[**唱吧DevOps的落地，微服务CI/CD的范本技术解读**](http://kb.cnblogs.com/page/565901/)

作者: 钮博彦、刘宇桐  来源: [InfoQ](http://www.infoq.com/cn/)  发布时间: 2017-04-19 22:21  阅读: 32321 次  推荐: 41   [原文链接](http://www.infoq.com/cn/articles/devops-landing-in-changba)   [[收藏]](javascript:add_to_wz())

**1、业务架构：从单体式到微服务**

　　K歌亭是唱吧的一条新业务线，旨在提供线下便捷的快餐式K歌方式，用户可以在一个电话亭大小的空间里完成K歌体验。K歌亭在客户端有VOD、微信和Web共三个交互入口，业务复杂度较高，如长连接池服务、用户系统服务、商户系统、增量更新服务、ERP等。对于服务端的稳定性要求也很高，因为K歌亭摆放地点不固定，很多场所的运营活动会造成突发流量。

　　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。

　　鉴于上述弊端，我们决定接下来的版本里采用微服务的架构模型。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？如果服务的粒度划分的过粗，那就回到了单体式的老路；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。目前为止还没有一个可以称之为服务边界划分的标准，只能根据不同的业务系统加以调节，目前K歌亭拆分的大原则是当一块业务不依赖或极少依赖其它服务，有独立的业务语义，为超过2个的其他服务或客户端提供数据，那么它就应该被拆分成一个独立的服务模块。

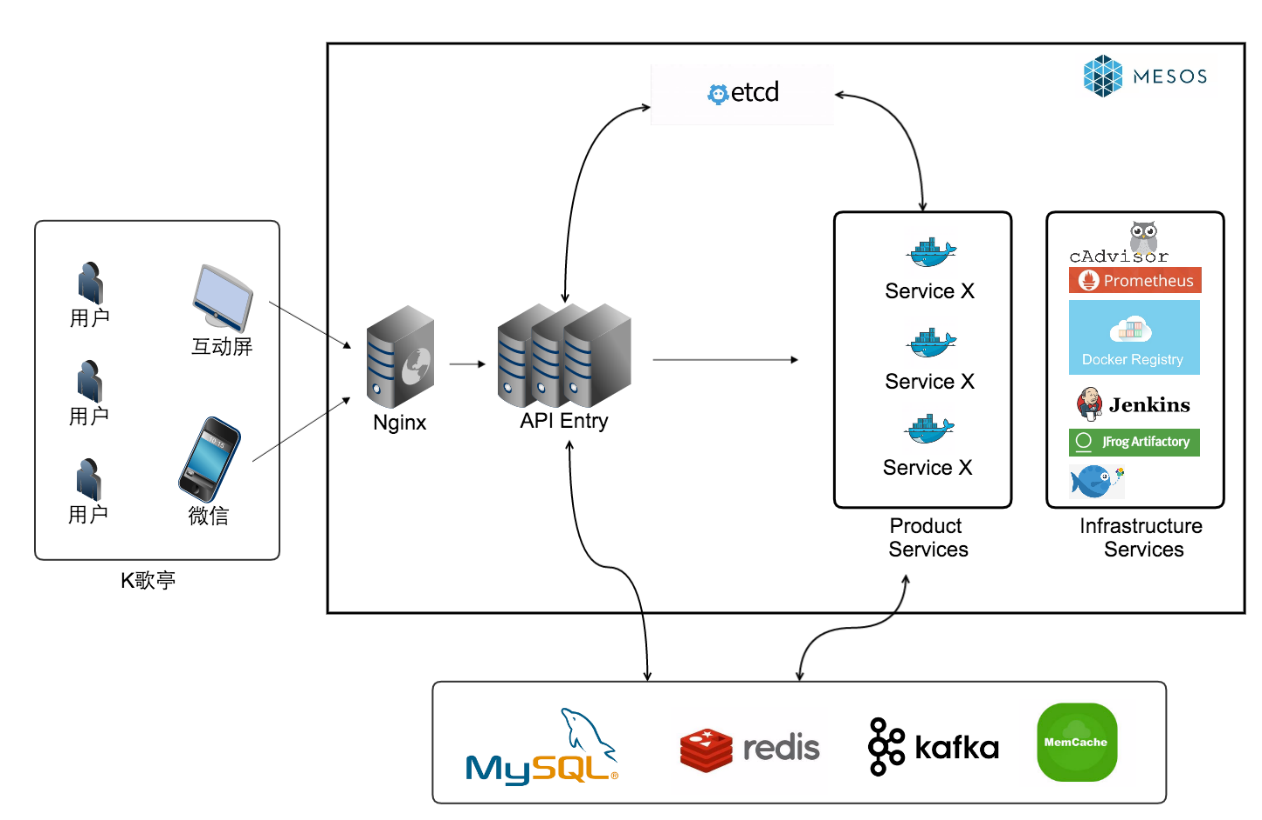
　　在采用了微服务架构之后，我们就可以动态调节服务的资源分配从而应对压力、服务自治、可独立部署、服务间解耦。开发人员可以自由选择自己开发服务的语言和存储结构等，目前整体上使用PHP做基础的Web服务和接口层，使用Go语言来做长连接池等其他核心服务，服务间采用thrift来做RPC交互。

