HZ BOOKS
华章教育

计 算 机 科 学 丛 书

CRC Press
Taylor & Francis Group

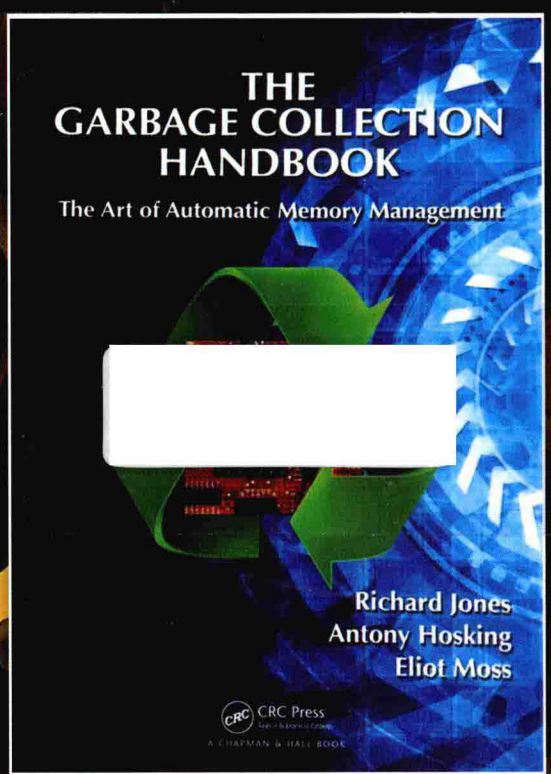
垃圾回收算法手册

自动内存管理的艺术

[英] 理查德·琼斯（Richard Jones）
[美] 安东尼·霍思金（Antony Hosking）著 王雅光 薛迪 译
艾略特·莫斯（Eliot Moss）

The Garbage Collection Handbook

The Art of Automatic Memory Management



机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

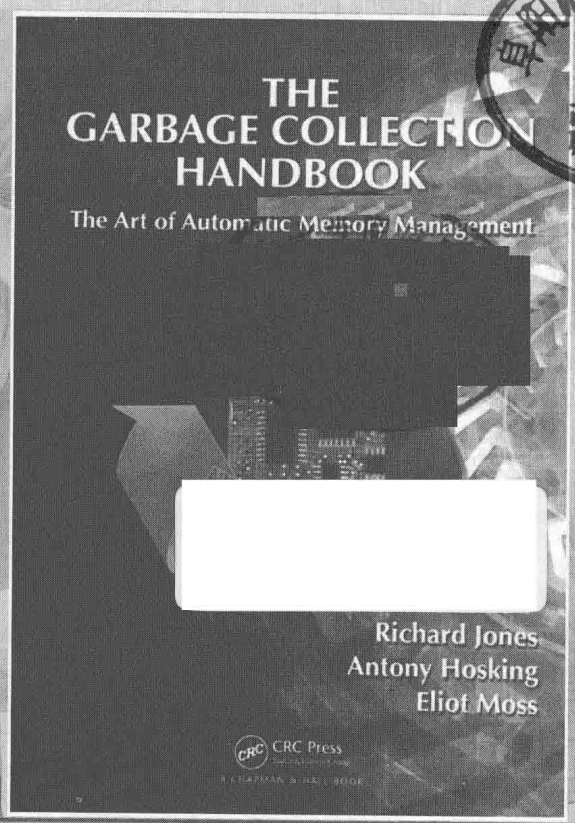
垃圾回收算法手册

自动内存管理的艺术

[英] 理查德·琼斯（Richard Jones）
[美] 安东尼·霍思金（Antony Hosking）著 王雅光 薛迪 译
艾略特·莫斯（Eliot Moss）

The Garbage Collection Handbook

The Art of Automatic Memory Management



机械工业出版社
China Machine Press

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037

华章科技图书出版中心

译者序

The Garbage Collection Handbook: the Art of Automatic Memory Management

垃圾回收技术给编程所带来的好处是不言而喻的，它能够从根本上解决软件开发过程中的内存管理问题，大大提升开发效率。但是目前，垃圾回收领域的书籍却非常少。

本书可以说是垃圾回收领域排名第二的经典著作之一，排名第一的也出自同一作者之手。1996 年，Jones 等的 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 一书出版，并于 2004 年译成中文。回顾 1996 年，垃圾回收技术的大范围应用才刚刚起步——C++ 正称霸着软件开发领域，Java 语言才推出一年之久，Anders Hejlsberg（C# 之父，.NET 的创立者）刚刚加入微软公司。近 20 年过去了，垃圾回收技术早已在各种编程语言中遍地开花，而且几乎成为每种新诞生语言的标配。与 1996 年的 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 一书相比，本书不仅在内容上更加丰富，而且更加注重充分利用近 20 年来硬件发展所带来的机遇与挑战。

