



# 分布式 Java应用 基础与实践

林昊 著

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

PDG

# 分布式Java应用：基础与实践

## 专家推荐

林昊总结了在淘宝进行大规模Java应用的经验，呕心沥血写出的这本书既有理论深度，又具有极强的实战指导意义。升级指南，不得不读。

——人间网创始人，CTO 曹晓钢

在概念满天飞的浮躁时代，林昊以朴实的文字和扎实的基础，由浅入深介绍分布式应用设计开发的架构和技术细节。在豆瓣正向SOA转型的节骨眼儿得到这样一本书，真是恰逢甘霖。

——豆瓣架构师 洪强宁

本书不仅深入分析了大规模Java系统间通讯、SOA架构、集群、可伸缩和高可用性系统，还有难得一见的JVM内幕分析和对CPU、IO、内存的性能调优实践，对开发高性能系统相当有帮助。

——搜狐工程师 邓正平

本书详述了构建大型分布式Java应用的相关知识与应用场景，使用大量代码进行实例分析，对构建高可用系统帮助很大，遗憾的是对系统架构集成等方面的影响未做更深入的讨论。

——网易杭州研究院工程师 尧飘海

在互联网领域，性能调优、分布式、高可用、可伸缩等总是困扰着大家，少有高手分享实战经验，一书难求。这本书系统分析了上述典型问题，并给出了多种解决方案和扩展阅读，着实让我过了一把瘾。期待着林昊今后还能在系统层面和原理层面做更深入的分享。

——中国移动数据业务运营中心门户  
技术部经理 朱岩

国内大部分Java工程师了解的知识面都偏向于SSH、MVC框架等，精通高访问量大并发应用的Java技术人员奇缺。本书讲到的构建可伸缩系统章节非常实用，不少方法我们也正在使用。书中对性能调优的介绍也非常专业和深入，对于大型系统的调优具有极大的参考价值。

——新浪架构师 杨卫华

对于大型分布式应用和性能，很多书或陷入细节，或流于空谈，本书则把细节、架构、底层、应用平衡得很好，技术功底之外，更有写作的诚意。若语言能更生动一些，可读性会更好。

——瑞友科技（原用友软件工程公司）  
IT应用研究院副院长 池建强

Java类图书，汗牛充栋，有关分布式高可用系统架构的书却很稀少，本书填补了这方面的空白。即便是非JAVA类开发者，也能从中学到大型分布式高可用系统的设计方法和思路。

——上海盛大网络发展有限公司  
技术保障中心总监 资深研究员 陈桂新

对每一个新的知识点，作者都列出了很多链接，供读者进一步学习。期待这本书让更多人受益。

——上海赢思软件技术有限公司  
资深系统/网络安全架构师 陈成才



策划编辑：徐定翔  
项目编辑：白爱萍  
责任编辑：杨绣国  
责任美编：杨小勤

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。



图书分类 程序设计

ISBN 978-7-121-10941-6



9 787121 109416 >

定价：49.80元



# 分布式 Java应用

## 基础与实践

林昊 基

电子工业出版社  
Publishing House of Electronics Industry  
北京 · BEIJING



## 内容简介

本书介绍分布式 Java 应用涉及的知识点，分为基于 Java 实现网络通信、RPC；基于 SOA 实现大型分布式 Java 应用；编写高性能 Java 应用；构建高可用、可伸缩的系统 4 个部分，共 7 章内容。作者结合自己在淘宝网的实际工作经验展开论述，既可供初学者学习，也可供同行参考。

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

分布式 Java 应用：基础与实践 / 林昊著. —北京：电子工业出版社，2010.6

ISBN 978-7-121-10941-6

I. ①分… II. ①林… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2010）第 093814 号

策划编辑：徐定翔

责任编辑：杨绣国

印 刷：北京市天竺颖华印刷厂

装 订：三河市鑫金马印装有限公司

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：18.5 字数：350 千字

印 次：2010 年 6 月第 1 次印刷

印 数：5 000 册 定价：49.80 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 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

献给我想共度一生的爱人宗伟



# 分布 & 分享

分布式计算不是一门年轻的技术，早在上个世纪 70 年代末便已是计算机科学的一个独立分支了；它也不是一门冷僻的技术，从 C/S 模式到 P2P 模式，从集群计算到网格计算，乃至风靡当下的云计算，都是其表演的舞台。另一方面，Java 作为一门应网络而生的语言，对分布式计算有着天然的友好性，同时也是当今最流行的编程语言。然而令人稍感意外的是，以“分布式 Java 应用”为专题的书籍并不多见，佳作则更少。至于国人所著者，请恕在下孤陋，尚未得见。不过细想之下，其实不足为奇。要开发一个高质量的分布式 Java 应用，以达到高性能、可伸缩、高可用、低延迟的要求，同时保证一致性、容错性、可恢复性和安全性，是何等的不易啊。它对程序开发的挑战在于：不仅需要对 Java 语言、类库、各种框架及相关工具极为熟悉，还需要对底层的 JVM 机制、各种网络协议有足够的了解；它对分析设计的挑战在于：不仅需要熟悉传统的应用架构模式，还需要掌握适用于分布式应用的架构模式和算法设计。此外，分布式应用对数据库管理员、测试分析人员等也提出了更高的要求。一本书若想涵盖这么多的内容，其难度可想而知。作者不仅需要具备起码的理论素养，更需要有丰富的实践经验。如果仅仅是纸上谈兵，对读者是无甚裨益的。

正因如此，半年多前收到林昊先生的部分书稿时，便窃感欣喜，认定这是一个很好的选题。作为淘宝网的系统架构师，他有着令人艳羡的得天独厚的实践机会。尤其难得的是，林昊亲身经历了淘宝网的快速成长和转型期，饱尝其间的成败甘苦，从中也获得了许多宝贵的第一手经验。如今，他选择将这些经验以书籍的形式与众共享，对有志于分布式应用开发的读者而言无疑是一大福音。

本书的基础部分介绍了分布式 Java 应用的基本实现方式（重点是 SOA）、相关的 JDK 类库和第三方框架，并对 JVM 的基本机制进行了深入解析；实践部分则关注于高性能、高可用和可伸缩系统的构建等。全书文风朴实，并附有大量的代码、数据和图表，比较符合大多数程序员的口味，也非常具有实践指导意义。如果用挑剔的眼光看，本书在深度、广度和高度上还有继续改进的余地，比如：对关键性的并发设计和算法设计可以介绍得更深入些；尚未涉及分布式应用中的安全问题；在性能调优一章中对 Java 源码级别的优化介绍似嫌不足；未能高屋建瓴地总结和提炼出分布式应用独有的编程和设计原则、架构和思维模式，等等。当然，对于这样一本上乏经典可参、下需躬身实践的书籍来说，以上多属苛求。事实上，本人在审稿过程中也是获益良多。

另值一提的是，林昊先生利用业余时间写作，已是经年有余，其间数易其稿，且从善如流，充分体现了技术人员的求道精神。本书的主题在明是**分布**，在暗则是**分享**——分享一段成长经历、分享一份宝贵经验，无论对作者还是读者，都是善莫大焉。

郑晖

<http://blog.zhenghui.org/>

《冒号课堂——编程范式与 OOP 思想》作者

2010 年 5 月于广州



# Get Architecture Done

提起诸如“高性能”、“高可用性”、“大规模并发”、“可扩展性”这些词汇，我相信多数技术人的心情都是激动而稍有点复杂的，当然，也或许是不屑一顾。毕竟不是谁都有机会面对这些富有挑战的技术场景，也不是每个架构师在面对这些挑战之前都能做好技术上的准备。那些意外故障总是不期而至，疲于奔命地解决问题的场景回顾起来对架构师来说犹如一场噩梦。

本书阐述当一个面向数以亿计用户的网站经过几年高速发展，技术团队不得不面临大规模、高并发、高扩展性等挑战带来的技术困境的时候，一个出色的架构师经过多年一线实践后累积的经过时间考验的解决方案以及宝贵的实战经验。在这本书里，你会看到作者在解决一些关乎 Web 应用问题的指导原则、实践方法、多重工具的综合运用以及作者本人的感悟。要强调的是，本书讲述的内容是一个 Web 应用从小到大过程中遇到的棘手问题的解决之道，并非宏观解析，亦非屠龙之技。无论您面对的站点是大是小，皆会有参考作用，毕竟大站点会越来越复杂，而小站点总有一天也将变大。

如今到计算机书店里走一下，会发现 Java 架构相关的技术图书虽已不少，但仍有理由相信本书内容填补了在 Java 架构实战方面的空白。在互联网应用大行其道的今天，有些名义上主题为 Java 架构的图书，要么单从 Java 本身阐述，缺乏整体应用的大局观；要么是高屋建瓴，从编程思想的高度坐而论道，缺乏实践性；要么是闭门造车之作，缺乏验证性。本书作者林昊多年来致力于推动 OSGi 在国内的发展，不乏理论功底，而后加盟淘宝网(Taobao.com)的几年间奋战在架构一线，爬摸滚打积累了丰富的实践心得。所以，本书是一本不折不扣的“理论结合实践”之作。

考虑国内的技术图书出版环境以及必须尽力适应读者的预期，写书本身是一件十分耗费心力的事情，但能将知识传递给更多人无疑又是让人快乐的。现在，经过作者近两年的梳理与总结，这本书即将出版，相信读者在研读本书之后会有所收获并可运用到所面对的 Web 应用上，也期待将来有更多朋友能够分享架构实践经验，一展才华，不亦快哉！

冯大辉

<http://dbanotes.net>

2010 年 5 月于杭州贝塔咖啡

# 实践是最好的成长 发表是最好的记忆

——作者序

分布式 Java 应用需要开发人员掌握较多的知识点，通常分布式 Java 应用的场景还会对性能、可用性以及可伸缩有较高的要求，而这也意味着开发人员需要掌握更多的知识点。我刚进淘宝的时候，曾经一直苦恼对于一个这样的分布式 Java 应用，我到底需要学习些什么。

随着在淘宝工作的不断开展，我的眼前终于慢慢呈现了高性能、高可用以及可伸缩的 Java 应用所需知识点的全景，这张知识点的全景图现在已经演变成了本书的目录。当看到自己整理出的知识点的全景图时，很惊讶地发现其中有些知识点其实是我之前已经学习过的，但到了真正需要使用的时候有些是完全遗忘了，有些则是在使用时碰到了很多的问题，从这里我看到，当学习到的知识不去经过实践检验时，这些知识就不算真正属于自己。

幸运的是，在淘宝我得到了分布式 Java 应用的绝佳实践机会，于是所学习到的网络通信、高性能、高并发、高可用以及可伸缩的一些知识，有机会在实践中得到验证。正是这样的机会，让我对这些知识点有了更深刻本质的理解，并能将其中的一些知识真正吸收，变为自己的经验，所以我一个很真切的体会就是：**实践是最好的成长**。

经历了这段艰难的成长，自己也希望不要忘却在这个过程中的收获，胡适先生曾说：“发表是最好的记忆”（这句话也长期放在台湾技术作家侯捷老师的网站上），于是萌生了写这本书的想法，一方面想梳理自己通过实践所获得的成长，另一方面也希望与正在从事分布式 Java 应用的技术人员分享一些实践的心得，同时给将要或打算从事分布式 Java 应用的技术人员提供一些参考。

从 2008 年 11 月确认要写这本书，到 2010 年 5 月完成这本书，历时一年半，过程可谓波折不断，前 3 个月的写作一帆风顺，顺利完成了第 1 章和第 3 章的撰写。

到了 2009 年 3 月底后，由于投入到了《OSGi 原理与最佳实践》一书的编写中，停顿了将近 3 个

月的时间。

2009 年 8 月重新开始了这本书的撰写，在 2009 年 10 月下旬前按计划完成了第 2 章、第 4 章和第 5 章，但随后由于项目进入冲刺阶段、忙于校园招聘以及迁居等事情，再度停顿了本书的撰写。

记得在刚开始写这本书的时候，周筠老师就告诉我，要坚持写，就算每天只写一点也是好的，千万不要停顿！确实如此，在停顿了两次后，就很难再找到继续写这本书的动力和激情了，得感谢徐定翔编辑在之后给我的鼓励和督促，终于在 2010 年 3 月我又开始了写作。待完成了第 6 章的编写后，由于剩下的第 7 章中的部分知识点和自己的工作联系不是非常紧密，导致了不断的拖延，这时周筠老师和徐定翔编辑给我的一个建议起到了关键作用，就是先放下第 7 章，先做之前完成的六章的定稿。

回到自己熟悉的前 6 章，终于再次有了写作的动力，随着工作中对自己书中所涉及的知识点的不断实践，此时再回头看自己半年前甚至一年前写的初稿时，发现其中有不少的错误以及条理不够清晰的地方，于是进行了大刀阔斧的改动，实践所获得的积累此时起了巨大的推动作用，这 6 章定稿除了第 3 章以外的章节，完成得较为顺利。

6 章定稿提交后，由白爱萍编辑带领的编辑小组再次对书稿进行细致的“田间管理”，给出了非常多的修改建议，印象中几乎平均每页都至少有两到三处需要修改的地方，正是他们的认真和专业，使得定稿中很多语言上以及技术上的错误得以纠正，感谢武汉博文的编辑们。

最后，在第 3 章定稿的修改过程中，得到了同事莫枢 (<http://rednaxelafx.javaeye.com>) 非常多的建议和帮助，在此非常感谢他的支持。

全书在编写的过程中，初稿、定稿也提交给了一些业内专家帮忙评审，主要有：郑晖、霍炬、曹祺、刘力祥，他们给出的很多意见一方面纠正了书中一些技术上的错误，另一方面也让书的条理性更加顺畅，衷心感谢他们的辛勤付出。尤其郑晖老师，在承诺为本书写推荐序时，又花时间把全稿通读一遍，他的耐心和专业精神，让我感佩不已。

书的撰写过程是如此漫长，每天晚上下班后、周末、假期，甚至过年期间，都成了写书的时间，感谢家人给我的包容、支持和理解，最要感谢的是我明年春节就将迎娶的准老婆：宗伟，感谢你忍受了我不断忘记买家里需要的各种东西，感谢你独立完成了新家的装修，更要感谢你允许我这么多的周末、假日都无法陪伴你，没有你的支持和鼓励，这本书是无法完成的。

回顾整个编写过程，从开始编写，到提交完全部终稿，经历了 15 个月的时间，写作时间大概为 11 个月，经过这 11 个月对这些知识点的不断实践、回顾和总结，它们在我的脑海中刻下了深深的烙印，的确，发表是最好的记忆。

林昊

2010 年 5 月 20 日晚于杭州家中

# 高效写作，敏捷出版

—— 出版人感言

和林昊认识了五六年了。他还在深圳的时候，来武汉出差，我们俩一起吃一大盘船帮鱼头，憨憨的他，话不多，就知道笑。

说话间，他的第二本书就要上市了。这本书，我更多地放手让编辑去做，当然，我扮演的还是黑头黑面的教练。编辑们在手记里都提到了被我“强力推动”做某事。

天下无难事，天下无易事。难，是因为问题没有被细分到可以去解决的程度；易，是因为找到了解决问题的办法。

很多初涉写作的朋友，因为读书多，读一流书还不少，眼界一高，就看不上自己初出茅庐的文笔。这种追求完美的心态，导致很多人的写作没效率。其实，谁刚开始学走路就姿势优美呢？当编辑这么些年，不知见过多少作者毁在这追求一步走到金字塔塔尖的误区中了。可惜当时的我不明就里，没有带给他们切实的帮助。

写作，其实可以是高效的，前提是：学会接受自己刚试手时的不完美，每天都不能停止写。

高效的写作取决于高效的时间管理，高效的时间管理离不开专注与灵活这两种能力。所有与我们合作成功的作者，都是很忙的人，但他们忙得有效率，既专注又灵活。专注，让他们懂得抓主要矛盾；灵活，让他们知道借力，能随时理解并配合出版社的各项要求。

写作是件很辛苦的事情，而且看起来似乎金钱上的回报未必高（除了某些畅销书）。但是，所有在博文出书的作者，他们都深谙写作的好处。有哪些好处呢？

1. 写作犹如教书，作者与读者教学相长。因为想把道理写给人看，要写明白，就会发现其实自己脑子里还有好多模糊点没搞清楚。因为要教读者，所以只能自己搞得更清楚。为了给读者一碗水，本来只有半桶水的作者，不得不让自己变得拥有一桶水。您瞧，这是谁赚了？

2. 写作，光教人怎么干不算高明，那是授人以鱼；还要告诉人为什么要这么干，这是授人以渔。上升到规律层面，就得要求作者有较扎实的理论功底，能把一件事说圆了。而在这个过程中，不少作者会痛苦地发现自己的系统思维不足、理论功底不扎实。这就迫使他们不断思考，直到能把事情说得让

自己和读者都信服。这就很好地训练了作者的系统思维能力。系统思维能力的提升，让一个人的视野更开阔，站得更高，看得更远。试想，又是谁赚了？

3. 现在，已经进入了个人品牌时代。想拿张名片就让人对你刮目相看，土！至少得有个博客吧。看看现在，IT 圈的朋友们，有几个没博客的？但，有几个人能写书咧？物以稀为贵。写了一本好书的人，声名远播，个人价值会放大。所以，现在愿意写书的人越来越多。

好处明白了，怎样才能做到高效写作？怎样才能做到敏捷出版？我们和作者们进行了一些探讨，虽然这方面的认识积累还不够，但，有一说一。比如，在和出版社共同确定了图书产品的市场定位后（需求分析的过程也颇熬人，编辑手记里有细述），作者可以尝试：

1. 对自己的写作进行难度分级管理。忙的时候写难度低的，闲下来就写难度高的。这样就能保持写作的平稳状态，不至于忙的时候完全不写，闲的时候写一大堆，起伏太大就不易坚持。难度分级越细，越容易对写作进行模块化管理。写好一个模块就丢在那里，等着日后拼装。书也是个产品，把各个模块拼装成一本书的过程，和拼装宜家的家具没啥两样。

2. 每天得坚持写，给自己规定每日最低写作量。我们通常希望作者的每日写作量不要低于 500 字。区区 500 字，也就是三篇微博的字数之和。能坚持每日为自己的主题写 500 字的，就一定能把这个最低量渐渐提高到 1000 字。

3. 每个模块需要有写作的时间量限制，比如每 5 天必须完成一个小节，每 15 天必须完成一章中的一到两个核心小节，等等。

上述讲的是原创书的写作，其实在翻译和制作外版书时，这种模块化、敏捷运作的思路同样可以借鉴。我们用老办法做出来的产品，以往常常存在着周期和质量问题。最近我们出版的《编程之魂》就因为翻译质量挨骂了，这是因为这本书的制作还是采用的老模式，编辑和译者都单枪匹马，得不到及时的交流和帮助。挨骂不可怕，谁做产品会一帆风顺？重要的是通过思考，找到正确的方法。挨骂还是因为方法不对，方法不对，还是因为思考没到位。

林昊的这本书，我们还有不少工作没做到位，只是尽可能把团队现有的能力都贡献出来了，没有偷懒。这是进步。以往尽管也有能力不足之处，但偷懒也处处可见。

做林昊的书，再次体会到淘宝文化的魅力。开放的淘宝、重视知识积累和传承的淘宝，让我们淘到了不少林昊这样的“宝贝”作者，他们心态开放，接纳批评、消化批评的能力很强。我多次和林昊说，他的谦虚，他的开朗，他的认真，让我和同事们受益匪浅。

闻道有先后，写作有专攻。期待着更多的朋友和我们一起分享“高效写作，敏捷出版”带来的愉悦——快乐因创造价值而生。

有创造，就会有真正的成长。

周筠

2010 年 5 月 21 日于武汉

# 前言

软件系统得到用户认可后，访问量通常会产生爆发性的增长，互联网网站，例如淘宝、豆瓣等更是如此。在不断完善功能和多元化发展的同时，如何应对不断上涨的访问量、数据量是互联网应用面临的最大挑战。

对于用户而言，除了功能以外，网站访问够不够快，网站能否持续提供服务，也是影响用户访问量的重要因素，因此如何保证网站的高性能，及高可用也是我们要关注的重点。

为了支撑巨大的访问量和数据量，通常需要海量的机器，例如现在的 google 已经拥有了百万台以上的机器，这些机器耗费了巨大的成本（硬件采购成本、机器的电力成本、网络带宽成本等）。网站规模越大，运维成本就越高，这意味着商业公司所能获得的利润越低，而通常这会导致商业公司必须从多种角度关注如何降低网站运维成本。

本书的目的是从以上几个问题出发，介绍搭建高性能、高可用以及可伸缩的分布式 Java 应用所需的关键技术。

## 目标读者

本书涵盖了编写高性能、高可用以及可伸缩的分布式 Java 应用所需的知识点，适合希望掌握这些知识点的读者。

在介绍各个知识点时，作者尽量结合自己的工作，分享经验与心得，希望能够对那些有相关工作经验的读者有所帮助。

## 内容导读

本书按照介绍的知识点分为五个部分：第一部分介绍基于 Java 实现系统间交互的相关知识，这些知识在第 1 章中进行介绍；第二部分为基于 SOA 构建大型分布式 Java 应用的知识点，这些在第 2 章中介绍；第三部分为高性能 Java 应用的相关知识，这些在第 3、4、5 章中介绍；第四部分介绍高可用 Java

## II 前 言

应用的相关知识，这些在第 6 章中介绍；第五部分介绍可伸缩 Java 应用，这些在第 7 章介绍，读者也可根据自己的兴趣选择相应的章节进行阅读。

网站业务多元化的发展会带来多个系统之间的通信问题，因此如何基于 Java 实现系统间的通信是首先要掌握的知识点，在本书的第 1 章中介绍了如何基于 Java 实现 TCP/IP（BIO、NIO）、UDP/IP（BIO、NIO）的系统间通信、以及如何基于 RMI、Webservice 实现系统间的调用，在基于这些技术实现系统间通信和系统间调用时，性能也是要考虑的重点因素，本章也讲解了如何实现高性能的网络通信。

在解决了系统之间的通信问题后，多元化的发展带来的另外一个问题是多个系统间出现了一些重复的业务逻辑，这就要将重复的业务逻辑抽象为一个系统。这样演变后，会出现多个系统，这些系统如何保持统一、标准的交互方式是要解决的问题，SOA 无疑是首选的方式，在本书的第 2 章中介绍了如何基于 SOA 来实现统一、标准的交互方式。

高性能是本书关注的重点，Java 程序均运行在 JVM 之上，因此理解 JVM 是理解高性能 Java 程序所必须的。本书的第 3 章以 Sun Hotspot JVM 为例讲述了 JVM 执行 Java 代码的机制、内存管理的机制，以及多线程支持的机制。执行代码的机制包含了 Sun hotspot 将 Java 代码编译为 class 文件，加载 class 文件到 JVM 中，以解释方式执行 class，以及 client 模式和 server 模式编译为机器码方式执行 class 的实现方式；内存管理的机制包含了 Sun hotspot 内存分配以及回收的机制，内存分配涉及的主要为堆上分配、TLAB 分配以及栈上分配。内存回收涉及的有常见的垃圾回收算法、Sun Hotspot JVM 中新生代可用的 GC、旧生代可用的 GC 及 G1；多线程支持的机制包含了多线程时资源的同步机制，以及线程之间交互的机制。

除 JVM 外，在编写分布式 Java 应用时，通常要使用到一些 Sun JDK 的类库。如何合理地根据需求来选择使用哪些类，以及这些类是如何实现的，是编写高性能、高可用的 Java 应用必须掌握的。在本书的第 4 章中介绍了 Sun JDK 中常用的集合包的集合类、并发包中的常用类（例如 ConcurrentHashMap、ThreadPoolExecutor）、序列化/反序列化的实现方式，同时对这些类进行了基准的性能测试和对比。

掌握 JVM 和使用到的类库是编写高性能 Java 应用的必备知识，但除了编写之外，通常还会面临对已有系统进行调优。调优是个非常复杂的过程，在本书的第 5 章中介绍了寻找系统性能瓶颈的一些方法以及针对这些瓶颈常用的一些调优方法。寻找性能瓶颈的方法主要是根据系统资源的消耗寻找对应问题代码的方法，常用的一些调优方法主要是降低锁竞争、降低内存消耗等。

除了高性能外，高可用也是大型 Java 应用要关注的重点，在本书的第 6 章中介绍了一些构建高可用系统常用的方法，例如负载均衡技术、构建可容错的系统、对资源使用有限制的系统等。

在面对不断访问的访问量和数据量时，最希望能做到的是仅升级硬件或增加机器就可支撑，但要达到这个效果，在软件上必须付出巨大的努力。本书的第 7 章介绍了构建可伸缩系统的一些常用方法，主要包括支持垂直伸缩时常用的降低锁竞争等方法；以及支持水平伸缩时常用的分布式缓存、分布式文件系统等方法。

## 关于本书的代码

书中大部分代码只列出了关键部分或伪代码，完整的代码请从 <http://bluedavy.com> 下载。

## 关于本书的反馈

每次重看书稿，都会觉得这个知识点尚须补充，那个知识点尚须完善，但书不可能一直写下去。在提交终稿的时候，我不特别兴奋，倒是有一点遗憾和担心，遗憾书里的知识点还没有足够讲开，担心自己对某些知识点理解不够，误导了大家。为此，我在自己的网站（<http://bluedavy.com>）上开辟了一个专区，用于维护勘误，并将不断地完善书中涉及的知识点，欢迎读者反馈和交流。



# 联系博文

您可以通过如下方式与本书的出版方取得联系。

读者信箱：*reader@broadview.com.cn*

投稿信箱：*bvtougao@gmail.com*

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室

邮政编码：430074

电 话：027-87690813

传 真：027-87690595

欢迎您访问博文视点官方博客：<http://blog.csdn.net/bvbook>



# 目 录

|  |           |
|--|-----------|
| 前言 .....                               | 1         |
| <b>第 1 章 分布式 Java 应用 .....</b>         | <b>1</b>  |
| 1.1 基于消息方式实现系统间的通信 .....               | 3         |
| 1.1.1 基于 Java 自身技术实现消息方式的系统间通信 .....   | 3         |
| 1.1.2 基于开源框架实现消息方式的系统间通信 .....         | 10        |
| 1.2 基于远程调用方式实现系统间的通信 .....             | 14        |
| 1.2.1 基于 Java 自身技术实现远程调用方式的系统间通信 ..... | 14        |
| 1.2.2 基于开源框架实现远程调用方式的系统间通信 .....       | 17        |
| <b>第 2 章 大型分布式 Java 应用与 SOA .....</b>  | <b>23</b> |
| 2.1 基于 SCA 实现 SOA 平台 .....             | 26        |
| 2.2 基于 ESB 实现 SOA 平台 .....             | 29        |
| 2.3 基于 Tuscany 实现 SOA 平台 .....         | 30        |
| 2.4 基于 Mule 实现 SOA 平台 .....            | 34        |
| <b>第 3 章 深入理解 JVM .....</b>            | <b>39</b> |
| 3.1 Java 代码的执行机制 .....                 | 40        |
| 3.1.1 Java 源码编译机制 .....                | 41        |
| 3.1.2 类加载机制 .....                      | 44        |
| 3.1.3 类执行机制 .....                      | 49        |
| 3.2 JVM 内存管理 .....                     | 63        |

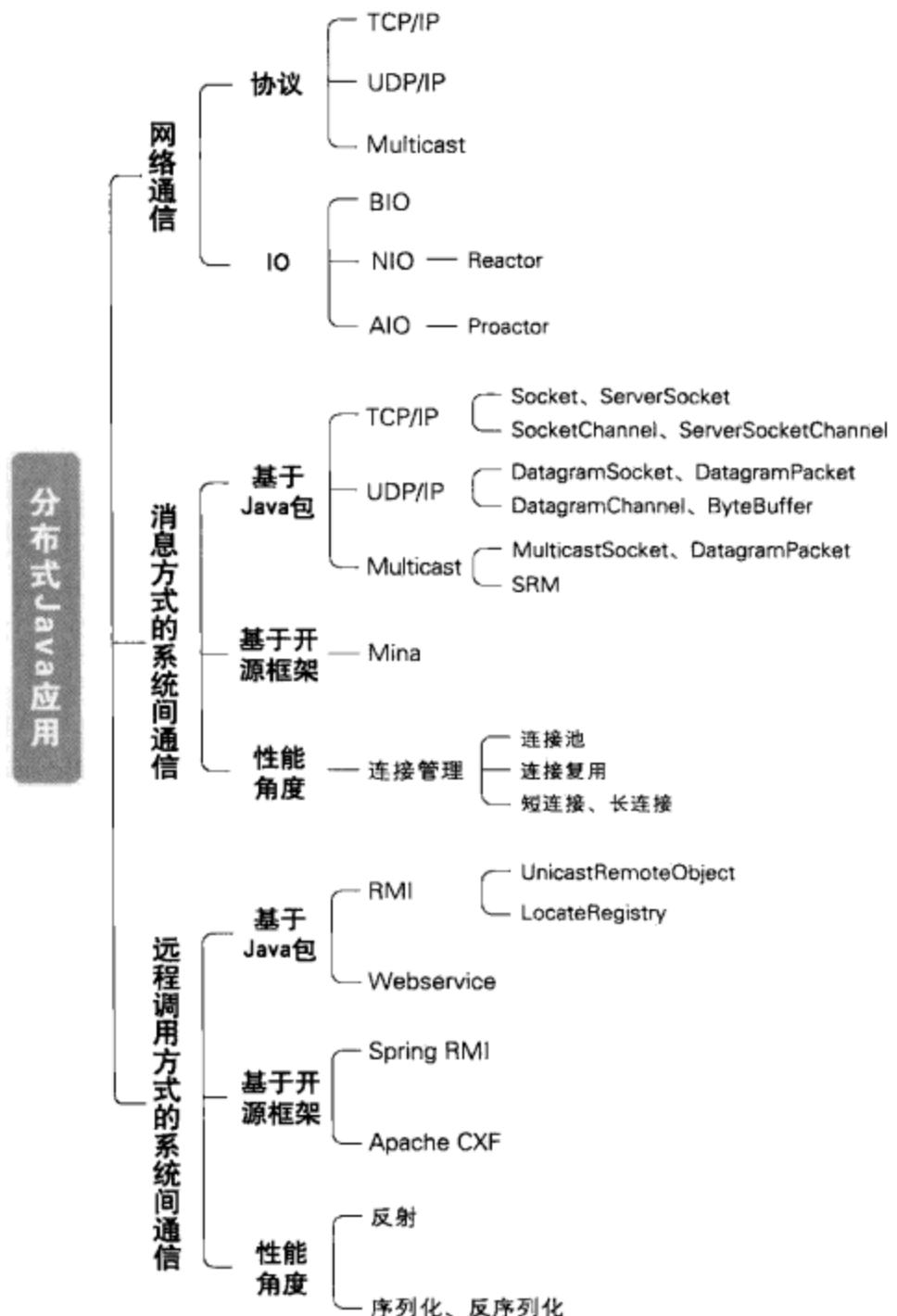
|  |            |
|--|------------|
| 3.2.1 内存空间 .....                           | 63         |
| 3.2.2 内存分配 .....                           | 65         |
| 3.2.3 内存回收 .....                           | 66         |
| 3.2.4 JVM 内存状况查看方法和分析工具 .....              | 92         |
| 3.3 JVM 线程资源同步及交互机制 .....                  | 100        |
| 3.3.1 线程资源同步机制 .....                       | 100        |
| 3.3.2 线程交互机制 .....                         | 104        |
| 3.3.3 线程状态及分析 .....                        | 105        |
| <b>第 4 章 分布式 Java 应用与 Sun JDK 类库 .....</b> | <b>111</b> |
| 4.1 集合包 .....                              | 112        |
| 4.1.1 ArrayList .....                      | 113        |
| 4.1.2 LinkedList .....                     | 116        |
| 4.1.3 Vector .....                         | 117        |
| 4.1.4 Stack .....                          | 118        |
| 4.1.5 HashSet .....                        | 119        |
| 4.1.6 TreeSet .....                        | 120        |
| 4.1.7 HashMap .....                        | 120        |
| 4.1.8 TreeMap .....                        | 123        |
| 4.1.9 性能测试 .....                           | 124        |
| 4.1.10 小结 .....                            | 138        |
| 4.2 并发包 (java.util.concurrent) .....       | 138        |
| 4.2.1 ConcurrentHashMap .....              | 139        |
| 4.2.2 CopyOnWriteArrayList .....           | 145        |
| 4.2.3 CopyOnWriteArraySet .....            | 149        |
| 4.2.4 ArrayBlockingQueue .....             | 149        |
| 4.2.5 AtomicInteger .....                  | 151        |
| 4.2.6 ThreadPoolExecutor .....             | 153        |
| 4.2.7 Executors .....                      | 157        |

|                                    |            |
|------------------------------------|------------|
| 4.2.8 FutureTask .....             | 158        |
| 4.2.9 Semaphore.....               | 161        |
| 4.2.10 CountDownLatch .....        | 162        |
| 4.2.11 CyclicBarrier.....          | 163        |
| 4.2.12 ReentrantLock.....          | 163        |
| 4.2.13 Condition.....              | 164        |
| 4.2.14 ReentrantReadWriteLock..... | 165        |
| 4.3 序列化/反序列化 .....                 | 167        |
| 4.3.1 序列化.....                     | 167        |
| 4.3.2 反序列化.....                    | 169        |
| <b>第 5 章 性能调优 .....</b>            | <b>173</b> |
| 5.1 寻找性能瓶颈 .....                   | 175        |
| 5.1.1 CPU 消耗分析.....                | 175        |
| 5.1.2 文件 IO 消耗分析.....              | 182        |
| 5.1.3 网络 IO 消耗分析.....              | 186        |
| 5.1.4 内存消耗分析.....                  | 187        |
| 5.1.5 程序执行慢原因分析.....               | 191        |
| 5.2 调优 .....                       | 192        |
| 5.2.1 JVM 调优.....                  | 192        |
| 5.2.2 程序调优.....                    | 202        |
| 5.2.3 对于资源消耗不多，但程序执行慢的情况 .....     | 214        |
| <b>第 6 章 构建高可用的系统 .....</b>        | <b>227</b> |
| 6.1 避免系统中出现单点.....                 | 228        |
| 6.1.1 负载均衡技术.....                  | 228        |
| 6.1.2 热备 .....                     | 236        |
| 6.2 提高应用自身的可用性.....                | 238        |
| 6.2.1 尽可能地避免故障.....                | 239        |
| 6.2.2 及时发现故障.....                  | 246        |

|                              |            |
|------------------------------|------------|
| 6.2.3 及时处理故障 .....           | 248        |
| 6.2.4 访问量及数据量不断上涨的应对策略 ..... | 249        |
| <b>第 7 章 构建可伸缩的系统.....</b>   | <b>251</b> |
| 7.1 垂直伸缩 .....               | 252        |
| 7.1.1 支撑高访问量 .....           | 252        |
| 7.1.2 支撑大数据量 .....           | 254        |
| 7.1.3 提升计算能力 .....           | 254        |
| 7.2 水平伸缩 .....               | 254        |
| 7.2.1 支撑高访问量 .....           | 254        |
| 7.2.2 支撑大数据量 .....           | 264        |
| 7.2.3 提升计算能力 .....           | 266        |



# 第1章 分布式Java应用



大型应用通常会拆分为多个子系统来实现，对于 Java 来说，这些子系统可能部署在同一台机器的多个不同的 JVM 中，也可能部署在不同的机器上，但这些子系统又不是完全独立的，要相互通信来共同实现业务功能，对于此类 Java 应用，我们称之为分布式 Java 应用。

Martin Fowler 在《企业应用架构模式》一书中曾经说过：“能不用分布式的情况下就不要用分布式”，当应用变为分布式 Java 应用时，会很大程度地增加应用实现的技术复杂度，对于分布式 Java 应用，通常有两种典型的方法来实现。

### 1. 基于消息方式实现系统间的通信

当系统之间要通信时，就向外发送消息，消息可以是字节流、字节数组，甚至是 Java 对象，其他系统接收到消息后则进行相应的业务处理。

消息方式的系统间通信，通常基于网络协议来实现，常用的实现系统间通信的协议有：TCP/IP 和 UDP/IP。

TCP/IP 是一种可靠的网络数据传输的协议。TCP/IP 要求通信双方首先建立连接，之后再进行数据的传输。TCP/IP 负责保证数据传输的可靠性，包括数据的可到达、数据到达的顺序等，但由于 TCP/IP 需要保证连接及数据传输的可靠，因此可能会牺牲一些性能。

UDP/IP 是一种不保证数据一定到达的网络数据传输协议。UDP/IP 并不直接给通信的双方建立连接，而是发送到网络上进行传递。由于 UDP/IP 不建立连接，并且不能保证数据传输的可靠，因此性能上表现相对较好，但可能会出现数据丢失以及数据乱序的现象。

TCP/IP 和 UDP/IP 可用于完成数据的传输，但要完成系统间通信，还需要对数据进行处理。例如读取和写入数据，按照 POSIX 标准分为同步 IO 和异步 IO 两种，其中同步 IO 中最常用的是 BIO（Blocking IO）和 NIO（Non-Blocking IO）。

从程序角度而言，BIO 就是当发起 IO 的读或写操作时，均为阻塞方式，只有当程序读到了流或将流写入操作系统后，才会释放资源。

NIO 是基于事件驱动思想<sup>1</sup>的，实现上通常采用 Reactor 模式<sup>2</sup>，从程序角度而言，当发起 IO 的读或写操作时，是非阻塞的；当 Socket 有流可读或可写入 Socket 时，操作系统会相应地通知应用程序进行处理，应用再将流读取到缓冲区或写入操作系统。对于网络 IO 而言，主要有连接建立、流读取及流写入三种事件，Linux 2.6 以后的版本采用 epoll<sup>3</sup>方式来实现 NIO。

下面再来看看另一种方式——AIO。AIO 为异步 IO 方式，同样基于事件驱动思想，实现上通常采用 Proactor 模式<sup>4</sup>。从程序角度而言，和 NIO 不同，当进行读写操作时，只须直接调用 API 的 read 或 write 方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流

1 [http://en.wikipedia.org/wiki/Event-driven\\_programming](http://en.wikipedia.org/wiki/Event-driven_programming)

2 [http://en.wikipedia.org/wiki/Reactor\\_pattern](http://en.wikipedia.org/wiki/Reactor_pattern)

3 <http://lse.sourceforge.net/epoll/index.html>

4 [http://en.wikipedia.org/wiki/Proactor\\_pattern](http://en.wikipedia.org/wiki/Proactor_pattern)

传入 `read` 方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将 `write` 方法传递的流写入完毕时，操作系统主动通知应用程序。较之 NIO 而言，AIO 一方面简化了程序的编写，流的读取和写入都由操作系统来代替完成；另一方面省去了 NIO 中程序要遍历事件通知队列(Selector)的代价。Windows 基于 IOCP<sup>5</sup>实现了 AIO，Linux 目前只有基于 epoll 模拟实现的 AIO。

Java 对 TCP/IP 和 UDP/IP 均支持，在网络 IO 的操作上，Java 7 以前的版本仅支持 BIO 和 NIO 两种方式，对 AIO 方式感兴趣的读者可自行下载 Sun JDK 7 进行尝试。

## 2. 基于远程调用方式实现系统间的通信

当系统之间要通信时，可通过调用本地的一个 Java 接口的方法，透明地调用远程的 Java 实现。具体的细节则由 Java 或框架来完成，这种方式在 Java 中主要用来实现基于 RMI 和 WebService 的应用。

本章通过举例来介绍如何基于 Java 的包及开源的产品来实现以上两种方式的系统间通信，这些是实现分布式 Java 应用的基础和必备知识，采用的例子如下。

示例程序由一个服务器端程序和一个客户端程序构成，是典型的请求-响应机制，即客户端发送请求，服务端响应。客户端读取用户的输入，并将输入的字符串信息发送给服务器端，服务器端接收到信息后响应，当客户端输入的是 `quit` 字符串时，则停止客户端和服务器端的程序。

# 1.1 基于消息方式实现系统间的通信

## 1.1.1 基于 Java 自身技术实现消息方式的系统间通信

基于 Java 自身包实现消息方式的系统间通信的方式有：TCP/IP+BIO、TCP/IP+NIO、UDP/IP+BIO 以及 UDP/IP+NIO 4 种，下面分别介绍如何实现这 4 种方式的系统间通信。

### TCP/IP+BIO

在 Java 中可基于 `Socket`、`ServerSocket` 来实现 TCP/IP+BIO 的系统间通信。`Socket` 主要用于实现建立连接及网络 IO 的操作，`ServerSocket` 主要用于实现服务器端端口的监听及 `Socket` 对象的获取。基于 `Socket` 实现客户端的关键代码如下：

```
// 创建连接，如果域名解析不了会抛出 UnknownHostException，当连接不上时会抛出 IOException，  
// 如果希望控制建立连接的超时，可先调用 new Socket()，然后调用 socket.connect(SocketAddress  
// 类型的目标地址，以毫秒为单位的超时时间)  
Socket socket=new Socket(目标 IP 或域名，目标端口)；  
// 创建读取服务器端返回流的 BufferedReader  
BufferedReader in=new BufferedReader(new  
InputStreamReader(socket.getInputStream()));
```

<sup>5</sup> [http://en.wikipedia.org/wiki/Input/output\\_completion\\_port](http://en.wikipedia.org/wiki/Input/output_completion_port)

```
// 创建向服务器写入流的 PrintWriter
PrintWriter out=new PrintWriter(socket.getOutputStream(),true);
// 向服务器发送字符串信息，要注意的是，此处即使写失败也不会抛出异常信息，并且一直会阻塞到写入
操作系统或网络 IO 出现异常为止
out.println("hello");
// 阻塞读取服务端的返回信息，以下代码会阻塞到服务端返回信息或网络 IO 出现异常为止，如果希望在超
过一段时间后就不阻塞了，那么要在创建 Socket 对象后调用 socket.setSoTimeout(以毫秒为单位的超
时时间)
in.readLine();
```

服务器端关键代码如下：

```
// 创建对本地指定端口的监听，如端口冲突则抛出 SocketException，其他网络 IO 方面的异常则抛出
IOException ServerSocket ss=new ServerSocket(监听的端口)
// 接受客户端建立连接的请求，并返回 Socket 对象，以便和客户端进行交互，交互的方式和客户端相同，
也是通过 Socket.getInputStream 和 Socket.getOutputStream 来进行读写操作，此方法会一直阻
塞到有客户端发送建立连接的请求，如果希望此方法最多阻塞一定的时间，则要在创建 ServerSocket 后
调用其 setSoTimeout(以毫秒为单位的超时时间)
Socket socket=ss.accept();
```

上面是基于 `Socket`、`ServerSocket` 实现的一个简单的系统间通信的例子。而在实际的系统中，通常要面对的是客户端同时要发送多个请求到服务器端，服务器端则同时要接受多个连接发送的请求，上面的代码显然是无法满足的。

为了满足客户端能同时发送多个请求到服务器端，最简单的方法就是生成多个 `Socket`。但这里会产生两个问题：一是生成太多的 `Socket` 会消耗过多的本地资源，在客户端机器多，服务器端机器少的情况下，客户端生成太多 `Socket` 会导致服务器端须要支撑非常高的连接数；二是生成 `Socket`（建立连接）通常是比较慢的，因此频繁地创建会导致系统性能不足。鉴于这两个问题，通常采用连接池的方式来维护 `Socket` 是比较好的，一方面限制了能创建的 `Socket` 的个数；另一方面由于将 `Socket` 放入了池中，避免了重复创建 `Socket` 带来的性能下降问题。数据库连接池就是这种方式的典型代表，但连接池的方式会带来另一个问题，连接池中的 `Socket` 个数是有限的，但同时要用 `Socket` 的请求可能会很多，在这种情况下就会造成激烈的竞争和等待；还有一个需要注意的问题是合理控制等待响应的超时时间，如不设定超时会导致当服务器端处理变慢时，客户端相关的请求都在做无限的等待，而客户端的资源必然是有限的。因此这种情况下很容易造成当服务器端出现问题时，客户端挂掉的现象。超时时间具体设置为多少取决于客户端能承受的请求量及服务器端的处理时间。既要保证性能，又要保证出错率不会过高，对于直接基于 TCP/IP+BIO 的方式，可采用 `Socket.setSoTimeout` 来设置等待响应的超时时间。

为了满足服务器端能同时接受多个连接发送的请求，通常采用的方法是在 `accept` 获取 `Socket` 后，将此 `Socket` 放入一个线程中处理，通常将此方式称为一连接一线程。这样服务器端就可接受多个连接发送请求了，这种方式的缺点是无论连接上是否有真实的请求，都要耗费一个线程。为避免创建过多

的线程导致服务器端资源耗尽，须限制创建的线程数量，这就造成了在采用 BIO 的情况下服务器端所能支撑的连接数是有限的。

## TCP/IP+NIO

在 Java 中可基于 `java.nio.channels` 中的 `Channel` 和 `Selector` 的相关类来实现 TCP/IP+NIO 方式的系统间通信。`Channel` 有 `SocketChannel` 和 `ServerSocketChannel` 两种，`SocketChannel` 用于建立连接、监听事件及操作读写，`ServerSocketChannel` 用于监听端口及监听连接事件；程序通过 `Selector` 来获取是否有要处理的事件。基于这两个类实现客户端的关键代码如下：

```

SocketChannel channel=SocketChannel.open();
// 设置为非阻塞模式
channel.configureBlocking(false);
// 对于非阻塞模式，立刻返回 false，表示连接正在建立中
channel.connect(SocketAddress);
Selector selector=Selector.open();
// 向 channel 注册 selector 以及感兴趣的连接事件
channel.register(selector,SelectionKey.OP_CONNECT);
// 阻塞至有感兴趣的 IO 事件发生，或到达超时时间，如果希望一直等至有感兴趣的 IO 事件发生，可调用无参数的 select 方法，如果希望不阻塞直接返回目前是否有感兴趣的事件发生，可调用 selectNow 方法
int nKeys=selector.select(以毫秒为单位的超时时间)
// 如 nKeys 大于零，说明有兴趣的 IO 事件发生
SelectionKey sKey=null;
if(nKeys>0){
    Set<SelectionKey> keys=selector.selectedKeys();
    for(SelectionKey key:keys){
        // 对于发生连接的事件
        if(key.isConnectable()){
            SocketChannel sc=(SocketChannel) key.channel();
            sc.configureBlocking(false);
            // 注册感兴趣的 IO 读事件，通常不直接注册写事件，在发送缓冲区未满的情况下，一直是可写的，因此如注册了写事件，而又不用写数据，很容易造成 CPU 消耗 100% 的现象
            sKey = sc.register(selector, SelectionKey.OP_READ);
        }
        // 完成连接的建立
        sc.finishConnect();
    }
    // 有流可读取
    else if(key.isReadable()){
        ByteBuffer buffer=ByteBuffer.allocate(1024);
        SocketChannel sc=(SocketChannel) key.channel();
        int readBytes=0;
        try{
            int ret=0;
            try{

```

```

        // 读取目前可读的流, sc.read 返回的为成功复制到 bytebuffer 中的字
节数, 此步为阻塞操作, 值可能为 0; 当已经是流的结尾时, 返回 -1
while((ret=sc.read(buffer))>0){
    readBytes+=ret;
}
}
finally{
    buffer.flip();
}
}
finally{
    if(buffer!=null){
        buffer.clear();
    }
}
}

// 可写入流
else if(key.isWritable()){
    // 取消对 OP_WRITE 事件的注册
    key.interestOps(key.interestOps() & (~SelectionKey.OP_WRITE));
    SocketChannel sc=(SocketChannel) key.channel();
    // 此步为阻塞操作, 直到写入操作系统发送缓冲区或网络 IO 出现异常, 返回的为成功
写入的字节数, 当操作系统的发送缓冲区已满, 此处会返回 0
int writtenSize=sc.write(ByteBuffer);
// 如未写入, 则继续注册感兴趣的 OP_WRITE 事件
if(writtenSize==0){
    key.interestOps(key.interestOps() | SelectionKey.OP_WRITE);
}
}
}

selector.selectedKeys().clear();
}

// 对于要写入的流, 可直接调用 channel.write 来完成, 只有在写入未成功时才要注册 OP_WRITE 事件
int wSize=channel.write(ByteBuffer);
if(wSize==0){
    key.interestOps(key.interestOps() | SelectionKey.OP_WRITE);
}
}

```

从上可见, NIO 是典型的 Reactor 模式的实现, 通过注册感兴趣的事件及扫描是否有感兴趣的事件发生, 从而做相应的动作。

服务器端关键代码如下:

```

ServerSocketChannel ssc=ServerSocketChannel.open();
ServerSocket serverSocket=ssc.socket();
// 绑定要监听的端口

```

```

serverSocket.bind(new InetSocketAddress(port));
ssc.configureBlocking(false);
// 注册感兴趣的连接建立事件
ssc.register(selector, SelectionKey.OP_ACCEPT);

```

之后则可采取和客户端同样的方式对 selector.select 进行轮询，只是要增加一个对 key.isAcceptable 的处理，代码如下：

```

if(key.isAcceptable()){
    ServerSocketChannel server=(ServerSocketChannel)key.channel();
    SocketChannel sc=server.accept();
    if(sc==null){
        continue;
    }
    sc.configureBlocking(false);
    sc.register(selector,SelectionKey.OP_READ);
}

```

上面只是基于 TCP/IP+NIO 实现的一个简单例子，同样来看看基于 TCP/IP+NIO 如何支撑客户端同时发送多个请求及服务器端接受多个连接发送的请求。

对于客户端发送多个请求的需求，采用 TCP/IP+NIO 和采用 TCP/IP+BIO 的方式没有任何不同。但 NIO 方式可做到不阻塞，因此如果服务器端返回的响应能带上请求标识，那么客户端则可采用连接复用的方式，即每个 SocketChannel 在发送消息后，不用等响应即可继续发送其他消息，这种方式可降低连接池带来的资源争抢的问题，从而提升系统性能；对于连接不复用的情况，可基于 Socket.setSoTimeout 的方式来控制同步请求的超时；对于连接复用的情况，同步请求的超时可基于 BlockingQueue、对象的 wait/notify 机制或 Future 机制来实现。

对于服务器端接受多个连接请求的需求，通常采用的是由一个线程来监听连接的事件，另一个或多个线程来监听网络流读写的事件。当有实际的网络流读写事件发生后，再放入线程池中处理。这种方式比 TCP/IP+BIO 的好处在于可接受很多的连接，而这些连接只在有真实的请求时才会创建线程来处理，这种方式通常又称为一请求一线程。当连接数不多，或连接数较多，且连接上的请求发送非常频繁时，TCP/IP+NIO 的方式不会带来太大的优势，但在实际的场景中，通常是服务器端要支持大量的连接数，但这些连接同时发送的请求并不会非常多。

在基于 Sun JDK 开发 Java NIO 程序时，尤其要注意 selector.select 抛出 IOException 异常的处理<sup>6</sup>及 selector.select 不阻塞就直接返回的情况<sup>7</sup>。这两种状况都有可能造成 CPU 消耗达到 100%，对于 selector.select 抛出 IOException 的状况，可以采用绕过的方法为捕捉异常并将当前 Thread sleep 一段时

<sup>6</sup> [http://bugs.sun.com/view\\_bug.do?bug\\_id=6693490](http://bugs.sun.com/view_bug.do?bug_id=6693490)

<sup>7</sup> [http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=6403933](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6403933)

间，或是重新创建 Selector。为避免 selector.select 不阻塞就直接返回，可采用 bug 库中提到的修改建议。

从上面可以看出，对于高访问量的系统而言，TCP/IP+NIO 方式结合一定的改造在客户端能够带来更高的性能，在服务器端能支撑更高的连接数。

## UDP/IP+BIO

Java 对 UDP/IP 方式的网络数据传输同样采用 Socket 机制，只是 UDP/IP 下的 Socket 没有建立连接的要求，由于 UDP/IP 是无连接的，因此无法进行双向的通信。这也也就要求如果要双向通信的话，必须两端都成为 UDP Server。

在 Java 中可基于 DatagramSocket 和 DatagramPacket 来实现 UDP/IP+BIO 方式的系统间通信，DatagramSocket 负责监听端口及读写数据。DatagramPacket 作为数据流对象进行传输，基于这两个类实现客户端的关键代码如下：

```
// 由于 UDP/IP 是无连接的，如果希望双向通信，就必须启动一个监听端口，承担服务器的职责，如不能
// 绑定到指定端口，则抛出 SocketException
DatagramSocket serverSocket=new DatagramSocket(监听的端口);
byte[] buffer=new byte[65507];
DatagramPacket receivePacket=new DatagramPacket(buffer,buffer.length);
DatagramSocket socket=new DatagramSocket();
DatagramPacket packet=new DatagramPacket(datas,datas.length,server,port);
// 阻塞发送 packet 到指定的服务器和端口，当出现网络 IO 异常时抛出 IOException，当连不上目标地
// 址和端口时，抛出 PortUnreachableException
socket.send(packet)
// 阻塞并同步读取流信息，如接收到的流信息比 packet 长度长，则删除更长的信息，可通过调用
// DatagramSocket.setSoTimeout(以毫秒为单位的超时时间)来设置读取流的超时时间
serverSocket.receive(receivePacket)
```

服务器端代码和客户端代码的结构基本一致，这里就不列了。

由于 UDP/IP 通信的两端不建立连接，就不会有 TCP/IP 通信连接竞争的问题，只是最终读写流的动作是同步的。

对于服务器端同时接收多请求的需求，通常采取每接收到一个 packet 就放入一个线程中进行处理的方式来实现。

## UDP/IP+NIO

在 Java 中可通过 DatagramChannel 和 ByteBuffer 来实现 UDP/IP+NIO 方式的系统间通信，DatagramChannel 负责监听端口及进行读写，ByteBuffer 则用于数据流传输。基于这两个类实现客户端的关键代码如下：

```

DatagramChannel receiveChannel=DatagramChannel.open();
receiveChannel.configureBlocking(false);
DatagramSocket socket=receiveChannel.socket();
socket.bind(new InetSocketAddress(rport));
Selector selector=Selector.open();
receiveChannel.register(selector, SelectionKey.OP_READ);
之后即可采取和 TCP/IP+NIO 中对 selector 遍历一样的方式进行流信息的读取。
DatagramChannel sendChannel=DatagramChannel.open();
sendChannel.configureBlocking(false);
SocketAddress target=new InetSocketAddress("127.0.0.1", sport);
sendChannel.connect(target);
// 阻塞写入流，如发送缓冲区已满，则返回 0，此时可通过注册 SelectionKey.OP_WRITE 事件，以便在可写入时再进行写操作，方式和 TCP/IP+NIO 基本一致
sendChannel.write(ByteBuffer);

```

服务端代码和客户端代码基本一致，就不再一一描述。

从以上代码来看，对于 UDP/IP 方式，NIO 带来的好处是只在有流要读取或可写入流时才做相应的 IO 操作，而不用像 BIO 方式直接阻塞当前线程。

以上列举了基于 Java 包实现一对一的系统间通信的方式，在实际的场景中，通常还会将消息发送到多台机器，此时可以选择为每个目标机器建立一个连接，这种方式对于发送消息端会造成很大的网络流量压力。例如传输的消息是视频数据的场景，在网络协议上还有一个基于 UDP/IP 扩展出来的多播协议，多播协议的传输方式是一份数据在网络上进行传输，而不是由发送者给每个接收者都传一份数据，这样，网络的流量就大幅度下降了。

在 Java 中可基于 MulticastSocket 和 DatagramPacket 来实现多播网络通信，MulticastSocket 是基于 DatagramSocket 派生出来的类，其作用即为基于 UDP/IP 实现多播方式的网络通信。在多播通信中，不同的地方在于接收数据端通过加入到多播组来进行数据的接收，同样发送数据也要求加入到多播组进行发送，多播的目标地址具有指定的地址范围，在 224.0.0.0 和 239.255.255.255 之间。基于多播方式实现网络通信的服务器端关键代码如下：

```

// 组播地址
InetAddress groupAddress= InetAddress.getByName("224.1.1.1");
MulticastSocket server=new MulticastSocket(port);
// 加入组播，如地址为非组播地址，则抛出 IOException，当已经不希望再发送数据到组播地址，或
// 不希望再读取数据时，可调用 server.leaveGroup(组播地址)
server.joinGroup(groupAddress);
MulticastSocket client=new MulticastSocket();
client.joinGroup(groupAddress);

```

之后则可和 UDP/IP+BIO 一样通过 `receive` 和 `send` 方法来进行读写操作。

Client 端代码和服务端代码基本一致，就不再列举了。

在 Java 应用中，多播通常用于多台机器的状态的同步。例如 JGroups，默认基于 UDP/IP 多播协议，由于 UDP/IP 协议在数据传输时不够可靠，对于可靠性要求很高的系统，会希望采用多播方式，同时又要做到可靠。对于这样的需求，业界提出了一些能够确保可靠实现多播的方式：SRM（Scalable Reliable Multicast）<sup>8</sup>、URGCP（Uniform Reliable Group Communication Protocol），其中 SRM 是在 UDP/IP 多播的基础上增加了确认机制，从而保证可靠，eBay 采用了 SRM 框架来实现将数据从主数据库同步到各个搜索节点机器<sup>9</sup>。

从上面的介绍来看，使用 Java 包来实现基于消息方式的系统间通信还是比较麻烦。为了让开发人员能更加专注对数据进行业务处理，而不用过多关注纯技术细节，开源业界诞生了很多优秀的基于以上各种协议的系统间通信的框架。这其中的佼佼者就是 Mina 了，1.1.2 节将会介绍这个佼佼者。

## 1.1.2 基于开源框架实现消息方式的系统间通信

这一节讲述基于 Mina 如何实现消息方式的系统间通信，同时分析开源通信框架的优势。

Mina<sup>10</sup> 是 Apache 的顶级项目<sup>10</sup>，基于 Java NIO 构建，同时支持 TCP/IP 和 UDP/IP 两种协议。Mina 对外屏蔽了 Java NIO 使用的复杂性，并在性能上做了不少的优化。

在使用 Mina 时，关键的类为 `IoConnector`、`IoAcceptor`、`IoHandler` 及 `IoSession`，Mina 采用 Filter Chain 的方式封装消息发送和接收的流程，在这个 Filter Chain 过程中可进行消息的处理、消息的发送和接收等。

`IoConnector` 负责配置客户端的消息处理器、IO 事件处理线程池、消息发送/接收的 Filter Chain 等。

`IoAcceptor` 负责配置服务器端的 IO 事件处理线程池、消息发送/接收的 Filter Chain 等。

`IoHandler` 作为 Mina 和应用的接口，当发生了连接事件、IO 事件或异常事件时，Mina 都会通知应用所实现的 `IoHandler`。

`IoSession` 有点类似 `SocketChannel` 的封装，不过 Mina 对连接做了进一步的抽象，因此可进行更多连接的控制及流信息的输出。

基于 Mina 实现 TCP/IP+NIO 客户端的关键代码如下：

8 [http://ee.lbl.gov/papers/srm\\_ton.pdf](http://ee.lbl.gov/papers/srm_ton.pdf)

9 <http://highscalability.com/ebay-architecture>

10 <http://mina.apache.org>

```

// 创建一个线程池大小为 CPU 核数+1 的 SocketConnector 对象
SocketConnector ioConnector = new
SocketConnector(Runtime.getRuntime().availableProcessors() + 1,
Executors.newCachedThreadPool());
// 设置 TCP NoDelay 为 true
ioConnector.getDefaultConfig().getSessionConfig().setTcpNoDelay(true);
// 增加一个将发送对象进行序列化以及接收字节流进行反序列化的类至 filter Chain
ioConnector.getFilterChain().addLast("stringserialize", new
ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
// IoHandler 的实现，以便当 mina 建立连接、接收到消息后通知应用
IoHandler handler=new IoHandlerAdapter(){

    public void messageReceived(IoSession session, Object message)
        throws Exception {
        System.out.println(message);
    }

};

// 异步建立连接
ConnectFuture connectFuture = ioConnector.connect(socketAddress,handler);
// 阻塞等待连接建立完毕，如须设置连接创建的超时时间，可调用
SocketConnectorConfig.setConnectTimeout(以秒为单位的超时时间)
connectFuture.join();
IoSession session=connectFuture.getSession();
// 发送对象
session.write(Object);

```

使用 Mina 后，客户端的代码变得简单多了。

服务器端关键代码如下：

```

// 创建一个线程池大小为 CPU 核数+1 的 IoAcceptor 对象
final IoAcceptor acceptor=new
SocketAcceptor(Runtime.getRuntime().availableProcessors() + 1,
Executors.newCachedThreadPool());
acceptor.getFilterChain().addLast("stringserialize", new
ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
IoHandler handler=new IoHandlerAdapter(){

    public void messageReceived(IoSession session, Object message)
        throws Exception {
        // 接收到客户端发送的对象
    }

};

// 绑定监听的端口以及当有连接建立、接收到对象等事件发送时需要通知的 IoHandler 对象
acceptor.bind(new InetSocketAddress(port), handler);

```

采用 Mina 后，服务器端代码无须再关注建立连接时的 OP\_ACCEPT 的事件，同样也无须去注册 OP\_READ、OP\_WRITE 这些 Key 的事件，取而代之的是更友好地使用接口。通过这些封装使用者可以非常方便地使用，而无须过多考虑 Java NIO 的用法。但 Mina 2.0 之前的版本中并未提供连接的管理（连接的创建、自动重连、连接的心跳、连接池等）、同步发送数据支持，因此在实际使用中通常还需在 Mina 的基础上进行封装。

在使用 Mina 2.0 之前的版本时，以下几个方面值得注意：

- 使用自定义的 ThreadModel

通过 `SocketConnectorConfig.setThreadModel(ThreadModel.MANUAL)` 将线程模式改为自定义模式，否则 Mina 会自行启动一个最大线程数为 16 个的线程池来处理具体的消息，这对于多数应用而言都不适用，因此最好是自行控制具体处理消息的线程池。

- 合理配置 IO 处理线程池

在创建 `SocketAcceptor` 或 `SocketConnector` 时要提供一个线程池及最大的线程数，也就是 Mina 用于 IO 事件处理的线程数，通常建议将这个线程数配置为 CPU 核数+1。

- 监听是否成功写入操作系统的发送缓冲区

在调用 `IoSession.write` 时，Mina 并不确保其会成功写入操作系统的发送缓冲区中（例如写入时连接刚好断开），为了确定是否成功，可采用如下方法：

```
WriteFuture writeResult=session.write(data);
writeResult.addListener(new IoFutureListener() {
    public void operationComplete(IoFuture future) {
        WriteFuture wfuture=(WriteFuture)future;
        // 写入成功
        if(wfuture.isWritten()){
            return;
        }
        // 写入失败，自行进行处理
    }
});
```

这对于同步请求而言特别重要，通常同步请求时都会设置一个等待响应的超时时间，如果不去监听是否成功写入的话，那么同步的请求一直要等到设定的超时时间才能返回。

- 监听写超时

当接收消息方的接收缓冲区占满时，发送方会出现写超时的现象，这时 Mina 会向外抛出 `WriteTimeoutException`，如有必要，可在 `IoHandler` 实现的 `exceptionCaught` 方法里进行处理。

- 借助 Mina IoSession 上未发送的 bytes 信息实现流控

当 IoSession 上堆积了过多未发送的 byte 时，会造成 jvm 内存消耗过多的现象。因此通常要控制 IoSession 上堆积的未发送的 byte 量，此值可通过 Mina IoSession 的 getScheduledWriteBytes 来获取，从而进行流控。

- messageReceived 方法占用 IO 处理线程

在使用 Thread.MANUAL 的情况下，IOHandler 里的 messageReceived 方法会占用 Mina 的 IO 处理线程，为了避免业务处理接收消息的速度影响 IO 处理性能，建议在此方法中另起线程来做业务处理。

- 序列化/反序列化过程会占用 IO 处理线程

由于 Mina 的序列化/反序列化过程是在 FilterChain 上做的，同样会占据 IO 处理线程。Mina 将同一连接上需要发送和接收的消息放在队列中串行处理。如果序列化/反序列化过程耗时较长，就会造成同一连接上其他消息的接收或发送变慢。

- 反序列化时注意继承 CumulativeProtocolDecoder

在使用 NIO 的情况下，每次读取的流并不一定完整，因此要通过继承 CumulativeProtocolDecoder 来确保当流没读完时，下次接着读，这同时也要求应用在协议头中保持此次发送流的长度信息。

- Mina 1.1.6 及以前的版本中 sessionClosed 可能会不被调用的 bug

在某些高压力的情况下，当连接断开时，Mina 1.1.6 及以前的版本并不会调用 IoHandler 中的 sessionClosed 方法，这对于某些要在 sessionClosed 做相应处理的应用来说会出现问题，这个 bug<sup>11</sup> 在 Mina 1.1.7 的版本中已修复。

除了 Mina 之外，JBoss Netty 也是现在一个广受关注的 Java 通信框架，其作者也是 Mina 的作者（Trustin Lee），据评测 JBoss Netty 的性能好于 Mina<sup>12</sup>，如读者感兴趣，可访问 <http://www.jboss.org/netty> 来了解更多的细节。

以上两节介绍了基于 Java 自身包及开源通信框架来实现消息方式的系统间通信，Java 系统内的通信都是以 Java 对象调用的方式来实现的，例如 A a = new AImpl(); a.call();，但当系统变成分布式后，就无法用以上的方式直接调用了，因为在调用端并不会有 AImpl 这个类。这时如果通过基于以上的消息方式来做，对于开发而言就会显得比较晦涩了，因此 Java 中也提供了各种各样的支持对象方式的系统间通信的技术，例如 RMI、WebService 等。同样，在 Java 中也有众多的开源框架提供了 RMI、WebService 的实现和封装，例如 Spring RMI、CXF 等，下面来看看基于远程调用方式如何实现系统间的通信。

<sup>11</sup> <https://issues.apache.org/jira/browse/DIRMINA-549>

<sup>12</sup> <http://www.jboss.org/netty/performance>

## 1.2 基于远程调用方式实现系统间的通信

远程调用方式就是尽可能地使系统间的通信和系统内一样，让使用者感觉调用远程同调用本地一样，但其实并没有办法做到完全透明，例如由于远程调用带来的网络问题、超时问题、序列化/反序列化问题、调试复杂的问题等，在远程调用时要注意对这些问题的处理。

### 1.2.1 基于Java自身技术实现远程调用方式的系统间通信

在Java中实现远程调用方式的技术主要有RMI和WebService两种，下面分别来看看基于这两种技术如何实现远程调用方式的系统间通信。

#### RMI

RMI（Remote Method Invocation）是Java用于实现透明远程调用的重要机制。在远程调用中，客户端仅有服务器端提供的接口。通过此接口实现对远程服务器端的调用。

远程调用基于网络通信来实现，RMI同样如此，其机制如图1.1所示：

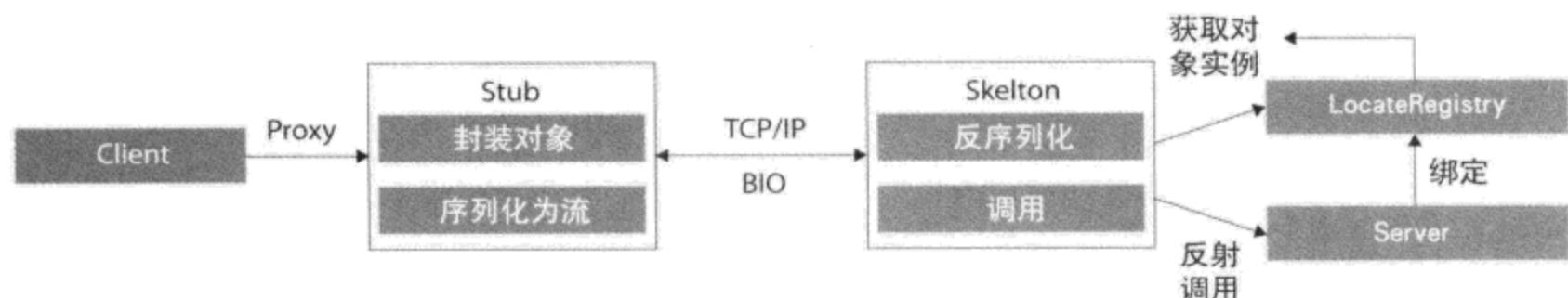


图1.1 RMI机制

Sun JDK 6.0以前版本中的RMI实现均是基于TCP/IP+BIO方式的，RMI服务器端通过启动RMI注册对象在一个端口上监听对外提供的接口，其实现实例以字符串的方式绑定到RMI注册对象上。RMI客户端通过proxy的方式代理了对服务器端接口的访问，RMI客户端将要访问的服务器端对象字符串、方法和参数封装成一个对象，序列化成流后通过TCP/IP+BIO传输到RMI服务器端。RMI服务器端接收到客户端的请求对象后，解析其中的对象字符串、方法及参数，通过对象字符串从RMI注册对象上找到提供业务功能的实例，之后结合要访问的方法来反射获取到方法实例对象，传入参数完成对服务器端对象实例的调用，返回的结果则序列化为流以TCP/IP+BIO方式返回给客户端，客户端在接收到此流后反序列化为对象，并返回给调用者。

RMI要求服务器端的接口继承Remote接口，接口上的每种方法必须抛出RemoteException，服务器端业务类通过实现此接口提供业务功能，然后通过调用UnicastRemoteObject.exportObject来将此对象绑定到某端口上，最后将此对象注册到本地的LocateRegistry上，此时形成一个字符串对应于对象实例的映射关系。基于RMI实现示例中的服务器端代码如下。

```

服务器端对外提供的接口
public interface Business extends Remote{
    public String echo(String message) throws RemoteException;
}

服务器端实现此接口的类
public class BusinessImpl implements Business {
    public String echo(String message) throws RemoteException {
        if("quit".equalsIgnoreCase(message.toString())){
            System.out.println("Server will be shutdown!");
            System.exit(0);
        }
        System.out.println("Message from client:"+message);
        return "Server response:"+message;
    }
}

基于 RMI 的服务器端
public class Server {

    public static void main(String[] args) throws Exception{
        int port=9527;
        String name="BusinessDemo";
        Business business=new BusinessImpl();
        UnicastRemoteObject.exportObject(business, port);
        Registry registry=LocateRegistry.createRegistry(1099);
        registry.rebind(name, business);
    }
}

```

RMI 的客户端首先通过 LocateRegistry.getRegistry 来获取 Registry 对象，然后通过 Registry.lookup 字符串获取要调用的服务器端接口的实例对象，最后以接口的方式透明地调用远程对象的方法。按照以上描述，基于 RMI 实现客户端的关键代码如下：

```

Registry registry=LocateRegistry.getRegistry("localhost");
String name="BusinessDemo";
// 创建 BusinessDemo 类的代理类，当调用时则调用 localhost:1099 上名称为 BusinessDemo 的对象，如服务器端没有对应名称的绑定，则抛出 NotBoundException；如通信出现错误，则抛出 RemoteException
Business business=(Business) registry.lookup(name);

```

从上面示例代码来看，基于 RMI 实现的客户端和服务端较之基于 TCP/IP+NIO 等实现的客户端和服务端简单很多，代码可维护性也高很多。

## WebService

**WebService** 是一种跨语言的系统间交互标准。在这个标准中，对外提供功能的一方以 HTTP 的方式提供服务，该服务采用 WSDL（Web Service Description Language）描述，在这个文件中描述服务所使用的协议、所期望的参数、返回的参数格式等。调用端和服务端通过 SOAP（Simple Object Access Protocol）方式进行交互。

在 Java 中使用 WebService 须首先将服务器端的服务根据描述生成相应的 WSDL 文件，并将应用及此 WSDL 文件放入 HTTP 服务器中，借助 Java 辅助工具根据 WSDL 文件生成客户端 stub 代码。此代码的作用是将产生的对象请求信息封装为标准的 SOAP 格式数据，并发送请求到服务器端，服务器端在接收到 SOAP 格式数据时进行转化，反射调用相应的 Java 类，过程如图 1.2 所示：

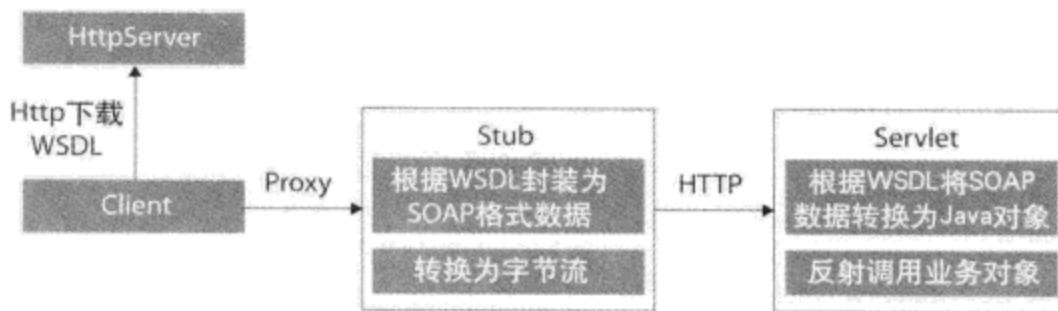


图 1.2 WebService 调用过程

Java SE6 中集成了 WebService，因此可以直接实现该方式的远程调用，服务器端通过 @WebService 来标记对外暴露的 WebService 实现类，通过调用 Endpoint.publish 将此 WebService 实现发布到指定的 HTTP 地址上。客户端通过 wsimport 来访问相应地址的 wsdl 文件，从而生成调用服务器端的辅助类，应用即可通过调用此类来实现远程调用了。基于 WebService 实现示例中的服务器端代码如下：

对外暴露的接口：

```

public interface Business {
    /**
     * 显示客户端提供的信息，并返回
     */
    public String echo(String message);
}
  
```

服务器端的实现类，通过 @WebService 来指定对外提供的 WebService 的名称和客户端生成的类名及包名：

```

@WebService(name="Business", serviceName="BusinessService", targetNamespace="http://WebService.chapter1.book/client")
@SOAPBinding(style=SOAPBinding.Style.RPC)

public class BusinessImpl implements Business {
  
```

```

public String echo(String message) {
    if("quit".equalsIgnoreCase(message.toString())){
        System.out.println("Server will be shutdown!");
        System.exit(0);
    }
    System.out.println("Message from client: "+message);
    return "Server response: "+message;
}

}

```

发布 WebService 的类：

```

public static void main(String[] args) {
    Endpoint.publish("http://localhost:9527/BusinessService", new
BusinessImpl());
    System.out.println("Server has been started");
}

```

客户端通过 JDK bin 目录下的 wsimport 命令来生成辅助调用代码，执行如下命令生成辅助代码：

```
wsimport -keep http://localhost:9527/BusinessService?wsdl
```

执行后，在当前目录下会生成 book/chapter1/WebService/client/ Business.java 和 book/chapter1/ WebService/client/ BusinessService.java 的代码，基于这两个生成的代码编写客户端的关键代码如下：

```

BusinessService businessService=new BusinessService();
Business business=businessService.getBusinessPort();
business.echo(command);

```

WebService 传输的数据协议采用 SOAP，SOAP 对于复杂的对象结构比较难支持，其好处是能够支持跨语言。

无论是采用 RMI 还是 WebService，都封装了网络通信的细节，因此使用起来会比较简单，但如果想对通信细节做一些调优或控制，也会比较麻烦。

## 1.2.2 基于开源框架实现远程调用方式的系统间通信

### Spring RMI

Spring RMI 是 Spring Remoting 中的一个子框架，基于 Spring RMI 可以很简单地就实现 RMI 方式的 Java 远程调用，Spring RMI 的工作原理如图 1.3 所示：

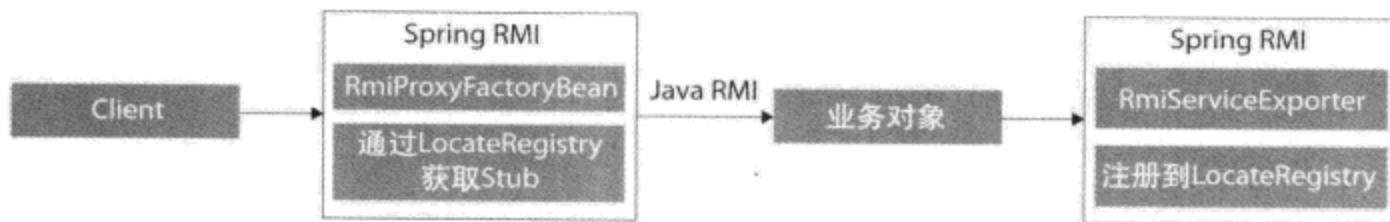


图 1.3 Spring RMI 工作原理

基于 Spring RMI 实现示例中的服务器端代码如下。

对外提供的接口类：

```

public interface Business {
    /**
     * 显示客户端提供的信息，并返回
     */
    public String echo(String message);
}
  
```

服务器端的实现类：

```

public class BusinessImpl implements Business {
    public String echo(String message) {
        if("quit".equalsIgnoreCase(message.toString())){
            System.out.println("Server will be shutdown!");
            System.exit(0);
        }
        System.out.println("Message from client:"+message);
        return "Server response:"+message;
    }
}
  
```

Spring 描述文件：

```

<bean id="businessService" class="book.chapter1.springrmi.BusinessImpl"/>

<bean id="rmiBusinessService"
class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="service"><ref bean="businessService"/></property>
    <property name="serviceName"><value>BusinessService</value></property>
    <property name="serviceInterface"><value>book.chapter1.springrmi.
        Business</value> </property>
</bean>
  
```

启动 Spring 容器，并让外部能够以 RMI 的方式访问到 BusinessImpl：

```
public static void main(String[] args) throws Exception{
    new
    ClassPathXmlApplicationContext("book/chapter1/springrmi/server.xml");
    System.out.println("Server has been started");
}
```

客户端代码如下。

Spring 描述文件：

```
<bean id="businessService"
      class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl">
      <value>rmi://localhost/BusinessService</value>
    </property>
    <property name="serviceInterface">
      <value>book.chapter1.springrmi.Business</value>
    </property>
  </bean>
```

客户端通过启动 Spring 容器，获取相应的 Spring bean 后即可以 RMI 方式调用远程的对象了：

```
ApplicationContext ac=new
ClassPathXmlApplicationContext("book/chapter1/springrmi/client.xml");
Business business=(Business)
ac.getBean("businessService");business.echo(command);
```

## CXF

**CXF** 是 Apache 的顶级项目，也是目前 Java 社区中用来实现 WebService 流行的一个开源框架（尤其是收编了 xfire 后）。基于 CXF 可以非常简单地以 WebService 的方式来实现 Java 甚至是跨语言的远程调用，CXF 的工作原理如图 1.4 所示：

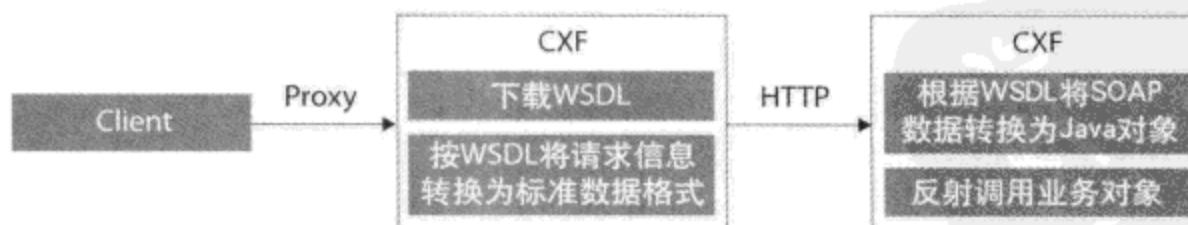


图 1.4 CXF 工作原理

CXF 对于 WebService 的服务器端并没做多少封装，它仍然采用目前 Java SE 本身的 WebService 方式，只是提供了一个 JaxWsServerFactoryBean 类，从而可以在 WebService 被调用时增加一些拦截器的处理。客户端方面 CXF 则增加了封装，以便能够直接以接口的方式来调用远程的 WebService，

简化了调用 WebService 的复杂性，CXF 提供的类为 JaxWsProxyFactoryBean，通过此类将 WebService 的接口类以及 WebService 的地址放入，即可获取对应接口的代理类了，基于 CXF 实现示例中的服务器端代码如下：

接口类：

```
@WebService
public interface Business {

    public String echo(String message);

}
```

业务实现类：

```
@WebService(serviceName="BusinessService", endpointInterface="book.chapter1.cxf
.Business")
public class BusinessImpl implements Business {

    public String echo(String message) {
        if("quit".equalsIgnoreCase(message.toString())){
            System.out.println("Server will be shutdown!");
            System.exit(0);
        }
        System.out.println("Message from client: "+message);
        return "Server response: "+message;
    }

}
```

调用 CXF 类完成 WebService 的发布：

```
public static void main(String[] args) throws Exception{
    Business service=new BusinessImpl();
    JaxWsServerFactoryBean svrFactory=new JaxWsServerFactoryBean();
    svrFactory.setServiceClass(Business.class);
    svrFactory.setAddress("http://localhost:9527/business");
    svrFactory.setServiceBean(service);
    svrFactory.create();
}
```

客户端由于 CXF 提供了良好的封装，调用 WebService 就比直接用 Java 的方式简单多了，关键代码如下：

```
JaxWsProxyFactoryBean factory=new JaxWsProxyFactoryBean();
factory.setServiceClass(Business.class);
factory.setAddress("http://localhost:9527/business");
Business business=(Business)factory.create();
```

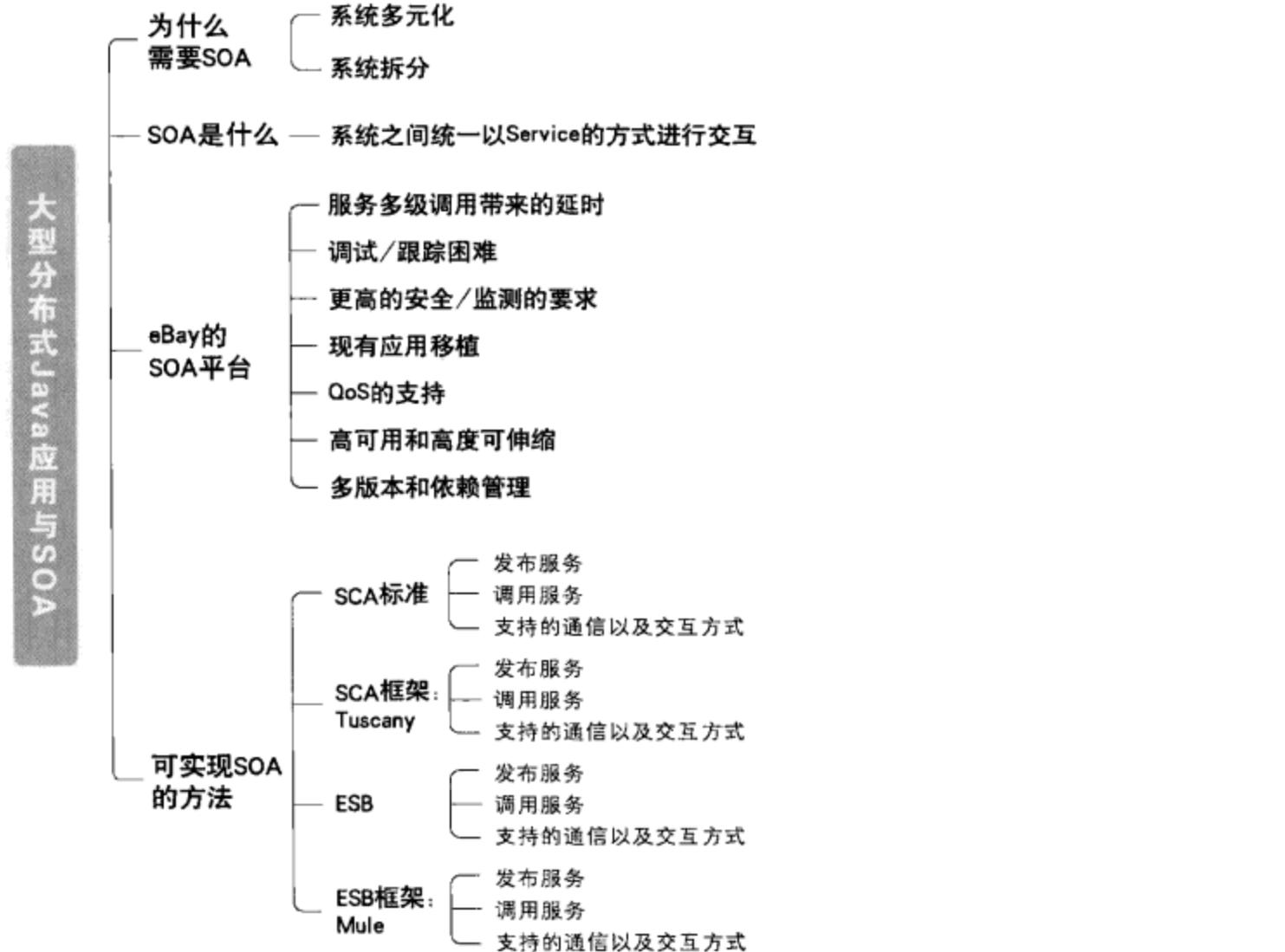
类似 CXF 这样的 WebService 框架在开源界中还有不少，例如 Axis。这些开源 WebService 框架除了对 Java WebService 的使用方式进行封装，提升了易用性外，还增加了多种协议来调用 WebService，例如 TCP/IP 和 JMS 等。

以上的章节讲解了 Java 中实现系统间通信的相关技术、基本原理及使用方法。由此也可看出，构建分布式的 Java 应用比构建集中式的复杂不少，要掌握的知识点多了很多。本章并未面面俱到地介绍 Java 中可用来实现系统间通信的技术。例如 JMS、EJB 等，这些技术通常是基于以上所描述的原理进一步扩展，以提升对于不同场景的需求满足度及易用性。在实际使用中不用拘泥于这些现有技术，可以选择已有原理来自行构建更为适合需求的框架，例如可以基于 RMI 原理来构建更为轻量级的远程调用框架。

在大型的应用中，分布式的概念不仅仅是上面的跨 JVM 或跨机器的系统间通信，还会涵盖如集群、负载均衡、分布式缓存、分布式文件系统等多方面的内容。这些将在后续的章节中进一步讲解，接下来先看看一个大型的分布式 Java 应用会是多大的规模，将面对哪些方面的挑战。



# 第2章 大型分布式Java应用与SOA



当应用获得用户的认可后，会不断地发展。以豆瓣网<sup>1</sup>为例，早期豆瓣网只有书评的功能，随着大家对豆瓣的认可，以及使用的用户越来越多，其发展出今天的豆瓣社区、豆瓣读书、豆瓣电影和豆瓣音乐等功能，这些功能有各自的特色，但又有很多可公用的业务逻辑。例如用户信息、评价等，如这四个系统都维护自己的用户信息读写或评价业务逻辑，会造成问题。一方面是当要修改评价逻辑或修改用户信息的读取方式时，所有系统都要修改，会相当复杂；另一方面是每个系统上都有多种业务逻辑，这就像在一个小超市中，一个人负责收银、清洁、摆货、咨询等各种各样的事情，当来超市的顾客多到一定程度，这个人就无法再负责这么多的事情了，系统也同样如此。

第一个现象是系统多元化带来的问题，可采用对共用逻辑的部分进行抽象的方法，形成多个按领域划分的共用业务逻辑系统；第二个现象是系统访问量、数据量上涨后带来的典型问题，当超市的顾客不断增加时，通常超市会采取分工的方式更好地服务顾客。同样，对于系统而言，也会采取拆分系统的方式来解决。

在构建了共用业务逻辑系统和拆分系统后，最明显的问题就是系统之间如何交互。如果不控制，会出现多个系统之间存在多种交互方式：Http、TCP/IP+NIO、Hessian、RMI、WebService 等；同步、异步等方式可能都会出现，这会导致开发人员每访问一个共用业务逻辑系统或拆分出来的系统后，都有可能要学习不同的交互方式；同时也会造成各开发团队重复造轮子，提供不同交互方式用的框架，这对于应用的性能、可用性而言也带来了极大的挑战。

对于上面的问题，很容易想到的解决方法就是统一交互的方式，SOA 无疑是实现这种方式的首选。

SOA 全称是面向服务架构，它强调系统之间以标准的服务方式进行交互，各系统可采用不同的语言、不同的框架来实现，交互则全部通过服务的方式进行。

eBay 也实现了一个 SOA 平台来支撑其业务的多元化发展<sup>2</sup>，eBay 认为 SOA 能给它带来的最大好处是提升业务的可重用性及灵敏性。SOA 平台除了要实现统一的交互外，还会碰到其他多方面的挑战。来看看 eBay 眼中 SOA 平台会带来哪些挑战。

- 服务多级调用带来的延时

当出现一项功能要调用服务 A，服务 A 又要调用服务 B，服务 B 又要调用服务 C 时，就会带来大幅度的延时。解决延时要做到高性能的服务交互，以及完善的服务调用过程控制。例如当调用在服务 B 执行时已超时，那么就没有必要继续调用服务 C，而应该直接抛出超时异常给调用端。

- 调试/跟踪困难

例如当调用服务 A，服务 A 报错时，调用者通常会去找服务 A 的开发人员来解决。服务 A 的开发人员去查问题，有可能认为是服务 B 报错，服务 B 的开发人员去查，有可能又认为是网络的问题。在这样的情况下，就会导致出现问题时调试/跟踪非常困难。

1 <http://www.douban.com>

2 [http://qconsf.com/sf2009/file?path=/qcon-sanfran-2009/slides/SastryMalladi\\_SOAEBayHowIsItAHit.pdf](http://qconsf.com/sf2009/file?path=/qcon-sanfran-2009/slides/SastryMalladi_SOAEBayHowIsItAHit.pdf)

- 更高的安全/监测的要求

在未拆分系统时，对系统的安全和监测都只须在一个系统上做即可。在拆分后，就要求在每个系统上都有相应的安全控制及监测。

- 现有应用移植

这一点是 SOA 平台在推广时的大挑战，移植的应用功能越多，花费的时间就越多。

- QoS 的支持

每个服务提供者能够支撑的访问量是有限的，因此设定服务的 QoS 非常重要，并且应尽可能采取一系列的措施来保障 QoS，例如流量控制、机器资源分配等。

- 高可用和高度可伸缩

这是互联网应用必须做到的两点，SOA 平台承担了所有服务的交互，因此其可用性及伸缩性就更影响巨大。

- 多版本和依赖管理

随着服务不断发展，以及使用面不断扩大，服务多版本的存在会成为一个大的需求，同时由于服务众多，这些服务的依赖关系也要管理起来，以便在升级时能进行相应的安排。

eBay 根据这些挑战自行实现了一个 SOA 平台，这个 SOA 平台主要包含了以下几点：

- 1) 高性能、可扩展的轻量级框架；
- 2) 对监测、安全控制、流量控制的支持；
- 3) 服务注册和服务仓库；
- 4) 开发工具。

在推行 SOA 平台时，eBay 采取的是按领域驱动的方式划分服务，同时不断培训开发人员，以便让开发人员明确服务化后和之前单系统方式的区别，例如可能会产生网络通信错误、超时等。

综上所述，对于一个大型应用中的 SOA 平台，至少应包含以下几点功能：

- 1) 统一的服务交互方式，并可实现和现有应用的无缝集成；
- 2) 提供调试/跟踪的支持；
- 3) 依赖管理；
- 4) 高性能及高可用。

在实现 SOA 时，可参考的标准或概念有 SCA、ESB，同时业界也有一些实现了 SCA、ESB 的框架，下面就具体介绍 SCA、ESB、SCA 及 ESB 框架。

## 2.1 基于SCA实现SOA平台

SOA思想推广了几年后，业界对于SOA的评价均为思想不错，但没有实际的实现方法指导。于是业界领先的几家厂商：IBM、Oracle（包括之前的BEA、Sun）、Red Hat、SAP及Siemens等成立了一个组织<sup>3</sup>，负责制定SOA的具体实现规范。在2007年3月左右完成了此规范的大部分内容的制定，规范的名字定为Service Component Architecture，简称SCA，版本为1.0，下面就具体来看看SCA标准。

按照之前对SOA平台的描述，首先来看看SCA标准是如何定义服务的交互的，包括了如何发布服务、如何调用服务及支持的通信协议和交互方式三个方面。

### 发布服务

服务遵照SOA以接口方式对外提供，发布服务首先要求系统本身已经有相应的接口实现。为了减少对系统实现的侵入，通过XML定义Component映射到系统本身的接口实现上。SCA支持了多种映射方式，并允许自行扩展，因此系统可采用Java、Spring Bean等多种方式来实现接口。在定义了Component后，即可将Component实现的接口以服务的方式发布，下面举例说明如何将系统中的接口实现发布为SCA服务。

系统中有一个名为chapter2.component.DefaultHelloWorld的类，它实现了chapter2.service.HelloWorld接口，要将此Java类发布为SCA服务，提供chapter2.service.HelloWorld接口定义的功能，编写一个如下格式的XML文件即可：

