1.Spark的Shuffle原理及调优

参考：https://www.zhihu.com/question/27643595

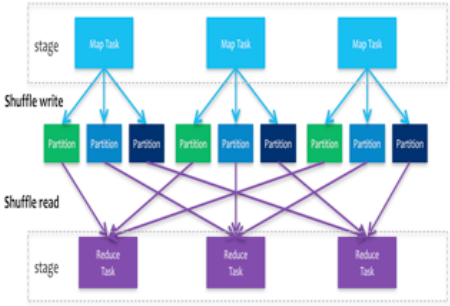
(1)**shuffle**原理

当使用reduceByKey、groupByKey、sortByKey、countByKey、join、cogroup等操作的时候，会发生shuffle操作。

Spark在DAG调度阶段将job划分成多个stage，上游stage做map操作，下游stage做reduce操作，其本质还是MR计算架构。Shuffle是连接map和reduce之间的桥梁，它将map的输出对应到reduce的输入，这期间涉及到序列化和反序列化、跨节点网络IO和磁盘读写IO等，所以说shuffle是整个应用过程特别昂贵的阶段。

与MapReduce计算框架一样，spark的shuffle实现大致如下图所示，在DAG阶段以shuffle为界，划分stage，上游stage做map task，每个map task将计算结果数据分成多份，每一份对应到下游stage的每个partition中，并将其临时写到磁盘，该过程就叫做shuffle write；下游stage叫做reduce task，每个reduce task通过网络拉取指定分区结果数据，该过程叫做shuffle read，最后完成reduce的业务逻辑。举例：上游stage有100个map task，下游有1000个reduce task，那么这100个map task中每个map task都会得到1000份数据，而这1000个reduce task中的每个reduce task都会拉取上游100个map task对应的那份数据，即第一个reduce task会拉取所有map task结果数据的第一份，以此类推。

在map阶段，除了map的业务逻辑外，还有shuffle write的过程，这个过程涉及序列化、磁盘IO等耗时操作；在reduce阶段，除了reduce的业务逻辑外，还有shuffle read过程，这个过程涉及到网络IO、反序列化等耗时操作。所以整个shuffle过程是极其昂贵的。



因为shuffle是一个涉及CPU(序列化反序列化)、网络IO(跨节点数据传输)以及磁盘IO(shuffle中间结果落地)的操作，所以应当考虑shuffle相关的调优，提升spark应用程序的性能。

(2)shuffle调优

(2-1)程序调优；

首先，尽量减少shuffle次数；

```
//两次shuffle
rdd.map().repartition(1000).reduceByKey(_+_,3000)
//一次shuffle
Rdd.map().repartition(3000).reduceByKey(_+_)
```

然后必要时主动shuffle，通常用于改变并行度，提高后续分布式运行速度；

```
rdd.repartition(largerNumPartition).map()
```

最后，使用treeReduce&treeAggregate替换reduce&aggregate。数据量较大时，reduce&aggregate一次性聚合，shuffle量太大，而treeReduce&treeAggregate是分批聚合，更为保险。

(2-2)参数调优；

spark.shuffle.file.buffer:map task到buffer到磁盘

默认值：32K

参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘；

调优建议：如果作业可用的内存资源较为充足的话，可以适当**增加这个参数的大小(比如64k)，从而减少shuffle write过程中溢写磁盘文件的次数，也就可以减少磁盘IO次数，进而提升性能**。在实践中发现，合理调节该参数，性能会有1到5%的提升。

spark.reducer.maxSizeFlight:reduce task去磁盘拉取数据

默认值：48m

参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。

调优建议：如果作业可用的内存资源较为充足的话，可以增加这个参数的大小(比如96M)，从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1到5%的提升。

Spark.shuffle.io.maxRetries

默认值：3

参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，时会自动进行重试的。该参数就代表了可以重试的最大次数，如果在指定次数内拉取属于还是没有成功，就可能会导致作业执行失败。

调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数(比如6次)，可以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于超大数据量(数十亿到上百亿)的shuffle过程，调节该参数可以大幅度提升稳定性。

Spark.shuffle.io.retryWait

默认值：5s

参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果拉取失败了每次重试拉取数据的等待时间间隔，默认是5s；

调优建议：建议加大时间间隔时长，比如60s，以增加shuffle操作的稳定性。

spark.shuffle.memoryFraction

默认值：0.2

参数说明：该参数代表了executor内存中，分配给shuffle read task进行聚合操作的内存比例，默认是20%；

调优建议：如果内存充足，而且很少使用持久化操作，建议调高和这个比例，给shuffle read的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘。在实践中发现，合理调节该参数可以将性能提升10%。

Spark.shuffle.manager

默认值：sort

参数说明：该参数用于设置shuffleManager的类型。Spark1.5以后有三个可选项：hash、sort和tungsten-sort。Tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率提高。

调优建议：由于sort shuffleManager默认会对数据进行排序，因此如果你的业务逻辑中需要该排序机制的话，则使用默认的sort ShuffleManager就可以；但是如果你的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过bypass机制或优化的hash ShuffleManager来避免排序操作，同时提供较好的磁盘读写性能。这里要注意的是，tungsten-sort要慎用，因为之前发现了一些相应的bug。

Spark.shuffle.sort.bypassMergeThreshold

默认值：200

参数说明：当shuffleManager为sortshuffleManager时，如果shuffle read task的数量小于这个阈值，则shuffle write过程中不会进行排序操作，而是直接按照未经优化的hashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。

调优建议：当你使用sortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量，那么此时就会自动启用bupass机制，map-side就不会进行排序，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

Spark.shuffle.consolidateFiles

默认值：false

参数说明：如果使用hashShuffleManager，该参数有效。如果设置为true，那么就会开启consilidate机制，会大幅度合并shuflle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。

调优建议：如果的确不需要sortHashShuffle的排序机制，那么除了使用bypass机制，还可以尝试将spark.shuffle.manager参数手动调节为hash，使用hashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的sortshuffleManager要高出10%到30%。

2.hadoop和spark使用场景?

