# 一、前言

容器

PaaS

Cloud Native

gRPC

ServiceMesh

Serverless

…

微服务2.0

POC：Proof of Concept 概念验证

# 二、选型准则

1、生产级

为了解决实际业务问题和上生产抗流量；

生产级 production ready

可运维 Ops Ready

可治理

成熟稳定

…

2、一线互联网公司落地产品

一线公司落地并且开源的，且社区口碑良好的产品

3、开源社区活跃度

Github stars

4、不同的业务体量和团队规模

创业公司：

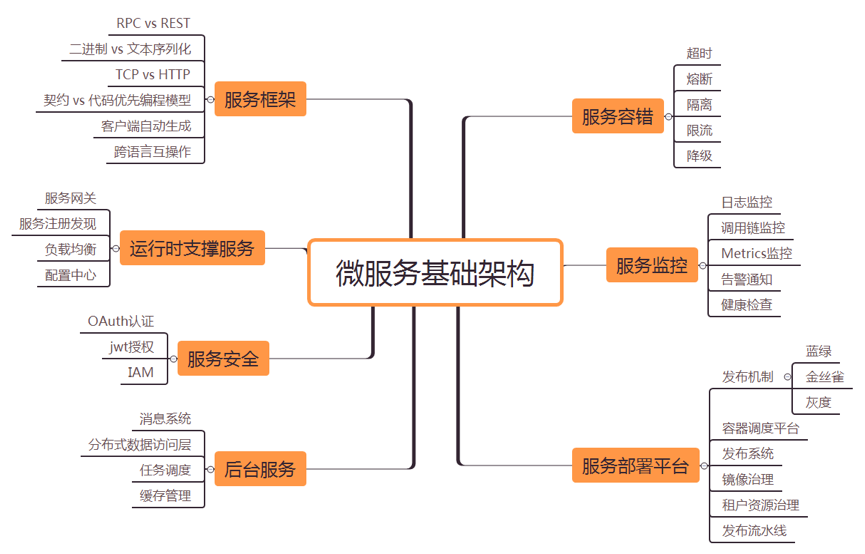
BAT：

…

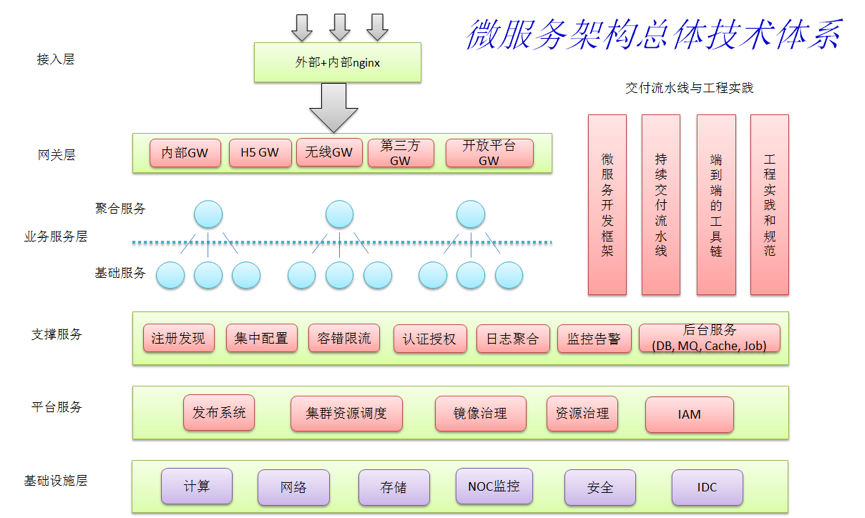
日流量 千万+；研发团队 50+

基础服务架构：自建 还是 购买成熟的产品；

# 三、微服务基础架构关键点



粉红色标注的模块是和微服务关系最密切的模块



# 四、服务框架

## 注解解释

### @EnableDiscoveryClient vs @EnableEurekaClient

@EnableDiscoveryClient

There are multiple implementations of "Discovery Service" (eureka, consul, zookeeper)

lives in spring-cloud-commons and picks the implementation on the classpath

@EnableEurekaClient

lives in spring-cloud-netflix and only works for eureka

### @EnableEurekaServer

将一个普通的Spring Boot应用变成了Eureka的注册中心

spring-cloud-starter-eureka-server依赖于spring-cloud-netflix-eureka-server，而spring-cloud-netflix-eureka-server又依赖于spring-boot-starter-web

### @EnableFeignClients

### @EnableCircuitBreaker @HystrixCommand

Hystrix

### @EnableZuulProxy vs @EnableZuulServer

@EnableZuulServer - 普通Zuul Server,只支持基本的route与filter功能.

