
    scheme: https
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
    relabel_configs:
    - action: labelmap
      regex: __meta_kubernetes_node_label_(.+)
    - target_label: __address__
      replacement: kubernetes.default.svc:443
    - source_labels: [__meta_kubernetes_node_name]
      regex: (.+)
      target_label: __metrics_path__
      replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor
```

上面的配置和我们之前配置 `node-exporter` 的时候几乎是一样的，区别是我们这里使用了 `https` 的协议，另外需要注意的是配置了 `ca.cart` 和 `token` 这两个文件，这两个文件是 Pod 启动后自动注入进来的，通过这两个文件我们可以在 Pod 中访问 `apiserver`，比如我们这里的 `__address__` 不再是 `nodeip` 了，而是 `kubernetes` 在集群中的服务地址，然后加上 `__metrics_path__` 的访问路径：`/api/v1/nodes/${1}/proxy/metrics/cadvisor`，现在同样更新下配置，然后查看 Targets 路径：

kubernetes-cadvisor (3/3 up) show less					
Endpoint	State	Labels	Last Scrape	Error	
https://kubernetes.default.svc:443/api/v1/nodes/master/proxy/metrics/cadvisor	UP	<code>beta_kubernetes_io_arch="amd64"</code> <code>beta_kubernetes_io_os="linux"</code> <code>instance="master"</code> <code>kubernetes_io_hostname="master"</code> <code>node_role_kubernetes_io_master="true"</code>	4.08s ago		
https://kubernetes.default.svc:443/api/v1/nodes/node02/proxy/metrics/cadvisor	UP	<code>beta_kubernetes_io_arch="amd64"</code> <code>beta_kubernetes_io_os="linux"</code> <code>com="youdianzhihei"</code> <code>course="k8s"</code> <code>instance="node02"</code> <code>kubernetes_io_hostname="node02"</code>	14.152s ago		
https://kubernetes.default.svc:443/api/v1/nodes/node03/proxy/metrics/cadvisor	UP	<code>beta_kubernetes_io_arch="amd64"</code> <code>beta_kubernetes_io_os="linux"</code> <code>instance="node03"</code> <code>jnp="haimaxy"</code> <code>kubernetes_io_hostname="node03"</code>	17.422s ago		

然后我们可以切换到 Graph 路径下面查询容器相关数据，比如我们这里来查询集群中所有 Pod 的 CPU 使用情况，这里用的数据指标是 `container_cpu_usage_seconds_total`，然后去除一些无效的数据，查询1分钟内的数据，由于查询到的数据都是容器相关的，最好要安装 Pod 来进行聚合，对应的 promQL 语句如下：

```
sum by (pod_name)(rate(container_cpu_usage_seconds_total{image!="", pod_name!=""}[1m] ))
```



prometheus cadvisor graph

我们可以看到上面的结果就是集群中的所有 Pod 在1分钟之内的 CPU 使用情况的曲线图，当然还有很多数据可以获取到，我们后面在需要的时候再和大家介绍。

apiserver 监控

apiserver 作为 Kubernetes 最核心的组件，当然他的监控也是非常有必要的，对于 apiserver 的监控我们可以直接通过 kubernetes 的 Service 来获取：

```
$ kubectl get svc
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes   ClusterIP  10.96.0.1    <none>        443/TCP     175d
```

上面这个 Service 就是我们集群的 apiserver 在集群内部的 Service 地址，要自动发现 Service 类型的服务，我们就需要用到 role 为 Endpoints 的 `kubernetes_sd_configs`，我们可以在 ConfigMap 对象中添加上一个 Endpoints 类型的服务的监控任务：

```
- job_name: 'kubernetes-apiservers'
  kubernetes_sd_configs:
  - role: endpoints
```

上面这个任务是定义的一个类型为 `endpoints` 的 `kubernetes_sd_configs`，添加到 Prometheus 的 `ConfigMap` 的配置文件中，然后更新配置：

```
$ kubectl delete -f prome-cm.yaml
$ kubectl create -f prome-cm.yaml
$ # 隔一会儿执行reload操作
$ kubectl get svc -n kube-ops
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
E
prometheus   NodePort    10.102.74.90    <none>        9090:30358/TCP   14d
d
$ curl -X POST "http://10.102.74.90:9090/-/reload"
```

更新完成后，我们再去查看 Prometheus 的 Dashboard 的 target 页面：

Endpoint	State	Labels	Last Scrape	Error
http://10.151.30.57:6443/metrics	DOWN	instance="10.151.30.57:6443"	3.008s ago	Get http://10.151.30.57:6443/metrics: net/http: HTTP/1.x transport connection broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
http://10.244.0.31:8443/metrics	DOWN	instance="10.244.0.31:8443"	4.623s ago	Get http://10.244.0.31:8443/metrics: net/http: HTTP/1.x transport connection broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
http://10.244.0.36:443/metrics	DOWN	instance="10.244.0.36:443"	18.948s ago	Get http://10.244.0.36:443/metrics: net/http: HTTP/1.x transport connection broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
http://10.244.0.36:80/metrics	DOWN	instance="10.244.0.36:80"	21.556s ago	server returned HTTP status 404 Not Found
http://10.244.0.36:8080/metrics	UP	instance="10.244.0.36:8080"	24.849s ago	
http://10.244.2.83:80/metrics	DOWN	instance="10.244.2.83:80"	8.522s ago	server returned HTTP status 404 Not Found
http://10.244.2.85:80/metrics	DOWN	instance="10.244.2.85:80"	13.917s ago	server returned HTTP status 404 Not Found
http://10.244.2.86:3000/metrics	UP	instance="10.244.2.86:3000"	18.968s ago	
http://10.244.2.86:80/metrics	DOWN	instance="10.244.2.86:80"	10.417s ago	Get http://10.244.2.86:80/metrics: dial tcp 10.244.2.86:80: connect: connection refused
http://10.244.2.88:9090/metrics	UP	instance="10.244.2.88:9090"	1.202s ago	

我们可以看到 `kubernetes-apiservers` 下面出现了很多实例，这是因为这里我们使用的是 `Endpoints` 类型的服务发现，所以 Prometheus 把所有的 `Endpoints` 服务都抓取过来了，同样的，上面我们需要的服务名为 `kubernetes` 这个 `apiserver` 的服务也在这个列表之中，那么我们应该怎样来过滤出这个服务来呢？还记得上节课的 `relabel_configs` 吗？没错，同样我们需要使用这个配置，只是我们这里不是使用 `replace` 这个动作了，而是 `keep`，就是只把符合我们要求的给保留下来，哪些才是符合我们要求的呢？我们可以把鼠标放置在任意一个 `target` 上，可以查看到 `Before relabeling` 里面所有的元数据，比如我们要过滤的服务是 `default` 这个 `namespace` 下面，服务名为 `kubernetes` 的元数据，所以这里我们就可以根据对应的 `_meta_kubernetes_namespace` 和 `_meta_kubernetes_service_name` 这两个元数据来 `relabel`

```

Before relabelling:
__address__="10.244.2.86:3000"
__meta_kubernetes_endpoint_address_target_kind="Pod"
__meta_kubernetes_endpoint_address_target_name="grafana-7c8b4d446b-wdzw7"
__meta_kubernetes_endpoint_port_name=""
__meta_kubernetes_endpoint_port_protocol="TCP"
__meta_kubernetes_endpoint_ready="true"
__meta_kubernetes_endpoints_name="grafana"
__meta_kubernetes_namespace="kube-ops"
__meta_kubernetes_pod_container_name="grafana"
__meta_kubernetes_pod_container_port_name="grafana"
__meta_kubernetes_pod_container_port_number="3000"
__meta_kubernetes_pod_container_port_protocol="TCP"
__meta_kubernetes_pod_controller_kind="ReplicaSet"
__meta_kubernetes_pod_controller_name="grafana-7c8b4d446b"
__meta_kubernetes_pod_host_ip="10.151.30.64"
__meta_kubernetes_pod_ip="10.244.2.86"
__meta_kubernetes_pod_label_app="grafana"
__meta_kubernetes_pod_label_template_hash="3746080026"
__meta_kubernetes_pod_name="grafana-7c8b4d446b-wdzw7"
__meta_kubernetes_pod_node_name="node03"
__meta_kubernetes_pod_ready="true"
__meta_kubernetes_pod_uid="132b280d-daa7-11e8-b37d-525400db4df7"
__meta_kubernetes_service_label_app="grafana"
__meta_kubernetes_service_name="grafana"
__metrics_path__="/metrics"
__scheme__="http"
job="kubernetes-apiservers" instance="10.244.2.86:3000"

```

UP instance="10.244.2.86:3000"

prometheus before label

另外由于 kubernetes 这个服务对应的端口是443，需要使用 https 协议，所以这里我们需要使用 https 的协议，对应的就需要将对应的 ca 证书配置上，如下：

```

- job_name: 'kubernetes-apiservers'
  kubernetes_sd_configs:
  - role: endpoints
    scheme: https
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      relabel_configs:
      - source_labels: [__meta_kubernetes_namespace, __meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
        action: keep
        regex: default;kubernetes;https

```

现在重新更新配置文件、重新加载 Prometheus，切换到 Prometheus 的 Targets 路径下查看：

Targets

All Unhealthy

kubernetes-apiservers (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Error
https://10.151.30.57:6443/metrics	UP	instance="10.151.30.57:6443"	26.504s ago	

kubernetes-kubelet (3/3 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Error
http://10.151.30.57:10255/metrics	UP	beta.kubernetes.io_arch="amd64", beta.kubernetes.io_os="linux", instance="master", kubernetes.io_hostname="master", node.role.kubernetes.io_master=""	15.199s ago	
http://10.151.30.63:10255/metrics	UP	beta.kubernetes.io_arch="amd64", beta.kubernetes.io_os="linux", com="youdianzhihui", course="k8s", instance="node02", kubernetes.io_hostname="node02"	14.2s ago	
http://10.151.30.64:10255/metrics	UP	beta.kubernetes.io_arch="amd64", beta.kubernetes.io_os="linux", instance="node03", job="halimayu", kubernetes.io_hostname="node03"	5.228s ago	

promethues apiserver

现在可以看到 kubernetes-apiserver 这个任务下面只有 apiserver 这一个实例了，证明我们的 relabel 是成功的，现在我们切换到 graph 路径下面查看下采集到数据，比如查询 apiserver 的总的请求数：

```
sum(rate(apiserver_request_count[1m]))
```

这里我们使用到了 promql 里面的 rate 和 sum 函数，表示的意思是 apiserver 在1分钟内总的请求数。



这样我们就完成了对 Kubernetes APIServer 的监控。

另外如果我们要来监控其他系统组件，比如 kube-controller-manager、kube-scheduler 的话应该怎么做呢？由于 apiserver 服务 namespace 在 default 使用默认的 Service kubernetes，而其余组件服务在 kube-system 这个 namespace 下面，如果我们想要来监控这些组件的话，需要手动创建单独的 Service，其中 kube-sheduler 的指标数据端口为 10251，kube-controller-manager 对应的端口为 10252，大家可以尝试下自己来配置下这几个系统组件。

Service 的监控

上面的 apiserver 实际上是一种特殊的 Service，现在我们同样来配置一个任务用来专门发现普通类型的 Service：

```
- job_name: 'kubernetes-service-endpoints'
  kubernetes_sd_configs:
    - role: endpoints
```

```

relabel_configs:
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scrape]
  action: keep
  regex: true
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scheme]
  action: replace
  target_label: __scheme__
  regex: (https?)
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
  action: replace
  target_label: __metrics_path__
  regex: (.)
- source_labels: [__address__, __meta_kubernetes_service_annotation_prometheus_io_port]
  action: replace
  target_label: __address__
  regex: ([^:]+)(?::(\d+)?;(\d+))
  replacement: $1:$2
- action: labelmap
  regex: __meta_kubernetes_service_label_(.+)
- source_labels: [__meta_kubernetes_namespace]
  action: replace
  target_label: kubernetes_namespace
- source_labels: [__meta_kubernetes_service_name]
  action: replace
  target_label: kubernetes_name

```

注意我们这里在 relabel_configs 区域做了大量的配置，特别是第一个保

留 __meta_kubernetes_service_annotation_prometheus_io_scrape 为 true 的才保留下，这就是说要想自动发现集群中的 Service，就需要我们在 Service 的 annotation 区域添加 prometheus.io/scrape=true 的声明，现在我们先将上面的配置更新，查看下效果：

kubernetes-service-endpoints (1/1 up) show less				
Endpoint	State	Labels	Last Scrape	Error
http://10.244.4.54:9090/metrics	UP	instance="10.244.4.54:9090" kubernetes.name="prometheus" kubernetes.namespace="istio-system" name="prometheus"	2.908s ago	

service endpoints

我们可以看到 kubernetes-service-endpoints 这一个任务下面只发现了一个服务，这是因为我们在 relabel_configs 中过滤了 annotation 有 prometheus.io/scrape=true 的 Service，而现在我们系统中只有这样一个服务符合要求，所以只出现了一个实例。

现在我们在之前创建的 redis 这个 Service 中添加上 prometheus.io/scrape=true 这个 annotation：(prome-redis-exporter.yaml)

```

kind: Service
apiVersion: v1
metadata:
  name: redis
  namespace: kube-ops
  annotations:
    prometheus.io/scrape: "true"

```

```

    prometheus.io/port: "9121"
spec:
  selector:
    app: redis
  ports:
    - name: redis
      port: 6379
      targetPort: 6379
    - name: prom
      port: 9121
      targetPort: 9121

```

由于 redis 服务的 metrics 接口在9121这个 redis-exporter 服务上面，所以我们还需要添加一个 `prometheus.io/port=9121` 这样的 `annotations`，然后更新这个 Service：

```

$ kubectl apply -f prome-redis-exporter.yaml
deployment.extensions "redis" unchanged
service "redis" changed

```

更新完成后，去 Prometheus 查看 Targets 路径，可以看到 redis 服务自动出现在了 `kubernetes-service-endpoints` 这个任务下面：

[kubernetes-service-endpoints \(2/2 up\)](#) show less

Endpoint	State	Labels	Last Scrape	Error
http://10.244.2.89:9121/metrics	UP	<code>instance="10.244.2.89:9121"</code> <code>kubernetes_name="redis"</code> <code>kubernetes_namespace="kube-ops"</code>	700ms ago	
http://10.244.4.54:9090/metrics	UP	<code>instance="10.244.4.54:9090"</code> <code>kubernetes_name="prometheus"</code> <code>kubernetes_namespace="istio-system"</code> <code>name="prometheus"</code>	25.118s ago	

kubernetes service endpoints

这样以后我们有了新的服务，服务本身提供了 `/metrics` 接口，我们就完全不需要用静态的方式去配置了，到这里我们就可以将之前配置的 redis 的静态配置去掉了。

大家可以尝试去将之前配置的 traefik 服务用动态发现的方式重新配置到上面的 `service-endpoints` 中。

同样的，大家可以自己去尝试下去配置下自动发现 Pod、ingress 这些资源对象。

kube-state-metrics

上面我们配置了自动发现 Service (Pod也是一样的) 的监控，但是这些监控数据都是应用内部的监控，需要应用本身提供一个 `/metrics` 接口，或者对应的 exporter 来暴露对应的指标数据，但是在 Kubernetes 集群上 Pod、DaemonSet、Deployment、Job、CronJob 等各种资源对象的状态也需要监控，这也反映了使用这些资源部署的应用的状态。但通过查看前面从集群中拉取的指标(这些指标主要来自 apiserver 和 kubelet 中集成的 cAdvisor)，并没有具体的各种资源对象的状态指标。对于 Prometheus 来说，当然是需要引入新的 exporter 来暴露这些指标，Kubernetes 提供了一个 [kube-state-metrics](#) 就是我们需要的。

kube-state-metrics 已经给出了在 Kubernetes 部署的 manifest 定义文件，我们直接将代码 Clone 到集群中(能用 kubectl 工具操作就行):

```
$ git clone https://github.com/kubernetes/kube-state-metrics.git
$ cd kube-state-metrics/kubernetes
$ kubectl create -f .
clusterrolebinding.rbac.authorization.k8s.io "kube-state-metrics" created
clusterrole.rbac.authorization.k8s.io "kube-state-metrics" created
deployment.apps "kube-state-metrics" created
rolebinding.rbac.authorization.k8s.io "kube-state-metrics" created
role.rbac.authorization.k8s.io "kube-state-metrics-resizer" created
serviceaccount "kube-state-metrics" created
service "kube-state-metrics" created
```

将 kube-state-metrics 部署到 Kubernetes 上之后，就会发现 Kubernetes 集群中的 Prometheus 会在 kubernetes-service-endpoints 这个 job 下自动服务发现 kube-state-metrics，并开始拉取 metrics，这是因为部署 kube-state-metrics 的 manifest 定义文件 kube-state-metrics-service.yaml 对 Service 的定义包含 `prometheus.io/scrape: 'true'` 这样的一个 annotation，因此 kube-state-metrics 的 endpoint 可以被 Prometheus 自动服务发现。

关于 kube-state-metrics 暴露的所有监控指标可以参考 kube-state-metrics 的文档[kube-state-metrics Documentation](#)。

到这里我们就完成了 Kubernetes 集群上部署应用的监控，下节课我们再来和大家介绍下怎样使用 Grafana 来展示我们的这些监控数据。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:05:46

56. Grafana 的安装使用

前面的课程中我们使用 Prometheus 采集了 Kubernetes 集群中的一些监控数据指标，我们也尝试使用 promQL 语句查询出了一些数据，并且在 Prometheus 的 Dashboard 中进行了展示，但是明显可以感觉到 Prometheus 的图表功能相对较弱，所以一般情况下我们会一个第三方的工具来展示这些数据，今天我们要和大家使用到的就是 grafana。

安装

grafana 是一个可视化面板，有着非常漂亮的图表和布局展示，功能齐全的度量仪表盘和图形编辑器，支持 Graphite、zabbix、InfluxDB、Prometheus、OpenTSDB、Elasticsearch 等作为数据源，比 Prometheus 自带的图表展示功能强大太多，更加灵活，有丰富的插件，功能更加强大。

接下来我们就来直接安装，同样的，我们将 grafana 安装到 Kubernetes 集群中，第一步同样是去查看 grafana 的 docker 镜像的介绍，我们可以在 dockerhub 上去搜索，也可以在官网去查看相关资料，镜像地址如下：<https://hub.docker.com/r/grafana/grafana/>，我们可以看到介绍中运行 grafana 容器的命令非常简单：

```
$ docker run -d --name=grafana -p 3000:3000 grafana/grafana
```

但是还有一个需要注意的是 Changelog 中 v5.1.0 版本的更新介绍：

- Major restructuring of the container
- Usage of chown removed
- File permissions incompatibility with previous versions
 - user id changed from 104 to 472
 - group id changed from 107 to 472
- Runs as the grafana user by default (instead of root)
- All default volumes removed

特别需要注意第3条，userid 和 groupid 都有所变化，所以我们在运行的容器的时候需要注意这个变化。现在我们将这个容器转化成 Kubernetes 中的 Pod：(grafana-deploy.yaml)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: grafana
  namespace: kube-ops
  labels:
    app: grafana
spec:
  revisionHistoryLimit: 10
  template:
    metadata:
      labels:
        app: grafana
    spec:
```

```

containers:
- name: grafana
  image: grafana/grafana:5.3.4
  imagePullPolicy: IfNotPresent
  ports:
  - containerPort: 3000
    name: grafana
  env:
  - name: GF_SECURITY_ADMIN_USER
    value: admin
  - name: GF_SECURITY_ADMIN_PASSWORD
    value: admin321
  readinessProbe:
    failureThreshold: 10
    httpGet:
      path: /api/health
      port: 3000
      scheme: HTTP
    initialDelaySeconds: 60
    periodSeconds: 10
    successThreshold: 1
    timeoutSeconds: 30
  livenessProbe:
    failureThreshold: 3
    httpGet:
      path: /api/health
      port: 3000
      scheme: HTTP
    periodSeconds: 10
    successThreshold: 1
    timeoutSeconds: 1
  resources:
    limits:
      cpu: 100m
      memory: 256Mi
    requests:
      cpu: 100m
      memory: 256Mi
  volumeMounts:
  - mountPath: /var/lib/grafana
    subPath: grafana
    name: storage
  securityContext:
    fsGroup: 472
    runAsUser: 472
  volumes:
  - name: storage
    persistentVolumeClaim:
      claimName: grafana

```

我们使用了最新的镜像 `grafana/grafana:5.3.4`，然后添加了监控检查、资源声明，另外两个比较重要的环境变量 `GF_SECURITY_ADMIN_USER` 和 `GF_SECURITY_ADMIN_PASSWORD`，用来配置 grafana 的管理员用户和密码的，由于 grafana 将 dashboard、插件这些数据保存在 `/var/lib/grafana` 这个目录下面的，所以我们这里如果需要做数据持久化的话，就需要针对这个目录进行 volume 挂载声明，其他的和我们之前的 Deployment 没什么区别，由于上面我们刚刚提到的 Changelog 中 grafana 的 userid 和 groupid 有所变化，所以我们这里需要增加一个 `securityContext` 的声明来进行声明。

当然如果要使用一个 pvc 对象来持久化数据，我们就需要添加一个可用的 pv 供 pvc 绑定使用：
(grafana-volume.yaml)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: grafana
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    server: 10.151.30.57
    path: /data/k8s
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: grafana
  namespace: kube-ops
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

最后，我们需要对外暴露 grafana 这个服务，所以我们需要一个对应的 Service 对象，当然用 NodePort 或者再建立一个 ingress 对象都是可行的：(grafana-svc.yaml)

```
apiVersion: v1
kind: Service
metadata:
  name: grafana
  namespace: kube-ops
  labels:
    app: grafana
spec:
  type: NodePort
  ports:
    - port: 3000
  selector:
    app: grafana
```

现在我们直接创建上面的这些资源对象：

```
$ kubectl create -f grafana-volume.yaml
persistentvolume "grafana" created
persistentvolumeclaim "grafana" created
$ kubectl create -f grafana-deploy.yaml
deployment.extensions "grafana" created
$ kubectl create -f grafana-svc.yaml
service "grafana" created
```

创建完成后，我们可以查看 grafana 对应的 Pod 是否正常：

```
$ kubectl get pods -n kube-ops
NAME                  READY   STATUS            RESTARTS   AGE
grafana-5f7b965b55-wxvvk   0/1    CrashLoopBackOff   1          22s
```

我们可以看到这里的状态是 CrashLoopBackOff，并没有正常启动，我们查看下这个 Pod 的日志：

```
$ kubectl logs -f grafana-5f7b965b55-wxvvk -n kube-ops
GF_PATHS_DATA='/var/lib/grafana' is not writable.
You may have issues with file permissions, more information here: http://docs.grafana.org/
installation/docker/#migration-from-a-previous-version-of-the-docker-container-to-5-1-or-later
mkdir: cannot create directory '/var/lib/grafana/plugins': Permission denied
```

上面的错误是在 5.1 版本之后才会出现的，当然你也可以使用之前的版本来规避这个问题。

可以看到是日志中错误很明显就是 /var/lib/grafana 目录的权限问题，这还是因为5.1版本后 groupid 更改了引起的问题，我们这里增加了 securityContext，但是我们将目录 /var/lib/grafana 挂载到 pvc 这边后目录的拥有者并不是上面的 grafana(472)这个用户了，所以我们需要更改下这个目录的所属用户，这个时候我们可以利用一个 Job 任务去更改下该目录的所属用户：(grafana-chown-job.yaml)

```
apiVersion: batch/v1
kind: Job
metadata:
  name: grafana-chown
  namespace: kube-ops
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: grafana-chown
          command: ["chown", "-R", "472:472", "/var/lib/grafana"]
          image: busybox
          imagePullPolicy: IfNotPresent
          volumeMounts:
            - name: storage
              subPath: grafana
              mountPath: /var/lib/grafana
      volumes:
        - name: storage
          persistentVolumeClaim:
            claimName: grafana
```

上面我们利用一个 busybox 镜像将 /var/lib/grafana 目录更改成了 472 这个 user 和 group，不过还需要注意的是下面的 volumeMounts 和 volumes 需要和上面的 Deployment 对应上。

现在我们删除之前创建的 Deployment 对象，重新创建：

```
$ kubectl delete -f grafana-deploy.yaml
deployment.extensions "grafana" deleted
```

```
$ kubectl create -f grafana-deploy.yaml
deployment.extensions "grafana" created
$ kubectl create -f grafana-chown-job.yaml
job.batch "grafana-chown" created
```

重新执行完成后，可以查看下上面的创建的资源对象是否正确了：

```
$ kubectl get pod -n kube-ops
NAME                  READY   STATUS    RESTARTS   AGE
grafana-79477fbb7c-2mb84   1/1     Running   0          2m
grafana-chown-k8zt7      0/1     Completed  0          2m
```

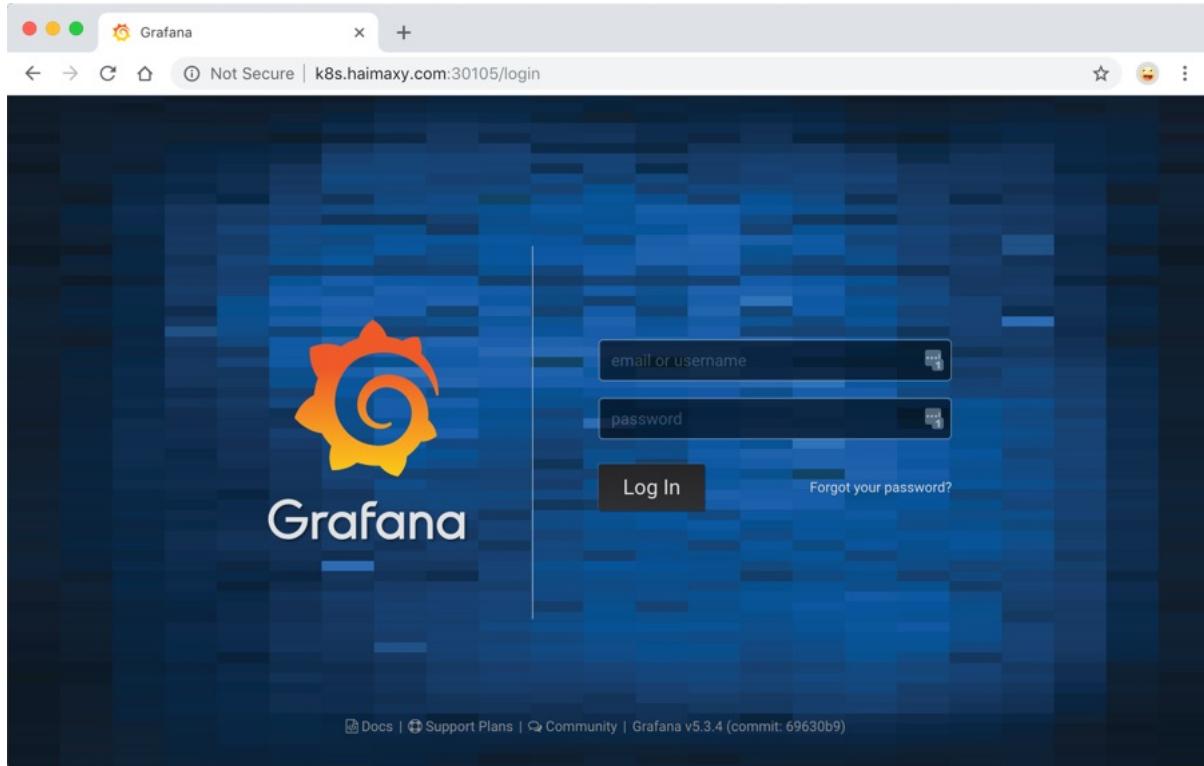
我们可以看到有一个状态为 `Completed` 的 Pod，这就是上面我们用来更改 `grafana` 目录权限的 Pod，是一个 Job 任务，所以执行成功后就退出了，状态变成了 `Completed`，而上面的 `grafana` 的 Pod 也已经是 `Running` 状态了，可以查看下该 Pod 的日志确认下：

```
$ kubectl logs -f grafana-79477fbb7c-2mb84 -n kube-ops
t=2018-11-14T19:57:31+0000 lvl=info msg="Starting Grafana" logger=server version=5.3.4 commit=69630b9 compiled=2018-11-13T12:19:12+0000
...
logger=settings var="GF_SECURITY_ADMIN_USER=admin"
t=2018-11-14T19:57:31+0000 lvl=info msg="Config overridden from Environment variable"
...
t=2018-11-14T19:57:32+0000 lvl=info msg="Initializing Stream Manager"
t=2018-11-14T19:57:32+0000 lvl=info msg="HTTP Server Listen" logger=http.server address=0.0.0.0:3000 protocol=http subUrl= socket=
```

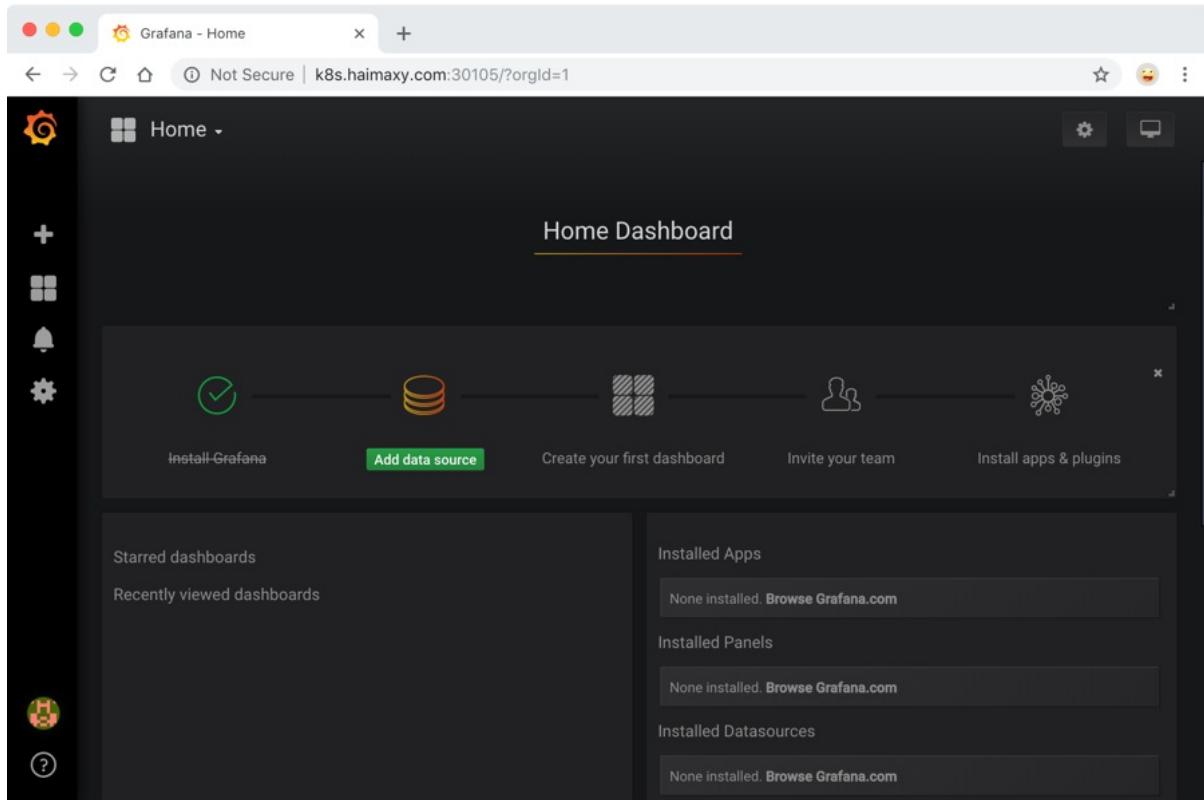
看到上面的日志信息就证明我们的 `grafana` 的 Pod 已经正常启动起来了。这个时候我们可以查看 Service 对象：

```
$ kubectl get svc -n kube-ops
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
grafana   NodePort  10.97.46.27    <none>        3000:30105/TCP 1h
```

现在我们就可以在浏览器中使用 `http://<30105>` 来访问 grafana 这个服务了：



由于上面我们配置了管理员的，所以第一次打开的时候会跳转到登录界面，然后就可以用上面我们配置的两个环境变量的值来进行登录了，登录完成后就可以进入到下面 Grafana 的首页：



配置

在上面的首页中我们可以看到已经安装了 Grafana，接下来点击 `Add data source` 进入添加数据源界面。

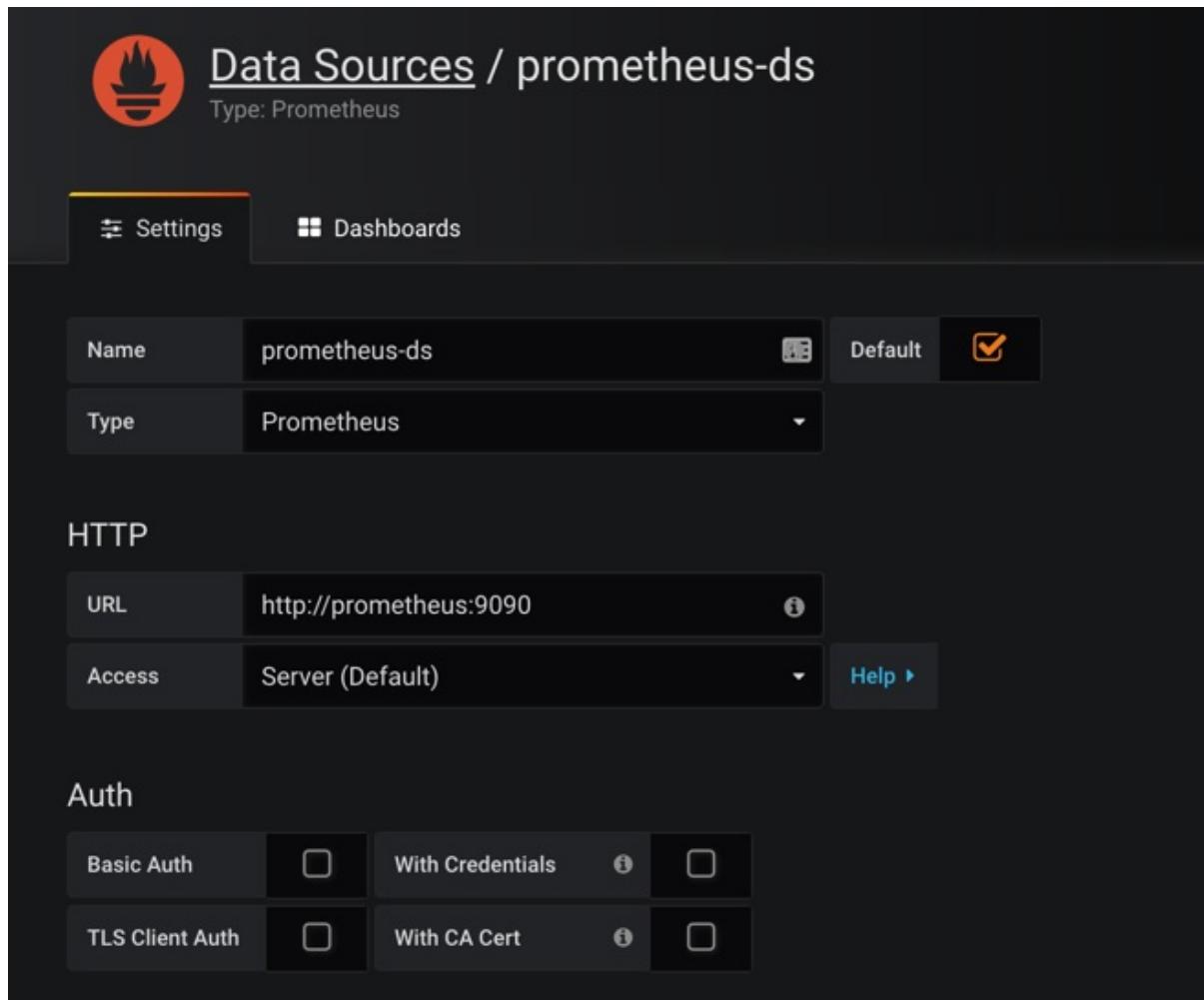
数据源

我们这个地方配置的数据源是 Prometheus，所以选择这个 Type 即可，给改数据源添加一个 name: `prometheus-ds`，最主要的是下面 `HTTP` 区域是配置数据源的访问模式。

访问模式是用来控制如何处理对数据源的请求的：

- 服务器(Server)访问模式（默认）：所有请求都将从浏览器发送到 Grafana 后端的服务器，后者又将请求转发到数据源，通过这种方式可以避免一些跨域问题，其实就是在 Grafana 后端做了一次转发，需要从 Grafana 后端服务器访问该 URL。
- 浏览器(Browser)访问模式：所有请求都将从浏览器直接发送到数据源，但是有可能会有一些跨域的限制，使用此访问模式，需要从浏览器直接访问该 URL。

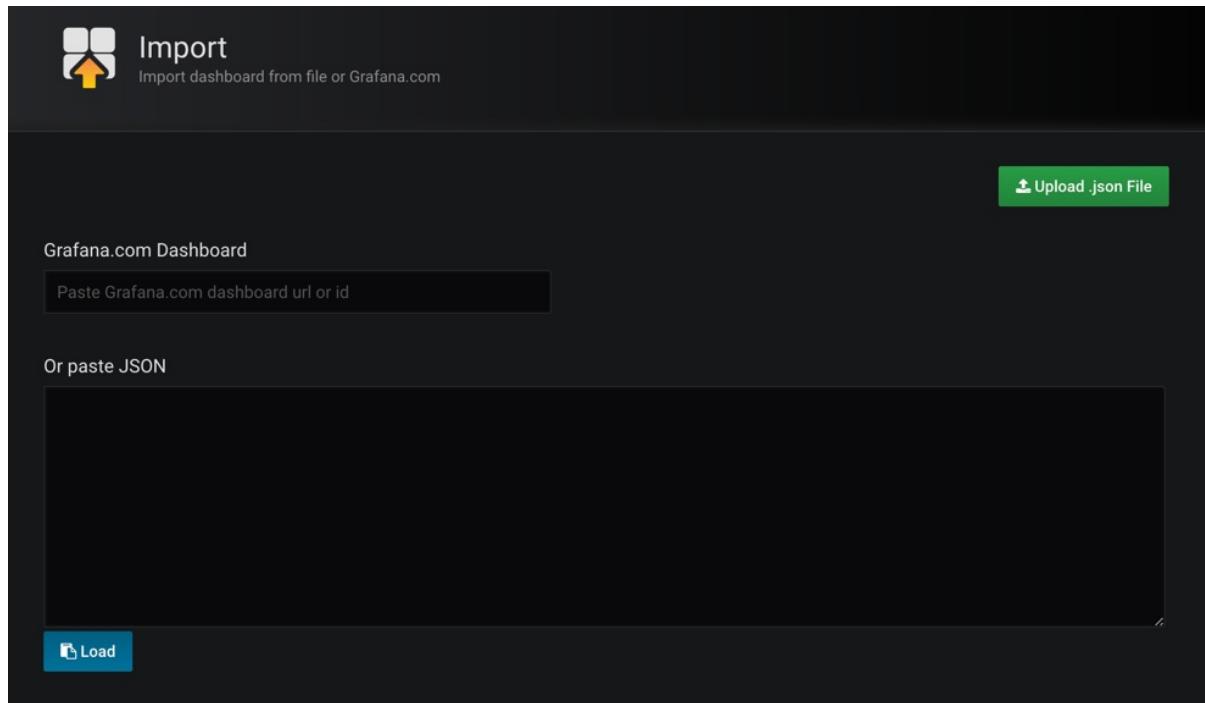
由于我们这个地方 Prometheus 通过 NodePort 的方式的对外暴露的服务，所以我们这个地方是不是可以使用浏览器访问模式直接访问 Prometheus 的外网地址，但是这种方式显然不是最好的，相当于走的是外网，而我们这里 Prometheus 和 Grafana 都处于 `kube-ops` 这同一个 namespace 下面，是不是在集群内部直接通过 DNS 的形式就可以访问了，而且还都是走的内网流量，所以我们这里用服务器访问模式显然更好，数据源地址：`http://prometheus:9090`（因为在同一个 namespace 下面所以直接用 Service 名也可以），然后其他的配置信息就根据实际情况了，比如 Auth 认证，我们这里没有，所以跳过即可，点击最下方的 `Save & Test` 提示成功证明我们的数据源配置正确：



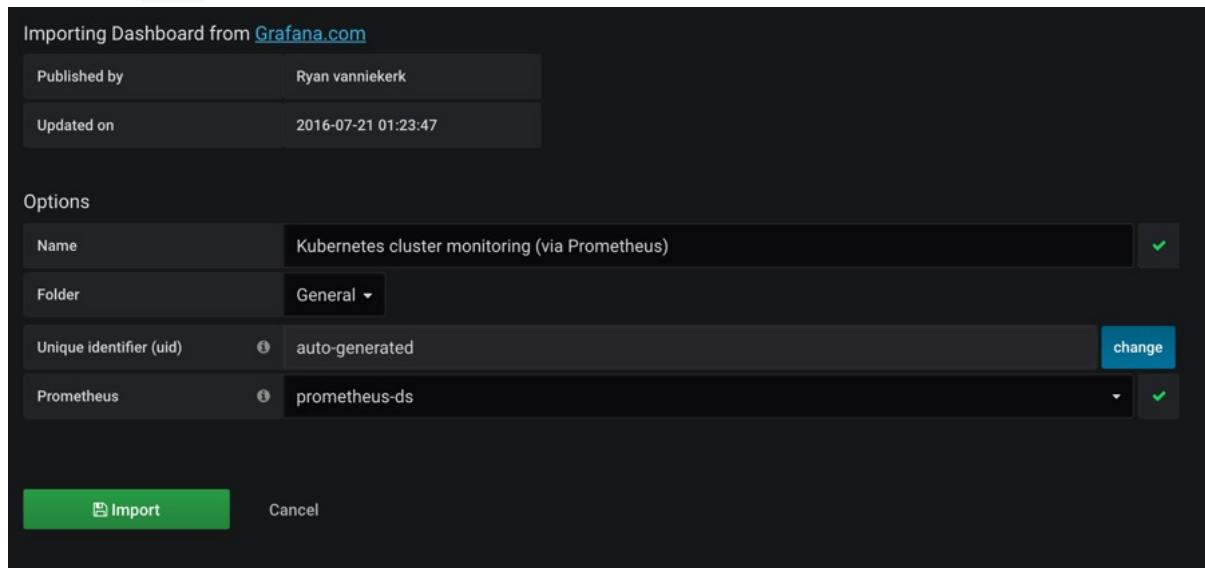
grafana datasource

数据源添加完成后，就可以来添加 Dashboard 了。

Dashboard



我们可以将上面编号 162 的 dashboard 下载到本地，然后这里重新上传即可，也可以在上面的文本框中直接输入 162 编号回车即可，导入这个 dashboard：



需要注意的是在执行上面的 import 之前要记得选择我们的 prometheus-ds 这个名字的数据源，执行 import 操作，就可以进入到 dashboard 页面：



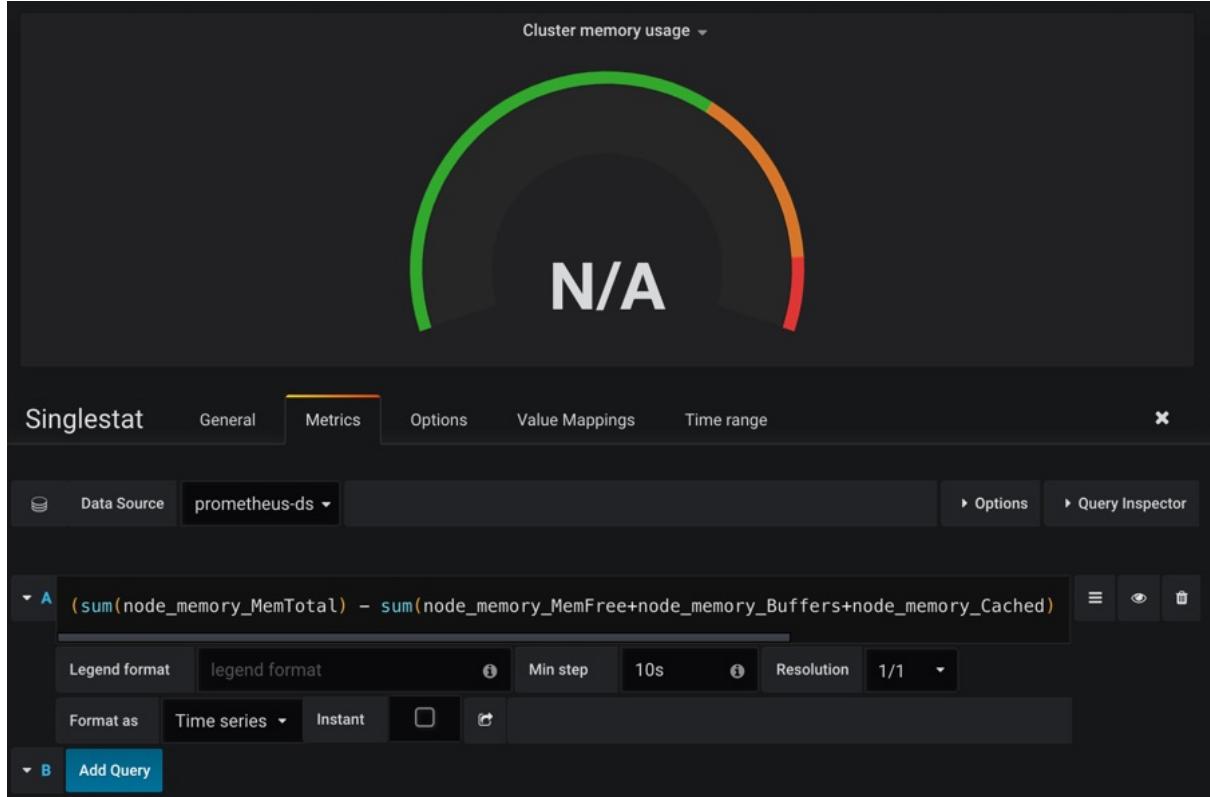
我们可以看到 dashboard 页面上出现了很多漂亮的图表，但是看上去数据不正常，这是因为这个 dashboard 里面需要的数据指标名称和我们 Prometheus 里面采集到的数据指标不一致造成的，比如，第一个 Cluster memory usage(集群内存使用情况)，我们可以点击标题 -> Edit，进入编辑这个图表



的编辑页面：

进入编辑页面我们就可以看到这个图表的查询语句：

```
(sum(node_memory_MemTotal) - sum(node_memory_MemFree+node_memory_Buffers+node_memory_Cached)) / sum(node_memory_MemTotal) * 100
```

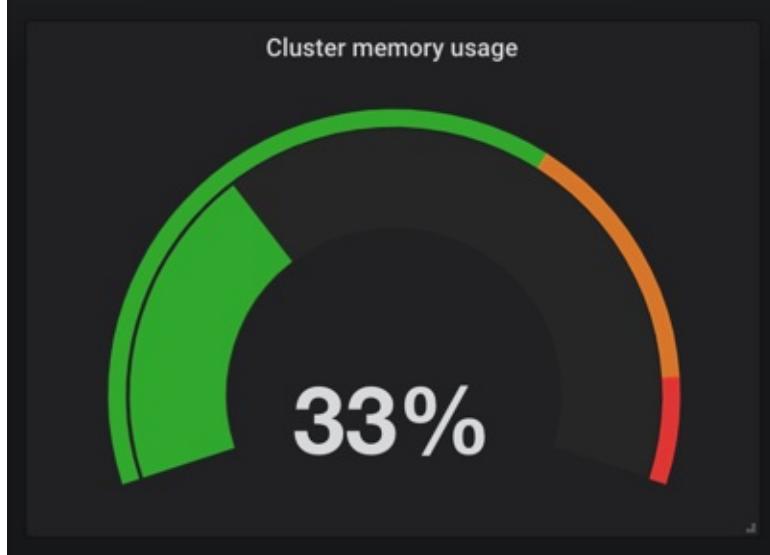


grafana dashboard edit2

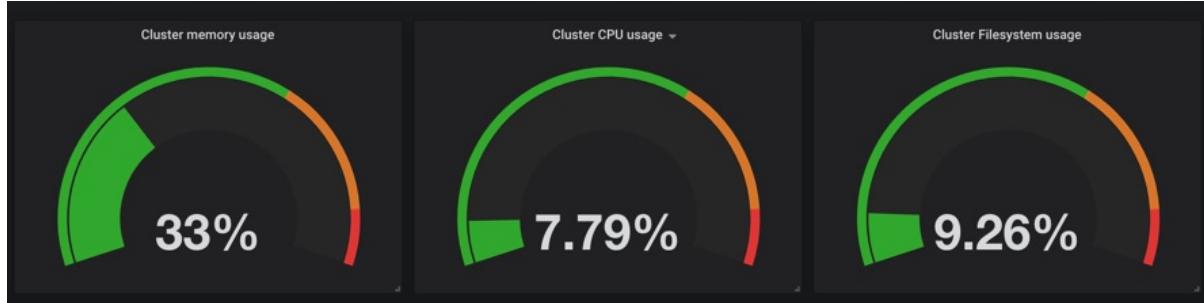
这就是我们之前在 Prometheus 里面查询的 `promQL` 语句，我们可以将上面的查询语句复制到 Prometheus 的 Graph 页面进行查询，其实可以预想到是没有对应的数据的，因为我们用 `node_exporter` 采集到的数据指标不是 `node_memory_MemTotal` 关键字，而是 `node_memory_MemTotal_bytes`，将上面的 `promQL` 语句做相应的更改：

```
(sum(node_memory_MemTotal_bytes) - sum(node_memory_MemFree_bytes + node_memory_Buffers_bytes+node_memory_Cached_bytes)) / sum(node_memory_MemTotal_bytes) * 100
```

这个语句的意思就是 (整个集群的内存 - (整个集群剩余的内存以及Buffer和Cached)) / 整个集群的内存，简单来说就是总的集群内存使用百分比。将上面 grafana 的 promQL 语句替换掉，就可以看到图表正常了：



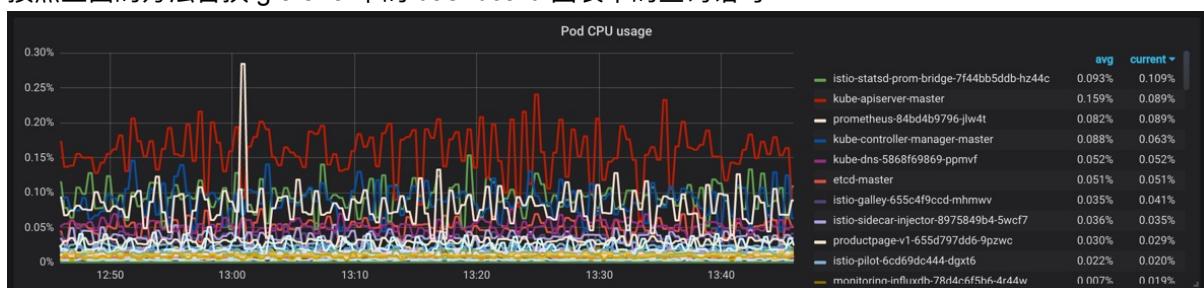
同样的，我们可以更改后面的 CPU 和 FileSystem 的使用率：



同样下面的 Pod CPU Usage 用来展示 Pod CPU 的使用情况，对应的 promQL 语句如下，根据 pod_name 来进行统计：

```
sum by (pod_name)(rate(container_cpu_usage_seconds_total{image!="", pod_name!=""}[1m]))
```

按照上面的方法替换 grafana 中的 dashboard 图表中的查询语句：



其他的也按照我们的实际需求重新编辑下就可以，下图是最终整个 dashboard 的效果图：



最后要记得保存这个 dashboard，下面的链接是我修改后的 dashboard json 文件地址，你可以直接下载下来导入到 grafana 当中，当然你也可以根据实际情况进行相应的更改：[k8s-cluster-grafana-dashboard.json](#)。

除此之外，我们也可以前往 grafana dashboard 的页面去搜索其他的关于 Kubernetes 的监控页面，地址：<https://grafana.com/dashboards>，比如id 为747和741的这两个 dashboard。

插件

上面是我们最常用的 grafana 当中的 dashboard 的功能的使用，然后我们也可以来进行一些其他的系统管理，比如添加用户，为用户添加权限等等，我们也可以安装一些其他插件，比如 grafana 就有一个专门针对 Kubernetes 集群监控的插件：[grafana-kubernetes-app](#)

要安装这个插件，需要到 grafana 的 Pod 里面去执行安装命令：

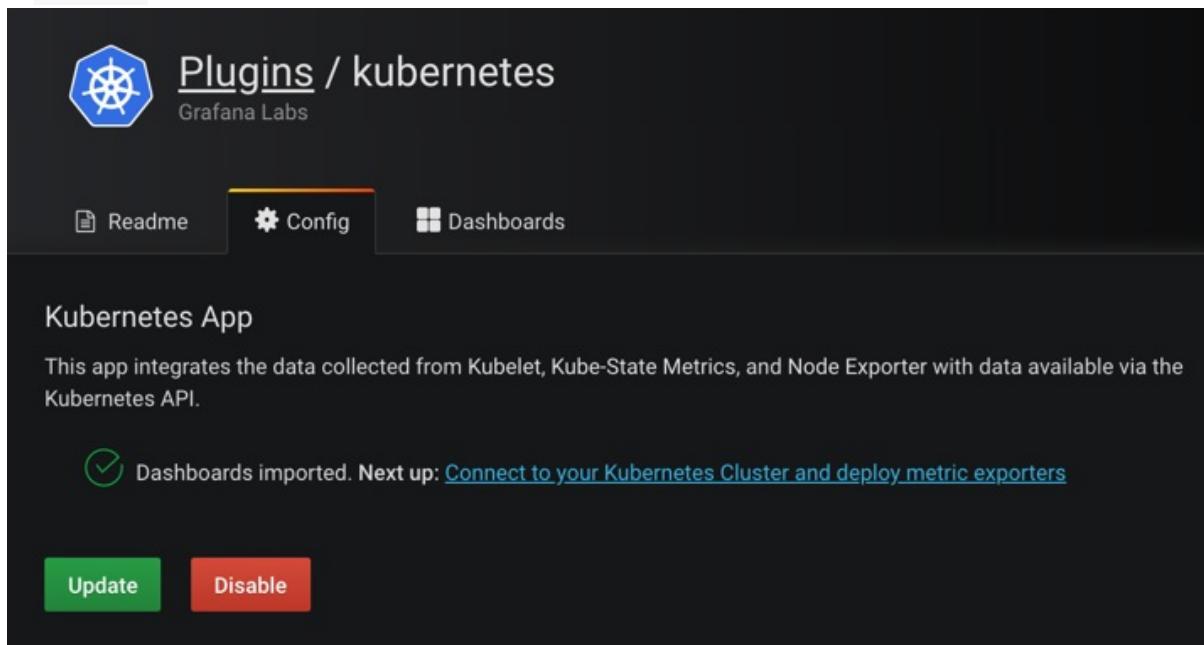
```
$ kubectl get pods -n kube-ops
NAME                    READY   STATUS    RESTARTS   AGE
grafana-79477fbb7c-v4prs   1/1    Running   0          23m
$ kubectl exec -it grafana-79477fbb7c-v4prs /bin/bash -n kube-ops
```

```
grafana@grafana-79477fbb7c-v4prs:/usr/share/grafana$ grafana-cli plugins install grafana-kubernetes-app
installing grafana-kubernetes-app @ 1.0.1
from url: https://grafana.com/api/plugins/grafana-kubernetes-app/versions/1.0.1/download
into: /var/lib/grafana/plugins

✓ Installed grafana-kubernetes-app successfully

Restart grafana after installing plugins . <service grafana-server restart>
grafana@grafana-79477fbb7c-v4prs:/usr/share/grafana$
```

安装完成后需要重启 grafana 才会生效，我们这里直接删除 Pod，重建即可，然后回到 grafana 页面中，切换到 plugins 页面可以发现下面多了一个 Kubernetes 的插件，点击进来启用即可，然后点击 Next up 旁边的链接配置集群



这里我们可以添加一个新的 Kubernetes 集群，这里需要填写集群的访问地址：`https://kubernetes.default`，然后比较重要的是集群访问的证书，勾选上 `TLS Client Auth` 和 `With CA Cert` 这两项。

Add a new cluster

Name: ydzs-cluster

HTTP

URL	https://kubernetes.default	?
Access	Server (Default)	Help ▾

Auth

Basic Auth	<input type="checkbox"/>	With Credentials	<input type="checkbox"/>
TLS Client Auth	<input checked="" type="checkbox"/>	With CA Cert	<input checked="" type="checkbox"/>

Skip TLS Verification (Insecure)

TLS Auth Details [?](#)

CA Cert	configured	reset
Client Cert	configured	reset
Client Key	configured	reset

集群访问的证书文件，用我们访问集群的 kubectl 的配置文件中的证书信息(~/.kube/config)即可，其中属性 `certificate-authority-data`、`client-certificate-data`、`client-key-data` 就对应这 CA 证书、Client 证书、Client 私钥，不过 config 文件里面的内容是 base64 编码过后的，所以我们这里填写的时候要做 base64 解码。

另外需要将解码过后的 \n 换成换行符，不然认证会失败。

配置完成后，可以直接点击 Deploy (实际上前面的课程中我们都已经部署过相关的资源了)，然后点击 Save，就可以获取到集群的监控资源信息了。

The screenshot shows the Grafana interface for monitoring a Kubernetes cluster. At the top, there's a header with the Kubernetes logo and the text "kubernetes / Cluster Info". Below the header, there's a section titled "Overview" with three cards:

- Cluster Dashboard**: A high level view of cluster health metrics.
- Nodes Dashboard**: CPU, Memory, Disk Usage and Network metrics per node.
- Pod/Container Dashboard**: Pod/Container metrics - CPU, Memory or Network stats.

On the right side of the overview, it says "Cluster: ydzs-cluster".

