# 【方案】去哪儿网徐磊：如何利用开源技术构建日处理130亿+的实时日志平台？

- HarkLee - 博客园 http://www.cnblogs.com/hark0623/p/5533593.html

【本文系互联网技术联盟（ITA1024）原创首发，转载或节选内容前需获授权（授权后一周以后可以转载），且必须在正文前注明：本文转自互联网技术联盟（ITA1024）技术分享实录,微信公众号：ita1024k】

徐磊

去哪儿网

高级运维开发工程师

互联网技术联盟

ITA1024讲师团成员

本篇文章整理自徐磊5月19日在『ITA1024运维技术精英群』里的分享实录：如何利用开源技术构建日处理130亿+的实时日志平台？

正文如下

平台建立的背景

随着业务规模的不断扩大，系统功能会变得越来越复杂，会自然演化为运行再多台服务器上的分布式系统。

为了应付业务的快速发展，降低开发难度，排除性能瓶颈，系统会不断拆分，演化成包含多种子服务的分布式系统，各子服务通过RPC相互调用，最后完成业务流程。

这个拆分和进化的过程是不可逆的，子系统越变越多，各种专用功能组件会不断被引入，系统和机器规模迅速膨胀。

当业务发展到像Qunar一样的规模时，系统会进化成为包含几千子服务，几万个服务器的庞大怪物，一个运维或者开发人员根本无法全面的了解系统中的每个逻辑，也无法通过人肉登录服务器grep日志的方式找到系统问题的产生的原因。

同时，随着多人协作问题定位，沟通变多，效率降低，反而阻碍业务的发展。比较有代表性的现象有新版本发布验证时间变长，完成一次发布要半天的时间，还有用户投诉问题定位时间变长。

为了解决这些问题，我们急需一个系统，具备汇总，检索，展示应用日志，串接事件，快速定位问题的能力，更需要满足：

●     可靠性，不丢消息

●    应对跨机房网络抖动或者故障

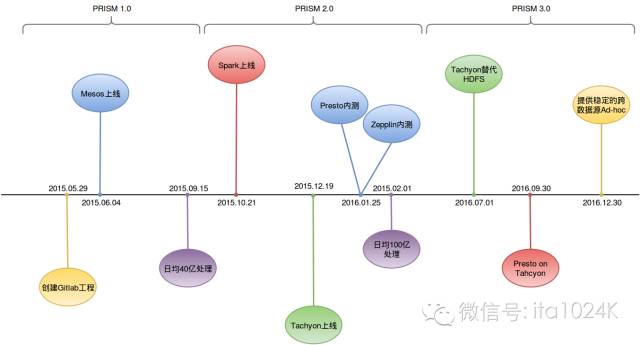
●    能够快速响应收集需求，并做相应的格式化

●    方便的查看实时数据

在这个需求的驱动下，2014年末开始着手建设实时日志平台。

平台演进的过程

平台目前已经经历了两个大版本的迭代，目前正在实施第三个版本。每天流经平台的日志数量在130亿（去重），写入ElasticSearch约10TB数据，分发给Spark Streaming大约3T左右数据，辐射140多个业务线。相关数据对接线上系统，做到实时反馈，如风控，推荐等功能。



技术选型

技术选型在我们平台演进的过程中一直都会有，这是因为每个阶段，平台功能的侧重点是不同的，导致选择相应的技术/框架时，除了要满足功能外，还要尽量匹配已有的结构。

基于这点出发，我们需要一个二次开发能力强，尽量轻量级的底层平台来统一管理资源和服务的接入，再此基础上，逐步构建我们的日志平台。

资源管理

对于日志类的应用，计算工作量会偏大一些，同时容易与业务压力成正比，比如access日志，订单日志和rpc调用日志等，同时又具备周期性，比如早8点至凌晨2点左右日志产生较多，凌晨2点至早8点反而是系统最闲的时候，日志基本没多少。

