

需要整本电子书，联系我QQ: [2667271557](#);
此处是样章，取的完整版的前面几页，和最后
面几页；完整版是带书签的，样章没带书签；
另外需要其他书，也可以找我。



大话 Java性能优化

周明耀 / 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

大话 Java性能优化

周明耀 / 著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书主要提供 Java 性能调优方面的参考建议及经验交流。作者力求做到知识的综合传播,而不是仅仅只针对 Java 虚拟机调优进行讲解,另外力求每一章节都有实际的案例支撑。具体包括:性能优化策略、程序编写及硬件服务器的基础知识、Java API 优化建议、算法类程序的优化建议、并行计算优化建议、Java 程序性能监控及检测、JVM 原理知识、其他相关优化知识等。

通读本书后,读者可以深入了解 Java 性能调优的许多主题及相关的综合性知识。读者也可以把本书作为参考,对于感兴趣的主题,直接跳到相应章节寻找答案。

总的来说,性能调优在很大程度上是一门艺术,解决的 Java 性能问题越多,技艺才会越精湛。我们不仅要关心 JVM 的持续演进,也要积极地去了解底层的硬件平台和操作系统的进步。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

大话 Java 性能优化 / 周明耀著. —北京: 电子工业出版社, 2016.4
ISBN 978-7-121-28481-6

I. ①大… II. ①周… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2016)第 063438 号

责任编辑: 董 英

印 刷: 北京京科印刷有限公司

装 订: 三河市皇庄路通装订厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×1092 1/16

印张: 35.25

字数: 993 千字

版 次: 2016 年 4 月第 1 版

印 次: 2016 年 4 月第 1 次印刷

印 数: 3000 册 定价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线: (010) 88258888。

序

最大的思想紊乱是相信人们想要相信的事情。

——路易斯·巴斯德 (Louis Pasteur)

Michael 周是个具有丰富程序经验的架构师和项目管理者，他从国内作坊式的软件开发公司起步，经历了著名的咨询公司凯捷的欧洲工作洗礼，后来于美国花旗软件担任高级软件技术总监，平时常常思考和总结 21 世纪以来我国软件开发者的，特别是 Java 开发工程师的困惑。

我们通常情况下，一开始可以有条不紊地进行软件需求定义和分析，随着上线时间的不断迫近，面对客户的咄咄逼人的需求修改和即刻变更需求上线压力，程序员作为弱势群体，往往会考虑时间优先原则，很难守住按部就班的开发计划和开发方式，从而导致出现了软件质量的大幅下降。**软件一定存在修改的余地**，但是程序员们通常不相信自己的系统存在诸多问题，尤其是感觉自己已经做得相当完美。系统调优在软件的后续改进和重构中占有很大的地位，能够弥补前述的不足，本书以通俗的语言和引人入胜的故事，重点讲述软件性能调优的方法论和具体实现路径，读者可以根据自己的实际情况进行参照比对，就像进了兵器库挑选合适自己的顺手武器。

程序凑合着上线是一回事，而能够优美地运行在压力下往往很不容易。本书对于所有有志于进行软件高级管理的人员而言，具有非常重要的意义。

海适云承 CEO 兼首席架构师 沈英桓 (Sam Shen)

前言

7 岁那年，当我合上《上下五千年》一套三册书籍时，我对自己说，我想当个作家。这一晃 27 年了，等待了 27 年，我的第一本书《大话 Java 性能优化》即将面世了。我是多么的忐忑、惊喜，就像第一次面对我的女儿“小顽子”，给她取这个小名，希望她顽强到底，因为我相信，你若顽强到底，一切皆有可能。

从 15 岁拥有自己第一台电脑算起，已经有接近 20 年的计算机学习时间，加上 11 年的工作经历，我对于工作，对于工程师这个职业，有一些自己的感悟。我认为，职业素养非常重要。

1929 年，在汪精卫的支持下，余云岫等人提出了全面废除中医、禁止中医的提案，并很快获得初审通过。在这样的局面下，全国各地中医师多次到南京请愿，虽有孙科等人的支持，但反响不大。相持阶段，无独有偶，汪精卫的岳母身患痢疾，西医师医治无效，京城四大名医之一的施今墨先生毅然赴汪府。施今墨凭脉，每言必中，使汪精卫的岳母心服口服，频频点头称是。处方时施今墨说：“安心服药，一诊可愈，不必复诊。”病危至此，一诊可愈？众人皆疑。据此处方仅服数剂，果如施今墨所言。汪精卫不得不服中医，最终撤回提案。施老先生医德高尚，死后遗体都捐献出来供科学研究，绝不是阿谀奉承之人，他赴汪府，完全是因为对中医生这个职业的尊重，为了让人知道中医的深奥。

戒口

佛教五戒之一的不妄语，要求我们不欺骗他人、不在不清楚实际情况的时候胡乱说话，放到职场，也可以加上信息安全的要求。

《越绝书》载文种述九术时说：“故曰九者勿患，戒口勿传，以取天下不难，况于吴乎？”文种希望勾践秘而不宣，以免人多口杂，泄露机密。每个人都有自己的岗位、职责，我们要做的是做好自己的事情，不对不属于自己工作范围内的事情评价、传播，不在背后说同事的坏话。作为一名技术人员，如果不能做到戒口、静心、专心，那我觉得你应该尽早转行，你不适合，也绝不会成为一名技术大拿。

气场

一位职业的工作者，他身上有一种称为气场的东西存在。人的气场是看不见的，但这种力量是巨大的，就像万有引力一样，我们每个人身上的这种气场无时无刻不在影响你的人生。这种气场的行程与你的观念、信仰、环境、朋友、呼吸、事物、欲望、静息与睡眠相关。一个人的气质

很好，外表精神、有修养、有道德，这个人的气场就好，就会吸引好的事，吸引好的运气。每个人都会遇到各种各样的苦难，但是我坚信，你若顽强到底，一切皆有可能。

教养

看不见的教养很难。在乌合之众中谁能保持优雅和教养？在群体无意识中谁能保持清醒和判断？更难的是那些“慎独”的教养。日本有一种文化，叫作“不给别人添麻烦”的文化，我们每个人在做事之前都应该考虑是否自己的行为会给别人造成麻烦。教养不是道德规范，也不是小学生行为准则，其实也并不跟文化程度、社会发展、经济水平挂钩，它更是一种体谅，体谅别人的不容易，体谅别人的处境和习惯。对于教养，我个人的理解是，谦逊是一种教养，自尊更是。

心态

尼克·胡哲说过，人们经常埋怨什么也做不来，但如果我们只记挂着想拥有或欠缺的东西，而不去珍惜所拥有的，那根本改变不了问题！真正改变命运的，并不是我们的机遇，而是我们的态度。

一个人的心态很是重要，心量小的人，芝麻大小的事情也能在心里翻江倒海。心量大的人，即使在危机面前也能镇静自若。同样一件事情，掀起的波澜大小却因人而异。有一句话很好，用于技术人员我觉得尤其合适，“想要成为一棵大树，就不要去和草争”。

一个人的成就，不得以金钱衡量，而是一生中，你善待过多少人，有多少人怀念你。成功并非单指事业，无论是爱好或职业上的成功都只是成就。成功应该是多元化的，如人的一生包含了很多追求一样，而非单一指向。然后，无论你多有成就，真正的成功，就是陪伴家人。所有的情感都是需要陪伴的，这些陪伴成为一个个美好的回忆，这些都是整个家庭最宝贵、最重要的财富，这些远远超越物质的重要性。在中国，因为价值观相对比较单一，社会显得很浮躁、很物质，所以大多以物质的追求为主，越多越好，内心也想过美好的生活。但当你的心完全趋向金钱的时候，很多美好的东西就会自动屏蔽了，不会出现在生活中。别让忙碌空白了回忆。

此外，作为一名技术人员，我觉得，职业生涯中可能很多次需要面对工作的变换、角色的变化，有很多知识需要学习，所以，我们应该把“归零”当成一种生活的新常态。

劝学

我觉得有一句话总结得特别好，“能干工作、干好工作职场生存的基本保障”。

荀子是儒家八派中一派的创始人，其思想学说以儒家为本，兼采道、法、名、墨诸家之长。荀子在他的著作《劝学》一文中这样写道，“君子曰：学不可以已。青，取之于蓝，而青于蓝；冰，水为之，而寒于水。”这段文字大体表达了学习是不可以停止的，君子广泛学习并且每天反省自己，就会明白道理，行为上也不会有什么过错。

全球成功的科技型企业，无论是微软的比尔·盖茨，还是苹果的乔布斯，Facebook 的扎克伯

格，无一不是技术专家，创新型企业必须由这样的企业家带队，懂技术，就会站在前沿。对于大型科技企业而言，光懂技术不够，还要懂市场。

诸葛亮在给他的儿子写的著名的《诫子书》中指出，宁静才能够修养身心，静思反省。不能够静下来，则不可以有效地计划未来，而且学习的首要条件，就是有宁静的环境。审慎理财，量入为出，不但可以摆脱负债的困扰，更可以过着简朴的生活，不会成为物质的奴隶。要计划人生，不要事事讲求名利，才能够了解自己的志向，要静下来，才能够细心计划将来。学习需要专注，平静心境才能事半功倍。学习的过程中，决心和毅力非常重要，因为缺乏了意志力，就会半途而废。拖延就不能够快速地掌握要点。时光飞逝，意志力也会随着时间消磨。

