Broadview®
www.broadview.com.cn 博文视点

分布式Java应用

基础与实践

林昊 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

分布式Java应用

基础与实践

林昊 著

電子工業出版社
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，既可作供初学者学习，也可供同行参考。

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 7 0 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，而这也就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，顺利完成了第一章和第三章的撰写。

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二章、第四章和第五章，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。待完成了第六章的编写后，由于剩下的第七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，就是先放下第七章，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 章定稿除了第三章以外的章节，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，在第三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

目 录

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

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。

第 3 章　深入理解 JVM

……

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。

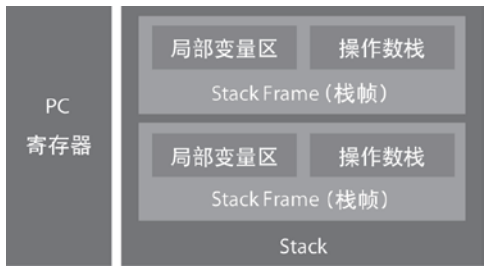


图 3.5　Sun JDK 基于栈的体系结构

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[1]。

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();
  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

1 *http://www.complang.tuwien.ac.at/forth/threaded-code.html*

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[2]，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 [3]，在其他方面的优化主要有：方法内联、去虚拟化、冗余削除等。

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 [4]，那么，会演变成类似这样的结构 [5]：

2　*http://www.bluedavy.com/book/reference/c1.pdf*

3　*http://www.bluedavy.com/book/reference/LSRA.pdf*

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

5　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 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{

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. 冗余削除

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 isDebug=log.isDebugEnabled();
public void execute(){
```

```
  if(isDebug){
      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[6]，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 [7]；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。

逃逸分析 [8]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);
```

6　*http://www.bluedavy.com/book/reference/c2.pdf*

7　*http://www.bluedavy.com/book/reference/GCRA.pdf*

8　关于逃逸分析可参见：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。

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：

```
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）[9]。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 中可能会引入多层编译的支持。多层编译是指在

9 *http://portal.acm.org/citation.cfm?id=776288*

-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 [10]。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 CompileThreshold * (OnStackReplacePercentage/100)，在 server 模式时，计算规则为(CompileThreshold * (OnStackReplacePercentage – 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。

10　*http://blogs.oracle.com/ohrstrom/2009/05/pulling_a_machine_code_rabbit.html*

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	Method (id=24)
annotationDefault	null
annotations	null
clazz	Class<T> (java.lang.Object) (id=9)
declaredAnnotations	null
exceptionTypes	Class<T>[0] (id=29)
genericInfo	null
methodAccessor	GeneratedMethodAccessor1 (id=37)
modifiers	1
name	"toString" (id=31)
override	false
parameterAnnotations	null
parameterTypes	Class<T>[0] (id=32)
returnType	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。