**2、系统架构的构思与解读**

**2.1 容器编排**

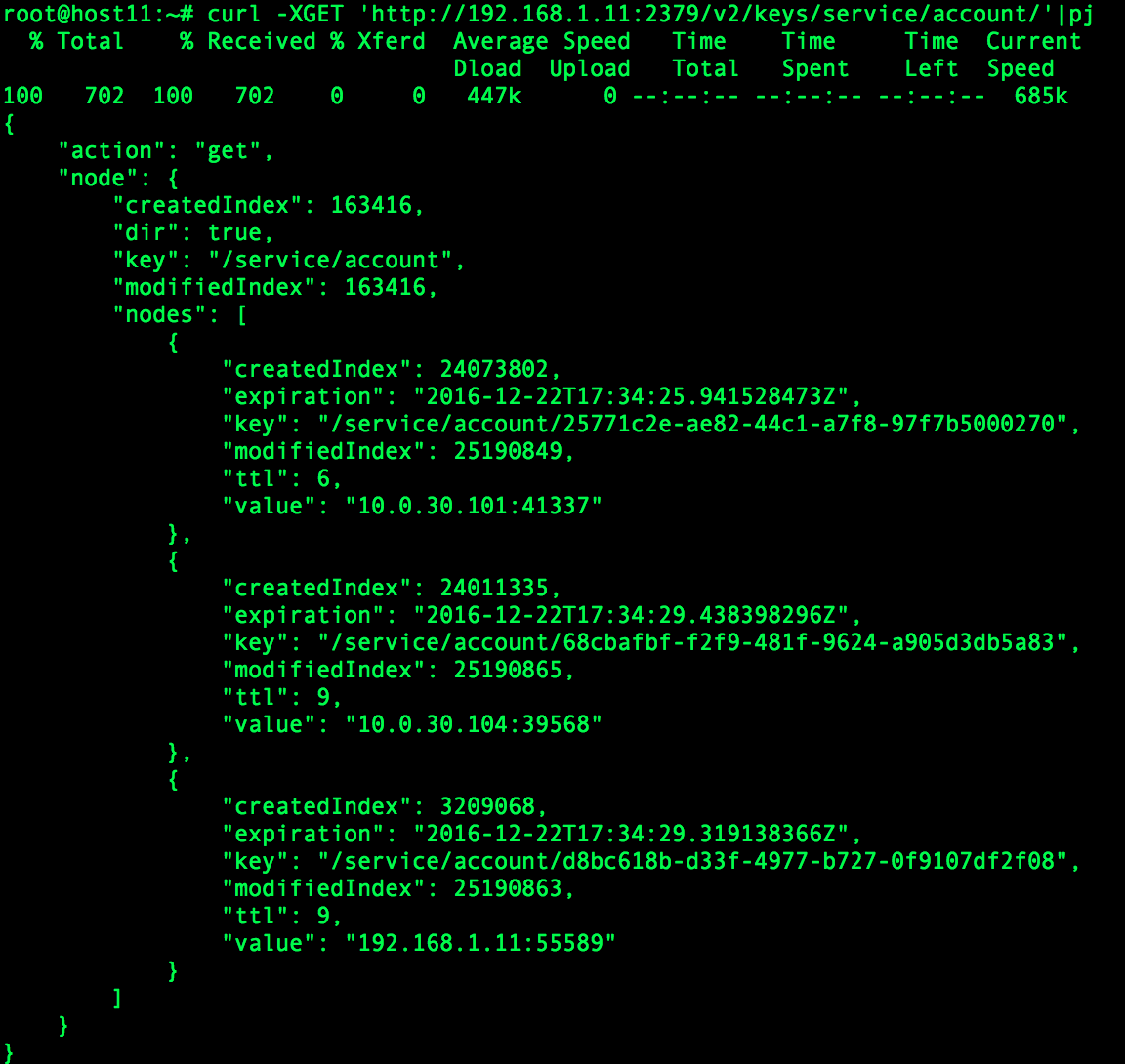
　　唱吧K歌亭的微服务架构采用了Mesos和Marathon作为容器编排的工具。在我们选型初期的时候还有三个其他选择，Kubernetes、 Swarm、 DC/OS：