归属感

每个足球队有 11 位球员在球场上比赛，估计最不引人注目的应该是守门员了吧，他要忍受着大多数时间的无聊，还要保持着警惕。当危机发生时，很有可能还要一个人战斗，需要勇敢地面对对方前锋，唯一的目的是，绝对不让你攻破球门。我们很多时候可能也是如此，艰苦奋斗，当解决了某个问题，或是帮助公司拿到某个招标，我们都会感到自豪感、成就感，这就是归属感，对于技术领域的归属感。

最后，自我介绍一下，我叫周明耀，研究生学历，一名九三学社社员，12 年工作经验，IBM 开发者论坛专家作者。我是一名 IT 技术狂热爱好者，一名顽强到底的工程师。我推崇技术创新、思维创新，对于新技术非常热爱。

感谢我的家人，和谐的家庭帮助我完成了这本书，我的妻子，她美丽、细心、博学、偶尔不那么温柔，但是我很爱她。我的小顽子，她天生的性格很像我，希望她能够踏踏实实做人，保持创新精神，平平安安、健健康康地生活下去。感谢我妻子父母、我的父母，他们帮我照顾小孩，我才有时间编写此书。感谢浙江省特级教师、杭州高级化学老师郑克良老师，郑老师的一句“永远不要放弃”，推动着我多年的发展。感谢数学老师张老师在公开场合对我智商的褒奖，第一次收获这样的赞赏，对我这样性格的孩子是多么的重要，谢谢。感谢王芳同学，因为你的插画天赋，让这本书的内容更加丰富、可读，不要忽视了自己的才华，你很有天赋。

我相信这本书不是终点，它是麦克叔叔此生一系列技术书籍的开端，下一本书籍见。

目 录

第 1 章 性能调优策略概述	1
1.1 为什么需要调优.....	1
1.2 性能优化的参考因素.....	5
1.2.1 传统计算机体系的分歧.....	5
1.2.2 导致系统瓶颈的计算资源.....	7
1.2.3 程序性能衡量指标.....	8
1.2.4 性能优化目标	9
1.2.5 性能优化策略	10
1.3 性能调优分类方法.....	11
1.3.1 业务方面	12
1.3.2 基础技术方面	12
1.3.3 组件方面	17
1.3.4 架构方面	19
1.3.5 层次方面	20
1.4 本章小结.....	21
第 2 章 优化前的准备知识	22
2.1 服务器知识.....	23
2.1.1 内存	23
2.1.2 GPU/CPU.....	44
2.1.3 硬盘	49
2.1.4 网络架构	51
2.2 新兴技术.....	53
第 3 章 Java API 调用优化建议	54
3.1 面向对象及基础类型.....	55
3.1.1 采用 Clone()方式创建对象	55
3.1.2 避免对 boolean 判断	55
3.1.3 多用条件操作符	56

3.1.4	静态方法代替实例方法.....	56
3.1.5	有条件地使用 final 关键字.....	58
3.1.6	避免不需要的 instanceof 操作.....	58
3.1.7	避免子类中存在父类转换.....	59
3.1.8	建议多使用局部变量.....	60
3.1.9	运算效率最高的方式——位运算.....	60
3.1.10	用一维数组代替二维数组.....	62
3.1.11	布尔运算代替位运算.....	64
3.1.12	提取表达式优化.....	65
3.1.13	不要总是使用取反操作符(!).....	66
3.1.14	不要重复初始化变量.....	66
3.1.15	变量初始化过程思考.....	66
3.1.16	对象的创建、访问过程.....	69
3.1.17	在 switch 语句中使用字符串.....	70
3.1.18	数值字面量的改进.....	73
3.1.19	优化变长参数的方法调用.....	74
3.1.20	针对基本数据类型的优化.....	75
3.1.21	空变量.....	76
3.2	集合类概念.....	77
3.2.1	快速删除 List 里面的数据.....	78
3.2.2	集合内部避免返回 null.....	80
3.2.3	ArrayList、LinkedList 比较.....	82
3.2.4	Vector、HashTable 比较.....	85
3.2.5	HashMap 使用经验.....	87
3.2.6	EnumSet、EnumMap.....	91
3.2.7	HashSet 使用经验.....	92
3.2.8	LinkedHashMap、TreeMap 比较.....	96
3.2.9	集合处理优化新方案.....	99
3.2.10	优先考虑并行计算.....	107
3.3	字符串概念.....	108
3.3.1	String 对象.....	108
3.3.2	善用 String 对象的 SubString 方法.....	111
3.3.3	用 charat()代替 startswith().....	113
3.3.4	在字符串相加的时候,使用'代替'".....	114
3.3.5	字符串切割.....	114
3.3.6	字符串重编码.....	117
3.3.7	合并字符串.....	118
3.3.8	正则表达式不是万能的.....	122
3.4	引用类型概念.....	123

3.4.1	强引用 (Strong Reference)	126
3.4.2	软引用 (Soft Reference)	131
3.4.3	弱引用 (Weak Reference)	135
3.4.4	引用队列	141
3.4.5	虚引用 (Phantom Reference)	142
3.5	其他相关概念.....	146
3.5.1	JNI 技术提升.....	146
3.5.2	异常捕获机制	150
3.5.3	ExceptionUtils 类.....	154
3.5.4	循环技巧	155
3.5.5	替换 switch.....	157
3.5.6	优化循环	158
3.5.7	使用 arrayCopy().....	159
3.5.8	使用 Buffer 进行 I/O 操作.....	161
3.5.9	使用 clone()代替 new.....	164
3.5.10	I/O 速度	166
3.5.11	Finally 方法里面释放或者关闭资源占用.....	167
3.5.12	资源管理机制	167
3.5.13	牺牲 CPU 时间.....	169
3.5.14	对象操作	172
3.5.15	正则表达式	172
3.5.16	压缩文件处理	174
3.6	本章小结.....	175
第 4 章	程序设计优化建议	176
4.1	算法优化概述.....	176
4.1.1	常用算法逻辑描述.....	177
4.1.2	多核算法优化原理.....	186
4.1.3	Java 算法优化实践	188
4.2	设计模式.....	196
4.2.1	设计模式的六大准则.....	196
4.2.2	单一对象控制	200
4.2.3	并行程序设计模式.....	202
4.2.4	接口适配	205
4.2.5	访问方式隔离	219
4.3	I/O 及网络相关优化	225
4.3.1	I/O 操作优化	225
4.3.2	Socket 编程.....	231
4.3.3	NIO 2.0 文件系统	235

4.4	数据应用优化.....	236
4.4.1	关系型数据库优化.....	236
4.4.2	向 HBase 插入大量数据.....	240
4.4.3	解决海量数据缓存.....	251
4.5	其他优化.....	256
4.5.1	Web 系统性能优化建议	256
4.5.2	死锁情况解决方案.....	259
4.5.3	JavaBeans 组件.....	268
4.6	本章小结.....	269
第 5 章	Java 并程序优化建议	270
5.1	并程序优化概述.....	270
5.1.1	资源限制带来的挑战.....	271
5.1.2	进程、线程、协程.....	272
5.1.3	使用多线程的原因.....	281
5.1.4	线程不安全范例	282
5.1.5	重排序机制	284
5.1.6	实例变量的数据共享.....	286
5.1.7	生产者与消费者模式.....	288
5.1.8	线程池的使用	290
5.2	锁机制对比.....	296
5.2.1	锁机制概述	296
5.2.2	Synchronized 使用技巧.....	298
5.2.3	Volatile 的使用技巧	303
5.2.4	队列同步器	304
5.2.5	可重入锁	307
5.2.6	读写锁	308
5.2.7	偏向锁和轻量级锁.....	309
5.3	增加程序并行性.....	310
5.3.1	并发计数器	311
5.3.2	减少上下文切换次数.....	312
5.3.3	针对 Thread 类的更新	314
5.3.4	Fork/Join 框架	314
5.3.5	Executor 框架	318
5.4	JDK 类库使用	319
5.4.1	原子值	320
5.4.2	并行容器	324
5.4.3	非阻塞队列	332
5.4.4	阻塞队列	338

5.4.5 并发工具类	365
5.5 本章小结.....	376
第 6 章 JVM 性能测试及监控	377
6.1 监控计算机设备层.....	378
6.1.1 监控 CPU.....	380
6.1.2 监控内存	405
6.1.3 监控磁盘	417
6.1.4 监控网络	423
6.2 监控 JVM 活动.....	428
6.2.1 监控垃圾收集目的.....	429
6.2.2 GC 垃圾回收报告分析	430
6.2.3 图形化工具	431
6.2.4 GC 跟踪示例	437
6.3 本章小结.....	438
第 7 章 JVM 性能调优建议	439
7.1 JVM 相关概念.....	439
7.1.1 内存使用相关概念.....	440
7.1.2 字节码相关知识	443
7.1.3 自动内存管理	448
7.2 JVM 系统架构.....	451
7.2.1 JVM 的基本架构.....	451
7.2.2 JVM 初始化过程.....	453
7.2.3 JVM 架构模型与执行引擎.....	456
7.2.4 解释器与 JIT 编译器	456
7.2.5 类加载机制	457
7.2.6 虚拟机	458
7.3 垃圾回收机制相关.....	459
7.3.1 GC 相关概念	459
7.3.2 垃圾回收算法	468
7.3.3 垃圾收集器	476
7.4 实用 JVM 实验.....	490
7.4.1 将新对象预留在年轻代.....	490
7.4.2 大对象进入年老代.....	494
7.4.3 设置对象进入年老代的年龄	495
7.4.4 稳定与震荡的堆大小.....	497
7.4.5 吞吐量优先案例	498

