# Spark性能优化

## 调优基本原则

### 基本概念和原则

首先，要搞清楚Spark的几个基本概念和原则，否则系统的性能调优无从谈起：



每一台host上面可以并行N个worker，每一个worker下面可以并行M个executor，task们会被分配到executor上面去执行。Stage指的是一组并行运行的task，stage内部是不能出现shuffle的，因为shuffle的就像篱笆一样阻止了并行task的运行，遇到shuffle就意味着到了stage的边界。

CPU的core数量，每个executor可以占用一个或多个core，可以通过观察CPU的使用率变化来了解计算资源的使用情况，例如，很常见的一种浪费是一个executor占用了多个core，但是总的CPU使用率却不高（因为一个executor并不总能充分利用多核的能力），这个时候可以考虑让一个executor占用更少的core，同时worker下面增加更多的executor，或者一台host上面增加更多的worker来增加并行执行的executor的数量，从而增加CPU利用率。但是增加executor的时候需要考虑好内存消耗，因为一台机器的内存分配给越多的executor，每个executor的内存就越小，以致出现过多的数据spill over甚至out of memory的情况。

partition和parallelism，partition指的就是数据分片的数量，每一次task只能处理一个partition的数据，这个值太小了会导致每片数据量太大，导致内存压力，或者诸多executor的计算能力无法利用充分；但是如果太大了则会导致分片太多，执行效率降低。在执行action类型操作的时候（比如各种reduce操作），partition的数量会选择parent RDD中最大的那一个。而parallelism则指的是在RDD进行reduce类操作的时候，默认返回数据的paritition数量（而在进行map类操作的时候，partition数量通常取自parent RDD中较大的一个，而且也不会涉及shuffle，因此这个parallelism的参数没有影响）。所以说，这两个概念密切相关，都是涉及到数据分片的，作用方式其实是统一的。通过spark.default.parallelism可以设置默认的分片数量，而很多RDD的操作都可以指定一个partition参数来显式控制具体的分片数量。

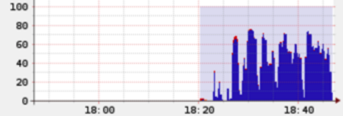
看这样几个例子：

（1）实践中跑的Spark job，有的特别慢，查看CPU利用率很低，可以尝试减少每个executor占用CPU core的数量，增加并行的executor数量，同时配合增加分片，整体上增加了CPU的利用率，加快数据处理速度。

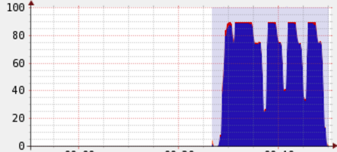
（2）发现某job很容易发生内存溢出，我们就增大分片数量，从而减少了每片数据的规模，同时还减少并行的executor数量，这样相同的内存资源分配给数量更少的executor，相当于增加了每个task的内存分配，这样运行速度可能慢了些，但是总比OOM强。

（3）数据量特别少，有大量的小文件生成，就减少文件分片，没必要创建那么多task，这种情况，如果只是最原始的input比较小，一般都能被注意到；但是，如果是在运算过程中，比如应用某个reduceBy或者某个filter以后，数据大量减少，这种低效情况就很少被留意到。

最后再补充一点，随着参数和配置的变化，性能的瓶颈是变化的，在分析问题的时候不要忘记。例如在每台机器上部署的executor数量增加的时候，性能一开始是增加的，同时也观察到CPU的平均使用率在增加；但是随着单台机器上的executor越来越多，性能下降了，因为随着executor的数量增加，被分配到每个executor的内存数量减小，在内存里直接操作的越来越少，spill over到磁盘上的数据越来越多，自然性能就变差了。   
 下面给这样一个直观的例子，当前总的cpu利用率并不高：



但是经过根据上述原则的的调整之后，可以显著发现cpu总利用率增加了：



其次，涉及性能调优我们经常要改配置，在Spark里面有三种常见的配置方式，虽然有些参数的配置是可以互相替代，但是作为最佳实践，还是需要遵循不同的情形下使用不同的配置：

1.设置环境变量，这种方式主要用于和环境、硬件相关的配置；

2.命令行参数，这种方式主要用于不同次的运行会发生变化的参数，用双横线开头；

3.代码里面（比如Scala）显式设置（SparkConf对象），这种配置通常是application级别的配置，一般不改变。

举一个配置的具体例子。slave、worker和executor之间的比例调整。我们经常需要调整并行的executor的数量，那么简单说有两种方式：

1.每个worker内始终跑一个executor，但是调整单台slave上并行的worker的数量。比如，SPARK\_WORKER\_INSTANCES可以设置每个slave的worker的数量，但是在改变这个参数的时候，比如改成2，一定要相应设置SPARK\_WORKER\_CORES的值，让每个worker使用原有一半的core，这样才能让两个worker一同工作；

2.每台slave内始终只部署一个worker，但是worker内部署多个executor。我们是在YARN框架下采用这个调整来实现executor数量改变的，一种典型办法是，一个host只跑一个worker，然后配置spark.executor.cores为host上CPU core的N分之一，同时也设置spark.executor.memory为host上分配给Spark计算内存的N分之一，这样这个host上就能够启动N个executor。

有的配置在不同的MR框架/工具下是不一样的，比如YARN下有的参数的默认取值就不同，这点需要注意。

明确这些基础的事情以后，再来一项一项看性能调优的要点。

### 性能监控方式

#### Spark Web UI

Spark提供了一些基本的Web监控页面，对于日常监控十分有用。

通过http://master:4040（默认端口是4040，可以通过spark.ui.port修改）我们可以获得运行中的程序信息：（1）stages和tasks调度情况；（2）RDD大小及内存使用；（3）系统环境信息；（4）正在执行的executor信息。

如果想当Spark应用退出后，仍可以获得历史Spark应用的stages和tasks执行信息，便于分析程序不明原因挂掉的情况。可以开启History Server。配置方法如下：

（1）$SPARK\_HOME/conf/spark-env.sh

export SPARK\_HISTORY\_OPTS="-Dspark.history.retainedApplications=50

Dspark.history.fs.logDirectory=hdfs://master01:9000/directory"

说明：spark.history.retainedApplica-tions仅显示最近50个应用spark.history.fs.logDirectory：Spark History Server页面只展示该路径下的信息。

（2）$SPARK\_HOME/conf/spark-defaults.conf

spark.eventLog.enabled true

spark.eventLog.dir hdfs://hadoop000:8020/directory #应用在运行过程中所有的信息均记录在该属性指定的路径下

spark.eventLog.compress true

（3）HistoryServer启动

$SPARK\_HOMR/bin/start-histrory-server.sh

（4）HistoryServer停止

$SPARK\_HOMR/bin/stop-histrory-server.sh

同时Executor的logs也是查看的一个出处：

Standalone模式：$SPARK\_HOME/logs

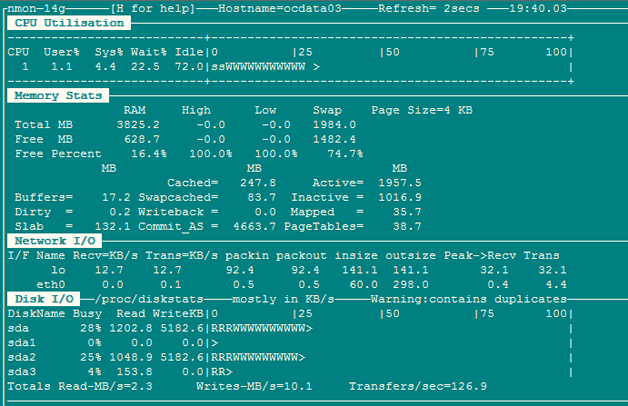
YARN模式：在yarn-site.xml文件中配置了YARN日志的存放位置：yarn.nodemanager.log-dirs，或使用命令获取yarn logs -applicationId。

同时通过配置ganglia，可以分析集群的使用状况和资源瓶颈，但是默认情况下ganglia是未被打包的，需要在mvn编译时添加-Pspark-ganglia-lgpl，并修改配置文件$SPARK\_HOME/conf/metrics.properties。

#### 其他监控工具

Nmon（http://www.ibm.com/developerworks/aix/library/au-analyze\_aix/）

Nmon 输入：c：CPU n：网络 m：内存 d：磁盘

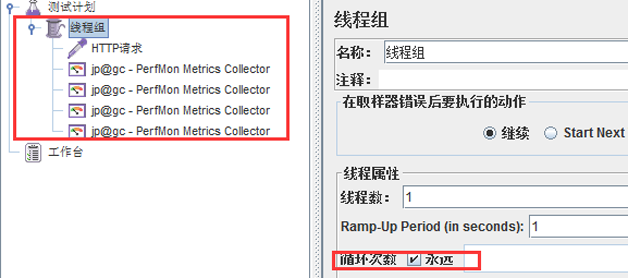


Jmeter（http://jmeter. apache.org/）

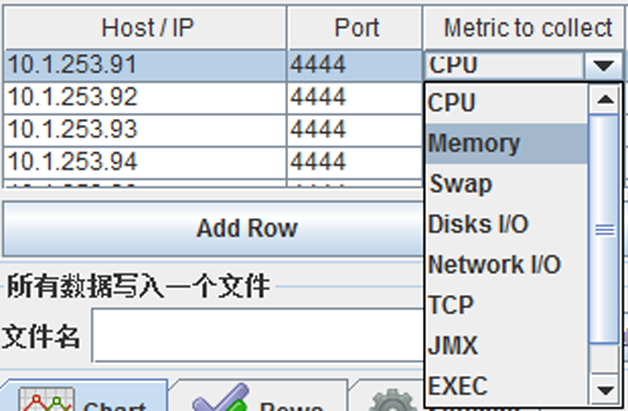
通常使用Jmeter做系统性能参数的实时展示，JMeter的安装非常简单，从官方网站上下载，解压之后即可使用。运行命令在%JMETER\_HOME%/bin下，对于 Windows 用户，直接使用jmeter.bat。

启动jmeter：创建测试计划，设置线程组设置循环次数。

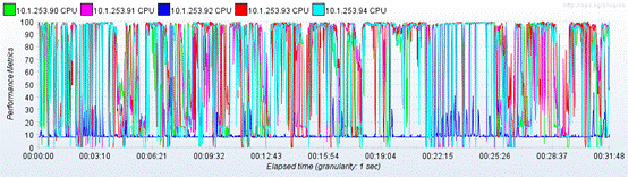
添加监听器：jp@gc - PerfMon Metrics Collector。



设置监听器：监听主机端口及监听内容，例如CPU。

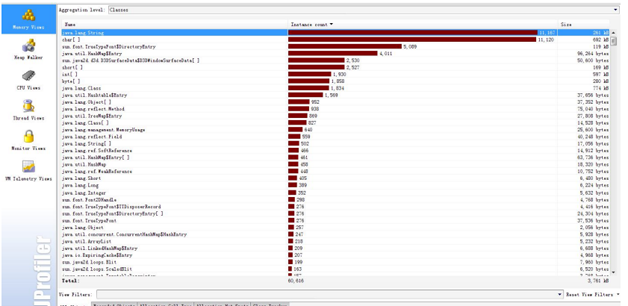


启动监听：可以实时获得节点的CPU状态信息，从下图可看出CPU已出现瓶颈。



Jprofiler（http://www.ej-technologies.com/products/jprofiler/overview.html）

JProfiler是一个全功能的Java剖析工具（profiler），专用于分析J2SE和J2EE应用程式。它把CPU、线程和内存的剖析组合在一个强大的应用中。JProfiler的GUI可以更方便地找到性能瓶颈、抓住内存泄漏（memory leaks），并解决多线程的问题。例如分析哪个对象占用的内存比较多；哪个方法占用较大的CPU资源等；我们通常使用Jprofiler来监控Spark应用在local模式下运行时的性能瓶颈和内存泄漏情况。



### 调优要点

#### 内存调整要点

Memory Tuning，Java对象会占用原始数据2~5倍甚至更多的空间。最好的检测对象内存消耗的办法就是创建RDD，然后放到cache里面去，然后在UI上面看storage的变化。使用-XX:+UseCompressedOops选项可以压缩指针（8字节变成4字节）。在调用collect等API的时候也要小心—大块数据往内存拷贝的时候心里要清楚。内存要留一些给操作系统，比如20%，这里面也包括了OS的buffercache，如果预留得太少了，会见到这样的错误：

“Required executor memory (235520+23552 MB) is above the max threshold (241664MB) of this cluster! Please increase the value of ‘yarn.scheduler.maximum-allocation-mb’.

或者干脆就没有这样的错误，但是依然有因为内存不足导致的问题，有的会有警告，比如这个：

“16/01/13 23:54:48 WARN scheduler.TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient memory

有的时候连这样的日志都见不到，而是见到一些不清楚原因的executor丢失信息：

“Exception in thread “main” org.apache.spark.SparkException: Job aborted due to stage failure: Task 12 in stage 17.0 failed 4 times, most recent failure: Lost task 12.3 in stage 17.0 (TID 1257, ip-10-184-192-56.ec2.internal): ExecutorLostFailure (executor 79 lost)

Reduce Task的内存使用。在某些情况下reduce task特别消耗内存，比如当shuffle出现的时候，比如sortByKey、groupByKey、reduceByKey和join等，要在内存里面建立一个巨大的hash table。其中一个解决办法是增大level of parallelism，这样每个task的输入规模就相应减小。另外，注意shuffle的内存上限设置，有时候有足够的内存，但是shuffle内存不够的话，性能也是上不去的。我们在有大量数据join等操作的时候，shuffle的内存上限经常配置到executor的50%。

注意原始input的大小，有很多操作始终都是需要某类全集数据在内存里面完成的，那么并非拼命增加parallelism和partition的值就可以把内存占用减得非常小的。我们遇到过某些性能低下甚至OOM的问题，是改变这两个参数所难以缓解的。但是可以通过增加每台机器的内存，或者增加机器的数量都可以直接或间接增加内存总量来解决。

另外，有一些RDD的API，比如cache，persist，都会把数据强制放到内存里面，如果并不明确这样做带来的好处，就不要用它们。

内存优化有三个方面的考虑：对象所占用的内存，访问对象的消耗以及垃圾回收所占用的开销。

1. 对象所占内存，优化数据结构

Spark 默认使用Java序列化对象，虽然Java对象的访问速度更快，但其占用的空间通常比其内部的属性数据大2-5倍。为了减少内存的使用，减少Java序列化后的额外开销，下面列举一些Spark官网提供的方法。

（1）使用对象数组以及原始类型（primitive type）数组以替代Java或者Scala集合类（collection class)。fastutil 库为原始数据类型提供了非常方便的集合类，且兼容Java标准类库。

（2）尽可能地避免采用含有指针的嵌套数据结构来保存小对象。

（3）考虑采用数字ID或者枚举类型以便替代String类型的主键。

（4）如果内存少于32GB，设置JVM参数-XX:+UseCom-pressedOops以便将8字节指针修改成4字节。与此同时，在Java 7或者更高版本，设置JVM参数-XX:+UseC-----ompressedStrings以便采用8比特来编码每一个ASCII字符。

2. 内存回收

（1）获取内存统计信息：优化内存前需要了解集群的内存回收频率、内存回收耗费时间等信息，可以在spark-env.sh中设置SPARK\_JAVA\_OPTS=“-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps $ SPARK\_JAVA\_OPTS”来获取每一次内存回收的信息。

（2）优化缓存大小：默认情况Spark采用运行内存（spark.executor.memory）的60%来进行RDD缓存。这表明在任务执行期间，有40%的内存可以用来进行对象创建。如果任务运行速度变慢且JVM频繁进行内存回收，或者内存空间不足，那么降低缓存大小设置可以减少内存消耗，可以降低spark.storage.memoryFraction的大小。

3. 频繁GC或者OOM

针对这种情况，首先要确定现象是发生在Driver端还是在Executor端，然后在分别处理。

Driver端：通常由于计算过大的结果集被回收到Driver端导致，需要调大Driver端的内存解决，或者进一步减少结果集的数量。

Executor端：

（1）以外部数据作为输入的Stage：这类Stage中出现GC通常是因为在Map侧进行map-side-combine时，由于group过多引起的。解决方法可以增加partition的数量（即task的数量）来减少每个task要处理的数据，来减少GC的可能性。

（2）以shuffle作为输入的Stage：这类Stage中出现GC的通常原因也是和shuffle有关，常见原因是某一个或多个group的数据过多，也就是所谓的数据倾斜，最简单的办法就是增加shuffle的task数量，比如在SparkSQL中设置SET spark.sql.shuffle.partitions=400，如果调大shuffle的task无法解决问题，说明你的数据倾斜很严重，某一个group的数据远远大于其他的group，需要你在业务逻辑上进行调整，预先针对较大的group做单独处理。

#### 集群并行度调整要点

在Spark集群环境下，只有足够高的并行度才能使系统资源得到充分的利用，可以通过修改spark-env.sh来调整Executor的数量和使用资源，Standalone和YARN方式资源的调度管理是不同的。

在Standalone模式下:

1. 每个节点使用的最大内存数：SPARK\_WORKER\_INSTANCES\*SPARK\_WORKER\_MEMORY；

2. 每个节点的最大并发task数：SPARK\_WORKER\_INSTANCES\*SPARK\_WORKER\_CORES。

在YARN模式下：

1. 集群task并行度：SPARK\_ EXECUTOR\_INSTANCES\* SPARK\_EXECUTOR\_CORES；

2. 集群内存总量：(executor个数) \* (SPARK\_EXECUTOR\_MEMORY+ spark.yarn.executor.memoryOverhead)  
+(SPARK\_DRIVER\_MEMORY+spark.yarn.driver.memoryOverhead)。

重点强调：Spark对Executor和Driver额外添加堆内存大小，Executor端：由spark.yarn.executor.memoryOverhead设置，默认值executorMemory \* 0.07与384的最大值。Driver端：由spark.yarn.driver.memoryOverhead设置，默认值driverMemory \* 0.07与384的最大值。

通过调整上述参数，可以提高集群并行度，让系统同时执行的任务更多，那么对于相同的任务，并行度高了，可以减少轮询次数。举例说明：如果一个stage有100task，并行度为50，那么执行完这次任务，需要轮询两次才能完成，如果并行度为100，那么一次就可以了。

但是在资源相同的情况，并行度高了，相应的Executor内存就会减少，所以需要根据实际实况协调内存和core。此外，Spark能够非常有效的支持短时间任务（例如：200ms），因为会对所有的任务复用JVM，这样能减小任务启动的消耗，Standalone模式下，core可以允许1-2倍于物理core的数量进行超配。

Level of Parallelism。指定它以后，在进行reduce类型操作的时候，默认partition的数量就被指定了。这个参数在实际工程中通常是必不可少的，一般都要根据input和每个executor内存的大小来确定。设置level of parallelism或者属性spark.default.parallelism来改变并行级别，通常来说，每一个CPU核可以分配2~3个task。

CPU core的访问模式是共享还是独占。即CPU核是被同一host上的executor共享还是瓜分并独占。比如，一台机器上共有32个CPU core的资源，同时部署了两个executor，总内存是50G，那么一种方式是配置spark.executor.cores为16，spark.executor.memory为20G，这样由于内存的限制，这台机器上会部署两个executor，每个都使用20G内存，并且各使用“独占”的16个CPU core资源；而在内存资源不变的前提下，也可以让这两个executor“共享”这32个core。根据测试，独占模式的性能要略好与共享模式。

GC调优。打印GC信息：-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps。要记得默认60%的executor内存可以被用来作为RDD的缓存，因此只有40%的内存可以被用来作为对象创建的空间，这一点可以通过设置spark.storage.memoryFraction改变。如果有很多小对象创建，但是这些对象在不完全GC的过程中就可以回收，那么增大Eden区会有一定帮助。如果有任务从HDFS拷贝数据，内存消耗有一个简单的估算公式——比如HDFS的block size是64MB，工作区内有4个task拷贝数据，而解压缩一个block要增大3倍大小，那么估算内存消耗就是：4\*3\*64MB。另外，还有一种情况：GC默认情况下有一个限制，默认是GC时间不能超过2%的CPU时间，但是如果大量对象创建（在Spark里很容易出现，代码模式就是一个RDD转下一个RDD），就会导致大量的GC时间，从而出现“OutOfMemoryError: GC overhead limit exceeded”，对于这个，可以通过设置-XX:-UseGCOverheadLimit关掉它。

#### 序列化和传输

Data Serialization，默认使用的是Java Serialization，这个程序员最熟悉，但是性能、空间表现都比较差。还有一个选项是Kryo Serialization，更快，压缩率也更高，但是并非支持任意类的序列化。在Spark UI上能够看到序列化占用总时间开销的比例，如果这个比例高的话可以考虑优化内存使用和序列化。

Broadcasting Large Variables。在task使用静态大对象的时候，可以把它broadcast出去。Spark会打印序列化后的大小，通常来说如果它超过20KB就值得这么做。有一种常见情形是，一个大表join一个小表，把小表broadcast后，大表的数据就不需要在各个node之间疯跑，安安静静地呆在本地等小表broadcast过来就好了。

Data Locality。数据和代码要放到一起才能处理，通常代码总比数据要小一些，因此把代码送到各处会更快。Data Locality是数据和处理的代码在屋里空间上接近的程度：PROCESS\_LOCAL（同一个JVM）、NODE\_LOCAL（同一个node，比如数据在HDFS上，但是和代码在同一个node）、NO\_PREF、RACK\_LOCAL（不在同一个server，但在同一个机架）、ANY。当然优先级从高到低，但是如果在空闲的executor上面没有未处理数据了，那么就有两个选择：

（1）要么等如今繁忙的CPU闲下来处理尽可能“本地”的数据，

（2）要么就不等直接启动task去处理相对远程的数据。

默认当这种情况发生Spark会等一会儿（spark.locality），即策略（1），如果繁忙的CPU停不下来，就会执行策略（2）。

代码里对大对象的引用。在task里面引用大对象的时候要小心，因为它会随着task序列化到每个节点上去，引发性能问题。只要序列化的过程不抛出异常，引用对象序列化的问题事实上很少被人重视。如果，这个大对象确实是需要的，那么就不如干脆把它变成RDD好了。绝大多数时候，对于大对象的序列化行为，是不知不觉发生的，或者说是预期之外的，比如在我们的项目中有这样一段代码：

|  |
| --- |
| rdd.map(r => {  *println*(BackfillTypeIndex) }) |

其实呢，它等价于这样：

|  |
| --- |
| rdd.map(r => {  println(**this**.BackfillTypeIndex) }) |

不要小看了这个this，有时候它的序列化是非常大的开销。

对于这样的问题，一种最直接的解决方法就是：

|  |
| --- |
| **val** *dereferencedVariable* = **this**.BackfillTypeIndex rdd.map(r => *println*(*dereferencedVariable*)) *// "this" is not serialized* |

相关地，注解@transient用来标识某变量不要被序列化，这对于将大对象从序列化的陷阱中排除掉是很有用的。另外，注意class之间的继承层级关系，有时候一个小的case class可能来自一棵大树。

#### 文件读写

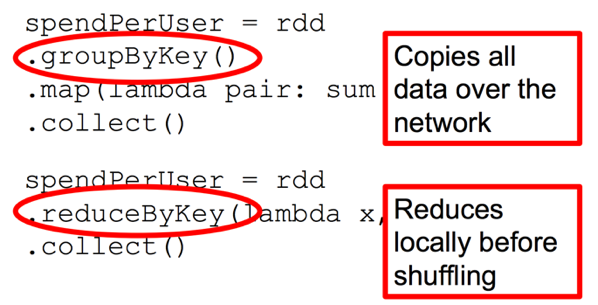
文件存储和读取的优化。比如对于一些case而言，如果只需要某几列，使用rcfile和parquet这样的格式会大大减少文件读取成本。再有就是存储文件到S3上或者HDFS上，可以根据情况选择更合适的格式，比如压缩率更高的格式。另外，特别是对于shuffle特别多的情况，考虑留下一定量的额外内存给操作系统作为操作系统的buffer cache，比如总共50G的内存，JVM最多分配到40G多一点。

文件分片。比如在S3上面就支持文件以分片形式存放，后缀是partXX。使用coalesce方法来设置分成多少片，这个调整成并行级别或者其整数倍可以提高读写性能。但是太高太低都不好，太低了没法充分利用S3并行读写的能力，太高了则是小文件太多，预处理、合并、连接建立等等都是时间开销啊，读写还容易超过throttle。

#### 任务调整要点

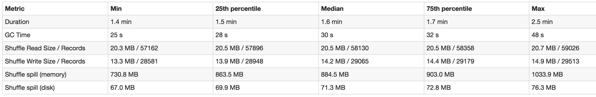
Spark的Speculation。通过设置spark.speculation等几个相关选项，可以让Spark在发现某些task执行特别慢的时候，可以在不等待完成的情况下被重新执行，最后相同的task只要有一个执行完了，那么最快执行完的那个结果就会被采纳。

减少Shuffle。其实Spark的计算往往很快，但是大量开销都花在网络和IO上面，而shuffle就是一个典型。举个例子，如果(k, v1) join (k, v2) => (k, v3)，那么，这种情况其实Spark是优化得非常好的，因为需要join的都在一个node的一个partition里面，join很快完成，结果也是在同一个node（这一系列操作可以被放在同一个stage里面）。但是如果数据结构被设计为(obj1) join (obj2) => (obj3)，而其中的join条件为obj1.column1 == obj2.column1，这个时候往往就被迫shuffle了，因为不再有同一个key使得数据在同一个node上的强保证。在一定要shuffle的情况下，尽可能减少shuffle前的数据规模，比如这个避免groupByKey的例子。下面这个比较的图片来自Spark Summit 2013的一个演讲，讲的是同一件事情：



Repartition。运算过程中数据量时大时小，选择合适的partition数量关系重大，如果太多partition就导致有很多小任务和空任务产生；如果太少则导致运算资源没法充分利用，必要时候可以使用repartition来调整，不过它也不是没有代价的，其中一个最主要代价就是shuffle。再有一个常见问题是数据大小差异太大，这种情况主要是数据的partition的key其实取值并不均匀造成的（默认使用HashPartitioner），需要改进这一点，比如重写hash算法。测试的时候想知道partition的数量可以调用rdd.partitions().size()获知。

Task时间分布。关注Spark UI，在Stage的详情页面上，可以看得到shuffle写的总开销，GC时间，当前方法栈，还有task的时间花费。如果你发现task的时间花费分布太散，就是说有的花费时间很长，有的很短，这就说明计算分布不均，需要重新审视数据分片、key的hash、task内部的计算逻辑等等，瓶颈出现在耗时长的task上面。



重用资源。有的资源申请开销巨大，而且往往相当有限，比如建立连接，可以考虑在partition建立的时候就创建好（比如使用mapPartition方法），这样对于每个partition内的每个元素的操作，就只要重用这个连接就好了，不需要重新建立连接。

同时Spark的任务数量是由stage中的起始的所有RDD的partition之和数量决定，所以需要了解每个RDD的partition的计算方法。以Spark应用从HDFS读取数据为例，HadoopRDD的partition切分方法完全继承于MapReduce中的FileInputFormat，具体的partition数量由HDFS的块大小、mapred.min.split.size的大小、文件的压缩方式等多个因素决定，详情需要参见FileInputFormat的代码。

#### 开启推测机制

推测机制后，如果集群中，某一台机器的几个task特别慢，推测机制会将任务分配到其他机器执行，最后Spark会选取最快的作为最终结果。

在spark-default.conf 中添加：spark.speculation true

推测机制与以下几个参数有关：

1. spark.speculation.interval 100：检测周期，单位毫秒；

2. spark.speculation.quantile 0.75：完成task的百分比时启动推测；

3. spark.speculation.multiplier 1.5：比其他的慢多少倍时启动推测。

## 数据倾斜优化

### 为何要处理数据倾斜（Data Skew）

什么是数据倾斜

对Spark/Hadoop这样的大数据系统来讲，数据量大并不可怕，可怕的是数据倾斜。

何谓数据倾斜？数据倾斜指的是，并行处理的数据集中，某一部分（如Spark或Kafka的一个Partition）的数据显著多于其它部分，从而使得该部分的处理速度成为整个数据集处理的瓶颈。

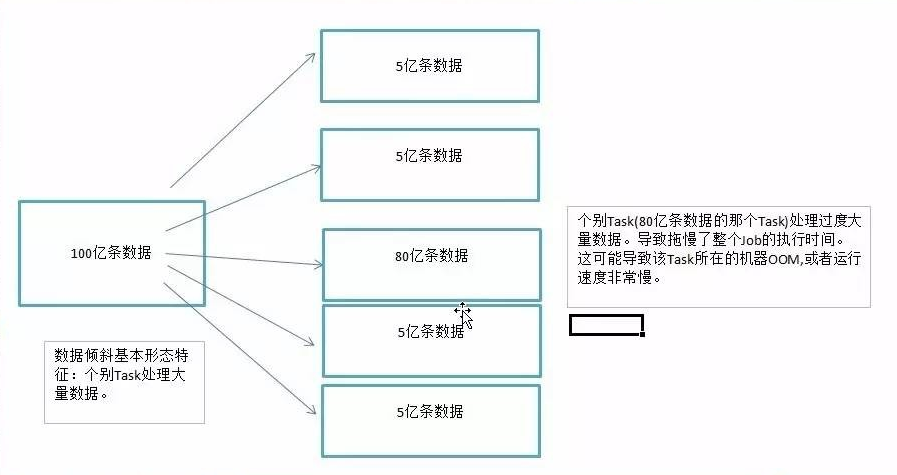
如果数据倾斜没有解决，完全没有可能进行性能调优，其他所有的调优手段都是一个笑话。数据倾斜是最能体现一个spark大数据工程师水平的性能调优问题。

数据倾斜如果能够解决的话，代表对spark运行机制了如指掌。

数据倾斜俩大直接致命后果。

1 数据倾斜直接会导致一种情况：Out Of Memory。

2 运行速度慢,特别慢，非常慢，极端的慢，不可接受的慢。



我们以100亿条数据为列子。

个别Task(80亿条数据的那个Task)处理过度大量数据。导致拖慢了整个Job的执行时间。这可能导致该Task所在的机器OOM,或者运行速度非常慢。

数据倾斜是如何造成的

在Shuffle阶段。同样Key的数据条数太多了。导致了某个key(上图中的80亿条)所在的Task数据量太大了。远远超过其他Task所处理的数据量。

而这样的场景太常见了。二八定律可以证实这种场景。

搞定数据倾斜需要：

1 搞定shuffle

2 搞定业务场景

3 搞定 cpu core的使用情况

4 搞定OOM的根本原因等。

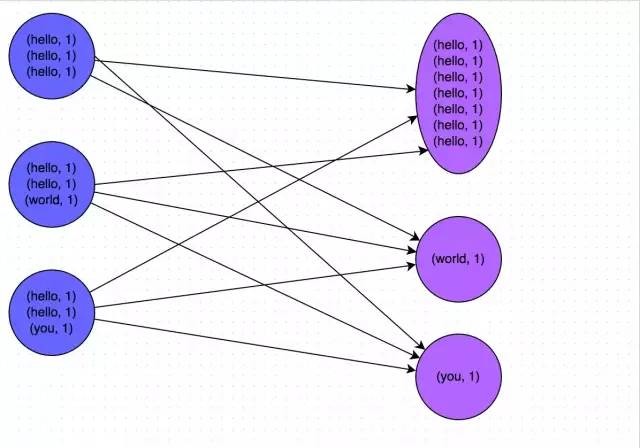
所以搞定了数据倾斜需要对至少以上的原理了如指掌。所以搞定数据倾斜是关键中的关键。

一个经验结论是：一般情况下，OOM的原因都是数据倾斜。某个task任务数据量太大，GC的压力就很大。这比不了Kafka,因为kafka的内存是不经过JVM的。是基于Linux内核的Page.