* 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。但是在整个2016年，越来越多的公司开始在线上使用Kubernetes，其稳定性逐步提高，如果再选型应该也是个好选择。
* Mesos：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。



**2.2 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:



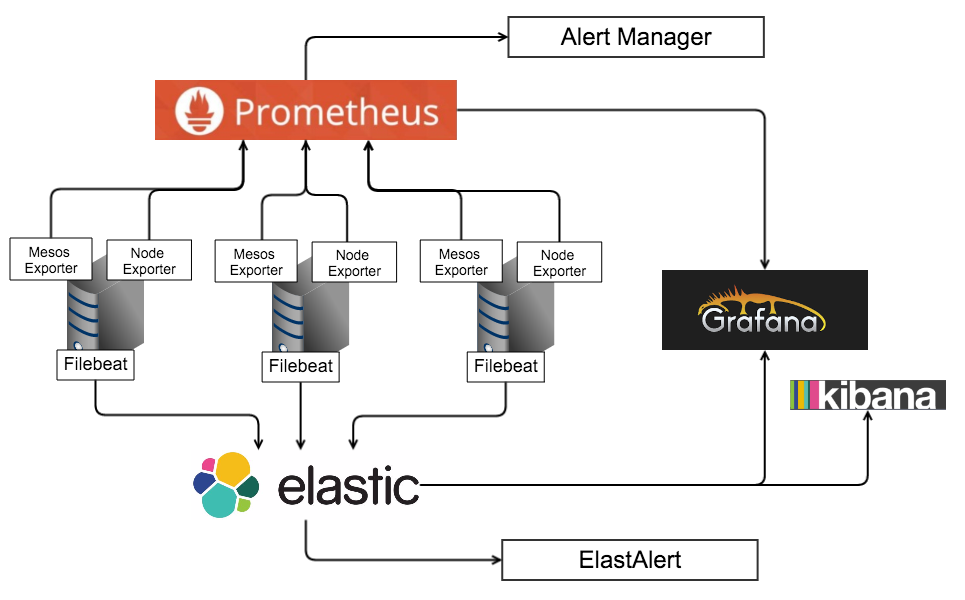
　　当一个服务实例向etcd写入地址成功时我们就可以认为当前服务实例已经注册成功，那么当这个服务实例由于种种原因down掉了之后，服务地址自然也需要失效，那么在etcd中要如何实现呢？

　　注意，图中的每个文件有一个ttl值，单位是秒，当ttl的值为0时对应的文件将会被etcd自动删除。当每个服务实例启动之后第一次注册时会把存活时间即ttl值初始化为10s，然后每隔一段时间去刷新ttl，用来像向etcd汇报自己的存活，比如7s，在这种情况下基本上可以保证服务有效性的更新的及时性。如果在一个ttl内服务down掉了，则会有10s钟的时间是服务地址有效；而服务本身不可用，这就需要服务的调用方做相应的处理，比如重试或这选择其它服务实例地址。

　　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。

　　同时，我们这样的机制也为容器以HOST的网络模式启动提供了保证。因为BRIDGE模式确实对于网络的损耗太大，在最开始就被我们否决了，采用了HOST模式之后网络方面的影响确实不是很大。

