Kubernetes 实践指南 (Kubernetes Practice

书栈(BookStack.CN)

目　录

致谢

当前文档 《Kubernetes 实践指南（Kubernetes Practice Guide）》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-11-16。

序言

Roadmap

本书正在起草初期，内容将包含大量 Kubernetes 实践干货，大量规划内容还正在路上，可以点击的表示是已经可以在左侧导航栏中找到的并预览的文章，但不代表已经完善，还会不断的补充和优化。

部署指南

自建的 k8s 集群有很多种方式部署方式，k8s 知识库将列举手工二进制部署与各种辅助工具部署的方法，可以根据自己使用场景选择对应合适的部署方法。除此之外，还会包含大量的常用应用的部署方法，比如各种数据库和存储基础设施部署，不同的业务场景和解决方案都可能依赖这些应用，每种应用部署方法都可能被书内其它多处地方引用。

- 部署方案选型
- 单机部署
- 手工部署
    - 部署前的准备工作
    - 部署 ETCD
    - 部署 Master
    - 部署 Worker 节点
    - 部署关键附加组件
- 使用 Kubeadm 部署集群
- 使用 Minikube 部署测试集群
- 使用 Bootkube 部署集群
- 使用 Ansible 部署集群
- 部署附加组件
    - 部署 CoreDNS
    - 以 Daemonset 方式部署 kube-proxy
- 常见应用部署
    - ElasticSearch 与 Kibana
        - 使用 elastic-oparator 部署
    - ETCD
    - Zookeeper
    - Consul
    - Kafka
    - Redis
    - TiKV

    - MySQL
    - TiDB
    - PostgreSQL
    - MongoDB
    - Cassandra
    - InfluxDB
    - OpenTSDB

集群方案

k8s 拥有惊人的扩展能力，针对不同环境和场景可以使用不同的方案，涵盖网络、存储、运行时、Ingress、Metrics 等。k8s 知识库会帮助你彻底理清这些机制，并深入剖析各种方案的原理、用法与使用场景。

- 网络方案
    - 彻底理解集群网络
    - Network Policy
    - 开源网络方案
    - Flannel
        - Flannel 网络原理
        - 部署 Flannel
    - Macvlan
    - Calico
    - Cilium
    - Kube-router
    - Kube-OVN
    - OpenVSwitch
- 运行时方案
    - Docker
        - Docker 介绍
        - Docker 安装
    - Containerd
        - containerd 介绍
        - 安装 containerd
    - CRI-O
        - CRI-O 介绍
        - CRI-O 安装
- 存储方案
    - Rook
    - OpenEBS

- Ingress 方案
    - Nginx Ingress
        - 安装 nginx ingress controller
    - Traefik Ingress
        - 安装 traefik ingress controller
    - Contour
    - Ambassador
    - Kong
    - Gloo
    - HAProxy
    - Istio
    - Skipper
- LoadBalancer 方案
    - MetalLB
    - Porter
- Metrics 方案
    - 安装 metrics server

最佳实践

k8s 有先进的设计理念，也包含了大量概念，并提供了非常丰富的能力，用法琳琅满目，但入门比较困难，k8s 知识库将提供使用 k8s 的各种场景里的最佳实践，帮助大家少走弯路，比如如何管理和运维集群、如何进行动态伸缩、如何保证部署的服务高可用、如何在更新服务或扩缩容节点保证业务零感知、如何部署有状态服务、如何针对大规模集群进行优化、如何对资源进行隔离和共享以及针对各种需求和问题的解决方案等。

- 服务高可用
- 本地 DNS 缓存
- 泛域名动态转发 Service
- 集群权限控制
    - 利用 CSR API 创建用户
    - 控制用户权限
    - 控制应用权限
- 有状态服务部署
- 实用工具和技巧
    - kubectl 高效技巧
    - 实用 yaml 片段
    - 实用命令与脚本
- 集群证书管理
    - 安装 cert-manager

    - 使用 cert-manager 自动生成证书
- 集群配置管理
    - Helm
        - 安装 Helm
        - Helm V2 迁移到 V3
        - 使用 Helm 部署与管理应用
        - 开发 Helm Charts
    - Kustomize
        - Kustomize 基础入门
- 大规模集群优化
- 弹性伸缩
    - 使用 HPA 对 Pod 水平伸缩
    - 使用 VPA 对 Pod 垂直伸缩
    - 使用 Cluster Autoscaler 对节点水平伸缩
- 资源分配与限制
- Master 高可用
- 资源隔离与共享
    - 利用 kata-container 隔离容器资源
    - 利用 gVisor 隔离容器资源
    - 利用 lvm 和 xfs 实现容器磁盘隔离
    - 利用 lxcfs 隔离 proc 提升容器资源可见性
    - 共享存储
    - 共享内存
- Pod 原地重启
- 集群升级
- 固定 IP
- 备份与恢复
- ETCD 性能优化
- 内核参数优化
- CPU 亲和性
- 使用大页内存
- 离在线混合部署
- 集群监控
    - Prometheus
    - Grafana
- 日志搜集
    - EFK/ELK
- 集群安全
    - 使用 PodSecurityPolicy 配置全局 Pod 安全策略

    - 集群审计
- 集群可视化管理
    - Kubernetes Dashboard
    - KubSphere
    - Weave Scope
    - Rancher
    - Kui
    - Kubebox
- 集群镜像管理
    - Harbor
    - Dragonfly
    - Kaniko
    - kpack

排错指南

正是 k8s 功能如此丰富强大，迭代速度如此之快，其复杂性和不确定性也非常之大。知识库会总结出各种问题的排查思路与可能原因，还有对应解决方案的最佳实践，也分享一些踩坑案例与排错技巧，与排错技巧，让大家少走弯路。

- 问题排查
    - Pod 排错
        - Pod 一直处于 Pending 状态
        - Pod 一直处于 ContainerCreating 或 Waiting 状态
        - Pod 一直处于 CrashLoopBackOff 状态
        - Pod 一直处于 Terminating 状态
        - Pod 一直处于 Unknown 状态
        - Pod 一直处于 Error 状态
        - Pod 一直处于 ImagePullBackOff 状态
        - Pod 一直处于 ImageInspectError 状态
        - Pod 健康检查失败
        - 容器进程主动退出
    - 网络排错
        - LB 健康检查失败
        - DNS 解析异常
        - Service 不通
        - 网络性能差
    - 集群排错
        - Node 全部消失
        - Daemonset 没有被调度

        - Apiserver 响应慢
        - ETCD 频繁选主
        - Node 异常
    - 经典报错
        - no space left on device
        - arp_cache: neighbor table overflow!
        - Cannot allocate memory
    - 其它排错
        - Job 无法被删除
        - kubectl 执行 exec 或 logs 失败
        - 内核软死锁
- 处理实践
    - 高负载
    - 内存碎片化
    - 磁盘爆满
    - inotify watch 耗尽
    - PID 耗尽
    - arp_cache 溢出
- 踩坑总结
    - cgroup 泄露
    - tcp_tw_recycle 引发丢包
    - 使用 oom-guard 在用户态处理 cgroup OOM
    - conntrack 冲突导致丢包
- 案例分享
    - 驱逐导致服务中断
    - DNS 5 秒延时
    - arp_cache 溢出导致健康检查失败
    - 跨 VPC 访问 NodePort 经常超时
    - 访问 externalTrafficPolicy 为 Local 的 Service 对应 LB 有时超时
    - Pod 偶尔存活检查失败
    - DNS 解析异常
    - Pod 访问另一个集群的 apiserver 有延时
    - LB 压测 NodePort CPS 低
    - kubectl edit 或者 apply 报 SchemaError
- 排错技巧
    - 分析 ExitCode 定位 Pod 异常退出原因
    - 容器内抓包定位网络问题
    - 使用 Systemtap 定位疑难杂症
    - 使用 kubectl-debug 帮助定位问题

    - 分析 Docker 磁盘占用

领域应用

k8s 在各个领域都发挥了巨大作用，我们会将 k8s 在这些领域的应用汇总，给出各种场景化应用的指南，比如近年来如火如荼的 DevOps 领域，其中 CI/CD 的应用更是大家迫切期望想要的。还有 AI，大数据，微服务架构，Service Mesh，Serverless 等。

- 微服务架构
    - 服务发现
    - 服务治理
    - 分布式追踪
        - Jaeger
- Service Mesh
    - Istio
    - Maesh
    - Kuma
- Serverless
    - Knative
    - Kubeless
    - Fission
- DevOps
    - Jenkins X
    - Tekton
    - Argo
    - GoCD
    - Argo
    - GitLab CI
    - Drone
- AI
    - nvidia-docker
    - Kubeflow
- 大数据
    - Spark on K8S
    - Hadoop on K8S

开发指南

k8s 开放了很多扩展能力，基于这些扩展机制可以开发出各种功能的应用，比如集群管理应用、部署有状态服务的应用（Operator）等，知识库将介绍如何开发这些应用。

- 开发环境搭建
- Go 语言编译原理与优化
- Operator
    - Operator 概述
    - operator-sdk
    - kubebuilder
- client-go
- 社区贡献

在线阅读

本书将支持中英文两个语言版本，通常文章会先用中文起草并更新，等待其内容较为成熟完善，更新不再频繁的时候才会翻译成英文，点击左上角切换语言。

- 中文: https://k8s.imroc.io
- English: https://k8s.imroc.io/v/en

项目源码

项目源码存放于 Github 上: https://github.com/imroc/kubernetes-practice-guide

贡献

欢迎参与贡献和完善内容，贡献方法参考 CONTRIBUTING

License

手工部署

部署详情

各组件版本：

- kubernetes 1.16.1
- containerd 1.3.0
- coredns v1.6.2
- cni v0.8.2
- flannel v0.11.0
- etcd v3.4.1

特点：

- kubelet 证书自动签发并轮转
- kube-proxy 以 daemonset 方式部署，无需为其手动签发管理证书
- 运行时没有 docker 直接使用 containerd，绕过 dockerd 的许多 bug

部署步骤

部署前的准备工作

准备节点

操作系统

使用 Linux 发行版，本教程主要以 Ubuntu 18.04 为例

Master 节点

部署 K8S 控制面组件，推荐三台以上数量的机器

ETCD 节点

部署 ETCD，可以跟 Master 节点用相同的机器，也可以用单独的机器，推荐三台以上数量的机器

Worker 节点

实际运行工作负载的节点，Master 节点也可以作为 Worker 节点，可以通过 kubelet 参数 `--kube-reserved` 多预留一些资源给系统组件。

通常会给 Master 节点打标签，让关键的 Pod 跑在 Master 节点上，比如集群 DNS 服务。

准备客户端工具

我们需要用 `cfssl` 和 `kubectl` 来为各个组件生成证书和 kubeconfig，所以先将这两个工具在某个机器下载安装好。

" class="reference-link">安装 cfssl

```
curl -L https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 -o cfssl
curl -L https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64 -o cfssljson
curl -L https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64 -o cfssl-certinfo

chmod +x cfssl cfssljson cfssl-certinfo
sudo mv cfssl cfssljson cfssl-certinfo /usr/local/bin/
```

安装 kubectl

```
wget -q --show-progress --https-only --timestamping \
  https://storage.googleapis.com/kubernetes-
release/release/v1.16.1/bin/linux/amd64/kubectl

chmod +x kubectl
mv kubectl /usr/local/bin/
```

" class="reference-link">生成 CA 证书

由于各个组件都需要配置证书，并且依赖 CA 证书来签发证书，所以我们首先要生成好 CA 证书以及后续的签发配置文件：

```
cat > ca-csr.json <<EOF
{
  "CN": "Kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "SiChuan",
      "L": "ChengDu",
      "O": "Kubernetes",
      "OU": "CA"
    }
  ]
}
EOF

cfssl gencert -initca ca-csr.json | cfssljson -bare ca

cat > ca-config.json <<EOF
{
  "signing": {
    "default": {
      "expiry": "8760h"
    },
    "profiles": {
```

```
29.       "kubernetes": {
            "usages": ["signing", "key encipherment", "server auth", "client
30. auth"],
31.         "expiry": "8760h"
32.       }
33.     }
34.   }
35. }
36. EOF
```

生成的文件中有下面三个后面会用到：

- `ca-key.pem`：CA 证书密钥
- `ca.pem`：CA 证书
- `ca-config.json`：证书签发配置

csr 文件字段解释：

- `CN`：`Common Name`，apiserver 从证书中提取该字段作为请求的用户名 (User Name)
- `Organization`，apiserver 从证书中提取该字段作为请求用户所属的组 (Group)

> 由于这里是 CA 证书，是签发其它证书的根证书，这个证书密钥不会分发出去作为 client 证书，所有组件使用的 client 证书都是由 CA 证书签发而来，所以 CA 证书的 CN 和 O 的名称并不重要，后续其它签发出来的证书的 CN 和 O 的名称才是有用的

部署 ETCD

为 ETCD 签发证书

这里证书可以只创建一次，所有 etcd 实例都公用这里创建的证书：

```
cat > etcd-csr.json <<EOF
{
    "CN": "etcd",
    "hosts": [
      "127.0.0.1",
      "10.200.16.79",
      "10.200.17.6",
      "10.200.16.70"
    ],
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
        {
            "C": "CN",
            "ST": "SiChuan",
            "L": "Chengdu",
            "O": "etcd",
            "OU": "etcd"
        }
    ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  etcd-csr.json | cfssljson -bare etcd
```

> hosts 需要包含 etcd 每个实例所在节点的内网 IP

会生成下面两个重要的文件：

- `etcd-key.pem`：kube-apiserver 证书密钥
- `etcd.pem`：kube-apiserver 证书

下载安装 ETCD

下载 release 包：

```
wget -q --show-progress --https-only --timestamping \
  "https://github.com/etcd-io/etcd/releases/download/v3.4.1/etcd-v3.4.1-linux-amd64.tar.gz"
```

解压安装 `etcd` 和 `etcdctl` 到 PATH：

```
tar -xvf etcd-v3.4.1-linux-amd64.tar.gz
sudo mv etcd-v3.4.1-linux-amd64/etcd* /usr/local/bin/
```

配置

创建配置相关目录，放入证书文件：

```
sudo mkdir -p /etc/etcd /var/lib/etcd
sudo cp ca.pem etcd-key.pem etcd.pem /etc/etcd/
```

etcd 集群每个成员都需要一个名字，这里第一个成员名字用 infra0，第二个可以用 infra1，以此类推，你也可以直接用节点的 hostname：

```
NAME=infra0
```

记当前部署 ETCD 的节点的内网 IP 为 INTERNAL_IP：

```
INTERNAL_IP=10.200.16.79
```

记所有 ETCD 成员的名称和成员间通信的 https 监听地址为 ETCD_SERVERS（注意是 2380 端口，不是 2379）：

```
ETCD_SERVERS="infra0=https://10.200.16.79:2380,infra1=https://10.200.17.6:2380,ir
```

创建 systemd 配置:

```
cat <<EOF | sudo tee /etc/systemd/system/etcd.service
[Unit]
Description=etcd
Documentation=https://github.com/coreos

[Service]
Type=notify
ExecStart=/usr/local/bin/etcd \\
  --name ${NAME} \\
  --cert-file=/etc/etcd/etcd.pem \\
  --key-file=/etc/etcd/etcd-key.pem \\
  --peer-cert-file=/etc/etcd/etcd.pem \\
  --peer-key-file=/etc/etcd/etcd-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \\
  --advertise-client-urls https://${INTERNAL_IP}:2379 \\
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster ${ETCD_SERVERS} \\
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
Restart=on-failure
RestartSec=5

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

启动

```
sudo systemctl daemon-reload
sudo systemctl enable etcd
sudo systemctl start etcd
```

验证

等所有 etcd 成员安装启动成功后，来验证下是否可用：

```
sudo ETCDCTL_API=3 etcdctl member list \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/etcd/ca.pem \
  --cert=/etc/etcd/etcd.pem \
  --key=/etc/etcd/etcd-key.pem
```

输出：

```
a7f995caeeaf7a59, started, infra1, https://10.200.17.6:2380,
https://10.200.17.6:2379, false
b90901a06e9aec53, started, infra2, https://10.200.16.70:2380,
https://10.200.16.70:2379, false
ba126eb695f5ba71, started, infra0, https://10.200.16.79:2380,
https://10.200.16.79:2379, false
```

部署 Master

" class="reference-link">准备证书

Master 节点的准备证书操作只需要做一次，将生成的证书拷到每个 Master 节点上以复用。

前提条件：

- 签发证书需要用到 生成 CA 证书 时创建的 CA 证书及其密钥文件，确保它们在当前目录
- 确保 cfssl 在当前环境已安装，安装方法参考 这里

" class="reference-link">为 kube-apiserver 签发证书

kube-apiserver 是 k8s 的访问核心，所有 K8S 组件和用户 kubectl 操作都会请求 kube-apiserver，通常启用 tls 证书认证，证书里面需要包含 kube-apiserver 可能被访问的地址，这样 client 校验 kube-apiserver 证书时才会通过，集群内的 Pod 一般通过 kube-apiserver 的 Service 名称访问，可能的 Service 名称有：

- `kubernetes`
- `kubernetes.default`
- `kubernetes.default.svc`
- `kubernetes.default.svc.cluster`
- `kubernetes.default.svc.cluster.local`

通过集群外也可能访问 kube-apiserver，比如使用 kubectl，或者部署在集群外的服务会连 kube-apiserver（比如部署在集群外的 Promethues 采集集群指标做监控），这里列一下通过集群外连 kube-apiserver 有哪些可能地址：

- `127.0.0.1`：在 Master 所在机器通过 127.0.0.1 访问本机 kube-apiserver
- Service CIDR 的第一个 IP，比如 flanneld 以 daemonset 部署在每个节点，使用 hostNetwork 而不是集群网络，这时无法通过 service 名称访问 apiserver，因为使用 hostNetwork 无法解析 service 名称（使用的 DNS 不是集群 DNS），它会使用 apiserver 内部的 CLUSTER IP 去请求 apiserver。 kube-controller-manager 的 `--service-cluster-ip-range` 启动参数是 `10.32.0.0/16`，那么第一个 IP 就是 `10.32.0.1`
- 自定义域名：配了 DNS，通过域名访问 kube-apiserver，也要将域名写入证书
- LB IP：如果 Master 节点前面挂了一个负载均衡器，外界可以通过 LB IP 来访问 kube-apiserver
- Master 节点 IP：如果没有 Master 负载均衡器，管理员在节点上执行 kubectl 通常使用 Master 节点 IP 访问 kube-apiserver

准备 CSR 文件：

```
cat > apiserver-csr.json <<EOF
{
    "CN": "kubernetes",
    "hosts": [
      "127.0.0.1",
      "10.32.0.1",
      "10.200.16.79",
      "kubernetes",
      "kubernetes.default",
      "kubernetes.default.svc",
      "kubernetes.default.svc.cluster",
      "kubernetes.default.svc.cluster.local"
    ],
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
        {
            "C": "CN",
            "ST": "SiChuan",
            "L": "Chengdu",
            "O": "Kubernetes",
            "OU": "Kube API Server"
        }
    ]
}
EOF
```

> hosts 这里只准备了必要的，根据需求可增加，通常 Master 节点 IP 也都要加进去，你可以执行了上面的命令后再编辑一下 `apiserver-csr.json` ，将需要 hosts 都加进去。

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  apiserver-csr.json | cfssljson -bare apiserver
```

会生成下面两个重要的文件：

- `apiserver-key.pem`：kube-apiserver 证书密钥
- `apiserver.pem`：kube-apiserver 证书

" class="reference-link">为 kube-controller-manager 签发证书

```
cat > kube-controller-manager-csr.json <<EOF
{
  "CN": "system:kube-controller-manager",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "SiChuan",
      "L": "Chengdu",
      "O": "system:kube-controller-manager",
      "OU": "Kube Controller Manager"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  kube-controller-manager-csr.json | cfssljson -bare kube-controller-manager
```

生成以下两个文件：

- `kube-controller-manager-key.pem`：kube-controller-manager 证书密钥
- `kube-controller-manager.pem`：kube-controller-manager 证书

" class="reference-link">为 kube-scheduler 签发证书

```
cat > kube-scheduler-csr.json <<EOF
{
```

```
  "CN": "system:kube-scheduler",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "SiChuan",
      "L": "Chengdu",
      "O": "system:kube-scheduler",
      "OU": "Kube Scheduler"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  kube-scheduler-csr.json | cfssljson -bare kube-scheduler
```

生成以下两个文件：

- `kube-scheduler-key.pem`：kube-scheduler 证书密钥
- `kube-scheduler.pem`：kube-scheduler 证书公钥

" class="reference-link">签发 Service Account 密钥对

`kube-controller-manager` 会使用此密钥对来给 service account 签发 token，更多详情参考官方文档：https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/

```
cat > service-account-csr.json <<EOF
{
  "CN": "service-accounts",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
```

```
  "names": [
    {
      "C": "CN",
      "ST": "SiChuan",
      "L": "Chengdu",
      "O": "Kubernetes",
      "OU": "Service Account"
    }
  ]
}
EOF

cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  service-account-csr.json | cfssljson -bare service-account
```

生成以下两个文件：

- `service-account-key.pem` : service account 证书公钥
- `service-account.pem` : service account 证书私钥

" class="reference-link">为管理员签发证书

为最高权限管理员证书：

```
cat > admin-csr.json <<EOF
{
  "CN": "admin",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "CN",
      "ST": "SiChuan",
      "L": "Chengdu",
      "O": "system:masters",
      "OU": "System"
```

```
    }
  ]
}
EOF

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

生成一下两个文件：

- `admin-key.pem`：管理员证书密钥
- `admin.pem`：管理员证书

给用户签发证书后，用户访问 kube-apiserver 的请求就带上此证书，kube-apiserver 校验成功后表示认证成功，但还需要授权才允许访问，kube-apiserver 会提取证书中字段 `CN` 作为用户名，这里用户名叫 `admin`，但这只是个名称标识，它有什么权限呢？`admin` 是预置最高权限的用户名吗？不是的！不过 kube-apiserver 确实预置了一个最高权限的 `ClusterRole`，叫做 `cluster-admin`，还有个预置的 `ClusterRoleBinding` 将 `cluster-admin` 这个 `ClusterRole` 与 `system:masters` 这个用户组关联起来了，所以说我们给用户签发证书只要在 `system:masters` 这个用户组就拥有了最高权限。

以此类推，我们签发证书时也可以将用户设置到其它用户组，然后为其创建 RBAC 规则来细粒度的控制权限，减少安全隐患。

更多 K8S 预置的 Role 与 RoleBinding 请参考：
https://kubernetes.io/docs/reference/access-authn-authz/rbac/#default-roles-and-role-bindings

" class="reference-link">准备 kubeconfig

部署 Master 的准备 kubeconfig 操作只需要做一次，将生成的 kubeconfig 拷到每个 Master 节点上以复用。

`kubeconfig` 主要是各组件以及用户访问 apiserver 的必要配置，包含 apiserver 地址、client 证书与 CA 证书等信息。下面介绍为各个组件生成 `kubeconfig` 的方法。

前提条件：

- 我们使用 `kubectl` 来辅助生成 kubeconfig，确保 kubectl 已安装。

- 生成 kubeconfig 会用到之前准备证书时创建的证书与密钥，确保这些生成的文件在当前目录。

确定 apiserver 访问入口

所有组件都会去连 apiserver，所以首先需要确定你的 apiserver 访问入口的地址：

- 如果所有 master 组件都部署在一个节点，它们可以通过 127.0.0.1 这个 IP访问 apiserver。
- 如果 master 有多个节点，但 apiserver 只有一个实例，可以直接写 apiserver 所在机器的内网 IP 访问地址。
- 如果做了高可用，有多个 apiserver 实例，前面挂了负载均衡器，就可以写负载均衡器的访问地址。
- 入口地址的域名或IP必须是在之前 为 kube-apiserver 签发证书 的 hosts 列表里。

这里我们用 `APISERVER` 这个变量表示 apiserver 的访问地址，其它组件都需要配置这个地址，根据自身情况改下这个变量的值：

```
APISERVER="https://10.200.16.79:6443"
```

" class="reference-link">为 kube-controller-manager 创建 kubeconfig

```
APISERVER="https://10.200.16.79:6443"
```

```
kubectl config set-cluster roc \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=${APISERVER} \
  --kubeconfig=kube-controller-manager.kubeconfig

kubectl config set-credentials system:kube-controller-manager \
  --client-certificate=kube-controller-manager.pem \
  --client-key=kube-controller-manager-key.pem \
  --embed-certs=true \
  --kubeconfig=kube-controller-manager.kubeconfig

kubectl config set-context default \
  --cluster=roc \
  --user=system:kube-controller-manager \
  --kubeconfig=kube-controller-manager.kubeconfig
```

```
17.
    kubectl config use-context default --kubeconfig=kube-controller-
18. manager.kubeconfig
```

生成文件：

```
1. kube-controller-manager.kubeconfig
```

" class="reference-link">为 kube-scheduler 创建 kubeconfig

```
1. APISERVER="https://10.200.16.79:6443"
```

```
1. kubectl config set-cluster roc \
2.   --certificate-authority=ca.pem \
3.   --embed-certs=true \
4.   --server=${APISERVER} \
5.   --kubeconfig=kube-scheduler.kubeconfig
6.
7. kubectl config set-credentials system:kube-scheduler \
8.   --client-certificate=kube-scheduler.pem \
9.   --client-key=kube-scheduler-key.pem \
10.  --embed-certs=true \
11.  --kubeconfig=kube-scheduler.kubeconfig
12.
13. kubectl config set-context default \
14.  --cluster=roc \
15.  --user=system:kube-scheduler \
16.  --kubeconfig=kube-scheduler.kubeconfig
17.
18. kubectl config use-context default --kubeconfig=kube-scheduler.kubeconfig
```

生成文件：

```
1. kube-scheduler.kubeconfig
```

" class="reference-link">为管理员创建 kubeconfig

这里为管理员生成 kubeconfig，方便使用 kubectl 来管理集群：

```
APISERVER="https://10.200.16.79:6443"
```

```
kubectl config set-cluster roc \
  --certificate-authority=ca.pem \
  --embed-certs=true \
  --server=${APISERVER} \
  --kubeconfig=admin.kubeconfig

kubectl config set-credentials admin \
  --client-certificate=admin.pem \
  --client-key=admin-key.pem \
  --embed-certs=true \
  --kubeconfig=admin.kubeconfig

kubectl config set-context default \
  --cluster=roc \
  --user=admin \
  --kubeconfig=admin.kubeconfig

kubectl config use-context default --kubeconfig=admin.kubeconfig
```

生成文件：

```
admin.kubeconfig
```

将 `admin.kubeconfig` 放到需要执行 kubectl 的机器的 `~/.kube/config` 这个目录，这是 kubectl 读取 kubeconfig 的默认路径，执行 kubectl 时就不需要指定 kubeconfig 路径了：

```
mv admin.kubeconfig ~/.kube/config
```

下载安装控制面组件

```
wget -q --show-progress --https-only --timestamping \
  https://storage.googleapis.com/kubernetes-release/release/v1.16.1/bin/linux/amd
apiserver \
  https://storage.googleapis.com/kubernetes-release/release/v1.16.1/bin/linux/amd
manager \
  https://storage.googleapis.com/kubernetes-release/release/v1.16.1/bin/linux/amd
scheduler

```

```
chmod +x kube-apiserver kube-controller-manager kube-scheduler
mv kube-apiserver kube-controller-manager kube-scheduler /usr/local/bin/
```

" class="reference-link">配置控制面组件

准备配置相关目录:

```
sudo mkdir -p /etc/kubernetes/config
sudo mkdir -p /var/lib/kubernetes
```

确定集群的集群网段 (Pod IP 占用网段)和 serivce 网段 (service 的 cluster ip 占用网段), 它们可以没有交集。

记集群网段为 CLUSTER_CIDR:

```
CLUSTER_CIDR=10.10.0.0/16
```

记 service 网段为 SERVICE_CIDR:

```
SERVICE_CIDR=10.32.0.0/16
```

" class="reference-link">配置 kube-apiserver

放入证书文件:

```
sudo cp ca.pem ca-key.pem apiserver-key.pem apiserver.pem \
  service-account-key.pem service-account.pem /var/lib/kubernetes/
```

记所有 ETCD 实例的访问地址为 `ETCD_SERVERS` (替换 IP 为所有 ETCD 节点内网 IP):

```
ETCD_SERVERS="https://10.200.16.79:2379,https://10.200.17.6:2379,https://10.200.1
```

记当前节点内网 IP 为 INTERNAL_IP:

```
INTERNAL_IP=10.200.16.79
```

配置 systemd:

```
cat <<EOF | sudo tee /etc/systemd/system/kube-apiserver.service
```

```
[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/kubernetes/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  --enable-bootstrap-token-auth=true \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --audit-log-maxage=30 \\
  --audit-log-maxbackup=3 \\
  --audit-log-maxsize=100 \\
  --audit-log-path=/var/log/audit.log \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --enable-admission-
plugins=NamespaceLifecycle,NodeRestriction,LimitRanger,ServiceAccount,DefaultStor
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/apiserver.pem \\
  --etcd-keyfile=/var/lib/kubernetes/apiserver-key.pem \\
  --etcd-servers=${ETCD_SERVERS} \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/apiserver.pem \\
  --kubelet-client-key=/var/lib/kubernetes/apiserver-key.pem \\
  --kubelet-https=true \\
  --runtime-config=api/all \\
  --service-account-key-file=/var/lib/kubernetes/service-account.pem \\
  --service-cluster-ip-range=${SERVICE_CIDR} \\
  --service-node-port-range=30000-32767 \\
  --tls-cert-file=/var/lib/kubernetes/apiserver.pem \\
  --tls-private-key-file=/var/lib/kubernetes/apiserver-key.pem \\
  --v=2
Restart=on-failure
RestartSec=5

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

- `--enable-bootstrap-token-auth=true` 启用 bootstrap token 方式为 kubelet 签发证书

" class="reference-link">配置 kube-controller-manager

放入 kubeconfig:

```
sudo cp kube-controller-manager.kubeconfig /var/lib/kubernetes/
```

准备 systemd 配置 `kube-controller-manager.service` :

```
CLUSTER_CIDR=10.10.0.0/16
SERVICE_CIDR=10.32.0.0/16
```

```
cat <<EOF | sudo tee /etc/systemd/system/kube-controller-manager.service
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/kubernetes/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-controller-manager \\
  --address=0.0.0.0 \\
  --cluster-cidr=${CLUSTER_CIDR} \\
  --allocate-node-cidrs \\
  --cluster-name=kubernetes \\
  --cluster-signing-cert-file=/var/lib/kubernetes/ca.pem \\
  --cluster-signing-key-file=/var/lib/kubernetes/ca-key.pem \\
  --kubeconfig=/var/lib/kubernetes/kube-controller-manager.kubeconfig \\
  --leader-elect=true \\
  --root-ca-file=/var/lib/kubernetes/ca.pem \\
  --service-account-private-key-file=/var/lib/kubernetes/service-account-
key.pem \\
  --service-cluster-ip-range=${SERVICE_CIDR} \\
  --use-service-account-credentials=true \\
  --v=2
Restart=on-failure
RestartSec=5

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

> 所有 kube-controller-manager 实例都使用相同的 systemd service 文件，可以直接将这里创建好的拷贝给其它 Master 节点

" class="reference-link">配置 kube-scheduler

放入 kubeconfig:

```
sudo cp kube-scheduler.kubeconfig /var/lib/kubernetes/
```

准备启动配置文件 `kube-scheduler.yaml` :

```
cat <<EOF | sudo tee /etc/kubernetes/config/kube-scheduler.yaml
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: "/var/lib/kubernetes/kube-scheduler.kubeconfig"
leaderElection:
  leaderElect: true
EOF
```

准备 systemd 配置 `kube-scheduler.service` :

```
cat <<EOF | sudo tee /etc/systemd/system/kube-scheduler.service
[Unit]
Description=Kubernetes Scheduler
Documentation=https://github.com/kubernetes/kubernetes

[Service]
ExecStart=/usr/local/bin/kube-scheduler \\
  --config=/etc/kubernetes/config/kube-scheduler.yaml \\
  --v=2
Restart=on-failure
RestartSec=5

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

启动

```
sudo systemctl daemon-reload
sudo systemctl enable kube-apiserver kube-controller-manager kube-scheduler
sudo systemctl start kube-apiserver kube-controller-manager kube-scheduler
```

RBAC 授权 kube-apiserver 访问 kubelet

kube-apiserver 有些情况也会访问 kubelet，比如获取 metrics、查看容器日志或登录容器，这是 kubelet 作为 server， kube-apiserver 作为 client，kubelet 监听的 https，kube-apiserver 经过证书认证访问 kubelet，但还需要经过授权才能成功调用接口，我们通过创建 RBAC 规则授权 kube-apiserver 访问 kubelet:

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-apiserver-to-kubelet
rules:
  - apiGroups:
      - ""
    resources:
      - nodes/proxy
      - nodes/stats
      - nodes/log
      - nodes/spec
      - nodes/metrics
    verbs:
      - "*"
EOF

cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: system:kube-apiserver
  namespace: ""
roleRef:
  apiGroup: rbac.authorization.k8s.io
```

```
  kind: ClusterRole
  name: system:kube-apiserver-to-kubelet
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: User
    name: kubernetes
EOF
```

" class="reference-link">RBAC 授权 kubelet 创建 CSR 与自动签发和更新证书

节点 kubelet 通过 Bootstrap Token 调用 apiserver CSR API 请求签发证书，kubelet 通过 bootstrap token 认证后会在 `system:bootstrappers` 用户组里，我们还需要给它授权调用 CSR API，为这个用户组绑定预定义的 `system:node-bootstrapper` 这个 ClusterRole 就可以：

```
cat <<EOF | kubectl apply -f -
# enable bootstrapping nodes to create CSR
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: create-csrs-for-bootstrapping
subjects:
- kind: Group
  name: system:bootstrappers
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: system:node-bootstrapper
  apiGroup: rbac.authorization.k8s.io
EOF
```

> 这里的 CSR API 主要用于 kubelet 发起 client 和 server 证书签发请求

给 kubelet 授权自动审批通过 client 证书的 CSR 请求权限以实现自动创建新的 client 证书(之前没创建过 client 证书，通过 bootstrap token 认证后在 `system:bootstrappers` 用户组里)：

```
cat <<EOF | kubectl apply -f -
# Approve all CSRs for the group "system:bootstrappers"
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: auto-approve-csrs-for-group
subjects:
- kind: Group
  name: system:bootstrappers
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: system:certificates.k8s.io:certificatesigningrequests:nodeclient
  apiGroup: rbac.authorization.k8s.io
EOF
```

给已启动过的 kubelet 授权自动审批通过 server 证书的 CSR 请求权限以实现自动轮转 client 证书 (之前创建过证书，在证书还未过期前通过证书认证后在 `system:nodes` 用户组里):

```
cat <<EOF | kubectl apply -f -
# Approve renewal CSRs for the group "system:nodes"
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: auto-approve-renewals-for-nodes
subjects:
- kind: Group
  name: system:nodes
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: system:certificates.k8s.io:certificatesigningrequests:selfnodeclient
  apiGroup: rbac.authorization.k8s.io
EOF
```

> 注意上面两个授权都只是针对于 client 证书签发的自动审批权限，server 证书目前不支持自动审批，需要管理员通过 `kubectl certificate approve <csr name>` 来人工审批或者自己写外部 controller 来实现自动审批 (kubelet 访问 apiserver 使用 client 证书，apiserver 主动访问 kubelet 时才会用到 server 证书，通常用于获取 metrics 的场景)

" class="reference-link">创建 Bootstrap Token 与 bootstrap-kubeconfig

bootstrap token 用于 kubelet 自动请求签发证书，以 Secret 形式存储，不需要事先给 apiserver 配置静态 token，这样也易于管理。

创建了 bootstrap token 后我们利用它使用它来创建 bootstrap-kubeconfig 以供后面部署 Worker 节点用 (kubelet 使用 bootstrap-kubeconfig 自动创建证书)，下面是创建方法：

```
APISERVER="https://10.200.16.79:6443"
```

```
# token id should match regex: [a-z0-9]{6}
TOKEN_ID=$(head -c 16 /dev/urandom | od -An -t x | tr -d ' ' | head -c 6)
# token secret should match regex: [a-z0-9]{16}
TOKEN_SECRET=$(head -c 16 /dev/urandom | od -An -t x | tr -d ' ' | head -c 16)

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  # Name MUST be of form "bootstrap-token-<token id>",
  name: bootstrap-token-${TOKEN_ID}
  namespace: kube-system

# Type MUST be 'bootstrap.kubernetes.io/token'
type: bootstrap.kubernetes.io/token
stringData:
  # Human readable description. Optional.
  description: "The default bootstrap token used for signing certificates"

  # Token ID and secret. Required.
  token-id: "${TOKEN_ID}"
  token-secret: "${TOKEN_SECRET}"

  # Expiration. Optional.
  # expiration: 2020-03-10T03:22:11Z

  # Allowed usages.
  usage-bootstrap-authentication: "true"
  usage-bootstrap-signing: "true"

  # Extra groups to authenticate the token as. Must start with "system:bootstrappers:"
  # auth-extra-groups: system:bootstrappers:worker,system:bootstrappers:ingress
EOF
```

```
34.
    kubectl config --kubeconfig=bootstrap-kubeconfig set-cluster bootstrap --
35. server="${APISERVER}" --certificate-authority=ca.pem --embed-certs=true
    kubectl config --kubeconfig=bootstrap-kubeconfig set-credentials kubelet-
36. bootstrap --token=${TOKEN_ID}.${TOKEN_SECRET}
    kubectl config --kubeconfig=bootstrap-kubeconfig set-context bootstrap --
37. user=kubelet-bootstrap --cluster=bootstrap
38. kubectl config --kubeconfig=bootstrap-kubeconfig use-context bootstrap
```

> bootstrap token 的 secret 格式参考: https://kubernetes.io/docs/reference/access-authn-authz/bootstrap-tokens/#bootstrap-token-secret-format

生成文件:

```
1. bootstrap-kubeconfig
```

部署 Worker 节点

Worker 节点主要安装 kubelet 来管理、运行工作负载 (Master 节点也可以部署为特殊 Worker 节点来部署关键服务)

安装依赖

```
1. sudo apt-get update
2. sudo apt-get -y install socat conntrack ipset
```

禁用 Swap

默认情况下，如果开启了 swap，kubelet 会启动失败，k8s 节点推荐禁用 swap。

验证一下是否开启:

```
1. sudo swapon --show
```

如果输出不是空的说明开启了 swap，使用下面的命令禁用 swap:

```
1. sudo swapoff -a
```

为了防止开机自动挂载 swap 分区，可以注释 /etc/fstab 中相应的条目:

```
1. sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
```

关闭 SELinux

关闭 SELinux，否则后续 K8S 挂载目录时可能报错 Permission denied:

```
1. sudo setenforce 0
```

修改配置文件，永久生效:

```
1. sudo sed -i "s/SELINUX=enforcing/SELINUX=disabled/g" /etc/selinux/config
```

准备目录

```
sudo mkdir -p \
  /etc/cni/net.d \
  /opt/cni/bin \
  /var/lib/kubelet \
  /var/lib/kubernetes \
  /var/run/kubernetes
```

下载安装二进制

下载二进制：

```
wget -q --show-progress --https-only --timestamping \
  https://github.com/opencontainers/runc/releases/download/v1.0.0-
rc8/runc.amd64 \
  https://github.com/containerd/containerd/releases/download/v1.3.0/containerd-1.
amd64.tar.gz \
  https://github.com/kubernetes-sigs/cri-tools/releases/download/v1.16.1/crictl-v
amd64.tar.gz \
  https://github.com/containernetworking/plugins/releases/download/v0.8.2/cni-plu
v0.8.2.tgz \
  https://storage.googleapis.com/kubernetes-
release/release/v1.16.1/bin/linux/amd64/kubelet

sudo mv runc.amd64 runc
```

安装二进制：

```
chmod +x crictl kubelet runc
tar -xvf crictl-v1.16.1-linux-amd64.tar.gz
mkdir containerd
tar -xvf containerd-1.3.0.linux-amd64.tar.gz -C containerd
sudo cp crictl kubelet runc /usr/local/bin/
sudo cp containerd/bin/* /bin/
sudo tar -xvf cni-plugins-linux-amd64-v0.8.2.tgz -C /opt/cni/bin/
```

配置

配置 containerd

创建 containerd 启动配置 `config.toml` :

```
sudo mkdir -p /etc/containerd/
cat << EOF | sudo tee /etc/containerd/config.toml
[plugins]
  [plugins.cri.containerd]
    snapshotter = "overlayfs"
    [plugins.cri.containerd.default_runtime]
      runtime_type = "io.containerd.runtime.v1.linux"
      runtime_engine = "/usr/local/bin/runc"
      runtime_root = ""
EOF
```

创建 systemd 配置 `containerd.service` :

```
cat <<EOF | sudo tee /etc/systemd/system/containerd.service
[Unit]
Description=containerd container runtime
Documentation=https://containerd.io
After=network.target

[Service]
ExecStartPre=/sbin/modprobe overlay
ExecStart=/bin/containerd
Restart=always
RestartSec=5
Delegate=yes
KillMode=process
OOMScoreAdjust=-999
LimitNOFILE=1048576
LimitNPROC=infinity
LimitCORE=infinity

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

配置 kubelet

放入 这里 创建好的 CA 证书与 这里 创建好的 bootstrap-kubeconfig:

```
sudo cp ca.pem /var/lib/kubernetes/
sudo cp bootstrap-kubeconfig /var/lib/kubelet/
```

事先确定好集群 DNS 的 CLUSTER IP 地址，通常可以用 service 网段的最后一个可用 IP 地址：

```
DNS=10.32.0.255
```

创建 kubelet 启动配置 `config.yaml`：

```
cat <<EOF | sudo tee /var/lib/kubelet/config.yaml
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: false
  webhook:
    enabled: true
  x509:
    clientCAFile: "/var/lib/kubernetes/ca.pem"
authorization:
  mode: Webhook
clusterDomain: "cluster.local"
clusterDNS:
  - "${DNS}"
resolvConf: "/run/systemd/resolve/resolv.conf"
runtimeRequestTimeout: "15m"
rotateCertificates: true
serverTLSBootstrap: true
EOF
```

