HZ BOOKS
华章IT

由资深Java工程师撰写，来自一线实践经验的结晶

详细分析G1的基本运行原理以及调优方法，讲解细腻，图示丰富，可帮助Java工程师深入理解垃圾回收技术

Java
核心技术
系列

JVM G1
源码分析和调优

JVM G1 Implementation and Performance Tuning

彭成寒 编著

机械工业出版社
China Machine Press

Java 核心技术系列

JVM G1 源码分析和调优

JVM G1 Implementation and Performance Tuning

彭成寒 编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

JVM G1 源码分析和调优 / 彭成寒编著 . —北京：机械工业出版社，2019.3
（Java 核心技术系列）

ISBN 978-7-111-62197-3

I. J… II. 彭… III. JAVA 语言 – 程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字（2019）第 043922 号

JVM G1 源码分析和调优

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：吴 怡 责任校对：李秋荣
印 刷：北京市荣盛彩色印刷有限公司 版 次：2019 年 4 月第 1 版第 1 次印刷
开 本：186mm×240mm 1/16 印 张：18.5
书 号：ISBN 978-7-111-62197-3 定 价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 读者信箱：hzit@hzbook.com

Preface 前　言

G1 是目前最成熟的垃圾回收器，已经广泛应用在众多公司的生产环境中。我们知道，CMS 作为使用最为广泛的垃圾回收器，也有令人头疼的问题，即如何对其众多的参数进行正确的设置。G1 的目标就是替代 CMS，所以在设计之初就希望降低程序员的负担，减少人工的介入。但这并不意味着我们完全不需要了解 G1 的原理和参数调优。笔者在实际工作中遇到过一些因参数设置不正确而导致 GC 停顿时间过长的问题。但要正确设置参数并不容易，这里涉及两个方面：第一，需要对 G1 的原理熟悉，只有熟悉 G1 的原理才知道调优的方向；第二，能分析和解读 G1 运行的日志信息，根据日志信息找到 G1 运行过程中的异常信息，并推断哪些参数可以解决这些异常。

本书尝试从 G1 的原理出发，系统地介绍新生代回收、混合回收、Full GC、并发标记、Refine 线程等内容；同时依托于 jdk8u 的源代码介绍 Hotspot 如何实现 G1，通过对源代码的分析来了解 G1 提供了哪些参数、这些参数的具体意义；最后本书还设计了一些示例代码，给出了 G1 在运行这些示例代码时的日志，通过日志分析来尝试调整参数并达到性能优化，还分析了参数调整可能带来的负面影响。

乍听起来，G1 非常复杂，应该会有很多的参数。实际上在 JDK8 的 G1 实现中，一共新增了 93 个参数，其中开发参数（develop）有 41 个，产品参数（product）有 31 个，诊断参数（diagnostic）有 9 个，实验参数（experimental）有 12 个。开发参数需要在调试版本中才能进行验证（本书只涉及个别参数），其余的三类参数都可以在发布版本中打开、验证和使用。本书除了几个用于验证的诊断参数外，覆盖了发布版本中涉

