Hadoop 平台下的内存溢出诊断方法①

姬 源, 覃 海, 周思明, 谢 东, 沈冠全

(贵州电网公司 电力调度控制中心, 贵阳 550002)

摘 要: 针对Hadoop平台的内存溢出问题, 结合框架结构和运行机理, 分析了常见的内存溢出原因, 提出一种诊断方法, 通过分析 Hadoop 运行时数据, 自动定位错误所在运行阶段, 并结合内存对象分配情况和系统配置参数, 诊断导致内存溢出的原因. 利用电网数据处理的典型应用场景进行实验, 实验结果验证了方法的有效性.

关键词: Hadoop; 内存溢出; 诊断

Method for Overflow Diagnosis of Memory in Hadoop Platform

JI Yuan, QIN Hai, ZHOU Si-Ming, XIE Dong, SHEN Guan-Quan

(Guizhou Electric Power Grid Dispatching and Control Center, Guiyang 550002, China)

Abstract: In this paper, it analyzes the mechanism of Hadoop and summarizes the common issue of memory leak, and proposes a method to diagnose this issue. The proposed approach could diagnose the phase of the overflow of memory occurs, the objects which consume most of the memory space, and the related configurations, to help the Hadoop user to find the root cause of error during out of memory. It also evaluates the effectiveness of the proposed approach under typical data processing applications for the power grid.

Key words: Hadoop; overflow of memory; diagnosis

当前, 工业信息系统所产生的数据量日益庞大, 据英国每日电讯报(Telegraph)报告[1,2], 英国对 4400 万户电表进行升级, 由原先每年两次采样周期, 调整为每天两次, 一年采集的数据量将从 8800 万条记录, 激增到 320 亿条记录. 太平洋煤气和电力公司(Pacific Gas and Electric), 增加了 1.2PB 的存储, 用于支持其管理的 70 万块智能电表. 为了能够处理分析这些海量数据, 需要使用 Hadoop 等大数据处理平台.

内存溢出错误是大数据处理平台的常见错误, 例如, 国际知名的程序开发者问答网站 stackoverflow 上关于“Hadoop out of memory”的问题超过 10000 个[3], 在 Spark 邮件列表上有 10%的问题是关于“out of memory” [4]. 内存溢出错误会导致处理数据的任务失败, 甚至会引发平台崩溃等严重后果. 目前对于内存溢出大部分的处理方法是重新执行任务, 然而, 对于由系统配置、数据流、用户代码等原因而导致的内存溢出错误, 即使用户重新执行任务依然无法避免.

Hadoop 平台的内存溢出错误难以调试, 在运行过程中具有一定的随机性, 并且由于框架的结构复杂, 数据处理量大, 每一步数据操作都会申请和释放内存, 因此传统的内存溢出诊断方法及工具已不适用. 用户通常采用查看脚本程序和运行日志等方式, 凭借经验猜测导致内存溢出的原因, 需要花费大量时间和精力.

针对上述问题, 本文结合 Hadoop 框架结构和运行机理, 综合分析造成内存溢出的常见原因, 总结常见的内存溢出规律和自动化诊断需要解决的问题. 进一步, 本文提出一种内存溢出诊断方法, 通过分析 Hadoop 运行时数据, 自动定位错误所在运行阶段, 并结合内存对象分配情况和系统配置参数, 诊断导致内存溢出的原因. 最后, 利用电网数据处理的典型应用场景进行实验, 实验结果验证了方法的有效性.

1 Hadoop内存溢出原因分析

1.1 Hadoop 平台的系统层次

① 收稿时间:2015-11-27;收到修改稿时间:2016-01-25 [doi:10.15888/j.cnki.csa.005285]

© 中国科学院软件研究所 http://www.c-s-a.org.cn

如图 1 所示, Hadoop 平台分为 3 个层次: 用户层、系统框架层以及物理运行层.

