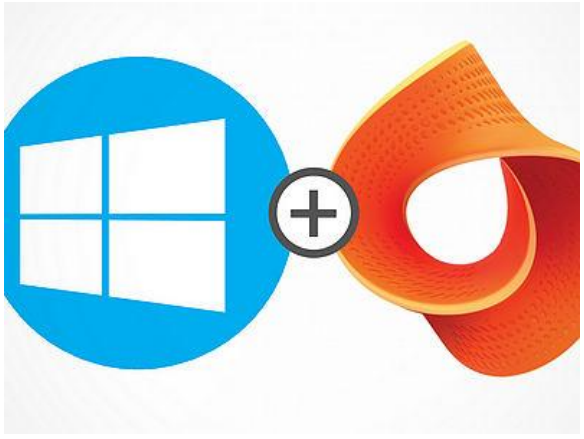
《Spark 官方文档》Spark调优

2016-04-22 10:23

Spark调优

由于大部分Spark计算都是在内存中完成的，所以Spark程序的瓶颈可能由集群中任意一种资源导致，如：CPU、网络带宽、或者内存等。最常见的情况是，数据能装进内存，而瓶颈是网络带宽;当然，有时候我们也需要做一些优化调整来减少内存占用，例如将RDD以序列化格式保存(storing RDDs in serialized form)。本文将主要涵盖两个主题：1.数据序列化(这对于优化网络性能极为重要);2.减少内存占用以及内存调优。同时，我们也会提及其他几个比较小的主题。



数据序列化

序列化在任何一种分布式应用性能优化时都扮演几位重要的角色。如果序列化格式序列化过程缓慢，或者需要占用字节很多，都会大大拖慢整体的计算效率。通常，序列化都是Spark应用优化时首先需要关注的地方。Spark着眼于要达到便利性(允许你在计算过程中使用任何Java类型)和性能的一个平衡。Spark主要提供了两个序列化库：

Java serialization : 默认情况，Spark使用Java自带的ObjectOutputStream 框架来序列化对象，这样任何实现了 java.io.Serializable 接口的对象，都能被序列化。同时，你还可以通过扩展 java.io.Externalizable 来控制序列化性能。Java序列化很灵活但性能较差，同时序列化后占用的字节数也较多。

Kryo serialization : Spark还可以使用Kryo 库(版本2)提供更高效的序列化格式。Kryo的序列化速度和字节占用都比Java序列化好很多(通常是10倍左右)，但Kryo不支持所有实现了 Serializable 接口的类型，它需要你在程序中 register 需要序列化的类型，以得到最佳性能。

要切换到使用 Kryo，你可以在 SparkConf 初始化的时候调用 conf.set(“spark.serializer”, “org.apache.spark.serializer.KryoSerializer”)。这个设置不仅控制各个worker节点之间的混洗数据序列化格式，同时还控制RDD存到磁盘上的序列化格式。目前，Kryo不是默认的序列化格式，因为它需要你在使用前注册需要序列化的类型，不过我们还是建议在对网络敏感的应用场景下使用Kryo。

Spark对一些常用的Scala核心类型(包括在 Twitter chill 库的AllScalaRegistrar中)自动使用Kryo序列化格式。

如果你的自定义类型需要使用Kryo序列化，可以用 registerKryoClasses 方法先注册：

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

```
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))

val sc = new SparkContext(conf)
```

Kryo的文档(Kryo documentation)中有详细描述了更多的高级选项，如：自定义序列化代码等。

如果你的对象很大，你可能需要增大 spark.kryoserializer.buffer 配置项(config)。其值至少需要大于最大对象的序列化长度。

最后，如果你不注册需要序列化的自定义类型，Kryo也能工作，不过每一个对象实例的序列化结果都会包含一份完整的类名，这有点浪费空间。

内存调优

内存占用调优主要需要考虑3点：1.数据占用的总内存(你多半会希望整个数据集都能装进内存吧);2.访问数据集中每个对象的开销;3.垃圾回收的开销(如果你的数据集中对象周转速度很快的话)。

一般，Java对象的访问时很快的，但同时Java对象会比原始数据(仅包含各个字段值)占用的空间多2~5倍。主要原因有：

每个Java对象都有一个对象头(object header)，对象头大约占用16字节，其中包含像其对应class的指针这样的信息。对于一些包含较少数据的对象(比如只包含一个Int字段)，这个对象头可能比对象数据本身还大。

