深入理解 Java 虚拟机（JVM 高级特性与最佳实践）

内容简介

作为一位 Java 程序员，你是否也曾经想深入理解 Java 虚拟机，但是却被它的复杂和深奥拒之门外？没关系，本书极尽化繁为简之妙，能带领你在轻松中领略 Java 虚拟机的奥秘。本书是近年来国内出版的唯一一本与 Java 虚拟机相关的专著，也是唯一一本同时从核心理论和实际运用这两个角度去探讨 Java 虚拟机的著作，不仅理论分析得透彻，而且书中包含的典型案例和最佳实践也极具现实指导意义。

全书共分为五大部分。第一部分从宏观的角度介绍了整个 Java 技术体系的过去、现在和未来，以及如何独立地编译一个 OpenJDK7，这对理解后面的内容很有帮助。第二部分讲解了 JVM 的自动内存管理，包括虚拟机内存区域的划分原理以及各种内存溢出异常产生的原因；常见的垃圾收集算法以及垃圾收集器的特点和工作原理；常见的虚拟机的监控与调试工具的原理和使用方法。第三部分分析了虚拟机的执行子系统，包括 Class 的文件结构以及如何存储和访问 Class 中的数据；虚拟机的类创建机制以及类加载器的工作原理和它对虚拟机的意义；虚拟机字节码的执行引擎以及它在实行代码时涉及的内存结构。第四部分讲解了程序的编译与代码的优化，阐述了泛型、自动装箱拆箱、条件编译等语法糖的原理；讲解了虚拟机的热点探测方法、HotSpot 的即时编译器、编译触发条件，以及如何从虚拟机外部观察和分析 JIT 编译的数据和结果。第五部分探讨了 Java 实现高效并发的原理，包括 JVM 内存模型的结构和操作；原子性、可见性和有序性在 Java 内存模型中的体现；先行发生原则的规则和使用；线程在 Java 语言中的实现原理；虚拟机实现高效并发所做的一系列锁优化措施。

本书适合所有 Java 程序员、系统调优师和系统架构师阅读。

作者简介

周志明，资深 Java 技术专家，对 JavaEE 企业级应用开发、OSGi、Java 虚拟机和工作流等都有深入的研究，并在大量的实践中积累了丰富的经验。尤其精通 Java 虚拟机，撰写了大量与 JVM 相关的经典文章，被各大技术社区争相转载，是 ITeye 等技术社区公认的 Java 虚拟机方面的领袖人物之一。现任远光软件股份有限公司平台开发部经理兼平台架构师，先后参加与过国家电网、南方电网等多个国家级大型 ERP 项目的平台架构工作，对软件系统架构也有深刻的认识和体会。

专家推荐

Java 程序是如何运行的？Java 虚拟机在其中扮演了怎样的角色？如何让 Java 程序具有更高的并发性？许多 Java 程序员都会诸如此类的疑问。无奈，国内在很长一段时间里都没有一本从实际应用的角度讲解 Java 虚拟机的著作，本书的出版可谓填补了这个空白。它从 Java 程序员的角度出发，系统地将 Java 程序运行过程中涉及的各种知识整合到了一起，并配以日常工作中可能会碰到的疑难案例，引领读者轻松踏上探索 Java 虚拟机的旅途，是广大对 Java 虚拟机感兴趣的读者的福音！——莫枢（RednaxelaFX） 虚拟机和编程语言爱好者

在武侠的世界里，无论是至刚至强的《易筋经》，还是阴柔无比的《葵花宝典》，都离不开内功修炼。没有了内功心法，这些武术只是花拳绣腿的拙劣表演而已。软件业是武林江湖的一个翻版，也有着大量的模式、套路、规范等外功，但“外功修行，内功修神”，要想成为“扫地僧”一样的绝世高人，此书是必备的。——秦小波 资深 Java 技术专家/著有畅销书《设计模式之禅》

对 Java 程序员来说，Java 虚拟机可以说是既熟悉又神秘，很少 Java 程序员能够抑制自己探究它的冲动。可惜，分析 JVM 实现原理的书籍（特别是国内作者出版的）是少之又少。本书的出版可谓 Java 程序员的福音，作者将自己多年来在 Java 虚拟机领域的实践经验和研究心得呈现在了这本书中，不仅系统地讲解了 Java 虚拟机工作机制和底层原理，而且更难能可贵的是与实践很好地结合了起来，具有非常强的实践指导意义，强烈推荐！——计文柯 资深 Java 技术专家/著有畅销书《Spring 技术内幕：深入解析 Spring 架构设计与实现原理》

前言

Java 是目前用户最多、使用范围最广的软件开发技术，Java 的技术体系主要由支撑 Java 程序运行的虚拟机、为各开发领域提供接口支持的 Java API、Java 编程语言及许许多多的第三方 Java 框架（如 Spring 和 Struts 等）构成。在国内，有关 Java API、Java 语言及第三方框架的技术资料和书籍非常丰富，相比之下，有关 Java 虚拟机的资料却显得异常贫乏。

这种状况很大程度上是由 Java 开发技术本身的一个重要优点导致的：在虚拟机层面隐藏了底层技术的复杂性以及机器与操作系统的差异性。运行程序的物理机器情况千差万别，而 Java 虚拟机则在千差万别的物理机上面建立了统一的运行平台，实现了在任意一台虚拟机上编译的程序都能在任何一台虚拟机上正常运行。这一极大的优势使得 Java 应用的开发比传统 C/C++应用的开发更高效和快捷，程序员可以把主要精力集中在具体业务逻辑上，而不是物理硬件的兼容性上。一般情况下，一个程序员只要了解了必要的 Java API、Java 语法并学习适当的第三方开发框架，就已经基本能满足日常开发的需要了，虚拟机会在用户不知不觉中完成对硬件平台的兼容以及对内存等资源的管理工作。因此，了解虚拟机的运作并不是一般开发人员必须掌握的知识。

然而，凡事都具备两面性。随着 Java 技术的不断发展，它被应用于越来越多的领域之中。其中一些领域，如电力、金融、通信等，对程序的性能、稳定性和可扩展性方面都有极高的要求。一个程序很可能在 10 个人同时使用时完全正常，但是在 10000 个人同时使用时就会变慢、死锁甚至崩溃。毫无疑问，要满足 10000 个人同时使用需要更高性能的物理硬件，但是在绝大多数情况下，提升硬件效能无法等比例地提升程序的性能和并发能力，有时甚至可能对程序的性能没有任何改善作用。这里面有 Java 虚拟机的原因：为了达到为所有硬件提供一致的虚拟平台的目的，牺牲了一些硬件相关的性能特性。更重要的是人为原因：开发人员如果不了解虚拟机的一些技术特性的运行原理，就无法写出最适合虚拟机运行和可自优化的代码。

其实，目前商用的高性能 Java 虚拟机都提供了相当多的优化特性和调节手段，用于满足应用程序在实际生产环境中对性能和稳定性的要求。如果只是为了入门学习，让程序在自己的机器上正常运行，那么这些特性可以说是可有可无的；如果用于生产环境，尤其是企业级应用开发中，就迫切需要开发人员中至少有一部分人对虚拟机的特性及调节方法具有很清晰的认识，所以在 Java 开发体系中，对架构师、系统调优师、高级程序员等角色的需求一直都非常大。学习虚拟机中各种自动运作的特性的原理也成为了 Java 程序员成长道路上必然会接触到的一课。通过本书，读者可以以一种相对轻松的方式学习虚拟机的运作原理，对 Java 程序员的成长也有较大的帮助。

本书读者对象

（1） 使用 Java 技术体系的中、高级开发人员

Java 虚拟机作为中、高级开发人员必须修炼的知识，有着较高的学习门槛，本书可作为学习虚拟机的优秀教材。

（2） 系统调优师

系统调优师是近几年才兴起的职业，本书中的大量案例、代码和调优实战将会对系统调优师的日常工作有直接的帮助。

（3） 系统架构师

保障系统高效、稳定和可伸缩是系统架构师的主要职责之一，而这与虚拟机的运作密不可分，本书可以作为他们设计应用系统底层框架的参考资料。

如何阅读本书

本书一共分为五个部分：走近 Java、自动内存管理机制、虚拟机执行子系统、程序编译与代码优化、高效并发。各个部分基本上是相互独立的，没有必然的前后依赖关系，读者可以从任何一个感兴趣的专题开始阅读，但是每个部分中的各个章节间有先后顺序。

本书并不假设读者在 Java 领域具备很专业的技术水平，因此在保证逻辑准确的前提下，尽量用通俗的语言和案例讲述虚拟机中与开发关系最为密切的内容。当然学习虚拟机技术本身就需要读者有一定的技术基础，且本书的读者定位是中、高级程序员，因此本书假设读者自己了解一些常用的开发

框架、Java API 和 Java 语法等基础知识。

语言约定

本书在语言和技术上有如下的约定：

本书中提到 HotSpot 虚拟机、JRockit 虚拟机、WebLogic 服务器等产品的所有者时，仍然使用 Sun 和 BEA 公司的名称。实际上 BEA 和 Sun 分别于 2008 年和 2010 年被 Oracle 公司收购，现在已经不存在这两个商标了，但是毫无疑问它们都是对 Java 领域做出过卓越贡献的、值得程序员纪念的公司。

JDK 从 1.5 版本开始，在官方的正式文档与宣传资料中已经不再使用类似“JDK 1.5”的名称，只有在程序员内部使用的开发版本号 （Developer Version，例如 java -version 的输出）中才继续沿用 1.5、1.6 和 1.7 的版本号，而公开版本号（Product Version）则改为 JDK 5、JDK 6 和 JDK 7 的命名方式。为了行文一致，本书所有场合统一采用开发版本号的命名方式。

由于版面关系，本书中的许多示例代码都没有遵循最优的代码编写风格，如使用的流没有关闭流等，请读者在阅读时注意这一点。

如果没有特殊说明，本书中所有的讨论都是以 Sun JDK 1.6 为技术平台的。不过如果有某个特性在各个版本间的变化较大，一般都会说明它在各个版本间的差异。

内容特色

第一部分　走近 Java

本书的第一部分为后文的讲解建立了良好的基础。尽管了解 Java 技术的来龙去脉，以及编译自己的 OpenJDK 对于读者理解 Java 虚拟机并不是必需的，但是这些准备过程可以为走近 Java 技术和 Java 虚拟机提供很好的引导。第一部分只有第 1 章：

第 1 章　介绍了 Java 技术体系的过去、现在和未来的发展趋势，并介绍了如何独立编译一个 OpenJDK 7。

第二部分　自动内存管理机制

因为程序员把内存控制的权力交给了 Java 虚拟机，所以可以在编码的时候享受自动内存管理的诸多优势，不过也正因为这个原因，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那么排查错误将会成为一项异常艰难的工作。第二部分包括第 2～5 章：

第 2 章　讲解了虚拟机中的内存是如何划分的，哪部分区域、什么样的代码和操作可能导致内存溢出异常，并讲解了各个区域出现内存溢出异常的常见原因。

第 3 章　分析了垃圾收集的算法和 JDK 1.6 中提供的几款垃圾收集器的特点及运作原理，通过代码实例验证了 Java 虚拟机中的自动内存分配及回收的主要规则。

第 4 章　介绍了随 JDK 发布的 6 个命令行工具与 2 个可视化的故障处理工具的使用方法。

第 5 章　与读者分享了几个比较有代表性的实际案例，还准备了一个所有开发人员都能“亲身实战”的练习，读者可通过实践来获得故障处理和调优的经验。

第三部分　虚拟机执行子系统

执行子系统是虚拟机中必不可少的组成部分，了解了虚拟机如何执行程序，才能写出更优秀的代码。第三部分包括第 6～9 章：

第 6 章　讲解了 Class 文件结构中的各个组成部分，以及每个部分的定义、数据结构和使用方法，以实战的方式演示了 Class 的数据是如何存储和访问的。

第 7 章　介绍了在类加载过程的“加载”、“验证”、“准备”、“解析”和“初始化”这五个阶段中虚拟机分别执行了哪些动作，还介绍了类加载器的工作原理及其对虚拟机的意义。

第 8 章　分析了虚拟机在执行代码时如何找到正确的方法，如何执行方法内的字节码，以及执行代码时涉及的内存结构。

第 9 章　通过四个类加载及执行子系统的案例，分享了使用类加载器和处理字节码的一些值得欣赏和借鉴的思路，并通过一个实战练习来加深对前面理论知识的理解。

第四部分　程序编译与代码优化

Java 程序从源码编译成字节码和从字节码编译成本地机器码的这两个过程，合并起来其实就等同于一个传统编译器所执行的编译过程。第四部分包括第 10 和 11 章：

第 10 章　分析了 Java 语言中的泛型、自动装箱拆箱、条件编译等多种语法糖的前因后果，并通过实战案例演示了如何使用插入式注解处理器来实现一个检查程序命名规范的编译器插件。

第 11 章　讲解了虚拟机的热点探测方法、HotSpot 的即时编译器、编译触发条件，以及如何从虚拟机外部观察和分析 JIT 编译的数据和结果。此外，还讲解了几种常见的编译期优化技术。

第五部分　高效并发

Java 语言和虚拟机提供了原生的、完善的多线程支持，使得它天生就适合开发多线程并发的应用程序。不过我们不能期望系统来完成所有与并发相关的处理，了解并发的内幕也是一个高级程序员不可缺少的课程。第五部分包括第 12 和 13 章：

第 12 章　讲解了虚拟机的 Java 内存模型的结构和操作，以及原子性、可见性和有序性在 Java 内存模型中的体现，介绍了先行发生原则及使用，还讲解了线程在 Java 语言中是如何实现的。

第 13 章　介绍了线程安全所涉及的概念和分类、同步实现的方式以及虚拟机的底层运作原理，并且还介绍了虚拟机实现高效并发所采取的一系列锁优化措施。

参考资料

本书名为“深入理解 Java 虚拟机”，但要想真的深入理解虚拟机，仅凭一本书肯定是远远不够的，读者可以通过下面的信息找到更多关于 Java 虚拟机方面的资料。我在写作此书的时候，也从下面这些参考资料中获得了很大的帮助。

（1）书籍

《The Java Virtual Machine Specification, Second Edition》①

《Java 虚拟机规范（第 2 版）》，1999 年 4 月出版。国内并没有引进这本书，自然也就没有中文译本，但全书的电子版是免费发布的，在 IT 书籍中它已经非常“高寿”了①（这本书的第 3 版已处于基本完成的草稿状态，在 JDK 1.7 正式版发布后这本书应该就会推出第 3 版）。要学习虚拟机，虚拟机规范无论如何都是必须读的。这本书的概念和细节描述与 Sun 的早期虚拟机（Sun Classic VM）高度吻合，不过，随着技术的发展，高性能虚拟机真正的细节实现方式与虚拟机规范所描述的差距已经越来越大。但是，如果只能选择一本参考书来了解虚拟机的话，那仍然是这本书。

《The Java Language Specification, Third Edition》②

《Java 语言规范（第 3 版）》，2005 年 7 月由机械工业出版社出版，不过出版的是影印版，没有中文译本。虽然 Java 虚拟机并不是 Java 语言专有的，但是了解 Java 语言的各种细节规定对虚拟机的行为也是很有帮助的，它与《Java 虚拟机规范（第 2 版）》都是 Sun 官方出品的书籍，而且这本书还是由 Java 之父 James Gosling 亲自撰写的。

《Oracle JRockit The Definitive Guide》

《Oracle JRockit 权威指南》，2010 年 7 月出版，国内也没有（可能是尚未）引进这本书，它是由 JRockit 的两位资深开发人员（其中一位是 JRockit Mission Control 团队的 TeamLeader）撰写的高级 JRockit 虚拟机使用指南。虽然 JRockit 的用户量可能不如 HotSpot 多，但也是最流行的三大商业虚拟机之一，并且不同虚拟机中的很多实现思路都是可以对比参照的。这本书是了解现代高性能虚拟机的很好的途径。

《Inside the Java 2 Virtual Machine, Second Edition》

《深入 Java 虚拟机（第 2 版）》，2000 年 1 月出版，2003 年机械工业出版社出版了中文译本。在相当长的时间里，这本书是唯一一本关于 Java 虚拟机的中文图书。

（2）网站资源

高级语言虚拟机圈子：http://hllvm.group.iteye.com/

里面有一些国内关于虚拟机的讨论，并不只限于 JVM，而是涉及对所有的高级语言虚拟机（High-Level Language Virtual Machine）的讨论，但该网站建立在 ITEye①上，自然还是以讨论 Java 虚拟机为主。圈主撒迦（莫枢）的博客（http://rednaxelafx.iteye.com/）是另外一个非常有价值的虚拟机及编译原理等资料的分享园地。

HotSpot Internals：http://wikis.sun.com/display/HotSpotInternals/Home

一个关于 OpenJDK 的 Wiki 网站，许多文章都由 JDK 的开发团队编写，更新很慢，但是仍然有很大的参考价值。

The HotSpot Group：http://openjdk.java.net/groups/hotspot/

HotSpot 组群，包含虚拟机开发、编译器、垃圾收集和运行时四个邮件组，其中有关于 HotSpot 虚拟机的最新讨论。

联系作者

在本书完稿时，我并没有像想象中那样兴奋或放松，写作时的那种“战战兢兢、如履薄冰”的感觉依然萦绕在心头。在每一章、每一节落笔之时，我都在考虑如何才能把各个知识点更有条理地讲述出来，都在担心会不会由于自己理解有偏差而误导了大家。囿于我的写作水平和写作时间，书中难免存在不妥之处，所以特地开通了一个读者邮箱（understandingjvm@gmail.com）与大家交流，大家如有任何意见或建议都欢迎与我联系。

此外，大家也可以通过我的微博（http://t.sina.com.cn//icyfenix）与我取得联系。

勘误

写书和写代码一样，刚开始都是不完美的，需要不断地修正和重构，本书也不例外。如果大家在阅读本书的过程中发现了本书中存在的任何问题，都欢迎反馈给我们。我们会把本书的勘误集中公布在 www.yangfuchuan.com。

目录

目　录

前　言

致　谢

第一部分　走近 Java

第 1 章　走近 Java / 2

1.1　概述 / 2

1.2　Java 技术体系 / 3

1.3　Java 发展史 / 5

1.4　展望 Java 技术的未来 / 9

1.4.1　模块化 / 9

1.4.2　混合语言 / 9

1.4.3　多核并行 / 11

1.4.4　进一步丰富语法 / 12

1.4.5　64 位虚拟机 / 13

1.5　实战：自己编译 JDK / 13

1.5.1　获取 JDK 源码 / 13

1.5.2　系统需求 / 14

1.5.3　构建编译环境 / 15

1.5.4　准备依赖项 / 17

1.5.5　进行编译 / 18

1.6　本章小结 / 21

第二部分　自动内存管理机制

第 2 章　Java 内存区域与内存溢出异常 / 24

2.1　概述 / 24

2.2　运行时数据区域 / 25

2.2.1　程序计数器 / 25

2.2.2　Java 虚拟机栈 / 26

2.2.3　本地方法栈 / 27

2.2.4　Java 堆 / 27

2.2.5　方法区 / 28

2.2.6　运行时常量池 / 29

2.2.7　直接内存 / 29

2.3　对象访问 / 30

2.4　实战：OutOfMemoryError 异常 / 32

2.4.1　Java 堆溢出 / 32

2.4.2　虚拟机栈和本地方法栈溢出 / 35

2.4.3　运行时常量池溢出 / 38

2.4.4　方法区溢出 / 39

2.4.5　本机直接内存溢出 / 41

2.5　本章小结 / 42

第 3 章　垃圾收集器与内存分配策略 / 43

3.1　概述 / 43

3.2　对象已死？ / 44

3.2.1　引用计数算法 / 44

3.2.2　根搜索算法 / 46

3.2.3　再谈引用 / 47

3.2.4　生存还是死亡？ / 48

3.2.5　回收方法区 / 50

3.3　垃圾收集算法 / 51

3.3.1 标记 -清除算法 / 51

3.3.2 复制算法 / 52

3.3.3 标记-整理算法 / 54

3.3.4 分代收集算法 / 54

3.4 垃圾收集器 / 55

3.4.1 Serial 收集器 / 56

3.4.2 ParNew 收集器 / 57

3.4.3 Parallel Scavenge 收集器 / 59

3.4.4 Serial Old 收集器 / 60

3.4.5 Parallel Old 收集器 / 61

3.4.6 CMS 收集器 / 61

3.4.7 G1 收集器 / 64

3.4.8 垃圾收集器参数总结 / 64

3.5 内存分配与回收策略 / 65

3.5.1 对象优先在 Eden 分配 / 66

3.5.2 大对象直接进入老年代 / 68

3.5.3 长期存活的对象将进入老年代 / 69

3.5.4 动态对象年龄判定 / 71

3.5.5 空间分配担保 / 73

3.6 本章小结 / 75

第 4 章 虚拟机性能监控与故障处理工具 / 76

4.1 概述 / 76

4.2 JDK 的命令行工具 / 76

4.2.1 jps：虚拟机进程状况工具 / 79

4.2.2 jstat：虚拟机统计信息监视工具 / 80

4.2.3 jinfo：Java 配置信息工具 / 82

4.2.4 jmap：Java 内存映像工具 / 82

4.2.5 jhat：虚拟机堆转储快照分析工具 / 84

4.2.6 jstack：Java 堆栈跟踪工具 / 85

4.3 JDK 的可视化工具 / 87

4.3.1 JConsole：Java 监视与管理控制台 / 88

4.3.2 VisualVM：多合一故障处理工具 / 96

4.4　本章小结 / 105
第 5 章　调优案例分析与实战 / 106
5.1　概述 / 106
5.2　案例分析 / 106
5.2.1　高性能硬件上的程序部署策略 / 106
5.2.2　集群间同步导致的内存溢出 / 109
5.2.3　堆外内存导致的溢出错误 / 110
5.2.4　外部命令导致系统缓慢 / 112
5.2.5　服务器 JVM 进程崩溃 / 113
5.3　实战：Eclipse 运行速度调优 / 114
5.3.1　调优前的程序运行状态 / 114
5.3.2　升级 JDK 1.6 的性能变化及兼容问题 / 117
5.3.3　编译时间和类加载时间的优化 / 122
5.3.4　调整内存设置控制垃圾收集频率 / 126
5.3.5　选择收集器降低延迟 / 130
5.4　本章小结 / 133
第三部分　虚拟机执行子系统
第 6 章　类文件结构 / 136
6.1　概述 / 136
6.2　无关性的基石 / 136
6.3　Class 类文件的结构 / 138
6.3.1　魔数与 Class 文件的版本 / 139
6.3.2　常量池 / 141
6.3.3　访问标志 / 147
6.3.4　类索引、父类索引与接口索引集合 / 148
6.3.5　字段表集合 / 149
6.3.6　方法表集合 / 153
6.3.7　属性表集合 / 155
6.4　Class 文件结构的发展 / 168
6.5　本章小结 / 170
第 7 章　虚拟机类加载机制 / 171
7.1　概述 / 171

