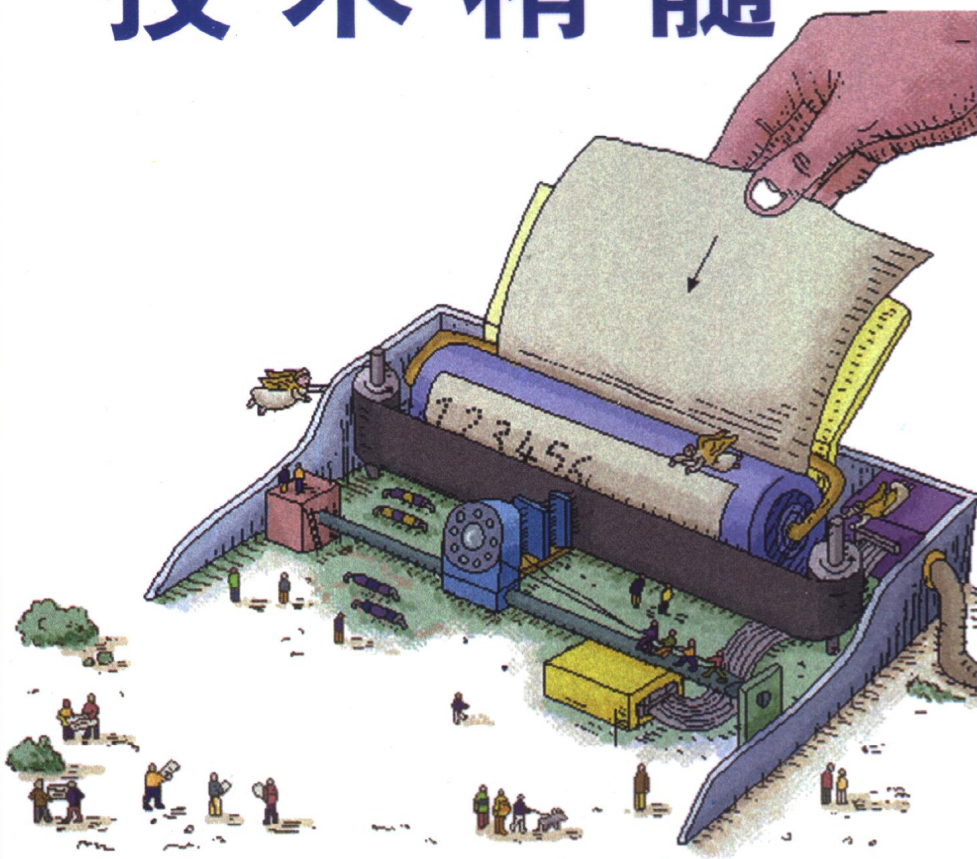
HZ BOOKS

Enterprise Java Performance

Sun公司核心技术丛书

Sun microsystems

Java 技术精髓



(美) Steven L. Halter
Steven J. Munroe 著

许崇梅 张雪莲 等译

机械工业出版社
China Machine Press

PH PTR

Sun 公司核心技术丛书

Java 技术精髓

（美） Steven L. Halter
Steven J. Munroe 著

许崇梅 张雪莲 等译
前导工作室 审校

机械工业出版社
China Machine Press

本书是一本关于如何调整 Java 系统性能的书。本书从基本的例子出发逐步引出企业性能中更加复杂的问题，从而方便读者理解 Java 性能技术。

书中大量的实践知识可帮助大规模的分布在线多供应商的 Java 系统正常地，甚至更快地工作。本书会为正在设计或建立企业级 Java 系统的技术人员节省大量的时间，同时也会给具有一定 Java 编程基础的读者极大的帮助。

本书版权登记号：图字：01－2001－3868

图书在版编目（CIP）数据

Java 技术精髓/（美）哈特（Halter, S.L.），（美）姆诺（Munroe, S. J.）著；许崇梅等译. －北京：机械工业出版社，2002. 2
（Sun 公司核心技术丛书）
书名原文：Enterprise Java Performance
ISBN 7-111-09643-6

Ⅰ.J… Ⅱ.①哈…②姆…③许… Ⅲ.JAVA 语言－程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2001）第 088571 号

