### 开发调优(1)

### 对多次使用的RDD进行持久化

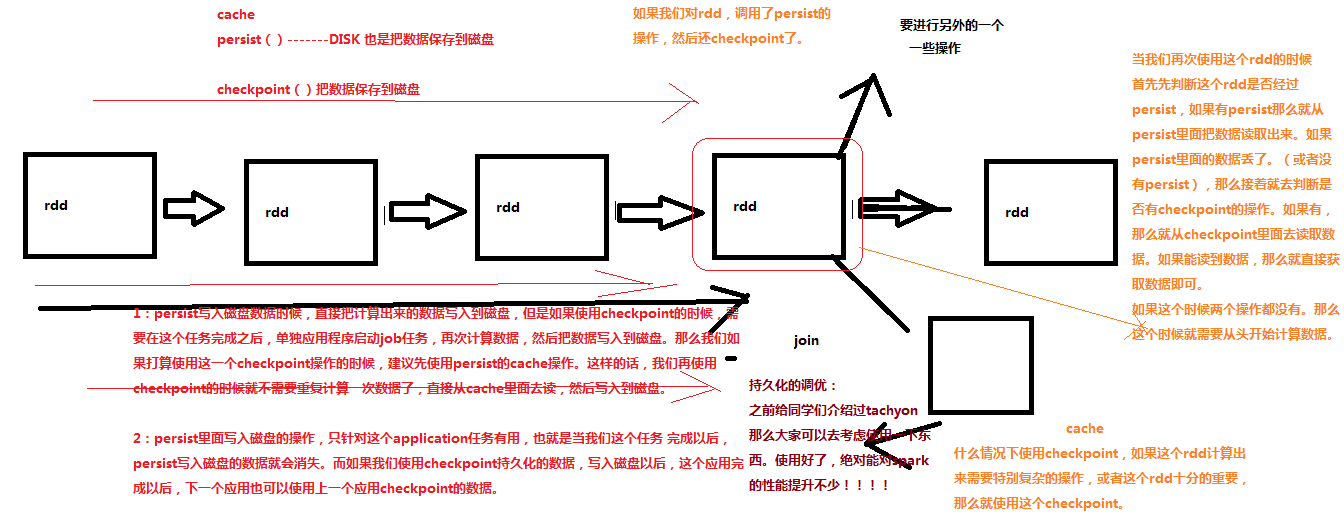
         Spark中对于一个RDD执行多次算子的默认原理是这样的：每次你对一个RDD执行一个算子操作时，都会重新从源头处计算一遍，计算出那个RDD来，然后再对这个RDD执行你的算子操作。这种方式的性能是很差的。

      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，而不会从源头处重新计算一遍这个RDD，再执行算子操作。**

cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法，而是等到 job 结束后另外启动专门的 job 去完成 checkpoint 。 也就是说需要 checkpoint 的 RDD 会被计算两次。因此，在使用 rdd.checkpoint() 的时候，建议加上 rdd.cache()， 这样第二次运行的 job 就不用再去计算该 rdd 了，直接读取 cache 写磁盘。

#### **cache 与 checkpoint 的区别？**

rdd.persist(StorageLevel.DISK\_ONLY) 与 checkpoint 也有区别。前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程 CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个 blockManager 使用的 local 文件夹被删除）。而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉，是一直存在的，也就是说可以被下一个 driver program 使用，而 cached RDD 不能被其他 dirver program 使用。



# 开发调优(2)

### 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。

### 案例演示

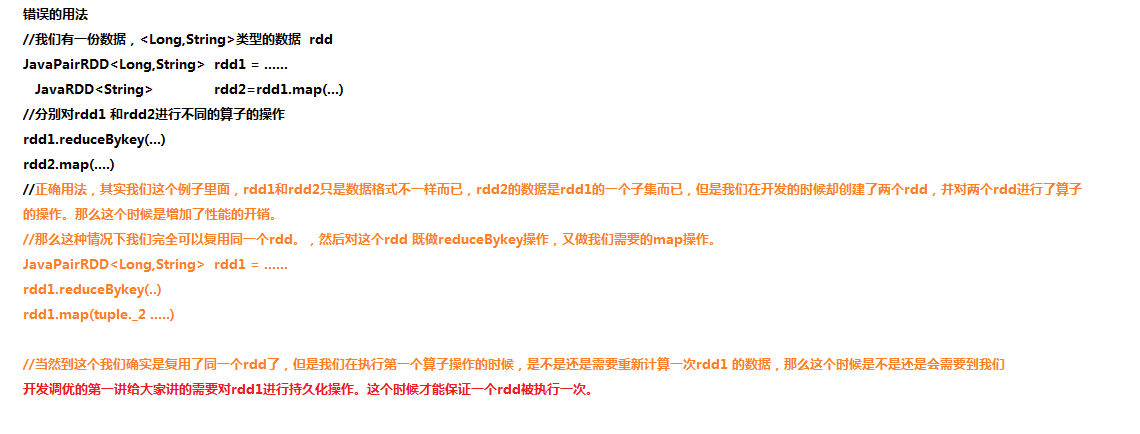


# 开发调优(3)

### 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rdd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。**

### 案例演示



# 开发调优(4)

### 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。**

spark中会导致shuffle操作的有以下几种算子

1、repartition类的操作：比如repartition、repartitionAndSortWithinPartitions、coalesce等

2、byKey类的操作：比如reduceByKey、groupByKey、sortByKey等

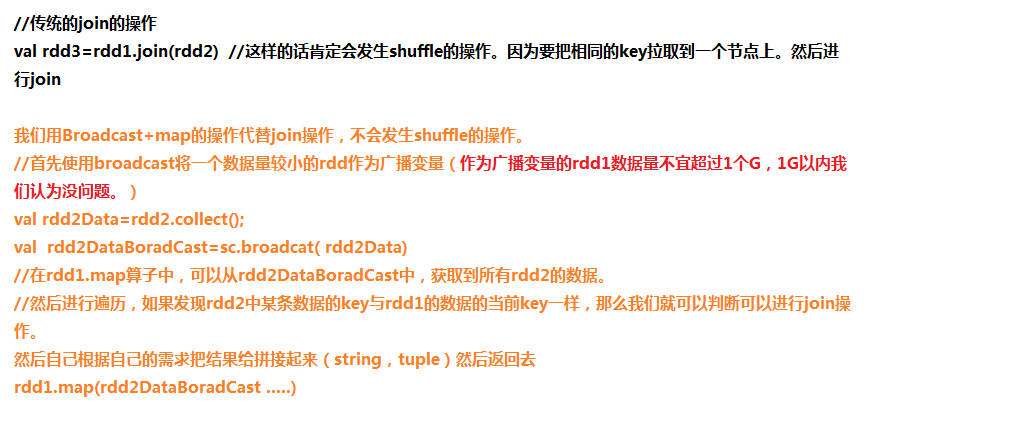
3、join类的操作：比如join、cogroup等

重分区: 一般会shuffle，因为需要在整个集群中，对之前所有的分区的数据进行随机，均匀的打乱，然后把数据放入下游新的指定数量的分区内

byKey类的操作：因为你要对一个key，进行聚合操作，那么肯定要保证集群中，所有节点上的，相同的key，一定是到同一个节点上进行处理

join类的操作：两个rdd进行join，就必须将相同join key的数据，shuffle到同一个节点上，然后进行相同key的两个rdd数据的笛卡尔乘积

### 案例演示



### 开发调优(5)

### 使用map-side预聚合的shuffle操作

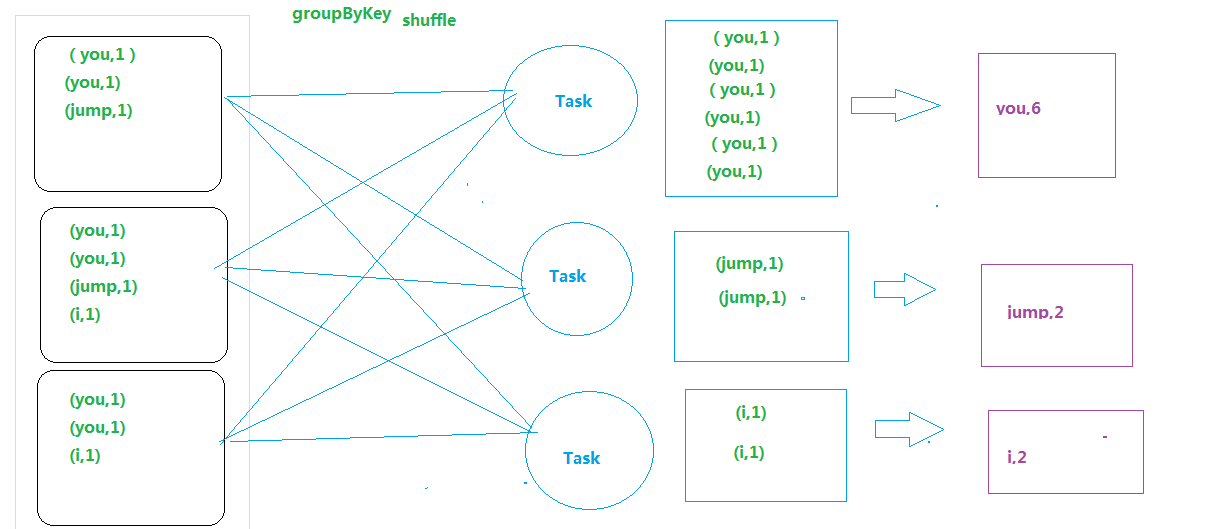
如果因为业务需要，一定要使用shuffle操作，无法用map类的算子来替代，那么尽量使用可以map-side预聚合的算子。

      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combin。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。**

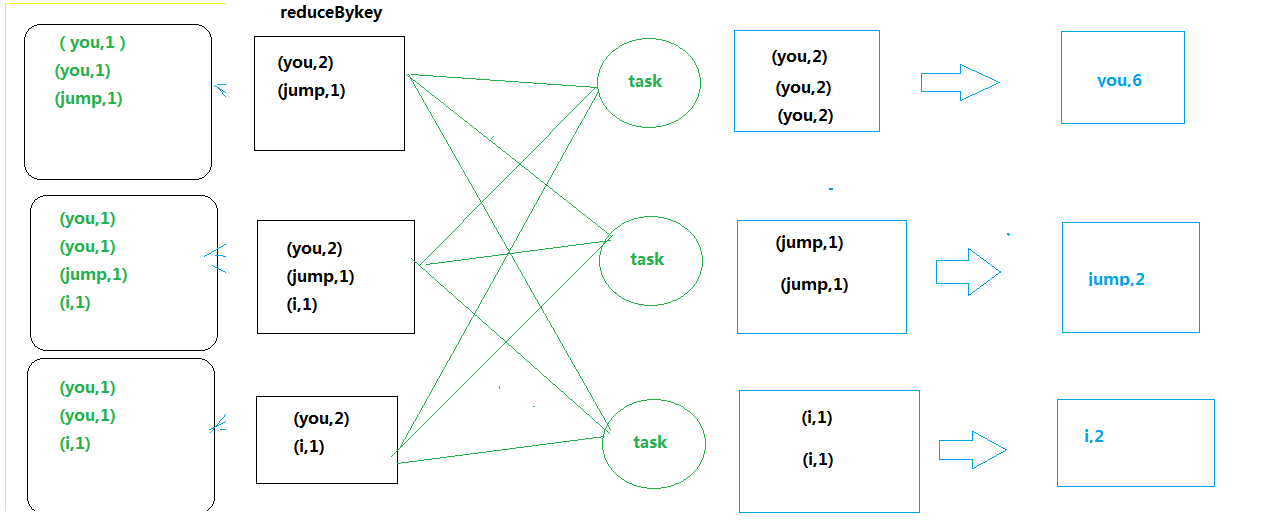
### 原理剖析

groupByKey原理图

（需要拉取全量的数据找key，消耗非常大）



reduceByKey原理图



# 开发调优(6)

### 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：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**](http://lib.csdn.net/base/14)，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**](http://lib.csdn.net/base/14)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。