**2.3 监控，日志与报警**



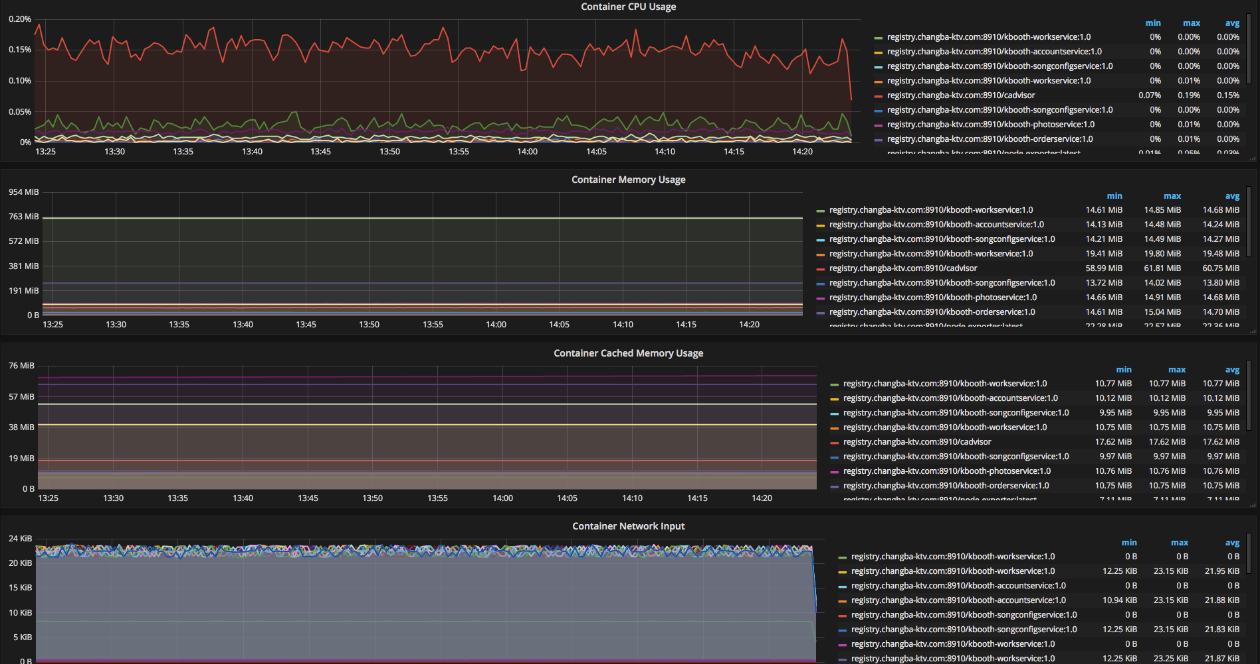
　　我们选择Prometheus汇总监控数据，用ElasticSearch汇总日志，主要的原因有：

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。

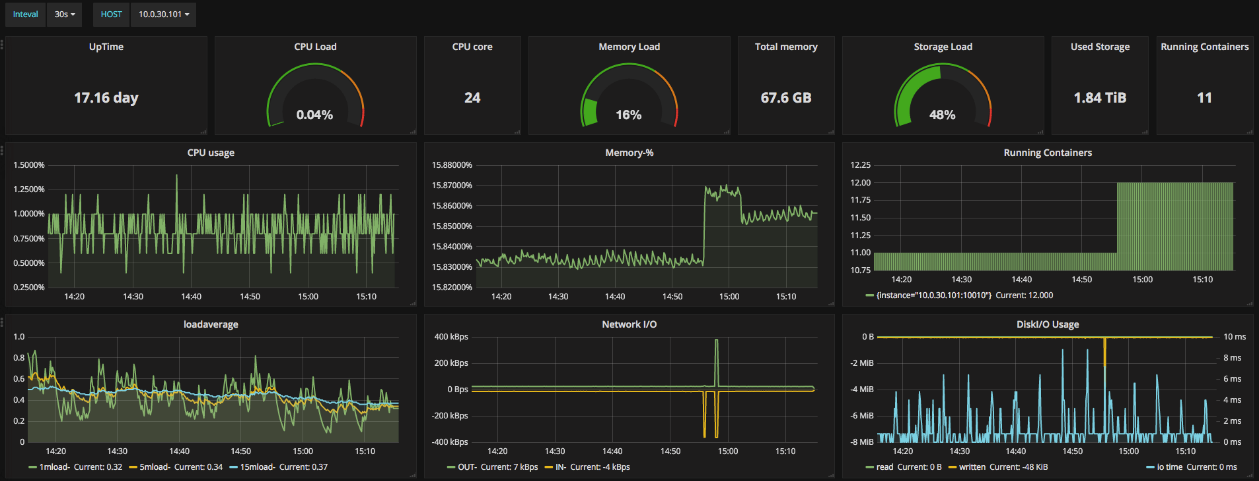
　　Mesos Exporter，是Prometheus开源的项目，可以用来收集容器的各项运行指标。我们主要使用了对于Docker容器的监控这部分功能，针对每个服务启动的容器数量，每个宿主机上启动的容器数量，每个容器的CPU、内存、网络IO、磁盘IO等。并且本身他消耗的资源也很少，每个容器分配0.2CPU，128MB内存也毫无压力。