数据倾斜的原理很简单：在进行shuffle的时候，必须将各个节点上相同的key拉取到某个节点上的一个task来进行处理，比如按照key进行聚合或join等操作。此时如果某个key对应的数据量特别大的话，就会发生数据倾斜。比如大部分key对应10条数据，但是个别key却对应了100万条数据，那么大部分task可能就只会分配到10条数据，然后1秒钟就运行完了；但是个别task可能分配到了100万数据，要运行一两个小时。因此，整个Spark作业的运行进度是由运行时间最长的那个task决定的。

因此出现数据倾斜的时候，Spark作业看起来会运行得非常缓慢，甚至可能因为某个task处理的数据量过大导致内存溢出。

下图就是一个很清晰的例子：hello这个key，在三个节点上对应了总共7条数据，这些数据都会被拉取到同一个task中进行处理；而world和you这两个key分别才对应1条数据，所以另外两个task只要分别处理1条数据即可。此时第一个task的运行时间可能是另外两个task的7倍，而整个stage的运行速度也由运行最慢的那个task所决定。



由于同一个Stage内的所有Task执行相同的计算，在排除不同计算节点计算能力差异的前提下，不同Task之间耗时的差异主要由该Task所处理的数据量决定。

### 如何定位导致数据倾斜的代码

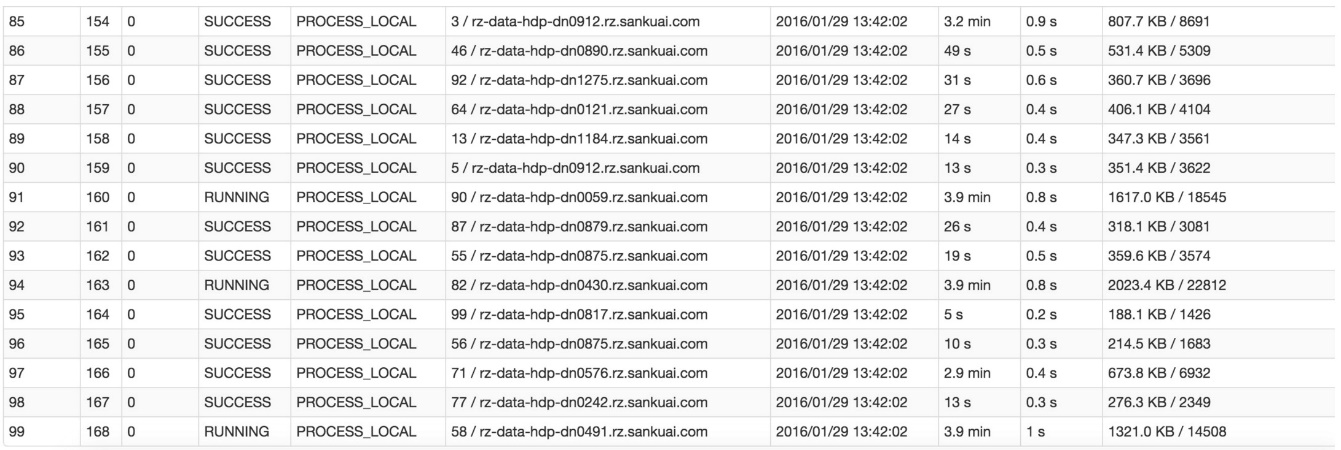
数据倾斜只会发生在shuffle过程中。这里给大家罗列一些常用的并且可能会触发shuffle操作的算子：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。出现数据倾斜时，可能就是你的代码中使用了这些算子中的某一个所导致的。

***某个task执行特别慢的情况***

首先要看的，就是数据倾斜发生在第几个stage中。

可以通过Spark Web UI来查看当前运行到了第几个stage,看一下当前这个stage各个task分配的数据量，从而进一步确定是不是task分配的数据不均匀导致了数据倾斜。

比如下图中，倒数第三列显示了每个task的运行时间。明显可以看到，有的task运行特别快，只需要几秒钟就可以运行完；而有的task运行特别慢，需要几分钟才能运行完，此时单从运行时间上看就已经能够确定发生数据倾斜了。此外，倒数第一列显示了每个task处理的数据量，明显可以看到，运行时间特别短的task只需要处理几百KB的数据即可，而运行时间特别长的task需要处理几千KB的数据，处理的数据量差了10倍。此时更加能够确定是发生了数据倾斜。



知道数据倾斜发生在哪一个stage之后，接着我们就需要根据stage划分原理，推算出来发生倾斜的那个stage对应代码中的哪一部分，这部分代码中肯定会有一个shuffle类算子。精准推算stage与代码的对应关系，这里介绍一个相对简单实用的推算方法：只要看到Spark代码中出现了一个shuffle类算子或者是Spark SQL的SQL语句中出现了会导致shuffle的语句（比如group by语句），那么就可以判定，以那个地方为界限划分出了前后两个stage。

这里我们就以Spark最基础的入门程序——单词计数来举例，如何用最简单的方法大致推算出一个stage对应的代码。如下示例，在整个代码中，只有一个reduceByKey是会发生shuffle的算子，因此就可以认为，以这个算子为界限，会划分出前后两个stage。

stage0，主要是执行从textFile到map操作，以及执行shuffle write操作。shuffle write操作，我们可以简单理解为对pairs RDD中的数据进行分区操作，每个task处理的数据中，相同的key会写入同一个磁盘文件内。

stage1，主要是执行从reduceByKey到collect操作，stage1的各个task一开始运行，就会首先执行shuffle read操作。执行shuffle read操作的task，会从stage0的各个task所在节点拉取属于自己处理的那些key，然后对同一个key进行全局性的聚合或join等操作，在这里就是对key的value值进行累加。stage1在执行完reduceByKey算子之后，就计算出了最终的wordCounts RDD，然后会执行collect算子，将所有数据拉取到Driver上，供我们遍历和打印输出。

|  |
| --- |
| **val** *conf* = **new** SparkConf() **val** *sc* = **new** SparkContext(*conf*) **val** *lines* = *sc*.textFile(**"hdfs://..."**) **val** *words* = *lines*.flatMap(\_.split(**" "**)) **val** *pairs* = *words*.map((\_, 1)) **val** *wordCounts* = *pairs*.reduceByKey(\_ + \_) *wordCounts*.collect().foreach(*println*(\_)) |

通过对单词计数程序的分析，希望能够让大家了解最基本的stage划分的原理，以及stage划分后shuffle操作是如何在两个stage的边界处执行的。然后我们就知道如何快速定位出发生数据倾斜的stage对应代码的哪一个部分了。比如我们在Spark Web UI或者本地log中发现，stage1的某几个task执行得特别慢，判定stage1出现了数据倾斜，那么就可以回到代码中定位出stage1主要包括了reduceByKey这个shuffle类算子，此时基本就可以确定是由reduceByKey算子导致的数据倾斜问题。比如某个单词出现了100万次，其他单词才出现10次，那么stage1的某个task就要处理100万数据，整个stage的速度就会被这个task拖慢。

***某个task莫名其妙内存溢出的情况***

这种情况下去定位出问题的代码就比较容易了。我们建议直接看yarn-client模式下本地log的异常栈，或者是通过YARN查看yarn-cluster模式下的log中的异常栈。一般来说，通过异常栈信息就可以定位到你的代码中哪一行发生了内存溢出。然后在那行代码附近找找，一般也会有shuffle类算子，此时很可能就是这个算子导致了数据倾斜。

但是大家要注意的是，不能单纯靠偶然的内存溢出就判定发生了数据倾斜。因为自己编写的代码的bug，以及偶然出现的数据异常，也可能会导致内存溢出。因此还是要按照上面所讲的方法，通过Spark Web UI查看报错的那个stage的各个task的运行时间以及分配的数据量，才能确定是否是由于数据倾斜才导致了这次内存溢出。

***查看导致数据倾斜的key的数据分布情况***

知道了数据倾斜发生在哪里之后，通常需要分析一下那个执行了shuffle操作并且导致了数据倾斜的RDD/Hive表，查看一下其中key的分布情况。这主要是为之后选择哪一种技术方案提供依据。针对不同的key分布与不同的shuffle算子组合起来的各种情况，可能需要选择不同的技术方案来解决。

此时根据你执行操作的情况不同，可以有很多种查看key分布的方式：

如果是Spark SQL中的group by、join语句导致的数据倾斜，那么就查询一下SQL中使用的表的key分布情况。

如果是对Spark RDD执行shuffle算子导致的数据倾斜，那么可以在Spark作业中加入查看key分布的代码，比如RDD.countByKey()。然后对统计出来的各个key出现的次数，collect/take到客户端打印一下，就可以看到key的分布情况。

举例来说，对于上面所说的单词计数程序，如果确定了是stage1的reduceByKey算子导致了数据倾斜，那么就应该看看进行reduceByKey操作的RDD中的key分布情况，在这个例子中指的就是pairs RDD。如下示例，我们可以先对pairs采样10%的样本数据，然后使用countByKey算子统计出每个key出现的次数，最后在客户端遍历和打印样本数据中各个key的出现次数。

|  |
| --- |
| **val** *sampledPairs* = pairs.sample(**false**, 0.1) **val** *sampledWordCounts* = *sampledPairs*.countByKey() *sampledWordCounts*.foreach(*println*(\_)) |

### 如何缓解/消除数据倾斜

#### 尽量避免数据源的数据倾斜

***比如数据源是Kafka：***

以Spark Stream通过DirectStream方式读取Kafka数据为例。由于Kafka的每一个Partition对应Spark的一个Task（Partition），所以Kafka内相关Topic的各Partition之间数据是否平衡，直接决定Spark处理该数据时是否会产生数据倾斜。

Kafka某一Topic内消息在不同Partition之间的分布，主要由Producer端所使用的Partition实现类决定。如果使用随机Partitioner，则每条消息会随机发送到一个Partition中，从而从概率上来讲，各Partition间的数据会达到平衡。此时源Stage（直接读取Kafka数据的Stage）不会产生数据倾斜。

但很多时候，业务场景可能会要求将具备同一特征的数据顺序消费，此时就需要将具有相同特征的数据放于同一个Partition中。一个典型的场景是，需要将同一个用户相关的PV信息置于同一个Partition中。此时，如果产生了数据倾斜，则需要通过其它方式处理。

***比如数据源是Hive：***

导致数据倾斜的是Hive表。如果该Hive表中的数据本身很不均匀（比如某个key对应了100万数据，其他key才对应了10条数据），而且业务场景需要频繁使用Spark对Hive表执行某个分析操作，那么比较适合使用这种技术方案。

方案实现思路：此时可以评估一下，是否可以通过Hive来进行数据预处理（即通过Hive ETL预先对数据按照key进行聚合，或者是预先和其他表进行join），然后在Spark作业中针对的数据源就不是原来的Hive表了，而是预处理后的Hive表。此时由于数据已经预先进行过聚合或join操作了，那么在Spark作业中也就不需要使用原先的shuffle类算子执行这类操作了。

方案实现原理：这种方案从根源上解决了数据倾斜，因为彻底避免了在Spark中执行shuffle类算子，那么肯定就不会有数据倾斜的问题了。但是这里也要提醒一下大家，这种方式属于治标不治本。因为毕竟数据本身就存在分布不均匀的问题，所以Hive ETL中进行group by或者join等shuffle操作时，还是会出现数据倾斜，导致Hive ETL的速度很慢。我们只是把数据倾斜的发生提前到了Hive ETL中，避免Spark程序发生数据倾斜而已。

方案优点：实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark作业的性能会大幅度提升。

方案缺点：治标不治本，Hive ETL中还是会发生数据倾斜。

方案实践经验：在一些Java系统与Spark结合使用的项目中，会出现Java代码频繁调用Spark作业的场景，而且对Spark作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次Java调用Spark作业时，执行速度都会很快，能够提供更好的用户体验。

项目实践经验：在美团·点评的交互式用户行为分析系统中使用了这种方案，该系统主要是允许用户通过Java Web系统提交数据分析统计任务，后端通过Java提交Spark作业进行数据分析统计。要求Spark作业速度必须要快，尽量在10分钟以内，否则速度太慢，用户体验会很差。所以我们将有些Spark作业的shuffle操作提前到了Hive ETL中，从而让Spark直接使用预处理的Hive中间表，尽可能地减少Spark的shuffle操作，大幅度提升了性能，将部分作业的性能提升了6倍以上。

#### 调整并行度分散同一个Task的不同Key

方案适用场景：如果我们必须要对数据倾斜迎难而上，那么建议优先使用这种方案，因为这是处理数据倾斜最简单的一种方案。

方案实现思路：在对RDD执行shuffle算子时，给shuffle算子传入一个参数，比如reduceByKey(1000)，该参数就设置了这个shuffle算子执行时shuffle read task的数量。对于Spark SQL中的shuffle类语句，比如group by、join等，需要设置一个参数，即spark.sql.shuffle.partitions，该参数代表了shuffle read task的并行度，该值默认是200，对于很多场景来说都有点过小。

方案实现原理：增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了。具体原理如下图所示。

方案优点：实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。

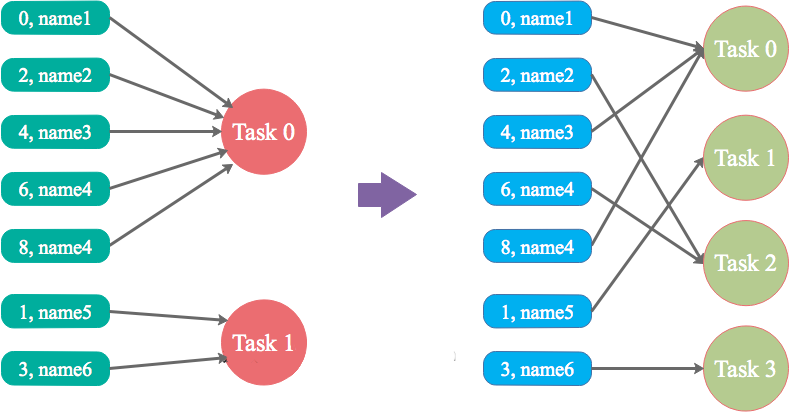
方案缺点：只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限。

方案实践经验：该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，这个对应着100万数据的key肯定还是会分配到一个task中去处理，因此注定还是会发生数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用最简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。

原理

Spark在做Shuffle时，默认使用HashPartitioner（非Hash Shuffle）对数据进行分区。如果并行度设置的不合适，可能造成大量不相同的Key对应的数据被分配到了同一个Task上，造成该Task所处理的数据远大于其它Task，从而造成数据倾斜。

如果调整Shuffle时的并行度，使得原本被分配到同一Task的不同Key发配到不同Task上处理，则可降低原Task所需处理的数据量，从而缓解数据倾斜问题造成的短板效应。



案例

现有一张测试数据集，内有100万条数据，每条数据有一个唯一的id值。现通过一些处理，使得id为90万之下的所有数据对12取模后余数为8（即在Shuffle并行度为12时该数据集全部被HashPartition分配到第8个Task），其它数据集id不变，从而使得id大于90万的数据在Shuffle时可被均匀分配到所有Task中，而id小于90万的数据全部分配到同一个Task中。处理过程如下

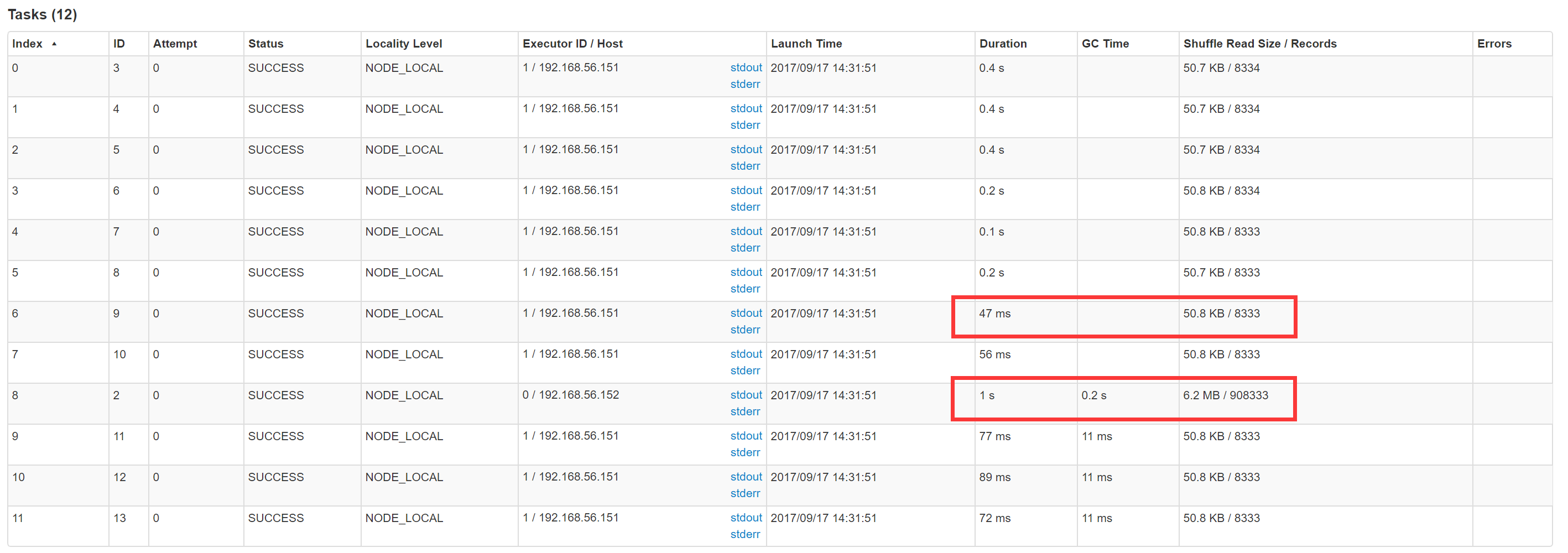
|  |
| --- |
| scala> val sourceRdd = sc.textFile("hdfs://master01:9000/source\_index")  sourceRdd: org.apache.spark.rdd.RDD[String] = hdfs://master01:9000/source\_index MapPartitionsRDD[1] at textFile at <console>:24  scala> case class brower(id:Int, time:Long, uid:String, keyword:String, url\_rank:Int, click\_num:Int, click\_url:String) extends Serializable  defined class brower  scala> val ds = sourceRdd.map(\_.split("\t")).map(attr => brower(attr(0).toInt, attr(1).toLong, attr(2), attr(3), attr(4).toInt, attr(5).toInt, attr(6))).toDS  ds: org.apache.spark.sql.Dataset[brower] = [id: int, time: bigint ... 5 more fields]  scala> ds.createOrReplaceTempView("sourceTable")  scala> val newSource = spark.sql("SELECT CASE WHEN id < 900000 THEN (8 + (CAST (RAND() \* 50000 AS bigint)) \* 12 ) ELSE id END, time, uid, keyword, url\_rank, click\_num, click\_url FROM sourceTable")  newSource: org.apache.spark.sql.DataFrame = [CASE WHEN (id < 900000) THEN (CAST(8 AS BIGINT) + (CAST((rand(-5486683549522524104) \* CAST(50000 AS DOUBLE)) AS BIGINT) \* CAST(12 AS BIGINT))) ELSE CAST(id AS BIGINT) END: bigint, time: bigint ... 5 more fields]  scala> newSource.rdd.map(\_.mkString("\t")).saveAsTextFile("hdfs://master01:9000/test\_data") |

通过上述处理，一份可能造成后续数据倾斜的测试数据即以准备好。接下来，使用Spark读取该测试数据，并通过groupByKey(12)对id分组处理，且Shuffle并行度为12。代码如下

|  |
| --- |
| scala> val sourceRdd = sc.textFile("hdfs://master01:9000/test\_data/p\*")  sourceRdd: org.apache.spark.rdd.RDD[String] = hdfs://master01:9000/test\_data/p\* MapPartitionsRDD[1] at textFile at <console>:24  scala> val kvRdd = sourceRdd.map(x =>{ val parm=x.split("\t");(parm(0).trim().toInt,parm(1).trim()) })  kvRdd: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[2] at map at <console>:26  scala> kvRdd.groupByKey(12).count  res0: Long = 150000  scala> :quit |

本次实验所使用集群节点数为3，每个节点可被Yarn使用的CPU核数为3，内存为2GB。在Spark-shell中进行提交

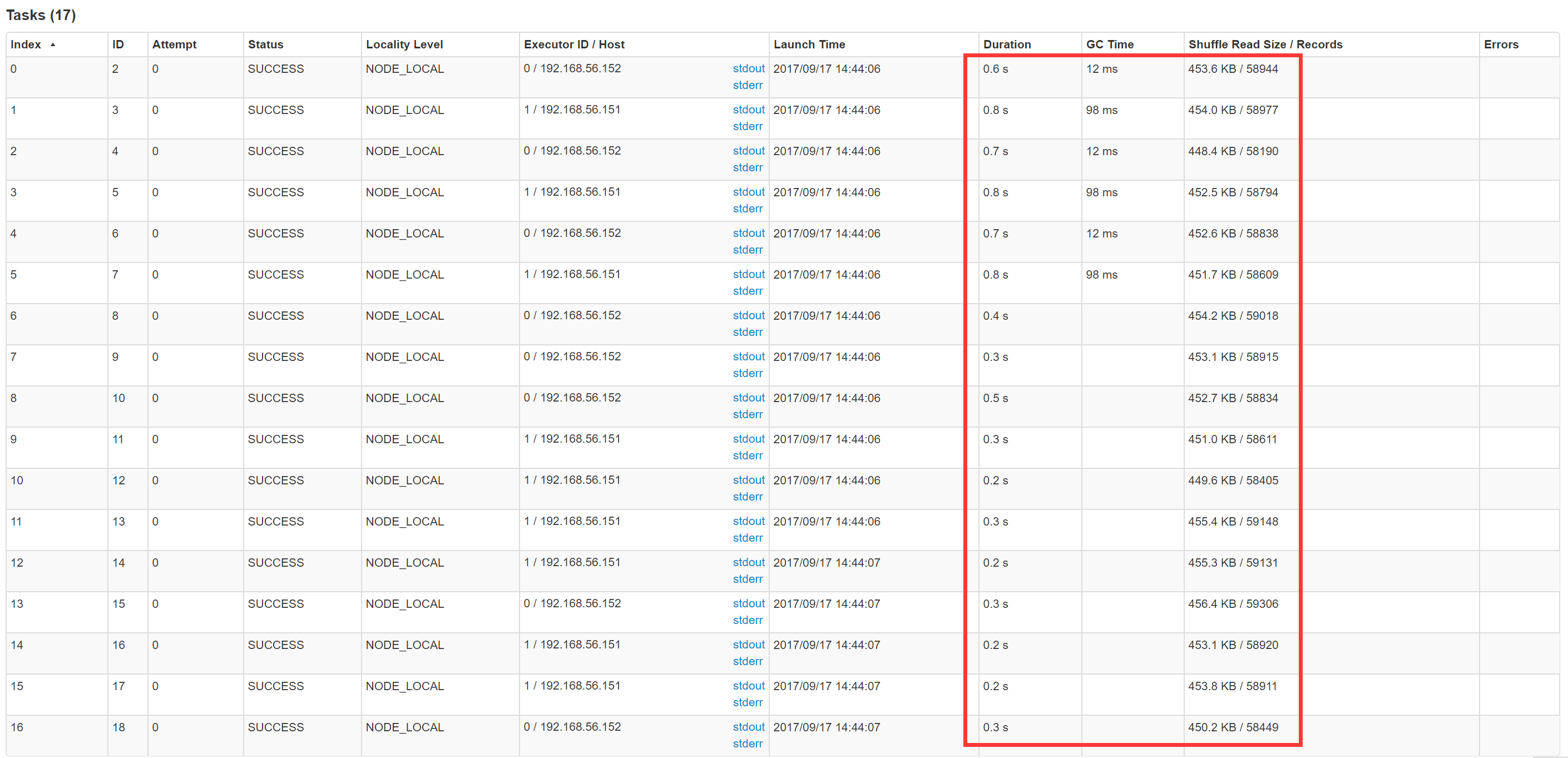
GroupBy Stage的Task状态如下图所示，Task 8处理的记录数为90万，远大于（9倍于）其它11个Task处理的10万记录。而Task 8所耗费的时间为1秒，远高于其它11个Task的平均时间。整个Stage的时间也为1秒，该时间主要由最慢的Task 8决定。数据之间处理的比例最大为105倍。



在这种情况下，可以通过调整Shuffle并行度，使得原来被分配到同一个Task（即该例中的Task 8）的不同Key分配到不同Task，从而降低Task 8所需处理的数据量，缓解数据倾斜。

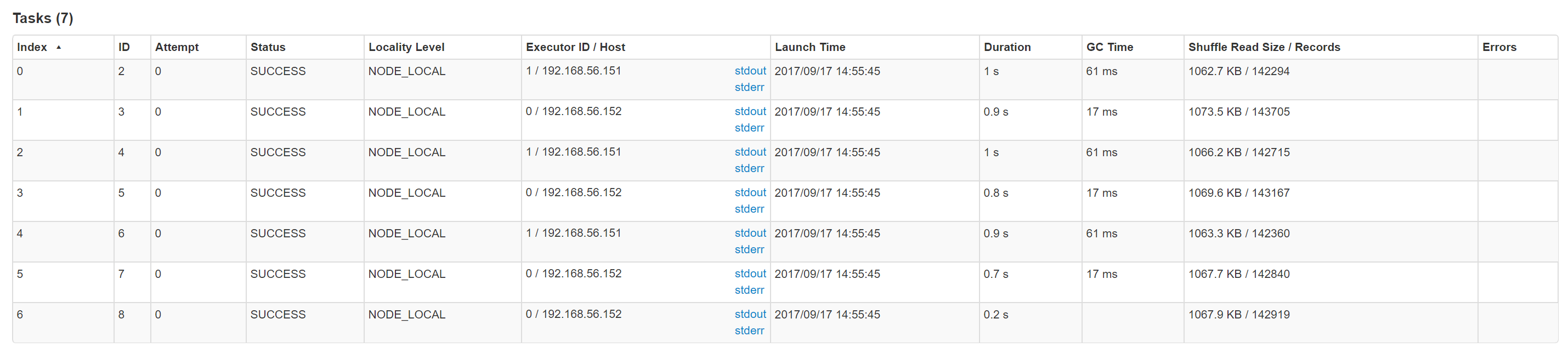
通过groupByKey(17)将Shuffle并行度调整为17，重新提交到Spark。新的Job的GroupBy Stage所有Task状态如下图所示。

|  |
| --- |
| scala> val sourceRdd = sc.textFile("hdfs://master01:9000/test\_data/p\*")  sourceRdd: org.apache.spark.rdd.RDD[String] = hdfs://master01:9000/test\_data/p\* MapPartitionsRDD[1] at textFile at <console>:24  scala> val kvRdd = sourceRdd.map(x =>{ val parm=x.split("\t");(parm(0).trim().toInt,parm(1).trim()) })  kvRdd: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[2] at map at <console>:26  scala> kvRdd.groupByKey(17).count  res0: Long = 150000  scala> :quit |



从上图可知，相比以上次一计算，目前每一个计算的数据都比较平均，数据之间的最大比例基本为1:1，总体时间降到了0.8秒。

在这种场景下，调整并行度，并不意味着一定要增加并行度，也可能是减小并行度。如果通过groupByKey(7)将Shuffle并行度调整为7，重新提交到Spark。新Job的GroupBy Stage的所有Task状态如下图所示。