Below the overview, there's a section titled "Browse Details from the Kubernetes API" with a sub-section "Namespaces" (click on a namespace to see its pods and deployments). It lists several namespaces:

- blog (Active)
- default (Active)
- istio-system (Active)
- kube-ops (Active)
- kube-public (Active)
- kube-system (Active)

Under "Component Statuses", there are three items:

- controller-manager (OK)
- scheduler (OK)
- etcd-0 (OK)

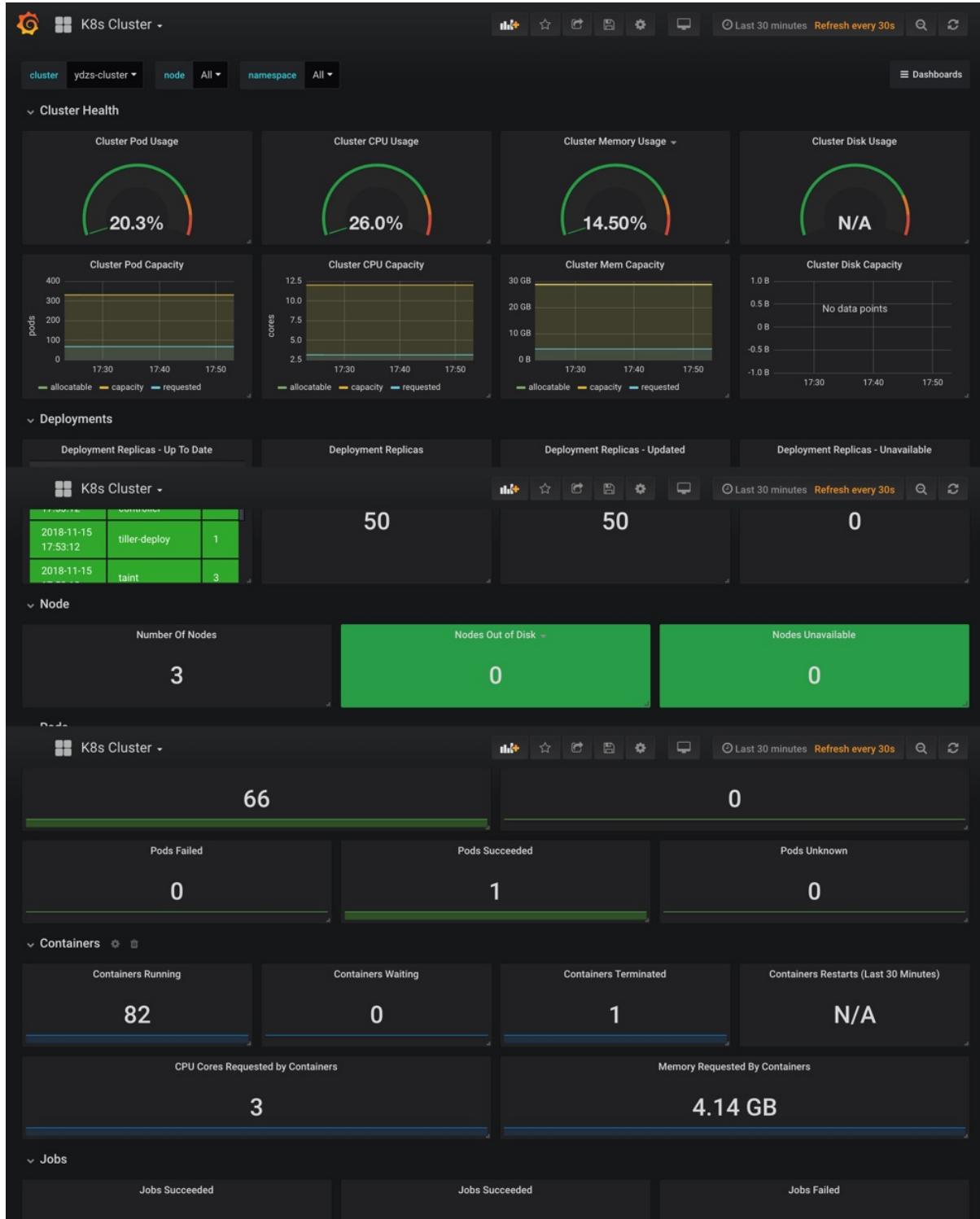
Under "Nodes" (click to see node details), there are three nodes listed:

- master (OK) with a "Node Stats Dashboard" link
- node02 (OK) with a "Node Stats Dashboard" link
- node03 (OK) with a "Node Stats Dashboard" link

At the bottom of the dashboard, there are links to "Docs", "Support Plans", "Community", and "Grafana v5.3.4 (commit: 69630b9)".

grafana k8s plugins

可以看到上面展示了整个集群的状态，可以查看上面的一些 Dashboard:



报警

grafana 4 版本以上就支持了报警功能，这使得我们利用 grafana 作为监控面板更为完整，因为报警是监控系统中必不可少的环节，grafana 支持很多种形式的报警功能，比如 email、钉钉、slack、webhook 等等，我们这里来测试下 email 和 钉钉。

email 报警

要启用 email 报警需要在启动配置文件中 `/etc/grafana/grafana.ini` 开启 SMTP 服务，我们这里同样利用一个 ConfigMap 资源对象挂载到 grafana Pod 中：（`grafana-cm.yaml`）

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-config
  namespace: kube-ops
data:
  grafana.ini: |
    [server]
    root_url = http://<你grafana的url地址>
    [smtp]
    enabled = true
    host = smtp.163.com:25
    user = ych_1024@163.com
    password = <邮箱密码>
    skip_verify = true
    from_address = ych_1024@163.com
    [alerting]
    enabled = true
    execute_alerts = true
```

上面配置了我的 163 邮箱，开启报警功能，当然我们还得将这个 ConfigMap 文件挂载到 Pod 中去：

```
volumeMounts:
- mountPath: "/etc/grafana"
  name: config
volumes:
- name: config
  configMap:
    name: grafana-config
```

创建 ConfigMap 对象，更新 Deployment：

```
$ kubectl create -f grafana-cm.yaml
$ kubectl apply -f grafana-deploy.yaml
```

更新完成后，在 grafana 的 webui 中 Alert 页面测试邮件报警：

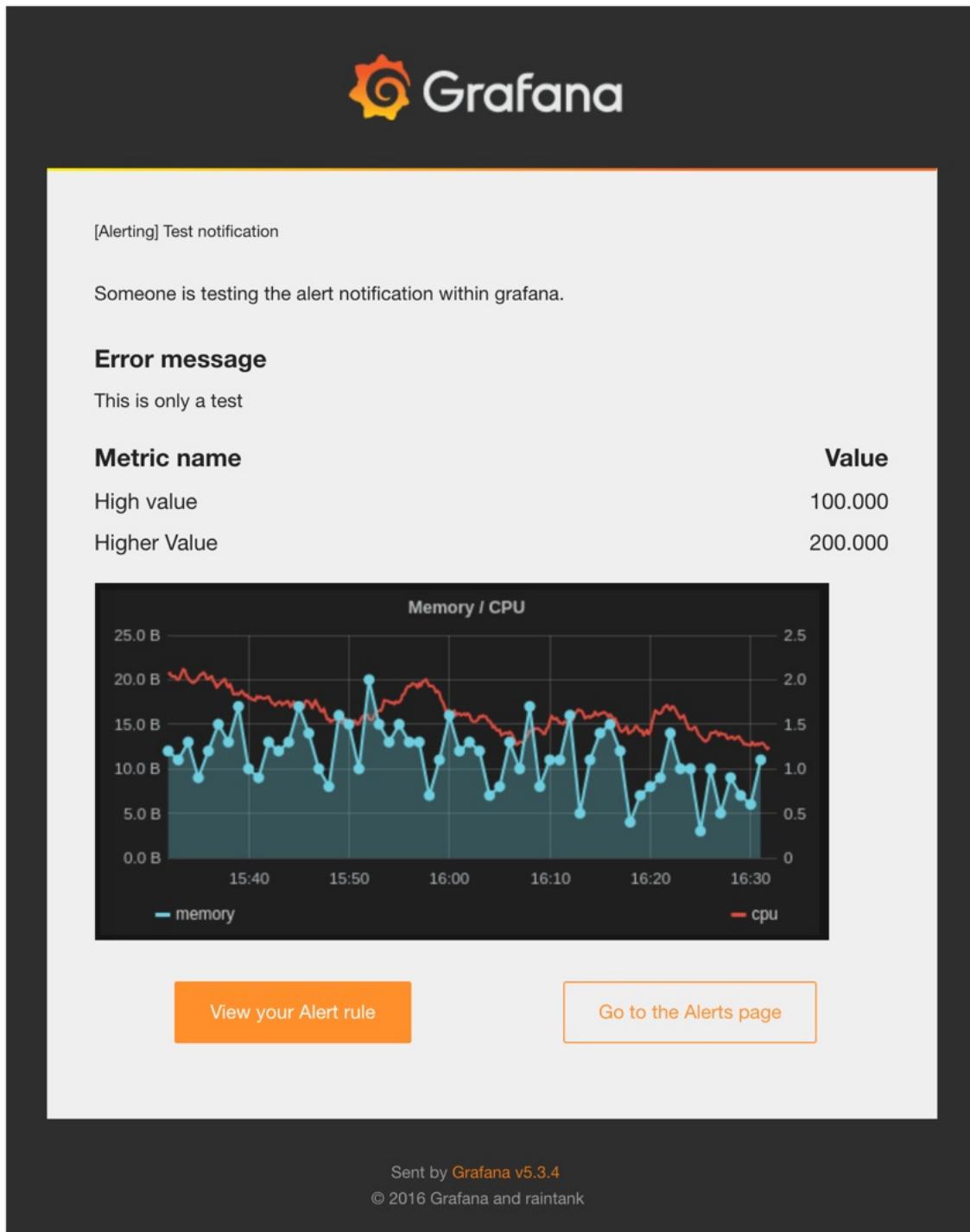
The screenshot shows the Grafana Alerting interface. At the top, there is a bell icon and the title "Alerting" followed by "Alert rules & notifications". Below this, there are two tabs: "Alert Rules" and "Notification channels", with "Notification channels" being the active tab. A sub-header "New Notification Channel" is displayed. The configuration form contains the following fields:

Name	test
Type	Email
Send on all alerts	<input checked="" type="checkbox"/>
Include image	<input checked="" type="checkbox"/>
Send reminders	<input type="checkbox"/>

Below the configuration form, there is a section titled "Email addresses" containing the email address "517554016@qq.com". A note below the input field says "You can enter multiple email addresses using a ";" separator". At the bottom of the screen, there are three buttons: "Save" (green), "Send Test" (blue), and "Back" (grey).

grafana email alert

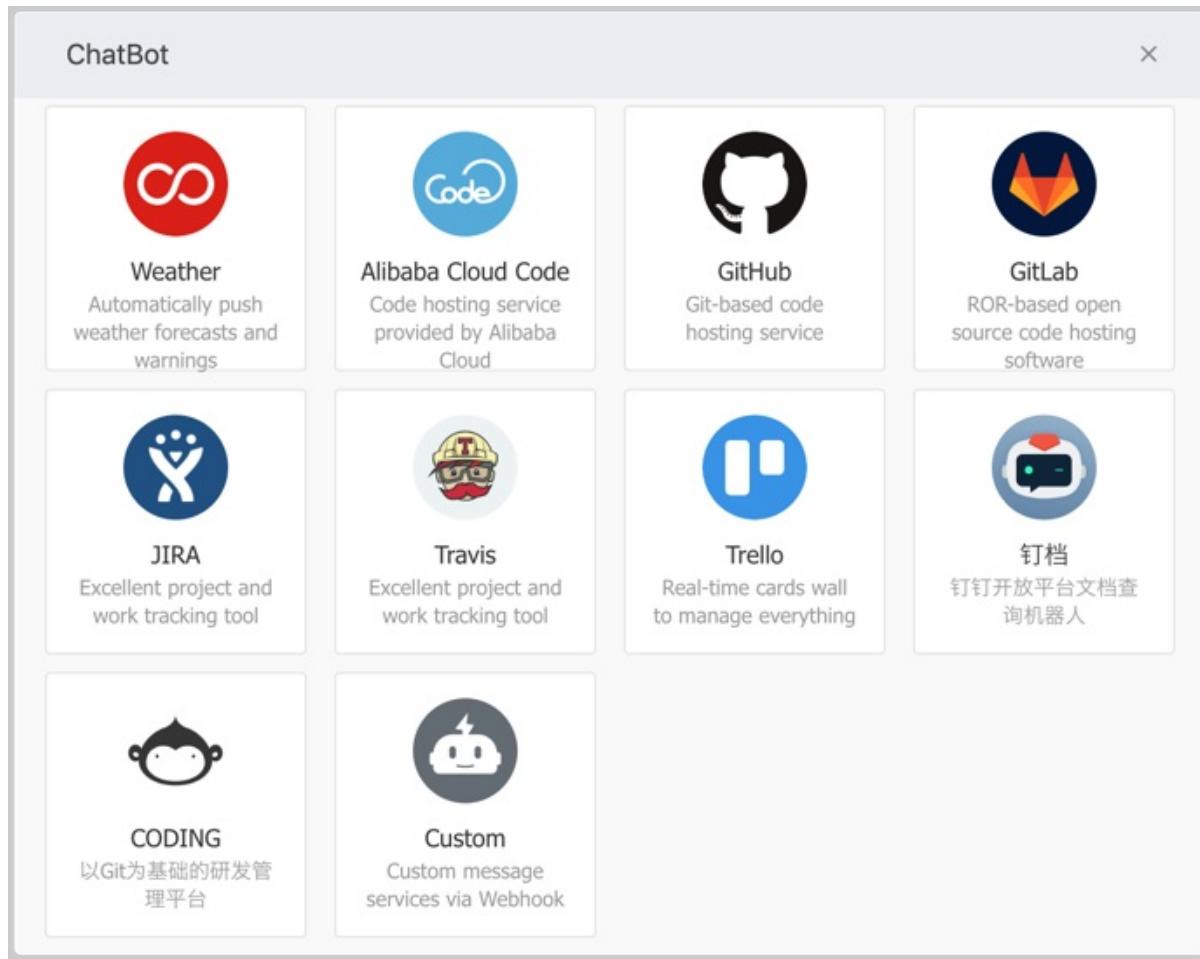
发送测试后，正常情况下就可以收到测试报警邮件：



grafana alert email

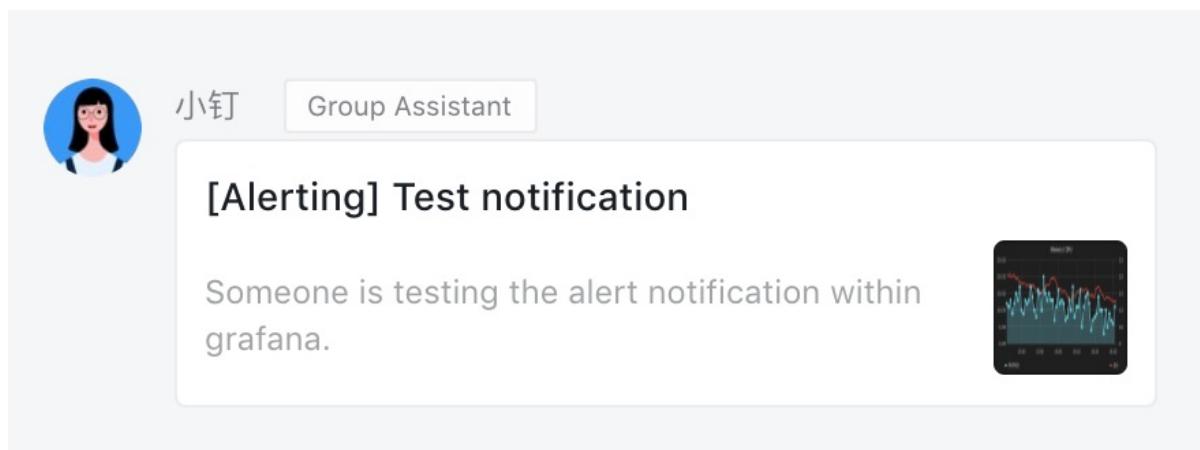
钉钉报警

上面我们也说了 grafana 也是支持钉钉报警的，在钉钉群里面添加群机器人，选择最后的自定义机器人：



grafana add dingtalk robot

添加完成后可以得到一个 webhook 的地址，然后将这个 webhook 地址添加到上面 grafana webui 的报警测试页面进行测试，就可以在钉钉群里面收到报警测试信息了：



grafana dingtalk alert

配置

目前只有 Graph 支持报警功能，所以我们选择 Graph 相关图表，点击编辑，进入 Graph 编辑页面可以看到有一个 Alert 模块，切换过来创建报警：

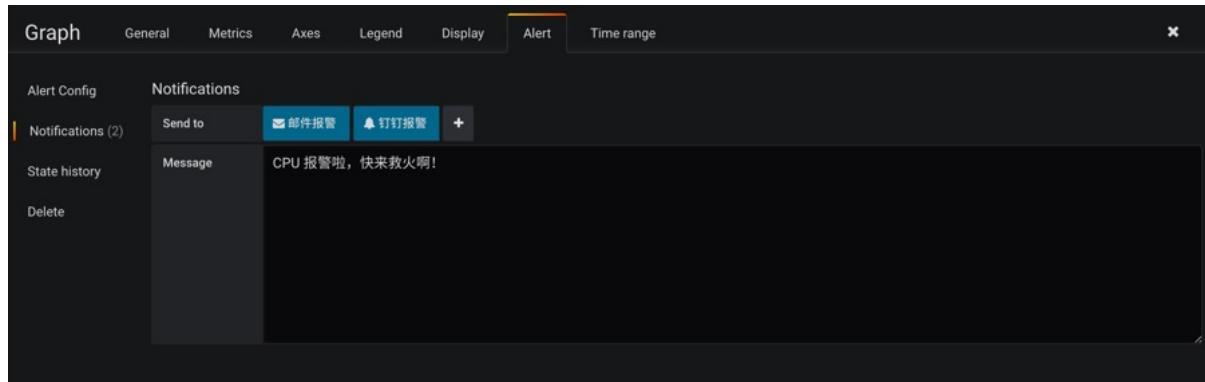


grafana graph alert

然后配置相关参数：

- 1、Alert 名称，可以自定义。
- 2、执行的频率，这里我选择每60s检测一次。
- 3、判断标准，默认是 avg，这里是下拉框，自己按需求选择。
- 4、query (A,5m,now) ，字母A代表选择的metrics 中设置的 sql，也可以选择其它在 metrics中设置的，但这里是单选。`5m` 代表从现在起往之前的五分钟，即 `5m` 之前的那个点为时间的起始点，`now` 为时间的结束点，此外这里可以自己手动输入时间。
- 5、设置的预警临界点，这里手动输入，和6是同样功能，6可以手动移动，两种操作是等同的。

然后需要设置报警发送信息，点击侧边的 `Notifications`：



grafana graph notify

其中 `Send to` 就是前面我们配置过的发送邮件和钉钉的报警频道的名称。

配置完成后需要保存下这个 `graph`, 否则发送报警可能会失败, 然后点击 Alert 区域的 `Test Rule` 可以来测试报警规则, 然后邮件和钉钉正常来说就可以收到报警信息了。

邮件报警信息：

The screenshot shows the Grafana alerting interface. At the top, the Grafana logo is visible. Below it, a title reads "[Alerting] Pod CPU usage alert". A message states "CPU报警啦，快来救火啊！". A table lists the metric names and their values:

Metric name	Value
etcd-master	0.053
istio-statsd-prom-bridge-7f44bb5ddb-hz44c	0.096
kube-apiserver-master	0.166
kube-controller-manager-master	0.094
kube-dns-5868f69869-ppmvf	0.050
prometheus-84bd4b9796-jlw4t	0.079

At the bottom, two buttons are present: "View your Alert rule" and "Go to the Alerts page".

grafana test rule email

钉钉报警信息：

The screenshot shows a DingTalk message. The header includes a profile picture, the name "小钉", "Group Assistant", and the time "12:07". The message content is identical to the one shown in the Grafana screenshot: "[Alerting] Pod CPU usage alert" and "CPU报警啦，快来救火啊!". To the right of the text is a blue square icon with a white chain symbol, and below it is a small ellipsis icon.

grafana test rule dingtalk

到这里就完成了使用 grafana 来展示 Kubernetes 集群的监控图表信息以及报警配置，但是我们明显可以感觉到 grafana 的优势在于图表的展示，报警功能有点弱，所以一般来说，在生产环境我们不会直接使用 grafana 的报警功能，更多的是使用功能更加强大的 AlertManager，下节课我们再来和大家介绍了。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



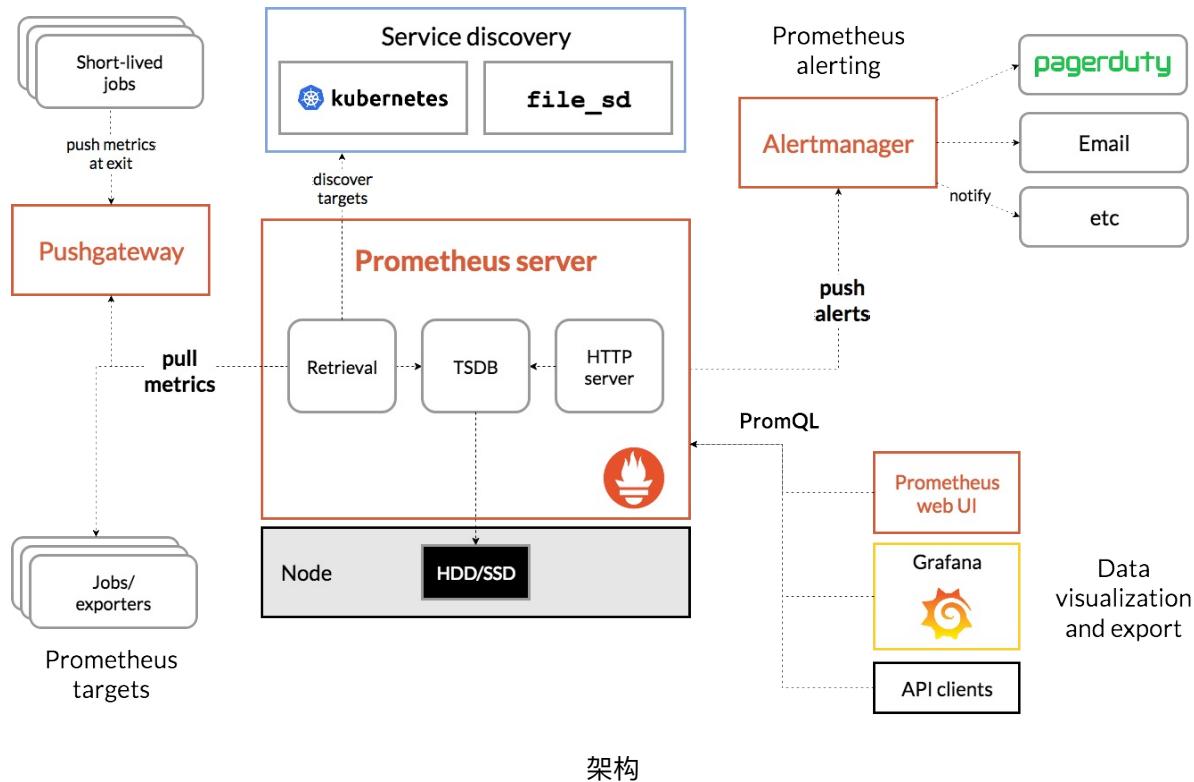
k8s技术圈二维码

57. AlertManager 的使用

上节课我们和大家一起学习了 Grafana 的使用，也测试了 Grafana 的报警功能，但是 Grafana 的报警功能目前还比较弱，只支持 Graph 的图表的报警。今天来给大家介绍一个功能更加强大的报警工具：AlertManager。

简介

之前我们学习 Prometheus 的时候就了解到 Prometheus 包含一个报警模块，就是我们的 AlertManager，Alertmanager 主要用于接收 Prometheus 发送的告警信息，它支持丰富的告警通知渠道，而且很容易做到告警信息进行去重，降噪，分组等，是一款前卫的告警通知系统。



接下来我们就来学习下 AlertManager 的具体使用方法。

安装

从官方文档<https://prometheus.io/docs/alerting/configuration/>中我们可以看到下载 AlertManager 二进制文件后，可以通过下面的命令运行：

```
$ ./alertmanager --config.file=simple.yml
```

其中 `-config.file` 参数是用来指定对应的配置文件的，由于我们这里同样要运行到 Kubernetes 集群中来，所以我们使用 `docker` 镜像的方式来安装，使用的镜像是：`prom/alertmanager:v0.15.3`。

首先，指定配置文件，同样的，我们这里使用一个 ConfigMap 资源对象：(`alertmanager-conf.yaml`)

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: alert-config
  namespace: kube-ops
data:
  config.yml: |-
    global:
      # 在没有报警的情况下声明为已解决的时间
      resolve_timeout: 5m
      # 配置邮件发送信息
      smtp_smarthost: 'smtp.163.com:25'
      smtp_from: 'ych_1024@163.com'
      smtp_auth_username: 'ych_1024@163.com'
      smtp_auth_password: '<邮箱密码>'
      smtp_hello: '163.com'
      smtp_require_tls: false
      # 所有报警信息进入后的根路由，用来设置报警的分发策略
    route:
      # 这里的标签列表是接收到报警信息后的重新分组标签，例如，接收到的报警信息里面有许多具有 cluster=A 和 alertname=LatencyHigh 这样的标签的报警信息将会批量被聚合到一个分组里面
      group_by: ['alertname', 'cluster']
      # 当一个新的报警分组被创建后，需要等待至少group_wait时间来初始化通知，这种方式可以确保您能有足够的时间为同一分组来获取多个警报，然后一起触发这个报警信息。
      group_wait: 30s
      # 当第一个报警发送后，等待'group_interval'时间来发送新的一组报警信息。
      group_interval: 5m
      # 如果一个报警信息已经发送成功了，等待'repeat_interval'时间来重新发送他们
      repeat_interval: 5m
      # 默认的receiver: 如果一个报警没有被一个route匹配，则发送给默认的接收器
      receiver: default
      # 上面所有的属性都由所有子路由继承，并且可以在每个子路由上进行覆盖。
    routes:
      - receiver: email
        group_wait: 10s
        match:
          team: node
    receivers:
      - name: 'default'
        email_configs:
          - to: '517554016@qq.com'
            send_resolved: true
      - name: 'email'
        email_configs:
          - to: '517554016@qq.com'
            send_resolved: true

```

这是 AlertManager 的配置文件，我们先直接创建这个 ConfigMap 资源对象：

```
$ kubectl create -f alertmanager-conf.yaml
configmap "alert-config" created
```

然后配置 AlertManager 的容器，我们可以直接在之前的 Prometheus 的 Pod 中添加这个容器，对应的 YAML 资源声明如下：

```
- name: alertmanager
  image: prom/alertmanager:v0.15.3
  imagePullPolicy: IfNotPresent
  args:
  - "--config.file=/etc/alertmanager/config.yml"
  ports:
  - containerPort: 9093
    name: http
  volumeMounts:
  - mountPath: "/etc/alertmanager"
    name: alertcfg
  resources:
  requests:
    cpu: 100m
    memory: 256Mi
  limits:
    cpu: 100m
    memory: 256Mi
  volumes:
  - name: alertcfg
    configMap:
      name: alert-config
```

这里我们将上面创建的 alert-config 这个 ConfigMap 资源对象以 Volume 的形式挂载到 /etc/alertmanager 目录下去，然后在启动参数中指定了配置文件 --config.file=/etc/alertmanager/config.yml，然后我们可以来更新这个 Prometheus 的 Pod：

```
$ kubectl apply -f prome-deploy.yaml
deployment.extensions "prometheus" configured
```

当然我们也可以将 AlertManager 的配置文件内容直接放入到之前的 Prometheus 的 ConfigMap 的资源对象中，也可以用一个单独的 Pod 来运行 AlertManager 这个容器，完整的资源清单文件可以参考这里：<https://github.com/cnchy/kubeapp/tree/master/prometheus>

AlertManager 的容器启动起来后，我们还需要在 Prometheus 中配置下 AlertManager 的地址，让 Prometheus 能够访问到 AlertManager，在 Prometheus 的 ConfigMap 资源清单中添加如下配置：

```
alerting:
  alertmanagers:
  - static_configs:
    - targets: ["localhost:9093"]
```

更新这个资源对象后，稍等一小会儿，执行 reload 操作：

```
$ kubectl delete -f prome-cm.yaml
configmap "prometheus-config" deleted
```

```
$ kubectl create -f prome-cm.yaml
configmap "prometheus-config" created
# 隔一会儿后
$ kubectl get svc -n kube-ops
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
prometheus   NodePort    10.102.74.90    <none>        9090:30358/TCP      3d
$ curl -X POST "http://10.102.74.90:9090/-/reload"
```

更新完成后，我们查看 Pod 发现有错误，查看下 alertmanager 容器的日志，发现有如下错误信息：

```
$ kubectl get pods -n kube-ops
NAME      READY     STATUS      RESTARTS   AGE
prometheus-56d64bf6f7-rpz9j   1/2      CrashLoopBackOff   491       1d
$ kubectl logs -f prometheus-56d64bf6f7-rpz9j alertmanager -n kube-ops
level=info ts=2018-11-28T10:33:51.830071513Z caller=main.go:174 msg="Starting Alertmanager"
version="(version=0.15.3, branch=HEAD, revision=d4a7697cc90f8bce62efe7c44b63b542578ec0a1)"

level=info ts=2018-11-28T10:33:51.830362309Z caller=main.go:175 build_context="(go=go1.11.
2, user=root@4ecc17c53d26, date=20181109-15:40:48)"
level=error ts=2018-11-28T10:33:51.830464639Z caller=main.go:179 msg="Unable to create dat
a directory" err="mkdir data/: read-only file system"
```

这个是因为新版本 dockerfile 中的默认 WORKDIR 发生了变化，变成了 /etc/alertmanager 目录，默认情况下存储路径 --storage.path 是相对目录 data/，因此，alertmanager 会在我们上面挂载的 ConfigMap 中去创建这个目录，所以会报错，我们可以通过覆盖 --storage.path 参数来解决这个问题，在容器启动参数中添加该参数：

```
- name: alertmanager
  image: prom/alertmanager:v0.15.3
  imagePullPolicy: IfNotPresent
  args:
    - "--config.file=/etc/alertmanager/config.yml"
    - "--storage.path=/alertmanager/data"
```

重新更新 Pod，可以发现 Prometheus 已经是 Running 状态了：

```
$ kubectl apply -f prome-deploy.yaml
deployment.extensions "prometheus" configured
$ kubectl get pods -n kube-ops
NAME      READY     STATUS      RESTARTS   AGE
prometheus-646f457455-gr8x5   2/2      Running     0       3m
$ kubectl logs -f prometheus-646f457455-gr8x5 alertmanager -n kube-ops
level=info ts=2018-11-28T11:03:16.054633463Z caller=main.go:174 msg="Starting Alertmanager"
version="(version=0.15.3, branch=HEAD, revision=d4a7697cc90f8bce62efe7c44b63b542578ec0a1)"

level=info ts=2018-11-28T11:03:16.054931931Z caller=main.go:175 build_context="(go=go1.11.
2, user=root@4ecc17c53d26, date=20181109-15:40:48)"
level=info ts=2018-11-28T11:03:16.351058702Z caller=cluster.go:155 component=cluster msg="
setting advertise address explicitly" addr=10.244.2.217 port=9094
level=info ts=2018-11-28T11:03:16.456683857Z caller=main.go:322 msg="Loading configuration
file" file=/etc/alertmanager/config.yml
level=info ts=2018-11-28T11:03:16.548558156Z caller=cluster.go:570 component=cluster msg="
Waiting for gossip to settle..." interval=2s
```

```
level=info ts=2018-11-28T11:03:16.556768564Z caller=main.go:398 msg=Listening address=:9093
level=info ts=2018-11-28T11:03:18.549158865Z caller=cluster.go:595 component=cluster msg="gossip not settled" polls=0 before=0 now=1 elapsed=2.000272112s
level=info ts=2018-11-28T11:03:26.558221484Z caller=cluster.go:587 component=cluster msg="gossip settled; proceeding" elapsed=10.009335611s
```

报警规则

现在我们只是把 AlertManager 容器运行起来了，也和 Prometheus 进行了关联，但是现在我们并不知道要做什么报警，因为没有任何地方告诉我们要报警，所以我们还需要配置一些报警规则来告诉我们对哪些数据进行报警。

报警规则允许你基于 Prometheus 表达式语言的表达式来定义报警报条件，并在触发警报时发送通知给外部的接收者。

同样在 Prometheus 的配置文件中添加如下报警规则配置：

```
rule_files:
  - /etc/prometheus/rules.yml
```

其中 `rule_files` 就是用来指定报警规则的，这里我们同样将 `rules.yml` 文件用 ConfigMap 的形式挂载到 `/etc/prometheus` 目录下面即可：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: kube-ops
data:
  prometheus.yml: |
    ...
  rules.yml: |
    groups:
    - name: test-rule
      rules:
      - alert: NodeMemoryUsage
        expr: (node_memory_MemTotal_bytes - (node_memory_MemFree_bytes + node_memory_Buffers_bytes + node_memory_Cached_bytes)) / node_memory_MemTotal_bytes * 100 > 20
        for: 2m
        labels:
          team: node
        annotations:
          summary: "{{$labels.instance}}: High Memory usage detected"
          description: "{{$labels.instance}}: Memory usage is above 20% (current value is: {{ $value }})"
```

上面我们定义了一个名为 `NodeMemoryUsage` 的报警规则，其中：

- `for` 语句会使 Prometheus 服务等待指定的时间，然后执行查询表达式。
- `labels` 语句允许指定额外的标签列表，把它们附加在告警上。

- `annotations` 语句指定了另一组标签，它们不被当做告警实例的身份标识，它们经常用于存储一些额外的信息，用于报警信息的展示之类的。

为了方便演示，我们将的表达式判断报警临界值设置为20，重新更新 ConfigMap 资源对象，由于我们在 Prometheus 的 Pod 中已经通过 Volume 的形式将 `prometheus-config` 这个一个 ConfigMap 对象挂载到了 `/etc/prometheus` 目录下面，所以更新后，该目录下面也会出现 `rules.yml` 文件，所以前面配置的 `rule_files` 路径也是正常的，更新完成后，重新执行 `reload` 操作，这个时候我们去 Prometheus 的 Dashboard 中切换到 `alerts` 路径下面就可以看到有报警配置规则的数据了：

The screenshot shows the Prometheus Dashboard's 'Alerts' section. At the top, there are navigation links: Prometheus, Alerts, Graph, Status, and Help. Below the navigation is a search bar with the placeholder 'Show annotations'. The main area is titled 'NodeMemoryUsage (3 active)'. It displays the configuration for three alerts:

```

alert: NodeMemoryUsage
expr: (node_memory_MemTotal_bytes
      - (node_memory_MemFree_bytes + node_memory_Buffers_bytes + node_memory_Cached_bytes))
      / node_memory_MemTotal_bytes * 100 > 20
for: 2m
labels:
  team: node
annotations:
  description: '{{$labels.instance}}: Memory usage is above 20% (current value is:
    {{ $value }})
  summary: '{{$labels.instance}}: High Memory usage detected'

```

Below the configuration, there is a table showing the details of each alert:

Labels	State	Active Since	Value
<code>alername="NodeMemoryUsage"</code> , <code>beta_kubernetes.io_arch="amd64"</code> , <code>beta_kubernetes.io_os="linux"</code> , <code>com="youdianzhishi"</code> , <code>course="k8s"</code> , <code>instance="node02"</code> , <code>job="kubernetes-node-exporter"</code> , <code>kubernetes.io_hostname="node02"</code> , <code>team="node"</code>	FIRING	2018-11-28 12:26:55.407746617 +0000 UTC	37.64065567804747
<code>alername="NodeMemoryUsage"</code> , <code>beta_kubernetes.io_arch="amd64"</code> , <code>beta_kubernetes.io_os="linux"</code> , <code>instance="master"</code> , <code>job="kubernetes-node-exporter"</code> , <code>kubernetes.io_hostname="master"</code> , <code>team="node"</code>	FIRING	2018-11-28 12:26:55.407746617 +0000 UTC	43.32059574012012
<code>alername="NodeMemoryUsage"</code> , <code>beta_kubernetes.io_arch="amd64"</code> , <code>beta_kubernetes.io_os="linux"</code> , <code>instance="node03"</code> , <code>jnlp="haimaxy"</code> , <code>job="kubernetes-node-exporter"</code> , <code>kubernetes.io_hostname="node03"</code> , <code>team="node"</code>	FIRING	2018-11-28 12:26:55.407746617 +0000 UTC	29.43982048530683

prometheus alerts

我们可以看到页面中出现了我们刚刚定义的报警规则信息，而且报警信息中还有状态显示。一个报警信息在生命周期内有下面3种状态：

- `inactive`: 表示当前报警信息既不是 `firing` 状态也不是 `pending` 状态
- `pending`: 表示在设置的阈值时间范围内被激活了
- `firing`: 表示超过设置的阈值时间被激活了

我们这里的状态现在是 `firing` 就表示这个报警已经被激活了，我们这里的报警信息有一个 `team=node` 这样的标签，而最上面我们配置 `alertmanager` 的时候就有如下的路由配置信息了：

```

routes:
- receiver: email
group_wait: 10s
match:
  team: node

```

所以我们这里的报警信息会被 `email` 这个接收器来进行报警，我们上面配置的是邮箱，所以正常来说这个时候我们会收到一封如下的报警邮件：

The screenshot shows an email from the Prometheus Alertmanager. The subject line is "3 alerts for alertname=NodeMemoryUsage". Below the subject, there is a blue button labeled "View In AlertManager". The main content of the email is as follows:

[3] Firing

Labels

```
alertname = NodeMemoryUsage
beta_kubernetes_io_arch = amd64
beta_kubernetes_io_os = linux
instance = master
job = kubernetes-node-exporter
kubernetes_io_hostname = master
team = node
```

Annotations

```
description = master: Memory usage is above 20% (current value is: 43.169104947050364
summary = master: High Memory usage detected
```

[Source](#)

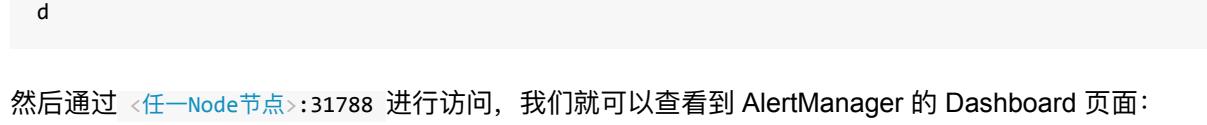
Labels

```
alertname = NodeMemoryUsage
beta_kubernetes_io_arch = amd64
beta_kubernetes_io_os = linux
com = youdianzhishi
course = k8s
instance = node02
job = kubernetes-node-exporter
kubernetes_io_hostname = node02
```

prometheus email receiver

我们可以看到收到的邮件内容中包含一个 `View In AlertManager` 的链接，我们同样可以通过 `NodePort` 的形式去访问到 AlertManager 的 Dashboard 页面：

\$ kubectl get svc -n kube-ops					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AG
E					
prometheus	NodePort	10.102.74.90	<none>	9093:31788/TCP,9090:30358/TCP	34



alertmanager webui

在这个页面中我们可以进行一些操作，比如过滤、分组等等，里面还有两个新的概念：Inhibition(抑制)和 Silences(静默)。

- **Inhibition**: 如果某些其他警报已经触发了，则对于某些警报，Inhibition 是一个抑制通知的概念。例如：一个警报已经触发，它正在通知整个集群是不可达的时，Alertmanager 则可以配置成关心这个集群的其他警报无效。这可以防止与实际问题无关的数百或数千个触发警报的通知，Inhibition 需要通过上面的配置文件进行配置。
- **Silences**: 静默是一个非常简单的方法，可以在给定时间内简单地忽略所有警报。Silences 基于 matchers 配置，类似路由树。来到的警告将会被检查，判断它们是否和活跃的 Silences 相等或者正则表达式匹配。如果匹配成功，则不会将这些警报发送给接收者。

由于全局配置中我们配置的 `repeat_interval: 5m`，所以正常来说，上面的测试报警如果一直满足报警条件(CPU 使用率大于 20%)的话，那么每 5 分钟我们就可以收到一条报警邮件。

现在我们添加一个 Silences，如下图所示，匹配 node02 节点的内存报警：

The screenshot shows the 'Edit Silence' page in the Prometheus Alertmanager interface. It includes fields for 'Start' (2018-11-28T13:10:32.051Z), 'Duration' (56m 57s), and 'End' (2018-11-28T14:07:29.244Z). The 'Matchers' section lists alert parameters: team (node), job (kubernetes-node-exporter), instance (node02), and alertname (NodeMemoryUsage). The 'Creator' field is set to '阳明'. A 'Comment' field contains the text '测试一个Silence'. At the bottom are buttons for 'Preview Alerts' (green), 'Update' (blue), and 'Reset' (red).

Name	Value	Regex	
team	node	<input type="checkbox"/>	剪切
job	kubernetes-node-exporter	<input type="checkbox"/>	剪切
instance	node02	<input type="checkbox"/>	剪切
alertname	NodeMemoryUsage	<input type="checkbox"/>	剪切

prometheus alertmanager Silences

添加完成后，等下一次的报警信息触发后，我们可以看到报警信息里面已经没有了节点 node02 的报警信息了：

2 alerts for alertname=NodeMemoryUsage

[View In AlertManager](#)

[2] Firing

Labels

```
alertname = NodeMemoryUsage
beta_kubernetes_io_arch = amd64
beta_kubernetes_io_os = linux
instance = master
job = kubernetes-node-exporter
kubernetes_io_hostname = master
team = node
```

Annotations

```
description = master: Memory usage is above 20% (current value is: 42.98164797249669
summary = master: High Memory usage detected
```

[Source](#)

Labels

```
alertname = NodeMemoryUsage
beta_kubernetes_io_arch = amd64
beta_kubernetes_io_os = linux
instance = node03
```

promtheus alertmanager email

由于我们上面添加的 Silences 是有过期时间的，所以在这个时间段过后，node02 的报警信息就会恢复了。

webhook接收器

上面我们配置的是 AlertManager 自带的邮件报警模板，我们也说了 AlertManager 支持很多中报警接收器，比如 slack、微信之类的，其中最为灵活的方式当然是使用 webhook 了，我们可以定义一个 webhook 来接收报警信息，然后在 webhook 里面去进行处理，需要发送怎样的报警信息我们自定义就可以。

比如我们这里用 Flask 编写了一个简单的处理钉钉报警的 webhook 的程序：

```
import os
```

```

import json
import requests

from flask import Flask
from flask import request

app = Flask(__name__)

@app.route('/', methods=['POST', 'GET'])
def send():
    if request.method == 'POST':
        post_data = request.get_data()
        send_alert(bytes2json(post_data))
        return 'success'
    else:
        return 'weclome to use prometheus alertmanager dingtalk webhook server!'

def bytes2json(data_bytes):
    data = data_bytes.decode('utf8').replace("''", "'")
    return json.loads(data)

def send_alert(data):
    token = os.getenv('ROBOT_TOKEN')
    if not token:
        print('you must set ROBOT_TOKEN env')
        return
    url = 'https://oapi.dingtalk.com/robot/send?access_token=%s' % token
    send_data = {
        "msgtype": "text",
        "text": {
            "content": data
        }
    }
    req = requests.post(url, json=send_data)
    result = req.json()
    if result['errcode'] != 0:
        print('notify dingtalk error: %s' % result['errcode'])

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)

```

代码非常简单，通过一个 ROBOT_TOKEN 的环境变量传入群机器人的 TOKEN，然后直接将 webhook 发送过来的数据直接以文本的形式转发给群机器人。

大家可以根据自己的需求来定制报警数据，上述代码仓库地址：github.com/cnych/alertmanager-dingtalk-hook

当然我们得将上面这个服务部署到集群中来，对应的资源清单如下：(dingtalk-hook.yaml)

```

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: dingtalk-hook
  namespace: kube-ops

```

```

spec:
  template:
    metadata:
      labels:
        app: dingtalk-hook
  spec:
    containers:
      - name: dingtalk-hook
        image: cnych/alertmanager-dingtalk-hook:v0.2
        imagePullPolicy: IfNotPresent
        ports:
          - containerPort: 5000
            name: http
        env:
          - name: ROBOT_TOKEN
            valueFrom:
              secretKeyRef:
                name: dingtalk-secret
                key: token
        resources:
          requests:
            cpu: 50m
            memory: 100Mi
          limits:
            cpu: 50m
            memory: 100Mi

---
apiVersion: v1
kind: Service
metadata:
  name: dingtalk-hook
  namespace: kube-ops
spec:
  selector:
    app: dingtalk-hook
  ports:
    - name: hook
      port: 5000
      targetPort: http

```

要注意上面我们声明了一个 ROBOT_TOKEN 的环境变量，由于这是一个相对于私密的信息，所以我们这里从一个 Secret 对象中去获取，通过如下命令创建一个名为 dingtalk-secret 的 Secret 对象，然后部署上面的资源对象即可：

```

$ kubectl create secret generic dingtalk-secret --from-literal=token=替换成钉钉群聊的机器人TOKEN -n kube-ops
secret "dingtalk-secret" created
$ kubectl create -f dingtalk-hook.yaml
deployment.extensions "dingtalk-hook" created
service "dingtalk-hook" created
$ kubectl get pods -n kube-ops
NAME                  READY   STATUS    RESTARTS   AGE
dingtalk-hook-c4fc8cd6-6r2b6   1/1     Running   0          45m
.....

```

部署成功后，现在我们就可以给 AlertManager 配置一个 webhook 了，在上面的配置中增加一个路由接收器

```

routes:
- receiver: webhook
  match:
    filesystem: node
receivers:
- name: 'webhook'
  webhook_configs:
- url: 'http://dingtalk-hook:5000'
  send_resolved: true

```

我们这里配置了一个名为 webhook 的接收器，地址为： `http://dingtalk-hook:5000`，这个地址当然就是上面我们部署的钉钉的 webhook 的接收程序的 Service 地址。

然后我们也在报警规则中添加一条关于节点文件系统使用情况的报警规则，注意 labels 标签要带上 `filesystem=node`，这样报警信息就会被 webhook 这一个接收器所匹配：

```

- alert: NodeFilesystemUsage
  expr: (node_filesystem_size_bytes{device="rootfs"} - node_filesystem_free_bytes{device="rootfs"}) / node_filesystem_size_bytes{device="rootfs"} * 100 > 10
  for: 2m
  labels:
    filesystem: node
  annotations:
    summary: "{$labels.instance}: High Filesystem usage detected"
    description: "{$labels.instance}: Filesystem usage is above 10% (current value is: { $value })"

```

更新 AlertManager 和 Prometheus 的 ConfigMap 资源对象（先删除再创建），更新完成后，隔一会儿执行 reload 操作是更新生效：

```

$ kubectl get svc -n kube-ops
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
prometheus   NodePort   10.102.74.90   <none>          9093:31788/TCP,9090:30358/TCP
  34d
$ curl -X POST "http://10.102.74.90:9093/-/reload"
$ curl -X POST "http://10.102.74.90:9090/-/reload"

```

AlertManager 和 Prometheus 都可以通过上面的 reload 操作进行重新加载
都完成更新后，再次去 Prometheus 的 Alert 路径下面查看报警信息：

Alerts

Show annotations

NodeMemoryUsage (3 active)

NodeFilesystemUsage (1 active)

```
alert: NodeFilesystemUsage
expr: (node_filesystem_size_bytes{device="rootfs"} - node_filesystem_free_bytes{device="rootfs"}) / node_filesystem_size_bytes{device="rootfs"} * 100 > 10
for: 2m
labels:
  filesystem: node
annotations:
  description: '$labels.instance: Filesystem usage is above 10% (current value is: $value)'
  summary: '$labels.instance: High Filesystem usage detected'
```

Labels	State	Active Since	Value
alertname="NodeFilesystemUsage" beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" device="rootfs" filesystem="node" fstype="rootfs" instance="node03" jnlp="haimaxy" job="kubernetes-node-exporter" kubernetes_io_hostname="node03" mountpoint="/" state="PENDING"	PENDING	2018-11-28 16:36:55.407746617 +0000 UTC	20.930974873065434

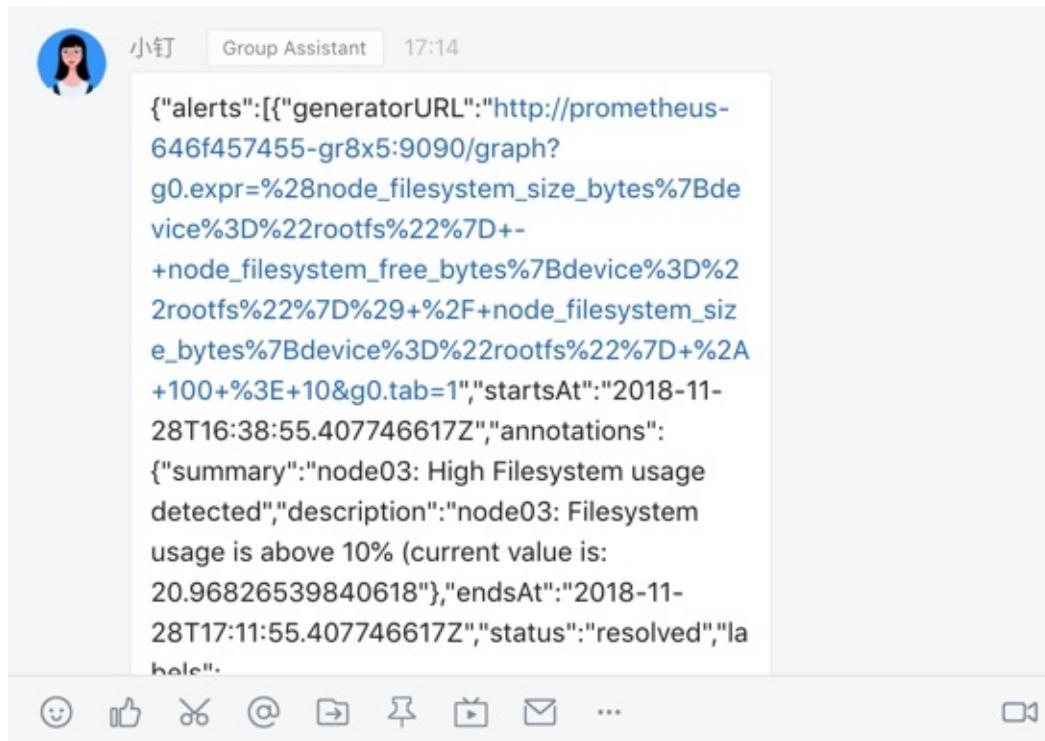
prometheus alerts

隔一会儿关于这个节点文件系统的报警就会被触发了，由于这个报警信息包含一个 `filesystem=node` 的 label 标签，所以会被路由到 `webhook` 这个接收器中，也就是上面我们自定义的这个 `dingtalk-hook`，触发后可以观察这个 Pod 的日志：

```
$ kubectl logs -f dingtalk-hook-cc677c46d-gf26f -n kube-ops
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

10.244.2.217 - - [28/Nov/2018 17:14:09] "POST / HTTP/1.1" 200 -
```

可以看到 POST 请求已经成功了，同时这个时候正常来说就可以收到一条钉钉消息了：



alertmanager dingtalk message

由于我们程序中是用一个非常简单的文本形式直接转发的，所以这里报警信息不够友好，没关系，有了这个示例我们完全就可以根据自己的需要来定制消息模板了，可以参考钉钉自定义机器人文档：<https://open-doc.dingtalk.com/microapp/serverapi2/qf2nxq>

到这里我们就完成了完全手动的控制 Prometheus、Grafana 以及我们的 AlertManager 的报警功能，接下来我们会给大家讲解 Kubernetes 中更加自动化的监控方案：Prometheus-Operator。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:05:57

58. Prometheus Operator 的安装

前面的课程中我们学习了用自定义的方式来对 Kubernetes 集群进行监控，但是还是有一些缺陷，比如 Prometheus、AlertManager 这些组件服务本身的高可用，当然我们也完全可以用自定义的方式来实现这些需求，我们也知道 Prometheus 在代码上就已经对 Kubernetes 有了原生的支持，可以通过服务发现的形式来自动监控集群，因此我们可以使用另外一种更加高级的方式来部署 Prometheus：Operator 框架。

Operator

Operator 是由 CoreOS 公司开发的，用来扩展 Kubernetes API，特定的应用程序控制器，它用来创建、配置和管理复杂的有状态应用，如数据库、缓存和监控系统。Operator 基于 Kubernetes 的资源和控制器概念之上构建，但同时又包含了应用程序特定的一些专业知识，比如创建一个数据库的 Operator，则必须对创建的数据库的各种运维方式非常了解，创建 Operator 的关键是 CRD（自定义资源）的设计。

CRD 是对 Kubernetes API 的扩展，Kubernetes 中的每个资源都是一个 API 对象的集合，例如我们在 YAML 文件里定义的那些 spec 都是对 Kubernetes 中的资源对象的定义，所有的自定义资源可以跟 Kubernetes 中内建的资源一样使用 kubectl 操作。

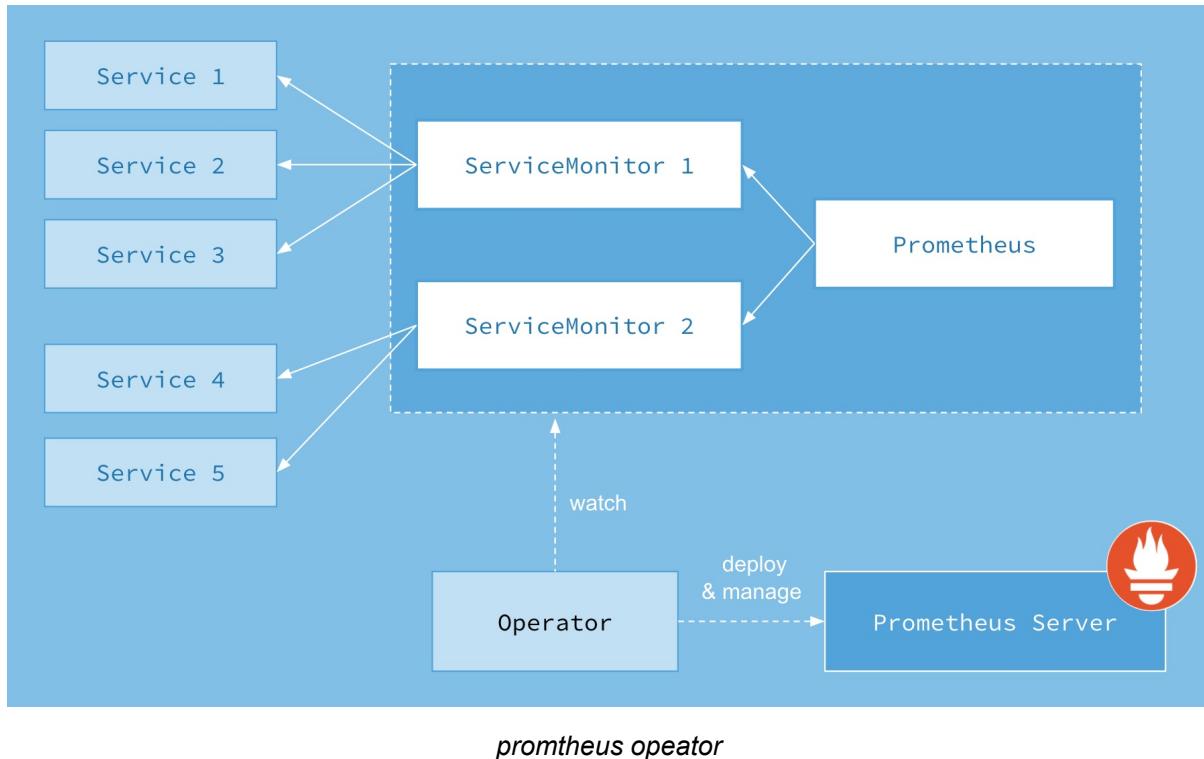
Operator 是将运维人员对软件操作的知识给代码化，同时利用 Kubernetes 强大的抽象来管理大规模的软件应用。目前 CoreOS 官方提供了几种 Operator 的实现，其中就包括我们今天的主角：Prometheus Operator，Operator 的核心实现就是基于 Kubernetes 的以下两个概念：

- 资源：对象的状态定义
- 控制器：观测、分析和行动，以调节资源的分布

当然我们如果有对应的需求也完全可以自己去实现一个 Operator，接下来我们就来给大家详细介绍下 Prometheus-Operator 的使用方法。

介绍

首先我们先来了解下 Prometheus-Operator 的架构图：



上图是 `Prometheus-Operator` 官方提供的架构图，其中 `Operator` 是最核心的部分，作为一个控制器，他会去创建 `Prometheus`、`ServiceMonitor`、`AlertManager` 以及 `PrometheusRule` 4个 CRD 资源对象，然后会一直监控并维持这4个资源对象的状态。

其中创建的 `prometheus` 这种资源对象就是作为 `Prometheus Server` 存在，而 `ServiceMonitor` 就是 `exporter` 的各种抽象，`exporter` 前面我们已经学习了，是用来提供专门提供 `metrics` 数据接口的工具，`Prometheus` 就是通过 `ServiceMonitor` 提供的 `metrics` 数据接口去 pull 数据的，当然 `alertmanager` 这种资源对象就是对应的 `AlertManager` 的抽象，而 `PrometheusRule` 是用来被 `Prometheus` 实例使用的报警规则文件。

这样我们要在集群中监控什么数据，就变成了直接去操作 `Kubernetes` 集群的资源对象了，是不是方便很多了。上图中的 `Service` 和 `ServiceMonitor` 都是 `Kubernetes` 的资源，一个 `ServiceMonitor` 可以通过 `labelSelector` 的方式去匹配一类 `Service`，`Prometheus` 也可以通过 `labelSelector` 去匹配多个 `ServiceMonitor`。

安装

我们这里直接通过 `Prometheus-Operator` 的源码来进行安装，当然也可以用 `Helm` 来进行一键安装，我们采用源码安装可以去了解更多的实现细节。首页将源码 Clone 下来：

```
$ git clone https://github.com/coreos/prometheus-operator
$ cd contrib/kube-prometheus/manifests/
$ ls
00namespace-namespace.yaml                               node-exporter-clusterRole
1e.yaml                                                 node-exporter-daemonset
0prometheus-operator-0alertmanagerCustomResourceDefinition.yaml
```

```
.yaml
```

```
....
```

进入到 manifests 目录下面，这个目录下面包含我们所有的资源清单文件，我们需要对其中的文件 `prometheus-serviceMonitorKubelet.yaml` 进行简单的修改，因为默认情况下，这个 ServiceMonitor 是关联的 kubelet 的 10250 端口去采集的节点数据，而我们前面说过为了安全，这个 metrics 数据已经迁移到 10255 这个只读端口上面去了，我们只需要将文件中的 `https-metrics` 更改成 `http-metrics` 即可，这个在 Prometheus-Operator 对节点端点同步的代码中有相关定义，感兴趣的可以[点此查看完整代码](#)：

```
Subsets: []v1.EndpointSubset{
    {
        Ports: []v1.EndpointPort{
            {
                Name: "https-metrics",
                Port: 10250,
            },
            {
                Name: "http-metrics",
                Port: 10255,
            },
            {
                Name: "cadvisor",
                Port: 4194,
            },
        },
    },
},
```

修改完成后，直接在该文件夹下面执行创建资源命令即可：

```
$ kubectl apply -f .
```

部署完成后，会创建一个名为 `monitoring` 的 namespace，所以资源对象对将部署在改命名空间下面，此外 Operator 会自动创建 4 个 CRD 资源对象：

```
$ kubectl get crd | grep coreos
alertmanagers.monitoring.coreos.com      5d
prometheuses.monitoring.coreos.com        5d
prometheusrules.monitoring.coreos.com     5d
servicemonitors.monitoring.coreos.com     5d
```

可以在 `monitoring` 命名空间下面查看所有的 Pod，其中 `alertmanager` 和 `prometheus` 是用 `StatefulSet` 控制器管理的，其中还有一个比较核心的 `prometheus-operator` 的 Pod，用来控制其他资源对象和监听对象变化的：

