基于 k8s 平台构建容器云平台

* 本文基于 k8s 1.5.1 官方文档及网上资料，结合实践经验，争取完整的汇总平台构建各个细节。

* k8s 介绍：

Kubernetes 是 Google 开源的容器集群管理系统，实现基于 Docker 构建容器，并能很方便管理多台 Docker 主机集群中的容器用于自动部署、扩展和操作的应用程序。

* 目标：

1、容器云平台为公司提供一个高性能、高可用、高扩展、易运维的弹性架构，同时规范公司工作流程，进行自动集成发布部署，提升研发运维整体效率

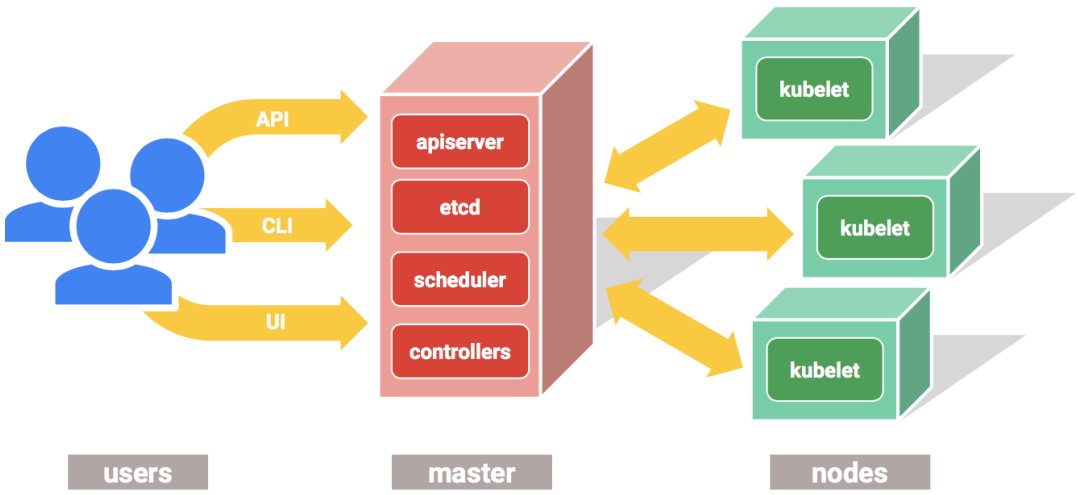
2、业务高可用弹性部署，微服务化

3、容器云第一阶段提供新项目使用，同时为老项目提供日志分析服务、监控汇总服务

4、系统运维工程师（2 年+工作经验）跟据文档，独立部署整个平台

一、 平台基础信息介绍

1）、平台架构



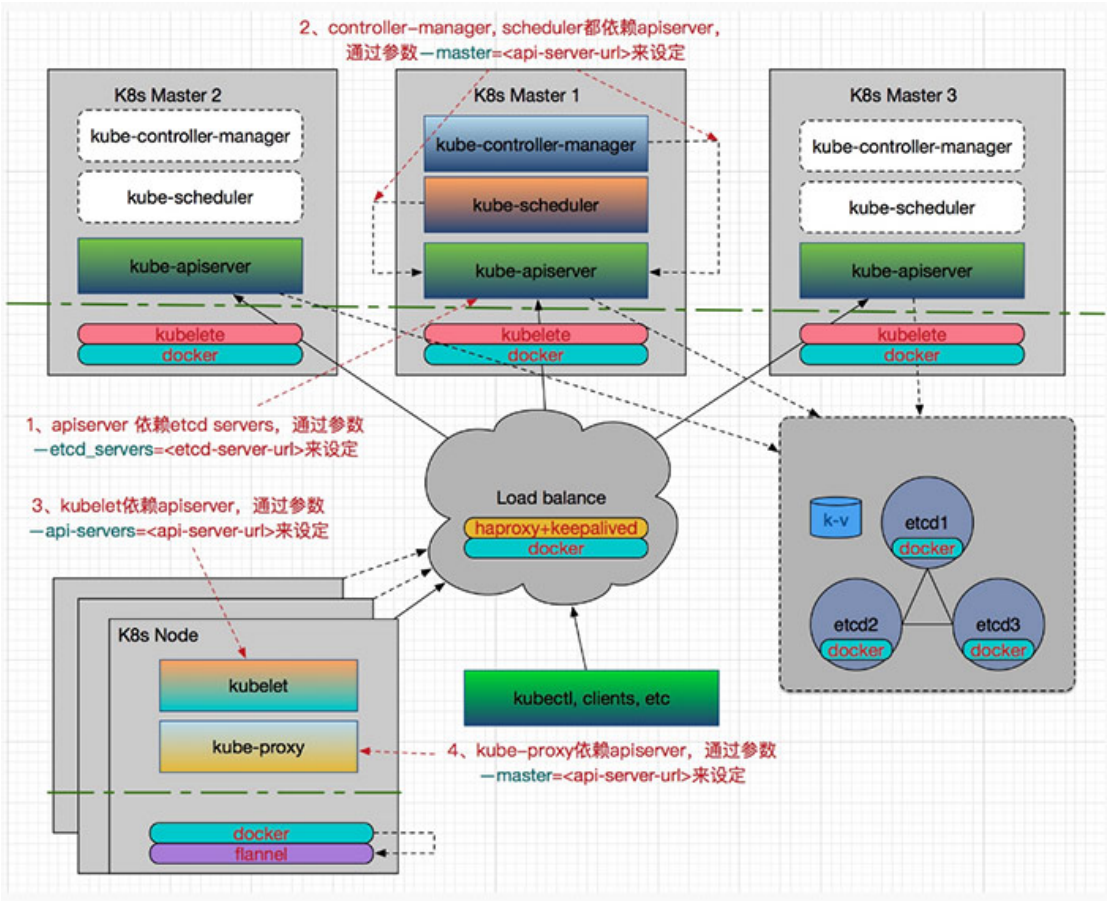
（图一）

* 用户可用通过 API、CLI、UI 三种方式与平台交互

* 主节点（master）包括 kube-apiserver、kube-scheduler、kube-controller、etcd 四个组件，才可以运行

* 节点（node）包括 kube-proxy、kubelet 两个组件

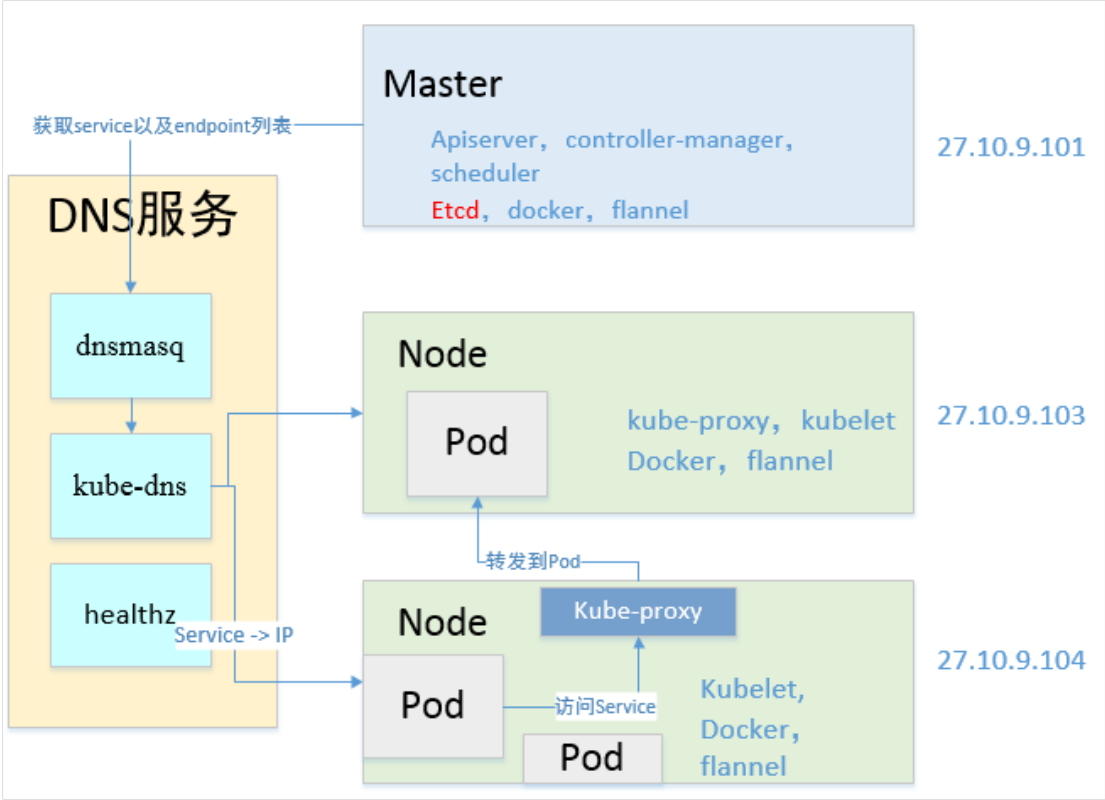
* 网络服务，各节点通过 openswitch vss、fannel 等构建 sdn 网络，达到各节点互通



（图二）

* kube-controller-manager 和 kube-scheduler 组件通过启动参数--leader-elect=true 开启选举模式，产生一个 master 提供服务，其余两个 standby。

