1、Spark性能调优

当程序能够运行起来时，开发人员开始关注这个程序能否节省空间占用、能否运行得更快。这些性能需求就需要开发者通过重构代码或者调整集群的配置参数进行优化。下面介绍一些Spark可以优化的场景和技巧。

- 9.1 配置参数

在Spark应用程序的开发中，需要根据具体的算法和应用场景选择调优方法，没有万能的解决方案，每一种调优方法对一些方面的性能可以优化，对另一些方面的性能可能会产生损耗，下面先介绍对性能调优产生影响的主要参数。

- 1.如何进行参数配置性能调优

熟悉Hadoop开发的用户对配置项应该不陌生。根据不同问题，调整不同的配置项参数是比较基本的调优方案。
配置项可以在脚本文件中添加，也可以在代码中添加。

（1）通过配置文件spark-env.sh添加

可以参照spark-env.sh.template模板文件中的格式进行配置，参见下面的格式。

```
export JAVA_HOME=/usr/local/jdk
export SCALA_HOME=/usr/local/scala
export SPARK_MASTER_IP=127.0.0.1
export SPARK_MASTER_WEBUI_PORT=8088
export SPARK_WORKER_WEBUI_PORT=8099
export SPARK_WORKER_CORES=4
export SPARK_WORKER_MEMORY=8g
```

（2）在程序中通过SparkConf对象添加 如果是在代码中添加，则需要在SparkContext定义之前修改配置项的修改。

例如：

```
val conf = new SparkConf（） .setMaster（"local"） .setAppName（"My application"） .set（"spark.executor.memory"， "1g"
）
val sc = new SparkContext（conf）
```

（3）在程序中通过System.setProperty添加

如果是在代码中添加，则需要在SparkContext定义之前修改配置项。

例如：

```
System.setProperty（"spark.executor.memory"， "14g"） System.setProperty（"spark.worker.memory"， "16g"）
val conf = new SparkConf（） .setAppName（"Simple Application"）
val sc = new SparkContext（conf）
```

Spark官网（http://spark.apache.org/docs/latest/configuration.html）推荐了很多配置参数，读者可以参阅。

- 2.如何观察性能问题 可以通过下面的方式监测性能。

1）Web UI。
2）Driver程序控制台日志。
3）logs文件夹下日志。
4）work文件夹下日志。
5）Profiler工具。

例如，一些JVM的Profiler工具，如Yourkit、Jconsole或者JMap、JStack等命令。更全面的可以通过集群的Profiler工具，如Ganglia、Ambaria等。

- 9.2 调优技巧

一个应用程序可以完成基本功能其实还不够，还有一些更加细节和有实际意义的问题需要考虑，尤其是性能优化问题，但以往的经验教训告诉我们，过早的性能优化是万恶之源，性能优化应该随着程序的开发、调试以及作业的运行观察性能瓶颈，进而进行性能调优。

性能方面的提高概括来说主要包括时间性能提升和空间性能提升，而这两个方面又是一个权衡和矛盾的地方，需要根据应用的具体需求运行环境适当调节，进而在正确完成功能的基础之上，使执行的时间尽可能的短，占用的空间尽量小。

当处理大规模数据时，调优是必须面对的问题，Spark是内存计算，内存问题就变得尤为重要。

下面介绍的调优方法并不能涵盖Spark的全部，更多细节的调优可以到Spark的社区进行提问和查看，上面会有很多和你遇到同样问题的解决方案，以及很多的高手乐于帮你解答。

下面从以下几个方面来介绍Spark的性能调优。

- 9.2.1 调度与分区优化

下面从几个方面讲解调度与分区优化问题。

- 1.小分区合并问题

在用户使用Spark的过程中，常常会使用filter算子进行数据过滤。而频繁的过滤或者过滤掉的数据量过大就会产生问题，造成大量小分区的产生（每个分区数据量小）。由于Spark是每个数据分区都会分配一个任务执行，如果任务过多，则每个任务处理的数据量很小，会造成线程切换开销大，很多任务等待执行，并行度不高的问题，是很不经济的。

例如：

```
val rdd2 = rdd1.filter（line=>lines.contains（"error"））.filter（line=>line.contains（info）.collect（）；
```

解决方式：

可以采用RDD中重分区的函数进行数据紧缩，减少分区数，将小分区合并变为大分区。

通过coalesce函数来减少分区，具体如下。

```
def coalesce（numPartitions： Int， shuffle： Boolean = false）（implicit ord： Ordering[T] = null）： RDD[T]
```

这个函数会返回一个含有numPartitions数量个分区的新RDD，即将整个RDD重分区。

以下几个情景请大家注意，当分区由10000重分区到100时，由于前后两个阶段的分区是窄依赖的，所以不会产生Shuffle 的操作。但是如果分区数量急剧减少，如极端状况从10000重分区为一个分区时，就会造成一个问题：数据会分布到一个节点上进行计算，完全无法开掘集群并行计算的能力。为了规避这个问题，可以设置shuffle=true。