　　在选择Mesos Exporter之前，我们也考虑过使用cAdvisor。cAdvisor是一个Google开源的项目，跟Mesos Exporter收集的信息八成以上都是类似的；而且也可以通过image字段也可以变相实现关联服务与容器，只是Mesos exporter里面的source字段可以直接关联到marathon的application id，更加直观一些。同时cAdvisor还可以统计一些自定义事件，而我们更多的用日志去收集类似数据，再加上Mesos Exporter也可以统计一些Mesos本身的指标，比如已分配和未分配的资源，所以我们最终选择了Mesos Exporter。

　　如下图，就是我们监控的部分容器相关指标在Grafana上面的展示：



　　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：



　　Filebeat是用来替换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。日志经由ElasticSearch汇总之后，聚合的Dashboard我们统一都会放在Grafana上面，具体排查线上问题的时候，会用Kibana去查看日志。

　　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。在下一阶段我们会引入一些Message Queue来自己的报警系统，加强对于报警的聚合和处理。

　　ElastAlert是Yelp的一个Python开源项目，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，也节省了我们很多二次开发的成本。

**2.4 事务追踪系统——KTrace**

　　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。

　　目前很多厂商使用的trace都是参考2010年Google发表的一篇论文《Dapper, a Large-Scale Distributed Systems Tracing Infrastructure》来实现的，其中最著名的当属twitter的zipkin，国内的如淘宝的eagle eye。由于用户规模量级的逐年提升，分布式设计的系统理念越来越为各厂商所接受，于是诞生了trace的一个实现标准opentracing ，opentracing标准目前支持[Go](http://github.com/opentracing/opentracing-go)、[JavaScript](http://github.com/opentracing/opentracing-javascript)、[Java](http://github.com/opentracing/opentracing-java)、 [Python](http://github.com/opentracing/opentracing-python)、[Objective-C](http://github.com/opentracing/opentracing-objc)、[C++](http://github.com/opentracing/opentracing-cpp)六种语言。 由sourcegraph开源的appdash是一款轻量级的，支持opentracing标准的开源trace组件，使用Go语言开发K歌亭目前对appdash进行了二次开发，并将其作为其后端trace服务（下文直接将其称之为Ktrace），主要原因是appdash足够轻量，修改起来比较容易。唱吧K歌亭业务的**胶水层**使用PHP来实现，appdash提供了对protobuf的支持，这样只需要我们自己在PHP层实现middleware即可。

　　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

1. cs - Client Start。 客户端发起一次请求时记录
2. sr - 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
3. ss - 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
4. cr - 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

