

Java并发编程的艺术 (Java核心技术系列)

作者：方腾飞,魏鹏,程晓明 著

Java核心技术系列

Java并发编程的艺术

方腾飞 魏鹏 程晓明 著

ISBN: 978-7-111-50824-3

本书纸版由机械工业出版社于2015年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @华章数媒

腾讯微博 @yanfabook

微信公众号 华章电子书（微信号：hzebook）

目录

前言

第1章 并发编程的挑战

1.1 上下文切换

1.1.1 多线程一定快吗

1.1.2 测试上下文切换次数和时长

1.1.3 如何减少上下文切换

1.1.4 减少上下文切换实战

1.2 死锁

1.3 资源限制的挑战

1.4 本章小结

第2章 Java并发机制的底层实现原理

2.1 volatile的应用

2.2 synchronized的实现原理与应用

2.2.1 Java对象头

2.2.2 锁的升级与对比

2.3 原子操作的实现原理

2.4 本章小结

第3章 Java内存模型

3.1 Java内存模型的基础

3.1.1 并发编程模型的两个关键问题

3.1.2 Java内存模型的抽象结构

3.1.3 从源代码到指令序列的重排序

3.1.4 并发编程模型分类

3.1.5 happens-before简介

3.2 重排序

3.2.1 数据依赖性

3.2.2 as-if-serial语义

3.2.3 程序顺序规则

3.2.4 重排序对多线程的影响

3.3 顺序一致性

3.3.1 数据竞争与顺序一致性

3.3.2 顺序一致性内存模型

3.3.3 同步程序的顺序一致性效果

3.3.4 未同步程序的执行特性

3.4 volatile的内存语义

3.4.1 volatile的特性

3.4.2 volatile写-读建立的happens-before关系

3.4.3 volatile写-读的内存语义

3.4.4 volatile内存语义的实现

3.4.5 JSR-133为什么要增强volatile的内存语义

3.5 锁的内存语义

3.5.1 锁的释放-获取建立的happens-before关系

3.5.2 锁的释放和获取的内存语义

3.5.3 锁内存语义的实现

3.5.4 concurrent包的实现

3.6 final域的内存语义

3.6.1 final域的重排序规则

3.6.2 写final域的重排序规则

3.6.3 读final域的重排序规则

3.6.4 final域为引用类型

3.6.5 为什么final引用不能从构造函数内“溢出”