Hadoop/MapReduce和Spark最适合的都是做离线型的数据分析，但Hadoop特别适合是单次分析的数据量“很大”的情景，而Spark则适用于数据量不是很大的情景。

(1)一般情况下，对于中小互联网和企业级的大数据应用而言，单次分析的数量都不会“很大”，因此可以优先考虑使用Spark。

(2)业务通常认为Spark更适用于机器学习之类的“迭代式”应用,80GB的压缩数据（解压后超过200GB），10个节点的集群规模，跑类似“sum+group-by”的应用，MapReduce花了5分钟，而spark只需要2分钟。

3.spark如何保证宕机迅速恢复?

(1)适当增加spark standby master

(2)编写shell脚本，定期检测master状态，出现宕机后对master进行重启操作

4.hadoop和spark的相同点和不同点?

1、Hadoop底层使用MapReduce计算架构，只有map和reduce两种操作，表达能力比较欠缺，而且在MR过程中会重复的读写hdfs，造成大量的磁盘io读写操作，所以适合高时延环境下批处理计算的应用；

2、Spark是基于内存的分布式计算架构，提供更加丰富的数据集操作类型，主要分成转化操作和行动操作，包括map、reduce、filter、flatmap、groupbykey、reducebykey、union和join等，数据分析更加快速，所以适合低时延环境下计算的应用；

3、spark与hadoop最大的区别在于迭代式计算模型。基于mapreduce框架的Hadoop主要分为map和reduce两个阶段，两个阶段完了就结束了，所以在一个job里面能做的处理很有限；spark计算模型是基于内存的迭代式计算模型，可以分为n个阶段，根据用户编写的RDD算子和程序，在处理完一个阶段后可以继续往下处理很多个阶段，而不只是两个阶段。所以spark相较于mapreduce，计算模型更加灵活，可以提供更强大的功能。

4、但是spark也有劣势，由于spark基于内存进行计算，虽然开发容易，但是真正面对大数据的时候，在没有进行调优的轻局昂下，可能会出现各种各样的问题，比如OOM内存溢出等情况，导致spark程序可能无法运行起来，而mapreduce虽然运行缓慢，但是至少可以慢慢运行完。

5.RDD持久化原理?

spark非常重要的一个功能特性就是可以将RDD持久化在内存中。调用cache()和persist()方法即可。cache()和persist()的区别在于，cache()是persist()的一种简化方式，cache()的底层就是调用persist()的无参版本persist(MEMORY_ONLY)，将数据持久化到内存中。如果需要从内存中清除缓存，可以使用unpersist()方法。

RDD持久化是可以手动选择不同的策略的。在调用persist()时传入对应的StorageLevel即可。

(1)MEMORY_ONLY:以非序列化的Java对象的方式持久化在JVM内存中。如果内存无法完全存储RDD所有的partition，那么那些没有持久化的partition就会在下一次需要使用它的时候，被重新计算。

(2)MEMORY_AND_DISK:同上，但是当某些partition无法存储在内存中时，会持久化到磁盘中。下次需要使用这些partition时，需要从磁盘上读取。

(3)MEMORY_ONLY_SER:同MEMORY_ONLY，但是会使用Java序列化方式，将Java对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大CPU开销。

(4)MEMORY_AND_DSK_SER:同MEMORY_AND_DSK，但是使用序列化方式持久化Java对象。

(5)DISK_ONLY:使用非序列化Java对象的方式持久化，完全存储到磁盘上。

(6)MEMORY_ONLY_2/MEMERY_AND_DISK_2：如果是尾部加了2的持久化级别，表示会将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可。

6.checkpoint检查点机制?

1、应用场景：当spark应用程序特别复杂，从初始的RDD开始到最后整个应用程序完成有很多的步骤，而且整个应用运行时间特别长，这种情况下就比较适合使用checkpoint功能。

2、原因：对于特别复杂的Spark应用，会出现某个反复使用的RDD，即使之前持久化过但由于节点的故障导致数据丢失了，没有容错机制，所以需要重新计算一次数据。

Checkpoint首先会调用SparkContext的setCheckPointDIR()方法，设置一个容错的文件系统的目录，比如说HDFS；然后对RDD调用checkpoint()方法。之后在RDD所处的job运行结束之后，会启动一个单独的job，来将checkpoint过的RDD数据写入之前设置的文件系统，进行高可用、容错的类持久化操作。

检查点机制是我们在spark streaming中用来保障容错性的主要机制，它可以使spark streaming阶段性的把应用数据存储到诸如HDFS等可靠存储系统中，以供恢复时使用。具体来说基于以下两个目的服务：

控制发生失败时需要重算的状态数。Spark streaming可以通过转化图的谱系图来重算状态，检查点机制则可以控制需要在转化图中回溯多远。提供驱动器程序容错。如果流计算应用中的驱动器程序崩溃了，你可以重启驱动器程序并让驱动器程序从检查点恢复，这样spark streaming就可以读取之前运行的程序处理数据的进度，并从那里继续。

参考：http://www.cnblogs.com/dt-zhw/p/5664663.html

7.checkpoint和持久化机制的区别?

1、最主要的区别在于持久化只是将数据保存在BlockManager中，但是RDD的lineage(血缘关系，依赖关系)是不变的。但是checkpoint执行完之后，rdd已经没有之前所谓的依赖rdd了，而只有一个强行为其设置的checkpointRDD，checkpoint之后rdd的lineage就改变了。

2、持久化的数据丢失的可能性更大，因为节点的故障会导致磁盘、内存的数据丢失。但是checkpoint的数据通常是保存在高可用的文件系统中，比如HDFS中，所以数据丢失可能性比较低。

8.Spark Streaming和Storm有何区别?