从上图可见，处理记录数都比较平均。

总结

适用场景

大量不同的Key被分配到了相同的Task造成该Task数据量过大。

解决方案

调整并行度。一般是增大并行度，但有时如本例减小并行度也可达到效果。

优势

实现简单，可在需要Shuffle的操作算子上直接设置并行度或者使用spark.default.parallelism设置。如果是Spark SQL，还可通过SET spark.sql.shuffle.partitions=[num\_tasks]设置并行度。可用最小的代价解决问题。一般如果出现数据倾斜，都可以通过这种方法先试验几次，如果问题未解决，再尝试其它方法。

劣势

适用场景少，只能将分配到同一Task的不同Key分散开，但对于同一Key倾斜严重的情况该方法并不适用。并且该方法一般只能缓解数据倾斜，没有彻底消除问题。从实践经验来看，其效果一般。

#### 自定义Partitioner

原理

使用自定义的Partitioner（默认为HashPartitioner），将原本被分配到同一个Task的不同Key分配到不同Task。

案例

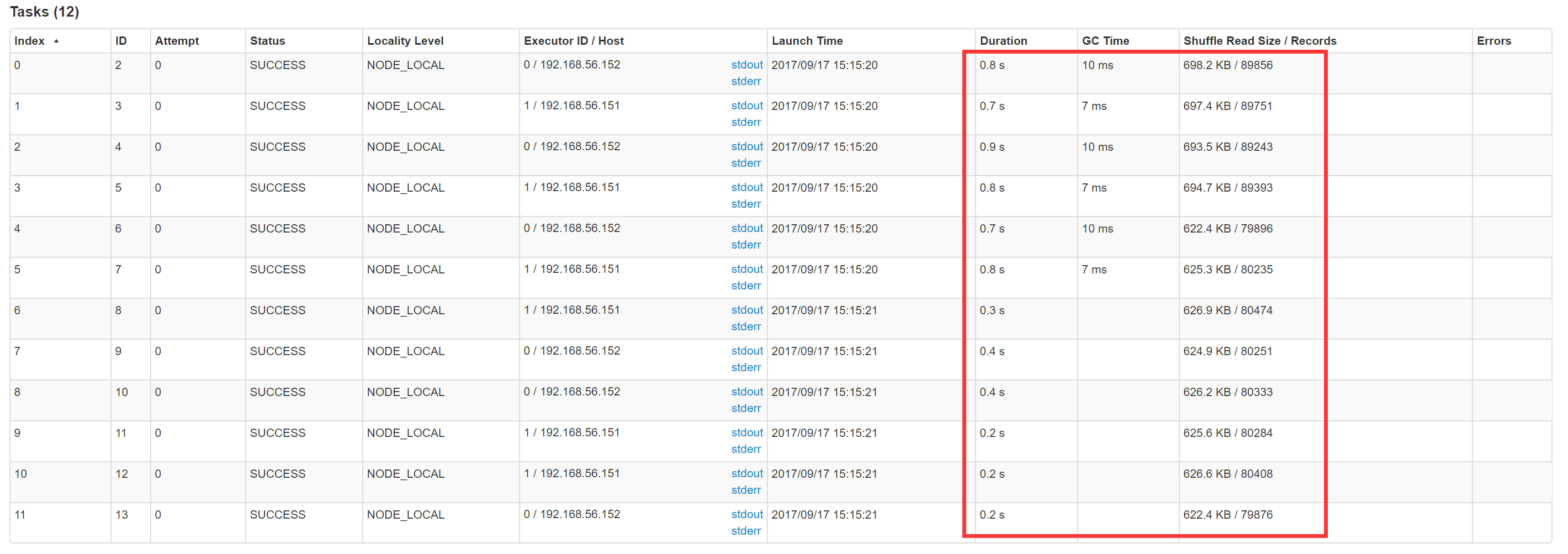
以上述数据集为例，继续将并发度设置为12，但是在groupByKey算子上，使用自定义的Partitioner（实现如下）

|  |
| --- |
| class CustomerPartitioner(numParts:Int) extends org.apache.spark.Partitioner {   //覆盖分区数  override def numPartitions: Int = numParts   //覆盖分区号获取函数  override def getPartition(key: Any): Int = {  val id: Int = key.toString.toInt  if (id <= 900000)  return new java.util.Random().nextInt(100) % 12  else  return id % 12  } } |

执行如下代码：

|  |
| --- |
| scala> :paste  // Entering paste mode (ctrl-D to finish)  class CustomerPartitioner(numParts:Int) extends org.apache.spark.Partitioner {  //覆盖分区数  override def numPartitions: Int = numParts  //覆盖分区号获取函数  override def getPartition(key: Any): Int = {  val id: Int = key.toString.toInt  if (id <= 900000)  return new java.util.Random().nextInt(100) % 12  else  return id % 12  }  }  // Exiting paste mode, now interpreting.  defined class CustomerPartitioner  scala> val sourceRdd = sc.textFile("hdfs://master01:9000/test\_data/p\*")  sourceRdd: org.apache.spark.rdd.RDD[String] = hdfs://master01:9000/test\_data/p\* MapPartitionsRDD[1] at textFile at <console>:24  scala> val kvRdd = sourceRdd.map(x =>{ val parm=x.split("\t");(parm(0).trim().toInt,parm(1).trim()) })  kvRdd: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[2] at map at <console>:26  scala> kvRdd.groupByKey(new CustomerPartitioner(12)).count  res0: Long = 565312  scala> :quit |

由下图可见，使用自定义Partition后，各Task所处理的数据集大小相当。



总结

适用场景

大量不同的Key被分配到了相同的Task造成该Task数据量过大。

解决方案

使用自定义的Partitioner实现类代替默认的HashPartitioner，尽量将所有不同的Key均匀分配到不同的Task中。

优势

不影响原有的并行度设计。如果改变并行度，后续Stage的并行度也会默认改变，可能会影响后续Stage。

劣势

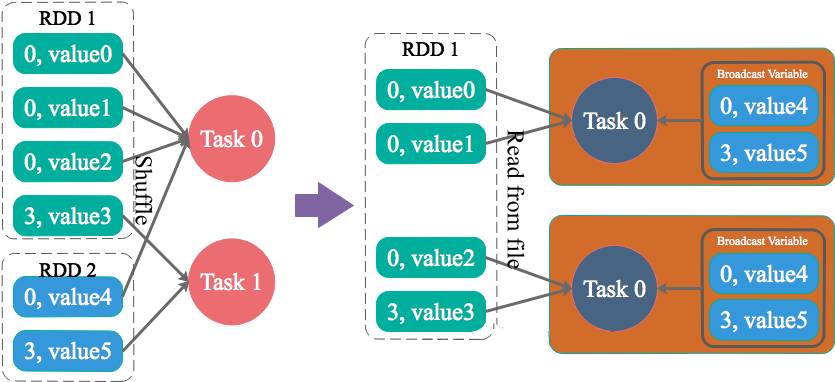
适用场景有限，只能将不同Key分散开，对于同一Key对应数据集非常大的场景不适用。效果与调整并行度类似，只能缓解数据倾斜而不能完全消除数据倾斜。而且需要根据数据特点自定义专用的Partitioner，不够灵活。

#### 将Reduce side Join转变为Map side Join

方案适用场景：在对RDD使用join类操作，或者是在Spark SQL中使用join语句时，而且join操作中的一个RDD或表的数据量比较小（比如几百M或者一两G），比较适用此方案。

方案实现思路：不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作，彻底避免数据倾斜的发生和出现。将较小RDD中的数据直接通过collect算子拉取到Driver端的内存中来，然后对其创建一个Broadcast变量；接着对另外一个RDD执行map类算子，在算子函数内，从Broadcast变量中获取较小RDD的全量数据，与当前RDD的每一条数据按照连接key进行比对，如果连接key相同的话，那么就将两个RDD的数据用你需要的方式连接起来。

方案实现原理：普通的join是会走shuffle过程的，而一旦shuffle，就相当于会将相同key的数据拉取到一个shuffle read task中再进行join，此时就是reduce join。但是如果一个RDD是比较小的，则可以采用广播小RDD全量数据+map算子来实现与join同样的效果，也就是map join，此时就不会发生shuffle操作，也就不会发生数据倾斜。具体原理如下图所示。



方案优点：对join操作导致的数据倾斜，效果非常好，因为根本就不会发生shuffle，也就根本不会发生数据倾斜。

方案缺点：适用场景较少，因为这个方案只适用于一个大表和一个小表的情况。毕竟我们需要将小表进行广播，此时会比较消耗内存资源，driver和每个Executor内存中都会驻留一份小RDD的全量数据。如果我们广播出去的RDD数据比较大，比如10G以上，那么就可能发生内存溢出了。因此并不适合两个都是大表的情况。

通过Spark的Broadcast机制，将Reduce侧Join转化为Map侧Join，避免Shuffle从而完全消除Shuffle带来的数据倾斜。

案例

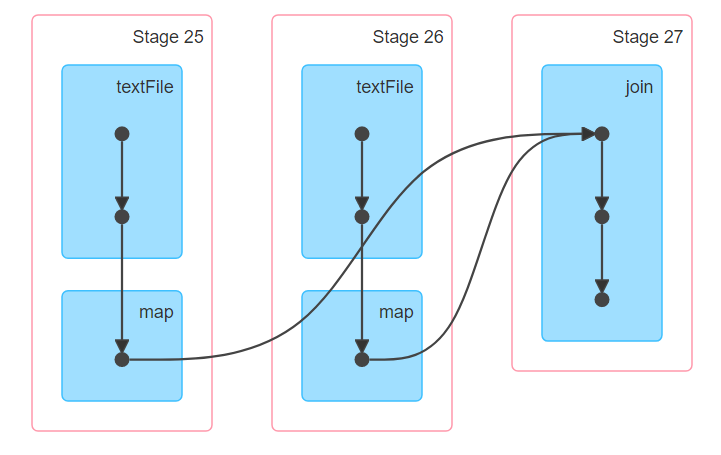
准备数据：

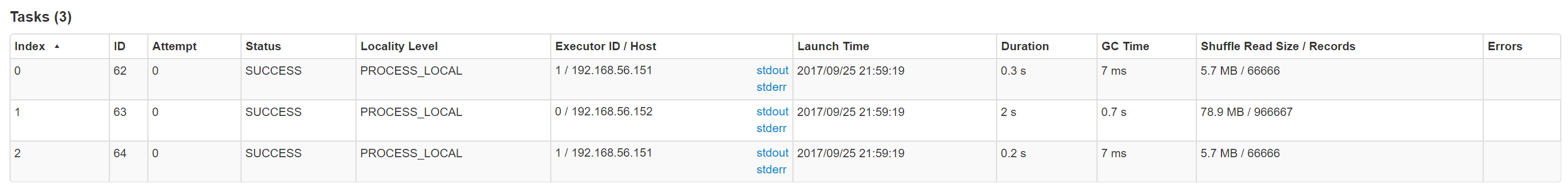
|  |
| --- |
| scala> val sourceRdd = sc.textFile("hdfs://master01:9000/source\_index/p\*")  scala> val kvRdd = sourceRdd.map(x =>{ val parm=x.split("\t");(parm(0).trim().toInt, x) })  scala> val kvRdd2 = kvRdd.map(x=>{if(x.\_1 < 900001) (900001,x.\_2) else x})  scala> kvRdd2.map(x=>x.\_1 +","+x.\_2).saveAsTextFile("hdfs://master01:9000/big\_data/")  scala> val joinRdd2 = kvRdd.filter(\_.\_1 > 900000)  scala> joinRdd2.map(x=>x.\_1 +","+x.\_2).saveAsTextFile("hdfs://master01:9000/small\_data/") |

测试案例：

|  |
| --- |
| scala> val sourceRdd = sc.textFile("hdfs://master01:9000/big\_data/p\*")  scala> val sourceRdd2 = sc.textFile("hdfs://master01:9000/small\_data/p\*")  scala> val joinRdd = sourceRdd.map(x =>{ val parm=x.split(",");(parm(0).trim().toInt, parm(1).trim) })  scala> val joinRdd2 = sourceRdd2.map(x =>{ val parm=x.split(",");(parm(0).trim().toInt, parm(1).trim) })  scala> joinRdd.join(joinRdd2).count |

直接通过将joinRdd（大数据集）和joinRdd2（小数据集）进行join计算，如下：

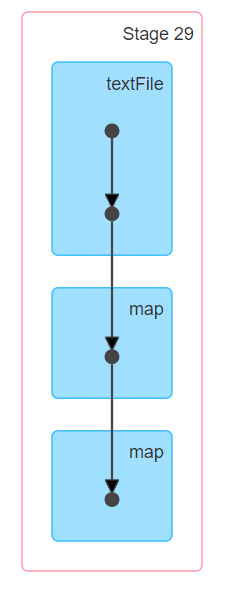




通过广播变量修正后：

|  |
| --- |
| scala> val sourceRdd = sc.textFile("hdfs://master01:9000/big\_data/p\*")  scala> val sourceRdd2 = sc.textFile("hdfs://master01:9000/small\_data/p\*")  scala> val joinRdd = sourceRdd.map(x =>{ val parm=x.split(",");(parm(0).trim().toInt, parm(1).trim) })  scala> val joinRdd2 = sourceRdd2.map(x =>{ val parm=x.split(",");(parm(0).trim().toInt, parm(1).trim) })  scala> val broadcastVar = sc.broadcast(joinRdd2.collectAsMap)  scala> joinRdd.map(x => (x.\_1,(x.\_2,broadcastVar.value.getOrElse(x.\_1,"")))).count |

通过广播变量 + Map完成了相同的工作（没有发生shuffle）：



总结

适用场景

参与Join的一边数据集足够小，可被加载进Driver并通过Broadcast方法广播到各个Executor中。

优势

避免了Shuffle，彻底消除了数据倾斜产生的条件，可极大提升性能。

劣势

要求参与Join的一侧数据集足够小，并且主要适用于Join的场景，不适合聚合的场景，适用条件有限。

#### 两阶段聚合（局部聚合+全局聚合）

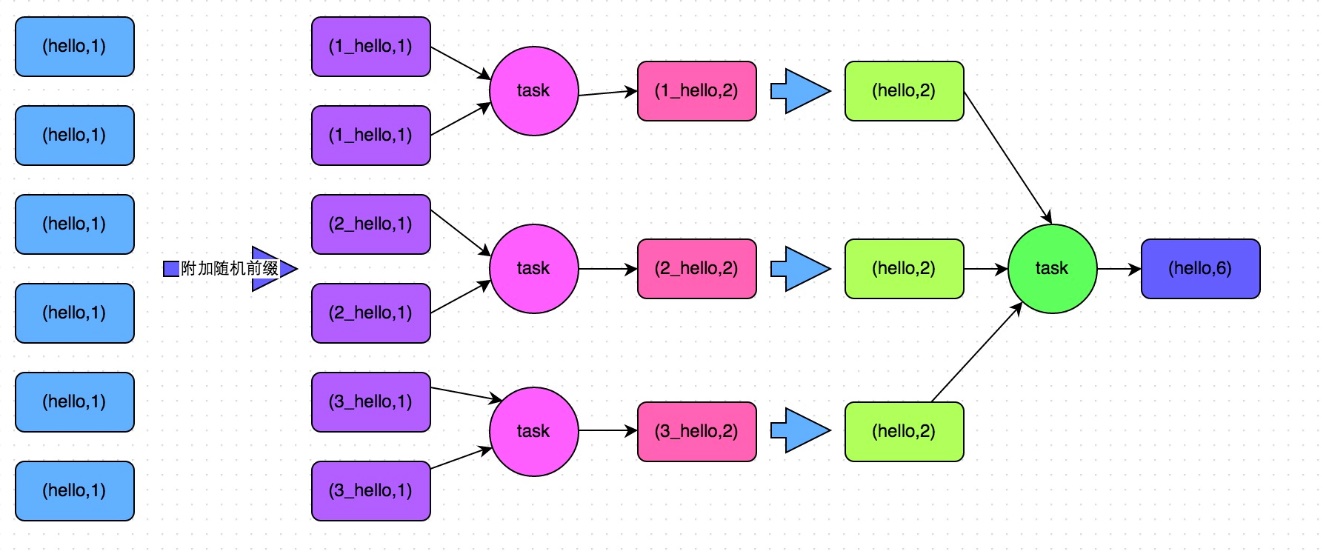
方案适用场景：对RDD执行reduceByKey等聚合类shuffle算子或者在Spark SQL中使用group by语句进行分组聚合时，比较适用这种方案。

方案实现思路：这个方案的核心实现思路就是进行两阶段聚合。第一次是局部聚合，先给每个key都打上一个随机数，比如10以内的随机数，此时原先一样的key就变成不一样的了，比如(hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成(1\_hello, 1) (1\_hello, 1) (2\_hello, 1) (2\_hello, 1)。接着对打上随机数后的数据，执行reduceByKey等聚合操作，进行局部聚合，那么局部聚合结果，就会变成了(1\_hello, 2) (2\_hello, 2)。然后将各个key的前缀给去掉，就会变成(hello,2)(hello,2)，再次进行全局聚合操作，就可以得到最终结果了，比如(hello, 4)。

方案实现原理：将原本相同的key通过附加随机前缀的方式，变成多个不同的key，就可以让原本被一个task处理的数据分散到多个task上去做局部聚合，进而解决单个task处理数据量过多的问题。接着去除掉随机前缀，再次进行全局聚合，就可以得到最终的结果。具体原理见下图。

方案优点：对于聚合类的shuffle操作导致的数据倾斜，效果是非常不错的。通常都可以解决掉数据倾斜，或者至少是大幅度缓解数据倾斜，将Spark作业的性能提升数倍以上。

方案缺点：仅仅适用于聚合类的shuffle操作，适用范围相对较窄。如果是join类的shuffle操作，还得用其他的解决方案。



|  |
| --- |
| *// 第一步，给RDD中的每个key都打上一个随机前缀。* JavaPairRDD<String, Long> randomPrefixRdd = rdd.mapToPair(  **new** PairFunction<Tuple2<Long,Long>, String, Long>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Tuple2<String, Long> call(Tuple2<Long, Long> tuple)  throws Exception {  Random random = **new** Random();  int prefix = random.nextInt(10);  **return new** Tuple2<String, Long>(prefix + **"\_"** + tuple.\_1, tuple.\_2);  }  }); *// 第二步，对打上随机前缀的key进行局部聚合。* JavaPairRDD<String, Long> localAggrRdd = randomPrefixRdd.reduceByKey(  **new** Function2<Long, Long, Long>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Long call(Long v1, Long v2) throws Exception {  **return** v1 + v2;  }  }); *// 第三步，去除RDD中每个key的随机前缀。* JavaPairRDD<Long, Long> removedRandomPrefixRdd = localAggrRdd.mapToPair(  **new** PairFunction<Tuple2<String,Long>, Long, Long>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Tuple2<Long, Long> call(Tuple2<String, Long> tuple)  throws Exception {  long originalKey = Long.valueOf(tuple.\_1.split(**"\_"**)[1]);  **return new** Tuple2<Long, Long>(originalKey, tuple.\_2);  }  }); *// 第四步，对去除了随机前缀的RDD进行全局聚合。* JavaPairRDD<Long, Long> globalAggrRdd = removedRandomPrefixRdd.reduceByKey(  **new** Function2<Long, Long, Long>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Long call(Long v1, Long v2) throws Exception {  **return** v1 + v2;  }  }); |

#### 为数据倾斜的key增加随机前/后缀

原理

为数据量特别大的Key增加随机前/后缀，使得原来Key相同的数据变为Key不相同的数据，从而使倾斜的数据集分散到不同的Task中，彻底解决数据倾斜问题。Join另一侧的数据中，与倾斜Key对应的部分数据，与随机前缀集作笛卡尔乘积，从而保证无论数据倾斜侧倾斜Key如何加前缀，都能与之正常Join。



案例

通过如下SQL，将id为9亿到9.08亿共800万条数据的id转为9500048或者9500096，其它数据的id除以100取整。从而该数据集中，id为9500048和9500096的数据各400万，其它id对应的数据记录数均为100条。这些数据存于名为test的表中。

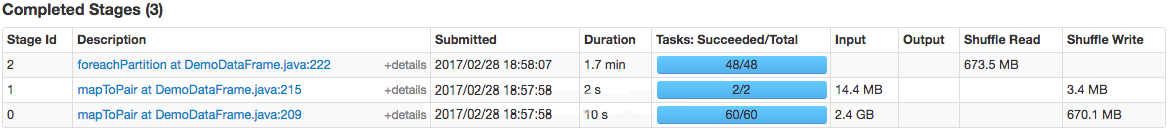
对于另外一张小表test\_new，取出50万条数据，并将id（递增且唯一）除以100取整，使得所有id都对应100条数据。

|  |
| --- |
| NSERT OVERWRITE TABLE test  SELECT CAST(CASE WHEN id < 908000000 THEN (9500000 + (CAST (RAND() \* 2 AS INT) + 1) \* 48 )  ELSE CAST(id/100 AS INT) END AS STRING), name FROM student\_external  WHERE id BETWEEN 900000000 AND 1050000000; INSERT OVERWRITE TABLE test\_new  SELECT CAST(CAST(id/100 AS INT) AS STRING), name FROM student\_delta\_external  WHERE id BETWEEN 950000000 AND 950500000; |

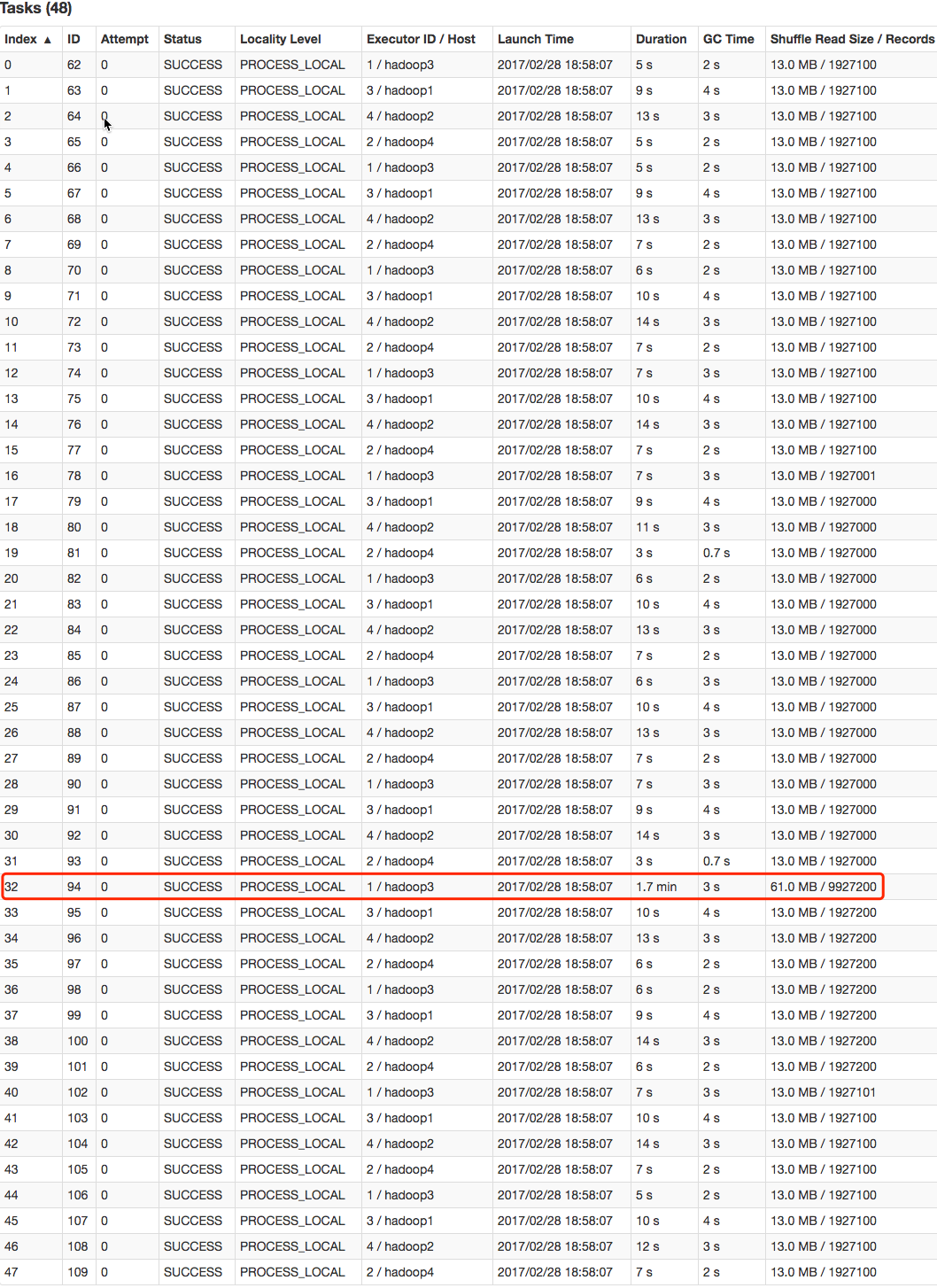
通过如下代码，读取test表对应的文件夹内的数据并转换为JavaPairRDD存于leftRDD中，同样读取test表对应的数据存于rightRDD中。通过RDD的join算子对leftRDD与rightRDD进行Join，并指定并行度为48。

|  |
| --- |
| public **class** SparkDataSkew{  public static void main(String[] args) {  SparkConf sparkConf = **new** SparkConf();  sparkConf.setAppName(**"DemoSparkDataFrameWithSkewedBigTableDirect"**);  sparkConf.set(**"spark.default.parallelism"**, parallelism + **""**);  JavaSparkContext javaSparkContext = **new** JavaSparkContext(sparkConf);  JavaPairRDD<String, String> leftRDD = javaSparkContext.textFile(**"hdfs://hadoop1:8020/apps/hive/warehouse/default/test/"**)  .mapToPair((String row) -> {  String[] str = row.split(**","**);  **return new** Tuple2<String, String>(str[0], str[1]);  });  JavaPairRDD<String, String> rightRDD = javaSparkContext.textFile(**"hdfs://hadoop1:8020/apps/hive/warehouse/default/test\_new/"**)  .mapToPair((String row) -> {  String[] str = row.split(**","**);  **return new** Tuple2<String, String>(str[0], str[1]);  });  leftRDD.join(rightRDD, parallelism)  .mapToPair((Tuple2<String, Tuple2<String, String>> tuple) -> **new** Tuple2<String, String>(tuple.\_1(), tuple.\_2().\_2()))  .foreachPartition((*Iterator*<Tuple2<String, String>> iterator) -> {  AtomicInteger atomicInteger = **new** AtomicInteger();  iterator.forEachRemaining((Tuple2<String, String> tuple) -> atomicInteger.incrementAndGet());  });  javaSparkContext.stop();  javaSparkContext.close();  } } |

从下图可看出，整个Join耗时1分54秒，其中Join Stage耗时1.7分钟。



通过分析Join Stage的所有Task可知，在其它Task所处理记录数为192.71万的同时Task 32的处理的记录数为992.72万，故它耗时为1.7分钟，远高于其它Task的约10秒。这与上文准备数据集时，将id为9500048为9500096对应的数据量设置非常大，其它id对应的数据集非常均匀相符合。



现通过如下操作，实现倾斜Key的分散处理

将leftRDD中倾斜的key（即9500048与9500096）对应的数据单独过滤出来，且加上1到24的随机前缀，并将前缀与原数据用逗号分隔（以方便之后去掉前缀）形成单独的leftSkewRDD

将rightRDD中倾斜key对应的数据抽取出来，并通过flatMap操作将该数据集中每条数据均转换为24条数据（每条分别加上1到24的随机前缀），形成单独的rightSkewRDD

将leftSkewRDD与rightSkewRDD进行Join，并将并行度设置为48，且在Join过程中将随机前缀去掉，得到倾斜数据集的Join结果skewedJoinRDD

将leftRDD中不包含倾斜Key的数据抽取出来作为单独的leftUnSkewRDD

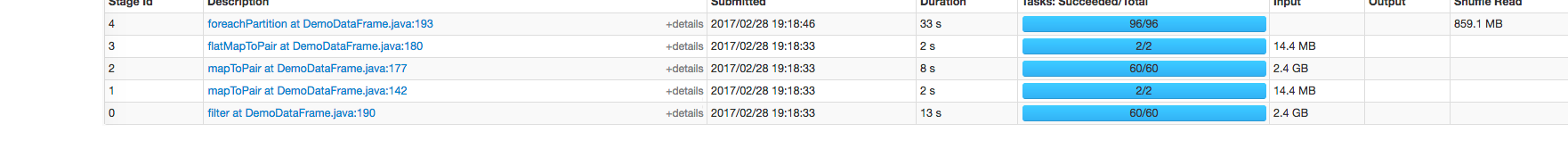
对leftUnSkewRDD与原始的rightRDD进行Join，并行度也设置为48，得到Join结果unskewedJoinRDD