请看源码：

```
new CoalescedRDD（new ShuffledRDD[Int， T， T， （Int，T）]（mapPartitionsWithIndex（distributePartition）， new H
ashPartitioner（numPartitions））， numPartitions）.values
```

由于Shuffle可以分隔Stage，这就保证了上一阶段Stage中的上游任务仍是10000个分区在并行计算。如果不加Shuffle，则两个上下游的任务合并为一个Stage计算，这个Stage便会在1个分区状况下进行并行计算。 同时还会遇到另一个需求，即当前的每个分区数据量过大，需要将分区数量增加，以利用并行计算能力，这就需要把Shuffle设置为true，然后执行coalesce函数，将分区数增大，在这个过程中，默认使用Hash分区器将数据进行重分区。

```
def repartition（numPartitions： Int）（implicit ord： Ordering[T] = null）： RDD[T]
```

rePartition方法会返回一个含有numPartitions个分区的新RDD。repartition的源码如下。

```
def repartition（numPartitions： Int）（implicit ord： Ordering[T] = null）： RDD[T] = { coalesce（numPartitions， shuffle
= true） }
```

reparition本质上就是调用的coalesce方法。因此如果用户不想进行Shuffle，就需用coalese配置重分区，为了方便起见，可以直接用repartition进行重分区。

- 2.倾斜问题

倾斜（skew）问题是分布式大数据计算中的重要问题，很多优化研究工作都围绕该问题展开。倾斜有数据倾斜和任务倾斜两种情况，数据倾斜导致的结果即为任务倾斜，在个别分区上，任务执行时间过长。当少量任务处理的数据量和其他任务差异过大时，任务进度长时间维持在99%（或100%），此时，任务监控页面中有少量（1个或几个）reduce子任务未完成。单一reduce的记录数与平均记录数差异过大，最长时长远大于平均时长，常可能达到3倍甚至更多。

（1）数据倾斜

产生数据倾斜的原因大致有以下几种。

1）key的数据分布不均匀（一般是分区key取得不好或者分区函数设计得不好）。

2）业务数据本身就会产生数据倾斜（像TPC-DS为了模拟真实环境负载特意用有倾斜的数据进行测试）。

3）结构化数据表设计问题。

4）某些SQL语句会产生数据倾斜。

（2）任务倾斜

产生任务倾斜的原因较为隐蔽，一般就是那台机器的正在执行的Executor执行时间过长，因为服务器架构，或JVM，也可能是来自线程池的问题，等等。

解决方式：可以通过考虑在其他并行处理方式中间加入聚集运算，以减少倾斜数据量。 数据倾斜一般可以通过在业务上将极度不均匀的数据剔除解决。这里其实还有Skew Join的一种处理方式，将数据分两个阶段处理，倾斜的key数据作为数据源处理，剩下的key的数据再做同样的处理。二者分开做同样的处理。

（3）任务执行速度倾斜

产生原因可能是数据倾斜，也可能是执行任务的机器在架构，OS、JVM各节点配置不同或其他原因。

解决方式：设置spark.speculation=true（推测机制）把那些执行时间过长的节点去掉，重新调度分配任务，这个方式和Hadoop MapReduce的speculation是相通的。同时可以配置多长时间来推测执行，spark.speculation.interval用来设置执行间隔进行配置。在源码中默认是配置的100，

示例如下。

```
val SPECULATION_INTERVAL = conf.getLong（"spark.speculation.interval"， 100）
```

（4）解决方案

1）增大任务数，减少每个分区数据量：增大任务数，也就是扩大分区量，同时减少单个分区的数据量。

2）对特殊key处理：空值映射为特定Key，然后分发到不同节点，对空值不做处理。

3）广播。 ①小数据量表直接广播。 ②数据量较大的表可以考虑切分为多个小表，多阶段进行Map Side Join。

4）聚集操作可以Map端聚集部分结果，然后Reduce端合并，减少Reduce端压力。

5）拆分RDD：将倾斜数据与原数据分离，分两个Job进行计算。

- 3.并行度

在分布式计算的环境下，如果不能正确配置并行度，就不能够充分利用集群的并行计算能力，浪费计算资源。Spark会根据文件的大小，默认配置Map阶段任务数量，也就是分区数量（也可以通过SparkContext.textFile等方法进行配置）。而Reduce的阶段任务数量配置可以有两种方式，下面分别进行介绍。

第一种方式：写函数的过程中通过函数的第二个参数进行配置。

```
def reduceByKey（func： （V，V）=> V， numPartitions： Int）： RDD[（K， V）]=
{
reduceByKey（new HashPartitioner（numPartitions）， func）
}
```

第二种方式：通过配置spark.default.parallelism来进行配置。

它们的本质原理一致，均是控制Shuffle过程的默认任务数量。

下面介绍通过配置spark.default.parallelism来配置默认任务数量（如groupByKey、reduceByKey等操作符需要用到此参数配置任务数），这里的数量选择也是权衡的过程，需要在具体生产环境中调整，Spark官方推荐选择每个CPU Core分配2~3个任务，即cpu core num*2（或3）数量的并行度。 如果并行度太高，任务数太多，就会产生大量的任务启动和切换开销。 如果并行度太低，任务数太小，就会无法发挥集群的并行计算能力，任务执行过慢，同时可能会造成内存combine数据过多占用内存，而出现内存溢出（out of memory）的异常。

