**经典大数据架构案例：酷狗音乐的大数据平台重构**

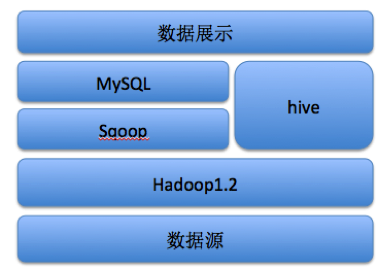
眨眼就新的一年了，时间过的真快，趁这段时间一直在写总结的机会，也总结下上一年的工作经验，避免重复踩坑。酷狗音乐大数据平台重构整整经历了一年时间，大头的行为流水数据迁移到新平台稳定运行，在这过程中填过坑，挖过坑，为后续业务的实时计算需求打下了很好的基础。在此感谢酷狗团队成员的不懈努力，大部分从开始只知道大数据这个概念，到现在成为团队的技术支柱，感到很欣慰。

从重构原因，技术架构，踩过的坑，后续持续改进四个方面来描述酷狗音乐大数据平台重构的过程，在此抛砖引玉，这次的内容与6月份在高可用架构群分享的大数据技术实践的有点不同，技术架构做了些调整。

其实大数据平台是一个庞大的系统工程，整个建设周期很长，涉及的生态链很长(包括：数据采集、接入，清洗、存储计算、数据挖掘，可视化等环节，每个环节都可以当做一个复杂的系统来建设)，风险也很大。

**一、重构原因**

在讲重构原因前，先介绍下原有的大数据平台架构，如下图：



从上图可知，主要基于Hadoop1.x+hive做离线计算(T+1)，基于大数据平台的数据采集、数据接入、数据清洗、作业调度、平台监控几个环节存在的一些问题来列举下。

数据采集：

数据收集接口众多，且数据格式混乱，基本每个业务都有自己的上报接口

存在较大的重复开发成本

不能汇总上报，消耗客户端资源，以及网络流量

每个接口收集数据项和格式不统一，加大后期数据统计分析难度

各个接口实现质量并不高，存在被刷，泄密等风险

数据接入:

通过rsync同步文件，很难满足实时流计算的需求

接入数据出现异常后，很难排查及定位问题，需要很高的人力成本排查

业务系统数据通过Kettle每天全量同步到数据中心，同步时间长，导致依赖的作业经常会有延时现象

数据清洗：

ETL集中在作业计算前进行处理

存在重复清洗

作业调度：

大部分作业通过crontab调度，作业多了后不利于管理

经常出现作业调度冲突

平台监控：

只有硬件与操作系统级监控

数据平台方面的监控等于空白

基于以上问题，结合在大数据中，数据的时效性越高，数据越有价值(如：实时个性化推荐系统，RTB系统，实时预警系统等)的理念，因此，开始大重构数据平台架构。

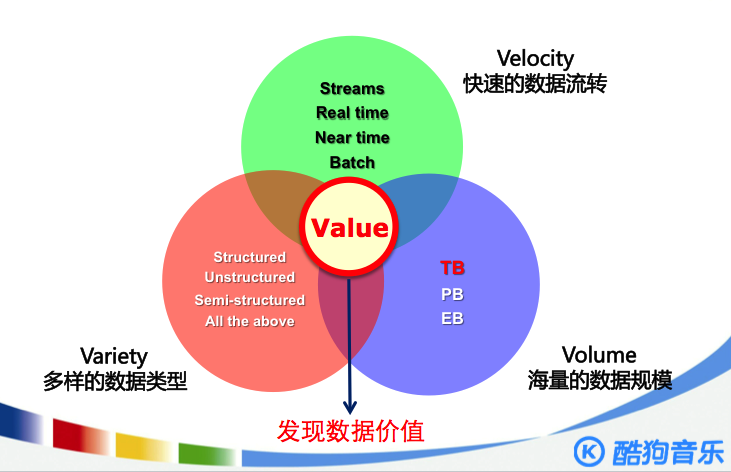
**二、新一代大数据技术架构**

在讲新一代大数据技术架构前，先讲下大数据特征与大数据技术要解决的问题。

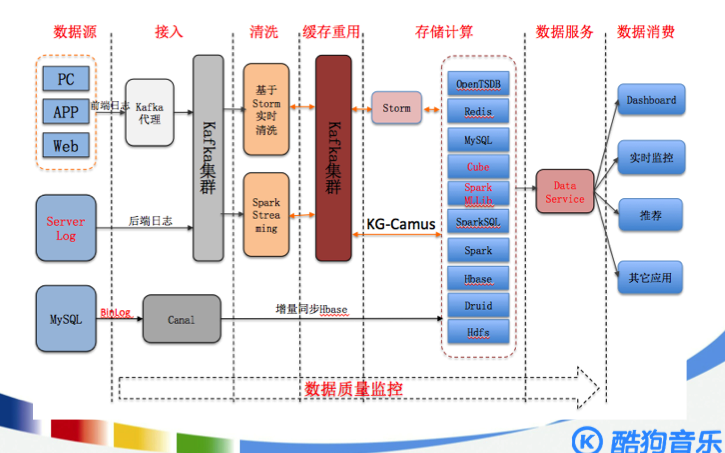
**1.大数据特征：“大量化(Volume)、多样化(Variety)、快速化(Velocity)、价值密度低（Value）”就是“大数据”显著的4V特征，或者说，只有具备这些特点的数据，才是大数据。**



2.大数据技术要解决的问题：大数据技术被设计用于在成本可承受的条件下，通过非常快速（velocity）地采集、发现和分析，从大量（volumes）、多类别（variety）的数据中提取价值（value），将是IT领域新一代的技术与架构。

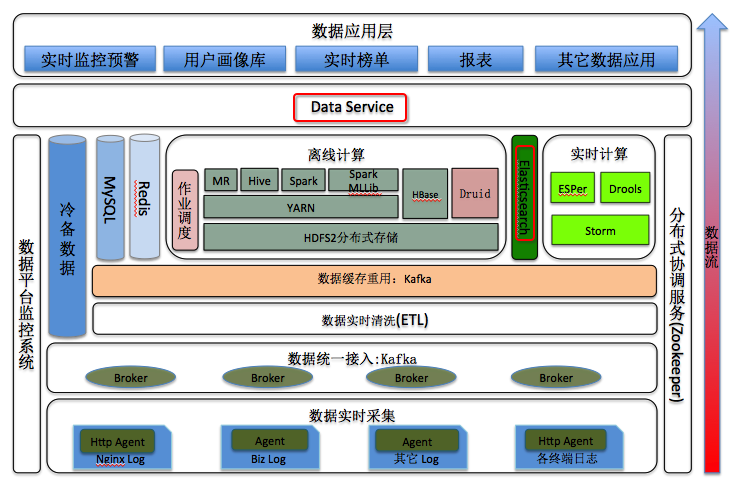


介绍了大数据的特性及大数据技术要解决的问题，我们先看看新一代大数据技术架构的数据流架构图：



从这张图中，可以了解到大数据处理过程可以分为数据源、数据接入、数据清洗、数据缓存、存储计算、数据服务、数据消费等环节，每个环节都有具有高可用性、可扩展性等特性，都为下一个节点更好的服务打下基础。整个数据流过程都被数据质量监控系统监控，数据异常自动预警、告警。

新一代大数据整体技术架构如图：



将大数据计算分为实时计算与离线计算，在整个集群中，奔着能实时计算的，一定走实时计算流处理，通过实时计算流来提高数据的时效性及数据价值，同时减轻集群的资源使用率集中现象。