Java字符串(String)有大约40子节点额外开销(Java String以Char数据的形式保存原始数据，所以需要一些额外的字段，如数组长度等)，并且每个字符都以两字节的UTF-16编码在内部保存。因此，10个字符的String很容易就占了60字节。

一些常见的集合类，如 HashMap、LinkedList，使用的是链表类数据结构，因此它们对每项数据都有一个包装器。这些包装器对象不仅其自身就有“对象头”，同时还有指向下一个包装器对象的链表指针(通常为8字节)。

原始类型的集合通常也是以“装箱”的形式包装成对象(如：java.lang.Integer)。

本节只是Spark内存管理的一个概要，下面我们会更详细地讨论各种Spark内存调优的具体策略。特别地，我们会讨论如何评估数据的内存使用量，以及如何改进 – 要么改变你的数据结构，要么以某种序列化格式存储数据。最后，我们还会讨论如何调整Spark的缓存大小，以及如何调优Java的垃圾回收器。

内存管理概览

Spark中内存主要用于两类目的：执行计算和数据存储。执行计算的内存主要用于混洗(Shuffle)、关联(join)、排序(sort)以及聚合(aggregation)，而数据存储的内存主要用于缓存和集群内部数据传播。Spark中执行计算和数据存储都是共享同一个内存区域(M)。如果执行计算没有占用内存，那么数据存储可以申请占用所有可用的内存，反之亦然。执行计算可能会抢占数据存储使用的内存，并将存储于内存的数据逐出内存，直到数据存储占用的内存比例降低到一个指定的比例(R)。换句话说，R是M基础上的一个子区域，这个区域的内存数据永远不会被逐出内存。然而，数据存储不会抢占执行计算的内存(否则实现太复杂了)。

这样设计主要有这么几个需要考虑的点。首先，不需要缓存数据的应用可以把整个空间用来执行计算，从而避免频繁地把数据吐到磁盘上。其次，需要缓存数据的应用能够有一个数据存储比例(R)的最低保证，也避免这部分缓存数据被全部逐出内存。最后，这个实现方式能够在默认情况下，为大多数使用场景提供合理的性能，而不需要专家级用户来设置内存使用如何划分。

虽然有两个内存划分相关的配置参数，但一般来说，用户不需要设置，因为默认值已经能够适用于绝大部分的使用场景：

spark.memory.fraction 表示上面M的大小，其值为相对于JVM堆内存的比例(默认0.75)。剩余的25%是为其他用户数据结构、Spark内部元数据以及避免OOM错误的安全预留空间(大量稀

疏数据和异常大的数据记录)。

spark.memory.storageFraction 表示上面R的大小，其值为相对于M的一个比例(默认0.5)。R是M中专门用于缓存数据块，且这部分数据块永远不会因执行计算任务而逐出内存。

评估内存消耗

确定一个数据集占用内存总量最好的办法就是，创建一个RDD，并缓存到内存中，然后再到web UI上"Storage"页面查看。页面上会展示这个RDD总共占用了多少内存。

要评估一个特定对象的内存占用量，可以用 SizeEstimator.estimate 方法。这个方法对试验哪种数据结构能够裁剪内存占用量比较有用，同时，也可以帮助用户了解广播变量在每个执行器堆上占用的内存量。

数据结构调优

减少内存消耗的首要方法就是避免过多的Java封装(减少对象头和额外辅助字段)，比如基于指针的数据结构和包装对象等。以下有几条建议：

设计数据结构的时候，优先使用对象数组和原生类型，减少对复杂集合类型(如：HashMap)的使用。 fastutil 提供了一些很方便的原声类型集合，同时兼容Java标准库。

尽可能避免嵌套大量的小对象和指针。

对应键值应尽量使用数值型或枚举型，而不是字符串型。

如果内存小于32GB，可以设置JVM标志参数 -XX:+UseCompressdOops 将指针设为4字节而不是8字节。你可以在 spark-env.sh 中设置这个参数。

序列化RDD存储

如果经过上面的调整后，存储的数据对象还是太大，那么你可以试试将这些对象以序列化格式存储，所需要做的只是通过 RDD persistence API 设置好存储级别，如：MEMORY_ONLY_SER。Spark会将RDD的每个分区以一个巨大的字节数组形式存储起来。以序列化格式存储的唯一缺点就是访问数据会变慢一点，因为Spark需要反序列化每个被访问的对象。如果你需要序列化缓存数据，我们强烈建议你使用Kryo(using Kryo)，和Java序列化相比，Kryo能大大减少序列化对象占用的空间(当然也比原始Java对象小很多)。

