Spark Streaming实践和优化

2016-02-20 徐鑫 hadoop123

点击hadoop123关注我哟

☀ 最知名的hadoop/spark大数据技术分享基地，分享hadoop/spark技术内幕，hadoop/spark最新技术进展，hadoop/spark行业技术应用，发布hadoop/spark相关职位和求职信息，hadoop/spark技术交流聚会、讲座以及会议等。

作者：徐鑫

作者简介：毕业于清华大学，就职于Hulu大数据基础平台部门，专注于大数据应用，现负责用户数据平台Lambda架构和Market Segment Analysis系统。

责任编辑：仲浩（zhonghao@csdn.net）

文章来源：《程序员》2月期 大数据专刊

版权声明：本文为《程序员》原创文章，未经允许不得转载，原文链接：http://geek.csdn.net/news/detail/54500

一、Spark Streaming概述

Spark是美国加州伯克利大学AMP实验室推出的新一代分布式计算框架，其核心概念是RDD，一个只读的、有容错机制的分布式数据集，RDD可以全部缓存在内存中，并在多次计算中重复使用。相比于MapReduce编程模型，Spark具有以下几个优点：

a. 更大的灵活性和更高的抽象层次，使得用户用更少的代码即可实现同样的功能；

b. 适合迭代算法，在MapReduce编程模型中，每一轮迭代都需要读写一次HDFS，磁盘IO负载很大，相比之下，Spark中的RDD可以缓存在内存中，只需第一次读入HDFS文件，之后迭代的数据全都存储在内存中，这使得程序的计算速度可提升约10-100倍。

SparkStreaming是Spark生态系统中的重要组成部分，在实现上复用Spark计算引擎。如图1所示，Spark Streaming支持的数据源有很多，如Kafka、Flume、TCP等。SparkStreaming内部的数据表示形式为DStream（DiscretizedStream，离散的数据流），其接口设计与RDD非常相似，这使得它对Spark用户非常友好。SparkStreaming的核心思想是把流式处理转化为“微批处理”，即以时间为单位切分数据流，每个切片内的数据对应一个RDD，进而可以采用Spark引擎进行快速计算。由于SparkStreaming采用了微批处理方式，是近实时的处理系统，而不是严格意义上的流式处理系统。

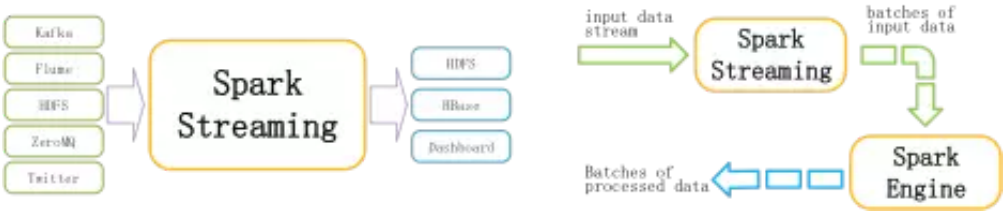

图1：Spark Streaming数据流

另一个著名的开源流式计算引擎是Storm，这是一个真正的流式处理系统，它每次从数据源读一条数据，然后单独处理。相比于SparkStreaming，Storm有更快速的响应时间（小于一秒），更适合低延迟的应用场景，比如信用卡欺诈系统，广告系统等。相比之下，SparkStreaming的优势是吞吐量大，响应时间也可以接受（秒级），并且兼容Spark系统中的其他工具库如MLlib和GraphX。对于时间不敏感且流量很大的系统，Spark Streaming是更优的选择。

二、Spark Streaming在Hulu应用

Hulu是美国的专业在线视频网站，每天会有大量用户在线观看视频，进而产生大量用户观看行为数据，这些数据通过收集系统进入Hulu的大数据平台，从而进行存储和进一步处理。在大数据平台之上，各个团队会根据需要设计相应的算法对数据进行分析和挖掘以便产生商业价值：推荐团队从这些数据里挖掘出用户感兴趣的内容并做精准推荐，广告团队根据用户的历史行为推送最合适的广告，数据团队从数据的各个维度进行分析从而为公司的策略制定提供可靠依据。