7.2　类加载的时机 / 172
7.3　类加载的过程 / 176
7.3.1　加载 / 176
7.3.2　验证 / 178
7.3.3　准备 / 181
7.3.4　解析 / 182
7.3.5　初始化 / 186
7.4　类加载器 / 189
7.4.1　类与类加载器 / 189
7.4.2　双亲委派模型 / 191
7.4.3　破坏双亲委派模型 / 194
7.5　本章小结 / 197
第 8 章　虚拟机字节码执行引擎 / 198
8.1　概述 / 198
8.2　运行时栈帧结构 / 199
8.2.1　局部变量表 / 199
8.2.2　操作数栈 / 204
8.2.3　动态连接 / 206
8.2.4　方法返回地址 / 206
8.2.5　附加信息 / 207
8.3　方法调用 / 207
8.3.1　解析 / 207
8.3.2　分派 / 209
8.4　基于栈的字节码解释执行引擎 / 221
8.4.1　解释执行 / 221
8.4.2　基于栈的指令集与基于寄存器的指令集 / 223
8.4.3　基于栈的解释器执行过程 / 224
8.5　本章小结 / 230
第 9 章　类加载及执行子系统的案例与实战 / 231
9.1　概述 / 231
9.2　案例分析 / 231
9.2.1　Tomcat：正统的类加载器架构 / 232

9.2.2　OSGi：灵活的类加载器架构 / 235
9.2.3　字节码生成技术与动态代理的实现 / 238
9.2.4　Retrotranslator：跨越 JDK 版本 / 242
9.3　实战：自己动手实现远程执行功能 / 246
9.3.1　目标 / 246
9.3.2　思路 / 247
9.3.3　实现 / 248
9.3.4　验证 / 255
9.4　本章小结 / 256
第四部分　程序编译与代码优化
第 10 章　早期（编译期）优化 / 258
10.1　概述 / 258
10.2　Javac 编译器 / 259
10.2.1　Javac 的源码与调试 / 259
10.2.2　解析与填充符号表 / 262
10.2.3　注解处理器 / 264
10.2.4　语义分析与字节码生成 / 264
10.3　Java 语法糖的味道 / 268
10.3.1　泛型与类型擦除 / 268
10.3.2　自动装箱、拆箱与遍历循环 / 273
10.3.3　条件编译 / 275
10.4　实战：插入式注解处理器 / 276
10.4.1　实战目标 / 276
10.4.2　代码实现 / 277
10.4.3　运行与测试 / 284
10.4.4　其他应用案例 / 286
10.5　本章小结 / 286
第 11 章　晚期（运行期）优化 / 287
11.1　概述 / 287
11.2　HotSpot 虚拟机内的即时编译器 / 288
11.2.1　解释器与编译器 / 288
11.2.2　编译对象与触发条件 / 291

11.2.3　编译过程 / 294
11.2.4　查看与分析即时编译结果 / 297
11.3　编译优化技术 / 301
11.3.1　优化技术概览 / 301
11.3.2　公共子表达式消除 / 305
11.3.3　数组边界检查消除 / 307
11.3.4　方法内联 / 307
11.3.5　逃逸分析 / 309
11.4　Java 与 C/C++的编译器对比 / 311
11.5　本章小结 / 313
第五部分　高效并发
第 12 章　Java 内存模型与线程 / 316
12.1　概述 / 316
12.2　硬件的效率与一致性 / 317
12.3　Java 内存模型 / 318
12.3.1　主内存与工作内存 / 319
12.3.2　内存间交互操作 / 320
12.3.3　对于 volatile 型变量的特殊规则 / 322
12.3.4　对于 long 和 double 型变量的特殊规则 / 327
12.3.5　原子性、可见性与有序性 / 328
12.3.6　先行发生原则 / 330
12.4　Java 与线程 / 333
12.4.1　线程的实现 / 333
12.4.2　Java 线程调度 / 337
12.4.3　状态转换 / 339
12.5　本章小结 / 341
第 13 章　线程安全与锁优化 / 342
13.1　概述 / 342
13.2　线程安全 / 343
13.2.1　Java 语言中的线程安全 / 343
13.2.2　线程安全的实现方法 / 348
13.3　锁优化 / 356

13.3.1　自旋锁与自适应自旋 / 356

13.3.2　锁消除 / 357

13.3.3　锁粗化 / 358

13.3.4　轻量级锁 / 358

13.3.5　偏向锁 / 361

13.4　本章小结 / 362

附录 A　Java 虚拟机家族 / 363

附录 B　虚拟机字节码指令表 / 366

附录 C　HotSpot 虚拟机主要参数表 / 372

附录 D　对象查询语言（OQL）简介 / 376

附录 E　JDK 历史版本轨迹 / 383

第 1 章　走近 Java

本章主要内容

概述

Java 技术体系

Java 发展史

展望 Java 技术的未来

实战：自己编译 JDK

世界上并没有完美的程序，但我们并不因此而沮丧，因为写程序本来就是一个不断追求完美的过程。

1.1　概述

Java 不仅仅是一门编程语言，它还是一个由一系列计算机软件和规范形成的技术体系，这个技术体系提供了完整的用于软件开发和跨平台部署的支持环境，并广泛应用于嵌入式系统、移动终端、企业服务器和大型机等各种场合，如图 1-1 所示。时至今日，Java 技术体系已经吸引了 600 多万软件开发者，这是全球最大的软件开发团队。使用 Java 的设备多达几十亿台，其中包括 8 亿多台个人计算机、21 亿部移动电话及其他手持设备、35 亿个智能卡，以及大量机顶盒、导航系统和其他设备①。

Java 能获得如此广泛的认可，除了因为它拥有一门结构严谨、面向对象的编程语言之外，还有许多不可忽视的优点：它摆脱了硬件平台的束缚，实现了“一次编写，到处运行”的理想；它提供了一种相对安全的内存管理和访问机制，避免了绝大部分的内存泄漏和指针越界问题；它实现了热点代码

检测和运行时编译及优化，这使得 Java 应用能随着运行时间的增加而获得更高的性能；它有一套完善的应用程序接口，还有无数的来自商业机构和开源社区的第三方类库来帮助实现各种各样的功能……Java 所带来的这些好处让程序的开发效率得到了很大的提升。作为一名 Java 程序员，在编写程序时除了尽情发挥 Java 的各种优势外，还应该去了解和思考一下 Java 技术体系中这些技术是如何实现的。认清这些技术的运作本质，是自己思考“程序这样写好不好”的基础和前提。当我们在使用一门技术时，如果不再依赖书本和他人就能得到这个问题的答案，那才算升华到了“不惑”的境界。

图 1-1　Java 技术的广泛应用

本书将会与读者一起分析 Java 技术中最重要的那些特性的实现原理。在本章中，我们将重点介绍 Java 技术体系所包括的内容，以及 Java 的历史、现在和未来的发展趋势。

1.2　Java 技术体系

从广义上讲，Clojure、JRuby、Groovy 等运行于 Java 虚拟机上的语言及其相关的程序都属于 Java 技术体系的一员。如果仅从传统意义上来看，Sun 官方所定义的 Java 技术体系包括了以下几个组成部分：

Java 程序设计语言

各种硬件平台上的 Java 虚拟机

Class 文件格式

Java API 类库

来自商业机构和开源社区的第三方 Java 类库

我们可以把 Java 程序设计语言、Java 虚拟机、Java API 类库这三部分统称为 JDK（Java Development Kit），JDK 是用于支持 Java 程序开发的最小环境，在后面的内容中，为了讲解方便，有一些地方会以 JDK 来代替整个 Java 技术体系。另外，可以把 Java API 类库中的 Java SE API 子集①和 Java 虚拟机这两部分统称为 JRE（Java Runtime Environment），JRE 是支持 Java 程序运行的标准环境。图 1-2 展示了 Java 技术体系所包括的内容，以及 JDK 和 JRE 所涵盖的范围。

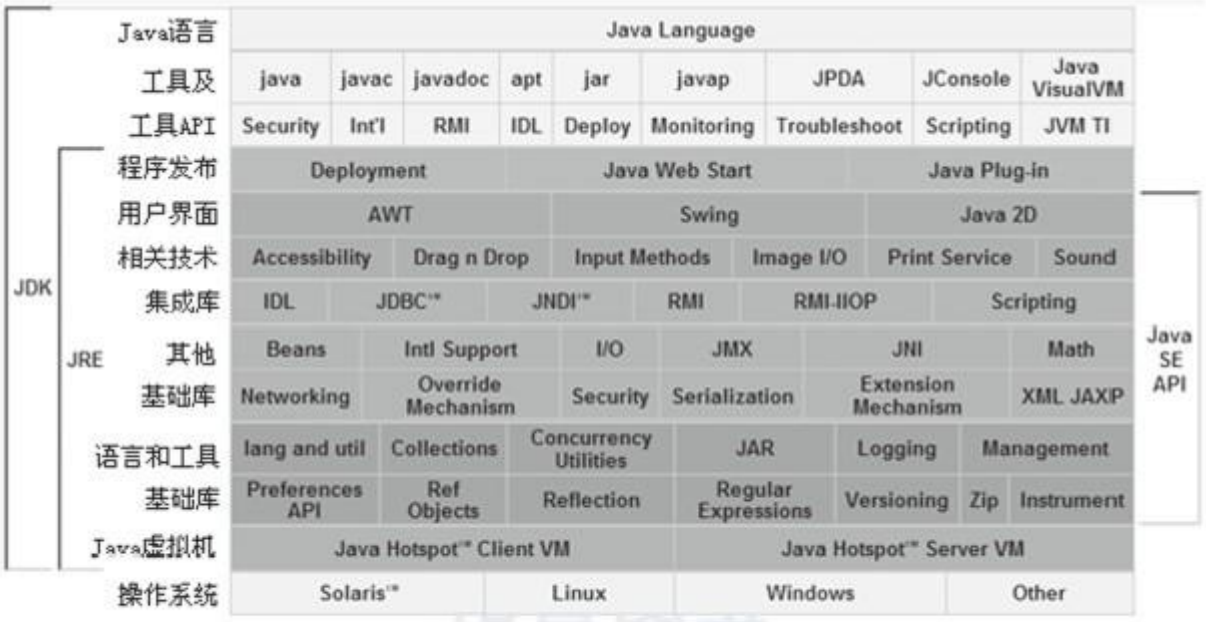

图 1-2　Java 技术体系所包括的内容②

以上是根据各个组成部分的功能来进行划分的，如果按照技术所服务的领域来划分，或者说按照 Java 技术关注的重点业务领域来划分，Java 技术体系可以分为四个平台，分别为：

Java Card：支持一些 Java 小程序（Applets）运行在小内存设备（如智能卡）上的平台。

Java ME（Micro Edition）：支持 Java 程序运行在移动终端（手机、PDA）上的平台，对 Java API 有所精简，并加入了针对移动终端的支持，这个版本以前称为 J2ME。

Java SE（Standard Edition）：支持面向桌面级应用（如 Windows 下的应用程序）的 Java 平台，提供了完整的 Java 核心 API，这个版本以前称为 J2SE。

Java EE（Enterprise Edition）：支持使用多层架构的企业应用（如 ERP、CRM 应用）的 Java 平台，除了提供 Java SE API 外，还对其做了大量的扩充①并提供了相关的部署支持，这个版本以前称为 J2EE。

1.3 Java 发展史

从第一个 Java 版本诞生到现在已经有 16 年的时间了。沧海桑田一瞬间，转眼 16 年过去了，在图 1-3 所展示的时间线中，我们看到 JDK 已经发展到了 1.7 版。在这 16 年里还诞生了无数和 Java 相关的产品、技术和标准。现在让我们进入时间隧道，从孕育 Java 语言的时代开始，再来回顾一下 Java 的发展轨迹和历史变迁。

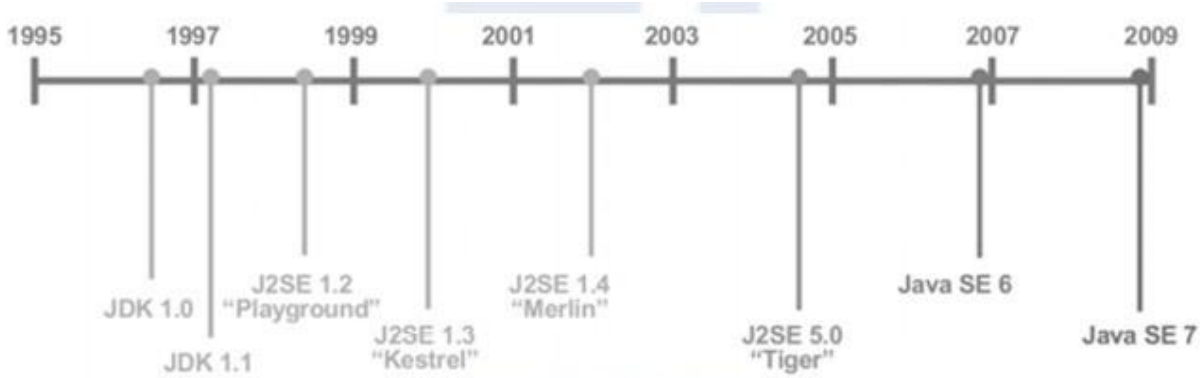

图 1-3 Java 技术发展的时间线

1991 年 4 月，由 James Gosling 博士领导的绿色计划（Green Project）开始启动，此计划的目的是开发一种能够在各种消费性电子产品（如机顶盒、冰箱、收音机等）上运行的程序架构。这个计划的产品就是 Java 语言的前身：Oak（橡树）。Oak 当时在消费品市场上并不算成功，但随着 1995 年互联网潮流的兴起，Oak 迅速找到了最适合自己发展的市场定位并蜕变成为 Java 语言。

1995 年 5 月 23 日，Oak 语言改名为 Java，并且在 SunWorld 大会上正式发布了 Java1.0 版本。Java 语言第一次提出了“Write Once，Run Anywhere”的口号。

1996 年 1 月 23 日，JDK 1.0 发布，Java 语言有了第一个正式版本的运行环境。JDK 1.0 提供了一个纯解释执行的 Java 虚拟机实现（Sun Classic VM）。JDK 1.0 版本的代表技术包括：Java 虚拟机、Applet 和 AWT 等。

1996 年 4 月，10 个最主要的操作系统供应商声明将在其产品中嵌入 Java 技术。同年 9 月，已有大约 8.3 万个网页应用 Java 技术来制作。在 1996 年 5 月底，Sun 于美国旧金山举行了首届 JavaOne 大会，从此 JavaOne 成为全世界数百万 Java 语言开发者每年一度的技术盛会。

1997 年 2 月 19 日，Sun 发布了 JDK 1.1，Java 技术的一些最基础的支撑点（如 JDBC 等）都是在 JDK 1.1 版本中发布的，JDK 1.1 的技术代表有：JAR 文件格式、JDBC、JavaBeans、RMI。Java 语法也有了一定的发展，如内部类（Inner Class）和反射（Reflection）都是在这个时候出现的。

直到 1999 年 4 月 8 日，JDK 1.1 一共发布了 1.1.0 至 1.1.8 九个版本。从 1.1.4 之后，每个 JDK 版本都有一个自己的名字（工程代号），分别为：JDK 1.1.4 - Sparkler（宝石）、JDK 1.1.5 - Pumpkin（南瓜）、JDK 1.1.6 - Abigail（阿比盖尔，女子名）、JDK 1.1.7- Brutus（布鲁图，古罗马政治家和将军）和 JDK 1.1.8 - Chelsea（切尔西，城市名）。

1998 年 12 月 4 日，JDK 迎来了一个里程碑式的版本 JDK 1.2，工程代号为 Playground（竞技场），Sun 在这个版本中把 Java 技术体系拆分为 3 个方向，分别是面向桌面应用开发的 J2SE（Java 2 Platform， Standard Edition）、面向企业级开发的 J2EE（Java 2 Platform， Enterprise Edition）和面向手机等移动终端开发的 J2ME（Java 2 Platform， Micro Edition）。在这个版本中出现的代表性技术非常多，如 EJB、Java Plug-in、Java IDL、Swing 等，并且在这个版本中 Java 虚拟机第一次内置了 JIT（Just In Time）编译器（JDK 1.1 也可以使用外挂方式的 JIT 编译器）。在语言和 API 级别上，Java 添加了 strictfp 关键字与现在 Java 编码之中极为常用的一系列 Collections 集合类。在 1999 年 3 月和 7 月，分别有 JDK 1.2.1 和 JDK 1.2.2 两个小版本发布。

1999 年 4 月 27 日，HotSpot 虚拟机发布，HotSpot 最初由一家名为“Longview Technologies”的小公司开发，因为 HotSpot 的优异表现，这家公司在 1997 年被 Sun 公司收购了。HotSpot 虚拟机发布时是作为 JDK 1.2 的附加程序提供的，后来它成为了 JDK 1.3 及之后所有版本的 Sun JDK 的默认虚拟机。

2000 年 5 月 8 日，工程代号为 Kestrel（美洲红隼）的 JDK 1.3 发布，JDK 1.3 相对于 JDK 1.2 的改进主要表现在一些类库（如数学运算和新的 Timer API 等）上，JNDI 服务从 JDK 1.3 开始被作为一项平台级服务提供（以前 JNDI 仅仅是一项扩展），使用 CORBA IIOP 来实现 RMI 的通讯协议，等等。这个版本还对 Java 2D 做了很多改进，提供了大量新的 Java 2D API，并且新添加了 JavaSound 类库。JDK 1.3 有 1 个修正版本 JDK 1.3.1，工程代号为 Ladybird（瓢虫）于 2001 年 5 月 17 日发布。

自从 JDK 1.3 开始，Sun 维持了一个习惯：大约每隔两年发布一个 JDK 的主版本，以动物名称命名，期间发布的各个修正版本则以昆虫名称作为工程名称。

2002 年 2 月 13 日，JDK 1.4 发布，工程代号为 Merlin（灰背隼）。JDK 1.4 是 Java 真正走向成熟的一个版本，Compaq、Fujitsu、SAS、Symbian、IBM 等著名公司都有参与甚至实现自己独立的 JDK 1.4。哪怕是在近 10 年后的今天，仍然有许多主流应用（Spring、Hibernate、Struts 等）能直接运行在 JDK 1.4 之上，或者继续发布能运行在 1.4 上的版本。JDK 1.4 同样发布了很多新的技术特性，如正则表达式、异常链、NIO、日志类、XML 解析器和 XSLT 转换器，等等。JDK 1.4 有两个后续修正版：2002 年 9 月 16 日发布的工程代号为 Grasshopper（蚱蜢）的 JDK 1.4.1 与 2003 年 6 月 26 日发布的工程代号为 Mantis（螳螂）的 JDK 1.4.2。

2002 年前后还发生了一件与 Java 没有直接关系，但事实上对 Java 的发展进程影响很大的事件，即微软的.NET Framework 发布。这个无论是技术实现还是目标用户上都与 Java 有很多相近之处的技