```
$ kubectl get pods -n monitoring
NAME                  READY   STATUS    RESTARTS   AGE
alertmanager-main-0   2/2     Running   0          21h
alertmanager-main-1   2/2     Running   0          21h
alertmanager-main-2   2/2     Running   0          21h
```

grafana-df9bfd765-f4dvw	1/1	Running	0	22h
kube-state-metrics-77c9658489-ntj66	4/4	Running	0	20h
node-exporter-4sr7f	2/2	Running	0	21h
node-exporter-9mh2r	2/2	Running	0	21h
node-exporter-m2gkp	2/2	Running	0	21h
prometheus-adapter-dc548cc6-r6lhb	1/1	Running	0	22h
prometheus-k8s-0	3/3	Running	1	21h
prometheus-k8s-1	3/3	Running	1	21h
prometheus-operator-bdf79ff67-9dc48	1/1	Running	0	21h

查看创建的 Service:

kubectl get svc -n monitoring					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
alertmanager-main	ClusterIP	10.110.204.224	<none>	9093/TCP	23h
alertmanager-operated	ClusterIP	None	<none>	9093/TCP, 6783/TCP	23h
grafana	ClusterIP	10.98.191.31	<none>	3000/TCP	23h
kube-state-metrics	ClusterIP	None	<none>	8443/TCP, 9443/TCP	23h
node-exporter	ClusterIP	None	<none>	9100/TCP	23h
prometheus-adapter	ClusterIP	10.107.201.172	<none>	443/TCP	23h
prometheus-k8s	ClusterIP	10.107.105.53	<none>	9090/TCP	23h
prometheus-operated	ClusterIP	None	<none>	9090/TCP	23h
prometheus-operator	ClusterIP	None	<none>	8080/TCP	23h

可以看到上面针对 grafana 和 prometheus 都创建了一个类型为 ClusterIP 的 Service, 当然如果我们想要在外网访问这两个服务的话可以通过创建对应的 Ingress 对象或者使用 NodePort 类型的 Service, 我们这里为了简单, 直接使用 NodePort 类型的服务即可, 编辑 grafana 和 prometheus-k8s 这两个 Service, 将服务类型更改为 NodePort:

\$ kubectl edit svc grafana -n monitoring					
\$ kubectl edit svc prometheus-k8s -n monitoring					
\$ kubectl get svc -n monitoring					
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
grafana	NodePort	10.98.191.31	<none>	3000:32333/TCP	23h
prometheus-k8s	NodePort	10.107.105.53	<none>	9090:30166/TCP	23h
.....					

更改完成后, 我们就可以通过去访问上面的两个服务了, 比如查看 prometheus 的 targets 页面:

Targets						
All		Unhealthy				
monitoring/alertmanager/0 (3/3 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
http://10.244.2.23:9093/metrics	UP	<code>endpoint="web" instance="10.244.2.23:9093" job="alertmanager-main" namespace="monitoring" pod="alertmanager-main-2" service="alertmanager-main"</code>	11.329s ago	12.02ms		
http://10.244.2.26:9093/metrics	UP	<code>endpoint="web" instance="10.244.2.26:9093" job="alertmanager-main" namespace="monitoring" pod="alertmanager-main-0" service="alertmanager-main"</code>	18.697s ago	11.01ms		
http://10.244.4.104:9093/metrics	UP	<code>endpoint="web" instance="10.244.4.104:9093" job="alertmanager-main" namespace="monitoring" pod="alertmanager-main-1" service="alertmanager-main"</code>	16.011s ago	10.33ms		
monitoring/coredns/0 (0/0 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
monitoring/kube-apiserver/0 (1/1 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
https://10.151.30.57:6443/metrics	UP	<code>endpoint="https" instance="10.151.30.57:6443" job="apiserver" namespace="default" service="kube-apiserver"</code>	3.118s ago	653.4ms		
monitoring/kube-controller-manager/0 (0/0 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
monitoring/kube-scheduler/0 (0/0 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
monitoring/kube-state-metrics/0 (1/1 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
https://10.244.2.29:8443/metrics	UP	<code>endpoint="https-main" instance="10.244.2.29:8443" job="kube-state-metrics" namespace="monitoring" pod="kube-state-metrics-77c96984d9-nj96t" service="kube-state-metrics"</code>	19.554s ago	1.298s		
monitoring/kube-state-metrics/1 (1/1 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
https://10.244.2.29:9443/metrics	UP	<code>endpoint="https-self" instance="10.244.2.29:9443" job="kube-state-metrics" namespace="monitoring" pod="kube-state-metrics-77c96984d9-nj96t" service="kube-state-metrics"</code>	3.046s ago	67.05ms		
monitoring/kubelet/0 (3/3 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
http://10.151.30.57:10255/metrics	UP	<code>endpoint="http-metrics" instance="10.151.30.57:10255" job="kubelet" namespace="kube-system" node="master" service="kubelet"</code>	25.521s ago	17.38ms		
http://10.151.30.63:10255/metrics	UP	<code>endpoint="http-metrics" instance="10.151.30.63:10255" job="kubelet" namespace="kube-system" node="node02" service="kubelet"</code>	19.602s ago	20.56ms		
http://10.151.30.64:10255/metrics	UP	<code>endpoint="http-metrics" instance="10.151.30.64:10255" job="kubelet" namespace="kube-system" node="node03" service="kubelet"</code>	21.768s ago	18.89ms		
monitoring/kubelet/1 (3/3 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
http://10.151.30.57:10255/metrics/cadvisor	UP	<code>endpoint="http-metrics" instance="10.151.30.57:10255" job="kubelet" namespace="kube-system" node="master" service="kubelet"</code>	2.41s ago	166.6ms		
http://10.151.30.63:10255/metrics/cadvisor	UP	<code>endpoint="http-metrics" instance="10.151.30.63:10255" job="kubelet" namespace="kube-system" node="node02" service="kubelet"</code>	17.334s ago	907ms		
http://10.151.30.64:10255/metrics/cadvisor	UP	<code>endpoint="http-metrics" instance="10.151.30.64:10255" job="kubelet" namespace="kube-system" node="node03" service="kubelet"</code>	15.285s ago	193ms		
monitoring/node-exporter/0 (3/3 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
https://10.151.30.57:9100/metrics	UP	<code>endpoint="https" instance="10.151.30.57:9100" job="node-exporter" namespace="monitoring" pod="node-exporter-mlqpl" service="node-exporter"</code>	6.311s ago	149.7ms		
https://10.151.30.63:9100/metrics	UP	<code>endpoint="https" instance="10.151.30.63:9100" job="node-exporter" namespace="monitoring" pod="node-exporter-9mzb2" service="node-exporter"</code>	27.538s ago	109.1ms		
https://10.151.30.64:9100/metrics	UP	<code>endpoint="https" instance="10.151.30.64:9100" job="node-exporter" namespace="monitoring" pod="node-exporter-4rt7f" service="node-exporter"</code>	8.625s ago	54.61ms		
monitoring/prometheus-operator/0 (1/1 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
http://10.244.4.100:8080/metrics	UP	<code>endpoint="http" instance="10.244.4.100:8080" job="prometheus-operator" namespace="monitoring" pod="prometheus-operator-lbf79mf7-8dc48" service="prometheus-operator"</code>	13.212s ago	4.815ms		
monitoring/prometheus/0 (2/2 up) <small>show less</small>						
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error	
http://10.244.2.25:9090/metrics	UP	<code>endpoint="web" instance="10.244.2.25:9090" job="prometheus-k8s" namespace="monitoring" pod="prometheus-k8s-0" service="prometheus-k8s"</code>	6.817s ago	14.3ms		
http://10.244.4.103:9090/metrics	UP	<code>endpoint="web" instance="10.244.4.103:9090" job="prometheus-k8s" namespace="monitoring" pod="prometheus-k8s-1" service="prometheus-k8s"</code>	8.15s ago	17.9ms		

promtheus operator targets

配置

我们可以看到大部分的配置都是正常的，只有两三个没有管理到对应的监控目标，比如 kube-controller-manager 和 kube-scheduler 这两个系统组件，这就和 ServiceMonitor 的定义有关系了，我们先来查看下 kube-scheduler 组件对应的 ServiceMonitor 资源的定义：(prometheus-serviceMonitorKubeScheduler.yaml)

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    k8s-app: kube-scheduler
  name: kube-scheduler
  namespace: monitoring
spec:
  endpoints:
  - interval: 30s # 每30s获取一次信息
    port: http-metrics # 对应service的端口名
  jobLabel: k8s-app
  namespaceSelector: # 表示去匹配某一命名空间中的service, 如果想从所有的namespace中匹配用any: true
  matchNames:
  - kube-system
  selector: # 匹配的 Service 的labels, 如果使用matchLabels, 则下面的所有标签都匹配时才会匹配该service, 如果使用matchExpressions, 则至少匹配一个标签的service都会被选择
    matchLabels:
      k8s-app: kube-scheduler
```

上面是一个典型的 ServiceMonitor 资源文件的声明方式，上面我们通过 `selector.matchLabels` 在 `kube-system` 这个命名空间下面匹配具有 `k8s-app=kube-scheduler` 这样的 Service，但是我们系统中根本就没有对应的 Service，所以我们需要手动创建一个 Service：(prometheus-kubeSchedulerService.yaml)

```
apiVersion: v1
kind: Service
metadata:
  namespace: kube-system
  name: kube-scheduler
  labels:
    k8s-app: kube-scheduler
spec:
  selector:
    component: kube-scheduler
  ports:
  - name: http-metrics
    port: 10251
    targetPort: 10251
    protocol: TCP
```

10251是 kube-scheduler 组件 metrics 数据所在的端口，10252是 kube-controller-manager 组件的监控数据所在端口。

其中最重要的是上面 labels 和 selector 部分，labels 区域的配置必须和我们上面的 ServiceMonitor 对象中的 selector 保持一致，selector 下面配置的是 component=kube-scheduler，为什么是这个 label 标签呢，我们可以去 describe 下 kube-scheduelr 这个 Pod：

```
$ kubectl describe pod kube-scheduler-master -n kube-system
Name:           kube-scheduler-master
Namespace:      kube-system
Node:          master/10.151.30.57
Start Time:    Sun, 05 Aug 2018 18:13:32 +0800
Labels:        component=kube-scheduler
               tier=control-plane
...
```

我们可以看到这个 Pod 具有 component=kube-scheduler 和 tier=control-plane 这两个标签，而前面这个标签具有更唯一的特性，所以使用前面这个标签较好，这样上面创建的 Service 就可以和我们的 Pod 进行关联了，直接创建即可：

```
$ kubectl create -f prometheus-kubeSchedulerService.yaml
$ kubectl get svc -n kube-system -l k8s-app=kube-scheduler
NAME            TYPE      CLUSTER-IP     EXTERNAL-IP   PORT(S)      AGE
kube-scheduler   ClusterIP  10.102.119.231 <none>       10251/TCP   18m
```

创建完成后，隔一小会儿后去 prometheus 查看 targets 下面 kube-scheduler 的状态：

monitoring/kube-scheduler/0 (0/1 up)					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.151.30.57:10251/metrics	DOWN	endpoint="http-metrics" instance="10.151.30.57:10251" job="kube-scheduler" namespace="kube-system" pods="kube-scheduler-master" service="kube-scheduler"	13.755s ago	1.126ms	Get http://10.151.30.57:10251/metrics: dial tcp 10.151.30.57:10251: connect: connection refused

prometheus kube-scheduler error

我们可以看到现在已经发现了 target，但是抓取数据结果出错了，这个错误是因为我们集群是使用 kubeadm 搭建的，其中 kube-scheduler 默认是绑定在 127.0.0.1 上面的，而上面我们这个地方是想通过节点的 IP 去访问，所以访问被拒绝了，我们只要把 kube-scheduler 绑定的地址改成 0.0.0.0 即可满足要求，由于 kube-scheduler 是以静态 Pod 的形式运行在集群中的，所以我们只需要更改静态 Pod 目录下相对应的 YAML 文件即可：

```
$ ls /etc/kubernetes/manifests/
etcd.yaml  kube-apiserver.yaml  kube-controller-manager.yaml  kube-scheduler.yaml
```

将 kube-scheduler.yaml 文件中 --command 的 --address 地址改成 0.0.0.0：

```
  containers:
```

```

- command:
- kube-scheduler
- --leader-elect=true
- --kubeconfig=/etc/kubernetes/scheduler.conf
- --address=0.0.0.0

```

修改完成后我们将该文件从当前文件夹中移除，隔一会儿再移回该目录，就可以自动更新了，然后再去看 prometheus 中 kube-scheduler 这个 target 是否已经正常了：

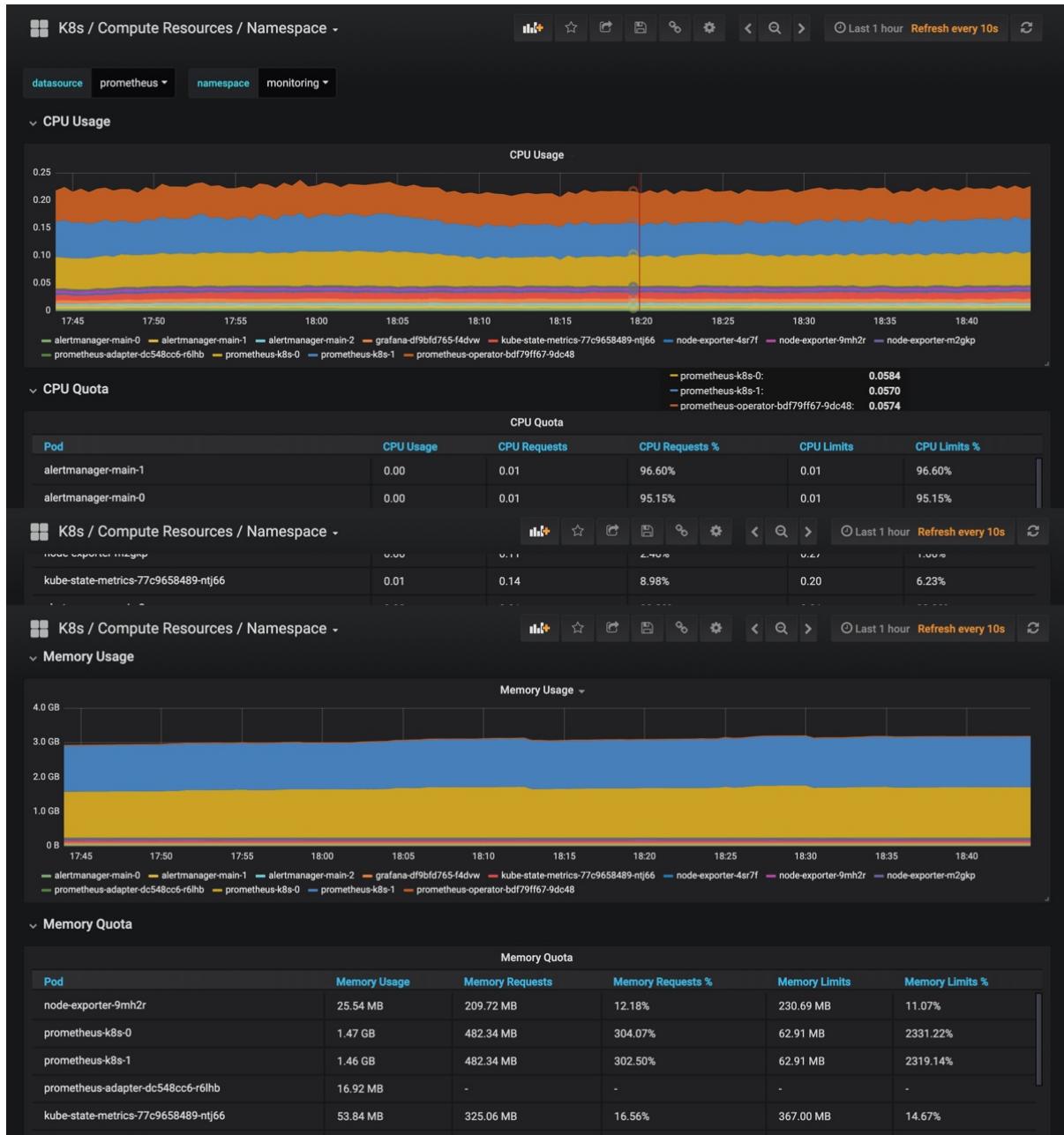
[monitoring/kube-scheduler/0 \(1/1 up\)](#) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.151.30.57:10251/metrics	UP	endpoint="http-metrics" Instance="10.151.30.57:10251" job="kube-scheduler" namespace="kube-system" pod="kube-scheduler-master" service="kubernetes-scheduler"	7.758s ago	8.968ms	

promethues-operator-kube-scheduler

大家可以按照上面的方法尝试去修复下 kube-controller-manager 组件的监控。

上面的监控数据配置完成后，现在我们可以去查看下 grafana 下面的 dashboard，同样使用上面的 NodePort 访问即可，第一次登录使用 admin:admin 登录即可，进入首页后，可以发现已经和我们的 Prometheus 数据源关联上了，正常来说可以看到一些监控图表了：



下节课我们再来和大家介绍怎样来完全自定义一个 ServiceMonitor 以及 AlertManager 相关的配置。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:06:26

59. 自定义 Prometheus Operator 监控项

上节课和大家讲解了 Prometheus Operator 的安装和基本使用方法，这节课给大家介绍如何在 Prometheus Operator 中添加一个自定义的监控项。

除了 Kubernetes 集群中的一些资源对象、节点以及组件需要监控，有的时候我们可能还需要根据实际的业务需求去添加自定义的监控项，添加一个自定义监控的步骤也是非常简单的。

- 第一步建立一个 ServiceMonitor 对象，用于 Prometheus 添加监控项
- 第二步为 ServiceMonitor 对象关联 metrics 数据接口的一个 Service 对象
- 第三步确保 Service 对象可以正确获取到 metrics 数据

接下来我们就来为大家演示如何添加 etcd 集群的监控。

无论是 Kubernetes 集群外的还是使用 Kubeadm 安装在集群内部的 etcd 集群，我们这里都将其视作集群外的独立集群，因为对于二者的使用方法没什么特殊之处。

etcd 证书

对于 etcd 集群一般情况下，为了安全都会开启 https 证书认证的方式，所以要想让 Prometheus 访问到 etcd 集群的监控数据，就需要提供相应的证书校验。

由于我们这里演示环境使用的是 Kubeadm 搭建的集群，我们可以使用 kubectl 工具去获取 etcd 启动的时候使用的证书路径：

```
$ kubectl get pods -n kube-system
NAME                  READY   STATUS    RESTARTS   AGE
etcd-master           1/1     Running   0          2h
$ kubectl get pod etcd-master -n kube-system -o yaml
...
spec:
  containers:
  - command:
    - etcd
    - --peer-cert-file=/etc/kubernetes/pki/etcd/peer.crt
    - --listen-client-urls=https://127.0.0.1:2379
    - --advertise-client-urls=https://127.0.0.1:2379
    - --client-cert-auth=true
    - --peer-client-cert-auth=true
    - --data-dir=/var/lib/etcd
    - --cert-file=/etc/kubernetes/pki/etcd/server.crt
    - --key-file=/etc/kubernetes/pki/etcd/server.key
    - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
    - --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
    - --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
  image: k8s.gcr.io/etcd-amd64:3.1.12
  imagePullPolicy: IfNotPresent
  livenessProbe:
    exec:
      command:
      - /bin/sh
      - -ec
```

```

- ETCDCTL_API=3 etcdctl --endpoints=127.0.0.1:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt
  --cert=/etc/kubernetes/pki/etcd/healthcheck-client.crt --key=/etc/kubernetes/pki/etcd/healthcheck-client.key
  get foo
failureThreshold: 8
initialDelaySeconds: 15
periodSeconds: 10
successThreshold: 1
timeoutSeconds: 15
name: etcd
resources: {}
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
volumeMounts:
- mountPath: /var/lib/etcd
  name: etcd-data
- mountPath: /etc/kubernetes/pki/etcd
  name: etcd-certs
.....
tolerations:
- effect: NoExecute
  operator: Exists
volumes:
- hostPath:
  path: /var/lib/etcd
  type: DirectoryOrCreate
  name: etcd-data
- hostPath:
  path: /etc/kubernetes/pki/etcd
  type: DirectoryOrCreate
  name: etcd-certs
.....

```

我们可以看到 etcd 使用的证书都对应在节点的 /etc/kubernetes/pki/etcd 这个路径下面，所以首先我们将需要使用的证书通过 secret 对象保存到集群中去：(在 etcd 运行的节点)

```
$ kubectl -n monitoring create secret generic etcd-certs --from-file=/etc/kubernetes/pki/etcd/healthcheck-client.crt --from-file=/etc/kubernetes/pki/etcd/healthcheck-client.key --from-file=/etc/kubernetes/pki/etcd/ca.crt
secret "etcd-certs" created
```

如果你是独立的二进制方式启动的 etcd 集群，同样将对应的证书保存到集群中的一个 secret 对象中去即可。

然后将上面创建的 etcd-certs 对象配置到 prometheus 资源对象中，直接更新 prometheus 资源对象即可：

```
$ kubectl edit prometheus k8s -n monitoring
```

添加如下的 secrets 属性：

```

nodeSelector:
  beta.kubernetes.io/os: linux
replicas: 2

```

```
secrets:
- etcd-certs
```

更新完成后，我们就可以在 Prometheus 的 Pod 中获取到上面创建的 etcd 证书文件了，具体的路径我们可以进入 Pod 中查看：

```
$ kubectl exec -it prometheus-k8s-0 /bin/sh -n monitoring
Defaulting container name to prometheus.
Use 'kubectl describe pod/prometheus-k8s-0 -n monitoring' to see all of the containers in
this pod.
/ $ ls /etc/prometheus/secrets/etcd-certs/
ca.crt      healthcheck-client.crt  healthcheck-client.key
```

创建 ServiceMonitor

现在 Prometheus 访问 etcd 集群的证书已经准备好了，接下来创建 ServiceMonitor 对象即可（prometheus-serviceMonitorEtcd.yaml）

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: etcd-k8s
  namespace: monitoring
  labels:
    k8s-app: etcd-k8s
spec:
  jobLabel: k8s-app
  endpoints:
  - port: port
    interval: 30s
    scheme: https
    tlsConfig:
      caFile: /etc/prometheus/secrets/etcd-certs/ca.crt
      certFile: /etc/prometheus/secrets/etcd-certs/healthcheck-client.crt
      keyFile: /etc/prometheus/secrets/etcd-certs/healthcheck-client.key
      insecureSkipVerify: true
  selector:
    matchLabels:
      k8s-app: etcd
  namespaceSelector:
    matchNames:
    - kube-system
```

上面我们在 monitoring 命名空间下面创建了名为 etcd-k8s 的 ServiceMonitor 对象，基本属性和前面章节中的一致，匹配 kube-system 这个命名空间下面的具有 k8s-app=etcd 这个 label 标签的 Service，jobLabel 表示用于检索 job 任务名称的标签，和前面不太一样的地方是 endpoints 属性的写法，配置上访问 etcd 的相关证书，endpoints 属性下面可以配置很多抓取的参数，比如 relabel、proxyUrl，tlsConfig 表示用于配置抓取监控数据端点的 tls 认证，由于证书 serverName 和 etcd 中签发的可能不匹配，所以加上了 insecureSkipVerify=true

TLSConfig specifies TLS configuration parameters.

Field	Description	Scheme	Required
caFile	The CA cert to use for the targets.	string	false
certFile	The client cert file for the targets.	string	false
keyFile	The client key file for the targets.	string	false
serverName	Used to verify the hostname for the targets.	string	false
insecureSkipVerify	Disable target certificate validation.	bool	false

tlsConfig

关于 ServiceMonitor 属性的更多用法可以查看文档：<https://github.com/coreos/prometheus-operator/blob/master/Documentation/api.md> 了解更多

直接创建这个 ServiceMonitor 对象：

```
$ kubectl create -f prometheus-serviceMonitorEtcd.yaml
servicemonitor.monitoring.coreos.com "etcd-k8s" created
```

创建 Service

ServiceMonitor 创建完成了，但是现在还没有关联的对应的 Service 对象，所以需要我们去手动创建一个 Service 对象 (prometheus-etcdService.yaml)：

```
apiVersion: v1
kind: Service
metadata:
  name: etcd-k8s
  namespace: kube-system
  labels:
    k8s-app: etcd
spec:
  type: ClusterIP
  clusterIP: None
  ports:
  - name: port
    port: 2379
    protocol: TCP

---
apiVersion: v1
kind: Endpoints
metadata:
  name: etcd-k8s
  namespace: kube-system
  labels:
    k8s-app: etcd
```

```

subsets:
- addresses:
  - ip: 10.151.30.57
    nodeName: etc-master
  ports:
  - name: port
    port: 2379
    protocol: TCP

```

我们这里创建的 Service 没有采用前面通过 label 标签的形式去匹配 Pod 的做法，因为前面我们说过很多时候我们创建的 etcd 集群是独立于集群之外的，这种情况下面我们就需要自定义一个 Endpoints，要注意 metadata 区域的内容要和 Service 保持一致，Service 的 clusterIP 设置为 None，对改知识点不太熟悉的，可以去查看我们前面关于 Service 部分的讲解。

Endpoints 的 subsets 中填写 etcd 集群的地址即可，我们这里是单节点的，填写一个即可，直接创建该 Service 资源：

```
$ kubectl create -f prometheus-etcdService.yaml
```

创建完成后，隔一会儿去 Prometheus 的 Dashboard 中查看 targets，便会有 etcd 的监控项了：

monitoring/etc-k8s/0 (0/1 up) show less					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
https://10.151.30.57:2379/metrics	DOWN	<code>endpoint="port"</code> <code>instance="10.151.30.57:2379"</code> <code>job="etcd"</code> <code>namespace="kube-system"</code> <code>service="etcd-k8s"</code>	23.611s ago	1.358ms	Get https://10.151.30.57:2379/metrics : dial tcp 10.151.30.57:2379: connect: connection refused

prometheus etcd

可以看到还是有一个明显的错误，和我们上节课监控 kube-scheduler 的错误比较类似于，因为我们这里的 etcd 的是监听在 127.0.0.1 这个 IP 上面的，所以访问会拒绝：

```
--listen-client-urls=https://127.0.0.1:2379
```

同样我们只需要在 /etc/kubernetes/manifest/ 目录下面（static pod 默认的目录）的 etcd.yaml 文件中将上面的 `listen-client-urls` 更改成 0.0.0.0 即可：

```
--listen-client-urls=https://0.0.0.0:2379
```

重启 etcd，生效后，查看 etcd 这个监控任务就正常了：

monitoring/etc-k8s/0 (1/1 up) show less					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
https://10.151.30.57:2379/metrics	UP	<code>endpoint="port"</code> <code>instance="10.151.30.57:2379"</code> <code>job="etcd"</code> <code>namespace="kube-system"</code> <code>service="etcd-k8s"</code>	18.893s ago	6.847ms	

prometheus etcd

数据采集到后，可以在 grafana 中导入编号为 3070 的 dashboard，获取到 etcd 的监控图表。



grafana etcd dashboard

配置 PrometheusRule

现在我们知道怎么自定义一个 ServiceMonitor 对象了，但是如果需要自定义一个报警规则的话呢？比如现在我们去查看 Prometheus Dashboard 的 Alert 页面下面就已经有一些报警规则了，还有一些是已经触发规则的了：

CPUThrottlingHigh (10 active)
CoreDNSDown (1 active)
DeadMansSwitch (1 active)
KubeControllerManagerDown (1 active)
AlertmanagerConfigInconsistent (0 active)
AlertmanagerDown (0 active)
AlertmanagerFailedReload (0 active)
AlertmanagerMembersInconsistent (0 active)
KubeAPIDown (0 active)
KubeAPIErrorsHigh (0 active)
KubeAPIErrorsHigh (0 active)
KubeAPILatencyHigh (0 active)

alerts

但是这些报警信息是哪里来的呢？他们应该用怎样的方式通知我们呢？我们知道之前我们使用自定义的方式可以在 Prometheus 的配置文件之中指定 AlertManager 实例和 报警的 rules 文件，现在我们通过 Operator 部署的呢？我们可以在 Prometheus Dashboard 的 Config 页面下面查看关于 AlertManager 的配置：

```

alerting:
  alert_relabel_configs:
    - separator: ;
      regex: prometheus_replica
      replacement: $1
      action: labeldrop
  alertmanagers:
    - kubernetes_sd_configs:
      - role: endpoints
        namespaces:
          names:
            - monitoring
        scheme: http
        path_prefix: /
        timeout: 10s
      relabel_configs:
        - source_labels: [__meta_kubernetes_service_name]
          separator: ;
          regex: alertmanager-main
          replacement: $1
          action: keep
        - source_labels: [__meta_kubernetes_endpoint_port_name]
          separator: ;

```

```

  regex: web
  replacement: $1
  action: keep
rule_files:
- /etc/prometheus/rules/prometheus-k8s-rulefiles-0/*.yaml

```

上面 alertmanagers 实例的配置我们可以看到是通过角色为 endpoints 的 kubernetes 的服务发现机制获取的，匹配的是服务名为 alertmanager-main，端口名未 web 的 Service 服务，我们查看下 alertmanager-main 这个 Service：

```

kubectl describe svc alertmanager-main -n monitoring
Name:           alertmanager-main
Namespace:      monitoring
Labels:         alertmanager=main
Annotations:   kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"labels":{"alertmanager":"main"},"name":"alertmanager-main","namespace":"monitoring"},...}
Selector:       alertmanager=main,app=alertmanager
Type:          NodePort
IP:            10.104.156.29
Port:          web  9093/TCP
TargetPort:    web/TCP
NodePort:      web  31918/TCP
Endpoints:     10.244.2.34:9093,10.244.2.37:9093,10.244.4.109:9093
Session Affinity: None
External Traffic Policy: Cluster
Events:        <none>

```

可以看到服务名正是 alertmanager-main，Port 定义的名称也是 web，符合上面的规则，所以 Prometheus 和 AlertManager 组件就正确关联上了。而对应的报警规则文件位于：/etc/prometheus/rules/prometheus-k8s-rulefiles-0/ 目录下面所有的 YAML 文件。我们可以进入 Prometheus 的 Pod 中验证下该目录下面是否有 YAML 文件：

```

$ kubectl exec -it prometheus-k8s-0 /bin/sh -n monitoring
Defaulting container name to prometheus.
Use 'kubectl describe pod/prometheus-k8s-0 -n monitoring' to see all of the containers in
this pod.
/prometheus $ ls /etc/prometheus/rules/prometheus-k8s-rulefiles-0/
monitoring-prometheus-k8s-rules.yaml
/prometheus $ cat /etc/prometheus/rules/prometheus-k8s-rulefiles-0/monitoring-pr
ometheus-k8s-rules.yaml
groups:
- name: k8s.rules
  rules:
  - expr: |
    sum(rate(container_cpu_usage_seconds_total{job="kubelet", image!="", container_name!=""}[5m])) by (namespace)
    record: namespace:container_cpu_usage_seconds_total:sum_rate
    .....

```

这个 YAML 文件实际上就是我们之前创建的一个 PrometheusRule 文件包含的：

```
$ cat prometheus-rules.yaml
```

```

apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    prometheus: k8s
    role: alert-rules
  name: prometheus-k8s-rules
  namespace: monitoring
spec:
  groups:
    - name: k8s.rules
      rules:
        - expr: |
            sum(rate(container_cpu_usage_seconds_total{job="kubelet", image!="", container_name != ""}[5m])) by (namespace)
            record: namespace:container_cpu_usage_seconds_total:sum_rate

```

我们这里的 PrometheusRule 的 name 为 prometheus-k8s-rules, namespace 为 monitoring, 我们可以猜想到我们创建一个 PrometheusRule 资源对象后, 会自动在上面的 prometheus-k8s-rulefiles-0 目录下面生成一个对应的 <namespace>-<name>.yaml 文件, 所以如果以后我们需要自定义一个报警选项的话, 只需要定义一个 PrometheusRule 资源对象即可。至于为什么 Prometheus 能够识别这个 PrometheusRule 资源对象呢? 这就需要查看我们创建的 prometheus 这个资源对象了, 里面有非常重要的一个属性 ruleSelector, 用来匹配 rule 规则的过滤器, 要求匹配具有 prometheus=k8s 和 role=alert-rules 标签的 PrometheusRule 资源对象, 现在明白了吧?

```

ruleSelector:
  matchLabels:
    prometheus: k8s
    role: alert-rules

```

所以我们要想自定义一个报警规则, 只需要创建一个具有 prometheus=k8s 和 role=alert-rules 标签的 PrometheusRule 对象就行了, 比如现在我们添加一个 etcd 是否可用的报警, 我们知道 etcd 整个集群有一半以上的节点可用的话集群就是可用的, 所以我们判断如果不可用的 etcd 数量超过了一半那么就触发报警, 创建文件 prometheus-etcdRules.yaml:

```

apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    prometheus: k8s
    role: alert-rules
  name: etcd-rules
  namespace: monitoring
spec:
  groups:
    - name: etcd
      rules:
        - alert: EtcdClusterUnavailable
          annotations:
            summary: etcd cluster small
            description: If one more etcd peer goes down the cluster will be unavailable
          expr: |
            count(up{job="etcd"} == 0) > (count(up{job="etcd"}) / 2 - 1)

```

```
for: 3m
labels:
  severity: critical
```

注意 label 标签一定至少要有 `prometheus=k8s` 和 `role=alert-rules`, 创建完成后, 隔一会儿再去容器中查看下 `rules` 文件夹:

```
kubectl exec -it prometheus-k8s-0 /bin/sh -n monitoring
Defaulting container name to prometheus.
Use 'kubectl describe pod/prometheus-k8s-0 -n monitoring' to see all of the containers in
this pod.
/prometheus $ ls /etc/prometheus/rules/prometheus-k8s-rulefiles-0/
monitoring-etcd-rules.yaml           monitoring-prometheus-k8s-rules.yaml
```

可以看到我们创建的 rule 文件已经被注入到了对应的 rulefiles 文件夹下面了, 证明我们上面的设想是正确的。然后再去 Prometheus Dashboard 的 Alert 页面下面就可以查看到上面我们新建的报警规则了:

EtcdbClusterUnavailable (0 active)

```
alert: EtcdbClusterUnavailable
expr: count(up{job="etcd"} == 0) > (count(up{job="etcd"}) / 2 - 1)
for: 3m
labels:
  severity: critical
annotations:
  description: If one more etcd peer goes down the cluster will be unavailable
  summary: etcd cluster small
```

etcd cluster

配置报警

我们知道了如何去添加一个报警规则配置项, 但是这些报警信息用怎样的方式去发送呢? 前面的课程中我们知道我们可以通过 AlertManager 的配置文件去配置各种报警接收器, 现在我们是通过 Operator 提供的 `alertmanager` 资源对象创建的组件, 应该怎样去修改配置呢?

首先我们将 `alertmanager-main` 这个 Service 改为 NodePort 类型的 Service, 修改完成后我们可以在页面上的 `status` 路径下面查看 AlertManager 的配置信息:

Config

```

global:
  resolve_timeout: 5m
  http_config: {}
  smtp_hello: localhost
  smtp_require_tls: true
  pagerduty_url: https://events.pagerduty.com/v2/enqueue
  hipchat_api_url: https://api.hipchat.com/
  opsgenie_api_url: https://api.opsgenie.com/
  wechat_api_url: https://qyapi.weixin.qq.com/cgi-bin/
  victorops_api_url: https://alert.victorops.com/integrations/generic/20131114/alert/
route:
  receiver: "null"
  group_by:
    - job
  routes:
    - receiver: "null"
      match:
        alertname: DeadMansSwitch
      group_wait: 30s
      group_interval: 5m
      repeat_interval: 12h
receivers:
  - name: "null"
templates: []

```

alertmanager config

这些配置信息实际上是来自于我们之前在 `prometheus-operator/contrib/kube-prometheus/manifests` 目录下面创建的 `alertmanager-secret.yaml` 文件：

```

apiVersion: v1
data:
  alertmanager.yaml: Imdsb2JhbCI6IAogICJyZXNvbHZlX3RpbWVvdXQiOiAiNW0iCiJyZWNlaXZlcMi0iAKLSaibmFtZSI6ICJudWxsIgoicm91dGUo0iAKICAiZ3JvdXBfYnki0iAKICAtICJqb2IiCiAgImdyb3VwX21udGVydmFsIjogIjVtIgogICJncm91cF93YWl0IjogIjMwcyIKICAicmVjZWl2ZXIIo0iAibnVsbCIKICAicmVwZWF0X21udGVydmFsIjogIjEyaCIKICAicm91dGVzIjogCiAgLSaibWF0Y2gi0iAKICAgICAgImFsZXJ0bmFtZSI6ICJEZWFkTWFuc1N3aXRjaCIKICAjCJyZWNlaXZlcii6ICJudWxsIg==
kind: Secret
metadata:
  name: alertmanager-main
  namespace: monitoring
  type: Opaque

```

可以将 `alertmanager.yaml` 对应的 `value` 值做一个 base64 解码：

```

$ echo "Imdsb2JhbCI6IAogICJyZXNvbHZlX3RpbWVvdXQiOiAiNW0iCiJyZWNlaXZlcMi0iAKLSaibmFtZSI6ICJudWxsIgoicm91dGUo0iAKICAiZ3JvdXBfYnki0iAKICAtICJqb2IiCiAgImdyb3VwX21udGVydmFsIjogIjVtIgogICJncm91cF93YWl0IjogIjMwcyIKICAicmVjZWl2ZXIIo0iAibnVsbCIKICAicmVwZWF0X21udGVydmFsIjogIjEyaCIKICAicm91dGVzIjogCiAgLSaibWF0Y2gi0iAKICAgICAgImFsZXJ0bmFtZSI6ICJEZWFkTWFuc1N3aXRjaCIKICAjCJyZWNlaXZlcii6ICJudWxsIg==" | base64 -d
"global":
  "resolve_timeout": "5m"
"receivers":
  - "name": "null"

```

```

"route":
  "group_by":
    - "job"
  "group_interval": "5m"
  "group_wait": "30s"
  "receiver": "null"
  "repeat_interval": "12h"
  "routes":
    - "match":
        "alertname": "DeadMansSwitch"
      "receiver": "null"

```

我们可以看到内容和上面查看的配置信息是一致的，所以如果我们想要添加自己的接收器，或者模板消息，我们就可以更改这个文件：

```

global:
  resolve_timeout: 5m
  smtp_smarthost: 'smtp.163.com:25'
  smtp_from: 'ych_1024@163.com'
  smtp_auth_username: 'ych_1024@163.com'
  smtp_auth_password: '<邮箱密码>'
  smtp_hello: '163.com'
  smtp_require_tls: false
route:
  group_by: ['job', 'severity']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 12h
  receiver: default
  routes:
    - receiver: webhook
      match:
        alertname: CoreDNSDown
receivers:
  - name: 'default'
    email_configs:
      - to: '517554016@qq.com'
        send_resolved: true
  - name: 'webhook'
    webhook_configs:
      - url: 'http://dingtalk-hook.kube-ops:5000'
        send_resolved: true

```

将上面文件保存为 alertmanager.yaml，然后使用这个文件创建一个 Secret 对象：

```

# 先将之前的 secret 对象删除
$ kubectl delete secret alertmanager-main -n monitoring
secret "alertmanager-main" deleted
$ kubectl create secret generic alertmanager-main --from-file=alertmanager.yaml -n monitoring
secret "alertmanager-main" created

```

我们添加了两个接收器，默认的通过邮箱进行发送，对于 CoreDNSDown 这个报警我们通过 webhook 来进行发送，这个 webhook 就是我们前面课程中定义的一个钉钉接收的 Server，上面的步骤创建完成后，很快我们就会收到一条钉钉消息：



小钉 Group Assistant

```
{"alerts": [{"generatorURL": "http://prometheus-k8s-0:9090/graph?g0.expr=absent%28up%7Bjob%3D%22kube-dns%22%7D+%3D%3D+1%29&g0.tab=1", "startsAt": "2018-12-17T11:44:17.64901609Z", "annotations": {"runbook_url": "https://github.com/kubernetes-monitoring/kubernetes-mixin/tree/master/runbook.md#alert-name-corednsdown", "message": "CoreDNS has disappeared from Prometheus target discovery."}, "endsAt": "0001-01-01T00:00:00Z", "status": "firing", "labels": {"severity": "critical", "alertname": "CoreDNSDown", "prometheus": "monitoring/k8s"}, "commonLabels": {"severity": "critical", "alertname": "CoreDNSDown", "prometheus": "monitoring/k8s"}, "externalURL": "http://alertmanager-main-0:9093", "groupLabels": {"severity": "critical"}, "receiver": "webhook", "version": "4", "commonAnnotations": {"runbook_url": "https://github.com/kubernetes-monitoring/kubernetes-mixin/tree/master/runbook.md#alert-name-corednsdown", "message": "CoreDNS has disappeared from Prometheus target discovery."}, "groupKey": "{}", "alertname": "CoreDNSDown", "severity": "critical", "status": "firing"}]}
```

钉钉

同样邮箱中也会收到报警信息：

9 alerts for severity=warning

[View In AlertManager](#)

[9] Firing

Labels

```
alertname = CPUThrottlingHigh
container_name = addon-resizer
namespace = monitoring
pod_name = kube-state-metrics-77c9658489-xcnbk
prometheus = monitoring/k8s
severity = warning
```

Annotations

```
message = 48% throttling of CPU in namespace monitoring for container addon-resizer in
pod kube-state-metrics-77c9658489-xcnbk.
runbook_url = https://github.com/kubernetes-monitoring/kubernetes-mixin/tree/master/runbook.md#alert-name-cputhrottlinghigh
```

[Source](#)

Labels

```
alertname = CPUThrottlingHigh
container_name = kube-flannel
namespace = kube-system
```

邮箱

我们再次查看 AlertManager 页面的 status 页面的配置信息可以看到已经变成上面我们的配置信息了：

Config

```

global:
  resolve_timeout: 5m
  http_config: {}
  smtp_from: ych_1024@163.com
  smtp_hello: 163.com
  smtp_smarthost: smtp.163.com:25
  smtp_auth_username: ych_1024@163.com
  smtp_auth_password: <secret>
  pagerduty_url: https://events.pagerduty.com/v2/enqueue
  hipchat_api_url: https://api.hipchat.com/
  opsgenie_api_url: https://api.opsgenie.com/
  wechat_api_url: https://qyapi.weixin.qq.com/cgi-bin/
  victorops_api_url: https://alert.victorops.com/integrations/generic/20131114/alert/
route:
  receiver: default
  group_by:
    - job
    - severity
  routes:
    - receiver: webhook
      match:
        alertname: CoreDNSDown
      group_wait: 30s
      group_interval: 5m
      repeat_interval: 12h
receivers:
- name: default
  email_configs:
    - send_resolved: true
      to: 517554016@qq.com
      from: ych_1024@163.com
      hello: 163.com
      smarthost: smtp.163.com:25
      auth_username: ych_1024@163.com
      auth_password: <secret>
      headers:
        From: ych_1024@163.com
        Subject: '{{ template "email.default.subject" . }}'
        To: 517554016@qq.com
        html: '{{ template "email.default.html" . }}'
        require_tls: false
    - name: webhook
      webhook_configs:
        - send_resolved: true
          http_config: {}
          url: http://dingtalk-hook.kube-ops:5000
templates: []

```

alertmanager config

AlertManager 配置也可以使用模板(.tmpl文件)，这些模板可以与 alertmanager.yaml 配置文件一起添加到 Secret 对象中，比如：

```

apiVersion: v1
kind: secret
metadata:
  name: alertmanager-example
data:
  alertmanager.yaml: {{BASE64_CONFIG}}
  template_1 tmpl: {{BASE64_TEMPLATE_1}}
  template_2 tmpl: {{BASE64_TEMPLATE_2}}
  ...

```

模板会被放置到与配置文件相同的路径，当然要使用这些模板文件，还需要在 alertmanager.yaml 配置文件中指定：

```
templates:  
- '*.tmpl'
```

创建成功后， Secret 对象将会挂载到 AlertManager 对象创建的 AlertManager Pod 中去。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

60. Prometheus Operator 高级配置

上节课我们一起学习了如何在 Prometheus Operator 下面自定义一个监控选项，以及自定义报警规则的使用。那么我们还能够直接使用前面课程中的自动发现功能吗？如果在我们的 Kubernetes 集群中有了很多的 Service/Pod，那么我们都需要一个一个的去建立一个对应的 ServiceMonitor 对象来进行监控吗？这样岂不是又变得麻烦起来了？

自动发现配置

为解决上面的问题，Prometheus Operator 为我们提供了一个额外的抓取配置的来解决这个问题，我们可以通过添加额外的配置来进行服务发现进行自动监控。和前面自定义的方式一样，我们想要在 Prometheus Operator 当中去自动发现并监控具有 `prometheus.io/scrape=true` 这个 annotations 的 Service，之前我们定义的 Prometheus 的配置如下：

```
- job_name: 'kubernetes-service-endpoints'
  kubernetes_sd_configs:
  - role: endpoints
    relabel_configs:
    - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scrape]
      action: keep
      regex: true
    - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scheme]
      action: replace
      target_label: __scheme__
      regex: (https?)
    - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
      action: replace
      target_label: __metrics_path__
      regex: (.)
    - source_labels: [__address__, __meta_kubernetes_service_annotation_prometheus_io_port]
      action: replace
      target_label: __address__
      regex: ([^:]+)(?::(\d+)?);(\d+)
      replacement: $1:$2
    - action: labelmap
      regex: __meta_kubernetes_service_label_(.+)
    - source_labels: [__meta_kubernetes_namespace]
      action: replace
      target_label: kubernetes_namespace
    - source_labels: [__meta_kubernetes_service_name]
      action: replace
      target_label: kubernetes_name
```

如果你对上面这个配置还不是很熟悉的话，建议去查看下前面关于 [Kubernetes常用资源对象监控章节的介绍](#)，要想自动发现集群中的 Service，就需要我们在 Service 的 annotation 区域添加 `prometheus.io/scrape=true` 的声明，将上面文件直接保存为 `prometheus-additional.yaml`，然后通过这个文件创建一个对应的 Secret 对象：

```
$ kubectl create secret generic additional-configs --from-file=prometheus-additional.yaml
-n monitoring
```

```
secret "additional-configs" created
```

注意我们所有的操作都在 Prometheus Operator 源码 `contrib/kube-prometheus/manifests/` 目录下面。

创建完成后，会将上面配置信息进行 base64 编码后作为 `prometheus-additional.yaml` 这个 key 对应的值存在：

```
$ kubectl get secret additional-configs -n monitoring -o yaml
apiVersion: v1
data:
  prometheus-additional.yaml: LSBqb2JfbmFtZTogJ2t1YmVybmV0ZXMc2Vydmljzs1lbnRwb2ludHMnCiAg
    a3ViZXJuZXRLc19zF9jb25maWdzOgogIC0gcmsZTogZW5kcG9pbnRzCiAgcmVsYWJ1bF9jb25maWdzOgogIC0gc2
    91cmN1X2xhYmVsczogW19fbWV0YV9rdwJ1cm51dGVzX3N1cnZpY2VfYW5ub3RhG1vb19wcm9tZXRoZXVzX21vX3Nj
    cmFwZ0KICAgIGFjdG1vbjoga2V1cAogICAgnVnZXg6IHRydWUKICAtIHNvdXJjZV9sYWJ1bHM6IFtfX211dGFfa3
    ViZXJuZXRLc19zZXJ2aWN1X2Fubm90YXRpb25fcHJvbWV0aGV1c19pb19zY2h1bWVdCiAgICBhY3Rpb246IHJ1cGxh
    Y2UKICAgIHRhcndlDf9sYWJ1bDogX19zY2h1bWVfXwogICAgnVnZXg6IChodHRwcz8pCiAgLSBzb3Vy2VfbGFizW
    xz0iBbX19tZXRhX2t1YmVybmV0ZXNfc2VydmljZV9hb5vdGF0aW9uX3Byb211dGh1dXNfaWfcGF0aF0KICAgIGFj
    dG1vbjogcmVwbGFjZQogICAgnGfY2ZV0X2xhYmVs0iBfx211dHJpY3NfcGF0aF9fCiAgICByZwdleDogKC4rKQogIC
    0gc291cmN1X2xhYmVsczogW19fYWRkcmVzc19fLCBFx211dGFfa3ViZXJuZXRLc19zZXJ2aWN1X2Fubm90YXRpb25f
    cHJvbWV0aGV1c19pb19wb3J0XQogICAgnYWN0aW9u0iByZXBsYWN1CiAgICB0YXJnZXrbfbGFizWw6IF9FYWRkcmVzc1
    9fc1AgICByZwdleDogKFte010rKSg/OjpcZCspPzsoXGQrKQogICAgnVwbGFjZW1lbnQ6ICQx0iQyCiAgLSBhY3Rp
    b246IGxhYmVsbWFwCiAgICByZwdleDogX19tZXRhX2t1YmVybmV0ZXNfc2VydmljZV9sYWJ1bF8oLispCiAgLSBzb3
    VyY2VfbGFizWxz0iBbx19tZXRhX2t1YmVybmV0ZXNfbmFtZXNwYWN1XQogICAgnYWN0aW9u0iByZXBsYWN1CiAgICB0
    YXJnZXrbfbGFizWw6IGt1YmVybmV0ZXNfbmFtZXNwYWN1CiAgLSBzb3VY2VfbGFizWxz0iBbx19tZXRhX2t1YmVybm
    V0ZXNfc2VydmljZV9uYW11XQogICAgnYWN0aW9u0iByZXBsYWN1CiAgICB0YXJnZXrbfbGFizWw6IGt1YmVybmV0ZXNf
    bmFtZQo=
kind: Secret
metadata:
  creationTimestamp: 2018-12-20T14:50:35Z
  name: additional-configs
  namespace: monitoring
  resourceVersion: "41814998"
  selfLink: /api/v1/namespaces/monitoring/secrets/additional-configs
  uid: 9bbe22c5-0466-11e9-a777-525400db4df7
  type: Opaque
```

然后我们只需要在声明 `prometheus` 的资源对象文件中添加上这个额外的配置：(`prometheus-prometheus.yaml`)

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  labels:
    prometheus: k8s
  name: k8s
  namespace: monitoring
spec:
  alerting:
    alertmanagers:
      - name: alertmanager-main
        namespace: monitoring
        port: web
  baseImage: quay.io/prometheus/prometheus
  nodeSelector:
    beta.kubernetes.io/os: linux
```

```
replicas: 2
secrets:
- etcd-certs
resources:
requests:
  memory: 400Mi
ruleSelector:
  matchLabels:
    prometheus: k8s
    role: alert-rules
securityContext:
  fsGroup: 2000
  runAsNonRoot: true
  runAsUser: 1000
additionalScrapeConfigs:
  name: additional-configs
  key: prometheus-additional.yaml
serviceAccountName: prometheus-k8s
serviceMonitorNamespaceSelector: {}
serviceMonitorSelector: {}
version: v2.5.0
```

添加完成后，直接更新 prometheus 这个 CRD 资源对象：

```
$ kubectl apply -f prometheus-prometheus.yaml
prometheus.monitoring.coreos.com "k8s" configured
```

隔一小会儿，可以前往 Prometheus 的 Dashboard 中查看配置是否生效：

```

- job_name: kubernetes-service-endpoints
  scrape_interval: 30s
  scrape_timeout: 10s
  metrics_path: /metrics
  scheme: http
  kubernetes_sd_configs:
  - role: endpoints
    relabel_configs:
    - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scrape]
      separator: ;
      regex: "true"
      replacement: $1
      action: keep
    - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scheme]
      separator: ;
      regex: (https?)
      target_label: __scheme__
      replacement: $1
      action: replace
    - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
      separator: ;
      regex: (.)
      target_label: __metrics_path__
      replacement: $1
      action: replace
    - source_labels: [__address__, __meta_kubernetes_service_annotation_prometheus_io_port]
      separator: ;
      regex: ([^:]+)(?::\d+)?;(\d+)
      target_label: __address__
      replacement: $1:$2
      action: replace

```

config

在 Prometheus Dashboard 的配置页面下面我们可以看到已经有了对应的的配置信息了，但是我们切换到 targets 页面下面却并没有发现对应的监控任务，查看 Prometheus 的 Pod 日志：

```

$ kubectl logs -f prometheus-k8s-0 prometheus -n monitoring
level=error ts=2018-12-20T15:14:06.772903214Z caller=main.go:240 component=k8s_client_runtime
err="github.com/prometheus/prometheus/discovery/kubernetes/kubernetes.go:302: Failed to
list *v1.Pod: pods is forbidden: User \"system:serviceaccount:monitoring:prometheus-k8s\"
" cannot list pods at the cluster scope"
level=error ts=2018-12-20T15:14:06.773096875Z caller=main.go:240 component=k8s_client_runtime
err="github.com/prometheus/prometheus/discovery/kubernetes/kubernetes.go:301: Failed to
list *v1.Service: services is forbidden: User \"system:serviceaccount:monitoring:prometh
eus-k8s\" cannot list services at the cluster scope"
level=error ts=2018-12-20T15:14:06.773212629Z caller=main.go:240 component=k8s_client_runtime
err="github.com/prometheus/prometheus/discovery/kubernetes/kubernetes.go:300: Failed to
list *v1.Endpoints: endpoints is forbidden: User \"system:serviceaccount:monitoring:prom
etheus-k8s\" cannot list endpoints at the cluster scope"
.....

```

可以看到有很多错误日志出现，都是 `xxx is forbidden`，这说明是 RBAC 权限的问题，通过 prometheus 资源对象的配置可以知道 Prometheus 绑定了一个名为 `prometheus-k8s` 的 ServiceAccount 对象，而这个对象绑定的是一个名为 `prometheus-k8s` 的 ClusterRole：(`prometheus-clusterRole.yaml`)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-k8s
rules:
- apiGroups:
  - ""
  resources:
  - nodes/metrics
  verbs:
  - get
- nonResourceURLs:
  - /metrics
  verbs:
  - get
```

上面的权限规则中我们可以看到明显没有对 Service 或者 Pod 的 list 权限，所以报错了，要解决这个问题，我们只需要添加上需要的权限即可：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-k8s
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  - services
  - endpoints
  - pods
  - nodes/proxy
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - configmaps
  - nodes/metrics
  verbs:
  - get
- nonResourceURLs:
  - /metrics
  verbs:
  - get
```

更新上面的 ClusterRole 这个资源对象，然后重建下 Prometheus 的所有 Pod，正常就可以看到 targets 页面下面有 kubernetes-service-endpoints 这个监控任务了：

kubernetes-service-endpoints (2/2 up) show less					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.244.2.39:9121/metrics	UP	instance="10.244.2.39:9121" job="kubernetes-service-endpoints" kubernetes_name="redis" kubernetes_namespace="kube-ops"	23.485s ago	18.43ms	
http://10.244.4.54:9090/metrics	UP	instance="10.244.4.54:9090" job="kubernetes-service-endpoints" kubernetes_name="prometheus" kubernetes_namespace="istio-system" name="prometheus"	27.481s ago	9.797ms	

endpoints

我们这里自动监控了两个 Service，第一个就是我们之前创建的 Redis 的服务，我们在 Redis Service 中有两个特殊的 annotations：

```
annotations:
  prometheus.io/scrape: "true"
  prometheus.io/port: "9121"
```

所以被自动发现了，当然我们也可以用同样的方式去配置 Pod、Ingress 这些资源对象的自动发现。

数据持久化

上面我们在修改完权限的时候，重启了 Prometheus 的 Pod，如果我们仔细观察的话会发现我们之前采集的数据已经没有了，这是因为我们通过 `prometheus` 这个 CRD 创建的 Prometheus 并没有做数据的持久化，我们可以直接查看生成的 Prometheus Pod 的挂载情况就清楚了：

```
$ kubectl get pod prometheus-k8s-0 -n monitoring -o yaml
.....
volumeMounts:
- mountPath: /etc/prometheus/config_out
  name: config-out
  readOnly: true
- mountPath: /prometheus
  name: prometheus-k8s-db
.....
volumes:
.....
- emptyDir: {}
  name: prometheus-k8s-db
.....
```

我们可以看到 Prometheus 的数据目录 `/prometheus` 实际上是通过 `emptyDir` 进行挂载的，我们知道 `emptyDir` 挂载的数据的生命周期和 Pod 生命周期一致的，所以如果 Pod 挂掉了，数据也就丢失了，这也就是为什么我们重建 Pod 后之前的数据就没有了的原因，对应线上的监控数据肯定需要做数据的持久化的，同样的 `prometheus` 这个 CRD 资源也为我们提供了数据持久化的配置方法，由于我们的 Prometheus 最终是通过 `Statefulset` 控制器进行部署的，所以我们这里需要通过 `storageclass` 来做数据持久化，首先创建一个 `StorageClass` 对象：

```
apiVersion: storage.k8s.io/v1
```

```

kind: StorageClass
metadata:
  name: prometheus-data-db
provisioner: fuseim.pri/ifs

```

这里我们声明一个 StorageClass 对象，其中 provisioner=fuseim.pri/ifs，则是因为我们集群中使用的是 nfs 作为存储后端，而前面我们课程中创建的 nfs-client-provisioner 中指定的 PROVISIONER_NAME 就为 fuseim.pri/ifs，这个名字不能随便更改，将该文件保存为 prometheus-storageclass.yaml：

```

$ kubectl create -f prometheus-storageclass.yaml
storageclass.storage.k8s.io "prometheus-data-db" created

```

然后在 prometheus 的 CRD 资源对象中添加如下配置：

```

storage:
  volumeClaimTemplate:
    spec:
      storageClassName: prometheus-data-db
      resources:
        requests:
          storage: 10Gi

```

注意这里的 storageClassName 名字为上面我们创建的 StorageClass 对象名称，然后更新 prometheus 这个 CRD 资源。更新完成后会自动生成两个 PVC 和 PV 资源对象：

```

$ kubectl get pvc -n monitoring
NAME                           STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   AGE
prometheus-k8s-db-prometheus-k8s-0   Bound   pvc-0cc03d41-047a-11e9-a777-525400db4df7
10Gi       RWO            prometheus-data-db 8m
prometheus-k8s-db-prometheus-k8s-1   Bound   pvc-1938de6b-047b-11e9-a777-525400db4df7
10Gi       RWO            prometheus-data-db 1m
$ kubectl get pv
NAME                           CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
US     CLAIM                           STORAGECLASS   REASON   AGE
pvc-0cc03d41-047a-11e9-a777-525400db4df7   10Gi       RWO           Delete   Bound
d     monitoring/prometheus-k8s-db-prometheus-k8s-0   prometheus-data-db 2m
pvc-1938de6b-047b-11e9-a777-525400db4df7   10Gi       RWO           Delete   Bound
d     monitoring/prometheus-k8s-db-prometheus-k8s-1   prometheus-data-db 1m

```

现在我们再去看 Prometheus Pod 的数据目录就可以看到是关联到一个 PVC 对象上了。

```

$ kubectl get pod prometheus-k8s-0 -n monitoring -o yaml
. . .
volumeMounts:
- mountPath: /etc/prometheus/config_out
  name: config-out
  readOnly: true
- mountPath: /prometheus
  name: prometheus-k8s-db
. . .
volumes:

```

```
.....
- name: prometheus-k8s-db
  persistentVolumeClaim:
    claimName: prometheus-k8s-db-prometheus-k8s-0
.....
```

现在即使我们的 Pod 挂掉了，数据也不会丢失了，最后，下面是我们 Prometheus Operator 系列课程中最终的创建资源清单文件，更多的信息可以在<https://github.com/cnchy/kubernetes-learning> 下面查看。

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  labels:
    prometheus: k8s
  name: k8s
  namespace: monitoring
spec:
  alerting:
    alertmanagers:
      - name: alertmanager-main
        namespace: monitoring
        port: web
  storage:
    volumeClaimTemplate:
      spec:
        storageClassName: prometheus-data-db
        resources:
          requests:
            storage: 10Gi
  baseImage: quay.io/prometheus/prometheus
  nodeSelector:
    beta.kubernetes.io/os: linux
  replicas: 2
  secrets:
    - etcd-certs
  additionalScrapeConfigs:
    name: additional-configs
    key: prometheus-additional.yaml
  resources:
    requests:
      memory: 400Mi
  ruleSelector:
    matchLabels:
      prometheus: k8s
      role: alert-rules
  securityContext:
    fsGroup: 2000
    runAsNonRoot: true
    runAsUser: 1000
  serviceAccountName: prometheus-k8s
  serviceMonitorNamespaceSelector: {}
  serviceMonitorSelector: {}
version: v2.5.0
```

到这里 Prometheus Operator 系列教程就告一段落了，大家还有什么问题可以到微信群里面继续交流，接下来会和大家介绍 Kubernetes 日志收集方便的知识点。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

61. 日志收集架构

前面的课程中和大家一起学习了 Kubernetes 集群中监控系统的搭建，除了对集群的监控报警之外，还有一项运维工作是非常重要的，那就是日志的收集。

介绍

应用程序和系统日志可以帮助我们了解集群内部的运行情况，日志对于我们调试问题和监视集群情况也是非常有用的。而且大部分的应用都会有日志记录，对于传统的应用大部分都会写入到本地的日志文件之中。对于容器化应用程序来说则更简单，只需要将日志信息写入到 stdout 和 stderr 即可，容器默认情况下就会把这些日志输出到宿主机上的一个 JSON 文件之中，同样我们也可以通过 docker logs 或者 kubectl logs 来查看到对应的日志信息。

但是，通常来说容器引擎或运行时提供的功能不足以记录完整的日志信息，比如，如果容器崩溃了、Pod 被驱逐了或者节点挂掉了，我们仍然也希望访问应用程序的日志。所以，日志应该独立于节点、Pod 或容器的生命周期，这种设计方式被称为 cluster-level-logging，即完全独立于 Kubernetes 系统，需要自己提供单独的日志后端存储、分析和查询工具。

Kubernetes 中的基本日志

下面这个示例是 Kubernetes 中的一个基本日志记录的示例，直接将数据输出到标准输出流，如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c,
           'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

将上面文件保存为 counter-pod.yaml，该 Pod 每秒输出一些文本信息，创建这个 Pod：