下面通过源码介绍这个参数是怎样发挥作用的。可以通过分区器的代码看到，分区器函数式决定分区数量和分区方式，因为Spark的任务数量由分区个数决定，一个分区对应一个任务。

```
object Partitioner {
def defaultPartitioner（rdd： RDD[_]， others： RDD[_]*）： Partitioner = {
val bySize = （Seq（rdd） ++ others）.sortBy（_.partitions.size）.reverse
for （r <- bySize if r.partitioner.isDefined） {
return r.partitioner.get
}
if （rdd.context.conf.contains（"spark.default.parallelism"））{
new HashPartitioner（rdd.context.defaultParallelism）
} else {
new HashPartitioner（bySize.head.partitions.size）
}
}}
```

从RDD的代码中可以看到，默认的分区函数设置了groupBy的分区数量。

```
def groupBy[K]（f： T => K）（implicit kt： ClassTag[K]）： RDD[（K， Iterable[T]）] =
groupBy[K]（f， defaultPartitioner（this））
```

reduceByKey中也是通过默认分区器设置分区数量。 def reduceByKey（func：（V，V）=> V）：RDD[（K，V）] = { reduceByKey（defaultPartitioner（self），func） } 4.DAG调度执行优化 1）同一个Stage中尽量容纳更多的算子，以减少Shuffle的发生。 由于Stage中的算子是按照流水线方式执行的，所以更多的Transformation放在一起执行能够减少Shuffle的开销和任务启动和切换的开销。 2）复用已经cache过的数据。可以使用cache和persist函数将数据缓存在内存，其实用户可以按单机的方式理解，存储仍然是多级存储，数据存储在访问快的存储设备中，提高快速存储命中率会提升整个应用程序的性能。 9.2.2 内存存储优化 下面将从以下几

