【译】Spark调优

原文地址：http://spark.incubator.apache.org/docs/0.7.3/tuning.html

译文地址：http://www.oschina.net/translate/spark-tuning

译文作者：http://my.oschina.net/u/559738

我的英语水平有限，此文是上传到oschina网站上由一位热心的网友（@sdzzboy）翻译的。oschina上的学习氛围不错，提出的问题会有很多热心的大牛帮忙解答，国内这样的社区还是比较少的，期待国内涌现例如google group，github,stack overflow这类的学习氛围的社区站点。

以下为正文内容：

因为大部分Spark程序都具有"内存计算"的天性，所以集群中的所有资源：CPU、网络带宽或者是内存都有可能成为Spark程序的瓶颈。通常情况下，如果数据完全加载到内存那么网络带宽就会成为瓶颈，但是你仍然需要对程序进行优化，例如采用序列化的方式保存RDD数据（Resilient Distributed Datasets），以便减少内存使用。该文章主要包含两个议题：数据序列化和内存优化，数据序列化不但能提高网络性能还能减少内存使用。与此同时，我们还讨论了其他几个的小议题。

数据序列化

序列化对于提高分布式程序的性能起到非常重要的作用。一个不好的序列化方式（如序列化模式的速度非常慢或者序列化结果非常大）会极大降低计算速度。很多情况下，这是你优化Spark应用的第一选择。Spark试图在方便和性能之间获取一个平衡。Spark提供了两个序列化类库：

- Java 序列化：在默认情况下，Spark采用Java的ObjectOutputStream序列化一个对象。该方式适用于所有实现了java.io.Serializable的类。通过继承 java.io.Externalizable，你能进一步控制序列化的性能。Java序列化非常灵活，但是速度较慢，在某些情况下序列化的结果也比较大。
- Kryo序列化：Spark也能使用Kryo（版本2）序列化对象。Kryo不但速度极快，而且产生的结果更为紧凑（通常能提高10倍）。Kryo的缺点是不支持所有类型，为了更好的性能，你需要提前注册程序中所使用的类（class）。
- 你可以在创建SparkContext之前，通过调用System.setProperty("spark.serializer", "spark.KryoSerializer")，将序列化方式切换成Kryo。Kryo不能成为默认方式的唯一原因是需要用户进行注册；但是，对于任何"网络密集型"（network-intensive)的应用，我们都建议采用该方式。

  最后，为了将类注册到Kryo，你需要继承 spark.KryoRegistrator并且设置系统属性spark.kryo.registrator指向该类，如下所示：Kryo 文档描述了很多便于注册的高级选项，例如添加用户自定义的序列化代码。

```
import com.esotericsoftware.kryo.Kryo

class MyRegistrator extends spark.KryoRegistrator {
  override def registerClasses(kryo: Kryo) {
    kryo.register(classOf[MyClass1])
    kryo.register(classOf[MyClass2])
  }
}

// Make sure to set these properties *before* creating a SparkContext!
System.setProperty("spark.serializer", "spark.KryoSerializer")
System.setProperty("spark.kryo.registrator", "mypackage.MyRegistrator")
val sc = new SparkContext(...)
```

- 如果对象非常大，你还需要增加属性spark.kryoserializer.buffer.mb的值。该属性的默认值是32，但是该属性需要足够大以便能够容纳需要序列化的最大对象。

昵称：vincent_hv
园龄：10个月
粉丝：7
关注：1
+加关注

< 2013年10月 >

日	一	二	三	四	五	六
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

最新随笔
1. linux解压zip乱码解决方案
2. 全能系统监控工具dstat
3. 【转】linux sar命令详解
4. 【原】gnome3增加自定义程序快捷方式
5. 【原】Ubuntu13.04安装、卸载Gnome3.8
6. 【原】安装、卸载、查看软件时常用的命令
7. 【原】中文Ubuntu主目录下的文档文件夹改回英文
8. 【原】Ubuntu ATI/Intel双显卡 驱动安装
9. 【原】Ubuntu 12.04 ATI显卡设置双屏显示
10. 【转】Hadoop vs Spark性能对比

随笔分类
Android(8)
Hadoop(2)
Java(20)
JVM(3)
Linux(23)
others(1)
Scala(5)
Spark(20)
数据结构与算法(2)

最后，如果你不注册你的类，Kryo仍然可以工作，但是需要为了每一个对象保存其对应的全类名（full class name),这是非常浪费的。

内存优化

内存优化有三个方面的考虑：对象所占用的内存（你或许希望将所有的数据都加载到内存），访问对象的消耗以及垃圾回收（garbage collection)所占用的开销。