```
$ kubectl create -f counter-pod.yaml
pod "counter" created
```

创建完成后，可以使用 kubectl logs 命令查看日志信息：

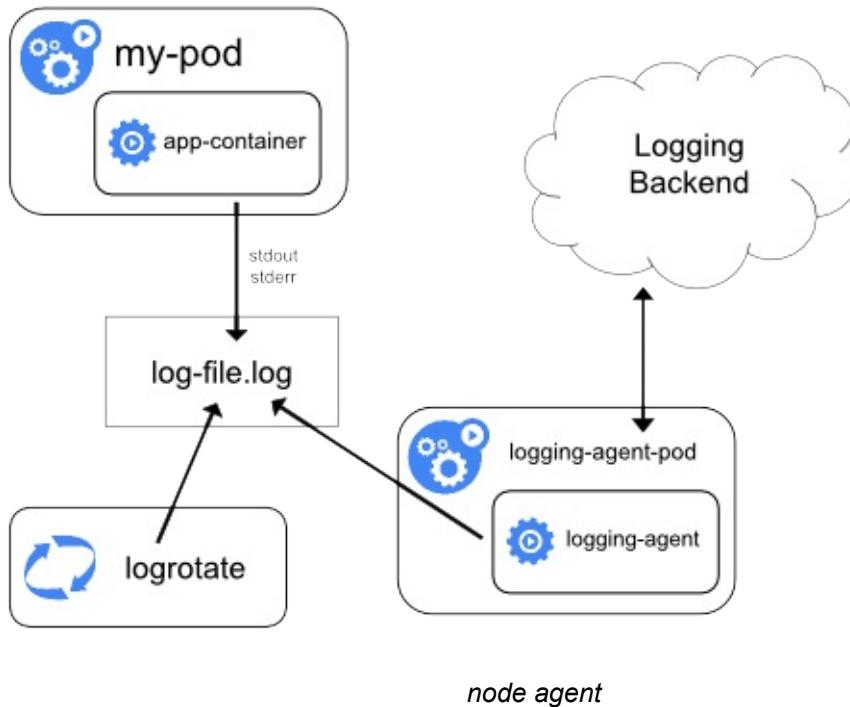
```
$ kubectl logs counter
0: Thu Dec 27 15:47:04 UTC 2018
1: Thu Dec 27 15:47:05 UTC 2018
2: Thu Dec 27 15:47:06 UTC 2018
3: Thu Dec 27 15:47:07 UTC 2018
.....
```

Kubernetes 日志收集

Kubernetes 集群本身不提供日志收集的解决方案，一般来说有主要的3种方案来做日志收集：

- 在节点上运行一个 agent 来收集日志
- 在 Pod 中包含一个 sidecar 容器来收集应用日志
- 直接在应用程序中将日志信息推送到采集后端

节点日志采集代理



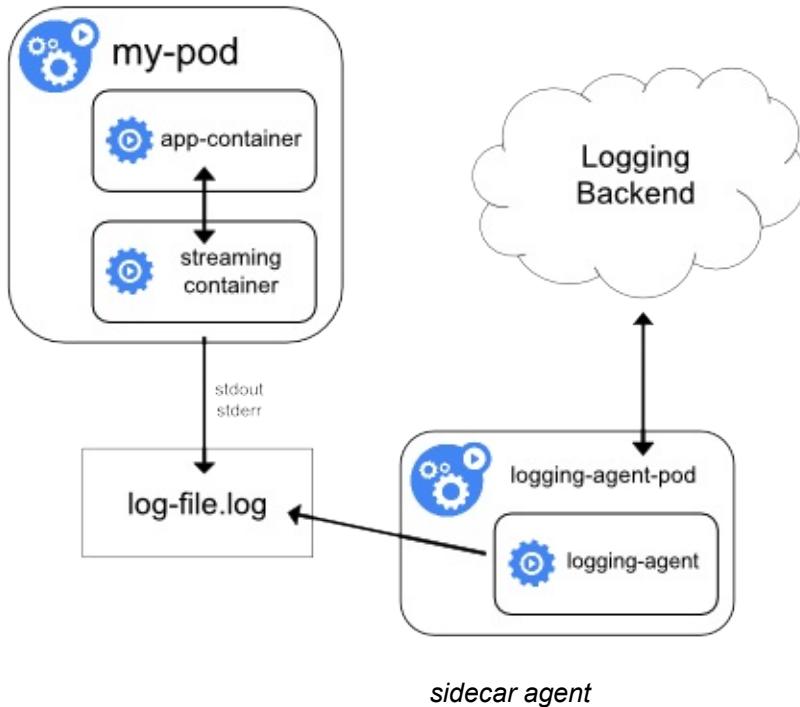
通过在每个节点上运行一个日志收集的 agent 来采集日志数据，日志采集 agent 是一种专用工具，用于将日志数据推送到统一的后端。一般来说，这种 agent 用一个容器来运行，可以访问该节点上所有应用程序容器的日志文件所在目录。

由于这种 agent 必须在每个节点上运行，所以直接使用 DaemonSet 控制器运行该应用程序即可。在节点上运行一个日志收集的 agent 这种方式是最常见的一直方法，因为它只需要在每个节点上运行一个代理程序，并不需要对节点上运行的应用程序进行更改，对应用程序没有任何侵入性，但是这种方法也仅仅适用于收集输出到 stdout 和 stderr 的应用程序日志。

以 sidecar 容器收集日志

我们看上面的图可以看到有一个明显的问题就是我们采集的日志都是通过输出到容器的 stdout 和 stderr 里面的信息，这些信息会在本地的容器对应目录中保留成 JSON 日志文件，所以直接在节点上运行一个 agent 就可以采集到日志。但是如果我们的应用程序的日志是输出到容器中的某个日志文件的话呢？这种日志数据显然只通过上面的方案是采集不到的了。

用 sidecar 容器重新输出日志



对于上面这种情况我们可以直接在 Pod 中启动另外一个 sidecar 容器，直接将应用程序的日志通过这个容器重新输出到 stdout，这样是不是通过上面的节点日志收集方案又可以完成了。

由于这个 sidecar 容器的主要逻辑就是将应用程序中的日志进行重定向打印，所以背后的逻辑非常简单，开销很小，而且由于输出到了 stdout 或者 stderr，所以我们也可以使用 `kubectl logs` 来查看日志了。

下面的示例是在 Pod 中将日志记录在了容器的两个本地文件之中：

```

apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done

```

```

volumeMounts:
- name: varlog
  mountPath: /var/log
volumes:
- name: varlog
  emptyDir: {}

```

由于 Pod 中容器的特性，我们可以利用另外一个 sidecar 容器去获取到另外容器中的日志文件，然后将日志重定向到自己的 stdout 流中，可以将上面的 YAML 文件做如下修改：(two-files-counter-pod-streaming-sidecar.yaml)

```

apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - |
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
  volumeMounts:
  - name: varlog
    mountPath: /var/log
  - name: count-log-1
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']
  volumeMounts:
  - name: varlog
    mountPath: /var/log
  - name: count-log-2
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']
  volumeMounts:
  - name: varlog
    mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}

```

直接创建上面的 Pod：

```
$ kubectl create -f two-files-counter-pod-streaming-sidecar.yaml
pod "counter" created
```

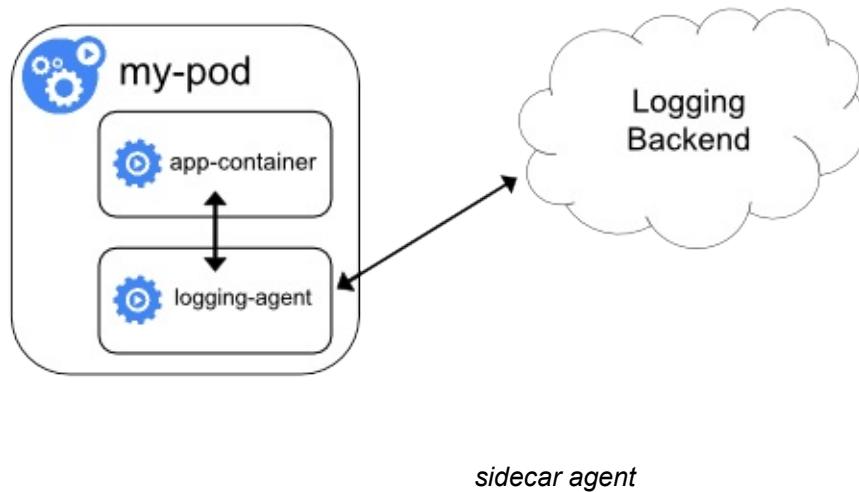
运行成功后，我们可以通过下面的命令来查看日志的信息：

```
$ kubectl logs counter count-log-1
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
$ kubectl logs counter count-log-2
Mon Jan  1 00:00:00 UTC 2001 INFO 0
Mon Jan  1 00:00:01 UTC 2001 INFO 1
Mon Jan  1 00:00:02 UTC 2001 INFO 2
...
```

这样前面节点上的日志采集 agent 就可以自动获取这些日志信息，而不需要其他配置。

这种方法虽然可以解决上面的问题，但是也有一个明显的缺陷，就是日志不仅会在原容器文件中保留下来，还会通过 stdout 输出后占用磁盘空间，这样无形中就增加了一倍磁盘空间。

使用 sidecar 运行日志采集 agent



如果你觉得在节点上运行一个日志采集的代理不够灵活的话，那么你也可以创建一个单独的日志采集代理程序的 sidecar 容器，不过需要单独配置和应用程序一起运行。

不过这样虽然更加灵活，但是在 sidecar 容器中运行日志采集代理程序会导致大量资源消耗，因为你有多少个要采集的 Pod，就需要运行多少个采集代理程序，另外还无法使用 kubectl logs 命令来访问这些日志，因为它们不受 kubelet 控制。

举个例子，你可以使用的Stackdriver，它使用fluentd作为记录剂。以下是两个可用于实现此方法的配置文件。第一个文件包含配置流利的ConfigMap。

下面是 Kubernetes 官方的一个 fluentd 的配置文件示例，使用 ConfigMap 对象来保存：

```
apiVersion: v1
kind: ConfigMap
```

```

metadata:
  name: fluentd-config
data:
  fluentd.conf: |
    <source>
      type tail
      format none
      path /var/log/1.log
      pos_file /var/log/1.log.pos
      tag count.format1
    </source>

    <source>
      type tail
      format none
      path /var/log/2.log
      pos_file /var/log/2.log.pos
      tag count.format2
    </source>

    <match **>
      type google_cloud
    </match>

```

上面的配置文件是配置收集原文件 /var/log/1.log 和 /var/log/2.log 的日志数据，然后通过 google_cloud 这个插件将数据推送到 Stackdriver 后端去。

下面是我们使用上面的配置文件在应用程序中运行一个 fluentd 的容器来读取日志数据：

```

apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - -
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
  volumeMounts:
  - name: varlog
    mountPath: /var/log
  - name: count-agent
    image: k8s.gcr.io/fluentd-gcp:1.30
    env:
    - name: FLUENTD_ARGS
      value: -c /etc/fluentd-config/fluentd.conf
  volumeMounts:

```

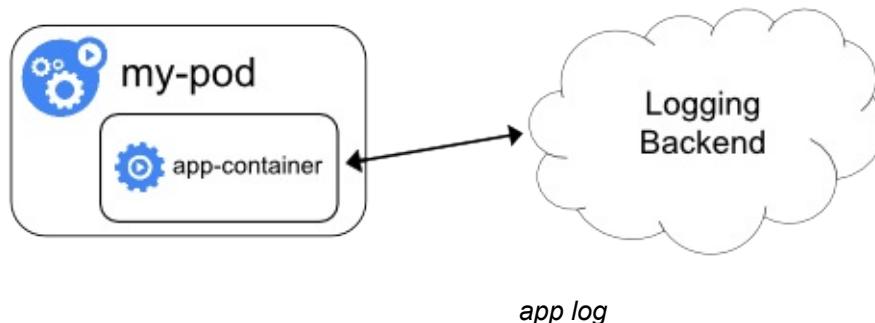
```

- name: varlog
  mountPath: /var/log
- name: config-volume
  mountPath: /etc/fluentd-config
volumes:
- name: varlog
  emptyDir: {}
- name: config-volume
  configMap:
    name: fluentd-config

```

上面的 Pod 创建完成后，容器 count-agent 就会将 count 容器中的日志进行收集然后上传。当然，这只是一个简单的示例，我们也完全可以使用其他的任何日志采集工具来替换 fluentd，比如 logstash、fluent-bit 等等。

直接从应用程序收集日志



除了上面的几种方案之外，我们也完全可以通过直接在应用程序中去显示的将日志推送到日志后端，但是这种方式需要代码层面的实现，也超出了 Kubernetes 本身的范围。

下节课我们给大家演示下具体日志收集的操作方法，或者你有更好的方案吗？

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:06:55

62. 搭建 EFK 日志系统

上节课和大家介绍了 Kubernetes 集群中的几种日志收集方案，Kubernetes 中比较流行的日志收集解决方案是 Elasticsearch、Fluentd 和 Kibana（EFK）技术栈，也是官方现在比较推荐的一种方案。

Elasticsearch 是一个实时的、分布式的可扩展的搜索引擎，允许进行全文、结构化搜索，它通常用于索引和搜索大量日志数据，也可用于搜索许多不同类型的文档。

Elasticsearch 通常与 Kibana 一起部署，Kibana 是 Elasticsearch 的一个功能强大的数据可视化 Dashboard，Kibana 允许你通过 web 界面来浏览 Elasticsearch 日志数据。

Fluentd 是一个流行的开源数据收集器，我们将在 Kubernetes 集群节点上安装 Fluentd，通过获取容器日志文件、过滤和转换日志数据，然后将数据传递到 Elasticsearch 集群，在该集群中对其进行索引和存储。

我们先来配置启动一个可扩展的 Elasticsearch 集群，然后在 Kubernetes 集群中创建一个 Kibana 应用，最后通过 DaemonSet 来运行 Fluentd，以便它在每个 Kubernetes 工作节点上都可以运行一个 Pod。

创建 Elasticsearch 集群

在创建 Elasticsearch 集群之前，我们先创建一个命名空间，我们将在其中安装所有日志相关的资源对象。

新建一个 kube-logging.yaml 文件：

```
apiVersion: v1
kind: Namespace
metadata:
  name: logging
```

然后通过 kubectl 创建该资源清单，创建一个名为 logging 的 namespace：

```
$ kubectl create -f kube-logging.yaml
namespace/logging created
$ kubectl get ns
NAME      STATUS   AGE
default   Active   244d
istio-system   Active   100d
kube-ops   Active   179d
kube-public   Active   244d
kube-system   Active   244d
logging    Active   4h
monitoring  Active   35d
```

现在创建了一个命名空间来存放我们的日志相关资源，接下来可以部署 EFK 相关组件，首先开始部署一个3节点的 Elasticsearch 集群。

这里我们使用3个 Elasticsearch Pod 来避免高可用下多节点集群中出现的“脑裂”问题，当一个或多个节点无法与其他节点通信时会产生“脑裂”，可能会出现几个主节点。

了解更多 Elasticsearch 集群脑裂问题，可以查看文档<https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-node.html#split-brain>

一个关键点是您应该将参数 `discover.zen.minimum_master_nodes=N/2+1`，其中 N 是 Elasticsearch 集群中符合主节点的节点数，比如我们这里3个节点，意味着 N 应该设置为2。这样，如果一个节点暂时与集群断开连接，则另外两个节点可以选择一个新的主节点，并且集群可以在最后一个节点尝试重新加入时继续运行，在扩展 Elasticsearch 集群时，一定要记住这个参数。

首先创建一个名为 `elasticsearch` 的无头服务，新建文件 `elasticsearch-svc.yaml`，文件内容如下：

```
kind: Service
apiVersion: v1
metadata:
  name: elasticsearch
  namespace: logging
  labels:
    app: elasticsearch
spec:
  selector:
    app: elasticsearch
  clusterIP: None
  ports:
    - port: 9200
      name: rest
    - port: 9300
      name: inter-node
```

定义了一个名为 `elasticsearch` 的 Service，指定标签 `app=elasticsearch`，当我们使用 `kubectl` 将 Elasticsearch StatefulSet 与此服务关联时，服务将返回带有标签 `app=elasticsearch` 的 Elasticsearch Pods 的 DNS A 记录，然后设置 `clusterIP=None`，将该服务设置成无头服务。最后，我们分别定义端口9200、9300，分别用于与 REST API 交互，以及用于节点间通信。

使用 `kubectl` 直接创建上面的服务资源对象：

```
$ kubectl create -f elasticsearch-svc.yaml
service/elasticsearch created
$ kubectl get services --namespace=logging
Output
NAME          TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)           AGE
elasticsearch  ClusterIP   None         <none>        9200/TCP,9300/TCP  26s
```

现在我们已经为 Pod 设置了无头服务和一个稳定的域名 `.elasticsearch.logging.svc.cluster.local`，接下来我们通过 StatefulSet 来创建具体的 Elasticsearch 的 Pod 应用。

Kubernetes StatefulSet 允许我们为 Pod 分配一个稳定的标识和持久化存储，Elasticsearch 需要稳定的存储来保证 Pod 在重新调度或者重启后的数据依然不变，所以需要使用 StatefulSet 来管理 Pod。

要了解更多关于 StatefulSet 的信息，可以查看官网关于 StatefulSet 的相关文档：<https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>。

新建名为 `elasticsearch-statefulset.yaml` 的资源清单文件，首先粘贴下面内容：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: es-cluster
  namespace: logging
spec:
  serviceName: elasticsearch
  replicas: 3
  selector:
    matchLabels:
      app: elasticsearch
  template:
    metadata:
      labels:
        app: elasticsearch
```

该内容中，我们定义了一个名为 `es-cluster` 的 StatefulSet 对象，然后定义 `serviceName=elasticsearch` 和前面创建的 Service 相关联，这可以确保使用以下 DNS 地址访问 StatefulSet 中的每一个 Pod：`es-cluster-[0,1,2].elasticsearch.logging.svc.cluster.local`，其中 [0,1,2] 对应于已分配的 Pod 序号。

然后指定3个副本，将 `matchLabels` 设置为 `app=elasticsearch`，所以 Pod 的模板部分 `.spec.template.metadata.labels` 也必须包含 `app=elasticsearch` 标签。

然后定义 Pod 模板部分内容：

```
...
spec:
  containers:
  - name: elasticsearch
    image: docker.elastic.co/elasticsearch/elasticsearch-oss:6.4.3
    resources:
      limits:
        cpu: 1000m
      requests:
        cpu: 100m
    ports:
    - containerPort: 9200
      name: rest
      protocol: TCP
    - containerPort: 9300
      name: inter-node
      protocol: TCP
    volumeMounts:
    - name: data
      mountPath: /usr/share/elasticsearch/data
  env:
  - name: cluster.name
    value: k8s-logs
  - name: node.name
    valueFrom:
```

```

    fieldRef:
      fieldPath: metadata.name
    - name: discovery.zen.ping.unicast.hosts
      value: "es-cluster-0.elasticsearch,es-cluster-1.elasticsearch,es-cluster-2.elasticsearch"
    - name: discovery.zen.minimum_master_nodes
      value: "2"
    - name: ES_JAVA_OPTS
      value: "-Xms512m -Xmx512m"

```

该部分是定义 StatefulSet 中的 Pod，我们这里使用一个 `-oss` 后缀的镜像，该镜像是 Elasticsearch 的开源版本，如果你想使用包含 `X-Pack` 之类的版本，可以去掉该后缀。然后暴露了 9200 和 9300 两个端口，注意名称要和上面定义的 Service 保持一致。然后通过 `volumeMount` 声明了数据持久化目录，下面我们再来定义 `VolumeClaims`。最后就是我们在容器中设置的一些环境变量了：

- `cluster.name`: Elasticsearch 集群的名称，我们这里命名成 `k8s-logs`。
- `node.name`: 节点的名称，通过 `metadata.name` 来获取。这将解析为 `es-cluster-[0,1,2]`，取决于节点的指定顺序。
- `discovery.zen.ping.unicast.hosts`: 此字段用于设置在 Elasticsearch 集群中节点相互连接的发现方法。我们使用 `unicastDiscovery` 方式，它为我们的集群指定了一个静态主机列表。由于我们之前配置的无头服务，我们的 Pod 具有唯一的 DNS 域 `es-cluster-[0,1,2].elasticsearch.logging.svc.cluster.local`，因此我们相应地设置此变量。由于都在同一个 namespace 下面，所以我们可以将其缩短为 `es-cluster-[0,1,2].elasticsearch`。要了解有关 Elasticsearch 发现的更多信息，请参阅 Elasticsearch 官方文档：<https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-discovery.html>。
- `discovery.zen.minimum_master_nodes`: 我们将其设置为 $(N/2) + 1$ ，`N` 是我们的群集中符合主节点的节点的数量。我们有 3 个 Elasticsearch 节点，因此我们将此值设置为 2（向下舍入到最接近的整数）。要了解有关此参数的更多信息，请参阅官方 Elasticsearch 文档：<https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-node.html#split-brain>。
- `ES_JAVA_OPTS`: 这里我们设置为 `-Xms512m -Xmx512m`，告诉 JVM 使用 512 MB 的最小和最大堆。您应该根据群集的资源可用性和需求调整这些参数。要了解更多信息，请参阅设置堆大小的相关文档：<https://www.elastic.co/guide/en/elasticsearch/reference/current/heap-size.html>。

接下来添加关于 `initContainer` 的内容：

```

    ...
    initContainers:
    - name: fix-permissions
      image: busybox
      command: ["sh", "-c", "chown -R 1000:1000 /usr/share/elasticsearch/data"]
      securityContext:
        privileged: true
      volumeMounts:
      - name: data
        mountPath: /usr/share/elasticsearch/data
    - name: increase-vm-max-map
      image: busybox
      command: ["sysctl", "-w", "vm.max_map_count=262144"]
      securityContext:
        privileged: true

```

```

- name: increase-fd-ulimit
  image: busybox
  command: ["sh", "-c", "ulimit -n 65536"]
  securityContext:
    privileged: true

```

这里我们定义了几个在主应用程序之前运行的 Init 容器，这些初始容器按照定义的顺序依次执行，执行完成后才会启动主应用容器。

第一个名为 fix-permissions 的容器用来运行 chown 命令，将 Elasticsearch 数据目录的用户和组更改为 1000:1000（Elasticsearch 用户的 UID）。因为默认情况下，Kubernetes 用 root 用户挂载数据目录，这会使得 Elasticsearch 无法方法该数据目录，可以参考 Elasticsearch 生产中的一些默认注意事项相关文档说明：https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html#_notes_for_production_use_and_defaults。

第二个名为 increase-vm-max-map 的容器用来增加操作系统对 mmap 计数的限制，默认情况下该值可能太低，导致内存不足的错误，要了解更多关于该设置的信息，可以查看 Elasticsearch 官方文档说明：<https://www.elastic.co/guide/en/elasticsearch/reference/current/vm-max-map-count.html>。

最后一个初始化容器是用来执行 ulimit 命令增加打开文件描述符的最大数量的。

此外 [Elasticsearch Notes for Production Use](#) 文档还提到了由于性能原因最好禁用 swap，当然对于 Kubernetes 集群而言，最好也是禁用 swap 分区。

现在我们已经定义了主应用容器和它之前运行的 Init Containers 来调整一些必要的系统参数，接下来我们可以添加数据目录的持久化相关的配置，在 StatefulSet 中，使用 volumeClaimTemplates 来定义 volume 模板即可：

```

...
volumeClaimTemplates:
- metadata:
  name: data
  labels:
    app: elasticsearch
spec:
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: es-data-db
  resources:
    requests:
      storage: 50Gi

```

我们这里使用 volumeClaimTemplates 来定义持久化模板，Kubernetes 会使用它为 Pod 创建 PersistentVolume，设置访问模式为 ReadWriteOnce，这意味着它只能被 mount 到单个节点上进行读写，然后最重要的是使用了一个名为 es-data-db 的 StorageClass 对象，所以我们需要提前创建该对象，我们这里使用的 NFS 作为存储后端，所以需要安装一个对应的 provisioner 驱动，前面关于 StorageClass 的课程中已经和大家介绍过方法，新建一个 `elasticsearch-storageclass.yaml` 的文件，文件内容如下：

```

apiVersion: storage.k8s.io/v1
kind: StorageClass

```

```

metadata:
  name: es-data-db
  provisioner: fuseim.pri/ifs # 该值需要和 provisioner 配置的保持一致

```

最后，我们指定了每个 PersistentVolume 的大小为 50GB，我们可以根据自己的实际需要进行调整该值。最后，完整的 Elasticsearch StatefulSet 资源清单文件内容如下：

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: es-cluster
  namespace: logging
spec:
  serviceName: elasticsearch
  replicas: 3
  selector:
    matchLabels:
      app: elasticsearch
  template:
    metadata:
      labels:
        app: elasticsearch
    spec:
      containers:
        - name: elasticsearch
          image: docker.elastic.co/elasticsearch/elasticsearch-oss:6.4.3
          resources:
            limits:
              cpu: 1000m
            requests:
              cpu: 100m
          ports:
            - containerPort: 9200
              name: rest
              protocol: TCP
            - containerPort: 9300
              name: inter-node
              protocol: TCP
          volumeMounts:
            - name: data
              mountPath: /usr/share/elasticsearch/data
        env:
          - name: cluster.name
            value: k8s-logs
          - name: node.name
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: discovery.zen.ping.unicast.hosts
            value: "es-cluster-0.elasticsearch,es-cluster-1.elasticsearch,es-cluster-2.elasticsearch"
          - name: discovery.zen.minimum_master_nodes
            value: "2"
          - name: ES_JAVA_OPTS
            value: "-Xms512m -Xmx512m"
      initContainers:
        - name: fix-permissions
          image: busybox

```

```

command: ["sh", "-c", "chown -R 1000:1000 /usr/share/elasticsearch/data"]
securityContext:
  privileged: true
volumeMounts:
- name: data
  mountPath: /usr/share/elasticsearch/data
- name: increase-vm-max-map
  image: busybox
  command: ["sysctl", "-w", "vm.max_map_count=262144"]
  securityContext:
    privileged: true
- name: increase-fd-ulimit
  image: busybox
  command: ["sh", "-c", "ulimit -n 65536"]
  securityContext:
    privileged: true
volumeClaimTemplates:
- metadata:
    name: data
    labels:
      app: elasticsearch
spec:
  accessModes: [ "ReadWriteOnce" ]
  storageClassName: es-data-db
  resources:
    requests:
      storage: 100Gi

```

现在直接使用 kubectl 工具部署即可：

```

$ kubectl create -f elasticsearch-storageclass.yaml
storageclass.storage.k8s.io "es-data-db" created
$ kubectl create -f elasticsearch-statefulset.yaml
statefulset.apps/es-cluster created

```

添加成功后，可以看到 logging 命名空间下面的所有的资源对象：

```

$ kubectl get sts -n logging
NAME      DESIRED   CURRENT   AGE
es-cluster   3         3        20h
$ kubectl get pods -n logging
NAME          READY   STATUS    RESTARTS   AGE
es-cluster-0   1/1     Running   0          20h
es-cluster-1   1/1     Running   0          20h
es-cluster-2   1/1     Running   0          20h
$ kubectl get svc -n logging
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
elasticsearch   ClusterIP   None      <none>      9200/TCP,9300/TCP   20h

```

Pods 部署完成后，我们可以通过请求一个 REST API 来检查 Elasticsearch 集群是否正常运行。使用下面的命令将本地端口9200转发到 Elasticsearch 节点（如es-cluster-0）对应的端口：

```

$ kubectl port-forward es-cluster-0 9200:9200 --namespace=logging
Forwarding from 127.0.0.1:9200 -> 9200
Forwarding from [::1]:9200 -> 9200

```

然后，在另外的终端窗口中，执行如下请求：

```
$ curl http://localhost:9200/_cluster/state?pretty
```

正常来说，应该会看到类似于如下的信息：

```
{
  "cluster_name" : "k8s-logs",
  "compressed_size_in_bytes" : 348,
  "cluster_uuid" : "QD06dK7CQgids-GQZooNVw",
  "version" : 3,
  "state_uuid" : "mjNIWXAzQVuxNNOQ7xR-qg",
  "master_node" : "IdM5B7cUQWqFgIHXBp0JDg",
  "blocks" : { },
  "nodes" : {
    "u7DoTpMmSCixOoictzHItA" : {
      "name" : "es-cluster-1",
      "ephemeral_id" : "Z1BfInXKRMC4RvEACHIVdg",
      "transport_address" : "10.244.4.191:9300",
      "attributes" : { }
    },
    "IdM5B7cUQWqFgIHXBp0JDg" : {
      "name" : "es-cluster-0",
      "ephemeral_id" : "JTk1FDdFQuWbSFAtBxdxAQ",
      "transport_address" : "10.244.2.215:9300",
      "attributes" : { }
    },
    "R8E7xcSUSbGbgrhAdyAKmQ" : {
      "name" : "es-cluster-2",
      "ephemeral_id" : "9wv6ke71Qqy9vk2LgJTqaA",
      "transport_address" : "10.244.40.4:9300",
      "attributes" : { }
    }
  },
  ...
}
```

看到上面的信息就表明我们名为 k8s-logs 的 Elasticsearch 集群成功创建了3个节点：es-cluster-0, es-cluster-1, 和es-cluster-2，当前主节点是 es-cluster-0。

创建 Kibana 服务

Elasticsearch 集群启动成功了，接下来我们可以来部署 Kibana 服务，新建一个名为 kibana.yaml 的文件，对应的文件内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: kibana
  namespace: logging
  labels:
    app: kibana
spec:
  ports:
    - port: 5601
```

```

type: NodePort
selector:
  app: kibana

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kibana
  namespace: logging
  labels:
    app: kibana
spec:
  selector:
    matchLabels:
      app: kibana
  template:
    metadata:
      labels:
        app: kibana
  spec:
    containers:
      - name: kibana
        image: docker.elastic.co/kibana/kibana-oss:6.4.3
        resources:
          limits:
            cpu: 1000m
        requests:
          cpu: 100m
    env:
      - name: ELASTICSEARCH_URL
        value: http://elasticsearch:9200
    ports:
      - containerPort: 5601

```

上面我们定义了两个资源对象，一个 Service 和 Deployment，为了测试方便，我们将 Service 设置为了 NodePort 类型，Kibana Pod 中配置都比较简单，唯一需要注意的是我们使用 ELASTICSEARCH_URL 这个环境变量来设置 Elasticsearch 集群的端点和端口，直接使用 Kubernetes DNS 即可，此端点对应服务名称为 elasticsearch，由于是一个 headless service，所以该域将解析为 3 个 Elasticsearch Pod 的 IP 地址列表。

配置完成后，直接使用 kubectl 工具创建：

```

$ kubectl create -f kibana.yaml
service/kibana created
deployment.apps/kibana created

```

创建完成后，可以查看 Kibana Pod 的运行状态：

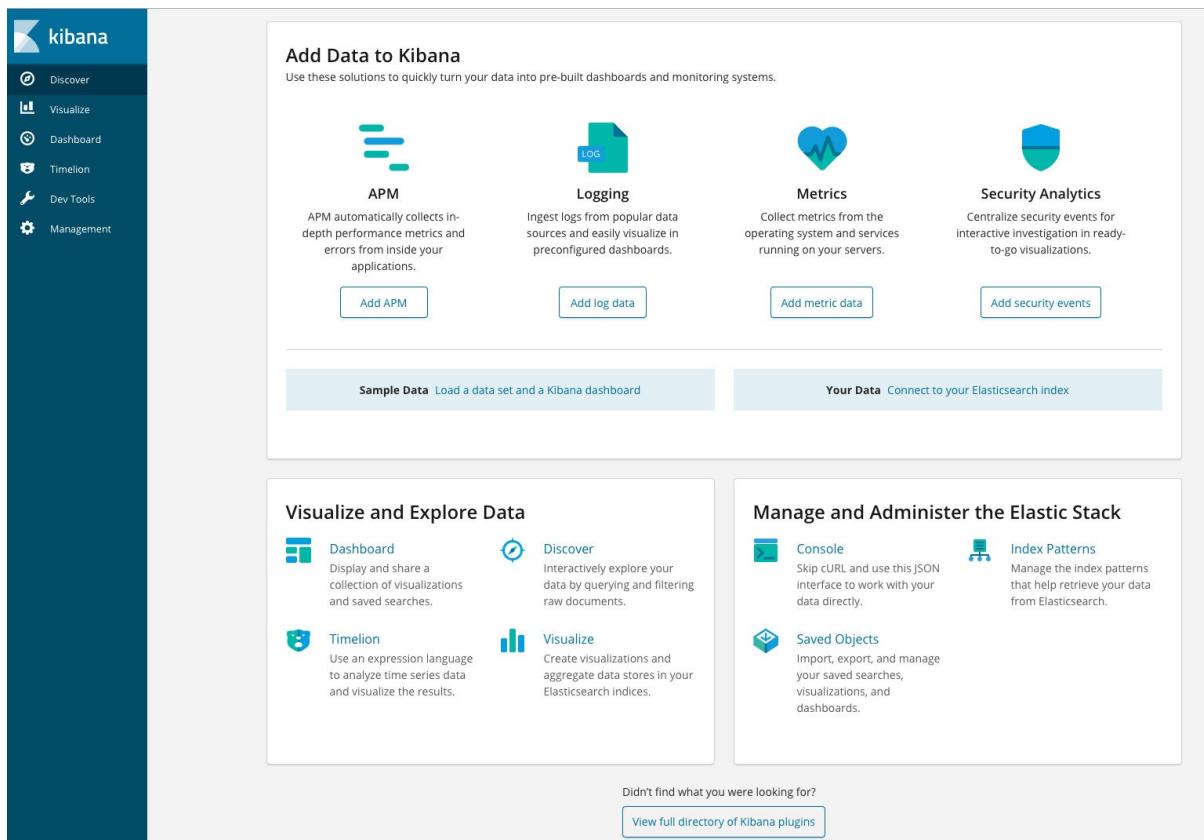
```

$ kubectl get pods --namespace=logging
NAME           READY   STATUS    RESTARTS   AGE
es-cluster-0   1/1     Running   0          20h
es-cluster-1   1/1     Running   0          20h
es-cluster-2   1/1     Running   0          20h
kibana-7558d4dc4d-5mqdz 1/1     Running   0          20h
$ kubectl get svc --namespace=logging

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
elasticsearch	ClusterIP	None	<none>	9200/TCP, 9300/TCP	20h
kibana	NodePort	10.105.208.253	<none>	5601:31816/TCP	20h

如果 Pod 已经是 Running 状态了，证明应用已经部署成功了，然后可以通过 NodePort 来访问 Kibana 这个服务，在浏览器中打开 `http://<任意节点IP>:31816` 即可，如果看到如下欢迎界面证明 Kibana 已经成功部署到了 Kubernetes 集群之中。



kibana welcome

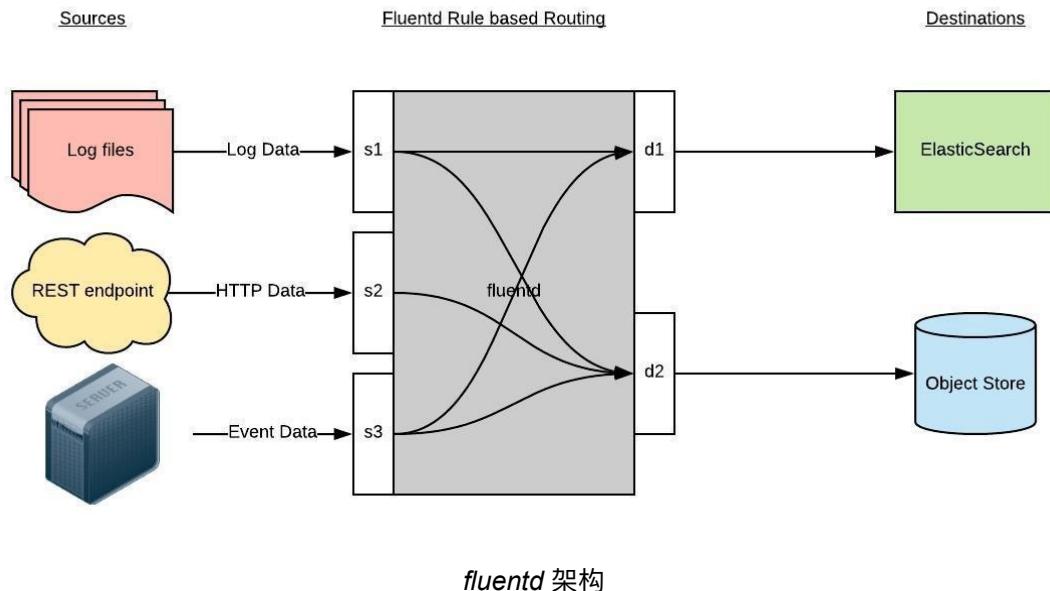
部署 Fluentd

Fluentd 是一个高效的日志聚合器，是用 Ruby 编写的，并且可以很好地扩展。对于大部分企业来说，Fluentd 足够高效并且消耗的资源相对较少，另外一个工具 Fluent-bit 更轻量级，占用资源更少，但是插件相对 Fluentd 来说不够丰富，所以整体来说，Fluentd 更加成熟，使用更加广泛，所以我们这里也同样使用 Fluentd 来作为日志收集工具。

工作原理

Fluentd 通过一组给定的数据源抓取日志数据，处理后（转换成结构化的数据格式）将它们转发给其他服务，比如 Elasticsearch、对象存储等等。Fluentd 支持超过 300 个日志存储和分析服务，所以在这些方面是非常灵活的。主要运行步骤如下：

- 首先 Fluentd 从多个日志源获取数据
- 结构化并且标记这些数据
- 然后根据匹配的标签将数据发送到多个目标服务去



配置

一般来说我们是通过一个配置文件来告诉 Fluentd 如何采集、处理数据的，下面简单和大家介绍下 Fluentd 的配置方法。

日志源配置

比如我们这里为了收集 Kubernetes 节点上的所有容器日志，就需要做如下的日志源配置：

```

<source>
@id fluentd-containers.log
@type tail
path /var/log/containers/*.log
pos_file /var/log/fluentd-containers.log.pos
time_format %Y-%m-%dT%H:%M:%S.%NZ
tag raw.kubernetes.-
format json
read_from_head true
</source>

```

上面配置部分参数说明如下：

- id: 表示引用该日志源的唯一标识符，该标识可用于进一步过滤和路由结构化日志数据
- type: Fluentd 内置的指令，tail 表示 Fluentd 从上次读取的位置通过 tail 不断获取数据，另外一个是 http 表示通过一个 GET 请求来收集数据。
- path: tail 类型下的特定参数，告诉 Fluentd 采集 /var/log/containers 目录下的所有日志，这是 docker 在 Kubernetes 节点上用来存储运行容器 stdout 输出日志数据的目录。
- pos_file: 检查点，如果 Fluentd 程序重新启动了，它将使用此文件中的位置来恢复日志数据收集。
- tag: 用来将日志源与目标或者过滤器匹配的自定义字符串，Fluentd 匹配源/目标标签来路由日志数据。

路由配置

上面是日志源的配置，接下来看看如何将日志数据发送到 Elasticsearch：

```
<match **>

@id elasticsearch

@type elasticsearch

@log_level info

include_tag_key true

type_name fluentd

host "#{ENV['OUTPUT_HOST']}"

port "#{ENV['OUTPUT_PORT']}"

logstash_format true

<buffer>

@type file

path /var/log/fluentd-buffers/kubernetes.system.buffer

flush_mode interval

retry_type exponential_backoff

flush_thread_count 2

flush_interval 5s

retry_forever

retry_max_interval 30

chunk_limit_size "#{ENV['OUTPUT_BUFFER_CHUNK_LIMIT']}
```

```

queue_limit_length "#{ENV['OUTPUT_BUFFER_QUEUE_LIMIT']}"

overflow_action block

</buffer>

```

- **match**: 标识一个目标标签，后面是一个匹配日志源的正则表达式，我们这里想要捕获所有的日志并将它们发送给 Elasticsearch，所以需要配置成 `**`。
- **id**: 目标的一个唯一标识符。
- **type**: 支持的输出插件标识符，我们这里要输出到 Elasticsearch，所以配置成 `elasticsearch`，这是 Fluentd 的一个内置插件。
- **log_level**: 指定要捕获的日志级别，我们这里配置成 `info`，表示任何该级别或者该级别以上 (INFO、WARNING、ERROR) 的日志都将被路由到 Elasticsearch。
- **host/port**: 定义 Elasticsearch 的地址，也可以配置认证信息，我们的 Elasticsearch 不需要认证，所以这里直接指定 host 和 port 即可。
- **logstash_format**: Elasticsearch 服务对日志数据构建反向索引进行搜索，将 `logstash_format` 设置为 `true`，Fluentd 将会以 logstash 格式来转发结构化的日志数据。
- **Buffer**: Fluentd 允许在目标不可用时进行缓存，比如，如果网络出现故障或者 Elasticsearch 不可用的时候。缓冲区配置也有助于降低磁盘的 IO。

安装

要收集 Kubernetes 集群的日志，直接用 DaemonSet 控制器来部署 Fluentd 应用，这样，它就可以从 Kubernetes 节点上采集日志，确保在集群中的每个节点上始终运行一个 Fluentd 容器。当然可以直接使用 Helm 来进行一键安装，为了能够了解更多实现细节，我们这里还是采用手动方法来进行安装。

首先，我们通过 ConfigMap 对象来指定 Fluentd 配置文件，新建 `fluentd-configmap.yaml` 文件，文件内容如下：

```

kind: ConfigMap
apiVersion: v1
metadata:
  name: fluentd-config
  namespace: logging
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
data:
  system.conf: |-
    <system>
      root_dir /tmp/fluentd-buffers/
    </system>
  containers.input.conf: |-
    <source>
      @id fluentd-containers.log
      @type tail
      path /var/log/containers/*.log
      pos_file /var/log/es-containers.log.pos
      time_format %Y-%m-%dT%H:%M:%S.%NZ
      localtime
      tag raw.kubernetes.*

```

```

format json
read_from_head true
</source>
# Detect exceptions in the log output and forward them as one log entry.
<match raw.kubernetes.*>
  @id raw.kubernetes
  @type detect_exceptions
  remove_tag_prefix raw
  message log
  stream stream
  multiline_flush_interval 5
  max_bytes 500000
  max_lines 1000
</match>
system.input.conf: |-
# Logs from systemd-journal for interesting services.
<source>
  @id journald-docker
  @type systemd
  filters [{"_SYSTEMD_UNIT": "docker.service"}]
  <storage>
    @type local
    persistent true
  </storage>
  read_from_head true
  tag docker
</source>
<source>
  @id journald-kubelet
  @type systemd
  filters [{"_SYSTEMD_UNIT": "kubelet.service"}]
  <storage>
    @type local
    persistent true
  </storage>
  read_from_head true
  tag kubelet
</source>
forward.input.conf: |-
# Takes the messages sent over TCP
<source>
  @type forward
</source>
output.conf: |-
# Enriches records with Kubernetes metadata
<filter kubernetes.*>
  @type kubernetes_metadata
</filter>
<match **>
  @id elasticsearch
  @type elasticsearch
  @log_level info
  include_tag_key true
  host elasticsearch
  port 9200
  logstash_format true
  request_timeout 30s
  <buffer>
    @type file
    path /var/log/fluentd-buffers/kubernetes.system.buffer

```

```

    flush_mode interval
    retry_type exponential_backoff
    flush_thread_count 2
    flush_interval 5s
    retry_forever
    retry_max_interval 30
    chunk_limit_size 2M
    queue_limit_length 8
    overflow_action block
  </buffer>
</match>

```

上面配置文件中我们配置了 docker 容器日志目录以及 docker、kubelet 应用的日志的收集，收集到数据经过处理后发送到 elasticsearch:9200 服务。

然后新建一个 fluentd-daemonset.yaml 的文件，文件内容如下：

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: fluentd-es
  namespace: logging
  labels:
    k8s-app: fluentd-es
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
  ---
  kind: ClusterRole
  apiVersion: rbac.authorization.k8s.io/v1
  metadata:
    name: fluentd-es
    labels:
      k8s-app: fluentd-es
      kubernetes.io/cluster-service: "true"
      addonmanager.kubernetes.io/mode: Reconcile
  rules:
  - apiGroups:
    - ""
    resources:
    - "namespaces"
    - "pods"
    verbs:
    - "get"
    - "watch"
    - "list"
  ---
  kind: ClusterRoleBinding
  apiVersion: rbac.authorization.k8s.io/v1
  metadata:
    name: fluentd-es
    labels:
      k8s-app: fluentd-es
      kubernetes.io/cluster-service: "true"
      addonmanager.kubernetes.io/mode: Reconcile
  subjects:
  - kind: ServiceAccount
    name: fluentd-es
    namespace: logging

```

```

apiGroup: ""
roleRef:
  kind: ClusterRole
  name: fluentd-es
  apiGroup: ""

---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-es
  namespace: logging
  labels:
    k8s-app: fluentd-es
    version: v2.0.4
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
spec:
  selector:
    matchLabels:
      k8s-app: fluentd-es
      version: v2.0.4
  template:
    metadata:
      labels:
        k8s-app: fluentd-es
        kubernetes.io/cluster-service: "true"
        version: v2.0.4
      # This annotation ensures that fluentd does not get evicted if the node
      # supports critical pod annotation based priority scheme.
      # Note that this does not guarantee admission on the nodes (#40573).
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ''
  spec:
    serviceAccountName: fluentd-es
    containers:
      - name: fluentd-es
        image: cnych/fluentd-elasticsearch:v2.0.4
        env:
          - name: FLUENTD_ARGS
            value: --no-supervisor -q
        resources:
          limits:
            memory: 500Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
          - name: varlog
            mountPath: /var/log
          - name: varlibdockercontainers
            mountPath: /data/docker/containers
            readOnly: true
          - name: config-volume
            mountPath: /etc/fluent/config.d
    nodeSelector:
      beta.kubernetes.io/fluentd-ds-ready: "true"
    tolerations:
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule

```

```

terminationGracePeriodSeconds: 30
volumes:
- name: varlog
  hostPath:
    path: /var/log
- name: varlibdockercontainers
  hostPath:
    path: /data/docker/containers
- name: config-volume
  configMap:
    name: fluentd-config

```

我们将上面创建的 fluentd-config 这个 ConfigMap 对象通过 volumes 挂载到了 Fluentd 容器中，另外为了能够灵活控制哪些节点的日志可以被收集，所以我们这里还添加了一个 nodSelector 属性：

```

nodeSelector:
  beta.kubernetes.io/fluentd-ds-ready: "true"

```

意思就是要想采集节点的日志，那么我们就需要给节点打上上面的标签，比如我们这里3个节点都打上了该标签：

```

$ kubectl get nodes --show-labels
NAME      STATUS   ROLES      AGE       VERSION   LABELS
master    Ready    master    245d     v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/fluentd-ds-ready=true,beta.kubernetes.io/os=linux,kubernetes.io/hostname=master,node-role.kubernetes.io/master=
node02    Ready    <none>    165d     v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/fluentd-ds-ready=true,beta.kubernetes.io/os=linux,com=youdianzhishi,course=k8s,kubernetes.io/hostname=node02
node03    Ready    <none>    225d     v1.10.0   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/fluentd-ds-ready=true,beta.kubernetes.io/os=linux,jnlp=haimaxy,kubernetes.io/hostname=node03

```

另外由于我们的集群使用的是 kubeadm 搭建的，默认情况下 master 节点有污点，所以要想也收集 master 节点的日志，则需要添加上容忍：

```

tolerations:
- key: node-role.kubernetes.io/master
  operator: Exists
  effect: NoSchedule

```

另外需要注意的地方是，我这里的测试环境更改了 docker 的根目录：

```

$ docker info
...
Docker Root Dir: /data/docker
...

```

所以上面要获取 docker 的容器目录需要改成 /data/docker/containers，这个地方非常重要，当然如果你没有更改 docker 根目录则使用默认的 /var/lib/docker/containers 目录即可。

分别创建上面的 ConfigMap 对象和 DaemonSet:

```
$ kubectl create -f fluentd-configmap.yaml
configmap "fluentd-config" created
$ kubectl create -f fluentd-daemonset.yaml
serviceaccount "fluentd-es" created
clusterrole.rbac.authorization.k8s.io "fluentd-es" created
clusterrolebinding.rbac.authorization.k8s.io "fluentd-es" created
daemonset.apps "fluentd-es" created
```

创建完成后，查看对应的 Pods 列表，检查是否部署成功：

```
$ kubectl get pods -n logging
NAME                      READY   STATUS    RESTARTS   AGE
es-cluster-0              1/1     Running   0          1d
es-cluster-1              1/1     Running   0          1d
es-cluster-2              1/1     Running   0          1d
fluentd-es-2z9jg          1/1     Running   1          35s
fluentd-es-6dfdd          1/1     Running   0          35s
fluentd-es-bfk7g          1/1     Running   0          35s
kibana-7558d4dc4d-5mqdz  1/1     Running   0          1d
```

Fluentd 启动成功后，我们可以前往 Kibana 的 Dashboard 页面中，点击左侧的 Discover，可以看到如下配置页面：

The screenshot shows the 'Management / Kibana' interface. On the left, there are tabs for 'Index Patterns', 'Saved Objects', and 'Advanced Settings'. A 'Warning' message states: 'No default index pattern. You must select or create one to continue.' The main area is titled 'Create index pattern' and contains the following steps:

- Step 1 of 2: Define index pattern**
- Index pattern:** A text input field containing 'index-name-*'.
- Help text:** 'You can use a * as a wildcard in your index pattern. You can't use spaces or the characters \, /, ?, ", <, >, |.'
- Next step:** A button at the bottom right of the form.
- Indices list:** A list of indices: 'logstash-2018.12.28', 'logstash-2018.12.29', and 'logstash-2019.01.15'.
- Rows per page:** A dropdown menu set to '10'.

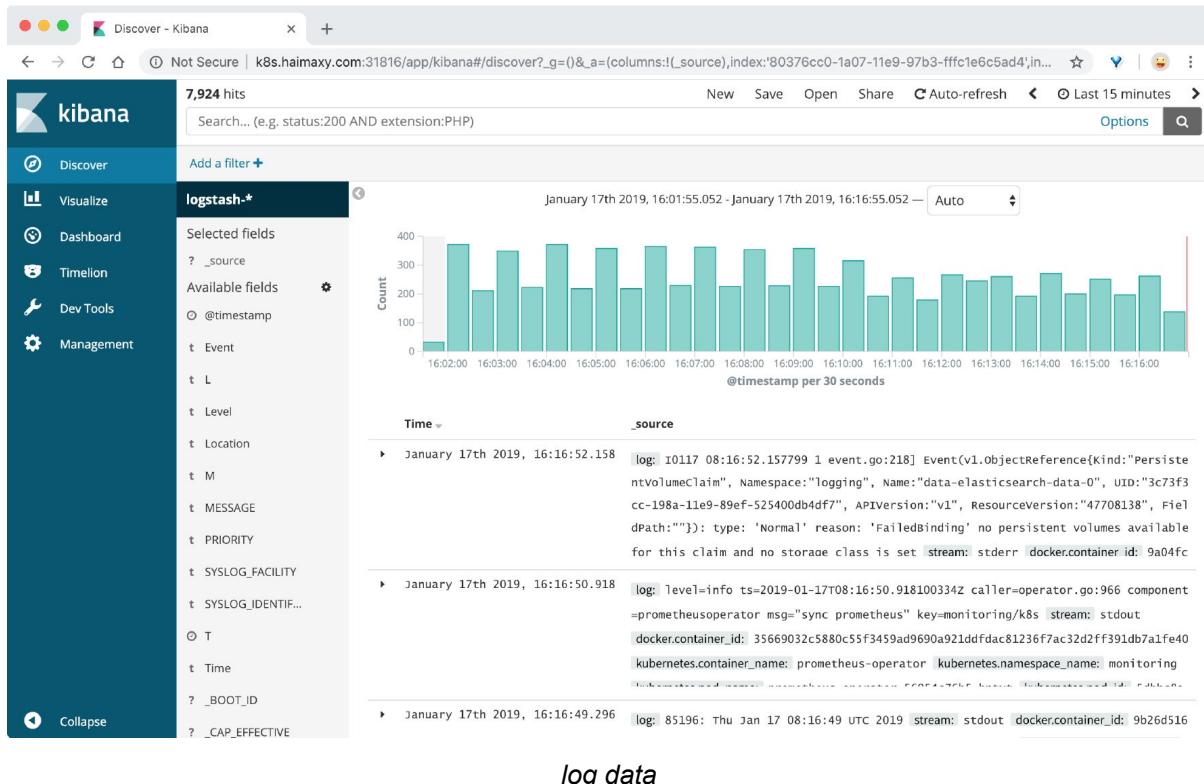
create index

在这里可以配置我们需要的 Elasticsearch 索引，前面 Fluentd 配置文件中我们采集的日志使用的是 logstash 格式，这里只需要在文本框中输入 `logstash-*` 即可匹配到 Elasticsearch 集群中的所有日志数据，然后点击下一步，进入以下页面：

The screenshot shows the 'Create index pattern' interface in Kibana. On the left, there is a 'Warning' message: 'No default index pattern. You must select or create one to continue.' At the top, there are tabs for 'Management / Kibana', 'Index Patterns', 'Saved Objects', and 'Advanced Settings'. The main area is titled 'Create index pattern' with the sub-section 'Step 2 of 2: Configure settings'. It displays the message: 'You've defined `logstash-*` as your index pattern. Now you can specify some settings before we create it.' Below this is a dropdown menu labeled 'Time Filter field name' with the value '@timestamp' selected. A 'Refresh' button is next to the dropdown. A note below says: 'The Time Filter will use this field to filter your data by time. You can choose not to have a time field, but you will not be able to narrow down your data by a time range.' There is also a link to 'Show advanced options'. At the bottom right are 'Back' and 'Create index pattern' buttons.

index config

在该页面中配置使用哪个字段按时间过滤日志数据，在下拉列表中，选择 `@timestamp` 字段，然后点击 `Create index pattern`，创建完成后，点击左侧导航菜单中的 `Discover`，然后就可以看到一些直方图和最近采集到的日志数据了：



测试

现在我们来将上一节课的计数器应用部署到集群中，并在 Kibana 中来查找该日志数据。

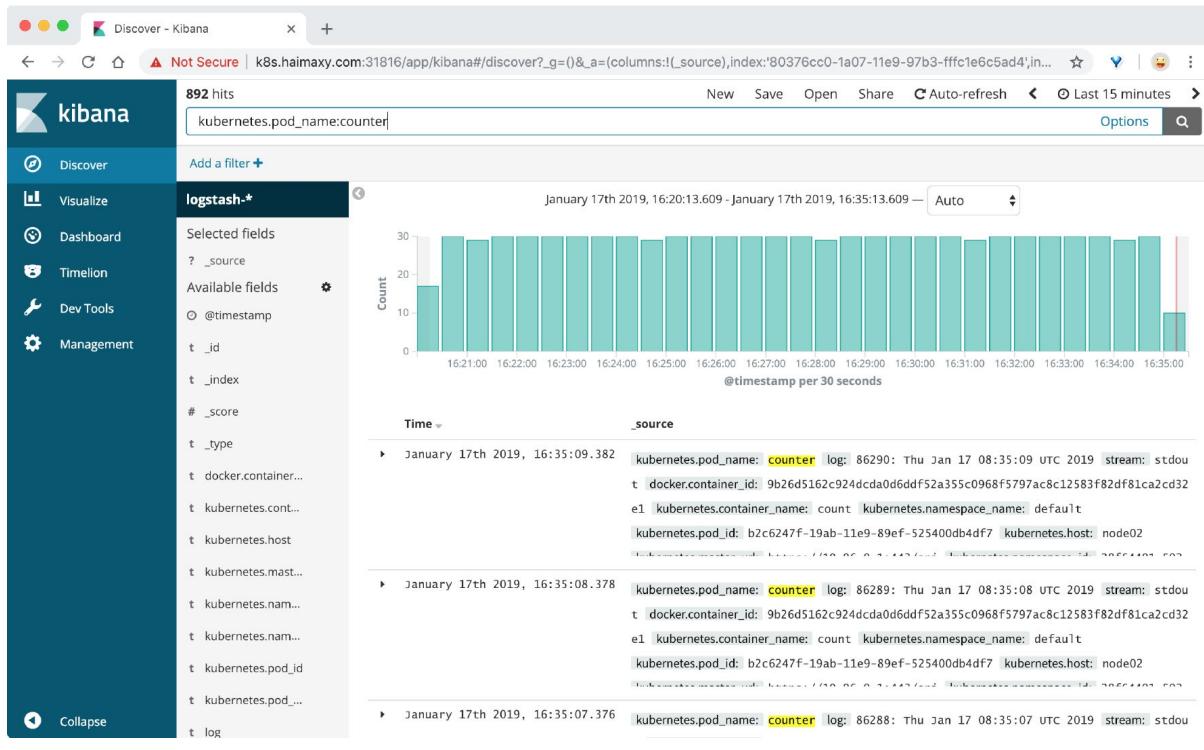
新建 counter.yaml 文件，文件内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c,
           'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

该 Pod 只是简单将日志信息打印到 stdout，所以正常来说 Fluentd 会收集到这个日志数据，在 Kibana 中也就可以找到对应的日志数据了，使用 kubectl 工具创建该 Pod：

```
$ kubectl create -f counter.yaml
```

Pod 创建并运行后，回到 Kibana Dashboard 页面，在上面的 Discover 页面搜索栏中输入 `kubernetes.pod name:counter`，就可以过滤 Pod 名为 counter 的日志数据：

*counter log data*

我们也可以通过其他元数据来过滤日志数据，比如 您可以单击任何日志条目以查看其他元数据，如容器名称，Kubernetes 节点，命名空间等。

到这里，我们就在 Kubernetes 集群上成功部署了 EFK，要了解如何使用 Kibana 进行日志数据分析，可以参考 Kibana 用户指南文档：<https://www.elastic.co/guide/en/kibana/current/index.html>

当然对于在生产环境上使用 Elasticsearch 或者 Fluentd，还需要结合实际的环境做一系列的优化工作，本文中涉及到的资源清单文件都可以在<https://github.com/cnzych/kubernetes-learning/tree/master/efkdemo>找到。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-04-24 15:29:51

基于 Jenkins 的 CI/CD (一)

前面的课程中我们学习了持久化数据存储在 Kubernetes 中的使用方法，其实接下来按照我们的课程进度来说应该是讲解服务发现这一部分的内容的，但是最近有很多同学要求我先讲解下 CI/CD 这块的内容，所以我们先把这块内容提前来讲解了。提到基于 Kubernetes 的 CI/CD，可以使用的工具有很多，比如 Jenkins、Gitlab CI 已经新兴的 drone 之类的，我们这里会使用大家最为熟悉的 Jenkins 来做 CI/CD 的工具。

安装

听我们课程的大部分同学应该都或多或少的听说过 Jenkins，我们这里就不再去详细讲述什么是 Jenkins 了，直接进入正题，后面我们会单独的关于 Jenkins 的学习课程，想更加深入学习的同学也可以关注下。既然要基于 Kubernetes 来做 CI/CD，当然我们这里需要将 Jenkins 安装到 Kubernetes 集群当中，新建一个 Deployment：(jenkins2.yaml)

```
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: jenkins2
  namespace: kube-ops
spec:
  template:
    metadata:
      labels:
        app: jenkins2
    spec:
      terminationGracePeriodSeconds: 10
      serviceAccount: jenkins2
      containers:
        - name: jenkins
          image: jenkins/jenkins:lts
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
              name: web
              protocol: TCP
            - containerPort: 50000
              name: agent
              protocol: TCP
      resources:
        limits:
          cpu: 1000m
          memory: 1Gi
        requests:
          cpu: 500m
          memory: 512Mi
      livenessProbe:
        httpGet:
          path: /login
          port: 8080
        initialDelaySeconds: 60
```

```

    timeoutSeconds: 5
    failureThreshold: 12
  readinessProbe:
    httpGet:
      path: /login
      port: 8080
    initialDelaySeconds: 60
    timeoutSeconds: 5
    failureThreshold: 12
  volumeMounts:
    - name: jenkinshome
      subPath: jenkins2
      mountPath: /var/jenkins_home
  env:
    - name: LIMITS_MEMORY
      valueFrom:
        resourceFieldRef:
          resource: limits.memory
          divisor: 1Mi
    - name: JAVA_OPTS
      value: -Xmx$(LIMITS_MEMORY)m -XshowSettings:vm -Dhudson.slaves.NodeProvisioner.initialDelay=0 -Dhudson.slaves.NodeProvisioner.MARGIN=50 -Dhudson.slaves.NodeProvisioner.MARGIN0=0.85 -Duser.timezone=Asia/Shanghai
  securityContext:
    fsGroup: 1000
  volumes:
    - name: jenkinshome
      persistentVolumeClaim:
        claimName: opspvc

---  

apiVersion: v1
kind: Service
metadata:
  name: jenkins2
  namespace: kube-ops
  labels:
    app: jenkins2
spec:
  selector:
    app: jenkins2
  type: NodePort
  ports:
    - name: web
      port: 8080
      targetPort: web
      nodePort: 30002
    - name: agent
      port: 50000
      targetPort: agent

```

为了方便演示，我们把本节课所有的对象资源都放置在一个名为 kube-ops 的 namespace 下面，所以我们需要添加创建一个 namespace：

```
$ kubectl create namespace kube-ops
```

我们这里使用一个名为 jenkins/jenkins:its 的镜像，这是 jenkins 官方的 Docker 镜像，然后也有一些环境变量，当然我们也可以根据自己的需求来定制一个镜像，比如我们可以将一些插件打包在自定义的镜像当中，可以参考文档：<https://github.com/jenkinsci/docker>，我们这里使用默认的官方镜像就行，另外一个还需要注意的是我们将容器的 /var/jenkins_home 目录挂载到了一个名为 opspvc 的 PVC 对象上面，所以我们同样还得提前创建一个对应的 PVC 对象，当然我们也可以使用我们前面的 StorageClass 对象来自动创建：(pvc.yaml)

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: opspv
spec:
  capacity:
    storage: 20Gi
  accessModes:
  - ReadWriteMany
  persistentVolumeReclaimPolicy: Delete
  nfs:
    server: 10.151.30.57
    path: /data/k8s

---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: opspvc
  namespace: kube-ops
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 20Gi

```

创建需要用到的 PVC 对象：

```
$ kubectl create -f pvc.yaml
```

另外我们这里还需要使用到一个拥有相关权限的 serviceAccount: jenkins2，我们这里只是给 jenkins 赋予了一些必要的权限，当然如果你对 serviceAccount 的权限不是很熟悉的话，我们给这个 sa 绑定一个 cluster-admin 的集群角色权限也是可以的，当然这样具有一定的安全风险：(rbac.yaml)

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins2
  namespace: kube-ops

---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: jenkins2

```

```

rules:
- apiGroups: ["extensions", "apps"]
  resources: ["deployments"]
  verbs: ["create", "delete", "get", "list", "watch", "patch", "update"]
- apiGroups: [""]
  resources: ["services"]
  verbs: ["create", "delete", "get", "list", "watch", "patch", "update"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create", "delete", "get", "list", "patch", "update", "watch"]
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: jenkins2
  namespace: kube-ops
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: jenkins2
subjects:
- kind: ServiceAccount
  name: jenkins2
  namespace: kube-ops

```

创建 rbac 相关的资源对象：

```

$ kubectl create -f rbac.yaml
serviceaccount "jenkins2" created
role.rbac.authorization.k8s.io "jenkins2" created
rolebinding.rbac.authorization.k8s.io "jenkins2" created

```

最后为了方便我们测试，我们这里通过 NodePort 的形式来暴露 Jenkins 的 web 服务，固定为30002 端口，另外还需要暴露一个 agent 的端口，这个端口主要是用于 Jenkins 的 master 和 slave 之间通信使用的。

一切准备的资源准备好过后，我们直接创建 Jenkins 服务：

```

$ kubectl create -f jenkins2.yaml
deployment.extensions "jenkins2" created
service "jenkins2" created

```

创建完成后，要去拉取镜像可能需要等待一会儿，然后我们查看下 Pod 的状态：

```

$ kubectl get pods -n kube-ops

```

NAME	READY	STATUS	RESTARTS	AGE
jenkins2-7f5494cd44-pqpzs	0/1	Running	0	2m

可以看到该 Pod 处于 Running 状态，但是 READY 值确为0，然后我们用 describe 命令去查看下该 Pod 的详细信息：

```
$ kubectl describe pod jenkins2-7f5494cd44-pqpzs -n kube-ops
...
Normal  Created            3m      kubelet, node01  Created container
Normal  Started           3m      kubelet, node01  Started container
Warning Unhealthy        1m (x10 over 2m)  kubelet, node01  Liveness probe fail
ed: Get http://10.244.1.165:8080/login: dial tcp 10.244.1.165:8080: getsockopt: connection refused
Warning Unhealthy        1m (x10 over 2m)  kubelet, node01  Readiness probe failed: Get http://10.244.1.165:8080/login: dial tcp 10.244.1.165:8080: getsockopt: connection refused
```

可以看到上面的 Warning 信息，健康检查没有通过，具体原因是什么引起的呢？可以通过查看日志进一步了解：

```
$ kubectl logs -f jenkins2-7f5494cd44-pqpzs -n kube-ops
touch: cannot touch '/var/jenkins_home/copy_reference_file.log': Permission denied
Can not write to /var/jenkins_home/copy_reference_file.log. Wrong volume permissions?
```

很明显可以看到上面的错误信息，意思就是我们没有权限在 jenkins 的 home 目录下面创建文件，这是因为默认的镜像使用的是 jenkins 这个用户，而我们通过 PVC 挂载到 nfs 服务器的共享数据目录下面却是 root 用户的，所以没有权限访问该目录，要解决该问题，也很简单，我只需要在 nfs 共享数据目录下面把我们的目录权限重新分配下即可：

```
$ chown -R 1000 /data/k8s/jenkins2
```

当然还有另外一种方法是我们自定义一个镜像，在镜像中指定使用 root 用户也可以

然后我们再重新创建：

```
$ kubectl delete -f jenkins.yaml
deployment.extensions "jenkins2" deleted
service "jenkins2" deleted
$ kubectl create -f jenkins.yaml
deployment.extensions "jenkins2" created
service "jenkins2" created
```

现在我们再去查看新生成的 Pod 已经没有错误信息了：

```
$ kubectl get pods -n kube-ops
NAME          READY   STATUS    RESTARTS   AGE
jenkins2-7f5494cd44-smn2r  1/1     Running   0          25s
```

等到服务启动成功后，我们就可以根据任意节点的 IP:30002 端口就可以访问 jenkins 服务了，可以根据提示信息进行安装配置即可：

Getting Started

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log (not sure where to find it?) and this file on the server:

`/var/jenkins_home/secrets/initialAdminPassword`

Please copy the password from either location and paste it below.

