孙振超---实践是最好的成长，发表是最好的记忆
天下无难事，天下无易事。难，是因为问题没有被细分到可以去解决的程度；易，是因为找到了解决问题的办法。

昵称：孙振超
园龄：7年
粉丝：98
关注：6
+加关注

< 2018年9月 >

日	一	二	三	四	五	六
26	27	28	29	30	31	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	1	2	3	4	5	6

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

我的标签

Java(10)
大促(10)
Mybatis(4)
源代码分析(4)

一次young gc耗时过长优化过程

1 问题源起

上游系统通过公司rpc框架调用我们系统接口超时（默认超时时间为100ms）数量从50次/分突然上涨到2000次/分，在发生变化时间段里我们的系统也没有做过代码变更，但上游系统的调用确发生了变化。由于处于主要链路上，sre同学找过来询问原因，所以开始了问题排查。

2 问题初步定位

排查rpc超时的基本思路是这样的：

1) 服务端处理确实超时

2) 服务端或者客户端由于某种原因卡住

a) 磁盘清理

b) tr线程池

c) gc

d) 网络

因为在发生问题时上游调用其他系统的服务并发生类似的情况，所以可以认为是服务端出现了问题而不是客户端。

2.1 服务端处理确实超时

别名(2)
httpservice(2)
web(2)
jvm(2)
源代码(2)
osgi(2)
更多

随笔分类(84)

Java基础(15)
Mybatis/iBatis(7)
Nosql(1)
OOA&OOD(2)
osgi(1)
PHP(1)
SSH(5)
读书笔记(7)
服务器(4)
工具使用问题汇集(5)
工作总结(18)
数据库(4)
网络知识(6)
项目管理(6)
消息中间件(1)
职业相关(1)

随笔档案(73)

2018年2月 (1)
2018年1月 (1)
2017年11月 (10)
2017年8月 (1)
2017年7月 (1)
2017年4月 (1)
2017年3月 (1)
2016年12月 (2)
2015年12月 (2)

rpc框架在有一个traceId，用于标识请求。同一个traceId在请求端的耗时是一百多毫秒，但是在接收端的耗时只有2-3ms，抽查了几天里的多个请求都是这样的情况请求端的耗时远大于接收端的耗时。这样就排除了超时是由服务端处理引起的。

2.2 磁盘清理

线上系统通常会打印一些日志，用于记录系统的运行情况，方便问题排查和进行监控，当这些日志的数量累计到一定量时，会进行磁盘清理。在磁盘清理期间，会对IO有较大的影响。通过查看磁盘清理日志，发现磁盘清理时间和调用超时时间并不匹配；另外磁盘清理是偶发性的，但是上游系统调用超时是持续性的，因而磁盘清理导致超时也可以排除。

2.3 rpc线程池

rpc框架采用的是NIO方式进行客户端和服务端进行通讯，在服务端会有一个线程池处理到达的请求，查看了对应的线程池日志，发现线程池的队列中出现堆积的次数一天只有几次而已，但请求端调用服务出现的超时问题会连续几个小时出现的。因而超时问题由线程池的配置引起也可以排除。

2.4 网络

我们的服务是多地多机房部署模式，从调用方最远的机房到我们这边机房网络耗时大概为25ms，来回就是50ms，加上服务端的处理时间，客户端的请求总时间应该在60ms左右，但是现在的总耗时在100ms以上，说明网络也不是该问题的主要原因。

2.5 gc

对上游系统调用我们系统超时的机器进行汇总，发现超时的请求主要集中在内存为4g的机器上，而10g的机器出现超时的情况很少，因而对4g和10g机器的gc进行了分析，发现二者有很大的不同：

	新生代大小	**Young gc**清理内存量	**Young gc**耗时	**Young gc** 时间间隔
4g机器	780m	707840k	0.8s	10s
10g机器	1800m	1382400k	0.05s	17s

从上面的表格可以看到，10g机器的新生代清理效率是4g机器的新生代清理效率20多倍，4g机器新生代的清理耗时过长。初步判断，请求方访问超时应该和4g机器的young gc执行时间过长可能有关。