……

3.2.3 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。

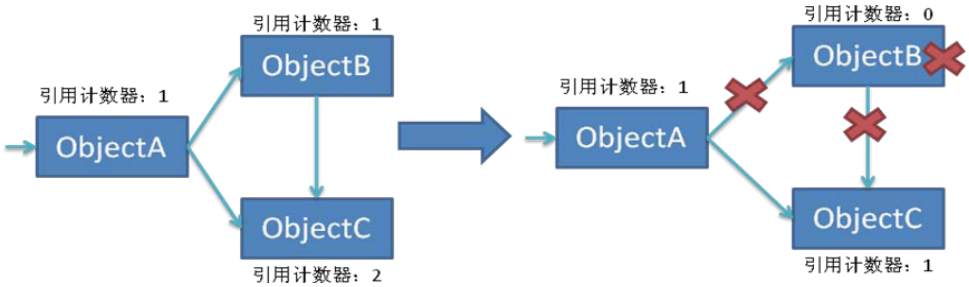


图 3.9　引用计数收集器

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的空间中，如图 3.10 所示。

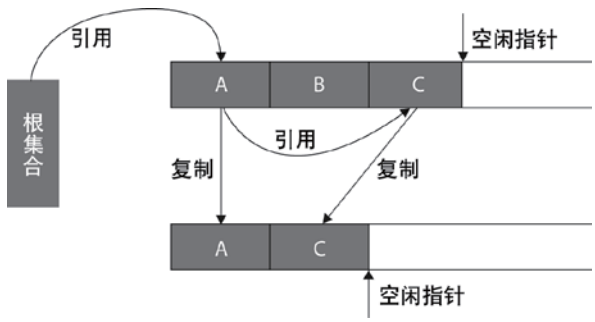


图 3.10　复制算法

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。

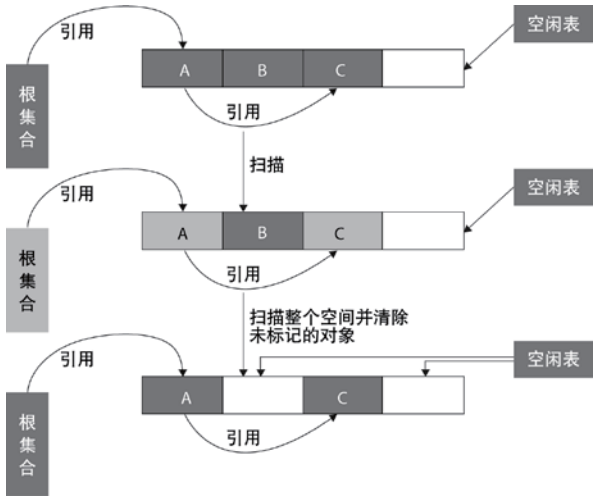


图 3.11 标记-清除算法

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。

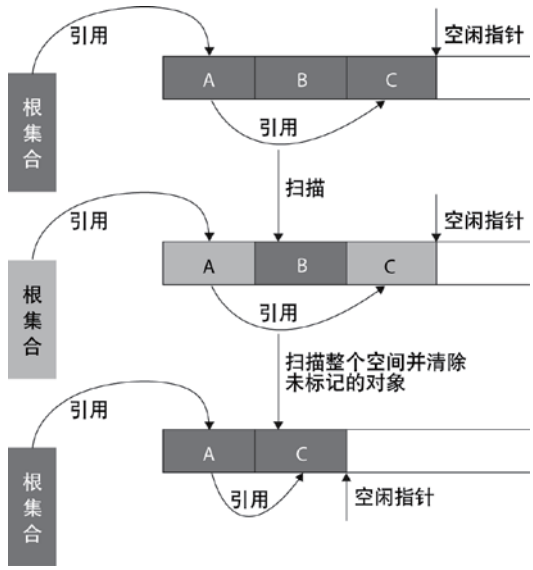


图 3.12　标记-压缩算法

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。

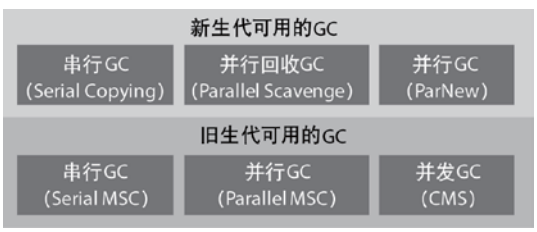


图 3.13　Sun JDK 中可用的 GC 方式

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 [11]。

- 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。

11 *http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html*

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 [12]。

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;
```

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 [13]。

当扫描存活的对象时，Minor GC 所做的动作为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，

12　*http://forums.sun.com/thread.jspa?threadID=5221725*

13　*http://www.pawlan.com/monica/articles/refobjs/*

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。

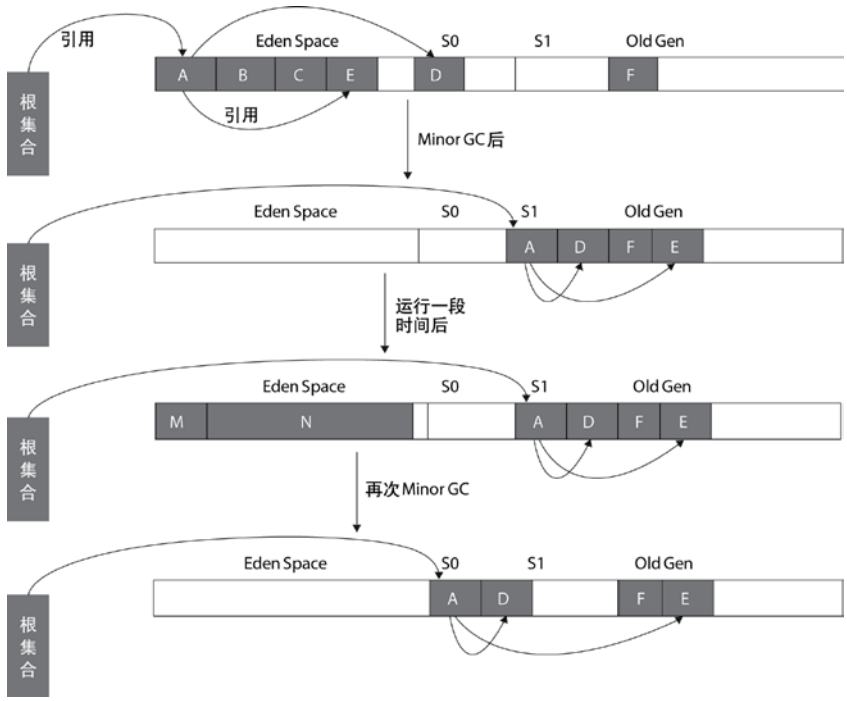


图 3.14 Minor GC 过程

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 [14]，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。

14 *http://forums.sun.com/thread.jspa?threadID=778029*

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+（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 [15]，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。

15　*http://blogs.sun.com/jonthecollector/entry/our_collectors*

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 happenMinorGCIndex) throws Exception{
            for(int i=0;i<happenMinorGCIndex;i++){
                if(i==happenMinorGCIndex-1){
                    Thread.sleep(2000);
                    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，对象直接进入旧生代的示例

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生代中，这就可以解释为什么在 minor GC 执行后旧生代空间被占用了 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。

……

Garbage First[16]

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

16　*http://research.sun.com/jtech/pubs/04-g1-paper-ismm.pdf*

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)*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*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 remebered 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 yound 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:+UnlockExperimentalVMOptions -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]
```

……

第 4 章　分布式 Java 应用与 Sun JDK 类库

……

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 voliate 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 voliate 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于 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 voliate 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。