Spark Streaming与Storm都可以用于进行实时流计算，但是他们两者的区别是非常大的。

1、Storm的优势在于以下两个方面：一方面，Spark Streaming和Storm的计算模型完全不一样。Spark Streaming是基于RDD的，因此需要将一小段时间内的，比如1秒内的数据，收集起来，作为一个RDD，然后再针对这个batch的数据进行处理。而Storm却可以做到每来一条数据，都可以立即进行处理和计算。因此，Spark Streaming只能称作准实时的流计算框架，而Storm是真正意义上的实时计算框架；另一方面，Storm支持在分布式流式计算程序运行过程中，可以动态地调整并行度，从而动态提高并发处理能力。而Spark Streaming是无法动态调整并行度的；

2、Spark Streaming的优势在于：一方面，由于Spark Streaming是基于batch进行处理的，因此相较于Storm基于单条数据进行处理，具有数倍甚至数十倍的吞吐量；另一方面，Spark Streaming由于身处于Spark生态圈内，因此可以和Spark Core、Spark SQL、Spark MLlib、Spark GraphX进行无缝整合。流式处理完的数据，可以立即进行各种map、reduce转换操作，可以立即使用sql进行查询，甚至可以立即使用machine learning或者图计算算法进行处理。这种一站式的大数据处理功能和优势，是Storm无法匹敌的。

3、综合上述来看，通常在对实时性要求特别高，而且实时数据量不稳定，比如在白天有高峰期的情况下，可以选择使用Storm。但是如果是对实时性要求一般，允许1秒的准实时处理，而且不要求动态调整并行度的话，选择Spark Streaming是更好的选择。

9.RDD机制?

1. rdd分布式弹性数据集，简单的理解成一种数据结构，是spark框架上的通用货币。 所有算子都是基于rdd来执行的，不同的场景会有不同的rdd实现类，但是都可以进行互相转换。 rdd执行过程中会形成dag图，然后形成lineage保证容错性等。 从物理的角度来看rdd存储的是block和node之间的映射。

2. RDD是spark提供的核心抽象，全称为弹性分布式数据集。

3. RDD在逻辑上是一个hdfs文件，在抽象上是一种元素集合，包含了数据。它是被分区的，分为多个分区，每个分区分布在集群中的不同结点上，从而让RDD中的数据可以被并行操作（分布式数据集）比如有个RDD有90W数据，3个partition，则每个分区上有30W数据。RDD通常通过Hadoop上的文件，即HDFS或者HIVE表来创建，还可以通过应用程序中的集合来创建；

4. RDD最重要的特性就是容错性，可以自动从节点失败中恢复过来。即如果某个结点上的RDD partition因为节点故障，导致数据丢失，那么RDD可以通过自己的数据来源重新计算该partition。这一切对使用者都是透明的RDD的数据默认存放在内存中，但是当内存资源不足时，spark会自动将RDD数据写入磁盘。比如某结点内存只能处理20W数据，那么这20W数据就会放入内存中计算，剩下10W放到磁盘中。RDD的弹性体现在于RDD上自动进行内存和磁盘之间权衡和切换的机制。

10.Spark streaming以及基本工作原理?

Spark streaming是spark core API的一种扩展，可以用于进行大规模、高吞吐量、容错的实时数据流的处理。它支持从多种数据源读取数据，比如Kafka、Flume、Twitter和TCP Socket，并且能够使用算子比如map、reduce、join和window等来处理数据，处理后的数据可以保存到文件系统、数据库等存储中。

Spark streaming内部的基本工作原理是：接受实时输入数据流，然后将数据拆分成batch，比如每收集一秒的数据封装成一个batch，然后将每个batch交给spark的计算引擎进行处理，最后会生产处一个结果数据流，其中的数据也是一个一个的batch组成的。

11.DStream以及基本工作原理?

DStream是spark streaming提供的一种高级抽象，代表了一个持续不断的数据流。DStream可以通过输入数据源来创建，比如Kafka、flume等，也可以通过其他DStream的高阶函数来创建，比如map、reduce、join和window等。

DStream内部其实不断产生RDD，每个RDD包含了一个时间段的数据。

Spark streaming一定是有一个输入的DStream接收数据，按照时间划分成一个一个的batch，并转化为一个RDD，RDD的数据是分散在各个子节点的partition中。

12.spark有哪些组件?

(1)master：管理集群和节点，不参与计算。

(2)worker：计算节点，进程本身不参与计算，和master汇报。

(3)Driver：运行程序的main方法，创建spark context对象。

(4)spark context：控制整个application的生命周期，包括dagsheduler和task scheduler等组件。

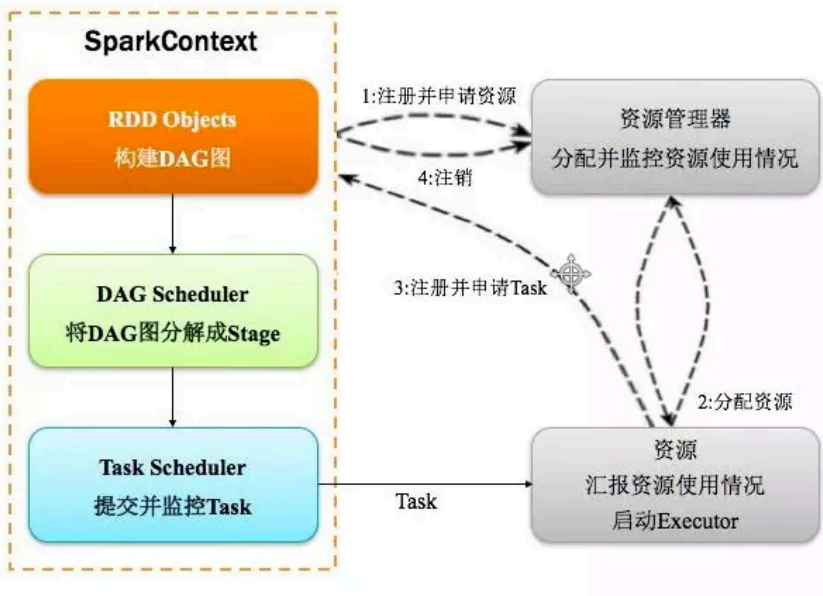
(5)client：用户提交程序的入口。

13.spark工作机制?

1、用户在client端提交作业后，会由Driver运行main方法并创建spark context上下文。

2、执行add算子，形成dag图输入dagscheduler，按照add之间的依赖关系划分stage输入task scheduler。 task scheduler会将stage划分为task set分发到各个节点的executor中执行。

14.说说Spark工作流程?



1、构建Spark Application的运行环境（启动SparkContext），SparkContext向资源管理器（可以是Standalone、Mesos或YARN）注册并申请运行Executor资源；

2、资源管理器分配Executor资源并启动Executor，Executor运行情况将随着心跳发送到资源管理器上；

3、SparkContext构建成DAG图，将DAG图分解成Stage，并把Taskset发送给Task Scheduler。Executor向SparkContext申请Task，Task Scheduler将Task发放给Executor运行同时SparkContext将应用程序代码发放给Executor。

4、Task在Executor上运行，运行完毕释放所有资源。

15.spark核心编程原理?