术平台给 Java 带来了很多讨论、比较和竞争，.NET 平台和 Java 平台之间声势浩大的孰优孰劣的论战到今天为止仍然在继续。

2004 年 9 月 30 日，JDK 1.5 发布①，工程代号为 Tiger（老虎）。从 JDK 1.2 以来，Java 在语法层面上的变化一直很小，而 JDK 1.5 在 Java 语法易用性上做出了非常大的改进。自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环（foreach 循环）等语法特性都是在 JDK 1.5 中加入的。在虚拟机和 API 层面上，这个版本改进了 Java 的内存模型（Java Memory Model，JMM）、提供了 java.util.concurrent 并发包等。另外，JDK 1.5 是官方声明可以支持 Windows 9x 平台的最后一个 JDK 版本。

2006 年 12 月 11 日，JDK 1.6 发布，工程代号为 Mustang（野马）。这是目前为止最新的正式版 JDK（截至本书完稿时，JDK 1.7 仍然处于 Early Access 版本）。在这个版本中，Sun 终结了从 JDK 1.2 开始已经有 8 年历史的 J2EE、J2SE、J2ME 的命名方式，启用了 Java SE 6、Java EE 6、Java ME 6 的命名来代替。JDK 1.6 的改进包括：提供动态语言支持（通过内置 Mozilla JavaScript Rhino 引擎实现）、提供编译 API 和微型 HTTP 服务器 API，等等。同时，这个版本对 Java 虚拟机的内部做了大量改进，包括锁与同步、垃圾收集、类加载等方面的算法都有相当多的改动。

在 2006 年 11 月 13 日的 JavaOne 大会上，Sun 宣布最终会把 Java 开源，并在随后的一年多时间内，陆续地在 GPL v2（GNU General Public License v2）协议下公开了 JDK 各个部分的源码，并建立了 OpenJDK 组织对这些源码进行独立管理。除了极少量的产权代码（Encumbered Code，这部分代码大多是 Sun 本身也无权限进行开源处理的）外，OpenJDK 几乎包括了 Sun JDK 的全部代码。OpenJDK 的质量主管曾经表示，在 JDK 1.7 中，Sun JDK 和 OpenJDK 除了代码文件头的版权注释之外，代码基本上完全一样，所以 OpenJDK 7 与 Sun JDK 1.7 本质上就是同一套代码库出来的产品。

JDK 1.6 发布以后，由于代码复杂性的增加、JDK 开源、开发 JavaFX、经济危机及 Sun 收购案等原因，Sun 在 JDK 发展以外的事情上耗费了很多资源，JDK 的更新没有再维持两年发布一个主版本的发展速度。JDK 1.6 到今天为止一共发布了 25 个 Update，最新的版本为 Java SE 6 Update 25，于 2011 年 4 月 21 日发布。

2009 年 2 月 19 日，工程代号为 Dolphin（海豚）的 JDK 1.7 完成了其第一个里程碑版本。根据 JDK 1.7 的功能规划，一共设置了 10 个里程碑。最后一个里程碑版本于 2010 年 9 月 9 日结束。从发布的 Early Access 版看来目前 JDK 1.7 的主体功能，已经比较完善，只剩下 Lambda 项目（Lambda 表达式）、Jigsaw（模块化支持）和 Coin（语言细节进化）子项目的部分工作尚未完成，Oracle 宣布 JDK 1.7 正式版将于 2011 年 7 月 28 日推出，可能会①把不能按时完成的 Lambda、Jigsaw 和部分 Coin 放入 JDK 1.8 之中。JDK 1.7 的主要改进包括：提供新的 G1 收集器、加强对非 Java 语言的调用、语言级的模块化支持（取决于 Jigsaw 项目能不能完成）、升级类加载架构，等等。

2009 年 4 月 20 日，Oracle 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（Java 语言本身并不属于哪家公司所有，它由 JCP 组织进行管理，尽管 JCP 主要是由 Sun 或者说 Oracle 所领导的）。由于此前 Oracle 已经收购了另外一家大型的中间件企业 BEA 公司，当完成对 Sun 公司的收购之后，Oracle 分别从 BEA 和 Sun 中取得了目前三大商业虚拟机的其中两个：JRockit 和 HotSpot，Oracle 宣布在未来 1 至 2 年的时间内，将把这两个优秀的虚拟机互相取长补短，最终合二为一①。可以预见在不久的将来，Java 技术体系将会产生相当巨大的变化。

1.4 展望 Java 技术的未来

在 Java 语言诞生 10 周年（2005 年）的 SunOne 技术大会上，Java 语言之父 James Gosling 做过一个题为“Java 技术的下一个十年”的演讲。笔者不具备 James Gosling 博士那样高屋建瓴的视角，这里仅从 Java 平台中几个新生的但已经开始展现出蓬勃之势的技术发展点来看一下后续 1 至 2 个 JDK 版本内的一些很有希望的技术重点

1.4.1 模块化

模块化是解决应用系统与技术平台越来越复杂、越来越庞大而产生的一系列问题的一个重要途径。无论是开发人员还是产品的最终用户，都不希望为了系统中的一小块功能而不得不下载、安装、部署及维护整套庞大的系统。最近几年 OSGi 技术的迅速发展正说明了通过模块化实现按需部署、降低复杂性和维护成本的需求是相当迫切的。

预计在未来的 Java 平台中，将会对模块化提供语法层面的支持。在 Java SE 7 发展初期，两个重要的 JSR 曾经试图解决依赖关系管理问题，分别是 JSR-294：Java 编程语言中的改进模块性支持（Improved Modularity Support in the Java Programming Language）和 JSR-277：Java 模块系统（Java Module System），两者分别关注 Java 模块概念的开发和部署方面。在具体实现方面，Java SE 7 中已建立了一个名为 Jigsaw（拼图）的项目来推动这两个规范在 Java 平台中转变为具体的实现。

1.4.2 混合语言

当单一的 Java 语言已经无法满足当前软件的复杂需求时，越来越多基于 Java 虚拟机的语言被应用到软件项目中。Java 平台上的多语言混合编程正成为主流，每种语言都可以针对自己擅长的方面更好地解决问题。试想一下，在一个项目之中，并行处理用 Clojure 语言编写，展示层使用 JRuby/Rails，中间层则是 Java，每个应用层都将使用不同的编程语言来完成，而且，接口对每一层的开发者都是透明的，各种语言之间的交互不存在任何困难，就像使用自己语言的原生 API 一样方便①，因为它们最终都运行在一个虚拟机之上。

在最近两年里，Clojure、JRuby、Groovy 等新生语言的使用人数如同滚动的雪球一般增长，而运行在 Java 虚拟机之上的语言数量也在迅速膨胀，图 1-4 中列举了其中的一部分。这两点证明混合编程在我们身边已经有所应用并被广泛认可。通过特定领域的语言去解决特定领域的问题是当前软件开发应对日趋复杂的项目需求的一个方向。

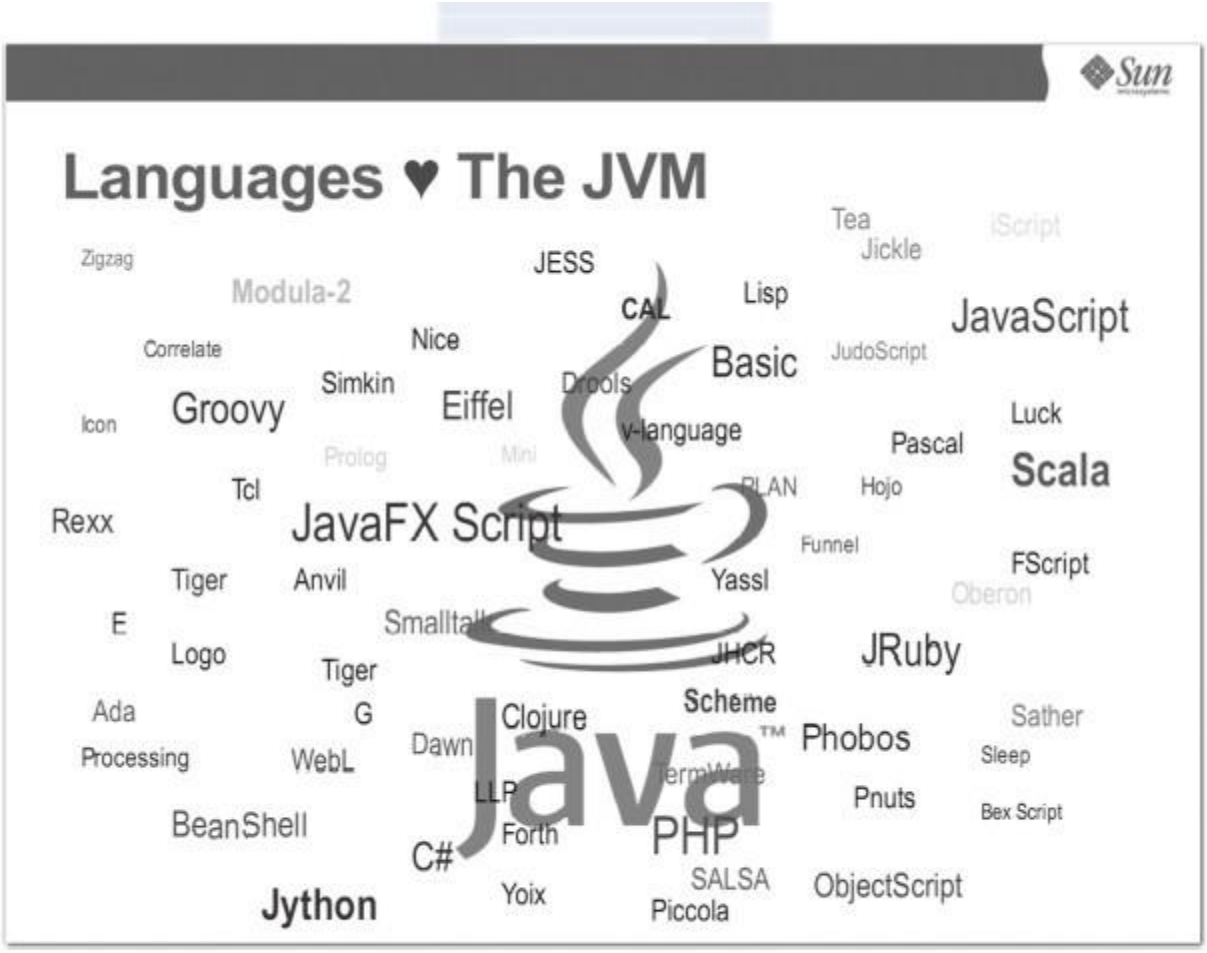

图 1-4　可以运行在 JVM 之上的语言[®]

除了催生出大量的新语言外，许多已经有很长历史的程序语言也出现了基于 Java 虚拟机实现的版本。这样的混合编程对许多以前使用其他语言的“老”程序员也有相当大的吸引力，软件企业投入了大量资本的现有代码资产也能被很好地保护起来。表 1-1 中列举了常见语言的 Java 虚拟机实现版本。

表 1-1　常见语言的 JVM 实现版本

语　言	基于 JVM 实现的版本
Ada	JGNAT
AWK	Jawk
C	C to Java Virtual Machine compilers
Cobol	Veryant is Cobol
ColdFusion	Adobe ColdFusion、Railo、Open BlueDragon
Common Lisp	Armed Bear Common Lisp、CLforJava、Jatha (Common LISP)
Component Pascal	Gardens Point Component Pascal
Erlang	Erjang
Forth	myForth
JavaScript	Rhino
LOGO	jLogo、XLogo
Lua	Kahlua、Luaj、Jill
Oberon-2	Canterbury Oberon-2 for JVM
Objective Caml (OCaml)	OCaml-Java
Pascal	Canterbury Pascal for JVM
PHP	IBM WebSphere sMash PHP (P8)、Caucho Quercus
Python	Jython
Rexx	IBM NetRexx
Ruby	JRuby
Scheme	Bigloo、Kawa、SISC、JScheme

1.4.3　多核并行

如今，CPU 硬件的发展方向已经从高频率转变为多核心，随着多核时代的来临，软件开发越来越关注并行编程的领域。早在 JDK 1.5 之中就已经引入 java.util.concurrent 包实现了一个粗粒度的并发框架，而 JDK 1.7 中将会加入的 java.util.concurrent.forkjoin 包则是对这个框架的一次重要扩充。Fork/Join 模式是处理并行编程的一种经典方法，如图 1-5 所示。虽然不能解决所有的问题，但是在它的适用范围之内，能够轻松地利用多个 CPU 核心提供的计算资源来协作完成一个复杂的计算任务。通过利用 Fork/Join 模式，我们能够更加顺畅地过渡到多核的时代。

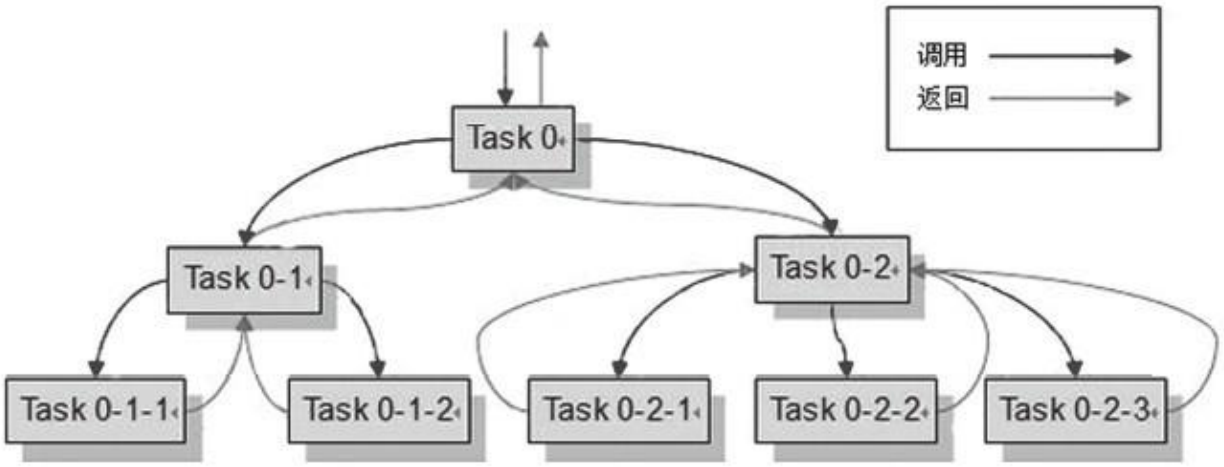

图 1-5　Fork/Join 模式示意图①

在 JDK 外围，也出现了专为满足并行计算需求的计算框架，如 Apache 的 Hadoop Map/Reduce，这是一个简单易懂的并行框架，能够运行在由上千个商用机器组成的大型集群上，并能以一种可靠的容错方式并行处理上 TB 级别的数据集。另外，还出现了诸如 Scala、Clojure 及 Erlang 等天生就具备并行计算能力的语言。

1.4.4　进一步丰富语法

JDK 1.5 曾经对 Java 语法进行了一次扩充，这次扩充加入了自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环等语法特性，使得 Java 语言的精确性和易用性有了很大的进步。在 JDK 1.7（由于进度压力，许多改进已被推迟至 JDK 1.8）中，将会对 Java 语法进行另一次大规模的扩充。Sun（Oracle）为此发起了 Coin 子项目②来统一处理对 Java 语法的细节修改，如二进制数的原生支持、在 switch 语句中支持字符串、“<>”操作符、异常处理的改进、简化变长参数方法调用、面向资源的 try-catch-finally 语句等都是在 Coin 项目之中提交的内容。另外，JSR-335（Lambda Expressions for the Java TM Programming Language）中定义的 Lambda 表达式③也将对 Java 的语法和语言习惯产生很大的影响，函数式编程可能会成为主流。

1.4.5　64 位虚拟机

几年之前，主流的 CPU 就开始支持 64 位架构。Java 虚拟机也在很早之前就推出了支持 64 位系统的版本。但 Java 程序运行在 64 位虚拟机上需要付出比较大的额外代价：首先是内存问题，由

于指针膨胀和各种数据类型对齐补白的原因，运行于 64 位系统上的 Java 应用需要消耗更多的内存，通常要比 32 位系统额外增加 10%～30%的内存消耗；其次是多个机构的测试结果显示，64 位虚拟机的运行速度在各个测试项上几乎都全面落后于 32 位虚拟机，两者大约有 15%左右的性能差距。

但是在 Java EE 方面，企业级应用经常需要使用超过 4G 的内存，对于 64 位虚拟机的需求是非常迫切的，由于上述的原因，许多企业应用都仍然选择使用虚拟集群等方式继续在 32 位虚拟机中进行部署。Sun 也注意到了这些问题，并做出了一些改善，在 JDK 1.6 Update 14 之后，提供了普通对象指针压缩功能（-XX:+ UseCompressedOops），在解释器解释字节码时，植入压缩指令以节省内存消耗。随着硬件的进一步发展，计算机终究会完全过渡到 64 位的时代，这是一件毫无疑问的事情，主流的虚拟机应用终究也会从 32 位发展至 64 位，而虚拟机对 64 位的支持也将会进一步完善。

1.5 实战：自己编译 JDK

想要一探 JDK 内部的实现机制，最便捷的路径之一就是自己编译一套 JDK。通过阅读和跟踪调试 JDK 源码去了解 Java 技术体系的原理，虽然门槛会高一点，但肯定会比阅读各种文章、书籍更加容易贴近本质。另外，JDK 中的很多底层方法都是 Native 的，当需要跟踪这些方法的运作或对 JDK 进行 Hack 的时候，都需要编译一套自己的 JDK。

现在网络上有不少开源的 JDK 实现可供选择，如 Apache Harmony、OpenJDK 等。考虑到 Sun 系列的 JDK 是现在使用得最广泛的 JDK 版本，本书选择了 OpenJDK 进行这次编译实战。

1.5.1 获取 JDK 源码

首先确定要使用的 JDK 版本，OpenJDK 6 和 OpenJDK 7 都是开源的，源码都可以在它们的主页（http://openjdk.java.net/）上找到，OpenJDK 6 的源码其实是从 OpenJDK 7 的某个基线中引出的，然后剥离掉 JDK 1.7 相关的代码，从而得到一份可以通过 TCK 6 的 JDK 1.6 实现，因此直接编译 OpenJDK 7 会更加“原汁原味”一些，其实这两个版本的编译过程差异并不大。

获取源码有两种方式：一种是通过 Mercurial 代码版本管理工具从 Repository 中直接取得源码（Repository 地址：http://hg.openjdk.java.net/jdk7/jdk7），这是最直接的方式，从版本管理中看变更轨迹比看任何 Release Note 都来得实在，不过坏处自然是太麻烦了一些，尤其是 Mercurial 远不如 SVN、ClearCase 或 CVS 之类的版本控制工具那样普及；另外一种就是直接下载官方打包好的源码包了，可以从 Source Releases 页面（地址：http://download.java.net/openjdk/jdk7/）取得打包好的源码，一般来说大概一个月左右会更新一次，虽然不够及时，但的确方便了许多。笔者下载的是 OpenJDK 7 Early Access Source Build b121 版，2010 年 12 月 9 日发布的，大概 81.7MB，解压后约 308MB。

1.5.2 系统需求

如果可能，笔者建议尽量在 Linux 或 Solaris 上构建 OpenJDK，这要比在 Windows 平台上轻松许多，而且网上能找到的资料绝大部分都是在 Linux 上编译的。如果一定要在 Windows 平台上编译，建议认真阅读一下源码中的 README-builds.html 文档（无论是在 OpenJDK 网站上，还是在下载的源码包里面都有这份文档），因为编译过程中需要注意的细节非常多。虽然不至于像文档上所描述的①那么夸张，但是如果大家是第一次编译，在上面耗费一整天乃至更多的时间都很正常。

本书在本次实战中演示的是在 32 位 Windows 7 平台下编译 x86 版的 OpenJDK（也就是 32 位的 JDK），如果需要编译 x64 版，那毫无疑问也需要一个 64 位的操作系统。另外，编译涉及的所有文件都必须存放在 NTFS 格式的文件系统中，因为 FAT32 格式无法支持大小写敏感的文件名。在官方文档上写道：编译至少需要 512MB 的内存和 600MB 的磁盘空间。如果大家耐心很好的话，512MB 的内存基本上也可以凑合使用，不过 600MB 的磁盘空间仅仅是指存放 OpenJDK 源码和相关依赖项的空间，要完成编译，600MB 肯定是无论如何都不够的，这次实战中所下载的工具、依赖项、源码，全部安装和解压完成最少（最少是指只下载 C++编译器，不下载 VS 的 IDE）需要超过 1GB 的空间。对系统的最后一点要求就是所有的文件，包括源码和依赖项目，都不要放在包含中文或空格的目录里面，这样做不是一定不可以，只是这样会为后续建立 CYGWIN 环境带来很多额外的工作，这是由于 Linux 和 Windows 的磁盘路径存在差别，我们也没有必要自己给自己找麻烦。

