Spark调优

性能调优

分配更多资源

- Executor的数量
- 每个Executor所能分配的CPU数量
- 每个Executor所能分配的内存量
- Driver端分配的内存数量

分配资源的位置

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
--total-executor-cores 3 \  //配置所有executor的cpu core数量
/usr/local/SparkTest-0.0.1-SNAPSHOT-jar-with-dependencies.jar \
```

调节到多大合适

常用的资源调度模式有Spark Standalone和Spark On Yarn。比如说你的每台机器能够给你使用60G内存，10个cpu core，20台机器。那么executor的数量是20。平均每个executor所能分配60G内存和10个cpu core

分配资源后，性能得到的提升

- 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，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4个。能够并行执行的task数量，就是80个task。就物理性能来看，执行的速度，提升了2倍。

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：

- 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。

- 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。

- 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。

调节并行度

并行度的概念

Spark作业中，各个stage的task数量，代表了Spark作业的在各个阶段（stage）的并行度

不调节并行度，导致并行度过低，结果会怎么样？

比如现在spark-submit脚本里面，给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，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。

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。

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，就是提升整个Spark作业的性能和运行速度。

设置并行度

- 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）。

- 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。

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架构以及RDD持久化

*RDD*架构重构与优化

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

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，就是将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。

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）。

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，仅仅针对内存资源极度充足的情况。

广播变量

概念及需求

Spark Application（自己写的Spark作业）最开始在Driver端，在提交任务的时候，需要传递到各个Executor的Task上运行。对于一些只读、固定的数据(比如从DB中读出的数据),每次都需要Driver广播到各个Task上，这样效率低下。广播变量允许将变量只广播（提前广播）给各个Executor。

该Executor上的各个Task再从所在节点的BlockManager获取变量，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。而不是从Driver获取变量，从而提升了效率。一个Executor只需要在第一个Task启动时，获得一份Broadcast数据，之后的Task都从本节点的BlockManager中获取相关数据。

使用方法

- 调用SparkContext.broadcast方法创建一个Broadcast[T]对象。任何序列化的类型都可以这么实现。
- 通过value方法访问该对象的值。
- 变量只会被发送到各个节点一次，应作为只读值处理（修改这个值不会影响到别的节点）

使用Kryo序列化

概念及需求

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

*Kryo*序列化机制启用以后生效的几个地方

- 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。
- 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。
- shuffle：可以优化网络传输的性能。

使用方法

- 在SparkConf中设置一个属性，spark.serializer，org.apache.spark.serializer.KryoSerializer类。
- 注册你使用的需要通过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})
```

使用fastutil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

如果算子函数使用了外部变量：

第一，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。

在算子函数里，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，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，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

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 相当于 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，还支持object、reference。

调节数据本地化等待时长

*task*的*locality*有五种

- **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*的任务调度

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。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。

有时可能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。所以这种时候， Spark会等待一段时间，默认情况下是3s（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。

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。

对于自己来说，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，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。

什么时候要调节这个参数*?*

观察spark作业的运行日志。推荐在测试的时候，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。要反复调节，每次调节完以后再运行并观察日志，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。

注意，不要本末倒置，不要本地化级别是提升了，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

堆内存存放创建的一些对象，有老年代和年轻代。理想情况下，老年代都是放一些生命周期很长的对象，数量应该是很少的，比如数据库连接池。在spark task执行算子函数（自己写的），可能会创建很多对象，这些对象都是要放入JVM年轻代中的。

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。会导致这种短生命周期（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，出现的问题：

- 频繁minor gc，也会导致频繁spark停止工作
- 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。