用 `NODE` 变量表示节点名称，kube-apiserver 所在节点需要能够通过这个名称访问到节点，这里推荐直接使用节点内网 IP，不需要配 hosts 就能访问：

```
NODE="10.200.16.79"
```

创建 systemd 配置 `kubelet.service`：

```
cat <<EOF | sudo tee /etc/systemd/system/kubelet.service
[Unit]
Description=Kubernetes Kubelet
```

```
Documentation=https://github.com/kubernetes/kubernetes
After=containerd.service
Requires=containerd.service

[Service]
ExecStart=/usr/local/bin/kubelet \\
  --config=/var/lib/kubelet/config.yaml \\
  --container-runtime=remote \\
  --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
  --image-pull-progress-deadline=2m \\
  --bootstrap-kubeconfig=/var/lib/kubelet/bootstrap-kubeconfig \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --register-node=true \\
  --hostname-override=${NODE} \\
  --v=2
Restart=on-failure
RestartSec=5

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

启动

```
sudo systemctl daemon-reload
sudo systemctl enable containerd kubelet
sudo systemctl start containerd kubelet
```

验证

配置好 kubectl，执行下 kubectl:

```
$ kubectl get node
NAME           STATUS     ROLES    AGE   VERSION
10.200.16.79   NotReady   <none>   11m   v1.16.1
```

没有装网络插件，节点状态会是 `NotReady`，带 `node.kubernetes.io/not-ready:NoSchedule` 这个污点，默认是无法调度普通 Pod，这个是正常的。后面会装网络插件，通常以 Daemonset 部署，使用 hostNetwork，并且容忍这个污点。

签发 kubelet server 证书

由于之前做过 RBAC 授权 kubelet 创建 CSR 与自动签发和更新证书，kubelet 启动时可以发起 client 与 server 证书的 CSR 请求，并自动审批通过 client 证书的 CSR 请求，kube-controller-manager 在自动执行证书签发，最后 kubelet 可以获取到 client 证书并加入集群，我们可以在 `/var/lib/kubelet/pki` 下面看到签发出来的 client 证书：

```
ls -l /var/lib/kubelet/pki
total 4
-rw------- 1 root root 1277 Oct 10 20:46 kubelet-client-2019-10-10-20-46-23.pem
lrwxrwxrwx 1 root root   59 Oct 10 20:46 kubelet-client-current.pem ->
/var/lib/kubelet/pki/kubelet-client-2019-10-10-20-46-23.pem
```

> kubeconfig 中引用这里的 `kubelet-client-current.pem` 这个证书，是一个指向证书 bundle 的软连接，包含证书公钥与私钥

但 server 证书默认无法自动审批，需要管理员人工审批，下面是审批方法，首先看下未审批的 CSR：

```
$ kubectl get csr
NAME        AGE     REQUESTOR                 CONDITION
csr-6gkn6   2m4s    system:bootstrap:360483   Approved,Issued
csr-vf285   103s    system:node:10.200.17.6   Pending
```

> 可以看到 `system:bootstrap` 开头的用户的 CSR 请求已经自动 approve 并签发证书了，这就是因为 kubelet 使用 bootstrap token 认证后在 `system:bootstrappers` 用户组，而我们创建了对应 RBAC 为此用户组授权自动 approve CSR 的权限。下面 `system:node` 开头的用户的 CSR 请求状态是 Pending，需要管理员来 approve。

```
$ kubectl certificate approve csr-vf285
certificatesigningrequest.certificates.k8s.io/csr-vf285 approved

$ ls -l /var/lib/kubelet/pki
total 8
-rw------- 1 root root 1277 Oct 10 20:46 kubelet-client-2019-10-10-20-46-23.pem
lrwxrwxrwx 1 root root   59 Oct 10 20:46 kubelet-client-current.pem ->
/var/lib/kubelet/pki/kubelet-client-2019-10-10-20-46-23.pem
-rw------- 1 root root 1301 Oct 10 21:09 kubelet-server-2019-10-10-21-09-15.pem
lrwxrwxrwx 1 root root   59 Oct 10 21:09 kubelet-server-current.pem ->
/var/lib/kubelet/pki/kubelet-server-2019-10-10-21-09-15.pem
```

> 和 client 证书一样，`kubelet-server-current.pem` 也是一个指向证书 bundle 的软连接，包含证书公钥与私钥，用与 kubelet 监听 10250 端口

部署关键组件

部署 kube-proxy

kube-proxy 会请求 apiserver 获取 Service 及其 Endpoint，将 Service 的 ClUSTER IP 与对应 Endpoint 的 Pod IP 映射关系转换成 iptables 或 ipvs 规则写到节点上，实现 Service 转发。

部署方法参考 以 Daemonset 方式部署 kube-proxy

部署网络插件

参考 部署 Flannel

部署集群 DNS

集群 DNS 是 Kubernetes 的核心功能之一，被许多服务所依赖，用于解析集群内 Pod 的 DNS 请求，包括:

- 解析 service 名称成对应的 CLUSTER IP
- 解析 headless service 名称成对应 Pod IP (选取一个 endpoint 的 Pod IP 返回)
- 解析外部域名(代理 Pod 请求上游 DNS)

可以通过部署 kube-dns 或 CoreDNS 作为集群的必备扩展来提供命名服务，推荐使用 CoreDNS，效率更高，资源占用率更小，部署方法参考 部署 CoreDNS

部署附加组件

部署 CoreDNS

下载部署脚本

```
$ git clone https://github.com/coredns/deployment.git
$ cd deployment/kubernetes
$ ls
CoreDNS-k8s_version.md  FAQs.md  README.md  Scaling_CoreDNS.md
Upgrading_CoreDNS.md  coredns.yaml.sed  corefile-tool  deploy.sh  migration
rollback.sh
```

部署脚本用法

查看 help:

```
$ ./deploy.sh -h
usage: ./deploy.sh [ -r REVERSE-CIDR ] [ -i DNS-IP ] [ -d CLUSTER-DOMAIN ] [ -t
YAML-TEMPLATE ]

        -r : Define a reverse zone for the given CIDR. You may specifcy this option
more
             than once to add multiple reverse zones. If no reverse CIDRs are
defined,
             then the default is to handle all reverse zones (i.e. in-addr.arpa and
ip6.arpa)
        -i : Specify the cluster DNS IP address. If not specificed, the IP address
of
             the existing "kube-dns" service is used, if present.
        -s : Skips the translation of kube-dns configmap to the corresponding
CoreDNS Corefile configuration.
```

部署

总体流程是我们使用 `deploy.sh` 生成 yaml 并保存成 `coredns.yaml` 文件并执行 `kubectl apply -f coredns.yaml` 进行部署 ，如果要卸载，执行 `kubectl delete -f coredns.yaml` 。

`deploy.sh` 脚本依赖 `jq` 命令，所以先确保 `jq` 已安装：

```
apt install -y jq
```

全新部署

如果集群中没有 kube-dns 或低版本 coredns，我们直接用 `-i` 参数指定集群 DNS 的 CLUSTER IP，这个 IP 是安装集群时就确定好的，示例：

```
./deploy.sh -i 10.32.0.255 > coredns.yaml
kubectl apply -f coredns.yaml
```

以 Daemonset 方式部署 kube-proxy

kube-proxy 可以用二进制部署，也可以用 kubelet 的静态 Pod 部署，但最简单使用 DaemonSet 部署。直接使用 ServiceAccount 的 token 认证，不需要签发证书，也就不用担心证书过期问题。

先在终端设置下面的变量：

```
APISERVER="https://10.200.16.79:6443"
CLUSTER_CIDR="10.10.0.0/16"
```

- `APISERVER` 替换为 apiserver 对外暴露的访问地址。有同学想问为什么不直接用集群内的访问地址(`kubernetes.default` 或对应的 CLUSTER IP)，这是一个鸡生蛋还是蛋生鸡的问题，CLSUTER IP 本身就是由 kube-proxy 来生成 iptables 或 ipvs 规则转发 Service 对应 Endpoint 的 Pod IP，kube-proxy 刚启动还没有生成这些转发规则，生成规则的前提是 kube-proxy 需要访问 apiserver 获取 Service 与 Endpoint，而由于还没有转发规则，kube-proxy 访问 apiserver 的 CLUSTER IP 的请求无法被转发到 apiserver。
- `CLUSTER_CIDR` 替换为集群 Pod IP 的 CIDR 范围，这个在部署 kube-controller-manager 时也设置过

为 kube-proxy 创建 RBAC 权限和配置文件：

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kube-proxy
  namespace: kube-system

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:kube-proxy
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:node-proxier
```

```
subjects:
- kind: ServiceAccount
  name: kube-proxy
  namespace: kube-system

---

kind: ConfigMap
apiVersion: v1
metadata:
  name: kube-proxy
  namespace: kube-system
  labels:
    app: kube-proxy
data:
  kubeconfig.conf: |-
    apiVersion: v1
    kind: Config
    clusters:
    - cluster:
        certificate-authority:
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        server: ${APISERVER}
      name: default
    contexts:
    - context:
        cluster: default
        namespace: default
        user: default
      name: default
    current-context: default
    users:
    - name: default
      user:
        tokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
  config.conf: |-
    apiVersion: kubeproxy.config.k8s.io/v1alpha1
    kind: KubeProxyConfiguration
    bindAddress: 0.0.0.0
    clientConnection:
      acceptContentTypes: ""
      burst: 10
```

```
      contentType: application/vnd.kubernetes.protobuf
      kubeconfig: /var/lib/kube-proxy/kubeconfig.conf
      qps: 5
    # 集群中 Pod IP 的 CIDR 范围
    clusterCIDR: ${CLUSTER_CIDR}
    configSyncPeriod: 15m0s
    conntrack:
      # 每个核心最大能跟踪的NAT连接数，默认32768
      maxPerCore: 32768
      min: 131072
      tcpCloseWaitTimeout: 1h0m0s
      tcpEstablishedTimeout: 24h0m0s
    enableProfiling: false
    healthzBindAddress: 0.0.0.0:10256
    iptables:
      # SNAT 所有 Service 的 CLUSTER IP
      masqueradeAll: false
      masqueradeBit: 14
      minSyncPeriod: 0s
      syncPeriod: 30s
    ipvs:
      minSyncPeriod: 0s
      # ipvs 调度类型，默认是 rr，支持的所有类型：
      # rr: round-robin
      # lc: least connection
      # dh: destination hashing
      # sh: source hashing
      # sed: shortest expected delay
      # nq: never queue
      scheduler: rr
      syncPeriod: 30s
    metricsBindAddress: 0.0.0.0:10249
    # 使用 ipvs 模式转发 service
    mode: ipvs
    # 设置 kube-proxy 进程的 oom-score-adj 值，范围 [-1000,1000]
    # 值越低越不容易被杀死，这里设置为 —999 防止发生系统OOM时将 kube-proxy 杀死
    oomScoreAdj: -999
EOF
```

在终端设置下面的变量：

```
ARCH="amd64"
```

```
VERSION="v1.16.1"
```

- `VERSION` 是 K8S 版本
- `ARCH` 是节点的 cpu 架构，大多数用的 `amd64`，即 x86_64。其它常见的还有：`arm64`，`arm`，`ppc64le`，`s390x`，如果你的集群有不同 cpu 架构的节点，可以分别指定 `ARCH` 部署多个 daemonset（每个节点不会有多个 kube-proxy，nodeSelector 会根据 cpu 架构来选中节点）

使用 hostNetwork 以 Daemonset 方式部署 kube-proxy 到每个节点：

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    k8s-app: kube-proxy-ds-${ARCH}
  name: kube-proxy-ds-${ARCH}
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: kube-proxy-ds-${ARCH}
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        k8s-app: kube-proxy-ds-${ARCH}
    spec:
      priorityClassName: system-node-critical
      containers:
      - name: kube-proxy
        image: k8s.gcr.io/kube-proxy-${ARCH}:${VERSION}
        imagePullPolicy: IfNotPresent
        command:
        - /usr/local/bin/kube-proxy
        - --config=/var/lib/kube-proxy/config.conf
        - --hostname-override=\$(NODE_NAME)
        securityContext:
          privileged: true
        volumeMounts:
        - mountPath: /var/lib/kube-proxy
```

```
            name: kube-proxy
          - mountPath: /run/xtables.lock
            name: xtables-lock
            readOnly: false
          - mountPath: /lib/modules
            name: lib-modules
            readOnly: true
          env:
            - name: NODE_NAME
              valueFrom:
                fieldRef:
                  fieldPath: spec.nodeName
        hostNetwork: true
        serviceAccountName: kube-proxy
        volumes:
        - name: kube-proxy
          configMap:
            name: kube-proxy
        - name: xtables-lock
          hostPath:
            path: /run/xtables.lock
            type: FileOrCreate
        - name: lib-modules
          hostPath:
            path: /lib/modules
        tolerations:
        - key: CriticalAddonsOnly
          operator: Exists
        - operator: Exists
        nodeSelector:
          beta.kubernetes.io/arch: ${ARCH}
EOF
```

常见应用部署

- ElasticSearch 与 Kibana

Elasticsearch 与 Kibana

- 使用 elastic-oparator 部署

使用 elastic-oparator 部署 Elasticsearch 和 Kibana

参考官方文档：

- https://www.elastic.co/cn/elasticsearch-kubernetes
- https://www.elastic.co/cn/blog/introducing-elastic-cloud-on-kubernetes-the-elasticsearch-operator-and-beyond

安装 elastic-operator

一键安装：

```
kubectl apply -f https://download.elastic.co/downloads/eck/0.9.0/all-in-one.yaml
```

部署 Elasticsearch

准备一个命名空间用来部署 elasticsearch，这里我们使用 `monitoring` 命名空间：

```
kubectl create ns monitoring
```

创建 CRD 资源部署 Elasticsearch，最简单的部署：

```
cat <<EOF | kubectl apply -f -
apiVersion: elasticsearch.k8s.elastic.co/v1alpha1
kind: Elasticsearch
metadata:
  name: es
  namespace: monitoring
spec:
  version: 7.2.0
  nodes:
  - nodeCount: 1
    config:
      node.master: true
      node.data: true
      node.ingest: true
EOF
```

多节点部署高可用 elasticsearch 集群:

```
cat <<EOF | kubectl apply -f -
apiVersion: elasticsearch.k8s.elastic.co/v1alpha1
kind: Elasticsearch
metadata:
  name: es
  namespace: monitoring
spec:
  version: 7.2.0
  nodes:
  - nodeCount: 1
    config:
      node.master: true
      node.data: true
      node.ingest: true
    volumeClaimTemplates:
    - metadata:
        name: elasticsearch-data
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 100Gi
    podTemplate:
      spec:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                - key: elasticsearch.k8s.elastic.co/cluster-name
                  operator: In
                  values:
                  - es
              topologyKey: "kubernetes.io/hostname"
  - nodeCount: 2
    config:
      node.master: false
      node.data: true
```

```
      node.ingest: true
    volumeClaimTemplates:
    - metadata:
        name: elasticsearch-data
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 80Gi
    podTemplate:
      spec:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                - key: elasticsearch.k8s.elastic.co/cluster-name
                  operator: In
                  values:
                  - es
              topologyKey: kubernetes.io/hostname
EOF
```

- `metadata.name` 是 elasticsearch 集群的名称
- `nodeCount` 大于 1（多副本）并且加了 pod 反亲和性（避免调度到同一个节点）可避免单点故障，保证高可用
- `node.master` 为 true 表示是 master 节点
- 可根据需求调整 `nodeCount`（副本数量）和 `storage`（数据磁盘容量）
- 反亲和性的 `labelSelector.matchExpressions.values` 中写 elasticsearch 集群名称，更改集群名称时记得这里要也改下
- 强制开启 ssl 不允许关闭：https://github.com/elastic/cloud-on-k8s/blob/576f07faaff4393f9fb247e58b87517f99b08ebd///pkg/controller/elasticsearch/settings/fields.go#L51

查看部署状态：

```
$ kubectl -n monitoring get es
NAME   HEALTH   NODES   VERSION   PHASE         AGE
es     green    3       7.2.0     Operational   3m
$
$ kubectl -n monitoring get pod -o wide
```

```
NAME                          READY   STATUS    RESTARTS   AGE    IP
NODE        NOMINATED NODE
es-es-c7pwnt5kz8              1/1     Running   0          4m3s   172.16.4.6
10.0.0.24   <none>
es-es-qpk7kkpdxh              1/1     Running   0          4m3s   172.16.5.6
10.0.0.48   <none>
es-es-vl56nv78hd              1/1     Running   0          4m3s   172.16.3.9
10.0.0.32   <none>
$
$ kubectl -n monitoring get svc
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
es-es-http       ClusterIP   172.16.15.74   <none>        9200/TCP   7m3s
```

elasticsearch 的默认用户名是 elastic，获取密码：

```
$ kubectl -n monitoring get secret es-es-elastic-user -o
jsonpath='{.data.elastic}' | base64 -d
rhd6jdw9brbj69d49k46px9j
```

后续连接 elasticsearch 时就用这对用户名密码：

- username: elastic
- password: rhd6jdw9brbj69d49k46px9j

部署 Kibana

还可以再部署一个 Kibana 集群作为 UI：

```
cat <<EOF | kubectl apply -f -
apiVersion: kibana.k8s.elastic.co/v1alpha1
kind: Kibana
metadata:
  name: kibana
  namespace: monitoring
spec:
  version: 7.2.0
  nodeCount: 2
  podTemplate:
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
```

```
          - labelSelector:
              matchExpressions:
              - key: kibana.k8s.elastic.co/name
                operator: In
                values:
                - kibana
            topologyKey: kubernetes.io/hostname
  elasticsearchRef:
    name: es
    namespace: monitoring
EOF
```

- `nodeCount` 大于 1（多副本）并且加了 pod 反亲和性（避免调度到同一个节点）可避免单点故障，保证高可用
- 反亲和性的 `labelSelector.matchExpressions.values` 中写 kibana 集群名称，更改集群名称时记得这里要也改下
- `elasticsearchRef` 引用已经部署的 elasticsearch 集群，`name` 和 `namespace` 分别填部署的 elasticsearch 集群名称和命名空间

查看部署状态：

```
$ kubectl -n monitoring get kb
NAME     HEALTH   NODES   VERSION   AGE
kibana   green    2       7.2.0     3m
$
$ kubectl -n monitoring get pod -o wide
NAME                          READY   STATUS    RESTARTS   AGE    IP
NODE        NOMINATED NODE
kibana-kb-58dc8994bf-224bl    1/1     Running   0          93s    172.16.0.92
10.0.0.3    <none>
kibana-kb-58dc8994bf-nchqt    1/1     Running   0          93s    172.16.3.10
10.0.0.32   <none>
$
$ kubectl -n monitoring get svc
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
kibana-kb-http   ClusterIP   172.16.8.71    <none>        5601/TCP   4m35s
```

还需要为 Kibana 暴露一个外部地址好让我们能从从浏览器访问，可以创建 Service 或 Ingress 来实现。

> 默认也会为 Kibana 创建 ClusterIP 类型的 Service，可以在 Kibana 的 CRD spec 里加 service 来自定义 service type 为 LoadBalancer 实现对外暴露，但我不建议这么做，因为一旦删除 CRD 对

> 象，service 也会被删除，在云上通常意味着对应的负载均衡器也被自动删除，IP 地址就会被回收，下次再创建的时候 IP 地址就变了，所以推荐对外暴露方式使用单独的 Service 或 Ingress 来维护

创建 Service

先看下当前 kibana 的 service:

```
$ kubectl -n monitoring get svc -o yaml kibana-kb-http
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2019-09-17T09:20:04Z
  labels:
    common.k8s.elastic.co/type: kibana
    kibana.k8s.elastic.co/name: kibana
  name: kibana-kb-http
  namespace: monitoring
  ownerReferences:
  - apiVersion: kibana.k8s.elastic.co/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: Kibana
    name: kibana
    uid: 54fd304b-d92c-11e9-89f7-be8690a7fdcf
  resourceVersion: "5668802758"
  selfLink: /api/v1/namespaces/monitoring/services/kibana-kb-http
  uid: 55a1198f-d92c-11e9-89f7-be8690a7fdcf
spec:
  clusterIP: 172.16.8.71
  ports:
  - port: 5601
    protocol: TCP
    targetPort: 5601
  selector:
    common.k8s.elastic.co/type: kibana
    kibana.k8s.elastic.co/name: kibana
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

仅保留端口和 `selector` 的配置，如果集群支持 `LoadBanlancer` 类型的 service，可以修

改 service 的 type 为 `LoadBalancer`：

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: kibana
  namespace: monitoring
spec:
  ports:
  - port: 443
    protocol: TCP
    targetPort: 5601
  selector:
    common.k8s.elastic.co/type: kibana
    kibana.k8s.elastic.co/name: kibana
  type: LoadBalancer
EOF
```

拿到负载均衡器的 IP 地址：

```
$ kubectl -n monitoring get svc
NAME             TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)
AGE
kibana           LoadBalancer   172.16.10.71   150.109.27.60   443:32749/TCP
47s
kibana-kb-http   ClusterIP      172.16.15.39   <none>          5601/TCP
118s
```

在浏览器访问：https://150.109.27.60:443

输入之前部署 elasticsearch 的用户名密码进行登录

- 网络方案
- 运行时方案
- Ingress 方案
- Metrics 方案

网络方案

- Flannel

Flannel

部署 Flannel

记集群网段为 CLUSTER_CIDR:

```
CLUSTER_CIDR=10.10.0.0/16
```

创建 flannel 资源文件:

```
cat <<EOF | sudo tee kube-flannel.yml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: psp.flannel.unprivileged
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: docker/default
    seccomp.security.alpha.kubernetes.io/defaultProfileName: docker/default
    apparmor.security.beta.kubernetes.io/allowedProfileNames: runtime/default
    apparmor.security.beta.kubernetes.io/defaultProfileName: runtime/default
spec:
  privileged: false
  volumes:
    - configMap
    - secret
    - emptyDir
    - hostPath
  allowedHostPaths:
    - pathPrefix: "/etc/cni/net.d"
    - pathPrefix: "/etc/kube-flannel"
    - pathPrefix: "/run/flannel"
  readOnlyRootFilesystem: false
  # Users and groups
  runAsUser:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  # Privilege Escalation
  allowPrivilegeEscalation: false
  defaultAllowPrivilegeEscalation: false
```

```
  # Capabilities
  allowedCapabilities: ['NET_ADMIN']
  defaultAddCapabilities: []
  requiredDropCapabilities: []
  # Host namespaces
  hostPID: false
  hostIPC: false
  hostNetwork: true
  hostPorts:
  - min: 0
    max: 65535
  # SELinux
  seLinux:
    # SELinux is unsed in CaaSP
    rule: 'RunAsAny'
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: flannel
rules:
  - apiGroups: ['extensions']
    resources: ['podsecuritypolicies']
    verbs: ['use']
    resourceNames: ['psp.flannel.unprivileged']
  - apiGroups:
      - ""
    resources:
      - pods
    verbs:
      - get
  - apiGroups:
      - ""
    resources:
      - nodes
    verbs:
      - list
      - watch
  - apiGroups:
      - ""
    resources:
      - nodes/status
```

```
    verbs:
      - patch
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: flannel
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: flannel
subjects:
- kind: ServiceAccount
  name: flannel
  namespace: kube-system
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: flannel
  namespace: kube-system
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: kube-flannel-cfg
  namespace: kube-system
  labels:
    tier: node
    app: flannel
data:
  cni-conf.json: |
    {
      "cniVersion": "0.2.0",
      "name": "cbr0",
      "plugins": [
        {
          "type": "flannel",
          "delegate": {
            "hairpinMode": true,
            "isDefaultGateway": true
          }
```

```
        },
        {
          "type": "portmap",
          "capabilities": {
            "portMappings": true
          }
        }
      ]
    }
  net-conf.json: |
    {
      "Network": "${CLUSTER_CIDR}",
      "Backend": {
        "Type": "vxlan"
      }
    }
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: kube-flannel-ds-amd64
  namespace: kube-system
  labels:
    tier: node
    app: flannel
spec:
  selector:
    matchLabels:
      app: flannel
  template:
    metadata:
      labels:
        tier: node
        app: flannel
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: beta.kubernetes.io/os
                    operator: In
```

```
                    values:
                      - linux
                  - key: beta.kubernetes.io/arch
                    operator: In
                    values:
                      - amd64
      hostNetwork: true
      tolerations:
      - operator: Exists
        effect: NoSchedule
      serviceAccountName: flannel
      initContainers:
      - name: install-cni
        image: quay.io/coreos/flannel:v0.11.0-amd64
        command:
        - cp
        args:
        - -f
        - /etc/kube-flannel/cni-conf.json
        - /etc/cni/net.d/10-flannel.conflist
        volumeMounts:
        - name: cni
          mountPath: /etc/cni/net.d
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      containers:
      - name: kube-flannel
        image: quay.io/coreos/flannel:v0.11.0-amd64
        command:
        - /opt/bin/flanneld
        args:
        - --ip-masq
        - --kube-subnet-mgr
        resources:
          requests:
            cpu: "100m"
            memory: "50Mi"
          limits:
            cpu: "100m"
            memory: "50Mi"
        securityContext:
          privileged: false
```

```
          capabilities:
             add: ["NET_ADMIN"]
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        volumeMounts:
        - name: run
          mountPath: /run/flannel
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      volumes:
        - name: run
          hostPath:
            path: /run/flannel
        - name: cni
          hostPath:
            path: /etc/cni/net.d
        - name: flannel-cfg
          configMap:
            name: kube-flannel-cfg
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: kube-flannel-ds-arm64
  namespace: kube-system
  labels:
    tier: node
    app: flannel
spec:
  selector:
    matchLabels:
      app: flannel
  template:
    metadata:
      labels:
```

```
        tier: node
        app: flannel
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: beta.kubernetes.io/os
                    operator: In
                    values:
                      - linux
                  - key: beta.kubernetes.io/arch
                    operator: In
                    values:
                      - arm64
      hostNetwork: true
      tolerations:
      - operator: Exists
        effect: NoSchedule
      serviceAccountName: flannel
      initContainers:
      - name: install-cni
        image: quay.io/coreos/flannel:v0.11.0-arm64
        command:
        - cp
        args:
        - -f
        - /etc/kube-flannel/cni-conf.json
        - /etc/cni/net.d/10-flannel.conflist
        volumeMounts:
        - name: cni
          mountPath: /etc/cni/net.d
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      containers:
      - name: kube-flannel
        image: quay.io/coreos/flannel:v0.11.0-arm64
        command:
        - /opt/bin/flanneld
        args:
        - --ip-masq
```

```
        - --kube-subnet-mgr
        resources:
          requests:
            cpu: "100m"
            memory: "50Mi"
          limits:
            cpu: "100m"
            memory: "50Mi"
        securityContext:
          privileged: false
          capabilities:
             add: ["NET_ADMIN"]
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        volumeMounts:
        - name: run
          mountPath: /run/flannel
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      volumes:
        - name: run
          hostPath:
            path: /run/flannel
        - name: cni
          hostPath:
            path: /etc/cni/net.d
        - name: flannel-cfg
          configMap:
            name: kube-flannel-cfg
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: kube-flannel-ds-arm
  namespace: kube-system
```

```
    labels:
      tier: node
      app: flannel
  spec:
    selector:
      matchLabels:
        app: flannel
    template:
      metadata:
        labels:
          tier: node
          app: flannel
      spec:
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: beta.kubernetes.io/os
                      operator: In
                      values:
                        - linux
                    - key: beta.kubernetes.io/arch
                      operator: In
                      values:
                        - arm
        hostNetwork: true
        tolerations:
        - operator: Exists
          effect: NoSchedule
        serviceAccountName: flannel
        initContainers:
        - name: install-cni
          image: quay.io/coreos/flannel:v0.11.0-arm
          command:
          - cp
          args:
          - -f
          - /etc/kube-flannel/cni-conf.json
          - /etc/cni/net.d/10-flannel.conflist
          volumeMounts:
          - name: cni
```

```
          mountPath: /etc/cni/net.d
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      containers:
      - name: kube-flannel
        image: quay.io/coreos/flannel:v0.11.0-arm
        command:
        - /opt/bin/flanneld
        args:
        - --ip-masq
        - --kube-subnet-mgr
        resources:
          requests:
            cpu: "100m"
            memory: "50Mi"
          limits:
            cpu: "100m"
            memory: "50Mi"
        securityContext:
          privileged: false
          capabilities:
             add: ["NET_ADMIN"]
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        volumeMounts:
        - name: run
          mountPath: /run/flannel
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      volumes:
        - name: run
          hostPath:
            path: /run/flannel
        - name: cni
          hostPath:
```

```
            path: /etc/cni/net.d
        - name: flannel-cfg
          configMap:
            name: kube-flannel-cfg
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: kube-flannel-ds-ppc64le
  namespace: kube-system
  labels:
    tier: node
    app: flannel
spec:
  selector:
    matchLabels:
      app: flannel
  template:
    metadata:
      labels:
        tier: node
        app: flannel
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: beta.kubernetes.io/os
                    operator: In
                    values:
                      - linux
                  - key: beta.kubernetes.io/arch
                    operator: In
                    values:
                      - ppc64le
      hostNetwork: true
      tolerations:
      - operator: Exists
        effect: NoSchedule
      serviceAccountName: flannel
      initContainers:
```

```
      - name: install-cni
        image: quay.io/coreos/flannel:v0.11.0-ppc64le
        command:
        - cp
        args:
        - -f
        - /etc/kube-flannel/cni-conf.json
        - /etc/cni/net.d/10-flannel.conflist
        volumeMounts:
        - name: cni
          mountPath: /etc/cni/net.d
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      containers:
      - name: kube-flannel
        image: quay.io/coreos/flannel:v0.11.0-ppc64le
        command:
        - /opt/bin/flanneld
        args:
        - --ip-masq
        - --kube-subnet-mgr
        resources:
          requests:
            cpu: "100m"
            memory: "50Mi"
          limits:
            cpu: "100m"
            memory: "50Mi"
        securityContext:
          privileged: false
          capabilities:
             add: ["NET_ADMIN"]
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        volumeMounts:
```

```
495.           - name: run
496.             mountPath: /run/flannel
497.           - name: flannel-cfg
498.             mountPath: /etc/kube-flannel/
499.         volumes:
500.           - name: run
501.             hostPath:
502.               path: /run/flannel
503.           - name: cni
504.             hostPath:
505.               path: /etc/cni/net.d
506.           - name: flannel-cfg
507.             configMap:
508.               name: kube-flannel-cfg
509.   ---
510.   apiVersion: apps/v1
511.   kind: DaemonSet
512.   metadata:
513.     name: kube-flannel-ds-s390x
514.     namespace: kube-system
515.     labels:
516.       tier: node
517.       app: flannel
518.   spec:
519.     selector:
520.       matchLabels:
521.         app: flannel
522.     template:
523.       metadata:
524.         labels:
525.           tier: node
526.           app: flannel
527.       spec:
528.         affinity:
529.           nodeAffinity:
530.             requiredDuringSchedulingIgnoredDuringExecution:
531.               nodeSelectorTerms:
532.                 - matchExpressions:
533.                     - key: beta.kubernetes.io/os
534.                       operator: In
535.                       values:
536.                         - linux
```

```
                  - key: beta.kubernetes.io/arch
                    operator: In
                    values:
                      - s390x
      hostNetwork: true
      tolerations:
      - operator: Exists
        effect: NoSchedule
      serviceAccountName: flannel
      initContainers:
      - name: install-cni
        image: quay.io/coreos/flannel:v0.11.0-s390x
        command:
        - cp
        args:
        - -f
        - /etc/kube-flannel/cni-conf.json
        - /etc/cni/net.d/10-flannel.conflist
        volumeMounts:
        - name: cni
          mountPath: /etc/cni/net.d
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      containers:
      - name: kube-flannel
        image: quay.io/coreos/flannel:v0.11.0-s390x
        command:
        - /opt/bin/flanneld
        args:
        - --ip-masq
        - --kube-subnet-mgr
        resources:
          requests:
            cpu: "100m"
            memory: "50Mi"
          limits:
            cpu: "100m"
            memory: "50Mi"
        securityContext:
          privileged: false
          capabilities:
             add: ["NET_ADMIN"]
```

```
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        volumeMounts:
        - name: run
          mountPath: /run/flannel
        - name: flannel-cfg
          mountPath: /etc/kube-flannel/
      volumes:
        - name: run
          hostPath:
            path: /run/flannel
        - name: cni
          hostPath:
            path: /etc/cni/net.d
        - name: flannel-cfg
          configMap:
            name: kube-flannel-cfg
EOF
```

部署：

```
kubectl apply -f kube-flannel.yml
```

> 以上资源文件参考 flannel 官方：
> https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml (仅提取了 CLUSTER_CIDR 变量)

运行时方案

- Containerd

Containerd

安装 containerd

" class="reference-link">二进制部署

下载二进制：

```
wget -q --show-progress --https-only --timestamping \
  https://github.com/opencontainers/runc/releases/download/v1.0.0-
rc8/runc.amd64 \
  https://github.com/containerd/containerd/releases/download/v1.3.0/containerd-1.
amd64.tar.gz \
  https://github.com/kubernetes-sigs/cri-tools/releases/download/v1.16.1/crictl-v
amd64.tar.gz

sudo mv runc.amd64 runc
```

安装二进制：

```
tar -xvf crictl-v1.16.1-linux-amd64.tar.gz
chmod +x crictl runc
sudo cp crictl runc /usr/local/bin/

mkdir containerd
tar -xvf containerd-1.3.0.linux-amd64.tar.gz -C containerd
sudo cp containerd/bin/* /bin/
```

创建 containerd 启动配置 `config.toml` ：

```
sudo mkdir -p /etc/containerd/
cat << EOF | sudo tee /etc/containerd/config.toml
[plugins]
  [plugins.cri.containerd]
    snapshotter = "overlayfs"
    [plugins.cri.containerd.default_runtime]
      runtime_type = "io.containerd.runtime.v1.linux"
      runtime_engine = "/usr/local/bin/runc"
      runtime_root = ""
EOF
```

创建 systemd 配置 `containerd.service` :

```
cat <<EOF | sudo tee /etc/systemd/system/containerd.service
[Unit]
Description=containerd container runtime
Documentation=https://containerd.io
After=network.target

[Service]
ExecStartPre=/sbin/modprobe overlay
ExecStart=/bin/containerd
Restart=always
RestartSec=5
Delegate=yes
KillMode=process
OOMScoreAdjust=-999
LimitNOFILE=1048576
LimitNPROC=infinity
LimitCORE=infinity

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

启动:

```
sudo systemctl daemon-reload
sudo systemctl enable containerd
sudo systemctl start containerd
```

配置 crictl (方便后面使用 crictl 管理与调试 containerd 的容器与镜像):

```
crictl config runtime-endpoint unix:///var/run/containerd/containerd.sock
```

Ingress 方案

- Nginx Ingress
- Traefik Ingress

Nginx Ingress

安装 nginx ingress controller

最佳安装方案

如何暴露 ingress 访问入口？最佳方案是使用 LoadBalancer 类型的 Service 来暴露，即创建外部负载均衡器来暴露流量，后续访问 ingress 的流量都走这个负载均衡器的地址，ingress 规则里面配的域名也要配置解析到这个负载均衡器的 IP 地址。

这种方式需要集群支持 LoadBalancer 类型的 Service，如果是云厂商提供的 k8s 服务，或者在云上自建集群并使用了云厂商提供的 cloud provider，也都是支持的，创建 LoadBalancer 类型的 Service 的时候会自动调云厂商的接口创建云厂商提供的负载均衡器产品(通常公网类型的负载均衡器是付费的)；如果你的集群不是前面说的情况，是自建集群并且有自己的负载均衡器方案，并部署了相关插件来适配，比如 MetalLB 和 Porter，这样也是可以支持 LoadBalancer 类型的 Service 的。

使用 helm 安装

```
1.
2.  helm install stable/nginx-ingress \
3.    --name nginx \
4.    --namespace kube-system \
5.    --set controller.ingressClass=nginx \
6.    --set controller.publishService.enabled=true \
```

- `controller.ingressClass`：创建的 ingress 中包含 `kubernetes.io/ingress.class` 这个 annotation 并且值与这里配置的一致，这个 nginx ingress controller 才会处理(生成转发规则)
- `controller.publishService.enabled`：这个置为 true 主要是为了让 ingress 的外部地址正确显示（显示为负载均衡器的地址），因为如果不配置这个，默认情况下会将 ingress controller 所有实例的节点 ip 写到 ingress 的 address 里

安装完成后如何获取负载均衡器的 IP 地址？查看 nginx ingress controller 的 service 的 `EXTERNAL-IP` 就可以：

```
1.  $ kubectl -n kube-system get service nginx-nginx-ingress-controller
    NAME                                TYPE           CLUSTER-IP       EXTERNAL-IP
2.  PORT(S)                     AGE
    nginx-nginx-ingress-controller   LoadBalancer   172.16.255.194   119.28.123.174
3.  80:32348/TCP,443:32704/TCP   10m
```

如果需要新的流量入口，可以按照同样的方法用 helm 安装新的 release，注意要设置不同的 `controller.ingressClass`，将希望用新流量入口暴露的 ingress 的 `kubernetes.io/ingress.class` annotation 设置成这里的值就可以。

如果转发性能跟不上，可以增加 controller 的副本，设置 `controller.replicaCount` 的值，或者启用 HPA 自动伸缩，将 `controller.autoscaling.enabled` 置为 true，更多细节控制请参考官方文档。

配置优化

配置更改如果比较多推荐使用覆盖 `values.yaml` 的方式来安装 nginx ingress:

1. 导出默认的 `values.yaml`:

```
1. helm inspect values stable/nginx-ingress > values.yaml
```

2. 修改 `values.yaml` 中的配置
3. 执行 helm install 的时候去掉 `--set` 的方式设置的变量，替换为使用 `-f values.yaml`

有时可能更新 nginx ingress 的部署，滚动更新时可能造成部分连接异常，可以参考服务平滑更新最佳实践 使用 preStopHook 和 readinessProbe 保证服务平滑更新不中断，nginx ingress 默认加了 readinessProbe，但 preStop 没有加，我们可以修改 `values.yaml` 中 `controller.lifecycle`，加上 preStop，示例:

```
1.   lifecycle:
2.     preStop:
3.       exec:
4.         command: ["/bin/bash", "-c", "sleep 30"]
```

还可以 使用反亲和性避免单点故障，修改 `controller.affinity` 字段示例:

```
1.   affinity:
2.    podAntiAffinity:
3.      requiredDuringSchedulingIgnoredDuringExecution:
4.      - weight: 100
5.        labelSelector:
6.          matchExpressions:
7.          - key: app
8.            operator: In
9.            values:
```

```
            - nginx-ingress
          - key: component
            operator: In
            values:
            - controller
          - key: release
            operator: In
            values:
            - nginx
        topologyKey: kubernetes.io/hostname
```

参考资料

- Github 主页: https://github.com/kubernetes/ingress-nginx
- helm hub 主页: https://hub.helm.sh/charts/nginx/nginx-ingress
- 官方文档: https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/

Traefik Ingress

- 安装 traefik ingress controller

安装 traefik ingress controller

最佳安装方案

如何暴露 ingress 访问入口？最佳方案是使用 LoadBalancer 类型的 Service 来暴露，即创建外部负载均衡器来暴露流量，后续访问 ingress 的流量都走这个负载均衡器的地址，ingress 规则里面配的域名也要配置解析到这个负载均衡器的 IP 地址。

这种方式需要集群支持 LoadBalancer 类型的 Service，如果是云厂商提供的 k8s 服务，或者在云上自建集群并使用了云厂商提供的 cloud provider，也都是支持的，创建 LoadBalancer 类型的 Service 的时候会自动调云厂商的接口创建云厂商提供的负载均衡器产品(通常公网类型的负载均衡器是付费的)；如果你的集群不是前面说的情况，是自建集群并且有自己的负载均衡器方案，并部署了相关插件来适配，比如 MetalLB 和 Porter，这样也是可以支持 LoadBalancer 类型的 Service 的。

使用 helm 安装

```
helm install stable/traefik \
  --name traefik \
  --namespace kube-system \
  --set kubernetes.ingressClass=traefik \
  --set kubernetes.ingressEndpoint.useDefaultPublishedService=true \
  --set rbac.enabled=true
```

- `kubernetes.ingressClass=traefik`：创建的 ingress 中包含 `kubernetes.io/ingress.class` 这个 annotation 并且值与这里配置的一致，这个 traefik ingress controller 才会处理 (生成转发规则)
- `kubernetes.ingressEndpoint.useDefaultPublishedService=true`：这个置为 true 主要是为了让 ingress 的外部地址正确显示 (显示为负载均衡器的地址)，因为如果不配置这个，默认情况下会将 ingress controller 所有实例的节点 ip 写到 ingress 的 address 里
- `rbac.enabled` 默认为 false，如果没有事先给 default 的 service account 绑足够权限就会报错，通常置为 true，自动创建 rbac 规则

参考资料

- Github 主页：https://github.com/containous/traefik

- helm hub 主页: https://hub.helm.sh/charts/stable/traefik
- 官方文档: https://docs.traefik.io

Metrics 方案

安装 metrics server

官方 yaml 安装

下载:

```
git clone --depth 1 https://github.com/kubernetes-sigs/metrics-server.git
cd metrics-server
```

修改 `deploy/1.8+/metrics-server-deployment.yaml`，在 `args` 里增加 `--kubelet-insecure-tls` (防止 metrics server 访问 kubelet 采集指标时报证书问题 `x509: certificate signed by unknown authority`):

```
      containers:
      - name: metrics-server
        image: k8s.gcr.io/metrics-server-amd64:v0.3.6
        args:
          - --cert-dir=/tmp
          - --secure-port=4443
          - --kubelet-insecure-tls # 这里是新增的一行
```

安装:

```
kubectl apply -f deploy/1.8+/
```

参考资料

- Github 主页: https://github.com/kubernetes-sigs/metrics-server

- 服务高可用
- 本地 DNS 缓存
- 泛域名动态转发 Service
- 集群权限控制
- 实用工具和技巧
- 证书管理
- 集群配置管理
- 大规模集群优化