整体架构从下往上解释下每层的作用：

数据实时采集：

主要用于数据源采集服务，从数据流架构图中，可以知道，数据源分为前端日志，服务端日志，业务系统数据。下面讲解数据是怎么采集接入的。

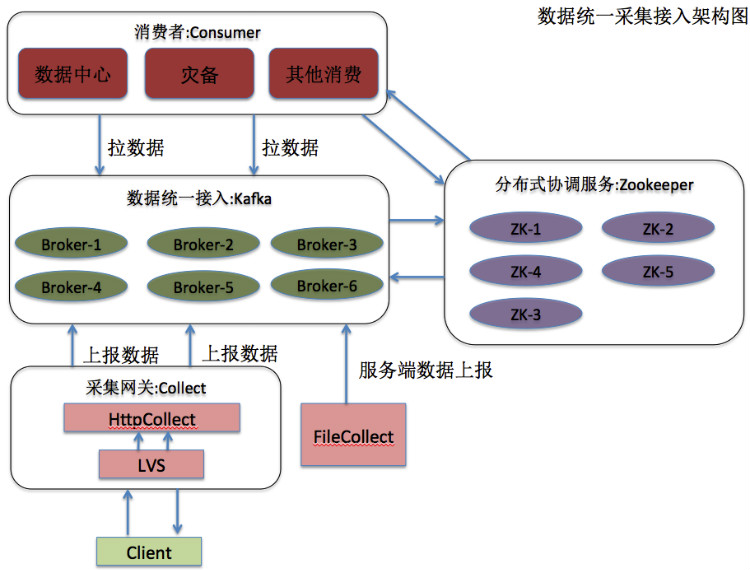
a.前端日志采集接入：

前端日志采集要求实时，可靠性，高可用性等特性。技术选型时，对开源的数据采集工具flume,scribe,chukwa测试对比，发现基本满足不了我们的业务场景需求。所以，选择基于kafka开发一套数据采集网关，来完成数据采集需求。数据采集网关的开发过程中走了一些弯路，最后采用nginx+lua开发，基于lua实现了kafka生产者协议。有兴趣同学可以去Github上看看，另一同事实现的，现在在github上比较活跃，被一些互联网公司应用于线上环境了。

b.后端日志采集接入：

FileCollect,考虑到很多线上环境的环境变量不能改动，为减少侵入式，目前是采用Go语言实现文件采集，年后也准备重构这块。

前端，服务端的数据采集整体架构如下图：



c.业务数据接入

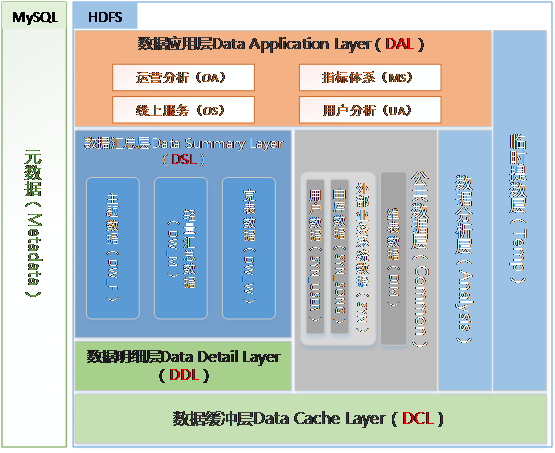
利用Canal通过MySQL的binlog机制实时同步业务增量数据。

数据统一接入：为了后面数据流环节的处理规范，所有的数据接入数据中心，必须通过数据采集网关转换统一上报给Kafka集群，避免后端多种接入方式的处理问题。

数据实时清洗(ETL)：为了减轻存储计算集群的资源压力及数据可重用性角度考虑，把数据解压、解密、转义，部分简单的补全，异常数据处理等工作前移到数据流中处理，为后面环节的数据重用打下扎实的基础(实时计算与离线计算)。

数据缓存重用：为了避免大量数据流(400+亿条/天)写入HDFS，导致HDFS客户端不稳定现象及数据实时性考虑，把经过数据实时清洗后的数据重新写入Kafka并保留一定周期，离线计算(批处理)通过KG-Camus拉到HDFS(通过作业调度系统配置相应的作业计划)，实时计算基于Storm/JStorm直接从Kafka消费，有很完美的解决方案storm-kafka组件。

