

Java

多线程编程核心技术

(第2版)

Java Multi-thread Programming
(Second Edition)

高洪岩 著



非外借



机械工业出版社
China Machine Press



Java

多线程编程核心技术

(第2版)

Java Multi-thread Programming
(Second Edition)

高洪岩 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Java 多线程编程核心技术 / 高洪岩著. —2 版. —北京: 机械工业出版社, 2019.1
(Java 核心技术系列)

ISBN 978-7-111-61490-6

I. J… II. 高… III. JAVA 语言 - 程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字 (2018) 第 274886 号

Java 多线程编程核心技术 (第 2 版)

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 赵亮宇

责任校对: 殷虹

印刷: 北京市荣盛彩色印刷有限公司

版次: 2019 年 5 月第 2 版第 1 次印刷

开本: 186mm × 240mm 1/16

印张: 28.25

书号: ISBN 978-7-111-61490-6

定价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

本书是国内首本整本系统、完整地介绍 Java 多线程技术的书籍，作为笔者，我要感谢大家的支持与厚爱。

本书第 1 版在出版后获得了广大 Java 程序员与学习者的关注，技术论坛、博客、公众号等平台大量涌现出针对 Java 多线程技术的讨论与分享。能为国内 IT 知识的建设贡献微薄之力是让我最欣慰的。

有些读者在第一时间就根据书中的知识总结了学习笔记，并在博客中进行分享，笔者非常赞赏这种传播知识的精神。知识就要分享，知识就要传播，这样才能共同进步。

第 2 版与第 1 版的区别

本书第 1 版上市后收到了大量的读者反馈，我对每一个建议都细心地进行整理，力求在第 2 版中得以完善。

第 2 版在第 1 版的基础上着重加强了 8 点更新：

- 1) 大量知识点重排，更有利于阅读与理解；
- 2) 更新了读者提出的共性问题并进行集中讲解；
- 3) 丰富 Thread.java 类 API 的案例，使其更具有实用性；
- 4) 对线程的信息进行监控实时采样；
- 5) 强化了 volatile 语义、多线程核心 synchronized 的案例；
- 6) 力求知识点连贯，方便深度学习与理解，增加原子与线程安全的内容；
- 7) 深入浅出地介绍代码重排特性；
- 8) 细化工具类 ThreadLocal 和 InheritableThreadLocal 的源代码分析与原理。

由于篇幅有限，有关线程池的知识请参考笔者的另一本书——《Java 并发编程：核心

方法与框架》，那本书中有针对 Java 并发编程技术的讲解。在向分布式领域进军时还需要用到 NIO 和 Socket 技术，故推荐笔者的拙作《NIO 与 Socket 编程技术指南》，希望可以给读者带来一些帮助。

本书秉承大道至简的主导思想，只介绍 Java 多线程开发中最值得关注的内容，希望抛砖引玉，以个人的一些想法和见解，为读者拓展出更深入、更全面的思路。

本书特色

在撰写本书的过程中，我尽量少用“啰唆”的文字，全部以 Demo 式案例来讲解技术点的实现，使读者看到代码及运行结果后就可以知道项目要解决的是什么问题，类似于网络中博客的风格，让读者用最短的时间学习知识点，明白知识点如何应用，以及在使用时要避免什么，使读者能够快速学习知识并解决问题。

读者对象

- Java 程序员；
- 系统架构师；
- Java 多线程开发者；
- Java 并发开发者；
- 大数据开发者；
- 其他对多线程技术感兴趣的人员。

如何阅读本书

本书本着实用、易懂的学习原则，利用 7 章来介绍 Java 多线程相关的技术。

第 1 章讲解了 Java 多线程的基础，包括 Thread 类的核心 API 的使用。

第 2 章讲解了在多线程中对并发访问的控制，主要是 synchronized 的使用。由于此关键字在使用上非常灵活，所以该章用很多案例来说明它的使用，为读者学习同步知识打好坚实的基础。

第 3 章讲解了线程之间的通信与交互细节。该章主要介绍 wait()、notifyAll() 和 notify() 方法的使用，使线程间能够互相通信，合作完成任务。该章还介绍了 ThreadLocal 类的使用。学习完该章，读者就能在 Thread 多线程中进行数据的传递了。

