



# Kubernetes Handbook

Jimmy Song

# 目錄

0.0 介绍	1.1
1.0 Kubernetes集群安装	1.2
1.1 创建 TLS 通信所需的证书和秘钥	1.2.1
1.2 创建kubeconfig 文件	1.2.2
1.3 创建三节点的高可用etcd集群	1.2.3
1.4 安装kubectl命令行工具	1.2.4
1.5 部署高可用master集群	1.2.5
1.6 部署node 节点	1.2.6
1.7 部署kubedns插件	1.2.7
1.8 安装dashboard插件	1.2.8
1.9 安装heapster插件	1.2.9
1.10 安装EFK插件	1.2.10
2.0 Kubernetes服务发现与负载均衡	1.3
2.1 Ingress解析	1.3.1
2.2 安装Traefik ingress	1.3.2
2.3 分布式负载测试	1.3.3
2.4 kubernetes网络和集群性能测试	1.3.4
2.5 边缘节点配置	1.3.5
3.0 Kubernetes中的容器设计模式	1.4
4.0 Kubernetes中的概念解析	1.5
4.1 Deployment概念解析	1.5.1
5.0 Kubernetes的安全设置	1.6
5.1 Kubernetes 中的RBAC支持	1.6.1
6.0 Kubernetes网络配置	1.7
6.1 Kubernetes 中的网络模式解析	1.7.1
7.0 Kubernetes存储配置	1.8
7.1 使用glusterfs做持久化存储	1.8.1

---

8.0 集群运维管理	1.9
8.1 服务滚动升级	1.9.1
9.0 Kubernetes领域应用	1.10
10.0 问题记录	1.11

---

# Kubernetes Handbook

玩转Kubernetes，我就看kubernetes handbook！

本书所有的组件安装、示例和操作等都基于**Kubernetes1.6.0**版本。

文章同步更新到[gitbook](#)，方便大家浏览和下载PDF。

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

## 目录

- 0.0 介绍
- 1.0 Kubernetes集群安装
  - 1.1 创建 TLS 通信所需的证书和秘钥
  - 1.2 创建kubeconfig 文件
  - 1.3 创建三节点的高可用etcd集群
  - 1.4 安装kubectl命令行工具
  - 1.5 部署高可用master集群
  - 1.6 部署node节点
  - 1.7 安装kubedns插件
  - 1.8 安装dashboard插件
  - 1.9 安装heapster插件
  - 1.10 安装EFK插件
- 2.0 Kubernetes服务发现与负载均衡
  - 2.1 Ingress解析
  - 2.2 安装traefik ingress
  - 2.3 分布式负载测试
  - 2.4 kubernetes网络和集群性能测试
  - 2.5 边缘节点配置
- 3.0 Kubernetes中的容器设计模式 TODO
- 4.0 Kubernetes中的概念解析
  - 4.1 Deployment概念解析
- 5.0 Kubernetes的安全设置
  - 5.1 Kubernetes中的RBAC支持

- 6.0 Kubernetes 网络配置
  - [6.1 Kubernetes 中的网络模式解析](#)
- 7.0 Kubernetes 存储配置
  - [7.1 使用 glusterfs 做持久化存储](#)
- 8.0 集群运维管理
  - [8.1 服务滚动升级](#)
- 9.0 Kubernetes 领域应用
  - [9.1 Spark on Kubernetes TODO](#)
- 10.0 问题记录

## 说明

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

Kubernetes集群安装部分（1.0-1.10章节）在[opsnull](#)的基础上进行了编辑、修改和整理而成。

## 如何使用

在线浏览

访问gitbook：<https://www.gitbook.com/book/rootsongjc/kubernetes-handbook/>

本地查看

1. 将代码克隆到本地
2. 安装gitbook：[Setup and Installation of GitBook](#)
3. 执行`gitbook serve`
4. 在浏览器中访问<http://localhost:4000>
5. 生成的文档在`_book` 目录下

生成pdf

[下载Calibre](#)

在Mac下安装后，使用该命令创建链接

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

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

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

生成单个章节的**pdf**

使用 `pandoc` 和 `latex` 来生成**pdf**格式文档。

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

## 贡献者

[Jimmy Song](#)

[opsnull](#)

[godliness](#)

for GitBook      update 2017-05-12 16:45:40

# 部署 kubernetes 集群

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

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

部署完成后，你将理解系统各组件的交互原理，进而能快速解决实际问题。

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

注：本文档中不包括 `docker` 和私有镜像仓库的安装。

## 提供所有的配置文件

集群安装时所有组件用到的配置文件，包含在以下目录中：

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

## 集群详情

- Kubernetes 1.6.0
- Docker 1.12.5 ( 使用 yum 安装 )
- Etcd 3.1.5
- Flanneld 0.7 vxlan 网络
- TLS 认证通信 ( 所有组件，如 etcd、kubernetes master 和 node )
- RBAC 授权
- kublet TLS BootStrapping
- kubedns、dashboard、heapster(influxdb、grafana)、EFK(elasticsearch、fluentd、kibana) 集群插件
- 私有 docker 镜像仓库 `harbor` ( 请自行部署，harbor 提供离线安装包，直接使用 `docker-compose` 启动即可 )

## 步骤介绍

1. 创建 TLS 通信所需的证书和秘钥
2. 创建 kubeconfig 文件
3. 创建三节点的高可用 etcd 集群
4. kubectl 命令行工具
5. 部署高可用 master 集群
6. 部署 node 节点
7. 安装 kubedns 插件
8. 安装 dashboard 插件
9. 安装 heapster 插件
10. 安装 EFK 插件

## 提醒

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

## 关于

[Jimmy Song](#)

我的 [Kubernetes 相关文章](#)

for GitBook      update 2017-05-12 16:45:40

# 创建 kubernetes 各组件 TLS 加密通信的证书和秘钥

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

生成的 CA 证书和秘钥文件如下：

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

使用证书的组件如下：

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

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

## 安装 CFSSL

方式一：直接使用二进制源码包安装

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

### 方式二：使用go命令安装

我们的系统中安装了Go1.7.5，使用以下命令安装更快捷：

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

在 `$GOPATH/bin` 目录下得到以`cfssl`开头的几个命令。

注意：以下文章中出现的`cat`的文件名如果不存在需要手工创建。

## 创建 CA (Certificate Authority)

### 创建 CA 配置文件

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

### 字段说明

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

### 创建 **CA** 证书签名请求

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

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

生成 **CA** 证书和私钥

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

## 创建 **kubernetes** 证书

创建 kubernetes 证书签名请求

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

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

生成 **kubernetes** 证书和私钥

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

或者直接在命令行上指定相关参数：

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

## 创建 **admin** 证书

创建 **admin** 证书签名请求

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

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

生成 `admin` 证书和私钥

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

## 创建 `kube-proxy` 证书

### 创建 kube-proxy 证书签名请求

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

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

### 生成 kube-proxy 客户端证书和私钥

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

### 校验证书

以 kubernetes 证书为例

## 使用 **openssl** 命令

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

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

```

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

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

## 使用 cfssl-certinfo 命令

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

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

```

## 分发证书

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

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

## 参考

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

## 1.1 创建 TLS 通信所需的证书和秘钥

---

## 创建 kubeconfig 文件

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

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

## 创建 TLS Bootstrapping Token

### Token auth file

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

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

后三行是一句，直接复制上面的脚本运行即可。

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

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

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

## 创建 kubelet bootstrapping kubeconfig 文件

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

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

## 创建 kube-proxy kubeconfig 文件

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

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

## 分发 kubeconfig 文件

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

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

## 1.2 创建kubeconfig 文件

---

## 创建高可用 etcd 集群

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

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

## TLS 认证文件

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

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

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

## 下载二进制文件

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

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

## 创建 etcd 的 systemd unit 文件

注意替换IP地址为自己的etcd集群的主机IP。

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

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

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

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

完整 `unit` 文件见：[etcd.service](#)

环境变量配置文件 `/etc/etcd/etcd.conf`。

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

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

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

## 启动 `etcd` 服务

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

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

## 验证服务

在任一 kubernetes master 机器上执行如下命令：

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

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

for GitBook      update 2017-05-12 16:45:40

## 下载和配置 **kubectl** 命令行工具

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

### 下载 **kubectl**

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

### 创建 **kubectl kubeconfig** 文件

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

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

`cluster-admin` 绑定，该 Role 授予了调用 `kube-apiserver` 相关 API 的权限；

- 生成的 `kubeconfig` 被保存到 `~/.kube/config` 文件；

for GitBook update 2017-05-12 16:45:40

## 部署高可用 **kubernetes master** 集群

kubernetes master 节点包含的组件：

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

目前这三个组件需要部署在同一台机器上。

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

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

暂时未实现 master 节点的高可用。

## TLS 证书文件

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

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

## 下载最新版本的二进制文件

有两种下载方式

方式一

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

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

### 方式二

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

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

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

将二进制文件拷贝到指定路径

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

## 配置和启动 **kube-apiserver**

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

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

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

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

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

/etc/kubernetes/config 文件的内容为：

```

###  

# kubernetes system config  

#  

# The following values are used to configure various aspects of  

all  

# kubernetes services, including  

#  

#   kube-apiserver.service  

#   kube-controller-manager.service  

#   kube-scheduler.service  

#   kubelet.service  

#   kube-proxy.service  

# logging to stderr means we get it in the systemd journal  

KUBE_LOGTOSTDERR="--logtostderr=true"  

# journal message level, 0 is debug  

KUBE_LOG_LEVEL="--v=0"  

# Should this cluster be allowed to run privileged docker contain  

ners  

KUBE_ALLOW_PRIV="--allow-privileged=true"  

# How the controller-manager, scheduler, and proxy find the apis  

erver  

#KUBE_MASTER="--master=http://sz-pg-oam-docker-test-001.tendclou  

d.com:8080"  

KUBE_MASTER="--master=http://172.20.0.113:8080"

```

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

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

```

###  

## kubernetes system config  

##  

## The following values are used to configure the kube-apiserver  

##  

#

```

```

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

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

```

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

未通过授权的请求；

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

完整 unit 见 [kube-apiserver.service](#)

启动 **kube-apiserver**

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

## 配置和启动 **kube-controller-manager**

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

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

```

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

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

[Install]
WantedBy=multi-user.target

```

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

```

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

# defaults from config and apiserver should be adequate

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

```

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

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

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

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

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

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

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

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

## 启动 **kube-controller-manager**

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

## 配置和启动 **kube-scheduler**

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

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

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

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

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

配置文件 `/etc/kubernetes/scheduler`。

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

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

完整 unit 见 [kube-scheduler.service](#)

## 启动 **kube-scheduler**

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

## 验证 master 节点功能

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

## 部署kubernetes node 节点

kubernetes node 节点包含如下组件：

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

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

## 目录和文件

我们再检查一下三个节点上，经过前几步操作生成的配置文件。

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

## 配置Flanneld

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

直接使用yum安装flanneld即可。

```
yum install -y flannel
```

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

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

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

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

/etc/sysconfig/flanneld 配置文件。

```
# Flanneld configuration options

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

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

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

在FLANNEL\_OPTIONS中增加TLS的配置。

在**etcd**中创建网络配置

执行下面的命令为**docker**分配IP地址段。

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

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

配置**Docker**

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

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

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

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

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

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

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

`/run/flannel/subnet.env`

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

`/run/docker_opts.env`

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

现在查询etcd中的内容可以看到：

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

设置**docker0**网桥的**IP地址**

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

这样**docker0**和**flannel**网桥会在同一个子网中，如

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

现在就可以重启 docker了。

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

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

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

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

启动 **flannel**

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

## 安装和配置 **kubelet**

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

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

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

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

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

## 创建 **kubelet** 的**service**配置文件

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

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

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

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

kubelet的配置文件 `/etc/kubernetes/kubelet`。其中的IP地址更改为你的每台 node 节点的IP地址。

注意：`/var/lib/kubelet` 需要手动创建。

```

###  

## kubernetes kubelet (minion) config  

#  

## The address for the info server to serve on (set to 0.0.0.0 or "" for all interfaces)  

KUBELET_ADDRESS="--address=172.20.0.113"  

#  

## The port for the info server to serve on  

#KUBELET_PORT="--port=10250"  

#  

## You may leave this blank to use the actual hostname  

KUBELET_HOSTNAME="--hostname-override=172.20.0.113"  

#  

## location of the api-server  

KUBELET_API_SERVER="--api-servers=http://172.20.0.113:8080"  

#  

## pod infrastructure container  

KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image=sz-pg-oam-docker-hub-001.tendcloud.com/library/pod-infrastructure:rhel7"  

#  

## Add your own!  

KUBELET_ARGS="--cgroup-driver=systemd --cluster-dns=10.254.0.2 --experimental-bootstrap-kubeconfig=/etc/kubernetes/bootstrap.kubeconfig --kubeconfig=/etc/kubernetes/kubelet.kubeconfig --require-kubeconfig --cert-dir=/etc/kubernetes/ssl --cluster-domain=cluster.local. --hairpin-mode promiscuous-bridge --serialize-image-pulls=false"

```

- `--address` 不能设置为 `127.0.0.1`，否则后续 Pods 访问 kubelet 的 API 接口时会失败，因为 Pods 访问的 `127.0.0.1` 指向自己而不是 kubelet；
- 如果设置了 `--hostname-override` 选项，则 `kube-proxy` 也需要设置该选项，否则会出现找不到 Node 的情况；
- `--experimental-bootstrap-kubeconfig` 指向 bootstrap kubeconfig 文件，kubelet 使用该文件中的用户名和 token 向 kube-apiserver 发送 TLS Bootstrapping 请求；
- 管理员通过了 CSR 请求后，kubelet 自动在 `--cert-dir` 目录创建证书和私

钥文件( `kubelet-client.crt` 和 `kubelet-client.key` ), 然后写入 `--kubeconfig` 文件;

- 建议在 `--kubeconfig` 配置文件中指定 `kube-apiserver` 地址, 如果未指定 `--api-servers` 选项, 则必须指定 `--require-kubeconfig` 选项后才从配置文件中读取 `kube-apiserver` 的地址, 否则 `kubelet` 启动后将找不到 `kube-apiserver`(日志中提示未找到 API Server) , `kubectl get nodes` 不会返回对应的 Node 信息;
- `--cluster-dns` 指定 `kubedns` 的 Service IP(可以先分配, 后续创建 `kubedns` 服务时指定该 IP), `--cluster-domain` 指定域名后缀, 这两个参数同时指定后才会生效;

完整 unit 见 [kubelet.service](#)

## 启动 **kublet**

```
$ systemctl daemon-reload
$ systemctl enable kubelet
$ systemctl start kubelet
$ systemctl status kubelet
```

## 通过 **kublet** 的 TLS 证书请求

`kubelet` 首次启动时向 `kube-apiserver` 发送证书签名请求, 必须通过后 `kubernetes` 系统才会将该 `Node` 加入到集群。

### 查看未授权的 CSR 请求

```
$ kubectl get csr
NAME      AGE      REQUESTOR      CONDITION
csr-2b308  4m      kubelet-bootstrap  Pending
$ kubectl get nodes
No resources found.
```

### 通过 CSR 请求

```
$ kubectl certificate approve csr-2b308
certificatesigningrequest "csr-2b308" approved
$ kubectl get nodes
NAME      STATUS    AGE     VERSION
10.64.3.7  Ready    49m    v1.6.1
```

自动生成了 kubelet kubeconfig 文件和公私钥

```
$ ls -l /etc/kubernetes/kubelet.kubeconfig
-rw----- 1 root root 2284 Apr  7 02:07 /etc/kubernetes/kubelet
.kubeconfig
$ ls -l /etc/kubernetes/ssl/kubelet*
-rw-r--r-- 1 root root 1046 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet-client.crt
-rw----- 1 root root 227 Apr  7 02:04 /etc/kubernetes/ssl/kub
elet-client.key
-rw-r--r-- 1 root root 1103 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet.crt
-rw----- 1 root root 1675 Apr  7 02:07 /etc/kubernetes/ssl/kub
elet.key
```

注意：假如你更新kubernetes的证书，只要没有更新 token.csv ，当重启kubelet后，该node就会自动加入到kubernetes集群中，而不会重新发送 certificaterequest ，也不需要在master节点上执行 kubectl certificate approve 操作。前提是不要删除node节点上的 /etc/kubernetes/ssl/kubelet\* 和 /etc/kubernetes/kubelet.kubeconfig 文件。否则kubelet启动时会提示找不到证书而失败。

## 配置 kube-proxy

创建 kube-proxy 的service配置文件

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

```
[Unit]
Description=Kubernetes Kube-Proxy Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=network.target

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

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

kube-proxy配置文件 /etc/kubernetes/proxy 。

```
###  
# kubernetes proxy config  
  
# default config should be adequate  
  
# Add your own!  
KUBE_PROXY_ARGS="--bind-address=172.20.0.113 --hostname-override  
=172.20.0.113 --kubeconfig=/etc/kubernetes/kube-proxy.kubeconfig  
--cluster-cidr=10.254.0.0/16"
```

- `--hostname-override` 参数值必须与 kubelet 的值一致，否则 kube-proxy 启动后会找不到该 Node，从而不会创建任何 iptables 规则；
- kube-proxy 根据 `--cluster-cidr` 判断集群内部和外部流量，指定 `--cluster-cidr` 或 `--masquerade-all` 选项后 kube-proxy 才会对访问 Service IP 的请求做 SNAT；
- `--kubeconfig` 指定的配置文件嵌入了 kube-apiserver 的地址、用户名、证

- 书、秘钥等请求和认证信息；
- 预定义的 RoleBinding `cluster-admin` 将User `system:kube-proxy` 与 Role `system:node-proxier` 绑定，该 Role 授予了调用 `kube-apiserver` Proxy 相关 API 的权限；

完整 unit 见 [kube-proxy.service](#)

## 启动 kube-proxy

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

## 验证测试

我们创建一个nginx的service试一下集群是否可用。

```
$ kubectl run nginx --replicas=2 --labels="run=load-balancer-example" --image=sz-pg-oam-docker-hub-001.tendcloud.com/library/nginx:1.9 --port=80
deployment "nginx" created
$ kubectl expose deployment nginx --type=NodePort --name=example-service
service "example-service" exposed
$ kubectl describe svc example-service
Name:           example-service
Namespace:      default
Labels:         run=load-balancer-example
Annotations:    <none>
Selector:       run=load-balancer-example
Type:          NodePort
IP:            10.254.62.207
Port:          <unset>   80/TCP
NodePort:       <unset>   32724/TCP
Endpoints:     172.30.60.2:80,172.30.94.2:80
Session Affinity: None
```

```
Events: <none>
$ curl "10.254.62.207:80"
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

提示：上面的测试示例中使用的nginx是我的私有镜像仓库中的镜像 sz-pg-oam-docker-hub-001.tendcloud.com/library/nginx:1.9，大家在测试过程中请换成自己的nginx镜像地址。

访问 172.20.0.113:32724 或 172.20.0.114:32724 或者 172.20.0.115:32724 都可以得到nginx的页面。



for GitBook      update 2017-05-12 16:45:40

## 安装和配置 **kubedns** 插件

官方的yaml文件目录：`kubernetes/cluster/addons/dns`。

该插件直接使用kubernetes部署，官方的配置文件中包含以下镜像：

```
gcr.io/google_containers/k8s-dns-dnsmasq-nanny-amd64:1.14.1  
gcr.io/google_containers/k8s-dns-kube-dns-amd64:1.14.1  
gcr.io/google_containers/k8s-dns-sidecar-amd64:1.14.1
```

我clone了上述镜像，上传到我的私有镜像仓库：

```
sz-pg-oam-docker-hub-001.tendcloud.com/library/k8s-dns-dnsmasq-nanny-amd64:1.14.1  
sz-pg-oam-docker-hub-001.tendcloud.com/library/k8s-dns-kube-dns-amd64:1.14.1  
sz-pg-oam-docker-hub-001.tendcloud.com/library/k8s-dns-sidecar-amd64:1.14.1
```

同时上传了一份到时速云备份：

```
index.tenxcloud.com/jimmy/k8s-dns-dnsmasq-nanny-amd64:1.14.1  
index.tenxcloud.com/jimmy/k8s-dns-kube-dns-amd64:1.14.1  
index.tenxcloud.com/jimmy/k8s-dns-sidecar-amd64:1.14.1
```

以下yaml配置文件中使用的是私有镜像仓库中的镜像。

```
kubedns-cm.yaml  
kubedns-sa.yaml  
kubedns-controller.yaml  
kubedns-svc.yaml
```

已经修改好的 yaml 文件见：[dns](#)

## 系统预定义的 **RoleBinding**

预定义的 RoleBinding `system:kube-dns` 将 `kube-system` 命名空间的 `kube-dns` ServiceAccount 与 `system:kube-dns` Role 绑定，该 Role 具有访问 kube-apiserver DNS 相关 API 的权限；