垃圾回收调优

JVM的垃圾回收在某些情况下可能会造成瓶颈，比如，你的RDD存储经常需要"换入换出"(新RDD抢占了老RDD内存，不过如果你的程序没有这种情况的话那JVM垃圾回收一般不是问题，比如，你的RDD只是载入一次，后续只是在这一个RDD上做操作)。当Java需要把老对象逐出内存的时候，JVM需要跟踪所有的Java对象，并找出那些对象已经没有用了。概括起来就是，垃圾回收的开销和对象个数成正比，所以减少对象的个数(比如用 Int数组取代LinkedList)，就能大大减少垃圾回收的开销。当然，一个更好的方法就如前面所说的，以序列化形式存储数据，这时每个RDD分区都只包含有一个对象了(一个巨大的字节数组)。在尝试其他技术方案前，首先可以试试用序列化RDD的方式(serialized caching)评估一下GC是不是一个瓶颈。

如果你的作业中各个任务需要的工作内存和节点上存储的RDD缓存占用的内存产生冲突，那么GC很可能会出现问题。下面我们将讨论一下如何控制好RDD缓存使用的内存空间，以减少这种冲突。

衡量GC的影响

GC调优的第一步是统计一下，垃圾回收启动的频率以及GC所使用的总时间。给JVM设置一下这几个参数(参考Spark配置指南 – configuration guide ，查看Spark作业中的Java选项参数)：-verbose:gc -XX:+PrintGCDetails，就可以在后续Spark作业的worker日志中看到每次GC花费的时间。注意，这些日志是在集群worker节点上(在各节点的工作目录下stdout文件中)，而不是你的驱动器所在节点。

高级GC调优

为了进一步调优GC，我们就需要对JVM内存管理有一个基本的了解：

Java堆内存可分配的空间有两个区域：新生代(Young generation)和老生代(Old generation)。新生代用以保存生存周期短的对象，而老生代则是保存生存周期长的对象。

新生代区域被进一步划分为三个子区域：Eden，Survivor1，Survivor2。

简要描述一下垃圾回收的过程：如果Eden区满了，则启动一轮minor GC回收Eden中的对象，生存下来(没有被回收掉)的Eden中的对象和Survivor1区中的对象一并复制到Survivor2中。两个Survivor区域是互相切换使用的(就是说，下次从Eden和Survivor2中复制到Survivor1中)。如果某个对象的年龄(每次GC所有生存下来的对象长一岁)超过某个阈值，或者Survivor2(下次是Survivor1)区域满了，则将对象移到老生代(Old区)。最终如果老生代也满了，就会启动full GC。

Spark GC调优的目标就是确保老生代(Old generation)只保存长生命周期RDD，而同时新生代(Young generation)的空间又能足够保存短生命周期的对象。这样就能在任务执行期间，避免启动full GC。以下是GC调优的主要步骤：

从GC的统计日志中观察GC是否启动太多。如果某个任务结束前，多次启动了full GC，则意味着用以执行该任务的内存不够。

如果GC统计信息中显示，老生代内存空间已经接近存满，可以通过降低 spark.memory.storageFraction 来减少RDD缓存占用的内存;减少缓存对象总比任务执行缓慢要强!

如果major GC比较少，但minor GC很多的话，可以多分配一些Eden内存。你可以把Eden的大小设为高于各个任务执行所需的工作内存。如果要把Eden大小设为E，则可以这样设置新生代区域大小：-Xmn=4/3*E。(放大4/3倍，主要是为了给Survivor区域保留空间)

举例来说，如果你的任务会从HDFS上读取数据，那么单个任务的内存需求可以用其所读取的HDFS数据块的大小来评估。需要特别注意的是，解压后的HDFS块是解压前的2~3倍大。所以如果我们希望保留3~4个任务并行的工作内存，并且HDFS块大小为64MB，那么可以评估Eden的大小应该设为 4*3*64MB。

最后，再观察一下垃圾回收的启动频率和总耗时有没有什么变化。

我们的很多经验表明，GC调优的效果和你的程序代码以及可用的总内存相关。网上还有不少调优的选项说明(many more tuning options)，但总体来说，就是控制好full GC的启动频率，就能有效减少垃圾回收开销。

其他注意事项

并行度

一般来说集群并不会满负荷运转，除非你吧每个操作的并行度都设得足够大。Spark会自动根据对应的输入文件大小来设置“map”类算子的并行度(当然你可以通过一个SparkContext.textFile等函数的可选参数来控制并行度)，而对于想 groupByKey 或 reduceByKey这类 “reduce” 算子，会使用其各父RDD分区数的最大值。你可以将并行度作为构建RDD第二个参数(参考 spark.PairRDDFunctions)，或者设置 spark.default.parallelism 这个默认值。一般来说，评估并行度的时候，我们建议2~3个任务共享一个CPU。