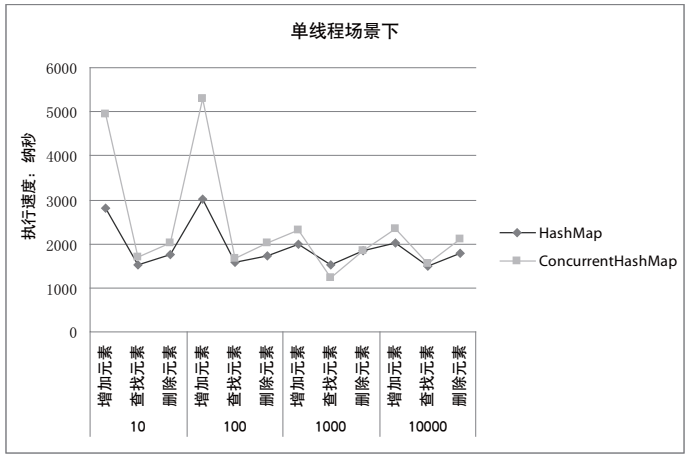


图 4.17　单线程场景下 ConcurrentHashMap 和 HashMap 运行结果的对比

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：

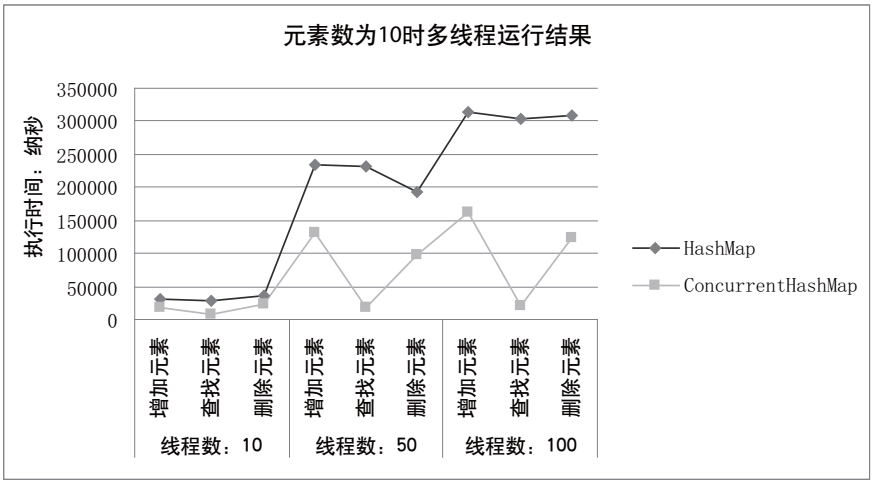


图 4.18 多线程场景，元素数为 10 时 ConcurrentHashMap 和 HashMap 运行结果的对比

当元素数量为 100 时，和 HashMap 的对比结果如图 4.19 所示：

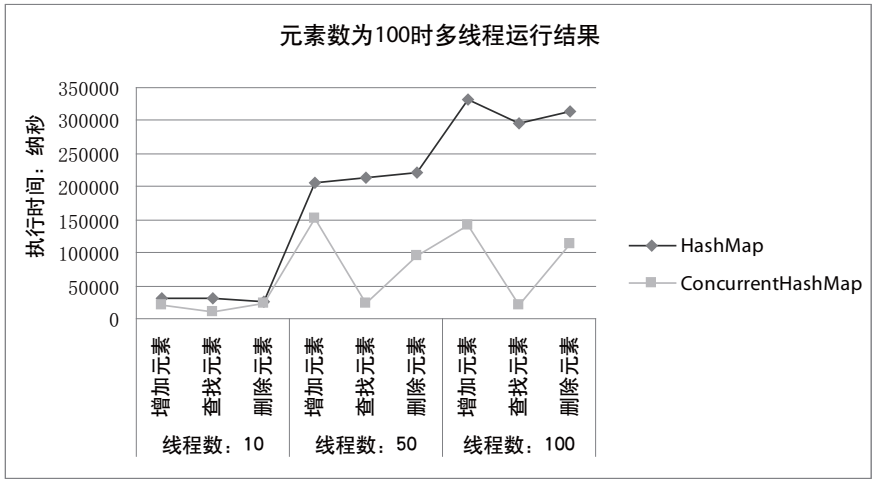


图 4.19　多线程场景，元素数为 100 时 ConcurrentHashMap 和 HashMap 运行结果的对比

当元素数量为 1000 时，和 HashMap 的对比结果如图 4.20 所示：

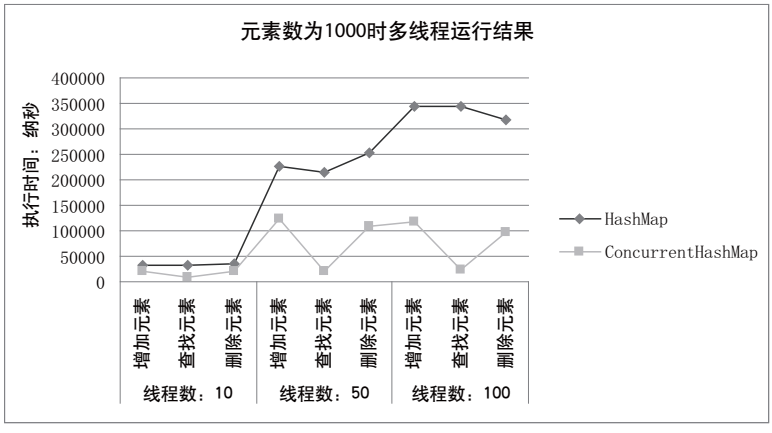


图 4.20　多线程场景，元素数为 1000 时 ConcurrentHashMap 和 HashMap 运行结果的对比

当元素数量为 10 000 时，和 HashMap 的对比结果如图 4.21 所示：

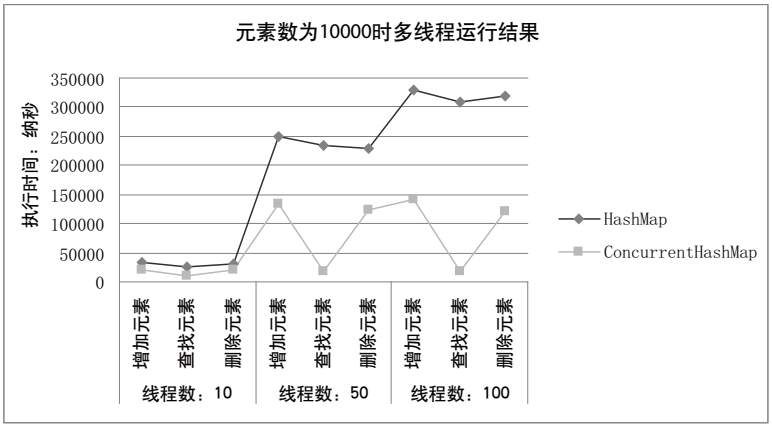


图 4.21 多线程场景，元素数为 10000 时 ConcurrentHashMap 和 HashMap 运行结果的对比

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。