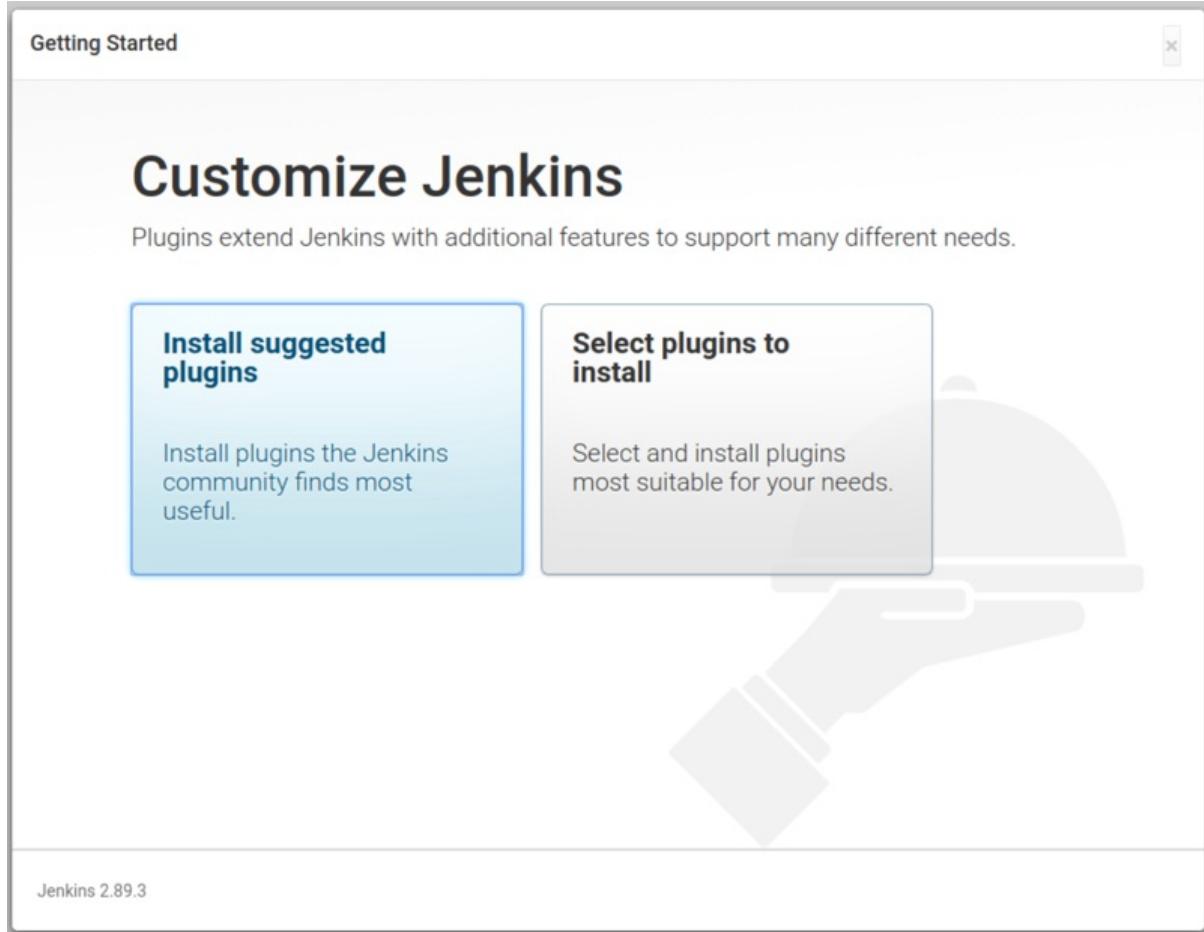
Administrator password

Continue

初始化的密码我们可以在 jenkins 的容器的日志中进行查看，也可以直接在 nfs 的共享数据目录中查看：

```
$ cat /data/k8s/jenkins2/secrets/initAdminPassword
```

然后选择安装推荐的插件即可。



安装完成后添加管理员帐号即可进入到 jenkins 主界面：

The screenshot shows the Jenkins Dashboard. At the top, it says 'Welcome to Jenkins!' and 'Please [create new jobs](#) to get started.' On the left, there is a sidebar with links: 'New Item', 'People', 'Build History', 'Project Relationship', 'Check File Fingerprint', 'Manage Jenkins', 'My Views', 'Open Blue Ocean', 'Credentials', and 'New View'. Below the sidebar, there are two sections: 'Build Queue' (which says 'No builds in the queue.') and 'Build Executor Status' (which shows '1 Idle' and '2 Idle'). At the bottom right, it says 'Page generated: Jul 23, 2018 1:56:03 AM CST REST API Jenkins ver. 2.121.2'.

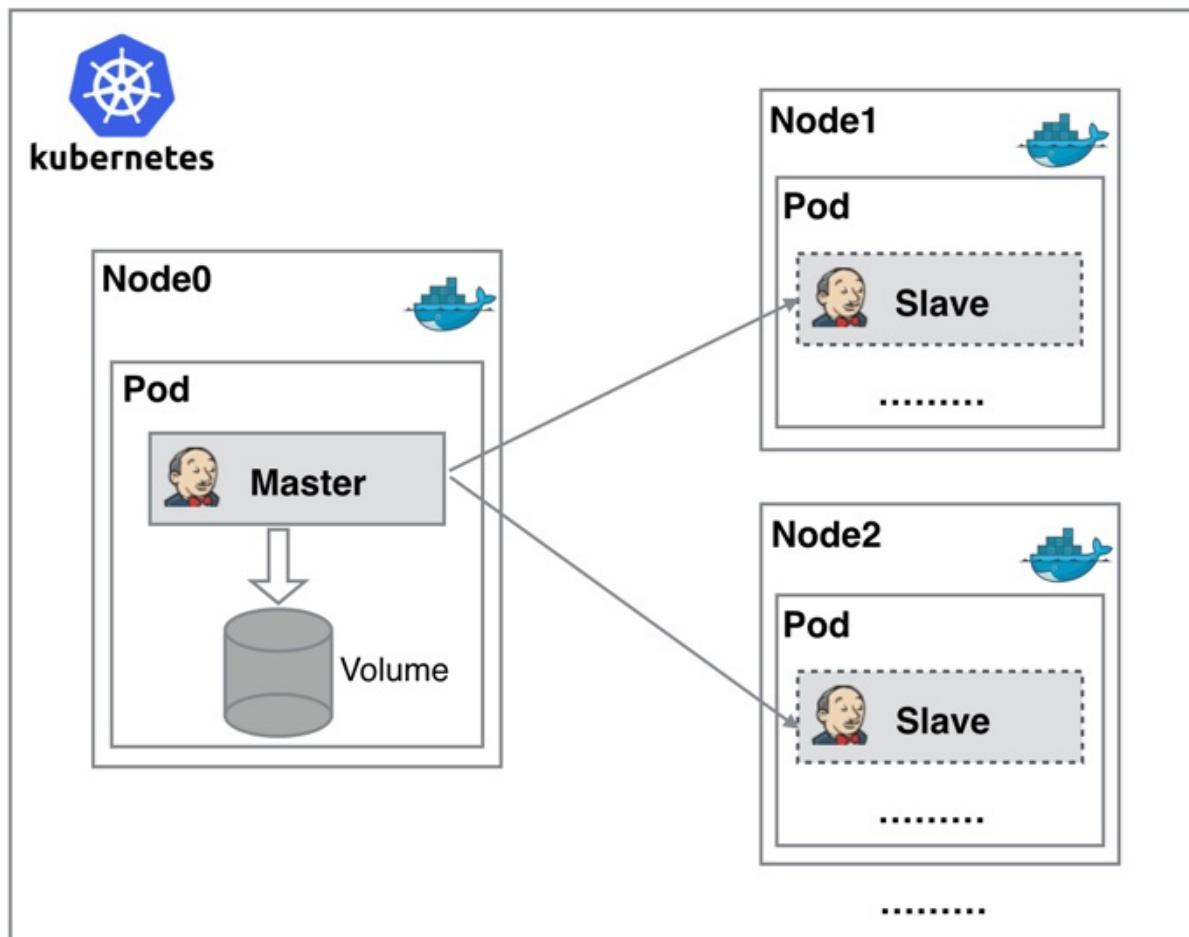
优点

Jenkins 安装完成了，接下来我们不用急着就去使用，我们要了解下在 Kubernetes 环境下面使用 Jenkins 有什么好处。

我们知道持续构建与发布是我们日常工作中必不可少的一个步骤，目前大多公司都采用 Jenkins 集群来搭建符合需求的 CI/CD 流程，然而传统的 Jenkins Slave 一主多从方式会存在一些痛点，比如：

- 主 Master 发生单点故障时，整个流程都不可用了
- 每个 Slave 的配置环境不一样，来完成不同语言的编译打包等操作，但是这些差异化的配置导致管理起来非常不方便，维护起来也是比较费劲
- 资源分配不均衡，有的 Slave 要运行的 job 出现排队等待，而有的 Slave 处于空闲状态
- 资源有浪费，每台 Slave 可能是物理机或者虚拟机，当 Slave 处于空闲状态时，也不会完全释放掉资源。

正因为上面的这些种种痛点，我们渴望一种更高效更可靠的方式来完成这个 CI/CD 流程，而 Docker 虚拟化容器技术能很好的解决这个痛点，又特别是在 Kubernetes 集群环境下面能够更好来解决上面的问题，下图是基于 Kubernetes 搭建 Jenkins 集群的简单示意图：



从图上可以看到 Jenkins Master 和 Jenkins Slave 以 Pod 形式运行在 Kubernetes 集群的 Node 上，Master 运行在其中一个节点，并且将其配置数据存储到一个 Volume 上去，Slave 运行在各个节点上，并且它不是一直处于运行状态，它会按照需求动态的创建并自动删除。

这种方式的工作流程大致为：当 Jenkins Master 接受到 Build 请求时，会根据配置的 Label 动态创建一个运行在 Pod 中的 Jenkins Slave 并注册到 Master 上，当运行完 Job 后，这个 Slave 会被注销并且这个 Pod 也会自动删除，恢复到最初状态。

那么我们使用这种方式带来了哪些好处呢？

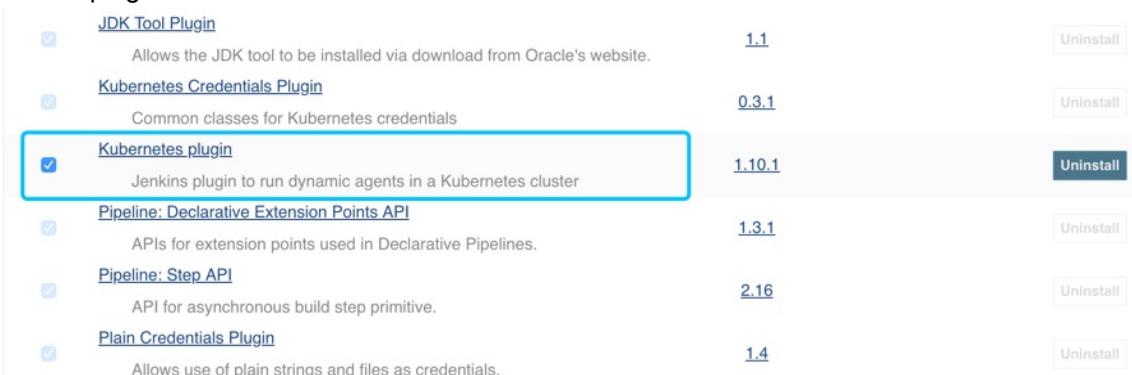
- 服务高可用，当 Jenkins Master 出现故障时，Kubernetes 会自动创建一个新的 Jenkins Master 容器，并且将 Volume 分配给新创建的容器，保证数据不丢失，从而达到集群服务高可用。
- 动态伸缩，合理使用资源，每次运行 Job 时，会自动创建一个 Jenkins Slave，Job 完成后，Slave 自动注销并删除容器，资源自动释放，而且 Kubernetes 会根据每个资源的使用情况，动态分配 Slave 到空闲的节点上创建，降低出现因某节点资源利用率高，还排队等待在该节点的情况。
- 扩展性好，当 Kubernetes 集群的资源严重不足而导致 Job 排队等待时，可以很容易的添加一个 Kubernetes Node 到集群中，从而实现扩展。

是不是以前我们面临的种种问题在 Kubernetes 集群环境下面是不是都没有了啊？看上去非常完美。

配置

接下来我们就需要来配置 Jenkins，让他能够动态的生成 Slave 的 Pod。

第1步. 我们需要安装kubernetes plugin， 点击 Manage Jenkins -> Manage Plugins -> Available -> Kubernetes plugin 勾选安装即可。



第2步. 安装完毕后, 点击 Manage Jenkins —> Configure System —> (拖到最下方)Add a new cloud —> 选择 Kubernetes, 然后填写 Kubernetes 和 Jenkins 配置信息。

The screenshot shows the Jenkins 'Configure System' page with the 'Cloud' section selected. A new cloud configuration is being added for 'Kubernetes'. The configuration includes:

- Name:** kubernetes
- Kubernetes URL:** https://kubernetes.default.svc.cluster.local
- Kubernetes server certificate key:** (Empty text area)
- Disable https certificate check:** (unchecked checkbox)
- Kubernetes Namespace:** kube-ops
- Credentials:** - none - (dropdown menu) with an 'Add' button.
- Connection test successful:** (Text message)
- Test Connection:** (Grayed-out button)
- Jenkins URL:** http://jenkins.kube-ops.svc.cluster.local:8080
- Jenkins tunnel:** (Empty text area)
- Connection Timeout:** 0
- Read Timeout:** 0

注意 namespace, 我们这里填 kube-ops, 然后点击Test Connection, 如果出现 Connection test successful 的提示信息证明 Jenkins 已经可以和 Kubernetes 系统正常通信了, 然后下方的 Jenkins URL 地址: <http://jenkins2.kube-ops.svc.cluster.local:8080>, 这里的格式为: 服务名.namespace.svc.cluster.local:8080, 根据上面创建的jenkins 的服务名填写, 我这里是之前创建的名为jenkins, 如果是用上面我们创建的就应该是jenkins2

另外需要注意, 如果这里 Test Connection 失败的话, 很有可能是权限问题, 这里就需要把我们创建的 jenkins 的 serviceAccount 对应的 secret 添加到这里的 Credentials 里面。

Kubernetes Pod Template

Name	jnlp
Namespace	kube-ops
Labels	haimaxy-jnlp
Usage	Only build jobs with label expressions matching this node <input type="checkbox"/>
The name of the pod template to inherit from	

Containers

Container Template

Name	jnlp
Docker image	cnych/jenkins:jnlp
Always pull image	<input type="checkbox"/>
Working directory	/home/jenkins
Command to run	
Arguments to pass to the command	
Allocate pseudo-TTY	<input checked="" type="checkbox"/>
EnvVars	Add

另外需要注意我们这里需要在下面挂载两个主机目录，一个是 /var/run/docker.sock，该文件是用于 Pod 中的容器能够共享宿主机的 Docker，这就是大家说的 docker in docker 的方式，Docker 二进制文件我们已经打包到上面的镜像中了，另外一个目录下 /root/.kube 目录，我们将这个目录挂载到容器

的 /home/jenkins/.kube 目录下面这是为了让我们能够在 Pod 的容器中能够使用 kubectl 工具来访问我们的 Kubernetes 集群，方便我们后面在 Slave Pod 部署 Kubernetes 应用。

The screenshot shows the Jenkins Slave configuration interface. At the top, there are sections for 'EnvVars' and 'Volumes'. The 'EnvVars' section has a dropdown 'Add Container' and a list 'List of container in the agent pod'. The 'Volumes' section has a dropdown 'Add Environment Variable' and a list 'List of environment variables to set in all container of the pod'. Under 'Volumes', there are two entries for 'Host Path Volume': one for Docker socket and one for the kubeconfig file. Each entry has 'Host path' and 'Mount path' fields and a 'Delete Volume' button. Below these, there is another 'Host Path Volume' entry for the kubeconfig file. At the bottom, there is a 'Delete Volume' button and a 'Add Volume' dropdown.

另外还有几个参数需要注意，如下图中的Time in minutes to retain slave when idle，这个参数表示的意思是当处于空闲状态的时候保留 Slave Pod 多长时间，这个参数最好我们保存默认就行了，如果你设置过大的话，Job 任务执行完成后，对应的 Slave Pod 就不会立即被销毁删除。

The screenshot shows the Jenkins Slave configuration interface with several parameters highlighted:

- Max number of instances:** A dropdown menu for selecting the maximum number of slave instances.
- Apply cap only on alive pods:** A checkbox for applying caps only to active pods.
- Time in minutes to retain slave when idle:** A highlighted input field for setting the retention time for idle slaves, currently set to 0.
- Time in seconds for Pod deadline:** An input field for the pod deadline, currently set to 0.
- Timeout in seconds for Jenkins connection:** An input field for the Jenkins connection timeout, currently set to 100.
- Annotations:** A dropdown menu for adding annotations to the slave pod.
- Raw yaml for the Pod:** A large text area for entering raw YAML configuration for the pod.

另外一些同学在配置了后运行 Slave Pod 的时候出现了权限问题，因为 Jenkins Slave Pod 中没有配置权限，所以需要配置上 ServiceAccount，在 Slave Pod 配置的地方点击下面的高级，添加上对应的 ServiceAccount 即可：

The screenshot shows the Jenkins Kubernetes Slave configuration page. It includes fields for:

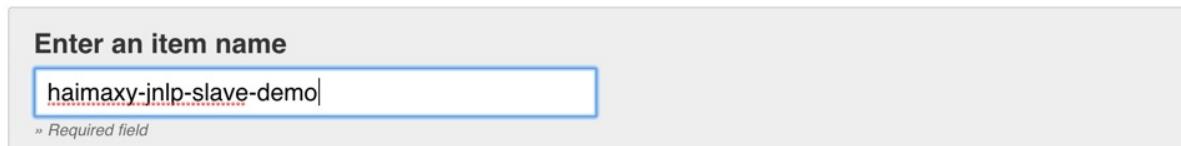
- Time in minutes to retain slave when idle (empty input field)
- Time in seconds for Pod deadline (empty input field)
- Timeout in seconds for Jenkins connection (input field set to 100)
- Annotations (button labeled "Add Annotation")
- Raw yaml for the Pod (large text area)
- ImagePullSecrets (button labeled "Add Image Pull Secret")
- Service Account (input field set to jenkins2, highlighted with a red border)
- Node Selector (empty input field)

到这里我们的 Kubernetes Plugin 插件就算配置完成了。

测试

Kubernetes 插件的配置工作完成了，接下来我们就来添加一个 Job 任务，看是否能够在 Slave Pod 中执行，任务执行完成后看 Pod 是否会被销毁。

在 Jenkins 首页点击create new jobs，创建一个测试的任务，输入任务名称，然后我们选择 Freestyle project 类型的任务：



Enter an item name

haimaxy-jnlp-slave-demo

» Required field

Freestyle project
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

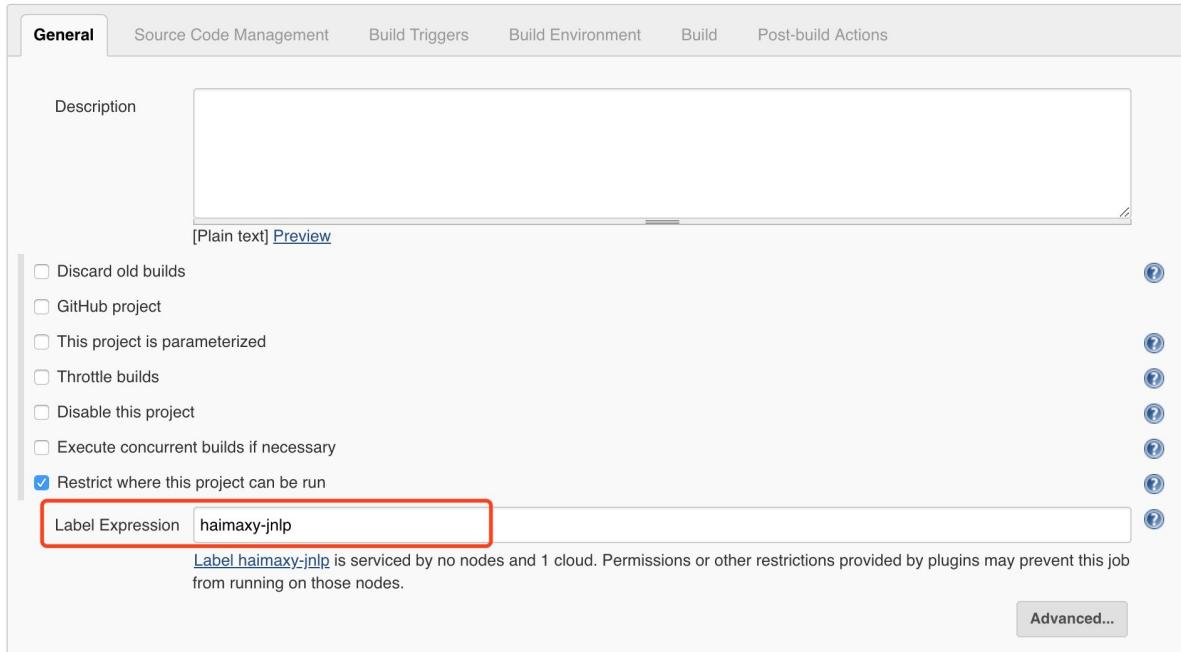
Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Multi-configuration project
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Bitbucket Team/Project
Scans a Bitbucket Cloud Team (or Bitbucket Server Project) for all repositories matching some defined markers.

Folder
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

注意在下面的 Label Expression 这里要填入 haimaxy-jnlp，就是前面我们配置的 Slave Pod 中的 Label，这两个地方必须保持一致



General Source Code Management Build Triggers Build Environment Build Post-build Actions

Description

[Plain text] Preview

Discard old builds ?

GitHub project ?

This project is parameterized ?

Throttle builds ?

Disable this project ?

Execute concurrent builds if necessary ?

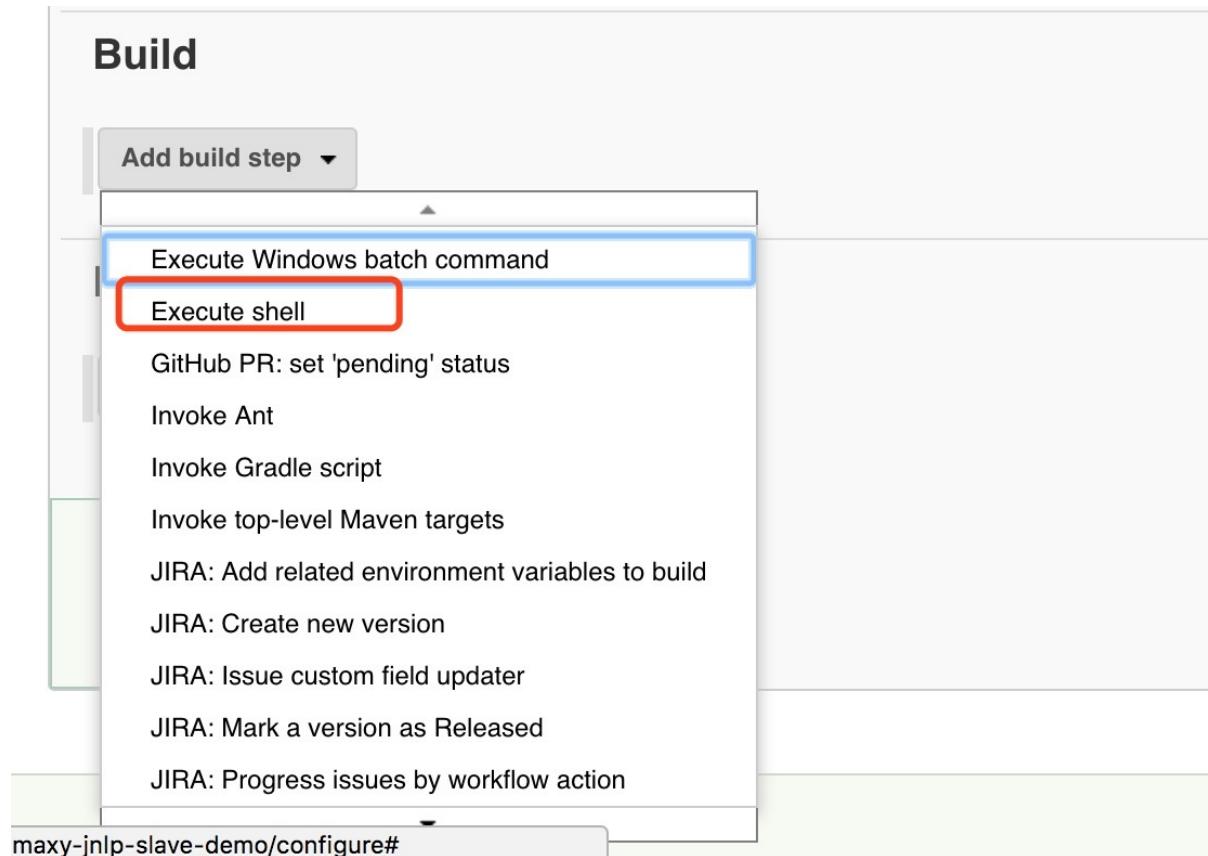
Restrict where this project can be run ?

Label Expression **haimaxy-jnlp**

Label haimaxy-jnlp is serviced by no nodes and 1 cloud. Permissions or other restrictions provided by plugins may prevent this job from running on those nodes.

Advanced...

然后往下拉，在 Build 区域选择 Execute shell



然后输入我们测试命令

```
echo "测试 Kubernetes 动态生成 jenkins slave"
echo "=====docker in docker====="
docker info

echo "=====kubectl====="
kubectl get pods
```

最后点击保存

Build

Execute shell

Command

```
echo "测试 Kubernetes 动态生成 jenkins slave"
echo "=====docker in docker====="
docker info

echo "=====kubectl====="
kubectl get pods
```

See [the list of available environment variables](#)

Post-build Actions

Add post-build action ▾

Save **Apply**

现在我们直接在页面点击做成的 Build now 触发构建即可，然后观察 Kubernetes 集群中 Pod 的变化

```
$ kubectl get pods -n kube-ops
NAME                  READY   STATUS    RESTARTS   AGE
jenkins2-7c85b6f4bd-rfqgv   1/1     Running   3          1d
jnlp-hfmvd            0/1     ContainerCreating   0          7s
```

我们可以看到在我们点击立刻构建的时候可以看到一个新的 Pod: jnlp-hfmvd 被创建了，这就是我们的 Jenkins Slave。任务执行完成后我们可以看到任务信息，比如我们这里是花费了 5.2s 时间在 jnlp-hfmvd 这个 Slave 上面

Build #2 (Jul 23, 2018 3:03:40 AM)

Started 1 min 51 sec ago
Took 5.2 sec on jnlp-hfmvd

No changes.

Started by user [haimaxyAdmin](#)

[add description](#)

同样也可以查看到对应的控制台信息：

```
runc version: 54296cf40ad8143b62dbcaald90e520a2136ddfe
init version: 949e6fa
Security Options:
    seccomp
        Profile: default
Kernel Version: 3.10.0-327.el7.x86_64
Operating System: CentOS Linux 7 (Core)
OSType: linux
Architecture: x86_64
CPUs: 4
Total Memory: 7.64GiB
Name: node03
ID: NY56:7P3T:ZMV6:UCCI:JD7L:FJZK:UAZW:MSID:THPF:AJK3:3ZFF:CEY2
Docker Root Dir: /var/lib/docker
Debug Mode (client): false
Debug Mode (server): false
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
    127.0.0.0/8
Registry Mirrors:
    https://ot2k4d59.mirror.aliyuncs.com/
Live Restore Enabled: false

WARNING: overlay: the backing xfs filesystem is formatted without d_type support, which leads to incorrect behavior.
        Reformat the filesystem with ftype=1 to enable d_type support.
        Running without d_type support will not be supported in future releases.
+ echo =====kubectl=====
=====kubectl=====
+ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
jenkins-7c85b6f4bd-rfqgv   1/1     Running   3          1d
jnlp-hfmvd      1/1     Running   0          21s
Finished: SUCCESS
```

到这里证明我们的任务已经构建完成，然后这个时候我们再去集群查看我们的 Pod 列表，发现 kube-ops 这个 namespace 下面已经没有之前的 Slave 这个 Pod 了。

```
$ kubectl get pods -n kube-ops
NAME          READY   STATUS    RESTARTS   AGE
jenkins2-7c85b6f4bd-rfqgv   1/1     Running   3          1d
```

到这里我们就完成了使用 Kubernetes 动态生成 Jenkins Slave 的方法。下节课我们来给大家介绍下怎么在 Jenkins 中来发布我们的 Kubernetes 应用。最后感谢圈友 @TangT-newhope-成都 提供的帮助。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-05-05 18:09:35

基于 Jenkins 的 CI/CD (二)

上节课我们实现了在 Kubernetes 环境中动态生成 Jenkins Slave 的方法，这节课我们来给大家讲解下如何在 Jenkins 中来部署一个 Kubernetes 应用。

Jenkins Pipeline 介绍

要实现在 Jenkins 中的构建工作，可以有多种方式，我们这里采用比较常用的 Pipeline 这种方式。Pipeline，简单来说，就是一套运行在 Jenkins 上的工作流框架，将原来独立运行于单个或者多个节点的任务连接起来，实现单个任务难以完成的复杂流程编排和可视化的工作。

Jenkins Pipeline 有几个核心概念：

- Node：节点，一个 Node 就是一个 Jenkins 节点，Master 或者 Agent，是执行 Step 的具体运行环境，比如我们之前动态运行的 Jenkins Slave 就是一个 Node 节点
- Stage：阶段，一个 Pipeline 可以划分为若干个 Stage，每个 Stage 代表一组操作，比如：Build、Test、Deploy，Stage 是一个逻辑分组的概念，可以跨多个 Node
- Step：步骤，Step 是最基本的操作单元，可以是打印一句话，也可以是构建一个 Docker 镜像，由各类 Jenkins 插件提供，比如命令：sh 'make'，就相当于我们平时 shell 终端中执行 make 命令一样。

那么我们如何创建 Jenkins Pipeline 呢？

- Pipeline 脚本是由 Groovy 语言实现的，但是我们没必要单独去学习 Groovy，当然你会的话最好
- Pipeline 支持两种语法：Declarative(声明式)和 Scripted Pipeline(脚本式)语法
- Pipeline 也有两种创建方法：可以直接在 Jenkins 的 Web UI 界面中输入脚本；也可以通过创建一个 Jenkinsfile 脚本文件放入项目源码库中
- 一般我们都推荐在 Jenkins 中直接从源代码控制(SCMD)中直接载入 Jenkinsfile Pipeline 这种方法

创建一个简单的 Pipeline

我们这里来给大家快速创建一个简单的 Pipeline，直接在 Jenkins 的 Web UI 界面中输入脚本运行。

- 新建 Job：在 Web UI 中点击 New Item -> 输入名称：pipeline-demo -> 选择下面的 Pipeline -> 点击 OK
- 配置：在最下方的 Pipeline 区域输入如下 Script 脚本，然后点击保存。

```
node {
    stage('Clone') {
        echo "1.Clone Stage"
    }
    stage('Test') {
        echo "2.Test Stage"
    }
    stage('Build') {
        echo "3.Build Stage"
    }
    stage('Deploy') {
```

```

        echo "4. Deploy Stage"
    }
}

```

- 构建：点击左侧区域的 Build Now，可以看到 Job 开始构建了

隔一会儿，构建完成，可以点击左侧区域的 Console Output，我们就可以看到如下输出信息：

The screenshot shows the Jenkins interface for a pipeline job named 'pipeline-demo'. On the left, there's a sidebar with various options like Status, Changes, and Pipeline Steps. The main area is titled 'Console Output' and displays the following log output:

```

Started by user haimaxyAdmin
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Running on Jenkins in /var/jenkins_home/workspace/pipeline-demo
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Clone)
[Pipeline] echo
1.Clone Stage
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] echo
2.Test Stage
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] echo
3.Build Stage
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] echo
4. Deploy Stage
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

我们可以看到上面我们 Pipeline 脚本中的4条输出语句都打印出来了，证明是符合我们的预期的。

如果大家对 Pipeline 语法不是特别熟悉的，可以前往输入脚本的下面的链接[Pipeline Syntax](#)中进行查看，这里有很多关于 Pipeline 语法的介绍，也可以自动帮我们生成一些脚本。

在 Slave 中构建任务

上面我们创建了一个简单的 Pipeline 任务，但是我们可以看到这个任务并没有在 Jenkins 的 Slave 中运行，那么如何让我们的任务跑在 Slave 中呢？还记得上节课我们在添加 Slave Pod 的时候，一定要记住添加的 label 吗？没错，我们就需要用到这个 label，我们重新编辑上面创建的 Pipeline 脚本，给 node 添加一个 label 属性，如下：

```

node('haimaxy-jnlp') {
    stage('Clone') {
        echo "1.Clone Stage"
    }
    stage('Test') {
        echo "2.Test Stage"
    }
}

```

```
stage('Build') {
    echo "3. Build Stage"
}
stage('Deploy') {
    echo "4. Deploy Stage"
}
```

我们这里只是给 node 添加了一个 haimaxy-jnlp 这样的一个label，然后我们保存，构建之前查看下 kubernetes 集群中的 Pod：

```
$ kubectl get pods -n kube-ops
NAME           READY   STATUS    RESTARTS   AGE
jenkins-7c85b6f4bd-rfqgv   1/1     Running   4          6d
```

然后重新触发立刻构建：

```
$ kubectl get pods -n kube-ops
NAME           READY   STATUS    RESTARTS   AGE
jenkins-7c85b6f4bd-rfqgv   1/1     Running   4          6d
jnlp-0hrrz      1/1     Running   0          23s
```

我们发现多了一个名叫jnlp-0hrrz的 Pod 正在运行，隔一会儿这个 Pod 就不再了：

```
$ kubectl get pods -n kube-ops
NAME           READY   STATUS    RESTARTS   AGE
jenkins-7c85b6f4bd-rfqgv   1/1     Running   4          6d
```

这也证明我们的 Job 构建完成了，同样回到 Jenkins 的 Web UI 界面中查看 Console Output，可以看到如下的信息：

Console Output

```

Started by user haimaxyAdmin
Replayed #1
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] node
Still waiting to schedule task
jnlp-0hrzz is offline
Agent jnlp-0hrzz is provisioned from template Kubernetes Pod Template
Agent specification [Kubernetes Pod Template] (haimaxy-jnlp):
* [jnlp] cnych/jenkins:jnlp(resourceRequestCpu: , resourceRequestMemory: , resourceLimitCpu: , resourceLimitMemory: )

Running on jnlp-0hrzz in /home/jenkins/workspace/pipeline-demo
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Clone)
[Pipeline] echo
1.Clone Stage
[Pipeline] {
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] echo
2.Test Stage
[Pipeline] {
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] echo
3.Build Stage

```

是不是也证明我们当前的任务在跑在上面动态生成的这个 Pod 中，也符合我们的预期。我们回到 Job 的主界面，也可以看到大家可能比较熟悉的 Stage View 界面：

	Clone	Test	Build	Deploy
Average stage times:	579ms	309ms	550ms	450ms
(Average full run time: ~20s)				

Recent Changes

Build History

- #2 Jul 28, 2018 7:17 PM
- #1 Jul 28, 2018 7:06 PM

Permalinks

- Last build (#2), 10 min ago
- Last stable build (#2), 10 min ago
- Last successful build (#2), 10 min ago
- Last completed build (#2), 10 min ago

Page generated: Jul 28, 2018 7:27:43 PM CST REST API Jenkins ver. 2.121.2

pipeline demo3

部署 Kubernetes 应用

上面我们已经知道了如何在 Jenkins Slave 中构建任务了，那么如何来部署一个原生的 Kubernetes 应用呢？要部署 Kubernetes 应用，我们就得对我们之前部署应用的流程要非常熟悉才行，我们之前的流程是怎样的：

- 编写代码
- 测试
- 编写 Dockerfile
- 构建打包 Docker 镜像
- 推送 Docker 镜像到仓库
- 编写 Kubernetes YAML 文件
- 更改 YAML 文件中 Docker 镜像 TAG
- 利用 kubectl 工具部署应用

我们之前在 Kubernetes 环境中部署一个原生应用的流程应该基本上是上面这些流程吧？现在我们就需要把上面这些流程放入 Jenkins 中来自动帮我们完成(当然编码除外)，从测试到更新 YAML 文件属于 CI 流程，后面对应部署属于 CD 的流程。如果按照我们上面的示例，我们现在要来编写一个 Pipeline 的脚本，应该怎么编写呢？

```
node('haiamaxy-jnlp') {
    stage('Clone') {
        echo "1.Clone Stage"
    }
    stage('Test') {
        echo "2.Test Stage"
    }
    stage('Build') {
        echo "3.Build Docker Image Stage"
    }
    stage('Push') {
        echo "4.Push Docker Image Stage"
    }
    stage('YAML') {
        echo "5. Change Yaml File Stage"
    }
    stage('Deploy') {
        echo "6. Deploy Stage"
    }
}
```

这里我们来将一个简单 golang 程序，部署到 kubernetes 环境中，代码链接：

<https://github.com/cnych/jenkins-demo>。如果按照之前的示例，我们是不是应该像这样来编写 Pipeline 脚本：

- 第一步，clone 代码，这个没得说吧
- 第二步，进行测试，如果测试通过了才继续下面的任务
- 第三步，由于 Dockerfile 基本上都是放入源码中进行管理的，所以我们这里就是直接构建 Docker 镜像了
- 第四步，镜像打包完成，就应该推送到镜像仓库中吧

- 第五步，镜像推送完成，是不是需要更改 YAML 文件中的镜像 TAG 为这次镜像的 TAG
- 第六步，万事俱备，只差最后一步，使用 kubectl 命令行工具进行部署了

到这里我们的整个 CI/CD 的流程是不是就都完成了。

接下来我们就来对每一步具体要做的事情进行详细描述就行了：

第一步，Clone 代码

```
stage('Clone') {
    echo "1.Clone Stage"
    git url: "https://github.com/cnych/jenkins-demo.git"
}
```

第二步，测试

由于我们这里比较简单，忽略该步骤即可

第三步，构建镜像

```
stage('Build') {
    echo "3.Build Docker Image Stage"
    sh "docker build -t cnych/jenkins-demo:${build_tag} ."
}
```

我们平时构建的时候是不是都是直接使用 docker build 命令进行构建就行了，那么这个地方呢？我们上节课给大家提供的 Slave Pod 的镜像里面是不是采用的 Docker In Docker 的方式，也就是说我们也可以直接在 Slave 中使用 docker build 命令，所以我们这里直接使用 sh 直接执行 docker build 命令即可，但是镜像的 tag 呢？如果我们使用镜像 tag，则每次都是 latest 的 tag，这对于以后的排查或者回滚之类的工作会带来很大麻烦，我们这里采用和git commit的记录为镜像的 tag，这里有一个好处就是镜像的 tag 可以和 git 提交记录对应起来，也方便日后对应查看。但是由于这个 tag 不只是我们这一个 stage 需要使用，下一个推送镜像是也需要，所以这里我们把这个 tag 编写成一个公共的参数，把它放在 Clone 这个 stage 中，这样一来我们前两个 stage 就变成了下面这个样子：

```
stage('Clone') {
    echo "1.Clone Stage"
    git url: "https://github.com/cnych/jenkins-demo.git"
    script {
        build_tag = sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
    }
}
stage('Build') {
    echo "3.Build Docker Image Stage"
    sh "docker build -t cnych/jenkins-demo:${build_tag} ."
}
```

第四步，推送镜像

镜像构建完成了，现在我们就需要将此处构建的镜像推送到镜像仓库中去，当然如果你有私有镜像仓库也可以，我们这里还没有自己搭建私有的仓库，所以直接使用 docker hub 即可。

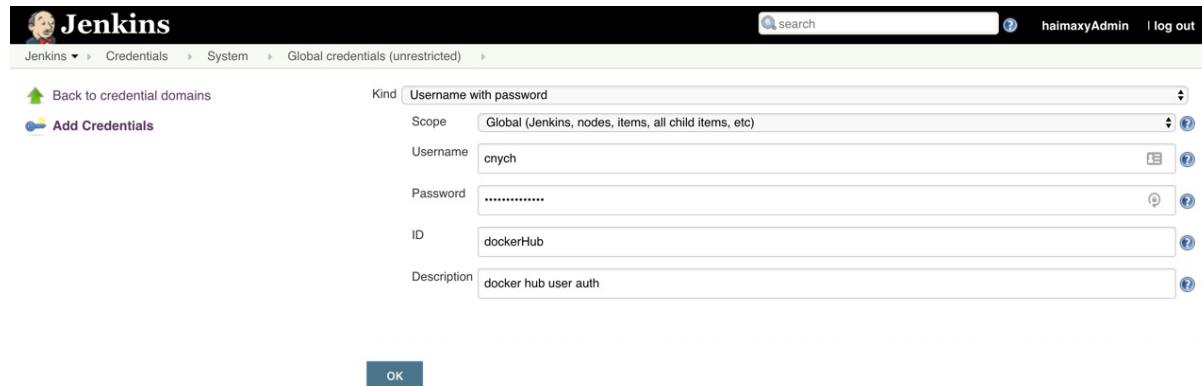
在后面的课程中我们学习了私有仓库 Harbor 的搭建后，再来更改成 Harbor 仓库

我们知道 docker hub 是公共的镜像仓库，任何人都可以获取上面的镜像，但是要往上推送镜像我们就需要用到一个帐号了，所以我们需要提前注册一个 docker hub 的帐号，记住用户名和密码，我们这里需要使用。正常来说我们在本地推送 docker 镜像的时候，是不是需要使用 docker login 命令，然后输入用户名和密码，认证通过后，就可以使用 docker push 命令来推送本地的镜像到 docker hub 上面去了，如果是这样的话，我们这里的 Pipeline 是不是就该这样写了：

```
stage('Push') {
    echo "4.Push Docker Image Stage"
    sh "docker login -u cnych -p xxxxx"
    sh "docker push cnych/jenkins-demo:${build_tag}"
}
```

如果我们只是在 Jenkins 的 Web UI 界面中来完成这个任务的话，我们这里的 Pipeline 是可以这样写的，但是我们是不是推荐使用 Jenkinsfile 的形式放入源码中进行版本管理，这样的话我们直接把 docker 仓库的用户名和密码暴露给别人这样很显然是非常非常不安全的，更何况我们这里使用的是 github 的公共代码仓库，所有人都可以直接看到我们的源码，所以我们应该用一种方式来隐藏用户名和密码这种私密信息，幸运的是 Jenkins 为我们提供了解决方法。

在首页点击 Credentials -> Stores scoped to Jenkins 下面的 Jenkins -> Global credentials (unrestricted) -> 左侧的 Add Credentials：添加一个 Username with password 类型的认证信息，如下：



输入 docker hub 的用户名和密码，ID 部分我们输入 dockerHub，注意，这个值非常重要，在后面 Pipeline 的脚本中我们需要使用到这个 ID 值。

有了上面的 docker hub 的用户名和密码的认证信息，现在我们可以在 Pipeline 中使用这里的用户名和密码了：

```
stage('Push') {
    echo "4.Push Docker Image Stage"
    withCredentials([usernamePassword(credentialsId: 'dockerHub', passwordVariable: 'dockerHubPassword', usernameVariable: 'dockerHubUser')]) {
        sh "docker login -u ${dockerHubUser} -p ${dockerHubPassword}"
        sh "docker push cnych/jenkins-demo:${build_tag}"
    }
}
```

注意我们这里在 stage 中使用了一个新的函数withCredentials，其中有一个 credentialsId 值就是我们刚刚创建的 ID 值，而对应的用户名变量就是 ID 值加上 User，密码变量就是 ID 值加上 Password，然后我们就可以在脚本中直接使用这里两个变量值来直接替换掉之前的登录 docker hub 的用户名和密码，现在是不是就很安全了，我只是传递进去了两个变量而已，别人并不知道我的真正用户名和密码，只有我们自己的 Jenkins 平台上添加的才知道。

第五步，更改 YAML

上面我们已经完成了镜像的打包、推送的工作，接下来我们是不是应该更新 Kubernetes 系统中应用的镜像版本了，当然为了方便维护，我们都是用 YAML 文件的形式来编写应用部署规则，比如我们这里的 YAML 文件：(k8s.yaml)

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: jenkins-demo
spec:
  template:
    metadata:
      labels:
        app: jenkins-demo
    spec:
      containers:
        - image: cnych/jenkins-demo:<BUILD_TAG>
          imagePullPolicy: IfNotPresent
          name: jenkins-demo
```

对于 Kubernetes 比较熟悉的同学，对上面这个 YAML 文件一定不会陌生，我们使用一个 Deployment 资源对象来管理 Pod，该 Pod 使用的就是我们上面推送的镜像，唯一不同的地方是 Docker 镜像的 tag 不是我们常见的具体的 tag，而是一个 的标识，实际上如果我们将这个标识替换成上面的 Docker 镜像的 tag，是不是就是最终我们本次构建需要使用到的镜像？怎么替换呢？其实也很简单，我们使用一个 sed 命令就可以实现了：

```
stage('YAML') {
  echo "5. Change YAML File Stage"
  sh "sed -i 's/<BUILD_TAG>/${build_tag}/' k8s.yaml"
}
```

上面的 sed 命令就是将 k8s.yaml 文件中的 标识给替换成变量 build_tag 的值。

第六步，部署

Kubernetes 应用的 YAML 文件已经更改完成了，之前我们手动的环境下，是不是直接使用 kubectl apply 命令就可以直接更新应用了啊？当然我们这里只是写入到了 Pipeline 里面，思路都是一样的：

```
stage('Deploy') {
  echo "6. Deploy Stage"
  sh "kubectl apply -f k8s.yaml"
}
```

这样到这里我们的整个流程就算完成了。

人工确认

理论上来说我们上面的6个步骤其实已经完成了，但是一般在我们的实际项目实践过程中，可能还需要一些人工干预的步骤，这是为什么呢？比如我们提交了一次代码，测试也通过了，镜像也打包上传了，但是这个版本并不一定就是要立刻上线到生产环境的，对吧，我们可能需要将该版本先发布到测试环境、QA 环境、或者预览环境之类的，总之直接就发布到线上环境去还是挺少见的，所以我们需要增加人工确认的环节，一般都是在 CD 的环节才需要人工干预，比如我们这里的最后两步，我们就可以在前面加上确认，比如：

```
stage('YAML') {
    echo "5. Change YAML File Stage"
    def userInput = input(
        id: 'userInput',
        message: 'Choose a deploy environment',
        parameters: [
            [
                [
                    $class: 'ChoiceParameterDefinition',
                    choices: "Dev\\nQA\\nProd",
                    name: 'Env'
                ]
            ]
        ]
    )
    echo "This is a deploy step to ${userInput.Env}"
    sh "sed -i 's/<BUILD_TAG>/${build_tag}/' k8s.yaml"
}
```

我们这里使用了 `input` 关键字，里面使用一个 `Choice` 的列表来让用户进行选择，然后在我们选择了部署环境后，我们当然也可以针对不同的环境再做一些操作，比如可以给不同环境的 YAML 文件部署到不同的 namespace 下面去，增加不同的标签等等操作：

```
stage('Deploy') {
    echo "6. Deploy Stage"
    if (userInput.Env == "Dev") {
        // deploy dev stuff
    } else if (userInput.Env == "QA"){
        // deploy qa stuff
    } else {
        // deploy prod stuff
    }
    sh "kubectl apply -f k8s.yaml"
}
```

由于这一步也属于部署的范畴，所以我们可以将最后两步都合并成一步，我们最终的 Pipeline 脚本如下：

```
node('haimaxy-jnlp') {
    stage('Clone') {
        echo "1.Clone Stage"
        git url: "https://github.com/cnych/jenkins-demo.git"
        script {
```

```

        build_tag = sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
    }
}
stage('Test') {
    echo "2. Test Stage"
}
stage('Build') {
    echo "3. Build Docker Image Stage"
    sh "docker build -t cnych/jenkins-demo:${build_tag} ."
}
stage('Push') {
    echo "4. Push Docker Image Stage"
    withCredentials([usernamePassword(credentialsId: 'dockerHub', passwordVariable: 'dockerHubPassword', usernameVariable: 'dockerHubUser')]) {
        sh "docker login -u ${dockerHubUser} -p ${dockerHubPassword}"
        sh "docker push cnych/jenkins-demo:${build_tag}"
    }
}
stage('Deploy') {
    echo "5. Deploy Stage"
    def userInput = input(
        id: 'userInput',
        message: 'Choose a deploy environment',
        parameters: [
            [
                $class: 'ChoiceParameterDefinition',
                choices: "Dev\\nQA\\nProd",
                name: 'Env'
            ]
        ]
    )
    echo "This is a deploy step to ${userInput}"
    sh "sed -i 's/<BUILD_TAG>/${build_tag}/' k8s.yaml"
    if (userInput == "Dev") {
        // deploy dev stuff
    } else if (userInput == "QA"){
        // deploy qa stuff
    } else {
        // deploy prod stuff
    }
    sh "kubectl apply -f k8s.yaml"
}
}

```

现在我们在 Jenkins Web UI 中重新配置 jenkins-demo 这个任务，将上面的脚本粘贴到 Script 区域，重新保存，然后点击左侧的 Build Now，触发构建，然后过一会儿我们就可以看到 Stage View 界面出现了暂停的情况：

The screenshot shows the Jenkins Pipeline pipeline-demo stage view. On the left, there's a sidebar with various Jenkins links like Back to Dashboard, Status, Changes, Build Now, Delete Pipeline, Configure, Full Stage View, Open Blue Ocean, Rename, and Pipeline Syntax. Below that is the Build History section, which lists three builds: #3 (Jul 29, 2018 6:49 AM), #2 (Jul 28, 2018 7:17 PM), and #1 (Jul 28, 2018 7:06 PM). The RSS feed links are also present.

The main area is titled "Pipeline pipeline-demo". It shows a "Stage View" grid with five columns: Clone, Test, Build, Push, and Deploy. The Clone and Test stages have average times of 15s and 304ms respectively. The Build stage has an average time of 14s. The Push stage has an average time of 1min 9s. The Deploy stage has an average time of 319ms. A tooltip for the Deploy stage says "Choose a deploy environment" with options "Env" (Dev) and "Proceed" or "Abort".

Below the stage view is a "Permalinks" section with a list of links:

- Last build (#3), 2 min 42 sec ago
- Last stable build (#2), 11 hr ago
- Last successful build (#2), 11 hr ago
- Last completed build (#2), 11 hr ago

At the bottom right, it says "Page generated: Jul 29, 2018 6:52:22 AM CST REST API Jenkins ver. 2.121.2".

这就是我们上面 Deploy 阶段加入了人工确认的步骤，所以这个时候构建暂停了，需要我们人为的确认下，比如我们这里选择 QA，然后点击 Proceed，就可以继续往下走了，然后构建就成功了，我们在 Stage View 的 Deploy 这个阶段可以看到如下的一些日志信息：

The screenshot shows the "Stage Logs (Deploy)" window. It contains the following log entries:

- Print Message -- 5. Deploy Stage -- (self time 78ms)
- Wait for interactive input (self time 7s)
- Print Message -- This is a deploy step to QA -- (self time 80ms)
- This is a deploy step to QA
- Shell Script -- sed -i 's/<BUILD_TAG>/8c2f83e/' k8s.yaml -- (self time 441ms)
- Shell Script -- kubectl apply -f k8s.yaml -- (self time 986ms)

打印出来了 QA，和我们刚刚的选择是一致的，现在我们去 Kubernetes 集群中观察下部署的应用：

```
$ kubectl get deployment -n kube-ops
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
jenkins   1          1          1           1           7d
jenkins-demo 1          1          1           0           1m

$ kubectl get pods -n kube-ops
NAME                  READY   STATUS    RESTARTS   AGE
jenkins-7c85b6f4bd-rfqgv  1/1    Running   4          7d
jenkins-demo-f6f4f646b-2zdrq  0/1    Completed  4          1m

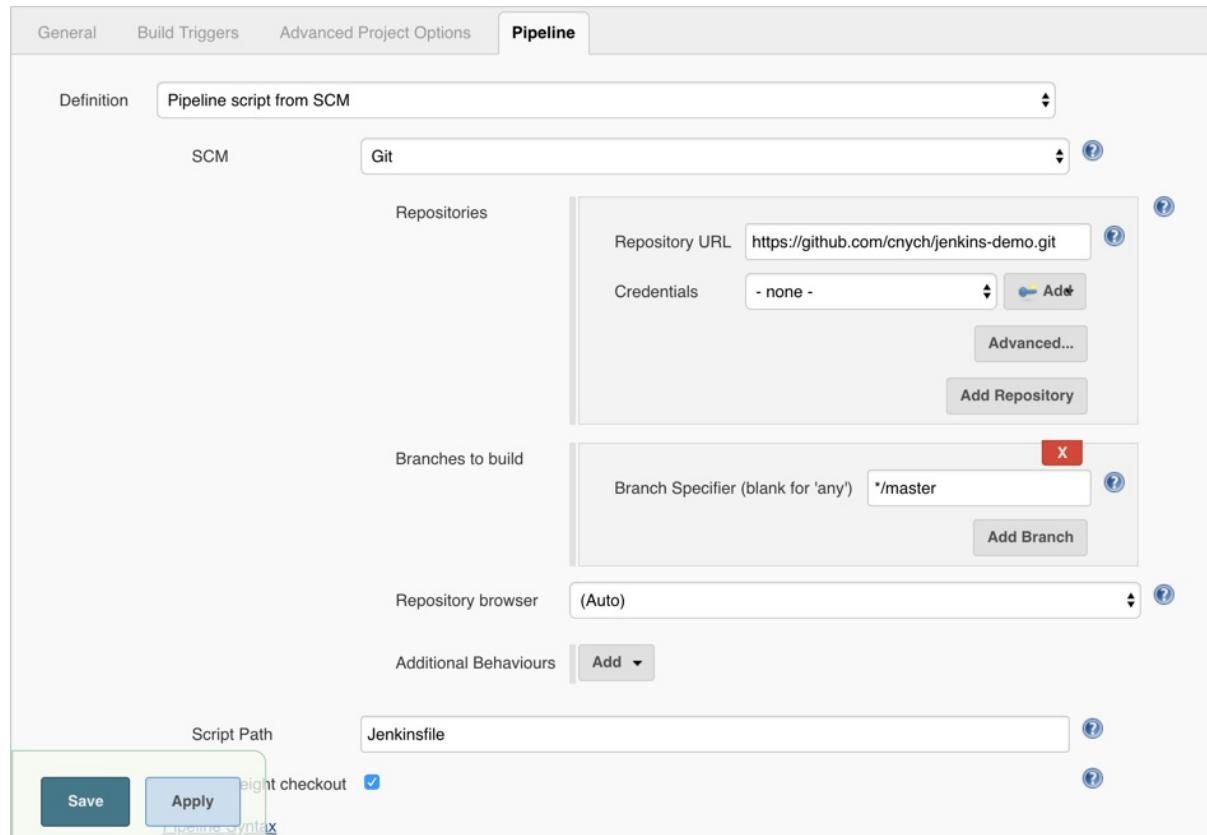
$ kubectl logs jenkins-demo-f6f4f646b-2zdrq -n kube-ops
Hello, Kubernetes! I'm from Jenkins CI!
```

我们可以看到我们的应用已经正确的部署到了 Kubernetes 的集群环境中了。

Jenkinsfile

万里长征，貌似我们的任务完成了，其实不然，我们这里只是完成了一次手动的添加任务的构建过程，在实际的工作实践中，我们更多的是将 Pipeline 脚本写入到 Jenkinsfile 文件中，然后和代码一起提交到代码仓库中进行版本管理。现在我们将上面的 Pipeline 脚本拷贝到一个 Jenkinsfile 中，将该文件放入上面的 git 仓库中，但是要注意的是，现在既然我们已经在 git 仓库中了，是不是就不需要 git clone 这一步骤了，所以我们需要将第一步 Clone 操作中的 git clone 这一步去掉，可以参考：<https://github.com/cnzych/jenkins-demo/>

然后我们更改上面的 jenkins-demo 这个任务，点击 Configure -> 最下方的 Pipeline 区域 -> 将之前的 Pipeline Script 更改成 Pipeline Script from SCM，然后根据我们的实际情况填写上对应的仓库配置，要注意 Jenkinsfile 脚本路径：



最后点击保存，现在我们随便更改下源码，比如把 Jenkinsfile 中第一步更改成 Prepare，然后提交代码。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-03-12 13:03:33

Jenkins BlueOcean

上节课我们讲解了使用 Jenkins Pipeline 来自动化部署一个 Kubernetes 应用的方法，在实际的项目中，往往一个代码仓库都会有很多分支的，比如开发、测试、线上这些分支都是分开的，一般情况下开发或者测试的分支我们希望提交代码后就直接进行 CI/CD 操作，而线上的话最好增加一个人工干预的步骤，这就需要 Jenkins 对代码仓库有多分支的支持，当然这个特性是被 Jenkins 支持的。

Jenkinsfile

同样的，我们可以使用上节课的方法直接把要构建的脚本配置在 Jenkins Web UI 界面中就可以，但是我们也提到过最佳的方式是将脚本写入一个名为 Jenkinsfile 的文件中，跟随代码库进行统一的管理。

我们这里在之前的 git 库中新建一个 dev 分支，然后更改 main.go 的代码，打印当前运行的代码分支，通过环境变量注入进去，所以我们通过 k8s.yaml 文件的环境变量把当前代码分支注入进去，具体代码可以参考<https://github.com/cnych/jenkins-demo/tree/dev>。

然后新建一个 Jenkinsfile 文件，将上节课的构建脚本拷贝进来，但是我们需要对其做一些修改：
(Jenkinsfile)

```
node('haimaxy-jnlp') {
    stage('Prepare') {
        echo "1.Prepare Stage"
        checkout scm
        script {
            build_tag = sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()

            if (env.BRANCH_NAME != 'master') {
                build_tag = "${env.BRANCH_NAME}-${build_tag}"
            }
        }
    }
    stage('Test') {
        echo "2.Test Stage"
    }
    stage('Build') {
        echo "3.Build Docker Image Stage"
        sh "docker build -t cnych/jenkins-demo:${build_tag} ."
    }
    stage('Push') {
        echo "4.Push Docker Image Stage"
        withCredentials([usernamePassword(credentialsId: 'dockerHub', passwordVariable: 'dockerHubPassword', usernameVariable: 'dockerHubUser')]) {
            sh "docker login -u ${dockerHubUser} -p ${dockerHubPassword}"
            sh "docker push cnych/jenkins-demo:${build_tag}"
        }
    }
    stage('Deploy') {
        echo "5. Deploy Stage"
        if (env.BRANCH_NAME == 'master') {
            input "确认要部署线上环境吗？"
        }
        sh "sed -i 's/<BUILD_TAG>/${build_tag}/' k8s.yaml"
```

```
sh "sed -i 's/<BRANCH_NAME>/${env.BRANCH_NAME}/' k8s.yaml"
sh "kubectl apply -f k8s.yaml --record"
}
```

在第一步中我们增加了checkout scm命令，用来检出代码仓库中当前分支的代码，为了避免各个环境的镜像 tag 产生冲突，我们为非 master 分支的代码构建的镜像增加了一个分支的前缀，在第五步中如果是 master 分支的话我们才增加一个确认部署的流程，其他分支都自动部署，并且还需要替换 k8s.yaml 文件中的环境变量的值。

更改完成后，提交 dev 分支到 github 仓库中。

BlueOcean

我们这里使用 BlueOcean 这种方式来完成此处 CI/CD 的工作，BlueOcean 是 Jenkins 团队从用户体验角度出发，专为 Jenkins Pipeline 重新设计的一套 UI 界面，仍然兼容以前的 freestyle 类型的 job，BlueOcean 具有以下的一些特性：

- 连续交付 (CD) Pipeline 的复杂可视化，允许快速直观的了解 Pipeline 的状态
- 可以通过 Pipeline 编辑器直观的创建 Pipeline
- 需要干预或者出现问题时快速定位，BlueOcean 显示了 Pipeline 需要注意的地方，便于异常处理和提高生产力
- 用于分支和拉取请求的本地集成可以在 GitHub 或者 Bitbucket 中与其他人进行代码协作时最大限度提高开发人员的生产力。

BlueOcean 可以安装在现有的 Jenkins 环境中，也可以使用 Docker 镜像的方式直接运行，我们这里直接在现有的 Jenkins 环境中安装 BlueOcean 插件：登录 Jenkins Web UI -> 点击左侧的 Manage Jenkins -> Manage Plugins -> Available -> 搜索查找 BlueOcean -> 点击下载安装并重启

Filter:

Install	Name	Version
<input checked="" type="checkbox"/>	Blue Ocean Blue Ocean is a new project that rethinks the user experience of Jenkins. Designed from the ground up for Jenkins Pipeline and compatible with Freestyle jobs, Blue Ocean reduces clutter and increases clarity for every member of your team.	1.0.0
<input type="checkbox"/>	Common API for Blue Ocean	1.0.0
<input type="checkbox"/>	Config API for Blue Ocean	1.0.0
<input type="checkbox"/>	Dashboard for Blue Ocean	1.0.0
<input type="checkbox"/>	Events API for Blue Ocean	1.0.0
<input type="checkbox"/>	Git Pipeline for Blue Ocean	1.0.0
<input type="checkbox"/>	GitHub Pipeline for Blue Ocean	1.0.0
<input type="checkbox"/>	i18n for Blue Ocean	1.0.0
<input type="checkbox"/>	JWT for Blue Ocean	1.0.0
<input type="checkbox"/>	Personalization for Blue Ocean	1.0.0
<input type="checkbox"/>	Pipeline REST API for Blue Ocean	1.0.0
<input type="checkbox"/>	REST API for Blue Ocean	1.0.0
<input type="checkbox"/>	REST Implementation for Blue Ocean	1.0.0
<input type="checkbox"/>	Web for Blue Ocean	1.0.0
<input type="checkbox"/>	Blue Ocean Pipeline Editor The Blue Ocean Pipeline Editor is the simplest way for anyone wanting to get started with creating Pipelines in Jenkins	0.2.0

[Install without restart](#)
 [Download now and install after restart](#)
 [Check now](#)

install BlueOcean

一般来说 Blue Ocean 在安装后不需要额外的配置，现有 Pipeline 和 Job 将继续照常运行。但是，Blue Ocean 在首次创建或添加 Pipeline 的时候需要访问您的存储库（Git 或 GitHub）的权限，以便根据这些存储库创建 Pipeline。

安装完成后，我们可以在 Jenkins Web UI 首页左侧看到会多一个 Open Blue Ocean 的入口，我们点击就可以打开，如果之前没有创建过 Pipeline，则打开 Blue Ocean 后会看到一个 Create a new pipeline 的对话框：



blue demo2

然后我们点击开始创建一个新的 Pipeline，我们可以看到可以选择 Git、Bitbucket、GitHub，我们这里选择 GitHub，可以看到这里需要一个访问我们 GitHub 仓库权限的 token，在 GitHub 的仓库中创建一个 Personal access token：

The screenshot shows the GitHub Developer settings page for creating a personal access token. The token is named 'jenkins-demo' and has the following scopes selected:

- repo**: Full control of private repositories, repo:status, Access commit status, repo:deployment, Access deployment status, public_repo, Access public repositories, repo:invite, Access repository invitations.
- admin:org**: Full control of orgs and teams, write:org, Read and write org and team membership, read:org, Read org and team membership.
- admin:public_key**: Full control of user public keys, write:public_key, Write user public keys, read:public_key, Read user public keys.
- admin:repo_hook**: Full control of repository hooks, write:repo_hook, Write repository hooks, read:repo_hook, Read repository hooks.
- admin:org_hook**: Full control of organization hooks.
- gist**: Create gists.
- notifications**: Access notifications.
- user**: Update all user data, read:user, Read all user profile data, user:email, Access user email addresses (read-only), user:follow, Follow and unfollow users.
- delete_repo**: Delete repositories.
- write:discussion**: Read and write team discussions, read:discussion, Read team discussions.
- admin:gpg_key**: Full control of user gpg keys (Developer Preview), write:gpg_key, Write user gpg keys, read:gpg_key, Read user gpg keys.

