DevopS

最佳实践

Seraphim-李玉峰

目录

CONTENTS

1 问题

2 处理

3 革新

4 实践

问题

让我们头疼的

PART ONE

- 我们眼中的产品
- 同事眼中的我们
- 我们眼中的研发

我们眼中的研发、产品

计划？

研发产品没规划，说上线就上线，暴风雨来的太突然。

重要？

各业务线老大都来催，谁最重要？

劳工

系统各种重置，今天去换硬盘、明天加内存 ……

同事眼中的我们

研发觉得你

1、他们又不让我们碰线上的系统，生产环境是什么样，我们都不知道，没法开发代码。

2、我们辛苦开发几个月，上线出问题又直接回滚了，心情很不好受。

3、代码在测试环境或我的机器跑的好好的呀，怎么一上线就出问题呢。

4、测试怎么测的，那么多问题发现不了。

5、我们希望运维同事帮忙搭一个跟线上一模一样的测试环境。

同事眼中的我们

业务觉得你

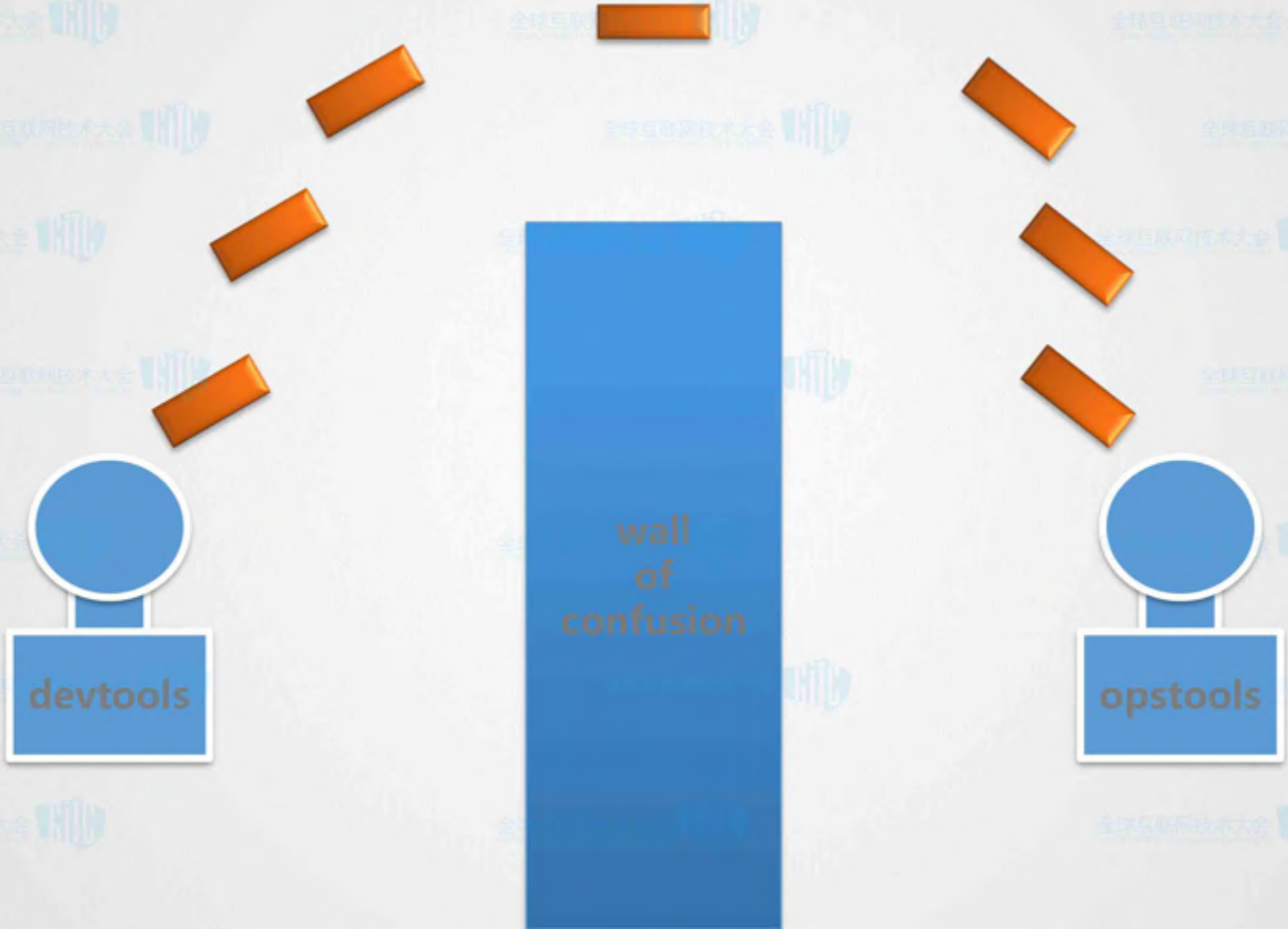
1、今天不能上线，明天不能上线，今天没服务器，明天没资源，让我等到哪天？

2、无结果的等待，没有专人负责。

3、好不容易项目能上线了，因为各种借口，各种等。

4、老出问题，今天光纤挖到了，明天机房遭到攻击了，后天运营商骨干有问题。


wall
of
confusion
develop
ment
op


wall
of
confusion
devtools
opstools

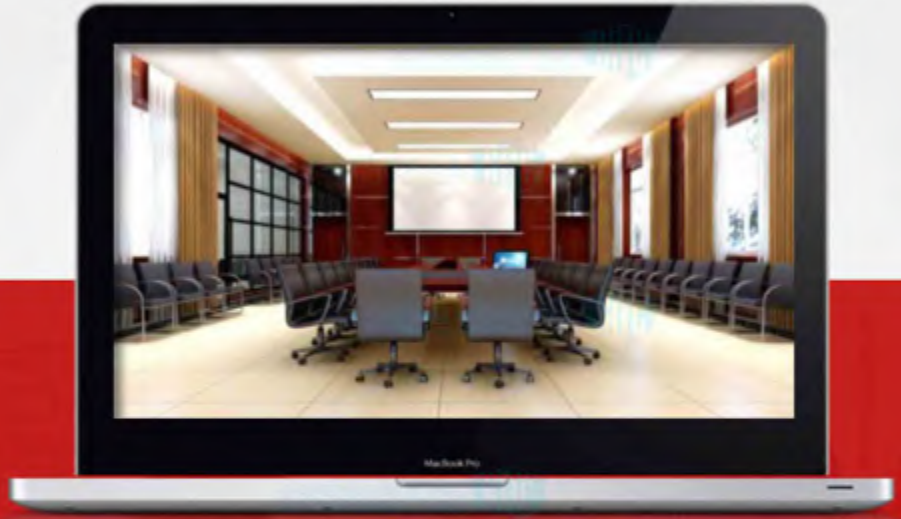
PART TWO

处理

常规处理方式

- 权限控制
- 信息对等
- 多环境

信息对等



运维同学

提供环境、架构、硬件等相关建议

业务同学

沟通需求、时间节点、进度

运维同学参加技术评审会，一起沟通时间进度。在代码中增加数据收集的接口和监控接口，收集产品的性能数据，方便地对运行状态进行监控与报警。
部分开发人员和运维同事协同工作，一起来互补，互相沟通，确保代码的适用性。