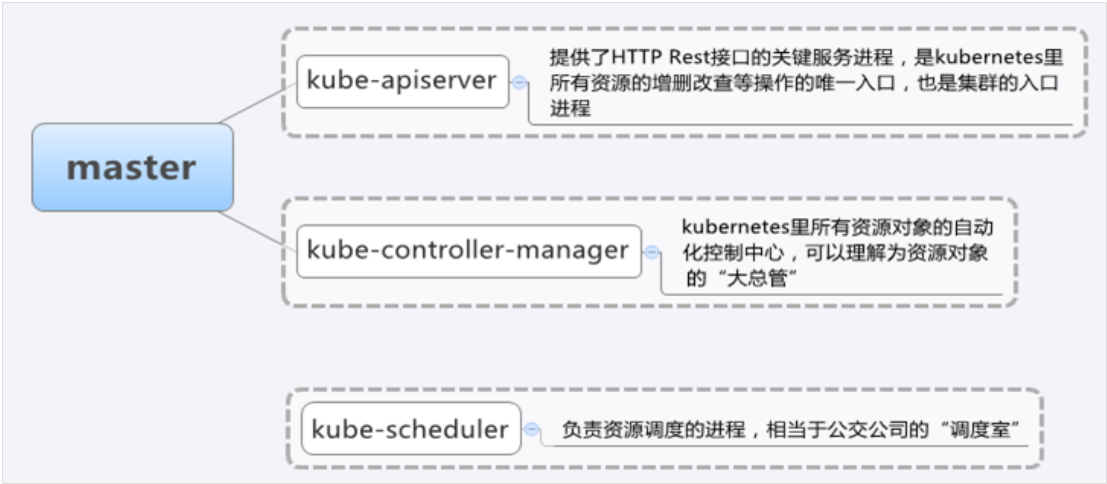
* etcd 和 kube-apiserver 都可以同时提供服务



（图三）

*k8s 系统内部服务发现

*该图主要希望描述清楚 dns 服务发现机制，不可以理解为机制



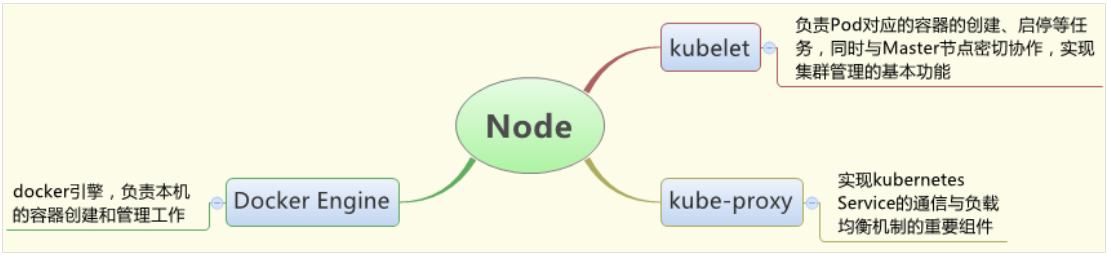
（图四）

*kube-controller-manager 集群的资源大脑，负责资源管控调度

*kube-scheduler 集群的执行者，负责将大脑指令分配下达

*kube-apiserver 集群的业务大脑，负责处理具体指令

*上图描述不到位，master 节点一般还有网路组件，核心插件



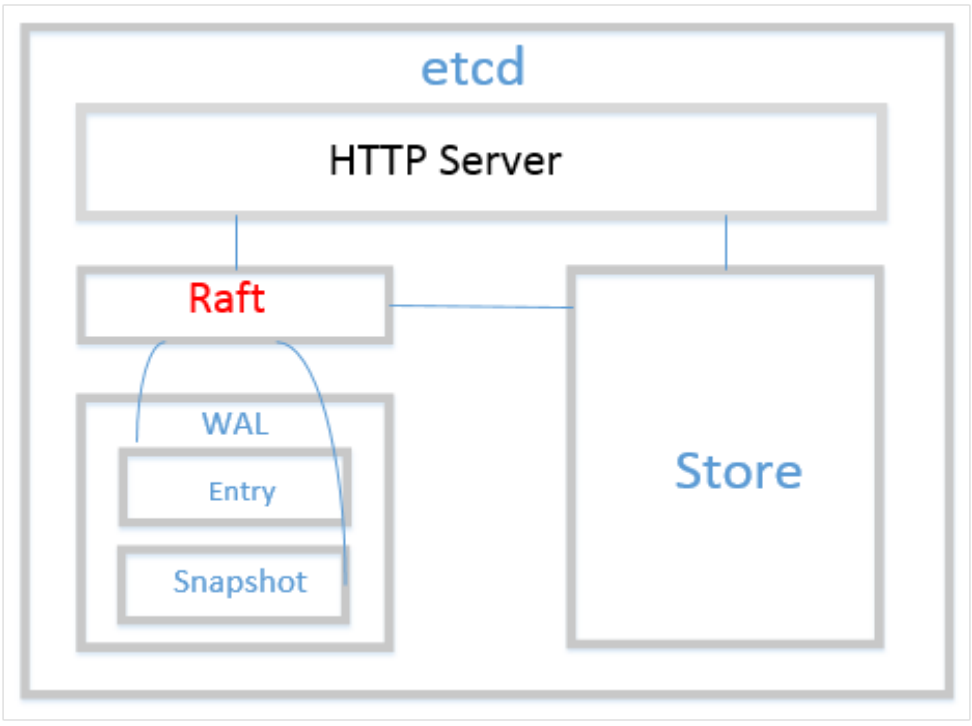
（图五）

*kube-proxy 将暴露出来的业务，通过 service 代理到业务 pod 及容器

*kubelet 负责将 node 系统数据上报给 apiserver，保存到 etcd 数据库

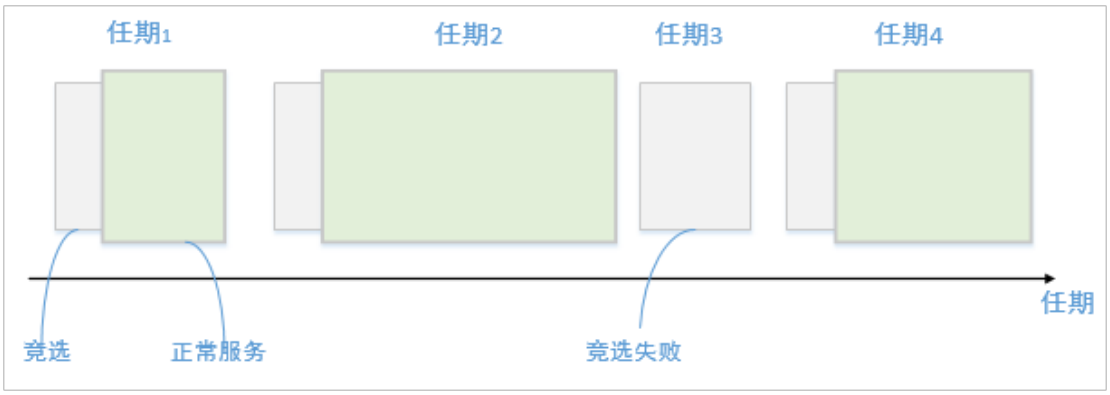
*kubelet 负责执行 kube-scheduler 下达的指令

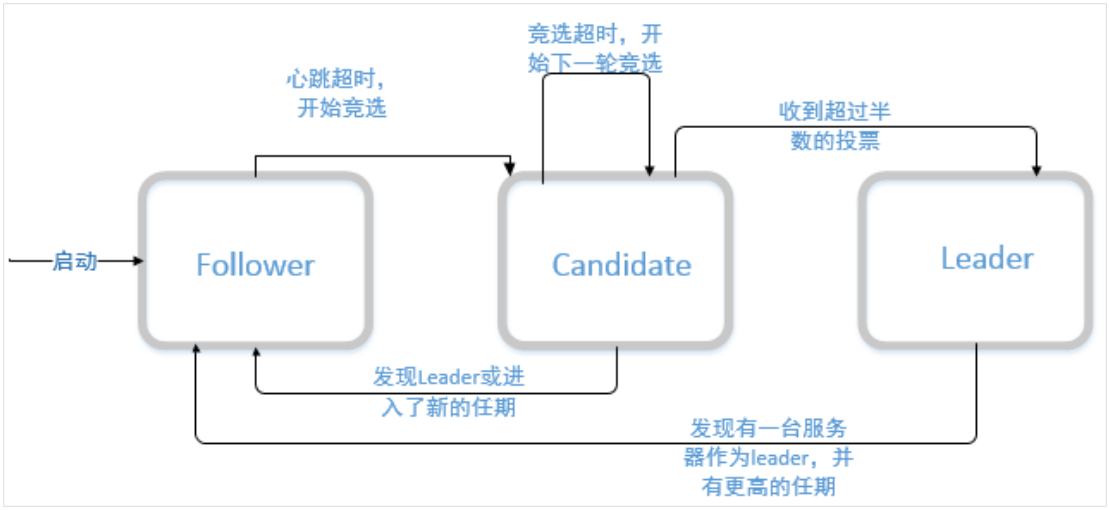
*docker 引擎，所有指令的落地



（图六）

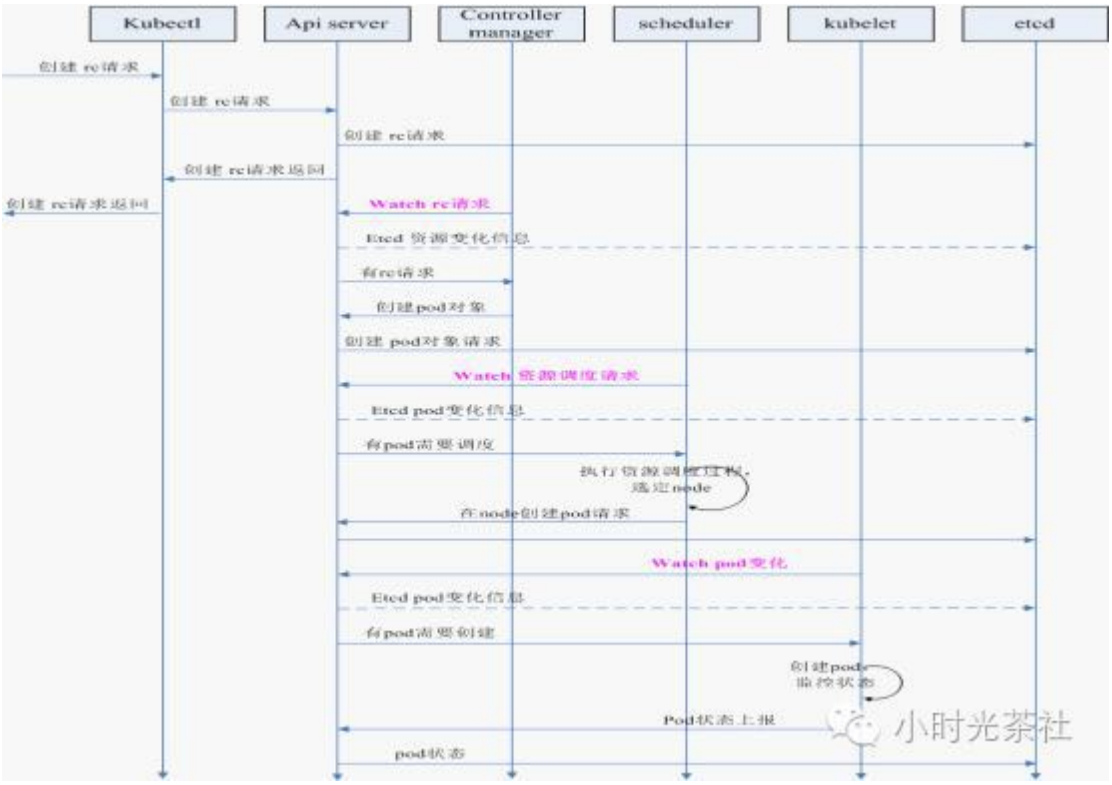
* etcd 服务 Raft 强一致性算法架构





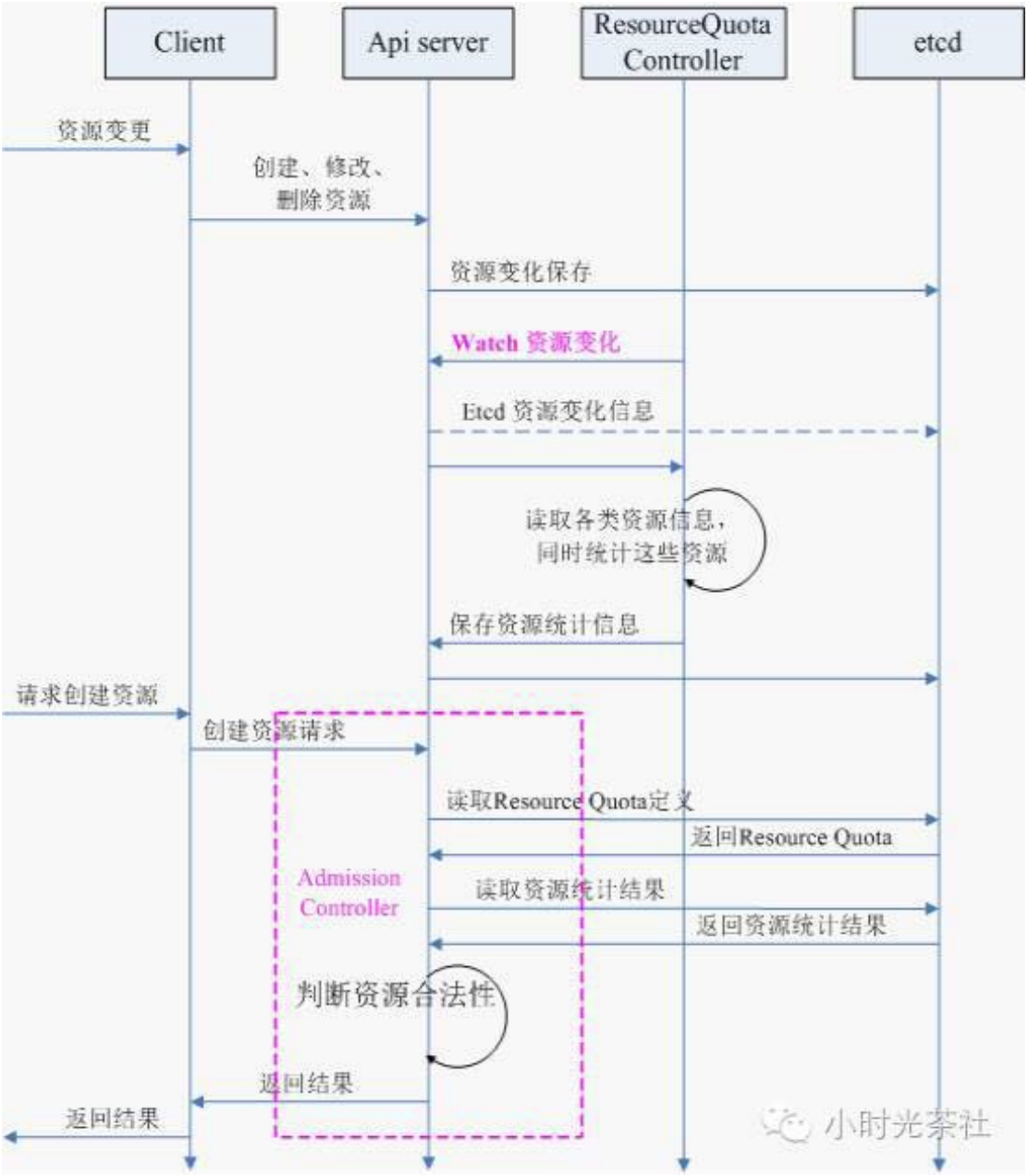
（图七）

* etcd 竞选机制，了解一下，后期深入掌握



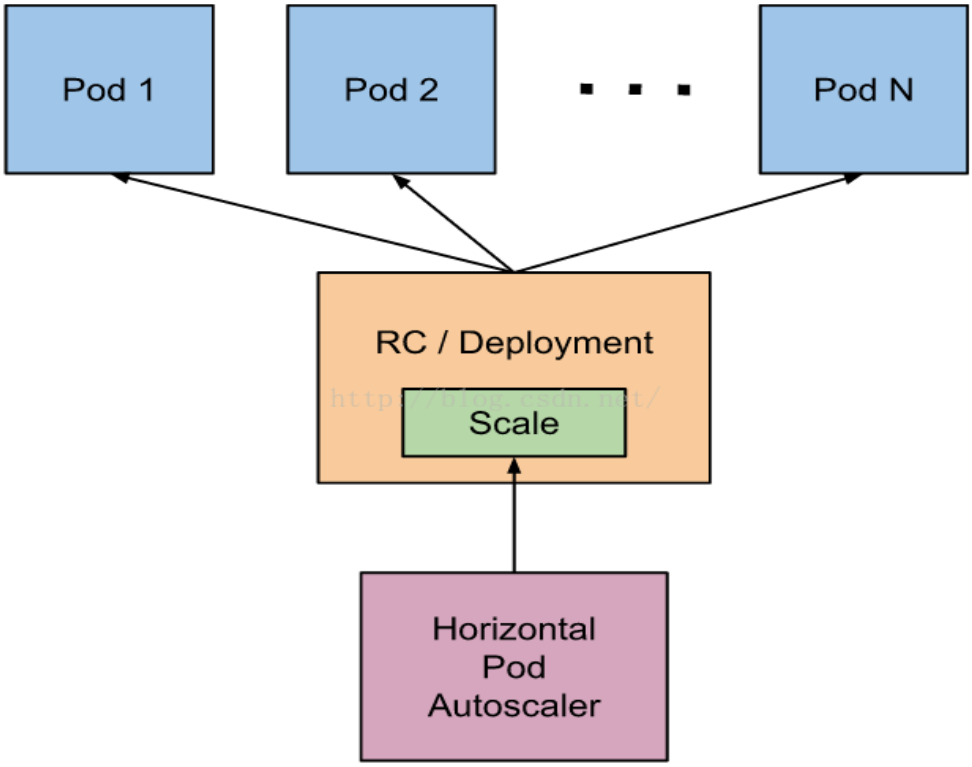
（图八）

* 主节点各组件调用关系及先后关系



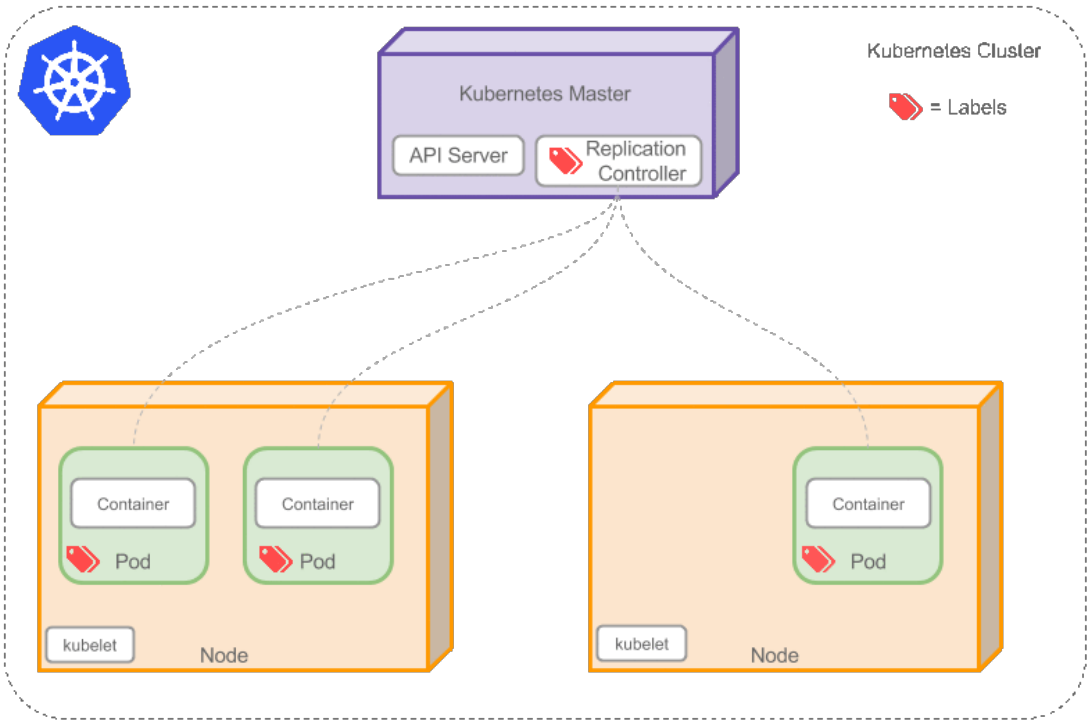
（图九）

* 集群中各类资源变更流程



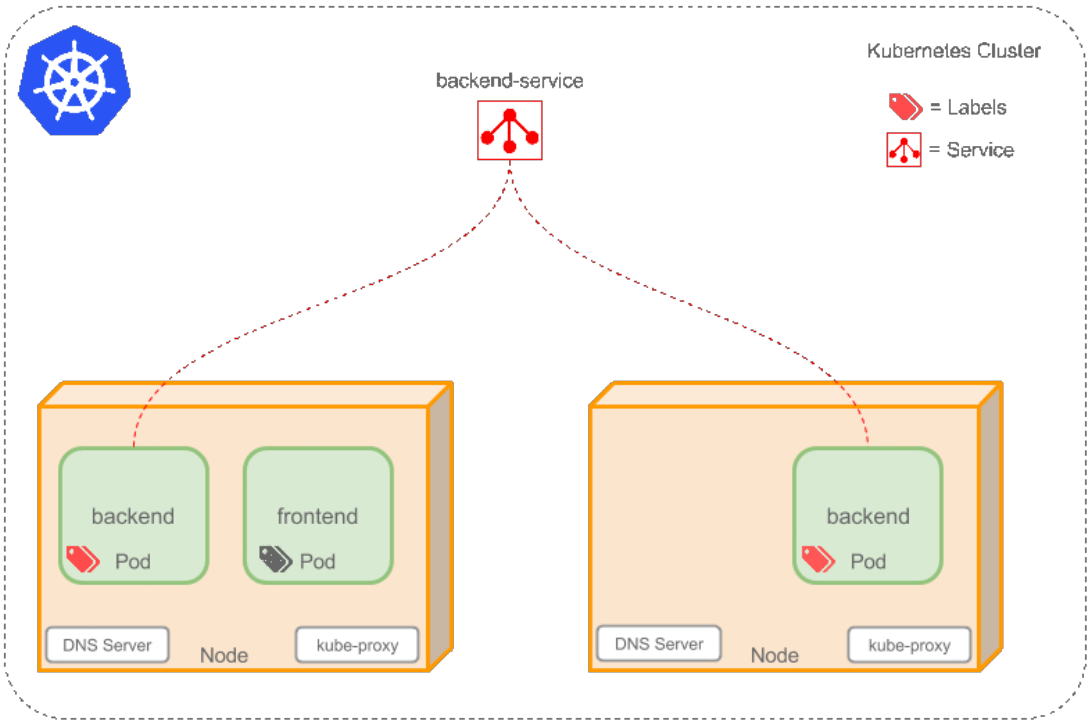
（图十）

*hpa 横向扩容机制



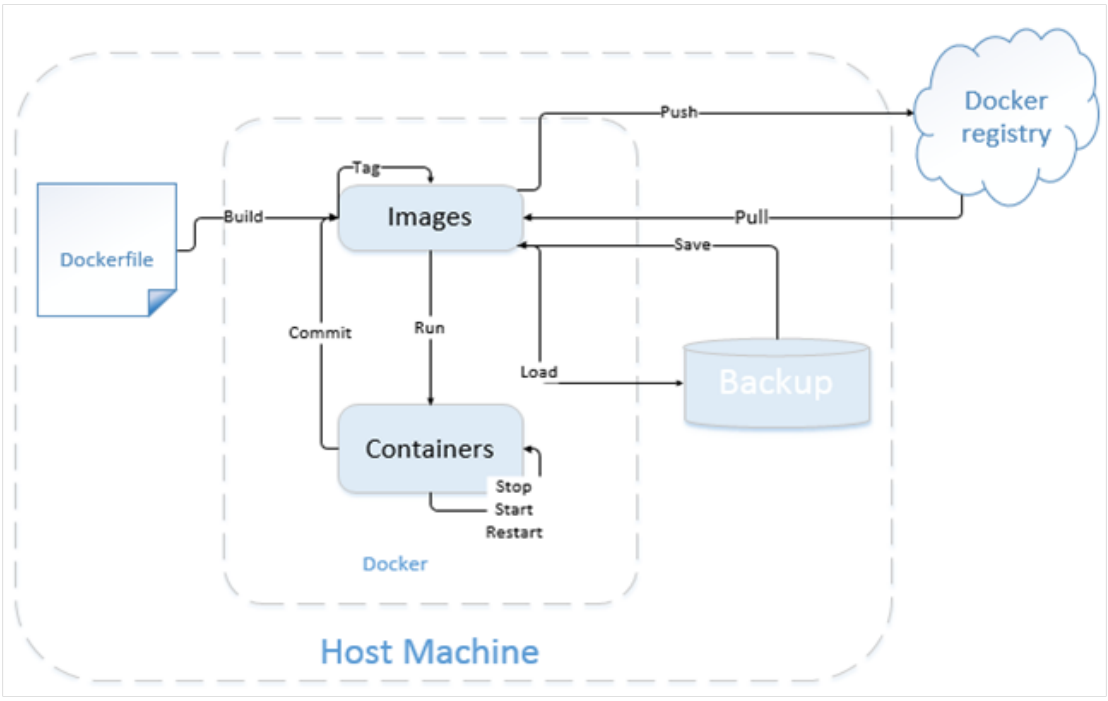
（图十一）

*k8s 平台 rc 通过标签机制调度 pod



（图十二）

*k8s 平台 service 机制，frontend->backend 通过 kube-proxy 负载均衡功能，映射到 backend pod，两次请求可能是不同 pod 响应

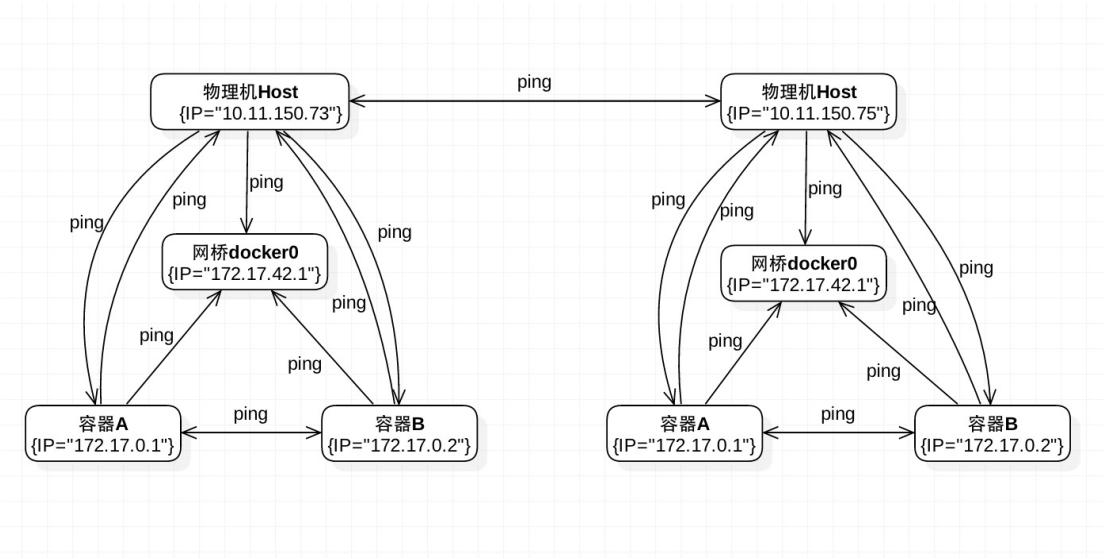


（图十三）

*业务镜像构建流程

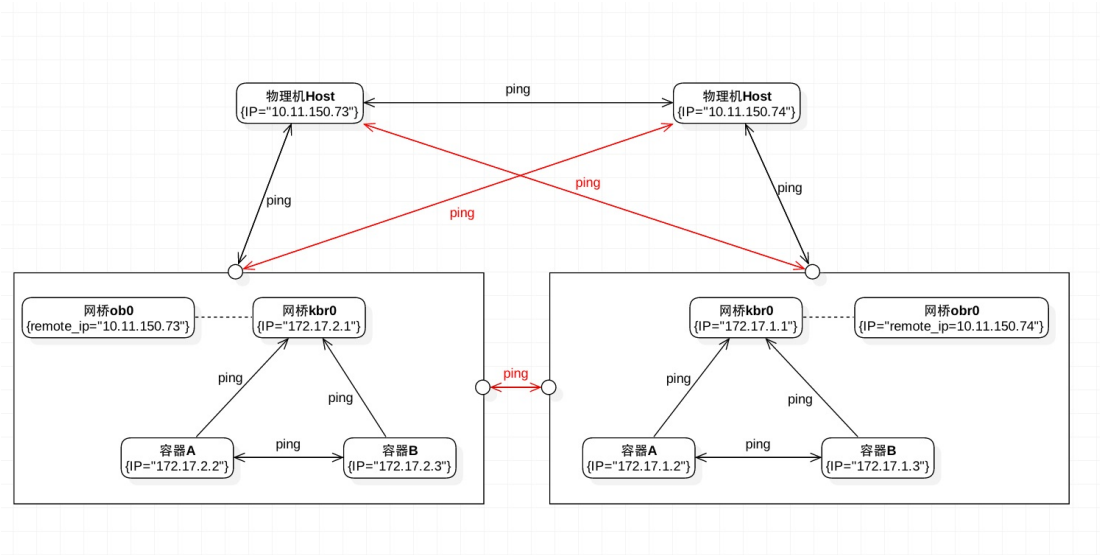
（图十四）

* k8s 平台主要资源及功能关系图



(图十五)

* 通过 openswitch vss 添加网桥，并添加路由规则实现节点与节点，节点与各容器，容器与各节点网络互通



（图十六）

* 节点各自容器网络隔离

2）、硬件部分

IP 地址	节点	Hostname	Os	Kernel
192.168.3.225	Master	3-225	CentOS 7.3.1611	4.12.5-1.el7.elrepo.x86_64
192.168.3.224	Node	3-224	CentOS 7.3.1611	4.12.5-1.el7.elrepo.x86_64
192.168.3.223	Node	3-223	CentOS 7.3.1611	4.12.5-1.el7.elrepo.x86_64

IP 地址	CPU	内存	容量