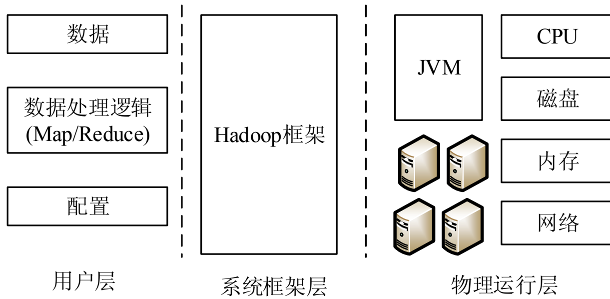

图 1　Hadoop 平台的系统层次

用户层主要包括所需处理的数据集、用户定义的数据处理逻辑(如 MapReduce 函数)、配置信息. 系统框架层将数据处理的任务划分成更小的子任务, 并且在多台机器上并行地执行这些子任务. 系统框架层将输入、输出以及中间数据以对象的形式申请内存, 并且分配缓冲区. 物理运行层是实际运行任务的进程, Hadoop 物理运行层是以 Java 进程形式在 Java 虚拟机(JVM)中运行. 内存的实际申请是由物理运行层完成.

1.2 影响因素

Hadoop 任务申请的内存既包含系统框架申请的内存, 也包含用户层数据处理逻辑申请的内存. 如公式 1 所示, 当用于数据处理和数据存储的内存对象大于内存限制时, 就会发生内存溢出. 公式 1 中的内存对象不但指已经被成功分配的内存对象, 也包括未申请成功的内存对象. 当 Java 进程向 JVM 虚拟机申请内存时, 如果未成功申请的内存对象和已经分配的内存对象的总和超出了内存溢出的限制, 则申请失败并且 JVM 虚拟机会抛出 OutofMemory Exception 异常, 导致内存溢出错误.

$$size(userCodeObject) + size(frameworkObject) > memoryLimit \quad (1)$$

由公式 1 可以看出内存溢出主要由框架对象、用户代码对象、内存限制等三方面因素共同导致.

1) Hadoop 框架对象内存

在 Hadoop 任务执行时, 框架自身为了完成数据传输和数据处理的工作会申请大量的内存, 这些内存的大小则受用户配置、数据流等因素的影响. 由于框架的逻辑是固定的, 通常可以根据数据流和用户配置计算出申请内存的大小.

2) 用户对象内存

Hadoop 为用户提供了 map()、reduce()等编程接口, 用户可以在函数内部编写任意逻辑的代码, 因此要找出造成用户代码申请大量内存的原因并不容易. Hadoop 提供给用户的函数一般是以流式处理的方式处理数据, 每个函数框架会向用户代码的函数输入一系列的键值对, 用户处理完后向框架输出一系列键值对. 因此, 可以通过分析每个对象和数据流之间的关系进行预估, 就有可能找出造成用户代码申请大量内存的原因.

3) 内存限制

Hadoop 内存的限制主要由集群每个节点的 JVM 虚拟机决定. JVM 在抛出 OutofMemory Exception 时, 可能是由于申请的内存容量大于 JVM 的堆空间导致, 也可能是由于 JVM 垃圾回收机制中不同对象管理区域的限制被违背而导致. 因此, 需要分析 JVM 存在哪些内存限制, 并且有哪些可能的情况会导致内存限制被违背.

2 Hadoop内存溢出诊断

2.1 方法组成

本文提出的 Hadoop 内存溢出诊断方法如图 2 所示, 包括 7 个方面的组成要素.