# 开发调优(7)

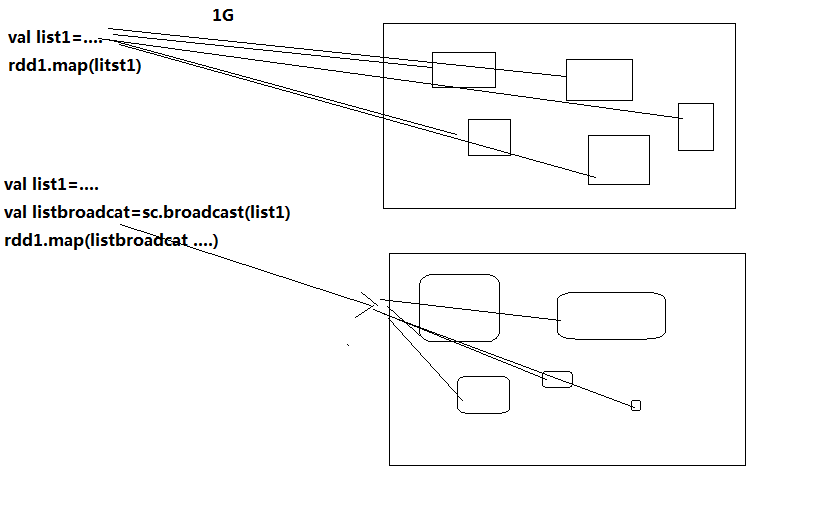
### 广播大变量

  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。

### 案例演示



# 开发调优(8)

### 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**](http://lib.csdn.net/base/17)**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。

### 案例演示

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

Scala版本：

val conf = new SparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Array(classOf[Counter] ))

val sc = new SparkContext(conf)

Java版本：

SparkConf conf = new SparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Counter.class)

JavaSparkContext sc = new JavaSparkContext(conf)

如果注册的要序列化的自定义的类型，本身特别大，比如包含了超过100个field。那么就会导致要序列化的对象过大。此时就需要对Kryo本身进行优化。因为Kryo内部的缓存可能不够存放那么大的class对象。此时就需要调用SparkConf.set()方法，设置spark.kryoserializer.buffer.mb参数的值，将其调大。

默认情况下它的值是2，就是说最大能缓存2M的对象，然后进行序列化。可以在必要时将其调大。比如设置为10。

# 开发调优(9)

### 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**](http://lib.csdn.net/base/31)，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。

建议：我们在实际的开发中，要做到如上所述，其实不容易。我们要考虑到代码可维护性，如果一个代码里面，完全没有抽象，全部都是字符串的拼接，对于后面的代码维护和修改难度很大。

所以我们应该考虑在合适的时候采取考虑这样的优化方式，因为我们首先得考虑代码的维护性。

# 数据本地化

数据本地化的背景:

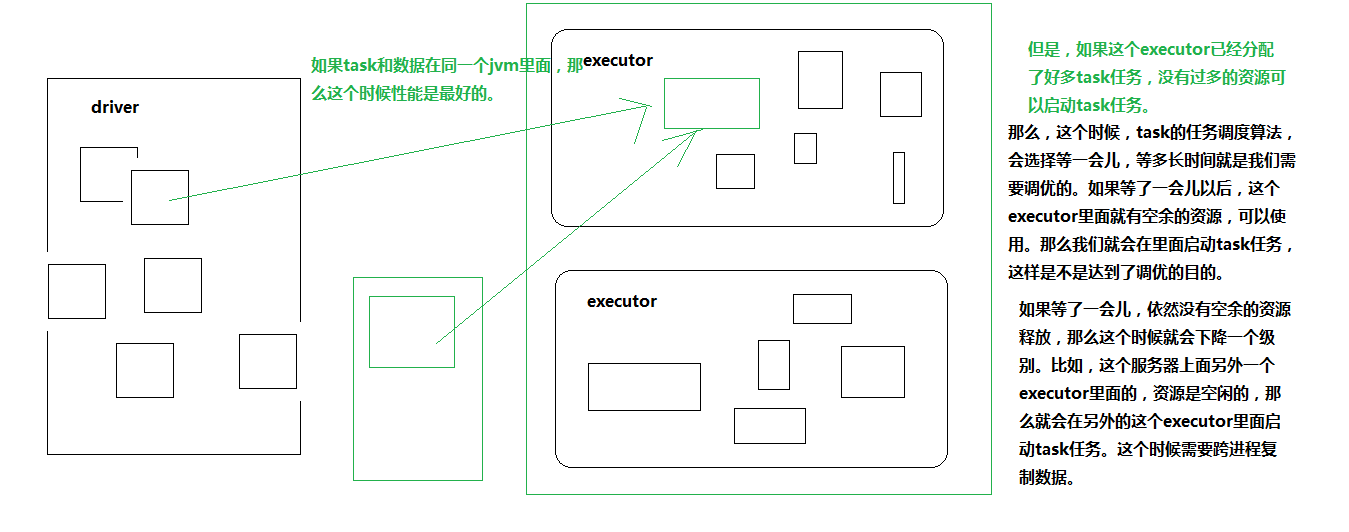
数据本地化对于Spark Job性能有着巨大的影响。如果数据以及要计算它的代码是在一起的，那么性能当然会非常高。但是，如果数据和计算它的代码是分开的，那么其中之一必须到另外一方的机器上。通常来说，移动代码到其他节点，会比移动数据到代码所在的节点上去，速度要快得多，因为代码比较小。Spark也正是基于这个数据本地化的原则来构建task调度算法的。

数据本地化，指的是，数据离计算它的代码有多近。基于数据距离代码的距离，有几种数据本地化级别：  
1、PROCESS\_LOCAL：数据和计算它的代码在同一个JVM进程中。  
2、NODE\_LOCAL：数据和计算它的代码在一个节点上，但是不在一个进程中，比如在不同的executor进程中，或者是数据在HDFS文件的block中。  
3、NO\_PREF：从任何地方访问数据速度都是一样，不关心数据的位置。  
4、RACK\_LOCAL：数据和计算它的代码在一个机架上。  
5、ANY：数据可能在任意地方，比如其他网络环境内，或者其他机架上。

配置如下参数，默认3S（偏小，50s或一分钟，尽量等待同一个JVM把资源释放出来）

（节点、进程、机架）可相对的去调





### 数据倾斜原理

### 调优概述

      有的时候，**我们可能会遇到**[**大数据**](http://lib.csdn.net/base/20)**计算中一个最棘手的问题——数据倾斜，此时Spark作业的性能会比期望差很多。数据倾斜调优，就是使用各种技术方案解决不同类型的数据倾斜问题，以保证Spark作业的性能。**

### 数据倾斜发生时的现象

绝大多数task执行得都非常快，但个别task执行极慢。比如，总共有1000个task，997个task都在1分钟之内执行完了，但是剩余两三个task却要一两个小时。这种情况很常见。

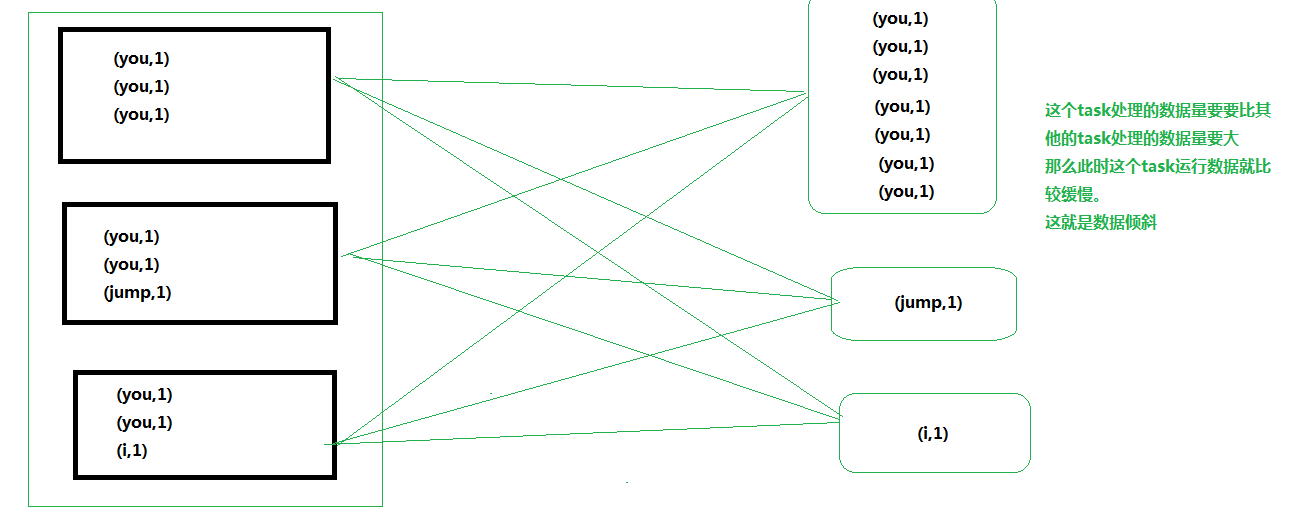
原本能够正常执行的Spark作业，某天突然报出OOM（内存溢出）异常，观察异常栈，是我们写的业务代码造成的。这种情况比较少见。