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
targetNamespace="http://foo.com"
name="HelloWorldComposite" >
    <component name="HelloWorldComponent">
        <implementation.java class=" chapter2.component.DefaultHelloWorld"/>
    </component>
    <service name="HelloWorldService" promote="HelloWorldComponent">
        <interface.java interface=" chapter2.service.HelloWorld"/>
    </service>
</composite>
```

在以上XML文件中，composite是SCA定义的最小部署单位。每个XML文件的根元素必须为composite，在composite下可以有多个component及service标签，来具体看看component标签和服务标签，这些主要标签的关系如图2.1所示：

<sup>3</sup> <https://osoa.org>

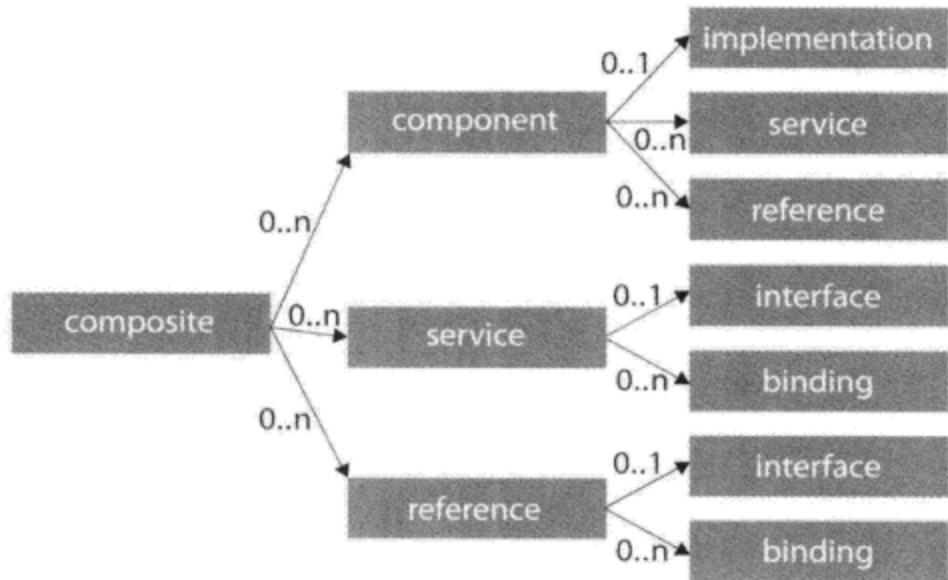


图 2.1

- **component 标签**

component 标签主要通过 implementation 子标签定义和已有系统的集成，SCA 提供了对多种实现集成的支持，例如 Spring、Java 或 C 等。

除了 implementation 子标签外，component 中还可定义 service、reference 等子标签，service 子标签用于表明当前 component 对外提供了什么 service；reference 子标签用于表明当前 component 引用了什么 service。

- **service 标签**

service 标签代表了对外提供服务的描述，包含了 name、promote、requires 和 policySets 四个属性。其中 promote 属性指定了提供此服务的 Component，Component 可以以多种方式实现，如纯 Java 方式，或 Spring 方式等。

service 标签下增加 interface 子标签即可定义对外提供的接口，SCA 标准规定支持三种方式对外提供接口：Java 语言方式的接口、WSDL 1.1 以及 WSDL 2.0 方式。可将 interface 标签写为 interface.Java 或 interface.wsdl，Java 方式的情况还支持 callbackInterface 属性，以满足双向的服务交互的需要。

对于发布的服务，可通过配置 binding 标签来指定其发布的方式，SCA 默认支持的方式有 SCA、WebService 以及 JMS 三种，如不指定则以服务的实现方式来决定。

从 service 标签来看，在采用 SCA 的情况下，可以很方便地将各种方式实现的功能以多种方式对外提供，对系统的侵入非常小。

## 调用服务

调用服务的方式也同样可通过简单地定义 xml 即可实现，例如要在 Spring 中调用上面的 HelloWorldService，只须定义如下的 xml 文件即可：

```

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
targetNamespace="http://foo.com"
name="HelloWorldComposite" >
  <component name="HelloWorldComponent">
    <implementation.spring location="beans.xml"/>
    <reference name="HelloWorldService" target="HelloWorldService"/>
  </component>
  <reference name="HelloWorldService" promote="HelloWorldComponent">
    <interface.java interface="chapter2.service.HelloWorld"/>
  </reference>
</composite>

```

Spring的beans.xml定义如下：

```

<beans>
  <sca:reference name="HelloWorldService" type=" chapter2.service.HelloWorld"/>
</beans>

```

通过这样的方式就可以直接在Spring获取HelloWorldService bean来调用SCA Service了，具体用到的reference标签如下。

reference标签代表了调用其他服务的描述，包含了name、promote、requires、policySets、multiplicity、target和wiredByImpl等属性，通过设置promote和name来指定此引用的服务需要注入的Component及其属性。multiplicity可用于指定需要引用的服务的数量：0..1、1..1、0..n或1..n。

和service标签一样，reference标签通过interface标签来指定要引用的服务的接口，通过bindings标签来指定引用服务的方式，例如SCA、WebService或JMS方式。

从reference标签来看，和service标签一样，SCA可以在不侵入系统的状况下以多种方式引用SCA服务，并将其注入需要引用服务的系统中，这些系统同样可以以Java、Spring或C++等各种方式实现。

## 支持的通信和交互方式

从之前对于发布服务和调用服务的描述中可看到，SCA标准默认提供的通信方式为SCA、WebService和JMS三种。SCA方式是指由框架根据运行状况来选择采用相应的通信方式，例如框架发现需要调用的服务在同一JVM中，则会自动切换为本地调用，如在不同JVM中，则会采用WebService或JMS等方式；WebService的实现为HTTP方式；JMS则可用多种方式来实现，例如TCP/IP、HTTP等，这取决于具体的SCA框架。

除了以上三种外，在SCA中也可非常方便地扩展其他通信方式。

在交互方式上，SCA标准中没有明确的定义。

从标准来看，在统一的服务交互方式上，SCA提供了清晰的发布服务、调用服务及无缝和现有应

用集成的支持。在调试/跟踪的支持上 SCA 标准中没有明确的定义，只能自行实现了；依赖管理方面也同样没有明确的定义，而且由于 SCA 标准中未定义服务仓库，这会导致依赖管理实现起来会比较困难；高性能及高可用这两点不好在标准中进行定义，完全取决于具体框架的实现。

## 2.2 基于 ESB 实现 SOA 平台

ESB 和 SCA 不同，它并不是由多个厂家联合制定的 SOA 实现的标准，可以认为 ESB 只是个概念，核心思想是基于消息中间件来实现系统间的交互。基于消息中间件所构建的此系统交互的中间场所称为总线，系统间交互的数据格式采用统一的消息格式，由总线完成消息的转化、路由，发送到相应的目标应用，基于 ESB 构建的系统结构如图 2.2 所示：

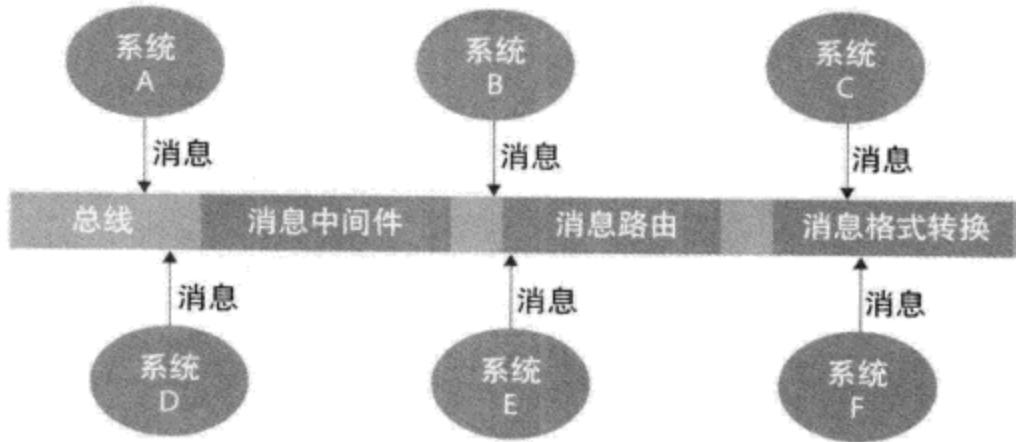


图 2.2

通常 ESB 框架须具备以下 5 个要素：

- 标准的消息通信格式

ESB 框架中要定义系统发送及接收消息时的消息格式，以便各个系统保持同样的方式与总线通信。

- 消息路由

消息路由是指当总线接到消息后，根据消息中的数据来决定需要调用的系统。在更为复杂的情况下，还可基于消息路由实现功能编排，即当某个功能需要由多个系统共同完成时，可在总线上以流程的方式编排访问系统的顺序。例如某功能需要首先访问 A 系统，然后根据 A 系统返回的结果来决定是访问 B 系统还是 C 系统，在访问了 B 系统或 C 系统后又要根据结果同时提交给 D、E 系统执行，在这种情况下如果有流程编排的话实现起来就很方便。

- 支持多种的消息交互类型

消息交互时要支持请求/响应和发布/订阅等方式，请求/响应方式会更加方便实现同步请求，发布/订阅方式则更加方便实现异步的消息广播。

- 支持多种网络协议

总线要和多个系统进行交互，通常要支持多种网络协议，例如 TCP/IP、UDP/IP、HTTP 等。

- 支持多种数据格式并能够进行相互转换

多个系统均须发送消息至总线，并由总线将消息转发，但各个系统消息中的数据格式可能不一致，此时总线要支持数据的转换。

结合之前对 SOA 平台的描述，在统一以服务方式进行交互这点上，可以认为 ESB 中的消息方式交互承担了这一功能，且支持多种通信及交互（同步、异步）方式，在调试/跟踪支持上没有定义，在依赖管理上，由于所有的交互都通过总线来进行，在此基础上可根据消息的流转来判断和形成各个系统的依赖关系。在高性能和高可用这两点上，则取决于实现框架。

在介绍完 SCA 和 ESB 后，下面就来看看基于 SCA 框架及 ESB 框架实现 SOA 平台的方式。

## 2.3 基于 Tuscany 实现 SOA 平台

在 SCA 实现框架中，本书选择了 Tuscany 1.5 来分析，Tuscany 是 IBM 和 Bea 捐献给 Apache 的产品，这也是目前最常用的 SCA 实现框架之一。以下为一个基于 Tuny 实现发布服务和调用服务的例子。

在这个例子中将提供一个 HelloWorld 接口的实现，配置为 Spring Bean，并基于 Tuscany 将其发布为 WebService 方式的 SCA Service，下面介绍这部分是如何实现的。

HelloWorld 的接口定义如下：

```
@Remotable
public interface HelloWorld {
    public String sayHello(String name);
}
```

HelloWorld 接口实现的代码如下：

```
public class DefaultHelloWorld implements HelloWorld {
    public String sayHello(String name) {
        System.out.println("Server receive: "+name);
        return "Server response: Hello "+name;
    }
}
```

Spring Bean xml 的配置如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:sca="http://www.springframework.org/schema/sca"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
```

```

http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/sca
http://www.osoa.org/xmlns/sca/1.0/spring-sca.xsd">

<sca:service name="HelloWorldService"
  type="chapter2.sca.tuscany.demo.HelloWorld"
  target="HelloWorldServiceBean"/>

<bean id="HelloWorldServiceBean"
  class="chapter2.sca.tuscany.demo.impl.DefaultHelloWorld">
</bean>
</beans>

```

将以上文件保存至 resources/spring 目录下，命名为 beans.xml。

基于 Tuscany 将其发布为 WebService 的配置如下：

```

<composite name="HelloWorld"
  targetNamespace="http://tuscany.apache.org/xmlns/sca/1.0"
  xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:nsl="http://www.osoa.org/xmlns/sca/1.0">
  <service name="HelloWorldService" promote="HelloWorldComponent">
    <interface.java interface="chapter2.sca.tuscany.demo.HelloWorld"/>
    <binding.ws uri="http://localhost:8080/services/HelloWorldService"/>
  </service>
  <component name="HelloWorldComponent">
    <implementation.spring location="resources/spring/beans.xml"/>
  </component>
</composite>

```

将以上文件保存至 src 目录下，命名为 publishservice.composite。

完成了上面的步骤后，编写一个启动类来完成服务的发布，代码示例如下：

```

public static void main(String[] args) throws Exception{
    SCADomain.newInstance("publishservice.composite");
    System.out.println("Server Started");
    while(true){
        Thread.sleep(1000000);
    }
}

```

执行此代码后在 console 中可看到 Server Started 的信息，通过浏览器访问 <http://localhost:8080/services>HelloWorldService?wsdl> 即可看到 HelloWorldService 的 WebService 描述信息。

接着来完成调用 HelloWorld Service 的代码，同样在 Spring Bean 中来调用此 Service，首先编写一个需要注入 HelloWorld Service 的测试用的 Spring Bean，代码如下：

```

private HelloWorld service=null;

public void setService(HelloWorld service) {
    this.service=service;
}

public String execute(String name) {
    return service.sayHello(name);
}

```

为了便于在Tuscany启动后在代码中获取Spring的ApplicationContext，从而获取相应的Spring Bean进行测试，编写一个实现 ApplicationContextAware 接口的类，代码如下：

```

public static ApplicationContext context=null;

@Override
public void setApplicationContext(ApplicationContext context)
    throws BeansException {
    SpringApplicationContextHolder.context=context;
}

```

按照Tuscany与Spring集成的实现，Spring配置文件如下：

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:sca="http://www.springframework.org/schema/sca"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/sca
           http://www.osoa.org/xmlns/sca/1.0/spring-sca.xsd">

    <sca:reference name="HelloWorldService"
    type="chapter2.sca.tuscany.demo.HelloWorld"/>

    <bean id="HelloWorldConsumer"
    class="chapter2.sca.tuscany.demo.impl.HelloWorldConsumer">
        <property name="service">
            <ref bean="HelloWorldService"/>
        </property>
    </bean>

    <bean id="SpringBeanFactoryHolder"
    class="chapter2.sca.tuscany.demo.SpringApplicationContextHolder"/>
</beans>

```

绑定 SCA Service 的配置文件编写如下：

```
<composite name="HelloWorld"
    targetNamespace="http://tuscany.apache.org/xmlns/sca/1.0"
    xmlns="http://www.osoa.org/xmlns/sca/1.0"
    xmlns:ns1="http://www.osoa.org/xmlns/sca/1.0">
    <component name="HelloWorldComponent">
        <implementation.spring location="resources/spring/consumebeans.xml"/>
        <reference name="HelloWorldService">
            <binding.ws
uri="http://localhost:8080/services/HelloWorldService"/>
        </reference>
    </component>
</composite>
```

用于测试 SCA Service 调用的类编写如下：

```
public static void main(String[] args) throws Exception{
    SCADomain.newInstance("consumeservice.composite");
    System.out.println("Client Started");
    HelloWorldConsumer consumer=(HelloWorldConsumer)
SpringApplicationContextHolder.context.getBean("HelloWorldConsumer");
    System.out.println(consumer.execute("BlueDavy"));
}
```

在启动了发布 SCA Service 的代码后，再启动调用 SCA Service 的代码，即可在 console 上看到相应的执行信息，基于上述步骤就很容易地实现在 Spring 中以 WebService 的方式发布和调用 SCA Service，SCA。以下介绍基于 Tuny 如何实现 SOA 平台。

首先还是从统一服务的交互这点分别来看看发布服务、调用服务和支持的网络协议及交互方式。

- 发布服务

Tuscany 遵守 SCA 标准而编写，因此其发布服务的方式和上面描述的 SCA 标准中发布服务的方式是一样的，这个从上面的例子中可看出，但它支持将更多种实现方式的系统中的功能发布为 SCA Service，这包括了 Java、xquery、Script、Spring、resource、bpel 和 OSGi。

Tuscany 支持更多种发布方式，包括发布为 WebService、Ajax、corba、erlang、jms、jsonrpc、rmi、ejb、HTTP 及 rss 方式。

- 调用服务

在调用服务的方式，和 SCA 标准中调用服务的方式是一样的。它和发布服务一样，支持更多种应用集成及调用方式，并且包括了更多种语言的支持，例如对于 Ruby 的支持等。

- \* 支持的通信以及交互方式

在上面发布服务中可以看到，Tuscany 支持的通信方式比 SCA 标准多出了很多，例如对 Ajax、jsonrpc、rmi 等都提供了支持。

在交互方式上，Tuscany 可通过设置 CallbackEndpoint 来实现异步调用，这要对代码做一些改动。

在调试/跟踪方面，当服务器端抛出异常时，会将此异常信息带回到调用端，这对于查错而言会带来一定的帮助；在依赖管理方面，Tuscany 并没有在 SCA 的标准上做扩展，因此仍然不是很好实现；在高性能和高可用方面，Tuscany 没有做专门的处理，如须确保性能，可自行基于 Tuscany 扩展实现交互方式。

## 2.4 基于 Mule 实现 SOA 平台

Mule 是常用的 ESB 实现框架之一，首先来看看基于 Mule 2.2.1 如何实现 Tuscany 章节（第 2.3 节）中的例子，先仍然是以 WebService 的方式对外提供 HelloWorldService，和 Tuscany 中不同的地方仅在于配置文件和启动代码两方面，配置文件上去掉 publishservice.composite，改为遵循 Mule 编写如下配置文件：

```

http
<spring:beans>
    <spring:import resource="resources/spring/mulepublisherbeans.xml"/>
</spring:beans>

<model name="HelloWorld">
    <service name="HelloWorldService">
        <inbound>
            <axis:inbound-endpoint address="http://localhost:12345/services">
                <soap:http-to-soap-request-transformer/>
            </axis:inbound-endpoint>
        </inbound>
        <component>
            <spring-object bean="HelloWorldBean"/>
        </component>
    </service>
</model>
</mule>

```

将上面文件保存为 publishservice.xml。

发布服务的启动代码遵循 Mule 改为如下方式：

```

MuleContext muleContext = new
DefaultMuleContextFactory().createMuleContext("publishservice.xml");

```

```
muleContext.start();
System.out.println("Server Started");
```

执行以上代码后，通过 `http://localhost:12345/services/HelloWorldService?wsdl` 可看到 `HelloWorldService` 的 WebService 描述符，表明已发布成功。

继续来看看怎么按照 Mule Service 的方式来调用这个 WebService，Mule 要求所有的 component 被触发执行的条件为 inbound 的信息。而除了上面通过 WebService 的触发方法外，其他就只能通过消息机制去触发了，支持的有 vm 内的，或者 jms 方式等。在这里选择用 vm 内的方式去实现例子，首先对 `HelloWorldConsumer` 的 execute 稍做了修改：

```
public String execute(String name){
    String response=service.sayHello(name);
    System.out.println(response);
    return response;
}
```

由于必须通过消息才能触发，在例子中就直接用 `MuleClient` 来发送消息触发了，因此去掉了 Spring Bean 的配置文件 `ApplicationContextAware` 接口实现类的配置，Mule 方式的 Service 配置文件内容如下：

```
<spring:beans>
    <spring:import resource="resources/spring/muleconsumebeans.xml"/>
</spring:beans>

<model name="HelloWorld">
    <service name="HelloWorldService">
        <inbound>
            <vm:inbound-endpoint path="helloworld.queue"/>
        </inbound>
        <component>
            <spring-object bean="HelloWorldConsumer">
            </spring-object>
            <binding interface="chapter2.esb.mule.demo.HelloWorld" method="sayHello">
                <axis:outbound-endpoint
address="http://localhost:12345/services/HelloWorldService?method=sayHello"
synchronous="true"/>
            </binding>
        </component>
    </service>
</model>
```

根据上面的配置，必须先发送消息至 `vm://helloworld.queue` 才能触发 `HelloWorldConsumer` 的执行，代码如下：

```

MuleContext muleContext = new
DefaultMuleContextFactory().createMuleContext("consumeservice.xml");
muleContext.start();
MuleClient client=new MuleClient();
client.send("vm://helloworld.queue", "BlueDavy", null);

```

执行以上代码即实现了以 Mule Service 的方式调用远端通过 Mule Service 发布为 WebService 的功能。

以下介绍基于，Mule 如何实现 SOA 平台，首先来看基于 Mule 如何实现服务方式的统一交互。

- **发布服务**

在发布服务上，Mule 支持以 WebService、jms 等方式将 Spring 或普通的 Java 对象发布为 Mule Service，配置上较为简单，但和 Tuscan 比还是相对弱了点。

- **调用服务**

在调用服务上，Mule 的用法相对比较麻烦，这也是由于 ESB 强调一切都以消息的方式发送给总线决定的。

- **支持的通信和交互方式**

通信方式上支持 WebService 及 jms 两种。

在交互方式上，Mule 可明确地指定 synchronous 的参数来实现同步通信，在不指定的情况下即为默认的异步通信。

在调试/跟踪上，Mule 未提供特别的支持；在依赖管理上，Mule 本身未提供支持，但 Mule 所在的 MuleSoft 提供了一个开源的服务治理框架：MuleGalaxy<sup>4</sup>。开发人员可将服务元数据信息注册到 MuleGalaxy，MuleGalaxy 可基于这些元数据信息提供服务查询及依赖分析等功能；在高性能和高可用上，Mule 未做专门的处理。

综合上面的介绍，SCA 标准及 SCA 标准实现的框架对于服务的统一交互支持得很好，ESB 及 ESB 框架则更适用于需要解耦方式的服务交互及复杂的多服务交互的场景，但无论是基于 SCA 标准、ESB，还是已有的 SCA 框架和 ESB 框架，在实现一个大型应用的 SOA 平台时都仍然有不少需要自行扩展实现的地方，尤其是在调试/跟踪、依赖管理及高性能、高可用方面，也许这就是 eBay 自行实现 SOA 平台的原因。对于大型应用的服务化，SOA 平台是一方面，如何推广实行也是一个重要因素，eBay 提及的现有应用移植及培训开发人员是在推广实行时需要仔细考虑的。

以上提及的为一个基本的大型应用的 SOA 平台的特征，而对于一个更加完善的 SOA 平台，作者认为还须具备以下几点：

<sup>4</sup> <http://www.mulesoft.org/display/GALAXY/Home>

- 支撑集群环境

对于大型应用而言，通常会借助集群来支撑大的访问量，如何让服务交互和集群环境结合得更好是 SOA 平台中值得考虑的，例如典型的有软件负载均衡、服务接口或方法级的路由策略等。

- 完善的服务治理

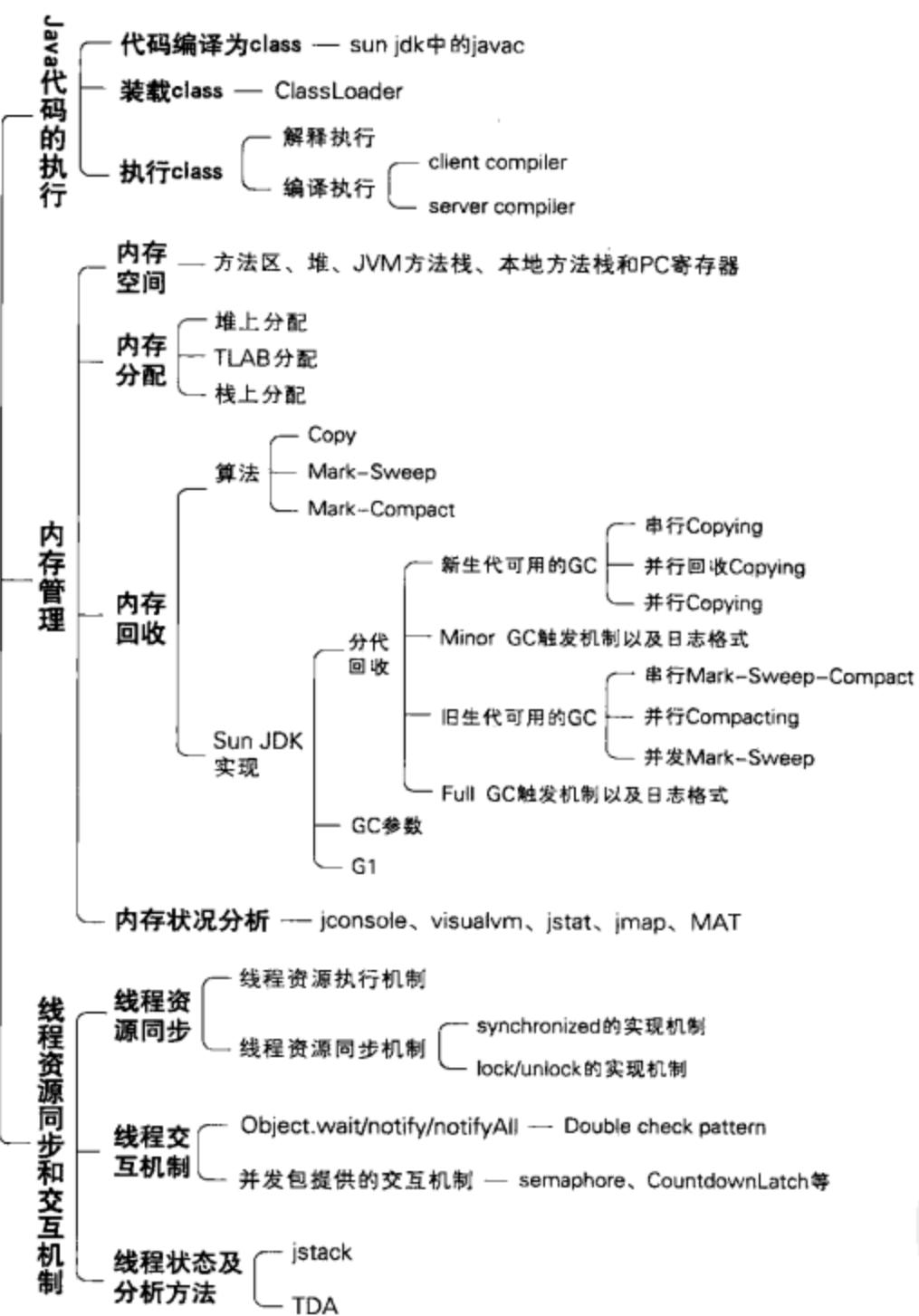
服务治理的主要目的是为了保障服务能够稳定、高性能地运转，之前提及的依赖管理也属于服务治理中的一项，服务运行状况的监测、服务的安全控制、服务的流量限制、服务故障根源的推测及服务可用性的保障也都属于服务治理的范畴。要实现这些功能仅仅靠 SOA 平台还很难做到，通常还要有系统架构的配合。

- 服务 QoS (Quality of Service) 的支持

服务 QoS 的支持是指 SOA 平台按照服务配置的 QoS 来分配相应的硬件资源，例如 A 服务的 QoS 配置为每秒最多支撑 5000 请求，且响应时间 95% 需要在 1 秒以内，SOA 平台要能收集目前服务的运行状况来合理地分配机器资源，要做到这点难度非常高。



# 第3章 深入理解 JVM



Java程序运行在JVM之上，JVM的运行状况对于Java程序而言会产生很大的影响，因此，掌握JVM中的关键机制对于编写稳定、高性能的Java程序至关重要。

JVM规范<sup>1</sup>定义的标准结构如图3.1所示。

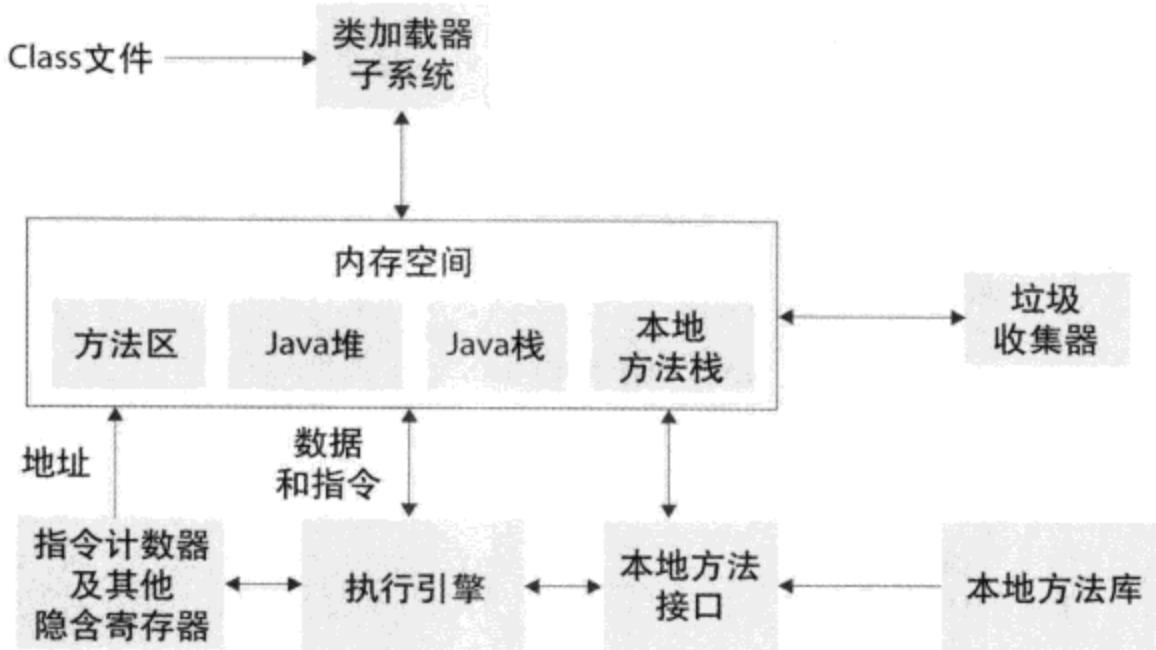


图3.1 JVM标准结构

以上标准结构是JVM规范中定义的，但各家厂商在实现时不一定会完全遵守。

JVM负责装载class文件并执行，因此，首先要掌握的是JDK如何将Java代码编译为class文件、如何装载class文件及如何执行class，将源码编译为class文件的实现取决于各个JVM实现或各种源码编译器。class文件通常由类加载器（ClassLoader）来完成加载；class的执行在Sun JDK中有解释执行和编译为机器码执行两种方式，其中编译为机器码又分为client和server两种模式。Sun JDK为了提升class的执行效率，对于解释执行和编译为机器码执行都设置了很多的优化策略。

Java程序无须显式分配和回收内存，因此JVM如何进行内存的分配和回收也是要关注的问题。

JVM提供了多线程支持，对于分布式Java应用而言，通常要借助线程来实现高并发，因此JVM中线程资源同步的机制及线程之间交互的机制也是需要掌握的。

各厂家在实现JVM时有所区别，本章以Sun JDK 1.6为例来对JVM中的这三个方面进行介绍。

## 3.1 Java代码的执行机制

要在JVM中执行Java代码，首先要编译为class文件。下面介绍Sun JDK是如何将Java代码编译为class文件的，这种机制通常称为Java源码编译机制。

<sup>1</sup> <http://java.sun.com/docs/books/jvms/>

### 3.1.1 Java 源码编译机制

JVM 规范中定义了 class 文件的格式，但并未定义 Java 源码如何编译为 class 文件，各厂商在实现 JDK 时通常会将符合 Java 语言规范的源码编译为 class 文件的编译器，例如在 Sun JDK 中就是 javac，javac 将 Java 源码编译为 class 文件的步骤如图 3.2 所示。



图 3.2 javac 编译源码为 class 文件的步骤

下面简单介绍以上三个步骤：

#### 1. 分析和输入到符号表 (Parse and Enter)

Parse 过程所做的为词法和语法分析。词法分析 (com.sun.tools.javac.parser.Scanner) 要完成的是将代码字符串转变为 token 序列（例如 Token.EQ(name:=)）；语法分析 (com.sun.tools.javac.parser.Parser) 要完成的是根据语法由 token 序列生成抽象语法树<sup>2</sup>。

Enter (com.sun.tools.javac.comp.Enter) 过程为将符号输入到符号表，通常包括确定类的超类型和接口、根据需要添加默认构造器、将类中出现的符号输入类自身的符号表中等。

#### 2. 注解处理 (Annotation Processing)

该步骤主要用于处理用户自定义的 annotation，可能带来的好处是基于 annotation 来生成附加的代码或进行一些特殊的检查，从而节省一些共用的代码的编写，例如当采用 Lombok<sup>3</sup>时，可编写如下代码：

```
public class User{
    private @Getter String username;
}
```

编译时引入 Lombok 对 User.java 进行编译后，再通过 javap 查看 class 文件可看到自动生成了 public String getUsername()方法。

此功能基于 JSR 269<sup>4</sup>，在 Sun JDK 6 中提供了支持，在 Annotation Processing 进行后，再次进入 Parse and Enter 步骤。

#### 3. 语义分析和生成 class 文件 (Analyse and Generate)

Analyse 步骤基于抽象语法树进行一系列的语义分析，包括将语法树中的名字、表达式等元素与变量、方法、类型等联系到一起；检查变量使用前是否已声明；推导泛型方法的类型参数；检查类型匹

<sup>2</sup> [http://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](http://en.wikipedia.org/wiki/Abstract_syntax_tree)

<sup>3</sup> <http://projectlombok.org>

<sup>4</sup> <http://jcp.org/en/jsr/detail?id=269>

配性；进行常量折叠；检查所有语句都可到达；检查所有 checked exception 都被捕获或抛出；检查变量的确定性赋值（例如有返回值的方法必须确定有返回值）；检查变量的确定性不重复赋值（例如声明为 final 的变量等）；解除语法糖（消除 if(false) {...} 形式的无用代码；将泛型 Java 转为普通 Java；将含有语法糖的语法树改为含有简单语言结构的语法树，例如 foreach 循环、自动装箱/拆箱等）等。

在完成了语义分析后，开始生成 class 文件（com.sun.tools.javac.jvm.Gen），生成的步骤为：首先将实例成员初始化器收集到构造器中，将静态成员初始化器收集为<clinit>()；接着将抽象语法树生成字节码，采用的方法为后序遍历语法树，并进行最后的少量代码转换（例如 String 相加转变为 StringBuilder 操作）；最后从符号表生成 class 文件。

上面简单介绍了基于 javac 如何将 java 源码编译为 class 文件<sup>5</sup>，除 javac 外，还可通过 ECJ（Eclipse Compiler for Java）<sup>6</sup>或 Jikes<sup>7</sup>等编译器来将 Java 源码编译为 class 文件。

class 文件中并不仅仅存放了字节码，还存放了很多辅助 jvm 来执行 class 的附加信息，一个 class 文件包含了以下信息。

- 结构信息

包括 class 文件格式版本号及各部分的数量与大小的信息。

- 元数据

简单来说，可以认为元数据对应的就是 Java 源码中“声明”与“常量”的信息，主要有：类/继承的超类/实现的接口的声明信息、域（Field）与方法声明信息和常量池。

- 方法信息

简单来说，可以认为方法信息对应的就是 Java 源码中“语句”与“表达式”对应的信息，主要有：字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试用符号信息。

以一段简单的代码来说明 class 文件格式。

```
public class Foo{
    private static final int MAX_COUNT=1000;
    private static int count=0;
    public int bar() throws Exception{
        if(++count >= MAX_COUNT){
            count=0;
            throw new Exception("count overflow");
        }
        return count;
    }
}
```

<sup>5</sup> 感兴趣的读者可进一步查看 com.sun.tools.javac.main.JavaCompiler 源码。

<sup>6</sup> Eclipse JDT 所使用的编译器。

<sup>7</sup> <http://jikes.sourceforge.net>

```

    }
}

```

执行 `javac -g Foo.java`（加上`-g`是为了生成所有的调试信息，包括局部变量名及行号信息，在不加`-g`的情况下默认只生成行号信息）编译此源码，之后通过 `javap -c -s -l -verbose Foo` 来查看编译后的 class 文件，结合 class 文件格式来看其中的关键内容。

```

// 类/继承的超类/实现的接口的声明信息
public class Foo extends java.lang.Object
  SourceFile: "Foo.java"
  // class 文件格式版本号, major version: 50 表示为 jdk 6, 49 为 jdk 5, 48 为 jdk 1.4, 只
  // 有高版本能执行低版本的 class 文件, 这也是 jdk 5 不能执行 jdk 6 编译的代码的原因。
  minor version: 0
  major version: 50
  // 常量池, 存放了所有的 Field 名称、方法名、方法签名、类型名、代码及 class 文件中的常量值。
  Constant pool:
const #1 = Method #7.#27; // java/lang/Object."<init>":()V
const #2 = Field #6.#28; // Foo.count:I
const #3 = class #29; // java/lang/Exception
const #4 = String #30; // count overflow
const #5 = Method #3.#31; //
java/lang/Exception."<init>":(Ljava/lang/String;)V
...
const #34 = Asciz (Ljava/lang/String;)V;
{
  // 将符号输入到符号表时生成的默认构造器方法
  public Foo();
  ...
  // bar 方法的元数据信息
  public int bar() throws java.lang.Exception;
    Signature: ()I
    // 对应字节码的源码行号信息, 可在编译的时候通过-g:none 去掉行号信息, 行号信息对于查找问题而
    // 言至关重要, 因此最好还是保留。
  LineNumberTable:
    line 9: 0
    line 10: 15
    line 11: 19
    line 13: 29
    // 局部变量信息, 如生成的 class 文件中无局部变量信息, 则无法知道局部变量的名称, 并且局部变量
    // 信息是和方法绑定的, 接口是没有方法体的, 所以 ASM 之类的在获取接口方法时, 是拿不到方法中参数的信
    // 息的。
  LocalVariableTable:
    Start  Length  Slot  Name   Signature
      0       33      0   this     LFoo;

```

```

Code:
Stack=3, Locals=1, Args_size=1
// 方法对应的字节码
0:    getstatic    #2; //Field count:I
..
29: getstatic    #2; //Field count:I
32: ireturn
...
// 记录有分支的情况(对应代码中 if...、for、while 等), 在下一节“类加载机制”中会讲解这个的作用
StackMapTable: number_of_entries = 1
  frame_type = 29 /* same */
// 异常处理器表
Exceptions:
  throws java.lang.Exception
..
}

```

从上可见, class 文件是个完整的自描述文件, 字节码在其中只占了很小的部分, 源码编译为 class 文件后, 即可放入 jvm 中执行。执行时 jvm 首先要做的是装载 class 文件, 这个机制通常称为类加载机制。

### 3.1.2 类加载机制

类加载机制是指.class 文件加载到 JVM, 并形成 Class 对象的机制, 之后应用就可对 Class 对象进行实例化并调用, 类加载机制可在运行时动态加载外部的类、远程网络下载过来的 class 文件等。除了该动态化的优点外, 还可通过 JVM 的类加载机制来达到类隔离的效果, 例如 Application Server 中通常要避免两个应用的类互相干扰。

JVM 将类加载过程划分为三个步骤: 装载、链接和初始化。装载和链接过程完成后, 即将二进制的字节码转换为 Class 对象; 初始化过程不是加载类时必须触发的, 但最迟必须在初次主动使用对象前执行, 其所作的动作是给静态变量赋值、调用<clinit>()等。

整个过程如图 3.3 所示。

#### 1. 装载 (Load)

装载过程负责找到二进制字节码并加载至 JVM 中, JVM 通过类的全限定名 (com.bluedavy.HelloWorld) 及类加载器 (ClassLoaderA 实例) 完成类的加载, 同样, 也采用以上两个元素来标识一个被加载了的类: 类的全限定名+ClassLoader 实例 ID。类名的命名方式如下:

对于接口或非数组型的类, 其名称即为类名, 此种类型的类由所在的 ClassLoader 负责加载;

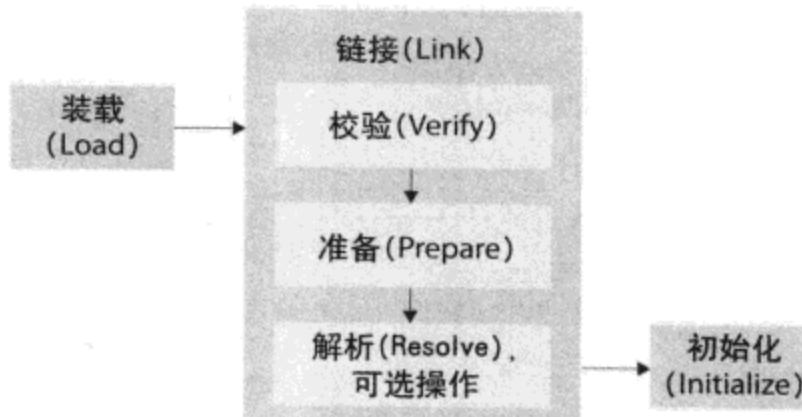


图 3.3 类加载过程

对于数组型的类，其名称为“[”+（基本类型或 L+引用类型类名；），例如 `byte[] bytes=new byte[512]`，该 `bytes` 的类名为：[B；`Object[] objects=new Object[10]`，`objects` 的类名则为：[Ljava.lang.Object；，数组型类中的元素类型由所在的 ClassLoader 负责加载，但数组类则由 JVM 直接创建。

## 2. 链接（Link）

链接过程负责对二进制字节码的格式进行校验、初始化装载类中的静态变量及解析类中调用的接口、类。

二进制字节码的格式校验遵循 Java Class File Format（具体请参见 JVM 规范）规范，如果格式不符合，则抛出 `VerifyError`；校验过程中如果碰到要引用到其他的接口和类，也会进行加载；如果加载过程失败，则会抛出 `NoClassDefFoundError`。

在完成了校验后，JVM 初始化类中的静态变量，并将其值赋为默认值。

最后对类中的所有属性、方法进行验证，以确保其要调用的属性、方法存在，以及具备相应的权限（例如 `public`、`private` 域权限等）。如果这个阶段失败，可能会造成 `NoSuchMethodError`、`NoSuchFieldError` 等错误信息。

## 3. 初始化（Initialize）

初始化过程即执行类中的静态初始化代码、构造器代码及静态属性的初始化，在以下四种情况下初始化过程会被触发执行：

- 1) 调用了 `new`；
- 2) 反射调用了类中的方法；
- 3) 子类调用了初始化；
- 4) JVM 启动过程中指定的初始化类。

在执行初始化过程之前，首先必须完成链接过程中的校验和准备阶段，解析阶段则不强制。

JVM 的类加载通过 ClassLoader 及其子类来完成，分为 Bootstrap ClassLoader、Extension ClassLoader、System ClassLoader 及 User-Defined ClassLoader。这 4 种 ClassLoader 的关系如图 3.4 所示。

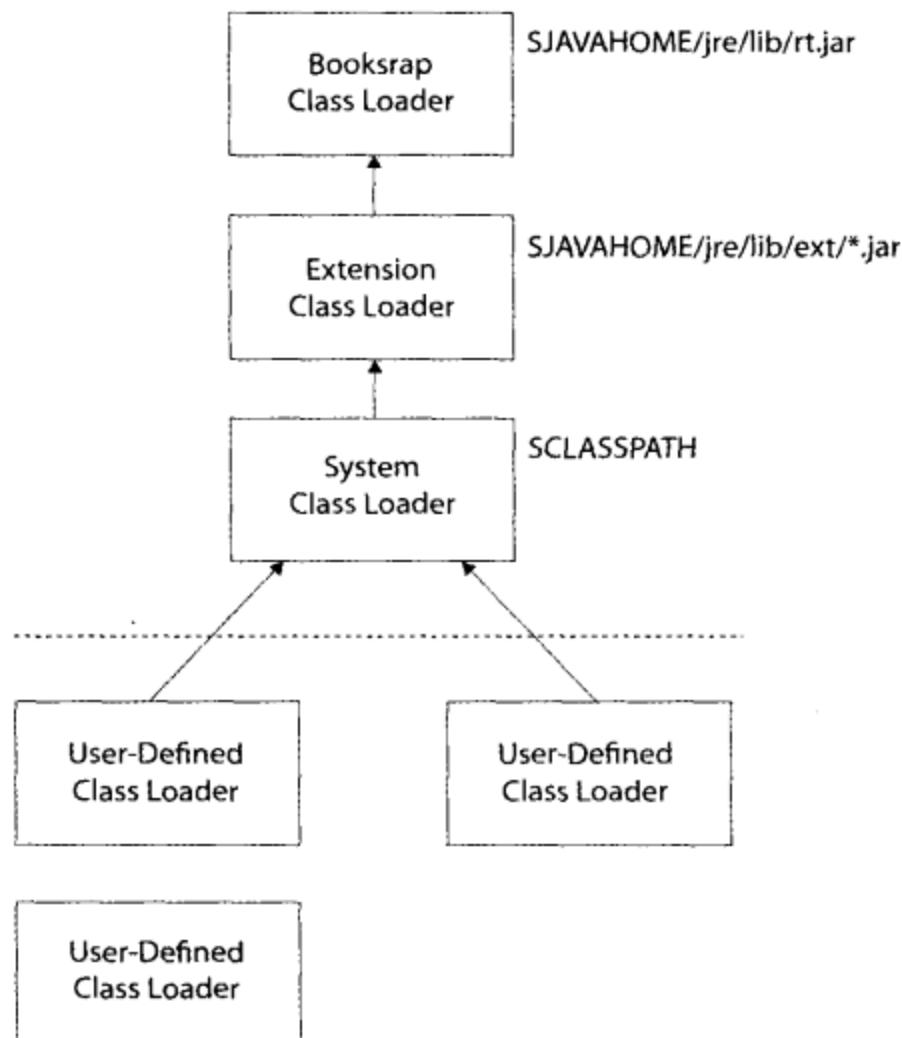


图 3.4 Sun JDK ClassLoader 继承关系

### 1. Bootstrap ClassLoader

Sun JDK 采用 C++ 实现了此类，此类并非 ClassLoader 的子类，在代码中没有办法拿到这个对象，Sun JDK 启动时会初始化此 ClassLoader，并由 ClassLoader 完成 \$JAVA\_HOME 中 jre/lib/rt.jar 里所有 class 文件的加载，jar 中包含了 Java 规范定义的所有接口及实现。

### 2. Extension ClassLoader

JVM 用此 ClassLoader 来加载扩展功能的一些 jar 包，例如 Sun JDK 中目录下有 dns 工具 jar 包等，在 Sun JDK 中 ClassLoader 对应的类名为 ExtClassLoader。

### 3. System ClassLoader

JVM 用此 ClassLoader 来加载启动参数中指定的 Classpath 中的 jar 包及目录，在 Sun JDK 中 ClassLoader 对应的类名为 AppClassLoader。

例如一段这样的代码：

```

public class ClassLoaderDemo {

    public static void main(String[] args) throws Exception{
        System.out.println(ClassLoaderDemo.class.getClassLoader());
    }
}
  
```

```

        System.out.println(ClassLoaderDemo.class.getClassLoader().getParent());
        System.out.println(ClassLoaderDemo.class.getClassLoader().getParent().getPar
ent());
    }
}

```

执行后显示的信息类似如下：

```

(sun.misc.Launcher$AppClassLoader)
(sun.misc.Launcher$ExtClassLoader)
null

```

按照上面的描述，就可看到典型的 System ClassLoader、Extension ClassLoader，而由于 Bootstrap ClassLoader 并不是 Java 中的 ClassLoader，因此 Extension ClassLoader 的 parent 为 null。

#### 4. User-Defined ClassLoader

User-Defined ClassLoader 是 Java 开发人员继承 ClassLoader 抽象类自行实现的 ClassLoader，基于自定义的 ClassLoader 可用于加载非 Classpath 中（例如从网络上下载的 jar 或二进制）的 jar 及目录、还可以在加载之前对 class 文件做一些动作，例如解密等。

JVM 的 ClassLoader 采用的是树形结构，除 BootstrapClassLoader 外，其他的 ClassLoader 都会有 parent ClassLoader，User-Defined ClassLoader 默认的 parent ClassLoader 为 System ClassLoader。加载类时通常按照树形结构的原则来进行，也就是说，首先应从 parent ClassLoader 中尝试进行加载，当 parent 中无法加载时，应再尝试从 System ClassLoader 中进行加载，System ClassLoader 同样遵循此原则，在找不到的情况下会自动从其 parent ClassLoader 中进行加载。值得注意的是，由于 JVM 是采用类名加 Classloader 的实例来作为 Class 加载的判断的，因此加载时不采用上面的顺序也是可以的，例如加载时不去 parent ClassLoader 中寻找，而只在当前的 ClassLoader 中寻找，会造成树上多个不同的 ClassLoader 中都加载了某 Class，并且这些 Class 的实例对象都不相同，JVM 会保证同一个 ClassLoader 实例对象中只能加载一次同样名称的 Class，因此可借助此来实现类隔离的需求，但有时也会带来困惑，例如 ClassCastException。因此在加载类的顺序上要根据需求合理把握，尽量保证从根到最下层的 ClassLoader 上的 Class 只加载了一次。

ClassLoader 抽象类提供了几个关键的方法：

- `loadClass`

此方法负责加载指定名字的类，ClassLoader 的实现方法为先从已经加载的类中寻找，如没有，则继续从 parent ClassLoader 中寻找；如果仍然没找到，则从 System ClassLoader 中寻找，最后再调用 `findClass` 方法来寻找；如果要改变类的加载顺序，则可覆盖此方法；如果加载顺序相同，则可通过覆盖 `findClass` 来做特殊的处理，例如解密、固定路径寻找等。当通过整个寻找类的过程仍然未获取 Class

对象时，则抛出 `ClassNotFoundException`。

如果类需要 `resolve`，则调用 `resolveClass` 进行链接。

- `findLoadedClass`

此方法负责从当前 `ClassLoader` 实例对象的缓存中寻找已加载的类，调用的为 `native` 的方法。

- `findClass`

此方法直接抛出 `ClassNotFoundException`，因此要通过覆盖 `loadClass` 或此方法来以自定义的方式加载相应的类。

- `findSystemClass`

此方法负责从 `System ClassLoader` 中寻找类，如未找到，则继续从 `Bootstrap ClassLoader` 中寻找，如果仍然未找到，则返回 `null`。

- `defineClass`

此方法负责将二进制的字节码转换为 `Class` 对象，这个方法对于自定义加载类而言非常重要。如果二进制的字节码的格式不符合 JVM Class 文件的格式，则抛出 `ClassFormatError`；如果生成的类名和二进制字节码中的不同，则抛出 `NoClassDefFoundError`；如果加载的 `class` 是受保护的、采用不同签名的，或者类名是以 `java.` 开头的，则抛出 `SecurityException`；如果加载的 `class` 在此 `ClassLoader` 中已加载，则抛出 `LinkageError`。

- `resolveClass`

此方法负责完成 `Class` 对象的链接，如果链接过，则会直接返回。

当 Java 开发人员调用 `Class.forName` 来获取一个对应名称的 `Class` 对象时，JVM 会从方法栈上寻找第一个 `ClassLoader`，通常也就是执行 `Class.forName` 所在类的 `ClassLoader`，并使用此 `ClassLoader` 来加载此名称的类。JVM 为了保护加载、执行的类的安全，它不允许 `ClassLoader` 直接卸载加载了的类，只有 JVM 才能卸载，在 Sun JDK 中，只有当 `ClassLoader` 对象没有引用时，此 `ClassLoader` 对象加载的类才会被卸载。

根据上面的描述，在实际的应用中，JVM 类加载过程会抛出这样那样的异常，这些情况下掌握各种异常产生的原因是最重要的，下面来看类加载方面的常见异常。

### 1. `ClassNotFoundException`

这是最常见的异常，产生这个异常的原因因为在当前的 `ClassLoader` 中加载类时未找到类文件，对位于 `System ClassLoader` 的类很容易判断，只要加载的类不在 `Classpath` 中，而对位于 `User-Defined ClassLoader` 的类则麻烦些，要具体查看这个 `ClassLoader` 加载类的过程，才能判断此 `ClassLoader` 要从什么位置加载到此类。

例如直接在代码中执行 `Class.forName("com.bluedavy.A")`，而当前类的 `classloader` 下根本就没有

该类所在的 jar 或没有该 class 文件，就会抛出 ClassNotFoundException。

## 2. NoClassDefFoundError

该异常较之 ClassNotFoundException 更难处理一些，造成此异常的主要原因是加载的类中引用到的另外的类不存在，例如要加载 A，而 A 中调用了 B，B 不存在或当前 ClassLoader 没法加载 B，就会抛出这个异常。

例如有一段这样的代码：

```
public class A{
    private B b=new B();
}
```

当采用 Class.forName 加载 A 时，虽能找到 A.class，但此时 B.class 不存在，则会抛出 NoClassDefFoundError。

因此，对于这个异常，须先查看是加载哪个类时报出的，然后再确认该类中引用的类是否存在于当前 ClassLoader 能加载到的位置。

## 3. LinkageError

该异常在自定义 ClassLoader 的情况下更容易出现，主要原因是此类已经在 ClassLoader 加载过了，重复地加载会造成该异常，因此要注意避免在并发的情况下出现这样的问题。

由于 JVM 的这个保护机制，使得在 JVM 中没办法直接更新一个已经 load 的 Class，只能创建一个新的 ClassLoader 来加载更新的 Class，然后将新的请求转入该 ClassLoader 中来获取类，这也是 JVM 中不好实现动态更新的原因之一，而其他更多的原因是对象状态的复制、依赖的设置等。

## 4. ClassCastException

该异常有多种原因，在 JDK 5 支持泛型后，合理使用泛型可相对减少此异常的触发。这些原因中比较难查的是两个 A 对象由不同的 ClassLoader 加载的情况，这时如果将其中某个 A 对象造型成另外一个 A 对象，也会报出 ClassCastException。

### 3.1.3 类执行机制

在完成将 class 文件信息加载到 JVM 并产生 Class 对象后，就可执行 Class 对象的静态方法或实例化对象进行调用了。在源码编译阶段将源码编译为 JVM 字节码，JVM 字节码是一种中间代码的方式，要由 JVM 在运行期对其进行解释并执行，这种方式称为字节码解释执行方式。

#### 字节码解释执行

由于采用的为中间码的方式，JVM 有一套自己的指令，对于面向对象的语言而言，最重要的是执

行方法的指令，JVM采用了`invokestatic`、`invokevirtual`、`invokeinterface`和`invokespecial`四个指令来执行不同的方法调用。`invokestatic`对应的是调用`static`方法，`invokevirtual`对应的是调用对象实例的方法，`invokeinterface`对应的是调用接口的方法，`invokespecial`对应的是调用`private`方法和编译源码后生成的`<init>`方法，此方法为对象实例化时的初始化方法，例如下面一段代码：

```
public class Demo{
    public void execute(){
        A.execute();
        A a=new A();
        a.bar();
        IFoo b=new B();
        b.bar();
    }
}
class A{
    public static int execute(){
        return 1+2;
    }
    public int bar(){
        return 1+2;
    }
}
class B implements IFoo{
    public int bar(){
        return 1+2;
    }
}
public interface IFoo{
    public int bar();
}
```

通过javac编译上面的代码后，使用javap -c Demo查看其execute方法的字节码：

```
public void execute();
Code:
0: invokestatic #2; //Method A.execute:()I
3: pop
4: new #3; //class A
7: dup
8: invokespecial #4; //Method A."<init>":()V
11: astore_1
12: aload_1
13: invokevirtual #5; //Method A.bar:()I
16: pop
17: new #6; //class B
```

```

20: dup
21: invokespecial #7; //Method B."<init>":()V
24: astore_2
25: aload_2
26: invokeinterface #8, 1; //InterfaceMethod IFoo.bar:()I
31: pop
32: return

```

从以上例子可看到 `invokestatic`、`invokespecial`、`invokevirtual` 及 `invokeinterface` 四种指令对应调用方法的情况。

Sun JDK 基于栈的体系结构来执行字节码，基于栈方式的好处为代码紧凑，体积小。

线程在创建后，都会产生程序计数器（PC）（或称为 PC registers）和栈（Stack）；PC 存放了下一条要执行的指令在方法内的偏移量；栈中存放了栈帧（StackFrame），每个方法每次调用都会产生栈帧。栈帧主要分为局部变量区和操作数栈两部分，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果，栈帧中还会有一些杂用空间，例如指向方法已解析的常量池的引用、其他一些 VM 内部实现需要的数据等，结构如图 3.5 所示。

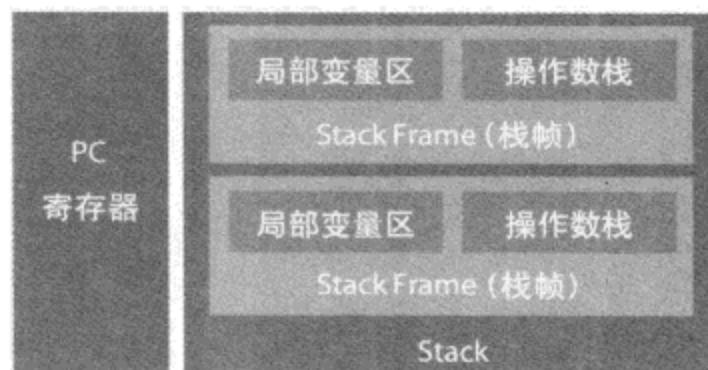


图 3.5 Sun JDK 基于栈的体系结构

下面来看一个方法执行时过程的例子，代码如下：

```

public class Demo(){
public static void foo(){
    int a=1;
    int b=2;
    int c=(a+b) * 5;
}
}

```

编译以上代码后，`foo` 方法对应的字节码为以及相应的解释如下：

```

public static void foo();
Code:
0:  iconst_1  //将类型为 int、值为 1 的常量放入操作数栈;
1:  istore_0  //将操作数栈中栈顶的值弹出放入局部变量区;

```

```

2:  iconst_2 //将类型为 int、值为 2 的常量放入操作数栈;
3:  istore_1 //将操作数栈中栈顶的值弹出放入局部变量区;
4:  iload_0 //装载局部变量区中的第一个值到操作数栈;
5:  iload_1 //装载局部变量区中的第二个值到操作数栈;
6:  iadd //执行 int 类型的 add 指令，并将计算出的结果放入操作数栈;
7:  iconst_5 //将类型为 int、值为 5 的常量放入操作数栈;
8:  imul //执行 int 类型的 mul 指令，并将计算出的结果放入操作数栈;
9:  istore_2 //将操作数栈中栈顶的值弹出放入局部变量区;
10: return // 返回

```

### 1. 指令解释执行

对于方法的指令解释执行，执行方式为经典冯·诺依曼体系中的 FDX 循环方式，即获取下一条指令，解码并分派，然后执行。在实现 FDX 循环时有 switch-threading、token-threading、direct-threading、subroutine-threading、inline-threading 等多种方式<sup>8</sup>。

switch-threading 是一种最简方式的实现，代码大致如下：

```

while(true) {
    int code=fetchNextCode();
    switch(code) {
    case IADD:
        // do add
    case ...:
        // do sth.
    }
}

```

以上方式很简单地实现了 FDX 的循环方式，但这种方式的问题是每次执行完都得重新回到循环开始点，然后重新获取下一条指令，并继续 switch，这导致了大部分时间都花费在了跳转和获取下一条指令上，而真正业务逻辑的代码非常短。

token-threading 在上面的基础上稍做了改进，改进后的代码大致如下：

```

IADD:{
    // do add;
    fetchNextCode();
    dispatch();
}
ICONST_0:{
    push(0);
    fetchNextCode();
}

```

<sup>8</sup> <http://www.complang.tuwien.ac.at/forth/threaded-code.html>

```

    dispatch();
}

```

这种方式较之 switch-threading 方式而言，冗余了 fetch 和 dispatch，消耗的内存量会大一些，但由于去除了 switch，因此性能会稍好一些。

其他 direct-threading、inline-threading 等做了更多的优化，Sun JDK 的重点为编译成机器码，并没有在解释器上做太复杂的处理，因此采用了 token-threading 方式。为了让解释执行能更加高效，Sun JDK 还做了一些其他的优化，主要有：栈顶缓存（top-of-stack caching）和部分栈帧共享。

## 2. 栈顶缓存

在方法的执行过程中，可看到有很多操作要将值放入操作数栈，这导致了寄存器和内存要不断地交换数据，Sun JDK 采用了一个栈顶缓存，即将本来位于操作数栈顶的值直接缓存在寄存器上，这对于大部分只需要一个值的操作而言，无须将数据放入操作数栈，可直接在寄存器计算，然后放回操作数栈。

## 3. 部分栈帧共享

当一个方法调用另外一个方法时，通常传入另一方法的参数为已存放在操作数栈的数据。Sun JDK 在此做了一个优化，就是当调用方法时，后一方法可将前一方法的操作数栈作为当前方法的局部变量，从而节省数据 copy 带来的消耗。

另外，在解释执行时，对于一些特殊的情况会直接执行机器指令，例如 Math.sin、Unsafe.compareAndSwapInt 等。

## 编译执行

解释执行的效率较低，为提升代码的执行性能，Sun JDK 提供将字节码编译为机器码的支持，编译在运行时进行，通常称为 JIT 编译器。Sun JDK 在执行过程中对执行频率高的代码进行编译，对执行不频繁的代码则继续采用解释的方式，因此 Sun JDK 又称为 Hotspot VM，在编译上 Sun JDK 提供了两种模式：client compiler (-client) 和 server compiler (-server)。

client compiler 又称为 C1<sup>9</sup>，较为轻量级，只做少量性能开销比高的优化，它占用内存较少，适合于桌面交互式应用。在寄存器分配策略上，JDK 6 以后采用的为线性扫描寄存器分配算法<sup>10</sup>，在其他方面的优化主要有：方法内联、去虚拟化、冗余削除等。

<sup>9</sup> <http://www.bluedavy.com/book/reference/c1.pdf>

<sup>10</sup> <http://www.bluedavy.com/book/reference/LSRA.pdf>

## 1. 方法内联

对于Java类面向对象的语言，通常要调用多个方法来完成功能。执行时，要经历多次参数传递、返回值传递及跳转等，于是C1采取了方法内联的方式，即把调用到的方法的指令直接植入当前方法中。

例如一段这样的代码：

```
public void bar() {
    ...
    bar2();
    ...
}
private void bar2() {
    // bar2
}
```

当编译时，如bar2代码编译后的字节数小于等于35字节<sup>11</sup>，那么，会演变成类似这样的结构<sup>12</sup>：

```
public void bar() {
    ...
    // bar2
    ...
}
```

可在debug版本的JDK的启动参数加上-XX:+PrintInlining来查看方法内联的信息。

方法内联除带来以上好处外，还能够辅助进行以下冗余削除等优化。

## 2. 去虚拟化

去虚拟化是指在装载class文件后，进行类层次的分析，如发现类中的方法只提供一个实现类，那么对于调用了此方法的代码，也可进行方法内联，从而提升执行的性能。

例如一段这样的代码：

```
public interface IFoo{
    public void bar();
}
public class Foo implements IFoo{
    public void bar(){
        // Foo bar method
    }
}
public class Demo{
```

<sup>11</sup> 这个值可通过在启动参数中增加-XX:MaxInlineSize=35来进行控制。

<sup>12</sup> 如须查看编译后的代码，可参考这篇文章：<http://wikis.sun.com/display/HotSpotInternals/PrintAssembly>。

```
public void execute(IFoo foo) {
    foo.bar();
}
```

当整个 JVM 中只有 Foo 实现了 IFoo 接口, Demo execute 方法被编译时, 就演变成类似这样的结构:

```
public void execute(){
    // Foo bar method
}
```

### 3. 兀余削除

冗余削除是指在编译时, 根据运行时状况进行代码折叠或削除。

例如一段这样的代码:

```
private static final Log log=LogFactory.getLog("BLUEDAVY");
private static final boolean isDebugEnabled=log.isDebugEnabled();
public void execute(){
    if(isDebugEnabled){
        log.debug("enter this method: execute");
    }
    // do something
}
```

如 log.isDebugEnabled 返回的为 false, 在执行 C1 编译后, 这段代码就演变成类似下面的结构:

```
public void execute(){
    // do something
}
```

这是为什么会在有些代码编写规则上写不要直接调用 log.debug, 而要先判断的原因。

Server compiler 又称为 C2<sup>13</sup>, 较为重量级, C2 采用了大量的传统编译优化技巧来进行优化, 占用内存相对会多一些, 适合于服务器端的应用。和 C1 不同的主要原因是寄存器分配策略及优化的范围, 寄存器分配策略上 C2 采用的为传统的图着色寄存器分配算法<sup>14</sup>; 由于 C2 会收集程序的运行信息, 因此其优化的范围更多在于全局的优化, 而不仅仅是一个方法块的优化。收集的信息主要有: 分支的跳转/不跳转的频率、某条指令上出现过的类型、是否出现过空值、是否出现过异常。

<sup>13</sup> <http://www.bluedavy.com/book/reference/c2.pdf>

<sup>14</sup> <http://www.bluedavy.com/book/reference/GCRA.pdf>

逃逸分析<sup>15</sup>是C2进行很多优化的基础，逃逸分析是指根据运行状况来判断方法中的变量是否会被外部读取。如不会则认为此变量是逃逸的，基于逃逸分析C2在编译时会做标量替换、栈上分配和同步削除等优化。

### 1. 标量替换

标量替换的意思简单来说就是用标量替换聚合量。

例如有这么一段代码：

```
Point point=new Point(1,2);
System.out.println("point.x="+point.x+"; point.y="+point.y);
```

当point对象在后面的执行过程中未用到时，经过编译后，代码会变成类似下面的结构：

```
int x=1;
int y=2;
System.out.println("point.x="+x+"; point.y="+y);
```

之后基于此可以继续做冗余削除。

这种方式能带来的好处是，如果创建的对象并未用到其中的全部变量，则可以节省一定的内存。而对于代码执行而言，由于无须去找对象的引用，也会更快一些。

### 2. 栈上分配

在上面的例子中，如果p没有逃逸，那么C2会选择在栈上直接创建Point对象实例，而不是在JVM堆上。在栈上分配的好处一方面是更加快速，另一方面是回收时随着方法的结束，对象也被回收了，这也是栈上分配的概念。

### 3. 同步削除

同步削除是指如果发现同步的对象未逃逸，那也没有同步的必要了，在C2编译时会直接去掉同步。

例如有这么一段代码：

```
Point point=new Point(1,2);
synchronized(point){
    // do something
}
```

经过分析如果发现point未逃逸，在编译后，代码就会变成下面的结构：

<sup>15</sup> 关于逃逸分析可参见：[http://en.wikipedia.org/wiki/Escape\\_analysis](http://en.wikipedia.org/wiki/Escape_analysis) sun jdk 在 6.0 的逃逸分析实现上有些影响性能，因此在 update 18 里临时禁用了，在 Java 7 中则默认打开。

```
Point point=new Point(1,2);
// do something
```

除了基于逃逸分析的这些外，C2 还会基于其拥有的运行信息来做其他的优化，例如编译分支频率执行高的代码等。

运行后 C1、C2 编译出来的机器码如果不再符合优化条件，则会进行逆优化（deoptimization），也就是回到解释执行的方式，例如基于类层次分析编译的代码，当有新的相应的接口实现类加入时，就执行逆优化。

除了 C1、C2 外，还有一种较为特殊的编译为：OSR（On Stack Replace）<sup>16</sup>。OSR 编译和 C1、C2 最主要的不同点在于 OSR 编译只替换循环代码体的入口，而 C1、C2 替换的是方法调用的入口，因此在 OSR 编译后会出现的现象是方法的整段代码被编译了，但只有在循环代码体部分才执行编译后的机器码，其他部分则仍然是解释执行方式。

默认情况下，Sun JDK 根据机器配置来选择 client 或 server 模式，当机器配置 CPU 超过 2 核且内存超过 2GB 即默认为 server 模式，但在 32 位 Windows 机器上始终选择的都是 client 模式时，也可在启动时通过增加-client 或-server 来强制指定，在 JDK 7 中可能会引入多层次编译的支持。多层次编译是指在-server 的模式下采用如下方式进行编译：

- 解释器不再收集运行状况信息，只用于启动并触发 C1 编译；
- C1 编译后生成带收集运行信息的代码；
- C2 编译，基于 C1 编译后代码收集的运行信息来进行激进优化，当激进优化的假设不成立时，再退回使用 C1 编译的代码。

从以上介绍来看，Sun JDK 为提升程序执行的性能，在 C1 和 C2 上做了很多的努力，其他各种实现的 JVM 也在编译执行上做了很多的优化，例如在 IBM J9、Oracle JRockit 中做了内联、逃逸分析等<sup>17</sup>。Sun JDK 之所以未选择在启动时即编译成机器码，有几方面的原因：

- 1) 静态编译并不能根据程序的运行状况来优化执行的代码，C2 这种方式是根据运行状况来进行动态编译的，例如分支判断、逃逸分析等，这些措施会对提升程序执行的性能会起到很大的帮助，在静态编译的情况下是无法实现的。给 C2 收集运行数据越长的时间，编译出来的代码会越优；
- 2) 解释执行比编译执行更节省内存；
- 3) 启动时解释执行的启动速度比编译再启动更快。

但程序在未编译期间解释执行方式会比较慢，因此需要取一个权衡值，在 Sun JDK 中主要依据方法上的两个计数器是否超过阈值，其中一个计数器为调用计数器，即方法被调用的次数；另一个计数

<sup>16</sup> <http://portal.acm.org/citation.cfm?id=776288>

<sup>17</sup> [http://blogs.oracle.com/ohrstrom/2009/05/pulling\\_a\\_machine\\_code\\_rabbit.html](http://blogs.oracle.com/ohrstrom/2009/05/pulling_a_machine_code_rabbit.html)

器为回边计数器，即方法中循环执行部分代码的执行次数。下面将介绍两个计数器对应的阈值。

- **CompileThreshold**

该值是指当方法被调用多少次后，就编译为机器码。在 client 模式下默认为 1 500 次，在 server 模式下默认为 10 000 次，可通过在启动时添加-XX:CompileThreshold=10 000 来设置该值。

- **OnStackReplacePercentage**

该值为用于计算是否触发 OSR 编译的阈值，默认情况下 client 模式时为 933，server 模式下为 140，该值可通过在启动时添加-XX: OnStackReplacePercentage=140 来设置，在 client 模式时，计算规则为  $\text{CompileThreshold} * (\text{OnStackReplacePercentage}/100)$ ，在 server 模式时，计算规则为  $(\text{CompileThreshold} * (\text{OnStackReplacePercentage} - \text{InterpreterProfilePercentage}))/100$ 。`InterpreterProfilePercentage` 的默认值为 33，当方法上的回边计数器到达这个值时，即触发后台的 OSR 编译，并将方法上累积的调用计数器设置为 `CompileThreshold` 的值，同时将回边计数器设置为 `CompileThreshold/2` 的值，一方面是为了避免 OSR 编译频繁触发；另一方面是以便当方法被再次调用时即触发正常的编译，当累积的回边计数器的值再次达到该值时，先检查 OSR 编译是否完成。如果 OSR 编译完成，则在执行循环体的代码时进入编译后的代码；如果 OSR 编译未完成，则继续把当前回边计数器的累积值再减掉一些，从这些描述可看出，默认情况下对于回边的情况，server 模式下只要回边次数达到 10 700 次，就会触发 OSR 编译。

用以下一段示例代码来模拟编译的触发。

```
public class Foo{
    public static void main(String[] args) {
        Foo foo=new Foo();
        for(int i=0;i<10;i++) {
            foo.bar();
        }
    }
    public void bar(){
        // some bar code
        for(int i=0;i<10700;i++) {
            bar2();
        }
    }
    private void bar2(){
        // bar2 method
    }
}
```

以上代码采用 `java -server` 方式执行，当 `main` 中第一次调用 `foo.bar` 时，`bar` 方法上的调用计数器为 1，回边计数器为 0；当 `bar` 方法中的循环执行完毕时，`bar` 方法的调用计数器仍然为 1，回边计数器则为 10 700，达到触发 OSR 编译的条件，于是触发 OSR 编译，并将 `bar` 方法的调用计数器设置为 10 000，回边计数器设置为 5 000。

当 main 中第二次调用 foo.bar 时，jdk 发现 bar 方法的调用次数已超过 compileThreshold，于是在后台执行 JIT 编译，并继续解释执行// some bar code，进入循环时，先检查 OSR 编译是否完成。如果完成，则执行编译后的代码，如果未编译完成，则继续解释执行。

当 main 中第三次调用 foo.bar 时，如果此时 JIT 编译已完成，则进入编译后的代码；如果编译未完成，则继续按照上面所说的方式执行。

由于 Sun JDK 的这个特性，在对 Java 代码进行性能测试时，要尤其注意是否事先做了足够次数的调用，以保证测试是公平的；对于高性能的程序而言，也应考虑在程序提供给用户访问前，自行进行一定的调用，以保证关键功能的性能。

## 反射执行

反射执行是 Java 的亮点之一，基于反射可动态调用某对象实例中对应的方法、访问查看对象的属性等，无需在编写代码时就确定要创建的对象。这使得 Java 可以很灵活地实现对象的调用，例如 MVC 框架中通常要调用实现类中的 execute 方法，但框架在编写时是无法知道实现类的。在 Java 中则可以通过反射机制直接去调用应用实现类中的 execute 方法，代码示例如下：

```
Class actionClass=Class.forName(外部实现类);
Method method=actionClass.getMethod("execute",null);
Object action=actionClass.newInstance();
method.invoke(action,null);
```

这种方式对于框架之类的代码而言非常重要，反射和直接创建对象实例，调用方法的最大不同在于创建的过程、方法调用的过程是动态的。这也使得采用反射生成执行方法调用的代码并不像直接调用实例对象代码，编译后就可直接生成对对象方法调用的字节码，而是只能生成调用 JVM 反射实现的字节码了。

要实现动态的调用，最直接的方法就是动态生成字节码，并加载到 JVM 中执行，Sun JDK 采用的即为这种方法，来看看在 Sun JDK 中以上反射代码的关键执行过程。

```
Class actionClass=Class.forName(外部实现类);
```

调用本地方法，使用调用者所在的 ClassLoader 来加载创建出的 Class 对象；

```
Method method=actionClass.getMethod("execute",null);
```

校验 Class 是否为 public 类型，以确定类的执行权限，如不是 public 类型的，则直接抛出 SecurityException。

调用 privateGetDeclaredMethods 来获取 Class 中的所有方法，在 privateGetDeclaredMethods 对 Class

中所有方法集合做了缓存，第一次会调用本地方法去获取。

扫描方法集合列表中是否有相同方法名及参数类型的方法，如果有，则复制生成一个新的 Method 对象返回；如果没有，则继续扫描父类、父接口中是否有该方法；如果仍然没找到方法，则抛出 NoSuchMethodException，代码如下：

```
Object action=actionClass.newInstance();
```

校验 Class 是否为 public 类型，如果权限不足，则直接抛出 SecurityException。

如果没有缓存的构造器对象，则调用本地方法获取构造器，并复制生成一个新的构造器对象，放入缓存；如果没有空构造器，则抛出 InstantiationException。

校验构造器对象的权限。

执行构造器对象的 newInstance 方法。

判断构造器对象的 newInstance 方法是否有缓存的 ConstructorAccessor 对象，如果没有，则调用 sun.reflect.ReflectionFactory 生成新的 ConstructorAccessor 对象。

判断 sun.reflect.ReflectionFactory 是否需要调用本地代码，可通过 sun.reflect.noInflation=true 来设置为不调用本地代码。在不调用本地代码的情况下，可转交给 MethodAccessorGenerator 来处理。本地代码调用的情况在此不进行阐述。

MethodAccessorGenerator 中的 generate 方法根据 Java Class 格式规范生成字节码，字节码中包括 ConstructorAccessor 对象需要的 newInstance 方法。该 newInstance 方法对应的指令为 invokespecial，所需参数则从外部压入，生成的 Constructor 类的名字以 sun/reflect/ GeneratedSerializationConstructorAccessor 或 sun/reflect/GeneratedConstructorAccessor 开头，后面跟随一个累计创建对象的次数。

在生成字节码后将其加载到当前的 ClassLoader 中，并实例化，完成 ConstructorAccessor 对象的创建过程，并将此对象放入构造器对象的缓存中。

执行获取的 constructorAccessor.newInstance，这步和标准的方法调用没有任何区别。

```
method.invoke(action,null);
```

这步的执行过程和上一步基本类似，只是在生成字节码时方法改为了 invoke，其调用目标改为了传入对象的方法，同时类名改为了：sun/reflect/GeneratedMethodAccessor。

综上所述，执行一段反射执行的代码后，在 debug 里查看 Method 对象中的 MethodAccessor 对象引用（参数为-Dsun.reflect.noInflation=true，否则要默认执行 15 次反射调用后才能动态生成字节码），如图 3.6 所示：

```

+-- method
  +-- annotationDefault
  +-- annotations
  +-- clazz
    +-- declaredAnnotations
    +-- exceptionTypes
    +-- genericInfo
    +-- methodAccessor
    +-- modifiers
  +-- name
    +-- override
    +-- parameterAnnotations
    +-- parameterTypes
  +-- returnType

```

|                                     |
|-------------------------------------|
| Method (id=24)                      |
| null                                |
| null                                |
| Class<T> (java.lang.Object) (id=9)  |
| null                                |
| Class<T>[0] (id=29)                 |
| null                                |
| GeneratedMethodAccessor1 (id=37)    |
| 1                                   |
| "toString" (id=31)                  |
| false                               |
| null                                |
| Class<T>[0] (id=32)                 |
| Class<T> (java.lang.String) (id=18) |

图 3.6 反射执行代码示例

Sun JDK 采用以上方式提供反射的实现，提升代码编写的灵活性，但也可以看出，其整个过程比直接编译成字节码的调用复杂很多，因此性能比直接执行的慢一些。Sun JDK 中反射执行的性能会随着 JDK 版本的提升越来越好，到 JDK 6 后差距就不大了，但要注意的是，`getMethod` 相对比较耗性能，一方面是权限的校验，另一方面是所有方法的扫描及 Method 对象的复制，因此在使用反射调用多的系统中应缓存 `getMethod` 返回的 Method 对象，而 `method.invoke` 的性能则仅比直接调用低一点。一段对比直接执行、反射执行性能的程序如下所示：

```

// Server OSR 编译阈值：10700
private static final int WARMUP_COUNT=10700;
private ForReflection testClass=new ForReflection();
private static Method method=null;
public static void main(String[] args) throws Exception{
    method=ForReflection.class.getMethod("execute",new
Class<?>[]{String.class});
    Demo demo=new Demo();
    // 保证反射能生成字节码及相关的测试代码能够被 JIT 编译
    for (int i = 0; i < 20; i++) {
        demo.testDirectCall();
        demo.testCacheMethodCall();
        demo.testNoCacheMethodCall();
    }
    long beginTime=System.currentTimeMillis();
    demo.testDirectCall();
    long endTime=System.currentTimeMillis();
    System.out.println("直接调用消耗的时间为：" +(endTime-beginTime)+"毫秒");
    beginTime=System.currentTimeMillis();
    demo.testNoCacheMethodCall();
    endTime=System.currentTimeMillis();
    System.out.println("不缓存 Method, 反射调用消耗的时间为：
    " +(endTime-beginTime)+"毫秒");
    beginTime=System.currentTimeMillis();
}

```

```

    demo.testCacheMethodCall();
    endTime=System.currentTimeMillis();
    System.out.println("缓存 Method, 反射调用消耗的时间为: "+(endTime-beginTime)+"毫秒");
}
public void testDirectCall(){
    for (int i = 0; i < WARMUP_COUNT; i++) {
        testClass.execute("hello");
    }
}
public void testCacheMethodCall() throws Exception{
    for (int i = 0; i < WARMUP_COUNT; i++) {
        method.invoke(testClass, new Object[]{"hello"});
    }
}
public void testNoCacheMethodCall() throws Exception{
    for (int i = 0; i < WARMUP_COUNT; i++) {
        Method testMethod=ForReflection.class.getMethod("execute",new Class<?>[]{String.class});
        testMethod.invoke(testClass, new Object[]{"hello"});
    }
}
public class ForReflection {
    private Map<String, String> caches=new HashMap<String, String>();
    public void execute(String message){
        String b=this.toString()+message;
        caches.put(b, message);
    }
}

```

执行后显示的性能如下（执行环境： Intel Duo CPU E8400 3G, windows 7, Sun JDK 1.6.0\_18，启动参数为-server -Xms128M -Xmx128M）：

- 直接调用消耗的时间为 5 毫秒；
- 不缓存 Method，反射调用消耗的时间为 11 毫秒；
- 缓存 Method，反射调用消耗的时间为 6 毫秒。

在启动参数上增加-Xint 来禁止 JIT 编译，执行上面代码，结果为：

- 直接调用消耗的时间为 133 毫秒；
- 不缓存 Method，反射调用消耗的时间为 215 毫秒；
- 缓存 Method，反射调用消耗的时间为 150 毫秒。

对比这段测试结果也可看出，C2 编译后代码的执行速度得到了大幅提升。

## 3.2 JVM 内存管理

Java 不需要开发人员来显式分配内存和回收内存，而是由 JVM 来自动管理内存的分配及回收（又称为垃圾回收、Garbage Collection 或 GC），这对开发人员来说确实大大降低了编写程序的难度，但副作用可能是在不知不觉中浪费了很多内存，导致 JVM 花费很多时间进行内存的回收。另外还可能会带来的副作用是由于不清楚 JVM 内存的分配和回收机制，造成内存泄露，最终导致 JVM 内存不够用。因此对于 Java 开发人员而言，不能因为 JVM 自动内存管理就不掌握内存分配和回收的知识了。

除了内存的分配及回收外，还须掌握跟踪分析 JVM 内存的使用状况，以便更加准确地判断程序的运行状况及进行性能的调优。

### 3.2.1 内存空间

Sun JDK 在实现时遵照 JVM 规范，将内存空间划分为方法区、堆、本地方法栈、PC 寄存器及 JVM 方法栈，如图 3.7 所示。

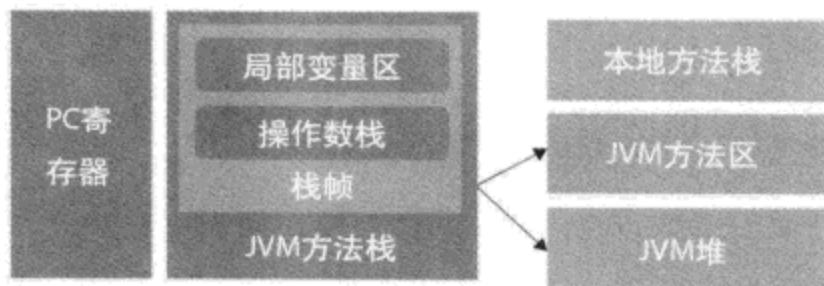


图 3.7 JVM 内存结构

#### 方法区

方法区存放了要加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为 final 类型的常量、类中的 Field 信息、类中的方法信息，当开发人员在程序中通过 Class 对象的 getName、isInterface 等方法来获取信息时，这些数据都来源于方法区域。方法区域也是全局共享的，在一定条件下它也会被 GC，当方法区域要使用的内存超过其允许的大小时，会抛出 OutOfMemory 的错误信息。

在 Sun JDK 中这块区域对应 Permanet Generation，又称为持久代，默认最小值为 16MB，最大值为 64MB，可通过-XX:PermSize 及-XX:MaxPermSize 来指定最小值和最大值。

#### 堆

堆用于存储对象实例及数组值，可以认为 Java 中所有通过 new 创建的对象的内存都在此分配，Heap 中对象所占用的内存由 GC 进行回收，在 32 位操作系统上最大为 2GB，在 64 位操作系统上则没有限制，其大小可通过-Xms 和-Xmx 来控制，-Xms 为 JVM 启动时申请的最小 Heap 内存，默认为物理内存的 1/64 但小于 1GB；-Xmx 为 JVM 可申请的最大 Heap 内存，默认为物理内存的 1/4 但小于 1GB，默认当空余堆内存小于 40% 时，JVM 会增大 Heap 到-Xmx 指定的大小，可通过-XX:MinHeapFreeRatio=

来指定这个比例；当空余堆内存大于 70%时，JVM 会减小 Heap 的大小到-Xms 指定的大小，可通过-XX:MaxHeapFreeRatio= 来指定这个比例，对于运行系统而言，为避免在运行时频繁调整 Heap 的大小，通常将-Xms 和-Xmx 的值设成一样。

为了让内存回收更加高效，Sun JDK 从 1.2 开始对堆采用了分代管理的方式，如图 3.8 所示。

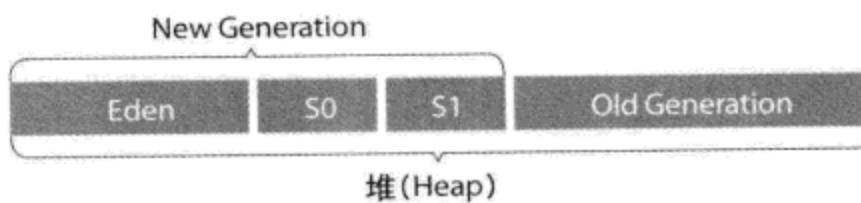


图 3.8 Sun JDK 堆的分代

## 1. 新生代 (New Generation)

大多数情况下 Java 程序中新建的对象都从新生代分配内存，新生代由 Eden Space 和两块相同大小的 Survivor Space（通常又称为 S0 和 S1 或 From 和 To）构成，可通过-Xmn 参数来指定新生代的大小，也可通过-XX:SurvivorRatio 来调整 Eden Space 及 Survivor Space 的大小。不同的 GC 方式会以不同的方式按此值来划分 Eden Space 和 Survivor Space，有些 GC 方式还会根据运行状况来动态调整 Eden、S0、S1 的大小。

## 2. 旧生代 (Old Generation 或 Tenuring Generation)

用于存放新生代中经过多次垃圾回收仍然存活的对象，例如缓存对象，新建的对象也有可能在旧生代上直接分配内存。主要有两种状况（由不同的 GC 实现来决定）：一种为大对象，可通过在启动参数上设置-XX:PretenureSizeThreshold=1024（单位为字节，默认值为 0）来代表当对象超过多大时就不在新生代分配，而是直接在旧生代分配，此参数在新生代采用 Parallel Scavenge GC 时无效，Parallel Scavenge GC 会根据运行状况决定什么对象直接在旧生代上分配内存；另一种为大的数组对象，且数组中无引用外部对象。

旧生代所占用的内存大小为-Xmx 对应的值减去-Xmn 对应的值。

## 本地方法栈

本地方法栈用于支持 native 方法的执行，存储了每个 native 方法调用的状态，在 Sun JDK 的实现中本地方法栈和 JVM 方法栈是同一个。

## PC 寄存器和 JVM 方法栈

每个线程均会创建 PC 寄存器和 JVM 方法栈，PC 寄存器占用的可能为 CPU 寄存器或操作系统内存，JVM 方法栈占用的为操作系统内存，JVM 方法栈为线程私有，其在内存分配上非常高效。当方法运行完毕时，其对应的栈帧所占用的内存也会自动释放。

当 JVM 方法栈空间不足时，会抛出 StackOverflowError 的错误，在 Sun JDK 中可以通过-Xss 来指定其大小，例如如下代码：

```

new Thread(new Runnable() {

    public void run() {
        loop(0);
    }

    private void loop (int i){
        if(i!=1000){
            i++;
        }
        else{
            return;
        }
    }
}).start();

```

当 JVM 参数设置为-Xss1K 时，运行后报出类似下面的错误：

```
Exception in thread "Thread-0" java.lang.StackOverflowError
```

### 3.2.2 内存分配

Java 对象所占用的内存主要从堆上进行分配，堆是所有线程共享的，因此在堆上分配内存时需要进行加锁，这导致了创建对象开销比较大。当堆上空间不足时，会触发 GC，如果 GC 后空间仍然不足，则抛出 OutOfMemory 错误信息。

Sun JDK 为了提升内存分配的效率，会为每个新创建的线程在新生代的 Eden Space 上分配一块独立的空间，这块空间称为 TLAB (Thread Local Allocation Buffer)，其大小由 JVM 根据运行情况计算而得，可通过-XX:TLABWasteTargetPercent 来设置 TLAB 可占用的 Eden Space 的百分比，默认值为 1%。JVM 将根据这个比率、线程数量及线程是否频繁分配对象来给每个线程分配合适大小的 TLAB 空间<sup>18</sup>。在 TLAB 上分配内存时不需要加锁，因此 JVM 在给线程中的对象分配内存时会尽量在 TLAB 上分配，如果对象过大或 TLAB 空间已用完，则仍然在堆上进行分配。因此在编写 Java 程序时，通常多个小的对象比大的对象分配起来更加高效，可通过在启动参数上增加-XX:+PrintTLAB 来查看 TLAB 空间的使用情况<sup>19</sup>。

在分配细节上取决于 GC 的实现，后续 GC 的实现章节会继续介绍。

<sup>18</sup> [http://blogs.sun.com/davidetlefs/entry/tlab\\_sizing\\_an\\_annoing\\_little](http://blogs.sun.com/davidetlefs/entry/tlab_sizing_an_annoing_little)

<sup>19</sup> [http://blogs.sun.com/jonthecollector/entry/the\\_real\\_thing](http://blogs.sun.com/jonthecollector/entry/the_real_thing)

除了从堆上分配及从 TLAB 上分配外，还有一种是基于逃逸分析直接在栈上进行分配的方式，此方式已在前文中提及。

### 3.2.3 内存回收

#### 收集器

JVM 通过 GC 来回收堆和方法区中的内存，GC 的基本原理首先会找到程序中不再被使用的对象，然后回收这些对象所占用的内存，通常采用收集器的方式实现 GC，主要的收集器有引用计数收集器和跟踪收集器。

##### 1. 引用计数收集器

引用计数收集器采用的为分散式的管理方式，通过计数器记录对象是否被引用。当计数器为零时，说明此对象已经不再被使用，于是可进行回收，如图 3.9 所示。

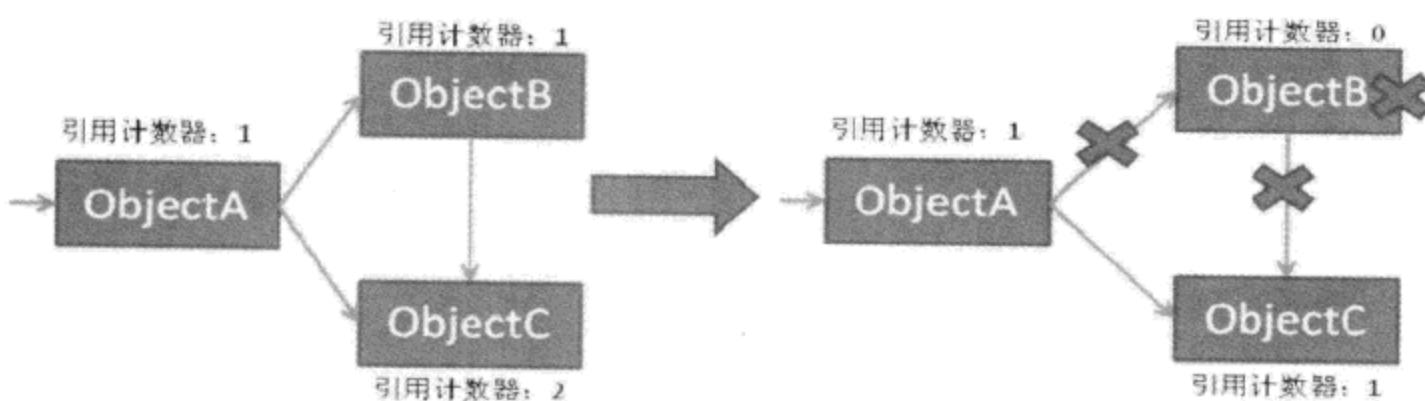


图 3.9 引用计数收集器

在图 3.9 中，当 ObjectA 释放了对 ObjectB 的引用后，ObjectB 的引用计数器即为 0，此时可回收 ObjectB 所占用的内存。

引用计数需要在每次对象赋值时进行引用计数器的增减，它有一定的消耗。另外，引用计数器对于循环引用的场景没有办法实现回收，例如上面的例子中，如果 ObjectB 和 ObjectC 互相引用，那么即使 ObjectA 释放了对 ObjectB、ObjectC 的引用，也无法回收 ObjectB、ObjectC，因此对于 Java 这种面向对象的会形成复杂引用关系的语言而言，引用计数收集器不是非常适合，Sun JDK 在实现 GC 时也未采用这种方式。

##### 2. 跟踪收集器

跟踪收集器采用的为集中式的管理方式，全局记录数据的引用状态。基于一定条件的触发（例如定时、空间不足时），执行时需要从根集合来扫描对象的引用关系，这可能会造成应用程序暂停，主要有复制（Copying）、标记-清除（Mark-Sweep）和标记-压缩（Mark-Compact）三种实现算法。

- 复制 (Copying)

复制采用的方式为从根集合扫描出存活的对象，并将找到的存活对象复制到一块新的完全未使用空间中，如图 3.10 所示。

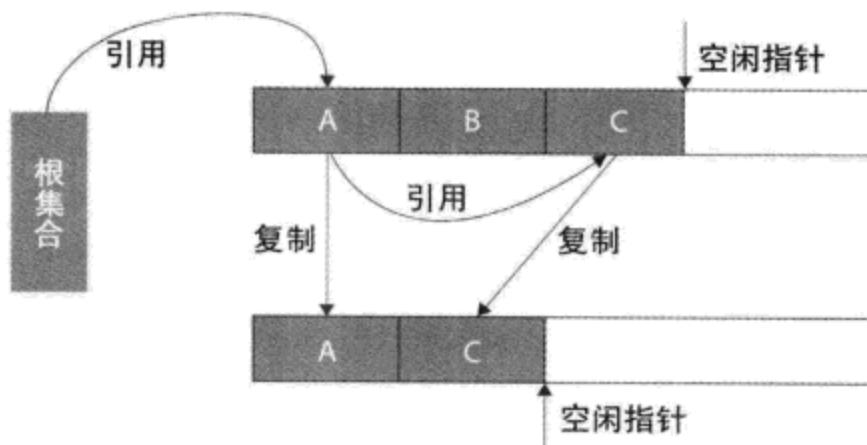


图 3.10 复制算法

复制收集器方式仅需从根集合扫描所有存活的对象，当要回收的空间中存活对象较少时，复制算法会比较高效，其带来的成本是要增加一块空的内存空间及进行对象的移动。

- 标记-清除 (Mark-Sweep)

标记-清除采用的方式为从根集合开始扫描，对存活的对象进行标记，标记完毕后，再扫描整个空间中未标记的对象，并进行回收，如图 3.11 所示。

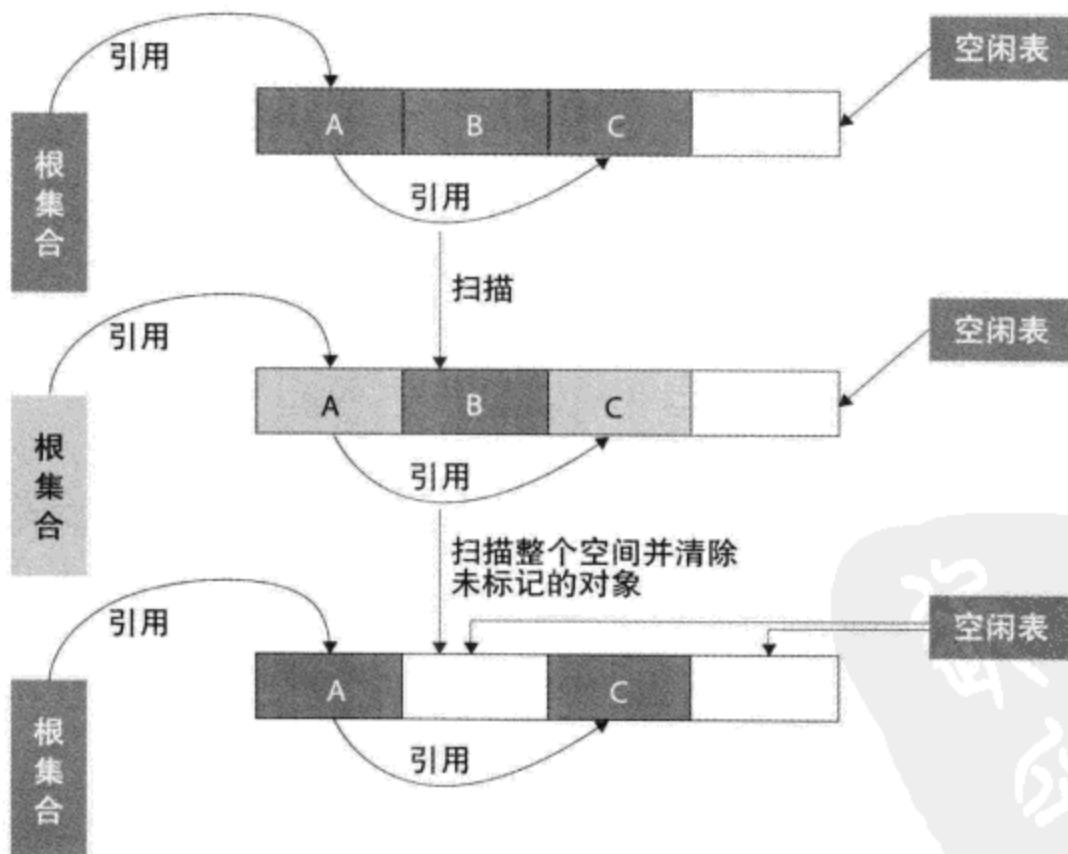


图 3.11 标记-清除算法

标记-清除动作不需要进行对象的移动，且仅对其不存活的对象进行处理。在空间中存活对象较

多的情况下较为高效，但由于标记-清除采用的为直接回收不存活对象所占用的内存，因此会造成内存碎片。

- 标记-压缩（Mark-Compact）

标记-压缩采用和标记-清除一样的方式对存活的对象进行标记，但在清除时则不同。在回收不存活对象所占用的内存空间后，会将其他所有存活对象都往左端空闲的空间进行移动，并更新引用其对象的指针，如图 3.12 所示。

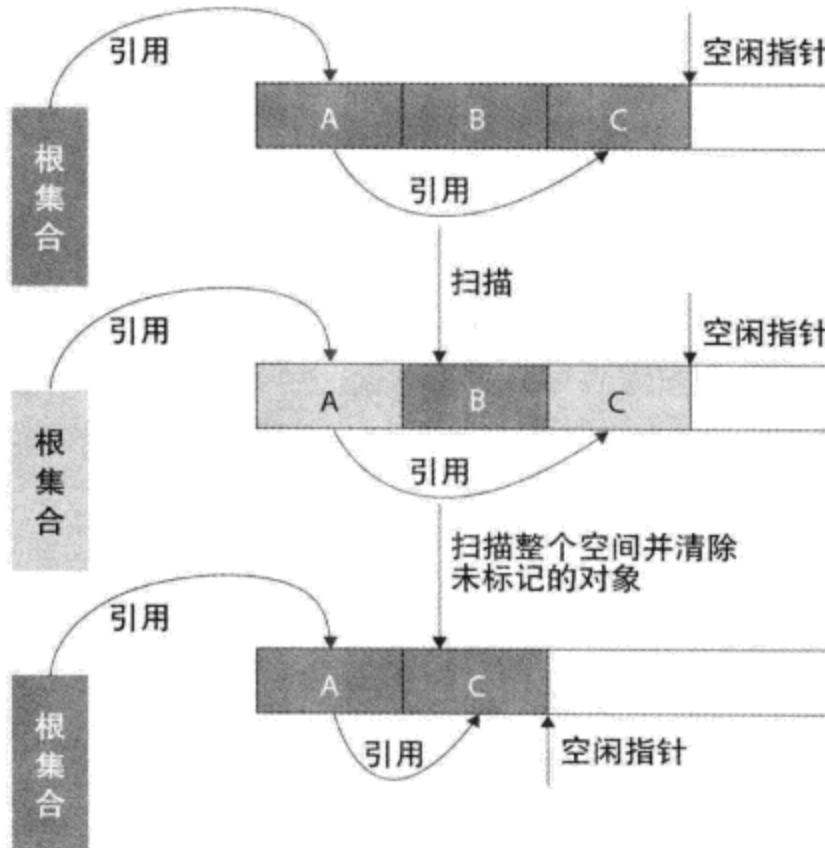


图 3.12 标记-压缩算法

标记-压缩在标记-清除的基础上还须进行对象的移动，成本相对更高，好处则是不产生内存碎片。

## Sun JDK 中可用的 GC

以上三种跟踪收集器各有优缺点，Sun JDK 根据运行的 Java 程序进行分析，认为程序中大部分对象的存活时间都是较短的，少部分对象是长期存活的。基于这个分析，Sun JDK 将 JVM 堆划分为了新生代和旧生代，并基于新生代和旧生代中对象存活时间的特征提供了不同的 GC 实现，如图 3.13 所示。



图 3.13 Sun JDK 中可用的 GC 方式

## 新生代可用 GC

Sun JDK 认为新生代中的对象通常存活时间较短，因此选择了基于 Copying 算法来实现对新生代对象的回收，根据以上 Copying 算法的介绍，在执行复制时，需要一块未使用的空间来存放存活的对象，这是新生代又被划分为 Eden、S0 和 S1 三块空间的原因。Eden Space 存放新创建的对象，S0 或 S1 的其中一块用于在 Minor GC 触发时作为复制的目标空间，当其中一块为复制的目标空间时，另一块中的内容则会被清空。因此通常又将 S0、S1 称为 From Space 和 To Space，Sun JDK 提供了串行 GC、并行回收 GC 和并行 GC 三种方式来回收新生代对象所占用的内存，对新生代对象所占用的内存进行的 GC 又通常称为 Minor GC。

### 1. 串行 GC (Serial GC)

当采用串行 GC 时，SurvivorRatio 的值对应 eden space/survivor space，SurvivorRatio 默认为 8，例如当-Xmn 设置为 10MB 时，采用串行 GC，eden space 即为 8MB，两个 survivor space 各 1MB。新生代分配内存采用的为空闲指针（bump-the-pointer）的方式，指针保持最后一个分配的对象在新生代内存区间的位置，当有新的对象要分配内存时，只须检查剩余的空间是否够存放新的对象，够则更新指针，并创建对象，不够则触发 Minor GC。

按照 Copying 算法，GC 首先需要从根集合扫描出存活的对象，对于 Minor GC 而言，其目标为扫描出在新生代中存活的对象。Sun JDK 认为以下对象为根集合对象：当前运行线程的栈上引用的对象、常量及静态（static）变量、传到本地方法中，还没有被本地方法释放的对象引用。

如果 Minor GC 仅从以上这些根集合对象中扫描新生代中的存活对象，则当旧生代中的对象引用了新生代的对象时会出现问题，但旧生代通常比较大。为提高性能，不可能每次 Minor GC 的时候去扫描整个旧生代，Sun JDK 采用了 remember set 的方式来解决这个问题。

Sun JDK 在进行对象赋值时，如果发现赋值的为一个对象引用，则产生 write barrier，然后检查需要赋值的对象是否在旧生代及赋值的对象引用是否指向新生代；如果满足条件，则在 remember set 做个标记，Sun JDK 采用了 Card Table 来实现 remember set。

因此，对于 Minor GC 而言，完整的根集合为 Sun JDK 认为的根集合对象加上 remember set 中标记的对象，在确认根集合对象后，即可进行扫描来寻找存活的对象。为了避免在扫描过程中引用关系变化，Sun JDK 采用了暂停应用的方式，Sun JDK 在编译代码时为每段方法注入了 SafePoint，通常 SafePoint 位于方法中循环的结束点及方法执行完毕的点，在暂停应用时需要等待所有的用户线程进入 SafePoint，在用户线程进入 SafePoint 后，如果发现此时要执行 Minor GC，则将其内存页设置为不可读的状态，从而实现暂停用户线程的执行。

在对象引用关系上，除了默认的强引用外，Sun JDK 还提供了软引用（SoftReference）、弱引用（WeakReference）和虚引用（PhantomReference）三种引用<sup>20</sup>。

20 [http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding\\_w.html](http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html)

- 强引用

`A a=new A();` 就是一个强引用，强引用的对象只有在主动释放了引用后才会被 GC。

- 软引用

软引用采用 `SoftReference` 来实现，采用软引用来建立引用的对象，当 JVM 内存不足时会被回收，因此 `SoftReference` 很适合用于实现缓存。另外，当 GC 认为扫描到的 `SoftReference` 不经常使用时，也会进行回收，存活时间可通过 `-XX:SoftRefLRUPolicyMSPerMB` 来进行控制，其含义为每兆堆空闲空间中 `SoftReference` 的存活时间，默认为 1 秒。

`SoftReference` 的使用方法如下：

