TURING 图灵程序设计丛书

PEARSON

Java性能优化圣经！Java之父重磅推荐！

Java性能优化权威指南

Java Performance

[美] Charlie Hunt
Binu John 著
柳飞 陆明刚 译

人民邮电出版社
POSTS & TELECOM PRESS

第1章 策略、方法和方法论

凡事预则立，不预则废，和许多事情一样，Java性能调优的成功，离不开行动计划、方法或策略以及特定领域的背景知识。为了在Java性能调优工作中有所成就，你得超越“花似雾中看”的状态，进入“悠然见南山”或者已然是“一览众山小”的境界。

这3个境界的说法可能让你有些糊涂吧，下面进一步解释。

- **花似雾中看**（I don't know what I don't know）。有时候你的任务会涉及你所不熟悉的问题域。理解陌生问题域首先面临的困难就是如何竭尽所能地了解它，因为你对它几乎一无所知。对于这类问题域，你有许多东西不了解，或者不知道重点。换句话说，这个问题域有哪些东西需要了解，你还傻傻看不清楚。这个阶段就是“花似雾中看”。
- **悠然见南山**（I know what I don't know）。刚进入不熟悉的问题域时，你对它知之甚少，随着时间的推移，你对它的许多重要方面都已有所认识，只是对重要的具体细节还缺乏了解。这时，你可以算是刚刚“见南山”。
- **一览众山小**（I already know what I need to know）。还有些时候，你对任务的问题域非常熟悉，或者已经具有该领域所必备的技能和知识，是这方面的专家。或者你对问题域足够了解，处理起来得心应手，比如你已经掌握了必要的知识，解决问题游刃有余。如果达到这个境界，那就意味着你已经是“一览众山小”了。

如果你已经或打算购买本书，说明可能还没有“一览众山小”，除非你只是想手边有本不错的参考手册。如果还在“花似雾中看”，本章将有助于你找到那些不了解的内容，以及应对Java性能问题的方法或策略。那些“悠然见南山”的读者也能从本章获得有用的信息。

本章首先讨论了性能问题的现状，并建议将性能调优集成到软件开发过程中，接着探讨了两种不同的性能调优方法——自顶向下和自底向上。

1.1 性能问题的现状

通常认为，传统的软件开发过程主要包括4个阶段：分析、设计、编码和测试，如图1-1所示。

分析是开发过程的第一步，用于评估需求、权衡各种架构的利弊以及构思高层抽象。设计则依据分析阶段的基本架构和高层抽象，进行更精细的抽象并着手考虑具体实现。编码自然就

是设计的实现。编码之后是测试，用以验证实现是否合乎应用需求。值得注意的是，测试阶段通常只包括功能测试，即检验应用的执行是否合乎需求规格。一旦测试完成，应用就可以发布给客户了。

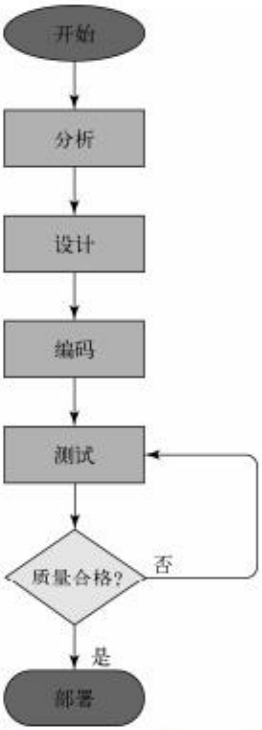

图1-1　传统软件开发过程

遵循这种传统软件开发过程的应用，通常要到测试或即将发布时才会关注性能或扩展性。为了解决这个问题，Wilson和Kesselman对传统软件开发过程做了些补充，即在传统开发模型基础上引入了性能测试分析阶段，参见他们的畅销书*Java Platform Performance*。他们建议在测试阶段之后增加性能测试，并将“性能测试是否通过”设定为产品是否发布的标准。如果达到性能和扩展性标准，应用就可以发布，否则就要转向性能分析，并依据分析结果回到之前的某个或者某些步骤。换句话说，通过性能分析来定位性能问题。Wilson和Kesselman添加的性能测试分析如图1-2所示。