通常，Java对象的访问速度更快，但其占用的空间通常比其内部的属性数据大2-5倍。这主要由以下几方面原因：

- 每一个Java对象都包含一个"对象头"（object header），对象头大约有16字节，包含了指向对象所对应的类(class)的指针等信息以。如果对象本身包含的数据非常少，那么对象头有可能会比对象数据还要大。
- Java String在实际的字符串数据之外，还需要大约40字节的额外开销（因为String将字符串保存在一个Char数组，需要额外保存类似长度等的其他数据）；同时，因为是Unicode编码，每一个字符需要占用两个字节。所以，一个长度为10的字符串需要占用60个字节。
- 通用的集合类，例如HashMap、LinkedList等，都采用了链表数据结构，对于每一个条目（entry）都进行了包装(wrapper)。每一个条目不仅包含对象头，还包含了一个指向下一条目的指针（通常为8字节）。
- 基本类型（primitive type）的集合通常都保存为对应的类，例如java.lang.Integer

该章节讨论如何估算对象所占用的内存以及如何进行改进——通过改变数据结构或者采用序列化方式。然后，我们将讨论如何优化Spark的缓存以及Java内存回收（garbage collection)。

确定内存消耗

确定对象所需要内存大小的最好方法是创建一个RDD，然后将其放入缓存，最后阅读驱动程序（driver program）中SparkContext的日志。日志会告诉你每一部分占用的内存大小；你可以收集该类信息以确定RDD消耗内存的最终大小。日志信息如下所示：

```
INFO BlockManagerMasterActor: Added rdd_0_1 in memory on mbk.local:50311 (size: 717.5KB, free: 332.3 MB)
```

该信息表明RDD0的第一部分消耗717.5KB的内存。

优化数据结构

减少内存使用的第一条途径是避免使用一些增加额外开销的Java特性，例如基于指针的数据结构以对对象进行再包装等。有很多方法：

1. 使用对象数组以及原始类型（primitive type)数组以替代Java或者Scala集合类（collection class)。 fastutil 库为原始数据类型提供了非常方便的集合类，且兼容Java标准类库。
2. 尽可能的避免采用还有指针的嵌套数据结构来保存小对象。
3. 考虑采用数字ID或者枚举类型一边替代String类型的主键。
4. 如果内存少于32G，设置JVM参数-XX:+UseCompressedOops以便将8字节指针修改成4字节。于此同时，在Java 7或者更高版本，设置JVM参数-XX:+UseCompressedStrings以便采用8比特来编码每一个ASCII字符。你可以将这些选项添加到spark-env.sh。

序列化RDD存储

经过上述优化，如果对象还是太大以至于不能有效存放，还有一个减少内存使用的简单方法——序列化，采用RDD持久化API的序列化StorageLevel，例如MEMORY_ONLY_SER。Spark将RDD每一部分都保存为byte数组。序列化带来的唯一缺点是会降低访问速度，因为需要将对象反序列化。如果需要采用序列化的方式缓存数据，我们强烈建议采用Kryo，Kryo序列化结果比Java标准序列化更小（其实比对象内部的原始数据都要小）。

优化内存回收

如果你需要不断的"翻动"程序保存的RDD数据，JVM内存回收就可能成为问题（通常，如果只需进行一次RDD读取然后进行操作是不会带来问题的）。当需要回收旧对象以便为新对象腾内存空间时，JVM需要跟踪所有的Java对象以确定哪些对象是不再需要的。需要记住的一点是，内存回收的代价与对象的数量正相关；因此，使用对象数量更小的数据结构（例如使用int数组而不是LinkedList）能显著降低这种消耗。另外一种更好的方法是采用对象序列化，如上面所描述的一样；这样，RDD的每一部分都会保存为唯一一个对象（一个byte数组）。如果内存回收存在问题，在尝试其他方法之前，首先尝试使用序列化缓存（serialized caching）。

每项任务（task）的工作内存以及缓存在节点的RDD之间会相互影响，这种影响也会带来内存回收问题。下面我们讨论如何为RDD分配空间以便减轻这种影响。

估算内存回收的影响

优化内存回收的第一步是获取一些统计信息，包括内存回收的频率、内存回收耗费的时间等。为了获取这些统计信息，我们可以把参数-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps添加到环境变量SPARK_JAVA_OPTS。设置完成后，Spark作业运行时，我们可以在日志中看到每一次内存回收的信息。注意，这些日志保存在集群的工作节点（work nodes)而不是你的驱动程序（driver program).

