垃圾收集简史

**简介：** Java 语言可能是使用最广泛的依赖于垃圾收集的编程语言，但是它并不是第一个。垃圾收集已经成为了包括 Lisp、Smalltalk、Eiffel、Haskell、ML、Scheme和 Modula-3 在内的许多编程语言的一个集成部分，并且从 20 世纪 60 年代早期就开始使用了。在 Java 理论与实践的本篇文章中，Brian Goetz 描述了垃圾收集最常用的技术。在以后的几个月，他将分析 1.4 JVM 所使用的垃圾收集策略、不同垃圾收集策略对性能的影响和如何才能（以及如何 不能）帮助垃圾收集器取得更好的性能。

垃圾收集的好处是无可争辩的 ―― 可靠性提高、使内存管理与类接口设计分离，并使开发者减少了跟踪内存管理错误的时间。著名的悬空指针和内存泄漏问题在 Java 程序中再也不会发生了（Java 程序可能会出现某种形式的内存泄漏，更精确地说是非故意的对象保留，但是这是一个不同的问题）。不过，垃圾收集不是没有代价的 ―― 其中包括对性能的影响、暂停、配置复杂性和不确定的结束 (nondeterministic finalization)。

一个理想的垃圾收集实现应该是完全不可见的 ―― 没有垃圾收集暂停、没有因为垃圾收集而产生的 CPU 时间损失、垃圾收集器不会与虚拟内存或者缓存有负面的互动，并且堆不需要大于应用程序的 *驻留空间*（即堆占用）。当然，没有十全十美的垃圾收集器，但是垃圾收集器在过去十年中已经有了很大改进。

**选项与选择**

1.3 JDK 包括三种不同的垃圾收集策略，1.4.1 JDK 包括六种垃圾收集策略以及 12 种以上用于配置和优化垃圾收集的命令行选项。它们有什么不同？为什么需要有这么多选项？

不同的垃圾收集实现使用不同的策略来识别和收回不可到达的对象，它们与用户程序和调度器以不同的方式互动。不同类型的应用程序对于垃圾收集有不同的要求 ―― 实时应用程序会将要求收集暂停的持续时间短并且有限制，而企业应用程序可能允许更长时间和可预测性更低的暂停以获得更高的吞吐能力。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp10283/#ibm-pcon)

**垃圾收集如何工作？**

有几种垃圾收集的基本策略：引用计数、标记-清除、标记-整理 (mark-compact) 和复制。此外，一些算法可以以 *增量* 方式完成它们的工作（不需要一次收集整个堆，使得收集暂停时间更短），一些算法可以在用户程序运行时运行（ *并发*收集）。其他算法则必须在用户程序暂停时一次进行整个收集（即所谓的 *stop-the-world*收集器）。最后，还有混合型的收集器，如 1.2 和以后版本的 JDK 使用的分代收集器，它对堆的不同区域使用不同的收集算法。

在对垃圾收集算法进行评价时，我们可能要考虑以下所有标准：

* **暂停时间。**收集器是否停止所有工作来进行垃圾收集？要停止多长时间？暂停是否有时间限制？
* **暂停的可预测性。**垃圾收集暂停是否规划为在用户程序方便而不是垃圾收集器方便的时间发生？
* **CPU 占用。**总的可用 CPU 时间用在垃圾收集上的百分比是多少？
* **内存大小。**许多垃圾收集算法需要将堆分割成独立的内存空间，其中一些空间在某些时刻对用户程序是不可访问的。这意味着堆的实际大小可能比用户程序的最大堆驻留空间要大几倍。
* **虚拟内存交互。**在具有有限物理内存的系统上，一个完整的垃圾收集在垃圾收集过程中可能会错误地将非常驻页面放到内存中来进行检查。因为页面错误的成本很高，所以垃圾收集器正确管理引用的区域性 (locality) 是很必要的。
* **缓存交互。**即使在整个堆可以放到主内存中的系统上 ―― 实际上几乎所有 Java 应用程序都可以做到这一点，垃圾收集也常常会有将用户程序使用的数据冲出缓存的效果，从而影响用户程序的性能。
* **对程序区域性的影响。**虽然一些人认为垃圾收集器的工作只是收回不可到达的内存，但是其他人认为垃圾收集器还应该尽量改进用户程序的引用区域性。整理收集器和复制收集器在收集过程中重新安排对象，这有可能改进区域性。
* **编译器和运行时影响。**一些垃圾收集算法要求编译器或者运行时环境的重要配合，如当进行指针分配时更新引用计数。这增加了编译器的工作，因为它必须生成这些簿记指令，同时增加了运行时环境的开销，因为它必须执行这些额外的指令。这些要求对性能有什么影响呢？它是否会干扰编译时优化呢？