及的所有参数，为读者理解 G1 以及调优 G1 提供了帮助。

本书共分为 12 章，主要内容如下：

- 第 1 章介绍垃圾回收的发展及使用的算法，同时还介绍一些重要并常见的术语。该章的知识不仅仅限于本书介绍的 G1，对于研读 JVM 文章或者 JVM 源码都有帮助。
- 第 2 章介绍 G1 中的基本概念，包括分区、卡表、根集合、线程栈等和垃圾回收相关的基本知识点。
- 第 3 章介绍 G1 是如何分配对象的，包括 TLAB 和慢速分配，G1 的对象分配和其他垃圾回收器的对象分配非常类似，只不过在分配的时候以分区为基础，除此之外没有额外的变化，所以该章知识不仅仅适用于 G1 也适用于其他垃圾回收器，最后介绍了参数调优，同样也适用于其他的垃圾回收器。
- 第 4 章介绍 G1 Refine 线程，包括 G1 如何管理和处理代际引用，从而加快垃圾回收速度，介绍了 Refinement 调优涉及的参数；虽然 CMS 也有卡表处理代际引用，但是 G1 的处理和 CMS 并不相同，Refine 线程是 G1 新引入的部分。
- 第 5 章介绍新生代回收，包括 G1 如何进行新生代回收，包括对象标记、复制、分区释放等细节，还介绍了新生代调优涉及的参数。
- 第 6 章介绍混合回收。主要介绍 G1 的并发标记算法及其难点，以及 G1 中如何解决这个难点，同时介绍了并发标记的步骤：并发标记、Remark（再标记）和清理阶段；最后还介绍了并发标记的调优参数。
- 第 7 章介绍 Full GC。在 G1 中，Full GC 对整个堆进行垃圾回收，该章介绍 G1 的串行 Full GC 和 JDK 10 之后的并行 Full GC 算法。
- 第 8 章介绍垃圾回收过程中如何处理引用，该功能不是 G1 独有的，也适用于其他垃圾回收器。
- 第 9 章介绍 G1 的新特性：字符串去重。根据 OpenJDK 的官方文档，该特性可平均节约内存 13% 左右，所以这是一个非常有用的特性，值得大家尝试和使用。另外，该特性和 JDK 中 String 类的 intern 方法有一些类似的地方，所以该章还比较了它们之间的不同。
- 第 10 章介绍线程中的安全点。安全点在实际调优中涉及的并不多，所以很多人并不是特别熟悉。实际上，垃圾回收发生时，在进入安全点中做了不少的工作，

而这些工作基本上是串行进行的，这些事情很有可能导致垃圾回收的时间过长。该章除了介绍如何进入安全点之外，还介绍了在安全点中做的一些回收工作，以及当发现它们导致 GC 过长时该如何调优。

- 第 11 章介绍如何选择垃圾回收器，以及选择 G1 遇到问题需要调优时我们该如何下手。该章属于理论性的指导，在实际工作中需要根据本书提到的参数正面影响和负面影响综合考虑，并不断调整。
- 第 12 章介绍了下一代垃圾回收器 Shenandoah 和 ZGC。G1 作为发挥重要作用的垃圾回收器仍有不足之处，因此未来的垃圾回收器仍会继续发展，该章介绍了下一代垃圾回收器 Shenandoah 和 ZGC 对 G1 的改进之处及其工作原理。

本书的附录包含如下内容：

- 附录 A 介绍如何开始阅读和调试 JVM 代码。这里简单介绍了 G1 的代码架构和组织形式。另外简单介绍了 Linux 的调试工具 GDB，这个工具对于想要了解 JVM 细节的同学必不可少。
- 附录 B 介绍如何使用 NMT 对 JVM 内存进行跟踪和调试。这个知识对于想要深入理解 JVM 内存的管理非常有帮助，另外在实际工作中，特别是 JDK 升级中我们必须比较同一应用在不同 JVM 运行情况下的内存使用。
- 附录 C 介绍了 Java 程序员阅读 JVM 时需要知道的一些 C++ 知识。这里并未罗列 C++ 的语法以及语法特性，仅仅介绍一些 C++ 语言特有的、而 Java 语言没有的语法，或者 Java 语言中的使用或理解不同于 C++ 语言的部分语法。这个知识是为 Java 程序员准备的，特别是为在阅读 JVM 代码时准备的。

G1 在 JDK 6 中出现，经历 JDK 7 的发展，到 JDK 8 已经相当成熟，在 JDK 9 之后 G1 就作为 JVM 的默认垃圾回收器。JDK 8 作为 Oracle 公司长期支持的版本，本书主要基于 JDK 8 进行分析，所用的版本是 jdk8u60。在第 7 章中为了扩展读者的视野，追踪最新的技术，还介绍了 JDK 10 中的并行 Full GC。读者可以自行到 OpenJDK 的官网下载，也可以使用笔者在 GitHub 中的备份（JDK 8: https://github.com/chenghanpeng/jdk8u60，JDK 10：https://github.com/chenghanpeng/jdk10u）。

