## Spark Core调优

### 分配更多资源

分配更多资源：性能调优的王道，就是增加和分配更多的资源，性能和速度上的提升，是显而易见的；基本上，在一定范围之内，增加资源与性能的提升，是成正比的；写完了一个复杂的spark作业之后，进行性能调优的时候，首先第一步，我觉得，就是要来调节最优的资源配置；在这个基础之上，如果说你的spark作业，能够分配的资源达到了你的能力范围的顶端之后，无法再分配更多的资源了，公司资源有限；那么才是考虑去做后面的这些性能调优的点。

#### 分配哪些资源

executor、cpu per executor、memory per executor、driver memory

#### 在哪分配资源

在我们在生产环境中，提交spark作业时，用的spark-submit shell脚本，里面调整对应的参数

/usr/local/spark/bin/spark-submit \

--class cn.spark.sparktest.core.WordCountCluster \

--num-executors 3 \ 配置executor的数量

--driver-memory 100m \ 配置driver的内存（影响不大）

--executor-memory 100m \ 配置每个executor的内存大小

--executor-cores 3 \ 配置每个executor的cpu core数量

/usr/local/SparkTest-0.0.1-SNAPSHOT-jar-with-dependencies.jar \

#### 怎么分配最大的资源

第一种，Spark Standalone，公司集群上，搭建了一套Spark集群，你心里应该清楚每台机器还能够给你使用的，大概有多少内存，多少cpu core；那么，设置的时候，就根据这个实际的情况，去调节每个spark作业的资源分配。比如说你的每台机器能够给你使用4G内存，2个cpu core；20台机器；executor，20；4G内存，2个cpu core，平均每个executor。

第二种，Yarn。资源队列。资源调度。应该去查看，你的spark作业，要提交到的资源队列，大概有多少资源？500G内存，100个cpu core；executor，50；10G内存，2个cpu core，平均每个executor。

一个原则，你能使用的资源有多大，就尽量去调节到最大的大小（executor的数量，几十个到上百个不等；executor内存；executor cpu core）

#### 分配了这些资源以后，性能为什么会得到提升

##### 增加executor

如果executor数量比较少，那么，能够并行执行的task数量就比较少，就意味着，我们的Application的并行执行的能力就很弱。比如有3个executor，每个executor有2个cpu core，那么同时能够并行执行的task，就是6个。6个执行完以后，再换下一批6个task。

增加了executor数量以后，那么，就意味着，能够并行执行的task数量，也就变多了。比如原先是6个，现在可能可以并行执行10个，甚至20个，100个。那么并行能力就比之前提升了数倍，数十倍。相应的，性能（执行的速度），也能提升数倍~数十倍。

##### 增加每个executor的cpu core

增加每个executor的cpu core，也是增加了执行的并行能力。原本20个executor，每个才2个cpu core。能够并行执行的task数量，就是40个task。

现在每个executor的cpu core，增加到了5个。能够并行执行的task数量，就是100个task。

执行的速度，提升了2.5倍。

##### 增加每个executor的内存

增加每个executor的内存量。增加了内存量以后，对性能的提升，有三点：

1、如果需要对RDD进行cache，那么更多的内存，就可以缓存更多的数据，将更少的数据写入磁盘，甚至不写入磁盘。减少了磁盘IO。

2、对于shuffle操作，reduce端，会需要内存来存放拉取的数据并进行聚合。如果内存不够，也会写入磁盘。如果给executor分配更多内存以后，就有更少的数据，需要写入磁盘，甚至不需要写入磁盘。减少了磁盘IO，提升了性能。

3、对于task的执行，可能会创建很多对象。如果内存比较小，可能会频繁导致JVM堆内存满了，然后频繁GC，垃圾回收，minor GC和full GC。（速度很慢）。内存加大以后，带来更少的GC，垃圾回收，避免了速度变慢，速度变快了。

### 调节Spark并行度

并行度：其实就是指的是，Spark作业中，各个stage的task数量，也就代表了Spark作业的在各个阶段（stage）的并行度。

#### 原因

如果不调节并行度，导致并行度过低，会怎么样？

假设，现在已经在spark-submit脚本里面，给我们的spark作业分配了足够多的资源，比如50个executor，每个executor有10G内存，每个executor有3个cpu core。基本已经达到了集群或者yarn队列的资源上限。

task没有设置，或者设置的很少，比如就设置了，100个task。50个executor，每个executor有3个cpu core，也就是说，你的Application任何一个stage运行的时候，都有总数在150个cpu core，可以并行运行。但是你现在，只有100个task，平均分配一下，每个executor分配到2个task，ok，那么同时在运行的task，只有100个，每个executor只会并行运行2个task。每个executor剩下的一个cpu core，就浪费掉了。

你的资源虽然分配足够了，但是问题是，并行度没有与资源相匹配，导致你分配下去的资源都浪费掉了。

#### 设置多少并行度

合理的并行度的设置，应该是要设置的足够大，大到可以完全合理的利用你的集群资源；比如上面的例子，总共集群有150个cpu core，可以并行运行150个task。那么就应该将你的Application的并行度，至少设置成150，才能完全有效的利用你的集群资源，让150个task，并行执行；而且task增加到150个以后，即可以同时并行运行，还可以让每个task要处理的数据量变少；比如总共150G的数据要处理，如果是100个task，每个task计算1.5G的数据；现在增加到150个task，可以并行运行，而且每个task主要处理1G的数据就可以。

很简单的道理，只要合理设置并行度，就可以完全充分利用你的集群计算资源，并且减少每个task要处理的数据量，最终，就是提升你的整个Spark作业的性能和运行速度。

1、task数量，至少设置成与Spark application的总cpu core数量相同（最理想情况，比如总共150个cpu core，分配了150个task，一起运行，差不多同一时间运行完毕）

2、官方是推荐，task数量，设置成spark application总cpu core数量的2~3倍，比如150个cpu core，基本要设置task数量为300~500；

实际情况，与理想情况不同的，有些task会运行的快一点，比如50s就完了，有些task，可能会慢一点，要1分半才运行完，所以如果你的task数量，刚好设置的跟cpu core数量相同，可能还是会导致资源的浪费，因为，比如150个task，10个先运行完了，剩余140个还在运行，但是这个时候，有10个cpu core就空闲出来了，就导致了浪费。那如果task数量设置成cpu core总数的2~3倍，那么一个task运行完了以后，另一个task马上可以补上来，就尽量让cpu core不要空闲，同时也是尽量提升spark作业运行的效率和速度，提升性能。

#### 如何设置一个Spark Application的并行度

spark.default.parallelism

SparkConf conf = new SparkConf()

.set("spark.default.parallelism", "500")

### 重构RDD架构与RDD持久化

#### 重构RDD架构与RDD持久化原理

第一，RDD架构重构与优化

尽量去复用RDD，差不多的RDD，可以抽取称为一个共同的RDD，供后面的RDD计算时，反复使用。

第二，公共RDD一定要实现持久化

对于要多次计算和使用的公共RDD，一定要进行持久化。

持久化，也就是说，将RDD的数据缓存到内存中/磁盘中，（BlockManager），以后无论对这个RDD做多少次计算，那么都是直接取这个RDD的持久化的数据，比如从内存中或者磁盘中，直接提取一份数据。

第三，持久化，是可以进行序列化的

如果正常将数据持久化在内存中，那么可能会导致内存的占用过大，这样的话，也许，会导致OOM内存溢出。

当纯内存无法支撑公共RDD数据完全存放的时候，就优先考虑，使用序列化的方式在纯内存中存储。将RDD的每个partition的数据，序列化成一个大的字节数组，就一个对象；序列化后，大大减少内存的空间占用。

序列化的方式，唯一的缺点就是，在获取数据的时候，需要反序列化。

如果序列化纯内存方式，还是导致OOM，内存溢出；就只能考虑磁盘的方式，内存+磁盘的普通方式（无序列化）。

内存+磁盘，序列化

第四，为了数据的高可靠性，而且内存充足，可以使用双副本机制，进行持久化

持久化的双副本机制，持久化后的一个副本，因为机器宕机了，副本丢了，就还是得重新计算一次；持久化的每个数据单元，存储一份副本，放在其他节点上面；从而进行容错；一个副本丢了，不用重新计算，还可以使用另外一份副本。

这种方式，仅仅针对你的内存资源极度充足

#### 如何设置RDD持久化

持久化，很简单，就是对RDD调用persist()方法，并传入一个持久化级别

如果是persist(StorageLevel.MEMORY\_ONLY())，纯内存，无序列化，那么就可以用cache()方法来替代

StorageLevel.MEMORY\_ONLY\_SER()，第二选择

StorageLevel.MEMORY\_AND\_DISK()，第三选择

StorageLevel.MEMORY\_AND\_DISK\_SER()，第四选择

StorageLevel.DISK\_ONLY()，第五选择

如果内存充足，要使用双副本高可靠机制

选择后缀带\_2的策略

StorageLevel.MEMORY\_ONLY\_2()

### 广播大变量

#### 原因

task执行的算子中，使用了外部的变量,例如一个map变量，每个task都会获取一份变量的副本，有什么缺点呢？在什么情况下，会出现性能上的恶劣的影响呢？

比如，map是1M。总共，你前面调优都调的特好，资源给的到位，配合着资源，并行度调节的绝对到位，1000个task。大量task的确都在并行运行。

这些task里面都用到了占用1M内存的map，那么首先，map会拷贝1000份副本，通过网络传输到各个task中去，给task使用。总计有1G的数据，会通过网络传输。网络传输的开销，不容乐观啊！！！网络传输，也许就会消耗掉你的spark作业运行的总时间的一小部分。

map副本，传输到了各个task上之后，是要占用内存的。1个map的确不大，1M；1000个map分布在你的集群中，一下子就耗费掉1G的内存。对性能会有什么影响呢？

不必要的内存的消耗和占用，就导致了，你在进行RDD持久化到内存，也许就没法完全在内存中放下；就只能写入磁盘，最后导致后续的操作在磁盘IO上消耗性能；

你的task在创建对象的时候，也许会发现堆内存放不下所有对象，也许就会导致频繁的垃圾回收器的回收，GC。GC的时候，一定是会导致工作线程停止，也就是导致Spark暂停工作那么一点时间。频繁GC的话，对Spark作业的运行的速度会有相当可观的影响。

#### 原理

广播变量，初始的时候，就在Drvier上有一份副本。task在运行的时候，想要使用广播变量中的数据，此时首先会在自己本地的Executor对应的BlockManager中，尝试获取变量副本；如果本地没有，那么就从Driver远程拉取变量副本，并保存在本地的BlockManager中；此后这个executor上的task，都会直接使用本地的BlockManager中的副本。

executor的BlockManager除了从driver上拉取，也可能从其他节点的BlockManager上拉取变量副本，举例越近越好。

#### 操作

声明:

final Broadcast<List<Tuple2<Long, Row>>> userInfosBroadcast = sc.broadcast(userInfos);

使用:

List<Tuple2<Long, Row>> userInfos = userInfosBroadcast.value();

### 使用Kryo序列化

#### 原理

默认情况下，Spark内部是使用Java的序列化机制，ObjectOutputStream / ObjectInputStream，对象输入输出流机制，来进行序列化

这种默认序列化机制的好处在于，处理起来比较方便；也不需要我们手动去做什么事情，只是，你在算子里面使用的变量，必须是实现Serializable接口的，可序列化即可。

但是缺点在于，默认的序列化机制的效率不高，序列化的速度比较慢；序列化以后的数据，占用的内存空间相对还是比较大。

可以手动进行序列化格式的优化

Spark支持使用Kryo序列化机制。Kryo序列化机制，比默认的Java序列化机制，速度要快，序列化后的数据要更小，大概是Java序列化机制的1/10。

当使用了序列化的持久化级别时，在将每个RDD partition序列化成一个大的字节数组时，就会使用Kryo进一步优化序列化的效率和性能

所以Kryo序列化优化以后，可以让网络传输的数据变少；在集群中耗费的内存资源大大减少。

#### 作用的地方

Kryo序列化机制，一旦启用以后，会生效的几个地方：

1、算子函数中使用到的外部变量

2、持久化RDD时进行序列化，StorageLevel.MEMORY\_ONLY\_SER

3、shuffle

1、算子函数中使用到的外部变量，使用Kryo以后：优化网络传输的性能，可以优化集群中内存的占用和消耗

2、持久化RDD，优化内存的占用和消耗；持久化RDD占用的内存越少，task执行的时候，创建的对象，就不至于频繁的占满内存，频繁发生GC。

3、shuffle：可以优化网络传输的性能

#### 操作

首先第一步，在SparkConf中设置一个属性，spark.serializer，org.apache.spark.serializer.KryoSerializer类；

Kryo之所以没有被作为默认的序列化类库的原因，就要出现了：主要是因为Kryo要求，如果要达到它的最佳性能的话，那么就一定要注册你自定义的类（比如，你的算子函数中使用到了外部自定义类型的对象变量，这时，就要求必须注册你的类，否则Kryo达不到最佳性能）。

第二步，注册你使用到的，需要通过Kryo序列化的，一些自定义类，SparkConf.registerKryoClasses()

项目中的使用：

.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

.registerKryoClasses(new Class[]{CategorySortKey.class})

public class CategorySortKey implements Ordered<CategorySortKey>, Serializable {…….}

