Shuffle 相关参数配置

2018年3月1日 19:18

概述

Shuffle是Spark Core比较复杂的模块，它也是非常影响性能的操作之一。因此，在这里整理了会影响Shuffle性能的各项配置。

1）spark.shuffle.manager

Spark 1.2.0官方版本支持两种方式的Shuffle，即Hash Based Shuffle和Sort Based Shuffle。其中在Spark 1.0之前仅支持Hash Based Shuffle。Spark 1.1引入了Sort Based Shuffle。Spark 1.2的默认Shuffle机制从Hash变成了Sort。如果需要Hash Based Shuffle，只需将spark.shuffle.manager设置成“hash”即可。

配置方式：

①进入spark安装目录的conf目录

②cp spark-defaults.conf.template spark-defaults.conf

③spark.shuffle.manager=hash

应用场景：当产生的临时文件不是很多时，性能可能会比sort shuffle要好。

如果对性能有比较苛刻的要求，那么就要理解这两种不同的Shuffle机制的原理，结合具体的应用场景进行选择。

对于不需要进行排序且Shuffle产生的文件数量不是特别多时，Hash Based Shuffle可能是更好的选择；因为Sort Based Shuffle会按照Reducer的Partition进行排序。

而Sort Based Shuffle的优势就在于可扩展性，它的出现实际上很大程度上是解决Hash Based Shuffle的可扩展性的问题。由于Sort Based Shuffle还在不断地演进中，因此它

的性能会得到不断改善。

对于选择哪种Shuffle，如果性能要求苛刻，最好还是通过实际测试后再做决定。不过选择默认的Sort，可以满足大部分的场景需要。

2）spark.shuffle.spill

这个参数的默认值是true，用于指定Shuffle过程中如果内存中的数据超过阈值（参考spark.shuffle.memoryFraction的设置）时是否需要将部分数据临时写入外部存储。如果设置为false，那么这个过程就会一直使用内存，会有内存溢出的风险。因此只有在确定内存足够使用时，才可以将这个选项设置为false。

3）spark.shuffle.memoryFraction

在启用spark.shuffle.spill的情况下，spark.shuffle.memoryFraction决定了当Shuffle过程中使用的内存达到总内存多少比例的时候开始spill。在Spark 1.2.0里，这个值是0.2。通过这个参数可以设置Shuffle过程占用内存的大小，它直接影响了写入到外部存储的频率和垃圾回收的频率。可以适当调大此值，可以减少磁盘I/O次数。

4）spark.shuffle.sort.bypassMergeThreshold

这个配置的默认值是200，用于设置在Reducer的Partition数目少于多少的时候，
Sort Based Shuffle内部不使用归并排序的方式处理数据，而是直接将每个Partition写入单独的文件。这个方式和Hash Based的方式类似，区别就是在最后这些文件还是会合并成一个单独的文件，并通过一个Index索引文件来标记不同Partition的位置信息。

这个可以看作Sort Based Shuffle在Shuffle量比较小的时候对于Hash Based Shuffle的一种

折中。当然了它和Hash Based Shuffle一样，也存在同时打开文件过多导致内存占用增加的问题。因此如果GC比较严重或者内存比较紧张，可以适当降低这个值。

5）spark.shuffle.blockTransferService

在Spark 1.2.0中这个配置的默认值是**netty**，而在之前的版本中是nio。它主要是用于在各个Executor之间传输Shuffle数据。netty的实现更加简洁，但实际上用户不用太关心这个选项。除非有特殊需求，否则采用默认配置即可。

6）spark.shuffle.consolidateFiles

这个配置的默认值是false。主要是为了解决在Hash Based Shuffle过程中产生过多文件的问题。如果配置选项为true，那么对于同一个Core上运行的Shuffle Map Task不会产生一个新的Shuffle文件而是重用原来的。

但是consolidateFiles的机制在Spark 0.8.1就引入了，到Spark 1.2.0还是没有稳定下来。从源码实现的角度看，实现源码是非常简单的，但是由于涉及本地文件系统等限制，这个策略可能会带来各种各样的问题。一般不建议开启

7）spark.shuffle.compress和spark.shuffle.spill.compress

这两个参数的默认配置都是true。spark.shuffle.compress和spark.shuffle.spill.compress都是用来设置Shuffle过程中是否对Shuffle数据进行压缩。其中，前者针对最终写入本地文件系统的输出文件；后者针对在处理过程需要写入到外部存储的中间数据，即针对最终的shuffle输出文件。

1. 设置spark.shuffle.compress

需要评估压缩解压时间带来的时间消耗和因为数据压缩带来的时间节省。如果网络成为瓶颈，比如集群普遍使用的是千兆网络，那么将这个选项设置为true可能更合理；如果计算是CPU密

集型的，那么将这个选项设置为false可能更好。

2. 设置spark.shuffle.spill.compress

如果设置为true，代表处理的中间结果在spill到本地硬盘时都会进行压缩，在将中间结果取回进行merge的时候，要进行解压。因此要综合考虑CPU由于引入压缩、解压的消耗时间和Disk IO因为压缩带来的节省时间的比较。在Disk IO成为瓶颈的场景下，设置为true可能比较合适；如果本地硬盘是SSD，那么设置为false可能比较合适。

8）spark.reducer.maxMbInFlight

这个参数用于限制一个Reducer Task向其他的Executor请求Shuffle数据时所占用的最大内存数，尤其是如果网卡是千兆和千兆以下的网卡时。默认值是 设置这个值需要综合考虑网卡带宽和内存。

Hadoop常见参数控制+调优策略

配置所在文件	参数	参数默认值	作用
hdfs-site.xml	dfs.namenode.support.allow.format	true	NN是否允许被格式化？在生产系统，把它设置为false，阻止任何格式化操作在一个运行的DFS上。 建议初次格式化后，修改配置禁止，改成false
hdfs-site.xml	dfs.heartbeat.interval	3	DN的心跳间隔，秒 在集群网络通信状态不好的时候，适当调大
hdfs-site.xml	dfs.blocksize	134217728	块大小，默认是128MB 必须得是1024的整数倍
hdfs-site.xml	dfs.namenode.checkpoint.period 手动合并:hadoop dfsadmin -rollEdits	3600	edits和fsimage文件合并周期阈值，默认1小时
hdfs-site.xml	dfs.stream-buffer-size	4096	文件流缓存大小。需要是硬件page大小的整数倍。在读写操作时，数据缓存大小。 注意：是1024的整数倍 注意和core-default.xml中指定文件类型的缓存是不同的，这个是dfs共用的
mapred-site.xml	mapreduce.task.io.sort.mb	100	任务内部排序缓冲区大小，默认是100MB 此参数调大，能够减少Spil溢写次数，减少磁盘I/O
mapred-site.xml	mapreduce.map.sort.spill.percent	0.8	Map阶段溢写文件的阈值。不建议修改此值
mapred-site.xml	mapreduce.reduce.shuffle.parallelcopies	5	Reduce Task 启动的并发拷贝数据的线程数 建议，尽可能等于或接近于Map任务数量，
mapred-site.xml	mapreduce.job.reduce.slowstart.completedmaps	0.05	当Map任务数量完成率在5%时，Reduce任务启动，这个参数建议不要轻易改动，如果Map任务总量非常大时，可以将此参数调低，让reduce更早开始工作。
mapred-site.xml	io.sort.factor	10	(merge)文件合并因子，如果文件数量太多，可以适当调大，从而减少I/O次数
mapred-site.xml	mapred.compress.map.output	false	是否对Map的输出结果文件进行压缩，默认是不压缩。但是如果Map的结果文件很大，可以开启压缩，在Reduce的远程拷贝阶段可以节省网络带宽。
mapred-site.xml	mapred.map.tasks.speculative.execution	true	**启动推测执行机制** 推测执行是Hadoop对“拖后腿”的任

			务的一种优化机制，当一个作业的某些任务运行速度明显慢于同作业的其他任务时，Hadoop会在另一个节点上为“慢任务”启动一个备份任务，这样两个任务同时处理一份数据，而Hadoop最终会将优先完成的那个任务的结果作为最终结果，并将另一个任务杀掉。 启动推测执行机制的目的是更快的完成job，但是在集群计算资源紧张时，比如同时在运行很多个job，启动推测机制可能会带来相反效果。如果是这样，就改成false。 对于这个参数的控制，轻易不要改动。
mapred-site.xml	mapred.reduce.tasks.speculative.execution	true	

MR调优策略

Map Task和Reduce Task调优的一个原则就是

1.减少数据的传输量

2.尽量使用内存

3.减少磁盘IO的次数

4.增大任务并行数

5.除此之外还有根据自己集群及网络的实际情况来调优。

硬件角度调优

Hadoop自身架构的基本特点决定了其硬件配置的选项。Hadoop采用了Master/Slave架构，其中，master维护了全局元数据信息，重要性远远大于slave。在较低Hadoop版本中，master存在单点故障问题，因此，master的配置应远远好于各个slave。