通过union算子将skewedJoinRDD与unskewedJoinRDD进行合并，从而得到完整的Join结果集

具体实现代码如下

|  |
| --- |
| public **class** SparkDataSkew{  public static void main(String[] args) {  int parallelism = 48;  SparkConf sparkConf = **new** SparkConf();  sparkConf.setAppName(**"SolveDataSkewWithRandomPrefix"**);  sparkConf.set(**"spark.default.parallelism"**, parallelism + **""**);  JavaSparkContext javaSparkContext = **new** JavaSparkContext(sparkConf);  JavaPairRDD<String, String> leftRDD = javaSparkContext.textFile(**"hdfs://hadoop1:8020/apps/hive/warehouse/default/test/"**)  .mapToPair((String row) -> {  String[] str = row.split(**","**);  **return new** Tuple2<String, String>(str[0], str[1]);  });  JavaPairRDD<String, String> rightRDD = javaSparkContext.textFile(**"hdfs://hadoop1:8020/apps/hive/warehouse/default/test\_new/"**)  .mapToPair((String row) -> {  String[] str = row.split(**","**);  **return new** Tuple2<String, String>(str[0], str[1]);  });  String[] skewedKeyArray = **new** String[]{**"9500048"**, **"9500096"**};  *Set*<String> skewedKeySet = **new** HashSet<String>();  *List*<String> addList = **new** ArrayList<String>();  **for**(int i = 1; i <=24; i++) {  addList.add(i + **""**);  }  **for**(String key : skewedKeyArray) {  skewedKeySet.add(key);  }  Broadcast<Set<String>> skewedKeys = javaSparkContext.broadcast(skewedKeySet);  Broadcast<List<String>> addListKeys = javaSparkContext.broadcast(addList);  JavaPairRDD<String, String> leftSkewRDD = leftRDD  .filter((Tuple2<String, String> tuple) -> skewedKeys.value().contains(tuple.\_1()))  .mapToPair((Tuple2<String, String> tuple) -> **new** Tuple2<String, String>((**new** Random().nextInt(24) + 1) + **","** + tuple.\_1(), tuple.\_2()));  JavaPairRDD<String, String> rightSkewRDD = rightRDD.filter((Tuple2<String, String> tuple) -> skewedKeys.value().contains(tuple.\_1()))  .flatMapToPair((Tuple2<String, String> tuple) -> addListKeys.value().stream()  .map((String i) -> **new** Tuple2<String, String>( i + **","** + tuple.\_1(), tuple.\_2()))  .collect(Collectors.toList())  .iterator()  );  JavaPairRDD<String, String> skewedJoinRDD = leftSkewRDD  .join(rightSkewRDD, parallelism)  .mapToPair((Tuple2<String, Tuple2<String, String>> tuple) -> **new** Tuple2<String, String>(tuple.\_1().split(**","**)[1], tuple.\_2().\_2()));  JavaPairRDD<String, String> leftUnSkewRDD = leftRDD.filter((Tuple2<String, String> tuple) -> !skewedKeys.value().contains(tuple.\_1()));  JavaPairRDD<String, String> unskewedJoinRDD = leftUnSkewRDD.join(rightRDD, parallelism).mapToPair((Tuple2<String, Tuple2<String, String>> tuple) -> **new** Tuple2<String, String>(tuple.\_1(), tuple.\_2().\_2()));  skewedJoinRDD.union(unskewedJoinRDD).foreachPartition((Iterator<Tuple2<String, String>> iterator) -> {  AtomicInteger atomicInteger = **new** AtomicInteger();  iterator.forEachRemaining((Tuple2<String, String> tuple) -> atomicInteger.incrementAndGet());  });  javaSparkContext.stop();  javaSparkContext.close();  } } |

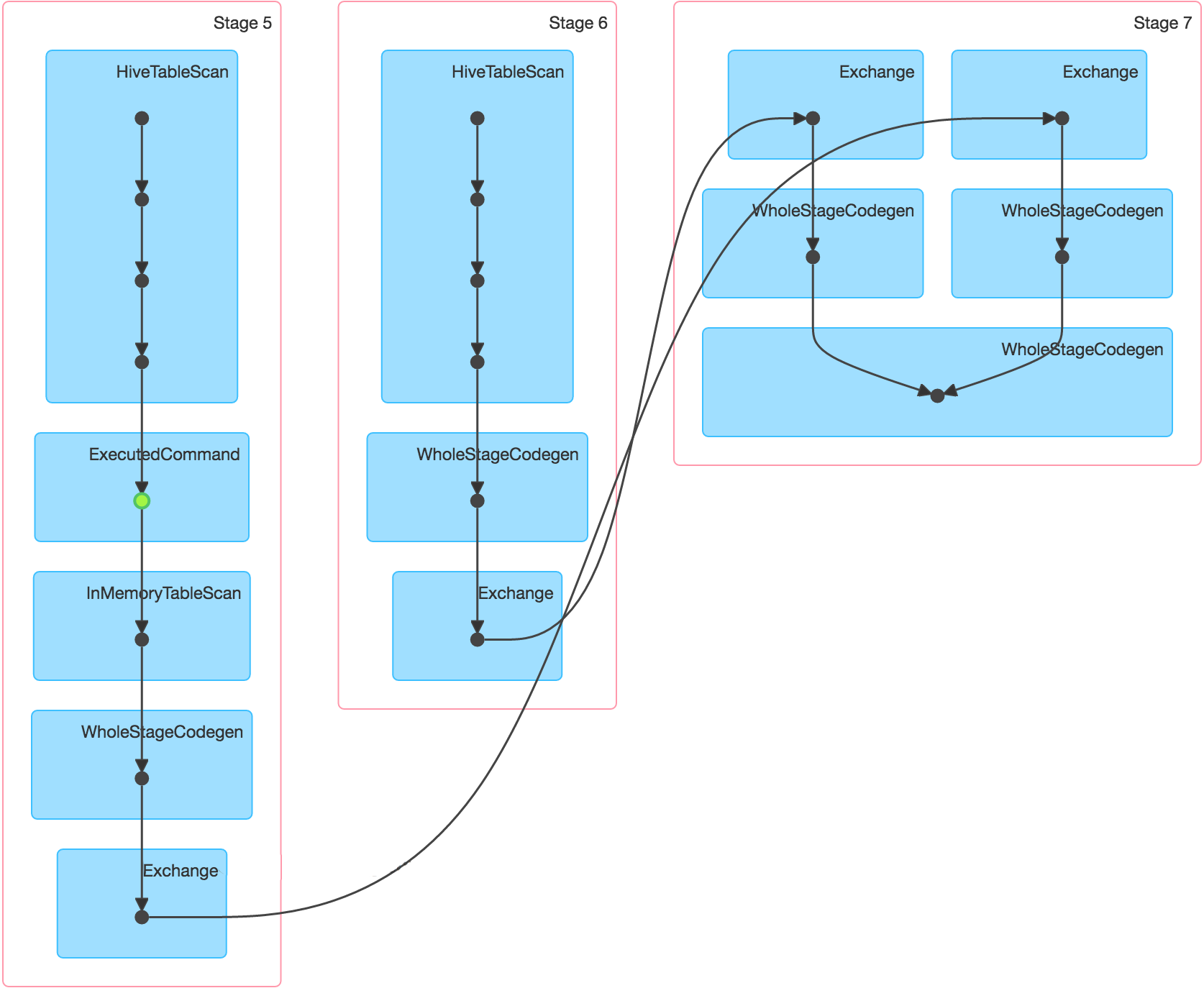
从下图可看出，整个Join耗时58秒，其中Join Stage耗时33秒。



通过分析Join Stage的所有Task可知

由于Join分倾斜数据集Join和非倾斜数据集Join，而各Join的并行度均为48，故总的并行度为96

由于提交任务时，设置的Executor个数为4，每个Executor的core数为12，故可用Core数为48，所以前48个Task同时启动（其Launch时间相同），后48个Task的启动时间各不相同（等待前面的Task结束才开始）  
由于倾斜Key被加上随机前缀，原本相同的Key变为不同的Key，被分散到不同的Task处理，故在所有Task中，未发现所处理数据集明显高于其它Task的情况



实际上，由于倾斜Key与非倾斜Key的操作完全独立，可并行进行。而本实验受限于可用总核数为48，可同时运行的总Task数为48，故而该方案只是将总耗时减少一半（效率提升一倍）。如果资源充足，可并发执行Task数增多，该方案的优势将更为明显。在实际项目中，该方案往往可提升数倍至10倍的效率。

总结

适用场景

两张表都比较大，无法使用Map则Join。其中一个RDD有少数几个Key的数据量过大，另外一个RDD的Key分布较为均匀。

解决方案

将有数据倾斜的RDD中倾斜Key对应的数据集单独抽取出来加上随机前缀，另外一个RDD每条数据分别与随机前缀结合形成新的RDD（相当于将其数据增到到原来的N倍，N即为随机前缀的总个数），然后将二者Join并去掉前缀。然后将不包含倾斜Key的剩余数据进行Join。最后将两次Join的结果集通过union合并，即可得到全部Join结果。

优势

相对于Map则Join，更能适应大数据集的Join。如果资源充足，倾斜部分数据集与非倾斜部分数据集可并行进行，效率提升明显。且只针对倾斜部分的数据做数据扩展，增加的资源消耗有限。

劣势

如果倾斜Key非常多，则另一侧数据膨胀非常大，此方案不适用。而且此时对倾斜Key与非倾斜Key分开处理，需要扫描数据集两遍，增加了开销。

#### 使用随机前缀和扩容RDD进行join

方案适用场景：如果在进行join操作时，RDD中有大量的key导致数据倾斜，那么进行分拆key也没什么意义，此时就只能使用最后一种方案来解决问题了。

方案实现思路：

该方案的实现思路基本和“解决方案六”类似，首先查看RDD/Hive表中的数据分布情况，找到那个造成数据倾斜的RDD/Hive表，比如有多个key都对应了超过1万条数据。

然后将该RDD的每条数据都打上一个n以内的随机前缀。

同时对另外一个正常的RDD进行扩容，将每条数据都扩容成n条数据，扩容出来的每条数据都依次打上一个0~n的前缀。

最后将两个处理后的RDD进行join即可。

方案实现原理：将原先一样的key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同key”分散到多个task中去处理，而不是让一个task处理大量的相同key。该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；而这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。

方案优点：对join类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错。

方案缺点：该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个RDD进行扩容，对内存资源要求很高。

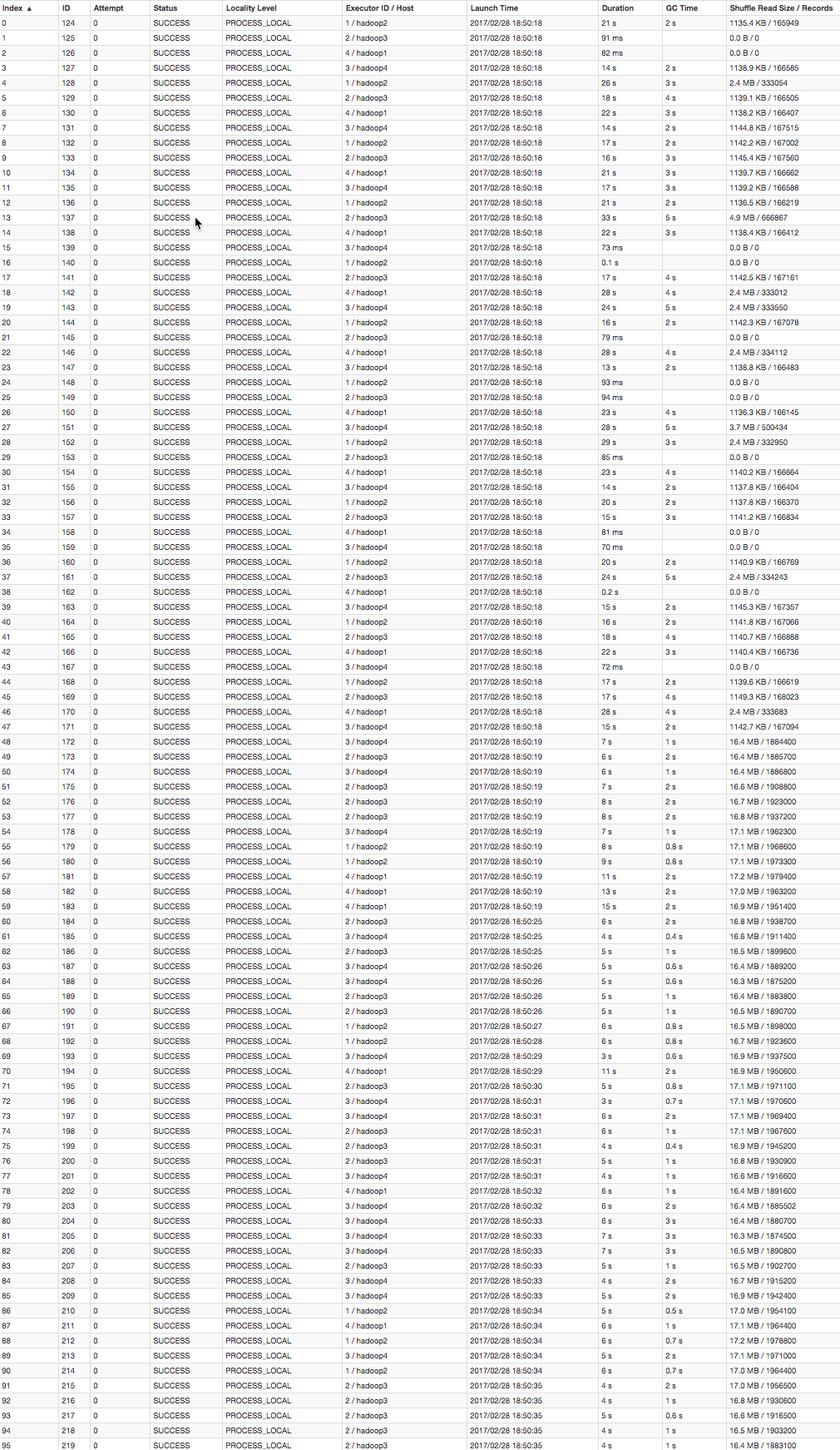
方案实践经验：曾经开发一个数据需求的时候，发现一个join导致了数据倾斜。优化之前，作业的执行时间大约是60分钟左右；使用该方案优化之后，执行时间缩短到10分钟左右，性能提升了6倍。

|  |
| --- |
| *// 首先将其中一个key分布相对较为均匀的RDD膨胀100倍。* JavaPairRDD<String, Row> expandedRDD = rdd1.flatMapToPair(  **new** PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Iterable<Tuple2<String, Row>> call(Tuple2<Long, Row> tuple)  throws Exception {  *List*<Tuple2<String, Row>> list = **new** ArrayList<Tuple2<String, Row>>();  **for**(int i = 0; i < 100; i++) {  list.add(**new** Tuple2<String, Row>(0 + **"\_"** + tuple.\_1, tuple.\_2));  }  **return** list;  }  }); *// 其次，将另一个有数据倾斜key的RDD，每条数据都打上100以内的随机前缀。* JavaPairRDD<String, String> mappedRDD = rdd2.mapToPair(  **new** PairFunction<Tuple2<Long,String>, String, String>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Tuple2<String, String> call(Tuple2<Long, String> tuple)  throws Exception {  Random random = **new** Random();  int prefix = random.nextInt(100);  **return new** Tuple2<String, String>(prefix + **"\_"** + tuple.\_1, tuple.\_2);  }  }); *// 将两个处理后的RDD进行join即可。* JavaPairRDD<String, Tuple2<String, Row>> joinedRDD = mappedRDD.join(expandedRDD); |

#### 大表随机添加N种随机前缀，小表扩大N倍

原理

如果出现数据倾斜的Key比较多，上一种方法将这些大量的倾斜Key分拆出来，意义不大。此时更适合直接对存在数据倾斜的数据集全部加上随机前缀，然后对另外一个不存在严重数据倾斜的数据集整体与随机前缀集作笛卡尔乘积（即将数据量扩大N倍）。



案例

这里给出示例代码，读者可参考上文中分拆出少数倾斜Key添加随机前缀的方法，自行测试。

|  |
| --- |
| public **class** SparkDataSkew {  public static void main(String[] args) {  SparkConf sparkConf = **new** SparkConf();  sparkConf.setAppName(**"ResolveDataSkewWithNAndRandom"**);  sparkConf.set(**"spark.default.parallelism"**, parallelism + **""**);  JavaSparkContext javaSparkContext = **new** JavaSparkContext(sparkConf);  JavaPairRDD<String, String> leftRDD = javaSparkContext.textFile(**"hdfs://hadoop1:8020/apps/hive/warehouse/default/test/"**)  .mapToPair((String row) -> {  String[] str = row.split(**","**);  **return new** Tuple2<String, String>(str[0], str[1]);  });  JavaPairRDD<String, String> rightRDD = javaSparkContext.textFile(**"hdfs://hadoop1:8020/apps/hive/warehouse/default/test\_new/"**)  .mapToPair((String row) -> {  String[] str = row.split(**","**);  **return new** Tuple2<String, String>(str[0], str[1]);  });  *List*<String> addList = **new** ArrayList<String>();  **for**(int i = 1; i <=48; i++) {  addList.add(i + **""**);  }  Broadcast<List<String>> addListKeys = javaSparkContext.broadcast(addList);  JavaPairRDD<String, String> leftRandomRDD = leftRDD.mapToPair((Tuple2<String, String> tuple) -> **new** Tuple2<String, String>(**new** Random().nextInt(48) + **","** + tuple.\_1(), tuple.\_2()));  JavaPairRDD<String, String> rightNewRDD = rightRDD  .flatMapToPair((Tuple2<String, String> tuple) -> addListKeys.value().stream()  .map((String i) -> **new** Tuple2<String, String>( i + **","** + tuple.\_1(), tuple.\_2()))  .collect(Collectors.toList())  .iterator()  );  JavaPairRDD<String, String> joinRDD = leftRandomRDD  .join(rightNewRDD, parallelism)  .mapToPair((Tuple2<String, Tuple2<String, String>> tuple) -> **new** Tuple2<String, String>(tuple.\_1().split(**","**)[1], tuple.\_2().\_2()));  joinRDD.foreachPartition((Iterator<Tuple2<String, String>> iterator) -> {  AtomicInteger atomicInteger = **new** AtomicInteger();  iterator.forEachRemaining((Tuple2<String, String> tuple) -> atomicInteger.incrementAndGet());  });  javaSparkContext.stop();  javaSparkContext.close();  } } |

总结

适用场景

一个数据集存在的倾斜Key比较多，另外一个数据集数据分布比较均匀。

优势

对大部分场景都适用，效果不错。

劣势

需要将一个数据集整体扩大N倍，会增加资源消耗。

总结

对于数据倾斜，并无一个统一的一劳永逸的方法。更多的时候，是结合数据特点（数据集大小，倾斜Key的多少等）综合使用上文所述的多种方法。

#### 采样倾斜key并分拆join操作

方案适用场景：两个RDD/Hive表进行join的时候，如果数据量都比较大，无法采用“解决方案五”，那么此时可以看一下两个RDD/Hive表中的key分布情况。如果出现数据倾斜，是因为其中某一个RDD/Hive表中的少数几个key的数据量过大，而另一个RDD/Hive表中的所有key都分布比较均匀，那么采用这个解决方案是比较合适的。

方案实现思路：

对包含少数几个数据量过大的key的那个RDD，通过sample算子采样出一份样本来，然后统计一下每个key的数量，计算出来数据量最大的是哪几个key。

然后将这几个key对应的数据从原来的RDD中拆分出来，形成一个单独的RDD，并给每个key都打上n以内的随机数作为前缀，而不会导致倾斜的大部分key形成另外一个RDD。

接着将需要join的另一个RDD，也过滤出来那几个倾斜key对应的数据并形成一个单独的RDD，将每条数据膨胀成n条数据，这n条数据都按顺序附加一个0~n的前缀，不会导致倾斜的大部分key也形成另外一个RDD。

再将附加了随机前缀的独立RDD与另一个膨胀n倍的独立RDD进行join，此时就可以将原先相同的key打散成n份，分散到多个task中去进行join了。

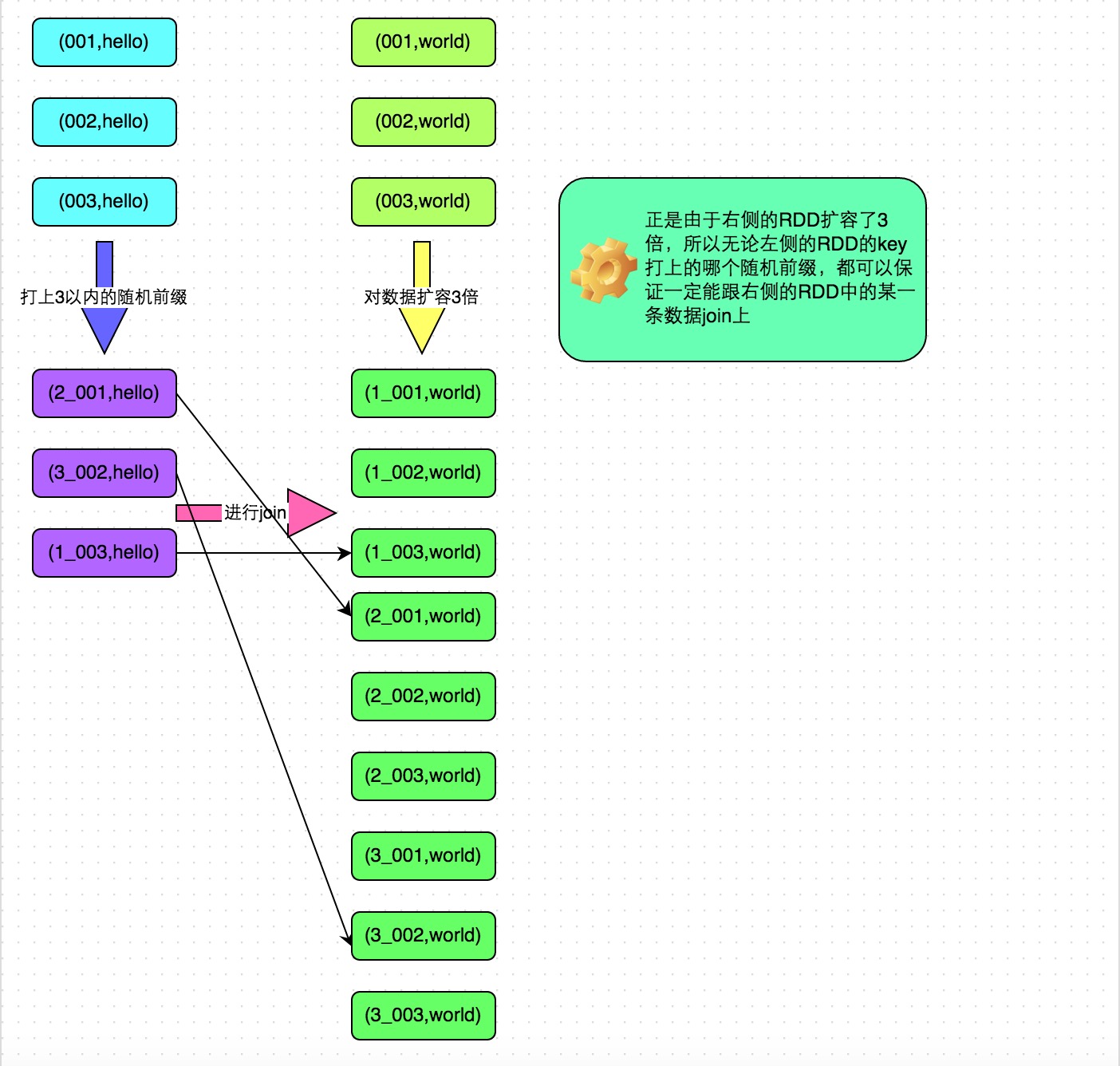
而另外两个普通的RDD就照常join即可。

最后将两次join的结果使用union算子合并起来即可，就是最终的join结果。

方案实现原理：对于join导致的数据倾斜，如果只是某几个key导致了倾斜，可以将少数几个key分拆成独立RDD，并附加随机前缀打散成n份去进行join，此时这几个key对应的数据就不会集中在少数几个task上，而是分散到多个task进行join了。具体原理见下图。

方案优点：对于join导致的数据倾斜，如果只是某几个key导致了倾斜，采用该方式可以用最有效的方式打散key进行join。而且只需要针对少数倾斜key对应的数据进行扩容n倍，不需要对全量数据进行扩容。避免了占用过多内存。

方案缺点：如果导致倾斜的key特别多的话，比如成千上万个key都导致数据倾斜，那么这种方式也不适合。



|  |
| --- |
| *// 首先从包含了少数几个导致数据倾斜key的rdd1中，采样10%的样本数据。* JavaPairRDD<Long, String> sampledRDD = rdd1.sample(**false**, 0.1); *// 对样本数据RDD统计出每个key的出现次数，并按出现次数降序排序。 // 对降序排序后的数据，取出top 1或者top 100的数据，也就是key最多的前n个数据。 // 具体取出多少个数据量最多的key，由大家自己决定，我们这里就取1个作为示范。* JavaPairRDD<Long, Long> mappedSampledRDD = sampledRDD.mapToPair(  **new** PairFunction<Tuple2<Long,String>, Long, Long>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Tuple2<Long, Long> call(Tuple2<Long, String> tuple)  throws Exception {  **return new** Tuple2<Long, Long>(tuple.\_1, 1L);  }  }); JavaPairRDD<Long, Long> countedSampledRDD = mappedSampledRDD.reduceByKey(  **new** Function2<Long, Long, Long>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Long call(Long v1, Long v2) throws Exception {  **return** v1 + v2;  }  }); JavaPairRDD<Long, Long> reversedSampledRDD = countedSampledRDD.mapToPair(  **new** PairFunction<Tuple2<Long,Long>, Long, Long>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Tuple2<Long, Long> call(Tuple2<Long, Long> tuple)  throws Exception {  **return new** Tuple2<Long, Long>(tuple.\_2, tuple.\_1);  }  }); **final** Long skewedUserid = reversedSampledRDD.sortByKey(**false**).take(1).get(0).\_2; *// 从rdd1中分拆出导致数据倾斜的key，形成独立的RDD。* JavaPairRDD<Long, String> skewedRDD = rdd1.filter(  **new** Function<Tuple2<Long,String>, Boolean>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Boolean call(Tuple2<Long, String> tuple) throws Exception {  **return** tuple.\_1.equals(skewedUserid);  }  }); *// 从rdd1中分拆出不导致数据倾斜的普通key，形成独立的RDD。* JavaPairRDD<Long, String> commonRDD = rdd1.filter(  **new** Function<Tuple2<Long,String>, Boolean>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Boolean call(Tuple2<Long, String> tuple) throws Exception {  **return** !tuple.\_1.equals(skewedUserid);  }  }); *// rdd2，就是那个所有key的分布相对较为均匀的rdd。 // 这里将rdd2中，前面获取到的key对应的数据，过滤出来，分拆成单独的rdd，并对rdd中的数据使用flatMap算子都扩容100倍。 // 对扩容的每条数据，都打上0～100的前缀。* JavaPairRDD<String, Row> skewedRdd2 = rdd2.filter(  **new** Function<Tuple2<Long,Row>, Boolean>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Boolean call(Tuple2<Long, Row> tuple) throws Exception {  **return** tuple.\_1.equals(skewedUserid);  }  }).flatMapToPair(**new** PairFlatMapFunction<Tuple2<Long,Row>, String, Row>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Iterable<Tuple2<String, Row>> call(  Tuple2<Long, Row> tuple) throws Exception {  Random random = **new** Random();  *List*<Tuple2<String, Row>> list = **new** ArrayList<Tuple2<String, Row>>();  **for**(int i = 0; i < 100; i++) {  list.add(**new** Tuple2<String, Row>(i + **"\_"** + tuple.\_1, tuple.\_2));  }  **return** list;  } }); *// 将rdd1中分拆出来的导致倾斜的key的独立rdd，每条数据都打上100以内的随机前缀。 // 然后将这个rdd1中分拆出来的独立rdd，与上面rdd2中分拆出来的独立rdd，进行join。* JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD1 = skewedRDD.mapToPair(  **new** PairFunction<Tuple2<Long,String>, String, String>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Tuple2<String, String> call(Tuple2<Long, String> tuple)  throws Exception {  Random random = **new** Random();  int prefix = random.nextInt(100);  **return new** Tuple2<String, String>(prefix + **"\_"** + tuple.\_1, tuple.\_2);  }  })  .join(skewedUserid2infoRDD)  .mapToPair(**new** PairFunction<Tuple2<String,Tuple2<String,Row>>, Long, Tuple2<String, Row>>() {  **private** static **final** long serialVersionUID = 1L;  @Override  public Tuple2<Long, Tuple2<String, Row>> call(  Tuple2<String, Tuple2<String, Row>> tuple)  throws Exception {  long key = Long.valueOf(tuple.\_1.split(**"\_"**)[1]);  **return new** Tuple2<Long, Tuple2<String, Row>>(key, tuple.\_2);  }  }); *// 将rdd1中分拆出来的包含普通key的独立rdd，直接与rdd2进行join。* JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD2 = commonRDD.join(rdd2); *// 将倾斜key join后的结果与普通key join后的结果，uinon起来。 // 就是最终的join结果。* JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD = joinedRDD1.union(joinedRDD2); |

#### 过滤少数导致倾斜的key

方案适用场景：如果发现导致倾斜的key就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。比如99%的key就对应10条数据，但是只有一个key对应了100万数据，从而导致了数据倾斜。

方案实现思路：如果我们判断那少数几个数据量特别多的key，对作业的执行和计算结果不是特别重要的话，那么干脆就直接过滤掉那少数几个key。比如，在Spark SQL中可以使用where子句过滤掉这些key或者在Spark Core中对RDD执行filter算子过滤掉这些key。如果需要每次作业执行时，动态判定哪些key的数据量最多然后再进行过滤，那么可以使用sample算子对RDD进行采样，然后计算出每个key的数量，取数据量最多的key过滤掉即可。

方案实现原理：将导致数据倾斜的key给过滤掉之后，这些key就不会参与计算了，自然不可能产生数据倾斜。

方案优点：实现简单，而且效果也很好，可以完全规避掉数据倾斜。

方案缺点：适用场景不多，大多数情况下，导致倾斜的key还是很多的，并不是只有少数几个。

方案实践经验：在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天Spark作业在运行的时候突然OOM了，追查之后发现，是Hive表中的某一个key在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个key之后，直接在程序中将那些key给过滤掉。

