手动搭建高可用的kubernetes 集群

之前按照和我一步步部署 kubernetes 集群的步骤一步一步的成功的使用二进制的方式安装了`kubernetes`集群，在该文档的基础上重新部署了最新的`v1.8.2`版本，实现了`kube-apiserver`的高可用、`traefik ingress`的部署、在`kubernetes`上安装`docker`的私有仓库`harbor`、容器化`kubernetes`部分组建、使用阿里云日志服务收集日志。
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。
本系列系文档适用于 `CentOS 7`、`Ubuntu 16.04` 及以上版本系统，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**，否则可能会认证、授权等失败！

目录

1. 组件版本 && 集群环境

组件版本

- Kubernetes 1.8.2
- Docker 17.10.0-ce
- Etcd 3.2.9
- Flanneld
- TLS 认证通信（所有组件，如etcd、kubernetes master 和node）
- RBAC 授权
- kubelet TLS Bootstrapping
- kubedns、dashboard、heapster等插件
- harbor，使用nfs后端存储

etcd 集群 && k8s master 机器 && k8s node 机器

- master01：192.168.1.137
- master02：192.168.1.138
- master03/node03：192.168.1.170
- 由于机器有限，所以我们将master03 也作为node 节点，后续有新的机器增加即可
- node01: 192.168.1.161
- node02: 192.168.1.162

集群环境变量

后面的嗯部署将会使用到的全局变量，定义如下（根据自己的机器、网络修改）：

```
# TLS Bootstrapping 使用的Token，可以使用命令 head -c 16 /dev/urandom | od -An -t x | tr -d
' ' 生成
BOOTSTRAP_TOKEN="8981b594122ebed7596f1d3b69c78223"

# 建议使用未用的网段来定义服务网段和Pod 网段
# 服务网段(Service CIDR)，部署前路由不可达，部署后集群内部使用IP:Port可达
SERVICE_CIDR="10.254.0.0/16"
# Pod 网段(Cluster CIDR)，部署前路由不可达，部署后路由可达(flanneld 保证)
CLUSTER_CIDR="172.30.0.0/16"

# 服务端口范围(NodePort Range)
NODE_PORT_RANGE="30000-32766"

# etcd集群服务地址列表
ETCD_ENDPOINTS="https://192.168.1.137:2379,https://192.168.1.138:2379,https://192.168.1.1


# flanneld 网络配置前缀
FLANNEL_ETCD_PREFIX="/kubernetes/network"

# kubernetes 服务IP(预先分配，一般为SERVICE_CIDR中的第一个IP)
CLUSTER_KUBERNETES_SVC_IP="10.254.0.1"

# 集群 DNS 服务IP(从SERVICE_CIDR 中预先分配)
CLUSTER_DNS_SVC_IP="10.254.0.2"

# 集群 DNS 域名
CLUSTER_DNS_DOMAIN="cluster.local."

# MASTER API Server 地址
MASTER_URL="k8s-api.virtual.local"
```

将上面变量保存为: **env.sh**，然后将脚本拷贝到所有机器的`/usr/k8s/bin`目录。

为方便后面迁移，我们在集群内定义一个域名用于访问`apiserver`，在每个节点的`/etc/hosts`文件中添加记录：**192.168.1.137 k8s-api.virtual.local k8s-api**

其中`192.168.1.137`为master01 的IP，暂时使用该IP 来做apiserver 的负载地址

> *如果你使用的是阿里云的ECS 服务，强烈建议你先将上述节点的安全组配置成允许所有访问，不然在安装过程中会遇到各种访问不了的问题，待集群配置成功以后再根据需要添加安全限制。*

2. 创建CA 证书和密钥

`kubernetes` 系统各个组件需要使用`TLS`证书对通信进行加密，这里我们使用`CloudFlare`的PKI 工具集cfssl 来生成Certificate Authority(CA) 证书和密钥文件， CA 是自签名的证书，用来签名后续创建的其他TLS 证书。

安装 CFSSL

```
$ wget https://pkg.cfssl.org/R1.2/cfssl_linux-amd64
$ chmod +x cfssl_linux-amd64
$ sudo mv cfssl_linux-amd64 /usr/k8s/bin/cfssl

$ wget https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
$ chmod +x cfssljson_linux-amd64
$ sudo mv cfssljson_linux-amd64 /usr/k8s/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 /usr/k8s/bin/cfssl-certinfo
```

```
$ export PATH=/usr/k8s/bin:$PATH
$ mkdir ssl && cd ssl
$ cfssl print-defaults config > config.json
$ cfssl print-defaults csr > csr.json
```

为了方便，将`/usr/k8s/bin`设置成环境变量，为了重启也有效，可以将上面的`export PATH=/usr/k8s/bin:$PATH`添加到`/etc/rc.local`文件中。

创建CA

修改上面创建的`config.json`文件为`ca-config.json`：

```
$ cat ca-config.json
{
    "signing": {
        "default": {
            "expiry": "8760h"
        },
        "profiles": {
            "kubernetes": {
                "expiry": "8760h",
                "usages": [
                    "signing",
                    "key encipherment",
                    "server auth",
                    "client auth"
                ]
            }
        }
    }
}
```

- `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 提供的证书进行校验；
- `client auth`: 表示server 可以用该CA 对client 提供的证书进行验证。

修改CA 证书签名请求为`ca-csr.json`：

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

分发证书

将生成的CA 证书、密钥文件、配置文件拷贝到所有机器的`/etc/kubernetes/ssl`目录下面：

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

3. 部署高可用etcd 集群

kubernetes 系统使用`etcd`存储所有的数据，我们这里部署3个节点的etcd 集群，这3个节点直接复用kubernetes master的3个节点，分别命名为`etcd01`、`etcd02`、`etcd03`:

- etcd01：192.168.1.137
- etcd02：192.168.1.138
- etcd03：192.168.1.170

定义环境变量

使用到的变量如下：

```
$ export NODE_NAME=etcd01 # 当前部署的机器名称(随便定义，只要能区分不同机器即可)
$ export NODE_IP=192.168.1.137 # 当前部署的机器IP
$ export NODE_IPS="192.168.1.137 192.168.1.138 192.168.1.170" # etcd 集群所有机器 IP
$ # etcd 集群间通信的IP和端口
$ export
ETCD_NODES=etcd01=https://192.168.1.137:2380,etcd02=https://192.168.1.138:2380,etcd03=htt

$ # 导入用到的其它全局变量：ETCD_ENDPOINTS、FLANNEL_ETCD_PREFIX、CLUSTER_CIDR
$ source /usr/k8s/bin/env.sh
```

下载etcd 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：

```
$ wget https://github.com/coreos/etcd/releases/download/v3.2.9/etcd-v3.2.9-linux-
amd64.tar.gz
$ tar -xvf etcd-v3.2.9-linux-amd64.tar.gz
$ sudo mv etcd-v3.2.9-linux-amd64/etcd* /usr/k8s/bin/
```

创建TLS 密钥和证书

为了保证通信安全，客户端(如etcdctl)与etcd 集群、etcd 集群之间的通信需要使用TLS 加密。

创建etcd 证书签名请求：

```
$ cat > etcd-csr.json <<EOF
{
  "CN": "etcd",
  "hosts": [
    "127.0.0.1",
    "${NODE_IP}"
  ],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
      "O": "k8s",
      "OU": "System"
    }
  ]
```

```
}
EOF
```

- hosts 字段指定授权使用该证书的etcd节点IP

生成etcd证书和私钥：

```
$ cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem \
  -ca-key=/etc/kubernetes/ssl/ca-key.pem \
  -config=/etc/kubernetes/ssl/ca-config.json \
  -profile=kubernetes etcd-csr.json | cfssljson -bare etcd
$ ls etcd*
etcd.csr  etcd-csr.json  etcd-key.pem  etcd.pem
$ sudo mkdir -p /etc/etcd/ssl
$ sudo mv etcd*.pem /etc/etcd/ssl/
```

创建etcd 的systemd unit 文件

```
$ sudo mkdir -p /var/lib/etcd  # 必须要先创建工作目录
$ cat > etcd.service <<EOF
[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/
ExecStart=/usr/k8s/bin/etcd \\
  --name=${NODE_NAME} \\
  --cert-file=/etc/etcd/ssl/etcd.pem \\
  --key-file=/etc/etcd/ssl/etcd-key.pem \\
  --peer-cert-file=/etc/etcd/ssl/etcd.pem \\
  --peer-key-file=/etc/etcd/ssl/etcd-key.pem \\
  --trusted-ca-file=/etc/kubernetes/ssl/ca.pem \\
  --peer-trusted-ca-file=/etc/kubernetes/ssl/ca.pem \\
  --initial-advertise-peer-urls=https://${NODE_IP}:2380 \\
  --listen-peer-urls=https://${NODE_IP}:2380 \\
  --listen-client-urls=https://${NODE_IP}:2379,http://127.0.0.1:2379 \\
  --advertise-client-urls=https://${NODE_IP}:2379 \\
  --initial-cluster-token=etcd-cluster-0 \\
  --initial-cluster=${ETCD_NODES} \\
  --initial-cluster-state=new \\
  --data-dir=/var/lib/etcd
Restart=on-failure
RestartSec=5
LimitNOFILE=65536

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

- 指定etcd的工作目录和数据目录为/var/lib/etcd，需要在启动服务前创建这个目录；
- 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)；
- --initial-cluster-state值为new时，--name的参数值必须位于--initial-cluster列表中；

启动etcd 服务

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

```
$ sudo systemctl enable etcd
$ sudo systemctl start etcd
$ sudo systemctl status etcd
```

最先启动的etcd 进程会卡住一段时间，等待其他节点启动加入集群，在所有的etcd 节点重复上面的步骤，直到所有的机器etcd 服务都已经启动。

验证服务

部署完etcd 集群后，在任一etcd 节点上执行下面命令：

```
for ip in ${NODE_IPS}; do
  ETCDCTL_API=3 /usr/k8s/bin/etcdctl \
  --endpoints=https://${ip}:2379  \
  --cacert=/etc/kubernetes/ssl/ca.pem \
  --cert=/etc/etcd/ssl/etcd.pem \
  --key=/etc/etcd/ssl/etcd-key.pem \
  endpoint health; done
```

输出如下结果：

```
https://192.168.1.137:2379 is healthy: successfully committed proposal: took = 1.509032ms
https://192.168.1.138:2379 is healthy: successfully committed proposal: took = 1.639228ms
https://192.168.1.170:2379 is healthy: successfully committed proposal: took = 1.4152ms
```

可以看到上面的信息3个节点上的etcd 均为**healthy**，则表示集群服务正常。

4. 配置kubectl 命令行工具

`kubectl`默认从`~/.kube/config`配置文件中获取访问kube-apiserver 地址、证书、用户名等信息，需要正确配置该文件才能正常使用`kubectl`命令。

需要将下载的kubectl 二进制文件和生产的`~/.kube/config`配置文件拷贝到需要使用kubectl 命令的机器上。

环境变量