@EnableZuulProxy - 普通Zuul Server+服务发现与熔断等功能的增强版,具有反向代理功能.

@EnableCircuitBreaker  
**@Target**(**ElementType**.***TYPE***)  
**@Retention**(**RetentionPolicy**.***RUNTIME***)  
@Import(**ZuulProxyMarkerConfiguration**.**class**)  
**public** @**interface EnableZuulProxy** {  
}

## Spring Boot/Cloud

RESTful

HTTP

JSON

Swagger契约编程模型

### Spring Cloud Config

### Spring Cloud Netflix

OSS（运营支撑系统） components

Eureka，Hystrix，Zuul，Archaius，etc.

### Spring Cloud Bus

### Spring Cloud for Cloud Foundry

Pivotal Cloud Foundry

SSO and OAuth2

### Spring Cloud Open Service Broker

Open Service Broker API

### Spring Cloud Cluster

Leadership election

Zookeeper, redis, hazelcast, consu

### Spring Cloud Consul

服务发现和配置管理

### Spring Cloud Security

均衡OAuth2 rest客户端和认证头，依赖Zuul proxy

### Spring Cloud Sleuth

基于Zipkin的日志追踪，ELK

### Sprig Cloud Data Flow

轻量级事件驱动微服务框架，集成Kafka、RabbitMQ

### Spring Cloud Stream App Starters

集成扩展系统

### Spring Cloud Task

### Spring Cloud Task App Starters

### Spring Cloud Zookeeper

服务发现和配置管理 zk

### Spring Cloud for Amazon Web Services

### Spring Cloud Connectors

PaaS

### Spring Cloud Starters

### Spring Cloud CLI

### Spring Cloud Contract

实现消费者驱动的契约方法

### Spring Cloud Gateway

### Spring Cloud OpenFeign

### Spring Cloud Piplines

部署相关

### Spring Cloud Function

## 2、Dubbo

服务治理能力丰富，框架重

RPC

Java：跨语言能力不足

### Dubbox

Dubbox（dangdang）

### Motan

Motan（Sina）

## 3、gRPC

RPC

Protobuf

基于 protobuf 的强契约编程模型，能自动生成各种语言客户端，且保证互操作

支持Http2

更适合内部服务相互调用场景，对我保留RESTful接口可以实现

新

# 五、运行时支持服务

## 1、服务注册中心

### Eureka

Spring + Eureka

### Consul

支持跨数据中心，还支持 KV 模型存储和灵活健康检查能力

## 2、服务路由网关

### Zuul

Spring Cloud

支持灵活的动态过滤器脚本机制，异步性能不足（基于 Netty 的异步 Zuul 迟迟未能推出正式版）

### Kong

基于 Nginx/OpenResty 的 API 网关

因为采用 Nginx 内核，Kong 的异步性能较强，另外基于 lua 的插件机制比较灵活，社区插件也比较丰富，从安全到限流熔断都有，还有不少开源的管理界面，能够集中管理 Kong 集群

## 3、集中式配置中心

### Spring Cloud Config

个人认为算不上生产级，很多治理能力缺失，小规模场景可以试用

### Apollo（携程）

携程经过生产级验证，具备高可用，配置实时生效（推拉结合），配置审计和版本化，多环境多集群支持等生产级特性，建议中大规模需要对配置集中进行治理的企业采用

# 六、服务监控

## 1、日志监控

### ELK

Elasticsearch

Logstash

Kibana

Elastalert：是 Yelp 开源的针对 ELK 的告警通知模块

## 2、调用链监控



### CAT（大众点评）

在点评和国内多家互联网公司有落地案例，生产级特性和治理能力较完善，另外 CAT 自带告警模块

### Zipkin

OpenZipkin

### Pinpoint

## 3、Metrics监控

Metrics 监控主要依赖于时间序列数据库 (TSDB)