7.4.6	使用大页案例	499
7.4.7	降低停顿案例	499
7.4.8	设置最大堆内存	499
7.4.9	设置最小堆内存	500
7.4.10	设置年轻代	503
7.4.11	设置持久代	504
7.4.12	设置线程栈	504
7.4.13	堆的比例分配	505
7.4.14	堆分配参数总结	508
7.4.15	垃圾回收器相关参数总结	509
7.4.16	查询 GC 命令	515
7.5	本章小结	515
第 8 章	其他优化建议	516
8.1	Java 现有机制及未来发展	516
8.1.1	Java 体系结构变化历史	516
8.1.2	Java 语言面临的挑战	520
8.1.3	Java 8 的新特性	522
8.1.4	Java 语言前景	523
8.1.5	物联网：Java 和你是一对	524
8.1.6	Java 模块化发展	525
8.1.7	OpenJDK 的发展	527
8.2	系统架构优化建议	528
8.2.1	系统架构调优	528
8.2.2	Java 项目优化方式分享	530
8.2.3	面向服务架构	534
8.2.4	程序隔离技术	538
8.2.5	团队并行开发准则	544
8.3	与编程无关	546
8.3.1	工程师品格	546
8.3.2	如何成为技术大牛	547
8.3.3	编程方法分享	548
8.4	本章小结	549

第 1 章 性能调优策略概述

2011 年 1 月，新加坡飞往杭州的航班。飞行持续时间很长，大约 6 个小时，坐在四周的人很快熟悉了，互相攀谈起来。有一位小姑娘，十六七岁的模样，长得很漂亮，默默地坐在座位上。热心的阿姨和她攀谈，问起她的情况，她带着疲倦自我介绍起来，“我在新加坡念初三，那所学校一点都不好，我在成都是最好的初中毕业的，也考上了成都最好的高中，但是，我的父母，他们一定要我来新加坡复读初三，让我考新加坡的高中，我一点都不喜欢这里，这里的同学看不起我们这些大陆学生，经常上课找大陆来的老师麻烦，经常辱骂我们，我烦透了!!!”对，这不是自我介绍，这是一个人接近奔溃边缘的歇斯底里。也就是在当时，我做出了决定，我绝不会让我的女儿这样远离我，一个人在很年幼的时候就必须独立面对生活的困难，绝不。无论她的父母出于什么原因让她去国外念书，我所看到的，是让一个不适合承受压力的人承担了巨大的压力，这就是本书的编写原因。在这本书里我想要和大家讨论的话题是基于 Java 语言的性能优化，我们不能随意地给出性能优化方案，就像随意指派由那位小姑娘来完成全家的未来方向一样。我们必须经过严密的研究、测试及验证，明确造成性能瓶颈真正的原因后才能开始着手，盲目地行动只会造成不必要的损失。当然，如果系统架构设计得很好，就可以在很大程度上避免类似事情发生，这不是本书的主要讨论范围。

本章主要介绍和解决以下问题，这些也是全书的基础：

- 为什么需要调优，这是您阅读本书的依据，只为需要调优而调优。
- 了解程序性能的各项指标，包括物理机器性能、程序性能。
- 性能调优分类方法，包括调优方向、调优方法、调优层次。

1.1 为什么需要调优

注意，这一节会提到许多技术名词，本着让 Java 初学者看懂本书的目的，笔者尽量第一时间做出注释，如有遗漏读者可以阅读后续章节，均有详细介绍。

经历了多年的发展，Java 已由一门单纯的计算机编程语言，逐渐演变为一套强大的技术体系平台。根据不同的技术规范，Java 设计者们将 Java 划分为 3 种结构独立但却又彼此依赖的技术体系分支，分别是 Java SE、Java EE 和 Java ME¹，其中 Java EE 被广泛使用在企业级领域，除了包括 Java API 组件外，还扩充有 Web 组件、事务组件、分布式组件、EJB 组件、消息组件等，并持续发展到现在。综合 Java EE 的这些技术，开发人员可以构建出一个具备高性能、结构严谨的企业级应用，并且 Java EE 也是用于构建 SOA 架构的首选平台。

Java 的持续发展要感谢 Google，正是 Google 将 Java 作为 Android 操作系统的应用层编程语言，使得 Java 可以在 PC 时代、移动互联网时代都得到迅猛发展，可以用于手持移动设备、嵌入式设备、个人电脑、高性能的集群服务器或大型机。

随着互联网业务的不断拓展、繁荣，越来越多的系统架构开始参照互联网企业的系统架构方式。无论是互联网、物联网，还是传统行业的软件设计，笔者认为，任何技术都离不开对业务需求的支撑²，所以开始展开研究程序性能问题之前，我们需要先了解系统业务逻辑。

铁道部的 12306³网站一直被全国人民所诟病，它确实存在一些问题，但是这些看似简单的问题，其背后牵扯着复杂的系统架构设计。这些设计最终是为业务需求服务的，即 12306 的职责是为所有旅客的需求服务的，而程序员设计的程序又是为 12306 服务的，所有的用户体验归到最终就是服务意识。我们来看一下 12306 的业务，12306 需要支持海量并发查询，即海量用户同时查时间、查车次、查座位、查铺位。此外，对应的下单过程也就会伴随着海量并发的数据库操作。据说，淘宝在双十一期间也只有几百万用户⁴，而春运期间抢购火车票是全国人民的统一活动，瞬时访问数量有千万级别甚至是亿级别的。据说 12306 的高峰访问是 10 亿 PV⁵，这些访问主要集中在早 8 点到 10 点，每秒 PV 在高峰时上千万⁶。

再来看看其他的业务系统。奥运会期间的奥运票务系统采用抽奖的方式，这样的业务设计让系统不存在先来先得抢购需求，由于是事后抽奖，因此事前只负责收集信息，所以不需要保证数据的一致性，这也就没有高强度并发锁⁷的需求，很容易通过水平扩展方式克服性能瓶颈。B2C 网站一般实时性要求不高，比如下单，用户提交订单后，订单并不是马上被处理的，而是等待一定时间后，用户才会收到订单是否确认的通知，这样就确保了数据不需要立即被处理，没有了数据高并发同步的需求。也就是说，在高并发要求下的数据一致性是通常情况下的性能瓶颈点，也是通常意义上的技术难点之一。

前面提起过，高并发情况下的数据高度实时一致性需求是很难实现的。对于一个网站来说，并发浏览网页造成的高负载较容易处理，高并发的查询负载也可以处理，但是实时下单是最难处

¹ Java SE (Java Platform, Standard Edition); Java EE (Java Platform, Enterprise Edition); Java ME (Java Platform, Micro Edition)。

² 例如金融系统，不能单纯按照程序员的思维设计系统和数据库结构，而是应该更加紧密地与业务结合。

³ 12306: 中国铁路客户服务中心 (12306 网) 是铁路服务客户的重要窗口，将集成全路客货运输信息，为社会和铁路客户提供客货运输业务和公共信息查询服务。

⁴ 来源 2013 年的网络数据，作者非阿里人士，也没有淘宝账户，所以数据不准确请读者见谅。

⁵ PV: 即页面浏览量，通常是衡量一个网络新闻频道或网站甚至一条网络新闻的主要指标。

⁶ 摘自 2013 年的官方数据。

⁷ 提高系统并发吞吐能力的方式，第 5 章会详细介绍。

理的，因为下单需要访问当前的库存量，对于 12306 网站来说，库存量就是指火车票的库存，由于这是一个全国联网系统，所以可以预见库存量保持数据一致性的难度。据说苹果 CEO 库克⁸正是因为处理好了库存问题才得以继任乔帮主⁹的宝座。目前来看，很多 B2C¹⁰网站的下单都是通过异步方式来实现的，这样的做法可以避免数据高度一致性要求。

淘宝模式相较于传统 B2C 网站有一个优势，即它不需要查询库存。B2C 网站拥有自己的仓库，每次下单前，都需要查找距离客户最近的仓库是否有库存，这样的计算量累计后会很大。比如，你在上海买一本书，如果上海附近的仓库没货，我们需要先计算哪个仓库既离上海最近又有这本书。淘宝网站由于本身商业模式的原因，它不需要去实时检查库存，反而对于性能扩展较为容易。

的确我们可以通过 Nginx¹¹来搞定每秒 10 万的静态请求，只要有足够的网络带宽、磁盘 I/O，服务器的并发计算能力够强，可以很容易地处理 10 万的并发连接。但是如果我们引入了大量的业务逻辑，那就不是单纯的访问问题了，该解决方案也就成了浮云。