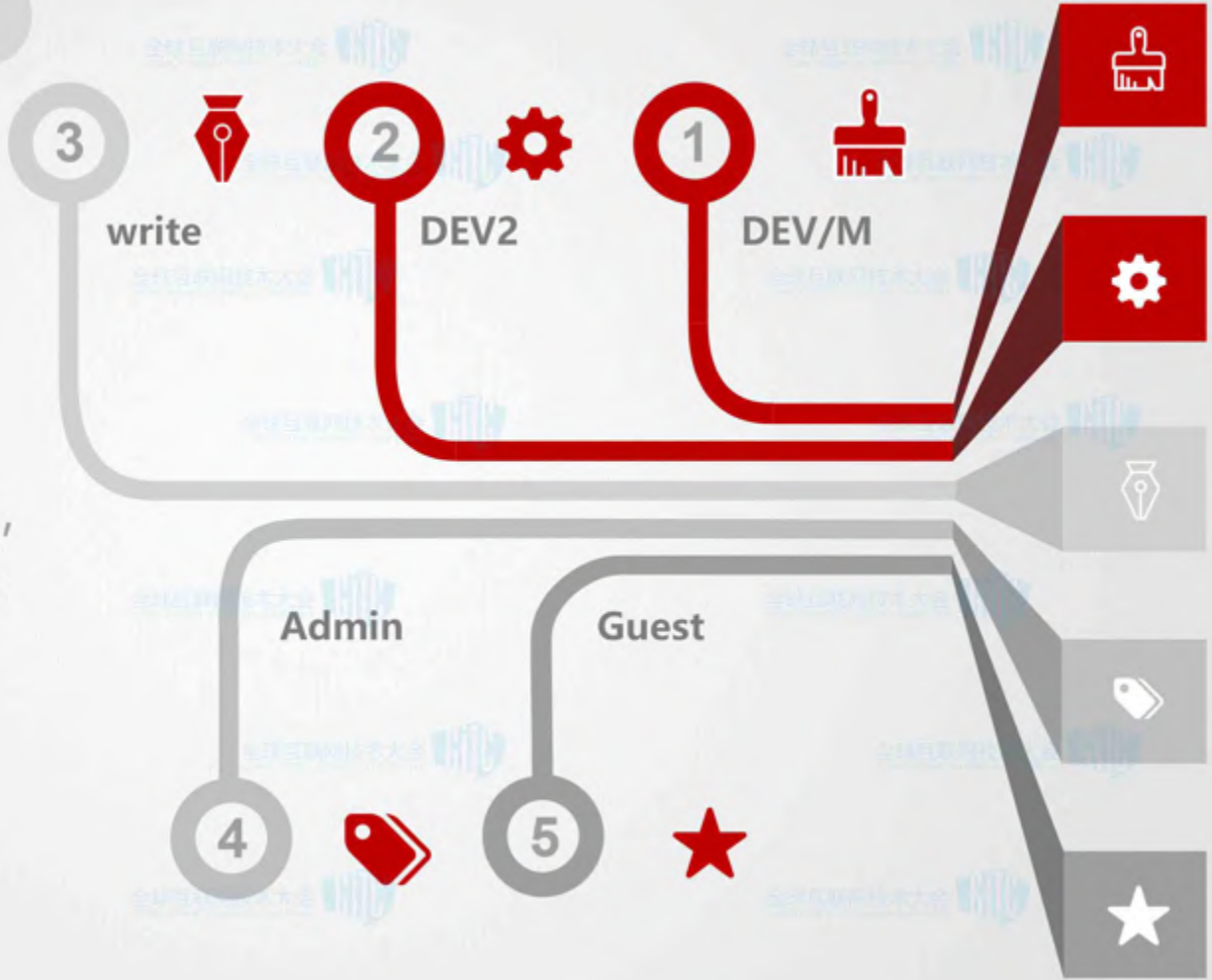
多环境

1. 测试环境为所有版本功能最终测试，测试环境最好依据各业务线分开进行。
2. 灰度环境是类生产环境，从数据到程序本身，相识度99%，是大版本生产迭代的最后一程。
3. 生产环境，最终软件应用承载环境，为用户提供企业所有主营服务，是所有用户的入口，也是企业对外提供应用的主要方式。