- 严重影响咱们的spark的性能和运行的速度。

降低cache操作的内存占比

spark中，堆内存又被划分成了两块，一块是专门用来给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，可以在任务运行界面，去查看spark作业的运行统计，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
```

调节executor堆外内存与连接等待时长

调节*executor*堆外内存

spark作业处理的数据量特别大，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，可能导致后续的stage的task在运行的时候，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。所以会报shuffle output file not found，resubmitting task，executor lost。spark作业彻底崩溃。

上述情况下,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。可以调节堆外内存的上限：

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

spark-submit脚本里面，去用--conf的方式，去添加配置。用new SparkConf().set()这种方式去设置是没有用的！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300M。通常在项目中，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，得到较大的升。

调节连接等待时长

executor会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。而此时上面executor去远程连接的那个executor，因为task创建的对象特别大，特别多，频繁的让JVM堆内存满溢，正在进行垃圾回收。而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。

此时呢，就会没有响应，无法建立网络连接，会卡住。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。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

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

原理概述：

在spark中，主要是以下几个算子：groupByKey、reduceByKey、countByKey、join（分情况，先groupByKey后再join是不会发生shuffle的）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，都要集中到一块儿，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，才能进行处理，<key, Iterable>。

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，统计一共有多少个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，进行处理。shuffle，一定是分为两个stage来完成的。因为这其实是个逆向的过程，不是stage决定shuffle，是shuffle决定stage。

reduceByKey(+)，在某个action触发job的时候，DAGScheduler，会负责划分job为多个stage。划分的依据，就是，如果发现有会触发shuffle操作的算子，比如reduceByKey，就将这个操作的前半部分，以及之前所有的RDD和transformation操作，划分为一个stage。shuffle操作的后半部分，以及后面的，直到action为止的RDD和transformation操作，划分为另外一个stage。

合并map端输出文件

如果不合并*map*端输出文件的话，会怎么样？

举例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。

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。

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。

开启*shuffle map*端输出文件合并的机制

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

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。

合并*map*端输出文件，对*spark*的性能的影响？

- map task写入磁盘文件的IO，减少：100万文件 -> 20万文件

- 第二个stage，原本要拉取第一个stage的task数量份文件，1000个task，第二个stage的每个task，都要拉取1000份文件，走网络传输。合并以后，100个节点，每个节点2个cpu core，第二个stage的每个task，主要拉取100 * 2 = 200个文件即可。此时网络传输的性能消耗也大大减少。

在生产环境中，使用了spark.shuffle.consolidateFiles机制以后，实际的性能调优的效果：对于上述的这种生产环境的配置，性能的提升，还是相当的可观的。

spark作业，5个小时 -> 2~3个小时。

不要小看这个map端输出文件合并机制。实际上，在数据量比较大，本身做了前面的性能调优executor上去->cpu core上去->并行度（task数量）上去，shuffle没调优，shuffle就很糟糕了。大量的map端输出文件的产生，对性能有比较恶劣的影响。这个时候，去开启这个机制，可以很有效的提升性能。

调节map端内存缓冲与reduce端内存占比

默认情况下可能出现的问题

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，默认的行为就是将在内存放不下的数据都spill（溢写）到磁盘文件中去。

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

调优方式

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

在生产环境中，什么时候来调节两个参数?

看Spark UI，如果采用standalone模式，spark跑起来，会显示一个Spark UI的地址，4040的端口。进去观察每个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，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。

首先当然是考虑开启map端输出文件合并机制。

其次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，调节的太大了，其他环节的内存使用就会有问题了。

调节以后的效果

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。

HashShuffleManager与SortShuffleManager

Shuffle调优概述

Spark作业的性能主要就是消耗在了shuffle环节，因为该环节包含了大量的磁盘IO、序列化、网络数据传输等操作。

因此，如果要让作业的性能更上一层楼，就有必要对shuffle过程进行调优。影响一个Spark作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle调优只能在整个Spark的性能调优中占 到一小部分而已。

ShuffleManager发展概述

在Spark的源码中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。

该ShuffleManager而HashShuffleManager有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘IO操作影响了性能。

因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。 SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。

主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（merge）成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

在spark 1.5.x以后，对于shuffle manager又出来了一种新的manager，tungsten-sort（钨丝），钨丝sort shuffle manager。官网上一般说，钨丝sort shuffle manager，效果跟sort shuffle manager是差不多的。但是，唯一的不同之处在于，钨丝manager，是使用了自己实现的一套内存管理机制，性能上有很大的提升， 而且可以避免shuffle过程中产生的大量的OOM，GC，等等内存相关的异常

hash、sort、tungsten-sort。如何来选择?

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。