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）
责任编辑：徐冬梅 张鸿斌
北京昌平奔腾印刷厂印刷·新华书店北京发行所发行
2002 年 2 月第 1 版第 1 次印刷
787mm×1092mm 1/16·17.25 印张
印数：0 001-5 000 册
定价：30.00 元

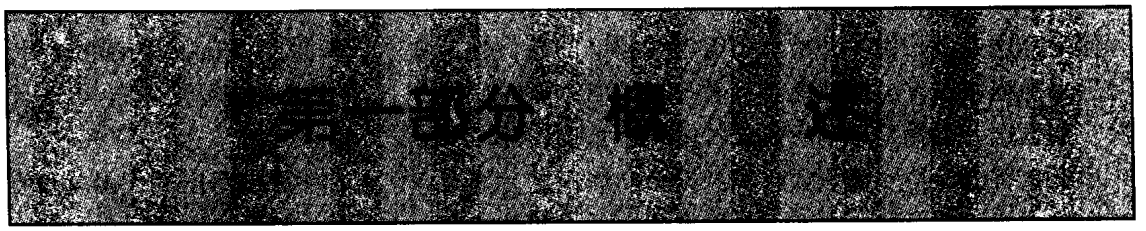
目　录

第三部分 基准测试

第四部分 应用程序环境

第五部分 应用程序模型

第六部分 扩大应用程序规模



欢迎进入 Java 的性能世界。本书第一部分主要介绍 Java 特有的基本性能概念。这些概念是本书后面要深入探讨的主题的基础。本部分的各章节所介绍的概念是 Java 性能整体讨论的基础，不管是企业级的还是更小规模的 Java 都会遇到性能问题。

第 1 章介绍性能问题的基本分类，第 2 章概述了测试 Java 性能的主要工具。第 3 章给出了在 Java 编程特有的普通性能上的一些背景知识。

当阅读完这三章以后，会准确地了解一些基本概念。这就为建立有关 Java 性能的概念和在本书的其余部分中进一步深入理解 Java 奠定了基础。

第 1 章　一 般 性 能

本章主题：

- 考虑性能的时机
- 性能问题的分类

本章探讨了在编码过程中所要考虑的性能之间的关系，并且介绍了一种性能问题的分类方法，在整本书中都会用到这样的分类方法。我们首先想到的是“性能是什么?”

实际上考察程序的性能有两种方式。一种是以客观的观点来看：“程序运行 3s，使用了 8MB 的内存空间。”在这种观点下，可以通过考察在特定硬件环境下执行特定应用程序的时间来测试性能。其他性能的客观量度可以包括应用程序执行时所消耗的资源，如内存大小。客观量度既可以是程序的速度也可以是程序运行的总体耗费。

另一种是以主观的观点来看：“程序的运行速度能满足我们的需要”。在这种观点下，性能是根据我们的需要而定义的。一个程序的运行，可能会满足某个用户的需求，但却不能满足其他用户的需求。程序性能优劣程度总是与用户的期望值有关的。

1.1　性能与优化

总是听到这样的建议，开始编码的时候才应该考虑性能问题。这样的说法一般是指不必考虑代码的低层优化。花费大量的时间来优化特定代码的执行路线，通常是一种浪费。因为经常不清楚需要执行多少个代码段。程序越大越是如此。

但是，这并不是说一开始就不需考虑程序的整体性能要求。如果不知道程序的性能要求，编写出来的程序很有可能就达不到程序的目标。例如，如果程序要求对一行汇编语句 1s 监听一次，那么这一点就是影响代码设计的一个重要信息。

尽最大努力保持平衡。如果等到设计和编码之后再考虑性能，那么编写出的代码就很有可能存在较大的问题。另一方面，如果把所有的时间都花费在优化每一条最终的代码上，那么永远也生产不出来产品。

一个良好而规整的设计是非常重要的。简单、可读性强的代码同样也是很重要的。幸运的是，在许多情况下，代码简洁和优化是没有冲突的。最简单的代码通常执行效率也最好。在许多情况下，这只是一个学习编写具有执行性能良好风格的代码的问题。在第 3 章中，我们给出了一些有关 Java 性能特征的简单例子。本书的其余部分也会帮助你培养起编写具有良好性能风格代码的技能。