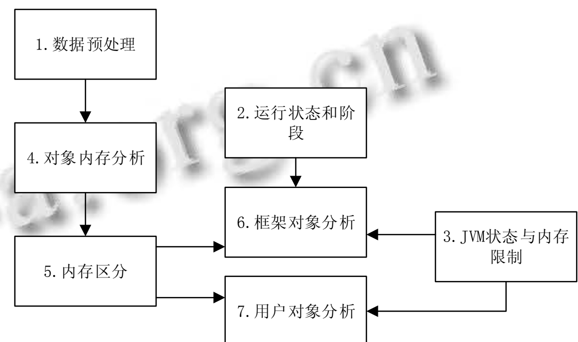

图 2　方法原理图

1) 内存溢出的表征

每种造成内存溢出的原因都有其对应的表征, 且表征可能分布于不同的层次. 针对每一类内存溢出错误所伴随的现象, 收集各个层次的运行信息, 例如, 用户的配置信息、运行时的数据流信息、对象数量、集群节点的系统资源状态等, 这些信息将作为内存溢出诊断方法的输入参数. 本文中, 内存溢出表征所需的具体数据包括:

© 中国科学院软件研究所 http://www.c-s-a.org.cn

① Hadoop 任务信息: 包括日期、优先级、map 完成百分比、reduce 完成百分比、map 成功数、reduce 成功数、taskId、task error 等;

② Hadoop 缓冲区信息: 包括 kvbuffer、kvindices、kvoffsets 等缓冲区大小;

③ Hadoop 任务日志, 包括 stdout、stdlog、system log;

④ JVM 信息: 包括内存大小、old 区和 new 区大小、每个区占用大小等.

⑤ 内存 Dump 文件: 包括每个对象占用的内存大小, 以及内存之间的相互引用关系等.

上述数据被封装为不同的数据对象类, 并建立了相应的引用关系(图 3 所示), 进而实现内存溢出信息的结构化表征.

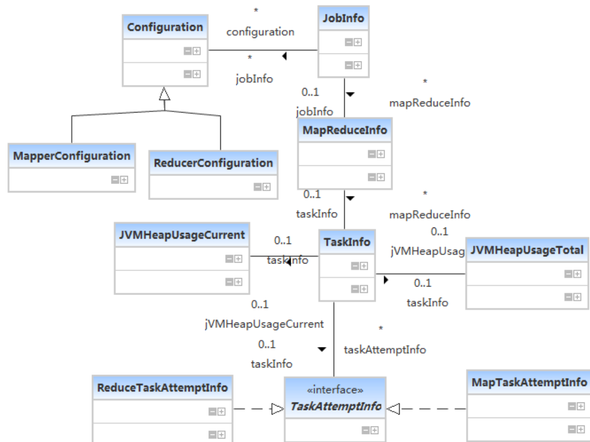

图 3 内存溢出信息的结构化表征

2) 运行状态与阶段

一个完整的 Hadoop 任务被拆分成更细粒度的子任务, 分发到集群上执行. 内存溢出的形式是子任务发生了内存溢出导致任务失败. 因此为了找到内存溢出的原因, 首先需要找到发生内存溢出而失败的子任务, 并对该任务做更为详细的分析. 此外, Hadoop 在不同的运行阶段会申请不同的内存, 并且会调用不同的用户代码. 一旦确定了运行阶段, 只需要对该阶段的原因进行分析, 就可以排除其他阶段的原因. 例如, 在 reduce 阶段发生内存溢出, 则无需检查 map 阶段框架申请的内存对象和 map()函数中的用户代码.

3) JVM 状态与内存限制

JVM 将整个堆的空间分为 new(young) 和 old(tenured)两个区. new 区用来存放新创建的 Java 对象, 当 new 区存满时 JVM 会进行对象收集, 将 new 区中还未被清除的对象移到 old 区. 本文方法定义了一系列造成内存溢出的条件. 如公式(2)(3)(4)所示, 一旦内存的使用情况违背了其中任意一个条件, 就会造成内存溢出错误. 其中: 公式(2)表示已经分配的内存和正在申请的对象内存之和大于 JVM 虚拟机堆的空间, 这种情况表示没有可用的堆空间; 公式(3)表示 new 区、old 区均无法容纳正要申请的对象内存; 公式(4)表示一些数据结构(如 ArrayList、HashMap)在进行自身容量扩展时(通常会放大至 1.5 到 2 倍)导致违背内存限制.