### Grafana

是 Metrics 报表展示的社区标配

### OpenTSDB

基于 HBase

具有分布式能力可以横向扩展，但是相对较重，适用于中大规模企业

本身不提供告警模块

### KariosDB

基于 Cassandra

它基本上是 OpenTSDB 针对 Cassandra 的一个改造版

### InfluxDB

函数报表能力丰富，自带告警模块，分布式能力不足，适用于中小规模企业

### Prometheus

函数报表能力丰富，自带告警模块，分布式能力不足，适用于中小规模企业

## 4、健康检查

### Sensu

能够对各种服务（例如 Spring Boot 暴露的健康检查端点，时间序列数据库中的 metrics，ELK 中的错误日志等）定制灵活的健康检查 (check)，然后用户可以针对 check 结果设置灵活的告警通知策略。

在 Yelp 等公司有落地案例

### Esty

定制 check 和告警配置的使用门槛比较高，社区不热

### ZMon

定制 check 和告警配置的使用门槛比较高，社区不热

ZMon 后台采用 KairosDB 存储，如果企业已经采用 KariosDB 作为时间序列数据库，则可以考虑 ZMon 作为告警通知模块

## 5、告警通知

### Argus

基于 OpenTSDB 的统一监控告警平台，支持丰富的告警函数和灵活的告警配置，可以作为 OpenTSDB 的告警补充

# 七、服务容错

## 1、Hystrix

把熔断、隔离、限流和降级等能力封装成组件，任何依赖调用（数据库，服务，缓存）都可以封装在 Hystrix Command 之内，封装后自动具备容错能力

Hystrix 一般需要在应用端或者框架内埋点，有一定的使用门槛。

对于采用集中式反向代理（边界和内部）做服务路由的公司，则可以集中在反向代理上做熔断限流，例如采用 Nginx或者 Kong这类反向代理，它们都插件支持灵活的限流容错配置。Zuul 网关也可以集成 Hystrix 实现网关层集中式限流容错。集中式反向代理需要有一定的研发和运维能力，但是可以对限流容错进行集中治理，可以简化客户端

# 八、后台服务选型

## 1、消息系统

### Kafka

对于日志等可靠性要求不高的场景

对于可靠性要求较高的业务场景，Kafka 其实也是可以胜任，但企业需要根据具体场景，对 Kafka 的监控和治理能力进行适当定制完善

### Hermes

在 Kafka 基础上封装了适合业务场景的企业级治理能力

### RocketMQ

具备更多适用于业务场景的特性

### RabbitMQ

队列特性和文档都很丰富，性能和分布式能力稍弱，中小规模场景可选

## 2、分布式缓存

### Cachecloud

搜狐

如果倾向于采用客户端直连模式（个人认为缓存直连更简单轻量）

是一款不错的 Redis 缓存治理平台，提供诸如监控统计，一键开启，自动故障转移，在线伸缩，自动化运维等生产级治理能力，另外其文档也比较丰富

### Twemproxy

如果倾向采用中间层 Proxy 模式

### Codis

如果倾向采用中间层 Proxy 模式

## 3、分布式数据库访问

<https://github.com/sharding-sphere/sharding-sphere.git>

### Shardingjdbc（当当）

分库分表逻辑做在客户端 jdbc driver 中，客户端直连数据库比较简单轻量，建议中小规模场景采用

### MyCAT

如果倾向采用数据库访问中间层 proxy 模式

Cobar 演化

proxy 模式运维成本较高，建议中大规模场景，有一定框架自研和运维能力的团队采用

## 4、任务调度系统

### xxl-job

部署简单轻量，大部分场景够用

### elastic-job

当当开源

## 5、分布式检索

### Solr

### Solandra

### SolrCloud

### Solr+Katta

### Elasticsearch（首选）

# 九、服务安全

对于微服务安全认证授权机制一块，目前业界虽然有 OAuth 和 OpenID connect 等标准协议，但是各家具体实现的做法都不太一样，企业一般有很多特殊的定制需求，整个社区还没有形成通用生产级开箱即用的产品。