基于以上的场景，我们最先考虑的是选择一个统一的资源管理程序/框架来支撑上层的日志服务：

1.     轻量级：占用尽量小的资源；

2.     高效率：足够支撑未来集群规模的上涨；

3.     易维护：有API可以获取内部的运行状态和监控指标；

4.     定制化：万一无法满足全部需求还可以自己折腾折腾；

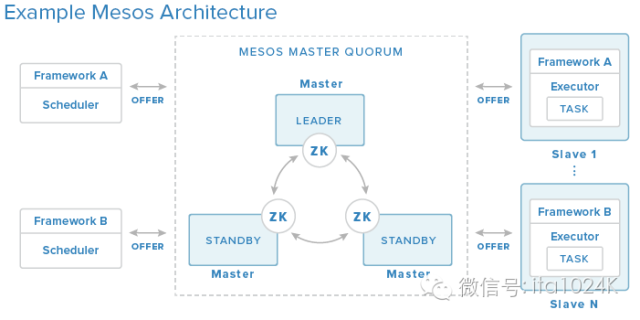
5.     社区好：用户/场景多，出了问题容易找到答案。

当时备选的主要有三个：ApacheYARN，Apache Mesos和GoogleKubernetes。我们简单的做了一些比较：

|  |  |  |  |
| --- | --- | --- | --- |
|  | YARN | Mesos | Kubernetes |
| 轻量级 | 每个slave部署一个node manager | 每个slave上部署一个mesos-slave进程 | 每个slave上部署一个kuberlet进程 |
| 高效率 | 可以支撑万台规模 | 可以支撑万台规模 | 当时支持的最大的集群规模还是百台 |
| 易维护 | Ambari可以做到部署+监控 | 有http api获取监控数据 | cAdvisor+Heapster做监控 |
| 定制化 | 支持JVM语言二次开发 | 支持JVM语言/Python/C++二次开发 | 支持Go的插件 |
| 社区支持 | 使用最广，社区活跃，案例多。几乎是数据平台的默认资源管理器/调度器。 | 在Docker热潮之前相对较默默无闻，最成功的大规模应用也仅是Twitter一家。社区在当时主要是Apache，Twitter和Mesosphere在支撑。 | 比较新的框架，社区同样火爆，发展速度非常快，当时除了GCE以外还未有知名案例。也考虑到Google的风格，比较害怕干一半跑了。 |

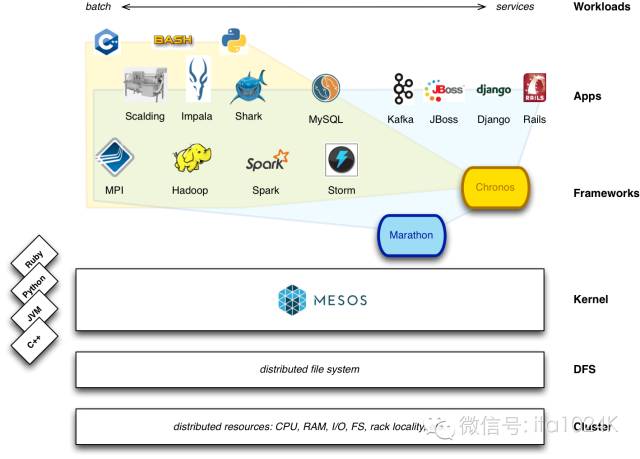
从三者的对比明显看出，Kubernetes在当时的环境下还不能算是一个production ready的框架，只能从YARN和Mesos两者中做出选择。另外我们当时还有一个需求，就是希望可以利用Docker实现快速扩容+日志的ETL定制化，所以结合上面的表格，我们选择了一个较为均衡的方案——Apache Mesos。

YARN和Mesos都属于两级调度，具有一定的相似性，从YARN移植数据分析类的应用在可控范围内，而且Spark源生支持Mesos，数据分析这块还是有一定的功能保障的。

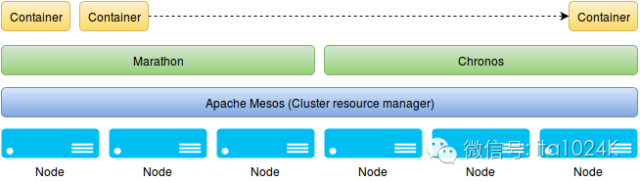


应用调度

单独一个Mesos是没任何作用的，必须搭配Framework才能达到申请资源，发布应用的目的，我们选择了Marathon和Chronos分别作为long-time-running service和crond job的调度层。同时在数据分析层面，Marathon和Chronos的覆盖面比较广（如下图），也满足我们日志这块的需求。

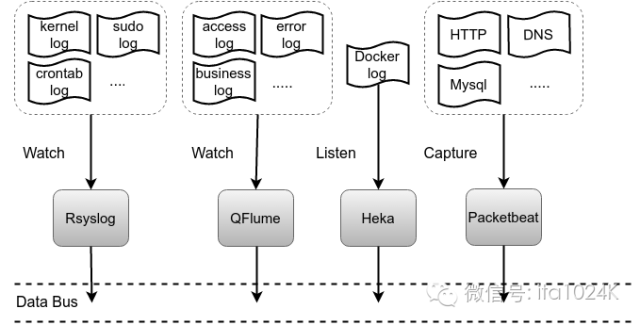


两个底层结构定下来以后，就可以考虑业务相关的技术选型了，到此为止，我们的基础平台结构就是这样，上层服务可以以container方式运行在任意节点上：



日志收集

这块的重要性仅次于底层平台，不但要稳定，还不能过多占用资源影响线上业务，同时又要保证吞吐量。



根据这几个前提，我们首先针对日志的来源做了一个划分：系统日志和业务日志：

|  |  |  |
| --- | --- | --- |
|  | 系统日志 | 业务日志 |
| 例子 | sudo.log/messages/dmsg/cron等 | access.log/dubbo/error.log等等 |
| 日志量 | 小 | 巨大 |
| 日志格式 | 固定 | 看开发心情 |
| 受众 | OPS团队/安全团队 | 业务线开发/QA |
| 收集情况 | 每台机器都要收集 | 按需决定 |
| 结论 | rsyslog | Qunar  Flume |