192.168.3.225	physical = 2, cores = 8, virtual = 8, hyperthreading = no R410	16G	300G
192.168.3.224	physical = 2, cores = 8, virtual = 16, hyperthreading = yes 12M 16xIntel(R) Xeon(R) CPU E5620 @ 2.40GHz R710	40G	900G
192.168.3.223	physical = 2, cores = 4, virtual = 8, hyperthreading = yes 2M 8xIntel(R) Xeon(TM) CPU 3.00GHz IBM x3650	4G	400G

3）、软件部分

1、操作系统

系统版本：CentOS Linux release 7.3.1611 (Core)

内核版本：Linux 4.12.5-1.el7.elrepo.x86_64

2、容器组件 docker

版本: Docker version 1.12.6, build 88a4867/1.12.6

3、平台软件 – kubernetes 及其组件

kube-scheduler: v1.5.1

kube-controller-manager: v1.5.1

kube-apiserver: v1.5.1

kubelet:v1.5.2

kube-proxy:v1.5.2

etcd: ovs-vsctl (Open vSwitch) 2.5.0 DB Schema 7.12.1

kube-dns:v1.7

kubernetes-dashboard: v1.6.1

heapster

monitoring-influxdb

nginx-ingress-controller

default-http-backend

openvswitch: openvswitch-2.5.0-2.el7.x86_64

haproxy: haproxy-1.6.3

keepalived

kubernetes 主要功能如下：

1）将多台 Docker 主机抽象为一个资源，以集群方式管理容器，包括任务调度、资源管理、弹性伸缩、滚动升级等功能。

2）使用编排系统（YAML File）快速构建容器集群，提供负载均衡，解决容器直接关联及通信问题

3）自动管理和修复容器，简单说，比如创建一个集群，里面有十个容器，如果某个容器异常关闭，那么，会尝试重启或重新分配容器，始终保证会有十个容器在运行，反而杀死多余的。

kubernetes 角色组成：

1）Pod

Pod 是 kubernetes 的最小操作单元，一个 Pod 可以由一个或多个容器组成；

同一个 Pod 只能运行在同一个主机上，共享相同的 volumes、network、namespace；

2）ReplicationController（RC）

RC 用来管理 Pod，一个 RC 可以由一个或多个 Pod 组成，在 RC 被创建后，系统会根据定义好的副本数来创建 Pod 数量。在运行过程中，如果 Pod 数量小于定义的，就会重启停止的或重新分配 Pod，反之则杀死多余的。当然，也可以动态伸缩运行的 Pods 规模。

RC 通过 label 关联对应的 Pods，在滚动升级中，RC 采用一个一个替换要更新的整个 Pods 中的 Pod。

3）Service

Service 定义了一个 Pod 逻辑集合的抽象资源，Pod 集合中的容器提供相同的功能。集合根据定义的 Label 和 selector 完成，当创建一个 Service 后，会分配一个 Cluster IP，这个 IP 与定义的端口提供这个集合一个统一的访问接口，并且实现负载均衡。