本书在分析源码的时候会给出源代码所属的文件，例如在介绍 G1 分区类型时，指出源代码位于 hotspot/src/share/vm/gc_implementation/g1/heapRegionType.hpp，这里的 hotspot 就是你下载的 jdk8u60 代码里面的一级目录。如果你不希望在本地保留源代码

可以直接浏览网址 https://github.com/chenghanpeng/jdk8u60，在此你可以找到这个一级目录 hotspot，然后通过逐个查看子目录 src、share、vm、gc_implementation、g1 就可以找到源文件 heapRegionType.hpp。

需要注意的是，在分析源码的时候为了节约篇幅，通常会对原始的代码进行一些调整，例如删除一些大括号、统计信息、打印信息，或者删除一些不影响理解原理和算法的代码，大家在和源码比较时需要注意这些变化。另外对于定义在 header 文件和 cpp 文件中的一些函数，为了使代码紧凑，通常会忽略头文件中的定义，直接按照 C++ 的语法，即类名 :: 成员函数的方式给出源码，这样的代码可能和原文件不完全一致，但是完全符合 C++ 语言的组织，阅读源码时要注意将定义和实现分开。

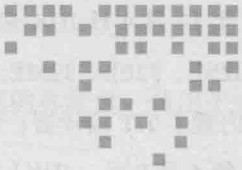
由于笔者水平有限，时间仓促，书中难免出现一些错误或者不准确的地方，恳请读者批评指正。可以通过 https://github.com/chenghanpeng/jdk8u60/issues 进行讨论，期待能够得到读者朋友们的真情反馈，在技术道路上互勉共进。

在本书的写作过程中，得到了很多朋友以及同事的帮助和支持，在此表示衷心的感谢！

感谢吴怡编辑的支持和鼓励，在写作过程中给出了非常多的意见和建议，不厌其烦地认真和笔者沟通，力争做到清晰、准确、无误。感谢你的耐心，为你的专业精神致敬！

感谢我的家人，特别是谢谢我的儿子，体谅爸爸牺牲了陪伴你的时间。有了你们的支持和帮助，我才有时间和精力去完成写作。

Contents 目　录



第 1 章 Chapter 1

垃圾回收概述

Java 的发展已经超过了 20 年，已是最流行的编程语言。为了更好地了解和使用 Java，越来越多的开发人员开始关注 Java 虚拟机（JVM）的实现技术，其中**垃圾回收**（也称垃圾收集）是最热门的技术点之一。目前 G1 作为 JVM 中最新、最成熟的垃圾回收器受到很多的人关注，本书从 G1 的原理出发，介绍新生代收集、混合收集、Full GC、并发标记、Refine、Evacuation 等内容。本章先回顾 Java 语言的发展历程，然后介绍 JVM 中一些常用的概念以便与读者统一术语，随后介绍垃圾回收的主要算法以及 JVM 中实现了哪些垃圾回收的算法。

1.1 Java 发展概述

Java 平台和语言最开始是 SUN 公司在 1990 年 12 月进行的一个内部研究项目，我们通常所说的 Java 一般泛指 JDK（Java Developer Kit），它既包含了 Java 语言和开发工具，也包含了执行 Java 的虚拟机（Java Virtual Machine，JVM）。从 1996 年 1 月 23 日开始，JDK 1.0 版本正式发布，到如今 Java 已经经历了 23 个春秋。以下是 Java 发展历程中值得纪念的几个时间点：

- 1998年12月4日JDK迎来了一个里程碑版本1.2。其技术体系被分为三个方向，J2SE、J2EE、J2ME。代表技术包括EJB、Java Plug-in、Swing；虚拟机第一次内置了JIT编译器；语言上引入了Collections集合类等。
- 2000年5月8日，JDK1.3发布。在该版本中Hotspot正式成为默认的虚拟机，Hotspot是1997年SUN公司收购LongView Technologies公司而获得的。
- 2002年2月13日，JDK1.4发布。该版本是Java走向成熟的一个版本。从此之后，每一个新的版本都会增加新的特性，比如JDK5改进了内存模型、支持泛型等；JDK6增强了锁同步等；JDK7正式支持G1垃圾回收、升级类加载的架构等；JDK8支持函数式编程等。
- 2006年11月13日的JavaOne大会上，SUN公司宣布最终会把Java开源，由OpenJDK组织对这些源码独立管理，从此之后Java程序员多了一个研究JVM的官方渠道。
- 2009年4月20日，Oracle公司宣布正式以74亿美元的价格收购SUN公司，Java商标从此正式归Oracle所有，自此Oracle对Java的管理和发布进入了一个新的时期。