```
$ kubectl get clusterrolebindings system:kube-dns -o yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: 2017-04-11T11:20:42Z
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-dns
  resourceVersion: "58"
  selfLink: /apis/rbac.authorization.k8s.io/v1beta1/clusterrolebindingssystem%3Akube-dns
  uid: e61f4d92-1ea8-11e7-8cd7-f4e9d49f8ed0
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:kube-dns
subjects:
- kind: ServiceAccount
  name: kube-dns
  namespace: kube-system
```

`kubedns-controller.yaml` 中定义的 Pods 时使用了 `kubedns-sa.yaml` 文件定义的 `kube-dns` ServiceAccount，所以具有访问 kube-apiserver DNS 相关 API 的权限。

## 配置 **kube-dns ServiceAccount**

无需修改。

## 配置 **kube-dns** 服务

```
$ diff kubedns-svc.yaml.base kubedns-svc.yaml
30c30
<   clusterIP: __PILLAR__DNS__SERVER__
---
>   clusterIP: 10.254.0.2
```

- spec.clusterIP = 10.254.0.2，即明确指定了 kube-dns Service IP，这个 IP 需要和 kubelet 的 --cluster-dns 参数值一致；

## 配置 **kube-dns Deployment**

```
$ diff kubedns-controller.yaml.base kubedns-controller.yaml
58c58
<           image: gcr.io/google_containers/k8s-dns-kube-dns-amd64
:1.14.1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
k8s-dns-kube-dns-amd64:v1.14.1
88c88
<           - --domain=__PILLAR__DNS__DOMAIN__.
---
>           - --domain=cluster.local.
92c92
<           __PILLAR__FEDERATIONS__DOMAIN__MAP__
---
>           #__PILLAR__FEDERATIONS__DOMAIN__MAP__
110c110
<           image: gcr.io/google_containers/k8s-dns-dnsmasq-nanny-
amd64:1.14.1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
k8s-dns-dnsmasq-nanny-amd64:v1.14.1
129c129
<           - --server=/__PILLAR__DNS__DOMAIN__/127.0.0.1#10053
---
```

```
>      - --server=/cluster.local./127.0.0.1#10053
148c148
<      image: gcr.io/google_containers/k8s-dns-sidecar-amd64:
1.14.1
---
>      image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
k8s-dns-sidecar-amd64:v1.14.1
161,162c161,162
<      - --probe=kubedns,127.0.0.1:10053,kubernetes.default.s
vc.__PILLAR__DNS__DOMAIN__,5,A
<      - --probe=dnsMasq,127.0.0.1:53,kubernetes.default.svc.
__PILLAR__DNS__DOMAIN__,5,A
---
>      - --probe=kubedns,127.0.0.1:10053,kubernetes.default.s
vc.cluster.local.,5,A
>      - --probe=dnsMasq,127.0.0.1:53,kubernetes.default.svc.
cluster.local.,5,A
```

- 使用系统已经做了 RoleBinding 的 kube-dns ServiceAccount，该账户具有访问 kube-apiserver DNS 相关 API 的权限；

## 执行所有定义文件

```
$ pwd
/root/kubedns
$ ls *.yaml
kubedns-cm.yaml  kubedns-controller.yaml  kubedns-sa.yaml  kubed
ns-svc.yaml
$ kubectl create -f .
```

## 检查 **kubedns** 功能

新建一个 Deployment

```
$ cat my-nginx.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: sz-pg-oam-docker-hub-001.tendcloud.com/library/nginx:1.9
      ports:
        - containerPort: 80
$ kubectl create -f my-nginx.yaml
```

Export 该 Deployment, 生成 my-nginx 服务

```
$ kubectl expose deploy my-nginx
$ kubectl get services --all-namespaces |grep my-nginx
default     my-nginx     10.254.179.239   <none>       80/TCP
                           42m
```

创建另一个 Pod，查看 /etc/resolv.conf 是否包含 kubelet 配置的 --cluster-dns 和 --cluster-domain，是否能够将服务 my-nginx 解析到 Cluster IP 10.254.179.239。

```
$ kubectl create -f nginx-pod.yaml
$ kubectl exec nginx -i -t -- /bin/bash
root@nginx:/# cat /etc/resolv.conf
nameserver 10.254.0.2
search default.svc.cluster.local. svc.cluster.local. cluster.local.
al. tendcloud.com
options ndots:5

root@nginx:/# ping my-nginx
PING my-nginx.default.svc.cluster.local (10.254.179.239): 56 dat
a bytes
76 bytes from 119.147.223.109: Destination Net Unreachable
^C--- my-nginx.default.svc.cluster.local ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss

root@nginx:/# ping kubernetes
PING kubernetes.default.svc.cluster.local (10.254.0.1): 56 data
bytes
^C--- kubernetes.default.svc.cluster.local ping statistics ---
11 packets transmitted, 0 packets received, 100% packet loss

root@nginx:/# ping kube-dns.kube-system.svc.cluster.local
PING kube-dns.kube-system.svc.cluster.local (10.254.0.2): 56 dat
a bytes
^C--- kube-dns.kube-system.svc.cluster.local ping statistics ---
6 packets transmitted, 0 packets received, 100% packet loss
```

从结果来看，service名称可以正常解析。

for GitBook update 2017-05-12 16:45:40

## 配置和安装 dashboard

官方文件目录：`kubernetes/cluster/addons/dashboard`

我们使用的文件

```
$ ls *.yaml
dashboard-controller.yaml  dashboard-service.yaml  dashboard-rbac
.yaml
```

已经修改好的 yaml 文件见：[dashboard](#)

由于 `kube-apiserver` 启用了 `RBAC` 授权，而官方源码目录的 `dashboard-controller.yaml` 没有定义授权的 `ServiceAccount`，所以后续访问 `kube-apiserver` 的 API 时会被拒绝，web 中提示：

```
Forbidden (403)

User "system:serviceaccount:kube-system:default" cannot list job
s.batch in the namespace "default". (get jobs.batch)
```

增加了一个 `dashboard-rbac.yaml` 文件，定义一个名为 `dashboard` 的 `ServiceAccount`，然后将它和 Cluster Role view 绑定。

## 配置dashboard-service

```
$ diff dashboard-service.yaml.orig dashboard-service.yaml
10a11
>   type: NodePort
```

- 指定端口类型为 `NodePort`，这样外界可以通过地址 `nodeIP:nodePort` 访问 `dashboard`；

## 配置dashboard-controller

```
$ diff dashboard-controller.yaml.orig dashboard-controller.yaml
23c23
<           image: gcr.io/google_containers/kubernetes-dashboard-a
md64:v1.6.0
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
kubernetes-dashboard-amd64:v1.6.0
```

## 执行所有定义文件

```
$ pwd
/root/kubernetes/cluster/addons/dashboard
$ ls *.yaml
dashboard-controller.yaml  dashboard-service.yaml
$ kubectl create -f .
service "kubernetes-dashboard" created
deployment "kubernetes-dashboard" created
```

## 检查执行结果

### 查看分配的 NodePort

```
$ kubectl get services kubernetes-dashboard -n kube-system
NAME                  CLUSTER-IP      EXTERNAL-IP    PORT(S)
AGE
kubernetes-dashboard   10.254.224.130  <nodes>        80:30312/T
CP      25s
```

- NodePort 30312映射到 dashboard pod 80端口；

### 检查 controller

```
$ kubectl get deployment kubernetes-dashboard -n kube-system
NAME                  DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
E   AGE
kubernetes-dashboard   1         1         1           1
3m
$ kubectl get pods -n kube-system | grep dashboard
kubernetes-dashboard-1339745653-pmn6z   1/1       Running   0
4m
```

## 访问dashboard

有以下三种方式：

- kubernetes-dashboard 服务暴露了 NodePort，可以使用 `http://NodeIP:nodePort` 地址访问 dashboard；
- 通过 kube-apiserver 访问 dashboard（https 6443 端口和 http 8080 端口方式）；
- 通过 kubectl proxy 访问 dashboard：

## 通过 kubectl proxy 访问 dashboard

启动代理

```
$ kubectl proxy --address='172.20.0.113' --port=8086 --accept-hosts='^*$'
Starting to serve on 172.20.0.113:8086
```

- 需要指定 `--accept-hosts` 选项，否则浏览器访问 dashboard 页面时提示“Unauthorized”；

浏览器访问 URL：`http://172.20.0.113:8086/ui` 自动跳转到：`http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard/#/workload?namespace=default`

## 通过 kube-apiserver 访问 dashboard

### 获取集群服务地址列表

```
$ kubectl cluster-info  
Kubernetes master is running at https://172.20.0.113:6443  
KubeDNS is running at https://172.20.0.113:6443/api/v1/proxy/nam  
espaces/kube-system/services/kube-dns  
kubernetes-dashboard is running at https://172.20.0.113:6443/api  
/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
```

### 浏览器访问

URL : `https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-  
system/services/kubernetes-dashboard` (浏览器会提示证书验证，因为通过  
加密通道，以改方式访问的话，需要提前导入证书到你的计算机中)。这是我当时  
在这遇到的坑：[通过 kube-apiserver 访问dashboard，提示User  
"system:anonymous" cannot proxy services in the namespace "kube-system".  
#5](#)，已经解决。

### 导入证书

将生成的 `admin.pem` 证书转换格式

```
openssl pkcs12 -export -in admin.pem -out admin.p12 -inkey admi  
n-key.pem
```

将生成的 `admin.p12` 证书导入的你的电脑，导出的时候记住你设置的密码，导入  
的时候还要用到。

如果你不想使用 **https** 的话，可以直接访问 `insecure port 8080` 端  
口：`http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-  
system/services/kubernetes-dashboard`

## 1.8 安装dashboard插件

The screenshot shows the Kubernetes dashboard interface. At the top, there's a blue header bar with the title 'kubernetes' and a 'CREATE' button. Below the header, the navigation path 'Admin > Namespaces > kube-system' is displayed. On the left, a sidebar menu lists 'Namespaces' (which is currently selected), 'Nodes', 'Persistent Volumes', 'Storage Classes', 'Namespace' (with 'default' selected), 'Workloads', 'Deployments', 'Replica Sets', 'Replication Controllers', 'Daemon Sets', 'Stateful Sets', 'Jobs', 'Pods', 'Services and discovery', 'Services', 'Ingresses', 'Storage', 'Persistent Volume Claims', 'Config', and 'Secrets'. The main content area is divided into two tabs: 'Details' and 'Events'. The 'Details' tab shows the namespace name 'kube-system', creation time '2017-04-11T11:20', and status 'Active'. The 'Events' tab lists several log entries:

Message	Source	Sub-object	Count	First seen	Last seen
Killing container with id docker://c857a23eb359fb5f46c080b0404f41731d7be47d9729eb1e5ae9d8b2be123d5f:Need to kill Pod	kubelet 172.20.0.114	spec.containers(kubernetes-dashboard)	1	2017-04-12T06:51 UTC	2017-04-12T06:51 UTC
Deleted pod: kubernetes-dashboard-1752429380-7vk0t	replicaset-controller	-	1	2017-04-12T06:51 UTC	2017-04-12T06:51 UTC
Successfully assigned kubernetes-dashboard-3966630548-61b48 to 172.20.0.113	default-scheduler	-	1	2017-04-12T06:56 UTC	2017-04-12T06:56 UTC
Container image "sz-pg-pam-docker-hub-001.tendcloud.com/library/kubernetes-dashboard-amd64:v1.6.0" already present on machine	kubelet 172.20.0.113	spec.containers(kubernetes-dashboard)	1	2017-04-12T06:57 UTC	2017-04-12T06:57 UTC
Created container with id c36e4cc8e1108805a34affd872449442a3bf628466b8ea2da3c99caf1d7cec23	kubelet 172.20.0.113	spec.containers(kubernetes-dashboard)	1	2017-04-12T06:57 UTC	2017-04-12T06:57 UTC
Started container with id c36e4cc8e1108805a34affd872449442a3bf628466b8ea2da3c99caf1d7cec23	kubelet 172.20.0.113	spec.containers(kubernetes-dashboard)	1	2017-04-12T06:57 UTC	2017-04-12T06:57 UTC
⚠ Error creating pods "kubernetes-dashboard-3966630548-": Is forbidden: service account kubernetes-dashboard was replicated controller	replicaset-controller	-	4	2017-04-12T06:56 UTC	2017-04-12T06:56 UTC

Figure: kubernetes-dashboard

由于缺少 Heapster 插件，当前 dashboard 不能展示 Pod、Nodes 的 CPU、内存等 metric 图形。

for GitBook update 2017-05-12 16:45:40

## 配置和安装 Heapster

到 [heapster release](#) 页面 下载最新版本的 heapster。

```
$ wget https://github.com/kubernetes/heapster/archive/v1.3.0.zip  
$ unzip v1.3.0.zip  
$ mv v1.3.0.zip heapster-1.3.0
```

文件目录： heapster-1.3.0/deploy/kube-config/influxdb

```
$ cd heapster-1.3.0/deploy/kube-config/influxdb  
$ ls *.yaml  
grafana-deployment.yaml  grafana-service.yaml  heapster-deployment.yaml  
heapster-service.yaml  influxdb-deployment.yaml  influxdb-service.yaml  
heapster-rbac.yaml
```

我们自己创建了heapster的rbac配置 `heapster-rbac.yaml`。

已经修改好的 yaml 文件见：[heapster](#)

## 配置 grafana-deployment

```
$ diff grafana-deployment.yaml.orig grafana-deployment.yaml
16c16
<           image: gcr.io/google_containers/heapster-grafana-amd64
:v4.0.2
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
heapster-grafana-amd64:v4.0.2
40,41c40,41
<           # value: /api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/
<           value: /
---
>           value: /api/v1/proxy/namespaces/kube-system/services
/monitoring-grafana/
>           #value: /
```

- 如果后续使用 kube-apiserver 或者 kubectl proxy 访问 grafana dashboard，则必须将 `GF_SERVER_ROOT_URL` 设置为 `/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/`，否则后续访问grafana时访问时提示找不到 `http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/api/dashboards/home` 页面；

## 配置 heapster-deployment

```
$ diff heapster-deployment.yaml.orig heapster-deployment.yaml
16c16
<           image: gcr.io/google_containers/heapster-amd64:v1.3.0-
beta.1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
heapster-amd64:v1.3.0-beta.1
```

## 配置 influxdb-deployment

influxdb 官方建议使用命令行或 HTTP API 接口来查询数据库，从 v1.1.0 版本开始默认关闭 admin UI，将在后续版本中移除 admin UI 插件。

开启镜像中 admin UI 的办法如下：先导出镜像中的 influxdb 配置文件，开启 admin 插件后，再将配置文件内容写入 ConfigMap，最后挂载到镜像中，达到覆盖原始配置的目的：

注意：manifests 目录已经提供了 [修改后的 ConfigMap 定义文件](#)

```
$ # 导出镜像中的 influxdb 配置文件
$ docker run --rm --entrypoint 'cat' -ti lvanneo/heapster-influxdb-amd64:v1.1.1 /etc/config.toml >config.toml.orig
$ cp config.toml.orig config.toml
$ # 修改：启用 admin 接口
$ vim config.toml
$ diff config.toml.orig config.toml
35c35
<   enabled = false
---
>   enabled = true
$ # 将修改后的配置写入到 ConfigMap 对象中
$ kubectl create configmap influxdb-config --from-file=config.toml -n kube-system
configmap "influxdb-config" created
$ # 将 ConfigMap 中的配置文件挂载到 Pod 中，达到覆盖原始配置的目的
$ diff influxdb-deployment.yaml.orig influxdb-deployment.yaml
16c16
<           image: grc.io/google_containers/heapster-influxdb-amd64:v1.1.1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/heapster-influxdb-amd64:v1.1.1
19a20,21
>           - mountPath: /etc/
>             name: influxdb-config
22a25,27
>           - name: influxdb-config
>             configMap:
>               name: influxdb-config
```

## 配置 monitoring-influxdb Service

```
$ diff influxdb-service.yaml.orig influxdb-service.yaml
12a13
>   type: NodePort
15a17,20
>     name: http
>     - port: 8083
>       targetPort: 8083
>     name: admin
```

- 定义端口类型为 NodePort，额外增加了 admin 端口映射，用于后续浏览器访问 influxdb 的 admin UI 界面；

## 执行所有定义文件

```
$ pwd
/root/heapster-1.3.0/deploy/kube-config/influxdb
$ ls *.yaml
grafana-service.yaml      heapster-rbac.yaml      influxdb-cm.yaml
influxdb-service.yaml
grafana-deployment.yaml  heapster-deployment.yaml  heapster-service.yaml
influxdb-deployment.yaml
$ kubectl create -f .
deployment "monitoring-grafana" created
service "monitoring-grafana" created
deployment "heapster" created
serviceaccount "heapster" created
clusterrolebinding "heapster" created
service "heapster" created
configmap "influxdb-config" created
deployment "monitoring-influxdb" created
service "monitoring-influxdb" created
```

## 检查执行结果

### 检查 Deployment

```
$ kubectl get deployments -n kube-system | grep -E 'heapster|monitoring'
heapster              1           1           1           1
  2m
monitoring-grafana   1           1           1           1
  2m
monitoring-influxdb  1           1           1           1
  2m
```

### 检查 Pods

```
$ kubectl get pods -n kube-system | grep -E 'heapster|monitoring'
heapster-110704576-gpg8v          1/1       Running    0
  2m
monitoring-grafana-2861879979-9z89f 1/1       Running    0
  2m
monitoring-influxdb-1411048194-lzrpc 1/1       Running    0
  2m
```

检查 kubernets dashboard 界面，看是显示各 Nodes、Pods 的 CPU、内存、负载等利用率曲线图；

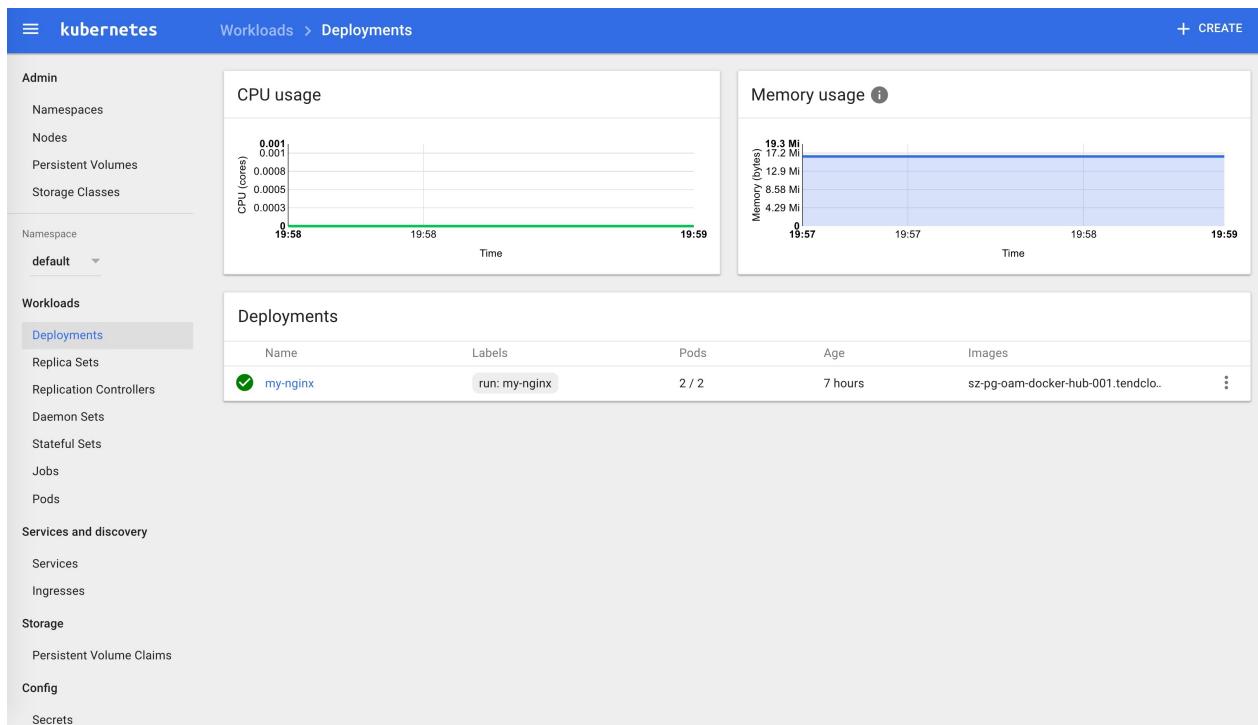


Figure: dashboard-heapster

## 访问 grafana

1. 通过 kube-apiserver 访问：

获取 monitoring-grafana 服务 URL

```
$ kubectl cluster-info
Kubernetes master is running at https://172.20.0.113:6443
Heapster is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/heapster
KubeDNS is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard
monitoring-grafana is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
monitoring-influxdb is running at https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