……

4.2.6　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.defaultThreadFac
tory(),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 e1) {
          e1.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：用地
              "+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。

……

4.2.8　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)

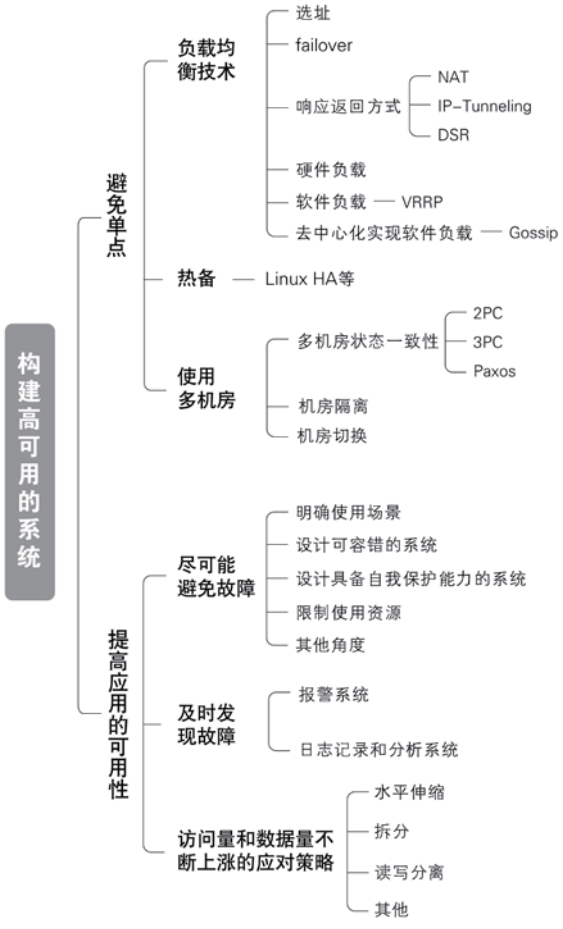
调用 Sync 的 innerCancel 方法实现。

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。

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 7×24 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%×365×24×60	525.6
99.99%	0.01%×365×24×60	52.56
99.999%	0.001%×365×24×60	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。