随着时间的推移，JDK 9和JDK 10也已经正式发布，但是JDK 9和JDK 10并不是Oracle长期支持的版本（Long Term Support），这意味着JDK 9和JDK 10只是JDK 11的一个过渡版本，它们只用于整合新的特性，当下一个版本发布之后，这些过渡版本将不再更新维护。2018年9月25日JDK 11正式发布，随着新版本的发布，Oracle公司未来对JDK的支持也会变化。按照现在的声明，从2019年1月起对于商业用户，Oracle公司对JDK 8不再提供公共的更新，从2020年12月起对个人用户也不再提供公共的更新。

G1作为CMS的替代者，一直吸引着众多Java开发者的目光，自从JDK 7正式推出以来，G1不断地增强，并从JDK 8开始越来越成熟，在JDK 9、JDK 10、JDK 11中都成为默认的垃圾回收器。实际上也有越来越多的公司开始在生产环境中使用G1作为垃圾回收器，有一篇文章描述了JDK 9中GC的基准测试（benchmark），表明G1已经优于其他的GC[⊖]。可以预见随着JDK 11的推出，会有越来越多的公司和个人使用G1

⊖ http://blog.mgm-tp.com/2018/01/g1-mature-in-java9

作为生产环境中的垃圾回收器。

G1 的目标是在满足短时间停顿的同时达到一个高的吞吐量，适用于多核处理器、大内存容量的系统。其实现特点为：

- 短停顿时间且可控：G1 对内存进行分区，可以应用在大内存系统中；设计了基于部分内存回收的新生代收集和混合收集。
- 高吞吐量：优化 GC 工作，使其尽可能与 Mutator 并发工作。设计了新的并发标记线程，用于并发标记内存；设计了 Refine 线程并发处理分区之间的引用关系，加快垃圾回收的速度。

新生代收集指针对全部新生代分区进行垃圾回收；混合收集指不仅仅回收新生代分区，同时回收一部分老生代分区，这通常发生在并发标记之后；Full GC 指内存不足时需要对全部内存进行垃圾回收。

并发标记是 G1 新引入的部分，指的是在 Mutator 运行的同时标记哪些对象是垃圾，看到这里大家一定非常好奇 G1 到底是怎么实现的，举一个简单的例子。比如你的妈妈正在打扫房间，扫房房间需要识别哪些物品有用哪些无用，无用的物品就是垃圾。同时你正在房间活动，活动的同时你可能往房间增加了新的物品，也可能把房间的物品重新组合，也可能产生新的无用物品。最简单的垃圾回收器如串行回收器的做法就是在打扫房间标识物品的时候，你要暂停一切活动，这个时候你的妈妈就能完美地识别哪些物品有用哪些无用。但最大的问题就是需要你暂停一切活动直到房间里面的物品识别完毕，在实际系统中意味着这段时间应用程序不能提供服务。G1 的并发标记就是在打扫房间识别物品有用或者无用的同时，你还可以继续活动，怎么正确做标记呢？一个简单的办法就是在打扫房间识别垃圾物品开始的时候记录你增加了哪些物品，动过哪些物品。然后在物品标记结束的时候对这些变更过的物品重新标记一次，当然在这一次标记时需要你暂停一切活动，否则永远也没有尽头，这通常称为再标记（Remark）。这个就是所谓的增量并发标记，在 G1 中具体的算法是 Snapshot-At-The-Beginning（SATB），关于这个算法我们会在第 6 章详细介绍。Refine 线程也是 G1 新引入的，它的目的是为了在进行部分收集的时候加速识别活跃对象，具体介绍参见第

