

lqf的专栏

目录视图

摘要视图

RSS 订阅

个人资料



lqf1403

访问：30342次

积分：755

等级：BLOG > 3

排名：千里之外

[2017直通软考，拿证无忧](#) [程序员简历优化指南！](#) [程序员1月书讯](#) [云端应用征文大赛，秀绝招，赢无人机！](#)

Java GC、新生代、老年代

标签：[java](#) [jvm](#) [内存](#)

2015-11-15 13:16

1876人阅读

[评论\(3\)](#)[收藏](#)[举报](#)分类：[java \(12\)](#)

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?)

[-]

1. [堆内存](#)
2. [GC堆](#)
3. [GC日志](#)
4. [JVM参数选项](#)
5. [实例分析](#)

1、堆内存

原创： 33篇 转载： 28篇
译文： 0篇 评论： 8条

文章搜索

文章分类

排序算法 (5)
java (13)
个人 (2)
python (1)
machine learning (2)
linux (4)
scala (2)
Hadoop (18)
Hbase (8)
MapReduce (12)
maven (2)
mahout (1)
Java混合编程 (1)
java设计模式 (1)
Hadoop源码 (1)
JVM (0)
数据结构 (3)

文章存档

2016年09月 (3)
2016年08月 (1)
2016年07月 (3)

Java 中的堆是 JVM 所管理的最大的一块内存空间，主要用于存放各种类的实例对象。

在 Java 中，堆被划分成两个不同的区域：新生代 (Young)、老年代 (Old)。新生代 (Young) 又被划分为三个区域：Eden、From Survivor、To Survivor。

这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。

堆的内存模型大致为：



从图中可以看出：堆大小 = 新生代 + 老年代。其中，堆的大小可以通过参数 `-Xms`、`-Xmx` 来指定。

(本人使用的是 JDK1.6，以下涉及的 JVM 默认值均以该版本为准。)

默认的，新生代 (Young) 与老年代 (Old) 的比例的值为 1:2 (该值可以通过参数 `-XX:NewRatio` 来指定)，即：新生代 (Young) = 1/3 的堆空间大小。老年代 (Old) = 2/3 的堆空间大小。其中，新生代 (Young) 被细分为 Eden 和 两 域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分。

默认的，Eden : from : to = 8 : 1 : 1 (可以通过参数 `-XX:SurvivorRatio` 来设定)，即：Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。

JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块 Survivor 区域是空闲着的。因此，新生代实际可用的内存空间为 9/10 (即90%)的新生代空间。

2、GC堆

2016年06月 (1)

2016年05月 (1)

展开

阅读排行

使用mapreduce向hbase: (2538)

Java GC、新生代、老年 (1874)

Hbase-1.1.2完全分布式 (1842)

使用IntelliJ IDEA编写Sp (1808)

Hadoop作业提交与停止 (1444)

Hbase写入速度优化 (1069)

hadoop集群后停命令 (1022)

解决关闭Hadoop时no na (934)

mapreduce中map方法一 (855)

浅谈AdaBoost算法--附有 (826)

评论排行

Java GC、新生代、老年 (3)

Hbase-1.1.2完全分布式 (2)

使用mapreduce向hbase: (1)

Eclipse中用Link方式安装 (1)

算法--排序算法的介绍与, (1)

Python:使用matplotlib绘 (0)

二叉树的四种遍历 (递归 (0)

算法--排序算法的介绍与, (0)

算法--排序算法的介绍与, (0)

java向上转型与向下转型 (0)