4）Label

Label 是用于区分 Pod、Service、RC 的 key/value 键值对；

Pod、Service、RC 可以有多个 label，但是每个 label 的 key 只能对应一个；

主要是将 Service 的请求通过 lable 转发给后端提供服务的 Pod 集合；

kubernetes 组件组成：

1）kubectl

客户端命令行工具，将接受的命令格式化后发送给 kube-apiserver，作为整个系统的操作入口。

2）kube-apiserver

作为整个系统的控制入口，以 REST API 服务提供接口。

3）kube-controller-manager

用来执行整个系统中的后台任务，包括节点状态状况、Pod 个数、Pods 和 Service 的关联等。

4）kube-scheduler

负责节点资源管理，接受来自 kube-apiserver 创建 Pods 任务，并分配到某个节点。

5）etcd

负责节点间的服务发现和配置共享。

6）kube-proxy

运行在每个计算节点上，负责 Pod 网络代理。定时从 etcd 获取到 service 信息来做相应的策略。

7）kubelet

运行在每个计算节点上，作为 agent，接受分配该节点的 Pods 任务及管理容器，周期性获取容器状态，反馈给 kube-apiserver。

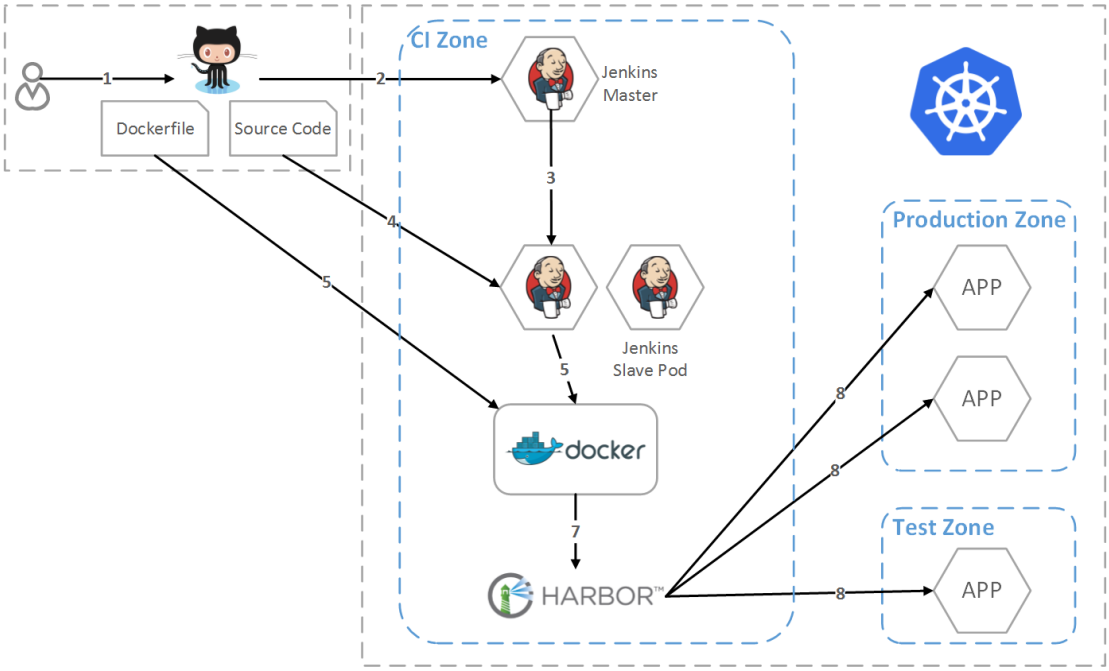
8）DNS

一个可选的 DNS 服务，用于为每个 Service 对象创建 DNS 记录，这样所有的 Pod 就可以通过 DNS 访问服务了。

4、镜像管理系统 - harbor

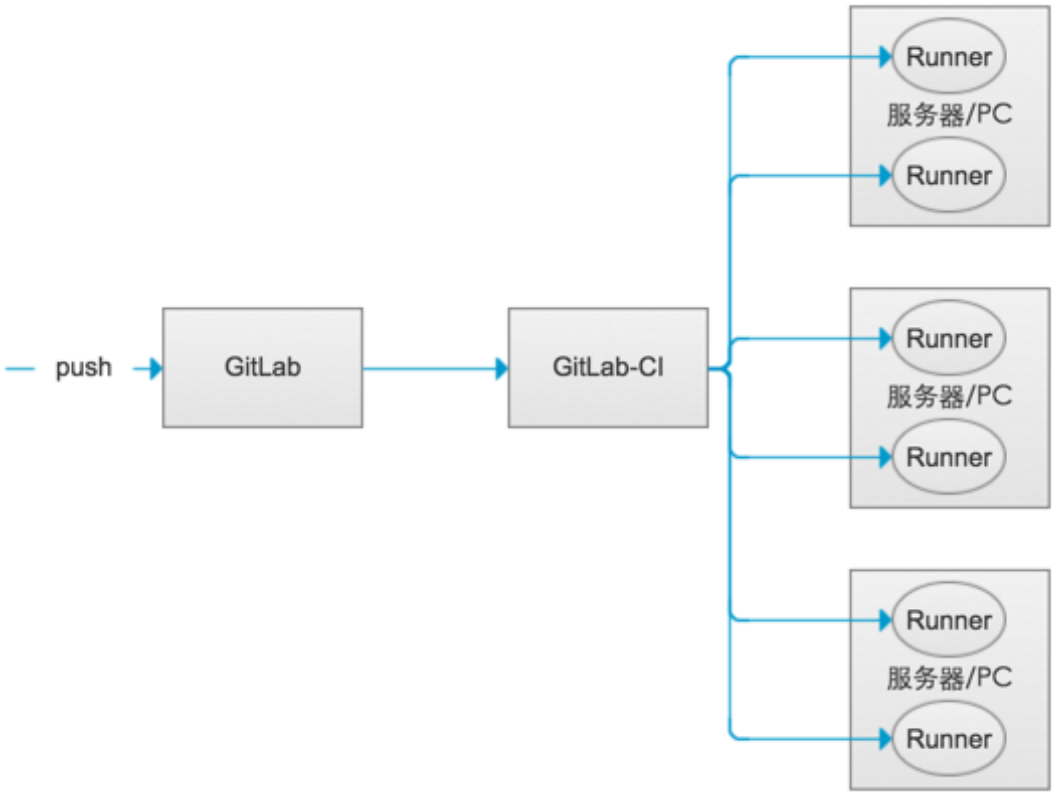
docker 模式运行，k8s 平台暂不兼容。

5、持续集成发布系统 - jenkins + gitlab + gitlab-runner



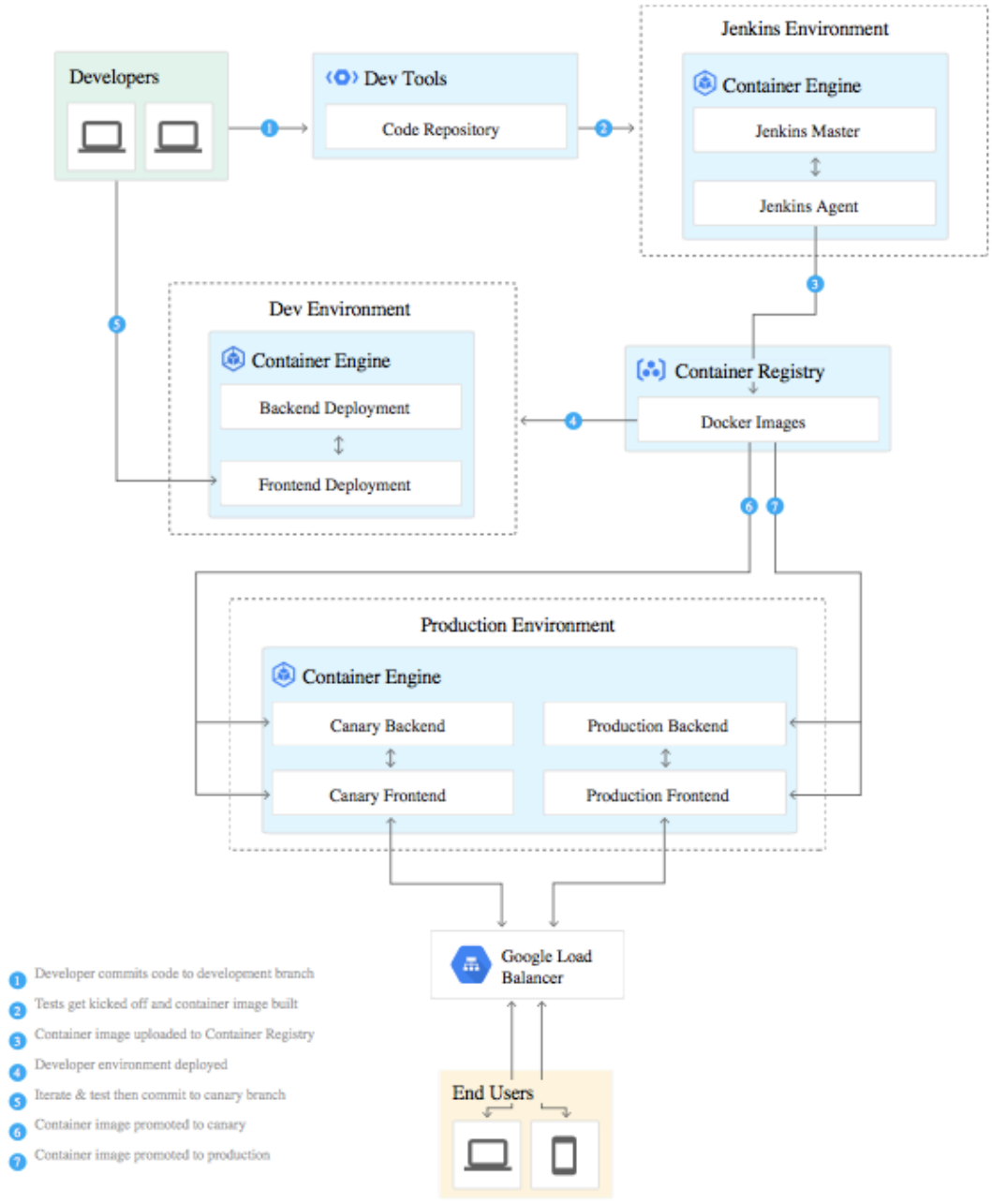
（图十七）

* CI/CD 流程：基于 jenkins master+slave 架构，从代码版本控制 pull 代码及 Dockerfile、Jenkinsfile 文件，通过 docker build 及 docker push 将新版镜像上传 harbor，再 k8s 平台 pull 镜像部署应用（test 和 prod）

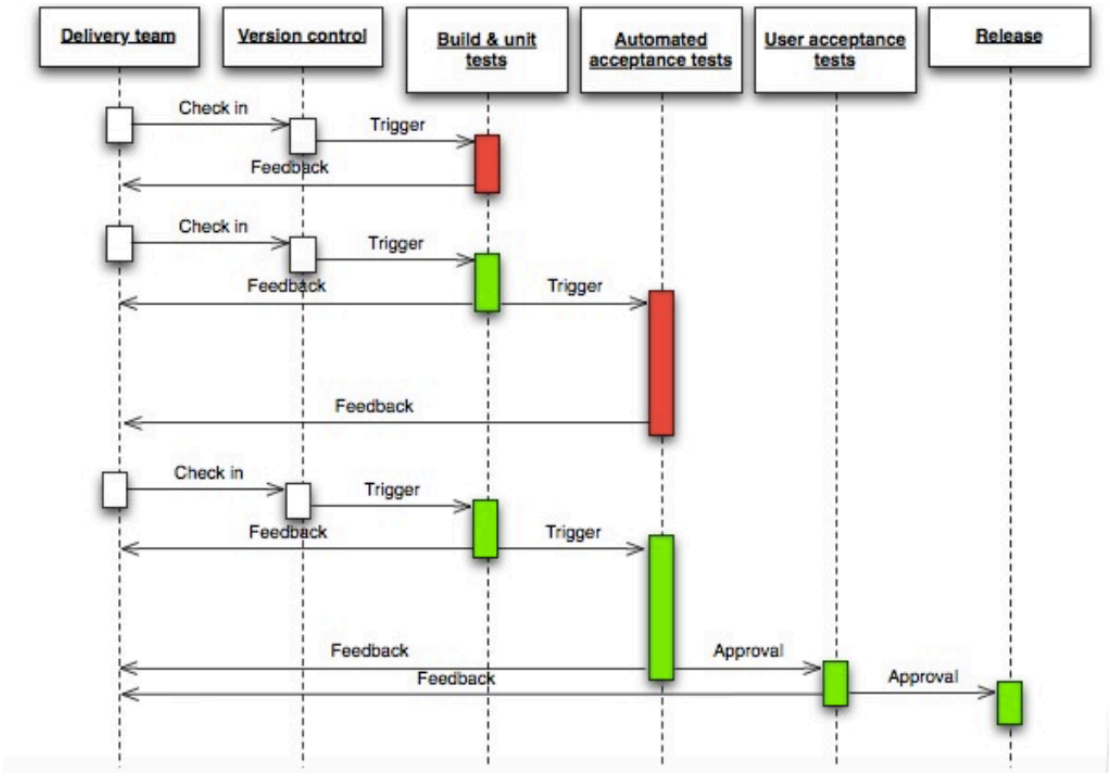


（图十八）

* 基于 gitlab-ci 与 gitlab-runner 机制，代码在 push 等操作时，触发 gitlab-ci 中定义的 runner 请求，调用 runner 定义好的部署步骤



（图十九）

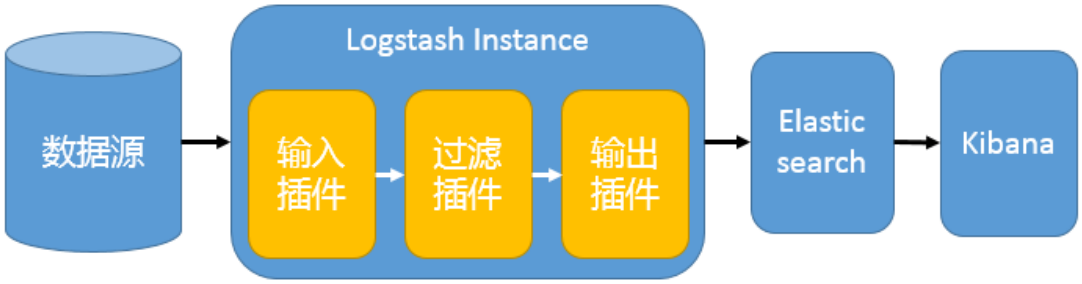


（图二十）

* 系统接入自动化集成发布之后的参考流程

6、elk 日志分析系统

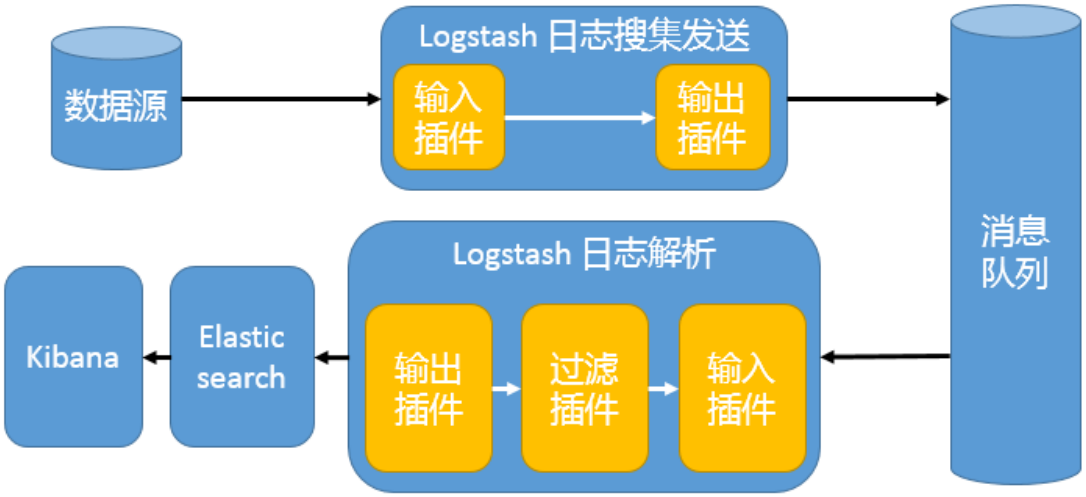
1）、elk 直线模式



（图二十一）

* 适用小规模日志分析需求，系统日志

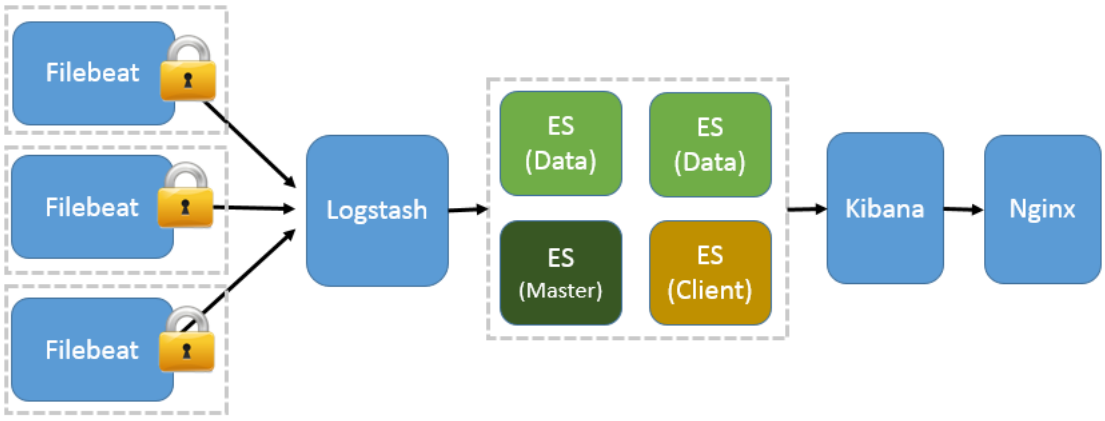
2）、elk 管道模式



（图二十二）

* 适用中型应用日志分析需求（pv <= 1000w）,比如：日志通过 logstash 代理格式化往 redis 推送，然后 elk 系统中 logstash 从 redis 拉数据处理给 elasticsearch 生成日志给 kibana 展示