浏览器访问 URL :

```
http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
```

2. 通过 kubectl proxy 访问：

创建代理

```
$ kubectl proxy --address='172.20.0.113' --port=8086 --accept-hosts='^*$'
Starting to serve on 172.20.0.113:8086
```

浏览器访问

```
URL : http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
```

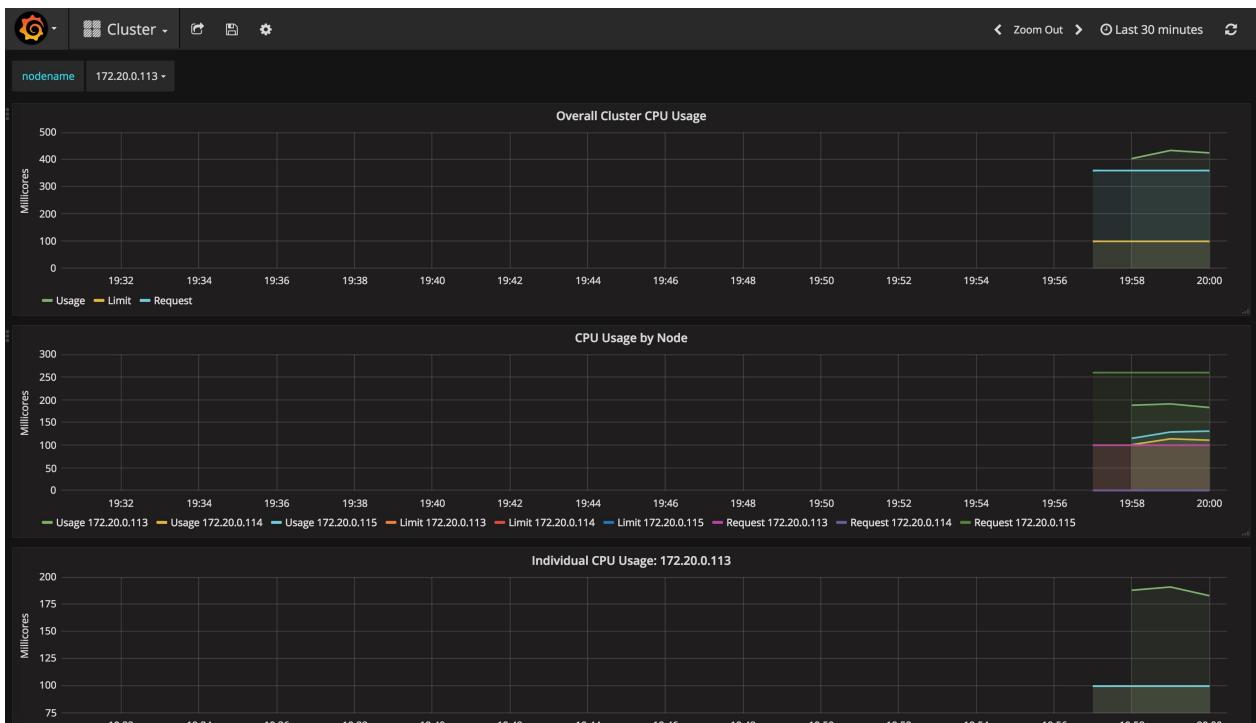


Figure: grafana

## 访问 influxdb admin UI

获取 influxdb http 8086 映射的 NodePort

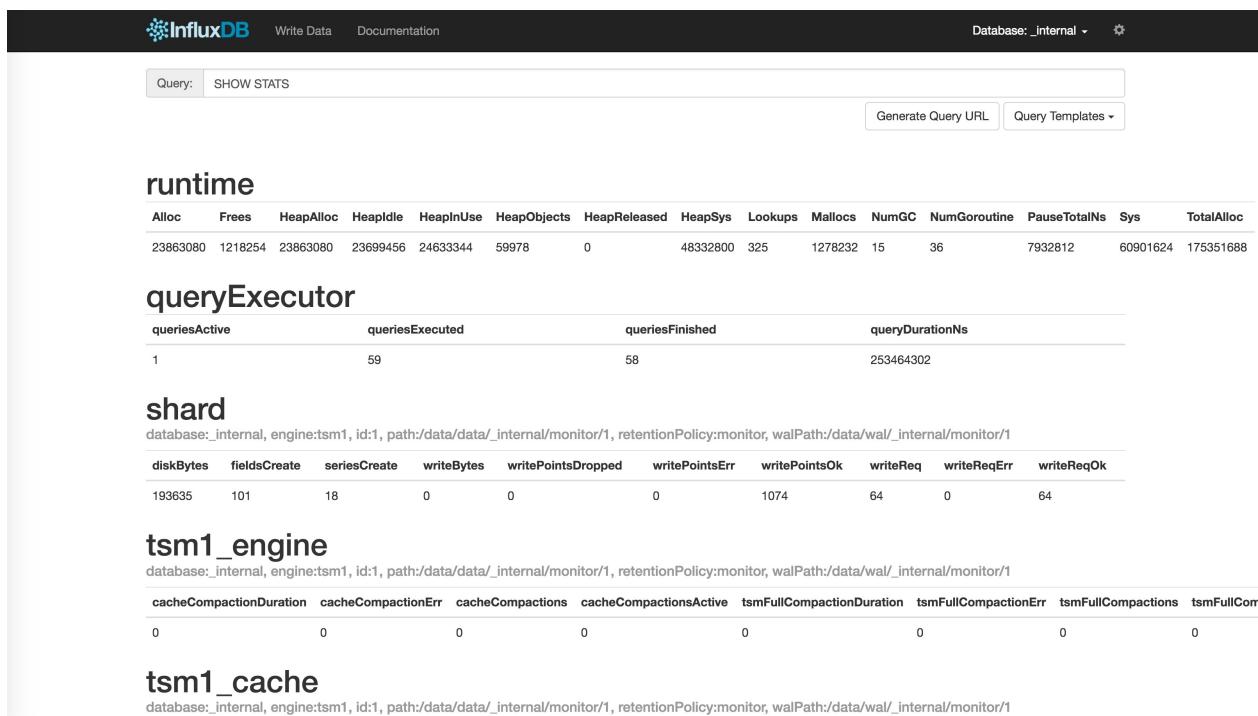
```
$ kubectl get svc -n kube-system|grep influxdb
monitoring-influxdb      10.254.22.46      <nodes>          8086:32299/
TCP, 8083:30269/TCP    9m
```

通过 kube-apiserver 的非安全端口访问 influxdb 的 admin UI 界面：

```
http://172.20.0.113:8080/api/v1/proxy/namespaces/kube-
system/services/monitoring-influxdb:8083/
```

在页面的“Connection Settings”的 Host 中输入 node IP，Port 中输入 8086 映射的 nodePort 如上面的 32299，点击“Save”即可（我的集群中的地址是 172.20.0.113:32299）：

## 1.9 安装heapster插件



*Figure: kubernetes-influxdb-heapster*

for GitBook

update 2017-05-12 16:45:40

## 配置和安装 EFK

官方文件目录： cluster/addons/fluentd-elasticsearch

```
$ ls *.yaml
es-controller.yaml  es-service.yaml  fluentd-es-ds.yaml  kibana-
controller.yaml  kibana-service.yaml  efk-rbac.yaml
```

同样EFK服务也需要一个 efk-rbac.yaml 文件，配置serviceaccount为 efk。

已经修改好的 yaml 文件见：[EFK](#)

### 配置 es-controller.yaml

```
$ diff es-controller.yaml.orig es-controller.yaml
24c24
<      - image: gcr.io/google_containers/elasticsearch:v2.4.1-2
---
>      - image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
elasticsearch:v2.4.1-2
```

### 配置 es-service.yaml

无需配置；

### 配置 fluentd-es-ds.yaml

```
$ diff fluentd-es-ds.yaml.orig fluentd-es-ds.yaml
26c26
<           image: gcr.io/google_containers/fluentd-elasticsearch:
1.22
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
fluentd-elasticsearch:1.22
```

## 配置 kibana-controller.yaml

```
$ diff kibana-controller.yaml.orig kibana-controller.yaml
22c22
<           image: gcr.io/google_containers/kibana:v4.6.1-1
---
>           image: sz-pg-oam-docker-hub-001.tendcloud.com/library/
kibana:v4.6.1-1
```

## 给 Node 设置标签

定义 DaemonSet fluentd-es-v1.22 时设置了 nodeSelector  
beta.kubernetes.io/fluentd-ds-ready=true，所以需要在期望运行 fluentd 的 Node 上设置该标签；

```
$ kubectl get nodes
NAME      STATUS     AGE      VERSION
172.20.0.113   Ready     1d      v1.6.0

$ kubectl label nodes 172.20.0.113 beta.kubernetes.io/fluentd-ds
-ready=true
node "172.20.0.113" labeled
```

给其他两台node打上同样的标签。

## 执行定义文件

```
$ kubectl create -f .
serviceaccount "efk" created
clusterrolebinding "efk" created
replicationcontroller "elasticsearch-logging-v1" created
service "elasticsearch-logging" created
daemonset "fluentd-es-v1.22" created
deployment "kibana-logging" created
service "kibana-logging" created
```

## 检查执行结果

```
$ kubectl get deployment -n kube-system|grep kibana
kibana-logging           1           1           1           1
                           2m

$ kubectl get pods -n kube-system|grep -E 'elasticsearch|fluentd|kibana'
elasticsearch-logging-v1-mlstp          1/1       Running   0
                                         1m
elasticsearch-logging-v1-nfbbf          1/1       Running   0
                                         1m
fluentd-es-v1.22-31sm0                1/1       Running   0
                                         1m
fluentd-es-v1.22-bpgqs               1/1       Running   0
                                         1m
fluentd-es-v1.22-qmn7h                1/1       Running   0
                                         1m
kibana-logging-1432287342-0gdng      1/1       Running   0
                                         1m

$ kubectl get service -n kube-system|grep -E 'elasticsearch|kibana'
elasticsearch-logging    10.254.77.62    <none>        9200/TCP
                           2m
kibana-logging          10.254.8.113     <none>        5601/TCP
                           2m
```

kibana Pod 第一次启动时会用较长时间(**10-20分钟**)来优化和 Cache 状态页面，可以 tailf 该 Pod 的日志观察进度：

```
$ kubectl logs kibana-logging-1432287342-0gdng -n kube-system -f
ELASTICSEARCH_URL=http://elasticsearch-logging:9200
server.basePath: /api/v1/proxy/namespaces/kube-system/services/k
ibana-logging
{"type":"log","@timestamp":"2017-04-12T13:08:06Z","tags":["info",
"optimize"],"pid":7,"message":"Optimizing and caching bundles fo
r kibana and statusPage. This may take a few minutes"}
{"type":"log","@timestamp":"2017-04-12T13:18:17Z","tags":["info",
"optimize"],"pid":7,"message":"Optimization of bundles for kiban
a and statusPage complete in 610.40 seconds"}
{"type":"log","@timestamp":"2017-04-12T13:18:17Z","tags":["statu
s","plugin:kibana@1.0.0","info"],"pid":7,"state":"green","messag
e":"Status changed from uninitialized to green - Ready","prevSta
te":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:18Z","tags":["statu
s","plugin:elasticsearch@1.0.0","info"],"pid":7,"state":"yellow",
"message":"Status changed from uninitialized to yellow - Waiting
 for Elasticsearch","prevState":"uninitialized","prevMsg":"unini
tialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:kbn_vislib_vis_types@1.0.0","info"],"pid":7,"state":"
green","message":"Status changed from uninitialized to green - R
eady","prevState":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:markdown_vis@1.0.0","info"],"pid":7,"state":"green",
"message":"Status changed from uninitialized to green - Ready","p
revState":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:metric_vis@1.0.0","info"],"pid":7,"state":"green","me
ssage":"Status changed from uninitialized to green - Ready","pre
vState":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:spyModes@1.0.0","info"],"pid":7,"state":"green","mess
age":"Status changed from uninitialized to green - Ready","prevS
tate":"uninitialized","prevMsg":"uninitialized"}
 {"type":"log","@timestamp":"2017-04-12T13:18:19Z","tags":["statu
s","plugin:statusPage@1.0.0","info"],"pid":7,"state":"green","me
```

```
ssage": "Status changed from uninitialized to green - Ready", "prevState": "uninitialized", "prevMsg": "uninitialized"}  
{"type": "log", "@timestamp": "2017-04-12T13:18:19Z", "tags": ["status", "plugin:table_vis@1.0.0", "info"], "pid": 7, "state": "green", "message": "Status changed from uninitialized to green - Ready", "prevState": "uninitialized", "prevMsg": "uninitialized"}  
{"type": "log", "@timestamp": "2017-04-12T13:18:19Z", "tags": ["listing", "info"], "pid": 7, "message": "Server running at http://0.0.0.0:5601"}  
{"type": "log", "@timestamp": "2017-04-12T13:18:24Z", "tags": ["status", "plugin:elasticsearch@1.0.0", "info"], "pid": 7, "state": "yellow", "message": "Status changed from yellow to yellow - No existing Kibana index found", "prevState": "yellow", "prevMsg": "Waiting for Elasticsearch"}  
{"type": "log", "@timestamp": "2017-04-12T13:18:29Z", "tags": ["status", "plugin:elasticsearch@1.0.0", "info"], "pid": 7, "state": "green", "message": "Status changed from yellow to green - Kibana index ready", "prevState": "yellow", "prevMsg": "No existing Kibana index found"}
```

## 访问 kibana

1. 通过 kube-apiserver 访问：

获取 monitoring-grafana 服务 URL

```
$ kubectl cluster-info
Kubernetes master is running at https://172.20.0.113:6443
Elasticsearch is running at https://172.20.0.113:6443/api/v
1/proxy/namespaces/kube-system/services/elasticsearch-logging
Heapster is running at https://172.20.0.113:6443/api/v1/pro
xy/namespaces/kube-system/services/heapster
Kibana is running at https://172.20.0.113:6443/api/v1/proxy
/namespaces/kube-system/services/kibana-logging
KubeDNS is running at https://172.20.0.113:6443/api/v1/prox
y/namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at https://172.20.0.113:644
3/api/v1/proxy/namespaces/kube-system/services/kubernetes-da
shboard
monitoring-grafana is running at https://172.20.0.113:6443/
api/v1/proxy/namespaces/kube-system/services/monitoring-graf
ana
monitoring-influxdb is running at https://172.20.0.113:6443
/api/v1/proxy/namespaces/kube-system/services/monitoring-inf
luxdb
```

浏览器访问 URL :

```
https://172.20.0.113:6443/api/v1/proxy/namespaces/kube-
system/services/kibana-logging/app/kibana
```

2. 通过 kubectl proxy 访问：

创建代理

```
$ kubectl proxy --address='172.20.0.113' --port=8086 --acce
pt-hosts='^*$'
Starting to serve on 172.20.0.113:8086
```

浏览器访问

```
URL : http://172.20.0.113:8086/api/v1/proxy/namespaces/kube-
system/services/kibana-logging
```

在 Settings -> Indices 页面创建一个 index (相当于 mysql 中的一个 database) , 选中 Index contains time-based events , 使用默认的 logstash-\* pattern , 点击 Create ;

可能遇到的问题

如果你在这里发现Create按钮是灰色的无法点击，且Time-filed name中没有选项，fluentd要读取 /var/log/containers/ 目录下的log日志，这些日志是从 /var/lib/docker/containers/\${CONTAINER\_ID}/\${CONTAINER\_ID}-json.log 链接过来的，查看你的docker配置， –log-dirver 需要设置为json-file格式，默认的可能是journald，参考[docker logging](#))。

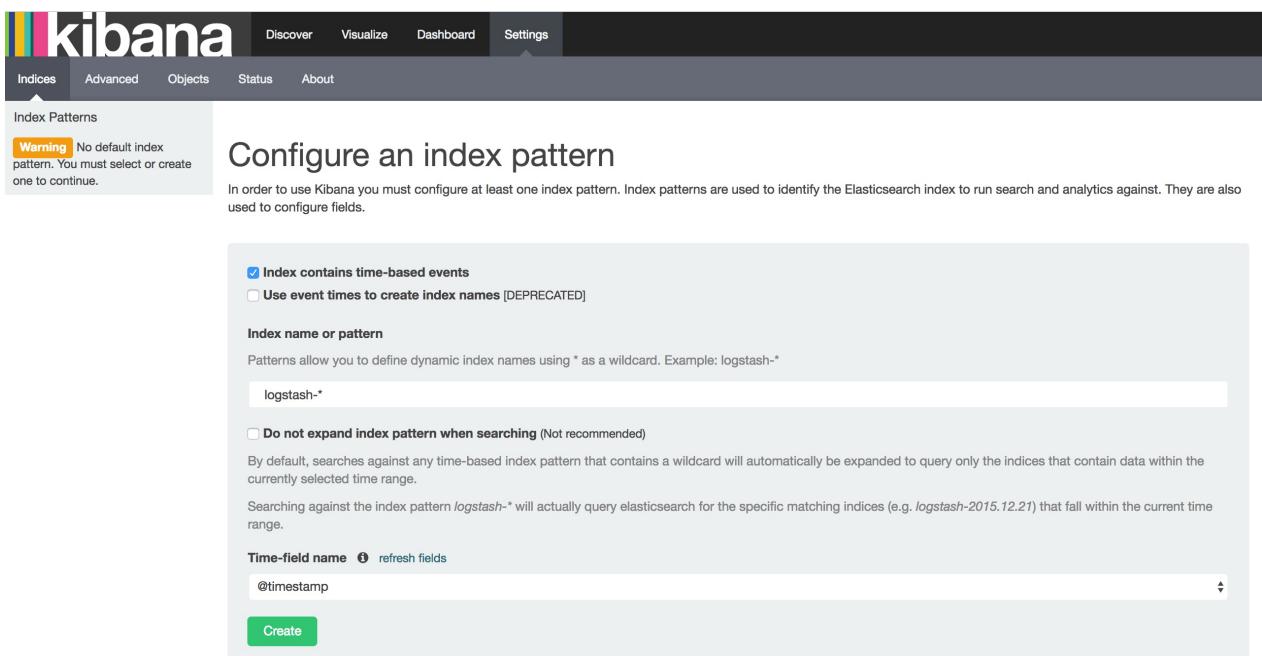
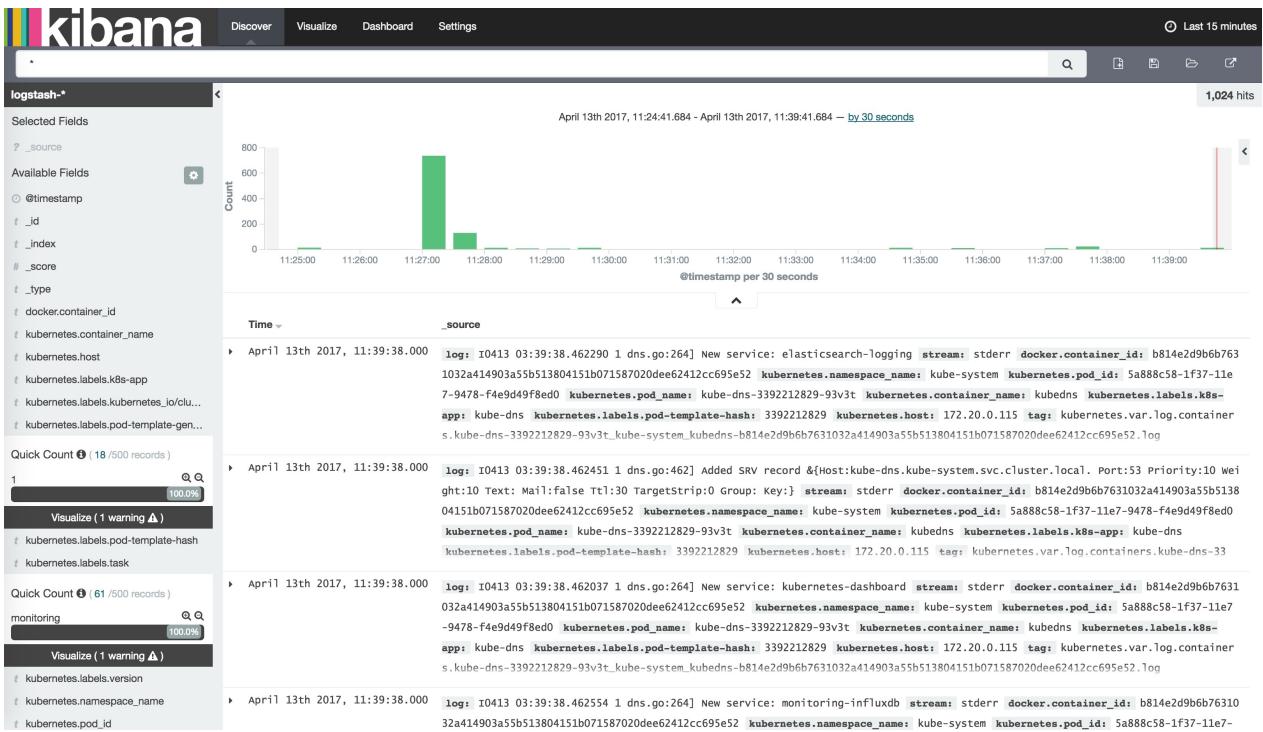