Java 中的堆也是 GC 收集垃圾的主要区域。GC 分为两种：Minor GC、Full GC (或称为 Major GC)。

Minor GC 是发生在新生代中的垃圾收集动作，所采用的是复制算法。

新生代几乎是所有 Java 对象出生的地方，即 Java 对象申请的内存以及存放都是在这个地方。Java 中的大部分对象通常不需长久存活，具有朝生夕灭的性质。当一个对象被判定为“死亡”的时候，GC 就有责任来回收掉这部分对象的内存空间。新生代是 GC 收集垃圾的频繁区域。当对象在 Eden (包括一个 Survivor 区域，这里假设是 from 区域) 出生后，在经过一次 Minor GC 后，如果对象还存活，并且能够被另外一块 Survivor 区域所容纳(上面已经假设为 from 区域，这里应为 to 区域，即 to 区域有足够的内存空间来存储 Eden 和 from 区域中存活的对象)，则使用复制算法将这些仍然还存活的对象复制到另外一块 Survivor 区域 (即 to 区域) 中，然后清理所使用过的 Eden 以及 Survivor 区域 (即 from 区域)，并且将这些对象的年龄设置为1，以后对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时 (默认是 15 岁，可以通过参数 -XX:MaxTenuringThreshold 来设定)，这些对象就会成为老年代。但这也不是一定的，对于一些较大的对象 (即需要分配一块较大的连续内存空间) 则是直接进入老年代。Full GC 是发生在老年代的垃圾收集动作，所采用的是标记-清除算法。现实的生活中，老年代的人通常会比新生代的人“早死”。堆内存中的老年代(Old)不同于个，老年代里面的对象几乎个个都是在 Survivor 区域中熬过来的，它们是不会那么就容易“死掉”了的。因此，Full GC 发生的次数不会有 Minor GC 那么频繁，并且做一次 Full GC 要比进行一次 Minor GC 的时间更长。另外，标记-清除算法收集垃圾的时候会产生许多的内存碎片 (即不连续的内存空间)，此后需要为较大的对象分配内存空间时，若无法找到足够的连续的内存空间，就会提前触发一次 GC 的收集动作。

3、GC日志

设置 JVM 参数为 -XX:+PrintGCDetails，使得控制台能够显示 GC 相关的日志信息，执行上面代码，下面是其结果。

新生代 MinorGC 后, 新生代内存使用 MinorGC 后, JVM堆内存使用

[GC [PSYoungGen: 316K -> 32K(18432K)] 475K -> 191K(60544K), 0.0002062 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

MinorGC 前, 新生代总的 MinorGC 前, 堆的可用 MinorGC 总耗时

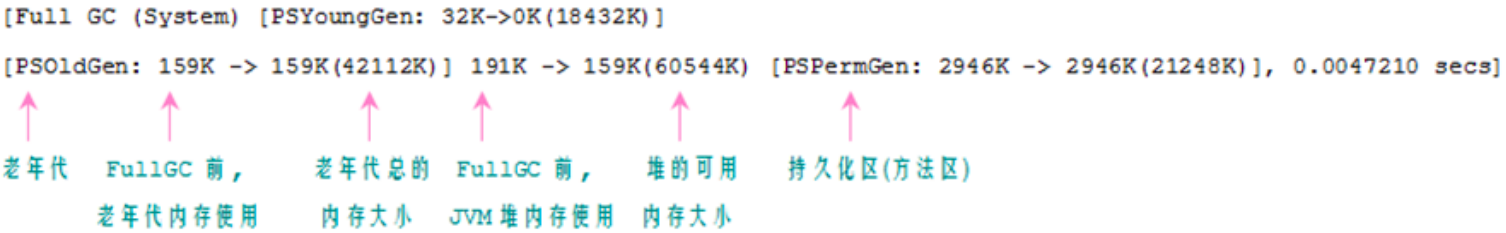
新生代内存使用 内存大小 JVM堆内存使用 内存大小

推荐文章

- * 造轮子 | 如何设计一个面向协议的 iOS 网络请求库
- * Android新特性介绍, ConstraintLayout完全解析
- * Android 热修复 Tinker接入及源码浅析
- * 创业公司做数据分析 (六) 数据仓库的建设
- * 【死磕Java并发】-----深入分析synchronized的实现原理

最新评论

- Java GC、新生代、老年代 咸鱼学编程: 学习了
- Eclipse中用Link方式安装Maven! hwffish: 下载地址呢
- Java GC、新生代、老年代 benbendeda: mark
- 使用mapreduce向hbase1.1.2插, ShawshankLin: 请问你的37个regions是怎么建立的?有什么特点吗?
- Java GC、新生代、老年代 _Bruce_Wong: Mark,
- Hbase-1.1.2完全分布式安装教程 铭毅天下: 启动 http://****:16030/rs-status 访问后: The RegionServer...
- Hbase-1.1.2完全分布式安装教程 power0405hf: 大兄弟写的不错
- 算法--排序算法的介绍与总结 (三 该昵称已经被占用: 写的好详细啊! 不错, 32个赞