操作系统参数调优

1.增大同时打开的文件描述符和网络连接上限

使用ulimit命令将允许同时打开的文件描述符数目上限增大至一个合适的值。同时调整内核参数net.core.somaxconn网络连接数目至一个足够大的值。

补充：net.core.somaxconn的作用

net.core.somaxconn是Linux中的一个kernel参数，表示socket监听（listen）的backlog上限。什么是backlog呢？backlog就是socket的监听队列，当一个请求（request）尚未被处理或建立时，它会进入backlog。而socket server可以一次性处理backlog中的所有请求，处理后的请求不再位于监听队列中。当server处理请求较慢，以至于监听队列被填满后，新来的请求会被拒绝。在Hadoop 1.0中，参数ipc.server.listen.queue.size控制了服务端socket的监听队列长度，即backlog长度，默认值是128。而Linux的参数net.core.somaxconn默认值同样为128。当服务端繁忙时，如NameNode或JobTracker，128是远远不够的。这样就需要增大backlog，例如我们的3000台集群就将ipc.server.listen.queue.size设成了32768，为了使得整个参数达到预期效

果，同样需要将kernel参数net.core.somaxconn设成一个大于等于32768的值。

2.关闭swap分区

避免使用swap分区，提供程序的执行效率。

除此之外，设置合理的预读取缓冲区的大小、文件系统选择与配置及I/O调度器选择等

Spark调优—上篇

2018年2月13日 9:29

更好的序列化实现

Spark用到序列化的地方

1）Shuffle时需要将对象写入到外部的临时文件。

2）每个Partition中的数据要发送到worker上，spark先把RDD包装成task对象，将task通过网络发给worker。

3）RDD如果支持内存+硬盘，只要往硬盘中写数据也会涉及序列化。

默认使用的是java的序列化。但java的序列化有两个问题，一个是性能相对比较低，另外它序列化完二进制的内容长度也比较大，造成网络传输时间拉长。业界现在有很多更好的实现，如**kryo**，比java的序列化快10倍以上。而且生成内容长度也短。时间快，空间小，自然选择它了。

方法一：修改spark-defaults.conf配置文件

设置：

```
spark.serializer   org.apache.spark.serializer.KryoSerializer
```

注意：用空格隔开

方法二：启动spark-shell或者spark-submit时配置

```
--conf spark.serializer=org.apache.spark.serializer.KryoSerializer
```

方法三：在代码中

```
val conf = new SparkConf()
conf.set("spark.serializer","org.apache.spark.serializer.KryoSerializer")
```

三种方式实现效果相同，优先级越来越高。

通过代码使用Kryo

文件数据：

rose 23

tom 25

Person类代码：

```
class Person(var1:String,var2:Int)  extends Serializable{

 var name=var1
 var age=var2

}
```

MyKryoRegister类代码：

```
import org.apache.spark.serializer.KryoRegistrator
import com.esotericsoftware.kryo.Kryo

class MyKryoRegister extends KryoRegistrator {
 def registerClasses(kryo: Kryo): Unit = {
  kryo.register(classOf[Person])
 }
}
```

KryoDriver代码：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.storage.StorageLevel

object KryoDriver {

 def main(args: Array[String]): Unit = {
  val conf = new SparkConf().setMaster("local").setAppName("kryoTest")
    .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    .set("spark.kryo.registrator", "cn.tedu.MyKryoRegister")
  val sc = new SparkContext(conf)
```

```
    val data=sc.textFile("d://people.txt")

    val userData=data.map { x => new Person(x.split(" ")(0),x.split(" ")(1).toInt) }

    userData.persist(StorageLevel.MEMORY_AND_DISK)