个人建议基于 OAuth 和 OpenID connect 标准，在参考一些开源产品的基础上（例如 Mitre 开源的 OpenID-Connect-Java-Spring-Server，定制自研轻量级授权服务器。

<https://github.com/mitreid-connect/OpenID-Connect-Java-Spring-Server.git>

## 1、CAS

Opinionated（一家观点和做法）

支持太多协议造成产品复杂，也缺乏足够灵活性

<https://github.com/apereo/cas.git>

## 2、keycloak

Opinionated（一家观点和做法）

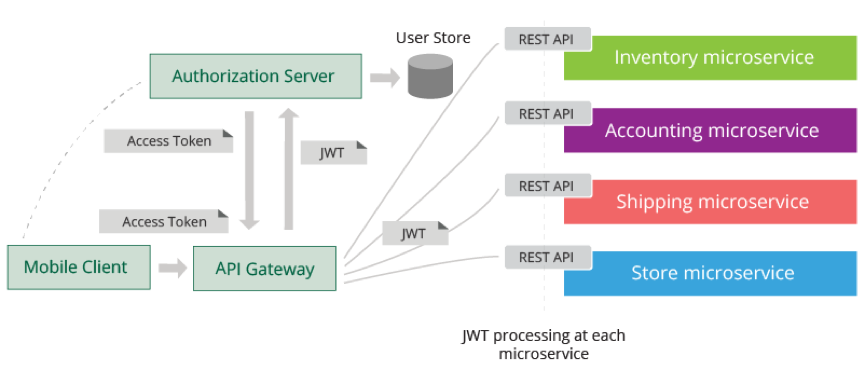
支持太多协议造成产品复杂，也缺乏足够灵活性

## 3、spring cloud security

Opinionated（一家观点和做法）

支持太多协议造成产品复杂，也缺乏足够灵活性

## 4、Wso2



使用支持 OAuth 2.0 和 OpenID Connect 标准协议的授权服务器（个人建议定制自研）；

使用 API 网关作为单一访问入口，统一实现安全治理；

客户在访问微服务之前，先通过授权服务器登录获取 access token，然后将 access token 和请求一起发送到网关；

网关获取 access token，通过授权服务器校验 token，同时做 token 转换获取 JWT token。

网关将 JWT Token 和请求一起转发到后台微服务；

JWT 中可以存储用户会话信息，该信息可以传递给后台的微服务，也可以在微服务之间传递，用作认证授权等用途；

每个微服务包含 JWT 客户端，能够解密 JWT 并获取其中的用户会话信息。

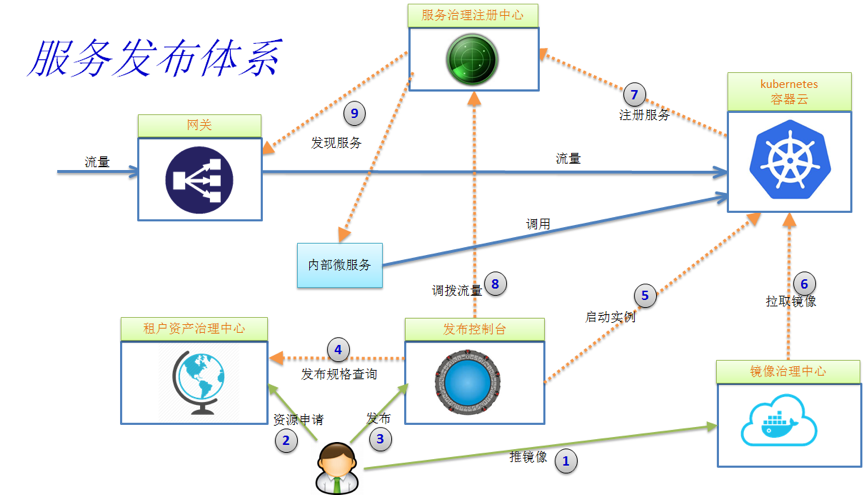
整个方案中，access token 是一种 by reference token，不包含用户信息可以直接暴露在公网上；JWT token 是一种 by value token，可以包含用户信息但不暴露在公网上。