### 使用fastutils优化数据格式

#### fastutil介绍

fastutil是扩展了Java标准集合框架（Map、List、Set；HashMap、ArrayList、HashSet）的类库，提供了特殊类型的map、set、list和queue；

fastutil能够提供更小的内存占用，更快的存取速度；我们使用fastutil提供的集合类，来替代自己平时使用的JDK的原生的Map、List、Set，好处在于，fastutil集合类，可以减小内存的占用，并且在进行集合的遍历、根据索引（或者key）获取元素的值和设置元素的值的时候，提供更快的存取速度；

fastutil也提供了64位的array、set和list，以及高性能快速的，以及实用的IO类，来处理二进制和文本类型的文件；

fastutil最新版本要求Java 7以及以上版本；

fastutil的每一种集合类型，都实现了对应的Java中的标准接口（比如fastutil的map，实现了Java的Map接口），因此可以直接放入已有系统的任何代码中。

fastutil还提供了一些JDK标准类库中没有的额外功能（比如双向迭代器）。

fastutil除了对象和原始类型为元素的集合，fastutil也提供引用类型的支持，但是对引用类型是使用等于号（=）进行比较的，而不是equals()方法。

fastutil尽量提供了在任何场景下都是速度最快的集合类库。

#### Spark中应用fastutil的场景

1、如果算子函数使用了外部变量；那么第一，你可以使用Broadcast广播变量优化；第二，可以使用Kryo序列化类库，提升序列化性能和效率；第三，如果外部变量是某种比较大的集合，那么可以考虑使用fastutil改写外部变量，首先从源头上就减少内存的占用，通过广播变量进一步减少内存占用，再通过Kryo序列化类库进一步减少内存占用。

2、在你的算子函数里，也就是task要执行的计算逻辑里面，如果有逻辑中，出现，要创建比较大的Map、List等集合，可能会占用较大的内存空间，而且可能涉及到消耗性能的遍历、存取等集合操作；那么此时，可以考虑将这些集合类型使用fastutil类库重写，使用了fastutil集合类以后，就可以在一定程度上，减少task创建出来的集合类型的内存占用。避免executor内存频繁占满，频繁唤起GC，导致性能下降。

#### 关于fastutil调优的说明

fastutil其实没有你想象中的那么强大，也不会跟官网上说的效果那么一鸣惊人。广播变量、Kryo序列化类库、fastutil，都是之前所说的，对于性能来说，类似于一种调味品，烤鸡，本来就很好吃了，然后加了一点特质的孜然麻辣粉调料，就更加好吃了一点。分配资源、并行度、RDD架构与持久化，这三个就是烤鸡；broadcast、kryo、fastutil，类似于调料。

比如说，你的spark作业，经过之前一些调优以后，大概30分钟运行完，现在加上broadcast、kryo、fastutil，也许就是优化到29分钟运行完、或者更好一点，也许就是28分钟、25分钟。

shuffle调优，15分钟；groupByKey用reduceByKey改写，执行本地聚合，也许10分钟；跟公司申请更多的资源，比如资源更大的YARN队列，1分钟。

#### fastutil的使用

第一步：在pom.xml中引用fastutil的包

<dependency>

<groupId>fastutil</groupId>

<artifactId>fastutil</artifactId>

<version>5.0.9</version>

</dependency>

速度比较慢，可能是从国外的网去拉取jar包，可能要等待5分钟，甚至几十分钟，不等

List<Integer> => IntList

基本都是类似于IntList的格式，前缀就是集合的元素类型；特殊的就是Map，Int2IntMap，代表了key-value映射的元素类型。除此之外，刚才也看到了，还支持object、reference。

项目中的使用：

|  |
| --- |
| 把Map<String,Map<String,List<Integer>>> dateHourExtractMap =  new HashMap<String, Map<String,List<Integer>>>();  变成fastutilDateHourExtractMap  /\*\*  \* fastutil的使用，比如List<Integer>的List，对应到fastutil，就是IntList  \*/  Map<String, Map<String,IntList>> fastutilDateHourExtractMap =  new HashMap<String, Map<String,IntList>>();  for(Map.Entry<String, Map<String,List<Integer>>> dateHourExtractEntry : dateHourExtractMap.entrySet()) {  String date = dateHourExtractEntry.getKey();  Map<String,List<Integer>> hourExtractMap = dateHourExtractEntry.getValue();  Map<String,IntList> fastutilHourExtractMap = new HashMap<String, IntList>();    for(Map.Entry<String, List<Integer>> hourExtractEntry : hourExtractMap.entrySet()) {  String hour = hourExtractEntry.getKey();  List<Integer> extractList = hourExtractEntry.getValue();  IntList fastutilExtractList = new IntArrayList();    for(int i = 0; i < extractList.size(); i++) {  fastutilExtractList.add(extractList.get(i));  }    fastutilHourExtractMap.put(hour, fastutilExtractList);  }  fastutilDateHourExtractMap.put(date, fastutilHourExtractMap);  } |

### 调节数据本地化等待时间

#### 原理

Spark在Driver上，对Application的每一个stage的task，进行分配之前，都会计算出每个task要计算的是哪个分片数据，RDD的某个partition；Spark的task分配算法，优先，会希望每个task正好分配到它要计算的数据所在的节点，这样的话，就不用在网络间传输数据；

但是呢，通常来说，有时，事与愿违，可能task没有机会分配到它的数据所在的节点，为什么呢，可能那个节点的计算资源和计算能力都满了；所以呢，这种时候，通常来说，Spark会等待一段时间，默认情况下是3s钟（不是绝对的，还有很多种情况，对不同的本地化级别，都会去等待），到最后，实在是等待不了了，就会选择一个比较差的本地化级别，比如说，将task分配到靠它要计算的数据所在节点，比较近的一个节点，然后进行计算。

但是对于第二种情况，通常来说，肯定是要发生数据传输，task会通过其所在节点的BlockManager来获取数据，BlockManager发现自己本地没有数据，会通过一个getRemote()方法，通过TransferService（网络数据传输组件）从数据所在节点的BlockManager中，获取数据，通过网络传输回task所在节点。

对于我们来说，当然不希望是类似于第二种情况的了。最好的，当然是task和数据在一个节点上，直接从本地executor的BlockManager中获取数据，纯内存，或者带一点磁盘IO；如果要通过网络传输数据的话，那么实在是，性能肯定会下降的，大量网络传输，以及磁盘IO，都是性能的杀手。

#### 本地化级别

PROCESS\_LOCAL：进程本地化，代码和数据在同一个进程中，也就是在同一个executor中；计算数据的task由executor执行，数据在executor的BlockManager中；性能最好

NODE\_LOCAL：节点本地化，代码和数据在同一个节点中；比如说，数据作为一个HDFS block块，就在节点上，而task在节点上某个executor中运行；或者是，数据和task在一个节点上的不同executor中；数据需要在进程间进行传输

NO\_PREF：对于task来说，数据从哪里获取都一样，没有好坏之分

RACK\_LOCAL：机架本地化，数据和task在一个机架的两个节点上；数据需要通过网络在节点之间进行传输

ANY：数据和task可能在集群中的任何地方，而且不在一个机架中，性能最差

spark.locality.wait，默认是3s

#### 调节该参数的时机

观察日志，spark作业的运行日志，推荐大家在测试的时候，先用client模式，在本地就直接可以看到比较全的日志。日志里面会显示，starting task。。。，PROCESS LOCAL、NODE LOCAL

，观察大部分task的数据本地化级别，如果大多都是PROCESS\_LOCAL，那就不用调节了

如果是发现，好多的级别都是NODE\_LOCAL、ANY，那么最好就去调节一下数据本地化的等待时长

调节完，应该是要反复调节，每次调节完以后，再来运行，观察日志

看看大部分的task的本地化级别有没有提升；看看，整个spark作业的运行时间有没有缩短

你别本末倒置，本地化级别倒是提升了，但是因为大量的等待时长，spark作业的运行时间反而增加了，那就还是不要调节了

#### 怎么调节

spark.locality.wait，默认是3s；6s，10s

默认情况下，下面3个的等待时长，都是跟上面那个是一样的，都是3s

spark.locality.wait.process

spark.locality.wait.node

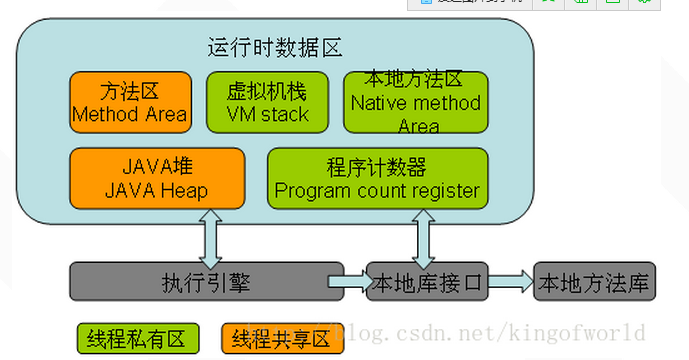
spark.locality.wait.rack

new SparkConf()

.set("spark.locality.wait", "10")

### JVM调优

#### JVM运行原理总述



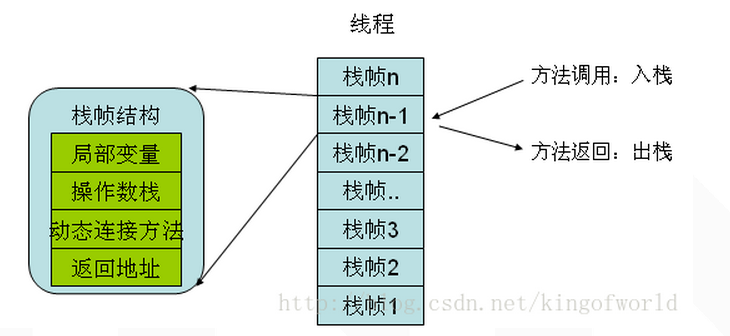
##### 程序计数器

多线程时，当线程数超过CPU数量或CPU内核数量，线程之间就要根据时间片轮询抢夺CPU时间资源。因此每个线程有要有一个独立的程序计数器，记录下一条要运行的指令。线程私有的内存区域。如果执行的是JAVA方法，计数器记录正在执行的java字节码地址，如果执行的是native方法，则计数器为空。

##### 虚拟机栈

线程私有的，与线程在同一时间创建。管理JAVA方法执行的内存模型。每个方法执行时都会创建一个桢栈来存储方法的的变量表、操作数栈、动态链接方法、返回值、返回地址等信息。栈的大小决定了方法调用的可达深度（递归多少层次，或嵌套调用多少层其他方法，-Xss参数可以设置虚拟机栈大小）。栈的大小可以是固定的，或者是动态扩展的。如果请求的栈深度大于最大可用深度，则抛出stackOverflowError；如果栈是可动态扩展的，但没有内存空间支持扩展，则抛出OutofMemoryError。

使用jclasslib工具可以查看class类文件的结构。下图为栈帧结构图：



##### 本地方法区

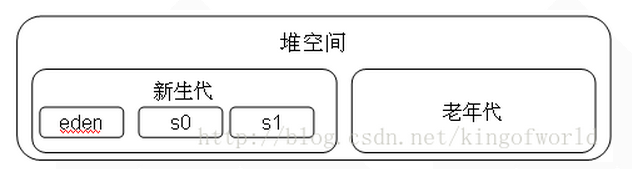
和虚拟机栈功能相似，但管理的不是JAVA方法，是本地方法，本地方法是用C实现的。

##### JAVA堆

线程共享的，存放所有对象实例和数组。垃圾回收的主要区域。可以分为新生代和老年代(tenured)。

新生代用于存放刚创建的对象以及年轻的对象，如果对象一直没有被回收，生存得足够长，老年对象就会被移入老年代。

新生代又可进一步细分为eden、survivorSpace0(s0,from space)、survivorSpace1(s1,to space)。刚创建的对象都放入eden,s0和s1都至少经过一次GC并幸存。如果幸存对象经过一定时间仍存在，则进入老年代(tenured)。



##### 方法区（永久代）

线程共享的，用于存放被虚拟机加载的类的元数据信息：如常量、静态变量、即时编译器编译后的代码。也成为永久代。如果hotspot虚拟机确定一个类的定义信息不会被使用，也会将其回收。回收的基本条件至少有：所有该类的实例被回收，而且装载该类的ClassLoader被回收

##### HotSpot虚拟机

GC算法采用分代收集算法：

1、一个人（对象）出来（new 出来）后会在Eden Space（伊甸园）无忧无虑的生活，直到GC到来打破了他们平静的生活。GC会逐一问清楚每个对象的情况，有没有钱（此对象的引用）啊，因为GC想赚钱呀，有钱的才可以敲诈嘛。然后富人就会进入Survivor Space（幸存者区），穷人的就直接kill掉。

2、并不是进入Survivor Space（幸存者区）后就保证人身是安全的，但至少可以活段时间。GC会定期（可以自定义）会对这些人进行敲诈，亿万富翁每次都给钱，GC很满意，就让其进入了Genured Gen(养老区)。万元户经不住几次敲诈就没钱了，GC看没有啥价值啦，就直接kill掉了。

3、进入到养老区的人基本就可以保证人身安全啦，但是亿万富豪有的也会挥霍成穷光蛋，只要钱没了，GC还是kill掉。

分区的目的：新生区由于对象产生的比较多并且大都是朝生夕灭的，所以直接采用标记-清理算法。而养老区生命力很强，则采用复制算法，针对不同情况使用不同算法。

非heap区域中Perm Gen中放着类、方法的定义，jvm Stack区域放着方法参数、局域变量等的引用，方法执行顺序按照栈的先入后出方式。

##### 小节

简单来讲，jvm的内存回收过程是这样的：

对象在Eden Space创建，当Eden Space满了的时候，gc就把所有在Eden Space中的对象扫描一次，把所有有效的对象复制到第一个Survivor Space，同时把无效的对象所占用的空间释放。当Eden Space再次变满了的时候，就启动移动程序把Eden Space中有效的对象复制到第二个Survivor Space，同时，也将第一个Survivor Space中的有效对象复制到第二个Survivor Space。如果填充到第二个Survivor Space中的有效对象被第一个Survivor Space或Eden Space中的对象引用，那么这些对象就是长期存在的，此时这些对象将被复制到Permanent Generation。