```
$ source /usr/k8s/bin/env.sh
$ export KUBE_APISERVER="https://{{MASTER_URL}}"
```

- 变量KUBE_APISERVER 指定kubelet 访问的kube-apiserver 的地址，后续被写入`~/.kube/config`配置文件

下载kubectl

```
$ wget https://dl.k8s.io/v1.8.2/kubernetes-client-linux-amd64.tar.gz # 如果服务器上下载不下来，可以想办法下载到本地，然后scp上去即可
$ tar -xzvf kubernetes-client-linux-amd64.tar.gz
$ sudo cp kubernetes/client/bin/kube* /usr/k8s/bin/
$ sudo chmod a+x /usr/k8s/bin/kube*
$ export PATH=/usr/k8s/bin:$PATH
```

创建admin 证书

kubectl 与kube-apiserver 的安全端口通信，需要为安全通信提供TLS 证书和密钥。创建admin 证书签名请求：

```
$ cat > admin-csr.json <<EOF
{
  "CN": "admin",
  "hosts": [],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "BeiJing",
      "L": "BeiJing",
```

```
      "O": "system:masters",
      "OU": "System"
    }
  ]
}
EOF
```

- 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所有API 的权限
- O 指定了该证书的Group 为system:masters，kubectl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 的劝降
- hosts 属性值为空列表

生成admin 证书和私钥：

```
$ cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem \
  -ca-key=/etc/kubernetes/ssl/ca-key.pem \
  -config=/etc/kubernetes/ssl/ca-config.json \
  -profile=kubernetes admin-csr.json | cfssljson -bare admin
$ ls admin
admin.csr  admin-csr.json  admin-key.pem  admin.pem
$ sudo mv admin*.pem /etc/kubernetes/ssl/
```

创建kubectl kubeconfig 文件

```
# 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 \
  --token=${BOOTSTRAP_TOKEN}
# 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O 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

分发kubeconfig 文件

将~/.kube/config文件拷贝到运行kubectl命令的机器的~/.kube/目录下去。

5. 部署Flannel 网络

kubernetes 要求集群内各节点能通过Pod 网段互联互通，下面我们来使用Flannel 在所有节点上创建互联互通的Pod 网段的步骤。

环境变量

```
$ export NODE_IP=192.168.1.137  # 当前部署节点的IP
# 导入全局变量
$ source /usr/k8s/bin/env.sh
```

创建TLS 密钥和证书

etcd 集群启用了双向TLS 认证，所以需要为flanneld 指定与etcd 集群通信的CA 和密钥。

创建flanneld 证书签名请求：

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

生成flanneld 证书和私钥：

```
$ cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem \
  -ca-key=/etc/kubernetes/ssl/ca-key.pem \
  -config=/etc/kubernetes/ssl/ca-config.json \
  -profile=kubernetes flanneld-csr.json | cfssljson -bare flanneld
$ ls flanneld*
flanneld.csr  flanneld-csr.json  flanneld-key.pem flanneld.pem
$ sudo mkdir -p /etc/flanneld/ssl
$ sudo mv flanneld*.pem /etc/flanneld/ssl
```

向etcd 写入集群Pod 网段信息

该步骤只需在第一次部署Flannel 网络时执行，后续在其他节点上部署Flanneld 时无需再写入该信息

```
$ etcdctl \
  --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/flanneld/ssl/flanneld.pem \
  --key-file=/etc/flanneld/ssl/flanneld-key.pem \
  set ${FLANNEL_ETCD_PREFIX}/config '{"Network":"'${CLUSTER_CIDR}'", "SubnetLen": 24,
"Backend": {"Type": "vxlan"}}'
# 得到如下反馈信息
{"Network":"172.30.0.0/16", "SubnetLen": 24, "Backend": {"Type": "vxlan"}}
```

- 写入的 Pod 网段(${CLUSTER_CIDR}，172.30.0.0/16) 必须与`kube-controller-manager` 的 `--cluster-cidr` 选项值一致；

安装和配置flanneld

前往flanneld release页面下载最新版的flanneld 二进制文件：

```
$ mkdir flannel
$ wget https://github.com/coreos/flannel/releases/download/v0.9.0/flannel-v0.9.0-linux-
amd64.tar.gz
$ tar -xzvf flannel-v0.9.0-linux-amd64.tar.gz -C flannel
$ sudo cp flannel/{flanneld,mk-docker-opts.sh} /usr/k8s/bin
```

创建flanneld的systemd unit 文件

```
$ cat > flanneld.service << EOF
[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
ExecStart=/usr/k8s/bin/flanneld \\
  -etcd-cafile=/etc/kubernetes/ssl/ca.pem \\
  -etcd-certfile=/etc/flanneld/ssl/flanneld.pem \\
  -etcd-keyfile=/etc/flanneld/ssl/flanneld-key.pem \\
  -etcd-endpoints=${ETCD_ENDPOINTS} \\
  -etcd-prefix=${FLANNEL_ETCD_PREFIX}
ExecStartPost=/usr/k8s/bin/mk-docker-opts.sh -k DOCKER_NETWORK_OPTIONS -d
/run/flannel/docker
Restart=on-failure

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

- `mk-docker-opts.sh`脚本将分配给flanneld 的Pod 子网网段信息写入到`/run/flannel/docker` 文件中，后续docker 启动时使用这个文件中的参数值为 docker0 网桥
- flanneld 使用系统缺省路由所在的接口和其他节点通信，对于有多个网络接口的机器(内网和公网)，可以用 `--iface` 选项值指定通信接口(上面的 systemd unit 文件没指定这个选项)

启动flanneld

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

检查flanneld 服务

```
ifconfig flannel.1
```

检查分配给各flanneld 的Pod 网段信息

```
$ # 查看集群 Pod 网段(/16)
$ etcdctl \
  --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/flanneld/ssl/flanneld.pem \
  --key-file=/etc/flanneld/ssl/flanneld-key.pem \
  get ${FLANNEL_ETCD_PREFIX}/config
{ "Network": "172.30.0.0/16", "SubnetLen": 24, "Backend": { "Type": "vxlan" } }
$ # 查看已分配的 Pod 子网段列表(/24)
$ etcdctl \
  --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/flanneld/ssl/flanneld.pem \
  --key-file=/etc/flanneld/ssl/flanneld-key.pem \
  ls ${FLANNEL_ETCD_PREFIX}/subnets
/kubernetes/network/subnets/172.30.77.0-24
$ # 查看某一 Pod 网段对应的 flanneld 进程监听的 IP 和网络参数
$ etcdctl \
  --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
```

```
  --cert-file=/etc/flanneld/ssl/flanneld.pem \
  --key-file=/etc/flanneld/ssl/flanneld-key.pem \
  get ${FLANNEL_ETCD_PREFIX}/subnets/172.30.77.0-24
{"PublicIP":"192.168.1.137","BackendType":"vxlan","BackendData":
{"VtepMAC":"62:fc:03:83:1b:2b"}}
```

确保各节点间Pod 网段能互联互通

在各个节点部署完Flanneld 后，查看已分配的Pod 子网段列表：

```
$ etcdctl \
  --endpoints=${ETCD_ENDPOINTS} \
  --ca-file=/etc/kubernetes/ssl/ca.pem \
  --cert-file=/etc/flanneld/ssl/flanneld.pem \
  --key-file=/etc/flanneld/ssl/flanneld-key.pem \
  ls ${FLANNEL_ETCD_PREFIX}/subnets

/kubernetes/network/subnets/172.30.19.0-24
/kubernetes/network/subnets/172.30.30.0-24
/kubernetes/network/subnets/172.30.77.0-24
/kubernetes/network/subnets/172.30.41.0-24
/kubernetes/network/subnets/172.30.83.0-24
```

当前五个节点分配的 Pod 网段分别是：172.30.77.0-24、172.30.30.0-24、172.30.19.0-24、172.30.41.0-24、172.30.83.0-24。

6. 部署master 节点

kubernetes master 节点包含的组件有：

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

目前这3个组件需要部署到同一台机器上：（后面再部署高可用的master）

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

master 节点与node 节点上的Pods 通过Pod 网络通信，所以需要在master 节点上部署Flannel 网络。

环境变量

```
$ export NODE_IP=192.168.1.137  # 当前部署的master 机器IP
$ source /usr/k8s/bin/env.sh
```

下载最新版本的二进制文件

在[kubernetes changelog](kubernetes changelog) 页面下载最新版本的文件：

```
$ wget https://dl.k8s.io/v1.8.2/kubernetes-server-linux-amd64.tar.gz
$ tar -xzvf kubernetes.tar.gz
```

将二进制文件拷贝到`/usr/k8s/bin`目录

```
$ sudo cp -r server/bin/{kube-apiserver,kube-controller-manager,kube-scheduler}
/usr/k8s/bin/
```

创建kubernetes 证书

创建kubernetes 证书签名请求：

```
$ cat > kubernetes-csr.json <<EOF
{
  "CN": "kubernetes",
  "hosts": [
    "127.0.0.1",
    "${NODE_IP}",
    "${MASTER_URL}",
    "${CLUSTER_KUBERNETES_SVC_IP}",
    "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"
    }
  ]
}
EOF
```

- 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**，所以上面分别指定了当前部署的 master 节点主机 IP 以及apiserver 负载的内部域名
- 还需要添加 kube-apiserver 注册的名为 `kubernetes` 的服务 IP (Service Cluster IP)，一般是 kube-apiserver `--service-cluster-ip-range` 选项值指定的网段的**第一个IP**，如 “10.254.0.1”

生成kubernetes 证书和私钥：

```
$ cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem \
  -ca-key=/etc/kubernetes/ssl/ca-key.pem \
  -config=/etc/kubernetes/ssl/ca-config.json \
  -profile=kubernetes kubernetes-csr.json | cfssljson -bare kubernetes
$ ls kubernetes*
kubernetes.csr  kubernetes-csr.json  kubernetes-key.pem  kubernetes.pem
$ sudo mkdir -p /etc/kubernetes/ssl/
$ sudo mv kubernetes*.pem /etc/kubernetes/ssl/
```

6.1 配置和启动kube-apiserver

创建kube-apiserver 使用的客户端token 文件

kubelet 首次启动时向kube-apiserver 发送TLS Bootstrapping 请求，kube-apiserver 验证请求中的token 是否与它配置的token.csv 一致，如果一致则自动为kubelet 生成证书和密钥。

```
$ # 导入的 environment.sh 文件定义了 BOOTSTRAP_TOKEN 变量
$ cat > token.csv <<EOF
${BOOTSTRAP_TOKEN},kubelet-bootstrap,10001,"system:kubelet-bootstrap"
EOF
$ sudo mv token.csv /etc/kubernetes/
```

创建kube-apiserver 的systemd unit文件

```
$ cat  > kube-apiserver.service <<EOF
[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=network.target

[Service]
ExecStart=/usr/k8s/bin/kube-apiserver \\
  --admission-
control=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,ResourceQuota
\\
  --advertise-address=${NODE_IP} \\
  --bind-address=0.0.0.0 \\
  --insecure-bind-address=${NODE_IP} \\
```