## 5、Mitre

# 十、服务部署平台

容器已经被社区接受为交付微服务的一种理想手段，可以实现不可变（immutable）发布模式。

一个轻量级的基于容器的服务部署平台主要包括容器资源调度，发布系统，镜像治理，资源治理和 IAM 等模块。



简化发布流程如下：

1. 应用通过 CI 集成后生成镜像，用户将镜像推到镜像治理中心；
2. 用户在资产治理中心申请发布，填报应用，发布和配额相关信息，然后等待审批通过；
3. 发布审批通过，开发人员通过发布控制台发布应用；
4. 发布系统通过查询资产治理中心获取发布规格信息；
5. 发布系统向容器云发出启动容器实例指令；
6. 容器云从镜像治理中心拉取镜像并启动容器；
7. 容器内服务启动后自注册到服务注册中心，并保持定期心跳；
8. 用户通过发布系统调用服务注册中心调拨流量，实现蓝绿，金丝雀或灰度发布等机制；
9. 网关和内部微服务客户端定期同步服务注册中心上的服务路由表，将流量按负载均衡策略分发到新的服务实例上。

## 集群资源调度系统

屏蔽容器细节，将整个集群抽象成容器资源池，支持按需申请和释放容器资源，物理机发生故障时能够实现自动故障迁移 (fail over)

### K8s

首选

#### 名称解释

##### Pod

相当于一个共享context的配置组，在同一个context下，应用可能还会有独立的cgroup隔离机制，一个Pod是一个容器环境下的“逻辑主机”，它可能包含一个或者多个紧密相连的应用，这些应用可能是在同一个物理主机或虚拟机上。

Pod 的context可以理解成多个linux命名空间的联合

* PID 命名空间（同一个Pod中应用可以看到其它进程）
* 网络 命名空间（同一个Pod的中的应用对相同的IP地址和端口有权限）
* IPC 命名空间（同一个Pod中的应用可以通过VPC或者POSIX进行通信）
* UTS 命名空间（同一个Pod中的应用共享一个主机名称）

同一个Pod中的应用可以共享磁盘，磁盘是Pod级的

Pod是一种相对短暂的存在，而不是持久存在的

Pod被安排到结点上，并且保持在这个节点上直到被终止（根据重启的设定）或者被删除，当一个节点死掉之后，上面的所有Pod均会被删除。特殊的Pod永远不会被转移到的其他的节点，作为替代，他们必须被replace

Pod使Pod内的数据共享及通信变得容易

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。

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。

Pod 容器 Volume 镜像

##### Labels

标签 key/value

例子：

release” : “stable”, “release” : “canary”, …

“environment” : “dev”, “environment” : “qa”, “environment” : “production”

“tier” : “frontend”, “tier” : “backend”, “tier” : “middleware”

“partition” : “customerA”, “partition” : “customerB”, …

“track” : “daily”, “track” : “weekly”

与name和UID不同，label不提供唯一性

##### Namespace

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

Namespace常用来隔离不同的用户，比如Kubernetes自带的服务一般运行在kube-system namespace中

apiVersion: v1

kind: Namespace

metadata:

name: development

##### Replication Controller

保证了在所有时间内，都有特定数量的Pod副本正在运行，如果太多了，Replication Controller就杀死几个，如果太少了，Replication Controller会新建几个，和直接创建的pod不同的是，Replication Controller会替换掉那些删除的或者被终止的pod，不管删除的原因是什么。

基于这个理由，我们建议即使是只创建一个pod，我们也要使用Replication Controller。Replication Controller 就像一个进程管理器，监管着不同node上的多个pod,而不是单单监控一个node上的pod,Replication Controller 会委派本地容器来启动一些节点上服务（Kubelet ,Docker）。

Replication Controller只会对那些RestartPolicy = Always的Pod的生效，（RestartPolicy的默认值就是Always）。