若垃圾收集器依据这种小幅度的调整收集不能腾出足够的空间，就会运行Full GC，此时jvm gc停止所有在堆中运行的线程并执行清除动作。

#### JVM与Spark

每一次放对象的时候，都是放入eden区域，和其中一个survivor区域；另外一个survivor区域是空闲的。

当eden区域和一个survivor区域放满了以后（spark运行过程中，产生的对象实在太多了），就会触发minor gc，小型垃圾回收。把不再使用的对象，从内存中清空，给后面新创建的对象腾出来点儿地方。

清理掉了不再使用的对象之后，那么也会将存活下来的对象（还要继续使用的），放入之前空闲的那一个survivor区域中。这里可能会出现一个问题。默认eden、survior1和survivor2的内存占比是8:1:1。问题是，如果存活下来的对象是1.5，一个survivor区域放不下。此时就可能通过JVM的担保机制（不同JVM版本可能对应的行为），将多余的对象，直接放入老年代了。

如果你的JVM内存不够大的话，可能导致频繁的年轻代内存满溢，频繁的进行minor gc。频繁的minor gc会导致短时间内，有些存活的对象，多次垃圾回收都没有回收掉。会导致这种短声明周期（其实不一定是要长期使用的）对象，年龄过大，垃圾回收次数太多还没有回收到，跑到老年代。

老年代中，可能会因为内存不足，囤积一大堆，短生命周期的，本来应该在年轻代中的，可能马上就要被回收掉的对象。此时，可能导致老年代频繁满溢。频繁进行full gc（全局/全面垃圾回收）。full gc就会去回收老年代中的对象。full gc由于这个算法的设计，是针对的是，老年代中的对象数量很少，满溢进行full gc的频率应该很少，因此采取了不太复杂，但是耗费性能和时间的垃圾回收算法。full gc很慢。

full gc / minor gc，无论是快，还是慢，都会导致jvm的工作线程停止工作，stop the world。简而言之，就是说，gc的时候，spark停止工作了。等着垃圾回收结束。

内存不充足的时候，问题：

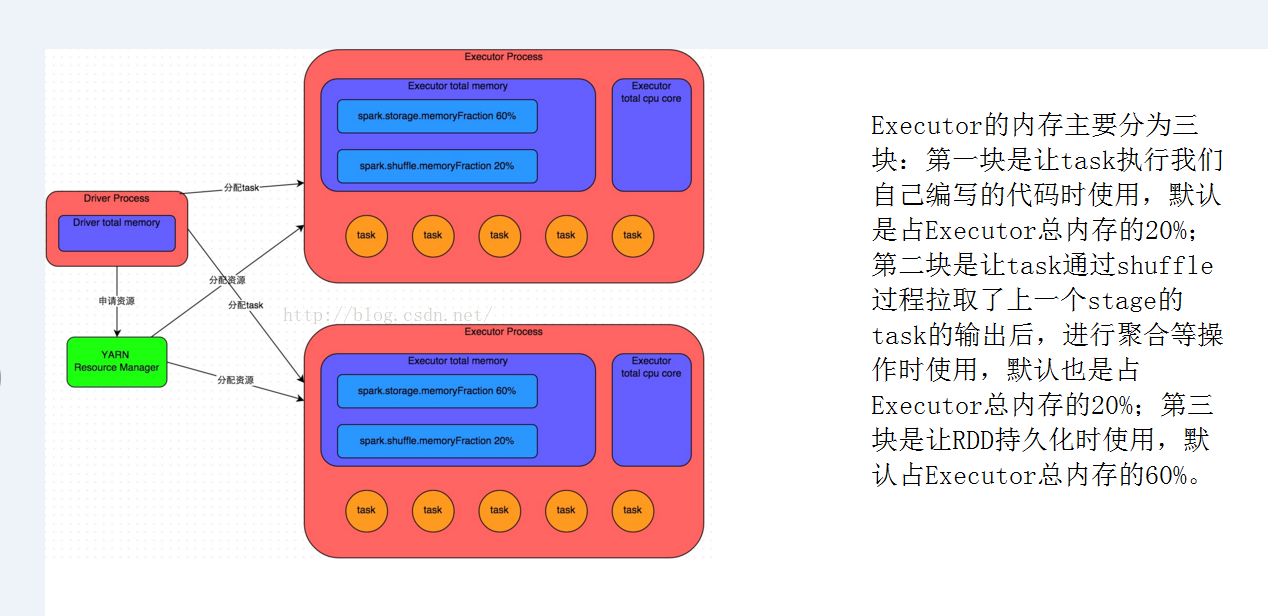
1、频繁minor gc，也会导致频繁spark停止工作

2、老年代囤积大量活跃对象（短生命周期的对象），导致频繁full gc，full gc时间很长，短则数十秒，长则数分钟，甚至数小时。可能导致spark长时间停止工作。

3、严重影响咱们的spark的性能和运行的速度。

#### JVM-降低cache操作占用的内存比

###### 原理



JVM调优的第一个点：降低cache操作的内存占比

spark中，堆内存又被划分成了两块儿，一块儿是专门用来给RDD的cache、persist操作进行RDD数据缓存用的；另外一块儿，就是我们刚才所说的，用来给spark算子函数的运行使用的，存放函数中自己创建的对象。

默认情况下，给RDD cache操作的内存占比，是0.6，60%的内存都给了cache操作了。但是问题是，如果某些情况下，cache不是那么的紧张，问题在于task算子函数中创建的对象过多，然后内存又不太大，导致了频繁的minor gc，甚至频繁full gc，导致spark频繁的停止工作。性能影响会很大。

针对上述这种情况，大家可以在之前我们讲过的那个spark ui。yarn去运行的话，那么就通过yarn的界面，去查看你的spark作业的运行统计，很简单，大家一层一层点击进去就好。可以看到每个stage的运行情况，包括每个task的运行时间、gc时间等等。如果发现gc太频繁，时间太长。此时就可以适当调价这个比例。

降低cache操作的内存占比，大不了用persist操作，选择将一部分缓存的RDD数据写入磁盘，或者序列化方式，配合Kryo序列化类，减少RDD缓存的内存占用；降低cache操作内存占比；对应的，算子函数的内存占比就提升了。这个时候，可能，就可以减少minor gc的频率，同时减少full gc的频率。对性能的提升是有一定的帮助的。

一句话，让task执行算子函数时，有更多的内存可以使用。

spark.storage.memoryFraction，0.6 -> 0.5 -> 0.4 -> 0.2

###### 操作

.set("spark.storage.memoryFraction", "0.5")//进行persist内存占比

#### JVM-堆外内存

###### 原理

有时候，如果你的spark作业处理的数据量特别特别大，几亿数据量；然后spark作业一运行，时不时的报错，shuffle file cannot find，executor、task lost，out of memory（内存溢出）；

可能是说executor的堆外内存不太够用，导致executor在运行的过程中，可能会内存溢出；然后可能导致后续的stage的task在运行的时候，可能要从一些executor中去拉取shuffle map output文件，但是executor可能已经挂掉了，关联的block manager也没有了；所以可能会报shuffle output file not found；resubmitting task；executor lost；spark作业彻底崩溃。

上述情况下，就可以去考虑调节一下executor的堆外内存。也许就可以避免报错；此外，有时，堆外内存调节的比较大的时候，对于性能来说，也会带来一定的提升。

###### 操作

--conf spark.yarn.executor.memoryOverhead=2048

spark-submit脚本里面，去用--conf的方式，去添加配置；一定要注意！！！切记，不是在你的spark作业代码中，用new SparkConf().set()这种方式去设置，不要这样去设置，是没有用的！一定要在spark-submit脚本中去设置。

spark.yarn.executor.memoryOverhead（看名字，顾名思义，针对的是基于yarn的提交模式）

默认情况下，这个堆外内存上限大概是300多M；后来我们通常项目中，真正处理大数据的时候，这里都会出现问题，导致spark作业反复崩溃，无法运行；此时就会去调节这个参数，到至少1G（1024M），甚至说2G、4G

通常这个参数调节上去以后，就会避免掉某些JVM OOM的异常问题，同时呢，会让整体spark作业的性能，得到较大的提升。

#### JVM-增加连接等待时间

###### 原理

JVM调优：垃圾回收

处于垃圾回收过程中，所有的工作线程全部停止；相当于只要一旦进行垃圾回收，spark / executor停止工作，无法提供响应。

executor，优先从自己本地关联的BlockManager中获取某份数据，如果本地block manager没有的话，那么会通过TransferService，去远程连接其他节点上executor的block manager去获取，尝试建立远程的网络连接，并且去拉取数据，正好碰到那个exeuctor的JVM在垃圾回收

###### 操作

此时呢，就会没有响应，无法建立网络连接；会卡住；ok，spark默认的网络连接的超时时长，是60s；如果卡住60s都无法建立连接的话，那么就宣告失败了。

碰到一种情况，偶尔，偶尔，偶尔！！！没有规律！！！某某file。一串file id。uuid（dsfsfd-2342vs--sdf--sdfsd）。not found。file lost。

这种情况下，很有可能是有那份数据的executor在jvm gc。所以拉取数据的时候，建立不了连接。然后超过默认60s以后，直接宣告失败。

报错几次，几次都拉取不到数据的话，可能会导致spark作业的崩溃。也可能会导致DAGScheduler，反复提交几次stage。TaskScheduler，反复提交几次task。大大延长我们的spark作业的运行时间。

可以考虑调节连接的超时时长。

--conf spark.core.connection.ack.wait.timeout=300

spark-submit脚本，切记，不是在new SparkConf().set()这种方式来设置的。

spark.core.connection.ack.wait.timeout（spark core，connection，连接，ack，wait timeout，建立不上连接的时候，超时等待时长）

调节这个值比较大以后，通常来说，可以避免部分的偶尔出现的某某文件拉取失败，某某文件lost掉了。。。

### Shuffle调优

#### Shuffle原理

##### Spark中发生shuffle地方

在spark中，主要是以下几个算子：groupByKey、reduceByKey、countByKey、join，等等。

##### 什么是shuffle

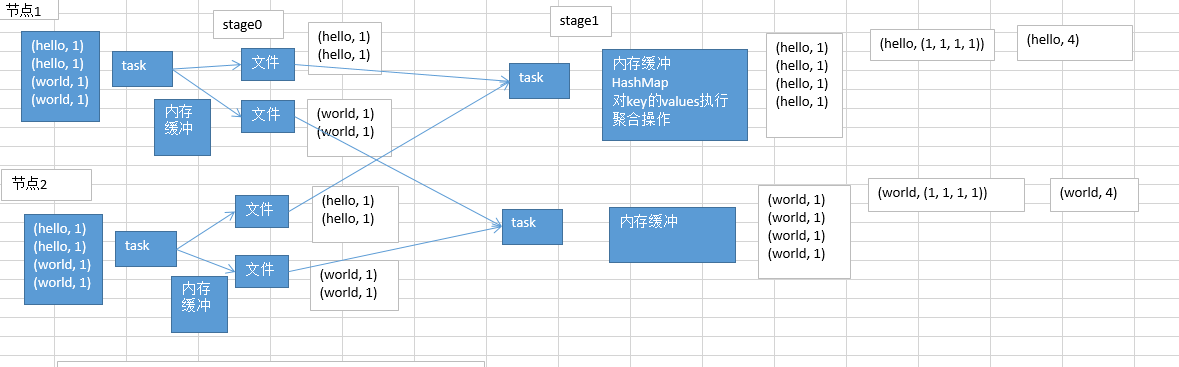
groupByKey，要把分布在集群各个节点上的数据中的同一个key，对应的values，都给集中到一块儿，集中到集群中同一个节点上，更严密一点说，就是集中到一个节点的一个executor的一个task中。

然后呢，集中一个key对应的values之后，才能交给我们来进行处理，<key, Iterable<value>>；reduceByKey，算子函数去对values集合进行reduce操作，最后变成一个value；countByKey，需要在一个task中，获取到一个key对应的所有的value，然后进行计数，统计总共有多少个value；join，RDD<key, value>，RDD<key, value>，只要是两个RDD中，key相同对应的2个value，都能到一个节点的executor的task中，给我们进行处理。

##### 举例

reduceByKey(\_+\_)

问题在于，同一个单词，比如说（hello, 1），可能散落在不同的节点上；对每个单词进行累加计数，就必须让所有单词都跑到同一个节点的一个task中，给一个task来进行处理。



每一个shuffle的前半部分stage的task，每个task都会创建下一个stage的task数量相同的文件，比如下一个stage会有100个task，那么当前stage每个task都会创建100份文件；会将同一个key对应的values，一定是写入同一个文件中的；不同节点上的task，也一定会将同一个key对应的values，写入下一个stage，同一个task对应的文件中。

shuffle的后半部分stage的task，每个task都会从各个节点上的task写的属于自己的那一份文件中，拉取key, value对；然后task会有一个内存缓冲区，然后会用HashMap，进行key, values的汇聚；(key ,values)；