(1)定义初始的RDD，即第一个RDD是从哪里来，读取数据，包括hdfs、linux本地文件、程序中的集合；

(2)定义对RDD的计算操作，这在spark中称为算子，转换操作和行动操作，包括map、reduce、flatmap、reducebykey等，比mapreduce提供的map和reduce强大的太多；

(3)就是循环往复迭代的过程。第一个计算完了以后，数据可能到了新的一批结点上，

变成了一个新的RDD。然后再次反复，针对新的RDD定义计算操作；

(4)获得最终的数据，将保存起来。

16.spark基本工作原理?

客户端:我们在本地编写了spark程序，必须在某台能够连接spark集群的机器上提交该spark程序；

spark集群：将spark程序提交到spark集群上进行运行。

17.spark性能优化有哪些?

(1)**使用Kryo进行序列化**。

在spark中主要有三个地方涉及到序列化：

第一，在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输；

第二，将自定义的类型作为RDD的泛型数据时(JavaRDD，Student是自定义类型)，所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现serializable借口；

第三，使用可序列化的持久化策略时，spark会将RDD中的每个partition都序列化成为一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过Kryo序列化类库，来优化序列化和反序列化的性能。Spark默认采用的是Java的序列化机制。但是Spark同时支持使用Kryo序列化库，而且Kryo序列化类库的性能比Java的序列化类库要高。官方介绍，Kryo序列化比Java序列化性能高出10倍。Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求最好要注册所有需要进行序列化的自定义类型，因此对于开发者来说这种方式比较麻烦。

(2)**优化数据结构**。

Java中有三种类型比较耗费内存：**对象**，每个Java对象都有对象头、引用等额外的信息，因此比较占用内存空间；**字符串**，每个字符串内部都有一个字符数组以及长度等额外信息；**集合类型**，比如HashMap、LinkedList等，因为集合类型内部通常会使用一些内部类来封装集合元素，比如Map.Entry。

因此Spark官方建议，在spark编码实现中，特别对于算子函数中的代码，尽量不要使用上述三种数据结构，尽量使用字符串替代对象，使用原始类型(比如int、long)替代字符串，使用数组替代集合类型，这样尽可能地减少内存占用，从而降低GC频率，提升性能。

使用数组替代集合类型：举例：有个List<Integer> list=new ArrayList<Integer>(),可以将其替换为int[] arr=new int[].这样array既比list少了额外信息的存储开销，还能使用原始数据类型（int）来存储数据，比list中用Integer这种包装类型存储数据，要节省内存的多。

使用字符串替代对象：还比如，通常企业级应用中的做法是，对于HashMap、List这种数据，统一用String拼接成特殊格式的字符串，比如Map<Integer,Person> persons = new HashMap<Integer,Person>(),可以优化为特殊的字符串格式：id：name，address|id：name，address。

再比如，**避免使用多层嵌套的对象结构**。比如说，public class Teacher{private List<Student> students =new ArrayList<Student>()}.就是非常不好的例子。因为Teacher类的内部又嵌套了大量的小Student对象。解决措施是，完全可以使用特殊的字符串来进行数据的存储，比如用json字符串来存储数据就是一个好的选择。{"teacherId":1,"teacherName":"leo",students:[{},{}]}

使用原始类型(比如int、long)替代字符串：对于有些能够避免的场景，尽量使用int代替String。因为String虽然比ArrayList、HashMap等数据结构高效多了，占用内存上少多了，但是还是有额外信息的消耗。比如之前用String表示id，那么现在完全可以用数字类型的int，来进行替代。这里提醒，在spark应用中，id就不要使用常用的uuid，因为无法转成int，就用自增的int类型的id即可。

但是在编码实践中要做到上述原则其实并不容易。因为要同时考虑到代码的可维护性，如果一个代码中，完全没有任何对象抽象，全部是字符串拼接的方式，那么对于后续的代码维护和修改，无疑是一场巨大的灾难。同理，如果所有操作都基于数组实现，而不是用HashMap、LinkedList等集合类型，那么对于我们编码的难度以及代码的可维护性，也是一个极大的挑战。因此建议是在保证代码可维护性的前提下，使用占用内存较少的数据结构。

(4)**对多次使用的RDD进行持久化并序列化**

原因：Spark中对于一个RDD执行多次算子的默认原理是这样的：每次对一个RDD执行一个算子操作时，都会重新从源头出计算一遍，计算出那个RDD来，然后再对这个RDD执行你的算子操作。这种方式的性能是很差的。

解决办法：因此对于这种情况，建议是对多次使用的RDD进行持久化。此时spark就会根据你的持久化策略，将RDD中的数据保存到内存或者磁盘中。以后每次对这个RDD进行算子操作时，都会直接从内存或磁盘中提取持久化的RDD数据，然后

执行算子，而不会从源头出重新计算一遍这个RDD。

(5)垃圾回收调优

α. 首先使用更高效的数据结构，比如array和string；

b. 其次是在持久化rdd时，使用序列化的持久化级别，而且使用Kryo序列化类库；这样每个partition就只是一个对象(一个字节数组)

c. 然后是监测垃圾回收，可以通过在spark-submit脚本中，增加一个配置即可--conf"spark.executor.extraJavaOptions=-verbose:gc-XX:+PrintGCDetails -XX:+PrintGCTimeStamps"。注意，这里打印出java虚拟机的垃圾回收的相关信息，但是输出到了worker上的日志，而不是driver日志上。还可以通过sparkUI（4040端口）来观察每个stage的垃圾回收的情况；

d. 然后，**优化executor内存比例**。对于垃圾回收来说，**最重要的是调节RDD缓存占用的内存空间，与算子执行时创建对象占用的内存空间的比例**。**默认是60%存放缓存RDD，40%存放task执行期间创建的对象**。出现的问题是，task创建的对象过大，一旦发现40%内存不够用了，就会频繁触发GC操作，从而频繁导致task工作线程停止，降低spark程序的性能。**解决措施是调优这个比例**，使用new SparkConf().set("spark.storage.memoryFraction", "0.5")即可，给年轻代更多的空间来存放短时间存活的对象。

e. 最后，如果发现task执行期间大量的Full GC发生，那么说明年轻代的Eden区域给的空间不够大，可以执行以下操作来优化垃圾回收行为：给Eden区域分配更大的空间，使用-Xmn即可，通常建议给Eden区域预计大小的4/3；如果使用hdfs文件，那么很好估计Eden区域大小。如果每个executor有4个task，然后每个hdfs压缩块解压后大小是3倍，此外每个hdfs块的大小是64M，那么Eden区域的预计代销就是4*3*64MB，通过-Xmn参数，将Eden区域大小设置为4*3*64*4/3。