本书从最基础的垃圾回收算法出发，进一步介绍了到目前为止已经十分成熟的工业级垃圾回收技术实现（例如分代回收机制），这些内容基本上算是垃圾回收领域的“经典”内容。与此同时，面对多核技术的发展以及并行程序的普及，本书使用接近一半的篇幅介绍了如何充分利用多处理器的能力来实现垃圾回收（例如并行回收、并发回收等），相关技术大都是近些年才研发出来的新成果，代表着垃圾回收领域最先进的发展方向。本书最后进一步介绍了垃圾回收技术在实时系统领域的最新研究成果。

对于开发人员而言，在享受垃圾回收机制所带来便利的同时，是否曾想过隐藏在它背后的秘密？在进行技术选型时，如何评估垃圾回收对性能可能造成的影响？面对编程语言所提供的种类繁多的垃圾回收相关参数，应当如何进行配置与调优？通过本书，开发人员能够更加深入地了解垃圾回收方面的相关问题、不同回收器的工作模式。对于研究生以及大学生而言，如果他们对编程语言的垃圾回收机制的技术实现感兴趣，本书将是不二之选。

最后，我要感谢那些在翻译过程中给予我帮助与支持的人。首先要感谢《Java 虚拟机规范（Java SE 7 版）》的译者薛迪将我引入技术书籍翻译这个领域，并在翻译过程中指出我的许多不足。同时还要感谢机械工业出版社的吴怡和张梦玲编辑对我的帮助，以及对我延迟交稿的包容。最后要特别感谢的是我的妻子，在这一年多的时间里，翻译工作让我没有太多的时间陪她，在此致以深深的歉意。

王雅光

2015 年 12 月

1960 年，McCarthy 和 Collins 发表了第一篇有关自动动态内存管理（即垃圾回收）的论文。弹指一挥间，50 多年后的今天，本书也已截稿。垃圾回收机制诞生于 Lisp 程序语言，无巧不成书，Lisp 语言诞生于 1958 年，在其 40 周年之际，第一届国际内存管理研讨会（International Symposium on Memory Management）于 1998 年 10 月举办，而本书开始写作的时间也恰逢此次会议召开 10 周年。McCathy[1978] 回忆他在麻省理工学院工业联络研讨会上第一次现场演示 Lisp 语言时的情形，他们本想给观众留下良好的第一印象，但不幸的是，IBM 704[⊖]在演示的中途就耗尽了全部的 32KB 内存空间，电传打字机以每秒十个字符的速度输出

```
THE GARBAGE COLLECTOR HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS
FOLLOWS:
```

以及其他一些更加冗长的错误信息，这一问题几乎占据了当时的整个演示时间，于是 McCarthy 的项目小组不得不省略刷新 Lisp 核心映像的相关内容，并在观众的笑声中无奈地结束演示。50 多年后的今天，垃圾回收早已不再是一个笑话，反而已经成为现代编程语言实现的关键组成部分之一。对于所有诞生于 1990 年之后且得到广泛应用的编程语言，Visual Basic（出现于 1991 年）是其中唯一一个没有采用自动内存管理的语言，但是其现代版本 VB.NET（出现于 2002 年）却依赖于具备垃圾回收能力的微软公共语言运行时（Microsoft common language runtime）。

垃圾回收给软件开发带来的收益不胜枚举。它可以消除开发过程中的几大类错误，例如尝试对悬挂指针（即指向已经回收或错误甚至被重新分配出去的内存）进行解引用，或者对已经释放的内存进行二次释放。尽管其不能保证完全消除内存泄漏问题，但也能大幅减少该问题的出现几率，还能够大幅简化并发数据结构的构建和使用 [Herlihy and Shavit, 2008]。综上所述，开发者能够基于垃圾回收所提供的抽象能力进行更好的软件工程实践。它简化了用户接口，使得代码更加容易理解和维护，进而更加可靠。由于用户接口不再需要关注内存管理，所以提升了代码的可复用性。