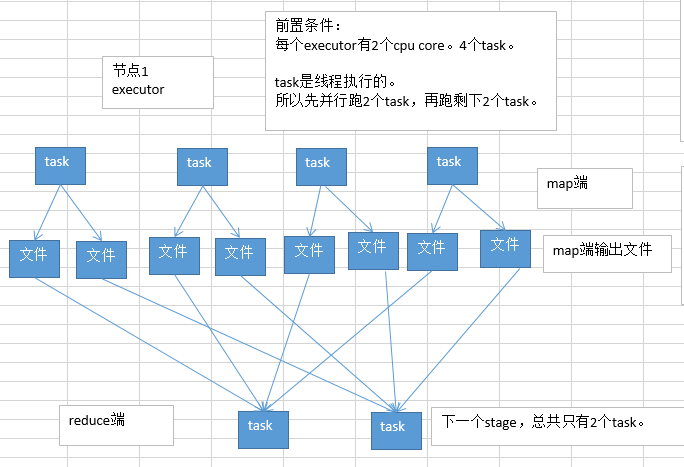
task会用我们自己定义的聚合函数，比如reduceByKey(\_+\_)，把所有values进行一对一的累加；聚合出来最终的值。就完成了shuffle。

注意：shuffle前半部分的task在写入数据到磁盘文件之前，都会先写入一个一个的内存缓冲，内存缓冲满溢之后，再spill溢写到磁盘文件中。

#### Shuffle-map端输出文件合并

###### 原理

Map端不进行合并（自带的最原始的HashShffle）



第一个stage，每个task，都会给第二个stage的每个task创建一份map端的输出文件

第二个stage，每个task，会到各个节点上面去，拉取第一个stage每个task输出的，属于自己的那一份文件。

问题来了：默认的这种shuffle行为，对性能有什么样的恶劣影响呢？

实际生产环境的条件：

100个节点（每个节点一个executor）：100个executor

每个executor：2个cpu core

总共1000个task：每个executor平均10个task

每个节点，10个task，每个节点会输出多少份map端文件？10 \* 1000=1万个文件

总共有多少份map端输出文件？100 \* 10000 = 100万。

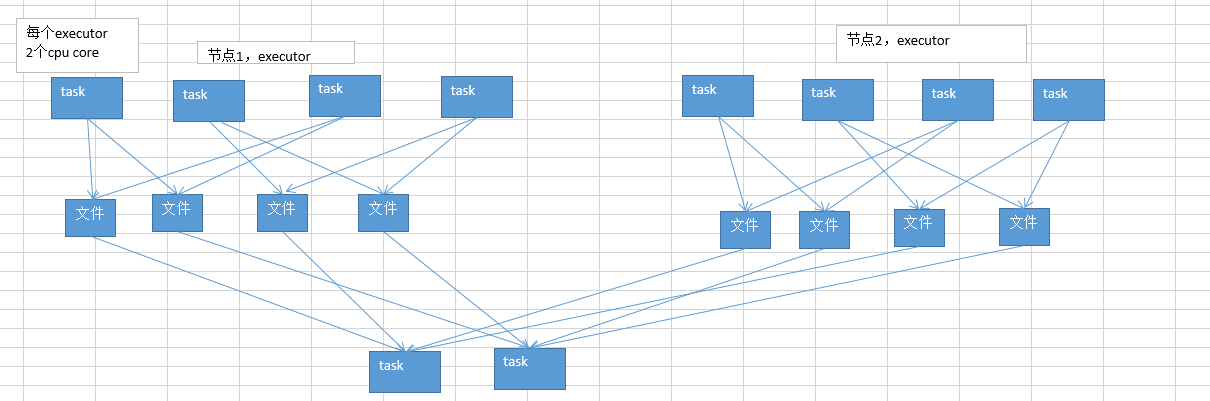
shuffle中的写磁盘的操作，基本上就是shuffle中性能消耗最为严重的部分。

通过上面的分析，一个普通的生产环境的spark job的一个shuffle环节，会写入磁盘100万个文件。磁盘IO对性能和spark作业执行速度的影响，是极其惊人和吓人的。基本上，spark作业的性能，都消耗在shuffle中了，虽然不只是shuffle的map端输出文件这一个部分，但是这里也是非常大的一个性能消耗点。

###### 操作

new SparkConf().set("spark.shuffle.consolidateFiles", "true")

开启shuffle map端输出文件合并的机制；默认情况下，是不开启的，就是会发生如上所述的大量map端输出文件的操作，严重影响性能



开启了map端输出文件的合并机制之后：

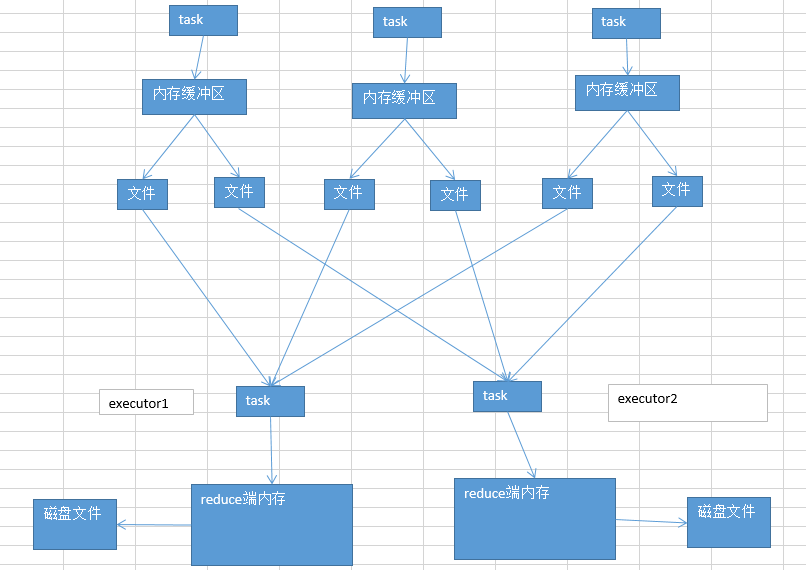
第一个stage，同时就运行cpu core个task，比如cpu core是2个，并行运行2个task；每个task都创建下一个stage的task数量个文件；

第一个stage，并行运行的2个task执行完以后；就会执行另外两个task；另外2个task不会再重新创建输出文件；而是复用之前的task创建的map端输出文件，将数据写入上一批task的输出文件中。

第二个stage，task在拉取数据的时候，就不会去拉取上一个stage每一个task为自己创建的那份输出文件了；而是拉取少量的输出文件，每个输出文件中，可能包含了多个task给自己的map端输出。

#### Shuffle-map端内存缓冲和reduce端内存占比

###### 原理



默认情况下，shuffle的map task，输出到磁盘文件的时候，统一都会先写入每个task自己关联的一个内存缓冲区。这个缓冲区大小，默认是32kb。每一次，当内存缓冲区满溢之后，才会进行spill操作，溢写操作，溢写到磁盘文件中去。

reduce端task，在拉取到数据之后，会用hashmap的数据格式，来对各个key对应的values进行汇聚。

针对每个key对应的values，执行我们自定义的聚合函数的代码，比如\_ + \_（把所有values累加起来）

reduce task，在进行汇聚、聚合等操作的时候，实际上，使用的就是自己对应的executor的内存，executor（jvm进程，堆），默认executor内存中划分给reduce task进行聚合的比例，是0.2。

问题来了，因为比例是0.2，所以，理论上，很有可能会出现，拉取过来的数据很多，那么在内存中，放不下；这个时候，默认的行为，就是说，将在内存放不下的数据，都spill（溢写）到磁盘文件中去。

默认，map端内存缓冲是每个task，32kb。

默认，reduce端聚合内存比例，是0.2，也就是20%。

如果map端的task，处理的数据量比较大，但是呢，你的内存缓冲大小是固定的。可能会出现什么样的情况？

每个task就处理320kb，32kb，总共会向磁盘溢写320 / 32 = 10次。

每个task处理32000kb，32kb，总共会向磁盘溢写32000 / 32 = 1000次。

在map task处理的数据量比较大的情况下，而你的task的内存缓冲默认是比较小的，32kb。可能会造成多次的map端往磁盘文件的spill溢写操作，发生大量的磁盘IO，从而降低性能。

reduce端聚合内存，占比。默认是0.2。如果数据量比较大，reduce task拉取过来的数据很多，那么就会频繁发生reduce端聚合内存不够用，频繁发生spill操作，溢写到磁盘上去。而且最要命的是，磁盘上溢写的数据量越大，后面在进行聚合操作的时候，很可能会多次读取磁盘中的数据，进行聚合。

默认不调优，在数据量比较大的情况下，可能频繁地发生reduce端的磁盘文件的读写。

###### 操作

调节map task内存缓冲：spark.shuffle.file.buffer，默认32k（spark 1.3.x不是这个参数，后面还有一个后缀，kb；spark 1.5.x以后，变了，就是现在这个参数）

调节reduce端聚合内存占比：spark.shuffle.memoryFraction，0.2

在实际生产环境中，我们在什么时候来调节两个参数？

看Spark UI，如果你的公司是决定采用standalone模式，那么狠简单，你的spark跑起来，会显示一个Spark UI的地址，4040的端口，进去看，依次点击进去，可以看到，你的每个stage的详情，有哪些executor，有哪些task，每个task的shuffle write和shuffle read的量，shuffle的磁盘和内存，读写的数据量；如果是用的yarn模式来提交，课程最前面，从yarn的界面进去，点击对应的application，进入Spark UI，查看详情。

如果发现shuffle 磁盘的write和read，很大。这个时候，就意味着最好调节一些shuffle的参数。进行调优。首先当然是考虑开启map端输出文件合并机制。

调节上面说的那两个参数。调节的时候的原则。spark.shuffle.file.buffer，每次扩大一倍，然后看看效果，64，128；spark.shuffle.memoryFraction，每次提高0.1，看看效果。

不能调节的太大，太大了以后过犹不及，因为内存资源是有限的，你这里调节的太大了，其他环节的内存使用就会有问题了。

调节了以后，效果？map task内存缓冲变大了，减少spill到磁盘文件的次数；reduce端聚合内存变大了，减少spill到磁盘的次数，而且减少了后面聚合读取磁盘文件的数量。

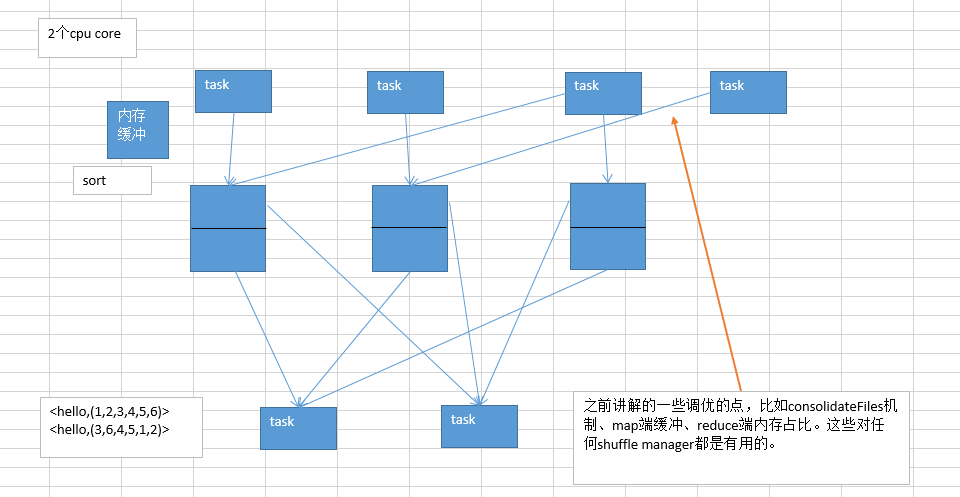
项目中

.set("spark.shuffle.file.buffer", "64")//调节map task 内存缓冲大小，这个缓冲在写入磁盘之前。

.set("spark.shuffle.memoryFraction", "0.3")//reduce 端task 后面，内存缓冲HashMap（0.2）对key的value执行聚合操作，然后spill磁盘。这里指的是内存缓冲

#### Shuffle-Sort Shuffle

###### 原理



SortShuffleManager与HashShuffleManager两点不同：

1、SortShuffleManager会对每个reduce task要处理的数据，进行排序（默认的）。

2、SortShuffleManager会避免像HashShuffleManager那样，默认就去创建多份磁盘文件。一个task，只会写入一个磁盘文件，不同reduce task的数据，用offset来划分界定。

SortShuffleManager：自己可以设定一个阈值，默认是200，当reduce task数量少于等于200；map task创建的输出文件小于等于200的；最后会将所有的输出文件合并为一份文件。

这样做的好处，就是避免了sort排序，节省了性能开销。而且还能将多个reduce task的文件合并成一份文件。节省了reduce task拉取数据的时候的磁盘IO的开销。

###### 操作

spark.shuffle.manager：hash、sort、tungsten-sort（自己实现内存管理）

spark.shuffle.sort.bypassMergeThreshold：200

#### Shuffle总结

###### hash、sort、tungsten-sort（Spark自己实现内存管理）。如何来选择

1、需不需要数据默认就让spark给你进行排序？就好像mapreduce，默认就是有按照key的排序。如果不需要的话，其实还是建议搭建就使用最基本的HashShuffleManager，因为最开始就是考虑的是不排序，换取高性能；

2、什么时候需要用sort shuffle manager？如果你需要你的那些数据按key排序了，那么就选择这种吧，而且要注意，reduce task的数量应该是超过200的，这样sort、merge（多个文件合并成一个）的机制，才能生效把。但是这里要注意，你一定要自己考量一下，有没有必要在shuffle的过程中，就做这个事情，毕竟对性能是有影响的。

3、如果你不需要排序，而且你希望你的每个task输出的文件最终是会合并成一份的，你自己认为可以减少性能开销；可以去调节bypassMergeThreshold这个阈值，比如你的reduce task数量是500，默认阈值是200，所以默认还是会进行sort和直接merge的；可以将阈值调节成550，不会进行sort，按照hash的做法，每个reduce task创建一份输出文件，最后合并成一份文件。（一定要提醒大家，这个参数，其实我们通常不会在生产环境里去使用，也没有经过验证说，这样的方式，到底有多少性能的提升）