### 数据倾斜发生的原理

      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。

举例



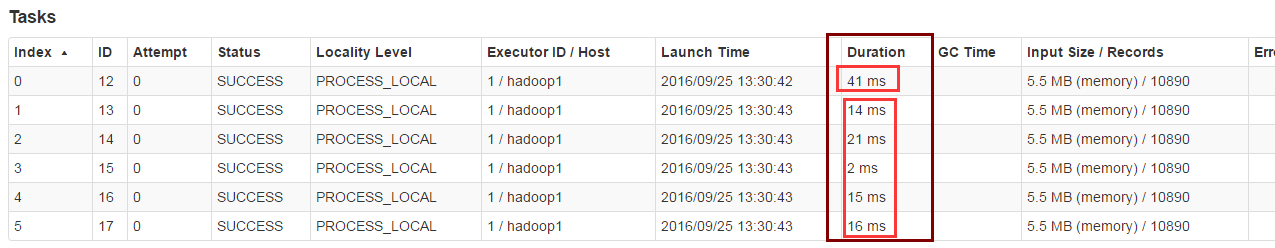
### 如何定位发生数据倾斜的代码

1） 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。

2）通过观察spark UI的界面，定位数据倾斜发生在第几个stage中。

      如果是用yarn-client模式提交，那么本地是直接可以看到log的，可以在log中找到当前运行到了第几个stage；如果是用yarn-cluster模式提交，则可以通过Spark Web UI来查看当前运行到了第几个stage。此外，无论是使用yarn-client模式还是yarn-cluster模式，我们都可以在Spark Web UI上深入看一下当前这个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。

1. 根据之前学的stage的划分算法定位到极有可能发生数据倾斜的代码



### 查看导致数据倾斜的key的分布情况

1. 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。
2. 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。

val sampledPairs = pairs.sample(false, 0.1)

val sampledWordCounts = sampledPairs.countByKey()

sampledWordCounts.foreach(println(\_))

# 数据倾斜解决方案一

### 解决方案一：使用Hive ETL预处理数据

**方案适用场景**：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**](http://lib.csdn.net/base/17)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。

# 数据倾斜解决方案二

### 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。

# 数据倾斜解决方案三

### 提高shuffle操作的并行度

**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。

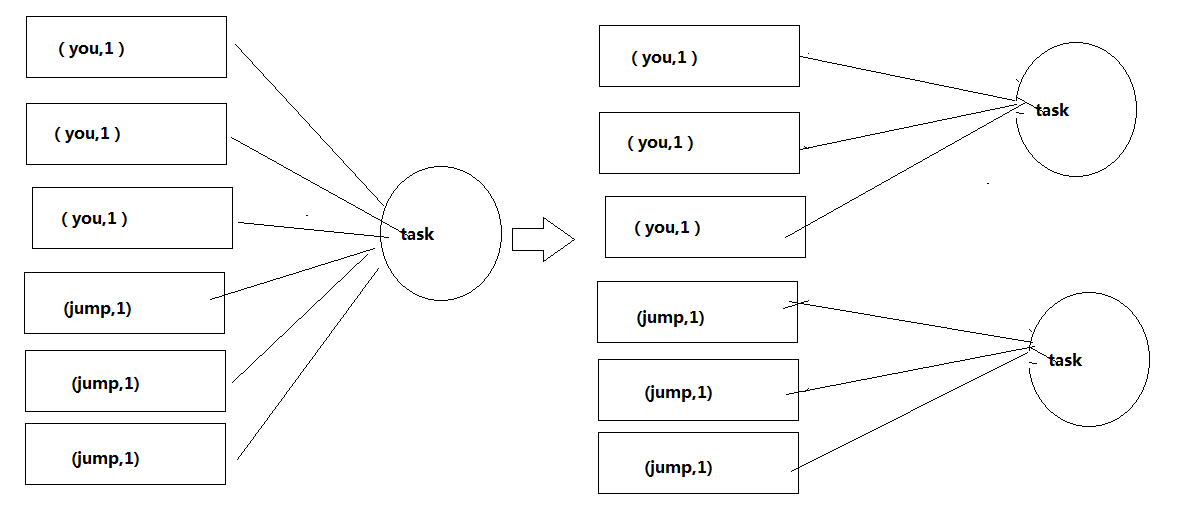
**方案实现原理**：增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了。

**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，尝试去用嘴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。

原理演示



# 数据倾斜解决方案四

### 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。

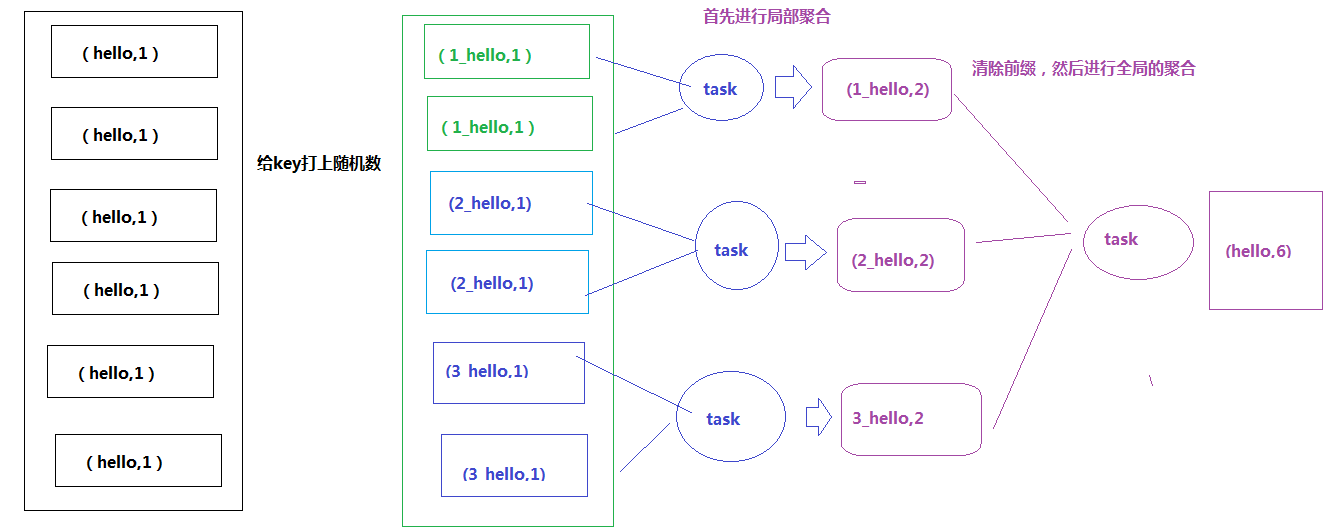
**方案实现思路**：这个方案的核心实现思路就是进行两阶段聚合。第一次是局部聚合，先给每个key都打上一个随机数，比如10以内的随机数，此时原先一样的key就变成不一样的了，比如(hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成(1\_hello, 1) (1\_hello, 1) (2\_hello, 1) (2\_hello, 1)。接着对打上随机数后的数据，执行reduceByKey等聚合操作，进行局部聚合，那么局部聚合结果，就会变成了(1\_hello, 2) (2\_hello, 2)。然后将各个key的前缀给去掉，就会变成(hello,2)(hello,2)，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

### 原理演示

s

### 代码演示

分别使用java 和scala的代码演示一下例子。

首先用java代码演示一下例子：单词计数的一个例子。

package com.xtwy.sparkcore

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

object AggWordCount {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setMaster("local").setAppName("AggWordCount")

val sc=new SparkContext(conf);

val list=Array("you jump","jump jump","jump jump","jump jump","jump jump","jump jump","jump jump","jump jump")

val listRDD=sc.parallelize(list, 1);

listRDD.flatMap { line => line.split("\t") }

.map { word => (word,1) }

.map(word =>{

val prefix= (new util.Random).nextInt(4) //4以内

(prefix+"\_"+word.\_1,word.\_2)

})

.reduceByKey(\_+\_)

.map(word =>{

val key=word.\_1.split("\_")(1)

(key,word.\_2)

})

.reduceByKey(\_+\_)

.foreach(result => println(result.\_1 + " : "+result.\_2))

}

}

# 数据倾斜解决方案五

### 将reduce join转为map 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。

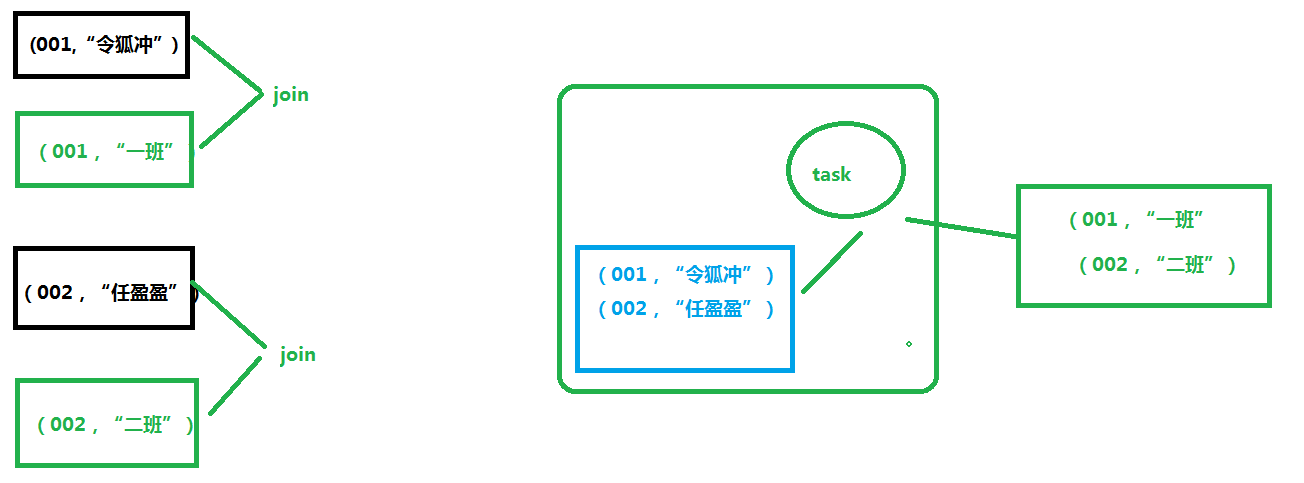
**方案实现思路**：不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作，彻底避免数据倾斜的发生和出现。将较小RDD中的数据直接通过collect算子拉取到Driver端的内存中来，然后对其创建一个Broadcast变量；接着对另外一个RDD执行map类算子，在算子函数内，从Broadcast变量中获取较小RDD的全量数据，与当前RDD的每一条数据按照连接key进行比对，如果连接key相同的话，那么就将两个RDD的数据用你需要的方式连接起来。

**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。

**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。

### 案例演示



package com.xtwy.sparkcore

import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

object MapJoinTest {

def main(args: Array[String]): Unit = {

val conf=new SparkConf().setMaster("local").setAppName("MapJoinTest")

val sc=new SparkContext(conf);

val lista=Array(

Tuple2[String,String]("001","令狐冲"),

Tuple2[String,String]("002","任盈盈")

)

val listb=Array(

Tuple2[String,String]("001","一班"),

Tuple2[String,String]("002","二班")