1.5.3 构建编译环境

准备编译环境的第一步是安装一个 CYGWIN①。这是一个在 Windows 平台下模拟 Linux 运行环境的软件，提供了一系列的 Linux 命令支持。需要 CYGWIN 的原因是因为在编译中要使用 GNU Make 来执行 Makefile 文件（C/C++程序员肯定很熟悉，如果只使用 Java，把它当做 C++版本的 ANT 看待就可以了）。安装 CYGWIN 时不能直接采用默认安装方式，因为表 1-2 中所示的工具在默认情况下都不会安装，但又是编译过程中需要的，因此要在图 1-6 的安装界面中进行手工选择。

表 1-2　需要手工选择安装的 CYGWIN 工具

文件名	分类	包	描　述
ar.exe	Devel	binutils	The GNU assembler, linker and binary utilities
make.exe	Devel	make	The GNU version of the 'make' utility built for CYGWIN.
m4.exe	Interpreters	m4	GNU implementation of the traditional Unix macro processor
cpio.exe	Utils	cpio	A program to manage archives of files
gawk.exe	Utils	awk	Pattern-directed scanning and processing language
file.exe	Utils	file	Determines file type using 'magic' numbers
zip.exe	Archive	zip	Package and compress (archive) files
unzip.exe	Archive	unzip	Extract compressed files in a ZIP archive
free.exe	System	procps	Display amount of free and used memory in the system

CYGWIN 安装时的定制包选择界面如图 1-6 所示。

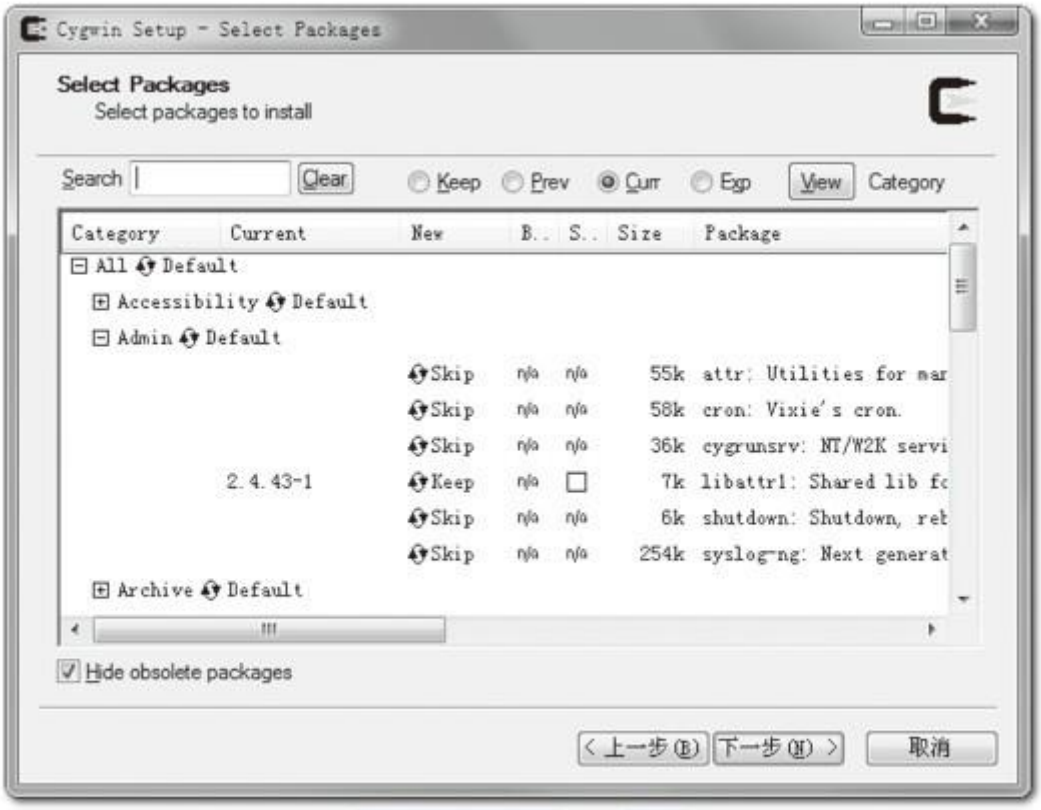

图 1-6　CYGWIN 安装界面

建立编译环境的第二步是安装编译器。JDK 中最核心的代码（Java 虚拟机及 JDK 中 Native 方法的实现等）是使用 C++语言及少量的 C 语言编写的，官方文档中说它们的内部开发环境是在 Microsoft Visual Studio C++ 2003（VS2003）中进行编译的，同时也在 Microsoft Visual Studio C++ 2010（VS2010）中测试过，所以最好只选择这两个编译器中的一个进行编译。如果选择 VS2010，那么在编译器之中已经包含了 Windows SDK v 7.0a，否则可能还要自己去下载这个 SDK，并且更新

PlatformSDK 目录。由于笔者没有购买 Visual Studio 2010 的 IDE，所以仅仅下载了 VS2010 Express 中提取出来的 C++编译器，这部分是免费的，但单独安装好编译器比较麻烦，建议读者选择使用整套 Visual Studio C++ 2010 或 Visual Studio C++ 2010 Express 版进行编译。

注意　CYGWIN 和 VS2010 安装之后都会在操作系统的 PATH 环境变量中写入自己的 bin 目录路径，必须检查并保证 VS2010 的 bin 目录一定要在 CYGWIN 的 bin 目录之前，因为这两个软件的 bin 目录之中各自都有个连接器“link.exe”，但是只有 VS2010 中的连接器可以完成 OpenJDK 的编译。

准备 JDK 编译环境的第三步是下载一个已经编译好的 JDK。这听起来也许有点滑稽—要用鸡蛋孵小鸡还真得先养一只母鸡呀？但仔细想想其实这个步骤很合理：因为 JDK 包含的各个部分（HotSpot、JDK API、JAXWS、JAXP……）有的是使用 C++编写的，而更多的代码则是使用 Java 自身实现的，因此编译这些 Java 代码需要用到一个可用的 JDK，官方称这个 JDK 为“Bootstrap JDK”。如果编译 OpenJDK 7，Bootstrap JDK 必须使用 JDK6 Update 14 或之后的版本，笔者选用的是 JDK6 Update 21。

最后一个步骤是下载一个 Apache ANT，JDK 中的 Java 代码部分都是使用 ANT 脚本进行编译的，ANT 版本要求在 1.6.5 以上，这部分是 Java 的基础知识，对本书的读者来说应该没有难度，笔者就不再详述。

1.5.4　准备依赖项

前面说过，OpenJDK 中开放的源码并没有达到 100%，还有极少量的无法开源的产权代码存在。OpenJDK 承诺日后将逐步使用开源实现来替换掉这部分产权代码，但至少在今天，编译 JDK 还需要这部分闭源包，官方称之为“JDK Plug”①，它们从前面的 Source Releases 页面就可以下载到。在 Windows 平台下的 JDK Plug 是以 Jar 包的形式提供的，通过下面这条命令可以安装它：

```
java -jar jdk-7-ea-plug-b121-windows-i586-09_dec_2010.jar
```

运行后将会显示如图 1-7 所示的协议，点击 ACCEPT 接受协议，然后把 Plug 安装到指定目录即可。安装完毕后建立一个环境变量“ALT_BINARY_PLUGS_PATH”，变量值为此 JDK Plug 的安装路径，后面编译程序时需要用到它。

除了要用到 JDK Plug 外，编译时还需要引用 JDK 的运行时包，它是编译 JDK 中用 Java 代码编写的那部分所需要的，如果仅仅是想编译一个 HotSpot 虚拟机，则可以不用。官方文档把这部分称之为“Optional Import JDK”，可以直接使用前面的 Bootstrap JDK 的运行时包，我们需要建立一个名为“ALT_JDK_IMPORT_PATH”的环境变量指向 JDK 的安装目录。

第三步是安装一个大于 2.3 版的 FreeType②，这是一个免费的字体渲染库，JDK 的 Swing 部分和 JConsole 这类工具要使用到它。安装好后建立两个环境变量“ALT_FREETYPE_LIB_PATH”和“ALT_FREETYPE_HEADERS_PATH”，分别指向 FreeType 安装目录下的 bin 目录和 include 目录。

另外，还有一点官方文档没有提到但必须要做的事情是把 FreeType 的 bin 目录加入到 PATH 环境变量中。

图 1-7　JDK Plug 安装协议

第四步是下载 Microsoft DirectX 9.0 SDK（Summer 2004），安装后大约有 298MB，在微软官方网站上搜索一下就可以找到下载地址，它是免费的。安装后建立环境变量“ALT_DXSDK_PATH”指向 DirectX 9.0 SDK 的安装目录。

第五步是去寻找一个名为“MSVCR100.DLL”的动态链接库，如果读者在前面安装了全套的 Visual Studio 2010，那这个文件在本机就能找到，否则上网搜索一下也能找到单独的下载地址，大概有 744KB。建立环境变量“ALT_MSVCRNN_DLL_PATH”指向这个文件所在的目录。如果读者选择的是 VS2003，这个文件名应当为“MSVCR73.DLL”。很多软件中都包含有这个文件，如果找不到，前面下载的“Bootstrap JDK”的 bin 目录中应该也有一个，直接拿来用吧。

1.5.5　进行编译

现在，需要下载的编译环境和依赖项目都准备齐全了，最后我们还需要对系统进行一些设置以便编译能够顺利通过。

首先执行 VS2010 中的 VCVARS32.BAT，这个批处理文件的主要目的是设置 INCLUDE、LIB 和 PATH 这几个环境变量。如果大家和笔者一样只是下载了编译器，则需要手工设置它们，各个环境变量的设

置值可以参考代码清单 1-1 中的内容。批处理运行完之后建立“ALT_COMPILER_PATH”环境变量，让 Makefile 知道在哪里可以找到编译器。

再建立“ALT_BOOTDIR”和“ALT_JDK_IMPORT_PATH”两个环境变量，指向前面提到的 JDK 1.6 的安装目录，建立“ANT_HOME”指向 Apache ANT 的安装目录。建立的环境变量很多，为了避免遗漏，笔者写了一个批处理文件以供读者参考，如代码清单 1-1 所示。

代码清单 1-1　环境变量设置

```
SET ALT_BOOTDIR=D:/_DevSpace/JDK 1.6.0_21
SET ALT_BINARY_PLUGS_PATH=D:/jdkBuild/jdk7plug/openjdk-binary-plugs
SET ALT_JDK_IMPORT_PATH=D:/_DevSpace/JDK 1.6.0_21
SET ANT_HOME=D:/jdkBuild/apache-ant-1.7.0
SET ALT_MSVCRNN_DLL_PATH=D:/jdkBuild/msvcr100
SET ALT_DXSDK_PATH=D:/jdkBuild/msdxsdk
SET ALT_COMPILER_PATH=D:/jdkBuild/vcpp2010.x86/bin
SET ALT_FREETYPE_HEADERS_PATH=D:/jdkBuild/freetype-2.3.5-1-bin/include
SET ALT_FREETYPE_LIB_PATH=D:/jdkBuild/freetype-2.3.5-1-bin/bin

SET INCLUDE=D:/jdkBuild/vcpp2010.x86/include;D:/jdkBuild/vcpp2010.x86/sdk/
Include;%INCLUDE%
SET LIB=D:/jdkBuild/vcpp2010.x86/lib;D:/jdkBuild/vcpp2010.x86/sdk/Lib;%LIB%
SET LIBPATH=D:/jdkBuild/vcpp2010.x86/lib;%LIB%
SET PATH=D:/jdkBuild/vcpp2010.x86/bin;D:/jdkBuild/vcpp2010.x86/dll/x86;D:/
Software/OpenSource/cygwin/bin;%ALT_FREETYPE_LIB_PATH%;%PATH%
```

最后还需要再做两项调整，官方文档没有说明这两项，但是必须要做完才能保证编译过程顺利完成：一项是取消环境变量 JAVA_HOME，这点很简单；另外一项是尽量在英文的操作系统上编译，估计大部分读者会感到比较为难吧。如果不能在英文的系统上编译，就把系统的文字格式调整为“英语（美国）”，在控制面板中的“区域和语言”选项的第一个页签中可以设置。如果这个设置还不能更改就建立一个“BUILD_CORBA”的环境变量，将值设置为 false，取消编译 CORBA 部分。否则 Java IDL（idlj.exe）为*.idl 文件生成 CORBA 适配器代码的时候会产生中文注释，而这些中文注释会因为字符集的问题而导致编译失败。

完成了上述繁琐的准备工作之后，我们终于可以开始编译了。进入控制台（Cmd.exe）后运行刚才准备好的设置环境变量的批处理文件，然后输入 bash 进入 Bourne Again Shell 环境（习惯 sh 或 ksh 的读者请自便）。如果 JDK 的安装源码中存在“jdk_generic_profile.sh”这个 Shell 脚本，先执行它，

笔者下载的 OpenJDK 7 B121 版没有这个文件了，所以可直接输入 make sanity 来检查我们前面所做的设置是否全部正确。如果一切顺利，几秒钟之后会有类似代码清单 1-2 所示的输出。

代码清单 1-2　make sanity 检查

```
D:\jdkBuild\openjdk7>bash
bash-3.2$ make sanity
cygwin warning:
  MS-DOS style path detected: C:/Windows/system32/wscript.exe
  Preferred POSIX equivalent is: /cygdrive/c/Windows/system32/wscript.exe
  CYGWIN environment variable option "nodosfilewarning" turns off this warning.
  Consult the user's guide for more details about POSIX paths:
    http://cygwin.com/cygwin-ug-net/using.html#using-pathnames
( cd  ./jdk/make && \

……

OpenJDK-specific settings:
  FREETYPE_HEADERS_PATH = D:/jdkBuild/freetype-2.3.5-1-bin/include
    ALT_FREETYPE_HEADERS_PATH = D:/jdkBuild/freetype-2.3.5-1-bin/include
  FREETYPE_LIB_PATH = D:/jdkBuild/freetype-2.3.5-1-bin/bin
    ALT_FREETYPE_LIB_PATH = D:/jdkBuild/freetype-2.3.5-1-bin/bin

OPENJDK Import Binary Plug Settings:
  IMPORT_BINARY_PLUGS = true
  BINARY_PLUGS_JARFILE = D:/jdkBuild/jdk7plug/openjdk-binary-plugs/jre/lib/rt-
closed.jar
    ALT_BINARY_PLUGS_JARFILE =
  BINARY_PLUGS_PATH = D:/jdkBuild/jdk7plug/openjdk-binary-plugs
    ALT_BINARY_PLUGS_PATH = D:/jdkBuild/jdk7plug/openjdk-binary-plugs
  BUILD_BINARY_PLUGS_PATH = J:/re/jdk/1.7.0/promoted/latest/openjdk/binaryplugs
    ALT_BUILD_BINARY_PLUGS_PATH =
  PLUG_LIBRARY_NAMES =
```

Previous JDK Settings:

PREVIOUS_RELEASE_PATH = USING-PREVIOUS_RELEASE_IMAGE

ALT_PREVIOUS_RELEASE_PATH =

PREVIOUS_JDK_VERSION = 1.6.0

ALT_PREVIOUS_JDK_VERSION =

PREVIOUS_JDK_FILE =

ALT_PREVIOUS_JDK_FILE =

PREVIOUS_JRE_FILE =

ALT_PREVIOUS_JRE_FILE =

PREVIOUS_RELEASE_IMAGE = D:/_DevSpace/JDK 1.6.0_21

ALT_PREVIOUS_RELEASE_IMAGE =

Sanity check passed.

Makefile 的 Sanity 检查过程输出了编译所需的所有环境变量，如果看到“Sanity check passed.”说明检查过程通过了，可以输入“make”执行整个 Makefile，然后去喝杯下午茶再回来。笔者的 Core i5 / 4GB RAM 的机器编译整个 JDK 大概需要半个多小时。如果失败，则需要根据系统输出的失败原因，回头再检查一下对应的设置，并且最好在下一次编译之前先执行“make clean”来清理掉上次编译遗留的文件。

编译完成之后，打开 OpenJDK 源码下的 build 目录，看看是不是已经有一个编译好的 JDK 在那里等着了？执行一下“java -version”，看到以自己的机器命名的 JDK 了吧，很有成就感吧？

1.6 本章小结

本章介绍了 Java 技术体系的过去、现在和未来的发展趋势，并通过实践的方式介绍了如何自己来独立编译一个 OpenJDK 7。作为全书的引言部分，本章建立了后文研究所必需的环境。在了解 Java 技术的来龙去脉后，后面的章节将分为四部分去讲解 Java 在内存管理、Class 文件结构与执行引擎、编译器优化和多线程并发方面的实现原理。

第 2 章 Java 内存区域与内存溢出异常

本章主要内容

概述

运行时数据区域

对象访问

实战：OutOfMemoryError 异常

Java 与 C++之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

2.1 概述

对于从事 C 和 C++程序开发的开发人员来说，在内存管理领域，他们既是拥有最高权力的皇帝，又是从事最基础工作的劳动人民—既拥有每一个对象的“所有权”，又担负着每一个对象生命开始到终结的维护责任。

对于 Java 程序员来说，在虚拟机的自动内存管理机制的帮助下，不再需要为每一个 new 操作去写配对的 delete/free 代码，而且不容易出现内存泄漏和内存溢出问题，看起来由虚拟机管理内存一切都很美好。不过，也正是因为 Java 程序员把内存控制的权力交给了 Java 虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那排查错误将会成为一项异常艰难的工作。

本章是第二部分的第 1 章，笔者将从概念上介绍 Java 虚拟机内存的各个区域，讲解这些区域的作用、服务对象以及其中可能产生的问题，这是翻越虚拟机内存管理这堵围墙的第一步。

2.2 运行时数据区域

Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有些区域则是依赖用户线程的启动和结束而建立和销毁。根据《Java 虚拟机规范（第 2 版）》的规定，Java 虚拟机所管理的内存将会包括以下几个运行时数据区域，如图 2-1 所示。

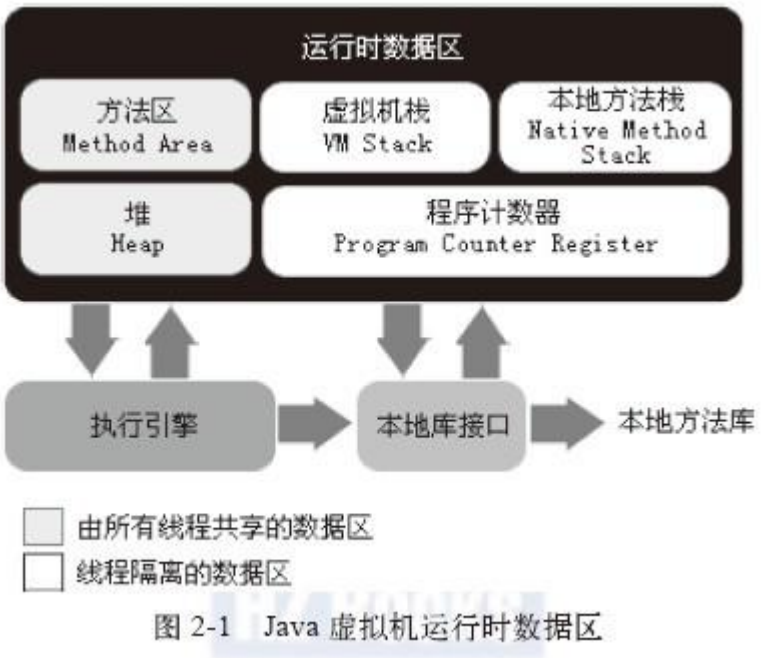

图 2-1　Java 虚拟机运行时数据区

2.2.1　程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Natvie 方法，这个计数器值则为空（Undefined）。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

2.2.2　Java 虚拟机栈

与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：每个方法被执行的时候都会同时创

建一个栈帧（Stack Frame①）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

经常有人把 Java 内存区分为堆内存（Heap）和栈内存（Stack），这种分法比较粗糙，Java 内存区域的划分实际上远比这复杂。这种划分方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。其中所指的“堆”在后面会专门讲述，而所指的“栈”就是现在讲的虚拟机栈，或者说是虚拟机栈中的局部变量表部分。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置）和 returnAddress 类型（指向了一条字节码指令的地址）。

其中 64 位长度的 long 和 double 类型的数据会占用 2 个局部变量空间（Slot），其余的数据类型只占用 1 个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在 Java 虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常；如果虚拟机栈可以动态扩展（当前大部分的 Java 虚拟机都可动态扩展，只不过 Java 虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存时会抛出 OutOfMemoryError 异常。

2.2.3 本地方法栈

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的 Native 方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如 Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

2.2.4 Java 堆

对于大多数应用来说，Java 堆（Java Heap）是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。这一点在 Java 虚拟机规范中的描述是：所有的对象实

例以及数组都要在堆上分配①，但是随着 JIT 编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换②优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“绝对”了。

Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC 堆”（Garbage Collected Heap，幸好国内没翻译成“垃圾堆”）。如果从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以 Java 堆中还可以细分为：新生代和老年代；再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空间等。如果从内存分配的角度看，线程共享的 Java 堆中可能划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer，TLAB）。不过，无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好地回收内存，或者更快地分配内存。在本章中，我们仅仅针对内存区域的作用进行讨论，Java 堆中的上述各个区域的分配和回收等细节将会是下一章的主题。