4、如果你想选用sort based shuffle manager，而且你们公司的spark版本比较高，是1.5.x版本的，那么可以考虑去尝试使用tungsten-sort shuffle manager。看看性能的提升与稳定性怎么样。

###### 总结

1、在生产环境中，不建议大家贸然使用第三点和第四点：

2、如果你不想要你的数据在shuffle时排序，那么就自己设置一下，用hash shuffle manager。

3、如果你的确是需要你的数据在shuffle时进行排序的，那么就默认不用动，默认就是sort shuffle manager；或者是什么？如果你压根儿不care是否排序这个事儿，那么就默认让他就是sort的。调节一些其他的参数（consolidation机制）。（80%，都是用这种）

spark.shuffle.manager：hash、sort、tungsten-sort

new SparkConf().set("spark.shuffle.manager", "hash")

new SparkConf().set("spark.shuffle.manager", "tungsten-sort")

// 默认就是，new SparkConf().set("spark.shuffle.manager", "sort")

new SparkConf().set("spark.shuffle.sort.bypassMergeThreshold", "550")

### 算子调优

#### 算子调优-使用MapPartitions提示map算子性能

###### 注意

什么时候比较适合用MapPartitions系列操作，就是说，数据量不是特别大的时候，都可以用这种MapPartitions系列操作，性能还是非常不错的，是有提升的。比如原来是15分钟，（曾经有一次性能调优），12分钟。10分钟->9分钟。

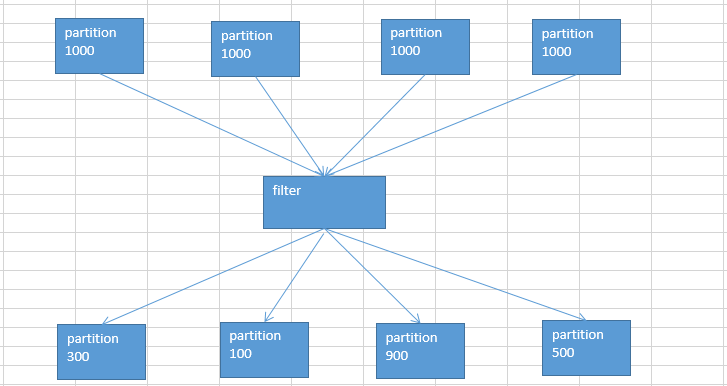
但是也有过出问题的经验，MapPartitions只要一用，直接OOM，内存溢出，崩溃。

在项目中，自己先去估算一下RDD的数据量，以及每个partition的量，还有自己分配给每个executor的内存资源。看看一下子内存容纳所有的partition数据，行不行。如果行，可以试一下，能跑通就好。性能肯定是有提升的。

但是试了一下以后，发现，不行，OOM了，那就放弃吧。

#### 算子调优-Filter算子之后使用coalesce

###### 原理



默认情况下，经过了这种filter之后，RDD中的每个partition的数据量，可能都不太一样了。（原本每个partition的数据量可能是差不多的）

问题：

1、每个partition数据量变少了，但是在后面进行处理的时候，还是要跟partition数量一样数量的task，来进行处理；有点浪费task计算资源。

2、每个partition的数据量不一样，会导致后面的每个task处理每个partition的时候，每个task要处理的数据量就不同，这个时候很容易发生什么问题？数据倾斜。。。。

比如说，第二个partition的数据量才100；但是第三个partition的数据量是900；那么在后面的task处理逻辑一样的情况下，不同的task要处理的数据量可能差别达到了9倍，甚至10倍以上；同样也就导致了速度的差别在9倍，甚至10倍以上。

这样的话呢，就会导致有些task运行的速度很快；有些task运行的速度很慢。这，就是数据倾斜。

针对上述的两个问题，我们希望应该能够怎么样？

1、针对第一个问题，我们希望可以进行partition的压缩吧，因为数据量变少了，那么partition其实也完全可以对应的变少。比如原来是4个partition，现在完全可以变成2个partition。那么就只要用后面的2个task来处理即可。就不会造成task计算资源的浪费。（不必要，针对只有一点点数据的partition，还去启动一个task来计算）

2、针对第二个问题，其实解决方案跟第一个问题是一样的；也是去压缩partition，尽量让每个partition的数据量差不多。那么这样的话，后面的task分配到的partition的数据量也就差不多。不会造成有的task运行速度特别慢，有的task运行速度特别快。避免了数据倾斜的问题。

###### 操作

有了解决问题的思路之后，接下来，我们该怎么来做呢？实现？

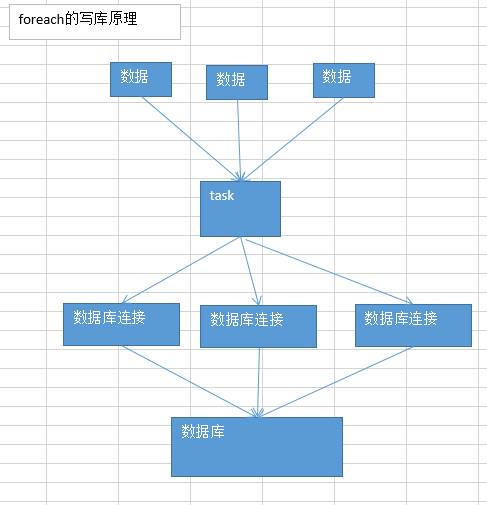
coalesce算子

主要就是用于在filter操作之后，针对每个partition的数据量各不相同的情况，来压缩partition的数量。减少partition的数量，而且让每个partition的数据量都尽量均匀紧凑。

从而便于后面的task进行计算操作，在某种程度上，能够一定程度的提升性能。

#### 算子调优-使用foreachPartition优化写数据库性能

###### 原理



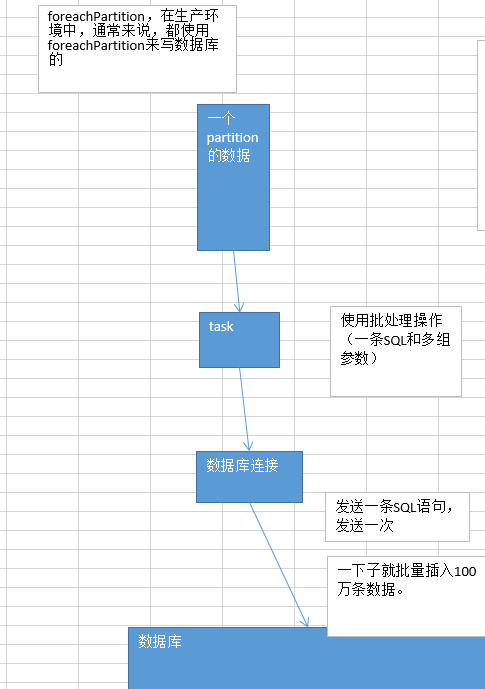
默认的foreach的性能缺陷在哪里？

首先，对于每条数据，都要单独去调用一次function，task为每个数据，都要去执行一次function函数。如果100万条数据，（一个partition），调用100万次。性能比较差。

另外一个非常非常重要的一点，如果每个数据，你都去创建一个数据库连接的话，那么你就得创建100万次数据库连接。但是要注意的是，数据库连接的创建和销毁，都是非常非常消耗性能的。虽然我们之前已经用了数据库连接池，只是创建了固定数量的数据库连接。

你还是得多次通过数据库连接，往数据库（MySQL）发送一条SQL语句，然后MySQL需要去执行这条SQL语句。如果有100万条数据，那么就是100万次发送SQL语句。

以上两点（数据库连接，多次发送SQL语句），都是非常消耗性能的。



用了foreachPartition算子之后，好处在哪里？

1、对于我们写的function函数，就调用一次，一次传入一个partition所有的数据

2、主要创建或者获取一个数据库连接就可以

3、只要向数据库发送一次SQL语句和多组参数即可

#### 算子调优-使用repartition解决Spark SQL低并行度的性能问题

###### 原理

并行度：之前说过，并行度是自己可以调节，或者说是设置的。

1、spark.default.parallelism

2、textFile()，传入第二个参数，指定partition数量（比较少用）

咱们的项目代码中，没有设置并行度，实际上，在生产环境中，是最好自己设置一下的。官网有推荐的设置方式，你的spark-submit脚本中，会指定你的application总共要启动多少个executor，100个；每个executor多少个cpu core，2~3个；总共application，有cpu core，200个。

官方推荐，根据你的application的总cpu core数量（在spark-submit中可以指定，200个），自己手动设置spark.default.parallelism参数，指定为cpu core总数的2~3倍。400~600个并行度。600。

承上启下，你设置的这个并行度，在哪些情况下会生效？哪些情况下，不会生效？

如果你压根儿没有使用Spark SQL（DataFrame），那么你整个spark application默认所有stage的并行度都是你设置的那个参数。（除非你使用coalesce算子缩减过partition数量）

问题来了，Spark SQL，用了。用Spark SQL的那个stage的并行度，你没法自己指定。Spark SQL自己会默认根据hive表对应的hdfs文件的block，自动设置Spark SQL查询所在的那个stage的并行度。你自己通过spark.default.parallelism参数指定的并行度，只会在没有Spark SQL的stage中生效。

比如你第一个stage，用了Spark SQL从hive表中查询出了一些数据，然后做了一些transformation操作，接着做了一个shuffle操作（groupByKey）；下一个stage，在shuffle操作之后，做了一些transformation操作。hive表，对应了一个hdfs文件，有20个block；你自己设置了spark.default.parallelism参数为100。

你的第一个stage的并行度，是不受你的控制的，就只有20个task；第二个stage，才会变成你自己设置的那个并行度，100。

问题在哪里？

Spark SQL默认情况下，它的那个并行度，咱们没法设置。可能导致的问题，也许没什么问题，也许很有问题。Spark SQL所在的那个stage中，后面的那些transformation操作，可能会有非常复杂的业务逻辑，甚至说复杂的算法。如果你的Spark SQL默认把task数量设置的很少，20个，然后每个task要处理为数不少的数据量，然后还要执行特别复杂的算法。

这个时候，就会导致第一个stage的速度，特别慢。第二个stage，1000个task，刷刷刷，非常快。

###### 操作

解决上述Spark SQL无法设置并行度和task数量的办法，是什么呢？

repartition算子，你用Spark SQL这一步的并行度和task数量，肯定是没有办法去改变了。但是呢，可以将你用Spark SQL查询出来的RDD，使用repartition算子，去重新进行分区，此时可以分区成多个partition，比如从20个partition，分区成100个。

然后呢，从repartition以后的RDD，再往后，并行度和task数量，就会按照你预期的来了。就可以避免跟Spark SQL绑定在一个stage中的算子，只能使用少量的task去处理大量数据以及复杂的算法逻辑。

#### 算子调优-reduceByKey本地聚合

###### 原理

reduceByKey，相较于普通的shuffle操作（比如groupByKey），它的一个特点，就是说，会进行map端的本地聚合。

对map端给下个stage每个task创建的输出文件中，写数据之前，就会进行本地的combiner操作，也就是说对每一个key，对应的values，都会执行你的算子函数（) + \_）

用reduceByKey对性能的提升：

1、在本地进行聚合以后，在map端的数据量就变少了，减少磁盘IO。而且可以减少磁盘空间的占用。

2、下一个stage，拉取数据的量，也就变少了。减少网络的数据传输的性能消耗。

3、在reduce端进行数据缓存的内存占用变少了。

4、reduce端，要进行聚合的数据量也变少了。

reduceByKey在什么情况下使用呢？

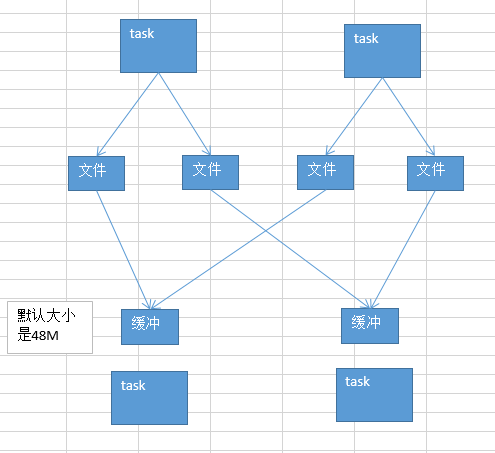
1、非常普通的，比如说，就是要实现类似于wordcount程序一样的，对每个key对应的值，进行某种数据公式或者算法的计算（累加、类乘）

2、对于一些类似于要对每个key进行一些字符串拼接的这种较为复杂的操作，可以自己衡量一下，其实有时，也是可以使用reduceByKey来实现的。但是不太好实现。如果真能够实现出来，对性能绝对是有帮助的。（shuffle基本上就占了整个spark作业的90%以上的性能消耗，主要能对shuffle进行一定的调优，都是有价值的）

### Throbleshooting

#### Troubleshooting-控制shuffle reduce端缓冲大小避免OOM

###### 原理



map端的task是不断的输出数据的，数据量可能是很大的。

但是，其实reduce端的task，并不是等到map端task将属于自己的那份数据全部写入磁盘文件之后，再去拉取的。map端写一点数据，reduce端task就会拉取一小部分数据，立即进行后面的聚合、算子函数的应用。

每次reduece能够拉取多少数据，就由buffer来决定。因为拉取过来的数据，都是先放在buffer中的。然后才用后面的executor分配的堆内存占比（0.2），hashmap，去进行后续的聚合、函数的执行。

reduce端缓冲（buffer），可能会出什么问题？

可能是会出现，默认是48MB，也许大多数时候，reduce端task一边拉取一边计算，不一定一直都会拉满48M的数据。可能大多数时候，拉取个10M数据，就计算掉了。