```
Object object=new Object();
SoftReference<Object> softRef=new SoftReference<Object>(object);
object=null;
```

当需要获取时，可通过 `softRef.get` 来获取，值得注意的是 `softRef.get` 有可能会返回 `null`。

- 弱引用

弱引用采用 `WeakReference` 来实现，采用弱引用建立引用的对象没有强引用后，GC 时即会被自动释放<sup>21</sup>。

`WeakReference` 的使用方法如下：

```
Object object=new Object();
WeakReference<Object> weakRef=new WeakReference<Object>(object);
object=null;
```

当需要获取时，可通过 `weakRef.get` 来获取，值得注意的是 `weakRef.get` 有可能会返回 `null`。

可传入一个 `ReferenceQueue` 对象到 `WeakReference` 的构造器中，当 `object` 对象被标识为可回收时，执行 `weakRef.isEnqueued` 会返回 `true`。

- 虚引用

虚引用采用 `PhantomReference` 来实现，采用虚引用可跟踪到对象是否已从内存中被删除。

`PhantomReference` 的使用方法如下：

```
Object object=new Object();
ReferenceQueue<Object> refQueue=new ReferenceQueue<Object>();
PhantomReference<Object> ref=new PhantomReference<Object>(object,refQueue);
object=null;
```

<sup>21</sup> <http://forums.sun.com/thread.jspa?threadID=5221725>

值得注意的是 `ref.get` 永远返回 `null`, 当 `object` 从内存中删除时, 调用 `ref.isEnqueued()` 会返回 `true`。

当扫描引用关系时, GC 会对这三种类型的引用进行不同的处理, 简单来说, GC 首先会判断所扫描到的引用是否为 `Reference` 类型。如果为 `Reference` 类型, 且其所引用的对象无强引用, 则认为该对象为相应的 `Reference` 类型, 之后 GC 在进行回收时这些对象则根据 `Reference` 类型的不同进行相应的处理<sup>22</sup>。

当扫描存活的对象时, Minor GC 所做的动作是将存活的对象复制到目前作为 To Space 的 S0 或 S1 中; 当再次进行 Minor GC 时, 之前作为 To Space 的 S0 或 S1 则转换为 From Space, 通常存活的对象在 Minor GC 后并不是直接进入旧生代, 只有经历过几次 Minor GC 仍然存活的对象, 才放入旧生代中, 这个在 Minor GC 中存活的次数在串行和 ParNew 方式时可通过 `-XX:MaxTenuringThreshold` 来设置, 在 Parallel Scavenge 时则由 Hotspot 根据运行状况来决定。当 To Space 空间满, 剩下的存活对象则直接转入旧生代中。

Minor GC 的过程如图 3.14 所示。

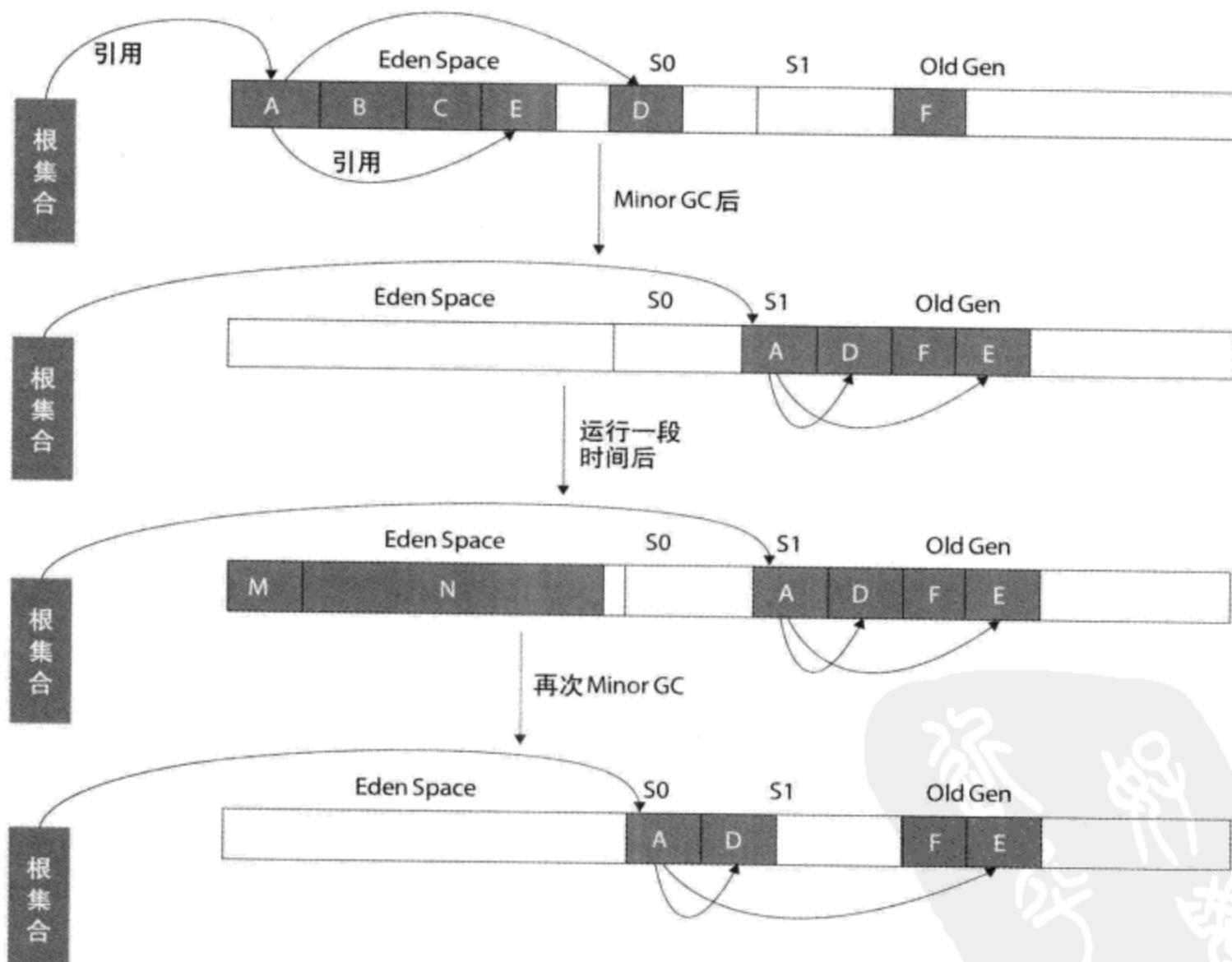


图 3.14 Minor GC 过程

22 <http://www.pawlan.com/monica/articles/refobjs/>

Serial GC 在整个扫描和复制过程均采用单线程的方式来进行，更加适用于单 CPU、新生代空间较小及对暂停时间要求不是非常高的应用上，也是 client 级别（CPU 核数小于 2 或物理内存小于 2GB）或 32 位 Windows 机器上默认采用的 GC 方式，也可通过-XX:+UseSerialGC 的方式来强制指定。

## 2. 并行回收 GC (Parallel Scavenge)

当采用并行回收 GC 时，默认情况下 Eden、S0、S1 的比例划分采用的为 InitialSurvivorRatio，此值默认为 8，对应的新生代大小为/survivor space，可通过-XX:InitialSurvivorRatio 来进行调整，在 Sun JDK 1.6.0 后也可通过-XX:SurvivorRatio 来调整<sup>23</sup>，但并行回收 GC 会将此值+2 赋给 InitialSurvivorRatio。当同时配置了 InitialSurvivorRatio 和 SurvivorRatio 时，以 InitialSurvivorRatio 对应的值为准，因此在采用并行回收 GC 时，如果不配置 InitialSurvivorRatio 或 SurvivorRatio，那么当-Xmn 设置为 16MB 时，eden space 则为 12MB，两个 Survivor Space 各 2MB；如果配置 SurvivorRatio 为 8，那么 eden space 则为 12.8MB，两个 Survivor Space 各 1.6MB，为保持和其他 GC 方式统一，建议配置 SurvivorRatio。

对于并行回收 GC 而言，在启动时 Eden、S0、S1 的比例按照上述方式进行分配，但在运行一段时间后，并行回收 GC 会根据 Minor GC 的频率、消耗时间等来动态调整 Eden、S0、S1 的大小。可通过-XX:-UseAdaptiveSizePolicy 来固定 Eden、S0、S1 的大小。

PS GC 不是根据-XX:PretenureSizeThreshold 来决定对象是否在旧生代上直接分配，而是当需要给对象分配内存时，eden space 空间不够的情况下，如果此对象的大小大于等于 eden space 一半的大小，就直接在旧生代上分配，代码示例如下：

```
public class PSGCDirectOldDemo{
    public static void main(String[] args) throws Exception{
        byte[] bytes=new byte[1024*1024*2];
        byte[] bytes2=new byte[1024*1024*2];
        byte[] bytes3=new byte[1024*1024*2];
        System.out.println("ready to direct allocate to old");
        Thread.sleep(3000);
        byte[] bytes4=new byte[1024*1024*4];
        Thread.sleep(3000);
    }
}
```

以-Xms20M -Xmn10M -Xmx20M -XX:SurvivorRatio=8 -XX:+UseParallelGC 执行以上代码，在输出 ready to direct allocate to old 后，通过 jstat 可查看到，bytes4 直接在旧生代上分配了，这里的原因就在于当给 bytes4 分配时，eden space 空间不足，而 bytes4 的大小又超过了 eden space 大小/2，因此 bytes4 就直接在旧生代上分配了。

并行回收 GC 采用的也是 Copying 算法，但其在扫描和复制时均采用多线程方式来进行，并且并行

<sup>23</sup> <http://forums.sun.com/thread.jspa?threadID=778029>

回收 GC 为大的新生代回收做了很多的优化。例如上面提到的动态调整 eden、S0、S1 的空间大小，在多 CPU 的机器上其回收时间耗费的会比串行方式短，适合于多 CPU、对暂停时间要求较短的应用上。并行回收 GC 是 server 级别（CPU 核数超过 2 且物理内存超过 2GB）的机器（32 位 Windows 机器除外）上默认采用的 GC 方式，也可通过-XX:+UseParallelGC 来强制指定，并行方式时默认的线程数根据 CPU 核数计算。当 CPU 核数小于等于 8 时，并行的线程数即为 CPU 核数；当 CPU 核数多于 8 时，则为  $3 + (\text{CPU 核数} * 5) / 8$ ，也可采用-XX:ParallelGCThreads=4 来强制指定线程数。

### 3. 并行 GC (ParNew)

并行 GC 在基于 SurvivorRatio 值划分 eden space 和两块 survivor space 的方式上和串行 GC 一样。

并行 GC 和并行回收 GC 的区别在于并行 GC 须配合旧生代使用 CMS GC，CMS GC 在进行旧生代 GC 时，有些过程是并发进行的。如此时发生 Minor GC，需要进行相应的处理<sup>24</sup>，而并行回收 GC 是没有做这些处理的，也正因为这些特殊处理，ParNew GC 不可与并行的旧生代 GC 同时使用。

在配置为使用 CMS GC 的情况下，新生代默认采用并行 GC 方式，也可通过-XX:+UseParNewGC 来强制指定。

当在 Eden Space 上分配内存时 Eden Space 空间不足，JVM 即触发 Minor GC 的执行，也可在程序中通过 System.gc 的方式（可通过在启动参数中增加-XX:+DisableExplicitGC 来避免程序中调用 System.gc 触发 GC）来触发。

## Minor GC 示例

以下通过几个例子来演示 minor GC 的触发、Minor GC 时 Survivor 空间不足的情况下对象直接进入旧生代、不同 GC 的日志。

### 1. Minor GC 触发示例

以下为一段展示 Eden Space 空间不足时 minor GC 状况的代码：

```
public class MinorGCDemo {
    public static void main(String[] args) throws Exception{
        MemoryObject object=new MemoryObject(1024*1024);
        for(int i=0;i<2;i++){
            happenMinorGC(11);
            Thread.sleep(2000);
        }
    }
    private static void happenMinorGC(int happenMinorGCIIndex) throws Exception{
        for(int i=0;i<happenMinorGCIIndex;i++){
            if(i==happenMinorGCIIndex-1){
                Thread.sleep(2000);
            }
        }
    }
}
```

<sup>24</sup> [http://blogs.sun.com/jonthecollector/entry/our\\_collectors](http://blogs.sun.com/jonthecollector/entry/our_collectors)

```

        System.out.println("minor gc should happen");
    }
    new MemoryObject(1024*1024);
}
}

class MemoryObject{
    private byte[] bytes;
    public MemoryObject(int objectSize){
        this.bytes=new byte[objectSize];
    }
}

```

以-Xms40M -Xmx40M -Xmn16M -verbose:gc -XX:+PrintGCDetails 参数执行（执行机器cpu为6核，物理内存4GB，os：linux 2.6.18 32 bit，jdk为sun 1.6.0 update 18）以上代码，此时的GC方式为默认的并行回收GC方式，按照这样的参数，Eden space大小为12MB，S0和S1各2MB，旧生代为24MB。通过jstat -gcutil [pid] 1000 10 查看Eden、S0、S1、old在minor GC时的变化情况。

在输出 minor gc should happen 后，jstat 输出如下信息：

| S0   | S1    | E    | O    | P     | YGC | YGCT  | FGC | FGCT  | GCT   |
|------|-------|------|------|-------|-----|-------|-----|-------|-------|
| 0.00 | 57.82 | 8.33 | 0.00 | 10.33 | 1   | 0.011 | 0   | 0.000 | 0.011 |

从代码分析可以看出，在输出 minor gc should happen 后，此时 Eden space 中已分配了 11 个 1MB 的对象，再分配 1 个 1MB 对象时，就导致 Eden space 空间不足，因此触发 minor GC，minor GC 执行完毕后可看到 S0、旧生代仍然为 0。新生代此时则由新创建的 1MB 对象占用了 8%，S1 则由于 object 对象而占用了 57.82%，此次 minor GC 耗时为 11ms。

同时程序执行的控制台上在 minor gc should happen 后出现了以下信息：

```
[GC [PSYoungGen: 11509K->1184K(14336K)] 11509K->1184K(38912K), 0.0113360 secs]
[Times: user=0.03 sys=0.01, real=0.01 secs]
```

上面信息中关键部分的含义为：

PSYoungGen 表示 GC 的方式为 PS，即 Parallel Scavenge GC；

PSYoungGen 后面的 11509K->1184K(14336K) 表示在 Minor GC 前，新生代使用空间为 11 509KB，回收后新生代占用空间为 1 184KB。为什么和 jstat 不一样，是因为 jstat 输出的时候新分配的对象已占用了 eden space 空间，新生代总共可用空间为 14 336KB；

11509K->1184K(38912K) 表示在 Minor GC 前，堆使用空间为 11 509KB，回收后使用空间为 1 184KB，总共可使用空间为 38 912KB；

0.0113360 secs 表示此次 Minor GC 消耗的时间；

Times: user=0.03 sys=0.01、real=0.01 secs 表示 Minor GC 占用 cpu user 和 sys 的百分比，以及消耗的总时间。

当再次输出 minor gc should happen 后，jstat 输出如下信息：

|       |      |      |      |       |   |       |   |       |       |
|-------|------|------|------|-------|---|-------|---|-------|-------|
| 57.81 | 0.00 | 8.33 | 0.00 | 10.33 | 2 | 0.020 | 0 | 0.000 | 0.020 |
|-------|------|------|------|-------|---|-------|---|-------|-------|

同样也是由于 Eden space 空间不足触发的 minor GC，这次 minor GC 后可看到 S0 空间使用了 57.81%，而 S1 空间变成了 0，表明每次 minor GC 时将进行 S0 和 S1 的交换，此次 Minor GC 耗时为 9ms。

## 2. Minor GC 时 survivor 空间不足，对象直接进入旧世代的示例

根据分析，按照示例一的启动参数，survivor 空间大小为 2MB，于是修改代码如下：

```
public static void main(String[] args) throws Exception{
    MemoryObject object=new MemoryObject(1024*1024);
    MemoryObject m2object=new MemoryObject(1024*1024*2);
    happenMinorGC(9);
    Thread.sleep(2000);
}
```

按同样的启动参数执行此段代码，在输出 minor gc should happen 信息后，jstat 输出的信息如下所示：

|      |       |      |      |       |   |       |   |       |       |
|------|-------|------|------|-------|---|-------|---|-------|-------|
| 0.00 | 57.43 | 8.33 | 8.33 | 10.33 | 1 | 0.020 | 0 | 0.000 | 0.020 |
|------|-------|------|------|-------|---|-------|---|-------|-------|

从以上代码可以看出，当 minor GC 触发时，此时需要放入 survivor 空间的有 object、m2object。两个对象加起来占据了 3MB 多的空间，而 survivor 空间只有 2MB，按顺序先放入的为 object，这样 m2object 就无法放入，于是 m2object 对象被直接放入了旧世代中，这就可以解释为什么在 minor GC 执行后旧世代空间被占用了 8.33%。

## 3. 不同 GC 的日志示例

将上面代码修改为仅触发一次 Minor GC 的情况：

```
MemoryObject object=new MemoryObject(1024*1024);
happenMinorGC(11);
Thread.sleep(2000);
```

由于 Serial GC 对 SurvivorRatio 值的使用和并行回收 GC 不同，因此在使用 Serial GC 运行上面代码时，需要调整下 SurvivorRatio 值，用以下参数启动：

```
-XX:+UseSerialGC -Xms40M -Xmx40M -Xmn16M -verbose:gc -XX:+PrintGCDetails  
-XX:SurvivorRatio=6
```

在控制台输出 minor gc should happen 后，可看到如下 gc 日志信息：

```
[GC [DefNew: 11509K->1138K(14336K), 0.0110060 secs] 11509K->1138K(38912K),  
0.0112610 secs] [Times: user=0.00 sys=0.01, real=0.01 secs]
```

和并行回收 GC 不同的地方在于标识 GC 方式的地方，这里为 DefNew，表明使用的为串行 GC 方式。

改为以下参数来查看并行 GC 时的日志信息如下：

```
-XX:+UseParNewGC -Xms40M -Xmx40M -Xmn16M -verbose:gc -XX:+PrintGCDetails  
-XX:SurvivorRatio=6
```

其输出的 GC 日志信息为：

```
[GC [ParNew: 11509K->1152K(14336K), 0.0129150 secs] 11509K->1152K(38912K),  
0.0131890 secs] [Times: user=0.05 sys=0.02, real=0.02 secs]
```

不同点在于标识 GC 方式的地方，这里为 ParNew，表明使用的为并行 GC 方式。

## 旧生代和持久代可用的 GC

JDK 提供了串行、并行及并发三种 GC 来对旧生代及持久代对象所占用的内存进行回收。

### 1. 串行

串行基于 Mark-Sweep-Compact 实现，Mark-Sweep-Compact 结合 Mark-Sweep、Mark-Compact 做了一些改进。

采用串行方式时，旧生代的内存分配方式和新生代采用的串行方式相同。

串行 GC 分为三个阶段来执行：

- 1) 从根集合对象开始扫描，按照三色着色<sup>25</sup>的方式对对象进行标识；
- 2) 遍历整个旧生代空间或持久代空间，找出其中未标识的对象，并回收其内存；
- 3) 执行滑动压缩（Sliding compaction），将存活的对象向旧生代空间的开始处进行滑动，最终留出一块连续的到结尾处的空间。

串行执行的整个过程需暂停应用，且采用的为单线程方式，通常要耗费较长的时间，可通过增加 -XX:+PrintGCApplicationStoppedTime 来查看 GC 造成的应用暂停的时间。

串行是 client 级别或 32 位的 Windows 机器上默认采用的 GC 方式，也可通过-XX:+UseSerialGC 来

<sup>25</sup> [http://www.algolist.net/Algorithms/Graph/Undirected/Depth-first\\_search](http://www.algolist.net/Algorithms/Graph/Undirected/Depth-first_search)

强制指定。

## 2. 并行<sup>26</sup>

并行采用 Mark- Compact 实现，在内存分配方式上则和串行方式相同。

并行 GC 分为三个阶段来执行，如图 3.15 所示。

1) 首先将代空间划分为并行线程个数的区域 (regions)，然后根据根集合可直接访问到的对象及并行线程的个数进行划分，并行地对这些对象进行三色着色。当对象被着色时，同时更新其所在的 region 存活的大小及存活对象所在的位置；

2) 经分析，大部分情况下经过多次 GC 后，通常旧生代空间左边存放的是一些活跃的对象。对这些对象进行压缩移动是不值得的，因此并行 GC 在这一步做了优化，方式为从最左边开始往右扫描 regions，直到找到第一个值得进行压缩移动的 region，并将此 region 左边的 region 作为密度高区 (dense prefix)，对这些区域则不进行回收；然后继续往右扫，对于右边的 regions 根据存活的空间来决定压缩移动的源 region 和目标 region，切换引用这些对象的指针，并在 region 上做标志，同时清除 regions 中其他不存活对象所占用的空间，目前此过程为单线程进行。

3) 基于 regions 上分析的信息，找到需要操作的目标 region 及完全没有存活对象的 region，并行地进行对象移动和 region 的回收。

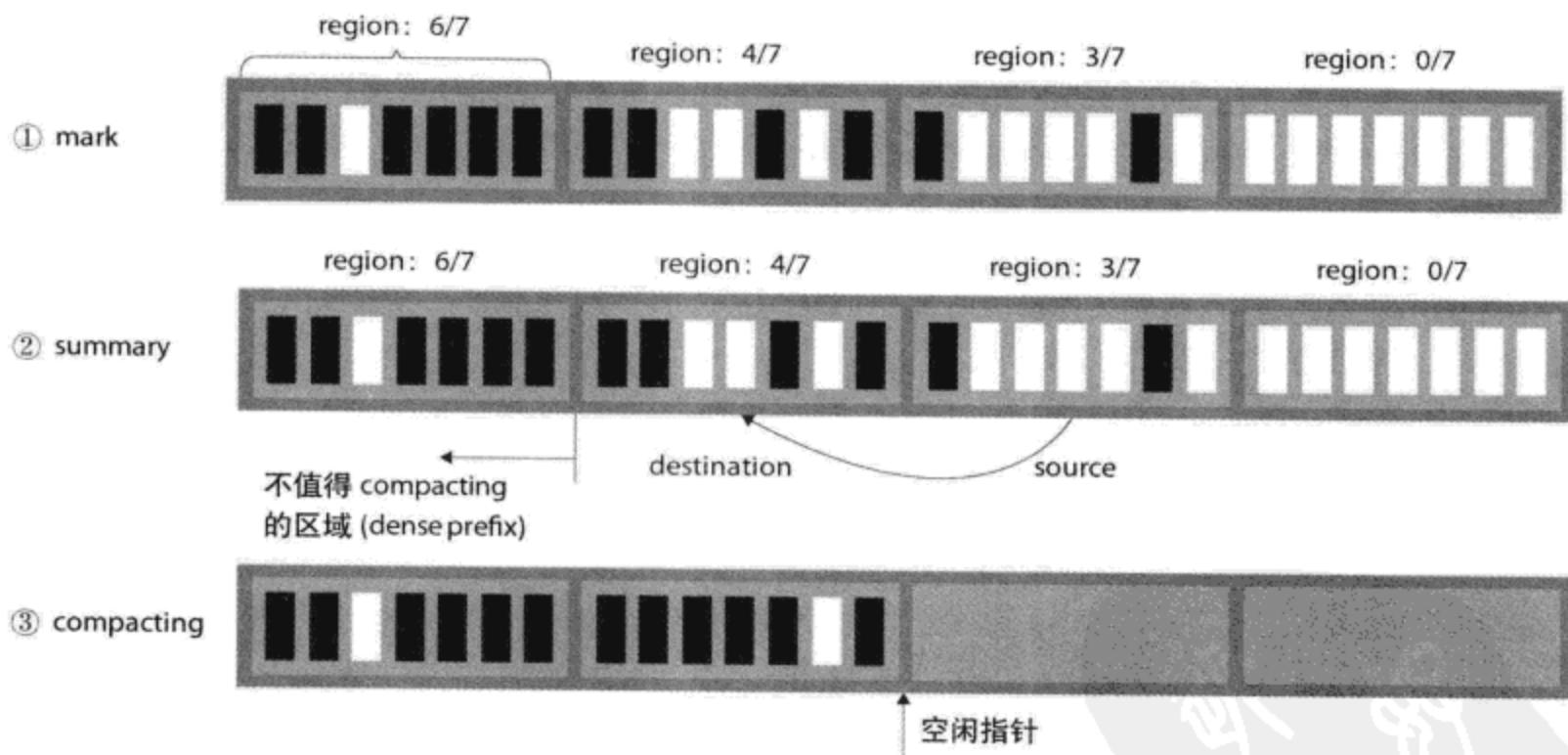


图 3.15 并行 GC 执行过程

较之串行方式而言，并行大部分时候是多线程同时进行操作，再加上其所做的 dense prefix 的优化，因此其对应用造成的暂停时间会缩短。但由于旧生代较大，在扫描和标识对象上需要花费较长的时间，

<sup>26</sup> <http://research.sun.com/jtech/pubs/01-pargc.pdf>

因此仍然要耗费一定的应用暂停的时间，并行是 server 级别机器（非 32 位 Windows）上默认采用的 GC 方式，也可通过-XX:+UseParallelGC 或-XX:+UseParallelOldGC 来强制指定。

### 3. 并发 (CMS: Concurrent Mark-Sweep GC)<sup>27</sup>

Mark-Sweep 方式要对整个空间中的对象进行扫描并标记，这个过程会造成较长时间的应用暂停，有些应用对响应时间有很高的要求。因此 Sun JDK 提供了 CMS GC，好处为 GC 的大部分动作均与应用并发进行，因此可大大缩短 GC 造成应用暂停的时间。

CMS 采用的是 Mark-Sweep 方式，其在回收完毕后可能会形成多个空闲的空间，这就没办法再采用 bump-the-pointer 的方式来分配内存了，于是 CMS 采用了 free list 的方式来记录旧生代空间中哪些部分是空闲的。当有对象要在旧生代分配内存时，就要先去 free list 中寻找哪个部分是可以放下这个对象的。多数情况下旧生代分配内存的请求都来源于 Minor GC 阶段，CMS 这种分配旧生代内存的方式会导致 Minor GC 的速度下降。

另外，由于 CMS 执行过程中大部分时候是和应用并发进行的，分配内存的动作有可能和回收内存的动作同时进行，这时会造成 free list 竞争激烈，CMS 为了避免这个现象，引入了 Mutual exclusion locks，以 JVM 分配内存为优先。

CMS 执行的扫描、着色和清除步骤如下。

#### 1. 第一次标记 (Initial Marking)

该步骤须暂停整个应用，扫描从根集合对象到旧生代中可直接访问的对象，并对这些对象进行着色。对于着色的对象 CMS 采用一个外部的 bit 数组来进行记录。

#### 2. 并发标记 (Concurrent Marking)

在初始化标记完毕后，CMS 恢复所有应用的线程，同时开始并发对之前着色过的对象进行轮循，以标记这些对象可访问的对象。

CMS 为确保能够扫描到所有的对象，避免在 Initial Marking 中还有遗漏的未着色的对象，采用的方法为找到着色的对象，并将这些对象放入 Stack 中。扫描时寻找所依赖的对象，如果依赖的对象地址在其之前，则将此对象也进行着色，并同时放入 Stack 中，如果依赖的对象地址在其之后，则仅着色。

在进行 Concurrent Marking 时 Minor GC 可能会同时进行，这时很容易造成旧生代对象引用关系改变，CMS 为应对这样的并发现象，提供了一个 Mod Union Table 来进行记录。在这个 Mod Union Table 中记录每次 Minor GC 后修改的 Card 的信息，这也是在采用 CMS 时新生代 GC 必须采用 Serial GC 或 ParNew GC 的原因。

在进行 Concurrent Marking 时还可能会上出现的一个并发现象是，应用修改了旧生代中对象的引用关系，CMS 中采用 Card Table 的方式进行记录，并在 Card 中将某对象标识为 dirty 状态。但即使这

<sup>27</sup> [http://research.sun.com/techrep/2000/sml\\_i\\_tr-2000-88.pdf](http://research.sun.com/techrep/2000/sml_i_tr-2000-88.pdf)

样，仍然可能出现一种导致不再被引用的对象是 marked 的状态，图 3.16 所示为一个这种情况的例子。

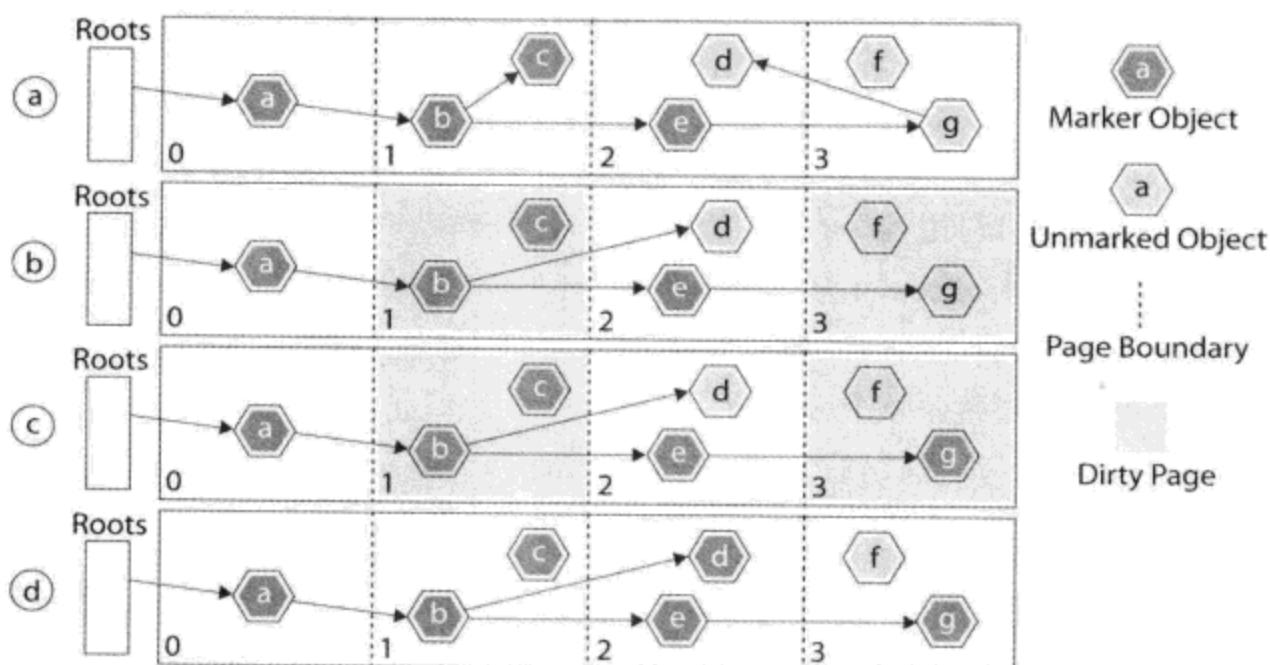


图 3.16 CMS GC 时浮动垃圾产生的示例

Concurrent Marking 扫描到 a 引用了对象 b，b 引用了 c 和 e。如果在此之后应用将 b 引用的对象由 c 改为了 d，同时 g 不再引用 d，此时会将 b、g 对象的状态在 card 中标识为 dirty，但 c 的状态并不会因此而改变。

### 3. 重新标记 (Final Marking (remark))

该步需要暂停整个应用，在 Concurrent Marking 时应用可能会修改对象的引用关系或创建新的对象，因此要对这些改变或新创建的对象也进行扫描，包括 Mod Union Table 及 Card Table 中 dirty 的对象，并重新进行着色。

### 4. 并发收集 (Concurrent Sweeping)

在完成了 Final Marking 后，恢复所有应用的线程，就进入到这步了，这步要负责的是将没有标记的对象进行回收。

由于内存碎片的原因，可能会造成每次回收的内存比之前分配出去的小。为避免这种现象，在进行 sweeping 的时候，CMS 会尽量将相邻的块重新组装为一个块，采用的方法为首先从 free list 中删除块，组装完毕后再重新放入 free list 中。

从以上整个步骤来看，CMS 中只有 Initial Marking 和 Final Marking 需要暂停整个应用，其他动作均与应用并发进行，这也是它能够做到 GC 过程中影响应用时间很短的原因。但同时由于并发进行，也意味着 CMS 会和应用线程争抢 CPU 资源，为降低和应用争抢 CPU 资源的现象发生，CMS 还提供了一种增量的模式，称为 i-CMS。在这种模式下，CMS 仅启动一个处理器线程来并发扫描标记和清除，并且该线程在执行一小段时间后会先将 CPU 使用权让出来，分多次多段的方式来完成整个扫描标记和清除的过程，这样降低了对 CPU 资源的消耗，同时也降低了 CMS 的性能和回收的效率，因此仅适用于

CPU少和内存分配不频繁的应用。

对比并行GC，CMS GC需要执行三次mark，因此其完整的一次GC执行的时间会比并行GC长。对于关注GC总耗时的应用而言，CMS GC并不是合适的选择。

另外，CMS回收内存的方式使得其很容易产生内存碎片，降低了内存空间的利用率。为了减少产生的内存碎片，提高空间的利用率，CMS提供了一个整理碎片的功能，可通过在JVM中指定`-XX:+UseCMSCompactAtFullCollection`来启动此功能。在启动此功能后默认为每次执行Full GC时都会进行整理，也可通过`-XX:CMSFullGCsBeforeCompaction=`来指定多少次Full GC后才执行整理。值得注意的是，整理这个步骤是需要暂停整个应用的。

除内存碎片外，CMS在回收时容易产生一些应该回收但要等到下次CMS才能被回收掉的对象，例如图3.16中的c对象，通常把这些对象称为“浮动垃圾”，再加上CMS回收过程中大部分时间是和应用并发进行的，因此该过程中应用可能会分配内存，这就要求采用CMS的情况下需要提供更多可用的旧生代空间。

默认情况下CMS GC并不开启，可通过在启动参数上增加`-XX:+UseConcMarkSweepGC`来启用CMS进行旧生代对象的GC，其默认开启的回收线程数为（并行GC线程数+3）/4，可通过`-XX:ParallelCMSThreads=10`来强行指定。

CMS GC触发的条件为旧生代已使用的空间达到设定的`CMSInitiatingOccupancyFraction`百分比，例如默认`CMSInitiatingOccupancyFraction`为68%，如旧生代空间为1 000MB，那么当旧生代已使用的空间达到680MB时，CMS GC即开始执行；另外一种触发方式是JVM自动触发，JVM基于之前GC的频率及旧生代的增长趋势来评估决定什么时候执行CMS GC，如果不希望JVM自行触发，可设置`-XX:UseCMSInitiatingOccupancyOnly=true`。

持久代的GC也可采用CMS方式，方式为设置以下参数：`-XX:+CMSPermGenSweepingEnabled`  
`-XX:+CMSClassUnloadingEnabled`。

由于CMS GC在执行时需要分为多个步骤，其输出的GC日志信息也会比较多，下面为一段CMS GC的日志及对其的解释<sup>28</sup>。

```
[GC [1 CMS-initial-mark: 13433K(20480K)] 14465K(29696K), 0.0001830 secs] [Times:  
user=0.00 sys=0.00, real=0.00 secs]
```

开始执行CMS GC，进行Initial Marking步骤，旧生代的空间为20 480KB，CMS GC在旧生代被占用了13 433KB后触发。

```
[CMS-concurrent-mark: 0.004/0.004 secs] [Times: user=0.01 sys=0.00, real=0.01  
secs]
```

28 [http://www.sun.com/bigadmin/contentsubmitted/cms\\_gc\\_logs.jsp](http://www.sun.com/bigadmin/contentsubmitted/cms_gc_logs.jsp)

完成 Concurrent mark 步骤，耗时 4ms。

```
[CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

该步用于重新扫描在 concurrent mark 阶段 CMS Heap 中被新创建的对象或从新生代晋升到旧生代对象的引用关系，以减少 remark 所需耗费的时间，这是 Sun JDK 1.5 后增加的一个优化步骤。

```
CMS: abort preclean due to time [CMS-concurrent-abortable-preclean: 0.007/5.042 secs] [Times: user=0.00 sys=0.00, real=5.04 secs]
```

当 eden space 占用超过 2MB 时，执行此步，并且将一直并发的执行到 eden space 的使用率超过 50%，之后触发 remark 动作，这两个值可通过 -XX: CMSScheduleRemarkEdenSizeThreshold 和 -XX: CMSScheduleRemarkEdenPenetration 来设置。以上日志信息表示为 preclean 动作执行了 5 秒后，eden space 使用仍然未超过 50%，此时也停止执行 preclean，触发 remark 动作。5 秒这个值可通过 -XX: CMSMaxAbortablePrecleanTime=5000（单位为毫秒）来进行设置。

```
[GC[YG occupancy: 3300 K (9216 K)] [Rescan (parallel), 0.0002740 secs] [weak refs processing, 0.0000090 secs] [1 CMS-remark: 13433K(20480K)] 16734K(29696K), 0.0003710 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

执行并完成 remark 动作。

```
[CMS-concurrent-sweep: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

执行并完成 concurrent sweeping 动作。

```
[CMS-concurrent-reset: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

重新初始化 CMS 的相关数据，为下次 CMS GC 执行做准备。

除以上 GC 实现外，在 JDK 5 以前的版本中还有一个 GC 实现是增量收集器，增量收集器可通过 -Xincgc 来启用，但在 JDK 5 及以上版本中废弃了此增量收集器。当在这些版本中设置 -Xincgc 时，会自动转为采用并行收集器去进行垃圾回收，原因是其性能低于并行收集器，因此本书中就不介绍此收集器了。

## Full GC

除 CMS GC 外，当旧生代和持久化触发 GC 时，其实是对新生代、旧生代及持久代都进行 GC，因此通常又称为 Full GC。当 Full GC 被触发时，首先按照新生代所配置的 GC 方式对新生代进行 GC（在新生代采用 PS GC 时，可通过 -XX:-ScavengeBeforeFullGC 来禁止 Full GC 时对新生代进行 GC），然后

按照旧生代的 GC 方式对旧生代、持久代进行 GC。但其中有一种特殊现象，如在进行 Minor GC 前，可能 Minor GC 后移到旧生代的对象多于旧生代的剩余空间，这种情况下 Minor GC 就不会执行了，而是直接采用旧生代的 GC 方式来对新生代、旧生代及持久代进行回收。

除直接调用 System.gc 外，触发 Full GC 执行的情况有如下四种。

### 1. 旧生代空间不足

旧生代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行 Full GC 后空间仍然不足，则抛出如下错误：

```
java.lang.OutOfMemoryError: Java heap space
```

为避免以上两种状况引起的 Full GC，调优时应尽量做到让对象在 Minor GC 阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。

### 2. Permanet Generation 空间满

Permanet Generation 中存放的为一些 class 的信息等，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation 可能会被占满，在未配置为采用 CMS GC 的情况下会执行 Full GC。如果经过 Full GC 仍然回收不了，那么 JVM 会抛出如下错误信息：

```
java.lang.OutOfMemoryError: PermGen space
```

为避免 Perm Gen 占满造成 Full GC 现象，可采用的方法为增大 Perm Gen 空间或转为使用 CMS GC。

### 3. CMS GC 时出现 promotion failed 和 concurrent mode failure

对于采用 CMS 进行旧生代 GC 的程序而言，尤其要注意 GC 日志中是否有 promotion failed 和 concurrent mode failure 两种状况，当这两种状况出现时可能会触发 Full GC。

promotion failed 是在进行 Minor GC 时，survivor space 放不下、对象只能放入旧生代，而此时旧生代也放不下造成的；concurrent mode failure 是在执行 CMS GC 的过程中同时有对象要放入旧生代，而此时旧生代空间不足造成的。

应对措施为：增大 survivor space、旧生代空间或调低触发并发 GC 的比率，但在 JDK 5.0+、6.0+ 的版本中有可能会由于 JDK 的 bug<sup>29</sup> 导致 CMS 在 remark 完毕后很久才触发 sweeping 动作。对于这种状况，可通过设置-XX: CMSMaxAbortablePrecleanTime=5（单位为 ms）来避免。

<sup>29</sup> <http://www.nabble.com/CMS-GC-tuning-under-JVM-5.0-td16759819.html>

#### 4. 统计得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间

这是一个较为复杂的触发情况，Hotspot 为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行 Minor GC 时，做了一个判断，如果之前统计所得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发 Full GC。

例如程序第一次触发 Minor GC 后，有 6MB 的对象晋升到旧生代，那么当下一次 Minor GC 发生时，首先检查旧生代的剩余空间是否大于 6MB，如果小于 6MB，则执行 Full GC。

当新生代采用 PS GC 时，方式稍有不同，PS GC 是在 Minor GC 后也会检查，例如上面的例子中第一次 Minor GC 后，PS GC 会检查此时旧生代的剩余空间是否大于 6MB，如小于，则触发对旧生代的回收。

除了以上 4 种状况外，对于使用 RMI 来进行 RPC 或管理的 Sun JDK 应用而言，默认情况下会一小时执行一次 Full GC。可通过在启动时通过 -Dsun.rmi.dgc.client.gcInterval=3600000 来设置 Full GC 执行的间隔时间或通过 -XX:+ DisableExplicitGC 来禁止 RMI 调用 System.gc。

### Full GC 示例

以下通过几个例子来演示 Full GC 在旧生代占满和 CMS GC 失败时的触发，以及不同 Full GC 时的日志信息。

#### 1. 旧生代空间不足触发的 Full GC

代码如下：

```
public class TestFullGC{
    public static void main(String[] args) throws Exception{
        List<MemoryObject> objects=new ArrayList<MemoryObject>(6);
        for(int i=0;i<10;i++){
            objects.add(new MemoryObject(1024*1024));
        }
        // 让上面的对象尽可能地转入旧生代中
        System.gc();
        System.gc();
        Thread.sleep(2000);
        objects.clear();
        for(int i=0;i<10;i++){
            objects.add(new MemoryObject(1024*1024));
            if(i%3==0){
                objects.remove(0);
            }
        }
        Thread.sleep(5000);
    }
}
```

```

    }
}

class MemoryObject{
    private byte[] bytes;
    public MemoryObject(int objectSize){
        this.bytes=new byte[objectSize];
    }
}

```

以-Xms20M -Xmx20M -Xmn10M -verbose:gc -XX:+PrintGCDetails 执行以上代码，可看到在输出的日志中出现了 Full GC 的信息：

```
[Full GC [PSYoungGen: 7248K->0K(8960K)] [PSOldGen: 9330K->4210K(10240K)]
16578K->4210K(19200K) [PSPermGen: 1692K->1692K(16384K)], 0.0085500 secs] [Times:
user=0.01 sys=0.00, real=0.01 secs]
```

上面日志的含义为执行了一次 Full GC，新生代的方式为并行回收 GC 方式，回收前内存使用了 7 248KB，回收后为 0KB，可使用的内存为 8 960KB；旧生代的方式为并行 GC 方式，回收前内存使用了 9 330KB，回收后为 4 210KB，可使用的内存为 10 240KB；回收前 JVM 堆使用的内存为 16 578KB，回收后为 4 210KB，可使用的内存为 19 200KB；PSPermGen 回收前为 1 692KB，回收后仍然为 1 692KB，可使用的内存为 16 384KB；回收 Perm Gen 消耗的时间为 7ms，整个 Full GC 耗费的时间为 10ms，通过 System.gc 调用的 Full GC 在日志上会显示为 Full GC (System)。

由于执行以上代码的机器为 server 级机器，因此其默认采用的为-XX:+UseParallelGC，改为采用-XX:+UseSerialGC，Full GC 的信息如下：

```
[Full GC [Tenured: 9216K->4210K(10240K), 0.0066570 secs] 16584K->4210K(19456K),
[Perm : 1692K->1692K(16384K)], 0.0067070 secs] [Times: user=0.00 sys=0.00,
real=0.01 secs]
```

上面日志的含义为执行了一次 Full GC，旧生代的回收方式采用的为串行方式，回收前为 9 216KB，回收后为 4 210KB，旧生代可用的内存为 10 240KB，回收耗时 6ms，其他信息则和 ParallelGC 时基本相同。

## 2. CMS GC 失败触发的 Full GC

按照 CMS GC concurrent mode failure 失败的原因，编写了如下代码：

```

public class TestCMSGC{
    public static void main(String[] args) throws Exception{
        List<MemoryObject> objects=new ArrayList<MemoryObject>(6);
        for(int i=0;i<9;i++){
            objects.add(new MemoryObject(1024*1024));
        }
        Thread.sleep(2000);
        objects.remove(0);
        objects.remove(0);
        objects.remove(0);
        for(int i=0;i<20;i++){
            objects.add(new MemoryObject(1024*1024));
            if(i%2==0){
                objects.remove(0);
            }
        }
        Thread.sleep(5000);
    }
}

```

以-Xms20M -Xmx20M -Xmn10M -verbose:gc -XX:+PrintGCDetails -XX:+UseConcMarkSweepGC 参数执行，在输出信息中可看到类似信息：

```

[Full GC [CMS [CMS-concurrent-mark: 0.004/0.006 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
(concurrent mode failure): 9331K->9331K(10240K), 0.0183120 secs]
16499K->15475K(19456K), [CMS Perm : 1692K->1692K(16384K)], 0.0184020 secs] [Times:
user=0.02 sys=0.00, real=0.02 secs]

```

由于是并发操作的，所以日志有重叠。

### 3. 统计得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间

测试代码如下：

```

public static void main(String[] args) throws Exception{
    byte[] bytes=new byte[1024*1024*2];
    byte[] bytes2=new byte[1024*1024*2];
    byte[] bytes3=new byte[1024*1024*2];
}

```

```

        System.out.println("ready to happen one minor gc,if parallel scavenge
gc,then should one full gc");
        byte[] bytes4=new byte[1024*1024*2];
        Thread.sleep(3000);
        System.out.println("minor gc end");
        byte[] bytes5=new byte[1024*1024*2];
        byte[] bytes6=new byte[1024*1024*2];
        System.out.println("minor gc again ,and should direct full gc");
        byte[] bytes7=new byte[1024*1024*2];
        Thread.sleep(3000);
    }
}

```

以 `java -Xms20M -Xmx20M -Xmn10M -verbose:gc -XX:+PrintGCDetails -XX:+UseParallelGC` 执行以上代码，可看到在输出 `ready to happen one minor gc,if parallel scanvege gc,then should one full gc` 后，系统输出了一次 `minor gc` 和一次 `full gc` 的信息。当输出 `minor gc again, and should direct full gc` 时，系统执行了一次 `full gc`，而观看这两次 `full gc` 的日志信息，会看到此时旧生代、perm 均没有满：

```

[Full GC [PSYoungGen: 176K->0K(8960K)] [PSOldGen: 6144K->6258K(10240K)]
6320K->6258K(19200K) [PSPermGen: 1685K->1685K(16384K)], 0.0065500 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
[Full GC [PSYoungGen: 6224K->4096K(8960K)] [PSOldGen: 6258K->8306K(10240K)]
12482K->12402K(19200K) [PSPermGen: 1692K->1692K(16384K)], 0.0222810 secs] [Times:
user=0.01 sys=0.01, real=0.02 secs]

```

结合代码以及之前介绍的触发机制，可看到在输出 `ready to happen one minor gc` 后，系统分配了一个 2MB 的对象，此时 `eden space` 空间不足，触发 `minor gc`，`minor gc` 后有 6MB 的对象进入了旧生代，此时旧生代剩余空间为 4MB，于是执行 `full gc`。

在输出 `minor gc again` 后，`eden space` 空间再次不足，于是再度触发 `minor gc`，此时 `minor gc` 发现之前统计出来的晋升到旧生代对象的平均值为 6MB，并和旧生代的剩余空间进行比较，发现旧生代空间更小，于是直接执行 `full gc`。

当将执行参数切换为 `java -Xms20M -Xmx20M -Xmn10M -verbose:gc -XX:+PrintGCDetails -XX:+UseSerialGC` 后，可看到系统先后执行了一次 `minor gc` 和一次 `full gc`。

## 小结

综上所述，Sun JDK 为高效实现内存的分配及内存的回收，提供了多种 GC 的实现，在 client 和 server 模式下的默认选择并不相同，如表 3.1 所示。

表 3.1 client、server 模式的默认 GC

|        | 新生代 GC 方式 | 旧生代和持久代 GC 方式               |
|--------|-----------|-----------------------------|
| Client | 串行 GC     | 串行 GC                       |
| Server | 并行回收 GC   | 并行 GC (JDK 5.0 Update 6 以后) |

Sun JDK 同时提供了一些参数供开发人员来自行选择或组合 GC，可使用的参数或组合如表 3.2 所示。

表 3.2 Sun JDK GC 的组合方式

|   | 新生代 GC 方式   | 旧生代和持久代 GC 方式   |
|---|---|---|
| -XX:+UseSerialGC                            | 串行 GC   | 串行 GC   |
| -XX:+UseParallelGC                          | 并行回收 GC   | 并行 GC   |
| -XX:+UseConcMarkSweepGC                     | 并行 GC   | 并发 GC<br>当出现 concurrent Mode failure 时采用串行 GC                     |
| -XX:+UseParNewGC                            | 并行 GC   | 串行 GC   |
| -XX:+UseParallelOldGC                       | 并行回收 GC   | 并行 GC   |
| -XX:+UseConcMarkSweepGC<br>-XX:-UseParNewGC | 串行 GC   | 并发 GC<br>当出现 Concurrent Mode failure 或 promotion failed 时则采用串行 GC |
| 不支持的组合方式                                    | 1. -XX:+UseParNewGC -XX:+UseParallelOldGC<br>2. -XX:+UseParNewGC -XX:+UseSerialGC |   |

为了避免开发人员选择哪种 GC 而头疼，Sun JDK 还提供了两种简单的方式来帮助选择 GC。

### 1. 吞吐量优先

吞吐量是指 GC 所耗费的时间占应用运行总时间的百分比，例如应用总共运行了 100 分钟，其中 GC 执行占用了 1 分钟，那么吞吐量就是 99%，JVM 默认的指标是 99%。

吞吐量优先的策略即为以吞吐量为指标，由 JVM 自行选择相应的 GC 策略及控制 New Generation、Old Generation 内存的大小，可通过在 JVM 参数中指定-XX:GCTimeRatio=n 来使用此策略。

### 2. 暂停时间优先

暂停时间是指每次 GC 造成的应用的停顿时间，默认不启用这个策略。

暂停时间优先的策略即为以暂停时间为指标，由 JVM 自行选择相应的 GC 策略及控制 New Generation、Old Generation 内存的大小，来尽量保证每次 GC 造成的应用停顿时间都在指定的数值范围内完成，可通过在 JVM 参数中指定-XX:MaxGCPauseMillis=n 来使用此策略。

当以上两参数都指定的情况下，首先满足暂停时间优先策略，再满足吞吐量优先策略，这两个策略要通过收集运行信息来做动态调整，实现难度较高，目前还很难做到精确的控制，并且动态地调整可能会给应用带来影响，以及大部分应用的需求其实是一定时间范围内吞吐量是多少或暂停时间最多是多少，这在这两种策略中无法配置。

大多数情况下只须配置 JVM 堆的大小及持久代的大小就可以让 GC 符合应用的要求运行，对于访问量、数据量较大的应用而言，如果确定 GC 对应用的运行状况造成了影响，则可根据应用状况来精确控制内存空间中每个代的大小、选择 GC 方式及调整 GC 参数，尽可能降低 GC 对应用运行的影响。

### Garbage First<sup>30</sup>

除了以上这些已有的 GC 外，为了能够做到控制某个时间片内 GC 所能占用的最大暂停时间，例如在 100 秒内最多允许 GC 导致的应用暂停时间为 1 秒，以满足对响应时间有很高要求的应用，Sun JDK 6 Update 14 以上版本及 JDK 7 中增加了一种称为 Garbage First 的 GC。

Garbage First 简称 G1，它的目标是要做到尽量减少 GC 所导致的应用暂停的时间，同时保持 JVM 堆空间的利用率。

G1 要做到这样的效果，是有前提的，一方面是硬件环境的要求，必须是多核的 CPU 及较大的内存（从规范来看，512MB 以上就满足条件了）；另一方面是要接受 GC 吞吐量的稍微降低，对于响应时间要求高的系统而言，这点应该是可以接受的。

G1 在原有的各种 GC 策略上进行了吸收和改进，G1 中吸收了增量收集器和 CMS GC 策略的优点，并在此基础上做了很多改进。

G1 将整个 jvm Heap 划分为多个固定大小的 region，扫描时采用 Snapshot-at-the-beginning 的并发 marking 算法对整个 heap 中的 region 进行 mark。回收时根据 region 中活跃对象的 bytes 进行排序，首先回收活跃对象 bytes 小及回收耗时短（预估出来的时间）的 region，回收的方法为将此 region 中的活跃对象复制到另外的 region 中，根据指定的 GC 所能占用的时间来估算能回收多少 region。这点和以前版本的 Full GC 时的处理整个 heap 非常不同，这样就做到了能够尽量短时间地暂停应用，又能回收内存，由于这种策略首先回收的是垃圾对象所占空间最多的 region，因此称为 Garbage First。

G1 将 Heap 划分为多个固定大小的 region，这也是 G1 能够实现控制 GC 导致的应用暂停时间的前提。region 之间的对象引用通过 remembered set 来维护，每个 region 都有一个 remembered set，remembered set 中包含了引用当前 region 中对象的 region 对象的 pointer，应用会造成 region 中对象的引用关系不断发生改变，G1 采用了 Card Table 来用于应用通知 region 修改 remembered sets，Card Table 由多个 512 字节的 Card 构成，这些 Card 在 Card Table 中以 1 个字节来标识，每个应用的线程都有一个关联的 remembered set log，用于缓存和顺序化线程运行时造成的对于 card 的修改。另外，还有一

<sup>30</sup> <http://research.sun.com/jtech/pubs/04-g1-paper-ismm.pdf>

个全局的 filled RS buffers，当应用线程执行时修改了 card 后，如果造成的改变仅为同一 region 中的对象之间的关联，则不记录 remembered set log；如果造成的改变为跨 region 中的对象的关联，则记录到线程的 remembered set log；如果线程的 remembered set log 满了，则放入全局的 filled RS buffers 中，线程自身则重新创建一个新的 remembered set log，remembered set 本身也是一个由一堆 cards 构成的哈希表。

尽管 G1 将 Heap 划分为了多个 region，但其默认采用的仍然是分代的方式，只是名称改为了年轻代（young）和非年轻代，这也是由于 G1 仍然坚信大多数新创建的对象是不需要长的生命周期的。对于应用新创建的对象，G1 将其放入标识为 young 的 region 中，这些 region 并不记录 remembered set logs，扫描时只扫描活跃的对象。

G1 在分代的方式上还可更细地划分为：fully young 或 partially young。fully young 方式暂停的时候仅处理 young regions，partially 同样处理所有的 young regions，但它还会根据允许的 GC 的暂停时间来决定是否要加入其他的非 young regions。G1 是采用 fully-young 方式还是 partially young 方式，外部是不能决定的。启动时，G1 采用的为 fully-young 方式，当 G1 完成一次 Concurrent Marking 后，则切换为 partially young 方式，随后 G1 跟踪每次回收的效率，如果回收 fully-young 中的 regions 已经可以满足内存需要，就切换回 fully young 方式。但当 heap size 的大小接近满的情况下，G1 会切换到 partially young 方式，以保证能提供足够的内存空间给应用使用。

除了分代方式的划分外，G1 还支持另外一种 pure G1 的方式，也就是不进行代的划分，pure 方式和分代方式的具体不同在以下具体执行步骤中进行描述。

下面介绍 G1 的具体执行步骤。

### 1. Initial Marking

G1 对于每个 region 都保存了两个标识用的 bitmap，一个为 previous marking bitmap，一个为 next marking bitmap，bitmap 中包含了一个 bit 的地址信息来指向对象的起始点。

开始 Initial Marking 之前，首先并发清空 next marking bitmap，然后停止所有应用线程，并扫描标识出每个 region 中 root 可直接访问到的对象，将 region 中 top 的值放入 next top at mark start (TAMS) 中，之后恢复所有应用线程。

触发该步骤执行的条件为：

- G1 定义了一个 JVM Heap 大小的百分比的阈值，称为 h，另外还有一个 H 值为  $(1-h) * \text{Heap Size}$ 。目前该 h 的值是固定的，后续 G1 也许会将其改为根据 jvm 的运行情况来动态地调整。在分代方式下，G1 还定义了一个 u 及 soft limit，soft limit 的值为  $H - u * \text{Heap Size}$ ，当 Heap 中使用的内存超过 soft limit 值时，就会在一次 clean up 执行完毕后在应用允许的 GC 暂停时间范围内尽快执行该步骤；
- 在 pure 方式下，G1 将 marking 与 clean up 组成一个环，以便 clean up 能充分使用 marking 的信息。当 clean up 开始回收时，首先回收能够带来最多内存空间的 regions；当经过多次的 clean up，回

收到没多少空间的 regions 时，G1 重新初始化一个新的 marking 与 clean up 构成的环。

## 2. Concurrent Marking

按照之前 Initial Marking 扫描到的对象进行遍历，以识别这些对象的下层对象的活跃状态，对于在此期间应用线程并发修改的对象的关系则记录到 remembered set logs 中，新创建的对象则放入比 top 值更高的地址区间中。这些新创建的对象默认状态即为活跃的，同时修改 top 值。

## 3. Final Marking Pause

当应用线程的 remembered set logs 未满时，是不会放入 filled RS buffers 中的，这些 remembered set logs 中记录的 card 修改就不会被更新了，因此需要这一步把应用线程中存在的 remembered set logs 的内容进行处理，并相应修改 remembered sets，此步需要暂停应用，并行运行。

## 4. Live Data Counting and Cleanup

值得注意的是，在 G1 中，并不是说 Final Marking Pause 执行完了，就肯定执行 Cleanup。由于这步需要暂停应用，G1 为了能够达到实时的要求，需要根据用户指定的最大的 GC 造成的暂停时间来合理规划什么时候执行 Cleanup，另外还有几种情况也会触发这个步骤的执行：

- G1 采用复制方法来进行收集，必须保证每次的“to space”的空间都是够的，因此 G1 采取的策略是当已经使用的内存空间达到 H 时，就执行 Cleanup 这个步骤；
- 对于 full-young 和 partially-young 的分代模式的 G1 而言，还有其他情况会触发 Cleanup 的执行。full-young 模式下，G1 根据应用可接受的暂停时间、回收 young regions 需要消耗的时间来估算出一个 young regions 的数量值，当 JVM 中分配对象的 young regions 的数量达到此值时，Cleanup 就会执行；partially-young 模式下，则会尽量频繁的在应用可接受的暂停时间范围内执行 Cleanup，并最大限度地去执行 non-young regions 的 Cleanup。

这一步中 GC 线程并行扫描所有 region，计算每个 region 中低于 next TAMS 值中 marked data 的大小，然后根据应用所期望的 GC 的短延时及 G1 对于 region 回收所需的耗时的预估，排序 region，将其中活跃的对象复制到其他 region 中。

从以上 G1 的步骤可看出，G1 和 CMS GC 非常类似，只是 G1 将 JVM 划分为了更小粒度的 regions，并在回收时进行了估算。回收能够带来最大内存空间的 regions，从而缩短了每次回收所须消耗的时间。

一个简单的演示 G1 的例子，代码如下：

```
public class TestG1{
    public static void main(String[] args) throws Exception{
        List<MemoryObject> objects=new ArrayList<MemoryObject>();
        for(int i=0;i<20;i++){
            objects.add(new MemoryObject(1024*1024));
            if(i%3==0){
                objects.remove(0);
            }
        }
    }
}
```

```
        }
        Thread.sleep(2000);
    }
}
```

在 Sun JDK 1.6.0 Update 18 上，以-Xms40M -Xmx40M -Xmn20M -verbose:gc -XX:+Unlock-ExperimentalVMOptions -XX:+UseG1GC -XX:MaxGCPauseMillis=5 -XX:GCPauseIntervalMillis=1000 -XX:+PrintGCDetails 执行以上代码，在输出信息中可看到类似如下的信息：

```

[GC pause (young), 0.00328600 secs]
[Parallel Time: 3.1 ms]
  [Update RS (Start) (ms): 142.8 143.9 143.1 142.6 142.9 143.9]
  [Update RS (ms): 0.2 0.0 0.0 0.0 0.0 0.0
   Avg: 0.0, Min: 0.0, Max: 0.2]
    [Processed Buffers : 5 0 0 0 0 0
     Sum: 5, Avg: 0, Min: 0, Max: 5]
  [Ext Root Scanning (ms): 1.4 2.3 1.0 0.0 0.0 0.0
   Avg: 0.8, Min: 0.0, Max: 2.3]
  [Mark Stack Scanning (ms): 0.0 0.0 0.0 0.0 0.0 0.0
   Avg: 0.0, Min: 0.0, Max: 0.0]
  [Scan-Only Scanning (ms): 0.0 0.0 0.0 0.0 0.0 0.0
   Avg: 0.0, Min: 0.0, Max: 0.0]
    [Scan-Only Regions : 0 0 0 0 0 0
     Sum: 0, Avg: 0, Min: 0, Max: 0]
  [Scan RS (ms): 0.0 0.0 0.0 0.0 0.0 0.0
   Avg: 0.0, Min: 0.0, Max: 0.0]
  [Object Copy (ms): 0.7 0.0 0.9 1.0 1.2 0.0
   Avg: 0.6, Min: 0.0, Max: 1.2]
  [Termination (ms): 0.5 0.2 0.4 0.5 0.0 0.2
   Avg: 0.3, Min: 0.0, Max: 0.5]
  [Other: 1.2 ms]
  [Clear CT: 0.1 ms]
  [Other: 0.1 ms]
  [ 199K->132K(40M) ]
[Times: user=0.01 sys=0.00, real=0.01 secs]

```

Real-Time JDK

为了满足实时领域系统使用 Java 的需求, Java 推出了 Real-Time 版的规范 (JSR-001, 更新的版本为 JSR-282), 各大厂商均积极响应, 相应推出了 Real-Time 实现的 JDK。Real-Time 版的 JDK 对 Java 做出了很多的改进, 例如强大的线程调度机制、异步的事件处理机制、更为精准的时间刻度等, 在此最为关心的是其在 Java 内存管理方面的加强。

GC 无疑是 Java 进入实时领域的一个很大障碍，毕竟无论 GC 怎么改进，它肯定会造成应用暂停的现象，而且会在运行时突然造成暂停。这对于实时系统来说是不可接受的，因此 Real-Time 版的 JDK

在此方面做了多方面的改进。

### 1. 新的内存管理机制

提供了两种内存区域：Immortal 内存区域和 Scoped 内存区域。

Immortal 内存区域用于保留永久的对象，这些对象仅在应用结束运行时才会释放内存，这对于支持需要缓存的系统而言非常重要。

Scoped 内存区域用于保留临时的对象，位于 scope 中的对象在 scope 退出时，这些对象所占用的内存会被直接回收，有点类似于栈上分配。

Immortal 内存区域和 Scoped 内存区域均不受 GC 管理，因此基于这两个内存区域来编写的应用完全不用担心 GC 会造成暂停的现象。

### 2. 允许 Java 应用直接访问物理内存

在保证安全的情况下，Real-Time JDK 允许 Java 应用直接访问物理内存，而非像以前的 Java 程序，需要通过 native code 才能访问。能够访问物理内存，也就意味着可以直接将对象放入物理内存，而非 JVM heap 中。

## 3.2.4 JVM 内存状况查看方法和分析工具

Java 本身提供了多种丰富的方法和工具来帮助开发人员查看和分析 GC 及 JVM 内存的状况，同时开源界和商业界也有一些工具可用于查看、分析 GC 及 JVM 内存的状况。通过这些分析，可以排查程序中内存泄露的问题及调优程序的性能。下面介绍几种常用的免费工具，其中知名的有 JProfiler<sup>31</sup>等。

### 1. 输出 GC 日志

输出 GC 日志对于跟踪分析 GC 的状况来说，无疑是最直接地分析内存回收状况的方法，只是 GC 日志输出后需要人为地进行分析，以判断 GC 的状况。

JVM 支持将日志输出到控制台或指定的文件中，方法有如下几种。

- 输出到控制台

在 JVM 的启动参数中加入 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCApplicationStoppedTime，按照参数的顺序分别输出 GC 的简要信息，GC 的详细信息、GC 的时间信息及 GC 造成的应用暂停的时间。

- 输出到指定的文件

在 1 中的 jvm 启动参数中再增加-Xloggc: gc.log 可指定将 gc 的信息输出到 gc.log 中。

可用于 GC 跟踪分析的参数还有-verbose:gc、-XX:+PrintTenuringDistribution 等。

<sup>31</sup> <http://www.ej-technologies.com/products/jprofiler/overview.html>

## 2. GC Portal

将 GC 日志输出固然有一定的作用，但如果要靠人为进行分析，还是相当复杂的。因此 Sun 提供了一个 GC Portal 来帮助分析这些 GC 日志，并生成相关的图形化的报表，GC Portal 部署起来会有些麻烦，它需要运行在老版本的 Tomcat 上，同时需要数据库，部署完毕后通过上传日志文件的方式即可完成 GC 日志的分析，此 GC 日志输出的 JVM 参数为：-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps [-Xloggc:文件名]，在上传日志时 GC Portal 的选项里只有 jdk 1.2 或 jdk 1.2—1.4 的版本。虽然经过测试，JDK 6 的日志也是可以分析出来的，但它的限制在于仅支持 5MB 的 gc 日志的分析，GC Portal 可提供吞吐量的分析、耗费的 CPU 的时间、造成应用暂停的时间、每秒从新生代转化到旧生代的数量、minor GC 的状况及 Full GC 的状况等，如图 3.17 所示。

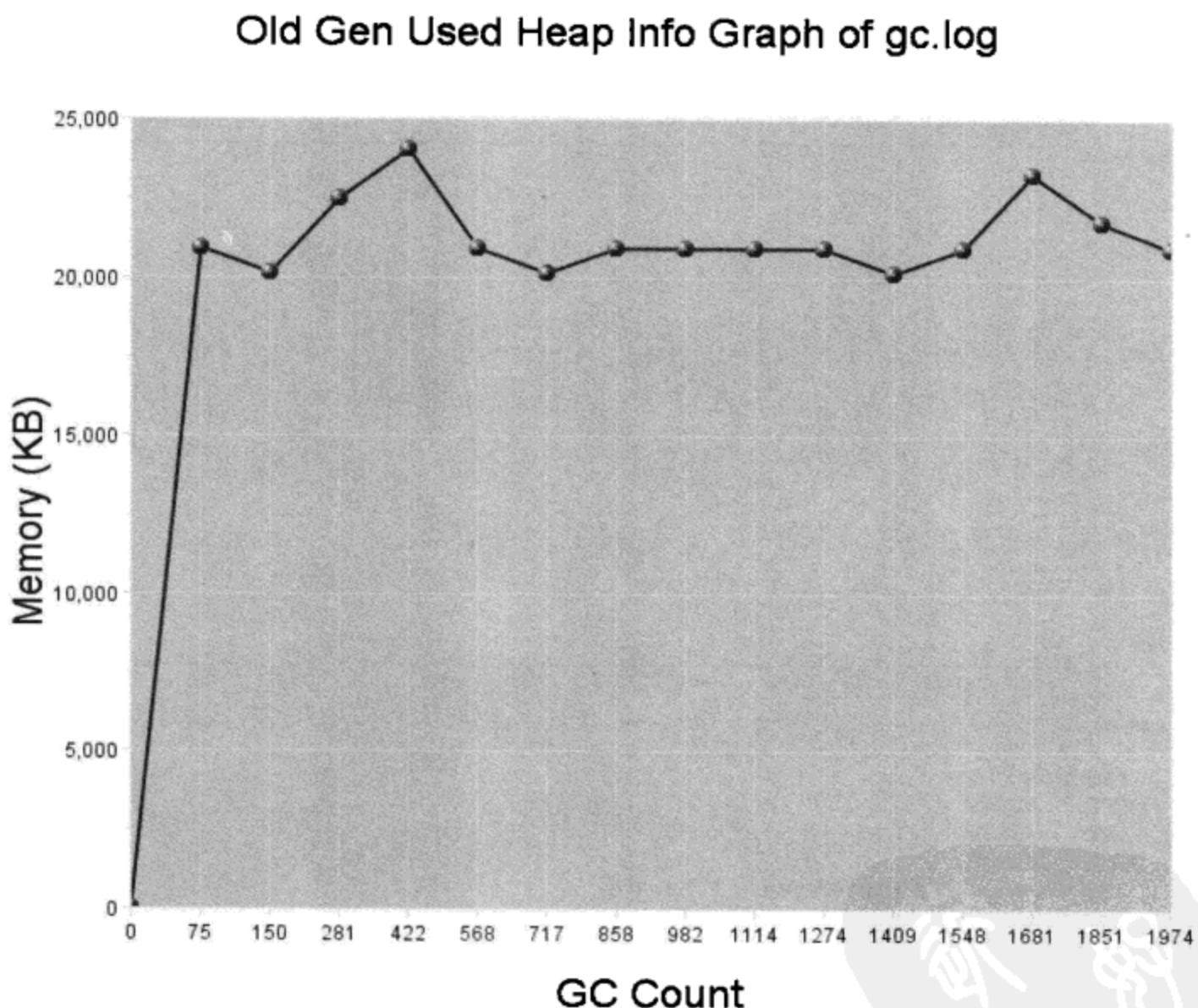


图 3.17 GCPortal 示例

GC Portal 中还有一个很有用的部分是提供调整 GC 参数的预测，例如可以选择给 young size 增加 20% 的空间。GC Portal 会根据当前的日志信息来评估在调整参数后的运行效果，不一定很准确，但毕竟能够带来一些参考意义。

### 3. JConsole

JConsole 可以图形化查看 JVM 中内存的变化状况, JConsole 是 JDK 5 及以上版本中自带的工具, 位于 JDK 的 bin 目录下, 运行时直接运行 JConsole.exe 或 JConsole.sh (要求支持图形界面)。在本地的 Tab 页上看到运行了 java 的 pid, 双击即可查看相应进程的 JVM 状况, 同时, JConsole 也支持查看远程的 JVM 的运行状况, 具体可参见 JConsole 的 User Guide。

JConsole 中显示了 JVM 中很多的信息: 内存、线程、类和 MBean 等, 在打开 JConsole 的内存 Tab 页后, 可看到 JVM 内存部分的运行状况。这对于分析内存是否有溢出及 GC 的效果更加直接明了, JConsole 的运行效果如图 3.18 所示。

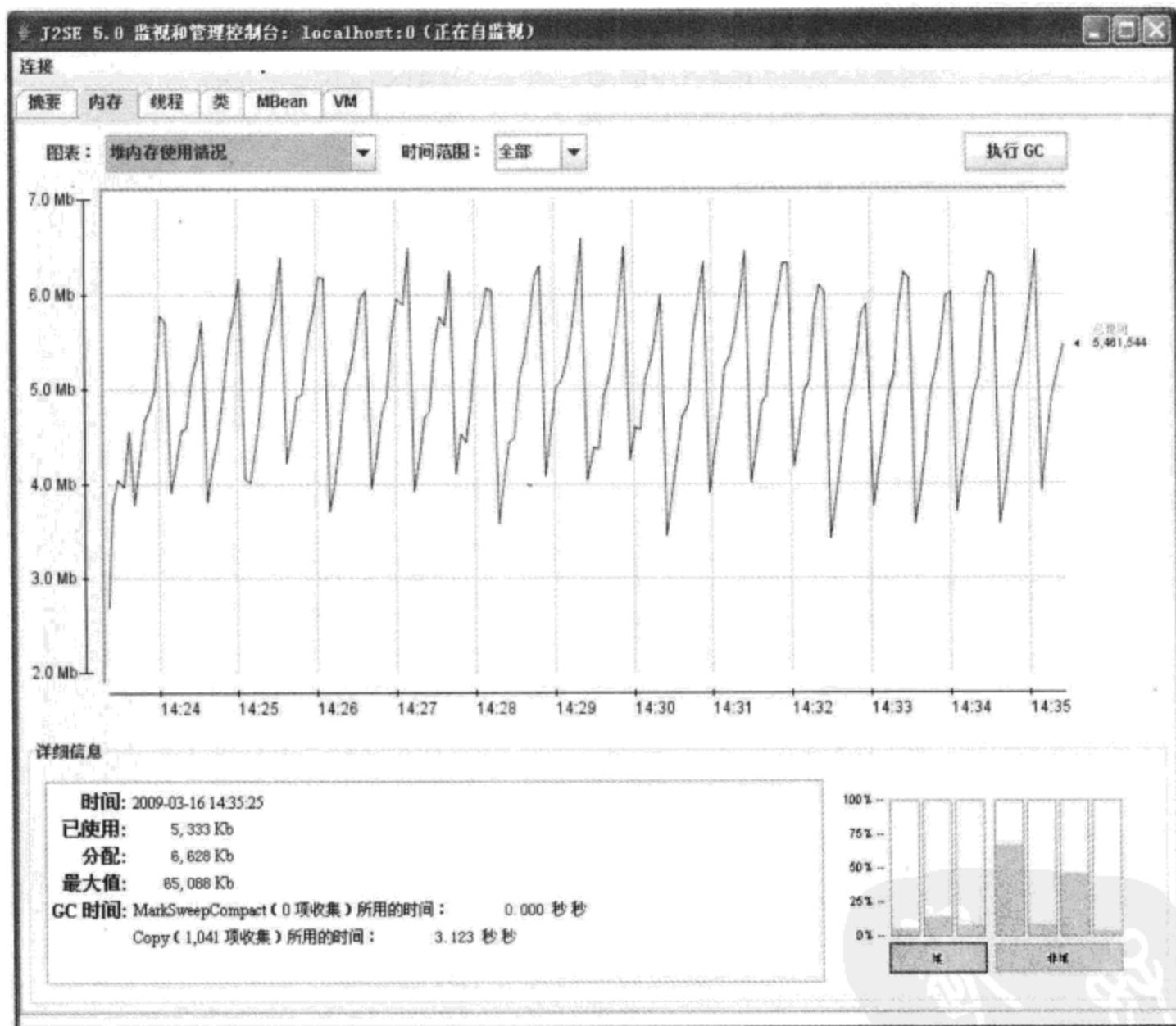


图 3.18 JConsole 运行效果

### 4. JVisualVM

JVisualVM 是 JDK 6 update 7 之后推出的一个工具, 它类似于 JProfiler 的工具, 基于此工具可查看内存的消耗情况、线程的执行状况及程序中消耗 CPU、内存的动作。

在内存分析上，JVisualVM 的最大好处是可通过安装 VisualGC 插件来分析 GC 趋势、内存消耗详细状况。

VisualGC 的运行如图 3.19 所示。

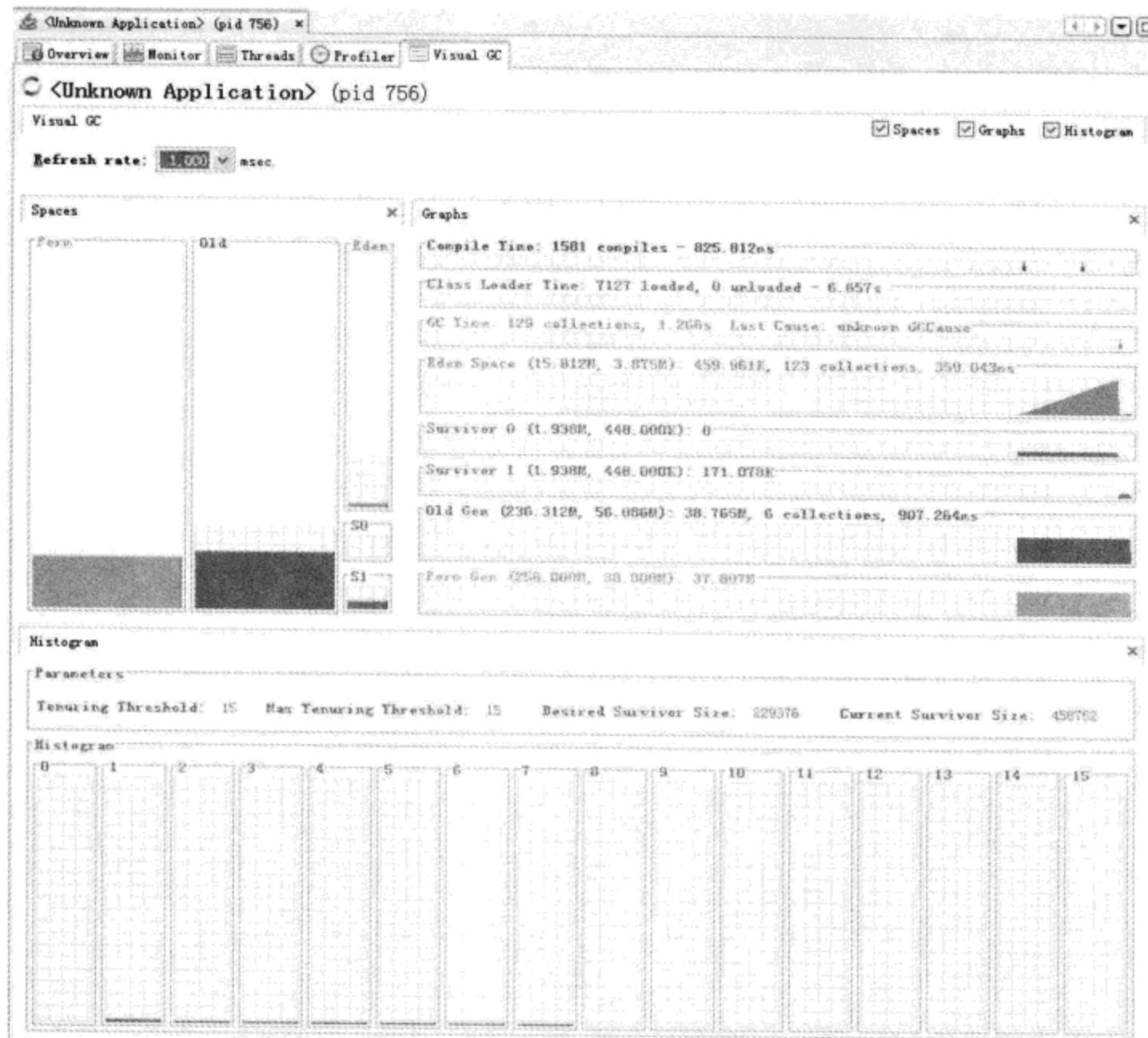


图 3.19 VisualGC 运行效果

从图 3.19 中可看到各区的内存消耗状况及 GC Time 的图表，其提供的 Histogram 视图对于调优也有很大帮助。

基于 JVisualVM 的 Profiler 中的 Memory 还可查看对象占用内存的状况，如图 3.20 所示。

#### 4. JMap

JMap 是 JDK 中自带的一个用于分析 JVM 内存状况的工具，位于 JDK 的 bin 目录下。使用 JMap 可查看目前 JVM 中各个代的内存状况、JVM 中对象的内存的占用状况，以及导出整个 JVM 中的内存信息。

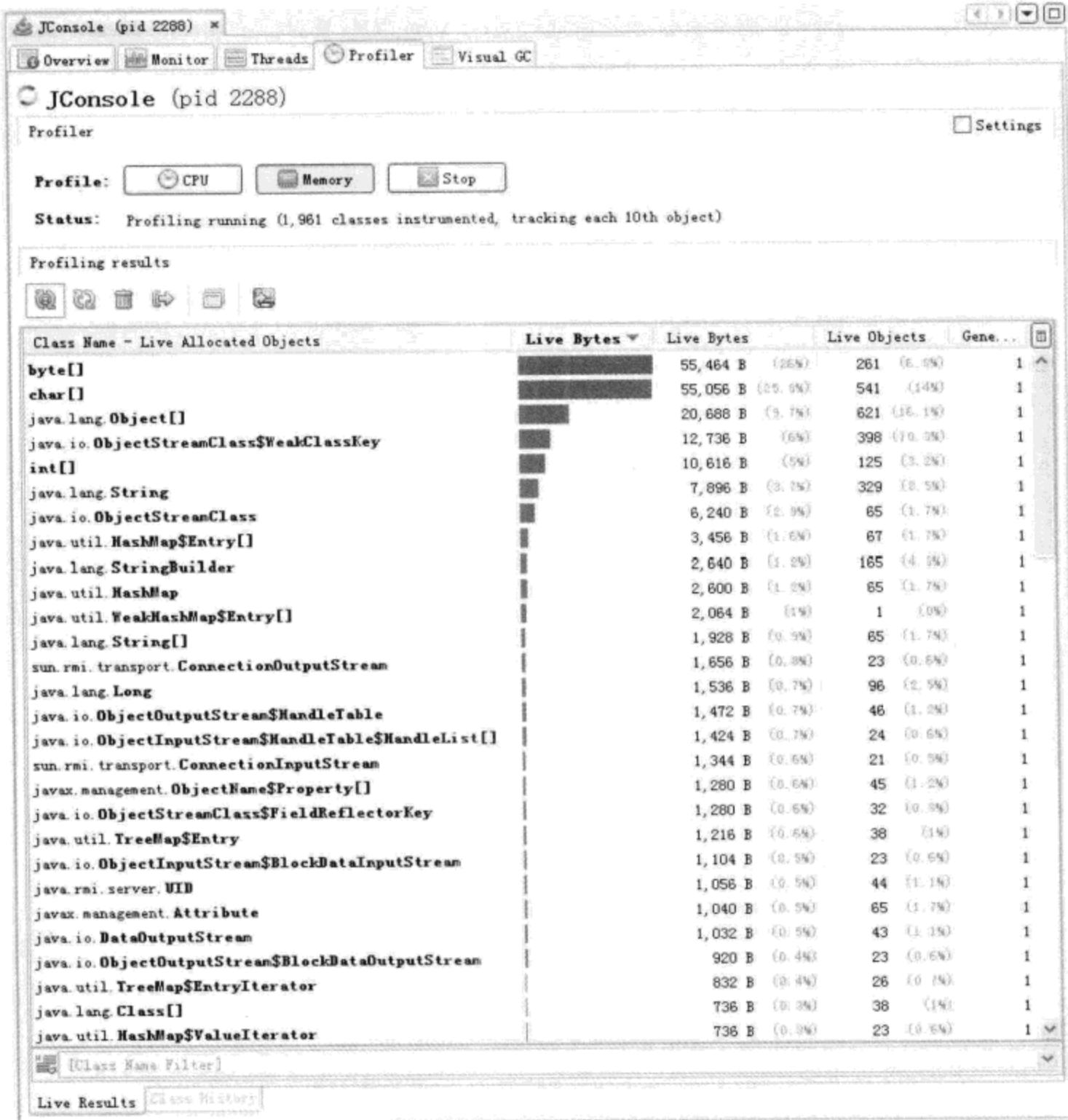


图 3.20 JVisualVM Memory Profiler 图示

### ● 查看 JVM 中各个代的内存状况

在 linux 上执行 jmap -heap [pid]，就可查看整个 JVM 中内存的状况，看到的信息类似如下（和 JDK 版本、GC 策略有关）：

```
using thread-local object allocation.
Parallel GC with 8 thread(s)
Heap Configuration:
  MinHeapFreeRatio = 40
  MaxHeapFreeRatio = 70
```

```

MaxHeapSize      = 1610612736 (1536.0MB)
NewSize          = 524288000 (500.0MB)
MaxNewSize       = 524288000 (500.0MB)
OldSize          = 4194304 (4.0MB)
NewRatio         = 8
SurvivorRatio   = 8
PermSize         = 100663296 (96.0MB)
MaxPermSize     = 268435456 (256.0MB)

Heap Usage:
PS Young Generation
Eden Space:
capacity = 430702592 (410.75MB)
used     = 324439936 (309.4100341796875MB)
free     = 106262656 (101.3399658203125MB)
75.32806675098904% used

From Space:
capacity = 46333952 (44.1875MB)
used     = 13016424 (12.413429260253906MB)
free     = 33317528 (31.774070739746094MB)
28.092626331550566% used

To Space:
capacity = 46792704 (44.625MB)
used     = 0 (0.0MB)
free     = 46792704 (44.625MB)
0.0% used

PS Old Generation
capacity = 1086324736 (1036.0MB)
used     = 945707880 (901.8973159790039MB)
free     = 140616856 (134.1026840209961MB)
87.05572548059884% used

PS Perm Generation
capacity = 100663296 (96.0MB)
used     = 46349592 (44.202415466308594MB)
free     = 54313704 (51.797584533691406MB)
46.044182777404785% used

```

从以上信息中可看出 JVM 堆的配置信息，如 NewSize、NewRatio、SurvivorRatio 等；JVM 堆的使用情况，新生代中的 Eden Space、From Space、To Space 的使用情况，旧生代和持久代的使用情况。

要注意的是在使用 CMS GC 的情况下，jmap -heap 的执行有可能会导致 Java 进程被挂起。

- JVM 中对象的内存的占用情况

在查看 JVM 内存状况时，除了要知道每个代的占用情况外，很多时候更要知道其中各个对象占用的内存大小，这样便于分析对象的内存占用情况，在分析 OutOfMemory 的场景中尤其适用。

输入 jmap -histo [pid]即可查看 jvm 堆中对象的详细占用情况，如图 3.21 所示。

| num | #instances | #bytes    | class name       |
|-----|------------|-----------|------------------|
| 1:  | 2437087    | 501638784 | [C               |
| 2:  | 280698     | 81661880  | [B               |
| 3:  | 2056370    | 49352880  | java.lang.String |

图 3.21 jmap -histo 运行效果

输出内容按照占用空间的大小排序，例如上面的[C，表示 char 类型的对象在 jvm 中总共有 243 707 个实例，占用了 501 638 784 bytes 的空间。

- 导出整个 JVM 中的内存信息

通过以上方法能查看到 JVM 中对象内存的占用情况，但很多时候还要知道这个对象到底是谁创建的。例如上面显示出来的[C，只知道它占用了那么多的空间，但不知道是什么对象创建出的[C，于是 jmap 提供了导出整个 jvm 中的内存信息的支持。基于一些 jvm 内存的分析工具，例如 sun JDK 6 中的 jhat、Eclipse Memory Analyzer，可以分析 jvm 中内存的详细信息，例如[C 是哪些对象创建的。

执行如下命令即可导出整个 jvm 中的内存信息：