不管选择什么算法，硬件和软件的发展使垃圾收集更具有实用性。20 世纪 70 和 80 年代的经验研究表明，对于大型 Lisp 程序，垃圾收集消耗 25％ 到 40％ 的运行时。垃圾收集还不能做到完全不可见，这肯定还有很长的路要走。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp10283/#ibm-pcon)

**基本算法**

所有垃圾收集算法所面临的问题是相同的 ―― 找出由分配器分配的，但是用户程序不可到达的内存块。不可到达是什么意思？可以以两种方式之一访问内存块 ―― 或者用户程序在 *根 (root)*中有对这一内存块的引用，或者在另一个可到达的块中有对这个块的引用。在 Java 程序中，根是对静态变量中或者活跃的堆栈框架上的本地变量中所包含的对象的引用。可到达的对象集是指向关系下根集的传递闭包。

**引用计数**

最直观的垃圾收集策略是引用计数。引用计数很简单，但是需要编译器的重要配合，并且增加了赋值 *函数 (mutator)*的开销（这个术语是针对用户程序的，是从垃圾收集器的角度来看的）。每一个对象都有一个关联的引用计数 ―― 对该对象的活跃引用的数量。如果对象的引用计数是零，那么它就是垃圾（用户程序不可到达它），并可以回收。每次修改指针引用时（比如通过赋值语句），或者当引用超出范围时，编译器必须生成代码以更新引用的对象的引用计数。如果对象的引用计数变为零，那么运行时就可以立即收回这个块（并且减少被回收的块所引用的所有块的引用计数），或者将它放到迟延收集队列中。

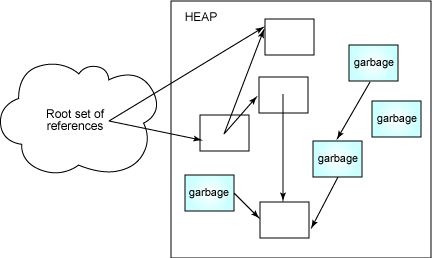
许多 ANSI C++ 库类，比如 string ，使用了引用计数来提供垃圾收集的特性。通过重载赋值操作符并利用 C++ 作用域提供的确定性结束，C++ 程序可以将 string 类当成是被收集的垃圾那样使用。引用计数很简单，很适用于增量收集，收集过程一般会得到好的引用区域性，但是出于几个理由，它很少在生产垃圾收集器中使用，如它不能回收不可到达的循环结构（彼此直接或者间接引用的几个对象，如循环链接的列表或者包含指向父节点的反向指针的树）。

**跟踪收集器**

JDK 中的标准垃圾收集器都没有使用引用计数，相反，它们都使用某种形式的 *跟踪收集器 (tracing collector)*。跟踪收集器停止所有工作（尽管不需要在收集的整个过程中都这样）并开始跟踪对象，从根集开始沿着引用跟踪，直到检查了所有可到达的对象。可以在程序注册表中、每一个线程堆栈中的（基于堆栈的）局部变量中以及静态变量中找到根。

**标记-清除收集器**

最早由 Lisp 的发明人 John McCarthy 于 1960 年提出的最基本的跟踪收集器形式是 *标记―清除*收集器，它停止所有工作，收集器从根开始访问每一个活跃的节点，标记它所访问的每一个节点。走过所有引用后，收集就完成了，然后就对堆进行清除（即对堆中的每一个对象进行检查），所有没有标记的对象都作为垃圾回收并返回空闲列表。图 1 展示了垃圾收集之前的堆，阴影块是垃圾，因为用户程序不能到达它们：