```
  --authorization-mode=Node,RBAC \\
  --runtime-config=rbac.authorization.k8s.io/v1alpha1 \\
  --kubelet-https=true \\
  --experimental-bootstrap-token-auth \\
  --token-auth-file=/etc/kubernetes/token.csv \\
  --service-cluster-ip-range=${SERVICE_CIDR} \\
  --service-node-port-range=${NODE_PORT_RANGE} \\
  --tls-cert-file=/etc/kubernetes/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/kubernetes/ssl/kubernetes.pem \\
  --etcd-keyfile=/etc/kubernetes/ssl/kubernetes-key.pem \\
  --etcd-servers=${ETCD_ENDPOINTS} \\
  --enable-swagger-ui=true \\
  --allow-privileged=true \\
  --apiserver-count=2 \\
  --audit-log-maxage=30 \\
  --audit-log-maxbackup=3 \\
  --audit-log-maxsize=100 \\
  --audit-log-path=/var/lib/audit.log \\
  --event-ttl=1h \\
  --logtostderr=true \\
  --v=6
Restart=on-failure
RestartSec=5
Type=notify
LimitNOFILE=65536

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

- kube-apiserver 1.6 版本开始使用 etcd v3 API 和存储格式
- `--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 通过使用证书里指定相关的 User、Group 来达到通过 RBAC 授权的目的
- 如果使用了 kubelet TLS Boostrap 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`，否则部署集群插件时会失败
- `--bind-address` 不能为 `127.0.0.1`
- `--service-cluster-ip-range` 指定 Service Cluster IP 地址段，该地址段不能路由可达
- `--service-node-port-range=${NODE_PORT_RANGE}` 指定 NodePort 的端口范围
- 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
- kube-apiserver 1.8版本后需要在`--authorization-mode`参数中添加`Node`，即：`--authorization-mode=Node,RBAC`，否则Node 节点无法注册

启动kube-apiserver

```
$ sudo cp kube-apiserver.service /etc/systemd/system/
$ sudo systemctl daemon-reload
```

```
$ sudo systemctl enable kube-apiserver
$ sudo systemctl start kube-apiserver
$ sudo systemctl status kube-apiserver
```

6.2 配置和启动kube-controller-manager

创建kube-controller-manager 的systemd unit 文件

```
$ cat > kube-controller-manager.service <<EOF
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes

[Service]
ExecStart=/usr/k8s/bin/kube-controller-manager \\
  --address=127.0.0.1 \\
  --master=http://${MASTER_URL} \\
  --allocate-node-cidrs=true \\
  --service-cluster-ip-range=${SERVICE_CIDR} \\
  --cluster-cidr=${CLUSTER_CIDR} \\
  --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 \\
  --v=2
Restart=on-failure
RestartSec=5

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

- `--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
- `--master=http://{MASTER_URL}`：使用`http`(非安全端口)与 kube-apiserver 通信
- `--cluster-cidr` 指定 Cluster 中 Pod 的 CIDR 范围，该网段在各 Node 间必须路由可达(flanneld保证)
- `--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**
- `--leader-elect=true` 部署多台机器组成的 master 集群时选举产生一处于工作状态的 `kube-controller-manager` 进程

启动kube-controller-manager

```
$ sudo cp kube-controller-manager.service /etc/systemd/system/
$ sudo systemctl daemon-reload
$ sudo systemctl enable kube-controller-manager
$ sudo systemctl start kube-controller-manager
$ sudo systemctl status kube-controller-manager
```

6.3 配置和启动kube-scheduler

创建kube-scheduler 的systemd unit文件

```
$ cat > kube-scheduler.service <<EOF
[Unit]
```

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

[Service]
ExecStart=/usr/k8s/bin/kube-scheduler \\
  --address=127.0.0.1 \\
  --master=http://${MASTER_URL} \\
  --leader-elect=true \\
  --v=2
Restart=on-failure
RestartSec=5

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

- `--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
- `--master=http://${MASTER_URL}`：使用`http`(非安全端口)与 kube-apiserver 通信
- `--leader-elect=true` 部署多台机器组成的 master 集群时选举产生一处于工作状态的 `kube-controller-manager` 进程

启动kube-scheduler

```
$ sudo cp kube-scheduler.service /etc/systemd/system/
$ sudo systemctl daemon-reload
$ sudo systemctl enable kube-scheduler
$ sudo systemctl start kube-scheduler
$ sudo systemctl status kube-scheduler
```

6.4 验证master 节点

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

7. kube-apiserver 高可用

按照上面的方式在`master01`与`master02`机器上安装`kube-apiserver`、`kube-controller-manager`、`kube-scheduler`，但是现在还不能正常允许，因为我们的域名`k8s-api.virtual.local`对应的`master01`节点直接通过http 和https 还不能访问，这里我们使用`haproxy` 来代替请求

安装haproxy

```
$ yum install -y haproxy
```

配置haproxy

由于集群内部有的组建是通过非安全端口访问apiserver 的，有的是通过安全端口访问apiserver 的，所以我们要配置http 和https 两种代理方式，配置文件 `/etc/haproxy/haproxy.cfg`：

```
listen stats
  bind    *:9000
  mode    http
  stats   enable
  stats   hide-version
  stats   uri       /stats
  stats   refresh   30s
  stats   realm     Haproxy\ Statistics
  stats   auth      Admin:Password

frontend k8s-api
```

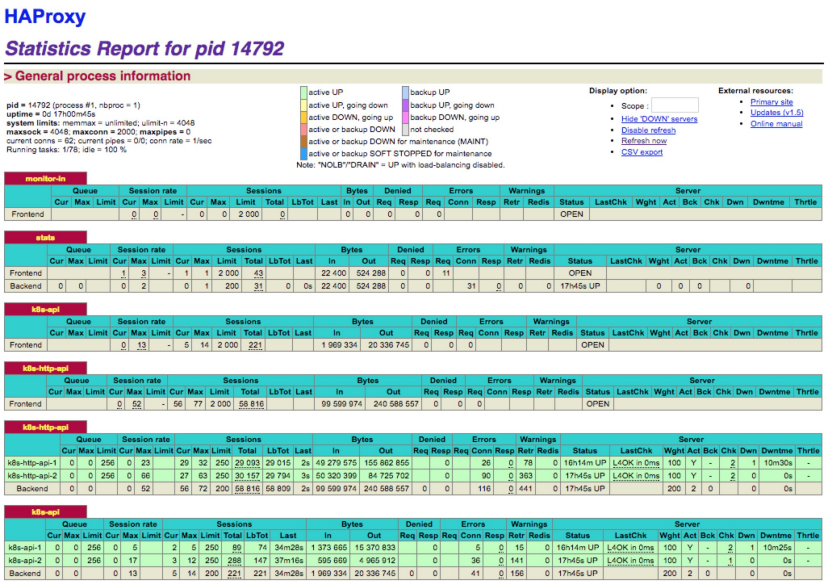
```
    bind 192.168.1.137:443
    mode tcp
    option tcplog
    tcp-request inspect-delay 5s
    tcp-request content accept if { req.ssl_hello_type 1 }
    default_backend k8s-api

backend k8s-api
    mode tcp
    option tcplog
    option tcp-check
    balance roundrobin
    default-server inter 10s downinter 5s rise 2 fall 2 slowstart 60s maxconn 250
maxqueue 256 weight 100
    server k8s-api-1 192.168.1.137:6443 check
    server k8s-api-2 192.168.1.138:6443 check

frontend k8s-http-api
    bind 192.168.1.137:80
    mode tcp
    option tcplog
    default_backend k8s-http-api

backend k8s-http-api
    mode tcp
    option tcplog
    option tcp-check
    balance roundrobin
    default-server inter 10s downinter 5s rise 2 fall 2 slowstart 60s maxconn 250
maxqueue 256 weight 100
    server k8s-http-api-1 192.168.1.137:8080 check
    server k8s-http-api-2 192.168.1.138:8080 check
```

通过上面的配置文件我们可以看出通过`https`的访问将请求转发给apiserver 的6443端口了，http的请求转发到了apiserver 的8080端口。

启动haproxy

```
$ sudo systemctl start haproxy
$ sudo systemctl enable haproxy
$ sudo systemcl status haproxy
```

然后我们可以通过上面`9000`端口监控我们的`haproxy`的运行状态(`192.168.1.137:9000/stats`):



haproxy stats

问题

上面我们的haproxy的确可以代理我们的两个master 上的apiserver 了，但是还不是高可用的，如果master01这个节点down 掉了，那么我们haproxy 就不能正常提供服务了。这里我们可以使用两种方法来实现高可用

方式1：使用阿里云SLB

这种方式实际上是最省心的，在阿里云上建一个内网的SLB，将master01 与master02 添加到SLB 机器组中，转发80(http)和443(https)端口即可

方式2：使用keepalived

KeepAlived 是一个高可用方案，通过 VIP（即虚拟 IP）和心跳检测来实现高可用。其原理是存在一组（两台）服务器，分别赋予 Master、Backup 两个角色，默认情况下Master 会绑定VIP 到自己的网卡上，对外提供服务。Master、Backup 会在一定的时间间隔向对方发送心跳数据包来检测对方的状态，这个时间间隔一般为 2 秒钟，如果Backup 发现Master 宕机，那么Backup 会发送ARP 包到网关，把VIP 绑定到自己的网卡，此时Backup 对外提供服务，实现自动化的故障转移，当Master 恢复的时候会重新接管服务。非常类似于路由器中的虚拟路由器冗余协议（VRRP）

开启路由转发，这里我们定义虚拟IP为：**192.168.1.139**

```
$ vi /etc/sysctl.conf
# 添加以下内容
net.ipv4.ip_forward = 1
net.ipv4.ip_nonlocal_bind = 1

# 验证并生效
$ systcl -p
# 验证是否生效
$ cat /proc/sys/net/ipv4/ip_forward
1
```

安装keepalived:

```
$ yum install -y keepalived
```

我们这里将master01 设置为Master，master02 设置为Backup，修改配置：

```
$ vi /etc/keepalived/keepalived.conf
! Configuration File for keepalived

global_defs {
   notification_email {
   }
   router_id kube_api
}

vrrp_script check_k8s {
    # 自身状态检测
    script "/etc/keepalived/chk_k8s_master.sh"
    interval 3
    weight 5
}

vrrp_instance haproxy-vip {
    # 使用单播通信，默认是组播通信
    unicast_src_ip 192.168.1.137
    unicast_peer {
        192.168.1.138
    }
    # 初始化状态
    state MASTER
    # 虚拟ip 绑定的网卡
    interface eth0
```

```
    # 此ID 要与Backup 配置一致
    virtual_router_id 51
    # 默认启动优先级，要比Backup 大点，但要控制量，保证自身状态检测生效
    priority 100
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        # 虚拟ip 地址
        192.168.1.139
    }
    track_script {
        check_k8s
    }
}