除了业务需求、程序运行方式之外，程序设计本身需要考虑基础编程技术、系统架构、网络技术、操作系统、硬件服务器等诸多因素。计算机专家在问题求解时非常重视表达式简洁性的价值。UNIX 的先驱者 Ken Thompson¹²曾经说过非常著名的一句话：“丢弃 1000 行代码的那一天是我最有成效的一天之一。”这对于任何一个需要持续支持和维护的软件项目来说，都是一个当之无愧的目标。早期的 Lisp¹³贡献者 Paul Graham¹⁴甚至将语言的简洁性等同为语言的能力。这种对能力的认识让我们把编写紧凑、简洁的代码作为许多现代软件项目选择语言的首要标准。

任何程序都可以通过重构代码方式去除多余的代码或无用的占位符，例如空格，删除空格后会让代码变得更加简短。不过某些语言天生就善于表达，也就特别适合于简短程序的编写。APL 语言的设计理念是利用特殊的图形符号让程序员用很少量的代码就可以编写功能强大的程序。这类程序如果实现得当，可以很好地映射成标准的数学表达式。简洁的语言在快速创建小脚本时非常高效，特别是在目的不会被简洁所掩盖的简洁明确的问题域中。

相比于其他程序设计语言，Java 语言的冗长已经名声在外，主要原因是由于程序开发社区中所形成的惯例。在完成任务时，很多情况下要更大程度地考虑描述性和控制能力。例如，长期来看，长变量名会让大型代码库的可读性和可维护性更强。描述性的类名通常会映射为文件名，在向已有系统中增加新功能时会显得很清晰。如果能够一直坚持下去，描述性名称可以极大简化用于表明应用中某一特定的功能的文本搜索。实践证明，这些定义方式让 Java 在大型复杂代码库的

⁸ 蒂姆·库克 (Tim Cook)，1960 年 11 月 1 日出生于美国阿拉巴马州，1982 年毕业于奥本大学工业工程专业。1988 年获得杜克大学企业管理硕士学位。曾在 IBM 供职 12 年，负责 PC 部门在北美和拉美的制造和分销运作。1998 年年初，库克进入苹果，任副总裁，主管苹果的电脑制造业务。2011 年接替乔布斯担任苹果公司 CEO。

⁹ 即乔布斯，1955 年 02 月 24 日—2011 年 10 月 05 日。

¹⁰ B2C: Business To Customer。

¹¹ 一个高性能的 HTTP 和反向代理服务器，也是一个 IMAP/POP3/SMTP 服务器。

¹² 肯·汤普森 (Kenneth Lane Thompson, 1943 年 2 月 4 日)，一般称之为 Ken Thompson，为美国计算机科学学者，与丹尼斯·里奇同为 1983 年图灵奖得主。

¹³ LISP 是一种通用高级计算机程序语言，长期以来垄断人工智能领域的应用。Lisp 作为因应人工智能而设计的语言，是第一个函数式程序设计语言，有别于 C、Fortran 等命令式程序设计语言和 Java、C#等面向对象语言。

¹⁴ 保罗·格雷厄姆 (Paul Graham)，美国著名程序员、风险投资家、博客和技术作家。他以 Lisp 方面的工作而知名，也是最早的 Web 应用 Viaweb 的创办者之一，后来以近 5 千万美元的价格被雅虎收购，成为 Yahoo! Store。

大规模实现中取得了极大的成功。

相对于传统的 32 位虚拟机，64 位虚拟机所具备的最大优势就是可以访问大内存。32 位虚拟机最大可用内存空间被限定在了 4GB，并且 Java 堆区的大小配置存在最大限制，如果是在 Windows 平台下最大只能设置到 1.5GB，而在 Linux 平台下最大也只能设置到 2GB~3GB。也就是说，Java 堆区的内存大小设置还需要依赖于具体的操作系统平台。

既然 32 位虚拟机无法满足大内存消耗的应用场景，那么 64 位虚拟机的出现则是顺理成章的。64 位虚拟机之所以能够访问大内存，是因为其采用了 64 位的指针架构，这也是寻址访问大内存的关键要素。

在 JDK1.6 Update14 版本之前，64 位虚拟机的综合性能表现实际上是不如 32 位虚拟机的，这主要是因为 OOPS（Ordinary Object Pointers，普通对象指针）从 32 位膨胀到 64 位后，CPU Cache Line 中的可用 OOPS 变少，这样一来就会直接影响并降低 CPU 的缓存使用率，这就是 64 位虚拟机在性能上之所以落后于 32 位虚拟机的主要原因。其次，由于部署在 64 位虚拟机上的性能都需要用到大内存，尤其是互联网项目，经常需要使用多达几十乃至几百 GB 的内存，这对于传统的 32 位虚拟机将无法承载，只能依靠 64 位虚拟机去支撑。但是管理这么大的内存开销对于 GC（Garbage Collection）来说将会是一场非常严峻的考验，甚至很有可能会导致 GC 在执行内存回收期间消耗更长的时间，同时也意味着工作线程的等待时间将会延长。如今随着 64 位虚拟机的逐渐成熟，指针压缩将会通过对齐补白等操作将 64 位指针压缩为 32 位，以此改善 CPU 缓存使用率达到提升 64 位虚拟机运行性能的目的。

对于小型项目来说，简洁性则更受青睐，某些语言非常适于短脚本编写或者在命令提示符下的交互式探索编程。Java 作为通用性语言，则更适用于编写跨平台的工具。在这种情况下，“冗长 Java”的使用并不一定能够带来额外的价值。虽然在变量命名等方面，代码风格可以改变，不过从历史情况来看，在一些基本的层面上，与其他语言相比，完成同样的任务，Java 语言仍需更多的字符。为了应对这些限制，Java 语言一直在不断地更新，尝试包含一些通常称为“语法糖”的功能。用这些习语可以实现用更少的字符表示相同功能的目标，与其对应的更加冗长的配对物相比，这些习语更受程序开发社区的欢迎，也会被社区作为通用用法快速地采用。

现代 CPU 架构将多核、多硬件执行线程技术推向前台，这意味着我们可以利用更多的 CPU 资源做更多的工作。然而，要利用好这些额外的 CPU 资源，运行于其上的程序必须要能够支持并行工作这个需求。通俗点讲，这些程序需要按照多线程的方式构造或设计才能充分地利用额外的硬件线程。

最近这几年，服务器端网络使用的基础通信技术并没有取得太大的进步。服务器端大多数在绝不允许服务中断的关键任务环境中，新技术很难渗透，也很难植根于这样的环境。但正因如此，服务器端的多余部分才得以被剔除，逐渐地形成了非常精简单纯的风格。网络的基础技术可以说已经成型了，然而在网络上运行的网络设备和服务器的技术仍然踩着现在进行时的节奏在持续不断地爆发性发展，由此出现了虚拟技术和网络存储技术等基于网络的创新技术。如今，它们已经在系统中不可或缺。随着这些技术的发展，人们追求的网络形态和网络设计的方式也在时刻发生着变化，基础架构工程师和服务器工程师必须能灵活应对这些变化才行。对应地，软件设计程序员也需要有针对性地做出应对措施。

虚拟化技术是一种资源管理技术，是将计算机的各种实体资源，如服务器、网络、内存及存储等，予以抽象、转换后呈现出来。此举打破了实体结构间的不可切割的障碍，使用户可以用比原本形态更好的方式来应用这些资源。虚拟化资源一般包括计算能力和资料存储介质，这些资源的虚拟部分不受现有资源的架设方式、地域或物理形态所限制。虚拟化技术在带来成本节省和运维便利的同时，也带来了一些新的挑战，对架构师、运维人员、程序员等角色提出了新的要求。在解决了物理设施的虚拟化问题之后，我们的目光可能会转移到应用程序本身上来，因为这才是真正为用户体现支付价值的关键所在。特别需要注意的是，部署在虚拟化环境上的 Java 应用与物理环境上的应用存在明显的区别。

综上所述，性能优化本身对于程序性能是至关重要的，同时性能优化也是一门综合性课程，虽然本书针对的是 Java 程序的性能优化，但是依然需要考虑综合性因素。作者认为，随着 IT 技术的蓬勃、快速发展，性能调优已经不单纯是代码级别的调优，它是一个对综合性知识的深入理解需求，我们只有结合多方面的技术才能真正找到合理的解决方案。这也是本书除了深入介绍 Java 程序调优、JVM 调优等之外，坚持引入服务器、网络、云计算、虚拟化等多维技术点的原因。

1.2 性能优化的参考因素

系统性能优化，或者称之为程序性能优化，它存在的理由有很多。举一个政治上的例子，意大利由于天生的漫长海岸线，所以它一直以来都是难民逃亡欧洲的跳板。意大利政府一直都受困于这个难民潮问题，不管阻拦还是放行难民，这些举措都会受到北欧国家的指责。后来意大利政府从很多难民口中知道他们其实想去德国，只不过路过这里，所以干脆就来一招狠的，让难民填写意愿国家，只要填写了就直接大巴送到国境线上去。这样一来，德国吃紧了，一下子很多难民涌入，就造成了整个国家的运转问题。就好似计算机程序的性能问题，面对海量数据或者任务时，无论如何你都会碰到性能压力，唯一的选择是你会把这个压力放在哪一层或者哪一个位置来应对，以及采取什么应对措施。下面开始具体解释这些造成性能问题的因素点。