服务高可用

为了提高服务容错能力，我们通常会设置 `replicas` 给服务创建多个副本，但这并不意味着服务就实现高可用了，下面来介绍服务高可用部署最佳实践。

" class="reference-link">使用反亲和性避免单点故障

k8s 的设计就是假设节点是不可靠的，节点越多，发生软硬件故障导致节点不可用的几率就越高，所以我们通常需要给服务部署多个副本，根据实际情况调整 `replicas` 的值，如果值为 1 就必然存在单点故障，如果大于 1 但所有副本都调度到同一个节点，那还是有单点故障，所以我们不仅要有合理的副本数量，还需要让这些不同副本调度到不同的节点，打散开来避免单点故障，这个可以利用反亲和性来实现，示例：

```
affinity:
 podAntiAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
   - weight: 100
     labelSelector:
       matchExpressions:
       - key: k8s-app
         operator: In
         values:
         - kube-dns
     topologyKey: kubernetes.io/hostname
```

- `requiredDuringSchedulingIgnoredDuringExecution` 调度时必须满足该反亲和性条件，如果没有节点满足条件就不调度到任何节点（Pending）。如果不用这种硬性条件可以使用 `preferredDuringSchedulingIgnoredDuringExecution` 来指示调度器尽量满足反亲和性条件，如果没有满足条件的也可以调度到某个节点。
- `labelSelector.matchExpressions` 写该服务对应 pod 中 labels 的 key 与 value。
- `topologyKey` 这里用 `kubernetes.io/hostname` 表示避免 pod 调度到同一节点，如果你有更高的要求，比如避免调度到同一个可用区，实现异地多活，可以用 `failure-domain.beta.kubernetes.io/zone` 。通常不会去避免调度到同一个地域，因为一般同一个集群的节点都在一个地域，如果跨地域，即使用专线时延也会很大，所以 `topologyKey` 一般不至于用 `failure-domain.beta.kubernetes.io/region` 。

" class="reference-link">使用

PodDisruptionBudget 避免驱逐导致服务不可用

驱逐节点是一种有损操作，驱逐的原理：

1. 封锁节点（设为不可调度，避免新的 Pod 调度上来）。
2. 将该节点上的 Pod 删除。
3. ReplicaSet 控制器检测到 Pod 减少，会重新创建一个 Pod，调度到新的节点上。

这个过程是先删除，再创建，并非是滚动更新，因此更新过程中，如果一个服务的所有副本都在被驱逐的节点上，则可能导致该服务不可用。

我们再来下什么情况下驱逐会导致服务不可用：

1. 服务存在单点故障，所有副本都在同一个节点，驱逐该节点时，就可能造成服务不可用。
2. 服务在多个节点，但这些节点都被同时驱逐，所以这个服务的所有服务同时被删，也可能造成服务不可用。

针对第一点，我们可以 使用反亲和性避免单点故障。

针对第二点，我们可以通过配置 PDB（PodDisruptionBudget）来避免所有副本同时被删除，下面给出示例。

示例一（保证驱逐时 zookeeper 至少有两个副本可用）：

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: zookeeper
```

示例二（保证驱逐时 zookeeper 最多有一个副本不可用，相当于逐个删除并在其它节点重建）：

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  maxUnavailable: 1
  selector:
```

```
    matchLabels:
      app: zookeeper
```

更多请参考官方文档：https://kubernetes.io/docs/tasks/run-application/configure-pdb/

" class="reference-link">使用 preStopHook 和 readinessProbe 保证服务平滑更新不中断

如果服务不做配置优化，默认情况下更新服务期间可能会导致部分流量异常，下面我们来分析并给出最佳实践。

服务更新场景

我们先看下服务更新有哪些场景：

- 手动调整服务的副本数量
- 手动删除 Pod 触发重新调度
- 驱逐节点（主动或被动驱逐，Pod会先删除再在其它节点重建）
- 触发滚动更新（比如修改镜像 tag 升级程序版本）
- HPA（HorizontalPodAutoscaler）自动对服务进行水平伸缩
- VPA（VerticalPodAutoscaler）自动对服务进行垂直伸缩

更新过程连接异常的原因

滚动更新时，Service 对应的 Pod 会被创建或销毁，Service 对应的 Endpoint 也会新增或移除相应的 Pod IP:Port，然后 kube-proxy 会根据 Service 的 Endpoint 里的 Pod IP:Port 列表更新节点上的转发规则，而这里 kube-proxy 更新节点转发规则的动作并不是那么及时，主要是由于 K8S 的设计理念，各个组件的逻辑是解耦的，各自使用 Controller 模式 listAndWatch 感兴趣的资源并做出相应的行为，所以从 Pod 创建或销毁到 Endpoint 更新再到节点上的转发规则更新，这个过程是异步的，所以会造成转发规则更新不及时，从而导致服务更新期间部分连接异常。

我们分别分析下 Pod 创建和销毁到规则更新期间的过程：

1. Pod 被创建，但启动速度没那么快，还没等到 Pod 完全启动就被 Endpoint Controller 加入到 Service 对应 Endpoint 的 Pod IP:Port 列表，然后 kube-proxy watch 到更新也同步更新了节点上的 Service 转发规则（iptables/ipvs），如果这个时候有请求过来就可能被转发到还没完全启动完全的 Pod，这时 Pod 还不能正常处理请求，就会导致连接被拒绝。
2. Pod 被销毁，但是从 Endpoint Controller watch 到变化并更新 Service 对应

Endpoint 再到 kube-proxy 更新节点转发规则这期间是异步的，有个时间差，Pod 可能已经完全被销毁了，但是转发规则还没来得及更新，就会造成新来的请求依旧还能被转发到已经被销毁的 Pod，导致连接被拒绝。

" class="reference-link">平滑更新最佳实践

- 针对第一种情况，可以给 Pod 里的 container 加 readinessProbe (就绪检查)，通常是容器完全启动后监听一个 HTTP 端口，kubelet 发就绪检查探测包，正常响应说明容器已经就绪，然后修改容器状态为 Ready，当 Pod 中所有容器都 Ready 了这个 Pod 才会被 Endpoint Controller 加进 Service 对应 Endpoint IP:Port 列表，然后 kube-proxy 再更新节点转发规则，更新完了即便立即有请求被转发到的新的 Pod 也能保证能够正常处理连接，避免了连接异常。
- 针对第二种情况，可以给 Pod 里的 container 加 preStop hook，让 Pod 真正销毁前先 sleep 等待一段时间，留点时间给 Endpoint controller 和 kube-proxy 更新 Endpoint 和转发规则，这段时间 Pod 处于 Terminating 状态，即便在转发规则更新完全之前有请求被转发到这个 Terminating 的 Pod，依然可以被正常处理，因为它还在 sleep，没有被真正销毁。

最佳实践 yaml 示例:

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      component: nginx
  template:
    metadata:
      labels:
        component: nginx
    spec:
      containers:
      - name: nginx
        image: "nginx"
        ports:
        - name: http
          hostPort: 80
          containerPort: 80
          protocol: TCP
```

```
        readinessProbe:
          httpGet:
            path: /healthz
            port: 80
            httpHeaders:
            - name: X-Custom-Header
              value: Awesome
          initialDelaySeconds: 15
          timeoutSeconds: 1
        lifecycle:
          preStop:
            exec:
              command: ["/bin/bash", "-c", "sleep 30"]
```

参考资料

- Container probes: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes
- Container Lifecycle Hooks: https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/

" class="reference-link">解决长连接服务扩容失效

在现网运营中，有很多场景为了提高效率，一般都采用建立长连接的方式来请求。我们发现在客户端以长连接请求服务端的场景下，K8S的自动扩容会失效。原因是客户端长连接一直保留在老的Pod容器中，新扩容的Pod没有新的连接过来，导致K8S按照步长扩容第一批Pod之后就停止了扩容操作，而且新扩容的Pod没能承载请求，进而出现服务过载的情况，自动扩容失去了意义。

对长连接扩容失效的问题，我们的解决方法是将长连接转换为短连接。我们参考了 nginx keepalive 的设计，nginx 中 keepalive_requests 这个配置项设定了一个TCP连接能处理的最大请求数，达到设定值(比如1000)之后服务端会在 http 的 Header 头标记 “ `Connection:close` ”，通知客户端处理完当前的请求后关闭连接，新的请求需要重新建立TCP连接，所以这个过程中不会出现请求失败，同时又达到了将长连接按需转换为短连接的目的。通过这个办法客户端和云K8S服务端处理完一批请求后不断的更新TCP连接，自动扩容的新Pod能接收到新的连接请求，从而解决了自动扩容失效的问题。

由于Golang并没有提供方法可以获取到每个连接处理过的请求数，我们重写了 `net.Listener` 和

`net.Conn`，注入请求计数器，对每个连接处理的请求做计数，并通过 `net.Conn.LocalAddr()` 获得计数值，判断达到阈值 1000 后在返回的 Header 中插入 “ `Connection:close` ” 通知客户端关闭连接，重新建立连接来发起请求。以上处理逻辑用 Golang 实现示例代码如下：

```
1.  package main
2.
3.  import (
4.      "net"
5.      "github.com/gin-gonic/gin"
6.      "net/http"
7.  )
8.
9.  // 重新定义net.Listener
10. type counterListener struct {
11.     net.Listener
12. }
13.
14. // 重写net.Listener.Accept(),对接收到的连接注入请求计数器
15. func (c *counterListener) Accept() (net.Conn, error) {
16.     conn, err := c.Listener.Accept()
17.     if err != nil {
18.         return nil, err
19.     }
20.     return &counterConn{Conn: conn}, nil
21. }
22.
23. // 定义计数器counter和计数方法Increment()
24. type counter int
25.
26. func (c *counter) Increment() int {
27.     *c++
28.     return int(*c)
29. }
30.
31. // 重新定义net.Conn,注入计数器ct
32. type counterConn struct {
33.     net.Conn
34.     ct counter
35. }
36.
37. // 重写net.Conn.LocalAddr(), 返回本地网络地址的同时返回该连接累计处理过的请求数
38. func (c *counterConn) LocalAddr() net.Addr {
```

```
39.     return &counterAddr{c.Conn.LocalAddr(), &c.ct}
40. }
41. 
42. // 定义TCP连接计数器,指向连接累计请求的计数器
43. type counterAddr struct {
44.     net.Addr
45.     *counter
46. }
47. 
48. func main() {
49.     r := gin.New()
50.     r.Use(func(c *gin.Context) {
51.         localAddr := c.Request.Context().Value(http.LocalAddrContextKey)
52.         if ct, ok := localAddr.(interface{ Increment() int }); ok {
53.             if ct.Increment() >= 1000 {
54.                 c.Header("Connection", "close")
55.             }
56.         }
57.         c.Next()
58.     })
59.     r.GET("/", func(c *gin.Context) {
60.         c.String(200, "plain/text", "hello")
61.     })
62.     l, err := net.Listen("tcp", ":8080")
63.     if err != nil {
64.         panic(err)
65.     }
66.     err = http.Serve(&counterListener{l}, r)
67.     if err != nil {
68.         panic(err)
69.     }
70. }
```

本地 DNS 缓存

为什么需要本地 DNS 缓存

- 减轻集群 DNS 解析压力，提高 DNS 性能
- 避免 netfilter 做 DNAT 导致 conntrack 冲突引发 DNS 5 秒延时

> 镜像底层库 DNS 解析行为默认使用 UDP 在同一个 socket 并发 A 和 AAAA 记录请求，由于 UDP 无状态，两个请求可能会并发创建 conntrack 表项，如果最终 DNAT 成同一个集群 DNS 的 Pod IP 就会导致 conntrack 冲突，由于 conntrack 的创建和插入是不加锁的，最终后面插入的 conntrack 表项就会被丢弃，从而请求超时，默认 5s 后重试，造成现象就是 DNS 5 秒延时；底层库是 glibc 的容器镜像可以通过配 resolv.conf 参数来控制 DNS 解析行为，不用 TCP 或者避免相同五元组并发(使用串行解析 A 和 AAAA 避免并发或者使用不同 socket 发请求避免相同源端口)，但像基于 alpine 镜像的容器由于底层库是 musl libc，不支持这些 resolv.conf 参数，也就无法规避，所以最佳方案还是使用本地 DNS 缓存。

原理

本地 DNS 缓存以 DaemonSet 方式在每个节点部署一个使用 hostNetwork 的 Pod，创建一个网卡绑上本地 DNS 的 IP，本机的 Pod 的 DNS 请求路由到本地 DNS，然后取缓存或者继续使用 TCP 请求上游集群 DNS 解析 (由于使用 TCP，同一个 socket 只会做一遍三次握手，不存在并发创建 conntrack 表项，也就不会有 conntrack 冲突)

IVPS 模式下需要修改 kubelet 参数

有两点需要注意下：

1. ipvs 模式下需要改 kubelet `--cluster-dns` 参数，指向一个非 kube-dns service 的 IP，通常用 `169.254.20.10`，Daemonset 会在每个节点创建一个网卡绑这个 IP，Pod 向本节点这个 IP 发 DNS 请求，本机 DNS 再代理到上游集群 DNS
2. iptables 模式下不需要改 kubelet `--cluster-dns` 参数，Pod 还是向原来的集群 DNS 请求，节点上有这个 IP 监听，被本机拦截，再请求集群上游 DNS (使用集群 DNS 的另一个 CLUSTER IP，来自事先创建好的 Service，跟原集群 DNS 的 Service 有相同的 selector 和 endpoint)

ipvs 模式下必须修改 kubelet 参数的原因是：如果不修改，DaemonSet Pod 在本机创建了网卡，会绑跟集群 DNS 的 CLUSTER IP， 但 kube-ipvs0 这个 dummy interface 上也会绑这个 IP (这是 ipvs 的机制，为了能让报文到达 INPUT 链被 ipvs 处理)，所以 Pod 请求集群

DNS 的报文最终还是会被 ipvs 处理，DNAT 成集群 DNS 的 Pod IP，最终路由到集群 DNS，相当于本机 DNS 就没有作用了。

IPVS 模式下部署方法

这里我们假设是 ipvs 模式，下面给出本地 DNS 缓存部署方法。

创建 ServiceAccount 与集群上游 DNS 的 Service:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: node-local-dns
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
---
apiVersion: v1
kind: Service
metadata:
  name: kube-dns-upstream
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
    kubernetes.io/name: "KubeDNSUpstream"
spec:
  ports:
  - name: dns
    port: 53
    protocol: UDP
    targetPort: 53
  - name: dns-tcp
    port: 53
    protocol: TCP
    targetPort: 53
  selector:
    k8s-app: kube-dns
EOF
```

获取 `kube-dns-upstream` 的 CLUSTER IP:

```
UPSTREAM_CLUSTER_IP=$(kubectl -n kube-system get services kube-dns-upstream -o
jsonpath="{.spec.clusterIP}")
```

部署 DaemonSet:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: node-local-dns
  namespace: kube-system
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
data:
  Corefile: |
    cluster.local:53 {
        errors
        cache {
                success 9984 30
                denial 9984 5
        }
        reload
        loop
        bind 169.254.20.10
        forward . ${UPSTREAM_CLUSTER_IP} {
                force_tcp
        }
        prometheus :9253
        health 169.254.20.10:8080
        }
    in-addr.arpa:53 {
        errors
        cache 30
        reload
        loop
        bind 169.254.20.10
        forward . ${UPSTREAM_CLUSTER_IP} {
                force_tcp
        }
```

```
        prometheus :9253
        }
    ip6.arpa:53 {
        errors
        cache 30
        reload
        loop
        bind 169.254.20.10
        forward . ${UPSTREAM_CLUSTER_IP} {
                force_tcp
        }
        prometheus :9253
        }
    .:53 {
        errors
        cache 30
        reload
        loop
        bind 169.254.20.10
        forward . /etc/resolv.conf {
                force_tcp
        }
        prometheus :9253
        }
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-local-dns
  namespace: kube-system
  labels:
    k8s-app: node-local-dns
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
spec:
  updateStrategy:
    rollingUpdate:
      maxUnavailable: 10%
  selector:
    matchLabels:
      k8s-app: node-local-dns
  template:
```

```
77.     metadata:
78.        labels:
79.           k8s-app: node-local-dns
80.     spec:
81.       priorityClassName: system-node-critical
82.       serviceAccountName: node-local-dns
83.       hostNetwork: true
84.       dnsPolicy: Default  # Don't use cluster DNS.
85.       tolerations:
86.       - key: "CriticalAddonsOnly"
87.         operator: "Exists"
88.       containers:
89.       - name: node-cache
90.         image: k8s.gcr.io/k8s-dns-node-cache:1.15.7
91.         resources:
92.           requests:
93.             cpu: 25m
94.             memory: 5Mi
          args: [ "-localip", "169.254.20.10", "-conf", "/etc/Corefile", "-
95. upstreamsvc", "kube-dns-upstream" ]
96.         securityContext:
97.           privileged: true
98.         ports:
99.         - containerPort: 53
100.          name: dns
101.          protocol: UDP
102.        - containerPort: 53
103.          name: dns-tcp
104.          protocol: TCP
105.        - containerPort: 9253
106.          name: metrics
107.          protocol: TCP
108.        livenessProbe:
109.          httpGet:
110.            host: 169.254.20.10
111.            path: /health
112.            port: 8080
113.          initialDelaySeconds: 60
114.          timeoutSeconds: 5
115.        volumeMounts:
116.        - mountPath: /run/xtables.lock
117.          name: xtables-lock
```

```
            readOnly: false
          - name: config-volume
            mountPath: /etc/coredns
          - name: kube-dns-config
            mountPath: /etc/kube-dns
        volumes:
        - name: xtables-lock
          hostPath:
            path: /run/xtables.lock
            type: FileOrCreate
        - name: kube-dns-config
          configMap:
            name: kube-dns
            optional: true
        - name: config-volume
          configMap:
            name: node-local-dns
            items:
              - key: Corefile
                path: Corefile.base
EOF
```

验证是否启动：

```
$ kubectl -n kube-system get pod -o wide | grep node-local-dns
node-local-dns-2m9b6                1/1       Running   0          15m
10.0.0.28    10.0.0.28
node-local-dns-qgrwl                1/1       Running   0          15m
10.0.0.186   10.0.0.186
node-local-dns-s5mhw                1/1       Running   0          51s
10.0.0.76    10.0.0.76
```

我们需要替换 kubelet 的 `--cluster-dns` 参数，指向 `169.254.20.10` 这个 IP。

在TKE上，对于存量节点，登录节点执行以下命令：

```
sed -i '/CLUSTER_DNS/c\CLUSTER_DNS="--cluster-dns=169.254.20.10"'
/etc/kubernetes/kubelet
systemctl restart kubelet
```

对于增量节点，可以将上述命令放入新增节点的 user-data，以便加入节点后自动执行。

后续新增才会用到本地 DNS 缓存，对于存量 Pod 可以销毁重建，比如改下 Deployment 中

template 里的 annotation，触发 Deployment 所有 Pod 滚动更新，如果怕滚动更新造成部分流量异常，可以参考 服务更新最佳实践

参考资料

- https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns
- https://github.com/kubernetes/enhancements/blob/master/keps/sig-network/20190424-NodeLocalDNS-beta-proposal.md

泛域名转发

需求

集群对外暴露了一个公网IP作为流量入口(可以是 Ingress 或 Service)，DNS 解析配置了一个泛域名指向该IP（比如 `*.test.imroc.io` ），现希望根据请求中不同 Host 转发到不同的后端 Service。比如 `a.test.imroc.io` 的请求被转发到 `my-svc-a` ， `b.test.imroc.io` 的请求转发到 `my-svc-b` 。当前 K8S 的 Ingress 并不原生支持这种泛域名转发规则，本文将给出一个解决方案来实现泛域名转发。

简单做法

先说一种简单的方法，这也是大多数人的第一反应：配置 **Ingress** 规则

假如泛域名有两个不同 Host 分别转发到不同 Service，Ingress 类似这样写：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - host: a.test.imroc.io
    http:
      paths:
      - backend:
          serviceName: my-svc-a
          servicePort: 80
        path: /
  - host: b.test.imroc.io
    http:
      paths:
      - backend:
          serviceName: my-svc-b
          servicePort: 80
        path: /
```

但是！如果 Host 非常多会怎样？（比如200+）

- 每次新增 Host 都要改 Ingress 规则，太麻烦
- 单个 Ingress 上面的规则越来越多，更改规则对 LB 的压力变大，可能会导致偶尔访问不了

正确姿势

我们可以约定请求中泛域名 Host 通配符的 `*` 号匹配到的字符跟 Service 的名字相关联（可以是相等，或者 Service 统一在前面加个前缀，比如 `a.test.imroc.io` 转发到 `my-svc-a` 这个 Service），集群内起一个反向代理服务，匹配泛域名的请求全部转发到这个代理服务上，这个代理服务只做一件简单的事，解析 Host，正则匹配抓取泛域名中 `*` 号这部分，把它转换为 Service 名字，然后在集群里转发（集群 DNS 解析）

这个反向代理服务可以是 Nginx+Lua脚本 来实现，或者自己写个简单程序来做反向代理，这里我用 OpenResty 来实现，它可以看成是 Nginx 的发行版，自带 lua 支持。

有几点需要说明下：

- 我们使用 nginx 的 `proxy_pass` 来反向代理到后端服务，`proxy_pass` 后面跟的变量，我们需要用 lua 来判断 Host 修改变量
- nginx 的 `proxy_pass` 后面跟的如果是可变的域名（非IP，需要 dns 解析），它需要一个域名解析器，不会走默认的 dns 解析，需要在 `nginx.conf` 里添加 `resolver` 配置项来设置一个外部的 dns 解析器
- 这个解析器我们是用 go-dnsmasq 来实现，它可以将集群的 dns 解析代理给 nginx，以 sidecar 的形式注入到 pod 中，监听 53 端口

`nginx.conf` 里关键的配置如下图所示：

```
    resolver      127.0.0.1:53 ipv6=off;
    server {
        listen 80;

        location / {
            set $service   '';
            rewrite_by_lua '
                local host = ngx.var.host
                local m = ngx.re.match(host, "(.+).test.imroc.io")
                if m then
                    ngx.var.service = "my-svc-" .. m[1]
                end
            ';
            proxy_pass http://$service;
        }
    }
}
```

下面给出完整的 yaml 示例

`proxy.yaml` :

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  labels:
    component: nginx
  name: proxy
spec:
  replicas: 1
  selector:
    matchLabels:
      component: nginx
  template:
    metadata:
      labels:
        component: nginx
    spec:
      containers:
      - name: nginx
        image: "openresty/openresty:centos"
        ports:
```

```
        - name: http
          containerPort: 80
          protocol: TCP
        volumeMounts:
        - mountPath: /usr/local/openresty/nginx/conf/nginx.conf
          name: config
          subPath: nginx.conf
      - name: dnsmasq
        image: "janeczku/go-dnsmasq:release-1.0.7"
        args:
          - --listen
          - "127.0.0.1:53"
          - --default-resolver
          - --append-search-domains
          - --hostsfile=/etc/hosts
          - --verbose
      volumes:
      - name: config
        configMap:
          name: configmap-nginx

---

apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    component: nginx
  name: configmap-nginx
data:
  nginx.conf: |-
    worker_processes  1;

    error_log  /error.log;

    events {
        accept_mutex on;
        multi_accept on;
        use epoll;
        worker_connections  1024;
    }

```

```
63.
64.     http {
65.         include       mime.types;
66.         default_type  application/octet-stream;
            log_format  main  '$time_local $remote_user $remote_addr $host
67. $request_uri $request_method $http_cookie '
68.                           '$status $body_bytes_sent "$http_referer" '
69.                           '"$http_user_agent" "$http_x_forwarded_for" '
                              '$request_time $upstream_response_time
70. "$upstream_cache_status"';
71.
            log_format  browser '$time_iso8601 $cookie_km_uid $remote_addr $host
72. $request_uri $request_method '
73.                           '$status $body_bytes_sent "$http_referer" '
74.                           '"$http_user_agent" "$http_x_forwarded_for" '
                              '$request_time $upstream_response_time
    "$upstream_cache_status" $http_x_requested_with $http_x_real_ip $upstream_addr
75. $request_body';
76.
77.         log_format client '{"@timestamp":"$time_iso8601",'
78.                           '"time_local":"$time_local",'
79.                           '"remote_user":"$remote_user",'
80.                           '"http_x_forwarded_for":"$http_x_forwarded_for",'
81.                           '"host":"$server_addr",'
82.                           '"remote_addr":"$remote_addr",'
83.                           '"http_x_real_ip":"$http_x_real_ip",'
84.                           '"body_bytes_sent":$body_bytes_sent,'
85.                           '"request_time":$request_time,'
86.                           '"status":$status,'
87.                           '"upstream_response_time":"$upstream_response_time",'
88.                           '"upstream_response_status":"$upstream_status",'
89.                           '"request":"$request",'
90.                           '"http_referer":"$http_referer",'
91.                           '"http_user_agent":"$http_user_agent"}';
92.
93.         access_log  /access.log  main;
94.
95.         sendfile        on;
96.
97.         keepalive_timeout 120s 100s;
98.         keepalive_requests 500;
99.         send_timeout 60000s;
100.        client_header_buffer_size 4k;
```

```
        proxy_ignore_client_abort on;
        proxy_buffers 16 32k;
        proxy_buffer_size 64k;

        proxy_busy_buffers_size 64k;

        proxy_send_timeout 60000;
        proxy_read_timeout 60000;
        proxy_connect_timeout 60000;
        proxy_cache_valid 200 304 2h;
        proxy_cache_valid 500 404 2s;
        proxy_cache_key $host$request_uri$cookie_user;
        proxy_cache_methods GET HEAD POST;

        proxy_redirect off;
        proxy_http_version 1.1;
        proxy_set_header Host                $http_host;
        proxy_set_header X-Real-IP           $remote_addr;
        proxy_set_header X-Forwarded-For     $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto   $scheme;
        proxy_set_header X-Frame-Options     SAMEORIGIN;

        server_tokens off;
        client_max_body_size 50G;
        add_header X-Cache $upstream_cache_status;
        autoindex off;

        resolver      127.0.0.1:53 ipv6=off;

        server {
            listen 80;

            location / {
                set $service  '';
                rewrite_by_lua '
                    local host = ngx.var.host
                    local m = ngx.re.match(host, "(.+).test.imroc.io")
                    if m then
                        ngx.var.service = "my-svc-" .. m[1]
                    end
                ';
                proxy_pass http://$service;
```

```
            }
        }
    }
```

让该代理服务暴露公网访问可以用 Service 或 Ingress

用 Service 的示例 (`service.yaml`):

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    component: nginx
  name: service-nginx
spec:
  type: LoadBalancer
  ports:
  - name: http
    port: 80
    targetPort: http
  selector:
    component: nginx
```

用 Ingress 的示例 (`ingress.yaml`):

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-nginx
spec:
  rules:
  - host: "*.test.imroc.io"
    http:
      paths:
      - backend:
          serviceName: service-nginx
          servicePort: 80
        path: /
```

集群权限控制

账户类型

K8S 主要有以下两种账户类型概念：

- 用户账户（`User`）：控制人的权限。
- 服务账户（`ServiceAccount`）：控制应用程序的权限

如果开启集群审计，就可以区分某个操作是哪个用户或哪个应程序执行的。

利用 CSR API 创建用户

k8s 支持 CSR API，通过创建 `CertificateSigningRequest` 资源就可以发起 CSR 请求，管理员审批通过之后 `kube-controller-manager` 就会为我们签发证书，确保 `kube-controller-manager` 配了根证书密钥对：

```
1.  --cluster-signing-cert-file=/var/lib/kubernetes/ca.pem
2.  --cluster-signing-key-file=/var/lib/kubernetes/ca-key.pem
```

创建步骤

我们用 cfssl 来创建 key 和 csr 文件，所以需要先安装 cfssl：

```
1.  curl -L https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 -o cfssl
2.  curl -L https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64 -o cfssljson
3.  curl -L https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64 -o cfssl-certinfo
4.
5.  chmod +x cfssl cfssljson cfssl-certinfo
6.  sudo mv cfssl cfssljson cfssl-certinfo /usr/local/bin/
```

指定要创建的用户名：

```
1.  USERNAME="roc"
```

再创建 key 和 csr 文件：

```
1.  cat <<EOF | cfssl genkey - | cfssljson -bare ${USERNAME}
2.  {
3.    "CN": "${USERNAME}",
4.    "key": {
5.      "algo": "rsa",
6.      "size": 2048
7.    }
8.  }
9.  EOF
```

生成以下文件：

```
1.  roc.csr
```

```
roc-key.pem
```

创建 `CertificateSigningRequest`（发起 CSR 请求）：

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: ${USERNAME}
spec:
  request: $(cat ${USERNAME}.csr | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - client auth
EOF
```

管理员审批 CSR 请求：

```
# 查看 csr
# kubectl get csr

# 审批 csr
kubectl certificate approve ${USERNAME}
```

获取证书：

```
kubectl get csr ${USERNAME} -o jsonpath={.status.certificate} | base64 --decode
> ${USERNAME}.pem
```

得到证书文件：

```
roc.pem
```

至此，我们已经创建好了用户，用户的证书密钥对文件：

```
roc.pem
roc-key.pem
```

配置 kubeconfig

```
# 增加 user
kubectl config set-credentials ${USERNAME} --embed-certs=true --client-certificate=${USERNAME}.pem --client-key=${USERNAME}-key.pem

# 如果还没配 cluster，可以通过下面命令配一下
kubectl config set-cluster <cluster> --server=<apiserver-url> --certificate-authority=<ca-cert-file>

# 增加 context，绑定 cluster 和 user
kubectl config set-context <context> --cluster=<cluster> --user=${USERNAME}

# 使用刚增加的 context
kubectl config use-context <context>
```

配置用户权限

我们可以用 RBAC 控制用户权限，参考 使用 RBAC 控制用户权限

参考资料

- Manage TLS Certificates in a Cluster: https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster/

控制用户权限

为了简单方便，小集群或测试环境集群我们通常使用最高权限的 admin 账号，可以做任何操作，但是如果是重要的生产环境集群，可以操作集群的人比较多，如果这时还用这个账号可能就会比较危险，一旦有人误操作或故意搞事就可能酿成大错，即使 apiserver 开启审计也无法知道是谁做的操作，所以最好控制下权限，根据人的级别或角色创建拥有对应权限的账号，这个可以通过 RBAC 来实现(确保 `kube-apiserver` 启动参数 `--authorization-mode=RBAC`)，基本思想是创建 User 或 ServiceAccount 绑定 Role 或 ClusterRole 来控制权限。

User 来源

User 的来源有多种:

- token 文件: 给 `kube-apiserver` 启动参数 `--token-auth-file` 传一个 token 认证文件，比如: `--token-auth-file=/etc/kubernetes/known_tokens.csv`
  - token 文件每一行表示一个用户，示例: `wJmq****PPWj,admin,admin,system:masters`
  - 第一个字段是 token 的值，最后一个字段是用户组，token 认证用户名不重要，不会识别
- 证书: 通过使用 CA 证书给用户签发证书，签发的证书中 `CN` 字段是用户名，`O` 是用户组

" class="reference-link">使用 RBAC 控制用户权限

下面给出几个 RBAC 定义示例。

给 roc 授权 test 命名空间所有权限，istio-system 命名空间的只读权限:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: admin
  namespace: test
rules:
  - apiGroups: ["*"]
    resources: ["*"]
    verbs: ["*"]

---

```

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: admin-to-roc
  namespace: test
subjects:
  - kind: User
    name: roc
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: admin
  apiGroup: rbac.authorization.k8s.io

---

kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: readonly
  namespace: istio-system
rules:
  - apiGroups: ["*"]
    resources: ["*"]
    verbs: ["get", "watch", "list"]

---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: readonly-to-roc
  namespace: istio-system
subjects:
  - kind: User
    name: roc
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: readonly
  apiGroup: rbac.authorization.k8s.io
```

给 roc 授权整个集群的只读权限：

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: readonly
rules:
  - apiGroups: ["*"]
    resources: ["*"]
    verbs: ["get", "watch", "list"]

---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: readonly-to-roc
subjects:
  - kind: User
    name: roc
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: readonly
  apiGroup: rbac.authorization.k8s.io
```

给 manager 用户组里所有用户授权 secret 读权限:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
```

```
    name: manager
    apiGroup: rbac.authorization.k8s.io
roleRef:
    kind: ClusterRole
    name: secret-reader
    apiGroup: rbac.authorization.k8s.io
```

配置 kubeconfig

```
# 如果使用证书认证，使用下面命令配置用户认证信息
kubectl config set-credentials <user> --embed-certs=true --client-certificate=<client-cert-file> --client-key=<client-key-file>

# 如果使用 token 认证，使用下面命令配置用户认证信息
# kubectl config set-credentials <user> --token='<token>'

# 配置cluster entry
kubectl config set-cluster <cluster> --server=<apiserver-url> --certificate-authority=<ca-cert-file>
# 配置context entry
kubectl config set-context <context> --cluster=<cluster> --user=<user>
# 配置当前使用的context
kubectl config use-context <context>
# 查看
kubectl config view
```

参考资料

- https://kubernetes.io/zh/docs/reference/access-authn-authz/service-accounts-admin/
- https://kubernetes.io/docs/reference/access-authn-authz/rbac/

控制应用权限

不仅用户（人）可以操作集群，应用（程序）也可以操作集群，通过给 Pod 设置 Serivce Account 来对应用进行授权，如果不设置会默认配置一个“default”的 Service Account，几乎没有权限。

原理

创建 Pod 时，在 apiserver 中的 service account admission controller 检测 Pod 是否指定了 ServiceAccount，如果没有就自动设置一个“default”，如果指定了会检测指定的 ServiceAccount 是否存在，不存在的话会拒绝该 Pod，存在话就将此 ServiceAccount 对应的 Secret 挂载到 Pod 中每个容器的 `/var/run/secrets/kubernetes.io/serviceaccount` 这个路径，这个 Secret 是 controller manager 中 token controller 去 watch ServiceAccount，为每个 ServiceAccount 生成对应的 token 类型的 Secret 得来的。

Pod 内的程序如果要调用 apiserver 接口操作集群，会使用 SDK，通常是 client-go ， SDK 使用 in-cluster 的方式调用 apiserver，从固定路径 `/var/run/secrets/kubernetes.io/serviceaccount` 读取认证配置信息去连 apiserver，从而实现认证，再结合 RBAC 配置可以实现权限控制。

使用 RBAC 细化应用权限

ServiceAccount 仅针对某个命名空间，所以 Pod 指定的 ServiceAccount 只能引用当前命名空间的 ServiceAccount 的，即便是“default” 每个命名空间也都是相互独立的，下面给出几个 RBAC 定义示例。

`build-robot` 这个 ServiceAccount 可以读取 build 命名空间中 Pod 的信息和 log:

```
1.  apiVersion: v1
2.  kind: ServiceAccount
3.  metadata:
4.    name: build-robot
5.    namespace: build
6.
7.  ---
8.
9.  apiVersion: rbac.authorization.k8s.io/v1
10. kind: Role
11. metadata:
```

```
  namespace: build
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]

---

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: build
subjects:
- kind: ServiceAccount
  name: build-robot
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

为 Pod 指定 ServiceAccount

示例:

```
apiVersion: v1
kind: Pod
metadata:
  name: build
  namespace: build
spec:
  containers:
  - image: imroc/build-robot:v1
    name: builder
  serviceAccountName: build-robot
```

" class="reference-link">为应用默认指定 imagePullSecrets

ServiceAccount 中也可以指定 imagePullSecrets，也就是只要给 Pod 指定了这个 ServiceAccount，就有对应的 imagePullSecrets，而如果不指定 ServiceAccount 会默认指定 “default”，我们可以给 “default” 这个 ServiceAccount 指定 imagePullSecrets 来实现给某个命名空间指定默认的 imagePullSecrets

创建 imagePullSecrets:

```
1. kubectl create secret docker-registry <secret-name> --docker-server=<your-registry-server> --docker-username=<your-name> --docker-password=<your-password> --docker-email=<your-email> -n <namespace>
```

- `<secret-name>`：是要创建的 imagePullSecrets 的名称
- `<namespace>`：是要创建的 imagePullSecrets 所在命名空间
- `<your-registry-server>`：是你的私有仓库的地址
- `<your-name>`：是你的 Docker 用户名
- `<your-password>` 是你的 Docker 密码
- `<your-email>` 是你的 Docker 邮箱

指定默认 imagePullSecrets:

```
1. kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "<secret-name>"}]}' -n <namespace>
```

- `<secret-name>`：是 ServiceAccount 要关联的 imagePullSecrets 的名称
- `<namespace>`：是 ServiceAccount 所在的命名空间，跟 imagePullSecrets 在同一个命名空间

参考资料

- Configure Service Accounts for Pods: https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

实用工具和技巧

kubectl 高效技巧

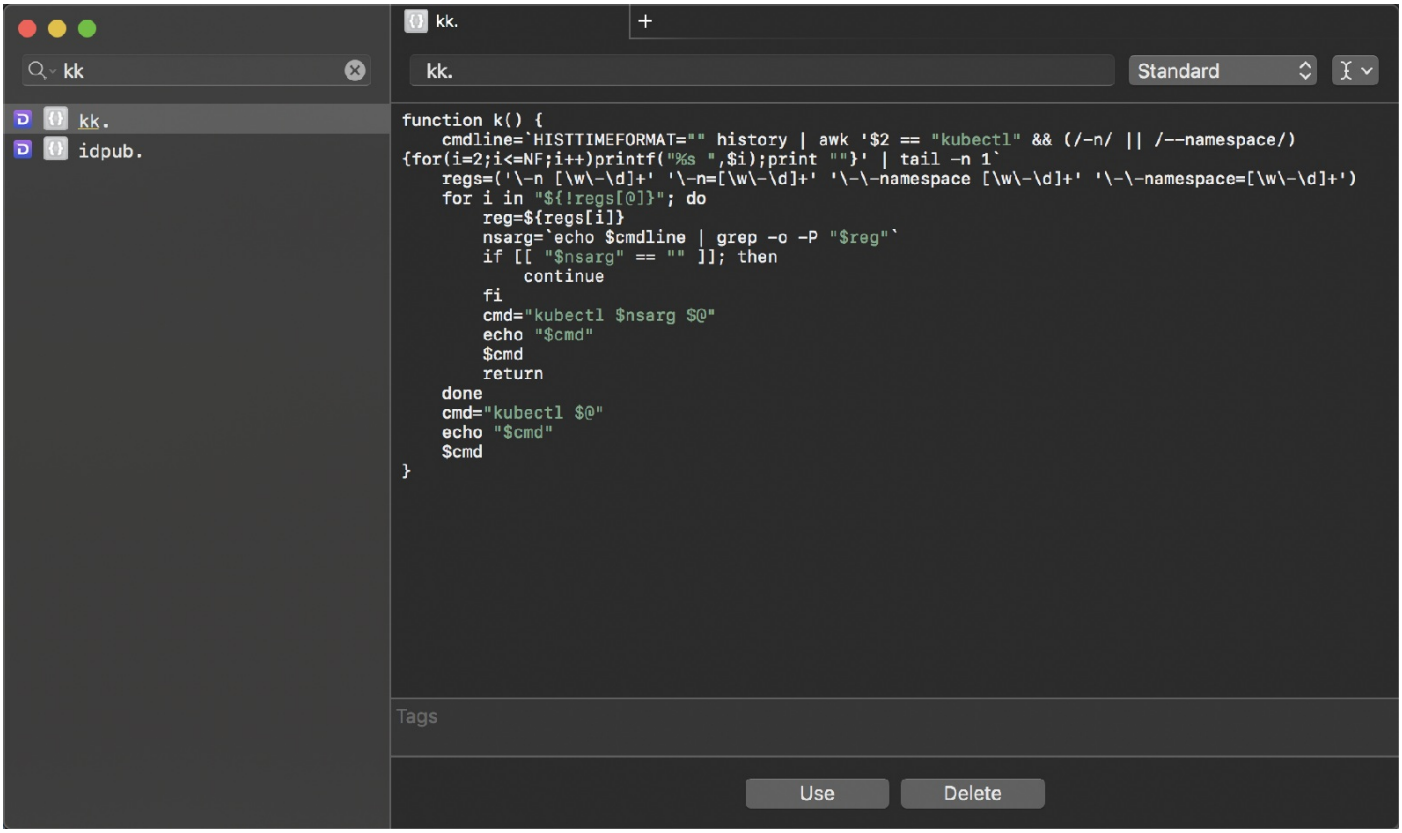
k 命令

是否有过因为使用 kubectl 经常需要重复输入命名空间而苦恼？是否觉得应该要有个记住命名空间的功能，自动记住上次使用的命名空间，不需要每次都输入？可惜没有这种功能，但是，本文会教你一个非常巧妙的方法完美帮你解决这个痛点。

将如下脚本粘贴到当前shell(注册k命令到当前终端session):

```
function k() {
    cmdline=`HISTTIMEFORMAT="" history | awk '$2 == "kubectl" && (/-n/ || /--
namespace/) {for(i=2;i<=NF;i++)printf("%s ",$i);print ""}' | tail -n 1`
    regs=('\-n [\w\-\d]+' '\-n=[\w\-\d]+' '\-\-namespace [\w\-\d]+' '\-\-
namespace=[\w\-\d]+')
    for i in "${!regs[@]}"; do
        reg=${regs[i]}
        nsarg=`echo $cmdline | grep -o -P "$reg"`
        if [[ "$nsarg" == "" ]]; then
            continue
        fi
        cmd="kubectl $nsarg $@"
        echo "$cmd"
        $cmd
        return
    done
    cmd="kubectl $@"
    echo "$cmd"
    $cmd
}
```

mac 用户可以使用 dash 的 snippets 功能快速将上面的函数粘贴，使用 `kk.` 作为触发键(dash snippets可以全局监听键盘输入，使用指定的输入作为触发而展开配置的内容，相当于是全局代码片段)，以后在某个终端想使用 `k` 的时候按下 `kk.` 就可以将 `k` 命令注册到当前终端，dash snippets 配置如图所示:



将 `k` 当作 `kubectl` 来用，只是不需要输入命名空间，它会调用 `kubectl` 并自动加上上次使用的非默认的命名空间，如果想切换命名空间，再常规的使用一次 `kubectl` 就行，下面是示范：

哈哈，是否感觉可以少输入很多字符，提高 `kubectl` 使用效率了？这是目前我探索解决 `kubectl` 重复输入命名空间的最好方案，一开始是受 `fuck命令` 的启发，想用 `go` 语言开发个 `k` 命令，但是发现两个缺点：

- 需要安装二进制才可以使用（对于需要在多个地方用kubectl管理多个集群的人来说实在太麻烦）
- 如果当前 shell 默认没有将历史输入记录到 history 文件(bash 的 history 文件默认是 `~/.bash_history`)，那么将无法准确知道上一次 kubectl 使用的哪个命名空间

这里解释下第二个缺点的原因：ssh 连上服务器会启动一个 shell 进程，通常是 bash，大多 bash 默认配置会实时将历史输入追加到 `~/.bash_history` 里，所以开多个ssh使用history命令看到的历史输入是一样的，但有些默认不会实时记录历史到 `~/.bash_history` ，而是记在当前 shell 进程的内存中，在 shell 退出时才会写入到文件。这种情况新起的进程是无法知道当前 shell 的最近历史输入的，fuck命令 也不例外。

所以最完美的解决方案就是注册函数到当前shell来调用，配合 dash 的 snippets 功能可以实现快速注册，解决复制粘贴的麻烦

实用 yaml 片段

最简单的 nginx 测试服务

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx

---

apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  type: ClusterIP
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    app: nginx
```