# 自身状态监测脚本，这里通过 api-server 是否在监听端口（6433 端口）来判断状态。
$ vi /etc/keepalived/chk_k8s_master.sh
#!/bin/bash

count=`ss -tnl | grep 6433 | wc -l`

if [ $count = 0 ]; then
    exit 1
else
    exit 0
fi
```

统一的方式在master02 节点上安装keepalived，修改配置，只需要将state 更改成BACKUP，priority更改成99，unicast_src_ip 与unicast_peer 地址修改即可。

启动keepalived:

```
$ systemctl start keepalived
$ systemctl enable keepalived
# 查看日志
$ journalctl -f -u keepalived
```

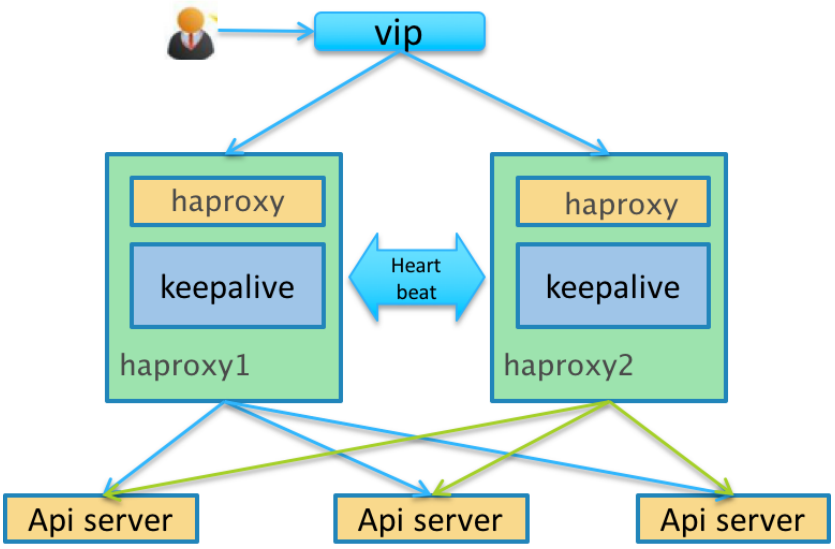
验证虚拟IP:

```
# 使用ifconfig -a 命令查看不到，要使用ip addr
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:16:3e:00:55:c1 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.137/24 brd 192.168.1.255 scope global dynamic eth0
       valid_lft 31447746sec preferred_lft 31447746sec
    inet 192.168.1.139/24 brd 192.168.1.255 scope global secondary eth0-vip
       valid_lft forever preferred_lft forever
```

验证apiserver：关闭master01 节点上的kube-apiserver 进程，然后查看虚拟ip是否漂移到了master02 节点。

然后我们就可以将第一步在`/etc/hosts`里面设置的域名对应的IP 更改为我们的虚拟IP了

master01 与master 02 节点都需要安装keepalived 和haproxy，实际上我们虚拟IP的自身检测应该是检测haproxy，脚本大家可以自行更改

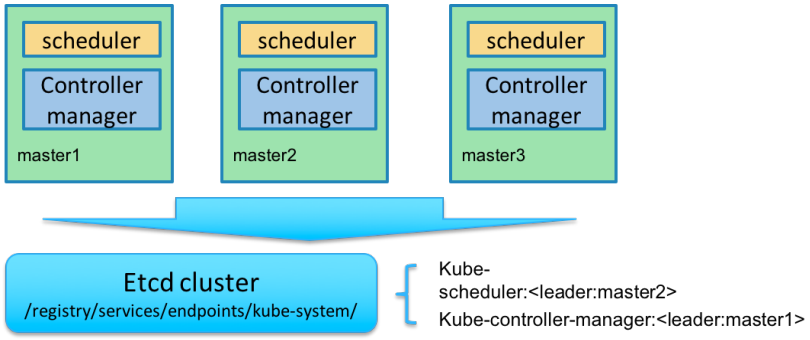


kube-apiserver ha

这样我们就实现了接入层apiserver 的高可用了，一个部分是多活的apiserver 服务，另一个部分是一主一备的haproxy 服务。

kube-controller-manager 和kube-scheduler 的高可用

Kubernetes 的管理层服务包括`kube-scheduler`和`kube-controller-manager`。kube-scheduler和kube-controller-manager使用一主多从的高可用方案，在**同一时刻只允许一个服务**处以具体的任务。Kubernetes中实现了一套简单的选主逻辑，依赖Etcd实现scheduler和controller-manager的选主功能。如果scheduler和controller-manager在启动的时候设置了`leader-elect`参数，它们在启动后会先尝试获取leader节点身份，只有在获取leader节点身份后才可以执行具体的业务逻辑。它们分别会在Etcd中创建kube-scheduler和kube-controller-manager的endpoint，endpoint的信息中记录了当前的leader节点信息，以及记录的上次更新时间。leader节点会定期更新endpoint的信息，维护自己的leader身份。每个从节点的服务都会定期检查endpoint的信息，如果endpoint的信息在时间范围内没有更新，它们会尝试更新自己为leader节点。scheduler服务以及controller-manager服务之间不会进行通信，利用Etcd的强一致性，能够保证在分布式高并发情况下leader节点的全局唯一性。整体方案如下图所示：



img

当集群中的leader节点服务异常后，其它节点的服务会尝试更新自身为leader节点，当有多个节点同时更新endpoint时，由Etcd保证只有一个服务的更新请求能够成功。通过这种机制sheduler和controller-manager可以保证在leader节点宕机后其它的节点可以顺利选主，保证服务故障后快速恢复。当集群中的网络出现故障时对服务的选主影响不是很大，因为scheduler和controller-manager是依赖Etcd进行选主的，在网络故障后，可以和Etcd通信的主机依然可以按照之前的逻辑进行选主，就算集群被切分，Etcd也可以保证同一时刻只有一个节点的服务处于leader状态。

8. 部署Node 节点

kubernetes Node 节点包含如下组件：

- flanneld
- docker
- kubelet

- kube-proxy

环境变量

```
$ source /usr/k8s/bin/env.sh
$ export KUBE_APISERVER="https://${MASTER_URL}"
$ export NODE_IP=192.168.1.170  # 当前部署的节点 IP
```

按照上面的步骤安装配置好flanneld

配置docker

你可以用二进制或yum install 的方式来安装docker，然后修改docker 的systemd unit 文件：

```
$ cat /usr/lib/systemd/system/docker.service  # 用systemctl status docker 命令可查看unit 文件路径
[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
After=network-online.target firewalld.service
Wants=network-online.target

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
EnvironmentFile=-/run/flannel/docker
ExecStart=/usr/bin/dockerd --log-level=info $DOCKER_NETWORK_OPTIONS
ExecReload=/bin/kill -s HUP $MAINPID
# Having non-zero Limit*s causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
LimitNOFILE=infinity
LimitNPROC=infinity
LimitCORE=infinity
# Uncomment TasksMax if your systemd version supports it.
# Only systemd 226 and above support this version.
#TasksMax=infinity
TimeoutStartSec=0
# set delegate yes so that systemd does not reset the cgroups of docker containers
Delegate=yes
# kill only the docker process, not all processes in the cgroup
KillMode=process
# restart the docker process if it exits prematurely
Restart=on-failure
StartLimitBurst=3
StartLimitInterval=60s

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

- dockerd 运行时会调用其它 docker 命令，如 docker-proxy，所以需要将 docker 命令所在的目录加到 PATH 环境变量中
- flanneld 启动时将网络配置写入到 `/run/flannel/docker` 文件中的变量 `DOCKER_NETWORK_OPTIONS`，dockerd 命令行上指定该变量值来设置 docker0 网桥参数
- 如果指定了多个 `EnvironmentFile` 选项，则必须将 `/run/flannel/docker` 放在最后(确保 docker0 使用 flanneld 生成的 bip 参数)
- 不能关闭默认开启的 `--iptables` 和 `--ip-masq` 选项
- 如果内核版本比较新，建议使用 `overlay` 存储驱动

- docker 从 1.13 版本开始，可能将 **iptables FORWARD chain的默认策略设置为DROP**，从而导致 ping 其它 Node 上的 Pod IP 失败，遇到这种情况时，需要手动设置策略为 `ACCEPT`：

```
$ sudo iptables -P FORWARD ACCEPT
```

并且把以下命令写入/etc/rc.local文件中，防止节点重启**iptables FORWARD chain的默认策略又还原为DROP**

```
sleep 60 && /sbin/iptables -P FORWARD ACCEPT
```

- 为了加快 pull image 的速度，可以使用国内的仓库镜像服务器，同时增加下载的并发数。(如果 dockerd 已经运行，则需要重启 dockerd 生效。)

```
$ cat /etc/docker/daemon.json
{
  "max-concurrent-downloads": 10
}
```

启动docker

```
$ sudo systemctl daemon-reload
$ sudo systemctl stop firewalld
$ sudo systemctl disable firewalld
$ sudo iptables -F && sudo iptables -X && sudo iptables -F -t nat && sudo iptables -X -t nat
$ sudo systemctl enable docker
$ sudo systemctl start docker
```

- 需要关闭 firewalld(centos7)/ufw(ubuntu16.04)，否则可能会重复创建的 iptables 规则
- 最好清理旧的 iptables rules 和 chains 规则
- 执行命令：docker version，检查docker服务是否正常

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 角色，然后kubelet 才有权限创建认证请求(certificatesigningrequests)：

```
$ kubectl create clusterrolebinding kubelet-bootstrap --clusterrole=system:node-bootstrapper --user=kubelet-bootstrap
```

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

另外1.8 版本中还需要为Node 请求创建一个RBAC 授权规则：

```
$ kubectl create clusterrolebinding kubelet-nodes --clusterrole=system:node --group=system:nodes
```

然后下载最新的kubelet 和kube-proxy 二进制文件（前面下载kubernetes 目录下面其实也有）：

```
$ wget https://dl.k8s.io/v1.8.2/kubernetes-server-linux-amd64.tar.gz
$ tar -xzvf kubernetes-server-linux-amd64.tar.gz
$ cd kubernetes
$ tar -xzvf  kubernetes-src.tar.gz
$ sudo cp -r ./server/bin/{kube-proxy,kubelet} /usr/k8s/bin/
```

创建kubelet bootstapping kubeconfig 文件

```
$ # 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
$ mv bootstrap.kubeconfig /etc/kubernetes/
```