离线计算(批处理)：通过spark，spark SQL实现，整体性能比hive提高5—10倍，hive脚本都在转换为Spark/Spark SQL；部分复杂的作业还是通过Hive/Spark的方式实现。在离线计算中大部分公司都会涉及到数据仓库的问题，酷狗音乐也不例外，也有数据仓库的概念，只是我们在做存储分层设计时弱化了数据仓库概念。数据存储分层模型如下图：



大数据平台数据存储模型分为：数据缓冲层Data Cache Layer（DCL）、数据明细层Data Detail Layer（DDL）、公共数据层（Common）、数据汇总层Data Summary Layer（DSL）、数据应用层Data Application Layer（DAL）、数据分析层（Analysis）、临时提数层（Temp）。

1）数据缓冲层(DCL)：存储业务系统或者客户端上报的，经过解码、清洗、转换后的原始数据，为数据过滤做准备。

2)数据明细层（DDL）：存储接口缓冲层数据经过过滤后的明细数据。

3）公共数据层（Common）：主要存储维表数据与外部业务系统数据。

4）数据汇总层（DSL）：存储对明细数据，按业务主题，与公共数据层数据进行管理后的用户行为主题数据、用户行为宽表数据、轻量汇总数据等。为数据应用层统计计算提供基础数据。数据汇总层的数据永久保存在集群中。

5）数据应用层（DAL）：存储运营分析（Operations Analysis ）、指标体系（Metrics System）、线上服务（Online Service）与用户分析（User Analysis）等。需要对外输出的数据都存储在这一层。主要基于热数据部分对外提供服务，通过一定周期的数据还需要到DSL层装载查询。