第 4 章讲解了 Lock 对象。因为 synchronized 关键字使用起来比较麻烦，所以 Java 5 提

供了 Lock 对象，更好地实现了并发访问时的同步处理，包括读写锁等。

第 5 章讲解了 Timer 定时器类，其内部原理是使用多线程技术。定时器在执行计划任务时是很重要的，在进行 Android 开发时也会深入使用。

第 6 章讲解的单例模式虽然很简单，但如果遇到多线程将会变得非常麻烦。如何在多线程中解决这么棘手的问题呢？本章会全面给出解决方案。

第 7 章对前面章节遗漏的技术空白点进行补充，通过案例使多线程的知识体系更加完整，尽量做到不出现技术空白点。

交流和支持

由于笔者水平有限，加上编写时间仓促，书中难免会出现一些疏漏或者不准确的地方，恳请读者批评指正，期待能够得到你们的真挚反馈，在技术之路上互勉共进。

联系笔者的邮箱是 279377921@qq.com。

致谢

在本书出版的过程中，感谢公司领导和同事的大力支持，感谢家人给予我充足的时间来撰写稿件，感谢出生 3 个多月的儿子高晟京，看到你，我有了更多动力，最后感谢在此稿件上耗费大量精力的高婧雅编辑与她的同事们，是你们的鼓励和帮助，引导我顺利完成了本书。

高洪岩

Contents 目 录

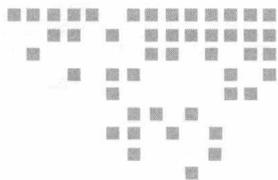
前言	
第 1 章 Java 多线程技能	1
1.1 进程和多线程概述	1
1.2 使用多线程	5
1.2.1 继承 Thread 类	5
1.2.2 使用常见命令分析线程的信息	8
1.2.3 线程随机性的展现	11
1.2.4 执行 start() 的顺序不代表 执行 run() 的顺序	12
1.2.5 实现 Runnable 接口	13
1.2.6 使用 Runnable 接口实现多线程 的优点	14
1.2.7 实现 Runnable 接口与继承 Thread 类的内部流程	16
1.2.8 实例变量共享造成的非线性安全 问题与解决方案	17
1.2.9 Servlet 技术造成的非线性安全 问题与解决方案	21
1.2.10 留意 i-- 与 System.out.println() 出现的非线性安全问题	24
1.3 currentThread() 方法	26
1.4 isAlive() 方法	29
1.5 sleep(long millis) 方法	31
1.6 sleep(long millis, int nanos) 方法	33
1.7 StackTraceElement[] getStackTrace() 方法	33
1.8 static void dumpStack() 方法	35
1.9 static Map<Thread, StackTrace- Element[]> getAllStackTraces() 方法	36
1.10 getId() 方法	38
1.11 停止线程	38
1.11.1 停止不了的线程	39
1.11.2 判断线程是否为停止状态	41
1.11.3 能停止的线程——异常法	43
1.11.4 在 sleep 状态下停止线程	47
1.11.5 用 stop() 方法暴力停止线程	49
1.11.6 stop() 方法与 java.lang. ThreadDeath 异常	51
1.11.7 使用 stop() 释放锁给数据造成 不一致的结果	52
1.11.8 使用“return;”语句停止线程 的缺点与解决方案	54
1.12 暂停线程	57
1.12.1 suspend() 方法与 resume() 方法的使用	57
1.12.2 suspend() 方法与 resume() 方法 的缺点——独占	58

