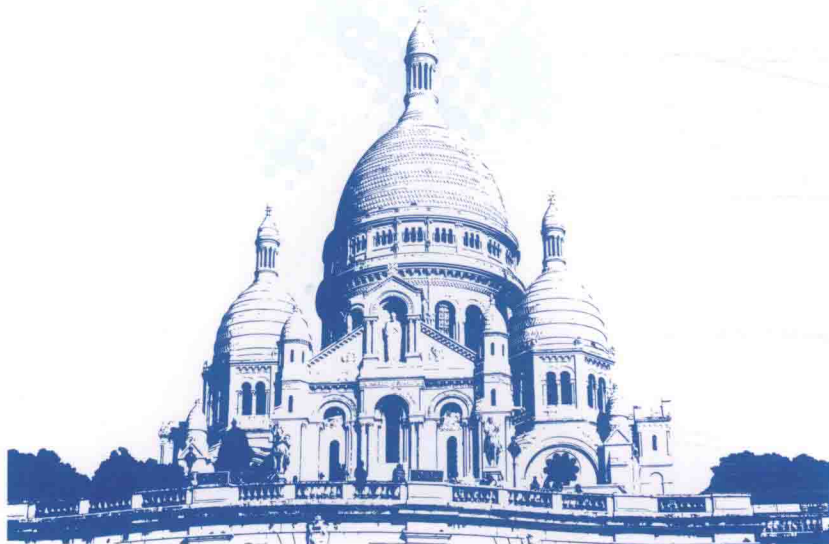


涵盖并行程序设计基础、思路、方法和实战
内容丰富，实例典型，实用性强

Broadview[®]
www.broadview.com.cn



实战 Java 高并发程序设计

葛一鸣 郭超 编著



中国工信出版集团



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

实战 Java 高并发程序设计

葛一鸣 郭超 编著



电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

在过去单核 CPU 时代，单任务在一个时间点只能执行单一程序，随着多核 CPU 的发展，并行程序开发就显得尤为重要。

本书主要介绍基于 Java 的并行程序设计基础、思路、方法和实战。第一，立足于并发程序基础，详细介绍 Java 中进行并行程序设计的基本方法。第二，进一步详细介绍 JDK 中对并行程序的强大支持，帮助读者快速、稳健地进行并行程序开发。第三，详细讨论有关“锁”的优化和提高并行程序性能级别的方法和思路。第四，介绍并行的基本设计模式及 Java 8 对并行程序的支持和改进。第五，介绍高并发框架 Akka 的使用方法。最后，详细介绍并行程序的调试方法。

本书内容丰富，实例典型，实用性强，适合有一定 Java 基础的技术开发人员阅读。

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

实战 Java 高并发程序设计 / 葛一鸣，郭超编著. —北京：电子工业出版社，2015.11
ISBN 978-7-121-27304-9

I. ①实… II. ①葛… ②郭… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 231507 号

责任编辑：董 英

印 刷：北京天宇星印刷厂

装 订：北京天宇星印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：22

字数：493 千字

版 次：2015 年 11 月第 1 版

印 次：2015 年 11 月第 1 次印刷

印 数：3000 册 定价：69.00 元

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

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

服务热线：（010）88258888。

前 言

关于 Java 与并行

由于单核 CPU 的主频逐步逼近极限，多核 CPU 架构成为了一种必然的技术趋势。所以，多线程并行程序便显得越来越重要。并行计算的一个重要应用场景就是服务端编程。可以看到，目前服务端 CPU 的核心数已经轻松超越 10 核心，而 Java 显然已经成为当下最流行的服务端编程语言，因此熟悉和了解基于 Java 的并行程序开发有着重要的实用价值。