根据 Java 虚拟机规范的规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx 和-Xms 控制）。如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 OutOfMemoryError 异常。

2.2.5 方法区

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

对于习惯在 HotSpot 虚拟机上开发和部署程序的开发者来说，很多人愿意把方法区称为“永久代”（Permanent Generation），本质上两者并不等价，仅仅是因为 HotSpot 虚拟机的设计团队选择把 GC 分代收集扩展至方法区，或者说使用永久代来实现方法区而已。对于其他虚拟机（如 BEA JRockit、IBM J9 等）来说是不存在永久代的概念的。即使是 HotSpot 虚拟机本身，根据官方发布的路线图信息，现在也有放弃永久代并“搬家”至 Native Memory 来实现方法区的规划了。

Java 虚拟机规范对这个区域的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载，一般来说这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。在 Sun 公司的 BUG 列表中，

曾出现过的若干个严重的 BUG 就是由于低版本的 HotSpot 虚拟机对此区域未完全回收而导致内存泄漏。

根据 Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。

2.2.6 运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

Java 虚拟机对 Class 文件的每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会被虚拟机认可、装载和执行。但对于运行时常量池，Java 虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存 Class 文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中①。

运行时常量池相对于 Class 文件常量池的另外一个重要特征是具备动态性，Java 语言并不要求常量一定只能在编译期产生，也就是并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是 String 类的 intern()方法。

既然运行时常量池是方法区的一部分，自然会受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

2.2.7 直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致 OutOfMemoryError 异常出现，所以我们放到这里一起讲解。

在 JDK 1.4 中新加入了 NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆里面的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

显然，本机直接内存的分配不会受到 Java 堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存（包括 RAM 及 SWAP 区或者分页文件）的大小及处理器寻址空间的限制。服务器管理

员配置虚拟机参数时，一般会根据实际内存设置-Xmx 等参数信息，但经常会忽略掉直接内存，使得各个内存区域的总和大于物理内存限制（包括物理上的和操作系统级的限制），从而导致动态扩展时出现 OutOfMemoryError 异常。

2.3 对象访问

介绍完 Java 虚拟机的运行时数据区之后，我们就可以来探讨一个问题：在 Java 语言中，对象访问是如何进行的？对象访问在 Java 语言中无处不在，是最普通的程序行为，但即使是最简单的访问，也会却涉及 Java 栈、Java 堆、方法区这三个最重要内存区域之间的关联关系，如下面的这句代码：

```
Object obj = new Object();
```

假设这句代码出现在方法体中，那“Object obj”这部分的语义将会反映到 Java 栈的本地变量表中，作为一个 reference 类型数据出现。而“new Object()”这部分的语义将会反映到 Java 堆中，形成一块存储了 Object 类型所有实例数据值（Instance Data，对象中各个实例字段的数据）的结构化内存，根据具体类型以及虚拟机实现的对象内存布局（Object Memory Layout）的不同，这块内存的长度是不固定的。另外，在 Java 堆中还必须包含能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些类型数据则存储在方法区中。

由于 reference 类型在 Java 虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到 Java 堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄和直接指针。

如果使用句柄访问方式，Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据和类型数据各自的具体地址信息，如图 2-2 所示。

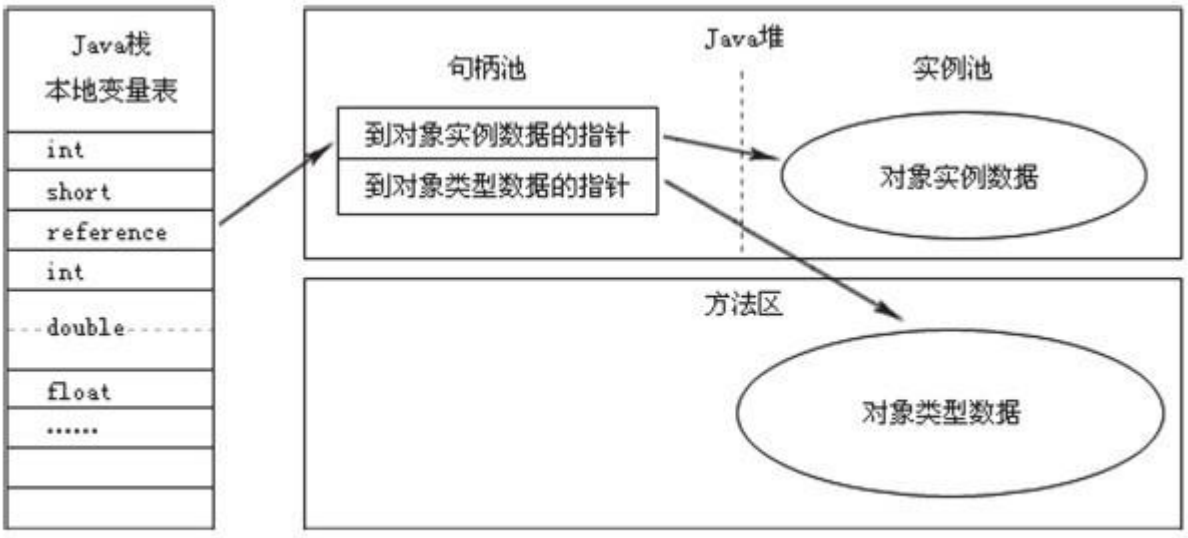

图 2-2　通过句柄访问对象

如果使用直接指针访问方式，Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference 中直接存储的就是对象地址，如图 2-3 所示。

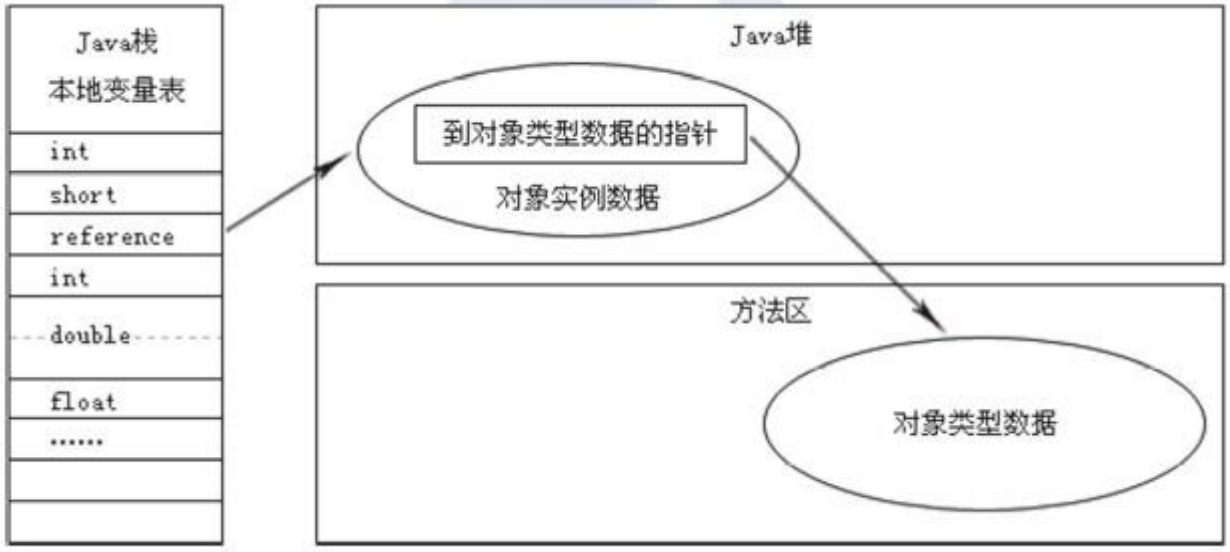

图 2-3　通过直接指针访问对象

这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是 reference 中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。就本书讨论的主要虚拟机 Sun HotSpot 而言，它是使用第二种方式进行对象访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。

2.4 实战：OutOfMemoryError 异常

在 Java 虚拟机规范的描述中，除了程序计数器外，虚拟机内存的其他几个运行时区域都有发生 OutOfMemoryError（下文称 OOM）异常的可能，本节将通过若干实例来验证异常发生的场景（代码清单 2-1 至代码清单 2-6 的几段简单代码），并且会初步介绍几个与内存相关的最基本的虚拟机参数。

本节内容的目的有两个：第一，通过代码验证 Java 虚拟机规范中描述的各个运行时区域储存的内容；第二，希望读者在工作中遇到实际的内存溢出异常时，能根据异常的信息快速判断是哪个区域的内存溢出，知道怎样的代码可能会导致这些区域的内存溢出，以及出现这些异常后该如何处理。

下面代码的开头都注释了执行时所需要设置的虚拟机启动参数（注释中“VM Args”后面跟着的参数），这些参数对实验的结果有直接影响，请读者调试代码的时候不要忽略掉。如果读者使用控制台命令来执行程序，那直接跟在 Java 命令之后书写就可以。如果读者使用 Eclipse IDE，则可以参考图 2-4 在 Debug/Run 页签中的设置。

下文的代码都是基于 Sun HotSpot 17.1-b03（JDK 1.6 Update 22 中带的虚拟机）运行的，对于不同公司的不同版本的虚拟机，参数和程序运行的结果可能会有所差别。

2.4.1 Java 堆溢出

Java 堆用于储存对象实例，我们只要不断地创建对象，并且保证 GC Roots 到对象之间有可达路径来避免垃圾回收机制清除这些对象，就会在对象数量到达最大堆的容量限制后产生内存溢出异常。

代码清单 2-1 中限制 Java 堆的大小为 20MB，不可扩展（将堆的最小值-Xms 参数与最大值-Xmx 参数设置为一样即可避免堆自动扩展），通过参数-XX:+HeapDump OnOutOfMemoryError 可以让虚拟机在出现内存溢出异常时 Dump 出当前的内存堆转储快照以便事后进行分析①。

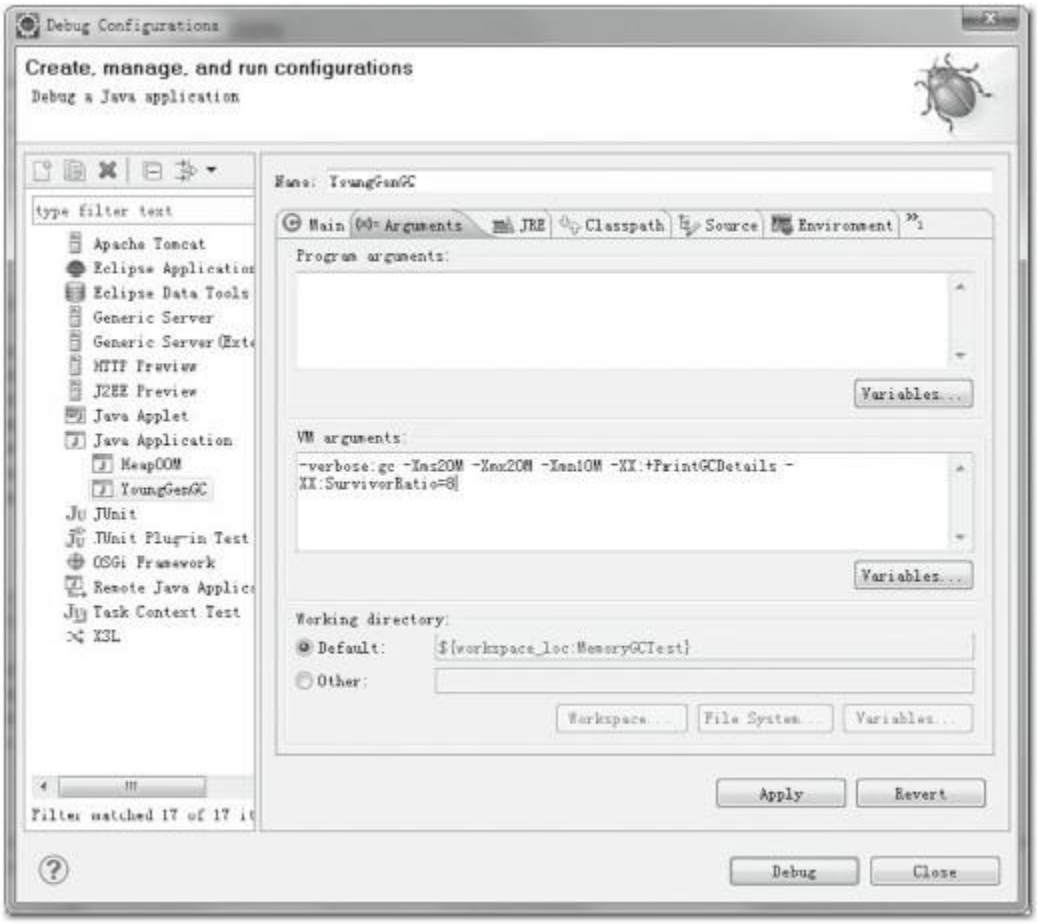

图 2-4　在 Eclipse 的 Debug 页签中设置虚拟机参数

代码清单 2-1　Java 堆内存溢出异常测试

```
/**
 * VM Args：-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
 * @author zzm
 */
public class HeapOOM {
        static class OOMObject {
        }
        public static void main(String[] args) {
         List<OOMObject> list = new ArrayList<OOMObject>();
         while (true) {
  list.add(new OOMObject());
         }
```

```
    }
}
```

运行结果：

```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid3404.hprof ...
Heap dump file created [22045981 bytes in 0.663 secs]
```

Java 堆内存的 OOM 异常是实际应用中最常见的内存溢出异常情况。出现 Java 堆内存溢出时，异常堆栈信息“java.lang.OutOfMemoryError”会跟着进一步提示“Java heap space”。

要解决这个区域的异常，一般的手段是首先通过内存映像分析工具（如 Eclipse Memory Analyzer）对 dump 出来的堆转储快照进行分析，重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏（Memory Leak）还是内存溢出（Memory Overflow）。图 2-5 显示了使用 Eclipse Memory Analyzer 打开的堆转储快照文件。

如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链。于是就能找到泄漏对象是通过怎样的路径与 GC Roots 相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息，以及 GC Roots 引用链的信息，就可以比较准确地定位出泄漏代码的位置。

如果不存在泄漏，换句话说就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数（-Xmx 与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

以上是处理 Java 堆内存问题的简略思路，处理这些问题所需要的知识、工具与经验是后面三章的主题。

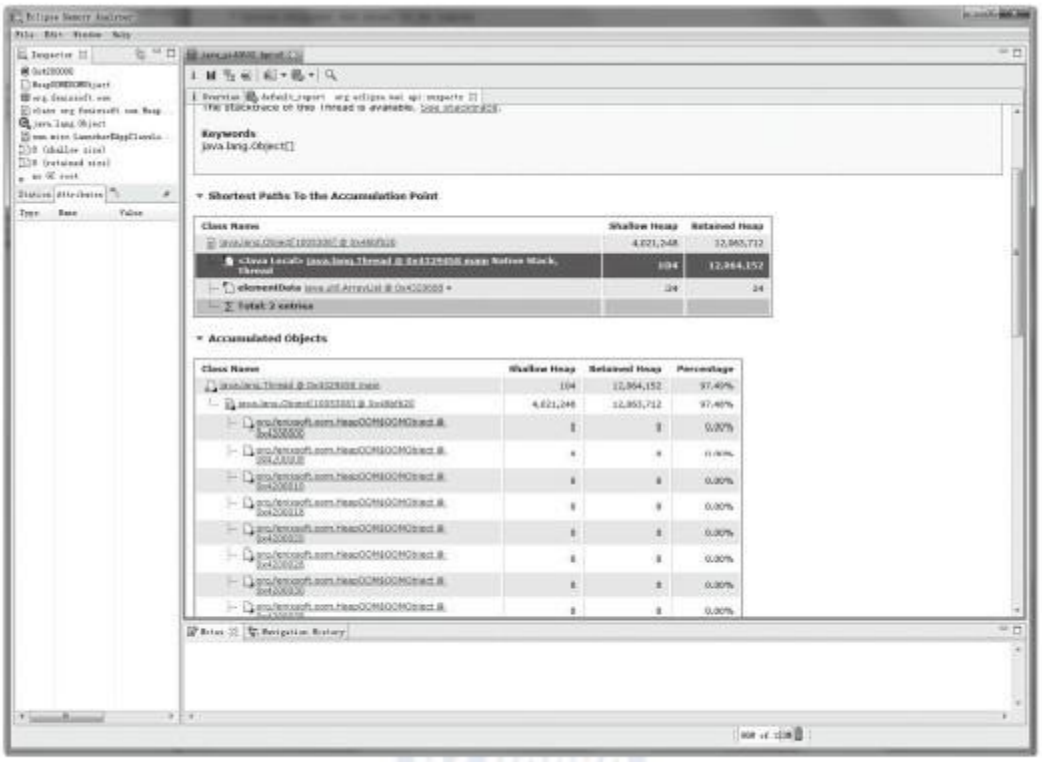

图 2-5　使用 Eclipse Memory Analyzer 打开的堆转储快照文件

2.4.2　虚拟机栈和本地方法栈溢出

由于在 HotSpot 虚拟机中并不区分虚拟机栈和本地方法栈，因此对于 HotSpot 来说，-Xoss 参数（设置本地方法栈大小）虽然存在，但实际上是无效的，栈容量只由-Xss 参数设定。关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常。

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常。

这里把异常分成两种情况看似更加严谨，但却存在着一些互相重叠的地方：当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质上只是对同一件事情的两种描述而已。

在笔者的实验中，如果将实验范围限制于单线程中的操作，尝试了下面两种方法均无法让虚拟机产生 OutOfMemoryError 异常，尝试的结果都是获得 StackOverflowError 异常，测试代码如清单 2-2 所示。

使用-Xss 参数减少栈内存容量。结果：抛出 StackOverflowError 异常，异常出现时输出的栈深度相应缩小。

定义了大量的本地变量，增加此方法帧中本地变量表的长度。结果：抛出 StackOverflowError 异常时输出的栈深度相应缩小。

代码清单 2-2　虚拟机栈和本地方法栈 OOM 测试（仅作为第 1 点测试程序）

```
/**
* VM Args：-Xss128k
* @author zzm
*/
public class JavaVMStackSOF {

    private int stackLength = 1;

    public void stackLeak() {
    stackLength++;
    stackLeak();
    }

    public static void main(String[] args) throws Throwable {
    JavaVMStackSOF oom = new JavaVMStackSOF();
    try {
    oom.stackLeak();
    } catch (Throwable e) {
    System.out.println("stack length:" + oom.stackLength);
    throw e;
    }
    }
}
```

运行结果：

```
stack length:2402
Exception in thread "main" java.lang.StackOverflowError
        at org.fenixsoft.oom.VMStackSOF.leak(VMStackSOF.java:20)
        at org.fenixsoft.oom.VMStackSOF.leak(VMStackSOF.java:21)
        at org.fenixsoft.oom.VMStackSOF.leak(VMStackSOF.java:21)
……后续异常栈信息省略
```

实验结果表明：在单个线程下，无论是由于栈帧太大，还是虚拟机栈容量太小，当内存无法分配

的时候，虚拟机抛出的都是 StackOverflowError 异常。

如果测试时不限于单线程，通过不断地建立线程的方式倒是可以产生内存溢出异常，如代码清单 2-3 所示。但是，这样产生的内存溢出异常与栈空间是否足够大并不存在任何联系，或者准确地说，在这种情况下，给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。

原因其实不难理解，操作系统分配给每个进程的内存是有限制的，譬如 32 位的 Windows 限制为 2GB。虚拟机提供了参数来控制 Java 堆和方法区的这两部分内存的最大值。剩余的内存为 2GB（操作系统限制）减去 Xmx（最大堆容量），再减去 MaxPermSize（最大方法区容量），程序计数器消耗内存很小，可以忽略掉。如果虚拟机进程本身耗费的内存不计算在内，剩下的内存就由虚拟机栈和本地方法栈“瓜分”了。每个线程分配到的栈容量越大，可以建立的线程数量自然就越少，建立线程时就越容易把剩下的内存耗尽。

这一点读者需要在开发多线程应用的时候特别注意，出现 StackOverflowError 异常时有错误堆栈可以阅读，相对来说，比较容易找到问题的所在。而且，如果使用虚拟机默认参数，栈深度在大多数情况下（因为每个方法压入栈的帧大小并不是一样的，所以只能说大多数情况下）达到 1000～2000 完全没有问题，对于正常的方法调用（包括递归），这个深度应该完全够用了。但是，如果是建立过多线程导致的内存溢出，在不能减少线程数或者更换 64 位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。如果没有这方面的经验，这种通过“减少内存”的手段来解决内存溢出的方式会比较难以想到。

代码清单 2-3　创建线程导致内存溢出异常

```
/**
* VM Args：-Xss128k
* @author zzm
*/
public class JavaVMStackSOF {

    private int stackLength = 1;

    public void stackLeak() {
    stackLength++;
    stackLeak();
    }

    public static void main(String[] args) throws Throwable {
```

```
        JavaVMStackSOF oom = new JavaVMStackSOF();
        try {
        oom.stackLeak();
        } catch (Throwable e) {
        System.out.println("stack length:" + oom.stackLength);
        throw e;
        }
        }
}
```