- `--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 kubelet 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kubelet 的systemd unit 文件

```
$ sudo mkdir /var/lib/kubelet # 必须先创建工作目录
$ cat > kubelet.service <<EOF
[Unit]
Description=Kubernetes Kubelet
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=docker.service
Requires=docker.service

[Service]
WorkingDirectory=/var/lib/kubelet
ExecStart=/usr/k8s/bin/kubelet \\
  --address=${NODE_IP} \\
  --hostname-override=${NODE_IP} \\
  --experimental-bootstrap-kubeconfig=/etc/kubernetes/bootstrap.kubeconfig \\
  --kubeconfig=/etc/kubernetes/kubelet.kubeconfig \\
  --require-kubeconfig \\
  --cert-dir=/etc/kubernetes/ssl \\
  --cluster-dns=${CLUSTER_DNS_SVC_IP} \\
  --cluster-domain=${CLUSTER_DNS_DOMAIN} \\
  --hairpin-mode promiscuous-bridge \\
  --allow-privileged=true \\
  --serialize-image-pulls=false \\
  --logtostderr=true \\
  --v=2
ExecStartPost=/sbin/iptables -A INPUT -s 10.0.0.0/8 -p tcp --dport 4194 -j ACCEPT
ExecStartPost=/sbin/iptables -A INPUT -s 172.17.0.0/12 -p tcp --dport 4194 -j ACCEPT
ExecStartPost=/sbin/iptables -A INPUT -s 192.168.1.0/16 -p tcp --dport 4194 -j ACCEPT
ExecStartPost=/sbin/iptables -A INPUT -p tcp --dport 4194 -j DROP
Restart=on-failure
RestartSec=5

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

- `-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(自动创建 `--kubeconfig` 指定的文件)
- 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 kue-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

启动kubelet

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

通过kubelet 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
node-csr--k3G2G1EoM4h9w1FuJRjJjfbIPNxa551A8TZfW9dG-g   2m        kubelet-bootstrap
Pending
$ kubectl get nodes
No resources found.
```

通过CSR 请求：

```
$ kubectl certificate approve node-csr--k3G2G1EoM4h9w1FuJRjJjfbIPNxa551A8TZfW9dG-g
certificatesigningrequest "node-csr--k3G2G1EoM4h9w1FuJRjJjfbIPNxa551A8TZfW9dG-g"
approved
$ kubectl get nodes
NAME             STATUS    ROLES     AGE       VERSION
192.168.1.170    Ready     <none>    48s       v1.8.1
```

自动生成了kubelet kubeconfig 文件和公私钥：

```
$ ls -l /etc/kubernetes/kubelet.kubeconfig
-rw------- 1 root root 2280 Nov  7 10:26 /etc/kubernetes/kubelet.kubeconfig
$ ls -l /etc/kubernetes/ssl/kubelet*
-rw-r--r-- 1 root root 1046 Nov  7 10:26 /etc/kubernetes/ssl/kubelet-client.crt
-rw------- 1 root root  227 Nov  7 10:22 /etc/kubernetes/ssl/kubelet-client.key
-rw-r--r-- 1 root root 1115 Nov  7 10:16 /etc/kubernetes/ssl/kubelet.crt
-rw------- 1 root root 1675 Nov  7 10:16 /etc/kubernetes/ssl/kubelet.key
```

配置kube-proxy

创建kube-proxy 证书签名请求：

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

```
}
EOF
```

- CN 指定该证书的 User 为 `system:kube-proxy`
- `kube-apiserver` 预定义的 RoleBinding `system:node-proxier` 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
- hosts 属性值为空列表

生成kube-proxy 客户端证书和私钥

```
$ cfssl gencert -ca=/etc/kubernetes/ssl/ca.pem \
  -ca-key=/etc/kubernetes/ssl/ca-key.pem \
  -config=/etc/kubernetes/ssl/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
$ sudo mv kube-proxy*.pem /etc/kubernetes/ssl/
```

创建kube-proxy kubeconfig 文件

```
$ # 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
$ mv kube-proxy.kubeconfig /etc/kubernetes/
```

- 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

创建kube-proxy 的systemd unit 文件

```
$ sudo mkdir -p /var/lib/kube-proxy # 必须先创建工作目录
$ cat > kube-proxy.service <<EOF
[Unit]
Description=Kubernetes Kube-Proxy Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=network.target

[Service]
WorkingDirectory=/var/lib/kube-proxy
ExecStart=/usr/k8s/bin/kube-proxy \\
  --bind-address=${NODE_IP} \\
  --hostname-override=${NODE_IP} \\
  --cluster-cidr=${SERVICE_CIDR} \\
```

```
  --kubeconfig=/etc/kubernetes/kube-proxy.kubeconfig \\
  --logtostderr=true \\
  --v=2
Restart=on-failure
RestartSec=5
LimitNOFILE=65536

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

- `--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
- `--cluster-cidr` 必须与 kube-apiserver 的 `--service-cluster-ip-range` 选项值一致
- 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

启动kube-proxy

```
$ sudo cp kube-proxy.service /etc/systemd/system/
$ sudo systemctl daemon-reload
$ sudo systemctl enable kube-proxy
$ sudo systemctl start kube-proxy
$ systemctl status kube-proxy
```

验证集群功能

定义yaml 文件：（将下面内容保存为：nginx-ds.yaml）

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ds
  labels:
    app: nginx-ds
spec:
  type: NodePort
  selector:
    app: nginx-ds
  ports:
  - name: http
    port: 80
    targetPort: 80
---
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: nginx-ds
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
spec:
  template:
    metadata:
      labels:
        app: nginx-ds
    spec:
      containers:
```

```
      - name: my-nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

创建 Pod 和服务：

```
$ kubectl create -f nginx-ds.yml
service "nginx-ds" created
daemonset "nginx-ds" created
```

执行下面的命令查看Pod 和SVC：

```
$ kubectl get pods -o wide
NAME             READY     STATUS    RESTARTS   AGE       IP            NODE
nginx-ds-f29zt   1/1       Running   0          23m       172.17.0.2    192.168.1.170
$ kubectl get svc
NAME         TYPE         CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
nginx-ds     NodePort     10.254.6.249    <none>        80:30813/TCP     24m
```

可以看到：

- 服务IP：10.254.6.249
- 服务端口：80
- NodePort端口：30813

在所有 Node 上执行：

```
$ curl 10.254.6.249
$ curl 192.168.1.170:30813
```

执行上面的命令预期都会输出nginx 欢迎页面内容，表示我们的Node 节点正常运行了。

9. 部署kubedns 插件

官方文件目录：[kubernetes/cluster/addons/dns]

使用的文件：

```
$ ls *.yaml *.base
kubedns-cm.yaml  kubedns-sa.yaml  kubedns-controller.yaml.base  kubedns-svc.yaml.base
```

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 相关的API 权限：

```
$ kubectl get clusterrolebindings system:kube-dns -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: 2017-11-06T10:51:59Z
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-dns
  resourceVersion: "78"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/system%3Akube-dns
  uid: 83a25fd9-c2e0-11e7-9646-00163e0055c1
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

无需更改

配置kube-dns 服务

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

- 需要将 spec.clusterIP 设置为集群环境变量中变量 `CLUSTER_DNS_SVC_IP` 值，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
88c88
<         - --domain=__PILLAR__DNS__DOMAIN__.
---
>         - --domain=cluster.local
128c128
<         - --server=/__PILLAR__DNS__DOMAIN__/127.0.0.1#10053
---
>         - --server=/cluster.local/127.0.0.1#10053
160,161c160,161
<         - --
probe=kubedns,127.0.0.1:10053,kubernetes.default.svc.__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.svc.cluster.local,5,A
>         - --probe=dnsmasq,127.0.0.1:53,kubernetes.default.svc.cluster.local,5,A
```

- `--domain` 为集群环境变量`CLUSTER_DNS_DOMAIN` 的值
- 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
/home/ych/k8s-repo/kube-dns
$ ls *.yaml
kubedns-cm.yaml  kubedns-controller.yaml  kubedns-sa.yaml  kubedns-svc.yaml
$ kubectl create -f .
```

检查kubedns 功能

新建一个Deployment

```
$ cat > my-nginx.yaml<<EOF
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx:1.7.9
        ports:
```

```
        - containerPort: 80
EOF
$ kubectl create -f my-nginx.yaml
deployment "my-nginx" created
```

Expose 该Deployment，生成my-nginx 服务

```
$ kubectl expose deploy my-nginx
$ kubectl get services
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.254.0.1      <none>        443/TCP    1d
my-nginx     ClusterIP   10.254.32.162   <none>        80/TCP     56s
```

然后创建另外一个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` 解析到上面显示的CLUSTER-IP `10.254.32.162`上

```
$ cat > pod-nginx.yaml<<EOF
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
EOF
$ kubectl create -f pod-nginx.yaml
pod "nginx" created
$ 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.
options ndots:5
root@nginx:/# ping my-nginx
PING my-nginx.default.svc.cluster.local (10.254.32.162): 48 data bytes
^C--- my-nginx.default.svc.cluster.local ping statistics ---
14 packets transmitted, 0 packets received, 100% packet loss

root@nginx:/# ping kubernetes
PING kubernetes.default.svc.cluster.local (10.254.0.1): 48 data bytes
^C--- kubernetes.default.svc.cluster.local ping statistics ---
6 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): 48 data bytes
^C--- kube-dns.kube-system.svc.cluster.local ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

10. 部署Dashboard 插件

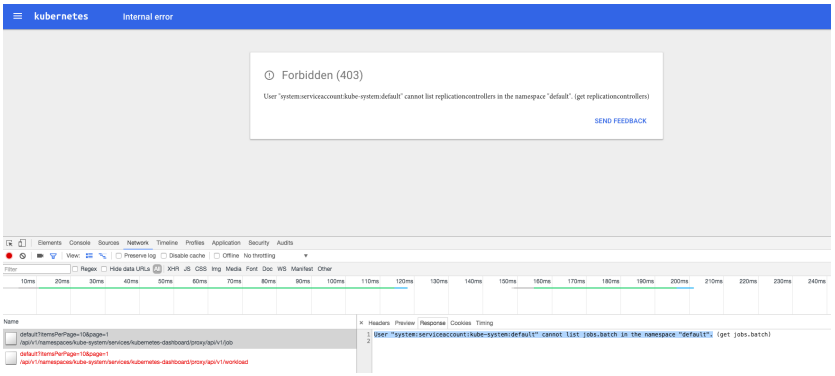
官方文件目录：[kubernetes/cluster/addons/dashboard]

使用的文件如下：

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

- 新加了 `dashboard-rbac.yaml` 文件，定义 dashboard 使用的 RoleBinding。

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，前端界面提示：



403

解决办法是：定义一个名为dashboard 的ServiceAccount，然后将它和Cluster Role view 绑定：

```
$ cat > dashboard-rbac.yaml<<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: dashboard
  namespace: kube-system
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1alpha1
metadata:
  name: dashboard
subjects:
  - kind: ServiceAccount
    name: dashboard
    namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
EOF
```

配置dashboard-controller

```
20a21
>       serviceAccountName: dashboard
```

- 使用名为 dashboard 的自定义 ServiceAccount

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

执行所有定义文件

```
$ pwd
/home/ych/k8s-repo/dashboard
$ ls *.yaml
dashboard-controller.yaml  dashboard-rbac.yaml  dashboard-service.yaml
$ kubectl create -f  .
```

检查执行结果

查看分配的 NodePort