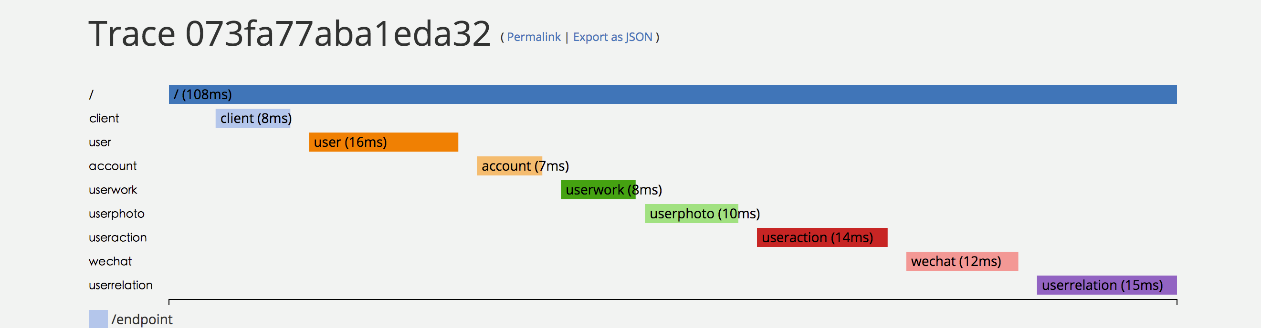
　　由特定RPC的一系列annotation构成Span序列，span记录了很多特定信息如 traceId， spandId， parentId和RPC name。

　　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。

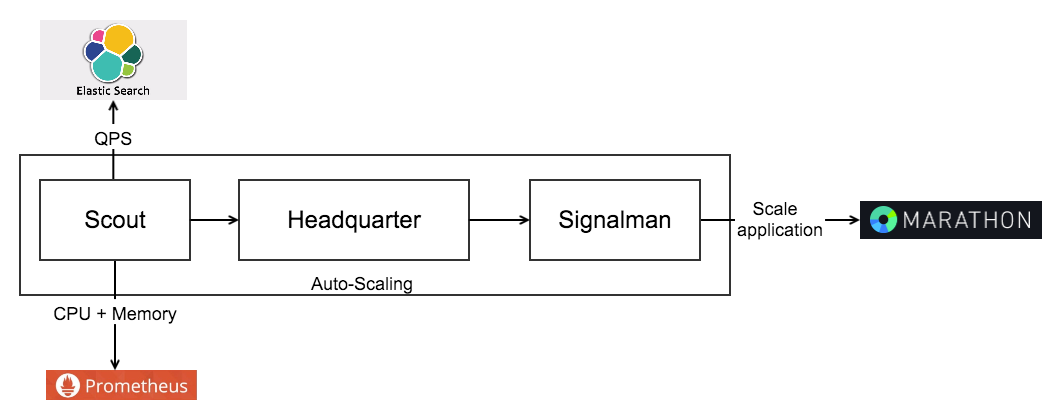
　　目前每次请求由PHP端生成traceid，并将span写入Ktrace，沿调用链传递traceid，每个service自己在有需要的地方埋点并写入Ktrace。举例如下图:



　　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。

　　目前K歌亭在生产环境里的QPS不超过1k，所以大部分的记录是直接写到ktrace里的，只有歌曲搜索服务尝试性的写在kafka里，由mqcollector收集并记录，ktrace的存储目前只支持MySQL。**一个好的trace设计可以极快的帮你定位问题，判断系统的瓶颈所在。**

**2.5 自动扩容**



　　在服务访问峰值的出现时，往往需要临时扩容来应对更多的请求。除了手动通过Marathon增加容器数量之外，我们也设计实现了一套自动扩缩容的系统来应对。我们扩缩容的触发机制很直接，根据各个服务的QPS、CPU占用、内存占用这三个指标来衡量，如果三个指标有两个指标达到，即启动自动扩容。我们的自动扩容系统包括3个模块：