$$size(allocatedObjects) + size(\text{allocating}) > size(heap) \quad (2)$$

$$size(allocatedObjects) + size(\text{allocating}) > boundary(new\ \&\ old) \quad (3)$$

$$size(allocatedObjects) + size(expantion) + size(\text{allocating}) > size(heap) \quad (4)$$

4) 对象内存分析

为了计算内存溢出时用户内存和框架内存的大小, 需要对 Java 内存对象的“镜像”文件(Heap Dump)进行分析. 内存中对象存在着复杂的引用关系, 对象的内存既包括自身的数据占用的内存, 还包括引用数据所占用的内存. 本文将 JVM 的 Heap Dump 文件转化为 Dominator Tree[5], 从而将对象的引用关系表示出来, 进而得到占用内存最大的内存对象. 此外, 可以通过遍历 Dominator Tree, 找到在非引用情况下的最大内存对象.

5) 内存区分

Hadoop 框架内存溢出和用户代码内存溢出是由不同的原因造成的两种不同类型的内存溢出错误, 需要区分框架申请的对象和用户代码申请的对象. 本文通过两者的线程号和函数名进行区分. 对于框架内存对象, 一旦确定了 Hadoop 发生内存溢出时所处的阶段, 就可以分析出该阶段 Hadoop 框架都申请了哪些内存对象; 对于用户对象, 由于框架只提供了 map()、reduce()、partition()等编程接口, 因此可以根据对象的函数堆栈判断其是否是通过上述函数申请的用户对象.

6) 框架对象分析

由于 Hadoop 框架的逻辑是固定的, 可以通过建立数据流和 Hadoop 框架申请内存大小之间的关系预测出处理全部数据所需的 Hadoop 框架内存大小. Hadoop

© 中国科学院软件研究所 http://www.c-s-a.org.cn

每个阶段都存在数据传输, 每一个处理过程中输入和输出数据都称为数据流

$$size(intermediateData) = f(previousDataFlow, configs) \quad (5)$$

$$size(inputRecordsOf \quad fun()) = f(intermediateData, configs) \quad (6)$$

公式(5)用于计算中间数据占用的内存, 是由前一步输出数据的大小和配置共同决定的. 例如, SpillBuffer的大小是由map()的输出数据的大小和配置的buffer大小所决定. 这样, 就可以通过收集上一步的输出数据信息和配置, 计算出当前中间数据的大小. 公式(6)表示对于某一个处理过程 fun()的输入数据的大小, 主要由前一步的数据流大小和用户配置决定, 用于计算某一处理阶段的输入数据流大小. 例如, 在shuffle阶段combine()函数的输入数据的大小是由上一步存储在 MergeBound 中的数据大小所决定因此. 通过上述计算, 可以分析得到框架的内存占用情况.

7) 用户对象分析

用户申请的内存大小与输入数据流的关系通常包括以下5种: 1)与输入数据无关, 该种情况如公式(7)所示. 输入数据为常量, 但该常量可能很大. 通常来说用户读入外部数据的情况较多, 也可能是用户申请了一段固定大小的buffer用于计算. 2)与输入数据成正比, 该种情况如公式(8)所示, 分配的内存大小会随着输入数据的流入而增加. 该种情况可能是用户缓存了数据, 也有可能是用户用于计算申请了内存空间. 3)与输入数据成二次函数的关系, 如公式(9)所示. 在这种关系下, 用户申请的空间会随着输入流的输入而显著增加, 对于该种情况, 内存对于输入流较为敏感, 一旦输入流过大就会导致内存溢出. 4)用户代码申请的内存大小和输入流成 O(nlogn)的关系, 如公式(10)所示. 5)用户代码的内存大小和输入流成O(logn)的关系, 如公式(11)所示.