1.2.1 传统计算机体系的分歧

如果说图灵¹⁵奠定的是计算机的理论基础，那么冯·诺依曼¹⁶则是将图灵的理论物化为实际的物理实体，成为了计算机体系结构的奠基者。从第一台冯·诺依曼计算机诞生到今天已经过去了将近 70 年，计算机的技术与性能也都发生了巨大的变化，但整个主流体系结构依然是冯·诺依曼结构。

冯·诺依曼体系结构是采用二进制形式存储数据，硬件由 5 个部分组成，分别是运算器、控制器、存储器、输入设备和输出设备。同时提出了“存储程序”原理，即使用同一个存储器，然后经由同一个总线传输，程序和数据统一存储，同时在程序控制下自动工作。特别要指出，它的

¹⁵ 艾伦·麦席森·图灵 (Alan Mathison Turing)，生于 1912 年 6 月 23 日，逝于 1954 年 6 月 7 日，被誉为“计算机科学之父”和“人工智能之父”。图灵和同事破译的情报，在盟军诺曼底登陆等重大军事行动中发挥了重要作用，图灵因此在 1946 年获得“不列颠帝国勋章”。历史学家认为，他让二战提早了 2 年结束，拯救了至少 1400 万人的生命。

¹⁶ 冯·诺依曼 (John von Neumann, 1903—1957)，20 世纪最重要的数学家之一，在现代计算机、博弈论和核武器等诸多领域内有杰出建树的最伟大的科学全才之一，被称为“计算机之父”和“博弈论之父”。

程序指令存储器和数据存储器是合并在一起的，程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置。因为程序指令和数据都用二进制码表示，且程序指令和被操作数据的地址又密切相关，所以几十年前选择这样的结构是合理的。

但是，随着对计算机处理速度要求的提高和对需要处理数据的种类、量级的需求不断增大，这种指令和数据共用一个总线的结构，使得信息流的传输成为限制计算机性能的一个瓶颈，制约了数据处理速度的提高。由此，体现出了冯·诺依曼体系结构的局限性，如下面 4 点：

- (1) 目前 CPU 的处理速度和内存容量的增长速率要远大于两者之间的流量，将大量数值从内存搬入搬出的操作占用了 CPU 大部分的执行时间，也造成了总线的瓶颈；
- (2) 程序指令的执行顺序是串行的，由程序计数器控制，这样使得即使有关数据已经准备好了，也必须遵循逐条执行指令序列，这样的设计影响了系统运行的速度；
- (3) 存储器是线性编址，按顺序排列的地址访问，这样设计有利于存储和执行机器语言，适用于数值计算。高级语言的存储采用的是一组有名字的变量，是按名字调用变量而不是按地址访问，且高级语言中的每个操作对于任何数据类型都是通用的，不管采用何种数据结构，多维数组¹⁷、二叉树¹⁸还是图，最终在存储器上都必须转换成一维的线性存储模型进行存储。这些因素都导致了机器语言和高级语言之间存在很大的语义差距，这些语义差距之间的映射大部分都要由编译程序来完成，在很大程度上增加了编译程序的工作量；
- (4) 冯·诺依曼体系结构计算机是为逻辑和数值运算而诞生的，它以 CPU 为中心，I/O 设备与存储器间的数据传送都要经过运算器，在数值处理方面已经达到很高的速度和精度，但对非数值数据的处理效率比较低，需要在体系结构方面有革命性突破。

科学家们一直在努力突破传统的冯·诺依曼体系结构框架，对冯·诺依曼计算机进行改良，主要体现在以下 3 点：

- (1) 将传统计算机只有一个处理器串行执行改成多个处理器并行执行，依靠时间上的重叠来提高处理效率，形成支持多指令流、多数据流的并行算法结构；
- (2) 改变传统计算机控制流驱动的工作方式，设计数据流驱动的工作方式，只要数据准备好了，就可以并行执行相关指令；
- (3) 跳出采用电信号二进制范畴，选取其他物质作为执行部件和信息载体，如光子、量子或生物分子等。

近几年，在计算机体系结构研究方面也已经有了重大进展，越来越多的非冯式计算机相继出现，如光子计算机、量子计算机、神经计算机以及 DNA 计算机等。

光子计算机 (Photonic Computer) 是一种采用光信号作为物质介质和信息载体，依靠激光束进入反射镜和透镜组成的阵列进行数值运算、逻辑操作和信息的存储和处理。它可以实现对复杂度、计算量大、实时性强的任务的高效、并行处理，比普通电子计算机快 1000 倍，在图像处理、

¹⁷ 二维数组以上的数组，既非线性也非平面的数组。

¹⁸ 在计算机科学中，二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”(left subtree)和“右子树”(right subtree)。二叉树常被用于实现二叉查找树和二叉堆。

模式识别和人工智能方面有着非常巨大的应用前景。

神经计算机（Neural Computer）是一种可以并行处理多种数据功能的神经网络计算机，它以神经元为处理信息的基本单元，将模仿大脑神经记忆的信息存放在神经元上。神经网络具有自组织、自学习、自适应及自修复功能，可以模仿人脑的判断能力和适应能力。美国科学家研究出的神经计算机可以模拟人的左脑和右脑，能识别语言文字和图形图像，能控制机器人行为，进行智能决策。它的左脑由 100 万个神经元组成，用于存储文字和语法规则，右脑由 1 万多个神经元组成，适用于图形图像识别。这将有可能会成为人工智能硬件发展的主攻方向。

量子计算机（Quantum Computer）是一种遵循量子力学规律进行高速数学和逻辑运算、存储及处理量子信息的物理装置。量子计算机本身的特性，扩充了逻辑和数学理论，通过核自旋、光子、束缚离子和原子等制成的量子位，创造出了经典条件下不可能存在的新的逻辑门。与经典的比特位不同，对量子位操作 1 次等同于对经典位操作 2 次，因为量子不像半导体只能记录 0 和 1，它可以同时表示多种状态。这些都为新的算法实现提供了条件，也为人工智能的发展提供了可能的硬件条件。

我们不可否认，冯·诺依曼计算机以其技术成熟、价格低廉、软件丰富和大众的使用习惯，可能在今后很长的一段时期里还将为人类的工作和生活发挥着重要作用。但是，为了满足人们对计算机更快速、更高效、更方便的使用要求，为了让计算机能够模拟人脑神经元和脑电信号脉冲这样复杂的结构，我们需要突破现有的体系结构框架并寻求新的物质介质作为计算机的信息载体，这样才能使计算机有质的飞跃。相信未来随着非冯式计算机的商业化推进，我们将会迎来一个崭新的信息时代。

1.2.2 导致系统瓶颈的计算资源

根据应用程序的不同特点，任何计算机部件都有可能成为系统瓶颈爆发点。其中，最有可能成为系统瓶颈的计算资源如图 1-1 所示，包括 CPU、内存、磁盘、网络、数据库等。

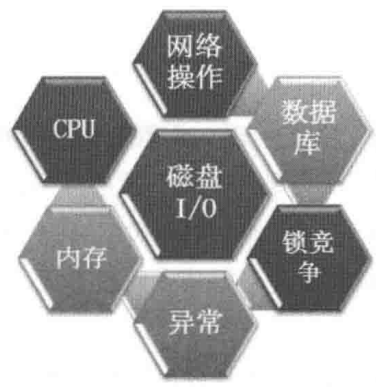


图1-1 系统瓶颈原因

- **CPU：**对计算资源要求较高的应用，由于其长时间、不间断地大量占用 CPU 资源，这样对 CPU 的争夺将导致性能问题。如视频分析、科学计算、3D 渲染等应用场景都对 CPU 资源需求较大。
- **内存：**一般来说，只要应用程序设计合理，内存在读写速度上不太可能成为性能瓶颈，除非应用程序进行了高频率的内存交换和扫描，但这些情况比较少见。内存制约系统性能的最可能发生的情况是内存大小不足，这种情况下会导致应用程序无法创建对象，更严重甚

至导致操作系统无法正常运行。与磁盘相比，内存的大小较小，这意味着应用软件只能尽可能将常用的核心数据读入内存，大量的数据还是需要存放在磁盘上，这个特性在一定程度上降低了系统性能。