生产环境中增加灰度、沙箱，避免更多的问题影响更多的用户。

权限控制

适当分配账号给业务开发，
让他们有一定的只读权限。



PART THREE

革新

我们怎么革新

- 回顾
- 目标
- 价值
- 最佳实践

回顾

权限控制

适当分配账号给业务开发，让他们拥一定的只读权限。

联合作业

部分开发人员和运维同事协同工作，一起来互补，互相沟通，确保代码的适用性。

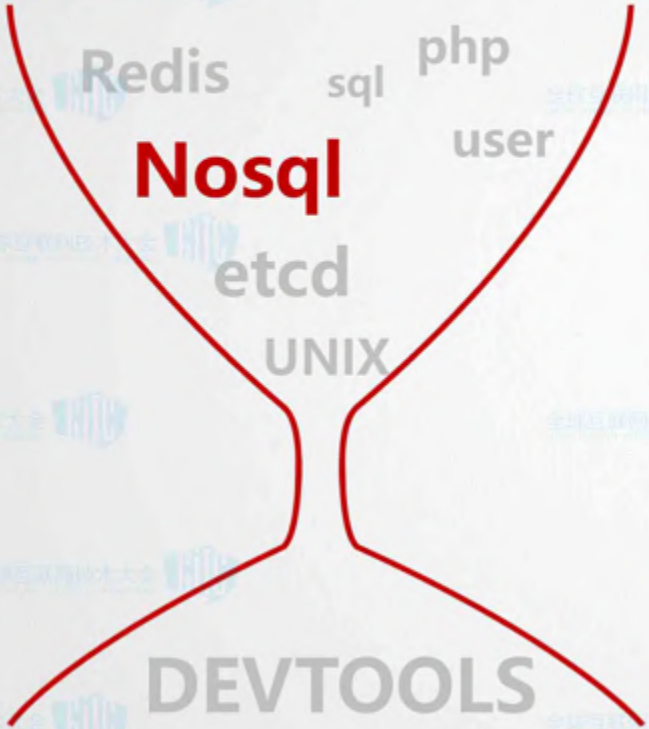
多环境

生产环境中增加灰度、沙箱，避免更多的问题影响更多的用户

信息对等

运维同学参加技术评审会，一起沟通时间进度。

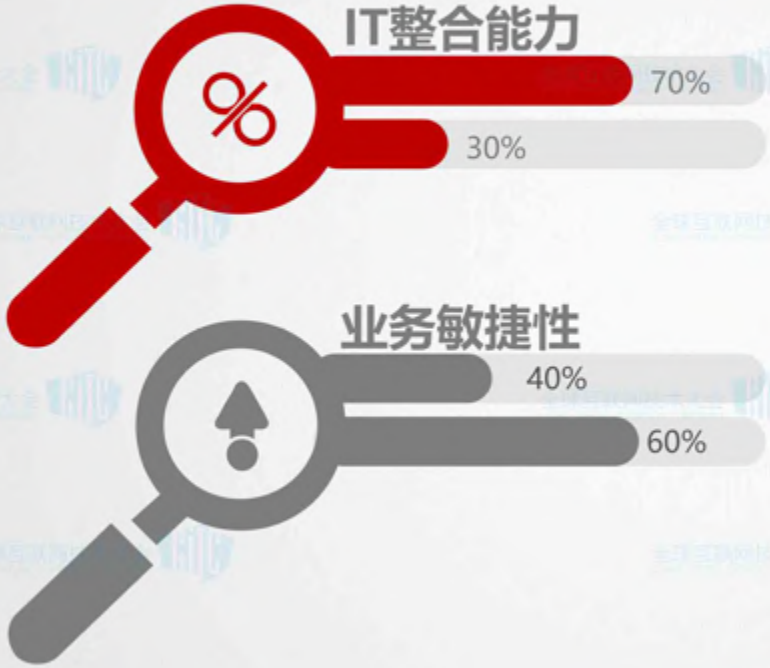
目标



DevOps

不管你是开发者、测试人员、管理者、DBA、网络工程师还是系统管理员，大家都是一起的，只有一起努力提供稳定而高质量的软件服务，实现公司的商业利益。

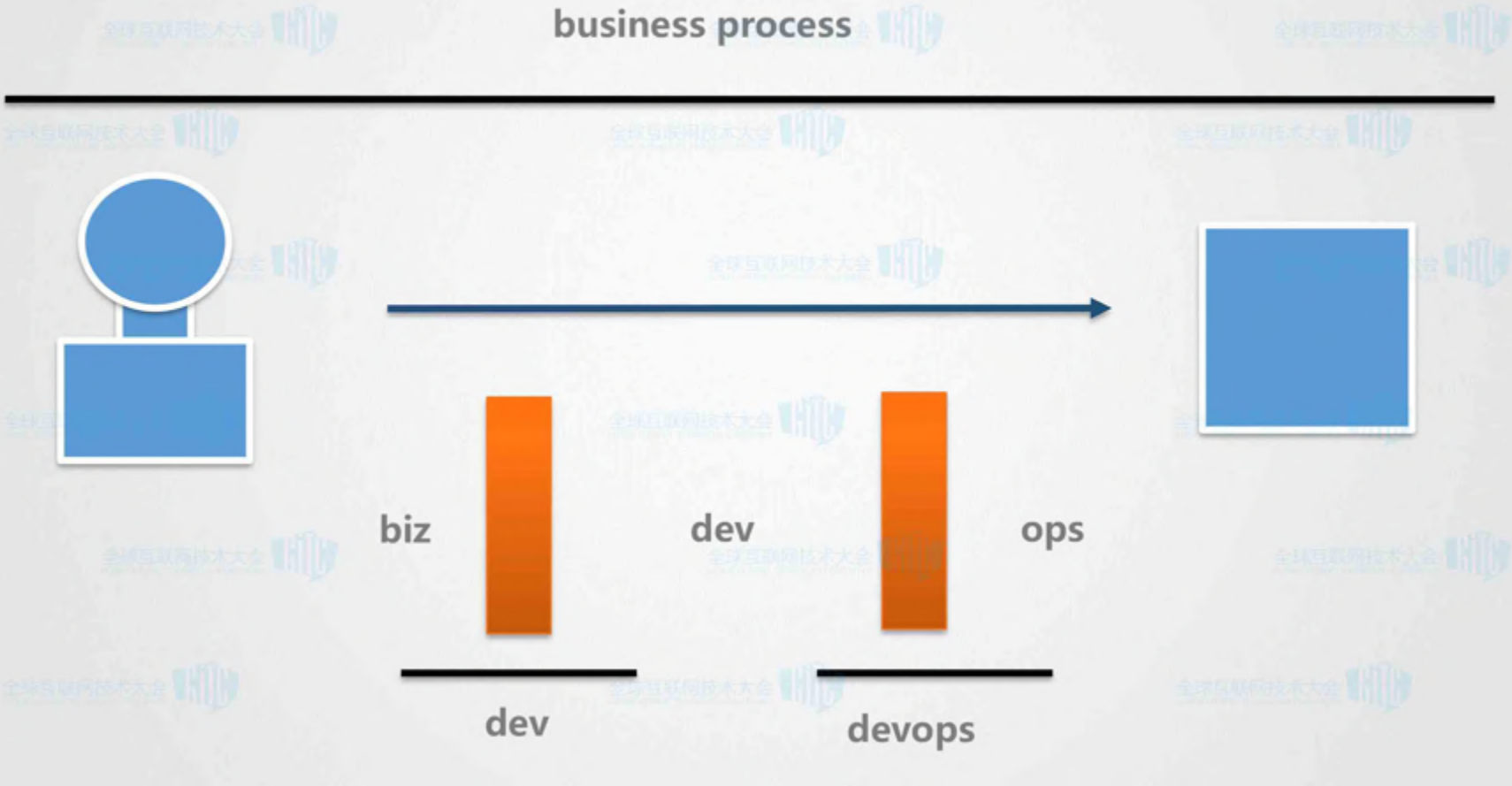
价值



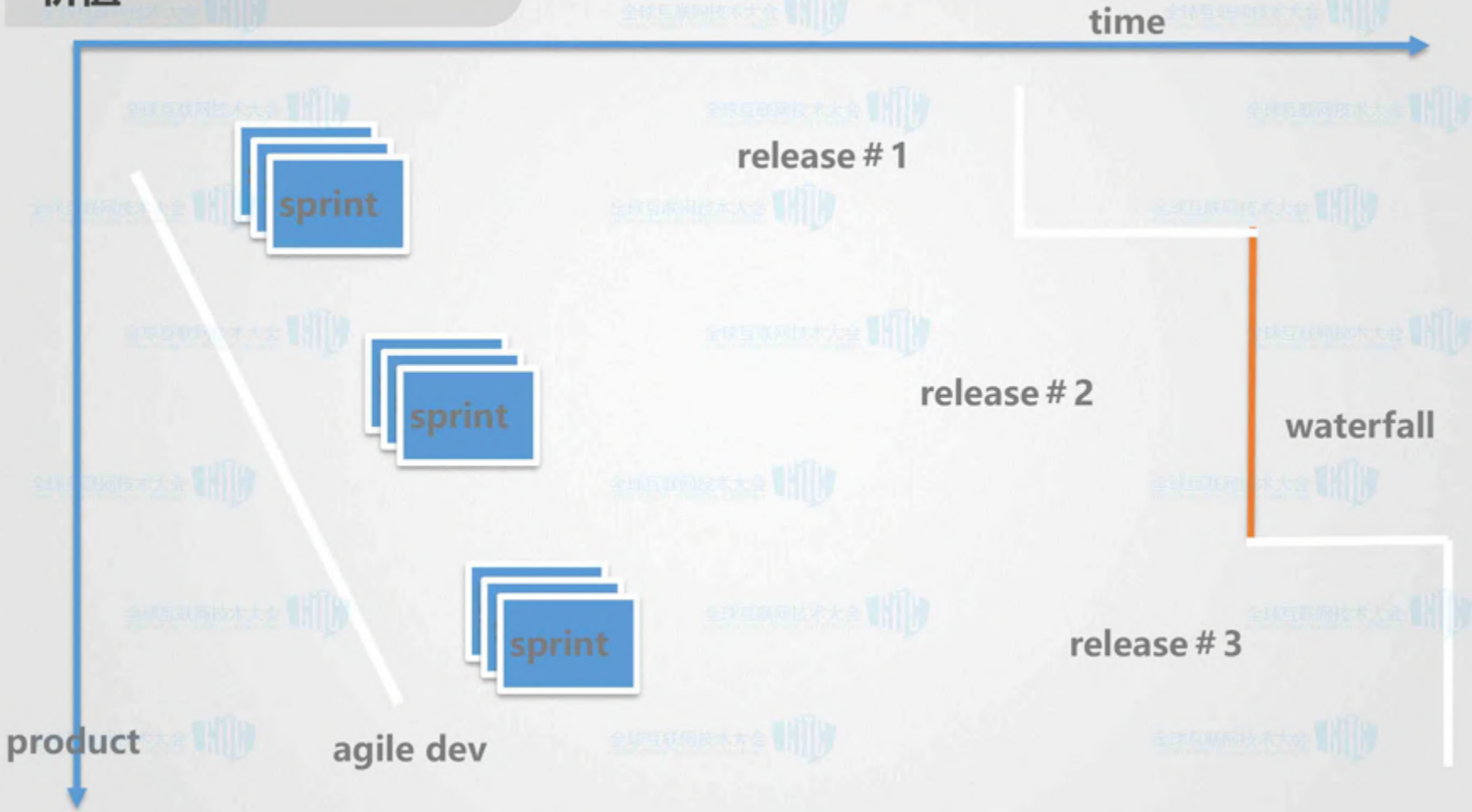
强力整合工作量，从研发、测试、产品、到运维，实现自助化研发、版本迭代，增加容错率，减少人工失误。

突破部门衔接问题，带来的瀑布效应，紧密结合，实现真正的敏捷性。

价值

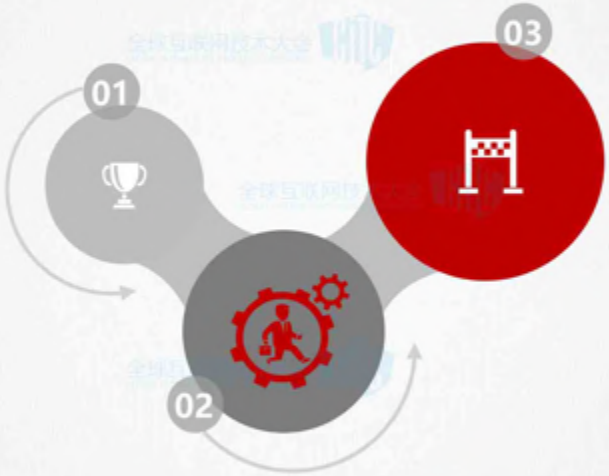


价值


time
release # 1
sprint
sprint
release # 2
waterfall
sprint
release # 3
product
agile dev

最佳实践

Measurement and incentives to change culture



Unified processes

Unified tooling

实践

举个“栗子”