什么时候需要用sort shuffle manager？如果需要那些数据按key排序了，那么就选择这种，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，一定要考量一下，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。

如果不需要排序，而且希望每个task输出的文件最终是会合并成一份的，认为可以减少性能开销。可以去调节bypassMergeThreshold这个阈值，比如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）

如果想选用sort based shuffle manager，而且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：

- 在生产环境中，不建议贸然使用第三点和第四点：
- 如果不想要数据在shuffle时排序，那么就自己设置一下，用hash shuffle manager。
- 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，spark.shuffle.sort.bypassMergeThreshold：200。

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。

算子调优

*MapPartitions*提升*Map*类操作性能

Spark中，最基本的原则，就是每个task处理一个RDD的partition。

*MapPartitions*的优缺点

MapPartitions操作的优点：

如果是普通的map，比如一个partition中有1万条数据。ok，那么function要执行和计算1万次。但是，使用MapPartitions操作之后，一个task仅仅会执行一次function，function一次接收所有的partition数据。只要执行一次就可以了，性能比较高。

MapPartitions的缺点：

如果是普通的map操作，一次function的执行就处理一条数据。那么如果内存不够用的情况下，比如处理了1千条数据了，那么这个时候内存不够了，那么就可以将已经处理完的1千条数据从内存里面垃圾回收掉，或者用其他方法，腾出空间来吧。所以说普通的map操作通常不会导致内存的OOM异常。

但是MapPartitions操作，对于大量数据来说，比如甚至一个partition，100万数据，一次传入一个function以后，那么可能一下子内存不够，但是又没有办法去腾出内存空间来，可能就OOM，内存溢出。

*MapPartitions*使用场景

当分析的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，先去估算一下RDD的数据量，以及每个partition的量，还有分配给每个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。

*filter*过后使用*coalesce*减少分区数量

出现问题

默认情况下，经过了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）

可能出现的问题：

- 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。

- 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，这样就会导致有些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，希望应该能够怎么样？

- 针对第一个问题，希望可以进行partition的压缩，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）

- 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。

解决问题方法

调用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，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。

使用*foreachPartition*优化写数据库性能

默认的foreach的性能缺陷

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，都去创建一个数据库连接的话，那么就得创建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。

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foreachPartition算子的好处

- 对于写的function函数，就调用一次，一次传入一个partition所有的数据。
- 主要创建或者获取一个数据库连接就可以。
- 只要向数据库发送一次SQL语句和多组参数即可。

使用*repartition*解决*Spark SQL*低并行度的性能问题

设置并行度

- spark.default.parallelism
- textFile()，传入第二个参数，指定partition数量（比较少用）

根据application的总cpu core数量（在spark-submit中可以指定），手动设置spark.default.parallelism参数，指定为cpu core总数的2~3倍

设置并行度，在什么情况下会生效？什么情况下不会生效？

如果没有使用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，用Spark SQL的情况下，stage的并行度没法自己指定。

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。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。设置spark.default.parallelism参数为100。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。

可能出现的问题？

Spark SQL默认情况下，并行度，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stage1000个task非常快。

解决*Spark SQL*无法设置并行度和*task*数量的办法

repartition算子，用Spark SQL这一步的并行度和task数量，肯定是没有办法去改变。但是，可以将Spark SQL查询出来的RDD，使用repartition算子去重新进行分区，此时可以分成多个partition。然后呢，从repartition以后的RDD，再往后，并行度和task数量，就会按照预期的来了。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。

*reduceByKey*本地聚合介绍

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，对每一个key，对应的values，都会执行算子函数（_ + _）

用*reduceByKey*对性能的提升

- 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。
- 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。
- 在reduce端进行数据缓存的内存占用变少了。
- reduce端，要进行聚合的数据量也变少了。

*reduceByKey*在什么情况下使用

非常普通的，比如说，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）。

对于一些要对每个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）

troubleshooting

控制*shuffle reduce*端缓冲大小以避免*OOM*

map端的task是不断的输出数据的，数据量可能是很大的。其实reduce端的task，并不是等到map端task将属于自己的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*端缓冲大小的另外一面，关于性能调优的一面

假如Map端输出的数据量也不是特别大，然后整个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。注意，要在资源充足的前提下操作。

*reduce*端缓冲（*buffer*），可能会出现的问题及解决方式

可能会出现，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。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，缓冲区48M，全部填满。

这个时候，再加上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，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）这是典型的以性能换执行的原理。reduce端缓冲小了，不容易OOM了，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。这种时候，只能采取牺牲性能的方式了，spark作业，首先，第一要义，就是一定要让它可以跑起来。

操作方法