如果不清楚一段代码使用的频率，最好不要把时间浪费在考虑可以确保最佳性能的方法上。在这种情况下，可以先编写一些合理的代码，当需要时再考虑性能。但是，在这里要强调代码的合理性。如果编写了一些不可救药的代码，那么它们总是会不断地困扰你。

在某些情况下，如果在设计阶段非常清楚将会频繁使用一段代码，那么就值得花费时间来考虑如何执行这段代码。这不仅要考虑所写代码的细节，而且还要考虑它如何和程序的其他部分交互。这时要在编码之前考虑优化。但是做这个要十分慎重，一般只有在非常清楚性能本质的时候才这样做。

1.2　性能的生命周期

软件设计的方法有很多。市面上有关软件设计方法的好书也很多。所以我们将不讨论任何的设计方法学。但是我们注意到，许多软件设计方法都可以归纳为 3 个基本步骤：编码前、编码期间和编码后。在开发过程中，这 3 个步骤不断地循环往复。

和软件设计可以合理地分成 3 个基本步骤一样，软件运行也可以分成这样的 3 个步骤。

1.2.1　编码前

首先，最容易确定的性能问题不会出现在编码前。如果在工程初期就把代码的性能作为工程的目标，很有可能就会很容易地得到要求的性能。

在开始编码前，一定要了解要编写的软件的基本目标。从使用的观点来看，这是很明显的事实——在写东西之前，要对所写的东西有个大概的了解。从性能的观点来看，同样也是这个道理。

程序要求的性能是由程序本身的特性来确定的。有些性能的要求是很明确的，例如：

- 系统每小时处理 1000 条命令请求。
- 每 3s 要检查一次处理器的温度。

其他的要求不是很明确：

- 用户应该得到平稳的控制流。
- 后台进程不应该明显地妨碍交互作业。

当给定的性能要求不明确时，要尽力弄清楚这些要求的具体内容。设置完要求的参数之后，还需要做许多工作来确定需要投入多少努力。例如，在进一步研究之后，可能发现这个要求。

- 用户应该得到平稳的控制流。

实际上它的意思是：

- 用户在屏幕之间切换时，延迟不应该大于0.25s。

除了对时间的简单讨论以外，还要确定用户期望以什么硬件环境达到指定性能水平。典型的台式机和高端服务器系统之间，存在着很大的差别。这就是服务器比简单的台式机昂贵的真正原因。尽力弄清楚程序运行要求的硬件环境，包括处理器速度、内存大小、磁盘设置、网络速度和其他任何可能影响代码的因素。

在工程设计阶段确定有关时间和平台的基本性能要求是重要的。这对实际检查设计的其他部分很有帮助。如果清楚代码的运行必须限制在特定数量的内存上，就会限制所使用的数据类型或算法。我们会在第3章详细讨论一些通用的Java数据类型。

1. 简单的设计准则

除了如时间和资源这样必要的性能以外，在代码设计时牢记一些设计准则也是很有用的。

- 尽可能使用最简单的类—而不是比较简单的类。

设计代码永远是平衡的训练。如果为程序选择过于复杂的数据结构，就会导致程序运行效率低下、代码冗长复杂。例如，当数组可以满足程序需要时，选择 *java.util.Vector* 就是不明智的。在第3章我们会给出一些例子，在这些例子中，只要做简单的选择就能导致性能的不同。然而，如果使用不满足设计要求的数据结构，也是不明智的。首先，不合适的数据结构会引起代码重复，并且会失去使用面向对象语言的许多优势。如果代码确实需要 *java.util.Vector* 的功能性，那么使用数组可能会导致在自定义的类中重新编写一个 *java.util.Vector*。

- 永远也不要做别人已经为你做好的东西。

如果别人已经写好了你所需要的类，只要使用它就可以了。只是要记住你所需要的东西必须满足程序的性能特征。异常处理是较底层细节方面的一个好例子。调用者可能会更好地处理一个特殊的异常。如果捕捉异常之后只是再次抛出而没有附加任何值，那么这样做仅仅是浪费时间。