3.6.6 final语义在处理器中的实现

3.6.7 JSR-133为什么要增强final的语义

3.7 happens-before

3.7.1 JMM的设计

3.7.2 happens-before的定义

3.7.3 happens-before规则

3.8 双重检查锁定与延迟初始化

3.8.1 双重检查锁定的由来

3.8.2 问题的根源

3.8.3 基于volatile的解决方案

3.8.4 基于类初始化的解决方案

3.9 Java内存模型综述

3.9.1 处理器的内存模型

3.9.2 各种内存模型之间的关系

3.9.3 JMM的内存可见性保证

3.9.4 JSR-133对旧内存模型的修补

3.10 本章小结

第4章 Java并发编程基础

4.1 线程简介

4.1.1 什么是线程

4.1.2 为什么要使用多线程

4.1.3 线程优先级

4.1.4 线程的状态

4.1.5 Daemon线程

4.2 启动和终止线程

- [4.2.1 构造线程](#)
- [4.2.2 启动线程](#)
- [4.2.3 理解中断](#)
- [4.2.4 过期的suspend\(\)、resume\(\)和stop\(\)](#)
- [4.2.5 安全地终止线程](#)
- [4.3 线程间通信](#)
 - [4.3.1 volatile和synchronized关键字](#)
 - [4.3.2 等待/通知机制](#)
 - [4.3.3 等待/通知的经典范式](#)
 - [4.3.4 管道输入/输出流](#)
 - [4.3.5 Thread.join\(\)的使用](#)
 - [4.3.6 ThreadLocal的使用](#)
- [4.4 线程应用实例](#)
 - [4.4.1 等待超时模式](#)
 - [4.4.2 一个简单的数据库连接池示例](#)
 - [4.4.3 线程池技术及其示例](#)
 - [4.4.4 一个基于线程池技术的简单Web服务器](#)
- [4.5 本章小结](#)
- [第5章 Java中的锁](#)
 - [5.1 Lock接口](#)
 - [5.2 队列同步器](#)
 - [5.2.1 队列同步器的接口与示例](#)
 - [5.2.2 队列同步器的实现分析](#)
 - [5.3 重入锁](#)
 - [5.4 读写锁](#)
 - [5.4.1 读写锁的接口与示例](#)
 - [5.4.2 读写锁的实现分析](#)
 - [5.5 LockSupport工具](#)
 - [5.6 Condition接口](#)
 - [5.6.1 Condition接口与示例](#)
 - [5.6.2 Condition的实现分析](#)
- [5.7 本章小结](#)
- [第6章 Java并发容器和框架](#)
 - [6.1 ConcurrentHashMap的实现原理与使用](#)
 - [6.1.1 为什么要使用ConcurrentHashMap](#)
 - [6.1.2 ConcurrentHashMap的结构](#)
 - [6.1.3 ConcurrentHashMap的初始化](#)
 - [6.1.4 定位Segment](#)
 - [6.1.5 ConcurrentHashMap的操作](#)
 - [6.2 ConcurrentLinkedQueue](#)
 - [6.2.1 ConcurrentLinkedQueue的结构](#)
 - [6.2.2 入队列](#)
 - [6.2.3 出队列](#)
 - [6.3 Java中的阻塞队列](#)
 - [6.3.1 什么是阻塞队列](#)
 - [6.3.2 Java里的阻塞队列](#)
 - [6.3.3 阻塞队列的实现原理](#)
 - [6.4 Fork/Join框架](#)
 - [6.4.1 什么是Fork/Join框架](#)
 - [6.4.2 工作窃取算法](#)
 - [6.4.3 Fork/Join框架的设计](#)
 - [6.4.4 使用Fork/Join框架](#)
 - [6.4.5 Fork/Join框架的异常处理](#)
 - [6.4.6 Fork/Join框架的实现原理](#)
- [6.5 本章小结](#)
- [第7章 Java中的13个原子操作类](#)
 - [7.1 原子更新基本类型类](#)
 - [7.2 原子更新数组](#)
 - [7.3 原子更新引用类型](#)
 - [7.4 原子更新字段类](#)
- [7.5 本章小结](#)
- [第8章 Java中的并发工具类](#)
 - [8.1 等待多线程完成的CountDownLatch](#)
 - [8.2 同步屏障CyclicBarrier](#)
 - [8.2.1 CyclicBarrier简介](#)
 - [8.2.2 CyclicBarrier的应用场景](#)
 - [8.2.3 CyclicBarrier和CountDownLatch的区别](#)
 - [8.3 控制并发线程数的Semaphore](#)
 - [8.4 线程间交换数据的Exchanger](#)
- [8.5 本章小结](#)
- [第9章 Java中的线程池](#)
 - [9.1 线程池的实现原理](#)
 - [9.2 线程池的使用](#)
 - [9.2.1 线程池的创建](#)
 - [9.2.2 向线程池提交任务](#)
 - [9.2.3 关闭线程池](#)
 - [9.2.4 合理地配置线程池](#)

[9.2.5 线程池的监控](#)

[9.3 本章小结](#)

[第10章 Executor框架](#)

[10.1 Executor框架简介](#)

[10.1.1 Executor框架的两级调度模型](#)

[10.1.2 Executor框架的结构与成员](#)

[10.2 ThreadPoolExecutor详解](#)

[10.2.1 FixedThreadPool详解](#)

[10.2.2 SingleThreadExecutor详解](#)

[10.2.3 CachedThreadPool详解](#)

[10.3 ScheduledThreadPoolExecutor详解](#)

[10.3.1 ScheduledThreadPoolExecutor的运行机制](#)

[10.3.2 ScheduledThreadPoolExecutor的实现](#)

[10.4 FutureTask详解](#)

[10.4.1 FutureTask简介](#)

[10.4.2 FutureTask的使用](#)

[10.4.3 FutureTask的实现](#)

[10.5 本章小结](#)

[第11章 Java并发编程实践](#)

[11.1 生产者和消费者模式](#)

[11.1.1 生产者消费者模式实战](#)

[11.1.2 多生产者和多消费者场景](#)

[11.1.3 线程池与生产者消费者模式](#)

[11.2 线上问题定位](#)

[11.3 性能测试](#)