```
jmap -dump:format=b,file=文件名 [pid]
```

## 5. JHat

JHat 是 Sun JDK 6 及以上版本中自带的一个用于分析 jvm 堆 dump 文件的工具，基于此工具可分析 jvm heap 中对象的内存占用状况、引用关系等。

执行如下命令分析 jvm 堆的 dump 文件：

```
jhat -J-Xmx1024M [file]
```

执行后等待 console 中输出 Started HTTP server on port 7000，看到后就可以通过浏览器访问 <http://ip:7000> 了，此页面默认为按 package 分类显示系统中所有的对象实例。在页面的最下端有 Other Queries 导航，其中有显示 jvm 中对象实例个数的链接、有显示 jvm 中对象大小的链接等，点击显示 jvm 中对象大小的链接，得到的结果如图 3.22 所示。

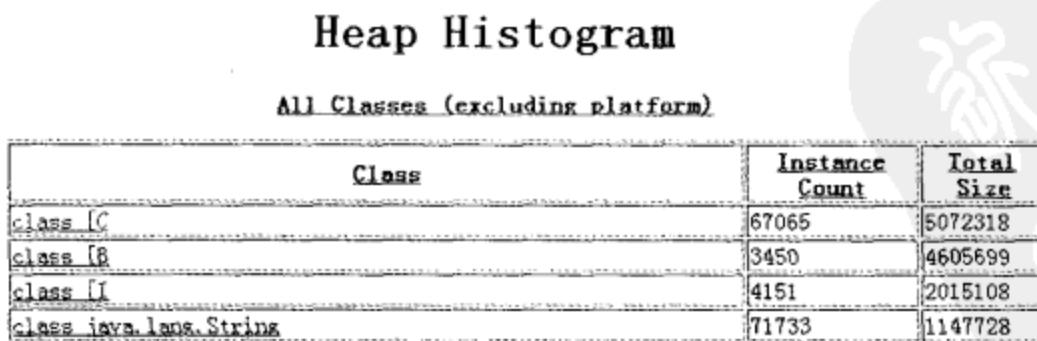


图 3.22 jhat 运行效果

点击图 3.22 中的 class [C，可以看到有哪些对象实例引用了这个对象，或者创建了这个对象，jhat

在分析大的堆 dump 文件时表现不好，速度很慢。

## 6. JStat

JStat 是 Sun JDK 自带的一个统计分析 JVM 运行状况的工具，位于 JDK 的 bin 目录下，除了可用于分析 GC 的状况外，还可用于分析编译的状况、class 加载的状况等。

JStat 用于 GC 分析的参数有：-gc、-gccapacity、-gccause、-gcnew、-gcnewcapacity、-gcold、-gcoldcapacity、-gcpermcapacity、-gcutil。常用的为-gcutil。通过-gcutil 可按一定频率查看 jvm 中各代的空间的占用情况、minor GC 的次数、消耗的时间、full GC 的次数及消耗的时间的统计，执行 jstat -gcutil [pid] [interval]，可看到类似如下的输出信息：

| S0    | S1    | E     | O     | P     | YGC   | YGCT    | FGC | FGCT    | GCT     |
|-------|-------|-------|-------|-------|-------|---------|-----|---------|---------|
| 0.00  | 74.24 | 96.73 | 73.43 | 46.05 | 17808 | 382.335 | 208 | 315.197 | 697.533 |
| 45.37 | 0.00  | 28.12 | 74.97 | 46.05 | 17809 | 382.370 | 208 | 315.197 | 697.568 |

其中 S0、S1 就是 Survivor 空间的使用率，E 表示 Eden 空间的使用率，O 表示旧生代空间的使用率，P 表示持久代的使用率，YGC 表示 minor GC 的执行次数，YGCT 表示 minor GC 执行消耗的时间，FGC 表示 Full GC 的执行次数，FGCT 表示 Full GC 执行消耗的时间，GCT 表示 Minor GC+Full GC 执行消耗的时间。

## 7. Eclipse Memory Analyzer

Eclipse Memory Analyzer 是 Eclipse 提供的一个用于分析 jvm 堆 dump 文件的插件，借助这个插件可查看对象的内存占用状况、引用关系、分析内存泄露等。

Eclipse Memory Analyzer (MAT) 的网站为：<http://www.eclipse.org/mat/>，在 eclipse 中可以远程安装此插件。不过由于此插件在分析堆 dump 文件时比较耗内存，因此在分析前最好先将 eclipse 的 jvm 的内存设置大一点，MAT 分析 dump 文件后的对象占用内存及引用关系如图 3.23 所示。

相对而言 MAT 功能比 jhat 强大很多，分析的速度也快一些，因此，如果要分析 jvm 堆 dump 文件，首选推荐的是 MAT。

在进行 JVM 内存状况分析时，通常要关注的主要有 GC 的趋势、内存的具体消耗状况。

GC 趋势对于可图形界面连到需查看 GC 状况的机器的情况而言，VisualVM 是常用的选择；对于不能采用图形界面方式的，输出 GC 日志<sup>32</sup>及采用 jstat 命令直接分析是常用的选择。

在查找内存是程序中的什么对象占用时，需要分析内存的具体消耗状况，对于有图形界面可用的情况，VisualVM 是常用的选择；对于不能采用图形界面方式的，可通过 jmap dump 生成文件后，再通过 MAT 进行分析是常用的选择。

<sup>32</sup> 可采用 <http://code.google.com/p/gclogviewer/> 来根据 GC 日志生成 GC 趋势图。

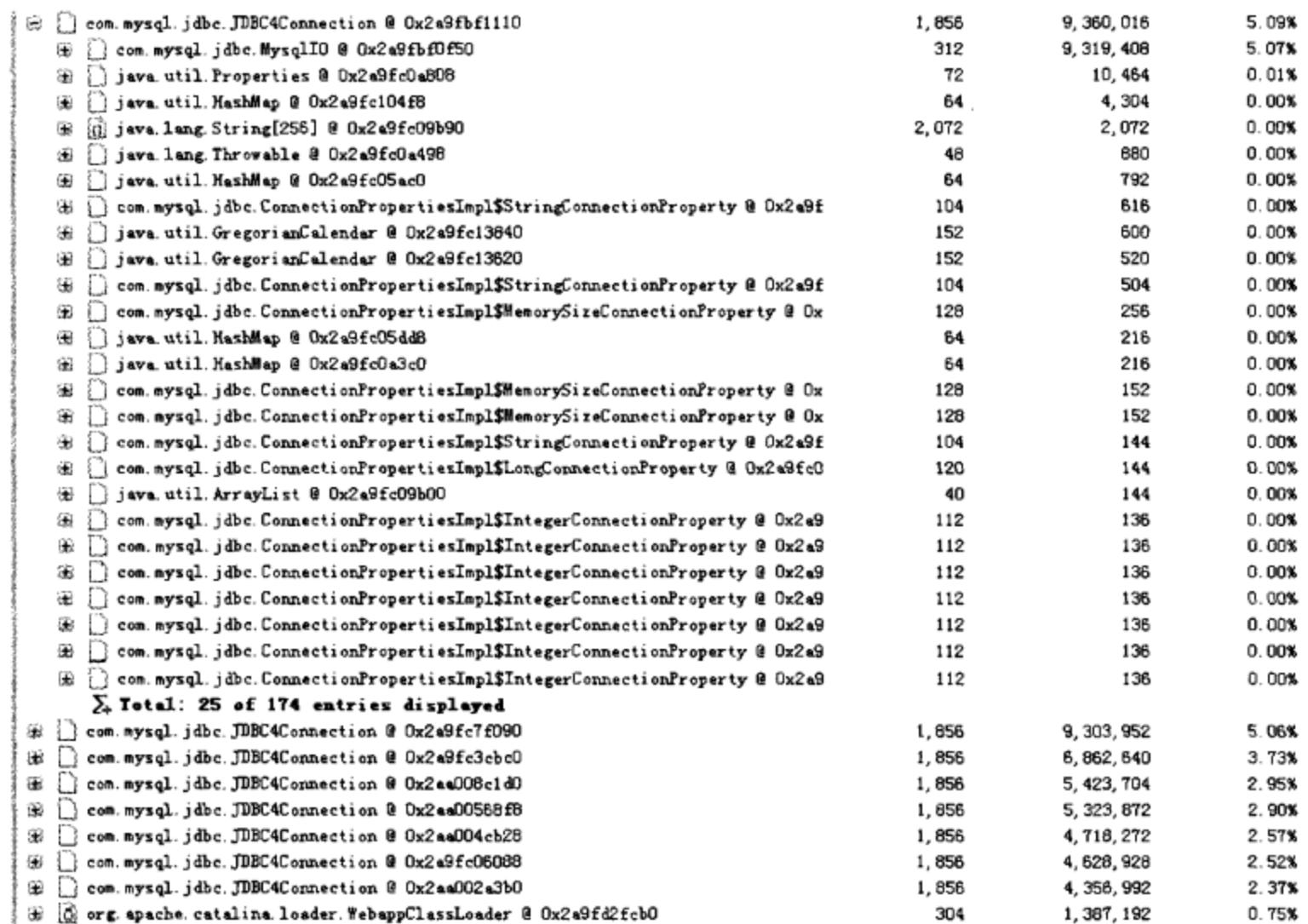


图 3.23 MAT 运行效果

### 3.3 JVM 线程资源同步及交互机制

Java 程序采用多线程的方式来支撑大量的并发请求处理，程序在多线程方式执行的情况下，复杂程度远高于单线程串行执行的程序。尤其是在多核或多 CPU 系统中，多线程执行的程序所带来的最明显的问题是线程之间共同管理的资源的竞争及线程之间的交互。JVM 的线程实现及调度方式（抢占式、协作式）取决于操作系统，超出了本书范围，本节中仅介绍 JVM 线程资源同步机制和线程之间的交互机制。

### 3.3.1 线程资源同步机制<sup>33</sup>

首先来看典型获取下一 ID 的程序：

```
int i=0;  
public int getNextId() {
```

33 <http://www.artima.com/insidejvm/ed2/threadsynch.html>

```

    return i++;
}

```

以下为上面程序在 JVM 中的执行步骤：

- 1) JVM 首先在 main memory (JVM 堆)<sup>34</sup>给 i 分配一个内存存储场所，并存储其值 0；
- 2) 线程启动后，会自动分配一片 working memory 区（通常是操作数栈），当线程执行到 return i++ 时，JVM 中并不是简单的一个步骤就可以完成的。i++动作在 JVM 中分为装载 i、读取 i、进行 i+1 操作、存储 i 及写入 i 五个步骤才得以完成。

- 装载 i

线程发起一个装载 i 的请求给 jvm 线程执行引擎，引擎接收请求后会向 main memory 发起一个 read i 的指令。

当 read i 执行完毕后，一段时间线程会将 i 的值从 main memory 区复制到 working memory 区中。

- 读取 i

此步负责的是从 main memory 中读取 i。

- 进行 i+1 操作

此步由线程完成。

- 存储 i

将 i+1 的值赋给 i，然后存储到 working memory 中。

- 写入 i

一段时间后 i 的值会写回到 main memory 中。

从以上步骤描述来看，这里面最关键的问题有两点：一是 working memory 中 i 值和 main memory 中的 i 值的同步是需要时间的；二是 i++ 由多个操作组成。只要多个线程在这个时间段内同时执行了操作，就会出现获取 i 值相同的现象。举个简单的例子：假设线程 A 已执行 i+1 操作，但尚未执行完写入 i 步骤，线程 B 就完成了装载 i 的过程，那么当线程 B 执行完毕时，其得到的值和 A 就是一样的了，这是以上代码在多线程执行时会出现获取相同值的原因。

JVM 把对于 working memory 的操作分为了 use、assign、load、store、lock 和 unlock，对于 working memory 的操作的指令由线程发出，对于 main memory 的操作分为了 read、write、lock 和 unlock；对于 main memory 的操作的指令由线程执行引擎发出<sup>35</sup>，其含义分别为：

<sup>34</sup> [http://www.javamex.com/tutorials/synchronization\\_concurrency\\_synchronized2.shtml](http://www.javamex.com/tutorials/synchronization_concurrency_synchronized2.shtml)

<sup>35</sup> [http://java.sun.com/docs/books/jvms/second\\_edition/html/Threads.doc.html](http://java.sun.com/docs/books/jvms/second_edition/html/Threads.doc.html)

- use

use 由线程发起，需要完成将变量的值从 working memory 中复制到线程执行引擎中。

- assign

assign 由线程发起，需要完成将变量值复制到线程的 working memory 中，例如 `a=i`，这时线程就会发起一个 assign 动作。

- load

load 由线程发起，需要完成将 main memory 中 read 到的值复制到 working memory 中。

- store

store 由线程发起，负责将变量的值从 working memory 中复制到 main memory 中，并等待 main memory 通过 write 动作写入此值。

- read

read 由 main memory 发起，负责从 main memory 中读取变量的值。

- write

write 由 main memory 发起，负责将 working memory 的值写入到 main memory 中。

- lock

lock 动作由线程发起，同步操作 main memory，给对象加上锁。

- unlock

unlock 动作由线程发起，同步操作 main memory，去除对象的锁。

JVM 中保证以下操作是顺序的：

- 1) 同一个线程上的操作一定是顺序执行的；
- 2) 对于 main memory 上的同一个变量的操作一定是顺序执行的，也就是不可能两个请求同时读取变量值；
- 3) 对于加了锁的 main memory 上的对象操作，一定是顺序执行的，也就是两个以上加了 lock 的操作，同时肯定只有一个是在执行的。

为了避免资源操作的脏数据问题，JVM 提供了 `synchronized` 关键字、`volatile` 关键字和 lock/unlock 机制。

采用 `synchronized` 关键字改造上面的代码为：

```
public synchronized int getNextId() {
    return i++;
}
```

当多线程执行此段代码时，线程 A 执行到 `getNextId()` 方法，JVM 知道该方法上有 `synchronized` 关键字，于是在执行其他动作前首先按照对象的实例 ID 加上一个 lock。然后再继续执行 `return i++`，而此时如线程 B 并发访问 `getNextId()` 方法，JVM 观察到这个对象的实例 ID 上有 lock，于是将线程 B 放入等待执行的队列中，只有当线程 A 的 `return i++` 执行完毕后，JVM 才会释放对象实例 ID 上的 lock，重新标记为 unlock。这时当下次线程调度到线程 B 时，线程 B 才得以执行 `getNextId()` 方法，由于这个过程是串行的，因此可以保证每个线程 `getNextId` 都是不一样的值。

`synchronized` 除了可直接写在方法上外，也可直接定义在对象上，区别仅在于 JVM 会根据这些情况来决定 lock 标记是打在什么上。需要注意的是，如果 `synchronized` 是标记在一个 `static` 方法上，那么执行时锁的粒度为当前 Class。

`lock/unlock` 机制的原理和 `synchronized` 相同，`synchronized`、`lock/unlock` 机制都可用于保证某段代码执行的原子性，由于锁会阻塞其他线程同样需要锁的部分的执行，因此在使用锁机制时要避免死锁的产生，尤其是在多把锁的情况下，例如：

```
private Object a=new Object();
private Object b=new Object();
public void callAB(){
    synchronized(b){
        synchronized(a){
            // Do something;
        }
    }
}

public void executeAB(){
    synchronized(a){
        synchronized(b){
            // Do something;
        }
    }
}
```

当上面的 `callAB` 和 `executeAB` 被两个线程同时执行时，就会产生死锁的现象，导致系统挂起。这是一个使用 `synchronized` 的例子，对于直接使用 `lock/unlock` 来编写的多线程程序而言，一定要保证 lock 和 unlock 是成对出现的，并且要保证 lock 后程序执行完毕时一定要 unlock，否则就有线程会出现锁饿死的现象。

`volatile` 的机制有所不同，它仅用于控制线程中对象的可见性，但并不能保证在此对象上操作的原子性。就像上面 `i++` 的场景，即使把 `i` 定义为 `volatile` 也是没用的，对于定义为 `volatile` 的变量，线程不会将其从 `main memory` 复制到 `work memory` 中，而是直接在 `main memory` 中进行操作。它的代价虽然比 `synchronized`、`lock/unlock` 低，但用起来要非常小心，毕竟它不能保证操作的原子性。

有了 `synchronized`、`lock/unlock` 及 `volatile` 后，在 Java 中就可以很好地控制多线程程序中资源的竞争。但由于这三个机制对于系统的性能都是有影响的，例如会将操作由并行化变为串行化，因此需要根据合理的需求对线程中的资源进行同步。

### 3.3.2 线程交互机制

线程之间除了会产生资源的竞争外，还会有交互的需求。例如最典型的连接池，连接池中通常都会有 `get` 和 `return` 两个方法，`return` 的时候需要将连接返回到缓存列表中，并将可使用的连接数加 1，而 `get` 方法在判断可使用的连接数已经到了 0 后，需要进入一个等待状态，当有连接返回到连接池时，应该通知下 `get` 方法，不需要再等待了，如果没有这个交互机制，就只能在 `get` 方法中不断轮循判断可使用的连接数的值了。JVM 提供 `wait/notify/notifyAll` 方式来支持这类需求，在基于 `Object` 的 `wait/notify/notifyAll` 实现连接池时，典型的代码如下：

```
public Connection get() {
    synchronized(this) {
        if(free>0) {
            free--;
            return cacheConnections.poll();
        }
        else{
            this.wait();
        }
    }
}
public void close(Connection conn) {
    synchronized(this){
        free++;
        cacheConnections.offer(conn);
        this.notifyAll();
    }
}
```

调用 `Object` 的 `wait` 方法可以让当前线程进入等待状态，只有当其他线程调用了此 `Object` 的 `notify` 或 `notifyAll` 方法，或者 `wait`（毫秒数）到达了指定的时间后，才会被激活继续执行，`notify` 只是随机找 `wait` 此 `Object` 的一个线程，而 `notifyAll` 则是通知 `wait` 此 `Object` 的所有线程。在 Sun JDK 中，`object.wait` 还有可能被假唤醒，因此通过在 `object.wait` 被唤醒后，应再次确认需要等待的状态是否变更了。如果未变更，则继续进入 `wait` 状态，这种做法通称为 `double check`，一段示例代码如下：

```
// 避免 object.wait 被假唤醒
while(!task.isFinished()){
    synchronized(task){
```

```

    task.wait();
}
}

```

当线程调用了对象的 wait 方法后, JVM 线程执行引擎会将此线程放入一个 wait sets 中, 并释放此对象的锁, 在 wait sets 中的线程将不会被 JVM 线程执行引擎调度执行; 当其他线程调用了此对象的 notify 方法时, 会从 wait sets 中随机找一个等待在此对象上的线程, 并将其从 wait sets 中删除, 这样 JVM 线程执行引擎就可以再次调度执行此线程了; 当调用的为 notifyAll 方法时, 则会删除 wait sets 上所有等待在此对象上的线程, 删除完毕后释放对象锁。

在 Sun JDK 5.0 的版本中, 增加了并发包, 其中提供了更多的方式来支持线程间的交互, 例如 Semaphore 的 acquire 和 release、Condition 的 await 和 signal、CountDownLatch 的 await 和 countDown 等。

### 3.3.3 线程状态及分析

JVM 把线程分为几种不同的状态, 并根据状态放入不同的 sets 中来进行调度。线程在创建完毕后进入 new 状态, 调用了线程的 start 方法后线程进入 Runnable 状态, 放入 JVM 的可运行线程队列中, 等待获取 CPU 的执行权; JVM 按线程优先级及时间分片、轮循的方式来执行 Runnable 状态的线程。当线程进入 start 代码段, 开始执行时, 其线程状态转变为 Running; 线程在执行过程中如果执行了 sleep、wait、join, 或者进入了 IO 阻塞、锁等待时, 则进入 Wait 或 Blocked 状态, 在这种状况下线程放弃 CPU 的使用权, 进入 wait sets 或锁 sets 中, 直到 wait 结束、线程被唤醒或获取到锁, 在这些情况下线程也再次进入 Runnable 状态; 在线程执行完毕后, 线程就从可运行线程队列中删除了, JVM 线程的状态转变如图 3.24 所示。

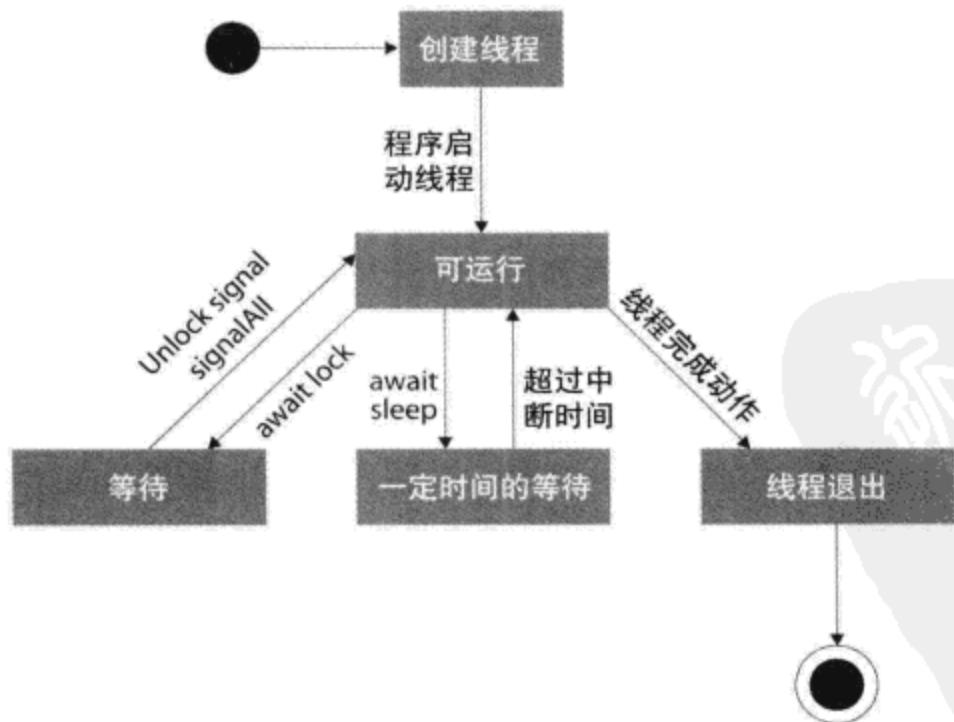


图 3.24 JVM 线程状态转换

为了跟踪运行时 JVM 中线程的状况, Sun JDK 及开源界提供了一些不错的工具。对于运行时判断什么操作是耗系统资源的, 哪些操作出现了锁竞争激烈或死锁现象等有很大的帮助。

- kill -3 [pid]

在 linux 上, 找出 java 所在的进程号, 然后执行 kill -3 [pid], 线程的相关信息就输出到 Console 上了。

- jstack

jstack 是 Sun JDK 中自带的工具, 通过该工具可以直接看到 jvm 中线程的运行状况, 包括锁的等待、线程是否在运行等。

执行 jstack [pid], 线程的所有堆栈信息即输出了, 类似如下:

```
"http-8080-10" daemon prio=10 tid=0xa949bb60 nid=0x884 waiting
for monitor entry [a765c000..a765e878]
at java.sql.DriverManager.getConnection(DriverManager.java:187)
- waiting to lock <0xaf5bcf10> (a java.lang.Class)
```

此行表示 http-8080-10 这个线程处于等锁的状态, waiting for monitor entry 如果在连续几次输出线程堆栈信息都存在于同一个或多个线程上时, 则说明系统中有锁竞争激烈、死锁或锁饿死的现象。

```
"http-8080-6" daemon prio=10 tid=0x00000002b031c6800 nid=0x565d in Object.wait()
[0x0000000041b43000..0x0000000041b43c30]
java.lang.Thread.State: WAITING (on object monitor)
```

该行表示 http-8080-6 的线程处于对象的 wait 上, 等待其他线程的唤醒, 这也是线程池的常见用法。

```
"Low Memory Detector" daemon prio=10 tid=0x00000002afff4c400 nid=0x5653 runnable
[0x0000000000000000..0x0000000000000000]
java.lang.Thread.State: RUNNABLE
```

该行表示 Low Memory Detector 的线程处于 runnable 状态, 等待获取 CPU 的使用权。

- JConsole

JConsole 是 SUN JDK 中自带的一个工具, 位于 JDK 的 bin 目录下, 其可用于图形化的跟踪查看运行时系统中线程的状况 (运行状态、锁等待、堆栈、检测死锁等), 效果如图 3.25 所示。

JConsole 的 demo 中还附带了一个 JTop 的程序, 由于可通过它更为形象地跟踪系统中线程消耗的 CPU 的时间, 因此, 在运行 JConsole 时通过指定 -pluginpath [JTop.jar] (默认情况下, JTop.jar 位于 demo\management\JTop 下)。打开 JConsole 后, 在 Tab 页上就会增加一个 JTop 的 tab, 效果如图 3.26 所示。

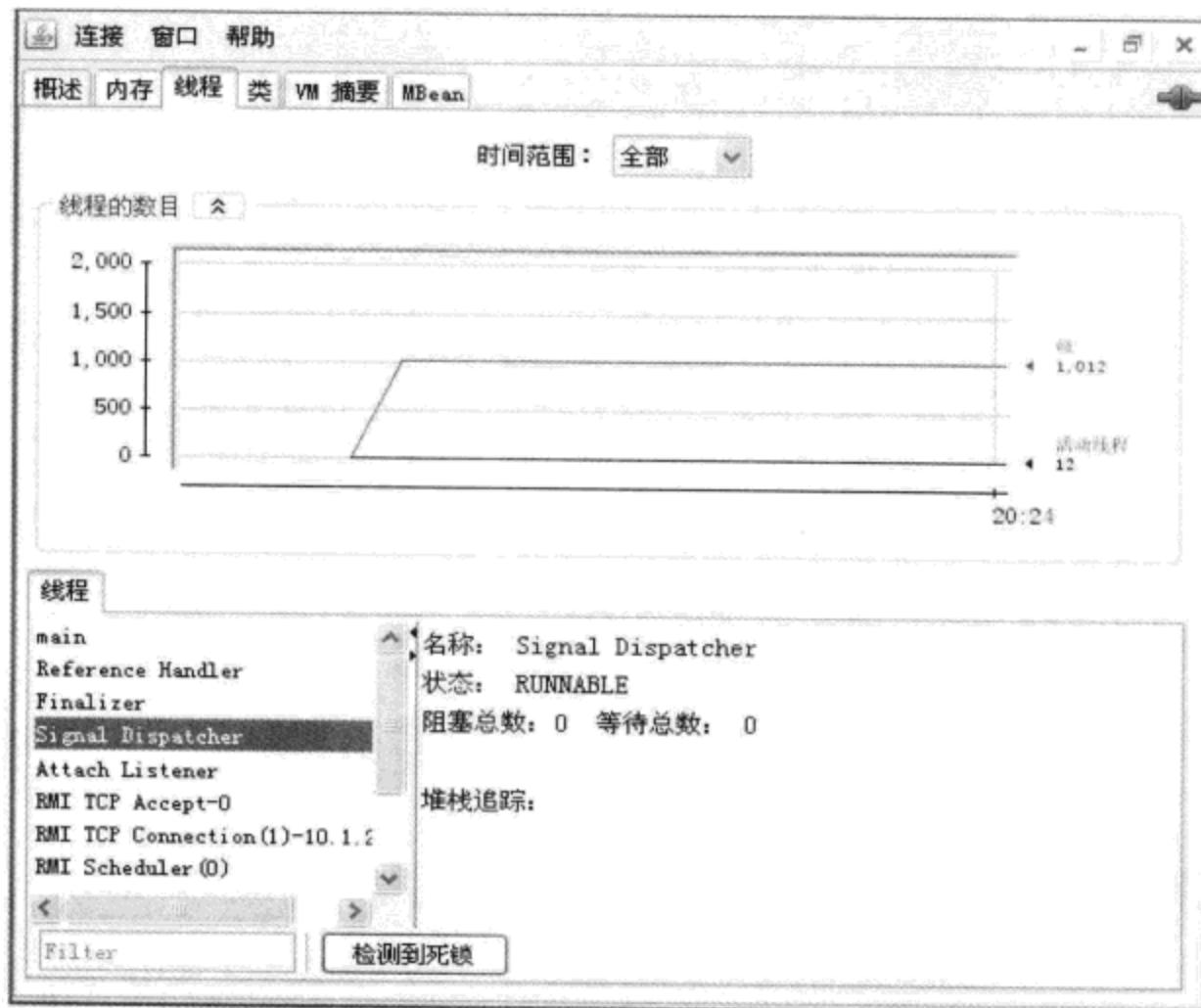


图 3.25 JConsole 查看线程状态

| ThreadName                       | CPU(sec) | State         |
|----------------------------------|----------|---------------|
| main                             | 25.0000  | RUNNABLE      |
| RMI TCP Connection(3)-10.1.26.53 | 0.0000   | RUNNABLE      |
| Attach Listener                  | 0.0000   | RUNNABLE      |
| RMI TCP Connection(2)-10.1.26.53 | 0.0000   | RUNNABLE      |
| RMI TCP Connection(1)-10.1.26.53 | 0.0000   | TIMED_WAITING |
| RMI TCP Accept-0                 | 0.0000   | RUNNABLE      |
| Reference Handler                | 0.0000   | WAITING       |

图 3.26 JTop 查看消耗 CPU 的线程

- ThreadXMBean

ThreadXMBean 是 Sun JDK 中自带的一个可直接访问的 MBean，该 MBean 的名称为 `java.lang.Threading`，通过 JConsole 访问此 MBean 可看到如图 3.27 所示的信息。

也可以编写程序直接访问此 MBean，通过该 MBean 可以获取 jvm 中所有线程、线程的运行状态、每个线程耗费的 CPU 时间、处于等待 Object 锁的线程数等。

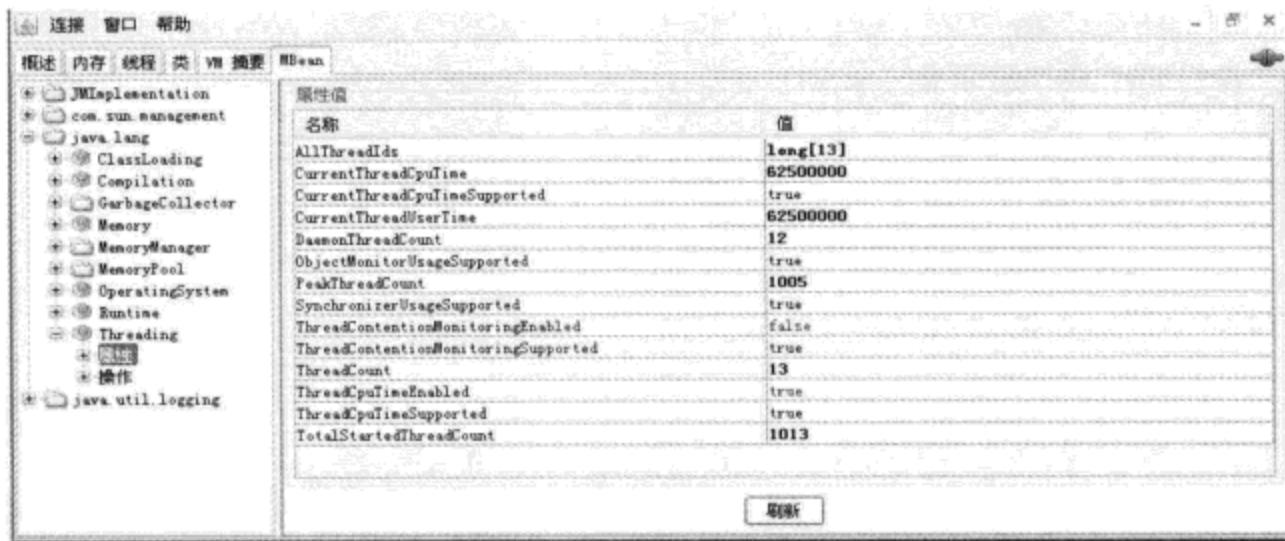


图 3.27 ThreadXMBean 查看线程状态

### ● TDA

TDA 是开源界一个不错的用于分析线程堆栈信息的图形化工具，其网站为：<http://tda.dev.java.net/>，执行 TDA 后，直接将包含线程堆栈信息的文件导入即可，效果如图 3.28 所示。

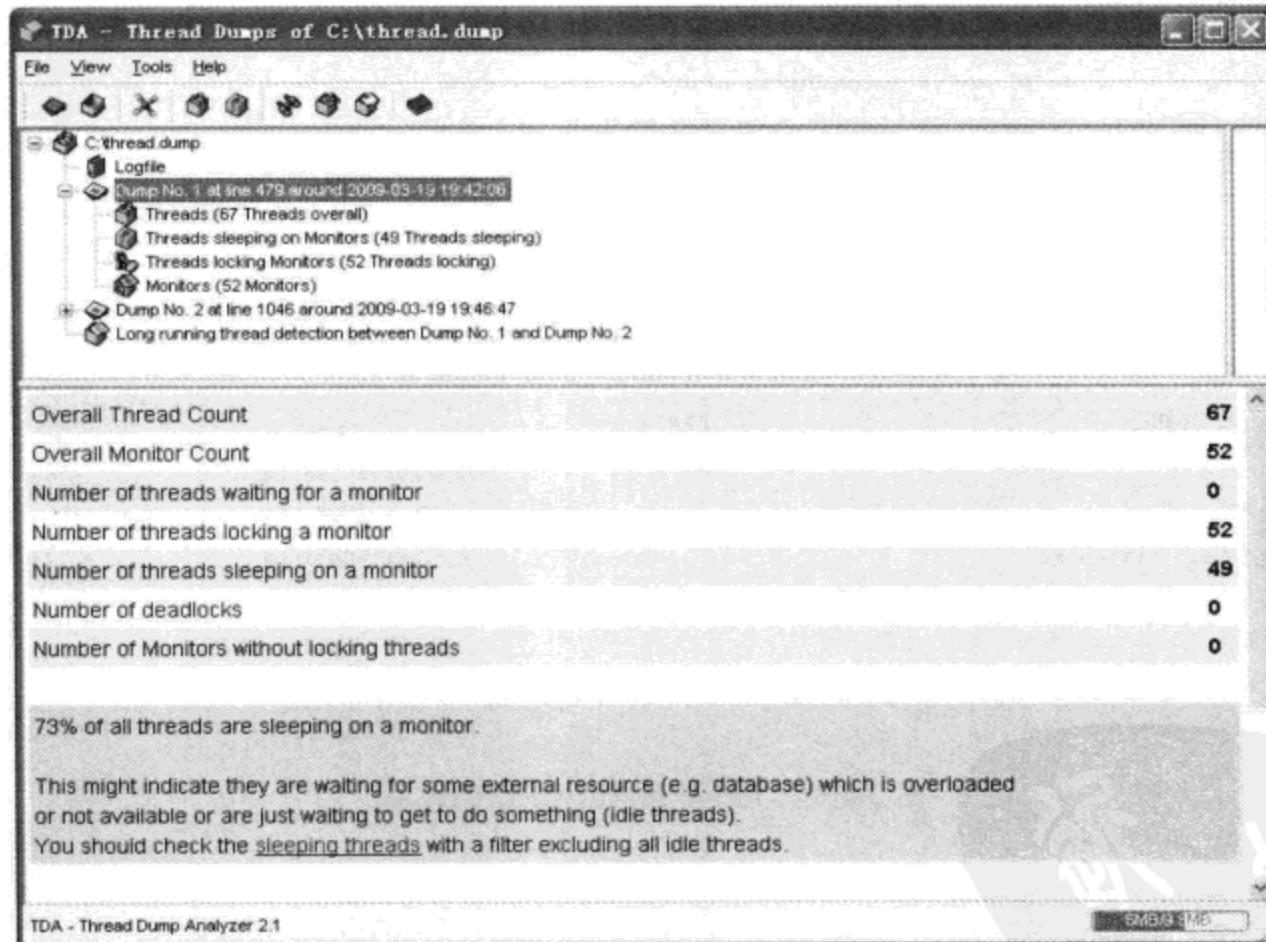


图 3.28 TDA 查看线程状态

通过此工具可以比较方便地分析线程堆栈中等待锁的线程、各种状态的线程及运行时间较长的线程等。

- TOP 命令 + ThreadDump

通过以上几种方式都可以 dump 出 JVM 中的线程信息，而结合 linux 中的 TOP 命令，可以更轻松地分析目前线程消耗 CPU 的状况，包括消耗的 CPU 总时间及实时 CPU 消耗比率，方法如下。

在 linux 上输入 TOP，在进入 TOP 后按[shift]+[H]组合键，再按线程的方式查看线程 ID、CPU 的消耗状况等。通过这种方式获取耗 CPU 的线程后，结合 ThreadDump 信息具体查看消耗 CPU 的线程在做的动作。这里要注意的是，TOP 中显示的是十进制的线程 ID，而 ThreadDump 中的 nid（即 Native ID）对应的为十六进制，因此在查找时需要先做次转换。

除了上面提及的这些工具外，在业界还有不少商业的可用于分析 Java 程序的线程状况的工具，例如大名鼎鼎的 JProfiler。

以上以 Sun Hotspot 为例对 JVM 中代码的执行、自动内存管理以及多线程支持的机制进行了介绍，掌握这些实现机制对于实现高性能、高并发的 Java 程序而言至关重要。



# 第4章 分布式 Java 应用与 Sun JDK 类库



JDK 是 Java 规范的实现，Java 程序运行在各家厂商的 JDK 上，其实现方式一定程度上决定了程序的运行性能表现，各家厂商（例如 Oracle、IBM、HP 等）在实现 Java 规范时采用的方法并不一定相同。在编写 Java 程序时，由于 JDK 提供了众多实现了同样接口的类，如何选择合适的类来实现需求就成了难题。这也是要深入理解 JDK 的原因，以做到根据需求来选择合适的类，而不是想当然地认为调用某个类的某个方法时就会达到预期的效果，否则很有可能因为对 JDK 的实现不了解而导致最终程序运行的效果和预期不一致。

Sun JDK 是目前使用最广泛的 JDK，本章基于 JDK 6 Update 12 的代码对 Sun JDK 常用 package 中的常用类分析，分析其中方法的实现方式，并评估类中的常用方法在不同场景下的性能表现，在掌握了这些知识点后，对于根据需求合理地选择类会有一定的帮助。JDK 常用 package 中的常用类进行分析，分析的方法为掌握常用类中方法的实现原理，并评估类中的常用方法在不同场景下的性能表现，在掌握了这些知识点后，对根据需求合理地选择类会有一定的帮助。

JDK 分为几个大的包，对于构建分布式 Java 应用而言，最重要的有集合、并发、网络（包括网络 BIO 以及网络 NIO）以及序列化/反序列化。其中网络包遵循通信协议和操作系统实现方式而实现，因此其表现更多地取决于通信协议和操作系统的实现方式以及程序的使用方法（例如连接池、长连接等），本章中将对除网络之外的包进行分析。

本章中的代码测试时的硬件环境为：6 核 Intel(R) Xeon(R) CPU E5530@ 2.40GHz，4GB 物理内存，软件环境为：32 位 Linux 2.6.18，Sun JDK 1.6.0 update 12。

## 4.1 集合包

集合包是 Java 中最常用的包，它最常用的有 Collection 和 Map 两个接口的实现类，Collection 用于存放多个单对象，Map 用于存放 Key-Value 形式的键值对。

Collection 中常用的又分为两种类型的接口：List 和 Set，两者最明显的差别为 List 支持放入重复的对象，而 Set 不支持。List 接口常用的实现类有：ArrayList、LinkedList、Vector 及 Stack；Set 接口常用的实现类有：HashSet、TreeSet，对于 Collection 的实现类而言，要重点掌握的为以下几点：

- Collection 的创建

对应的为 Collection 实现类的构造器，需要掌握在构造器方法中 Collection 的实现类都做了些什么。

- 往 Collection 中增加对象

对应的为 Collection 中的 add(E)方法，往 Collection 中增加对象时 Collection 的实现方式决定了此方法的性能。

- 删 除 Collection 中的对象

对应的为 Collection 中的 remove(E)方法，实现类的实现方式决定了此方法的性能。

- 获取 Collection 中的单个对象

对应的为 Collection 中的 `get(int index)` 方法，实现类的实现方式决定了此方法的性能。

- 遍历 Collection 中的对象

对应的是通过 Collection 的 `iterator` 方法获取迭代器，进而遍历。

- 判断对象是否存在于 Collection 中

对应的是 Collection 中的 `contains(E)` 方法，实现类的实现方式决定了此方法的性能。

- Collection 中对象的排序

如何对 Collection 中对象合理地排序也是使用 Collection 对象时经常要考虑的问题，但由于排序主要取决于所采取的排序算法，在此就不多讲解了。

按照这几点来分析下常用的 List 和 Set 实现类的实现方式。

### 4.1.1 ArrayList

#### 实现方式

对应上述要掌握的几点，来看看 ArrayList 的实现方式。

##### 创建：ArrayList()

此默认构造器通过调用 `ArrayList(int)` 来完成 ArrayList 的创建，传入的值为 10，`ArrayList(int)` 方法的实现代码为：

```
super();
if (initialCapacity < 0)
    throw new IllegalArgumentException("Illegal Capacity: "+
                                       initialCapacity);
this.elementData = new Object[initialCapacity];
```

`super()` 调用的为 `AbstractList` 的默认构造器方法，该方法为一个空方法，因此这段代码中最关键的是实例化了一个 `Object` 的数组，并将此数组赋给了当前实例的 `elementData` 属性，此 `Object` 数组的大小即为传入的 `initialCapacity` 的值，因此在调用空构造器的情况下会创建一个大小为 10 的 `Object` 数组。据此也可看出，`ArrayList` 采用的是数组的方式来存放对象。`super()` 调用的是 `AbstractList` 的默认构造器方法，该方法为一个空方法，因此这段代码中最关键的是实例化了一个 `Object` 的数组，并将此数组赋给了当前实例的 `elementData` 属性，此 `Object` 数组的大小即为传入的 `initialCapacity` 的值，因此在调用空构造器的情况下会创建一个大小为 10 的 `Object` 数组。据此也可看出，`ArrayList` 采用的是数组的方式来存放对象。

### 插入对象：add(E)

`add`方法简单来看就是将数组中某元素的值赋值为传入的对象，但在`add`时有个很明显的问题是：如果此时数组满了，该怎么办？带着这个问题，来看看`ArrayList`的实现方式。

当调用`ArrayList`的`add`方法时，首先基于`ArrayList`中已有的元素数量加1，产生一个名为`minCapacity`的变量，然后比较此值和`Object`数组的大小，如果此值大于`Object`数组值，那么先将当前的`Object`数组值赋给一个数组对象，接着产生一个新的数组的容量值。此值的计算方法为当前数组值 $\times 1.5 + 1$ ，如得出的容量值仍然小于`minCapacity`，那么就以`minCapacity`作为新的容量值，在得出这个容量值后，调用`Arrays.copyOf`来生成新的数组对象，如想调整容量的增长策略，可继承`ArrayList`，并覆盖`ensureCapacity`方法。

`Arrays.copyOf`的实现方法简单来说，首先是创建一个新的数组对象，该数组对象的类型和之前`ArrayList`中元素的类型是一致的，在这里JDK做了个小优化。如果是`Object`类型，则直接通过`new Object[newLength]`的方式来创建。如不是`Object`类型，则通过`Array.newInstance`调用native方法来创建相应类型的数组；在创建完新的数组对象后，调用`System.arraycopy`通过native方法将之前数组中的对象复制到新的数组中。

在确保有足够的空间放入新的元素后，可将数组的末尾赋值为传入的对象，例如目前数组值为10，已经有5个元素了，那么新加入的元素就自动放在数组6的位置。

在`Collection`中增加对象时，`ArrayList`还提供了`add(int,E)`这样的方法，允许将元素直接插入指定的int位置上，这个方法的实现首先要确保插入的位置是目前的`Array`数组中存在的，之后还要确保数组的容量是够用的。在完成了这些动作后，和`add(E)`不同的地方就出现了，它要将当前的数组对象进行一次复制，即将目前`index`及其后的数据都往后挪动一位，然后才能将指定的`index`位置的赋值为传入的对象，可见这种方式要多付出一次复制数组的代价。

除了`add(int,E)`这种方法可将对象插入指定的位置外，`ArrayList`还提供了`set(int,E)`这样的方法来替换指定位置上的对象。

为了方便开发人员的使用，`ArrayList`还对外提供了`addAll(Collection<? extends E>)`及`addAll(int,Collection<? extends E>)`这样的方法，其实现方式和`add(E)`、`add(int,E)`基本类似。

### 删除对象：remove(E)

`remove`对于集合的性能而言也非常重要，当执行此方法时，`ArrayList`首先判断对象是否为`null`，如为`null`，则遍历数组中已有值的元素，并比较其是否为`null`，如为`null`，则调用`fastRemove`来删除相应位置的对象。`fastRemove`方法的实现方式为将`index`后的对象往前复制一位，并将数组中的最后一个元素的值设置为`null`，即释放了对此对象的引用；如对象非`null`，唯一的不同在于通过`E`的`equals`来比较元素的值是否相同，如相同则认为是需要删除对象的位置，然后同样是调用`fastRemove`来完成对象的删除。

`ArrayList` 中还另外提供了 `remove(int)` 这样的方法来删除指定位置的对象，`remove(int)` 的实现比 `remove(E)` 多了一个数组范围的检测，但少了对象位置的查找，因此性能会更好。

### 获取单个对象：`get(int)`

`get` 传入的参数为数组元素的位置，因此 `ArrayList` 仅须先做数组范围的检测，然后即可直接返回数组中位于此位置的对象。

### 遍历对象：`iterator()`

`iterator` 由 `ArrayList` 的父类 `AbstractList` 实现，当每次调用 `iterator` 方法时，都会创建一个新的 `AbstractList` 内部类 `Itr` 的实例。当调用此实例的 `hasNext` 方法时，比较当前指向的数组的位置是否和数组中已有的元素大小相等，如相等则返回 `false`，否则返回 `true`。

当调用实例的 `next` 方法时，首先比较在创建此 `Iterator` 时获取的 `modCount` 与目前的 `modCount`，如果这两个 `modCount` 不相等，则说明在获取 `next` 元素时，发生了对于集合大小产生影响（新增、删除）的动作。当发生这种情况时，则抛出 `ConcurrentModificationException`。如果 `modCount` 相等，则调用 `get` 方法来获取相应位置的元素，当 `get` 获取不到时抛出 `IndexOutOfBoundsException`，在捕捉到 `IndexOutOfBoundsException` 后，检测 `modCount`，如 `modCount` 相等，则抛出 `NoSuchElementException`，如不相等，则抛出 `ConcurrentModificationException`。

### 判断对象是否存在：`contains(E)`

为了判断 `E` 在 `ArrayList` 中是否存在，做法为遍历整个 `ArrayList` 中已有的元素，如 `E` 为 `null`，则直接判断已有元素是否为 `null`，如为 `null`，则返回 `true`；如 `E` 不为 `null`，则通过判断 `E.equals` 和元素是否相等，如相等则返回 `true`。

`indexOf` 和 `lastIndexOf` 是 `ArrayList` 中用于获取对象所在位置的方法，其中 `indexOf` 为从前往后寻找，而 `lastIndexOf` 为从后向前寻找。

## 注意要点

对于 `ArrayList` 而言，最须注意的有以下几点：

- `ArrayList` 基于数组方式实现，无容量的限制；

• `ArrayList` 在执行插入元素时可能要扩容，在删除元素时并不会减小数组的容量（如希望相应的缩小数组容量，可以调用 `ArrayList` 的 `trimToSize()`），在查找元素时要遍历数组，对于非 `null` 的元素采取 `equals` 的方式寻找；

- `ArrayList` 是非线程安全的。

## 4.1.2 LinkedList

### 实现方式

LinkedList 也是常用的一种 List 实现，LinkedList 基于双向链表机制，所谓双向链表<sup>1</sup>机制，就是集合中的每个元素都知道其前一个元素及其后一个元素的位置。在 LinkedList 中，以一个内部的 Entry 类来代表集合中的元素，元素的值赋给 element 属性，Entry 中的 next 属性指向元素的后一个元素，Entry 中的 previous 属性指向元素的前一个元素，基于这样的机制可以快速实现集合中元素的移动。LinkedList 的具体实现方式如下：

#### LinkedList()

在创建 LinkedList 对象时，应首先创建一个 element 属性为 null、next 属性为 null 及 previous 属性为 null 的 Entry 对象，并赋值给全局的 header 属性。

在执行构造器时，LinkedList 将 header 的 next 及 previous 都指向 header，以形成双向链表所需的闭环。

#### add(E)

当向 LinkedList 中增加元素时，要做的就是创建一个 Entry 对象，并将此 Entry 对象的 next 指向 header，previous 指向 header.previous。在完成自己的 next、previous 的设置后，同时将位于当前元素的后一元素的 previous 指向自己，并将位于当前元素的前一元素的 next 指向自己，这样就保持了双向链表的闭环。

LinkedList 的 add 方法不用像 ArrayList 那样，要考虑扩容及复制数组的问题，但它每增加一个元素，都要创建一个新的 Entry 对象，并要修改相邻的两个元素的属性。

#### remove(E)

要删除 LinkedList 的一个元素，首先同样要遍历整个 LinkedList 中的元素，遍历和寻找匹配的元素的方法和 ArrayList 基本相同，寻找到相匹配的元素后，删除元素的方法比 ArrayList 简单很多。

删除时只须直接删除链表上的当前元素，并将当前元素中的 element、previous 及 next 属性设置为 null，即可完成对象的删除。

这个动作比 ArrayList 删除元素简单很多，毕竟 ArrayList 还要将当前元素所在的位置后的元素通过复制往前移动一位。

#### get(int)

由于 LinkedList 的元素并没有存储在一个数组中，因此其 get 操作过程比 ArrayList 更为复杂，在执行 get 操作时，首先要判断传入的 index 值是否小于 0 或者大于/等于当前 LinkedList 的 size 值。如符

<sup>1</sup> <http://zh.wikipedia.org/zh/双向链表>

合这两个条件之一，则抛出 `IndexOutOfBoundsException`，如不符合，则进行下面的步骤。

首先判断当前要获取的位置是否小于 `LinkedList` 值的一半，如小于，则从头一直找到 `index` 位置所对应的 `next` 元素；如大于，则从队列的尾部往前，一直找到 `index` 位置所对应的 `previous` 元素。

### iterator()

`iterator` 方法由父类 `AbstractList` 实现，当调用 `iterator` 方法时，每次都会创建一个 `ListItr` 对象，创建时该对象负责保存 `cursor` 位置。

当调用 `iterator` 返回的遍历对象的 `hasNext` 方法时，判断当前 `cursor` 的位置是否等于 `LinkedList` 的 `size` 变量，如等于则返回 `true`，不等于则返回 `false`。

当调用 `iterator` 返回的遍历对象的 `next` 方法时，调用 `get` 方法实现，传入 `cursor` 位置，即可获取到相应的元素对象。如果在遍历过程中，`LinkedList` 中的元素增加或删除，则会抛出 `ConcurrentModificationException`。

由于 `LinkedList` 是基于双向链接实现的，因此其在遍历时还可往前遍历，通过调用 `hasPrevious` 和 `previous` 来完成遍历过程。

### contains(E)

为了判断 `E` 是否存在于 `LinkedList` 中，`LinkedList` 采用的方法是遍历所有元素。如传入的 `E` 为 `null`，则判断元素是否为 `null`，如找到为 `null` 的元素，则返回 `true`；如传入的 `E` 非 `null`，则判断 `E` 是否有 `equals` 元素，如找到 `equals` 的元素，则返回 `true`，如遍历完仍未找到匹配的元素，则返回 `false`。

## 注意要点

对 `LinkedList` 而言，最要注意以下几点：

- `LinkedList` 基于双向链表机制实现；
- `LinkedList` 在插入元素时，须创建一个新的 `Entry` 对象，并切换相应元素的前后元素的引用；在查找元素时，须遍历链表；在删除元素时，要遍历链表，找到要删除的元素，然后从链表上将此元素删除即可；
- `LinkedList` 是非线程安全的。

### 4.1.3 Vector

## 实现方式

`Vector` 是从 JDK 1.2 就已提供的 `List` 实现，`Vector` 和 `ArrayList` 一样，也是基于 `Object` 数组的方式来实现的，其具体实现如下。

**Vector()**

默认创建一个大小为 10 的 Object 数组，并将 capacityIncrement 设置为 0。

**add(E)**

Vector 中的 add 方法是加了 synchronized 关键字的，因此此方法是线程安全的。除此之外，它和 ArrayList 基本相同，不同点是当数组大小不够时，其擴大数组的方法有所不同，Vector 的策略为：

如果 capacityIncrement 大于 0，则将 Object 数组的大小擴大为现有 size 加上 capacityIncrement 的值；如果 capacityIncrement 等于或小于 0，则将 Object 数组的大小擴大为现有 size 的两倍，这种容量的控制策略比 ArrayList 更为可控。

**remove(E)**

除了其调用的 removeElement 方法上有 synchronized 关键字外，和 ArrayList 完全相同。

**get(int)**

此方法同样有 synchronized 关键字，实现和 ArrayList 相同。

**iterator()**

和 ArrayList 的实现完全相同。

**contains(E)**

和 ArrayList 唯一的不同是 contains 中调用的 indexOf 方法是有 synchronized 关键字的。

根据上面的分析，可以看出，Vector 除了擴大数组时采用的方法和 ArrayList 不同及线程安全外，其他实现完全相同。

## 注意要点

对于 Vector 而言，最要注意的只有一点：

Vector 是基于 Synchronized 实现的线程安全的 ArrayList，但在插入元素时容量扩充的机制和 ArrayList 稍有不同，并可通过传入 capacityIncrement 来控制容量的扩充。

### 4.1.4 Stack

#### 实现方式

Stack 继承于 Vector，在其基础上实现了 Stack 所要求的后进先出（LIFO）的弹出及压入操作，其提供了 push、pop、peek 三个主要的方法：

##### **push**

push 操作通过调用 Vector 中的 addElement 来完成。

### pop

pop 操作通过调用 peek 来获取元素，并同时删除数组的最后一个元素。

### peek

peek 操作通过获取当前 Object 数组的大小，并获取数组上的最后一个元素。

## 注意要点

对于 Stack 而言，要注意的只有一点：

Stack 基于 Vector 实现，支持 LIFO。

## 4.1.5 HashSet

### 实现方式

HashSet 是 Set 接口的实现，Set 和 List 最明显的区别在于 Set 不允许元素重复，而 List 允许。Set 为了做到不允许元素重复，采用的是基于 HashMap 来实现，HashMap 在 4.1.7 节中再行讲解，在此先看看 HashSet 的实现方式。

#### HashSet()

此时所要做的为创建一个 HashMap 对象。

#### add(E)

调用 HashMap 的 put(Object, Object)方法来完成此操作，将需要增加的元素作为 Map 中的 key，value 则传入一个之前已创建的 Object 对象。

#### remove(E)

调用 HashMap 的 remove(E)方法来完成此操作。

#### contains(E)

调用 HashMap 的 containsKey(E)方法来完成此操作。

#### iterator()

调用 HashMap 的 keySet 的 iterator 方法来完成此操作。

HashSet 不支持通过 get(int)获取指定位置的元素，只能自行通过 iterator 方法来获取。

## 注意要点

对于 HashSet 而言，最要注意的有以下几点：

- HashSet 基于 HashMap 实现，无容量限制；
- HashSet 是非线程安全的。

## 4.1.6 TreeSet

### 实现方式

TreeSet 和 HashSet 的主要不同在于 TreeSet 对于排序的支持，TreeSet 基于 TreeMap 实现，来看看它的具体实现方式。

#### TreeSet()

此时所要做的是创建一个 TreeMap 对象。

#### add(E)

调用 TreeMap 的 put(Object, Object)方法完成此操作，用要增加的元素作为 key，用之前已创建的一个 final 的 Object 对象作为 value。

#### remove(E)

调用 TreeMap 的 remove(Object)方法完成此操作。

#### iterator()

调用 TreeMap 的 navigableKeySet 的 iterator 方法完成此操作。

综上所述，TreeSet 和 HashSet 一样，也是完全基于 Map 来完成的，并且同样也不支持 get(int)来获取指定位置的元素，只是 TreeSet 基于的是 TreeMap，除了这些基本的 Set 实现外，TreeSet 还提供了一些排序方面的支持。例如传入 Comparator 实现、descendingSet 及 descendingIterator 等。

### 注意要点

对于 TreeSet 而言，最要注意的有以下几点：

- TreeSet 基于 TreeMap 实现，支持排序；
- TreeSet 是非线程安全的。

以上分析了 Java 提供的若干常用单对象存储的集合对象之实现。除了这些单对象存储的集合对象外，Java 还提供了 key-value 的集合对象，接口为 Map，常用的主要有 HashMap、TreeMap，接下来就来看看这两个对象的实现方法。

## 4.1.7 HashMap

### 实现方式

HashMap 是 Map 实现中最常使用的，具体实现方式如下。

## HashMap()

将 loadFactor 设为默认的 0.75, threshold 设置为 12, 并创建一个大小为 16 的 Entry 对象数组。

可通过调用 HashMap 的另外两个构造器来控制初始的容量值及 loadFactor, 至于创建的 Entry 对象数组的大小, 并非传入的初始容量值, 而是采用如下方法来决定:

```
int capacity = 1;
while (capacity < initialCapacity)
    capacity <<= 1;
```

capacity 才是真正的创建的 Entry 对象数组的大小, 即真实的 Entry 对象数组的大小应为大于 initialCapacity 的 2 的倍数。例如调用 new HashMap(5,0.6), 按照 HashMap 的实现, 则会将 loadFactor 值设为 0.6, 并创建一个大小为 8 的 Entry 对象数组, threshold 值则为 4。

## put(Object key, Object value)

对于 key 为 null 的情况, HashMap 的做法为获取 Entry 数组中的第一个 Entry 对象, 并基于 Entry 对象的 next 属性遍历。当找到了其中 Entry 对象的 key 属性为 null 时, 则将其 value 赋值为新的 value, 然后返回。如没有 key 属性为 null 的 Entry, 则增加一个 Entry 对象, 增加时为先获取当前数组的第一个 Entry 对象: e, 并创建 Entry 对象, key 为 null, value 为新传入的对象, next 为之前获取的 e, 如此时 Entry 数组中已用的大小  $\geq$  threshold, 则将 Entry 数组扩大为当前大小的两倍, 扩大时对当前 Entry 对象数组中的元素重新 hash, 并填充数组, 最后重新设置 threshold 值。

对于 key 不为 null 的情况, 首先获取 key 对象本身的 hashCode, 然后再对此 hashCode 做 hash 操作, hash 操作的代码为:

```
h ^= (h >>> 20) ^ (h >>> 12);
return h ^ (h >>> 7) ^ (h >>> 4);
```

hash 完毕后, 将 hash 出来的值与 Entry 对象数组的大小减 1 的值进行按位与操作, 从而得出当前 key 要存储到数组的位置。从这个过程可以看出, 可能会出现不同的 key 找到相同的存储位置的问题, 也就是数据结构中经典的 hash 冲突的问题了, 来看看 HashMap 的解决方法。

在找到要存储的目标数组的位置后, 获取该数组对应的 Entry 对象, 通过调用 Entry 对象的 next 来进行遍历, 寻找 hash 值和计算出来的 hash 值相等, 且 key 相等或 equals 的 Entry 对象, 如寻找到, 则替换此 Entry 对象的值, 完成 put 操作, 并返回旧的值; 如未找到, 则往对应的数组位置增加新的 Entry 对象, 增加时采取的方法和 key 为 null 的情况基本相同, 只是它是替换指定位置的 Entry 对象, 可以看出, HashMap 解决 hash 冲突采用的是链表的方式, 而不是开放定址的方法。

## get(Object key)

get 的过程和 put 一样, 也是根据 key 是否为 null 来分别处理的。对于 key 为 null 的情况, 则直接

获取数组中第一个 Entry 对象，并基于 next 属性进行遍历，寻找 key 为 null 的 Entry 对象，如找到则返回 Entry 对象的 value，如未找到，则返回 null；对于 key 为非 null 的情况，则对 key 进行 hash 和按位与操作，找到其对应的存储位置，然后获取此位置对应的 Entry 对象，基于 next 属性遍历，寻找到 hash 值相等，且 key 相等或 equals 的 Entry 对象，返回其 value，如未找到，则返回 null。

### **remove(Object key)**

remove 的过程和 get 类似，只是在找到匹配的 key 后，如数组上的元素等于 key，则将数组上的元素的值置为其 next 元素的值；如数据上的元素不等于 key，则对链表遍历，一直到找到匹配的 key 或链表的结尾。

### **containsKey(Object key)**

通过调用 getEntry 方法来完成，getEntry 方法和 get 过程基本相同。只是在找到匹配的 key 后，直接返回 Entry 对象，而 containsKey 判断返回的 Entry 对象是否为 null，为 null 则返回 false，不为 null 则返回 true。

### **keySet()**

在使用 Map 时，经常会通过 keySet 来遍历 Map 对象，调用 keySet 方法后会返回一个 HashMap 中的 KeySet 对象实例，此 KeySet 对象继承了 AbstractSet。当调用 iterator 方法时，返回一个 KeyIterator 对象实例，调用 next 方法时，遍历整个数组及 Entry 对象的链表，如在遍历过程中有新的元素加入或删除了元素，则会抛出 ConcurrentModificationException。

HashMap 在遍历时是无法保证顺序的，如果要保证 Map 中的对象是按顺序排列的，最好是使用 TreeMap。

从上面来看，HashMap 是非线程安全的，在并发场景中如果不保持足够的同步，就有可能在执行 HashMap.get 时进入死循环，将 CPU 消耗到 100%。具体的现象请参阅相关说明<sup>2</sup>，这个现象在 velocity 的老版本中也会出现<sup>3</sup>，在并发场景中可通过 Collections.synchronizedMap、Collections.unmodifiableMap 或自行同步来保障线程安全，但这些实现方式通常性能会在高并发时下降迅速，最好的方法仍然是使用 4.2 节“并发包”中提到的 ConcurrentHashMap。

## **注意要点**

对 HashMap 而言，最要注意以下几点：

- HashMap 采用数组方式存储 key、value 构成的 Entry 对象，无容量限制；
- HashMap 基于 key hash 寻找 Entry 对象存放到数组的位置，对于 hash 冲突采用链表的方式来解决；

<sup>2</sup> [http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=6423457](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6423457)

<sup>3</sup> <https://issues.apache.org/jira/browse/VELOCITY-718>

- `HashMap` 在插入元素时可能会要扩大数组的容量，在扩大容量时须要重新计算 hash，并复制对象到新的数组中；
- `HashMap` 是非线程安全的。

## 4.1.8 TreeMap

### 实现方式

`TreeMap` 是一个支持排序的 Map 实现，其实现方式和 `HashMap` 并不相同，下面具体看看它的实现方式。

#### `TreeMap()`

在此步 `TreeMap` 只是将 `comparator` 属性赋值为 `null`，如希望控制 `TreeMap` 中元素的存储顺序，可使用带 `Comparator` 参数的构造器。

#### `put(Object key, Object value)`

当调用 `put` 时，先要判断 `root` 属性是否为 `null`，如是，则创建一个新的 `Entry` 对象，并赋值给 `root` 属性。如不是，则首先判断是否传入了指定的 `Comparator` 实现，如已传入，则基于红黑树的方式遍历，基于 `comparator` 来比较 `key` 应放在树的左边还是右边，如找到相等的 `key`，则直接替换其 `value`，并返回结束 `put` 操作，如没有找到相等的 `key`，则一直寻找到左边或右边节点为 `null` 的元素，如 `comparator` 实现为 `null`，则判断 `key` 是否为 `null`，是则抛出 `NullPointerException`，并将 `key` 造型为 `Comparable`，进行与上面同样的遍历和比较过程。

通过上面的步骤，如未找到相同的 `key`，则进入以下过程，即创建一个新的 `Entry` 对象，并将其 `parent` 设置为上面所寻找到的元素，并根据和 `parent key` 比较的情况来设置 `parent` 的 `left` 或 `right` 属性。

综上所述，`TreeMap` 是一个典型的基于红黑树的实现，因此它要求一定要有 `key` 比较的方法，要么传入 `Comparator` 实现，要么 `key` 对象实现 `Comparable` 接口。

#### `get(Object)`

`TreeMap` 在查找 `key` 时就是个典型的红黑树查找过程，从根对象开始往下比较，一直找到相等的 `key`，并返回其 `value`。

和 `put` 时同样的处理方式，如未传入 `Comparator` 实现，当传入的 `Object` 为 `null` 时，则直接抛出 `NullPointerException`。

#### `remove(Object)`

`remove` 首先要做的是 `getEntry`，然后则是将此 `Entry` 从红黑树上删除，并重新调整树上的相关的节点。

### containsKey(Object)

和 get 方法一样，都通过 getEntry 方法来完成，因此过程和 get 方法一样，只是 containsKey 直接判断返回的 Entry 是否为 null，为 null 则返回 false，不为 null 则返回 true。

### keySet()

调用 keySet 方法后返回 TreeMap 的内部类 KeySet 对象的实例，iterator 的遍历从根开始，基于红黑树顺序完成。

## 注意要点

对 TreeMap 而言，最应了解的有以下几点：

- TreeMap 基于红黑树实现，无容量限制；
- TreeMap 是非线程安全的。

## 4.1.9 性能测试<sup>4</sup>

在性能方面，关注的主要是不同的集合对象在相同的场景下的性能对比状况，场景设计上分为单线程场景和多线程场景。

在单线程下，按照如下场景测试：

- 测试在不同的集合大小（分别为 10、100、1000、10000）下增加、查找及删除 100 个元素的性能变化情况；

在多线程下，按照如下场景进行测试：

- 测试在不同的集合大小（分别为 10、100、1000、10000）及不同的线程数（分别为 10、50、100）的情况下，增加、查找及删除 100 个元素的性能变化情况；

对于本节提及的集合对象，都进行以上场景的性能测试，每个场景均运行 10 次，取平均值，以尽量保证测试能充分地体现集合对象的性能状况。

## 单线程下运行结果

在集合元素数量为 10 时，运行结果如图 4.1 所示。

从图 4.1 来看，在增加元素上，ArrayList 的性能相对较差，原因是 ArrayList 在容量不够时需要扩充。使 Vector 性能表现更好的原因在于虽然其扩充容量的机制不同，但由于评测的单位为纳秒，因此可以认为在目前这个场景中，各种集合类在增加元素时性能的变化差距不大。

在查找元素上，ArrayList、LinkedList、Vector、Stack 的性能略差一点点，这是由于它们在查找时

<sup>4</sup> 性能测试代码请从 <http://bluedavy.com> 下载

需要遍历整个集合，而 Set、Map 类型的都是通过 hash 后再到链表上查找，因此速度会更快。

在删除元素上，除 TreeSet 和 TreeMap 外，其他集合类的性能基本无差距，TreeSet 基于 TreeMap 而实现，TreeMap 之所以性能相对较差的原因是它在删除时需要排序。

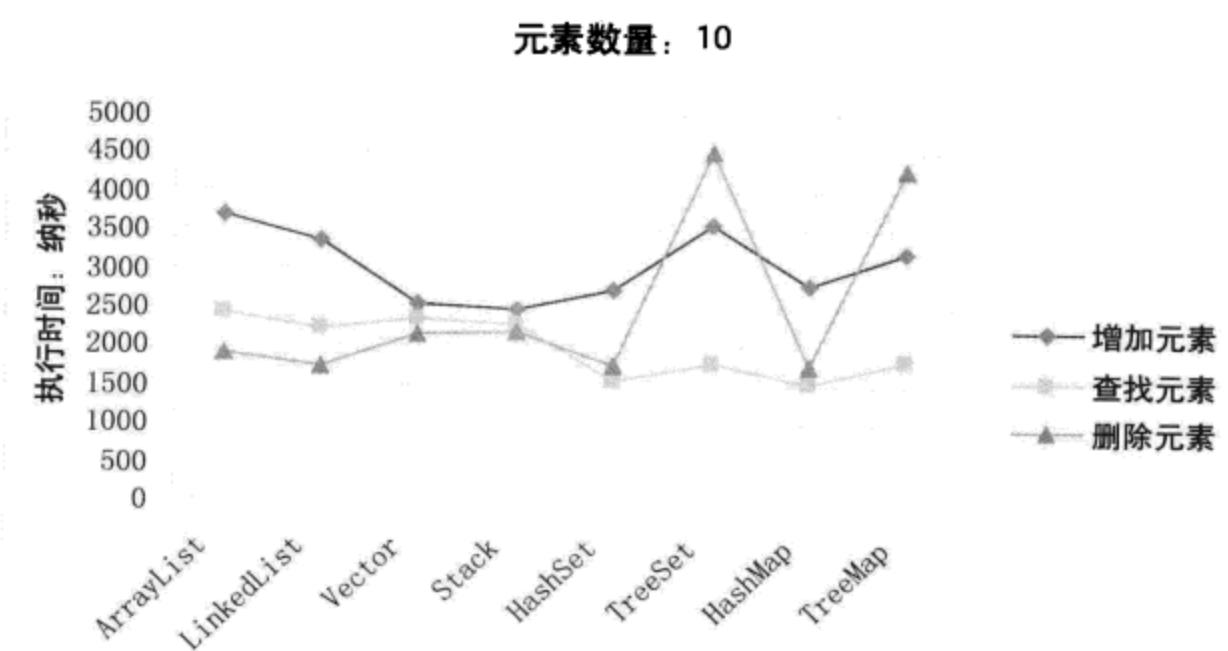


图 4.1 单线程下元素数量为 10 时的运行结果对比

运行结果数据如表 4.1 所示：

表 4.1 单线程下元素数量为 10 时的详细运行结果 单位：纳秒

| 集合类型       | 增加元素 | 查找元素 | 删除元素 |
|------------|------|------|------|
| ArrayList  | 3656 | 2396 | 1887 |
| LinkedList | 3334 | 2200 | 1712 |
| Vector     | 2547 | 2335 | 2157 |
| Stack      | 2469 | 2261 | 2173 |
| HashSet    | 2716 | 1571 | 1754 |
| TreeSet    | 3556 | 1776 | 4504 |
| HashMap    | 2805 | 1520 | 1743 |
| TreeMap    | 3217 | 1814 | 4277 |

从整体来看，大多数集合类在元素数量为 10 的情况下，增加元素的性能相对比查找和删除差一些，这里的原因在于大多数集合类在增加元素时都要扩容，并且元素数量只有 10，而插入的元素有 100 个，有可能会造成多次扩容。

在集合元素数量为 100 时，运行结果如图 4.2 所示。

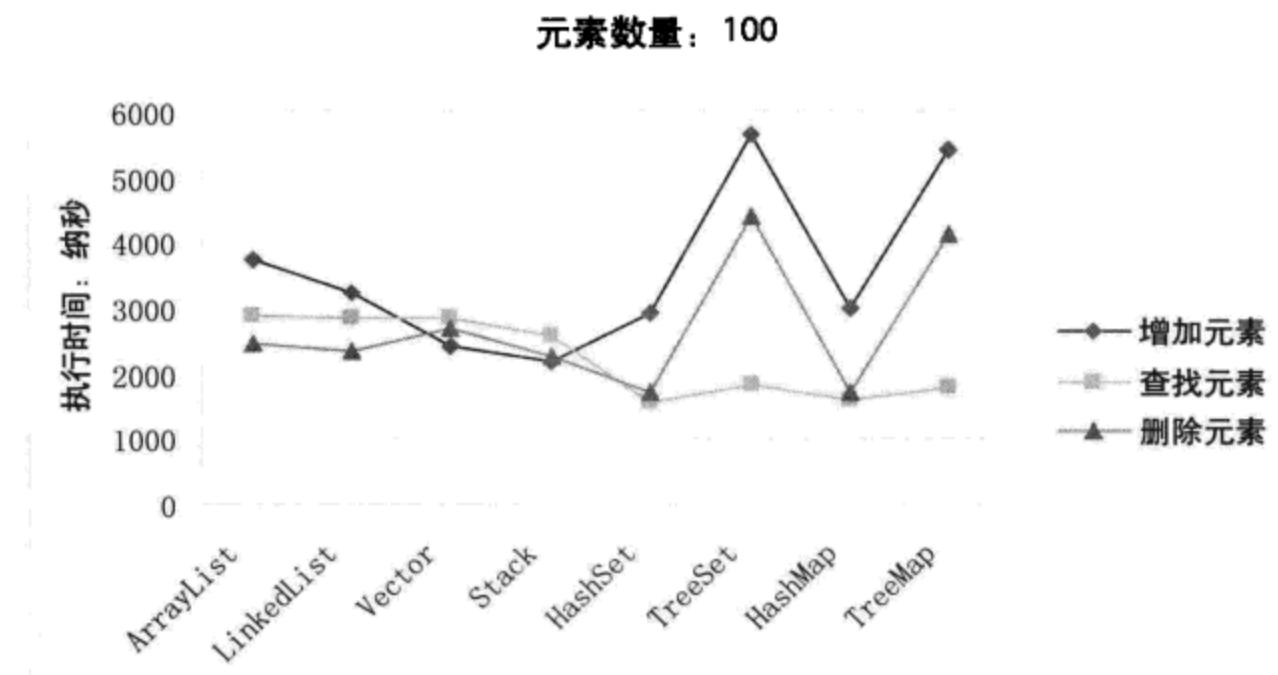


图4.2 单线程下元素数量为100时的运行结果对比

运行结果数据如表4.2所示。

表4.2 单线程下元素数量为100时的详细运行结果 单位：纳秒

| 集合类型       | 增加元素 | 查找元素 | 删除元素 |
|------------|------|------|------|
| ArrayList  | 3756 | 2873 | 2444 |
| LinkedList | 3251 | 2829 | 2330 |
| Vector     | 2402 | 2846 | 2704 |
| Stack      | 2180 | 2572 | 2277 |
| HashSet    | 2930 | 1559 | 1720 |
| TreeSet    | 5662 | 1846 | 4413 |
| HashMap    | 3004 | 1579 | 1726 |
| TreeMap    | 5410 | 1804 | 4123 |

从图4.2来看，在增加元素上，此时TreeMap和TreeSet的性能下降比较明显，其他几个集合类则基本没变化，仍然是ArrayList稍差一点点，但基本可忽略。从查找元素来看，和元素数量为10时基本没有变化。

删除元素的表现和元素数量为10时也基本一致。

从整体来看，当元素数量为100时，除TreeSet和TreeMap外，和元素数量为10时基本没有太大差距。

在集合元素数量为 1000 时，运行结果如图 4.3 所示。

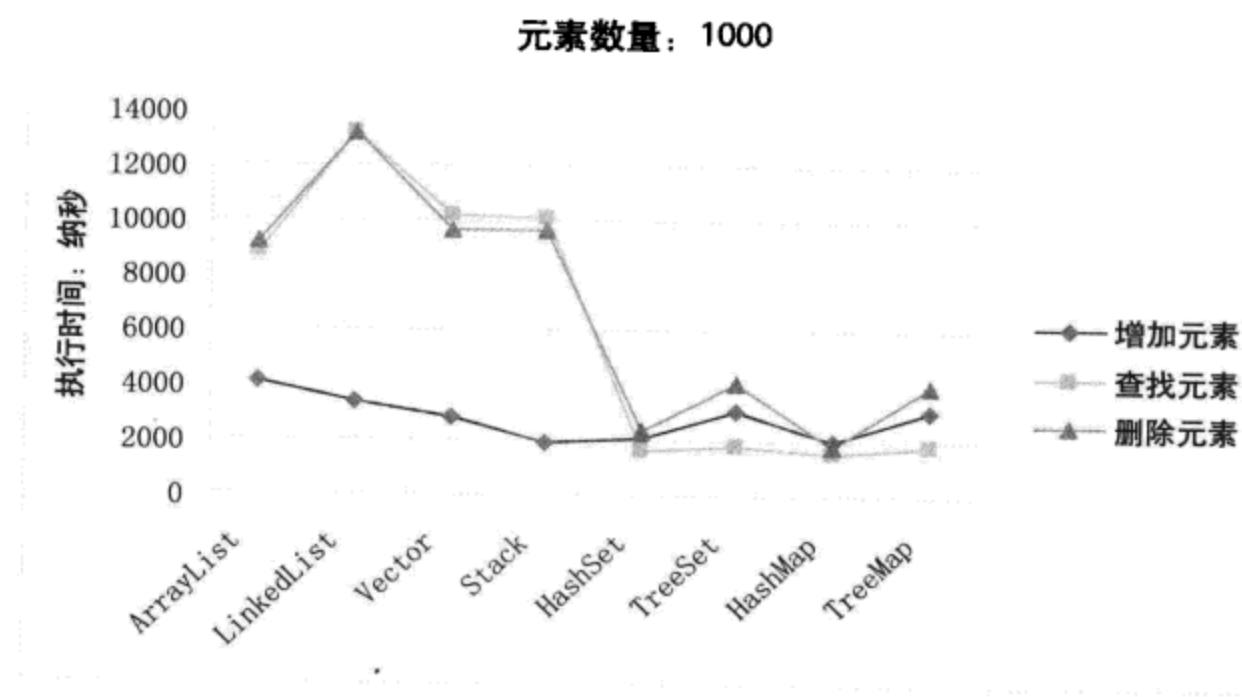


图 4.3 单线程下元素数量为 1000 时的运行结果对比

运行结果数据如表 4.3 所示。

表 4.3 单线程下元素数量为 1000 时的详细运行结果 单位：纳秒

| 集合类型       | 增加元素 | 查找元素  | 删除元素  |
|------------|------|-------|-------|
| ArrayList  | 4107 | 8857  | 9155  |
| LinkedList | 3332 | 13165 | 13166 |
| Vector     | 2819 | 10177 | 9610  |
| Stack      | 1864 | 10091 | 9598  |
| HashSet    | 2117 | 1632  | 2371  |
| TreeSet    | 3049 | 1820  | 4088  |
| HashMap    | 1995 | 1528  | 1846  |
| TreeMap    | 3135 | 1832  | 4004  |

从图 4.3 来看，在增加元素上，与元素数 100 比执行时间稍有上升，但没有太大变化。

在查找元素上，变化就非常明显了，ArrayList、LinkedList、Vector 及 Stack 的查找速度大幅度下降，这是查找时要遍历整个集合，而 HashSet、TreeSet、HashMap、TreeMap 则基本没有变化。

在删除元素上，性能变化和查找元素基本一致。

从整体来看，当元素数量上升到 1000 后，List 的几个实现类查找、删除元素的性能下降非常明显。

在集合元素数量为 10000 时，运行结果如图 4.4 所示。

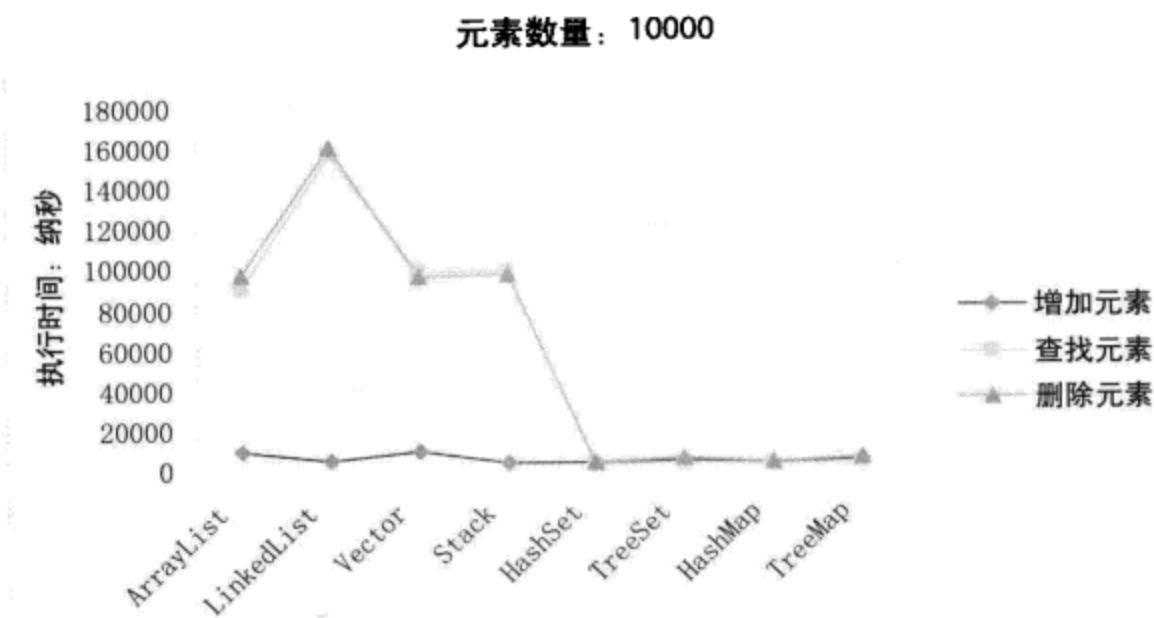


图 4.4 单线程下元素数量为 10000 时的运行结果对比

运行结果数据如表 4.4 所示：

表 4.4 单线程下元素数量为 10000 时的详细运行结果 单位：纳秒

| 集合类型       | 增加元素 | 查找元素   | 删除元素   |
|------------|------|--------|--------|
| ArrayList  | 8136 | 88324  | 95948  |
| LinkedList | 3348 | 155586 | 159163 |
| Vector     | 8382 | 96557  | 94890  |
| Stack      | 1919 | 96740  | 95862  |
| HashSet    | 2091 | 1510   | 1849   |
| TreeSet    | 3275 | 2044   | 4279   |
| HashMap    | 2032 | 1501   | 1787   |
| TreeMap    | 3908 | 2130   | 4819   |

从图 4.4 来看，当元素数量上升至 10000 后，插入元素时除 ArrayList、Vector 扩容影响较大外，其他实现类基本没有变化，在查找、删除元素上，几个 List 的实现性能进一步下降，其他基本没有变化。根据以上对各种集合类在不同元素数量的测试来看，在目前的测试场景中，List 的实现随着元素数量的上升，查找和删除元素时性能下降较为严重，Set、Map 的实现则基本不会受元素数量的影响。从这样的测试结果来看，对于查找和删除较为频繁，且元素数量较多的应用，Set 或 Map 是更好的选择，而对于其他场景，则可根据需要的数据结构来进行相应的选择。

## 多线程下运行结果

### 元素数量为 10 时不同线程数的运行结果对比

线程数为 10 时结果如图 4.5 所示。

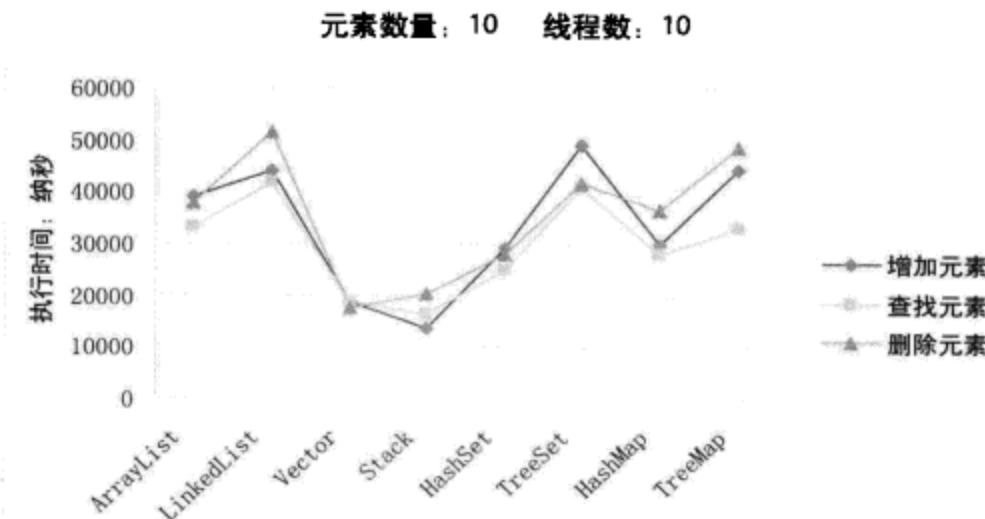


图 4.5 线程数及元素数量均为 10 时的运行结果对比

详细结果数据如表 4.5 所示。

表 4.5 线程数及元素数均为 10 时的运行结果数据 单位：纳秒

| 集合类型       | 增加元素  | 查找元素  | 删除元素  |
|------------|-------|-------|-------|
| ArrayList  | 38820 | 33275 | 37653 |
| LinkedList | 44112 | 41815 | 51428 |
| Vector     | 18559 | 18689 | 17507 |
| Stack      | 13456 | 16459 | 20102 |
| HashSet    | 29366 | 24787 | 27968 |
| TreeSet    | 49142 | 40774 | 41776 |
| HashMap    | 29814 | 28076 | 36539 |
| TreeMap    | 44591 | 33013 | 48794 |

线程数为 50 时结果如图 4.6 所示。

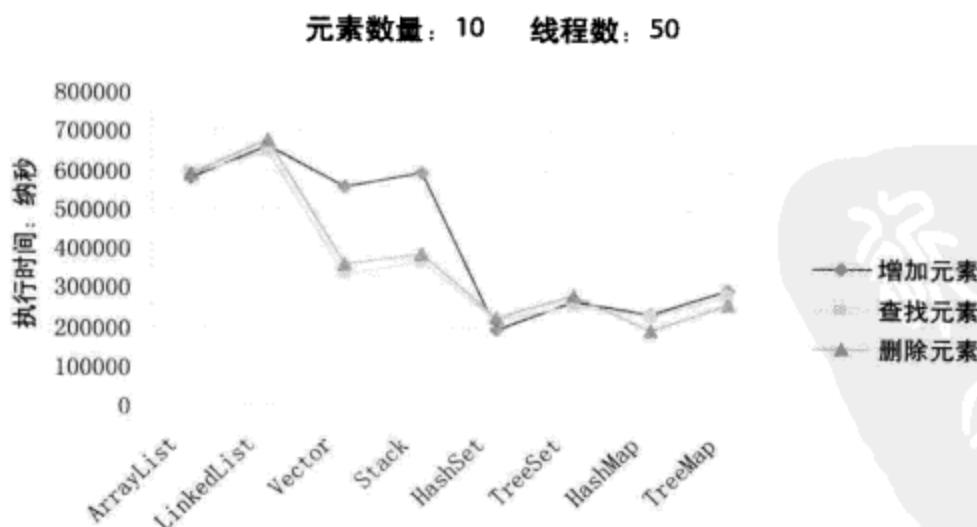


图 4.6 线程数：50，元素数量：10 时的运行结果对比

详细结果数据如表4.6所示。

表4.6 线程数：50，元素数量：10时的运行结果数据 单位：纳秒

| 集合类型       | 增加元素   | 查找元素   | 删除元素   |
|------------|--------|--------|--------|
| ArrayList  | 576732 | 586332 | 586700 |
| LinkedList | 657509 | 644244 | 676147 |
| Vector     | 553418 | 334359 | 356423 |
| Stack      | 593575 | 365376 | 383725 |
| HashSet    | 191416 | 216589 | 220968 |
| TreeSet    | 265843 | 253497 | 278616 |
| HashMap    | 233674 | 230404 | 193918 |
| TreeMap    | 295981 | 278430 | 260596 |

线程数为100时结果如图4.7所示。

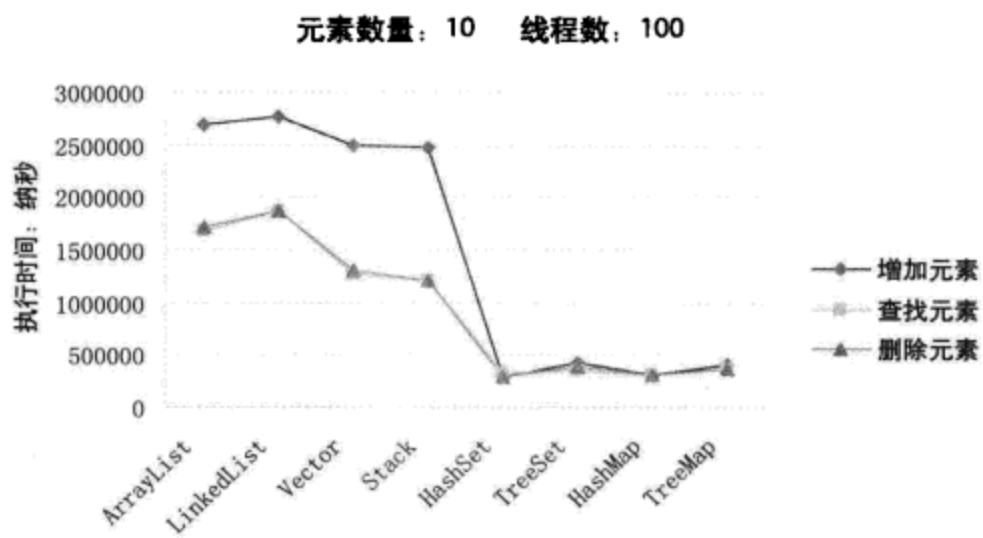


图4.7 线程数：100，元素数量：10时的运行结果对比

详细结果数据如表4.7所示：

表4.7 线程数：100，元素数量：10时的运行结果数据 单位：纳秒

| 集合类型       | 增加元素    | 查找元素    | 删除元素    |
|------------|---------|---------|---------|
| ArrayList  | 2692233 | 1674142 | 1706657 |
| LinkedList | 2762555 | 1876966 | 1866544 |
| Vector     | 2492930 | 1267761 | 1302873 |
| Stack      | 2483026 | 1209293 | 1210255 |
| HashSet    | 294091  | 323572  | 286909  |
| TreeSet    | 424753  | 359113  | 394129  |
| HashMap    | 313442  | 302591  | 309336  |
| TreeMap    | 409097  | 370090  | 379483  |

从以上结果来看，在多线程时各集合类的性能下降非常明显，并且随着线程数增加，下降得就越多，相对而言，Set 和 Map 的实现表现较好一些。

### 元素数量为 100 时不同线程数的运行结果对比

线程数为 10 时结果如图 4.8 所示：

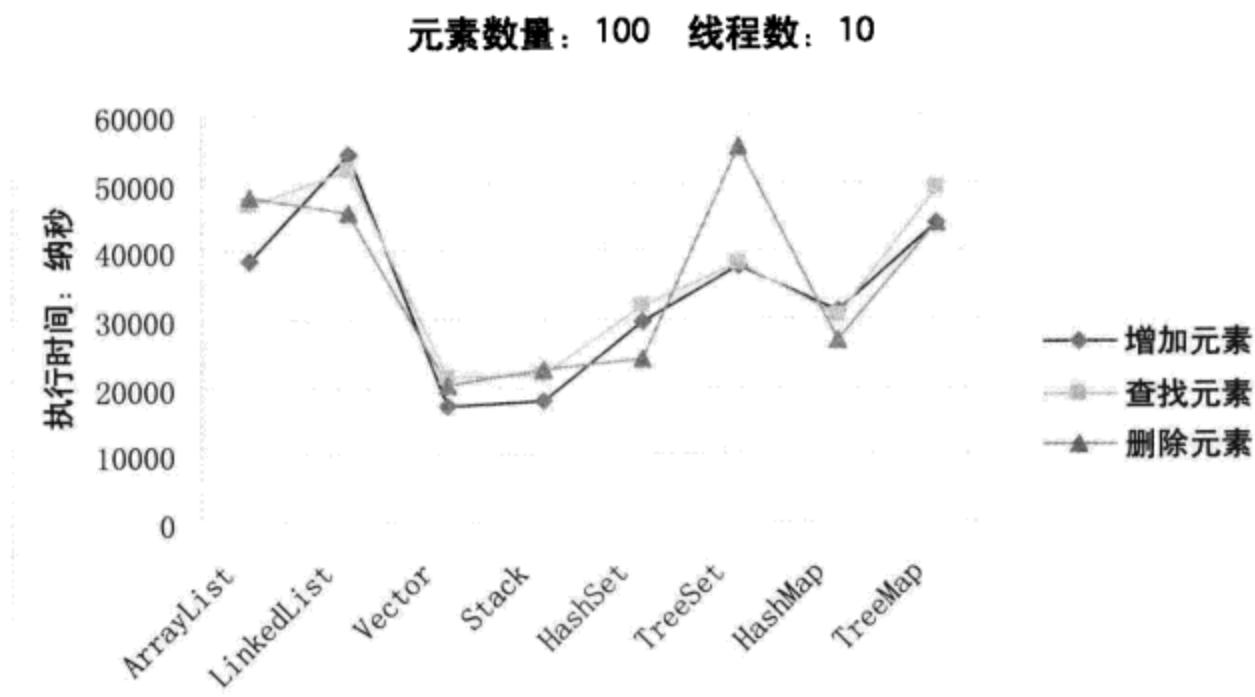


图 4.8 线程数：10，元素数量：100 时的运行结果对比

详细结果数据如表 4.8 所示：

表 4.8 线程数：10，元素数：100 时的详细运行数据 单位：纳秒

| 集合类型       | 增加元素  | 查找元素  | 删除元素  |
|------------|-------|-------|-------|
| ArrayList  | 38748 | 46759 | 47861 |
| LinkedList | 54119 | 51783 | 45735 |
| Vector     | 17104 | 21256 | 20290 |
| Stack      | 18045 | 21793 | 22514 |
| HashSet    | 29519 | 31888 | 24257 |
| TreeSet    | 37749 | 38156 | 55178 |
| HashMap    | 31180 | 30199 | 26876 |
| TreeMap    | 44013 | 48965 | 44204 |

线程数为 50 时结果如图 4.9 所示：

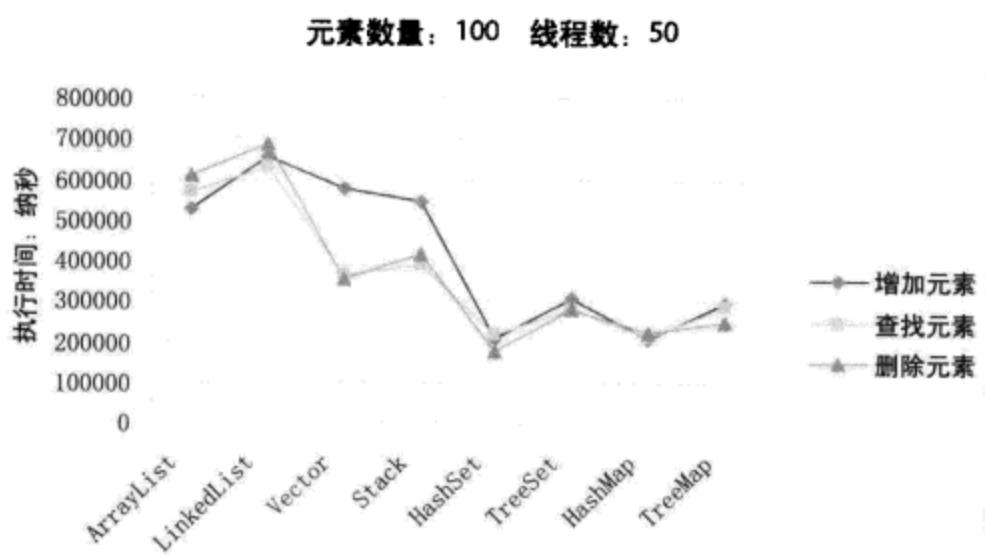


图 4.9 线程数: 50, 元素数量: 100 时的运行结果对比

详细结果数据如表 4.9 所示：

表 4.9 线程数: 50, 元素数: 100 时的详细运行数据 单位: 纳秒

| 集合类型       | 增加元素   | 查找元素   | 删除元素   |
|------------|--------|--------|--------|
| ArrayList  | 526320 | 566639 | 608195 |
| LinkedList | 653135 | 627526 | 686270 |
| Vector     | 578554 | 366000 | 352505 |
| Stack      | 547978 | 390510 | 414465 |
| HashSet    | 209067 | 216422 | 175491 |
| TreeSet    | 306060 | 278831 | 279674 |
| HashMap    | 205577 | 213651 | 221995 |
| TreeMap    | 296959 | 283949 | 251197 |

线程数为 100 时结果如图 4.10 所示：

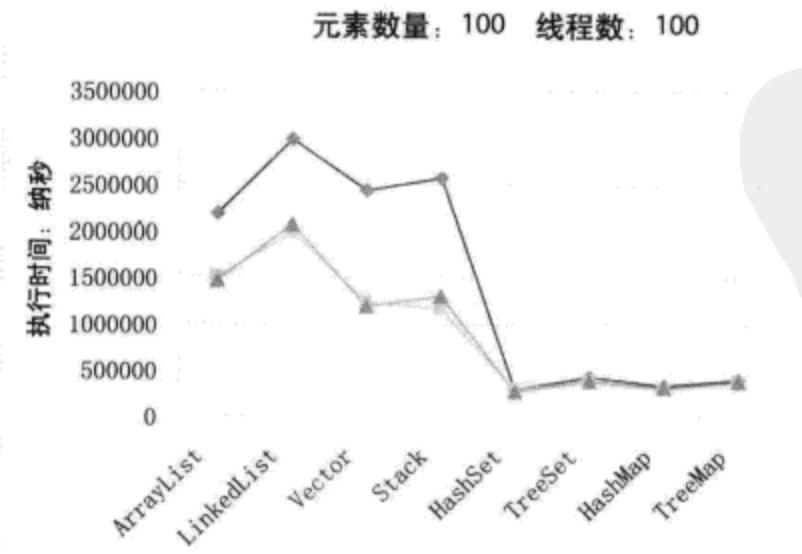


图 4.10 线程数: 100, 元素数量: 100 时的运行结果对比

详细结果数据如表 4.10 所示：

表 4.10 线程数：100，元素数：100 时的详细运行数据 单位：纳秒

| 集合类型       | 增加元素    | 查找元素    | 删除元素    |
|------------|---------|---------|---------|
| ArrayList  | 2174587 | 1498958 | 1444283 |
| LinkedList | 2970990 | 1986583 | 2062245 |
| Vector     | 2422422 | 1236455 | 1182016 |
| Stack      | 2572466 | 1160000 | 1298391 |
| HashSet    | 299294  | 294524  | 261959  |
| TreeSet    | 435873  | 381115  | 384677  |
| HashMap    | 332389  | 296360  | 314167  |
| TreeMap    | 419681  | 388058  | 378543  |

和元素数量在 10 时的表现基本一致，没有太大变化，主要是 LinkedList 开始显现出相对性能较差。

#### 元素数量为 1000 时不同线程数的运行结果对比

线程数为 10 时结果如图 4.11 所示：

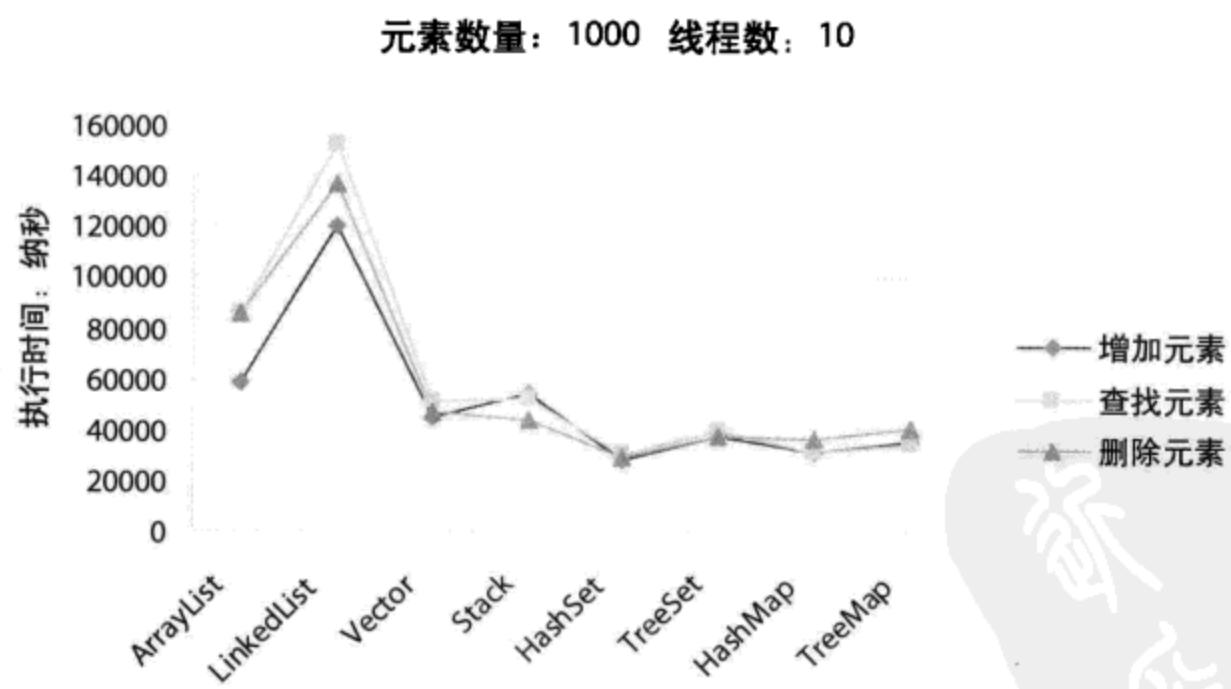


图 4.11 线程数：10，元素数量：1000 时的运行结果对比

详细结果数据如表 4.11 所示。

表4.11 线程数：10，元素数：1000时的详细运行数据 单位：纳秒

| 集合类型       | 增加元素   | 查找元素   | 删除元素   |
|------------|--------|--------|--------|
| ArrayList  | 58677  | 85960  | 86104  |
| LinkedList | 120314 | 152338 | 137309 |
| Vector     | 44570  | 50776  | 47469  |
| Stack      | 54375  | 52010  | 44360  |
| HashSet    | 28674  | 30253  | 29007  |
| TreeSet    | 37695  | 39499  | 37292  |
| HashMap    | 31217  | 31795  | 36746  |
| TreeMap    | 35916  | 34621  | 41179  |

线程数为50时结果如图4.12所示：

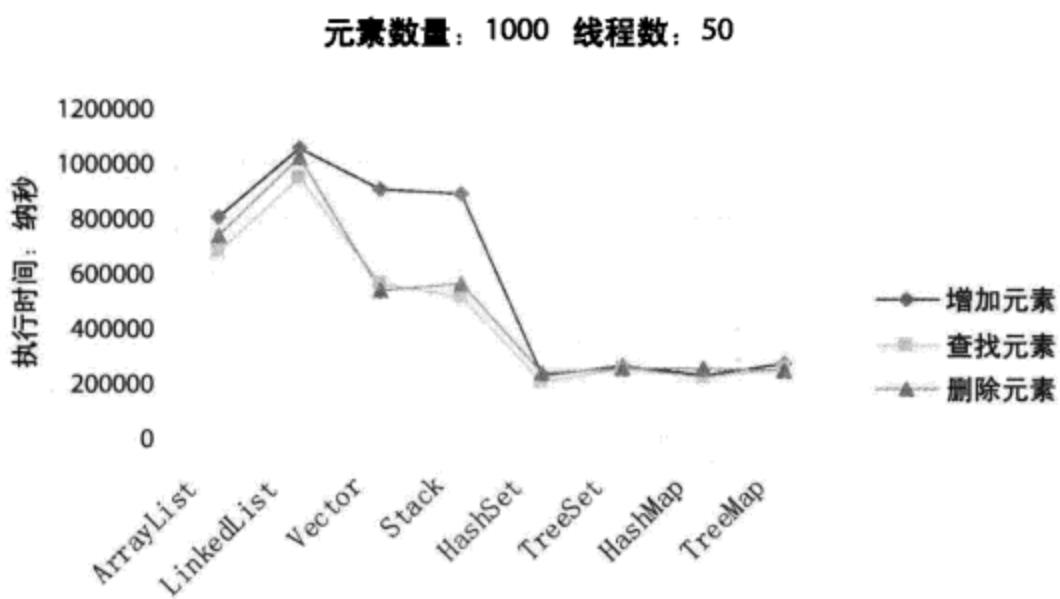


图4.12 线程数：50，元素数量：1000时的运行结果对比

详细结果数据如表4.12所示：

表4.12 线程数：50，元素数：1000时的详细运行数据 单位：纳秒

| 集合类型       | 增加元素    | 查找元素   | 删除元素    |
|------------|---------|--------|---------|
| ArrayList  | 799344  | 678087 | 731936  |
| LinkedList | 1049523 | 943351 | 1013457 |
| Vector     | 895905  | 560230 | 530693  |
| Stack      | 886182  | 506775 | 559000  |
| HashSet    | 224973  | 200811 | 234216  |
| TreeSet    | 255927  | 252306 | 246441  |
| HashMap    | 226758  | 213811 | 251577  |
| TreeMap    | 263651  | 249513 | 245534  |

线程数为 100 时结果如图 4.13 所示：

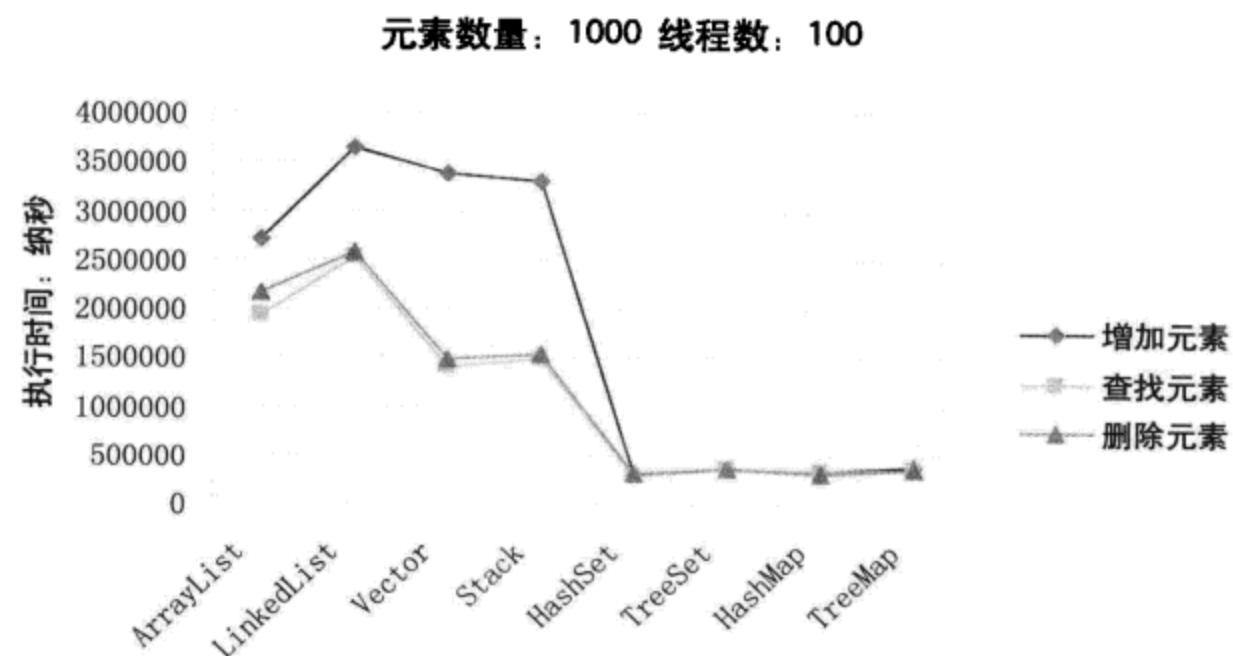


图 4.13 线程数：100，元素数量：1000 时的运行结果对比

详细结果数据如表 4.13 所示：

表 4.13 线程数：100，元素数：1000 时的运行结果数据 单位：纳秒

| 集合类型       | 增加元素    | 查找元素    | 删除元素    |
|------------|---------|---------|---------|
| ArrayList  | 2690648 | 1916825 | 2153334 |
| LinkedList | 3628969 | 2523953 | 2575092 |
| Vector     | 3381701 | 1415252 | 1483548 |
| Stack      | 3300619 | 1486736 | 1520346 |
| HashSet    | 311353  | 282810  | 304965  |
| TreeSet    | 368060  | 356689  | 372603  |
| HashMap    | 343005  | 343153  | 318603  |
| TreeMap    | 377826  | 373630  | 352816  |

从以上运行结果来看，并没有像单线程那样，在元素数量增长为 1000 时查找、删除速度大幅度下降，而是基本保持了和元素数量为 100 时差不多的速度。但和元素数量为 100 时同样，LinkedList 的性能相对更差一些。

#### 元素数量为 10000 时不同线程数的运行结果对比

线程数为 10 时结果如图 4.14 所示：

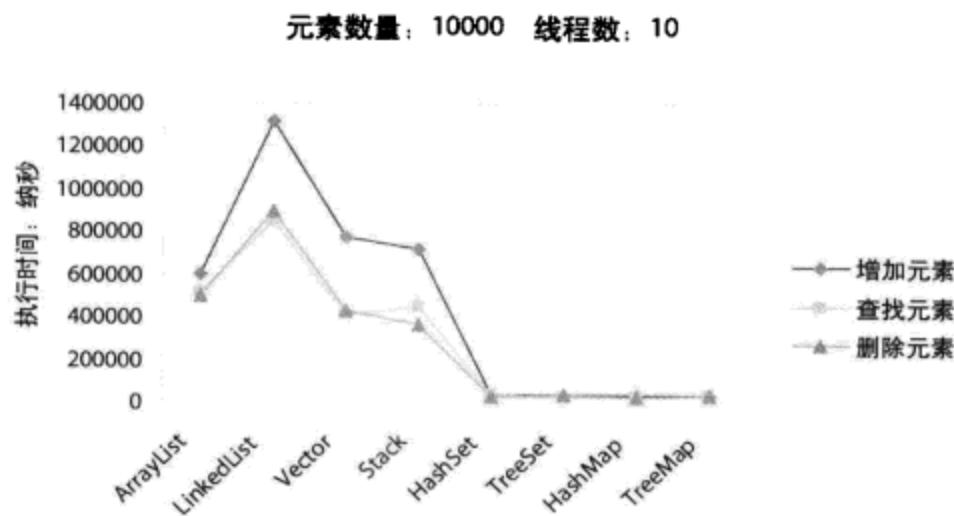


图 4.14 线程数: 10, 元素数量: 10000 时的运行结果对比

详细结果数据如表 4.14 所示:

表 4.14 线程数 10, 元素数: 10000 时的运行结果数据表 单位: 纳秒

| 集合类型       | 增加元素    | 查找元素   | 删除元素   |
|------------|---------|--------|--------|
| ArrayList  | 601045  | 514425 | 490858 |
| LinkedList | 1311618 | 848818 | 891982 |
| Vector     | 778250  | 412841 | 427850 |
| Stack      | 718416  | 451952 | 362086 |
| HashSet    | 34121   | 32800  | 30102  |
| TreeSet    | 37188   | 30160  | 37357  |
| HashMap    | 33314   | 26488  | 30497  |
| TreeMap    | 36214   | 35960  | 34675  |

线程数为 50 时结果如图 4.15 所示:

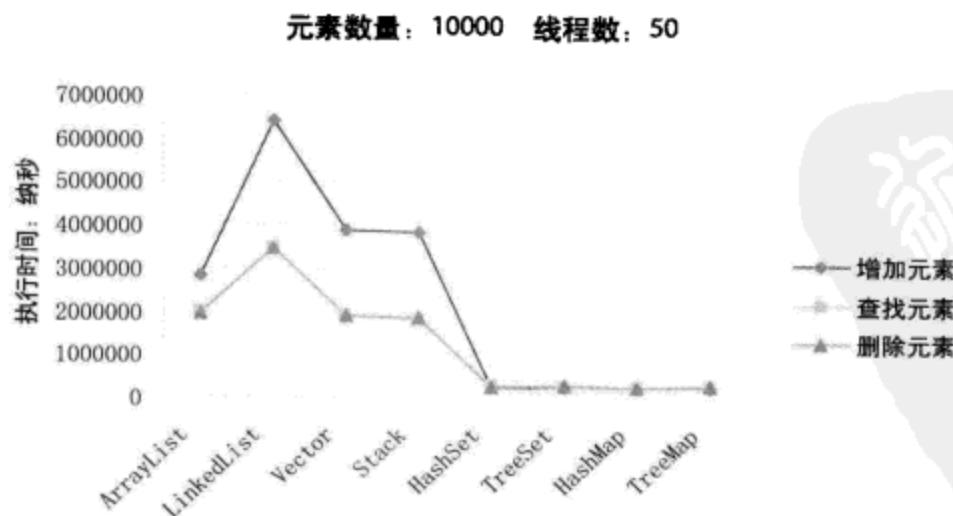


图 4.15 线程数: 50, 元素数量: 10000 时的运行结果对比

详细结果数据如表 4.15 所示：

表 4.15 线程数：50，元素数量：10000 时的运行结果数据 单位：纳秒

| 集合类型       | 增加元素    | 查找元素    | 删除元素    |
|------------|---------|---------|---------|
| ArrayList  | 2814190 | 1967233 | 1928655 |
| LinkedList | 6386225 | 3466409 | 3473760 |
| Vector     | 3887223 | 1898861 | 1887354 |
| Stack      | 3828071 | 1808925 | 1852438 |
| HashSet    | 230218  | 275915  | 220992  |
| TreeSet    | 251450  | 246466  | 261872  |
| HashMap    | 249921  | 234555  | 229954  |
| TreeMap    | 230799  | 231221  | 255469  |

线程数为 100 时结果如图 4.16 所示：

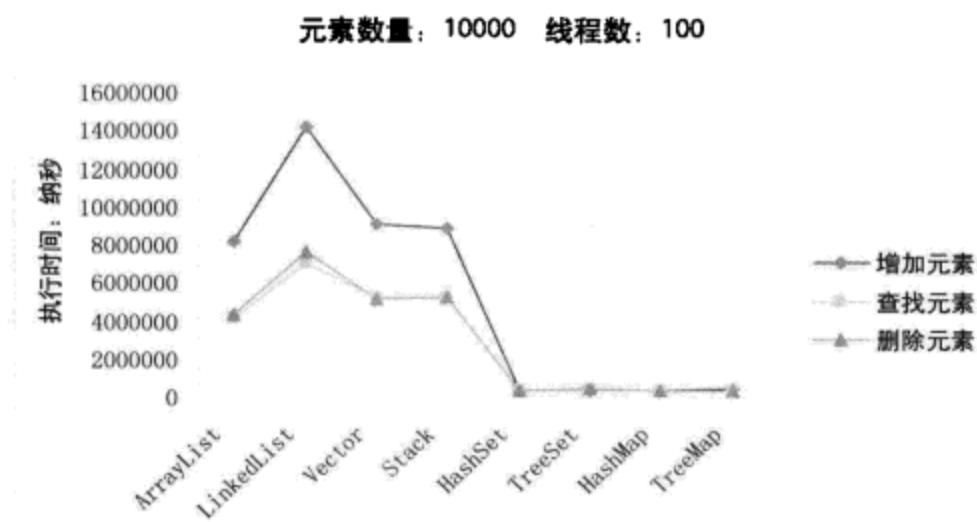


图 4.16 线程数：100，元素数量：10000 时的运行结果对比

详细结果数据如表 4.16 所示：

表 4.16 线程数：100，元素数量：10000 时的运行结果数据 单位：纳秒

| 集合类型       | 增加元素     | 查找元素    | 删除元素    |
|------------|----------|---------|---------|
| ArrayList  | 8066502  | 4090260 | 4302891 |
| LinkedList | 14085903 | 6987035 | 7536079 |
| Vector     | 9089352  | 5225642 | 5139321 |
| Stack      | 8827585  | 5231105 | 5215321 |
| HashSet    | 339069   | 297783  | 284861  |
| TreeSet    | 345273   | 345266  | 411913  |
| HashMap    | 329171   | 308225  | 320326  |
| TreeMap    | 369650   | 354593  | 352704  |

各集合类的性能和元素少时基本一致，仍然是 `LinkedList` 稍差，但当元素数量上涨到 10000 后，各集合类的操作的响应速度均大幅度下降。

从整体来看，在多线程的场景下，各集合类的性能较之单线程而言都大幅度下降，但在元素数量为 1000 左右时，性能的影响因素主要取决于操作的线程数。线程数越多，性能下降得越多，`Set` 和 `Map` 的实现再各种场景下表现均更出色，随着元素数量或线程数增多，`LinkedList` 在性能上表现则相对差一些。

#### 4.1.10 小结

以上只是对于各种集合类的基准测试，感兴趣的读者可用基准测试的代码测试更多的集合类，或对测试代码优化，更深入地理解各集合类中的控制参数，提升它们在基本测试中的性能表现，例如通过控制 `HashMap` 的 `loadFactor` 来提升性能等。

在实际的使用中首先要根据功能需求来选择是用 `List`、`Set` 还是 `Map`，`List` 适用于允许重复元素的单个对象集合场景，`Set` 适用于不允许重复元素的单个对象集合场景，`Map` 则适用于 `key-value` 结构的集合场景。

在选择好 `List`、`Set` 和 `Map` 后，就要选择相应的实现类了，`ArrayList` 适用于要通过位置来读取元素的场景；`LinkedList` 适用于要头尾操作及插入指定位置的场景；`Vector` 适用于要线程安全的 `ArrayList` 场景；`Stack` 适用线程安全的 LIFO 场景，如需支持 FIFO，可参考 4.2.4 节中的 `ArrayBlockingQueue`；`HashSet` 适用于对排序没有要求的非重复元素的存放；`TreeSet` 适用于要排序的非重复元素的存放；`HashMap` 适用于大部分 `key-value` 的存取场景；`TreeMap` 适用于须排序存放的 `key-value` 场景，根据相应的场景选择以上的类，或其他的一些 Sun JDK 类库中的类，例如 `Hashtable`、`LinkedHashSet` 等。

最后要根据场景中需要存储的数据量、操作（例如增加元素、删除元素）、并发量来进行相应的性能测试，如可满足性能需求则直接使用即可，如无法满足性能需求，则通常要自行实现或选择并发包中提供的集合类。在考量性能的同时，内存的消耗、CPU 的使用率也是要关注的因素。

对于分布式 Java 应用，并发是通常要面对的场景，为了使开发人员能够编写高性能的并发程序，Sun JDK 从 1.5.0 后提供了一个并发包，基于这个并发包，开发人员能够更简单地编写出高性能的并发程序。

## 4.2 并发包（java.util.concurrent）

并发包中除了提供高性能的线程安全的集合对象外，还提供了很多并发场景需要的原子操作类，例如 `AtomicInteger`，另外提供了一些用于避免并发时资源冲突的 `Lock` 及 `Condition` 类。下面来看看并发包中一些常用的类的实现。

## 4.2.1 ConcurrentHashMap

### 实现方式

ConcurrentHashMap 是线程安全的 HashMap 的实现，其实现方式如下。

#### ConcurrentHashMap()

和 HashMap 一样，它同样有 initialCapacity 和 loadFactor 属性，不过还多了一个 concurrencyLevel 属性，在调用空构造器的情况下，这三个属性的值分别设置为 16、0.75 及 16。

在设置了以上三个属性值后，基于以下方式计算 ssize 值：