**图 1. 可到达和不可到达的对象**  
 

标记-清除实现起来很简单，可以容易地回收循环的结构，并且不像引用计数那样增加编译器或者赋值函数的负担。但是它也有不足 ―― 收集暂停可能会很长，在清除阶段整个堆都是可访问的，这对于可能有页面交换的堆的虚拟内存系统有非常负面的性能影响。

标记-清除的最大问题是，每一个活跃的（即已分配的）对象，不管是不是可到达的，在清除阶段都是可以访问的。因为很多对象都可能成为垃圾，这意思着收集器花费大量精力去检查并处理垃圾。标记-清除收集器还容易使堆产生碎片，这会产生区域性问题并可以造成分配失败，即使看来有足够的自由内存可用。

**复制收集器**

在另一种形式的跟踪收集器 ―― *复制收集器*中，堆被分成两个大小相等的半空间，其中一个包含活跃的数据，另一个未使用。当活跃的空间占满以后，程序就会停止，活跃的对象被从活跃的空间复制到不活跃的空间中。空间的角色就会转换，原来不活跃的空间成为了新的活跃空间。

复制收集的优点是只访问活跃的对象，这意味着不会检查垃圾对象，也不需要将它们页交换到内存中或者送到缓存中。复制收集器的收集周期时间是由活跃对象的数量决定的。不过，复制收集器因为要将数据从一个空间复制到另一个空间、调整所有引用以指向新备份而增加了成本。特别是，长寿的对象在每次收集时都要来回复制。

**堆整理**

复制收集器有另一个好处，活跃对象集会被整理到堆的底部。这不仅改进了用户程序的引用区域性并消除了堆碎片，而且极大地减少了对象分配的成本 ―― 对象分配变成了在堆顶部的指针上增加指针。不需要维护自由列表或者后备列表，或者使用性能最佳或者第一合适的算法 ―― 分配 N 字节就是在堆顶部指针上加 N 并返回前一个值这么简单，如清单 1 所示：

**清单 1. 复制收集器中廉价的内存分配**

|  |
| --- |
| void \*malloc(int n) {  if (heapTop - heapStart < n)  doGarbageCollection();  void \*wasStart = heapStart;  heapStart += n;  return wasStart;  } |

为非垃圾收集语言实现了复杂内存管理方案的开发人员可能会对复制收集器中廉价的内存分配感到吃惊 ―― 就是指针加法这么简单。以前的 JVM 实现没有使用复制收集器 ―― 这可能是对象分配是昂贵的这一想法是如此普遍的原因之一，开发人员仍然下意识地假设分配成本与其他语言（如 C）类似，而事实上在 Java 运行时中可能要廉价得多。不但是分配成本减少了，而且对于在下次收集之前成为垃圾的对象，解除分配的成本为零，因为既不会访问也不会复制垃圾对象。

**标记-整理收集器**

复制算法的性能很优异，但是它有一个缺点是需要两倍于标记-清除收集器所需要的内存。 *标记-整理* 算法结合了标记-清除和复制，避免了这个问题，代价是增加了一些收集复杂性。与标记-清除类似，标记-整理是两阶段过程，在标记阶段访问并标记每个活跃对象。然后，复制标记的对象，使所有活跃对象被整理到堆的底部。如果每一次收集时进行彻底的整理，那么得到的堆就类似于复制收集器的结果 ―― 在堆的活跃部分与自由部分有明确的界线，这样分配成本与复制收集器相当。长寿的对象趋向于沉在堆的底部，这样就不会像在复制收集器中那样反复复制它们。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp10283/#ibm-pcon)

**选择哪一种呢？**

那么 JDK 使用了哪种方式进行垃圾收集呢？在某种意义上，使用了所有的方式。早期的 JDK 使用了单线程的标记-清除或者标记-清除-整理收集器。1.2 及以后的 JDK 使用了混合的方式，称为 *分代收集*，其中根据对象的年龄将堆分为几个部分，不同的代是用不同的收集算法收集的。

分代收集证明是非常高效的，尽管在运行时它需要更多的簿记。在下一个月的 *Java 理论与实践* 中，除了介绍 1.4.1 JVM 提供的所有其他垃圾收集选项之外，我们还将探讨分代收集是如何工作的以及 1.4.1 JVM 是如何使用它的。在下下篇文章中，我们将分析垃圾收集对性能的影响，包括揭示与内存管理有关的性能神话。