- 永远不要把今天应该做的事情推迟到明天去做。

如果一段代码的设计或编写是异常困难的，一个好建议就是用其他的类来代替它。有关这条准则如何减少性能开销的例子可以在本地的详细信息的翻译文档中找到。例如，如果发送的错误消息中含有可读取信息，那么只有当这条错误信息被显示的时候，才会把它翻译成具体的语言。所以，最后的错误消息最好由客户程序代码来翻译，而不是在首次出现异常（举例来说）的时候。

2. 设计的注意事项

当进行工程设计时，确保把关于性能要求的所有知识都添加到设计中。这一点对编码期间设法估计哪部分代码最影响性能有很大帮助。例如，如果在设计阶段知道应用程序的某一部分比其他部分使用频率高，就要把这一点记录下来。即使设计者和编码者是同一个人，这些信息也是相当有用的。如果设计者和编码者不是同一个人，那么这些信息就更加关键了。

1.2.2 编码期间

经常听见这样的说法："首先编写一个可以执行的代码，然后才能确定性能。即编码未完成不能确定性能。"

在某种程度上，我们同意这个观点。因为如果代码不能正确执行，那么多快的运行速度都是毫无意义的。然而，这个观点实际上只是暴露了对性能良好代码的特性的误解。这个误解就是性能良好的代码是很难编写的。其实正好相反，性能良好的代码并不比性能差的代码更难编写。实际上，它经常比性能差的代码更容易编写。窍门就是学习和使用可以编写性能良好代码的编码习惯。

第3章我们将给出一些更加通用的简单Java编码技巧，通过使用这些技巧，就可以编写出性能良好的代码，避免编写性能差的代码。如果练习了这些编码技巧，就会发现理解它们并没有花费任何额外的时间。下面是一些需要注意的地方。

- 循环中的代码。

如果代码处于循环中，它就会被多次执行。不要重新计算在循环周期内不变的值。如果存在提前跳出循环的方法，一定要使用它们。

- 对象的创建。

创建对象要占用时间和空间资源。确定你确实需要创建对象。通常是重新使用已存在的对象而不是每次都创建一个新的实例。

- 代码使用的集合。

选择容纳对象的正确的集合对应用程序的性能有较大的影响。

代码选择

当编写代码时，会发现一些特定的功能可以用很多方法来实现。在下面介绍的Primes类中，将给出几种实现厄拉多塞筛法（the Sieve of Eratosthenes）的方法。这是一种可以计算出不大于给定数字的所有素数的简单方法。称之为筛选法，因为它用排除非素数的方法来筛选出素数。

在 *eratosthenes*1 方法中，我们使用一个 *java.util.BitSet* 对象。如果它的第 i 位为真，那么 (i×2) + i 就是素数。因为 *java.util.BitSet* 类的构造器把所有的位都赋值为假，所以首先必须把所有的位都重新赋值为真。对于 *eratoshenes*1 方法的使用者来说，素数如何被计算出来的所有细节都被隐藏了。*eratosthenes*2 方法与前者在逻辑上正好相反，它用 *java.util.BitSet* 对象的相应位为假来指出素数。因为构造器开始就把所有的位赋值为假，所以就不用再次初始化

java.util.BitSet 对象了。这是使用设计准则“永远也不要做别人已经为你做好的东西”的一个非常好的例子。既然 Java 已经适当地初始化了 *java.util.BitSet* 对象——就没有必要再次进行初始化了。

在 *eratosthenes3* 方法中，使用布尔数组来代替 *java.util.BitSet*。在这个例子中，数组在功能上和 *java.util.BitSet* 是一样。记住尽可能使用最简单的类。

键入下面这句话来运行 Primes：

```
java Primes
```

在机器 A[⊖] 中运行该程序，输出结果如下：

```
eratosthenes1 found 78498 primes in the first 1000000 numbers in 1212 milliseconds
eratosthenes2 found 78498 primes in the first 1000000 numbers in 801 milliseconds
eratosthenes3 found 78498 primes in the first 1000000 numbers in 151 milliseconds
```

通过在代码中简单地用 *eratosthenes2* 方法来代替 *eratoshtenes1* 方法，节省了大约 30% 的运行时间。因为所有的实现细节都对使用者隐藏，所以我们可以自由地做这样的改变。通过在代码中用 *eratosthenes3* 方法代替 *eratosthenes2* 方法，即用布尔数组代替 *Java.util.BitSet*，又节省了 81% 的运行时间。