3 具体原因定位

3.1 young gc过程

2015年7月 (1)
2014年12月 (1)
2014年11月 (1)
2014年10月 (2)
2014年9月 (1)
2014年8月 (2)
2014年6月 (1)
2014年5月 (1)
2014年3月 (1)
2014年1月 (1)
2013年12月 (2)
2013年11月 (1)
2013年10月 (1)
2013年9月 (1)
2013年6月 (3)
2013年5月 (3)
2013年4月 (8)
2013年2月 (1)
2013年1月 (6)
2012年12月 (9)
2012年11月 (6)

文章分类

测试

最新评论

1. Re:java 之DelayQueue实际运用示例
第一个场景的代码Studnet不是根据getdelay()来进行排序的，实现了compareable接口利用workTimke来进行排序，submitTime这个没啥用，getdelay（）直接返回零.....
--林冲—first
2. Re:一次young gc耗时过长优化过程

jvm中的young gc过程大致分为如下几个步骤：

1) 存活对象标注

2) 存活对象从Edge区拷贝到Survivor 1，重置指针

3) 清理Edge区和Survivor 2

4g和10g机器都是4核，二者的cpu的主频和一级二级缓存是相同的，从理论上讲二者的清理效率应该是相等的，但现在4g机器清理的内存的性能却比10g清理却相差如此多，说明问题可能发生在存活对象标注上。同时young gc过程中的存活对象标注是要STW，这个阶段jvm会对外停止响应，很有可能是因为这个STW导致响应超时。

3.2 GC root

Jvm中的young gc是从GC roots开始的，GC root作为tracing GC的“根集合”，主要包含：

1) Class - 由系统类加载器(system class loader)加载的对象，这些类是不能够被回收的，他们可以以静态字段的方式保存持有其它对象。我们需要注意的一点就是，通过用户自定义的类加载器加载的类，除非相应的java.lang.Class实例以其它的某种（或多种）方式成为roots，否则它们并不是roots，.

2) Thread - 活着的线程

3) Stack Local - Java方法的local变量或参数

4) JNI Local - JNI方法的local变量或参数

5) JNI Global - 全局JNI引用

6) Monitor Used - 用于同步的监控对象

7) Held by JVM - 用于JVM特殊目的由GC保留的对象，但实际上这个与JVM的实现是有关的。可能已知的一些类型是：系统类加载器、一些JVM知道的重要的异常类、一些用于处理异常的预分配对象以及一些自定义的类加载器等。然而，JVM并没有为这些对象提供其它的信息，因此就只有留给分析分员去确定哪些是属于"JVM持有"的了。

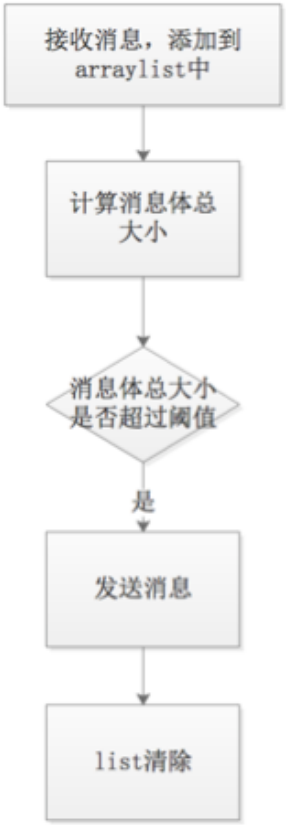
4g机器和10g机器运行的代码都是相同，因而能够导致gc root不同个地方应该是第二和第六项。

3.3 Thread分析

对4g机器的线程进行了dump，在zprofile中进行了下分析，分析结果显示存在线程阻塞的情况，而后定位到所有的线程阻塞都是在一个发送消息的方法上，而这个方法是synchronized的，由于synchronized导致了调用线程获取不到锁的时候发生了阻塞。