多数的Linux发行版都配备了rsyslog，配置简单，性能好，插件多，足以满足系统日志的收集需求，上线进需要批量推送配置并重启服务即可，运维成本最小化。

Qunar Flume是我们的技术部针对业务日志开发的一个agent，借鉴了Apache Flume的结构，同时与公司的应用中心，部署平台整合到了一起，支持日志发现，日志聚合，以及配置热发等实用功能，满足业务线按需收集日志的要求，随时开启/停止日志收集。

同时，考虑到部分应用的并发量和请求量非常大，不适合开启磁盘日志的应用，我们选择elastic的packetbeats做为补充方案，通过分流TCP数据包的方式收集相应的日志，适合用在Nginx的access日志，MySQL的请求等场景上。

日志队列

我们直接选择了Kafka，高吞吐量，高可用性，针对日志类消息的完美搭配。更重要的是，low level api搭配offset可以回放数据，尽最大可能保证消息不丢失和至少处理一次。

日志清洗

又是一个脏活累活，选择性较多，比较常见的如logstash，rsyslog，storm，spark等，前两者依靠配置，后两者则是靠编程，这里主要把logstash/storm/spark三者做个对比：

|  |  |  |  |
| --- | --- | --- | --- |
|  | logstash | storm | spark（streamimg） |
| 开发成本 | 写配置文件 | 写拓扑 | 写job |
| 学习成本 | 低，QA也可以修改 | 高，非开发人员无法使用 | 高，非开发人员无法使用 |
| 部署成本 | 单进程 | 集群模式 | 集群模式 |
| 性能 | 低（1.x主要在ruby和java内存结构多次转换上） | 高 | 高 |
| 可靠性 | 低，input自身的queue导致 | 高，ack保证 | 高，direct api + checkpoint |
| 数据聚合，如TOP N | 依赖filter，比如aggregate | 拓扑方式可以做到 | 时间窗 |
| 扩展性 | 相同配置启动新实例 | 增加worker/命令行调整并发 | 增加executor |
| 监控/指标 | 无（1.x） | 有 | 有 |

比较下来发现logstash跟Storm/Spark相比，稍逊一筹，不过可取之处是开发和学习成本，毕竟针对日志清洗这块，人力成本占大部分比重，时间主要耗费在与业务线核对格式，类型，做数据关联等等步骤上。

可是我们就3个人，无法支撑这么多日志格式的清洗工作，选择的重心就倾向于logstash，并开发相应的debug工具，由业务线的开发或QA来完成数据清理工作，从编码向配置过度，按需解析，这样不但释放了我们的精力，解析结果还100%匹配业务线的需求。

日志分析

这部分我们选择Logstash +Spark共同完成，针对单条（注意不是单行，日志经过了Qunar Flume聚合后，已经涵盖了部分上下文关系）日志的处理，使用Logstash，针对需要分析上下文/时间窗口一类的场景则选用Spark。

除此之外，我们还接管了一些Storm集群，准备在有精力的时候替换成Flink。

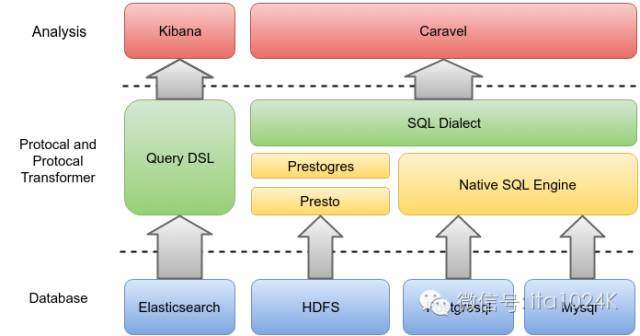
日志存储

我们针对日志的存储选择了ElasticSearch，正好搭配Kibana可以直接检索日志了，简单好用，其他的数据仍然存储在HDFS上，有少部分数据会写入Redis，MySQL对接业务。

日志展示

ElasticSearch和Logstash都上了，Kibana就别闲着了，针对日志的检索，报表等事情，Kibana能够很好的完成，美中不足就是我们使用的版本是4.1的，无法自己调整Timezone，对于某些日志的时间戳还要额外转换成UTC来满足Kibana的展示。