)

val listaRDD= sc.parallelize(lista);

val listbRDD= sc.parallelize(listb);

val listadata=listaRDD.collect();

val listaboradcast=sc.broadcast(listadata);

listbRDD.map( tuple =>{

import scala.collection.mutable.Map;

val rdd1map:Map[String,String]=Map();

for(t <- listaboradcast.value){

rdd1map+=(t.\_1 -> t.\_2);

}

//get rddb key value

val key=tuple.\_1

val value=tuple.\_2

val rdd1value= rdd1map.get(key).get

//return

(key,Tuple2(value,rdd1value))

}).foreach(result => println(result.\_1 + " "+ result.\_2.\_1 + " "+result.\_2.\_2))

}

}

# 数据倾斜解决方案六

### 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。**

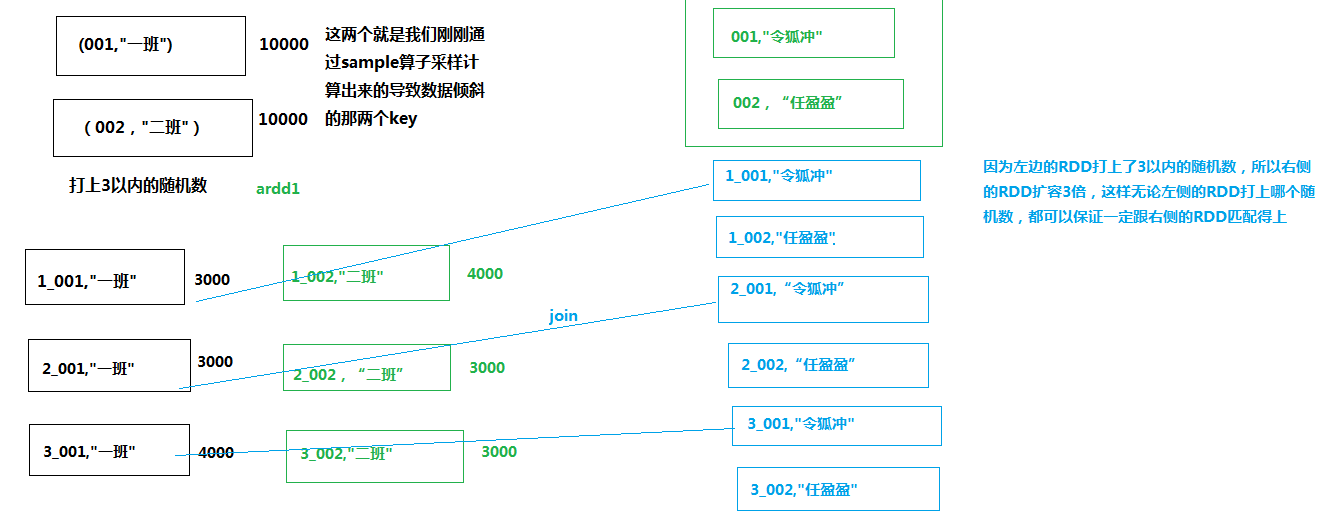
**方案实现思路**：

* 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。

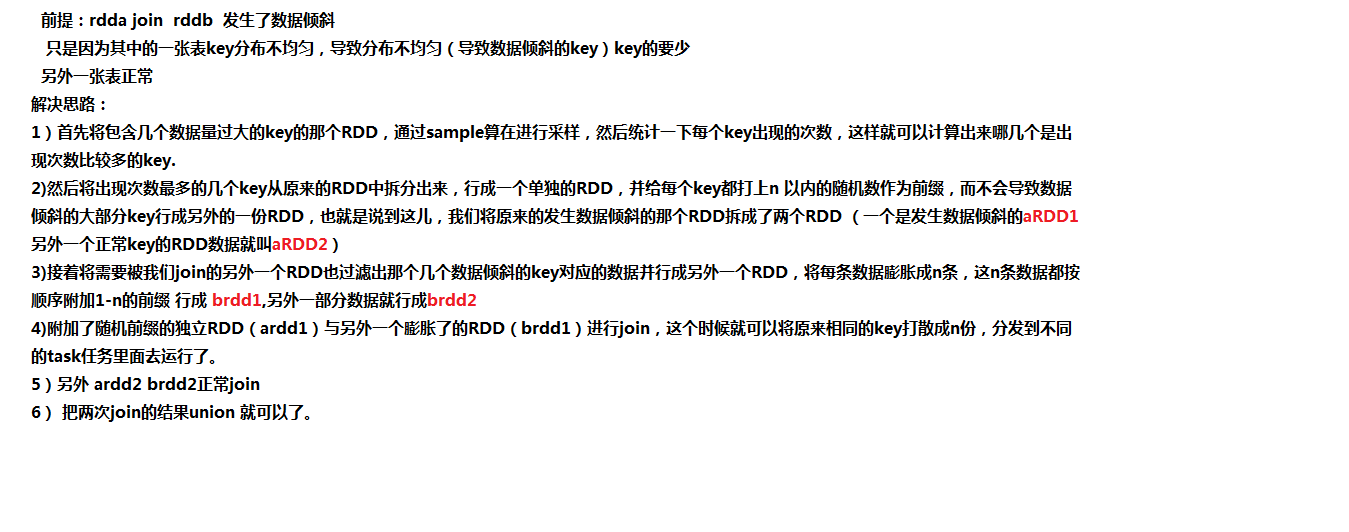
**方案优点**：**对于join导致的数据倾斜，如果只是某几个key导致了倾斜，采用该方式可以用最有效的方式打散key进行join。**而且只需要针对少数倾斜key对应的数据进行扩容n倍，不需要对全量数据进行扩容。避免了占用过多内存。

**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。**



### 原理剖析

### 案例演示



# 数据倾斜解决方案七

### 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（倾斜的）的每条数据都打上一个n以内的随机前缀。（因为它多呀，所以不扩容它）

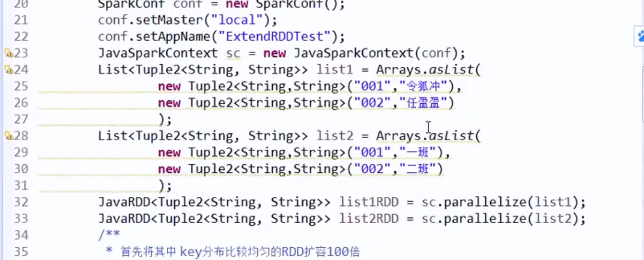
* 同时对另外一个正常的RDD进行扩容，将每条数据都扩容成n条数据，扩容出来的每条数据都依次打上一个0~n的前缀。
* 最后将两个处理后的RDD进行join即可。

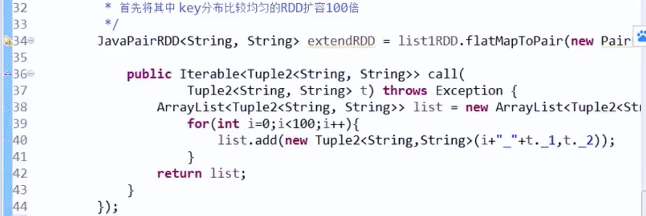
**方案实现原理**：将原先一样的key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同key”分散到多个task中去处理，而不是让一个task处理大量的相同key。该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；而这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。

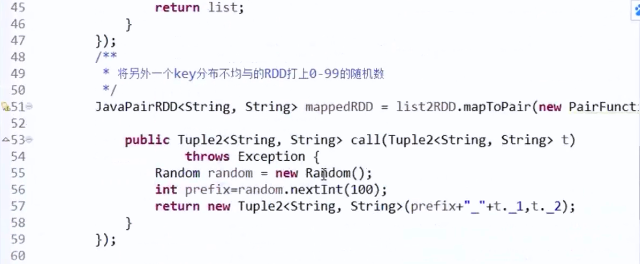
**方案优点**：对join类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错。

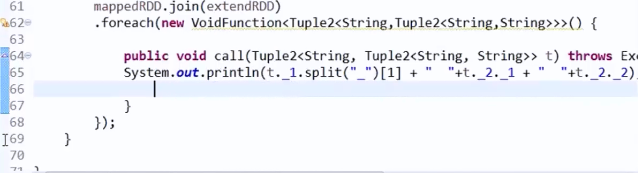
**方案缺点：该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个RDD进行扩容，对内存资源要求很高。**