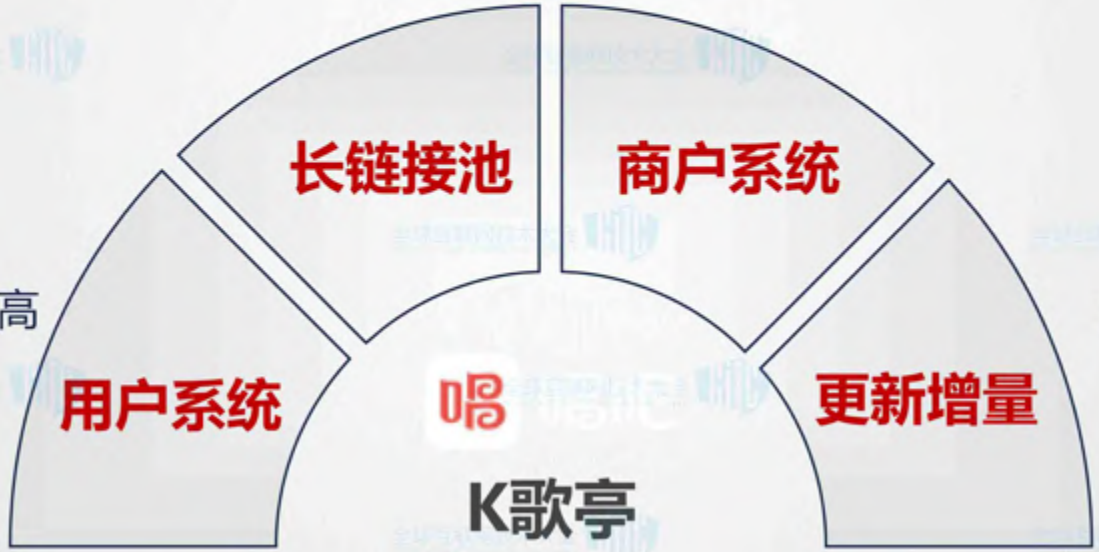
PART FOUR

- 业务场景
- 架构思路
- 集成测试交付

业务场景

突发流量

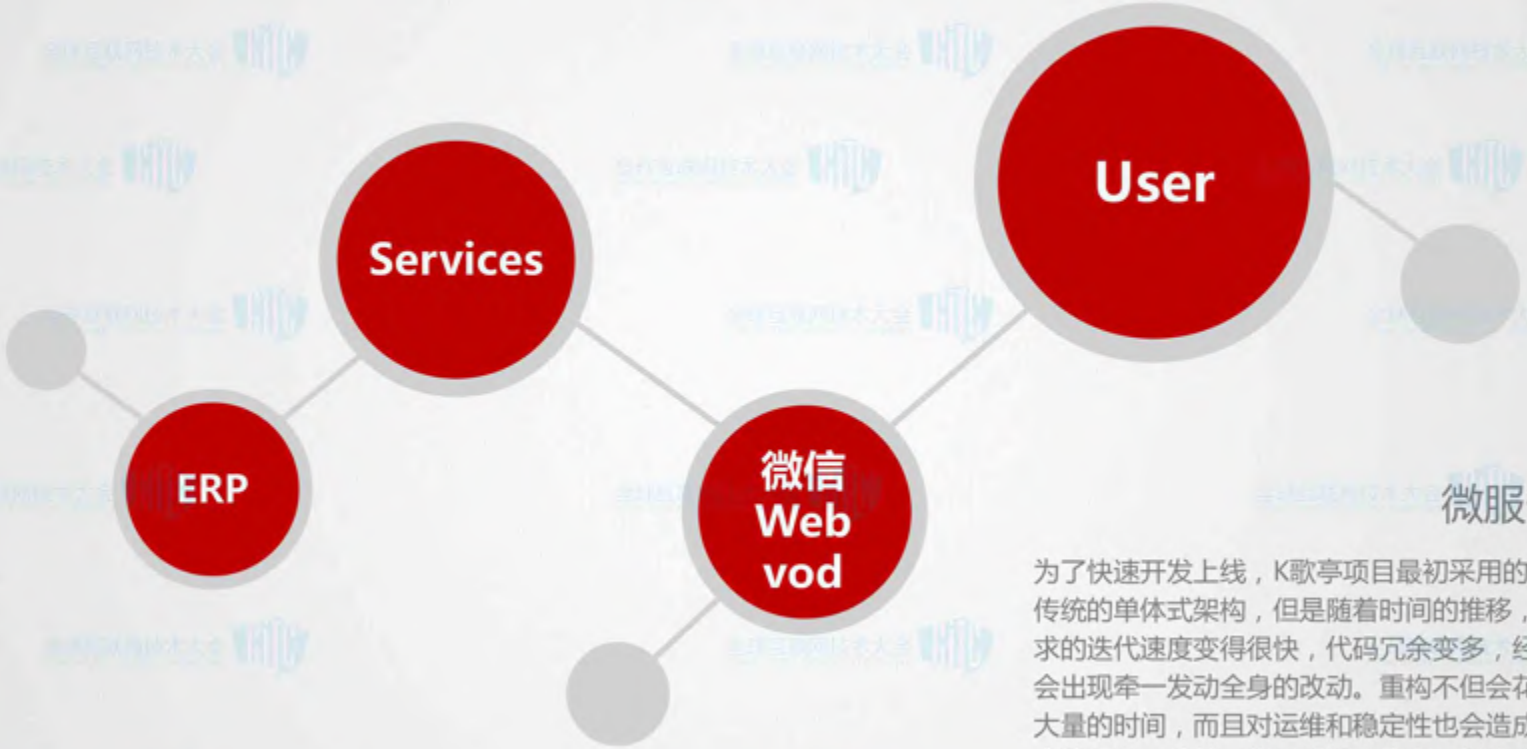
峰值没有规律



稳定性要求高

业务复杂度高

业务场景



微服务

为了快速开发上线，K歌亭项目最初采用的是传统的单体式架构，但是随着时间的推移，需求的迭代速度变得很快，代码冗余变多，经常会出现牵一发动全身的改动。重构不但会花费大量的时间，而且对运维和稳定性也会造成很大的压力；此外，代码的耦合度高，新人上手较困难，往往需要通读大量代码才不会踩进坑里。

业务场景

思考

从单体式结构转向微服务架构中会持续碰到服务边界划分的问题：比如，我们有user服务来提供用户的基础信息，那么用户的头像和图片等是应该单独划分为一个新的service更好还是应该合并到user服务里呢？

颗粒度大

服务的粒度划分的过粗，那就回到了单体式的老路；

颗粒度小

如果过细，那服务间调用的开销就变得不可忽视了，管理难度也会指数级增加。

业务场景

1. 不依赖或者极少依赖其他的业务服务。
2. 有自己的独立业务语义
3. 为超过2个客户端提供数据

架构思路

在采用了微服务架构之后，我们就可以动态调节服务的资源分配从而应对压力、服务自治、可独立部署、服务间解耦。开发人员可以自由选择自己开发服务的语言和存储结构等，目前整体上使用PHP做基础的Web服务和接口层，使用Go语言来做长连接池等其他核心服务，服务间采用thrift来做RPC交互。

架构思路-容器编排

DC/OS

作为Mesosphere公司的拳头产品，基本上是希望一统天下的节奏。所以组件很多，功能也很全面。但是对于我们在进行微服务架构初期，功能过于庞大，学习成本比较高，后期的生产环境维护压力也比较大

Swarm

Docker公司自己做的容器编排工具，当时了解到100个以上物理节点会有无响应的情况，对于稳定性有一些担忧。

kubernetes

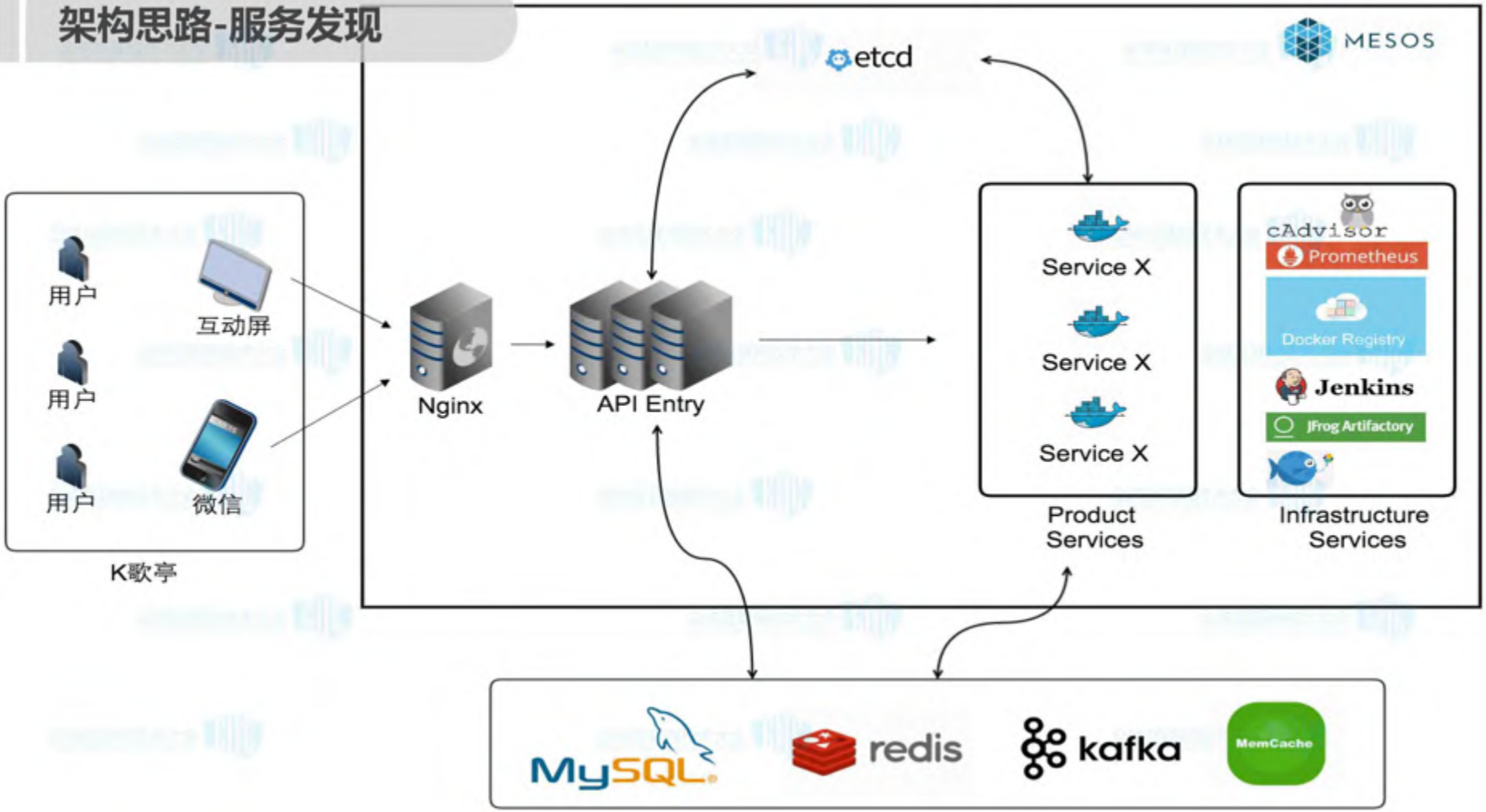
Google开源的的容器编排工具，在选型初期还没有很多公司使用的案例，同时也听到了很多关于稳定性的声音，所以没有考虑

mesos

marathon

因为了解到Twitter已经把Mesos用于生产环境，并且感觉架构和功能也相对简单，所以最后选择了Mesos+Marathon作为容器编排的工具

架构思路-服务发现


etcd
MESOS
用户
用户
用户
互动屏
微信
K歌亭
Nginx
API Entry
Service X
Service X
Service X
Product Services
cAdvisor
Prometheus
Docker Registry
Jenkins
JFrog Artifactory
Infrastructure Services
MySQL
redis
kafka
MemCache