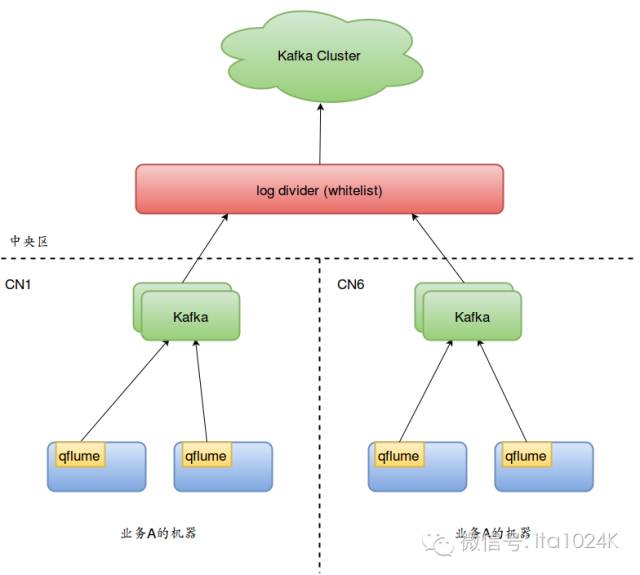
除了Kibana外，我们还缺少针对SQL的展示组件，主要是对接Hive的数据，最开始的时候我们使用Zeppelin自带的图表暂时支持下，后来利用Presto + Prestogres +Hue的方式升级了一版本，目前正在尝试Airbnb开源的Caravel对接Presto/Prestogres，支持更自由的报表展示。



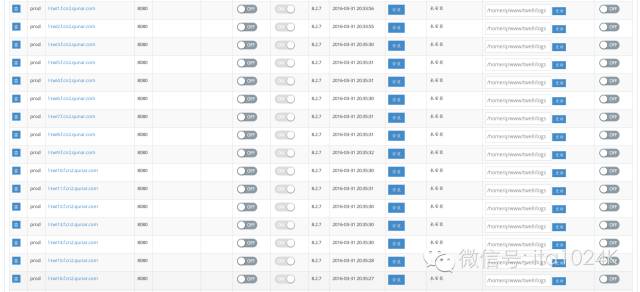
平台1.0：解决日志收集/存储/展示

整个项目在2015年5月份开始启动，首要目标就是解决一个由160台KVM组成服务发布时的无人值守功能，提供线上日志的检索和筛选，快速定位故障机器，再考虑接入更多的业务线日志，提供检索和统计的服务。

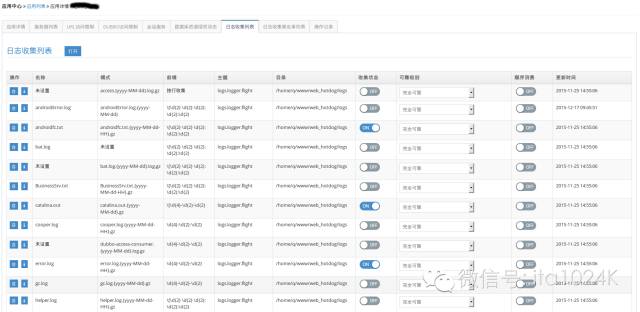
首先考虑的是解决机房间数据的可用性问题，要保证在机房间网络故障时仍然可以缓存一定时间的日志，并且自带冗余数据，我们采取的措施是在每个机房内都放置一组Kafka（0.8.1）集群，日志采取就近原则发送到同机房的Kafka内，再由程序同步到中央Kafka。



其次是qflume的推广和运维问题，我们采取与应用中心绑定的策略。应用中心是技术部开发的一套服务治理系统，已经覆盖了配置热发，监控，报警等功能，搭配qflume正合适：

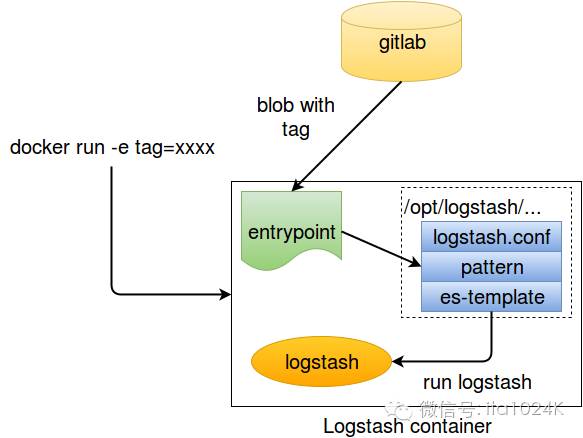


日志收集相关的配置也在应用中心控制，随时开启/关闭收集，还可以配置日志合并的策略，无需OPS更新线上配置，监控和报警也一步到位，简直是运维的好帮手：



收集端和日志队列都上线以后，我们开始着手部署Mesos（0.22.0），Marathon（0.8.0），Chronos（2.3.2），Zookeeper和ElasticSearch，使用saltstack + ansible完成。接着就开始Docker化Logstash和Kibana，还有我们提供的一些接入/发布工具：