    userData.foreach(x=>println(x.name))
  }
}
```

Spark调优—中篇

2018年2月19日 15:54

配置多临时文件目录

spark.local.dir参数。当shuffle、归并排序（sort、merge）时都会产生临时文件。这些临时文件都在这么指定的目录下。那这个文件夹有很多临时文件，如果都发生读写操作，有的线程在读这个文件，有的线程在往这个文件里写，IO性能就非常低。

怎么解决呢？可以创建多个文件夹，每个文件夹都对应一个真实的硬盘。假如原来是3个程序同时读写一个硬盘，效率肯定低，现在让三个程序分别读取3个磁盘，这样冲突减少，效率就提高了。这样就有效提高外部文件读和写的效率。怎么配置呢？只需要在这个配置时配置多个路径就可以。中间用逗号分隔。

spark.local.dir=/home/tmp,/home/tmp2

启用推测执行机制

可以设置spark.speculation true

开启后，spark会检测执行较慢的Task，并复制这个Task在其他节点运行，最后哪个节点先运行完，就用其结果，然后将慢Task 杀死

collect速度慢

我们在讲的时候一直强调，collect只适合在测试时，因为把结果都收集到Driver服务器上，数据要跨网络传输，同时要求Driver服务器内存大，所以收集过程慢。解决办法就是直接输出到分布式文件系统中。

有些情况下，RDD操作使用MapPartitions替代map

map方法对RDD的每一条记录逐一操作。mapPartitions是对RDD里的每个分区操作

rdd.map{ x=>conn=getDBConn.conn;write(x.toString);conn close;}

这样频繁的链接、断开数据库，效率差。

rdd.mapPartitions{(record:=>conn.getDBConn;for(item<-recorders；

write(item.toString);conn close;}

这样就一次链接一次断开，中间批量操作，效率提升。

扩展：GC回收机制及算法

2018年2月19日 19:46

概述

说起垃圾收集（Garbage Collection, GC），大部分人都把这项技术当做Java语言的伴生产物。事实上，GC的历史比Java久远，1960年诞生于MIT的Lisp是第一门真正使用内存动态分配和垃圾收集技术的语言。当Lisp还在胚胎时期时，人们就在思考GC需要完成的3件事情：

1）哪些内存数据需要回收？

2）什么时候回收？

3）如何回收？

经过半个多世纪的发展，目前内存的动态分配与内存回收技术已经相当成熟，一切看起来都进入了“自动化”时代，那为什么我们还要去了解GC和内存分配呢？答案很简单：当需要排查各种内存溢出、内存泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些“自动化”的技术实施必要的监控和调节。

回到我们熟悉的Java语言。Java内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈3个区域随线程而生，随线程而灭，因此这几个区域的内存分配和回收都具备确定性，在这几个区域内就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了。而Java堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，**所以垃圾收集器所关注的是java的堆内存。**

哪些内存数据需要被回收？

在堆里面存放着Java世界中几乎所有的对象实例，垃圾收集器在对**堆**进行回收前，第一件事情就是要确定这些对象之中哪些还“存活”着，哪些已经“死去”（即不可能再被任何途径使用的对象）。

1）引用计数算法

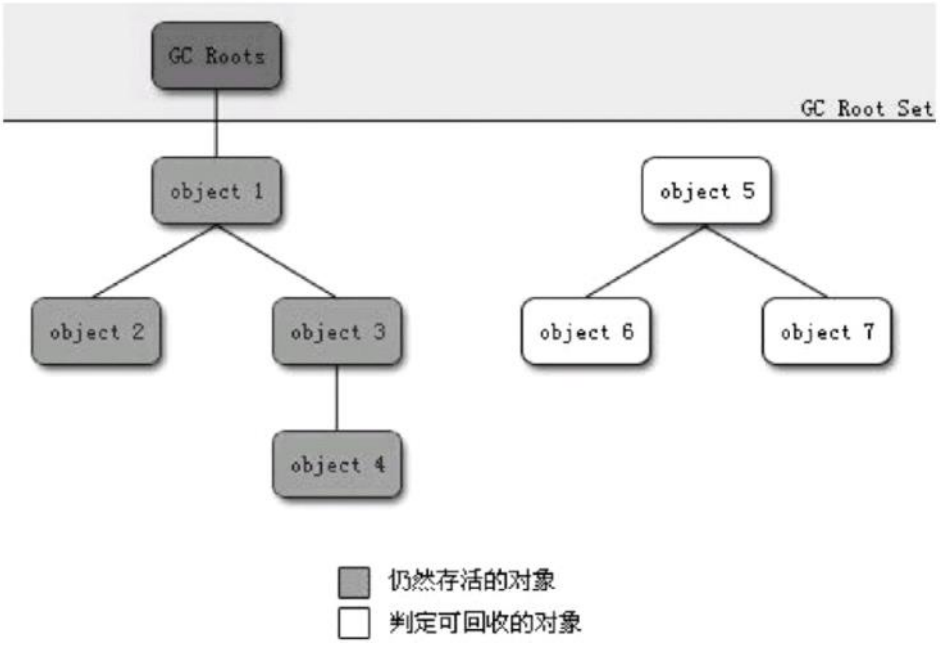
很多教科书判断对象是否存活的算法是这样的：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

客观地说，引用计数算法（Reference Counting）的实现简单，判定效率也很高，在大部分情况下它都是一个不错的算法，也有一些比较著名的应用案例，例如微软公司的COM（Component Object Model）技术、使用ActionScript 3的FlashPlayer、Python语言和在游戏脚本领域被广泛应用的Squirrel中都使用了引用计数算法进行内存管理。但是，至少主流的Java虚拟机里面没有选用引用计数算法来管理内存，其中最主要的原因是它很难解决对象之间相互循环引用的问题。

2）可达性分析算法

在主流的商用程序语言（Java、C#，甚至包括前面提到的古老的Lisp）的主流实现中，都是称通过可达性分析（Reachability Analysis）来判定对象是否存活的。这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。

如下图所示，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。



在Java语言中，可作为GC Roots的对象包括下面几种：

1）虚拟机栈（栈帧中的本地变量表）中引用的对象。

2）方法区中类静态属性引用的对象。

3）方法区中常量引用的对象。

垃圾收集算法

1）标记-清除算法

最基础的收集算法是“标记-清除”（Mark-Sweep）算法，如同它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其不足进行改进而得到的。它的主要不足有两个：一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清除之后会产生**大量不连续的内存碎片**，**空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存**而不得不提前触发另一次垃圾收集动作。

标记—清除算法的执行过程如下图所示。

比如Linux，关闭Swap分区

虚拟内存技术——用磁盘空间当做内存，可以将一部分不常用的内存数据交换到磁盘上，待用时，再从磁盘上进行恢复。但一般在使用技术框架时，会避免此情况产生，主要是为了避免磁盘I/O。比如使用Linux时，关闭Swap分区

2）复制算法

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。

复制算法的执行过程如下图所示。

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM公司的专门研究表明，新生代中的对象98%是“朝生夕死”的，所以并不需要按照1:1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存会被“浪费”。

当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

内存的分配担保指的是：如果另外一块Survivor空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代。此外，对象进入老年代还有另外一个触发条件，就是一个对象在两个Survivor间进行过多次的移动。

3）标记-整理算法

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如

果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存，“标记-整理”算法的示意如下图所示。

所以很多教材说，老年代很少发生GC，因为都是常用的对象，但当老年代的空间满了之后，同样会发生GC回收，称为Major GC，也有的叫Full GC，此时底层一般用的算法就是这种标记-整理算法。

新生代的GC: Minor GC

老生代的GC: Major GC（full GC)

4）分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法并没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只

需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

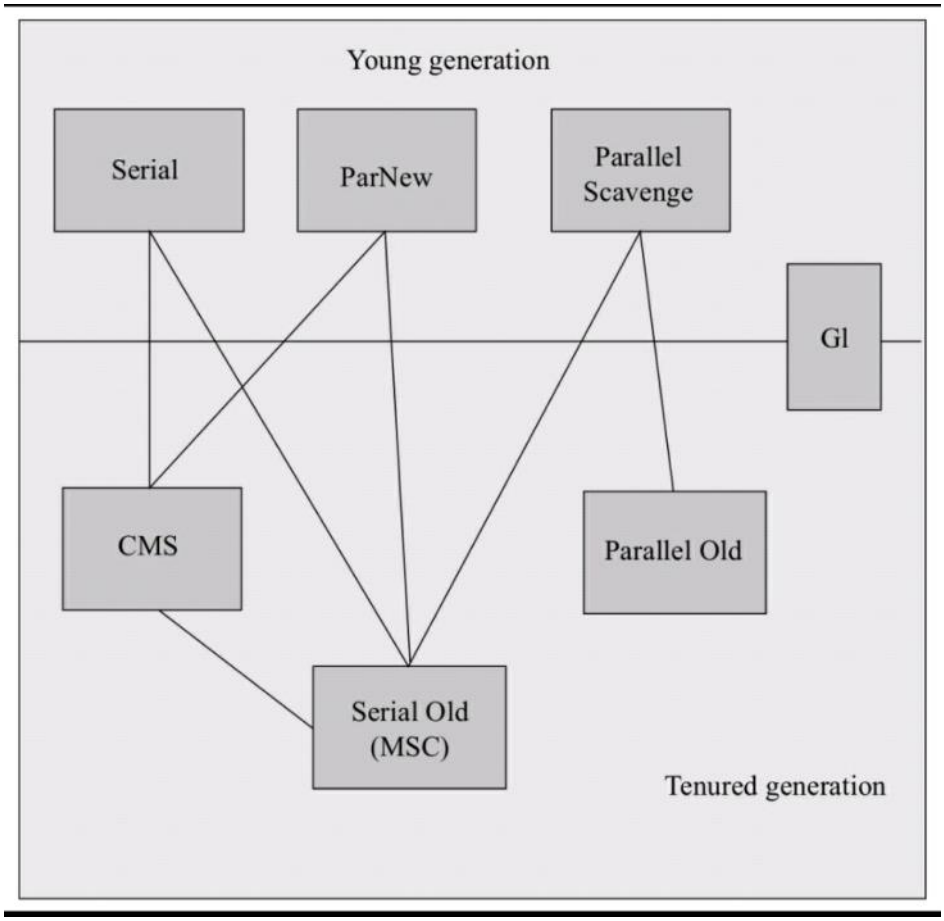
扩展：GC收集器

2018年3月4日 20:10

概述

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。Java虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同的厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器。

这里讨论的收集器基于JDK 1.7 Update 14之后的HotSpot虚拟机（在这个版本中正式提供了商用的G1收集器，之前G1仍处于实验状态）



上图展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用。虚拟机所处的区域，则表示它是属于新生代收集器还是老年代收集器。

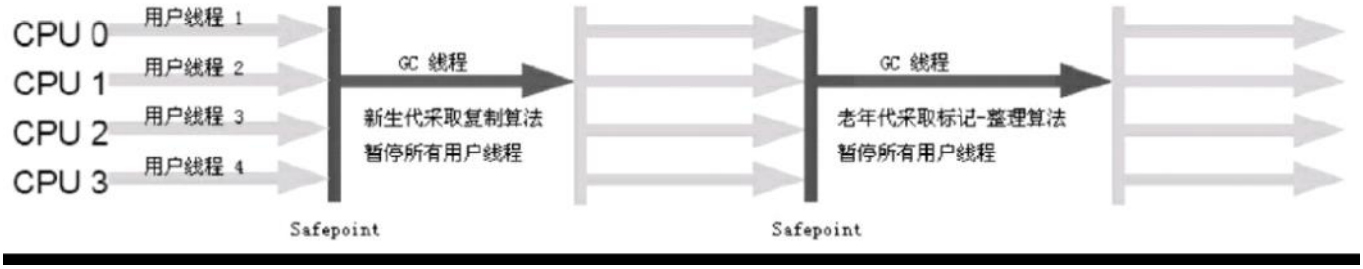
我们先来明确一个观点：虽然我们是在对各个收集器进行比较，但并非为了挑选出一个最好的收集器，更加没有万能的收集器，所以我们选择的只是对具体应用最合适的收集器。这点不需要多加解释就能证明：如果有一种放之四海皆准、任何场景下都适用的完美收集器存在，那

HotSpot虚拟机就没必要实现那么多不同的收集器了。

Serial收集器

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK 1.3.1之前）是虚拟机新生代收集的唯一选择。

这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。“Stop The World”这个名字也许听起来很酷，但这项工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说都是难以接受的。不妨试想一下，要是你的计算机每运行一个小时就会暂停响应5分钟，你会有什么样的心情？下图示意了Serial/Serial Old收集器的运行过程。



对于“Stop The World”带给用户的不良体验，虚拟机的设计者们表示完全理解，但也表示非常委屈：“你妈妈在给你打扫房间的时候，肯定也会让你老老实实地在椅子上或者房间外待着，如果她一边打扫，你一边乱扔纸屑，这房间还能打扫完？”这确实是一个合情合理的矛盾，虽然垃圾收集这项工作听起来和打扫房间属于一个性质的，但实际上肯定还要比打扫房间复杂得多。

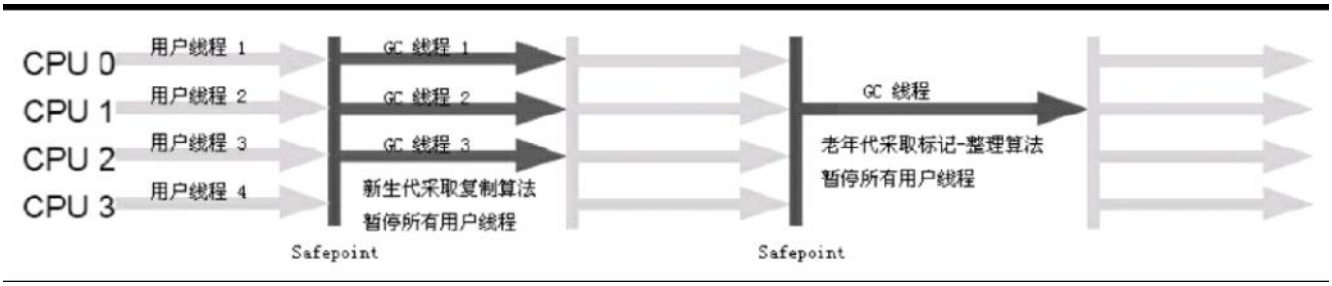
从JDK 1.3开始，一直到现在最新的JDK ，HotSpot虚拟机开发团队为消除或者减少工作线程因内存回收而导致停顿的努力一直在进行着，从Serial收集器到Parallel收集器，再到Concurrent Mark Sweep（CMS）乃至GC收集器的 最前沿成果Garbage First（G1）收集器，我们看到了一个个越来越优秀（也越来越复杂）的收集器的出现，**用户线程的停顿时间在不断缩短，但是仍然没有办法完全消除。**寻找更优秀的垃圾收集器的工作仍在继续！