3）、elk 集群模式



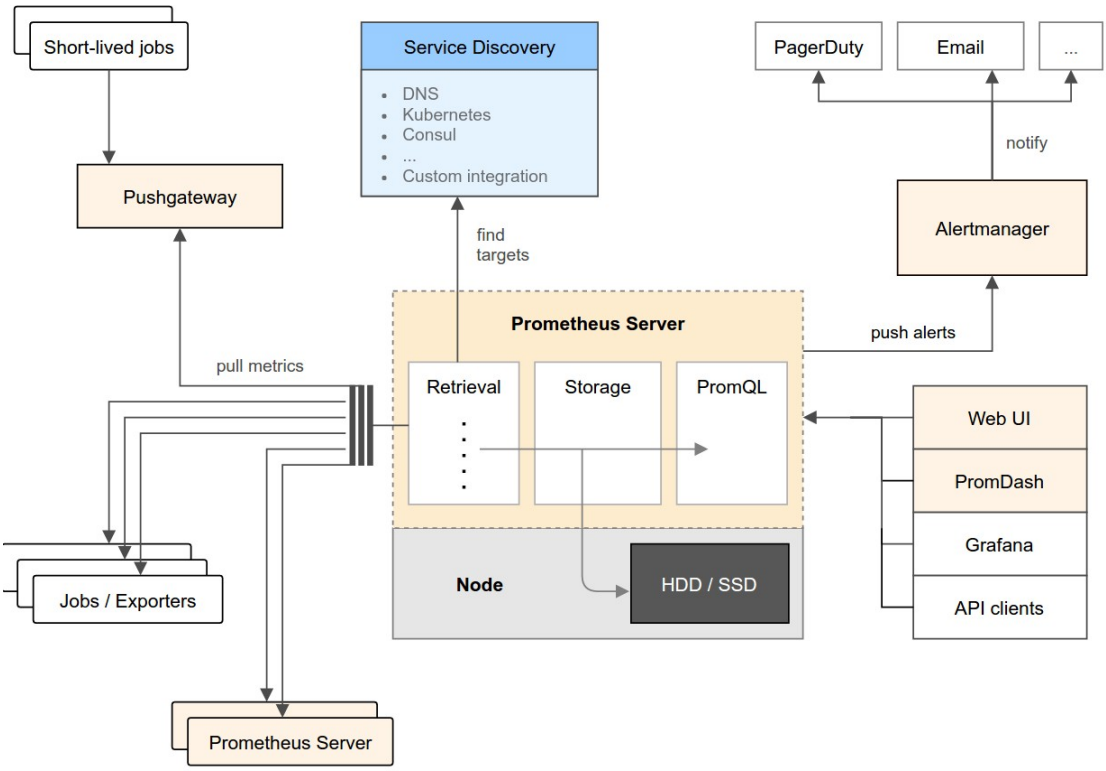
（图二十三）

* 结合 elasticsearch 支持的 master + client + data 集群功能，将 elastcsearch 功能分布式部署，充分利用 cpu 计算能力，提升整体系统性能和效率

7、监控系统

1）Prometheus + prometheus-node-exporter + node-directory-size-metrics+ altermanager + grafana

zabbix 辅助监控



（图二十四）

*详细搭建及配置文件查看附件

2）、sematext-agent

注册账号之后，登录后台创建监控项目，生成 token，按照说明启动 agent pod 即可收集监控数据

二、 平台详细部署细节

1、平台搭建

1）、服务器评测选购

详见浪潮及曙光测试报告

2）、基础环境搭建

a、操作系统安装

1）、下载 centos 7 操作系统镜像，制作 U 盘安装盘或者搭建基于 pxe 的 kickstart server

2）、制定存储分区策略，其它参考 centos 操作系统

b、系统优化

1)、系统优化

1. 更新系统并安装必备的组件

yum upgrade or yum update

yum install –y bash-completion wget bind-utils vim-enhanced tree telnet perl perl-devel net-tools kernel-devel epel-release

yum groupinstall "Development tools" -y

其中 net-tools 是为了提供 dig, nslookup, ipconfig 等命令。

2. 添加源（repository）REMI & EPEL

yum 安装时，要想安装比较新的版本软件，可以用一些国内镜像，比如：阿里云、163。

#163

http://mirrors.163.com/.help/centos.html

#国外

rpm -Uvh http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-5.noarch.rpm

rpm -Uvh http://rpms.famillecollet.com/enterprise/remi-release-7.rpm

#中科大镜像源

rpm -Uvh http://mirrors.ustc.edu.cn/centos/7.0.1406/extras/x86_64/Packages/epel-release-7-5.noarch.rpm

#浙大源

rpm -Uvh http://mirrors.zju.edu.cn/epel/7/x86_64/e/epel-release-7-5.noarch.rpm

#上海交大源

rpm -Uvh http://ftp.sjtu.edu.cn/fedora/epel/7/x86_64/e/epel-release-7-5.noarch.rpm

使用方法：

yum --enablerepo=remi install php mysql php-mysql mysql-server phpmyadmin

或者

yum --enablerepo=epel install php mysql php-mysql mysql-server phpmyadmin

3. FQDN 配置，全称 Fully Qualified Domain Name

有些软件，特别是邮件系统对这个要求比较高。

vi /etc/hosts

127.0.0.1 localhost.localdomain localhost geeker

::1 localhost.localdomain localhost geeker

vi /etc/sysconfig/network

HOSTNAME=geeker

设置好之后，查询是否完整

hostname -f

4. 关闭 Selinux

这是 Centos 系统的安装机制，单单往往导致很多软件无法正常安装，让我们关掉它吧！

/etc/selinux/config

在 SELINUX=enforcing 前面加个#号注释掉它

#SELINUX=enforcing

然后新加一行

SELINUX=disabled

#SELINUXTYPE=targeted #注释掉这行

保存，退出，重启系统，搞定。

5. CentOS 7 的防火墙关闭和 iptables 安装

CentOS 7.0 默认使用的是 firewall 作为防火墙，但可能一下子很难适应，让我们先改回原先的 iptables 防火墙吧！

关闭 CentOS 7 的 firewall：

systemctl stop firewalld.servic #停止 firewall

systemctl disable firewalld.service #禁止 firewall 开机启动

安装 iptables 防火墙

yum install iptables-services #安装

vi /etc/sysconfig/iptables #编辑防火墙配置文件

启动 iptables 防火墙

systemctl restart iptables.service #最后重启防火墙使配置生效

systemctl enable iptables.service #设置防火墙开机启动

6. 本地 SMTP 邮件发送功能（Postfix）

很多软件和服务可以用到这个功能给用户发送通知邮件，需要配置一下。

最好加上一个认证，使用 Postfix + Saslauthd

yum remove sendmail　#如果有原先的 sendmail，先移除

yum install postfix

vi /etc/postfix/main.cf　#编辑 postfix 主配置文件

useradd itgeeker　#增加用户

passwd itgeeker　#设置用户密码

yum install cyrus-sasl*

/bin/systemctl restart saslauthd.service && /bin/systemctl restart postfix.service　#启动 postfix 和 saslauth 服务

最好用 telnet 测试一下，前面安装的 telnet 就发挥作用了。

telnet localhost smtp

ehlo localhost

mail from:

rcpt to:<xxxx@qq.com>

data

Welcome to phpdba mail server

.

quit

#查看邮件内容

less /var/log/maillog

cd /root/Maildir/new #注意 M 要大写

ll

cat ***** #*代表列出的文件名，可以查看新的邮件内容

vi /var/log/maillog

Tips 小技巧：

有时候 telnet 登陆后就退不出来了 ctrl+c 也不管用此时可以使用 ctl+] 切换，然后 quit 退出。

7. CentOS 7 时间同步及更改

和之前基本一样：

date

yum install ntpdate -y

ntpdate ntp1.aliyun.com && hwclock -w

#连网更新时间，如果成功，将系统时间，写入 BOIS

hwclock -w 或 hwclock --systohc

8. Shell 登陆操作显示中文乱码问题

vi /etc/sysconfig/i18n 文件中修改 LANG 的设置为：

#LANG="en_US.UTF-8"

#SYSFONT="latarcyrheb-sun16"

LANG="zh_CN.GBK"

LANGUAGE="zh_CN.GBK:zh_CN.GB18030:zh_CN.GB2312:zh_CN"

SUPPORTED="zh_CN.GB18030:zh_CN:zh:en_US.UTF-8:en_US:en"

SYSFONT="lat0-sun16"

然后在/etc/profile 文件中增加 export LC_ALL=zh_CN.GBK 内容。使得全部的 LC*都统一

9、设置普通 username 有执行 sudo 的权限#visudo

在 root ALL=(ALL) ALL 行 下添加

username ALL=(ALL) ALL

2）、tcp 优化

vim /etc/sysctl.d/centos-7-sysctl.conf

#CTCDN 系统优化参数

#关闭 ipv6

net.ipv6.conf.all.disable_ipv6 = 1

net.ipv6.conf.default.disable_ipv6 = 1

避免放大攻击

net.ipv4.icmp_echo_ignore_broadcasts = 1

开启恶意 icmp 错误消息保护

net.ipv4.icmp_ignore_bogus_error_responses = 1

#关闭路由转发

net.ipv4.ip_forward = 1

#net.ipv4.conf.all.send_redirects = 0

#net.ipv4.conf.default.send_redirects = 0

#开启反向路径过滤

net.ipv4.conf.all.rp_filter = 1

net.ipv4.conf.default.rp_filter = 1

#处理无源路由的包

#net.ipv4.conf.all.accept_source_route = 0

#net.ipv4.conf.default.accept_source_route = 0

#关闭 sysrq 功能

kernel.sysrq = 0

#core 文件名中添加 pid 作为扩展名

kernel.core_uses_pid = 1

开启 SYN 洪水攻击保护

net.ipv4.tcp_syncookies = 1

#修改消息队列长度

kernel.msgmnb = 65536

kernel.msgmax = 65536

#设置最大内存共享段大小 bytes

kernel.shmmax = 68719476736

kernel.shmall = 4294967296

#timewait 的数量，默认 180000

net.ipv4.tcp_max_tw_buckets = 6000

net.ipv4.tcp_sack = 1

net.ipv4.tcp_window_scaling = 1

net.ipv4.tcp_rmem = 4096 87380 4194304

net.ipv4.tcp_wmem = 4096 16384 4194304

net.core.wmem_default = 8388608

net.core.rmem_default = 8388608

net.core.rmem_max = 16777216

net.core.wmem_max = 16777216

#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 = 262144

#限制仅仅是为了防止简单的 DoS 攻击

net.ipv4.tcp_max_orphans = 3276800

#未收到客户端确认信息的连接请求的最大值

net.ipv4.tcp_max_syn_backlog = 262144

net.ipv4.tcp_timestamps = 0

#内核放弃建立连接之前发送 SYNACK 包的数量

net.ipv4.tcp_synack_retries = 1

#内核放弃建立连接之前发送 SYN 包的数量

net.ipv4.tcp_syn_retries = 1

#启用 timewait 快速回收

net.ipv4.tcp_tw_recycle = 1

#开启重用。允许将 TIME-WAIT sockets 重新用于新的 TCP 连接

net.ipv4.tcp_tw_reuse = 1

net.ipv4.tcp_mem = 94500000 915000000 927000000

net.ipv4.tcp_fin_timeout = 1

#当 keepalive 起用的时候，TCP 发送 keepalive 消息的频度。缺省是 2 小时

net.ipv4.tcp_keepalive_time = 30

#允许系统打开的端口范围

net.ipv4.ip_local_port_range = 1024 65000

#修改防火墙表大小，默认 65536

#net.netfilter.nf_conntrack_max=655350

#net.netfilter.nf_conntrack_tcp_timeout_established=1200

确保无人能修改路由表

net.ipv4.conf.all.accept_redirects = 0

net.ipv4.conf.default.accept_redirects = 0

net.ipv4.conf.all.secure_redirects = 0

net.ipv4.conf.default.secure_redirects = 0

3）、io 优化

略

3）、网络架构设计构建

a、软件安装

1）、openvswitch

[root@3-225 ~]# cat /etc/yum.repos.d/naulinux-extras.repo

[naulinux-extras]

name=NauLinux Extras

baseurl=http://downloads.naulinux.ru/pub/NauLinux/7/$basearch/Extras/RPMS/

enabled=0

gpgcheck=1

gpgkey=http://downloads.naulinux.ru/pub/NauLinux/RPM-GPG-KEY-linux-ink

[root@3-225 ~]# yum --enablerepo=naulinux-extras install openvswitch

[root@3-225 ~]# yum -y install bridge-utils

[root@3-225 ~]# ovs-vsctl add-br ob0

2）、Flannel

[root@3-225 ~]# wget https://github.com/coreos/flannel/releases/download/v0.7.1/flannel-v0.7.1-linux-amd64.tar.gz

[root@3-225 ~]# mkdir /usr/src/flannel

[root@3-225 ~]# tar zxf flannel-v0.7.1-linux-amd64.tar.gz -C /usr/src/flannel

[root@3-225 ~]# cp /usr/src/flannel/{flanneld,mk-docker-opts.sh} /usr/local/bin/

[root@3-225 ~]# etcdctl --endpoints "http://192.168.3.225:2379" mkdir /phpdba.com/network

[root@3-225 ~]# etcdctl --endpoints "http://192.168.3.225:2379" set /phpdba.com/network/config '{"NetWork":"10.0.0.0/16", "Backend": {"Type": "vxlan"}}'

[root@3-225 ~]#cat > /usr/lib/systemd/system/flanneld.service <<EOF

[Unit]

Description=Flanneld overlay address etcd agent

After=network.target

After=network-online.target

Wants=network-online.target

After=etcd.service

Before=docker.service

[Service]

Type=notify

ExecStart=/root/local/bin/flanneld \

-etcd-endpoints=${ETCD_ENDPOINTS} \

-etcd-prefix=${FLANNEL_ETCD_PREFIX} \

$FLANNEL_OPTIONS

ExecStartPost=/root/local/bin/mk-docker-opts.sh -k DOCKER_NETWORK_OPTIONS -d /run/flannel/docker

Restart=on-failure

[Install]