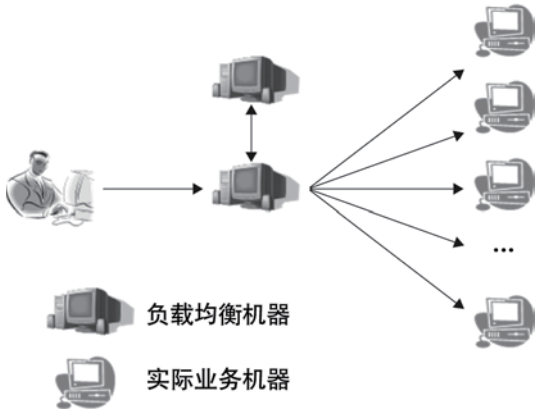


图 6.1 采用负载均衡后的系统结构

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用户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，但会带来的问题是一旦机器出现故障，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[1]来解决。

Twitter的工程师在分享unicorn带来的作用时 [2]，首先对原有方式做了个比喻，原来的方式就像是超

1　*http://unicorn.bogomips.org/*

2　*http://engineering.twitter.com/2010/03/unicorn-power.html*

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lighthttpd上也有一个支持这种方式的均衡算法：Shortest Queue First[3]。

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：

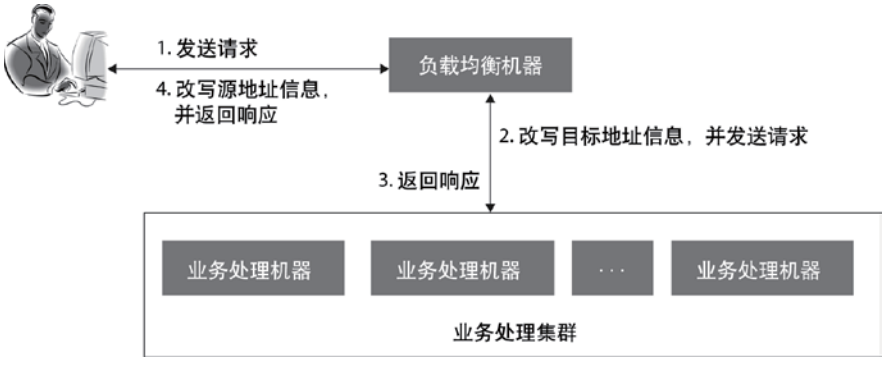


图 6.2 NAT 方式时的响应返回

3 *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：

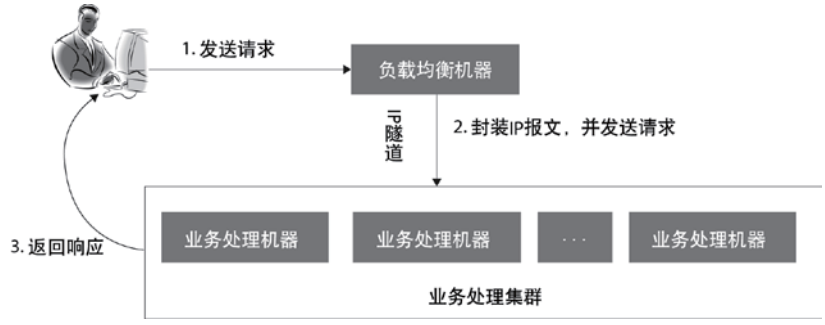


图 6.3　IP Tunneling 方式时的响应返回

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：

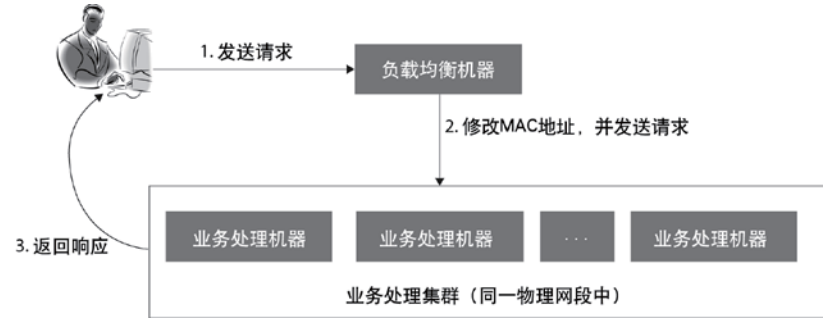


图 6.4　DSR 方式时的响应返回

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）[4]，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[5]基于VRRP（Virtual Router Redundancy Protocol）协议[6]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内都收不到

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*

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[7]，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，还有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，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

7　*http://www.linux-ha.org*

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[8]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：