在过去的数年中，内存管理技术在软件和硬件方面都取得了长足进步。1996 年，典型的 Intel 奔腾处理器的时钟速度只有 120MHz，就连基于 Digital 的 Alpha 芯片的高端工作站主频也只有 266MHz。而在今天，主频达到 3GHz 以上的高端处理器以及多核芯片已经非常普遍，主存空间也几乎取得了 1000 倍的增长，普通台式计算机的内存大小已经从最初的几兆字节扩展到了 4GB。尽管如此，DRAM 内存的性能提升速度依然赶不上处理器的主频增长速度。我们曾在 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 中指出，“垃圾回收是能够解决所有内存管理问题的灵丹妙药”，并特别指出“垃圾回收机制尚无法应用于

⊖ IBM 704 为 Lisp 贡献了 car 和 cdr 这两个概念，并沿用至今。在 IBM 704 中，每个 36 位的字都包含一个地址部分（address part）以及一个减量部分（decrement part），它们均占用 15 位。Lisp 的链表或者 cons 单元将指针存储在这两部分中。链表头部或者 cons 单元的第一个元素可以使用 IBM 704 的 car（contents of the address part of register）指令获取；相应地，链表尾部或者 cons 单元的第二元素可以使用 cdr（contents of the decrement part of register）指令获取。

硬实时系统（即系统必须在给定时限内对事件做出响应）”。但时至今日，硬实时垃圾回收器已经走出实验室并进入到商业应用领域。尽管现代垃圾回收器已经解决了大多数内存管理问题，但新硬件、新环境以及新应用的出现仍会在内存管理领域不断抛出新的问题与挑战。

致读者

本书试图将过去 50 多年间学者和开发者们在自动内存管理领域所积累的丰富经验加以总结。所涉文献数量庞大，在写作期间我们的在线资源库收集了多达 2500 条记录。在描述最重要的实现策略以及代表最先进水平的实现技术时，我们尽量在一个统一的、易于接受的框架内进行讨论与比较。我们特别注意使用统一的风格和术语来介绍相关的算法与概念，同时辅以伪代码和插图来描述具体细节。对于关乎性能的部分，我们特别注意对底层细节的描述，例如同步操作原语的选择、硬件组件（如高速缓存）对算法设计的影响。

在过去的 10 年间，硬件和软件设施的发展给垃圾回收领域带来了许多新的挑战。处理器和内存之间的性能差距总体在不断扩大。处理器时钟速度得到大幅增长，单个芯片上集成的处理器核心数量越来越多，使用多处理器的模块也越来越普遍。本书重点关注了这些变化对高性能垃圾回收器的设计与实现所造成的影响。由于高速缓存对性能的影响至关重要，所以垃圾回收算法必须考虑到局部性问题。越来越多的应用程序已经多线程化，且运行在多核处理器之上，因而我们应当避免内存管理器成为性能瓶颈。另外，垃圾回收器的设计应当充分利用硬件的并行能力。在 Jones[1996]⊖中，我们完全没有考虑如何使用多线程进行并行回收，只用一章的篇幅来介绍增量回收与并发回收，这在当时的书中显得格外引人注目。

本书自始至终都密切关注现代硬件所带来的机遇与限制，对局部性问题的考量将贯穿全书。我们默认应用程序可能是多线程的。尽管本书涵盖了很多更加简单、更加传统的算法，但我们还是花了全书近一半的篇幅来介绍并行回收、增量回收、并发回收以及实时回收。

我们希望本书能够帮助到对编程语言实现感兴趣的研究生、研究人员和开发人员。对于选修了编程语言、编译器构建、软件工程或操作系统方面高级课程的本科生而言，本书也会有所帮助。此外，我们希望专业开发人员能够通过本书更加深入地了解垃圾回收面临的相关问题、不同回收器的工作模式，我们相信，与具体的专业知识相结合，开发人员在面对多种垃圾回收方法时，能够更好地进行回收器的选型与配置。由于几乎所有的现代编程语言都提供了垃圾回收机制，所以全面了解这一课题对所有开发者来说都是不可或缺的。

本书结构

本书第 1 章以探讨为什么需要自动内存管理作为开篇，简要介绍了对不同垃圾回收策略进行比较的方法。该章结尾介绍了贯穿全书的抽象记法与伪代码描述方式。

接下来的 4 章详细描述了 4 种经典的垃圾回收算法，分别是标记 – 清扫算法、标记 – 整理算法、复制式回收算法以及引用计数算法。本书对这些回收算法进行了深入的研究，并特别关注了其在现代硬件设施上的实现。如果读者需要一些更加基础的介绍，可以参阅我们先前的一本书 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*（Richard Jones and Rafael Lins, Wiley, 1996）。第 6 章深入比较了第 2 ～ 5 章所介绍的回收策略与算法，评估了它们各自的优缺点以及在不同情况下的适用性。