写到这里，似乎已经把Serial收集器描述成一个“老而无用、食之无味弃之可惜”的鸡肋了，但实际上它也有着优于其他收集器的地方：简单而高效（与其他收集器的单线程比），Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

比如在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本上不会再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。

ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数（例如：-XX:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure等）、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。ParNew收集器的工作过程如下图所示。



ParNew收集器除了多线程收集之外，其他与Serial收集器相比并没有太多创新之处，但其中有一个与性能无关但很重要的原因是，除了Serial收集器外，目前只有它能与CMS收集器配合工作。在JDK 1.5时期，HotSpot推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器——CMS收集器（Concurrent Mark Sweep），这款收集器是HotSpot虚拟机中**第一款真正意义上的并发（Concurrent）收集器**，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作，用前面那个例子的话来说，就是做到了在你的妈妈打扫房间的时候你还能一边往地上扔纸屑。

注意：有必要先解释两个名词：并发和并行。这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，它们可以解释如下。

并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

并发（Concurrent）：指用户线程与垃圾收集线程同时工作，可能会交替执行，用户程序不

必一直等待。

Parallel Scavenge收集器

Parallel Scavenge收集器是一个**新生代**收集器，它也是使用**复制算法**的收集器，又是并行的多线程收集器，看上去和ParNew都一样，那它有什么特别之处呢？

Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，**而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）。所以，也称之为吞吐量优先收集器。**

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，**主要适合在后台运算而不需要太多交互的任务。**
Spark默认用的是Parallel Scavenge收集器。

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX:GCTimeRatio参数。

MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽可能地保证内存回收花费的时间不超过设定值。

不过大家不要认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾收集发生得更频繁一些，原来10秒收集一次、每次停顿100毫秒，现在变成5秒收集一次、每次停顿70毫秒。停顿时间的确在下降，但吞吐量也降下来了。

GCTimeRatio参数的值应当是一个大于0且小于100的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。如果把此参数设置为19，那允许的最大GC时间就占总时间的5%（即1/（1+19）），默认值为99，就是允许最大1%（即1/（1+99））的垃圾收集时间。

GC的吞吐量定义： 用户执行时间/用户执行时间+GC回收时间
用户执行时间：99分钟 GC 1分钟

吞吐量：99%

由于与吞吐量关系密切，**Parallel Scavenge收集器也经常称为“吞吐量优先”收集器。**除上述两个参数之外，Parallel Scavenge收集器还有一个参数-XX:+UseAdaptiveSizePolicy值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）。

如果你对于收集器运作原理不太了解，手工优化存在困难的时候，使用Parallel Scavenge收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择。只需要把基本的内存数据设置好（如-Xmx设置最大堆），然后使用MaxGCPauseMillis参数（更关注最大停顿时间）或GCTimeRatio（更关注吞吐量）参数给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。**自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。**

Parallel Old收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在JDK 1.6中才开始提供的。

Parallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。

CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取**最短回收停顿时间**为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。

从名字（包含“Mark Sweep”）上就可以看出，CMS收集器是基于“标记—清除”算法实

现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤，包括：

1）初始标记（CMS initial mark）

2）并发标记（CMS concurrent mark）

3）重新标记（CMS remark）

4）并发清除（CMS concurrent sweep）

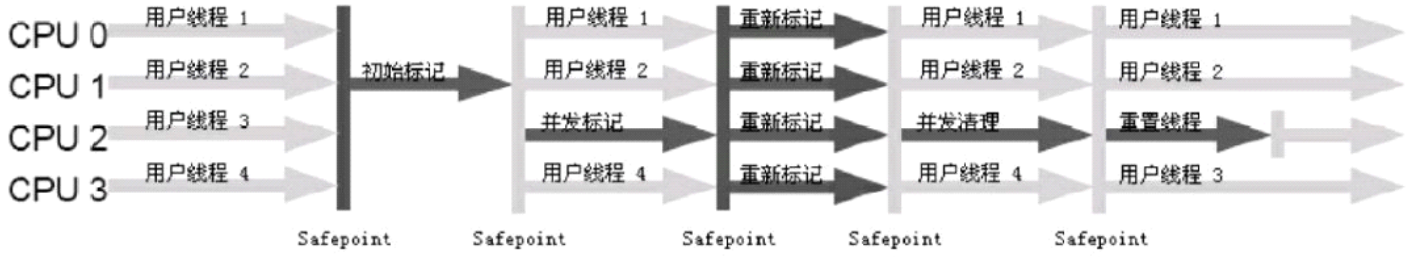
其中，**1）初始标记、2）重新标记这两个步骤仍然需要“Stop The World”。**

初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快。

并发标记阶段就是进行GC RootsTracing的过程。CMS收集器允许在并发标记阶段，用户线程是可以继续工作，所以这个过程可能会导致GC Roots的路径规则发生变化，所以需要下一个阶段重新标记去进行修正。

重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些。

由于整个过程中耗时最长的**并发标记**和**并发清除**过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。如下图：



CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿，Sun公司的一些官方文档中也称之为**并发低停顿收集器**（Concurrent Low Pause Collector）。

但是CMS还远达不到完美的程度，它有以下3个明显的缺点：

1）CMS收集器对CPU资源非常敏感。其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。CMS默认启动的回收线程数是（CPU数量+

3）/4，也就是当CPU在4个以上时，并发回收时垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加而下降。但是当CPU不足4个（譬如2个）时，CMS对用户程序的影响就可能变得很大，如果本来CPU负载就比较大，还分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了50%，其实也让人无法接受。

为了应付这种情况，虚拟机提供了一种称为“增量式并发收集器”（Incremental Concurrent Mark Sweep/i-CMS）的CMS收集器变种，所做的事情和单CPU年代PC机操作系统使用抢占式来模拟多任务机制的思想一样，就是在并发标记、清理的时候让GC线程、用户线程交替运行，尽量减少GC线程的独占资源的时间，这样整个垃圾收集的过程会更长，但对用户程序的影响就会显得少一些，也就是速度下降没有那么明显。实践证明，增量时的CMS收集器效果很一般，在目前版本中，i-CMS已经被声明为“deprecated”，即不再提倡用户使用。

2）此外，CMS收集器无法处理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。

正是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在JDK 1.5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，在JDK 1.6中，CMS收集器的启动阈值已经提升至92%。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CM SInitiatingOccupancyFraction设置得太高很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

3）还有最后一个缺点，CMS是一款基于“标记—”清除算法实现的收集器，这就意味着收集结束时可能会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提

前触发一次Full GC。为了解决这个问题，CMS收集器提供了一个-XX:+UseCMSCompactAtFullCollection开关参数（默认就是开启的），用于在CMS收集器顶不住要进行FullGC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。

扩展：Spark调优—下篇

2018年2月19日 15:49

Spark的GC调优

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

在运行Spark应用时，有些问题是由于GC所带来的，例如垃圾回收时间久、程序长时间无响应，甚至造成程序崩溃或者作业失败。

按照经验来说，当我们配置垃圾收集器时，主要有两种策略——Parallel GC和CMS GC。

前者注重更高的吞吐量，而后者则注重更低的延迟。两者似乎是鱼和熊掌，不能兼得。在实际应用中，我们只能根据应用对性能瓶颈的侧重性，来选取合适的垃圾收集器。例如，当我们运行需要有实时响应的场景的应用时，我们一般选用CMS GC，而运行一些离线分析程序时，则选用Parallel GC。