对分析阶段提炼出来的性能需求，Wilson和Kesselman建议以用例的方式特别标识出来，这有助于在分析阶段制定性能评估指标。不过应用的需求文档中通常都不会明确描述性能或扩展性需求。如果你正在开发的应用还没有明确定义这些需求，那就应该想办法将它们挖掘出来。以吞吐量和延迟性需求为例，下面的清单列举了挖掘这些需求所要考虑的问题。

- 应用预期的吞吐量是多少?
- 请求和响应之间的延迟预期是多少?
- 应用支持多少并发用户或者并发任务?
- 当并发用户数或并发任务数达到最大时，可接受的吞吐量和延迟是多少?
- 最差情况下的延迟是多少?
- 要使垃圾收集引入的延迟在可容忍范围之内，垃圾收集的频率应该是多少?

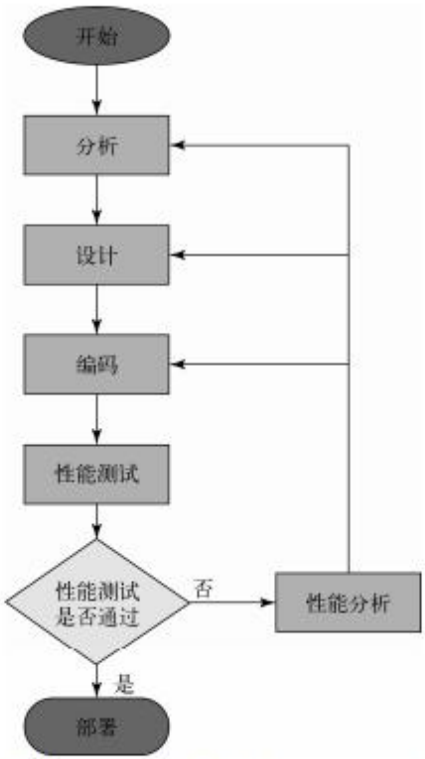

图1-2 Wilson和Kesselman添加性能测试分析之后的软件开发过程

需求和对应的用例文档应该回答上述问题，并以此制定基准测试和性能测试，确保应用能够满足性能和扩展性需求。基准测试和性能测试应该在性能测试阶段执行。有些用例在评估时发现风险很高，难以实现，就应该在结束分析阶段之前，通过一些原型、基准测试和微基准测试来降低这类风险。分析结束后再变更决策的代价非常高，这个方法可以让你事先对决策进行评估。软件开发周期中的软件缺陷、低劣设计和糟糕实现发现得越晚，修复的代价就越大，这是一条颠扑不破的金科玉律。降低用例的高风险有助于避免这些代价昂贵的错误。

现在许多应用在开发过程中都会使用自动构建和测试。Wilson和Kesselman建议改进软件开发过程，在自动构建或测试中进一步添加自动性能测试。自动性能测试可以发出通知，比如用电子邮件将性能测试结果（譬如性能是衰减还是改善、性能指标的达成度）发送给干系人。这个过

程可以把由于应用性能不达标而失败的测试，以及测试的统计数据自动记录到追踪系统。

将性能测试集成到自动构建过程中后，每次代码变更提交到源代码库时，都能很容易地追踪因变更而导致的性能变化，也就能在软件开发的早期发现性能衰减。

另外，也可以将统计方法和自动统计分析添加到自动性能测试系统。运用统计方法可以进一步验证性能测试的结果。如何用统计方法获取指导和建议呢？对于极少接触这些知识的软件开发者来说颇有难度，这方面内容将在第8章的后半部分介绍。