## Shuffle调优

### 调优概述

大多数Spark作业的性能主要就是消耗在了shuffle环节，因为该环节包含了大量的磁盘IO、序列化、网络数据传输等操作。因此，如果要让作业的性能更上一层楼，就有必要对shuffle过程进行调优。但是也必须提醒大家的是，影响一个Spark作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle调优只能在整个Spark的性能调优中占到一小部分而已。因此大家务必把握住调优的基本原则，千万不要舍本逐末。下面我们就给大家详细讲解shuffle的原理，以及相关参数的说明，同时给出各个参数的调优建议。

### ShuffleManager发展概述

在Spark的源码中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。而随着Spark的版本的发展，ShuffleManager也在不断迭代，变得越来越先进。

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。该ShuffleManager有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘IO操作影响了性能。

因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（merge）成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

下面我们详细分析一下HashShuffleManager和SortShuffleManager的原理。

### HashShuffleManager运行原理

未经优化的HashShuffleManager

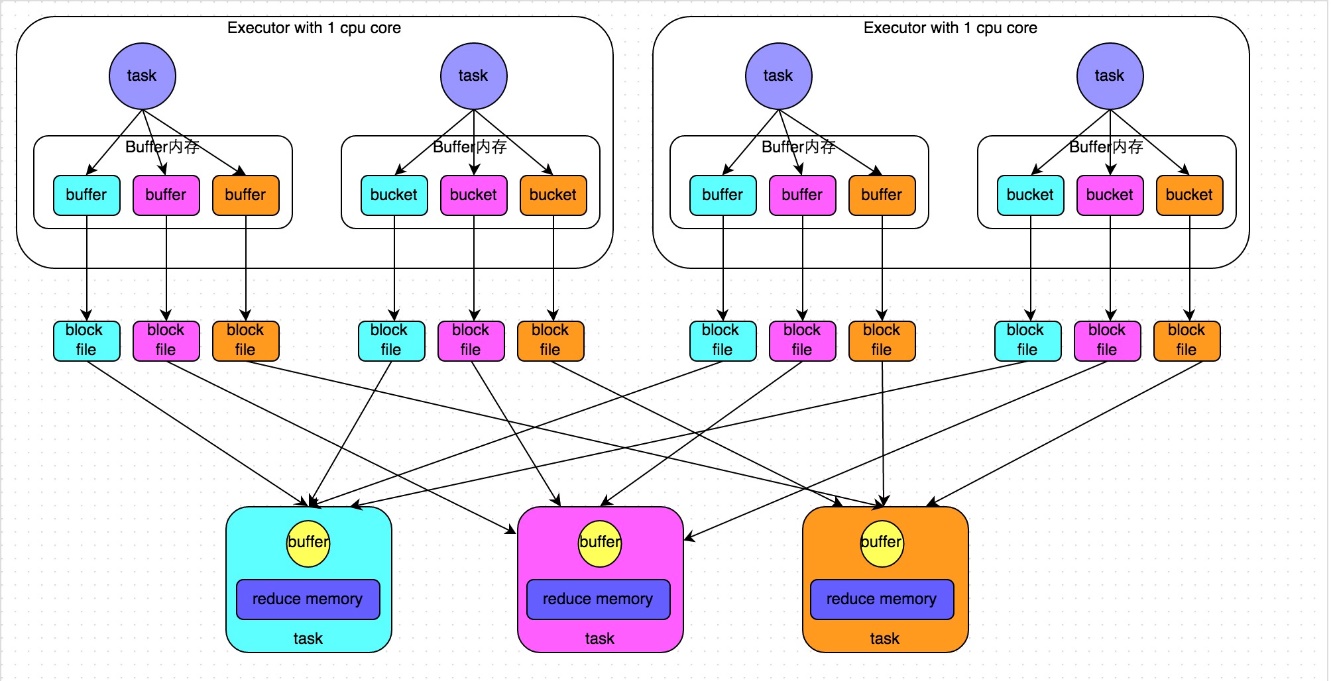
下图说明了未经优化的HashShuffleManager的原理。这里我们先明确一个假设前提：每个Executor只有1个CPU core，也就是说，无论这个Executor上分配多少个task线程，同一时间都只能执行一个task线程。

我们先从shuffle write开始说起。shuffle write阶段，主要就是在一个stage结束计算之后，为了下一个stage可以执行shuffle类的算子（比如reduceByKey），而将每个task处理的数据按key进行“分类”。所谓“分类”，就是对相同的key执行hash算法，从而将相同key都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游stage的一个task。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

那么每个执行shuffle write的task，要为下一个stage创建多少个磁盘文件呢？很简单，下一个stage的task有多少个，当前stage的每个task就要创建多少份磁盘文件。比如下一个stage总共有100个task，那么当前stage的每个task都要创建100份磁盘文件。如果当前stage有50个task，总共有10个Executor，每个Executor执行5个Task，那么每个Executor上总共就要创建500个磁盘文件，所有Executor上会创建5000个磁盘文件。由此可见，未经优化的shuffle write操作所产生的磁盘文件的数量是极其惊人的。

接着我们来说说shuffle read。shuffle read，通常就是一个stage刚开始时要做的事情。此时该stage的每一个task就需要将上一个stage的计算结果中的所有相同key，从各个节点上通过网络都拉取到自己所在的节点上，然后进行key的聚合或连接等操作。由于shuffle write的过程中，task给下游stage的每个task都创建了一个磁盘文件，因此shuffle read的过程中，每个task只要从上游stage的所有task所在节点上，拉取属于自己的那一个磁盘文件即可。

shuffle read的拉取过程是一边拉取一边进行聚合的。每个shuffle read task都会有一个自己的buffer缓冲，每次都只能拉取与buffer缓冲相同大小的数据，然后通过内存中的一个Map进行聚合等操作。聚合完一批数据后，再拉取下一批数据，并放到buffer缓冲中进行聚合操作。以此类推，直到最后将所有数据到拉取完，并得到最终的结果。



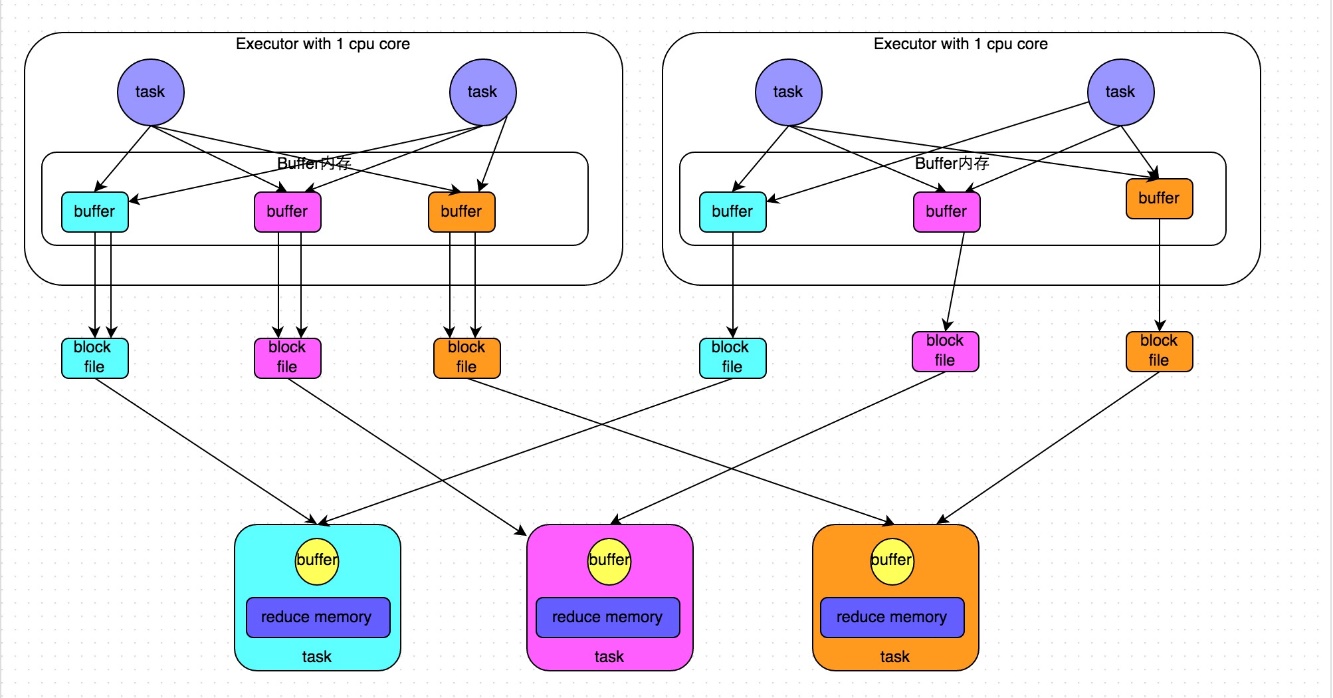
优化后的HashShuffleManager

下图说明了优化后的HashShuffleManager的原理。这里说的优化，是指我们可以设置一个参数，spark.shuffle.consolidateFiles。该参数默认值为false，将其设置为true即可开启优化机制。通常来说，如果我们使用HashShuffleManager，那么都建议开启这个选项。

开启consolidate机制之后，在shuffle write过程中，task就不是为下游stage的每个task创建一个磁盘文件了。此时会出现shuffleFileGroup的概念，每个shuffleFileGroup会对应一批磁盘文件，磁盘文件的数量与下游stage的task数量是相同的。一个Executor上有多少个CPU core，就可以并行执行多少个task。而第一批并行执行的每个task都会创建一个shuffleFileGroup，并将数据写入对应的磁盘文件内。

当Executor的CPU core执行完一批task，接着执行下一批task时，下一批task就会复用之前已有的shuffleFileGroup，包括其中的磁盘文件。也就是说，此时task会将数据写入已有的磁盘文件中，而不会写入新的磁盘文件中。因此，consolidate机制允许不同的task复用同一批磁盘文件，这样就可以有效将多个task的磁盘文件进行一定程度上的合并，从而大幅度减少磁盘文件的数量，进而提升shuffle write的性能。

假设第二个stage有100个task，第一个stage有50个task，总共还是有10个Executor，每个Executor执行5个task。那么原本使用未经优化的HashShuffleManager时，每个Executor会产生500个磁盘文件，所有Executor会产生5000个磁盘文件的。但是此时经过优化之后，每个Executor创建的磁盘文件的数量的计算公式为：CPU core的数量 \* 下一个stage的task数量。也就是说，每个Executor此时只会创建100个磁盘文件，所有Executor只会创建1000个磁盘文件。



### SortShuffleManager运行原理

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是bypass运行机制。当shuffle read task的数量小于等于spark.shuffle.sort.bypassMergeThreshold参数的值时（默认为200），就会启用bypass机制。

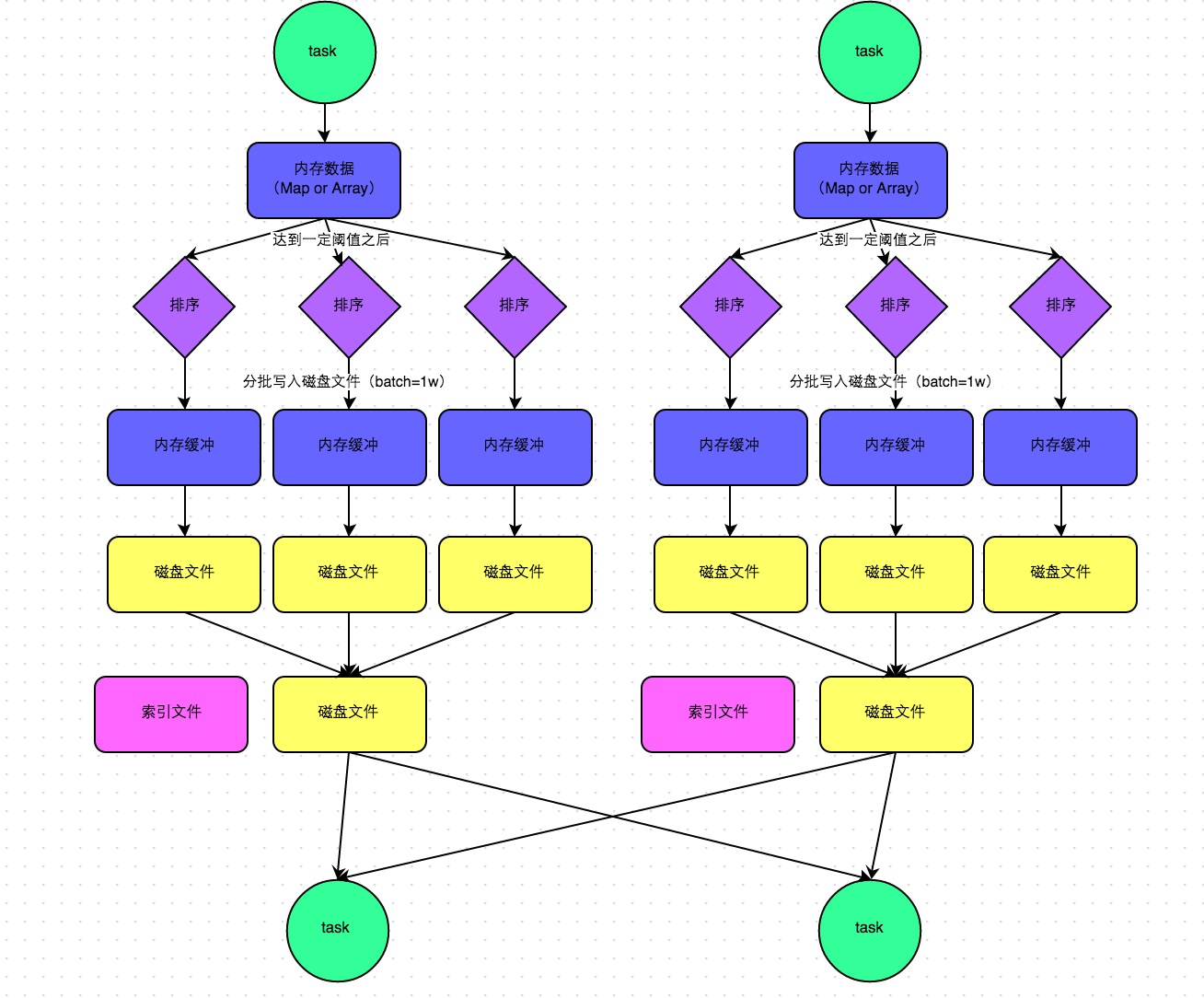
普通运行机制

下图说明了普通的SortShuffleManager的原理。在该模式下，数据会先写入一个内存数据结构中，此时根据不同的shuffle算子，可能选用不同的数据结构。如果是reduceByKey这种聚合类的shuffle算子，那么会选用Map数据结构，一边通过Map进行聚合，一边写入内存；如果是join这种普通的shuffle算子，那么会选用Array数据结构，直接写入内存。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。

在溢写到磁盘文件之前，会先根据key对内存数据结构中已有的数据进行排序。排序过后，会分批将数据写入磁盘文件。默认的batch数量是10000条，也就是说，排序好的数据，会以每批1万条数据的形式分批写入磁盘文件。写入磁盘文件是通过Java的BufferedOutputStream实现的。BufferedOutputStream是Java的缓冲输出流，首先会将数据缓冲在内存中，当内存缓冲满溢之后再一次写入磁盘文件中，这样可以减少磁盘IO次数，提升性能。

一个task将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也就会产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，这就是merge过程，此时会将之前所有临时磁盘文件中的数据读取出来，然后依次写入最终的磁盘文件之中。此外，由于一个task就只对应一个磁盘文件，也就意味着该task为下游stage的task准备的数据都在这一个文件中，因此还会单独写一份索引文件，其中标识了下游各个task的数据在文件中的start offset与end offset。

SortShuffleManager由于有一个磁盘文件merge的过程，因此大大减少了文件数量。比如第一个stage有50个task，总共有10个Executor，每个Executor执行5个task，而第二个stage有100个task。由于每个task最终只有一个磁盘文件，因此此时每个Executor上只有5个磁盘文件，所有Executor只有50个磁盘文件。



bypass运行机制

下图说明了bypass SortShuffleManager的原理。bypass运行机制的触发条件如下：

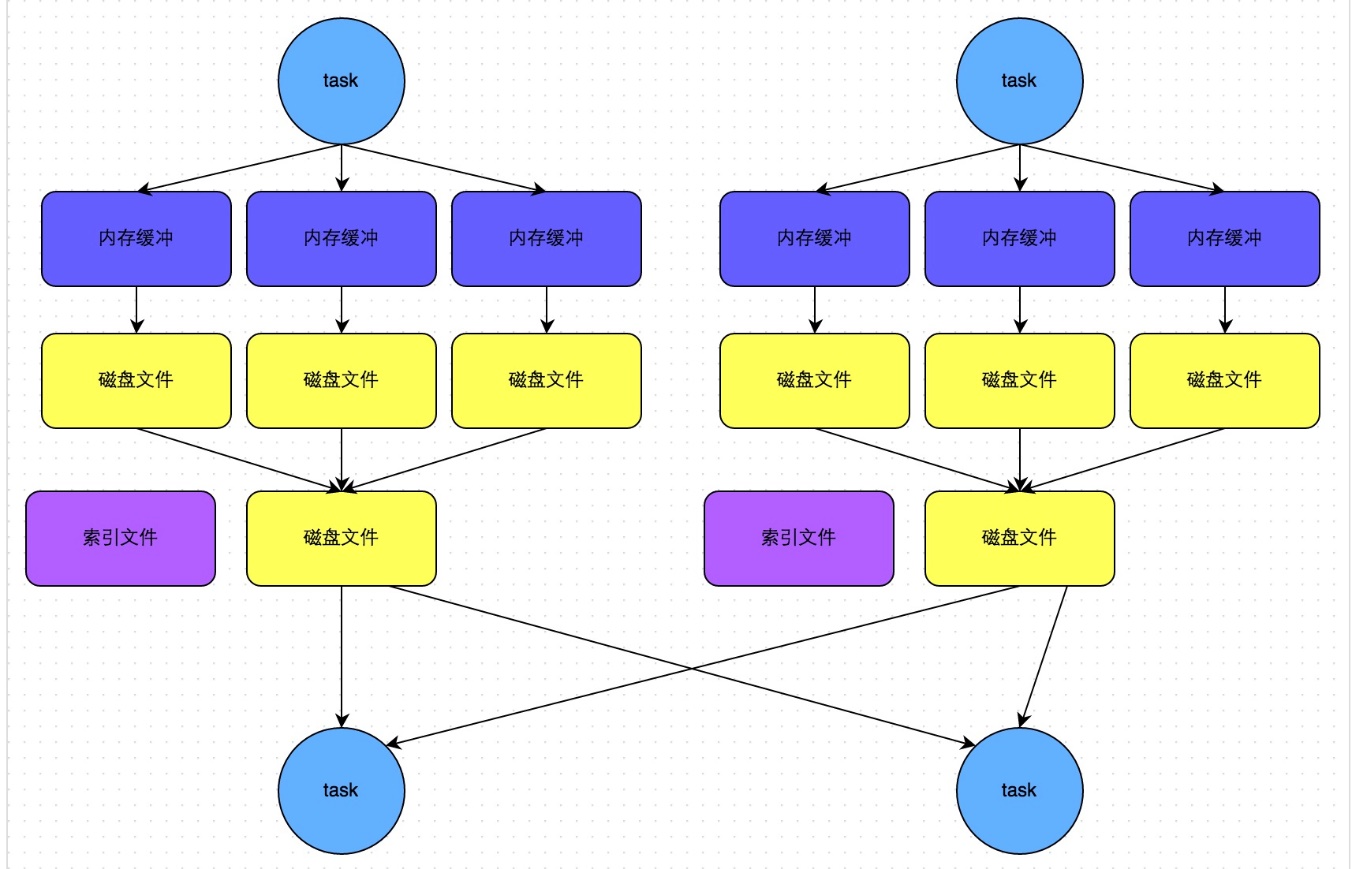
shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值。

不是聚合类的shuffle算子（比如reduceByKey）。

此时task会为每个下游task都创建一个临时磁盘文件，并将数据按key进行hash然后根据key的hash值，将key写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的HashShuffleManager是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的HashShuffleManager来说，shuffle read的性能会更好。

而该机制与普通SortShuffleManager运行机制的不同在于：第一，磁盘写机制不同；第二，不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。



### shuffle相关参数调优

以下是Shffule过程中的一些主要参数，这里详细讲解了各个参数的功能、默认值以及基于实践经验给出的调优建议。

spark.shuffle.file.buffer

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%~5%的提升。

spark.reducer.maxSizeInFlight

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，可以适当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%~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，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，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60次），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，对于针对超大数据量（数十亿~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：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是5s。

调优建议：建议加大间隔时长（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：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，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%左右。

spark.shuffle.manager

默认值：sort

参数说明：该参数用于设置ShuffleManager的类型。Spark 1.5以后，有三个可选项：hash、sort和tungsten-sort。HashShuffleManager是Spark 1.2以前的默认选项，但是Spark 1.2以及之后的版本默认都是SortShuffleManager了。tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率更高。

调优建议：由于SortShuffleManager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SortShuffleManager就可以；而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HashShuffleManager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（默认是200），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bypass机制，map-side就不会进行排序了，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consolidate机制，会大幅度合并shuffle 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SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为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%~30%。

## 程序开发开发调优

Spark性能优化的第一步，就是要在开发Spark作业的过程中注意和应用一些性能优化的基本原则。开发调优，就是要让大家了解以下一些Spark基本开发原则，包括：RDD lineage设计、算子的合理使用、特殊操作的优化等。在开发过程中，时时刻刻都应该注意以上原则，并将这些原则根据具体的业务以及实际的应用场景，灵活地运用到自己的Spark作业中。

### 原则一：避免创建重复的RDD

通常来说，我们在开发一个Spark作业时，首先是基于某个数据源（比如Hive表或HDFS文件）创建一个初始的RDD；接着对这个RDD执行某个算子操作，然后得到下一个RDD；以此类推，循环往复，直到计算出最终我们需要的结果。在这个过程中，多个RDD会通过不同的算子操作（比如map、reduce等）串起来，这个“RDD串”，就是RDD lineage，也就是“RDD的血缘关系链”。

我们在开发过程中要注意：对于同一份数据，只应该创建一个RDD，不能创建多个RDD来代表同一份数据。

一些Spark初学者在刚开始开发Spark作业时，或者是有经验的工程师在开发RDD lineage极其冗长的Spark作业时，可能会忘了自己之前对于某一份数据已经创建过一个RDD了，从而导致对于同一份数据，创建了多个RDD。这就意味着，我们的Spark作业会进行多次重复计算来创建多个代表相同数据的RDD，进而增加了作业的性能开销。

一个简单的例子

|  |
| --- |
| *//需要对名为“hello.txt”的HDFS文件进行一次map操作，再进行一次reduce操作。 //也就是说，需要对一份数据执行两次算子操作。 //错误的做法：对于同一份数据执行多次算子操作时，创建多个RDD。 //这里执行了两次textFile方法，针对同一个HDFS文件，创建了两个RDD出来， //然后分别对每个RDD都执行了一个算子操作。 //这种情况下，Spark需要从HDFS上两次加载hello.txt文件的内容，并创建两个单独的RDD； //第二次加载HDFS文件以及创建RDD的性能开销，很明显是白白浪费掉的。* **val** *rdd1* = sc.textFile(**"hdfs://192.168.0.1:9000/hello.txt"**) *rdd1*.map(...) **val** *rdd2* = sc.textFile(**"hdfs://192.168.0.1:9000/hello.txt"**) *rdd2*.reduce(...)  *//正确的用法：对于一份数据执行多次算子操作时，只使用一个RDD。 //这种写法很明显比上一种写法要好多了，因为我们对于同一份数据只创建了一个RDD， //然后对这一个RDD执行了多次算子操作。 //但是要注意到这里为止优化还没有结束，由于rdd1被执行了两次算子操作，第二次执行reduce操作的时候， //还会再次从源头处重新计算一次rdd1的数据，因此还是会有重复计算的性能开销。 //要彻底解决这个问题，必须结合“原则三：对多次使用的RDD进行持久化”， //才能保证一个RDD被多次使用时只被计算一次。* **val** *rdd1* = sc.textFile(**"hdfs://192.168.0.1:9000/hello.txt"**) *rdd1*.map(...) *rdd1*.reduce(...) |

### 原则二：尽可能复用同一个RDD

除了要避免在开发过程中对一份完全相同的数据创建多个RDD之外，在对不同的数据执行算子操作时还要尽可能地复用一个RDD。比如说，有一个RDD的数据格式是key-value类型的，另一个是单value类型的，这两个RDD的value数据是完全一样的。那么此时我们可以只使用key-value类型的那个RDD，因为其中已经包含了另一个的数据。对于类似这种多个RDD的数据有重叠或者包含的情况，我们应该尽量复用一个RDD，这样可以尽可能地减少RDD的数量，从而尽可能减少算子执行的次数。

一个简单的例子

|  |
| --- |
| *// 错误的做法。 // 有一个<long , String>格式的RDD，即rdd1。 // 接着由于业务需要，对rdd1执行了一个map操作，创建了一个rdd2， //而rdd2中的数据仅仅是rdd1中的value值而已，也就是说，rdd2是rdd1的子集。* JavaPairRDD<long , String> rdd1 = ... JavaRDD<string> rdd2 = rdd1.map(...)  *// 分别对rdd1和rdd2执行了不同的算子操作。* rdd1.reduceByKey(...) rdd2.map(...)  *// 正确的做法。 // 上面这个case中，其实rdd1和rdd2的区别无非就是数据格式不同而已， //rdd2的数据完全就是rdd1的子集而已，却创建了两个rdd，并对两个rdd都执行了一次算子操作。 // 此时会因为对rdd1执行map算子来创建rdd2，而多执行一次算子操作，进而增加性能开销。 // 其实在这种情况下完全可以复用同一个RDD。 // 我们可以使用rdd1，既做reduceByKey操作，也做map操作。 // 在进行第二个map操作时，只使用每个数据的tuple.\_2，也就是rdd1中的value值，即可。* JavaPairRDD<long , String> rdd1 = ... rdd1.reduceByKey(...) rdd1.map(tuple.\_2...) *// 第二种方式相较于第一种方式而言，很明显减少了一次rdd2的计算开销。 // 但是到这里为止，优化还没有结束，对rdd1我们还是执行了两次算子操作，rdd1实际上还是会被计算两次。 // 因此还需要配合“原则三：对多次使用的RDD进行持久化”进行使用， //才能保证一个RDD被多次使用时只被计算一次。* |

### 原则三：对多次使用的RDD进行持久化

当你在Spark代码中多次对一个RDD做了算子操作后，你已经实现Spark作业第一步的优化了，也就是尽可能复用RDD时就该在这个基础之上，进行第二步优化了，也就是要保证对一个RDD执行多次算子操作时，这个RDD本身仅仅被计算一次。

Spark中对于一个RDD执行多次算子的默认原理是这样的：每次你对一个RDD执行一个算子操作时，都会重新从源头处计算一遍，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。

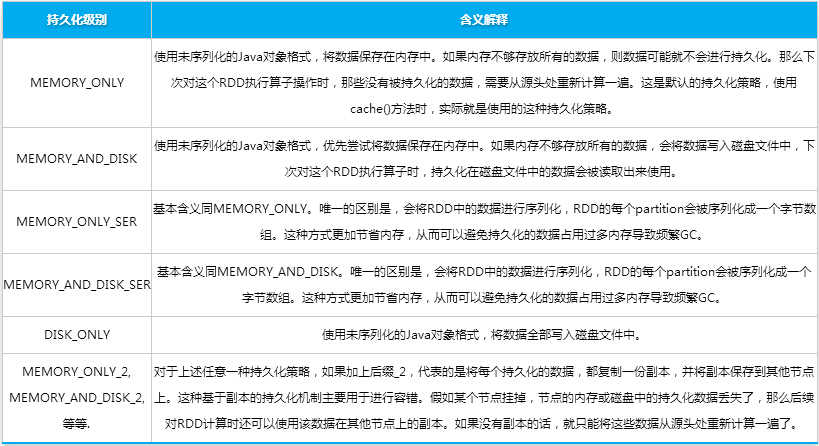
因此对于这种情况，我们的建议是：对多次使用的RDD进行持久化。此时Spark就会根据你的持久化策略，将RDD中的数据保存到内存或者磁盘中。以后每次对这个RDD进行算子操作时，都会直接从内存或磁盘中提取持久化的RDD数据，然后执行算子，而不从源头处重新计算一遍这个RDD，再执行算子操作。

对多次使用的RDD进行持久化的代码示例

|  |
| --- |
| *// 如果要对一个RDD进行持久化，只要对这个RDD调用cache()和persist()即可。 // 正确的做法。 // cache()方法表示：使用非序列化的方式将RDD中的数据全部尝试持久化到内存中。 // 此时再对rdd1执行两次算子操作时，只有在第一次执行map算子时，才会将这个rdd1从源头处计算一次。 // 第二次执行reduce算子时，就会直接从内存中提取数据进行计算，不会重复计算一个rdd。* **val** *rdd1* = sc.textFile(**"hdfs://192.168.0.1:9000/hello.txt"**).cache() *rdd1*.map(...) *rdd1*.reduce(...)  *// persist()方法表示：手动选择持久化级别，并使用指定的方式进行持久化。 // 比如说，StorageLevel.MEMORY\_AND\_DISK\_SER表示，内存充足时优先持久化到内存中， //内存不充足时持久化到磁盘文件中。 // 而且其中的\_SER后缀表示，使用序列化的方式来保存RDD数据，此时RDD中的每个partition //都会序列化成一个大的字节数组，然后再持久化到内存或磁盘中。 // 序列化的方式可以减少持久化的数据对内存/磁盘的占用量，进而避免内存被持久化数据占用过多， //从而发生频繁GC。* **val** *rdd1* = sc.textFile(**"hdfs://192.168.0.1:9000/hello.txt"**)  .persist(StorageLevel.MEMORY\_AND\_DISK\_SER) *rdd1*.map(...) *rdd1*.reduce(...) |