本书的体系结构

本书立足于实际开发，又不缺乏理论介绍，力求通俗易懂、循序渐进。本书共分为 8 章。

第 1 章主要介绍了并行计算中相关的一些基本概念，树立读者对并行计算的基本认识；介绍了两个重要的并行性能评估定律，以及 Java 内存模型 JMM。

第 2 章介绍了 Java 并行程序开发的基础，包括 Java 中 Thread 的基本使用方法等，也详细介绍了并行程序容易引发的一些错误和误用。

第 3 章介绍了 JDK 内部对并行程序开发的支持，主要介绍 JUC (Java.util.concurrent) 中一些工具的使用方法、各自特点及它们的内部实现原理。

第 4 章介绍了在开发过程中可以进行的对锁的优化，也进一步简要描述了 Java 虚拟机层面对并行程序的优化支持。此外，还花费一定篇幅介绍了有关无锁的计算。

第 5 章介绍了并行程序设计中常见的一些设计模式以及一些典型的并行算法和使用方法，其中包括重要的 Java NIO 和 AIO 的介绍。

第 6 章介绍了 Java 8 中为并行计算做的新的改进，包括并行流、CompletableFuture、StampedLock 和 LongAdder。

第 7 章主要介绍了高并发框架 Akka 的基本使用方法，并使用 Akka 框架实现了一个简单的粒子群算法，模拟超高并发的场景。

第 8 章介绍了使用 Eclipse 进行多线程调试的方法，并演示了通过 Eclipse 进行多线程调试重现 ArrayList 的线程不安全问题。

本书特色

本书的主要特点如下。

1. **结构清晰。**本书一共 8 章，总体上循序渐进，逐步提升。每一章都各自有鲜明的侧重点，有利于读者快速抓住重点。
2. **理论结合实战。**本书注重实战，书中重要的知识点都安排了代码实例，帮助读者理解。同时也不忘记对系统的内部实现原理进行深度剖析。
3. **通俗易懂。**本书尽量避免采用过于理论的描述方式，简单的白话文风格贯穿全书，配图基本上为手工绘制，降低了理解难度，并尽量做到读者在阅读过程中少盲点、无盲点。

适合阅读人群

虽然本书力求通俗，但要通读本书并取得良好的学习效果，要求读者需要具备基本的 Java 知识或者一定的编程经验。因此，本书适合以下读者：

- 拥有一定开发经验的 Java 平台开发人员（Java、Scala、JRuby 等）
- 软件设计师、架构师
- 系统调优人员
- 有一定的 Java 编程基础并希望进一步加深对并行的理解的研发人员

本书的约定

本书在叙述过程中，有如下约定：

- 本书中所述的 JDK 1.5、JDK 1.6、JDK 1.7、JDK 1.8 分别等同于 JDK 5、JDK 6、JDK 7、JDK 8。
- 如无特殊说明，本书的程序、示例均在 JDK 1.7 环境中运行。

联系作者

本书的写作过程远比我想象得更艰辛，为了让全书能够更清楚、更正确地表达和论述，我经历了很多个不眠之夜，即使现在回想起来，我也忍不住会打个寒战。由于写作水平的限制，书中难免会有不妥之处，望读者谅解。

为此，如果读者有任何疑问或者建议，非常欢迎大家加入 QQ 群 397196583，一起探讨学习中的困难、分享学习的经验，我期待与大家交流、共同进步。同时，也希望大家可以关注我的博客 <http://www.uucode.net/>。

感谢

这本书能够面世，是因为得到了众人的支持。首先，要感谢我的妻子，她始终不辞辛劳、毫无怨言地对我照顾有加，才让我得以腾出大量时间，并可以安心工作。其次，要感谢所有编辑为我一次又一次地审稿改错，批评指正，才能让本书逐步完善。最后，感谢我的母亲 30 年如一日对我的体贴和关心。

参与本书编写的还有安继宏、白慧、薛淑英、蒋玺、曹静、马玉杰、陈明明、张丽萍、任娜娜、李清艺、荆海霞、赵全利、孙迪，在此一并感谢！

葛一鸣

十载耕耘奠定专业地位

以书为证彰显卓越品质

博文视点诚邀精锐作者加盟

《C++Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博古通今、文圆质方、视角独特、点石成金之计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于IT专业出版之巔。