1.     我们重新build了Logstash的镜像，采取启动后拉取配置的方式来应对日志解析规则的变更，配置文件放在Gitlab里，开放给业务线编辑，用tag区分不同release的版本。容器启动后根据传入的环境变量tag自动拉取对应配置，比如logstash.conf，自定义的pattern和elasticsearch模板，放到对应的路径再启动logstash。没有考虑一次变更一个镜像的原因是每次的变动主要是logstash.conf这个文件，为了一个文件重新build & pushimage显得有些繁琐了。



2.     Kibana我们给每一个业务线都部署了一个，通过环境变量传入app code，每个业务线的indexsettings/virtuals/dashboard都是独立的，通过Chronos定时备份到swift上。

3.     利用Openrestry开发了一个简单的七层服务发现，通过泛域名的形式将Kibana和app code关联起来（如my\_tomcat.kibana.corp.qunar.com），lua解析url拿到app code，请求MarathonREST API获取task的hostname和port，直接proxy pass过去。后续又追加了针对Marathon任务的支持（如my\_tts.marathon.corp.qunar.com）。

4.     四层的服务发现使用Bamboo + Haproxy，相比Marathon Eventbus + Etcd + Confd + Haproxy的方式，优势是工作量小，主要是配置工作，劣势是细节控制没有后者精确，无法服用，例如信息同时汇总到报警系统。

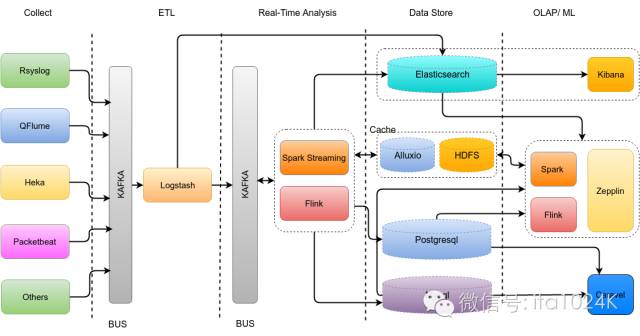
5.     Kafka的管理使用Yahoo开源的Kafka-manager，监控数据收集使用kafka-http-metrics-reporter + kafka\_metric\_2\_graphite.py，直接发送到graphite，包括了offset，topic的input/output统计，under replicate等等指标。

6.     针对日志的接入开发了一个发布系统，串接Jenkins和监控系统，调用Marathon的API发布Logstash和Kibana，同时创建响应的报警，提交定时任务备份settings等工作。

这一阶段的集群的规模较小，大约用掉了30-40台机器，随后开始向业务线推广使用，2015年9月，每日处理量超过了40亿条。

平台2.0：实时日志分发

在1.0打下的基础上，我们把目标升级成了数据分发平台，除了保证日志收集存储外，还要联通线上日志与各个业务线的数据组和分析系统，降低独自获取实时日志的成本，同时扩大数据的复用程度，较少重复解析造成的资源浪费。



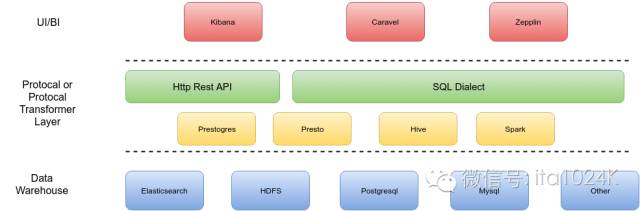
我们工作的重心开始瞄准了Spark（1.6.2），以及开放Kafka/Logstash/ElasticSearch的访问权限，同时调研了Presto/Zeppelin/Alluxio（原名Tachyon）三个数据框架，提供从测试，发布，运行，缓存加速等一系列的功能。

在日志收集方面，我们引入了Heka和Packetbeats，针对容器日志和Nginx一类的高QPS服务（ElasticSearch的HTTPREST API访问监控也是通过Packetbeats完成）。也允许业务线向Kafka broker写入数据，提高数据流通效率。

ETL层仍然首选Logstash，所有数据均经过Logstash的处理后写入ElasticSearch或Kafka，留给Kibana和Spark使用。

实时处理从Storm迁移至Spark（Flink调研中），Block和Checkpoint默认存储在Alluxio内，计算结果则通过编码控制写入HDFS/RBDMS/NoSQL等系统备用。

OLAP以Kianba/Zeppelin（需要编程）/Caravel为主，辅以Presto/Prestoges/Hue完成简单报表/聚合查询等工作。



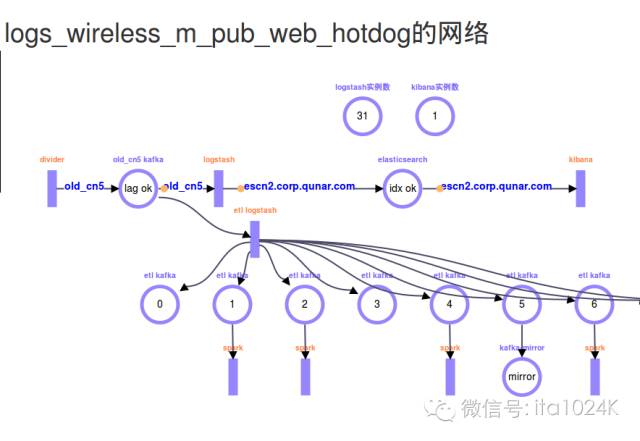
在Spark on Mesos的实施上遇到了不少的问题，主要是整合部分的代码逻辑比较简单，不能很好的匹配生产环境的调度策略，扩容也不方便（需要重发streaming），重写了部分代码后才算是较为方便的在Mesos集群上调度driver和executor。我们没有使用Docker运行Spark任务，而是选择了Mesos container（cgroup），通过tar包的方式发布任务。