Figure: es-setting

创建Index后，可以在 Discover 下看到 ElasticSearch logging 中汇聚的日志；

## 1.10 安装EFK插件



# for GitBook

update 2017-05-12 16:45:40

### 前言

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

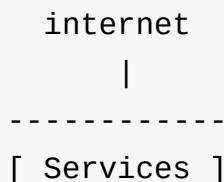
#### 术语

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

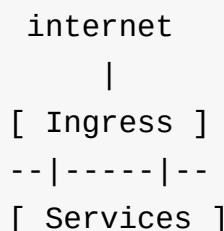
- 节点：Kubernetes集群中的一台物理机或者虚拟机。
- 集群：位于Internet防火墙后的节点，这是Kubernetes管理的主要计算资源。
- 边界路由器：为集群强制执行防火墙策略的路由器。这可能是由云提供商或物理硬件管理的网关。
- 集群网络：一组逻辑或物理链接，可根据Kubernetes[网络模型](#)实现群集内的通信。集群网络的实现包括Overlay模型的[flannel](#) 和基于SDN的[OVS](#)。
- 服务：使用标签选择器标识一组pod成为的Kubernetes[服务](#)。除非另有说明，否则服务假定在集群网络内仅可通过虚拟IP访问。

### 什么是Ingress？

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



Ingress是授权入站连接到达集群服务的规则集合。



## 2.1 Ingress解析

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

### 先决条件

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

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

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

## Ingress Resource

最简化的Ingress配置：

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

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

### 配置说明

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

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

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

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

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

## Ingress controllers

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

## 在你开始前

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

## Ingress类型

## 单Service Ingress

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

ingress.yaml定义文件：

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

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

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

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

## 简单展开

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

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

你需要一个这样的ingress：

## 2.1 Ingress解析

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

使用 `kubectl create -f` 创建完ingress后：

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

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

## 基于名称的虚拟主机

Name-based的虚拟主机在同一个IP地址下拥有多个主机名。

## 2.1 Ingress解析

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

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

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

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

## TLS

你可以通过指定包含TLS私钥和证书的[secret](#)来加密Ingress。目前，Ingress仅支持单个TLS端口443，并假定TLS termination。如果Ingress中的TLS配置部分指定了不同的主机，则它们将根据通过SNI TLS扩展指定的主机名（假如Ingress

## 2.1 Ingress解析

controller支持SNI) 在多个相同端口上进行复用。 TLS secret中必须包含名为 `tls.crt` 和 `tls.key` 的密钥，这里面包含了用于TLS的证书和私钥，例如：

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

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

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

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

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

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

## 更新Ingress

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

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

这会弹出一个包含已有的yaml文件的编辑器，修改它，增加新的Host配置。

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

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

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

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

## 跨可用域故障

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

## 未来计划

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

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

## 替代方案

你可以通过很多种方式暴露service而不必直接使用ingress：

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

### 参考

[Kubernetes Ingress Resource](#)

[使用NGINX Plus负载均衡Kubernetes服务](#)

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

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

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

[Kubernetes Ingress Controller-Traefik](#)

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

for GitBook      update 2017-05-12 16:45:40

# Kubernetes traefik ingress安装

## Ingress简介

如果你还不了解，ingress是什么，可以先看下我翻译的Kubernetes官网上ingress的介绍[Kubernetes Ingress解析](#)。

### 理解Ingress

简单的说，ingress就是从kubernetes集群外访问集群的入口，将用户的URL请求转发到不同的service上。Ingress相当于nginx、apache等负载均衡方向代理服务器，其中还包括规则定义，即URL的路由信息，路由信息得的刷新由[Ingress controller](#)来提供。

### 理解Ingress Controller

Ingress Controller 实质上可以理解为是个监视器，Ingress Controller 通过不断地跟kubernetes API 打交道，实时的感知后端 service、pod 等变化，比如新增和减少 pod，service 增加与减少等；当得到这些变化信息后，Ingress Controller 再结合下文的 Ingress 生成配置，然后更新反向代理负载均衡器，并刷新其配置，达到服务发现的作用。

## 部署Traefik

### 介绍traefik

[Traefik](#)是一款开源的反向代理与负载均衡工具。它最大的优点是能够与常见的微服务系统直接整合，可以实现自动化动态配置。目前支持Docker, Swarm, Mesos/Marathon, Mesos, Kubernetes, Consul, Etcd, Zookeeper, BoltDB, Rest API 等等后端模型。

以下配置文件可以在[kubernetes-handbook](#)GitHub仓库中的[manifests/traefik-ingress/](#)目录下找到。

### 创建ingress-rbac.yaml

将用于service account验证。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: ingress
  namespace: kube-system

---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: ingress
subjects:
- kind: ServiceAccount
  name: ingress
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

创建名为 **traefik-ingress** 的**ingress**，文件名**traefik.yaml**

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ingress
spec:
  rules:
    - host: traefik.nginx.io
      http:
        paths:
          - path: /
            backend:
              serviceName: my-nginx
              servicePort: 80
    - host: traefik.frontend.io
      http:
        paths:
          - path: /
            backend:
              serviceName: frontend
              servicePort: 80
```

这其中的 `backend` 中要配置 default namespace 中启动的 `service` 名字。`path` 就是 URL 地址后的路径，如 `traefik.frontend.io/path`，`service` 将会接受 `path` 这个路径，`host` 最好使用 `service-name.file1.file2.domain-name` 这种类似主机名称的命名方式，方便区分服务。

根据你自己环境中部署的 `service` 的名字和端口自行修改，有新 `service` 增加时，修改该文件后可以使用 `kubectl replace -f traefik.yaml` 来更新。

我们现在集群中已经有两个 `service` 了，一个是 `nginx`，另一个是官方的 `guestbook` 例子。

创建 **Depeloyment**

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: traefik-ingress-lb
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      terminationGracePeriodSeconds: 60
      hostNetwork: true
      restartPolicy: Always
      serviceAccountName: ingress
      containers:
        - image: traefik
          name: traefik-ingress-lb
          resources:
            limits:
              cpu: 200m
              memory: 30Mi
            requests:
              cpu: 100m
              memory: 20Mi
          ports:
            - name: http
              containerPort: 80
              hostPort: 80
            - name: admin
              containerPort: 8580
              hostPort: 8580
          args:
            - --web
            - --web.address=:8580
            - --kubernetes
```

## 2.2 安装Traefik ingress

注意我们这里用的是Deploy类型，没有限定该pod运行在哪个主机上。Traefik的端口是8580。

### Traefik UI

```
apiVersion: v1
kind: Service
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  selector:
    k8s-app: traefik-ingress-lb
  ports:
    - name: web
      port: 80
      targetPort: 8580
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-web-ui
  namespace: kube-system
spec:
  rules:
    - host: traefik-ui.local
      http:
        paths:
          - path: /
            backend:
              serviceName: traefik-web-ui
              servicePort: web
```

配置完成后就可以启动traefik ingress了。

```
kubectl create -f .
```

我查看到traefik的pod在 172.20.0.115 这台节点上启动了。

## 2.2 安装 Traefik ingress

访问该地址 `http://172.20.0.115:8580/` 将可以看到dashboard。

The screenshot shows the Kubernetes dashboard interface with the 'Traefik' section selected. It displays four panels: 1) A summary panel for traefik-ui.local/ showing one rule (PathPrefix: /) and one backend (Host: traefik-ui.local). 2) A detailed panel for traefik-ui.local/ showing a single server entry (traefik-ingress-lb-4237248072-4009r) mapped to URL http://172.20.0.115:8580 with weight 1, using wrr load balancing. 3) A summary panel for traefik.frontend.io/ showing one rule (PathPrefix: /) and one backend (Host: traefik.frontend.io). 4) A detailed panel for traefik.frontend.io/ showing three servers (frontend-1289468719-6l4v7, frontend-1289468719-sfkvb, frontend-1289468719-vg4zz) mapped to URLs http://172.30.60.11:80, http://172.30.71.5:80, and http://172.30.94.9:80 respectively, all with weight 1 using wrr load balancing.

Route	Rule
/	PathPrefix: /
traefik-ui.local	Host: traefik-ui.local

Server	URL	Weight
traefik-ingress-lb-4237248072-4009r	http://172.20.0.115:8580	1

Route	Rule
/	PathPrefix: /
traefik.frontend.io	Host: traefik.frontend.io

Server	URL	Weight
frontend-1289468719-6l4v7	http://172.30.60.11:80	1
frontend-1289468719-sfkvb	http://172.30.71.5:80	1
frontend-1289468719-vg4zz	http://172.30.94.9:80	1

Route	Rule
/	PathPrefix: /
traefik.nginx.io	Host: traefik.nginx.io

Server	URL	Weight
my-nginx-2096504489-1jg9c	http://172.30.60.5:80	1
my-nginx-2096504489-1vl95	http://172.30.94.6:80	1

Figure: kubernetes-dashboard

左侧黄色部分部分列出的是所有的rule，右侧绿色部分是所有的backend。

## 测试

在集群的任意一个节点上执行。假如现在我要访问nginx的"/"路径。

```
$ curl -H Host:traefik.nginx.io http://172.20.0.115/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

如果你需要在kubernetes集群以外访问就需要设置DNS，或者修改本机的hosts文件。

在其中加入：

```
172.20.0.115 traefik.nginx.io
172.20.0.115 traefik.frontend.io
```

所有访问这些地址的流量都会发送给172.20.0.115这台主机，就是我们启动traefik的主机。

## 2.2 安装Traefik ingress

Traefik会解析http请求header里的Host参数将流量转发给Ingress配置里的相应service。

修改hosts后就可以在kubernetes集群外访问以上两个service，如下图：

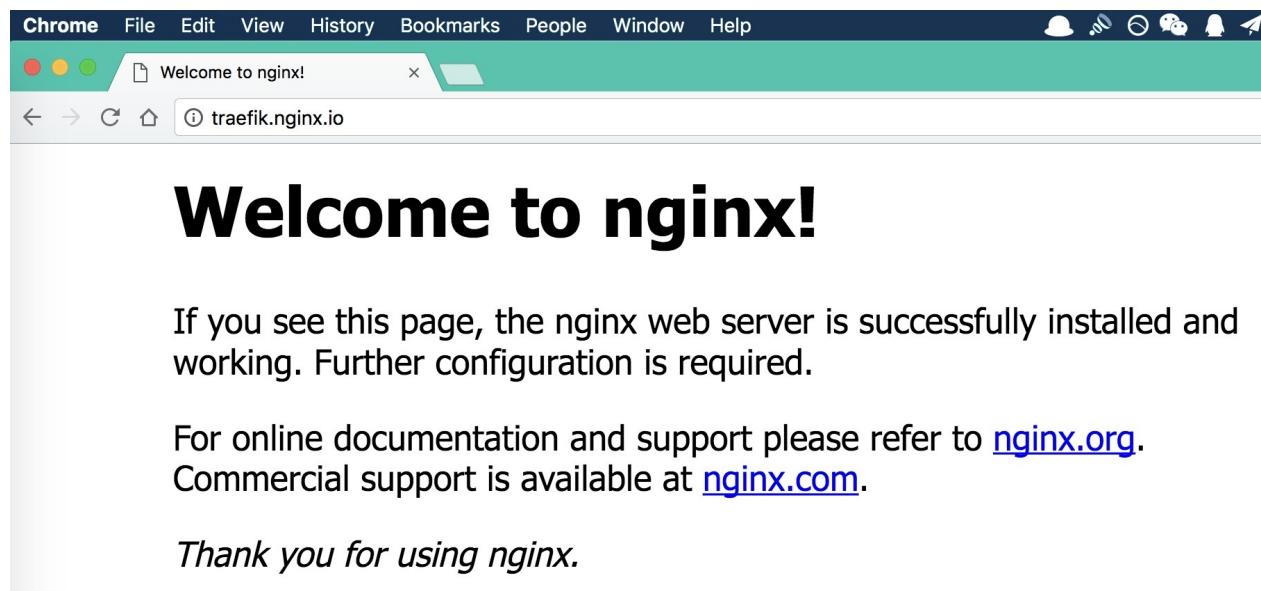


Figure: traefik-nginx

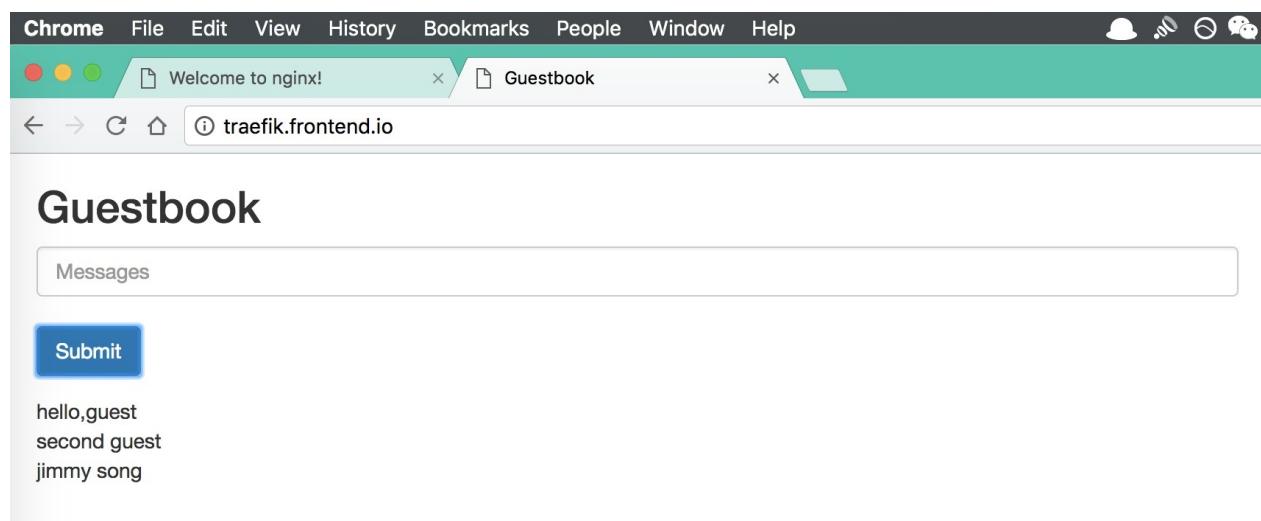


Figure: traefik-guestbook

## 参考

[Traefik-kubernetes 初试](#)

### Traefik 简介

#### Guestbook example

for GitBook      update 2017-05-12 16:45:40

## 运用Kubernetes进行分布式负载测试

该教程描述如何在[Kubernetes](#)中进行分布式负载均衡测试，包括一个web应用、docker镜像和Kubernetes controllers/services。更多资料请查看[Distributed Load Testing Using Kubernetes](#)。

### 准备

不需要[GCE](#)及其他组件，你只需要有一个[kubernetes](#)集群即可。

如果你还没有[kubernetes](#)集群，可以参考[kubernetes-handbook](#)部署一个。

### 部署Web应用

`sample-webapp` 目录下包含一个简单的web测试应用。我们将其构建为docker镜像，在[kubernetes](#)中运行。你可以自己构建，也可以直接用这个我构建好的镜像 `index.tenxcloud.com/jimmy/k8s-sample-webapp:latest`。

在[kubernetes](#)上部署[sample-webapp](#)。

```
$ cd kubernetes-config
$ kubectl create -f sample-webapp-controller.yaml
$ kubectl create -f kubectl create -f sample-webapp-service.yaml
```

### 部署Locust的Controller和Service

`locust-master` 和 `locust-work` 使用同样的docker镜像，修改cotnroller 中 `spec.template.spec.containers.env` 字段中的`value`为你 `sample-webapp service`的名字。

```
- name: TARGET_HOST
  value: http://sample-webapp:8000
```

### 创建Controller Docker镜像（可选）

`locust-master` 和 `locust-work controller` 使用的都是 `locust-tasks docker` 镜像。你可以直接下载 `gcr.io/cloud-solutions-images/locust-tasks`，也可以自己编译。自己编译大概要花几分钟时间，镜像大小为 820M。

```
$ docker build -t index.tenxcloud.com/jimmy/locust-tasks:latest  
.  
$ docker push index.tenxcloud.com/jimmy/locust-tasks:latest
```

注意：我使用的是时速云的镜像仓库。

每个 controller 的 yaml 的 `spec.template.spec.containers.image` 字段指定的是我的镜像：

```
image: index.tenxcloud.com/jimmy/locust-tasks:latest
```

## 部署 **locust-master**

```
$ kubectl create -f locust-master-controller.yaml  
$ kubectl create -f locust-master-service.yaml
```

## 部署 **locust-worker**

Now deploy `locust-worker-controller` :

```
$ kubectl create -f locust-worker-controller.yaml
```

你可以很轻易的给 work 扩容，通过命令行方式：

```
```ba sh $ kubectl scale --replicas=20 replicationcontrollers locust-worker
```

当然你也可以通过WebUI : Dashboard - Workloads - Replication Controllers - \*\*ServiceName\*\* - Scale来扩容。

![dashboard-scale](images/dashbaord-scale.jpg)