⊖ 指的是 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*。——编辑注

内存回收策略同样依赖于内存分配策略。第 7 章介绍了多种不同的内存分配技术，并进一步探究了自动垃圾回收与显式内存管理这两种场景下分配策略的不同之处。

前 7 章假定所有堆中的对象均采用相同的管理策略，但根据许多因素可知这并非一种良好的设计策略。第 8 章讨论了为何需要将堆划分为多个不同的空间，以及如何管理这些空间；第 9 章介绍了最成功的对象管理策略之一：分代垃圾回收；第 10 章介绍了大对象的管理策略以及其他分区策略。

在构建垃圾回收器的过程中，与运行时系统其他部分的对接是最复杂的内容之一[⊖]。因此第 11 章用了一整章的篇幅来介绍运行时接口，包括指针查找、能够安全发起垃圾回收的代码位置、读写屏障等。第 12 章讨论了特定语言相关内容，包括终结机制和弱引用。

在接下来的章节中，我们将注意力集中在并发环境下。第 13 章探讨了现代硬件系统给垃圾回收器的实现者所带来的新机遇与挑战，同时介绍了同步、前进、结束、一致等问题的相关算法。第 14 章介绍如何在挂起所有应用程序线程的前提下使用多个线程进行垃圾回收。接下来的 4 章介绍了多种不同种类的并发回收器，它们均放宽了“万物静止”这一要求，其回收过程只需要给用户程序引入十分短暂的停顿。最后，第 19 章探讨了最富挑战性的课题，即垃圾回收在硬实时系统中的应用。

每一章结尾都总结了一些需要考虑的问题，其目的在于引导读者去思考自己的系统有什么样的需求，以及如何满足这些需求，这些问题不仅关乎用户程序的行为，也关乎操作系统，甚至底层硬件的形为。但这些问题并不能替代对具体章节的阅读，它们并不是描述现有解决方案，而是提供进一步研究的焦点。

本书缺少了哪些内容？我们仅仅讨论了内嵌于运行时系统的自动内存管理技术，即使编程语言指定了垃圾回收相关的规范，我们也没有深入探讨其所支持的其他内存管理机制。最明显的例子是区域（region）的应用 [Tofte and Talpin, 1994]，其在 Java 实时规范中占据着显著的地位。我们仅花费了少量的篇幅来介绍区域推断以及栈上分配技术，并且几乎没有涉及其他通过编译期分析来替代，甚至辅助垃圾回收的技术。尽管引用计数策略在 C++ 等语言中得到了广泛应用，但我们依然认为它不是在用户程序中进行自动内存管理的最佳选择。最后，我们认为，下一代计算机将采用高度非一致内存架构，并配备异构垃圾回收器（heterogeneous collector）。这方面的技术与分布式垃圾回收（distributed garbage collection）的相关性较大，但在过去的数十年间，分布式垃圾回收领域鲜有新的研究成果发表，这不得不说是一件憾事。本书没有介绍分布式垃圾回收的相关内容。

在线资源

本书相关的电子资料参见：http://www.gchandbook.org。

该网站包含了大量垃圾回收相关资源，包括本书完整的参考文献。本书末尾所列的参考文献超过了 400 条，但我们的在线数据库中有超过 2500 条垃圾回收相关文献。该数据库支持在线搜索，同时还支持 BibTeX、PostScript、PDF 格式的下载。除了相关文章、论文、书籍之外，该参考文献还包含了某些文献的摘要，对于大多数存在电子版的文献，我们还给出了相关 URL 以及 DOI 信息。

我们将持续更新本书参考文献，并将其作为一项社区服务。如果有更多文献（或者修正意见），欢迎联系 Richard（R.E.Jones@kent.ac.uk），我们将不胜感激。