4章。

本书依托于jdk8u的源代码来介绍JVM如何实现G1，通过源代码的分析理解算法以及了解G1提供的参数的具体意义；最后还会给出一些例子，通过日志，分析该如何调整参数以达到性能优化。

这里提到的jdk8u是指OpenJDK的代码，OpenJDK是SUN公司（现Oracle）推出的JDK开源代码，因为标准的JDK（这里指Oracle版的JDK）会有一些内部功能的代码，那些代码在开源的时候并未公开。在2017年9月Oracle公司宣布Oracle JDK和OpenJDK将能自由切换，Oracle JDK也会依赖OpenJDK的代码进行构建，所以通常都是使用OpenJDK的代码进行分析和研究。读者可以自行到OpenJDK的官网上下载源代码，值得一提的是，JDK的代码会随着bug修复不断改变，所以为了保持阅读的一致性，我把本书使用的代码推送到GitHub上[⊖]，也使用该版本进行编译调试。

1.2 本书常见术语

JVM系统非常复杂，市面上有很多中英文书籍从不同的角度来介绍JVM，其中都用到了很多术语，但是大家对某些术语的解释并不完全相同。为了便于读者的理解，在这里统一定义和解释本书使用的一些术语。这些术语有些是我们约定俗成的叫法，有些是JVM里面的特别约定，还有一些是G1算法引入的。为了保持准确性，这里仅仅解释这些术语的含义，后续会进一步解释相关内容，本书将尽量使用这里定义的术语。

- **并行**（parallelism），指两个或者多个事件在同一时刻发生，在现代计算机中通常指多台处理器上同时处理多个任务。
- **并发**（concurrency），指两个或多个事件在同一时间间隔内发生，在现代计算机中一台处理器“同时”处理多个任务，那么这些任务只能交替运行，从处理器的角度上看任务只能串行执行，从用户的角度看这些任务“并行”执行，实际上是处理器根据一定的策略不断地切换执行这些“并行”的任务。

⊖ https://github.com/chenghanpeng/jdk8u60

在 JVM 中，我们也常看到并行和并发。比如，典型的 ParNew 一般称为**并行收集器**，CMS 一般称为**并发标记清除**（Concurrent Mark Sweep）。这看起来很奇怪，因为并行和并发是从处理器角度出发，但是这里明显不是，实际上并行和并发在 JVM 被重新定义了。

JVM 中的并行，指多个垃圾回收相关线程在操作系统之上并发运行，这里的并行强调的是只有垃圾回收线程工作，Java 应用程序都暂停执行，因此 ParNew 工作的时候一定发生了 STW。本书提到的 ***ParTask（例如 G1ParTask）指的就是在这些任务运行的时候应用程序都必须暂停。

JVM 中的并发，指垃圾回收相关的线程并发运行（如果启动多个线程），同时这些线程会和 Java 应用程序并发运行。本书提到的 ***Concurrent***Thread（例如 ConcurrentG1RefineThread）就是指这些线程和 Java 应用程序同时运行。

- Stop-the-world（STW），直译就是停止一切，在 JVM 中指停止一切 Java 应用线程。
- **安全点**（Safepoint），指 JVM 在执行一些操作的时需要 STW，但并不是任何线程在任何地方都能进入 STW，例如我们正在执行一段代码时，线程如何能够停止？设计安全点的目的是，当线程进入到安全点时，线程就会主动停止。
- Mutator，在很多英文文献和 JVM 源码中，经常看到这个单词，它指的是我们的 Java 应用线程。Mutator 的含义是可变的，在这里的含义是因为线程运行，导致了内存的变化。GC 中通常需要 STW 才能使 Mutator 暂停。
- **记忆集**（Remember Set），简称为 RSet。主要记录不同代际对象的引用关系。
- Refine，尚未有统一的翻译，有时翻译为细化，但是不太准确，本书中不做翻译。G1 中的 ConcurrentG1RefineThread 主要指处理 RSet 的线程。
- Evacuation，转移、撤退或者回收，简称为 Evac，本书中不做翻译。在 G1 中指的是发现活跃对象，并将对象复制到新地址的过程。
- **回收**（Reclaim），通常指的是分区对象已经死亡或者已经完成 Evac，分区可以被 JVM 再次使用。
- Closure，闭包，本书中不做翻译。在 JVM 中是一种辅助类，类似于我们已知