大多数时候，也许不会出现什么问题。但是有的时候，map端的数据量特别大，然后写出的速度特别快。reduce端所有task，拉取的时候，全部达到自己的缓冲的最大极限值，缓冲，48M，全部填满。

这个时候，再加上你的reduce端执行的聚合函数的代码，可能会创建大量的对象。也许，一下子，内存就撑不住了，就会OOM。reduce端的内存中，就会发生内存溢出的问题。

针对上述的可能出现的问题，我们该怎么来解决呢？

这个时候，就应该减少reduce端task缓冲的大小。我宁愿多拉取几次，但是每次同时能够拉取到reduce端每个task的数量，比较少，就不容易发生OOM内存溢出的问题。（比如，可以调节成12M），但是，性能一定是有所下降的，你要拉取的次数就多了。就走更多的网络传输开销。

再来说说，reduce端缓冲大小的另外一面，关于性能调优的一面：

咱们假如说，你的Map端输出的数据量也不是特别大，然后你的整个application的资源也特别充足。200个executor、5个cpu core、10G内存。

其实可以尝试去增加这个reduce端缓冲大小的，比如从48M，变成96M。那么这样的话，每次reduce task能够拉取的数据量就很大。需要拉取的次数也就变少了。比如原先需要拉取100次，现在只要拉取50次就可以执行完了。

对网络传输性能开销的减少，以及reduce端聚合操作执行的次数的减少，都是有帮助的。

最终达到的效果，就应该是性能上的一定程度上的提升。

一定要注意，资源足够的时候，再去做这个事儿。

###### 操作

spark.reducer.maxSizeInFlight，48

.set("spark.reducer.maxSizeInFlight", "24")

#### Troubleshooting-解决JVM GC导致的shuffle文件拉取失败

###### 原理

比如，executor的JVM进程，可能内存不是很够用了。那么此时可能就会执行GC。minor GC or full GC。总之一旦发生了JVM之后，就会导致executor内，所有的工作线程全部停止，比如BlockManager，基于netty的网络通信。

下一个stage的executor，可能是还没有停止掉的，task想要去上一个stage的task所在的exeuctor，去拉取属于自己的数据，结果由于对方正在gc，就导致拉取了半天没有拉取到。就很可能会报出，shuffle file not found。但是，可能下一个stage又重新提交了stage或task以后，再执行就没有问题了，因为可能第二次就没有碰到JVM在gc了。

###### 操作

spark.shuffle.io.maxRetries 3

第一个参数，意思就是说，shuffle文件拉取的时候，如果没有拉取到（拉取失败），最多或重试几次（会重新拉取几次文件），默认是3次。

spark.shuffle.io.retryWait 5s

第二个参数，意思就是说，每一次重试拉取文件的时间间隔，默认是5s钟。

默认情况下，假如说第一个stage的executor正在进行漫长的full gc。第二个stage的executor尝试去拉取文件，结果没有拉取到，默认情况下，会反复重试拉取3次，每次间隔是五秒钟。最多只会等待3 \* 5s = 15s。如果15s内，没有拉取到shuffle file。就会报出shuffle file not found。

针对这种情况，我们完全可以进行预备性的参数调节。增大上述两个参数的值，达到比较大的一个值，尽量保证第二个stage的task，一定能够拉取到上一个stage的输出文件。避免报shuffle file not found。然后可能会重新提交stage和task去执行。那样反而对性能也不好。

spark.shuffle.io.maxRetries 60

spark.shuffle.io.retryWait 60s

.set("spark.shuffle.io.maxRetries", "60")

.set("spark.shuffle.io.retryWait", "60")

#### Thoubleshooting-YARN队列资源不足导致的application直接失败

###### 现象

如果说，你是基于yarn来提交spark。比如yarn-cluster或者yarn-client。你可以指定提交到某个hadoop队列上的。每个队列都是可以有自己的资源的。

跟大家说一个生产环境中的，给spark用的yarn资源队列的情况：500G内存，200个cpu core。

比如说，某个spark application，在spark-submit里面你自己配了，executor，80个；每个executor，4G内存；每个executor，2个cpu core。你的spark作业每次运行，大概要消耗掉320G内存，以及160个cpu core。

乍看起来，咱们的队列资源，是足够的，500G内存，280个cpu core。

首先，第一点，你的spark作业实际运行起来以后，耗费掉的资源量，可能是比你在spark-submit里面配置的，以及你预期的，是要大一些的。400G内存，190个cpu core。

那么这个时候，的确，咱们的队列资源还是有一些剩余的。但是问题是，如果你同时又提交了一个spark作业上去，一模一样的。那就可能会出问题。

第二个spark作业，又要申请320G内存+160个cpu core。结果，发现队列资源不足。。。。

此时，可能会出现两种情况：（备注，具体出现哪种情况，跟你的YARN、Hadoop的版本，你们公司的一些运维参数，以及配置、硬件、资源肯能都有关系）

1、YARN，发现资源不足时，你的spark作业，并没有hang在那里，等待资源的分配，而是直接打印一行fail的log，直接就fail掉了。

2、YARN，发现资源不足，你的spark作业，就hang在那里。一直等待之前的spark作业执行完，等待有资源分配给自己来执行。

###### 采用如下方案

1、在你的J2EE（我们这个项目里面，spark作业的运行，之前说过了，J2EE平台触发的，执行spark-submit脚本），限制，同时只能提交一个spark作业到yarn上去执行，确保一个spark作业的资源肯定是有的。

2、你应该采用一些简单的调度区分的方式，比如说，你有的spark作业可能是要长时间运行的，比如运行30分钟；有的spark作业，可能是短时间运行的，可能就运行2分钟。此时，都提交到一个队列上去，肯定不合适。很可能出现30分钟的作业卡住后面一大堆2分钟的作业。分队列，可以申请（跟你们的YARN、Hadoop运维的同学申请）。你自己给自己搞两个调度队列。每个队列的根据你要执行的作业的情况来设置。在你的J2EE程序里面，要判断，如果是长时间运行的作业，就干脆都提交到某一个固定的队列里面去把；如果是短时间运行的作业，就统一提交到另外一个队列里面去。这样，避免了长时间运行的作业，阻塞了短时间运行的作业。

3、你的队列里面，无论何时，只会有一个作业在里面运行。那么此时，就应该用我们之前讲过的性能调优的手段，去将每个队列能承载的最大的资源，分配给你的每一个spark作业，比如80个executor；6G的内存；3个cpu core。尽量让你的spark作业每一次运行，都达到最满的资源使用率，最快的速度，最好的性能；并行度，240个cpu core，720个task。

4、在J2EE中，通过线程池的方式（一个线程池对应一个资源队列），来实现上述我们说的方案。例如：

ExecutorService threadPool = Executors.newFixedThreadPool(1);

threadPool.submit(new Runnable() {

@Override

public void run() {

}

});

因为线程池本身就是一个队列的模式，符合我们的要求。

#### Troubleshooting-各种序列化导致的报错

###### 现象

用client模式去提交spark作业，观察本地打印出来的log。如果出现了类似于Serializable、Serialize等等字眼，报错的log，那么恭喜大家，就碰到了序列化问题导致的报错。

###### 原因

1、你的算子函数里面，如果使用到了外部的自定义类型的变量，那么此时，就要求你的自定义类型，必须是可序列化的。

final Teacher teacher = new Teacher("leo");

studentsRDD.foreach(new VoidFunction() {

public void call(Row row) throws Exception {

String teacherName = teacher.getName();

....

}

});

public class Teacher implements Serializable {

}

2、如果要将自定义的类型，作为RDD的元素类型，那么自定义的类型也必须是可以序列化的

JavaPairRDD<Integer, Teacher> teacherRDD

JavaPairRDD<Integer, Student> studentRDD

studentRDD.join(teacherRDD)

public class Teacher implements Serializable {

}

public class Student implements Serializable {

}

3、不能在上述两种情况下，去使用一些第三方的，不支持序列化的类型

Connection conn =

studentsRDD.foreach(new VoidFunction() {

public void call(Row row) throws Exception {

conn.....

}

});

Connection是不支持序列化的

#### Troubleshooting-算子函数返回NULL导致的问题

###### 现象

在算子函数中，返回null

// return actionRDD.mapToPair(new PairFunction<Row, String, Row>() {

//

// private static final long serialVersionUID = 1L;

//

// @Override

// public Tuple2<String, Row> call(Row row) throws Exception {

// return new Tuple2<String, Row>("-999", RowFactory.createRow("-999"));

// }

//

// });

大家可以看到，在有些算子函数里面，是需要我们有一个返回值的。但是，有时候，我们可能对某些值，就是不想有什么返回值。我们如果直接返回NULL的话，那么可以不幸的告诉大家，是不行的，会报错的。

Scala.Math(NULL)，异常

###### 解决方案

1、在返回的时候，返回一些特殊的值，不要返回null，比如“-999”

2、在通过算子获取到了一个RDD之后，可以对这个RDD执行filter操作，进行数据过滤。filter内，可以对数据进行判定，如果是-999，那么就返回false，给过滤掉就可以了。

3、大家不要忘了，之前咱们讲过的那个算子调优里面的coalesce算子，在filter之后，可以使用coalesce算子压缩一下RDD的partition的数量，让各个partition的数据比较紧凑一些。也能提升一些性能。

#### Troubleshooting-yarn-clent模式导致的网卡流量剧增

###### 现象

yarn-client模式下，会产生什么样的问题呢？

由于咱们的driver是启动在本地机器的，而且driver是全权负责所有的任务的调度的，也就是说要跟yarn集群上运行的多个executor进行频繁的通信（中间有task的启动消息、task的执行统计消息、task的运行状态、shuffle的输出结果）。

咱们来想象一下。比如你的executor有100个，stage有10个，task有1000个。每个stage运行的时候，都有1000个task提交到executor上面去运行，平均每个executor有10个task。接下来问题来了，driver要频繁地跟executor上运行的1000个task进行通信。通信消息特别多，通信的频率特别高。运行完一个stage，接着运行下一个stage，又是频繁的通信。

在整个spark运行的生命周期内，都会频繁的去进行通信和调度。所有这一切通信和调度都是从你的本地机器上发出去的，和接收到的。这是最要人命的地方。你的本地机器，很可能在30分钟内（spark作业运行的周期内），进行频繁大量的网络通信。那么此时，你的本地机器的网络通信负载是非常非常高的。会导致你的本地机器的网卡流量会激增！！！

###### 解决方案

实际上解决的方法很简单，就是心里要清楚，yarn-client模式是什么情况下，可以使用的？yarn-client模式，通常咱们就只会使用在测试环境中，你写好了某个spark作业，打了一个jar包，在某台测试机器上，用yarn-client模式去提交一下。因为测试的行为是偶尔为之的，不会长时间连续提交大量的spark作业去测试。还有一点好处，yarn-client模式提交，可以在本地机器观察到详细全面的log。通过查看log，可以去解决线上报错的故障（troubleshooting）、对性能进行观察并进行性能调优。

实际上线了以后，在生产环境中，都得用yarn-cluster模式，去提交你的spark作业。

yarn-cluster模式，就跟你的本地机器引起的网卡流量激增的问题，就没有关系了。也就是说，就算有问题，也应该是yarn运维团队和基础运维团队之间的事情了。使用了yarn-cluster模式以后，就不是你的本地机器运行Driver，进行task调度了。是yarn集群中，某个节点会运行driver进程，负责task调度。

#### Troubleshooting-解决yarn-cluster模式的JVM内存溢出无法执行

###### 现象

实践经验，碰到的yarn-cluster的问题：

有的时候，运行一些包含了spark sql的spark作业，可能会碰到yarn-client模式下，可以正常提交运行；yarn-cluster模式下，可能是无法提交运行的，会报出JVM的PermGen（永久代）的内存溢出，OOM。

yarn-client模式下，driver是运行在本地机器上的，spark使用的JVM的PermGen的配置，是本地的spark-class文件（spark客户端是默认有配置的），JVM的永久代的大小是128M，这个是没有问题的；但是呢，在yarn-cluster模式下，driver是运行在yarn集群的某个节点上的，使用的是没有经过配置的默认设置（PermGen永久代大小），82M。

spark-sql，它的内部是要进行很复杂的SQL的语义解析、语法树的转换等等，特别复杂，在这种复杂的情况下，如果说你的sql本身特别复杂的话，很可能会比较导致性能的消耗，内存的消耗。可能对PermGen永久代的占用会比较大。

所以，此时，如果对永久代的占用需求，超过了82M的话，但是呢又在128M以内；就会出现如上所述的问题，yarn-client模式下，默认是128M，这个还能运行；如果在yarn-cluster模式下，默认是82M，就有问题了。会报出PermGen Out of Memory error log。

###### 解决放啊

既然是JVM的PermGen永久代内存溢出，那么就是内存不够用。咱们呢，就给yarn-cluster模式下的，driver的PermGen多设置一些。

spark-submit脚本中，加入以下配置即可：

--conf spark.driver.extraJavaOptions="-XX:PermSize=128M -XX:MaxPermSize=256M"

这个就设置了driver永久代的大小，默认是128M，最大是256M。那么，这样的话，就可以基本保证你的spark作业不会出现上述的yarn-cluster模式导致的永久代内存溢出的问题。

#### Troubleshooting-错误的持久化方式以及checkpoint的使用

###### 持久化

错误的持久化使用方式

usersRDD，想要对这个RDD做一个cache，希望能够在后面多次使用这个RDD的时候，不用反复重新计算RDD；可以直接使用通过各个节点上的executor的BlockManager管理的内存 / 磁盘上的数据，避免重新反复计算RDD。

usersRDD.cache()

usersRDD.count()

usersRDD.take()

上面这种方式，不要说会不会生效了，实际上是会报错的。会报什么错误呢？会报一大堆file not found的错误。

正确的持久化使用方式

usersRDD

usersRDD = usersRDD.cache()

val cachedUsersRDD = usersRDD.cache()