- **磁盘 I/O**：磁盘 I/O 读写速度比内存慢很多。随着硬盘技术的不断发展，SSD¹⁹固态硬盘的引入确实已经加快了磁盘的读写速度，但是性价比不高，速度也还是慢于内存。因为读写性能原因，程序在运行过程中，如果需要等待磁盘 I/O 完成，那么低效的 I/O 操作会拖累整个系统。
- **网络传送**：对网络数据进行读写的情况与磁盘 I/O 类似。由于网络环境的不确定性，尤其是对互联网上数据的读写，网络操作的速度可能比本地磁盘 I/O 更慢。因此，如果不加特殊处理，也极有可能成为系统瓶颈。
- **数据库**：大部分应用程序都离不开数据库，无论是关系型数据库，还是列式数据库，它们都存在连接数量、读写速度、数据合并等制约因素，而针对海量数据的读、写操作则可能更加耗费时间。应用程序可能需要等待数据库操作完成或者等待返回检索请求需要的结果集，即这类同步操作容易成为系统瓶颈。我们可以通过一些异步操作、多数据中心等方式来解决局部问题，但是无法解决所有由于数据库导致的问题。总的来说，数据库是最容易导致应用程序性能瓶颈的原因之一。
- **锁竞争**：对高并发程序来说，如果存在激烈的锁竞争，无疑是对性能极大的打击。锁竞争将会明显增加线程上下文切换的开销，而且这些开销都是与应用需求无关的系统开销，导致大量占用宝贵的 CPU 资源，却不带来任何好处。总的来说，锁竞争问题对于程序员来说最难处理，因为处理这方面问题需要大量的操作系统、编程语言并发知识及经验。
- **异常**：对 Java 应用来说，异常的捕获和处理是非常消耗资源的。如果程序高频率地进行异常处理，则整体性能便会有明显下降。高级编程语言一般都提供异常捕获及处理机制，相对来说程序员比较容易掌握。

1.2.3 程序性能衡量指标

如果抛开所有的内部技术因素，我们只看应用程序的性能指标，那么一般来说，程序的性能大体可以通过以下几个方面来衡量。

- **响应时间**：系统对用户行为或者事件做出响应的的时间。响应时间越短，性能一定越好，所以我们在系统设计过程中应该尽量采用异步处理方式，让用户能够尽快收到回执，这样用户体验会较好。
- **启动时间**：应用系统从运行到可以正常处理业务所需要花费的时间，对于用户来说，肯定是越快启动越好，所以我们在系统设计过程中应该尽量采用异步加载数据的方式启动应用程序，避免等待所有数据加载完毕后才启动。
- **执行时间**：一段代码从开始运行到运行结束，所使用的时间称为执行时间。对于执行时间，

¹⁹ Solid State Drives，简称固态硬盘，固态硬盘（Solid State Drive）用固态电子存储芯片阵列而制成的硬盘，由控制单元和存储单元（FLASH 芯片、DRAM 芯片）组成。

有些时候可能无法减少全局化的时间，但是可以通过把业务逻辑切分到多段连续的程序段中，让用户感觉执行时间减短了。

- **执行速度**：程序的反应是否迅速，响应时间是否足够短。该指标与响应时间、执行时间是相关联的。
- **计算资源分配**：计算资源，包括 CPU、内存、磁盘等，如果其中的任何一项分配不合理，可能会导致整个系统始终处于计算资源紧张的情况下，这样对于整个系统的性能影响一定是毁灭性的。
- **内存分配**：内存分配是否合理，是否过多地消耗内存或者存在泄漏，JVM 性能也与内存分配有一定关系。
- **磁盘吞吐量**：描述 I/O 的使用情况。IOPS (Input/Output Per Second) 即每秒的输入输出量（或读写次数），是衡量磁盘性能的主要指标之一。IOPS 是指单位时间内系统能处理的 I/O 请求数量，I/O 请求通常为读或写数据操作请求。随机读写频繁的应用，如 OLTP (Online Transaction Processing)，IOPS 是关键衡量指标。另一个重要指标是数据吞吐量 (Throughput)，指单位时间内可以成功传输的数据数量。对于大量顺序读写的应用，如 VOD (Video On Demand)，则更关注吞吐量指标。每秒 I/O 吞吐量 = IOPS × 平均 I/O SIZE。从公式可以看出，I/O SIZE 越大，IOPS 越高，那么每秒 I/O 的吞吐量就越高。因此，我们会认为 IOPS 和吞吐量的数值越高越好。实际上，对于一个磁盘来讲，这两个参数均有其最大值，而且这两个参数也存在着一一定的关系。
- **网络吞吐量**：描述网络的使用情况。网络中的数据由一个个数据包组成，防火墙对每个数据包的处理要耗费资源。吞吐量是指在没有帧丢失的情况下，设备能够接受的最大速率。其测试方法是：在测试中以一定速率发送一定数量的帧，并计算待测设备传输的帧，如果发送的帧与接收的帧数量相等，那么就将发送速率提高并重新测试；如果接收的帧少于发送的帧则降低发送速率重新测试，直至得出最终结果。吞吐量测试结果以“比特/秒”或“字节/秒”表示。
- **负载承受能力**：当系统压力上升时，系统的执行速度、响应时间的上升曲线是否平缓。负载承受能力与计算资源、内存、磁盘、网络等多方面因素都有关联。

1.2.4 性能优化目标

与前面章节不同，我们这里抛开所有的外部因素，只单纯分析 Java 程序本身，那么大多数 Java 程序性能优化目标都可以归纳到下面几类。

- **编写更有效率的代码**：应用程序的每 CPU 指令时钟周期 (Clocks Per Instruction, CPI) 指的是执行一条 CPU 指令所消耗的 CPU 时钟数。编译器将 Java 源代码编译成字节码，而 CPI 正是被用来衡量字节码效率的指标。由于应用程序的最终执行基于字节码，所以调整应用程序、Java 虚拟机或操作系统，缩短应用程序的 CPI，让编译器生成更优的指令等，这些举措都有助于提升应用程序的性能。执行路径长度与 CPI 之间有微妙的差别，执行路径长度与应用程序的算法选择关系密切，而 CPI 与编译器生成更有效的代码有关。前者着眼于通过选择合理的算法生成最短的 CPU 指令序列，后者着眼于让编译器生成最高效的代码，减少每条 CPU 指令上消耗的 CPU 时钟周期数。如果 CPU 时钟周期被用来执行操

作系统指令或内核代码，那么这部分时钟周期就无法用于执行应用程序。因此，改善应用程序性能的策略之一是减少消耗在系统或内核 CPU 上的时钟周期数。注意，该策略不适用于在操作系统或内核态上消耗时间极少的应用程序。

- **使用更高效的算法：**对应用程序进行性能调优所获得的最大收益来自于算法效率的提高。高效的算法主要可以在业务逻辑处理层面起到重要的作用，可以让应用程序使用更少的 CPU 指令、更短的执行路径来实现程序功能。通常情况下，拥有更短执行路径的应用程序运行得更快。从应用程序来看，使用更好的数据结构或者改进算法可以构造出更短的执行路径，很多应用程序的性能问题都源于使用了不合适的数据结构。因此，使用恰当的数据结构及算法是提升程序性能最有效的方法。在性能分析过程中，我们要充分注意程序使用的数据结构及算法，尽可能采用更优的方式，这样做才能最大限度地提高程序性能。
- **减少锁竞争：**对共享资源的竞争容易限制应用程序的可扩展性。频繁进行锁竞争的应用程序的性能是无法随线程数和 CPU 数增加而提高的，因此，减少锁竞争的频率、缩短锁持有的时间都能够有效地优化应用程序。

1.2.5 性能优化策略

上面各子章节对性能调优可能出现的位置、性能衡量指标、优化目标等进行了一些总结，下面列举几点常见的性能优策略。

- **用空间换时间。**该策略属于系统架构层面的优化。我们知道，各种缓存机制，从 CPU L1/L2/RAM 到硬盘，都可以通过空间换时间的策略。这类策略基本上是通过采用把计算的过程一步一步地保存或者缓存等方式，避免重复计算的发生。具体的方式可以采用数据缓冲、CDN 等。类似的策略还表现为诸如冗余数据，比如数据镜像、负载均衡等。
- **用时间换空间。**该策略也属于系统架构层面的优化。有时候使用少量的空间，可能性会更好。比如网络传输，如果有一些压缩数据的算法，例如 Huffman 编码压缩算法²⁰，该算法本身运行过程其实很耗时，但是因为整体来看性能瓶颈在网络传输部分，所以用时间来换空间反而能省时间。
- **简化代码。**该策略属于基础技术层面的优化。最高效的程序就是不执行任何代码的程序，所以，大多数情况下我们可以认为代码越少性能就越高。关于代码级优化的技术，大学的教科书中有很多示例了。比如，减少循环的层数、减少递归、在循环中少声明变量、少做分配和释放内存的操作、尽量把循环体内的表达式抽到循环外、条件表达式中的多个条件判断的次序、尽量在程序启动时把一些东西准备好、注意函数调用的开销（栈上开销）、注意面向对象语言中临时对象的开销、小心使用异常（不要用异常来检查一些可接受可忽略并经常发生的错误），等。这类知识需要我们非常了解编程语言和常用的语言自带资源库。从根本上来讲，不一定越少的代码越好，我们需要了解 Java 本地库的实现原理，例如同步问题，如果单纯采用同步锁（Synchronized）的方式，代码确实很少，但是实际的性

²⁰ 一种经典的压缩算法。于 1952 年问世，迄今为止经久不衰，广泛应用于各种数据压缩技术中，且仍不失为熵编码中的最佳编码方法，deflate 等压缩算法也结合了 huffman 算法。

能并不好，当大量线程同时访问代码块时，它会生成大量的锁，导致系统内部代码块互相等待，具体我们会在第 5 章详细介绍。所以，我们还是要有技术基础之后才能做到代码上的简化。

