附录 D 性能

“本附录由 Joe Sharp 投稿，并获得他的同意在这儿转载。请联系 SharpJoe@aol.com”

Java 语言特别强调准确性，但可靠的行为要以性能作为代价。这一特点反映在自动收集垃圾、严格的运行期检查、完整的字节码检查以及保守的运行期同步等等方面。对一个解释型的虚拟机来说，由于目前有大量平台可供挑选，所以进一步阻碍了性能的发挥。

“先做完它，再逐步完善。幸好需要改进的地方通常不会太多。”（Steve McConnell 的《About performance》[16]）

本附录的宗旨就是指导大家寻找和优化“需要完善的那一部分”。

D.1 基本方法

只有正确和完整地检测了程序后，再可着手解决性能方面的问题：

(1) 在现实环境中检测程序的性能。若符合要求，则目标达到。若不符合，则转到下一步。

(2) 寻找最致命的性能瓶颈。这也许要求一定的技巧，但所有努力都不会白费。如简单地猜测瓶颈所在，并试图进行优化，那么可能是白花时间。

(3) 运用本附录介绍的提速技术，然后返回步骤 1。

为使努力不至白费，瓶颈的定位是至关重要的一环。Donald Knuth[9]曾改进过一个程序，那个程序把 50%的时间都花在约 4%的代码量上。在仅一个工作小时里，他修改了几行代码，使程序的执行速度倍增。此时，若将时间继续投入到剩余代

码的修改上，那么只会得不偿失。Knuth 在编程界有一句名言：“过早的优化是一切麻烦的根源”（Premature optimization is the root of all evil）。最明智的做法是抑制过早优化的冲动，因为那样做可能遗漏多种有用的编程技术，造成代码更难理解和操控，并需更大的精力进行维护。

D.2 寻找瓶颈

为找出最影响程序性能的瓶颈，可采取下述几种方法：

D.2.1 安插自己的测试代码

插入下述“显式”计时代码，对程序进行评测：

```
long start = System.currentTimeMillis();
// 要计时的运算代码放在这儿
long time = System.currentTimeMillis() - start;
```

利用 System.out.println()，让一种不常用到的方法将累积时间打印到控制台窗口。由于一旦出错，编译器会将其忽略，所以可用一个“静态最终布尔值”（Static final boolean）打开或关闭计时，使代码能放心留在最终发行的程序里，这样任何时候都可以拿来应急。尽管还可以选用更复杂的评测手段，但若仅仅为了量度一个特定任务的执行时间，这无疑是最简便的方法。

System.currentTimeMillis()返回的时间以千分之一秒（1 毫秒）为单位。然而，有些系统的时间精度低于 1 毫秒（如 Windows PC），所以需要重复 n 次，再将总时间除以 n，获得准确的时间。

D.2.2 JDK 性能评测[2]

JDK 配套提供了一个内建的评测程序，能跟踪花在每个例程上的时间，并将评测结果写入一个文件。不幸的是，JDK 评测器并不稳定。它在 JDK 1.1.1 中能正常工作，但在后续版本中却非常不稳定。

为运行评测程序，请在调用 Java 解释器的未优化版本时加上-prof 选项。例如：

```
java_g -prof myClass
```

或加上一个程序片（Applet）：

```
java_g -prof sun.applet.AppletViewer applet.html
```

理解评测程序的输出信息并不容易。事实上，在 JDK 1.0 中，它居然将方法名称截短为 30 字符。所以可能无法区分出某些方法。然而，若您用的平台确实能支持-prof 选项，那么可试试 Vladimir Bulatov 的“HyperPorf”[3]或者 Greg White 的“ProfileViewer”来解释一下结果。

D.2.3 特殊工具

如果想随时跟上性能优化工具的潮流，最好的方法就是作一些 Web 站点的常客。比如由 Jonathan Hardwick 制作的“Tools for Optimizing Java”（Java 优化工具）网站：

http://www.cs.cmu.edu/~jch/java/tools.html

D.2.4 性能评测的技巧

■由于评测时要用到系统时钟，所以当时不要运行其他任何进程或应用程序，以免影响测试结果。
■如对自己的程序进行了修改，并试图（至少在开发平台上）改善它的性能，那么在修改前后应分别测试一下代码的执行时间。
■尽量在完全一致的环境中进行每一次时间测试。
■如果可能，应设计一个不依赖任何用户输入的测试，避免用户的不同反应导致结果出现误差。