(6)提高并行度

减少批处理所消耗时间的常见方式还有提高并行度。首先可以增加接收器数目，当记录太多导致但台机器来不及读入并分发的话，接收器会成为系统瓶颈，这时需要创建多个输入DStream来增加接收器数目，然后使用union来把数据合并为一个数据源；然后可以将接收到的数据显式的重新分区，如果接收器数目无法在增加，可以通过使用DStream.repartition来显式重新分区输入流来重新分配收到的数据；最后可以提高聚合计算的并行度，对于像reduceByKey()这样的操作，可以在第二个参数中制定并行度。

(7)广播大数据集

有时会遇到在算子函数中使用外部变量的场景，建议使用spark的广播功能来提升性能。默认情况下，算子函数使用外部变量时会将该变量复制多个副本，通过网络传输到task中，此时每个task都有一个变量副本。如果变量本身比较大，那么大量的变量副本在网络中传输的性能开销以及在各个节点的executor中占用过多的内存导致频繁GC，都会极大影响性能。所以**建议使用spark的广播性能，对该变量进行广播。广播的好处在于，会保证每个executor的内存中，只驻留一份变量副本，而executor中的task执行时会共享该executor中的那份变量副本**。这样的话，可以大大降低变量副本的数量，从而减少网络传输的性能开销，并减少对executor内存的占用开销，降低GC的频率。

(8)数据本地化

数据本地化对于spark job性能有着巨大的影响。如果数据以及要计算它的代码是在一起的，那么性能自然会高。但是如果数据和计算它的代码是分开的，那么其中之一必须要另外一方的机器上。通常来说，移动代码到其他节点，会比移动数据到所在节点上去，速度要快的多，因为代码比较小。Spark也正是基于整个数据本地化的原则来构建task调度算法的。

数据本地化，指的是数据距离它的代码有多近。基于数据距离代码的距离，有几种数据本地化级别：

(a)PROCESS_LOCAL:数据和计算它的代码在同一个JVM进程里面；

(b)NODE_LOCAL:数据和计算它的代码在一个节点上，但是不在一个进程中，比如不在同一个executor进程中，或者是数据在hdfs文件的block中；

(c)NO_PREF:数据从哪里过来，性能都是一样的；

(d)RACK_LOCAL:数据和计算它的代码在一个机架上；

(e)ANY：数据可能在任意地方，比如其他网络环境内，或者其他机架上。

Spark倾向于使用最好的本地化级别来调度task，但是这是不可能的。如果没有任何未处理的数据在空闲的executor上，那么Spark就会放低本地化级别。这时有两个选择：第一，等待，直到executor上的cpu释放出来，那么就分配task过去；第二，立即在任意一个executor上启动一个task。Spark默认会等待一会，来期望task要处理的数据所在的节点上的executor空闲出一个cpu，从而将task分配过去。只要超过了时间，那么spark就会将task分配到其他任意一个空闲的executor上。**可以设置参数，spark.locality系列参数，来调节spark等待task可以进行数据本地化的时间**。

saprk.locality.wait(3000ms)、spark.locality.wait.node、spark.locality.wait.process、spark.locality.wait.rack。

(9)尽量使用高性能的算子

α. 使用**reduceBykey/aggregateBykey替代groupByKey**。原因是：如果因为业务需要，一定要使用shuffle操作，无法

用map类算子来代替，那么尽量使用可以map-side预聚合的算子。所谓的**map-side预聚合**，说的是在每个节点本地对相同的key进行一次聚合操作，类似于MR的本地combiner。**map-side预聚合之后，每个节点本地就只会有一条相同的key**，因为多条相同的key都被聚合起来了。其他节点在拉取所有节点上的相同key时，就会大大减少需要拉取的数据量，从而也就减少了磁盘IO以及网络传输开销。通过来说，在可能的情况下，建议尽量使用reduceByKey或者aggregateByKey算子来替代groupBykey算子。因为reduceBykey和aggregateBykey算子都会使用用户自定义的函数对每个节点本地相同的key进行预聚合。但是groupbykey算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

b. 使用mapPartitions替代普通map；

c. 使用foreachPartitions替代foreach；

d. 使用filter之后进行coalesce操作：通常对一个RDD执行filter算子过滤掉RDD中以后比较多的数据后，建议使用coalesce算子，手动减少RDD的partitioning数量，将RDD中的数据压缩到更少的partition中去，只要使用更少的task即可处理完所有的partition，在某些场景下对性能有提升。

e. 使用repartitionAndSortWithinPartitions替代repartition与sort类操作：repartitionAndSortWithinPartitions是spark官网推荐的一个算子。官方建议，如果需要在repartition重分区之后，还要进行排序，建议直接使用是这个算子。因为该算子可以一边进行重分区的shuffle操作，一边进行排序。Shuffle和sort两个操作同时进行，比先shuffle再sort来说，性能更高。

(10)**shuffle性能调优**

(11)**批次和窗口大小的设置(针对spark streaming中的特殊优化)**

最常见的问题是Spark Streaming可以使用的最小批次间隔是多少。寻找最小批次大小的最佳实践是从一个比较大的批次开始，不断使用更小的批次大小。如果streaming用户界面中显示的处理时间保持不变，那么就可以进一步减小批次大小。对于窗口操作，计算结果的间隔对于性能也有巨大的影响。