⊖ 我们在 Jones[1996] 中省略了相关内容。

致谢

感谢各位同事在本书编写时所给予的各项支持，没有大家的鼓励（与压力），本书的问世可能依然遥遥无期。特别需要感谢的是 Steve Blackburn、Hans Boehm、David Bacon、Cliff Click、David Detlefs、Daniel Frampton、Robin Garner、Barry Hayes、Laurence Hellyer、Maurice Herlihy、Martin Hirzel、Tomáš Kalibera、Doug Lea、Simon Marlow、Slan Mycroft、Cosmin Oancea、Erez Petrank、Fil Pizlo、Tony Printezis、John Reppy、David Siegwart、Gil Tene 以及 Mario Wolczko，感谢诸位十分耐心地解答了我们的许多疑问，并对本书的草稿给予诸多有用的反馈意见。同时我们也在此向 1958 年以来所有致力于自动内存管理研究的计算机科学家们致敬，没有他们的努力，本书也无从而来。

此外，我们还要向 Taylor and Francis 出版社的编辑 Randi Cohen 女士表示衷心的感谢，感谢她的支持和耐心。她总是能够及时地给予我们帮助，对我们的延误也展现出了最大程度的忍耐。同时还要感谢 Elizabeth Haylett 以及英国作家协会[⊖]的帮助，并极力向各位作者推荐他们。

Richard Jones、Antony Hosking、Eliot Moss

首先我要特别感谢 Robbie，本书从开始计划到最终完成，编写时间超过了预期，耗时两年，在此期间她忍受着我无法想象的巨大压力。本书的出版都归功于她！此外，如果没有另外两位合作者无尽的热情，本书是否能够问世恐怕还是很大一个问号。Tony 和 Eliot，很高兴也非常荣幸能与这两位勤奋博学的同事一起完成此书。

Richard Jones

2002 年的夏天，Richard 和我计划为他 1996 年的 *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 续写一部新书。在这 6 年当中，垃圾回收领域诞生了很多新的工作成果，因此有必要对前书的内容进行更新。当时我们不知道，这一本书的问世会再需要 9 年的时间，由此我不得不佩服 Richard 的耐心。当设想变成具体的计划时，我们有幸邀请到 Eliot 加入本书的编写工作中，没有他的全力协助，我们现在可能还处在焦虑的工作之中。本书的前期计划与工作是 Richard 与我在 2008 年休假期间展开的，且本书的编写工作受到了来自英国工程和物理科学研究委员会以及美国国家科学基金会的支持，我们在此表示感谢。在此，还要特别感谢 Mandi 的鼓励，感谢你同意我把大量的时间花费在这个项目之上，否则我是不可能完成这项工作的。

Antony Hosking

感谢另外两位合著者邀请我参与此项已经充分构思且已拟定出版的书籍编写项目。非常荣幸能与你们一同工作，也非常感谢你们能容忍我另类的写作风格。在此还要感谢英国皇家工程学院为我 2009 年 11 月的英国之行提供支持，这在很大程度上推进了此书的完成。除此之外，还要感谢其他基金会间接资助我们参加各种会议，并给予我们面对面交流的机会。最需要感谢的是我的妻子以及女儿能包容我出差或心思不在家庭上。她们的支持是最重要的，也是我最珍惜的！

Eliot Moss

⊖ http://www.societyofauthors.org。

目　录

The Garbage Collection Handbook: the Art of Automatic Memory Management

第 1 章

The Garbage Collection Handbook: the Art of Automatic Memory Management

引　言

托管语言（managed language）以及*托管运行时系统*（managed run-time system）不仅能够提升程序的安全性，而且可以通过对操作系统和硬件架构的抽象来提升代码的灵活性，因而受到越来越多开发者的青睐。*托管代码*（managed code）的优点已经得到广泛认可[Butters，2007]。虚拟机（virtual machine）本身提供的多种服务可以减少开发者的工作量，如果编程语言是类型安全的（type-safe），并且运行时系统可以在程序加载时进行代码检查，在运行时对资源访问冲突、数组以及其他容器进行边界检查，同时提供自动内存管理能力，那么代码的安全性将更有保障。尽管“一次编译，到处运行”（write once，run anywhere）的说法有些言过其实，但虚拟机确实使跨平台程序开发变得更加简单、开发成本更低。开发者也可将主要精力专注在应用程序的逻辑上。

几乎所有的现代编程语言都使用动态内存*分配*（allocation），即允许进程在运行时分配或者释放无法在编译期确定大小的对象，且允许对象的存活时间超出创建这些对象的子程序时间[⊖]。动态分配的对象存在于*堆*（heap）中而非*栈*（stack）或者*静态区*（statically）中。所谓栈，即程序的*活动记录*（activation record）或者*栈帧*（stack frame）；静态区则是指在编译期或者链接期就可以确定范围的存储区域。堆分配是十分重要的功能，它允许开发者：