由于增加了许多服务在Mesos（0.25.0）上，资源分配成了一个比较严重的问题，需要对cpu/mem调整超售比，适当提高下利用率，同时还要针对不同的Framework做静态资源分配，比如Spark的cpu上限为物理核的一半，尽量散步在集群的各个节点上，防止堆积到某个节点导致处理缓慢，以下是当时我们采取的一个资源配比策略：

MESOS\_resources="cpus(logstash):{{num\_cpus}}"  
MESOS\_resources="${MESOS\_resources};cpus(common):4"  
MESOS\_resources="${MESOS\_resources};cpus(kibana):4"  
MESOS\_resources="${MESOS\_resources};cpus(ops):4"  
MESOS\_resources="${MESOS\_resources};cpus(spark):{{num\_cpus/2}}"  
MESOS\_resources="${MESOS\_resources};cpus(tachyon):4"  
MESOS\_resources="${MESOS\_resources};cpus(others):{{num\_cpus/2}}"  
MESOS\_resources="${MESOS\_resources};cpus(test):8"  
MESOS\_resources="${MESOS\_resources};ports(\*):[8000-32000]"

同时Marathon在日益增长的应用面前也开始出现了效率问题，我们不得不按照用途重新规划应用，并拆分成多个Marathon框架，控制不同任务的资源上限。

再优化了基础平台后，数据的日处理量增长到了每日100亿的规模，大量的数据在平台内流通，带来了一个新的问题，一个没接触过系统的人如何能方便的获取想要的数据，我们整理了平台内的数据流的信息，绘制了相应的数据拓扑，对外提供查询。



平台3.0：stateful service & cache

这个阶段正在实施中，主要是针对ElasticSearch和Alluxio服务的平台化，借助Mesos的持久化卷和动态预留功能，提供stateful service的部署。

我们最先要解决的是ElasticSearch服务化，目前许多业务线都开始使用ElasticSearch，申请资源和运维是都是独自在做，形成不了统一的运维标准，经验也不容易分享。对于我们OPS也希望统一管理底层的资源，减少业务线的压力。

我们基于Mesos（0.28.0）+Marathon（1.1.1）重新构建了一套系统，部署相互隔离的ElasticSearch集群，指定3个节点的node tag为master（不同rack）,其余节点标记成data，并配合groupby rack保证物理资源的冗余。Master和Datanode的发现通过Bamboo + Haproxy实现，ACL考虑search-guard。

不推荐使用ElasticSearchon Mesos这个项目，不支持持久化卷和动态预留，贡献也不太活跃，但是测试系统的话可以考虑下，用完就回收。

另外一个要解决的问题是Alluxio的服务化，把计算节点的磁盘资源利用起来，作为一个临时文件的DFS，同时提供给其他系统作为block cache的一个备选方案。

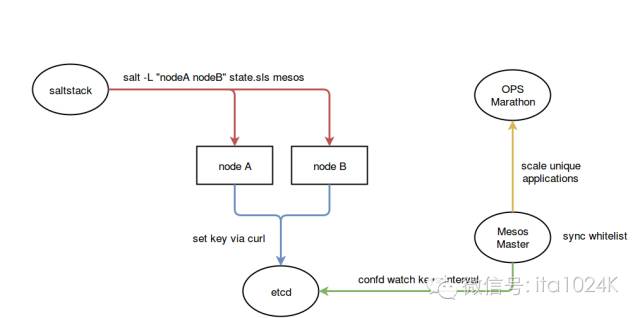
平台填坑指南

平台演化到今天，经历了不少的难题，从最初的几台机器到现在接近两百台的规模，坑没少跳。除了能力暂时还达不到无法修改（比如Mesos），基本还都可以搞定。借着今天的机会分享下我们填坑和优化的一些经验。

Mesos

在maintenance接口出来以前，白名单就是运维的利器，升级/维护/人肉调度就靠它了，我们利用saltstack+ etcd + confd + 白名单做了一个监控基础服务扩容的daemon。新机器升级好内核后利用saltstack部署Mesos和预热，并curl etcd的服务注册自身，confd监控到变化后生成白名单，并调用Marathon的REST API扩容相应的服务。