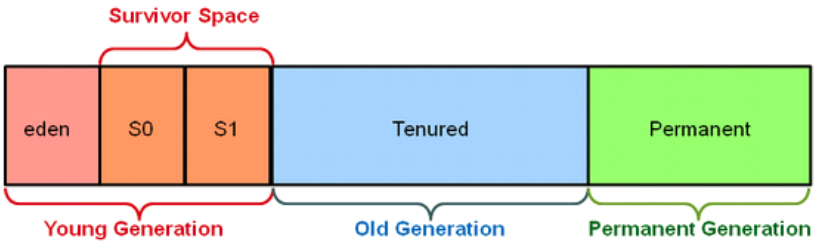
那么对于Spark这种既支持流式计算，又支持传统的批处理运算的计算框架来说，是否存在一组通用的配置选项呢？

通常CMS GC是企业比较常用的GC配置方案，并在长期实践中取得了比较好的效果。例如对于进程中若存在大量寿命较长的对象，Parallel GC经常带来较大的性能下降。因此，即使是批处理的程序也能从CMS GC中获益。不过，在从1.7开始的HOTSPOT JVM中，我们发现了一个新的GC设置项：**Garbage-First GC(G1 GC)**。Oracle将其定位为CMS GC的长期演进，这让我们重燃了鱼与熊掌兼得的希望！

GC算法原理

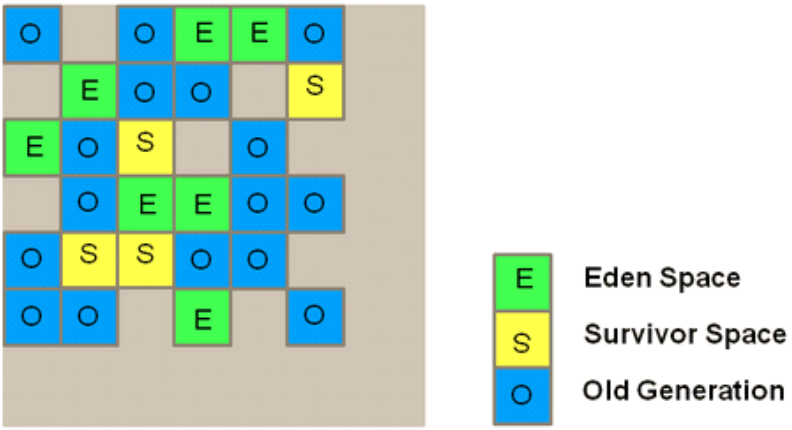
在传统JVM内存管理中，我们把Heap空间分为Young/Old两个分区，Young分区又包括一个Eden和两个Survivor分区，如下图所示。新产生的对象首先会被存放在Eden区，而每次minor GC发生时，JVM一方面将Eden分区内存活的对象拷贝到一个空的Survivor分区，另一方面将另一个正在被使用的Survivor分区中的存活对象也拷贝到空的Survivor分区内。

在此过程中，JVM始终保持一个Survivor分区处于全空的状态。一个对象在两个Survivor之间的拷贝到一定次数后，如果还是存活的，就将其拷入Old分区。当Old分区没有足够空间时，GC会停下所有程序线程，进行Full GC，即对Old区中的对象进行整理。注意：Full GC时，所有线程都暂停，所以这个阶段被称为Stop-The-World(STW)，也是大多数GC算法中对性能影响最大的部分。



而G1 GC则完全改变了这一传统思路。它将整个Heap分为若干个预先设定的小区域块，每个区域块内部不再进行新旧分区， 而是将整个区域块标记为Eden/Survivor/Old。当创建新对象时，它首先被存放到某一个可用区块（Region）中。当该区块满了，JVM就会创建新的区块存放对象。当发生minor GC时，JVM将一个或几个区块中存活的对象拷贝到一个新的区块中，并在空余的空间中选择几个全新区块作为新的Eden分区。当所有区域中都有存活对象，找不到全空区块时，才发生Full GC。而在标记存活对象时，G1使用RememberSet的概念，将每个分区外指向分区内的引用记录在该分区的RememberSet中，避免了对整个Heap的扫描，使得各个分区的GC更加独立。

在这样的背景下，我们可以看出G1 GC大大提高了触发Full GC时的Heap占用率，同时也使得Minor GC的暂停时间更加可控，**对于内存较大的环境非常友好**



关于Hotspot JVM所支持的完整的GC参数列表，可以参见Oracle官方的文档中对部分参数的解释。

Spark的内存管理

Spark的核心概念是RDD，实际运行中内存消耗都与RDD密切相关。Spark允许用户将应用中重复使用的RDD数据持久化缓存起来，从而避免反复计算的开销，而RDD的持久化形态之一就是将全部或者部分数据缓存在JVM的Heap中。当我们观察到GC延迟影响效率时，应当先检查Spark应用本身是否有效利用有限的内存空间。RDD占用的内存空间比较少的话，程序运行的heap空间也会比较宽松，GC效率也会相应提高；而RDD如果占用大量空间的话，则会带来巨大的性能损失。

下面从某个用户案例来说明：

这个应用其本质就是一个简单的迭代计算。而每次迭代计算依赖于上一次的迭代结果，因此每次迭代结果都会被主动持久化到内存空间中。当运行用户程序时，我们观察到随着迭代次数的增加，进程占用的内存空间不断快速增长，GC问题越来越突出。

造成这个问题的原因是没有及时释放掉不再使用的RDD，从而造成了内存空间不断增长，触发了更多GC执行。

迭代轮数	单次迭代缓存大小	总缓存大小（优化前）	总缓存大小（优化后）
初始化	4.3GB	4.3GB	4.3GB
1	8.2GB	12.5GB	8.2GB
2	98.8GB	111.3GB	98.8GB
3	90.8GB	202.1GB	90.8GB

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

选择垃圾收集器

在解决了应用本身的问题之后，我们就要开始针对Spark应用的GC调优了。

Spark默认使用的是Parallel GC。经调研我们发现，Parallel GC常常受困于Full GC，而每次Full GC都给性能带来了较大的下降。而Parallel GC可以进行参数调优的空间也非常有限，我们只能通过调节一些基本参数来提高性能，如各年代分区大小比例、进入老年代前的拷贝次数等。而且这些调优策略只能推迟Full GC的到来，如果是长期运行的应用，Parallel GC调优的意义就非常有限了。

Configuration Options	-XX:+UseParallelGC -XX:+UseParallelOldGC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -Xms88g -Xmx88g

G1 GC的配置

Configuration Options	-XX:+UseG1GC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark -Xms88g -Xmx88g

大多数情况下，最大的性能下降是由Full GC导致的，G1 GC也不例外，所以当使用G1 GC时，需要根据一定的实际情况进行参数配置，这需要很丰富的工作经验和运维经验，以下仅提供一些处理思路。

比如G1 GC收集器在将某个需要垃圾回收的分区进行回收时，无法找到一个能将其中存活对象拷贝过去的空闲分区。这种情况被称为Evacuation Failure，常常会引发Full GC。对于这种情况，我们常见的处理办法有两种：

1. 将InitiatingHeapOccupancyPercent参数调低（默认值是45），可以使G1 GC收集器更早开始Mixed GC（Minor GC)；但另一方面，会增加GC发生频率。（启动并发GC周期时的堆内存占用百分比. G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示"一直执行GC循环". 默认值为 45.）
   降低此值，会提高Minor GC的频率，但是会推迟Full GC的到来。

2. 提高ConcGCThreads的值，在Mixed GC阶段投入更多的并发线程，争取提高每次暂停的效率。但是此参数会占用一定的有效工作线程资源。

调试这两个参数可以有效降低Full GC出现的概率。Full GC被消除之后，最终的性能获得了大幅提升。

此外，可能还会遇到这样的情况：出现了一些比G1的一个分区的一半更大的对象。对于这些对象，G1会专门在Heap上开出一个个Humongous Area来存放，每个分区只放一个对象。但是申请这么大的空间是比较耗时的，而且这些区域也仅当Full GC时才进行处理，所以我们要尽量减少这样的对象产生。或者提高G1HeapRegionSize的值减少HumongousArea的创建。（G1HeapRegionSize=n 使用G1时Java堆会被分为大小统一的的区(region)。此参数可以指定每个heap区的大小. 默认值将根据 heap size 算出最优解. 最小值为 1Mb, 最大值为 32Mb.）

不过在内存比较大的时，JVM默认把这个值设到了最大(32M)，此时我们只能通过分析程序本身找到这些对象并且尽量减少这样的对象产生。当然，相信随着G1 GC的发展，在后期的版本中相信这个最大值也会越来越大，毕竟G1号称是在1024～2048个Region时能够获得最佳性能。

Configuration Options	-XX:+UseG1GC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark -Xms88g -Xmx88g -XX:InitiatingHeapOccupancyPercent=35 -XX:ConcGCThread=20

总结

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

扩展：GC的一些配置

2018年2月19日 20:44

常见的4种GC

1. SerialGC

参数-XX:+UseSerialGC

就是Young区和old区都使用serial 垃圾回收算法

2. ParallelGC

参数-XX:+UseParallelGC

Young区：使用Parallel scavenge 回收算法

Old 区：可以使用单线程的或者Parallel 垃圾回收算法，由 -XX:+UseParallelOldGC 来控制

3. CMS

参数-XX:+UseConcMarkSweepGC

Young区：可以使用普通的或者parallel 垃圾回收算法，由参数 -XX:+UseParNewGC来控制

Old 区：只能使用Concurrent Mark Sweep

4. G1

参数：-XX:+UseG1GC

没有young/old区

一些配置解释

- -XX:+UseG1GC 使用 G1 (Garbage First) 垃圾收集器
- -XX:MaxGCPauseMillis=n 设置最大GC停顿时间(GC pause time)指标(target). 这是一个

软性指标(soft goal), JVM 会尽量去达成这个目标.

- -XX:InitiatingHeapOccupancyPercent=n 启动并发GC周期时的堆内存占用百分比. G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示"一直执行GC循环". 默认值为 45.
- -XX:NewRatio=n 新生代与老生代(new/old generation)的大小比例(Ratio). 默认值为 2.
- -XX:SurvivorRatio=n eden/survivor 空间大小的比例(Ratio). 默认值为 8.
- -XX:MaxTenuringThreshold=n 提升年老代的最大临界值(tenuring threshold). 默认值为 15.
- -XX:ParallelGCThreads=n 设置垃圾收集器在并行阶段使用的线程数,默认值随JVM运行的平台不同而不同.
- -XX:ConcGCThreads=n 并发垃圾收集器使用的线程数量. 默认值随JVM运行的平台不同而不同(调节CMS)
- -XX:G1ReservePercent=n 设置堆内存保留为假天花板的总量,以降低提升失败的可能性. 默认值是 10.
- -XX:G1HeapRegionSize=n 使用G1时Java堆会被分为大小统一的的区(region)。此参数可以指定每个heap区的大小. 默认值将根据 heap size 算出最优解. 最小值为 1Mb, 最大值为 32Mb.

手册：Spark配置详解

2018年2月19日 20:58

这些参数皆可在 spark-defaults.conf配置，或者部分可在 sparkconf().set设置

应用程序属性

属性名称	默认值	含义
spark.app.name	(none)	你的应用程序的名字。这将在UI和日志数据中出现