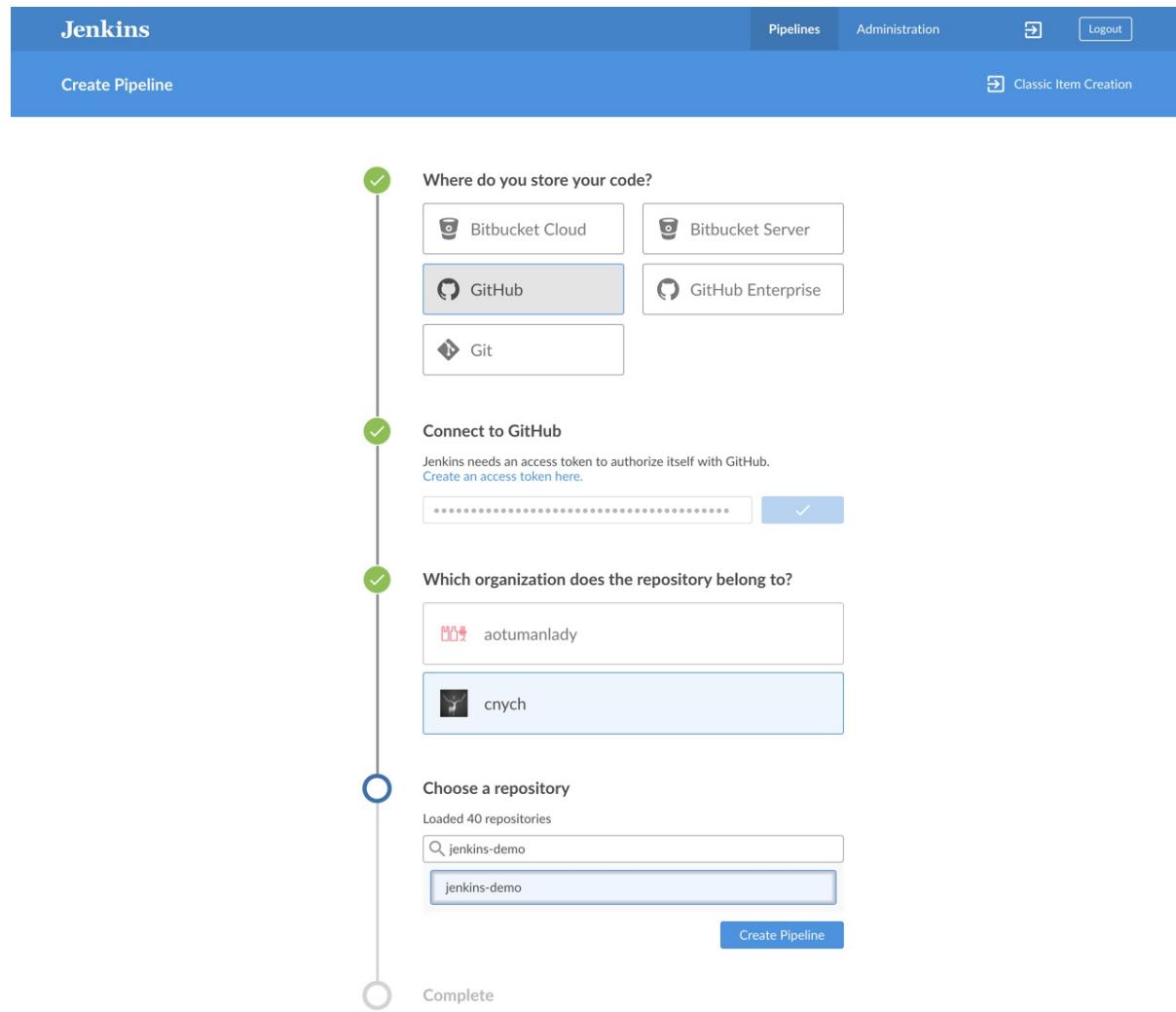
At the bottom, there are 'Update token' and 'Cancel' buttons.

Delete personal access token

Any applications or scripts using this token will no longer be able to access the GitHub API. You cannot undo this action.

Delete this token

然后将生成的 token 填入下面的创建 Pipeline 的流程中，然后我们就有权限选择自己的仓库，包括下面需要构建的仓库，比如我们这里需要构建的是 jenkins-demo 这个仓库，然后创建 Pipeline 即可：



The screenshot shows the Jenkins BlueOcean interface for creating a new pipeline. The process is a wizard with four steps:

- Where do you store your code?**: GitHub is selected. Other options include Bitbucket Cloud, Bitbucket Server, and Git.
- Connect to GitHub**: Jenkins needs an access token. A placeholder field contains '*****' and a 'Create an access token here.' link is shown.
- Which organization does the repository belong to?**: 'cnych' is selected. Other options include 'aotumanlady'.
- Choose a repository**: 'jenkins-demo' is selected from a list of 40 repositories. A search bar also shows 'jenkins-demo'. A 'Create Pipeline' button is at the bottom.

The final step is 'Complete'.

1.7.1 · Core 2.121.2 · fcec860 · 19th July 2018 02:16 AM

blue demo4

Blue Ocean 会自动扫描仓库中的每个分支，会为根文件夹中包含Jenkinsfile的每个分支创建一个Pipeline，比如我们这里有 master 和 dev 两个分支，并且两个分支下面都有 Jenkinsfile 文件，所以创建完成后会生成两个 Pipeline：

The screenshot shows the Jenkins BlueOcean interface. At the top, there's a navigation bar with tabs for Pipelines, Administration, Logout, Activity, Branches, and Pull Requests. Below the navigation bar, there's a search bar with the text 'jenkins-demo'. Underneath, a table displays two pipeline runs:

STATUS	RUN	COMMIT	BRANCH	MESSAGE	DURATION	COMPLETED
	1	-	master	Branch indexing	1m 2s	a few seconds ago
	1	-	dev	Branch indexing	1m 23s	-

blue demo5

我们可以看到有两个任务在运行了，我们可以把 master 分支的任务停止掉，我们只运行 dev 分支即可，然后我们点击 dev 这个 pipeline 就可以进入本次构建的详细页面：

The screenshot shows the Jenkins BlueOcean pipeline details page for the 'dev' branch. The pipeline has six stages: Start, Prepare, Test, Build, Push, and End. The 'Build' stage is currently active, indicated by a green checkmark. The pipeline took 7m 13s and has no changes. The commit ID is 3169344.

Build - 15s

```

    1 3.Build Docker Image Stage — Print Message
    2
    3 docker build -t cnych/jenkins-demo:dev-3169344 . — Shell Script
    4
    5 [jenkins-demo_dev_1Z6MFU1FOC1FGRPNH3SHU7IZIMHEOMHPUN2TP60CYSwHFFFFH0A] Running shell script
    6 + docker build -t cnych/jenkins-demo:dev-3169344
    7 Sending build context to Docker daemon 116.7kB
    8 Step 1/5 : FROM golang:1.8.0-alpine
    9     --> bc0935bbf1da
    10 Step 2/5 : ADD . /go/src/app
    11     --> 4dd4ab49f34
    12 Removing intermediate container 0d4dec756d46
    13 Step 3/5 : WORKDIR /go/src/app
    14     --> d218f8b6f9fd
    15 Removing intermediate container 27a44d4f90ed
    16 Step 4/5 : RUN GOOS=linux GOARCH=386 go build -v -o /go/src/app/jenkins-app
    17     --> Running in 7c6db83559fa
    18 runtime/internal/sys
    19 runtime/internal/atomic
    20 runtime/internal/runtime
    21 errors
    22 internal/race
    23 sync/atomic
    24 math
    25 unicode/utf8
    26 sync
    27 io
    28 syscall
    29 strconv
    30 time
    31 reflect
    32 os
    33 fmt
    34 app
    35     --> 2cc0de03f5de
    36 Removing intermediate container 7c6db83559fa
    37 Step 5/5 : CMD ./jenkins-app
    38     --> Running in 112105a4d636
    39     --> 2c79574b27f2
    40 Removing intermediate container 112105a4d636
    41 Successfully built 2c79574b27f2
  
```

blue demo6

在上面的图中每个阶段我们都可以点击进去查看对应的构建结果，比如我们可以查看 Push 阶段下面的日志信息：

```
...
[jenkins-demo_dev-I2WMFUIFQCIFGRPNHN3HU7IZIMHEQMHWPU2TP6DCYSWHFFFFHOA] Running shell scri
pt
+ docker push *****/jenkins-demo:dev-ee90aa5
The push refers to a repository [docker.io/***/jenkins-demo]
...
...
```

我们可以看到本次构建的 Docker 镜像的 Tag 为 dev-ee90aa5，是符合我们在 Jenkinsfile 中的定义的吧

现在我们更改下 k8s.yaml 将环境变量的值的标记改成 BRANCH_NAME，当然 Jenkinsfile 也要对应的更改，然后提交代码到 dev 分支并且 push 到 Github 仓库，我们可以看到 Jenkins Blue Ocean 里面自动触发了一次构建工作，最好同样我们可以看到本次构建能够正常完成，最后我们查看下本次构建的结果：

```
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
...
jenkins-demo-648876568d-q5mbx      0/1     Completed   3          57s
...
$ kubectl logs jenkins-demo-648876568d-q5mbx
Hello, Kubernetes! I'm from Jenkins CI!
BRANCH: dev
```

我们可以看到打印了一句 BRANCH: dev，证明我本次 CI/CD 是正常的。

现在我们来把 dev 分支的代码合并到 master 分支，然后来触发一次自动构建：

```
- jenkins-demo [dev] git status
On branch dev
nothing to commit, working directory clean
- jenkins-demo [dev] git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
- jenkins-demo [master] git merge dev
Updating 50e0401..ee90aa5
Fast-forward
 Jenkinsfile | 29 ++++++-----
 k8s.yaml    |  3 ++
 main.go     |  2 ++
 3 files changed, 14 insertions(+), 20 deletions(-)
- jenkins-demo [master] git push origin master
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:cnych/jenkins-demo.git
 50e0401..ee90aa5  master -> master
```

然后我们回到 Jenkins 的 Blue Ocean 页面中，可以看到一个 master 分支下面的任务被自动触发了，同样我们进入详情页可以查看 Push 阶段下面的日志：

```
...
[jenkins-demo_master-XA3VZ5LP4XTCFAHHXIN3G5ZB4XA4J5H6I4DNKOH6JAXZXARF7LYQ] Running shell script
+ docker push *****/jenkins-demo:ee90aa5
...
```

我们可以查看到此处推送的镜像 TAG 为 ee90aa5，没有分支的前缀，是不是和我们前面在 Jenkinsfile 中的定义是一致的，镜像推送完成后，进入 Deploy 阶段的时候我们可以看到出现了一个暂停的操作，让我们选择是否需要部署到线上，我们前面是不是定义的如果是 master 分支的话，在部署的阶段需要我们人工确认：

The screenshot shows the Jenkins BlueOcean interface for a pipeline named 'jenkins-demo'. At the top, it displays the branch 'master' and commit 'ee90aa5'. Below this is a timeline diagram with stages: Start, Prepare, Test, Build, Push, Deploy, and End. The 'Push' stage is green with a checkmark, while 'Deploy' is blue with a pause icon. Under the timeline, there's a detailed view of the 'Deploy' stage which took 1m 45s. It contains two steps: '5. Deploy Stage - Print Message' (green checkmark) and '确认要部署线上环境吗? - Wait for interactive input' (blue bar). A confirmation dialog box is open, asking '确认要部署线上环境吗?' with 'Proceed' and 'Abort' buttons.

bule demo7

然后我们点击 Proceed 才会继续后面的部署工作，确认后，我们同样可以去 Kubernetes 环境中查看下部署的结果：

```
$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
...
jenkins-demo-c69dc6fdf-6ssjf      0/1     Completed   5          4m
...
$ kubectl logs jenkins-demo-c69dc6fdf-6ssjf
Hello, Kubernetes! I'm from Jenkins CI!
BRANCH: master
```

现在我们可以看到打印出来的信息是 master，证明部署是没有问题的。

到这里我们就实现了多分支代码仓库的完整的 CI/CD 流程。

当然我们这里的示例还是太简单，只是单纯为了说明 CI/CD 的步骤，在后面的课程中，我们会结合其他的工具进一步对我们现有的方式进行改造，比如使用 Helm、Gitlab 等等。

另外如果你对声明式的 Pipeline 比较熟悉的话，我们推荐使用这种方式来编写 Jenkinsfile 文件，因为使用声明式的方式编写的 Jenkinsfile 文件在 Blue Ocean 中不但支持得非常好，我们还可以直接在 Blue Ocean Editor 中可视化的对我们的 Pipeline 进行编辑操作，非常方便。

我们在单独的课程中对 Jenkinsfile 的声明式语法进行详细讲解，感兴趣的同学可以关注下。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

63. Harbor



Harbor 是一个 CNCF 基金会托管的开源的可信的云原生 docker registry 项目，可以用于存储、签名、扫描镜像内容，Harbor 通过添加一些常用的功能如安全性、身份权限管理等来扩展 docker registry 项目，此外还支持在 registry 之间复制镜像，还提供更加高级的安全功能，如用户管理、访问控制和活动审计等，在新版本中还添加了 Helm 仓库托管的支持。

Harbor 最核心的功能就是给 docker registry 添加上一层权限保护的功能，要实现这个功能，就需要我们在使用 docker login、pull、push 等命令的时候进行拦截，先进行一些权限相关的校验，再进行操作，其实这一系列的操作 docker registry v2 就已经为我们提供了支持，v2 集成了一个安全认证的功能，将安全认证暴露给外部服务，让外部服务去实现。

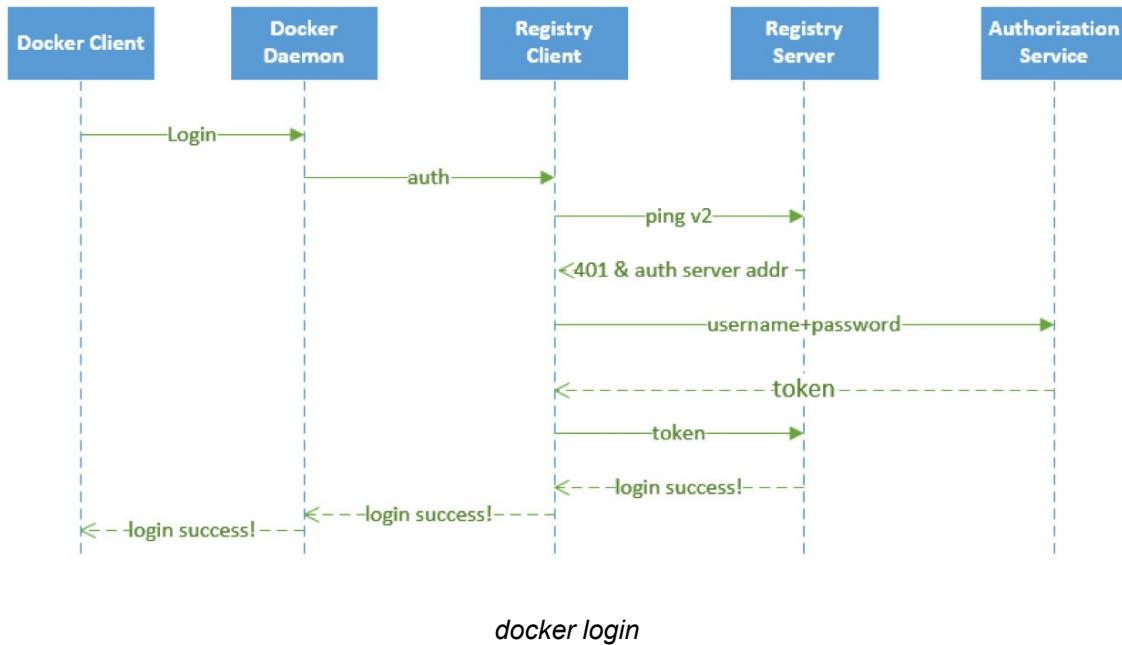
Harbor 认证原理

上面我们说了 docker registry v2 将安全认证暴露给了外部服务使用，那么是怎样暴露的呢？我们在命令行中输入 `docker login https://registry.qikqiak.com` 为例来为大家说明下认证流程：

- 1.docker client 接收到用户输入的 docker login 命令，将命令转化为调用 engine api 的 RegistryLogin 方法
- 2.在 RegistryLogin 方法中通过 http 盗用 registry 服务中的 auth 方法
- 3.因为我们这里使用的是 v2 版本的服务，所以会调用 loginV2 方法，在 loginV2 方法中会进行 /v2/ 接口调用，该接口会对请求进行认证
- 4.此时的请求中并没有包含 token 信息，认证会失败，返回 401 错误，同时会在 header 中返回去哪里请求认证的服务器地址
- 5.registry client 端收到上面的返回结果后，便会去返回的认证服务器那里进行认证请求，向认证服务器发送的请求的 header 中包含有加密的用户名和密码
- 6.认证服务器从 header 中获取到加密的用户名和密码，这个时候就可以结合实际的认证系统进行认证了，比如从数据库中查询用户认证信息或者对接 ldap 服务进行认证校验
- 7.认证成功后，会返回一个 token 信息，client 端会拿着返回的 token 再次向 registry 服务发送请求，这次需要带上得到的 token，请求验证成功，返回状态码就是200了
- 8.docker client 端接收到返回的200状态码，说明操作成功，在控制台上打印 Login Succeeded 的

信息

至此，整个登录过程完成，整个过程可以用下面的流程图来说明：



要完成上面的登录认证过程有两个关键点需要注意：怎样让 registry 服务知道服务认证地址？我们自己提供的认证服务生成的 token 为什么 registry 就能够识别？

对于第一个问题，比较好解决，registry 服务本身就提供了一个配置文件，可以在启动 registry 服务的配置文件中指定上认证服务地址即可，其中有如下这样的一段配置信息：

```

.....
auth:
  token:
    realm: token-realm
    service: token-service
    issuer: registry-token-issuer
    rootcertbundle: /root/certs/bundle
.....

```

其中 `realm` 就可以用来指定一个认证服务的地址，下面我们可以看到 Harbor 中该配置的内容

关于 registry 的配置，可以参考官方文档：<https://docs.docker.com/registry/configuration/>

第二个问题，就是 registry 怎么能够识别我们返回的 token 文件？如果按照 registry 的要求生成一个 token，是不是 registry 就可以识别了？所以我们需要在我们的认证服务器中按照 registry 的要求生成 token，而不是随便乱生成。那么要怎么生成呢？我们可以在 docker registry 的源码中可以看到 token 是通过 JWT (JSON Web Token) 来实现的，所以我们按照要求生成一个 JWT 的 token 就可以了。

对 golang 熟悉的同学可以去 clone 下 Harbor 的代码查看下，Harbor 采用 beego 这个 web 开发框架，源码阅读起来不是特别困难。我们可以很容易的看到 Harbor 中关于上面我们讲解的认证服务部分的实现方法。

安装 Harbor

Harbor 支持多种安装方式，源码目录下面默认有一个安装脚本（make/install.sh），采用 docker-compose 的形式运行 Harbor 各个组件，和前面的课程一样，我们这里依然还是将 Harbor 安装到 Kubernetes 集群中，如果我们对 Harbor 的各个组件之间的运行关系非常熟悉，同样的，我们可以自己手动编写资源清单文件进行部署，不过 Harbor 源码目录中也为我们提供了生成这些资源清单的脚本文件了（make/kubernetes/k8s-prepare），我们只需要执行下面的命令即可为我们生成所需要用到的 YAML 文件了：

```
$ python make/kubernetes/k8s-prepare
```

当然了如果上面的一些基本配置不能满足你的需求，你也可以做一些更高级的配置。你可以在 make/common/templates 目录下面找到所有的 Harbor 的配置模板，做相应的修改即可。

不过我们这里给大家介绍另外一种简单的安装方法：Helm，Harbor 官方提供了对应的 Helm Chart 包，所以我们可以很容易安装。

首先下载 Harbor Chart 包到要安装的集群上：

```
$ git clone https://github.com/goharbor/harbor-helm
```

切换到我们需要安装的分支，比如我们这里使用 1.0.0 分支：

```
$ cd harbor-helm
$ git checkout 1.0.0
```

安装 Helm Chart 包最重要的当然是 values.yaml 文件了，我们可以通过覆盖该文件中的属性来改变配置：

```
expose:
# 设置暴露服务的方式。将类型设置为 ingress、clusterIP 或 nodePort 并补充对应部分的信息。
type: ingress
tls:
# 是否开启 tls，注意：如果类型是 ingress 并且 tls 被禁用，则在 pull/push 镜像时，则必须包含端口
# 。详细查看文档：https://github.com/goharbor/harbor/issues/5291
enabled: true
# 如果你想使用自己的 TLS 证书和私钥，请填写这个 secret 的名称，这个 secret 必须包含名为 tls.crt 和 tls.key 的证书和私钥文件，如果没有设置则会自动生成证书和私钥文件。
secretName: ""
# 默认 Notary 服务会使用上面相同的证书和私钥文件，如果你想用一个独立的则填充下面的字段，注意只有类型是 ingress 的时候才需要。
notarySecretName: ""
# common name 是用于生成证书的，当类型是 clusterIP 或者 nodePort 并且 secretName 为空的时候才需要
commonName: ""
```

```

ingress:
  hosts:
    core: core.harbor.domain
    notary: notary.harbor.domain
  annotations:
    ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    ingress.kubernetes.io/proxy-body-size: "0"
    nginx.ingress.kubernetes.io/proxy-body-size: "0"
  clusterIP:
    # ClusterIP 服务的名称
    name: harbor
  ports:
    httpPort: 80
    httpsPort: 443
    # Notary 服务监听端口, 只有当 notary.enabled 设置为 true 的时候有效
    notaryPort: 4443
  nodePort:
    # NodePort 服务名称
    name: harbor
    ports:
      http:
        port: 80
        nodePort: 30002
      https:
        port: 443
        nodePort: 30003
      notary:
        port: 4443
        nodePort: 30004

# Harbor 核心服务外部访问 URL。主要用于:
# 1) 补全 portal 页面上面显示的 docker/helm 命令
# 2) 补全返回给 docker/notary 客户端的 token 服务 URL

# 格式: protocol://domain[:port]。
# 1) 如果 expose.type=ingress, "domain"的值就是 expose.ingress.hosts.core 的值
# 2) 如果 expose.type=clusterIP, "domain"的值就是 expose.clusterIP.name 的值
# 3) 如果 expose.type=nodePort, "domain"的值就是 k8s 节点的 IP 地址

# 如果在代理后部部署 Harbor, 请将其设置为代理的 URL
externalURL: https://core.harbor.domain

# 默认情况下开启数据持久化, 在k8s集群中需要动态的挂载卷默认需要一个StorageClass对象。
# 如果你有已经存在可以使用的持久卷, 需要在"storageClass"中指定你的 storageClass 或者设置 "existingClaim"。
#
# 对于存储 docker 镜像和 Helm charts 包, 你也可以用 "azure"、"gcs"、"s3"、"swift" 或者 "oss",
直接在 "imageChartStorage" 区域设置即可
persistence:
  enabled: true
  # 设置成"keep"避免在执行 helm 删除操作期间移除 PVC, 留空则在 chart 被删除后删除 PVC
  resourcePolicy: "keep"
  persistentVolumeClaim:
    registry:
      # 使用一个存在的 PVC(必须在绑定前先手动创建)
      existingClaim: ""
      # 指定"storageClass", 或者使用默认的 StorageClass 对象, 设置成"-禁用动态分配挂载卷
      storageClass: ""
      subPath: ""

```

```

accessMode: ReadWriteOnce
size: 5Gi
chartmuseum:
existingClaim: ""
storageClass: ""
subPath: ""
accessMode: ReadWriteOnce
size: 5Gi
jobservice:
existingClaim: ""
storageClass: ""
subPath: ""
accessMode: ReadWriteOnce
size: 1Gi
# 如果使用外部的数据库服务，下面的设置将会被忽略
database:
existingClaim: ""
storageClass: ""
subPath: ""
accessMode: ReadWriteOnce
size: 1Gi
# 如果使用外部的 Redis 服务，下面的设置将会被忽略
redis:
existingClaim: ""
storageClass: ""
subPath: ""
accessMode: ReadWriteOnce
size: 1Gi
# 定义使用什么存储后端来存储镜像和 charts 包，详细文档地址: https://github.com/docker/distribution/blob/master/docs/configuration.md#storage
imageChartStorage:
# 正对镜像和chart存储是否禁用跳转，对于一些不支持的后端(例如对于使用minio的`s3`存储)，需要禁用它。为了禁止跳转，只需要设置`disableredirect=true`即可，详细文档地址: https://github.com/docker/distribution/blob/master/docs/configuration.md#redirect
disableredirect: false
# 指定存储类型: "filesystem", "azure", "gcs", "s3", "swift", "oss"，在相应的区域填上对应的信息。
# 如果你想使用 pv 则必须设置成"filesystem"类型
type: filesystem
filesystem:
rootdirectory: /storage
#maxthreads: 100
azure:
accountname: accountname
accountkey: base64encodedaccountkey
container: containername
realm: core.windows.net
gcs:
bucket: bucketname
# The base64 encoded json file which contains the key
encodedkey: base64-encoded-json-key-file
#rootdirectory: /gcs/object/name/prefix
#chunksize: "5242880"
s3:
region: us-west-1
bucket: bucketname
#accesskey: awsaccesskey
#secretkey: awsssecretkey
#regionendpoint: http://myobjects.local
#encrypt: false

```

```

#keyid: mykeyid
#secure: true
#v4auth: true
#chunksize: "5242880"
#rootdirectory: /s3/object/name/prefix
#storageclass: STANDARD

swift:
  authurl: https://storage.myprovider.com/v3/auth
  username: username
  password: password
  container: containername
  #region: fr
  #tenant: tenantname
  #tenantid: tenantid
  #domain: domainname
  #domainid: domainid
  #trustid: trustid
  #insecureskipverify: false
  #chunksize: 5M
  #prefix:
  #secretkey: secretkey
  #accesskey: accesskey
  #authversion: 3
  #endpointtype: public
  #tempurlcontainerkey: false
  #tempurlmethods:

oss:
  accesskeyid: accesskeyid
  accesskeysecret: accesskeysecret
  region: regionname
  bucket: bucketname
  #endpoint: endpoint
  #internal: false
  #encrypt: false
  #secure: true
  #chunksize: 10M
  #rootdirectory: rootdirectory

imagePullPolicy: IfNotPresent

logLevel: debug
# Harbor admin 初始密码, Harbor 启动后通过 Portal 修改该密码
harborAdminPassword: "Harbor12345"
# 用于加密的一个 secret key, 必须是一个16位的字符串
secretKey: "not-a-secure-key"

# 如果你通过"ingress"保留服务, 则下面的Nginx不会被使用
nginx:
  image:
    repository: goharbor/nginx-photon
    tag: v1.7.0
  replicas: 1
  # resources:
  #   requests:
  #     memory: 256Mi
  #     cpu: 100m
  nodeSelector: {}
  tolerations: []
  affinity: {}
## 额外的 Deployment 的一些 annotations

```

```
podAnnotations: {}

portal:
  image:
    repository: goharbor/harbor-portal
    tag: v1.7.0
    replicas: 1
  # resources:
  #   requests:
  #     memory: 256Mi
  #     cpu: 100m
  #   nodeSelector: {}
  #   tolerations: []
  #   affinity: {}
  podAnnotations: {}

core:
  image:
    repository: goharbor/harbor-core
    tag: v1.7.0
    replicas: 1
  # resources:
  #   requests:
  #     memory: 256Mi
  #     cpu: 100m
  #   nodeSelector: {}
  #   tolerations: []
  #   affinity: {}
  podAnnotations: {}

adminserver:
  image:
    repository: goharbor/harbor-adminserver
    tag: v1.7.0
    replicas: 1
  # resources:
  #   requests:
  #     memory: 256Mi
  #     cpu: 100m
  #   nodeSelector: {}
  #   tolerations: []
  #   affinity: {}
  podAnnotations: {}

jobservice:
  image:
    repository: goharbor/harbor-jobservice
    tag: v1.7.0
    replicas: 1
    maxJobWorkers: 10
  # jobs 的日志收集器: "file", "database" or "stdout"
  jobLogger: file
  # resources:
  #   requests:
  #     memory: 256Mi
  #     cpu: 100m
  #   nodeSelector: {}
  #   tolerations: []
  #   affinity: {}
  podAnnotations: {}
```

```
registry:
  registry:
    image:
      repository: goharbor/registry-photon
      tag: v2.6.2-v1.7.0
  controller:
    image:
      repository: goharbor/harbor-registryctl
      tag: v1.7.0
  replicas: 1
  nodeSelector: {}
  tolerations: []
  affinity: {}
  podAnnotations: {}

chartmuseum:
  enabled: true
  image:
    repository: goharbor/chartmuseum-photon
    tag: v0.7.1-v1.7.0
  replicas: 1
  # resources:
  #   requests:
  #     memory: 256Mi
  #     cpu: 100m
  nodeSelector: {}
  tolerations: []
  affinity: {}
  podAnnotations: {}

clair:
  enabled: true
  image:
    repository: goharbor/clair-photon
    tag: v2.0.7-v1.7.0
  replicas: 1
  # 用于从 Internet 更新漏洞数据库的http(s)代理
  httpProxy:
  httpsProxy:
  # clair 更新程序的间隔, 单位为小时, 设置为0来禁用
  updatersInterval: 12
  # resources:
  #   requests:
  #     memory: 256Mi
  #     cpu: 100m
  nodeSelector: {}
  tolerations: []
  affinity: {}
  podAnnotations: {}

notary:
  enabled: true
  server:
    image:
      repository: goharbor/notary-server-photon
      tag: v0.6.1-v1.7.0
    replicas: 1
    # resources:
    #   requests:
```

```

#      memory: 256Mi
#      cpu: 100m
signer:
  image:
    repository: goharbor/notary-signer-photon
    tag: v0.6.1-v1.7.0
  replicas: 1
  # resources:
  # requests:
  #      memory: 256Mi
  #      cpu: 100m
  nodeSelector: {}
  tolerations: []
  affinity: {}
  podAnnotations: {}

database:
  # 如果使用外部的数据库，则设置 type=external，然后填写 external 区域的一些连接信息
  type: internal
  internal:
    image:
      repository: goharbor/harbor-db
      tag: v1.7.0
    # 内部的数据库的初始化超级用户的密码
    password: "changeit"
    # resources:
    # requests:
    #      memory: 256Mi
    #      cpu: 100m
    nodeSelector: {}
    tolerations: []
    affinity: {}
  external:
    host: "192.168.0.1"
    port: "5432"
    username: "user"
    password: "password"
    coreDatabase: "registry"
    clairDatabase: "clair"
    notaryServerDatabase: "notary_server"
    notarySignerDatabase: "notary_signer"
    sslmode: "disable"
  podAnnotations: {}

redis:
  # 如果使用外部的 Redis 服务，设置 type=external，然后补充 external 部分的连接信息。
  type: internal
  internal:
    image:
      repository: goharbor/redis-photon
      tag: v1.7.0
    # resources:
    # requests:
    #      memory: 256Mi
    #      cpu: 100m
    nodeSelector: {}
    tolerations: []
    affinity: {}
  external:
    host: "192.168.0.2"

```

```

port: "6379"
# coreDatabaseIndex 必须设置为0
coreDatabaseIndex: "0"
jobserviceDatabaseIndex: "1"
registryDatabaseIndex: "2"
chartmuseumDatabaseIndex: "3"
password: ""
podAnnotations: {}

```

有了上面的配置说明，则我们可以根据自己的需求来覆盖上面的值，比如我们这里新建一个 qikqiak-values.yaml 的文件，文件内容如下：

```

expose:
  type: ingress
  tls:
    enabled: true
ingress:
  hosts:
    core: registry.qikqiak.com
    notary: notary.qikqiak.com
  annotations:
    kubernetes.io/ingress.class: "traefik"
    ingress.kubernetes.io/ssl-redirect: "true"
    ingress.kubernetes.io/proxy-body-size: "0"

  externalURL: https://registry.qikqiak.com

persistence:
  enabled: true
  resourcePolicy: "keep"
  persistentVolumeClaim:
    registry:
      storageClass: "harbor-data"
    chartmuseum:
      storageClass: "harbor-data"
    jobservice:
      storageClass: "harbor-data"
    database:
      storageClass: "harbor-data"
    redis:
      storageClass: "harbor-data"

```

其中需要我们定制的部分很少，我们将域名替换成我们自己的，使用默认的 Ingress 方式暴露服务，其他需要我们手动配置的部分就是数据持久化的部分，我们需要提前为上面的这些服务创建好可用的 PVC 或者 StorageClass 对象，比如我们这里使用一个名为 harbor-data 的 StorageClass 资源对象，当然也可以根据我们实际的需求修改 accessMode 或者存储容量：(harbor-data-sc.yaml)

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: harbor-data
provisioner: fuseim.pri/ifs

```

先新建上面的 StorageClass 资源对象：

```
$ kubectl create -f harbor-data-sc.yaml
storageclass.storage.k8s.io "harbor-data" created
```

创建完成后，使用上面自定义的 values 文件安装：

```
$ helm install --name harbor -f qikqiak-values.yaml . --namespace kube-ops
NAME: harbor
LAST DEPLOYED: Fri Feb 22 22:39:22 2019
NAMESPACE: kube-ops
STATUS: DEPLOYED

RESOURCES:
==> v1/StatefulSet
NAME          DESIRED  CURRENT  AGE
harbor-harbor-database  1        1        0s
harbor-harbor-redis    1        1        0s

==> v1beta1/Ingress
NAME          HOSTS          ADDRESS      PORTS  AGE
harbor-harbor-ingress  registry.qikqiak.com,notary.qikqiak.com  80, 443  0s

==> v1/Pod(related)
NAME          READY  STATUS          RESTARTS  AGE
harbor-harbor-adminserver-58c855568c-jnpvq  0/1  ContainerCreating  0  0s
harbor-harbor-chartmuseum-58d6c9b898-4csmd  0/1  Pending          0  0s
harbor-harbor-clair-5c7689585-hd2br       0/1  ContainerCreating  0  0s
harbor-harbor-core-6f56879469-rbthd       0/1  ContainerCreating  0  0s
harbor-harbor-jobservice-74d7795cdb-bhzdm  0/1  ContainerCreating  0  0s
harbor-harbor-notary-server-69cdbfb56-ggc49  0/1  Pending          0  0s
harbor-harbor-notary-signer-8499dc4db7-f78cd 0/1  Pending          0  0s
harbor-harbor-portal-55c45c558d-dmj48     0/1  Pending          0  0s
harbor-harbor-registry-5569fcbf78-5grds   0/2  Pending          0  0s
harbor-harbor-database-0                  0/1  Pending          0  0s
harbor-harbor-redis-0                   0/1  Pending          0  0s

==> v1/Secret
NAME          TYPE  DATA  AGE
harbor-harbor-adminserver  Opaque  4  1s
harbor-harbor-chartmuseum  Opaque  1  1s
harbor-harbor-core         Opaque  4  1s
harbor-harbor-database    Opaque  1  1s
harbor-harbor-ingress     kubernetes.io/tls  3  1s
harbor-harbor-jobservice  Opaque  1  1s
harbor-harbor-registry    Opaque  1  1s

==> v1/ConfigMap
NAME          DATA  AGE
harbor-harbor-adminserver 39  1s
harbor-harbor-chartmuseum 24  1s
harbor-harbor-clair       1   1s
harbor-harbor-core        1   1s
harbor-harbor-jobservice  1   1s
harbor-harbor-notary-server 5   1s
harbor-harbor-registry    2   1s

==> v1/PersistentVolumeClaim
NAME          STATUS  VOLUME           CAPACITY  AC
CESS MODES  STORAGECLASS  AGE
```

harbor-harbor-chartmuseum	Pending	harbor-data		1s		
harbor-harbor-jobservice	Bound	pvc-a8a35d0e-36af-11e9-bcd8-525400db4df7	1Gi	RWO	ha	rbor-data 1s
harbor-harbor-registry	Bound	pvc-a8a466e9-36af-11e9-bcd8-525400db4df7	5Gi	RWO	ha	rbor-data 1s
==> v1/Service						
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	
harbor-harbor-adminserver	ClusterIP	10.108.3.242	<none>	80/TCP	1s	E
harbor-harbor-chartmuseum	ClusterIP	10.101.49.103	<none>	80/TCP	1s	
harbor-harbor-clair	ClusterIP	10.110.173.153	<none>	6060/TCP	1s	
harbor-harbor-core	ClusterIP	10.105.178.198	<none>	80/TCP	1s	
harbor-harbor-database	ClusterIP	10.102.101.155	<none>	5432/TCP	0s	
harbor-harbor-jobservice	ClusterIP	10.100.127.32	<none>	80/TCP	0s	
harbor-harbor-notary-server	ClusterIP	10.105.25.64	<none>	4443/TCP	0s	
harbor-harbor-notary-signer	ClusterIP	10.108.92.82	<none>	7899/TCP	0s	
harbor-harbor-portal	ClusterIP	10.103.111.161	<none>	80/TCP	0s	
harbor-harbor-redis	ClusterIP	10.107.205.3	<none>	6379/TCP	0s	
harbor-harbor-registry	ClusterIP	10.100.87.29	<none>	5000/TCP,8080/TCP	0s	
==> v1/Deployment						
NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE	
harbor-harbor-adminserver	1	1	1	0	0s	
harbor-harbor-chartmuseum	1	1	1	0	0s	
harbor-harbor-clair	1	1	1	0	0s	
harbor-harbor-core	1	1	1	0	0s	
harbor-harbor-jobservice	1	1	1	0	0s	
harbor-harbor-notary-server	1	1	1	0	0s	
harbor-harbor-notary-signer	1	1	1	0	0s	
harbor-harbor-portal	1	1	1	0	0s	
harbor-harbor-registry	1	0	0	0	0s	

NOTES:

Please wait **for** several minutes **for** Harbor deployment to complete.
 Then you should be able to visit the Harbor portal at <https://registry.qikqiak.com>.
 For more details, please visit <https://github.com/goharbor/harbor>.

上面是我们通过 Helm 安装所有涉及到的一些资源对象，稍微等一会儿，就可以安装成功了，查看对应的 Pod 状态：

\$ kubectl get pods -n kube-ops						
NAME	READY	STATUS	RESTARTS	AGE		
harbor-harbor-adminserver-58c855568c-7dqqb	1/1	Running	0	37m		
harbor-harbor-chartmuseum-58d6c9b898-4csmd	1/1	Running	0	49m		
harbor-harbor-clair-5c7689585-hd2br	1/1	Running	0	49m		
harbor-harbor-core-6f56879469-rbthd	1/1	Running	8	49m		
harbor-harbor-database-0	1/1	Running	0	49m		
harbor-harbor-jobservice-74d7795cdb-bhzdm	1/1	Running	7	49m		
harbor-harbor-notary-server-69cdbdfb56-vklbt	1/1	Running	0	20m		
harbor-harbor-notary-signer-8499dc4db7-f78cd	1/1	Running	0	49m		
harbor-harbor-portal-55c45c558d-dmj48	1/1	Running	0	49m		
harbor-harbor-redis-0	1/1	Running	0	49m		
harbor-harbor-registry-5569fcbf78-5grds	2/2	Running	0	49m		

现在都是 Running 状态了，都成功运行起来了，查看下对应的 Ingress 对象：

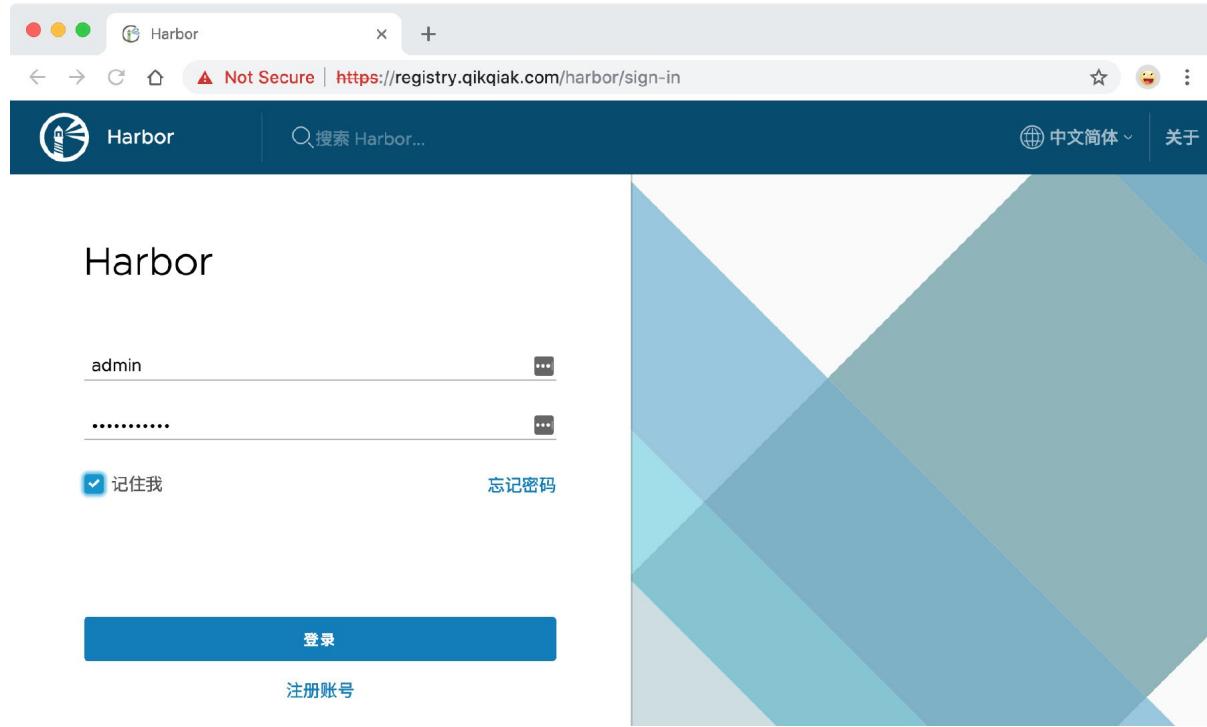
```
$ kubectl get ingress -n kube-ops
NAME          HOSTS
harbor-harbor-ingress  registry.qikqiak.com,notary.qikqiak.com
                        ADDRESS      PORTS      AGE
                        80, 443    50m
```

如果你有自己的真正的域名，则将上面的两个域名解析到你的任意一个 Ingress Controller 的 Pod 所在的节点即可，我们这里为了演示方便，还是自己在本地的 `/etc/hosts` 里面添加上 `registry.qikqiak.com` 和 `notary.qikqiak.com` 的映射。

Harbor 使用

Harbor Portal

添加完成后，在浏览器中输入 `registry.qikqiak.com` 就可以打开熟悉的 Harbor 的 Portal 界面了，当然我们配置的 Ingress 中会强制跳转到 https，所以如果你的浏览器有什么安全限制的话，需要信任我们这里 Ingress 对应的证书，证书文件可以通过查看 Secret 资源对象获取：



然后输入用户名：admin，密码：Harbor12345（当然我们也可以通过 Helm 安装的时候自己覆盖 `harborAdminPassword`）即可登录进入 Portal 首页：

The screenshot shows the Harbor Portal Home page. The left sidebar includes links for '项目' (Projects), '日志' (Logs), '系统管理' (System Management) (with '用户管理' (User Management), '仓库管理' (Repository Management), '复制管理' (Copy Management), and '配置管理' (Configuration Management)), and a '帮助' (Help) link. The main content area is titled '项目' (Projects) and displays a table of projects. The table has columns for '项目' (Project), '私有' (Private), '公开' (Public), and '总计' (Total). It also shows statistics for '镜像仓库' (Image Repositories) and 'Helm Chart 数目' (Number of Helm Charts). A search bar at the top right allows filtering by '所有项目' (All Projects). The table lists one project named 'library'.

项目	0 私有	1 公开	1 总计
library	0 私有	1 公开	0 总计

1 - 1 共计 1 条记录

Harbor Portal Home

我们可以看到有很多功能，默认情况下会有一个名叫 library 的项目，改项目默认是公开访问权限的，进入项目可以看到里面还有 Helm Chart 包的管理，可以手动在这里上传，也可以对改项目里面的镜像进行一些配置，比如是否开启自动扫描镜像功能：

项目 < library 系统管理员

镜像仓库 Helm Charts 成员 复制 标签 日志 配置管理

项目仓库 公开
所有人都可访问公开的项目仓库。

部署安全 内容信任
仅允许部署通过认证的镜像。
 阻止潜在漏洞镜像
阻止危害级别 严重 ▾ 以上的镜像运行。

漏洞扫描 自动扫描镜像
当镜像上传后，自动进行扫描

保存 取消

Harbor project settings

docker cli

然后我们来测试下使用 docker cli 来进行 pull/push 镜像，由于上面我们安装的时候通过 Ingress 来暴露的 Harbor 的服务，而且强制使用了 https，所以如果我们要在终端中使用我们这里的私有仓库的话，就需要配置上相应的证书：

```
$ docker login registry.qikqiak.com
Warning: failed to get default registry endpoint from daemon (Cannot connect to the Docker
daemon at unix:///var/run/docker.sock. Is the docker daemon running?). Using system defau
lt: https://index.docker.io/v1/
Username: admin
Password:
INFO[0007] Error logging in to v2 endpoint, trying next endpoint: Get https://registry.qik
qiak.com/v2/: x509: certificate has expired or is not yet valid
INFO[0007] Error logging in to v1 endpoint, trying next endpoint: Get https://registry.qik
qiak.com/v1/users/: x509: certificate has expired or is not yet valid
Get https://registry.qikqiak.com/v1/users/: x509: certificate has expired or is not yet va
lid
```

这是因为我们没有提供证书文件，我们将使用到的 ca.crt 文件复制到 /etc/docker/certs.d/registry.qikqiak.com 目录下面，如果该目录不存在，则创建它。ca.crt 这个证书文件我们可以通过 Ingress 中使用的 Secret 资源对象来提供：

```
$ kubectl get secret harbor-harbor-ingress -n kube-ops -o yaml
apiVersion: v1
data:
  ca.crt: <ca.crt>
  tls.crt: <tls.crt>
  tls.key: <tls.key>
kind: Secret
metadata:
  creationTimestamp: 2019-02-22T14:39:28Z
  labels:
    app: harbor
    chart: harbor
    heritage: Tiller
    release: harbor
  name: harbor-harbor-ingress
  namespace: kube-ops
  resourceVersion: "50400208"
  selfLink: /api/v1/namespaces/kube-ops/secrets/harbor-harbor-ingress
  uid: a899c57a-36af-11e9-bcd8-525400db4df7
type: kubernetes.io/tls
```

其中 data 区域中 ca.crt 对应的值就是我们需要证书，不过需要注意还需要做一个 base64 的解码，这样证书配置上以后就可以正常访问了。

不过由于上面的方法较为繁琐，所以一般情况下我们在使用 docker cli 的时候是在 docker 启动参数后面添加一个 --insecure-registry 参数来忽略证书的校验的，在 docker 启动配置文件 /usr/lib/systemd/system/docker.service 中修改 ExecStart 的启动参数：

```
ExecStart=/usr/bin/dockerd --insecure-registry registry.qikqiak.com
```

然后保存重启 docker，再使用 docker cli 就没有任何问题了：

```
$ docker login registry.qikqiak.com
Username: admin
Password:
Login Succeeded
```

比如我们本地现在有一个名为 busybox 的镜像，现在我们想要将该镜像推送到我们的私有仓库中去，应该怎样操作呢？首先我们需要给该镜像重新打一个 registry.qikqiak.com 的前缀，然后推送的时候就可以识别到推送到哪个镜像仓库：

```
$ docker tag busybox registry.qikqiak.com/library/busybox
$ docker push registry.qikqiak.com/library/busybox
The push refers to repository [registry.qikqiak.com/library/busybox]
adab5d09ba79: Pushed
latest: digest: sha256:4415a904b1aca178c2450fd54928ab362825e863c0ad5452fd020e92f7a6a47e size: 527
```

推送完成后，我们同样可以在 Portal 页面上看到这个镜像的信息：

The screenshot shows the Harbor Portal interface. At the top, there's a header bar with the Harbor logo, a search bar, and language and user dropdowns. Below the header, the main content area shows a project named 'library' with a member named '系统管理员'. Underneath, there's a table listing a single image entry: 'library/busybox' with 1 tag and 0 downloads. A warning message at the bottom right states: '缺陷数据库可能没有完全准备好! 1 - 1 共计 1 条记录'.

Harbor image info

镜像 push 成功，同样可以测试下 pull：

```
$ docker rmi registry.qikqiak.com/library/busybox
Untagged: registry.qikqiak.com/library/busybox:latest
Untagged: registry.qikqiak.com/library/busybox@sha256:4415a904b1aca178c2450fd54928ab362825
e863c0ad5452fd020e92f7a6a47e

$ docker pull registry.qikqiak.com/library/busybox:latest
latest: Pulling from library/busybox
Digest: sha256:4415a904b1aca178c2450fd54928ab362825e863c0ad5452fd020e92f7a6a47e
Status: Downloaded newer image for registry.qikqiak.com/library/busybox:latest

$ docker images |grep busybox
busybox                         latest              d8233ab899d4      7 days ago
                                1.2MB
registry.qikqiak.com/library/busybox    latest              d8233ab899d4      7 days ago
                                1.2MB
```

到这里证明上面我们的私有 docker 仓库搭建成功了，大家可以尝试去创建一个私有的项目，然后创建一个新的用户，使用这个用户来进行 pull/push 镜像，Harbor 还具有其他的一些功能，比如镜像复制，大家可以自行测试，感受下 Harbor 和官方自带的 registry 仓库的差别。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号，在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



*k8s技术圈*二维码

64. Gitlab 的安装使用

Gitlab 官方提供了 Helm 的方式在 Kubernetes 集群中来快速安装，但是在使用的过程中发现 Helm 提供的 Chart 包中有很多其他额外的配置，所以我们这里使用自定义的方式来安装，也就是自己来定义一些资源清单文件。

Gitlab 主要涉及到3个应用：Redis、Postgresql、Gitlab 核心程序，实际上我们只要将这3个应用分别启动起来，然后加上对应的配置就可以很方便的安装 Gitlab 了，我们这里选择使用的镜像不是官方的，而是 Gitlab 容器化中使用非常多的一个第三方镜像：`sameersbn/gitlab`，基本上和官方保持同步更新，地址：<http://www.damagehead.com/docker-gitlab/>

如果我们已经有可使用的 Redis 或 Postgresql 服务的话，那么直接配置在 Gitlab 环境变量中即可，如果没有的话就单独部署。

首先部署需要的 Redis 服务，对应的资源清单文件如下：(gitlab-redis.yaml)

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: redis
  namespace: kube-ops
  labels:
    name: redis
spec:
  template:
    metadata:
      name: redis
      labels:
        name: redis
    spec:
      containers:
        - name: redis
          image: sameersbn/redis
          imagePullPolicy: IfNotPresent
          ports:
            - name: redis
              containerPort: 6379
          volumeMounts:
            - mountPath: /var/lib/redis
              name: data
      livenessProbe:
        exec:
          command:
            - redis-cli
            - ping
        initialDelaySeconds: 30
        timeoutSeconds: 5
      readinessProbe:
        exec:
          command:
            - redis-cli
            - ping
        initialDelaySeconds: 5
        timeoutSeconds: 1
```

```

  volumes:
    - name: data
      emptyDir: {}

  ...
  apiVersion: v1
  kind: Service
  metadata:
    name: redis
    namespace: kube-ops
    labels:
      name: redis
  spec:
    ports:
      - name: redis
        port: 6379
        targetPort: redis
    selector:
      name: redis

```

然后是数据库 Postgresql，对应的资源清单文件如下：(gitlab-postgresql.yaml)

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: postgresql
  namespace: kube-ops
  labels:
    name: postgresql
spec:
  template:
    metadata:
      name: postgresql
    labels:
      name: postgresql
    spec:
      containers:
        - name: postgresql
          image: sameersbn/postgresql:10
          imagePullPolicy: IfNotPresent
          env:
            - name: DB_USER
              value: gitlab
            - name: DB_PASS
              value: passw0rd
            - name: DB_NAME
              value: gitlab_production
            - name: DB_EXTENSION
              value: pg_trgm
          ports:
            - name: postgres
              containerPort: 5432
          volumeMounts:
            - mountPath: /var/lib/postgresql
              name: data
      livenessProbe:
        exec:
          command:
            - pg_isready

```

```

      - -h
      - localhost
      - -U
      - postgres
  initialDelaySeconds: 30
  timeoutSeconds: 5
readinessProbe:
  exec:
    command:
      - pg_isready
      - -h
      - localhost
      - -U
      - postgres
  initialDelaySeconds: 5
  timeoutSeconds: 1
volumes:
  - name: data
    emptyDir: {}

---
apiVersion: v1
kind: Service
metadata:
  name: postgresql
  namespace: kube-ops
  labels:
    name: postgresql
spec:
  ports:
    - name: postgres
      port: 5432
      targetPort: postgres
  selector:
    name: postgresql

```

然后就是我们最核心的 Gitlab 的应用，对应的资源清单文件如下：(gitlab.yaml)

```

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: gitlab
  namespace: kube-ops
  labels:
    name: gitlab
spec:
  template:
    metadata:
      name: gitlab
      labels:
        name: gitlab
    spec:
      containers:
        - name: gitlab
          image: sameersbn/gitlab:11.8.1
          imagePullPolicy: IfNotPresent
          env:
            - name: TZ
              value: Asia/Shanghai

```

```
- name: GITLAB_TIMEZONE
  value: Beijing
- name: GITLAB_SECRETS_DB_KEY_BASE
  value: long-and-random-alpha-numeric-string
- name: GITLAB_SECRETS_SECRET_KEY_BASE
  value: long-and-random-alpha-numeric-string
- name: GITLAB_SECRETS_OTP_KEY_BASE
  value: long-and-random-alpha-numeric-string
- name: GITLAB_ROOT_PASSWORD
  value: admin321
- name: GITLAB_ROOT_EMAIL
  value: 517554016@qq.com
- name: GITLAB_HOST
  value: git.qikqiak.com
- name: GITLAB_PORT
  value: "80"
- name: GITLAB_SSH_PORT
  value: "22"
- name: GITLAB_NOTIFY_ON_BROKEN_BUILDS
  value: "true"
- name: GITLAB_NOTIFY_PUSHER
  value: "false"
- name: GITLAB_BACKUP_SCHEDULE
  value: daily
- name: GITLAB_BACKUP_TIME
  value: 01:00
- name: DB_TYPE
  value: postgres
- name: DB_HOST
  value: postgresql
- name: DB_PORT
  value: "5432"
- name: DB_USER
  value: gitlab
- name: DB_PASS
  value: passw0rd
- name: DB_NAME
  value: gitlab_production
- name: REDIS_HOST
  value: redis
- name: REDIS_PORT
  value: "6379"
ports:
- name: http
  containerPort: 80
- name: ssh
  containerPort: 22
volumeMounts:
- mountPath: /home/git/data
  name: data
livenessProbe:
  httpGet:
    path: /
    port: 80
    initialDelaySeconds: 180
    timeoutSeconds: 5
readinessProbe:
  httpGet:
    path: /
    port: 80
```

