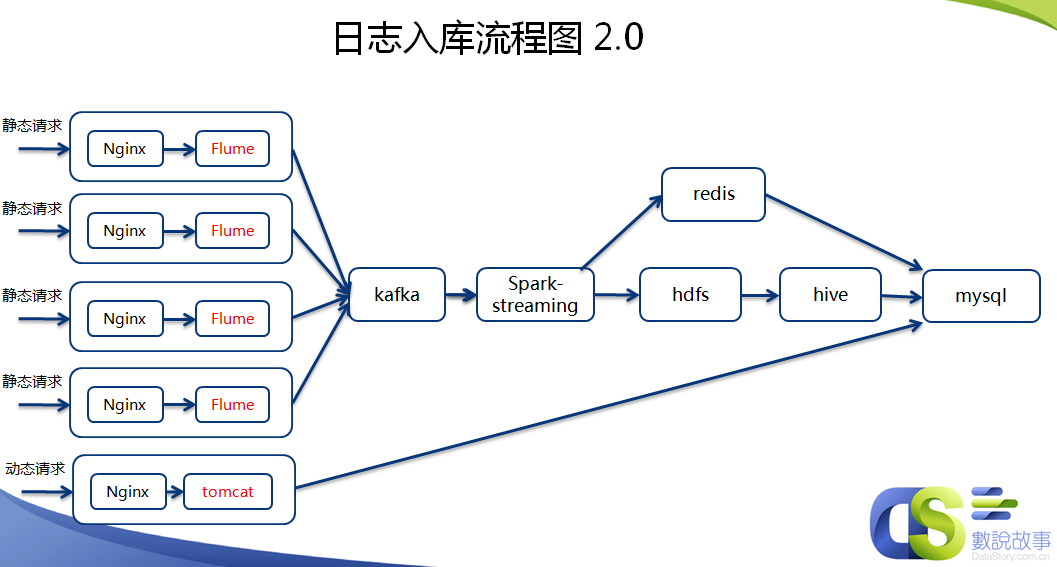
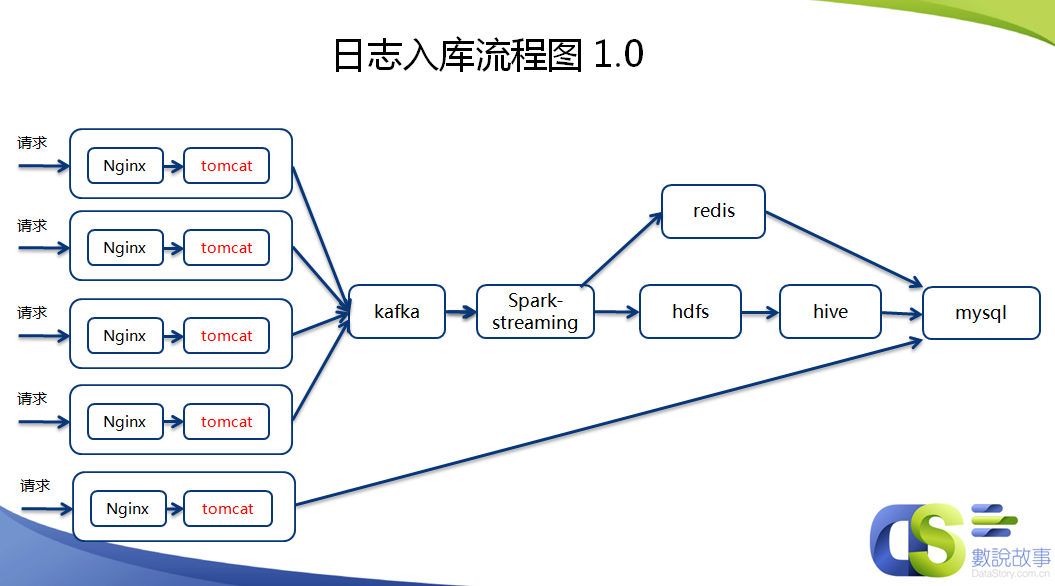
**数说传播-日志和流式处理技术文档**

数说传播是一个实时的日志接收和流式处理的web服务，最高峰时间会有每秒1W个请求发送到tongji.datastory.com.cn这个域名。入库后要提供特定的统计结果返回给用户。后面将分为三个步骤介绍整个流程：**请求接收，ETL，存储与统计**