- 在运行时动态地确定新创建对象的大小（从而避免程序在运行时遭遇硬编码数组长度不足产生的失败）。
- 定义和使用具有递归特征的数据结构，例如链表（list）、树（tree）和映射（map）。
- 向父过程返回新创建的对象，例如工厂方法。
- 将一个函数作为另一个函数的返回值，例如函数式语言中的*闭包*（closure）或者*悬挂*（suspension）[⊖]。

堆中分配的对象需要通过*引用*（reference）进行访问。一般情况下，引用即指向对象的*指针*（pointer）（也就是对象在内存中的地址）。引用也可以通过间接方式实现，例如它可以是一个间接指向对象的*句柄*（handle）。使用句柄的好处在于，当迁移某一对象时，可以仅修改对象的句柄而避免通过程序改变这个对象或句柄的所有引用。

1.1　显式内存释放

任何一个运行在有限内存环境下的程序都需要不时地回收运行过程中不再需要的对象。堆中对象使用的内存可以使用*显式释放*（explicit deallocation）策略（例如 C 语言的 `free` 函数或者 C++ 的 `delete` 操作符）、基于引用计数的运行时系统 [Collins，1960]、追踪式垃圾回收器 [McCarthy，1960] 进行回收。显式内存释放会在两个方面增加程序出错的风险。

一方面，开发者可能过早地回收依然在引用的对象，这种情况将引发*悬挂指针*（dangling pointer）问题（如图 1.1 所示）。正常情况下，开发者将不会访问已经释放的内存，

⊖ 本书中，*方法*（method）、*函数*（function）、*过程*（procedure）、*子程序*（subroutine）这几个概念等价。

⊖ 即待计算的表达式。——译者注

所以运行时系统可以将其清空（填充0），也可以将其重新分配出去，甚至可以将其归还给操作系统。因此，如果程序对悬挂指针进行访问，那么执行结果将是不可预知的。此时开发者最期望的结果是程序立即崩溃，但更常见的情况却是程序会在崩溃前继续运行一段时间（导致调试困难），或者一直运行下去但输出错误的结果（甚至这些错误很难被检测到）。检测悬挂指针的一种策略是使用*肥指针*（fat pointer），它将指针目标对象的版本号作为指针本身的一部分。当对肥指针进行解引用时，运行系统会首先判断肥指针中记录的版本号与其目标对象的版本号是否一致。这一方法存在额外开销，而且并非完全可靠，因此其应用范围几乎局限于调试工具上[⊖]。

另一方面，开发者很可能在程序将对象使用完毕之后未将对象释放，从而导致*内存泄漏*（memory leak）现象的出现。在小程序中，内存泄漏可能不会造成太大影响，但对于较大的程序，内存泄漏却有可能导致显著的性能下降（即内存管理器难以满足新的内存分配需求），甚至是崩溃（即程序的内存被耗光）。一次错误的内存释放通常不仅会导致悬挂指针的出现，而且会引发内存泄漏（如图1.1所示）。

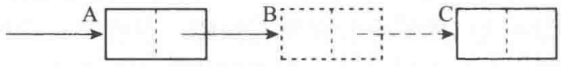


图1.1 过早地删除对象可能引发错误。此处对象B被错误地释放，从而导致对象A中产生悬挂指针，且对象C所占用的空间发生泄漏，即对象C不可达，但无法将其释放

上述两种错误在对象共享的情况下尤为普遍，即当两个或更多子过程持有同一个对象的引用时。在并发编程情况下，当两个或多个线程引用同一个对象时问题将更加严重。随着多核处理器的普及，开发者需要投入相当多的精力来构建线程安全的数据结构库。实现线程安全的数据结构库的算法必须面临一系列问题，例如*死锁*（deadlock）、*活锁*（livelock）、*ABA问题*（ABA problem）[⊖]。自动内存管理可以显著降低这些问题的解决难度（例如可以消除某些情况下的ABA问题），否则编程方案将十分复杂[Herlihy and Shavit，2008]。

更为根本的问题在于，显式释放需要花费开发者更多的精力。如何正确进行内存管理往往是编程中的固有难题[⊜]，正如Wilson[1994]所指出的“存活性是一个*全局*（global）特征”，但是调用`free`函数将对象释放却是局部行为，所以如何将对象正确地释放是一个十分复杂的问题。