[11.4 异步任务池](#)

[11.5 本章小结](#)

前言

为什么要写这本书

记得第一次写并发编程的文章时还是在2012年，当时花了几个星期的时间写了一篇文章《深入分析volatile的实现原理》，准备在自己的博客中发表。在同事建法的建议下，怀着试一试的心态投向了InfoQ，庆幸的是半小时后得到InfoQ主编采纳的回复，高兴之情无以言表。这也是我第一次在专业媒体上发表文章，而后在InfoQ编辑张龙的不断鼓励和支持下，我陆续在InfoQ发表了几篇与并发编程相关的文章，于是便形成了“聊聊并发”专栏。在这个专栏的写作过程中，我得到快速的成长和非常多的帮助，在此非常感谢InfoQ的编辑们。2013年，华章的福川兄找到我，问有没有兴趣写一本书，当时觉得自己资历尚浅，婉言拒绝了。后来和福川兄一直保持联系，最后允许我花两年的时间来完成本书，所以答应了下来。由于并发编程领域的技术点非常多且深，所以陆续又邀请了同事魏鹏和朋友晓明一起参与到本书的编写当中。

写书的过程也是对自己研究和掌握的技术点进行整理的过程，希望本书能帮助读者快速掌握并发编程技术。

本书一共11章，由三名作者共同编写完成，其中第3章和第10章节由程晓明编写，第4章和第5章由魏鹏编写，其他7章由方腾飞编写。

本书特色

本书结合JDK的源码介绍了Java并发框架、线程池的实现原理，帮助读者做到知其所以然。

本书对原理的剖析不仅仅局限于Java层面，而是深入到JVM，甚至CPU层面来进行讲解，帮助读者从更底层看并发技术。

本书结合线上应用，给出了一些并发编程实战技巧，以及线上处理并发问题的步骤和思路。

读者对象

- Java开发工程师
- 架构师
- 并发编程爱好者
- 开设相关课程的大专院校师生

如何阅读本书

阅读本书之前，你必须有一定的Java基础和开发经验，最好还有一定的并发编程基础。如果你是一名并发编程初学者，建议按照顺序阅读本书，并按照书中的例子进行编码和实战。如果你有一定的并发编程经验，可以把本书当做一个手册，直接看需要学习的章节。以下是各章节的基本介绍。

第1章介绍Java并发编程的挑战，向读者说明进入并发编程的世界可能会遇到哪些问题，以及如何解决。

第2章介绍Java并发编程的底层实现原理，介绍在CPU和JVM这个层面是如何帮助Java实现并发编程的。

第3章介绍深入介绍了Java的内存模型。Java线程之间的通信对程序员完全透明，内存可见性问题很容易困扰Java程序员，本章试图揭开Java内存模型的神秘面纱。

第4章从介绍多线程技术带来的好处开始，讲述了如何启动和终止线程以及线程的状态，详细阐述了多线程之间进行通信的基本方式和等待/通知经典范式。

第5章介绍Java并发包中与锁相关的API和组件，以及这些API和组件的使用方式与实现细节。

第6章介绍了Java中的大部分并发容器，并深入剖析其实现原理，让读者领略大师的设计技巧。

第7章介绍了Java中的原子操作类，并给出一些实例。

第8章介绍了Java中提供的并发工具类，这是并发编程中的瑞士军刀。

第9章介绍了Java中的线程池实现原理和使用建议。

第10章介绍了Executor框架的整体结构和成员组件。

第11章介绍几个并发编程的实战，以及排查并发编程造成问题的方法。

勘误和支持

由于笔者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为此，特意创建一个在线支持与应急方案的站点<http://ifcve.com/book/>。你可以将书中的错误发布在勘误表页面中，同时如果你遇到任何问题，也可以访问Q&A页面，我将尽量在线上为读者提供最满意的解答。书中的全部源文件除可以从华章网站^[1]下载外，还可以从并发编程网站^[2]下载，我也会将相应的功能更新及时发布出来。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱tengfei@ifcve.com，期待能够得到你的真挚反馈。