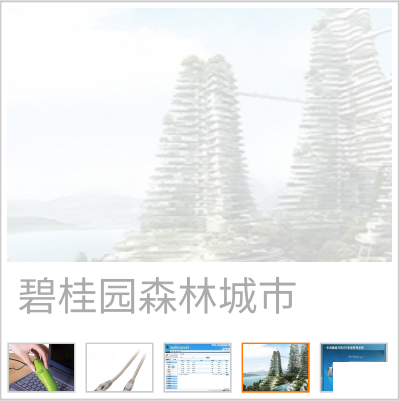


Full GC 信息与 Minor GC 的信息是相似的, 这里就不一个一个的画出来了。从 Full GC 信息可知, 新生代可用的内存大小约为 18M, 则新生代实际分配得到的内存空间约为 20M(为什么是 20M? 请继续看下面...)。老年代分得的内存大小约为 42M, 堆的可用内存的大小约为 60M。可以计算出: 18432K (新生代可用空间) + 42112K (老年代空间) = 60544K (堆的可用空间) 新生代约占堆大小的 1/3, 老年代约占堆大小的 2/3。也可以看出, GC 对新生代的回收比较乐观, 而对老年代以及方法区的回收并不明显或者说不及新生代。并且在这里 Full GC 耗时是 Minor GC 的 22.89 倍。

4、JVM参数选项

jvm 可配置的参数选项可以参考 **Oracle** 官方网站给出的相关信息: <http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html>
下面只列举其中的几个常用和容易掌握的配置选项

配置参数	功能
-Xms	初始堆大小。如: -Xms256m
-Xmx	最大堆大小。如: -Xmx512m
-Xmn	新生代大小。通常为 Xmx 的 1/3 或 1/4。新生代 = Eden + 2 个 Survivor 空间。实际可用空间为 = Eden + 1 个 Survivor, 即 90%
-Xss	JDK1.5+ 每个线程堆栈大小为 1M, 一般来说如果栈不是很深的话, 1M 是绝对够用了的。
-XX:NewRatio	新生代与老年代的比例, 如 -XX:NewRatio=2, 则新生代占整个堆空间的1/3, 老年代占2/3



碧桂园森林城市

配置参数	功能
-XX:SurvivorRatio	新生代中 Eden 与 Survivor 的比值。默认值为 8。即 Eden 占新生代空间的 8/10，另外两个 Survivor 各占 1/10
-XX:PermSize	永久代(方法区)的初始大小
-XX:MaxPermSize	永久代(方法区)的最大值
-XX:+PrintGCDetails	打印 GC 信息
-XX:+HeapDumpOnOutOfMemoryError	让虚拟机在发生内存溢出时 Dump 出当前的内存堆转储快照，以便分析用

注意：**PermSize**永久代的概念在**jdk1.8**中已经不存在了，取而代之的是**metaspace**元空间，当认为执行永久代的初始大小以及最大值是**jvm**会给出如此下提示：

Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=30m; support was removed in 8.0

Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=30m; support was removed in 8.0

5、实例分析

```
1  /**
2   -Xms60m
3   -Xmx60m
4   -Xmn20m
5   -XX:NewRatio=2 ( 若 Xms = Xmx，并且设定了 Xmn，那么该项配置就不需要配置了 )
6   -XX:SurvivorRatio=8
7   -XX:PermSize=30m
8   -XX:MaxPermSize=30m
9   -XX:+PrintGCDetails
10  */
11  public class TestVm{
12      public void doTest(){
13          Integer M = new Integer(1024 * 1024 * 1); //单位，兆(M)
14          byte[] bytes = new byte[1 * M]; //申请 1M 大小的内存空间
```

```

15         bytes = null; //断开引用链
16         System.gc(); //通知 GC 收集垃圾
17         System.out.println(); //
18         bytes = new byte[1 * M]; //重新申请 1M 大小的内存空间
19         bytes = new byte[1 * M]; //再次申请 1M 大小的内存空间
20         System.gc();
21         System.out.println();
22     }
23     public static void main(String[] args) {
24         new TestVm().doTest();
25     }
26 }

```

按上面代码中注释的信息设定 jvm 相关的参数项，并执行程序，下面是一次执行完成控制台打印的结果：

```

1  [ GC [ PSYoungGen: 1351K -> 288K (18432K) ] 1351K -> 288K (59392K), 0.0012389 se
2  [ Full GC (System) [ PSYoungGen: 288K -> 0K (18432K) ] [ PSOldGen: 0K -> 160K
3
4  [ GC [ PSYoungGen: 2703K -> 1056K (18432K) ] 2863K -> 1216K(59392K), 0.0008206
5  [ Full GC (System) [ PSYoungGen: 1056K -> 0K (18432K) ] [ PSOldGen:
6
7
8  Heap
9  PSYoungGen
10     total 18432K, used 327K [0x00000000fec00000, 0x00000000100000000, 0x0000000010
11     eden space 16384K, 2% used [0x00000000fec00000,0x00000000fec51f58,0x00000000
12     from space 2048K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x0000000010
13     to   space 2048K, 0% used [0x00000000ffc00000,0x00000000ffc00000,0x00000000f
14
15  PSOldGen
16     total 40960K, used 1184K [0x00000000fc400000, 0x00000000fec00000, 0x00000000fe
17
18  PSPermGen
19