架构思路-服务发现

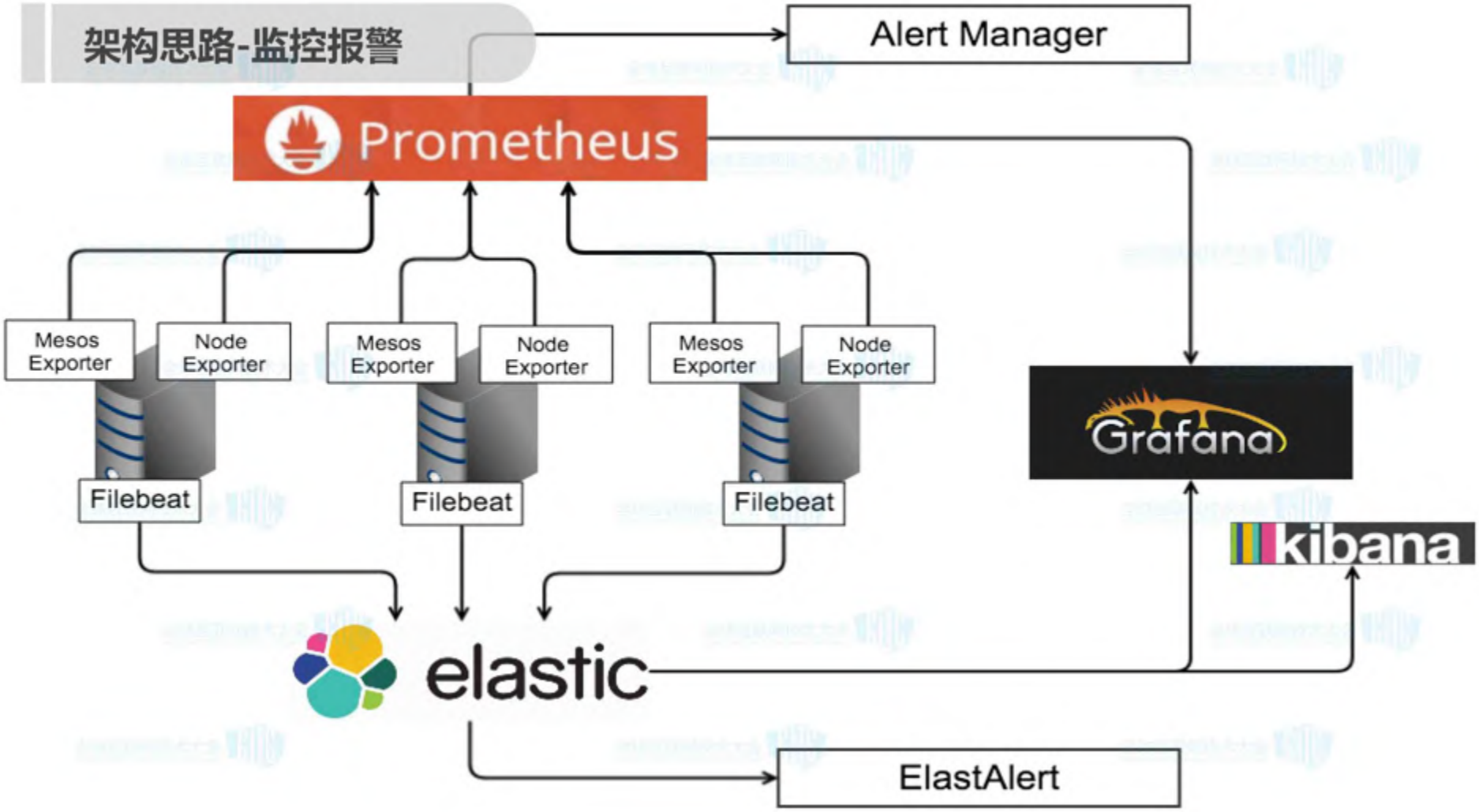
我们采用了etcd作为服务发现的组件，etcd是一个高可用的分布式环境下的 key/value 存储服务。在etcd中，存储是以树形结构来实现的，非叶结点定义为文件夹，叶结点则是文件。我们约定每个服务的根路径为/v2/keys/service/$service_name/，每个服务实例的实际地址则存储于以服务实例的uuid为文件名的文件中，比如账户服务account service当前启动了3个可以实例，那么它在etcd中的表现形式则如下图:

当一个服务实例向etcd写入地址成功时我们就可以认为当前服务实例已经注册成功，那么当这个服务实例由于种种原因down掉了之后，服务地址自然也需要失效，那么在etcd中要如何实现呢？

我们服务发现的机制是每个服务自注册，即每个服务启动的时候先得到宿主机器上面的空闲端口；然后随机一个或多个给自己并监听，当服务启动完毕时开始向etcd集群注册自己的服务地址，而服务的使用者则从etcd中获取所需服务的所有可用地址，从而实现服务发现。

```
root@host11:~# curl -XGET 'http://192.168.1.11:2379/v2/keys/service/account/'|pj
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   702  100   702    0     0   447k      0 --:--:-- --:--:-- --:--:--  685k
{
    "action": "get",
    "node": {
        "createdIndex": 163416,
        "dir": true,
        "key": "/service/account",
        "modifiedIndex": 163416,
        "nodes": [
            {
                "createdIndex": 24073802,
                "expiration": "2016-12-22T17:34:25.941528473Z",
                "key": "/service/account/25771c2e-ae82-44c1-a7f8-97f7b5000270",
                "modifiedIndex": 25190849,
                "ttl": 6,
                "value": "10.0.30.101:41337"
            },
            {
                "createdIndex": 24011335,
                "expiration": "2016-12-22T17:34:29.438398296Z",
                "key": "/service/account/68cbafbf-f2f9-481f-9624-a965d3db5a83",
                "modifiedIndex": 25190865,
                "ttl": 9,
                "value": "10.0.30.104:39568"
            },
            {
                "createdIndex": 3209068,
                "expiration": "2016-12-22T17:34:29.319138366Z",
                "key": "/service/account/d8bc618b-d33f-4977-b727-0f9107df2f08",
                "modifiedIndex": 25190863,
                "ttl": 9,
                "value": "192.168.1.11:55589"
            }
        ]
    }
}
```

架构思路-监控报警


Alert Manager
Prometheus
Mesos Exporter
Node Exporter
Mesos Exporter
Node Exporter
Mesos Exporter
Node Exporter
Filebeat
Filebeat
Filebeat
Grafana
kibana
elastic
ElastAlert

架构思路-监控报警

我们选择Prometheus汇总监控数据，用ElasticSearch汇总日志，主要的原因有：

1生态相对成熟，相关文档很全面，从通用的到专用的各种exporter也很丰富。

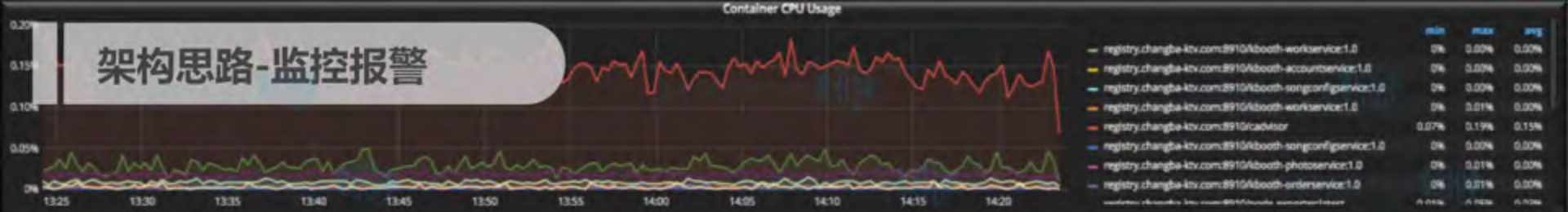
2查询语句和配置简单易上手。

3原生具有分布式属性。

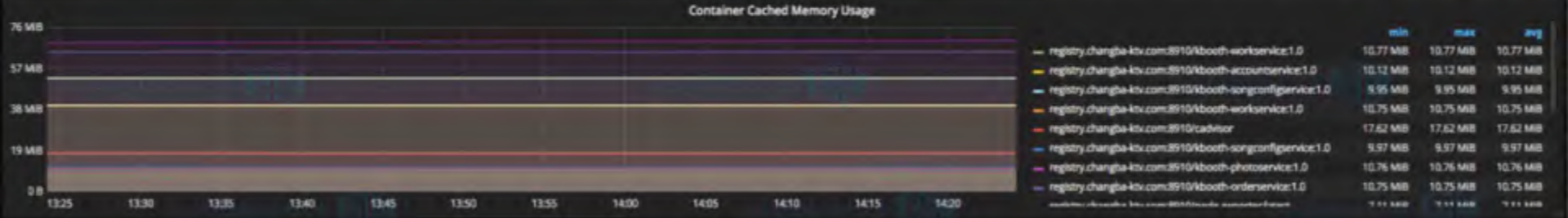
4所有组件都可以部署在Docker容器内。

Mesos Exporter，是Prometheus开源的项目，可以用来收集容器的各项运行指标。我们主要使用了对于Docker容器的监控这部分功能，针对每个服务启动的容器数量，每个宿主机上启动的容器数量，每个容器的CPU、内存、网络IO、磁盘IO等。并且本身他消耗的资源也很少，每个容器分配0.2CPU，128MB内存也毫无压力。

架构思路-监控报警


Container CPU Usage
0.20%
0.15%
0.10%
0.05%
0%
13:25
13:30
13:35
13:40
13:45
13:50
13:55
14:00
14:05
14:10
14:15
14:20
min
max
avg
registry.changba-ktv.com:8910/kbooth-workservice:1.0 0% 0.00% 0.00%
registry.changba-ktv.com:8910/kbooth-accountservice:1.0 0% 0.00% 0.00%
registry.changba-ktv.com:8910/kbooth-songconfigservice:1.0 0% 0.00% 0.00%
registry.changba-ktv.com:8910/kbooth-workservice:1.0 0% 0.01% 0.00%
registry.changba-ktv.com:8910/cadvisor 0.07% 0.19% 0.15%
registry.changba-ktv.com:8910/kbooth-songconfigservice:1.0 0% 0.00% 0.00%
registry.changba-ktv.com:8910/kbooth-photoservice:1.0 0% 0.01% 0.00%
registry.changba-ktv.com:8910/kbooth-orderservice:1.0 0% 0.01% 0.00%