D.3 提速方法

现在，关键的性能瓶颈应已隔离出来。接下来，可对其应用两种类型的优化：常规手段以及依赖 Java 语言。

D.3.1 常规手段

通常，一个有效的提速方法是用更现实的方式重新定义程序。例如，在《Programming Pearls》（编程拾贝）一书中[14]，Bentley 利用了一段小说数据描写，它可以生成速度非常快、而且非常精简的拼写检查器，从而介绍了 Doug McIlroy 对英语语言的表述。除此以外，与其他方法相比，更好的算法也许能带来更大的性能提升——特别是在数据集的尺寸越来越大的时候。欲了解这些常规手段的详情，请参考本附录末尾的“一般书籍”清单。

D.3.2 依赖语言的方法

为进行客观的分析，最好明确掌握各种运算的执行时间。这样一来，得到的结果可独立于当前使用的计算机——通过除以花在本地赋值上的时间，最后得到的就是“标准时间”。

运算 示例 标准时间

本地赋值 i=n; 1.0
实例赋值 this.i=n; 1.2
int 增值 i++; 1.5
byte 增值 b++; 2.0
short 增值 s++; 2.0
float 增值 f++; 2.0
double 增值 d++; 2.0
空循环 while(true) n++; 2.0
三元表达式 (x<0) ?-x : x 2.2
算术调用 Math.abs(x); 2.5
数组赋值 a[0] = n; 2.7
long 增值 l++; 3.5
方法调用 funct(); 5.9
throw 或 catch 异常 try{ throw e; }或 catch(e){} 320
同步方法调用 synchMehod(); 570
新建对象 new Object(); 980
新建数组 new int[10]; 3100

通过自己的系统（如我的 Pentium 200 Pro，Netscape 3 及 JDK 1.1.5），这些相对时间向大家揭示出：新建对象和数组会造成最沉重的开销，同步会造成比较沉重的开销，而一次不同步的方法调用会造成适度的开销。参考资源[5]和[6]为大家总结了测量用程序片的 Web 地址，可到自己的机器上运行它们。

1. 常规修改

下面是加快 Java 程序关键部分执行速度的一些常规操作建议（注意对比修改前后的测试结果）。

将... 修改成... 理由

接口 抽象类（只需一个父时） 接口的多个继承会妨碍性能的优化

非本地或数组循环变量 本地循环变量 根据前表的耗时比较，一次实例整数赋值的时间是本地整数赋值时间的 1.2 倍，但数组赋值的时间是本地整数赋值的 2.7 倍

链接列表（固定尺寸） 保存丢弃的链接项目，或将列表替换成一个循环数组（大致知道尺寸） 每新建一个对象，都相当于本地赋值 980 次。参考“重复利用对象”（下一节）、Van Wyk[12] p.87 以及 Bentley[15] p.81

x/2（或 2 的任意次幂） X>>2（或 2 的任意次幂） 使用更快的硬件指令

D.3.3 特殊情况

■字串的开销：字串连接运算符+看似简单，但实际需要消耗大量系统资源。编译器可高效地连接字串，但变量字串却要求可观的处理器时间。例如，假设 s 和 t 是字串变量：

```
System.out.println("heading" + s + "trailer" + t);
```

上述语句要求新建一个 StringBuffer（字串缓冲），追加自变量，然后用 toString()将结果转换回一个字串。因此，无论磁盘空间还是处理器时间，都会受到严重消耗。若准备追加多个字串，则可考虑直接使用一个字串缓冲——特别是能在一个循环里重复利用它的时候。通过在每次循环里禁止新建一个字串缓冲，可节省 980 单位的对象创建时间（如前所述）。利用 substring()以及其他字串方法，可进一步地改善性能。如果可行，字符数组的速度甚至能够更快。也要注意由于同步的关系，所以 StringTokenizer 会造成较大的开销。

■同步：在 JDK 解释器中，调用同步方法通常会比调用不同步方法慢 10 倍。经 JIT 编译器处理后，这一性能上的差距提升到 50 到 100 倍（注意前表总结的时间显示出要慢 97 倍）。所以要尽可能避免使用同步方法——若不能避免，方法的同步也要比代码块的同步稍快一些。