WantedBy=multi-user.target

RequiredBy=docker.service

EOF

***下载地址：https://github.com/coreos/flannel/releases，选择对应的版本下载**

*** flannel 默认使用 8285 端口作为 UDP 封装报文的端口，VxLan 使用 8472 端口**

*** flanneld、mk-docker-opts.sh 这两个文件，其中 flanneld 为主要的执行文件，sh 脚本用于生成 Docker 启动参数**

*** flannel backend 为 vxlan 比起预设的 udp 性能相对好一些。写入的 Pod 网段(10.244.0.0/16) 必须与 kube-controller-manager 的 --cluster-cidr 选项值一致**

b、服务配置

[root@3-225 ~]# cat /etc/sysconfig/network-scripts/ifcfg-br225

DEVICE=br225

ONBOOT=yes

DEVICETYPE=ovs

TYPE=OVSBridge

HOTPLUG=no

USERCTL=no

[root@3-225 ~]# cat /etc/sysconfig/network-scripts/ifcfg-docker225

DEVICE=docker225

ONBOOT=yes

BOOTPROTO=static

IPADDR=10.0.225.1

NETMASK=255.255.255.0

GATEWAY=10.0.225.0

USERCTL=no

TYPE=Bridge

IPV6INIT=no

[root@3-225 network-scripts]# cat route-em1

10.0.223.0/24 via 192.168.3.223 dev em1

10.0.224.0/24 via 192.168.3.224 dev em1

10.0.226.0/24 via 192.168.3.226 dev em1

[root@3-225 ~]# systemctl restart network

4）、平台软件部署

a、docker

CentOS 7 中 Docker 软件包已经包括在默认的 CentOS-Extras 软件源里。因此想要安装 docker，只需要运行下面的 yum 命令：

[root@3-225 ~]# yum install docker -y

[root@3-225 ~]# docker –version //验证安装是否成功

Docker version 1.12.6, build 88a4867/1.12.6

[root@3-225 ~]# vim /etc/sysconfig/docker

OPTIONS='--selinux-enabled --log-driver=journald --signature-verification=false --storage-driver=overlay -b docker225'

[root@3-225 ~]# cat /etc/docker/daemon.json //加速器配置

```
{
   "registry-mirrors": ["https://3d13mnz1.mirror.aliyuncs.com"]
}
```

* 参考上述安装步骤，安装各个节点 docker 服务

b、etcd

[root@3-225 ~]# yum install etcd -y

[root@3-225 ~]# cat /etc/etcd/etcd.conf |grep -v '^#'

```
ETCD_NAME=infra225
ETCD_DATA_DIR="/var/lib/etcd/infra225.etcd"
ETCD_LISTEN_PEER_URLS="http://192.168.3.225:2380"
ETCD_LISTEN_CLIENT_URLS="http://192.168.3.225:2379,http://127.0.0.1:2379"
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://192.168.3.225:2380"
ETCD_INITIAL_CLUSTER="infra224=http://192.168.3.224:2380,infra225=http://192.168.3.225:2380,infra223=http://192.168.3.223:2380"
ETCD_INITIAL_CLUSTER_STATE="new"
ETCD_INITIAL_CLUSTER_TOKEN="etcdk8s"
ETCD_ADVERTISE_CLIENT_URLS="http://192.168.3.225:2379"
```

[root@3-225 ~]# cat /usr/lib/systemd/system/etcd.service

```
[Unit]
Description=Etcd Server
After=network.target
After=network-online.target
Wants=network-online.target

[Service]
Type=notify
WorkingDirectory=/var/lib/etcd/
```

EnvironmentFile=-/etc/etcd/etcd.conf

User=etcd

set GOMAXPROCS to number of processors

ExecStart=/bin/bash -c "GOMAXPROCS=$(nproc) /usr/bin/etcd --name=\"${ETCD_NAME}\" --data-dir=\"${ETCD_DATA_DIR}\" --listen-peer-urls=\"${ETCD_LISTEN_PEER_URLS}\" --listen-client-urls=\"${ETCD_LISTEN_CLIENT_URLS}\" --advertise-client-urls=\"${ETCD_ADVERTISE_CLIENT_URLS}\" --initial-cluster-token=\"${ETCD_INITIAL_CLUSTER_TOKEN}\" --initial-cluster=\"${ETCD_INITIAL_CLUSTER}\" --initial-cluster-state=\"${ETCD_INITIAL_CLUSTER_STATE}\""

#ExecStart=/bin/bash -c "GOMAXPROCS=$(nproc) /usr/bin/etcd --name=\"${ETCD_NAME}\" --data-dir=\"${ETCD_DATA_DIR}\" --listen-client-urls=\"${ETCD_LISTEN_CLIENT_URLS}\""

Restart=on-failure

LimitNOFILE=65536

[Install]

WantedBy=multi-user.target

[root@3-225 ~]# systemctl start etcd

[root@3-225 ~]# etcdctl member list

34c0e6ebccaa9dd0: name=infra224 peerURLs=http://192.168.3.224:2380 clientURLs=http://192.168.3.224:2379 isLeader=true

43125075e28da97c: name=infra225 peerURLs=http://192.168.3.225:2380 clientURLs=http://192.168.3.225:2379 isLeader=false

851cabd90657702d: name=infra223 peerURLs=http://192.168.3.223:2380 clientURLs=http://192.168.3.223:2379 isLeader=false

*** 上述标红部分根据 IP 位数修改**

以上配置项说明:

--name　　etcd 集群中的节点名，这里可以随意，可区分且不重复就行

---listen-peer-urls　　监听的用于节点之间通信的 url，可监听多个，集群内部将通过这些 url 进行数据交互(如选举，数据同步等)

---initial--advertise-peer-urls　　建议用于节点之间通信的 url，节点间将以该值进行通信。

---listen-client-urls　　监听的用于客户端通信的 url,同样可以监听多个。

---advertise-client-urls　　建议使用的客户端通信 url,该值用于 etcd 代理或 etcd 成员与 etcd 节点通信。

---initial-cluster-token etcdk8s　　节点的 token 值，设置该值后集群将生成唯一 id,并为每个节点也生成唯一 id,当使用相同配置文件再启动一个集群时，只要该 token 值不一样，etcd 集群就不会相互影响。

---initial-cluster　　也就是集群中所有的 initial--advertise-peer-urls 的合集

---initial-cluster-state new　　新建集群的标志

c、kubernetes

1、三种安装模式方式

1）、[root@3-225 ~]# yum install kubernetes –y //centos 官方源

2）、[root@3-225 ~]#rpm --import https://www.elrepo.org/RPM-GPG-KEY-elrepo.org

[root@3-225 ~]#rpm -Uvh http://www.elrepo.org/elrepo-release-7.0-3.el7.elrepo.noarch.rpm

[root@3-225 ~]# yum install kubernetes –y

3）、[root@3-225 ~]#cat ／etc/yum.repos.d/virt7-docker-common-release.repo

[virt7-docker-common-release]

name=virt7-docker-common-release

baseurl=http://cbs.centos.org/repos/virt7-docker-common-release/x86_64/os/

gpgcheck=0

[root@3-225 ~]# yum -y install --enablerepo=virt7-docker-common-release kubernetes

[root@3-225 ~]#

2、集群模式搭建

[root@3-225 kubernetes]# pwd

/etc/kubernetes

[root@3-225 kubernetes]# tree

.

├── apiserver

├── config

├── controller-manager

├── kubelet

├── manifests

│ ├── kube-apiserver.yaml

│ ├── kube-controller-manager.yaml

│ └── kube-scheduler.yaml

├── proxy

└── scheduler

1 directory, 9 files

[root@3-225 kubernetes]# cat apiserver |grep -v '^$'|grep -v '^#'

KUBE_API_ADDRESS="--insecure-bind-address=0.0.0.0"

KUBE_API_PORT="--port=8080"

KUBELET_PORT="--kubelet-port=10250"

KUBE_ETCD_SERVERS="--etcd-servers=http://192.168.3.225:2379"

KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=10.254.0.0/16"

KUBE_ADMISSION_CONTROL="--admission-

control=NamespaceLifecycle,NamespaceExists,LimitRanger,SecurityContextDeny,ResourceQuota"

KUBE_API_ARGS=""

[root@3-225 kubernetes]# cat config |grep -v '^$'|grep -v '^#'

KUBE_LOGTOSTDERR="--logtostderr=true"

KUBE_LOG_LEVEL="--v=0"

KUBE_ALLOW_PRIV="--allow-privileged=false"

KUBE_MASTER="--master=http://127.0.0.1:8080"

[root@3-225 kubernetes]# cat kubelet |grep -v '^$'|grep -v '^#'

KUBELET_ADDRESS="--address=0.0.0.0"

KUBELET_HOSTNAME="--hostname-override=192.168.3.225"

KUBELET_API_SERVER="--api-servers=http://127.0.0.1:8080"

KUBELET_POD_INFRA_CONTAINER="--pod-infra-container-image=registry.access.redhat.com/rhel7/pod-

infrastructure:latest"

KUBELET_ARGS="--cluster-dns=10.254.0.100 --cluster-domain=cluster.local --allow-privileged=true --

config=**/etc/kubernetes/manifests**"

[root@3-225 kubernetes]# cat proxy |grep -v '^$'|grep -v '^#'

KUBE_PROXY_ARGS=""

[root@3-225 kubernetes]# cat scheduler |grep -v '^$'|grep -v '^#'

KUBE_SCHEDULER_ARGS=""

[root@3-225 manifests]# pwd

/etc/kubernetes/manifests

[root@3-225 manifests]# ls

kube-apiserver.yaml kube-controller-manager.yaml kube-scheduler.yaml

*** yaml 文件详细内容查看附件**

```
for SERVICES in flanneld docker kube-proxy kubelet ; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

3、效验

```
[root@3-225 ~]# kubectl get nodes
NAME                STATUS      AGE
192.168.3.223       Ready       38d
192.168.3.224       Ready       38d
192.168.3.225       Ready       38d
[root@3-225 ~]# kubectl get svc
NAME                CLUSTER-IP          EXTERNAL-IP     PORT(S)           AGE
kubernetes          10.254.0.1          <none>          443/TCP           38d
nginx-php5-fpm      10.254.30.197       <nodes>         80:30088/TCP      20d
[root@3-225 ~]# kubectl get ns
NAME                STATUS      AGE
chen-123            Active      37d
default             Active      38d
dev                 Active      15d
efk-logs            Active      7d
kube-system         Active      38d
monitoring          Active      37d
production          Active      15d
```

5）、平台监控

a）、granfa

https://grafana-prod.phpdba.com

b）、zabbix

暂不部署

6）、平台测试验证

a）、cli

kubectl xxx

b）、ui

https://192.168.0.1:8443

c）、api

https://192.168.0.1:6443

7）、业务部署

a）、kube-dns

详见配置文件 目录：/root/chen-123/ha-k8s

b）、heapster

详见配置文件 目录：/root/chen-123/ha-k8s

c）、nginx-controller

详见配置文件 目录：/root/chen-123/ha-k8s

d）、gitlab gitlab-runner

详见配置文件 目录：/root/chen-123/ha-k8s

e）、jenkins master+slave

详见配置文件 目录：/root/chen-123/ha-k8s

f）、elk 日志分析管理系统

一、日志分析系统服务器端

1、服务部署

直接制作 elk 镜像或者使用网站共享的镜像不属

[root@3-224 ~]# docker pull sebp/elk

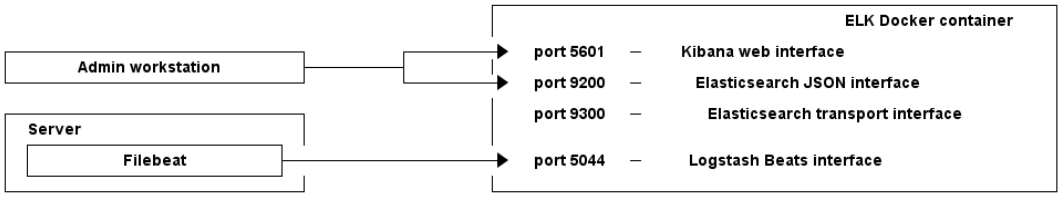
[root@3-224 ~]# docker run -p 5601:5601 -p 9200:9200 -p 9300:9300 -p 5044:5044 -it --name elk sebp/elk

[root@3-224 ~]# docker ps -a|grep elk

a82e702a1c3b 23d7754bfa62 "/usr/local/bin/start"

10 days ago Up About an hour 0.0.0.0:5044->5044/tcp, 0.0.0.0:5601->5601/tcp,

0.0.0.0:9200->9200/tcp, 0.0.0.0:9300->9300/tcp elk



（图二十四）

*** 5601 (Kibana web interface).**

*** 9200 (Elasticsearch JSON interface).**

*** 9200 (Elasticsearch's transport interface).**

*** 5044 (Logstash Beats interface, receives logs from Beats).**

**Example: docker run -p 5601:5601 -p 9200:9200 -p 5044:5044 -it **

-e LOGSTASH_START=0 -e KIBANA_START=0 –e TZ= 'Asia/shanghai' --name elk sebp/elk

*** ELASTICSEARCH_START: if set and set to anything other than 1, then Elasticsearch will not be started.**

*** LOGSTASH_START: if set and set to anything other than 1, then Logstash will not be started.**

*** KIBANA_START: if set and set to anything other than 1, then Kibana will not be started.**

二、日志分析系统收集客户端

1、安装软件

1.1、logstash 安装

[root@3-224 ~]# rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch

[root@3-224 ~]# cat > /etc/yum.repos.d/logstatsh.repo << EOF

[logstash-5.x]

name=Elastic repository for 5.x packages

baseurl=https://artifacts.elastic.co/packages/5.x/yum

gpgcheck=1

gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch

enabled=1

autorefresh=1

type=rpm-md

EOF

[root@3-224 ~]# yum install logstash -y

1.2、topbeat 安装

[root@3-224 ~]# curl -L -O https://download.elastic.co/beats/topbeat/topbeat-1.3.1-x86_64.rpm

[root@3-224 ~]# rpm -vi topbeat-1.3.1-x86_64.rpm

2、配置软件

2.1、logstash 配置

[root@3-224 logstash-all]# pwd

/etc/logstash-all

[root@3-224 logstash-all]# grep -v '^#' logstash.yml|grep -v '^$'

node.name: test-all

path.data: /var/lib/logstash-all

pipeline.workers: 2

pipeline.output.workers: 1

pipeline.batch.size: 125

pipeline.batch.delay: 5

path.config: /etc/logstash/conf.d

path.logs: /var/log/logstash-all

[root@3-224 logstash-all]# cat conf.d/syslog.conf

```
  input {
    file {
      type => "allsyslog"
      #path => [ "/var/log/*.log", "/var/log/messages", "/var/log/syslog" ]
      path => [ "/var/log/kube-apiserver.log" ]
      start_position => beginning
      #sincedb_path => "/opt/logstash/sincedb-access"
    }
    tcp {
      port => 5000
      type => syslog
    }
    udp {
      port => 5000
      type => syslog
    }
  }


  filter {
    if [type] == "syslog" {
            grok {
            match => { "message" =>
"%{SYSLOGTIMESTAMP:syslog_timestamp} %{SYSLOGHOST:syslog_hostname} %{DATA:syslog_program}(?:\[%{POSINT:syslo
g_pid}\])?: %{GREEDYDATA:syslog_message}" }
                    add_field => [ "received_at", "%{@timestamp}" ]
                    add_field => [ "received_from", "%{host}" ]
            }
```

```
            syslog_pri { }

            date {
                match => [ "syslog_timestamp", "MMM  d HH:mm:ss", "MMM dd HH:mm:ss" ]
            }