Hulu大数据平台的实现依循Lambda架构。Lambda架构是一个通用的大数据处理框架，包含离线的批处理层、在线的加速层和服务层三部分，具体如图2所示。服务层一般使用HTTP服务或自定制的客户端对外提供数据访问，离线的批处理层一般使用批处理计算框架Spark和MapReduce进行数据分析，在线的加速层一般使用流式实时计算框架SparkStreaming和Storm进行数据分析。

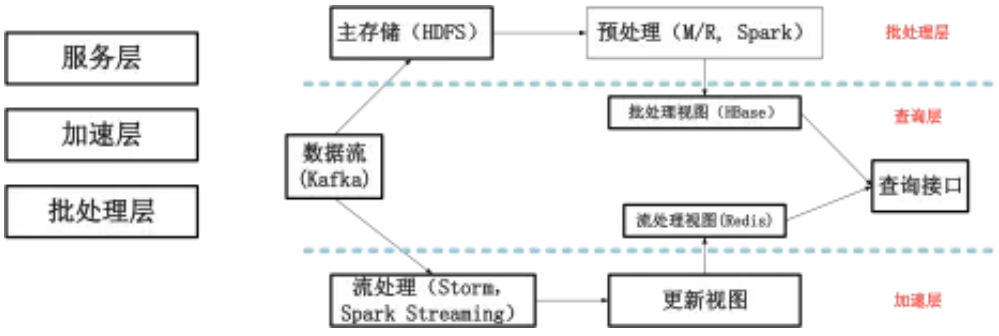

图2：lambda架构原理图

对于实时计算部分，Hulu内部使用了Kafka、Codis和SparkStreaming。下面按照数据流的过程，介绍我们的项目。

1. 收集数据 - 从服务器日志中收集数据，流程如图3所示：

a. 来自网页、手机App、机顶盒等设备的用户产生视频观看、广告点击等行为，这些行为数据记录在各自的Nginx服务的日志中；

b. 使用Flume将用户行为数据同时导入HDFS和Kafka，其中HDFS中的数据用于离线分析，而Kafka中数据则用于流式实时分析。

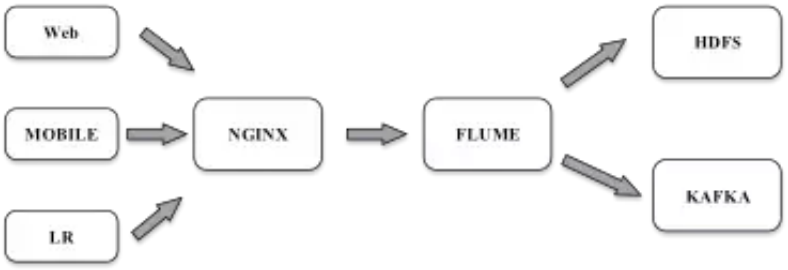

图3：Hulu数据收集流程

2. 存储标签数据 - Hulu使用HBase存储用户标签数据，包括基本信息如性别、年龄、是否付费，以及其他模型推测出来的偏好属性。这些属性需要作为计算模型的输入，同时HBase随机读取的速度比较慢，所以需要将数据同步到缓存服务器中以加快数据读取速度。Redis是一个应用广泛的开源缓存服务器，一个免费开源的高性能Key-Value数据库，但其本身是个单机系统，不能很好地支持大量数据的缓存。为解决Redis扩展性差的问题，豌豆荚开源了Codis，一个分布式Redis解决方案。Hulu将Codis打成Docker镜像，并实现一键式构建缓存系统，附带自动监控和修复功能。为了更精细的监控，我们构建了多个Codis缓存，分别是：

a. ··codis-profile，同步HBase中的用户属性；
b. ··codis-action，缓存来自Kafka的用户行为；
c. ··codis-result，记录计算结果。