注意　特别提示一下，如果读者要尝试运行上面这段代码，记得要先保存当前的工作，由于在 Windows 平台的虚拟机中，Java 的线程是映射到操作系统的内核线程上的①，所以上述代码执行时有较大的风险，可能会导致操作系统假死。

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread
```

2.4.3　运行时常量池溢出

如果要向运行时常量池中添加内容，最简单的做法就是使用 String.intern()这个 Native 方法。该方法的作用是：如果池中已经包含一个等于此 String 对象的字符串，则返回代表池中这个字符串的 String 对象；否则，将此 String 对象包含的字符串添加到常量池中，并且返回此 String 对象的引用。由于常量池分配在方法区内，我们可以通过-XX:PermSize 和-XX:MaxPermSize 限制方法区的大小，从而间接限制其中常量池的容量，如代码清单 2-4 所示。

代码清单 2-4　运行时常量池导致的内存溢出异常

```
/**
* VM Args：-XX:PermSize=10M -XX:MaxPermSize=10M
* @author zzm
*/
public class RuntimeConstantPoolOOM {

        public static void main(String[] args) {
        // 使用 List 保持着常量池引用，避免 Full GC 回收常量池行为
        List<String> list = new ArrayList<String>();
```

```
    // 10MB 的 PermSize 在 integer 范围内足够产生 OOM 了
    int i = 0;
    while (true) {
    list.add(String.valueOf(i++).intern());
    }
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
at java.lang.String.intern(Native Method)
at org.fenixsoft.oom.RuntimeConstantPoolOOM.main(RuntimeConstantPoolOOM.java:18)
```

从运行结果中可以看到，运行时常量池溢出，在 OutOfMemoryError 后面跟随的提示信息是“PermGen space”，说明运行时常量池属于方法区（HotSpot 虚拟机中的永久代）的一部分。

2.4.4 方法区溢出

方法区用于存放 Class 的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。对于这个区域的测试，基本的思路是运行时产生大量的类去填满方法区，直到溢出。虽然直接使用 Java SE API 也可以动态产生类（如反射时的 GeneratedConstructorAccessor 和动态代理等），但在本次实验中操作起来比较麻烦。在代码清单 2-5 中，笔者借助 CGLib①直接操作字节码运行时，生成了大量的动态类。

值得特别注意的是，我们在这个例子中模拟的场景并非纯粹是一个实验，这样的应用经常会出现在实际应用中：当前的很多主流框架，如 Spring 和 Hibernate 对类进行增强时，都会使用到 CGLib 这类字节码技术，增强的类越多，就需要越大的方法区来保证动态生成的 Class 可以加载入内存。

代码清单 2-5　借助 CGLib 使得方法区出现内存溢出异常

```
/**
* VM Args：  -XX:PermSize=10M -XX:MaxPermSize=10M
* @author zzm
*/
public class JavaMethodAreaOOM {

    public static void main(String[] args) {
```

```
        while (true) {
    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(OOMObject.class);
    enhancer.setUseCache(false);
    enhancer.setCallback(new MethodInterceptor() {
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws
Throwable {
    return proxy.invokeSuper(obj, args);
    }
    });
    enhancer.create();
        }
    }

    static class OOMObject {


    }
}
```

运行结果：

```
Caused by: java.lang.OutOfMemoryError: PermGen space
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClassCond(ClassLoader.java:632)
at java.lang.ClassLoader.defineClass(ClassLoader.java:616)
... 8 more
```

方法区溢出也是一种常见的内存溢出异常，一个类如果要被垃圾收集器回收掉，判定条件是非常苛刻的。在经常动态生成大量 Class 的应用中，需要特别注意类的回收状况。这类场景除了上面提到的程序使用了 GCLib 字节码增强外，常见的还有：大量 JSP 或动态产生 JSP 文件的应用（JSP 第一次运行时需要编译为 Java 类）、基于 OSGi 的应用（即使是同一个类文件，被不同的加载器加载也会视为不同的类）等。

2.4.5 本机直接内存溢出

DirectMemory 容量可通过-XX:MaxDirectMemorySize 指定，如果不指定，则默认与 Java 堆的最大值（-Xmx 指定）一样。代码清单 2-6 越过了 DirectByteBuffer 类，直接通过反射获取 Unsafe 实例并进行内存分配（Unsafe 类的 getUnsafe()方法限制了只有引导类加载器才会返回实例，也就是设计者希望只有 rt.jar 中的类才能使用 Unsafe 的功能）。因为，虽然使用 DirectByteBuffer 分配内存也会抛出内存溢出异常，但它抛出异常时并没有真正向操作系统申请分配内存，而是通过计算得知内存无法分配，于是手动抛出异常，真正申请分配内存的方法是 unsafe.allocateMemory()。

代码清单 2-6　使用 unsafe 分配本机内存

```
/**
* VM Args：-Xmx20M -XX:MaxDirectMemorySize=10M
* @author zzm
*/
public class DirectMemoryOOM {

        private static final int _1MB = 1024 * 1024;

        public static void main(String[] args) throws Exception {
           Field unsafeField = Unsafe.class.getDeclaredFields()[0];
           unsafeField.setAccessible(true);
           Unsafe unsafe = (Unsafe) unsafeField.get(null);
           while (true) {
           unsafe.allocateMemory(_1MB);
           }
        }
}
```

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError
      at sun.misc.Unsafe.allocateMemory(Native Method)
      at org.fenixsoft.oom.DMOOM.main(DMOOM.java:20)
```

2.5　本章小结

通过本章的学习，我们明白了虚拟机里面的内存是如何划分的，哪部分区域、什么样的代码和操作可能导致内存溢出异常。虽然 Java 有垃圾收集机制，但内存溢出异常离我们并不遥远，本章只是讲解了各个区域出现内存溢出异常的原因，下一章将详细讲解 Java 垃圾收集机制为了避免内存溢出异常的出现都做了哪些努力。

第 3 章　垃圾收集器与内存分配策略

本章主要内容

概述

对象已死？

垃圾收集算法

垃圾收集器

内存分配与回收策略

Java 与 C++之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

3.1　概述

说起垃圾收集（Garbage Collection，GC），大部分人都把这项技术当做 Java 语言的伴生产物。事实上，GC 的历史远远比 Java 久远，1960 年诞生于 MIT 的 Lisp 是第一门真正使用内存动态分配和垃圾收集技术的语言。当 Lisp 还在胚胎时期时，人们就在思考 GC 需要完成的三件事情：

哪些内存需要回收？

什么时候回收？

如何回收？

经过半个世纪的发展，内存的动态分配与内存回收技术已经相当成熟，一切看起来都进入了“自动化”时代，那为什么我们还要去了解 GC 和内存分配呢？答案很简单：当需要排查各种内存溢出、内存泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些“自动化”的技术实施必要的监控和调节。

把时间从半个世纪以前拨回到现在，回到我们熟悉的 Java 语言。第 2 章介绍了 Java 内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈三个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。每一个栈帧中分配多少内存基本

上是在类结构确定下来时就已知的（尽管在运行期会由 JIT 编译器进行一些优化，但在本章基于概念模型的讨论中，大体上可以认为是编译期可知的），因此这几个区域的内存分配和回收都具备确定性，在这几个区域内不需要过多考虑回收的问题，因为方法结束或线程结束时，内存自然就跟随着回收了。而 Java 堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分内存，本书后续讨论中的“内存”分配与回收也仅指这一部分内存。

3.2 对象已死？

堆中几乎存放着 Java 世界中所有的对象实例，垃圾收集器在对堆进行回收前，第一件事情就是要确定这些对象有哪些还“存活”着，哪些已经“死去”（即不可能再被任何途径使用的对象）。

3.2.1 引用计数算法

很多教科书判断对象是否存活的算法是这样的：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器都为 0 的对象就是不可能再被使用的。笔者面试过很多的应届生和一些有多年工作经验的开发人员，他们对于这个问题给予的都是这个答案。

客观地说，引用计数算法（Reference Counting）的实现简单，判定效率也很高，在大部分情况下它都是一个不错的算法，也有一些比较著名的应用案例，例如微软的 COM（Component Object Model）技术、使用 ActionScript 3 的 FlashPlayer、Python 语言以及在游戏脚本领域中被广泛应用的 Squirrel 中都使用了引用计数算法进行内存管理。但是，Java 语言中没有选用引用计数算法来管理内存，其中最主要的原因是它很难解决对象之间的相互循环引用的问题。

举个简单的例子，请看代码清单 3-1 中的 testGC()方法：对象 objA 和 objB 都有字段 instance，赋值令 objA.instance = objB 及 objB.instance = objA，除此之外，这两个对象再无任何引用，实际上这两个对象已经不可能再被访问，但是它们因为互相引用着对方，导致它们的引用计数都不为 0，于是引用计数算法无法通知 GC 收集器回收它们。

代码清单 3-1　引用计数算法的缺陷

```
/**
* testGC()方法执行后，objA 和 objB 会不会被 GC 呢？
* @author zzm
```

```
*/
public class ReferenceCountingGC {

    public Object instance = null;

    private static final int _1MB = 1024 * 1024;

    /**
    * 这个成员属性的唯一意义就是占点内存，以便能在 GC 日志中看清楚是否被回收过
    */
    private byte[] bigSize = new byte[2 * _1MB];

    public static void testGC() {
        ReferenceCountingGC objA = new ReferenceCountingGC();
        ReferenceCountingGC objB = new ReferenceCountingGC();
        objA.instance = objB;
        objB.instance = objA;

        objA = null;
        objB = null;

        // 假设在这行发生 GC，那么 objA 和 objB 是否能被回收？
        System.gc();
    }
}
```

运行结果：

```
[Full GC (System) [Tenured: 0K->210K(10240K), 0.0149142 secs] 4603K->210K(19456K), [Perm :
2999K->2999K(21248K)], 0.0150007 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
Heap
def new generation    total 9216K, used 82K [0x00000000055e0000, 0x0000000005fe0000,
0x0000000005fe0000)
  Eden space 8192K,    1% used [0x00000000055e0000, 0x00000000055f4850,
```

0x0000000005de0000)

from space 1024K, 0% used [0x0000000005de0000, 0x0000000005de0000, 0x0000000005ee0000)

to space 1024K, 0% used [0x0000000005ee0000, 0x0000000005ee0000, 0x0000000005fe0000)

tenured generation total 10240K, used 210K [0x0000000005fe0000, 0x00000000069e0000, 0x00000000069e0000)

the space 10240K, 2% used [0x0000000005fe0000, 0x0000000006014a18, 0x0000000006014c00, 0x00000000069e0000)

compacting perm gen total 21248K, used 3016K [0x00000000069e0000, 0x0000000007ea0000, 0x000000000bde0000)

the space 21248K, 14% used [0x00000000069e0000, 0x0000000006cd2398, 0x0000000006cd2400, 0x0000000007ea0000)

No shared spaces configured.

从运行结果中可以清楚地看到 GC 日志中包含“4603K->210K”，意味着虚拟机并没有因为这两个对象互相引用就不回收它们，这也从侧面说明虚拟机并不是通过引用计数算法来判断对象是否存活的。

3.2.2 根搜索算法

在主流的商用程序语言中（Java 和 C#，甚至包括前面提到的古老的 Lisp），都是使用根搜索算法（GC Roots Tracing）判定对象是否存活的。这个算法的基本思路就是通过一系列的名为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连（用图论的话来说就是从 GC Roots 到这个对象不可达）时，则证明此对象是不可用的。如图 3-1 所示，对象 object 5、object 6、object 7 虽然互相有关联，但是它们到 GC Roots 是不可达的，所以它们将会被判定为是可回收的对象。

在 Java 语言里，可作为 GC Roots 的对象包括下面几种：

虚拟机栈（栈帧中的本地变量表）中的引用的对象。

方法区中的类静态属性引用的对象。

方法区中的常量引用的对象。

本地方法栈中 JNI（即一般说的 Native 方法）的引用的对象。

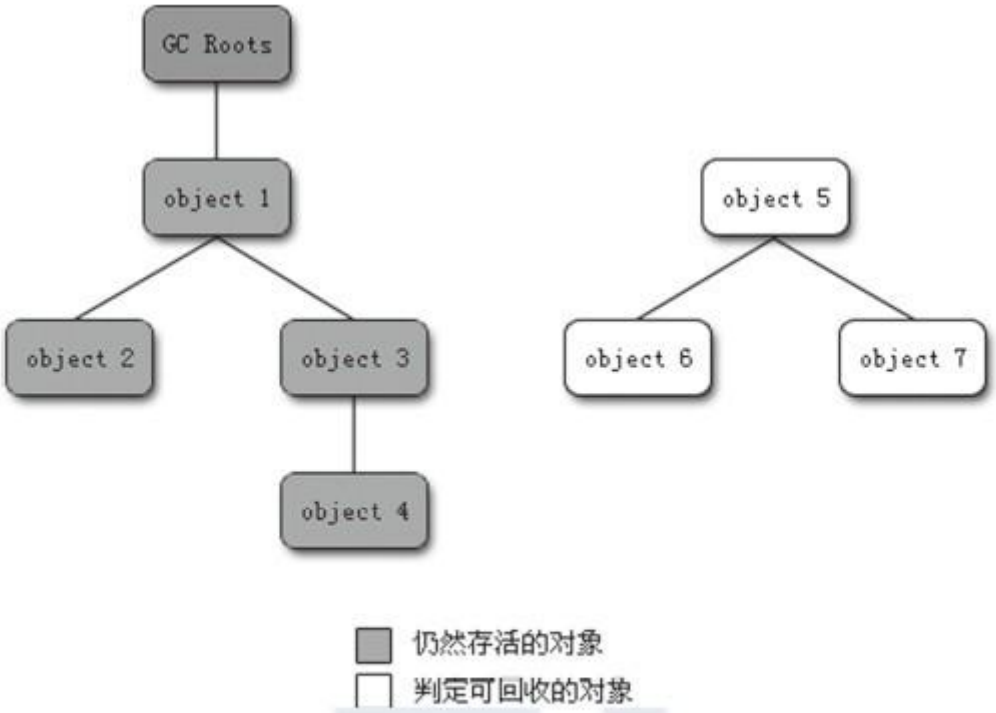

图 3-1　根搜索算法判定对象是否可回收

3.2.3　再谈引用

无论是通过引用计数算法判断对象的引用数量，还是通过根搜索算法判断对象的引用链是否可达，判定对象是否存活都与“引用”有关。在 JDK 1.2 之前，Java 中的引用的定义很传统：如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。这种定义很纯粹，但是太过狭隘，一个对象在这种定义下只有被引用或者没有被引用两种状态，对于如何描述一些“食之无味，弃之可惜”的对象就显得无能为力。我们希望能描述这样一类对象：当内存空间还足够时，则能保留在内存之中；如果内存在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。

在 JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）四种，这四种引用强度依次逐渐减弱。

强引用就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

软引用用来描述一些还有用，但并非必需的对象。对于软引用关联着的对象，在系统将要发生内

存溢出异常之前，将会把这些对象列进回收范围之中并进行第二次回收。如果这次回收还是没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 SoftReference 类来实现软引用。

弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在 JDK 1.2 之后，提供了 WeakReference 类来实现弱引用。

虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被收集器回收时收到一个系统通知。在 JDK 1.2 之后，提供了 PhantomReference 类来实现虚引用。

3.2.4 生存还是死亡？

在根搜索算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行根搜索后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize()方法。当对象没有覆盖 finalize()方法，或者 finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行 finalize()方法，那么这个对象将会被放置在一个名为 F-Queue 的队列之中，并在稍后由一条由虚拟机自动建立的、低优先级的 Finalizer 线程去执行。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这样做的原因是，如果一个对象在 finalize()方法中执行缓慢，或者发生了死循环（更极端的情况），将很可能会导致 F-Queue 队列中的其他对象永久处于等待状态，甚至导致整个内存回收系统崩溃。finalize()方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize()中成功拯救自己—只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this 关键字）赋值给某个类变量或对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那它就真的离死不远了。从代码清单 3-2 中我们可以看到一个对象的 finalize()被执行，但是它仍然可以存活。

从代码清单 3-2 的运行结果可以看到，SAVE_HOOK 对象的 finalize()方法确实被 GC 收集器触发过，并且在被收集前成功逃脱了。

Java 代码

```
1.  代码清单 3-2  一次对象自我拯救的演示
2.  /**
3.  * 此代码演示了两点：
```

```
4.  * 1.对象可以在被 GC 时自我拯救。
5.  * 2.这种自救的机会只有一次，因为一个对象的 finalize()方法最多只会被系统自动调用一次
6.  * @author zzm
7.  */
8.  public class FinalizeEscapeGC {
9.
10.     public static FinalizeEscapeGC SAVE_HOOK = null;
11.
12.     public void isAlive() {
13.       System.out.println(&quot;yes, i am still alive[img]/images/smiles/icon_smile.gif&
    quot; alt=&quot;[/img]&quot;);
14.     }
15.
16.     @Override
17.     protected void finalize() throws Throwable {
18.         super.finalize();
19.       System.out.println(&quot;finalize mehtod executed!&quot;);
20.       FinalizeEscapeGC.SAVE_HOOK = this;
21.     }
22.
23.     public static void main(String[] args) throws Throwable {
24.       SAVE_HOOK = new FinalizeEscapeGC();
25.
26.       //对象第一次成功拯救自己
27.       SAVE_HOOK = null;
28.       System.gc();
29.       // 因为 Finalizer 方法优先级很低，暂停 0.5 秒，以等待它
30.       Thread.sleep(500);
31.       if (SAVE_HOOK != null) {
32.       SAVE_HOOK.isAlive();
33.       } else {
34.       System.out.println(&quot;no, i am dead[img]/images/smiles/icon_sad.gif&quot; alt=
    &quot;[/img]&quot;);
35.       }
36.
37.       // 下面这段代码与上面的完全相同，但是这次自救却失败了
38.       SAVE_HOOK = null;
39.       System.gc();
40.       // 因为 Finalizer 方法优先级很低，暂停 0.5 秒，以等待它
41.       Thread.sleep(500);
42.       if (SAVE_HOOK != null) {
43.       SAVE_HOOK.isAlive();
44.       } else {
```

```
45.       System.out.println(&quot;no, i am dead[img]/images/smiles/icon_sad.gif&quot; alt=
    &quot;[/img]&quot;);
46.       }
47.     }
48. }
49. 运行结果:
50. finalize mehtod executed!
51. yes, i am still alive[img]/images/smiles/icon_smile.gif&quot; alt=&quot;[/img]
52. no, i am dead[img]/images/smiles/icon_sad.gif&quot; alt=&quot;[/img]
```

另外一个值得注意的地方就是，代码中有两段完全一样的代码片段，执行结果却是一次逃脱成功，一次失败，这是因为任何一个对象的 finalize()方法都只会被系统自动调用一次，如果对象面临下一次回收，它的 finalize()方法不会被再次执行，因此第二段代码的自救行动失败了。

需要特别说明的是，上面关于对象死亡时 finalize()方法的描述可能带有悲情的艺术色彩，笔者并不鼓励大家使用这种方法来拯救对象。相反，笔者建议大家尽量避免使用它，因为它不是 C/C++中的析构函数，而是 Java 刚诞生时为了使 C/C++程序员更容易接受它所做出的一个妥协。它的运行代价高昂，不确定性大，无法保证各个对象的调用顺序。有些教材中提到它适合做“关闭外部资源”之类的工作，这完全是对这种方法的用途的一种自我安慰。finalize()能做的所有工作，使用 try-finally 或其他方式都可以做得更好、更及时，大家完全可以忘掉 Java 语言中还有这个方法的存在。

3.2.5 回收方法区

很多人认为方法区（或者 HotSpot 虚拟机中的永久代）是没有垃圾收集的，Java 虚拟机规范中确实说过可以不要求虚拟机在方法区实现垃圾收集，而且在方法区进行垃圾收集的“性价比”一般比较低：在堆中，尤其是在新生代中，常规应用进行一次垃圾收集一般可以回收 70%~95%的空间，而永久代的垃圾收集效率远低于此。

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。回收废弃常量与回收 Java 堆中的对象非常类似。以常量池中字面量的回收为例，假如一个字符串“abc”已经进入了常量池中，但是当前系统没有任何一个 String 对象是叫做“abc”的，换句话说是没有任何 String 对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，如果在这时候发生内存回收，而且必要的话，这个“abc”常量就会被系统“请”出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是“无用的类”：

该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。

加载该类的 ClassLoader 已经被回收。

该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而不是和对象一样，不使用了就必然会回收。是否对类进行回收，HotSpot 虚拟机提供了-Xnoclassgc 参数进行控制，还可以使用-verbose:class 及-XX:+TraceClassLoading、 -XX:+TraceClassUnLoading 查看类的加载和卸载信息。

在大量使用反射、动态代理、CGLib 等 bytecode 框架的场景，以及动态生成 JSP 和 OSGi 这类频繁自定义 ClassLoader 的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