**一 、请求接收**

入库流程图1.0版本和2.0版本如下：



简单介绍下请求的来源，请求都是来自添加了我们的监测代码的H5，请求可以分为两大类，一类是不要求实时性，可以延迟处理；另一类是实时请求，经过处理后返回特定值。实时请求的量占总请求的量不到5%。非实时类请求可以进一步细分为visit.gif（页面被访问时发出），repost.gif（页面被分享时发出），unload.gif（离开页面时发出），button.gif（点击页面某个按钮时发出）。请求中最主要的字段是taskId和cookieId，taskId用来唯一识别H5，同一个taskId，表明是从同一H5发出来的；cookieId用来唯一标识用户，同一个cookieId，表明是从同一个用户发出来的。

1.0版本中，Nginx把所有收到的请求都转给Tomcat，Tomcat实时的处理所有类型的请求。Tomcat是比较重web服务器，在我们线上运行情况来看，一台Tomcat每秒200个请求的时候，就会出现严重延迟。用它处理对实时性要求不高的请求，非常的不划算。

2.0版本中，我们把请求分为上述两大类，对实时性要求不高的请求，我们称之为静态请求。Nginx收到静态请求不再转发到Tomcat，而是直接返回，请求的所有内容会写入accessLog。静态和动态请求的区分，可以在Nginx直接配置。Nginx在处理静态请求的性能非常好，压测可以达到每秒1W个请求无明显延迟。

所有请求写到accessLog后，用Flume把accessLog中的静态请求转发到Kafka。Flume是Cloudera提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统。只需写一个配置文件即可非常高效的转发accessLog，不会把请求阻塞住。

accesslog还起到一个短期备份的作用，如果后面的流程出错，可以把当天入库的数据清理，把accesslog的log重新放一遍。如果接收到请求，却没有写accesslog，那就没法挽回数据了。accessLog重放我们至少搞过5次，数据备份很重要！

遇到的坑：accesslog的量太大，把分区的磁盘写满了,高峰时间一台机一小时5G，导致新的请求无法写accesslog。

解决方案：定期把旧accesslog搬走。一定要做好磁盘监控。

**二、ETL**

Kafka 是一种高吞吐量的分布式发布订阅消息系统。1.0版本的Tomcat和2.0版本Flume转发的静态请求都被放Kafka队列中，由SparkStreaming来进行批处理。

1）消息队列对比

主要考虑是在redis、kafka、rabbitMQ三个中选择，其中Kafka和rabbitMQ是分布式消息队列。

redis: 做轻量级消息缓存队列是不错的选择，基于内存，写入速度略逊于kafka和rabbitMQ（受限于服务端的单线程写入），但是查询读取速度快。 缺点也是基于内存，短时间内日志量大了容易爆内存，需要删处理过的数据（不同业务容易冲突），如果redis不配内存大小，会用swap区来存，应该会很低效。

kafka:分布式消息队列、基于磁盘，写入速度快（多机写入），读取速度也快，支持基于Topic的订阅模式，也支持普通的Queue的模式， 给不同模块来消费数据，避免数据混用，保障模块间解耦（如实时流业务 和离线存储模块各自持有自己当前的consumer offset）。 没有消息确认机制、负载均衡直接用zookeeper来做。

rabbitMQ：比较重量级，支持事务，吞吐量略逊kafka,存储支持内存和磁盘, 有消息确认机制， 也支持基于topic的订阅模式，负载均衡需要自己单独搞。

综上，对比如下：

写入速度： kafka>rabbitMQ>redis

读取速度： redis> kafka>rabbiMQ

吞吐量： kakfa>rabbitMQ>redis

可靠性： rabbitMQ>kafka>redis

综合考虑后，我们选择了kafka， kafka现在业界最流行、相关工具和支持都比较多。

遇到的坑 1）：我们的R平台kafka有3个节点，有一次删除kafka的没用的 topic。然后没有依次重启各个kafka节点， 导致kafka server进程的元信息没更新报找不到本地某些文件的错， 然后两个节点挂了， 只剩下一个节点可读写，严重影响了flume节点的对接和数据采集。