```
int sshift = 0;
int ssize = 1;
while (ssize < concurrencyLevel) {
    ++sshift;
    ssize <<= 1;
}
```

在 concurrencyLevel 为 16 的情况下，最终计算出的 ssize 为 16，并使用此 ssize 作为参数传入 Segment 的 newArray 方法，创建大小为 16 的 Segment 对象数组，接着采用如下方法计算 cap 变量的值：

```
int c = initialCapacity / ssize;
if (c * ssize < initialCapacity)
    ++c;
int cap = 1;
while (cap < c)
    cap <<= 1;
```

根据以上参数值，计算出的 cap 为 1，最后为 Segment 对象数组创建 Segment 对象，传入的参数为 cap 和 loadFactor。Segment 对象继承 ReentrantLock，在创建 Segment 对象时，其所做的动作为创建一个指定大小为 cap 的 HashEntry 对象数组，并基于数组的大小以及 loadFactor 计算 threshold 的值：threshold = (int)(newTable.length \* loadFactor);。

#### put(Object key, Object value)

ConcurrentHashMap 并没有在此方法上加上 synchronized，首先判断 value 是否为 null，如为 null 则抛出 NullPointerException，如不为 null，则继续下面的步骤。

和 HashMap 一样，首先对 key.hashCode 进行 hash 操作，得到 key 的 hash 值。hash 操作的算法和 HashMap 也不同。

根据此 hash 值计算并获取其对应的数组中的 Segment 对象，方法如下：

```
return segments[(hash >>> segmentShift) & segmentMask];
```

在找到了数组中的 Segment 对象后，接着调用 Segment 对象的 put 方法来完成当前操作。

当调用 Segment 对象的 put 方法时，首先进行 lock 操作，接着判断当前存储的对象个数加 1 后是否大于 threshold。如大于，则将当前的 HashEntry 对象数组大小扩大两倍，并将之前存储的对象重新 hash，转移到新的对象数组中。在确保了数组的大小足够后，继续下面的步骤。

接下去的动作则和 HashMap 基本一样，通过对 hash 值和对象数组大小减 1 的值进行按位与操作后，得到当前 key 需要放入数组的位置，接着寻找对应位置上的 HashEntry 对象链表是否有 key、hash 值和当前 key 相同的。如有，则覆盖其 value，如没有，则创建一个新的 HashEntry 对象，赋值给对应位置的数组对象，并构成链表。

完成了上面的步骤后，释放锁，整个 put 动作得以完成。

根据以上分析，可以看出，ConcurrentHashMap 基于 concurrencyLevel 划分出了多个 Segment 来对 key-value 进行存储，从而避免每次 put 操作都得锁住整个数组。在默认的情况下，最佳情况下可以允许 16 个线程并发无阻塞的操作集合对象，尽可能地减少并发时的阻塞现象。

### remove(Object key)

首先对 key.hashCode 进行 hash 操作，基于得到 hash 的值找到对应的 Segment 对象，调用其 remove 方法完成当前操作。

Segment 的 remove 方法进行加锁操作，然后对 hash 值和对象数组大小减 1 的值进行按位与操作，获取数组上对应位置的 HashEntry 对象，接下去遍历此 HashEntry 对象及其 next 对象，找到和传入的 hash 值相等的 hash 值，以及 key 和传入的 key equals 的 HashEntry 对象。如未找到，则返回 null。如找到，则重新创建 HashEntry 链表中位于删除元素之前的所有 HashEntry，位于其后的元素则不用处理。

最后释放锁，完成 remove 操作。

### get(Object key)

首先对 key.hashCode 进行 hash 操作，基于其值找到对应的 Segment 对象，调用其 get 方法完成当前操作。

Segment 的 get 方法首先判断当前 HashEntry 对象数组中已存储的对象是否为 0，如为 0，则直接返回 null，如不为 0，则继续下面的步骤。

对 hash 值和对象数组大小减 1 的值进行按位与操作，获取数组上对应位置的 HashEntry 对象，接下去遍历此 HashEntry 及其 next 对象，寻找 hash 值相等及 key equals 的 HashEntry 对象。在找到的情况下，获取其 value，如 value 不为 null，则直接返回此 value，如为 null，则调用 readValueUnderLock 方法。readValueUnderLock 方法首先进行 lock 操作，然后直接返回 HashEntry 的 value 属性，最后释放锁。

经过以上步骤，`get` 操作就完成了，从上面 `Segment` 的 `get` 步骤来看，仅在寻找到的 `HashEntry` 对象的 `value` 为 `null` 时，才进行锁操作。其他情况下并没有锁操作，也就是可以认为 `ConcurrentHashMap` 在读数据时大部分情况下是没有采用锁的，那么它如何保证并发场景下数据的一致性呢？

对上面的实现步骤进行分析，`get` 操作首先通过 `hash` 值和对象数组大小减 1 的值进行按位与操作来获取数组上对应位置的 `HashEntry`。在这个步骤中，可能会因为对象数组大小的改变，以及数组上对应位置的 `HashEntry` 产生不一致性，那么 `ConcurrentHashMap` 是如何保证的？

对象数组大小的改变只有在 `put` 操作时有可能发生，由于 `HashEntry` 对象数组对应的变量是 `volatile` 类型的，因此可以保证如 `HashEntry` 对象数组大小发生改变，读操作时可看到最新的对象数组大小。

在 `put` 和 `remove` 操作进行时，都有可能造成 `HashEntry` 对象数组上对应位置的 `HashEntry` 发生改变。如在读操作已获取到 `HashEntry` 对象后，有一个 `put` 或 `remove` 操作完成，此时读操作尚未完成，那么这时会造成读的不一致性，但这种几率相对而言非常低。

在获取到了 `HashEntry` 对象后，怎么能保证它及其 `next` 属性构成的链表上的对象不会改变呢？这点 `ConcurrentHashMap` 采用了一个简单的方式，即 `HashEntry` 对象中的 `hash`、`key` 以及 `next` 属性都是 `final` 的，这也就意味着没办法插入一个 `HashEntry` 对象到 `HashEntry` 基于 `next` 属性构成的链表中间或末尾。这样可以保证当获取到 `HashEntry` 对象后，其基于 `next` 属性构建的链表是不会发生变化的。

至于为什么要判断获取的 `HashEntry` 的 `value` 是否为 `null`，原因在于 `put` 操作创建一个新的 `HashEntry` 时，并发读取时有可能 `value` 属性尚未完成设置，因此将读取到默认值，不过具体出现这种现象的原因还未知。据说只在老版本的 JDK 中才会有，而其他属性由于是 `final` 的，所以可以保证是线程安全的，因此这里做了个保护，当 `value` 为 `null` 则通过加锁操作来确保读到的 `value` 是一致的。

### `containsKey(Object key)`

它和 `get` 操作一样，没有进行加锁操作，整个步骤和 `get` 相同，只是当寻找到有 `key`、`hash` 相等的 `HashEntry` 时，才返回 `true`，否则返回 `false`。

### `keySet().iterator()`

它其实也是个类似的读取过程，只是要读取所有分段中的数据，`ConcurrentHashMap` 采用的方法即为遍历每个分段中的 `HashEntry` 对象数组，完成集合中所有对象的读取。这个过程也是不加锁的，因此遍历 `ConcurrentHashMap` 不会抛出 `ConcurrentModificationException`，这点和 `HashMap` 不同。

从上面的分析可以看出，`ConcurrentHashMap` 默认情况下采用将数据分为 16 个段进行存储，并且 16 个段分别持有各自的锁，锁仅用于 `put` 和 `remove` 等改变集合对象的操作，基于 `volatile` 及 `HashEntry` 链表的不变性实现读取的不加锁。这些方式使得 `ConcurrentHashMap` 能够保持极好的并发支持，尤其是对于读远比插入和删除频繁的 `Map` 而言，而它采用的这些方法也可谓是对 Java 内存模型、并发机制深刻掌握的体现，是一个设计得非常不错的支持高并发的集合对象，不过对于 `ConcurrentHashMap` 而言，由于没有一个全局锁，`size` 这样的方法就比较复杂了。在计算 `size` 时，`ConcurrentHashMap` 采取的方法为：

在不加锁的情况下遍历所有的段，读取其 count 以及 modCount，这两个属性都是 volatile 类型的，并进行统计，完毕后，再遍历一次所有的段，比较 modCount 是否有改变。如有改变，则再尝试两次以上动作。

如执行了三次上述动作后，仍然有问题，则遍历所有段，分别进行加锁，然后进行计算，计算完毕后释放所有锁，从而完成计算动作。

以上的方法使得 size 方法在大部分情况下也可通过不加锁的方式计算出来。

## 和 HashMap 的性能对比

采用对 Map 进行测试的场景对 ConcurrentHashMap 进行测试，其在不同的场景下和 HashMap 对比的结果如下。

### 单线程下的运行结果对比

如图 4.17 所示。

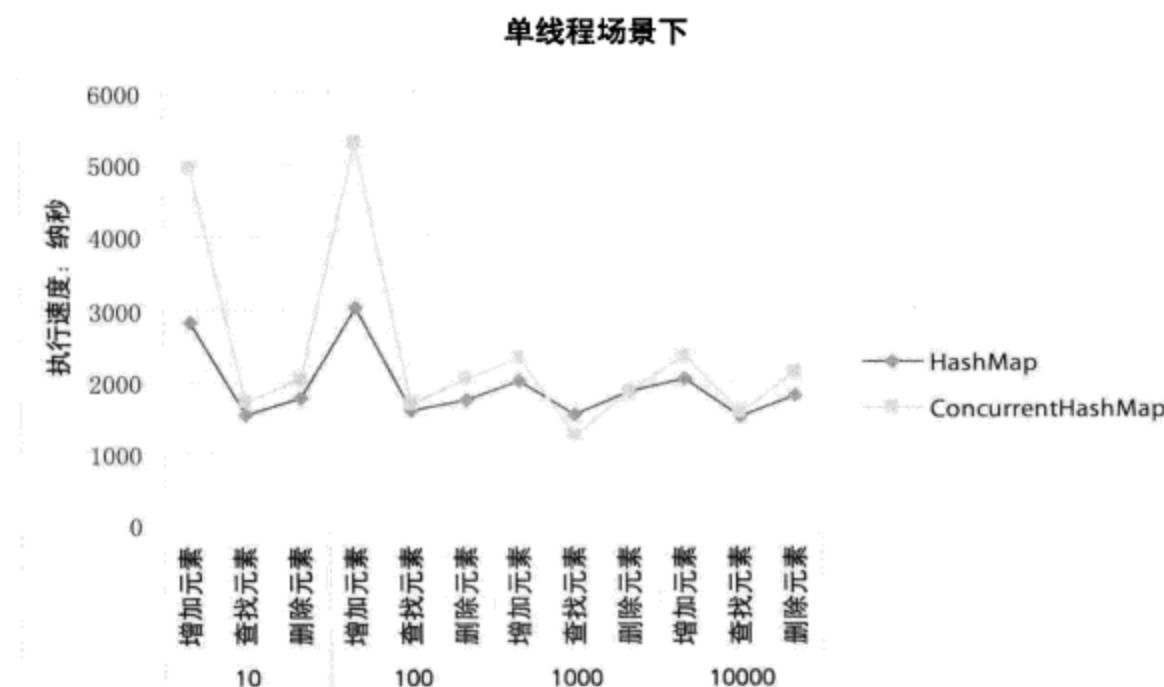


图 4.17 单线程场景下 ConcurrentHashMap 和 HashMap 运行结果的对比

ConcurrentHashMap 的执行速度详细结果如表 4.17 所示：

表 4.17 ConcurrentHashMap 的执行速度数据 单位：纳秒

| 集合元素数量 | 增加元素 | 查找元素 | 删除元素 |
|--------|------|------|------|
| 10     | 4947 | 1699 | 2010 |
| 100    | 5292 | 1661 | 2005 |
| 1000   | 2322 | 1243 | 1842 |
| 10000  | 2351 | 1541 | 2113 |

从运行结果对比来看，单线程场景下，ConcurrentHashMap 的性能稍差于 HashMap，但由于衡量单位为纳秒，可以认为是差不多的。

### 多线程下的运行结果对比

当元素数量为 10 时，在线程数为 10、50、100 的条件下，和 HashMap 的对比结果如图 4.18 所示：

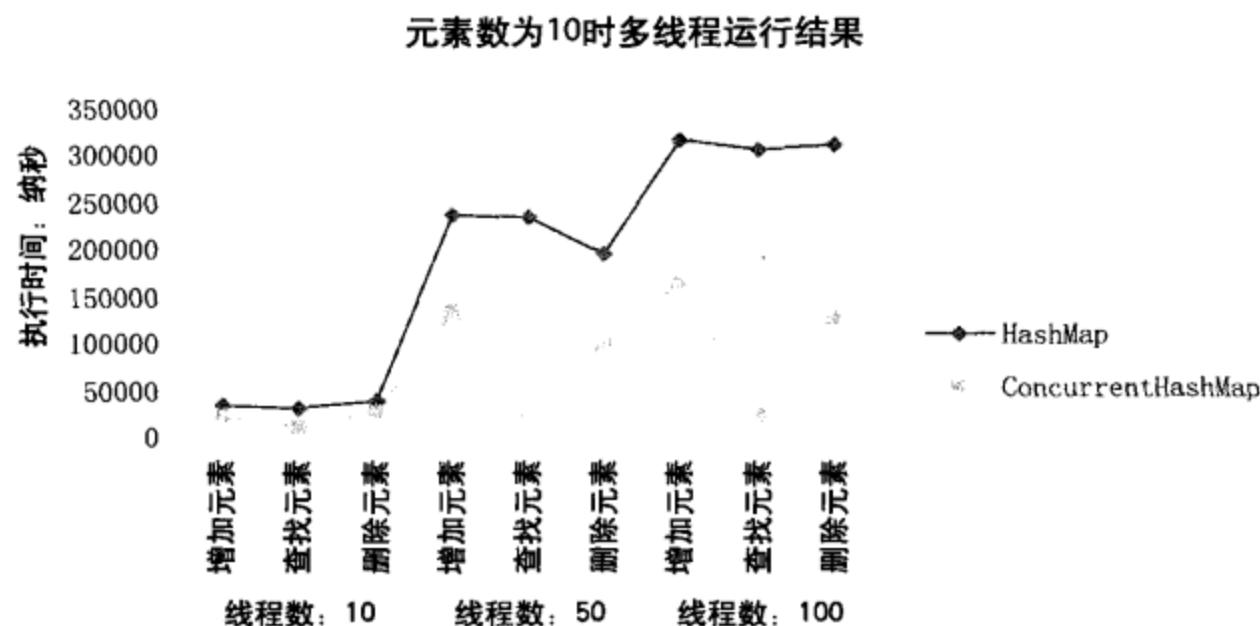


图 4.18 多线程场景，元素数为 10 时 ConcurrentHashMap 和 HashMap 运行结果的对比

当元素数量为 100 时，和 HashMap 的对比结果如图 4.19 所示：

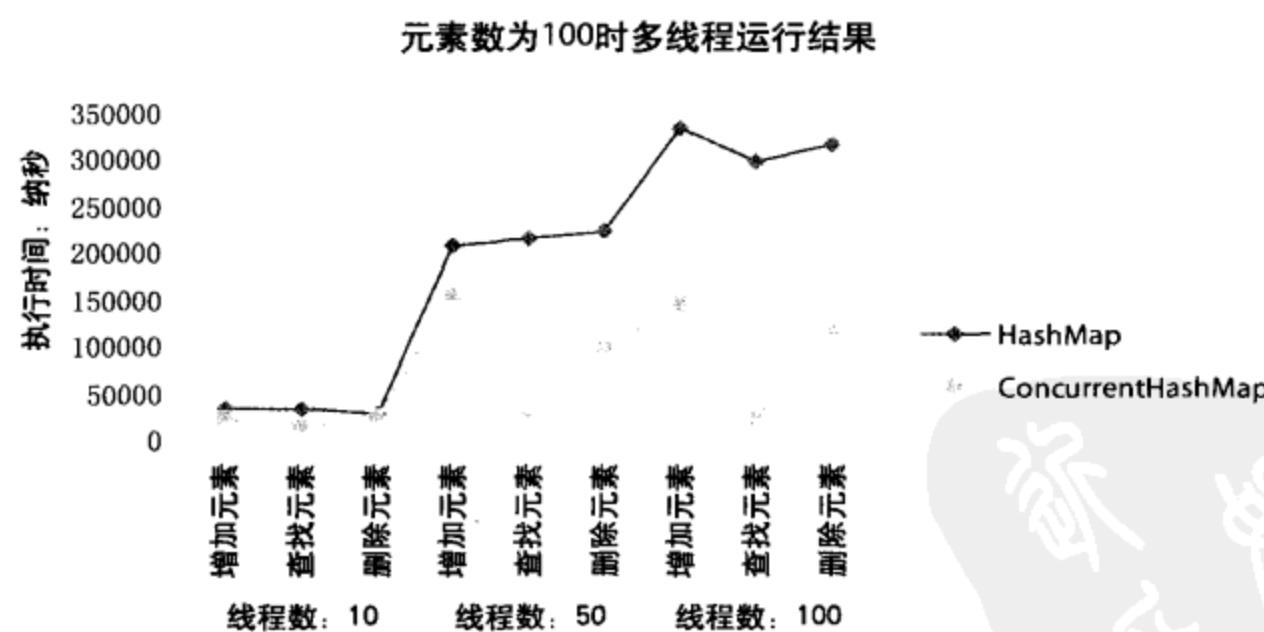


图 4.19 多线程场景，元素数为 100 时 ConcurrentHashMap 和 HashMap 运行结果的对比

当元素数量为 1000 时，和 HashMap 的对比结果如图 4.20 所示：

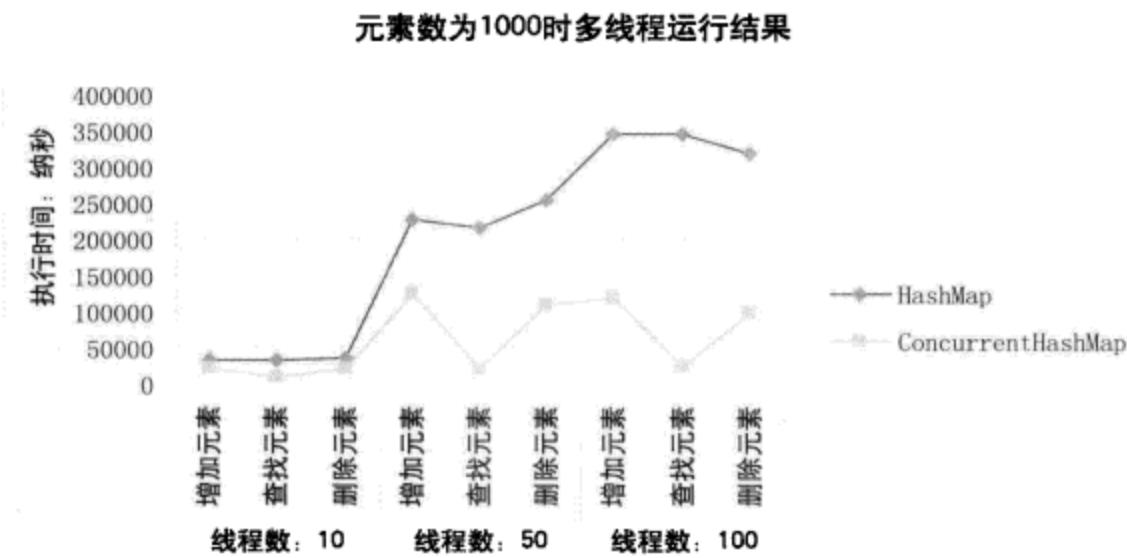


图 4.20 多线程场景，元素数为 1000 时 ConcurrentHashMap 和 HashMap 运行结果的对比

当元素数量为 10 000 时，和 HashMap 的对比结果如图 4.21 所示：

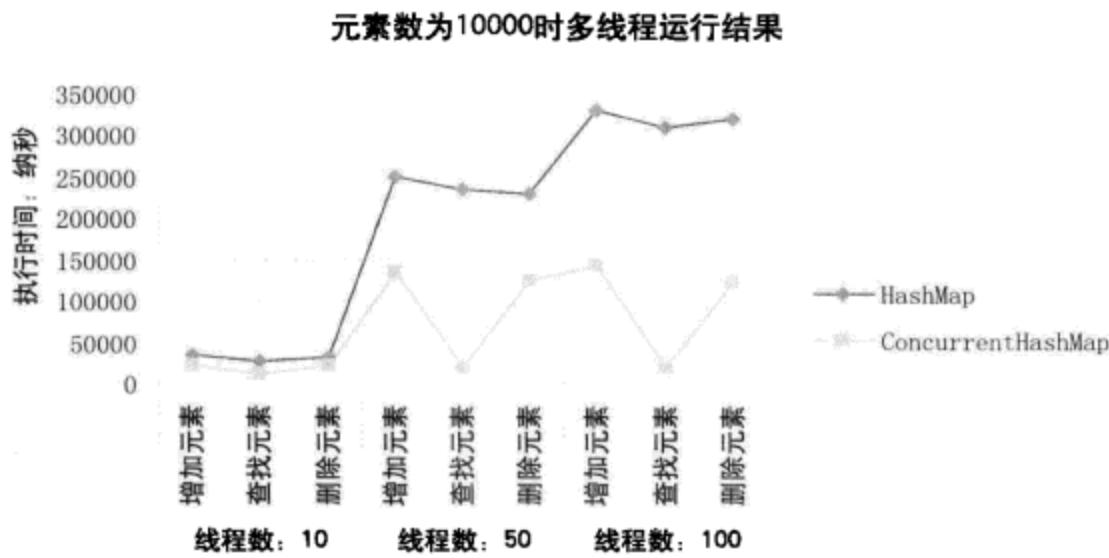


图 4.21 多线程场景，元素数为 10000 时 ConcurrentHashMap 和 HashMap 运行结果的对比

ConcurrentHashMap 在各种元素数量以及线程数量下的详细执行速度结果数据如表 4.18 所示：

表 4.18

单位：纳秒

| 集合元素数量 | 线程数量 | 增加元素   | 查找元素  | 删除元素   |
|--------|------|--------|-------|--------|
| 10     | 10   | 18364  | 8064  | 22086  |
|        | 50   | 131573 | 16778 | 98534  |
|        | 100  | 161866 | 21369 | 122582 |
| 100    | 10   | 21420  | 9932  | 22805  |
|        | 50   | 152609 | 24233 | 96412  |
|        | 100  | 141274 | 21875 | 114333 |

续表

| 集合元素数量 | 线程数量 | 增加元素   | 查找元素  | 删除元素   |
|--------|------|--------|-------|--------|
| 1000   | 10   | 20164  | 7800  | 20875  |
|        | 50   | 123199 | 20156 | 108388 |
|        | 100  | 116758 | 24098 | 97985  |
| 10000  | 10   | 19383  | 10254 | 19483  |
|        | 50   | 134971 | 18799 | 122927 |
|        | 100  | 140902 | 18746 | 120458 |

从上面的运行结果来看，在目前的测试场景中，无论元素数量为多少，在线程数为 10 时，`ConcurrentHashMap` 带来的性能提升不明显。但在线程数为 50 和 100 时，`ConcurrentHashMap` 在增加元素和删除元素时带来了一倍左右的性能提升，在查找元素上更是带来了 10 倍左右的性能提升，并且随线程数增长，`ConcurrentHashMap` 性能并没有出现下降的现象。

根据这样的测试结果，在并发场景中 `ConcurrentHashMap` 较之 `HashMap` 是更好的选择。

## 4.2.2 CopyOnWriteArrayList

### 实现方式

`CopyOnWriteArrayList` 是一个线程安全、并且在读操作时无锁的 `ArrayList`，其具体其实现方法如下。

#### `CopyOnWriteArrayList()`

和 `ArrayList` 不同，此步的做法为创建一个大小为 0 的数组。

#### `add(E)`

`add` 方法并没有加上 `synchronized` 关键字，它通过使用 `ReentrantLock` 来保证线程安全，关于 `ReentrantLock` 在 4.2.12 节中会进一步介绍。

除了使用 `ReentrantLock` 来保证线程安全外，此处和 `ArrayList` 的不同是每次都会创建一个新的 `Object` 数组，此数组的大小为当前数组大小加 1，将之前数组中的内容复制到新的数组中，并将新增加的对象放入数组末尾，最后做引用切换将新创建的数组对象赋值给全局的数组对象。

#### `remove(E)`

和 `add` 方法一样，此方法也通过 `ReentrantLock` 来保证其线程安全，但它和 `ArrayList` 删除元素采用的方式并不一样。

首先创建一个比当前数组小 1 的数组，遍历新数组，如找到 `equals` 或均为 `null` 的元素，则将之后的元素全部赋值给新的数组对象，并做引用切换，返回 `true`；如未找到，则将当前的元素赋值给新的数组对象，最后特殊处理数组中的最后一个元素，如最后一个元素等于要删除的元素，即将当前数组对

象赋值为新创建的数组对象，完成删除动作，如最后一个元素也不等于要删除的元素，那么返回 false。

此方法和 ArrayList 除了锁不同外，最大的不同在于其复制过程并没有调用 System 的 arrayCopy 来完成，理论上来说会导致性能有一定下降。

#### get(int)

此方法非常简单，直接获取当前数组对应位置的元素，这种方法是没有加锁保护的，因此可能会出现读到脏数据的现象。但相对而言，性能会非常高，对于写少读多且脏数据影响不大的场景而言，CopyOnWriteArrayList 是不错的选择。

#### iterator()

调用 iterator 方法后创建一个新的 COWIterator 对象实例，并保存了一个当前数组的快照，在调用 next 遍历时则仅对此快照数组进行遍历，因此遍历 CopyOnWriteArrayList 时不会抛出 ConcurrentModificatiedException。

从以上的分析可见，CopyOnWriteArrayList 基于 ReentrantLock 保证了增加元素和删除元素动作的互斥。在读上没有做任何锁操作，这样就保证了读的性能，带来的副作用是有些时候可能会读取到脏数据。

## 和 ArrayList 的性能对比

采用对 ArrayList 测试的场景对 CopyOnWriteArrayList 进行测试，二者在不同场景下对比的结果如下。

### 单线程下的运行结果对比

如图 4.22 所示。

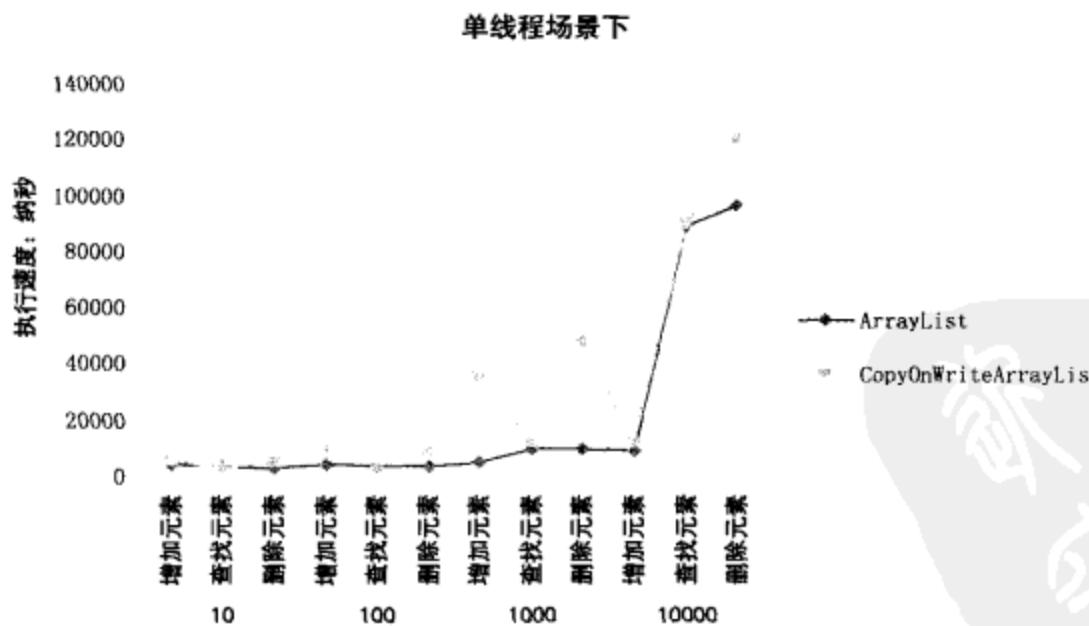


图 4.22 • 单线程下 CopyOnWriteArrayList 与 ArrayList 的对比

CopyOnWriteArrayList 执行速度的详细结果数据如表 4.19 所示：

表 4.19

单位：纳秒

| 集合元素数量 | 增加元素  | 查找元素  | 删除元素   |
|--------|-------|-------|--------|
| 10     | 5109  | 2299  | 3836   |
| 100    | 7882  | 2557  | 7403   |
| 1000   | 34457 | 11852 | 46991  |
| 10000  | 11176 | 88577 | 119500 |

从以上结果来看，在元素数量为 10、100 时，两个类的性能基本一致，在元素数量为 1000、10000 时，CopyOnWriteArrayList 在增加元素和删除元素时性能更差一些。

### 多线程下的运行结果对比

元素数量为 10 时，运行结果对比如图 4.23 所示：

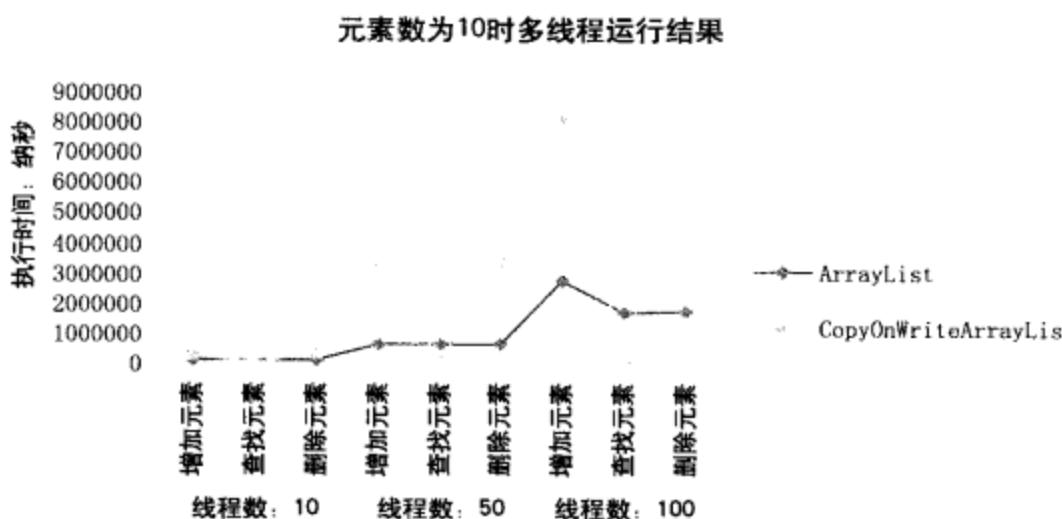


图 4.23 多线程下，元素数为 10 时 CopyOnWriteArrayList 和 ArrayList 的对比

元素数量为 100 时，运行结果对比如图 4.24 所示：

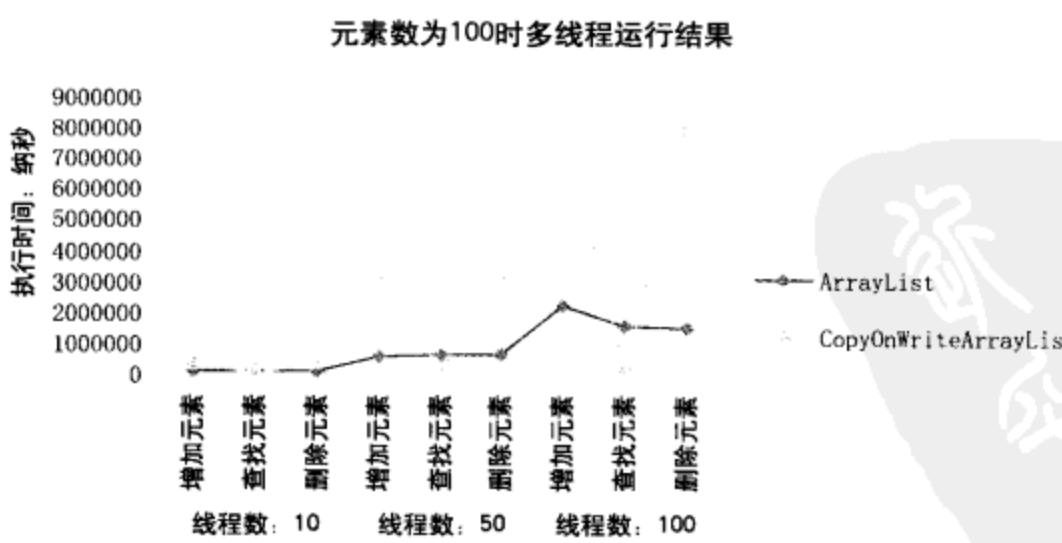


图 4.24 多线程下，元素数为 100 时 CopyOnWriteArrayList 和 ArrayList 的对比

元素数量为1000时，运行结果对比如图4.25所示：

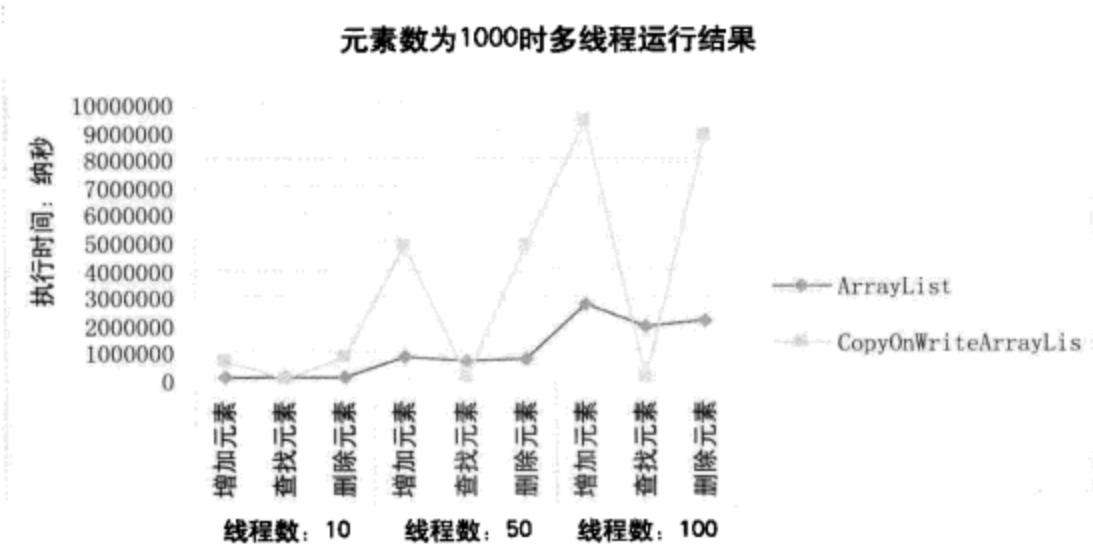


图4.25 多线程下，元素数为1000时 CopyOnWriteArrayList 和 ArrayList 的对比

元素数量为10000时，运行结果对比如图4.26所示：

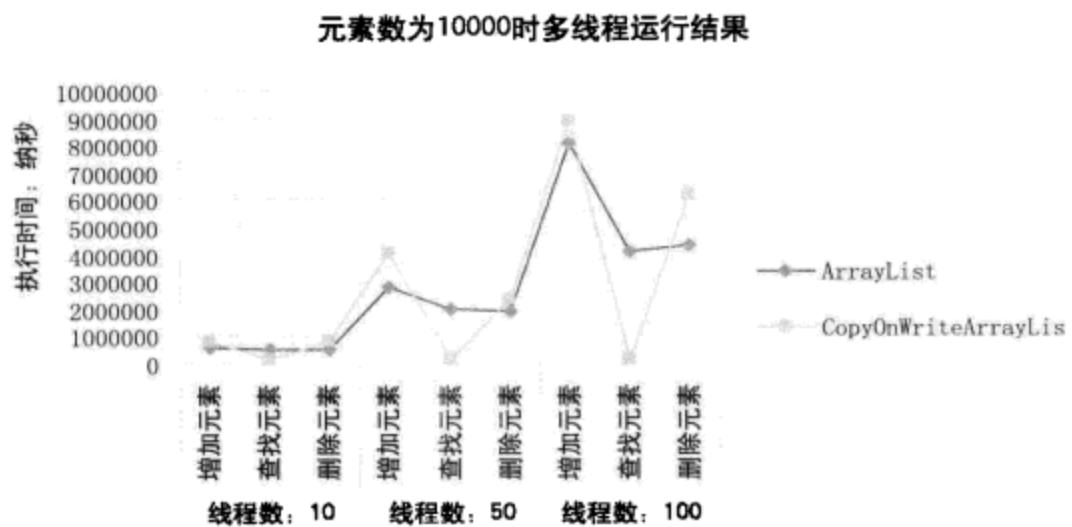


图4.26 多线程下，元素数为10000时 CopyOnWriteArrayList 和 ArrayList 的对比

CopyOnWriteArrayList在各种元素数量以及线程数下执行速度的详细结果数据如表4.20所示：

表4.20

单位：纳秒

| 集合元素数量 | 线程数量 | 增加元素    | 查找元素  | 删除元素    |
|--------|------|---------|-------|---------|
| 10     | 10   | 191805  | 10115 | 244703  |
|        | 50   | 3202174 | 36304 | 3146936 |
|        | 100  | 8027990 | 48569 | 8113666 |
| 100    | 10   | 244893  | 11017 | 274772  |
|        | 50   | 3173129 | 38340 | 3247771 |
|        | 100  | 8219795 | 48093 | 7940006 |

续表

| 集合元素数量 | 线程数量 | 增加元素    | 查找元素   | 删除元素    |
|--------|------|---------|--------|---------|
| 1000   | 10   | 664470  | 23662  | 838285  |
|        | 50   | 4860127 | 49222  | 4851592 |
|        | 100  | 9337542 | 56486  | 8825930 |
| 10000  | 10   | 814755  | 116010 | 782334  |
|        | 50   | 4012258 | 112545 | 2317102 |
|        | 100  | 8872324 | 119931 | 6194978 |

从以上运行结果来看，随着元素数量和线程数量增加，CopyOnWriteArrayList 在增加元素和删除元素时的性能下降非常明显，并且性能会比 ArrayList 低。但在查找元素这点上随着线程数的增长，性能较 ArrayList 会好很多。例如在元素数量为 100、线程数量为 50 时，CopyOnWriteArrayList 查找元素的性能大概为 ArrayList 的 15 倍，当线程数增长到 100 时，更是达到了 31 倍之多。

根据这样的运行结果，在读多写少的并发场景中，CopyOnWriteArrayList 较之 ArrayList 是更好的选择。

### 4.2.3 CopyOnWriteArrayList

CopyOnWriteArrayList 基于 CopyOnWriteArrayList 实现，其唯一的不同是在 add 时调用的是 CopyOnWriteArrayList 的 addIfAbsent 方法。addIfAbsent 方法同样采用锁保护，并创建一个新的大小+1 的 Object 数组。遍历当前 Object 数组，如 Object 数组中已有了当前元素，则直接返回，如没有则放入 Object 数组的尾部，并返回。

从以上分析可见，CopyOnWriteArrayList 在 add 时每次都要进行数组的遍历，因此其性能会略低于 CopyOnWriteArrayList。

### 4.2.4 ArrayBlockingQueue

ArrayBlockingQueue 是一个基于数组、先进先出、线程安全的集合类，其特色为可实现指定时间的阻塞读写，并且容量是可限制的，其具体实现如下。

#### ArrayBlockingQueue(int)

没有默认构造器，传入的参数即为创建的对象数组的大小，同时初始化锁和两个锁上的 Condition，一个为 notEmpty，一个为 notFull。

#### offer(E, long, TimeUnit)

此方法用于插入元素至数组的尾部，如数组已满，则进入等待，直到出现以下三种情况时才继续：被唤醒、到达指定的时间或当前线程被中断（interrupt），来看看具体实现。

此方法首先将指定的时间转换为纳秒，然后加锁，如数组未满，则将对象插入数组，如数组已满，且已超过指定的时间，则返回 false；如未超过指定的时间，则调用 notFull condition 的 awaitNanos 方法进行等待，如为被唤醒或超时，则继续判断数组是否已满；如线程被 interrupt，则直接抛出 InterruptedException。

另一个不带时间的 offer 方法在数组满的情况下则不进入等待，而是直接返回 false，如果有需求，也可以选择调用 put 方法，此方法在数组已满的情况下会一直等待，直到数组不为空或线程被 interrupt。  
**poll(E,long,TimeUnit)**

此方法用于获取队列中的第一个元素，如队列中没有元素，则进入等待，与 offer(E,long,TimeUnit) 相同，它也是在三种情况后继续，来看看具体实现。

将指定的时间转化为纳秒，并加锁，如数组中的元素个数不为零，则从当前的对象数组中获取最后一个元素，在获取后将该位置上的元素设置为 null；如数组中的元素个数为零，首先判断剩余的等待时间是否小于零，如小于则返回 null，如大于则调用 notEmpty condition 的 awaitNanos 方法进行等待，如为被唤醒或超时，则继续判断数组中元素个数是否不为零；如线程被 interrupt，则直接抛出 InterruptedException。

另一个不带时间的 poll 方法在数组中元素个数为零的情况下则不进入等待，而是直接返回 null，如果有需求，也可以选择调用 take 方法，此方法在数组为空的情况下会一直等待，直到数组不为空或线程被 interrupt。

#### **iterator()**

在调用 iterator 方法时，首先进行加锁，然后创建一个 Itr 对象实例，Itr 对象实例在创建时首先获取到当前数组元素尾部的 index 以及尾部的元素，调用完毕后释放锁。

在调用 Itr 的 next 方法时，首先进行加锁，然后获取 Itr 中的 nextItem，增加需要获取的下一元素 index，并检查下一元素是否存在，如存在则获取并赋值到 nextItem，如不存在则将 nextIndex 设置为 -1，nextItem 设置为 null。

根据以上分析可以看出，ArrayBlockingQueue 为一个基于固定大小数组、ReentrantLock 以及 Condition 实现的可阻塞的先进先出的 Queue。

除 ArrayBlockingQueue 之外，BlockingQueue 的实现上常用的还有 LinkedBlockingQueue，LinkedBlockingQueue 实现的不同为采用对象的 next 构成链表的方式来存储对象。由于读只操作队头，而写只操作队尾，这里巧妙地采用了两把锁，对于 put 和 offer 采用一把锁，对于 take 和 poll 则采用另外一把锁，避免了读写时互相竞争锁的现象，因此 LinkedBlockingQueue 在高并发读写操作都多的情况下，性能会较 ArrayBlockingQueue 好很多，在遍历以及删除元素则要两把锁都锁住。

BlockingQueue 的几个实现在 JDK 5 Update 12 以及 JDK 6 Update 2 之前的版本有内存泄漏<sup>5</sup>，因此如需要使用 BlockingQueue，请升级到修复 bug 后的版本。

<sup>5</sup> [http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=6460501](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6460501)

## 4.2.5 AtomicInteger

AtomicInteger 是一个支持原子操作的 Integer 类，在没有 AtomicInteger 前，要实现一个按顺序获取的 ID，就必须在每次获取时进行加锁操作，以避免出现并发时获取到同样 ID 的现象。而在有了 AtomicInteger 后，则无须这么做。以下是一个典型的用 AtomicInteger 实现的顺序 ID 获取的程序片段：

```
private static AtomicInteger counter=new AtomicInteger();
public static int getNextId(){
    return counter.incrementAndGet();
}
```

来具体地看看 AtomicInteger 是如何保证过程的原子性的。

### incrementAndGet()

此方法为先获取当前的 value 属性值，然后将 value 加 1，赋值给一个局部的 next 变量，很明显，这两步都是非线程安全的，关键的方法是如下这行代码：

```
if (compareAndSet(current, next))
    return next;
```

compareAndSet 传入的为执行方法时获取的 value 属性值，next 为加 1 后的值，compareAndSet 调用 Sun 的 UnSafe 的 compareAndSwapInt 方法，此方法是 native 方法，compareAndSwapInt 是基于 CPU 的 CAS 原语<sup>6</sup>来实现的。CAS 简单来说就是由 CPU 比较内存位置上的值是否为当前值，如是则将其设置为 next，如不是则返回 false，因此上面的代码片段要在一个无限循环的代码段中执行，这样可以保证并发时 ID 的顺序。

基于 CAS 的操作可认为是无阻塞的，并且由于 CAS 操作是 CPU 原语，因此其性能会好于之前同步锁的方式。

对 CAS 和同步锁的方式进行测试，测试场景为：启动 50 个线程，每个线程获取 1000 次递增的 ID，统计 CAS 方式和同步锁方式的耗时情况，代码如下：

```
private static int id=0;

private static AtomicInteger atomicId=new AtomicInteger();

private static CountDownLatch latch=null;

public synchronized static int getNextId(){
    return ++id;
}
```

<sup>6</sup> <http://en.wikipedia.org/wiki/Compare-and-swap>

```

public static int getNextIdWithAtomic() {
    return atomicId.incrementAndGet();
}

public static void main(String[] args) throws Exception{
    latch=new CountDownLatch(50);
    long beginTime=System.nanoTime();
    for (int i = 0; i < 50; i++) {
        new Thread(new Task(false)).start();
    }
    latch.await();
    System.out.println("Synchronized style consume time:
"+(System.nanoTime()-beginTime));
    latch=new CountDownLatch(50);
    beginTime=System.nanoTime();
    for (int i = 0; i < 50; i++) {
        new Thread(new Task(true)).start();
    }
    latch.await();
    System.out.println("CAS style consume time:
"+(System.nanoTime()-beginTime));
}

static class Task implements Runnable{

    private boolean isAtomic;

    public Task(boolean isAtomic){
        this.isAtomic=isAtomic;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            if(isAtomic)
                getNextIdWithAtomic();
            else
                getNextId();
        }
        latch.countDown();
    }
}

```

执行上面的测试，结果如下，单位为纳秒。

```
Synchronized style consume time: 26806240
CAS style consume time: 9263488
```

从以上测试可看出，基于 CAS 的方式比使用 synchronized 的方式性能高了 2.8 倍，因此对于使用 JDK 5 以上版本且须支持并发的应用而言，应尽量选择使用 atomic 的类，例如本章节的 AtomicInteger、还有其他的像 AtomicLong、AtomicBoolean、AtomicReference 等。并且同样可以基于这些原子类的 CAS 来自主实现无阻塞的操作。例如在方法中要判断当前方法是否初始化过，如未初始化则执行初始化操作，如已初始化则直接返回，在采用 AtomicBoolean 的情况下，可以这么实现：

```
private AtomicBoolean init=new AtomicBoolean();
private void init(){
    if(init.compareAndSet(false,true)){
        return;
    }
    // 执行初始化操作
}
```

## 4.2.6 ThreadPoolExecutor

ThreadPoolExecutor 是并发包中提供的一个线程池的服务，基于 ThreadPoolExecutor 可以很容易地将一个实现了 Runnable 接口的任务放入线程池中执行，来具体看看 ThreadPoolExecutor 的实现。

- `ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue, ThreadFactory, RejectedExecutionHandler)`  
执行构造器时，所做的动作仅为保存了这些值。
- `execute(Runnable)`

`execute` 方法负责将 `Runnable` 任务放入线程池中执行，其实现方法如下。

首先判断传入的 `Runnable` 实现是否为 `null`，如为 `null`，则抛出 `NullPointerException`，不为 `null`，则继续下面的步骤。

如当前线程数小于配置的 `corePoolSize`，则调用 `addIfUnderCorePoolSize` 方法，`addIfUnderCorePoolSize` 方法首先调用 `mainLock` 加锁。如当前线程数小于 `corePoolSize` 并且线程池处于 `RUNNING` 状态，则调用 `addThread` 增加线程，`addThread` 方法首先创建 `Worker` 对象，然后调用 `threadFactory` 创建新的线程。如创建的线程不为 `null`，则将 `Worker` 对象的 `thread` 设置为为此创建出来的线程，并将此 `Worker` 对象放入 `workers` 中，最后增加当前线程池中的线程数。线程增加后，释放 `mainLock`，最后启动这个新创建的线程，完成 `addIfUnderCorePoolSize` 方法的执行，在 `addIfUnderCorePoolSize` 执行成功的情况下，通过启动新创建的线程来执行传入的 `Runnable` 任务，`Worker` 的 `run` 方法代码如下：

```

try {
    Runnable task = firstTask;
    firstTask = null;
    while (task != null || (task = getTask()) != null) {
        runTask(task);
        task = null;
    }
} finally {
    workerDone(this);
}

```

从以上方法可以看出，Worker所在的线程启动后，首先执行创建其时传入的Runnable任务，在执行完毕该Task后，循环调用ThreadPoolExecutor中的getTask来获取新的Task。在没有Task的情况下，则退出此线程，getTask简单来说，就是通过workQueue的poll（如配置keepAliveTime，则等待指定的时间）或Take来获取要执行的Task。

如当前线程数大于或等于corePoolSize，或addIfUnderCorePoolSize执行失败，则执行后续步骤。

如线程池处于运行状态，且往workQueue中成功放入Runnable任务后，则执行后续步骤，如线程池的状态不为运行状态或线程池中当前运行的线程数为零，则调用ensureQueuedTaskHandled方法并执行。ensureQueuedTaskHandled方法用于确保workQueue中的command被拒绝或被处理，如线程池状态在运行或线程池中当前运行的线程数不为零，则不做任何动作。

如不符合以上步骤的条件，则调用addIfUnderMaximumPoolSize方法，addIfUnderMaximumPoolSize方法的做法和addIfUnderCorePoolSize基本一致，不同点在于根据最大线程数进行比较。如超过最大线程数，则返回false，如addIfUnderMaximumPoolSize返回false，则执行reject方法，调用设置的RejectedExecutionHandler的rejectedExecution方法，ThreadPoolExecutor提供了4种RejectedExecutionHandler的实现：

- CallerRunsPolicy

采用此种策略的情况下，当线程池中的线程数等于最大线程数后，则交由调用者线程来执行此Runnable任务。

- AbortPolicy

采用此种策略，当线程池中的线程数等于最大线程数时，直接抛出RejectedExecutionException。

- DiscardPolicy

采用此种策略，当线程池中的线程数等于最大线程数时，不做任何动作。

- DiscardOldestPolicy

采用此种策略，当线程池中的线程数等于最大线程数时，抛弃要执行的最后一个Runnable任务，并执行此新传入的Runnable任务。

从以上分析来看, JDK 提供了一个高效的支持并发的 ThreadPoolExecutor, 但想使用好这个线程池, 必须合理地配置 corePoolSize、最大线程数、任务缓冲的队列, 以及线程池满时的处理策略。这些需要根据实际的项目需求来决定, 常见的需求有如下两种。

### 1. 高性能

如果希望高性能地执行 Runnable 任务, 即当线程池中的线程数尚未到达最大个数, 则立刻提交给线程执行或在最大线程数量的保护下创建线程来执行。可选的方式为使用 SynchronousQueue 作为任务缓冲的队列, SynchronousQueue 在进行 offer 时, 如没有其他线程调用 poll, 则直接返回 false, 按照 ThreadPoolExecutor 的实现, 此时就会在最大线程数允许的情况下创建线程; 如有其他线程调用 poll, 则返回 true, 按照 ThreadPoolExecutor execute 方法的实现, 采用这样的 Queue 就可实现要执行的任务不会在队列里缓冲, 而是直接交由线程执行。在这种情况下, ThreadPoolExecutor 能支持最大线程数的任务数的执行。

### 2. 缓冲执行

如希望 Runnable 任务尽量被 corePoolSize 范围的线程执行掉, 可选的方式为使用 ArrayBlockingQueue 或 LinkedBlockingQueue 作为任务缓冲的队列。这样, 当线程数等于或超过 corePoolSize 后, 会先加入到缓冲队列中, 而不是直接交由线程执行。在这种情况下, ThreadPoolExecutor 最多能支持的是最大线程数+BlockingQueue 大小的任务数的执行。

以下面的例子来看上面两种配置情况下不同的执行效果。

例子为同时发起 1000 个请求, 这些请求的处理交由 ThreadPoolExecutor 来执行, 每次的处理消耗 3 秒左右, 采用以上第一种配置方式的代码示例如下:

```
final BlockingQueue<Runnable> queue=new SynchronousQueue<Runnable>();

final ThreadPoolExecutor executor=new
ThreadPoolExecutor(10, 600, 30, TimeUnit.SECONDS, queue, Executors.defaultThreadFactory(),new ThreadPoolExecutor.AbortPolicy());

final AtomicInteger completedTask=new AtomicInteger(0);

final AtomicInteger rejectedTask=new AtomicInteger(0);

static long beginTime;

final int count=1000;

/**
 * @param args
 */
public static void main(String[] args) {
    beginTime=System.currentTimeMillis();
```

```

ThreadPoolExecutorDemo demo=new ThreadPoolExecutorDemo();
demo.start();
}

public void start(){
    CountDownLatch latch=new CountDownLatch(count);
    CyclicBarrier barrier=new CyclicBarrier(count);
    for (int i = 0; i < count; i++) {
        new Thread(new TestThread(latch,barrier)).start();
    }
    try {
        latch.await();
        executor.shutdownNow();
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
}

class TestThread implements Runnable{

    private CountDownLatch latch;

    private CyclicBarrier barrier;

    public TestThread(CountDownLatch latch,CyclicBarrier barrier){
        this.latch=latch;
        this.barrier=barrier;
    }

    public void run() {
        try {
            barrier.await();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        try{
            executor.execute(new Task(latch));
        }
        catch(RejectedExecutionException exception){
            latch.countDown();
            System.err.println("被拒绝的任务数为: " + 
                "rejectedTask.incrementAndGet()");
        }
    }
}

```

```

class Task implements Runnable{

    private CountDownLatch latch;

    public Task(CountDownLatch latch){
        this.latch=latch;
    }

    public void run() {
        try {
            Thread.sleep(3000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("执行的任务数为: "+
            completedTask.incrementAndGet());
        System.out.println("任务耗时为:
            "+(System.currentTimeMillis()-beginTime)+" ms");
        latch.countDown();
    }

}
}

```

以上代码执行后被拒绝的任务数输出为 400 次，执行了的任务数为 600 次，任务耗时在 3 200~3 560ms 之间。

将以上代码中的 queue 换成 new ArrayBlockingQueue<Runnable>(10);，再次执行，被拒绝的任务数为输出 390 次，执行的任务数为 610 次，但任务的耗时则在 3 200~6 200ms 之间。

## 4.2.7 Executors

Executors 提供了一些方便创建 ThreadPoolExecutor 的方法，主要有以下几个。

### **newFixedThreadPool(int)**

创建固定大小的线程池，线程 keepAliveTime 为 0，默认情况下，ThreadPoolExecutor 中启动的 corePoolSize 数量的线程启动后就一直运行，并不会由于 keepAliveTime 时间到达后仍没有任务需要执行就退出。缓冲任务的队列为 LinkedBlockingQueue，大小为整型的最大数。当使用此线程池时，在同时执行的 task 数量超过传入的线程池大小值后，将会放入 LinkedBlockingQueue，在 LinkedBlockingQueue 中的 task 需要等待线程空闲后来执行，当放入 LinkedBlockingQueue 中的 task 超过整型最大数时，抛出 RejectedExecutionException。

### newSingleThreadExecutor()

相当于创建大小为1单位的固定线程池，当使用此线程池时，同时执行的task只有1个，其他task都在LinkedBlockingQueue中。

### newCachedThreadPool()

创建corePoolSize为0，最大线程数为整型的最大数，线程keepAliveTime为1分钟，缓存任务的队列为SynchronousQueue的线程池。在使用时，放入线程池的task都会复用线程或启动新线程来执行，直到启动的线程数达到整型最大数后抛出RejectedExecutionException，启动后的线程存活时间为1分钟。

### newScheduledThreadPool(int)

创建corePoolSize为传入参数，最大线程数为整型的最大数，线程keepAliveTime为0，缓存任务的队列为DelayedWorkQueue的线程池。在实际业务中，通常会有一些需要定时或延迟执行的任务，而对于分布式Java应用而言，更为典型的则是在异步操作时需要超时回调的场景。这种情况下ScheduledThreadPoolExecutor是不错的选择，在JDK5以前的版本中更多的是借助Timer来实现，Timer和ScheduledThreadPoolExecutor主要有以下三方面的区别。

- Timer只能单线程，一旦task执行缓慢，就会导致其他的task执行推迟，而如果使用ScheduledThreadPoolExecutor，则可自行控制线程数；
- 当Timer中的task抛出RuntimeException时，会导致Timer中所有的task不再执行；
- ScheduledThreadPoolExecutor可执行Callable的task，从而在执行完毕后得到执行结果。

当要执行的Runnable或Callable的task加入时，ScheduledThreadPoolExecutor会将其放入内部的DelayedWorkQueue中，DelayedWorkQueue又基于DelayQueue来实现；当有新的task加入时，DelayQueue会将其加入内部的数组对象中，并进行排序。对于ScheduledThreadPoolExecutor而言，排序的规则为执行的时间，执行时间越近的排在越前，线程池中的线程在获取要执行的task时，方式为获取最近要执行的task，并调用condition的awaitNanos来等待唤醒。ScheduledThreadPoolExecutor和Timer一样，无法确保task在指定的延时时间点执行，这主要是由于到达时间点的时候CPU可能没有调度到执行task的线程。

## 4.2.8 FutureTask

FutureTask可用于要异步获取执行结果或取消执行任务的场景，通过传入Runnable或Callable的任务给FutureTask，直接调用其run方法或放入线程池执行，之后可在外部通过FutureTask的get异步获取执行结果。FutureTask可以确保即使调用了多次run方法，它都只会执行一次Runnable或Callable任务，或者通过cancel取消FutureTask的执行等。先来看一个FutureTask使用的例子。

假设有一个带key的连接池，当key已存在时，即直接返回key对应的对象；当key不存在时，则创建连接。

对于这样的应用场景，通常采用的方法为使用一个 Map 对象来存储 key 和连接池对象的对应关系，典型的实现代码如下：

```
Map<String, Connection> connectionPool=new HashMap<String, Connection> ();
ReentrantLock lock=new ReentrantLock();
public Connection getConnection(String key){
    try{
        lock.lock();
        if(connectionPool.containsKey(key)){
            return connectionPool.get(key);
        }
        else{
            // create Connection
            connectionPool.put(key, connection);
            return connection;
        }
    }
    finally{
        lock.unlock();
    }
}
```

改用 ConcurrentHashMap 的情况下，几乎可以避免加锁的操作，但在并发情况下有可能出现 Connection 被创建多次的现象。这时最需要的解决方案就是当 key 不存在时，创建 Connection 的动作能放在 connectionPool 之后执行，这正是 FutureTask 发挥作用的时机，基于 ConcurrentHashMap 和 FutureTask 的改造代码如下：

```
Map<String, Connection> connectionPool=new ConcurrentHashMap<String, Connection> ();
public Connection getConnection(String key){
    FutureTask<Connection> connectionTask=connectionPool.get(key);
    if(connectionTask!=null){
        return connectionTask.get();
    }
    else{
        Callable<Connection> callable=new Callable<Connection>{// create Connection};
        FutureTask<Connection> newTask=new FutureTask<Connection>(callable);
        connectionTask=connectionPool.putIfAbsent(key, newTask);
        if(connectionTask==null){
            connectionTask= newTask;
            connectionTask.run();
        }
        return connectionTask.get();
    }
}
```

经过这样的改造，可以避免由于并发带来的多次创建连接及锁的出现。

下面具体来看 FutureTask 的实现。

### FutureTask(Callable)

创建一个内部类 Sync 的对象实例。

#### run()

调用 Sync 对象的 innerRun 方法实现。

Sync 对象的 innerRun 方法首先基于 CAS 将 state 由 0 设置为 RUNNING，如果设置失败，则直接返回，否则继续下面的步骤。

获取当前线程，判断当前 state 是否为 RUNNING，如果是，则调用 innerSet 方法，传入的参数为 callable.call，即执行 callable 任务并返回执行结果；如果不是，则调用 releaseShared 方法。

innerSet 方法首先获取当前 state，如果当前 state 为 RAN，则直接返回；如果当前 state 为 CANCELLED，则调用 releaseShared 方法，传入 0，执行完毕后返回；如果不为以上两种 state，则基于 CAS 将当前 state 设置为 RAN，设置成功则先将 result 设置为 v，然后调用 releaseShared 方法，最后调用 done 方法，执行完毕后返回。

执行 releaseShared 方法时将 runner 属性设置为 null。

#### get(long, TimeUnit)

首先判断当前 state 的状态是否为 RAN 或 CANCELLED，如果满足以上两种状态之一且 runner 属性为 null，则直接进入后续步骤；如果 state 状态不为 RAN、CANCELLED，或者 runner 属性不为 null，则进入等待状态，直到有 run 或 cancel 动作执行才继续后续步骤。

如果 state 为 CANCELLED，则抛出 CancellationException；如果之前执行 callable 时出现了异常，则抛出 ExecutionException；如果不为以上两种情况，则返回 result 属性。

#### cancel(boolean)

调用 Sync 的 innerCancel 方法实现。

获取当前 state，并判断其状态是否为 RAN 或 CANCELLED，如果是，则返回 false；如果不是，则继续下面的步骤。

基于 CAS 将当前 state 设置为 CANCELLED，如果设置失败，则继续以上过程，直到设置成功，才退出循环。

如果传入的参数为 true，则调用 runner 的 interrupt 方法；如果为 false，则直接进入后续步骤，最后调用 releaseShared 方法及 done 方法，完成 cancel 步骤。

## 4.2.9 Semaphore

**Semaphore** 是并发包中提供的用于控制某资源同时被访问的个数的类，例如连接池中通常要控制创建的连接的个数，传统方式下，通常是这样来控制的，代码如下：

```

int maxActive;
int numActive;
int maxWait;
LinkedList pool;
public void get() throws Exception{
    long startTime=System.currentTimeMillis();
    Object object=null;
    for(;;){
        synchronized(this){
            object=pool.removeFirst();
            if((object==null)&&(numActive>=maxActive)){
                long waitTime=maxWait-(System.currentTimeMillis()-startTime);
                wait(waitTime);
                if((System.currentTimeMillis()-startTime)>maxWait){
                    //抛出超时异常
                }
            }else{
                continue;
            }
        }
        numActive++;
    }
    // if needed then create object & validate object
    return object;
}
}

```

而改用 **Semaphore** 后，就无须这么复杂了，改造后的 for 循环中的代码如下：

```

final long elapsed = (System.currentTimeMillis() - starttime);
final long waitTime = maxWait - elapsed;
boolean timeouted=semaphore.tryAcquire(waitTime, TimeUnit.MILLISECONDS);
if(!timeouted)
    throw new NoSuchElementException("Timeout waiting for idle object");
try {
    synchronized (pool) {
        pair = (ObjectTimestampPair)(pool.removeFirst());
    }
}
catch(NoSuchElementException e) {
    ;
}

```

从以上代码可见，改为基于 `Semaphore` 后，同步锁的地方可以少一些，并且代码也简洁不少，具体来看 `Semaphore` 的实现。

#### `Semaphore(int)`

创建一个 `NonfairSync` 的对象实例，并将 `state` 值设为传入的值。

#### `tryAcquire(long, TimeUnit)`

调用 `NonfairSync` 的 `tryAcquireSharedNanos` 方法来完成。

`NonfairSync` 的 `tryAcquireSharedNanos` 方法继承自 `AbstractQueuedSynchronizer`，该方法首先执行 `NonfairSync` 的 `tryAcquireShared` 方法。

`NonfairSync` 的 `tryAcquireShared` 方法通过获取当前的 `state`，以此 `state` 减去需要获取的信号量个数。作为剩余的个数，如果结果小于零，则返回此剩余的个数；如果结果大于零，则基于 CAS 将 `state` 的值设置为剩余的个数，当前步骤只有在结果小于零或设置 `state` 值成功的情况下才会退出。

如果返回的剩余个数大于零，则返回 `true`，完成 `tryAcquire` 动作；如果返回的剩余个数小于零，则进入等待状态，直到有可用的信号量或超时后才返回结果。

#### `release()`

调用 `Semaphore` 中内部类 `Sync` 的 `tryReleaseShared` 方法来完成。

该方法所做的动作为基于 CAS 将 `state` 设置为 `state` 值+1，一直循环确保此 CAS 操作成功，操作成功后返回 `true`，完成整个 `release` 动作。

根据以上分析，可以看出，`Semaphore` 和 `FutureTask` 一样，都充分采用了 CAS 来尽量避免锁的使用，提升高并发下的性能。

### 4.2.10 CountDownLatch

`CountDownLatch` 是并发包中提供的一个可用于控制多个线程同时开始某动作的类，其采用的方式为减计数的方式。当计数减至零时，位于 `latch.await` 后的代码才会被执行，在之前讲 `ThreadPoolExecutor` 的例子已经使用到 `CountDownLatch`，该例子中采用了 `CountDownLatch` 等待所有 task 执行完毕，具体来看 `CountDownLatch` 的实现。

#### `CountDownLatch(int)`

创建内部类 `Sync` 的对象实例，并将 `state` 设置为传入的参数值。

#### `await()`

调用 `Sync` 继承的 `AbstractQueuedSynchronizer` 的 `acquireSharedInterruptibly` 来完成。

`acquireSharedInterruptibly` 首先调用 `Sync` 的 `tryAcquireShared` 方法，该方法判断当前 `state` 是否为零。

如果为零，则返回 1，否则返回-1；如果返回 1，则 await 直接返回；如果返回-1，则将此线程放入队列进行等待，直到 tryAcquireShared 方法返回 1 或线程被 interrupt。

#### countdown()

调用 Sync 的 tryReleaseShared 方法，如果 state 不为零，基于 CAS 将 state 的值设置为减 1 后的值；如果减 1 后的值为零，则返回 true，否则返回 false。

如果为 true，则通知所有在队列中等待的线程。

### 4.2.11 CyclicBarrier

CyclicBarrier 和 CountDownLatch 不同，CyclicBarrier 是当 await 的数量到达了设定的数量后，才继续往下执行。在之前讲 ThreadPoolExecutor 的例子曾使用到 CyclicBarrier，例子中采用了 CyclicBarrier 来确保所有的线程几乎同时开始运行，具体来看 CyclicBarrier 的实现。

#### CyclicBarrier(int)

设置 parties、count 及 barrierCommand 属性。

另外一个构造器允许传入一个实现了 Runnable 的对象，当 await 的数量到达了设定的数量后，会首先执行此 Runnable 的对象。

#### await()

首先进行加锁操作，然后对 count 属性执行减 1 操作，如果减后的值等于零，则执行传入的 Runnable 对象。执行完毕后将 ranAction 设置为 true，调用 nextGeneration 方法并返回零，nextGeneration 方法主要调用 trip Condition 的 signalAll；如果 count 减 1 后的值不等于零，则调用 trip Condition 的 await 或 await（设定的时间）进入等待状态，直到 trip Condition 被唤醒、线程被 interrupt 或超过设定的时间，从等待状态恢复后；如果设定了等待的时间，则检查是否超过了等待的时间；如果超过了，则将 generation 的 broken 属性设置为 true，调用 trip Condition 的 signalAll，并抛出 TimeoutException。

根据以上分析可以看出，CyclicBarrier 和 CountDownLatch 在实现上也不同，它基于 ReentrantLock 和 Condition 来实现。

### 4.2.12 ReentrantLock

ReentrantLock 是并发包中提供的一个更为方便的控制并发资源的类，且和 synchronized 语法达到的效果是一致的。在分析如 CopyOnWriteArrayList 等时都可看到对于 ReentrantLock 的使用，具体来看 ReentrantLock 的实现。

#### ReentrantLock()

创建一个内部类 NonfairSync 的对象实例，NonfairSync 继承自内部的 Sync 类，而 Sync 类继承了

`AbstractQueuedSynchronizer` 类。也可调用带 `boolean` 参数的构造器，通过传入 `true` 来创建内部类 `FairSync` 的对象实例，`FairSync` 同样继承自内部的 `Sync` 类。

#### lock()

调用构造器时创建的 `sync` 对象实例的 `lock` 方法。

`NonfairSync` 的 `lock` 方法的实现方式为：首先基于 CAS 将 `state` 从 0 设置为 1，如果设置成功，则将当前线程设置为 `exclusiveOwnerThread`，说明此时并没有其他线程持有锁；如果设置失败，则表明在当时已经有其他线程拥有了锁，可再次获取 `state`；如果为 0，则继续尝试将 `state` 设置为 1，成功则设置为 `exclusiveOwnerThread`；如果 `state` 不为 0，则判断当前线程是否为 `exclusiveOwnerThread`；如果是，则将当前 `state` 改变为当前 `state` 值+1，否则将当前线程放入 `Queue` 并挂起，挂起线程的方法为：`Unsafe.getUnsafe().park(false,0L);`。

`FairSync` 的 `lock` 方法的实现方式和 `NonfairSync` 的区别在于它并不尝试基于 CAS 将 `state` 从 0 设置为 1，而是直接进入 `NonfairSync` 在设置失败后的处理步骤。

#### unlock()

`NonfairSync` 和 `FairSync` 的实现相同，方式为获取现在的 `state`，以此 `state` 值减去释放的锁的个数。如果减出来后的值为 0，则释放锁，并通知等待在 `Queue` 上的线程，并将等待在 `Queue` 上的第一个线程恢复，恢复的方法为：`Unsafe.getUnsafe().unpark(thread)`。

从以上分析可见，`ReentrantLock` 也是一个基于 `AbstractQueuedSynchronizer` 的实现，`AbstractQueuedSynchronizer` 为基于整数状态值实现资源的并发控制访问提供了很好的支持。

### 4.2.13 Condition

`Condition` 是并发包中提供的一个接口，典型的实现有 `ReentrantLock`，`ReentrantLock` 提供了一个 `newCondition` 的方法，以便用户在同一个锁的情况下可以根据不同的情况执行等待或唤醒动作。典型的用法可参考 `ArrayBlockingQueue` 的实现，下面来看 `ReentrantLock` 中 `newCondition` 的实现。

#### ReentrantLock.newCondition()

创建一个 `AbstractQueuedSynchronizer` 的内部类 `ConditionObject` 的对象实例。

#### ReentrantLock.newCondition().await()

将当前线程加入此 `condition` 的等待队列中，并将线程置为等待状态。

#### ReentrantLock.newCondition().signal()

从此 `condition` 的等待队列中获取一个等待节点，并将节点上的线程唤醒，如果要唤醒全部等待节点的线程，则可调用 `signalAll` 方法。

## 4.2.14 ReentrantReadWriteLock

ReentrantReadWriteLock 和 ReentrantLock 没有任何继承关系，ReentrantReadWriteLock 提供了读锁（ReadLock）和写锁（WriteLock），相比较 ReentrantLock 只有一把锁的机制而言，读写锁分离的好处是在读多写少的场景中可大幅提升读的性能。在实现上 ReentrantReadWriteLock 主要为基于 AbstractQueuedSynchronizer 来实现，自行实现判断是否可获取读锁或写锁。当调用读锁的 lock 方法时，如果没有线程持有写锁，就可获得读锁，这也意味着只要进行读的时候没有其他线程在进行写操作，读的操作就是无阻塞的；当调用写锁的 lock 方法时，如果此时没有线程持有读锁或写锁，则可继续执行，这也意味着要进行写动作时，如果有其他线程在读或在写，就会被阻塞，因此写的性能有可能会下降。

读写锁在使用时尤其要注意锁的升级和降级机制：

- 在同一线程中，持有读锁后，不能直接调用写锁的 lock 方法。按照上面的机制说明可以看出，如果这样操作，会造成死锁，这也通常称为读锁不可升级；
- 在同一线程中，持有写锁后，可调用读锁的 lock 方法，在此之后如果调用写锁的 unlock 方法，那么当前锁将降级为读锁。

读写锁在使用时要注意其中一个最典型的 bug：当没有线程持有锁时被挂起<sup>7</sup>，这个 bug 在 JDK 6 update 18 中已修复，对于读多写少的场景而言，读写锁是一个可选的用于实现高性能并发的机制。

以一段代码来测试 ReentrantReadWriteLock 的性能，测试场景为同时启动 102 个线程，其中 100 个进行读操作，2 个进行写操作，代码如下：