3. 实时处理数据 - 准备就绪，启动Spark Streaming程序：

1) SparkStreaming启动Kafka Receiver，持续地从Kafka服务器拉取数据；
2) 每隔两秒，Kafka的数据被整理成一个RDD，交给Spark引擎处理；
3) 对一条用户行为，Spark会从codis-action缓存中拿到该用户的行为记录，然后把新的行为追加进去；
4) Spark从codis-action和codis-profile中获得该用户的所有相关属性，然后执行广告和推荐的计算模型，最后把结果写入codis-result，进而供服务层实时读取这些结果。

三、Spark Streaming优化经验

实践中，业务逻辑首先保证完成，使得在Kafka输入数据量较小的情况下系统稳定运行，且输入输出满足项目需求。然后开始调优，修改SparkStreaming的参数，比如Executor的数量，Core的数量，Receiver的流量等。最后发现仅调参数无法完全满足本项目的业务场景，所以有更进一步的优化方案，总结如下：

1. Executor初始化

很多机器学习的模型在第一次运行时，需要执行初始化方法，还会连接外部的数据库，常常需要5-10分钟，这会成为潜在的不稳定因素。在Spark Streaming应用中，当Receiver完成初始化，它就开始源源不断地接收数据，并且由Driver定期调度任务消耗这些数据。如

果刚启动时Executor需要几分钟做准备，会导致第一个作业一直没有完成，这段时间内Driver不会调度新的作业。这时候在Kafka Receiver端会有数据积压，随着积压的数据量越来越大，大部分数据会撑过新生代进入老年代，进而给Java GC带来严重的压力，容易引发应用程序崩溃。

本项目的解决方案是，修改Spark内核，在每个Executor接收任务之前先执行一个用户自定义的初始化函数，初始化函数中可以执行一些独立的用户逻辑。示例代码如下：

```
// sc:是SparkContext, setupEnvironment是Hulu扩展的API
sc.setupEnvironment(() => {
  application.initialize() // 用户应用程序初始化，需执行几分钟
  println( “Invoke executor  setup method successfully.” )
})
```

该方案需要更改Spark的任务调度器，首先将每个Executor设置为未初始化状态。此时，调度器只会给未初始化状态的Executor分配初始化任务（执行前面提到的初始化函数）。等初始化任务完毕，调度器更新Executor的状态为已初始化，这样的Executor才可以分配正常的计算任务。

2. 异步处理Task中的业务逻辑

本项目中，模型的输入参数均来自Codis，甚至模型内部也可能访问外部存储，直接导致模型计算时长不稳定，很多时间消耗在网络等待上。

为提高系统吞吐量，增大并行度是比较通用的优化方案，但在本项目的场景中并不适用。因为Spark作业的调度策略是，等待上一个作业的所有Task执行完毕，然后调度下一个作业。如果单个Task的运行时间不稳定，易发生个别Task拖慢整个作业的情况，以至于资源利用率不高，系统吞吐量上不去；甚至并行度越大，该问题越严重。一种常用的解决Task不稳定的方案是增大Spark Streaming的micro batch的时间间隔，该方案会使整个实时系统的延迟变长，并不推荐。

该问题的解决方案是异步处理Task中的业务逻辑。如下文的代码所示，同步方案中，Task内执行业务逻辑，处理时间不定；异步方案中，Task把业务逻辑嵌入线程，交给线程池执行，Task立刻结束，Executor向Driver报告执行完毕，异步处理的时间非常短，在100ms以内。另外，当线程池中积压的线程数量太大时（代码中qsize>100的情况），会暂时使用同步处理，配合反压机制（见下文的参数spark.streaming.backpressure.enabled），可以保证不会因为数据积压过多而导致系统崩溃。为设置合适的线程池大小，我们借助JVisualVM工具监控Executor的CPU使用率，通过调整参数找到最优并发线程数。经实验验证，该方案大大提高了系统的吞吐量。

```
// 同步处理
// 函数runBusinessLogic是 Task 中的业务逻辑，执行时间不定
rdd.foreachPartition(partition  => runBusinessLogic (partition))

// 异步处理，threadPool是线程池
```

```
rdd.foreachPartition(partition => {
 val  qsize = threadPool.getQueue.size // 线程池中积压的线程数
 if  (qsize > 100) {
   runBusinessLogic(partition)  // 暂时同步处理
 }
  threadPool.execute(new Runnable {
    override def run() = runBusinessLogic(partition)
 })
})
```