1.2　性能分析的两种方法：自顶向下和自底向上

自顶向下和自底向上是两种常用的性能分析方法。顾名思义，自顶向下着眼于软件栈顶层的应用，从上往下寻找优化机会和问题。相反，自底向上则从软件栈最底层的CPU统计数据（例如CPU高速缓存未命中率、CPU指令效率）开始，逐渐上升到应用自身的结构或应用常见的使用方式。应用开发人员常常使用自顶向下的方法，而性能问题专家则通常自底向上，用以辨别因不同硬件架构、操作系统或不同的Java虚拟机实现所导致的性能差异。如你所想，不同的方法可以用来查找不同类型的性能问题。

后面两节将详细介绍这两种方法。

1.2.1　自顶向下

如前所述，自顶向下是最常用的性能调优方法。如果调优涉及软件栈顶层应用代码的更改，也常用这招。

自顶向下时，你通常会从干系人发现性能问题的负载开始监控应用。应用的配置变化或日常负荷变化都可能导致性能降低，所以需要持续监控应用。此外，一旦应用的性能和扩展性需求发生变化，应用就可能无法满足新要求，所以也要监控应用程序的性能。

不管何种原因引起的性能调优，自顶向下的第一步总是对运行在特定负载之下的应用进行监控。监控的范围包括操作系统、Java虚拟机、Java EE容器以及应用的性能测量统计指标。基于监控信息所给出的提示再开展下一步工作，例如JVM垃圾收集器调优、JVM命令行选项调优、操作系统调优，或者应用程序性能分析。性能分析可能导致应用程序的更改，或者发现第三方库或Java SE类库在实现上的不足。

第2章和第4章可以帮助你了解自顶向下所需要监控的对象。这两章介绍了需要重点监控的统计数据，并且对统计值达到多少时需要进一步调查给出了建议。对于这些需要进一步调查的统计数据，你可以翻阅其他章节获得适当的行动建议。例如，操作系统监控数据显示系统CPU使用率高，你就可以分析应用程序的哪个方法耗费了最多的系统CPU周期。如何使用NetBeans Profiler和Oracle Solaris Studio Performance Analyzer（前身是Sun Studio Performance Analyzer），可以参见第5章和第6章。如果指示JVM垃圾收集器需要调优，你可以翻阅第7章。如果你想熟悉Java HotSpot VM垃圾收集器的基本运转机制，可以考虑在阅读JVM调优之前阅读第3章中的垃圾收集。如果你在监控应用的统计数据，例如Java EE容器所提供的统计数据，可以阅读Java EE性能调优的相关

章节——第10章“Web应用的性能调优”、第11章“Web Service的性能”、第12章“Java持久化及Enterprise Java Bean的性能”，从中学习如何解决企业级应用的性能问题。

1.2.2 自底向上

在不同平台（指底层的CPU架构或CPU数量不同）上进行应用性能调优时，性能专家常使用自底向上的方法。将应用迁移到其他操作系统上时，也常用这种方法改善性能。在无法更改应用源代码，例如应用已经部署在生产环境中，或者系统供应商为了在竞争中占得先机而必须将性能发挥到极致时，也常常会使用这种方法。

自底向上需要收集和监控最底层CPU的性能统计数据。监控的CPU统计数据包括执行特定任务所需要的CPU指令数（通常称为路径长度，Path Length），以及应用在一定负载下运行时的CPU高速缓存未命中率。虽然还有其他重要的CPU统计数据，但这两项是自底向上中最常用的。在一定负载下，应用执行和扩展所需的CPU指令越少，运行得就越快。降低CPU高速缓存未命中率也能改善应用的性能，因为CPU高速缓存未命中会导致CPU为了等待从内存获取数据而浪费若干个周期，而降低CPU高速缓存未命中率，意味着CPU可以减少等待内存数据的时间，应用也就能运行得更快。