### 配置Traefik

参考[kubernetes的traefik ingress安装](<http://rootsongjc.github.io/blogs/traefik-ingress-installation/>)，在`ingress.yaml`中加入如下配置：

```
```Yaml
- host: traefik.locust.io
  http:
    paths:
      - path: /
        backend:
          serviceName: locust-master
          servicePort: 8089
```

然后执行 `kubectl replace -f ingress.yaml` 即可更新traefik。

通过Traefik的dashboard就可以看到刚增加的 `traefik.locust.io` 节点。

## 2.3 分布式负载测试

The screenshot shows the Traefik dashboard interface with three main sections:

- traefik.guestbook.io/**: Shows a route table with one entry: / PathPrefix:/ and a host rule Host:traefik.guestbook.io. Below it is a summary bar with buttons for http, Backend:traefik.guestbook.io/, PassHostHeader, and Priority:1.
- traefik.locust.io/**: Shows a route table with two entries: / PathPrefix:/ and traefik.locust.io Host:traefik.locust.io. Below it is a summary bar with buttons for http, Backend:traefik.locust.io/, PassHostHeader, and Priority:1. A green button labeled "Load Balancer: wrr" is visible.
- traefik.nginx.io/**: Shows a route table with one entry: / PathPrefix:/ and a host rule Host:traefik.nginx.io. Below it is a summary bar with buttons for http, Backend:traefik.nginx.io/, PassHostHeader, and Priority:1. A green button labeled "Load Balancer: wrr" is visible.

Each section also contains a table for servers, URL, and weight, listing the configured backends and their details.

Server	URL	Weight
frontend-1289468719-4565s	http://172.30.94.9:80	1
frontend-1289468719-rf8rw	http://172.30.71.3:80	1
frontend-1289468719-s4m6p	http://172.30.94.10:80	1

Server	URL	Weight
locust-master-p8vpn	http://172.30.94.3:8089	1

Server	URL	Weight
my-nginx-2096504489-6l1zs	http://172.30.71.11:80	1
my-nginx-2096504489-9jh91	http://172.30.71.10:80	1
my-nginx-2096504489-9s58t	http://172.30.94.8:80	1
my-nginx-2096504489-gt621	http://172.30.71.12:80	1
my-nginx-2096504489-mp4gt	http://172.30.94.11:80	1

Figure: *traefik-dashboard-locust*

## 执行测试

打开 `http://traefik.locust.io` 页面，点击 `Edit` 输入伪造的用户数和用户每秒发送的请求个数，点击 `Start Swarming` 就可以开始测试了。

## 2.3 分布式负载测试

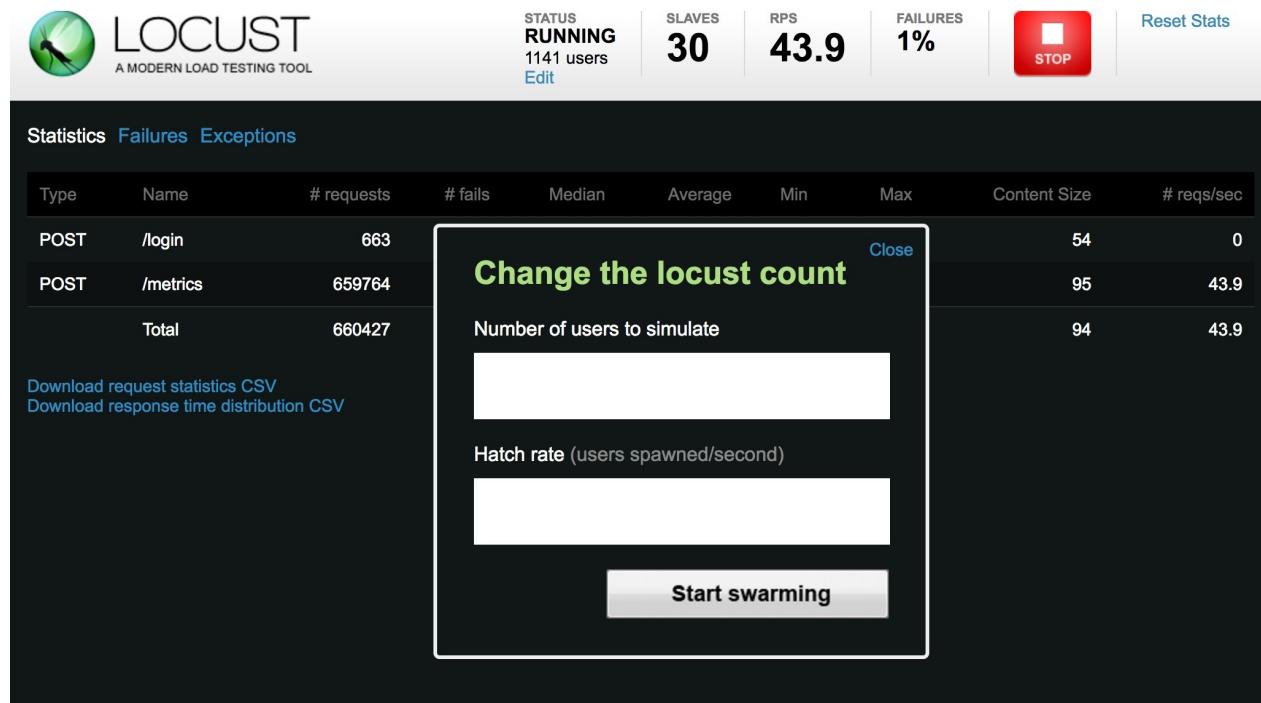


Figure: locust-start-swarming

在测试过程中调整 sample-webapp 的 pod 个数（默认设置了 1 个 pod），观察 pod 的负载变化情况。

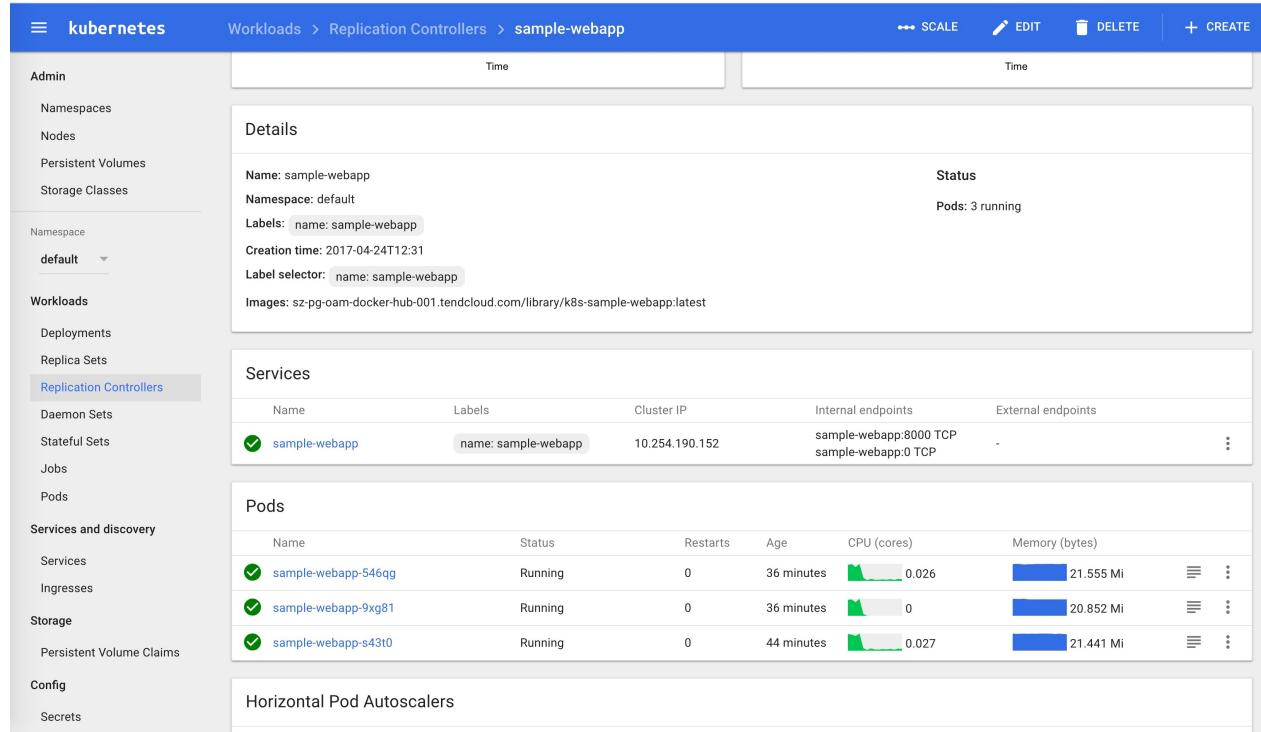


Figure: sample-webapp-rc

从一段时间的观察中可以看到负载被平均分配给了3个pod。

在locust的页面中可以实时观察也可以下载测试结果。

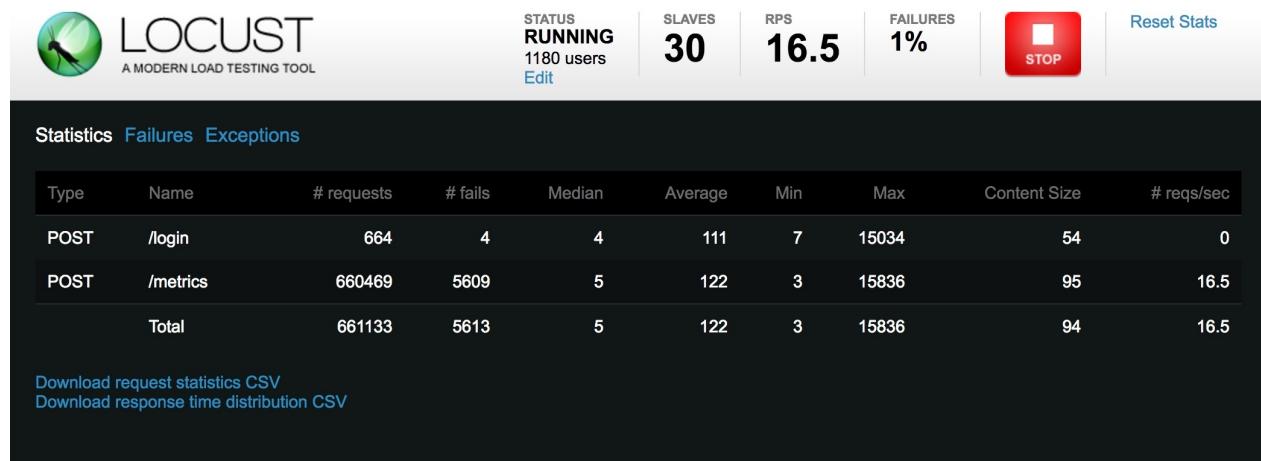


Figure: locust-dashboard

## License

This code is Apache 2.0 licensed and more information can be found in [LICENSE](#). For information on licenses for third party software and libraries, refer to the [docker-image/licenses](#) directory.

for GitBook update 2017-05-12 16:45:40

# Kubernetes网络和集群性能测试

## 准备

### 测试环境

在以下几种环境下进行测试：

- Kubernetes 集群 node 节点上通过 Cluster IP 方式访问
- Kubernetes 集群 内部通过 service 访问
- Kubernetes 集群 外部通过 traefik ingress 暴露的地址访问

### 测试地址

Cluster IP: 10.254.149.31

Service Port : 8000

Ingress Host : traefik.sample-webapp.io

### 测试工具

- [Locust](#)：一个简单易用的用户负载测试工具，用来测试 web 或其他系统能够同时处理的并发用户数。
- curl
- [kubemark](#)
- 测试程序：[sample-webapp](#)，源码见 Github [kubernetes](#) 的分布式负载测试

### 测试说明

通过向 sample-webapp 发送 curl 请求获取响应时间，直接 curl 后的结果为：

```
$ curl "http://10.254.149.31:8000/"
Welcome to the "Distributed Load Testing Using Kubernetes" sample web app
```

## 网络延迟测试

## 场景一、Kubernetes 集群node节点上通过Cluster IP 访问

测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://10.254.149.31:8000/"
```

10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.000	0.003	0.003
2	0.000	0.002	0.002
3	0.000	0.002	0.002
4	0.000	0.002	0.002
5	0.000	0.002	0.002
6	0.000	0.002	0.002
7	0.000	0.002	0.002
8	0.000	0.002	0.002
9	0.000	0.002	0.002
10	0.000	0.002	0.002

平均响应时间：2ms

时间指标说明

单位：秒

time\_connect：建立到服务器的 TCP 连接所用的时间

time\_starttransfer：在发出请求之后，Web 服务器返回数据的第一个字节所用的时间

time\_total：完成请求所用的时间

## 场景二、Kubernetes 集群内部通过service访问

## 测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://sample-webapp:8000/"
```

## 10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.004	0.006	0.006
2	0.004	0.006	0.006
3	0.004	0.006	0.006
4	0.004	0.006	0.006
5	0.004	0.006	0.006
6	0.004	0.006	0.006
7	0.004	0.006	0.006
8	0.004	0.006	0.006
9	0.004	0.006	0.006
10	0.004	0.006	0.006

平均响应时间 : **6ms**

场景三、在公网上通过**traefik ingress**访问

## 测试命令

```
curl -o /dev/null -s -w '%{time_connect} %{time_starttransfer} %{time_total}' "http://traefik.sample-webapp.io" >>result
```

## 10组测试结果

No	time_connect	time_starttransfer	time_total
1	0.043	0.085	0.085
2	0.052	0.093	0.093
3	0.043	0.082	0.082
4	0.051	0.093	0.093
5	0.068	0.188	0.188
6	0.049	0.089	0.089
7	0.051	0.113	0.113
8	0.055	0.120	0.120
9	0.065	0.126	0.127
10	0.050	0.111	0.111

平均响应时间：**110ms**

## 测试结果

在这三种场景下的响应时间测试结果如下：

- Kubernetes 集群node节点上通过Cluster IP方式访问：2ms
- Kubernetes 集群内部通过service访问：6ms
- Kubernetes 集群外部通过traefik ingress暴露的地址访问：110ms

注意：执行测试的node节点/Pod与service所在的pod的距离（是否在同一台主机上），对前两个场景可能会有一定影响。

## 网络性能测试

网络使用flannel的vxlan模式。

使用iperf进行测试。

服务端命令：

```
iperf -s -p 12345 -i 1 -M
```

客户端命令：

```
iperf -c ${server-ip} -p 12345 -i 1 -t 10 -w 20K
```

## 场景一、主机之间

[ ID]	Interval	Transfer	Bandwidth
[ 3]	0.0- 1.0 sec	598 MBytes	5.02 Gbits/sec
[ 3]	1.0- 2.0 sec	637 MBytes	5.35 Gbits/sec
[ 3]	2.0- 3.0 sec	664 MBytes	5.57 Gbits/sec
[ 3]	3.0- 4.0 sec	657 MBytes	5.51 Gbits/sec
[ 3]	4.0- 5.0 sec	641 MBytes	5.38 Gbits/sec
[ 3]	5.0- 6.0 sec	639 MBytes	5.36 Gbits/sec
[ 3]	6.0- 7.0 sec	628 MBytes	5.26 Gbits/sec
[ 3]	7.0- 8.0 sec	649 MBytes	5.44 Gbits/sec
[ 3]	8.0- 9.0 sec	638 MBytes	5.35 Gbits/sec
[ 3]	9.0-10.0 sec	652 MBytes	5.47 Gbits/sec
[ 3]	0.0-10.0 sec	6.25 GBytes	5.37 Gbits/sec

## 场景二、不同主机的Pod之间(使用flannel的vxlan模式)

[ ID]	Interval	Transfer	Bandwidth
[ 3]	0.0- 1.0 sec	372 MBytes	3.12 Gbits/sec
[ 3]	1.0- 2.0 sec	345 MBytes	2.89 Gbits/sec
[ 3]	2.0- 3.0 sec	361 MBytes	3.03 Gbits/sec
[ 3]	3.0- 4.0 sec	397 MBytes	3.33 Gbits/sec
[ 3]	4.0- 5.0 sec	405 MBytes	3.40 Gbits/sec
[ 3]	5.0- 6.0 sec	410 MBytes	3.44 Gbits/sec
[ 3]	6.0- 7.0 sec	404 MBytes	3.39 Gbits/sec
[ 3]	7.0- 8.0 sec	408 MBytes	3.42 Gbits/sec
[ 3]	8.0- 9.0 sec	451 MBytes	3.78 Gbits/sec
[ 3]	9.0-10.0 sec	387 MBytes	3.25 Gbits/sec
[ 3]	0.0-10.0 sec	3.85 GBytes	3.30 Gbits/sec

### 场景三、Node与非同主机的Pod之间（使用flannel的vxlan模式）

[ ID]	Interval	Transfer	Bandwidth
[ 3]	0.0- 1.0 sec	372 MBytes	3.12 Gbits/sec
[ 3]	1.0- 2.0 sec	420 MBytes	3.53 Gbits/sec
[ 3]	2.0- 3.0 sec	434 MBytes	3.64 Gbits/sec
[ 3]	3.0- 4.0 sec	409 MBytes	3.43 Gbits/sec
[ 3]	4.0- 5.0 sec	382 MBytes	3.21 Gbits/sec
[ 3]	5.0- 6.0 sec	408 MBytes	3.42 Gbits/sec
[ 3]	6.0- 7.0 sec	403 MBytes	3.38 Gbits/sec
[ 3]	7.0- 8.0 sec	423 MBytes	3.55 Gbits/sec
[ 3]	8.0- 9.0 sec	376 MBytes	3.15 Gbits/sec
[ 3]	9.0-10.0 sec	451 MBytes	3.78 Gbits/sec
[ 3]	0.0-10.0 sec	3.98 GBytes	3.42 Gbits/sec

### 场景四、不同主机的Pod之间（使用flannel的host-gw模式）

[ ID]	Interval	Transfer	Bandwidth
[ 5]	0.0- 1.0 sec	530 MBytes	4.45 Gbits/sec
[ 5]	1.0- 2.0 sec	576 MBytes	4.84 Gbits/sec
[ 5]	2.0- 3.0 sec	631 MBytes	5.29 Gbits/sec
[ 5]	3.0- 4.0 sec	580 MBytes	4.87 Gbits/sec
[ 5]	4.0- 5.0 sec	627 MBytes	5.26 Gbits/sec
[ 5]	5.0- 6.0 sec	578 MBytes	4.85 Gbits/sec
[ 5]	6.0- 7.0 sec	584 MBytes	4.90 Gbits/sec
[ 5]	7.0- 8.0 sec	571 MBytes	4.79 Gbits/sec
[ 5]	8.0- 9.0 sec	564 MBytes	4.73 Gbits/sec
[ 5]	9.0-10.0 sec	572 MBytes	4.80 Gbits/sec
[ 5]	0.0-10.0 sec	5.68 GBytes	4.88 Gbits/sec

### 场景五、Node与非同主机的Pod之间（使用flannel的host-gw模式）

[ ID]	Interval	Transfer	Bandwidth
[ 3]	0.0- 1.0 sec	570 MBytes	4.78 Gbits/sec
[ 3]	1.0- 2.0 sec	552 MBytes	4.63 Gbits/sec
[ 3]	2.0- 3.0 sec	598 MBytes	5.02 Gbits/sec
[ 3]	3.0- 4.0 sec	580 MBytes	4.87 Gbits/sec
[ 3]	4.0- 5.0 sec	590 MBytes	4.95 Gbits/sec
[ 3]	5.0- 6.0 sec	594 MBytes	4.98 Gbits/sec
[ 3]	6.0- 7.0 sec	598 MBytes	5.02 Gbits/sec
[ 3]	7.0- 8.0 sec	606 MBytes	5.08 Gbits/sec
[ 3]	8.0- 9.0 sec	596 MBytes	5.00 Gbits/sec
[ 3]	9.0-10.0 sec	604 MBytes	5.07 Gbits/sec
[ 3]	0.0-10.0 sec	5.75 GBytes	4.94 Gbits/sec

## 网络性能对比综述

使用 Flannel 的 **vxlan** 模式实现每个 pod 一个 IP 的方式，会比宿主机直接互联的网络性能损耗 30%~40%，符合网上流传的测试结论。而 flannel 的 host-gw 模式比起宿主机互连的网络性能损耗大约是 10%。

Vxlan 会有一个封包解包的过程，所以会对网络性能造成较大的损耗，而 host-gw 模式是直接使用路由信息，网络损耗小，关于 host-gw 的架构请访问 [Flannel host-gw architecture](#)。

## Kubernetes 的性能测试

参考 [Kubernetes 集群性能测试](#) 中的步骤，对 kubernetes 的性能进行测试。

我的集群版本是 Kubernetes 1.6.0，首先克隆代码，将 kubernetes 目录复制到 `$GOPATH/src/k8s.io/` 下然后执行：

```

$ ./hack/generate-bindata.sh
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes
Generated bindata file : test/e2e/generated/bindata.go has 13498
test/e2e/generated/bindata.go lines of lovely automated artifacts
No changes in generated bindata file: pkg/generated/bindata.go
/usr/local/src/k8s.io/kubernetes
$ make WHAT="test/e2e/e2e.test"
...
+++ [0425 17:01:34] Generating bindata:
    test/e2e/generated/gobindata_util.go
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes/test/e2e/generated
/usr/local/src/k8s.io/kubernetes/test/e2e/generated
+++ [0425 17:01:34] Building go targets for linux/amd64:
    test/e2e/e2e.test
$ make ginkgo
+++ [0425 17:05:57] Building the toolchain targets:
    k8s.io/kubernetes/hack/cmd/teststale
    k8s.io/kubernetes/vendor/github.com/jteeuwen/go-bindata/go-bindata
+++ [0425 17:05:57] Generating bindata:
    test/e2e/generated/gobindata_util.go
/usr/local/src/k8s.io/kubernetes /usr/local/src/k8s.io/kubernetes/test/e2e/generated
/usr/local/src/k8s.io/kubernetes/test/e2e/generated
+++ [0425 17:05:58] Building go targets for linux/amd64:
    vendor/github.com/onsi/ginkgo/ginkgo

$ export KUBERNETES_PROVIDER=local
$ export KUBECTL_PATH=/usr/bin/kubectl
$ go run hack/e2e.go -v -test --test_args="--host=http://172.20
.0.113:8080 --ginkgo.focus=\[Feature:Performance\]" >>log.txt

```

测试结果

```
Apr 25 18:27:31.461: INFO: API calls latencies: {
```

```

"apicalls": [
  {
    "resource": "pods",
    "verb": "POST",
    "latency": {
      "Perc50": 2148000,
      "Perc90": 13772000,
      "Perc99": 14436000,
      "Perc100": 0
    }
  },
  {
    "resource": "services",
    "verb": "DELETE",
    "latency": {
      "Perc50": 9843000,
      "Perc90": 11226000,
      "Perc99": 12391000,
      "Perc100": 0
    }
  },
  ...
]

Apr 25 18:27:31.461: INFO: [Result:Performance] {
  "version": "v1",
  "dataItems": [
    {
      "data": {
        "Perc50": 2.148,
        "Perc90": 13.772,
        "Perc99": 14.436
      },
      "unit": "ms",
      "labels": {
        "Resource": "pods",
        "Verb": "POST"
      }
    },
    ...
  ]
}

2.857: INFO: Running AfterSuite actions on all node
Apr 26 10:35:32.857: INFO: Running AfterSuite actions on node 1

```

```
Ran 2 of 606 Specs in 268.371 seconds
SUCCESS! -- 2 Passed | 0 Failed | 0 Pending | 604 Skipped PASS