异步化Task也存在缺点：如果Executor发生异常，存放在线程池中的业务逻辑无法重新计算，会导致部分数据丢失。经实验验证，仅当Executor异常崩溃时有数据丢失，且不常见，在本项目的场景中可以接受。

3. Kafka Receiver的稳定性

本项目使用了SparkStreaming中的Kafka Receiver，本质上调用Kafka官方的客户端ZookeeperConsumerConnector。其策略是每个客户端在Zookeeper的固定路径下把自己注册为临时节点，于是所有客户端都知道其他客户端的存在，然后自动协调和分配Kafka的数据资源。该策略存在一个弊端，当一个客户端与Zookeeper的连接状态发生改变（断开或者连上），所有的客户端都会通过Zookeeper协调，重新分配Kafka的数据资源；在此期间所有客户端都断开与Kafka的连接，系统接收不到Kafka的数据，直到重新分配成功。如果网络质量不佳，并且Receiver的个数较多，这种策略会造成数据输入不稳定，很多SparkStreaming用户遇到这样的问题。在我们的系统中，该策略并没有产生明显的负面影响。值得注意的是，Kafka 客户端与Zookeeper有个默认的参数zookeeper.session.timeout.ms=6000，表示客户端与Zookeeper连接的session有效时间为6秒，我们的客户端多次出现因为Full GC超过6秒而与Zookeeper断开连接，之后再次连接上，期间所有客户端都受到影响，系统表现不稳定。所以项目中设置参数zookeeper.session.timeout.ms=30000。

4. YARN资源抢占问题

在Hulu内部，Spark Streaming这样的长时服务与MapRedue，Spark，Hive等批处理应用共享YARN集群资源。在共享环境中，经常因一个批处理应用占用大量网络资源或者CPU资源，导致Spark Streaming服务不稳定（尽管我们采用了CGroup进行资源隔离，但效果不佳）。更严重的问题是，如果个别Container崩溃Driver需要向YARN申请新的Container，或者如果整个应用崩溃需要重启，SparkStreaming不能保证很快申请到足够的资源，也就无法保证线上服务的质量。为解决该问题，Hulu使用label-based scheduling的调度策略，从YARN集群中隔离出若干节点专门运行SparkStreaming和其他长时服务，避免与批处理程序竞争资源。

5. 完善监控信息

监控反映系统运行的性能状态，也是一切优化的基础。SparkStreaming Web界面提供了比较丰富的监控信息，同时本项目依据业务逻辑的特点增加了更多监控。Hulu使用Graphite和Grafana作为第三方监控系统，本项目把系统中关键的性能参数（如计算时长