      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。



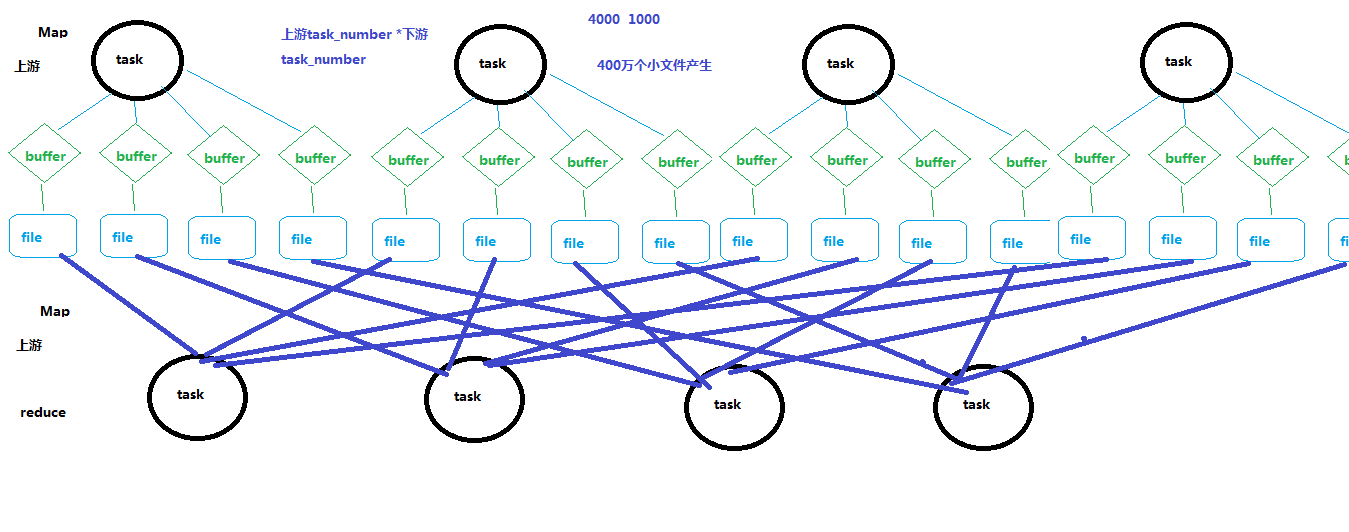




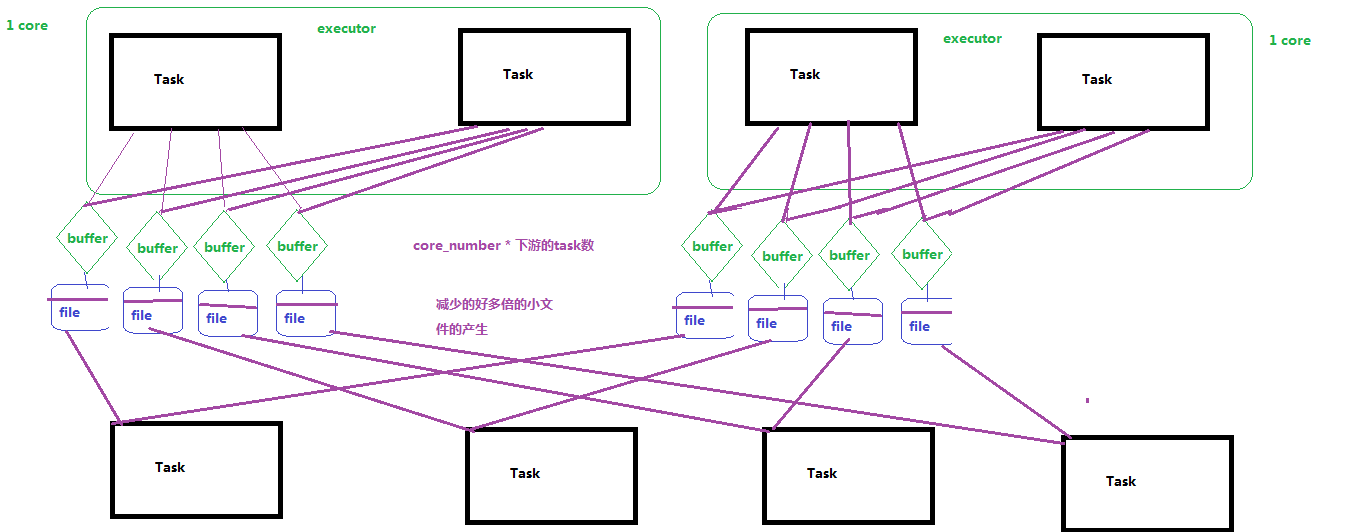


# shuffle调优

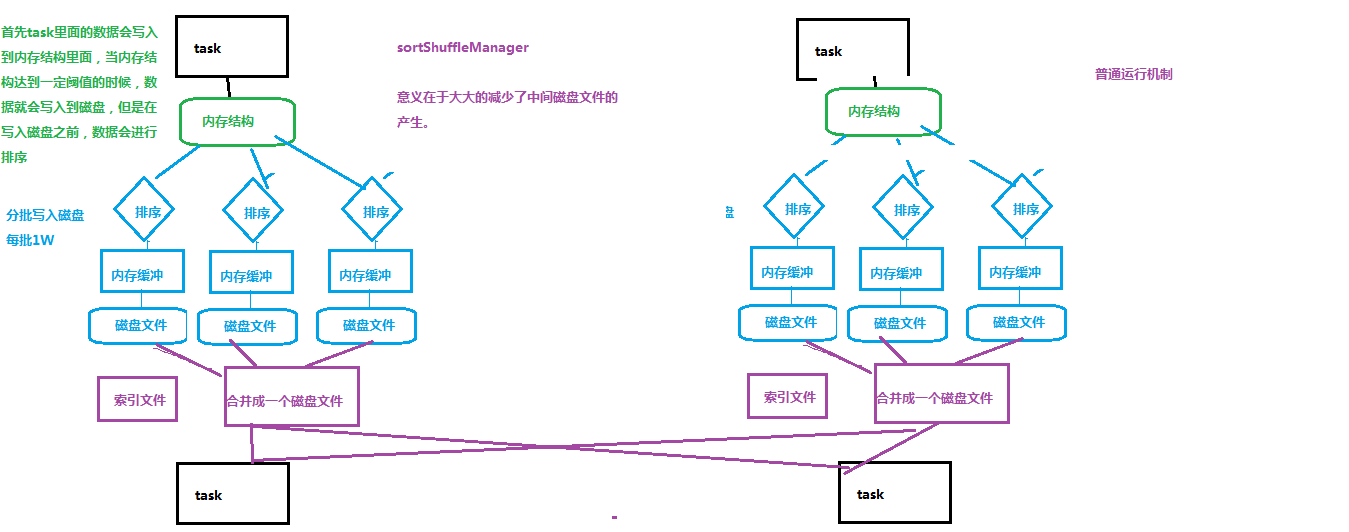
未经过优化



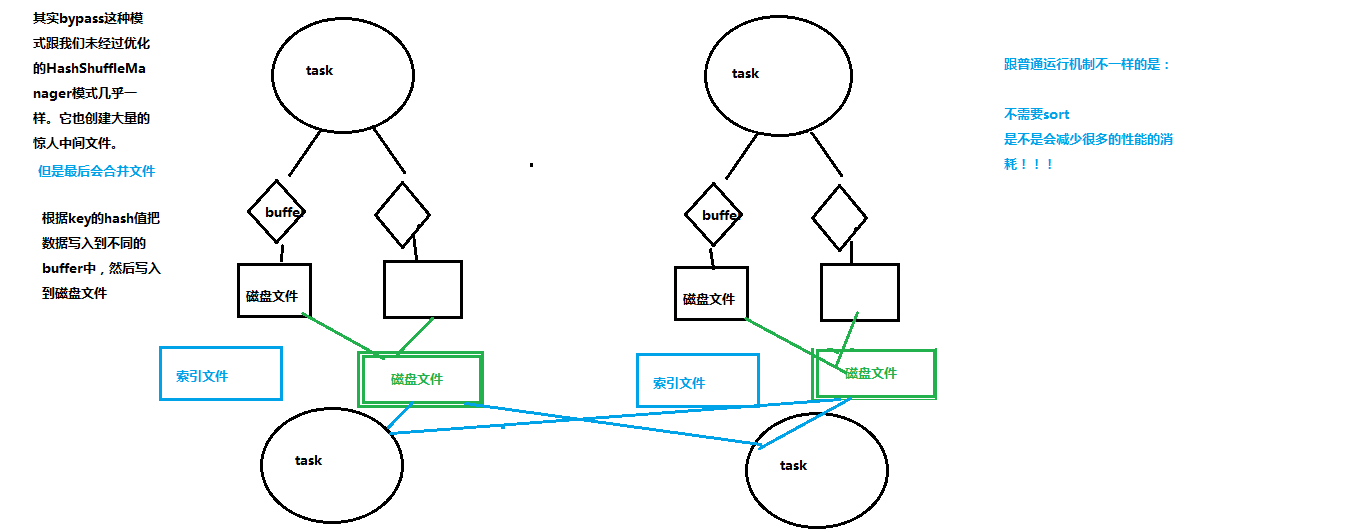
经过优化



Sort 普通



Bypass



# 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 （版本间有变化）

* 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

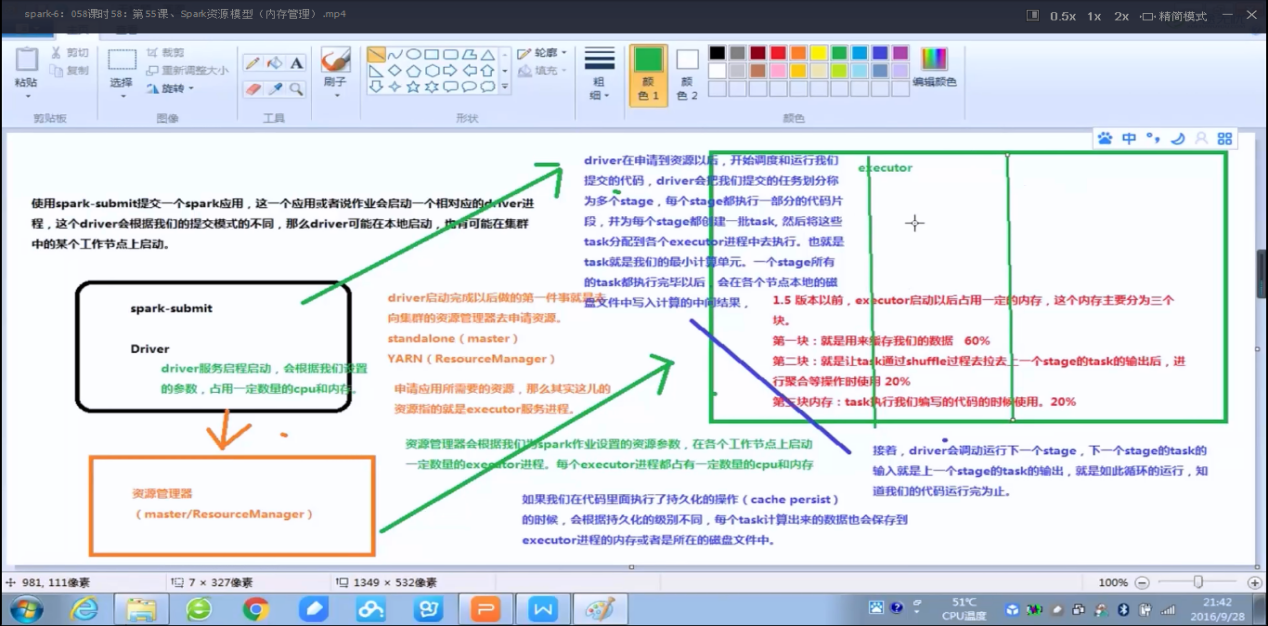
* 默认值：200
* 参数说明：当ShuffleManager为SortShuffleManager时，如果shuffle read task的数量小于这个阈值（默认是200），则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。
* 调优建议：**当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量。那么此时就会自动启用bypass机制，map-side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%。**

# Spark资源模型

Spark作业运行图：



**Spark 内存模型：**

    Spark在一个Executor中的内存分为三块，一块是execution内存，一块是storage内存，一块是other内存。

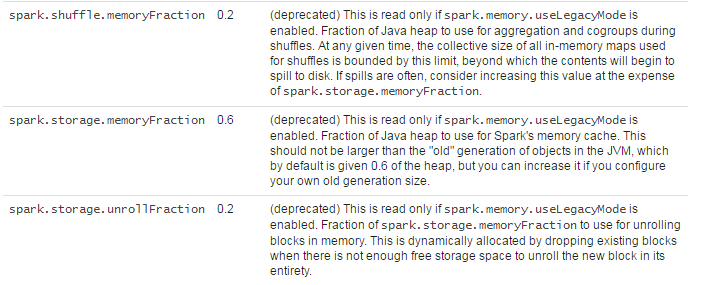
* execution内存是执行内存，文档中说join，aggregate都在这部分内存中执行，shuffle的数据也会先缓存在这个内存中，满了再写入磁盘，能够减少IO。其实map过程也是在这个内存中执行的。（0.25）-（不发生shullfer时这部分闲置，可供storage借用）
* storage内存是存储broadcast，cache，persist数据的地方。（0.5）
* other内存是程序执行时预留给自己的内存。(执行代码的时候使用)（0.25）

    execution和storage是Spark Executor中内存的大户，other占用内存相对少很多，这里就不说了。在spark-1.6.0以前的版本，execution和storage的内存分配是固定的，使用的参数配置分别是spark.shuffle.memoryFraction（execution内存占Executor总内存大小，default 0.2）和spark.storage.memoryFraction（storage内存占Executor内存大小，default 0.6），因为是1.6.0以前这两块内存是互相隔离的，这就导致了Executor的内存利用率不高，而且需要根据Application的具体情况，使用者自己来调节这两个参数才能优化Spark的内存使用。在spark-1.6.0以上的版本，execution内存和storage内存可以相互借用，提高了内存的Spark中内存的使用率，同时也减少了OOM的情况。