18.说说updateStateByKey

对整个实时计算的所有时间间隔内产生的相关数据进行统计。

spark streaming的解决方案是累加器，工作原理是定义一个类似全局的可更新的变量，每个时间窗口内得到的统计值都累加到上个时间窗口得到的值，这样整个累加值就是跨越多个时间间隔。

updateStateByKey操作可以让我们为每个key维护一份state，并持续不断的更新该state。

首先，要定义一个state，可以是任意的数据类型；

其次，要定义state更新函数(指定一个函数如何使用之前的state和新值来更新state)。对于每个batch，spark都会为每个之前已经存在的key去应用一次state更新函数，无论这个key在batch中是否有新的数据。如果state更新函数返回none，那么key对应的state就会被删除。当然对于每个新出现的key也会执行state更新函数。注意updatestateBykey要求必须开启checkpoint机制。

updateStateByKey返回的都是DStream类型。根据updateFunc这个函数来更新状态。其中参数：Seq[V]是本次的数据类型，Option[S]是前次计算结果类型，本次计算结果类型也是Option[S]。计算肯定需要Partitioner。因为Hash高效率且不做排序，默认Partitioner是HashPartitoner。

由于cogroup会对所有数据进行扫描，再按key进行分组，所以性能上会有问题。特别是随着时间的推移，这样的计算到后面会越算越慢。所以数据量大的计算、复杂的计算，都不建议使用updateStateByKey。

19.宽依赖和窄依赖

1. 宽依赖：shuffle dependency，本质就是shuffle。父RDD的每一个partition中的数据，都可能会传输一部分到下一个子RDD的每一个partition中，此时会出现父RDD和子RDD的partition之间具有交互错综复杂的关系，这种情况就叫做两个RDD之间是宽依赖。
2. 窄依赖：narrow dependency，父RDD和子RDD的partition之间的对应关系是一对一的。

20.spark streaming中有状态转化操作?

DStream的有状态转化操作是跨时间区间跟踪数据的操作，即一些先前批次的数据也被用来在新的批次中计算结果。主要包括滑动窗口和updateStateByKey()，前者以一个时间阶段为滑动窗口进行操作，后者则用来跟踪每个键的状态变化。(例如构建一个代表用户会话的对象)。

有状态转化操作需要在你的Streaming Context中打开检查点机制来确保容错性。

滑动窗口：基于窗口的操作会在一个比Streaming Context的批次间隔更长的时间范围内，通过整合多个批次的结果，计算出整个窗口的结果。基于窗口的操作需要两个参数，分别为窗口时长以及滑动时长，两者都必须是Streaming Context的批次间隔的

整数倍。窗口时长控制每次计算最近的多少个批次的数据，滑动步长控制对新的DStream进行计算的间隔。

最简单的窗口操作是window()，它返回的DStream中的每个RDD会包含多个批次中的户数，可以分别进行其他transform()操作。在轨迹异常项目中，duration设置为15s，窗口函数设置为kafkadstream.windo(Durations.seconds(600),Durations.seconds(600));updateStateByKey转化操作：需要在DStream中跨批次维护状态(例如跟踪用户访问网站的会话)。针对这种情况，用于键值对形式的DStream。给定一个由(键、事件)对构成的DStream，并传递一个指定根据新的事件更新每个键对应状态的函数。

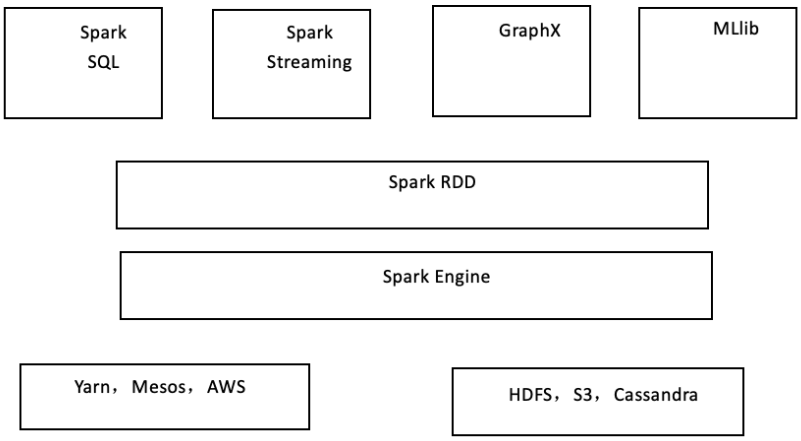
举例：在网络服务器日志中，事件可能是对网站的访问，此时键是用户的ID。使用UpdateStateByKey()可以跟踪每个用户最近访问的10个页面。这个列表就是“状态”对象，我们会在每个事件到来时更新这个状态。

21.spark常用的计算框架?

Spark Core用于离线计算，Spark SQL用于交互式查询，Spark Streaming用于实时流式计算，Spark MLlib用于机器学习，Spark GraphX用于图计算。

Spark主要用于大数据的计算，而hadoop主要用于大数据的存储(比如hdfs、hive和hbase等)，以及资源调度yarn。Spark+hadoop的组合是未来大数据领域的热门组合。

22.spark整体架构?



23.Spark的特点是什么?

(1)速度快：Spark基于内存进行计算（当然也有部分计算基于磁盘，比如shuffle）。

(2)容易上手开发：Spark的基于RDD的计算模型，比Hadoop的基于Map-Reduce的计算模型要更加易于理解，更加易于上手开发，实现各种复杂功能，比如二次排序、topn等复杂操作时，更加便捷。

(3)超强的通用性：Spark提供了Spark RDD、Spark SQL、Spark Streaming、Spark MLlib、Spark GraphX等技术组件，可以一站式地完成大数据领域的离线批处理、交互式查询、流式计算、机器学习、图计算等常见的任务。

(4)集成Hadoop：Spark并不是要成为一个大数据领域的“独裁者”，一个人霸占大数据领域所有的“地盘”，而是与Hadoop进行了高度的集成，两者可以完美的配合使用。Hadoop的HDFS、Hive、HBase负责存储，YARN负责资源调度；Spark复杂大数据计算。实际上，Hadoop+Spark的组合，是一种“double win”的组合。

(5)极高的活跃度：Spark目前是Apache基金会的顶级项目，全世界有大量的优秀工程师是Spark的committer。并且世界上很多顶级的IT公司都在大规模地使用Spark。

24.搭建spark集群步骤?