和次数）发送给Graphite服务器，就能够在Grafana网页上看到直观的统计图。

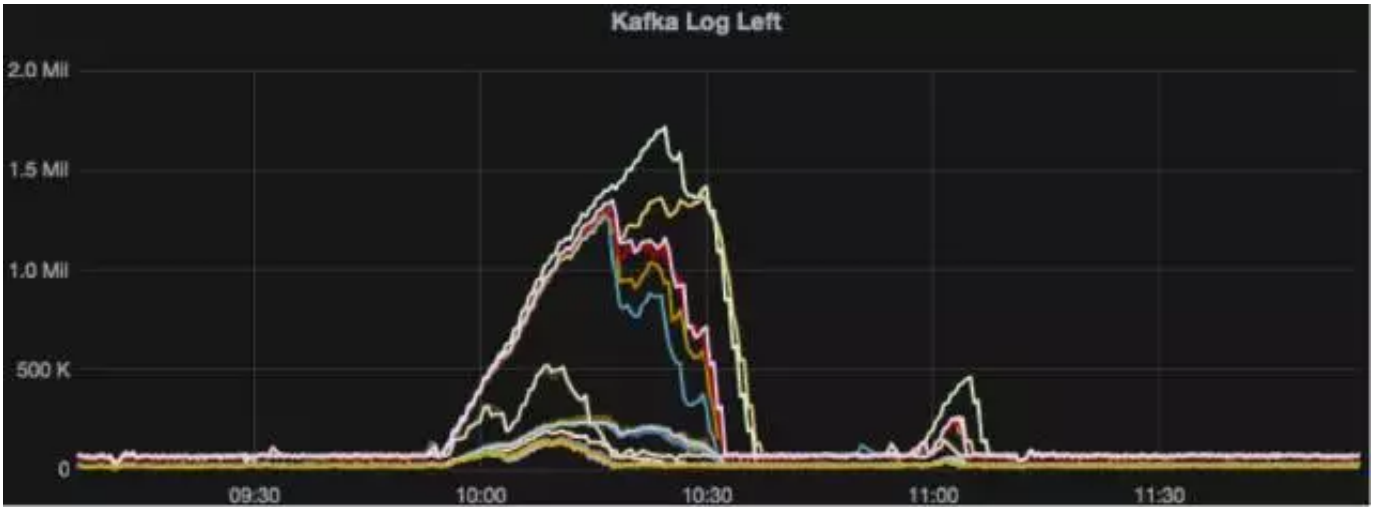

图4：Graphite监控信息，展示了Kafka中日志的剩余数量，一条线对应于一个partition的历史余量

图4是统计Kafka中日志的剩余数量，一条线对应于一个partition的历史余量，大部分情况下余量接近零，符合预期。图中09:55左右日志余量开始出现很明显的尖峰，之后又迅速逼近零。事后经过多种数据核对，证实Kafka的数据一直稳定，而当时Spark Streaming执行作业突然变慢，反压机制生效，于是Kafka Receiver减小读取日志的速率，造成Kafka数据积压；一段时间之后SparkStreaming又恢复正常，快速消耗了Kafka中的数据余量。

直观的监控系统能有效地暴露问题，进而理解和强化系统。对于不同的业务逻辑，需要监控的信息也不相同。在我们的实践中，主要的监控指标有：

a. ·· Kafka的剩余数据量
b. ··Spark的作业运行时间和调度时间
c. ··每个Task的计算时间
d. ··Codis的访问次数、时间、命中率

另外，有脚本定期分析这些统计数据，出现异常则发邮件报警。比如图4中 Kafka 的日志余量过大时，会有连续的报警邮件。我们的经验是，监控越细致，之后的优化工作越轻松。同时，优秀的监控也需要对系统深刻的理解。

6. 参数优化

下表列出本项目中比较关键的几个参数：

spark.yarn.max.executor.failures	Executor允许的失败上限；如果超过该上限，整个Spark Streaming会失败，需要设置比较大
spark.yarn.executor.memoryOverhead	Executor中JVM的开销，与堆内存不一样，设置太小会导致内存溢出异常
spark.receivers.num	Kafka Receiver的个数
spark.streaming.receiver.maxRate	每个Receiver能够接受数据的最大速率；这个值超过峰值约50%

spark.streaming.backpressure.enabled	反压机制；如果目前系统的延迟较长，Receiver端会自动减小接受数据的速率，避免系统因数据积压过多而崩溃
spark.locality.wait	系统调度Task会尽量考虑数据的局部性，如果超过spark.locality.wait设置时间的上限，就放弃局部性；该参数直接影响Task的调度时间
spark.cleaner.ttl	Spark系统内部的元信息的超时时间；Streaming长期运行，元信息累积太多会影响性能

四、总结

Spark Streaming的产品上线运行一年多，期间进行了多次Spark版本升级，从最早期的0.8版本到最近的1.5.x版本。总体上Spark Streaming是一款优秀的实时计算框架，可以在线上使用。但仍然存在一些不足，包括：

1. Spark同时使用堆内和堆外的内存，缺乏一些有效的监控信息，遇到OOM时分析和调试比较困难；
2. 缺少Executor初始化接口；
3. Spark采用函数式编程方式，抽象层次高，好处是使用方便，坏处是理解和优化困难；
4. 新版本的Spark有一些异常，如Shuffle过程中Block丢失、内存溢出。

阅读原文