$$size(userCodeObject) = C \quad (7)$$

$$size(userCodeObject) = an + C \quad (8)$$

$$size(userCodeObject) = an^2 + bn + C \quad (9)$$

$$size(userCodeObject) = an * \log n + bn + C \quad (10)$$

$$size(userCodeObject) = a \log n + C \quad (11)$$

为了能够准确地检查出用户代码申请内存大小与输入流之间的关系, 需要分析它们之间的关系是属于以上哪种情况. 本文首先查看它们在输入数据 5%和95%的状态, 作为其始末的状态, 这是由于用户代码有可能在初始输入时和发生内存溢出时的状态不稳定, 因此取5%和95%的位置作为其始末状态. 接下来使用公式(12)的算法计算. 如果计算结果为 0, 则说明分配的内存并没有随着数据输入的增加而增加, 因此属于第1种情况; 如果结果接近1, 则说明输入为第2种情况; 如果结果接近于2, 那么属于第3种情况.

$$relationship_{\exp} = \log_{\Delta input}(\Delta(size(userObject))) \quad (12)$$

在确定了输入数据流和用户代码申请空间之间的函数关系后, 可以用拟合的方法确定参数, 最后将用户代码的输入流大小带入方程求出当所有数据输入用户代码时所使用的内存.

2.2 内存溢出诊断

首先, 根据日志信息找出当前失败的任务, 并确定当前的运行阶段. 如图 3 所示, 针对发生内存溢出的任务进程, 通过框架对象分析、用户对象分析, 计算用户代码内存大小、框架内存大小, 并判断违背了何种内存溢出的限制. 例如, 当前 Hadoop 框架占用400M 内存, 用户代码占用 100M 内存, 使得 JVM 的500M 堆空间被占满, 从而导致了内存溢出. 此时, 方法可以解释为何发生内存溢出, 但还无法定位出原因. 基于上述状态信息, 可以得到用户代码申请内存和Hadoop 框架内存所占比例, 以及已分配内存占 JVM堆空间的比例. 如果是用户代码对象占用了大部分内存, 那么很可能是由用户代码申请的内存造成的内存溢出, 反之亦然. 如果两者相差不明显, 则可推测是Hadoop框架内存和用户代码内存共同作用导致的. 然后, 结合内存限制条件, 分析已分配的内存占用 JVM堆空间的比例, 当已分配的内存占用了绝大部分 JVM堆空间时, 则已分配内存造成内存溢出的可能性更大.

进一步, 在完成内存计算和预估后, 可以得到Hadoop 内存溢出时框架对象和用户代码对象的分布情况. 对于用户对象, 由于开发人员熟悉自己编写的代码, 通过对象名称、对象所在的代码行号, 以及用户代码和输入流大小的关系和输入流大小等信息, 可以进行内存溢出问题的修复. 对于框架对象, 如果对Hadoop框架内部机理不熟悉, 将仍然无法判断造成该对象占用内存大的原因, 因此需要进一步分析影响该对象占用内存的配置和数据流大小. 例如: 框架对象mergeQueue 占用内存过多, 可以利用先验规则, 给出影响 mergeQueue 的参数: shuffledSegments 和数据流mergeQueueData, 并且给出它们的关系 min (shuffled

© 中国科学院软件研究所 http://www.c-s-a.org.cn

Segments,mergeQueueData). 基于上述信息, 开发人员可以通过修改 shuffledSegments 配置来修复内存溢出问题.

3 实验与分析

为了验证本文方法的有效性, 搭建了10个节点的Hadoop 集群, 每个节点的内存为 8G、磁盘 500G. 我们结合电网信息系统中典型的流式数据处理与分析场景, 设计了 3 个简单的测试基准应用, 包括线损分析、拓扑分析、电费结算等, 利用这些基准应用模拟在Hadoop 任务执行的阶段出现内存溢出. 篇幅原因, 这里仅给出 reduce 阶段的实验结果及分析. 表 1 给出了reduce 阶段的实验参数配置. 每个基准应用随机地配置表中参数, 各产生10组任务运行配置, 共提交30个任务. 运行结果显示, 其中 22 个任务发生了内存溢出错误.