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。

**3、基于Mesos+Marathon的CI/CD**

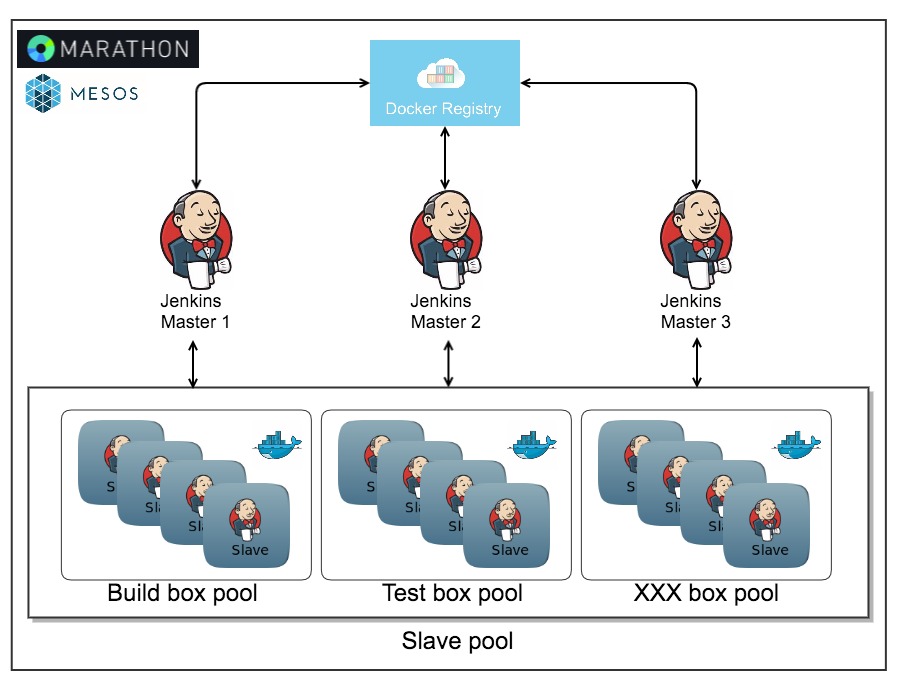
**3.1 持续集成与容器调度**

　　在唱吧，我们使用Jenkins作为持续集成的工具。主要原因是我们想在自己的机房维护持续集成的后端，所以放弃了Travis之类的系统。

　　在实施持续集成的工作过程中，我们碰到了下列问题：

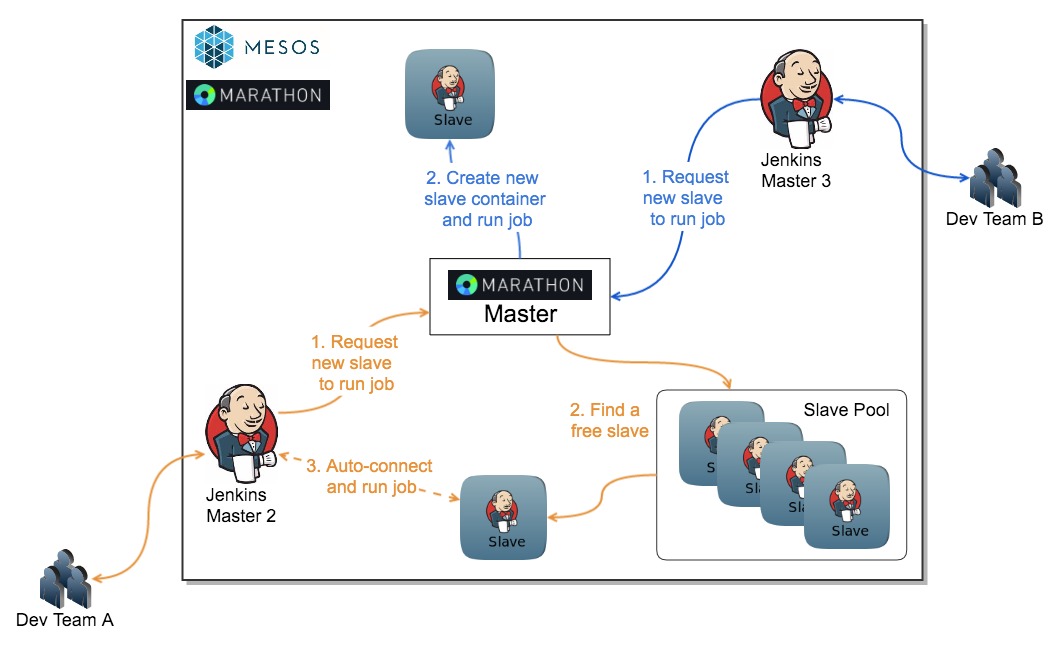
1. Jenkins Master的管理问题。多个团队共享一个Master，会导致权限管理困难，配置改动、升级门槛很高，Job创建和修改有很多规则；每个团队用自己的Master，会导致各个Master之间的插件、更新、环境维护有很多的重复工作。
2. Jenkins Slave 资源分配不平均：忙时Jenkins slave数量不足，Job运行需要排队；闲时Jenkins Slave又出现空闲，非常浪费资源。
3. Jenkins job运行需要的环境多种多样，比如我们就有PHP，java，maven，Go，python等多种编译运行环境，搭建和维护slave非常费时。
4. 多个开发人员的同时提交，各自的代码放到各自独立的测试环境进行测试。