6）数据分析层（Analysis）：存储对数据明细层、公共数据层、数据汇总层关联后经过算法计算的、为推荐、广告、榜单等数据挖掘需求提供中间结果的数据。

7）临时提数层（Temp）：存储临时提数、数据质量校验等生产的临时数据。

实时计算：基于Storm/JStorm，Drools,Esper。主要应用于实时监控系统、APM、数据实时清洗平台、实时DAU统计等。

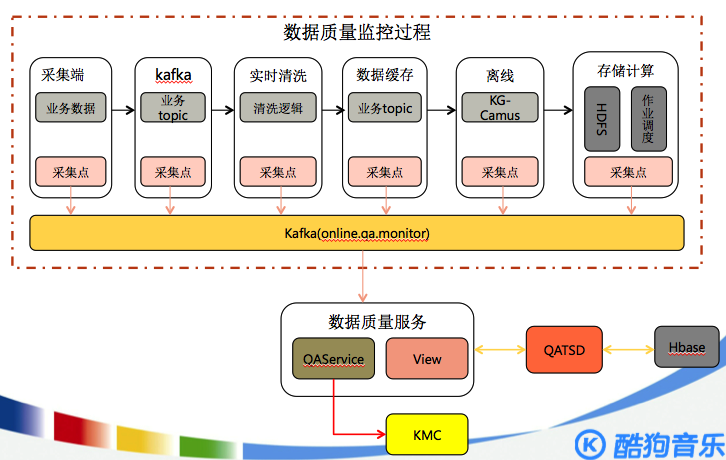
HBase/MySQL：用于实时计算，离线计算结果存储服务。

Redis：用于中间计算结果存储或字典数据等。

Elasticsearch：用于明细数据实时查询及HBase的二级索引存储(这块目前在数据中心还没有大规模使用，有兴趣的同学可以加入我们一起玩ES)。

Druid：目前用于支持大数据集的快速即席查询(ad-hoc)。

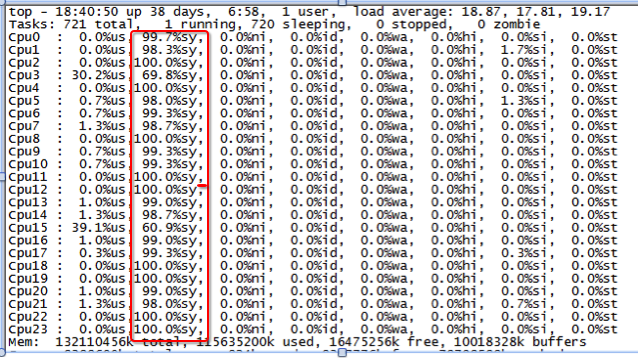
数据平台监控系统：数据平台监控系统包括基础平台监控系统与数据质量监控系统，数据平台监控系统分为2大方向，宏观层面和微观层面。宏观角度的理解就是进程级别,拓扑结构级别,拿Hadoop举例，如：DataNode，NameNode，JournalNode，ResourceManager，NodeManager，主要就是这5大组件，通过分析这些节点上的监控数据，一般你能够定位到慢节点，可能某台机器的网络出问题了，或者说某台机器执行的时间总是大于正常机器等等这样类似的问题。刚刚说的另一个监控方向，就是微观层面，就是细粒度化的监控，基于user用户级别，基于单个job，单个task级别的监控，像这类监控指标就是另一大方向，这类的监控指标在实际的使用场景中特别重要，一旦你的集群资源是开放给外面的用户使用，用户本身不了解你的这套机制原理，很容易会乱申请资源，造成严重拖垮集群整体运作效率的事情，所以这类监控的指标就是为了防止这样的事情发生。目前我们主要实现了宏观层面的监控。如：数据质量监控系统实现方案如下。

**三、大数据平台重构过程中踩过****的坑**

我们在大数据平台重构过程中踩过的坑，大致可以分为操作系统、架构设计、开源组件三类，下面主要列举些比较典型的，花时间比较长的问题。

**1、操作系统级的坑**

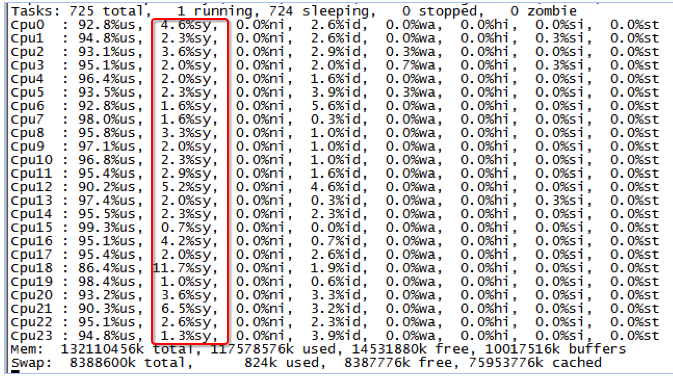
Hadoop的I/O性能很大程度上依赖于Linux本地文件系统的读写性能。Linux中有多种文件系统可供选择，比如ext3和ext4，不同的文件系统性能有一定的差别。我们主要想利用ext4文件系统的特性，由于之前的操作系统都是CentOS5.9不支持ext4文件格式，所以考虑操作系统升级为CentOS6.3版本，部署Hadoop集群后，作业一启动，就出现CPU内核过高的问题。如下图：



经过很长时间的测试验证，发现CentOS6优化了内存申请的效率，引入了THP的特性，而Hadoop是高密集型内存运算系统，这个改动给hadoop带来了副作用。通过以下内核参数优化关闭系统THP特性，CPU内核使用率马上下降，如下图:

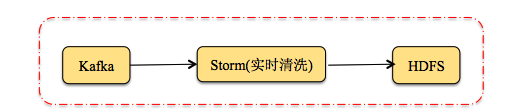
echo never > /sys/kernel/mm/redhat\_transparent\_hugepage/enabled

echo never > /sys/kernel/mm/redhat\_transparent\_hugepage/defrag



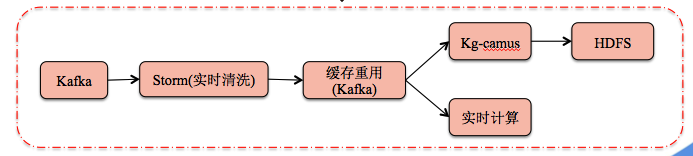
**2、架构设计的坑**

最初的数据流架构是数据采集网关把数据上报给Kafka，再由数据实时清洗平台(ETL)做预处理后直接实时写入HDFS，如下图：



此架构，需要维持HDFS Client的长连接，由于网络等各种原因导致Storm实时写入HDFS经常不稳定，隔三差五的出现数据异常，使后面的计算结果异常不断，当时尝试过很多种手段去优化，如：保证长连接、连接断后重试机制、调整HDFS服务端参数等，都解决的不是彻底。