英雄帖

江湖风云起,代有才人出。
IT界群雄并起,逐鹿中原。
博文视点诚邀天下技术英豪加入,
指点江山,激扬文字
传播信息技术,分享IT心得

· 专业的作者服务 ·

博文视点自成立以来一直专注于IT专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照IT技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身定制写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn



@博文视点Broadview



博文视点Broadview



目 录

第 1 章 走入并行世界.....	1
1.1 何去何从的并行计算	1
1.1.1 忘掉那该死的并行.....	2
1.1.2 可怕的现实：摩尔定律的失效.....	4
1.1.3 柳暗花明：不断地前进.....	5
1.1.4 光明或是黑暗.....	6
1.2 你必须知道的几个概念	6
1.2.1 同步（Synchronous）和异步（Asynchronous）	7
1.2.2 并发（Concurrency）和并行（Parallelism）	8
1.2.3 临界区.....	9
1.2.4 阻塞（Blocking）和非阻塞（Non-Blocking）	9
1.2.5 死锁（Deadlock）、饥饿（Starvation）和活锁（Livelock）	9
1.3 并发级别	11
1.3.1 阻塞（Blocking）	11
1.3.2 无饥饿（Starvation-Free）	11
1.3.3 无障碍（Obstruction-Free）	12
1.3.4 无锁（Lock-Free）	12
1.3.5 无等待（Wait-Free）	13
1.4 有关并行的两个重要定律.....	13
1.4.1 Amdahl 定律.....	13
1.4.2 Gustafson 定律	16

1.4.3	Amdahl 定律和 Gustafson 定律是否相互矛盾	16
1.5	回到 Java: JMM.....	17
1.5.1	原子性 (Atomicity)	18
1.5.2	可见性 (Visibility)	20
1.5.3	有序性 (Ordering)	22
1.5.4	哪些指令不能重排: Happen-Before 规则	27
1.6	参考文献	27
第 2 章	Java 并行程序基础	29
2.1	有关线程你必须知道的事.....	29
2.2	初始线程: 线程的基本操作.....	32
2.2.1	新建线程.....	32
2.2.2	终止线程.....	34
2.2.3	线程中断.....	38
2.2.4	等待 (wait) 和通知 (notify)	41
2.2.5	挂起 (suspend) 和继续执行 (resume) 线程.....	44
2.2.6	等待线程结束 (join) 和谦让 (yield)	48
2.3	volatile 与 Java 内存模型 (JMM)	50
2.4	分门别类的管理: 线程组.....	52
2.5	驻守后台: 守护线程 (Daemon)	54
2.6	先干重要的事: 线程优先级.....	55
2.7	线程安全的概念与 synchronized	57
2.8	程序中的幽灵: 隐蔽的错误.....	61
2.8.1	无提示的错误案例.....	61
2.8.2	并发下的 ArrayList	62
2.8.3	并发下诡异的 HashMap	63
2.8.4	初学者常见问题: 错误的加锁.....	66
2.9	参考文献	68
第 3 章	JDK 并发包.....	70
3.1	多线程的团队协作: 同步控制.....	70
3.1.1	synchronized 的功能扩展: 重入锁.....	71
3.1.2	重入锁的好搭档: Condition 条件.....	80