Ginkgo ran 1 suite in 4m28.667870101s
Test Suite Passed
```

从kubemark输出的日志中可以看到**API calls latencies**和**Performance**。

日志里显示，创建**90**个**pod**用时**40**秒以内，平均创建每个**pod**耗时**0.44**秒。

## 不同**type**的资源类型**API**请求耗时分布

<b>Resource</b>	<b>Verb</b>	<b>50%</b>	<b>90%</b>	<b>99%</b>
services	DELETE	8.472ms	9.841ms	38.226ms
endpoints	PUT	1.641ms	3.161ms	30.715ms
endpoints	GET	931μs	10.412ms	27.97ms
nodes	PATCH	4.245ms	11.117ms	18.63ms
pods	PUT	2.193ms	2.619ms	17.285ms

从 `log.txt` 日志中还可以看到更多详细请求的测试指标。

Name	Labels	Pods	Age	Images
load-medium-1	name: load-medium-1	30 / 30	44 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-1	name: load-small-1	5 / 5	43 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-10	name: load-small-10	5 / 5	42 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-11	name: load-small-11	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-12	name: load-small-12	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-2	name: load-small-2	5 / 5	36 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-3	name: load-small-3	5 / 5	38 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-4	name: load-small-4	5 / 5	39 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-5	name: load-small-5	5 / 5	41 seconds	gcr.io/google_containers/serve_hostname:v1.4
load-small-6	name: load-small-6	5 / 5	38 seconds	gcr.io/google_containers/serve_hostname:v1.4

Figure: kubernetes-dashboard

## 注意事项

测试过程中需要用到docker镜像存储在GCE中，需要翻墙下载，我没看到哪里配置这个镜像的地址。该镜像副本已上传时速云：

用到的镜像有如下两个：

- gcr.io/google\_containers/pause-amd64:3.0
- gcr.io/google\_containers/serve\_hostname:v1.4

时速云镜像地址：

- index.tenxcloud.com/jimmy/pause-amd64:3.0
- index.tenxcloud.com/jimmy/serve\_hostname:v1.4

将镜像pull到本地后重新打tag。

## Locust 测试

### 请求统计

Method	Name	# requests	# failures	Median response time	Average response time	Min response time
POST	/login	5070	78	59000	80551	11218
POST	/metrics	5114232	85879	63000	82280	29518
None	Total	5119302	85957	63000	82279	11218

响应时间分布

Name	# requests	50%	66%	75%	80%	90%	99%
POST /login	5070	59000	125000	140000	148000	160000	166000
POST /metrics	5114993	63000	127000	142000	149000	160000	166000
None Total	5120063	63000	127000	142000	149000	160000	166000

以上两个表格都是瞬时值。请求失败率在2%左右。

Sample-webapp起了48个pod。

Locust模拟10万用户，每秒增长100个。

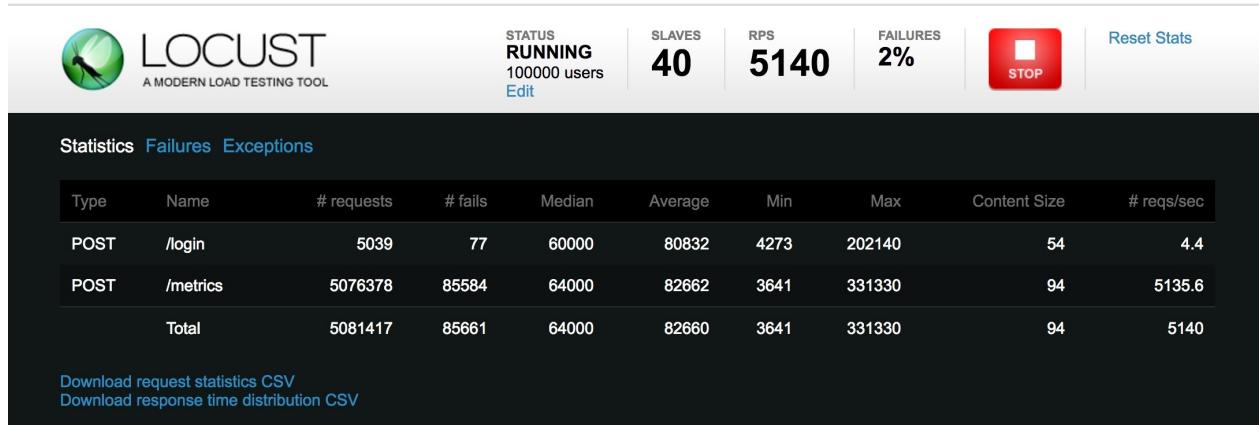


Figure: locust-test

参考

基于 Python 的性能测试工具 locust (与 LR 的简单对比)

[Locust docs](#)

[python用户负载测试工具 : locust](#)

[Kubernetes集群性能测试](#)

[CoreOS是如何将Kubernetes的性能提高10倍的](#)

[Kubernetes 1.3 的性能和弹性 —— 2000 节点，60,0000 Pod 的集群](#)

[运用Kubernetes进行分布式负载测试](#)

[Kubemark User Guide](#)

[Flannel host-gw architecture](#)

for GitBook      update 2017-05-12 16:45:40

# 边缘节点配置

## 前言

为了配置kubernetes中的traefik ingress的高可用，对于kubernetes集群以外只暴露一个访问入口，需要使用keepalived排除单点问题。本文参考了[kube-keepalived-vip](#)，但并没有使用容器方式安装，而是直接在node节点上安装。

## 定义

首先解释下什么叫边缘节点（Edge Node），所谓的边缘节点即集群内部用来向集群外暴露服务能力的节点，集群外部的服务通过该节点来调用集群内部的服务，边缘节点是集群内外交流的一个Endpoint。

边缘节点要考虑两个问题

- 边缘节点的高可用，不能有单点故障，否则整个kubernetes集群将不可用
- 对外的一致暴露端口，即只能有一个外网访问IP和端口

## 架构

为了满足边缘节点的以上需求，我们使用[keepalived](#)来实现。

在Kubernetes集群外部配置nginx来访问边缘节点的VIP。

选择Kubernetes的三个node作为边缘节点，并安装keepalived。

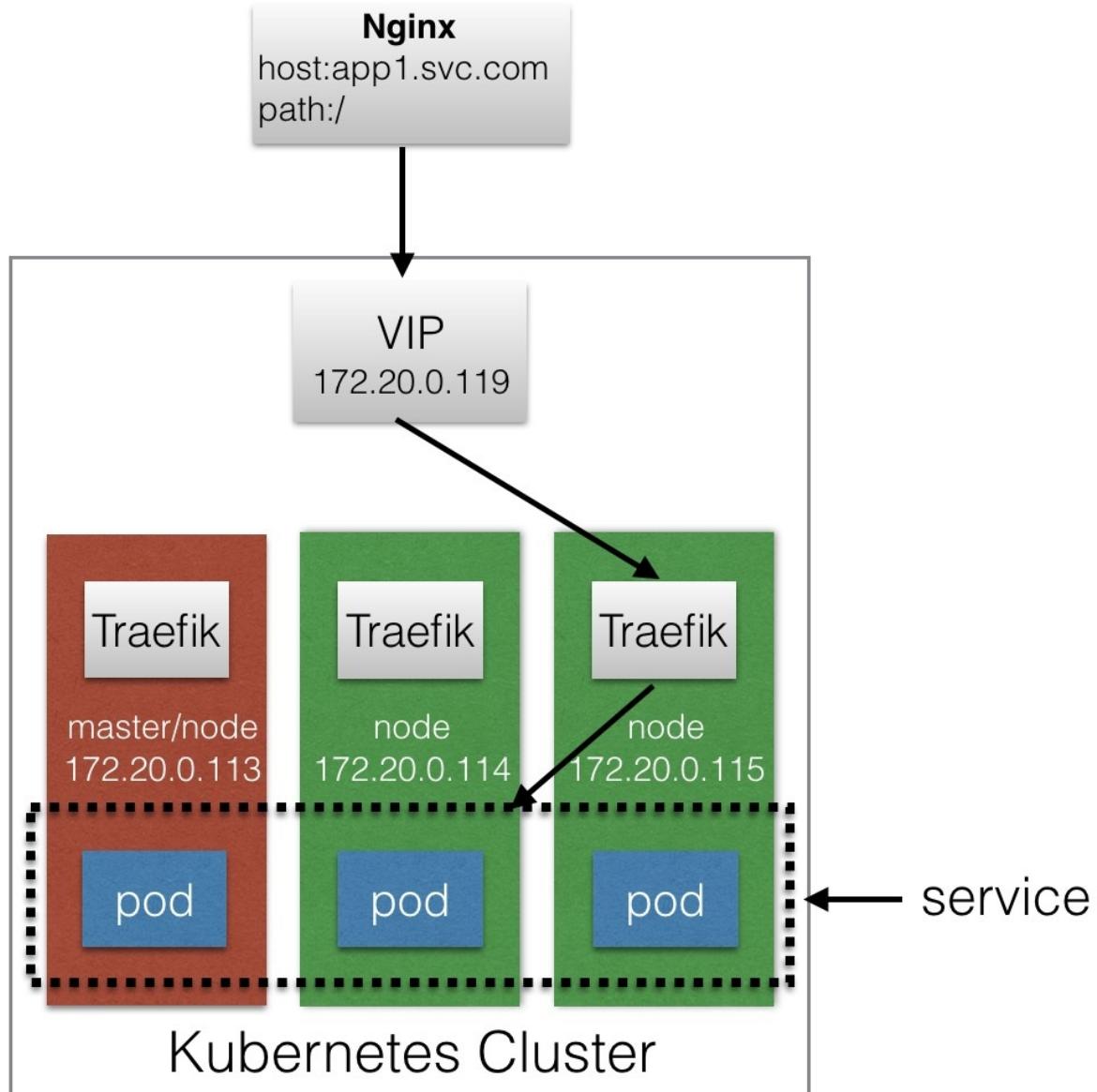


Figure: 边缘节点架构

## 准备

复用 kubernetes 测试集群的三台主机。

172.20.0.113

172.20.0.114

172.20.0.115

## 安装

使用keepalived管理VIP，VIP是使用IPVS创建的，IPVS已经成为linux内核的模块，不需要安装

LVS的工作原理请参

考：<http://www.cnblogs.com/codebean/archive/2011/07/25/2116043.html>

不使用镜像方式安装了，直接手动安装，指定三个节点为边缘节点（Edge node）。

因为我们的测试集群一共只有三个node，所有在在三个node上都要安装keepalived和ipvsadmin。

```
yum install keepalived ipvsadm
```

## 配置说明

需要对原先的traefik ingress进行改造，从以Deployment方式启动改成DeamonSet。还需要指定一个与node在同一网段的IP地址作为VIP，我们指定成172.20.0.119，配置keepalived前需要先保证这个IP没有被分配。。

- Traefik以DaemonSet的方式启动
- 通过nodeSelector选择边缘节点
- 通过hostPort暴露端口
- 当前VIP漂移到了172.20.0.115上
- Traefik根据访问的host和path配置，将流量转发到相应的service上

## 配置keepalived

参考基于keepalived 实现VIP转移，lvs，nginx的高可用，配置keepalived。

keepalived的官方配置文档见：<http://keepalived.org/pdf/UserGuide.pdf>

配置文件 /etc/keepalived/keepalived.conf 文件内容如下：

```
! Configuration File for keepalived
```

```
global_defs {  
    notification_email {  
        root@localhost  
    }  
    notification_email_from kaadmin@localhost  
    smtp_server 127.0.0.1  
    smtp_connect_timeout 30  
    router_id LVS_DEVEL  
}  
  
vrrp_instance VI_1 {  
    state MASTER  
    interface eth0  
    virtual_router_id 51  
    priority 100  
    advert_int 1  
    authentication {  
        auth_type PASS  
        auth_pass 1111  
    }  
    virtual_ipaddress {  
        172.20.0.119  
    }  
}  
  
virtual_server 172.20.0.119 80{  
    delay_loop 6  
    lb_algo loadbalance  
    lb_kind DR  
    nat_mask 255.255.255.0  
    persistence_timeout 0  
    protocol TCP  
  
    real_server 172.20.0.113 80{  
        weight 1  
        TCP_CHECK {  
            connect_timeout 3  
        }  
    }  
    real_server 172.20.0.114 80{
```

```
    weight 1
    TCP_CHECK {
        connect_timeout 3
    }
}
real_server 172.20.0.115 80{
    weight 1
    TCP_CHECK {
        connect_timeout 3
    }
}
}
```

Realserver 的IP和端口即traefik供外网访问的IP和端口。

将以上配置分别拷贝到另外两台node的 /etc/keepalived 目录下。

我们使用转发效率最高的 `lb_kind DR` 直接路由方式转发，使用 `TCP_CHECK` 来检测 `real_server` 的 `health`。

### 启动keepalived

```
systemctl start keepalived
```

三台node都启动了keepalived后，观察eth0的IP，会在三台node的某一台上发现一个VIP是172.20.0.119。

```
$ ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether f4:e9:d4:9f:6b:a0 brd ff:ff:ff:ff:ff:ff
    inet 172.20.0.115/17 brd 172.20.127.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet 172.20.0.119/32 scope global eth0
        valid_lft forever preferred_lft forever
```

关掉拥有这个VIP主机上的keepalived，观察VIP是否漂移到了另外两台主机的其中之一上。

## 改造Traefik

在这之前我们启动的traefik使用的是deployment，只启动了一个pod，无法保证高可用（即需要将pod固定在某一台主机上，这样才能对外提供一个唯一的访问地址），现在使用了keepalived就可以通过VIP来访问traefik，同时启动多个traefik的pod保证高可用。

配置文件 `traefik.yaml` 内容如下：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: traefik-ingress-lb
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress-lb
spec:
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress-lb
        name: traefik-ingress-lb
    spec:
      terminationGracePeriodSeconds: 60
      hostNetwork: true
      restartPolicy: Always
      serviceAccountName: ingress
      containers:
        - image: traefik
          name: traefik-ingress-lb
          resources:
            limits:
              cpu: 200m
              memory: 30Mi
            requests:
              cpu: 100m
              memory: 20Mi
          ports:
            - name: http
```

```

        containerPort: 80
        hostPort: 80
      - name: admin
        containerPort: 8580
        hostPort: 8580
      args:
        - --web
        - --web.address=:8580
        - --kubernetes
      nodeSelector:
        edgenode: "true"

```

注意，我们使用了 `nodeSelector` 选择边缘节点来调度traefik-ingress-lb运行在它上面，所有你需要使用：

```

kubectl label nodes 172.20.0.113 edgenode=true
kubectl label nodes 172.20.0.114 edgenode=true
kubectl label nodes 172.20.0.115 edgenode=true

```

给三个node打标签。

查看DaemonSet的启动情况：

```

$ kubectl -n kube-system get ds
NAME           DESIRED   CURRENT   READY   UP-TO-DATE   AGE
AVAILABLE     NODE-SELECTOR
traefik-ingress-lb   3         3         3       3            2h
3             edgenode=true

```

现在就可以在外网通过172.20.0.119:80来访问到traefik ingress了。

## 参考

[kube-keepalived-vip](#)

<http://www.keepalived.org/>

[keepalived工作原理与配置说明](#)

### LVS简介及使用

基于keepalived 实现VIP转移，lvs，nginx的高可用

for GitBook      update 2017-05-12 16:45:40

# Deployment概念解析

本文翻译自 kubernetes 官方文

档：<https://github.com/kubernetes/kubernetes.github.io/blob/master/docs/concepts/workloads/controllers/deployment.md>

根据2017年5月10日的Commit 8481c02 翻译。

## Deployment是什么？

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

你只需要在Deployment中描述你想要的目标状态是什么，Deployment controller就会帮你将Pod和Replica Set的实际状态改变到你的目标状态。你可以定义一个全新的Deployment，也可以创建一个新的替换旧的Deployment。

一个典型的用例如下：

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

## 创建Deployment

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

下载示例文件并执行命令：

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

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

然后立即执行 `get` 将获得如下结果：

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

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

过几秒后再执行 `get` 命令，将获得如下输出：

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

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

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

你可能会注意到Replica Set的名字总是 <Deployment的名字>-<pod template的hash值>。

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

刚创建的Replica Set将保证总是有3个nginx的pod存在。

注意：你必须在Deployment中的selector指定正确pod template label（在该示例中是 app = nginx），不要跟其他的controller搞混了（包括Deployment、Replica Set、Replication Controller等）。Kubernetes本身不会阻止你这么做，如果你真的这么做了，这些controller之间会相互打架，并可能导致不正确的行为。

## 更新Deployment

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

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

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

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

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

查看rollout的状态，只要执行：

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

Rollout成功后，`get Deployment`：

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

UP-TO-DATE的replica的数目已经达到了配置中要求的数目。

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

We can run `kubectl get rs` to see that the Deployment updated the Pods by creating a new Replica Set and scaling it up to 3 replicas, as well as scaling down the old Replica Set to 0 replicas.

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

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

执行 `get pods` 只会看到当前的新的pod:

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

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

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

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

在未来的**Kubernetes**版本中，将从**1-1变成25%-25%**。

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

```
$ kubectl describe deployments
Name:           nginx-deployment
Namespace:      default
CreationTimestamp: Tue, 15 Mar 2016 12:01:06 -0700
Labels:          app=nginx
Selector:        app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
OldReplicaSets: <none>
NewReplicaSet:  nginx-deployment-1564180365 (3/3 replicas created)
Events:
FirstSeen  LastSeen  Count  From                Subobject
ctPath    Type      Reason             Message
-----  -----  -----  -----  -----
-----  -----  -----  -----
36s       36s       1      {deployment-controller } 
          Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-2035384211 to 3
23s       23s       1      {deployment-controller } 
          Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-1564180365 to 1
23s       23s       1      {deployment-controller } 
          Normal    ScalingReplicaSet  Scaled down replica set nginx-deployment-2035384211 to 2
23s       23s       1      {deployment-controller } 
          Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-1564180365 to 2
21s       21s       1      {deployment-controller } 
          Normal    ScalingReplicaSet  Scaled down replica set nginx-deployment-2035384211 to 0
21s       21s       1      {deployment-controller } 
          Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-1564180365 to 3
```

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

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

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

## Rollover（多个rollout并行）

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

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

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

## 回退Deployment

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

默认情况下，kubernetes会在系统中保存前两次的Deployment的rollout历史记录，以便你可以随时会退（你可以修改 revision history limit 来更改保存的revision数）。 $\beta$

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

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

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

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

Rollout将会卡住。

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

按住Ctrl-C停止上面的rollout状态监控。

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

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

看下创建Pod，你会看到有两个新的呃Replica Set创建的Pod处于ImagePullBackOff状态，循环拉取镜像。

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

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

```
$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700
Labels:          app=nginx
Selector:        app=nginx
Replicas:       2 updated | 3 total | 2 available | 2 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
OldReplicaSets:  nginx-deployment-1564180365 (2/2 replicas created)
NewReplicaSet:   nginx-deployment-3066724191 (2/2 replicas created)
Events:
FirstSeen  LastSeen   Count  From             Subobject
tPath     Type       Reason            Message
-----  -----  -----  -----  -----
1m        1m        1      {deployment-controller } 
Normal      ScalingReplicaSet  Scaled up replica set nginx-deployment-2035384211 to 3
22s        22s        1      {deployment-controller }
```

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

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

### 检查Deployment升级的历史记录

首先，检查下Deployment的revision：

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

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

查看单个revision的详细信息：

```
$ kubectl rollout history deployment/nginx-deployment --revision=2
deployments "nginx-deployment" revision 2
  Labels:      app=nginx
               pod-template-hash=1159050644
  Annotations: kubernetes.io/change-cause=kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
  Containers:
    nginx:
      Image:      nginx:1.9.1
      Port:       80/TCP
      QoS Tier:
        cpu:      BestEffort
        memory:   BestEffort
      Environment Variables: <none>
    No volumes.
```

## 回退到历史版本

现在，我们可以决定回退当前的rollout到之前的版本：

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

也可以使用 `--revision` 参数指定某个历史版本：

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

与rollout相关的命令详细文档见[kubectl rollout](#)。

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

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

$ kubectl describe deployment
Name:           nginx-deployment
Namespace:      default
CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700
Labels:          app=nginx
Selector:        app=nginx
Replicas:       3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
OldReplicaSets: <none>
NewReplicaSet:   nginx-deployment-1564180365 (3/3 replicas created)
Events:
  FirstSeen  LastSeen  Count  From                Subobject
  tPath      Type      Reason             Message
  -----  -----  -----  -----  -----

```

```

----- ----- -----
 30m      30m      1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-2035384211 to 3
 29m      29m      1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-1564180365 to 1
 29m      29m      1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled down replica set nginx-deployment-2035384211 to 2
 29m      29m      1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-1564180365 to 2
 29m      29m      1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled down replica set nginx-deployment-2035384211 to 0
 29m      29m      1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-3066724191 to 2
 29m      29m      1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-3066724191 to 1
 29m      29m      1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled down replica set nginx-deployment-1564180365 to 2
 2m       2m       1      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled down replica set nginx-deployment-3066724191 to 0
 2m       2m       1      {deployment-controller }
       Normal    DeploymentRollback  Rolled back deployment "nginx-deployment" to revision 2
 29m      2m       2      {deployment-controller }
       Normal    ScalingReplicaSet  Scaled up replica set nginx-deployment-1564180365 to 3

```

## 清理Policy

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

## Deployment扩容

你可以使用以下命令扩容Deployment：

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

假设你的集群中启用了[horizontal pod autoscaling](#)，你可以给Deployment设置一个autoscaler，基于当前Pod的CPU利用率选择最少和最多的Pod数。

```
$ kubectl autoscale deployment nginx-deployment --min=10 --max=1
5 --cpu-percent=80
deployment "nginx-deployment" autoscaled
```

## 比例扩容

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

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

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

你更新了一个镜像，而在集群内部无法解析。

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

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

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

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

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