```
$ kubectl get services kubernetes-dashboard -n kube-system
NAME                   TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes-dashboard   NodePort   10.254.104.90   <none>        80:31202/TCP     1m
```

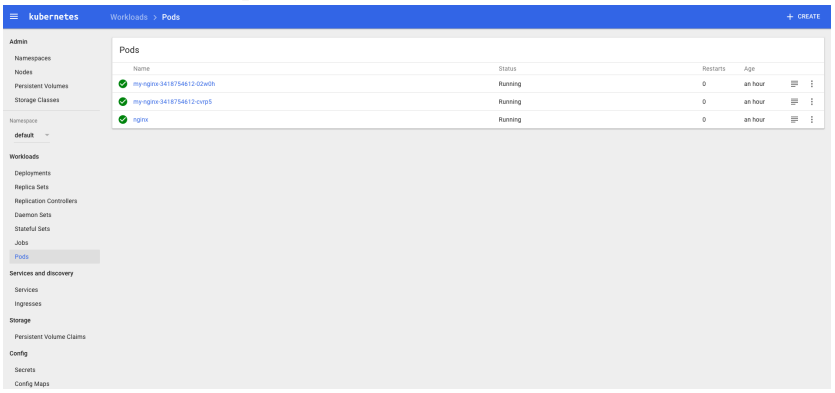
- NodePort 31202映射到dashboard pod 80端口；

检查 controller

```
$ kubectl get deployment kubernetes-dashboard  -n kube-system
NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kubernetes-dashboard   1         1         1            1           3m

$ kubectl get pods  -n kube-system | grep dashboard
kubernetes-dashboard-6667f9b4c-4xbpz   1/1       Running   0          3m
```

访问dashboard

1. kubernetes-dashboard 服务暴露了 NodePort，可以使用 `http://NodeIP:nodePort` 地址访问 dashboard
2. 通过 kube-apiserver 访问 dashboard
3. 通过 kubectl proxy 访问 dashboard



dashboard ui

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

11. 部署Heapster 插件

到[heapster release](#) 页面下载最新版的heapster

```
$ wget https://github.com/kubernetes/heapster/archive/v1.4.3.tar.gz
$ tar -xzvf v1.4.3.tar.gz
```

部署相关文件目录：`/home/ych/k8s-repo/heapster-1.4.3/deploy/kube-config`

```
$ ls influxdb/ && ls rbac/
grafana.yaml  heapster.yaml  influxdb.yaml
heapster-rbac.yaml
```

为方便测试访问，将`grafana.yaml`下面的服务类型设置为`type=NodePort`

执行所有文件

```
$ kubectl create -f rbac/heapster-rbac.yaml
clusterrolebinding "heapster" created
$ kubectl create -f influxdb
deployment "monitoring-grafana" created
service "monitoring-grafana" created
serviceaccount "heapster" created
deployment "heapster" created
service "heapster" 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            0           2m
monitoring-influxdb    1         1         1            1           2m
```

检查 Pods

```
$ kubectl get pods -n kube-system | grep -E 'heapster|monitoring'
heapster-7cf895f48f-p98tk              1/1       Running             0          2m
monitoring-grafana-c9d5cd98d-gb9xn     0/1       CrashLoopBackOff    4          2m
monitoring-influxdb-67f8d587dd-zqj6p   1/1       Running             0          2m
```

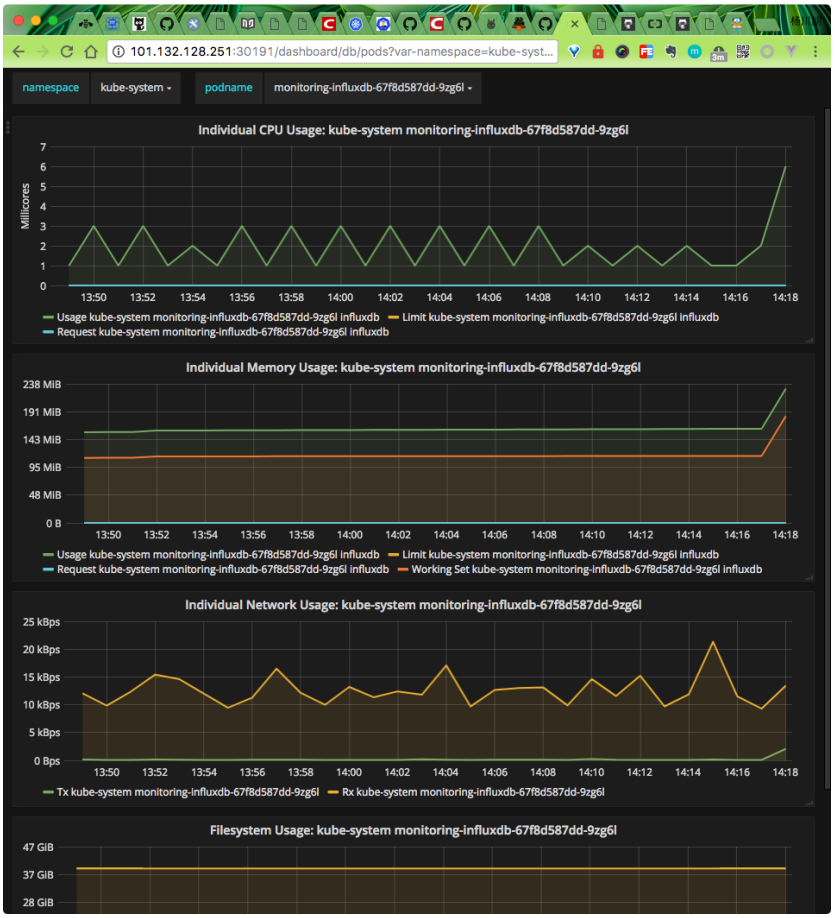
我们可以看到`monitoring-grafana`的POD 是没有执行成功的，通过查看日志可以看到下面的错误信息：
要解决这个问题(heapster issues)我们需要将grafana 的镜像版本更改成：
`gcr.io/google_containers/heapster-grafana-amd64:v4.0.2`，然后重新执行，即可正常。

访问 grafana

上面我们修改grafana 的Service 为NodePort 类型：

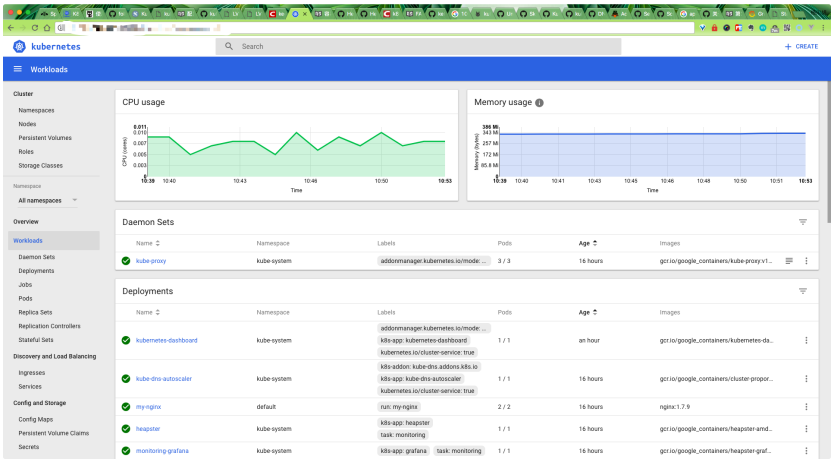
```
$ kubectl get svc -n kube-system
NAME                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
monitoring-grafana    NodePort    10.254.34.89    <none>        80:30191/TCP     28m
```

则我们就可以通过任意一个节点加上上面的30191端口就可以访问grafana 了。



grafana ui

heapster 正确安装后，我们便可以回去看我们的dashboard 是否有图表出现了：



dashboard

12. 安装Ingress

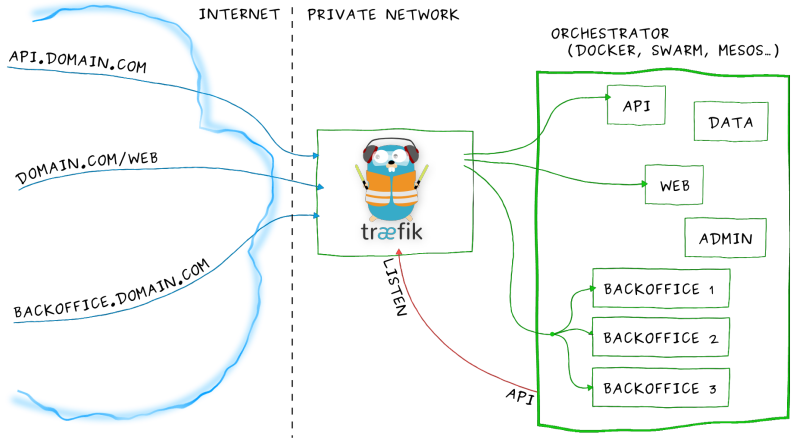
`Ingress`其实就是从`kuberenets`集群外部访问集群的一个入口，将外部的请求转发到集群内不同的Service 上，其实就相当于nginx、apache 等负载均衡代理服务器，再加上一个规则定义，路由信息的刷新需要靠

`Ingress controller`来提供

`Ingress controller`可以理解为一个监听器，通过不断地与`kube-apiserver`打交道，实时的感知后端service、pod 等的变化，当得到这些变化信息后，`Ingress controller`再结合`Ingress`的配置，更新反向代理负载均衡器，达到服务发现的作用。其实这点和服务发现工具`consul`的`consul-template`非常类似。

部署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。



traefik

创建rbac

创建文件：`ingress-rbac.yaml`，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
```

DaemonSet 形式部署traefik

创建文件：`traefik-daemonset.yaml`，为保证traefik 总能提供服务，在每个节点上都部署一个traefik，所以这里使用`DaemonSet` 的形式

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: traefik-conf
  namespace: kube-system
data:
  traefik-config: |-
    defaultEntryPoints = ["http","https"]
    [entryPoints]
      [entryPoints.http]
```

```
      address = ":80"
        [entryPoints.http.redirect]
          entryPoint = "https"
      [entryPoints.https]
      address = ":443"
        [entryPoints.https.tls]
          [[entryPoints.https.tls.certificates]]
          CertFile = "/ssl/ssl.crt"
          KeyFile = "/ssl/ssl.key"

---
kind: DaemonSet
apiVersion: extensions/v1beta1
metadata:
  name: traefik-ingress
  namespace: kube-system
  labels:
    k8s-app: traefik-ingress
spec:
  template:
    metadata:
      labels:
        k8s-app: traefik-ingress
        name: traefik-ingress
    spec:
      terminationGracePeriodSeconds: 60
      restartPolicy: Always
      serviceAccountName: ingress
      containers:
      - image: traefik:latest
        name: traefik-ingress
        ports:
        - name: http
          containerPort: 80
          hostPort: 80
        - name: https
          containerPort: 443
          hostPort: 443
        - name: admin
          containerPort: 8080
        args:
        - --configFile=/etc/traefik/traefik.toml
        - -d
        - --web
        - --kubernetes
        - --logLevel=DEBUG
        volumeMounts:
        - name: traefik-config-volume
          mountPath: /etc/traefik
        - name: traefik-ssl-volume
          mountPath: /ssl
      volumes:
      - name: traefik-config-volume
        configMap:
          name: traefik-conf
          items:
          - key: traefik-config
```