3.3 垃圾收集算法

由于垃圾收集算法的实现涉及大量的程序细节，而且各个平台的虚拟机操作内存的方法又各不相同，因此本节不打算过多地讨论算法的实现，只是介绍几种算法的思想及其发展过程。

3.3.1 标记 -清除算法

最基础的收集算法是“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象，它的标记过程其实在前一节讲述对象标记判定时已经基本介绍过了。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进而得到的。它的主要缺点有两个：一个是效率问题，标记和清除过程的效率都不高；另外一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。标记-清除算法的执行过程如图 3-2 所示。

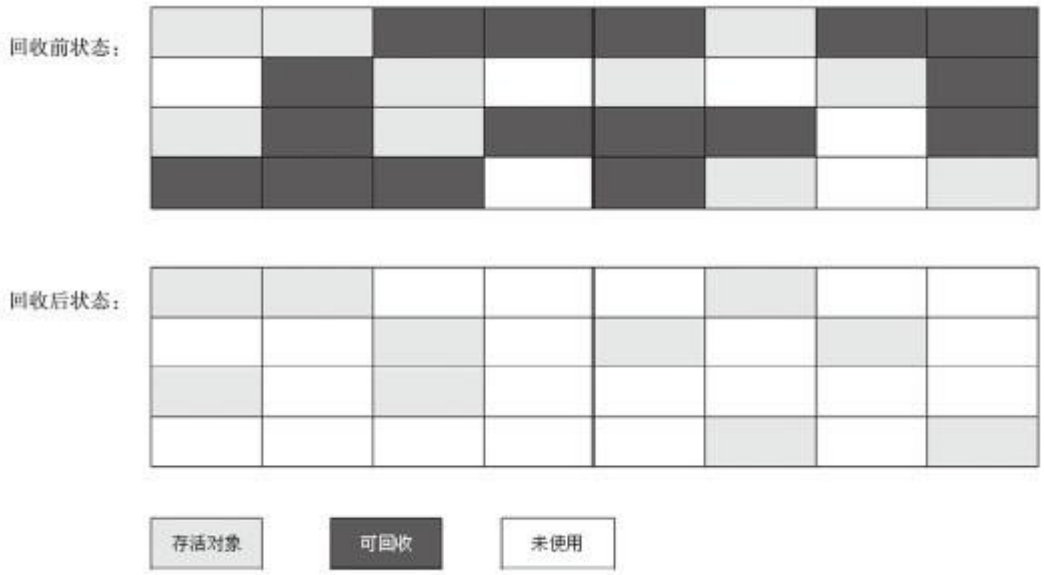

图 3-2 “标记－清除”算法示意图

3.3.2 复制算法

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半，未免太高了一点。复制算法的执行过程如图 3-3 所示。

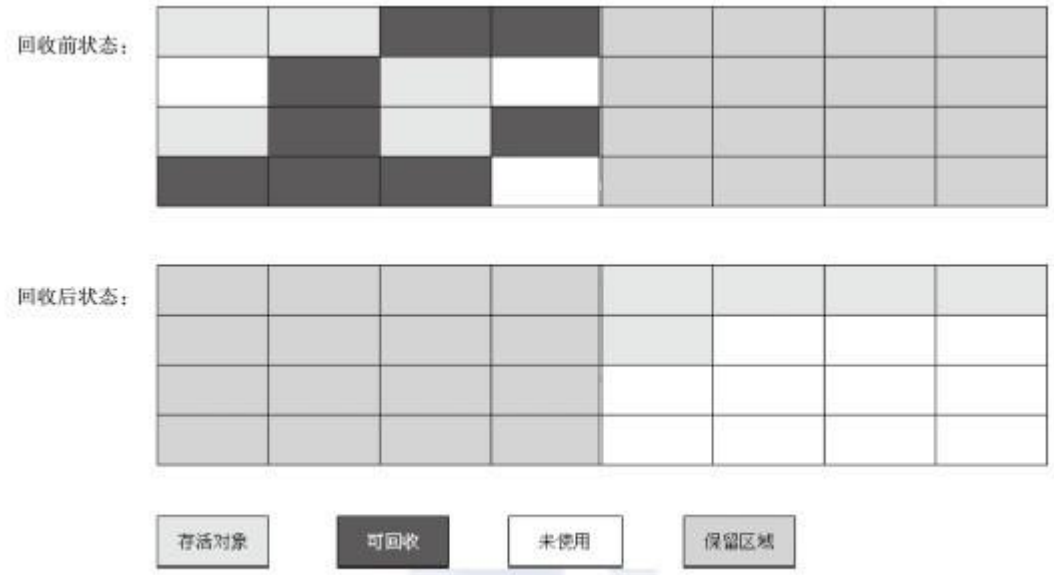

图 3-3　复制算法示意图

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM 的专门研究表明，新生代中的对象98%是朝生夕死的，所以并不需要按照 1：1 的比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中的一块 Survivor①。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地拷贝到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 的空间。HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8：1，也就是每次新生代中可用内存空间为整个新生代容量的 90%（80%+10%），只有 10%的内存是会被“浪费”的。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于 10%的对象存活，当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

内存的分配担保就好比我们去银行借款，如果我们信誉很好，在 98%的情况下都能按时偿还，于是银行可能会默认我们下一次也能按时按量地偿还贷款，只需要有一个担保人能保证如果我不能还款时，可以从他的账户扣钱，那银行就认为没有风险了。内存的分配担保也一样，如果另外一块 Survivor 空间没有足够的空间存放上一次新生代收集下来的存活对象，这些对象将直接通过分配担保机制进入老年代。关于对新生代进行分配担保的内容，本章稍后在讲解垃圾收集器执行规则时还会再详细讲解。

3.3.3　标记-整理算法

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费 50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都 100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存，“标记-整理”算法的示意图如图 3-4 所示。

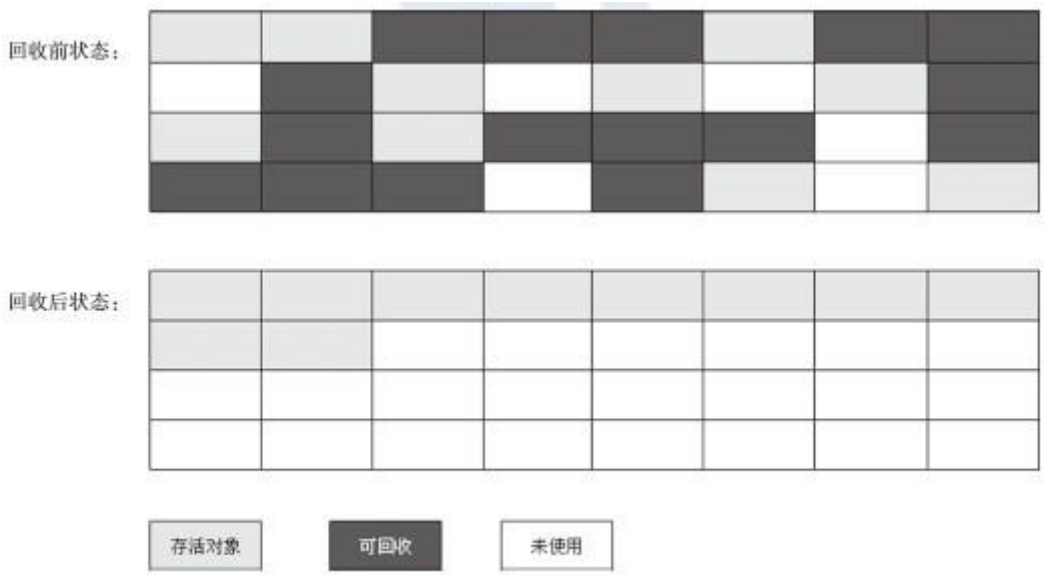

图 3-4 “标记－整理”算法示意图

3.3.4 分代收集算法

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法并没有什么新的思想，只是根据对象的存活周期的不同将内存划分为几块。一般是把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

3.4 垃圾收集器

如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现。Java 虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同的厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大的差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器。这里讨论的收集器基于 Sun HotSpot 虚拟机 1.6 版 Update 22，这个虚拟机包含的所有收集器如图 3-5 所示。

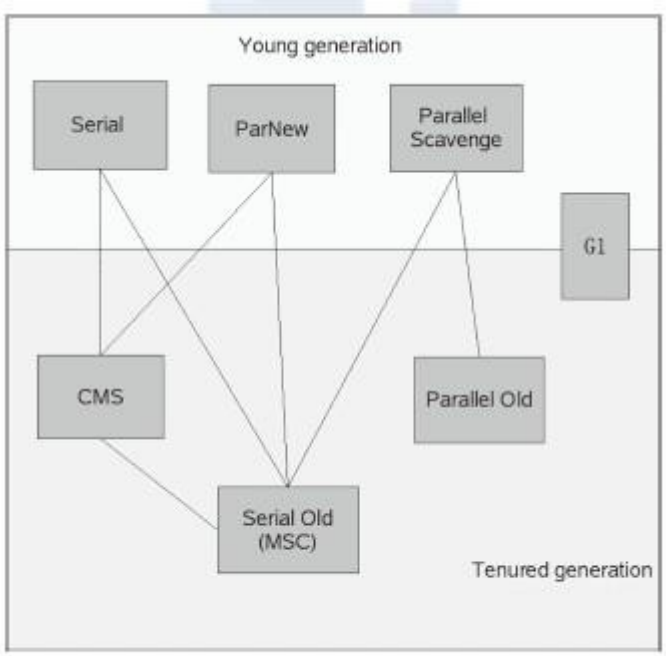

图 3-5 HotSpot JVM1.6 的垃圾收集器[①]

图 3-5 展示了 7 种作用于不同分代的收集器（包括 JDK 1.6_Update14 后引入的 Early Access 版 G1 收集器），如果两个收集器之间存在连线，就说明它们可以搭配使用。

在介绍这些收集器各自的特性之前，我们先来明确一个观点：虽然我们是在对各个收集器进行比较，但并非为了挑选一个最好的收集器出来。因为直到现在为止还没有最好的收集器出现，更加没有万能的收集器，所以我们选择的只是对具体应用最合适的收集器。这点不需要多加解释就能证明：如果有一种放之四海皆准、任何场景下都适用的完美收集器存在，那 HotSpot 虚拟机就没必要实现那么多不同的收集器了。

3.4.1 Serial 收集器

Serial 收集器是最基本、历史最悠久的收集器，曾经（在 JDK 1.3.1 之前）是虚拟机新生代收集的唯一选择。大家看名字就知道，这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅是说明它只会使用一个 CPU 或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程（Sun 将这件事情称之为“Stop The World”），直到它收集结束。“Stop

The World”这个名字也许听起来很酷，但这项工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户的正常工作的线程全部停掉，这对很多应用来说都是难以接受的。你想想，要是你的电脑每运行一个小时就会暂停响应 5 分钟，你会有什么样的心情？图 3-6 示意了 Serial / Serial Old 收集器的运行过程。

对于“Stop The World”带给用户的恶劣体验，虚拟机的设计者们表示完全理解，但也表示非常委屈：“你妈妈在给你打扫房间的时候，肯定也会让你老老实实地在椅子上或房间外待着，如果她一边打扫，你一边乱扔纸屑，这房间还能打扫完吗？”这确实是一个合情合理的矛盾，虽然垃圾收集这项工作听起来和打扫房间属于一个性质的，但实际上肯定还要比打扫房间复杂得多啊！

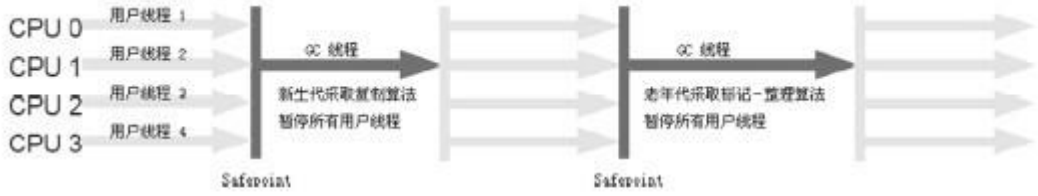

图 3-6　Serial / Serial Old 收集器运行示意图

从 JDK 1.3 开始，一直到现在还没正式发布的 JDK 1.7，HotSpot 虚拟机开发团队为消除或减少工作线程因内存回收而导致停顿的努力一直在进行着，从 Serial 收集器到 Parallel 收集器，再到 Concurrent Mark Sweep（CMS）现在还未正式发布的 Garbage First（G1）收集器，我们看到了一个个越来越优秀（也越来越复杂）的收集器的出现，用户线程的停顿时间在不断缩短，但是仍然没有办法完全消除（这里暂不包括 RTSJ 中的收集器）。寻找更优秀的垃圾收集器的工作仍在继续！

写到这里，笔者似乎已经把 Serial 收集器描述成一个老而无用，食之无味弃之可惜的鸡肋了，但实际上到现在为止，它依然是虚拟机运行在 Client 模式下的默认新生代收集器。它也有着优于其他收集器的地方：简单而高效（与其他收集器的单线程比），对于限定单个 CPU 的环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本上不会再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。所以，Serial 收集器对于运行在 Client 模式下的虚拟机来说是一个很好的选择。

3.4.2　ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括 Serial 收集器可用的所有控制参数（例如：-XX:SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure 等）、收集算法、Stop The World、对象分配规则、回收策略等都与 Serial 收集器完全一样，实现上这两种收集器也共用了相当多的代码。ParNew 收集器的工作过程如图 3-7 所示。

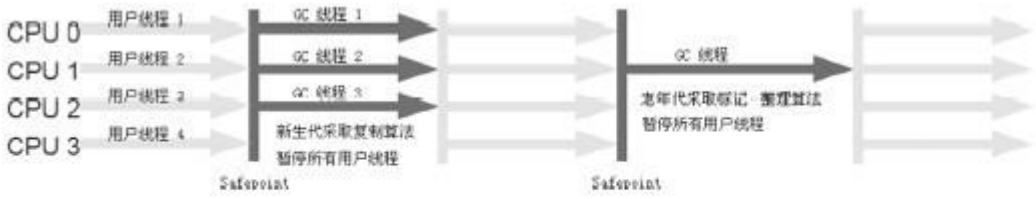

图 3-7　ParNew / Serial Old 收集器运行示意图

ParNew 收集器除了多线程收集之外，其他与 Serial 收集器相比并没有太多创新之处，但它却是许多运行在 Server 模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关但很重要的原因是，除了 Serial 收集器外，目前只有它能与 CMS 收集器配合工作。在 JDK 1.5 时期，HotSpot 推出了一款在强交互应用中几乎可称为有划时代意义的垃圾收集器—CMS 收集器（Concurrent Mark Sweep，本节稍后将详细介绍这款收集器），这款收集器是 HotSpot 虚拟机中第一款真正意义上的并发（Concurrent）收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作，用前面那个例子的话来说，就是做到了在你妈妈打扫房间的时候你还能同时往地上扔纸屑。

不幸的是，它作为老年代的收集器，却无法与 JDK 1.4.0 中已经存在的新生代收集器 Parallel Scavenge 配合工作①，所以在 JDK 1.5 中使用 CMS 来收集老年代的时候，新生代只能选择 ParNew 或 Serial 收集器中的一个。ParNew 收集器也是使用 -XX: +UseConcMarkSweepGC 选项后的默认新生代收集器，也可以使用 -XX:+UseParNewGC 选项来强制指定它。

ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百地保证能超越 Serial 收集器。当然，随着可以使用的 CPU 的数量的增加，它对于 GC 时系统资源的利用还是很有好处的。它默认开启的收集线程数与 CPU 的数量相同，在 CPU 非常多（譬如 32 个，现在 CPU 动辄就 4 核加超线程，服务器超过 32 个逻辑 CPU 的情况越来越多了）的环境下，可以使用 -XX:ParallelGCThreads 参数来限制垃圾收集的线程数。

注意　从 ParNew 收集器开始，后面还将会接触到几款并发和并行的收集器。在大家可能产生疑惑之前，有必要先解释两个名词：并发和并行。这两个名词都是并发编程中的概念，在谈论垃圾收集

器的上下文语境中，他们可以解释为：　　并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。 并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序继续运行，而垃圾收集程序运行于另一个 CPU 上

3.4.3　Parallel Scavenge 收集器

Parallel Scavenge 收集器也是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器……看上去和 ParNew 都一样，那它有什么特别之处呢？

Parallel Scavenge 收集器的特点是它的关注点与其他收集器不同，CMS 等收集器的关注点尽可能地缩短垃圾收集时用户线程的停顿时间，而 Parallel Scavenge 收集器的目标则是达到一个可控制的吞吐量（Throughput）。所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间），虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那吞吐量就是 99%。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户的体验；而高吞吐量则可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge 收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis 参数及直接设置吞吐量大小的 -XX:GCTimeRatio 参数。

MaxGCPauseMillis 参数允许的值是一个大于 0 的毫秒数，收集器将尽力保证内存回收花费的时间不超过设定值。不过大家不要异想天开地认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC 停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集 300MB 新生代肯定比收集 500MB 快吧，这也直接导致垃圾收集发生得更频繁一些，原来 10 秒收集一次、每次停顿 100 毫秒，现在变成 5 秒收集一次、每次停顿 70 毫秒。停顿时间的确在下降，但吞吐量也降下来了。

GCTimeRatio 参数的值应当是一个大于 0 小于 100 的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。如果把此参数设置为 19，那允许的最大 GC 时间就占总时间的 5%（即 1 /（1+19）），默认值为 99，就是允许最大 1%（即 1 /（1+99））的垃圾收集时间。

由于与吞吐量关系密切，Parallel Scavenge 收集器也经常被称为“吞吐量优先”收集器。除上述两个参数之外，Parallel Scavenge 收集器还有一个参数-XX:+UseAdaptiveSizePolicy 值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden 与 Survivor 区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为 GC 自适应的调节策略（GC Ergonomics）①。如果读者对于收集

器运作原理不太了解，手工优化存在困难的时候，使用 Parallel Scavenge 收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个很不错的选择。只需要把基本的内存数据设置好（如-Xmx 设置最大堆），然后使用 MaxGCPauseMillis 参数（更关注最大停顿时间）或 GCTimeRatio 参数（更关注吞吐量）给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。自适应调节策略也是 Parallel Scavenge 收集器与 ParNew 收集器的一个重要区别。

3.4.4 Serial Old 收集器

Serial Old 是 Serial 收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是被 Client 模式下的虚拟机使用。如果在 Server 模式下，它主要还有两大用途：一个是在 JDK 1.5 及之前的版本中与 Parallel Scavenge 收集器搭配使用②，另外一个就是作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 的时候使用。这两点都将在后面的内容中详细讲解。Serial Old 收集器的工作过程如图 3-8 所示。

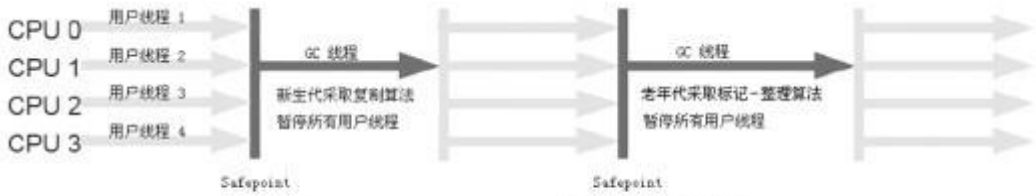

图 3-8 Serial / Serial Old 收集器运行示意图

3.4.5 Parallel Old 收集器

Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记－整理”算法。这个收集器是在 JDK 1.6 中才开始提供的，在此之前，新生代的 Parallel Scavenge 收集器一直处于比较尴尬的状态。原因是，如果新生代选择了 Parallel Scavenge 收集器，老年代除了 Serial Old（PS MarkSweep）收集器外别无选择（还记得上面说过 Parallel Scavenge 收集器无法与 CMS 收集器配合工作吗？）。由于单线程的老年代 Serial Old 收集器在服务端应用性能上的“拖累”，即便使用了 Parallel Scavenge 收集器也未必能在整体应用上获得吞吐量最大化的效果，又因为老年代收集中无法充分利用服务器多 CPU 的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有 ParNew 加 CMS 的组合“给力”。

直到 Parallel Old 收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注

重吞吐量及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。Parallel Old收集器的工作过程如图3-9所示。

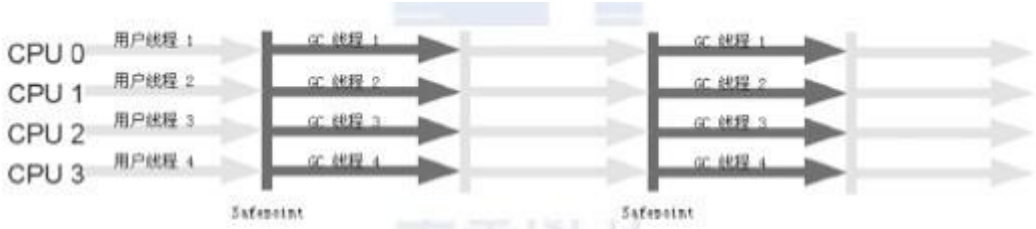

图3-9　Parallel Scavenge / Parallel Old收集器运行示意图

3.4.6　CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。从名字（包含“Mark Sweep”）上就可以看出CMS收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为4个步骤，包括：