这些改变不仅非常简单而且还产生了相当惊人的结果。这是非常普通的事情。通常情况下，使用比较简单的结构和利用设置好的值会得到相当好的结果。

下面是包含方法 *eratosthenes1*、*eratoshenes2* 和 *eratoshenes3* 的类 *Primes*：

```
import java.util.BitSet;

public class Primes {
  public int eratosthenes1(int numberToLookAt) {
    //no even number is prime so don't look
    // so the array only represents odd numbers
    int max = numberToLookAt/2;

    // remove the last even
    if ((numberToLookAt % 2) == 0)
      max = max - 1;

    //If bit i = 1 then the number (i*2)+1 is prime
    BitSet possiblePrimes = new BitSet(max+1);

    // Assume all the numbers are prime,
    // set them to true
    for (int i = 1; i <= max; i++)
      possiblePrimes.set(i);


  int countOfPrimes = 1; // 2 is prime
  int currentBit = 1, step = 3, nextNonPrime = 4;
```

⊖ 见附录 A——本书为计时给出的机器和环境列表。

```
 while (nextNonPrime <= max) {
   if (possiblePrimes.get(currentBit)) {
     countOfPrimes++;
     int nonPrime = nextNonPrime;
     while (nonPrime <= max) {
       possiblePrimes.clear(nonPrime);
       nonPrime += step;
     }
   }

   currentBit++;
   step += 2;
   nextNonPrime += ((2 * step) - 2);
 }

 for (; currentBit <= max; ++currentBit) {
   if (possiblePrimes.get(currentBit))
     countOfPrimes++;
 }
 return countOfPrimes;
}

public int eratosthenes2(int numberToLookAt) {
  // no even number is prime so don't look
  // so the array only represents odd numbers
  int max = numberToLookAt/2;

  // remove the last even
  if ((numberToLookAt % 2) == 0)
    max = max - 1;

  //If bit i = 0 then the number (i*2)+1 is prime
  //Assume all the numbers are prime, already set
  BitSet possiblePrimes = new BitSet(max+1);

  int countOfPrimes = 1; // 2 is prime
  int currentBit = 1, step = 3, nextNonPrime = 4;
  while (nextNonPrime <= max) {
    if (possiblePrimes.get(currentBit) == false){
      countOfPrimes++;
      int nonPrime = nextNonPrime;
      while (nonPrime <= max) {
        possiblePrimes.set(nonPrime);
        nonPrime += step;
      }
    }

    currentBit++;
    step += 2;
    nextNonPrime += ((2 * step) - 2);
  }

  for (; currentBit <= max; ++currentBit) {
    if (possiblePrimes.get(currentBit) == false)
      countOfPrimes++;
  }
  return countOfPrimes;
```

```
}

public int eratosthenes3(int numberToLookAt) {
  // no even number is prime so don't look
  // so the array only represents odd numbers
  int max = numberToLookAt/2;

  // remove the last even
  if ((numberToLookAt % 2) == 0)
    max = max - 1;

  //If bit i = 0 then the number (i*2)+1 is prime
  //Assume all the numbers are prime, already set
  boolean[] possiblePrimes = new boolean[max+1];

  int countOfPrimes = 1; // 2 is prime
  int currentBit = 1, step = 3, nextNonPrime = 4;
  while (nextNonPrime <= max) {
    if (possiblePrimes[currentBit] == false) {
      countOfPrimes++;
      int nonPrime = nextNonPrime;
      while (nonPrime <= max) {
        possiblePrimes[nonPrime] = true;
        nonPrime += step;
      }
    }

    currentBit++;
    step += 2;
    nextNonPrime += ((2 * step) - 2);
  }

  for (; currentBit <= max; ++currentBit) {
    if (possiblePrimes[currentBit] == false)
      countOfPrimes++;
  }
  return countOfPrimes;
}

public static void main(String[] s) {
  int n = 1000000;
  Primes p = new Primes();
  int count = 0;

  long time1 = System.currentTimeMillis();
  count = p.eratosthenes1(n);
  long time2 = System.currentTimeMillis();
  System.out.println("eratosthenes1 found " +
                     count +
                    " primes in the first " + n +
                    " numbers in " +
                    (time2-time1) +
                    " milliseconds");
time1 = System.currentTimeMillis();
count = p.eratosthenes2(n);
time2 = System.currentTimeMillis();
System.out.println("eratosthenes2 found " +
```

```
                    count +
                    " primes in the first " + n +
                    " numbers in " +
                    (time2-time1) +
                    " milliseconds");

    time1 = System.currentTimeMillis();
    count = p.eratosthenes3(n);
    time2 = System.currentTimeMillis();
    System.out.println("eratosthenes3 found " +
                    count +
                    " primes in the first " + n +
                    " numbers in " +
                    (time2-time1) +
                    " milliseconds");
  }
}
```

