附录 E 关于垃圾收集的一些话

“很难相信 Java 居然能和 C++一样快，甚至还能更快一些。”

据我自己的实践，这种说法确实成立。然而，我也发现许多关于速度的怀疑都来自一些早期的实现方式。由于这些方式并非特别有效，所以没有一个模型可供参考，不能解释 Java 速度快的原因。

我之所以想到速度，部分原因是由于 C++模型。C++将自己的主要精力放在编译期间“静态”发生的所有事情上，所以程序的运行期版本非常短小和快速。C++也直接建立在 C 模型的基础上（主要为了向后兼容），但有时仅仅由于它在 C 中能按特定的方式工作，所以也是 C++中最方便的一种方法。最重要的一种情况是 C 和 C++对内存的管理方式，它是某些人觉得 Java 速度肯定慢的重要依据：在 Java 中，所有对象都必须在内存“堆”里创建。

而在 C++中，对象是在堆栈中创建的。这样可达到更快的速度，因为当我们进入一个特定的作用域时，堆栈指针会向下移动一个单位，为那个作用域内创建的、以堆栈为基础的所有对象分配存储空间。而当我们离开作用域的时候（调用完毕所有局部构建器后），堆栈指针会向上移动一个单位。然而，在 C++里创建“内存堆”（Heap）对象通常会慢得多，因为它建立在 C 的内存堆基础上。这种内存堆实际是一个大的内存池，要求必须进行再循环（再生）。在 C++里调用 delete 以后，释放的内存会在堆里留下一个洞，所以再调用 new 的时候，存储分配机制必须进行某种形式的搜索，使对象的存储与堆内任何现成的洞相配，否则就会很快用光堆的存储空间。之所以内存堆的分配会在 C++里对性能造成如此重大的

性能影响，对可用内存的搜索正是一个重要的原因。所以创建基于堆栈的对象要快得多。

同样地，由于 C++如此多的工作都在编译期间进行，所以必须考虑这方面的因素。但在 Java 的某些地方，事情的发生却要显得“动态”得多，它会改变模型。创建对象的时候，垃圾收集器的使用对于提高对象创建的速度产生了显著的影响。从表面上看，这种说法似乎有些奇怪——存储空间的释放会对存储空间的分配造成影响，但它正是 JVM 采取的重要手段之一，这意味着在 Java 中为堆对象分配存储空间几乎能达到与 C++中在堆栈里创建存储空间一样快的速度。

可将 C++的堆（以及更慢的 Java 堆）想象成一个庭院，每个对象都拥有自己的一块地皮。在以后的某个时间，这种“不动产”会被抛弃，而且必须再生。但在某些 JVM 里，Java 堆的工作方式却是颇有不同的。它更象一条传送带：每次分配了一个新对象后，都会朝前移动。这意味着对象存储空间的分配可以达到非常快的速度。“堆指针”简单地向前移至处女地，所以它与 C++的堆栈分配方式几乎是完全相同的（当然，在数据记录上会多花一些开销，但要比搜索存储空间快多了）。

现在，大家可能注意到了堆事实并非一条传送带。如按那种方式对待它，最终就要求进行大量的页交换（这对性能的发挥会产生巨大干扰），这样终究会用光内存，出现内存分页错误。所以这儿必须采取一个技巧，那就是著名的“垃圾收集器”。它在收集“垃圾”的同时，也负责压缩堆里的所有对象，将“堆指针”移至尽可能靠近传送带开头的地方，远离发生（内存）分页错误的地点。垃圾收集器会重新安排所有东西，使其成为一个高速、无限自由的堆模型，同时游刃有余地分配存储空间。

为真正掌握它的工作原理，我们首先需要理解不同垃圾收集器（GC）采取的工作方案。一种简单、但速度较慢的 GC 技术是引用计数。这意味着每个对象都包含了一个引用计数器。每当一个句柄同一个对象连接起来时，引用计数器就会增值。每当一个句柄超出自己的作用域，或者设为 null 时，引用计数就会减值。这样一来，只要程序处于运行状态，就需要连续进行引用计数管理——尽管这种管理本身的开销比较少。垃圾收集器会在整个对象列表中移动巡视，一旦它发现其中一个引用计数成为 0，就释放它占据的存储空间。但这样做也有一个缺点：若对象相互之间进行循环引用，那么即使引用计数不是 0，仍有可能属于应收掉的“垃圾”。为了找出这种自引用的组，要求垃圾收集器进行大量额外的工作。引用计数属于垃圾收集的一种类型，但它看起来并不适合在所有 JVM 方案中采用。

在速度更快的方案里，垃圾收集并不建立在引用计数的基础上。相反，它们基于这样一个原理：所有非死锁的对象最终都肯定能回溯至一个句柄，该句柄要么存在于堆栈中，要么存在于静态存储空间。这个回溯链可能经历了几层对象。所以，如果从堆栈和静态存储区域开始，并经历所有句柄，就能找出所有活动的对象。对于自己找到的每个句柄，都必须跟踪到它指向的那个对象，然后跟随那个对象中的所有句柄，“跟踪追击”到它们指向的对象……等等，直到遍历了从堆栈或静态存储区域中的句柄发起的整个链接网路为止。中途移经的每个对象都必须仍