实用命令与脚本

获取集群所有节点占用的 podCIDR:

```
$ kubectl get node -o jsonpath='{range .items[*]}{@.spec.podCIDR}{"\n"}{end}'
172.16.4.0/24
172.16.0.0/24
172.16.6.0/24
```

集群证书管理

- 安装 cert-manager
- 使用 cert-manager 自动生成证书

安装 cert-manager

参考官方文档：https://docs.cert-manager.io/en/latest/getting-started/install/kubernetes.html

介绍几种安装方式，不管是用哪种我们都先规划一下使用哪个命名空间，推荐使用 `cert-manger` 命名空间，如果使用其它的命名空间需要做些更改，会稍微有点麻烦，先创建好命名空间：

```
1. kubectl create namespace cert-manager
```

cert-manager 部署时会生成 ValidatingWebhookConfiguration 来注册 [ValidatingAdmissionWebhook])(ValidatingAdmissionWebhook) 来实现 CRD 校验，而 `ValidatingWebhookConfiguration` 里需要写入 `cert-manager` 自身校验服务端的证书信息，就需要在自己命名空间创建 `ClusterIssuer` 和 `Certificate` 来自动创建证书，创建这些 CRD 资源又会被校验服务端校验，但校验服务端证书还没有创建所以校验请求无法发送到校验服务端，这就是一个鸡生蛋还是蛋生鸡的问题了，所以我们需要关闭 `cert-manager` 所在命名空间的 CRD 校验，通过打 label 来实现：

```
1. kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true
```

使用原生 yaml 资源安装

直接执行 `kubectl apply` 来安装：

```
    kubectl apply --validate=false -f https://github.com/jetstack/cert-
1. manager/releases/download/v0.11.0/cert-manager.yaml
```

> 使用 `kubectl v1.15` 及其以下的版本需要加上 `--validate=false`，否则会报错。

使用 Helm 安装

安装 CRD：

```
    kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/release-
1. 0.11/deploy/manifests/00-crds.yaml
```

添加 repo：

```
helm repo add jetstack https://charts.jetstack.io
helm repo update
```

执行安装：

```
helm install \
  --name cert-manager \
  --namespace cert-manager \
  --version v0.11.0 \
  jetstack/cert-manager
```

校验是否安装成功

检查 cert-manager 相关的 pod 是否启动成功：

```
$ kubectl get pods --namespace cert-manager

NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-cainjector-74bb68d67c-4vplr   1/1     Running   0          101s
cert-manager-f7f8bf74d-j4hcz               1/1     Running   0          101s
cert-manager-webhook-665df569b5-5p6x8      1/1     Running   0          101s
```

使用 cert-manager 自动生成证书

确保 `cert-manager` 已安装，参考 安装 cert-manager

利用 Let’s Encrypt 生成免费证书

免费证书颁发原理

Let’s Encrypt 利用 ACME 协议来校验域名是否真的属于你，校验成功后就可以自动颁发免费证书，证书有效期只有 90 天，在到期前需要再校验一次来实现续期，幸运的是 cert-manager 可以自动续期，这样就可以使用永久免费的证书了。如何校验你对这个域名属于你呢？主流的两种校验方式是 HTTP-01 和 DNS-01，下面简单介绍下校验原理：

HTTP-01 校验原理

HTTP-01 的校验原理是给你域名指向的 HTTP 服务增加一个临时 location ， `Let’s Encrypt` 会发送 http 请求到 `http://<YOUR_DOMAIN>/.well-known/acme-challenge/<TOKEN>` ， `YOUR_DOMAIN` 就是被校验的域名， `TOKEN` 是 ACME 协议的客户端负责放置的文件，在这里 ACME 客户端就是 cert-manager，它通过修改 Ingress 规则来增加这个临时校验路径并指向提供 `TOKEN` 的服务。 `Let’s Encrypt` 会对比 `TOKEN` 是否符合预期，校验成功后就会颁发证书。此方法仅适用于给使用 Ingress 暴露流量的服务颁发证书，并且不支持泛域名证书。

DNS-01 校验原理

DNS-01 的校验原理是利用 DNS 提供商的 API Key 拿到你的 DNS 控制权限， 在 Let’s Encrypt 为 ACME 客户端提供令牌后，ACME 客户端 (cert-manager) 将创建从该令牌和您的帐户密钥派生的 TXT 记录，并将该记录放在 `_acme-challenge.<YOUR_DOMAIN>` 。 然后 Let’s Encrypt 将向 DNS 系统查询该记录，如果找到匹配项，就可以颁发证书。此方法不需要你的服务使用 Ingress，并且支持泛域名证书。

创建 Issuer/ClusterIssuer

我们需要先创建一个用于签发证书的 Issuer 或 ClusterIssuer，它们唯一区别就是 Issuer 只能用来签发自己所在 namespace 下的证书，ClusterIssuer 可以签发任意 namespace 下的证书，除了名称不同之外，两者所有字段完全一致，下面给出一些示例，简单起见，我们仅以 ClusterIssuer 为例。

创建使用 DNS-01 校验的 ClusterIssuer

假设域名是用 `cloudflare` 管理的，先登录 `cloudflare` 拿到 API Key，然后创建一个 Secret：

```
kubectl -n cert-manager create secret generic cloudflare-apikey --from-literal=apikey=213807bd0fb1ca59bba24a58eac90492e6287
```

> 由于 `ClusterIssuer` 是 NonNamespaced 类型的资源，不在任何命名空间，它需要引用 Secret，而 Secret 必须存在某个命名空间下，所以就规定 `ClusterIssuer` 引用的 Secret 要与 cert-manager 在同一个命名空间下。

创建 DNS-01 方式校验的 `ClusterIssuer`：

```
cat <<EOF | kubectl apply -f -
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-dns01
spec:
  acme:
    # The ACME server URL
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: roc@imroc.io
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-dns01
    solvers:
    - selector: {} # An empty 'selector' means that this solver matches all domains
      dns01: # ACME DNS-01 solver configurations
        cloudflare:
          email: roc@imroc.io
          # A secretKeyRef to a cloudflare api key
          apiKeySecretRef:
            name: cloudflare-apikey
            key: apikey
EOF
```

- `metadata.name`：是我们创建的签发机构的名称，后面我们创建证书的时候会引用它
- `acme.email`：是你自己的邮箱，证书快过期的时候会有邮件提醒，不过 cert-manager 会利用 acme 协议自动给我们重新颁发证书来续期
- `acme.server`：是 acme 协议的服务端，我们这里用 Let’s Encrypt，这个地址就写死成

这样就行

- `acme.privateKeySecretRef` 指示此签发机构的私钥将要存储到哪个 Secret 中，在 cert-manager 所在命名空间
- `solvers.dns01`：配置 DNS-01 校验方式所需的参数，最重要的是 API Key (引用提前创建好的 Secret)，不同 DNS 提供商配置不一样，具体参考官方API文档
- 更多字段参考 API 文档：https://docs.cert-manager.io/en/latest/reference/api-docs/index.html\#clusterissuer-v1alpha2

创建使用 HTTP-01 校验的 `ClusterIssuer`

使用 HTTP-01 方式校验，ACME 服务端 (Let’s Encrypt) 会向客户端 (cert-manager) 提供令牌，客户端会在 web server 上特定路径上放置一个文件，该文件包含令牌以及帐户密钥的指纹。ACME 服务端会请求该路径并校验文件内容，校验成功后就会签发免费证书，更多细节参考：https://letsencrypt.org/zh-cn/docs/challenge-types/

有个问题，ACME 服务端通过什么地址去访问 ACME 客户端的 web server 校验域名？答案是通过将被签发的证书中的域名来访问。这个机制带来的问题是：

1. 不能签发泛域名证书，因为如果是泛域名，没有具体域名，ACME 服务端就不能知道该用什么地址访问 web server 去校验文件。
2. 域名需要提前在 DNS 提供商配置好，这样 ACME 服务端通过域名请求时解析到正确 IP 才能访问成功，也就是需要提前知道你的 web server 的 IP 是什么。

cert-manager 作为 ACME 客户端，它将这个要被 ACME 服务端校验的文件通过 Ingress 来暴露，我们需要提前知道 Ingress 对外的 IP 地址是多少，这样才好配置域名。

一些云厂商自带的 ingress controller 会给每个 Ingress 都创建一个外部地址 (通常对应一个负载均衡器)，这个时候我们需要提前创建好一个 Ingress，拿到外部 IP 并配置域名到此 IP，ACME 客户端 (cert-manager) 修改此 Ingress 的 rules，临时增加一个路径指向 cert-manager 提供的文件，ACME 服务端请求这个域名+指定路径，根据 Ingress 规则转发会返回 cert-manger 提供的这个文件，最终 ACME 服务端 (Let’s Encrypt) 校验该文件，通过后签发免费证书。

指定 Ingress 的创建 `ClusterIssuer` 的示例：

```
cat <<EOF | kubectl apply -f -
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-http01
spec:
```

```
  acme:
    # The ACME server URL
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: roc@imroc.io
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-http01
    solvers:
    - selector: {} # An empty 'selector' means that this solver matches all
domains
      http01: # ACME HTTP-01 solver configurations
        ingress:
          name: challenge
EOF
```

- `solvers.http01`：配置 HTTP-01 校验方式所需的参数，`ingress.name` 指定提前创建好的 ingress 名称

有些自己安装的 ingress controller，所有具有相同 ingress class 的 ingress 都共用一个流量入口，通常是用 LoadBalancer 类型的 Service 暴露 ingress controller，这些具有相同 ingress class 的 ingress 的外部 IP 都是这个 Service 的外部 IP。这种情况我们创建 `ClusterIssuer` 时可以指定 ingress class，校验证书时，cert-manager 会直接创建新的 Ingress 资源并指定 `kubernetes.io/ingress.class` 这个 annotation。

指定 ingress class 的创建 `ClusterIssuer` 的示例：

```
cat <<EOF | kubectl apply -f -
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-http01
spec:
  acme:
    # The ACME server URL
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: roc@imroc.io
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-http01
    solvers:
```

```
      - selector: {} # An empty 'selector' means that this solver matches all
domains
      http01: # ACME HTTP-01 solver configurations
        ingress:
          class: nginx
EOF
```

- `solvers.http01`：配置 HTTP-01 校验方式所需的参数，`ingress.class` 指定 ingress class 名称

创建证书（Certificate）

有了 Issuer/ClusterIssuer，接下来我们就可以生成免费证书了，cert-manager 给我们提供了 Certificate 这个用于生成证书的自定义资源对象，它必须局限在某一个 namespace 下，证书最终会在这个 namespace 下以 Secret 的资源对象存储，假如我想在 dashboard 这个 namespace 下生成免费证书（这个 namespace 已存在），创建一个 Certificate 资源来为我们自动生成证书，示例：

```
cat <<EOF | kubectl apply -f -
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: dashboard-imroc-io
  namespace: kubernetes-dashboard
spec:
  secretName: dashboard-imroc-io-tls
  issuerRef:
    name: letsencrypt-dns01
    kind: ClusterIssuer
  dnsNames:
  - dashboard.imroc.io
EOF
```

- `secretName`：指示证书最终存到哪个 Secret 中
- `issuerRef.kind`：ClusterIssuer 或 Issuer，ClusterIssuer 可以被任意 namespace 的 Certificate 引用，Issuer 只能被当前 namespace 的 Certificate 引用。
- `issuerRef.name`：引用我们创建的 Issuer/ClusterIssuer 的名称
- `commonName`：对应证书的 common name 字段
- `dnsNames`：对应证书的 Subject Alternative Names (SANs) 字段

检查结果

创建完成等待一段时间，校验成功颁发证书后会将证书信息写入 Certificate 所在命名空间的 `secretName` 指定的 Secret 中，其它应用需要证书就可以直接挂载该 Secret 了。

```
Events:
  Type    Reason              Age   From          Message
  ----    ------              ----  ----          -------
  Normal  Generated           15s   cert-manager  Generated new private key
  Normal  GenerateSelfSigned  15s   cert-manager  Generated temporary self
signed certificate
  Normal  OrderCreated        15s   cert-manager  Created Order resource
"dashboard-imroc-io-780134401"
  Normal  OrderComplete       9s    cert-manager  Order "dashboard-imroc-io-
780134401" completed successfully
  Normal  CertIssued          9s    cert-manager  Certificate issued
successfully
```

看下我们的证书是否成功生成：

```
kubectl -n dashboard get secret kubernetes-dashboard-certs -o yaml
apiVersion: v1
data:
  ca.crt: null
  tls.crt: LS0***0tLQo=
  tls.key: LS0***0tCg==
kind: Secret
metadata:
  annotations:
    certmanager.k8s.io/alt-names: dashboard.imroc.io
    certmanager.k8s.io/certificate-name: dashboard-imroc-io
    certmanager.k8s.io/common-name: dashboard.imroc.io
    certmanager.k8s.io/ip-sans: ""
    certmanager.k8s.io/issuer-kind: ClusterIssuer
    certmanager.k8s.io/issuer-name: letsencrypt-prod
  creationTimestamp: 2019-09-19T13:53:55Z
  labels:
    certmanager.k8s.io/certificate-name: dashboard-imroc-io
  name: kubernetes-dashboard-certs
  namespace: dashboard
  resourceVersion: "5689447213"
  selfLink: /api/v1/namespaces/dashboard/secrets/kubernetes-dashboard-certs
  uid: ebfc4aec-dae4-11e9-89f7-be8690a7fdcf
```

```
24.  type: kubernetes.io/tls
```

- `tls.crt` 就是颁发的证书
- `tls.key` 是证书密钥

将 secret 挂载到需要证书的应用，通常应用也要配置下证书路径。

集群配置管理

- Helm

Helm

安装 Helm

Helm 是 Kubernetes 的包管理器，可以帮我们简化 kubernetes 的操作，一键部署应用。假如你的机器上已经安装了 kubectl 并且能够操作集群，那么你就可以安装 Helm 了。当前最新稳定版是 V2，Helm V3 还未正式发布，下面分别说下安装方法。

安装 Helm V2

执行脚本安装 helm 客户端：

```
$ curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  6737  100  6737    0     0  12491      0 --:--:-- --:--:-- --:--:-- 12475
Downloading https://kubernetes-helm.storage.googleapis.com/helm-v2.9.1-linux-amd64.tar.gz
Preparing to install into /usr/local/bin
helm installed into /usr/local/bin/helm
Run 'helm init' to configure helm.
```

查看客户端版本：

```
$ helm version
Client: &version.Version{SemVer:"v2.9.1", GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}
```

安装 tiller 服务端到 kubernetes 集群：

```
$ helm init
Creating /root/.helm
Creating /root/.helm/repository
Creating /root/.helm/repository/cache
Creating /root/.helm/repository/local
Creating /root/.helm/plugins
Creating /root/.helm/starters
Creating /root/.helm/cache/archive
Creating /root/.helm/repository/repositories.yaml
Adding stable repo with URL: https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
```

```
$HELM_HOME has been configured at /root/.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes
Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow
unauthenticated users' policy.
For more information on securing your installation see:
https://docs.helm.sh/using_helm/#securing-your-helm-installation
Happy Helming!
```

查看 tiller 是否启动成功：

```
$ kubectl get pods --namespace=kube-system | grep tiller
tiller-deploy-dccdb6fd9-2df4r          0/1       ImagePullBackOff   0
14h
```

如果状态是 ImagePullBackOff ，说明是镜像问题，一般是未拉取到镜像（国内机器拉取不到 gcr.io 下的镜像）可以查看下是什么镜像：

```
$ kubectl describe pod tiller-deploy-dccdb6fd9-2df4r --namespace=kube-system
Events:
  Type     Reason   Age                  From               Message
  ----     ------   ----                 ----               -------
  Warning  Failed   36m (x5 over 12h)    kubelet, k8s-node1  Failed to pull
image "gcr.io/kubernetes-helm/tiller:v2.9.1": rpc error: code = Unknown desc =
Get https://gcr.io/v1/_ping: dial tcp 64.233.189.82:443: i/o timeout
  Normal   BackOff  11m (x3221 over 14h)  kubelet, k8s-node1  Back-off pulling
image "gcr.io/kubernetes-helm/tiller:v2.9.1"
  Warning  Failed   6m (x3237 over 14h)   kubelet, k8s-node1  Error:
ImagePullBackOff
  Warning  Failed   1m (x15 over 14h)     kubelet, k8s-node1  Failed to pull
image "gcr.io/kubernetes-helm/tiller:v2.9.1": rpc error: code = Unknown desc =
Get https://gcr.io/v1/_ping: dial tcp 64.233.188.82:443: i/o timeout
```

把这个没拉取到镜像想办法下载到这台机器上。当我们看到状态为 `Running` 说明 tiller 已经成功运行了：

```
$ kubectl get pods -n kube-system | grep tiller
tiller-deploy-dccdb6fd9-2df4r                  1/1       Running   1
41d
```

默认安装的 tiller 权限很小，我们执行下面的脚本给它加最大权限，这样方便我们可以用 helm 部

署应用到任意 namespace 下：

```
kubectl create serviceaccount --namespace=kube-system tiller

kubectl create clusterrolebinding tiller-cluster-rule --clusterrole=cluster-admin --serviceaccount=kube-system:tiller

kubectl patch deploy --namespace=kube-system tiller-deploy -p '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'
```

更多参考官方文档：https://helm.sh/docs/using_helm/#quickstart-guide

安装 Helm V3

在 https://github.com/helm/helm/releases 找到对应系统的二进制包下载，比如下载 `v3.0.0-beta.3` 的 `linux amd64` 版：

```
$ wget https://get.helm.sh/helm-v3.0.0-beta.3-linux-amd64.tar.gz
```

解压并移动到 `PATH` 下面：

```
$ tar -zxvf helm-v3.0.0-beta.3-linux-amd64.tar.gz
linux-amd64/
linux-amd64/LICENSE
linux-amd64/helm
linux-amd64/README.md
$ cd linux-amd64/
$ ls
LICENSE  README.md  helm
$ mv helm /usr/local/bin/helm3
```

Helm V2 迁移到 V3

Helm V3 与 V2 版本架构变化较大，数据迁移比较麻烦，官方提供了一个名为 `helm-2to3` 的插件来简化迁移工作，本文将介绍如何利用此插件迁移 Helm V2 到 V3 版本。这里前提是 Helm V3 已安装，安装方法请参考 这里。

安装 2to3 插件

一键安装：

```
$ helm3 plugin install https://github.com/helm/helm-2to3
Downloading and installing helm-2to3 v0.1.1 ...
https://github.com/helm/helm-2to3/releases/download/v0.1.1/helm-2to3_0.1.1_linux_
Installed plugin: 2to3
```

检查插件是否安装成功：

```
$ helm3 plugin list
NAME    VERSION    DESCRIPTION
2to3    0.1.1      migrate Helm v2 configuration and releases in-place to Helm
v3
```

迁移 Helm V2 配置

```
$ helm3 2to3 move config
[Helm 2] Home directory: /root/.helm
[Helm 3] Config directory: /root/.config/helm
[Helm 3] Data directory: /root/.local/share/helm
[Helm 3] Create config folder "/root/.config/helm" .
[Helm 3] Config folder "/root/.config/helm" created.
[Helm 2] repositories file "/root/.helm/repository/repositories.yaml" will copy
to [Helm 3] config folder "/root/.config/helm/repositories.yaml" .
[Helm 2] repositories file "/root/.helm/repository/repositories.yaml" copied
successfully to [Helm 3] config folder "/root/.config/helm/repositories.yaml" .
[Helm 3] Create data folder "/root/.local/share/helm" .
[Helm 3] data folder "/root/.local/share/helm" created.
[Helm 2] plugins "/root/.helm/plugins" will copy to [Helm 3] data folder
"/root/.local/share/helm/plugins" .
```

```
    [Helm 2] plugins "/root/.helm/plugins" copied successfully to [Helm 3] data
12. folder "/root/.local/share/helm/plugins" .
    [Helm 2] starters "/root/.helm/starters" will copy to [Helm 3] data folder
13. "/root/.local/share/helm/starters" .
    [Helm 2] starters "/root/.helm/starters" copied successfully to [Helm 3] data
14. folder "/root/.local/share/helm/starters" .
```

上面的操作主要是迁移：

- Chart 仓库
- Helm 插件
- Chart starters

检查下 repo 和 plugin：

```
1.  $ helm3 repo list
2.  NAME      URL
3.  stable    https://kubernetes-charts.storage.googleapis.com
4.  local     http://127.0.0.1:8879/charts
5.  $
6.  $
7.  $ helm3 plugin list
8.  NAME    VERSION    DESCRIPTION
    2to3    0.1.1      migrate Helm v2 configuration and releases in-place to Helm
9.  v3
10. push    0.1.1      Push chart package to TencentHub
```

迁移 Heml V2 Release

已经用 Helm V2 部署的应用也可以使用 `2to3` 的 `convert` 子命令迁移到 V3，先看下有哪些选项：

```
1.  $ helm3 2to3 convert --help
2.  migrate Helm v2 release in-place to Helm v3
3.
4.  Usage:
5.    2to3 convert [flags] RELEASE
6.
7.  Flags:
        --delete-v2-releases       v2 releases are deleted after migration. By
8.  default, the v2 releases are retained
9.      --dry-run                  simulate a convert
```

```
10.   -h, --help                    help for convert
      -l, --label string            label to select tiller resources by (default
11. "OWNER=TILLER")
      -s, --release-storage string   v2 release storage type/object. It can be
    'secrets' or 'configmaps'. This is only used with the 'tiller-out-cluster' flag
12. (default "secrets")
13.   -t, --tiller-ns string         namespace of Tiller (default "kube-system")
          --tiller-out-cluster       when  Tiller is not running in the cluster
14. e.g. Tillerless
```

- `--tiller-out-cluster`：如果你的 Helm V2 是 tiller 在集群外面（tillerless）的安装方式，请带上这个参数
- `--dry-run`：模拟迁移但不做真实迁移操作，建议每次迁移都先带上这个参数测试下效果，没问题的话再去掉这个参数做真实迁移
- `--tiller-ns`：通常 tiller 如果部署在集群中，并且不在 `kube-system` 命名空间才指定

看下目前有哪些 helm v2 的 release：

```
1. $ helm ls
   NAME     REVISION    UPDATED                     STATUS      CHART          APP
2. VERSION    NAMESPACE
   redis    1           Mon Sep 16 14:46:58 2019    DEPLOYED    redis-9.1.3
3. 5.0.5          default
```

选一个用 `--dry-run` 试下效果：

```
1. $ helm3 2to3 convert redis --dry-run
2. NOTE: This is in dry-run mode, the following actions will not be executed.
3. Run without --dry-run to take the actions described below:
4.
5. Release "redis" will be converted from Helm 2 to Helm 3.
6. [Helm 3] Release "redis" will be created.
7. [Helm 3] ReleaseVersion "redis.v1" will be created.
```

没有报错，去掉 `--dry-run` 执行迁移：

```
1. $ helm3 2to3 convert redis
2. Release "redis" will be converted from Helm 2 to Helm 3.
3. [Helm 3] Release "redis" will be created.
4. [Helm 3] ReleaseVersion "redis.v1" will be created.
5. [Helm 3] ReleaseVersion "redis.v1" created.
```

```
[Helm 3] Release "redis" created.
Release "redis" was converted successfully from Helm 2 to Helm 3. Note: the v2 releases still remain and should be removed to avoid conflicts with the migrated v3 releases.
```

检查迁移结果：

```
$ helm ls
NAME    REVISION    UPDATED                     STATUS      CHART           APP VERSION    NAMESPACE
redis   1           Mon Sep 16 14:46:58 2019    DEPLOYED    redis-9.1.3     5.0.5          default
$
$
$ helm3 ls -a
NAME    NAMESPACE    REVISION    UPDATED                                 STATUS      CHART
redis   default      1           2019-09-16 06:46:58.541391356 +0000 UTC deployed    redis-9.1.3
```

- helm 3 的 release 区分了命名空间，带上 `-a` 参数展示所有命名空间的 release

参考资料

- How to migrate from Helm v2 to Helm v3: https://helm.sh/blog/migrate-from-helm-v2-to-helm-v3/

大规模集群优化

Kubernetes 自 v1.6 以来，官方就宣称单集群最大支持 5000 个节点。不过这只是理论上，在具体实践中从 0 到 5000，还是有很长的路要走，需要见招拆招。

官方标准如下：

- 不超过 5000 个节点
- 不超过 150000 个 pod
- 不超过 300000 个容器
- 每个节点不超过 100 个 pod

内核参数调优

```
# max-file 表示系统级别的能够打开的文件句柄的数量， 一般如果遇到文件句柄达到上限时，会碰到
# "Too many open files" 或者 Socket/File: Can’t open so many files 等错误
fs.file-max=1000000

# 配置 arp cache 大小
# 存在于 ARP 高速缓存中的最少层数，如果少于这个数，垃圾收集器将不会运行。缺省值是 128
net.ipv4.neigh.default.gc_thresh1=1024
# 保存在 ARP 高速缓存中的最多的记录软限制。垃圾收集器在开始收集前，允许记录数超过这个数字 5 秒。缺省值是 512
net.ipv4.neigh.default.gc_thresh2=4096
# 保存在 ARP 高速缓存中的最多记录的硬限制，一旦高速缓存中的数目高于此，垃圾收集器将马上运行。缺省值是 1024
net.ipv4.neigh.default.gc_thresh3=8192
# 以上三个参数，当内核维护的 arp 表过于庞大时候，可以考虑优化

# 允许的最大跟踪连接条目，是在内核内存中 netfilter 可以同时处理的“任务”（连接跟踪条目）
net.netfilter.nf_conntrack_max=10485760
net.netfilter.nf_conntrack_tcp_timeout_established=300
# 哈希表大小（只读）（64位系统、8G内存默认 65536，16G翻倍，如此类推）
net.netfilter.nf_conntrack_buckets=655360

# 每个网络接口接收数据包的速率比内核处理这些包的速率快时，允许送到队列的数据包的最大数目
net.core.netdev_max_backlog=10000

# 默认值: 128 指定了每一个 real user ID 可创建的 inotify instatnces 的数量上限
fs.inotify.max_user_instances=524288
```

```
# 默认值: 8192 指定了每个inotify instance相关联的watches的上限
fs.inotify.max_user_watches=524288
```

ETCD 优化

高可用部署

部署一个高可用ETCD集群可以参考官方文档：https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/clustering.md

> 如果是 self-host 方式部署的集群，可以用 etcd-operator 部署 etcd 集群；也可以使用另一个小集群专门部署 etcd（使用 etcd-operator）

提高磁盘 IO 性能

ETCD 对磁盘写入延迟非常敏感，对于负载较重的集群建议磁盘使用 SSD 固态硬盘。可以使用 diskbench 或 fio 测量磁盘实际顺序 IOPS。

提高 ETCD 的磁盘 IO 优先级

由于 ETCD 必须将数据持久保存到磁盘日志文件中，因此来自其他进程的磁盘活动可能会导致增加写入时间，结果导致 ETCD 请求超时和临时 leader 丢失。当给定高磁盘优先级时，ETCD 服务可以稳定地与这些进程一起运行：

```
sudo ionice -c2 -n0 -p $(pgrep etcd)
```

提高存储配额

默认 ETCD 空间配额大小为 2G，超过 2G 将不再写入数据。通过给 ETCD 配置 `--quota-backend-bytes` 参数增大空间配额，最大支持 8G。

分离 events 存储

集群规模大的情况下，集群中包含大量节点和服务，会产生大量的 event，这些 event 将会对 etcd 造成巨大压力并占用大量 etcd 存储空间，为了在大规模集群下提高性能，可以将 events 存储在单独的 ETCD 集群中。

配置 kube-apiserver：

```
    --etcd-servers="http://etcd1:2379,http://etcd2:2379,http://etcd3:2379" --etcd-ser
overrides="/events#http://etcd4:2379,http://etcd5:2379,http://etcd6:2379"
```

减小网络延迟

如果有大量并发客户端请求 ETCD leader 服务，则可能由于网络拥塞而延迟处理 follower 对等请求。在 follower 节点上的发送缓冲区错误消息：

```
dropped MsgProp to 247ae21ff9436b2d since streamMsg's sending buffer is full
dropped MsgAppResp to 247ae21ff9436b2d since streamMsg's sending buffer is full
```

可以通过在客户端提高 ETCD 对等网络流量优先级来解决这些错误。在 Linux 上，可以使用 tc 对对等流量进行优先级排序：

```
$ tc qdisc add dev eth0 root handle 1: prio bands 3
$ tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip sport 2380 0xffff flowid 1:1
$ tc filter add dev eth0 parent 1: protocol ip prio 1 u32 match ip dport 2380 0xffff flowid 1:1
$ tc filter add dev eth0 parent 1: protocol ip prio 2 u32 match ip sport 2379 0xffff flowid 1:1
$ tc filter add dev eth0 parent 1: protocol ip prio 2 u32 match ip dport 2379 0xffff flowid 1:1
```

Master 节点配置优化

GCE 推荐配置：

- 1-5 节点：n1-standard-1
- 6-10 节点：n1-standard-2
- 11-100 节点：n1-standard-4
- 101-250 节点：n1-standard-8
- 251-500 节点：n1-standard-16
- 超过 500 节点：n1-standard-32

AWS 推荐配置：

- 1-5 节点：m3.medium
- 6-10 节点：m3.large
- 11-100 节点：m3.xlarge
- 101-250 节点：m3.2xlarge
- 251-500 节点：c4.4xlarge

- 超过 500 节点：c4.8xlarge

对应 CPU 和内存为：

- 1-5 节点：1vCPU 3.75G内存
- 6-10 节点：2vCPU 7.5G内存
- 11-100 节点：4vCPU 15G内存
- 101-250 节点：8vCPU 30G内存
- 251-500 节点：16vCPU 60G内存
- 超过 500 节点：32vCPU 120G内存

kube-apiserver 优化

高可用

- 方式一：启动多个 kube-apiserver 实例通过外部 LB 做负载均衡。
- 方式二：设置 `--apiserver-count` 和 `--endpoint-reconciler-type`，可使得多个 kube-apiserver 实例加入到 Kubernetes Service 的 endpoints 中，从而实现高可用。

不过由于 TLS 会复用连接，所以上述两种方式都无法做到真正的负载均衡。为了解决这个问题，可以在服务端实现限流器，在请求达到阀值时告知客户端退避或拒绝连接，客户端则配合实现相应负载切换机制。

控制连接数

kube-apiserver 以下两个参数可以控制连接数：

```
    --max-mutating-requests-inflight int          The maximum number of mutating
    requests in flight at a given time. When the server exceeds this, it rejects
1.  requests. Zero for no limit. (default 200)
    --max-requests-inflight int                   The maximum number of non-
    mutating requests in flight at a given time. When the server exceeds this, it
2.  rejects requests. Zero for no limit. (default 400)
```

节点数量在 1000 - 3000 之间时，推荐：

```
1.  --max-requests-inflight=1500
2.  --max-mutating-requests-inflight=500
```

节点数量大于 3000 时，推荐：

```
--max-requests-inflight=3000
--max-mutating-requests-inflight=1000
```

kube-scheduler 与 kube-controller-manager 优化

高可用

kube-controller-manager 和 kube-scheduler 是通过 leader election 实现高可用，启用时需要添加以下参数：

```
--leader-elect=true
--leader-elect-lease-duration=15s
--leader-elect-renew-deadline=10s
--leader-elect-resource-lock=endpoints
--leader-elect-retry-period=2s
```

控制 QPS

与 kube-apiserver 通信的 qps 限制，推荐为：

```
--kube-api-qps=100
```

集群 DNS 高可用

设置反亲和，让集群 DNS (kube-dns 或 coredns) 分散在不同节点，避免单点故障：

```
affinity:
 podAntiAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
   - weight: 100
     labelSelector:
       matchExpressions:
       - key: k8s-app
         operator: In
         values:
         - kube-dns
     topologyKey: kubernetes.io/hostname
```

问题排查

Pod 排错

本文是本书排查指南板块下问题排查章节的 Pod排错 一节，介绍 Pod 各种异常现象，可能的原因以及解决方法。

常用命令

排查过程常用的命名如下：

- 查看 Pod 状态：`kubectl get pod <pod-name> -o wide`
- 查看 Pod 的 yaml 配置：`kubectl get pod <pod-name> -o yaml`
- 查看 Pod 事件：`kubectl describe pod <pod-name>`
- 查看容器日志：`kubectl logs <pod-name> [-c <container-name>]`

Pod 状态

Pod 有多种状态，这里罗列一下：

- `Error`：Pod 启动过程中发生错误
- `NodeLost`：Pod 所在节点失联
- `Unkown`：Pod 所在节点失联或其它未知异常
- `Waiting`：Pod 等待启动
- `Pending`：Pod 等待被调度
- `ContainerCreating`：Pod 容器正在被创建
- `Terminating`：Pod 正在被销毁
- `CrashLoopBackOff`：容器退出，kubelet 正在将它重启
- `InvalidImageName`：无法解析镜像名称
- `ImageInspectError`：无法校验镜像
- `ErrImageNeverPull`：策略禁止拉取镜像
- `ImagePullBackOff`：正在重试拉取
- `RegistryUnavailable`：连接不到镜像中心
- `ErrImagePull`：通用的拉取镜像出错
- `CreateContainerConfigError`：不能创建 kubelet 使用的容器配置
- `CreateContainerError`：创建容器失败
- `RunContainerError`：启动容器失败
- `PreStartHookError`：执行 preStart hook 报错
- `PostStartHookError`：执行 postStart hook 报错
- `ContainersNotInitialized`：容器没有初始化完毕

- `ContainersNotReady`： 容器没有准备完毕
- `ContainerCreating`：容器创建中
- `PodInitializing`：pod 初始化中
- `DockerDaemonNotReady`：docker还没有完全启动
- `NetworkPluginNotReady`： 网络插件还没有完全启动

问题导航

有时候我们无法直接通过异常状态找到异常原因，这里我们罗列一下各种现象，点击即可进入相应的文章，帮助你分析问题，罗列各种可能的原因，进一步定位根因：

- Pod 一直处于 Pending 状态
- Pod 一直处于 ContainerCreating 或 Waiting 状态
- Pod 一直处于 CrashLoopBackOff 状态
- Pod 一直处于 Terminating 状态
- Pod 一直处于 Unknown 状态
- Pod 一直处于 Error 状态
- Pod 一直处于 ImagePullBackOff 状态
- Pod 一直处于 ImageInspectError 状态
- Pod Terminating 慢
- Pod 健康检查失败
- 容器进程主动退出

> 更多内容还在不断 Loading，如果发发现了更多奇怪现象或相同现象但不同的原因导致的，欢迎一起分享，也可以给本书提 PR，一起完善补充。

Pod 一直处于 Pending 状态

Pending 状态说明 Pod 还没有被调度到某个节点上，需要看下 Pod 事件进一步判断原因，比如：

```
$ kubectl describe pod tikv-0
...
Events:
  Type     Reason            Age                 From               Message
  ----     ------            ----                ----               -------
  Warning  FailedScheduling  3m (x106 over 33m)  default-scheduler  0/4 nodes are available: 1 node(s) had no available volume zone, 2 Insufficient cpu, 3 Insufficient memory.
```

下面列举下可能原因和解决方法。

节点资源不够

节点资源不够有以下几种情况：

- CPU 负载过高
- 剩余可以被分配的内存不够
- 剩余可用 GPU 数量不够（通常在机器学习场景，GPU 集群环境）

如果判断某个 Node 资源是否足够？ 通过 `kubectl describe node <node-name>` 查看 node 资源情况，关注以下信息：

- `Allocatable`：表示此节点能够申请的资源总和
- `Allocated resources`：表示此节点已分配的资源（Allocatable 减去节点上所有 Pod 总的 Request）

前者与后者相减，可得出剩余可申请的资源。如果这个值小于 Pod 的 request，就不满足 Pod 的资源要求，Scheduler 在 Predicates（预选）阶段就会剔除掉这个 Node，也就不会调度上去。

不满足 nodeSelector 与 affinity

如果 Pod 包含 nodeSelector 指定了节点需要包含的 label，调度器将只会考虑将 Pod 调度到包含这些 label 的 Node 上，如果没有 Node 有这些 label 或者有这些 label 的 Node 其它条件不满足也将会无法调度。参考官方文档：https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#nodeselector

如果 Pod 包含 affinity（亲和性）的配置，调度器根据调度算法也可能算出没有满足条件的 Node，从而无法调度。affinity 有以下几类：

- nodeAffinity：节点亲和性，可以看成是增强版的 nodeSelector，用于限制 Pod 只允许被调度到某一部分 Node。
- podAffinity：Pod 亲和性，用于将一些有关联的 Pod 调度到同一个地方，同一个地方可以是指同一个节点或同一个可用区的节点等。
- podAntiAffinity：Pod 反亲和性，用于避免将某一类 Pod 调度到同一个地方避免单点故障，比如将集群 DNS 服务的 Pod 副本都调度到不同节点，避免一个节点挂了造成整个集群 DNS 解析失败，使得业务中断。

Node 存在 Pod 没有容忍的污点

如果节点上存在污点（Taints），而 Pod 没有响应的容忍（Tolerations），Pod 也将不会调度上去。通过 describe node 可以看下 Node 有哪些 Taints：

```
$ kubectl describe nodes host1
...
Taints:             special=true:NoSchedule
...
```

污点既可以是手动添加也可以是被自动添加，下面来深入分析一下。

手动添加的污点

通过类似以下方式可以给节点添加污点：

```
$ kubectl taint node host1 special=true:NoSchedule
node "host1" tainted
```

另外，有些场景下希望新加的节点默认不调度 Pod，直到调整完节点上某些配置才允许调度，就给新加的节点都加上 `node.kubernetes.io/unschedulable` 这个污点。

自动添加的污点

如果节点运行状态不正常，污点也可以被自动添加，从 v1.12 开始，`TaintNodesByCondition` 特性进入 Beta 默认开启，controller manager 会检查 Node 的 Condition，如果命中条件就自动为 Node 加上相应的污点，这些 Condition 与 Taints 的对应关系如下：

```
Conditon                Value       Taints
--------                -----       ------
```

```
3.  OutOfDisk             True        node.kubernetes.io/out-of-disk
4.  Ready                 False       node.kubernetes.io/not-ready
5.  Ready                 Unknown     node.kubernetes.io/unreachable
6.  MemoryPressure        True        node.kubernetes.io/memory-pressure
7.  PIDPressure           True        node.kubernetes.io/pid-pressure
8.  DiskPressure          True        node.kubernetes.io/disk-pressure
9.  NetworkUnavailable    True        node.kubernetes.io/network-unavailable
```

解释下上面各种条件的意思：

- OutOfDisk 为 True 表示节点磁盘空间不够了
- Ready 为 False 表示节点不健康
- Ready 为 Unknown 表示节点失联，在 `node-monitor-grace-period` 这么长的时间内没有上报状态 controller-manager 就会将 Node 状态置为 Unknown (默认 40s)
- MemoryPressure 为 True 表示节点内存压力大，实际可用内存很少
- PIDPressure 为 True 表示节点上运行了太多进程，PID 数量不够用了
- DiskPressure 为 True 表示节点上的磁盘可用空间太少了
- NetworkUnavailable 为 True 表示节点上的网络没有正确配置，无法跟其它 Pod 正常通信

另外，在云环境下，比如腾讯云 TKE，添加新节点会先给这个 Node 加上 `node.cloudprovider.kubernetes.io/uninitialized` 的污点，等 Node 初始化成功后才自动移除这个污点，避免 Pod 被调度到没初始化好的 Node 上。

低版本 kube-scheduler 的 bug

可能是低版本 `kube-scheduler` 的 bug，可以升级下调度器版本。

kube-scheduler 没有正常运行

检查 maser 上的 `kube-scheduler` 是否运行正常，异常的话可以尝试重启临时恢复。

驱逐后其它可用节点与当前节点有状态应用不在同一个可用区

有时候服务部署成功运行过，但在某个时候节点突然挂了，此时就会触发驱逐，创建新的副本调度到其它节点上，对于已经挂载了磁盘的 Pod，它通常需要被调度到跟当前节点和磁盘在同一个可用区，如果集群中同一个可用区的节点不满足调度条件，即使其它可用区节点各种条件都满足，但不跟当前节点在同一个可用区，也是不会调度的。为什么需要限制挂载了磁盘的 Pod 不能漂移到其它可用区的节点？试想一下，云上的磁盘虽然可以被动态挂载到不同机器，但也只是相对同一个数据中心，通常不允许跨