之后再去使用usersRDD，或者cachedUsersRDD，就可以了。就不会报错了。所以说，这个是咱们的持久化的正确的使用方式。

###### Checkpoint

Checkpoint

持久化，大多数时候，都是会正常工作的。但是就怕，有些时候，会出现意外。

比如说，缓存在内存中的数据，可能莫名其妙就丢失掉了。或者说，存储在磁盘文件中的数据，莫名其妙就没了，文件被误删了。

出现上述情况的时候，接下来，如果要对这个RDD执行某些操作，可能会发现RDD的某个partition找不到了。

对消失的partition重新计算，计算完以后再缓存和使用。

有些时候，计算某个RDD，可能是极其耗时的。可能RDD之前有大量的父RDD。那么如果你要重新计算一个partition，可能要重新计算之前所有的父RDD对应的partition。

这种情况下，就可以选择对这个RDD进行checkpoint，以防万一。进行checkpoint，就是说，会将RDD的数据，持久化一份到容错的文件系统上（比如hdfs）。

在对这个RDD进行计算的时候，如果发现它的缓存数据不见了。优先就是先找一下有没有checkpoint数据（到hdfs上面去找）。如果有的话，就使用checkpoint数据了。不至于说是去重新计算。

checkpoint，其实就是可以作为是cache的一个备胎。如果cache失效了，checkpoint就可以上来使用了。

checkpoint有利有弊，利在于，提高了spark作业的可靠性，一旦发生问题，还是很可靠的，不用重新计算大量的rdd；但是弊在于，进行checkpoint操作的时候，也就是将rdd数据写入hdfs中的时候，还是会消耗性能的。

checkpoint，用性能换可靠性。

checkpoint原理

1、在代码中，用SparkContext，设置一个checkpoint目录，可以是一个容错文件系统的目录，比如hdfs；

2、在代码中，对需要进行checkpoint的rdd，执行RDD.checkpoint()；

3、RDDCheckpointData（spark内部的API），接管你的RDD，会标记为marked for checkpoint，准备进行checkpoint

4、你的job运行完之后，会调用一个finalRDD.doCheckpoint()方法，会顺着rdd lineage，回溯扫描，发现有标记为待checkpoint的rdd，就会进行二次标记，inProgressCheckpoint，正在接受checkpoint操作

5、job执行完之后，就会启动一个内部的新job，去将标记为inProgressCheckpoint的rdd的数据，都写入hdfs文件中。（备注，如果rdd之前cache过，会直接从缓存中获取数据，写入hdfs中；如果没有cache过，那么就会重新计算一遍这个rdd，再checkpoint）

6、将checkpoint过的rdd之前的依赖rdd，改成一个CheckpointRDD\*，强制改变你的rdd的lineage。后面如果rdd的cache数据获取失败，直接会通过它的上游CheckpointRDD，去容错的文件系统，比如hdfs，中，获取checkpoint的数据。

checkpoint的使用方法

1、SparkContext，设置checkpoint目录

2、对RDD执行checkpoint操作

### 数据倾斜

#### 发生数据倾斜以后的现象

1、你的大部分的task，都执行的特别特别快，刷刷刷，就执行完了（你要用client模式，standalone client，yarn client，本地机器主要一执行spark-submit脚本，就会开始打印log），task175 finished；剩下几个task，执行的特别特别慢，前面的task，一般1s可以执行完5个；最后发现1000个task，998，999 task，要执行1个小时，2个小时才能执行完一个task。

2、运行的时候，其他task都刷刷刷执行完了，也没什么特别的问题；但是有的task，就是会突然间，啪，报了一个OOM，JVM Out Of Memory，内存溢出了，task failed，task lost，resubmitting task。反复执行几次都到了某个task就是跑不通，最后就挂掉。

某个task就直接OOM，那么基本上也是因为数据倾斜了，task分配的数量实在是太大了！！！所以内存放不下，然后你的task每处理一条数据，还要创建大量的对象。内存爆掉了。

#### 定位原因与出现问题的位置

1、你在自己的程序里面找找，哪些地方用了会产生shuffle的算子，groupByKey、countByKey、reduceByKey、join

2、看log

log一般会报是在你的哪一行代码，导致了OOM异常；或者呢，看log，看看是执行到了第几个stage！！！

#### 解决方案

###### 第一个方案：聚合源数据

咱们现在，做一些聚合的操作，groupByKey、reduceByKey；groupByKey，说白了，就是拿到每个key对应的values；reduceByKey，说白了，就是对每个key对应的values执行一定的计算。

现在这些操作，比如groupByKey和reduceByKey，包括之前说的join。都是在spark作业中执行的。

spark作业的数据来源，通常是哪里呢？90%的情况下，数据来源都是hive表（hdfs，大数据分布式存储系统）。hdfs上存储的大数据。hive表，hive表中的数据，通常是怎么出来的呢？有了spark以后，hive比较适合做什么事情？hive就是适合做离线的，晚上凌晨跑的，ETL（extract transform load，数据的采集、清洗、导入），hive sql，去做这些事情，从而去形成一个完整的hive中的数据仓库；说白了，数据仓库，就是一堆表。

spark作业的源表，hive表，其实通常情况下来说，也是通过某些hive etl生成的。hive etl可能是晚上凌晨在那儿跑。今天跑昨天的数九。

数据倾斜，某个key对应的80万数据，某些key对应几百条，某些key对应几十条；现在，咱们直接在生成hive表的hive etl中，对数据进行聚合。比如按key来分组，将key对应的所有的values，全部用一种特殊的格式，拼接到一个字符串里面去，比如“key=sessionid, value: action\_seq=1|user\_id=1|search\_keyword=火锅|category\_id=001;action\_seq=2|user\_id=1|search\_keyword=涮肉|category\_id=001”。

对key进行group，在spark中，拿到key=sessionid，values<Iterable>；hive etl中，直接对key进行了聚合。那么也就意味着，每个key就只对应一条数据。在spark中，就不需要再去执行groupByKey+map这种操作了。直接对每个key对应的values字符串，map操作，进行你需要的操作即可。key,values串。

spark中，可能对这个操作，就不需要执行shffule操作了，也就根本不可能导致数据倾斜。或者是，对每个key在hive etl中进行聚合，对所有values聚合一下，不一定是拼接起来，可能是直接进行计算。reduceByKey，计算函数，应用在hive etl中，每个key的values。

有时候你可能没有办法对每个key，就聚合出来一条数据；

那么也可以做一个妥协；对每个key对应的数据，10万条；有好几个粒度，比如10万条里面包含了几个城市、几天、几个地区的数据，现在放粗粒度；直接就按照城市粒度，做一下聚合，几个城市，几天、几个地区粒度的数据，都给聚合起来。比如说

city\_id date area\_id

select ... from ... group by city\_id

尽量去聚合，减少每个key对应的数量，也许聚合到比较粗的粒度之后，原先有10万数据量的key，现在只有1万数据量。减轻数据倾斜的现象和问题。

###### 第二个方案：过滤导致倾斜的key

如果你能够接受某些数据，在spark作业中直接就摒弃掉，不使用。比如说，总共有100万个key。只有2个key，是数据量达到10万的。其他所有的key，对应的数量都是几十。

这个时候，你自己可以去取舍，如果业务和需求可以理解和接受的话，在你从hive表查询源数据的时候，直接在sql中用where条件，过滤掉某几个key。

那么这几个原先有大量数据，会导致数据倾斜的key，被过滤掉之后，那么在你的spark作业中，自然就不会发生数据倾斜了。

###### 第三个方案：提高shuffle操作的reduce并行度

将reduce task的数量，变多，就可以让每个reduce task分配到更少的数据量，这样的话，也许就可以缓解，或者甚至是基本解决掉数据倾斜的问题。

操作方法

很简单，主要给我们所有的shuffle算子，比如groupByKey、countByKey、reduceByKey。在调用的时候，传入进去一个参数。一个数字。那个数字，就代表了那个shuffle操作的reduce端的并行度。那么在进行shuffle操作的时候，就会对应着创建指定数量的reduce task。

###### 第四种方案：使用随机key实现双重聚合

使用场景

（1）groupByKey

（2）reduceByKey

比较适合使用这种方式；join，咱们通常不会这样来做，后面会讲三种，针对不同的join造成的数据倾斜的问题的解决方案。

第一轮聚合的时候，对key进行打散，将原先一样的key，变成不一样的key，相当于是将每个key分为多组；先针对多个组，进行key的局部聚合；接着，再去除掉每个key的前缀，然后对所有的key，进行全局的聚合。

对groupByKey、reduceByKey造成的数据倾斜，有比较好的效果。

示例

|  |
| --- |
| /\*\*  \* 使用随机key实现双重聚合  \*/    // /\*\*  // \* 第一步，给每个key打上一个随机数  // \*/  // JavaPairRDD<String, Long> mappedClickCategoryIdRDD = clickCategoryIdRDD.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，执行第一轮局部聚合  // \*/  // JavaPairRDD<String, Long> firstAggrRDD = mappedClickCategoryIdRDD.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，去除掉每个key的前缀  // \*/  // JavaPairRDD<Long, Long> restoredRDD = firstAggrRDD.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 categoryId = Long.valueOf(tuple.\_1.split("\_")[1]);  // return new Tuple2<Long, Long>(categoryId, tuple.\_2);  // }  //  // });  //  // /\*\*  // \* 第四步，最第二轮全局的聚合  // \*/  // JavaPairRDD<Long, Long> clickCategoryId2CountRDD = restoredRDD.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;  // }  //  // }); |

###### 第五种方案：将reduce join转换为map join

适合的场景

reduce join转换为map join，适合在什么样的情况下，可以来使用？

如果两个RDD要进行join，其中一个RDD是比较小的。一个RDD是100万数据，一个RDD是1万数据。（一个RDD是1亿数据，一个RDD是100万数据）

一个RDD必须是比较小的，broadcast出去那个小RDD的数据以后，就会在每个executor的block manager中都驻留一份。要确保你的内存足够存放那个小RDD中的数据

这种方式下，根本不会发生shuffle操作，肯定也不会发生数据倾斜；从根本上杜绝了join操作可能导致的数据倾斜的问题；

不适合的情况：

两个RDD都比较大，那么这个时候，你去将其中一个RDD做成broadcast，就很笨拙了。很可能导致内存不足。最终导致内存溢出，程序挂掉。

而且其中某些key（或者是某个key），还发生了数据倾斜；此时可以采用最后两种方式。

示例

|  |
| --- |
| /\*\*  \* reduce join转换为map join  \*/    // List<Tuple2<Long, Row>> userInfos = userid2InfoRDD.collect();  // final Broadcast<List<Tuple2<Long, Row>>> userInfosBroadcast = sc.broadcast(userInfos);  //  // JavaPairRDD<String, String> sessionid2FullAggrInfoRDD = userid2PartAggrInfoRDD.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 {  // // 得到用户信息map  // List<Tuple2<Long, Row>> userInfos = userInfosBroadcast.value();  //  // Map<Long, Row> userInfoMap = new HashMap<Long, Row>();  // for(Tuple2<Long, Row> userInfo : userInfos) {  // userInfoMap.put(userInfo.\_1, userInfo.\_2);  // }  //  // // 获取到当前用户对应的信息  // String partAggrInfo = tuple.\_2;  // Row userInfoRow = userInfoMap.get(tuple.\_1);  //  // String sessionid = StringUtils.getFieldFromConcatString(  // partAggrInfo, "\\|", Constants.FIELD\_SESSION\_ID);  //  // int age = userInfoRow.getInt(3);  // String professional = userInfoRow.getString(4);  // String city = userInfoRow.getString(5);  // String sex = userInfoRow.getString(6);  //  // String fullAggrInfo = partAggrInfo + "|"  // + Constants.FIELD\_AGE + "=" + age + "|"  // + Constants.FIELD\_PROFESSIONAL + "=" + professional + "|"  // + Constants.FIELD\_CITY + "=" + city + "|"  // + Constants.FIELD\_SEX + "=" + sex;  //  // return new Tuple2<String, String>(sessionid, fullAggrInfo);  // }  //  // }); |

###### 第六种方案：sample采样倾斜key进行两次join

将发生数据倾斜的key，单独拉出来，放到一个RDD中去；就用这个原本会倾斜的key RDD跟其他RDD，单独去join一下，这个时候，key对应的数据，可能就会分散到多个task中去进行join操作。就不至于说是，这个key跟之前其他的key混合在一个RDD中时，肯定是会导致一个key对应的所有数据，都到一个task中去，就会导致数据倾斜。

适合场景

优先对于join，肯定是希望能够采用上一讲讲的，reduce join转换map join。两个RDD数据都比较大，那么就不要那么搞了。

针对你的RDD的数据，你可以自己把它转换成一个中间表，或者是直接用countByKey()的方式，你可以看一下这个RDD各个key对应的数据量；此时如果你发现整个RDD就一个，或者少数几个key，是对应的数据量特别多；尽量建议，比如就是一个key对应的数据量特别多。

此时可以采用咱们的这种方案，单拉出来那个最多的key；单独进行join，尽可能地将key分散到各个task上去进行join操作。

不适合场景

如果一个RDD中，导致数据倾斜的key，特别多；那么此时，最好还是不要这样了；还是使用我们最后一个方案，终极的join数据倾斜的解决方案。

进一步优化

就是说，咱们单拉出来了，一个或者少数几个可能会产生数据倾斜的key，然后还可以进行更加优化的一个操作；

对于那个key，从另外一个要join的表中，也过滤出来一份数据，比如可能就只有一条数据。userid2infoRDD，一个userid key，就对应一条数据。

然后呢，采取对那个只有一条数据的RDD，进行flatMap操作，打上100个随机数，作为前缀，返回100条数据。

单独拉出来的可能产生数据倾斜的RDD，给每一条数据，都打上一个100以内的随机数，作为前缀。

再去进行join，是不是性能就更好了。肯定可以将数据进行打散，去进行join。join完以后，可以执行map操作，去将之前打上的随机数，给去掉，然后再和另外一个普通RDD join以后的结果，进行union操作。