```
$ kubectl get deploy
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   15        18        7            8           7m

$ kubectl get rs
NAME              DESIRED   CURRENT   READY   AGE
nginx-deployment-1989198191   7         7         0       7m
nginx-deployment-618515232     11        11        11      7m
```

## 暂停和恢复Deployment

## 4.1 Deployment概念解析

你可以在出发一次或多次更新前暂停一个Deployment，然后再恢复它。这样你就能多次暂停和恢复Deployment，在此期间进行一些修复工作，而不会出发不必要的rollout。

例如使用刚刚创建Deployment：

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

使用以下命令暂停Deployment：

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

然后更新Deployment中的镜像：

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

注意新的rollout启动了：

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

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

你可以进行任意多次更新，例如更新使用的资源：

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

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

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

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

注意：在恢复Deployment之前你无法回退一个暂停了个Deployment。

## Deployment状态

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

## Progressing Deployment

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

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

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

## Complete Deployment

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

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

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

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

## Failed Deployment

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

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

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

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

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

Once the deadline has been exceeded, the Deployment controller adds a with the following attributes to the Deployment's

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

- Type=Progressing
- Status=False
- Reason=ProgressDeadlineExceeded

浏览 [Kubernetes API conventions](#) 查看关于status conditions的更多信息。

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

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

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

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

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

```

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

```

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

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

你可以通过缩容Deployment的方式解决配额不足的问题，或者增加你的namespace的配额。如果你满足了配额条件后，Deployment controller就会完成你的Deployment rollout，你将看到Deployment的状态更新为成功状态（`Status=True` 并且 `Reason>NewReplicaSetAvailable`）。

### Conditions:

Type	Status	Reason
---	-----	-----
Available	True	MinimumReplicasAvailable
Progressing	True	NewReplicaSetAvailable

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

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

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

## 操作失败的Deployment

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

## 清理Policy

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

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

## 用例

### 金丝雀Deployment

如果你想要使用Deployment对部分用户或服务器发布relaese，你可以创建多个Deployment，每个对一个release，参照[managing resources](#) 中对金丝雀模式的描述。

### 编写Deployment Spec

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

Deployment也需要 `.spec section`.

### Pod Template

`.spec.template` 是 `.spec` 中唯一要求的字段。

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

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

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

### Replicas

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

## Selector

`.spec.selector` 是可选字段，用来指定 [label selector](#)，圈定Deployment管理的pod范围。

如果被指定，`.spec.selector` 必须匹配

`.spec.template.metadata.labels`，否则它将被API拒绝。如果

`.spec.selector` 没有被指定，`.spec.selector.matchLabels` 默认是

`.spec.template.metadata.labels`。

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

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

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

## 策略

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

## Recreate Deployment

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

## Rolling Update Deployment

`.spec.strategy.type==RollingUpdate` 时，Deployment使用[rolling update](#) 的方式更新Pod。你可以指定`maxUnavailable` 和 `maxSurge` 来控制 rolling update 进程。

## Max Unavailable

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

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

## Max Surge

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

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

## Progress Deadline Seconds

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

如果设置该参数，该值必须大于 `.spec.minReadySeconds` 。

## Min Ready Seconds

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

## Rollback To

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

## Revision

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

## Revision History Limit

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

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

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

## Paused

`.spec.paused` 是一个可选配置项，boolean值。用来指定暂停和恢复Deployment。Paused和没有paused的Deployment之间的唯一区别就是，所有对paused deployment中的PodTemplateSpec的修改都不会触发新的rollout。Deployment被创建之后默认是非paused。

## Alternative to Deployments

## kubectl rolling update

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

for GitBook      update 2017-05-12 16:45:40

# Kubernetes 中的 RBAC 支持

在 Kubernetes 1.6 版本中新增角色访问控制机制（Role-Based Access，RBAC）让集群管理员可以针对特定使用者或服务账号的角色，进行更精确的资源访问控制。在 RBAC 中，权限与角色相关联，用户通过成为适当角色的成员而得到这些角色的权限。这就极大地简化了权限的管理。在一个组织中，角色是为了完成各种工作而创造，用户则依据它的责任和资格来被指派相应的角色，用户可以很容易地从一个角色被指派到另一个角色。

## 前言

本文翻译自 [RBAC Support in Kubernetes](#)，转载自 [kubernetes 中文社区](#)，译者催总，[Jimmy Song](#) 做了稍许修改。该文章是 [5 天内了解 Kubernetes 1.6 新特性的系列文章之一](#)。

One of the highlights of the [Kubernetes 1.6](#) 中的一个亮点时 RBAC 访问控制机制升级到了 beta 版本。RBAC，基于角色的访问控制机制，是用来管理 kubernetes 集群中资源访问权限的机制。使用 RBAC 可以很方便的更新访问授权策略而不用重启集群。

本文主要关注新特性和最佳实践。

## RBAC vs ABAC

目前 kubernetes 中已经有一系列 [鉴权机制](#)。鉴权的作用是，决定一个用户是否有权使用 Kubernetes API 做某些事情。它除了会影响 kubectl 等组件之外，还会对一些运行在集群内部并对集群进行操作的软件产生作用，例如使用了 Kubernetes 插件的 Jenkins，或者是利用 Kubernetes API 进行软件部署的 Helm。ABAC 和 RBAC 都能够对访问策略进行配置。

ABAC（Attribute Based Access Control）本来是不错的概念，但是在 Kubernetes 中的实现比较难于管理和理解，而且需要对 Master 所在节点的 SSH 和文件系统权限，而且要使得对授权的变更成功生效，还需要重新启动 API Server。

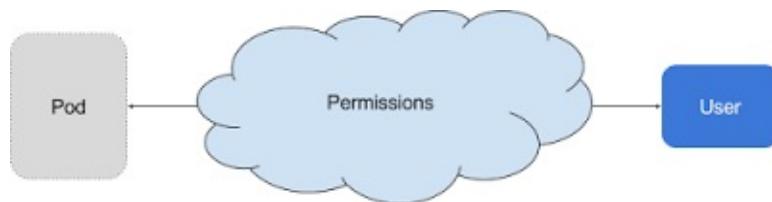
而 RBAC 的授权策略可以利用 `kubectl` 或者 Kubernetes API 直接进行配置。

**RBAC** 可以授权给用户，让用户有权进行授权管理，这样就可以无需接触节点，直接进行授权管理。RBAC 在 Kubernetes 中被映射为 API 资源和操作。

因为 Kubernetes 社区的投入和偏好，相对于 ABAC 而言，RBAC 是更好的选择。

### 基础概念

需要理解 RBAC 一些基础的概念和思路，RBAC 是让用户能够访问 [Kubernetes API 资源](#) 的授权方式。



*Figure: img*

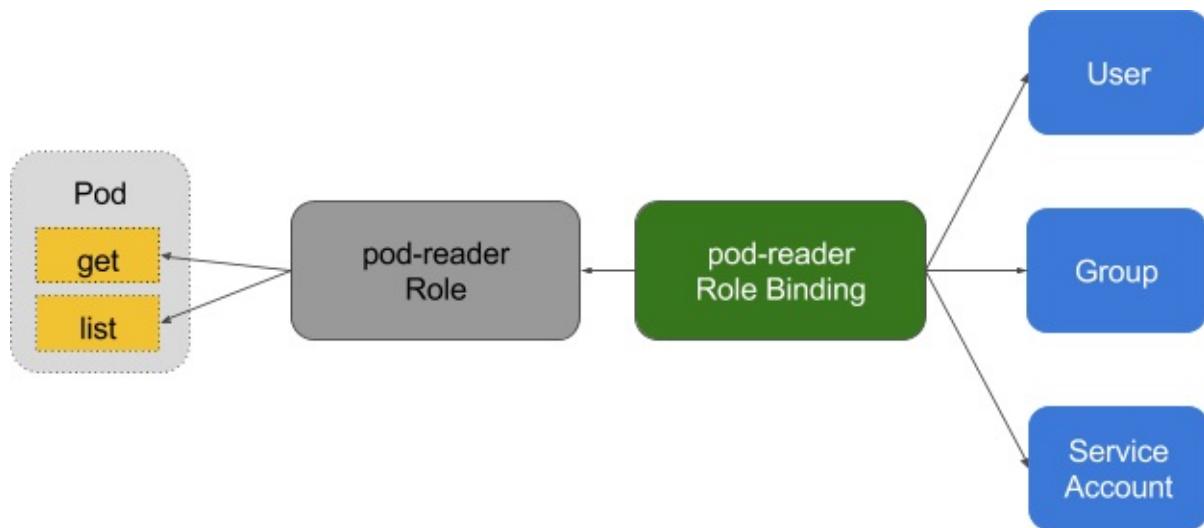
在 RBAC 中定义了两个对象，用于描述在用户和资源之间的连接权限。

#### role

角色是一系列权限的集合，例如一个角色可以包含读取 Pod 的权限和列出 Pod 的权限，ClusterRole 跟 Role 类似，但是可以在集群中到处使用（Role 是 namespace 一级的）。

#### role binding

RoleBinding 把角色映射到用户，从而让这些用户继承角色在 namespace 中的权限。ClusterRoleBinding 让用户继承 ClusterRole 在整个集群中的权限。

*Figure: img*

另外还要考虑 cluster roles 和 cluster role binding。cluster role 和 cluster role binding 方法跟 role 和 role binding 一样，出了它们有更广的 scope。详细差别请访问 [role binding](#) 与 [clsuter role binding](#).

## Kubernetes 中的 RBAC

RBAC 现在被 Kubernetes 深度集成，并使用他给系统组件进行授权。[System Roles](#) 一般具有前缀 `system:`，很容易识别：

```
$ kubectl get clusterroles --namespace=kube-system
NAME                                     AGE
admin                                      10d
cluster-admin                             10d
edit                                       10d
system:auth-delegator                      10d
system:basic-user                           10d
system:controller:attachdetach-controller   10d
system:controller:certificate-controller    10d
system:controller:cronjob-controller        10d
system:controller:daemon-set-controller     10d
system:controller:deployment-controller     10d
system:controller:disruption-controller    10d
system:controller:endpoint-controller      10d
system:controller:generic-garbage-collector 10d
```

system:controller:horizontal-pod-autoscaler	10d
system:controller:job-controller	10d
system:controller:namespace-controller	10d
system:controller:node-controller	10d
system:controller:persistent-volume-binder	10d
system:controller:pod-garbage-collector	10d
system:controller:replicaset-controller	10d
system:controller:replication-controller	10d
system:controller:resourcequota-controller	10d
system:controller:route-controller	10d
system:controller:service-account-controller	10d
system:controller:service-controller	10d
system:controller:statefulset-controller	10d
system:controller:ttl-controller	10d
system:discovery	10d
system:heapster	10d
system:kube-aggregator	10d
system:kube-controller-manager	10d
system:kube-dns	10d
system:kube-scheduler	10d
system:node	10d
system:node-bootstrapper	10d
system:node-problem-detector	10d
system:node-proxier	10d
system:persistent-volume-provisioner	10d
view	10d

RBAC 系统角色已经完成足够的覆盖，让集群可以完全在 RBAC 的管理下运行。

在 ABAC 到 RBAC 进行迁移的过程中，有些在 ABAC 集群中缺省开放的权限，在 RBAC 中会被视为不必要的授权，会对其进行降级。这种情况会影响到使用 Service Account 的负载。ABAC 配置中，从 Pod 中发出的请求会使用 Pod Token，API Server 会为其授予较高权限。例如下面的命令在 ABAC 集群中会返回 JSON 结果，而在 RBAC 的情况下则会返回错误。

```
$ kubectl run nginx --image=nginx:latest
$ kubectl exec -it $(kubectl get pods -o jsonpath='{.items[0].metadata.name}') bash
$ apt-get update && apt-get install -y curl
$ curl -ik \
  -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io
/serviceaccount/token)" \
  https://kubernetes/api/v1/namespaces/default/pods
```

所有在 Kubernetes 集群中运行的应用，一旦和 API Server 进行通信，都会有可能受到迁移的影响。

要平滑的从 ABAC 升级到 RBAC，在创建 1.6 集群的时候，可以同时启用 [ABAC](#) 和 [RBAC](#)。当他们同时启用的时候，对一个资源的权限请求，在任何一方获得放行都会获得批准。然而在这种配置下的权限太过粗放，很可能无法在单纯的 RBAC 环境下工作。

目前 RBAC 已经足够了，ABAC 可能会被弃用。在可见的未来 ABAC 依然会保留在 kubernetes 中，不过开发的重心已经转移到了 RBAC。

## 参考

[RBAC documentation](#)

[Google Cloud Next talks 1](#)

[Google Cloud Next talks 2](#)

[在 Kubernetes Pod 中使用 Service Account 访问 API Server](#)

for GitBook      update 2017-05-12 16:45:40

# Kubernetes中的网络模式解析

Kubernetes本身不提供网络模式，而是通过[CNI网络插件](#)实现，这与docker中使用的libnetwork（CNM的一种实现）不同，可关于CNM和CNI的详细信息可以参考[Rancher网络探讨和扁平网络实现](#)。

一般情况下第一次安装和试用kubernetes的时候都会推荐使用flannel这个网络插件，这也是官方文档中推荐的，vxlan网络模式是最常使用的，但是这种模式对网络的损耗较大，大约在40%至50%，而host-gw模式对网络的损耗比较小，只有10%左右。

关于flannel的几种网络模式和性能测试请参考[Comparison of Networking Solutions for Kubernetes](#)。

Flannel的配置参考[Flannel configuration](#)。

## Flannel host-gw模式架构

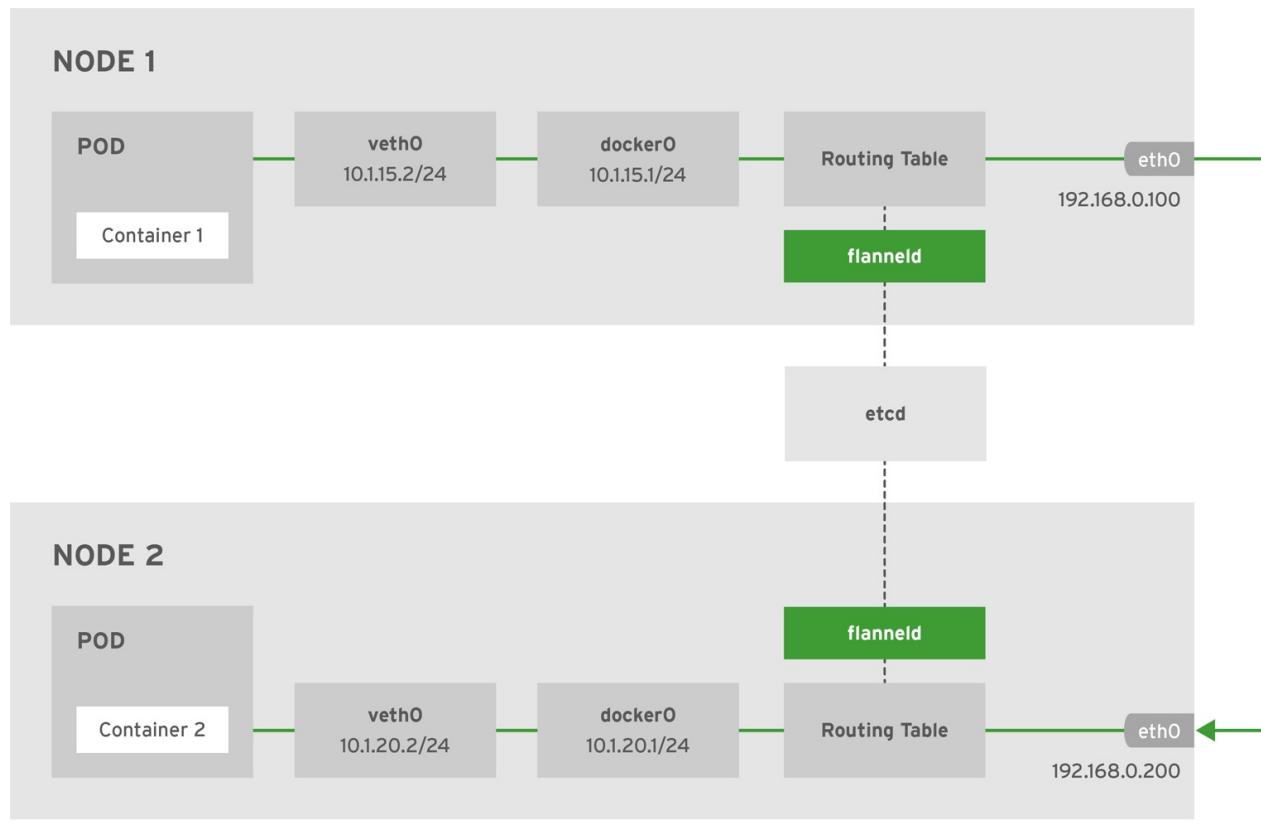
参考[OpenShift Flannel Architectrue](#)

Flannel的host-gw模式映射容器到容器的路由信息，kubernetes的每个node都会运行一个**flanneld**进程，它有以下几个职责：

- 为每个node分配一个独立的subnet
- 为每个pod分配一个独立的IP地址
- 映射容器到容器的路由信息，即便是不同主机上的容器

每个flanneld进程都会将信息发送到etcd集群中存储，这样每个node就都可以在flannel网络中获取容器的路由信息。

下图是flannel host-gw模式的架构图



OPENSIFT\_436034\_0317

Figure: arch

图片来源：[OpenShift Doc](#)

查看Node1和Node2的路由信息，你将会看到：

### Node1

```
default via 192.168.0.100 dev eth0 proto static metric 100  
10.1.15.0/24 dev docker0 proto kernel scope link src 10.1.15.1  
10.1.20.0/24 via 192.168.0.200 dev eth0
```

### Node2

```
default via 192.168.0.200 dev eth0 proto static metric 100  
10.1.20.0/24 dev docker0 proto kernel scope link src 10.1.20.1  
10.1.15.0/24 via 192.168.0.100 dev eth0
```



# 使用glusterfs做持久化存储

我们复用kubernetes的三台主机做glusterfs存储。

以下步骤参考自：<https://www.xf80.com/2017/04/21/kubernetes-glusterfs/>

## 安装glusterfs

我们直接在物理机上使用yum安装，如果你选择在kubernetes上安装，请参考：<https://github.com/gluster/gluster-kubernetes/blob/master/docs/setup-guide.md>

```
# 先安装 gluster 源
$ yum install centos-release-gluster -y

# 安装 glusterfs 组件
$ yum install -y glusterfs glusterfs-server glusterfs-fuse glust
erfs-rdma glusterfs-geo-replication glusterfs-devel

## 创建 glusterfs 目录
$ mkdir /opt/glusterd

## 修改 glusterd 目录
$ sed -i 's/var\/lib/opt/g' /etc/glusterfs/glusterd.vol

# 启动 glusterfs
$ systemctl start glusterd.service

# 设置开机启动
$ systemctl enable glusterd.service

# 查看状态
$ systemctl status glusterd.service
```

## 配置glusterfs

```
# 配置 hosts

$ vi /etc/hosts
172.20.0.113  sz-pg-oam-docker-test-001.tendcloud.com
172.20.0.114  sz-pg-oam-docker-test-002.tendcloud.com
172.20.0.115  sz-pg-oam-docker-test-003.tendcloud.com
```

```
# 开放端口
$ iptables -I INPUT -p tcp --dport 24007 -j ACCEPT

# 创建存储目录
$ mkdir /opt/gfs_data
```

```
# 添加节点到 集群
# 执行操作的本机不需要probe 本机
[root@sz-pg-oam-docker-test-001 ~]#
gluster peer probe sz-pg-oam-docker-test-002.tendcloud.com
gluster peer probe sz-pg-oam-docker-test-003.tendcloud.com

# 查看集群状态
$ gluster peer status
Number of Peers: 2

Hostname: sz-pg-oam-docker-test-002.tendcloud.com
Uuid: f25546cc-2011-457d-ba24-342554b51317
State: Peer in Cluster (Connected)

Hostname: sz-pg-oam-docker-test-003.tendcloud.com
Uuid: 42b6cad1-aa01-46d0-bbba-f7ec6821d66d
State: Peer in Cluster (Connected)
```

## 配置 volume

GlusterFS中的volume的模式有很多中，包括以下几种：

- 分布卷（默认模式）：即DHT, 也叫 分布卷: 将文件已hash算法随机分布到一

台服务器节点中存储。

- 复制模式：即AFR，创建volume时带 replica x 数量：将文件复制到 replica x 个节点中。
- 条带模式：即Striped，创建volume时带 stripe x 数量：将文件切割成数据块，分别存储到 stripe x 个节点中（类似raid 0）。
- 分布式条带模式：最少需要4台服务器才能创建。创建volume时 stripe 2 server = 4 个节点：是DHT与Striped的组合型。
- 分布式复制模式：最少需要4台服务器才能创建。创建volume时 replica 2 server = 4 个节点：是DHT与AFR的组合型。
- 条带复制卷模式：最少需要4台服务器才能创建。创建volume时 stripe 2 replica 2 server = 4 个节点：是 Striped 与 AFR 的组合型。
- 三种模式混合：至少需要8台服务器才能创建。stripe 2 replica 2，每4个节点组成一个组。

这几种模式的示例图参考：[CentOS7安装GlusterFS](#)。

因为我们只有三台主机，在此我们使用默认的分布卷模式。请勿在生产环境上使用该模式，容易导致数据丢失。