数据中心挂载磁盘设备，因为网络时延会极大的降低 IO 速率。

Pod 一直处于 ContainerCreating 或 Waiting 状态

Pod 配置错误

- 检查是否打包了正确的镜像
- 检查配置了正确的容器参数

挂载 Volume 失败

Volume 挂载失败也分许多种情况，先列下我这里目前已知的。

Pod 漂移没有正常解挂之前的磁盘

在云尝试托管的 K8S 服务环境下，默认挂载的 Volume 一般是块存储类型的云硬盘，如果某个节点挂了，kubelet 无法正常运行或与 apiserver 通信，到达时间阀值后会触发驱逐，自动在其它节点上启动相同的副本 (Pod 漂移)，但是由于被驱逐的 Node 无法正常运行并不知道自己被驱逐了，也就没有正常执行解挂，cloud-controller-manager 也在等解挂成功后再调用云厂商的接口将磁盘真正从节点上解挂，通常会等到一个时间阀值后 cloud-controller-manager 会强制解挂云盘，然后再将其挂载到 Pod 最新所在节点上，这种情况下 ContainerCreating 的时间相对长一点，但一般最终是可以启动成功的，除非云厂商的 cloud-controller-manager 逻辑有 bug。

命中 K8S 挂载 configmap/secret 的 subpath 的 bug

最近发现如果 Pod 挂载了 configmap 或 secret， 如果后面修改了 configmap 或 secret 的内容，Pod 里的容器又原地重启了(比如存活检查失败被 kill 然后重启拉起)，就会触发 K8S 的这个 bug，团队的小伙伴已提 PR: https://github.com/kubernetes/kubernetes/pull/82784

如果是这种情况，容器会一直启动不成功，可以看到类似以下的报错:

```
$ kubectl -n prod get pod -o yaml manage-5bd487cf9d-bqmvm
...
lastState: terminated
containerID:
containerd://e6746201faa1dfe7f3251b8c30d59ebf613d99715f3b800740e587e681d2a903
exitCode: 128
finishedAt: 2019-09-15T00:47:22Z
```

```
    message: 'failed to create containerd task: OCI runtime create failed:
 7. container_linux.go:345:
 8. starting container process caused "process_linux.go:424: container init
    caused \"rootfs_linux.go:58: mounting \\\"/var/lib/kubelet/pods/211d53f4-d08c-
 9. 11e9-b0a7-b6655eaf02a6/volume-subpaths/manage-config-volume/manage/0\\\"
    to rootfs
10. \\\"/run/containerd/io.containerd.runtime.v1.linux/k8s.io/e6746201faa1dfe7f3251b8
    at
11. \\\"/run/containerd/io.containerd.runtime.v1.linux/k8s.io/e6746201faa1dfe7f3251b8
12. caused \\\"no such file or directory\\\"\"": unknown'
```

磁盘爆满

启动 Pod 会调 CRI 接口创建容器，容器运行时创建容器时通常会在数据目录下为新建的容器创建一些目录和文件，如果数据目录所在的磁盘空间满了就会创建失败并报错：

```
1. Events:
     Type     Reason                  Age                  From
2. Message
     ----     ------                  ----                 ----
3. -------
     Warning  FailedCreatePodSandBox  2m (x4307 over 16h)  kubelet, 10.179.80.31  (c
    similar events): Failed create pod sandbox: rpc error: code = Unknown desc = fail
    sandbox for pod "apigateway-6dc48bf8b6-l8xrw": Error response from daemon: mkdir
    /var/lib/docker/aufs/mnt/1f09d6c1c9f24e8daaea5bf33a4230de7dbc758e3b22785e8ee21e3e
4. no space left on device
```

解决方法参考本书 处理实践：磁盘爆满

节点内存碎片化

如果节点上内存碎片化严重，缺少大页内存，会导致即使总的剩余内存较多，但还是会申请内存失败，参考 处理实践： 内存碎片化

limit 设置太小或者单位不对

如果 limit 设置过小以至于不足以成功运行 Sandbox 也会造成这种状态，常见的是因为 memory limit 单位设置不对造成的 limit 过小，比如误将 memory 的 limit 单位像 request 一样设置为小 `m` ，这个单位在 memory 不适用，会被 k8s 识别成 byte， 应该用 `Mi` 或

M 。，

举个例子：如果 memory limit 设为 1024m 表示限制 1.024 Byte，这么小的内存， pause 容器一起来就会被 cgroup-oom kill 掉，导致 pod 状态一直处于 ContainerCreating。

这种情况通常会报下面的 event：

```
1.  Pod sandbox changed, it will be killed and re-created。
```

kubelet 报错：

```
    to start sandbox container for pod ... Error response from daemon: OCI runtime
    create failed: container_linux.go:348: starting container process caused
    "process_linux.go:301: running exec setns process for init caused \"signal:
1.  killed\"": unknown
```

拉取镜像失败

镜像拉取失败也分很多情况，这里列举下：

- 配置了错误的镜像
- Kubelet 无法访问镜像仓库（比如默认 pause 镜像在 gcr.io 上，国内环境访问需要特殊处理）
- 拉取私有镜像的 imagePullSecret 没有配置或配置有误
- 镜像太大，拉取超时（可以适当调整 kubelet 的 –image-pull-progress-deadline 和 –runtime-request-timeout 选项）

CNI 网络错误

如果发生 CNI 网络错误通常需要检查下网络插件的配置和运行状态，如果没有正确配置或正常运行通常表现为：

- 无法配置 Pod 网络
- 无法分配 Pod IP

controller-manager 异常

查看 master 上 kube-controller-manager 状态，异常的话尝试重启。

安装 docker 没删干净旧版本

如果节点上本身有 docker 或者没删干净，然后又安装 docker，比如在 centos 上用 yum 安装：

```
yum install -y docker
```

这样可能会导致 dockerd 创建容器一直不成功，从而 Pod 状态一直 ContainerCreating，查看 event 报错：

```
  Type     Reason                  Age                     From
Message
  ----     ------                  ----                    ----
-------
  Warning  FailedCreatePodSandBox  18m (x3583 over 83m)    kubelet, 192.168.4.5
(combined from similar events): Failed create pod sandbox: rpc error: code = Unkn
desc = failed to start sandbox container for pod "nginx-7db9fccd9b-2j6dh": Error
response from daemon: ttrpc: client shutting down: read unix @->@/containerd-
shim/moby/de2bfeefc999af42783115acca62745e6798981dff75f4148fae8c086668f667/shim.s
read: connection reset by peer: unknown
  Normal   SandboxChanged          3m12s (x4420 over 83m)  kubelet, 192.168.4.5
Pod sandbox changed, it will be killed and re-created.
```

可能是因为重复安装 docker 版本不一致导致一些组件之间不兼容，从而导致 dockerd 无法正常创建容器。

存在同名容器

如果节点上已有同名容器，创建 sandbox 就会失败，event：

```
  Warning  FailedCreatePodSandBox  2m                kubelet, 10.205.8.91
Failed create pod sandbox: rpc error: code = Unknown desc = failed to create a
sandbox for pod "lomp-ext-d8c8b8c46-4v8tl": operation timeout: context deadline
exceeded
  Warning  FailedCreatePodSandBox  3s (x12 over 2m)  kubelet, 10.205.8.91
Failed create pod sandbox: rpc error: code = Unknown desc = failed to create a
sandbox for pod "lomp-ext-d8c8b8c46-4v8tl": Error response from daemon:
Conflict. The container name "/k8s_POD_lomp-ext-d8c8b8c46-
4v8tl_default_65046a06-f795-11e9-9bb6-b67fb7a70bad_0" is already in use by
container "30aa3f5847e0ce89e9d411e76783ba14accba7eb7743e605a10a9a862a72c1e2".
You have to remove (or rename) that container to be able to reuse that name.
```

关于什么情况下会产生同名容器，这个有待研究。

Pod 处于 CrashLoopBackOff 状态

Pod 如果处于 `CrashLoopBackOff` 状态说明之前是启动了，只是又异常退出了，只要 Pod 的 `restartPolicy` 不是 Never 就可能被重启拉起，此时 Pod 的 `RestartCounts` 通常是大于 0 的，可以先看下容器进程的退出状态码来缩小问题范围，参考本书 排错技巧：分析 ExitCode 定位 Pod 异常退出原因

容器进程主动退出

如果是容器进程主动退出，退出状态码一般在 0-128 之间，除了可能是业务程序 BUG，还有其它许多可能原因，参考：容器进程主动退出

系统 OOM

如果发生系统 OOM，可以看到 Pod 中容器退出状态码是 137，表示被 `SIGKILL` 信号杀死，同时内核会报错：`Out of memory: Kill process ...` 。大概率是节点上部署了其它非 K8S 管理的进程消耗了比较多的内存，或者 kubelet 的 `--kube-reserved` 和 `--system-reserved` 配的比较小，没有预留足够的空间给其它非容器进程，节点上所有 Pod 的实际内存占用总量不会超过 `/sys/fs/cgroup/memory/kubepods` 这里 cgroup 的限制，这个限制等于 `capacity - "kube-reserved" - "system-reserved"` ，如果预留空间设置合理，节点上其它非容器进程（kubelet, dockerd, kube-proxy, sshd 等）内存占用没有超过 kubelet 配置的预留空间是不会发生系统 OOM 的，可以根据实际需求做合理的调整。

cgroup OOM

如果是 cgrou OOM 杀掉的进程，从 Pod 事件的下 `Reason` 可以看到是 `OOMKilled` ，说明容器实际占用的内存超过 limit 了，同时内核日志会报：``。 可以根据需求调整下 limit。

节点内存碎片化

如果节点上内存碎片化严重，缺少大页内存，会导致即使总的剩余内存较多，但还是会申请内存失败，参考 处理实践：内存碎片化

健康检查失败

参考 Pod 健康检查失败 进一步定位。

Pod 一直处于 Terminating 状态

磁盘爆满

如果 docker 的数据目录所在磁盘被写满，docker 无法正常运行，无法进行删除和创建操作，所以 kubelet 调用 docker 删除容器没反应，看 event 类似这样：

```
    Normal  Killing  39s (x735 over 15h)  kubelet, 10.179.80.31  Killing container
1.  with id docker://apigateway:Need to kill Pod
```

处理建议是参考本书 处理实践：磁盘爆满

存在 “i” 文件属性

如果容器的镜像本身或者容器启动后写入的文件存在 “i” 文件属性，此文件就无法被修改删除，而删除 Pod 时会清理容器目录，但里面包含有不可删除的文件，就一直删不了，Pod 状态也将一直保持 Terminating，kubelet 报错：

```
    Sep 27 14:37:21 VM_0_7_centos kubelet[14109]: E0927 14:37:21.922965   14109 remot
    RemoveContainer "19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257
    failed: rpc error: code = Unknown desc = failed to remove container
    "19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257": Error respons
    19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257: driver "overlay
    filesystem: remove
    /data/docker/overlay2/b1aea29c590aa9abda79f7cf3976422073fb3652757f0391db885340275
1.  operation not permitted
    Sep 27 14:37:21 VM_0_7_centos kubelet[14109]: E0927 14:37:21.923027   14109 kuber
    to remove container "19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df3
    Unknown desc = failed to remove container
    "19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257": Error respons
    19d837c77a3c294052a99ff9347c520bc8acb7b8b9a9dc9fab281fc09df38257: driver "overlay
    filesystem: remove
    /data/docker/overlay2/b1aea29c590aa9abda79f7cf3976422073fb3652757f0391db885340275
2.  operation not permitted
```

通过 `man chattr` 查看 “i” 文件属性描述：

```
          A file with the 'i' attribute cannot be modified: it cannot be deleted
1.  or renamed, no
```

```
    link can be created to this file and no data can be written to the file.  Only
2.  the superuser
    or a process possessing the CAP_LINUX_IMMUTABLE capability can set or clear
3.  this attribute.
```

彻底解决当然是不要在容器镜像中或启动后的容器设置 “i” 文件属性，临时恢复方法： 复制 kubelet 日志报错提示的文件路径，然后执行 `chattr -i <file>` ：

```
    chattr -i
1.  /data/docker/overlay2/b1aea29c590aa9abda79f7cf3976422073fb3652757f0391db885340275
```

执行完后等待 kubelet 自动重试，Pod 就可以被自动删除了。

docker 17 的 bug

docker hang 住，没有任何响应，看 event:

```
    Warning FailedSync 3m (x408 over 1h) kubelet, 10.179.80.31 error determining
1.  status: rpc error: code = DeadlineExceeded desc = context deadline exceeded
```

怀疑是17版本dockerd的BUG。可通过 `kubectl -n cn-staging delete pod apigateway-6dc48bf8b6-clcwk --force --grace-period=0` 强制删除pod，但 `docker ps` 仍看得到这个容器

处置建议:

- 升级到docker 18. 该版本使用了新的 containerd，针对很多bug进行了修复。
- 如果出现terminating状态的话，可以提供让容器专家进行排查，不建议直接强行删除，会可能导致一些业务上问题。

存在 Finalizers

k8s 资源的 metadata 里如果存在 `finalizers` ，那么该资源一般是由某程序创建的，并且在其创建的资源的 metadata 里的 `finalizers` 加了一个它的标识，这意味着这个资源被删除时需要由创建资源的程序来做删除前的清理，清理完了它需要将标识从该资源的 `finalizers` 中移除，然后才会最终彻底删除资源。比如 Rancher 创建的一些资源就会写入 `finalizers` 标识。

处理建议： `kubectl edit` 手动编辑资源定义，删掉 `finalizers` ，这时再看下资源，就会发现已经删掉了

低版本 kubelet list-watch 的 bug

之前遇到过使用 v1.8.13 版本的 k8s，kubelet 有时 list-watch 出问题，删除 pod 后 kubelet 没收到事件，导致 kubelet 一直没做删除操作，所以 pod 状态一直是 Terminating

dockerd 与 containerd 的状态不同步

判断 dockerd 与 containerd 某个容器的状态不同步的方法：

- describe pod 拿到容器 id
- docker ps 查看的容器状态是 dockerd 中保存的状态
- 通过 docker-container-ctr 查看容器在 containerd 中的状态，比如：

```
$ docker-container-ctr --namespace moby --address
/var/run/docker/containerd/docker-containerd.sock task ls |grep
a9a1785b81343c3ad2093ad973f4f8e52dbf54823b8bb089886c8356d4036fe0
a9a1785b81343c3ad2093ad973f4f8e52dbf54823b8bb089886c8356d4036fe0    30639
STOPPED
```

containerd 看容器状态是 stopped 或者已经没有记录，而 docker 看容器状态却是 runing，说明 dockerd 与 containerd 之间容器状态同步有问题，目前发现了 docker 在 aufs 存储驱动下如果磁盘爆满可能发生内核 panic ：

```
aufs au_opts_verify:1597:dockerd[5347]: dirperm1 breaks the protection by the
permission bits on the lower branch
```

如果磁盘爆满过，dockerd 一般会有下面类似的日志：

```
Sep 18 10:19:49 VM-1-33-ubuntu dockerd[4822]: time="2019-09-18T10:19:49.903943652
write
/opt/docker/containers/54922ec8b1863bcc504f6dac41e40139047f7a84ff09175d2800100aac
json.log: no space left on device"
```

随后可能发生状态不同步，已提issue：https://github.com/docker/for-linux/issues/779

- 临时恢复：执行 `docker prune` 或重启 dockerd
- 长期方案：运行时推荐直接使用 containerd，绕过 dockerd 避免 docker 本身的各种 BUG

Daemonset Controller 的 BUG

有个 k8s 的 bug 会导致 daemonset pod 无限 terminating，1.10 和 1.11 版本受影响，原因是 daemonset controller 复用 scheduler 的 predicates 逻辑，里面将 nodeAffinity 的 nodeSelector 数组做了排序（传的指针），spec 就会跟 apiserver 中的不一致，daemonset controller 又会为 rollingUpdate类型计算 hash（会用到spec），用于版本控制，造成不一致从而无限启动和停止的循环。

- issue: https://github.com/kubernetes/kubernetes/issues/66298
- 修复的PR: https://github.com/kubernetes/kubernetes/pull/66480

升级集群版本可以彻底解决，临时规避可以给 rollingUpdate 类型 daemonset 不使用 nodeAffinity，改用 nodeSelector。

Pod 一直处于 Unknown 状态

TODO: 完善

通常是节点失联，没有上报状态给 apiserver，到达阀值后 controller-manager 认为节点失联并将其状态置为 `Unknown` 。

可能原因:

- 节点高负载导致无法上报
- 节点宕机
- 节点被关机
- 网络不通

Pod 一直处于 Error 状态

TODO: 展开优化

通常处于 Error 状态说明 Pod 启动过程中发生了错误。常见的原因包括:

- 依赖的 ConfigMap、Secret 或者 PV 等不存在
- 请求的资源超过了管理员设置的限制，比如超过了 LimitRange 等
- 违反集群的安全策略，比如违反了 PodSecurityPolicy 等
- 容器无权操作集群内的资源，比如开启 RBAC 后，需要为 ServiceAccount 配置角色绑定

Pod 一直处于 ImagePullBackOff 状态

http 类型 registry，地址未加入到 insecure-registry

dockerd 默认从 https 类型的 registry 拉取镜像，如果使用 https 类型的 registry，则必须将它添加到 insecure-registry 参数中，然后重启或 reload dockerd 生效。

https 自签发类型 resitry，没有给节点添加 ca 证书

如果 registry 是 https 类型，但证书是自签发的，dockerd 会校验 registry 的证书，校验成功才能正常使用镜像仓库，要想校验成功就需要将 registry 的 ca 证书放置到 `/etc/docker/certs.d/<registry:port>/ca.crt` 位置。

私有镜像仓库认证失败

如果 registry 需要认证，但是 Pod 没有配置 imagePullSecret，配置的 Secret 不存在或者有误都会认证失败。

镜像文件损坏

如果 push 的镜像文件损坏了，下载下来也用不了，需要重新 push 镜像文件。

镜像拉取超时

如果节点上新起的 Pod 太多就会有许多可能会造成容器镜像下载排队，如果前面有许多大镜像需要下载很长时间，后面排队的 Pod 就会报拉取超时。

kubelet 默认串行下载镜像：

```
    --serialize-image-pulls   Pull images one at a time. We recommend *not*
    changing the default value on nodes that run docker daemon with version < 1.9
1.  or an Aufs storage backend. Issue #10959 has more details. (default true)
```

也可以开启并行下载并控制并发：

```
    --registry-qps int32   If > 0, limit registry pull QPS to this value.  If 0,
1.  unlimited. (default 5)
    --registry-burst int32   Maximum size of a bursty pulls, temporarily allows
    pulls to burst to this number, while still not exceeding registry-qps. Only
2.  used if --registry-qps > 0 (default 10)
```

镜像不不存在

kubelet 日志:

```
    PullImage "imroc/test:v0.2" from image service failed: rpc error: code =
    Unknown desc = Error response from daemon: manifest for imroc/test:v0.2 not
1.  found
```

Pod 一直处于 ImageInspectError 状态

通常是镜像文件损坏了，可以尝试删除损坏的镜像重新拉取

TODO: 完善

Pod 健康检查失败

- Kubernetes 健康检查包含就绪检查(readinessProbe)和存活检查(livenessProbe)
- pod 如果就绪检查失败会将此 pod ip 从 service 中摘除，通过 service 访问，流量将不会被转发给就绪检查失败的 pod
- pod 如果存活检查失败，kubelet 将会杀死容器并尝试重启

健康检查失败的可能原因有多种，除了业务程序BUG导致不能响应健康检查导致 unhealthy，还能有有其它原因，下面我们来逐个排查。

健康检查配置不合理

`initialDelaySeconds` 太短，容器启动慢，导致容器还没完全启动就开始探测，如果 successThreshold 是默认值 1，检查失败一次就会被 kill，然后 pod 一直这样被 kill 重启。

节点负载过高

cpu 占用高（比如跑满）会导致进程无法正常发包收包，通常会 timeout，导致 kubelet 认为 pod 不健康。参考本书 处理实践：高负载 一节。

容器进程被木马进程杀死

参考本书 处理实践：使用 systemtap 定位疑难杂症 进一步定位。

容器内进程端口监听挂掉

使用 `netstat -tunlp` 检查端口监听是否还在，如果不在了，抓包可以看到会直接 reset 掉健康检查探测的连接：

```
20:15:17.890996 IP 172.16.2.1.38074 > 172.16.2.23.8888: Flags [S], seq
96880261, win 14600, options [mss 1424,nop,nop,sackOK,nop,wscale 7], length 0
20:15:17.891021 IP 172.16.2.23.8888 > 172.16.2.1.38074: Flags [R.], seq 0, ack
96880262, win 0, length 0
20:15:17.906744 IP 10.0.0.16.54132 > 172.16.2.23.8888: Flags [S], seq
1207014342, win 14600, options [mss 1424,nop,nop,sackOK,nop,wscale 7], length 0
20:15:17.906766 IP 172.16.2.23.8888 > 10.0.0.16.54132: Flags [R.], seq 0, ack
1207014343, win 0, length 0
```

连接异常，从而健康检查失败。发生这种情况的原因可能在一个节点上启动了多个使用 `hostNetwork` 监听相同宿主机端口的 Pod，只会有一个 Pod 监听成功，但监听失败的 Pod 的业务逻辑允许了监听失败，并没有退出，Pod 又配了健康检查，kubelet 就会给 Pod 发送健康检查探测报文，但 Pod 由于没有监听所以就会健康检查失败。

SYN backlog 设置过小

SYN backlog 大小即 SYN 队列大小，如果短时间内新建连接比较多，而 SYN backlog 设置太小，就会导致新建连接失败，通过 `netstat -s | grep TCPBacklogDrop` 可以看到有多少是因为 backlog 满了导致丢弃的新连接。

如果确认是 backlog 满了导致的丢包，建议调高 backlog 的值，内核参数为 `net.ipv4.tcp_max_syn_backlog` 。

容器进程主动退出

容器进程如果是自己主动退出(不是被外界中断杀死)，退出状态码一般在 0-128 之间，根据约定，正常退出时状态码为 0，1-127 说明是程序发生异常，主动退出了，比如检测到启动的参数和条件不满足要求，或者运行过程中发生 panic 但没有捕获处理导致程序退出。除了可能是业务程序 BUG，还有其它许多可能原因，这里我们一一列举下。

DNS 无法解析

可能程序依赖 集群 DNS 服务，比如启动时连接数据库，数据库使用 service 名称或外部域名都需要 DNS 解析，如果解析失败程序将报错并主动退出。解析失败的可能原因：

- 集群网络有问题，Pod 连不上集群 DNS 服务
- 集群 DNS 服务挂了，无法响应解析请求
- Service 或域名地址配置有误，本身是无法解析的地址

程序配置有误

- 配置文件格式错误，程序启动解析配置失败报错退出
- 配置内容不符合规范，比如配置中某个字段是必选但没有填写，配置校验不通过，程序报错主动退出

网络排错

LB 健康检查失败

可能原因:

- 节点防火墙规则没放开 nodeport 区间端口 (默认 30000-32768) 检查iptables和云主机安全组
- LB IP 绑到 `kube-ipvs0` 导致丢源 IP为 LB IP 的包: https://github.com/kubernetes/kubernetes/issues/79783

TODO: 完善

DNS 解析异常

5 秒延时

如果DNS查询经常延时5秒才返回，通常是遇到内核 conntrack 冲突导致的丢包，详见 案例分享：DNS 5秒延时

解析超时

如果容器内报 DNS 解析超时，先检查下集群 DNS 服务（ `kube-dns` / `coredns` ）的 Pod 是否 Ready，如果不是，请参考本章其它小节定位原因。如果运行正常，再具体看下超时现象。

解析外部域名超时

可能原因：

- 上游 DNS 故障
- 上游 DNS 的 ACL 或防火墙拦截了报文

所有解析都超时

如果集群内某个 Pod 不管解析 Service 还是外部域名都失败，通常是 Pod 与集群 DNS 之间通信有问题。

可能原因：

- 节点防火墙没放开集群网段，导致如果 Pod 跟集群 DNS 的 Pod 不在同一个节点就无法通信，DNS 请求也就无法被收到

Service 不通

集群 dns 故障

TODO

节点防火墙没放开集群容器网络 (iptables/安全组)

TODO

kube-proxy 没有工作，命中 netlink deadlock 的 bug

- issue: https://github.com/kubernetes/kubernetes/issues/71071
- 1.14 版本已修复，修复的 PR: https://github.com/kubernetes/kubernetes/pull/72361

Service 无法解析

集群 DNS 没有正常运行(kube-dns或CoreDNS)

检查集群 DNS 是否运行正常:

- kubelet 启动参数 `--cluster-dns` 可以看到 dns 服务的 cluster ip:

```
$ ps -ef | grep kubelet
... /usr/bin/kubelet --cluster-dns=172.16.14.217 ...
```

- 找到 dns 的 service:

```
$ kubectl get svc -n kube-system | grep 172.16.14.217
kube-dns              ClusterIP   172.16.14.217   <none>        53/TCP,53/UDP
47d
```

- 看是否存在 endpoint:

```
$ kubectl -n kube-system describe svc kube-dns | grep -i endpoints
Endpoints:         172.16.0.156:53,172.16.0.167:53
Endpoints:         172.16.0.156:53,172.16.0.167:53
```

- 检查 endpoint 的 对应 pod 是否正常:

```
$ kubectl -n kube-system get pod -o wide | grep 172.16.0.156
kube-dns-898dbbfc6-hvwlr            3/3       Running   0          8d
172.16.0.156   10.0.0.3
```

Pod 与 DNS 服务之间网络不通

检查下 pod 是否连不上 dns 服务，可以在 pod 里 telnet 一下 dns 的 53 端口:

```
# 连 dns service 的 cluster ip
$ telnet 172.16.14.217 53
```

如果检查到是网络不通，就需要排查下网络设置:

- 检查节点的安全组设置，需要放开集群的容器网段

- 检查是否还有防火墙规则，检查 `iptables`

网络性能差

IPVS 模式吞吐性能低

内核参数关闭 `conn_reuse_mode` ：

```
1.  sysctl net.ipv4.vs.conn_reuse_mode=0
```

参考 issue: https://github.com/kubernetes/kubernetes/issues/70747

集群排错

- Node 全部消失
- Daemonset 没有被调度

Node 全部消失

Rancher 清除 Node 导致集群异常

现象

安装了 rancher 的用户，在卸载 rancher 的时候，可能会手动执行 `kubectl delete ns local` 来删除这个 rancher 创建的 namespace，但直接这样做会导致所有 node 被清除，通过 `kubectl get node` 获取不到 node。

原因

看了下 rancher 源码，rancher 通过 `nodes.management.cattle.io` 这个 CRD 存储和管理 node，会给所有 node 创建对应的这个 CRD 资源，metadata 中加入了两个 finalizer，其中 `user-node-remove_local` 对应的 finalizer 处理逻辑就是删除对应的 k8s node 资源，也就是 `delete ns local` 时，会尝试删除 `nodes.management.cattle.io` 这些 CRD 资源，进而触发 rancher 的 finalizer 逻辑去删除对应的 k8s node 资源，从而清空了 node，所以 `kubectl get node` 就看不到 node 了，集群里的服务就无法被调度。

规避方案

不要在 rancher 组件卸载完之前手动 `delete ns local` 。

Daemonset 没有被调度

Daemonset 的期望实例为 0，可能原因：

- controller-manager 的 bug，重启 controller-manager 可以恢复
- controller-manager 挂了

经典报错

本章包含各种经典报错，分析可能原因并给出相应的解决方案

- `no space left on device`
- `arp_cache: neighbor table overflow!`
- `Cannot allocate memory`

no space left on device

- 有时候节点 NotReady， kubelet 日志报 `no space left on device`
- 有时候创建 Pod 失败， `describe pod` 看 event 报 `no space left on device`

出现这种错误有很多中可能原因，下面我们来根据现象找对应原因。

inotify watch 耗尽

节点 NotReady，kubelet 启动失败，看 kubelet 日志：

```
    Jul 18 15:20:58 VM_16_16_centos kubelet[11519]: E0718 15:20:58.280275   11519 raw
    "/sys/fs/cgroup/memory/kubepods": inotify_add_watch /sys/fs/cgroup/memory/kubepod
1.  52540048533c/6e85761a30707b43ed874e0140f58839618285fc90717153b3cbe7f91629ef5a: no
```

系统调用 `inotify_add_watch` 失败，提示 `no space left on device`， 这是因为系统上进程 watch 文件目录的总数超出了最大限制，可以修改内核参数调高限制，详细请参考本书 处理实践: inotify watch 耗尽

cgroup 泄露

查看当前 cgroup 数量：

```
1.  $ cat /proc/cgroups | column -t
2.  #subsys_name  hierarchy  num_cgroups  enabled
3.  cpuset        5          29           1
4.  cpu           7          126          1
5.  cpuacct       7          126          1
6.  memory        9          127          1
7.  devices       4          126          1
8.  freezer       2          29           1
9.  net_cls       6          29           1
10. blkio         10         126          1
11. perf_event    3          29           1
12. hugetlb       11         29           1
13. pids          8          126          1
14. net_prio      6          29           1
```

cgroup 子系统目录下面所有每个目录及其子目录都认为是一个独立的 cgroup，所以也可以在文件系

统中统计目录数来获取实际 cgroup 数量，通常跟 `/proc/cgroups` 里面看到的应该一致：

```
$ find -L /sys/fs/cgroup/memory -type d | wc -l
127
```

当 cgroup 泄露发生时，这里的数量就不是真实的了，低版本内核限制最大 65535 个 cgroup，并且开启 kmem 删除 cgroup 时会泄露，大量创建删除容器后泄露了许多 cgroup，最终总数达到 65535，新建容器创建 cgroup 将会失败，报 `no space left on device`

详细请参考本书 案例分享：cgroup 泄露

磁盘被写满(TODO)

Pod 启动失败，状态 `CreateContainerError` ：

```
    csi-cephfsplugin-27znb                          0/2     CreateContainerError
167         17h
```

Pod 事件报错：

```
  Warning  Failed   5m1s (x3397 over 17h)  kubelet, ip-10-0-151-35.us-west-2.comp
events): Error: container create failed: container_linux.go:336: starting contair
"process_linux.go:399: container init caused \"rootfs_linux.go:58: mounting \\\"/
\\\"/var/lib/containers/storage/overlay/051e985771cc69f3f699895a1dada9ef6483e912b
\\\"/var/lib/containers/storage/overlay/051e985771cc69f3f699895a1dada9ef6483e912b
caused \\\"no space left on device\\\"\""
```

arp_cache: neighbor table overflow!

节点内核报这个错说明当前节点 arp 缓存满了。

查看当前 arp 记录数:

```
$ arp -an | wc -l
1335
```

查看 gc 阀值:

```
$ sysctl -a | grep net.ipv4.neigh.default.gc_thresh
net.ipv4.neigh.default.gc_thresh1 = 128
net.ipv4.neigh.default.gc_thresh2 = 512
net.ipv4.neigh.default.gc_thresh3 = 1024
```

当前 arp 记录数接近 gc_thresh3 比较容易 overflow，因为当 arp 记录达到 gc_thresh3 时会强制触发 gc 清理，当这时又有数据包要发送，并且根据目的 IP 在 arp cache 中没找到 mac 地址，这时会判断当前 arp cache 记录数加 1 是否大于 gc_thresh3，如果没有大于就会时就会报错： `neighbor table overflow!`

什么场景下会发生

集群规模大，node 和 pod 数量超多，参考本书避坑宝典的 案例分享： ARP 缓存爆满导致健康检查失败

解决方案

调整部分节点内核参数，将 arp cache 的 gc 阀值调高 (`/etc/sysctl.conf`):

```
net.ipv4.neigh.default.gc_thresh1 = 80000
net.ipv4.neigh.default.gc_thresh2 = 90000
net.ipv4.neigh.default.gc_thresh3 = 100000
```

并给 node 打下label，修改 pod spec，加下 nodeSelector 或者 nodeAffnity，让 pod 只调度到这部分改过内核参数的节点

参考资料

- Scaling Kubernetes to 2,500 Nodes: https://openai.com/blog/scaling-kubernetes-to-2500-nodes/

Cannot allocate memory

容器启动失败，报错 `Cannot allocate memory` 。

PID 耗尽

如果登录 ssh 困难，并且登录成功后执行任意命名经常报 `Cannot allocate memory` ，多半是 PID 耗尽了。

处理方法参考本书 处理实践：PID 耗尽

其它排错

Job 无法被删除

原因

- 可能是 k8s 的一个bug: https://github.com/kubernetes/kubernetes/issues/43168
- 本质上是脏数据问题，Running+Succeed != 期望Completions 数量，低版本 kubectl 不容忍，delete job 的时候打开debug(加-v=8)，会看到kubectl不断在重试，直到达到timeout时间。新版kubectl会容忍这些，删除job时会删除关联的pod

解决方法

1. 升级 kubectl 版本，1.12 以上
2. 低版本 kubectl 删除 job 时带 `--cascade=false` 参数(如果job关联的pod没删完，加这个参数不会删除关联的pod)

```
1.  kubectl delete job --cascade=false  <job name>
```

kubectl 执行 exec 或 logs 失败

通常是 apiserver –> kubelet:10250 之间的网络不通，10250 是 kubelet 提供接口的端口，`kubectl exec` 和 `kubectl logs` 的原理就是 apiserver 调 kubelet，kubelet 再调运行时（比如 dockerd）来实现的，所以要保证 kubelet 10250 端口对 apiserver 放通。检查防火墙、iptables 规则是否对 10250 端口或某些 IP 进行了拦截。

内核软死锁

内核报错

```
Oct 14 15:13:05 VM_1_6_centos kernel: NMI watchdog: BUG: soft lockup - CPU#5
stuck for 22s! [runc:[1:CHILD]:2274]
```

原因

发生这个报错通常是内核繁忙（扫描、释放或分配大量对象），分不出时间片给用户态进程导致的，也伴随着高负载，如果负载降低报错则会消失。

什么情况下会导致内核繁忙

- 短时间内创建大量进程（可能是业务需要，也可能是业务bug或用法不正确导致创建大量进程）

参考资料

- What are all these “Bug: soft lockup” messages about : https://www.suse.com/support/kb/doc/?id=7017652

处理实践

高负载

TODO 优化

节点高负载会导致进程无法获得足够的 cpu 时间片来运行，通常表现为网络 timeout，健康检查失败，服务不可用。

过多 IO 等待

有时候即便 cpu ‘us’ (user) 不高但 cpu ‘id’ (idle) 很高的情况节点负载也很高，这是为什么呢？通常是文件 IO 性能达到瓶颈导致 IO WAIT 过多，从而使得节点整体负载升高，影响其它进程的性能。

使用 `top` 命令看下当前负载:

```
top - 19:42:06 up 23:59,  2 users,  load average: 34.64, 35.80, 35.76
Tasks: 679 total,   1 running, 678 sleeping,   0 stopped,   0 zombie
Cpu(s): 15.6%us,  1.7%sy,  0.0%ni, 74.7%id,  7.9%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  32865032k total, 30989168k used,  1875864k free,   370748k buffers
Swap:  8388604k total,     5440k used,  8383164k free,  7982424k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 9783 mysql     20   0 17.3g  16g 8104 S 186.9 52.3   3752:33 mysqld
 5700 nginx     20   0 1330m  66m 9496 S  8.9  0.2   0:20.82 php-fpm
 6424 nginx     20   0 1330m  65m 8372 S  8.3  0.2   0:04.97 php-fpm
 6573 nginx     20   0 1330m  64m 7368 S  8.3  0.2   0:01.49 php-fpm
 5927 nginx     20   0 1320m  56m 9272 S  7.6  0.2   0:12.54 php-fpm
 5956 nginx     20   0 1330m  65m 8500 S  7.6  0.2   0:12.70 php-fpm
 6126 nginx     20   0 1321m  57m 8964 S  7.3  0.2   0:09.72 php-fpm
 6127 nginx     20   0 1319m  54m 9520 S  6.6  0.2   0:08.73 php-fpm
 6131 nginx     20   0 1320m  56m 9404 S  6.6  0.2   0:09.43 php-fpm
 6174 nginx     20   0 1321m  56m 8444 S  6.3  0.2   0:08.92 php-fpm
 5790 nginx     20   0 1319m  54m 9468 S  5.6  0.2   0:17.33 php-fpm
 6575 nginx     20   0 1320m  55m 8212 S  5.6  0.2   0:02.11 php-fpm
 6160 nginx     20   0 1310m  44m 8296 S  4.0  0.1   0:10.05 php-fpm
 5597 nginx     20   0 1310m  46m 9556 S  3.6  0.1   0:21.03 php-fpm
 5786 nginx     20   0 1310m  45m 8528 S  3.6  0.1   0:15.53 php-fpm
 5797 nginx     20   0 1310m  46m 9444 S  3.6  0.1   0:14.02 php-fpm
 6158 nginx     20   0 1310m  45m 8324 S  3.6  0.1   0:10.20 php-fpm
 5698 nginx     20   0 1310m  46m 9184 S  3.3  0.1   0:20.62 php-fpm
 5779 nginx     20   0 1309m  44m 8336 S  3.3  0.1   0:15.34 php-fpm
```

```
 6540 nginx     20   0 1306m  40m 7884 S  3.3  0.1   0:02.46 php-fpm
 5553 nginx     20   0 1300m  36m 9568 S  3.0  0.1   0:21.58 php-fpm
 5722 nginx     20   0 1310m  45m 8552 S  3.0  0.1   0:17.25 php-fpm
 5920 nginx     20   0 1302m  36m 8208 S  3.0  0.1   0:14.23 php-fpm
 6432 nginx     20   0 1310m  45m 8420 S  3.0  0.1   0:05.86 php-fpm
 5285 nginx     20   0 1302m  38m 9696 S  2.7  0.1   0:23.41 php-fpm
```

`wa` (wait) 表示 IO WAIT 的 cpu 占用，默认看到的是所有核的平均值，要看每个核的 `wa` 值需要按下 “1”：

```
top - 19:42:08 up 23:59,  2 users,  load average: 34.64, 35.80, 35.76
Tasks: 679 total,   1 running, 678 sleeping,   0 stopped,   0 zombie
Cpu0  : 29.5%us,  3.7%sy,  0.0%ni, 48.7%id, 17.9%wa,  0.0%hi,  0.1%si,  0.0%st
Cpu1  : 29.3%us,  3.7%sy,  0.0%ni, 48.9%id, 17.9%wa,  0.0%hi,  0.1%si,  0.0%st
Cpu2  : 26.1%us,  3.1%sy,  0.0%ni, 64.4%id,  6.0%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu3  : 25.9%us,  3.1%sy,  0.0%ni, 65.5%id,  5.4%wa,  0.0%hi,  0.1%si,  0.0%st
Cpu4  : 24.9%us,  3.0%sy,  0.0%ni, 66.8%id,  5.0%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu5  : 24.9%us,  2.9%sy,  0.0%ni, 67.0%id,  4.8%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu6  : 24.2%us,  2.7%sy,  0.0%ni, 68.3%id,  4.5%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu7  : 24.3%us,  2.6%sy,  0.0%ni, 68.5%id,  4.2%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu8  : 23.8%us,  2.6%sy,  0.0%ni, 69.2%id,  4.1%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu9  : 23.9%us,  2.5%sy,  0.0%ni, 69.3%id,  4.0%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu10 : 23.3%us,  2.4%sy,  0.0%ni, 68.7%id,  5.6%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu11 : 23.3%us,  2.4%sy,  0.0%ni, 69.2%id,  5.1%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu12 : 21.8%us,  2.4%sy,  0.0%ni, 60.2%id, 15.5%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu13 : 21.9%us,  2.4%sy,  0.0%ni, 60.6%id, 15.2%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu14 : 21.4%us,  2.3%sy,  0.0%ni, 72.6%id,  3.7%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu15 : 21.5%us,  2.2%sy,  0.0%ni, 73.2%id,  3.1%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu16 : 21.2%us,  2.2%sy,  0.0%ni, 73.6%id,  3.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu17 : 21.2%us,  2.1%sy,  0.0%ni, 73.8%id,  2.8%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu18 : 20.9%us,  2.1%sy,  0.0%ni, 74.1%id,  2.9%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu19 : 21.0%us,  2.1%sy,  0.0%ni, 74.4%id,  2.5%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu20 : 20.7%us,  2.0%sy,  0.0%ni, 73.8%id,  3.4%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu21 : 20.8%us,  2.0%sy,  0.0%ni, 73.9%id,  3.2%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu22 : 20.8%us,  2.0%sy,  0.0%ni, 74.4%id,  2.8%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu23 : 20.8%us,  1.9%sy,  0.0%ni, 74.4%id,  2.8%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  32865032k total, 30209248k used,  2655784k free,   370748k buffers
Swap:  8388604k total,     5440k used,  8383164k free,  7986552k cached
```

`wa` 通常是 0%，如果经常在 1 之上，说明存储设备的速度已经太慢，无法跟上 cpu 的处理速度。

使用 `atop` 看下当前磁盘 IO 状态：

```
ATOP - lemp                2017/01/23  19:42:32                ---------
10s elapsed
PRC | sys    3.18s | user  33.24s | #proc    679 | #tslpu    28 | #zombie    0
| #exit      0 |
CPU | sys      29% | user    330% | irq       1% | idle   1857% | wait    182%
| curscal  69% |
CPL | avg1   33.00 | avg5   35.29 | avg15  35.59 | csw    62610 | intr   76926
| numcpu    24 |
MEM | tot    31.3G | free    2.1G | cache   7.6G | dirty  41.0M | buff  362.1M
| slab    1.2G |
SWP | tot     8.0G | free    8.0G |              |              | vmcom  23.9G
| vmlim  23.7G |
DSK |          sda | busy    100% | read       4 | write   1789 | MBw/s   2.84
| avio 5.58 ms |
NET | transport    | tcpi   10357 | tcpo    9065 | udpi       0 | udpo       0
| tcpao    174 |
NET | network      | ipi    10360 | ipo     9065 | ipfrw      0 | deliv  10359
| icmpo      0 |
NET | eth0      4% | pcki    6649 | pcko    6136 | si 1478 Kbps | so 4115 Kbps
| erro       0 |
NET | lo      ---- | pcki    4082 | pcko    4082 | si 8967 Kbps | so 8967 Kbps
| erro       0 |

  PID   TID  THR  SYSCPU  USRCPU  VGROW  RGROW  RDDSK  WRDSK ST EXC S CPUNR
CPU CMD       1/12
 9783     -  156   0.21s  19.44s     0K  -788K     4K  1344K --    - S     4
197% mysqld
 5596     -    1   0.10s   0.62s 47204K 47004K     0K   220K --    - S    18
7% php-fpm
 6429     -    1   0.06s   0.34s 19840K 19968K     0K     0K --    - S    21
4% php-fpm
 6210     -    1   0.03s   0.30s -5216K -5204K     0K     0K --    - S    19
3% php-fpm
 5757     -    1   0.05s   0.27s 26072K 26012K     0K     4K --    - S    13
3% php-fpm
 6433     -    1   0.04s   0.28s -2816K -2816K     0K     0K --    - S    11
3% php-fpm
 5846     -    1   0.06s   0.22s -2560K -2660K     0K     0K --    - S     7
3% php-fpm
 5791     -    1   0.05s   0.21s  5764K  5692K     0K     0K --    - S    22
3% php-fpm
 5860     -    1   0.04s   0.21s 48088K 47724K     0K     0K --    - S     1
3% php-fpm
```

```
  6231     -    1   0.04s   0.20s  -256K    -4K     0K     0K --    - S     1
23. 2% php-fpm
  6154     -    1   0.03s   0.21s -3004K -3184K     0K     0K --    - S    21
24. 2% php-fpm
  6573     -    1   0.04s   0.20s  -512K  -168K     0K     0K --    - S     4
25. 2% php-fpm
  6435     -    1   0.04s   0.19s -3216K -2980K     0K     0K --    - S    15
26. 2% php-fpm
  5954     -    1   0.03s   0.20s     0K   164K     0K     4K --    - S     0
27. 2% php-fpm
  6133     -    1   0.03s   0.19s 41056K 40432K     0K     0K --    - S    18
28. 2% php-fpm
  6132     -    1   0.02s   0.20s 37836K 37440K     0K     0K --    - S    11
29. 2% php-fpm
  6242     -    1   0.03s   0.19s -12.2M -12.3M     0K     4K --    - S    12
30. 2% php-fpm
  6285     -    1   0.02s   0.19s 39516K 39420K     0K     0K --    - S     3
31. 2% php-fpm
  6455     -    1   0.05s   0.16s 29008K 28560K     0K     0K --    - S    14
32. 2% php-fpm
```