个方面讲解内存存储的优化。[1] 1.JVM调优 内存调优过程的大方向上有三个方向是值得考虑的。 1）应用程序中对象所占用的内存空间。 2）访问这些内存对象的代价。 3）垃圾回收的开销。 通常状况下，Java的对象访问速度是很快的，但是相对于对象中存储的原始数据，Java对象整体会耗费2~5倍的内存空间。 （1）内存耗损原因 内存耗损是由以下几个原因造成的，熟悉JVM的用户可能会比较熟悉其中的原因。 1）不同的Java对象都会有一个对象头（object header），这个对象头大约为16byte，包含指向这个对象的类的指针等信息，对一些只有少量数据的对象，这是极为不经济的。例如，只有一个Int属性的对象，这个头的信息所占空间会大于对象的数据空间。）Java中的字符串（String）占用40byte空间。String的内存是将真正字符串的信息存储在一个char数组中，并且还会存储其他的信息，如字符串长度，同时如果采用UTF-16编码，一个字符就占用2byte的空间。综合以上，一个10字符的字符串会占用超过60byte的内存空间。 3）常用的一些集合类，如LinkedList等是采用链式数据结构存储的，对底层的每个数据项进行了包装，这个对象不只存储数据，还会存储指向其他数据项的指针，这些指针也会产生数据空间的占用和开销。 4）集合类中的基本数据类型常常采用一些装箱的对象存储，如java.lang.Ingeger。装箱与拆箱的机制在很多程序设计语言中都有，Java中装箱意味着将这些基本数据类型包装为对象存储在内存的Java堆中，而拆箱意味着将堆中对象转换为栈中存储的数据。 （2）计算内存的消耗 计算数据在集群内存占用的空间的大小的最好方法是创建一个RDD，读取这些数据，将数据加载到cache，在驱动程序的控制台查看SparkContext的日志。这些日志信息会显示每个分区占用多少空间（当内存空间不够时，将数据写到磁盘上），然后用户可以根据分区的总数大致估计出整个RDD占用的空间。例如，下面的日志信息。 INFO BlockManagerMasterActor： Added rdd_0_1 in memory on mbk.local：50311（size： 717.5 KB， free： 332.3 MB）这表示RDD0的partition1消耗了717.5KB内存空间。 （3）调整数据结构 减少内存消耗的第一步就是减少一些除原始数据以外的Java特有信息的消耗，如链式结构中的指针消耗、包装数据产生的元数据消耗等。 1）在设计和选用数据结构时能用数组类型和基本数据类型最好，尽量减少一些链式的Java集合或者Scala集合类型的使用。可以采用fastutil这个第三方库，其中有很多对基本数据类型的集合，能够基本覆盖大部分的Java标准库集合和数据类型。官网地址为http://fastutil.di.unimi.it/。 2）减少对象嵌套。例如，使用大量数据量小、个数多的对象和内含指针的集合数据结构，这样会产生大量的指针和对象头元数据的开销。《编程之美》中提出的“程序简单就是美”的思想在这里也能够体现，不是数据结构设计多复杂，这个程序就多好，而是能解决问题，采用的数据结构又很简单，代码量小，开销小，这样才是最见功力的。 3）考虑使用数字的ID或者枚举对象，而不是使用字符串作为key键的数据类型。从前面也看到，字符串的元数据和本身的字符编码问题产生的空间占用过大。 4）当内存小于32GB时，官方推荐配置JVM参数-XX：+UseCompressedOops，进而将指针由8byte压缩为4byte。OOP的全称是ordinary object pointer，即普通对象指针。在64位HotSpot中，OOP使用32位指针，默认64位指针会比32位指针使用的内存多1.5倍，启用CompressOops后，会压缩的对象如下。 ①每个Class的属性指针（静态成员变量）。 ②每个对象的属性指针。 ③普通对象数组每个元素的指针。 但是，指向PermGen的Class对象指针、本地变量、堆栈元素、入参、返回值、NULL指针不会被压缩。可以通过配置文件spark-env.sh配置这个参数，从而在Spark中启用JVM指针压缩。 （4）序列化存储RDD 如果通过上面的优化方式进行优化，对象存储空间仍然很大，一个更加简便的减少内存消耗的方法是以序列化的格式来存储这些对象。在程序中可以通过设置StorageLevels这个枚举类型来配置RDD的数据存储方式，官网的API文档中提供了更为丰富的RDD存储方式，有兴趣的读者可以自行学习参考。例如，当配置RDD为MEMORY_ONLY_SER存储方式时，Spark将这个RDD的每个分区存储为一个大的byte数组。当然这也是一个权衡的过程，这样的存储会带来数据访问变慢的问题，这是由于每次访问数据还需要经过反序列化的过程。用户如果希望在内存中缓存数据，则官方推荐使用Kyro的序列化库进行序列化，因为Kyro相比于Java的标准序列化库序列化后的对象占用空间更小，性能更好。 （5）JVM垃圾回收（GC）调优当Spark程序产生大数据量的RDD时，JVM的垃圾回收就会成为一个问题。当JVM需要替换和回收旧对象所占空间来为新对象提供存储空间时，根据JVM垃圾回收算法，JVM将遍历所有Java对象，然后找到不再使用的对象进而回收。这里其实开销最大的因素是程序中使用了大量的对象，所以设计数据结构时应该尽量使用创建更少的对象的数据结构，如尽量采用数组Array，而少用链表的LinkedList，从而减少垃圾回收开销。更好的一个方式是将数据缓存为序列化的形式，这些将在序列化的优化方法中详细介绍，这样只有一个对象，即一个byte数组作为一个RDD的分区存储。当遇到GC（垃圾回收）问题时，首先考虑用序列化的方式尝试解决。 当Spark任务的工作内存空间和RDD的缓存数据空间产生干扰时，垃圾回收同样会成为一个问题，可以通过控制分给RDD的缓存来缓解这个问题。 1）度量GC的影响。 GC调优的第一步是统计GC的频率和GC的时间开销。可以设置spark-env.sh中的SPARK_JAVA_OPTS参数，添加选项-verbose：gc-XX：+PrintGCDetails-XX：+PrintGCTime-Stamps。当用户下一次的Spark任务运行时，将会看到worker的日志信息中出现打印GC的时间等信息，需要注意的是，这些信息都Worker节点显示，而不在驱动程序的控制台显示。 2）缓存大小调优。 对GC来说，一个重要的配置参数就是内存给RDD用于缓存的空间大小。默认情况下，Spark用配置好的Executor 60%的内存（spark.executor.memory）缓存RDD。这就意味着40%的剩余内存空间可以让Task在执行过程中缓存新创建的对象。在有些情况下，用户的任务变慢，而且JVM频繁地进行垃圾回收或者出现内存溢出（out of memory异常），这时可以调整这个百分比参数为50%。这个百分比参数可以通过配置spark-env.sh中的变量spark.storage.memoryFraction=0.5进行配置。同时结合序列化的缓存存储对象减少内存空间占用，将会更加有效地缓解垃圾回收问题。下面介绍一些高级GC调优技术。图9-1为Java堆中的各代内存分布。 图9-1 JVM内存分布 ①Young（年轻代）。 年轻代分为3个区：一个Eden区和两个Survivor区（Survivor Space）。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当一个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当这个Survivor区也满时，从第一个Survivor区复制过来的且还存活的对象，将被复制Tenured（老年）区。需要注意的是，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden区复制过来的对象和从前一个Survivor区复制过来的对象，而复制到年老区的只有从第一个Survivor区过来的对象。而且，Survivor区总有一个是空的。大多数情况下Java程序新建的对象都是从新生代分配内存。 不同的GC方式会以不同的方式按此值来划分Eden区和Survivor区的大小，有的GC还会根据运行情况动态调整这3个区的大小。 ②Tenured（年老代）。 年老代存放从年轻代存活的对象。一般来说，年老代存放的都是生命期较长的对象。 ③Perm（持久代）。 持久代用于存放静态文件、Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，这时需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小可通过-XX：MaxPermSize=设置。 持久代对应内存模型中的方法区，存放了加载类的信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的field信息、类中的方法信息，开发人员通过反射机制访问该区域。在sun jdk中，该区默认的最小值为16MB，最大值为64MB，可以