```
private static ReentrantReadWriteLock lock=new ReentrantReadWriteLock();
private static WriteLock writeLock=lock.writeLock();
private static ReadLock readLock=lock.readLock();
private static Map<String, String> maps=new HashMap<String, String>();
private static CountDownLatch latch=new CountDownLatch(102);
private static CyclicBarrier barrier=new CyclicBarrier(102);
public static void main(String[] args) throws Exception{
    long beginTime=System.currentTimeMillis();
    for (int i = 0; i < 100; i++) {
        new Thread(new ReadThread()).start();
    }
    for (int i = 0; i < 2; i++) {
        new Thread(new WriteThread()).start();
    }
    latch.await();
    long endTime=System.currentTimeMillis();
    System.out.println("Consume Time is: "+(endTime-beginTime)+" ms");
}
```

<sup>7</sup> [http://bugs.sun.com/view\\_bug.do?bug\\_id=6822370](http://bugs.sun.com/view_bug.do?bug_id=6822370)

```

}

static class WriteThread implements Runnable{
    @Override
    public void run() {
        try{
            barrier.await();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        try{
            writeLock.lock();
            maps.put("1", "2");
            Thread.sleep(100);
        }
        catch(Exception e){
            e.printStackTrace();
        }
        finally{
            writeLock.unlock();
        }
        latch.countDown();
    }
}

static class ReadThread implements Runnable{

    @Override
    public void run() {
        try{
            barrier.await();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        try{
            readLock.lock();
            maps.get("1");
            Thread.sleep(100);
        }
        catch(Exception e){
            e.printStackTrace();
        }
        finally{
            readLock.unlock();
        }
    }
}

```

```

        latch.countDown();
    }

}

```

执行上面的代码，结果为 310ms，将上面的读写锁改为使用 ReentrantLock，运行后的结果为 10209ms，可见在这类场景下读写锁带来的性能提升是显著的。

以上分析了 JDK 5 以后版本提供的并发包中一些常用类的实现方法，这些类能够帮助开发者更好地控制高并发下的资源操作，尽可能地避免出现不一致及资源锁竞争激烈等现象。总的来说，基于 CAS、拆分锁、 volatile 及 AbstractQueuedSynchronizer 是这些类的主要实现方法，这些方法都是为了尽量减少高并发时的竞争现象，对于实际编写代码时有一定的参考价值，从而保障程序即使在高并发时也能保持较高的性能。

## 4.3 序列化/反序列化

对于 Java 的网络通信而言，将对象转化为流然后进行网络传输是最基本也最常用的方法，而要把对象转化为流及将流还原为对象，最常用的方法就是 Java 自带的序列化，下面介绍 Java 序列化及反序列化的实现。

### 4.3.1 序列化

将 Java 对象转化为字节流常用的方法为：

```

// 创建一个字节数组输出流
ByteArrayOutputStream output=new ByteArrayOutputStream();
// 将字节数组输出流包装为 ObjectOutputStream
ObjectOutputStream objectOut=new ObjectOutputStream(output);
// 将对象写入 ObjectOutputStream
objectOut.writeObject(object);
objectOut.close();
output.close();
// 返回字节数组输出流中的字节数组
return output.toByteArray();

```

下面来看这个过程中 ObjectOutputStream 的具体实现。

#### ObjectOutputStream(OutputStream)

该步中最为关键的是往流中写入 STREAM\_MAGIC 和 STREAM\_VERSION 两个头信息。

#### writeObject(Object object)

在 writeObject 时，首先要做的是获取当前序列化对象的类信息，在获取类信息时，调用的是 ObjectStreamClass 的 lookup 方法。

ObjectStreamClass 采用内部的 Caches 类来缓存类信息，key 为 WeakClassKey 对象，value 为 SoftReference 对象，SoftReference 对象指向 EntryFuture 对象，基于这样的结构，ObjectStreamClass 在实现 lookup 时采用如下步骤。

首先从缓存中获取当前类构成的 WeakClassKey 对象为 key 值，如果值不为 null，则执行此 Reference 的 get 方法；如果为 null，或者 get 方法返回的值为 null，则创建 EntryFuture 对象，并创建一个指向此 EntryFuture 的 SoftReference 对象，将它放入缓存中。

判断此时获取的 entry 对象是否为 ObjectStreamClass，如果是，则直接返回；如果为 EntryFuture 对象，则判断 Future 对象创建的线程是否为当前线程。如果是，则将 entry 设置为 null；如果不是，则执行 EntryFuture 的 get 方法。此 get 方法将阻塞线程，直到有其他线程设置 EntryFuture 的值。

如果 entry 对象为 null，则创建 ObjectStreamClass 对象。ObjectStreamClass 对象在创建时要读取序列化所需的一些类信息，包括：

- 当前类是否为 Proxy 类；
- 是否为 Enum 类型的类；
- 是否为 Serializable 类型的类；
- 是否为 Externalizable 类型的类；
- 获取其父类及父类的信息。

如果为 Serializable 类型的类，则继续读取以下的步骤：

- 对于 Enum 类型的类，生成一个值为 0 的 uid，并将 fields 设置为空的 ObjectStreamField 数组；
- 对于 Array 类型的类，将 fields 设置为空的 ObjectStreamField 数组；

对于非以上两种类型的类，则继续以下步骤：

- 获取类中定义的 serialVersionUID 属性的值，如果未定义，则返回 null；
- 如果类实现了 Serializable，并且不是 Externalizable 类型、Proxy 类型及接口类型，则获取类中定义的要序列化的属性域（可通过属性 serialPersistentFields 来定义要序列化的 Field，或者非 static 和非 transient 类型的 Field，这也意味着 static 或 transient 的 Field 不会被序列化）；
- 计算 Field 要占用的空间的大小，以及类中对象引用属性（非基本对象）的个数；
- 如果类实现了 Externalizable 接口，则获取其构造器；如果类未实现 Externalizable 接口，则获取构造器、私有的 writeObject 方法、readObject 方法和 readObjectNoData 方法；
- 获取 writeReplace 方法及 readResolve 方法。

如果类未实现 Serializable 接口，则将其 uid 设置为 0，并将 fields 设置为空的 ObjectStreamField 数组。

完成了以上步骤后，根据 fields 获取其对应的 Reflector，对于 fields 的 Reflector，采用和 ObjectStreamClass

缓存类信息同样的缓存结构。

最后根据要序列化的 Object 类型调用相应的方法进行序列化，在此以普通对象的序列化来进行分析，主要执行以下步骤。

- 检查类是否可序列化；
- 向流中写入对象类型的字节；
- 向流中写入类信息，在写入类信息时如果发现类中没有 uid，则自动生成一个 uid，uid 是根据类签名信息来生成的，包括构造器、类接口、类中的属性等，也就是只要类签名信息相同，那么生成的 uid 就是相同的；
- 如果类实现了 Externalizable 接口且不是 proxy 类型，则调用对象实现的 writeExternal 方法来完成写入；如果不是以上类型，则首先判断是否有 writeObject 方法；如果有，则调用对象自身的 writeObject 方法；如果无 writeObject 方法，则根据要序列化的 Field 信息进行序列化，调用同样 ObjectOutputStream 的 writeObject0 方法；如果 Field 为对象引用，则仅写入一个引用的头信息及相应引用对象的位置信息。

经过以上步骤，就完成了将一个对象转化为二进制流的过程，通过以上分析可见，在采用 Java 序列化的情况下，类必须实现 Serializable 接口或 Externalizable 接口。可通过实现 Externalizable 接口、编写私有的 writeObject 或 writeReplace 方法，给属性增加 transient 或 serialPersistentFields 属性这些方法来控制序列化时的行为。在使用 ObjectOutputStream 将对象转换为流后，要注意调用 reset 或 close，以避免产生内存泄露<sup>8</sup>。

由于 Java 在将类信息写入流时，直接采用全类名的写法，在类中结构比较复杂的情况下，写出来的流会相对比较大，这对网络传输来说会造成一定的压力，因此在业界除了 Java 序列化外，还有像 hessian<sup>9</sup>、phprpc<sup>10</sup>、google protocol buffers<sup>11</sup>等方式的序列化，用于提升序列化的性能及降低序列化后流的大小。同时这几种方式也更有利与跨语言交互的实现，但在使用 Hessian 3.1.1 以前的版本时，要特别注意其未加锁操作 Hashmap 的 get 和 put 时可能会导致 CPU 消耗 100% 的 bug<sup>12</sup>。

### 4.3.2 反序列化

反序列化的操作为读取流并重新组装为 Java Object，常用的方法为：

<sup>8</sup> <http://www.ibm.com/developerworks/cn/java/j-lo-streamleak/index.html?ca=drs-cn-0429>

<sup>9</sup> <http://hessian.caucho.com/>

<sup>10</sup> [http://www.phprpc.org/zh\\_CN/](http://www.phprpc.org/zh_CN/)

<sup>11</sup> <http://code.google.com/apis/protocolbuffers/>

<sup>12</sup> <http://bugs.caucho.com/view.php?id=1588>

```

ByteArrayInputStream in=new ByteArrayInputStream(bytes);
ObjectInputStream oin=new ObjectInputStream(in);
Object object=oin.readObject();
oin.close();
in.close();

```

下面来看这个过程中 `ObjectInputStream` 的具体实现。

### `ObjectInputStream(InputStream)`

该步的操作主要为读取流中的前两个 `short`，并检查其是否等于 `STREAM_HEAD` 和 `STREAM_VERSION`，如果不等，则抛出 `StreamCorruptedException` 异常。

#### `readObject()`

读取普通类型的对象是将流转化为对象最复杂的步骤，首先要做的是读取流中的类名、`suid`、是否有 `writeObject` 方法、是否为 `Enum` 类型及有多少 `Field` 等，读取完毕后尝试基于类名创建 `Class` 对象，如果找不到相应的类，则抛出 `ClassNotFoundException`；如果找到相应的类，则继续以下步骤。

根据创建的 `Class` 对象获取其类信息，获取类信息的方法和序列化时相同，获取完毕后进行如下的检查。

- 如果加载的类为 `Proxy` 类型的类，则抛出 `InvalidClassException`，异常信息为：cannot bind non-proxy descriptor to a proxy class；
- 如果从流中读取的 `Enum` 类型值和当前加载类的 `Enum` 类型的值不同，则抛出 `InvalidClassException`，异常信息为：cannot bind enum descriptor to a non-enum class 或 cannot bind non-enum descriptor to an enum class；
- 如果流中读取的为 `Serializable` 类型，当前加载的类也为 `Serializable` 类型且不是 `Array` 类型。在这种情况下，如果流中读取的 `suid` 和当前加载的类的 `suid` 值不等，同样抛出 `InvalidClassException`，异常信息为：local class incompatible: stream classdesc serialVersionUID = 流中的 `suid`, local class `serialVersionUID` = 当前类的 `suid`；
- 如果流中读取的不是 `Enum` 类型，且流中的 `externalizable` 类型的值和当前加载的类的 `externalizable` 类型的值不相同，则抛出 `InvalidClassException`，异常信息为：Serializable incompatible with Externalizable；如果流中的 `serializable` 类型和当前加载类的 `serializable` 类型不同，或者流中的 `externalizable` 类型和当前加载类的 `externalizable` 类型不同，同样抛出 `InvalidClassException`，异常信息为：class invalid for deserialization。

完成了这些检查后，就去获取当前类的构造器、`writeObject` 方法、`readObject` 方法、`readObjectNoData` 方法、`writeReplace` 方法及 `readResolve` 方法，最后获取对应的 `Field` 的 `Reflector`，接着继续后续步骤。

基于获取的空构造器或创建的空构造器创建对象实例。

对于实现了 Externalizable 接口的类，可直接提交由类对象实例的 readExternal 方法来完成；对于未实现 Externalizable 接口的类，如果类中有 readObject 方法，则交由对象实例的 readObject 方法来完成；如果类中没有 readObject 方法，则按照序列化流中序列化的 Field 进行读取。

经过以上步骤，反序列化过程得以完成，和序列化过程一样，反序列化也可通过多种方式来控制，可通过实现 Externalizable 接口、提供私有的 readObject 方法或继承 readResolve 方法来控制反序列化读取对象值的过程，还可通过覆盖 resolveClass 方法来控制反序列化时类的加载。

一个简单的测试 Java 序列化/反序列化的例子如下。

```
public class AObject implements Serializable{
    private String a="java";
    private String b="bluedavy";
    private String c="chapter4";
    private String d="hello";
    private int i=100;
    private int j=10;
    private long m=100L;
    private boolean isA=true;
    private boolean isB=false;
    private boolean isC=false;
    private BObject object=new BObject();
    private BObject bobject=new BObject();
    private BObject cobject=new BObject();
    private BObject dobject=new BObject();
}
public class BObject implements Serializable{}
```

测试性能的类的代码如下：

```

public static void main(String[] args) throws Exception{
    for(int i=0;i<10;i++){
        AObject object=new AObject();
        long beginTime=System.currentTimeMillis();
        ByteArrayOutputStream byteOutput=new ByteArrayOutputStream();
        ObjectOutputStream objectOutput=new ObjectOutputStream(byteOutput);
        objectOutput.writeObject(object);
        objectOutput.close();
        byteOutput.close();
        byte[] bytes=byteOutput.toByteArray();
        System.out.println("Java 序列化耗时:
            "+(System.currentTimeMillis()-beginTime)+"ms");
        System.out.println("Java 序列化后的字节大小为: "+bytes.length);;

        beginTime=System.currentTimeMillis();
        ByteArrayInputStream byteIn=new ByteArrayInputStream(bytes);
        ObjectInputStream objectInput=new ObjectInputStream(byteIn);
        objectInput.readObject();
        objectInput.close();
        byteIn.close();
        System.out.println("Java 反序列化耗时:
            "+(System.currentTimeMillis()-beginTime)+"ms");
    }
}

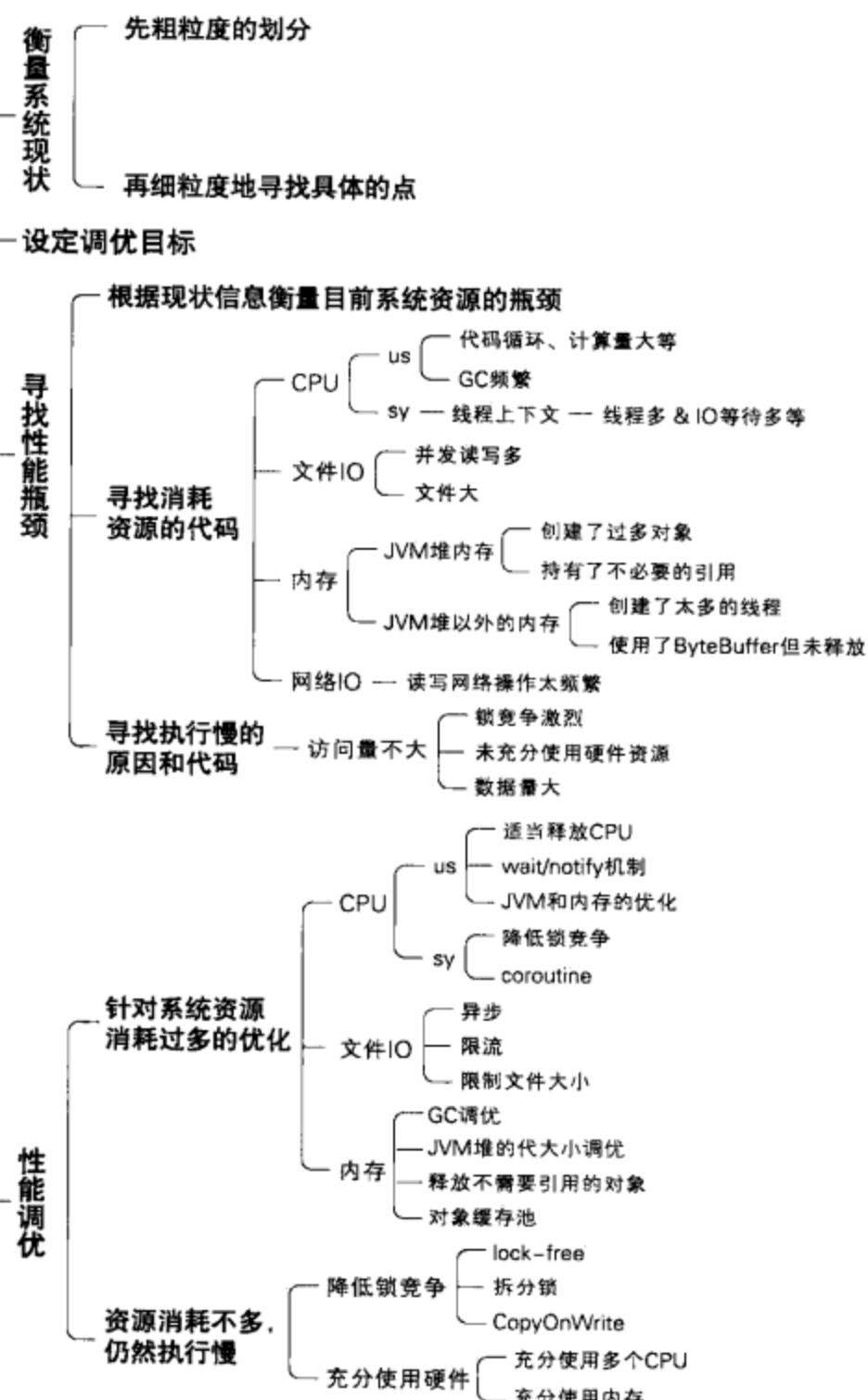
```

在作者的电脑上运行以上代码，可看到在第一次运行时序列化/反序列化差不多要消耗十多 ms 的时间。但在之后的运行中则可以很快完成序列化/反序列化动作，这主要是由于在第一次运行后类信息被缓存了的原因。

本章对 Sun JDK 中分布式应用包中常用的类进行了分析，并做了一些基准性质的性能测试，在实际场景中当使用 Sun JDK 中的其他类时，也应深入查看并掌握其实现，以便判断这些使用到的类在实际场景中运行时会是什么样的表现。这样对于编写出高性能和高可用的程序会更有帮助，除了掌握编写高性能程序的 JVM、JDK 知识外，对于高性能而言，还要掌握如何进行性能调优，下一章将介绍性能调优。

# 第5章 性能调优

## 性能调优



读书笔记  
老曾  
PDG

随着系统数据量的不断增长，访问量的不断提升，系统的响应通常会越来越慢，又或是编写的新应用在性能上无法满足需求，这个时候需要对系统的性能进行调优。调优过程是一个相当复杂的过程，涉及很多的方面：硬件、操作系统、运行环境软件以及应用本身，通常调优的步骤如图 5.1 所示：

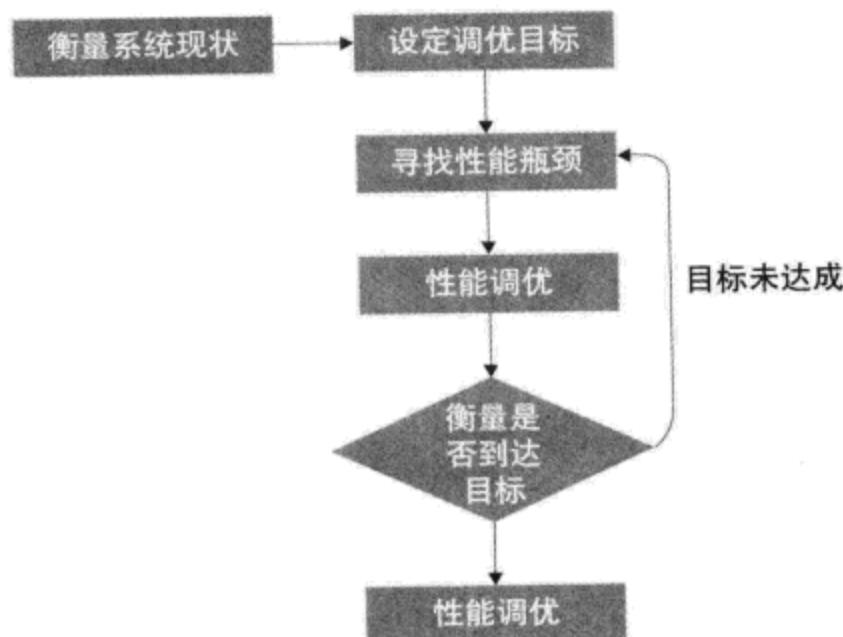


图 5.1 调优步骤

调优前首先要做的是衡量系统现状，这包括目前系统的请求次数、响应时间、资源消耗等信息，例如 A 系统目前 95% 的请求响应时间为 1 秒。

在有了系统现状后可设定调优目标，通常调优目标是根据用户所能接受的响应速度或系统所拥有的机器以及所支撑的用户量制定出来的，因此通常会设定出调优目标：95% 的请求要在 500ms 内返回。

在设定了调优的目标后，需要做的是寻找出性能瓶颈，这一步最重要的是找出造成目前系统性能不足的最大瓶颈点。找出后，可结合一些工具来找出造成瓶颈点的代码，到此才完成了这个步骤。

在寻找到了造成瓶颈点的代码后，通常需要分析其需求场景，然后结合一些优化的技巧制定优化的策略。优化策略或简或繁，选择其中收益比（优化后的预期效果/优化需要付出的代价）最高的优化方案，进行优化。

优化部署后，继续衡量系统的状况，如已达到目标，则可结束此次调优；如仍未达到目标，则要看是否产生了新的性能瓶颈。或可以考虑继续尝试上一步中制定的其他优化方案，直到达成调优目标或论证在目前的体系结构上无法达成调优目标为止。

Facebook 为了将其网站的访问速度提升两倍<sup>1</sup>，在上面的寻找性能瓶颈、性能优化、衡量是否达到调优的目标中循环了多次，最终在花费了 6 个月的时间后终于达到了调优的目标。

本章主要介绍如何寻找性能的瓶颈以及性能调优常用的一些方法。

<sup>1</sup> [http://www.facebook.com/note.php?note\\_id=307069903919](http://www.facebook.com/note.php?note_id=307069903919)

## 5.1 寻找性能瓶颈

通常性能瓶颈的表象是资源消耗过多、外部处理系统的性能不足，或者资源消耗不多，但程序的响应速度却仍达不到要求。

资源主要消耗在 CPU、文件 IO、网络 IO 以及内存方面，机器的资源是有限的，当某资源消耗过多时，通常会造成系统的响应速度慢。

外部处理的性能不够主要是所调用的其他系统提供的功能或数据库操作的响应速度不够，所调用的其他系统性能不足，多数情况下也是资源消耗过多，但程序的性能不足造成的；数据库操作性能不足通常可以根据数据库的 sql 执行速度、数据库机器的 IOPS、数据库的 Active Sessions 等分析出来。

资源消耗不多，但程序的响应速度仍达不到要求的主要原因是程序代码运行效率不够高、未充分使用资源或程序结构不合理。

对于 Java 应用而言，寻找性能瓶颈的方法通常为首先分析资源的消耗，然后结合 Java 的一些工具来查找程序中造成资源消耗过多的代码，下面就以 Linux 和 Sun JDK 为例来介绍如何查找 Java 应用的性能瓶颈。

### 5.1.1 CPU 消耗分析

在 Linux 中，CPU 主要用于中断、内核以及用户进程的任务处理，优先级为中断>内核>用户进程，在学习如何分析 CPU 消耗状况前，还有三个重要的概念要阐述。

- 上下文切换

每个 CPU（或多核 CPU 中的每核 CPU）在同一时间只能执行一个线程<sup>2</sup>，Linux 采用的是抢占式调度。即为每个线程分配一定的执行时间，当到达执行时间、线程中有 IO 阻塞或高优先级线程要执行时，Linux 将切换执行的线程，在切换时要存储目前线程的执行状态，并恢复要执行的线程的状态，这个过程就称为上下文切换。对于 Java 应用，典型的是在进行文件 IO 操作、网络 IO 操作、锁等待或线程 Sleep 时，当前线程会进入阻塞或休眠状态，从而触发上下文切换，上下文切换过多会造成内核占据较多的 CPU 使用，使得应用的响应速度下降。

- 运行队列

每个 CPU 核都维护了一个可运行的线程队列，例如一个 4 核的 CPU，Java 应用中启动了 8 个线程，且这 8 个线程都处于可运行状态，那么在分配平均的情况下每个 CPU 中的运行队列里就会有两个线程。通常而言，系统的 load 主要由 CPU 的运行队列来决定，假设以上状况维持了 1 分钟，那么这 1 分钟内系统的 load 就会是 2，但由于 load 是个复杂的值<sup>3</sup>，因此也不是绝对的，运行队列值越大，就意味着线

<sup>2</sup> 有些 CPU 可通过超线程（Hyper Threading）方式支持同时执行多个线程

<sup>3</sup> <http://www.linuxjournal.com/article/9001>

程会要消耗越长的时间才能执行完。Linux System and Network Performance Monitoring<sup>4</sup>中建议控制在每个CPU核上的运行队列为1~3个。

- 利用率

CPU利用率为CPU在用户进程、内核、中断处理、IO等待以及空闲五个部分使用百分比，这五个值是用来分析CPU消耗情况的关键指标。Linux System and Network Performance Monitoring中建议用户进程的CPU消耗/内核的CPU消耗的比率在65%~70%/30%~35%左右。

在Linux中，可通过top或pidstat方式来查看进程中线程的CPU的消耗状况。

### 1. top

输入top命令后即可查看CPU的消耗情况，CPU的信息在TOP视图的上面几行中（如图5.2所示）。

```
top - 14:54:42 up 39 days, 21:45, 4 users, load average: 0.88, 1.22, 1.23
Tasks: 68 total, 2 running, 66 sleeping, 0 stopped, 0 zombie
Cpu(s): 4.0% us, 8.9% sy, 0.0% ni, 87.0% id, 0.0% wa, 0.2% hi, 0.0% si
Mem: 4194436k total, 3397528k used, 796908k free, 126972k buffers
Swap: 2096472k total, 5216k used, 2091256k free, 1759876k cached
```

图5.2 TOP查看CPU消耗

在此需要关注的是第三行的信息，其中4.0% us表示为用户进程处理所占的百分比；8.9% sy表示为内核线程处理所占的百分比；0.0% ni表示被nice命令改变优先级的任务所占的百分比；87.0% id表示CPU的空闲时间所占的百分比；0.0% wa表示为在执行的过程中等待IO所占的百分比；0.2% hi表示为硬件中断所占的百分比；0.0% si表示为软件中断所占的百分比。

对于多个或多核的CPU，上面的显示则会是多个CPU所占用的百分比的总和，因此会出现160% us这样的现象。如须查看每个核的消耗情况，可在进入top视图后按1，就会按核来显示消耗情况，如图5.3所示：

```
top - 15:53:29 up 39 days, 22:44, 4 users, load average: 0.90, 1.28, 1.26
Tasks: 68 total, 2 running, 66 sleeping, 0 stopped, 0 zombie
Cpu0 : 10.3% us, 20.0% sy, 0.0% ni, 69.7% id, 0.0% wa, 0.0% hi, 0.0% si
Cpu1 : 2.7% us, 7.3% sy, 0.0% ni, 90.0% id, 0.0% wa, 0.0% hi, 0.0% si
Cpu2 : 1.7% us, 5.0% sy, 0.0% ni, 93.3% id, 0.0% wa, 0.0% hi, 0.0% si
Cpu3 : 2.0% us, 7.3% sy, 0.0% ni, 90.7% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 4194436k total, 3407136k used, 787300k free, 127012k buffers
Swap: 2096472k total, 5216k used, 2091256k free, 1762696k cached
```

图5.3 TOP查看每CPU的消耗

默认情况下，TOP视图中显示的为进程的CPU消耗状况，在TOP视图中按shift+h后，可按线程查看CPU的消耗状况，如图5.4所示。

此时的PID即为线程ID，其后的%CPU表示该线程所消耗的CPU百分比。

<sup>4</sup> <http://www.ufsdump.org/papers/oscon2009-linux-monitoring.pdf>

| PID   | USER  | PR | NI | VIRT  | RES  | SHR | S | %CPU | %MEM | TIME+     | COMMAND |
|-------|-------|----|----|-------|------|-----|---|------|------|-----------|---------|
| 6129  | admin | 16 | 0  | 2170m | 1.4g | 17m | S | 6    | 34.5 | 227:41.23 | java    |
| 19444 | admin | 16 | 0  | 2170m | 1.4g | 17m | S | 5    | 34.5 | 0:21.58   | java    |
| 18156 | admin | 15 | 0  | 2170m | 1.4g | 17m | S | 4    | 34.5 | 1:18.17   | java    |
| 19449 | admin | 16 | 0  | 2170m | 1.4g | 17m | S | 4    | 34.5 | 0:20.60   | java    |
| 19455 | admin | 16 | 0  | 2170m | 1.4g | 17m | S | 4    | 34.5 | 0:15.87   | java    |
| 19722 | admin | 16 | 0  | 2170m | 1.4g | 17m | S | 4    | 34.5 | 0:06.59   | java    |
| 6133  | admin | 16 | 0  | 2170m | 1.4g | 17m | S | 3    | 34.5 | 115:15.68 | java    |
| 6163  | admin | 16 | 0  | 2170m | 1.4g | 17m | S | 3    | 34.5 | 114:55.27 | java    |
| 6380  | admin | 16 | 0  | 2170m | 1.4g | 17m | S | 3    | 34.5 | 115:05.34 | java    |

图 5.4 TOP 查看线程 CPU 消耗

## 2. pidstat

pidstat 是 SYSSTAT 中的工具，如须使用 pidstat，请先安装 SYSSTAT<sup>5</sup>。

输入 pidstat 1 2，在 console 上将会每隔 1 秒输出目前活动进程的 CPU 消耗状况，共输出 2 次，结果如图 5.5 所示：

| 10:25:28 | PID   | %usr  | %system | %guest | %CPU  | CPU | Command |
|----------|-------|-------|---------|--------|-------|-----|---------|
| 10:25:29 | 29500 | 14.85 | 0.99    | 0.00   | 15.84 | 2   | java    |
| 10:25:29 | PID   | %usr  | %system | %guest | %CPU  | CPU | Command |
| 10:25:30 | 29500 | 2.00  | 0.00    | 0.00   | 2.00  | 2   | java    |
| Average: | PID   | %usr  | %system | %guest | %CPU  | CPU | Command |
| Average: | 29500 | 8.46  | 0.50    | 0.00   | 8.96  | -   | java    |

图 5.5 pidstat 查看进程 CPU 消耗

其中 CPU 表示的为当前进程所使用到的 CPU 个数，如须查看某进程中线程的 CPU 消耗状况，可输入 pidstat -p [PID] -t 1 5 这样的方式来查看，执行后的输出如图 5.6 所示：

| 10:28:49 | TGID  | TID   | %usr  | %system | %guest | %CPU  | CPU | Command |
|----------|-------|-------|-------|---------|--------|-------|-----|---------|
| 10:28:50 | 29500 | -     | 62.00 | 0.00    | 0.00   | 62.00 | 2   | java    |
| 10:28:50 | -     | 29500 | 0.00  | 0.00    | 0.00   | 0.00  | 2   | java    |
| 10:28:50 | -     | 29501 | 0.00  | 0.00    | 0.00   | 0.00  | 2   | java    |
| 10:28:50 | -     | 29502 | 14.00 | 0.00    | 0.00   | 14.00 | 3   | java    |
| 10:28:50 | -     | 29503 | 14.00 | 0.00    | 0.00   | 14.00 | 3   | java    |
| 10:28:50 | -     | 29504 | 14.00 | 0.00    | 0.00   | 14.00 | 2   | java    |
| 10:28:50 | -     | 29505 | 14.00 | 1.00    | 0.00   | 15.00 | 1   | java    |
| 10:28:50 | -     | 29506 | 0.00  | 0.00    | 0.00   | 0.00  | 3   | java    |
| 10:28:50 | -     | 29507 | 0.00  | 0.00    | 0.00   | 0.00  | 2   | java    |
| 10:28:50 | -     | 29508 | 0.00  | 0.00    | 0.00   | 0.00  | 2   | java    |
| 10:28:50 | -     | 29509 | 0.00  | 0.00    | 0.00   | 0.00  | 2   | java    |
| 10:28:50 | -     | 29510 | 0.00  | 0.00    | 0.00   | 0.00  | 1   | java    |
| 10:28:50 | -     | 29511 | 0.00  | 0.00    | 0.00   | 0.00  | 2   | java    |
| 10:28:50 | -     | 29512 | 0.00  | 0.00    | 0.00   | 0.00  | 2   | java    |
| 10:28:50 | -     | 29513 | 0.00  | 0.00    | 0.00   | 0.00  | 2   | java    |
| 10:28:50 | -     | 29514 | 0.00  | 0.00    | 0.00   | 0.00  | 0   | java    |
| 10:28:50 | -     | 29515 | 0.00  | 0.00    | 0.00   | 0.00  | 3   | java    |

图 5.6 pidstat 查看线程 CPU 消耗

图中的 TID 即为线程 ID，较之 top 命令方式而言，pidstat 的好处为可查看每个线程的具体 CPU 利用率的状况（例如%system）。

5 <http://www.icewalkers.com/Linux/Software/59040/sysstat.html>

除 top、pidstat 外，linux 中还可使用 vmstat 来采样（例如每秒 vmstat 1）查看 CPU 的上下文切换、运行队列、利用率的具体信息。ps Hh -eo tid,pcpu 方式也可用来查看具体线程的 CPU 消耗状况；sar 来查看一定时间范围内以及历史的 cpu 消耗状况信息。

当 CPU 消耗严重时，主要体现在 us、sy、wa 或 hi 的值变高，wa 的值是 IO 等待造成的，这个在之后的章节中阐述；hi 的值变高主要为硬件中断造成的，例如网卡接收数据频繁的状况。

对于 Java 应用而言，CPU 消耗严重主要体现在 us、sy 两个值上，来分别看看 Java 应用在这两个值高的情况下应如何寻找对应造成瓶颈的代码。

### 1. us

当 us 值过高时，表示运行的应用消耗了大部分的 CPU。在这种情况下，对于 Java 应用而言，最重要的为找到具体消耗 CPU 的线程所执行的代码，可采用如下方法做到。

首先通过 linux 提供的命令找到消耗 CPU 严重的线程及其 ID，将此线程 ID 转化为十六进制的值。之后通过 kill -3 [javapid] 或 jstack 的方式 dump 出应用的 java 线程信息，通过之前转化出的十六进制的值找到对应的 nid 值的线程。该线程即为消耗 CPU 的线程，在采样时须多执行几次上述的过程，以确保找到真实的消耗 CPU 的线程。

Java 应用造成 us 高的原因主要是线程一直处于可运行（Runnable）状态，通常是这些线程在执行无阻塞、循环、正则或纯粹的计算等动作造成；另外一个可能也会造成 us 高的原因是频繁的 GC。如每次请求都需要分配较多内存，当访问量高的时候就将导致不断地进行 GC，系统响应速度下降，进而造成堆积的请求更多，消耗的内存更严重，最严重的时候有可能会导致系统不断进行 Full GC，对于频繁 GC 的状况要通过分析 JVM 内存的消耗来查找原因。

以下为一个模拟这种状况的代码。

```
public static void main(String[] args) throws Exception{
    Demo demo=new Demo();
    demo.runTest();
}

private void runTest() throws Exception{
    int count=Runtime.getRuntime().availableProcessors();
    for (int i = 0; i < count; i++) {
        new Thread(new ConsumeCPUTask()).start();
    }
    for (int i = 0; i < 200; i++) {
        new Thread(new NotConsumeCPUTask()).start();
    }
}

class ConsumeCPUTask implements Runnable{
```

```

public void run() {
    String
str="fwljfdsklvnxcewewrewrew12wre5rewflew2few4few2few2few3few3few5fsd1sdewu324
9gdfkvdvx" +
    "wefsdjfewvmdxlvdsfofewmvdmvfd;lvds;vds;vdsvdsxcnzgewgdfuvxmvx.;f" +
    "fsaffsdjlvcx.vcxgdfjkf;dsfdas#vdsjlfdsmv.xc.vcxjk;fewipvdmsvzlfsjlf;afdjsl;
fdsp[euiprenvs" +
        "fsdovxc.vmxceworupg;";
    float i=0.002f;
    float j=232.13243f;
    while (true) {
        j=i*j;
        str.indexOf("#");
        ArrayList<String> list=new ArrayList<String>();
        for (int k = 0; k < 10000; k++) {
            list.add(str+String.valueOf(k));
        }
        list.contains("iii");
        try {
            Thread.sleep(10);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class NotConsumeCPUTask implements Runnable{

    public void run() {
        while (true) {
            try {
                Thread.sleep(10000000);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

在linux上运行上面的程序，使用top命令并打开线程查看后看到的状况如图5.7所示：

| top - 15:37:33 up 330 days, 31 min, 3 users, load average: 1.07, 0.89, 0.39 |       |    |    |       |      |      |   |      |      |         |         |
|---|-------|----|----|-------|------|------|---|------|------|---------|---------|
| Tasks: 539 total, 3 running, 536 sleeping, 0 stopped, 0 zombie              |       |    |    |       |      |      |   |      |      |         |         |
| Cpu(s): 82.7% us, 0.7% sy, 0.0% ni, 16.5% id, 0.0% wa, 0.1% hi, 0.0% si     |       |    |    |       |      |      |   |      |      |         |         |
| Mem: 4151276k total, 3652676k used, 498600k free, 159968k buffers           |       |    |    |       |      |      |   |      |      |         |         |
| Swap: 2096472k total, 192k used, 2096280k free, 1168112k cached             |       |    |    |       |      |      |   |      |      |         |         |
| PID   | USER  | PR | NI | VIRT  | RES  | SHR  | S | %CPU | %MEM | TIME+   | COMMAND |
| 26697   | admin | 16 | 0  | 1223m | 183m | 6600 | R | 66   | 4.5  | 0:09.06 | java    |
| 26698   | admin | 16 | 0  | 1223m | 183m | 6600 | S | 66   | 4.5  | 0:09.02 | java    |
| 26699   | admin | 17 | 0  | 1223m | 183m | 6600 | S | 65   | 4.5  | 0:09.09 | java    |
| 26696   | admin | 16 | 0  | 1223m | 183m | 6600 | R | 65   | 4.5  | 0:08.99 | java    |

图5.7 top查看线程CPU消耗示例

以如上最耗CPU的线程26697为例，将26697换算成十六进制的值，结合java thread dump(jstack [pid] | grep 'nid=0x6849')找到此线程为：

```
"Thread-1" prio=10 tid=0x706cc400 nid=0x6849 runnable [0x6fd8d000]
    java.lang.Thread.State: RUNNABLE
        at chapter6.Demo$ConsumeCPUTask.run (Demo.java:36)
        at java.lang.Thread.run (Thread.java:619)
```

从上可以看到，主要是ConsumeCPUTask的执行消耗了CPU，但由于jstack需要时间，因此基于jstack并不一定能分析出真正的耗CPU的代码是哪行。例如在一个操作中循环调用了很多次其他的操作，如其他的操作每次都比较快，但由于循环太多次，造成了CPU消耗，在这种情况下jstack是无法捕捉出来的。最佳方式是通过intel vtune<sup>6</sup>来进行分析，vtune是商业软件，就不在此书中进行阐述了。在不使用vtune的情况下，则只能通过认真查看整个线程中执行的动作来分析原因，例如在上面的代码中，可看出ConsumeCPUTask一直处于运行状态，可以分析ConsumeCPUTask这个线程具体在做的动作，从其代码可看出整个线程一直处于运行过程中，中途没有IO中断、锁等待等现象，因此造成了CPU消耗严重。

## 2. sy

当sy值高时，表示Linux花费了更多的时间在进行线程切换，Java应用造成这种现象的主要原因是启动的线程比较多，且这些线程多数都处于不断的阻塞（例如锁等待、IO等待状态）和执行状态的变化过程中，这就导致了操作系统要不断地切换执行的线程，产生大量的上下文切换。在这种状况下，对Java应用而言，最重要的是找出线程要不断切换状态的原因，可采用的方法为通过kill -3 [javapid]或jstack -l [javapid]的方式dump出Java应用程序的线程信息，查看线程的状态信息以及锁信息，找出等待状态或锁竞争过多的线程。

以下为一个模拟这种状况的代码。

<sup>6</sup> <http://software.intel.com/en-us/intel-vtune/>

```

private static int threadCount=500;

public static void main(String[] args) throws Exception {
    if(args.length==1){
        threadCount=Integer.parseInt(args[0]);
    }
    SyHighDemo demo = new SyHighDemo();
    demo.runTest();
}

private Random random = new Random();

private Object[] locks;

private void runTest() throws Exception {
    locks = new Object[threadCount];
    for (int i = 0; i < threadCount; i++) {
        locks[i] = new Object();
    }
    for (int i = 0; i < threadCount; i++) {
        new Thread(new ATask(i)).start();
        new Thread(new BTask(i)).start();
    }
}

class ATask implements Runnable {

    private Object lockObject = null;

    public ATask(int i) {
        lockObject = locks[i];
    }

    public void run() {
        while (true) {
            try {
                synchronized (lockObject) {
                    lockObject.wait(random.nextInt(10));
                }
            } catch (Exception e) {
                ;
            }
        }
    }
}

class BTask implements Runnable {

```

```

private Object lockObject = null;

public BTask(int i) {
    lockObject = locks[i];
}

public void run() {
    while (true) {
        synchronized (lockObject) {
            lockObject.notifyAll();
        }
        try {
            Thread.sleep(random.nextInt(5));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}

```

在1台4核的机器执行以上代码，结合vmstat查看CPU的消耗状况，可看到类似如下的状况：

| procs |   | memory |         |        |         | swap |    | io |    | system |        |    |    | cpu |    |
|-------|---|--------|---------|--------|---------|------|----|----|----|--------|--------|----|----|-----|----|
| r     | b | swpd   | free    | buff   | cache   | si   | so | bi | bo | in     | cs     | us | sy | id  | wa |
| 9     | 0 | 144    | 2254860 | 164320 | 1564160 | 0    | 0  | 0  | 0  | 0      | 382104 | 19 | 41 | 40  | 0  |
| 61    | 0 | 144    | 2250356 | 164320 | 1564160 | 0    | 0  | 0  | 0  | 0      | 450540 | 27 | 60 | 13  | 0  |
| 175   | 0 | 144    | 2249820 | 164320 | 1564160 | 0    | 0  | 0  | 0  | 0      | 350749 | 35 | 64 | 2   | 0  |
| 149   | 0 | 144    | 2249820 | 164320 | 1564160 | 0    | 0  | 0  | 0  | 0      | 336513 | 36 | 63 | 1   | 0  |
| 31    | 0 | 144    | 2249820 | 164336 | 1564144 | 0    | 0  | 0  | 0  | 0      | 331039 | 33 | 58 | 8   | 0  |

由上可知，CPU在cs（上下文切换）以及sy上消耗很大，运行时采用jstack -l查看程序的线程状况，可看到启动了很多线程，并且很多的线程都经常处于TIMED\_WAITING(on object monitor)状态和Runnable状态的转换中。通过on object monitor对应的堆栈信息，可查找到系统中锁竞争激烈的代码，这是造成系统更多时间耗费在线程上下文切换的原因。

## 5.1.2 文件IO消耗分析

Linux在操作文件时，将数据放入文件缓存区，直到内存不够或系统要释放内存给用户进程使用，因此在查看Linux内存状况时经常会发现可用(free)的物理内存不多，但cached用了很多，这是Linux提升文件IO速度的一种做法。在这样的做法下，如物理空闲内存够用，通常在Linux上只有写文件和

第一次读取文件时会产生真正的文件 IO。

在 Linux 中，要跟踪线程的文件 IO 的消耗，主要方法是通过 pidstat 来查找。

- pidstat

输入如 pidstat -d -t -p [pid] 1 100 类似的命令即可查看线程的 IO 消耗状况，必须在 2.6.20 以上版本的内核中执行才有效，执行后的效果如图 5.8 所示：

| 11:44:41 | TGID | TID  | KB_rd/s | KB_wr/s | KB_ccwr/s | Command |
|----------|------|------|---------|---------|-----------|---------|
| 11:44:42 | 2013 | -    | 0.00    | 0.00    | 0.00      | java    |
| 11:44:42 | -    | 2013 | 0.00    | 0.00    | 0.00      | __java  |
| 11:44:42 | -    | 2014 | 0.00    | 0.00    | 0.00      | -__java |
| 11:44:42 | -    | 2015 | 0.00    | 0.00    | 0.00      | -__java |
| 11:44:42 | -    | 2016 | 0.00    | 0.00    | 0.00      | -__java |
| 11:44:42 | -    | 2017 | 0.00    | 0.00    | 0.00      | -__java |

图 5.8 pidstat 查看线程 IO 消耗

其中 KB\_rd/s 表示每秒读取的 KB 数，KB\_wr/s 表示每秒写入的 KB 数。

在没有安装 pidstat 或内核版本为 2.6.20 以后的版本的情况下，可通过 iostat 来查看，但 iostat 只能查看整个系统的文件 IO 消耗情况，无法跟踪到进程的文件 IO 消耗状况。

- iostat

直接输入 iostat 命令，可查看各个设备的 IO 历史状况，如图 5.9 所示：

| avg-cpu: | %user | %nice      | %sys       | %iowait  | %idle    |
|----------|-------|------------|------------|----------|----------|
|          | 0.04  | 0.00       | 0.02       | 0.01     | 99.93    |
| <hr/>    |       |            |            |          |          |
| Device:  | tps   | Blk_read/s | Blk_wrtn/s | Blk_read | Blk_wrtn |
| xvda     | 0.51  | 0.21       | 5.91       | 1044056  | 29019296 |
| xvda1    | 0.00  | 0.00       | 0.00       | 1568     | 30       |
| xvda2    | 0.13  | 0.15       | 0.99       | 759506   | 4862144  |
| xvda3    | 0.00  | 0.00       | 0.00       | 972      | 0        |
| xvda5    | 0.04  | 0.02       | 0.32       | 103354   | 1558984  |
| xvda6    | 0.00  | 0.01       | 0.01       | 28546    | 63096    |
| xvda7    | 0.58  | 0.03       | 4.59       | 150070   | 22534970 |

图 5.9 iostat 查看 IO 消耗

在上面的几项指标中，其中 Device 表示设备卷标名或分区名；tps 是每秒的 IO 请求数，这也是 IO 消耗情况中值得关注的数字；Blk\_read/s 是指每秒读的块数量，通常块的大小为 512 字节；Blk\_wrtn/s 是指每秒写的块数量；Blk\_read 是指总共读取的块数量；Blk\_wrtn 是指总共写入的块数量。

除了上面的方式外，还可通过输入 iostat -x xvda 3 5 这样的方式来定时采样查看 IO 的消耗状况，当使用上面的命令方式时，其输出信息会比直接输入 iostat 多一些：

```
avg-cpu: %user %nice %sys %iowait %idle
          3.30    0.00   0.24   0.02  96.45

Device: rrqm/s wrqm/s   r/s    w/s   rsec/s  wsec/s   rkB/s   wkB/s  avgrq-sz
avgqu-sz  await  svctm %util
```

其中值得关注的主要有：r/s 表示每秒读的请求数；w/s 表示每秒写的请求数；await 表示平均每次 IO 操作的等待时间，单位为毫秒；avgqu-sz 表示等待请求的队列的平均长度；svctm 表示平均每次设备执行 IO 操作的时间；util 表示一秒之中有百分之几用于 IO 操作。

在使用 iostat 查看 IO 的消耗情况时，首先要关注的是 CPU 中的 iowait% 所占的百分比，当 iowait 占据了主要的百分比时，就表示要关注 IO 方面的消耗状况了，这时可以再通过 iostat -x 这样的方式来详细地查看具体状况。

当文件 IO 消耗过高时，对于 Java 应用最重要的是找到造成文件 IO 消耗高的代码，寻找的最佳方法为通过 pidstat 直接找到文件 IO 操作多的线程。之后结合 jstack 找到对应的 Java 代码，如没有 pidstat，也可直接根据 jstack 得到的线程信息来分析其中文件 IO 操作较多的线程。

Java 应用造成文件 IO 消耗严重主要是多个线程需要进行大量内容写入（例如频繁的日志写入）的动作；或磁盘设备本身的处理速度慢；或文件系统慢；或操作的文件本身已经很大造成的。

在下面的例子中，通过往一个文件中不断地增加内容，文件越来越大，造成写速度慢，最终 IOWait 值高，代码如下：

```

private String fileName="/tmp/iowait.log";

private static int threadCount=Runtime.getRuntime().availableProcessors();

private Random random=new Random();

public static void main(String[] args) throws Exception{
    if(args.length==1){
        threadCount=Integer.parseInt(args[1]);
    }
    IOWaitHighDemo demo=new IOWaitHighDemo();
    demo.runTest();
}

private void runTest() throws Exception{
    File file=new File(fileName);
    file.createNewFile();
    for (int i = 0; i < threadCount; i++) {
        new Thread(new Task()).start();
    }
}

class Task implements Runnable{

    public void run() {
}

```

```

        while(true) {
            try{
                BufferedWriter writer=new BufferedWriter(new
FileWriter(fileName,true));
                StringBuilder strBuilder=new
StringBuilder("====begin====\n");
                String threadName=Thread.currentThread().getName();
                for (int i = 0; i < 100000; i++) {
                    strBuilder.append(threadName);
                    strBuilder.append("\n");
                }
                strBuilder.append("====end====\n");
                writer.write(strBuilder.toString());
                writer.close();
                Thread.sleep(random.nextInt(10));
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}

```

执行以上代码，通过 iostat 可看到类似图 5.10 的信息：

```

avg-cpu: %user %nice %sys %iowait %idle
      9.50    0.00   6.50   83.50    0.50

Device: rrqm/s wrqm/s   r/s   w/s rsec/s  usec/s   rkB/s   wkB/s  avgrrq-sz avgqu-sz  await svctm  %util
xvda     0.00  27.50  0.50 9884.00    4.00 79276.00    2.00 39638.00    8.02 8206.79  805.18   0.10 100.00

avg-cpu: %user %nice %sys %iowait %idle
     24.75    0.00  14.25   52.75    8.25

Device: rrqm/s wrqm/s   r/s   w/s rsec/s  usec/s   rkB/s   wkB/s  avgrrq-sz avgqu-sz  await svctm  %util
xvda     0.00  31.50  0.00 5496.00    0.00 11444.00    0.00 5722.00    2.08 3803.63  899.55   0.13  73.50

avg-cpu: %user %nice %sys %iowait %idle
     13.75    0.00  12.25   54.00   20.00

Device: rrqm/s wrqm/s   r/s   w/s rsec/s  usec/s   rkB/s   wkB/s  avgrrq-sz avgqu-sz  await svctm  %util
xvda     0.00 253.50  0.50 6537.50    4.00 87164.00    2.00 43582.00   13.33 5329.84  467.50   0.10   65.00

```

图 5.10 iostat 查看示例代码的 IO 消耗

从上面可看出，iowait 占据了很多的 CPU，结合 iostat 的信息来看，主要是写的消耗，并且花在 await 的时间上要远大于 svctm 的时间。至于具体是什么动作导致了 iowait，仍然要通过对应用的线程 dump 来分析，找出其中 IO 操作相关的动作。对上面的操作进行线程的 dump，可看到如下信息：

```

"Thread-1" prio=10 tid=0x08111800 nid=0x5c97 runnable [0x9157b000..0x9157bea0]
java.lang.Thread.State: RUNNABLE
    at java.io.FileOutputStream.writeBytes(Native Method)
    at java.io.FileOutputStream.write(FileOutputStream.java:260)
    at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:202)

```

```

at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:263)
at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:106)
- locked <0x9338da60> (a java.io.FileWriter)
at java.io.OutputStreamWriter.write(OutputStreamWriter.java:190)
at java.io.BufferedWriter.flushBuffer(BufferedWriter.java:111)
- locked <0x9338da60> (a java.io.FileWriter)
at java.io.BufferedWriter.write(BufferedWriter.java:212)
- locked <0x9338da60> (a java.io.FileWriter)
at java.io.Writer.write(Writer.java:140)
at chapter6.fileio.demo.IOWaitHighDemo$Task.run(IOWaitHighDemo.java:59)
at java.lang.Thread.run(Thread.java:619)

```

从上面的线程堆栈中，可看到线程停留在了 `FileOutputStream.writeBytes` 这个 Native 方法上，这方法所做的动作是将数据写入文件中，也就是所要寻找的 IO 操作相关的动作。继续跟踪堆栈往上查找，直到查找到系统中的代码，例如上面的例子中则为 `IOWaitHighDemo.java:59`。

使用 `pidstat` 则简单很多，直接通过 `pidstat` 找到 IO 读写量大的线程 ID，然后结合上面的线程 dump 即可找到相应的耗文件 IO 多的动作。

### 5.1.3 网络 IO 消耗分析

对于分布式 Java 应用而言，网络 IO 的消耗非常值得关注，尤其要注意网卡中断是不是均衡地分配到各 CPU 的（可通过 `cat /proc/interrupts` 查看）。对于网卡中断只分配到一个 CPU 的现象，google 采用了修改 `kernel` 的方法对网卡中断分配不均的问题进行修复，据其测试性能大概能提升 3x 左右<sup>7</sup>，或是采用支持 MSI-X<sup>8</sup>的网卡来修复。

在 Linux 中可采用 `sar` 来分析网络 IO 的消耗状况，具体结果如图 5.11 所示。

#### `sar`

输入 `sar -n FULL 1 2`，执行后以 1 秒为频率，总共输出两次网络 IO 的消耗情况，示例如下。

上面的信息中输出的主要有三部分，第一部分为网卡上成功接包和发包的信息，其报告的信息中主要有 `rxpck/s`、`txpck/s`、`rxbyt/s`、`txbyt/s`、`rxmcst/s`；第二部分为网卡上失败的接包和发包的信息，其报告的信息中主要有 `rxerr/s`、`txerr/s`、`rxdrop/s`、`txdrop/s`；第三部分为 `sockets` 上的统计信息，其报告的信息中主要有 `tolcck`、`tcpsck`、`udpsck`、`rawsck`。关于这些数值的具体含义可通过 `man sar` 来进行了解，对于 Java 应用而言，使用的主要为 `tcpsck` 和 `udpsck`。

如须详细跟踪 `tcp/ip` 通信过程的信息，则可通过 `tcpdump`<sup>9</sup>来进行。

7 <http://lwn.net/Articles/362339/>

8 [http://en.wikipedia.org/wiki/Message\\_Signaled\\_Interrupts](http://en.wikipedia.org/wiki/Message_Signaled_Interrupts)

9 <http://www.computerhope.com/unix/tcpdump.htm>

由于没办法分析具体每个线程所消耗的网络 IO，因此当网络 IO 消耗高时，对于 Java 应用而言只能对线程进行 dump，查找产生了大量网络 IO 操作的线程。这些线程的特征是读取或写入网络流，在用 Java 实现网络通信时，通常要将对象序列化为字节流，进行发送，或读取字节流，并反序列化为对象。这个过程要消耗 JVM 堆内存，JVM 堆的内存大小通常是有限的，因此 Java 应用一般不会造成网络 IO 消耗严重。

```

09:47:07 AM    IFACE  rxpck/s  txpck/s  rxbyt/s  txbyt/s  rxcmp/s  txcmp/s  rxmcst/s
09:47:08 AM      lo    0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:08 AM     eth0  5339.22  4650.00  3592126.47  3131720.59     0.00     0.00     0.00
09:47:08 AM     eth1    0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:08 AM     sit0    0.00     0.00     0.00     0.00     0.00     0.00     0.00

09:47:07 AM    IFACE  rxerr/s  txerr/s  coll/s  rxdrop/s  txdrop/s  txcarr/s  rxfram/s  rxfifo/s  txfifo/s
09:47:08 AM      lo    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:08 AM     eth0    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:08 AM     eth1    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:08 AM     sit0    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00

09:47:07 AM    totsck  tcpsck   udpsck   rawsck   ip-frag
09:47:08 AM      1642     1577      10        0         0

09:47:08 AM    IFACE  rxpck/s  txpck/s  rxbyt/s  txbyt/s  rxcmp/s  txcmp/s  rxmcst/s
09:47:09 AM      lo    0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:09 AM     eth0  3942.86  3325.51  2451343.88  2115575.51     0.00     0.00     0.00
09:47:09 AM     eth1    0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:09 AM     sit0    0.00     0.00     0.00     0.00     0.00     0.00     0.00

09:47:08 AM    IFACE  rxerr/s  txerr/s  coll/s  rxdrop/s  txdrop/s  txcarr/s  rxfram/s  rxfifo/s  txfifo/s
09:47:09 AM      lo    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:09 AM     eth0    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:09 AM     eth1    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
09:47:09 AM     sit0    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00

09:47:08 AM    totsck  tcpsck   udpsck   rawsck   ip-frag
09:47:09 AM      1642     1577      10        0         0

Average:    IFACE  rxpck/s  txpck/s  rxbyt/s  txbyt/s  rxcmp/s  txcmp/s  rxmcst/s
Average:      lo    0.00     0.00     0.00     0.00     0.00     0.00     0.00
Average:     eth0  4655.00  4001.00  3033143.00  2633809.50     0.00     0.00     0.00
Average:     eth1    0.00     0.00     0.00     0.00     0.00     0.00     0.00
Average:     sit0    0.00     0.00     0.00     0.00     0.00     0.00     0.00

Average:    IFACE  rxerr/s  txerr/s  coll/s  rxdrop/s  txdrop/s  txcarr/s  rxfram/s  rxfifo/s  txfifo/s
Average:      lo    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
Average:     eth0    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
Average:     eth1    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
Average:     sit0    0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00

Average:    totsck  tcpsck   udpsck   rawsck   ip-frag
Average:      1642     1577      10        0         0

```

图 5.11 sar 查看网络 IO 消耗

## 5.1.4 内存消耗分析

根据之前 JVM 第 4 章中对于 Java 对象内存分配以及回收的介绍，可以看出 Java 应用对于内存的消耗主要是在 JVM 堆内存上，在正式环境中，多数 Java 应用都会将-Xms 和-Xmx 设为相同的值，避免运行期要不断申请内存。

目前的 Java 应用只有在创建线程和使用 Direct ByteBuffer 时才会操作 JVM 堆外的内存 JVM，因此在内存消耗方面最为值得关注的是 JVM 内存的消耗状况。对于 JVM 内存消耗状况分析的方法在深入 JVM 中已介绍 (jmap、jstat、mat、visualvm 等方法)，在此就不再进行阐述，JVM 内存消耗过多会导致 GC 执行频繁，CPU 消耗增加，应用线程的执行速度严重下降，甚至造成 OutOfMemoryError，最终导致 Java 进程退出。

对于 JVM 堆以外的内存方面的消耗，最为值得关注的是 swap 的消耗以及物理内存的消耗，这两方面的消耗都可基于 os 提供的命令来查看。

在 Linux 中可通过 vmstat、sar、top、pidstat 等方式来查看 swap 和物理内存的消耗状况。

### vmstat

在命令行中输入 vmstat，其中的信息和内存相关的主要是在 memory 下的 swpd、free、buff、cache 以及 swap 下的 si 和 so。

其中 swpd 是指虚拟内存已使用的部分，单位为 kb；free 表示空闲的物理内存，buff 表示用于缓冲的内存，cache 表示用于作为缓存的内存，swap 下的 si 是指每秒从 disk 读至内存的数据量，so 是指每秒从内存中写入 disk 的数据量。

swpd 值过高通常是由于物理内存不够用了，os 将物理内存中的一部分数据转为放入硬盘上进行存储，以腾出足够的空间给当前运行的程序使用。在目前运行的程序变化后，即从硬盘上重新读取数据到内存中，以便恢复程序的运行，这个过程会产生 swap IO，因此看 swap 的消耗情况主要关注的是 swap IO 的状况，如 swap IO 发生得较频繁，那么会严重影响系统的性能。

由于 Java 应用是单进程应用，因此只要 JVM 的内存设置不是过大，是不会操作到 swap 区域的。物理内存消耗过高可能是由于 JVM 内存设置过大、创建的 Java 线程过多或通过 Direct ByteBuffer 往物理内存中放置了过多的对象造成的。

### sar

通过 sar 的 -r 参数可查看内存的消耗状况如图 5.12 所示，例如 sar -r 2 5：

|             | kmemfree | kmemused | %memused | kbuffers | kbcached | kbswpfree | kbswpused | %swpused | kbswpcad |
|-------------|----------|----------|----------|----------|----------|-----------|-----------|----------|----------|
| 04:05:39 PM |          |          |          |          |          |           |           |          |          |
| 04:05:41 PM | 465728   | 3682048  | 88.77    | 343596   | 1674524  | 2096264   | 208       | 0.01     | 0        |
| 04:05:43 PM | 465728   | 3682048  | 88.77    | 343596   | 1674524  | 2096264   | 208       | 0.01     | 0        |
| 04:05:45 PM | 465864   | 3681912  | 88.77    | 343596   | 1674524  | 2096264   | 208       | 0.01     | 0        |
| 04:05:47 PM | 465800   | 3681976  | 88.77    | 343596   | 1674524  | 2096264   | 208       | 0.01     | 0        |
| 04:05:49 PM | 465992   | 3681784  | 88.77    | 343596   | 1674524  | 2096264   | 208       | 0.01     | 0        |
| Average:    | 465822   | 3681954  | 88.77    | 343596   | 1674524  | 2096264   | 208       | 0.01     | 0        |

图 5.12 sar 查看内存消耗

其中和 swap 相关的信息主要是 kbswpfree、kbswpused、%swpused，kbswpfree 表示 swap 空闲的大小，kbswpused 表示已使用的 swap 大小，%swpused 表示使用的 swap 空间比率。

其中和物理内存相关的信息主要是 kmemfree、kmemused、%memused、kbuffers、kbcached，当物理内存有空闲时，linux 会使用一些物理内存用于 buffer 以及 cache，以提升系统的运行效率，因此可以认为系统中可用的物理内存为：kmemfree+kbuffers+kbcached。

sar 相比 vmstat 的好处是可以查询历史状况，以更加准确地分析趋势状况，例如 sar -r -f /tmp/log/sa/sa12。

vmstat 和 sar 的共同弱点是不能分析进程所占用的内存量。

### top

通过 top 可查看进程所消耗的内存量，不过 top 中看到的 Java 进程的消耗内存是包括了 JVM 已分配的内存加上 Java 应用所耗费的 JVM 以外的物理内存，这会导致 top 中看到 Java 进程所消耗的内存大

小有可能超过-Xmx 加上-XX:MaxPermSize 设置的内存大小，并且 java 程序在启动后也只是占据了-Xms 的地址空间，但并没有占据实际的内存，只有在相应的地址空间被使用过后才会被计入消耗的内存中。因此纯粹根据 top 很难判断出 Java 进程消耗的内存中有多少是属于 JVM 的，有多少是属于消耗 JVM 外的内存。一个小技巧是，对由于内存满而发生过 Full GC 的应用而言（不是主动调用 System.gc 的应用），多数情况下（例如由于产生的对象过大导致执行 Full GC 并抛出 OutOfMemoryError 的现象就要除外）可以认为其 Java 进程中显示出来的内存消耗值即为 JVM -Xmx 的值+消耗的 JVM 外的内存值。

## pidstat

通过 pidstat 也可查看进程所消耗的内存量，命令格式为：pidstat -r -p [pid] [interval] [times]，例如 pidstat -r -p 2013 1 100，执行此命令后可查看该进程所占用的物理内存和虚拟内存的大小，示例如图 5.13 所示：

|          | PID  | minflt/s | majflt/s | Vsz     | RSS     | %MEM  | Command |
|----------|------|----------|----------|---------|---------|-------|---------|
| 12:02:48 | 2013 | 1.00     | 0.00     | 1822224 | 1615900 | 38.96 | java    |
| 12:02:49 | 2013 | 0.00     | 0.00     | 1822224 | 1615900 | 38.96 | java    |
| 12:02:50 | 2013 | 0.00     | 0.00     | 1822224 | 1615900 | 38.96 | java    |
| 12:02:51 | 2013 | 0.00     | 0.00     | 1822224 | 1615900 | 38.96 | java    |
| Average: | 2013 | 0.33     | 0.00     | 1822224 | 1615900 | 38.96 | java    |

图 5.13 pidstat 查看进程内存消耗

从以上的几个工具来看，最佳的内存消耗分析方法是结合 top 或 pidstat，以及 JVM 的内存分析工具来共同分析内存的消耗状况。

下面的两个例子分别展示了 Java 应用对物理内存的消耗和对 JVM 堆内存的消耗。

## 对物理内存的消耗

基于 Direct ByteBuffer 可以很容易地实现对物理内存的直接操作，而无须耗费 JVM heap 区，以下是一个简单的例子：

```
public static void main(String[] args) throws Exception{
    Thread.sleep(20000);
    System.out.println("read to create bytes, so JVM heap will be used");
    byte[] bytes=new byte[128*1000*1000];
    bytes[0]=1;
    bytes[1]=2;
    Thread.sleep(10000);
    System.out.println("read to allocate & put direct bytebuffer, no JVM heap
should be used");
    ByteBuffer buffer=ByteBuffer.allocateDirect(128*1024*1024);
    buffer.put(bytes);
    buffer.flip();
    Thread.sleep(10000);
    System.out.println("ready to gc, JVM heap will be freed");
    bytes=null;
    System.gc();
    Thread.sleep(10000);
```

```

System.out.println("read to get bytes,then JVM heap will be used");
byte[] resultbytes=new byte[128*1000*1000];
buffer.get(resultbytes);
System.out.println("resultbytes[1] is: "+resultbytes[1]);
Thread.sleep(10000);
System.out.println("read to gc all");
buffer=null;
resultbytes=null;
System.gc();
Thread.sleep(10000);
}

```

在上面的例子中，为了更清晰地观察内存的变化状况，放入了多个 Thread.sleep，加上-Xms140M -Xmx140M 参数执行上面的代码，在执行过程中结合 top 命令和 jstat 查看 java 进程占用内存的大小，以及 JVM heap 的变化情况。

通过 top 命令和 jstat 观察到的 Java 进程和 JVM 堆内存的变化状况如表 5.1 所示：

表 5.1

|  | Java 进程内存                     | JVM 堆旧生代内存 |
|--|-------------------------------|------------|
| ready to create bytes                        | 上涨                            | 98%左右      |
| ready to allocate & put direct<br>bytebuffer | 翻了一倍                          | 无变化        |
| ready to gc                                  | 无变化                           | 0.1%左右     |
| ready to get bytes                           | 无变化                           | 98%左右      |
| ready to gc all                              | 回到 ready to create bytes 时的大小 | 0.1%左右     |

## JVM

结合上面 top 和 jstat 观察到的状况，可以看出 direct bytebuffer 消耗的是 JVM heap 外的物理内存。但它同样是基于 GC 方式来释放的，同时也可以看出 JVM heap 一旦使用后，即使进行了 GC，进程中仍然会显示之前其所消耗的内存大小，因此 JVM 内存中具体的消耗状况必须通过 jdk 提供的命令才能准确分析。

除了示例中 Direct ByteBuffer 方式对 JVM 外物理内存的消耗外，创建线程也会消耗一定大小的内存。这一方面取决于-Xss 对应值的大小，另一方面也取决于线程 stack 的深度，当线程退出时，其所占用的内存将自动释放。

## 对 JVM 内存的消耗

Java 应用中除了以上少数几种操作 JVM 外物理内存的方法外，大多数都是对于 JVM heap 区的消耗，这些在第 3 章“深入理解 JVM”中已有讲解。

从上面的状况来看，在 Java 程序出现内存消耗过多、GC 频繁或 OutOfMemoryError 的情况后，要首先分析其所耗费的是 JVM 外的物理内存还是 JVM heap 区。如为 JVM 外的物理内存，则要分析程序中线程的数量以及 Direct ByteBuffer 的使用情况；如为 JVM heap 区，则要结合 JDK 提供的工具或外部的工具来分析程序中具体对象的内存占用状况。

硬件资源毕竟是有限的，当资源消耗过多时，系统性能必然会变差，以上均为资源消耗过多造成的性能差的现象，但还有一些状况为资源消耗不多，程序执行仍然很慢的现象。

## 5.1.5 程序执行慢原因分析

有些情况是资源消耗不多，但程序执行仍然慢，这种现象多出现于访问量不是非常大的情况下，造成这种现象的原因主要有以下三种：

### 1. 锁竞争激烈

锁竞争激烈直接就会造成程序执行慢，例如一个典型的例子是数据库连接池，通常数据库连接池提供的连接数都是有限的。假设提供的是 10 个，那么就意味着同时能够进行数据库操作的就只有 10 个线程，而如果此时有 50 个线程要进行数据库操作，那就会造成另外的 40 个线程处于等待状态，这种情况下对于 4 核类型的机器而言，CPU 的消耗并不会高，但程序的执行仍然会较慢。

### 2. 未充分使用硬件资源

例如机器上有双核 CPU，但程序中都是单线程串行的操作，并没有充分发挥硬件资源的作用，此时就可进行一定的优化来充分使用硬件资源，提升程序的执行速度。

### 3. 数据量增长

数据量增长通常也是造成程序执行慢的典型原因，例如当数据库中单表的数据从 100 万个上涨到 1 亿个后，数据库的读写速度将大幅度下降，相应的操作此表的程序的执行速度也就下降了。

对于以上这两种状况，要记录程序执行的整个过程的时间消耗或使用 JProfiler 等商业工具，从而找到执行耗时比率最大的代码，图 5.14 为一个基于 JProfiler 跟踪代码执行速度的截图：

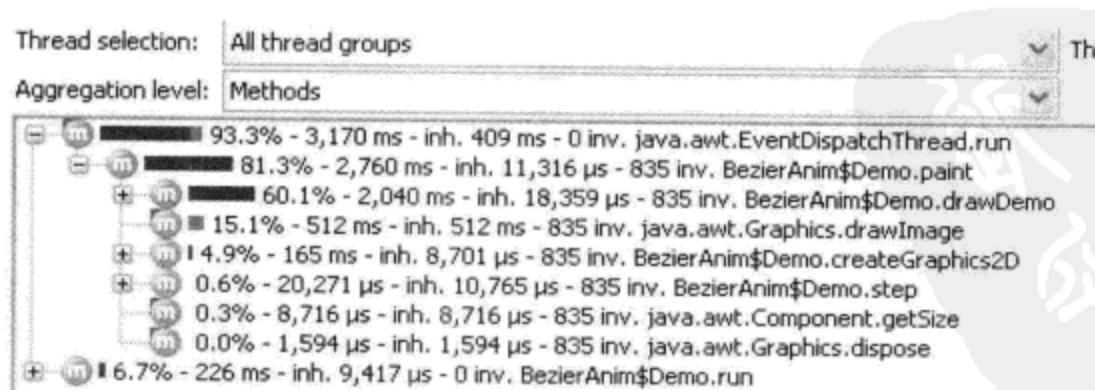


图 5.14 JProfiler 分析程序执行速度

## 5.2 调优

在寻找到系统的性能瓶颈后，接下来的步骤就是调优，以提高系统性能，调优通常可从硬件、操作系统、JVM 以及程序四个方面来着手，硬件和操作系统方面的知识不是本书的重点，请读者查阅相关的书籍<sup>10</sup>。下面就结合性能瓶颈的分析从 JVM 及程序方面来介绍一些常用的调优方法。

### 5.2.1 JVM 调优

JVM 调优主要是内存管理方面的调优，包括各个代的大小、GC 策略等。由于 GC 动作会挂起应用线程，严重影响性能，这些调优对于应用而言至关重要，根据应用的情况选择不同的内存管理策略有些时候能够大幅度地提升应用的性能，尤其是对于内存消耗较多的应用。下面就来看一些常用的内存管理调优的方法，这些方法都是为了尽量降低 GC 所导致的应用暂停时间。

#### 代大小的调优

在不采用 G1（G1 不区分 minor GC 和 Full GC）的情况下，通常 minor GC 会远快于 Full GC，各个代的大小设置直接决定了 minor GC 和 Full GC 触发的时机，在代大小的调优上，最关键的参数为：  
**-Xms -Xmx -Xmn -XX:SurvivorRatio -XX:MaxTenuringThreshold**。

**-Xms** 和 **-Xmx** 通常设置为相同的值，避免运行时要不断地扩展 JVM 内存空间，这个值决定了 JVM Heap 所能使用的最大空间。

**-Xmn** 决定了新生代（New Generation）空间的大小，新生代中 Eden、S0 和 S1 三个区域的比率可通过 **-XX:SurvivorRatio** 来控制。

**-XX:MaxTenuringThreshold** 控制对象在经历多少次 Minor GC 后才转入旧生代，通常又将此值称为新生代存活周期，此参数只有在串行 GC 时有效，其他 GC 方式时则由 Sun JDK 自行决定。

在第 3 章“深入理解 JVM”一章中已经介绍了 Minor GC 和 Full GC 触发的时机，在此处就直接举例来看看不同的代大小设置情况下，应用耗费在 GC 上的时间，从中也可看出在不同的场景下代大小调优的方法。

##### 1. 避免新生代大小设置过小

当新生代大小设置过小时，会产生两种比较明显的现象，一是 minor GC 的次数更加频繁；二是有可能导致 minor gc 对象直接进入旧生代，此时如进入旧生代的对象占据了旧生代剩余空间，则触发 Full GC。

以下的代码模拟了上面的这两种现象：

```
public class GCDemo {
    public static void main(String[] args) throws Exception{}
```

<sup>10</sup> 如《深入理解计算机系统 (Computer Systems A Programmer's Perspective)》，中国电力出版社 2004 年 10 月出版，作者：Randal E.Bryant、David O'Hallaron。

```
System.out.println("ready to start");
Thread.sleep(10000);
List<GCDataObject> oldGenObjects=new ArrayList<GCDataObject>();
for (int I = 0; I < 51200; i++) {
    oldGenObjects.add(new GCDataObject(2));
}
System.gc();
oldGenObjects.size();
oldGenObjects=null;
Thread.sleep(5000);
List<GCDataObject> tmpObjects=new ArrayList<GCDataObject>();
for (int I = 0; I < 3200; i++) {
    tmpObjects.add(new GCDataObject(5));
}
tmpObjects.size();
tmpObjects=null;
}

}

class GCDataObject{

byte[] bytes=null;

RefObject object=null;

public GCDataObject(int factor){
    // create object in kb
    bytes=new byte[factor*1024];
    object=new RefObject();
}

}

class RefObject{

RefChildObject object;

public RefObject(){
    object=new RefChildObject();
}

}

class RefChildObject{

public RefChildObject(){

}}
```

```

    ;
}

}

```

以-Xms135M -Xmx135M -Xmn20M -XX:+UseSerialGC 执行上面的代码，通过 jstat 跟踪到的 GC 状况为：

共触发 7 次 minor GC，耗时为 0.318 秒，共触发 2 次 Full GC，耗时为 0.09 秒，GC 总耗时为 0.408 秒。

其中第一次 Full GC 是代码中主动调用 System.gc 造成的，而第二次 Full GC 则是因为 minor GC 时 Survivor 区空间不足，导致了对象直接进入了旧生代，这些对象加上旧生代原有的一些对象占满了旧生代空间，于是 Full GC 被触发。而根据代码来看，其实只要新生代的空间能够支撑到 tmpObjects 中的对象填充完毕，那么下次 minor GC 就可以把 tmpObjects 所占据的空间全部回收掉，避免掉这次不必要的 Full GC。按照这样的思路，调大新生代到 30MB，重新执行，执行后通过 jstat 跟踪到的 GC 状况为：

共触发 4 次 minor GC，耗时为 0.303 秒，共触发 1 次 Full GC，耗时为 0.063 秒，GC 总耗时为 0.366 秒。

从上面的结果可以看出，经过这样的调整，有效地减少了 Full GC 的频率。

除了调大新生代大小外，如果能够调大 JVM Heap 的大小，那就更好了，但 JVM Heap 调大通常意味着单次 GC 时间的增加。

当 minor GC 过于频繁，或发现经常出现 minor GC 时，Survivor 的一个区域空间满，且 Old Gen 增长超过了 Survivor 区域大小时，就需要考虑新生代大小的调整了。调整时的原则是在不能调大 JVM Heap 的情况下，尽可能放大新生代空间，尽量让对象在 minor GC 阶段被回收，但新生代空间也不可过大；在能够调大 JVM Heap 的情况下，则可以按照增加的新生代空间大小增加 JVM Heap 大小，以保证旧生代空间够用。

## 2. 避免新生代设置过大

新生代设置过大带来的两个典型的现象，一是旧生代变小了，有可能导致 Full GC 频繁执行；二是 minor GC 的耗时大幅度增加。

仍然用上面的例子，调整为以下参数执行：-Xms135M -Xmx135M -Xmn105M，通过 jstat 观察到其 GC 状况为：

共触发 1 次 minor GC，耗时为 0.141 秒，共触发 2 次 Full GC，耗时为 0.152 秒，GC 总耗时为 0.293 秒。

从这个调整和之前把新生代调为 30MB 时对比，此时 minor GC 下降了，但 Full GC 仍然多了一次。原因在于，当第二次到达 minor GC 的触发条件时，JVM 基于悲观原则，判断目前 old 区的剩余空间小

于可能会从新生代晋升到 old 区的对象的大小，于是执行了 Full GC，而从 minor GC 消耗的时间来看，单次 minor GC 的时间也比以前慢了不少。

从上面的分析来看，可见新生代通常不能设置得过大，大多数场景下都应设置得比旧生代小，通常推荐的比例是新生代占 JVM Heap 区大小的 33%左右。

### 3. 避免 Survivor 区过小或过大

根据第 3 章“深入理解 JVM”中的讲解，在采用串行 GC 时，默认情况下 Eden、S0、S1 的大小比例为 8:1:1，调整为以下参数执行上面示例的代码：`-Xms135M -Xmx135M -Xmn20M -XX:SurvivorRatio=10 -XX:+UseSerialGC`，通过 jstat 观察到其 GC 状况为：

共触发 6 次 minor GC，耗时为 0.324 秒，共触发 1 次 Full GC，耗时为 0.055 秒，GC 总耗时为 0.379 秒。

从上面的调整结果来看，在没有调大新生代空间的情况下，同样避免了第二次 Full GC 的发生。简单分析一下原因：在上面的场景中，`tmpObjects` 在创建的过程中需要大概 16MB 的空间，新生代大小设置为 20MB，默认情况下 Eden 区的大小为 16MB，两个 Survivor 区分别为 2MB，当 `tmpObjects` 创建时，会填满 Eden space，从而触发 minor GC。而此时这 16MB 的对象都是有引用的对象，minor GC 时只能将其放入 Survivor 区，但 Survivor 区只有 2MB 的空间，因此将有 14MB 的对象转入旧生代中，而旧生代的空间大小为 115MB，之前已用了 101MB 左右的空间，当这 14MB 对象加入时，旧生代空间被占满，于是 Full GC 被触发。在加入了`-XX:SurvivorRatio=10` 参数后，Eden 区的大小调整为 16.7MB，当 `tmpObjects` 创建完毕时，还不足以触发 minor GC，自然 Full GC 也被避免了。

从上面的分析来看，在无法调整 JVM Heap 以及新生代的大小时，合理调整 Survivor 区的大小也能带来一些效果。调大 `SurvivorRatio` 值意味着 Eden 区域变大，minor GC 的触发次数会降低，但此时 Survivor 区域的空间变小了，如有超过 Survivor 空间大小的对象在 minor GC 后仍没有被回收，则会直接进入旧生代；调小 `SurvivorRatio` 则意味着 Eden 区域变小，minor GC 的触发次数会增加，Survivor 区域变大，意味着可以存储更多在 minor GC 后仍存活的对象，避免其进入旧生代。

### 4. 合理设置新生代存活周期

新生代存活周期的值决定了新生代的对象经过多少次 Minor GC 后进入旧生代，因此这个值也需要根据应用的状况来做针对性的调优，JVM 参数上这个值对应的为`-XX:MaxTenuringThreshold`，默认值为 15 次。

下面是一段示例代码，该代码的目的是先让旧生代中放置一些对象，然后让某对象在经过了 16 次 minor GC 后仍存活，超过默认的 `TenuringThreshold` 值从而进入旧生代，示例代码如下：

```
public class TenuringThresholdDemo {
    public static void main(String[] args) throws Exception{
```

```

        System.out.println("wait jstat");
        Thread.sleep(10000);
        List<DataObject> objects=new ArrayList<DataObject>();
        for (int I = 0; I < 51200; i++) {
            objects.add(new DataObject(1));
        }
        List<DataObject> tmpobjects=new ArrayList<DataObject>();
        for (int I = 0; I < 10240; i++) {
            tmpobjects.add(new DataObject(4));
        }
        System.gc();
        Thread.sleep(1000);
        tmpobjects.size();
        tmpobjects=null;
        long beginTime=System.currentTimeMillis();
        for (int I = 0; I < 30; i++) {
            DataObject toOldObject=new DataObject(1024);
            for (int j = 0; j < 16; j++) {
                for (int m = 0; m < 23; m++) {
                    new DataObject(1024);
                }
            }
            toOldObject.toString();
            toOldObject=null;
        }
        objects.size();
        long endTime=System.currentTimeMillis();
        System.out.println("Execute Summary: Execute
Time( "+(endTime-beginTime)+"ms )");
        Thread.sleep(10000);
    }

}

class DataObject{
    byte[] bytes=null;

    public DataObject(int factor){
        bytes=new byte[factor*1024];
    }

}

```

以-Xms150M -Xmx150M -Xmn30M -XX:+UseSerialGC 执行以上代码，执行结果如下：

|   |
|---|
| Execute Summary: Execute Time( 4908ms ) |
|---|

通过 jstat 观察到的 GC 结果信息为：

共执行了 484 次 minor GC，耗时为 0.528 秒，共执行了 2 次 Full GC，耗时为 0.107 秒，GC 时间总共为 0.635 秒。

调整对象在新生代中的生存周期为 16，即让示例代码中 toOldObject 在新生代中多存活一次 minor GC，从而可以让 toOldObject 对象在 minor GC 阶段被回收，增加参数：-XX:MaxTenuringThreshold=16，执行结果如下：

```
Execute Summary: Execute Time( 4893ms )
```

通过 jstat 观察到的 GC 结果信息为：

共执行了 484 次 minor GC，耗时为 0.521 秒，共执行了 1 次 Full GC，耗时为 0.073 秒，GC 时间总共为 0.593 秒。

从上面的调整结果可见，在增大了存活周期后，对象在 Minor GC 阶段被回收的机会就增加了，但同时带来的是 survivor 区被占用，但此值仅在串行 GC 和 ParNew GC 时可调整。

总结上面的几个例子来看，对于代大小的调优，主要是合理调整-Xms、-Xmx、-Xmn 以及-XX:SurvivorRatio 的值，尽可能减少 GC 所占用的时间。

-Xms、-Xmx 适用于调整整个 JVM Heap 区大小，在内存不够用的情况下可适当加大此值，这个值能调整到多大取决于操作系统位数以及 CPU 的能力。

-Xmn 适用于调整新生代的大小，新生代的大小决定了多少比例的对象有机会在 minor GC 阶段被回收，但此值相应的也决定了旧生代的大小。新生代越大，通常意味着多数对象能够在 minor GC 阶段被回收掉，但同时意味着旧生代的空间会变小，可能会造成更频繁的 Full GC，甚至是 OutOfMemoryError。

-XX:SurvivorRatio 适用于调整 Eden 区和 Survivor 区的大小，Eden 区越大通常也就意味着 minor GC 发生的频率越低。但有可能会造成 Survivor 区太小，导致对象在经过 minor GC 后直接就进入旧生代了，从而更频繁的触发 Full GC，这取决于当 Eden 区满的时候其中存活对象的比例。

在清楚掌握 minor GC、Full GC 的触发时机以及代大小的调整后，结合应用的状况（例如创建出的对象都可很快被回收掉、缓存对象多等）通常就可较好设置代的大小，减少 GC 所占用的时间。在调整后可结合 jstat、VisualVM 等查看 GC 的变化是否达到了调优的目的。

## GC 策略的调优

在第 3 章“深入理解 JVM”中，已讲解了 Sun JDK 提供的几种 GC 策略，串行 GC 性能太差，因此在实际场景中使用的主要为并行和并发 GC，在此举例来看看两种 GC 策略的具体表现。

通过以下代码来触发多次 GC，查看并行 GC 以及并发 GC 时对于应用造成不同的暂停时间。

```

public class GCPolicyDemo {
    public static void main(String[] args) throws Exception{
        System.out.println("ready to start");
        Thread.sleep(10000);
        List<GCPolicyDataObject> cacheObjects=new
ArrayList<GCPolicyDataObject>();
        for (int i = 0; i < 2048; i++) {
            cacheObjects.add(new GCPolicyDataObject(100));
        }
        System.gc();
        Thread.sleep(1000);
        for (int i = 0; i < 10; i++) {
            System.out.println("Round: "+(i+1));
            for (int j = 0; j < 5; j++) {
                System.out.println("put 64M objects");
                List<GCPolicyDataObject> tmpObjects=new
ArrayList<GCPolicyDataObject>();
                for (int m = 0; m < 1024; m++) {
                    tmpObjects.add(new GCPolicyDataObject(64));
                }
                tmpObjects=null;
            }
        }
        cacheObjects.size();
        cacheObjects=null;
    }

}

class GCPolicyDataObject{

    byte[] bytes=null;

    GCPolicyRefObject object=null;

    public GCPolicyDataObject(int factor){
        bytes=new byte[factor*1024];
        object=new GCPolicyRefObject();
    }

}

class GCPolicyRefObject{

    GCPolicyRefChildObject object;

    public GCPolicyRefObject(){
}

```

```

        object=new GCPolicyRefChildObject();
    }

}

class GCPolicyRefChildObject{
    public GCPolicyRefChildObject(){
    }
}

```

以 -Xms680M -Xmx680M -Xmn80M -XX:+UseConcMarkSweepGC -XX:+PrintGCAplicationStoppedTime -XX:+UseCMSCompactAtFullCollection -XX:CMSMaxAbortablePrecleanTime=5 参数执行以上代码，通过 jstat 观察到的 GC 状况如下：

共触发 39 次 minor GC，耗时为 1.197 秒，共触发 21 次 CMS GC，耗时为 0.136 秒，GC 总耗时为 1.333 秒。

GC 动作造成应用暂停的时间为：1.74 秒。

以-Xms680M -Xmx680M -Xmn80M -XX:+PrintGCAplicationStoppedTime -XX:+UseParallelGC 参数执行以上代码，通过 jstat 观察到的 GC 状况如下：

共触发 119 次 minor GC，耗时为 2.774 秒，共触发 8 次 Full GC，耗时为 0.243 秒，GC 总耗时为 3.016 秒。

GC 动作造成应用暂停的时间为：3.11 秒。

从上面的结果来看，由于 CMS GC 多数动作是和应用并发进行的，确实可以减小 GC 动作给应用造成的暂停。

大部分 Web 应用在处理请求时设置了一个最大可同时处理的请求数，当超过此请求数时，会将之后的请求放入等待队列中，而这个等待队列也限制了大小。当等待队列满了后仍然有请求进入，那么这些请求将会直接被丢弃，所有的请求又都是有超时限制的。在这种情况下如触发了造成应用暂停时间较长的 Full GC，那么有可能在这次 Full GC 后，应用中很多请求就超时或被丢弃了。例如请求的超时时间为 3 秒，在排队时应用中触发了一次 Full GC，造成了 3 秒的暂停，那么之前在此应用上等待处理的请求就全部超时了。从上可看出，Web 应用非常需要一个对应用造成暂停时间短的 GC，再加上大部分 Web 应用的瓶颈都不在 CPU 上。因此对于 Web 应用而言，在 G1 还不够成熟的情况下，CMS GC 是不错的选择。

## JVM 调优案例

以下为一个系统一段时间内的 GC 状况，此系统运行的机器操作系统为 32 位，CPU 为 4 核，物理内存为 4GB，启动参数为：

```
-server -Xms1536m -Xmx1536m -Xmn700m -XX:PermSize=96m -XX:MaxPermSize=96m
```

在系统运行到 67919.837 秒时发生了一次 Full GC，日志信息如下：

```
67919.817: [GC [PSYoungGen: 588706K->70592K(616832K)]  
1408209K->906379K(1472896K), 0.0197090 secs] [Times: user=0.06 sys=0.00,  
real=0.02 secs]  
67919.837: [Full GC [PSYoungGen: 70592K->0K(616832K)] [PSOldGen:  
835787K->375316K(856064K)] 906379K->375316K(1472896K) [PSPermGen:  
64826K->64826K(98304K)], 0.5478600 secs] [Times: user=0.55 sys=0.00, real=0.55  
secs]
```

之后的一段时间内则进行了多次 Minor GC，Minor GC 的信息摘取后如图 5.15 所示：

| 运行时间(秒)   | 回收前内存(新生代)(K) | 回收后内存(新生代)(K) | 回收前Heap内存(K) | 回收后Heap内存(K) |
|-----------|---------------|---------------|--------------|--------------|
| 67928.498 | 517582        | 79330         | 892898       | 454647       |
| 67936.102 | 600674        | 65291         | 975991       | 461745       |
| 67943.903 | 586635        | 75470         | 983089       | 478401       |
| 67951.596 | 599299        | 69213         | 1002230      | 489764       |
| 67959.4   | 593364        | 90948         | 1013916      | 524668       |
| 67967.135 | 610884        | 84565         | 1044604      | 540072       |
| 67974.962 | 604501        | 74881         | 1060008      | 539665       |
| 67982.773 | 595201        | 80943         | 1059985      | 558693       |
| 67990.266 | 601019        | 94343         | 1078769      | 583736       |
| 67998.25  | 609415        | 83303         | 1098808      | 603481       |
| 68005.724 | 598375        | 75359         | 1118553      | 618518       |
| 68013.166 | 593072        | 75485         | 1136231      | 634750       |
| 68021.166 | 593309        | 82164         | 1152574      | 651869       |
| 68028.848 | 604336        | 80730         | 1174041      | 666878       |
| 68036.946 | 602899        | 72126         | 1189046      | 671185       |
| 68045.671 | 597950        | 87318         | 1197009      | 699782       |
| 68054.738 | 612724        | 77563         | 1225188      | 712503       |
| 68062.25  | 601437        | 78306         | 1236377      | 723256       |
| 68070.145 | 602759        | 82143         | 1247708      | 735654       |
| 68078.762 | 611231        | 83398         | 1264742      | 748412       |
| 68086.71  | 612313        | 93628         | 1277327      | 778526       |
| 68094.576 | 601148        | 94080         | 1286046      | 812816       |
| 68102.945 | 601600        | 89560         | 1320336      | 830992       |
| 68110.533 | 593688        | 78322         | 1335120      | 849042       |
| 68117.99  | 582064        | 87033         | 1352785      | 868130       |
| 68124.878 | 594489        | 87892         | 1375586      | 894380       |
| 68132.862 | 594736        | 63715         | 1401225      | 891090       |

图 5.15 JVM 调优案例--Minor GC 信息

在 68132.893 时又发生了一次 Full GC，日志信息如下：

```
68132.862: [GC [PSYoungGen: 594736K->63715K(609920K)]  
1401225K->891090K(1465984K), 0.0309810 secs] [Times: user=0.06 sys=0.01,  
real=0.04 secs]  
68132.893: [Full GC [PSYoungGen: 63715K->0K(609920K)] [PSOldGen:
```

```
827375K->368026K(856064K) ] 891090K->368026K(1465984K) [PSPermGen:  
64869K->64690K(98304K) ], 0.5341070 secs] [Times: user=0.53 sys=0.00, real=0.53  
secs]
```

从以上信息来看，系统在 213 秒内就发生了一次 Full GC，27 次 minor GC，这显然过于频繁，在优化时仍然本着减少 Full GC 的首要原则来看。

在 67919.837 秒进行 Full GC 后，旧生代的内存消耗已下降到 375316K，而在 213 秒后的 Full GC 执行时，旧生代的内存消耗已上涨到了 827375KB。根据对这 213 秒中 27 次 minor GC 的信息分析来看，几乎每次 minor GC 后都会有部分对象从新生代转入了旧生代，但从第二次 Full GC 来看，这些转入旧生代的对象其实都是可以被回收的，从这个角度来看，也就是说，只要这些对象能够在新生代多存活一段时间，那么就可以在 minor GC 阶段被回收掉。