在本例中磁盘 `sda` 已经 100% busy，已经严重达到性能瓶颈。按 ‘d’ 看下是哪些进程在使用磁盘IO:

```
  ATOP - lemp                2017/01/23  19:42:46                ---------
 1. 2s elapsed
  PRC | sys    0.24s | user   1.99s | #proc    679 | #tslpu    54 | #zombie    0
 2. | #exit      0 |
  CPU | sys      11% | user    101% | irq       1% | idle   2089% | wait    208%
 3. | curscal  63% |
  CPL | avg1   38.49 | avg5   36.48 | avg15  35.98 | csw     4654 | intr    6876
 4. | numcpu    24 |
  MEM | tot    31.3G | free    2.2G | cache   7.6G | dirty  48.7M | buff  362.1M
 5. | slab    1.2G |
  SWP | tot     8.0G | free    8.0G |              |              | vmcom  23.9G
 6. | vmlim  23.7G |
  DSK |          sda | busy    100% | read       2 | write    362 | MBw/s   2.28
 7. | avio 5.49 ms |
  NET | transport    | tcpi    1031 | tcpo     968 | udpi       0 | udpo       0
 8. | tcpao     45 |
  NET | network      | ipi     1031 | ipo      968 | ipfrw      0 | deliv   1031
 9. | icmpo      0 |
  NET | eth0      1% | pcki     558 | pcko     508 | si  762 Kbps | so 1077 Kbps
10. | erro       0 |
```

```
    NET | lo      ---- | pcki     406 | pcko     406 | si 2273 Kbps | so 2273 Kbps
11. | erro       0 |
12.
      PID          TID         RDDSK         WRDSK        WCANCL          DSK
13. CMD         1/5
       9783            -            0K          468K           16K          40%
14. mysqld
       1930            -            0K          212K            0K          18%
15. flush-8:0
       5896            -            0K          152K            0K          13%
16. nginx
        880            -            0K          148K            0K          13%
17. jbd2/sda5-8
       5909            -            0K           60K            0K           5%
18. nginx
       5906            -            0K           36K            0K           3%
19. nginx
       5907            -           16K            8K            0K           2%
20. nginx
       5903            -           20K            0K            0K           2%
21. nginx
       5901            -            0K           12K            0K           1%
22. nginx
       5908            -            0K            8K            0K           1%
23. nginx
       5894            -            0K            8K            0K           1%
24. nginx
       5911            -            0K            8K            0K           1%
25. nginx
       5900            -            0K            4K            4K           0%
26. nginx
       5551            -            0K            4K            0K           0%
27. php-fpm
       5913            -            0K            4K            0K           0%
28. nginx
       5895            -            0K            4K            0K           0%
29. nginx
       6133            -            0K            0K            0K           0%
30. php-fpm
       5780            -            0K            0K            0K           0%
31. php-fpm
       6675            -            0K            0K            0K           0%
32. atop
```

也可以使用 `iotop -oPa` 查看哪些进程占用磁盘 IO:

```
Total DISK READ: 15.02 K/s | Total DISK WRITE: 3.82 M/s
  PID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 1930 be/4 root          0.00 B   1956.00 K  0.00 % 83.34 % [flush-8:0]
 5914 be/4 nginx         0.00 B      0.00 B  0.00 % 36.56 % nginx: cache
manager process
  880 be/3 root          0.00 B     21.27 M  0.00 % 35.03 % [jbd2/sda5-8]
 5913 be/2 nginx        36.00 K   1000.00 K  0.00 %  8.94 % nginx: worker
process
 5910 be/2 nginx         0.00 B   1048.00 K  0.00 %  8.43 % nginx: worker
process
 5896 be/2 nginx        56.00 K    452.00 K  0.00 %  6.91 % nginx: worker
process
 5909 be/2 nginx        20.00 K   1144.00 K  0.00 %  6.24 % nginx: worker
process
 5890 be/2 nginx        48.00 K    692.00 K  0.00 %  6.07 % nginx: worker
process
 5892 be/2 nginx        84.00 K    736.00 K  0.00 %  5.71 % nginx: worker
process
 5901 be/2 nginx        20.00 K    504.00 K  0.00 %  5.46 % nginx: worker
process
 5899 be/2 nginx         0.00 B    596.00 K  0.00 %  5.14 % nginx: worker
process
 5897 be/2 nginx        28.00 K   1388.00 K  0.00 %  4.90 % nginx: worker
process
 5908 be/2 nginx        48.00 K    700.00 K  0.00 %  4.43 % nginx: worker
process
 5905 be/2 nginx        32.00 K   1140.00 K  0.00 %  4.36 % nginx: worker
process
 5900 be/2 nginx         0.00 B   1208.00 K  0.00 %  4.31 % nginx: worker
process
 5904 be/2 nginx        36.00 K   1244.00 K  0.00 %  2.80 % nginx: worker
process
 5895 be/2 nginx        16.00 K    780.00 K  0.00 %  2.50 % nginx: worker
process
 5907 be/2 nginx         0.00 B   1548.00 K  0.00 %  2.43 % nginx: worker
process
 5903 be/2 nginx        36.00 K   1032.00 K  0.00 %  2.34 % nginx: worker
process
 6130 be/4 nginx         0.00 B     72.00 K  0.00 %  2.18 % php-fpm: pool www
 5906 be/2 nginx        12.00 K    844.00 K  0.00 %  2.10 % nginx: worker
process
 5889 be/2 nginx        40.00 K   1164.00 K  0.00 %  2.00 % nginx: worker
process
```

```
 5894 be/2 nginx        44.00 K    760.00 K  0.00 %  1.61 % nginx: worker
process
 5902 be/2 nginx        52.00 K    992.00 K  0.00 %  1.55 % nginx: worker
process
 5893 be/2 nginx        64.00 K    972.00 K  0.00 %  1.22 % nginx: worker
process
 5814 be/4 nginx        36.00 K     44.00 K  0.00 %  1.06 % php-fpm: pool www
 6159 be/4 nginx         4.00 K      4.00 K  0.00 %  1.00 % php-fpm: pool www
 5693 be/4 nginx         0.00 B      4.00 K  0.00 %  0.86 % php-fpm: pool www
 5912 be/2 nginx        68.00 K    300.00 K  0.00 %  0.72 % nginx: worker
process
 5911 be/2 nginx        20.00 K    788.00 K  0.00 %  0.72 % nginx: worker
process
```

通过 `man iotop` 可以看下这几个参数的含义：

```
-o, --only
       Only show processes or threads actually doing I/O, instead of showing
all processes or threads. This can be dynamically toggled by pressing o.
-P, --processes
       Only show processes. Normally iotop shows all threads.

-a, --accumulated
       Show accumulated I/O instead of bandwidth. In this mode, iotop shows the
amount of I/O processes have done since iotop started.
```

节点上部署了其它非 K8S 管理的服务

TODO 优化

比如在节点上装了数据库，但不被 K8S 所管理，这是用法不正确，不建议在 K8S 节点上部署其它进程。

参考资料

- Linux server performance: Is disk I/O slowing your application: https://haydenjames.io/linux-server-performance-disk-io-slowing-application/

内存碎片化

判断是否内存碎片化严重

内存页分配失败，内核日志报类似下面的错：

```
1.  mysqld: page allocation failure. order:4, mode:0x10c0d0
```

- `mysqld` 是被分配的内存的程序
- `order` 表示需要分配连续页的数量(2^order)，这里 4 表示 2^4=16 个连续的页
- `mode` 是内存分配模式的标识，定义在内核源码文件 `include/linux/gfp.h` 中，通常是多个标识相与运算的结果，不同版本内核可能不一样，比如在新版内核中 `GFP_KERNEL` 是 `__GFP_RECLAIM | __GFP_IO | __GFP_FS` 的运算结果，而 `__GFP_RECLAIM` 又是 `___GFP_DIRECT_RECLAIM|___GFP_KSWAPD_RECLAIM` 的运算结果

当 order 为 0 时，说明系统以及完全没有可用内存了，order 值比较大时，才说明内存碎片化了，无法分配连续的大页内存。

内存碎片化造成的问题

容器启动失败

K8S 会为每个 pod 创建 netns 来隔离 network namespace，内核初始化 netns 时会为其创建 nf_conntrack 表的 cache，需要申请大页内存，如果此时系统内存已经碎片化，无法分配到足够的大页内存内核就会报错(`v2.6.33 - v4.6`)：

```
1.  runc:[1:CHILD]: page allocation failure: order:6, mode:0x10c0d0
```

Pod 状态将会一直在 ContainerCreating，dockerd 启动容器失败，日志报错：

```
    Jan 23 14:15:31 dc05 dockerd: time="2019-01-23T14:15:31.288446233+08:00"
    level=error msg="containerd: start container" error="oci runtime error:
    container_linux.go:247: starting container process caused
    \"process_linux.go:245: running exec setns process for init caused \\\"exit
    status 6\\\"\"\n"
1.  id=5b9be8c5bb121264899fac8d9d36b02150269d41ce96ba6ad36d70b8640cb01c
```

```
Jan 23 14:15:31 dc05 dockerd: time="2019-01-23T14:15:31.317965799+08:00"
level=error msg="Create container failed with error: invalid header field value
\"oci runtime error: container_linux.go:247: starting container process caused
\\\"process_linux.go:245: running exec setns process for init caused
\\\\\\\"exit status 6\\\\\\\"\\\"\\n\""
```

kubelet 日志报错：

```
Jan 23 14:15:31 dc05 kubelet: E0123 14:15:31.352386   26037
remote_runtime.go:91] RunPodSandbox from runtime service failed: rpc error:
code = 2 desc = failed to start sandbox container for pod "matchdataserver-
1255064836-t4b2w": Error response from daemon: {"message":"invalid header field
value \"oci runtime error: container_linux.go:247: starting container process
caused \\\"process_linux.go:245: running exec setns process for init caused
\\\\\\\"exit status 6\\\\\\\"\\\"\\n\""}
Jan 23 14:15:31 dc05 kubelet: E0123 14:15:31.352496   26037
kuberuntime_sandbox.go:54] CreatePodSandbox for pod "matchdataserver-
1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)" failed: rpc
error: code = 2 desc = failed to start sandbox container for pod
"matchdataserver-1255064836-t4b2w": Error response from daemon:
{"message":"invalid header field value \"oci runtime error:
container_linux.go:247: starting container process caused
\\\"process_linux.go:245: running exec setns process for init caused
\\\\\\\"exit status 6\\\\\\\"\\\"\\n\""}
Jan 23 14:15:31 dc05 kubelet: E0123 14:15:31.352518   26037
kuberuntime_manager.go:618] createPodSandbox for pod "matchdataserver-
1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)" failed: rpc
error: code = 2 desc = failed to start sandbox container for pod
"matchdataserver-1255064836-t4b2w": Error response from daemon:
{"message":"invalid header field value \"oci runtime error:
container_linux.go:247: starting container process caused
\\\"process_linux.go:245: running exec setns process for init caused
\\\\\\\"exit status 6\\\\\\\"\\\"\\n\""}
Jan 23 14:15:31 dc05 kubelet: E0123 14:15:31.352580   26037 pod_workers.go:182]
Error syncing pod 485fd485-1ed6-11e9-8661-0a587f8021ea ("matchdataserver-
1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)"), skipping:
failed to "CreatePodSandbox" for "matchdataserver-1255064836-
t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)" with CreatePodSandboxError:
"CreatePodSandbox for pod \"matchdataserver-1255064836-t4b2w_basic(485fd485-
1ed6-11e9-8661-0a587f8021ea)\" failed: rpc error: code = 2 desc = failed to
start sandbox container for pod \"matchdataserver-1255064836-t4b2w\": Error
response from daemon: {\"message\":\"invalid header field value \\\"oci runtime
error: container_linux.go:247: starting container process caused
\\\\\\\"process_linux.go:245: running exec setns process for init caused
\\\\\\\\\\\\\\\"exit status 6\\\\\\\\\\\\\\\"\\\\\\\"\\\\n\\\"\"}"
```

```
Jan 23 14:15:31 dc05 kubelet: I0123 14:15:31.372181   26037 kubelet.go:1916] SyncLoop (PLEG): "matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)", event: &pleg.PodLifecycleEvent{ID:"485fd485-1ed6-11e9-8661-0a587f8021ea", Type:"ContainerDied", Data:"5b9be8c5bb121264899fac8d9d36b02150269d41ce96ba6ad36d70b8640cb01c"}
Jan 23 14:15:31 dc05 kubelet: W0123 14:15:31.372225   26037 pod_container_deletor.go:77] Container "5b9be8c5bb121264899fac8d9d36b02150269d41ce96ba6ad36d70b8640cb01c" not found in pod's containers
Jan 23 14:15:31 dc05 kubelet: I0123 14:15:31.678211   26037 kuberuntime_manager.go:383] No ready sandbox for pod "matchdataserver-1255064836-t4b2w_basic(485fd485-1ed6-11e9-8661-0a587f8021ea)" can be found. Need to start a new one
```

查看slab（后面的0多表示伙伴系统没有大块内存了）：

```
$ cat /proc/buddyinfo
Node 0, zone      DMA      1      0      1      0      2      1      1      0      1      1      3
Node 0, zone    DMA32   2725    624    489    178      0      0      0      0      0      0      0
Node 0, zone   Normal   1163   1101    932    222      0      0      0      0      0      0      0
```

系统 OOM

内存碎片化会导致即使当前系统总内存比较多，但由于无法分配足够的大页内存导致给进程分配内存失败，就认为系统内存不够用，需要杀掉一些进程来释放内存，从而导致系统 OOM

解决方法

- 周期性地或者在发现大块内存不足时，先进行drop_cache操作：

```
echo 3 > /proc/sys/vm/drop_caches
```

- 必要时候进行内存整理，开销会比较大，会造成业务卡住一段时间(慎用)：

```
echo 1 > /proc/sys/vm/compact_memory
```

如何防止内存碎片化

TODO

附录

相关链接:

- https://huataihuang.gitbooks.io/cloud-atlas/content/os/linux/kernel/memory/drop_caches_and_compact_memory.html

磁盘爆满

什么情况下磁盘可能会爆满 ？

kubelet 有 gc 和驱逐机制，通过 `--image-gc-high-threshold` ， `--image-gc-low-threshold` ， `--eviction-hard` ， `--eviction-soft` ， `--eviction-minimum-reclaim` 等参数控制 kubelet 的 gc 和驱逐策略来释放磁盘空间，如果配置正确的情况下，磁盘一般不会爆满。

通常导致爆满的原因可能是配置不正确或者节点上有其它非 K8S 管理的进程在不断写数据到磁盘占用大量空间导致磁盘爆满。

磁盘爆满会有什么影响 ？

影响 K8S 运行我们主要关注 kubelet 和容器运行时这两个最关键的组件，它们所使用的目录通常不一样，kubelet 一般不会单独挂盘，直接使用系统磁盘，因为通常占用空间不会很大，容器运行时单独挂盘的场景比较多，当磁盘爆满的时候我们也要看 kubelet 和 容器运行时使用的目录是否在这个磁盘，通过 `df` 命令可以查看磁盘挂载点。

容器运行时使用的目录所在磁盘爆满

如果容器运行时使用的目录所在磁盘空间爆满，可能会造成容器运行时无响应，比如 docker，执行 docker 相关的命令一直 hang 住， kubelet 日志也可以看到 PLEG unhealthy，因为 CRI 调用 timeout，当然也就无法创建或销毁容器，通常表现是 Pod 一直 ContainerCreating 或 一直 Terminating。

docker 默认使用的目录主要有：

- `/var/run/docker` ：用于存储容器运行状态，通过 dockerd 的 `--exec-root` 参数指定。
- `/var/lib/docker` ：用于持久化容器相关的数据，比如容器镜像、容器可写层数据、容器标准日志输出、通过 docker 创建的 volume 等

Pod 启动可能报类似下面的事件：

```
  Warning  FailedCreatePodSandBox    53m                  kubelet, 172.22.0.44
Failed create pod sandbox: rpc error: code = DeadlineExceeded desc = context
1. deadline exceeded
```

```
  Warning  FailedCreatePodSandBox  2m (x4307 over 16h)  kubelet, 10.179.80.31  (c
similar events): Failed create pod sandbox: rpc error: code = Unknown desc = fail
sandbox for pod "apigateway-6dc48bf8b6-l8xrw": Error response from daemon: mkdir
/var/lib/docker/aufs/mnt/1f09d6c1c9f24e8daaea5bf33a4230de7dbc758e3b22785e8ee21e3e
no space left on device
```

```
  Warning  Failed   5m1s (x3397 over 17h)  kubelet, ip-10-0-151-35.us-west-2.comp
similar events): Error: container create failed: container_linux.go:336: starting
"process_linux.go:399: container init caused \"rootfs_linux.go:58: mounting \\\"/
\\\"/var/lib/dockerd/storage/overlay/051e985771cc69f3f699895a1dada9ef6483e912b46a
\\\"/var/lib/dockerd/storage/overlay/051e985771cc69f3f699895a1dada9ef6483e912b46a
caused \\\"no space left on device\\\"\""
```

Pod 删除可能报类似下面的事件：

```
  Normal  Killing  39s (x735 over 15h)  kubelet, 10.179.80.31  Killing container
with id docker://apigateway:Need to kill Pod
```

kubelet 使用的目录所在磁盘爆满

如果 kubelet 使用的目录所在磁盘空间爆满(通常是系统盘)，新建 Pod 时连 Sandbox 都无法创建成功，因为 mkdir 将会失败，通常会有类似这样的 Pod 事件：

```
  Warning  UnexpectedAdmissionError  44m                kubelet, 172.22.0.44
Update plugin resources failed due to failed to write checkpoint file
"kubelet_internal_checkpoint": write /var/lib/kubelet/device-
plugins/.728425055: no space left on device, which is unexpected.
```

kubelet 默认使用的目录是 `/var/lib/kubelet` ， 用于存储插件信息、Pod 相关的状态以及挂载的 volume (比如 `emptyDir` ， `ConfigMap` ， `Secret`)，通过 kubelet 的 `--root-dir` 参数指定。

如何分析磁盘占用 ？

- 如果运行时使用的是 Docker，请参考本书 排错技巧: 分析 Docker 磁盘占用 (TODO)

如何恢复 ？

如果容器运行时使用的 Docker，我们无法直接重启 dockerd 来释放一些空间，因为磁盘爆满后

dockerd 无法正常响应，停止的时候也会卡住。我们需要先手动清理一点文件腾出空间好让 dockerd 能够停止并重启。

可以手动删除一些 docker 的 log 文件或可写层文件，通常删除 log:

```
$ cd /var/lib/docker/containers
$ du -sh * # 找到比较大的目录
$ cd dda02c9a7491fa797ab730c1568ba06cba74cecd4e4a82e9d90d00fa11de743c
$ cat /dev/null >
dda02c9a7491fa797ab730c1568ba06cba74cecd4e4a82e9d90d00fa11de743c-json.log.9 #
删除log文件
```

- 注意：使用 `cat /dev/null >` 方式删除而不用 `rm`，因为用 rm 删除的文件，docker 进程可能不会释放文件，空间也就不会释放；log 的后缀数字越大表示越久远，先删除旧日志。

然后将该 node 标记不可调度，并将其已有的 pod 驱逐到其它节点，这样重启 dockerd 就会让该节点的 pod 对应的容器删掉，容器相关的日志(标准输出)与容器内产生的数据文件(没有挂载 volume，可写层)也会被清理:

```
kubectl drain <node-name>
```

重启 dockerd:

```
systemctl restart dockerd
# or systemctl restart docker
```

等重启恢复，pod 调度到其它节点，排查磁盘爆满原因并清理和规避，然后取消节点不可调度标记:

```
kubectl uncordon <node-name>
```

如何规避 ？

正确配置 kubelet gc 和 驱逐相关的参数，即便到达爆满地步，此时节点上的 pod 也都早就自动驱逐到其它节点了，不会存在 Pod 一直 ContainerCreating 或 Terminating 的问题。

inotify watch 耗尽

每个 linux 进程可以持有多个 fd，每个 inotify 类型的 fd 可以 watch 多个目录，每个用户下所有进程 inotify 类型的 fd 可以 watch 的总目录数有个最大限制，这个限制可以通过内核参数配置： `fs.inotify.max_user_watches`

查看最大 inotify watch 数:

```
$ cat /proc/sys/fs/inotify/max_user_watches
8192
```

使用下面的脚本查看当前有 inotify watch 类型 fd 的进程以及每个 fd watch 的目录数量，降序输出，带总数统计:

```
#!/usr/bin/env bash
#
# Copyright 2019 (c) roc
#
# This script shows processes holding the inotify fd, alone with HOW MANY
directories each inotify fd watches(0 will be ignored).
total=0
result="EXE PID FD-INFO INOTIFY-WATCHES\n"
while read pid fd; do \
  exe="$(readlink -f /proc/$pid/exe || echo n/a)"; \
  fdinfo="/proc/$pid/fdinfo/$fd" ; \
  count="$(grep -c inotify "$fdinfo" || true)"; \
  if [ $((count)) != 0 ]; then
    total=$((total+count)); \
    result+="$exe $pid $fdinfo $count\n"; \
  fi
done <<< "$(lsof +c 0 -n -P -u root|awk '/inotify$/ { gsub(/[urw]$/,"",$4);
print $2" "$4 }')" && echo "total $total inotify watches" && result="$(echo -e
$result|column -t)\n" && echo -e "$result" | head -1 && echo -e "$result" | sed
"1d" | sort -k 4rn;
```

示例输出:

```
total 7882 inotify watches
EXE                                          PID    FD-INFO
INOTIFY-WATCHES
/usr/local/qcloud/YunJing/YDEyes/YDService  25813  /proc/25813/fdinfo/8   7077
```

```
/usr/bin/kubelet                          1173   /proc/1173/fdinfo/22   665
/usr/bin/ruby2.3                          13381  /proc/13381/fdinfo/14  54
/usr/lib/policykit-1/polkitd              1458   /proc/1458/fdinfo/9    14
/lib/systemd/systemd-udevd                450    /proc/450/fdinfo/9     13
/usr/sbin/nscd                            7935   /proc/7935/fdinfo/3    6
/usr/bin/kubelet                          1173   /proc/1173/fdinfo/28   5
/lib/systemd/systemd                      1      /proc/1/fdinfo/17      4
/lib/systemd/systemd                      1      /proc/1/fdinfo/18      4
/lib/systemd/systemd                      1      /proc/1/fdinfo/26      4
/lib/systemd/systemd                      1      /proc/1/fdinfo/28      4
/usr/lib/policykit-1/polkitd              1458   /proc/1458/fdinfo/8    4
/usr/local/bin/sidecar-injector           4751   /proc/4751/fdinfo/3    3
/usr/lib/accountsservice/accounts-daemon  1178   /proc/1178/fdinfo/7    2
/usr/local/bin/galley                     8228   /proc/8228/fdinfo/10   2
/usr/local/bin/galley                     8228   /proc/8228/fdinfo/9    2
/lib/systemd/systemd                      1      /proc/1/fdinfo/11      1
/sbin/agetty                              1437   /proc/1437/fdinfo/4    1
/sbin/agetty                              1440   /proc/1440/fdinfo/4    1
/usr/bin/kubelet                          1173   /proc/1173/fdinfo/10   1
/usr/local/bin/envoy                      4859   /proc/4859/fdinfo/5    1
/usr/local/bin/envoy                      5427   /proc/5427/fdinfo/5    1
/usr/local/bin/envoy                      6058   /proc/6058/fdinfo/3    1
/usr/local/bin/envoy                      6893   /proc/6893/fdinfo/3    1
/usr/local/bin/envoy                      6950   /proc/6950/fdinfo/3    1
/usr/local/bin/galley                     8228   /proc/8228/fdinfo/3    1
/usr/local/bin/pilot-agent                3819   /proc/3819/fdinfo/5    1
/usr/local/bin/pilot-agent                4244   /proc/4244/fdinfo/5    1
/usr/local/bin/pilot-agent                5901   /proc/5901/fdinfo/3    1
/usr/local/bin/pilot-agent                6789   /proc/6789/fdinfo/3    1
/usr/local/bin/pilot-agent                6808   /proc/6808/fdinfo/3    1
/usr/local/bin/pilot-discovery            6231   /proc/6231/fdinfo/3    1
/usr/local/bin/sidecar-injector           4751   /proc/4751/fdinfo/5    1
/usr/sbin/acpid                           1166   /proc/1166/fdinfo/6    1
/usr/sbin/dnsmasq                         7572   /proc/7572/fdinfo/8    1
```

如果看到总 watch 数比较大，接近最大限制，可以修改内核参数调高下这个限制。

临时调整：

```
sudo sysctl fs.inotify.max_user_watches=524288
```

永久生效：

```
1.  echo "fs.inotify.max_user_watches=524288" >> /etc/sysctl.conf && sysctl -p
```

打开 inotify_add_watch 跟踪，进一步 debug inotify watch 耗尽的原因：

```
    echo 1 >>
1.  /sys/kernel/debug/tracing/events/syscalls/sys_exit_inotify_add_watch/enable
```

PID 耗尽

如何判断 PID 耗尽

首先要确认当前的 PID 限制，检查全局 PID 最大限制：

```
1. cat /proc/sys/kernel/pid_max
```

也检查下当前用户是否还有 `ulimit` 限制最大进程数。

然后要确认当前实际 PID 数量，检查当前用户的 PID 数量：

```
1. ps -eLf | wc -l
```

如果发现实际 PID 数量接近最大限制说明 PID 就可能会爆满导致经常有进程无法启动，低版本内核可能报错：`Cannot allocate memory`，这个报错信息不准确，在内核 4.1 以后改进了：https://github.com/torvalds/linux/commit/35f71bc0a09a45924bed268d8ccd0d3407bc476f

如何解决

临时调大：

```
1. echo 65535 > /proc/sys/kernel/pid_max
```

永久调大：

```
1. echo "kernel.pid_max=65535 " >> /etc/sysctl.conf && sysctl -p
```

k8s 1.14 支持了限制 Pod 的进程数量：https://kubernetes.io/blog/2019/04/15/process-id-limiting-for-stability-improvements-in-kubernetes-1.14/

arp_cache 溢出

如何判断 arp_cache 溢出？

内核日志会有有下面的报错：

```
arp_cache: neighbor table overflow!
```

查看当前 arp 记录数：

```
$ arp -an | wc -l
1335
```

查看 arp gc 阀值：

```
$ systectl -a | grep gc_thresh
net.ipv4.neigh.default.gc_thresh1 = 128
net.ipv4.neigh.default.gc_thresh2 = 512
net.ipv4.neigh.default.gc_thresh3 = 1024
net.ipv6.neigh.default.gc_thresh1 = 128
net.ipv6.neigh.default.gc_thresh2 = 512
net.ipv6.neigh.default.gc_thresh3 = 1024
```

当前 arp 记录数接近 `gc_thresh3` 比较容易 overflow，因为当 arp 记录达到 `gc_thresh3` 时会强制触发 gc 清理，当这时又有数据包要发送，并且根据目的 IP 在 arp cache 中没找到 mac 地址，这时会判断当前 arp cache 记录数加 1 是否大于 `gc_thresh3` ，如果没有大于就会 时就会报错： `arp_cache: neighbor table overflow!`

解决方案

调整节点内核参数，将 arp cache 的 gc 阀值调高 (`/etc/sysctl.conf`)：

```
net.ipv4.neigh.default.gc_thresh1 = 80000
net.ipv4.neigh.default.gc_thresh2 = 90000
net.ipv4.neigh.default.gc_thresh3 = 100000
```

分析是否只是部分业务的 Pod 的使用场景需要节点有比较大的 arp 缓存空间。

如果不是，就需要调整所有节点内核参数。

如果是，可以将部分 Node 打上标签，比如：

```
kubectl label node host1 arp_cache=large
```

然后用 nodeSelector 或 nodeAffnity 让这部分需要内核有大 arp_cache 容量的 Pod 只调度到这部分节点，推荐使用 nodeAffnity，yaml 示例：

```yaml
template:
  spec:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: arp_cache
              operator: In
              values:
              - large
```

踩坑总结

- cgroup 泄露
- tcp_tw_recycle 引发丢包
- 使用 oom-guard 在用户态处理 cgroup OOM
- no space left on device

cgroup 泄露

内核 Bug

`memcg` 是 Linux 内核中用于管理 cgroup 内存的模块，整个生命周期应该是跟随 cgroup 的，但是在低版本内核中(已知3.10)，一旦给某个 memory cgroup 开启 kmem accounting 中的 `memory.kmem.limit_in_bytes` 就可能会导致不能彻底删除 memcg 和对应的 cssid，也就是说应用即使已经删除了 cgroup (`/sys/fs/cgroup/memory` 下对应的 cgroup 目录已经删除)，但在内核中没有释放 cssid，导致内核认为的 cgroup 的数量实际数量不一致，我们也无法得知内核认为的 cgroup 数量是多少。

关于 cgroup kernel memory，在 kernel.org 中有如下描述：

```
2.7 Kernel Memory Extension (CONFIG_MEMCG_KMEM)
-----------------------------------------------

With the Kernel memory extension, the Memory Controller is able to limit
the amount of kernel memory used by the system. Kernel memory is fundamentally
different than user memory, since it can't be swapped out, which makes it
possible to DoS the system by consuming too much of this precious resource.

Kernel memory accounting is enabled for all memory cgroups by default. But
it can be disabled system-wide by passing cgroup.memory=nokmem to the kernel
at boot time. In this case, kernel memory will not be accounted at all.

Kernel memory limits are not imposed for the root cgroup. Usage for the root
cgroup may or may not be accounted. The memory used is accumulated into
memory.kmem.usage_in_bytes, or in a separate counter when it makes sense.
(currently only for tcp).

The main "kmem" counter is fed into the main counter, so kmem charges will
also be visible from the user counter.

Currently no soft limit is implemented for kernel memory. It is future work
to trigger slab reclaim when those limits are reached.
```

这是一个 cgroup memory 的扩展，用于限制对 kernel memory 的使用，但该特性在老于 4.0 版本中是个实验特性，存在泄露问题，在 4.x 较低的版本也还有泄露问题，应该是造成泄露的代码路径没有完全修复，推荐 4.3 以上的内核。

造成容器创建失败

这个问题可能会导致创建容器失败，因为创建容器为其需要创建 cgroup 来做隔离，而低版本内核有个限制：允许创建的 cgroup 最大数量写死为 65535 (点我跳转到 commit)，如果节点上经常创建和销毁大量容器导致创建很多 cgroup，删除容器但没有彻底删除 cgroup 造成泄露(真实数量我们无法得知)，到达 65535 后再创建容器就会报创建 cgroup 失败并报错 `no space left on device` ，使用 kubernetes 最直观的感受就是 pod 创建之后无法启动成功。

pod 启动失败，报 event 示例：

```
Events:
  Type     Reason                  Age                From
Message
  ----     ------                  ----               ----
-------
  Normal   Scheduled               15m                default-scheduler
Successfully assigned jenkins/jenkins-7845b9b665-nrvks to 10.10.252.4
  Warning  FailedCreatePodContainer  25s (x70 over 15m)  kubelet, 10.10.252.4
unable to ensure pod container exists: failed to create container for [kubepods
besteffort podc6eeec88-8664-11e9-9524-5254007057ba] : mkdir
/sys/fs/cgroup/memory/kubepods/besteffort/podc6eeec88-8664-11e9-9524-5254007057ba
no space left on device
```

dockerd 日志报错示例：

```
Dec 24 11:54:31 VM_16_11_centos dockerd[11419]: time="2018-12-
24T11:54:31.195900301+08:00" level=error msg="Handler for POST
/v1.31/containers/b98d4aea818bf9d1d1aa84079e1688cd9b4218e008c58a8ef6d6c3c106403e7
returned error: OCI runtime create failed: container_linux.go:348: starting conta
process caused \"process_linux.go:279: applying cgroup configuration for process
\\\"mkdir /sys/fs/cgroup/memory/kubepods/burstable/pod79fe803c-072f-11e9-90ca-
525400090c71/b98d4aea818bf9d1d1aa84079e1688cd9b4218e008c58a8ef6d6c3c106403e7b: no
left on device\\\"\": unknown"
```

kubelet 日志报错示例：

```
    Sep 09 18:09:09 VM-0-39-ubuntu kubelet[18902]: I0909 18:09:09.449722   18902
    remote_runtime.go:92] RunPodSandbox from runtime service failed: rpc error:
    code = Unknown desc = failed to start sandbox container for pod "osp-xxx-com-
    ljqm19-54bf7678b8-bvz9s": Error response from daemon: oci runtime error:
    container_linux.go:247: starting container process caused
    "process_linux.go:258: applying cgroup configuration for process caused \"mkdir
    /sys/fs/cgroup/memory/kubepods/burstable/podf1bd9e87-1ef2-11e8-afd3-
    fa163ecf2dce/8710c146b3c8b52f5da62e222273703b1e3d54a6a6270a0ea7ce1b194f1b5053:
1.  no space left on device\""
```

新版的内核限制为 `2^31` (可以看成几乎不限制，点我跳转到代码)： `cgroup_idr_alloc()` 传入 end 为 0 到 `idr_alloc()`， 再传给 `idr_alloc_u32()`，end 的值最终被三元运算符 `end>0 ? end-1 : INT_MAX` 转成了 `INT_MAX` 常量，即 `2^31`。所以如果新版内核有泄露问题会更难定位，表现形式会是内存消耗严重，幸运的是新版内核已经修复，推荐 4.3 以上。

规避方案

如果你用的低版本内核(比如 CentOS 7 v3.10 的内核)并且不方便升级内核，可以通过不开启 kmem accounting 来实现规避，但会比较麻烦。

kubelet 和 runc 都会给 memory cgroup 开启 kmem accounting，所以要规避这个问题，就要保证kubelet 和 runc 都别开启 kmem accounting，下面分别进行说明:

runc

runc 在合并 这个PR (2017-02-27) 之后创建的容器都默认开启了 kmem accounting，后来社区也注意到这个问题，并做了比较灵活的修复， PR 1921 给 runc 增加了 “nokmem” 编译选项，缺省的 release 版本没有使用这个选项， 自己使用 nokmem 选项编译 runc 的方法:

```
1.  cd $GO_PATH/src/github.com/opencontainers/runc/
2.  make BUILDTAGS="seccomp nokmem"
```

docker-ce v18.09.1 之后的 runc 默认关闭了 kmem accounting，所以也可以直接升级 docker 到这个版本之后。

kubelet

如果是 1.14 版本及其以上，可以在编译的时候通过 build tag 来关闭 kmem accounting:

```
1.  KUBE_GIT_VERSION=v1.14.1 ./build/run.sh make kubelet GOFLAGS="-tags=nokmem"
```

如果是低版本需要修改代码重新编译。kubelet 在创建 pod 对应的 cgroup 目录时，也会调用

libcontianer 中的代码对 cgroup 做设置，在 `pkg/kubelet/cm/cgroup_manager_linux.go` 的 `Create` 方法中，会调用 `Manager.Apply` 方法，最终调用 `vendor/github.com/opencontainers/runc/libcontainer/cgroups/fs/memory.go` 中的 `MemoryGroup.Apply` 方法，开启 kmem accounting。这里也需要进行处理，可以将这部分代码注释掉然后重新编译 kubelet。

参考资料

- 一行 kubernetes 1.9 代码引发的血案（与 CentOS 7.x 内核兼容性问题）：http://dockone.io/article/4797
- Cgroup泄漏—潜藏在你的集群中：https://tencentcloudcontainerteam.github.io/2018/12/29/cgroup-leaking/

tcp_tw_recycle 引发丢包

`tcp_tw_recycle` 这个内核参数用来快速回收 `TIME_WAIT` 连接，不过如果在 NAT 环境下会引发问题。

RFC1323 中有如下一段描述：

> An additional mechanism could be added to the TCP, a per-host cache of the last timestamp received from any connection. This value could then be used in the PAWS mechanism to reject old duplicate segments from earlier incarnations of the connection, if the timestamp clock can be guaranteed to have ticked at least once since the old connection was open. This would require that the TIME-WAIT delay plus the RTT together must be at least one tick of the sender’s timestamp clock. Such an extension is not part of the proposal of this RFC.