优化缓存大小

用多大的内存来缓存RDD是内存回收一个非常重要的配置参数。默认情况下，Spark采用运行内存（executor memory，spark.executor.memory或者SPARK_MEM）的66%来进行RDD缓存。这表明在任务执行期间，有33%的内存可以用来进行对象创建。

如果任务运行速度变慢且JVM频繁进行内存回收，或者内存空间不足，那么降低缓存大小设置可以减少内存消耗。为了将缓存大小修改为50%，你可以调用方法System.setProperty("spark.storage.memoryFraction", "0.5")。结合序列化缓存，使用较小缓存足够解决内存回收的大部分问题。如果你有兴趣进一步优化Java内存回收，请继续阅读下面文章。

内存回收高级优化

为了进一步优化内存回收，我们需要了解JVM内存管理的一些基本知识。

- Java堆（heap）空间分为两部分：新生代和老生代。新生代用于保存生命周期较短的对象；老生代用于保存生命周期较长的对象。
- 新生代进一步划分为三部分[Eden, Survivor1, Survivor2]
- 内存回收过程的简要描述：如果Eden区域已满则在Eden执行minor GC并将Eden和Survivor1中仍然活跃的对象拷贝到Survivor2。然后将Survivor1和Survivor2对换。如果对象活跃的时间已经足够长或者Survivor2区域已满，那么会将对象拷贝到Old区域。最终，如果Old区域消耗殆尽，则执行full GC。

Spark内存回收优化的目标是确保只有长时间存活的RDD才保存到老生代区域；同时，新生代区域足够大以保存生命周期比较短的对象。这样，在任务执行期间可以避免执行full GC。下面是一些可能有用的执行步骤：

- 通过收集GC信息检查内存回收是不是过于频繁。如果在任务结束之前执行了很多次full GC，则表明任务执行的内存空间不足。
- 在打印的内存回收信息中，如果老生代接近消耗殆尽，那么减少用于缓存的内存空间。可这可以通过属性spark.storage.memoryFraction来完成。减少缓存对象以提高执行速度是非常值得的。
- 如果有过多的minor GC而不是full GC，那么为Eden分配更大的内存是有益的。你可以为Eden分配大于任务执行所需要的内存空间。如果Eden的大小确定为E，那么可以通过-Xmn=4/3*E来设置新生代的大小（将内存扩大到4/3是考虑到survivor所需要的空间）。
- 举一个例子，如果任务从HDFS读取数据，那么任务需要的内存空间可以从读取的block数量估算出来。注意，解压后的blcok通常为解压前的2-3倍。所以，如果我们需要同时执行3或4个任务，block的大小为64M，我们可以估算出Eden的大小为4*3*64MB。
- 监控内存回收的频率以及消耗的时间并修改相应的参数设置。

我们的经历表明有效的内存回收优化取决于你的程序和内存大小。 在网上还有很多其他的优化选项， 总体而言有效控制内存回收的频率非常有助于降低额外开销。

其他考虑

并行度

集群不能有效的被利用，除非为每一个操作都设置足够高的并行度。Spark会根据每一个文件的大小自动设置运行在该文件"Map"任务的个数（你也可以通过SparkContext的配置参数来控制）；对于分布式"reduce"任务（例如group by key或者reduce by key），则利用最大RDD的分区数。你可以通过第二个参数传入并行度（阅读文档spark.PairRDDFunctions ）或者通过设置系统参数spark.default.parallelism来改变默认值。通常来讲，在集群中，我们建议为每一个CPU核（core）分配2-3个任务。

Reduce Task的内存使用

有时，你会碰到OutOfMemory错误，这不是因为你的RDD不能加载到内存，而是因为任务执行的数据集过大，例如正在执行groupByKey操作的reduce任务。Spark的"混洗"（shuffle）操作（sortByKey、groupByKey、reduceByKey、join等）为了完成分组会为每一个任务创建哈希表,哈希表有可能非常大。最简单的修复方法是增加并行度，这样，每一个任务的输入会变的更小。Spark能够非常有效的支持段时间任务（例如200ms），因为他会对所有的任务复用JVM，这样能减小任务启动的消耗。所以，你可以放心的使任务的并行度远大于集群的CPU核数。

广播"大变量"

使用SparkContext的 广播功能可以有效减小每一个任务的大小以及在集群中启动作业的消耗。如果任务会使用驱动程序（driver program)中比较大的对象（例如静态查找表），考虑将其变成可广播变量。Spark会在master打印每一个任务序列化后的大小，所以你可以通过它来检查任务是不是过于庞大。通常来讲，大于20KB的任务可能都是值得优化的。

总结

该文指出了Spark程序优化所需要关注的几个关键点——最主要的是数据序列化和内存优化。对于大多数程序而言，采用Kryo框架以及序列化能够解决性能有关的大部分问题。非常欢迎在Spark mailing list提问优化相关的问题。

分类: Spark

« 上一篇：【转】Spark源码分析之-scheduler模块
» 下一篇：【转】Hadoop vs Spark性能对比

posted @ 2013-09-27 09:55 vincent_hv 阅读(15) 评论(0) 编辑 收藏