通过-XX：PermSize和-XX：MaxPermSize来指定最小值和最大值。 3）全局GC调优。 Spark中全局的GC调优要确保只有存活时间长的RDD存储在老年代（Old generation）区域，这样保证年轻代（Young）有足够的空间存储存活时间短的对象。这有助于减少Spark任务执行时需要给数据分配的空间，用户可以通过下面的方法观察和解决full GC问题。 可以通过观察日志信息查看是否存在过多过频繁的GC。如果full GC在任务执行完成之前被触发多次，就表示对正在执行的任务没有足够的内存空间分配。 如果从打印的GC日志来看，老年代将要满了，就应该减少缓存数据的内存使用量，可以通过配置spark.storage.memoryFraction属性进行配置，缓存更少的对象还是比减慢内存执行时间更加经济。下面具体讲解spark.storage.memoryFraction属性。 spark.storage.memoryFraction控制用于Spark缓存的Java堆空间，默认值为0.67，即2/3的Java堆空间用于Spark的缓存。如果任务的计算过程中需要用到较多的内存，而RDD所需内存较少，就可以调低这个值，以减少计算过程中因为内存不足而产生的GC过程。在调优过程中发现，GC过多是导致任务运行时间较长的一个常见原因。如果任务运行较慢，想确定是否是GC太多导致的，可以在spark-env.sh中设置JAVA_OPTS参数，以打印GC的相关信息，设置如下。 JAVA_OPTS=" -verbose：gc -XX：+PrintGCDetails -XX：+PrintGCTimeStamps" 这样如果有GC发生，就可以在master和work的日志上看到。 下面通过源码看看这个参数是怎样发挥作用的。 在BlockManager中对memoryFraction private def getMaxMemory（conf： SparkConf）： Long = {　val memoryFraction = conf.getDouble（"spark.storage.memoryFraction"，0.6）　（Runtime.getRuntime.maxMemory * memoryFraction）.toLong　} 如果看到GC日志中有很多minor GC信息，而非major GC信息，分配更多的内存给Eden区将会很有帮助。可以设定估计出的每个任务执行需要的内存为Eden区内存大小。如果已经设置Eden区内存大小为E，就可以通过JVM配置参数-Xmn=4/3*E设置年轻代大小，多出的空间分配给Survivor区域使用。 例如，如果任务是从HDFS读取数据，内存空间的占用可以通过从HDFS读取的数据块大小和数量估计。需要注意的是，一般情况下，压缩的数据压缩之后通常为原来数据块大小的2~3倍。因此如果一个JVM中要执行3~4个任务，同时HDFS的数据块大小是64MB，就可以估计需要的Eden代大小是4×3×64MB大小的空间。 最后监控修改了配置参数之后，Spark应用的GC频率和时间开销，进一步调优。 官方给出的GC调优建议是，GC调优依赖于两个关键因素：应用程序和集群能够提供的可用内存大小。在Oracle的官网（http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html）上有更多高级的GC调优方法。不论怎样减少GC的频度，都可以明显减少开销。 2.OOM问题优化 相信有一定Spark或者Hadoop开发经验的用户或多或少都遇到过OutOfMemoryError内存溢出问题。 通过之前的介绍，读者已经对JVM的内存管理有了大致的了解，JVM管理大致分为这几个区域：permanent generation space（持久代区域）、heap space（堆区域）、Java stacks（Java栈）。 持久代区域主要存放类和元数据信息，Class第一次加载时被放入PermGen space区域，Class需要存储的内容主要包括方法和静态属性。 堆区域用来存放对象，对象需要存储的内容主要是非静态属性，包括年轻代和年老代。每次用new创建一个对象实例后，对象实例存储在堆区域中，这部分空间也由JVM的垃圾回收机制管理。Java栈与大多数编程语言，包括汇编语言的栈功能相似，主要存储基本类型变量以及方法的输入输出参数。Java程序的每个线程中都有一个独立的堆栈，然后值类型会存储在栈上。 容易发生内存溢出问题的内存空间包括permanent generation space（持久代空间）和heap space（堆空间），笔者常遇到的情景就是heap space的问题。 发生内存溢出问题的原因是Java虚拟机创建的对象太多，在进行垃圾回收时，虚拟机分配到的堆内存空间已经用满了，与heap space有关。