- 大概意思是说TCP有一种行为，可以缓存每个连接最新的时间戳，后续请求中如果时间戳小于缓存的时间戳，即视为无效，相应的数据包会被丢弃。
- Linux是否启用这种行为取决于tcp_timestamps和tcp_tw_recycle，因为tcp_timestamps缺省就是开启的，所以当tcp_tw_recycle被开启后，实际上这种行为就被激活了，当客户端或服务端以NAT方式构建的时候就可能出现问题，下面以客户端NAT为例来说明：
- 当多个客户端通过NAT方式联网并与服务端交互时，服务端看到的是同一个IP，也就是说对服务端而言这些客户端实际上等同于一个，可惜由于这些客户端的时间戳可能存在差异，于是乎从服务端的视角看，便可能出现时间戳错乱的现象，进而直接导致时间戳小的数据包被丢弃。如果发生了此类问题，具体的表现通常是是客户端明明发送的SYN，但服务端就是不响应ACK。
- 在4.12之后的内核已移除tcp_tw_recycle内核参数：
  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4396e46187ca5070219b81773c4e65088dac50cc
  https://github.com/torvalds/linux/commit/4396e46187ca5070219b81773c4e65088dac50cc

使用 oom-guard 在用户态处理 cgroup OOM

背景

由于 linux 内核对 cgroup OOM 的处理，存在很多 bug，经常有由于频繁 cgroup OOM 导致节点故障(卡死， 重启， 进程异常但无法杀死)，于是 TKE 团队开发了 `oom-guard` ，在用户态处理 cgroup OOM 规避了内核 bug。

原理

核心思想是在发生内核 cgroup OOM kill 之前，在用户空间杀掉超限的容器， 减少走到内核 cgroup 内存回收失败后的代码分支从而触发各种内核故障的机会。

threshold notify

参考文档: https://lwn.net/Articles/529927/

`oom-guard` 会给 memory cgroup 设置 threshold notify， 接受内核的通知。

以一个例子来说明阀值计算通知原理: 一个 pod 设置的 memory limit 是 1000M， `oom-guard` 会根据配置参数计算出 margin:

```
1.  margin = 1000M * margin_ratio = 20M // 缺省margin_ratio是0.02
```

margin 最小不小于 mim_margin(缺省1M)， 最大不大于 max_margin(缺省为30M)。如果超出范围，则取 mim_margin 或 max_margin。计算 threshold = limit - margin ，也就是 1000M - 20M = 980M，把 980M 作为阈值设置给内核。当这个 pod 的内存使用量达到 980M 时， `oom-guard` 会收到内核的通知。

在触发阈值之前， `oom-gurad` 会先通过 `memory.force_empty` 触发相关 cgroup 的内存回收。 另外，如果触发阈值时，相关 cgroup 的 memory.stat 显示还有较多 cache， 则不会触发后续处理策略，这样当 cgroup 内存达到 limit 时，会内核会触发内存回收。 这个策略也会造成部分容器内存增长太快时，还是会触发内核 cgroup OOM

达到阈值后的处理策略

通过 `--policy` 参数来控制处理策略。目前有三个策略， 缺省策略是 process。

- `process` : 采用跟内核cgroup OOM killer相同的策略，在该cgroup内部，选择一个 oom_score 得分最高的进程杀掉。 通过 oom-guard 发送 SIGKILL 来杀掉进程

- `container`：在该cgroup下选择一个 docker 容器，杀掉整个容器
- `noop`：只记录日志，并不采取任何措施

事件上报

通过 webhook reporter 上报 k8s event，便于分析统计，使用 `kubectl get event` 可以看到：

```
    LAST SEEN   FIRST SEEN   COUNT     NAME                              KIND
    SUBOBJECT                  TYPE      REASON                    SOURCE
1.  MESSAGE
    14s         14s          1         172.21.16.23.158b732d352bcc31   Node
    guard, 172.21.16.23   {"hostname":"172.21.16.23","timestamp":"2019-03-
    13T07:12:14.561650646Z","oomcgroup":"/sys/fs/cgroup/memory/kubepods/burstable/poc
    06925242d7ea/223d4795cc3b33e28e702f72e0497e1153c4a809de6b4363f27acc12a6781cdb","p
    455f-11e9-a7e5-
    06925242d7ea/223d4795cc3b33e28e702f72e0497e1153c4a809de6b4363f27acc12a6781cdb","t
    ","stats":"cache 20480|rss 205938688|rss_huge 199229440|mapped_file 0|dirty 0|wri
    0|inactive_anon 8192|active_anon 203816960|inactive_file 0|active_file 0|unevicta
    20480|total_rss 205938688|total_rss_huge 199229440|total_mapped_file 0|total_dirt
    104|total_pgfault 2059|total_pgmajfault 0|total_inactive_anon 8192|total_active_a
2.  0|total_unevictable 0|","policy":"Container"}
```

使用方法

部署

保存部署 yaml：`oom-guard.yaml`：

```
1.  apiVersion: v1
2.  kind: ServiceAccount
3.  metadata:
4.    name: oomguard
5.    namespace: kube-system
6.  ---
7.  apiVersion: rbac.authorization.k8s.io/v1
8.  kind: ClusterRoleBinding
9.  metadata:
10.   name: system:oomguard
11. roleRef:
12.   apiGroup: rbac.authorization.k8s.io
```

```
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: oomguard
    namespace: kube-system
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: oom-guard
  namespace: kube-system
  labels:
    app: oom-guard
spec:
  selector:
    matchLabels:
      app: oom-guard
  template:
    metadata:
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ""
      labels:
        app: oom-guard
    spec:
      serviceAccountName: oomguard
      hostPID: true
      hostNetwork: true
      dnsPolicy: ClusterFirst
      containers:
      - name: k8s-event-writer
        image: ccr.ccs.tencentyun.com/paas/k8s-event-writer:v1.6
        resources:
          limits:
            cpu: 10m
            memory: 60Mi
          requests:
            cpu: 10m
            memory: 30Mi
        args:
        - --logtostderr
        - --unix-socket=true
```

```
        env:
          - name: NODE_NAME
            valueFrom:
              fieldRef:
                fieldPath: status.hostIP
        volumeMounts:
        - name: unix
          mountPath: /unix
      - name: oomguard
        image: ccr.ccs.tencentyun.com/paas/oomguard:nosoft-v2
        imagePullPolicy: Always
        securityContext:
          privileged: true
        resources:
          limits:
            cpu: 10m
            memory: 60Mi
          requests:
            cpu: 10m
            memory: 30Mi
        volumeMounts:
        - name: cgroupdir
          mountPath: /sys/fs/cgroup/memory
        - name: unix
          mountPath: /unix
        - name: kmsg
          mountPath: /dev/kmsg
          readOnly: true
        command: ["/oom-guard"]
        args:
        - --v=2
        - --logtostderr
        - --root=/sys/fs/cgroup/memory
        - --walkIntervalSeconds=277
        - --inotifyResetSeconds=701
        - --port=0
        - --margin-ratio=0.02
        - --min-margin=1
        - --max-margin=30
        - --guard-ms=50
        - --policy=container
        - --openSoftLimit=false
```

```
        - --webhook-url=http://localhost/message
        env:
          - name: NODE_NAME
            valueFrom:
              fieldRef:
                fieldPath: status.hostIP
      volumes:
      - name: cgroupdir
        hostPath:
          path: /sys/fs/cgroup/memory
      - name: unix
        emptyDir: {}
      - name: kmsg
        hostPath:
          path: /dev/kmsg
```

一键部署：

```
kubectl apply -f oom-guard.yaml
```

检查是否部署成功：

```
$ kubectl -n kube-system get ds oom-guard
NAME         DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE   NODE
SELECTOR   AGE
oom-guard   2         2         2         2            2           <none>
6m
```

其中 **AVAILABLE** 数量跟节点数一致，说明所有节点都已经成功运行了 `oom-guard` 。

查看 oom-guard 日志

```
kubectl -n kube-system logs oom-guard-xxxxx oomguard
```

查看 oom 相关事件

```
kubectl get events |grep CgroupOOM
kubectl get events |grep SystemOOM
kubectl get events |grep OomGuardKillContainer
kubectl get events |grep OomGuardKillProcess
```

卸载

```
1.  kubectl delete -f oom-guard.yaml
```

这个操作可能有点慢，如果一直不返回（有节点 NotReady 时可能会卡住），`ctrl+C` 终止，然后执行下面的脚本：

```
1.  for pod in `kubectl get pod -n kube-system | grep oom-guard | awk '{print $1}'`
2.  do
3.   kubectl delete pod $pod -n kube-system --grace-period=0 --force
4.  done
```

检查删除操作是否成功

```
1.  kubectl -n kube-system get ds oom-guard
```

提示 `...not found` 就说明删除成功了

关于开源

当前 `oom-gaurd` 暂未开源，正在做大量生产试验，后面大量反馈效果统计比较好的时候会考虑开源出来。

no space left on device

- 有时候节点 NotReady， kubelet 日志报 `no space left on device`
- 有时候创建 Pod 失败， `describe pod` 看 event 报 `no space left on device`

出现这种错误有很多中可能原因，下面我们来根据现象找对应原因。

inotify watch 耗尽

节点 NotReady，kubelet 启动失败，看 kubelet 日志：

```
    Jul 18 15:20:58 VM_16_16_centos kubelet[11519]: E0718 15:20:58.280275   11519 raw
    "/sys/fs/cgroup/memory/kubepods": inotify_add_watch /sys/fs/cgroup/memory/kubepod
1.  52540048533c/6e85761a30707b43ed874e0140f58839618285fc90717153b3cbe7f91629ef5a: no
```

系统调用 `inotify_add_watch` 失败，提示 `no space left on device`， 这是因为系统上进程 watch 文件目录的总数超出了最大限制，可以修改内核参数调高限制，详细请参考本书内核相关章节的 inotify watch 耗尽

cgroup 泄露

查看当前 cgroup 数量：

```
1.  $ cat /proc/cgroups | column -t
2.  #subsys_name  hierarchy  num_cgroups  enabled
3.  cpuset        5          29           1
4.  cpu           7          126          1
5.  cpuacct       7          126          1
6.  memory        9          127          1
7.  devices       4          126          1
8.  freezer       2          29           1
9.  net_cls       6          29           1
10. blkio         10         126          1
11. perf_event    3          29           1
12. hugetlb       11         29           1
13. pids          8          126          1
14. net_prio      6          29           1
```

cgroup 子系统目录下面所有每个目录及其子目录都认为是一个独立的 cgroup，所以也可以在文件系

统中统计目录数来获取实际 cgroup 数量，通常跟 `/proc/cgroups` 里面看到的应该一致：

```
$ find -L /sys/fs/cgroup/memory -type d | wc -l
127
```

当 cgroup 泄露发生时，这里的数量就不是真实的了，低版本内核限制最大 65535 个 cgroup，并且开启 kmem 删除 cgroup 时会泄露，大量创建删除容器后泄露了许多 cgroup，最终总数达到 65535，新建容器创建 cgroup 将会失败，报 `no space left on device`

详细请参考本书内核相关章节的 cgroup 泄露

磁盘被写满(TODO)

- pod 启动失败 (状态 `CreateContainerError`)

```
csi-cephfsplugin-27znb                          0/2     CreateContainerError
167        17h
```

```
  Warning  Failed   5m1s (x3397 over 17h)  kubelet, ip-10-0-151-35.us-west-2.comp
events): Error: container create failed: container_linux.go:336: starting contair
"process_linux.go:399: container init caused \"rootfs_linux.go:58: mounting \\\"/
\\\"/var/lib/containers/storage/overlay/051e985771cc69f3f699895a1dada9ef6483e912b
\\\"/var/lib/containers/storage/overlay/051e985771cc69f3f699895a1dada9ef6483e912b
caused \\\"no space left on device\\\"\""
```

案例分享

驱逐导致服务中断

TODO 优化

案例

TKE 一客户的某个节点有问题，无法挂载nfs，通过新加节点，驱逐故障节点的 pod 来规避，但导致了业务 10min 服务不可用，排查发现其它节点 pod 很多集体出现了重启，主要是连不上 kube-dns 无法解析 service，业务调用不成功，从而对外表现为服务不可用。

为什么会中断？驱逐的原理是先封锁节点，然后将旧的 node 上的 pod 删除，replicaset 控制器检测到 pod 减少，会重新创建一个 pod，调度到新的 node上，这个过程是先删除，再创建，并非是滚动更新，因此更新过程中，如果一个deployment的所有 pod 都在被驱逐的节点上，则可能导致该服务不可用。

那为什么会影响其它 pod？分析kubelet日志，kube-dns 有两个副本，都在这个被驱逐的节点上，所以驱逐的时候 kube-dns 不通，影响了其它 pod 解析 service，导致服务集体不可用。

那为什么会中断这么久？通常在新的节点应该很会快才是，通过进一步分析新节点的 kubelet 日志，发现 kube-dns 从拉镜像到容器启动之间花了很长时间，检查节点上的镜像发现有很多大镜像(1~2GB)，猜测是拉取镜像有并发限制，kube-dns 的镜像虽小，但在排队等大镜像下载完，检查 kubelet 启动参数，确实有 `--registry-burst` 这个参数控制镜像下载并发数限制。但最终发现其实应该是 `--serialize-image-pulls` 这个参数导致的，kubelet 启动参数没有指定该参数，而该参数默认值为 true，即默认串行下载镜像，不并发下载，所以导致镜像下载排队，是的 kube-dns 延迟了很长时间才启动。

解决方案

- 避免服务单点故障，多副本，并加反亲和性
- 设置 preStop hook 与 readinessProbe，更新路由规则

DNS 5 秒延时

延时现象

客户反馈从 pod 中访问服务时，总是有些请求的响应时延会达到5秒。正常的响应只需要毫秒级别的时延。

抓包

- 通过 `nsenter` 进入 pod netns，使用节点上的 tcpdump 抓 pod 中的包 (抓包方法参考这里)，发现是有的 DNS 请求没有收到响应，超时 5 秒后，再次发送 DNS 请求才成功收到响应。
- 在 kube-dns pod 抓包，发现是有 DNS 请求没有到达 kube-dns pod，在中途被丢弃了。

为什么是 5 秒？ `man resolv.conf` 可以看到 glibc 的 resolver 的缺省超时时间是 5s:

```
1.  timeout:n
           Sets the amount of time the resolver will wait for a response from a
    remote name server before retrying the query via a different name server.
2.  Measured in seconds, the default is RES_TIMEOUT (currently  5,  see
3.          <resolv.h>).  The value for this option is silently capped to 30.
```

丢包原因

经过搜索发现这是一个普遍问题。

根本原因是内核 conntrack 模块的 bug，netfilter 做 NAT 时可能发生资源竞争导致部分报文丢弃。

Weave works的工程师 Martynas Pumputis 对这个问题做了很详细的分析：Racy conntrack and DNS lookup timeouts

相关结论：

- 只有多个线程或进程，并发从同一个 socket 发送相同五元组的 UDP 报文时，才有一定概率会发生
- glibc, musl(alpine linux的libc库)都使用 “parallel query”, 就是并发发出多个查询请求，因此很容易碰到这样的冲突，造成查询请求被丢弃
- 由于 ipvs 也使用了 conntrack, 使用 kube-proxy 的 ipvs 模式，并不能避免这个问题

问题的根本解决

Martynas 向内核提交了两个 patch 来 fix 这个问题，不过他说如果集群中有多个DNS server 的情况下，问题并没有完全解决。

其中一个 patch 已经在 2018-7-18 被合并到 linux 内核主线中: netfilter: nf_conntrack: resolve clash for matching conntracks

目前只有4.19.rc 版本包含这个patch。

规避办法

规避方案一：使用TCP发送DNS请求

由于TCP没有这个问题，有人提出可以在容器的resolv.conf中增加 `options use-vc` ，强制glibc使用TCP协议发送DNS query。下面是这个man resolv.conf中关于这个选项的说明:

```
use-vc (since glibc 2.14)
                    Sets RES_USEVC in _res.options.  This option forces the
                    use of TCP for DNS resolutions.
```

笔者使用镜像”busybox:1.29.3-glibc” (libc 2.24) 做了试验，并没有见到这样的效果，容器仍然是通过UDP发送DNS请求。

规避方案二：避免相同五元组DNS请求的并发

resolv.conf还有另外两个相关的参数:

- single-request-reopen (since glibc 2.9)
- single-request (since glibc 2.10)

man resolv.conf中解释如下:

```
single-request-reopen (since glibc 2.9)
                    Sets RES_SNGLKUPREOP in _res.options.  The resolver
                    uses the same socket for the A and AAAA requests.  Some
                    hardware mistakenly sends back only one reply.  When
                    that happens the client system will sit and wait for
                    the second reply.  Turning this option on changes this
                    behavior so that if two requests from the same port are
                    not handled correctly it will close the socket and open
                    a new one before sending the second request.
```

```
10.
11.  single-request (since glibc 2.10)
12.                       Sets RES_SNGLKUP in _res.options.  By default, glibc
13.                       performs IPv4 and IPv6 lookups in parallel since
14.                       version 2.9.  Some appliance DNS servers cannot handle
15.                       these queries properly and make the requests time out.
16.                       This option disables the behavior and makes glibc
17.                       perform the IPv6 and IPv4 requests sequentially (at the
18.                       cost of some slowdown of the resolving process).
```

用自己的话解释下：

- `single-request-reopen`：发送 A 类型请求和 AAAA 类型请求使用不同的源端口，这样两个请求在 conntrack 表中不占用同一个表项，从而避免冲突
- `single-request`：避免并发，改为串行发送 A 类型和 AAAA 类型请求，没有了并发，从而也避免了冲突

要给容器的 `resolv.conf` 加上 options 参数，有几个办法：

1) 在容器的 “ENTRYPOINT” 或者 “CMD” 脚本中，执行 /bin/echo ‘options single-request-reopen’ >> /etc/resolv.conf

2) 在 pod 的 postStart hook 中：

```
1.          lifecycle:
2.            postStart:
3.              exec:
4.                command:
5.                - /bin/sh
6.                - -c
7.                - "/bin/echo 'options single-request-reopen' >> /etc/resolv.conf"
```

3) 使用 template.spec.dnsConfig (k8s v1.9 及以上才支持)：

```
1.    template:
2.      spec:
3.        dnsConfig:
4.          options:
5.            - name: single-request-reopen
```

4) 使用 ConfigMap 覆盖 pod 里面的 /etc/resolv.conf：

configmap:

```
apiVersion: v1
data:
  resolv.conf: |
    nameserver 1.2.3.4
    search default.svc.cluster.local svc.cluster.local cluster.local
ec2.internal
    options ndots:5 single-request-reopen timeout:1
kind: ConfigMap
metadata:
  name: resolvconf
```

pod spec:

```
        volumeMounts:
        - name: resolv-conf
          mountPath: /etc/resolv.conf
          subPath: resolv.conf
...

      volumes:
      - name: resolv-conf
        configMap:
          name: resolvconf
          items:
          - key: resolv.conf
            path: resolv.conf
```

5) 使用 MutatingAdmissionWebhook

MutatingAdmissionWebhook 是 1.9 引入的 Controller，用于对一个指定的 Resource 的操作之前，对这个 resource 进行变更。 istio 的自动 sidecar注入就是用这个功能来实现的。我们也可以通过 MutatingAdmissionWebhook，来自动给所有POD，注入以上3)或者4)所需要的相关内容。

以上方法中， 1)和2)都需要修改镜像， 3)和4)则只需要修改POD的spec， 能适用于所有镜像。不过还是有不方便的地方：

- 每个工作负载的yaml都要做修改，比较麻烦
- 对于通过helm创建的工作负载，需要修改helm charts

方法5)对集群使用者最省事，照常提交工作负载即可。不过初期需要一定的开发工作量。

规避方案三：使用本地DNS缓存

容器的DNS请求都发往本地的DNS缓存服务(dnsmasq, nscd等)，不需要走DNAT，也不会发生conntrack冲突。另外还有个好处，就是避免DNS服务成为性能瓶颈。

使用本地DNS缓存有两种方式：

- 每个容器自带一个DNS缓存服务
- 每个节点运行一个DNS缓存服务，所有容器都把本节点的DNS缓存作为自己的 nameserver

从资源效率的角度来考虑的话，推荐后一种方式。官方也意识到了这个问题比较常见，给出了coredns 以 cache 模式作为 daemonset 部署的解决方案：https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/

实施办法

条条大路通罗马，不管怎么做，最终到达上面描述的效果即可。

POD中要访问节点上的DNS缓存服务，可以使用节点的IP。 如果节点上的容器都连在一个虚拟bridge上， 也可以使用这个bridge的三层接口的IP(在TKE中，这个三层接口叫cbr0)。 要确保DNS缓存服务监听这个地址。

如何把POD的/etc/resolv.conf中的nameserver设置为节点IP呢？

一个办法，是设置 POD.spec.dnsPolicy 为 “Default”， 意思是POD里面的 /etc/resolv.conf， 使用节点上的文件。缺省使用节点上的 /etc/resolv.conf(如果kubelet通过参数—resolv-conf指定了其他文件，则使用—resolv-conf所指定的文件)。

另一个办法，是给每个节点的kubelet指定不同的—cluster-dns参数，设置为节点的IP，POD.spec.dnsPolicy仍然使用缺省值”ClusterFirst”。 kops项目甚至有个issue在讨论如何在部署集群时设置好—cluster-dns指向节点IP：https://github.com/kubernetes/kops/issues/5584

参考资料

- Racy conntrack and DNS lookup timeouts: https://www.weave.works/blog/racy-conntrack-and-dns-lookup-timeouts
- 5 – 15s DNS lookups on Kubernetes? : https://blog.quentin-machu.fr/2018/06/24/5-15s-dns-lookups-on-kubernetes/
- DNS intermittent delays of 5s: https://github.com/kubernetes/kubernetes/issues/56903
- 记一次Docker/Kubernetes上无法解释的连接超时原因探寻之旅：

https://mp.weixin.qq.com/s/VYBs8iqf0HsNg9WAxktzYQ

ARP 缓存爆满导致健康检查失败

案例

TKE 一用户某集群节点数 1200+，用户监控方案是 daemonset 部署 node-exporter 暴露节点监控指标，使用 hostNework 方式，statefulset 部署 promethues 且仅有一个实例，落在了一个节点上，promethues 请求所有节点 node-exporter 获取节点监控指标，也就是或扫描所有节点，导致 arp cache 需要存所有 node 的记录，而节点数 1200+，大于了 `net.ipv4.neigh.default.gc_thresh3` 的默认值 1024，这个值是个硬限制，arp cache记录数大于这个就会强制触发 gc，所以会造成频繁gc，当有数据包发送会查本地 arp，如果本地没找到 arp 记录就会判断当前 arp cache 记录数+1是否大于 gc_thresh3，如果没有就会广播 arp 查询 mac 地址，如果大于了就直接报 `arp_cache: neighbor table overflow!` ，并且放弃 arp 请求，无法获取 mac 地址也就无法知道探测报文该往哪儿发(即便就在本机某个 veth pair)，kubelet 对本机 pod 做存活检查发 arp 查 mac 地址，在 arp cahce 找不到，由于这时 arp cache已经满了，刚要 gc 但还没做所以就只有报错丢包，导致存活检查失败重启 pod

解决方案

调整部分节点内核参数，将 arp cache 的 gc 阀值调高 (`/etc/sysctl.conf`):

```
net.ipv4.neigh.default.gc_thresh1 = 80000
net.ipv4.neigh.default.gc_thresh2 = 90000
net.ipv4.neigh.default.gc_thresh3 = 100000
```

并给 node 打下label，修改 pod spec，加下 nodeSelector 或者 nodeAffnity，让 pod 只调度到这部分改过内核参数的节点，更多请参考本书 处理实践: arp_cache 溢出

跨 VPC 访问 NodePort 经常超时

现象：从 VPC a 访问 VPC b 的 TKE 集群的某个节点的 NodePort，有时候正常，有时候会卡住直到超时。

原因怎么查？

当然是先抓包看看啦，抓 server 端 NodePort 的包，发现异常时 server 能收到 SYN，但没响应 ACK：

```
12:01:30.862212 IP (tos 0x10, ttl 64, id 12665, offset 0, flags [DF], proto TCP (6), length 60)
    10.10.1.198.58694 > 10.52.16.3.30347: Flags [S], cksum 0x9316 (correct), seq 1872604432, win 29200, options [mss 142
        0x0000:  0000 0001 0006 feee 2fc6 b077 0000 0800  ......../..w....
        0x0010:  4510 003c 3179 4000 4006 e32c 0a0a 01c6  E..<1y@.@..,....
        0x0020:  0a34 1003 e546 768b 6f9d ad10 0000 0000  .4...Fv.o.......
        0x0030:  a002 7210 9316 0000 0204 0590 0402 080a  ..r.............
        0x0040:  080d 9c69 0000 0000 0103 0307 0000 0000  ...i............
        0x0050:  0000 0000 0000 0000 0000 0000            ............
12:01:31.864659 IP (tos 0x10, ttl 64, id 12666, offset 0, flags [DF], proto TCP (6), length 60)
    10.10.1.198.58694 > 10.52.16.3.30347: Flags [S], cksum 0x8f2b (correct), seq 1872604432, win 29200, options [mss 142
        0x0000:  0000 0001 0006 feee 2fc6 b077 0000 0800  ......../..w....
        0x0010:  4510 003c 317a 4000 4006 e32b 0a0a 01c6  E..<1z@.@..+....
        0x0020:  0a34 1003 e546 768b 6f9d ad10 0000 0000  .4...Fv.o.......
        0x0030:  a002 7210 8f2b 0000 0204 0590 0402 080a  ..r..+..........
        0x0040:  080d a054 0000 0000 0103 0307 0000 0000  ...T............
        0x0050:  0000 0000 0000 0000 0000 0000            ............
12:01:33.868318 IP (tos 0x10, ttl 64, id 12667, offset 0, flags [DF], proto TCP (6), length 60)
    10.10.1.198.58694 > 10.52.16.3.30347: Flags [S], cksum 0x8757 (correct), seq 1872604432, win 29200, options [mss 142
        0x0000:  0000 0001 0006 feee 2fc6 b077 0000 0800  ......../..w....
        0x0010:  4510 003c 317b 4000 4006 e32a 0a0a 01c6  E..<1{@.@..*....
        0x0020:  0a34 1003 e546 768b 6f9d ad10 0000 0000  .4...Fv.o.......
        0x0030:  a002 7210 8757 0000 0204 0590 0402 080a  ..r..W..........
        0x0040:  080d a828 0000 0000 0103 0307 0000 0000  ...(............
        0x0050:  0000 0000 0000 0000 0000 0000            ............
```

反复执行 `netstat -s | grep LISTEN` 发现 SYN 被丢弃数量不断增加：

```
[root@VM_16_3_centos ~]# netstat -s | grep LISTEN
    1474222 SYNs to LISTEN sockets dropped
[root@VM_16_3_centos ~]# netstat -s | grep LISTEN
    1474222 SYNs to LISTEN sockets dropped
[root@VM_16_3_centos ~]# netstat -s | grep LISTEN
    1474222 SYNs to LISTEN sockets dropped
[root@VM_16_3_centos ~]# netstat -s | grep LISTEN
    1474225 SYNs to LISTEN sockets dropped
[root@VM_16_3_centos ~]# netstat -s | grep LISTEN
    1474225 SYNs to LISTEN sockets dropped
[root@VM_16_3_centos ~]# netstat -s | grep LISTEN
    1474226 SYNs to LISTEN sockets dropped
[root@VM_16_3_centos ~]#
```

分析：

- 两个VPC之间使用对等连接打通的，CVM 之间通信应该就跟在一个内网一样可以互通。
- 为什么同一 VPC 下访问没问题，跨 VPC 有问题？两者访问的区别是什么？

再仔细看下 client 所在环境，发现 client 是 VPC a 的 TKE 集群节点，捋一下：

- client 在 VPC a 的 TKE 集群的节点
- server 在 VPC b 的 TKE 集群的节点

因为 TKE 集群中有个叫 `ip-masq-agent` 的 daemonset，它会给 node 写 iptables 规则，默认 SNAT 目的 IP 是 VPC 之外的报文，所以 client 访问 server 会做 SNAT，也就是这里跨 VPC 相比同 VPC 访问 NodePort 多了一次 SNAT，如果是因为多了一次 SNAT 导致的这个问题，直觉告诉我这个应该跟内核参数有关，因为是 server 收到包没回包，所以应该是 server 所在 node 的内核参数问题，对比这个 node 和 普通 TKE node 的默认内核参数，发现这个 node `net.ipv4.tcp_tw_recycle = 1`，这个参数默认是关闭的，跟用户沟通后发现这个内核参数确实在做压测的时候调整过。

解释一下，TCP 主动关闭连接的一方在发送最后一个 ACK 会进入 `TIME_AWAIT` 状态，再等待 2 个 MSL 时间后才会关闭(因为如果 server 没收到 client 第四次挥手确认报文，server 会重发第三次挥手 FIN 报文，所以 client 需要停留 2 MSL的时长来处理可能会重复收到的报文段；同时等待 2 MSL 也可以让由于网络不通畅产生的滞留报文失效，避免新建立的连接收到之前旧连接的报文)，了解更详细的过程请参考 TCP 四次挥手。

参数 `tcp_tw_recycle` 用于快速回收 `TIME_AWAIT` 连接，通常在增加连接并发能力的场景会

开启，比如发起大量短连接，快速回收可避免 `tw_buckets` 资源耗尽导致无法建立新连接（`time wait bucket table overflow`）

查得 `tcp_tw_recycle` 有个坑，在 RFC1323 有段描述：

> An additional mechanism could be added to the TCP, a per-host cache of the last timestamp received from any connection. This value could then be used in the PAWS mechanism to reject old duplicate segments from earlier incarnations of the connection, if the timestamp clock can be guaranteed to have ticked at least once since the old connection was open. This would require that the TIME-WAIT delay plus the RTT together must be at least one tick of the sender's timestamp clock. Such an extension is not part of the proposal of this RFC.

大概意思是说 TCP 有一种行为，可以缓存每个连接最新的时间戳，后续请求中如果时间戳小于缓存的时间戳，即视为无效，相应的数据包会被丢弃。

Linux 是否启用这种行为取决于 `tcp_timestamps` 和 `tcp_tw_recycle`，因为 `tcp_timestamps` 缺省开启，所以当 `tcp_tw_recycle` 被开启后，实际上这种行为就被激活了，当客户端或服务端以 `NAT` 方式构建的时候就可能出现问题。

当多个客户端通过 NAT 方式联网并与服务端交互时，服务端看到的是同一个 IP，也就是说对服务端而言这些客户端实际上等同于一个，可惜由于这些客户端的时间戳可能存在差异，于是乎从服务端的视角看，便可能出现时间戳错乱的现象，进而直接导致时间戳小的数据包被丢弃。如果发生了此类问题，具体的表现通常是是客户端明明发送的 SYN，但服务端就是不响应 ACK。

回到我们的问题上，client 所在节点上可能也会有其它 pod 访问到 server 所在节点，而它们都被 SNAT 成了 client 所在节点的 NODE IP，但时间戳存在差异，server 就会看到时间戳错乱，因为开启了 `tcp_tw_recycle` 和 `tcp_timestamps` 激活了上述行为，就丢掉了比缓存时间戳小的报文，导致部分 SYN 被丢弃，这也解释了为什么之前我们抓包发现异常时 server 收到了 SYN，但没有响应 ACK，进而说明为什么 client 的请求部分会卡住直到超时。

由于 `tcp_tw_recycle` 坑太多，在内核 4.12 之后已移除：remove tcp_tw_recycle

访问 externalTrafficPolicy 为 Local 的 Service 对应 LB 有时超时

现象：用户在 TKE 创建了公网 LoadBalancer 类型的 Service，externalTrafficPolicy 设为了 Local，访问这个 Service 对应的公网 LB 有时会超时。

externalTrafficPolicy 为 Local 的 Service 用于在四层获取客户端真实源 IP，官方参考文档：Source IP for Services with Type=LoadBalancer

TKE 的 LoadBalancer 类型 Service 实现是使用 CLB 绑定所有节点对应 Service 的 NodePort，CLB 不做 SNAT，报文转发到 NodePort 时源 IP 还是真实的客户端 IP，如果 NodePort 对应 Service 的 externalTrafficPolicy 不是 Local 的就会做 SNAT，到 pod 时就看不到客户端真实源 IP 了，但如果是 Local 的话就不做 SNAT，如果本机 node 有这个 Service 的 endpoint 就转到对应 pod，如果没有就直接丢掉，因为如果转到其它 node 上的 pod 就必须要做 SNAT，不然无法回包，而 SNAT 之后就无法获取真实源 IP 了。

LB 会对绑定节点的 NodePort 做健康检查探测，检查 LB 的健康检查状态：发现这个 NodePort 的所有节点都不健康 !!!

那么问题来了：

1. 为什么会全不健康，这个 Service 有对应的 pod 实例，有些节点上是有 endpoint 的，为什么它们也不健康?
2. LB 健康检查全不健康，但是为什么有时还是可以访问后端服务?

跟 LB 的同学确认：如果后端 rs 全不健康会激活 LB 的全死全活逻辑，也就是所有后端 rs 都可以转发。

那么有 endpoint 的 node 也是不健康这个怎么解释?

在有 endpoint 的 node 上抓 NodePort 的包：发现很多来自 LB 的 SYN，但是没有响应 ACK。

看起来报文在哪被丢了，继续抓下 cbr0 看下：发现没有来自 LB 的包，说明报文在 cbr0 之前被丢了。

再观察用户集群环境信息：

1. k8s 版本1.12
2. 启用了 ipvs
3. 只有 local 的 service 才有异常

尝试新建一个 1.12 启用 ipvs 和一个没启用 ipvs 的测试集群。也都创建 Local 的

LoadBalancer Service，发现启用 ipvs 的测试集群复现了那个问题，没启用 ipvs 的集群没这个问题。

再尝试创建 1.10 的集群，也启用 ipvs，发现没这个问题。

看起来跟集群版本和是否启用 ipvs 有关。

1.12 对比 1.10 启用 ipvs 的集群：1.12 的会将 LB 的 `EXTERNAL-IP` 绑到 `kube-ipvs0` 上，而 1.10 的不会：

```
$ ip a show kube-ipvs0 | grep -A2 170.106.134.124
    inet 170.106.134.124/32 brd 170.106.134.124 scope global kube-ipvs0
       valid_lft forever preferred_lft forever
```

- 170.106.134.124 是 LB 的公网 IP
- 1.12 启用 ipvs 的集群将 LB 的公网 IP 绑到了 `kube-ipvs0` 网卡上

`kube-ipvs0` 是一个 dummy interface，实际不会接收报文，可以看到它的网卡状态是 DOWN，主要用于绑 ipvs 规则的 VIP，因为 ipvs 主要工作在 netfilter 的 INPUT 链，报文通过 PREROUTING 链之后需要决定下一步该进入 INPUT 还是 FORWARD 链，如果是本机 IP 就会进入 INPUT，如果不是就会进入 FORWARD 转发到其它机器。所以 k8s 利用 `kube-ipvs0` 这个网卡将 service 相关的 VIP 绑在上面以便让报文进入 INPUT 进而被 ipvs 转发。

当 IP 被绑到 `kube-ipvs0` 上，内核会自动将上面的 IP 写入 local 路由：

```
$ ip route show table local | grep 170.106.134.124
local 170.106.134.124 dev kube-ipvs0  proto kernel  scope host  src
170.106.134.124
```

内核认为在 local 路由里的 IP 是本机 IP，而 linux 默认有个行为：忽略任何来自非回环网卡并且源 IP 是本机 IP 的报文。而 LB 的探测报文源 IP 就是 LB IP，也就是 Service 的 `EXTERNAL-IP` 猜想就是因为这个 IP 被绑到 `kube-ipvs0`，自动加进 local 路由导致内核直接忽略了 LB 的探测报文。

带着猜想做实现， 试一下将 LB IP 从 local 路由中删除：

```
ip route del table local local 170.106.134.124 dev kube-ipvs0  proto kernel
scope host  src 170.106.134.124
```

发现这个 node 的在 LB 的健康检查的状态变成健康了！看来就是因为这个 LB IP 被绑到 `kube-ipvs0` 导致内核忽略了来自 LB 的探测报文，然后 LB 收不到回包认为不健康。

那为什么其它厂商没反馈这个问题？应该是 LB 的实现问题，腾讯云的公网 CLB 的健康探测报文源

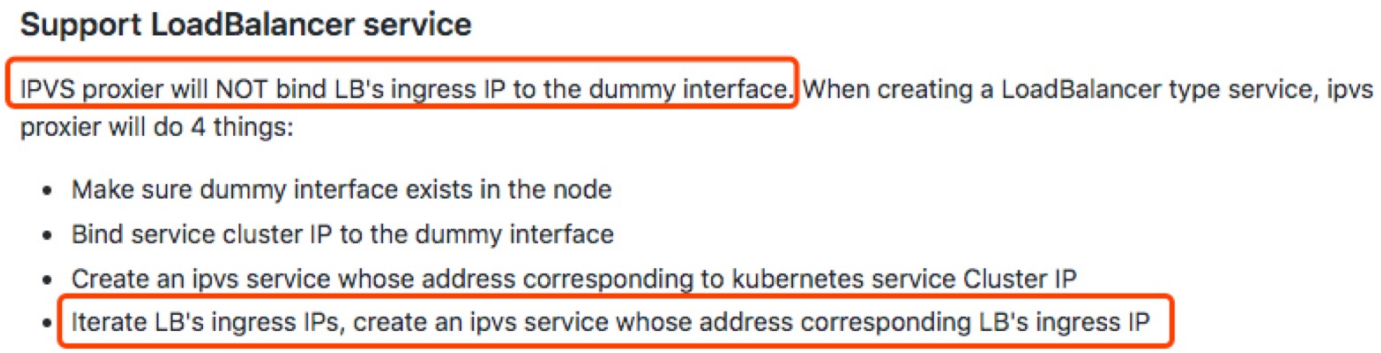
IP 就是 LB 的公网 IP，而大多数厂商的 LB 探测报文源 IP 是保留 IP 并非 LB 自身的 VIP。

如何解决呢？发现一个内核参数：accept_local 可以让 linux 接收源 IP 是本机 IP 的报文。

试了开启这个参数，确实在 cbr0 收到来自 LB 的探测报文了，说明报文能被 pod 收到，但抓 eth0 还是没有给 LB 回包。

为什么没有回包？分析下五元组，要给 LB 回包，那么 `目的IP:目的Port` 必须是探测报文的 `源IP:源Port`，所以目的 IP 就是 LB IP，由于容器不在主 netns，发包经过 veth pair 到 cbr0 之后需要再经过 netfilter 处理，报文进入 PREROUTING 链然后发现目的 IP 是本机 IP，进入 INPUT 链，所以报文就出不去了。再分析下进入 INPUT 后会怎样，因为目的 Port 跟 LB 探测报文源 Port 相同，是一个随机端口，不在 Service 的端口列表，所以没有对应的 IPVS 规则，IPVS 也就不会转发它，而 `kube-ipvs0` 上虽然绑了这个 IP，但它是一个 dummy interface，不会收包，所以报文最后又被忽略了。

再看看为什么 1.12 启用 ipvs 会绑 `EXTERNAL-IP` 到 `kube-ipvs0`，翻翻 k8s 的 kube-proxy 支持 ipvs 的 proposal，发现有个地方说法有点漏洞：



LB 类型 Service 的 status 里有 ingress IP，实际就是 `kubectl get service` 看到的 `EXTERNAL-IP`，这里说不会绑定这个 IP 到 kube-ipvs0，但后面又说会给它创建 ipvs 规则，既然没有绑到 `kube-ipvs0`，那么这个 IP 的报文根本不会进入 INPUT 被 ipvs 模块转发，创建的 ipvs 规则也是没用的。

后来找到作者私聊，思考了下，发现设计上确实有这个问题。

看了下 1.10 确实也是这么实现的，但是为什么 1.12 又绑了这个 IP 呢？调研后发现是因为 #59976 这个 issue 发现一个问题，后来引入 #63066 这个 PR 修复的，而这个 PR 的行为就是让 LB IP 绑到 `kube-ipvs0`，这个提交影响 1.11 及其之后的版本。

#59976 的问题是因为没绑 LB IP到 `kube-ipvs0` 上，在自建集群使用 `MetalLB` 来实现 LoadBalancer 类型的 Service，而有些网络环境下，pod 是无法直接访问 LB 的，导致 pod 访问 LB IP 时访问不了，而如果将 LB IP 绑到 `kube-ipvs0` 上就可以通过 ipvs 转发到 LB 类型 Service 对应的 pod 去， 而不需要真正经过 LB，所以引入了 #63066 这个PR。

临时方案：将 #63066 这个 PR 的更改回滚下，重新编译 kube-proxy，提供升级脚本升级存量

kube-proxy。

如果是让 LB 健康检查探测支持用保留 IP 而不是自身的公网 IP ，也是可以解决，但需要跨团队合作，而且如果多个厂商都遇到这个问题，每家都需要为解决这个问题而做开发调整，代价较高，所以长期方案需要跟社区沟通一起推进，所以我提了 issue，将问题描述的很清楚：#79783

小思考：为什么 CLB 可以不做 SNAT ？回包目的 IP 就是真实客户端 IP，但客户端是直接跟 LB IP 建立的连接，如果回包不经过 LB 是不可能发送成功的呀。

是因为 CLB 的实现是在母机上通过隧道跟 CVM 互联的，多了一层封装，回包始终会经过 LB。

就是因为 CLB 不做 SNAT，正常来自客户端的报文是可以发送到 nodeport，但健康检查探测报文由于源 IP 是 LB IP 被绑到 `kube-ipvs0` 导致被忽略，也就解释了为什么健康检查失败，但通过LB能访问后端服务，只是有时会超时。那么如果要做 SNAT 的 LB 岂不是更糟糕，所有报文都变成 LB IP，所有报文都会被忽略?

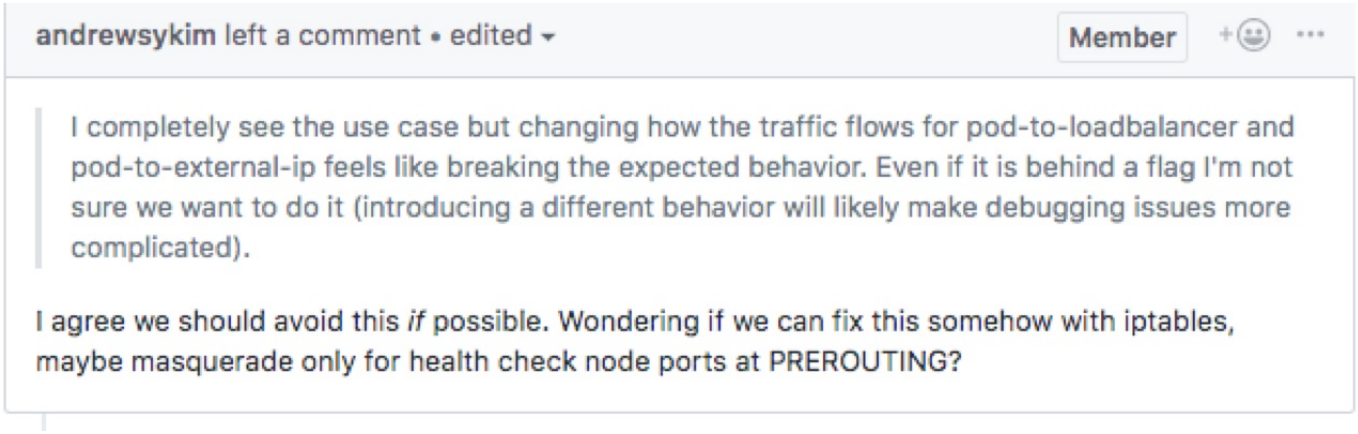
我提的 issue 有回复指出，AWS 的 LB 会做 SNAT，但它们不将 LB 的 IP 写到 Service 的 Status 里，只写了 hostname，所以也不会绑 LB IP 到 `kube-ipvs0` ：

> The problem you describe makes perfect sense. It would also impact AWS because packets are SNAT to the loadbalancer IP. However they don't set this `EXTERNAL-IP` in the status field but only the load-balancer name:
>
> ```
> ingress:
> - hostname: internal-xxxx.us-east-1.elb.amazonaws.com
> ```

但是只写 hostname 也得 LB 支持自动绑域名解析，并且个人觉得只写 hostname 很别扭，通过 `kubectl get svc` 或者其它 k8s 管理系统无法直接获取 LB IP，这不是一个好的解决方法。

我提了 #79976 这个 PR 可以解决问题：给 kube-proxy 加 `--exclude-external-ip` 这个 flag 控制是否为 LB IP创建 ipvs 规则和绑定 `kube-ipvs0` 。

但有人担心增加 kube-proxy flag 会增加 kube-proxy 的调试复杂度，看能否在 iptables 层面解决：



andrewsykim left a comment • edited ▾ Member

> I completely see the use case but changing how the traffic flows for pod-to-loadbalancer and pod-to-external-ip feels like breaking the expected behavior. Even if it is behind a flag I'm not sure we want to do it (introducing a different behavior will likely make debugging issues more complicated).

I agree we should avoid this *if* possible. Wondering if we can fix this somehow with iptables, maybe masquerade only for health check node ports at PREROUTING?

仔细一想，确实可行，打算有空实现下，重新提个 PR：

> I agree we should avoid this *if* possible. Wondering if we can fix this somehow with iptables, maybe masquerade only for health check node ports at PREROUTING?

@andrewsykim Yeah, I think this is feasible, but we should consider this: Linux will ignore all packets from the non-loopback interface which source IP is the local IP by default. If we bind LB's IP to the kube-ipvs0, the LB's IP will become local IP, that means we need to change the linux's default behaviour, this could be done by enabling accept_local parameter which has been backported till kernel v2.6.33, we can enable it during kube-proxy startup. And I think we should masquerade LB's IP not just only for node ports, masquerade all LB's IP at PREROUTING instead, because the implementation of LoadBalancer Service may not based on node ports, LB can bind pod ip directly in some implemetation. I think I can initiate another PR for implementing this idea.