- **并行处理。**该策略属于一种综合性策略。试想，如果 CPU 只有一个核，你要是还采用多进程、多线程的方式编写代码，那么计算密集型需求的应用程序反而会更慢，这是由巨大的操作系统调度和切换开销所导致的，这类型优化策略只能依靠多核 CPU 才能真正体现出多进程多线程的优势。实际应用过程中，我们会针对不同 CPU 进行大量高并发、密集型任务测试，不一定核数多就一定是最佳选择，也有些情况需要单核能力强的 CPU，然后采用资源隔离技术对 CPU 核上的计算资源进行切分，把应用程序线程部署到不同的隔离区域，这样可以保证更多的并行程序同步进行，这也与整个系统的架构、内存共享策略、任务调度机制等多方面因素相关联。并行处理需要我们的程序拥有扩展性，不能水平或垂直扩展的程序无法进行并行处理。

综上所述，我们把经典的二八原则²¹移植到性能优化上来看，如图 1-2 所示，统计学认为，20%的系统设计或程序代码消耗了 80%的系统性能。如果我们找到那 20%的缺陷设计或者劣质代码，那么我们就可以较为容易地优化、解决 80%的性能问题。

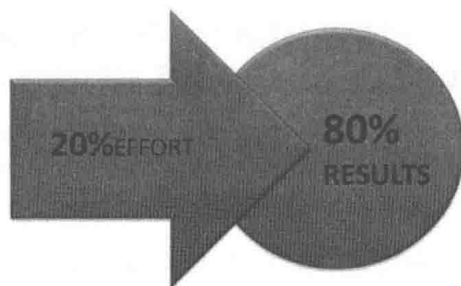
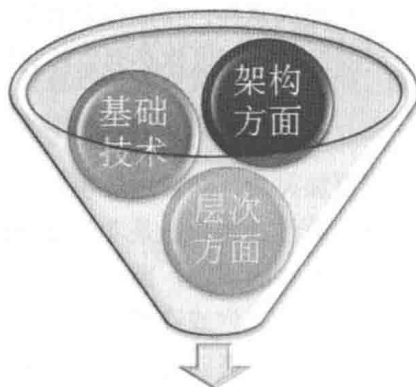


图1-2 二八原则

1.3 性能调优分类方法

性能调优方法一般来说可以分为基础技术、架构、层次这三个方面，如图 1-3 所示，解决了这三个方面的问题，沙漏才能正常工作。



性能调优分类方法

图1-3 性能调优分类方法

²¹ 即巴莱多定律，是 19 世纪末 20 世纪初意大利经济学家巴莱多发现的。他认为，在任何一组东西中，最重要的只占其中一小部分，约 20%，其余 80% 尽管是多数，却是次要的，因此又称二八定律。

1.3.1 业务方面

1.3.1.1 商业事务

商业事务是真实用户体验的直观反映，它们在抓取用户与应用交互时用户体验到的实时性能数据。测量商业事务的性能，需要抓取一件商业事务整体的响应时间及其各个组件的响应时间。这些响应时间再与满足业务需求的基准进行比较，从而决定应用是否正常。

商业事务通过其入口进行辨别，即它是用户与你的业务进行互动的入口。这类互动包括，一个网页请求、一个网页服务调用、消息队列中的一条消息等。当然，你也可以基于一个 URL 参数为同样的网页请求定义多个入口，或基于一个服务调用的内容定义多个入口点。关键在于，商业交易必须与你的业务流程相关联，比如说中国移动的空中缴费业务对应到系统中是多个原子服务，我们就应该将这几个原子服务通过相应的关联聚合成一个空中缴费业务来进行监控。

每个商业交易的性能会与其基准进行比较，判定其是否正常。譬如，如果某个商业事务的响应时间大于你设定的阈值，我们便判定其运行异常。

总而言之，商业事务最能反映用户体验，因此它们也是最重要的抓取维度。

1.3.1.2 外部服务

外部服务的形式多种多样，包括从属的网页服务、遗留系统或数据库等。外部服务是与应用交互的系统。运行在外部服务系统中的代码常常无法控制，但是我们可以控制这些系统的配置，因此了解他们是否运行正常以及何时出错也很重要。此外，我们必须有能力区分问题是出自应用本身，还是出自这些外部服务系统。

从商业事务的角度来说，我们可以辨别并测量这些处于自身应用的外部服务。例如，我们需要配置监控方法从而辨别那些包裹了外部服务调用的方法。但是对于常见的协议，诸如 HTTP 和 JDBC，外部服务可以自动检测。

商业事务让你对应用的性能有了全局的掌控，帮助你对性能问题进行分类。但是外部服务总能以意想不到的方式极大地影响应用的运行，所以你必须监控它们。

1.3.1.3 应用布局

因为云的出现，现在的应用变得更加灵活，即应用环境可以根据用户需求调节大小。因此，对应用的布局进行检测从而决定实例的数量是否合适是非常重要的。如果你的应用实例太多，你的云主机成本就会增加。但如果你没有足够的实例，商业事务就会受到影响。具体来说，你需要确定是否有应用中的实例负载过大，如果有，你或许想在那个应用中添加实例。从应用的角度查看实例状态很重要，因为单个实例可能由于垃圾回收之类的因素负载过大，但如果应用中大多数实例都负载过大，则该应用可能已经无法支持它接受的访问量。

1.3.2 基础技术方面

无论你现在正在专注于前端软件技术，还是后端软件技术，所有的语言都需要使用到编译器，那么一般来说编译器是如何工作的呢？

编译器在开始工作之前，需要知道当前的系统环境，比如标准库在哪里、软件的安装位置在哪里、需要安装哪些组件等。这是因为不同计算机的系统环境不一样，通过指定编译参数，编译

到晚学习，但是想要成为技术大牛，只有不断学习，让技术成为你的爱好，没有捷径。

- (3) 天赋，就是所谓的聪明。个人认为，好的程序员通常可能是你认识的人里最聪明的那个，而且出乎意料的，好的程序员可能不是我们通常想象的那样不善言辞。个人感觉，好的程序员一般逻辑思维能力很强，可能知识领域会跨域多个领域，比如浙江大学已故的陈教授天洲，心理学、计算机学都非常精通，即便患病期间还在国际顶级期刊上发表了 3 篇顶级的医学文章，陈教授安息!!!
- (4) 好的程序员通常有自己的私人的一些研究、爱好、项目，而这些大多数情况下他们不会写在简历上，但表现出来却可能恰恰是他的潜能、深度和后劲所在。
- (5) 技术多样性、先进性，在多种技术方向都有涉及，而且对多种技术的优劣有个人独到的见解，喜好尝试新鲜技术。云计算、大数据技术不就是这样产生的吗。
- (6) 基本上还是忽略证书吧，高手哪有时间去考证书，有这个闲工夫还不如多做点东西出来，多写点文章呢。

当然，以上 6 点可能有点片面，请读者自己多多体会，也欢迎微信讨论，本文作者微信号 michael_tec，欢迎添加。

8.3.2 如何成为技术大牛

当一个程序员的技术能力和解决问题的能力达到一定水平之后，他就能够轻松胜任某些开发任务、解决特定实际问题，从而给用户带来某方面的便利。他的能力与他接触到的问题匹配，此时程序员处于工作满意度较高的状态，这个舒适区的大小是由他解决问题能力的大小定义和界定的。什么时候挑战的情况开始发生呢？当问题超出程序员先有技能和经验说明，程序员能看到并了解，但还不能解决，这个时候随时都可能给程序员带来挑战的感觉。这还只是挑战而已，如果我们引入了一个全新的编程语言，需要让程序员在有限时间内快速掌握该技能，那么这时才会进入真正的挑战区域，为了与前面的挑战区相区别，我们定义它为未知区域。对大多数程序员来讲，未知即痛苦。这个区域往往是程序员看不清或看不到的，是百慕大三角的一片未知而神秘的区域，贸然跳入，可能折戟沉沙铩羽而归。假如一个程序员愿意跳出舒适区，踏入挑战区，接受一定的不适，那他就能够有机会拓展他的能力，将自己的舒适区扩展得更大，他的舒适区就会变大，挑战区也会跟着变大，未知区也会变大，这也是符合人类认知规律的：知道得越多，未知的也越多。如果一个程序员连轻微不适都不愿意接受，那他就会渐渐故步自封，落后于别人，落后于时代，渐渐被这个日新月异的时代所抛弃，成为一个别人眼中没什么用的老家伙。

一个程序员的能力，是可以通过锻炼不断变强的。就像人的肌肉，一段时间让锻炼强度超负荷一点，肌肉变得比原来强了，就再超负荷一点，通过这样的螺旋式递进，肌肉就会越来越强。程序员也是一样的，你的学习能力、代码能力、设计能力、沟通能力、管理能力等，都是可以通过锻炼来加强的（但我们也得考虑一个人适合做什么，如果他没有某方面的才干，虽然通过锻炼也可以加强，但违背天性的事儿通常为事倍功半）。在软件开发过程中，一个程序员，他会什么语言懂什么框架水平如何，自己心里有数，项目经理通过他的表现也认为自己心里有数。那么在新的项目要做时，通常的做法是，哪个程序员熟悉实现 Tx 任务相关的技术，就让这个程序员做 Tx 任务，这通常又是出于交付期、生产率、成本等各方面的考虑。在这种情况下，每个人都做自己驾轻就熟的事情，对整个项目来讲，自然是最经济的。可是对程序员自己来讲，却是不经济的。