```
new SparkConf().set(spark.reducer.maxSizeInFlight，”48”)
```

解决*JVM GC*导致的*shuffle*文件拉取失败

问题描述

有时会出现一种情况，在spark的作业中，log日志会提示shuffle file not found。（spark作业中，非常常见的）而且有的时候，它是偶尔才会出现的一种情况。有的时候，出现这种情况以后，重新去提交task。重新执行一遍，发现就好了。没有这种错误了。

log怎么看？用client模式去提交spark作业。比如standalone client或yarn client。一提交作业，直接可以在本地看到更新的log。

问题原因：比如，executor的JVM进程可能内存不够用了。那么此时就会执行GC。minor GC or full GC。此时就会导致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。

解决方案

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

最多可以忍受1个小时没有拉取到shuffle file。只是去设置一个最大的可能的值。full gc不可能1个小时都没结束吧。这样呢，就可以尽量避免因为gc导致的shuffle file not found，无法拉取到的问题

*YARN*队列资源不足导致的*application*直接失败

问题描述

如果说，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。看起来，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，以及预期的，是要大一些的。400G内存，190个cpu core。那么这个时候，的确，队列资源还是有一些剩余的。但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）

- 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。

- YARN，发现资源不足，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。

解决方案

在J2EE（项目里面，spark作业的运行， J2EE平台触发的，执行spark-submit脚本的平台）进行限制，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。

采用一些简单的调度区分的方式，比如说，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YARN、Hadoop的申请）。搞两个调度队列。每个队列的根据要执行的作业的情况来设置。在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。

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，分配给每一个spark作业，比如80个executor，6G的内存，3个cpu core。尽量让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。

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。

解决各种序列化导致的报错

问题描述

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，就碰到了序列化问题导致的报错。

序列化报错及解决方法

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 {
}
```