```

    initialDelaySeconds: 5
    timeoutSeconds: 1
  volumes:
  - name: data
    emptyDir: {}

  ---
apiVersion: v1
kind: Service
metadata:
  name: gitlab
  namespace: kube-ops
  labels:
    name: gitlab
spec:
  ports:
  - name: http
    port: 80
    targetPort: http
  - name: ssh
    port: 22
    targetPort: ssh
  selector:
    name: gitlab

  ---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: gitlab
  namespace: kube-ops
  annotations:
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: git.qikqiak.com
    http:
      paths:
      - backend:
          serviceName: gitlab
          servicePort: http

```

要注意我们这里应用数据没有做数据持久化，只是使用的 `emptyDir: {}` 类型的 volume，Pod 挂掉后，对应的数据也就没有了，所以要在正式的环境中使用一定要做数据的持久化，比如添加 PV/PVC 或者 StorageClass。

要注意的是其中 Redis 和 Postgresql 相关的环境变量配置，另外，我们这里添加了一个 Ingress 对象，来为我们的 Gitlab 配置一个域名 `git.qikqiak.com`，这样应用部署完成后，我们就可以通过该域名来访问了，然后直接部署即可：

```
$ kubectl create -f gitlab-redis.yaml gitlab-postgresql.yaml gitlab.yaml
```

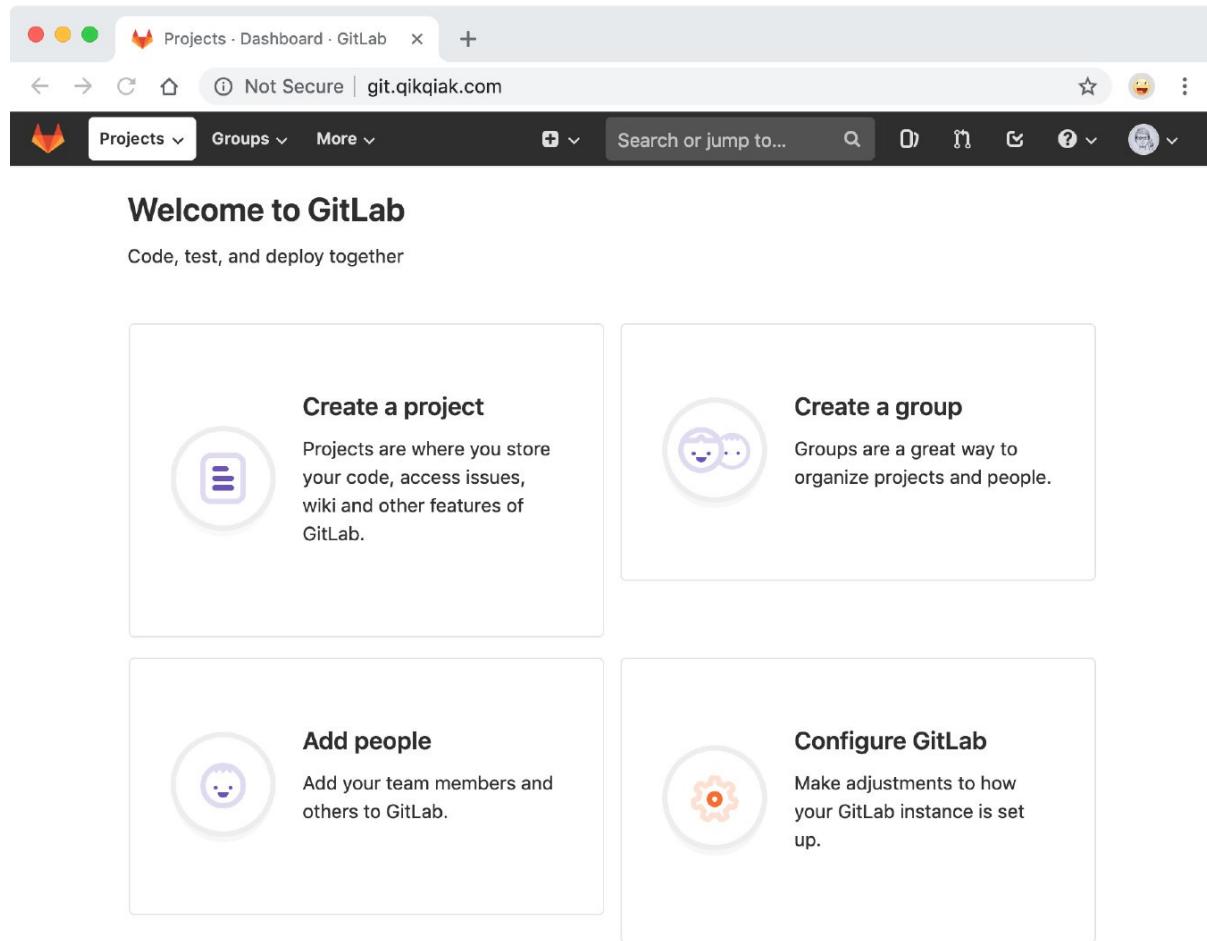
创建完成后，查看 Pod 的部署状态：

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

gitlab-7d855554cb-twh7c	1/1	Running	0	10m
postgresql-8566bb959c-2tnvr	1/1	Running	0	17h
redis-8446f57bdf-4v62p	1/1	Running	0	17h

可以看到都已经部署成功了，然后我们可以通过 Ingress 中定义的域名 `git.qikqiak.com` (需要做 DNS 解析或者在本地 `/etc/hosts` 中添加映射) 来访问 Portal：

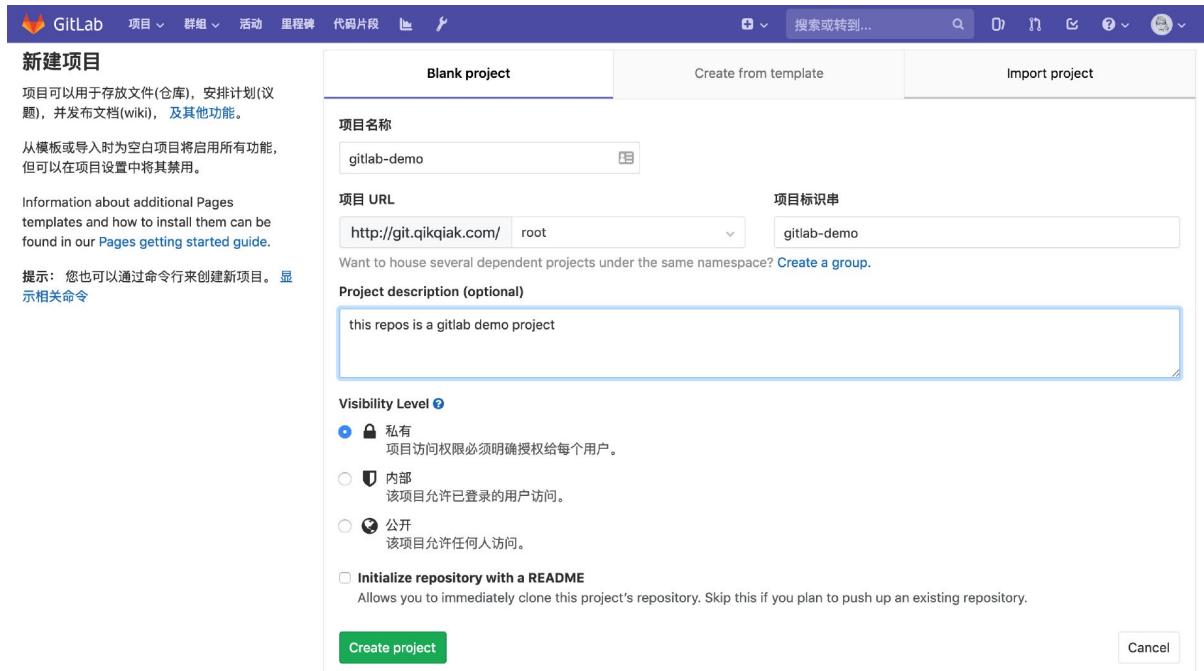
使用用户名 `root`, 和部署的时候指定的超级用户密码 `GITLAB_ROOT_PASSWORD=admin321` 即可登录进入到首页：



gitlab homepage

Gitlab 运行后，我们可以注册为新用户并创建一个项目，还可以做很多的其他系统设置，比如设置语言、设置应用风格样式等等。

点击 `Create a project` 创建一个新的项目，和之前 Github 使用上没有多大的差别：



create gitlab project

创建完成后，我们可以添加本地用户的一个 `SSH-KEY`，这样我们就可以通过 SSH 来拉取或者推送代码了。SSH 公钥通常包含在 `~/.ssh/id_rsa.pub` 文件中，并以 `ssh-rsa` 开头。如果没有的话可以使用 `ssh-keygen` 命令来生成，`id_rsa.pub` 里面的内容就是我们需要的 SSH 公钥，然后添加到 Gitlab 中。

由于平时使用的 ssh 默认是 22 端口，现在如果用默认的 22 端口去连接，是没办法和 Gitlab 容器中的 22 端口进行映射的，因为我们只是通过 Service 的 22 端口进行了映射，要想通过节点去进行 ssh 链接就需要在节点上一个端口和容器内部的22端口进行绑定，所以这里我们可以通过 NodePort 去映射 Gitlab 容器内部的22端口，比如我们将环境变量设置为 `GITLAB_SSH_PORT=30022`，将 Gitlab 的 Service 也设置为 NodePort 类型：

```
apiVersion: v1
kind: Service
metadata:
  name: gitlab
  namespace: kube-ops
  labels:
    name: gitlab
spec:
  ports:
    - name: http
      port: 80
      targetPort: http
    - name: ssh
      port: 22
      targetPort: ssh
      nodePort: 30022
  type: NodePort
  selector:
```

```
name: gitlab
```

注意上面 ssh 对应的 nodePort 端口设置为 30022，这样就不会随机生成了，重新更新下 Deployment 和 Service，更新完成后，现在我们在项目上面 Clone 的时候使用 ssh 就会带上端口号了：

gitlab ssh

现在就可以使用 `clone with SSH` 的地址了，由于上面我们配置了 SSH 公钥，所以就可以直接访问上面的仓库了：

```
$ git clone ssh://git@git.qikqiak.com:30022/root/gitlab-demo.git
Cloning into 'gitlab-demo'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
```

然后随便在该项目下面添加一些资源：

```
$ echo "# hello world" > README.md
$ git add .
$ git commit -m 'hello world'
[master (root-commit) 63de7cb] hello world
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 224 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@git.qikqiak.com:30022/root/gitlab-demo.git
 * [new branch]      master -> master
```

然后刷新浏览器，就可以看到刚刚创建的 Git 仓库中多了一个 `README.md` 的文件：

gitlab-demo 🔒
项目ID: 1

添加许可证 1 Commit 1 Branch 0 Tag 41 KB Files

this repos is a gitlab demo project

master gitlab-demo / +

历史 查找文件 Web IDE ⌂

hello world
由 yangchuanhu 提交于 2 分钟前
63de7cb6 ⌂

README Add CHANGELOG Add CONTRIBUTING 启用Auto DevOps 添加 Kubernetes 集群

名称	最后提交	最后更新
README.md	hello world	2 分钟前

README.md

hello world

git commit

到这里就表明我们的 Gitlab 就成功部署到了 Kubernetes 集群当中了。

本文中涉及到的所有资源清单文件参考地址: <https://github.com/cnych/kubeapp/tree/master/gitlab>
下节课和大家开始介绍基于 Gitlab 的 CI/CD 实现。

[点击查看本文视频](#)

扫描下面的二维码(或微信搜索 k8s技术圈)关注我们的微信公众帐号, 在微信公众帐号中回复 加群 即可加入到我们的 kubernetes 讨论群里面共同学习。



k8s技术圈二维码

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-04-24 15:29:49

65. Gitlab CI

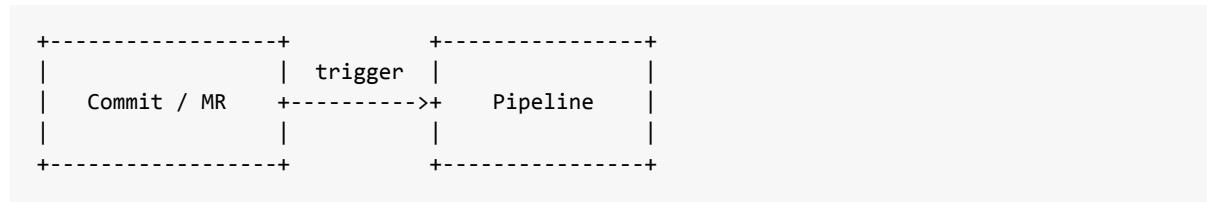
上节课我们使用 Helm 快速的将 Gitlab 安装到了我们的 Kubernetes 集群中，这节课来和大家介绍如何使用 Gitlab CI 来做持续集成。

简介

从 Gitlab 8.0 开始，Gitlab CI 就已经集成在 Gitlab 中，我们只要在项目中添加一个 `.gitlab-ci.yml` 文件，然后添加一个 `Runner`，即可进行持续集成。在介绍 Gitlab CI 之前，我们先看看一些 Gitlab CI 的一些相关概念。

Pipeline

一次 Pipeline 其实相当于一次构建任务，里面可以包含很多个流程，如安装依赖、运行测试、编译、部署测试服务器、部署生产服务器等流程。任何提交或者 Merge Request 的合并都可以触发 Pipeline 构建，如下图所示：

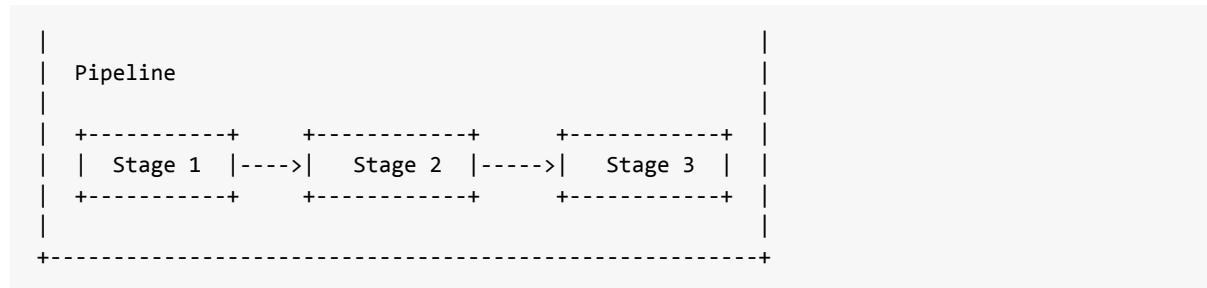


Stages

Stages 表示一个构建阶段，也就是上面提到的一个流程。我们可以在一次 Pipeline 中定义多个 Stages，这些 Stages 会有以下特点：

- 所有 Stages 会按照顺序运行，即当一个 Stage 完成后，下一个 Stage 才会开始
- 只有当所有 Stages 完成后，该构建任务 (Pipeline) 才会成功
- 如果任何一个 Stage 失败，那么后面的 Stages 不会执行，该构建任务 (Pipeline) 失败

Stages 和 Pipeline 的关系如下所示：

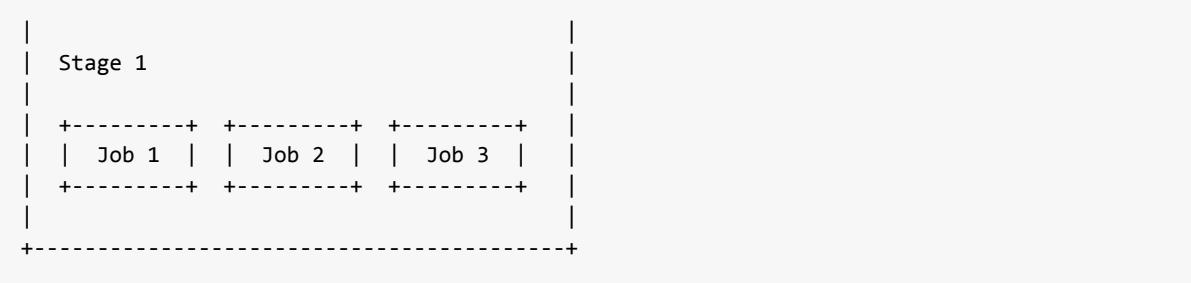


Jobs

Jobs 表示构建工作，表示某个 Stage 里面执行的工作。我们可以在 Stages 里面定义多个 Jobs，这些 Jobs 会有以下特点：

- 相同 Stage 中的 Jobs 会并行执行
- 相同 Stage 中的 Jobs 都执行成功时，该 Stage 才会成功
- 如果任何一个 Job 失败，那么该 Stage 失败，即该构建任务 (Pipeline) 失败

Jobs 和 Stage 的关系如下所示：



Gitlab Runner

如果理解了上面的基本概念之后，可能我们就会发现一个问题，我们的构建任务在什么地方来执行呢，以前用 Jenkins 在 Master 和 Slave 节点都可以用来运行构建任务，而来执行我们的 Gitlab CI 构建任务的就是 Gitlab Runner。

我们知道大多数情况下构建任务都是会占用大量的系统资源的，如果直接让 Gitlab 本身来运行构建任务的话，显然 Gitlab 的性能会大幅度下降的。GitLab CI 最大的作用是管理各个项目的构建状态，因此，运行构建任务这种浪费资源的事情交给一个独立的 Gitlab Runner 来做就会好很多，更重要的是 Gitlab Runner 可以安装到不同的机器上，甚至是我们本机，这样完全就不会影响到 Gitlab 本身了。

安装

安装 Gitlab Runner 非常简单，我们可以完全安装官方文档：<https://docs.gitlab.com/runner/install/> 即可，比如可以直接使用二进制、Docker 等来安装。同样的，我们这里还是将 Gitlab Runner 安装到 Kubernetes 集群中来，让我们的集群来统一管理 Gitlab 相关的服务。

1.验证 Kubernetes 集群

执行下面的命令验证 Kubernetes 集群：

```

$ kubectl cluster-info
Kubernetes master is running at https://10.151.30.11:6443
KubeDNS is running at https://10.151.30.11:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

```

`cluster-info` 这个命令会显示当前链接的集群状态和可用的集群服务列表。

2.获取 Gitlab CI Register Token

前面的章节中我们已经成功安装了 Gitlab，在浏览器中打开 `git.qikqiak.com` 页面，然后登录后进入到管理页面 `http://git.qikqiak.com/admin`，然后点击导航栏中的 `Runner`，可以看到该页面中有两个主要的参数，一个是 URL，另外一个就是 Register Token，下面的步骤中需要用到这两个参数值。

管理中心 > Runner

The screenshot shows the 'Runner' configuration page in the GitLab interface. On the left, there's a note about what a Runner is and its states (shared, group, specific, locked, paused). On the right, there's a 'manual setup shared Runner' section with steps: 1. Install GitLab Runner, 2. Set the Runner URL to `http://git.qikqiak.com/`, 3. Use the registration token `rcVZF-mdHt9qCyyrCDgS`, and 4. Start the Runner.

`gitlab runner`

注意：不要随便泄露 Token

3. 编写 Gitlab CI Runner 资源清单文件

同样我们将 Runner 相关的资源对象都安装到 `kube-ops` 这个 namespace 下面，首先，我们通过 ConfigMap 资源来传递 Runner 镜像所需的环境变量 (`runner-cm.yaml`)：

```
apiVersion: v1
data:
  REGISTER_NON_INTERACTIVE: "true"
  REGISTER_LOCKED: "false"
  METRICS_SERVER: "0.0.0.0:9100"
  CI_SERVER_URL: "http://git.qikqiak.com/ci"
  RUNNER_REQUEST_CONCURRENCY: "4"
  RUNNER_EXECUTOR: "kubernetes"
  KUBERNETES_NAMESPACE: "kube-ops"
  KUBERNETES_PRIVILEGED: "true"
  KUBERNETES_CPU_LIMIT: "1"
  KUBERNETES_CPU_REQUEST: "500m"
  KUBERNETES_MEMORY_LIMIT: "1Gi"
  KUBERNETES_SERVICE_CPU_LIMIT: "1"
  KUBERNETES_SERVICE_MEMORY_LIMIT: "1Gi"
  KUBERNETES_HELPER_CPU_LIMIT: "500m"
  KUBERNETES_HELPER_MEMORY_LIMIT: "100Mi"
  KUBERNETES_PULL_POLICY: "if-not-present"
  KUBERNETES_TERMINATIONGRACEPERIODSECONDS: "10"
  KUBERNETES_POLL_INTERVAL: "5"
  KUBERNETES_POLL_TIMEOUT: "360"
kind: ConfigMap
metadata:
  labels:
    app: gitlab-ci-runner
    name: gitlab-ci-runner-cm
    namespace: kube-ops
```

要注意 `CI_SERVER_URL` 对应的值需要指向我们的 Gitlab 实例的 URL，并加上 `/ci` (<http://git.qikqiak.com/ci>)。此外还添加了一些构建容器运行的资源限制，我们可以自己根据需要进行更改即可。

注意：在向 ConfigMap 添加新选项后，需要删除 GitLab CI Runner Pod。因为我们将使用 `envFrom` 来注入上面的这些环境变量而不是直接使用 `env` 的（`envFrom` 通过将环境变量放置到 `ConfigMaps` 或 `Secrets` 来帮助减小清单文件）。

另外如果要添加其他选项的话，我们可以在 Pod 中运行 `gitlab-ci-multi-runner register --help` 命令来查看所有可使用的选项，只需为要配置的标志添加 `env` 变量即可，如下所示：

```
gitlab-runner@gitlab-ci-runner-0:/ $ gitlab-ci-multi-runner register --help
[...]
--kubernetes-cpu-limit value                                     The CPU allocation given to build co
ntainers (default: "1") [${KUBERNETES_CPU_LIMIT}]
--kubernetes-memory-limit value                                 The amount of memory allocated to bu
ild containers (default: "4Gi") [${KUBERNETES_MEMORY_LIMIT}]
--kubernetes-service-cpu-limit value                           The CPU allocation given to build se
rvice containers (default: "1") [${KUBERNETES_SERVICE_CPU_LIMIT}]
--kubernetes-service-memory-limit value                      The amount of memory allocated to bu
ild service containers (default: "1Gi") [${KUBERNETES_SERVICE_MEMORY_LIMIT}]
--kubernetes-helper-cpu-limit value                         The CPU allocation given to build he
lper containers (default: "500m") [${KUBERNETES_HELPER_CPU_LIMIT}]
--kubernetes-helper-memory-limit value                     The amount of memory allocated to bu
ild helper containers (default: "3Gi") [${KUBERNETES_HELPER_MEMORY_LIMIT}]
--kubernetes-cpu-request value                            The CPU allocation requested for bui
ld containers [${KUBERNETES_CPU_REQUEST}]
[...]
```

除了上面的一些环境变量相关的配置外，还需要一个用于注册、运行和取消注册 Gitlab CI Runner 的小脚本。只有当 Pod 正常通过 Kubernetes (TERM信号) 终止时，才会触发转轮取消注册。如果强制终止 Pod (SIGKILL信号)，Runner 将不会注销自身。必须手动完成对这种被杀死的 Runner 的清理，配置清单文件如下：(runner-scripts-cm.yaml)

```
apiVersion: v1
data:
  run.sh: |
    #!/bin/bash
    unregister() {
      kill %1
      echo "Unregistering runner ${RUNNER_NAME} ..."
      /usr/bin/gitlab-ci-multi-runner unregister -t "$(./usr/bin/gitlab-ci-multi-runner lis
t 2>&1 | tail -n1 | awk '{print $4}' | cut -d'=' -f2)" -n ${RUNNER_NAME}
      exit $?
    }
    trap 'unregister' EXIT HUP INT QUIT PIPE TERM
    echo "Registering runner ${RUNNER_NAME} ..."
    /usr/bin/gitlab-ci-multi-runner register -r ${GITLAB_CI_TOKEN}
    sed -i 's/^concurrent.*$/concurrent = \'"${RUNNER_REQUEST_CONCURRENCY}"\' /home/gitlab-
runner/.gitlab-runner/config.toml
    echo "Starting runner ${RUNNER_NAME} ..."
    /usr/bin/gitlab-ci-multi-runner run -n ${RUNNER_NAME} &
    wait
kind: ConfigMap
metadata:
```

```

labels:
  app: gitlab-ci-runner
  name: gitlab-ci-runner-scripts
  namespace: kube-ops

```

我们可以看到需要一个 GITLAB_CI_TOKEN，然后我们用 Gitlab CI runner token 来创建一个 Kubernetes secret 对象。将 token 进行 base64 编码：

```
$ echo rcVZF-mdHt9qCyyrCDgS | base64 -w0
cmNWWkYtbWRIdDlxQ315ckNEZ1MK
```

base64 命令在大部分 Linux 发行版中都是可用的。

然后使用上面的 token 创建一个 Secret 对象：(gitlab-ci-token-secret.yaml)

```

apiVersion: v1
kind: Secret
metadata:
  name: gitlab-ci-token
  namespace: kube-ops
  labels:
    app: gitlab-ci-runner
data:
  GITLAB_CI_TOKEN: cmNWWkYtbWRIdDlxQ315ckNEZ1MK

```

然后接下来我们就可以来编写一个用于真正运行 Runner 的控制器对象，我们这里使用 Statefulset。首先，在开始运行的时候，尝试取消注册所有的同名 Runner，当节点丢失时（即 NodeLost 事件），这尤其有用。然后再尝试重新注册自己并开始运行。在正常停止 Pod 的时候，Runner 将会运行 unregister 命令来尝试取消自己，所以 Gitlab 就不能再使用这个 Runner 了，这个是通过 Kubernetes Pod 生命周期中的 hooks 来完成的。

另外我们通过使用 envFrom 来指定 Secrets 和 ConfigMaps 来用作环境变量，对应的资源清单文件如下：(runner-statefulset.yaml)

```

apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: gitlab-ci-runner
  namespace: kube-ops
  labels:
    app: gitlab-ci-runner
spec:
  updateStrategy:
    type: RollingUpdate
  replicas: 2
  serviceName: gitlab-ci-runner
  template:
    metadata:
      labels:
        app: gitlab-ci-runner
    spec:
      volumes:
        - name: gitlab-ci-runner-scripts

```

```

projected:
  sources:
    - configMap:
        name: gitlab-ci-runner-scripts
        items:
          - key: run.sh
            path: run.sh
            mode: 0755
  serviceAccountName: gitlab-ci
  securityContext:
    runAsNonRoot: true
    runAsUser: 999
    supplementalGroups: [999]
  containers:
    - image: gitlab/gitlab-runner:latest
      name: gitlab-ci-runner
      command:
        - /scripts/run.sh
      envFrom:
        - configMapRef:
            name: gitlab-ci-runner-cm
        - secretRef:
            name: gitlab-ci-token
      env:
        - name: RUNNER_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
      ports:
        - containerPort: 9100
          name: http-metrics
          protocol: TCP
      volumeMounts:
        - name: gitlab-ci-runner-scripts
          mountPath: "/scripts"
          readOnly: true
  restartPolicy: Always

```

可以看到上面我们使用了一个名为 gitlab-ci 的 serviceAccount，新建一个 rbac 资源清单文件：(runner-rbac.yaml)

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: gitlab-ci
  namespace: kube-ops
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: gitlab-ci
  namespace: kube-ops
rules:
  - apiGroups: []
    resources: ["*"]
    verbs: ["*"]
---
kind: RoleBinding

```

```

apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: gitlab-ci
  namespace: kube-ops
subjects:
- kind: ServiceAccount
  name: gitlab-ci
  namespace: kube-ops
roleRef:
  kind: Role
  name: gitlab-ci
  apiGroup: rbac.authorization.k8s.io

```

4. 创建 Runner 资源对象

资源清单文件准备好后，我们直接创建上面的资源对象：

```

$ ls
gitlab-ci-token-secret.yaml  runner-cm.yaml  runner-rbac.yaml  runner-scripts-cm.yaml  runner-statefulset.yaml
$ kubectl create -f .
secret "gitlab-ci-token" created
configmap "gitlab-ci-runner-cm" created
serviceaccount "gitlab-ci" created
role.rbac.authorization.k8s.io "gitlab-ci" created
rolebinding.rbac.authorization.k8s.io "gitlab-ci" created
configmap "gitlab-ci-runner-scripts" created
statefulset.apps "gitlab-ci-runner" created

```

创建完成后，可以通过查看 Pod 状态判断 Runner 是否运行成功：

```

$ kubectl get pods -n kube-ops
NAME                               READY   STATUS    RESTARTS   AGE
gitlab-7bff969fb-c-k5z14           1/1     Running   0          4d
gitlab-ci-runner-0                 1/1     Running   0          3m
gitlab-ci-runner-1                 1/1     Running   0          3m
.....

```

可以看到已经成功运行了两个（具体取决于 StatefulSet 清单中的副本数）Runner 实例，然后切换到 Gitlab Admin 页面下面的 Runner 页面：

Runner是一个执行任务的进程。您可以根据需要配置任意数量的Runner。Runner可以放在不同的用户、服务器，甚至本地机器上。

每个Runner可以处于以下状态中的其中一种：

- **shared** - Runner将运行所有未指定的项目的作业
- **group** - Runner将运行群组中所有未指定项目的作业
- **specific** - Runner将运行指定项目的作业
- **locked** - 无法将Runner分配给其他项目
- **paused** - Runner不会接受新的作业

手动设置shared Runner

1. 安装GitLab Runner
2. 在Runner设置时指定以下URL: <http://git.qikqiak.com/>
3. 在安装过程中使用以下注册令牌: `rcVZF-mdHt9qCyyrCDgS`
4. 启动Runner!

[重置 Runner 注册令牌](#)

类型	Runner 令牌	描述	版本	IP地址	项目	作业	标签	最后联系	
shared	Tt7PR3ss	gitlab-ci-runner-1	11.8.0	123.59.188...	不适用	0		7 小时后	编辑 禁用 删除
shared	v2UZ_22H	gitlab-ci-runner-0	11.8.0	123.59.188...	不适用	0		7 小时后	编辑 禁用 删除

gitlab runner list

当然我们也可以根据需要更改 Runner 的一些配置，比如添加 tag 标签等。

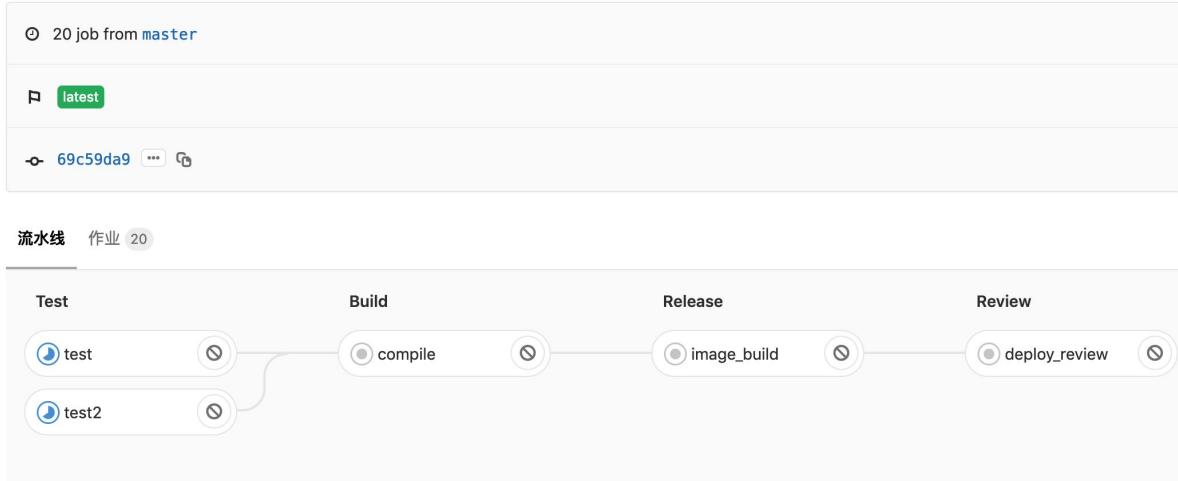
Gitlab CI

基本配置

接下来使用 Gitlab CI 所用到的代码库可以从 Github 上获得：[cnych/presentation-gitlab-k8s](https://github.com/cnych/presentation-gitlab-k8s)，可以在 Gitlab 上新建一个项目导入该仓库，当然也可以新建一个空白的仓库，然后将 Github 上面的项目 Clone 到本地后，更改远程仓库地址即可：

```
$ git clone https://github.com/cnych/presentation-gitlab-k8s.git
$ cd presentation-gitlab-k8s
# Change the remote of the repository
$ git remote set-url origin ssh://git@git.qikqiak.com:30022/root/presentation-gitlab-k8s.git
# Now to push/"import" the repository run:
$ git push -u origin master
```

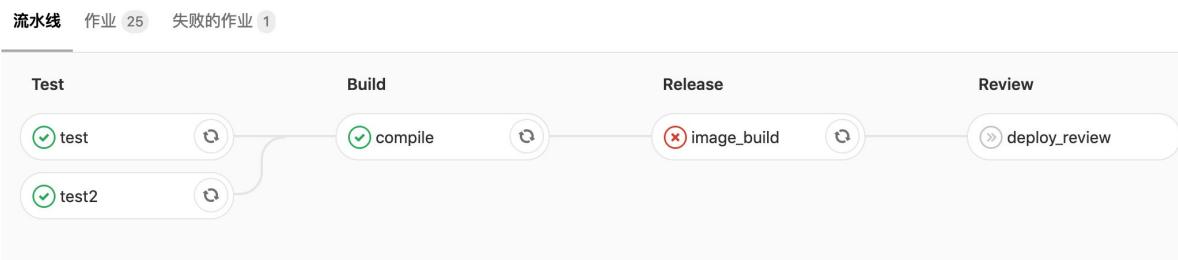
当我们把仓库推送到 Gitlab 以后，应该可以看到 Gitlab CI 开始执行构建任务了：



此时 Runner Pod 所在的 namespace 下面也会出现两个新的 Pod：

```
$ kubectl get pods -n kube-ops
NAME                               READY   STATUS    RESTARTS   AGE
gitlab-7bff969fbc-k5z14           1/1     Running   0          4d
gitlab-ci-runner-0                1/1     Running   0          4m
gitlab-ci-runner-1                1/1     Running   0          4m
runner-9rixsyft-project-2-concurrent-06g5w4   0/2     ContainerCreating   0          4m
runner-9rixsyft-project-2-concurrent-1t74t9   0/2     ContainerCreating   0          4m
...
```

这两个新的 Pod 就是用来执行具体的 Job 任务的，这里同时出现两个证明第一步是并行执行的两个任务，从上面的 Pipeline 中也可以看到是 test 和 test2 这两个 Job。我们可以看到在执行 image_build 任务的时候出现了错误：



pipeline

我们可以点击查看这个 Job 失败详细信息：

```
$ docker login -u "${CI_REGISTRY_USER}" -p "${CI_REGISTRY_PASSWORD}" "${CI_REGISTRY}"
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
Error response from daemon: Get https://registry-1.docker.io/v2/: unauthorized: incorrect
username or password
ERROR: Job failed: command terminated with exit code 1
```

出现上面的错误是因为我们并没有在 Gitlab 中开启 Container Registry，所以环境变量中并没有这些值，还记得前面章节中我们安装的 Harbor 吗？我们这里使用 Harbor 来作为我们的镜像仓库，这里我们需要把 Harbor 相关的配置以参数的形式配置到环境中就可以了。

定位到项目 -> 设置 -> CI/CD，展开 `Environment variables` 栏目，配置镜像仓库相关的参数值：

Environment variables

Environment variables are applied to environments via the runner. They can be protected by only exposing them to protected branches or tags. You can use environment variables for passwords, secret keys, or whatever you want. You may also add variables that are made available to the running application by prepending the variable key with `K8S_SECRET_`. [更多信息](#)

CI_REGISTRY	*****	受保护 <input checked="" type="checkbox"/>
CI_REGISTRY_IMAGE	*****	受保护 <input checked="" type="checkbox"/>
CI_REGISTRY_PASSWORD	*****	受保护 <input checked="" type="checkbox"/>
CI_REGISTRY_USER	*****	受保护 <input checked="" type="checkbox"/>
输入变量的名称	输入变量的值	受保护 <input checked="" type="checkbox"/>

保存变量 **显示值**

`gitlab ci env`

配置上后，我们在上面失败的 Job 任务上点击“重试”，在重试过后依然可以看到会出现下面的错误信息：

```
$ docker login -u "${CI_REGISTRY_USER}" -p "${CI_REGISTRY_PASSWORD}" "${CI_REGISTRY}"
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
Error response from daemon: Get https://registry.qikqiak.com/v2/: x509: certificate signed
by unknown authority
ERROR: Job failed: command terminated with exit code 1
```

从错误信息可以看出这是因为登录私有镜像仓库的时候证书验证错误，因为我们根本就没有提供任何证书，所以肯定会失败的，还记得我们之前在介绍 Harbor 的时候的解决方法吗？第一种是在 Docker 的启动参数中添加上 `insecure-registries`，另外一种是在目录 `/etc/docker/certs.d/` 下面添加上私有仓库的 CA 证书，同样，我们只需要在 `dind` 中添加 `insecure` 的参数即可：

```
services:
- name: docker:dind
  command: [--insecure-registry=registry.qikqiak.com]
```

其中 `registry.qikqiak.com` 就是我们之前配置的私有镜像仓库地址。

然后保存 `.gitlab-ci.yml` 文件，重新提交到代码仓库，可以看到又触发了正常的流水线构建了，在最后的阶段 `deploy_review` 仍然可以看到失败了，这是因为在最后的部署阶段我们使用 `kubectl` 工具操作集群的时候并没有关联上任何集群。

我们在 Gitlab CI 中部署阶段使用到的镜像是 `cnych/kubectl`，该镜像的 `Dockerfile` 文件可以在仓库 <https://github.com/cnych/docker-kubectl> 中获取：

```
FROM alpine:3.8

MAINTAINER cnych <icnycz@gmail.com>

ENV KUBE_LATEST_VERSION="v1.13.4"

RUN apk add --update ca-certificates \
&& apk add --update -t deps curl \
&& apk add --update gettext \
&& apk add --update git \
&& curl -L https://storage.googleapis.com/kubernetes-release/release/${KUBE_LATEST_VERSION} \
/bin/linux/amd64/kubectl -o /usr/local/bin/kubectl \
&& chmod +x /usr/local/bin/kubectl \
&& apk del --purge deps \
&& rm /var/cache/apk/*

ENTRYPOINT ["kubectl"]
CMD ["--help"]
```

我们知道 `kubectl` 在使用的时候默认会读取当前用户目录下面的 `~/.kube/config` 文件来链接集群，当然我们可以把连接集群的信息直接内置到上面的这个镜像中去，这样就可以直接操作集群了，但是也有一个不好的地方就是不方便操作，假如要切换一个集群还得重新制作一个镜像。所以一般我们这里直接在 Gitlab 上配置集成 Kubernetes 集群。

在项目页面点击 `Add Kubernetes Cluster -> Add existing cluster`：

1.Kubernetes cluster name 可以随便填

2.API URL 是你的集群的 `apiserver` 的地址，一般可以通过输入 `kubectl cluster-info` 获取，`Kubernetes master` 地址就是需要的

```
$ kubectl cluster-info
Kubernetes master is running at https://10.151.30.11:6443
KubeDNS is running at https://10.151.30.11:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

3.CA证书、Token、项目命名空间

对于我们这个项目准备部署在一个名为 `gitlab` 的 namespace 下面，所以首先我们需要到目标集群中创建一个 namespace：

```
$ kubectl create ns gitlab
```

由于我们在部署阶段需要去创建、删除一些资源对象，所以我们也需要对象的 RBAC 权限，这里为了简单，我们直接新建一个 ServiceAccount，绑定上一个 `cluster-admin` 的权限：(`gitlab-sa.yaml`)

```

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: gitlab
  namespace: gitlab

---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: gitlab
  namespace: gitlab
subjects:
- kind: ServiceAccount
  name: gitlab
  namespace: gitlab
roleRef:
  kind: ClusterRole
  name: cluster-admin

```

然后创建上面的 ServiceAccount 对象：

```
$ kubectl apply -f sa.yaml
serviceaccount "gitlab" created
clusterrolebinding.rbac.authorization.k8s.io "gitlab" created
```

可以通过上面创建的 ServiceAccount 获取 CA 证书和 Token：

```

$ kubectl get serviceaccount gitlab -n gitlab -o json | jq -r '.secrets[0].name'
gitlab-token-f9zp7

# 然后根据上面的Secret找到CA证书
$ kubectl get secret gitlab-token-f9zp7 -n gitlab -o json | jq -r '.data["ca.crt"]' | base64 -d
xxxxxxCA证书内容xxxxx

# 当然要找到对应的 Token 也很简单
$ kubectl get secret gitlab-token-f9zp7 -n gitlab -o json | jq -r '.data.token' | base64 -d
xxxxxxxxtoken值xxxx

```

填写上面对应的值添加集群：

输入Kubernetes集群的详细信息

请输入Kubernetes集群的访问信息。如需帮助，可以阅读Kubernetes集群的 [文档](#)

Kubernetes 集群名称

ydzs-k8s-cluster

API地址

<https://10.151.30.11:6443>

CA证书

-----BEGIN CERTIFICATE-----

MII CyDCCAbCgAwIBAgIBADANBgkqhkiG9w0BAQsFADAVMRMwEQYDVQQDEwprdWJl

令牌

eyJhbGciOiJSUzI1NjlsImtpZCI6Ij9.eyJpc3MiOiJrdWJlcmt5dGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRIcI

项目命名空间(可选, 唯一)

gitlab

启用RBAC的群集

如果使用基于角色的访问控制 (RBAC)，请启用此设置。此选项允许您在启用RBAC的群集上安装应用程序。 [更多信息](#)

[添加 Kubernetes 集群](#)

add k8s cluster

.gitlab-ci.yml

现在 Gitlab CI 的环境都准备好了，我们可以来看下用于描述 Gitlab CI 的 `.gitlab-ci.yml` 文件。

一个 Job 在 `.gitlab-ci.yml` 文件中一般如下定义：

```
# 运行golang测试用例
test:
  stage: test
  script:
    - go test ./...
```

上面这个 Job 会在 test 这个 Stage 阶段运行。

为了指定运行的 Stage 阶段，可以在 `.gitlab-ci.yml` 文件中放置任意一个简单的列表：

```
# 所有 Stage
stages:
  - test
  - build
```

- release
- deploy

你可以指定用于在全局或者每个作业上执行命令的镜像：

```
# 对于未指定镜像的作业，会使用下面的镜像
image: golang:1.10.3-stretch
# 或者对于特定的job使用指定的镜像
test:
  stage: test
  image: python:3
  script:
    - echo Something in the test step in a python:3 image
```

对于 `.gitlab-ci.yml` 文件的其他部分，请查看如下文档介绍：

<https://docs.gitlab.com/ce/ci/yaml/README.html>。

在我们当前的项目中定义了 4 个构建阶段：test、build、release、review、deploy，完整的 `.gitlab-ci.yml` 文件如下：

```
image:
  name: golang:1.10.3-stretch
  entrypoint: ["/bin/sh", "-c"]

# 为了能够使用go get，需要将代码放在 $GOPATH 中，比如你的 gitlab 域名是 mydomain.com，你的代码仓库是 repos/projectname，默认的 GOPATH 是 /go，然后你就需要将你的代码放置到 GOPATH 下面，/go/src /mydomain.com/repos/projectname，用一个软链接指过来就可以了
before_script:
  - mkdir -p "/go/src/git.qikqiak.com/${CI_PROJECT_NAMESPACE}"
  - ln -sf "${CI_PROJECT_DIR}" "/go/src/git.qikqiak.com/${CI_PROJECT_PATH}"
  - cd "/go/src/git.qikqiak.com/${CI_PROJECT_PATH}/"

stages:
  - test
  - build
  - release
  - review
  - deploy

test:
  stage: test
  script:
    - make test

test2:
  stage: test
  script:
    - sleep 3
    - echo "We did it! Something else runs in parallel!"

compile:
  stage: build
  script:
    # 添加所有的依赖，或者使用 glide/govendor/...
    - make build
artifacts:
```

```

paths:
  - app

image_build:
  stage: release
  image: docker:latest
  variables:
    DOCKER_DRIVER: overlay
    DOCKER_HOST: tcp://localhost:2375
  services:
    - name: docker:17.03-dind
      command: ["--insecure-registry=registry.qikqiak.com"]
  script:
    - docker info
    - docker login -u "${CI_REGISTRY_USER}" -p "${CI_REGISTRY_PASSWORD}" registry.qikqiak.com
    - docker build -t "${CI_REGISTRY_IMAGE}:latest" .
    - docker tag "${CI_REGISTRY_IMAGE}:latest" "${CI_REGISTRY_IMAGE}:${CI_COMMIT_REF_NAME}"
    - test ! -z "${CI_COMMIT_TAG}" && docker push "${CI_REGISTRY_IMAGE}:latest"
    - docker push "${CI_REGISTRY_IMAGE}:${CI_COMMIT_REF_NAME}"

deploy_review:
  image: cnych/kubectl
  stage: review
  only:
    - branches
  except:
    - tags
  environment:
    name: dev
    url: https://dev-gitlab-k8s-demo.qikqiak.com
    on_stop: stop_review
  script:
    - kubectl version
    - cd manifests/
    - sed -i "s/_CI_ENVIRONMENT_SLUG_/${CI_ENVIRONMENT_SLUG}/" deployment.yaml ingress.yaml service.yaml
    - sed -i "s/_VERSION_/${CI_COMMIT_REF_NAME}/" deployment.yaml ingress.yaml service.yaml
    - |
      if kubectl apply -f deployment.yaml | grep -q unchanged; then
        echo "> Patching deployment to force image update."
        kubectl patch -f deployment.yaml -p "{\"spec\":{\"template\":{\"metadata\":{\"annotations\":{\"ci-last-updated\":$(date +'%s')\"}}}}}"
      else
        echo "> Deployment apply has changed the object, no need to force image update.
      "
    fi
    - kubectl apply -f service.yaml || true
    - kubectl apply -f ingress.yaml
    - kubectl rollout status -f deployment.yaml
    - kubectl get all,ing -l ref=${CI_ENVIRONMENT_SLUG}

stop_review:
  image: cnych/kubectl
  stage: review
  variables:
    GIT_STRATEGY: none
  when: manual

```

```

only:
  - branches
except:
  - master
  - tags
environment:
  name: dev
  action: stop
script:
  - kubectl version
  - kubectl delete ing -l ref=${CI_ENVIRONMENT_SLUG}
  - kubectl delete all -l ref=${CI_ENVIRONMENT_SLUG}

deploy_live:
  image: cnych/kubectl
  stage: deploy
  environment:
    name: live
    url: https://live-gitlab-k8s-demo.qikqiak.com
  only:
    - tags
  when: manual
  script:
    - kubectl version
    - cd manifests/
    - sed -i "s/_CI_ENVIRONMENT_SLUG_/${CI_ENVIRONMENT_SLUG}/" deployment.yaml ingress.yaml service.yaml
    - sed -i "s/_VERSION_/${CI_COMMIT_REF_NAME}/" deployment.yaml ingress.yaml service.yaml
    - kubectl apply -f deployment.yaml
    - kubectl apply -f service.yaml
    - kubectl apply -f ingress.yaml
    - kubectl rollout status -f deployment.yaml
    - kubectl get all,ing -l ref=${CI_ENVIRONMENT_SLUG}

```

上面的 `.gitlab-ci.yml` 文件中还有一些特殊的属性，如限制运行的的 `when` 和 `only` 参数，例如 `only: ["tags"]` 表示只为创建的标签运行，更多的信息，我可以通过查看 Gitlab CI YAML 文件查看：<https://docs.gitlab.com/ce/ci/yaml/README.html>

由于我们在 `.gitlab-ci.yml` 文件中将应用的镜像构建完成后推送到我们的私有仓库，而 Kubernetes 资源清单文件中使用的私有镜像，所以我们需要配置一个 `imagePullSecret`，否则在 Kubernetes 集群中是无法拉取我们的私有镜像的：(替换下面相关信息为自己的)

```
$ kubectl create secret docker-registry myregistry --docker-server=registry.qikqiak.com --
  docker-username=xxxx --docker-password=xxxxxx --docker-email=xxxx -n gitlab
secret "myregistry" created
```

在下面的 Deployment 的资源清单文件中会使用到创建的 `myregistry`。

接下来为应用创建 Kubernetes 资源清单文件，添加到代码仓库中。首先创建 Deployment 资源：
(`deployment.yaml`)

```

---
apiVersion: apps/v1
kind: Deployment
```

```

metadata:
  name: gitlab-k8s-demo-__CI_ENVIRONMENT_SLUG__
  namespace: gitlab
  labels:
    app: gitlab-k8s-demo
    ref: __CI_ENVIRONMENT_SLUG__
    track: stable
spec:
  replicas: 2
  selector:
    matchLabels:
      app: gitlab-k8s-demo
      ref: __CI_ENVIRONMENT_SLUG__
  template:
    metadata:
      labels:
        app: gitlab-k8s-demo
        ref: __CI_ENVIRONMENT_SLUG__
        track: stable
    spec:
      imagePullSecrets:
        - name: myregistry
      containers:
        - name: app
          image: registry.qikqiak.com/gitdemo/gitlab-k8s:__VERSION__
          imagePullPolicy: Always
          ports:
            - name: http-metrics
              protocol: TCP
              containerPort: 8000
          livenessProbe:
            httpGet:
              path: /health
              port: 8000
              initialDelaySeconds: 3
              timeoutSeconds: 2
          readinessProbe:
            httpGet:
              path: /health
              port: 8000
              initialDelaySeconds: 3
              timeoutSeconds: 2

```

注意用上面创建的 myregistry 替换 imagePullSecrets。

这是一个基本的 Deployment 资源清单的描述，像 __CI_ENVIRONMENT_SLUG__ 和 __VERSION__ 这样的占位符用于区分不同的环境，__CI_ENVIRONMENT_SLUG__ 将由 dev 或 live (环境名称) 和 __VERSION__ 替换为镜像标签。

为了能够连接到部署的 Pod，还需要 Service。对应的 Service 资源清单如下 (service.yaml)：

```

---
apiVersion: v1
kind: Service
metadata:
  name: gitlab-k8s-demo-__CI_ENVIRONMENT_SLUG__
  namespace: gitlab
  labels:

```

```

app: gitlab-k8s-demo
ref: __CI_ENVIRONMENT_SLUG__
annotations:
  prometheus.io/scrape: "true"
  prometheus.io/port: "8000"
  prometheus.io/scheme: "http"
  prometheus.io/path: "/metrics"
spec:
  type: ClusterIP
  ports:
    - name: http-metrics
      port: 8000
      protocol: TCP
  selector:
    app: gitlab-k8s-demo
    ref: __CI_ENVIRONMENT_SLUG__

```

我们的应用程序运行8000端口上，端口名为 `http-metrics`，如果你还记得前面我们监控的课程中应该还记得我们使用 `prometheus-operator` 为 Prometheus 创建了自动发现的配置，所以我们 在 `annotations` 里面配置上面的这几个注释后，Prometheus 就可以自动获取我们应用的监控指标数据了。

现在 Service 创建成功了，但是外部用户还不能访问到我们的应用，当然我们可以把 Service 设置成 NodePort 类型，另外一个常见的方式当然就是使用 Ingress 了，我们可以通过 Ingress 来将应用暴露给外面用户使用，对应的资源清单文件如下： (`ingress.yaml`)

```

---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: gitlab-k8s-demo-__CI_ENVIRONMENT_SLUG__
  namespace: gitlab
  labels:
    app: gitlab-k8s-demo
    ref: __CI_ENVIRONMENT_SLUG__
  annotations:
    kubernetes.io/ingress.class: "traefik"
spec:
  rules:
    - host: __CI_ENVIRONMENT_SLUG__.gitlab-k8s-demo.qikqiak.com
      http:
        paths:
          - path: /
            backend:
              serviceName: gitlab-k8s-demo-__CI_ENVIRONMENT_SLUG__
              servicePort: 8000

```

当然如果想配置 https 访问的话我们可以自己用 CA 证书创建一个 `tls` 密钥，也可以使用 `cert-manager` 来自动为我们的应用程序添加 https。

当然要通过上面的域名进行访问，还需要进行 DNS 解析的，`__CI_ENVIRONMENT_SLUG__.gitlab-k8s-demo.qikqiak.com` 其中 `__CI_ENVIRONMENT_SLUG__` 值为 live 或 dev，所以需要创建 `dev-gitlab-k8s-demo.qikqiak.com` 和 `live-gitlab-k8s-demo.qikqiak.com` 两个域名的解析。

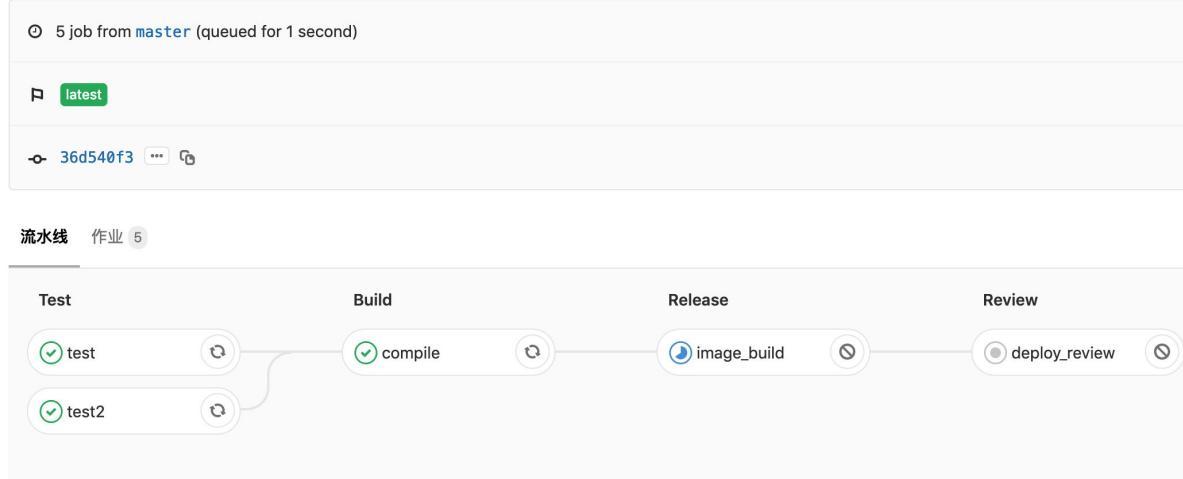
我们可以使用 DNS 解析服务商的 API 来自动创建域名解析，也可以使用 [Kubernetes incubator](#) 孵化的项目 [external-dns operator](#) 来进行操作。

所需要的资源清单和 `.gitlab-ci.yml` 文件已经准备好了，我们可以小小的添加一个文件去触发下 Gitlab CI 构建：

```
$ touch test1
$ git add .
$ git commit -m "Testing the GitLab CI functionality #1"
$ git push origin master
```

现在回到 Gitlab 中可以看到我们的项目触发了一个新的 Pipeline 的构建：

Testing the GitLab CI functionality #1



gitlab pipeline

可以查看最后一个阶段 (stage) 是否正确，如果通过了，证明我们已经成功将应用程序部署到 Kubernetes 集群中了，一个成功的 `review` 阶段如下所示：