解决这类问题有两种思路，一种是减少App的内存占用消耗，另一种是增大内存资源的供给，具体的做法如下。 1）检查程序，看是否有死循环或不必要重复创建大量对象的地方。找到原因后，修改程序和算法。有很多Java profile工具可以使用，官方推荐的是YourKit其他还有JvisualVM、Jcohsole等工具可以使用。 2）按照之前内存调优中总结的能够减少对象在内存数据存储空间的方法开发程序开发和配置参数。 3）增加Java虚拟机中Xms（初始堆大小）和Xmx（最大堆大小）参数的大小，如set JAVA_OPTS=-Xms256m-Xmx1024m。 引起这个问题的原因还很可能是Shuffle类操作符在任务执行过程中在内存建立的Hash表过大。在这种情况下，可以通过增加任务数，即分区数来提升并行性度，减小每个任务的输入数据，减少内存占用来解决。 3.磁盘临时目录空间优化 配置参数spark.local.dir能够配置Spark在磁盘的临时目录，默认是/tmp目录。在Spark进行Shuffle的过程中，中间结果会写入Spark在磁盘的临时目录中，或者当内存不能够完全存储RDD时，内存放不下的数据会写到配置的磁盘临时目录中。 这个临时目录设置过小会造成No space left on device异常。也可以配置多个盘块spark.local.dir=/mn1/spark，/mnt2/spar，/mnt3/spark来扩展Spark的磁盘临时目录，让更多的数据可以写到磁盘，加快I/O速度。[1] 参考自http://spark.apache.org/docs/latest/tuning.html#memory-tuning ，Spark官网。 9.2.3　网络传输优化 1.大任务分发优化 在任务的分发过程中会序列化任务的元数据信息，以及任务需要的jar和文件。任务的分发是通过AKKA库中的Actor模型之间的消息传送的。因为Spark采用了Scala的函数式风格，传递函数的变量引用采用闭包方式传递，所以当需要传输的数据通过Task进行分发时，会拖慢整体的执行速度。配置参数spark.akka.frameSize（默认buffer的大小为10MB）可以缓解过大的任务造成AKKA缓冲区溢出的问题，但是这个方式并不能解决本质的问题。下面具体讲解配置参数spark.akka.frameSize。 spark.akka.frameSize控制Spark框架内使用的AKKA框架中，Actor通信消息的最大容量（如任务（Task）的输出结果），因为整个Spark集群的消息传递都是通过Actor进行的，默认为10MB。当处理大规模数据时，任务的输出可能会大于这个值，需要根据实际数据设置一个更高的值。如果是这个值不够大而产生的错误，则可以从Worker节点的日志中排查。通常Worker上的任务失败后，主节点Master的运行日志上提示“Lost TID：”，可通过查看失败的Worker日志文件$SPARK_HOME/work/目录下面的日志文件中记录的任务的Serialized size of result是否超过10MB来确定通信数据超过AKKA的Buffer异常。 2.Broadcast在调优场景的使用 Spark的Broadcast（广播）变量对数据传输进行优化，通过Broadcast变量将用到的大数据量数据进行广播发送，可以提升整体速度。Broadcast主要用于共享Spark在计算过程中各个task都会用到的只读变量，Broadcast变量只会在每台计算机器上保存一份，而不会每个task都传递一份，这样就大大节省了空间，节省空间的同时意味着传输时间的减少，效率也高。在Spark的HadoopRDD实现中，就采用Broadcast进行Hadoop JobConf的传输。官方文档的说法是，当task大于20KB时，可以考虑使用Broadcast进行优化，还可以在控制台日志看到任务是多大，进而决定是否优化。还需要注意，每次迭代所传输的Broadcast变量都会保存在从节点Worker的内存中，直至内存不够用，Spark才会把旧的Broadcast变量释放掉，不能提前进行释放。BroadCast变量有一些应用场景，如MapSideJoin中的小表进行广播、机器学习中需要共享的矩阵的广播等。 用户可以调用SparkContext中的方法生成广播变量。 def broadcast[T]（value： T）（implicit arg0： ClassTag[T]）： Broadcast[T] 3.Collect结果过大优化 在开发程序的过程中，会常常用到Collect操作符。Collect函数的实现如下。 def collect（）： Array[T] = {　val results =