对于persist()方法而言，我们可以根据不同的业务场景选择不同的持久化级别。

Spark的持久化级别



MEMORY\_ONLY 使用未序列化的Java对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据可能就不会进行持久化。那么下次对这个RDD执行算子操作时，那些没有被持久化的数据，需要从源头处重新计算一遍。这是默认的持久化策略，使用cache()方法时，实际就是使用的这种持久化策略。

MEMORY\_AND\_DISK 使用未序列化的Java对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据写入磁盘文件中，下次对这个RDD执行算子时，持久化在磁盘文件中的数据会被读取出来使用。

MEMORY\_ONLY\_SER 基本含义同MEMORY\_ONLY。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。

MEMORY\_AND\_DISK\_SER 基本含义同MEMORY\_AND\_DISK。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。

DISK\_ONLY 使用未序列化的Java对象格式，将数据全部写入磁盘文件中。

MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, 等等. 对于上述任意一种持久化策略，如果加上后缀\_2，代表的是将每个持久化的数据，都复制一份副本，并将副本保存到其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中的持久化数据丢失了，那么后续对RDD计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将这些数据从源头处重新计算一遍了。

如何选择一种最合适的持久化策略

1、默认情况下，性能最高的当然是MEMORY\_ONLY，但前提是你的内存必须足够足够大，可以绰绰有余地存放下整个RDD的所有数据。因为不进行序列化与反序列化操作，就避免了这部分的性能开销；对这个RDD的后续算子操作，都是基于纯内存中的数据的操作，不需要从磁盘文件中读取数据，性能也很高；而且不需要复制一份数据副本，并远程传送到其他节点上。但是这里必须要注意的是，在实际的生产环境中，恐怕能够直接用这种策略的场景还是有限的，如果RDD中数据比较多时（比如几十亿），直接用这种持久化级别，会导致JVM的OOM内存溢出异常。

2、如果使用MEMORY\_ONLY级别时发生了内存溢出，那么建议尝试使用MEMORY\_ONLY\_SER级别。该级别会将RDD数据序列化后再保存在内存中，此时每个partition仅仅是一个字节数组而已，大大减少了对象数量，并降低了内存占用。这种级别比MEMORY\_ONLY多出来的性能开销，主要就是序列化与反序列化的开销。但是后续算子可以基于纯内存进行操作，因此性能总体还是比较高的。此外，可能发生的问题同上，如果RDD中的数据量过多的话，还是可能会导致OOM内存溢出的异常。

3、如果纯内存的级别都无法使用，那么建议使用MEMORY\_AND\_DISK\_SER策略，而不是MEMORY\_AND\_DISK策略。因为既然到了这一步，就说明RDD的数据量很大，内存无法完全放下。序列化后的数据比较少，可以节省内存和磁盘的空间开销。同时该策略会优先尽量尝试将数据缓存在内存中，内存缓存不下才会写入磁盘。

4、通常不建议使用DISK\_ONLY和后缀为\_2的级别：因为完全基于磁盘文件进行数据的读写，会导致性能急剧降低，有时还不如重新计算一次所有RDD。后缀为\_2的级别，必须将所有数据都复制一份副本，并发送到其他节点上，数据复制以及网络传输会导致较大的性能开销，除非是要求作业的高可用性，否则不建议使用。

### 原则四：尽量避免使用shuffle类算子

如果有可能的话，要尽量避免使用shuffle类算子。因为Spark作业运行过程中，最消耗性能的地方就是shuffle过程。shuffle过程，简单来说，就是将分布在集群中多个节点上的同一个key，拉取到同一个节点上，进行聚合或join等操作。比如reduceByKey、join等算子，都会触发shuffle操作。

shuffle过程中，各个节点上的相同key都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中的相同key。而且相同key都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的key过多，导致内存不够存放，进而溢写到磁盘文件中。因此在shuffle过程中，可能会发生大量的磁盘文件读写的IO操作，以及数据的网络传输操作。磁盘IO和网络数据传输也是shuffle性能较差的主要原因。

因此在我们的开发过程中，能避免则尽可能避免使用reduceByKey、join、distinct、repartition等会进行shuffle的算子，尽量使用map类的非shuffle算子。这样的话，没有shuffle操作或者仅有较少shuffle操作的Spark作业，可以大大减少性能开销。

Broadcast与map进行join代码示例

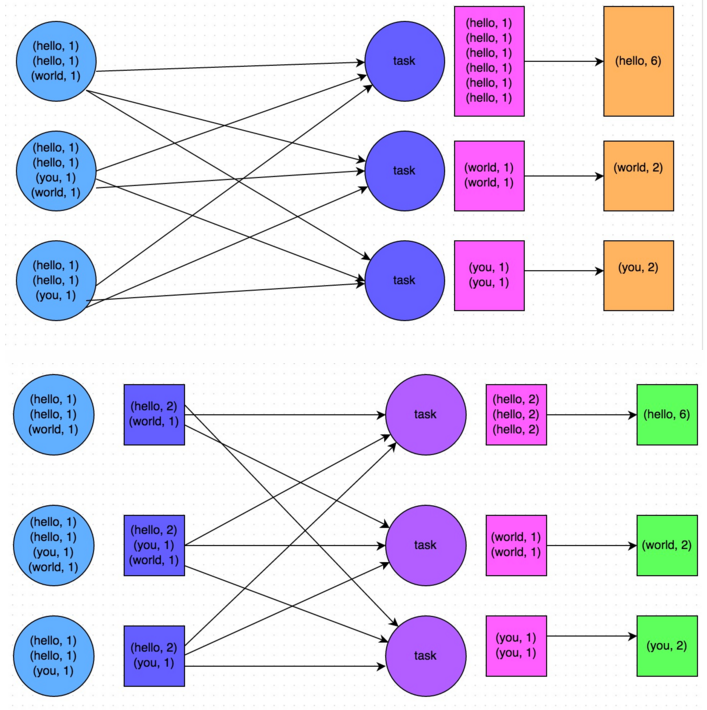
|  |
| --- |
| *// 传统的join操作会导致shuffle操作。 // 因为两个RDD中，相同的key都需要通过网络拉取到一个节点上，由一个task进行join操作。* **val** *rdd3* = rdd1.join(rdd2) *// Broadcast+map的join操作，不会导致shuffle操作。 // 使用Broadcast将一个数据量较小的RDD作为广播变量。* **val** *rdd2Data* = rdd2.collect() **val** *rdd2DataBroadcast* = sc.broadcast(*rdd2Data*) *// 在rdd1.map算子中，可以从rdd2DataBroadcast中，获取rdd2的所有数据。 // 然后进行遍历，如果发现rdd2中某条数据的key与rdd1的当前数据的key是相同的， //那么就判定可以进行join。 // 此时就可以根据自己需要的方式，将rdd1当前数据与rdd2中可以连接的数据， //拼接在一起（String或Tuple）。* **val** *rdd3* = rdd1.map(*rdd2DataBroadcast*...) *// 注意，以上操作，建议仅仅在rdd2的数据量比较少（比如几百M，或者一两G）的情况下使用。 // 因为每个Executor的内存中，都会驻留一份rdd2的全量数据。* |

### 原则五：使用map-side预聚合的shuffle操作

如果因为业务需要，一定要使用shuffle操作，无法用map类的算子来替代，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MapReduce中的本地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数量，从而也就减少了磁盘IO以及网络传输开销。通常来说，在可能的情况下，建议使用reduceByKey或者aggregateByKey算子来替代掉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的相同key进行预聚合。而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。

比如下图，就是典型的例子，分别基于reduceByKey和groupByKey进行单词计数。其中第一张图是groupByKey的原理图，可以看到，没有进行任何本地聚合时，所有数据都会在集群节点之间传输；第二张图是reduceByKey的原理图，可以看到，每个节点本地的相同key数据，都进行了预聚合，然后才传输到其他节点上进行全局聚合。



### 原则六：使用高性能的算子

除了shuffle相关的算子有优化原则之外，其他的算子也都有着相应的优化原则。

使用reduceByKey/aggregateByKey替代groupByKey

详情见“原则五：使用map-side预聚合的shuffle操作”。

使用mapPartitions替代普通map

mapPartitions类的算子，一次函数调用会处理一个partition所有的数据，而不是一次函数调用处理一条，性能相对来说会高一些。但是有的时候，使用mapPartitions会出现OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个partition所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现OOM异常。所以使用这类操作时要慎重！

使用foreachPartitions替代foreach

原理类似于“使用mapPartitions替代map”，也是一次函数调用处理一个partition的所有数据，而不是一次函数调用处理一条数据。在实践中发现，foreachPartitions类的算子，对性能的提升还是很有帮助的。比如在foreach函数中，将RDD中所有数据写MySQL，那么如果是普通的foreach算子，就会一条数据一条数据地写，每次函数调用可能就会创建一个数据库连接，此时就势必会频繁地创建和销毁数据库连接，性能是非常低下；但是如果用foreachPartitions算子一次性处理一个partition的数据，那么对于每个partition，只要创建一个数据库连接即可，然后执行批量插入操作，此时性能是比较高的。实践中发现，对于1万条左右的数据量写MySQL，性能可以提升30%以上。

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中较多数据后（比如30%以上的数据），建议使用coalesce算子，手动减少RDD的partition数量，将RDD中的数据压缩到更少的partition中去。因为filter之后，RDD的每个partition中都会有很多数据被过滤掉，此时如果照常进行后续的计算，其实每个task处理的partition中的数据量并不是很多，有一点资源浪费，而且此时处理的task越多，可能速度反而越慢。因此用coalesce减少partition数量，将RDD中的数据压缩到更少的partition之后，只要使用更少的task即可处理完所有的partition。在某些场景下，对于性能的提升会有一定的帮助。

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repartitionAndSortWithinPartitions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与sort两个操作同时进行，比先shuffle再sort来说，性能可能是要高的。

### 原则七：广播大变量

有时在开发过程中，会遇到需要在算子函数中使用外部变量的场景（尤其是大变量，比如100M以上的大集合），那么此时就应该使用Spark的广播（Broadcast）功能来提升性能。

在算子函数中使用到外部变量时，默认情况下，Spark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的话（比如100M，甚至1G），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中占用过多内存导致的频繁GC，都会极大地影响性能。

因此对于上述情况，如果使用的外部变量比较大，建议使用Spark的广播功能，对该变量进行广播。广播后的变量，会保证每个Executor的内存中，只驻留一份变量副本，而Executor中的task执行时共享该Executor中的那份变量副本。这样的话，可以大大减少变量副本的数量，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。

广播大变量的代码示例

|  |
| --- |
| *// 以下代码在算子函数中，使用了外部的变量。 // 此时没有做任何特殊操作，每个task都会有一份list1的副本。* **val** *list1* = ... rdd1.map(*list1*...) *// 以下代码将list1封装成了Broadcast类型的广播变量。 // 在算子函数中，使用广播变量时，首先会判断当前task所在Executor内存中，是否有变量副本。 // 如果有则直接使用；如果没有则从Driver或者其他Executor节点上远程拉取一份放到本地Executor内存中。 // 每个Executor内存中，就只会驻留一份广播变量副本。* **val** *list1* = ... **val** *list1Broadcast* = sc.broadcast(*list1*) rdd1.map(*list1Broadcast*...) |

### 原则八：使用Kryo优化序列化性能

在Spark中，主要有三个地方涉及到了序列化：

1、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（见“原则七：广播大变量”中的讲解）。

2、将自定义的类型作为RDD的泛型类型时（比如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接口。

3、使用可序列化的持久化策略时（比如MEMORY\_ONLY\_SER），Spark会将RDD中的每个partition都序列化成一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过使用Kryo序列化类库，来优化序列化和反序列化的性能。Spark默认使用的是Java的序列化机制，也就是ObjectOutputStream/ObjectInputStream API来进行序列化和反序列化。但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。官方介绍，Kryo序列化机制比Java序列化机制，性能高10倍左右。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。

以下是使用Kryo的代码示例，我们只要设置序列化类，再注册要序列化的自定义类型即可（比如算子函数中使用到的外部变量类型、作为RDD泛型类型的自定义类型等）：

|  |
| --- |
| *// 创建SparkConf对象。* **val** *conf* = **new** SparkConf().setMaster(...).setAppName(...) *// 设置序列化器为KryoSerializer。 conf*.set(**"spark.serializer"**, **"org.apache.spark.serializer.KryoSerializer"**) *// 注册要序列化的自定义类型。 conf*.registerKryoClasses(*Array*(*classOf*[MyClass1], *classOf*[MyClass2])) |

### 原则九：优化数据结构

Java中，有三种类型比较耗费内存：

1、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。

2、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。

3、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，特别是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。

## 运行资源调优

### 调优概述

在开发完Spark作业之后，就该为作业配置合适的资源了。Spark的资源参数，基本都可以在spark-submit命令中作为参数设置。很多Spark初学者，通常不知道该设置哪些必要的参数，以及如何设置这些参数，最后就只能胡乱设置，甚至压根儿不设置。资源参数设置的不合理，可能会导致没有充分利用集群资源，作业运行会极其缓慢；或者设置的资源过大，队列没有足够的资源来提供，进而导致各种异常。总之，无论是哪种情况，都会导致Spark作业的运行效率低下，甚至根本无法运行。因此我们必须对Spark作业的资源使用原理有一个清晰的认识，并知道在Spark作业运行过程中，有哪些资源参数是可以设置的，以及如何设置合适的参数值。

### Spark作业基本运行原理



详细原理见上图。我们使用spark-submit提交一个Spark作业之后，这个作业就会启动一个对应的Driver进程。根据你使用的部署模式（deploy-mode）不同，Driver进程可能在本地启动，也可能在集群中某个工作节点上启动。Driver进程本身会根据我们设置的参数，占有一定数量的内存和CPU core。而Driver进程要做的第一件事情，就是向集群管理器（可以是Spark Standalone集群，也可以是其他的资源管理集群，美团、大众点评使用的是YARN作为资源管理集群）申请运行Spark作业需要使用的资源，这里的资源指的就是Executor进程。YARN集群管理器会根据我们为Spark作业设置的资源参数，在各个工作节点上，启动一定数量的Executor进程，每个Executor进程都占有一定数量的内存和CPU core。

在申请到了作业执行所需的资源之后，Driver进程就会开始调度和执行我们编写的作业代码了。Driver进程会将我们编写的Spark作业代码分拆为多个stage，每个stage执行一部分代码片段，并为每个stage创建一批task，然后将这些task分配到各个Executor进程中执行。task是最小的计算单元，负责执行一模一样的计算逻辑（也就是我们自己编写的某个代码片段），只是每个task处理的数据不同而已。一个stage的所有task都执行完毕之后，会在各个节点本地的磁盘文件中写入计算中间结果，然后Driver就会调度运行下一个stage。下一个stage的task的输入数据就是上一个stage输出的中间结果。如此循环往复，直到将我们自己编写的代码逻辑全部执行完，并且计算完所有的数据，得到我们想要的结果为止。

Spark是根据shuffle类算子来进行stage的划分。如果我们的代码中执行了某个shuffle类算子（比如reduceByKey、join等），那么就会在该算子处，划分出一个stage界限来。可以大致理解为，shuffle算子执行之前的代码会被划分为一个stage，shuffle算子执行以及之后的代码会被划分为下一个stage。因此一个stage刚开始执行的时候，它的每个task可能都会从上一个stage的task所在的节点，去通过网络传输拉取需要自己处理的所有key，然后对拉取到的所有相同的key使用我们自己编写的算子函数执行聚合操作（比如reduceByKey()算子接收的函数）。这个过程就是shuffle。

当我们在代码中执行了cache/persist等持久化操作时，根据我们选择的持久化级别的不同，每个task计算出来的数据也会保存到Executor进程的内存或者所在节点的磁盘文件中。

因此Executor的内存主要分为三块：第一块是让task执行我们自己编写的代码时使用，默认是占Executor总内存的20%；第二块是让task通过shuffle过程拉取了上一个stage的task的输出后，进行聚合等操作时使用，默认也是占Executor总内存的20%；第三块是让RDD持久化时使用，默认占Executor总内存的60%。

task的执行速度是跟每个Executor进程的CPU core数量有直接关系的。一个CPU core同一时间只能执行一个线程。而每个Executor进程上分配到的多个task，都是以每个task一条线程的方式，多线程并发运行的。如果CPU core数量比较充足，而且分配到的task数量比较合理，那么通常来说，可以比较快速和高效地执行完这些task线程。

以上就是Spark作业的基本运行原理的说明，大家可以结合上图来理解。理解作业基本原理，是我们进行资源参数调优的基本前提。

### 资源参数调优

了解完了Spark作业运行的基本原理之后，对资源相关的参数就容易理解了。所谓的Spark资源参数调优，其实主要就是对Spark运行过程中各个使用资源的地方，通过调节各种参数，来优化资源使用的效率，从而提升Spark作业的执行性能。以下参数就是Spark中主要的资源参数，每个参数都对应着作业运行原理中的某个部分，我们同时也给出了一个调优的参考值。

num-executors

参数说明：该参数用于设置Spark作业总共要用多少个Executor进程来执行。Driver在向YARN集群管理器申请资源时，YARN集群管理器会尽可能按照你的设置来在集群的各个工作节点上，启动相应数量的Executor进程。这个参数非常之重要，如果不设置的话，默认只会给你启动少量的Executor进程，此时你的Spark作业的运行速度是非常慢的。

参数调优建议：每个Spark作业的运行一般设置50~100个左右的Executor进程比较合适，设置太少或太多的Executor进程都不好。设置的太少，无法充分利用集群资源；设置的太多的话，大部分队列可能无法给予充分的资源。

executor-memory

参数说明：该参数用于设置每个Executor进程的内存。Executor内存的大小，很多时候直接决定了Spark作业的性能，而且跟常见的JVM OOM异常，也有直接的关联。

参数调优建议：每个Executor进程的内存设置4G~8G较为合适。但是这只是一个参考值，具体的设置还是得根据不同部门的资源队列来定。可以看看自己团队的资源队列的最大内存限制是多少，num-executors乘以executor-memory，是不能超过队列的最大内存量的。此外，如果你是跟团队里其他人共享这个资源队列，那么申请的内存量最好不要超过资源队列最大总内存的1/3~1/2，避免你自己的Spark作业占用了队列所有的资源，导致别的同学的作业无法运行。

executor-cores

参数说明：该参数用于设置每个Executor进程的CPU core数量。这个参数决定了每个Executor进程并行执行task线程的能力。因为每个CPU core同一时间只能执行一个task线程，因此每个Executor进程的CPU core数量越多，越能够快速地执行完分配给自己的所有task线程。

参数调优建议：Executor的CPU core数量设置为2~4个较为合适。同样得根据不同部门的资源队列来定，可以看看自己的资源队列的最大CPU core限制是多少，再依据设置的Executor数量，来决定每个Executor进程可以分配到几个CPU core。同样建议，如果是跟他人共享这个队列，那么num-executors \* executor-cores不要超过队列总CPU core的1/3~1/2左右比较合适，也是避免影响其他同学的作业运行。

driver-memory

参数说明：该参数用于设置Driver进程的内存。

参数调优建议：Driver的内存通常来说不设置，或者设置1G左右应该就够了。唯一需要注意的一点是，如果需要使用collect算子将RDD的数据全部拉取到Driver上进行处理，那么必须确保Driver的内存足够大，否则会出现OOM内存溢出的问题。

spark.default.parallelism

参数说明：该参数用于设置每个stage的默认task数量。这个参数极为重要，如果不设置可能会直接影响你的Spark作业性能。

参数调优建议：Spark作业的默认task数量为500~1000个较为合适。很多同学常犯的一个错误就是不去设置这个参数，那么此时就会导致Spark自己根据底层HDFS的block数量来设置task的数量，默认是一个HDFS block对应一个task。通常来说，Spark默认设置的数量是偏少的（比如就几十个task），如果task数量偏少的话，就会导致你前面设置好的Executor的参数都前功尽弃。试想一下，无论你的Executor进程有多少个，内存和CPU有多大，但是task只有1个或者10个，那么90%的Executor进程可能根本就没有task执行，也就是白白浪费了资源！因此Spark官网建议的设置原则是，设置该参数为num-executors \* executor-cores的2~3倍较为合适，比如Executor的总CPU core数量为300个，那么设置1000个task是可以的，此时可以充分地利用Spark集群的资源。

spark.storage.memoryFraction

参数说明：该参数用于设置RDD持久化数据在Executor内存中能占的比例，默认是0.6。也就是说，默认Executor 60%的内存，可以用来保存持久化的RDD数据。根据你选择的不同的持久化策略，如果内存不够时，可能数据就不会持久化，或者数据会写入磁盘。

参数调优建议：如果Spark作业中，有较多的RDD持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致数据只能写入磁盘中，降低了性能。但是如果Spark作业中的shuffle类操作比较多，而持久化操作比较少，那么这个参数的值适当降低一些比较合适。此外，如果发现作业由于频繁的gc导致运行缓慢（通过spark web ui可以观察到作业的gc耗时），意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

spark.shuffle.memoryFraction

参数说明：该参数用于设置shuffle过程中一个task拉取到上个stage的task的输出后，进行聚合操作时能够使用的Executor内存的比例，默认是0.2。也就是说，Executor默认只有20%的内存用来进行该操作。shuffle操作在进行聚合时，如果发现使用的内存超出了这个20%的限制，那么多余的数据就会溢写到磁盘文件中去，此时就会极大地降低性能。

参数调优建议：如果Spark作业中的RDD持久化操作较少，shuffle操作较多时，建议降低持久化操作的内存占比，提高shuffle操作的内存占比比例，避免shuffle过程中数据过多时内存不够用，必须溢写到磁盘上，降低了性能。此外，如果发现作业由于频繁的gc导致运行缓慢，意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

资源参数的调优，没有一个固定的值，需要同学们根据自己的实际情况（包括Spark作业中的shuffle操作数量、RDD持久化操作数量以及spark web ui中显示的作业gc情况），同时参考给出的原理以及调优建议，合理地设置上述参数。

资源参数参考示例

以下是一份spark-submit命令的示例，大家可以参考一下，并根据自己的实际情况进行调节：

|  |
| --- |
| ./bin/spark-submit \  --master yarn-cluster \  --num-executors 100 \  --executor-memory 6G \  --executor-cores 4 \  --driver-memory 1G \  --conf spark.default.parallelism=1000 \  --conf spark.storage.memoryFraction=0.5 \  --conf spark.shuffle.memoryFraction=0.3 \ |

## GC调优

Spark立足内存计算，常常需要在内存中存放大量数据，因此也更依赖JVM的垃圾回收机制。与此同时，它也兼容批处理和流式处理，对于程序吞吐量和延迟都有较高要求，因此GC参数的调优在Spark应用实践中显得尤为重要。

按照经验来说，当我们配置垃圾收集器时，主要有两种策略——Parallel GC和CMS GC。前者注重更高的吞吐量，而后者则注重更低的延迟。两者似乎是鱼和熊掌，不能兼得。在实际应用中，我们只能根据应用对性能瓶颈的侧重性，来选取合适的垃圾收集器。例如，当我们运行需要有实时响应的场景的应用时，我们一般选用CMS GC，而运行一些离线分析程序时，则选用Parallel GC。那么对于Spark这种既支持流式计算，又支持传统的批处理运算的计算框架来说，是否存在一组通用的配置选项呢？

通常CMS GC是企业比较常用的GC配置方案，并在长期实践中取得了比较好的效果。例如对于进程中若存在大量寿命较长的对象，Parallel GC经常带来较大的性能下降。因此，即使是批处理的程序也能从CMS GC中获益。不过，在从1.6开始的HOTSPOT JVM中，我们发现了一个新的GC设置项：Garbage-First GC(G1 GC)，Oracle将其定位为CMS GC的长期演进。

### JVM虚拟机

每个Java开发者都知道Java字节码是执行在JRE(Java Runtime Environment Java运行时环境）上的。JRE中最重要的部分是Java虚拟机（JVM），JVM负责分析和执行Java字节码。Java开发人员并不需要去关心JVM是如何运行的。在没有深入理解JVM的情况下，许多开发者已经开发出了非常多的优秀的应用以及Java类库。不过，如果你了解JVM的话，你会更加了解Java的，并且你会轻松解决那些看似简单但是无从下手的问题。

#### 虚拟机（Virtual Machine)

JRE是由Java API和JVM组成的。JVM的主要作用是通过Class Loader来加载Java程序，并且按照Java API来执行加载的程序。

虚拟机是通过软件的方式来模拟实现的机器（比如说计算机），它可以像物理机一样运行程序。设计虚拟机的初衷是让Java能够通过它来实现WORA(Write Once Run Anywher 一次编译，到处运行），尽管这个目标现在已经被大多数人忽略了。因此，JVM可以在不修改Java代码的情况下，在所有的硬件环境上运行Java字节码。

***Java虚拟机的特点如下：***

1. 基于栈的虚拟机：Intel x86和ARM这两种最常见的计算机体系的机构都是基于寄存器的。不同的是，JVM是基于栈的。
2. 符号引用：除了基本类型以外的数据（类和接口）都是通过符号来引用，而不是通过显式地使用内存地址来引用。
3. 垃圾回收机制：类的实例都是通过用户代码进行创建，并且自动被垃圾回收机制进行回收。
4. 通过对基本类型的清晰定义来保证平台独立性：传统的编程语言，例如C/C++，int类型的大小取决于不同的平台。JVM通过对基本类型的清晰定义来保证它的兼容性以及平台独立性。
5. 网络字节码顺序：Java class文件用网络字节码顺序来进行存储:为了保证和小端的Intel x86架构以及大端的RISC系列的架构保持无关性，JVM使用用于网络传输的网络字节顺序，也就是大端。

虽然是Sun公司开发了Java，但是所有的开发商都可以开发并且提供遵循Java虚拟机规范的JVM。正是由于这个原因，使得Oracle HotSpot和IBM JVM等不同的JVM能够并存。Google的Android系统里的Dalvik VM也是一种JVM，虽然它并不遵循Java虚拟机规范。和基于栈的Java虚拟机不同，Dalvik VM是基于寄存器的架构，因此它的Java字节码也被转化成基于寄存器的指令集。

#### Java字节码（Java bytecode)

为了保证WORA，JVM使用Java字节码这种介于Java和机器语言之间的中间语言。字节码是部署Java代码的最小单位。

在解释Java字节码之前，我们先通过实例来简单了解它。这个案例是一个在开发环境出现的真实案例的总结。

***现象***

一个一直运行正常的应用突然无法运行了。在类库被更新之后，返回下面的错误。

1. Exception in thread "main" java.lang.NoSuchMethodError: com.atguigu.user.UserAdmin.addUser(Ljava/lang/String;)V
2. at com.atguigu.service.UserService.add(UserService.java:14)
3. at com.atguigu.service.UserService.main(UserService.java:19)

应用的代码如下，而且它没有被改动过。

1. // UserService.java
2. …
3. public void add(String userName) {
4. admin.addUser(userName);
5. }

更新后的类库的源代码和原始的代码如下。

1. // UserAdmin.java - Updated library source code
2. …
3. public User addUser(String userName) {
4. User user = new User(userName);
5. User prevUser = userMap.put(userName, user);
6. return prevUser;
7. }
8. // UserAdmin.java - Original library source code
9. …
10. public void addUser(String userName) {
11. User user = new User(userName);
12. userMap.put(userName, user);
13. }

简而言之，之前没有返回值的addUser()被改修改成返回一个User类的实例的方法。不过，应用的代码没有做任何修改，因为它没有使用addUser()的返回值。

咋一看，com.atguigu.user.UserAdmin.addUser()方法似乎仍然存在，如果存在的话，那么怎么还会出现NoSuchMethodError的错误呢？

***原因***

上面问题的原因是在于应用的代码没有用新的类库来进行编译。换句话来说，应用代码似乎是调了正确的方法，只是没有使用它的返回值而已。不管怎样，编译后的class文件表明了这个方法是有返回值的。你可以从下面的错误信息里看到答案。

1. java.lang.NoSuchMethodError: com.aiguigu.user.UserAdmin.addUser(Ljava/lang/String;)V

NoSuchMethodError出现的原因是“com.atguigu.user.UserAdmin.addUser(Ljava/lang/String;)V”方法找不到。注意一下”Ljava/lang/String；”和最后面的“V”。在Java字节码的表达式里，”L<classname>;”表示的是类的实例。这里表示addUser()方法有一个java/lang/String的对象作为参数。在这个类库里，参数没有被改变，所以它是正常的。最后面的“V”表示这个方法的返回值。在Java字节码的表达式里，”V”表示没有返回子（Void）。综上所述，上面的错误信息是表示有一个java.lang.String类型的参数，并且没有返回值的com.atguigu.user.UserAdmin.addUser方法没有找到。

因为应用是用之前的类库编译的，所以返回值为空的方法被调用了。但是在修改后的类库里，返回值为空的方法不存在，并且添加了一个返回值为“Lcom/atguigu/user/User”的方法。因此，就出现了NoSuchMethodError。

这个错误出现的原因是因为开发者没有用新的类库来重新编译应用。不过，出现这种问题的大部分责任在于类库的提供者。这个public的方法本来没有返回值的，但是后来却被修改成返回User类的实例。很明显，方法的签名被修改了，这也表明了这个类库的后向兼容性被破坏了。因此，这个类库的提供者应该告知使用者这个方法已经被改变了。