```
            path: traefik.toml
      - name: traefik-ssl-volume
        secret:
          secretName: traefik-ssl
```

注意上面的yaml 文件中我们添加了一个名为`traefik-conf`的`ConfigMap`，该配置是用来将http 请求强制跳转成https，并指定https 所需CA 文件地址，这里我们使用`secret`的形式来指定CA 文件的路径：

```
$ ls
ssl.crt     ssl.key
$ kubectl create secret generic traefik-ssl --from-file=ssl.crt --from-file=ssl.key --
namespace=kube-system
secret "traefik-ssl" created
```

创建ingress

创建文件：`traefik-ingress.yaml`，现在可以通过创建`ingress`文件来定义请求规则了，根据自己集群中的service 自己修改相应的`serviceName` 和`servicePort`

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

执行创建命令：

```
$ kubectl create -f ingress-sa.yaml
serviceaccount "ingress" created
clusterrolebinding "ingress" created
$ kubectl create -f traefik-daemonset.yaml
configmap "traefik-conf" created
daemonset "traefik-ingress" created
$ kubectl create -f traefik-ingress.yaml
ingress "traefik-ingress" created
```

Traefik UI

创建文件：`traefik-ui.yaml`，

```
apiVersion: v1
kind: Service
metadata:
  name: traefik-ui
  namespace: kube-system
spec:
  selector:
    k8s-app: traefik-ingress
  ports:
  - name: web
    port: 80
    targetPort: 8080
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ui
  namespace: kube-system
```

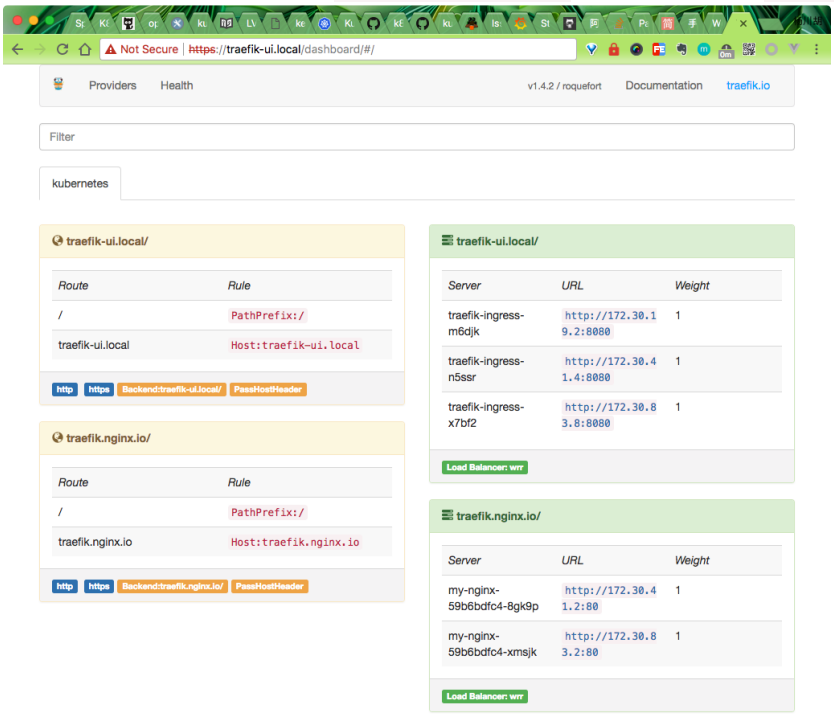
```
spec:
  rules:
  - host: traefik-ui.local
    http:
      paths:
      - path: /
        backend:
          serviceName: traefik-ui
          servicePort: web
```

测试

部署完成后，在本地`/etc/hosts`添加一条配置：

```
# 将下面的xx.xx.xx.xx替换成任意节点IP
xx.xx.xx.xx master03 traefik.nginx.io traefik-ui.local
```

配置完成后，在本地访问：`traefik-ui.local`，则可以访问到`traefik`的`dashboard`页面：



traefik dashboard

同样的可以访问`traefik.nginx.io`，得到正确的结果页面：

WX20171110-140306

上面配置完成后，就可以将我们的所有节点加入到一个SLB中，然后配置相应的域名解析到SLB即可。

13. 日志收集

两种方式：阿里云日志服务或者EFK方案

阿里云日志服务

> *阿里云日志服务（Log Service，简称 Log）是针对日志类数据一站式服务，在阿里巴巴集团经历了大量大数据场景锤炼而成。用户无需开发就能快捷完成数据采集、消费、投递以及查询分析等功能，帮助提升运维、运营效率，建立 DT 时代海量日志处理能力。*

实际上对于小团队自己搭建ELK或者EFK成本是很高的，特别是ElasticSearch本身非常消耗资源不说，能够将其性能优化到一定程度都是非常不容易的一件事情，所以对于中小团队能使用第三方的日志服务更加方便，即使你一定要使用ES的话，我也推荐使用阿里云的ElasticSearch的云服务。

同样的我们可以集成阿里云日志服务来查看和管理您的Kubernetes集群应用的日志。

创建日志项目

首先登录日志服务管理控制台，创建一个项目(Project)：`k8slog-project`

创建日志手机Agent

直接使用阿里云提供的fluentd组件

```
$ curl http://aliacs-k8s.oss.aliyuncs.com/conf%2Flogging%2Ffluentd-pilot.yml > fluentd-pilot.yml
```

将上面yaml 文件中对应的env 环境值更改成实际的值，其中：

- **FLUENTD_OUTPUT：** 固定值 aliyun_sls，代表将日志收集到阿里云日志服务。
- **ALIYUNSLS_PROJECT：** 您第一步创建的阿里云日志服务的 project 名称.
- **ALIYUNSLS_REGION_ENDPOINT:** 日志服务的服务入口。根据您的日志服务所处的地域和网络类型填写日志服务的服务入口，参见 日志服务服务入口。
- **ALIYUNSLS_ACCESS_KEY_ID：** 您的阿里云账号的 access_key_id。
- **ALIYUNSLS_ACCESS_KEY_SECRET:** 您的阿里云账号的 access_key_secret。
- **ALIYUNSLS_NEED_CREATE_LOGSTORE:** 当 Logstore 不存在的时候是否自动创建，`true` 表示自动创建。

```
apiVersion: extensions/v1beta1
kind: DaemonSet
```

```
metadata:
  name: fluentd-pilot
  namespace: kube-system
  labels:
    k8s-app: fluentd-pilot
    kubernetes.io/cluster-service: "true"
spec:
  template:
    metadata:
      labels:
        k8s-app: fluentd-es
        kubernetes.io/cluster-service: "true"
        version: v1.22
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ''
        scheduler.alpha.kubernetes.io/tolerations: '[{"key":
"node.alpha.kubernetes.io/ismaster", "effect": "NoSchedule"}]'
    spec:
      containers:
      - name: fluentd-pilot
        image: registry.cn-hangzhou.aliyuncs.com/wangbs/fluentd-pilot:latest
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        env:
          - name: "FLUENTD_OUTPUT"
            value: "aliyun_sls"
          - name: "ALIYUNSLS_PROJECT"
            value: "k8slog-project"
          - name: "ALIYUNSLS_REGION_ENDPOINT"
            value: "cn-shanghai.log.aliyuncs.com"
          - name: "ALIYUNSLS_ACCESS_KEY_ID"
            value: "xxxxxxx"
          - name: "ALIYUNSLS_ACCESS_KEY_SECRET"
            value: "xxxxxx"
          - name: "ALIYUNSLS_NEED_CREATE_LOGSTORE"
            value: "true"
        volumeMounts:
        - name: sock
          mountPath: /var/run/docker.sock
        - name: root
          mountPath: /host
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: sock
        hostPath:
          path: /var/run/docker.sock
      - name: root
        hostPath:
          path: /
```

创建完成后可以查看运行状态：

```
$ kubectl get ds -n kube-system
NAME                DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE   NODE SELECTOR
AGE
fluentd-pilot       4         4         4         4            4           <none>
5h
```

收集应用日志

为了让`Fluentd`收集您的应用日志，您需要在应用的环境变量中设置参数`aliyun_logs_fluentd=stdout`来启用应用的日志收集功能。其中，`fluentd`是您上面创建的日志Project的`Logstore`，如果该Logstore不存在，系统会自动为您创建该名称的Logstore；`stdout`代表收集标准输出的日志，您还可以配置收集文件日志，具体使用方式请参考Fluentd-pilot。

这里我们按照上面的方式在`kubedns`上面添加环境变量：`aliyun_logs_kubedns=stuout`，然后更新`kubedns`，设置好应用后，就可以前往 日志服务管理控制台 查看并使用日志了。

aliyun log server

EFK 方案

TODO

14. 私有仓库harbor 搭建

先clone harbor 代码到本地，我们可以查看harbor 提供的`kubernetes`部署方案文档：**kubernetes_deployment.md**。

harbor 提供了一个python 脚本(`make/kubernetes/prepare`)来生成`kubernetes`的`ConfigMap`文件，由于这个脚本是python 编写的，所以你需要一个能运行python 的环境，该脚本也需要使用`openssl`来生成私钥和证书，所以也要确保你的运行环境下面有可执行的`openssl`。

下面是该脚本的一些参数：

There are some args of the python script:

- -f: 默认值是`../harbor.cfg`，当然你可以指定另外的配置文件。
- -k: https 私钥路径，你可以在`harbor.cfg`文件中修改`ssl_cert_key`字段。
- -c: https 证书路径，同样可以在`harbor.cfg`文件中修改`ssl_cert`字段。
- -s: 密码路径，必须是16位字符串，如果没有设置，脚本将自动生成。

基本配置

有些基本配置必须设置，否则不能部署成功。

- `make/harbor.cfg`: Harbor 的基本配置文件。
- `make/kubernetes/**/*.rc.yaml`: 一些容器的配置文件。

你需要将所有的`*.rc.yaml`文件中的镜像替换成正确的镜像地址。例如：

```
  containers:
      - name: nginx-app
        # it's very importent that you need modify the path of image.
        image: harbor/nginx
```

- `make/kubernetes/pv/*.pvc.yaml`: **Persistent Volume Claim**。你可以设置存储这些文件的容量，例如：

```
  resources:
    requests:
      # you can set another value to adapt to your needs
      storage: 100Gi
```

- `make/kubernetes/pv/*.pv.yaml`: **Persistent Volume**， 被绑定到上面的*.pvc.yaml`，PV 和 PVC 是一一对应的，如果你改变了PVC 的容量，那么你也需要相应的设置PV 的容量，例如：

```
  capacity:
    # same value with PVC
    storage: 100Gi
```

将上述相关的参数修改完成后，执行下面的命令生成`ConfigMap`文件：