监控则通过Logstash的http poller input请求Mesos的API获取相应的数据，配合json filter筛选数据后存入监控系统内。



还遇到一个未解决的问题（0.25），就是Spark的framework有一定概率残留在某些Slave上，消耗资源，每个残留进程0.1cpu/32mb内存，积累起来浪费还是很可观的，社区里暂时没有发现相应的解释。

Docker

主要说一下daemon内存的问题，我们的Docker使用1.7.1版本，之前对log-driver没太关注，采取了默认的json-log配置，后来一次logstash的filter报错打印了大量的日志到stderr，导致daemon内存一直增长，最后启动容器都申请不到内存，解决办法就是log-driver=none，不在通过daemon中转日志数据，直接通过Mesos docker executor记录到sandbox里。

ElasticSearch

我们存放日志的ElasticSearch机器有50台左右，最初是SAS盘+raid10，线上跑了一段时间发现IO并不是瓶颈，就更换成了容量更大的SATA盘，单机容量40多T，足够支撑存储的需要。

首先遇到的问题是fielddatacache不释放的问题，官方文档是不建议设置fielddata的过期时间，主要是相应的数据结构从内存中移除代价较高，但是结合我们实际的使用情况，我们将fielddata失效时间调整到5min。

然后是最头疼的问题，datanode的fullgc，再调整了cache比例，失效时间后，仍然会发生fullgc，对比了监控后发现此时的fullgc主要和merge相关，仔细定位后发现是由于shard分布不均匀导致的，修改了total\_shards\_per\_node=2后merge明显引起的fullgc明显下降了很多。

最后是写入QPS不稳定的问题，这个问题在日志处理量越多时越明显，在data node的日志上我们发现了大量的“now throttlingindexing”提示，考虑到日志的ElasticSearch并非100%都需要写入后立即能查询到，我们就调整了indices.store.throttle.type=none，防止因为merge限流导致的写入变慢，同时又加大了indices.store.throttle.max\_bytes\_per\_sec=100mb。



ElasticSearch的监控我们选择es2graphite.py这个脚本，配合公司的监控系统watcher，可以满足日常的运维需要。

Logstash

最主要的是问题监控不足，吞吐量降低时不知道卡在哪个环节，针对这个问题我们修改了Logstash的部分代码，针对input/filter/output埋点，统计每条日志的处理时延，同时定期获取两个queue上的wait thread数量已确定哪个部分托慢了整个pipeline。

Spark on Mesos

遇到的问题非常多，小到临时文件的存放，大到checkpoint失效，算是从头到尾踩了一遍。1.6之前的Spark对于Mesos的支持并不是很好，比如认证和调度约束都没有做，需要自己写patch。

通过mesosdispatcher提交的job，一些配置信息，环境变量带不过去，看了代码才发现环境变量是通过文件传递的，简单的解决办法就是把需要的信息写到spark-env.sh内。

1.5之前的临时文件不能放到Mesos的sandbox里，无法利用Mesos的GC机制释放磁盘空间，1.5开始通过spark.local.dir和java.io.tmp配置写入sandbox。

Spark on Mesos默认不支持多executor，需要自己patch对应代码，或者利用Marathon启动executor，我们更推荐后者，针对Streaming任务扩容更方便。

Streaming的任务以Kafka作为数据源使用时，推荐使用direct api，通过编码方式控制offset的提交，同时每个executor都有自己的consumer，consumer的并发粒度远远好于reciver的方式，消息可靠性也比reciver高，美中不足是没有主动设置kafka partition的owner，需要自己编码实现。

另外，checkpoint记录在Alluxio（0.8.2）上会出现“fileis not complete”的情况，这是client实现的问题，需要升级到1.0.0+版本解决，而HDFS无此问题。

最后一个问题是Streaming通过checkpoint恢复后丢失了一些依赖包（表现形式多为ClassNotFound），这是因为在Spark on Mesos在启动Driver后，相应的jar包放置在了对应的sandbox内，Driver恢复后路径已经变化了，新的sandbox内没有对应的jar包，较为简单的解决办法如下：

sparkConf.setJars(Array(s"http://stor.corp.qunar.com/qae/spark/$dep-$appCode-jar-with-dependencies.jar"))

把Jar包放在FTP服务器上，每次Driver启动都去FTP同步jar包恢复执行情况。

以上就是今晚的分享，谢谢大家。

Q&A

问：将所有的服务运行在容器之上的初衷是什么？

徐磊：混布+资源控制+方便部署，是优先考虑的。

追问： 这样做了投入产出 觉得如何？

徐磊： 针对配置型的，比如logstash这些组件，以前用ansible或者saltstack去替换配置，重启服务，现在这些的工作由docker的entrypoint和mesos slave来做。投入就是不需要开发针对ansible/saltstack的对接系统了，直接点鼠标就好了。