```
# 创建分布卷
$ gluster volume create k8s-volume transport tcp sz-pg-oam-docker-test-001.tendcloud.com:/opt/gfs_data sz-pg-oam-docker-test-002.tendcloud.com:/opt/gfs_data sz-pg-oam-docker-test-003.tendcloud.com:/opt/gfs_data force

# 查看volume状态
$ gluster volume info
Volume Name: k8s-volume
Type: Distribute
Volume ID: 9a3b0710-4565-4eb7-abae-1d5c8ed625ac
Status: Created
Snapshot Count: 0
Number of Bricks: 3
Transport-type: tcp
Bricks:
Brick1: sz-pg-oam-docker-test-001.tendcloud.com:/opt/gfs_data
Brick2: sz-pg-oam-docker-test-002.tendcloud.com:/opt/gfs_data
Brick3: sz-pg-oam-docker-test-003.tendcloud.com:/opt/gfs_data
Options Reconfigured:
transport.address-family: inet
nfs.disable: on

# 启动 分布卷
$ gluster volume start k8s-volume
```

## Glusterfs调优

```
# 开启 指定 volume 的配额
$ gluster volume quota k8s-volume enable

# 限制 指定 volume 的配额
$ gluster volume quota k8s-volume limit-usage / 1TB

# 设置 cache 大小, 默认32MB
$ gluster volume set k8s-volume performance.cache-size 4GB

# 设置 io 线程, 太大会导致进程崩溃
$ gluster volume set k8s-volume performance.io-thread-count 16

# 设置 网络检测时间, 默认42s
$ gluster volume set k8s-volume network.ping-timeout 10

# 设置 写缓冲区的大小, 默认1M
$ gluster volume set k8s-volume performance.write-behind-window-size 1024MB
```

## Kubernetes中配置glusterfs

官方的文档

见：<https://github.com/kubernetes/kubernetes/tree/master/examples/volumes/glusterfs>

以下用到的所有yaml和json配置文件可以在[glusterfs](#)中找到。注意替换其中私有镜像地址为自己的镜像地址。

## kubernetes安装客户端

```
# 在所有 k8s node 中安装 glusterfs 客户端  
  
$ yum install -y glusterfs glusterfs-fuse  
  
# 配置 hosts  
  
$ vi /etc/hosts  
  
172.20.0.113    sz-pg-oam-docker-test-001.tendcloud.com  
172.20.0.114    sz-pg-oam-docker-test-002.tendcloud.com  
172.20.0.115    sz-pg-oam-docker-test-003.tendcloud.com
```

因为我们glusterfs跟kubernetes集群复用主机，因此这一步可以省去。

## 配置 endpoints

```
$ curl -o https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/volumes/glusterfs/glusterfs-endpoints.json

# 修改 endpoints.json , 配置 glusters 集群节点ip
# 每一个 addresses 为一个 ip 组

{
  "addresses": [
    {
      "ip": "172.22.0.113"
    }
  ],
  "ports": [
    {
      "port": 1990
    }
  ]
},

# 导入 glusterfs-endpoints.json

$ kubectl apply -f glusterfs-endpoints.json

# 查看 endpoints 信息
$ kubectl get ep
```

## 配置 service

```
$ curl -o https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/volumes/glusterfs/glusterfs-service.json

# service.json 里面查找的是 endpoints 的名称与端口，端口默认配置为 1，我改成了1990

# 导入 glusterfs-service.json
$ kubectl apply -f glusterfs-service.json

# 查看 service 信息
$ kubectl get svc
```

## 创建测试 pod

```
$ curl -o https://raw.githubusercontent.com/kubernetes/kubernetes/master/examples/volumes/glusterfs/glusterfs-pod.json

# 编辑 glusterfs-pod.json
# 修改 volumes 下的 path 为上面创建的 volume 名称

"path": "k8s-volume"

# 导入 glusterfs-pod.json
$ kubectl apply -f glusterfs-pod.json

# 查看 pods 状态
$ kubectl get pods
NAME                      READY   STATUS    RESTARTS
AGE
glusterfs                 1/1     Running   0
1m

# 查看 pods 所在 node
$ kubectl describe pods/glusterfs

# 登陆 node 物理机，使用 df 可查看挂载目录
$ df -h
172.20.0.113:k8s-volume 1073741824          0 1073741824  0% 172.
20.0.113:k8s-volume  1.0T      0  1.0T    0% /var/lib/kubelet/pods
/3de9fc69-30b7-11e7-bfbd-8af1e3a7c5bd/volumes/kubernetes.io~glus
terfs/glusterfsvol
```

## 配置 PersistentVolume

PersistentVolume (PV) 和 PersistentVolumeClaim (PVC) 是 Kubernetes 提供的两种 API 资源，用于抽象存储细节。管理员关注于如何通过 PV 提供存储功能而无需关注用户如何使用，同样的用户只需要挂载 PVC 到容器中而不需要关注存储卷采用何种技术实现。

PVC 和 PV 的关系跟 pod 和 node 关系类似，前者消耗后者的资源。PVC 可以向 PV 申请指定大小的存储资源并设置访问模式。

## PV属性

- storage容量
- 读写属性：分别为ReadWriteOnce：单个节点读写； ReadOnlyMany：多节点只读； ReadWriteMany：多节点读写

```
$ cat glusterfs-pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: gluster-dev-volume
spec:
  capacity:
    storage: 8Gi
  accessModes:
    - ReadWriteMany
  glusterfs:
    endpoints: "glusterfs-cluster"
    path: "k8s-volume"
    readOnly: false

# 导入PV
$ kubectl apply -f glusterfs-pv.yaml

# 查看 pv
$ kubectl get pv
NAME          CAPACITY   ACCESSMODES   RECLAIMPOLICY   ST
ATUS          CLAIM      STORAGECLASS  REASON        AGE
gluster-dev-volume   8Gi        RWX           Retain       Av
ailable                  3s
```

## PVC属性

- 访问属性与PV相同
- 容量：向PV申请的容量  $\leq$  PV总容量

## 配置PVC

```
$ cat glusterfs-pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: glusterfs-nginx
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 8Gi

# 导入 pvc
$ kubectl apply -f glusterfs-pvc.yaml

# 查看 pvc

$ kubectl get pv
NAME           STATUS    VOLUME          CAPACITY   ACCE
SSMODES        STORAGECLASS AGE
glusterfs-nginx Bound    gluster-dev-volume 8Gi       RWX
                           4s
```

## 创建 nginx deployment 挂载 volume

```
$ vi nginx-deployment.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-dm
spec:
  replicas: 2
  template:
    metadata:
      labels:
        name: nginx
  spec:
    containers:
```

```
- name: nginx
  image: nginx:alpine
  imagePullPolicy: IfNotPresent
  ports:
    - containerPort: 80
  volumeMounts:
    - name: gluster-dev-volume
      mountPath: "/usr/share/nginx/html"
  volumes:
    - name: gluster-dev-volume
      persistentVolumeClaim:
        claimName: glusterfs-nginx

# 导入 deployment
$ kubectl apply -f nginx-deployment.yaml

# 查看 deployment
$ kubectl get pods |grep nginx-dm
nginx-dm-3698525684-g0mvt      1/1      Running   0          6
s
nginx-dm-3698525684-hbzq1      1/1      Running   0          6
s

# 查看 挂载
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- df -h|grep k8s-volume
172.20.0.113:k8s-volume       1.0T      0  1.0T  0% /usr/share
/nginx/html

# 创建文件 测试
$ kubectl exec -it nginx-dm-3698525684-g0mvt -- touch /usr/share
/nginx/html/index.html

$ kubectl exec -it nginx-dm-3698525684-g0mvt -- ls -lt /usr/shar
e/nginx/html/index.html
-rw-r--r-- 1 root root 0 May  4 11:36 /usr/share/nginx/html/inde
x.html

# 验证 glusterfs
# 因为我们使用分布卷，所以可以看到某个节点中有文件
```

```
[root@sz-pg-oam-docker-test-001 ~] ls /opt/gfs_data/  
[root@sz-pg-oam-docker-test-002 ~] ls /opt/gfs_data/  
index.html  
[root@sz-pg-oam-docker-test-003 ~] ls /opt/gfs_data/
```

## 参考

[在kubernetes中安装glusterfs](#)

[CentOS 7 安装 GlusterFS](#)

[GlusterFS with kubernetes](#)

for GitBook      update 2017-05-12 16:45:40

## 服务滚动升级

当有镜像发布新版本，新版本服务上线时如何实现服务的滚动和平滑升级？

如果你使用 **ReplicationController** 创建的 pod 可以使用 `kubectl rollingupdate` 命令滚动升级，如果使用的是 **Deployment** 创建的 Pod 可以直接修改 yaml 文件后执行 `kubectl apply` 即可。

**Deployment** 已经内置了 **RollingUpdate strategy**，因此不用再调用 `kubectl rollingupdate` 命令，升级的过程是先创建新版的 pod 将流量导入到新 pod 上后销毁原来的旧的 pod。

**Rolling Update** 适用于 **Deployment**、**Replication Controller**，官方推荐使用 **Deployment** 而不再使用 **Replication Controller**。

使用 **ReplicationController** 时的滚动升级请参考官网说

明：<https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/>

## ReplicationController与Deployment的关系

**ReplicationController** 和 **Deployment** 的 **RollingUpdate** 命令有些不同，但是实现的机制是一样的，关于这两个 kind 的关系我引用了 [ReplicationController与Deployment的区别](#) 中的部分内容如下，详细区别请查看原文。

## ReplicationController

**Replication Controller** 为 **Kubernetes** 的一个核心内容，应用托管到 **Kubernetes** 之后，需要保证应用能够持续的运行，**Replication Controller** 就是这个保证的 key，主要的功能如下：

- 确保 pod 数量：它会确保 **Kubernetes** 中有指定数量的 Pod 在运行。如果少于指定数量的 pod，**Replication Controller** 会创建新的，反之则会删除掉多余的以保证 Pod 数量不变。
- 确保 pod 健康：当 pod 不健康，运行出错或者无法提供服务时，**Replication Controller** 也会杀死不健康的 pod，重新创建新的。

- 弹性伸缩：在业务高峰或者低峰期的时候，可以通过Replication Controller动态的调整pod的数量来提高资源的利用率。同时，配置相应的监控功能（Horizontal Pod Autoscaler），会定时自动从监控平台获取Replication Controller关联pod的整体资源使用情况，做到自动伸缩。
- 滚动升级：滚动升级为一种平滑的升级方式，通过逐步替换的策略，保证整体系统的稳定，在初始化升级的时候就可以及时发现和解决问题，避免问题不断扩大。

## Deployment

Deployment同样为Kubernetes的一个核心内容，主要职责同样是为了保证pod的数量和健康，90%的功能与Replication Controller完全一样，可以看做新一代的Replication Controller。但是，它又具备了Replication Controller之外的新特性：

- Replication Controller全部功能：Deployment继承了上面描述的Replication Controller全部功能。
- 事件和状态查看：可以查看Deployment的升级详细进度和状态。
- 回滚：当升级pod镜像或者相关参数的时候发现问题，可以使用回滚操作回滚到上一个稳定的版本或者指定的版本。
- 版本记录：每一次对Deployment的操作，都能保存下来，给予后续可能的回滚使用。
- 暂停和启动：对于每一次升级，都能够随时暂停和启动。
- 多种升级方案：Recreate：删除所有已存在的pod,重新创建新的；  
RollingUpdate：滚动升级，逐步替换的策略，同时滚动升级时，支持更多的附加参数，例如设置最大不可用pod数量，最小升级间隔时间等等。

## 创建测试镜像

我们来创建一个特别简单的web服务，当你访问网页时，将输出一句版本信息。通过区分这句版本信息输出我们就可以断定升级是否完成。

所有配置和代码见[manifests/test/rolling-update-test](#)目录。

Web服务的代码[main.go](#)

```

package main

import (
    "fmt"
    "log"
    "net/http"
)

func sayhello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "This is version 1.") //这个写入到w的是输出到客户
    端的
}

func main() {
    http.HandleFunc("/", sayhello) //设置访问的路由
    log.Println("This is version 1.")
    err := http.ListenAndServe(":9090", nil) //设置监听的端口
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}

```

## 创建Dockerfile

```

FROM alpine:3.5
MAINTAINER Jimmy Song<rootsongjc@gmail.com>
ADD hellov2 /
ENTRYPOINT ["/hellov2"]

```

注意修改添加的文件的名称。

## 创建Makefile

修改镜像仓库的地址为你自己的私有镜像仓库地址。

修改 `Makefile` 中的 `TAG` 为新的版本号。

```
all: build push clean
.PHONY: build push clean

TAG = v1

# Build for linux amd64
build:
    GOOS=linux GOARCH=amd64 go build -o hello${TAG} main.go
    docker build -t sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:${TAG} .

# Push to tenxcloud
push:
    docker push sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:${TAG}

# Clean
clean:
    rm -f hello${TAG}
```

编译

```
make all
```

分别修改main.go中的输出语句、Dockerfile中的文件名称和Makefile中的TAG，创建两个版本的镜像。

## 测试

我们使用Deployment部署服务来测试。

配置文件 rolling-update-test.yaml :

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: rolling-update-test
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: rolling-update-test
    spec:
      containers:
        - name: rolling-update-test
          image: sz-pg-oam-docker-hub-001.tendcloud.com/library/he
llo:v1
      ports:
        - containerPort: 9090
    ---
apiVersion: v1
kind: Service
metadata:
  name: rolling-update-test
  labels:
    app: rolling-update-test
spec:
  ports:
    - port: 9090
      protocol: TCP
      name: http
  selector:
    app: rolling-update-test
```

### 部署**service**

```
kubectl create -f rolling-update-test.yaml
```

### 修改**traefik ingress**配置

在 `ingress.yaml` 文件中增加新service的配置。

```
- host: rolling-update-test.traefik.io
  http:
    paths:
      - path: /
        backend:
          serviceName: rolling-update-test
          servicePort: 9090
```

修改本地的host配置，增加一条配置：

```
172.20.0.119 rolling-update-test.traefik.io
```

注意：172.20.0.119是我们之前使用keepalived创建的VIP。

打开浏览器访问<http://rolling-update-test.traefik.io>将会看到以下输出：

```
This is version 1.
```

### 滚动升级

只需要将 `rolling-update-test.yaml` 文件中的 `image` 改成新版本的镜像名，然后执行：

```
kubectl apply -f rolling-update-test.yaml
```

也可以参考[Kubernetes Deployment Concept](#)中的方法，直接设置新的镜像。

```
kubectl set image deployment/rolling-update-test rolling-update-test=sz-pg-oam-docker-hub-001.tendcloud.com/library/hello:v2
```

或者使用 `kubectl edit deployment/rolling-update-test` 修改镜像名称后保存。

使用以下命令查看升级进度：

```
kubectl rollout status deployment/rolling-update-test
```

升级完成后在浏览器中刷新<http://rolling-update-test.traefik.io>将会看到以下输出：

```
This is version 2.
```

说明滚动升级成功。

## 使用**ReplicationController**创建的Pod如何RollingUpdate

以上讲解使用**Deployment**创建的Pod的RollingUpdate方式，那么如果使用传统的**ReplicationController**创建的Pod如何Update呢？

举个例子：

```
$ kubectl -n spark-cluster rolling-update zeppelin-controller --image sz-pg-oam-docker-hub-001.tendcloud.com/library/zeppelin:0.7.1
Created zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b
Scaling up zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b from 0 to 1, scaling down zeppelin-controller from 1 to 0 (keep 1 pods available, don't exceed 2 pods)
Scaling zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b up to 1
Scaling zeppelin-controller down to 0
Update succeeded. Deleting old controller: zeppelin-controller
Renaming zeppelin-controller-99be89dbbe5cd5b8d6feab8f57a04a8b to zeppelin-controller
replicationcontroller "zeppelin-controller" rolling updated
```

只需要指定新的镜像即可，当然你可以配置RollingUpdate的策略。

## 参考

## Rolling update机制解析

Running a Stateless Application Using a Deployment

Simple Rolling Update

使用kubernetes的deployment进行RollingUpdate

for GitBook      update 2017-05-12 16:45:40

## 问题记录

安装、使用kubernetes的过程中遇到的所有问题的记录。

推荐直接在Kubernetes的GitHub上[提issue](#)，在此记录所提交的issue。

### 1. Failed to start ContainerManager failed to initialise top level QOS containers #43856

重启kubelet时报错，目前的解决方法是：

1. 在docker.service配置中增加的 `--exec-opt native.cgroupdriver=systemd` 配置。
2. 手动删除slice（貌似不管用）
3. 重启主机，这招最管用☺

```
for i in $(systemctl list-unit-files --no-legend --no-pager -l | grep --color=never -o *.slice | grep kubepod); do systemctl stop $i; done
```

上面的几种方法在该bug修复前只有重启主机管用，该bug已于2017年4月27日修复，merge到了master分支，见

<https://github.com/kubernetes/kubernetes/pull/44940>

### 2. High Availability of Kube-apiserver #19816

API server的HA如何实现？或者说这个master节点上的服务 `api-server`、`scheduler`、`controller` 如何实现HA？目前的解决方案是什么？

目前的解决方案是api-server是无状态的可以启动多个，然后在前端再加一个nginx或者ha-proxy。而scheduler和controller都是直接用容器的方式启动的。

### 3.Kubelet启动时Failed to start ContainerManager systemd version does not support ability to start a slice as transient unit

CentOS系统版本7.2.1511

kubelet启动时报错systemd版本不支持start a slice as transient unit。

尝试升级CentOS版本到7.3，看看是否可以修复该问题。

与[kubeadm init waiting for the control plane to become ready on CentOS 7.2 with kubeadm 1.6.1 #228](#)类似。

另外有一个使用systemd管理kubelet的[proposal](#)。

### 4.kube-proxy报错kube-proxy[2241]: E0502 15:55:13.889842 2241 conntrack.go:42] conntrack returned error: error looking for path of conntrack: exec: "conntrack": executable file not found in \$PATH

导致的现象

kubedns启动成功，运行正常，但是service之间无法解析，kubernetes中的DNS解析异常

解决方法

CentOS中安装 conntrack-tools 包后重启kubernetes集群即可。

### 5. Pod stuck in terminating if it has a privileged container but has been scheduled to a node which doesn't allow privilege issue#42568

当pod被调度到无法权限不足的node上时，pod一直处于pending状态，且无法删除pod，删除时一直处于terminating状态。

## kubelet中的报错信息

```
Error validating pod kube-keepalived-vip-1p62d_default(5d79ccc0-3173-11e7-bfbd-8af1e3a7c5bd) from api, ignoring: spec.containers[0].securityContext.privileged: Forbidden: disallowed by cluster policy
```

## 6.PVC中对Storage的容量设置不生效

使用[glusterfs做持久化存储](#)文档中我们构建了PV和PVC，当时给 `glusterfs-nginx` 的PVC设置了8G的存储限额，`nginx-dm` 这个Deployment使用了该PVC，进入该Deployment中的Pod执行测试：

```
dd if=/dev/zero of=test bs=1G count=10
```

```
root@nginx-dm-3698525684-g0mvt:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/docker-8:20-2513719-68fc56b9d12f454a80ef69105f2dc6a42ae8ba8132d51764dbf2710b7da6962e  10G  228M  9.8G  3% /
tmpfs          63G    0   63G   0% /dev
tmpfs          63G    0   63G   0% /sys/fs/cgroup
/dev/sdb4       2.0T  28G  2.0T  2% /etc/hosts
shm            64M    0   64M   0% /dev/shm
172.20.0.113:k8s-volume  1.0T    0  1.0T   0% /usr/share/nginx/html
tmpfs          63G   12K  63G   1% /run/secrets/kubernetes.io
/serviceaccount
root@nginx-dm-3698525684-g0mvt:~# dd if=/dev/zero of=test bs=1G count=11
dd: error writing 'test': No space left on device
10+0 records in
9+0 records out
10486870016 bytes (10 GB) copied, 14.6692 s, 715 MB/s
root@nginx-dm-3698525684-g0mvt:~#
```

*Figure: pvc-storage-limit*

从截图中可以看到创建了9个size为1G的block后无法继续创建了，已经超出了8G的限额。

## 参考

[Persistent Volume](#)

[Resource Design Proposals](#)

for GitBook      update 2017-05-12 16:45:40