3.1.3	允许多个线程同时访问：信号量（Semaphore）	83
3.1.4	ReadWriteLock 读写锁	85
3.1.5	倒计时器：CountDownLatch	87
3.1.6	循环栅栏：CyclicBarrier	89
3.1.7	线程阻塞工具类：LockSupport	92
3.2	线程复用：线程池	95
3.2.1	什么是线程池	96
3.2.2	不要重复发明轮子：JDK 对线程池的支持	97
3.2.3	刨根究底：核心线程池的内部实现	102
3.2.4	超负载了怎么办：拒绝策略	106
3.2.5	自定义线程创建：ThreadFactory	109
3.2.6	我的应用我做主：扩展线程池	110
3.2.7	合理的选择：优化线程池线程数量	112
3.2.8	堆栈去哪里了：在线程池中寻找堆栈	113
3.2.9	分而治之：Fork/Join 框架	117
3.3	不要重复发明轮子：JDK 的并发容器	121
3.3.1	超好用的工具类：并发集合简介	121
3.3.2	线程安全的 HashMap	122
3.3.3	有关 List 的线程安全	123
3.3.4	高效读写的队列：深度剖析 ConcurrentLinkedQueue	123
3.3.5	高效读取：不变模式下的 CopyOnWriteArrayList	129
3.3.6	数据共享通道：BlockingQueue	130
3.3.7	随机数据结构：跳表（SkipList）	134
3.4	参考资料	136
第 4 章	锁的优化及注意事项	138
4.1	有助于提高“锁”性能的几点建议	139
4.1.1	减小锁持有时间	139
4.1.2	减小锁粒度	140
4.1.3	读写分离锁来替换独占锁	142
4.1.4	锁分离	142
4.1.5	锁粗化	144

4.2	Java 虚拟机对锁优化所做的努力.....	146
4.2.1	锁偏向.....	146
4.2.2	轻量级锁.....	146
4.2.3	自旋锁.....	146
4.2.4	锁消除.....	146
4.3	人手一支笔: ThreadLocal	147
4.3.1	ThreadLocal 的简单使用	148
4.3.2	ThreadLocal 的实现原理	149
4.3.3	对性能有何帮助.....	155
4.4	无锁	157
4.4.1	与众不同的并发策略: 比较交换 (CAS)	158
4.4.2	无锁的线程安全整数: AtomicInteger.....	159
4.4.3	Java 中的指针: Unsafe 类	161
4.4.4	无锁的对象引用: AtomicReference.....	162
4.4.5	带有时间戳的对象引用: AtomicStampedReference.....	165
4.4.6	数组也能无锁: AtomicIntegerArray	168
4.4.7	让普通变量也享受原子操作: AtomicIntegerFieldUpdater.....	169
4.4.8	挑战无锁算法: 无锁的 Vector 实现.....	171
4.4.9	让线程之间互相帮助: 细看 SynchronousQueue 的实现.....	176
4.5	有关死锁的问题	179
4.6	参考文献	183
第 5 章	并行模式与算法	184
5.1	探讨单例模式	184
5.2	不变模式	187
5.3	生产者-消费者模式	190
5.4	高性能的生产者-消费者: 无锁的实现	194
5.4.1	无锁的缓存框架: Disruptor	195
5.4.2	用 Disruptor 实现生产者-消费者案例	196
5.4.3	提高消费者的响应时间: 选择合适的策略.....	199
5.4.4	CPU Cache 的优化: 解决伪共享问题	200
5.5	Future 模式.....	204

5.5.1	Future 模式的主要角色	206
5.5.2	Future 模式的简单实现	207
5.5.3	JDK 中的 Future 模式	210
5.6	并行流水线	212
5.7	并行搜索	216
5.8	并行排序	218
5.8.1	分离数据相关性：奇偶交换排序	218
5.8.2	改进的插入排序：希尔排序	221
5.9	并行算法：矩阵乘法	226
5.10	准备好了再通知我：网络 NIO	230
5.10.1	基于 Socket 的服务端的多线程模式	230
5.10.2	使用 NIO 进行网络编程	235
5.10.3	使用 NIO 来实现客户端	243
5.11	读完了再通知我：AIO	245
5.11.1	AIO EchoServer 的实现	245
5.11.2	AIO Echo 客户端实现	248
5.12	参考文献	249
第 6 章	Java 8 与并发	251
6.1	Java 8 的函数式编程简介	251
6.1.1	函数作为一等公民	252
6.1.2	无副作用	252
6.1.3	申明式的（Declarative）	253
6.1.4	不变的对象	254
6.1.5	易于并行	254
6.1.6	更少的代码	254
6.2	函数式编程基础	255
6.2.1	FunctionalInterface 注释	255
6.2.2	接口默认方法	256
6.2.3	lambda 表达式	259
6.2.4	方法引用	260
6.3	一步一步走入函数式编程	263