我们再回到Java字节码上来。Java字节码是JVM很重要的部分。JVM是模拟执行Java字节码的一个模拟器。Java编译器不会直接把高级语言（例如C/C++）编写的代码直接转换成机器语言（CPU指令）；它会把开发者可以理解的Java语言转换成JVM能够理解的Java字节码。因为Java字节码本身是平台无关的，所以它可以在任何安装了JVM（确切地说，是相匹配的JRE）的硬件上执行，即使是在CPU和OS都不相同的平台上（在Windows PC上开发和编译的字节码可以不做任何修改就直接运行在Linux机器上）。编译后的代码的大小和源代码大小基本一致，这样就可以很容易地通过网络来传输和执行编译后的代码。

Java class文件是一种人很难去理解的二进文件。为了便于理解它，JVM提供者提供了javap，反汇编器。使用javap产生的结果是Java汇编语言。在上面的例子中，下面的Java汇编代码是通过javap-c对UserServiceadd()方法进行反汇编得到的。

1. public void add(java.lang.String);
2. Code:
3. 0:   aload\_0
4. 1:   getfield        #15; //Field admin:Lcom/atguigu/user/UserAdmin;
5. 4:   aload\_1
6. 5:   invokevirtual   #23; //Method com/atguigu/user/UserAdmin.addUser:(Ljava/lang/String;)V
7. 8:   return

invokeinterface:调用一个接口方法在这段Java汇编代码中，addUser()方法是在第四行的“5:invokevitual#23″进行调用的。这表示对应索引为23的方法会被调用。索引为23的方法的名称已经被javap给注解在旁边了。invokevirtual是Java字节码里调用方法的最基本的操作码。在Java字节码里，有四种操作码可以用来调用一个方法，分别是：invokeinterface，invokespecial，invokestatic以及invokevirtual。操作码的作用分别如下：

1. invokespecial: 调用一个初始化方法，私有方法或者父类的方法
2. invokestatic:调用静态方法
3. invokevirtual:调用实例方法

Java字节码的指令集由操作码和操作数组成。类似invokevirtual这样的操作数需要2个字节的操作数。

用更新的类库来编译上面的应用代码，然后反编译它，将会得到下面的结果。

1. public void add(java.lang.String);
2. Code:
3. 0:   aload\_0
4. 1:   getfield        #15; //Field admin:Lcom/atguigu/user/UserAdmin;
5. 4:   aload\_1
6. 5:   invokevirtual   #23; //Method com/atguigu/user/UserAdmin.addUser:(Ljava/lang/String;)Lcom/atguigu/user/User;
7. 8:   pop
8. 9:   return

你会发现，对应索引为23的方法被替换成了一个返回值为”Lcom/atguigu/user/User”的方法。

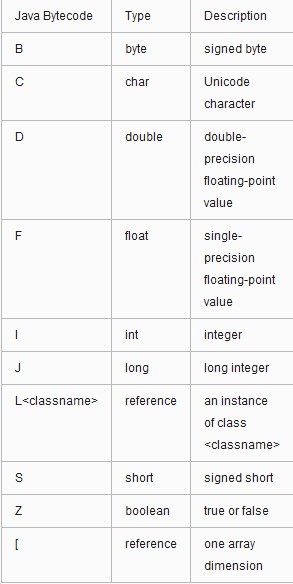
在上面的反汇编代码里，代码前面的数字代码什么呢？

它表示的是字节数。大概这就是为什么运行在JVM上面的代码成为Java“字节”码的原因。简而言之，Java字节码指令的操作码，例如aload\_0，getfield和invokevirtual等，都是用一个字节的数字来表示的（aload\_0=0x2a,getfield=0xb4,invokevirtual=0xb6)。由此可知Java字节码指令的操作码最多有256个。

aload\_0和aload\_1这样的指令不需要任何操作数。因此，aload\_0指令的下一个字节是下一个指令的操作码。不过，getfield和invokevirtual指令需要2字节的操作数。因此，getfiled的下一条指令是跳过两个字节，写在第四个字节的位置上的。十六进制编译器里查看字节码的结果如下所示。

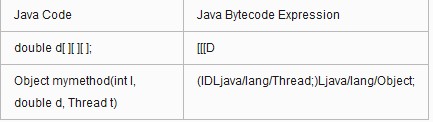
1. 2a b4 00 0f 2b b6 00 17 57 b1

表一：Java字节码中的类型表达式在Java字节码里，类的实例用字母“L;”表示，void 用字母“V”表示。通过这种方式，其他的类型也有对应的表达式。下面的表格对此作了总结。



下面的表格给出了字节码表达式的几个实例。

表二：Java字节码表达式范例



#### Class文件格式

在讲解Java class文件格式之前，我们先看看一个在Java Web应用中经常出现的问题。

当我们编写完Jsp代码，并且在Tomcat运行时，Jsp代码没有正常运行，而是出现了下面的错误。

***现象***

当我们编写完Jsp代码，并且在Tomcat运行时，Jsp代码没有正常运行，而是出现了下面的错误。

1. Servlet.service() for servlet jsp threw exception org.apache.jasper.JasperException: Unable to compile class for JSP Generated servlet error:
2. The code of method \_jspService(HttpServletRequest, HttpServletResponse) is exceeding the 65535 bytes limit"

原因在不同的Web服务器上，上面的错误信息可能会有点不同，不过有有一点肯定是相同的，它出现的原因是65535字节的限制。这个65535字节的限制是JVM规范里的限制，它规定了一个方法的大小不能超过65535字节。

下面我会更加详细地讲解这个65535字节限制的意义以及它出现的原因。

Java字节码里的分支和跳转指令分别是”goto"和"jsr"。

1. goto [branchbyte1] [branchbyte2]
2. jsr [branchbyte1] [branchbyte2]

这两个指令都接收一个2字节的有符号的分支跳转偏移量做为操作数，因此偏移量最大只能达到65535。不过，为了支持更多的跳转，Java字节码提供了"goto\_w"和"jsr\_w"这两个可以接收4字节分支偏移的指令。

1. goto\_w [branchbyte1] [branchbyte2] [branchbyte3] [branchbyte4]
2. jsr\_w [branchbyte1] [branchbyte2] [branchbyte3] [branchbyte4]

有了这两个指令，索引超过65535的分支也是可用的。因此，Java方法的65535字节的限制就可以解除了。不过，由于Java class文件的更多的其他的限制，使得Java方法还是不能超过65535字节。

为了展示其他的限制，我会简单讲解一下class 文件的格式。

Java class文件的大致结构如下：

1. ClassFile {
2. u4 magic;
3. u2 minor\_version;
4. u2 major\_version;
5. u2 constant\_pool\_count;
6. cp\_info constant\_pool[constant\_pool\_count-1];
7. u2 access\_flags;
8. u2 this\_class;
9. u2 super\_class;
10. u2 interfaces\_count;
11. u2 interfaces[interfaces\_count];
12. u2 fields\_count;
13. field\_info fields[fields\_count];
14. u2 methods\_count;
15. method\_info methods[methods\_count];
16. u2 attributes\_count;
17. attribute\_info attributes[attributes\_count];}

之前反汇编的UserService.class文件反汇编的结果的前16个字节在十六进制编辑器中如下所示：

1. ca fe ba be 00 00 00 32 00 28 07 00 02 01 00 1b

通过这些数值，我们可以来看看class文件的格式。

1. magic：class文件最开始的四个字节是魔数。它的值是用来标识Java class文件的。从上面的内容里可以看出，魔数 的值是0xCAFEBABE。简而言之，只有一个文件的起始4字节是0xCAFEBABE的时候，它才会被当作Java class文件来处理。
2. minor\_version,major\_version:接下来的四个字节表示的是class文件的版本。UserService.class文件里的是0x00000032，所以这个class文件的版本是50.0。JDK 1.6编译的class文件的版本是50.0，JDK 1.5编译出来的class文件的版本是49.0。JVM必须对低版本的class文件保持后向兼容性，也就是低版本的class文件可以运行在高版本的JVM上。不过，反过来就不行了，当一个高版本的class文件运行在低版本的JVM上时，会出现java.lang.UnsupportedClassVersionError的错误。
3. constant\_pool\_count,constant\_pool[]:在版本号之后，存放的是类的常量池。这里保存的信息将会放入运行时常量池(Runtime Constant Pool)中去，这个后面会讲解的。在加载一个class文件的时候，JVM会把常量池里的信息存放在方法区的运行时常量区里。UserService.class文件里的constant\_pool\_count的值是0x0028，这表示常量池里有39(40-1)个常量。
4. access\_flags:这是表示一个类的描述符的标志；换句话说，它表示一个类是public,final还是abstract以及是不是接口的标志。
5. fields\_count,fields[]:当前类的成员变量的数量以及成员变量的信息。成员变量的信息包含变量名，类型，修饰符以及变量在constant\_pool里的索引。
6. methods\_count,methods[]:当前类的方法数量以及方法的信息。方法的信息包含方法名，参数的数量和类型，返回值的类型，修饰符，以及方法在constant\_pool里的索引，方法的可执行代码以及异常信息。
7. attributes\_count,attributes[]:attribution\_info结构包含不同种类的属性。field\_info和method\_info里都包含了attribute\_info结构。

javap简要地给出了class文件的一个可读形式。当你用"java -verbose"命令来分析UserService.class时，会输出如下的内容：

1. Compiled from "UserService.java"
3. public class com.atguigu.service.UserService extends java.lang.Object
4. SourceFile: "UserService.java"
5. minor version: 0
6. major version: 50
7. Constant pool:const #1 = class        #2;     //  com/atguigu/service/UserService
8. const #2 = Asciz        com/atguigu/service/UserService;
9. const #3 = class        #4;     //  java/lang/Object
10. const #4 = Asciz        java/lang/Object;
11. const #5 = Asciz        admin;
12. const #6 = Asciz        Lcom/atguigu/user/UserAdmin;;// … omitted - constant pool continued …
14. {
15. // … omitted - method information …
17. public void add(java.lang.String);
18. Code:
19. Stack=2, Locals=2, Args\_size=2
20. 0:   aload\_0
21. 1:   getfield        #15; //Field admin:Lcom/atguigu/user/UserAdmin;
22. 4:   aload\_1
23. 5:   invokevirtual   #23; //Method com/atguigu/user/UserAdmin.addUser:(Ljava/lang/String;)Lcom/nhn/user/User;
24. 8:   pop
25. 9:   return  LineNumberTable:
26. line 14: 0
27. line 15: 9  LocalVariableTable:
28. Start  Length  Slot  Name   Signature
29. 0      10      0    this       Lcom/atguigu/service/UserService;
30. 0      10      1    userName       Ljava/lang/String; // … Omitted - Other method information …
31. }

javap输出的内容太长，我这里只是提出了整个输出的一部分。整个的输出展示了constant\_pool里的不同信息，以及方法的内容。

关于方法的65565字节大小的限制是和method\_info struct相关的。method\_info结构包含Code,LineNumberTable,以及LocalViriable attribute几个属性，这个在“javap -verbose"的输出里可以看到。Code属性里的LineNumberTable，LocalVariableTable以及exception\_table的长度都是用一个固定的2字节来表示的。因此，方法的大小是不能超过LineNumberTable，LocalVariableTable以及exception\_table的长度的，它们都是65535字节。

许多人都在抱怨方法的大小限制，而且在JVM规范里还说名了”这个长度以后有可能会是可扩展的“。不过，到现在为止，还没有为这个限制做出任何动作。从JVM规范里的把class文件里的内容直接拷贝到方法区这个特点来看，要想在保持后向兼容性的同时来扩展方法区的大小是非常困难的。

如果因为Java编译器的错误而导致class文件的错误，会怎么样呢？或者，因为网络传输的错误导致拷贝的class文件的损坏呢？

为了预防这种场景，Java的类装载器通过一个严格而且慎密的过程来校验class文件。在JVM规范里详细地讲解了这方面的内容。

***注意***

我们怎样能够判断JVM正确地执行了class文件校验的所有过程呢？我们怎么来判断不同提供商的不同JVM实现是符合JVM规范的呢？为了能够验证以上两点，Oracle提供了一个测试工具TCK(Technology Compatibility Kit)。这个TCK工具通过执行成千上万的测试用例来验证一个JVM是否符合规范，这些测试里面包含了各种非法的class文件。只有通过了TCK的测试的JVM才能称作JVM。

和TCK相似，有一个组织JCP(Java Community Process;[http://jcp.org](http://jcp.org/" \t "_blank))负责Java规范以及新的Java技术规范。对于JCP而言，如果要完成一项Java规范请求(Java Specification Request, JSR)的话，需要具备规范文档，可参考的实现以及通过TCK测试。任何人如果想使用一项申请JSR的新技术的话，他要么使用RI提供许可的实现，要么自己实现一个并且保证通过TCK的测试。

#### JVM结构

Java编写的代码会按照下图的流程来执行

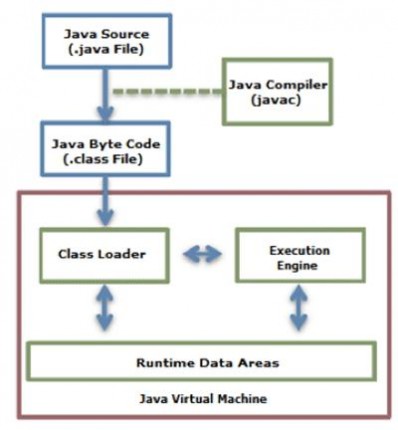


图 1:Java代码执行流程

类装载器装载负责装载编译后的字节码，并加载到运行时数据区（Runtime Data Area），然后执行引擎执行会执行这些字节码。

**类加载器（Class Loader）**

Java提供了动态的装载特性；它会在运行时的第一次引用到一个class的时候对它进行装载和链接，而不是在编译期进行。JVM的类装载器负责动态装载。Java类装载器有如下几个特点：

* 层级结构：Java里的类装载器被组织成了有父子关系的层级结构。Bootstrap类装载器是所有装载器的父亲。
* 代理模式：基于层级结构，类的装载可以在装载器之间进行代理。当装载器装载一个类时，首先会检查它是否在父装载器中进行装载了。如果上层的装载器已经装载了这个类，这个类会被直接使用。反之，类装载器会请求装载这个类。
* 可见性限制：一个子装载器可以查找父装载器中的类，但是一个父装载器不能查找子装载器里的类。
* 不允许卸载：类装载器可以装载一个类但是不可以卸载它，不过可以删除当前的类装载器，然后创建一个新的类装载器。

每个类装载器都有一个自己的命名空间用来保存已装载的类。当一个类装载器装载一个类时，它会通过保存在命名空间里的类全局限定名(Fully Qualified Class Name)进行搜索来检测这个类是否已经被加载了。如果两个类的全局限定名是一样的，但是如果命名空间不一样的话，那么它们还是不同的类。不同的命名空间表示class被不同的类装载器装载。

下图展示了类装载器的代理模型。

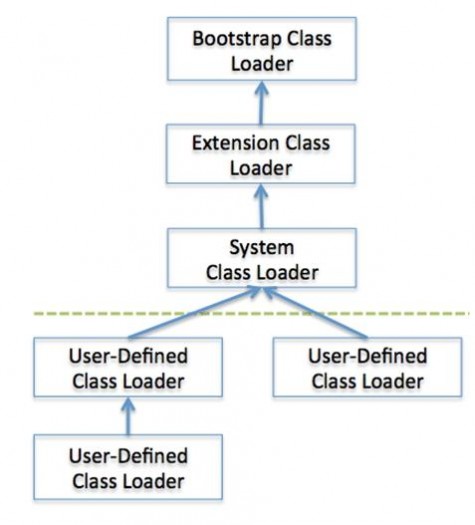


图 2: 类加载器代理模型

当一个类装载器（class loader）被请求装载类时，它首先按照顺序在上层装载器、父装载器以及自身的装载器的缓存里检查这个类是否已经存在。简单来说，就是在缓存里查看这个类是否已经被自己装载过了，如果没有的话，继续查找父类的缓存，直到在bootstrap类装载器里也没有找到的话，它就会自己在文件系统里去查找并且加载这个类。

* 启动类加载器（Bootstrap class loader）:这个类装载器是在JVM启动的时候创建的。它负责装载Java API，包含Object对象。和其他的类装载器不同的地方在于这个装载器是通过native code来实现的，而不是用Java代码。
* 扩展类加载器（Extension class loader）:它装载除了基本的Java API以外的扩展类。它也负责装载其他的安全扩展功能。
* 系统类加载器（System class loader）:如果说bootstrap class loader和extension class loader负责加载的是JVM的组件，那么system class loader负责加载的是应用程序类。它负责加载用户在$CLASSPATH里指定的类。
* 用户自定义类加载器（User-defined class loader）:这是应用程序开发者用直接用代码实现的类装载器。

类似于web应用服务(WAS)之类的框架会用这种结构来对Web应用和企业级应用进行分离。换句话来说，类装载器的代理模型可以用来保证不同应用之间的相互独立。WAS类装载器使用这种层级结构，不同的WAS供应商的装载器结构有稍许区别。

如果类装载器查找到一个没有装载的类，它会按照下图的流程来装载和链接这个类：

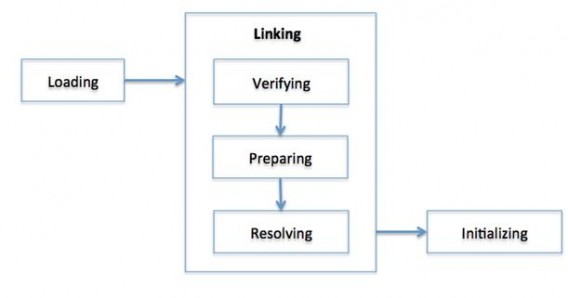


图 3: 类加载的各个阶段

每个阶段的描述如下：

* Loading: 类的信息从文件中获取并且载入到JVM的内存里。
* Verifying:检查读入的结构是否符合Java语言规范以及JVM规范的描述。这是类装载中最复杂的过程，并且花费的时间也是最长的。并且JVM TCK工具的大部分场景的用例也用来测试在装载错误的类的时候是否会出现错误。
* Preparing:分配一个结构用来存储类信息，这个结构中包含了类中定义的成员变量，方法和接口的信息。
* Resolving:把这个类的常量池中的所有的符号引用改变成直接引用。
* Initializing:把类中的变量初始化成合适的值。执行静态初始化程序，把静态变量初始化成指定的值。

JVM规范定义了上面的几个任务，不过它允许具体执行的时候能够有些灵活的变动。

**运行时数据区(Runtime Data Areas)**

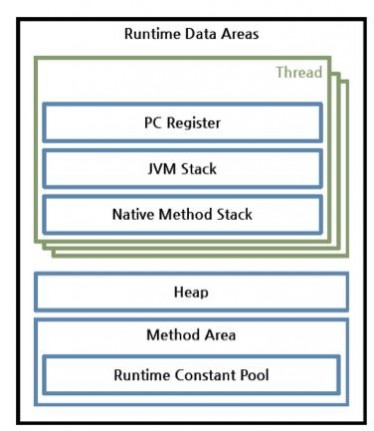


图 4: 运行时数据区

运行时数据区是在JVM运行的时候操作所分配的内存区。运行时内存区可以划分为6个区域。在这6个区域中，一个PC Register,JVM stack 以及Native Method Statck都是按照线程创建的，Heap,Method Area以及Runtime Constant Pool都是被所有线程公用的。

* PC寄存器(PC register):每个线程启动的时候，都会创建一个PC(Program Counter ,程序计数器)寄存器。PC寄存器里保存有当前正在执行的JVM指令的地址。
* JVM 堆栈(JVM stack)：每个线程启动的时候，都会创建一个JVM堆栈。它是用来保存栈帧的。JVM只会在JVM堆栈上对栈帧进行push和pop的操作。如果出现了异常，堆栈跟踪信息的每一行都代表一个栈帧立的信息，这些信息它是通过类似于printStackTrace()这样的方法来展示的。

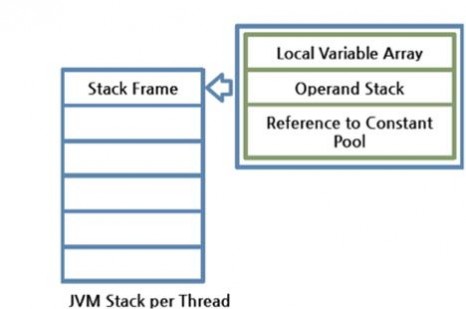


图 5: JVM堆栈

* 栈帧(stack frame)：每当一个方法在JVM上执行的时候，都会创建一个栈帧，并且会添加到当前线程的JVM堆栈上。当这个方法执行结束的时候，这个栈帧就会被移除。每个栈帧里都包含有当前正在执行的方法所属类的本地变量数组，操作数栈，以及运行时常量池的引用。本地变量数组的和操作数栈的大小都是在编译时确定的。因此，一个方法的栈帧的大小也是固定不变的。
* 局部变量数组(Local variable array)：这个数组的索引从0开始。索引为0的变量表示这个方法所属的类的实例。从1开始，首先存放的是传给该方法的参数，在参数后面保存的是方法的局部变量。
* 操作数栈(Operand stack)：方法实际运行的工作空间。每个方法都在操作数栈和局部变量数组之间交换数据，并且压入或者弹出其他方法返回的结果。操作数栈所需的最大空间是在编译期确定的。因此，操作数栈的大小也可以在编译期间确定。
* 本地方法栈(Native method stack)：供用非Java语言实现的本地方法的堆栈。换句话说，它是用来调用通过JNI(Java Native Interface Java本地接口）调用的C/C++代码。根据具体的语言，一个C堆栈或者C++堆栈会被创建。
* 方法区(Method area)：方法区是所有线程共享的，它是在JVM启动的时候创建的。它保存所有被JVM加载的类和接口的运行时常量池，成员变量以及方法的信息，静态变量以及方法的字节码。JVM的提供者可以通过不同的方式来实现方法区。在Oracle 的HotSpot JVM里，方法区被称为永久区或者永久代（PermGen）。是否对方法区进行垃圾回收对JVM的实现是可选的。
* 运行时常量池(Runtime constant pool)：这个区域和class文件里的constant\_pool是相对应的。这个区域是包含在方法区里的，不过，对于JVM的操作而言，它是一个核心的角色。因此在JVM规范里特别提到了它的重要性。除了包含每个类和接口的常量，它也包含了所有方法和变量的引用。简而言之，当一个方法或者变量被引用时，JVM通过运行时常量区来查找方法或者变量在内存里的实际地址。
* 堆(Heap)：用来保存实例或者对象的空间，而且它是垃圾回收的主要目标。当讨论类似于JVM性能之类的问题时，它经常会被提及。JVM提供者可以决定怎么来配置堆空间，以及不对它进行垃圾回收。

现在我们再会过头来看看之前反汇编的字节码

1. public void add(java.lang.String);
2. Code:
3. 0:   aload\_0
4. 1:   getfield        #15; //Field admin:Lcom/nhn/user/UserAdmin;
5. 4:   aload\_1
6. 5:   invokevirtual   #23; //Method com/nhn/user/UserAdmin.addUser:(Ljava/lang/String;)Lcom/nhn/user/User;
7. 8:   pop
8. 9:   return

把上面的反汇编代码和我们平时所见的x86架构的汇编代码相比较，我们会发现这两者的结构有点相似，都使用了操作码；不过，有一点不同的地方是Java字节码并不会在操作数里写入寄存器的名称、内存地址或者偏移量。之前已经说过，JVM用的是栈，它不会使用寄存器。和使用寄存器的x86架构不同，它自己负责内存的管理。它用索引例如15和23来代替实际的内存地址。15和23都是当前类（这里是UserService类）的常量池里的索引。简而言之，JVM为每个类创建了一个常量池，并且这个常量池里保存了实际目标的引用。

每行反汇编代码的解释如下：

* aload\_0:把局部变量数组中索引为#0的变量添加到操作数栈上。索引#0所表示的变量是this，即是当前实例的引用。
* getfield #15:把当前类的常量池里的索引为#15的变量添加到操作数栈。这里添加的是UserAdmin的admin成员变量。因为admin变量是个类的实例，因此添加的是一个引用。
* aload\_1:把局部变量数组里的索引为#1的变量添加到操作数栈。来自局部变量数组里的索引为1的变量是方法的一个参数。因此，在调用add()方法的时候，会把userName指向的String的引用添加到操作数栈上。
* invokevirtual #23:调用当前类的常量池里的索引为#23的方法。这个时候，通过getfile和aload\_1添加到操作数栈上的引用都被作为方法的参数。当方法运行完成并且返回时，它的返回值会被添加到操作数栈上。
* pop:把通过invokevirtual调用的方法的返回值从操作数栈里弹出来。你可以看到，在前面的例子里，用老的类库编译的那段代码是没有返回值的。简而言之，正因为之前的代码没有返回值，所以没必要吧把返回值从操作数栈上给弹出来。
* return：结束当前方法调用

下图可以帮助你更好地理解上面的内容。

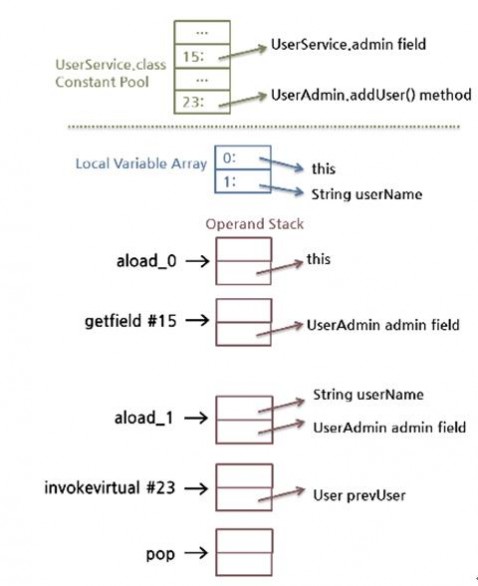


图 6: Java字节码装载到运行时数据区示例

顺便提一下，在这个方法里，局部变量数组没有被修改。所以上图只显示了操作数栈的变化。不过，大部分的情况下，局部变量数组也是会改变的。局部变量数组和操作数栈之间的数据传输是使用通过大量的load指令(aload,iload)和store指令（astore,istore)来实现的。

在这个图里，我们简单验证了运行时常量池和JVM栈的描述。当JVM运行的时候，每个类的实例都会在堆上进行分配，User，UserAdmin，UserService以及String等类的信息都会保存在方法区。

**执行引擎（Execution Engine）**通过类装载器装载的，被分配到JVM的运行时数据区的字节码会被执行引擎执行。执行引擎以指令为单位读取Java字节码。它就像一个CPU一样，一条一条地执行机器指令。每个字节码指令都由一个1字节的操作码和附加的操作数组成。执行引擎取得一个操作码，然后根据操作数来执行任务，完成后就继续执行下一条操作码。

不过Java字节码是用一种人类可以读懂的语言编写的，而不是用机器可以直接执行的语言。因此，执行引擎必须把字节码转换成可以直接被JVM执行的语言。字节码可以通过以下两种方式转换成合适的语言。

* 解释器：一条一条地读取，解释并且执行字节码指令。因为它一条一条地解释和执行指令，所以它可以很快地解释字节码，但是执行起来会比较慢。这是解释执行的语言的一个缺点。字节码这种“语言”基本来说是解释执行的。
* 即时（Just-In-Time)编译器：即时编译器被引入用来弥补解释器的缺点。执行引擎首先按照解释执行的方式来执行，然后在合适的时候，即时编译器把整段字节码编译成本地代码。然后，执行引擎就没有必要再去解释执行方法了，它可以直接通过本地代码去执行它。执行本地代码比一条一条进行解释执行的速度快很多。编译后的代码可以执行的很快，因为本地代码是保存在缓存里的。

不过，用JIT编译器来编译代码所花的时间要比用解释器去一条条解释执行花的时间要多。因此，如果代码只被执行一次的话，那么最好还是解释执行而不是编译后再执行。因此，内置了JIT编译器的JVM都会检查方法的执行频率，如果一个方法的执行频率超过一个特定的值的话，那么这个方法就会被编译成本地代码。

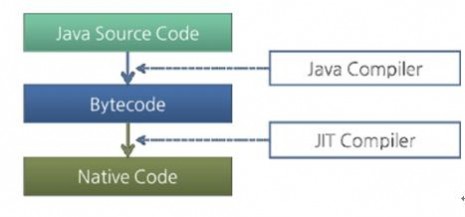


图 7:Java编译器和JIT编译器

JVM规范没有定义执行引擎该如何去执行。因此，JVM的提供者通过使用不同的技术以及不同类型的JIT编译器来提高执行引擎的效率。

大部分的JIT编译器都是按照下图的方式来执行的：

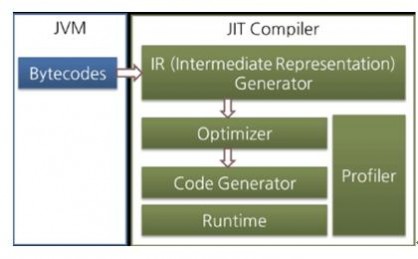


图 8: JIT编译器

JIT编译器把字节码转换成一个中间层表达式，一种中间层的表示方式，来进行优化，然后再把这种表示转换成本地代码。

Oracle Hotspot VM使用一种叫做热点编译器的JIT编译器。它之所以被称作”热点“是因为热点编译器通过分析找到最需要编译的“热点”代码，然后把热点代码编译成本地代码。如果已经被编译成本地代码的字节码不再被频繁调用了，换句话说，这个方法不再是热点了，那么Hotspot VM会把编译过的本地代码从cache里移除，并且重新按照解释的方式来执行它。Hotspot VM分为Server VM和Client VM两种，这两种VM使用不同的JIT编译器。