Container Memory Usage
954 MiB
763 MiB
572 MiB
381 MiB
191 MiB
0 B
13:25
13:30
13:35
13:40
13:45
13:50
13:55
14:00
14:05
14:10
14:15
14:20
min
max
avg
registry.changba-ktv.com:8910/kbooth-workservice:1.0 14.61 MiB 14.85 MiB 14.68 MiB
registry.changba-ktv.com:8910/kbooth-accountservice:1.0 14.13 MiB 14.48 MiB 14.24 MiB
registry.changba-ktv.com:8910/kbooth-songconfigservice:1.0 14.21 MiB 14.49 MiB 14.27 MiB
registry.changba-ktv.com:8910/kbooth-workservice:1.0 19.41 MiB 19.80 MiB 19.48 MiB
registry.changba-ktv.com:8910/cadvisor 58.99 MiB 61.81 MiB 60.75 MiB
registry.changba-ktv.com:8910/kbooth-songconfigservice:1.0 13.72 MiB 14.02 MiB 13.80 MiB
registry.changba-ktv.com:8910/kbooth-photoservice:1.0 14.66 MiB 14.91 MiB 14.68 MiB
registry.changba-ktv.com:8910/kbooth-orderservice:1.0 14.61 MiB 15.04 MiB 14.70 MiB

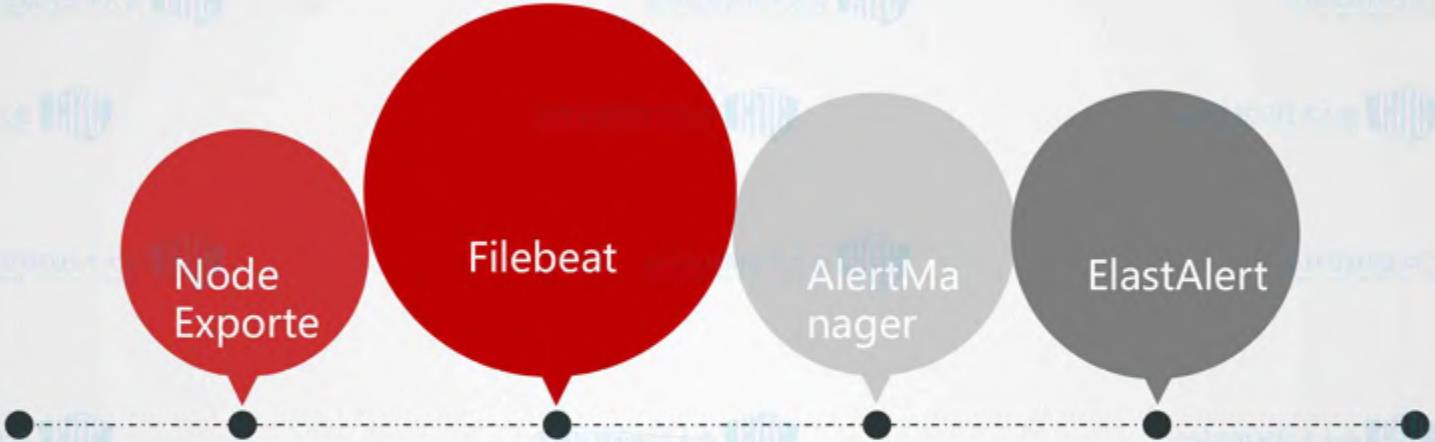

Container Cached Memory Usage
76 MiB
57 MiB
38 MiB
19 MiB
0 B
13:25
13:30
13:35
13:40
13:45
13:50
13:55
14:00
14:05
14:10
14:15
14:20
min
max
avg
registry.changba-ktv.com:8910/kbooth-workservice:1.0 10.77 MiB 10.77 MiB 10.77 MiB
registry.changba-ktv.com:8910/kbooth-accountservice:1.0 10.12 MiB 10.12 MiB 10.12 MiB
registry.changba-ktv.com:8910/kbooth-songconfigservice:1.0 9.95 MiB 9.95 MiB 9.95 MiB
registry.changba-ktv.com:8910/kbooth-workservice:1.0 10.75 MiB 10.75 MiB 10.75 MiB
registry.changba-ktv.com:8910/cadvisor 17.62 MiB 17.62 MiB 17.62 MiB
registry.changba-ktv.com:8910/kbooth-songconfigservice:1.0 9.97 MiB 9.97 MiB 9.97 MiB
registry.changba-ktv.com:8910/kbooth-photoservice:1.0 10.76 MiB 10.76 MiB 10.76 MiB
registry.changba-ktv.com:8910/kbooth-orderservice:1.0 10.75 MiB 10.75 MiB 10.75 MiB


Container Network Input
24 KiB
20 KiB
15 KiB
10 KiB
5 KiB
0 B
min
max
avg
registry.changba-ktv.com:8910/kbooth-workservice:1.0 0 B 0 B 0 B
registry.changba-ktv.com:8910/kbooth-workservice:1.0 12.25 KiB 23.15 KiB 21.95 KiB
registry.changba-ktv.com:8910/kbooth-accountservice:1.0 0 B 0 B 0 B
registry.changba-ktv.com:8910/kbooth-accountservice:1.0 10.94 KiB 23.15 KiB 21.88 KiB
registry.changba-ktv.com:8910/kbooth-songconfigservice:1.0 0 B 0 B 0 B
registry.changba-ktv.com:8910/kbooth-songconfigservice:1.0 12.25 KiB 23.15 KiB 21.83 KiB
registry.changba-ktv.com:8910/kbooth-workservice:1.0 0 B 0 B 0 B
registry.changba-ktv.com:8910/kbooth-workservice:1.0 12.25 KiB 23.25 KiB 21.87 KiB

架构思路-监控报警

Node exporter，是Prometheus开源的项目，用来收集物理机器上面的各项指标。之前一直使用Zabbix来监控物理机器的各项指标，这次使用NodeExporter+Prometheus主要是出于效率和对于容器生态的支持两方面考虑。时序数据库在监控数据的存储和查询的效率方面较关系数据库的优势确实非常明显，具体展示在Grafana上面如下图：

架构思路-监控报警



是Prometheus开源的项目，用来收集物理机器上面的各项指标。

用来替换Logstash-forwarder的日志收集组件，可以收集宿主机上面的各种日志。我们所有的服务都会挂载宿主机的本地路径，每个服务容器的会把自己的GUID写入日志来区分来源。

Prometheus配置好了报警之后可以通过AlertManager发送，但是对于报警的聚合的支持还是很弱的。

主要的功能是定时轮询ElasticSearch的API来发现是否达到报警的临界值，它的一个特色是预定义了各种报警的类型，比如frequency、change、flatline、cardinality等，非常灵活

架构思路-trace

对于一套微服务的系统结构来说，**最大的难点并不是实际业务代码的编写，而是服务的监控和调试以及容器的编排。**微服务相对于其他分布式架构的设计来说会把服务的粒度拆到更小，一次请求的路径层级会比其他结构更深，同一个服务的实例部署很分散，当出现了性能瓶颈或者bug时如何第一时间定位问题所在的节点极为重要，所以对于微服务来说，完善的trace机制是系统的核心之一。