    }else if [type] == "allsyslog" {
    #I0920 09:24:15.961348             7 handlers.go:162] GET /api/v1/namespaces/kube-system/endpoints/kube-controller-manager:
(1.711216ms) 200 [[kube-controller-manager/v1.6.8 (linux/amd64) kubernetes/d74e09b/leader-election] 127.0.0.1:17938]

    grok {
    match => { "message" => "(?<sverity>([A-
Z]{1}))(?<syslog_timestamp>(%{MONTHNUM2}%{MONTHDAY}%{SPACE}%{TIME}))%{SPACE}%{POSINT:syslog_pid}%{S
PACE}%{DATA:syslog_program}\]%{SPACE}%{WORD:syslog_methed}%{SPACE}%{URIPATHPARAM:syslog_uri}\:\s*\(%{DA
TA:syslog_reqtime}\)\s*%{POSINT:syslog_reqstatus}\s*\[\[%{GREEDYDATA:syslog_ua}\] %{GREEDYDATA:syslog_client_port}\]
" }
    }
   }

if "_grokparsefailure" in [tags] {
      grok {
            match => ["message", "(?<sverity>([A-
Z]{1}))(?<syslog_timestamp>(%{MONTHNUM2}%{MONTHDAY}%{SPACE}%{TIME}))%{SPACE}%{POSINT:syslog_pid}%{S
PACE}%{DATA:syslog_program}\]%{GREEDYDATA:syslog_str}"]
            remove_field => ["message"]
            remove_tag => ["_grokparsefailure"]
            add_field => {
                LogSource => "-"
                LogLevel => "-"
                LogTime => "-"
    }
        }
}
```

```
}


output {
   elasticsearch {
    hosts => "192.168.3.224:9200"
    index => "syslog-%{+YYYY.MM.dd}"
   }
   stdout { codec => rubydebug }
}
```

[root@3-224 ~]# logstash –f /etc/logstash-all/conf.d/syslog.conf

（图二十五）

2、topbeat 配置

[root@3-224 topbeat]# pwd

/etc/topbeat

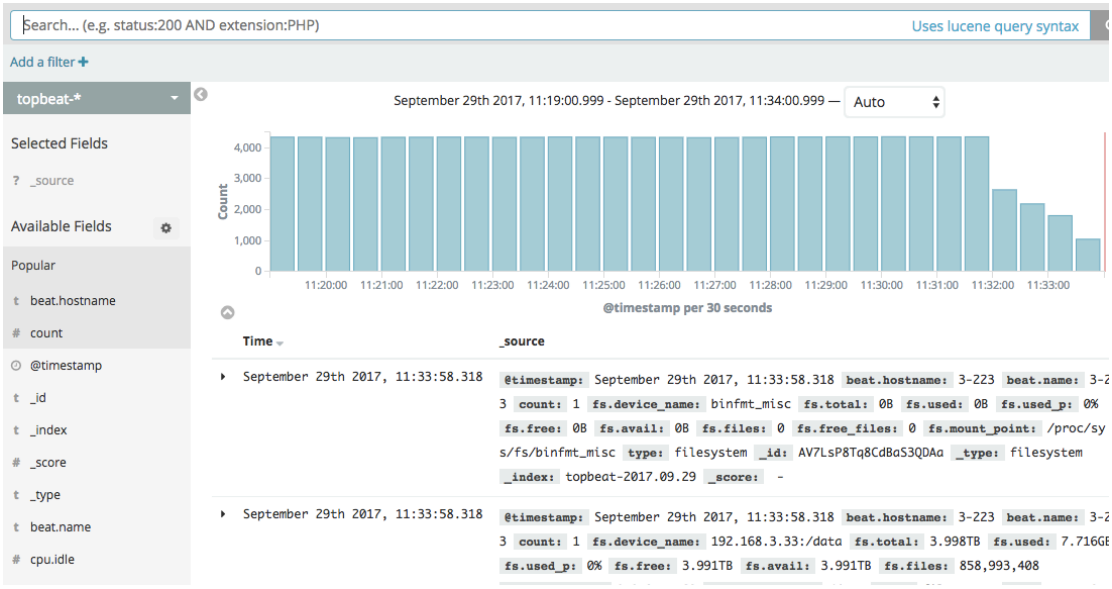
[root@3-224 topbeat]# curl -XPUT 'http://localhost:9200/_template/topbeat' -d@/etc/topbeat/topbeat.template.json

[root@3-224 topbeat]# grep -v '^\s*#' topbeat.yml |grep -v '^$'

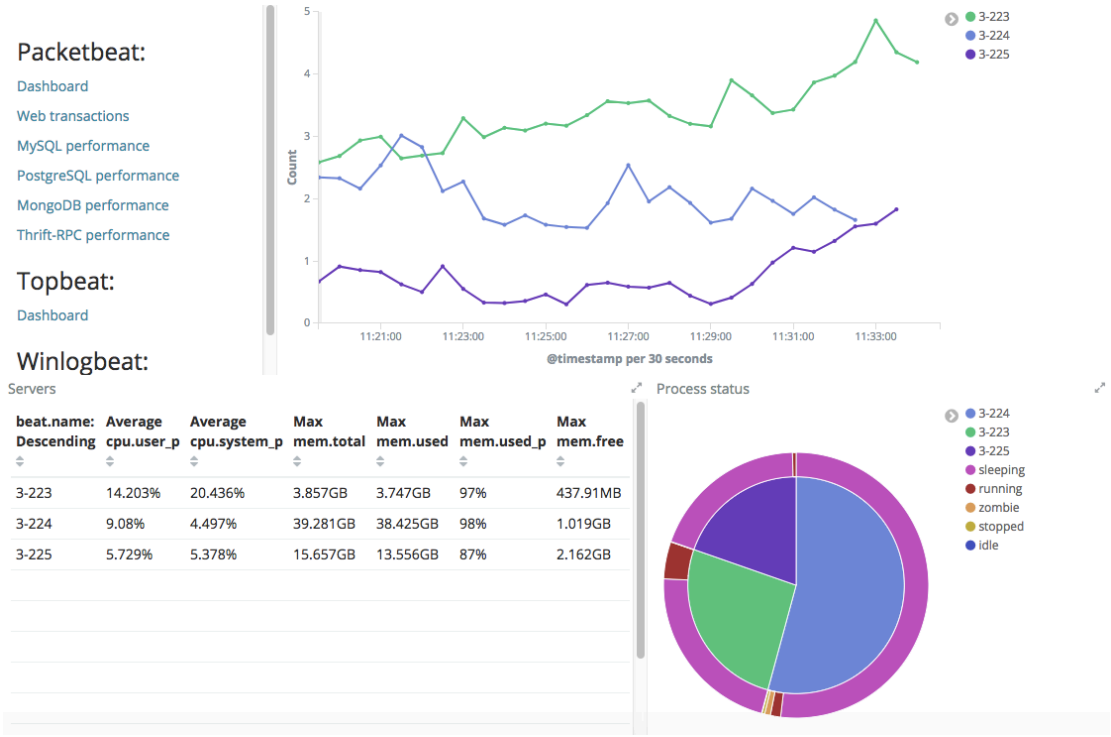
```
input:
  period: 10
  procs: [".*"]
  stats:
    system: true
    process: true
    filesystem: true
```

```
    cpu_per_core: false
output:
  elasticsearch:
    hosts: ["192.168.3.224:9200"]
shipper:
logging:
  files:
    rotateeverybytes: 10485760 # = 10MB
```

[root@3-224 topbeat]# systemctl start topbeat



（图二十六）



beat.name: Descending	Average cpu.user_p	Average cpu.system_p	Max mem.total	Max mem.used	Max mem.used_p	Max mem.free
3-223	14.203%	20.436%	3.857GB	3.747GB	97%	437.91MB
3-224	9.08%	4.497%	39.281GB	38.425GB	98%	1.019GB
3-225	5.729%	5.378%	15.657GB	13.556GB	87%	2.162GB

（图二十七）

8）、平台运维

a）、问题故障列表

1、服务告警不可用，node 异常下线导致，重启恢复

解决办法：后期完善脚本，检查自动恢复常规故障机制

b）、升级维护日志

c）、告警阈值

https://prom-prod.phpdba.com

d）、常规操作

1、停止所有的 container，这样才能够删除其中的 images：

docker stop $(docker ps -a -q)

如果想要删除所有 container 的话再加一个指令：

docker rm $(docker ps -a -q)

2、查看当前有些什么 images

docker images

3、删除 images，通过 image 的 id 来指定删除谁

docker rmi <image id>

想要删除 untagged images，也就是那些 id 为<None>的 image 的话可以用

docker rmi $(docker images | grep "^<none>" | awk "{print $3}")

要删除全部 image 的话

docker rmi $(docker images -q)

三、平台机制介绍

1）、资源限制

Kubernetes 通过 cgroups 限制容器的 CPU 和内存等计算资源，包括 requests（请求，调度器保证调度到资源充足的 Node 上）和 limits（上限）等：

spec.containers[].resources.limits.cpu：CPU 上限，可以短暂超过，容器也不会被停止

spec.containers[].resources.limits.memory：内存上限，不可以超过；如果超过，容器可能会被停止或调度到其他资源充足的机器上

spec.containers[].resources.requests.cpu：CPU 请求，可以超过

spec.containers[].resources.requests.memory：内存请求，可以超过；但如果超过，容器可能会在 Node 内存不足时清理

比如 nginx 容器请求 30%的 CPU 和 56MB 的内存，但限制最多只用 50%的 CPU 和 128MB 的内存：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
      resources:
        requests:
          cpu: "300m"
          memory: "56Mi"
```

```
        limits:
          cpu: "500m"
          memory: "128Mi"
```

注意，CPU 的单位是 milicpu，500mcpu=0.5cpu；而内存的单位则包括 E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki 等。

2）、Pod 资源的共享及通信

1、Pod 使 Pod 内的数据共享及通信变得容易

2、Pod 的中的应用均使用相同的网络命名空间及端口，并且可以通过 localhost 发现并沟通其他应用，每个 Pod 都有一个扁平化的网络命名空间下 IP 地址，它是 Pod 可以和其他的物理机及其他的容器进行无障碍通信，（The hostname is set to the pod's Name for the application containers within the pod）主机名被设置为 Pod 的名称

3、除了定义了在 Pod 中运行的应用之外，Pod 还定义了一系列的共享的磁盘，磁盘让这些数据在容器重启的时候不会丢失，并且可以将这些数据在 Pod 中的应用进行共享

4、Pod 通过提供一个高层次抽象而不是底层的接口简化了应用的部署及管理，Pod 作为最小的部署及管理单位，位置管理，拷贝复制，资源共享，依赖关系都是自动处理的。

3）、Ingress

1、通常情况下，service 和 pod 的 IP 仅可在集群内部访问。集群外部的请求需要通过负载均衡转发到 service 在 Node 上暴露的 NodePort 上，然后再由 kube-proxy 将其转发给相关的 Pod。

2、Ingress 为进入集群的请求提供路由规则的集合，如下图所示

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

Ingress 可以给 service 提供集群外部访问的 URL、负载均衡、SSL、HTTP 路由等。

3、为了配置 Ingress 规则，集群管理员需要部署一个 Ingress controller，它监听 Ingress 和 service 的变化，并根据规则配置负载均衡并提供访问入口。每个 Ingress 都需要配置 rules，目前 Kubernetes 仅支持 http 规则。

4、根据 Ingress Spec 配置的不同，Ingress 可以分为以下几种类型：

1）、单服务 Ingress

单服务 Ingress 即该 Ingress 仅指定一个没有任何规则的后端服务。

```
apiVersion: extensions/v1beta1
kind: Ingress
```

```
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

注：单个服务还可以通过设置 Service.Type=NodePort 或者 Service.Type=LoadBalancer 来对外暴露。

路由到多服务的 Ingress

2）、路由到多服务的 Ingress 即根据请求路径的不同转发到不同的后端服务上，比如

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

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

使用 kubectl create -f 创建完 ingress 后：

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

3）、虚拟主机 Ingress

虚拟主机 Ingress 即根据名字的不同转发到不同的后端服务上，而他们共用同一个的 IP 地址，如下所示

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

基于 Host header 路由请求的 Ingress：

注：没有定义规则的后端服务称为默认后端服务，可以用来方便的处理 404 页面。

4）、TLS Ingress

TLS Ingress 通过 Secret 获取 TLS 私钥和证书(名为 tls.crt 和 tls.key)，来创建 TLS 证书。如果 Ingress 中的 TLS 配置部分指定了不同的主机，则它们将根据通过 SNI TLS 扩展指定的主机名（假如 Ingress controller 支持 SNI）在多个相同端口上进行复用。

定义一个包含 tls.crt 和 tls.key 的 secret：

```
apiVersion: v1
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
kind: Secret
metadata:
  name: xxx-secret
  namespace: xxxx
type: Opaque
```

Ingress 中引用 secret：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: xxxx
```

```
spec:
  tls:
    - secretName: xxx-secret
  backend:
    serviceName: xxxxx-xxx
    servicePort: xxx
```

注意：不同 Ingress controller 支持的 TLS 功能不尽相同。 请参阅有关 Ingress controller 的文档，了解 TLS 的支持情况。

4）、CronJob

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

.spec.schedule 指定任务运行周期，格式同 Cron

.spec.jobTemplate 指定需要运行的任务，格式同 Job

.spec.startingDeadlineSeconds 指定任务开始的截止期限

.spec.concurrencyPolicy 指定任务的并发策略，支持 Allow、Forbid 和 Replace 三个选项

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

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

5）、Job

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

Kubernetes 支持以下几种 Job：

非并行 Job：通常创建一个 Pod 直至其成功结束

固定结束次数的 Job：设置.spec.completions，创建多个 Pod，直到.spec.completions 个 Pod 成功结束

带有工作队列的并行 Job：设置.spec.Parallelism 但不设置.spec.completions，当所有 Pod 结束并且至少一个成功时，Job 就认为是成功

根据.spec.completions 和.spec.Parallelism 的设置，可以将 Job 划分为以下几种 pattern：

Job 类型	使用示例	行为	c
一次性 Job	数据库迁移	创建一个 Pod 直至其成功结束	1
固定结束次数的 Job	处理工作队列的 Pod	依次创建一个 Pod 运行直至 completions 个成功结束	2
固定结束次数的并行 Job	多个 Pod 同时处理工作队列	依次创建多个 Pod 运行直至 completions 个成功结束	2
并行 Job	多个 Pod 同时处理工作队列	创建一个或多个 Pod 直至有一个成功结束	1

Job Controller 负责根据 Job Spec 创建 Pod，并持续监控 Pod 的状态，直至其成功结束。如果失败，则根据 restartPolicy（只支持 OnFailure 和 Never，不支持 Always）决定是否创建新的 Pod 再次重试任务。