```

```

20      total 30720K, used 2959K [0x00000000fa600000, 0x00000000fc400000, 0x00000000f
      object space 30720K, 9% used [0x00000000fa600000,0x00000000fa8e3ce0,0x00000000

```

从打印结果可以看出，堆中新生代的内存空间为 18432K (约 18M)，eden 的内存空间为 16384K (约 16M)，from / to survivor 的内存空间为 2048K (约 2M)。

这里所配置的 Xmn 为 20M，也就是指定了新生代的内存空间为 20M，可是从打印的堆信息来看，新生代怎么就只有 18M 呢？另外的 2M 哪里去了？别急，是这样的。新生代 = eden + from + to = 16 + 2 + 2 = 20M，可见新生代的内存空间确实是按 Xmn 参数分配得到的。而且这里指定了 SurvivorRatio = 8，因此，eden = 8/10 的新生代空间 = 8/10 * 20 = 16M。from = to = 1/10 的新生代空间 = 1/10 * 20 = 2M。堆信息中新生代的 total 18432K 是这样来的：eden + 1 个 survivor = 16384K + 2048K = 18432K，即约为 18M。因为 jvm 每次只是用新生代中的 eden 和一个 survivor，因此新生代实际的可用内存空间大小为所指定的 90%。因此可以知道，这里新生代的内存空间指的是新生代可用的总的内存空间，而不是指整个新生代的空间大小。另外，可以看出老年代的内存空间为 40960K (约 40M)，堆大小 = 新生代 + 老年代。因此在这里，老年代 = 堆大小 - 新生代 = 60 - 20 = 40M。

最后，这里还指定了 PermSize = 30m，PermGen 即永久代 (方法区)，它还有一个名字，叫非堆，主要用来存储由 jvm 加载的类文件信息、常量、静态变量等。

回到 doTest() 方法中，可以看到代码在第 14、18、19 这三行中分别申请了一块 1M 大小的内存空间，并在 16 行和 18 行中分别显式的调用了 System.gc()。从控制台打印的信息来看，每次调 System.gc()，是先进入 Minor GC，然后 GC。

第 16 行触发的 **Minor GC** 收集分析：

从信息 PSYoungGen : 1351K -> 288K，可以知道，在第 14 行为 bytes 分配的内存空间已经被回收完成。引起 GC 回收这 1M 内存空间的因素是第 15 行的 bytes = null; bytes 为 null 表明之前申请的那 1M 大小的内存空间现在已经没有任何引用变量在使用它了，并且在内存中它处于一种不可到达状态 (即没有任何引用链与 GC Roots 相连)。那么，当 Minor GC 发生的时候，GC 就会来回收掉这部分的内存空间。

第 16 行触发的 **Full GC** 收集分析：

在 Minor GC 的时候，信息显示 PSYoungGen : 1351K -> 288K，再看看 Full GC 中显示的 PSYoungGen : 288K -> 0K，可以看出，Full GC 后，新生代的内存使用变成 0K 了，那么这 288K 到底哪去了？难道都被 GC 当成垃圾回收掉了？当然不是了。

我还特意在 main 方法中 new 了一个 Test 类的实例，这里的 Test 类的实例属于小对象，它应该被分配到新生代内存当中，现在还在调用这个实例的 doTest 方法呢，GC 不可能在这个时候来回收它的。

接着往下看 Full GC 的信息，会发现一个很有趣的现象，PSOldGen: 0K -> 160K，可以看到，Full GC 后，老年代的内存使用从 0K 变成了 160K，想必你已经猜到大概是怎么回事了。当 Full GC 进行的时候，默认的方式是尽量清空新生代 (YoungGen)，因此在调 System.gc() 时，新生代 (YoungGen) 中存活的对象会提前进入老年代。

第 20 行触发的 **Minor GC** 收集分析：

从信息 PSYoungGen : 2703K -> 1056K，可以知道，在第 18 行创建的，大小为 1M 的数组被 GC 回收了。在第 19 行创建的，大小也为 1M 的数组由于 bytes 引用变量还在引用它，因此，它暂时未被 GC 回收。