自底向上关注的通常是在不更改应用的前提下，改善CPU使用率。假如应用可以更改，自底向上也能为如何修改应用提供建议。这些更改包括应用源代码的变动，如将经常使用的数据移到一起，使得只要访问一条CPU高速缓存行（CPU Cache Line）就能获取所有这些数据，而不用从内存获取数据。这个改动可以降低CPU高速缓存未命中率，从而减少CPU等待内存数据的时间。

现代Java虚拟机集成了成熟的JIT编译器，可以在Java应用的执行过程中进行优化，比如依据应用的内存访问模式或应用特定的代码路径，生成更有效的机器码。也可以调整操作系统的设置来改善性能，例如更改CPU调度算法，或者修改操作系统的等待时间（操作系统在将应用执行线程迁移到其他CPU硬件线程之前所花费的时间）。

如果你觉得可以用自底向上的方法，那应该先从收集操作系统和JVM的统计数据开始。监控这些统计数据可以为下一步应该关注哪些重点提供线索。第2章、第4章介绍了重点需要监控的统计数据。依据这些数据，你再判断对应用和JVM进行性能分析是否有意义。应用和JVM的性能分析可以借助于工具。Oracle Solaris SPARC、Oracle Solaris x86/x64及Linux x86/x64上可以用Oracle Solaris Studio Performance Analyzer进行性能分析。其他流行的工具如Intel VTune或AMD的CodeAnalyst Performance Analyzer在Windows和Linux上也可以提供类似的信息。这3种工具都可以收集特定的CPU计数器信息，例如Java虚拟机执行特定Java方法或功能所用的CPU指令数和CPU高速缓存未命中率。如何在自底向上中使用这些性能分析工具非常重要。你可以在第5章、第6章中找到更多如何使用Oracle Solaris Studio Performance Analyzer的信息。

1.3 选择正确的平台并评估系统性能

我们会请专家来帮助改善应用性能，他们有时会发现，性能差只是因为应用运行的CPU架构

或系统不合适。引入多核和每核多硬件线程（也称为CMT，Chip Multithreading）以后，CPU架构和系统已经发生了天翻地覆的变化，因此为特定应用选择正确的平台和CPU架构就显得尤为重要了。此外，随着CPU架构的演变，评估系统性能的方法也需要与时俱进。本节将考察现代系统中几种不同的CPU架构，并提出一些选择底层系统时的注意事项，还会解释为何现代每核多线程CPU架构（例如SPARC T系列处理器）无法使用传统评估系统性能的方法。

1.3.1　选择正确的CPU架构

Oracle的SPARC T系列处理器引入了芯片多处理和芯片多线程。SPARC T系列处理器设计上的主要亮点是引入了每核多硬件线程，以应对CPU高速缓存未命中所带来的问题。第一代SPARC T系列UltraSPARC T1，每个CPU有4、6或8个核，每核有4个硬件线程。从操作系统的角度来看，8核的UltraSPARC T1处理器就像是有32个处理器的系统。即操作系统把每个核中的每个硬件线程都看成一个处理器，所以配置为8核的UltraSPARC T1系统，操作系统会把它当成32个处理器。

UltraSPARC T1的独特之处在于每个核有4个硬件线程。在一个时钟周期内，每核4个硬件线程中只有一个可以运行。发生长延迟时，例如CPU高速缓存未命中，如果同一个UltraSARC T1核中还有其他就绪的硬件线程（Runnable Hardware Thread），下一个时钟周期就会让这个硬件线程运行。相比而言，其他每核单硬件线程（即便是超线程）的CPU，就会被诸如CPU高速缓存未命中这样的长延迟事件所阻塞，从而因等待事件完成而浪费时钟周期。对于这类CPU，如果就绪的应用线程已经准备好运行却没有可用的硬件线程，运行前就必须进行线程上下文切换。线程上下文切换通常需要耗费数百个时钟周期。由于SPARC T系列处理器可以在下一时钟周期切换到同核上的另一个就绪线程，因此，对于有许多待执行线程、高度线程化的应用来说，在SPARC T系列处理器可以执行得更快。不过，这种每核多硬件线程和下一时钟周期切换的设计代价是CPU的时钟频率比较低。换句话说，像SPARC T系列这样有多硬件线程的CPU，与其他每核单硬件线程或者无法在下一周期切换的CPU相比，运行的时钟频率通常较低。