在trace系统中有如下几个概念

（1）Annotation

一个annotation是用来即时的记录一个事件的发生，以下是一系列预定义的用来记录一次请求开始和结束的核心annotation

1cs - Client Start。 客户端发起一次请求时记录

2sr - Server Receive。 服务器收到请求并开始处理，sr和cs的差值就是网络延时和时钟误差

3ss - Server Send: 服务器完成处理并返回给客户端，ss和sr的差值就是实际的处理时长

4cr - Client Receive: 客户端收到回复时建立。 标志着一个span的结束。我们通常认为一但cr被记录了，一个RPC调用也就完成了。

其他的annotation则在整个请求的生命周期里建立以记录更多的信息 。

（2）Span

由特定RPC的一系列annotation构成Span序列，span记录了很多特定信息如 traceId， spandId， parentId和RPC name。

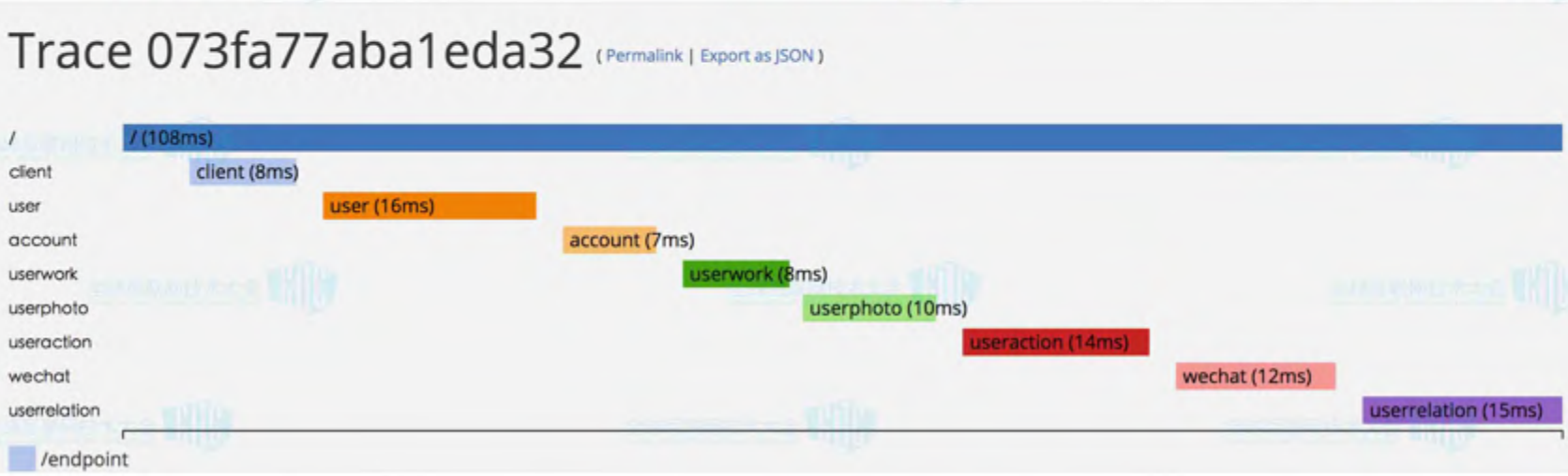
Span通常都很小，例如序列化后的span通常都是kb级别或者更小。 如果span超过了kb量级那就会有很多其他的问题，比如超过了kafka的单条消息大小限制(1M)。 就算你提高kafka的消息大小限制，过大的span也会增大开销，降低trace系统的可用性。 因此，只存储那些能表示系统行为的信息即可。

（3）Trace

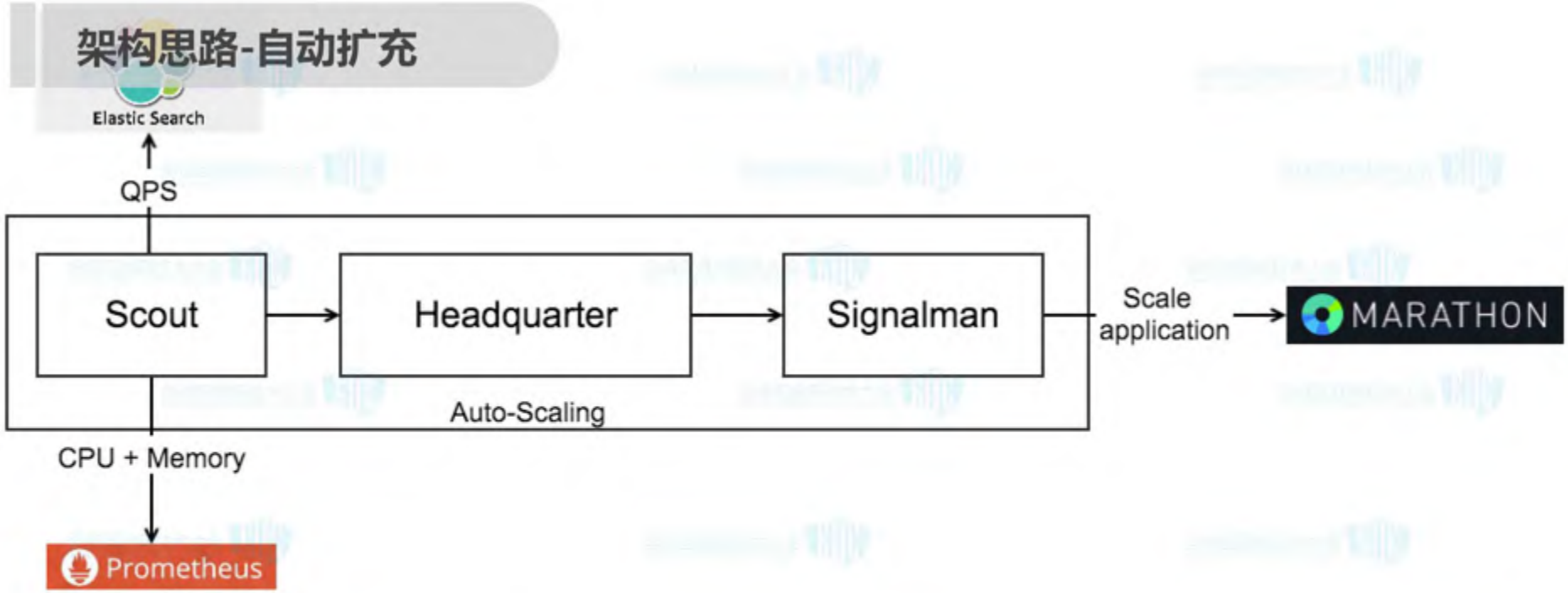
一个trace中所有的span都共享一个根span，trace就是一个拥有共同traceid的span的集合，所有的span按照spanid和父spanid来整合成树形，从而展现一次请求的调用链。

架构思路-trace

每个色块是一个span，表明了实际的执行时间，通常的调用层级不会超过10，点击span则会看到每个span里的annotation记录的很多附加信息，比如服务实例所在的物理机的IP和端口等，trace系统的消耗一般不会对系统的表现影响太大，通常情况下可以忽略，但是当QPS很高时trace的开销就要加以考量，通常会调整采样率或者使用消息队列等来异步处理。不过，异步处理会影响trace记录的实时性，需要针对不同业务加以取舍。



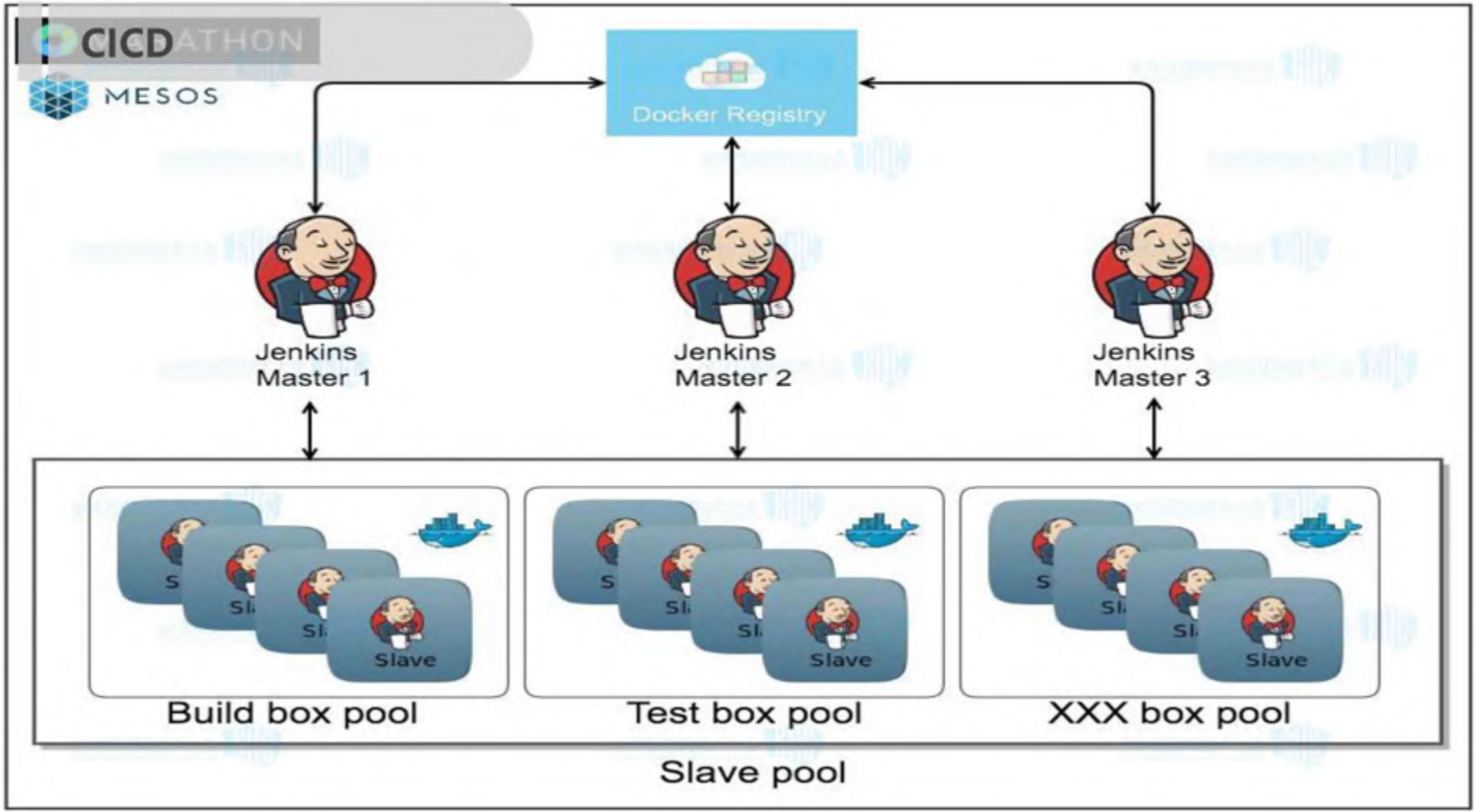
架构思路-自动扩充



1、Scout：用于从各个数据源取得自动扩容所需要的数据。由于我们的日志全部都汇总在ElasticSearch里面，容器的运行指标都汇总到Prometheus里面，所以我们的自动扩容系统会定时的请求二者的API，得到每个服务的实时QPS、CPU和内存信息，然后送给Headquarter。