Replication Controller永远不会自己关闭，但是，我们并不希望Replication Controller成为一个长久存在的服务。

###### 工作原理

Pod template

模板创建Pod

Lables

RC 监控的Pod数量是由 label selector 决定的；

###### 常见使用模式

Rescheduling

重新规划，确保指定数量的pod运行

Scaling

缩放，修改relicas的值

Rolling updates

动态更新，允许一个一个的替换pod

##### Node

是pod真正运行的主机，可以是物理机也可以是虚拟机；

每个node节点上至少要运行container runtime （docker 或 rkt）、kubelet和kube-proxy服务

K8s只是管理node上的资源；

{

"kind": "Node",

"apiVersion": "v1",

"metadata": {

"name": "10.240.79.157",

"labels": {

"name": "my-first-k8s-node"

}

}

}

Node Controller 负责：

* 维护Node状态
* 与Cloud Provider同步Node
* 给Node分配容器CIDR
* 删除带有NoExecute taint的Node上的Pods

默认情况下，kubelet在启动时会向master注册自己，并创建Node资源；

Node 的状态

* 地址：包括hostname、外网IP和内网IP
* 条件（Condition）：包括OutOfDisk、Ready、MemoryPressure和DiskPressure
* 容量（Capacity）：Node上的可用资源，包括CPU、内存和Pod总数
* 基本信息（Info）：包括内核版本、容器引擎版本、OS类型等

Taints & Tolerations

用于保证Pod不被调度到不合适的Node上，Taint应用于Node上，而toleration则应用于Pod上（Toleration是可选的

kubectl taint nodes node1 key1=value1:NoSchedule

kubectl taint nodes node1 key1=value2:NoExecute

##### ReplicaSets

下一代复本控制器

ReplicaSet和 [Replication Controller](https://www.kubernetes.org.cn/replication-controller-kubernetes)之间的唯一区别是现在的选择器支持;

Replication Controller只支持基于等式的selector（env=dev或environment!=qa），但ReplicaSet还支持新的，基于集合的selector（version in (v1.0, v2.0)或env notin (dev, qa)）。在试用时官方推荐ReplicaSet。

大多数[kubectl](https://www.kubernetes.org.cn/installkubectl)支持Replication Controller的命令也支持ReplicaSets。[rolling-update](https://kubernetes.io/docs/user-guide/kubectl/v1.7/" \l "rolling-update" \t "_blank)命令有一个例外 。如果您想要滚动更新功能，请考虑使用Deployments。此外， rolling-update命令是必须的，而Deployments是声明式的，因此我们建议通过rollout命令使用Deployments。

何时使用ReplicaSet

ReplicaSet可确保指定数量的pod“replicas”在任何设定的时间运行。然而，Deployments是一个更高层次的概念，它管理ReplicaSets，并提供对pod的声明性更新以及许多其他的功能。因此，我们建议您使用Deployments而不是直接使用ReplicaSets，除非您需要自定义更新编排或根本不需要更新。

##### Services

Pod会被创建，杀死，不可复活；

Service：定义了一组Pod的策略的抽象，我们有时候也叫做宏观服务；

被服务标记的Pod都是（一般）通过label Selector决定的

##### Volumes

卷

emptyDir: 和pod同生命周期

HostPath: 文件或目录

GcePersistentDisk: 解除挂载或添加挂载，不删除

AwsElasticBlockStore:

Nfs: 解除挂载，不删除，可以同时被多个pod挂载；

iscsi: 挂载 不删除

glusterfs

rbd

gitRepo

Secrets

##### PV/PVC/StorageClass

PV(Persistent Volume) 集群中已由管理员配置的一段网络存储。

PVC(Persistent Volume Claim) 用户存储的请求

##### Deployment

Deployment为Pod和ReplicaSet提供了一个声明式定义(declarative)方法，用来替代以前的[ReplicationController](https://www.kubernetes.org.cn/replication-controller-kubernetes)来方便的管理应用。

##### Secret

Secret解决了密码、token、密钥等敏感数据的配置问题，而不需要把这些敏感数据暴露到镜像或者Pod Spec中。

apiVersion: v1

kind: Secret

metadata:

name: mysecret

type: Opaque

data:

password: MWYyZDFlMmU2N2Rm

username: YWRtaW4=

##### StatefulSet

为了解决有状态服务的问题（对应Deployments和ReplicaSets是为无状态服务而设计）

应用场景:

##### DaemonSet

DaemonSet保证在每个Node上都运行一个容器副本，常用来部署一些集群的日志、监控或者其他系统管理应用。

* 日志收集，比如fluentd，logstash等
* 系统监控，比如Prometheus Node Exporter，collectd，New Relic agent，Ganglia gmond等
* 系统程序，比如kube-proxy, kube-dns, glusterd, ceph等

##### Service Account

User Account

##### CronJob

定时任务

##### Job

批量处理短暂一次性任务（short lived one-off tasks）

apiVersion: batch/v1

kind: Job

metadata:

name: pi

spec:

template:

metadata:

name: pi

spec:

containers:

- name: pi

image: perl

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

##### Security Context & PSP

Security Context: 限制不可信容器的行为，保护系统和其他容器不受其影响

* Container-level Security Context：仅应用到指定的容器
* Pod-level Security Context：应用到Pod内所有容器以及Volume
* Pod Security Policies（PSP）：应用到集群内部所有Pod以及Volume

PSP (Pod Security Policies)



SELinux (security-enhanced linux)

一种强制访问控制（mandatory access control）的实现;

以最小权限原则（principle of least privilege）为基础，在Linux核心中使用Linux安全模块（Linux Security Modules）

开启与关闭SELinux

修改/etc/selinux/config文件方法：

开启：SELINUX=enforcing

关闭：SELINUX=disabled

通过命令临时修改：

开启：setenforce 1

关闭：setenforce 0

##### Resource Quotas

'kwotə 配额

限制用户资源用量的一种机制

工作原理为

* 资源配额应用在Namespace上，并且每个Namespace最多只能有一个ResourceQuota对象
* 开启计算资源配额后，创建容器时必须配置计算资源请求或限制（也可以用[LimitRange](https://kubernetes.io/docs/tasks/administer-cluster/cpu-memory-limit/)设置默认值）
* 用户超额后禁止创建新的资源
* apiVersion: v1
* kind: **ResourceQuota**
* metadata:
* name: compute-resources
* spec:
* hard:
* pods: "4"
* requests.cpu: "1"
* requests.memory: 1Gi
* limits.cpu: "2"
* limits.memory: 2Gi

LimitRange

apiVersion: v1

kind: **LimitRange**

metadata:

name: mylimits

spec:

limits:

- max:

cpu: "2"

memory: 1Gi

min:

cpu: 200m

memory: 6Mi

type: Pod

- default:

cpu: 300m

memory: 200Mi

defaultRequest:

cpu: 200m

memory: 100Mi

max:

cpu: "2"

memory: 1Gi

min:

cpu: 100m

memory: 3Mi

type: Container

##### Network Policy

基于策略的网络控制；

使用标签选择器模拟传统的分段网络，并通过策略控制他们之间的流量以及来自外部的流量；

* + apiserver开启extensions/v1beta1/networkpolicies
  + 网络插件要支持Network Policy，如Calico、Romana、Weave Net和trireme等

策略：

Namespace隔离

Pod隔离

apiVersion: extensions/v1beta1

kind: NetworkPolicy

metadata:

name: test-network-policy

namespace: default

spec:

podSelector:

matchLabels:

role: db

ingress:

- from:

- namespaceSelector:

matchLabels:

project: myproject

- podSelector:

matchLabels:

role: frontend

ports:

- protocol: tcp

port: 6379

##### Ingress

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。

为进入集群的请求提供路由规则的集合。

Ingress可以给service提供集群外部访问的URL、负载均衡、SSL终止、HTTP路由等。为了配置这些Ingress规则，集群管理员需要部署一个[Ingress controller](https://kubernetes.io/docs/concepts/services-networking/ingress/#ingress-controllers)，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。

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: test-ingress

spec:

rules:

- http:

paths:

- path: /testpath

backend:

serviceName: test

servicePort: 80

单服务Ingress即该Ingress仅指定一个没有任何规则的后端服务。

#### 命令

##### Kubectl

### Mesos

### Swarm

## 发布平台

面向用户的发布管理控制台，支持发布流程编排。它和其它子系统对接交互，实现基本的应用发布能力，也实现如蓝绿，金丝雀和灰度等高级发布机制。

目前这块生产级的开源产品很少。

Netflix 开源的 spinnake是一个，但是这个产品比较复杂重量（因为它既要支持适配对接各种 CI 系统，同时还要适配对接各种公有云和容器云，使得整个系统异常复杂），一般企业建议根据自己的场景定制自研轻量级的解决方案。