问：日志的搜集是实时还是固定时间啊？

徐磊：实时时候，类似tail，qflume来做的。

问：日志按什么规则汇总的？   spark多久计算一次，从进入kafka到spark结果延时大概多少？

徐磊：这个默认是按照行(\n)收集，也可以根据一些规则，比如java的stacktrace，可以按照日志时间的前缀收集，自动merge。 spark的batch时间最长有10min的，最短的是200ms，从收集开始算，到进入kafka的时间不超过150ms。

问：是否可以将flume完全替代logstash？

徐磊：收集端可以，而且还有beats系列可以用，没必要用logstash收集。

追问：那就是可以完全替换楼？

徐磊：完全可以flume。

追问：有考虑过 solr 来做搜索吗？

徐磊：没，目前跟定了es。

问：logstash 的配置文档我感觉 很傻瓜化 ，里面的核心的内部参数都没有办法配置，比如类似flume的channel 没有办法配置？

徐磊：对。。。这是配置类应用的劣势，如果有精力开发的话，效果肯定要比logstash好多了，比如我们自己的qflume。

问：利用sprak 和storm在，在日志实时监控上有什么具体功能？

徐磊：比如search rank，商品推荐，风控等等，以前通过系统间埋点调用换成了消费日志的方式。

问：这套结构是否尝试过用过分析Nginx日志用于流量控制，秒级日志聚合统计效果如何？

徐磊：http server + packetbeats相对好一点，但是目前我们没有把access log直接对接流控这部分，日志聚合依赖es的功能，我们目前是通过Kibana+定时刷新来看，如果想更加灵敏，可以用elasticsearch的一个特性（叫p什么来着，忘记名字了），提前构建search，es会根据search的匹配程度自动推送数据出来，类似hook，这种的延时性要好于通过kibana来看。

问：opsdev  你们都用什么语言？

徐磊：Python为主，Java和Scala为辅，少量的golang。

问：实时收集，会对生产系统产生影响吗？

徐磊：会有影响，我们许多应用都是运行在4core/4G的kvm上，收集agent如果资源占用过大会影响到线上服务，这也是我们的技术部重新开发qflume的一个目的，降低agent在资源的损耗，但是不能说100%避免极端情况，比如我们有个服务的日志会把request/response的数据记录下来，单行日志有过夸张的5mb，这种情况在producer queue的时候容易oom。

问：日志不做汇聚和合并   对spark计算有影响吧，对hadoop也不好    请问这块你们怎么做的？

徐磊：给spark的日志会根据要求提前把数据进行初步的处理，我们一般会采用ruby代码的方式来做这些事情，但是跑在logstash里，相当于是数据通过logstash再做一次处理，写回kafka，再交给spark做，能满足一部分的情况。有些时候数据实在太散了我们也不好用logstash做，只能依靠spark/storm来了。

问：当时1.0 - 3.0这些阶段中，做这套系统的人员有多少人？这中间经历了多少时间？

徐磊：1.0两个人，主要是我和我的leader一起做，2.0维持在3个人，现在3.0 4个人。

问：点击流日志，大家用什么收集？

徐磊：收集方式不太一样，如果是公共数据要用的，一般都是分析入口应用的日志，缺点就是费时费力，格式一遍就歇菜了，我们公司有一个更方便的东西，可以直接把app的点击流发送回来，比直接分析日志的方式节省人力。

问：太散的话  貌似对spark和hadoop性能有影响    现在你们是通过spark处理么，以做前期处理么  flume收集的话，我没找到方案，收集上做合并也有局限性？

徐磊：如果是担心频繁写入HDFS的话，我们使用了Alluxio，Spark数据直接写入Alluxio里面，异步写到underfs，这种方式不会因为Spark上层的mirco batch太频繁导致的请求过大，同时异步写还能保证批量刷新。

问：关于application日志和access日志上下文聚合有什么最佳实践没？

徐磊：单纯的access日志的话，我们目前的方法是解析完直接写入es了，通过es提供aggregate来做聚合，如果要关联其他日志的话，我们暂时还没开始做，这个也是我们3.0要干的事情，所以我们也是在摸索中。

**◆  ◆  ◆**

**关于ITA1024**

互联网技术联盟（ITA1024）是由京东、美团点评、小米、滴滴、携程、网易、搜狐、乐视、当当、途牛、饿了么、58、猎豹等TOP100的互联网服务和七牛、青云、听云、DaoCloud、UCloud、有云等技术服务联合发起的国内最大的企业间技术交流组织，专注于互联网+技术与创新。  
  
联盟精心组织的1024系列技术峰会，由每周一期的线上万人课堂和每月一次的技术大会组成。每月一个技术主题，由联盟成员企业推荐的国内一流技术专家联手打造，分享如何通过一线技术应用案例和最佳实践，支撑和驱动业务成长。  
  
联盟还通过官方网站（www.ita1024.com），官方微信公众（ita1024k），ITA1024技术月刊等多种形式，将精品技术内容精准推送给细分领域专业人群。