　　基于以上问题，我们选择使用Mesos和Marathon来管理Jenkins集群，把Jenkins Master和Jenkins Slave都放到Docker容器里面，可以非常有效的解决以上问题。基础架构如下图：



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，如下图：

* 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。



**3.2 CI/CD流程**

　　基于上述的基础架构，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。

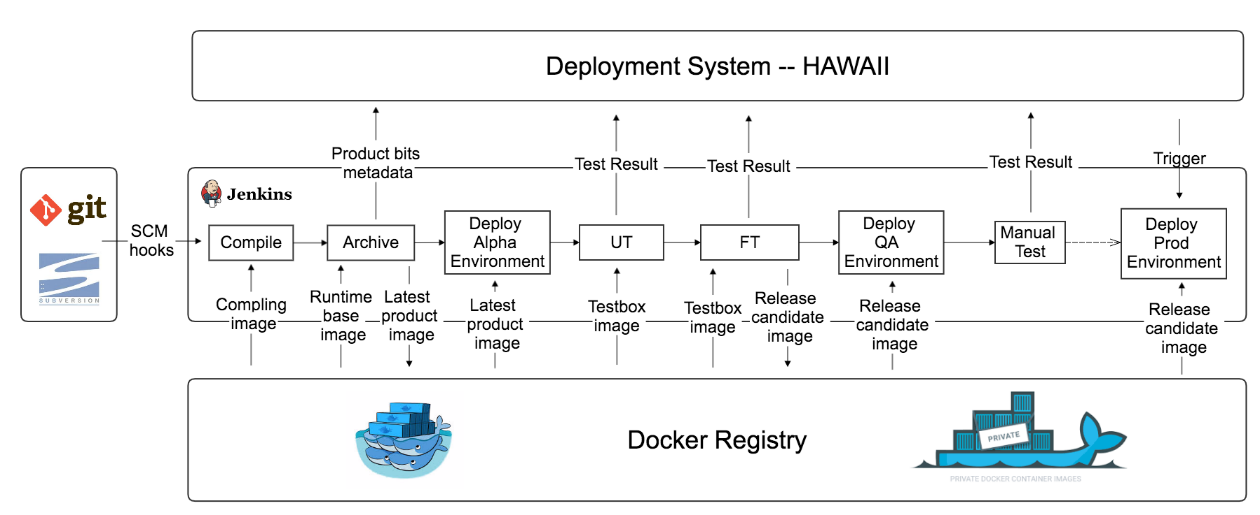
　　在HAWAII中可以很直观的查看各个服务与模块的持续集成结果，包括最新的版本，SCM revision，测试结果等信息，然后选择相应的版本去部署生产环境。



　　在部署之前，可以查看详细的测试结果和与线上版本的区别，以及上线过程中的各个步骤运行的状态。



　　基于上述基础架构，我们的CI/CD流程如下：



1. 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。
2. 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。
3. 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
4. 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）。
5. 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。
6. 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。
7. 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。

**4、小结**

　　随着互联网的高速发展，各个公司都面临着巨大的产品迭代压力，如何更快的发布高质量的产品，也是每个互联网公司都面临的问题。在这个大趋势下，微服务与DevOps的概念应运而生，在低耦合的同时实现高聚合，也对新时代的DevOps提出了更高的技术与理念要求。

　　这也是我们公司在这个新的业务线上面进行，进行尝试的主要原因之一。对于微服务、容器编排、虚拟化、DevOps这些领域，我们一步一步经历了从无到有的过程，所以很多方面都是本着从满足业务的目标来尽量严谨的开展，包括所有的服务与基础架构都进行了高并发且长时间的压力测试。

　　在下一步的工作中，我们也有几个核心研究的方向与目标，也希望能跟大家一起学习与探讨：

1. 完善服务降级机制
2. 完善报警机制
3. 完善负载均衡机制