## 3、镜像治理

基于 Docker Registry，封装一些轻量级的治理功能

### Harbor

VMware 开源

在 Docker Registry 基础上扩展了权限控制，审计，镜像同步，管理界面等治理能力，可以考虑采用

## 4、资源治理

类似于 CMDB 思路，在容器云环境中，企业仍然需要对应用 app，组织 org，容器配额和数量等相关信息进行轻量级的治理。目前这块还没有生产级的开源产品，一般企业需要根据自己的场景定制自研。

## 5、IAM

是 identity & access management 的简称，对发布平台各个组件进行身份认证和安全访问控制。

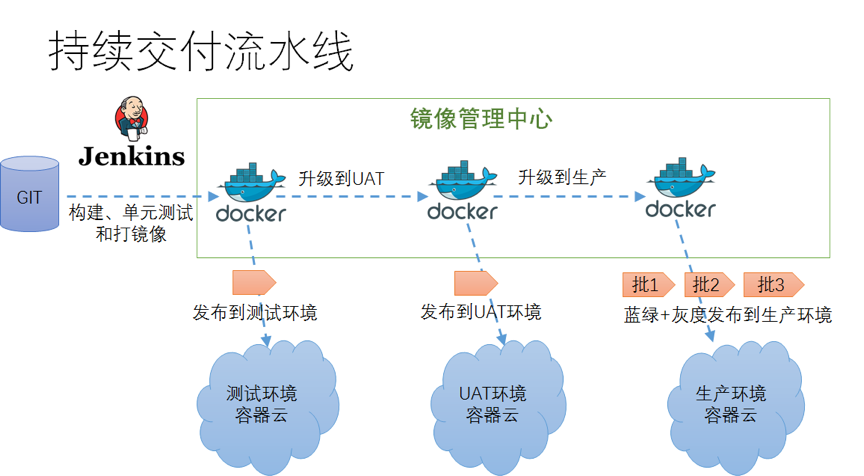
CAS

Keycloak

# 十一、持续交付流水线（CD Pipeline）

在镜像治理中心上封装一些轻量级的治理流程，例如只有通过测试环境测试的镜像才能升级发布到 UAT 环境，只有通过 UAT 环境测试的镜像才能升级发布到生产环境，通过在流水线上设置一些质量门，保障应用高质量交付到生产。

CI（Continuous integration持续集成）



## 1、Maven/Gradle/Ant

## 2、Git/SVN

## 3、Jenkins

## 4、Docker

## 5、K8s

## 6、Linux

bash

## 7、代码质量管理

单一职责：

功能单一

如果不得不加 and，考虑拆分

方法超过100行，一定要进行拆分

注：这里的100可能有点多，只是对我个人而言，100算是我的极限，总之就是绝对不要将一个函数写的太长。

查询方法不修改，幂等性（没有副作用）

易用性：

低耦合，高内聚

可读性

命名

复杂性守恒原则：无论你怎么写代码，复杂性都是不会消失的

注：如果逻辑很复杂，那么代码看起来就应该是复杂的。如果逻辑很简单，代码看起来就应该是简单的。

### SonarQube

### Findbugs

# 十二、测试

UI测试（界面测试）

## 1、单元测试

Junit

Mockito

JMock

Hamcrest

## 2、CDCT（消费者驱动契约测试）

消费者定义他们期望的API或消息是什么样子，这些期望即为契约，从这些契约可以生成存根，此后消费者团队可以在构建过程中重复使用它们。消费者和生产者都需要验证契约。

CDCT强调契约由消费者来驱动，并由双方**共同遵守**，核心是**共同遵守**。

### Spring Cloud Contract

使用Groovy DSL定义测试契约并生成测试套件，测试套件去验证服务提供方是否满足契约，测试通过之后会生成一个jar文件，该jar文件随后会作为一个可运行的Stub server，消费方基于Stub server编写测试，从而验证功能是否满足契约：

### Pact

支持多语言

# 附录

<https://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa/>