spark.driver.cores	1	driver程序运行需要的cpu内核数
spark.driver.maxResultSize	1g	每个Spark action(如collect)所有分区的序列化结果的总大小限制。设置的值应该不小于1m，0代表没有限制。如果总大小超过这个限制，程序将会终止。大的限制值可能导致driver出现内存溢出错误（依赖于spark.driver.memory和JVM中对象的内存消耗）。
spark.driver.memory	512m	driver进程使用的内存数
spark.executor.memory	512m	每个executor进程使用的内存数。和JVM内存串拥有相同的格式（如512m,2g）
spark.extraListeners	(none)	注册监听器，需要实现SparkListener
spark.local.dir	/tmp	Spark中暂存空间的使用目录。在Spark1.0以及更高的版本中，这个属性被SPARK_LOCAL_DIRS环境变量覆盖。
spark.logConf	false	当SparkContext启动时，将有效的SparkConf记录为INFO。
spark.master	(none)	集群管理器连接的地方

运行环境

属性名称	默认值	含义
spark.driver.extraClassPath	(none)	附加到driver的classpath的额外的classpath实体。
spark.driver.extraJavaOptions	(none)	传递给driver的JVM选项字符串。例如GC设置或者其它日志设置。注意，在这个选项中设置Spark属性或者堆大小是不合法的。Spark属性需要用--driver-class-path设置。
spark.driver.extraLibraryPath	(none)	指定启动driver的JVM时用到的库路径
spark.driver.userClassPathFirst	false	(实验性)当在driver中加载类时，是否用户添加的jar比Spark自己的jar优先级高。这个属性可以降低Spark依赖和用户依赖的

		冲突。它现在还是一个实验性的特征。
spark.executor.extraClassPath	(none)	附加到executors的classpath的额外的classpath实体。这个设置存在的主要目的是Spark与旧版本的向后兼容问题。用户一般不用设置这个选项
spark.executor.extraJavaOptions	(none)	传递给executors的JVM选项字符串。例如GC设置或者其它日志设置。注意，在这个选项中设置Spark属性或者堆大小是不合法的。Spark属性需要用SparkConf对象或者spark-submit脚本用到的spark-defaults.conf文件设置。堆内存可以通过spark.executor.memory设置
spark.executor.extraLibraryPath	(none)	指定启动executor的JVM时用到的库路径
spark.executor.logs.rolling.maxRetainedFiles	(none)	设置被系统保留的最近滚动日志文件的数量。更老的日志文件将被删除。默认没有开启。
spark.executor.logs.rolling.size.maxBytes	(none)	executor日志的最大滚动大小。默认情况下没有开启。值设置为字节
spark.executor.logs.rolling.strategy	(none)	设置executor日志的滚动(rolling)策略。默认情况下没有开启。可以配置为time和size。对于time，用spark.executor.logs.rolling.time.interval设置滚动间隔；对于size，用spark.executor.logs.rolling.size.maxBytes设置最大的滚动大小
spark.executor.logs.rolling.time.interval	daily	executor日志滚动的时间间隔。默认情况下没有开启。合法的值是daily, hourly, minutely以及任意的秒。
spark.files.userClassPathFirst	false	(实验性)当在Executors中加载类时，是否用户添加的jar比Spark自己的jar优先级高。这个属性可以降低Spark依赖和用户依赖的冲突。它现在还是一个实验性的特征。
spark.python.worker.memory	512m	在聚合期间，每个python worker进程使用的内存数。在聚合期间，如果内存超过了这个限制，它将会将数据塞进磁盘中
spark.python.profile	false	在Python worker中开启profiling。通过sc.show_profiles()展示分析结果。或者在driver退出前展示分析结果。可以通过sc.dump_profiles(path)将结果dump到磁盘中。如果一些分析结果已经手动展示，那么在driver退出前，它们再不会自动展示
spark.python.profile.dump	(none)	driver退出前保存分析结果的dump文件的目录。每个RDD都会分别dump一个文件。可以通过ptats.Stats()加载这些文件。如果指定了这个属性，分析结果不会自动展示
spark.python.worker.reuse	true	是否重用python worker。如果是，它将使用固定数量的Python workers，而不需要为每个任务fork()一个Python进

		程。如果有一个非常大的广播，这个设置将非常有用。因为，广播不需要为每个任务从JVM到Python worker传递一次
spark.executorEnv.[EnvironmentVariableName]	(none)	通过EnvironmentVariableName添加指定的环境变量到executor进程。用户可以指定多个EnvironmentVariableName，设置多个环境变量
spark.mesos.executor.home	driver side SPARK_HOME	设置安装在Mesos的executor上的Spark的目录。默认情况下，executors将使用driver的Spark本地（home）目录，这个目录对它们不可见。注意，如果没有通过 spark.executor.uri 指定Spark的二进制包，这个设置才起作用
spark.mesos.executor.memoryOverhead	executor memory * 0.07, 最小384m	这个值是spark.executor.memory的补充。它用来计算mesos任务的总内存。另外，有一个7%的硬编码设置。最后的值将选择spark.mesos.executor.memoryOverhead或者spark.executor.memory的7%二者之间的大者

Shuffle行为

属性名称	默认值	含义
spark.reducer.maxMbInFlight	48	从递归任务中同时获取的map输出数据的最大大小（mb）。因为每一个输出都需要我们创建一个缓存用来接收，这个设置代表每个任务固定的内存上限，所以除非你有更大的内存，将其设置小一点
spark.shuffle.blockTransferService	netty	实现用来在executor直接传递shuffle和缓存块。有两种可用的实现：netty和nio。基于netty的块传递在具有相同的效率情况下更简单
spark.shuffle.compress	true	是否压缩map操作的输出文件。一般情况下，这是一个好的选择。
spark.shuffle.consolidateFiles	false	如果设置为”true”，在shuffle期间，合并的中间文件将会被创建。创建更少的文件可以提供文件系统的shuffle的效率。这些shuffle都伴随着大量递归任务。当用ext4和dfs文件系统时，推荐设置为”true”。在ext3中，因为文件系统的限制，这个选项可能机器（大于8核）降低效率
spark.shuffle.file.buffer.kb	32	每个shuffle文件输出流内存内缓存的大小，单位是kb。这个缓存减少了创建只中间shuffle文件中磁盘搜索和系统访问的数量
spark.shuffle.io.maxRetries	3	Netty only，自动重试次数
spark.shuffle.io.numConnectionsPerPeer	1	Netty only
spark.shuffle.io.preferDirectBufs	true	Netty only

spark.shuffle.io.retryWait	5	Netty only
spark.shuffle.manager	sort	它的实现用于shuffle数据。有两种可用的实现：sort和hash。基于sort的shuffle有更高的内存使用率
spark.shuffle.memoryFraction	0.2	如果spark.shuffle.spill为true，shuffle中聚合和合并组操作使用的java堆内存占总内存的比重。在任何时候，shuffles使用的所有内存内maps的集合大小都受这个限制的约束。超过这个限制，spilling数据将会保存到磁盘上。如果spilling太过频繁，考虑增大这个值
spark.shuffle.sort.bypassMergeThreshold	200	(Advanced) In the sort-based shuffle manager, avoid merge-sorting data if there is no map-side aggregation and there are at most this many reduce partitions
spark.shuffle.spill	true	如果设置为”true”，通过将多出的数据写入磁盘来限制内存数。通过spark.shuffle.memoryFraction来指定spilling的阈值
spark.shuffle.spill.compress	true	在shuffle时，是否将spilling的数据压缩。压缩算法通过spark.io.compression.codec指定。

Spark UI

属性名称	默认值	含义
spark.eventLog.compress	false	是否压缩事件日志。需要spark.eventLog.enabled为true
spark.eventLog.dir	file:///tmp/spark-events	Spark事件日志记录的基本目录。在这个基本目录下，Spark为每个应用程序创建一个子目录。各个应用程序记录日志到直到的目录。用户可能想设置这为统一的地点，像HDFS一样，所以历史文件可以通过历史服务器读取
spark.eventLog.enabled	false	是否记录Spark的事件日志。这在应用程序完成后，重新构造web UI是有用的
spark.ui.killEnabled	true	运行在web UI中杀死stage和相应的job
spark.ui.port	4040	你的应用程序dashboard的端口。显示内存和工作量数据
spark.ui.retainedJobs	1000	在垃圾回收之前，Spark UI和状态API记住的job数