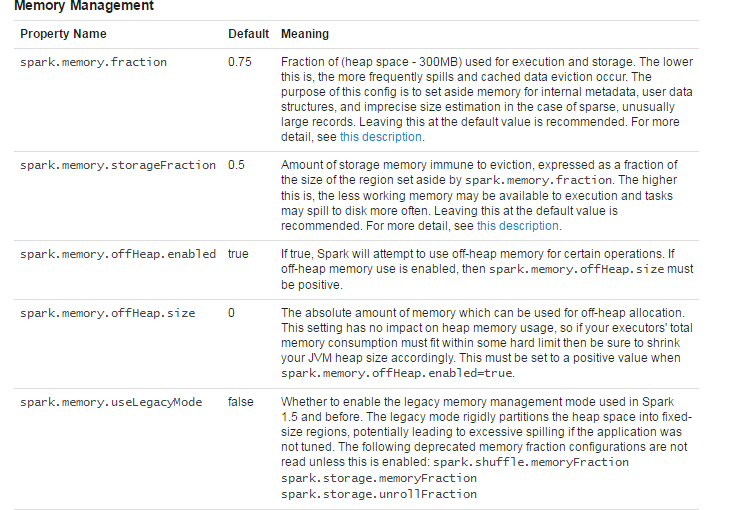
在Spark-1.6.0后加入了堆外内存，进一步优化了Spark的内存使用，堆外内存使用JVM堆以外的内存，不会被gc回收，可以减少频繁的full gc，所以在Spark程序中，会长时间逗留再Spark程序中的大内存对象可以使用堆外内存存储。使用堆外内存有两种方式，一种是在rdd调用persist的时候传入参数StorageLevel.OFF\_HEAP，这种使用方式需要配合Tachyon一起使用。另外一种是使用Spark自带的spark.memory.offHeap.enabled 配置为true进行使用，但是这种方式在1.6.0的版本还不支持使用，只是多了这个参数，在以后的版本中会开放。

OOM的问题通常出现在execution这块内存中，因为storage这块内存在存放数据满了之后，会直接丢弃内存中旧的数据，对性能有影响但是不会有OOM的问题。

1.5 之前



1.5 之后



# 资源调优

## ****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，就代表了你的Spark作业申请到的总内存量（也就是所有Executor进程的内存总和），这个量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，那么申请的总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。（一个cpu core对应2-3 个task）

## ****spark.storage.memoryFraction(包括1.5和1.5以前)****

**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）

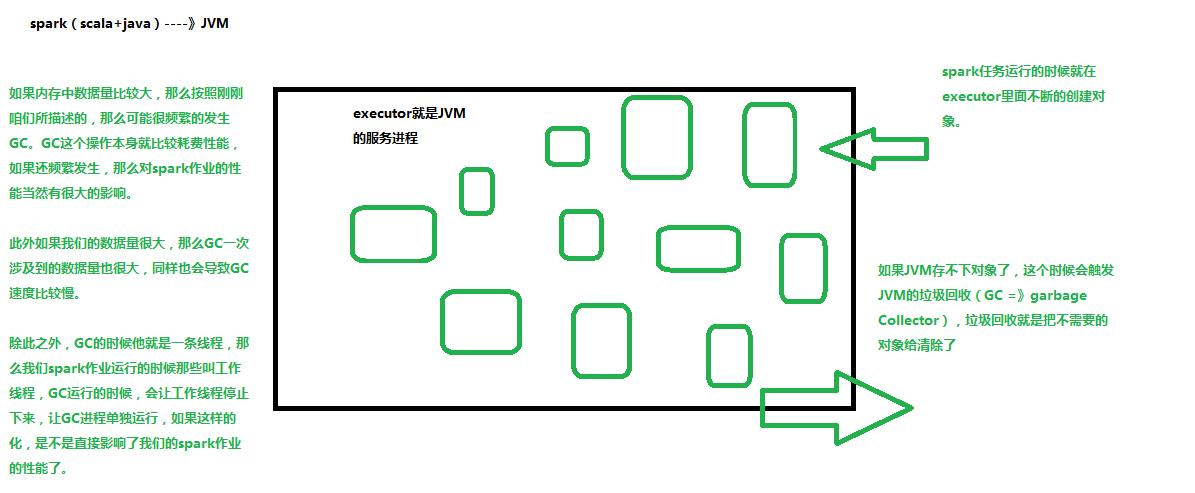
|  |  |  |
| --- | --- | --- |
| spark.memory.fraction | 0.65 | Fraction of (heap space - 300MB) used for execution and storage. The lower this is, the more frequently spills and cached data eviction occur. The purpose of this config is to set aside memory for internal metadata, user data structures, and imprecise size estimation in the case of sparse, unusually large records. Leaving this at the default value is recommended. For more detail, see [this description](http://spark.apache.org/docs/1.6.2/tuning.html" \l "memory-management-overview). |
| spark.memory.storageFraction | 0.4 | Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by s​park.memory.fraction. The higher this is, the less working memory may be available to execution and tasks may spill to disk more often. Leaving this at the default value is recommended. For more detail, see [this description](http://spark.apache.org/docs/1.6.2/tuning.html" \l "memory-management-overview). |

Spark.memory.fraction （如果作业运行缓慢，调低这个值给更多的内存给task运行）

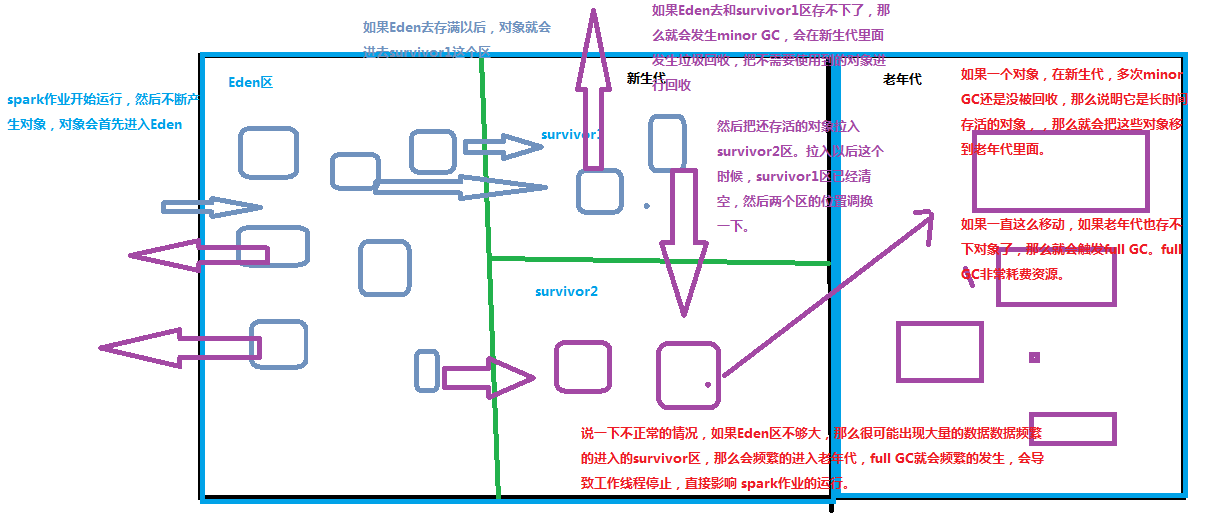
# Spark JVM调优(1)

### Spark 发生垃圾回收的的原理：

垃圾回收的原理



画图详解



### 基础调优思路：

垃圾回收的性能开销，是跟内存中的对象的数量，成正比的。

1. 对于垃圾回收的性能问题，首先要做的就是，使用更高效的数据结构，比如array（替换集合）和string（替换对象）；
2. 持久化rdd时，使用序列化的持久化级别，而且用Kryo序列化类库，这样，每个partition就只是一个对象——一个字节数组。

3）给Eden区域分配更大的空间，使用-Xmn即可，通常建议给Eden区域，预计大小的4/3；

如果使用的是HDFS文件，那么很好估计Eden区域大小，如果每个executor有4个task，然后每个hdfs压缩块解压缩后大小是3倍，此外每个hdfs块的大小是64M，那么Eden区域的预计大小就是：4 \* 3 \* 64MB，然后呢，再通过-Xmn参数，将Eden区域大小设置为4 \* 3 \* 64 \* 4/3。

# Spark JVM调优(2)

在使用大数据工作的时候，我们通常使用如下的工具：

Hadoop

HDFS

Namenode

datanode

YARN

ResourceManager

NodeManager

Hbase

Hmaster

HRegionserver

Spark

Master

Worker

都是Java虚拟机服务进程（JVM）

JVM：

1:内存的设置

2：垃圾回收的设置

最受欢迎的一个hadoop版本:

CDH(clodera 公司将Apache的版本拿过来以后进行打补丁，进行修改，然后产生一个新的版本，这个版本就是CDH版本。)

Apache

CM（cloudera Manager）

1:可以用来安装大数据集群（hadoop hbase flume zookeeper等等），进行界面化的操作

2：可以用来监控大数据集群

比如我们自己使用了Apache的spark，自己使用tar包去安装的，那么所有的值都是默认值。

如果涉及到spark JVM，那么我们当时压根就没去设置。

如果是使用了CM这个软件去安装的spark集群，那么CM默认就会对这个集群的参数进行优化。

大家看到的如下的参数其实就是从CM的默认参数里面复制过来的，也就是CM对JVM虚拟机进行调优后的配置的参数。

-Xms1073741824 -Xmx1073741824

-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:-CMSConcurrentMTEnabled -XX:CMSInitiatingOccupancyFraction=70 -XX:+CMSParallelRemarkEnabled -XX:OnOutOfMemoryError=/usr/lib64/cmf/service/common/killparent.sh

# Spark JVM调优(3)

如果去开发Java程序

Public class Person{

Public int age;

Public String name;

Public static void main(String[] args){

Int a=0;

String str=””;

Person p=new Person(“zhangsan”,12);