■重复利用对象：要花很长的时间来新建一个对象（根据前表总结的时间，对象的新建时间是赋值时间的 980 倍，而新建一个小数组的时间是赋值时间的 3100 倍）。因此，最明智的做法是保存和更新老对象的字段，而不是创建一个新对象。例如，不要在自己的 paint()方法中新建一个 Font 对象。相反，应将其声明成实例对象，再初始化一次。在这以后，可在 paint()里需要的时候随时进行更新。参见 Bentley 编著的《编程拾贝》，p.81[15]。

■异常：只有在不正常的情况下，才应放弃异常处理模块。什么才叫“不正常”

呢？这通常是指程序遇到了问题，而这一般是不愿见到的，所以性能不再成为优先考虑的目标。进行优化时，将小的“try-catch”块合并到一起。由于这些块将代码分割成小的、各自独立的片断，所以会妨碍编译器进行优化。另一方面，若过份热衷于删除异常处理模块，也可能造成代码健壮程度的下降。

■散列处理：首先，Java 1.0 和 1.1 的标准“散列表”（Hashtable）类需要造型以及特别消耗系统资源的同步处理（570 单位的赋值时间）。其次，早期的 JDK 库不能自动决定最佳的表格尺寸。最后，散列函数应针对实际使用项（Key）的特征设计。考虑到所有这些原因，我们可特别设计一个散列类，令其与特定的应用程序配合，从而改善常规散列表的性能。注意 Java 1.2 集合库的散列映射（HashMap）具有更大的灵活性，而且不会自动同步。

■方法内嵌：只有在方法属于 final（最终）、private（专用）或 static（静态）的情况下，Java 编译器才能内嵌这个方法。而且某些情况下，还要求它绝对不可以有局部变量。若代码花大量时间调用一个不含上述任何属性的方法，那么请考虑为其编写一个“final”版本。

■I/O：应尽可能使用缓冲。否则，最终也许就是一次仅输入 / 输出一个字节的恶果。注意 JDK 1.0 的 I/O 类采用了大量同步措施，所以若使用象 readFully()这样的一个“大批量”调用，然后由自己解释数据，就可获得更佳的性能。也要注意 Java 1.1 的“reader”和“writer”类已针对性能进行了优化。

■造型和实例：造型会耗去 2 到 200 个单位的赋值时间。开销更大的甚至要求上溯继承（遗传）结构。其他高代价的操作会损失和恢复更低层结构的能力。

■图形：利用剪切技术，减少在 repaint()中的工作量；倍增缓冲区，提高接收速度；同时利用图形压缩技术，缩短下载时间。来自 JavaWorld 的“Java Applets”以及来自 Sun 的“Performing Animation”是两个很好的教程。请记着使用最贴切的命令。例如，为根据一系列点画一个多边形，和 drawLine()相比，drawPolygon()的速度要快得多。如必须画一条单像素粗细的直线，drawLine(x,y,x,y)的速度比 fillRect(x,y,1,1)快。

■使用 API 类：尽量使用来自 Java API 的类，因为它们本身已针对机器的性能进行了优化。这是用 Java 难于达到的。比如在复制任意长度的一个数组时，arraryCopy()比使用循环的速度快得多。

■替换 API 类：有些时候，API 类提供了比我们希望更多的功能，相应的执行时间也会增加。因此，可定做特别的版本，让它做更少的事情，但可更快地运行。例如，假定一个应用程序需要一个容器来保存大量数组。为加快执行速度，可将原来的 Vector（矢量）替换成更快的动态对象数组。

1. 其他建议

■将重复的常数计算移至关键循环之外——比如计算固定长度缓冲区的 buffer.length。

■static final（静态最终）常数有助于编译器优化程序。

■实现固定长度的循环。

■使用 javac 的优化选项：-O。它通过内嵌 static，final 以及 private 方法，从而优化编译过的代码。注意类的长度可能会增加（只对 JDK 1.1 而言——更早的版本也许不能执行字节查证）。新型的“Just-in-time”（JIT）编译器会动态加速代码。