(1)安装spark包

(2)修改spark-env.sh

(3)修改slaves文件

(4)安装spark集群

(5)启动spark集群

25.Spark的三种提交模式是什么?

(1)Spark内核架构，即standalone模式，基于Spark自己的Master-Worker集群；

(2)基于Yarn的yarn-cluster模式；

(3)基于Yarn的yarn-client模式。

如果你要切换到第二种和第三种模式，将之前提交spark应用程序的spark-submit脚本，加上--master参数，设置为yarn-cluster，或yarn-client即可。如果没设置就是standalone模式。

26..spark内核架构原理

1. Master进程：主要负责资源的调度和分配，还有集群的监控等职责。

2. Driver进程：我们编写的spark程序就在Driver上由Driver进程执行。

3. Worker进程：主要负责两个，第一个是用自己的内存去存储RDD的某个或者某些partition，第二个是启动其他进程和线程，对RDD上的partition进行并行的处理和计算。

4. Executor和Task：负责执行对RDD的partition进行并行的计算，也就是执行我们对RDD定义的各种算子。

(1)将spark程序通过spark-submit(shell)命令提交到结点上执行，其实会通过反射的方式，创建和构造一个DriverActor进程出来，通过Driver进程执行我们的Application应用程序。

(2) 应用程序的第一行一般是先构造SparkConf，再构造SparkContext；

(3)SparkContext在初始化的时候，做的最重要的两件事就是构造DAGScheduler和TaskScheduler；

(4)TaskScheduler会通过对应的一个后台进程去连接Master，向Master注册Application；

(5)Master通知Worker启动executor；

(6)executor启动之后会自己反向注册到TaskSchduler，所有的executor都反向注册到Driver之后，Driver结束SparkContext初始化，然后继续执行自己编写的代码；

(7)每执行一个action就会创建一个job；

(8)DAGScheduler会根据Stage划分算法，将job划分为多个stage，并且为每个stage创建TaskSet(里面有多个task)；

(9)TaskScheduler会根据task分配算法，把TaskSet里面每一个task提交到executor上执行；

(10)executor每接收到一个task，都会用taskRunner来封装task，然后从线程池取出一个线程，执行这个task；

(11)taskRunner将我们编写的代码(算子及函数)进行拷贝，反序列化，然后执行task；

(12)task有两种，shuffleMapTask和resultTask，只有最后一个stage是resultTask，之前的都是shuffleMapTask；

(13)所以最后整个spark应用程序的执行，就是job划分为stage，然后stage分批次作为taskset提交到executor执行，每个task针对RDD的一个partition执行我们定义的算子和函数，以此类推，直到所有的操作执行完止。

27.Spark yarn-cluster架构?

Yarn-cluster用于生产环境，优点在于driver运行在NM，没有网卡流量激增的问题。缺点在于调试不方便，本地用spark-submit提交后，看不到log，只能通过yarm application-logs application_id这种命令来查看，很麻烦。

(1)将spark程序通过spark-submit命令提交，会发送请求到RM(相当于Master)，请求启动AM；

(2)在yarn集群上，RM会分配一个container，在某个NM上启动AM；

(3)在NM上会启动AM(相当于Driver)，AM会找RM请求container，启动executor；

(4)RM会分配一批container用于启动executor；

(5)AM会连接其他NM(相当于worker)，来启动executor；

(6)executor启动后，会反向注册到AM。

28.Spark yarn-client架构?

Yarn-client用于测试，因为driver运行在本地客户端，负责调度application，会与yarn集群产生大量的网络通信，从而导致网卡流量激增，可能会被公司的SA警告。好处在于，直接执行时本地可以看到所有的log，方便调试。

(1)将spark程序通过spark-submit命令提交，会发送请求到RM，请求启动AM；

(2)在yarn集群上，RM会分配一个container在某个NM上启动application；

(3)在NM上会启动application master，但是这里的AM其实只是一个ExecutorLauncher，功能很有限，只会去申请资源。AM会找RM申请container，启动executor；

(4)RM会分配一批container用于启动executor；

(5)AM会连接其他NM(相当于worker)，用container的资源来启动executor；

(6)executor启动后，会反向注册到本地的Driver进程。通过本地的Driver去执行DAGsheduler和Taskscheduler等资源调度。和Spark yarn-cluster的区别在于，cluster模式会在某一个NM上启动AM作为Driver。

29.SparkContext初始化原理?

(1) TaskScheduler如何注册application，executor如何反向注册到TaskScheduler；

(2) DAGScheduler；

(3) SparkUI。

30.Spark主备切换机制原理剖析?

1. Master实际上可以配置两个，Spark原声的standalone模式是支持Master主备切换的。当Active Master节点挂掉以后，我们可以将Standby Master切换为Active Master。

2. Spark Master主备切换可以基于两种机制，一种是基于文件系统的，一种是基于ZooKeeper的。基于文件系统的主备切换机制，需要在Active Master挂掉之后手动切换到Standby Master上；而基于Zookeeper的主备切换机制，可以实现自动切换Master。

31.spark支持故障恢复的方式?

主要包括两种方式：一种是通过血缘关系lineage，当发生故障的时候通过血缘关系，再执行一遍来一层一层恢复数据；另一种方式是通过checkpoint()机制，将数据存储到持久化存储中来恢复数据。

32.spark解决了hadoop的哪些问题?

(1)MR:抽象层次低，需要使用手工代码来完成程序编写，使用上难以上手；

Spark:Spark采用RDD计算模型，简单容易上手。

(2)MR:只提供map和reduce两个操作，表达能力欠缺；

Spark:Spark采用更加丰富的算子模型，包括map、flatmap、groupbykey、reducebykey等；

(3)MR:一个job只能包含map和reduce两个阶段，复杂的任务需要包含很多个job，这些job之间的管理以来需要开发者自己进行管理；

Spark:Spark中一个job可以包含多个转换操作，在调度时可以生成多个stage，而且如果多个map操作的分区不变，是可以放在同一个task里面去执行；

(4)MR:中间结果存放在hdfs中；

Spark:Spark的中间结果一般存在内存中，只有当内存不够了，才会存入本地磁盘，而不是hdfs；

(5)MR:只有等到所有的map task执行完毕后才能执行reduce task；

Spark:Spark中分区相同的转换构成流水线在一个task中执行，分区不同的需要进行shuffle操作，被划分成不同的stage需要等待前面的stage执行完才能执行。

(6)MR:只适合batch批处理，时延高，对于交互式处理和实时处理支持不够；

Spark:Spark streaming可以将流拆成时间间隔的batch进行处理，实时计算。

(7)MR:对于迭代式计算处理较差；

Spark:Spark将中间数据存放在内存中，提高迭代式计算性能。

总结：Spark替代hadoop，其实应该是spark替代mapreduce计算模型，因为spark本身并不提供存储，所以现在一般比较常用的大数据架构是基于spark+hadoop这样的模型。

33.数据倾斜的产生和解决办法?