处于活动状态。注意对于那些特殊的自引用组，并不会出现前述的问题。由于它们根本找不到，所以会自动当作垃圾处理。

在这里阐述的方法中，JVM 采用一种“自适应”的垃圾收集方案。对于它找到的那些活动对象，具体采取的操作取决于当前正在使用的是什么变体。其中一个变体是“停止和复制”。这意味着由于一些不久之后就会非常明显的原因，程序首先会停止运行（并非一种后台收集方案）。随后，已找到的每个活动对象都会从一个内存堆复制到另一个，留下所有的垃圾。除此以外，随着对象复制到新堆，它们会一个接一个地聚焦在一起。这样可使新堆显得更加紧凑（并使新的存储区域可以简单地抽离末尾，就象前面讲述的那样）。

当然，将一个对象从一处挪到另一处时，指向那个对象的所有句柄（引用）都必须改变。对于那些通过跟踪内存堆的对象而获得的句柄，以及那些静态存储区域，都可以立即改变。但在“遍历”过程中，还有可能遇到指向这个对象的其他句柄。一旦发现这个问题，就当即进行修正（可想象一个散列表将老地址映射成新地址）。

有两方面的问题使复制收集器显得效率低下。第一个问题是我们拥有两个堆，所有内存都在这两个独立的堆内来回移动，要求付出的管理量是实际需要的两倍。为解决这个问题，有些 JVM 根据需要分配内存堆，并将一个堆简单地复制到另一个。

第二个问题是复制。随着程序变得越来越“健壮”，它几乎不产生或产生很少的垃圾。尽管如此，一个副本收集器仍会将所有内存从一处复制到另一处，这显得非常浪费。为避免这个问题，有些 JVM 能侦测是否没有产生新的垃圾，并随即改换另一种方案（这便是“自适应”的缘由）。另一种方案叫作“标记和清除”，Sun 公司的 JVM 一直采用的都是这种方案。对于常规性的应用，标记和清除显得非常慢，但一旦知道自己不产生垃圾，或者只产生很少的垃圾，它的速度就会非常快。

标记和清除采用相同的逻辑：从堆栈和静态存储区域开始，并跟踪所有句柄，寻找活动对象。然而，每次发现一个活动对象的时候，就会设置一个标记，为那个对象作上“记号”。但此时尚不收集那个对象。只有在标记过程结束，清除过程才正式开始。在清除过程中，死锁的对象会被释放然而，不会进行任何形式的复制，所以假若收集器决定压缩一个断续的内存堆，它通过移动周围的对象来实现。

“停止和复制”向我们表明这种类型的垃圾收集并不是在后台进行的；相反，一旦发生垃圾收集，程序就会停止运行。在 Sun 公司的文档库中，可发现许多地方都将垃圾收集定义成一种低优先级的后台进程，但它只是一种理论上的实验，实际根本不能工作。在实际应用中，Sun 的垃圾收集器会在内存减少时运行。除此以外，“标记和清除”也要求程序停止运行。

正如早先指出的那样，在这里介绍的 JVM 中，内存是按大块分配的。若分配一个大块头对象，它会获得自己的内存块。严格的“停止和复制”要求在释放旧堆

之前，将每个活动的对象从源堆复制到一个新堆，此时会涉及大量的内存转换工作。通过内存块，垃圾收集器通常可利用死块复制对象，就象它进行收集时那样。每个块都有一个生成计数，用于跟踪它是否依然“存活”。通常，只有自上次垃圾收集以来创建的块才会得到压缩；对于其他所有块，如果已从其他某些地方进行了引用，那么生成计数都会溢出。这是许多短期的、临时的对象经常遇到的情况。会周期性地进行一次完整清除工作——大块头的对象仍未复制（只是让它们的生成计数溢出），而那些包含了小对象的块会进行复制和压缩。JVM 会监视垃圾收集器的效率，如果由于所有对象都属于长期对象，造成垃圾收集成为浪费时间的一个过程，就会切换到“标记和清除”方案。类似地，JVM 会跟踪监视成功的“标记与清除”工作，若内存堆变得越来越“散乱”，就会换回“停止和复制”方案。“自定义”的说法就是从这种行为来的，我们将其最后总结为：“根据情况，自动转换停止和复制 / 标记和清除这两种模式”。

JVM 还采用了其他许多加速方案。其中一个特别重要的涉及装载器以及 JIT 编译器。若必须装载一个类（通常是我们首次想创建那个类的一个对象时），会找到.class 文件，并将那个类的字节码送入内存。此时，一个方法是用 JIT 编译所有代码，但这样做有两方面的缺点：它会花更多的时间，若与程序的运行时间综合考虑，编译时间还有可能更长；而且它增大了执行文件的长度（字节码比扩展过的 JIT 代码精简得多），这有可能造成内存页交换，从而显著放慢一个程序的执行速度。另一种替代办法是：除非确有必要，否则不经 JIT 编译。这样一来，那些根本不会执行的代码就可能永远得不到 JIT 的编译。

由于 JVM 对浏览器来说是外置的，大家可能希望在使用浏览器的时候从一些 JVM 的速度提高中获得好处。但非常不幸，JVM 目前不能与不同的浏览器进行沟通。为发挥一种特定 JVM 的潜力，要么使用内建了那种 JVM 的浏览器，要么只有运行独立的 Java 应用程序。