致谢

感谢机械工业出版社华章公司的杨福川、高婧雅、孙海亮，在这一年多的时间中始终支持我的写作，你们的鼓励和帮助引导我顺利完成全部书稿。

感谢方正电子的刘老师，是他带我进入了面向对象的世界。

感谢我的主管朱老板，他在工作和生活上给予我很多的帮助和支持，还经常激励我完成本书编写。

最后感谢我的爸妈、岳父母和老婆，感谢你们的支持，并时时刻刻为我灌输信心和力量！

谨以此书献给我的儿子方熙皓，希望他能健康成长，以及众多热爱并发编程的朋友们，希望你们能快乐工作，认真生活！

方腾飞

[1] 参见华章网站www.hzbook.com。——编辑注

[2] <http://ifeve.com>。——编者注

第1章 并发编程的挑战

并发编程的目的是为了让程序运行得更快，但是，并不是启动更多的线程就能让程序最大限度地并发执行。在进行并发编程时，如果希望通过多线程执行任务让程序运行得更快，会面临非常多的挑战，比如上下文切换的问题、死锁的问题，以及受限于硬件和软件的资源限制问题，本章会介绍几种并发编程的挑战以及解决方案。

1.1 上下文切换

即使是单核处理器也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程是同时执行的，时间片一般是几十毫秒（ms）。

CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

这就像我们同时读两本书，当我们在读一本英文的技术书时，发现某个单词不认识，于是便打开中英文字典，但是在放下英文技术书之前，大脑必须先记住这本书读到了多少页的第多少行，等查完单词之后，能够继续读这本书。这样的切换是会影响读书效率的，同样上下文切换也会影响多线程的执行速度。

1.1.1 多线程一定快吗

下面的代码演示串行和并发执行并累加操作的时间，请分析：下面的代码并发执行一定比串行执行快吗？

```
public class ConcurrencyTest {
    private static final long count = 100001;
    public static void main(String[] args) throws InterruptedException {
        concurrency();
        serial();
    }
    private static void concurrency() throws InterruptedException {
        long start = System.currentTimeMillis();
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                int a = 0;
                for (long i = 0; i < count; i++) {
                    a += 5;
                }
            }
        });
        thread.start();
        int b = 0;
        for (long i = 0; i < count; i++) {
            b--;
        }
        long time = System.currentTimeMillis() - start;
        thread.join();
        System.out.println("concurrency : " + time + "ms, b=" + b);
    }
    private static void serial() {
        long start = System.currentTimeMillis();
        int a = 0;
        for (long i = 0; i < count; i++) {
            a += 5;
        }
        int b = 0;
        for (long i = 0; i < count; i++) {
            b--;
        }
        long time = System.currentTimeMillis() - start;
        System.out.println("serial: " + time + "ms, b=" + b + ", a=" + a);
    }
}
```

上述问题的答案是“不一定”，测试结果如表1-1所示。

表1-1 测试结果

□

从表1-1可以发现，当并发执行累加操作不超过百万次时，速度会比串行执行累加操作要慢。那么，为什么并发执行的速度会比串行慢呢？这是因为线程有创建和上下文切换的开销。

1.1.2 测试上下文切换次数和时长

下面我们来看看有什么工具可以度量上下文切换带来的消耗。

- 使用Lmbench3[\[1\]](#)可以测量上下文切换的时长。
- 使用vmstat可以测量上下文切换的次数。

下面是利用vmstat测量上下文切换次数的示例。

```
$ vmstat 1
procs -----memory----- --swap-- ----io---- --system-- -----cpu-----
 r  b   swpd   free   buff   cache   si   so    bi    bo    in    cs us sy id wa st
0  0     0 127876 398928 2297092    0    0     0     4     2     2  0  0 99  0  0
0  0     0 127868 398928 2297092    0    0     0     0 595 1171  0  1 99  0  0
0  0     0 127868 398928 2297092    0    0     0     0 590 1180  1  0 100  0  0
0  0     0 127868 398928 2297092    0    0     0     0 567 1135  0  1 99  0  0
```

CS（Content Switch）表示上下文切换的次数，从上面的测试结果中我们可以看到，上下文每1秒切换1000多次。

[\[1\]](#) Lmbench3是一个性能分析工具。

欢迎访问：电子书学习和下载网站 (<https://www.shgis.cn>)

文档名称：Java并发编程的艺术（Java核心技术系列）- 方腾飞, 魏鹏, 程晓明 著.epub

请登录 <https://shgis.cn/post/1825.html> 下载完整文档。

手机端请扫码查看：