}

}

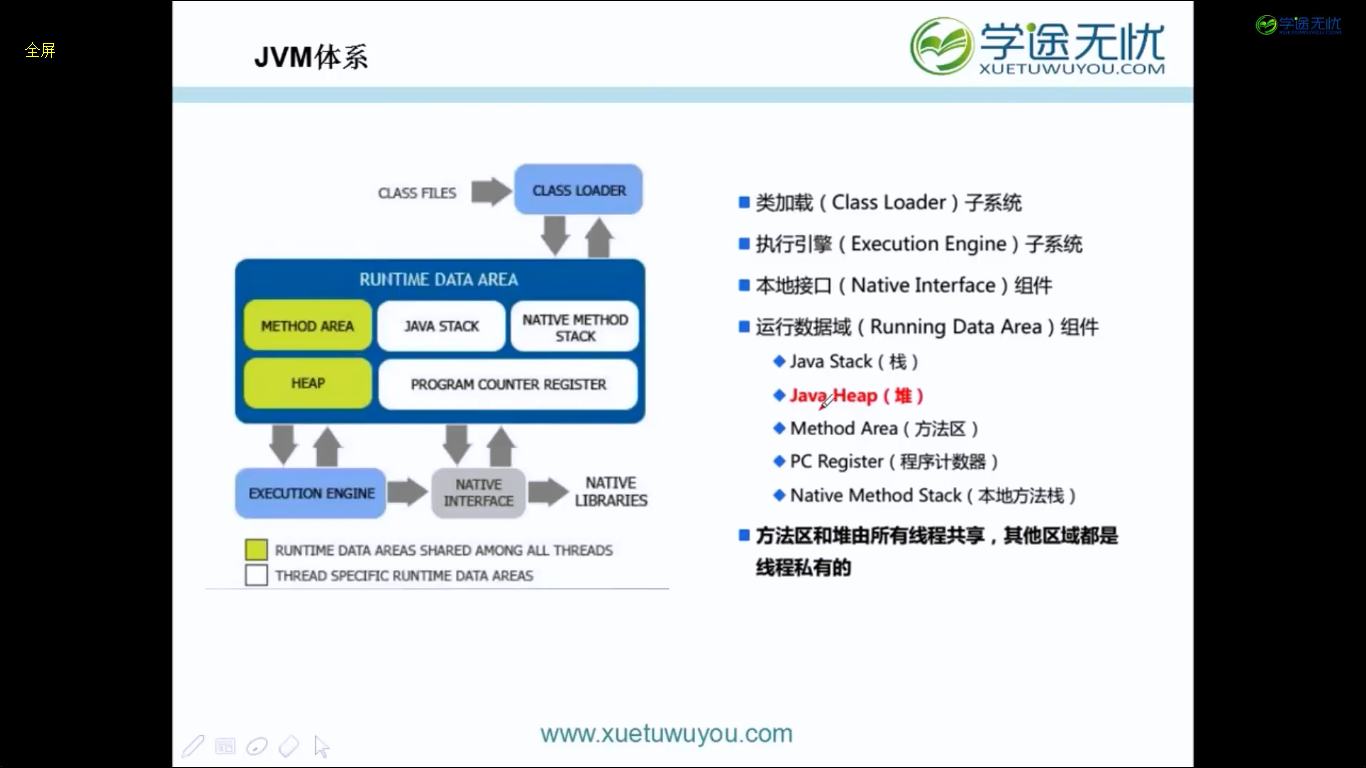
编译：

javac Person.java

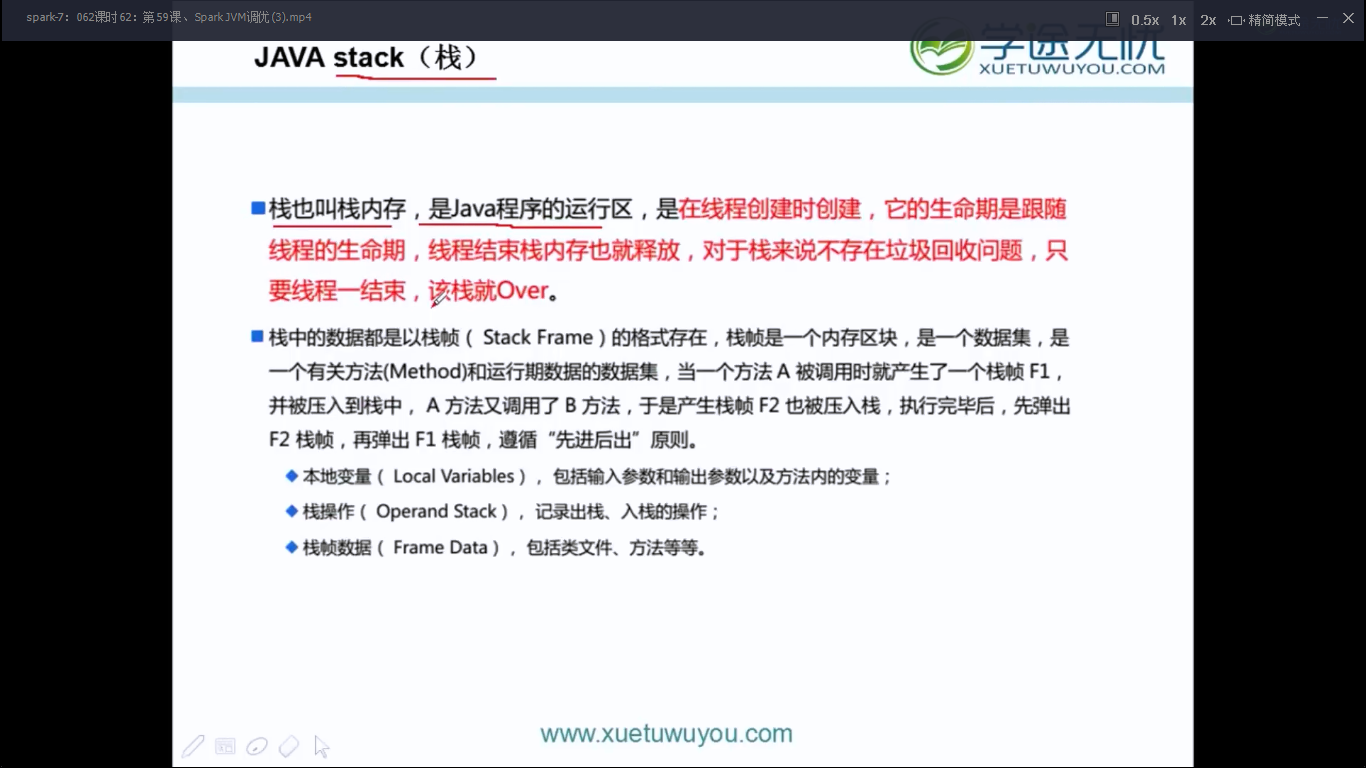
Person.class

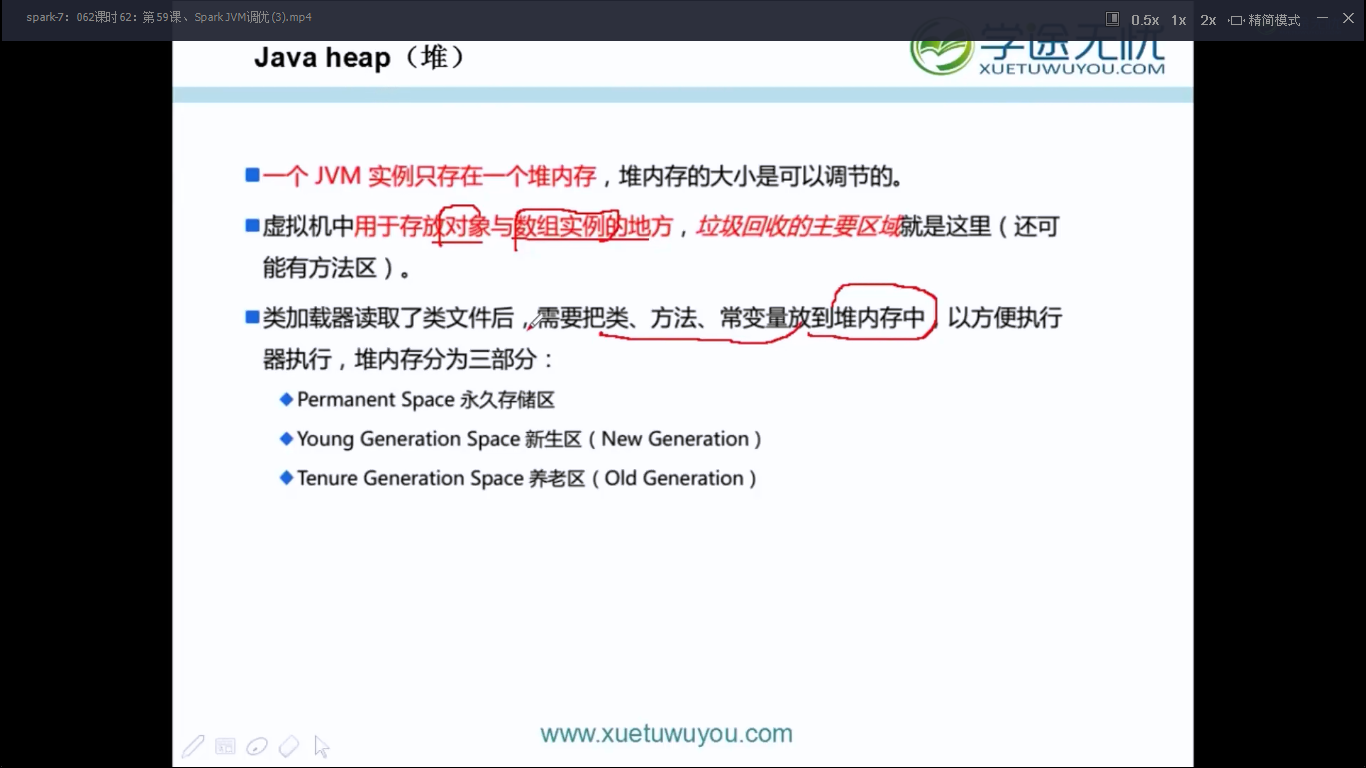
运行：

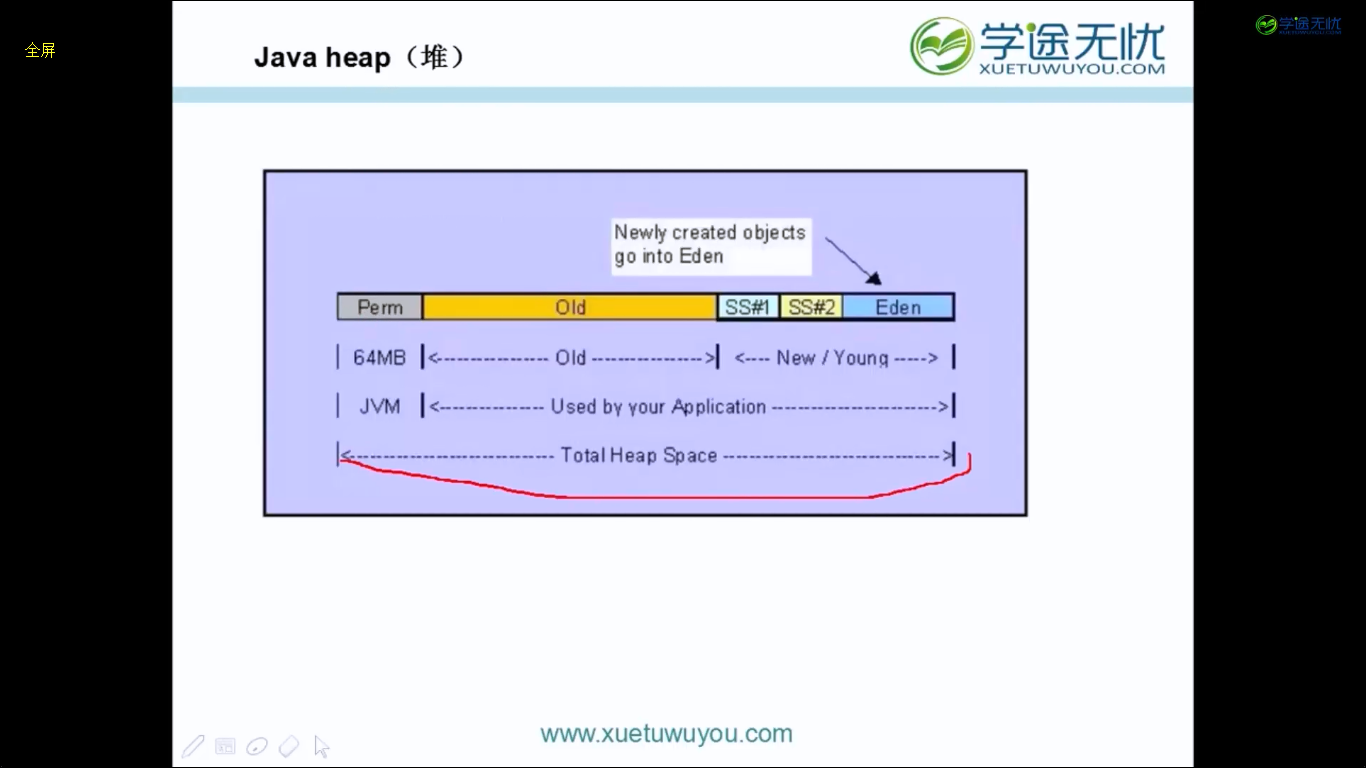
Java Person

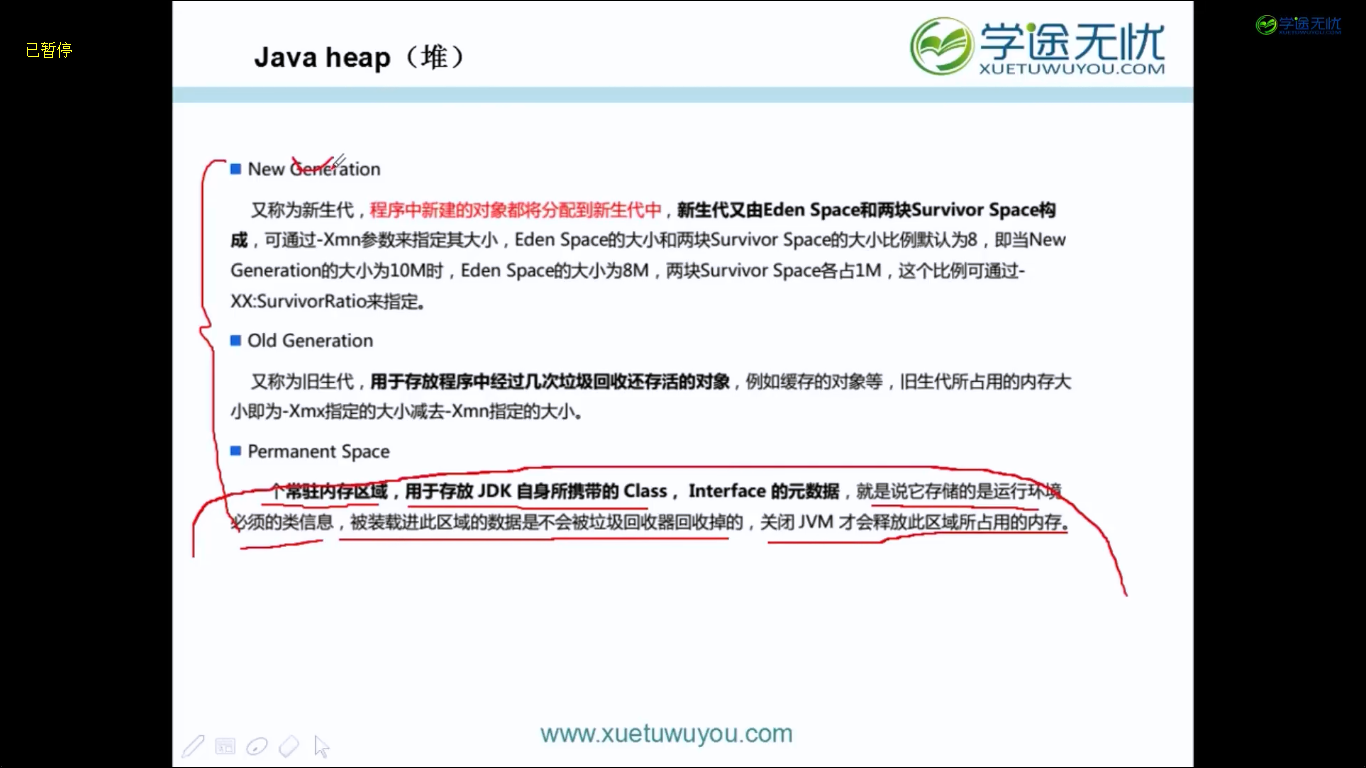


GC调优主要是针对（栈堆内存），栈内存-存放的是-堆内存的引用地址，栈内存不是调优的重点。









有PPT。。

# Spark JVM调优(4)

# Spark JVM调优(5)

-Xmx1073741824(=1G,3个1024相乘) -Xms1073741824

-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:-CMSConcurrentMTEnabled -XX:CMSInitiatingOccupancyFraction=70 -XX:+CMSParallelRemarkEnabled -XX:OnOutOfMemoryError=/usr/lib64/cmf/service/common/killparent.sh