Pod 偶尔存活检查失败

现象：Pod 偶尔会存活检查失败，导致 Pod 重启，业务偶尔连接异常。

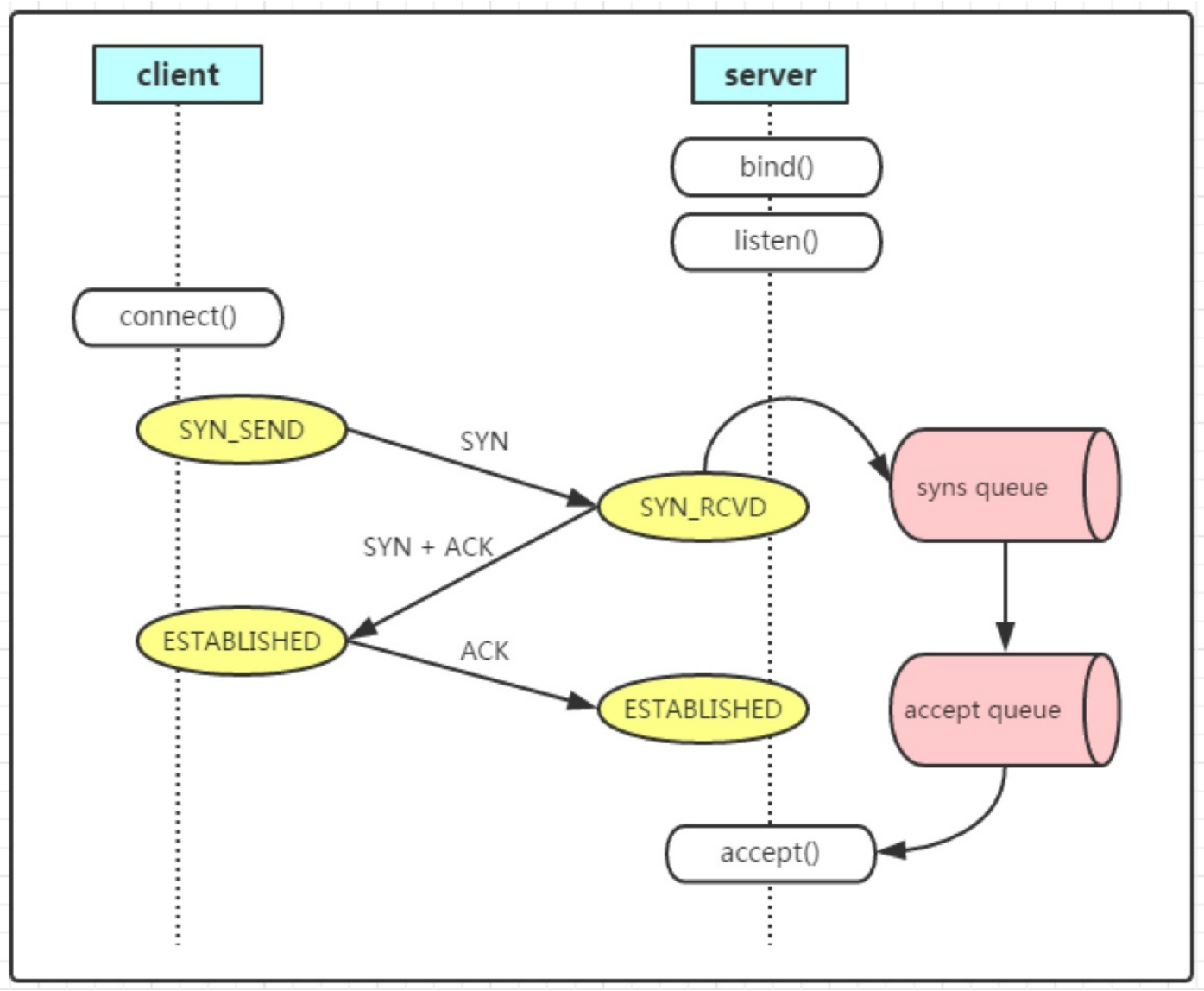
之前从未遇到这种情况，在自己测试环境尝试复现也没有成功，只有在用户这个环境才可以复现。这个用户环境流量较大，感觉跟连接数或并发量有关。

用户反馈说在友商的环境里没这个问题。

对比友商的内核参数发现有些区别，尝试将节点内核参数改成跟友商的一样，发现问题没有复现了。

再对比分析下内核参数差异，最后发现是 backlog 太小导致的，节点的 `net.ipv4.tcp_max_syn_backlog` 默认是 1024，如果短时间内并发新建 TCP 连接太多，SYN 队列就可能溢出，导致部分新连接无法建立。

解释一下：



TCP 连接建立会经过三次握手，server 收到 SYN 后会将连接加入 SYN 队列，当收到最后一个 ACK 后连接建立，这时会将连接从 SYN 队列中移动到 ACCEPT 队列。在 SYN 队列中的连接都是没有建立完全的连接，处于半连接状态。如果 SYN 队列比较小，而短时间内并发新建的连接比较多，同时处于半连接状态的连接就多，SYN 队列就可能溢出， `tcp_max_syn_backlog` 可以控制 SYN 队列大小，用户节点的 backlog 大小默认是 1024，改成 8096 后就可以解决问题。

DNS 解析异常

现象：有个用户反馈域名解析有时有问题，看报错是解析超时。

第一反应当然是看 coredns 的 log:

```
1. [ERROR] 2 loginspub.xxxxmobile-inc.net.
   A: unreachable backend: read udp 172.16.0.230:43742->10.225.30.181:53: i/o
2. timeout
```

这是上游 DNS 解析异常了，因为解析外部域名 coredns 默认会请求上游 DNS 来查询，这里的上游 DNS 默认是 coredns pod 所在宿主机的 `resolv.conf` 里面的 nameserver (coredns pod 的 dnsPolicy 为 “Default”，也就是会将宿主机里的 `resolv.conf` 里的 nameserver 加到容器里的 `resolv.conf`，coredns 默认配置 `proxy . /etc/resolv.conf`，意思是非 service 域名会使用 coredns 容器中 `resolv.conf` 文件里的 nameserver 来解析)

确认了下，超时的上游 DNS 10.225.30.181 并不是期望的 nameserver，VPC 默认 DNS 应该是 180 开头的。看了 coredns 所在节点的 `resolv.conf`，发现确实多出了这个非期望的 nameserver，跟用户确认了下，这个 DNS 不是用户自己加上去的，添加节点时这个 nameserver 本身就在 `resolv.conf` 中。

根据内部同学反馈， 10.225.30.181 是广州一台年久失修将被撤裁的 DNS，物理网络，没有 VIP，撤掉就没有了，所以如果 coredns 用到了这台 DNS 解析时就可能 timeout。后面我们自己测试，某些 VPC 的集群确实会有这个 nameserver，奇了怪了，哪里冒出来的？

又试了下直接创建 CVM，不加进 TKE 节点发现没有这个 nameserver，只要一加进 TKE 节点就有了 !!!

看起来是 TKE 的问题，将 CVM 添加到 TKE 集群会自动重装系统，初始化并加进集群成为 K8S 的 node，确认了初始化过程并不会写 `resolv.conf`，会不会是 TKE 的 OS 镜像问题？尝试搜一下除了 `/etc/resolv.conf` 之外哪里还有这个 nameserver 的 IP，最后发现 `/etc/resolvconf/resolv.conf.d/base` 这里面有。

看下 `/etc/resolvconf/resolv.conf.d/base` 的作用：Ubuntu 的 `/etc/resolv.conf` 是动态生成的，每次重启都会将 `/etc/resolvconf/resolv.conf.d/base` 里面的内容加到 `/etc/resolv.conf` 里。

经确认：这个文件确实是 TKE 的 Ubuntu OS 镜像里自带的，可能发布 OS 镜像时不小心加进去的。

那为什么有些 VPC 的集群的节点 `/etc/resolv.conf` 里面没那个 IP 呢？它们的 OS 镜像里也都有那个文件那个 IP 呀。

请教其它部门同学发现：

- 非 dhcp 子机，cvm 的 cloud-init 会覆盖 `/etc/resolv.conf` 来设置 dns
- dhcp 子机，cloud-init 不会设置，而是通过 dhcp 动态下发
- 2018 年 4 月 之后创建的 VPC 就都是 dhcp 类型了的，比较新的 VPC 都是 dhcp 类型的

真相大白：`/etc/resolv.conf` 一开始内容都包含 `/etc/resolvconf/resolv.conf.d/base` 的内容，也就是都有那个不期望的 nameserver，但老的 VPC 由于不是 dhcp 类型，所以 cloud-init 会覆盖 `/etc/resolv.conf`，抹掉了不被期望的 nameserver，而新创建的 VPC 都是 dhcp 类型，cloud-init 不会覆盖 `/etc/resolv.conf`，导致不被期望的 nameserver 残留在了 `/etc/resolv.conf`，而 coredns pod 的 dnsPolicy 为 “Default”，也就是会将宿主机的 `/etc/resolv.conf` 中的 nameserver 加到容器里，coredns 解析集群外的域名默认使用这些 nameserver 来解析，当用到那个将被撤裁的 nameserver 就可能 timeout。

临时解决：删掉 `/etc/resolvconf/resolv.conf.d/base` 重启

长期解决：我们重新制作 TKE Ubuntu OS 镜像然后发布更新

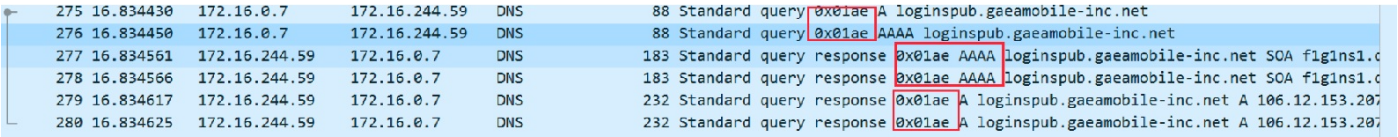
这下应该没问题了吧，But，用户反馈还是会偶尔解析有问题，但现象不一样了，这次并不是 dns timeout。

用脚本跑测试仔细分析现象：

- 请求 `loginspub.xxxxmobile-inc.net` 时，偶尔提示域名无法解析
- 请求 `accounts.google.com` 时，偶尔提示连接失败

进入 dns 解析偶尔异常的容器的 netns 抓包：

- dns 请求会并发请求 A 和 AAAA 记录
- 测试脚本发请求打印序号，抓包然后 wireshark 分析对比异常时请求序号偏移量，找到异常时的 dns 请求报文，发现异常时 A 和 AAAA 记录的请求 id 冲突，并且 AAAA 响应先返回



正常情况下id不会冲突，这里冲突了也就能解释这个 dns 解析异常的现象了：

- `loginspub.xxxxmobile-inc.net` 没有 AAAA (ipv6) 记录，它的响应先返回告知 client 不存在此记录，由于请求 id 跟 A 记录请求冲突，后面 A 记录响应返回了 client 发现 id 重复就忽略了，然后认为这个域名无法解析
- `accounts.google.com` 有 AAAA 记录，响应先返回了，client 就拿这个记录去尝试请求，但当前容器环境不支持 ipv6，所以会连接失败

那为什么 dns 请求 id 会冲突?

继续观察发现: 其它节点上的 pod 不会复现这个问题，有问题这个节点上也不是所有 pod 都有这个问题，只有基于 alpine 镜像的容器才有这个问题，在此节点新起一个测试的 `alpine:latest` 的容器也一样有这个问题。

为什么 alpine 镜像的容器在这个节点上有问题在其它节点上没问题？ 为什么其他镜像的容器都没问题？它们跟 alpine 的区别是什么？

发现一点区别: alpine 使用的底层 c 库是 musl libc，其它镜像基本都是 glibc

翻 musl libc 源码, 构造 dns 请求时，请求 id 的生成没加锁，而且跟当前时间戳有关 (`network/res_mkquery.c`):

```
1.  /* Make a reasonably unpredictable id */
2.  clock_gettime(CLOCK_REALTIME, &ts);
3.  id = ts.tv_nsec + ts.tv_nsec/65536UL & 0xffff;
```

看注释，作者应该认为这样id基本不会冲突，事实证明，绝大多数情况确实不会冲突，我在网上搜了很久没有搜到任何关于 musl libc 的 dns 请求 id 冲突的情况。这个看起来取决于硬件，可能在某种类型硬件的机器上运行，短时间内生成的 id 就可能冲突。我尝试跟用户在相同地域的集群，添加相同配置相同机型的节点，也复现了这个问题，但后来删除再添加时又不能复现了，看起来后面新建的 cvm 又跑在了另一种硬件的母机上了。

OK，能解释通了，再底层的细节就不清楚了，我们来看下解决方案:

- 换基础镜像 (不用alpine)
- 完全静态编译业务程序(不依赖底层c库)，比如go语言程序编译时可以关闭 cgo (CGO_ENABLED=0)，并告诉链接器要静态链接 (`go build` 后面加 `-ldflags '-d'`)，但这需要语言和编译工具支持才可以

最终建议用户基础镜像换成另一个比较小的镜像: `debian:stretch-slim` 。

Pod 访问另一个集群的 apiserver 有延时

现象：集群 a 的 Pod 内通过 kubectl 访问集群 b 的内网地址，偶尔出现延时的情况，但直接在宿主机上用同样的方法却没有这个问题。

提炼环境和现象精髓：

1. 在 pod 内将另一个集群 apiserver 的 ip 写到了 hosts，因为 TKE apiserver 开启内网集群外内网访问创建的内网 LB 暂时没有支持自动绑内网 DNS 域名解析，所以集群外的内网访问 apiserver 需要加 hosts
2. pod 内执行 kubectl 访问另一个集群偶尔延迟 5s，有时甚至10s

观察到 5s 延时，感觉跟之前 conntrack 的丢包导致 DNS 解析 5S 延时 有关，但是加了 hosts 呀，怎么还去解析域名？

进入 pod netns 抓包：执行 kubectl 时确实有 dns 解析，并且发生延时的时候 dns 请求没有响应然后做了重试。

看起来延时应该就是之前已知 conntrack 丢包导致 dns 5s 超时重试导致的。但是为什么会去解析域名？明明配了 hosts 啊，正常情况应该是优先查找 hosts，没找到才去请求 dns 呀，有什么配置可以控制查找顺序？

搜了一下发现：`/etc/nsswitch.conf` 可以控制，但看有问题的 pod 里没有这个文件。然后观察到有问题的 pod 用的 alpine 镜像，试试其它镜像后发现只有基于 alpine 的镜像才会有这个问题。

再一搜发现：musl libc 并不会使用 `/etc/nsswitch.conf` ，也就是说 alpine 镜像并没有实现用这个文件控制域名查找优先顺序，瞥了一眼 musl libc 的 `gethostbyname` 和 `getaddrinfo` 的实现，看起来也没有读这个文件来控制查找顺序，写死了先查 hosts，没找到再查 dns。

这么说，那还是该先查 hosts 再查 dns 呀，为什么这里抓包看到是先查的 dns？(如果是先查 hosts 就能命中查询，不会再发起dns请求)

访问 apiserver 的 client 是 kubectl，用 go 写的，会不会是 go 程序解析域名时压根没调底层 c 库的 `gethostbyname` 或 `getaddrinfo` ？

搜一下发现果然是这样：go runtime 用 go 实现了 glibc 的 `getaddrinfo` 的行为来解析域名，减少了 c 库调用 (应该是考虑到减少 cgo 调用带来的的性能损耗)

issue: net: replicate DNS resolution behaviour of getaddrinfo(glibc) in the go dns resolver

翻源码验证下：

Unix 系的 OS 下，除了 openbsd， go runtime 会读取 `/etc/nsswitch.conf` (`net/conf.go`):

```
1.  if runtime.GOOS != "openbsd" {
2.      confVal.nss = parseNSSConfFile("/etc/nsswitch.conf")
3.  }
```

`hostLookupOrder` 函数决定域名解析顺序的策略，Linux 下，如果没有 `nsswitch.conf` 文件就 dns 比 hosts 文件优先 (`net/conf.go`):

```
1.  // hostLookupOrder determines which strategy to use to resolve hostname.
2.  // The provided Resolver is optional. nil means to not consider its options.
    func (c *conf) hostLookupOrder(r *Resolver, hostname string) (ret
3.  hostLookupOrder) {
4.      ......
5.      // If /etc/nsswitch.conf doesn't exist or doesn't specify any
6.      // sources for "hosts", assume Go's DNS will work fine.
7.      if os.IsNotExist(nss.err) || (nss.err == nil && len(srcs) == 0) {
8.          ......
9.          if c.goos == "linux" {
10.             // glibc says the default is "dns [!UNAVAIL=return] files"
                // https://www.gnu.org/software/libc/manual/html_node/Notes-on-NSS-
11. Configuration-File.html.
12.             return hostLookupDNSFiles
13.         }
14.         return hostLookupFilesDNS
15.     }
```

可以看到 `hostLookupDNSFiles` 的意思是 dns first (`net/dnsclient_unix.go`):

```
1.  // hostLookupOrder specifies the order of LookupHost lookup strategies.
2.  // It is basically a simplified representation of nsswitch.conf.
3.  // "files" means /etc/hosts.
4.  type hostLookupOrder int
5.
6.  const (
7.      // hostLookupCgo means defer to cgo.
8.      hostLookupCgo      hostLookupOrder = iota
9.      hostLookupFilesDNS                 // files first
10.     hostLookupDNSFiles                 // dns first
```

```
    hostLookupFiles                    // only files
    hostLookupDNS                      // only DNS
)

var lookupOrderName = map[hostLookupOrder]string{
    hostLookupCgo:      "cgo",
    hostLookupFilesDNS: "files,dns",
    hostLookupDNSFiles: "dns,files",
    hostLookupFiles:    "files",
    hostLookupDNS:      "dns",
}
```

所以虽然 alpine 用的 musl libc 不是 glibc，但 go 程序解析域名还是一样走的 glibc 的逻辑，而 alpine 没有 `/etc/nsswitch.conf` 文件，也就解释了为什么 kubectl 访问 apiserver 先做 dns 解析，没解析到再查的 hosts，导致每次访问都去请求 dns，恰好又碰到 conntrack 那个丢包问题导致 dns 5s 延时，在用户这里表现就是 pod 内用 kubectl 访问 apiserver 偶尔出现 5s 延时，有时出现 10s 是因为重试的那次 dns 请求刚好也遇到 conntrack 丢包导致延时又叠加了 5s 。

解决方案：

1. 换基础镜像，不用 alpine
2. 挂载 `nsswitch.conf` 文件（可以用 hostPath）

LB 压测 NodePort CPS 低

现象：LoadBalancer 类型的 Service，直接压测 NodePort CPS 比较高，但如果压测 LB CPS 就很低。

环境说明：用户使用的黑石TKE，不是公有云TKE，黑石的机器是物理机，LB的实现也跟公有云不一样，但 LoadBalancer 类型的 Service 的实现同样也是 LB 绑定各节点的 NodePort，报文发到 LB 后转到节点的 NodePort， 然后再路由到对应 pod，而测试在公有云 TKE 环境下没有这个问题。

- client 抓包：大量SYN重传。
- server 抓包：抓 NodePort 的包，发现当 client SYN 重传时 server 能收到 SYN 包但没有响应。

又是 SYN 收到但没响应，难道又是开启 `tcp_tw_recycle` 导致的？检查节点的内核参数发现并没有开启，除了这个原因，还会有什么情况能导致被丢弃？

`conntrack -S` 看到 `insert_failed` 数量在不断增加，也就是 conntrack 在插入很多新连接的时候失败了，为什么会插入失败？什么情况下会插入失败？

挖内核源码：netfilter conntrack 模块为每个连接创建 conntrack 表项时，表项的创建和最终插入之间还有一段逻辑，没有加锁，是一种乐观锁的过程。conntrack 表项并发刚创建时五元组不冲突的话可以创建成功，但中间经过 NAT 转换之后五元组就可能变成相同，第一个可以插入成功，后面的就会插入失败，因为已经有相同的表项存在。比如一个 SYN 已经做了 NAT 但是还没到最终插入的时候，另一个 SYN 也在做 NAT，因为之前那个 SYN 还没插入，这个 SYN 做 NAT 的时候就认为这个五元组没有被占用，那么它 NAT 之后的五元组就可能跟那个还没插入的包相同。

在我们这个问题里实际就是 netfilter 做 SNAT 时源端口选举冲突了，黑石 LB 会做 SNAT，SNAT 时使用了 16 个不同 IP 做源，但是短时间内源 Port 却是集中一致的，并发两个 SYN a 和SYN b，被 LB SNAT 后源 IP 不同但源 Port 很可能相同，这里就假设两个报文被 LB SNAT 之后它们源 IP 不同源 Port 相同，报文同时到了节点的 NodePort 会再次做 SNAT 再转发到对应的 Pod，当报文到了 NodePort 时，这时它们五元组不冲突，netfilter 为它们分别创建了 conntrack 表项，SYN a 被节点 SNAT 时默认行为是 从 port_range 范围的当前源 Port 作为起始位置开始循环遍历，选举出没有被占用的作为源 Port，因为这两个 SYN 源 Port 相同，所以它们源 Port 选举的起始位置相同，当 SYN a 选出源 Port 但还没将 conntrack 表项插入时，netfilter 认为这个 Port 没被占用就很可能给 SYN b 也选了相同的源 Port，这时他们五元组就相同了，当 SYN a 的 conntrack 表项插入后再插入 SYN b 的 conntrack 表项时，发现已经有相同的记录就将 SYN b 的 conntrack 表项丢弃了。

解决方法探索：不使用源端口选举，在 iptables 的 MASQUERADE 规则如果加 `--random-fully` 这个 flag 可以让端口选举完全随机，基本上能避免绝大多数的冲突，但也无法完全杜绝。

最终决定开发 LB 直接绑 Pod IP，不基于 NodePort，从而避免 netfilter 的 SNAT 源端口冲突问题。

kubectl edit 或者 apply 报 SchemaError

问题现象

kubectl edit 或 apply 资源报如下错误:

```
    error: SchemaError(io.k8s.apimachinery.pkg.apis.meta.v1.APIGroup): invalid
1.  object doesn't have additional properties
```

集群版本: v1.10

排查过程

1. 使用 `kubectl apply -f tmp.yaml --dry-run -v8` 发现请求 `/openapi/v2` 这个 api 之后，kubectl在 validate 过程报错。
2. 换成 kubectl 1.12 之后没有再报错。
3. `kubectl get --raw '/openapi/v2'` 发现返回的 json 内容与正常集群有差异，刚开始返回的 json title 为 `Kubernetes metrics-server`，正常的是 Kubernetes。
4. 怀疑是 `metrics-server` 的问题，发现集群内确实安装了 k8s 官方的 `metrics-server`，询问得知之前是 0.3.1，后面升级为了 0.3.5。
5. 将 metrics-server 回滚之后恢复正常。

原因分析

初步怀疑，新版本的 metrics-server 使用了新的 openapi-generator，生成的 openapi 格式和之前 k8s 版本生成的有差异。导致旧版本的 kubectl 在解析 openapi 的 schema 时发生异常，查看代码发现1.10 和 1.12 版本在解析 openapi 的 schema 时，实现确实有差异。

排错技巧

分析 ExitCode 定位 Pod 异常退出原因

使用 `kubectl describe pod <pod name>` 查看异常 pod 的状态：

```
Containers:
  kubedns:
    Container ID:
docker://5fb8adf9ee62afc6d3f6f3d9590041818750b392dff015d7091eaaf99cf1c945
    Image:          ccr.ccs.tencentyun.com/library/kubedns-amd64:1.14.4
    Image ID:       docker-pullable://ccr.ccs.tencentyun.com/library/kubedns-
amd64@sha256:40790881bbe9ef4ae4ff7fe8b892498eecb7fe6dcc22661402f271e03f7de344
    Ports:          10053/UDP, 10053/TCP, 10055/TCP
    Host Ports:     0/UDP, 0/TCP, 0/TCP
    Args:
      --domain=cluster.local.
      --dns-port=10053
      --config-dir=/kube-dns-config
      --v=2
    State:          Running
      Started:      Tue, 27 Aug 2019 10:58:49 +0800
    Last State:     Terminated
      Reason:       Error
      Exit Code:    255
      Started:      Tue, 27 Aug 2019 10:40:42 +0800
      Finished:     Tue, 27 Aug 2019 10:58:27 +0800
    Ready:          True
    Restart Count:  1
```

在容器列表里看 `Last State` 字段，其中 `ExitCode` 即程序上次退出时的状态码，如果不为 0，表示异常退出，我们可以分析下原因。

退出状态码的区间

- 必须在 0-255 之间
- 0 表示正常退出
- 外界中断将程序退出的时候状态码区间在 129-255，(操作系统给程序发送中断信号，比如 `kill -9` 是 `SIGKILL`，`ctrl+c` 是 `SIGINT`)
- 一般程序自身原因导致的异常退出状态区间在 1-128 (这只是一般约定，程序如果一定要用 129-255的状态码也是可以的)

假如写代码指定的退出状态码时不在 0-255 之间，例如： `exit(-1)` ，这时会自动做一个转换，最终呈现的状态码还是会在 0-255 之间。我们把状态码记为 `code`

- 当指定的退出时状态码为负数，那么转换公式如下：

```
256 - (|code| % 256)
```

- 当指定的退出时状态码为正数，那么转换公式如下：

```
code % 256
```

常见异常状态码

- 137（被 `SIGKILL` 中断信号杀死）
  - 此状态码一般是因为 pod 中容器内存达到了它的资源限制(`resources.limits`)，一般是内存溢出(OOM)，CPU达到限制只需要不分时间片给程序就可以。因为限制资源是通过 linux 的 cgroup 实现的，所以 cgroup 会将此容器强制杀掉，类似于 `kill -9` ，此时在 `describe pod` 中可以看到 Reason 是 `OOMKilled`
  - 还可能是宿主机本身资源不够用了(OOM)，内核会选取一些进程杀掉来释放内存
  - 不管是 cgroup 限制杀掉进程还是因为节点机器本身资源不够导致进程死掉，都可以从系统日志中找到记录：

    > ubuntu 的系统日志在 `/var/log/syslog` ，centos 的系统日志在 `/var/log/messages` ，都可以用 `journalctl -k` 来查看系统日志

  - 也可能是 livenessProbe（存活检查）失败，kubelet 杀死的 pod
  - 还可能是被恶意木马进程杀死
- 1 和 255
  - 这种可能是一般错误，具体错误原因只能看容器日志，因为很多程序员写异常退出时习惯用 `exit(1)` 或 `exit(-1)` ，-1 会根据转换规则转成 255

状态码参考

这里罗列了一些状态码的含义：Appendix E. Exit Codes With Special Meanings

Linux 标准中断信号

Linux 程序被外界中断时会发送中断信号，程序退出时的状态码就是中断信号值加上 128 得到的，比如 `SIGKILL` 的中断信号值为 9，那么程序退出状态码就为 9+128=137。以下是标准信号值参考：

```
Signal     Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP        1       Term    Hangup detected on controlling terminal
                                     or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction
SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating-point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                                     readers; see pipe(7)
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated
SIGCONT   19,18,25    Cont    Continue if stopped
SIGSTOP   17,19,23    Stop    Stop process
SIGTSTP   18,20,24    Stop    Stop typed at terminal
SIGTTIN   21,21,26    Stop    Terminal input for background process
SIGTTOU   22,22,27    Stop    Terminal output for background process
```

C/C++ 退出状态码

`/usr/include/sysexits.h` 试图将退出状态码标准化(仅限 C/C++)：

```c
#define EX_OK           0       /* successful termination */

#define EX__BASE        64      /* base value for error messages */

#define EX_USAGE        64      /* command line usage error */
#define EX_DATAERR      65      /* data format error */
#define EX_NOINPUT      66      /* cannot open input */
#define EX_NOUSER       67      /* addressee unknown */
#define EX_NOHOST       68      /* host name unknown */
```

```
#define EX_UNAVAILABLE  69      /* service unavailable */
#define EX_SOFTWARE     70      /* internal software error */
#define EX_OSERR        71      /* system error (e.g., can't fork) */
#define EX_OSFILE       72      /* critical OS file missing */
#define EX_CANTCREAT    73      /* can't create (user) output file */
#define EX_IOERR        74      /* input/output error */
#define EX_TEMPFAIL     75      /* temp failure; user is invited to retry */
#define EX_PROTOCOL     76      /* remote error in protocol */
#define EX_NOPERM       77      /* permission denied */
#define EX_CONFIG       78      /* configuration error */

#define EX__MAX 78      /* maximum listed value */
```

容器内抓包定位网络问题

在使用 kubernetes 跑应用的时候，可能会遇到一些网络问题，比较常见的是服务端无响应(超时)或回包内容不正常，如果没找出各种配置上有问题，这时我们需要确认数据包到底有没有最终被路由到容器里，或者报文到达容器的内容和出容器的内容符不符合预期，通过分析报文可以进一步缩小问题范围。那么如何在容器内抓包呢？本文提供实用的脚本一键进入容器网络命名空间(netns)，使用宿主机上的tcpdump进行抓包。

使用脚本一键进入 pod netns 抓包

- 发现某个服务不通，最好将其副本数调为1，并找到这个副本 pod 所在节点和 pod 名称

```
kubectl get pod -o wide
```

- 登录 pod 所在节点，将如下脚本粘贴到 shell (注册函数到当前登录的 shell，我们后面用)

```
function e() {
    set -eu
    ns=${2-"default"}
    pod=`kubectl -n $ns describe pod $1 | grep -A10 "^Containers:" | grep -Eo 'docker://.*$' | head -n 1 | sed 's/docker:\/\/\(.*\)$/\1/'`
    pid=`docker inspect -f {{.State.Pid}} $pod`
    echo "entering pod netns for $ns/$1"
    cmd="nsenter -n --target $pid"
    echo $cmd
    $cmd
}
```

- 一键进入 pod 所在的 netns，格式：`e POD_NAME NAMESPACE`，示例：

```
e istio-galley-58c7c7c646-m6568 istio-system
e proxy-5546768954-9rxg6 # 省略 NAMESPACE 默认为 default
```

- 这时已经进入 pod 的 netns，可以执行宿主机上的 `ip a` 或 `ifconfig` 来查看容器的网卡，执行 `netstat -tunlp` 查看当前容器监听了哪些端口，再通过 `tcpdump` 抓包：

```
tcpdump -i eth0 -w test.pcap port 80
```

- `ctrl-c` 停止抓包，再用 `scp` 或 `sz` 将抓下来的包下载到本地使用 `wireshark` 分析，提供一些常用的 `wireshark` 过滤语法：

```
# 使用 telnet 连上并发送一些测试文本，比如 "lbtest"，
# 用下面语句可以看发送的测试报文有没有到容器
tcp contains "lbtest"
# 如果容器提供的是http服务，可以使用 curl 发送一些测试路径的请求，
# 通过下面语句过滤 uri 看报文有没有都容器
http.request.uri=="/mytest"
```

脚本原理

我们解释下步骤二中用到的脚本的原理

- 查看指定 pod 运行的容器 ID

```
kubectl describe pod <pod> -n mservice
```

- 获得容器进程的 pid

```
docker inspect -f {{.State.Pid}} <container>
```

- 进入该容器的 network namespace

```
nsenter -n --target <PID>
```

依赖宿主机的命名：`kubectl`，`docker`，`nsenter`，`grep`，`head`，`sed`

使用 Systemtap 定位疑难杂症

安装

Ubuntu

安装 systemtap:

```
apt install -y systemtap
```

运行 `stap-prep` 检查还有什么需要安装:

```
$ stap-prep
Please install linux-headers-4.4.0-104-generic
You need package linux-image-4.4.0-104-generic-dbgsym but it does not seem to
be available
 Ubuntu -dbgsym packages are typically in a separate repository
 Follow https://wiki.ubuntu.com/DebuggingProgramCrash to add this repository

apt install -y linux-headers-4.4.0-104-generic
```

提示需要 dbgsym 包但当前已有软件源中并不包含，需要使用第三方软件源安装，下面是 dbgsym 安装方法(参考官方wiki: https://wiki.ubuntu.com/Kernel/Systemtap):

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys C8CAB6595FDFF622

codename=$(lsb_release -c | awk  '{print $2}')
sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ ${codename}      main restricted universe
multiverse
deb http://ddebs.ubuntu.com/ ${codename}-security main restricted universe
multiverse
deb http://ddebs.ubuntu.com/ ${codename}-updates  main restricted universe
multiverse
deb http://ddebs.ubuntu.com/ ${codename}-proposed main restricted universe
multiverse
EOF

sudo apt-get update
```

配置好源后再运行下 `stap-prep` ：

```
$ stap-prep
Please install linux-headers-4.4.0-104-generic
Please install linux-image-4.4.0-104-generic-dbgsym
```

提示需要装这两个包，我们安装一下：

```
apt install -y linux-image-4.4.0-104-generic-dbgsym
apt install -y linux-headers-4.4.0-104-generic
```

CentOS

安装 systemtap:

```
yum install -y systemtap
```

默认没装 `debuginfo` ，我们需要装一下，添加软件源 `/etc/yum.repos.d/CentOS-Debug.repo` ：

```
[debuginfo]
name=CentOS-$releasever - DebugInfo
baseurl=http://debuginfo.centos.org/$releasever/$basearch/
gpgcheck=0
enabled=1
protect=1
priority=1
```

执行 `stap-prep` （会安装 `kernel-debuginfo` ）

最后检查确保 `kernel-debuginfo` 和 `kernel-devel` 均已安装并且版本跟当前内核版本相同，如果有多个版本，就删除跟当前内核版本不同的包(通过 `uname -r` 查看当前内核版本)。

重点检查是否有多个版本的 `kernel-devel` ：

```
$ rpm -qa | grep kernel-devel
kernel-devel-3.10.0-327.el7.x86_64
kernel-devel-3.10.0-514.26.2.el7.x86_64
kernel-devel-3.10.0-862.9.1.el7.x86_64
```

如果存在多个，保证只留跟当前内核版本相同的那个，假设当前内核版本是 `3.10.0-`

`862.9.1.el7.x86_64` ，那么使用 rpm 删除多余的版本：

```
    rpm -e kernel-devel-3.10.0-327.el7.x86_64 kernel-devel-3.10.0-
1.  514.26.2.el7.x86_64
```

使用 systemtap 揪出杀死容器的真凶

Pod 莫名其妙被杀死？可以使用 systemtap 来监视进程的信号发送，原理是 systemtap 将脚本翻译成 C 代码然后调用 gcc 编译成 linux 内核模块，再通过 `modprobe` 加载到内核，根据脚本内容在内核做各种 hook，在这里我们就 hook 一下信号的发送，找出是谁 kill 掉了容器进程。

首先，找到被杀死的 pod 又自动重启的容器的当前 pid，describe 一下 pod：

```
1.      ......
        Container ID:
2.  docker://5fb8adf9ee62afc6d3f6f3d9590041818750b392dff015d7091eaaf99cf1c945
3.      ......
4.      Last State:     Terminated
5.        Reason:       Error
6.        Exit Code:    137
7.        Started:      Thu, 05 Sep 2019 19:22:30 +0800
8.        Finished:     Thu, 05 Sep 2019 19:33:44 +0800
```

拿到容器 id 反查容器的主进程 pid：

```
    $ docker inspect -f "{{.State.Pid}}"
1.  5fb8adf9ee62afc6d3f6f3d9590041818750b392dff015d7091eaaf99cf1c945
2.  7942
```

通过 `Exit Code` 可以看出容器上次退出的状态码，如果进程是被外界中断信号杀死的，退出状态码将在 129-255 之间，137 表示进程是被 SIGKILL 信号杀死的，但我们从这里并不能看出是被谁杀死的。

如果问题可以复现，我们可以使用下面的 systemtap 脚本来监视容器是被谁杀死的(保存为 `sg.stp`)：

```
1.  global target_pid = 7942
2.  probe signal.send{
3.    if (sig_pid == target_pid) {
        printf("%s(%d) send %s to %s(%d)\n", execname(), pid(), sig_name, pid_name,
4.  sig_pid);
```

```
5.      printf("parent of sender: %s(%d)\n", pexecname(), ppid())
6.      printf("task_ancestry:%s\n", task_ancestry(pid2task(pid()), 1));
7.    }
8.  }
```

- 变量 `pid` 的值替换为查到的容器主进程 pid

运行脚本：

```
1.  stap sg.stp
```

当容器进程被杀死时，脚本捕捉到事件，执行输出：

```
1.  pkill(23549) send SIGKILL to server(7942)
2.  parent of sender: bash(23495)
3.  task_ancestry:swapper/0(0m0.000000000s)=>systemd(0m0.080000000s)=>vGhyM0(19491m2
```

通过观察 `task_ancestry` 可以看到杀死进程的所有父进程，在这里可以看到有个叫 `vGhyM0` 的奇怪进程名，通常是中了木马，需要安全专家介入继续排查。

Go 语言编译原理与优化

编译阶段 (Compilation)

debug 参数

`-m` 打印编译器更多想法的细节

```
1.  -gcflags '-m'
```

`-S` 打印汇编

```
1.  -gcflags '-S'
```

优化和内联

默认开启了优化和内联，但是debug的时候开启可能会出现一些奇怪的问题，通过下面的参数可以禁止任何优化

```
1.  -gcflags '-N -l'
```

内联级别:

- `-gcflags='-l -l'` 内联级别2，更积极，可能更快，可能会制作更大的二进制文件。
- `-gcflags='-l -l -l'` 内联级别3，再次更加激进，二进制文件肯定更大，也许更快，但也许会有 bug。
- `-gcflags=-l=4` (4个-l)在 Go 1.11 中将支持实验性的中间栈内联优化。

逃逸分析

- 如果一个局部变量值超越了函数调用的生命周期，编译器自动将它逃逸到堆
- 如果一个通过new或make来分配的对象，在函数内即使将指针传递给了其它函数，其它函数会被内联到当前函数，相当于指针不会逃逸出本函数，最终不返回指针的话，该指针对应的值也都会分配在栈上，而不是在堆

链接阶段 (Linking)

- Go 支持 internal 和 external 两种链接方式: internal 使用 go 自身实现的

linker，external 需要启动外部的 linker

- linker 的主要工作是将 `.o` (object file) 链接成最终可执行的二进制
- 对应命令：`go tool link`，对应源码：`$GOROOT/src/cmd/link`
- 通过 `-ldflags` 给链接器传参，参数详见：`go tool link --help`

关于 CGO

- 启用cgo可以调用外部依赖的c库
- go的编译器会判断环境变量 `CGO_ENABLED` 来决定是否启用cgo，默认 `CGO_ENABLED=1` 即启用cgo
- 源码文件头部的 `build tag` 可以根据cgo是否启用决定源码是否被编译(`// +build cgo` 表示希望cgo启用时被编译，相反的是 `// +build !cgo`)
- 标准库中有部分实现有两份源码，比如：`$GOROOT/src/os/user/lookup_unix.go` 和 `$GOROOT/src/os/user/cgo_lookup_unix.go` ，它们有相同的函数，但实现不一样，前者是纯go实现，后者是使用cgo调用外部依赖来实现，标准库中使用cgo比较常见的是 `net` 包。

internal linking

- link 默认使用 internal 方式
- 直接使用 go 本身的实现的 linker 来链接代码，
- 功能比较简单，仅仅是将 `.o` 和预编译的 `.a` 写到最终二进制文件中(`.a` 文件在 `$GOROOT/pkg` 和 `$GOPATH/pkg` 中，其实就是 `.o` 文件打包的压缩包，通过 `tar -zxvf` 可以解压出来查看)

external linking

- 会启动外部 linker (gcc/clang)，通过 `-ldflags '-linkmode "external"'` 启用 external linking
- 通过 `-extldflags` 给外部 linker 传参，比如：`-ldflags '-linkmode "external" -extldflags "-static"'`

static link

go编译出来就是一个二进制，自带runtime，不需要解释器，但并不意味着就不需要任何依赖，但也可以通过静态链接来做到完全不用任何依赖，全部”揉“到一个二进制文件中。实现静态链接的方法：

- 如果是 `external linking`，可以这样：`-ldflags '-linkmode external -extldflags -static'`
- 如果用默认的 `internal linking`，可以这样：`-ldflags '-d'`

ldflags 其它常用参数

- `-s -w` 是去除符号表和DWARF调试信息(可以减小二进制体积，但不利于调试，可在用于生产环境)，示例：`-ldflags '-s -w'`
- `-X` 可以给变量注入值，比如编译时用脚本动态注入当前版本和 `commit id` 到代码的变量中，通常程序的 `version` 子命令或参数输出当前版本信息时就用这种方式实现，示例：`-ldflags '-X myapp/pkg/version/version=v1.0.0'`

使用 Docker 编译

使用 Docker 编译可以不用依赖本机 go 环境，将编译环境标准化，特别在有外部动态链接库依赖的情况下很有用，可以直接 run 一个容器来编译，给它挂载源码目录和二进制输出目录，这样我们就可以拿到编译出来的二进制了，这里以编译cfssl为例：

```
ROOT_PKG=github.com/cloudflare/cfssl
CMD_PKG=$ROOT_PKG/cmd
LOCAL_SOURCE_PATH=/Users/roc/go/src/$ROOT_PKG
LOCAL_OUTPUT_PATH=$PWD
GOPATH=/go/src
ROOT_PATH=$GOPATH/$ROOT_PKG
CMD_PATH=$GOPATH/$CMD_PKG
docker run --rm \
  -v $LOCAL_SOURCE_PATH:$ROOT_PATH \
  -v $LOCAL_OUTPUT_PATH:/output \
  -w $ROOT_PATH \
  golang:1.13 \
  go build -v \
  -ldflags '-d' \
  -o /output/ \
  $CMD_PATH/...
```

编译镜像可以参考下面示例（使用docker多阶段构建，完全静态编译，没有外部依赖）：

```
FROM golang:1.12-stretch as builder
MAINTAINER rockerchen@tencent.com
ENV BUILD_DIR /go/src/cloud.tencent.com/qc_container_cluster/hpa-metrics-server
WORKDIR $BUILD_DIR

COPY ./ $BUILD_DIR
RUN CGO_ENABLED=0 go build -v -o /hpa-metrics-server \
    -ldflags '-d' \
    ./

```

```
FROM ubuntu:16.04
MAINTAINER rockerchen@tencent.com
RUN apt-get update -y
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y curl iproute2 inetutils-
tools telnet inetutils-ping
RUN apt-get install --no-install-recommends --no-install-suggests ca-
certificates -y
COPY --from=builder /hpa-metrics-server /hpa-metrics-server
RUN chmod a+x /hpa-metrics-server
```

参考资料

- Go 性能调优之 —— 编译优化: https://segmentfault.com/a/1190000016354799