2、Headquarter：用于数据的处理和是否触发扩缩容的判断。把从Scout收到的各项数据与本地预先定义好的规则进行比对，如果有两个指标超过定义好的规则，则通知到Signalman模块。

3、Signalman：用于调用各个下游组件执行具体扩缩容的动作。目前我们只会调用Marathon的/v2/apps/{app_id}接口，去完成对应服务的扩容。因为我们的服务在容器启动之后会自己向etcd注册，所以查询完容器状态之后，扩缩容的任务就完成了。


CICD ATHON
MESOS
Docker Registry
Jenkins Master 1
Jenkins Master 2
Jenkins Master 3
Slave
Slave
Slave
Build box pool
Test box pool
XXX box pool
Slave pool

CICD

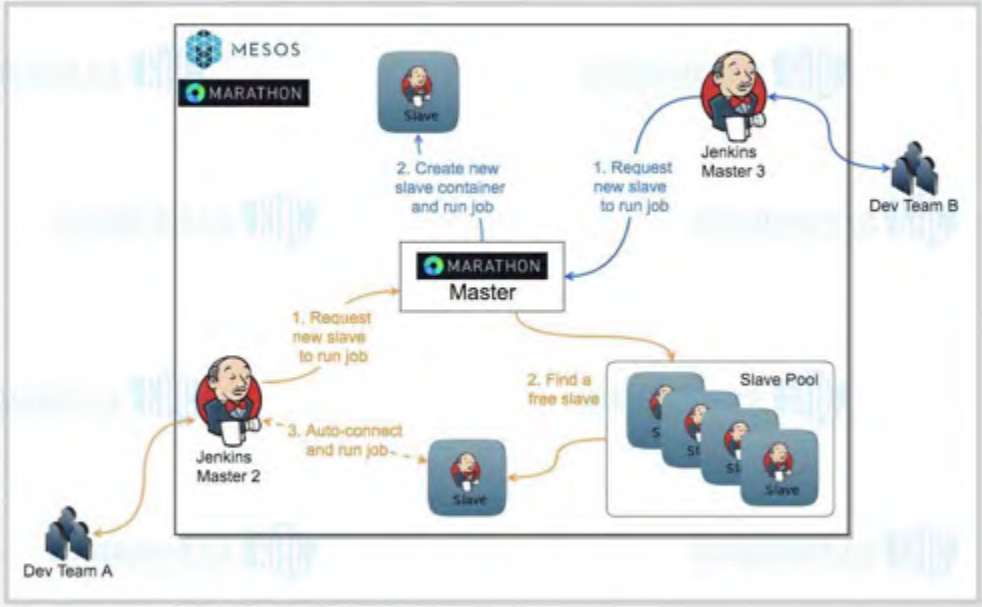
1**不同开发团队之间使用不同的Jenkins Master。**把公用的权限、升级、配置和插件更新到私有Jenkins Master镜像里面，推到私有镜像仓库，然后通过Marathon部署新的Master镜像，新团队拿到的Jenkins Master就预安装好了各种插件，各个现有团队可以无缝接收到整体Jenkins的升级。

2**各种不同环境的Jenkins Slave，做成Slave镜像。**按照需要，可以通过Swarm Plugin自动注册到Jenkins master，从而组织成slave pool的形式；也可以每个job自己去启动自己的容器，然后在容器里面去执行任务。

3**Jenkins job从容器调度的角度分成两类。**

•环境敏感型：比如编译任务，需要每次编译的环境完全干净，我们会从镜像仓库拉取一个全新的镜像启动容器，去执行Job，然后再Job执行完成之后关闭容器。

•时间敏感型：比如执行测试的Job，需要尽快得到测试结果，但是测试机器的环境对于测试结果没什么影响，我们就会从已经启动好的Slave Pool里面去拉取一个空闲的Slave去执行Job。然后再根据Slave被使用的频率去动态的扩缩容Slave pool的大小就好了。



CICD

我们定义了我们自己的持续集成与持续交付的流程。其中除了大规模使用Jenkins与一些自定制的Jenkins插件之外，我们也自己研发了自己的部署系统——HAWAII。

在HAWAII中可以很直观的查看各个服务与模块的持续集成结果，包括最新的版本，SCM revision，测试结果等信息，然后选择相应的版本去部署生产环境。

Dashboard
版本库
部署历史

产品	marsapi
版本号	2016.12.29.006
SCM地址	http://www.changbagroup.com/repos/kulv/newtv/apitvserver
SCM版本	152953
checksum	19066fb828e6a68ec7ecca7ce0764b14
版本库地址	192.168.32.101
版本库路径	/home/ci/artifactory
存档时间	2016-12-29 16:16:20
状态	archived
部署时间	
主机名:	192.168.32.105
targetdirectory:	/home/wwwroot/api.newtv.com

guid	buildid	部署时间	上一个svn版本
2068	2016.10.25.001	2016-10-25 11:08:28	143226

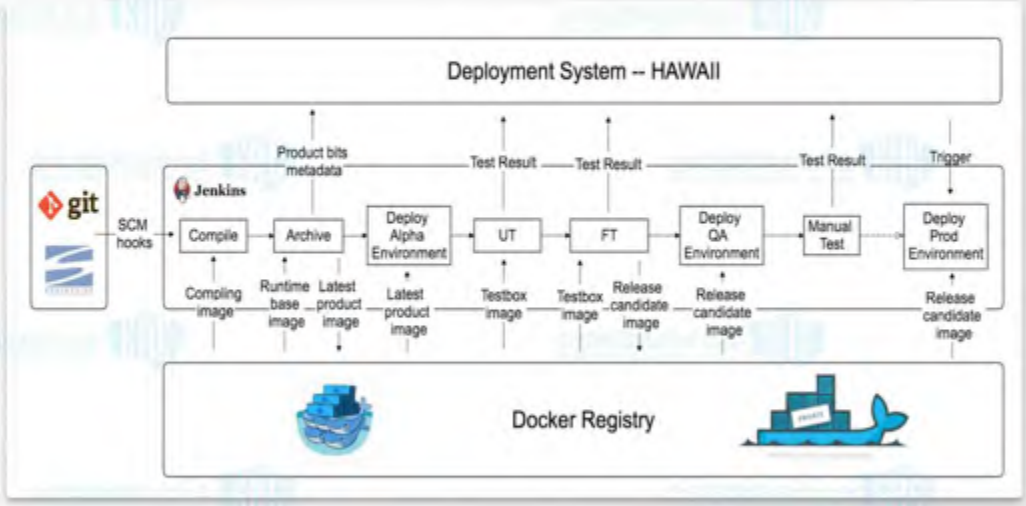
```
dex: tools/cdn_auto_fix_rito.php
===================================================================
 tools/cdn_auto_fix_rito.php (revision 152953)
+ tools/cdn_auto_fix_rito.php (revision 143226)
@ -1,39 +0,0 @@
equire __DIR__ . '/../common/common.inc.php';
obal $newtv_read,$db_internal_read;

egintime = "2016-10-15 00:00:00";
ndtime = "2016-10-19 00:00:00";
ype = "userid";
$type = "sessionid";
naxrito = 0.89;
otallive = 0;
hile(true) {
nexttime = date('Y-m-d H:i:s', strtotime($begintime) + 86400);
' 这里用Userid更严格
sql = "select count(sessionid) as total,userid from session_info where starttime between '{$begintime}' and '{$nexttime}' group by
erid having total > '{$totallive}' order by total desc";
res = $newtv_read->getAll($sql);
reach($res as $line) {
userid = $line['userid'];
sql = "select count(*) from cdn_error_record where userid='{$userid}' and time between '{$begintime}' and '{$nexttime}'";
```

	结果	Total	Pass	fail	skip
Smoke	build passing	57	57	0	0
Uat	build passing	35	35	0	0

部署	部署主机	文件分发&Checksum	API校验 - Host	API校验 - 线上
	192.168.32.105	build passing	build running	未开始

CICD

- SVN或者GIT收到新的代码提交之后，会通过hook启动相应的Jenkins job，触发整个CI流程。
- Jenkins从私有镜像仓库拉取相对应的编译环境，完成代码的编译。
- Jenkins从私有镜像仓库拉取相对应的运行时环境，把上一步编译好的产品包打到镜像里面，并生成一个新版本的产品镜像。产品镜像是最终可以部署到线上环境的镜像，该镜像的metadata也会被提交到部署系统HAWAII，包括GUID，SCM revision，Committer，时间戳等信息
- 将步骤3中生成的产品镜像部署到Alpha环境（该环境主要用于自动化回归测试，实际启动的容器其实是一个完整环境，包括数据容器，依赖的服务等）。
- Jenkins从私有镜像仓库拉取相对应的UT和FT的测试机镜像，进行测试。测试完成之后，会销毁Alpha环境和所有的测试机容器，测试结果会保存到部署系统HAWAII，并会邮件通知到相关人员。
- 如果测试通过，会将第3步生成的产品镜像部署到QA环境，进行一系列更大范围的回归测试和集成测试。测试结果也会记录到HAWAII，有测试不通过的地方会从第1步从头开始迭代。
- 全部测试通过后，就开始使用HAWAII把步骤3中生成的产品镜像部署到线上环境。



Thanks