-Xmx:最大堆大小(生产环境需要调高)

-Xms：初始堆大小

-XX:+UseConcMarkSweepGC  
该标志首先是激活CMS收集器。默认HotSpot JVM使用的是并行收集器。

（+启用，-禁用）

-XX:UseParNewGC  
当使用CMS收集器时，该标志激活年轻代使用多线程并行执行垃圾回收。

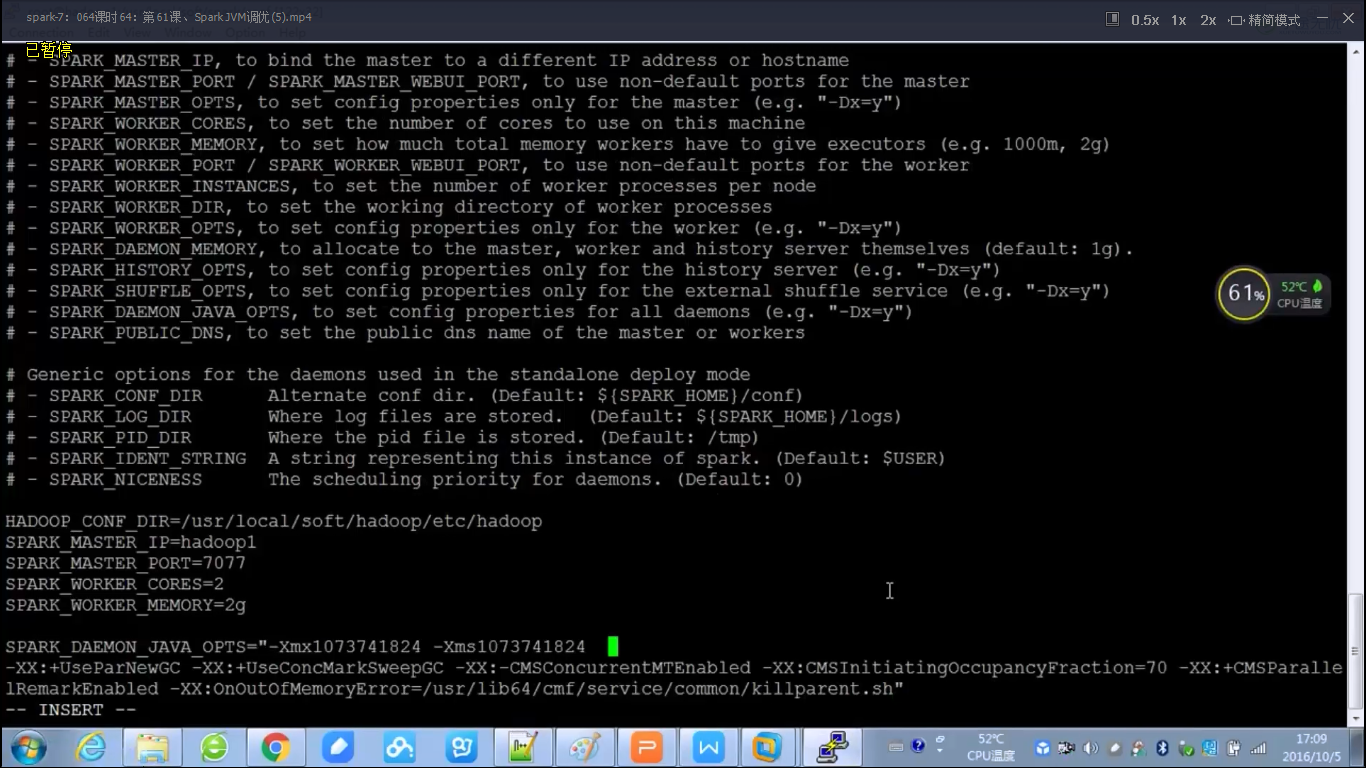
-XX:+CMSConcurrentMTEnabled  
当该标志被启用时，并发的CMS阶段将以多线程执行（因此，多个GC线程会与所有的应用程序线程并行工作）。该标志已经默认开启，如果顺序执行更好，这取决于所使用的硬件，多线程执行可以通过-XX:-CMSConcurremntMTEnabled禁用。

-XX:+CMSParallelRemarkEnabled:降低标记停顿

CMSInitiatingOccupancyFraction: 使用cms作为垃圾回收 年老代使用70％后开始CMS收集

http://www.csdn.net/article/1970-01-01/2824823

在spark-env.sh中配置JVM调优参数，如下：



|  |  |
| --- | --- |
|  |  |

# spark调优总结

开发调优

资源调优

数据本地化

Shuffle调优

数据倾斜调优

JVM调优