数据倾斜以为着某一个或者某几个partition的数据特别大，导致这几个partition上的计算需要耗费相当长的时间。在spark中同一个应用程序划分成多个stage，这些stage之间是串行执行的，而一个stage里面的多个task是可以并行执行，task数目由partition数目决定，如果一个partition的数目特别大，那么导致这个task执行时间很长，导致接下来的stage无法执行，从而导致整个job执行变慢。

避免数据倾斜，一般是要选用合适的key，或者自己定义相关的partitioner，通过加盐或者哈希值来拆分这些key，从而将这些数据分散到不同的partition去执行。

如下算子会导致shuffle操作，是导致数据倾斜可能发生的关键点所在：groupByKey；reduceByKey；aggregaByKey；join；cogroup；

http://blog.csdn.net/u012102306/article/details/51322209

Spark streaming 空RDD判断和处理？http://www.aboutyun.com/thread-19303-1-1.html

34.spark 实现高可用性：High Availability

如果有些数据丢失，或者节点挂掉；那么不能让你的实时计算程序挂了；必须做一些数据上的冗余副本，保证你的实时计算程序可以7 * 24小时的运转。

(1).updateStateByKey、window等有状态的操作，自动进行checkpoint，必须设置checkpoint目录：容错的文件系统的目录，比如说，常用的是HDFS

SparkStreaming.checkpoint("hdfs://192.168.1.105:9090/checkpoint")

设置完这个基本的checkpoint目录之后，有些会自动进行checkpoint操作的DStream，就实现了HA高可用性；checkpoint，相当于是会把数据保留一份在容错的文件系统中，一旦内存中的数据丢失掉；那么就可以直接从文件系统中读取数据；不需要重新进行计算

(2).Driver高可用性

第一次在创建和启动StreamingContext的时候，那么将持续不断地将实时计算程序的元数据（比如说，有些dstream或者job执行到了哪个步骤），如果后面，不幸，因为某些原因导致driver节点挂掉了；那么可以让spark集群帮助我们自动重启driver，然后继续运行实时计算程序，并且是接着之前的作业继续执行；没有中断，没有数据丢失

第一次在创建和启动StreamingContext的时候，将元数据写入容错的文件系统（比如hdfs）；spark-submit脚本中加一些参数；保证在driver挂掉之后，spark集群可以自己将driver重新启动起来；而且driver在启动的时候，不会重新创建一个streaming context，而是从容错文件系统（比如hdfs）中读取之前的元数据信息，包括job的执行进度，继续接着之前的进度，继续执行。

使用这种机制，就必须使用cluster模式提交，确保driver运行在某个worker上面；但是这种模式不方便我们调试程序，一会儿还要最终测试整个程序的运行，打印不出log；我们这里仅仅是用我们的代码给大家示范一下：

```
JavaStreamingContextFactory contextFactory = new JavaStreamingContextFactory() {
  @Override
  public JavaStreamingContext create() {
    JavaStreamingContext jssc = new JavaStreamingContext(...);
    JavaDStream<String> lines = jssc.socketTextStream(...);
    jssc.checkpoint(checkpointDirectory);
    return jssc;
  }
};

JavaStreamingContext context = JavaStreamingContext.getOrCreate(checkpointDirectory, con
context.start();
context.awaitTermination();

spark-submit
--deploy-mode cluster
--supervise
```

(3).实现RDD高可用性：启动WAL预写日志机制

spark streaming，从原理上来说，是通过receiver来进行数据接收的；接收到的数据，会被划分成一个一个的block；block会被组合成一个batch；针对一个batch，会创建一个rdd；启动一个job来执行我们定义的算子操作。

receiver主要接收到数据，那么就会立即将数据写入一份到容错文件系统（比如hdfs）上的checkpoint目录中的，一份磁盘文件中去；作为数据的冗余副本。无论你的程序怎么挂掉，或者是数据丢失，那么数据都不肯能会永久性的丢失；因为肯定有副本。

WAL（Write-Ahead Log）预写日志机制

```
spark.streaming.receiver.writeAheadLog.enable true
```

35.spark实际工作中，是怎么来根据任务量，判定需要多少资源的?

```
/mydata/spark-1.6.3-bin-spark-1.6.3-2.11-withhive/bin/spark-submit \
--master yarn \
--deploy-mode client \
--num-executors 1 \
--executor-memory 7G \
--executor-cores 6 \
--conf spark.ui.port=5052 \
--conf spark.yarn.executor.memoryOverhead=1024 \
--conf spark.storage.memoryFraction=0.2 \
 --class UserAnalytics \
/mydata/dapeng/comecarsparkstreamingproject.jar
--num-executors              表示启动多少个executor来运行该作业
--executor-memory           表示每一个executor进程允许使用的内存空间
--executor-cores               在同一个executor里，最多允许多少个task可同时并发运行
--executor.memoryOverhead              用于存储已被加载的类信息、常量、静态变量等数据
--shuffle.memoryFraction 用于shuffle阶段缓存拉取到的数据所使用的内存空间
--storage.memoryFraction 用于Java堆栈的空间

/mydata/spark-1.6.3-bin-spark-1.6.3-2.11-withhive/bin/spark-submit \
--master yarn \
--deploy-mode client \
--num-executors 6 \
--executor-memory 7G \
--executor-cores 5 \
--conf spark.ui.port=5051 \
--conf spark.yarn.executor.memoryOverhead=1024 \
--conf spark.memory.storageFraction=0.2 \
--conf spark.executor.extraJavaOptions=-XX:+UseG1GC \

配置了6个executor，每个executor有5个core，每个executor有7G内存
```