提示

Sun公司的第一代SPARC T系列处理器，从UltraSPARC T1到 UltraSPARC T2 和T3，每核的硬件线程数依次增加，每核每时钟周期内执行多个硬件线程的能力也逐步增强。此处讨论的目的是为了便于大家探讨和理解UltraSPARC T1处理器与其他现代CPU的不同，一旦弄清楚了CPU架构上的差异，也就容易将UltraSPARC T1背后的设计思路推广到UltraSPARC T2和T3处理器。

选择硬件系统时，如果预计目标应用有大量的并发线程，那么它在SPARC T系列处理器上的性能和扩展性，就要好于每核硬件线程少的处理器。与之相比，如果应用只需少量线程，特别是预计同时运行的线程数少于SPARC T系列处理器的硬件线程数，那么它在每核硬件线程数不多但时钟频率更高的处理器上的性能，就要好于时钟频率较低的SPARC T系列处理器。简言之，要想发挥SPARC T系列处理器的性能，就需要大量并发的线程，让大量硬件线程保持负荷，从而在发

生例如CPU高速缓存未命中这样的事件时，发挥它在下一时钟周期切换到另一硬件线程的能力。如果没有大量的并发线程，SPARC T系列就和传统低时钟频率的处理器差不多了。需要同时用大量线程将许多SPARC T系列硬件线程跑满的观点也表明，传统判断系统性能是否合格的方法或许没有真正展现系统的性能。下一节将谈论这个主题。

1.3.2 评估系统性能

SPARC T系列处理器可以在下一时钟周期切换到同核中其他就绪的硬件线程，所以为了评估它的性能，必须加载大量的并发线程。

通常评估新系统性能的方法是将预期目标负载的一部分加载到系统上，或者执行一个或多个微基准测试，然后监控系统性能或单位时间内应用完成的运算量。然而，为了评估SPARC T系列处理器的性能，必须加载足够多的并发线程以便将众多硬件线程跑满。对于长时间延迟（如CPU高速缓存未命中）事件引起的下一周期线程切换，SPARC T系列需要足够大的负载才能从中受益。CPU高速缓存未命中引起的阻塞和等待会耗费许多CPU周期，大约要数百个时钟周期。因此，为了充分利用SPARC T系列处理器，系统需要加载足够多的并发任务，这样下个周期线程切换这种任务所带来的好处才能体现出来。

如果SPARC T系列处理器不能以目标负载的方式运行，系统性能反而不会很好，因为并非所有的硬件线程都满载。记住SPARC T系列处理器的主要设计点就是允许其他硬件线程在下一时钟周期执行，从而应对CPU的长时间延迟。对于每核单硬件线程的处理器来说，长时间延迟（例如CPU高速缓存未命中）意味着需要浪费大量的CPU时钟周期，以等待从内存获取数据。为了切换到另一个线程，必须用其他可运行线程及其状态信息替换当前的线程。这不仅需要时钟周期以便进行上下文切换，也需要CPU高速缓存为新运行的线程获取不同的状态信息。

因此，评估SPARC T系列处理器性能时，很重要的一点就是在系统上加载足够大的负载，从而充分利用更多的硬件线程，以及在下一时钟周期同一个CPU核内切换到另一个硬件线程的能力。

1.4 参考资料

Dagastine, David, and Brian Doherty. *Java Platform Performance*. JavaOne 大会. 美国加州旧金山，2005年.

Wilson, Steve, and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Reading, MA, Addison-Wesley, 2000. ISBN 0-201-70969-4.