■尽可能地将计数减至 0——这使用了一个特殊的 JVM 字节码。

D.4 参考资源

D.4.1 性能工具

[1] 运行于 Pentium Pro 200，Netscape 3.0，JDK 1.1.4 的 MicroBenchmark（参见下面的参考资源[5]）
[2] Sun 的 Java 文档页——JDK Java 解释器主题：
http://java.sun.com/products/JDK/tools/win32/java.html
[3] Vladimir Bulatov 的 HyperProf
http://www.physics.orst.edu/~bulatov/HyperProf
[4] Greg White 的 ProfileViewer
http://www.inetmi.com/~gwhi/ProfileViewer/ProfileViewer.html

D.4.2 Web 站点

[5] 对于 Java 代码的优化主题，最出色的在线参考资源是 Jonathan Hardwick 的“Java Optimization”网站：
http://www.cs.cmu.edu/~jch/java/optimization.html
“Java 优化工具”主页：
http://www.cs.cmu.edu/~jch/java/tools.html
以及“Java Microbenchmarks”（有一个 45 秒钟的评测过程）：
http://www.cs.cmu.edu/~jch/java/benchmarks.html

D.4.3 文章

[6] “Make Java fast:Optimize! How to get the greatest performanceout of your code through low-level optimizations in Java”（让 Java 更快：优化！如何通过在 Java 中的低级优化，使代码发挥最出色的性能）。作者：Doug Bell。网址：
http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html
（含一个全面的性能评测程序片，有详尽注释）
[7] “Java Optimization Resources”（Java 优化资源）
http://www.cs.cmu.edu/~jch/java/resources.html
[8] “Optimizing Java for Speed”（优化 Java，提高速度）：
http://www.cs.cmu.edu/~jch/java/speed.html
[9] “An Empirical Study of FORTRAN Programs”（FORTRAN 程序实战解析）。作者：Donald Knuth。1971 年出版。第 1 卷，p.105-33，“软件——实践和练习”。
[10] “Building High-Performance Applications and Servers in Java:An Experiential Study”。作者:Jimmy Nguyen，Michael Fraenkel，RichardRedpath，Binh Q. Nguyen 以及 Sandeep K. Singhal。IBM T.J. Watson ResearchCenter,IBM Software Solutions。
http://www.ibm.com/java/education/javahipr.html

D.4.4 Java 专业书籍

[11] 《Advanced Java，Idioms，Pitfalls，Styles, and Programming Tips》。作者：Chris Laffra。Prentice Hall 1997 年出版（Java 1.0）。第 11 章第 20 小节。

D.4.5 一般书籍

[12] 《Data Structures and C Programs》（数据结构和 C 程序）。作者：J.Van Wyk。

Addison-Wesly 1998 年出版。
[13] 《Writing Efficient Programs》(编写有效的程序)。作者：Jon Bentley。Prentice Hall 1982 年出版。特别参考 p.110 和 p.145-151。
[14] 《More Programming Pearls》(编程拾贝第二版)。作者：JonBentley。“Association for Computing Machinery”，1998 年 2 月。
[15] 《Programming Pearls》(编程拾贝)。作者：Jone Bentley。Addison-Wesley 1989 年出版。第 2 部分强调了常规的性能改善问题。 [16] 《Code Complete:A Practical Handbook of Software Construction》(完整代码索引：实用软件开发手册)。作者：Steve McConnell。Microsoft 出版社 1993 年出版，第 9 章。
[17] 《Object-Oriented System Development》(面向对象系统的开发)。作者：Champeaux，Lea 和 Faure。第 25 章。
[18] 《The Art of Programming》(编程艺术)。作者：Donald Knuth。第 1 卷“基本算法第 3 版”；第 3 卷“排序和搜索第 2 版”。Addison-Wesley 出版。这是有关程序算法的一本百科全书。
[19] 《Algorithms in C:Fundammentals,Data Structures, Sorting,Searching》(C 算法：基础、数据结构、排序、搜索）第 3 版。作者：RobertSedgewick。Addison-Wesley 1997 年出版。作者是 Knuth 的学生。这是专门讨论几种语言的七个版本之一。对算法进行了深入浅出的解释。

英文版主页 | 中文版主页 | 详细目录 | 关于译者