根据启动的参数以及机器配置来看，目前系统新生代 GC 的类型为 Parallel Scavenge，启动时 Eden Space 为 525MB，S0、S1 分别为 87.5MB。但随着系统运行，在当前的参数下 Eden、S0、S1 将会由 HotSpot 自动调整，于是查看日志对应时间，Eden、From、To 的空间大小，多数情况下大致为：561、59 和 72 这样的分配情况，假设多数情况下均为这样的占比，目前的运行状况为每次 Minor GC 后大概有 16MB 对象进入旧生代，假设这些对象都是因为超过 Survivor Space 才进入旧生代的，那么可以认为每次 Minor GC 时有  $16+72 = 88$ MB 的对象是存活的。从目前 Full GC 的状况来看，其实这 88MB 的对象在一段时间后都是可以回收的，那么理论上来说只用将 Survivor Space 扩大到 88MB 以上即可，按照这个想法，可将启动参数调整为：

```
-server -Xms1536m -Xmx1536m -Xmn700m -XX:PermSize=96m -XX:MaxPermSize=96m  
-XX:InitialSurvivorRatio=7 -XX: -UseAdaptiveSizePolicy
```

但调整为这样的参数后，Eden Space 就缩小为 500MB，Minor GC 的执行会更加频繁，但由于 Survivor Space 的扩大，进入旧生代的对象就缩小了，从而减少了 Full GC 发生的几率，更好的方法则是调大整个 JVM Heap，并同比例增大新生代空间。在增大新生代空间后，要注意观察 Minor GC 消耗的时间。

除了以上方法外，还可采用的一种方法为将 GC 策略调整为 CMS GC。从目前的数据来看，如采用 CMS GC，预计会在 14 次 minor GC 后触发一次 CMS GC，按照现在的间隔时间，也就是差不多  $14 \times 8 = 112$  秒的时候执行一次 CMS GC，此时之前从新生代转入旧生代的对象应大部分都可回收，只要 CMS GC 耗时不太长。那么调整为 CMS GC 后就基本可做到不触发 Full GC 了，可采用类似如下的参数：

```
-server -Xms1536m -Xmx1536m -Xmn700m -XX:PermSize=96m -XX:MaxPermSize=96m  
-XX:+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection  
-XX:CMSMaxAbortablePrecleanTime=500 -XX:+CMSPermGenSweepingEnabled  
-XX:+CMSClassUnloadingEnabled
```

在做完以上的调优后，都需继续结合 jstat 工具来查看是否达到了调优的目标，如果没有则需要继

续按照以上步骤进行参数的调整。

在进行参数调整时，可根据目前收集到的顶峰时系统请求次数、响应时间以及 GC 的信息，来估计系统每次请求需要消耗的内存，以及每次 Minor GC 时存活的对象所占的内存，从而估计需要设置多大的 Survivor 才能够尽可能地避免对象进入旧生代。例如每秒的请求为 60 次，GC 信息显示每 10 秒执行一次 minor GC，每次 Minor GC 会有 10MB 对象转入旧生代，每次 Minor GC 在 Eden 分配的内存为 600MB，Survivor 为 100MB。根据这些信息，可以简单认为系统中每次请求消耗的内存大致为 1MB，并粗略估计为在开始 Minor GC 时，还有 110 个请求未处理完。对于这样的状况，简单的调优方式可以为在保持 Eden Space 600 MB 的情况下，将 Survivor Space 增长到 120MB，那就基本可以做到在当前的响应速度下，如 10 秒内接受的请求最多为 600 个时，Minor GC 时大部分情况不会有对象转入旧生代，但毕竟系统中的请求响应时间、内存消耗分布不会这么平均，并且还会出现直接在旧生代分配的现象，因此通常按这样粗略的估计设置的参数仍然会达不到目标，要继续进行一些细微的调节来逐步达到目标。

由于参数的估计是以请求次数和响应时间为基准的，因此一旦系统的响应速度下降或请求的次数上升，就可能仍然会导致大量对象进入旧生代，从而触发频繁的 Full GC，频繁的 Full GC 又导致系统的响应速度下降，从这个层面来看，根本上需要做的调优仍然是提升请求的处理速度以及降低每次请求需要分配的内存，只有这样才能使得应用能够支撑更高的并发量，否则就会随着并发量的上涨而迅速出现瓶颈。

旧生代大小的调整一方面要依据新生代的大小，另外一方面要依据系统中持久存活的对象会消耗多大的内存来决定。

如系统不是 CPU 密集型，且从新生代进入旧生代的大部分对象是可回收的，那么采用 CMS GC 可以更好地在旧生代满之前完成对象的回收，更大程度降低 Full GC 发生的可能。

目前内存管理方面，JVM 自身已经做得非常不错了，因此如果不是有确切的 GC 造成性能低的理由，就没必要做过多细节方面的调优（例如 survivor 区大小的设置等）。多数情况下只须选择 GC 策略并设置 JVM Heap 的大小即可。在调整了内存管理方面的参数后应通过 -XX:+PrintGCDetails、-XX:+PrintGCTimeStamps、-XX:+PrintGCApplicationStoppedTime 及 jstat 或 visualvm 等方式来观察调整后 GC 的状况，除内存管理方面的调优外，Sun JDK 还提供了一些其他方面的调优参数：如 -XX:CompileThreshold、-XX:+UseFastAccessorMethods 及 -XX:+UseBiasedLocking 等。

除了以上基于对 JDK 实现及 JDK 调优参数的掌握进行的调优外，关注 JDK 的新版本也是不错的选择。每次 JDK 新版本的发布几乎都会在性能上做出一些优化，也许这些优化正是应用所需要的，那就事半功倍了。

## 5.2.2 程序调优

程序调优根据资源的消耗情况及分析，对程序的实现进行一定的调优，下面就来看看常用的一些调优方法。

## CPU 消耗严重的解决方法

### 1. CPU us 高的解决方法

根据之前的分析，CPU us 高的原因主要是执行线程无任何挂起动作，且一直执行，导致 CPU 没有机会去调度执行其他的线程，造成线程饿死的现象。对于这种情况，常见的一种优化方法是对这种线程的动作增加 Thread.sleep，以释放 CPU 的执行权，降低 CPU 的消耗。

按照这样的思想，对 5.1.1 节“CPU 消耗分析”中的例子进行修改，在往集合中增加元素的部分增加 sleep，修改如下：

```
for (int k = 0; k < 10000; k++) {
    list.add(str+String.valueOf(k));
    if (k%50==0) {
        try{
            Thread.sleep(1);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

重新执行以上代码，通过 top 查看效果如图 5.16 所示：

```
top - 15:43:28 up 330 days, 37 min, 3 users, load average: 0.05, 0.68, 0.53
Tasks: 539 total, 1 running, 538 sleeping, 0 stopped, 0 zombie
Cpu0 : 7.0% us, 0.0% sy, 0.0% ni, 93.0% id, 0.0% wa, 0.0% hi, 0.0% si
Cpu1 : 5.3% us, 0.0% sy, 0.0% ni, 94.7% id, 0.0% wa, 0.0% hi, 0.0% si
Cpu2 : 5.3% us, 0.0% sy, 0.0% ni, 94.7% id, 0.0% wa, 0.0% hi, 0.0% si
Cpu3 : 5.6% us, 0.7% sy, 0.0% ni, 93.7% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 4151276k total, 3654356k used, 496920k free, 160140k buffers
Swap: 2096472k total, 192k used, 2096280k free, 1167940k cached

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
26981 admin     16   0 1223m 185m 6568 S      5  4.6   0:01.61 java
26982 admin     16   0 1223m 185m 6568 S      5  4.6   0:01.62 java
26984 admin     16   0 1223m 185m 6568 S      5  4.6   0:01.62 java
26983 admin     16   0 1223m 185m 6568 S      5  4.6   0:01.61 java
```

图 5.16 top 查看优化后代码的 CPU us 的消耗

从上结果可见，CPU 的消耗大幅度下降，当然，这种修改方式是以损失单次执行性能为代价的，但由于降低了 CPU 的消耗，对于多线程的应用而言，反而提高了总体的平均性能。

在实际的 Java 应用中会有很多类似的场景，例如多线程的任务执行管理器，它通常要通过扫描任务集合列表来执行任务。对于这些类似的场景，都可通过增加一定的 sleep 时间来避免消耗过多的 CPU。

除了上面的场景外，还有一种经典的场景是状态的扫描，例如某线程要等其他线程改变了值后才可继续执行。对于这种场景，最佳的方式是改为采用 wait/notify 机制。

对于其他类似循环次数太多、正则、计算等造成的 CPU us 过高的状况，则要结合业务需求来进行调优。

对于 GC 频繁造成 CPU us 高的现象，则要通过 JVM 调优或程序调优，降低 GC 的执行次数。

## 2. CPU sy 高的解决方法

CPU sy 高的原因主要是线程的运行状态要经常切换，对于这种情况，最简单的优化方法是减少线程数。

按照这样的思路，将 CPU 资源消耗中的例子重新执行，将线程数降低，传入参数 100，执行结果如图 5.17 所示：

| 16时12分31秒 | CPU | %user | %nice | %system | %iowait | %idle |
|-----------|-----|-------|-------|---------|---------|-------|
| 16时12分32秒 | a11 | 0.00  | 0.00  | 0.50    | 0.00    | 99.50 |
| 16时12分33秒 | a11 | 0.25  | 0.00  | 1.01    | 0.00    | 98.74 |
| 16时12分34秒 | a11 | 0.25  | 0.00  | 0.50    | 0.00    | 99.24 |
| 16时12分35秒 | a11 | 0.25  | 0.00  | 0.50    | 0.00    | 99.24 |
| 16时12分36秒 | a11 | 0.00  | 0.00  | 0.25    | 0.00    | 99.75 |

图 5.17 sar 查看优化后 cpu sy 的消耗

可见减少线程数是能让 sy 值下降的，所以不是线程数越多吞吐量就越高，线程数需要设置为合理的值，这要根据应用情况来具体决定，同时使用线程池避免要不断地创建线程。如应用要支撑大量的并发，在减少线程数的情况下最好是增加一个缓冲队列，避免因为线程数的减少造成系统出错率上升。

造成 CPU sy 高的原因除了启动的线程过多以外，还有一个重要的原因是线程之间锁竞争激烈，造成了线程状态经常要切换，因此尽可能降低线程间的锁竞争也是常见的优化方法。锁竞争降低后，线程的状态切换的次数也就会下降，sy 值会相应下降。但值得注意的是，如线程数过多，调优后有可能会造成 us 值过高，所以合理地设置线程数非常关键。锁竞争更有可能造成系统资源消耗不多，但系统性能不足的现象，因此关于降低线程之间锁竞争的调优技巧放入了后续的章节中进行讲述。

除了以上两种情况外，对于分布式 Java 应用而言，还有一种典型现象是应用中有较多的网络 IO 操作或确实需要一些锁竞争机制（例如数据库连接池），但为了能够支撑高的并发量，在 Java 应用中又只能借助启动更多的线程来支撑<sup>11</sup>。在这样的情况下当并发量增长到一定程度后，可能会造成 CPU sy 高的现象，对于这种现象，可采用协程（Coroutine<sup>12</sup>）来支撑更高的并发量，避免并发量上涨后造成 CPU sy 消耗严重、系统 load 迅速上涨和系统性能下降。

在目前的 Sun JDK 实现中，创建并启动一个 Thread 对象就意味着运行了一个原生线程，当这个线程中有任何的阻塞动作（例如同步文件 IO、同步网络 IO、锁等待、Thread.sleep 等）时，这个线程就会被挂起，但仍然占据着线程的资源。当线程中的阻塞动作完成时，由操作系统来恢复线程的上下文，并调度执行，这是一种标准的遵循目前操作系统的实现方式，这种方式对于 Java 应用而言，当并发量上涨后，有可能出现的现象是启动的大量线程都处于浪费状态。例如一个线程在等待数据库执行结果的返回，如这个数据库执行操作需要花费 2 秒，那么就意味着这个线程资源被白白占用了 2 秒，一方面导致了其他的请求只能是放在缓冲队列中等待执行，性能下降；另一方面是造成系统中线程切换频

<sup>11</sup> 在 JRockit 中可选择是采用原生的线程或模拟的线程

<sup>12</sup> <http://en.wikipedia.org/wiki/Coroutine>

繁, CPU 运行队列过长, 协程要改变的就是不浪费相对昂贵的原生线程资源。

采用协程后, 能做到当线程等待数据库执行结果时, 就立即释放此线程资源给其他请求, 等到数据库执行结果返回后才继续执行, 在 Java 中目前主要可用于实现协程的框架为 Kilim<sup>13</sup>。在使用 Kilim 执行一项任务时, 并不创建 Thread, 而是改为创建 Task, Task 相对于 Thread 而言就轻量级多了。当此 Task 要做阻塞动作时, 可通过 Mailbox.get 或 Task.pause 来阻塞当前 Task, Kilim 会保存 Task 之后执行需要的对象信息, 并释放 Task 执行所占用的线程资源; 当 Task 的阻塞动作完成或被唤醒时, 此时 Kilim 会重新载入 Task 所需的对象信息, 恢复 Task 的执行, 相当于 Kilim 来承担了线程的调度以及上下文切换动作。这种方式相对原生 Thread 方式更为轻量, 且能够更好地利用 CPU, 因此可做到仅启动 CPU 核数的线程数, 以及大量的 Task 来支撑高并发量, Kilim 带来的是线程使用率的提升, 但同时由于要在 JVM 堆中保存 Task 上下文信息, 因此在采用 Kilim 的情况下要消耗更多的内存。

下面是一个传统方式和基于 Kilim 采用 Coroutine 方式支撑高并发请求对比的例子。

传统方式如下:

```
public class SimpleBenchMark {

    private CountDownLatch latch=new CountDownLatch(10000);

    private Random random=new Random();

    public static void main(String[] args) throws Exception{
        SimpleBenchMark benchmark=new SimpleBenchMark();
        benchmark.runBenchmark();
    }

    public void runBenchmark() throws Exception{
        long beginTime=System.currentTimeMillis();
        ExecutorService executor=Executors.newFixedThreadPool(400);
        for (int i = 0; i < 10000; i++) {
            executor.execute(new Handler());
        }
        latch.await();
        System.out.println("Consume Time:
"+(System.currentTimeMillis()-beginTime)+" ms");
        executor.shutdown();
    }

    class Handler implements Runnable{

        @Override
        public void run() {
```

<sup>13</sup> <http://www.malhar.net/sriram/kilim>

```

        BlockingQueue<String> resultQueue=new ArrayBlockingQueue<String>(1);
        try {
            resultQueue.poll(random.nextInt(10)+100, TimeUnit.MILLISECONDS);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        latch.countDown();
    }

}

```

为了避免随着并发量的上涨线程数无限增长，这里采用了一个固定大小的线程池。

基于 Kilim 采用 Coroutine 的方式如下：

```

public class SimpleBenchMark {

    private CountDownLatch latch=new CountDownLatch(10000);

    private Random random=new Random();

    public static void main(String[] args) throws Exception{
        SimpleBenchMark benchmark=new SimpleBenchMark();
        benchmark.runBenchmark();
    }

    public void runBenchmark() throws Exception{
        long beginTime=System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            new Handler().start();
        }
        latch.await();
        System.out.println("Consume Time:
"+(System.currentTimeMillis()-beginTime)+" ms");
        System.exit(0);
    }

    class Handler extends Task{

        public void execute() throws Pausable,Exception{
            Mailbox<String> resultMailbox=new Mailbox<String>(1);
            resultMailbox.get(random.nextInt(10)+100);
            latch.countDown();
        }
    }
}

```

```

    }
}

}

```

在一台 linux 机器 (2 核 Intel(R) Xeon(R) CPU E5410 @ 2.33GHz, 2GB 内存) 上执行, 传统方式耗时大概为 3077ms, 而基于 Kilim 采用协程方式的耗时大概为 277ms。可见在这种高并发的情况下, 协程方式对性能提升以及支撑更高的并发量可以起到很大的作用。

目前 Kilim 版本仅为 0.7, 而且没有商用的实际例子, 如打算在实际的系统中使用, 还需要慎重。一方面是 0.7 中基于 `object.wait/notify` 机制实现的 `Scheduler` 在高压下会出现 bug, 可自行基于 `ThreadPoolExecutor` 进行改造; 另一方面 `Mailbox.get(timeout)` 是基于 `Timer` 实现的, 由于 `Timer` 在增加 task 到队列时和运行 task 队列是互斥的 (即使是 `ScheduledThreadPoolExecutor` 也同样需要锁整个队列), 对于大并发的应用而言这里是个潜在的瓶颈。对于 Java 应用而言, `Timer` 是一个经常用来实现定时任务的类, 但 `Timer` 的性能在高并发下是一般的, 感兴趣的读者可以尝试基于 `TimerWheel` 算法<sup>14</sup> 来提升 `Timer` 的性能。

现在要在 Java 应用中使用 Kilim 来实现协程方式并没有例子中这么简单, 因为协程方式要求所有的操作都不阻塞原生线程, 这就要求应用中不能使用目前 Java 里的同步、锁等机制。除了这些之外, 还需要解决同步访问数据库、操作文件等问题, 这些都必须改为是异步方式或 Kilim 中的 Task 暂停的机制。目前 Sun JDK 7 中也有一个支持协程方式的实现, 感兴趣的读者可进一步阅读<sup>15</sup>, 另外基于 JVM 的 Scala 的 Actor<sup>16</sup> 也可用于在 Java 中使用协程。

除了软件方面对提升 CPU 使用率做出的努力外, 硬件方面的 CPU 专业化 (例如 GPU 进行图形计算) 也很值得关注, 这些同样有可能会给 Java 应用带来一些帮助。

## 文件 IO 消耗严重的解决方法

从程序角度而言, 造成文件 IO 消耗严重的原因主要是多个线程在写大量的数据到同一文件, 导致文件很快变得很大, 从而写入速度越来越慢, 并造成各线程激烈争抢文件锁, 对于这类情况, 常用的调优方法有以下几种。

- 异步写文件

将写文件的同步动作改为异步动作, 避免应用由于写文件慢而性能下降太多, 例如写日志, 可以使用 log4j 提供的 `AsyncAppender`。

- 批量读写

<sup>14</sup> <http://www.ibm.com/developerworks/aix/library/au-lowertime/index.html#N100BC>

<sup>15</sup> <http://weblogs.java.net/blog/forax/archive/2009/11/19/holy-crap-jvm-has-coroutinecontinuationfiber-etc>

<sup>16</sup> <http://www.scala-lang.org/node/242>

频繁的读写操作对 IO 消耗会很严重，批量操作将大幅度提升 IO 操作的性能。

- 限流

频繁读写的另外一个调优方式是限流，从而将文件 IO 消耗控制到一个能接受的范围，例如通常在记录日志时会采用如下方式：

```
log.error(errorInfo, throwable);
```

如以上方式不做任何处理，在大量出现异常时，会出现所有的线程都在执行 log.error(...)，此时可采取的一个简单策略为统计一段时间内 log.error 的执行频率。当超过这个频率时，一段时间内不再写 log，或塞入一个队列后缓慢地写，简单的一段实现代码示例如下：

```
public class LogControl{
    private static final long INTERVAL=1000;
    private static final long PUNISH_TIME=5000;
    private static final int ERROR_THRESHOLD=100;
    private static AtomicInteger count;
    private static long beginTime;
    private static long punishTimeEnd;
    // 由于控制不用非常精确，因此忽略此处的并发问题
    public static boolean isLog(){
        // 如尚处于不写日志阶段，则返回 false
        if((punishTimeEnd>0)&&punishTimeEnd<System. currentTimeMillis()){
            return false;
        }
        // 如 count 为 0，则说明是重新计数，设置 beginTime
        if(count.getAndIncrement()==0){
            beginTime=System. currentTimeMillis();
            return true;
        }
        // 如已在计数
        else{
            // 判断累积数是否超过了阈值，超过了则将 count 置为 0，并设置一定的不写日志的时间
            if(count> ERROR_THRESHOLD){
                count.set(0);
                punishTimeEnd =PUNISH_TIME+ System. currentTimeMillis();
                return false;
            }
            // 如累积数没超过阈值，且当前时间已超过计数周期，则重新计算
            else if(System. currentTimeMillis()>(beginTime+INTERVAL)){
                count.set(0);
            }
            return true;
        }
    }
}
```

```

    }
}

if(LogControl.isLog()){
    log.error(errorInfo,throwable);
}

```

- 限制文件大小

操作太大的文件也是造成文件 IO 效率低的一个原因，因此对于每个输出的文件，都应做大小的限制，在超出最大值后可生成一个新的文件，类似 log4j 中 RollingFileAppender 的 maxFileSize 属性的作用。

除了以上这些外，还有就是尽可能采用缓冲区等方式来读取文件内容，避免不断与操作系统交互，具体可参见 Sun 官方的关于 Java 文件 IO 优化的文章<sup>17</sup>。

## 网络 IO 消耗严重的解决方法

从程序角度而言，造成网络 IO 消耗严重的原因主要是同时需要发送或接收的包太多。对于这类情况，常用的调优方法为进行限流，限流通常是限制发送 packet 的频率，从而在网络 IO 消耗可接受的情况下发送 packet。

## 对于内存消耗严重的情况

在内存消耗方面，最明显的在于消耗了过多的 JVM Heap 内存，造成 GC 频繁执行的现象，而物理内存方面的消耗通常来说不会成为 Java 应用中的主要问题。除了 JVM 的调优外，在寻找到内存消耗严重的代码后，可从代码本身进行优化，避免内存资源消耗过多，此处就介绍一些 JVM Heap 内存消耗严重时常用的程序调优方法。

### 1. 释放不必要的引用

内存消耗严重的情况中最典型的一种现象是代码中持有了不需要的对象引用，造成这些对象无法被 GC，从而占据了 JVM 堆内存。这种情况最典型的一个例子是在复用线程的情况下使用 ThreadLocal，由于线程复用，ThreadLocal 中存放的对象如未做主动释放的话则不会被 GC，代码例子如下：

```

public class ThreadLocalDemo {

    public static void main(String[] args) throws Exception{
        ThreadLocalDemo demo=new ThreadLocalDemo();
        demo.run();
    }

    public void run(){

```

<sup>17</sup> <http://java.sun.com/developer/technicalArticles/Programming/PerfTuning/>

```

        ExecutorService executor=Executors.newFixedThreadPool(1);
        executor.execute(new Task());
        System.gc();
    }

    class Task implements Runnable{

        public void run() {
            ThreadLocal<byte[]> localString=new ThreadLocal<byte[]>();
            localString.set(new byte[1024*1024*30]);
        }
    }
}

```

执行上面的代码，通过 jstat 观察，会发现在 Old Generation JVM 内存一直被使用了 30MB 左右。对于这种情况，要注意在线程内的动作执行完毕时执行 ThreadLocal.set 把对象清除，避免持有不必要的对象引用。

## 2. 使用对象缓存池

创建对象的实例要耗费一定的 CPU 以及内存，使用对象缓存池一定程度上可降低 JVM Heap 内存的使用。

一个简单的示例如下：

```

public class ObjectPoolDemo {private static int executeTimes=10;
private static int maxFactor=10;
private static int threadCount=100;
private static final int NOTUSE_OBJECTPOOL=1;
private static final int USE_OBJECTPOOL=2;
private static int runMode=NOTUSE_OBJECTPOOL;
private static CountDownLatch latch=null;

public static void main(String[] args) throws Exception{
    Thread.sleep(20000);
    if(args.length==1)
        runMode=Integer.parseInt(args[0]);
    if(args.length==2){
        runMode=Integer.parseInt(args[0]);
        executeTimes=Integer.parseInt(args[1]);
    }
    if(args.length==3){
        runMode=Integer.parseInt(args[0]);
    }
}

```

```

        executeTimes=Integer.parseInt(args[1]);
        maxFactor=Integer.parseInt(args[2]);
    }
    if(args.length==4){
        runMode=Integer.parseInt(args[0]);
        executeTimes=Integer.parseInt(args[1]);
        maxFactor=Integer.parseInt(args[2]);
        threadCount=Integer.parseInt(args[3]);
    }
    long beginTime=System.currentTimeMillis();
    Task task=new Task();
    for (int i = 0; i < executeTimes; i++) {
        System.out.println("Round: "+(i+1));
        latch=new CountDownLatch(threadCount);
        for (int j = 0; j < threadCount; j++) {
            new Thread(task).start();
        }
        latch.await();
    }
    long endTime=System.currentTimeMillis();
    System.out.println("Execute summary: Round( "+executeTimes+" ) Thread Per
Round( 100 ) Object Factor( "+maxFactor+" ) Execute Time( "+(endTime-beginTime)+" 
ms )");
}
static class Task implements Runnable{
public void run() {
    for (int j = 0; j < maxFactor; j++) {
        if(runMode==USE_OBJECTPOOL){
            BigObjectPool.getInstance().getBigObject(j);
        }
        else{
            new BigObject(j);
        }
    }
    latch.countDown();
}
}
static class BigObjectPool{

private static final BigObjectPool self=new BigObjectPool();

private final Map<Integer, BigObject> cacheObjects=new HashMap<Integer,
BigObject>();

private BigObjectPool(){
;
}
}

```

```

public static BigObjectPool getInstance() {
    return self;
}
public BigObject getBigObject(int factor) {
    if(cacheObjects.containsKey(factor)) {
        return cacheObjects.get(factor);
    }
    else{
        BigObject object=new BigObject(factor);
        cacheObjects.put(factor, object);
        return object;
    }
}
static class BigObject{
    private byte[] bytes=null;
    public BigObject(int factor){
        bytes=new byte[(factor+1)*1024*1024];
    }
    public byte[] getBytes() {
        return bytes;
    }
}
}

```

以-Xms128M -Xmx128M -Xmn64M 的参数执行上面的代码，执行结果如下：

```
Execute summary: Round( 10 ) Thread Per Round( 100 ) Object Factor( 10 ) Execute Time( 27494 ms)
```

对其执行过程中的 GC 信息进行统计，结果为：

共执行 1137 次 minor GC，总耗时为 6.959 秒，共执行 222 次 Full GC，耗时 1.969 秒，minor GC+Full GC 共耗时为 8.928 秒。

加上执行参数 2，重新执行上面的代码，执行结果如下：

```
Execute summary: Round( 10 ) Thread Per Round( 100 ) Object Factor( 10 ) Execute Time( 563 ms)
```

GC 信息为：

共执行了 2 次 minor GC，总耗时为 0.161 秒，共执行了 1 次 Full GC，耗时为 0.045 秒，minor GC+Full GC 共耗时为 0.206 秒。

从上面的结果对比可看出，在内存消耗严重的情况下，采用对象缓存池可大幅度提升性能，避免

创建对象所耗费的时间及频繁 GC 造成的消耗。

### 3. 采用合理的缓存失效算法

上面说到了采用对象缓存池来降低内存的消耗，但如果放入太多的对象在缓存池中，反而会造成内存的严重消耗。同时由于缓存池一直对这些对象持有引用，从而会造成 Full GC 增多，对于这种状况要合理控制缓存池的大小。

控制缓存池大小的问题在于当到达缓存池的最大容量后，如果要加入新的对象该如何处理，有一些经典的缓存失效算法来清除缓存池中的对象，例如 FIFO、LRU、LFU 等。采用这些算法可控制缓存池中的对象数目，避免缓存池中的对象数量无限上涨。

在 Java 中，基于 LinkedHashMap 可简单实现支持 FIFO 和 LRU 策略 的 CachePool，代码如下：

```
public class ObjectCachePool<K, V> {

    public static final int FIFO_POLICY=1;

    public static final int LRU_POLICY=2;

    private static final int DEFAULT_SIZE=10;

    private Map<K, V> cacheObjects;

    public ObjectCachePool(){
        this(DEFAULT_SIZE);
    }

    public ObjectCachePool(final int size){
        this(size,FIFO_POLICY);
    }

    public ObjectCachePool(final int size,final int policy){
        switch (policy) {
            case FIFO_POLICY:
                cacheObjects=new LinkedHashMap<K, V>(size){

                    private static final long serialVersionUID = 1L;

                    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
                        return size() > size;
                    }
                };
                break;
            case LRU_POLICY:
                cacheObjects=new LinkedHashMap<K, V>(size,0.75f,true){

```

```
private static final long serialVersionUID = 1L;

protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
    return size() > size;
}

};

break;
default:
    throw new IllegalArgumentException("Unknown policy: "+policy);
}
}

public void put(K key,V value){
    cacheObjects.put(key, value);
}

public V get(K key){
    return cacheObjects.get(key);
}

public void remove(K key){
    cacheObjects.remove(key);
}

public void clear(){
    cacheObjects.clear();
}
```

#### 4. 合理使用 SoftReference 和 WeakReference

对于占据内存但又不是必须存在的对象，例如缓存对象，也可以基于 SoftReference 或 WeakReference 的方式来进行缓存。根据之前第 3 章中的描述，SoftReference 的对象会在内存不够用的时候回收，WeakReference 的对象则会在 Full GC 的时候回收，采用这两种方式也能一定程度上减少 JVM Heap 区内存的消耗。

对于以上硬件资源消耗过多造成性能不足的现象，除了软件方面调优外，在大多数情况下还可通过升级或增加硬件来提升程序的性能。

### 5.2.3 对于资源消耗不多，但程序执行慢的情况

对于分布式 Java 应用而言，造成这种情况的主要原因通常有锁竞争激烈及未充分发挥硬件资源两种。

## 锁竞争激烈

线程多了后，锁竞争的状况会比较明显，这时线程很容易处于等待锁的状况，从而导致性能下降以及 CPU sy 上升，典型的例子如下：

```

public class LockHotDemo {

    private static CountDownLatch latch;

    private static int threadCount=Runtime.getRuntime().availableProcessors()*100;

    private static int executeTimes=10;

    public static void main(String[] args) throws Exception{
        if((args.length==1) || (args.length==2))
            threadCount=Integer.parseInt(args[0]);
        if(args.length==2)
            executeTimes=Integer.parseInt(args[1]);
        HandleTask task=new HandleTask();
        long beginTime=System.currentTimeMillis();
        for (int i = 0; i < executeTimes; i++) {
            System.out.println("Round: "+(i+1));
            latch=new CountDownLatch(threadCount);
            for (int j = 0; j < threadCount; j++) {
                new Thread(task).start();
            }
            latch.await();
        }
        long endTime=System.currentTimeMillis();
        System.out.println("Execute summary: Round( "+executeTimes+" ) ThreadCount Per Round( "+threadCount+" ) Execute Time( "+(endTime-beginTime)+" ms)");
    }

    static class HandleTask implements Runnable{

        private final Random random=new Random();

        public void run() {
            Handler.getInstance().handle(random.nextInt(10000));
            latch.countDown();
        }
    }

    static class Handler{
}

```

```

private static final Handler self=new Handler();

private final Random random=new Random();

private final Lock lock=new ReentrantLock();

private Handler(){

}

public static Handler getInstance(){

    return self;
}

public void handle(int id){

    try{

        lock.lock();
        // execute sth
        try {

            Thread.sleep(random.nextInt(10));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    finally{
        lock.unlock();
    }
}
}

```

这是多线程程序中常见的一种场景，在一个 2 核的机器上执行以上代码，执行完毕后的统计信息如下：

|   |
|---|
| Execute summary: Round( 10 ) ThreadCount Per Round( 200 ) Execute Time( 36143 ms) |
|---|

此时各方面的资源消耗并不高，但性能比没有锁的情况下下降了非常多。从这个例子可见，锁是影响性能的重要因素，但为了保证资源的一致性，多线程应用中锁的使用是不可避免的，只能尽量去降低线程间的锁竞争，常见的方法如下。

## 1. 使用并发包中的类

根据之前在 4.2 节中对于并发包中类的分析，可以看出，并发包中的类多数都采用了 lock-free、nonblocking 算法，减少了多线程情况下资源的锁竞争，因此对于线程间要共享操作的资源而言，应尽量使用并发包中的类来实现。这一点从深入 JDK 一章的性能测试结果（AtomicInteger、ConcurrentHashMap 等）中可看出，如并发包中的类无法满足需求时，可参考学习一些 nonblocking 算法来自行实现，nonblocking 算法的机制，为基于 CAS 来做到无需 lock 就可实现资源一致性的保证，主要的实现 nonblocking 的算法有：

## 2. 使用 Treiber 算法

Treiber 算法主要用于实现 Stack，基于 Treiber 算法实现的无阻塞的 Stack 代码如下：

```
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = head.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!head.compareAndSet(oldHead, newHead));
        return oldHead.item;
    }

    static class Node<E> {
        final E item;
        Node<E> next;
        public Node(E item) { this.item = item; }
    }
}
```

以上代码摘自 IBM DeveloperWorks 网站上的 Java theory and practice 系列文章<sup>18</sup>，由于 Stack 是 LIFO

<sup>18</sup> <http://www.ibm.com/developerworks/java/library/j-jtp04186/index.html>

方式，因此不能采取类似 `LinkedBlockingQueue` 中两把锁的机制。这里巧妙地采用 `AtomicReference` 来实现了无阻塞的 `push` 和 `pop`，在 `push` 时基于 `AtomicReference` 的 CAS 方法来比较目前的 `head` 是否一致。如不一致，说明有其他线程改动了，如有改动则继续循环，直到一致，才修改 `head` 元素，在 `pop` 时可采用同样的方式进行操作。

简单用一个多线程来对比无阻塞 `Stack` 和传统 `Stack` 的性能，测试方法为创建 300 个线程，每个线程均执行 10 次 `push` 和 `pop` 动作，代码如下：

```
public class StackBenchmark {
    private Stack<String> stack=new Stack<String>();
    private ConcurrentStack<String> concurrentStack=new ConcurrentStack<String>();
    private static final int THREAD_COUNT=300;
    private CountDownLatch latch=new CountDownLatch(THREAD_COUNT);
    private CyclicBarrier barrier=new CyclicBarrier(THREAD_COUNT);
    public static void main(String[] args) throws Exception{
        StackBenchmark benchmark=new StackBenchmark();
        benchmark.run();
    }
    public void run() throws Exception{
        long beginTime=System.currentTimeMillis();
        for (int i = 0; i < THREAD_COUNT; i++) {
            new Thread(new StackBenchmarkTask()).start();
        }
        latch.await();
        System.out.println("Stack consume Time:
"+(System.currentTimeMillis()-beginTime)+" ms");
        latch=new CountDownLatch(THREAD_COUNT);
        barrier=new CyclicBarrier(THREAD_COUNT);
        beginTime=System.currentTimeMillis();
        for (int i = 0; i < THREAD_COUNT; i++) {
            new Thread(new ConcurrentStackBenchmarkTask()).start();
        }
        latch.await();
        System.out.println("ConcurrentStack consume Time:
"+(System.currentTimeMillis()-beginTime)+" ms");
    }
    class StackBenchmarkTask implements Runnable{
```

```

@Override
public void run() {
    try {
        barrier.await();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    for (int i = 0; i < 10; i++) {
        stack.push(Thread.currentThread().getName());
        stack.pop();
    }
    latch.countDown();
}

class ConcurrentStackBenchmarkTask implements Runnable{

    @Override
    public void run() {
        try {
            barrier.await();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        for (int i = 0; i < 10; i++) {
            concurrentStack.push(Thread.currentThread().getName());
            concurrentStack.pop();
        }
        latch.countDown();
    }
}
}

```

在一个双核机器上运行以上代码，采用 `ConcurrentStack` 的情况下消耗的时间大概会比采用 `Stack` 的情况少 10ms 左右。

### 3. 使用 Michael-Scott 非阻塞队列算法<sup>19</sup>

和 Treiber 算法类似，Michael-Scott 算法也是基于 CAS 以及 `AtomicReference` 来实现队列的非阻塞操作，`java.util.concurrent` 中的 `ConcurrentLinkedQueue` 就是典型的基于 Michael-Scott 实现的非阻塞队列。

<sup>19</sup> [http://www.cs.rochester.edu/u/scott/papers/1996\\_PODC\\_queues.pdf](http://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf)

ConcurrentLinkedQueue 在执行 offer 动作时，通过 CAS 比较拿到的 tail 元素是否为当前处于末尾的元素，如不是则继续循环，如是则将 tail 元素更新为新的元素。

在执行 poll 动作时，通过 CAS 比较拿到的 head 元素是否为当前处于首位的元素，如不是则继续循环，如是则将 head 后的元素赋值给 head，同时获取之前 head 元素中的值并返回。

从上面两种算法来看，基于 CAS 和 AtomicReference 来实现无阻塞是不错的选择。但值得注意的是，由于 CAS 是基于不断的循环比较来保证资源一致性的，对于冲突较多的应用场景而言，CAS 会带来更高的 CPU 消耗，因此不一定采用 CAS 实现无阻塞的就一定比采用 Lock 方式的性能好。业界中还有一些无阻塞算法的改进，例如 MCAS、WSTM<sup>20</sup>等。

#### 4. 尽可能少用锁

尽可能让锁仅在需要的地方出现，通常没必要对整个方法加锁，而只对需要控制的资源做加锁操作。

假设锁竞争激烈的例子中有 5ms 左右的时间是耗在不用加锁的操作上，另外 5ms 用在加锁的操作上，那么重构 Handler 类的 handle 方法如下：

```
public void handle(int id) {
    // execute sth don't need lock
    try {
        Thread.sleep(random.nextInt(5));
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
    try{
        lock.lock();
        // execute sth
        try {
            Thread.sleep(random.nextInt(5));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    finally{
        lock.unlock();
    }
}
```

重新执行，执行后的统计信息如下。

<sup>20</sup> <http://www.cl.cam.ac.uk/research/srg/netos/papers/2007-cpwl.pdf>

```
Execute summary: Round( 10 ) ThreadCount Per Round( 200 ) Execute Time( 31935 ms)
```

从统计信息来看，将锁最小化后，性能有了不小的提升，所以在编写多线程程序时，要仔细考虑哪些地方是要加锁的，哪些地方是不要加锁的。尽可能让锁最小化，只对互斥及原子操作的地方加锁，加锁时尽可能以被保护资源的最小粒度为单位。例如一个操作中需要保护的资源只有 `HashMap`，那么在加锁时则可只 `synchronized(map)`，而没必要 `synchronized(this)`，并且可以只在对 `HashMap` 操作时才加锁。

## 5. 拆分锁

拆分锁即把独占锁拆分为多把锁，常见的有读写锁拆分及类似 `ConcurrentHashMap` 中默认拆分为 16 把锁的方法。拆分锁很大程度上能提高读写的速度，但需要注意的是在采用拆分锁后，全局性质的操作会变得比较复杂，例如 `ConcurrentHashMap` 中 `size` 操作。

按照这样的思路，在上述锁最小化的优化基础上再进行拆分锁的优化，将 `Handler` 类中的单把锁拆分为 10 把锁，`Handler` 类重构代码如下：

```
static class Handler{

    private static final Handler self=new Handler();

    private final Random random=new Random();

    private int lockCount=10;

    private Lock[] locks=new Lock[lockCount];

    private Handler(){
        for (int i = 0; i < lockCount; i++) {
            locks[i]=new ReentrantLock();
        }
    }

    public static Handler getInstance(){
        return self;
    }

    public void handle(int id){
        int mod=id%lockCount;
        try{
            // execute sth don't need lock
            try {
                Thread.sleep(random.nextInt(5));
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

        locks[mod].lock();
        // execute sth
        try {
            Thread.sleep(random.nextInt(5));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    finally{
        locks[mod].unlock();
    }
}

```

重新执行以上代码，执行后的统计信息如下：

```
Execute summary: Round( 10 ) ThreadCount Per Round( 200 ) Execute Time( 4409 ms)
```

从上面的统计信息来看，拆分锁后的性能有了大幅度的上升，但是拆分锁还得根据业务场景来决定。有些场景并不适合做锁的拆分，而且锁拆分得太多也会造成其他副作用，例如 CPU 的消耗明显增加等，因此锁拆分要在合理的业务场景以及 CPU 消耗下进行。

## 6. 去除读写操作的互斥锁

在修改时加锁，并复制对象进行修改，修改完毕后切换对象的引用，而读取时则不加锁，这种方式称为 CopyOnWrite。CopyOnWriteArrayList 是 CopyOnWrite 方法的典型实现，CopyOnWrite 的好处是可以明显提升读的性能，对于读多写少的应用非常适合，但由于写操作时每次都要复制一份对象，会造成更多的内存消耗。

## 未充分使用硬件资源

这种情况也是性能低的应用中经常出现的，主要体现在未充分使用 CPU 和内存。

### 未充分使用 CPU

对于 Java 应用而言，未充分使用 CPU 的原因主要是在能并行处理的场景中未使用足够的线程，一个典型的例子如下：

```

public class CPUNotUseEffectiveDemo {
    private static int executeTimes=10;
}

```

```

private static int taskCount=200;

public static void main(String[] args) throws Exception{
    if((args.length==1)|| (args.length==2))
        taskCount=Integer.parseInt(args[0]);
    if(args.length==2)
        executeTimes=Integer.parseInt(args[1]);
    Task task=new Task();
    for (int i = 0; i < taskCount; i++) {
        task.addTask(Integer.toString(i));
    }
    long beginTime=System.currentTimeMillis();
    for (int i = 0; i < executeTimes; i++) {
        System.out.println("Round: "+(i+1));
        Thread thread=new Thread(task);
        thread.start();
        thread.join();
    }
    long endTime=System.currentTimeMillis();
    System.out.println("Execute summary: Round( "+executeTimes+" ) TaskCount
Per Round( "+taskCount+" ) Execute Time( "+(endTime-beginTime)+" ms)");
}

static class Task implements Runnable{
    List<String> tasks=new ArrayList<String>();
    Random random=new Random();
    boolean exitFlag=false;
    public void addTask(String task){
        List<String> copyTasks=new ArrayList<String>(tasks);
        copyTasks.add(task);
        tasks=copyTasks;
    }
    public void run() {
        List<String> runTasks=tasks;
        List<String> removeTasks=new ArrayList<String>();
        for (String task : runTasks) {
            try {
                Thread.sleep(random.nextInt(10));
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
            removeTasks.add(task);
        }
        try {
            Thread.sleep(10);
        }
    }
}

```

```
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

执行以上代码，其统计信息输出为：

Execute summary: Round( 10 ) TaskCount Per Round( 200 ) Execute Time( 36139 ms)

对于有多个或多核的 CPU 而言，以上方式是没有充分使用 CPU 的。可重构为以下方式，更多使用 CPU，重构后 main 部分的代码如下：

```
public static void main(String[] args) throws Exception{
    if((args.length==1) || (args.length==2))
        taskCount=Integer.parseInt(args[0]);
    if(args.length==2)
        executeTimes=Integer.parseInt(args[1]);
    Task[] tasks=new Task[TASK_THREADCOUNT];
    for (int i = 0; i < TASK_THREADCOUNT; i++) {
        tasks[i]=new Task();
    }
    for (int i = 0; i < taskCount; i++) {
        int mod=i%TASK_THREADCOUNT;
        tasks[mod].addTask(Integer.toString(i));
    }
    long beginTime=System.currentTimeMillis();
    for (int i = 0; i < executeTimes; i++) {
        latch=new CountDownLatch(TASK_THREADCOUNT);
        System.out.println("Round: "+(i+1));
        for (int j = 0; j < tasks.length; j++) {
            Thread thread=new Thread(tasks[j]);
            thread.start();
        }
        latch.await();
    }
    long endTime=System.currentTimeMillis();
    System.out.println("Execute summary: Round( "+executeTimes+" ) TaskCount
Per Round( "+taskCount+" ) Execute Time( "+(endTime-beginTime)+" ms)");
}
```

Task run 部分的代码重构如下：

```

public void run() {
    List<String> runTasks=tasks;
    List<String> removeTasks=new ArrayList<String>();
    for (String task : runTasks) {
        try {
            Thread.sleep(random.nextInt(10));
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        removeTasks.add(task);
    }
    try {
        Thread.sleep(10);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
    latch.countDown();
}

```

在一个两核 CPU 的机器上执行以上代码，结果如下：

| Execute summary: Round( 10 ) TaskCount Per Round( 200 ) Execute Time( 18454 ms) |
|---|
|---|

从上可见，对于此类演变为多线程也无须加锁的场景而言，启动多个线程后的性能会远高于单线程，并且只要启动的线程数合理，也不会给 CPU 造成过大的负担。如从单线程演变为多线程要加锁，则要引入尽量减少锁竞争的方法，并进行性能测试以保证调优后的资源消耗以及性能满足要求。

此种类型的场景还有不少，例如单线程的计算，可以拆分为多线程来分别计算，最后将结果合并，这样方式也可很大程度提升系统的性能，JDK 7 中的 fork-join 框架可以给以上这类场景提供一个好的支撑方法<sup>21</sup>。

在 CPU 资源消耗可接受，且不会因为线程增加带来激烈锁竞争的场景下，应适当对处理过程进行分解，增加线程数从而能并行处理以提升系统的运行性能。但在重构为并行时，要注意控制内存消耗，而且通过重构为并行能提升的性能也是有限的。著名的 Amdahl 定律<sup>22</sup>中的简单计算公式为： $1/(F+(1-F)/N)$ ，其中 F 为必须串行化的执行在整个执行过程中所占的比率，N 为处理器个数，例如在整个执行过程中串行化的过程占 50%，那么最多只能提升 2 倍的性能。

21 <http://www.ibm.com/developerworks/cn/java/j-jtp11137.html>

22 [http://en.wikipedia.org/wiki/Amdahl's\\_law](http://en.wikipedia.org/wiki/Amdahl's_law)

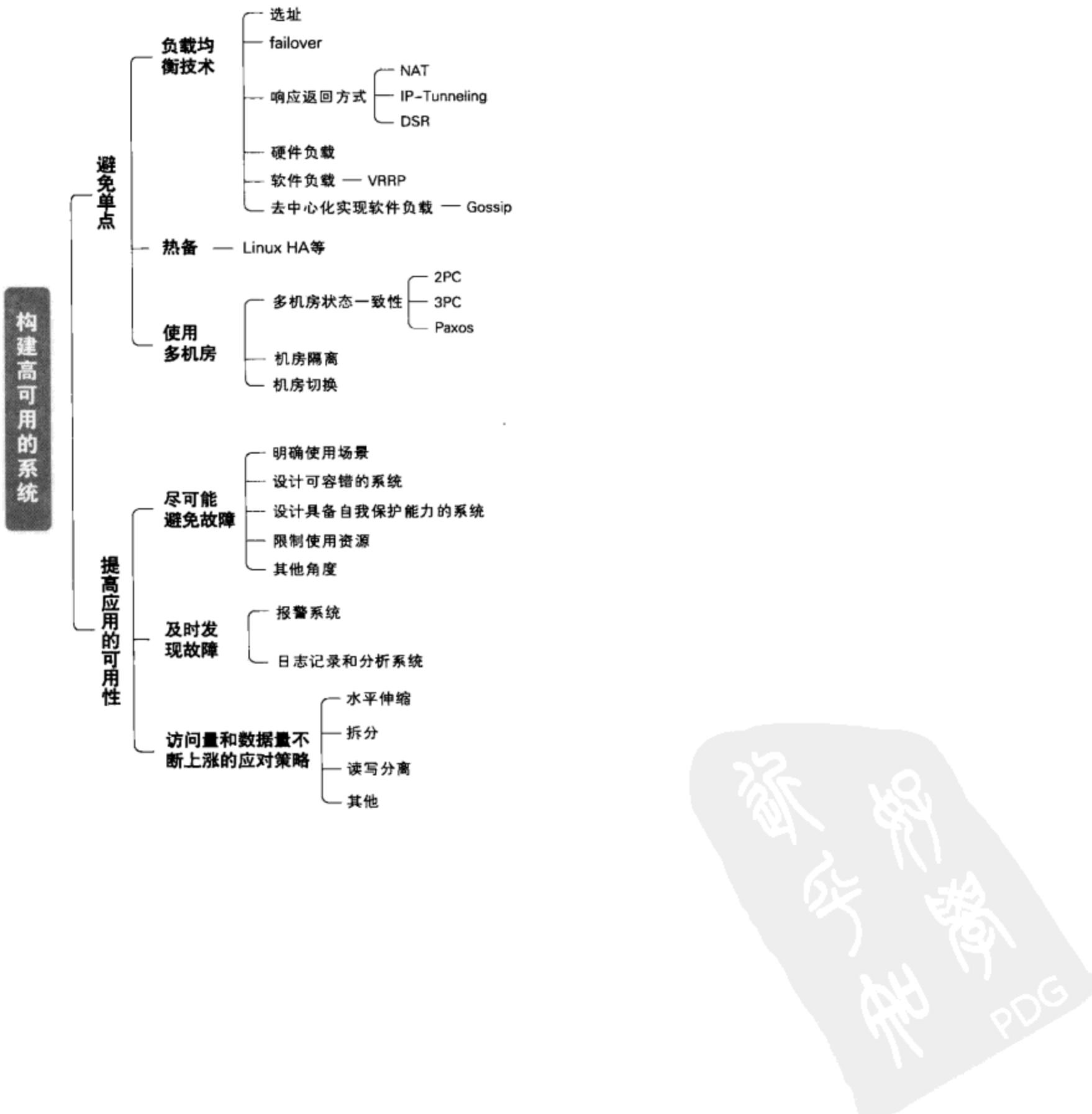
## 未充分使用内存

未充分使用内存的场景非常多，如数据的缓存、耗时资源的缓存（例如数据库连接的创建、网络连接的创建等）、页面片段的缓存等，这样的场景比较容易理解，毕竟内存的读取肯定远快于硬盘、网络的读取。但也要避免内存资源的过度使用，在内存资源消耗可接受、GC频率及系统结构（例如集群环境可能会带来缓存的同步等）可接受的情况下，应充分使用内存来缓存数据，提升系统的性能。

对于数据量大造成的性能不足，在第7章“构建可伸缩的系统”一章中提供了一些优化方案。好的调优策略应是收益比（调优后提升的效果/调优改动所付出的代价）最高的，通常来说功能简单的系统调优比较好做，否则有可能出现调优了当前功能影响到了其他功能，因此应尽量保持单机上应用功能的纯粹性，这是大型系统的基本架构原则。从纯粹的软件调优角度来讲，充分而不过分使用硬件资源，合理调整JVM以及合理使用JDK包是调优的三大有效原则，调优没有“银弹”，结合系统现状和多尝试不同的调优策略是找到合适的调优方法的唯一途径。



# 第6章 构建高可用的系统



对于互联网应用或企业中的大型应用而言，多数都要求尽可能地做到  $7 \times 24$  小时不间断地运行，要完全做到不间断地运行，基本上不太可能，因此经常会看到各大网站或大型应用在总结一年的状况时会有当年的可用性为 99.9% 这样的内容。表 6.1 给出了从三个 9 到五个 9 的不可用时间的计算方法：

表 6.1

| 可用性指标   | 计算方式                                     | 不可用时间（分钟） |
|---------|--|-----------|
| 99.9%   | $0.1\% \times 365 \times 24 \times 60$   | 525.6     |
| 99.99%  | $0.01\% \times 365 \times 24 \times 60$  | 52.56     |
| 99.999% | $0.001\% \times 365 \times 24 \times 60$ | 5.256     |

对于一个功能、用户、数据量不断增加的应用，要保持如此高的可用性并非易事。为了实现高可用，要避免系统中出现单点、保障应用自身的高可用、面对访问量及数据量不断增长带来的挑战。

## 6.1 避免系统中出现单点

单点现象是指系统部署在单台机器上，一旦这台机器出现问题（硬件损坏、网络不通等），系统就不可用。解决这种单点现象最常见的方法是将系统部署到多台机器上，每台机器都对外提供同样的功能，通常将这种系统环境称为集群。当系统从单机演变为集群时，要求系统能够支持水平伸缩，关于如何做到水平伸缩，在第 7 章“构建可伸缩的系统”一章中会详细阐述。除了水平伸缩外，要解决的问题还有以下两个：

- 如何均衡地访问到提供业务功能的机器；
- 如何保证当机器出现问题时，用户不会访问到这些机器。

### 6.1.1 负载均衡技术

为了做到这两点，通常采用负载均衡技术，负载均衡又分为硬件负载均衡和软件负载均衡。硬件负载均衡功能基本在硬件芯片上完成；而软件负载均衡功能由软件来完成。这两种负载均衡在解决以上两个问题时均采用类似的方法，系统结构如图 6.1 所示：

从图 6-1 可看出，无论是采用硬件负载还是软件负载均衡技术，都在系统环境中增加了负载均衡机器。负载均衡机器为避免自己成为单点，通常由两台机器构成，但只有一台处于服务状态，另一台则处于 `standby` 状态。一旦处于服务的那台机器出现问题，`standby` 这台会自动接管，具体的实现方式在后续的章节中会详细阐述。

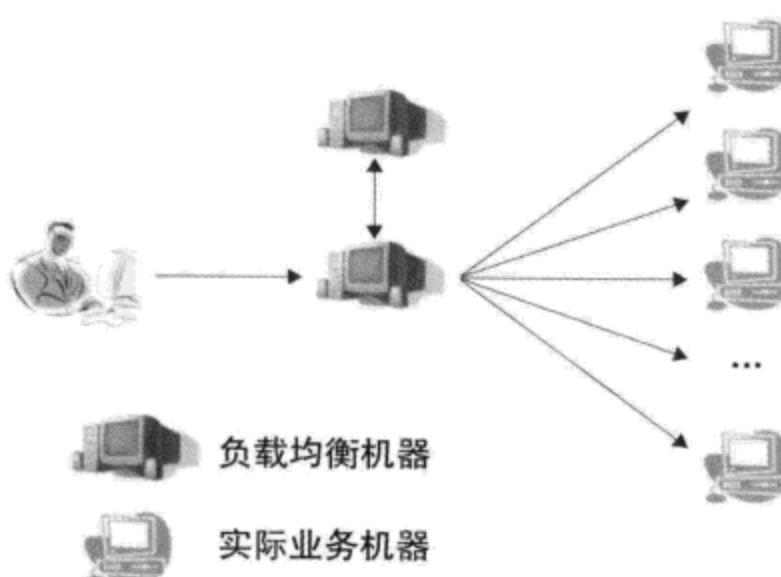


图 6.1 采用负载均衡后的系统结构

## 选择实际业务处理机器的方式

在增加了负载均衡机器后，用户请求的方式变为发送请求给负载均衡机器，负载均衡机器再将请求转发给实际的业务处理机器。转发时涉及的问题是如何选择实际的业务处理机器，这首先要求负载均衡机器知道实际业务处理机器的 IP 地址，通常采用的方法为在负载均衡机器上直接配置业务处理机器的 IP 地址。在选择时，主要有以下几种方式：

### 1. 随机 (Random) 选择

即从地址列表中随机选择一台，这种方法实现起来最为简单，性能也最高，在实际运行中，如业务处理机器在处理各种请求时所需消耗的资源相差不是特别大，那么采用随机方式能保持后端的机器的负载基本上是均衡的。

### 2. Hash 选择

即对应用层的请求信息做 hash，从而分派到相应的机器上，典型的应用场景是静态图片的加载。对请求的图片的 url 串做 hash，这样基本可以保证每次请求的是同一台机器，命中缓存，提升性能，这种方式多用于以上类型的应用场景。值得注意的是，由于它要读网络协议第七层（应用层）的信息，又要做 hash，因此要消耗更多的 CPU 资源并导致些许的性能下降。

### 3. (Round-Robin) 选择

即根据地址列表按顺序选择。由于要保持顺序，这种方法在选择时比随机多了一个同步操作，由于这个同步操作非常短，性能上损失很小，和随机方式一样，如业务处理机器处理各种请求时所需消耗的资源相差不大，那么采用顺序方式也是基本上可以保持后端实际业务处理机器负载时均衡的，这种方式目前的硬件负载和软件负载都支持，实际使用较多。

### 4. 按权重 (Weight) 选择

即根据每个地址的权重进行选择，权重有静态权重和动态权重两种。

静态权重是指配置好集群中各机器的权重，在运行时会根据这个配置选址。当集群中机器档次不同

时，可以给配置高的机器分配更高的权重，更好地利用机器性能。对于权重相同的多台机器，通常又支持按随机或顺序的方式选择，静态权重适用于仅按机器配置来分配机器承担的请求比例的业务场景。

动态权重是指负载均衡设备或软件根据业务处理机器的 load、连接数等动态权重来分配任务，以更加合理地发挥业务处理机器的性能，动态权重适用于根据运行状态分配承担请求比例的业务场景。

### 5. 按负载（Load）选择

即根据实际业务处理机器的负载来选择。选择负载相对较低的机器来进行处理，尽可能地保证所有业务处理机器的负载是均衡的。此方法适合于业务处理机器在处理多种请求时所需资源相差较大的情况。它可以避免将消耗资源多的请求都分配到同一台机器上，出现虽请求分配均衡，但负载严重不均衡的现象，这就要求负载均衡机器每隔一段时间就向实际的业务处理机器搜集其负载的状况，这种方式会给负载均衡机器增加一些负担，并且一旦搜集负载状况过程中出现较大延时或搜集不到时，很可能造成更严重的负载不均衡，因此在实际的应用中使用较少。

### 6. 按连接（Connection）选择

即根据实际业务处理机器连接数的多少进行选择，选择连接数相对较少的机器来处理业务，此方法适用于连接数不均衡的场景中。实现这个方法只是增加了连接数对可用地址列表做排序的负担，其他方面不会有太多影响。如使用这种方式，要特别注意类似下面的场景：假设后端业务处理的机器是 10 台，现在每台上的连接数大概是 1000，这时若重启 1 台，采用按连接这种负载均衡算法，且瞬间请求量大，那么 1000 个请求就会同时发到新启动的这台机器上，这很有可能造成这台机器宕掉，因此这种方式在实际应用中使用较少。

除以上的选址方式外，通常还会支持按 cookie 信息绑定访问相同机器的方式。这样的好处是只用把相关信息缓存在各自机器上，避免须要引入分布式缓存等复杂技术，但会带来的问题是旦机器出现故障，在该机器上登录过的用户就要重新登录了。

尽管硬件负载设备或软件负载方案提供了以上多种选址方式尽可能地保证后端服务器的负载均衡，但在实际的场景中，还会出现如下的典型问题。

对于一个 Web 站点而言，通常用户请求处理的方式为：用户发送请求到负载均衡机器，负载均衡机器再将请求分派到后端的 Web Server，通常这个 Web Server 会配置可处理的请求数及当请求数最大时等待队列的大小。这样 Web Server 接到请求后，如可以处理，则分配线程来处理，如已到达最大请求数，则放入队列中等待。这种方式会出现的现象是如果前面的请求处理缓慢，排在队列中的请求就要等待很久，但此时负载均衡设备仍然是根据顺序选址的话，就会造成有些请求被丢弃或超时，Twitter 在此时改为采用 unicorn<sup>1</sup>来解决。

Twitter 的工程师在分享 unicorn 带来的作用时<sup>2</sup>，首先对原有方式做了个比喻，原来的方式就像是

1 <http://unicorn.bogomips.org/>

2 <http://engineering.twitter.com/2010/03/unicorn-power.html>

超市多个收银台队列，一旦队列中有一个顾客慢了，就会影响到后面的所有顾客，而这个时候其他的收银台是可以协助处理的。新的处理方式则修改为所有的顾客到同一个收银台排队，当某个收银员处理完毕时，则按信号灯通知顾客，这样就避免了由于某些顾客或收银员慢而导致其他顾客等待。

在采用 unicorn 后，当用户访问 `twitter.com` 时，都在同一地方排队等候请求的执行，当有 Server 的处理队列空闲时，请求就分派给该 Server 执行，如此一来，Twitter 的请求延时被丢弃的情况减少了大概 30%。

从上面的例子来看，在此情况下如采用 RR 或随机等方式，会由于 Server 对每个请求的处理速度不同而造成请求严重延时或请求被丢弃的现象，而采用类似 unicorn 的策略则可降低此类现象发生的几率。在 lighttpd 上也有一个支持这种方式的均衡算法：Shortest Queue First<sup>3</sup>。

为了保证访问时跳过出问题的机器，通常采用的方法是负载均衡机器定时和实际的业务处理机器进行心跳（ping、端口检测或是 url 侦测），发现心跳失败的机器即将其从可用地址列表中拿掉，在心跳成功后再重新加入可用地址列表。

## 响应返回方式

业务处理机器处理完毕后，要将响应返回给用户，通常负载均衡方案都支持以下两种返回方式：

### 1. 响应通过负载均衡机器返回

响应通过负载均衡机器返回是最常见的方式，对于系统的部署环境也没什么要求，但这种方式的问题在于请求包和响应包都要经过负载均衡机器。随着请求量上涨，负载均衡机器所承受的压力会迅速上升，尤其是对于请求包小、响应包大的 Web 应用而言，就更是如此了。

这种方式基于 NAT 实现，当请求从客户端发送至负载均衡机器时，负载均衡机器首先选择一台实际的业务处理机器，然后将请求报文的目标地址和端口改为实际业务处理机器的 IP 地址和端口，并将报文发送出去。当响应回到负载均衡机器上时，将报文中的源地址和端口修改为负载均衡机器的 VIP 地址和端口，过程如图 6.2 所示：

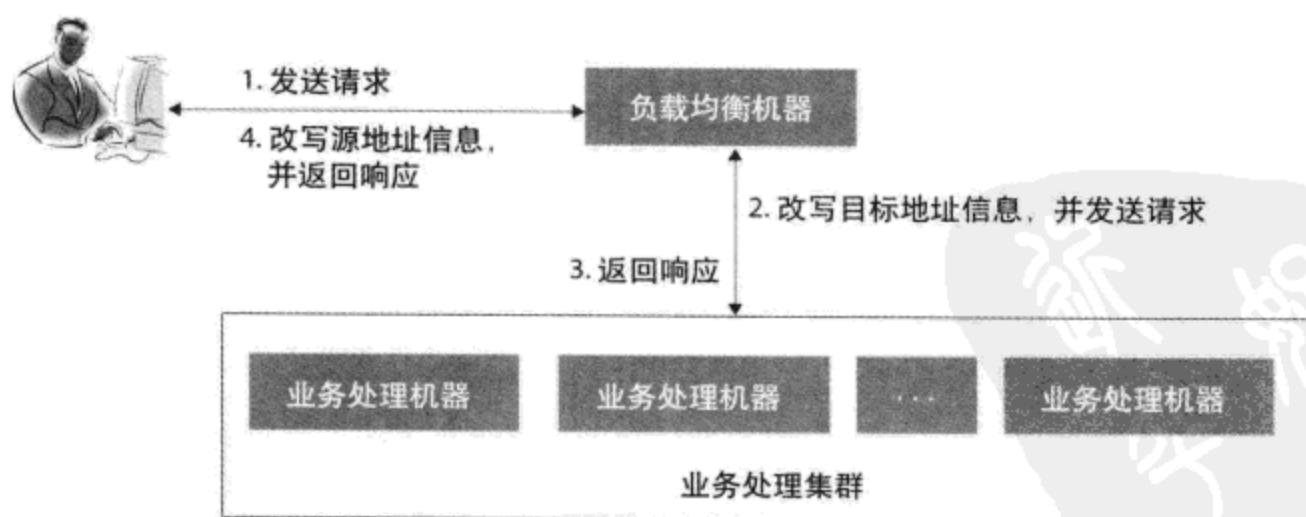


图 6.2 NAT 方式时的响应返回

<sup>3</sup> <http://blog.lighttpd.net/articles/2006/11/14/mod-proxy-core-and-sqf>

## 2. 响应直接返回至请求发起方

响应直接返回至请求发起方可将请求包和响应包分开处理，以分散负载均衡机器的压力，使负载均衡机器可支撑更大的请求量，对于 Web 应用而言效果会更加明显。要达到响应直接返回的效果，须要采用 IP Tunneling 或 DR（Direct Routing，硬件负载设备中又简写为 DSR：Direct Service Routing）方式，IP Tunneling 或 DR 方式对负载均衡机器和实际业务处理机器的系统环境都有要求。

当采用 IP Tunneling 方式时，请求从客户端发送至负载均衡机器，负载均衡机器首先选择一台实际的业务处理机器，然后将请求的 IP 报文基于 IP 封装技术封装成另外一个 IP 报文，在做完以上处理后将报文发送出去，实际的业务处理机器收到报文后，先将报文解开获得目标地址为 VIP 的报文，处理完毕请求后，处理机器发现此 VIP 地址配置在本地的 IP 隧道设备上，则根据路由表将响应报文直接返回至请求方。IP Tunneling 方式要求负载均衡机器和实际的业务处理机器的 os 都支持 IP Tunneling，并将 VIP 地址同时配置在实际业务处理机器的 IP 隧道设备上，过程如图 6.3 所示：

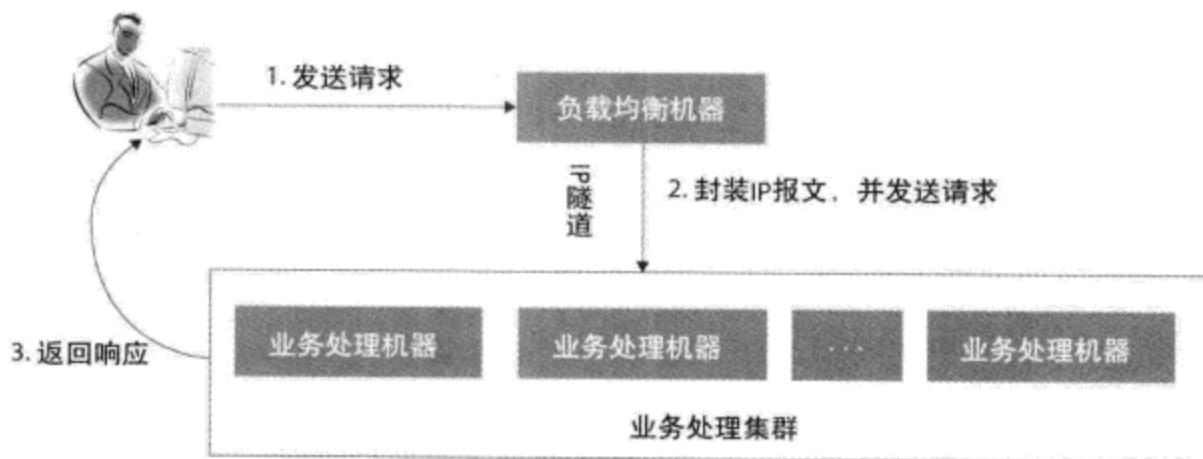


图 6.3 IP Tunneling 方式时的响应返回

当采用 Direct Routing 方式时，请求从客户端发送至负载均衡机器，负载均衡机器首先选择一台实际的业务处理机器，然后将请求数据帧的 MAC 地址修改为此业务处理机器的 MAC 地址，并发送出去，实际的业务处理机器收到请求后，获取 IP 报文，当发现 IP 报文中的目标地址 VIP 配置在本地的网络设备上时，根据路由表将响应报文直接返回给用户。Direct Routing 方式要求负载均衡机器和实际的业务处理机器在同一个物理网段中，并且不响应 ARP，过程如图 6.4 所示：

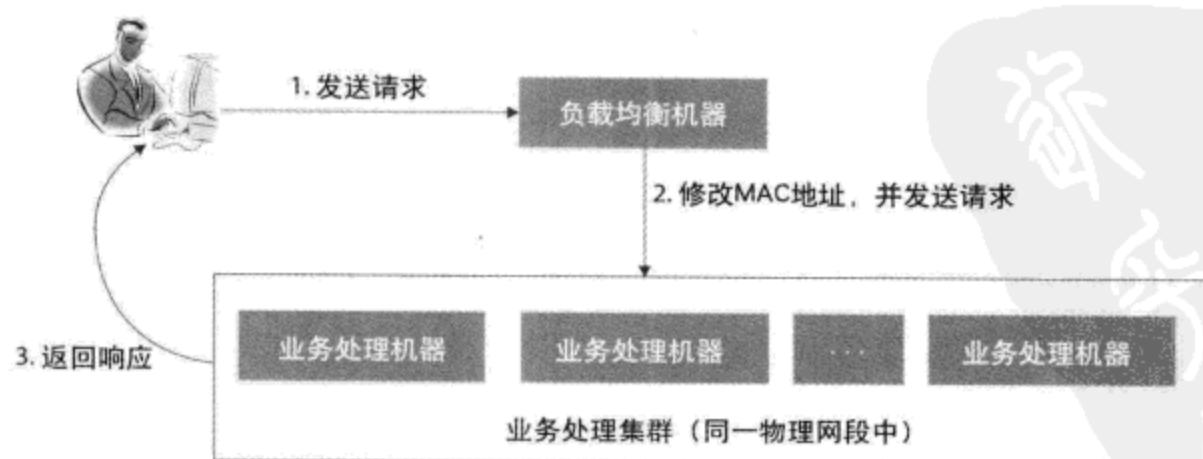


图 6.4 DSR 方式时的响应返回

根据上面的描述可以看出，IP Tunneling 方式对系统环境的要求并不高，目前大部分的 OS 都支持 IP Tunneling，Direct Routing 方式对系统环境的要求则比较高，因此 IP Tunneling 方式更适合实现将响应直接返回给请求发起方，从而大幅度提升负载均衡机器所能支撑的请求量。

除了以上一些通用的实现外，硬件负载和软件负载实现时还有些细节上的不同。

## 硬件负载设备

硬件负载设备中最为知名的有 F5 和 Netscalar，它们在实现负载均衡机器自动接管上通常采用的是两台负载均衡机器之间用一根单独的心跳线连接的方式。服务的机器和 standby 的机器通过心跳线保持心跳，一旦服务的机器出问题，standby 机器自动接管。这两种硬件负载设备获得实际业务处理机器 IP 地址的方法均为通过负载设备管理界面或命令行手工增减。

硬件负载设备大部分操作都在硬件芯片上直接做（例如 hash 计算），所能支撑的量非常高，运行也非常稳定，厂家又提供维护，因此在经费充裕的情况下采用硬件负载设备是不错的选择。对于采用硬件负载设备的应用而言，主要应注意以下两点：

### 1. 负载设备的流量

一旦负载设备的流量达到上限，就会出现大量访问出错等异常问题，因此要注意对负载设备流量状况的监测，一旦接近上限，就必须立即扩充带宽或增加负载设备，但增加硬件负载设备时，通常要修改应用，比较麻烦。

### 2. 长连接

长连接方式的应用时采用硬件负载设备很容易出现问题，当要连接的实际业务服务器有几台重启同时又有大量请求进来时，所有的请求都转发到健康的服务器上，并建立起长连接，几台重启的机器重启后只有很少请求或完全没有请求，这样就造成了严重的负载不均衡的现象。这种现象在客户端以连接池的方式建立多个长连接的情况下会非常明显，这时可采用手工断开负载设备上某 VIP 所有连接的方法来避免，强迫所有客户端重建连接来达到均衡，或者每个长连接上只允许执行一定次数的请求，当达到阀值时即关闭此连接，在一定时间后负载也能够逐渐均衡。

## 软件负载方案

软件负载方案中最常用的为 LVS（Linux Virtual Server）<sup>4</sup>，多数情况下采取 LVS+Keepalived 来避免负载均衡机器的单点，实现负载均衡机器的自动接管，具体实现如下。

Keepalived<sup>5</sup>基于 VRRP（Virtual Router Redundancy Protocol）协议<sup>6</sup>实现，在 VRRP 协议中，由一个 Master 的 VRRP 路由器和多个 Backup 的 VRRP 路由器构成 VRRP 虚拟路由器，但 Master 并不是永远不变的，Master 的 VRRP 路由器会每隔一段时间发送广播包。当 Backup VRRP 路由器在连续三个周期

4 <http://www.linux-vs.org>

5 <http://www.sanotes.net/wp-content/uploads/2009/04/keepalived%20the%20definitive%20guide.pdf>

6 [http://en.wikipedia.org/wiki/Virtual\\_Router\\_Redundancy\\_Protocol](http://en.wikipedia.org/wiki/Virtual_Router_Redundancy_Protocol)

内都收不到广播包时，即认为 Master VRRP 路由器出现问题，或收到优先级为 0 的广播包后，所有 Backup VRRP 路由器都发送 VRRP 广播信息，声称自己是 Master，并将虚拟 IP 增加到当前机器上，从而保持对外提供的 IP 地址及 MAC 地址不变。Backup VRRP 路由器收到 VRRP 广播信息后，首先比较优先级，如优先级比收到的 VRRP 广播信息中的优先级低，则重新将状态置为 BACKUP。如优先级相等，则比较 IP 地址，IP 值小的则重新将状态恢复为 BACKUP，整个切换过程对于请求端而言是透明的。但由于 VRRP 方式依靠广播信息来确认是否健康，如网络上出现异常，有可能会出现多个 Master 的现象，这个时候会出现一些问题，因此当使用 VRRP 方式时要特别监测是否出现此类现象，一旦出现就要迅速人工介入处理。

除了采用 Keepalived 方式实现自动接管外，也可采用类似硬件负载设备的方式来实现，即采用心跳线+高可用软件来实现。在 linux 目前使用范畴最广的高可用软件为 heartbeat<sup>7</sup>，默认情况下 heartbeat 通过 UDP 方式来监测。

除 LVS 外，软件负载方案中还有像 HAProxy 这样的佼佼者，在考察采用哪种软件负载方案时，则要从应用场景、系统环境等多方面考虑。

在系统从单机演变为集群后，可用性确实会得到一定提升，但随着系统功能的不断丰富，会出现多个系统访问同一系统提供的功能的情况，在这种情况下有可能会出现其中某个系统的访问导致其他系统故障。例如一个博客应用，其中一个系统对外提供了获取博客数据和发布博客的功能，访问这两项功能的系统可能会有多个，假设其中有一个是提供开放 API 的系统，有些时候可能会出现这个系统访问获取博客数据和发布博客的量太大，导致获取博客数据和发布博客的功能出现问题，进而影响所有使用这两个功能的系统。对于以上这种故障传播的现象，通常会采取根据应用性质的不同做隔离的方案，对于采用硬件负载设备和 LVS 类型的软件负载方案而言，通常采取配置多个不同的 VIP 的方法，各个系统通过域名访问，通过 dns 等方法使域名根据不同的系统解析为不同的 VIP，从而实现根据应用性质不同来隔离，避免故障传播。

除此以外，还可能出现的问题是提供获取博客数据和发布博客的功能两者消耗的资源比重不同，有时会出现由于写消耗资源过多导致读功能出现故障，因此又希望能够将读、写功能分开，此时可选的方案有：

- 将获取博客数据和发布博客的系统拆分为两个系统，并分开部署，这样会导致开发比较麻烦；
- 为获取博客数据功能和发布博客功能配置两个 VIP，当访问者在访问获取博客数据时访问其中一个 VIP，当访问者发布博客时则访问另一个 VIP，这种方式的问题在于访问者要明确知道功能的划分，也比较麻烦。当访问者不透明时可以在客户端配置路由策略，此路由策略基于访问的功能来划分到不同的 VIP，这样的优点是可以让访问者仍然透明地访问；
- 基于硬件负载设备提供的应用层路由，当访问者要获取博客数据时，则路由到其中的一组机器，当访问者发布博客时，则路由到另一组机器。这种方式的优点是对访问这两个功能的系统而言是透明的。

<sup>7</sup> <http://www.linux-ha.org>

的，并且可以根据压力状况来调整机器的分配，如请求的方式为 Http，在硬件负载设备很容易配置出上述结果，但如果不是 Http，而是自定义的通信协议和方式，要在硬件负载设备上配置出来就极为复杂了。另外由于每次请求都要进行第七层信息的解析，并做规则匹配，会造成硬件负载设备性能下降，LVS 不支持第七层路由，因此无法实现此方案。

从上面的例子可看出，在构建集群后，还须充分从业务角度考虑如何合理地进行应用的隔离，以更好地提升可用性，此时通常要借助第七层路由功能。

## 去中心化实现负载均衡

无论是硬件负载，还是软件负载，都可以简单、透明地构建集群，避免单点故障，它们的共同点是在请求的过程中引入了一个中间点，这会对性能、可用性带来一些影响。并且由于硬件负载设备或软件负载在水平伸缩时，要修改调用方应用，比较麻烦，于是业界出现了基于 Gossip<sup>8</sup>实现无中间点的软件负载。

Gossip 是个去中心化传播事件的模型，在 Gossip 论文中用了一个现实生活中谣言传播的例子来讲述 Gossip。在现实生活中，谣言通常是 A 传播给 B，然后 B 传播给 C，同时 A 又传播给 D，之后就变成 A、B、C、D 四个人传播，于是很快这个谣言就传播开了，在这样的传播方式下，即使后面 A、B 不传播了，其他人也会慢慢知道这个谣言，只是可能速度会慢点。传播的时间复杂度为  $\log(n)$ ，这种方式的好处在于可以不依赖任何一个点，无中心化，Gossip 模型采用类似 TCP/IP 三次握手的机制来进行一次请求，其模型如图 6.5 所示：

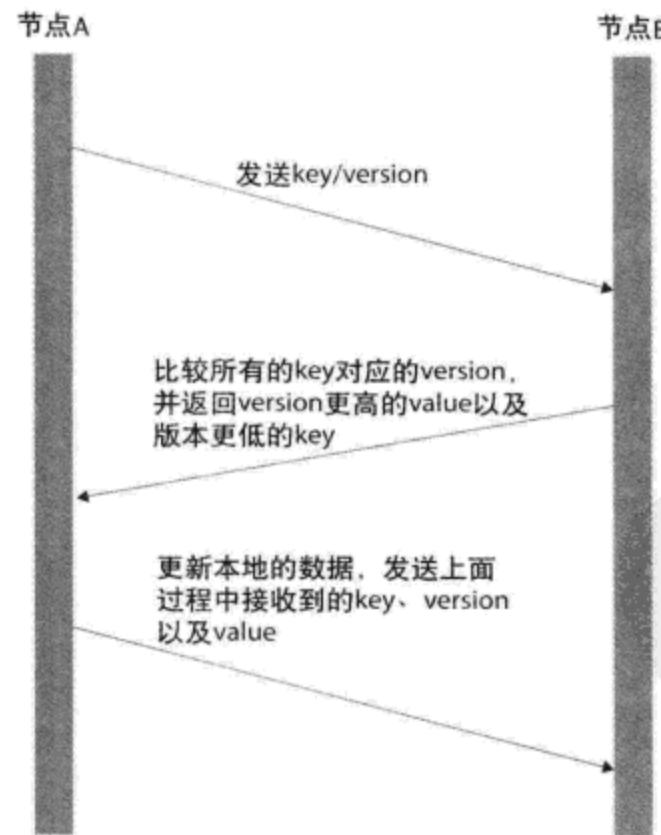


图 6.5 Gossip 模型

<sup>8</sup> <http://www.cs.cornell.edu/home/rvr/papers/flowgossip.pdf>

每个节点上都存放了一个 key、value、version 构成的列表，每隔一定时间会从节点列表中挑选一个在线的节点进行上述的三次握手过程，同时还会挑选一个不在线的节点也尝试进行上述的三次握手过程，节点只能修改自己的 key、value 和 version，在这样的模型下，当数据变化时，通过多台机器的散播可以较快地完成数据的同步。

从上面的模型可看出，基于 Gossip 模型在交互过程中并没有一个中心点，数据在同步传播过程中即使一个节点出问题了，其他节点也会继续将数据同步。Facebook 开源的 Cassandra 是一个基于 Gossip 实现的无中心的 NoSQL 数据库，Cassandra 在实现 Gossip 模型时采用的为基于 UDP/IP 实现，节点的发现则通过 seed 的机制来实现，即每个 Cassandra 节点在启动时首先向 seed 机器请求，seed 机器可以是多台，在连上 seed 机器后即进行上述的三次握手，获取到其他的节点机器列表，然后继续基于 Gossip 模型与其他节点机器进行数据同步。

在去掉中间点后，请求/响应可直接进行，这对于性能、可用性都会带来提升。同时，无中间点后，还可以将路由策略放在访问端，从而在不影响性能的情况下实现复杂的路由策略。

以上所讲的均为实现本地机器的负载均衡访问，当系统规模扩大到一定程度后，会要在不同的地域建设机房，在不同地域的集群实现负载均衡的技术通常称为全局负载均衡（Global Load Balance）。各负载设备的硬件厂商通常提供了一种称为 GSLB（Global Server Load Balance）的设备来实现全局负载均衡，在采用 GSLB 的情况下，可避免由于一个地方的机房出现故障导致用户不可访问的现象，同时 GSLB 还可根据地理位置来选择最近的机器，这无疑也在一定程度上提高了性能。

集群是用来避免单点常用的方案，但由于集群要求系统本身支持水平伸缩，这对于系统而言成本偏高，在这些系统中，可采用热备方式来避免单点故障。

## 6.1.2 热备

热备通常对程序的要求不高，热备的情况下真正对外服务的机器只有一台，其他机器处于 standby 状态。standby 机器通过心跳机制检查对外服务机器的健康状况，当出现问题时，其中一台 standby 机器即进行接管，机器间的状态同步至其他 standby 机器或写入一个集中存储设备，例如上述章节中 LVS+Keepalived 实现自动接管的方式。

对于大型应用而言，除了单机故障外，还须考虑整个机房出现不可用的情况。如所有的应用都部署在单个机房，也可以认为是单点现象，一旦发生机房断电或机房出现不可抗力的灾难性事故时，整个系统的可用性就完全无法保障了，对于此类现象，通常采用多个机房的方法来避免，一方面可以做到其中一个机房出现问题时对整个系统不会产生太大的影响，另一方面也可以分流，提升性能。

### 使用多机房

多机房对技术上的要求较高，难点主要有以下几个：

## 1. 跨机房的状态同步

跨机房后就会涉及到应用在多个机房的状态同步，包括有持久数据的同步、内存数据的同步。当机房在不同地方时，将会面临的最大问题是机房间网络的延时和各种异常状况，这对于实时性要求高的应用而言是非常大的挑战，要同步的主要有数据库数据、文件及内存状态。

数据库数据的同步通常采用单 master、多 slave 或多 master 方案，单 master 方案只有一个写入点，其主要解决的是 master 同步到 slave 的问题，通常采取的是数据库自带的同步方案，例如 oracle standby 方案或 mysql replication 方案。无论是采取哪种方案，都要注意同步带来的延时，尤其是异地机房，如面临的是异构的 master/slave 数据库，那就比较复杂了，通常需要自行基于消息中间件方式来实现；多 master 方案有多个写入点，相对单 master 方案就复杂多了，通常采取的两阶段提交、三阶段提交或基于 Paxos 的方式来保持多 master 数据的一致性。

## 2. 两阶段提交（2PC）<sup>9</sup>保持一致性

在采用两阶段提交保证多 master 数据的一致性时，步骤为：

- 1) 开启事务；
- 2) 通知每个 master 执行某操作；
- 3) 所有 master 在接到请求后，锁定执行此操作需要的资源，例如假设是个扣款动作，那么先冻结相应的款项，冻结完毕后返回；
- 4) 在收到所有 master 的反馈后，如均为可执行此操作，则继续之后的步骤，如有一个 master 反馈不能执行或一段时间内无反馈，则通知所有 master 回滚操作；
- 5) 通知所有 master 完成操作。

两阶段提交方式相对而言比较易于实现，但问题在于所有的 master 都要冻结资源，而且一旦有一个 master 出现问题就要全部回滚。

## 3. 三阶段提交（3PC）<sup>10</sup>保持一致性

为了避免在通知所有 master 提交时，其中一个 master crash 不一致时，就出现了三阶段提交的方式。三阶段提交在两阶段提交的基础上增加了 preCommit 的过程，当所有 master 收到 preCommit 后，并不执行动作，直到收到 commit 或超过一定时间后才完成操作。

在实现两阶段或三阶段提交时，为了避免通知所有 master 时出现问题，通常会借助消息中间件或让任意的一个 master 能够接管成为通知者。

<sup>9</sup> [http://en.wikipedia.org/wiki/Two-phase\\_commit\\_protocol](http://en.wikipedia.org/wiki/Two-phase_commit_protocol)

<sup>10</sup> [http://en.wikipedia.org/wiki/Three-phase\\_commit\\_protocol](http://en.wikipedia.org/wiki/Three-phase_commit_protocol)

#### 4. 基于 Paxos<sup>11</sup>保持一致性

Paxos 最大的改变在于不要求所有 master 都反馈成功，只须有大多数反馈成功就执行了，更多具体的细节请参考相关文献。

除了以上三种方式外，还可选择 PNUTS<sup>12</sup>方式来实现，对多机房数据一致方案感兴趣的读者可以进一步参考 google 工程师介绍 GAE 后端数据服务的 PPT<sup>13</sup>。在实现一致性方案时，通常都遵循 CAP 理论<sup>14</sup>，选择所关注的两个点，例如 Paxos 关注的是 CP。

文件的同步和内存数据的同步采取的方案和数据库数据同步的方案基本相同。

总的来说，由于采用多机房后带来的网络延时问题，技术上会出现不少的挑战，不过对于要求高可用的应用，采用多机房还是很有必要的。

##### 1. 机房隔离

机房隔离是建设多机房时必须做到的，意思是核心的应用在每个机房中都必须存在，以便即使只有当前机房健康时，也能正常地对外提供服务。其难点在于如何找出系统的应用，对于没有清晰依赖关系的系统而言，会非常痛苦，同时机房隔离一定程度上是有冗余的，因此成本方面会高很多。

##### 2. 机房切换

无论是采用单 master 还是双 master 实现跨机房状态一致的方案，在某个 master 出问题时，都有很明显的切换问题，例如当缓存系统在两个机房都存在时，其中一个机房出问题了，则要求能够将这个机房中使用的缓存系统切换到另外的机房，这个时候如何做无缝的切换就很重要了。避免所有的应用都要修改，对于整机房性质的切换而言，最复杂的问题则在于机房切换后流量的增加，通常在少一个机房的情况下，不太可能支撑全部的流量，因此在做整机房切换时尤其要考虑如何合理地限制流量。

以上介绍了为了保障可用性，避免单点故障中最典型的单机、单机房故障的常用应对策略，但除了单点故障外，另外一个最明显的也通常是造成系统不可用比例最高的故障就是应用本身的故障。下面就来看看如何保障应用自身的高可用。

## 6.2 提高应用自身的可用性

应用通常要满足多种多样的功能需求，尤其是互联网应用在不断添加功能的同时还必须保持高速发展，应用中还难免出现 bug。在这里就来介绍一下保障应用自身高可用的常用方法。

为了保障应用自身的高可用，首先要做的是尽可能地避免应用出故障。但要做到完全不出故障不太可能，互联网是一个无限放大故障的地方，任何一个看似小的、甚至以为绝对不可能发生的问题，

<sup>11</sup> [http://labs.google.com/papers/paxos\\_made\\_live.html](http://labs.google.com/papers/paxos_made_live.html)

<sup>12</sup> <http://research.yahoo.com/files/pnmts.pdf>

<sup>13</sup> [http://snarfed.org/space/transactions\\_across\\_datacenters\\_io.html](http://snarfed.org/space/transactions_across_datacenters_io.html)

<sup>14</sup> <http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>

都会在互联网上发生，导致故障。因此要及时地发现故障，并在发现故障后能及时地处理，从而把故障时间尽可能地缩短，下面分别来看看这三个步骤中要做的事情及通常采用的方法。

### 6.2.1 尽可能地避免故障

要做到编写的代码尽可能地避免故障，一方面要深入理解所使用的 Java 类库和框架，这至关重要，可以帮助你清楚地了解程序在运行时的状况；另一方面则是经验，经验主要靠亲身经历或学习来获得，亲身经历的体会最深，摔过跤的人多数不会在同一个地方再摔，学习其他人的经验时一定要了解清楚故障产生的原因，否则很难变成自己的经验。以下是自己根据一些常见故障的点形成的可用性设计原则，这些原则如下。

#### 明确使用场景

这点说起来简单，但在某些情况下非常容易出问题，一个典型情况是产品发展得不错时，其承担的期望通常会超越最初设定的使用场景，在这种情况下，通常会在原基础上做很多的复用，来支持新的略微不同的使用场景。但就是在这样的方式下，会导致设计时过多地考虑复用，而没有去仔细分析使用场景的不同，很有可能给新的使用场景加上了一些不必要的实现，最终出现故障；另一种就是在设计时没有从使用场景去考虑，更多的是从纯技术角度去考虑，设计了很多不必要的功能，例如没必要的系统扩展、系统功能等，将系统复杂化。这也非常容易造成系统故障。因此在设计时应贴近使用场景，保持系统的简单，对于复杂的系统功能，则应分解为多个阶段来完成，保持每个阶段的简单。

#### 设计可容错的系统

要使系统具备高度的容错能力，主要从两方面进行掌控，一是 Fail Fast 原则，二是保证接口和对象设计的严谨。

Fail Fast 原则是当主流程的任何一步出现问题时，都应快速地结束整个流程，而不是等到后续才来处理，举例如下：

A 系统在启动阶段要从本地加载一些数据放入缓存，如加载失败，会造成请求处理过程失败，对于这样的情况，最佳的方式是如加载数据失败，则直接 JVM 退出，避免启动后不可用。

从上例可见，fail fast 的作用是避免系统在出错的情况下做一些无谓的处理，造成不可用的问题，这一点从开源的很多框架的 API 实现代码中也可以看出，通常代码中都会判断传入的参数是否符合要求，不符合则直接抛错。

接口和对象设计的严谨最容易被忽略，通常我们在对外提供接口或对象时，总是会要求使用者应该怎么去用，通过 JavaDoc 或文档方式去声明，例如需要使用者保证单例等。接口的实现内部则假定外部一定会按照要求去使用，不采取任何的保护措施，在这种情况下是非常容易产生问题的，一个简单的例子如下：

```
public class AImpl implements A{
    public void register(String key){
        // 执行一系列复杂动作
    }
}
```

对于实现了 A 接口的 AImpl，希望使用者能够保证对同样的 key 只调用一次 register，但这一点并没有从接口的使用上做出限制，而且也没有在内部的实现上做相应的处理。那么这种情况下就有可能出现由于使用者误用，多次注册引发问题，这种类似的现象在实际的系统中有很多，可以采用下列方法避免用户对以上接口的误用。

### 1. 内部处理，避免误用

内部处理，避免误用是指在代码内部保证不论外部如何使用，内部的行为都以期望的方式运行，例如将上面示例的代码改造如下：

```
public class AImpl implements A{
    private static List<String> registered=new ArrayList<String>();
    public void register(String key){
        if(registered.contains(key)){
            return;
        }
        // 执行一系列复杂动作
        registered.add(key);
    }
}
```

修改成上面的代码后，即使用户误用，重复调用此方法，也不会造成问题。当然，如果是并发场景，还得做加锁动作。

除了上面这种状况外，还有一个常见的例子，DomainObject 是对外提供使用的一个对象：

```
DomainObject object=new DomainObject();
object.setType(DomainObject.TYPEA);
// DomainObjectManager 将根据 object 的类型进行不同的逻辑处理
DomainObjectManager.handle(object);
```

这个对象设计的问题在于依赖外部通过设置 Type 从而获得正确的执行流程，这也要求使用者非常清楚，但往往很难做到，因此对于以上这种对象的设计，可以采用以下两种方法来避免误用。

### 2. 强制用户设置 Type

不提供默认 Type，可以采用构造器中强制指定 Type，或在运行时做检查，运行时做检查会显得更不友好，因此最佳方式仍然是构造器中需要传入 Type，或采用一种能在编译时进行检查的方式。

### 3. 强制使用明确的对象

对于上面这种情况，可以将不同类型的对象不通过 type 属性来决定，而是直接采用对象方式，例如修改为：DomainObject object=new ADomainObject(); 这样可以强制用户在创建对象时就明白其中的差别。

从以上两个例子来看，主要是设计者希望使用者能够按照一定的方式去使用所提供的接口和对象，但并没有从接口和对象上做强制的措施，也没有在内部做相应的误用保护措施。在这种情况下，很有可能会因使用者的误用而产生问题，因此对于提供给外部使用的接口和对象一定要谨慎，尽可能地保证使用者按照希望使用的方式去使用，关于这点在《Effective Java（第 2 版）》一书中也提到了几条设计原则，感兴趣的读者可以去看看书中的第 1 条、第 3 条、第 15 条等。

## 设计具备自我保护能力的系统

通常系统对第三方都会有依赖，例如对数据库的依赖，对存储设备的依赖，对其他系统提供的功能的依赖。对于这些依赖，在设计时都要做充分的考虑，避免依赖的第三方出现问题时连锁反应，导致宕机。最佳的设计方法是对所有第三方依赖的地方都持怀疑态度，对这些地方都相应地做一些保护措施，尽可能地做到当第三方出现问题时，对应用本身不产生太大的影响，例如只是某功能暂时不可用。

举例子如下：

A 系统的数据要从数据库中获取，当数据库反应慢时，会造成 A 系统的请求慢，在请求多的情况下就会造成很多线程都处于等待数据库连接上，更糟糕的是有可能造成大面积的请求等待，最终导致 A 系统宕掉的现象。

对于上面这种现象，通常可选择的方法是只允许有一定数量请求等在数据库连接上，如果超过了这个数量，那就不再等待，而是直接抛出异常，由应用自行控制。

在 Flickr 网站的设计中，他们为了避免某个系统由于全文检索出问题后不可用，增加了一个缓存策略，即当全文检索不可用时，会临时地从缓存中获取，这也是自我保护的一种常用设计手段。

对于这类第三方依赖变慢导致应用本身宕掉的原因有很多种，如果能在设计阶段就加以考虑，那么对于提高系统的可用性会有很大帮助，但具体的解决方法则要依据实际状况来决定。

## 限制使用资源

对于资源使用的限制是设计高可用性系统中非常重要的一点，也是很容易被遗忘的一点，资源毕竟是有限的，用得过多了，自然就会导致应用宕机，因此对于资源的使用应做出适当的限制，尤其要注意以下几点：

### 1. 限制内存的使用

主要要注意对 JVM 内存的使用限制，避免导致频繁 Full GC 或内存溢出的现象，在内存的消耗上容易导致问题的主要有以下几种现象。

- 集合容量过大

在集合的使用上要注意限制容量，尤其是大多数集合对象都会自增长，例如 ArrayList、HashMap 等，稍不注意很有可能出现内存消耗过多的现象，举例如下：

```
public class A{
    private static Map<String,User> users=new HashMap<String,User>();
    public User query(String userName){
        if(users.containsKey(userName)){
            return users.get(userName);
        }
        // 获取 User
        users.put(userName,user);
        return user;
    }
}
```