6.4	并行流与并行排序	267
6.4.1	使用并行流过滤数据	267
6.4.2	从集合得到并行流	268
6.4.3	并行排序	268
6.5	增强的 Future: CompletableFuture	269
6.5.1	完成了就通知我	269
6.5.2	异步执行任务	270
6.5.3	流式调用	272
6.5.4	CompletableFuture 中的异常处理	272
6.5.5	组合多个 CompletableFuture	273
6.6	读写锁的改进: StampedLock	274
6.6.1	StampedLock 使用示例	275
6.6.2	StampedLock 的小陷阱	276
6.6.3	有关 StampedLock 的实现思想	278
6.7	原子类的增强	281
6.7.1	更快的原子类: LongAdder	281
6.7.2	LongAdder 的功能增强版: LongAccumulator	287
6.8	参考文献	288
第 7 章	使用 Akka 构建高并发程序	289
7.1	新并发模型: Actor	290
7.2	Akka 之 Hello World	290
7.3	有关消息投递的一些说明	293
7.4	Actor 的生命周期	295
7.5	监督策略	298
7.6	选择 Actor	303
7.7	消息收件箱 (Inbox)	303
7.8	消息路由	305
7.9	Actor 的内置状态转换	308
7.10	询问模式: Actor 中的 Future	311
7.11	多个 Actor 同时修改数据: Agent	313
7.12	像数据库一样操作内存数据: 软件事务内存	316

7.13 一个有趣的例子：并发粒子群的实现.....	319
7.13.1 什么是粒子群算法.....	320
7.13.2 粒子群算法的计算过程.....	320
7.13.3 粒子群算法能做什么.....	322
7.13.4 使用 Akka 实现粒子群.....	323
7.14 参考文献.....	330
第 8 章 并行程序调试.....	331
8.1 准备实验样本.....	331
8.2 正式起航.....	332
8.3 挂起整个虚拟机.....	334
8.4 调试进入 ArrayList 内部.....	336

1

第 1 章

走入并行世界

当你打开本书，也许你正试图将你的应用改造成并行模式运行，也许你只是单纯地对并行程序感兴趣。无论出于何种原因，你正对并行计算充满好奇、疑问和求知欲。如果是这样，那就对了，带着你的好奇和疑问，让我们一起遨游并行程序的世界，深入了解它们究竟是如何工作的吧！

不过首先，我想要公布一条令人沮丧的消息。就在大伙儿都认为并行计算必然成为未来的大趋势时，2014 年底，Avoiding ping pong 论坛上，伟大的 Linus Torvalds 提出了一个截然不同的观点，他说：“忘掉那该死的并行吧！”（原文：Give it up. The whole "parallel computing is the future" is a bunch of crock.）

1.1 何去何从的并行计算

到底我们该如何选择呢？本节的目的就是拨云见日。

1.1.1 忘掉那该死的并行

Linus Torvalds 是一个传奇式的人物（图 1.1），是他给出了 Linux 的原型，并一直致力于推广和发展 Linux 系统。他在 1991 年首先在网络上发布了 Linux 源码，从此一发而不可收。Linux 迅速崛起壮大，成为目前使用最广泛的操作系统之一。