Reduce任务的内存占用

如果RDD比内存要大，有时候你可能收到一个OutOfMemoryError，但其实这是因为你的任务集中的某个任务太大了，如reduce任务groupByKey。Spark的混洗(Shuffle)算子(sortByKey，groupByKey，reduceByKey，join等)会在每个任务中构建一个哈希表，以便在任务中对数据分组，这个哈希表有时会很大。最简单的修复办法就是增大并行度，以减小单个任务的输入集。Spark对于200ms以内的短任务支持非常好，因为Spark可以跨任务复用执行器JVM，任务的启动开销很小，因此把并行度增加到比集群中总CPU核数还多是没有任何问题的。

广播大变量

使用SparkContext中的广播变量相关功能(broadcast functionality)能大大减少每个任务本身序列化的大小，以及集群中启动作业的开销。如果你的Spark任务正在使用驱动器(driver)程序中定义的巨大对象(比如：静态查询表)，请考虑使用广播变量替代之。Spark会在master上将各个任务的序列化后大小打印出来，所以你可以检查一下各个任务是否过大;通常来说，大于20KB的任务就值得优化一下。

数据本地性

数据本地性对Spark作业往往会有较大的影响。如果代码和其所操作的数据在统一节点上，那么计算速度肯定会更快一些。但如果二者不在一起，那必然需要挪动其中之一。一般来说，挪动序列化好的代码肯定比挪动一大堆数据要快。Spark就是基于这个一般性原则来构建数据本地性的调度。

数据本地性是指代码和其所处理的数据的距离。基于数据当前的位置，数据本地性可以划分成以下几个层次(按从近到远排序)：

PROCESS_LOCAL 数据和运行的代码处于同一个JVM进程内。

NODE_LOCAL 数据和代码处于同一节点。例如，数据处于HDFS上某个节点，而对应的执行器(executor)也在同一个机器节点上。这会比PROCESS_LOCAL稍微慢一些，因为数据需要跨进程传递。

NO_PREF 数据在任何地方处理都一样，没有本地性偏好。

RACK_LOCAL 数据和代码处于同一个机架上的不同机器。这时，数据和代码处于不同机器上，需要通过网络传递，但还是在同一个机架上，一般也就通过一个交换机传输即可。

ANY 数据在网络中其他未知，即数据和代码不在同一个机架上。

Spark倾向于让所有任务都具有最佳的数据本地性，但这并非总是可行的。某些情况下，可能会出现一些空闲的执行器(executor)没有待处理的数据，那么Spark可能就会牺牲一些数据本地性。有两种可能的选项：a)等待已经有任务的CPU，待其释放后立即在同一台机器上启动一个任务;b)立即在其他节点上启动新任务，并把所需要的数据复制过去。

而通常，Spark会等待一小会，看看是否有CPU会被释放出来。一旦等待超时，则立即在其他

可以单独配置，也

ality 相关参数。如

个参数，不过默认值

总结

本文是一个简短的Spark调优指南，列举了Spark应用调优一些比较重要的考虑点 – 最重要的就是，数据序列化和内存调优。对于绝大多数应用来说，用Kryo格式序列化数据能够解决大多数的性能问题。如果您有其他关于性能调优最佳实践的问题，欢迎邮件咨询(Spark mailing list)。

干货-Http请求get、post工具类

在网上找了好久都没有找到post、get请求的工具类，现在整理了一下分享出来。

java学习笔记 · 15评论 · 04-22 05:26

大数据：大数据之快速搭建hadoop2.6集群指南

本文详细介绍了在RedHat6.2以上版本的Linux服务器之上快速搭建hadoop2.6版本的集群方法。以下操作步骤是笔者在安装hadoop集群的安装笔记，如有对hadoop感兴趣的博友可按照本文操作进行无障碍搭建。可以确认以下所有操作步骤的准…

数据分析与数据可视化 · 2评论 · 04-22 10:15

Airbnb的知识管理

作者： Chetan Sharma and Jan Overgoor，译者：李端@成都敏捷社区问题Airbnb数据团队的一个责任是提升使用数据来做决策的能力。我们让数据访问民主化，允许所有员工基于数据做决…

董老师在硅谷 · 0评论 · 04-22 07:25

文章评论 0条

写下你的评论...　发表