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 {
}
```

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

解决算子函数返回*NULL*导致的问题

问题描述

在有些算子函数里面，是需要有一个返回值的。但是，有时候不需要返回值。如果直接返回NULL的话，是会报错的。Scala.Math(NULL)，异常

解决方案

如果碰到是对于某些值不想要有返回值的话，有一个解决的办法：

- 在返回的时候，返回一些特殊的值，不要返回null，比如“-999”

- 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。

- 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。

解决*yarn-client*模式导致的网卡流量激增问题

Spark-On-Yarn任务执行流程

Driver到底是什么?

写的spark程序，打成jar包，用spark-submit来提交。jar包中的一个main类，通过jvm的命令启动起来。

JVM进程，其实就是Driver进程。Driver进程启动起来以后，执行写的main函数，从new SparkContext()开始。

driver接收到属于自己的executor进程的注册之后，就可以去进行写的spark作业代码的执行了。此时会一行一行的去执行咱们写的那些spark代码。执行到某个action操作的时候，就会触发一个job，然后DAGScheduler会将job划分为一个一个的stage，为每个stage都创建指定数量的task。TaskScheduler将每个stage的task分配到各个executor上面去执行。

task就会执行写的算子函数。spark在yarn-client模式下，application的注册（executor的申请）和计算task的调度，是分离开来的。standalone模式下，这两个操作都是driver负责的。ApplicationMaster(ExecutorLauncher)负责executor的申请，driver负责job和stage的划分，以及task的创建、分配和调度。每种计算框架（mr、spark），如果想要在yarn上执行自己的计算应用，那么就必须自己实现和提供一个ApplicationMaster。相当于是实现了yarn提供的接口，spark自己开发的一个类。

*yarn-client*模式下，会产生什么样的问题呢?

由于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）。想象一下，比如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本地机器上发出去的，和接收到的。这是最要命的地方。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，本地机器的网络通信负载是非常非常高的。

会导致本地机器的网卡流量会激增！本地机器的网卡流量激增，当然不是一件好事了。因为在一些大的公司里面，对每台机器的使用情况，都是有监控的。不会允许单个机器出现耗费大量网络带宽等等这种资源的情况。

解决方案

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，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spark作业。

yarn-cluster模式，就跟本地机器引起的网卡流量激增的问题，就没有关系了。使用了yarn-cluster模式以后，就不是本地机器运行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。

解决*yarn-cluster*模式的*JVM*栈内存溢出问题

问题描述

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。

解决方案

既然是JVM的PermGen永久代内存溢出，那么就是内存不够用。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"
```

设置了driver永久代的大小，默认是128M，最大是256M。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。

spark sql中，写sql，要注意一个问题：如果sql有大量的or语句。比如where keywords='' or keywords='' or keywords=''当达到or语句，有成百上千的时候，此时可能就会出现一个driver端的jvm stack overflow，JVM栈内存溢出的问题。

JVM栈内存溢出，基本上就是由于调用的方法层级过多，因为产生了大量的，非常深的，超出了JVM栈深度限制的递归方法。猜测，spark sql有大量or语句的时候，spark sql内部源码中，在解析sql，比如转换成语法树，或者进行执行计划的生成的时候，对or的处理是递归。or特别多的话，就会发生大量的递归。

JVM Stack Memory Overflow，栈内存溢出。这种时候，建议不要搞那么复杂的spark sql语句。采用替代方案：**将一条sql语句，拆解成多条sql语句来执行。**每条sql语句，就只有100个or子句以内。一条一条SQL语句来执行。根据生产环境经验的测试，一条sql语句，100个or子句以内，是还可以的。通常情况下，不会报那个栈内存溢出。