1.2.3　编码后

在编写完代码并且得到正确的操作结果之后，一定要测试代码的性能是否符合设计阶段确定的性能要求。本书的第三部分将有一个更加详细的讨论，并且还将给一些关于应用程序测试方法的例子。

很有可能在程序一开始运行时，代码就不能满足性能要求。即使能满足，寻找一些方法来提高性能也是值得的。

1.3　性能问题的类型

图 1-1 说明了考虑性能问题的一个有用的方法。如图 1-1 所示，性能问题分为三个基本层次：第一层是指非常明显并且最容易处理的性能问题（简单的性能）；第二层是指应用程序设计产生的性能问题；第三层是指在“物理”状态上产生的性能问题——我们使用这个术语来表示事情是如何工作的，尤其是那些不能够直接改变的事情。如果我们明白了事情是如何工作的，那么就会做出更好的选择、就能充分利用现有的东西或者避免出现问题。

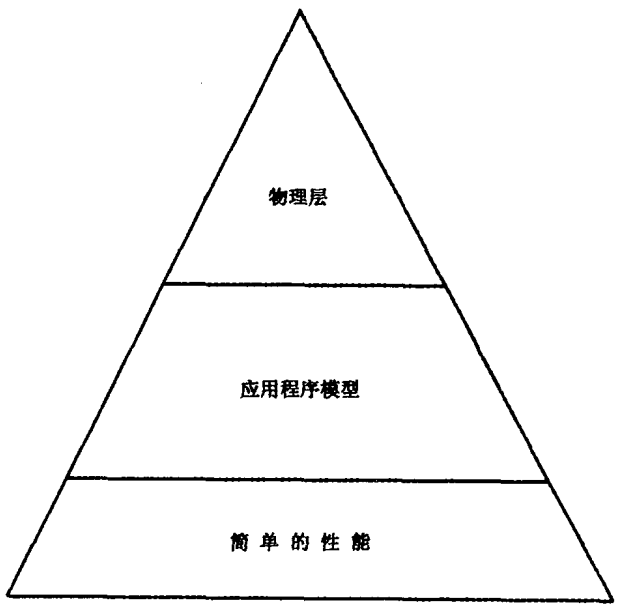


图 1-1　性能金字塔

要牢牢记住性能问题的一个有趣特征就是当解决了一组问题时，可能又会出现另一组问题。在分析性能的

过程中，经常会发生这种情况，一个性能问题产生的瓶颈掩盖了分析过程中所有种类的潜在问题。这就使得解决性能问题的工作不是一次就能完成的而是需要反复的去做。当解决性能问题的时候，经常会发现需要在每个层次的性能问题上不断地进行重复。

另外，尽管可以发现解决性能问题的一般顺序是从简单的性能层（low-hanging fruit）到顶层，但是有时还是需要从物理层开始或者从应用程序设计层开始。当前任务通常都会指出解决性能问题的出发点，在本书的所有例子中都可以看到这一点。

1.4 简单性能层

和从最低的树枝摘苹果是最容易的一样，第一层的性能问题通常是最容易发现和解决的。在简单的性能层出现的性能问题经常是由应用程序的实现引起的。这一层的性能问题包括从最简单的语言错误到完全的程序错误的所有问题。

即使最优秀的程序员也会犯错误。首要的问题就是如何找到错误产生的地方。第2章将详细地讨论如何找到发生性能问题的地方。这些性能问题的解决方案通常是在没有改变整个应用程序的结构或设计的情况下，进行一个简单的替换（一个更好的算法、一个更合适的实用对象）。这些问题虽然很难发现，但是却很容易解决。