表 1 实验参数配置

参数	作用	范围
mapred.job.reduce.input.buffer.percent	调整 reduce buffer 大小	0-1
JVM heap size	JVM 虚拟机堆大小	1G – 5G
mapred.reduce.tasks	reduce 个数	1-10
Cache percent	reduce()函数中数据缓存概率	0%~100%

表 2 给出本文方法的诊断结果, 并与人工分析结果进行对比. 在出现内存泄露的 22 个任务中, 方法成功诊断出18个内存溢出的原因, 4个未成功. 在诊断出错的 4 个问题中, 其中 3 个任务的诊断结果虽然错误, 但仍能够为进一步的人工诊断提供帮助. 这是因为方法发现 reduce()函数中对象时, 不但给出了对象和数据流的函数关系, 还给出了 reduce()函数输入数据流, 开发人员可以进一步判断是由于reduce个数过少导致.

表 2 实验结果

本文方法诊断结果	人工分析结果	个数	是否正确
reduce buffer 配置错误	配置 reducebuffer	10	正确
ArrayList 对象过大	用户缓存了过多的数据导致内存溢出	4	正确
reduce buffer 配置错误	堆空间太小	1	错误
reduce buffer 配置错误并且 ArrayList 对象过大	reduce buffer 和 ArrayList 共同导致内存溢出	3	正确
ArrayList 对象过大	reduce 个数太少, 导致某个 reduce 处理太大的数据	3	错误

4 相关工作

由于大数据框架需要处理的数据量庞大, 而系统的内存空间有限, 在系统运行时容易造成内存溢出错误. 目前存在一些针对大数据处理框架的优化工作, 例如 Google 的 FlumeJava[6]将多个 MapReduce 以Pipeline 连接, 减少了多个 MapReduce 任务之间的读写开销; MapReduce Online[7]的工作对MapReduce进行改进, 使得Hadoop的中间数据得到优化; Mesos[8]能够有效地对计算框架资源进行分配、回收以及调度. 虽然上述工作能够减少内存溢出的可能, 但仍无法避免内存溢出问题, 并且也难以定位错误原因.

5 结语

本文对 Java 平台环境下的内存溢出问题进行研究, 提出一种针对 Hadoop 平台内存溢出问题的诊断方法. 方法通过分析Hadoop运行时数据, 自动定位错误所在运行阶段, 并结合内存对象分配情况和系统配置参数, 诊断导致内存溢出的原因. 方法对大数据处理和分析场景下的内存溢出错误的自动化诊断提供了技术支撑.

参考文献

1 Danaly J. The Coming Smart Grid Data Surge. http://www.smartgridnews.com/artman/publish/News_Blogs_News/The-Coming-Smart-Grid-Data-Surge-1247.html.

2 Pariseau B. Senior News Writer. Energy IT sees smart-grid boon for data storage. http://searchstoragechannel.techtarget.com/news/1355355/Energy-IT-sees-smart-grid-boon-for-data-storage.

3 Stack Overflow. http://stackoverflow.com. [2015].

4 Apache Spark. https://spark.apache.org. [2015].

5 Dominator Tree. https://en.wikipedia.org/wiki/Dominator_(graph_theory). [2015].

6 Chambers C, et al. FlumeJava: easy, efficient data-parallel pipelines. ACM Sigplan Notices, 2010, 45(6): 363–375.

7 Condie T, et al. MapReduce online. Proc. of the 7th USENIX Conference on Networked Systems Design and Implementation USENIX Association. 2010. 21–21.

8 Hindman B, et al. Mesos: A platform for fine-grained resource sharing in the data center. Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation USENIX Association. 2011. 429–483.

© 中国科学院软件研究所 http://www.c-s-a.org.cn