错误的持久化方式以及*checkpoint*的使用

使用持久化方式

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() // Java

val cachedUsersRDD = usersRDD.cache() // Scala
```

之后再去使用usersRDD，或者cachedUsersRDD就可以了。

checkpoint的使用

对于持久化，大多数时候都是会正常工作的。但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，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。

下来task就会对消失的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。不至于去重新计算。checkpoint，其实就是可以作为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：

- 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。
- 在代码中，对需要进行checkpoint的rdd，执行RDD.checkpoint()。
- RDDCheckpointData（spark内部的API），接管你的RDD，会标记为marked for checkpoint，准备进行checkpoint。
- 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。
- 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）。
- 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：

- sc.checkpointFile("hdfs://")，设置checkpoint目录
- 对RDD执行checkpoint操作

数据倾斜解决方案

数据倾斜的解决，跟之前讲解的性能调优，有一点异曲同工之妙。性能调优中最有效最直接最简单的方式就是加资源加并行度，并注意RDD架构（复用同一个RDD，加上cache缓存）。相对于前面，shuffle、jvm等是次要的。

原理以及现象分析

数据倾斜怎么出现的

在执行shuffle操作的时候，是按照key，来进行values的数据的输出、拉取和聚合的。同一个key的values，一定是分配到一个reduce task进行处理的。多个key对应的values，比如一共是90万。可能某个key对应了88万数据，被分配到一个task上去面去执行。另外两个task，可能各分配到了1万数据，可能是数百个key，对应的1万条数据。

这样就会出现数据倾斜问题。

想象一下，出现数据倾斜以后的运行的情况。很糟糕！

其中两个task，各分配到了1万数据，可能同时在10分钟内都运行完了。另外一个task有88万条，88 * 10 = 880分钟 = 14.5个小时。本来另外两个task很快就运行完毕了（10分钟），但是由于一个拖后腿的家伙，第三个task，要14.5个小时才能运行完，就导致整个spark作业，也得14.5个小时才能运行完。数据倾斜，一旦出现，是不是性能杀手！

发生数据倾斜以后的现象

Spark数据倾斜，有两种表现：

大部分的task，都执行的特别特别快，（你要用client模式，standalone client，yarn client，本地机器一执行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。

出现以上loginfo，就表明出现数据倾斜了。这样还算好的，因为虽然老牛拉破车一样非常慢，但是至少还能跑。

另一种情况是，运行的时候，其他task都执行完了，也没什么特别的问题，但是有的task，就是会突然间报了一个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task每处理一条数据，还要创建大量的对象，内存爆掉了。这样也表明出现数据倾斜了。这种就不太好了，因为程序如果不去解决数据倾斜的问题，压根儿就跑不出来。作业都跑不完，还谈什么性能调优这些东西?

定位数据倾斜出现的原因与出现问题的位置

根据log去定位

出现数据倾斜的原因，基本只可能是因为发生了shuffle操作，在shuffle的过程中，出现了数据倾斜的问题。因为某个或者某些key对应的数据，远远的高于其他的key。

- 在程序里面找找，哪些地方用了会产生shuffle的算子，groupByKey、countByKey、reduceByKey、join

- 看log：log一般会报是在哪一行代码，导致了OOM异常。或者看log，看看是执行到了第几个stage。spark代码，是怎么划分成一个一个的stage的。哪一个stage生成的task特别慢，就能够自己用肉眼去对spark代码进行stage的划分，就能够通过stage定位到代码，到底哪里发生了数据倾斜。

聚合源数据以及过滤导致倾斜的*key*

数据倾斜解决方案，第一个方案和第二个方案，一起来讲。这两个方案是最直接、最有效、最简单的解决数据倾斜问题的方案。第一个方案：聚合源数据。第二个方案：过滤导致倾斜的key。

后面的五个方案，尤其是最后4个方案，都是那种特别狂拽炫酷吊炸天的方案。但没有第一二个方案简单直接。如果碰到了数据倾斜的问题。上来就先考虑第一个和第二个方案看能不能做，如果能做的话，后面的5个方案，都不用去搞了。有效、简单、直接才是最好的，彻底根除了数据倾斜的问题。

方案一：聚合源数据

一些聚合的操作，比如groupByKey、reduceByKey，groupByKey说白了就是拿到每个key对应的values。reduceByKey就是对每个key对应的values执行一定的计算。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，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。今天跑昨天的数据。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，如“key=sessionid, value: action_seq=1|user_id=1|search_keyword= 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。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字符串进行map操作，进行你需要的操作即可。

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，从而得到每个key的values。

聚合源数据方案第二种做法是，可能没有办法对每个key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。

方案二：过滤导致倾斜的key

如果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，对应的数量都是几十万。这个时候，可以去取舍，如果业务和需求可以理解和接受的话，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spark作业中，自然就不会发生数据倾斜了。