1.5 应用程序设计层

第二层的性能问题是由应用程序的设计引起的。在抽象设计层看起来非常好的方法可能会产生相当严重的影响。有时只需要对设计做较小的调整就可以解决这些性能问题。但是，通常情况下是需要改变基本设计的。

应用程序的设计及它如何与系统的所有层交互与程序的性能有很大关系。虽然看起来相当明显，但是性能问题却还是经常被忽视或者被错误理解，这就导致了必须重做部分（或全部）设计。最后考虑性能问题这个传统警告总是对人们产生误导。性能问题虽然不需要在设计阶段占支配地位，但是必需要加以考虑。

1.5.1 不好的设计选择

良好设计的基石之一就是在适当的时候做出正确的选择。这既包括选择数据结构也包括选择逻辑算法。举例来说，如果正在设计的代码需要使用集合，那么集合的选择将是非常重要的。数据需要排序吗？如果不需要，那么选择可以保持有序状态数据的集合将会产生很大的性能损失。因为选择这样的集合不能给需要的功能带来任何好处 。正在设计的类会被远程访问吗？如果会，那么在接口上包含许多小的方法可能是一个不好的设计。对象要求共享吗？如果要求共享，那么就要考虑加锁或者同步。

重要的设计选择要遵循前面提到的设计准则：“永远不要把今天应该做的事情推迟到明天去做。”假定需要插入对象到共享的持久集合中。那么当修改集合之后，需要给数据库加一个写锁，来保证其他事务不会使用它。如果在更新共享对象前还有工作要干，那么要首先进行这一操作。例如，如果在完成更新集合之前需要从其他数据库中读取数据，那么首先要做的是读取数据而不是不必要地长时间保持写锁。然后在最后的时刻更新共享对象和完成事务。我们将

在第6章“瓶颈”中详细讨论这个问题。

1.5.2 信息隐藏

为了在适当的时候做出正确的选择，必须有正确的信息。关于应用程序设计为何没能满足性能目标的许多讨论都是这样开头的：“我忘了……”或者“我没有意识到……”。在某些情况下，这些问题是由隐藏信息引起的。“数据隐藏”和“黑盒”是经常用来描述面向对象编程语言优点的。有时，不知道（或忘记）的问题会给你带来很大的苦恼。不知道的问题有时候指的是在不正确假设的基础上做出重要的设计决定（见下面的物理层讨论）。为了改正这类问题经常要求有效的重新设计。

例如，考虑一个日程安排应用程序（见图1-2）。程序的安装是一系列的循环。在 *EmployeeCalendar* 类中包含了一个 *WorkWeek* 对象的集合，而在 *WorkWeek* 对象中又包含了工作日的日期，即从星期一到星期五。所有这些对象的初始化都会在安装程序中完成。这是非常简单的，但是应用程序必须考虑到公司承认的国家和本地的节假日。这也是相当简单的：当添加工作日时，首先使用 *getActiveCompany*（）方法来存取当前的公司，然后选择节假日政策，最后通过应用程序循环一周的日期来进行核查。迄今为止，很明显只有 *WorkWeek* 类需要存取 *Company* 对象。但是，*WorkWeek* 类的外部接口并没有提到它要存取 *Company* 对象。隐藏这个信息是正确的，因为 *Company* 对象的使用是 *WorkWeek* 类的一个内部设计细节，而且以后还可能改变。

在单服务程序和小型（玩具）数据库上进行测试，这一初始设计是可行的，并可能会得到非常好的性能。但是，当应用程序被配置到地理上是分布的公司时，*Company* 对象现在要被配置在总公司的服务程序上。这时应用程序就要在雇员日程程序的最内循环中调用一个远程方法。