sc.runJob（this，（iter： Iterator[T]）=> iter.toArray） Array.concat（results： _*）｝函数通过SparkContext将每个分区执行变为数组，返回主节点后，将所有分区的数组合并成一个数组。这时，如果进行Collect的数据过大，就会产生问题，大量的从节点将数据写回同一个节点，拖慢整体运行时间，或者可能造成内存溢出的问题。 解决方式：当收集的最终结果数据过大时，可以将数据存储在分布式的HDFS或其他分布式持久化层上。将数据分布式地存储，可以减小单机数据的I/O开销和单机内存存储压力。或者当数据不太大，但会超出AKKA传输的Buffer大小时，需要增加AKKA Actor的buffer，可以通过配置参数spark.akka.frameSize（默认大小为10MB）进行调整。 9.2.4 序列化与压缩 前面章节详细介绍了Spark的I/O机制，下面介绍I/O中的主要调优方向。 1.通过序列化优化 序列化的本质作用是将链式存储的对象数据，转化为连续空间的字节数组存储的数据。这样的存储方式就会产生以下几个好处。 1）对象可以以数据流方式进行进程间传输（包含网络传输），同样可以以连续空间方式存储到文件或者其他持久化层中。 2）连续空间的存储意味着可以进行压缩。这样减少数据存储空间和传输时间。 3）减少了对象本身的元数据信息和基本数据类型的元数据信息的开销。 4）对象数减少也会减少GC的开销和压力。 综上所述，数据进行序列化还是很有价值的。 Spark中提供了两个序列化库和两种序列化方式：使用Java标准序列化库进行序列化的方式和使用Kyro库进行序列化的方式。Java标准序列化库兼容性好，但体积大、速度慢，Kyro库兼容性略差，但是体积小、速度快。所以在能使用Kyro的情况下，还是推荐使用Kyro进行序列化。 可以通过spark.serializer="org.apache.spark.serializer.KryoSerializer"来配置是否使用Kyro进行序列化，这个配置参数决定了Shuffle进行网络传输和当内存无法容纳RDD将分区写入磁盘时，使用的序列化器的类型。 在分布式应用中，序列化处于举足轻重的地位。那些需要大量时间进行序列化的数据格式和占据过大空间的对象会拖慢整个应用。通常情况下，序列化是Spark调优的第一步。Spark为了权衡兼容性和性能提供了两种序列化库。 （1）Java标准序列化库 在默认情况下，Spark使用ObjectOutputStream框架进行对象的序列化。可以通过实现java.io.Serializable接口，使对象可以被序列化，也可以扩展java.io.Externalizable进而控制序列化的性能。Java标准序列化库很灵活，并且兼容性好，但是通常情况下，速度较慢，而且导致序列化后数据量较大。 （2）Kyro序列化库 Spark也可以使用Kyro序列化库来更加快速地序列化对象。Kyro相对于Java序列化库能够更加快速和紧凑地进行序列化（通常有10倍的性能优势），但是Kyro并不能支持所有可序列化的类型，如果对程序有较高的性能优化要求，就需要自定义注册类。官方推荐对于网络传输密集型（network-intensive）计算，采用Kyro序列化性能更好。 Spark自动引入了对许多常用的Scala核心类的Kyro的序列化支持，这些类均是在Spark使用的Twitter chill库支持的类。 （3）序列化示例 下面通过一个例子演示如何自定义一个Kryo的可序列化的类。 创建一个公共类，这个类要扩展org.apache.spark.serializer.KryoRegistrator，然后配置spark.kryo.registrator指向它，代码如下。 import com.esotericsoftware.kryo.Kryo import org.apache.spark.serializer.KryoRegistrator class MyRegistrator extends KryoRegistrator { override def registerClasses（kryo： Kryo）{ kryo.register（classOf[MyClass1]） kryo.register（classOf[MyClass2]） }} val conf = new SparkConf（ ）.setMaster（...）.setAppName（...） conf.set（"spark.serializer"，"org.apache.spark.serializer.KryoSerializer"） conf.set（"spark.kryo.registrator"，"mypackage.MyRegistrator"） val sc = new SparkContext（conf） Kyro序列化库的官方文档上描述了更加高级的一些注册方式，如增加自定义序列化代码，用户在使用时可以参考相应文档中的样例，文档网址是https://code.google.com/p/kryo/。 如果对象占用空间很大，需要增加Kryo的缓冲区容量，就需要增加配置项spark.kryoserializer.buffer.mb的数值，默认是2MB，但参数值应该足够大，以便容纳最大的序列化后对象的传输。 如果用户不注册自定义的类，Kyro仍可以运行，但是它会针对每个对象存储一次整个类名，这样会造成很大的空间浪费。 2.通过压缩方式优化 在Spark中对RDD或者Broadcast数据进行压缩，是提高数据吞吐量和性能的一种手段。压缩数据，可以大量减少磁盘的存储空间，同时压缩后的文件在磁盘间传输和I/O以及网络传输的通信开销也会减小；当然压缩和解压缩也会带来额外的CPU开销，但可以节省更多的I/O和使用更少的内存开销。 在Spark应用中，有很大一部分作业是I/O密集型的。数据压缩对I/O密集型的作业带来性能的大大提升，但是如果用户的jobs作业是CPU密集型的，那么再压缩就会降低性能，这就要判断作业的类型，权衡是否要压缩数据。 压缩数据，可以最大限度地减少文件所需的磁盘空间和网络I/O的开销，但压缩和解压缩数据总会增加CPU的开销，故最好对那些I/O密集型的作业使用数据压缩——这样的作业会有富余的CPU资源，或者对那些磁盘空间不富裕的系统。 Spark目前支持LZF和Snappy两种解压缩方式。Snappy提供了更高的压缩速度，LZF提供了更高的压缩比，用户可以根据具体的需求选择压缩方式。具体的介绍可以参见第2章。可以通过表9-1的配置参数配置压缩。 表9-1 压缩配置