提高*shuffle*操作*reduce*并行度

问题描述

第一个和第二个方案，都不适合做，然后再考虑这个方案。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甚至是基本解决掉数据倾斜的问题。

提升shuffle reduce端并行度的操作方法

很简单，主要给所有的shuffle算子，比如groupByKey、countByKey、reduceByKey。在调用的时候，传入进去一个参数。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。这样的话，就可以让每个reduce task分配到更少的数据。基本可以缓解数据倾斜的问题。

比如说，原本某个task分配数据特别多，直接OOM，内存溢出了，程序没法运行，直接挂掉。按照log，找到发生数据倾斜的shuffle操作，给它传入一个并行度数字，这样的话，原先那个task分配到的数据，肯定会变少。就至少可以避免OOM的情况，程序至少是可以跑的。

提升shuffle reduce并行度的缺陷

治标不治本的意思，因为它没有从根本上改变数据倾斜的本质和问题。不像第一个和第二个方案（直接避免了数据倾斜的发生）。原理没有改变，只是说，尽可能地去缓解和减轻shuffle reduce task的数据压力，以及数据倾斜的问题。

实际生产环境中的经验：

- 如果最理想的情况下，提升并行度以后，减轻了数据倾斜的问题，或者甚至可以让数据倾斜的现象忽略不计，那么就最好。就不用做其他的数据倾斜解决方案了。

- 不太理想的情况下，比如之前某个task运行特别慢，要5个小时，现在稍微快了一点，变成了4个小时。或者是原先运行到某个task，直接OOM，现在至少不会OOM了，但是那个task运行特别慢，要5个小时才能跑完。

那么，如果出现第二种情况的话，各位，就立即放弃第三种方案，开始去尝试和选择后面的四种方案。

使用随机*key*实现双重聚合

使用场景

groupByKey、reduceByKey比较适合使用这种方式。join咱们通常不会这样来做。

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。如果说，之前的第一、第二、第三种方案，都没法解决数据倾斜的问题，那么就只能依靠这一种方式了。

将*reduce join*转换为*map join*

使用方式

普通的join，肯定是要走shuffle。既然是走shuffle，那么普通的join就是走的是reduce join。那怎么将reduce join 转换为mapjoin呢？先将所有相同的key，对应的value汇聚到一个task中，然后再进行join。

使用场景

这种方式适合在什么样的情况下来使用？如果两个RDD要进行join，其中一个RDD是比较小的。比如一个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）。

其中一个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中都保存一份。要确保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。

对于join中有数据倾斜的情况，尽量第一时间先考虑这种方式，效果非常好。不适合的情况：两个RDD都比较大，那么这个时候，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。对于join这种操作，不光是考虑数据倾斜的问题。即使是没有数据倾斜问题，也完全可以优先考虑，用这种高级的reduce join转map join的技术，不要用普通的join，去通过shuffle，进行数据的join。完全可以通过简单的map，使用map join的方式，牺牲一点内存资源。在可行的情况下，优先这么使用。不走shuffle，直接走map，是不是性能也会高很多？这是肯定的。

*Sample*采样倾斜*key*单独进行*join*

方案实现思路

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。

使用场景

这种方案什么时候适合使用？优先对于join，肯定是希望能够采用上一个方案，即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RDD的数据，可以自己把它转换成一个中间表，或者是直接用countByKey()的方式，可以看一下这个RDD各个key对应的数据量。此时如果发现整个RDD就一个，或者少数几个key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。什么时候不适用呢？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。还是使用最后一个方案，终极的join数据倾斜的解决方案。

就是说，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。

使用随机数以及扩容表进行*join*

使用场景及步骤

当采用随机数和扩容表进行join解决数据倾斜的时候，就代表着，之前的数据倾斜的解决方案，都没法使用。

这个方案是没办法彻底解决数据倾斜的，更多的，是一种对数据倾斜的缓解。

步骤：

- 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。

- 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。

- 最后将两个处理后的RDD进行join操作。

局限性

- 因为两个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。

- 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。

sample采样倾斜key并单独进行join，将key，从另外一个RDD中过滤出的数据，可能只有一条或者几条，此时，可以任意进行扩容，扩成1000倍。将从第一个RDD中拆分出来的那个倾斜key RDD，打上1000以内的一个随机数。这种情况下，还可以配合上，提升shuffle reduce并行度，join(rdd, 1000)。通常情况下，效果还是非常不错的。打散成100份，甚至1000份，2000份，去进行join，那么就肯定没有数据倾斜的问题了吧

实时计算程序性能调优

- 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();
```

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

inputStream.repartition()：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batch每隔5秒生成一次；batch处理时间要达到6秒；就会出现，batch在内存中日积月累，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。