第 20 行触发的 **Full GC** 收集分析：

在 Minor GC 的时候，信息显示 PSYoungGen : 2703K -> 1056K，Full GC 中显示的 PSYoungGen : 1056K -> 0K，以及 PSOldGen: 160K -> 1184K，可以知道，新生代 (YoungGen) 中存活的对象又提前进入老年代了。

顶
3

踩
0

上一篇 Hbase1.1.2采用javaAPI插入批量数据

下一篇 Exception in thread "main" java.lang.IllegalArgumentException: Wrong FS: hdfs://localhost:9000/user/

我的同类文章

java (12)

- | | | | | | |
|--------------------------|------------|--------|-------------------------|------------|--------|
| • 谈谈数据库的ACID | 2016-07-20 | 阅读 124 | • 谈谈数据库的ACID | 2016-07-20 | 阅读 120 |
| • JAVA泛型 | 2016-06-27 | 阅读 102 | • Java 调用Matlab | 2016-05-31 | 阅读 282 |
| • Java8中Lambda表达式的10个... | 2016-03-06 | 阅读 218 | • SimpleDateFormat使用详解 | 2015-05-11 | 阅读 158 |
| • java内存分配和String类型的深... | 2015-05-08 | 阅读 151 | • java中"=="和"equals"的比较 | 2015-05-07 | 阅读 176 |
| • java的数据类型——基本类型... | 2015-05-06 | 阅读 289 | • Eclipse Java注释模板设置详解 | 2015-04-02 | 阅读 221 |

[更多文章](#)

参考知识库



Java 知识库

22629 关注 | 1440 收录



Oracle知识库

3999 关注 | 252 收录



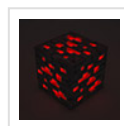
Java SE知识库

21792 关注 | 468 收录



Java EE知识库

14552 关注 | 1215 收录



算法与数据结构知识库

12994 关注 | 2320 收录

猜你在找

[Java经典算法讲解](#)[java 虚拟机--新生代与老年代GC](#)[Redis内存管理和优化](#)[java 虚拟机--新生代与老年代GC](#)[网络工程师内存存储容量计算强化训练教程](#)[java 虚拟机--新生代与老年代GC](#)[Oracle初级管理](#)[java 虚拟机--新生代与老年代GC](#)[spring3.2入门到大神（备java基础、jsp、servlet，javaee精髓）](#)[java 虚拟机--新生代与老年代GC](#)

广告

穹顶穿越-Chrome浏览器番羽墙插件

¥ 9/月 秒开Google\Facebook\Youtube等站点

[查看评论](#)

3楼 [咸鱼学编程](#) 2016-11-21 16:17发表



学习了

2楼 [benbendeda](#) 2016-09-15 17:27发表



mark

1楼 [_Bruce_Wong](#) 2016-05-20 17:13发表



Mark,

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

[全部主题](#) [Hadoop](#) [AWS](#) [移动游戏](#) [Java](#) [Android](#) [iOS](#) [Swift](#) [智能硬件](#) [Docker](#) [OpenStack](#) [VPN](#)
[Spark](#) [ERP](#) [IE10](#) [Eclipse](#) [CRM](#) [JavaScript](#) [数据库](#) [Ubuntu](#) [NFC](#) [WAP](#) [jQuery](#) [BI](#) [HTML5](#)
[Spring](#) [Apache](#) [.NET](#) [API](#) [HTML](#) [SDK](#) [IIS](#) [Fedora](#) [XML](#) [LBS](#) [Unity](#) [Splashtop](#) [UML](#)
[components](#) [Windows Mobile](#) [Rails](#) [QEMU](#) [KDE](#) [Cassandra](#) [CloudStack](#) [FTC](#) [coremail](#) [OPhone](#)

[CouchBase](#) [云计算](#) [iOS6](#) [Rackspace](#) [Web App](#) [SpringSide](#) [Maemo](#) [Compuware](#) [大数据](#) [aptech](#)
[Perl](#) [Tornado](#) [Ruby](#) [Hibernate](#) [ThinkPHP](#) [HBase](#) [Pure](#) [Solr](#) [Angular](#) [Cloud Foundry](#) [Redis](#)
[Scala](#) [Django](#) [Bootstrap](#)

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#) | [合作伙伴](#) | [论坛反馈](#)

[网站客服](#) [杂志客服](#) [微博客服](#) [webmaster@csdn.net](#) 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 | 江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved



□