spark.ui.retainedStages	1000	在垃圾回收之前，Spark UI和状态API记住的stage数

压缩和序列化

属性名称	默认值	含义
spark.broadcast.compress	true	在发送广播变量之前是否压缩它

spark.closure.serializer	org.apache.spark.serializer.JavaSerializer	闭包用到的序列化类。目前只支持java序列化器
spark.io.compression.codec	snappy	压缩诸如RDD分区、广播变量、shuffle输出等内部数据的编码解码器。默认情况下，Spark提供了三种选择：lz4、lzf和snappy，你也可以用完整的类名来制定。
spark.io.compression.lz4.block.size	32768	LZ4压缩中用到的块大小。降低这个块的大小也会降低shuffle内存使用率
spark.io.compression.snappy.block.size	32768	Snappy压缩中用到的块大小。降低这个块的大小也会降低shuffle内存使用率
spark.kryo.classesToRegister	(none)	如果你用Kryo序列化，给定的用逗号分隔的自定义类名列表表示要注册的类
spark.kryo.referenceTracking	true	当用Kryo序列化时，跟踪是否引用同一对象。如果你的对象图有环，这是必须的设置。如果他们包含相同对象的多个副本，这个设置对效率是有用的。如果你知道不在这两个场景，那么可以禁用它以提高效率
spark.kryo.registrationRequired	false	是否需要注册为Kyro可用。如果设置为true，然后如果一个没有注册的类序列化，Kyro会抛出异常。如果设置为false，Kryo将会同时写每个对象和其非注册类名。写类名可能造成显著地性能瓶颈。
spark.kryo.registrator	(none)	如果你用Kryo序列化，设置这个类去注册你的自定义类。如果你需要用自定义的方式注册你的类，那么这个属性是有用的。否则spark.kryo.classesToRegister会更简单。它应该设置一个继承自KryoRegistrator的类
spark.kryoserializer.buffer.max.mb	64	Kryo序列化缓存允许的最大值。这个值必须大于你尝试序列化的对象
spark.kryoserializer.buffer.mb	0.064	Kyro序列化缓存的大小。这样worker上的每个核都有一个缓存。如果有需要，缓存会涨到spark.kryoserializer.buffer.max.mb设置的值那么大。
spark.rdd.compress	true	是否压缩序列化的RDD分区。在花费一些额外的CPU时间的同时节省大量的空间
spark.serializer	org.apache.spark.serializer.JavaSerializer	序列化对象使用的类。默认的Java序列化类可以序列化任何可序列化的java对象但是它很慢。所有我们建议用org.apache.spark.serializer.KryoSerializer
spark.serializer.objectStreamReset	100	当用org.apache.spark.serializer.JavaSerializer序列化时，序列化器通过缓存对象防止写多余的数据，然而这会造成这些对象的垃圾回收停止。通过请求’ reset’ ，你从序列化器中

		flush这些信息并允许收集老的数据。为了关闭这个周期性的reset，你可以将值设为-1。默认情况下，每一百个对象reset一次

运行时行为

属性名称	默认值	含义
spark.broadcast.blockSize	4096	TorrentBroadcastFactory传输的块大小，太大值会降低并发，太小的值会出现性能瓶颈
spark.broadcast.factory	org.apache.spark.broadcast.TorrentBroadcastFactory	broadcast实现类
spark.cleaner.ttl	(infinite)	spark记录任何元数据（stages生成、task生成等）的持续时间。定期清理可以确保将超期的元数据丢弃，这在运行长时间任务是很有用的，如运行7*24的sparkstreaming任务。RDD持久化在内存中的超期数据也会被清理
spark.default.parallelism	本地模式：机器核数；Mesos：8；其他：max(executor的core，2)	如果用户不设置，系统使用集群中运行shuffle操作的默认任务数（groupByKey、 reduceByKey等）
spark.executor.heartbeatInterval	10000	executor 向 the driver 汇报心跳的时间间隔，单位毫秒
spark.files.fetchTimeout	60	driver 程序获取通过SparkContext.addFile()添加的文件时的超时时间，单位秒
spark.files.useFetchCache	true	获取文件时是否使用本地缓存
spark.files.overwrite	false	调用SparkContext.addFile()时候是否覆盖文件
spark.hadoop.cloneConf	false	每个task是否克隆一份hadoop的配置文件
spark.hadoop.validateOutputSpecs	true	是否校验输出
spark.storage.memoryFraction	0.6	Spark内存缓存的堆大小占用总内存比例，该值不能大于老年代内存大小，默认值为0.6，但是，如果你手动设置老年代大小，你可以增加该值
spark.storage.memoryMapThreshold	2097152	内存块大小

spark.storage.unrollFraction	0.2	Fraction of spark.storage.memoryFraction to use for unrolling blocks in memory.
spark.tachyonStore.baseDir	System.getProperty（“java.io.tmpdir”）	Tachyon File System临时目录
spark.tachyonStore.url	tachyon://localhost:19998	Tachyon File System URL

网络

属性名称	默认值	含义
spark.driver.host	(local hostname)	driver监听的主机名或者IP地址。这用于和executors以及独立的master通信
spark.driver.port	(random)	driver监听的接口。这用于和executors以及独立的master通信
spark.fileserver.port	(random)	driver的文件服务器监听的端口
spark.broadcast.port	(random)	driver的HTTP广播服务器监听的端口
spark.replClassServer.port	(random)	driver的HTTP类服务器监听的端口
spark.blockManager.port	(random)	块管理器监听的端口。这些同时存在于driver和executors
spark.executor.port	(random)	executor监听的端口。用于与driver通信
spark.port.maxRetries	16	当绑定到一个端口，在放弃前重试的最大次数
spark.akka.frameSize	10	在”control plane”通信中允许的最大消息大小。如果你的任务需要发送大的结果到driver中，调大这个值
spark.akka.threads	4	通信的actor线程数。当driver有很多CPU核时，调大它是有用的
spark.akka.timeout	100	Spark节点之间的通信超时。单位是秒
spark.akka.heartbeat.pauses	6000	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). Acceptable heart beat pause in seconds for akka. This can be used to control sensitivity to gc pauses. Tune this in combination of spark.akka.heartbeat.interval and spark.akka.failure-detector.threshold if you need to.

spark.akka.failure-detector.threshold	300.0	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). This maps to akka’ s akka.remote.transport-failure-detector.threshold. Tune this in combination of spark.akka.heartbeat.pauses and spark.akka.heartbeat.interval if you need to.
spark.akka.heartbeat.interval	1000	This is set to a larger value to disable failure detector that comes inbuilt akka. It can be enabled again, if you plan to use this feature (Not recommended). A larger interval value in seconds reduces network overhead and a smaller value (~ 1 s) might be more informative for akka’ s failure detector. Tune this in combination of spark.akka.heartbeat.pauses and spark.akka.failure-detector.threshold if you need to. Only positive use case for using failure detector can be, a sensistive failure detector can help evict rogue executors really quick. However this is usually not the case as gc pauses and network lags are expected in a real Spark cluster. Apart from that enabling this leads to a lot of exchanges of heart beats between nodes leading to flooding the network with those.

调度相关属性

属性名称	默认值	含义
spark.task.cpus	1	为每个任务分配的内核数