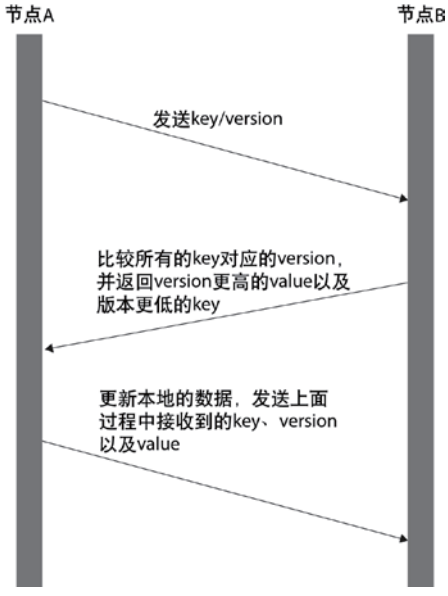


图 6.5 Gossip 模型

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

8 *http://www.cs.cornell.edu/home/rvr/papers/flowgossip.pdf*

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）[9]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）[10]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。

9 *http://en.wikipedia.org/wiki/Two-phase_commit_protocol*

10 *http://en.wikipedia.org/wiki/Three-phase_commit_protocol*

4. 基于Paxos[11]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[12]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[13]。在实现一致性方案时，通常都遵循CAP理论[14]，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核心应用，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，

11　*http://labs.google.com/papers/paxos_made_live.html*

12　*http://research.yahoo.com/files/pnuts.pdf*

13　*http://snarfed.org/space/transactions_across_datacenters_io.html*

14　*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。以上场景自然是以第二种方法优先。

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	
风险所造成的后果	
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力下的可用性。

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 [15]。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。

15 *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阀值时，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[16]，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

16 *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[17]。这要求系统以功能为粒度进行管理，并制定相应的功能级别，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[18]。