因为你无法接受新的挑战，你的能力边界的拓展就会很慢。所以，合理的情况是，项目经理在划分任务时，要对程序员负责，既给一个程序员能轻松完成的任务，也要给他需要费点儿劲儿才能完成的任务，通过具有挑战性的任务来锻炼这个程序员，让他更好更快的成长。但是这样做的管理成本太高，所以，现实当中，很少公司的项目经理会主动这么做（没合适人手 take 某个任务时会被动这么做）。鉴于这种现实，作为程序员自己，如果你想更快地成长，就要表现得勇敢一些，主动走到挑战区域，去抢具有挑战性的任务。一旦你拿到了对你来讲具有挑战性的任务，虽然你会为此殚精竭虑，虽然你可能为此加班，虽然你可能为此在别人看不见的地方付出，但是你拥有了机会和更多可能性，如果你顺利完成了，那你的舒适区会扩大，你接触新挑战的机会也会变大，你就进入了良性循环，你会越来越强大。So，某个技术没搞过？不是问题。某个语言没学过？不是问题。软件结构太复杂，一时掌控不了？不是问题。业务不熟悉？不是问题。

在渴望成就自我的程序员眼里，问题即机会。只有抓住机会，我们解决问题的能力才会在痛苦的历练中像雪球一样越滚越大。

8.3.3 编程方法分享

编程是个实在活，千万不要以为买了一堆书，你就能自动成为大牛，你必须耐心地把这些书看完，不断练习、不断思考，这样你才有机会成为大牛。

对于初学者来说，我觉得基础知识非常重要，而这些基础知识不能光靠记忆，记忆是不可靠的，还是要学会自我思考，理解编程、设计的精髓。这就好像我们小时候背乘法口诀，老师天天追在屁股后面让我们背，光死背能背出来吗？我还记得读高中时候化学老师让我们把元素周期表全部背诵下来，差一点就磨灭了我的化学热情。

学习一门弱类型的编程语言，不要先学习那种具有强制类型的、面向对象的编程语言。严格而言，如果有人对你提到 `class`(类)或继承，那么你就应该去选择其他的途径了。虽然我认同类和继承相关技术是软件开发中必不可少的，但是我强烈认为它们不应该是初学者的选择。

考虑到这一点，这里有一些具体的建议给那些正在学习或准备学习 Web 应用开发的初学者。实际上，说得更远点更抽象点，这就是一个如何开始学习软件开发的一个好计划。很显然，这不是一个适合所有人的计划，但是我认为它一定适合大部分初学者。鉴于此，我认为 JavaScript 和 Java 是对于初学者而言最理想的编程语言，因为 JS 解释器在绝大部分浏览器上都可用，而 Java 只需要安装 JDK 和 Eclipse 这样的 IDE 工具就可以直接调试，再则 JavaScript 的面向对象特性并不是强制型的，并且无论 Java 或 JavaScript，它们都在工业界被广泛使用。

以 JavaScript 为例，说得更具体点，我建议你以这个顺序学习。

学习如何打印出一些东西，学习如何声明和定义变量，学习基本算术运算操作（包括余数操作），学习循环（特别是 for 循环），学习把抽象重复的代码写成函数，学习字符串和用循环操作字符串，学习数组和数组的循环方法（特别是 foreach 循环），学习创建和操作对象数据集。记住上面的这些并每天写一个程序来实践，直到这些都轻而易举地想起来。学习 Git 的基本操作，学习通过命令行使用 Git。这意味着要先学习四个 Unix/Linux 命令（`ls`,`pwd`,`mkdir`,`cd`）。当学习了这几个命令，也就学会了以“树型”或层次结构的呈现方式查询文件系统。

一旦你掌握了上面的几个 Unix/Linux 命令，并会从命令行进入文件系统，你就应该学几个基础的 Git 命令。主要是 `git init`, `git status`, `git add` and `git commit`。一旦你掌握了 Git 的基本操作，在

学习下面的技术时将其集成到你的工作流中。学习 HTML 基础，能够凭记忆创建简单的 HTML 页面。学习 DOM 和如何理解 HTML 作为指定的分层树结构。花点时间来思考它如何关系到你在前面步骤中学到的分层文件系统。学习 CSS 选择器，了解它如何让你选定 DOM 的某些部分。了解 DOM 元素之间的关系。了解一个 DOM 元素作为另一个 DOM 元素的父元素或子元素的含义。理解这与后代和祖先之间的关系有什么不同。记住选择器可以让你通过这些关系来选定某些元素。学习 jQuery，并主要专注于 DOM 的操作能力。学会用 jQuery 对 DOM 插入或删除元素，实践可视化如何影响用 DOM 定义的树型结构。实践 jQuery 中的事件处理和 DOM 操作（比如，实践操作 DOM 当用户点击某个东西，或在指定的时间间隔）。多练习 JavaScript 对象，并把它们当作可变的聚合器。学习如何用 JavaScript 来表示更复杂的数据而不是基本数据类型。学会应用并操作这些数据结构。理解并定义 JSON、理解它如何与 JavaScript 对象相关联。学会使用 jQuery 的 getJSON 函数从文件中获取数据到 JavaScript 对象中。使用类似的技术，用一个简单的 JSONP API 去练习用 AJAX 拉取数据。练习向 DOM 插入和删除这个数据。在这个阶段，做一个简单的幻灯片来循环播放 Flickr 图片，这将是一个令人难以置信的项目，将真正考验你的能力，使用之前学过的基础技术来实现它。如果你做了这一步，那么你已经掌握了大量必备的编程和计算机科学基本概念。具体来说，你掌握了计算机程序的最重要元素（如果 if-else 语句，循环，变量，对象，函数，数组等），你已经学会了链式或树型的数据结构。这时，无疑你已经准备好转移到更高级的主题。

8.4 本章小结

本章是针对前面各个章节没有提到的一些性能优化建议、全局性建议的补充。首先针对 Web 应用、Web 容器的性能优化提出自己的建议，然后讲解了一些数据库应用方面的优化建议，接下来对企业级应用和系统整体架构方面的优化提出自己的看法，最后是一些与个人品质、思维方式相关的建议。通过本章的分享，全书完成了所有与 Java 程序相关的知识分享，希望读者能够受益。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010)88254396；(010)88258888

传 真：(010)88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱 电子工业出版社总编办公室

十 十 邮 十 编：100036 十 十 十 十
十 十 十 十 十 十 十 十 十 十
十 十 十 十 十 十 十 十 十 十
十 十 十 十 十 十 十 十 十 十 十 十 十 十
十 十 十 十 十 十 十 十 十 十 十 十 十 十
十
十
十
十
十 十 十 十 十 十 十 十 十 十 十 十 十 十 十 十 十 十 十 十

专家热评

系统调优在软件的后续改进和重构中占有很重要的地位，能够弥补软件的不足，本书以通俗的语言和引人入胜的故事，重点讲述软件性能调优的方法论和具体实现路径，读者可以根据自己的实际情况进行参照比对，就像进了兵器库挑选适合自己的顺手武器。

程序凑合着上线是一回事，而在压力下能够优美地运行往往很不容易。本书对于所有有志于进行软件高级管理的人员而言，具有非常重要的意义。

海适云承CEO兼首席架构师 沈英桓 (Sam Shen)

当我翻开周明耀先生编写的《大话Java性能优化》这本书时，一下子被他生动朴实的语言所深深吸引，他将生硬、深奥的IT系统技术问题深入浅出地层层剥开，娓娓道来，并结合时下大家最为熟知的12306、电商等案例，系统地分析和介绍了系统调优的重要性、解决思路和技术实现。

作为金融IT的一名同行，我对系统性能对用户体验和业务处理的重要性深有体会，尤其是高频交易系统（HFT），对系统性能的要求近乎苛刻，对业务的处理和响应要求毫秒级。本书作者从系统架构、系统设计、开发、编码、算法等多层次、多角度提供思路和优化策略，是一本很务实的技术贴，值得大家学习、借鉴和探讨。

德意志银行（中国）有限公司环球科技运营经理 黄正兵

在我自己使用Java开发项目的过程中，经常会切实地感受到系统调优的重要性。然而Java性能调优并不是一项一蹴而就的简单任务，而是如同并发编程需要关注算法、内存、I/O等各种问题，以及丰富的经验积累。

本书中作者结合自己的实践经验总结了一些性能优化的方案。这些经验涉及Java基本语法、对象和引用、String类型和集合类的使用等各个方面且附有示例，使人受益匪浅，如果能够将其灵活运用到自己的系统中，相信能够对读者处理性能优化问题提供不小的帮助。此外，作者看待性能优化问题的视角相对开阔，系统且详尽地讨论了可能导致性能问题的各个环节和不同角度下性能优化的问题，读后令人豁然开朗。

西安工业大学2016应届硕士毕业生 Fenny



博文视点Broadview



@博文视点Broadview



责任编辑：董 英

封面设计：李 玲

上架建议：计算机>Java

ISBN 978-7-121-28481-6



定价：89.00元