初始标记（CMS initial mark）

并发标记（CMS concurrent mark）

重新标记（CMS remark）

并发清除（CMS concurrent sweep）

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行的。通过图3-10可以比较清楚地看到CMS收集器的运作步骤中并发和需要停顿的时间。

图 3-10　Concurrent Mark Sweep 收集器运行示意图

CMS 是一款优秀的收集器，它的最主要优点在名字上已经体现出来了：并发收集、低停顿，Sun 的一些官方文档里面也称之为并发低停顿收集器（Concurrent Low Pause Collector）。但是 CMS 还远达不到完美的程度，它有以下三个显著的缺点：

CMS 收集器对 CPU 资源非常敏感。其实，面向并发设计的程序都对 CPU 资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说 CPU 资源）而导致应用程序变慢，总吞吐量会降低。CMS 默认启动的回收线程数是（CPU 数量+3）/ 4，也就是当 CPU 在 4 个以上时，并发回收时垃圾收集线程最多占用不超过 25%的 CPU 资源。但是当 CPU 不足 4 个时（譬如 2 个），那么 CMS 对用户程序的影响就可能变得很大，如果 CPU 负载本来就比较大的时候，还分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了 50%，这也很让人受不了。为了解决这种情况，虚拟机提供了一种称为“增量式并发收集器”（Incremental Concurrent Mark Sweep / i-CMS）的 CMS 收集器变种，所做的事情和单 CPU 年代 PC 机操作系统使用抢占式来模拟多任务机制的思想一样，就是在并发标记和并发清理的时候让 GC 线程、用户线程交替运行，尽量减少 GC 线程的独占资源的时间，这样整个垃圾收集的过程会更长，但对用户程序的影响就会显得少一些，速度下降也就没有那么明显，但是目前版本中，i-CMS 已经被声明为“deprecated”，即不再提倡用户使用。

CMS 收集器无法处理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生。由于 CMS 并发清理阶段用户线程还在运行着，伴随程序的运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在本次收集中处理掉它们，只好留待下一次 GC 时再将其清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，即还需要预留足够的内存空间给用户线程使用，因此 CMS 收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在默认设置下，CMS 收集器在老年代使用了 68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction 的值来提高触发百分比，以便降低内存回收次数以获取更好的性能。要是 CMS 运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时候虚拟机将启动后备预案：临

时启用 Serial Old 收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数 -XX:CMSInitiatingOccupancyFraction 设置得太高将会很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

还有最后一个缺点，在本节在开头说过，CMS 是一款基于“标记-清除”算法实现的收集器，如果读者对前面这种算法介绍还有印象的话，就可能想到这意味着收集结束时会产生大量空间碎片。空间碎片过多时，将会给大对象分配带来很大的麻烦，往往会出现老年代还有很大的空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次 Full GC。为了解决这个问题，CMS 收集器提供了一个-XX:+UseCMSCompactAtFullCollection 开关参数，用于在“享受”完 Full GC 服务之后额外免费附送一个碎片整理过程，内存整理的过程是无法并发的。空间碎片问题没有了，但停顿时间不得不变长了。虚拟机设计者们还提供了另外一个参数-XX: CMSFullGCsBeforeCompaction，这个参数用于设置在执行多少次不压缩的 Full GC 后，跟着来一次带压缩的。

3.4.7 G1 收集器

G1（Garbage First）收集器是当前收集器技术发展的最前沿成果，在 JDK 1.6_Update14 中提供了 Early Access 版本的 G1 收集器以供试用。在将来 JDK 1.7 正式发布的时候，G1 收集器很可能会有一个成熟的商用版本随之发布。这里只对 G1 收集器进行简单介绍①。

G1 收集器是垃圾收集器理论进一步发展的产物，它与前面的 CMS 收集器相比有两个显著的改进：一是 G1 收集器是基于“标记-整理”算法实现的收集器，也就是说它不会产生空间碎片，这对于长时间运行的应用系统来说非常重要。二是它可以非常精确地控制停顿，既能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒，这几乎已经是实时 Java（RTSJ）的垃圾收集器的特征了。

G1 收集器可以实现在基本不牺牲吞吐量的前提下完成低停顿的内存回收，这是由于它能够极力地避免全区域的垃圾收集，之前的收集器进行收集的范围都是整个新生代或老年代，而 G1 将整个 Java 堆（包括新生代、老年代）划分为多个大小固定的独立区域（Region），并且跟踪这些区域里面的垃圾堆积程度，在后台维护一个优先列表，每次根据允许的收集时间，优先回收垃圾最多的区域（这就是 Garbage First 名称的来由）。区域划分及有优先级的区域回收，保证了 G1 收集器在有限的时间内可以获得最高的收集效率。

3.4.8 垃圾收集器参数总结

JDK 1.6 中的各种垃圾收集器到此已全部介绍完毕，在描述过程中提到了很多虚拟机非稳定的运行参数，表 3-1 整理了这些参数以供读者实践时参考。

表 3-1　垃圾收集相关的常用参数

表 3-1　垃圾收集相关的常用参数

参　数	描　述
UseSerialGC	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial + Serial Old 的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为 CMS 收集器出现 Concurrent Mode Failure 失败后的后备收集器使用
UseParallelGC	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收
SurvivorRatio	新生代中 Eden 区域与 Survivor 区域的容量比值，默认为 8，代表 Eden：Survivor=8：1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄。每个对象在坚持过一次 Minor GC 之后，年龄就加 1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整 Java 堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	设置并行 GC 时进行内存回收的线程数
GCTimeRatio	GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效
MaxGCPauseMillis	设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为 68%，仅在使用 CMS 收集器时生效
UseCMSCompactAtFullCollection	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效
CMSFullGCsBeforeCompaction	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效

3.5　内存分配与回收策略

Java 技术体系中所提倡的自动内存管理最终可以归结为自动化地解决了两个问题：给对象分配内存以及回收分配给对象的内存。关于回收内存这一点，我们已经使用了大量的篇幅去介绍虚拟机中的垃圾收集器体系及其运作原理，现在我们再一起来探讨一下给对象分配内存的那点事儿。

对象的内存分配，往大方向上讲，就是在堆上分配（但也可能经过 JIT 编译后被拆散为标量类型并间接地在栈上分配①），对象主要分配在新生代的 Eden 区上，如果启动了本地线程分配缓冲，将按线程优先在 TLAB 上分配。少数情况下也可能会直接分配在老年代中，分配的规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。

接下来我们将会讲解几条最普遍的内存分配规则，并通过代码去验证这些规则。本节中的代码在测试时使用 Client 模式虚拟机运行，没有手工指定收集器组合，换句话说，验证的是使用 Serial / Serial Old 收集器下（ParNew / Serial Old 收集器组合的规则也基本一致）的内存分配和回收的策略。读者不妨根据自己项目中使用的收集器写一些程序去验证一下使用其他几种收集器的内存分配策略。

3.5.1 对象优先在 Eden 分配

大多数情况下，对象在新生代 Eden 区中分配。当 Eden 区没有足够的空间进行分配时，虚拟机将发起一次 Minor GC。

虚拟机提供了-XX:+PrintGCDetails 这个收集器日志参数，告诉虚拟机在发生垃圾收集行为时打印内存回收日志，并且在进程退出的时候输出当前内存各区域的分配情况。在实际应用中，内存回收日志一般是打印到文件后通过日志工具进行分析，不过本实验的日志并不多，直接阅读就能看得很清楚。代码清单 3-3 的 testAllocation()方法中，尝试分配 3 个 2MB 大小和 1 个 4MB 大小的对象，在运行时通过-Xms20M、 -Xmx20M 和 -Xmn10M 这 3 个参数限制 Java 堆大小为 20MB，且不可扩展，其中 10MB 分配给新生代，剩下的 10MB 分配给老年代。-XX:SurvivorRatio=8 决定了新生代中 Eden 区与一个 Survivor 区的空间比例是 8 比 1，从输出的结果也能清晰地看到“eden space 8192K、from space 1024K、to space 1024K”的信息，新生代总可用空间为 9216KB（Eden 区+1 个 Survivor 区的总容量）。

执行 testAllocation()中分配 allocation4 对象的语句时会发生一次 Minor GC，这次 GC 的结果是新生代 6651KB 变为 148KB，而总内存占用量则几乎没有减少（因为 allocation1、2、3 三个对象都是存活的，虚拟机几乎没有找到可回收的对象）。这次 GC 发生的原因是给 allocation4 分配内存的时候，发现 Eden 已经被占用了 6MB，剩余空间已不足以分配 allocation4 所需的 4MB 内存，因此发生 Minor GC。GC 期间虚拟机又发现已有的 3 个 2MB 大小的对象全部无法放入 Survivor 空间（Survivor 空间只有 1MB 大小），所以只好通过分配担保机制提前转移到老年代去。

这次 GC 结束后，4MB 的 allocation4 对象被顺利分配在 Eden 中。因此程序执行完的结果是 Eden 占用 4MB（被 allocation4 占用），Survivor 空闲，老年代被占用 6MB（被 allocation1、2、3 占用）。通过 GC 日志可以证实这一点。

注意　作者多次提到的 Minor GC 和 Full GC 有什么不一样吗？

新生代 GC（Minor GC）：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的

特性，所以 Minor GC 非常频繁，一般回收速度也比较快。

老年代 GC（Major GC / Full GC）：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程）。MajorGC 的速度一般会比 Minor GC 慢 10 倍以上。

代码清单 3-3　新生代 Minor GC

```
private static final int _1MB = 1024 * 1024;
/**
 * VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
 */
public static void testAllocation() {
  byte[] allocation1, allocation2, allocation3, allocation4;
  allocation1 = new byte[2 * _1MB];
  allocation2 = new byte[2 * _1MB];
  allocation3 = new byte[2 * _1MB];
  allocation4 = new byte[4 * _1MB];   //  出现一次 Minor GC
 }
```

运行结果：

```
[GC [DefNew: 6651K->148K(9216K), 0.0070106 secs] 6651K->6292K(19456K), 0.0070426 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation    total 9216K, used 4326K [0x029d0000, 0x033d0000, 0x033d0000)
   eden space 8192K,   51% used [0x029d0000, 0x02de4828, 0x031d0000)
   from space 1024K,   14% used [0x032d0000, 0x032f5370, 0x033d0000)
   to    space 1024K,    0% used [0x031d0000, 0x031d0000, 0x032d0000)
 tenured generation    total 10240K, used 6144K [0x033d0000, 0x03dd0000, 0x03dd0000)
    the space 10240K,   60% used [0x033d0000, 0x039d0030, 0x039d0200, 0x03dd0000)
 compacting perm gen   total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
    the space 12288K,   17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

3.5.2　大对象直接进入老年代

所谓大对象就是指，需要大量连续内存空间的 Java 对象，最典型的大对象就是那种很长的字符串及数组（笔者例子中的 byte[]数组就是典型的大对象）。大对象对虚拟机的内存分配来说就是一个坏消息（替 Java 虚拟机抱怨一句，比遇到一个大对象更加坏的消息就是遇到一群“朝生夕灭”的“短命大对象”，写程序的时候应当避免），经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。

虚拟机提供了一个-XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代中分配。这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存拷贝（复习一下：新生代采用复制算法收集内存）。

执行代码清单 3-4 中的 testPretenureSizeThreshold()方法后，我们看到 Eden 空间几乎没有被使用，而老年代 10MB 的空间被使用了 40%，也就是 4MB 的 allocation 对象直接就分配在老年代中，这是因为 PretenureSizeThreshold 被设置为 3MB（就是 3145728B，这个参数不能与-Xmx 之类的参数一样直接写 3MB），因此超过 3MB 的对象都会直接在老年代中进行分配。

注意　PretenureSizeThreshold 参数只对 Serial 和 ParNew 两款收集器有效，Parallel Scavenge 收集器不认识这个参数，Parallel Scavenge 收集器一般并不需要设置。如果遇到必须使用此参数的场合，可以考虑 ParNew 加 CMS 的收集器组合。

代码清单 3-4　大对象直接进入老年代

```
private static final int _1MB = 1024 * 1024;


/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
* -XX:PretenureSizeThreshold=3145728
*/
public static void testPretenureSizeThreshold() {
        byte[] allocation;
        allocation = new byte[4 * _1MB];  //直接分配在老年代中
}
```

运行结果：

```
Heap
def new generation   total 9216K, used 671K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,   8% used [0x029d0000, 0x02a77e98, 0x031d0000)
  from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
```

tenured generation total 10240K, used 4096K [0x033d0000, 0x03dd0000, 0x03dd0000)

  the space 10240K, 40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)

compacting perm gen total 12288K, used 2107K [0x03dd0000, 0x049d0000, 0x07dd0000)

  the space 12288K, 17% used [0x03dd0000, 0x03fdefd0, 0x03fdf000, 0x049d0000)

No shared spaces configured.

3.5.3 长期存活的对象将进入老年代

虚拟机既然采用了分代收集的思想来管理内存，那内存回收时就必须能识别哪些对象应当放在新生代，哪些对象应放在老年代中。为了做到这点，虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1。对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁）时，就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数-XX:MaxTenuringThreshold 来设置。

读者可以试试分别以-XX:MaxTenuringThreshold=1 和-XX:MaxTenuringThreshold=15 两种设置来执行代码清单 3-5 中的 testTenuringThreshold()方法，此方法中 allocation1 对象需要 256KB 的内存空间，Survivor 空间可以容纳。当 MaxTenuringThreshold=1 时，allocation1 对象在第二次 GC 发生时进入老年代，新生代已使用的内存 GC 后会非常干净地变成 0KB。而 MaxTenuringThreshold=15 时，第二次 GC 发生后，allocation1 对象则还留在新生代 Survivor 空间，这时候新生代仍然有 404KB 的空间被占用。

代码清单 3-5　长期存活的对象进入老年代

```
private static final int _1MB = 1024 * 1024;


/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
-XX:MaxTenuringThreshold=1
* -XX:+PrintTenuringDistribution
*/
@SuppressWarnings("unused")
public static void testTenuringThreshold() {
        byte[] allocation1, allocation2, allocation3;
        allocation1 = new byte[_1MB / 4];
```

```
        // 什么时候进入老年代取决于 XX:MaxTenuringThreshold 设置
        allocation2 = new byte[4 * _1MB];
        allocation3 = new byte[4 * _1MB];
        allocation3 = null;
        allocation3 = new byte[4 * _1MB];
}
```

以 MaxTenuringThreshold=1 的参数设置来运行的结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 1)
- age   1:     414664 bytes,     414664 total
: 4859K->404K(9216K), 0.0065012 secs] 4859K->4500K(19456K), 0.0065283 secs] [Times:
user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 1)
: 4500K->0K(9216K), 0.0009253 secs] 8596K->4500K(19456K), 0.0009458 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation   total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,   51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,    0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,    0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation   total 10240K, used 4500K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,   43% used [0x033d0000, 0x03835348, 0x03835400, 0x03dd0000)
compacting perm gen   total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,   17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

以 MaxTenuringThreshold=15 的参数设置来运行的结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
- age   1:     414664 bytes,     414664 total
: 4859K->404K(9216K), 0.0049637 secs] 4859K->4500K(19456K), 0.0049932 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
```

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
- age   2:     414520 bytes,     414520 total
: 4500K->404K(9216K), 0.0008091 secs] 8596K->4500K(19456K), 0.0008305 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation   total 9216K, used 4582K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,  39% used [0x031d0000, 0x03235338, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)
compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

3.5.4 动态对象年龄判定

为了能更好地适应不同程序的内存状况，虚拟机并不总是要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到 MaxTenuringThreshold 中要求的年龄。

执行代码清单 3-6 中的 testTenuringThreshold2()方法，并设置参数-XX: MaxTenuringThreshold=15，会发现运行结果中 Survivor 的空间占用仍然为 0%，而老年代比预期增加了 6%，也就是说 allocation1、allocation2 对象都直接进入了老年代，而没有等到 15 岁的临界年龄。因为这两个对象加起来已经达到了 512KB，并且它们是同年的，满足同年对象达到 Survivor 空间的一半规则。我们只要注释掉其中一个对象的 new 操作，就会发现另外一个不会晋升到老年代中去了。

代码清单 3-6　动态对象年龄判定

```
private static final int _1MB = 1024 * 1024;


/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8
```

```
-XX:MaxTenuringThreshold=15
* -XX:+PrintTenuringDistribution
*/
@SuppressWarnings("unused")
public static void testTenuringThreshold2() {
        byte[] allocation1, allocation2, allocation3, allocation4;
        allocation1 = new byte[_1MB / 4];
                // allocation1+allocation2 大于 survivor 空间的一半
        allocation2 = new byte[_1MB / 4];
        allocation3 = new byte[4 * _1MB];
        allocation4 = new byte[4 * _1MB];
        allocation4 = null;
        allocation4 = new byte[4 * _1MB];
}
```

运行结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 15)
- age   1:     676824 bytes,     676824 total
: 5115K->660K(9216K), 0.0050136 secs] 5115K->4756K(19456K), 0.0050443 secs] [Times:
user=0.00 sys=0.01, real=0.01 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
: 4756K->0K(9216K), 0.0010571 secs] 8852K->4756K(19456K), 0.0011009 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation    total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,   51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,    0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,    0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation    total 10240K, used 4756K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,   46% used [0x033d0000, 0x038753e8, 0x03875400, 0x03dd0000)
compacting perm gen   total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
```

the space 12288K, 17% used [0x03dd0000, 0x03fe09a0, 0x03fe0a00, 0x049d0000)

No shared spaces configured.

3.5.5 空间分配担保

在发生 Minor GC 时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则改为直接进行一次 Full GC。如果小于，则查看 HandlePromotionFailure 设置是否允许担保失败；如果允许，那只会进行 Minor GC；如果不允许，则也要改为进行一次 Full GC。前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个 Survivor 空间来作为轮换备份，因此当出现大量对象在 Minor GC 后仍然存活的情况时（最极端就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，让 Survivor 无法容纳的对象直接进入老年代。与生活中的贷款担保类似，老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共有多少对象会活下来，在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进行 Full GC 来让老年代腾出更多空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说如果某次 Minor GC 存活后的对象突增，远远高于平均值的话，依然会导致担保失败（Handle Promotion Failure）。如果出现了 HandlePromotionFailure 失败，那就只好在失败后重新发起一次 Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将 HandlePromotionFailure 开关打开，避免 Full GC 过于频繁，参见代码清单 3-7。

代码清单 3-7　空间分配担保

```
private static final int _1MB = 1024 * 1024;


/**
* VM 参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:SurvivorRatio=8 -XX:-
HandlePromotionFailure
*/
@SuppressWarnings("unused")
public static void testHandlePromotion() {
        byte[] allocation1, allocation2, allocation3, allocation4, allocation5, allocation6, allocation7;
        allocation1 = new byte[2 * _1MB];
        allocation2 = new byte[2 * _1MB];
```

```
        allocation3 = new byte[2 * _1MB];
        allocation1 = null;
        allocation4 = new byte[2 * _1MB];
        allocation5 = new byte[2 * _1MB];
        allocation6 = new byte[2 * _1MB];
        allocation4 = null;
        allocation5 = null;
        allocation6 = null;
        allocation7 = new byte[2 * _1MB];
}
```

以 HandlePromotionFailure = false 的参数设置来运行的结果：

```
[GC [DefNew: 6651K->148K(9216K), 0.0078936 secs] 6651K->4244K(19456K), 0.0079192 secs]
[Times: user=0.00 sys=0.02, real=0.02 secs]
[GC [DefNew: 6378K->6378K(9216K), 0.0000206 secs][Tenured: 4096K->4244K(10240K),
0.0042901 secs] 10474K->4244K(19456K), [Perm : 2104K->2104K(12288K)], 0.0043613 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
```

以 MaxTenuringThreshold= true 的参数设置来运行的结果：

```
[GC [DefNew: 6651K->148K(9216K), 0.0054913 secs] 6651K->4244K(19456K), 0.0055327 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 6378K->148K(9216K), 0.0006584 secs] 10474K->4244K(19456K), 0.0006857 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
```

3.6 本章小结

本章介绍了垃圾收集的算法、几款 JDK 1.6 中提供的垃圾收集器特点及其运作原理。通过代码实例验证了 Java 虚拟机中自动内存分配及回收的主要规则。

内存回收与垃圾收集器在很多时候都是影响系统性能、并发能力的主要因素之一，虚拟机之所以提供多种不同的收集器及大量的调节参数，是因为只有根据实际应用需求、实现方式选择最优的收集方式才能获取最好的性能。没有固定收集器、参数组合，也没有最优的调优方法，虚拟机也没有什么必然的内存回收行为。因此学习虚拟机内存知识，如果要到实践调优阶段，必须了解每个具体收集器的行为、优势和劣势、调节参数。在接下来的两章中，作者将会介绍内存分析的工具和调优的一些具体案例。