```
python make/kubernetes/prepare
```

脚本执行完成后会生成下面的一些文件：

- make/kubernetes/jobservice/jobservice.cm.yaml
- make/kubernetes/mysql/mysql.cm.yaml
- make/kubernetes/nginx/nginx.cm.yaml
- make/kubernetes/registry/registry.cm.yaml
- make/kubernetes/ui/ui.cm.yaml

高级配置

当然了如果上面的一些基本配置不能满足你的需求，你也可以做一些更高级的配置。你可以在`make/kubernetes/templates`目录下面找到所有的`Harbor`的配置模板：

- `jobservice.cm.yaml`: jobservice 的环境变量和WEB 配置
- `mysql.cm.yaml`: MySQL 的Root 用户密码
- `nginx.cm.yaml`: Https 的证书和nginx 配置
- `registry.cm.yaml`: Token 服务认证和 Registry的相关配置，默认用文件系统来存储镜像数据，你能看到这样的数据： `yaml storage: filesystem: rootdirectory: /storage` 如果你想使用其他的存储后端的话，那么就需要去查看Docker 的相关文档了。
- `ui.cm.yaml`： Token 服务的私钥，UI Dashboard 的环境变量和WEB 配置

ui 和 jobservice 是用`beego`编写的，如果你对`beego`比较熟悉的话，你可以修改相关的配置。

自定义

- 为方便管理，这里我们在模板文件中均添加一个`namespace=kube-ops`，这样让`Harbor`都放在该 namespace 下运行。
- 由于我们的系统是通过`traefik ingress`来和外部进行交流的，所以这里实际上是不需要nginx 这一层的，所以可以将下面nginx 想关的操作移除掉就行。
- 将`make/kubernetes`下面的`*.rc.yaml`文件中的`ReplicationController`改成`Deployment`，因为`Deployment`比`ReplicationController`功能更加丰富, apiVersion 改成对应的`apiVersion: extensions/v1beta1`版本，将sepc 下面的selector 删除。
- 同样的在`make/kubernetes`下面的所有`*.rc.yaml`(除mysql.rc.yaml外)和`*.svc.yaml`文件中添加`namespace=kube-ops`的命名空间。
- 由于mysql 是有状态的应用，所以我们将`mysql.rc.yaml`改成`StatefulSet`，apiVersion 需要改成对应的`apps/v1beta1`版本
- 我们这里使用共享存储`nfs`来存储我们的相关数据，所以我们将`make/kubernetes/pv`下面添加一个`ops.pv.yaml`（忽略其他的pv 相关的文件）：

```
  apiVersion: v1
  kind: PersistentVolume
  metadata:
    name: opspv
    labels:
```

```
      k8s-app: opspv
  spec:
    accessModes:
    - ReadWriteMany
    capacity:
      storage: 100Gi
    persistentVolumeReclaimPolicy: Retain
    nfs:
      path: /ops/data
      server: 192.168.1.139  # 替换成你自己的nfs 服务器地址
  ---
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: opspvc
    namespace: kube-ops
    labels:
      k8s-app: opspvc
  spec:
    accessModes:
    - ReadWriteMany
    resources:
      requests:
        storage: 100Gi
    selector:
      matchLabels:
        k8s-app: opspv
```

- 上面我们新建了一个共享的`pv`和`pvc`，所以需要将`*.rc.yaml`下面所有的**claimName**的值替换成`opspvc`，另外需要在`volumeMounts`声明的地方加上`subPath`来区分存储的文件路径，如下：

```
  volumeMounts:
  - name: logs
    mountPath: /var/log/jobs
    subPath: harbor/logs
volumes:
- name: logs
  persistentVolumeClaim:
    claimName: opspvc
```

- 关于镜像： 可以前往Harbor realse页面下载最新的离线包，然后解压离线安装包可以得到镜像文件`harbor.*.tgz`，然后可以利用下面的`docker`命令将相关的镜像加载进来： `shell $ docker load -i harbor.*.tgz`

由于我们没有指定相关的POD 固定的运行在某个节点上，所以理论上是需要在每个节点上执行上面的步骤的，为了方便，我们这里直接将相关的镜像替换成`vmware`在docker 官方镜像仓库上的镜像 * **adminserver**: 如果你clone 代码下面`make/kubernetes/template`下面没有`adminserver.cm.yaml`文件，则可以前往我整理好的github 仓库下面查看。

上面配置完成的文件可以前往github查看

运行

如果你完成了你的配置并且已经生成了`ConfigMap`文件，现在你可以使用下面的命令来运行`Harbor`了：

```
# create pv & pvc
$ kubectl apply -f make/kubernetes/pv/ops.pv.yaml

# create config map
$ kubectl apply -f make/kubernetes/adminserver/adminserver.cm.yaml
$ kubectl apply -f make/kubernetes/jobservice/jobservice.cm.yaml
$ kubectl apply -f make/kubernetes/mysql/mysql.cm.yaml
$ kubectl apply -f make/kubernetes/registry/registry.cm.yaml
```

```
$ kubectl apply -f make/kubernetes/ui/ui.cm.yaml

# create service
$ kubectl apply -f make/kubernetes/adminserver/adminserver.svc.yaml
$ kubectl apply -f make/kubernetes/jobservice/jobservice.svc.yaml
$ kubectl apply -f make/kubernetes/mysql/mysql.svc.yaml
$ kubectl apply -f make/kubernetes/registry/registry.svc.yaml
$ kubectl apply -f make/kubernetes/ui/ui.svc.yaml

# create k8s deployment/statefulset
$ kubectl apply -f make/kubernetes/adminserver/adminserver.rc.yaml
$ kubectl apply -f make/kubernetes/registry/registry.rc.yaml
$ kubectl apply -f make/kubernetes/mysql/mysql.rc.yaml
$ kubectl apply -f make/kubernetes/jobservice/jobservice.rc.yaml
$ kubectl apply -f make/kubernetes/ui/ui.rc.yaml
```

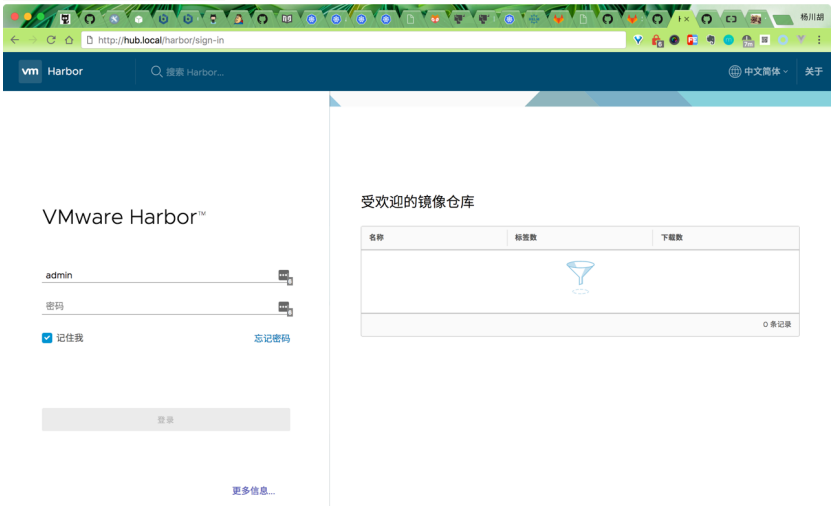
上面的相关yaml 文件执行完成后，我们就可以通过`traefik ingress`给上面的`ui`绑定一个域名:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: traefik-ops
  namespace: kube-ops
spec:
  rules:
  - host: hub.local
    http:
      paths:
      - path: /
        backend:
          serviceName: ui
          servicePort: 80
      - path: /v2/
        backend:
          serviceName: registry
          servicePort: 5000
```

注意上面的`v2`的配置，要在客户端使用docker 命令操作的话就需要配置`v2`的转发

配置完成后我们就可以在本地使用`docker`命令进行登录了:

```
➜  ~ docker login -u admin hub.local
Password:
Login Succeeded
```



Harbor Dashboard

15. 问题汇总

15.1 dashboard无法显示监控图

dashboard 和heapster influxdb都部署完成后 dashboard依旧无法显示监控图 通过排查 heapster log有超时错误

```
$ kubectl logs -f pods/heapster-2882613285-58d9r -n kube-system

E0630 17:23:47.339987 1 reflector.go:203]
k8s.io/heapster/metrics/sources/kubelet/kubelet.go:342: Failed to list *api.Node: Get
http://kubernetes.default/api/v1/nodes?resourceVersion=0: dial tcp: i/o timeout E0630
17:23:47.340274 1 reflector.go:203] k8s.io/heapster/metrics/heapster.go:319: Failed to
list *api.Pod: Get http://kubernetes.default/api/v1/pods?resourceVersion=0: dial tcp:
i/o timeout E0630 17:23:47.340498 1 reflector.go:203]
k8s.io/heapster/metrics/processors/namespace_based_enricher.go:84: Failed to list
*api.Namespace: Get http://kubernetes.default/api/v1/namespaces?resourceVersion=0: dial
tcp: lookup kubernetes.default on 10.254.0.2:53: dial udp 10.254.0.2:53: i/o timeout
E0630 17:23:47.340563 1 reflector.go:203] k8s.io/heapster/metrics/heapster.go:327:
Failed to list *api.Node: Get http://kubernetes.default/api/v1/nodes?resourceVersion=0:
dial tcp: lookup kubernetes.default on 10.254.0.2:53: dial udp 10.254.0.2:53: i/o
timeout E0630 17:23:47.340623 1 reflector.go:203]
k8s.io/heapster/metrics/processors/node_autoscaling_enricher.go💯 Failed to list
*api.Node: Get http://kubernetes.default/api/v1/nodes?resourceVersion=0: dial tcp:
lookup kubernetes.default on 10.254.0.2:53: dial udp 10.254.0.2:53: i/o timeout E0630
17:23:55.014414 1 influxdb.go:150] Failed to create infuxdb: failed to ping InfluxDB
server at "monitoring-influxdb:8086" - Get http://monitoring-influxdb:8086/ping: dial
tcp: lookup monitoring-influxdb on 10.254.0.2:53: read udp 172.30.45.4:48955-
>10.254.0.2:53: i/o timeout`
```

我是docker的systemd Unit文件忘记添加

```
ExecStart=/root/local/bin/dockerd --log-level=error $DOCKER_NETWORK_OPTIONS
```

后边的`$DOCKER_NETWORK_OPTIONS`，导致`docker0`的网段跟`flannel.1`不一致。

15.2 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。

15.3 Unable to access kubernetes services: no route to host

导致现象: 在POD 内访问集群的某个服务的时候出现`no route to host`

```
$ curl my-nginx.nx.svc.cluster.local
curl: (7) Failed connect to my-nginx.nx.svc.cluster.local:80; No route to host
```

解决方法：清除所有的防火墙规则，然后重启docker 服务

```
$ iptables --flush && iptables -tnat --flush
$ systemctl restart docker
```

15.4 使用NodePort 类型的服务，只能在POD 所在节点进行访问

导致现象: 使用`NodePort` 类型的服务，只能在POD 所在节点进行访问，其他节点通过NodePort 不能正常访问

解决方法: `kube-proxy` 默认使用的是`proxy_model`就是`iptables`，正常情况下是所有节点都可以通过NodePort 进行访问的，我这里将阿里云的安全组限制全部去掉即可，然后根据需要进行添加安全限制。

参考资料

- 和我一步步部署 kubernetes 集群
- keepalived 配置
- kubernetes issue
- kubernetes heapster issue