每天异常不断，旧异常没解决，新异常又来了，在压力山大的情况下，考虑从架构角度调整，不能只从具体的技术点去优化了，在做架构调整时，考虑到我们架构重构的初衷，提高数据的实时性，尽量让计算任务实时化，但重构过程中要考虑现有业务的过渡，所以架构必须支持实时与离线的需求，结合这些需求，在数据实时清洗平台(ETL)后加了一层数据缓存重用层(kafka)，也就是经过数据实时清洗平台后的数据还是写入kafka集群，由于kafka支持重复消费，所以同一份数据可以既满足实时计算也满足离线计算，从上面的整体技术架构也可以看出，如下图：



KG-Camus组件也是基于架构调整后，重新实现了一套离线消费Kafka集群数据的组件，此组件是参考LinkedIn的Camus实现的。此方式，使数据消费模式由原来的推方式改为拉模式了，不用维持HDFS Client的长连接等功能了，直接由作业调度系统每隔时间去拉一次数据，不同的业务可以设置不同的时间间隔，从此架构调整上线后，基本没有类似的异常出现了。

这个坑，是我自己给自己挖的，导致我们的重构计划延期2个月，主要原因是由最初技术预研究测试不充分所导致。

**3、开源组件的坑**

由于整个数据平台涉及到的开源组件很多，踩过的坑也是十个手指数不过来。

1）、当我们的行为数据全量接入到Kafka集群(几百亿/天)，数据采集网卡出现大量连接超时现象，但万兆网卡进出流量使用率并不是很高，只有几百Mbit/s，经过大量的测试排查后，调整以下参数，就是顺利解决了此问题。调整参数后网卡流量如下图：

a)、num.network.threads(网络处理线程数)值应该比cpu数略大

b)、num.io.threads(接收网络线程请求并处理线程数)值提高为cpu数两倍

2）、在hive0.14 版本中，利用函数ROW\_NUMBER() OVER对数据进行数据处理后，导致大量的作业出现延时很大的现象，经异常排查后，发现在数据记录数没变的情况，数据的存储容量扩大到原来的5倍左右，导致MapReduce执行很慢造成的。改为自己实现类似的函数后，解决了容量扩大为原来几倍的现象。说到这里，也在此请教读到此处的读者一个问题，在海量数据去重中采用什么算法或组件进行比较合适，既能高性能又能高准确性，有好的建议或解决方案可以加happyjim2010微信私我。

3）、在业务实时监控系统中，用OpenTSDB与实时计算系统（storm）结合，用于聚合并存储实时metric数据。在这种实现中，通常需要在实时计算部分使用一个时间窗口（window），用于聚合实时数据，然后将聚合结果写入tsdb。但是，由于在实际情况中，实时数据在采集、上报阶段可能会存在延时，而导致tsdb写入的数据不准确。针对这个问题，我们做了一个改进，在原有tsdb写入api的基础上，增加了一个原子加api。这样，延迟到来的数据会被叠加到之前写入的数据之上，实时的准确性由于不可避免的原因（采集、上报阶段）产生了延迟，到最终的准确性也可以得到保证。另外，添加了这个改进之后，实时计算端的时间窗口就不需要因为考虑延迟问题设置得比较大，这样既节省了内存的消耗，也提高了实时性。

**四、后续持续改进**

数据存储(分布式内存文件系统(Tachyon)、数据多介质分层存储、数据列式存储)、即席查询(OLAP)、资源隔离、数据安全、平台微观层面监控、数据对外服务等。