3.4 发送消息为什么会采用同步

这个方法的作用是发送消息，但是为了减少消息发送的次数，在内部进行了消息的合并，当消息内容达到一定量时才发送（现在设定的是3k），整体流程如下



这其实是一个简单的流程，但由于会存在多个线程对存储消息的list和存储消息体大小的size进行写操作，特别是对list的操作时，会存在有些线程在向list中添加数据，而有些线程间接触发了arraylist的clear操作，这样就会报ConcurrentModificationException

。同时为避免在计算消息体大小时多线程操作引起的计算值比实际值小的问题就给这个方法用了**synchronized** 关键字加锁让其顺序操作。

4 修改

主要的修改有3个

1) 把用arraylist存储消息体采用ConcurrentHashMap，map中的key是一个uudi，value是消息体本身。从arraylist改为ConcurrentHashMap，可以大大增加并发性能。

2) 采用AtomicInteger来保存当前map中消息内容的大小。

3) 当map中的消息体总大小到达阈值后，先将当前map中的所有的key取出，然后将map中的数据封装到一条消息中发送出去，将当前map中消息体的大小置为0，而后for循环删除已成功发送的key。

4. Mybatis之reflection包源代码解析（一）(3)
5. Jenkins+Maven+SVN快速搭建持续集成环境(转)(3)

推荐排行榜

1. 《一万小时理论》读书笔记(11)
2. Jenkins+Maven+SVN快速搭建持续集成环境(转)(9)
3. java 之DelayQueue实际运用示例(6)
4. 高性能网站建设指南---前端工程师技能精髓(6)
5. 计算机编码基础知识及Java中编码转换(3)

5 效果对比

5.1 young gc耗时

机器	young gc耗时（单位秒）		yong gc间隔（单位秒）		总提升率 （young gc耗时优化前/young gc耗时优化后） *（young gc间隔优化后/young gc间隔优化前）
	优化前	优化后	优化前	优化后	
4g内存	0.8	0.06	10	12	16
10g内存	0.05	0.05	17	20	1.176470588

5.2 线程阻塞

优化后线程阻塞情况不存在。

5.3 调用报错情况

报错从之前的近2000/分回到了50/分。

6 后记

问题虽然得到了解决，但是线程阻塞和young gc耗时的确切关系还是没有不太清除，需要后续继续了解，也看哪位大神给解释下。

孙振超
关注 - 6
粉丝 - 98
+加关注

« 上一篇：java 非阻塞算法实现基础：unsafe类介绍
» 下一篇：面试总结

posted on 2017-04-14 22:42 孙振超 阅读(7760) 评论(3) 编辑 收藏

评论:

#1楼 2017-04-15 03:40 | 叶知
Stop the world有一个前置条件：到达safetyPoint。也就是在GC前可能有一个等待时段。
所以，猜测可能和这个有关：当大量线程争用同一个锁时，等所有线程执行到safetyPoint，可能需要一段时间。
当然，这个可能并不大；除非在同步代码附近，没有放置safetyPoint。

支持(0) 反对(0)

#2楼[楼主] 2017-04-16 11:47 | 孙振超
@ 叶知
对于 gc大多都知道其过程，但是对于内部的细节、涉及到的底层数据结构、调度执行策略就不太清楚了，出现问题后以猜测尝试为主，碰巧问题解决了也不知道具体的原因，导致无法复用

支持(0) 反对(0)

#3楼 2018-06-06 11:12 | xiantianzju
@ 孙振超，这个问题后面有深入跟进么？Yong gc过程是STW，那么按道理来说只跟当时参与的Yong gc的线程数有关系。你的4g和10g的机器cpu是一样的，应该来说不会有任何差异的。

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· 有效感知距离突破1000米，图森未来是如何做到的?
· 马云的退，阿里的进
· 关于.NET Core是否应该支持WCF Hosting的争论
· 不到4年估值60亿美元 商汤科技值吗?
· 年轻的奥秘在于复杂性：人体日渐衰老生理过程变简单
» 更多新闻...

最新知识库文章:
· 为什么说 Java 程序员必须掌握 Spring Boot ?
· 在学习中，有一个比掌握知识更重要的能力
· 如何招到一个靠谱的程序员
· 一个故事看懂"区块链"
· 被踢出去的用户
» 更多知识库文章...