举例

|  |
| --- |
| /\*\*  \* sample采样倾斜key单独进行join  \*/    // JavaPairRDD<Long, String> sampledRDD = userid2PartAggrInfoRDD.sample(false, 0.1, 9);  //  // 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> computedSampledRDD = 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 = computedSampledRDD.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;  //  // JavaPairRDD<Long, String> skewedRDD = userid2PartAggrInfoRDD.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);  // }  //  // });  //  // JavaPairRDD<Long, String> commonRDD = userid2PartAggrInfoRDD.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);  // }  //  // });  //  // JavaPairRDD<String, Row> skewedUserid2infoRDD = userid2InfoRDD.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++) {  // int prefix = random.nextInt(100);  // list.add(new Tuple2<String, Row>(prefix + "\_" + tuple.\_1, tuple.\_2));  // }  //  // return list;  // }  //  // });  //  // 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 userid = Long.valueOf(tuple.\_1.split("\_")[1]);  // return new Tuple2<Long, Tuple2<String, Row>>(userid, tuple.\_2);  // }  //  // });  //  // JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD2 = commonRDD.join(userid2InfoRDD);  //  // JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD = joinedRDD1.union(joinedRDD2);  //  // JavaPairRDD<String, String> sessionid2FullAggrInfoRDD = joinedRDD.mapToPair(  //  // new PairFunction<Tuple2<Long,Tuple2<String,Row>>, String, String>() {  //  // private static final long serialVersionUID = 1L;  //  // @Override  // public Tuple2<String, String> call(  // Tuple2<Long, Tuple2<String, Row>> tuple)  // throws Exception {  // String partAggrInfo = tuple.\_2.\_1;  // Row userInfoRow = tuple.\_2.\_2;  //  // String sessionid = StringUtils.getFieldFromConcatString(  // partAggrInfo, "\\|", Constants.FIELD\_SESSION\_ID);  //  // int age = userInfoRow.getInt(3);  // String professional = userInfoRow.getString(4);  // String city = userInfoRow.getString(5);  // String sex = userInfoRow.getString(6);  //  // String fullAggrInfo = partAggrInfo + "|"  // + Constants.FIELD\_AGE + "=" + age + "|"  // + Constants.FIELD\_PROFESSIONAL + "=" + professional + "|"  // + Constants.FIELD\_CITY + "=" + city + "|"  // + Constants.FIELD\_SEX + "=" + sex;  //  // return new Tuple2<String, String>(sessionid, fullAggrInfo);  // }  //  // }); |

###### 第七种方案：使用随机数以及扩容表进行join

步骤

1、选择一个RDD，要用flatMap，进行扩容，将每条数据，映射为多条数据，每个映射出来的数据，都带了一个n以内的随机数，通常来说，会选择10。

2、将另外一个RDD，做普通的map映射操作，每条数据，都打上一个10以内的随机数。

3、最后，将两个处理后的RDD，进行join操作。

局限性

1、因为你的两个RDD都很大，所以你没有办法去将某一个RDD扩的特别大，一般咱们就是10倍。

2、如果就是10倍的话，那么数据倾斜问题，的确是只能说是缓解和减轻，不能说彻底解决。

举例

举例

|  |
| --- |
| /\*\*  \* 使用随机数和扩容表进行join  \*/    // JavaPairRDD<String, Row> expandedRDD = userid2InfoRDD.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 < 10; i++) {  // list.add(new Tuple2<String, Row>(0 + "\_" + tuple.\_1, tuple.\_2));  // }  //  // return list;  // }  //  // });  //  // JavaPairRDD<String, String> mappedRDD = userid2PartAggrInfoRDD.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(10);  // return new Tuple2<String, String>(prefix + "\_" + tuple.\_1, tuple.\_2);  // }  //  // });  //  // JavaPairRDD<String, Tuple2<String, Row>> joinedRDD = mappedRDD.join(expandedRDD);  //  // JavaPairRDD<String, String> finalRDD = joinedRDD.mapToPair(  //  // new PairFunction<Tuple2<String,Tuple2<String,Row>>, String, String>() {  //  // private static final long serialVersionUID = 1L;  //  // @Override  // public Tuple2<String, String> call(  // Tuple2<String, Tuple2<String, Row>> tuple)  // throws Exception {  // String partAggrInfo = tuple.\_2.\_1;  // Row userInfoRow = tuple.\_2.\_2;  //  // String sessionid = StringUtils.getFieldFromConcatString(  // partAggrInfo, "\\|", Constants.FIELD\_SESSION\_ID);  //  // int age = userInfoRow.getInt(3);  // String professional = userInfoRow.getString(4);  // String city = userInfoRow.getString(5);  // String sex = userInfoRow.getString(6);  //  // String fullAggrInfo = partAggrInfo + "|"  // + Constants.FIELD\_AGE + "=" + age + "|"  // + Constants.FIELD\_PROFESSIONAL + "=" + professional + "|"  // + Constants.FIELD\_CITY + "=" + city + "|"  // + Constants.FIELD\_SEX + "=" + sex;  //  // return new Tuple2<String, String>(sessionid, fullAggrInfo);  // }  //  // }); |

## SparkSQL性能调优

### 聚合源数据

Spark Core和Spark SQL没有任何的区别

### 过滤导致倾斜的key

在sql中用where条件

### 提高shuffle并行度

groupByKey(1000)，spark.sql.shuffle.partitions（默认是200）

### 双重group by

改写SQL，两次group by

#### 举例

|  |
| --- |
| /\*\*  \* 双重group by  \*/    // String \_sql =  // "SELECT "  // + "product\_id\_area,"  // + "count(click\_count) click\_count,"  // + "group\_concat\_distinct(city\_infos) city\_infos "  // + "FROM ( "  // + "SELECT "  // + "remove\_random\_prefix(product\_id\_area) product\_id\_area,"  // + "click\_count,"  // + "city\_infos "  // + "FROM ( "  // + "SELECT "  // + "product\_id\_area,"  // + "count(\*) click\_count,"  // + "group\_concat\_distinct(concat\_long\_string(city\_id,city\_name,':')) city\_infos "  // + "FROM ( "  // + "SELECT "  // + "random\_prefix(concat\_long\_string(product\_id,area,':'), 10) product\_id\_area,"  // + "city\_id,"  // + "city\_name "  // + "FROM tmp\_click\_product\_basic "  // + ") t1 "  // + "GROUP BY product\_id\_area "  // + ") t2 "  // + ") t3 "  // + "GROUP BY product\_id\_area "; |

### reduce join转换为map join

spark.sql.autoBroadcastJoinThreshold（默认是10485760 ）

你可以自己将表做成RDD，自己手动去实现map join

Spark SQL内置的map join，默认是如果有一个小表，是在10M以内，默认就会将该表进行broadcast，然后执行map join；调节这个阈值，比如调节到20M、50M、甚至1G。20 971 520

### 采样倾斜key并单独进行join

纯Spark Core的一种方式，sample、filter等算子

### 随机key与扩容表

Spark SQL+Spark Core

举例

|  |
| --- |
| // JavaRDD<Row> rdd = sqlContext.sql("select \* from product\_info").javaRDD();  // JavaRDD<Row> flattedRDD = rdd.flatMap(new FlatMapFunction<Row, Row>() {  //  // private static final long serialVersionUID = 1L;  //  // @Override  // public Iterable<Row> call(Row row) throws Exception {  // List<Row> list = new ArrayList<Row>();  //  // for(int i = 0; i < 10; i ++) {  // long productid = row.getLong(0);  // String \_productid = i + "\_" + productid;  //  // Row \_row = RowFactory.create(\_productid, row.get(1), row.get(2));  // list.add(\_row);  // }  //  // return list;  // }  //  // });  //  // StructType \_schema = DataTypes.createStructType(Arrays.asList(  // DataTypes.createStructField("product\_id", DataTypes.StringType, true),  // DataTypes.createStructField("product\_name", DataTypes.StringType, true),  // DataTypes.createStructField("product\_status", DataTypes.StringType, true)));  //  // DataFrame \_df = sqlContext.createDataFrame(flattedRDD, \_schema);  // \_df.registerTempTable("tmp\_product\_info");  //  // String \_sql =  // "SELECT "  // + "tapcc.area,"  // + "remove\_random\_prefix(tapcc.product\_id) product\_id,"  // + "tapcc.click\_count,"  // + "tapcc.city\_infos,"  // + "pi.product\_name,"  // + "if(get\_json\_object(pi.extend\_info,'product\_status')=0,'自营商品','第三方商品') product\_status "  // + "FROM ("  // + "SELECT "  // + "area,"  // + "random\_prefix(product\_id, 10) product\_id,"  // + "click\_count,"  // + "city\_infos "  // + "FROM tmp\_area\_product\_click\_count "  // + ") tapcc "  // + "JOIN tmp\_product\_info pi ON tapcc.product\_id=pi.product\_id "; |

## SparkStreaming性能调优

### HA高可用性

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

#### 方案

通过一整套方案（3个步骤），开启和实现实时计算程序的HA高可用性，保证一些关键数据都有其冗余副本，不至于因为节点挂掉或者其他原因导致数据丢失。

1、updateStateByKey、window等有状态的操作，自动进行checkpoint，必须设置checkpoint目录

checkpoint目录：容错的文件系统的目录，比如说，常用的是HDFS

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

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

2、Driver高可用性

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

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

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

JavaStreamingContextFactory contextFactory = new JavaStreamingContextFactory() {

@Override

public JavaStreamingContext create() {

JavaStreamingContext jssc = new JavaStreamingContext(...);

JavaDStream<String> lines = jssc.socketTextStream(...);

jssc.checkpoint(checkpointDirectory);

return jssc;

}

};

JavaStreamingContext context = JavaStreamingContext.getOrCreate(checkpointDirectory, contextFactory);

context.start();

context.awaitTermination();

spark-submit

--deploy-mode cluster

--supervise

3、实现RDD高可用性：启动WAL预写日志机制

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

receiver主要接收到数据，那么就会立即将数据写入一份到容错文件系统（比如hdfs）上的checkpoint目录中的，一份磁盘文件中去；作为数据的冗余副本。

无论你的程序怎么挂掉，或者是数据丢失，那么数据都不肯能会永久性的丢失；因为肯定有副本。

WAL（Write-Ahead Log）预写日志机制

spark.streaming.receiver.writeAheadLog.enable true

### 性能调优

#### 并行化数据接收

处理多个topic的数据时比较有效

int numStreams = 5;

List<JavaPairDStream<String, String>> kafkaStreams = new ArrayList<JavaPairDStream<String, String>>(numStreams);

for (int i = 0; i < numStreams; i++) {

kafkaStreams.add(KafkaUtils.createStream(...));

}

JavaPairDStream<String, String> unifiedStream = streamingContext.union(kafkaStreams.get(0), kafkaStreams.subList(1, kafkaStreams.size()));

unifiedStream.print();

#### 增加block数量

Spark.streaming.blockInterval：增加block数量，增加每个batch rdd的partition数量，增加处理并行度

receiver从数据源源源不断地获取到数据；首先是会按照block interval，将指定时间间隔的数据，收集为一个block；默认时间是200ms，官方推荐不要小于50ms；接着呢，会将指定batch interval时间间隔内的block，合并为一个batch；创建为一个rdd，然后启动一个job，去处理这个batch rdd中的数据

batch rdd，它的partition数量是多少呢？一个batch有多少个block，就有多少个partition；就意味着并行度是多少；就意味着每个batch rdd有多少个task会并行计算和处理。

当然是希望可以比默认的task数量和并行度再多一些了；可以手动调节block interval；减少block interval；每个batch可以包含更多的block；有更多的partition；也就有更多的task并行处理每个batch rdd。

定死了，初始的rdd过来，直接就是固定的partition数量了

#### 重分区

inputStream.repartition(<number of partitions>)：重分区，增加每个batch rdd的partition数量

有些时候，希望对某些dstream中的rdd进行定制化的分区

对dstream中的rdd进行重分区，去重分区成指定数量的分区，这样也可以提高指定dstream的rdd的计算并行度

#### 调节并行度

spark.default.parallelism

reduceByKey(numPartitions)

#### 使用Kryo序列化机制

spark streaming，也是有不少序列化的场景的

提高序列化task发送到executor上执行的性能，如果task很多的时候，task序列化和反序列化的性能开销也比较可观

默认输入数据的存储级别是StorageLevel.MEMORY\_AND\_DISK\_SER\_2，receiver接收到数据，默认就会进行持久化操作；首先序列化数据，存储到内存中；如果内存资源不够大，那么就写入磁盘；而且，还会写一份冗余副本到其他executor的block manager中，进行数据冗余。

#### batch interval：每个的处理时间必须小于batch interval

实际上你的spark streaming跑起来以后，其实都是可以在spark ui上观察它的运行情况的；可以看到batch的处理时间；

如果发现batch的处理时间大于batch interval，就必须调节batch interval

尽量不要让batch处理时间大于batch interval

比如你的batch每隔5秒生成一次；你的batch处理时间要达到6秒；就会出现，batch在你的内存中日积月累，一直囤积着，没法及时计算掉，释放内存空间；而且对内存空间的占用越来越大，那么此时会导致内存空间快速消耗

如果发现batch处理时间比batch interval要大，就尽量将batch interval调节大一些