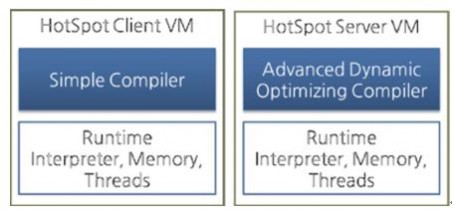


Figure 9: Hotspot Client VM and Server VM

Client VM 和Server VM使用完全相同的运行时，不过如上图所示，它们所使用的JIT编译器是不同的。Server VM用的是更高级的动态优化编译器，这个编译器使用了更加复杂并且更多种类的性能优化技术。

IBM 在IBM JDK 6里不仅引入了JIT编译器，它同时还引入了AOT(Ahead-Of-Time)编译器。它使得多个JVM可以通过共享缓存来共享编译过的本地代码。简而言之，通过AOT编译器编译过的代码可以直接被其他JVM使用。除此之外，IBM JVM通过使用AOT编译器来提前把代码编译器成JXE（Java EXecutable)文件格式来提供一种更加快速的执行方式。

大部分Java程序的性能都是通过提升执行引擎的性能来达到的。正如JIT编译器一样，很多优化的技术都被引入进来使得JVM的性能一直能够得到提升。最原始的JVM和最新的JVM最大的差别之处就是在于执行引擎。

Hotspot编译器在1.3版本的时候就被引入到Oracle Hotspot VM里了，JIT编译技术在Anroid 2.2版本的时候被引入到Dalvik VM里。

引入一种中间语言，例如字节码，虚拟机执行字节码，并且通过JIT编译器来提升JVM的性能的这种技术以及广泛应用在使用中间语言的编程语言上。例如微软的.Net，CLR（Common Language Runtime 公共语言运行时），也是一种VM，它执行一种被称作CIL（Common Intermediate Language）的字节码。CLR提供了AOT编译器和JIT编译器。因此，用C#或者VB.NET编写的源代码被编译后，编译器会生成CIL并且CIL会执行在有JIT编译器的CLR上。CLR和JVM相似，它也有垃圾回收机制，并且也是基于堆栈运行。

**Java 虚拟机规范，Java SE 第7版**2011年7月28日，Oracle发布了Java SE的第7个版本，并且把JVM规也更新到了相应的版本。在1999年发布《The Java Virtual Machine Specification,Second Edition》后，Oracle花了12年来发布这个更新的版本。这个更新的版本包含了这12年来累积的众多变化以及修改，并且更加细致地对规范进行了描述。此外，它还反映了《The Java Language Specificaion,Java SE 7 Edition》里的内容。主要的变化总结如下：

* 来自Java SE 5.0里的泛型，支持可变参数的方法
* 从Java SE 6以来，字节码校验的处理技术所发生的改变
* 添加invokedynamic指令以及class文件对于该指令的支持
* 删除了关于Java语言概念的内容，并且指引读者去参考Java语言规范
* 删除关于Java线程和锁的描述，并且把它们移到Java语言规范里

最大的改变是添加了invokedynamic指令。也就是说JVM的内部指令集做了修改，使得JVM开始支持动态类型的语言，这种语言的类型不是固定的，例如脚本语言以及来自Java SE 7里的Java语言。之前没有被用到的操作码186被分配给新指令invokedynamic，而且class文件格式里也添加了新的内容来支持invokedynamic指令。

Java SE 7的编译器生成的class文件的版本号是51.0。Java SE 6的是50.0。class文件的格式变动比较大，因此，51.0版本的class文件不能够在Java SE 6的虚拟机上执行。

尽管有了这么多的变动，但是Java方法的65535字节的限制还是没有被去掉。除非class文件的格式彻底改变，否者这个限制将来也是不可能去掉的。

值得说明的是，Oracle Java SE 7 VM支持G1这种新的垃圾回收机制，不过，它被限制在Oracle JVM上，因此，JVM本身对于垃圾回收的实现不做任何限制。也因此，在JVM规范里没有对它进行描述。

**switch语句里的String**Java SE 7里添加了很多新的语法和特性。不过，在Java SE 7的版本里，相对于语言本身而言，JVM没有多少的改变。那么，这些新的语言特性是怎么来实现的呢？我们通过反汇编的方式来看看switch语句里的String（把字符串作为switch()语句的比较对象）是怎么实现的？

例如，下面的代码：

1. // SwitchTest
2. public class SwitchTest {
3. public int doSwitch(String str) {
4. switch (str) {
5. case "abc":        return 1;
6. case "123":        return 2;
7. default:         return 0;
8. }
9. }
10. }

因为这是Java SE 7的一个新特性，所以它不能在Java SE 6或者更低版本的编译器上来编译。用Java SE 7的javac来编译。下面是通过javap -c来反编译后的结果。

1. C:Test>javap -c SwitchTest.classCompiled from "SwitchTest.java"
2. public class SwitchTest {
3. public SwitchTest();
4. Code:
5. 0: aload\_0
6. 1: invokespecial #1                  // Method java/lang/Object."<init>":()V
7. 4: return  public int doSwitch(java.lang.String);
8. Code:
9. 0: aload\_1
10. 1: astore\_2
11. 2: iconst\_m1
12. 3: istore\_3
13. 4: aload\_2
14. 5: invokevirtual #2                  // Method java/lang/String.hashCode:()I
15. 8: lookupswitch  { // 2
16. 48690: 50
17. 96354: 36
18. default: 61
19. }
20. 36: aload\_2
21. 37: ldc           #3                  // String abc
22. 39: invokevirtual #4                  // Method java/lang/String.equals:(Ljava/lang/Object;)Z
23. 42: ifeq          61
24. 45: iconst\_0
25. 46: istore\_3
26. 47: goto          61
27. 50: aload\_2
28. 51: ldc           #5                  // String 123
29. 53: invokevirtual #4                  // Method java/lang/String.equals:(Ljava/lang/Object;)Z
30. 56: ifeq          61
31. 59: iconst\_1
32. 60: istore\_3
33. 61: iload\_3
34. 62: lookupswitch  { // 2
35. 0: 88
36. 1: 90
37. default: 92
38. }
39. 88: iconst\_1
40. 89: ireturn
41. 90: iconst\_2
42. 91: ireturn
43. 92: iconst\_0
44. 93: ireturn

在#5和#8字节处，首先是调用了hashCode()方法，然后它作为参数调用了switch(int)。在lookupswitch的指令里，根据hashCode的结果进行不同的分支跳转。字符串“abc"的hashCode是96354，它会跳转到#36处。字符串”123“的hashCode是48690，它会跳转到#50处。生成的字节码的长度比Java源码长多了。首先，你可以看到字节码里用lookupswitch指令来实现switch()语句。不过，这里使用了两个lookupswitch指令，而不是一个。如果反编译的是针对Int的switch()语句的话，字节码里只会使用一个lookupswitch指令。也就是说，针对string的switch语句被分成用两个语句来实现。留心标号为#5，#39和#53的指令，来看看switch()语句是如何处理字符串的。

在第#36，#37，#39，以及#42字节的地方，你可以看见str参数被equals()方法来和字符串“abc”进行比较。如果比较的结果是相等的话，‘0’会被放入到局部变量数组的索引为#3的位置，然后跳抓转到第#61字节。

在第#50，#51，#53，以及#56字节的地方，你可以看见str参数被equals()方法来和字符串“123”进行比较。如果比较的结果是相等的话，10’会被放入到局部变量数组的索引为#3的位置，然后跳转到第#61字节。

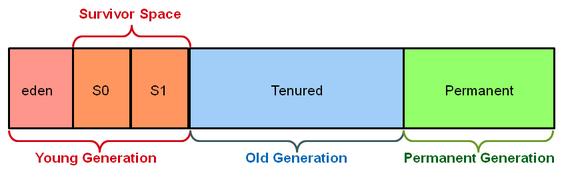
在第#61和#62字节的地方，局部变量数组里索引为#3的值，这里是'0'，‘1’或者其他的值，被lookupswitch用来进行搜索并进行相应的分支跳转。

换句话来说，在Java代码里的用来作为switch()的参数的字符串str变量是通过hashCode()和equals()方法来进行比较，然后根据比较的结果，来执行swtich()语句。

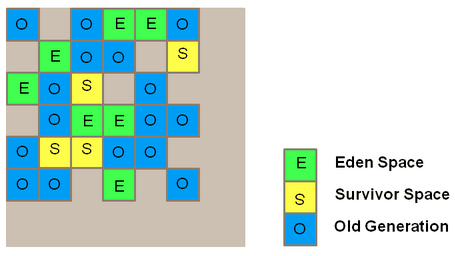
在这个结果里，编译后的字节码和之前版本的JVM规范没有不兼容的地方。Java SE 7的这个用字符串作为switch参数的特性是通过Java编译器来处理的，而不是通过JVM来支持的。通过这种方式还可以把其他的Java SE 7的新特性也通过Java编译器来实现。

### GC算法原理

在传统JVM内存管理中，我们把Heap空间分为Young/Old两个分区，Young分区又包括一个Eden和两个Survivor分区，如下图所示。新产生的对象首先会被存放在Eden区，而每次minor GC发生时，JVM一方面将Eden分区内存活的对象拷贝到一个空的Survivor分区，另一方面将另一个正在被使用的Survivor分区中的存活对象也拷贝到空的Survivor分区内。在此过程中，JVM始终保持一个Survivor分区处于全空的状态。一个对象在两个Survivor之间的拷贝到一定次数后，如果还是存活的，就将其拷入Old分区。当Old分区没有足够空间时，GC会停下所有程序线程，进行Full GC，即对Old区中的对象进行整理。这个所有线程都暂停的阶段被称为Stop-The-World(STW)，也是大多数GC算法中对性能影响最大的部分。



而G1 GC则完全改变了这一传统思路。它将整个Heap分为若干个预先设定的小区域块（如图2），每个区域块内部不再进行新旧分区， 而是将整个区域块标记为Eden/Survivor/Old。当创建新对象时，它首先被存放到某一个可用区块（Region）中。当该区块满了，JVM就会创建新的区块存放对象。当发生minor GC时，JVM将一个或几个区块中存活的对象拷贝到一个新的区块中，并在空余的空间中选择几个全新区块作为新的Eden分区。当所有区域中都有存活对象，找不到全空区块时，才发生Full GC。而在标记存活对象时，G1使用RememberSet的概念，将每个分区外指向分区内的引用记录在该分区的RememberSet中，避免了对整个Heap的扫描，使得各个分区的GC更加独立。在这样的背景下，我们可以看出G1 GC大大提高了触发Full GC时的Heap占用率，同时也使得Minor GC的暂停时间更加可控，对于内存较大的环境非常友好。这些颠覆性的改变，将给GC性能带来怎样的变化呢？最简单的方式，我们可以将老的GC设置直接迁移为G1 GC，然后观察性能变化。

   
图 2 G1 Heap结构示意

由于G1取消了对于heap空间不同新旧对象固定分区的概念，所以我们需要在GC配置选项上作相应的调整，使得应用能够合理地运行在G1 GC收集器上。一般来说，对于原运行在Parallel GC上的应用，需要去除的参数包括-Xmn, -XX:-UseAdaptiveSizePolicy, -XX:SurvivorRatio=n等；而对于原来使用CMS GC的应用，我们需要去掉-Xmn -XX:InitialSurvivorRatio -XX:SurvivorRatio -XX:InitialTenuringThreshold -XX:MaxTenuringThreshold等参数。另外在CMS中已经调优过的-XX:ParallelGCThreads -XX:ConcGCThreads参数最好也移除掉，因为对于CMS来说性能最好的不一定是对于G1性能最好的选择。我们先统一置为默认值，方便后期调优。此外，当应用开启的线程较多时，最好使用-XX:-ResizePLAB来关闭PLAB()的大小调整，以避免大量的线程通信所导致的性能下降。

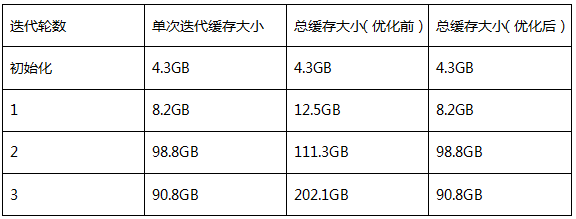
关于Hotspot JVM所支持的完整的GC参数列表，可以使用参数-XX:+PrintFlagsFinal打印出来，也可以参见Oracle官方的文档中对部分参数的解释。

### Spark的内存管理

Spark的核心概念是RDD，实际运行中内存消耗都与RDD密切相关。Spark允许用户将应用中重复使用的RDD数据持久化缓存起来，从而避免反复计算的开销，而RDD的持久化形态之一就是将全部或者部分数据缓存在JVM的Heap中。Spark Executor会将JVM的heap空间大致分为两个部分，一部分用来存放Spark应用中持久化到内存中的RDD数据，剩下的部分则用来作为JVM运行时的堆空间，负责RDD转化等过程中的内存消耗。我们可以通过spark.storage.memoryFraction参数调节这两块内存的比例，Spark会控制缓存RDD总大小不超过heap空间体积乘以这个参数所设置的值，而这块缓存RDD的空间中没有使用的部分也可以为JVM运行时所用。因此，分析Spark应用GC问题时应当分别分析两部分内存的使用情况。

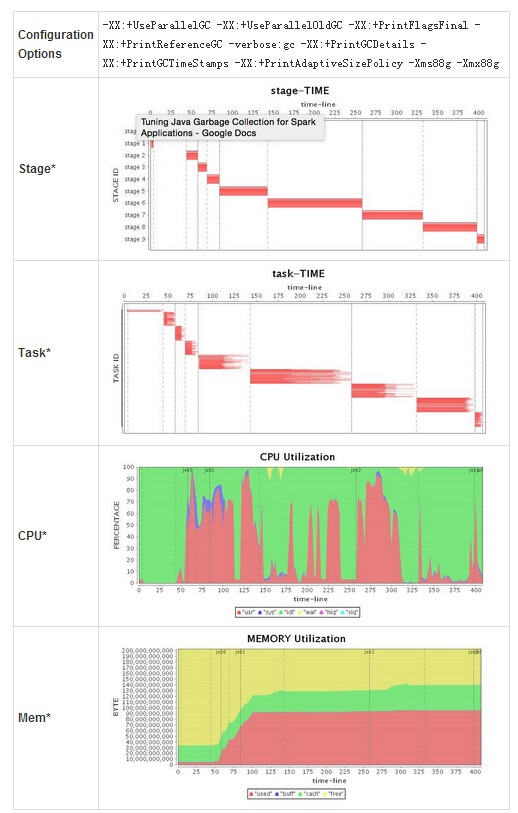
而当我们观察到GC延迟影响效率时，应当先检查Spark应用本身是否有效利用有限的内存空间。RDD占用的内存空间比较少的话，程序运行的heap空间也会比较宽松，GC效率也会相应提高；而RDD如果占用大量空间的话，则会带来巨大的性能损失。下面我们从一个用户案例展开：

该应用是利用Spark的组件Bagel来实现的，其本质就是一个简单的迭代计算。而每次迭代计算依赖于上一次的迭代结果，因此每次迭代结果都会被主动持续化到内存空间中。当运行用户程序时，我们观察到随着迭代次数的增加，进程占用的内存空间不断快速增长，GC问题越来越突出。但是，仔细分析Bagel实现机制，我们很快发现Bagel将每次迭代产生的RDD都持久化下来了，而没有及时释放掉不再使用的RDD，从而造成了内存空间不断增长，触发了更多GC执行。经过简单的修改，我们修复了这个问题（SPARK-2661）。应用的内存空间得到了有效的控制后，迭代次数三次以后RDD大小趋于稳定，缓存空间得到有效控制（如表1所示），GC效率得以大大提高，程序总的运行时间缩短了10%~20%。



小结：当观察到GC频繁或者延时长的情况，也可能是Spark进程或者应用中内存空间没有有效利用。所以可以尝试检查是否存在RDD持久化后未得到及时释放等情况。

### 选择垃圾收集器

在解决了应用本身的问题之后，我们就要开始针对Spark应用的GC调优了。基于修复了SPARK-2661的Spark版本，我们搭建了一个4个节点的集群，给每个Executor分配88G的Heap，在Spark的Standalone模式下来进行我们的实验。在使用默认的Parallel GC运行我们的Spark应用时，我们发现，由于Spark应用对于内存的开销比较大，而且大部分对象并不能在一个较短的生命周期中被回收，Parallel GC也常常受困于Full GC，而每次Full GC都给性能带来了较大的下降。而Parallel GC可以进行参数调优的空间也非常有限，我们只能通过调节一些基本参数来提高性能，如各年代分区大小比例、进入老年代前的拷贝次数等。而且这些调优策略只能推迟Full GC的到来，如果是长期运行的应用，Parallel GC调优的意义就非常有限了。因此，本文中不会再对Parallel GC进行调优。表2列出了Parallel GC的运行情况，其中CPU利用率较低的部分正是发生Full GC的时候。  
Parallel GC运行情况(未调优)

至于CMS GC，也没有办法消除这个Spark应用中的Full GC，而且CMS的Full GC的暂停时间远远超过了Parallel GC，大大拖累了该应用的吞吐量。

接下来，我们就使用最基本的G1 GC配置来运行我们的应用。实验结果发现，G1 GC竟然也出现了不可忍受的Full GC（表3的CPU利用率图中，可以明显发现Job 3中出现了将近100秒的暂停），超长的暂停时间大大拖累了整个应用的运行。如表4所示，虽然总的运行时间比Parallel GC略长，不过G1 GC表现略好于CMS GC。

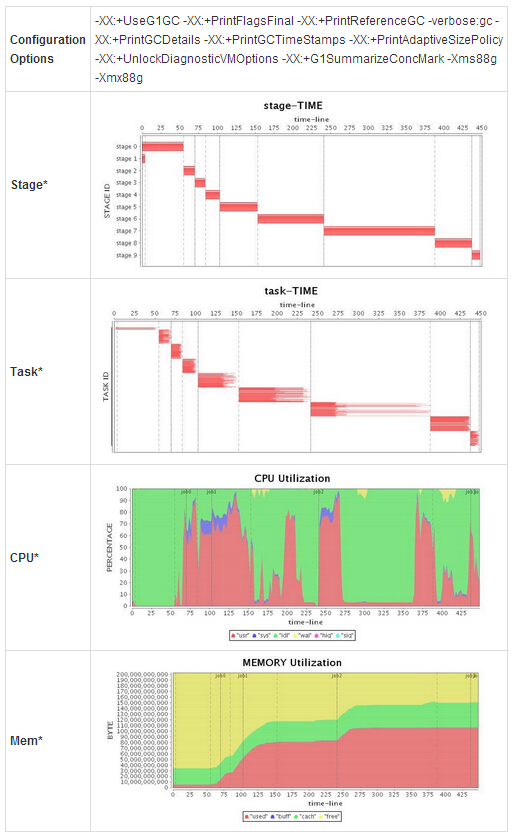
   
表 3 G1 GC运行情况(未调优)

  
表 4 三种垃圾收集器对应的程序运行时间比较（88GB heap未调优）

### 根据日志进一步调优

在让G1 GC跑起来之后，我们下一步就是需要根据GC log，来进一步进行性能调优。首先，我们要让JVM记录比较详细的GC日志. 对于Spark而言，我们需要在SPARK\_JAVA\_OPTS中设置参数使得Spark保留下我们需要用到的日志. 一般而言，我们需要设置这样一串参数：

|  |
| --- |
| -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark |

有了这些参数，我们就可以在SPARK的EXECUTOR日志中（默认输出到各worker节点的$SPARK\_HOME/work/$app\_id/$executor\_id/stdout中）读到详尽的GC日志以及生效的GC 参数了。接下来，我们就可以根据GC日志来分析问题，使程序获得更优性能。我们先来了解一下G1中一次GC的日志结构。

|  |
| --- |
| 251.354: [G1Ergonomics (Mixed GCs) continue mixed GCs, reason: candidate old regions available, candidate old regions: 363 regions, reclaimable: 9830652576 bytes (10.40 %), threshold: 10.00 %] [Parallel Time: 145.1 ms, GC Workers: 23] [GC Worker Start (ms): Min: 251176.0, Avg: 251176.4, Max: 251176.7, Diff: 0.7] [Ext Root Scanning (ms): Min: 0.8, Avg: 1.2, Max: 1.7, Diff: 0.9, Sum: 28.1] [Update RS (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.6, Sum: 5.8] [Processed Buffers: Min: 0, Avg: 1.6, Max: 9, Diff: 9, Sum: 37] [Scan RS (ms): Min: 6.0, Avg: 6.2, Max: 6.3, Diff: 0.3, Sum: 143.0] [Object Copy (ms): Min: 136.2, Avg: 136.3, Max: 136.4, Diff: 0.3, Sum: 3133.9] [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3] [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 1.9] [GC Worker Total (ms): Min: 143.7, Avg: 144.0, Max: 144.5, Diff: 0.8, Sum: 3313.0] [GC Worker End (ms): Min: 251320.4, Avg: 251320.5, Max: 251320.6, Diff: 0.2] [Code Root Fixup: 0.0 ms] [Clear CT: 6.6 ms] [Other: 26.8 ms] [Choose CSet: 0.2 ms] [Ref Proc: 16.6 ms] [Ref Enq: 0.9 ms] [Free CSet: 2.0 ms] [Eden: 3904.0M(3904.0M)->0.0B(4448.0M) Survivors: 576.0M->32.0M Heap: 63.7G(88.0G)->58.3G(88.0G)] [Times: user=3.43 sys=0.01, real=0.18 secs] |

以G1 GC的一次mixed GC为例，从这段日志中，我们可以看到G1 GC日志的层次是非常清晰的。日志列出了这次暂停发生的时间、原因，并分级各种线程所消耗的时长以及CPU时间的均值和最值。最后，G1 GC列出了本次暂停的清理结果，以及总共消耗的时间。

而在我们现在的G1 GC运行日志中，我们明显发现这样一段特殊的日志：

|  |
| --- |
| (to-space exhausted), 1.0552680 secs] [Parallel Time: 958.8 ms, GC Workers: 23] [GC Worker Start (ms): Min: 759925.0, Avg: 759925.1, Max: 759925.3, Diff: 0.3] [Ext Root Scanning (ms): Min: 1.1, Avg: 1.4, Max: 1.8, Diff: 0.6, Sum: 33.0] [SATB Filtering (ms): Min: 0.0, Avg: 0.0, Max: 0.3, Diff: 0.3, Sum: 0.3] [Update RS (ms): Min: 0.0, Avg: 1.2, Max: 2.1, Diff: 2.1, Sum: 26.9] [Processed Buffers: Min: 0, Avg: 2.8, Max: 11, Diff: 11, Sum: 65] [Scan RS (ms): Min: 1.6, Avg: 2.5, Max: 3.0, Diff: 1.4, Sum: 58.0] [Object Copy (ms): Min: 952.5, Avg: 953.0, Max: 954.3, Diff: 1.7, Sum: 21919.4] [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 2.2] [GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.6] [GC Worker Total (ms): Min: 958.1, Avg: 958.3, Max: 958.4, Diff: 0.3, Sum: 22040.4] [GC Worker End (ms): Min: 760883.4, Avg: 760883.4, Max: 760883.4, Diff: 0.0] [Code Root Fixup: 0.0 ms] [Clear CT: 0.4 ms] [Other: 96.0 ms] [Choose CSet: 0.0 ms] [Ref Proc: 0.4 ms] [Ref Enq: 0.0 ms] [Free CSet: 0.1 ms] [Eden: 160.0M(3904.0M)->0.0B(4480.0M) Survivors: 576.0M->0.0B Heap: 87.7G(88.0G)->87.7G(88.0G)] [Times: user=1.69 sys=0.24, real=1.05 secs] 760.981: [G1Ergonomics (Heap Sizing) attempt heap expansion, reason: allocation request failed, allocation request: 90128 bytes] 760.981: [G1Ergonomics (Heap Sizing) expand the heap, requested expansion amount: 33554432 bytes, attempted expansion amount: 33554432 bytes] 760.981: [G1Ergonomics (Heap Sizing) did not expand the heap, reason: heap expansion operation failed] 760.981: [Full GC 87G->36G(88G), 67.4381220 secs] |

显然最大的性能下降是这样的Full GC导致的，我们可以在日志中看到类似To-space Exhausted或者To-space Overflow这样的输出（取决于不同版本的JVM，输出略有不同）。这是G1 GC收集器在将某个需要垃圾回收的分区进行回收时，无法找到一个能将其中存活对象拷贝过去的空闲分区。这种情况被称为Evacuation Failure，常常会引发Full GC。而且很显然，G1 GC的Full GC效率相对于Parallel GC实在是相差太远，我们想要获得比Parallel GC更好的表现，一定要尽力规避Full GC的出现。对于这种情况，我们常见的处理办法有两种：

将InitiatingHeapOccupancyPercent参数调低（默认值是45），可以使G1 GC收集器更早开始Mixed GC；但另一方面，会增加GC发生频率。  
提高ConcGCThreads的值，在Mixed GC阶段投入更多的并发线程，争取提高每次暂停的效率。但是此参数会占用一定的有效工作线程资源。  
调试这两个参数可以有效降低Full GC出现的概率。Full GC被消除之后，最终的性能获得了大幅提升。但是我们发现，仍然有一些地方GC产生了大量的暂停时间。比如，我们在日志中读到很多类似这样的片断：

|  |
| --- |
| 280.008: [G1Ergonomics (Concurrent Cycles) request concurrent cycle initiation, reason: occupancy higher than threshold, occupancy: 62344134656 bytes, allocation request: 46137368 bytes, threshold: 42520176225 bytes (45.00 %), source: concurrent humongous allocation] |

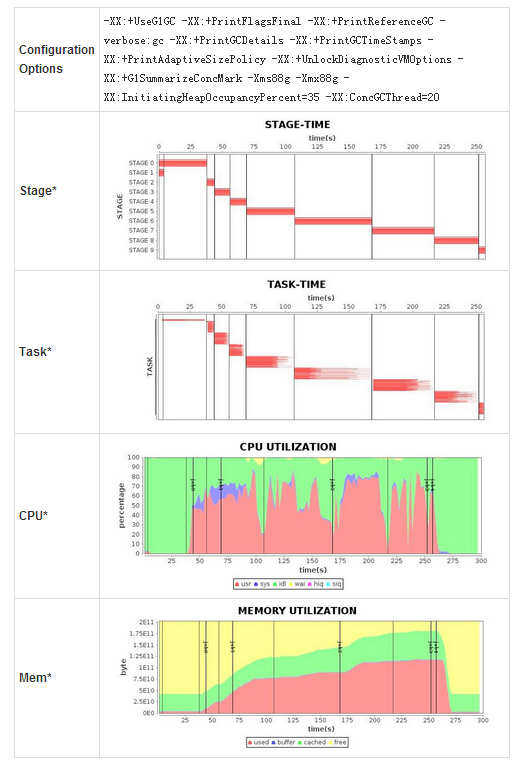
这里就是Humongous object，一些比G1的一个分区的一半更大的对象。对于这些对象，G1会专门在Heap上开出一个个Humongous Area来存放，每个分区只放一个对象。但是申请这么大的空间是比较耗时的，而且这些区域也仅当Full GC时才进行处理，所以我们要尽量减少这样的对象产生。或者提高G1HeapRegionSize的值减少HumongousArea的创建。不过在内存比较大的时，JVM默认把这个值设到了最大(32M)，此时我们只能通过分析程序本身找到这些对象并且尽量减少这样的对象产生。当然，相信随着G1 GC的发展，在后期的版本中相信这个最大值也会越来越大，毕竟G1号称是在1024～2048个Region时能够获得最佳性能。

接下来，我们可以分析一下单次cycle start到Mixed GC为止的时间间隔。如果这一时间过长，可以考虑进一步提升ConcGCThreads，需要注意的是，这会进一步占用一定CPU资源。

对于追求更短暂停时间的在线应用，如果观测到较长的Mixed GC pause，我们还要把G1RSetUpdatingPauseTimePercent调低，把G1ConcRefinementThreads调高。前文提到G1 GC通过为每个分区维护RememberSet来记录分区外对分区内的引用，G1RSetUpdatingPauseTimePercent则正是在STW阶段为G1收集器指定更新RememberSet的时间占总STW时间的期望比例，默认为10。而G1ConcRefinementThreads则是在程序运行时维护RememberSet的线程数目。通过对这两个值的对应调整，我们可以把STW阶段的RememberSet更新工作压力更多地移到Concurrent阶段。

另外，对于需要长时间运行的应用，我们不妨加上AlwaysPreTouch参数，这样JVM会在启动时就向OS申请所有需要使用的内存，避免动态申请，也可以提高运行时性能。但是该参数也会大大延长启动时间。

最终，经过几轮GC参数调试，其结果如下表5所示。较之先前的结果，我们最终还是获得了较满意的运行效率。

   
表 5 使用G1 GC调优完成后的表现

小结：综合考虑G1 GC是较为推崇的默认Spark GC机制。进一步的GC日志分析，可以收获更多的GC优化。经过上面的调优过程，我们将该应用的运行时间缩短到了4.3分钟，相比调优之前，我们获得了1.7倍左右的性能提升，而相比Parallel GC也获得了1.5倍左右的性能提升。

对于大量依赖于内存计算的Spark应用，GC调优显得尤为重要。在发现GC问题的时候，不要着急调试GC。而是先考虑是否存在Spark进程内存管理的效率问题，例如RDD缓存的持久化和释放。至于GC参数的调试，首先我们比较推荐使用G1 GC来运行Spark应用。相较于传统的垃圾收集器，随着G1的不断成熟，需要配置的选项会更少，能同时满足高吞吐量和低延迟的寻求。当然，GC的调优不是绝对的，不同的应用会有不同应用的特性，掌握根据GC日志进行调优的方法，才能以不变应万变。最后，也不能忘了先对程序本身的逻辑和代码编写进行考量，例如减少中间变量的创建或者复制，控制大对象的创建，将长期存活对象放在Off-heap中等等。