解决方案：调用节点的配置，以及依次重启各个节点。

遇到的坑2）：Kafka的消息，一开始设置了保存到磁盘30天，这个数据量是非常大的，导致爆了一次磁盘。

解决方案：消息保存调整为7天，配置到更大的空间的分区，做好磁盘监控

2）ETL

我们配置SparkStreaming的间隔时间是1分钟，收到的数据先解析成多个字段，然后再进行过滤，把含有不正常的taskId或者其它字段的请求丢掉。ETL之后的数据，先写到HDFS上，同时还要把这次ETL的请求的统计结果算出来，写到Redis中。

a.写到HDFS的数据-文件存储的坑

根据日志内容和**taskId前缀**进行分类，一分钟数据会写8个文件（4个日志类型分别为：visti.gif、repost.gif、unload.gif、button.gif；taskId前缀分为带maka-前缀，和没带maka-前缀）。

一开始，考虑到后面查询的效率，是每天每个taskId写一个文件，这样查单个任务的时候速度比较快。一天的taskId有8w左右，而Linux系统的文件句柄数最大为65536。这意味着如果要查询一天所有的taskId的文件都会很快把系统句柄数用满，系统就崩了，所以用文件做存储一定要考虑到文件的数量问题

后面决定按时间分区，同一天的taskId可以写在一个文件上。一分钟产生8个文件，一天会产生11520个文件，这个文件数还是偏大，如果要查询同时查几天的数据，也会很快把句柄数用完。所以，我们另起了一个定时程序，每30分钟，把当天一分钟产生的文件合并到一起，再搬到hive。这样每半小时8个文件，一天产生的文件数约为384。这样即使查一个月的数据也不会达到句柄数的上限。

b.写到redis的数据

由于客户对统计结果的延迟要求是5分钟内，就是一个请求requestA发出来，最迟5分钟，返回的统计已经有这个请求requestA的字段信息了。如果都等数据入库到hive后，再做统计，显然是来不及的。

我们把统计结果分成两部分，一部分是昨天及以前的数据，在每天凌晨的时候算好统计结果，储存到mysql，这部分称之为离线统计；另一部分是当天的数据，当天的数据在内存中算好后，储存到Redis，每5分钟，把Redis中统计结果写到mysql，这部分称之为实时统计；这样需要查询的时候，只要把离线统计的结果和实时统计的结果相加（合并），即可当前时刻的统计结果。

离线统计和实时统计能够相加的前提，该统计算法必须是增量的，有的字段的统计算法不能增量计算，一般这个字段占的资源都会比较大，只能直接取离线统计的结果来替代。

c.Sparkstreaming重启会丢失数据

发布新版本的时候，必须把Sparkstreaming关闭后再重启，此时如果Sparkstreaming正在消费数据，处理中的数据就会丢失。即使是1分钟数据的丢失，也是不可接受，特别是关键的repost.gif类型请求丢失，会影响好几个字段的统计。

要解决这个问题，有两种方案：1）每次消费都要记录kafka的offset和本次消费的数据是否已经入库的情况到外部存储。重启后的程序根据这两个信息就能决定自己从哪个offset开始消费，不会造成数据丢失，等于要自己实现事务。2）程序增加信号量监测，监测到kill命令时，调用spark的stop()方法来停止消费，这样它会等当前正在处理的数据全部完成后，才停止。

我们选择了第二种方案，因为看起来更加的优雅，毕竟自己实现方案一这种事务性操作还是比较麻烦的。这里要提醒大家注意是，重启后台程序的时候，都要考虑如果程序正在执行任务，被kill掉会不会有问题，有没有可能产生脏数据。

二、存储和统计

1.hive

hive是基于Hadoop的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供简单的sql查询功能，可以将sql语句转换为MapReduce任务进行运行。也就是说，只需要把数据按照一定的规范存储到HDFS，就可以映射成关系型数据库，然后用类SQL语言--HQL来做查询。

我们选择HQL最大的原因就在于，开发统计功能的时候，开发时间比较短。大部分字段的统计用group by 和count就能完成，只有一个复杂的字段（share\_level\_dist）必须用mr来统计。每个字段是分开统计的，这样就解耦了，一个字段统计出错，不会影响到其它字段。在debug和查bug的阶段都是非常方便的。平时需要快速查询某些统计信息的时候也是非常方便，直接在beeline命令行输入HQL就可以查询，不需要写个MR来查。

hive的缺点也是显而意见的，统计一个字段就要跑一次mr，有30个的字段，就要跑30次mr，也就是说，同一个文件要被扫30次。如果自己写mr，可能一次就可以统计大部分的字段了。现在离线统计是每天凌晨把昨天的数据统计一遍，整个过程大概3个小时左右。数据量增加5倍的时候，统计时间并未发生太大变化，说明这个统计时间跟数据量不是呈线性相关，所以还是可以接受的。如果后面对统计效率有更高的要求，那么可能就要把一些用HQL统计的字段，合并到一个MR去统计。

hive遇到的坑:

HiveServer爆内存：计算“分享层次分布（share\_level\_dist）”这个字段，逻辑比较复杂，HQL的条件长度会随着一个taskId的数据量增长。但遇到一个数据量达百万级别的taskId，HQL的长度可能达到100M以上，HiveServer需要把这个100M HQL解析成对应的MR，可能需要十几倍的内存，一开始HiveServer只有１G内存，就爆了。HiveServer内存调大这后，暂时解决了这个问题。但是taskId的数据量增长还会继续，所以要根本解决这个问题，就不能把逻辑都丢给HQL，必须自己写MR，这样才能更细的分配资源。

依赖Yarn：我们的hive的依赖yarn的，如果yarn跪了，hive就跪了。ZK的NoNode exception 会导致yarn RM挂掉，这是yarn的一个bug，可以通过升级hadoop或者自己打补丁的方式来解决。不再详述。

2.外部索引

在上面ETL部分已经提到，存到hive的数据是按天进行分区的，不同的taskId会写在同一个文件里面。如果某个字段的统计不能增量计算，必须一次读出所有指定taskId的所有数据，在上述说的存储情况下，只能遍历至今为止所有的文件才能完成这个统计。

为了解决上面的问题，我们在hive以外，自建一个索引，这个索引的key是taskId，value是dateList, dateListd存储是taskId所有有数据的date。这个索引每天更新一次，把当天有数据的taskId和当天的date加进索引。当你需要用到某个taskId所有的数据时，通过这个索引，可以限定只查某些date（分区），这样效率就提高了很多。现在这个索引是写在mysql，一个两个字段（task\_id,date）的表t\_h5m\_task\_date。

4.统计结果存储到mysql

上文已经提到，离线统计和实时统计的结果都会写到mysql，分别是t\_h5m\_stat\_dist\_history表和t\_h5m\_stat\_dist\_today表，用户或者自己的前端项目查询统计时，直接访问mysql这两个表，就能做到实时返回结果。

一天的taskId约8W个，一个taskId的统计结果一般写两行到mysql，一天大概就要新增16W行记录到mysql的一个表。100天的记录行数就会达到1600W行，列数目前是36个。这两个表超过1000W行后，性能就变得特别差，复杂一点的查询就会的几十秒的延迟，比如条件中含有in语句，即使in的字段有建索引，还是很慢。

我们做了几点优化：

1）PV，UV，share这3个字段是被查询的最多的，而其它字段被查询的频率要低很多。单独为这3个字段冗余一个新表（t\_h5m\_task\_day\_trend），只存储taskId，date和这3个字段，这样原表（t\_h5m\_stat\_dist\_history）的压力就会小一些。这种做法近似于竖向拆表。

2）t\_h5m\_stat\_dist\_history存储的是每个taskId每天（date）的数据。在实际的查询中，不用具体某天的结果，只需要从第一条数据开始到目前为止合并的统计结果。所以我们可以把一个taskId每一天的数据进行合并，这样行数就可以降低。在我们的项目中，每个月初，会把上个月（假设是11月）的数据进行合并，比如taskA上个月有数据的天数是20天，合并后只剩下一条11月的统计数据，把合并后的统计数据存储到月表（t\_h5m\_stat\_dist\_month）。目前实测的结果，把一个月每个taskId的数据合并到一行，平均总行数大概是原来的10%。查询的时候，则需要从月表（t\_h5m\_stat\_dist\_month）和旧的离线统计表（t\_h5m\_stat\_dist\_history）两个表里拿数据，进行合并后返回。“月表”也是业界解决此类问题最常见的解决方案。  
 3）实时统计表（t\_h5m\_stats\_dist\_today）有用的数据其实只有当天的数据，近几天的数据在出现错误的时候，还有查询的价值，几天前数据几乎是再也没有用了。所以today表的数据，只保留最近7天的，每天定时删除7天前的数据。这样实时统计表的行数基本在100W行以下，再也不会有性能的问题。