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。

17 *http://www.bluedavy.com/iarch/ebay/ebay_arch_principles.pdf*

18 *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。

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

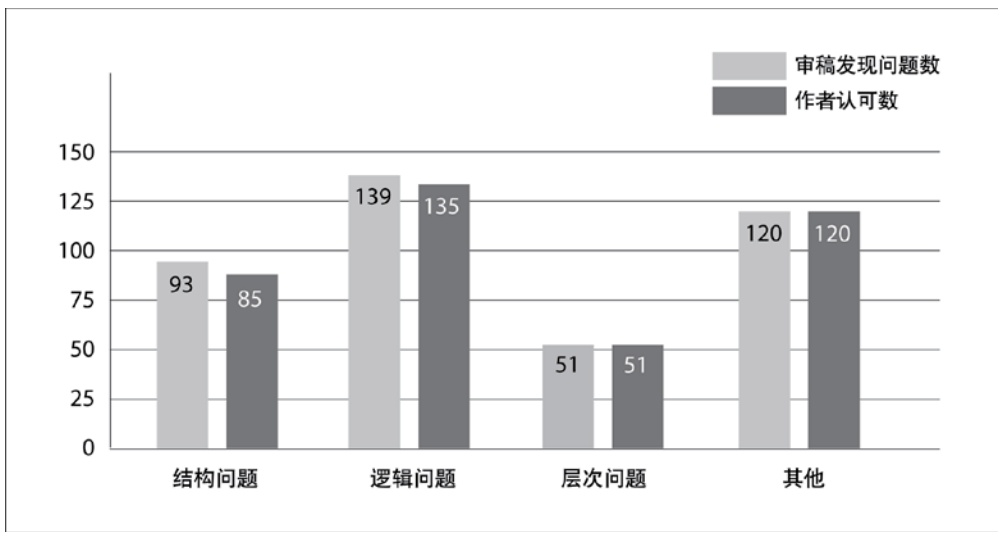
这是一本让我“难受”的书。田间管理阶段，畏难情绪差点让我止步不前，看着同事都在进步，自己却处于停滞状态，其实是很难受的，这时周老师不断为大家鼓劲，推着我们向前走，最终才坚持了下来。编加阶段中，我又一次看到了自己的问题所在：不够敏捷，没有将工作模块化。这些问题差点儿导致本书无法如期付印，还好在最后关头，周老师出面亲自协助，终于按时收尾。

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。在她强有力的指导和督促下，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；卢鹄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 卢鹄翔

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、卢鹄翔）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

北京博文视点（www.broadview.com.cn）资讯有限公司成立于2003年，是工业和信息化部直属的中央一级科技与教育出版社——电子工业出版社（PHEI）下属旗舰级子公司，在六年的开拓、探索和成长中，已成为中国颇具影响力的专业IT图书策划和服务提供商。

六年来，博文视点以开发IT类图书选题为主业，励精图治、兢兢业业，打造了一支团结一心的专业队伍，并形成了自身独特的竞争优势。一直以来，博文视点始终以传播完美知识为己任，用诚挚之心奉献精品佳作，年组织策划图书达300个品种，同时开展相关信息和知识增值服务，赢得了众多才华横溢的作者朋友和肝胆相照的合伙伙伴，已经成为IT图书领域的高端品牌。

我们的理念：创新专业图书服务体制；培养职业策划图书服务队伍；打造精品图书品牌；完善全面出版服务平台。

我们的目标：面向IT专业人员的出版物提供相关服务。

我们的团队：一个整合了专业技术人员和专业服务人员的团队；一个充满创新意识和创作激情的团队；一个不断进取、追求卓越的团队。

我们的服务：善待作者　尊重作者　提升作者

我们的实力：优秀的专业编辑队伍
全方位立体化的强大的市场推广平台
实力雄厚的电子工业出版社的渠道平台

“走出软件作坊独辟蹊经　人道编程之美，
追踪加密解密庖丁解牛　精雕夜读天书。”

路漫漫其修远，博文视点愿与所有曾经帮助、关心过我们的朋友、作者、合作伙伴携手奋斗。未来之路，不可限量！

地址：北京市万寿路173信箱电子工业出版社博文视点资讯有限公司
邮编：100036　**总机：**010-88254356　**传真：**010-88254356-802

武汉分部地址：武汉市洪山区吴家湾湖北信息产业科技大厦1402室
邮编：430074　**总机：**027-87690813　**传真：**027-87690013

欢迎投稿：bvtougao@gmail.com
读者邮箱：reader@broadview.com.cn

博文视点官方博客：http://blog.csdn.net/bvbook
博文视点官方网站：http://www.broadiew.com.cn

反侵权盗版声明

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

图书分类 程序设计

ISBN 978-7-121-10941-6

9 787121 109416 >

定价：49.80元