2

在不支持自动化动态内存管理的语言中，众多研究者已经付出了相当大的努力来解决这一难题，其方法主要是管理对象的*所有权*（ownership）[Belotsky，2003；Cline，Lomow，1995]。Belotsky[2003]等人针对C++提出了几个可行策略：第一，开发者在任何情况下都应当避免堆分配，例如可以将对象分配在栈上，当创建对象的函数返回之后，栈的弹出（pop）操作会自动将对象释放。第二，在传递参数与返回值时，应尽量传值而非引用。尽管这些方法避免了分配、释放错误，但其不仅会造成内存方面的压力，而且失去了对象共享的能力。另外，开发者也可以在一些特殊场景下使用自定义内存分配器，例如*对象池*（pool of object），在程序的一个阶段完成之后，池中的对象将作为一个整体全部释放。

C++语言尝试通过特殊的指针类和模板来改善内存管理。此类方法通过对指针操作进行*重载*（overload）来提升内存回收的安全性，但是这些*智能指针*（smart pointer）通常存在

⊖ 开源工具valgrind（http://valgrind.org）框架中使用的内存泄漏检测工具memcheck虽然速度较慢，但是更加可靠。另外还有很多商业化的内存调试工具。

⊖ ABA问题：一个变量的初始值是A，被改写为B，然后再次被改写为A（参见第13章）。

⊜ “如果你手里拿着C++这把锤子，那么所有东西看起来都像钉子”，Steven M. Haflich，Common Lisp ANSI标准NCITS/J13技术委员会主席。

一些限制。auto_ptr 与标准库不兼容，并且即将在下一个 C++ 标准版本中废弃 [Boehm, Spertus, 2009]⊖，取而代之的是更加先进的 unique_ptr，它将提供严格的所有权控制语义，即在该指针被摧毁时，其目标对象将自动得到释放。新标准也将提供一个基于引用计数的 shared_ptr⊖，但其仍存在一定限制，即引用计数指针无法处理自引用（环状引用）数据结构。大多数智能指针都是以库的形式提供，因此当需要关注性能时，其适用范围可能会受到限制。智能指针可能只适用于管理数据块非常大、引用关系变更较少的场景，因为只有在这种情况下，智能指针的开销才有可能远小于追踪式垃圾回收。另外，在没有编译器和运行时系统支持的情况下，基于引用计数的智能指针算不上是一种高效的、通用的小对象管理策略，特别是在指针操作非线程安全的情况下。

若安全地进行手动内存管理有过多的方法也会引发另一个问题：如果开发者始终需要考虑对象的所有权管理问题，那么他究竟应当使用哪种方法？当使用第三方代码库时，这一问题将更加严重。例如，第三方代码库使用哪种方法进行内存管理？所有的第三方库是否都使用同一种方法？

1.2　自动动态内存管理

自动动态内存管理可以解决大多数悬挂指针和内存泄漏问题。垃圾回收（garbage collection，GC）可以将未被任何可达对象引用的对象回收，从而避免悬挂指针的出现。原则上讲，回收器最终都会将所有不可达对象回收，但是有两个注意事项：第一，追踪式回收（tracing collection）引入“垃圾”这一具有明确判定标准的概念，但它不一定包含所有不再使用的对象；第二，后面章节将要描述，在实际情况下，出于效率原因，某些对象可能不会被回收。只有回收器可以释放对象，所以不会出现二次释放（double-freeing）问题。回收器掌握堆中对象的全局信息以及所有可能访问堆中对象的线程信息，因而其可以决定任意对象是否需要回收。显式释放的主要问题在于其无法在局部上下文中掌握全局信息，而自动动态内存管理则简单地解决了这一问题。

总之，内存管理是一个软件工程学问题。设计良好的程序应当是由一系列“高内聚，低耦合”的组件（从广义上来讲）构建而成。“高内聚”可以简化程序的维护，即在理想情况下，开发者如需理解一个模块，只需要关注该模块本身的代码或者其他相关模块的少量代码；“低耦合”意味着一个模块不依赖其他模块的实现。在这种情况下，如果要考虑正确的内存管理，就意味着一个模块要了解其他模块的内存管理规则。相比之下，显式内存管理显然不能满足软件工程学中“低耦合”的原则，它需要引入一些额外的接口，例如需要显式传递额外参数来协商对象的所有权，或者隐式要求开发者遵从一些特定惯例，这些要求都限制了模块的可复用性。