上面这段代码在用户量小的情况下问题不大，但当用户数增加到一定程度，随着 query 的不同的 userName 越来越多，最终就会造成 users 中的 User 对象越来越多，导致 JVM 内存消耗过多，出现频繁的 Full GC 或 OutOfMemory，而这样的代码在压力测试时很难发现。对于类似上面这样的场景，要注意限制集合的大小，避免出现内存消耗过多的问题，最佳的方法是控制集合中对象所占 JVM 内存的比率，但此法实现上较为麻烦，因此多数都采用限制集合中对象数量的方式。可用的固定大小的集合对象不多，一种方法是基于现有集合对象做扩展来实现控制，另一种方法是通过计数或获取集合大小来实现控制。在限制了集合中对象数量后，要准备好当放入集合的对象超过集合可接纳的数量后如何处理的策略，例如是根据 LRU 踢出旧的对象，还是转为存储到磁盘等方式，这些策略目前开源的 cache 框架基本都是支持的。

- 未释放已经不用的对象引用

有些对象是程序中已经不再使用的，却没有在程序中释放对该对象的引用，这种情况也容易造成内存消耗过多的常见问题，举例如下：

```
public class TaskManager{
    private static List<Task> tasks=new ArrayList<Task>();
    private static Map<String,TaskProcessor> taskProcessors=new
    HashMap<String,TaskProcessor>();
    public void addProcessor(String taskType,TaskProcessor processor){
        // ...
    }
    class TaskThread implements Runnable{
        public void run(){
            // 扫描 tasks
            // 根据获取的 Task type 找到相应的 processor，并进行处理
        }
    }
}
```

```

    // 如处理成功，则标记 task 的状态为 Finished，并将 task 从 tasks 中移除
}
}
}

```

上面的代码中存在一个隐患，如每次都创建不同的 taskType，放入不同的 processor 对象，那么最终将会造成 TaskManager 的 taskProcessors 中存放了过多的 TaskProcessor 对象。解决办法是限制能放入的 TaskProcessor 的数量；另外一种方法则是将已经不需要使用的 TaskProcessor 从 TaskManager 中移除，避免因为持有引用导致无法被 GC。以上场景自然是第二种方法优先。

在 Java 应用中，还有一种状况是在使用 ThreadLocal 时容易造成不需要的对象仍然被引用，例如：

```

public static ThreadLocal<Object> threadLocal=new ThreadLocal<Object>();
public void execute(){
    threadLocal.set("//some object");
}

```

如在整个线程执行完毕后，都没有进行 threadLocal.set(null) 的操作，那么当此线程处于复用的情况下，放入 threadLocal 的对象就会一直等待线程退出才能回收，这对于大量使用线程池技术的应用而言尤其要注意。

由于 Java 程序并不是通过应用程序直接去做内存的分配和回收管理的，因此，对内存使用的控制就要特别注意，这就要求程序设计人员和编写人员对 JVM 的内存管理机制应有深刻的理解。

## 2. 限制文件的使用

限制文件的使用是非常必要的，但比较容易被忽略。例如最典型的是日志文件，在出现异常的情况下，日志频繁写入文件，所有线程都在争抢写文件的锁，倘若文件写得太大，就会造成写入速度严重下降，最终导致应用崩溃。对于上面这种情况，一方面是要控制单个日志文件的大小，另外一方面是要控制写日志的频率。

在 Java 中控制对文件使用的具体策略要根据不同的应用场景来具体制定。

## 3. 限制网络的使用

对于分布式 Java 应用而言，网络的使用是其中重要的一环，对网络资源的限制使用也是非常重要的，具体反映在以下两方面：

- 连接资源

连接是消耗资源非常明显的一个地方，毕竟每个连接都是要消耗资源的，客户端基本上都会采用连接池方式来避免对服务器端创建过多连接，而服务器端自身也必须控制避免有过多的连接，否则就有可能由于连接客户端机器数量太多，或程序问题导致服务器端创建的连接数过多，出现不可用问题。

- 操作系统 sendBuffer 区资源

在向服务器发送流时，都是先放入操作系统的 sendBuffer 区，而 sendBuffer 区是有限的。因此要适当控制往操作系统 sendBuffer 区写入的流数量，避免出现问题。

另一方面问题是由于在发送数据时都要先序列化成流，流在未发到操作系统 sendBuffer 区之前会在 jvm 中存活，这时就要特别注意不要有太多这样的流存在，造成 JVM 内存被耗光的现象。

因此，在网络的使用上最重要的是客户端通过连接池及服务端通过控制可接受的连接数来控制连接资源的消耗。对于 sendBuffer 区资源和 JVM 内存消耗的控制则通过流控来进行，流控通常的做法有当到达某阀值时直接拒绝发送，又或是根据内存的消耗状况逐步延迟应用方发送的速度。

#### 4. 限制线程的使用

为了支持更多的用户请求或提升系统的性能，在 Java 程序中通常都采用多线程的方式来实现，线程一方面要消耗物理内存和 JVM 内存，另一方面线程多也会导致 CPU 花费更多的时间在切换上，因此合理地控制线程的数量非常重要。

在 Java 程序中，通常会通过直接 new Thread 的方式，或使用 ThreadPoolExecutor 线程池的方式来处理多线程。new Thread 方式控制起来较为麻烦，要自行计数目前活跃的线程数。对于采用 Executors 创建线程池的方式则要注意合理地使用 Sun JDK 提供的接口，下面介绍 Executors 中三个典型的接口使用的风险：

- Executors.newFixedThreadPool

这个接口的风险在于使用了一个 Integer.MAX\_VALUE 大小的 LinkedBlockingQueue，这就意味着当线程不够用的情况下，所有需要处理的 Runnable 动作都会放入 LinkedBlockingQueue 中，这有可能造成内存上的风险。

- Executors.newSingleThreadExecutor

它的风险和 newFixedThreadPool 的风险是一样的。

- Executors.newCachedThreadPool

这个接口的风险在于可以创建出 Integer.MAX\_VALUE 的线程数，这有可能出现线程创建过多，应用宕掉的现象。

如希望避免以上风险，最好的方法是直接 new ThreadPoolExecutor，传入适当的参数。

在资源使用方面，始终都要记得资源是有限的，没有控制地使用必然会给应用带来风险，因此对于消耗资源的方面在设计阶段及在代码 Review 阶段都要特别注意。

### 从其他角度避免故障

除了以上从设计角度的考虑外，在代码编写过程中，还须确保对所使用到的框架及所使用到的 Java 类库的实现都有较深的掌握。一方面是避免使用不当，另一方面则是为了在出现问题时能够快速处理，

这也是之前第3章“深入理解JVM”以及第4章“分布式应用与SUN JDK类库”中所传递的信息；同时还要不时地组织代码Review，结合知识库以及团队的智慧尽可能避免代码中潜在的bug，在Review时可按照下面的风险卡对系统的潜在风险进行评估，如表6.2所示。

表6.2 风险卡模板

|           |             |
|-----------|-------------|
| 风险所属领域    |             |
| 风险点名称     |             |
| 风险所造成的结果  |             |
| 风险点发生的概率  | (非常低、低、中、高) |
| 检测风险发生的方法 |             |
| 风险应对的措施   |             |
| 造成风险的可能原因 |             |

其中风险所属领域主要分为：功能领域、内部功能领域、容错领域、自我保护领域及资源使用限制领域。

功能领域风险评估要求从使用者角度来看，看看其使用的是系统的哪些功能，如这些功能出现问题，会造成什么后果。

内部功能是指不对外提供的功能，评估时主要判断当内部功能出现问题时，会造成什么后果。

容错领域评估是指对外提供的接口、对象在误用的情况下是否有可能出现问题。

自我保护领域的评估是指对第三方的依赖出现问题的情况下，系统是否会出现问题。

资源使用限制领域的评估是指系统所使用到的资源是否会出现使用过度的现象。

通过对这几个领域的分析，寻找出系统的风险点，在寻找风险时，要记住的原则是即使风险发生的原因目前不知道，但还是要认为是有可能发生的，对于找出的风险制定相应的应对措施。在制定措施时最重要的是考虑收益比，即风险发生的概率及为了避免这项风险要付出的代价。很多时候，为了避免某低概率风险的发生，将系统改造得复杂了很多，反而增加了更多的风险，这就得不偿失了。在风险的应对措施上，通常监测、报警并进行手工处理是最简单且最有效的方法，这个在后面的6.2.2节“及时发现故障”中会进行阐述；原因可以随后再推测，毕竟有些时候造成风险的原因无法事先判断，但应该提供的支持是能够在风险发生时记录下必要的信息，以分析风险产生的原因。

除了从设计、代码编写、代码review、风险推测这四个方面尽可能地保障系统不出故障外，测试及最后的部署也要注意。

在测试阶段，功能测试可帮助保障系统的基本可用性，压力测试可帮助保障系统在高压下的可用性。

在部署阶段，要注意部署时不要对系统的可用性产生影响。这个说起来简单，做到并不容易，例如一个典型的部署步骤为先停掉集群中一半的机器，更新应用包，然后重启，如应用的启动过程太长，

另外一半提供服务的机器就无法支撑全部的流量，最终导致宕机，因此在部署时要尽量保证平滑，具体的方法可以参见 google 发的一篇关于平滑部署的论文<sup>15</sup>。而随着同一时间段需要部署的应用越来越多，如果是由人工方式来完成整个部署过程的话，就意味着部署的过程所消耗的时间会越来越长。因此对于一个大型应用而言，自动化的部署系统也非常重要，但要做到自动化部署还比较困难，要提供很多的基础设施，包括平滑部署的实现、部署包的快速分发、部署后的自动验证及回滚等。

从上面可看出，要保证系统尽可能不出故障，要从设计阶段、编码阶段、测试阶段及部署阶段多方位采取相应的措施。

## 6.2.2 及时发现故障

无论怎样去尽量地避免故障，毕竟不可预测的原因太多，基本上不可能在非运行期阶段做到完全避免故障，因此如何及时地发现故障就至关重要了，故障发现得越早，就可以更好地保障可用性，一个没有任何运行状况提示和报警的应用就像是一个没有仪表盘的汽车，无法知道车速、油量、转向灯是否亮，在这种完全不知情的状况下即使是一个熟练的司机，也是相当危险的。同样，应用也非常需要运行状况的监测数据，并在可能出现危险时提前报警，这主要依靠报警及对日志的记录和分析来做到，报警主要有单机状况的报警、集群状况的报警及关键数据的报警。

单机状况的报警通常用于报告产生了致命影响的点，例如 CPU 使用率过高、某功能点失败率过高、依赖的第三方系统连续出现问题等。

集群状况的报警，通常是在集群的访问状况、响应状况等指标和同期或基线指标对比出现过大偏差时发出。

关键数据的报警通常针对系统中的关键功能，例如交易类的系统，其最关键的数据是实时交易额，因此当交易额数据和基线指标对比出现过大偏差时，就需要报警。

日志的记录和分析主要是为集群状况及关键数据的报警提供数据，另一方面也是为了能够提前判断出系统中潜在的风险点，制定应对策略。

### 报警系统

要实现单机状况的报警，首先是要整理出报警点，基本上报警点可以来源于之前整理出的风险卡，对于风险卡中认为会造成严重后果的风险都应制定相应的监测方法，表 6.3 为不同风险领域常用的监测方法。

<sup>15</sup> [http://www.bluedavy.com/iarch/google/modular\\_software\\_upgrades\\_for\\_distributed\\_program.pdf](http://www.bluedavy.com/iarch/google/modular_software_upgrades_for_distributed_program.pdf)

表 6.3

| 风险领域     | 监测方法   |
|----------|--|
| 外部功能领域   | 通常可通过定时从外部执行相应的功能，判断返回的结果是否符合预期，从而监测风险，这种方式可以较快地发现风险；另外一种常用的方法就是当功能频繁出错时，程序内部主动向外报告。 |
| 内部功能领域   | 通常可采取和外部功能领域同样的方式去监测风险。  |
| 容错领域     | 通常采取的方法为在程序内部主动报告。   |
| 自我保护领域   | 通常采取的方法为在程序内部主动报告，例如当程序依赖的数据库执行频繁超过阈值时，就可主动报告发生了此现象。                                 |
| 资源使用限制领域 | 通常采取的方法为定时监测系统的资源使用状况，对于 java 程序而言，主要是系统的 load、jvm 内存状况。                             |

单机状况报警可以通过一个统一的报警包来实现，也可以是简单地基于 nagios 的 SNMP 方式定时扫描远程机器的报警信息文件，并根据报警规则来决定是否报警。单机状况报警的优点是能够及时地发现系统的故障，缺点是有可能在出现问题时会大量报警，导致寻找不到故障的根源，因此对于报警系统而言，分级处理非常重要。

要实现集群状况的报警，主要依靠日志分析的结果来报警，因此难度主要在日志记录和分析系统上。

要实现关键数据的报警，主要是找出系统中哪些是最关键的数据，然后通过操作数据库、消息系统或日志分析的结果等方式来获取关键数据，并根据一定的规则确定是否报警。

## 日志记录和分析系统

对于一个大型系统而言，每天要记录和分析的日志量将会非常大，因此实现起来会有一些难度。在实现时可参考一些开源的产品，例如 Facebook 的 Scribe<sup>16</sup>，此类产品基本的思路为各应用将需要分析的数据以特殊的格式写入日志文件中，然后通过 SNMP 采集的方式或同机器上 agent 推送的方式汇总到数据收集服务器上，并对数据进行分析生成报表，通常报表分即时报表、日报表、同期对比的报表等，eBay 则采用将要记录和分析的数据通过消息中间件异步发送出去，然后由相应的数据分析程序订阅此类消息进行分析，无论采用什么方式，这里面的关键点在于：

- 不能因为记录数据导致应用出现问题；
- 快速记录和分析所有应用的数据。

要做到这两点有一定的难度，因为通常要记录和分析的数据量很大，因此多数情况都会涉及按数据来源的分派、数据库的分库分表等技术。

有了报警系统和日志记录、分析系统后，就很容易快速发现故障。但当系统大了以后，当出现故

<sup>16</sup> <http://developers.facebook.com/scribe/>

障时，报警点太多，一方面会导致关键的报警信息被掩盖，另一方面不易知道造成故障的根本原因，也就是俗称的 root cause，最终造成虽然已经知道出现了故障，但查找故障的 root cause 花费了很长的时间，从而整个故障从发生到处理完毕的时间持续较长，对可用性造成极大的影响。

为了快速发现系统故障的 root cause，通常要设置一个系统总揽图来显示，结合系统的依赖关系，快速判断出 root cause 在哪里，更理想的情况是能够推测出目前系统中受影响的功能点，从而判断故障的后果，并制定相应的处理措施快速处理故障。

即时发现故障在各大互联网公司中都非常重视，从 Twitter、Facebook、盛大、51.com 等公开的 PPT 来看，可以看到它们都拥有一套强大的监测系统，这些监测系统通常具备了以下的一些特征：

- 有系统依赖关系的分析，便于辅助分析系统故障的 root cause；
- 有系统全局状态图显示，以便能迅速发现故障的 root cause，并根据故障点告知目前所影响的功能点；
- 根据报警规则、级别进行报警，对单机直接处理，避免单机问题扩散到全局；
- 根据报警信息的记录，跟进记录分析报警的原因及后续的处理步骤，方便为将来再次出现时做更快速的处理。

在有了一套强大的监测系统后，在故障发生前就可以依据报表系统提前发现潜在的风险点，或在故障发生时迅速得知 root cause，从而避免故障或减少故障持续的时间，保障系统的高可用性。

### 6.2.3 及时处理故障

在故障发生后，首先要做的不是去寻找故障发生的原因，而是最快速度地处理故障，保障系统的高可用，常见的故障快速处理措施有下面一些：

#### 1. 自动修复

自动修复是指在出现故障后系统自动对故障进行处理，修复故障，要做到自动治愈要求系统具备可学习性，通过学习总结所发生过的各类故障的处理措施，能够在将来发生故障时智能化地采取处理措施，这样实现难度很大，在未实现的情况下可以尝试在故障发生时，让系统提供一些建议，逐步完善。

#### 2. 执行风险点应对措施

当出现的故障属于之前判断出来的系统的风险点时，则可直接执行风险点的应对措施，快速修复故障。

通常而言，这些应对措施有手工发送命令给出现风险的系统，或是通过一个集中点发送命令给集群中的所有机器，快速修复故障。相对而言，到出故障的机器上手工执行命令修复是最保险的一种方式，但在机器数量太多的情况下，操作起来会比较麻烦。

### 3. 全局资源调整

当出现的故障属于某集群的压力过大时，可通过全局的资源调整，例如流量的分配、路由策略的调整来重新平衡全局的资源，从而保障系统的可用性。

更为先进的则是根据系统 QoS (Quality of Service) 来自动平衡全局资源，所谓 QoS 通常是指每秒需要支撑多少的请求量，但 QoS 也有可能是个随时段不同而变化的指标，这也是云时代的一个显著特征。

### 4. 功能降级

功能降级是指在出现故障又无法快速修复的情况下先把系统中的某些功能关闭，以保证核心功能的可用，也就是 eBay 在其 PPT 上提到的 Graceful Degradation<sup>17</sup>。这要求系统以功能为粒度进行管理，并制定相应功能级别，以便当需要的时候快速执行功能降级，这种情况的另一个实现策略是在部署时即根据功能级别划分为多套环境，或通过路由策略的调整将核心功能进行隔离，避免非核心功能的不可用影响到了核心功能。

### 5. 降低对资源的使用量

这个策略是指逐渐降低系统对资源的使用量，例如正常情况下配置的数据库连接池最大是 10 个，在出故障时调整为 5 个，同时缩短数据库连接等待的时间等，又或者是在资源紧张的情况下关闭消耗资源的操作，这种方式目前在国外的几家互联网公司也有使用，例如 Twitter<sup>18</sup>。

在实现上面这些故障处理的措施时，由于会对系统运行产生较大的影响，因此要特别注意谨慎进行这些措施的操作。就像 rm -rf 这种命令的执行，通常要提供完善的操作控制，包括操作权限、操作影响的预览、操作的审批、操作的记录、操作的报警以及操作的回滚等。

在处理完故障后，应注重分析和总结故障发生的根本原因，放入知识库，一方面是为了再次碰到类似故障时能够自动处理或快速处理，另一方面是为了积累经验，形成更多的保障可用性的设计原则、review 原则等，尽可能地避免故障再次发生。

避免单点故障和保障应用自身的高可用性是构建高可用的系统时最重要的两点要求。除了这两点外，在面对不断上涨的访问量及数据量时，还要采取一些其他措施来保障系统的高可用。

## 6.2.4 访问量及数据量不断上涨的应对策略

对于访问量不断上涨的情况，通常采取的应对措施是拆分系统及水平伸缩，拆分系统后简化了各个系统中的功能，并且使得其拥有更多的系统资源，从而提升了其所能支撑的用户访问量，通常系统的拆分按照功能进行，例如 eBay 将系统拆分为交易、商品、评价等。

<sup>17</sup> [http://www.bluedavy.com/iarch/ebay/ebay\\_arch\\_principles.pdf](http://www.bluedavy.com/iarch/ebay/ebay_arch_principles.pdf)

<sup>18</sup> <http://www.bluedavy.com/iarch/twitter/fixingtwitter.pdf>

随着数据量不断上涨，数据库中表的数据条数越来越多，读写越来越慢，同时随着前端机器的水平伸缩，数据库连接数很容易达到瓶颈，对于以上的状况，通常采取的方法为拆分数据库、拆分表及读写分离。

拆分数据库通常是按业务来进行，例如 eBay 将数据库拆分为用户、交易、商品、评价等，拆分后就可支撑更多的数据库连接，并提供更快的响应速度，缺点是跨库的操作比较麻烦。

拆分表通常是对业务中数据量大的表按照一定的规则来拆分，例如按照时间、hash 算法、取模等，表拆分后单表的读写速度会得到一定的提升，缺点是跨表的操作比较麻烦。

读写分离适用于读写比例高，且对实时性要求不是非常高的应用，通过提供一个写库、多个读库，来提升读写的速度，此时技术上的挑战是如何快速地完成数据从写库复制到其他的读库。

以上这些应对数据量上涨的技术在下一章中将详细介绍。

水平伸缩则是用增加机器的方法来支撑更高的访问量，通常增加机器的动作必须事前进行，如等到访问量上涨后才增加，必然会出现问题，因此什么时候需要增加多少台机器是门大学问，通常又称为容量规划。

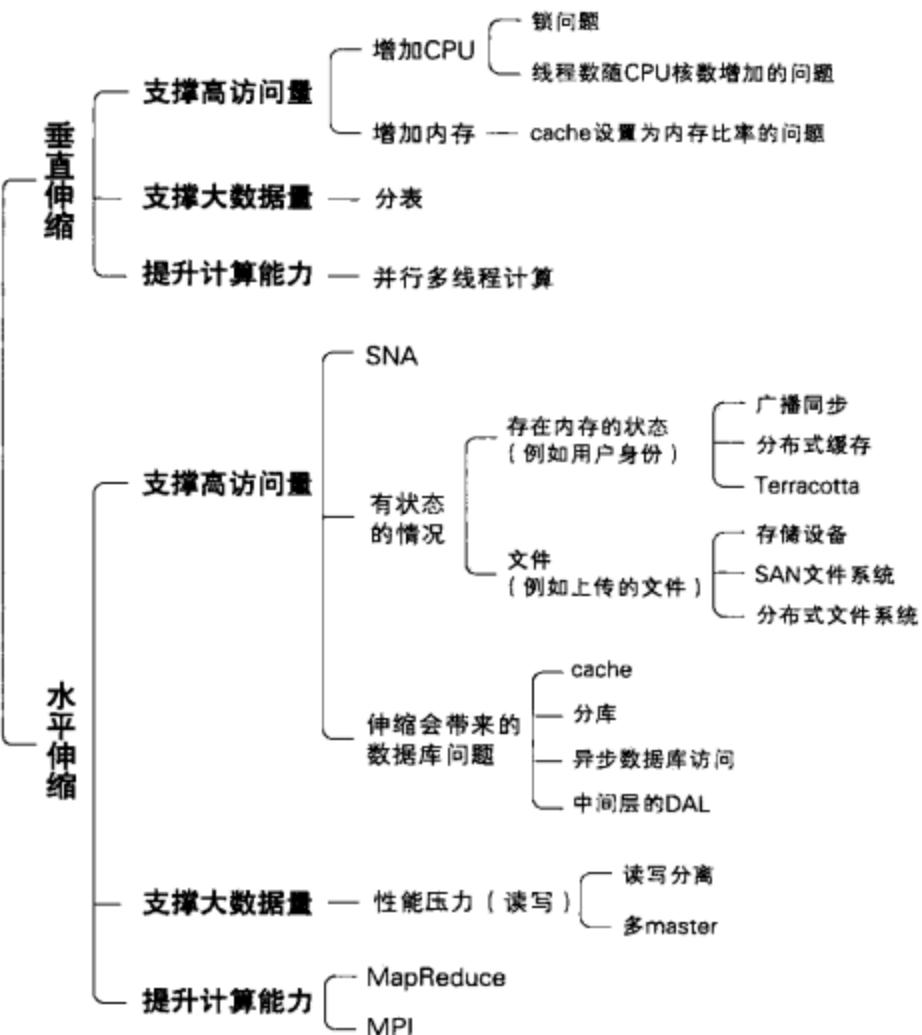
容量规划首先要求能够监测当前系统中相关的信息，例如对于数据库而言，通常需要监测的功能信息有每秒执行的查询数、插入数、更新数、删除数及活动的连接数等。要监测的系统信息有 CPU、文件 IO、网络 IO 及内存，基于这些监测信息分析系统的关键资源消耗点；并基于性能指标（例如必须在 3 秒内响应）等划定上限，通过测试来寻找达到此上限能够支撑的访问量、数据量或流量；最后根据历史的数据对未来的发展趋势进行预测，并结合上面的上限值评估什么时候要进行扩容或必须调整架构，通常扩容比架构调整会容易得多，但扩容有时也取决于架构是否能够支撑。在做容量规划时最重要的一点是结合现状去规划，而不用去做性能调优后状况的假设，容量规划涉及众多的知识点，也是业务和技术结合的体现。在本书中不再拓展这方面的知识，感兴趣的朋友可以进一步阅读 Flickr 工程师写的《Web 容量规划的艺术》一书。

保障系统高可用性是一项全面的工作，要从多方面进行考虑，在制定方案时要从收益比、性能影响、硬件成本等角度来做出适当的权衡。



# 第7章 构建可伸缩的系统

构建可伸缩的系统



对于大型应用而言，一直都在追求以一种优雅、简单的方式来应对访问量及数据量的增长。通常这种优雅、简单的方式是指无须改动软件程序，仅通过升级硬件或增加机器即可应对访问量及数据量增长带来的挑战，毕竟软件改造的成本相对较大。如果能做到仅靠升级或增加硬件就能解决问题，那就太完美了，但现实通常并不会这么理想，为了能够达到这样的目标，在软件上往往也要付出巨大的努力。

通常将通过升级或增加单台机器的硬件来支撑访问量及数据量增长的方式称为垂直伸缩，垂直伸缩的好处是技术难度相对较低，对于小型应用而言是一种不错的选择。其缺点是机器的硬件是无法不断升级和增加的，很容易达到瓶颈，而如果想升级为更高级别的机器时通常带来的成本是指数级的，因此对于大型应用而言，垂直伸缩不是一种好的选择。

通常将通过增加机器来支撑访问量及数据量增长的方式称为水平伸缩，水平伸缩从理论上来说并没有瓶颈，其缺点是对技术上有较高的要求，另外在增加机器时，要考虑机器的增加对于空间、能源的占用。空间的占用要求机柜要足够充足，能源的占用则意味着电力、网络带宽的消耗，这些都会给应用的运营带来更高的成本。除此之外，当机器数量大幅度增加后，机器硬件出现故障的几率也将大幅度上升（Google 的经验表明，每 10000 台机器中，每天都将报废一台），此时软件上的容错、机器的管理和维护就显得至关重要了。

无论是采用垂直伸缩还是水平伸缩，我们都希望在伸缩后是线性的，即当一台机器能够支撑 200 万访问量时，我们希望增加到 5 台机器时，就可支撑 1000 万的访问量。但在实际的场景中，随着不断地伸缩，最终将会到达一个瓶颈，在这个瓶颈后再做垂直或水平伸缩，有可能带来负效果，此时要考虑的就是从技术上做出相应的改进。

垂直伸缩和水平伸缩除了可用于支撑更高的访问量和更大的数据量外，对于计算类型的系统，在设计良好的情况下还可提升计算能力。

垂直伸缩和水平伸缩各有一定的优点，因此在应用中通常可以混合采用，具体采用什么方案通常取决于成本。一方面要考虑硬件升级所增加的成本，另一方面要考虑软件改造的成本，以下具体介绍如何基于垂直伸缩、水平伸缩来支撑高访问量、大数据量及提升计算能力。

## 7.1 垂直伸缩

在垂直伸缩前，要分析系统的瓶颈，从而针对性地根据瓶颈进行硬件的升级或增加，另外就是要从软件方面保证系统在垂直伸缩后服务能力的线性增长，以下就具体介绍软件方面要作出的努力。

### 7.1.1 支撑高访问量

Web 应用随着访问量的增长，通常其瓶颈会出现在 CPU 或内存上，网络 IO 或磁盘 IO 出现瓶颈的几率较低，在此仅分析当增加 CPU 或内存时，应用如何做到线性地增长。

## 1. 增加 CPU 后

要做到增加 CPU 后系统的服务能力线性地增长，要求系统能够随着 CPU 的增加，响应速度提升或同时可用于处理请求的线程增加，主要有以下三种情况会造成增加 CPU 后系统的服务能力无法线性增长。

- 锁竞争激烈

锁竞争激烈时造成很多线程都在等待锁，此时即使增加 CPU，却无法让线程得到更快的处理，也无法开启更多的线程来处理请求，应对的策略是尽可能地降低系统中锁竞争的现象。具体的方法可参见性能调优中降低锁竞争的部分，在降低了锁竞争现象后，一方面是响应速度的提升，另一方面则可开启更多的线程来支撑高访问量。

- 用于支撑并发请求的线程数是固定的

在 Java 应用中，依靠启动多个线程来支撑高并发量，如启动的线程数是固定的，那么即使 CPU 增加了，系统的服务能力也不会得到提升，因此最佳的方法是根据 CPU 数 (`Runtime.getRuntime().availableProcessors()`) 计算一个合理的线程数。例如 Sun JDK 的并行 GC 线程数就是根据 CPU 数计算得到的，这样当 CPU 增加后，GC 的速度就可提升。

- 单线程任务

对于单线程的任务，增加 CPU 不会带来任何的提升。此时可以考虑按照 CPU 数对任务进行合理地划分，以能够通过启动多个线程来并行地将任务分解成多个任务完成，在 Sun JDK 7 中提供了 Fork/Join 来实现<sup>1</sup>。

在解决了上述三个问题后，通常只要增加 CPU 就可提升系统所能支撑的访问量。

## 2. 增加内存后

要做到增加内存后系统的服务能力线性增长，要求系统能够随着内存的增加，响应速度提升，主要有以下两种情况会造成增加内存后系统的服务能力无法线性增长。

- cache 的集合大小是固定的

系统中通常会借助 cache 来提升性能，而为了避免内存资源消耗过多，会限制用于 cache 的集合的大小，如这个大小是固定的，那么即使增加了内存，能放入 cache 的数据也不会增多。

解决这种问题的方法是根据可用的内存计算出一个比例来控制 cache 的数据的大小。

- JVM 堆内存是固定的

JVM 堆通常是在启动参数中设置的，因此有些时候可能会出现增加了内存，但 JVM 堆大小没有调整。因此在增加了内存后要相应地对 JVM 堆内存的大小进行调整，操作系统位数对可设置的大小有限制，对于要使用超过 2GB 内存的 JVM 堆而言，操作系统升级到 64 位是很有必要的，在调整了 JVM 堆内存大小后，值得注意的是要避免 GC 造成 CPU 出现瓶颈。

<sup>1</sup> <http://www.ibm.com/developerworks/cn/java/j-jtp11137.html>

解决了上面两个问题后，通常只需要增加内存就可提升系统的响应速度，从而提升系统所能支撑的访问量。

### 7.1.2 支撑大数据量

数据量增加到一定程度通常会造成数据库的读写速度大幅度下降，除了数据库软件本身要做到在 CPU、磁盘或内存增加后能提升响应速度外，从数据库使用的角度也可做出一些优化，主要的优化手段是分表。

当单表中存储的数据量增大后，即使垂直伸缩了也有可能仍然达不到所期望的响应速度，此时可选择采取分表的方式。分表通常采用的方法为按主键 ID（例如 Twitter<sup>2</sup>）、按时间等方式，具体的分表策略要根据业务而定，采用分表方式通常要求应用做一定的处理。例如根据 `userid` 对 `user` 表进行分表，那么应用在访问 `user` 的信息时，就不能再用 `select username from user where userID=1220`，而必须根据 `userID` 及分表的规则先行计算出要查询的表，然后再组装出相应的 sql。

分表带来的好处是单张表的数据量减少，因此其读写的速度可以得到一定的提升，其带来的问题是一方面是开发的复杂，通常须借助 DAL（后续章节中进行介绍）来解决；另一方面是分页查询会比较难处理及有可能产生跨表查询的需求。

### 7.1.3 提升计算能力

对于计算而言，通常最关键的是 CPU 资源，在增加了 CPU 后，可通过增加用于计算的线程或拆分计算的步骤采用并行的方式来提升计算能力。例如在计算  $1+2+\dots+40000$  时，可根据 CPU 数进行切分，假设 CPU 为 4 个，那么可切分为 4 个线程来并行计算（例如一个计算  $1+2+\dots+10000$ ），在 4 个线程计算完毕后再进行合并，这要求这类拆分并行计算的规则是按照 CPU 核数来计算的，这样才能在增加 CPU 后得到计算能力提升。

## 7.2 水平伸缩

水平伸缩相对垂直伸缩而言，在技术上要求高得多，但也可带来无限扩展的可能，这也是目前各大网站通常采用的支撑大访问量的方法，为了做到增加机器后系统的服务能力能线性增长，在软件方面要做很多的努力，以下来介绍一下软件方面要做的变动。

### 7.2.1 支撑高访问量

对于水平伸缩的方式而言，首先要解决的并不是能否做到线性增长的问题，而是系统能否水平伸缩。

<sup>2</sup> <http://www.slideshare.net/nkallen/q-con-3770885>

系统在运行时通常会存在一些状态，例如用户登录状态。对于这类系统，当从一台机器增加到两台时，就会出现一个问题：假设用户登录时访问到其中一台机器，如其在操作需要登录的页面时又访问到另外一台机器时，则需要重新登录，对于这类有状态的系统，需要借助一些技术方法来解决才可水平伸缩。

因此对于水平伸缩而言，最佳的情况是应用是无状态的，但系统很难做到完全没有状态，业界通常采用一种称为 SNA (Share Nothing Architecture)<sup>3</sup>的体系来指导如何构建无状态的应用。SNA 架构在实现时通常采用的方法是将有状态的部分集中放入缓存或数据库中，通常数据库采用集中存储的方式，因此这里最需要关注的是放在缓存中的状态如何做到支持水平伸缩。

## 缓存状态的水平伸缩方法

例如对于用户登录的信息，通常采用缓存的方法来记录它们，当进行水平伸缩时，要保证的就是如何让各台机器获取的缓存信息一致，对于 Java 应用而言，通常可采取的方法有如下三种。

### 1. 广播同步

广播同步通常基于 Multicast 实现，当用户登录时，系统的行为如图 7.1 所示：

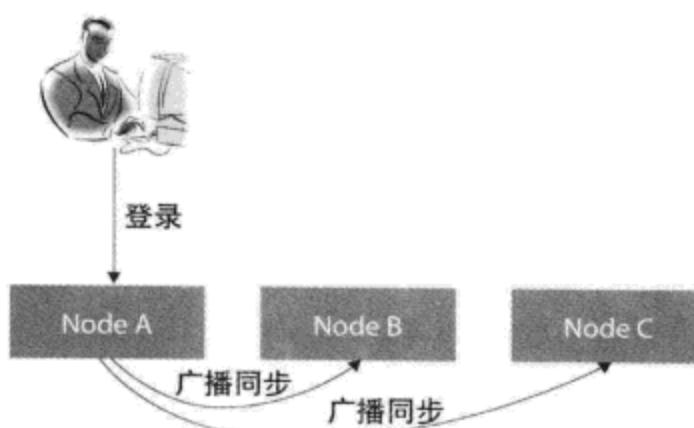


图 7.1 基于广播同步实现缓存中的状态一致

用户登录时，假设其访问到的为 Node A 机器，Node A 机器在验证用户的身份后，如通过，则将此用户已登录的信息广播，广播后 Node B、Node C 机器也就收到了此信息，因此即使之后用户访问到 Node B，也不会要求其重新登录。

Java 中用于实现广播同步的开源软件主要为 JGroups<sup>4</sup>，JGroups 除支持默认的 Multicast 方式广播外，也可借助 TCP/IP 实现更加可靠的广播，Jetty 和 Tomcat 的 Http Session 信息的同步就基于 JGroups 实现。

广播同步的方式实现起来较为简单，且不会带来机器成本等的增加，在集群中机器数量及需要缓存的数据很少的情况下，采用广播同步的方式来实现节点中内存状态的一致是不错的选择。并且由于状态信息缓存在和系统同一个 JVM 中，性能上可以得到保证。值得注意的是广播同步会有一定的延时，要确保这个延时对于应用场景而言是可接受的。

<sup>3</sup> [http://en.wikipedia.org/wiki/Shared\\_nothing\\_architecture](http://en.wikipedia.org/wiki/Shared_nothing_architecture)

<sup>4</sup> <http://www.jgroups.org/>

在要缓存的数据增多了后，单台机器会无法容纳，节点增多的情况下，同步消耗的时间会越来越长，最终导致所产生的延时会超过可接受的范围。

## 2. 分布式缓存

对于需要缓存的状态数据增多的情况，可采取分布式缓存的方案来解决，分布式缓存是指由多台机器来构成一个巨大的缓存池，每台机器上缓存一部分数据，其本身也是水平伸缩的。例如假设一台机器可支撑 4GB 的数据缓存，如需要支撑 24GB，则可采用 6 台机器，在采用分布式缓存后，当用户登录时，系统的行为如图 7.2 所示：

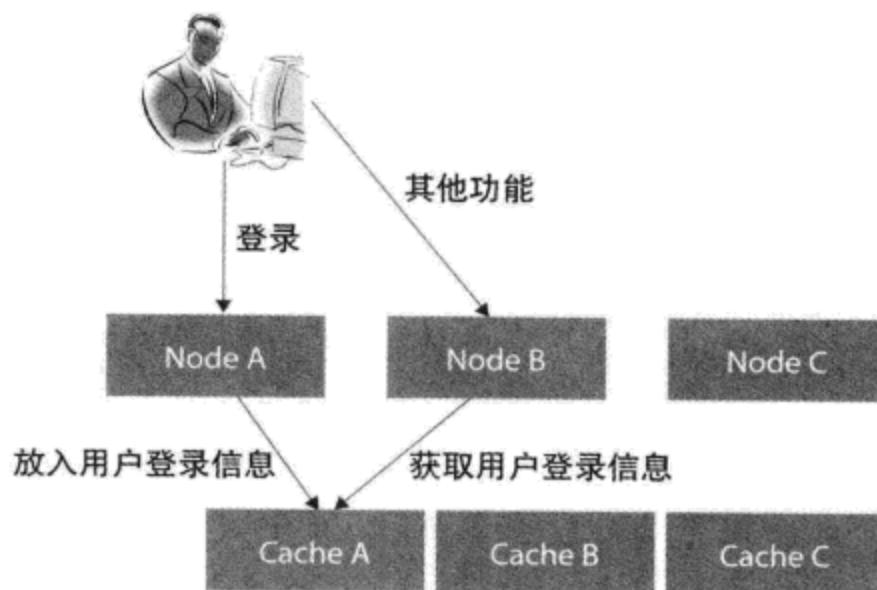


图 7.2 基于分布式缓存实现缓存中的状态一致

用户登录时访问的是 Node A 机器，Node A 机器在验证通过了用户的信息后，将用户的信息放入分布式缓存集群中的某台机器上，当用户登录后访问其他功能进入 Node B 机器时，Node B 即可从分布式缓存集群的某台机器上寻找用户的登录信息。

从上可见，对于分布式缓存而言，需要解决的是 Node A 和 Node B 在操作同一用户的登录信息时能到分布式缓存集群的同一台机器上操作。最简单的方法是对用户 ID 进行 hash，并用此 hash 值与缓存集群中可用机器的总数进行取模，这种方式通常又称为 hash 取模，其缺点在于当分布式缓存集群的机器发生增减时，可能会出现大量缓存失效的现象，一致性 hash<sup>5</sup>是解决这种问题常用的方法。

一致性 hash 的方法如图 7.3 所示。

它采用的方法为首先求出缓存的节点机器的 hash 值，将其落到  $0/2^{32}$  的圆环上，当有 key 要存储时，即按同样的方法计算出 key 的 hash 值，并同样落到圆环上，从此位置顺时针寻找圆环上的第一台节点机器。当 hash 值超过  $2^{32}$  仍然未找到节点机器，则以第一台机器作为该 key 存储的位置。

这种方式的好处是当增加节点时，仅影响落在节点逆时针方向的一小段范围的 key，例如增加一个节点到 node2 和 node4 之间，那么影响到的仅为落在 node2 到这个新增加的 node 之间的 key，当减少

<sup>5</sup> <http://portal.acm.org/citation.cfm?id=258660>

一个节点时，影响到的也只是其逆时针方向的一小段范围的 key。例如去掉上面的 node2 节点，影响的仅为之前存储在 node2 节点上的 key，这相对 hash 取模的方式有了很大的改善。

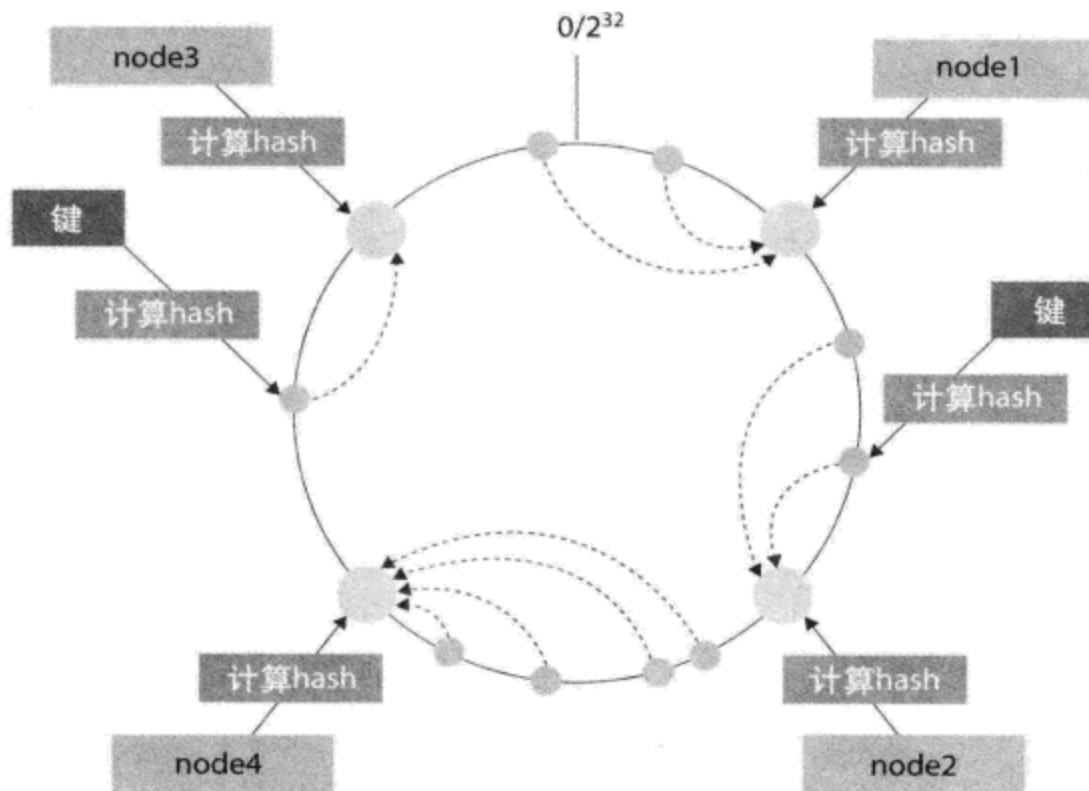


图 7.3 一致性 hash 算法图示

但由于一致性 hash 算法采用对机器做 hash，然后落在圆上，当节点较少时，有可能会出现这些机器节点是均匀分布的现象，这可以采用虚节点的方式来改进。例如现在有四个节点，直接按 hash 计算后，可能会出现某节点负责的区域更广，而其他区域更窄，虚节点的方式下则可划分为两百份，然后每五十个虚节点指向一个真实的节点机器，这样就可保证节点的均匀分布了。

目前开源界中分布式缓存使用最为广泛的是 memcached<sup>6</sup>，由 Facebook 开源，多家互联网公司均采用它实现缓存，memcached server 采用 c 实现，提供了多种语言版本的客户端，包括 php、Java 等。memcached 客户端支持简单的对 key hash，然后除以缓存机器的节点数取模的方式来寻找机器；也支持按一致性 hash 的方法来进行寻找，在实现时多数采用 ketama 算法<sup>7</sup>实现。

每个 key 的数据在 memcached 集群中只保存一份，这对于有些缓存高度敏感（例如缓存失效，数据库无法承受相应的压力）的应用而言，会产生一定的风险。Facebook 采用的方法为构建多个 memcached 集群，将 key 进行冗余存储，这样当一个集群中的某台机器出现问题时，并不会影响到这个 key 缓存的值。

另外由于标准的客户端需要配置好缓存集群的地址列表，一旦缓存集群节点要增减时，就必须修改所有客户端配置的地址列表，这样使用起来就不太方便。

<sup>6</sup> <http://memcached.org/>

<sup>7</sup> <http://www.audioscrobbler.net/development/ketama/>

除了以上两种可支持用户登录信息水平伸缩的方式外<sup>8</sup>, Terracotta<sup>9</sup>也是一个可以考虑的选择。Terracotta 是一个 JVM 级的集群框架, 可实现多台机器的 JVM 堆对象的共享等; 另外用户登录信息还可直接改造为放入 cookie 中, 那就不要广播同步或引入分布式缓存了, 其他的一些不适合放入缓存的状态信息则可基于数据库来统一存储, 这样仍然能够保证应用是可水平伸缩的。

## 文件的水平伸缩方法

在系统中通常会有上传文件这类场景, 对于水平伸缩的情况而言, 就必须实现在集群中 Node A 机器上传的文件, 在 Node B 也能看到, 通常有以下三种办法可实现。

### 1. 直连式存储 (DAS: Direct-Attached Storage)

直连式存储通常是指各系统直接与一个集中的存储设备连接, 从而进行文件的读取, 对于系统而言, 完全无须程序方面的改造, 在采用这种方式时, 上传文件的系统行为示例如图 7.4 所示:

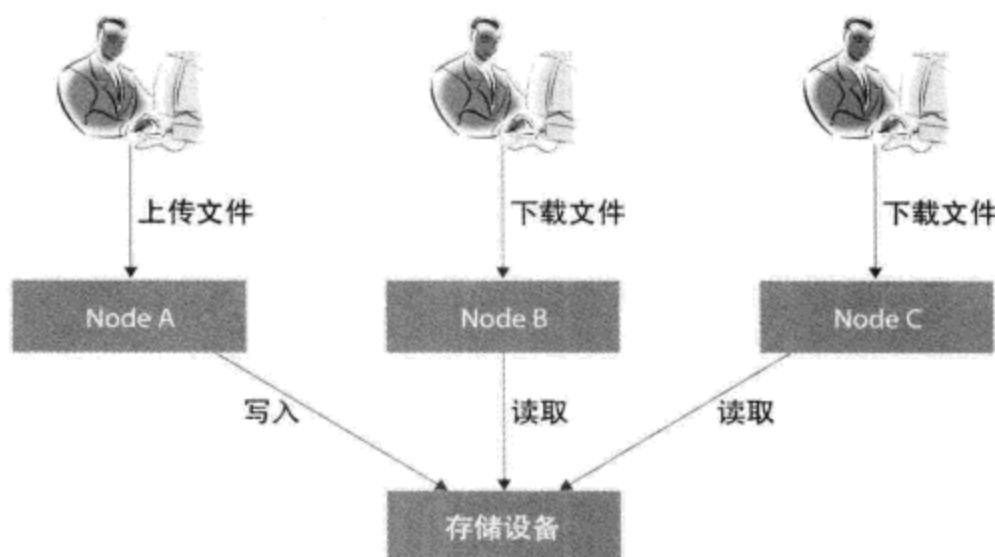


图 7.4 DAS 方式示例

它带来的问题一方面是存储设备在水平伸缩上支持得不好, 会导致存储的文件容量是有限的, 并且随着存储的文件越来越多及需要访问存储设备的节点越来越多, 性能明显下降; 另一方面则是成本的增加, 存储设备通常比较昂贵, 并且要有主备两台机器来避免单点故障。如系统希望存储海量的文件, 那么就要特别在成本上多加考虑了。

### 2. 网络存储 (Fabric-Attached Storage)

网络存储主要有 NAS<sup>10</sup> (Network-Attached Storage) 和 SAN<sup>11</sup> (Storage Area Network) 两种方式。

NAS 采用网络 (TCP/IP、ATM 等) 技术, 通过网络交换机连接存储系统和服务器主机, 采用标准的文件共享协议 (NFS、CIFS 等), 从而实现异构平台下的文件共享, 但当并发访问同一数据时, 文件

<sup>8</sup> 用户登录信息比较特殊, 还可基于负载均衡的 sticky session 来做到。

<sup>9</sup> <http://terracotta.org/>

<sup>10</sup> [http://en.wikipedia.org/wiki/Network-attached\\_storage](http://en.wikipedia.org/wiki/Network-attached_storage)

<sup>11</sup> [http://en.wikipedia.org/wiki/Storage\\_area\\_network](http://en.wikipedia.org/wiki/Storage_area_network)

的读写速度会大幅度下降。

对于使用者而言，NAS 提供的是文件系统，只须在机器上 mount 相应的文件系统即可操作，使用较为简单。但由于 NAS 方式没办法将多个 NAS 设备联合成对外的一体，因此如一个 NAS 设备的空间不足，那么就只能在机器上 mount 多个 NAS 设备的文件系统了。

SAN 采用光纤方式连接磁盘阵列和服务器主机，实现将多个磁盘阵列构成一个对外统一的存储区域。一方面这可以提升存储空间的利用率，另一方面也具有较好的伸缩性。当需要扩充某对外提供的 block 空间时，只需继续增加 SAN 设备即可。

对于使用者而言，SAN 提供的是块设备（Block Device），因此如希望多台机器共同操作同一 Block，则要采用具有同时操作同一 block 功能的文件系统，例如 IBM 的 TotalStorage SAN 文件系统。相对而言，SAN 的硬件成本及使用成本都更高。

NAS 和 SAN 各有优缺点，目前已逐步发展为采用 NAS+SAN 的方式来更好地满足网络存储的需求。

### 3. 分布式文件系统

分布式文件系统采用的方法由众多普通 PC Server 机器构成巨大的存储池，每台机器只存储一部分数据，其本身通常可非常好地支持水平伸缩。例如一台机器能存储 500GB 数据，那么当要存储 2000GB 数据时，只要增加到四台机器即可，在采用分布式文件系统后，上传文件的系统行为如图 7.5 所示：

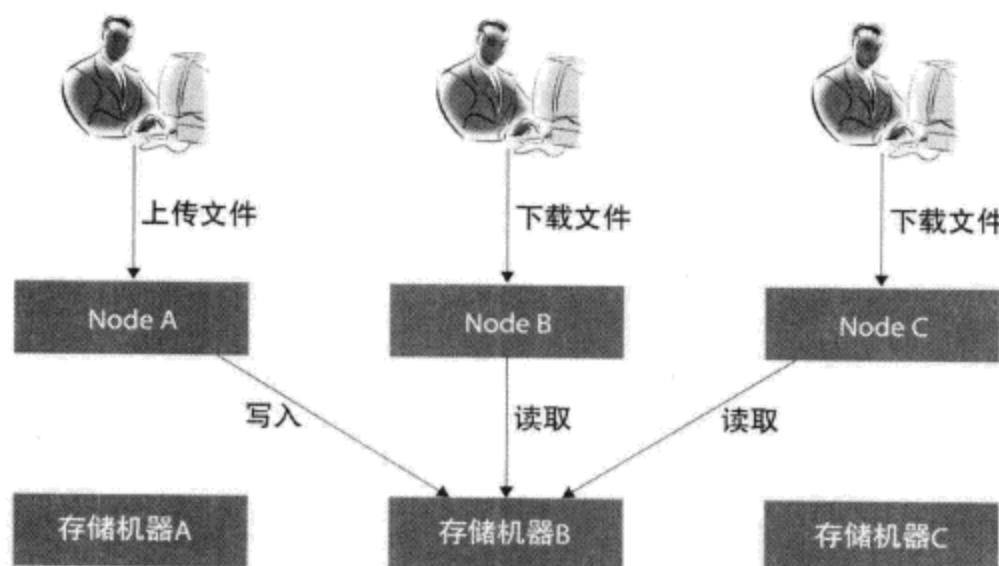


图 7.5 分布式文件系统示例

和分布式缓存一样，分布式文件系统也要保证在 Node A 上传 helloworld.txt 到存储机器 B 后，Node B、Node C 在下载 helloworld.txt 时能找到存储机器 B，典型的分布式文件系统有 Google 的 GFS<sup>12</sup> 及 Yahoo! 的 HDFS<sup>13</sup>，HDFS 遵循 GFS 实现，因此可以认为 HDFS 和 GFS 是相同的。

在 GFS 中，对于上面问题采取的解决方法是增加一个称为主服务器的单点机器。当 Node A 要上传文件时，Node A 上的 GFS Client 会将文件按固定大小划分，并向主服务器提交文件名和块索引信息，

12 <http://labs.google.com/papers/gfs-sosp2003.pdf>

13 [http://hadoop.apache.org/common/docs/current/hdfs\\_design.html](http://hadoop.apache.org/common/docs/current/hdfs_design.html)

从而得到要存储的目标机器及位置，主服务器根据目前各存储机器的存活状态、硬件使用率等来决定块需要存储到的目标机器，之后 Node A 将数据存储到目标机器的相应位置上。主服务器负责记录文件和块的命名空间、文件到块的映射及每个块副本的位置。为了保证安全可靠，同时将数据复制到多个存储机器上，复制的份数可在主服务器上进行设置，当 Node B 要读取此文件时，则只要从主服务器上获取此文件划分的存储位置列表，然后随机挑选机器进行读取，最后根据块的索引进行合并即可。

主服务器与各个存储机器进行心跳，以保证存储机器是有效的。Node A 这类应用端为了避免每次都要和主服务器进行通信，可在一定时间内缓存文件的元数据信息。

GFS 在存储时将文件分割为固定的块，块的大小默认情况下固定为 64MB。对于大文件，划分为多块存储在不同的服务器上，读写时都是同时操作多台服务器，因此速度比较快。但对于小文件而言，则意味每次只能从一台服务器上读取，当存取大量小文件时，会对向主服务器的查询造成一定的压力。因此 GFS 并不适用于存储大量小文件的系统，GFS 之所以将块设置为 64MB 的原因是 Google 的业务场景中大部分都是大文件，另外对于主服务器而言，块越大，要存储的信息就越少，这样可以降低主服务器的内存消耗和压力。

由于 GFS 这类分布式文件系统只须采用大量廉价的机器就可构成一个巨大的存储空间，并且具备了很好的水平伸缩能力，因此目前多数互联网公司都选择采用分布式文件系统来存储海量文件。

## 应用的水平伸缩方法

在系统建设初期，会采用将各种业务都放在同一个系统的方式，这会导致这个系统日渐庞大，所需的资源（CPU、内存、数据库连接）越来越多，在进行水平伸缩时要考虑系统里各种业务会造成资源增加的现象，这种状况会导致水平伸缩很难进行。例如增加机器后就造成了多个数据库连接的增加，对于这样的状况，通常采取拆分应用的方式来解决。

拆分应用通常按照业务领域来划分，即将原在同一系统中处理的功能拆分到各个不同的业务系统中，例如 eBay 将其业务系统拆分为商品、用户、评价、交易等，拆分后的结构如图 7.6 所示：

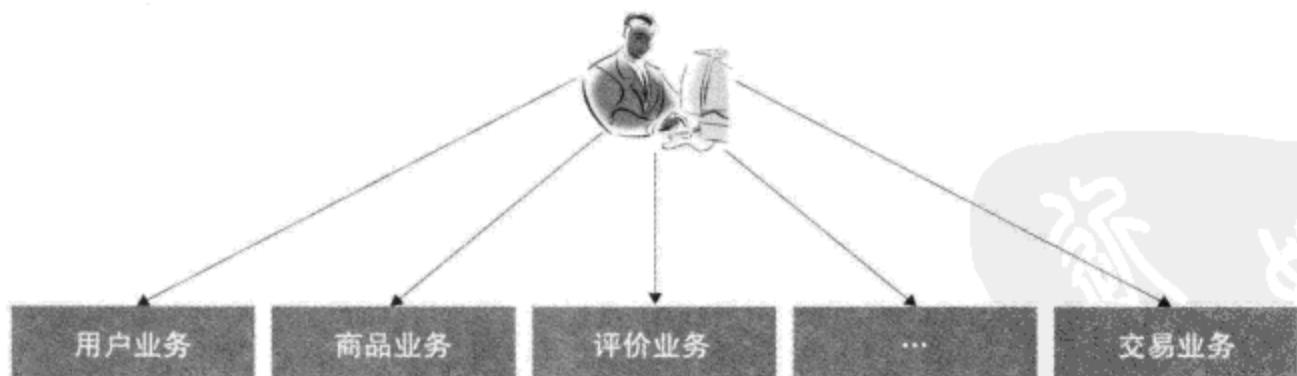


图 7.6 拆分应用后的系统结构

拆分后单个系统的功能较为单一，在进行水平伸缩时更容易判断其伸缩会带来哪些资源的增加，并且由于之前由众多功能共同分享的机器资源现在变为独享，因此除能更好地支撑水平伸缩外，对于提升系统的响应速度也会起到很好的作用。

## 水平伸缩后带来的数据库问题

系统水平伸缩后，对于数据库而言，通常会带来的一个问题是数据库连接池的增加，而由于大多数数据库对水平伸缩支持得不好，因此通常要做一些处理来解决这个问题，避免由于数据库连接资源不够限制了系统的水平伸缩能力，通常采用的有以下四种方法。

### 1. 缓存（cache）

通常可在系统中会采用多种缓存尽可能地避免访问数据库，其结构如图 7.7 所示：

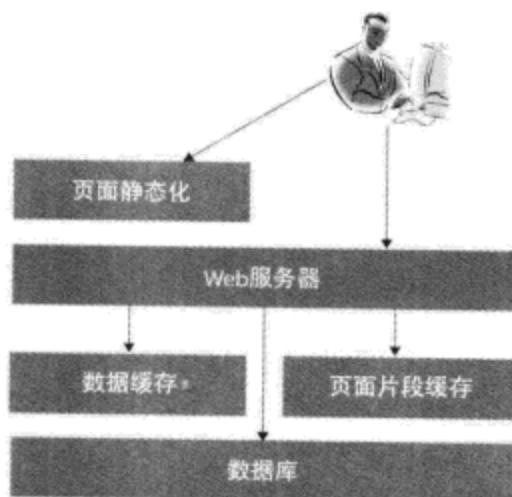


图 7.7 缓存的使用示例

- **页面静态化**

对于一些信息变动不多，且无须根据访问用户来进行展示的页面而言，可转为生成静态的页面，这样当用户访问这些页面时，就无须再从数据库中读取了，一方面可提升系统的响应速度；另一方面可降低对后端的访问压力，从而降低数据库连接数，例如新闻页面或网站的通告页面等可如此。

- **页面片段缓存**

对于一些不能静态化的页面，页面中仍然会有些部分相对变化不大，且无须根据访问用户来展示的片段。对于这样的页面，可将这些片段信息进行缓存，当用户访问时，Web 服务器只需从缓存中获取这些片段信息即可，而无须对数据库进行访问，通常可基于 ESI<sup>14</sup>等方式实现。

- **数据缓存**

对于一个系统而言，大部分的功能仍须与数据库交互才可完成，对于这些需要访问数据库的功能，可通过将数据缓存来提升响应速度和降低对数据库的压力。例如用户信息，需要缓存的数据量会比较大，因此通常会采用分布式缓存来实现。

数据缓存仅适用于变化不多的数据信息，如变化太多，一方面无法保证缓存的高命中率，另一方面导致需要频繁地更新缓存，性能反而不如直接操作数据库好。

<sup>14</sup> [http://en.wikipedia.org/wiki/Edge\\_Side\\_Includes](http://en.wikipedia.org/wiki/Edge_Side_Includes)

以上三种缓存方式对于降低数据库连接可起到明显的作用，这对于实现水平伸缩也会有一定的帮助。

## 2. 分库

在系统发展的初期，通常会将各种不同的数据放在同一个数据库中，随着业务的多元化及业务系统的水平伸缩，数据库的连接数会成为稀有资源，对于这种情况，分库是个不错的选择。

分库通常按照业务领域将原来存储在同一个数据库的数据拆分到多个数据库中。例如 eBay 就按照其业务领域拆分为商品、用户、评价、交易等数据库，每个数据库只用处理相关业务的数据，因此可用的数据库连接数就会得到很大提升，分库后系统转变为类似如图 7.8 的结构：

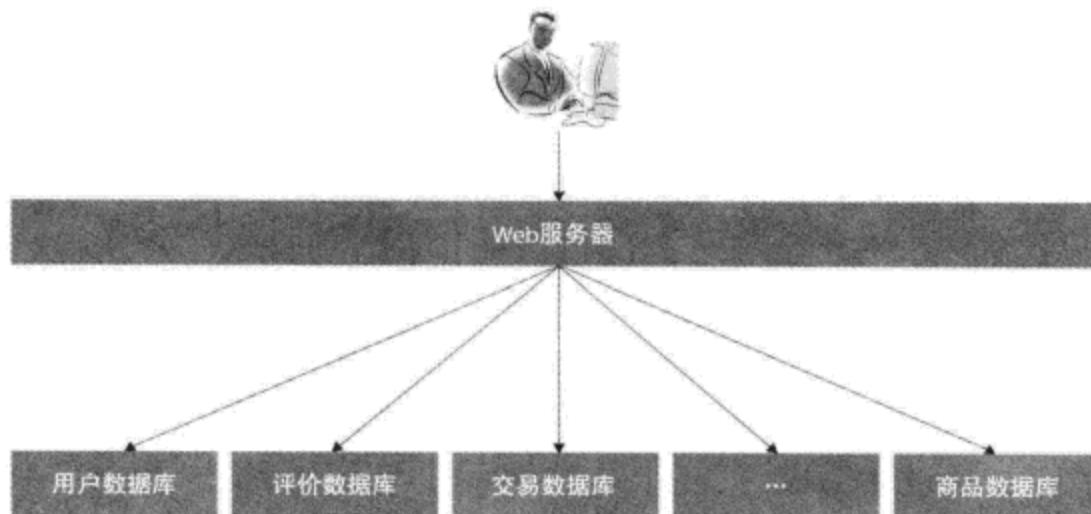


图 7.8 分库后的系统结构

分库对于系统而言，通常会带来如下几个问题：

- 对于已有的系统，这意味着要将之前访问同一个库的地方修改为访问相应的库，通常这会造成已有众多系统都要修改；
- 对于有些要跨数据库操作的业务而言，就变得较为复杂了，通常要从原来的联合查询转变为多次查询，而写入类的如要保证事务则可能要引入分布式事务。

分库是系统发展到一定规模后通常采用的手段，其对于支撑业务系统的水平伸缩可以起到很大的作用。

## 3. 异步数据库访问

目前大部分数据库访问仍然采用同步方式，每进行一次数据库操作就要占用一个数据库连接，并且要等到数据库操作执行完毕才会将连接释放。这对于高并发的系统而言就很容易出现数据库连接不够用及数据库资源竞争激烈的现象，异步数据库访问是解决这种现象的一种方法。

异步数据库访问要将传统的通过阻塞 IO 访问数据库的方式转变为采用非阻塞或异步 IO 的方式来访问。为了支持连接的复用，访问数据库端在执行数据库操作时要生成一个请求 ID，数据库服务器在执行完毕后要将此请求 ID 传回访问端，在支持了连接复用后，就可做到用很少的连接来支撑大量的数据库访问及避免连接资源竞争的现象，在采用异步数据库访问后访问数据库的操作流程如图 7.9 所示：

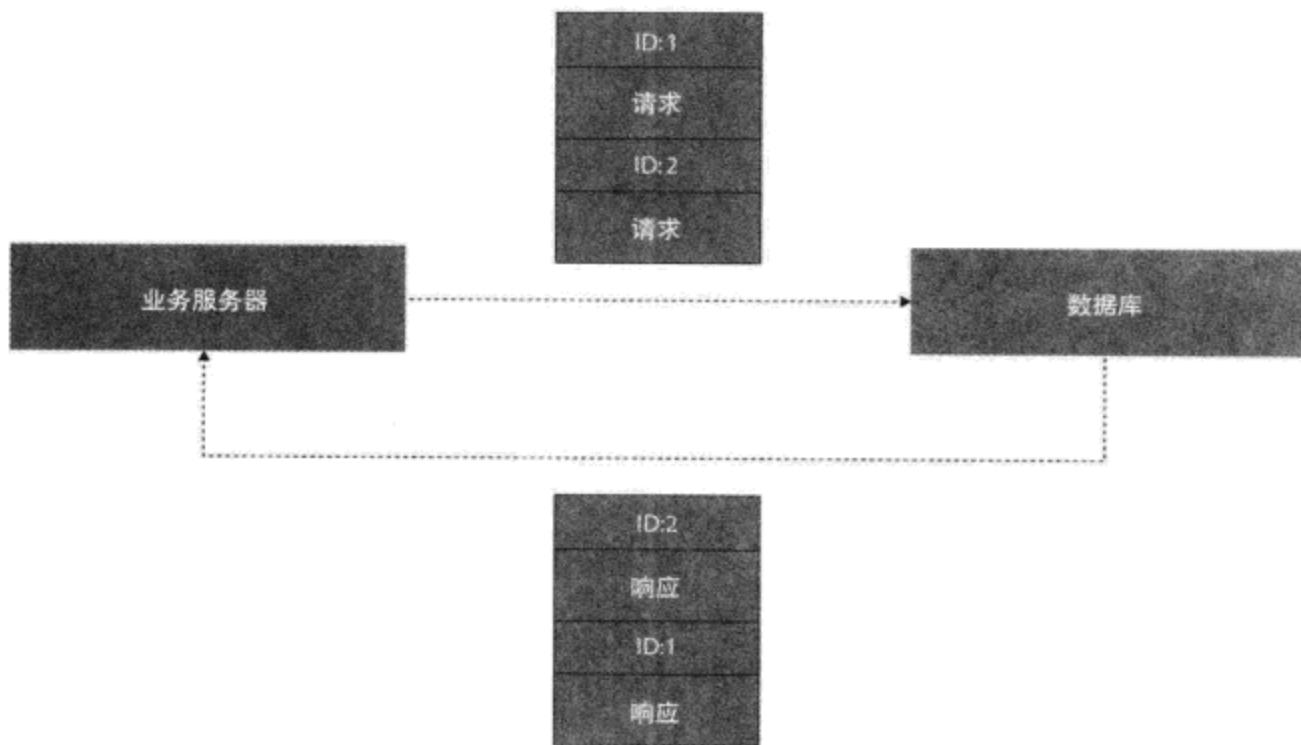


图 7.9 异步数据库访问

目前有不少数据库都有了异步访问的客户端<sup>15</sup>，在 JBoss Netty 的测试报告<sup>16</sup>中，异步的数据库访问方式较之传统的阻塞方式的数据库访问带来了极大的性能提升。

#### 4. DAL (Data Access Layer)

DAL 是指系统直接提交给数据库的操作转变为通过 DAL 来进行提交，在这种情况下，可借助 DAL 来降低数据库连接的使用，其系统结构如图 7.10 所示：

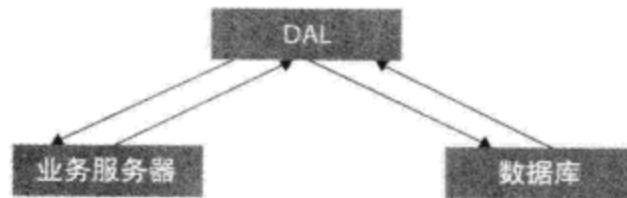


图 7.10 引入 DAL 后的系统结构

在采用这种方式后，无论业务服务器如何水平伸缩，数据库连接都可在 DAL 上统一控制，但从图中也可看出，这种方式的问题是增加了一个中间层，会导致性能有一定的下降，其稳定性也会受到影响。目前 DAL 开源的有 Amoeba<sup>17</sup>，DAL 方式的另一个好处是可透明化分库、分表对于业务服务器带来的影响。

由于所有的操作都经过 DAL，无论 DAL 是直接内嵌到业务服务器应用的 jar，还是作为中间层，它都有机会根据分库分表规则改变业务请求的 sql 及要连接的目标数据库，这也是在使用 DAL 后能够透明化分库、分表的原因。

<sup>15</sup> <http://code.google.com/p/adbcl/>

<sup>16</sup> <http://www.jboss.org/netty/performance/20091125-mheath.html>

<sup>17</sup> <http://amoeba.meidusa.com.wordpress/>

DAL 为了能够改变业务请求的 sql，通常要对请求的 sql 进行解析或提供一种专门的 sql 语言，以便能够匹配分表的规则，例如 `select username from user where userID=1220`，如 `user` 表的分表规则为 `userID`，那么 DAL 就必须提取出这段 sql 中的 `userID=1220` 的信息，并用 `1220` 这个值放入分表规则中计算，从而得到表名并改写 sql。对于提供专门的 sql 语言的方式而言，好处是无须解析 sql，可以更简单地获取分表规则所需要的信息。但会带来的问题是开发的易用性降低及提升了 DBA 审核 sql 的难度，根据同样的方式可得到 `user` 应连的目标数据库，从而提交给相应的目标数据库执行，在这样的方式下，借助 DAL 就可避免分库、分表对应用产生影响，DAL 的执行过程如图 7.11 所示：

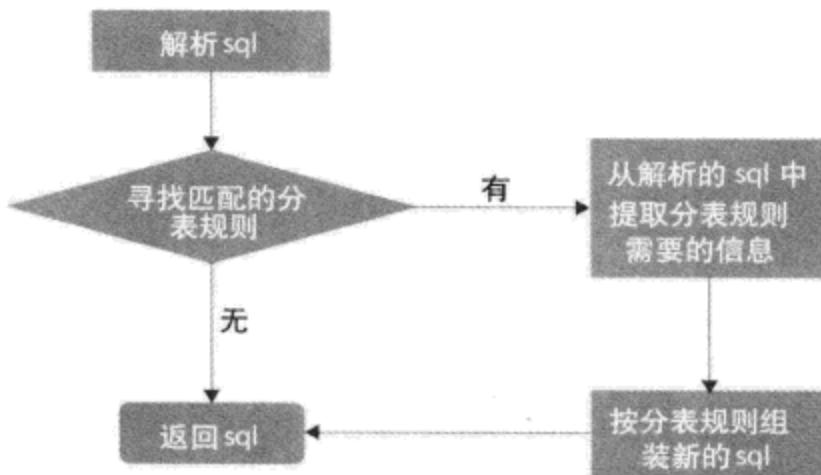


图 7.11 DAL 的执行过程

从上可见，在依靠水平伸缩来支撑高访问量时，除须考虑应用本身是否可水平伸缩，还须考虑伸缩后对后端带来的压力，只有在同时解决了这两方面的问题后，才可依靠水平伸缩来支撑高访问量。

## 7.2.2 支撑大数据量

数据量增长后带来的问题主要是读写性能的下降。从数据库使用角度来看，可采用一些方法借助机器的增加来提升数据读写的性能，主要有读写分离和多 master 这两种方式。

### 读写分离

读写分离采用的方法为当写数据库时在一个数据库上写入，而要读取时则从多个其他的数据库中读取，通常将用于写入的库称为 master 库，用于读取的库称为 slave 库。

多数数据库均提供了机制用于实现读写分离，例如 mysql replication、oracle standby 等，mysql replication 支持对称复制和非对称复制两种方式。

对称复制是指从 master 库复制数据库到所有 slave 库，slave 库的数据和 master 库的数据保持一致。对称复制的优点是各 slave 库均一致，系统在读取时可以任意挑选其中一个库读取，并且即使其中有几个 slave 库出现问题，也不会影响业务。其缺点是可能会造成每个 slave 上的数据量都非常大，从而使 slave 库的硬件配置也要非常好，并且每次要从 master 复制到所有的 slave，随着 slave 机器增加，延时现象可能会越来越严重。

非对称复制是指从 master 库复制部分数据到 slave 库，各 slave 库的数据可能不同，其优点在于各个 slave 库仅持有部分数据，可提升其读取数据的响应速度，并且每次 master 写入时无须复制到所有的 slave，可以尽可能地减少延时现象；还可根据业务的不同为不同的 slave 库配置不同的索引，从而保证写入的速度；缺点在于一旦 slave 机器出现故障，就会导致一些数据要回到 master 读取，同时由于每个 slave 的数据不同，使业务服务器在访问的时候较为复杂，和分库后带来的挑战类似。

在采用读写分离后，系统的结构如图 7.12 所示：

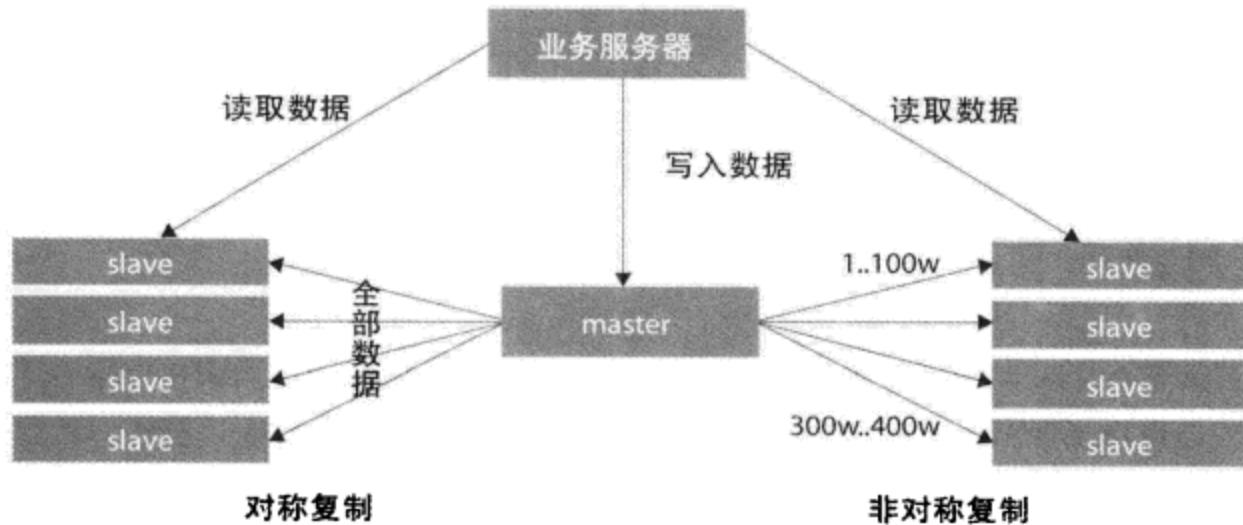


图 7.12 读写分离后的系统结构

对于异构数据库之间的读写分离，例如写入的为 oracle，读取的为 mysql，这种情况下通常只能自行实现，方案可以采用类似 eBay 实现实时搜索的方式，即写入数据库的同时，写入一个消息中间件，由消息中间件通知相应的订阅者处理。

读写分离适用于读多写少，并允许一定延时的业务中，对于读写比例基本相等的业务而言，如采用读写分离反而会带来大幅度复制，造成系统运行更缓慢。

在读写分离实现后，系统可通过水平伸缩来降低数据库读的压力和提升数据库读的响应速度。

## 多 master

为提升数据写的速度，通常可采用的方法是建立多个 master，在建立多个 master 时最理想的状态是多个 master 没有关联，也就是多个 master 的数据并不相同。例如按用户将其所拥有的物品划分到一个单独的 master 机器去写，这样当用户需要操作时只要操作所对应的 master 机器即可，这种仅适用于读取数据时无须联合读取的情况；另外一种多 master 的状况则为每个 master 的数据要保持一致，此时一个明显的挑战是数据一致性的问题，另外一个明显的挑战是自增 ID 的问题。

数据一致性的问题通常采用复制、两阶段提交、三阶段提交或 google paxos 来解决，复制要求数据具备版本信息，而对于有些场景而言，必须做到同步的一致性。例如用户的存款余额，在 master A 数据库修改的同时，在其他的 master 数据库也必须同时修改成功，对于此类现象，需要借助两阶段提交、三阶段提交或 google paxos 来解决，这些技术在第 6 章“构建高可用系统”中会有提及：自增 ID 的问题通常要改为由程序来生成 ID 的方式解决。

在解决了以上问题后，即可通过水平伸缩来提升数据库的写速度。

多个 master 系统的结构通常如图 7.13 所示：



图 7.13 多 master 的系统结构

### 7.2.3 提升计算能力

在单机计算的情况下，即使进行垂直伸缩，PC Server 的计算能力也是无法和小型机抗衡的，因此要有合理的设计来通过增加 PC Server 提升系统的计算能力。在垂直伸缩场景中采取的方法是将计算任务拆分，多线程并行计算然后汇总结果。在水平伸缩场景中则可演变为将计算任务拆分，然后分派给不同的机器进行计算，最后汇总。它复杂的地方在于拆分的方法、拆分后的分派调度及机器执行失败时的处理，在 Java 中主要可采用的方法有 MapReduce<sup>18</sup> 和 MPI<sup>19</sup>，MapReduce 及 MPI 的方式都支持将任务拆分后分解到多台机器上执行。在计算任务不变的情况下，增加机器可使得每台机器上执行的计算任务减少，从而提高速度；倘若计算任务增加，增加机器可让每台机器需要执行的计算任务不变，从而保持系统的计算能力。

<sup>18</sup> <http://labs.google.com/papers/mapreduce-osdi04.pdf>

<sup>19</sup> [http://en.wikipedia.org/wiki/Message\\_Passing\\_Interface](http://en.wikipedia.org/wiki/Message_Passing_Interface)

# 索引

## A

AIO, 2, 3  
ArrayBlockingQueue, 138, 149, 150, 155, 157, 164, 206  
ArrayList, 83-85, 90, 112-118, 124-138, 145-149, 179, 193, 196, 198, 223, 225, 240, 242  
AtomicInteger, 138, 151, 153, 155, 208, 217

## B

BIO, 2-5, 7-10, 14, 112  
本地方法栈, 63, 64  
避免 Survivor 区过小或过大, 195  
避免新生代设置过大, 194  
标记-清除 (Mark-Sweep), 66, 67  
标记-压缩 (Mark-Compact), 66, 68  
标量替换, 56

## C

ClassCastException, 47, 49  
ClassNotFoundException, 48, 49, 170  
client compiler, 53  
CompileThreshold, 58, 202  
ConcurrentHashMap, 122, 139, 140-145, 159, 217, 221  
Condition, 105, 138, 149, 150, 163, 164  
CopyOnWriteArrayList, 145-149, 163, 222  
CopyOnWriteArraySet, 149  
CountDownLatch, 105, 151, 152, 156, 157, 162, 163, 165, 205, 206, 210, 211, 215, 218, 224  
cpu sy, 204  
cpu user, 75  
CyclicBarrier, 156, 163, 165, 218  
串行 GC (Serial GC), 69

垂直伸缩, 252, 254, 266

## D

DAL, 254, 263, 264  
DAS, 258  
Direct Routing, 232, 233  
调用计数器, 57, 58  
堆, 13, 56, 63-66, 68, 70, 74, 84, 88, 89, 97-99, 101, 106, 108, 178, 182, 186, 187, 189, 190, 205, 209, 253, 258

## E

Eclipse Memory Analyzer, 98, 99  
ESB, 25, 29, 30, 34, 36

## F

Executors, 11, 155, 157, 205, 209, 244  
FutureTask, 158-160, 162  
反射执行, 59, 60, 61  
反序列化, 11, 13, 14, 112, 167, 169, 171, 172, 187  
方法内联, 53, 54  
方法区, 63, 66  
分库, 247, 262-265  
复制 (Copying), 66, 67

## G

Garbage First, 88  
GFS, 259, 260  
Gossip, 235, 236  
跟踪收集器, 66, 68  
功能降级, 249

**H**

HashMap, 62, 119-123, 125-145, 159, 165, 211, 221, 242  
 HashSet, 112, 119, 120, 125-138  
 HDFS, 259  
 合理设置新生代存活周期, 195  
 回边计数器, 58

**I**

iostat, 183, 184, 185  
 IP Tunneling, 232, 233

**J**

Java 源码编译机制, 40, 41  
 jmap, 96-99, 187  
 JSR 269, 41  
 jstat, 72, 74, 75, 99, 187, 190, 194-197, 199, 201, 202, 210  
 JVisualVM, 94-96  
 基于栈的体系结构, 51  
 旧生代, 64, 68, 69, 71-87, 93, 97, 99, 192, 194, 195, 197, 201, 202

**K**

Keepalived, 233, 234, 236  
 Kilim, 205-207

**L**

LinkageError, 48, 49  
 LinkedList, 112, 116, 117, 124-138, 161  
 类加载机制, 44  
 利用率, 80, 88, 176-178, 259  
 两阶段提交, 237, 265

**M**

MapReduce, 266  
 Michael-Scott 非阻塞队列算法, 219  
 Mina, 10-13  
 MPI, 266  
 Mule, 34-36

**N**

NAS, 258, 259  
 NAT, 231  
 NIO, 2, 3, 5-10, 12, 13, 15, 24, 112  
 NoClassDefFoundError, 45, 48, 49

**O**

OnStackReplacePercentage, 58

**P**

Paxos, 237, 238

**Q**

去虚拟化, 53, 54

**R**

Real-Time JDK, 91, 92  
 ReentrantLock, 139, 145, 146, 150, 159, 163-165, 167, 216, 221  
 ReentrantReadWriteLock, 165  
 RMI, 3, 13-15, 17, 19, 21, 24  
 冗余削除, 53-56  
 软引用, 69, 70  
 弱引用, 69, 70

**S**

SAN, 258, 259  
 SCA, 25-30, 33, 34, 36  
 Semaphore, 161, 162  
 server compiler, 53, 55  
 SNA, 255  
 SOA, 23-26, 29, 30, 33, 34, 36, 37  
 Spring RMI, 13, 17, 18  
 SRM, 10  
 Stack, 44, 51, 57, 78, 91, 112, 118, 119, 124-138, 217-219  
 switch-threading, 52, 53  
 三阶段提交, 237, 265  
 上下文切换, 175, 178, 180, 182, 205  
 水平伸缩, 228, 235, 236, 249, 250, 252, 254-256, 258-266  
 随机(Random)选择, 229

**T**

TDA, 108  
 ThreadPoolExecutor, 153-155, 157, 162, 163, 207, 244  
 TLAB, 65, 66  
 token-threading, 52, 53  
 TreeMap, 120, 122-138  
 TreeSet, 112, 120, 125-138  
 Tuscany, 30-34, 36

同步削除, 56

## V

Vector, 112, 117-119, 124-138

vmstat, 178, 182, 188

volatile, 102-104

VRRP, 233

## W

Webservice, 3, 13, 16, 17, 19-21, 24, 27, 28, 31, 33-36

## X

线程交互机制, 104

线程资源同步机制, 100

协程, 204, 205, 207

新生代, 64, 65, 68, 69, 72-74, 76, 78, 81, 82, 84, 86, 87, 93, 97, 192, 194, 195, 197, 201, 202

虚引用, 69, 70

序列化, 11, 13, 14, 112, 167-172, 187, 244

## Y

一致性 hash, 256, 257

引用计数收集器, 66

运行队列, 175, 178, 205

## Z

栈上分配, 56, 92

自动修复, 248

# 已是悬崖百丈冰，犹有花枝俏

——美编寄语

当初周老师提出“梅、兰、竹、菊”系列封面构想的时候，一语中我心田。

中国古代的梅、兰、竹、菊并称为“花草四君子”，中国古代绘画，特别是花鸟画中，有相当多的作品是以它们为题材的，它们常被文人高士用来表现清高拔俗的情趣：正直的气节、虚心的品质和纯洁的思想感情，因此，方有“君子”之称。我乐于做这样的设计，不仅仅是宣扬国画艺术，更多的在于提炼中国文化的精神。

我们为林昊的书选择了梅花为素材。寒冬腊月，梅花迎着刺骨的寒风、漫天的大雪怒放。越是风欺雪压，梅花开得愈精神愈美丽。人的一生不会总是一帆风顺，逆境中尤其要具备梅花的这种精神。

红岩上红梅开，

千里冰霜脚下踩。

三九严寒何所惧，

一片丹心向阳开。

杨小勤

2010年5月于武汉

# 过去了是快乐，过不去是折磨

—— 编辑手记

作者写一本书可谓历尽千辛万苦，一般都要占用他们全部的节假日和休息时间。而为了应对编辑善意而客气的催稿，往往还要经常写到凌晨一两点、两三点。所以每当收到一部作者送来的书稿时，我们总是难免心中充满歉疚和感谢！然后心里想的只有一件事，如何尽力把这本书出好，以不愧对作者的辛劳和信任，不愧对读者的期待与渴望。

怎样才能问心无愧呢？只有争取在书稿中贡献编辑一点微薄之力，能给书稿作一点添砖加瓦的工作，使书稿尽量完善，以不负作者的托付。编辑与作者比，只是接触过的书稿多一些，对完善书稿多少有些心得，可以与作者起一点互补的作用。

为了充分体现编辑的作用，这次我们采取了集体讨论的方式，分头阅稿，集中意见。在讨论中我们在结构上提出了调整某些章节篇幅不平衡的问题。在层次上提出了如何使书稿眉目更清楚的问题……总之我们尽力想做到不要像某些作者批评编辑的：如果只改改错别字，改改标点符号，要你们编辑做什么？

当然，每一条修改建议都会在作者好不容易写完了书稿的时候又给他增加了重重的负担。我们有幸得到了作者的充分理解和支持。他不厌其烦地配合我们进行了充分的讨论，共同对书稿作出修改。

经过一个多月的共同努力，书稿的编辑加工终于完成。如果作者认为我们的参与对书稿的完善起了一些有益的作用。我们将感到十分欣慰。

文字编辑 郑兆昭

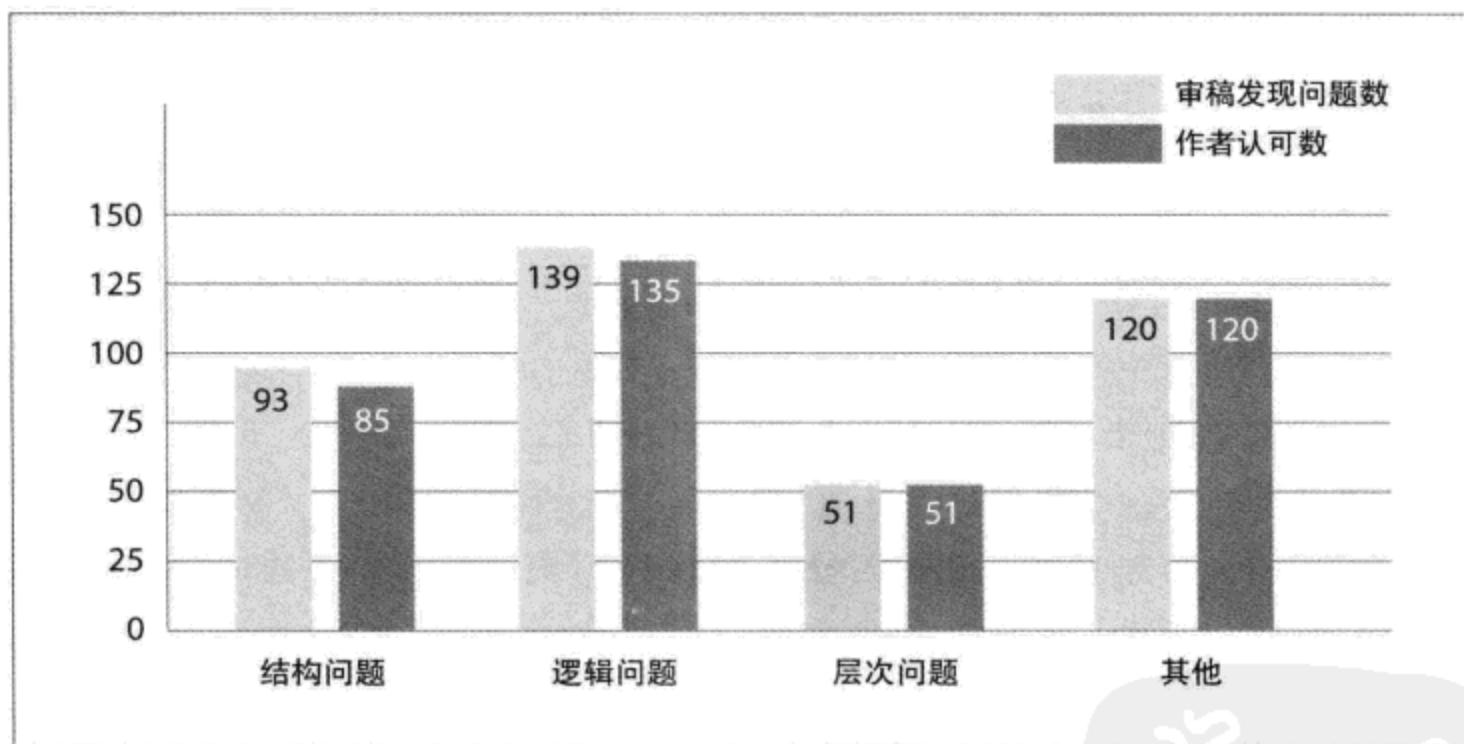
这是一本让我“难受”的书。田间管理阶段，畏难情绪差点让我止步不前，看着同事都在进步，自己却处于停滞状态，其实是很难受的，这时周老师不断为大家鼓劲，推着我们向前走，最终才坚持了下来。编加阶段中，我又一次看到了自己的问题所在：不够敏捷，没有将工作模块化。这些问题差点儿导致本书无法如期付印，还好在最后关头，周老师出面亲自协助，终于按时收尾。

回想这一路下来，其实学到了不少东西，比如田间管理中的坚持，使我知道，很多时候，很多事情，非不能也，是不为也。编加阶段，周老师的指导让我明白，遇事儿要列细致的计划，要把任务分块，不要太相信记忆，写下来才是正理，特别是事儿多的时候，清晰的计划可以为你披荆斩棘。

责任编辑 杨绣国

刚接到本书的审稿任务时，我畏难了！那么多的专业术语，那么多的知识点瞬间向我轰炸而来，我慌了手脚，忘记了以前审编人文的书稿的经验，而且习惯性地想要退缩：我就看文字算了，技术方面让懂技术的人去看吧。周老师果断地拦住了我的退缩。在她强有力地指导和督促下，我硬着头皮上了。

静下心来一点一点地读书稿，我发现技术书稿的审阅也是有规律可循的。审稿的重点在于结构问题、逻辑问题、层次问题，而小组成员在集体审稿时，有自己比较关注的地方和思维的盲点，可以做到优势互补。如郑老师比较关注规范化问题，根据出版规范提出了一些建议；卢鹤翔比较关注行文的严谨，论点的可信度；徐定翔长于归纳，能够发现标题与内容不相符的问题；杨绣国比较细致，发现了多处语句逻辑问题；而我则比较关注结构问题，提出了多处章节结构上调整的建议。而我们提出的修改建议绝大多数都得到了作者的认可。



《分布式 Java 应用：基础与实践》带着清新的梅香即将上市，回顾这本书的审稿过程，我最想说的话就是：不要自我设限，战胜自我，你会获得更广阔的天空。感谢作者的坚持和宽容，感谢审稿的小组成员们。

项目编辑 白爱萍

田间管理小组成立之初，曾是程序员的我向周老师建议，郑老师、杨绣国、白爱萍三位没有技术背景的编辑主审文字，技术审稿则由徐定翔和我共同负责。既是自信，也是对同事们的低估。然而周老师告诉大家：不克服困难，能力难有提高！

事实证明，我错了。随着审稿工作的进行，书稿中的瑕疵经过他们认真细致的阅读被发现出来，并且发现的错误常常多于我能看到的。而我这个新编辑则从大家的讨论中收获更多。

遗憾的是我没有参加这本书最后几章的讨论，相比其他几位成员，我的付出很少，感谢各位同事的努力。

技术编辑 卢鹤翔

第一次见到林昊是 2008 年 5 月 24 日，在杭州举办的第二届中国网络工程师侠客行大会上。那时他到淘宝担任架构师不到一年，但是已经参与淘宝分布式服务平台的建设，帮助淘宝扩展分布式应用和采用低成本服务器集群。此外他还担任 OSGi 中国用户组的负责人，利用业余时间在国内推广 OSGi，在 Java 技术圈内很有影响力。会议休息间隙我找到他，邀请他写书分享 Java 应用的经验。巧的是他正在酝酿写书介绍分布式 Java 应用的知识体系，于是我们达成了初步的合作意向。

回到武汉后不久，我收到林昊发来的目录大纲。目录大纲是用思维导图制作的，内容覆盖面很广，从介绍基本的 Java 网络编程知识到如何充分利用硬件资源，无所不包。我们不敢怠慢，马上邀请包括李锟、王翔等多位专家针对目录提建议。专家普遍认为目录很全面，如果能按预想完成，内容会非常扎实。但是我作为编辑，看着这份复杂的目录，却有些不知所措：一是抓不住书稿的主体脉络；二是担心书稿的内容太多、篇幅太长。在田间管理的最初阶段，这两个问题一直没有解决，所以我的工作变得被动，对作者的支持不多，心里很着急。这时周老师提醒我，并给我指出了解决办法：首先要弄清读者定位。只有搞明白这本书是写给哪些读者看的，要解决读者的哪些问题，才能抓住内容的关键点，才能有针对性地帮助作者调整内容，哪些部分该详写，哪些该略；其次，一定要避免犯闭门造车，应该多与专家沟通，多收集建议，帮助作者减少盲点。我按照周老师的建议，调整了工作方式，果然收到了效果。重新与林昊讨论图书的定位后，我们把目标读者定位为具有一定工作经验的 Java 程序员。同时邀请郑晖老师、霍炬，还有武汉大学的同学曹祺、刘力祥试读初稿。专家和同学针对书稿的内容都给予了认真的反馈，尤其是郑晖老师，充分肯定了林昊这本书的实践意义，对技术细节提出了许多详细、中肯的建议。他们的帮助，让编辑和作者都信心大振。

林昊提交定稿后，周老师又安排白老师带领“田间管理”小组（白爱萍，郑兆昭、杨绣国、卢鹤翔）集中审阅书稿。大家先分头提出修改建议，然后碰头逐条讨论分析，去粗取精，同时完善留下来的建议。最后提交给作者的建议有 90% 获得了认可，从调整书稿的结构到完善局部细节；从减少正文中的代码篇幅，节省纸张，到对关键知识点进行必要的补充说明，降低阅读难度，等等。这个过程体现了集体合作的力量，进一步加深了我个人对书稿的理解，感谢同事们的帮助。此外，还要感谢周老

师不断鼓励我们把工作做细致，做周全。感谢设计部的同事为本书制作精美的插图和封面。

为了完成这本书，林昊牺牲了个人的业余休息时间，牺牲了陪伴家人的时间，甚至“狠心”地把装修新家的任务甩给了未婚妻。但是从这本书里，我看到了林昊的努力、坚持，以及对软件行业、对工作、对生活的热情。期待林昊为读者带来更多的好作品。

策划编辑 徐定翔

2010年5月于武汉