```

GitCommit:"c27b913fddd1a6c480c229191a087698aa92f0b1", GitTreeState:"clean", BuildDate:"2019-02-28T13:37:52Z",
GoVersion:"go1.11.5", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.0",
GitCommit:"fc32d2f3698e36b93322a3465f63a14e9f0eaead", GitTreeState:"clean", BuildDate:"2018-03-26T16:44:10Z",
GoVersion:"go1.9.3", Compiler:"gc", Platform:"linux/amd64"}
$ cd manifests/
$ sed -i "s/_CI_ENVIRONMENT_SLUG_/${CI_ENVIRONMENT_SLUG}/" deployment.yaml ingress.yaml service.yaml
$ sed -i "s/_VERSION_/${CI_COMMIT_REF_NAME}/" deployment.yaml ingress.yaml service.yaml
$ if kubectl apply -f deployment.yaml | grep -q unchanged; then # collapsed multi-line command
=> Deployment apply has changed the object, no need to force image update.
$ kubectl apply -f service.yaml || true
service/gitlab-k8s-demo-dev created
$ kubectl apply -f ingress.yaml
ingress.extensions/gitlab-k8s-demo-dev created
$ kubectl rollout status -f deployment.yaml
Waiting for deployment "gitlab-k8s-demo-dev" rollout to finish: 0 of 2 updated replicas are available...
Waiting for deployment "gitlab-k8s-demo-dev" rollout to finish: 1 of 2 updated replicas are available...
deployment "gitlab-k8s-demo-dev" successfully rolled out
$ kubectl get all,ing -l ref=${CI_ENVIRONMENT_SLUG}
NAME                               READY   STATUS    RESTARTS   AGE
pod/gitlab-k8s-demo-dev-5b68ff8c9b-d4plv  1/1    Running   0          27s
pod/gitlab-k8s-demo-dev-5b68ff8c9b-fnxkd  1/1    Running   0          27s
NAME              TYPE      CLUSTER-IP        EXTERNAL-IP      PORT(S)        AGE
service/gitlab-k8s-demo-dev   ClusterIP   10.98.171.179   <none>           8000/TCP     26s
NAME             DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
deployment.apps/gitlab-k8s-demo-dev  2         2         2            2          27s
NAME             DESIRED  CURRENT  READY     AGE
replicaset.apps/gitlab-k8s-demo-dev-5b68ff8c9b  2         2         2          27s
NAME           HOSTS          ADDRESS        PORTS        AGE
ingress.extensions/gitlab-k8s-demo-dev   dev-gitlab-k8s-demo.qikqiak.com       80          26s

```

Job succeeded

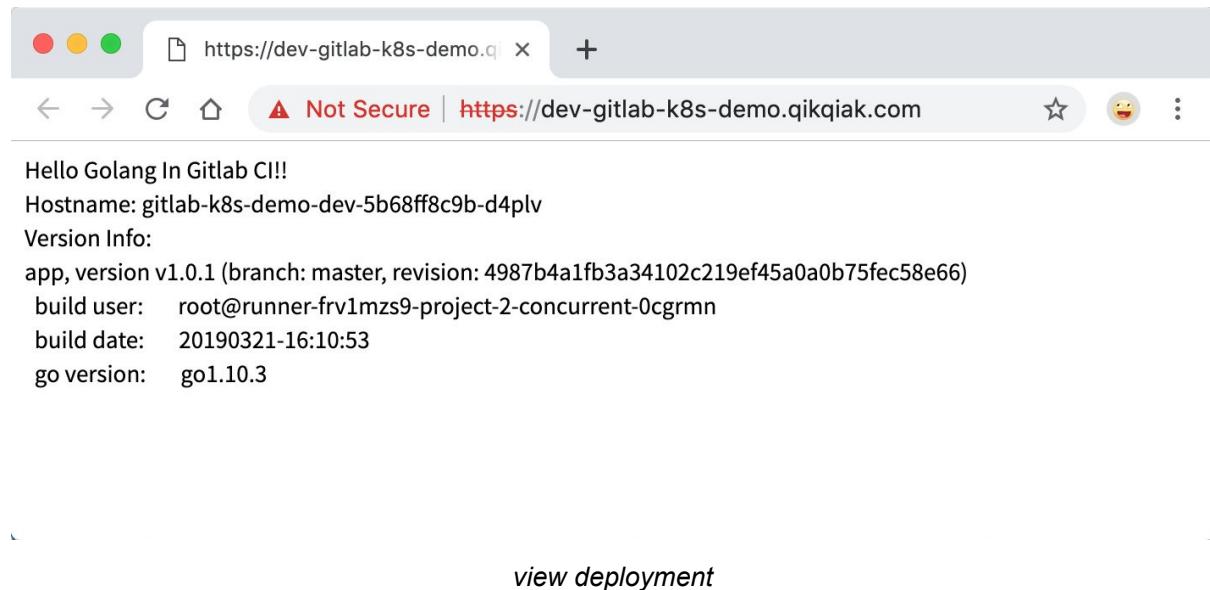
review success

整个 Pipeline 构建成功后，我们可以在项目的环境菜单下面看到多了一个环境：

dev							
ID	提交	作业	状态				
#33	master -> 4987b4a1 [update] 修改 gitlab-ci 脚本	deploy_review (#181) by	已创建 7 小时后				

env

如果我们点击 终止，就会调用 .gitlab-ci.yml 中定义的钩子 on_stop: stop_review，点击 View deployment 就可以看到这次我们的部署结果（前提是DNS解析已经完成）：



这就是关于 Gitlab CI 结合 Kubernetes 进行 CI/CD 的过程，具体详细的构建任务还需要结合我们自己的应用实际情况而定。下节课给大家介绍使用 Jenkins + Gitlab + Harbor + Helm + Kubernetes 来实现一个完整的 CI/CD 流水线作业。

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-04-24 15:29:53

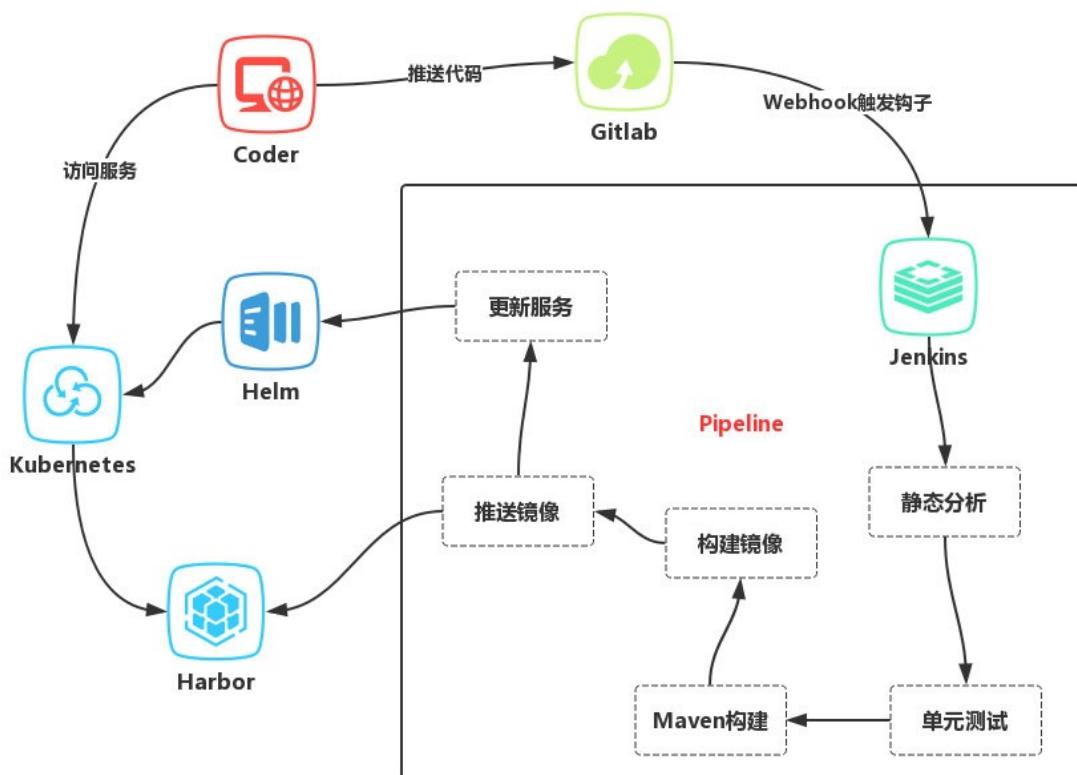
66. Devops

上节课和大家介绍了 Gitlab CI 结合 Kubernetes 进行 CI/CD 的完整过程。这节课结合前面所学的知识点给大家介绍一个完整的示例：使用 Jenkins + Gitlab + Harbor + Helm + Kubernetes 来实现一个完整的 CI/CD 流水线作业。

其实前面的课程中我们就已经学习了 Jenkins Pipeline 与 Kubernetes 的完美结合，我们利用 Kubernetes 来动态运行 Jenkins 的 Slave 节点，可以很好地解决传统的 Jenkins Slave 浪费大量资源的缺点。之前的示例中我们是将项目放置在 Github 仓库上的，将 Docker 镜像推送到了 Docker Hub，这节课我们来结合我们前面学习的知识点来综合运用下，使用 Jenkins、Gitlab、Harbor、Helm、Kubernetes 来实现一个完整的持续集成和持续部署的流水线作业。

流程

下图是我们当前示例的流程图



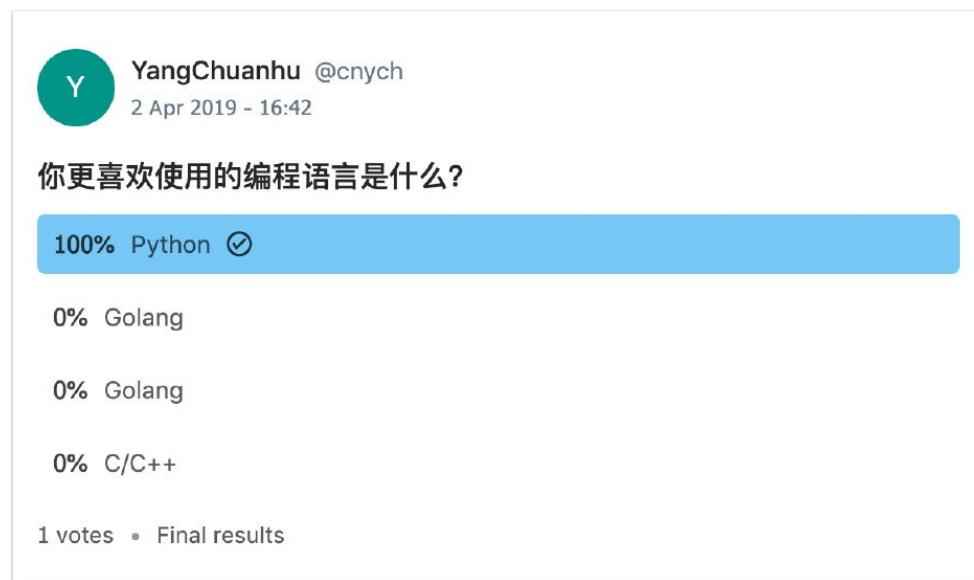
ci/cd demo

1. 开发人员提交代码到 Gitlab 代码仓库
- 1. 通过 Gitlab 配置的 Jenkins Webhook 触发 Pipeline 自动构建
- 1. Jenkins 触发构建构建任务，根据 Pipeline 脚本定义分步骤构建
- 1. 先进行代码静态分析，单元测试
- 1. 然后进行 Maven 构建（Java 项目）
- 1. 根据构建结果构建 Docker 镜像
- 1. 推送 Docker 镜像到 Harbor 仓库
- 1. 触发更新服务阶段，使用 Helm 安装/更新 Release
- 1. 查看服务是否更新成功。

项目

本次示例项目是一个完整的基于 Spring Boot、Spring Security、JWT、React 和 Ant Design 构建的一个开源的投票应用，项目地址：<https://github.com/callicoder/spring-security-react-ant-design-polls-app>。

Polling App



polling app1

我们将会在该项目的基础上添加部分代码，并实践 CI/CD 流程。

服务端

首先需要更改的是服务端配置，我们需要将数据库链接的配置改成环境变量的形式，写死了的话就没办法进行定制了，修改服务端文件 `src/main/resources/application.properties`，将下面的数据库配置部分修改成如下形式：

```
spring.datasource.url= jdbc:mysql://${DB_HOST:localhost}:${DB_PORT:3306}/${DB_NAME:polling_app}?useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=false
spring.datasource.username= ${DB_USER:root}
spring.datasource.password= ${DB_PASSWORD:root}
```

当环境变量中有上面的数据配置的时候，就会优先使用环境变量中的值，没有的时候就会用默认的值进行数据库配置。

由于我们要将项目部署到 Kubernetes 集群中去，所以我们需要将服务端进行容器化，所以我们在项目根目录下面添加一个 `Dockerfile` 文件进行镜像构建：

```
FROM openjdk:8-jdk-alpine

MAINTAINER cnych <cicnycy@gmail.com>

ENV LANG en_US.UTF-8
ENV LANGUAGE en_US:en
ENV LC_ALL en_US.UTF-8
ENV TZ=Asia/Shanghai

RUN mkdir /app

WORKDIR /app

COPY target/polls-0.0.1-SNAPSHOT.jar /app/polls.jar

EXPOSE 8080

ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app/polls.jar"]
```

由于服务端代码是基于 Spring Boot 构建的，所以我们这里使用一个 `openjdk` 的基础镜像，将打包过后的 `jar` 包放入镜像之中，然后用过 `java -jar` 命令直接启动即可，这里就会存在一个问题了，我们是在 Jenkins 的 Pipeline 中去进行镜像构建的，这个时候项目中并没有打包好的 `jar` 包文件，那么我们应该如何获取打包好的 `jar` 包文件呢？这里我们可以使用两种方法：

第一种就是如果你用于镜像打包的 Docker 版本大于 17.06 版本的话，那么我墙裂推荐你使用 Docker 的多阶段构建功能来完成镜像的打包过程，我们只需要将上面的 `Dockerfile` 文件稍微更改下即可，将使用 `maven` 进行构建的工作放到同一个文件中：

```
FROM maven:3.6-alpine as BUILD

COPY src /usr/app/src
COPY pom.xml /usr/app

RUN mvn -f /usr/app/pom.xml clean package -Dmaven.test.skip=true

FROM openjdk:8-jdk-alpine

MAINTAINER cnych <cicnycy@gmail.com>

ENV LANG en_US.UTF-8
ENV LANGUAGE en_US:en
ENV LC_ALL en_US.UTF-8
ENV TZ=Asia/Shanghai

RUN mkdir /app

WORKDIR /app

COPY --from=BUILD /usr/app/target/polls-0.0.1-SNAPSHOT.jar /app/polls.jar
```

```
EXPOSE 8080

ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "/app/polls.jar"]
```

前面课程中我们就讲解过 Docker 的多阶段构建，这里我们定义了两个阶段，第一个阶段利用 `maven:3.6-alpine` 这个基础镜像将我们的项目进行打包，然后将该阶段打包生成的 `jar` 包文件复制到第二阶段进行最后的镜像打包，这样就可以很好的完成我们的 Docker 镜像的构建工作。

第二种方式就是我们传统的方式，在 Jenkins Pipeline 中添加一个 `maven` 构建的阶段，然后在第二个 Docker 构建的阶段就可以直接获取到前面的 `jar` 包了，也可以很方便的完成镜像的构建工作，为了更加清楚的说明 Jenkins Pipeline 的用法，我们这里采用这种方式，所以 Dockerfile 文件还是使用第一个就行。

现在我们可以将服务端的代码推送到 Gitlab 上去，我们这里的仓库地址为：<http://git.qikqiak.com/course/polling-app-server.git>

注意，这里我们只推送的服务端代码。

客户端

客户端我们需要修改 API 的链接地址，修改文件 `src/constants/index.js` 中 `API_BASE_URL` 的地址，我们同样通过环境变量来进行区分，如果有环境变量 `APISERVER_URL`，则优先使用这个环境变量来作为 API 请求的地址：

```
let API_URL = 'http://localhost:8080/api';
if (process.env.APISERVER_URL) {
    API_URL = `${process.env.APISERVER_URL}/api`;
}
export const API_BASE_URL = API_URL;
```

因为我们这里的项目使用的就是前后端分离的架构，所以我们同样需要将前端代码进行单独的部署，同样我们要将项目部署到 Kubernetes 环境中，所以也需要做容器化，同样在项目根目录下面添加一个 `Dockerfile` 文件：

```
FROM nginx:1.15.10-alpine
ADD build /usr/share/nginx/html

ADD nginx.conf
/etc/nginx/conf.d/default.conf
```

由于前端页面是单纯的静态页面，所以一般我们使用一个 `nginx` 镜像来运行，所以我们提供一个 `nginx.conf` 配置文件：

```
server {
    gzip on;

    listen      80;
    server_name localhost;
```

```

root /usr/share/nginx/html;
location / {
    try_files $uri /index.html;
    expires 1h;
}

error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root /usr/share/nginx/html;
}

}

```

这里我们可以看到我们需要将前面页面打包到一个 `build` 目录，然后将改目录添加到 nginx 镜像中的 `/usr/share/nginx/html` 目录，这样当 nginx 镜像启动的时候就是直接使用的改文件夹下面的文件。

所以现在我们需要获取打包后的目录 `build`，同样的，和上面服务端项目一样，我们可以使用两种方式来完成这个工作。

第一种方式自然是推荐的 Docker 的多阶段构建，我们在一个 `node` 镜像的环境中就可以打包我们的前端项目了，所以我们可以更改下 `Dockerfile` 文件，先进行 `node` 打包，然后再进行 `nginx` 启动：

```

FROM node:alpine as BUILD

WORKDIR /usr/src/app

RUN mkdir -p /usr/src/app

ADD . /usr/src/app

RUN npm install && \
    npm run build

FROM nginx:1.15.10-alpine
MAINTAINER cnych <cicnycy@gmail.com>

COPY --from=BUILD /usr/src/app/build /usr/share/nginx/html

ADD nginx.conf
/etc/nginx/conf.d/default.conf

```

第二种方式和上面一样在 Jenkins Pipeline 中添加一个打包构建的阶段即可，我们这里采用这种方式，所以 `Dockerfile` 文件还是使用第一个就行。

现在我们可以将客户端的代码推送到 Gitlab 上去，我们这里的仓库地址为：<http://git.qikqiak.com/course/polling-app-client.git>

Jenkins

现在项目准备好了，接下来我们可以开始 Jenkins 的配置，还记得前面在 Pipeline 结合 Kubernetes 的课程中我们使用了一个 `kubernetes` 的 Jenkins 插件，但是之前使用的方式有一些不妥的地方，我们 Jenkins Pipeline 构建任务绑定到了一个固定的 Slave Pod 上面，这样就需要我们的 Slave Pod 中必须

包含一系列构建所需要的依赖，比如 docker、maven、node、java 等等，这样就难免需要我们自己定义一个很庞大的 Slave 镜像，我们直接直接在 Pipeline 中去自定义 Slave Pod 中所需要用到的容器模板，这样我们需要什么镜像只需要在 Slave Pod Template 中声明即可，完全不需要去定义一个庞大的 Slave 镜像了。

首先去掉 Jenkins 中 kubernetes 插件中的 Pod Template 的定义，Jenkins -> 系统管理 -> 系统设置 -> 云 -> Kubernetes 区域，删除下方的 Kubernetes Pod Template -> 保存。

云

Kubernetes	
名称	kubernetes
Kubernetes 地址	https://kubernetes.default.svc.cluster.local
Kubernetes 服务证书 key	
禁用 HTTPS 证书检查	<input type="checkbox"/>
Kubernetes 命名空间	kube-ops
凭据	- 无 - <input type="button" value="添加"/>
Connection test successful <input style="float: right;" type="button" value="连接测试"/>	
Jenkins 地址	http://jenkins.kube-ops.svc.cluster.local:8080
Jenkins 通道	
Connection Timeout	5
Read Timeout	15
容器数量	10
Pod Retention	Never <input style="float: right;" type="button" value="高级..."/>
连接 Kubernetes API 的最大连接数	32
Seconds to wait for pod to be running	600
<input type="button" value="高级..."/>	
默认提供的模板名称	
镜像	<input type="button" value="添加 Pod 模板"/>
作为代理启动的镜像列表	
<input type="button" value="Delete cloud"/>	

jenkins kubernetes plugin

然后新建一个名为 polling-app-server 类型为 流水线(Pipeline) 的任务：

输入一个任务名称

polling-app-server

» 必填项

 **构建一个自由风格的软件项目**
这是Jenkins的主要功能.Jenkins将会结合任何SCM和任何构建系统来构建你的项目,甚至可以构建软件以外的系统.

 **流水线**
精心地组织一个可以长期运行在多个节点上的任务。适用于构建流水线（更加正式地应当称为工作流），增加或者组织难以采用自由风格的任务类型。

 **构建一个多配置项目**
适用于多配置项目,例如多环境测试,平台指定构建,等等.

 **Bitbucket Team/Project**
Scans a Bitbucket Cloud Team (or Bitbucket Server Project) for all repositories matching some defined markers.

 **文件夹**
创建一个可以嵌套存储的容器。利用它可以进行分组。视图仅仅是一个过滤器，而文件夹则是一个独立的命名空间，因此你可以有多个相同名称的内容，只要它们在不同的文件夹里即可。

 **GitHub 组织**
扫描一个 GitHub 组织（或者个人账户）的所有仓库来匹配已定义的标记。

确定

多分支流水线

new pipeline task

然后在这里需要勾选 触发远程构建 的触发器，其中令牌我们可以随便写一个字符串，然后记住下面的 URL，将 JENKINS_URL 替换成 Jenkins 的地址,我们这里的地址就是：<http://jenkins.qikqiak.com/job/polling-app-server/build?token=server321>

General 构建触发器 高级项目选项 流水线

构建触发器

- 其他工程构建后触发
- 定时构建
- GitHub hook trigger for GITScm polling
- 轮询 SCM
- 关闭构建
- 安静期
- 触发远程构建 (例如,使用脚本)

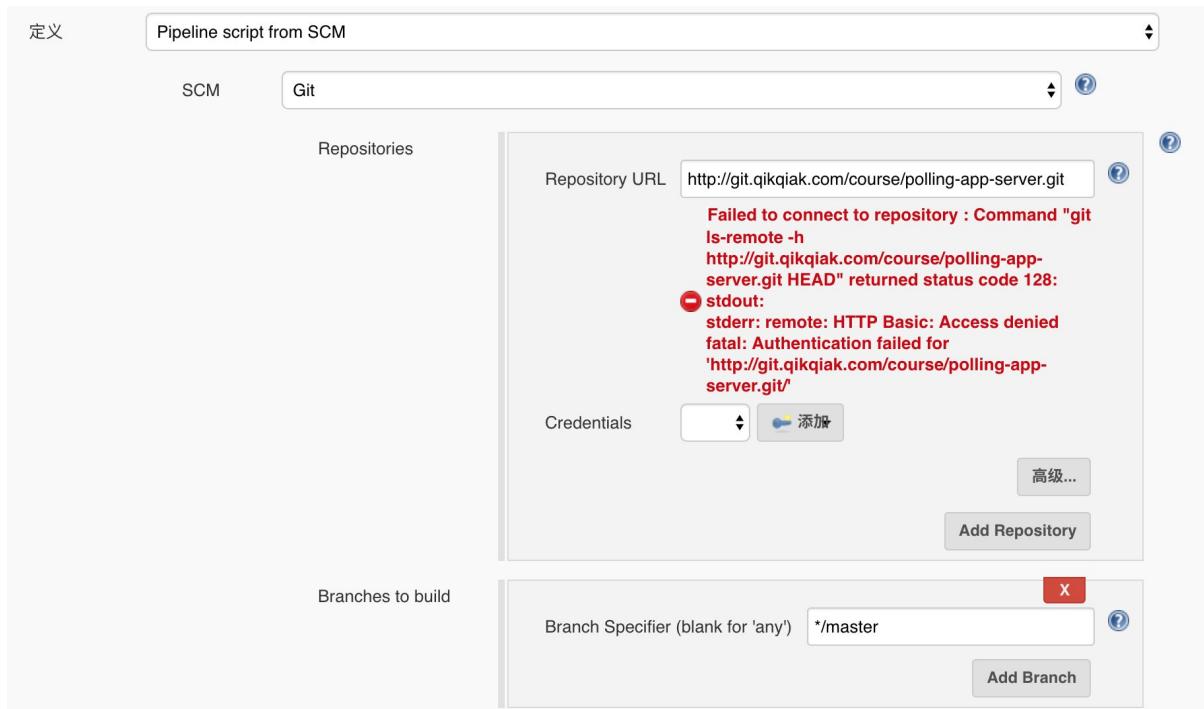
身份验证令牌 **server321**

Use the following URL to trigger build remotely: **JENKINS_URL/job/polling-app-server/build?token=TOKEN_NAME** 或者 **/buildWithParameters?token=TOKEN_NAME**

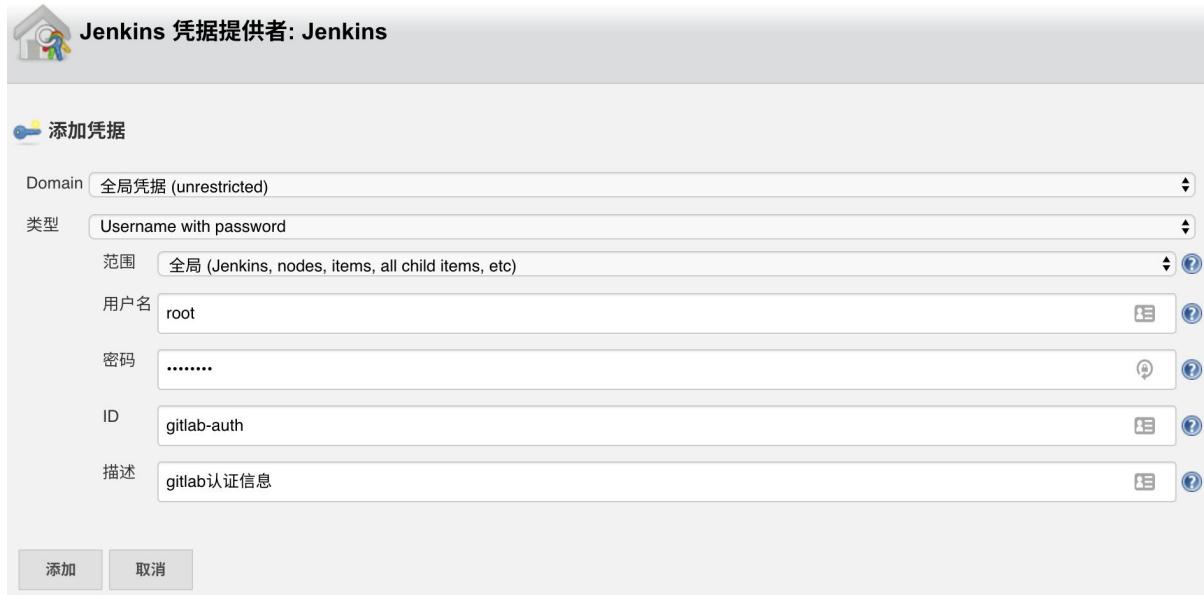
Optionally append &cause=Cause+Text to provide text that will be included in the recorded build cause.

trigger

然后在下面的 流水线 区域我们可以选择 Pipeline script 然后在下面测试流水线脚本，我们这里选择 Pipeline script from SCM，意思就是从代码仓库中通过 Jenkinsfile 文件获取 Pipeline script 脚本定义，然后选择 SCM 来源为 Git，在出现的列表中配置上仓库地址 `http://git.qikqiak.com/course/polling-app-server.git`，由于我们是在一个 Slave Pod 中去进行构建，所以如果使用 SSH 的方式去访问 Gitlab 仓库的话就需要频繁的去更新 SSH-KEY，所以我们这里采用直接使用用户名和密码的形式来方式：



在 Credentials 区域点击 添加 按钮添加我们访问 Gitlab 的用户名和密码：



gitlab auth

然后需要我们配置用于构建的分支，如果所有的分支我们都想要进行构建的话，只需要将 Branch Specifier 区域留空即可，一般情况下不同的环境对应的分支才需要构建，比如 master、develop、test 等，平时开发的 feature 或者 bugfix 的分支没必要频繁构建，我们这里就只配置 master 和 develop 两个分支用户构建：

Pipeline script from SCM

SCM Git

Repositories

Repository URL http://git.qikqiak.com/course/polling-app-server.git

Credentials root***** (gitlab认证信息) 添加

高级...

Add Repository

Branches to build

Branch Specifier (blank for 'any') */master

Branch Specifier (blank for 'any') */develop

Add Branch

源码库浏览器 (自动)

gitlab branch config

然后前往 Gitlab 中配置项目 `polling-app-server` Webhook, `settings -> Integrations`, 填写上面得到的 trigger 地址:

Integrations

Webhooks can be used for binding events when something is happening within the project.

URL: http://jenkins.qikqiak.com/job/polling-app-server/build?token=server321

Secret Token

Use this token to validate received payloads. It will be sent with the request in the X-Gitlab-Token HTTP header.

Trigger

- Push events**
This URL will be triggered by a push to the repository
Branch name or wildcard pattern to trigger on (leave blank for all)
- Tag push events**
This URL will be triggered when a new tag is pushed to the repository

webhook

保存后，可以直接点击 `Test -> Push Event` 测试是否可以正常访问 Webhook 地址，这里需要注意的是我们需要配置下 Jenkins 的安全配置，否则这里的触发器没权限访问 Jenkins，系统管理 -> 全局安全配置：取消 `防止跨站点请求伪造`，勾选上 `匿名用户具有可读权限`：

安全矩阵

登录用户可以做任何事

匿名用户具有可读权限

遗留模式

项目矩阵授权策略

标记格式器

标记格式器: 纯文本

把所有的输入都当作纯文本。HTML中像 < 和 & 的不安全字符会被转义。

代理

JNLP代理协议的TCP端口 指定端口 : 50000 随机选取 禁用

跨站请求伪造保护

防止跨站点请求伪造

security config

如果测试出现了 `Hook executed successfully: HTTP 201` 则证明 Webhook 配置成功了，否则就需要检查下 Jenkins 的安全配置是否正确了。

配置成功后我们只需要往 Gitlab 仓库推送代码就会触发 Pipeline 构建了。接下来我们直接在服务端代码仓库根目录下面添加 `Jenkinsfile` 文件，用于描述流水线构建流程。

首先定义最简单的流程，要注意这里和前面课程的不同之处，这里我们使用 `podTemplate` 来定义不同阶段使用的容器，有哪些阶段呢？

Clone 代码 -> 代码静态分析 -> 单元测试 -> Maven 打包 -> Docker 镜像构建/推送 -> Helm 更新服务。

Clone 代码在默认的 Slave 容器中即可；静态分析和单元测试我们这里直接忽略，有需要这个阶段的同学自己添加上即可；Maven 打包肯定就需要 Maven 的容器了；Docker 镜像构建/推送是不是就需要 Docker 环境了呀；最后的 Helm 更新服务是不是就需要一个有 Helm 的容器环境了，所以我们这里就可以很简单的定义 `podTemplate` 了，如下定义：(添加一个 `kubectl` 工具用于测试)

```
def label = "slave-${UUID.randomUUID().toString()}"  
  
podTemplate(label: label, containers: [  
    containerTemplate(name: 'maven', image: 'maven:3.6-alpine', command: 'cat', ttyEnabled: true),  
    containerTemplate(name: 'docker', image: 'docker', command: 'cat', ttyEnabled: true),  
    containerTemplate(name: 'kubectl', image: 'cnych/kubectl', command: 'cat', ttyEnabled: true),  
    containerTemplate(name: 'helm', image: 'cnych/helm', command: 'cat', ttyEnabled: true)  
], volumes: [  
    hostPathVolume(mountPath: '/root/.m2', hostPath: '/var/run/m2'),  
    hostPathVolume(mountPath: '/home/jenkins/.kube', hostPath: '/root/.kube'),  
    hostPathVolume(mountPath: '/var/run/docker.sock', hostPath: '/var/run/docker.sock')  
]) {  

```

}

上面这段 groovy 脚本比较简单，我们需要注意的是 volumes 区域的定义，将容器中的 /root/.m2 目录挂载到宿主机上是为了给 Maven 构建添加缓存的，不然每次构建的时候都需要去重新下载依赖，这样就非常慢了；挂载 .kube 目录是为了能够让 kubectl 和 helm 两个工具可以读取到 Kubernetes 集群的连接信息，不然我们是没办法访问到集群的；最后挂载 /var/run/docker.sock 文件是为了能够让我们的 docker 这个容器获取到 Docker Daemon 的信息的，因为 docker 这个镜像里面只有客户端的二进制文件，我们需要使用宿主机的 Docker Daemon 来构建镜像，当然我们也需要在运行 Slave Pod 的节点上拥有访问集群的文件，然后在每个 Stage 阶段使用特定需要的容器来进行任务的描述即可，所以这几个 volumes 都是非常重要的

```
volumes: [
  hostPathVolume(mountPath: '/root/.m2', hostPath: '/var/run/m2'),
  hostPathVolume(mountPath: '/home/jenkins/.kube', hostPath: '/root/.kube'),
  hostPathVolume(mountPath: '/var/run/docker.sock', hostPath: '/var/run/docker.sock')
]
```

另外一个值得注意的就是 `label` 标签的定义，我们这里使用 UUID 生成一个随机的字符串，这样可以让 Slave Pod 每次的名称都不一样，而且这样就不会被固定在一个 Pod 上面了，以后有多个构建任务的时候就不会存在等待的情况了，这和我们之前的课程中讲到的固定在一个 `label` 标签上有所不同。

然后我们将上面的 Jenkinsfile 文件提交到 Gitlab 代码仓库上：

```
$ git add Jenkinsfile  
$ git commit -m "添加 Jenkinsfile 文件"  
$ git push origin master
```

然后切换到 Jenkins 页面上，正常情况就可以看到我们的流水线任务 `polling-app-server` 已经被触发构建了，然后回到我们的 Kubernetes 集群中可以看到多了一个 `slave` 开头的 Pod，里面有5个容器，就是我们上面 `podTemplate` 中定义的4个容器，加上一个默认的 `jenkins slave` 容器，同样的，构建任务完成后，这个 Pod 也会被自动销毁掉：

Kubernetes Pod Status					
NAME	READY	STATUS	RESTARTS	A	
jenkins-7fbfcc5ddc-xsqmt	1/1	Running	0	1	
slave-6e898009-62a2-4798-948f-9c80c3de419b-0jwml-6t6hb	5/5	Running	0	36	

正常可以看到 Jenkins 中的任务构建成功了：

The screenshot shows the Jenkins interface for a pipeline job named 'polling-app-server'. The left sidebar contains various Jenkins navigation links. The main content area is titled '控制台输出' (Console Output) and displays the build logs. The logs show a successful checkout of the repository from 'git://git.qikqiaok.com/course/polling-app-server.git', followed by a series of Maven commands: 'mvn clean package -Dmaven.test.skip=true'. The logs conclude with 'Finished: SUCCESS'.

```

Started by user admin
Lightweight checkout support not available, falling back to full checkout.
Checking out git http://git.qikqiaok.com/course/polling-app-server.git into /var/jenkins_home/workspace/polling-app-server@script to read Jenkinsfile
using credential gitlab-auth
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git config remote.origin.url http://git.qikqiaok.com/course/polling-app-server.git # timeout=10
Fetching upstream changes from http://git.qikqiaok.com/course/polling-app-server.git
> git --version # timeout=10
using GIT_ASKPASS to set credentials gitlab认证信息
> git fetch -tags --progress http://git.qikqiaok.com/course/polling-app-server.git +refs/heads/*:refs/remotes/origin/*
Seen branch in repository origin/master
Seen 1 remote branch
> git show-ref --tags -d # timeout=10
Checking out Revision 80b954e4ac8c5d982cdb765d233319b5c3bd2608 (origin/master)
> git config core.sparsecheckout # timeout=10
> git checkout -f 80b954e4ac8c5d982cdb765d233319b5c3bd2608
Commit message: "添加 Jenkinsfile 文件"
First time build. Skipping changelog.
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] podTemplate
[Pipeline] {
[Pipeline] node
Still waiting to schedule task
'slave-6e898009-62a2-4798-948f-9c80c3de419b-0jwml-6t6hb' is offline
Agent slave-6e898009-62a2-4798-948f-9c80c3de419b-0jwml-6t6hb is provisioned from template Kubernetes Pod Template
Agent specification [Kubernetes Pod Template] (slave-6e898009-62a2-4798-948f-9c80c3de419b):
* [maven] maven:3.6-alpine
* [docker] docker
* [kubectl] cnych/kubectl
* [helm] cnych/helm

Running on slave-6e898009-62a2-4798-948f-9c80c3de419b-0jwml-6t6hb in /home/jenkins/workspace/polling-app-server
[Pipeline] { (show)
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
Finished: SUCCESS

```

build successfully

接下来的工作就是来实现上面具体的 Pipeline 脚本了。

Pipeline

第一个阶段：单元测试，我们可以在这个阶段运行一些单元测试或者静态代码分析的脚本，我们这里直接忽略。

第二个阶段：代码编译打包，我们可以看到我们是在一个 maven 的容器中来执行的，所以我们只需要在该容器中获取到代码，然后在代码目录下面执行 maven 打包命令即可，如下所示：

```

stage('代码编译打包') {
    try {
        container('maven') {
            echo "2. 代码编译打包阶段"
            sh "mvn clean package -Dmaven.test.skip=true"
        }
    } catch (exc) {
        println "构建失败 - ${currentBuild.displayName}"
        throw(exc)
    }
}

```

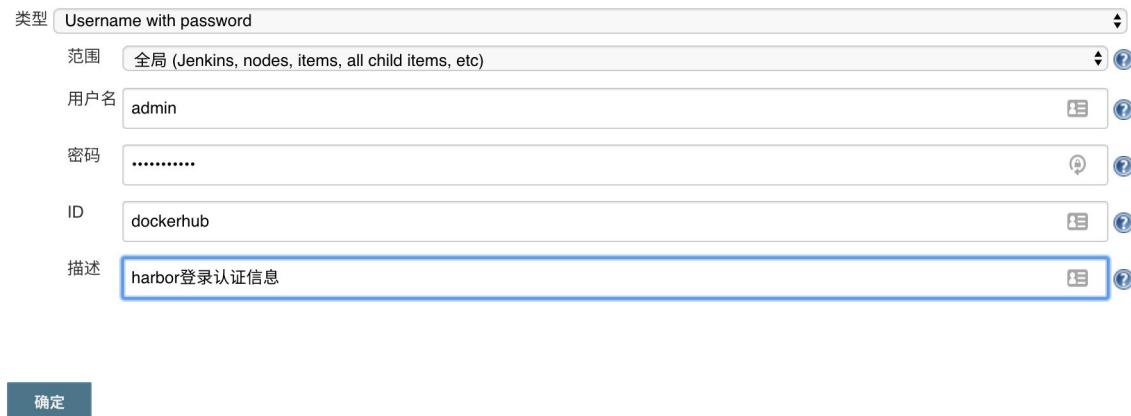
第三个阶段：构建 Docker 镜像，要构建 Docker 镜像，就需要提供镜像的名称和 tag，要推送到 Harbor 仓库，就需要提供登录的用户名和密码，所以我们这里使用到了 `withCredentials` 方法，在里面可以提供一个 `credentialsId` 为 `dockerhub` 的认证信息，如下：

```
container('构建 Docker 镜像') {
    withCredentials([[${class: 'UsernamePasswordMultiBinding',
        credentialsId: 'dockerhub',
        usernameVariable: 'DOCKER_HUB_USER',
        passwordVariable: 'DOCKER_HUB_PASSWORD'}]])
        container('docker') {
            echo "3. 构建 Docker 镜像阶段"
            sh """
                docker login ${dockerRegistryUrl} -u ${DOCKER_HUB_USER} -p ${DOCKER_HUB_PASSWORD}
            """
            docker build -t ${image}:${imageTag} .
            docker push ${image}:${imageTag}
            """
        }
    }
}
```

其中 `${image}` 和 `${imageTag}` 我们可以在上面定义成全局变量：

```
def imageTag = sh(script: "git rev-parse --short HEAD", returnStdout: true).trim()
def dockerRegistryUrl = "registry.qikqiak.com"
def imageEndpoint = "course/polling-app-server"
def image = "${dockerRegistryUrl}/${imageEndpoint}"
```

docker 的用户名和密码信息则需要通过 凭据 来进行添加，进入 jenkins 首页 -> 左侧菜单 凭据 -> 添加凭据，选择用户名和密码类型的，其中 ID 一定要和上面的 `credentialsId` 的值保持一致：



add docker hub credential

第四个阶段：运行 `kubectl` 工具，其实在我们当前使用的流水线中是用不到 `kubectl` 工具的，那么为什么我们这里要使用呢？这还不是因为我们暂时还没有去写应用的 Helm Chart 包吗？所以我们先去用原始的 YAML 文件来编写应用部署的资源清单文件，这也是我们写出 Chart 包前提，因为只有知道了应

用如何部署才可能知道 Chart 包如何编写，所以我们先编写应用部署资源清单。

首先当然就是 Deployment 控制器了，如下所示：（k8s.yaml）

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: polling-server
  namespace: course
  labels:
    app: polling-server
spec:
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: polling-server
    spec:
      restartPolicy: Always
      imagePullSecrets:
        - name: myreg
      containers:
        - image: <IMAGE>:<IMAGE_TAG>
          name: polling-server
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
              name: api
          env:
            - name: DB_HOST
              value: mysql
            - name: DB_PORT
              value: "3306"
            - name: DB_NAME
              value: polling_app
            - name: DB_USER
              value: polling
            - name: DB_PASSWORD
              value: polling321
      ---  

  kind: Service
  apiVersion: v1
  metadata:
    name: polling-server
    namespace: course
  spec:
    selector:
      app: polling-server
    type: ClusterIP
    ports:
      - name: api-port
        port: 8080
        targetPort: api
```

```


apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mysql
  namespace: course
spec:
  template:
    metadata:
      labels:
        app: mysql
    spec:
      restartPolicy: Always
      containers:
        - name: mysql
          image: mysql:5.7
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 3306
              name: dbport
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: rootPassW0rd
            - name: MYSQL_DATABASE
              value: polling_app
            - name: MYSQL_USER
              value: polling
            - name: MYSQL_PASSWORD
              value: polling321
          volumeMounts:
            - name: db
              mountPath: /var/lib/mysql
      volumes:
        - name: db
          hostPath:
            path: /var/lib/mysql

kind: Service
apiVersion: v1
metadata:
  name: mysql
  namespace: course
spec:
  selector:
    app: mysql
  type: ClusterIP
  ports:
    - name: dbport
      port: 3306
      targetPort: dbport


```

可以看到我们上面的 YAML 文件中添加使用的镜像是用标签代替的：`<IMAGE>:<IMAGE_TAG>`，这是因为我们的镜像地址是动态的，下依赖我们在上一个阶段打包出来的镜像地址的，所以我们这里用标签代替，然后将标签替换成真正的值即可，另外为了保证应用的稳定性，我们还在应用中添加了健康检查，所以需要在代码中添加一个健康检查的 Controller：

(src/main/java/com/example/polls/controller/StatusController.java)

```

package com.example.polls.controller;

import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/_status/healthz")
public class StatusController {

    @GetMapping
    public String healthCheck() {
        return "UP";
    }

}

```

最后就是环境变量了，还记得前面我们更改了资源文件中数据库的配置吗？

(src/main/resources/application.properties) 因为要尽量通用，我们在部署应用的时候很有可能已经有一个外部的数据库服务了，所以这个时候通过环境变量传入进来即可。另外由于我们这里使用的是私有镜像仓库，所以需要在集群中提前创建一个对应的 Secret 对象：

```

$ kubectl create secret docker-registry myreg --docker-server=registry.qikqiak.com --docke
r-username=DOCKER_USER --docker-password=DOCKER_PASSWORD --docker-email=DOCKER_EMAIL --nam
espace course

```

在代码根目录下面创建一个 manifests 的目录，用来存放上面的资源清单文件，正常来说是不是我们只需要在镜像构建成功后，将上面的 k8s.yaml 文件中的镜像标签替换掉就 OK，所以这一步的动作如下：

```

stage('运行 Kubectl') {
    container('kubectl') {
        echo "查看 K8S 集群 Pod 列表"
        sh "kubectl get pods"
        sh """
            sed -i "s/<IMAGE>/${image}" manifests/k8s.yaml
            sed -i "s/<IMAGE_TAG>/${imageTag}" manifests/k8s.yaml
            kubectl apply -f k8s.yaml
        """
    }
}

```

第五阶段：运行 Helm 工具，就是直接使用 Helm 来部署应用了，现在有了上面的基本的资源对象了，要创建 Chart 模板就相对容易了，Chart 模板仓库地址：<https://github.com/cnyc/helm-polling-helm>，我们可以根据 values.yaml 文件来进行自定义安装，模板中我们定义了可以指定使用外部数据库服务或者内部独立的数据库服务，具体的我们可以去看模板中的定义。首先我们可以先使用这个模板在集群中来测试下。首先在集群中 Clone 上面的 Chart 模板：

```

$ git clone https://github.com/cnyc/helm-polling-helm.git

```

然后我们使用内部的数据库服务，新建一个 custom.yaml 文件来覆盖 values.yaml 文件中的值：

```

persistence:
  enabled: true
  persistentVolumeClaim:
    database:
      storageClass: "database"

database:
  type: internal
  internal:
    database: "polling"
    # 数据库用户
    username: "polling"
    # 数据库用户密码
    password: "polling321"

```

可以看到我们这里使用了一个名为 `database` 的 StorageClass 对象，所以还得创建先创建这个资源对象：

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: database
provisioner: fuseim.pri/ifs

```

然后我们就可以在 Chart 根目录下面安装应用，执行下面的命令：

```

$ helm upgrade --install polling -f custom.yaml . --namespace course
Release "polling" does not exist. Installing it now.
NAME: polling
LAST DEPLOYED: Sat May 4 23:31:42 2019
NAMESPACE: course
STATUS: DEPLOYED

RESOURCES:
==> v1/Pod(related)
NAME                           READY   STATUS        RESTARTS   AGE
polling-polling-api-6b699478d6-lqwhw  0/1     ContainerCreating  0          0s
polling-polling-ui-587bbfb7b5-xr2ff   0/1     ContainerCreating  0          0s
polling-polling-database-0           0/1     Pending       0          0s

==> v1/Secret
NAME          TYPE  DATA  AGE
polling-polling-database  Opaque  1      0s

==> v1/Service
NAME            TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
polling-polling-api   ClusterIP  10.109.19.220  <none>        8080/TCP  0s
polling-polling-database ClusterIP  10.98.136.190  <none>        3306/TCP  0s
polling-polling-ui    ClusterIP  10.108.170.43   <none>        80/TCP    0s

==> v1beta2/Deployment
NAME        DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
polling-polling-api  1         1         1           0          0s
polling-polling-ui   1         1         1           0          0s

==> v1/StatefulSet

```

```
NAME          DESIRED  CURRENT  AGE
polling-polling-database  1        1        0s

==> v1beta1/Ingress
NAME          HOSTS          ADDRESS      PORTS  AGE
polling-polling-ingress  ui.polling.domain  80        0s

NOTES:
1. Get the application URL by running these commands:
http://ui.polling.domain

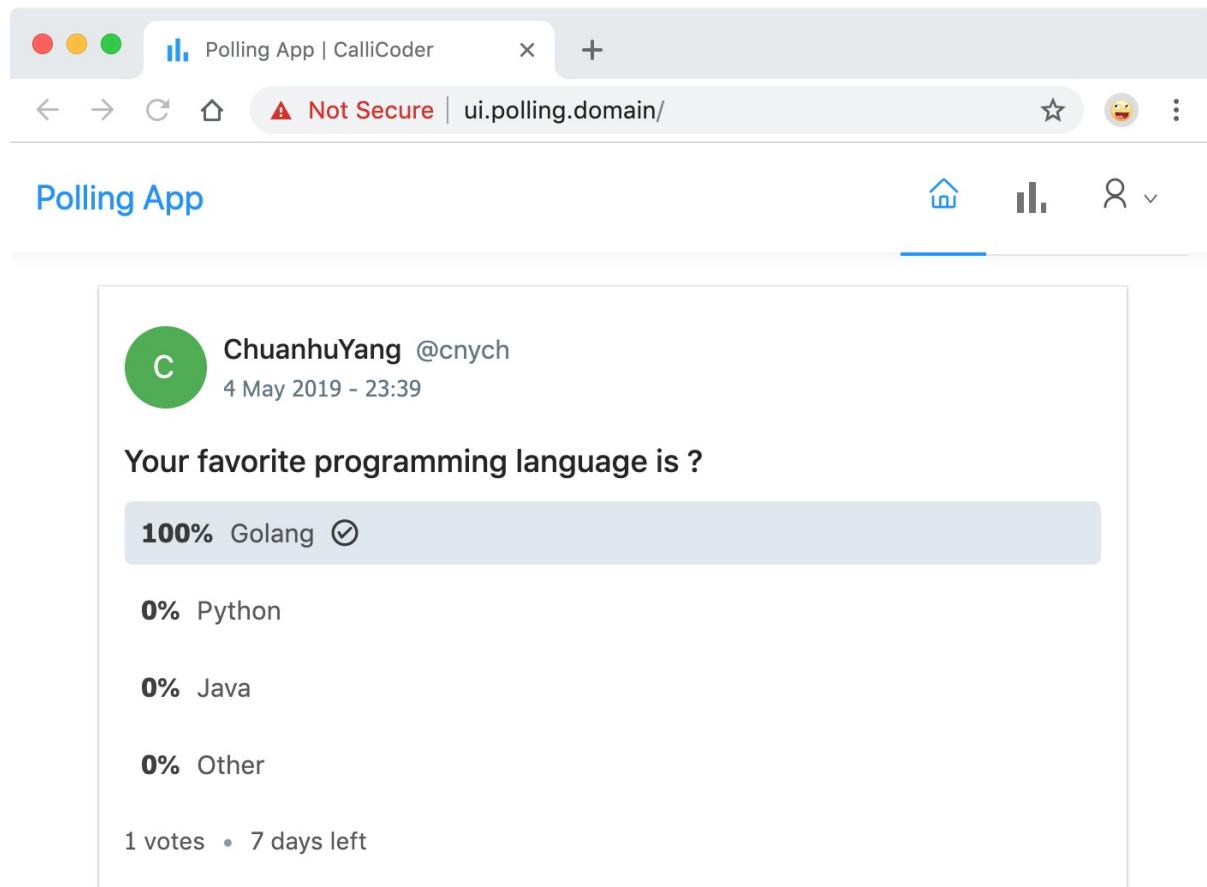
You have new mail in /var/spool/mail/root
```

注意我们这里安装也是使用的 `helm upgrade` 命令，这样有助于安装和更新的时候命令统一。

安装完成后，查看下 Pod 的运行状态：

```
$ kubectl get pods -n course
NAME                  READY   STATUS    RESTARTS   AGE
polling-polling-api-6b699478d6-lqwhw  1/1     Running   0          3m
polling-polling-database-0              1/1     Running   0          3m
polling-polling-ui-587bbfb7b5-xr2ff   1/1     Running   0          3m
```

然后我们可以在本地 `/etc/hosts` 里面加上 `http://ui.polling.domain` 的的映射，这样我们就可以通过这个域名来访问我们安装的应用了，可以注册、登录、发表投票内容了：



polling app

这样我们就完成了使用 Helm Chart 安装应用的过程，但是现在我们使用的包还是直接使用的 git 仓库中的，平常我们正常安装的时候都是使用的 Chart 仓库中的包，所以我们需要将该 Chart 包上传到一个仓库中去，比较幸运的是我们的 Harbor 也是支持 Helm Chart 包的。我们可以选择手动通过 Harbor 的 Dashboard 将 Chart 包进行上传，也可以通过使用 Helm Push 插件：

```
$ helm plugin install https://github.com/chartmuseum/helm-push
Downloading and installing helm-push v0.7.1 ...
https://github.com/chartmuseum/helm-push/releases/download/v0.7.1/helm-push_0.7.1_linux_amd64.tar.gz

Installed plugin: push
```

当然我们需要首先将 Harbor 提供的仓库添加到 helm repo 中，由于是私有仓库，所以在添加的时候我们需要添加用户名和密码：

```
$ helm repo add course https://registry.qikqiak.com/chartrepo/course --username=<harbor用户名>
> --password=<harbor密码>
"course" has been added to your repositories
```

这里的 repo 的地址是 <Harbor URL>/chartrepo/<Harbor中项目名称>， Harbor 中每个项目是分开的 repo，如果不提供项目名称，则默认使用 library 这个项目。

需要注意的是如果你的 Harbor 是采用的自建的 https 证书，这里就需要提供 ca 证书和私钥文件了，否则会出现证书校验失败的错误 x509: certificate signed by unknown authority 。我们这里是通过 cert-manager 为 Harbor 提供的一个信任的 https 证书，所以没有指定 ca 证书相关的参数。

然后我们将上面的 polling-helm 这个 Chart 包上传到 Harbor 仓库中去：

```
$ helm push polling-helm course
Pushing polling-0.1.0.tgz to course...
Done.
```

这个时候我们登录的 Harbor 仓库中去，查看 course 这个项目下面的 Helm Charts 就可以发现多了一个 polling 的应用了：

The screenshot shows the Harbor UI interface. The top navigation bar includes a back button, forward button, search bar, and user information for 'admin'. The main content area shows the 'course' project under 'Charts'. A specific Helm chart named 'polling:0.1.0' is selected. The left sidebar has sections like '项目', '日志', '系统管理', '用户管理', '仓库管理', '复制管理', and '配置管理'. The right panel displays detailed information about the 'polling-helm' chart, including its version (0.1.0), maintainer (阳明), and creation date (2019年5月5日). It also lists commands for adding a repository and installing the chart, and a status section indicating a pending file download.

helm chart

我们也可以在右下角看到有添加仓库和安装 Chart 的相关命令。

到这里 Helm 相关的工作就准备好了。那么我们如何在 Jenkins Pipeline 中去使用 Helm 呢？我们可以回顾下，我们平时的一个 CI/CD 的流程：开发代码 -> 提交代码 -> 触发镜像构建 -> 修改镜像tag -> 推送到镜像仓库中去 -> 然后更改 YAML 文件镜像版本 -> 使用 kubectl 工具更新应用。

现在我们是不是直接使用 Helm 了，就不需要去手动更改 YAML 文件了，也不需要使用 kubectl 工具来更新应用了，而是只需要去覆盖下 helm 中的镜像版本，直接 upgrade 是不是就可以达到应用更新的结果了。我们可以去查看下 chart 包的 values.yaml 文件中关于 api 服务的定义：

```
api:
  image:
    repository: cnych/polling-api
    tag: 0.0.7
    pullPolicy: IfNotPresent
```

我们是不是只需要将上面关于 api 服务使用的镜像用我们这里 Jenkins 构建后的替换掉就可以了，这样我们更改上面的最后 运行 Helm 的阶段如下：

```
stage('运行 Helm') {
  container('helm') {
    echo "更新 polling 应用"
    sh """
      helm upgrade --install polling polling --set persistence.persistentVolumeClaim.database.storageClass=database --set database.type=internal --set database.internal.database=polling --set database.internal.username=polling --set database.internal.password=polling321
      --set api.image.repository=${image} --set api.image.tag=${imageTag} --set imagePullSecrets[0].name=myreg --namespace course
    """
  }
}
```

当然我们可以将需要更改的值都放入一个 YAML 之中来进行修改，我们这里通过 --set 来覆盖对应的值，这样整个 API 服务的完整 Jenkinsfile 文件如下所示：

```
def label = "slave-${UUID.randomUUID().toString()}"  
  
def helmLint(String chartDir) {  
  println "校验 chart 模板"  
  sh "helm lint ${chartDir}"  
}  
  
def helmInit() {  
  println "初始化 helm client"  
  sh "helm init --client-only --stable-repo-url https://mirror.azure.cn/kubernetes/charts/"  
}  
  
def helmRepo(Map args) {  
  println "添加 course repo"  
  sh "helm repo add --username ${args.username} --password ${args.password} course https://registry.qikqiak.com/chartrepo/course"  
  
  println "更新 repo"  
  sh "helm repo update"  
  
  println "获取 Chart 包"  
  sh """  
    helm fetch course/polling  
    tar -xzvf polling-0.1.0.tgz  
  """
```

```

}

def helmDeploy(Map args) {
    helmInit()
    helmRepo(args)

    if (args.dry_run) {
        println "Debug 应用"
        sh "helm upgrade --dry-run --debug --install ${args.name} ${args.chartDir} --set persistence.persistentVolumeClaim.database.storageClass=database --set database.type=internal --set database.internal.database=polling --set database.internal.username=polling --set database.internal.password=polling321 --set api.image.repository=${args.image} --set api.image.tag=${args.tag} --set imagePullSecrets[0].name=myreg --namespace=${args.namespace}"
    } else {
        println "部署应用"
        sh "helm upgrade --install ${args.name} ${args.chartDir} --set persistence.persistentVolumeClaim.database.storageClass=database --set database.type=internal --set database.internal.database=polling --set database.internal.username=polling --set database.internal.password=polling321 --set api.image.repository=${args.image} --set api.image.tag=${args.tag} --set imagePullSecrets[0].name=myreg --namespace=${args.namespace}"
        echo "应用 ${args.name} 部署成功. 可以使用 helm status ${args.name} 查看应用状态"
    }
}

podTemplate(label: label, containers: [
    containerTemplate(name: 'maven', image: 'maven:3.6-alpine', command: 'cat', ttyEnabled: true),
    containerTemplate(name: 'docker', image: 'docker', command: 'cat', ttyEnabled: true),
    containerTemplate(name: 'helm', image: 'cnych/helm', command: 'cat', ttyEnabled: true)
], volumes: [
    hostPathVolume(mountPath: '/root/.m2', hostPath: '/var/run/m2'),
    hostPathVolume(mountPath: '/home/jenkins/.kube', hostPath: '/root/.kube'),
    hostPathVolume(mountPath: '/var/run/docker.sock', hostPath: '/var/run/docker.sock')
]) {
    node(label) {
        def myRepo = checkout scm
        def gitCommit = myRepo.GIT_COMMIT
        def gitBranch = myRepo.GIT_BRANCH
        def imageTag = sh(script: "git rev-parse --short HEAD", returnStdout: true).trim()
        def dockerRegistryUrl = "registry.qikqiak.com"
        def imageEndpoint = "course/polling-api"
        def image = "${dockerRegistryUrl}/${imageEndpoint}"

        stage('单元测试') {
            echo "1. 测试阶段"
        }
        stage('代码编译打包') {
            try {
                container('maven') {
                    echo "2. 代码编译打包阶段"
                    sh "mvn clean package -Dmaven.test.skip=true"
                }
            } catch (exc) {
                println "构建失败 - ${currentBuild.displayName}"
                throw(exc)
            }
        }
        container('构建 Docker 镜像') {
            withCredentials([[${class: 'UsernamePasswordMultiBinding',

```

```

credentialsId: 'dockerhub',
usernameVariable: 'DOCKER_HUB_USER',
passwordVariable: 'DOCKER_HUB_PASSWORD'])] {
    container('docker') {
        echo "3. 构建 Docker 镜像阶段"
        sh """
            docker login ${dockerRegistryUrl} -u ${DOCKER_HUB_USER} -p ${DOCKER_HUB_PASS
WORD}
            docker build -t ${image}:${imageTag} .
            docker push ${image}:${imageTag}
        """
    }
}

stage('运行 Helm') {
    withCredentials([[$class: 'UsernamePasswordMultiBinding',
    credentialsId: 'dockerhub',
    usernameVariable: 'DOCKER_HUB_USER',
    passwordVariable: 'DOCKER_HUB_PASSWORD']] {
        container('helm') {
            // todo, 可以做分支判断
            echo "4. [INFO] 开始 Helm 部署"
            helmDeploy(
                dry_run      : false,
                name         : "polling",
                chartDir     : "polling",
                namespace   : "course",
                tag          : "${imageTag}",
                image        : "${image}",
                username     : "${DOCKER_HUB_USER}",
                password     : "${DOCKER_HUB_PASSWORD}"
            )
            echo "[INFO] Helm 部署应用成功..."
        }
    }
}
}

```

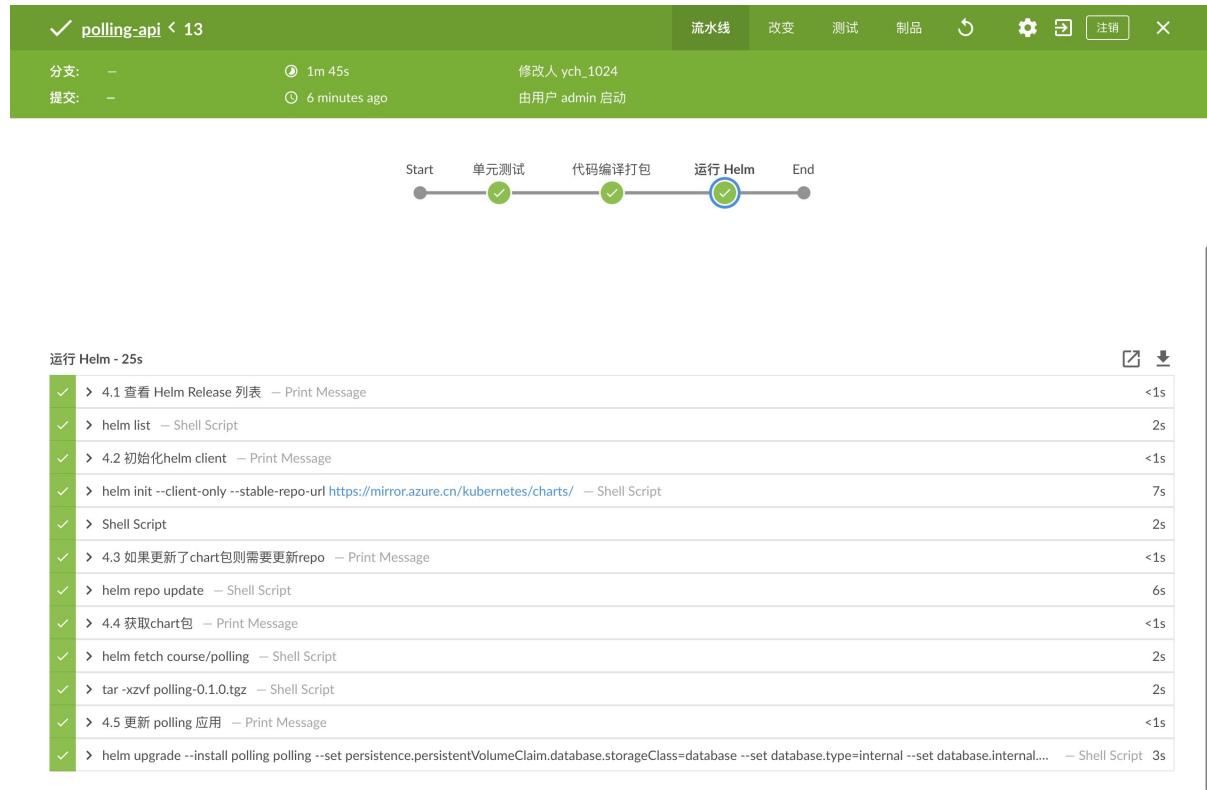
由于我们没有将 chart 包放入到 API 服务的代码仓库中，这是因为我们这里使用的 chart 包涉及到两个应用，一个 API 服务，一个是前端展示的服务，所以我们这里是通过脚本里面去主动获取到 chart 包来进行安装的，如果 chart 包跟随代码仓库一起管理当然就要简单许多了。

现在我们去更新 Jenkinsfile 文件，然后提交到 gitlab 中，然后去观察下 Jenkins 中的构建是否成功，我们重点观察下 Helm 阶段：

```
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "course" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. ✨ Happy Helming! ✨
[Pipeline] echo (hide)
4.4 获取chart包
[Pipeline] sh
+ helm fetch course/polling
[Pipeline] sh
+ tar -xvf polling-0.1.0.tgz
polling/Chart.yaml
polling/values.yaml
polling/templates/NOTES.txt
polling/templates/_helpers.tpl
polling/templates/api/deployment.yaml
polling/templates/api/service.yaml
polling/templates/database/database-secret.yaml
polling/templates/database/database-ss.yaml
polling/templates/database/database-svc.yaml
polling/templates/ingress/ingress.yaml
polling/templates/ui/deployment.yaml
polling/templates/ui/service.yaml
polling/.helmignore
polling/README.md
polling/custom.yaml
[Pipeline] echo
4.5 更新 polling 应用
[Pipeline] sh
+ helm upgrade --install polling polling --set 'persistence.persistentVolumeClaim.database.storageClass=database' --set 'database.type=internal' --
set 'database.internal.database=polling' --set 'database.internal.username=polling' --set 'database.internal.password=polling321' --set
'api.image.repository=registry.qikqiak.com/course/polling-api' --set 'api.image.tag=57530aa' --set 'imagePullSecrets[0].name=myreg' --namespace
course
Release "polling" has been upgraded. Happy Helming!
LAST DEPLOYED: Sat May 4 18:44:26 2019
NAMESPACE: course
STATUS: DEPLOYED
```

jenkins helm console

当然我们还可以去做一些必要的判断工作，比如根据分支判断是否需要自动部署等等，同样也可以切换到 Blue Ocean 界面查看构建结果。



jenkins blue ocean

现在大家可以尝试去修改下代码，然后提交代码到 gitlab 上，观察下 Jenkins 是否能够自动帮我们完成整个 CI/CD 的过程。

作业：现在还有一个前端展示的项目：<http://git.qikqiak.com/course/polling-app-client.git>，大家针对这个项目使用上面的 gitlab + jenkins + harbor + helm 来完成一个 Jenkins Pipeline 流水线的编写，尝试去修改下前端页面内容，看是否能够生效。

Copyright © qikqiak.com 2018 all right reserved, powered by Gitbook Updated: 2019-05-05 09:19:18