Job Spec 格式

spec.template 格式同 Pod

RestartPolicy 仅支持 Never 或 OnFailure

单个 Pod 时，默认 Pod 成功运行后 Job 即结束

.spec.completions 标志 Job 结束需要成功运行的 Pod 个数，默认为 1

.spec.parallelism 标志并行运行的 Pod 的个数，默认为 1

spec.activeDeadlineSeconds 标志失败 Pod 的重试最大时间，超过这个时间不会继续重试

6)、HostAliases

Adding Additional Entries with HostAliases

In addition to the default boilerplate, we can add additional entries to the hosts file to resolve foo.local, bar.local to 127.0.0.1 and foo.remote, bar.remote to 10.1.2.3, we can by adding HostAliases to the Pod under .spec.hostAliases:

```
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  restartPolicy: Never
  hostAliases:
  - ip: "127.0.0.1"
    hostnames:
    - "foo.local"
    - "bar.local"
  - ip: "10.1.2.3"
    hostnames:
    - "foo.remote"
    - "bar.remote"
  containers:
  - name: cat-hosts
    image: busybox
    command:
    - cat
```

args:

- "/etc/hosts"

7）、HPA（Horizontal Pod Autoscaling）

Horizontal Pod Autoscaling (HPA) 可以根据 CPU 使用率或应用自定义 metrics 自动扩展 Pod 数量（支持 replication controller、deployment 和 replica set ）。

- 控制管理器每隔30s（可以通过--horizontal-pod-autoscaler-sync-period修改）查询metrics的资源使用情况
- 支持三种metrics类型
  - 预定义metrics（比如Pod的CPU）以利用率的方式计算
  - 自定义的Pod metrics，以原始值（raw value）的方式计算
  - 自定义的object metrics
- 支持两种metrics查询方式：Heapster和自定义的REST API
- 支持多metrics

示例

```
# 创建 pod 和 service
$ kubectl run php-apache --image=gcr.io/google_containers/hpa-example --
requests=cpu=200m --expose --port=80
service "php-apache" created
deployment "php-apache" created

# 创建 autoscaler
$ kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
deployment "php-apache" autoscaled
$ kubectl get hpa
NAME        REFERENCE                     TARGET    CURRENT   MINPODS   MAXPODS   AGE
php-apache   Deployment/php-apache/scale   50%       0%        1         10        18s

# 增加负载
$ kubectl run -i --tty load-generator --image=busybox /bin/sh
```

```
Hit enter for command prompt

$ while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done


# 过一会就可以看到负载升高了

$ kubectl get hpa

NAME        REFERENCE                   TARGET    CURRENT   MINPODS   MAXPODS   AGE

php-apache   Deployment/php-apache/scale   50%      305%     1        10       3m


# autoscaler 将这个 deployment 扩展为 7 个 pod

$ kubectl get deployment php-apache

NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE

php-apache   7        7        7           7          19m


# 删除刚才创建的负载增加 pod 后会发现负载降低，并且 pod 数量也自动降回 1 个

$ kubectl get hpa

NAME        REFERENCE                   TARGET    CURRENT   MINPODS   MAXPODS   AGE

php-apache   Deployment/php-apache/scale   50%      0%       1        10       11m


$ kubectl get deployment php-apache

NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE

php-apache   1        1        1           1          27m
```

自定义metrics

使用方法

- 控制管理器开启--horizontal-pod-autoscaler-use-rest-clients
- 控制管理器的--apiserver指向API Server Aggregator
- 在API Server Aggregator中注册自定义的metrics API

比如 HorizontalPodAutoscaler 保证每个 Pod 占用 50% CPU、1000pps 以及 10000 请求/s：

HPA 示例

```
apiVersion: autoscaling/v2alpha1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
  - type: Pods
    pods:
      metricName: packets-per-second
      targetAverageValue: 1k
  - type: Object
    object:
      metricName: requests-per-second
      target:
        apiVersion: extensions/v1beta1
        kind: Ingress
        name: main-route
      targetValue: 10k
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
```

```
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
      currentAverageUtilization: 0
      currentAverageValue: 0
```

状态条件

v1.7+可以在客户端中看到 Kubernetes 为 HorizontalPodAutoscaler 设置的状态条件 `status.conditions`，用来判断 HorizontalPodAutoscaler 是否可以扩展（AbleToScale）、是否开启扩展（ScalingActive）以及是否受到限制（ScalingLimitted）。

```
$ kubectl describe hpa cm-test
Name:                       cm-test
Namespace:                   prom
Labels:                     <none>
Annotations:                 <none>
CreationTimestamp:            Fri, 16 Jun 2017 18:09:22 +0000
Reference:                  ReplicationController/cm-test
Metrics:                    ( current / target )
  "http_requests" on pods:     66m / 500m
Min replicas:                1
Max replicas:                4
ReplicationController pods:    1 current / 1 desired
Conditions:
  Type                Status  Reason               Message
  ----                ------  ------               -------
  AbleToScale         True    ReadyForNewScale       the last scale time was sufficiently
old as to warrant a new scale
  ScalingActive       True    ValidMetricFound       the HPA was able to successfully
calculate a replica count from pods metric http_requests
```

```
  ScalingLimited       False   DesiredWithinRange     the desired replica count is within
the acceptible range
Events:
```

8）、Service Account

Service account 是为了方便 Pod 里面的进程调用 Kubernetes API 或其他外部服务而设计的。它与 User account 不同

a、User account 是为人设计的，而 service account 则是为 Pod 中的进程调用 Kubernetes API 而设计；

b、User account 是跨 namespace 的，而 service account 则是仅局限它所在的 namespace；

c、每个 namespace 都会自动创建一个 default service account

d、Token controller 检测 service account 的创建，并为它们创建 secret

e、开启 ServiceAccount Admission Controller 后

1、每个 Pod 在创建后都会自动设置 spec.serviceAccount 为 default（除非指定了其他 ServiceAccout）

2、验证 Pod 引用的 service account 已经存在，否则拒绝创建

3、如果 Pod 没有指定 ImagePullSecrets，则把 service account 的 ImagePullSecrets 加到 Pod 中

4、每个 container 启动后都会挂载该 service account 的 token 和 ca.crt 到 /var/run/secrets/kubernetes.io/serviceaccount/

四、为什么是 Kubernetes（k8s）？

（1）什么是 Kubernetes?

Kubernetes 是一个管理集群主机间自动化部署、扩展和容器操作，且以容器为中心基础设施的开源软件平台。

Kubernetes 始于 Google 2014 年的一个项目。 Kubernetes 的构建基于 Google 十多年运行大规模负载产品的经验，同时也吸取了社区中最好的意见和经验。**以“一切以服务为中心，一切围绕服务运转”为指导思想。**

通过 Kubernetes,可以快速有效地响应用户需求：

快速而有预期地部署你的应用

极速地扩展你的应用

无缝铺展新的应用功能

节省资源，优化硬件资源的使用

k8s 的目标是孕育一个组件和工具的生态系统，以资源在公有云和私有云中运行应用。

Kubernetes 特点:

可移植: 支持公有云，私有云，混合云，复合云

可扩展: 模块化，热插拔，可组合

自愈: 自动替换，自动重启，自动复制，自动扩展

（2）为什么要选择容器？

a、当容器技术这么热门的时候，你是不是在疑惑为什么要选用这样的技术呢？

1）、传统的应用部署方式是通过操作系统的包管理器来安装应用。然而这样做的一个劣势在于，它把应用的运行，配置，库和生存周期和机器的操作系统纠缠在一起。当然也可以通过创建虚机镜像的方式来获得可以预期的前滚和回滚操作，然而虚拟机太重量级并且不可移植。

2）、新的方式是通过部署基于操作系统级别虚拟化的容器进行虚拟化而非通过硬件来进行虚拟化。这些容器之间相互隔离：它们有自己的文件系统，然而它们也无法看到彼此之间的进程，并且它们之间的计算资源也是有界限的。相较于虚拟机容器也更容易部署，并且因为它们是和底层设施和机器文件系统解耦的，它们可以在云和不同版本的操作系统间进行迁移。

b、因为容器小而快，一个应用可以被打包进一个容器映像。正是应用与容器镜像间一对一的关系解锁了容器的很多优点：

1. 在 build 或者 release 的阶段（而非部署阶段）可以创建不变的容器镜像，因为每个应用都不用和其他的应用栈相组合，也不依赖于生产环境基础设施。这使得从研发到生产过程中可以采用持续的环境。
2. 容器比虚机要更加透明，这更便于监控和管理。尤其是因为窗口的进程的生命周期是被基础设施直接管理而不是被容器中的进程管理器隐藏起来管理。
3. 因为一个容器包含一个应用，这让对容器的管理等同于对应用部署的管理。

c、总结一下容器的优点：

- **敏捷地应用创建和部署：**相较于 VM 增加了容器镜像创建的效率。
- **持续开发，集成和部署：**通过快速的回滚操作（因为镜像的稳定性）提供可靠的经常的容器镜像的创建和部署。

- **开发和运行相分离：**在 build 或者 release 的阶段（而非部署阶段），使得应用和基础设施解耦。
- **开发，测试和生产环境的持续：**在笔记本上可以像在云中一样的运行。
- **云操作系统版本的可移植性：**可以运行在 Ubuntu, RHEL, CoreOS, on-prem, Google Container Engine 和任何其它的运行环境中。
- **应用为中心的管理：**提升了虚拟化的层次，从虚拟硬件上运行操作系统的抽象到操作系统中应用逻辑资源的虚拟。
- **松耦合，分布式，弹性，自由的微服务：**应用被打散成更小的，独立的小碎片并且可以动态地部署和管理——而非是一个在用途单一的庞大机器中运行的一个臃肿堆栈中。
- **资源隔离：**可以预测的应用性能。
- **资源使用：**高效。

（3）为什么需要 Kubernetes，利用它又能做些什么呢？

a、Kubernetes 可以安排物理机或者虚拟机上运行的应用容器的使用。为了充分发挥它的潜能，你需要剪断物理机虚拟机的束缚。然而，一旦特定的容器不再局限于特定的主机，主机为中心的基础设施便不再适用了：组管理，负载均衡，自动扩展等。你需要的是以容器为中心的基础设施。而这正是 Kubernetes 所提供的。

b、Kubernetes 可以满足很多运行环境中应用的通用的需求，比如：

- 进程协同，利用复合应用保证应用和容器一对一的模型。
- 存储系统挂载
- 分发密钥
- 应用健康检测
- 应用实例复制
- 水平自动扩展
- 命名和发现
- 负载均衡
- 滚动更新
- 资源监控
- 日志访问
- 自检和调试
- 识别和认证

这为 PaaS 提供了 IaaS 层的便利，提供了基础设施提供者间的可移植性。

（4）Kubernetes 是怎么样的一个平台呢？

尽管 Kubernetes 提供了很多功能，总有一些新的场景可以从这些功能中获益。特定的应用工作流可以提高开发者的开发速度。最新可以接受的组合常常需要强劲的大规模的自动化。这也是为什么 Kubernetes 构建以来为什么要做一个让应用的部署、扩展和管理更便捷的生态平台的原因。

Labels 让用户可以随心所欲地组织自己的资源。Annotations 让用户可以给资源添加定制化的信息以充分使用自己的工作流，提供一种简单的管理工具。

此外，Kubernetes control plane 本身也是基于公布给开发者和用户相同 的一组 API。用户可以自己定开发自己的 controllers, schedulers 等。如果愿意，它们甚至可以用自己的 API 开发自己的 command-line tool。这样的设计也让很多其它系统可以构建于 Kubernetes 之上。

（5）Kubernetes 不是什么？

1、**Kubernetes 不是一个传统的，包罗一切的 PaaS 系统。我们保留用户的选择**，这一点非常重要。

- Kubernetes 不限制支持应用的种类。它不限制应用框架，或者支持的运行时语言，也不去区分 “apps” 或者 “services”。 Kubernetes 致力于支持不同负载应用，包括有状态、无状态、数据处理类型的应用。只要这个应用可以在容器里运行，那么它就可以在 Kubernetes 上很多地运行。
- Kubernetes 不提供中间件（如 message buses），数据处理框架（如 Spark），数据库(如 Mysql)，或者集群存储系统(如 Ceph)。但这些应用都可以运行于 Kubernetes。
- Kubernetes 没有一个点击即可用的应用市场。
- Kubernetes 不部署源码不编译应用。持续集成的 (CI)工作流方面，不同的用户有不同的需求和偏好，因此，我们提供分层的 CI 工作流，但并不定义它应该怎么做。
- Kubernetes 允许用户选择自己的日志、监控和报警系统。
- Kubernetes 不提供可理解的应用配置语言。
- Kubernetes 不提供或者任何综合的机器配置，维护，管理或者自愈系统。

2、大量的 Paas 系统都可以运行在 Kubernetes 上，比如 Openshift, Deis，和 Gondor。你可以构建自己的 Paas 平台，集成 CI。因为 Kubernetes 运行在应用而非硬件层面，它提供了普通的 Paas 平台提供的一些**通用功能**，比如部署，扩展，负载均衡，日志，监控等。然而，**Kubernetes 并非一个庞然大物，这些功能是可选的**。

3、Kubernetes 不仅仅是一个“编排系统”；它消弥了编排的需要。“编排”的定义是指执行一个预定的工作流：先做 A，之后 B，然后 C。相反地，Kubernetes 是由一系列独立的、可组合的驱使当前状态走向预想状态的控制进程组成的。怎么样从 A 到 C 并不重要：达到目的就好。当然也是需要中心控制的；方法更像排舞的过程。这让这个系统更加好用更加强大、健壮、 有弹性且可扩展。

（6）、*Kubernetes 是什么意思呢*? K8s?

Kubernetes 起源于希腊语，是“舵手”或者“领航员”的意思，是“管理者”和“控制论”的根源。 **K8s** 是把用 8 代替 8 个字符“ubernete”而成的缩写。

(7) 、Kubernetes 中概念的简要概述

- Cluster： 集群是指由 Kubernetes 使用一系列的物理机、虚拟机和其他基础资源来运行你的应用程序。
- Node： 一个 node 就是一个运行着 Kubernetes 的物理机或虚拟机，并且 pod 可以在其上面被调度。.
- Pod： 一个 pod 对应一个由相关容器和卷组成的容器组
- Label： 一个 label 是一个被附加到资源上的键/值对，譬如附加到一个 Pod 上，为它传递一个用户自定的并且可识别的属性.Label 还可以被应用来组织和选择子网中的资源（了解 Label 详情）
- selector 是一个通过匹配 labels 来定义资源之间关系得表达式，例如为一个负载均衡的 service 指定所目标 Pod.（了解 selector 详情）
- Replication Controller : replication controller 是为了保证一定数量被指定的 Pod 的复制品在任何时间都能正常工作.它不仅允许复制的系统易于扩展，还会处理当 pod 在机器在重启或发生故障的时候再次创建一个
- Service： 一个 service 定义了访问 pod 的方式，就像单个固定的 IP 地址和与其相对应的 DNS 名之间的关系。
- Volume: 一个 volume 是一个目录，可能会被容器作为未见系统的一部分来访问。
- Kubernetes volume 构建在 Docker Volumes 之上,并且支持添加和配置 volume 目录或者其他存储设备。
- Secret : Secret 存储了敏感数据，例如能允许容器接收请求的权限令牌。
- Name： 用户为 Kubernetes 中资源定义的名字
- Namespace : Namespace 好比一个资源名字的前缀。它帮助不同的项目、团队或是客户可以共享 cluster,例如防止相互独立的团队间出现命名冲突
- Annotation： 相对于 label 来说可以容纳更大的键值对，它对我们来说可能是不可读的数据，只是为了存储不可识别的辅助数据，尤其是一些被工具或系统扩展用来操作的数据