的 iterator，它通常提供了对内存的访问。

- GC Root，垃圾回收的根。在 JVM 的垃圾回收过程中，需要从 GC Root 出发标记活跃对象，确保正在使用的对象在垃圾回收后都是存活的。
- **根集合**（Root Set）。在 JVM 的垃圾回收过程中，需要从不同的 GC Root 出发，这些 GC Root 有线程栈、monitor 列表、JNI 对象等，而这些 GC Root 就构成了 Root Set。
- Full GC，简称为 FGC，整个堆的垃圾回收动作。通常 Full GC 是串行的，G1 的 Full GC 不仅有串行实现，在 JDK10 中还有并行实现。
- **再标记**（Remark）。在本书中指的是并发标记算法中，处理完并发标记后，需要更新并发标记中 Mutator 变更的引用，这一步需要 STW。

1.3 回收算法概述

垃圾回收（Garbage Collection，GC）指的是程序不用关心对象在内存中的生存周期，创建后只需要使用对象，不用关心何时释放以及如何释放对象，由 JVM 自动管理内存并释放这些对象所占用的空间。GC 的历史非常悠久，从 1960 年 Lisp 语言开始就支持 GC。垃圾回收针对的是堆空间，目前垃圾回收算法主要有两类：

- 引用计数法：在堆内存中分配对象时，会为对象分配一段额外的空间，这个空间用于维护一个计数器，如果对象增加了一个新的引用，则将增加计数器。如果一个引用关系失效则减少计数器。当一个对象的计数器变为 0，则说明该对象已经被废弃，处于不活跃状态，可以被回收。引用计数法需要解决循环依赖的问题，在我们众所周知的 Python 语言里，垃圾回收就使用了引用计数法。
- 可达性分析法（根引用分析法），基本思路就是将根集合作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象没有被任何引用链访问到时，则证明此对象是不活跃的，可以被回收。

这两种算法各有优缺点，具体可以参考其他文献。JVM 的垃圾回收采用了可达性分析法。垃圾回收算法也一直不断地演化，主要有以下分类：

- 垃圾回收算法实现主要分为**复制**（Copy）、**标记清除**（Mark-Sweep）和**标记压缩**（Mark-Compact）。
- 在回收方法上又可以分为串行回收、并行回收、并发回收。
- 在内存管理上可以分为代管理和非代管理。

我们首先看一下基本的收集算法。

1.3.1　分代管理算法

分代管理就是把内存划分成不同的区域进行管理，其思想来源是：有些对象存活的时间短，有些对象存活的时间长，把存活时间短的对象放在一个区域管理，把存活时间长的对象放在另一个区域管理。那么可以为两个不同的区域选择不同的算法，加快垃圾回收的效率。我们假定内存被划分成 2 个代：新生代和老生代。把容易死亡的对象放在新生代，通常采用**复制算法**回收；把预期存活时间较长的对象放在老生代，通常采用**标记清除算法**。

1.3.2　复制算法

复制算法的实现也有很多种，可以使用两个分区，也可以使用多个分区。使用两个分区时内存的利用率只有 50%；使用多个分区（如 3 个分区），则可以提高内存的使用率。我们这里演示把堆空间分为 1 个新生代（分为 3 个分区：Eden、Survivor0、Survivor1）、1 个老生代的收集过程。

普通对象创建的时候都是放在 Eden 区，S0 和 S1 分别是两个存活区。第一次垃圾收集前 S0 和 S1 都为空，在垃圾收集后，Eden 和 S0 里面的活跃对象（即可以通过根集合到达的对象）都放入了 S1 区，如图 1-1 所示。

图 1-1　复制算法第一次回收