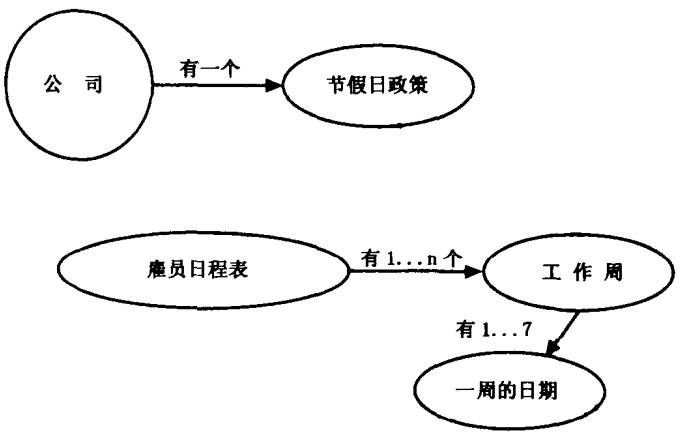


图1-2　雇员日程表

这是早先提到的“循环内部不变计算”问题的一个变种。在通常情况下，公司和它的节假日政策是变化的。但是，对于雇员日程安装程序，它们实际上是不变的。设计的解决办法就是在日程安装程序开始之前检索公司和节假日政策的信息，然后，通过应用层把这个信息做为参数传递给 *WorkWeek* 的内部循环。

这个解决方法要求改变应用程序的一些层次。应用程序的某一层将访问公司的节假日政策，但是实际上在这一层上并不需要存取公司的节假日政策的数据。这就暗示了 *EmployeeCalendar* 类应该了解 *WorkWeek* 类的内部细节（实际上 *WorkWeek* 类是要核查节假日政策的）。这就要求改变 *WorkWeek* 类以及 *EmployeeCalendar* 类和 *WorkWeek* 类之间的类的方法的签名。

1.6 物理层

性能问题的最后一层解决的是由应用程序和它的运行环境的基本物理性质引起的问题。因为物理性质是不可改变的，所以有关它的内容是非常少的。但是，必须理解物理性质，以便充分地利用当前的状况。这一点包括一系列的要素，如下：

- 通信因素：如通信网络的物理组件和速度、输入/输出信道带宽和磁盘存取访问速度。
- 购买的软件的要求和限制：如用于对象序列化的数据库、垃圾收集器如何为特定的Java虚拟机（JVM）工作或者为什么Java不同于C++、COBOL、RPC。
- 环境改变和环境之间数据转换的开销。
- 经济限制（要求使用开销少于20000美元的计算机操作系统）。
- 政治因素（保存雇员数据的人力资源系统必须在物理上被隔离，并且还要放在防火墙的后面）。

建造一个发送宇航员到月球的火箭或者制造一个可以工作的晶体管要求完全了解物理学，同样在创建Java企业级应用程序中完全理解Java的物理性质也是非常重要的。这种理解必须扩展到Java以外，包括Java与操作系统、中件和分布计算的交互。如果体系结构设计者和软件设计者完全理解了Java的物理性质，他们就会权衡设计并且可以构造出一个切实可行的系统。

当然，没有任何一个系统一开始就能良好地工作。Java的物理性质可以用来帮助查找和解决在系统环境中遇到的问题。好的物理学家不仅知道如何提出一个假说，而且还知道如何设计实验来验证或推翻这个假说。同样，Java的物理学家也需要相似的技能。最重要的技能之一就是设计测试程序，并且在要求的方式下使用它们来分离组件和组件之间的相互作用。

即使好的编码训练可以避免或解决一些问题，但是还是会在设计中出现其他问题。这些问题是由于误解或违反了Java物理性质的原则而产生的。而且这些问题在开发环境[⊖]中是很难被发现的，只有在客户环境中运行最终程序的时候，它们才会出现。

1.6.1 Java语言及其环境

理解Java的物理性质也就意味着理解了整个Java的实质。因为Java是比较新的语言，而且它的类库也在迅速地更新，所以对于未建立的假设来说它是成熟的。另一个重要的问题与Java如何被使用有关。如果Java直接替换其他语言书写的程序，那么新老应用程序之间是可以比较的。然而，Java本身有一些独特的性能：

- 平台无关性。
- 支持多线程。
- 支持内嵌网络互连。
- 支持内嵌分布式设计。

所有这些性能都是由一个易于使用的包来实现的。利用Java的这些新特征不仅很难使用以前的经验，而且还经常引入新的性能难题。如果新的应用程序比旧的应用程序有更多的功

⊖ 开发环境通常指的是单（单处理器）工作站或者是用来测试小型数据库的单元部件。把它升级为完整的数据库、多处理器的服务器或分布式主机容易暴露潜在问题。