spark.task.maxFailures	4	Task的最大重试次数
spark.scheduler.mode	FIFO	Spark的任务调度模式，还有一种Fair模式
spark.cores.max		当应用程序运行在Standalone集群或者粗粒度共享模式Mesos集群时，应用程序向集群请求的最大CPU内核总数（不是指每台机器，而是整个集群）。如果不设置，对于Standalone集群将使用spark.deploy.defaultCores中数值，而Mesos将使用集群中可用的内核
spark.mesos.coarse	False	如果设置为true，在Mesos集群中运行时使用粗粒度共享模式
spark.speculation	False	以下几个参数是关于Spark推测执行机制的相关参数。此参数设定是否使用推测执行机制，如果设置为true则spark使用推测执行机制，对于Stage中拖后腿的Task在其他节点中重新启动，并将最先完成的Task的计算结果最为最终结果

spark.speculation.interval	100	Spark多长时间进行检查task运行状态用以推测，以毫秒为单位
spark.speculation.quantile		推测启动前，Stage必须要完成总Task的百分比
spark.speculation.multiplier	1.5	比已完成Task的运行速度中位数慢多少倍才启用推测
spark.locality.wait	3000	以下几个参数是关于Spark数据本地性的。本参数是以毫秒为单位启动本地数据task的等待时间，如果超出就启动下一本地优先级别的task。该设置同样可以应用到各优先级别的本地性之间（本地进程 -> 本地节点 -> 本地机架 -> 任意节点 ），当然，也可以通过spark.locality.wait.node等参数设置不同优先级别的本地性
spark.locality.wait.process	spark.locality.wait	本地进程级别的本地等待时间
spark.locality.wait.node	spark.locality.wait	本地节点级别的本地等待时间
spark.locality.wait.rack	spark.locality.wait	本地机架级别的本地等待时间
spark.scheduler.revive.interval	1000	复活重新获取资源的Task的最长时间间隔（毫秒），发生在Task因为本地资源不足而将资源分配给其他Task运行后进入等待时间，如果这个等待时间内重新获取足够的资源就继续计算

Dynamic Allocation

属性名称	默认值	含义
spark.dynamicAllocation.enabled	false	是否开启动态资源搜集
spark.dynamicAllocation.executorIdleTimeout	600	
spark.dynamicAllocation.initialExecutors	spark.dynamicAllocation.minExecutors	
spark.dynamicAllocation.maxExecutors	Integer.MAX_VALUE	
spark.dynamicAllocation.minExecutors	0	
spark.dynamicAllocation.schedulerBacklogTimeout	5	
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout	schedulerBacklogTimeout	

安全

属性名称	默认值	含义
spark.authenticate	false	是否Spark验证其内部连接。如果不是运行在YARN上，请看spark.authenticate.secret
spark.authenticate.secret	None	设置Spark两个组件之间的密匙验证。如果不是运行在YARN上，但是需要验证，这个选项必须设置
spark.core.connection.auth.wait.timeout	30	连接时等待验证的实际。单位为秒
spark.core.connection.ack.wait.timeout	60	连接等待回答的时间。单位为秒。为了避免不希望的超时，你可以设置更大的值
spark.ui.filters	None	应用到Spark web UI的用于过滤类名的逗号分隔的列表。过滤器必须是标准的javax servlet Filter。通过设置java系统属性也可以指定每个过滤器的参数。spark.<class name of filter>.params='param1=value1,param2=value2'。例如-Dspark.ui.filters=com.test.filter1、-Dspark.com.test.filter1.params='param1=foo,param2=testing'
spark.acls.enable	false	是否开启Spark acls。如果开启了，它检查用户是否有权限去查看或修改job。UI利用使用过滤器验证和设置用户
spark.ui.view.acls	empty	逗号分隔的用户列表，列表中的用户有查看Spark web UI的权限。默认情况下，只有启动Spark job的用户有查看权限
spark.modify.acls	empty	逗号分隔的用户列表，列表中的用户有修改Spark job的权限。默认情况下，只有启动Spark job的用户有修改权限
spark.admin.acls	empty	逗号分隔的用户或者管理员列表，列表中的用户或管理员有查看和修改所有Spark job的权限。如果你运行在一个共享集群，有一组管理员或开发者帮助debug，这个选项有用

加密

属性名称	默认值	含义
spark.ssl.enabled	false	是否开启ssl
spark.ssl.enabledAlgorithms	Empty	JVM支持的加密算法列表，逗号分隔
spark.ssl.keyPassword	None	
spark.ssl.keyStore	None	
spark.ssl.keyStorePassword	None	
spark.ssl.protocol	None	
spark.ssl.trustStore	None	
spark.ssl.trustStorePassword	None	

Spark Streaming

属性名称	默认值	含义
spark.streaming.blockInterval	200	在这个时间间隔（ms）内，通过Spark Streaming receivers接收的数据在保存到Spark之前，chunk为数据块。推荐的最小值为50ms
spark.streaming.receiver.maxRate	infinite	每秒钟每个receiver将接收的数据的最大记录数。有效的情况下，每个流将消耗至少这个数目的记录。设置这个配置为0或者-1将会不作限制
spark.streaming.receiver.writeAheadLogs.enable	false	Enable write ahead logs for receivers. All the input data received through receivers will be saved to write ahead logs that will allow it to be recovered after driver failures
spark.streaming.unpersist	true	强制通过Spark Streaming生成并持久化的RDD自动从Spark内存中非持久化。通过Spark Streaming接收的原始输入数据也将清除。设置这个属性为false允许流应用程序访问原始数据和持久化RDD，因为它们没有被自动清除。但是它会造成更高的内存花费

集群管理

Spark On YARN

属性名称	默认值	含义
spark.yarn.am.memory	512m	client 模式时，am的内存大小；cluster模式时，使用spark.driver.memory变量
spark.driver.cores	1	claster模式时，driver使用的cpu核数，这时候driver运行在am中，其实也就是am和核数；client模式时，使用spark.yarn.am.cores变量
spark.yarn.am.cores	1	client 模式时，am的cpu核数
spark.yarn.am.waitTime	100000	启动时等待时间
spark.yarn.submit.file.replication	3	应用程序上传到HDFS的文件的副本数
spark.yarn.preserve.staging.files	False	若为true，在job结束后，将stage相关的文件保留而不是删除
spark.yarn.scheduler.heartbeat.interval-ms	5000	Spark AppMaster发送心跳信息给YARN RM的时间间隔
spark.yarn.max.executor.failures	2倍于executor数，最小值3	导致应用程序宣告失败的最大executor失败次数
spark.yarn.applicatio	10	RM等待Spark AppMaster启动重试次数，也就是

nMaster.waitTries		SparkContext初始化次数。超过这个数值，启动失败
spark.yarn.historyServer.address		Spark history server的地址（不要加 http:// ）。这个地址会在Spark应用程序完成后提交给YARN RM，然后RM将信息从RM UI写到history server UI上。
spark.yarn.dist.archives	(none)	
spark.yarn.dist.files	(none)	
spark.executor.instances	2	executor实例个数
spark.yarn.executor.memoryOverhead	executorMemory * 0.07, with minimum of 384	executor的堆内存大小设置
spark.yarn.driver.memoryOverhead	driverMemory * 0.07, with minimum of 384	driver的堆内存大小设置
spark.yarn.am.memoryOverhead	AM memory * 0.07, with minimum of 384	am的堆内存大小设置，在client模式时设置
spark.yarn.queue	default	使用yarn的队列
spark.yarn.jar	(none)	
spark.yarn.access.namenodes	(none)	
spark.yarn.appMasterEnv.[EnvironmentVariableName]	(none)	设置am的环境变量
spark.yarn.containerLauncherMaxThreads	25	am启动executor的最大线程数
spark.yarn.am.extraJavaOptions	(none)	
spark.yarn.maxAppAttempts	yarn.resourcemanager.am.max-attempts in YARN	am重试次数