垃圾回收可以带来的好处绝不仅仅是简化编码，它同时可以解除模块之间内存管理层次的耦合，且不需要额外的内存管理接口，从而提升了模块的可复用性。正是由于这些原因，垃圾回收机制几乎成为现代编程语言的标配（如表 1.1 所示），甚至下一代 C++ 标准都有可能引入垃圾回收机制⊜[Boehm and Spertus, 2009]。大量证据表明，包含自动内存管理机制

⊖ 当前，下一代符合 ISO C++ 标准的最终提案是 C++0x（该标准最终在 2011 年通过，被称为 C++11）。——译者注

⊖ 见 http://boost.org。

⊜ 作者这里提到的“下一代”其实就是 C++ 11 标准，但该标准最终并未引入垃圾回收机制。——译者注

的托管代码可以降低开发成本 [Butters，2007]，但不幸的是，这些证据大多是未经证实的，或者只是不同语言、不同系统之间的比较（因此比较结果可能受到内存管理策略之外因素的影响），更加详细的比较研究则少有表明这一观点。还有人建议将垃圾回收机制作为复杂系统中软件设计的主要关注点 [Nagle，1995]。Rovner[1985] 估计，在施乐公司 Mesa 系统的开发过程中，40% 的开发时间花费在了内存管理相关问题的调试上。或许，层出不穷的内存错误检测工具能够在经济方面间接地给予自动内存管理最有力的支持。

表 1.1 现代编程语言与垃圾回收（这些语言都是基于垃圾回收的）

ActionScript（2000 年）	AppleScript（1993 年）	Beta（1983 年）
Managed C++（2002 年）	Clean（1984 年）	Dylan（1992 年）
Eiffel（1986 年）	Erlang（1990 年）	Fortress（2006 年）
Groovy（2004 年）	Icon（1977 年）	Liana（1991 年）
Lisp（1958 年）	LotusScript（1995 年）	MATLAB（20 世纪 70 年代）
ML（1990 年）	Objective-C（2007 ～至今）	Pike（1996 年）
POP-2（1970 年）	Python（1991 年）	Sather（1990 年）
Self（1986 年）	SISAL（1983 年）	Squeak（1996 年）
VB.NET（2001 年）	VHDL（1987 年）	Algol-68（1965 年）
AspectJ（2001 年）	C#（1999 年）	Cecil（1992 年）
CLU（1974 年）	Dynace（1993 年）	Elasti-C（1997 年）
Euphoria（1993 年）	Green（1998 年）	Haskell（1990 年）
Java（1994 年）	Limbo（1996 年）	Lua（1994 年）
Mercury（1993 年）	Modula-3（1988 年）	Obliq（1993 年）
PHP（1995 年）	PostScript（1982 年）	Rexx（1979 年）
Scala（2003 年）	SETL（1969 年）	Smalltalk（1972 年）
Tel（1990 年）	VBScript（1996 年）	X10（2004 年）
APL（1964 年）	Awk（1977 年）	Cyclone（2006 年）
Cedar（1983 年）	D（2007 年）	E（1997 年）
Emerald（1988 年）	F#（2005 年）	Go（2010 年）
Hope（1978 年）	JavaScript（1994 年）	Lingo（1991 年）
Mathematica（1987 年）	Miranda（1985 年）	Oberon（1985 年）
Perl（1986 年）	Pliant（1999 年）	Prolog（1972 年）
Ruby（1993 年）	Scheme（1975 年）	Simula（1964 年）
SNOBOL（1962 年）	Theta（1994 年）	Visual Basic（1991 年）
YAFL（1993 年）		

诚然，并不是说垃圾回收是根除所有与内存相关的编程错误且适用于所有场景的银弹（silver bullet）。内存泄漏是最普遍的一种内存错误，尽管垃圾回收可以减少内存泄漏现象的出现，但也不能保证完全根除。如果某个对象在程序未来的运行过程中不可达（例如，从已知根集合开始通过任何指针链都不可达），则回收器会将其回收，这是删除对象的唯一方法，因此悬挂指针不会出现；如果删除某个对象导致其子对象也不可达，则它也将得到回收，所以图 1.1 所示的任何一种情况都不会出现。但是，垃圾回收仍不能确保根除内存泄漏，对于那种一直可达但无限增长的数据结构（例如不停地将数据添加到缓冲区中，但却不从中移除任何对象），或者一直可达但永远不会再用到的对象，垃圾回收也无能为力。

自动动态内存管理的设计目标仅限于其字面所表达的内容。一些评论者抱怨垃圾回收器不能提供通用的资源管理功能，例如关闭不再使用的文件或者窗口，但这一批评有失公允：