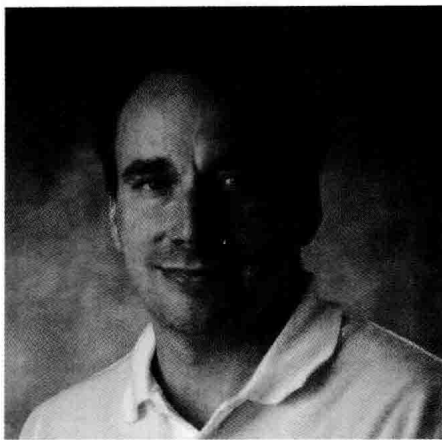


图 1.1 传奇的 Linus Torvalds

自 2002 年起，Linus 就决定使用 BitKeeper 作为 Linux 内核开发的版本控制工具，以此来维护 Linux 的内核源码。BitKeeper 是一套分布式版本控制软件，它是一套商用系统，由 BitMover 公司开发。2005 年，BitKeeper 宣称发现 Linux 内核开发人员使用逆向工程来试图解析 BitKeeper 内部协议。因此，决定向 Linus 收回 BitKeeper 授权。尽管 Linux 核心团队与 BitMover 公司进行了协商，但是无法解决他们之间的分歧。因此，Linus 决定自行研发版本控制工具来代替 BitKeeper。于是，Git 诞生了。

如果大家正在使用 Git，我相信你们一定会被 Git 的魅力所折服，如果还没有了解过 Git，那么我强烈建议你去看一下这款优秀的产品。

而正是这位传奇人物，给目前红红火火的并行计算泼了一大盆冷水。那么，并行计算究竟应该何去何从呢？

在 Linus 的发言中这么说道：

Where the hell do you envision that those magical parallel algorithms would be used?

The only place where parallelism matters is in graphics or on the server side, where we already largely have it. Pushing it anywhere else is just pointless.

需要有多么奇葩的想象力才能想象出并行计算的用武之地？

并行计算只有在图像处理和服务端编程 2 个领域可以使用，并且它在这 2 个领域确实有着大量广泛的使用。但是在其他任何地方，并行计算毫无建树！

So the whole argument that people should parallelize their code is fundamentally flawed. It rests on incorrect assumptions. It's a fad that has been going on too long.

因此，人们在争论是否应该将他们的代码并行化是一个本质上的错误。这完全就基于一个错误的假设。“并行”是一个早该结束的时髦用语。

看了这段较为完整的表述，大家应该对 Linus 的观点有所感触，我对此也表示赞同。与串行程序不同，并行程序的设计和实现异常复杂，不仅仅体现在程序的功能分离上，多线程间的协调性、乱序性都会成为程序正确执行的障碍。只要你稍不留神，就会失之毫厘，谬以千里！混乱的程序难以阅读、难以理解，更难以调试。所谓并行，也就是把简单问题复杂化的典型。因此，只有“疯子”才会叫嚣并行就是未来（the crazies talking about scaling to hundreds of cores are just that - crazy）。

但是，Linus 也提出了两个特例，那就是图像处理和服务端程序是可以、也需要使用并行技术的。仔细想想，为什么图像处理和服务端程序是特例呢？

和用户终端程序不同，图像处理往往拥有极大的计算量。一张 1024×768 像素的图片，包含多达 78 万 6 千多个像素。即使将所有的像素遍历一遍，也得花不少时间。更何况，图像处理涉及大量的矩阵计算。矩阵的规模和数量都会非常大。面对如此密集的计算，很有可能超过单核 CPU 的计算能力，所以自然需要引入多核计算了。

而服务端程序与一般的用户终端程序相比，一方面，服务端程序需要承受很重的用户访问压力。根据淘宝的数据，它在“双十一”一天，支付宝核心数据库集群处理了 41 亿个事务，执行 285 亿次 SQL，生成 15TB 日志，访问 1931 亿次内存数据块，13 亿个物理读。如此密集访问，恐怕任何一台单机都难以胜任，因此，并行程序也就自然成了唯一的出路。另一方面，服务端程序往往会比用户终端程序拥有更复杂的业务模型。面对复杂业务模型，并行程序会比串行程序更容易适应业务需求，更容易模拟我们的现实世界。毕竟，我们的世界本质上是并行的。比如，当你开开心心去上学的时候，妈妈可能在家里忙着家务，爸爸在外打工赚钱，一家人其乐融融。如果有一天，你需要使用你的计算机来模拟这个场景，你会怎么做呢？如果你就在一个线程里，既做了你自己，又做了妈妈，又做了爸爸，显然这不是一种好的解决方案。但如果你使用三个线程，分别模拟这三个人，一切看起来又是那么自然，而且容易被人理解。