参数	参数值	说明
spark.broadcast.compress	true	设置这次参数决定 broadcast 变量是否进行压缩。通常情况下压缩它是一个好的选择
spark.rdd.compress	false	设置此参数决定是否压缩一个已经序列化的 RDD，可以在创建 RDD 时，通过 StorageLevel.MEMORY_ONLY_SER 设定是否序列化。这样虽然耗费一些压缩时间，但是可以节省大量的内存空间
spark.io.compression.codec	org.apache.spark.io.LZFCompressionCodec	通过这个参数决定是采用 LZF，还是 Snappy 压缩算法。LZF 压缩率较高，Snappy 的压缩时间较短，用户可以根据需求具体权衡
spark.io.compression.snappy.block.size	32768	通过这个参数设置 Snappy 压缩算法的块大小

9.2.5 其他优化方法 除了之前介绍的性能调优方法，还有一些其他方法可供使用。 1.批处理 有些程序可能会调用外部资源，如数据库连接等，这些连接通过JDBC或者ODBC与外部数据源进行交互。用户可能会在编写程序时忽略掉一个问题。例如，将所有数据写入数据库，如果是一条一条地写： rdd.map{line=>con=getConnection； con.write（line.toString）； con.close} 因为整个RDD的数据项很大，整个集群会在短时间内产生高并发写入数据库的操作，对数据库压力很大，将产生很大的写入开销。 这里，可以将单条记录写转化为数据库的批量写，每个分区的数据写一次，这样可以利用数据库的批量写优化减少开销和减轻数据库压力。 rdd.mapPartitions（lines => conn.getDBConn； for（item <- lines）write（item.toString）； conn.close） 同理，对于其他类型的需要和外部资源进行交互的操作，也是应该采用这种处理方式。 2.reduce和reduceByKey的优化 reduce是Action操作，reduceByKey是Transformation操作。

reduce的源码如下。 def reduce（f：（T，T）=> T）：T = { val cleanF = sc.clean（f） val reducePartition：Iterator[T] => Option[T] = iter => { if（iter.hasNext）{ Some（iter.reduceLeft（cleanF））} else { None } } var jobResult：Option[T] = None val mergeResult =（index：Int，taskResult：Option[T]）=> { if（taskResult.isDefined）{ jobResult = jobResult match { case Some（value）=> Some（f（value，taskResult.get））case None => taskResult }} } sc.runJob（this，reducePartition，mergeResult） jobResult.getOrElse（throw new UnsupportedOperationException（"empty collection"））} 在reduce函数中会触发sc.runJob，提交任务，reduce是一个Action操作符。 reduceByKey的源码如下。 def reduceByKey（partitioner：Partitioner，func：（V，V）=> V）：RDD[（K，V）] = { combineByKey[V]（（v：V）=> v，func，func，partitioner）} 由代码可知，reduceByeKey并没有触发runJob，而是调用了combineByKey，该函数调用聚集器聚集数据。 reduce是一种聚合函数，可以把各个任务的执行结果汇集到一个节点，还可以指定自定义的函数传入reduce执行。Spark也对reduce的实现进行了优化，可以把同一个任务内的结果先在本地Worker节点执行聚合函数，再把结果传给Driver执行聚合。但最终数据还是要汇总到主节点，而且reduce会把接收到的数据保存到内存中，直到所有任务都完成为止。因此，当任务很多，任务的结果数据又比较大时Driver容易造成性能瓶颈，这样就应该考虑尽量避免reduce的使用，而将数据转化为Key-Value对，并使用reduceByKey实现逻辑，使计算变为分布式计算。 reduceByKey也是聚合操作，是根据key聚合对应的value。同样的，在每一个mapper把数据发送给reducer前，会在Map端本地先合并（类似于MapReduce中的Combiner）。与reduce不同的是，reduceByKey不是把数据汇集到Driver节点，是分布式进行的，因此不会存在reduce那样的性能瓶颈。 3.Shuffle操作符的内存使用 在有些情况下，应用将会遇到OutOfMemory的错误，其中并不是因为内存大小不能够容纳RDD，而是因为执行任务中使用的数据集合太大（如groupByKey）。Spark的Shuffle操作符（sortByKey、groupByKey、reduceByKey、join等都可以算是Shuffle操作符，因为这些操作会引发Shuffle）在执行分组操作的过程中，会在每个任务执行过程中，在内存创建Hash表来对数据进行分组，而这个Hash表在很多情况下通常变得很大。最简单的一种解决方案就是增加并行度，即增加任务数量和分区数量。这样每轮次每个Executor执行的任务数是固定的，每个任务接收的输入数据变少会减少Hash表的大小，占用的内存就会减少，从而避免内存溢出OOM的发生。 Spark通过多任务复用Worker的JVM，每个节点所有任务的执行是在同一个JVM上的线程池中执行的，这样就减少了线程的启动开销，可以高效地支持单个任务200ms的执行时间。通过这个机制，可以安全地将任务数量的配置扩展到超过集群的整体的CPU core数，而不会出现问题。 9.3 本章小结 本章主要介绍了Spark程序的性能调优。在应用开发中首先应该是能够让程序运行，第二步才是在静态代码或者运行程序中诊断性能瓶颈，查找造成性能问题的代码或配置项，然后通过性能调优的原则指导Spark的调优，优化改进代码和配置项。过早的优化是万恶之源，在不恰当的时间进行优化会增加程序复杂性以及延缓开发周期。同时我们也看到大数据系统软件栈多，集群环境复杂，需要考虑更多的因素进行性能调优，这是挑战，同时也是机遇。