1.12.3	suspend() 方法与 resume() 方法的缺点——数据不完整	62	2.2.4	一半异步，一半同步	105	
1.13	yield() 方法	63	2.2.5	synchronized 代码块间的同步性	108	
1.14	线程的优先级	64	2.2.6	println() 方法也是同步的	110	
1.14.1	线程优先级的继承特性	65	2.2.7	验证同步 synchronized(this) 代码块是锁定当前对象的	110	
1.14.2	优先级的规律性	66	2.2.8	将任意对象作为锁	113	
1.14.3	优先级的随机性	68	2.2.9	多个锁就是异步执行	116	
1.14.4	优先级对线程运行速度的影响	70	2.2.10	验证方法被调用是随机的	118	
1.15	守护线程	71	2.2.11	不同步导致的逻辑错误及其解决方法	121	
1.16	本章小结	73	2.2.12	细化验证 3 个结论	124	
第 2 章 对象及变量的并发访问			74	2.2.13	类 Class 的单例性	129
2.1	synchronized 同步方法	74	2.2.14	静态同步 synchronized 方法与 synchronized(class) 代码块	130	
2.1.1	方法内的变量为线程安全	74	2.2.15	同步 syn static 方法可以对类的所有对象实例起作用	135	
2.1.2	实例变量非线性安全问题与解决方案	77	2.2.16	同步 syn(class) 代码块可以对类的所有对象实例起作用	137	
2.1.3	同步 synchronized 在字节码指令中的原理	80	2.2.17	String 常量池特性与同步相关的问题与解决方案	138	
2.1.4	多个对象多个锁	81	2.2.18	同步 synchronized 方法无限等待问题与解决方案	141	
2.1.5	将 synchronized 方法与对象作为锁	84	2.2.19	多线程的死锁	143	
2.1.6	脏读	89	2.2.20	内置类与静态内置类	146	
2.1.7	synchronized 锁重入	91	2.2.21	内置类与同步：实验 1	149	
2.1.8	锁重入支持继承的环境	93	2.2.22	内置类与同步：实验 2	151	
2.1.9	出现异常，锁自动释放	94	2.2.23	锁对象改变导致异步执行	153	
2.1.10	重写方法不使用 synchronized	96	2.2.24	锁对象不改变依然同步执行	156	
2.1.11	public static boolean holdsLock (Object obj) 方法的使用	99	2.2.25	同步写法案例比较	158	
2.2	synchronized 同步语句块	99	2.3	volatile 关键字	159	
2.2.1	synchronized 方法的弊端	99	2.3.1	可见性的测试	159	
2.2.2	synchronized 同步代码块的使用	102	2.3.2	原子性的测试	168	
2.2.3	用同步代码块解决同步方法的弊端	104	2.3.3	禁止代码重排序的测试	176	
2.2.4	一半异步，一半同步	105	2.4	本章小结	187	
2.2.5	synchronized 代码块间的同步性	108				
2.2.6	println() 方法也是同步的	110				
2.2.7	验证同步 synchronized(this) 代码块是锁定当前对象的	110				
2.2.8	将任意对象作为锁	113				
2.2.9	多个锁就是异步执行	116				
2.2.10	验证方法被调用是随机的	118				
2.2.11	不同步导致的逻辑错误及其解决方法	121				
2.2.12	细化验证 3 个结论	124				
2.2.13	类 Class 的单例性	129				
2.2.14	静态同步 synchronized 方法与 synchronized(class) 代码块	130				
2.2.15	同步 syn static 方法可以对类的所有对象实例起作用	135				
2.2.16	同步 syn(class) 代码块可以对类的所有对象实例起作用	137				
2.2.17	String 常量池特性与同步相关的问题与解决方案	138				
2.2.18	同步 synchronized 方法无限等待问题与解决方案	141				
2.2.19	多线程的死锁	143				
2.2.20	内置类与静态内置类	146				
2.2.21	内置类与同步：实验 1	149				
2.2.22	内置类与同步：实验 2	151				
2.2.23	锁对象改变导致异步执行	153				
2.2.24	锁对象不改变依然同步执行	156				
2.2.25	同步写法案例比较	158				

第3章 线程间通信	188
3.1 wait/notify 机制.....	188
3.1.1 不使用 wait/notify 机制实现 线程间通信.....	188
3.1.2 wait/notify 机制.....	191
3.1.3 wait/notify 机制的原理.....	192
3.1.4 wait() 方法的基本使用.....	192
3.1.5 完整实现 wait/notify 机制.....	194
3.1.6 使用 wait/notify 机制实现 list.size() 等于 5 时的线程销毁.....	195
3.1.7 对业务代码进行封装.....	198
3.1.8 线程状态的切换.....	201
3.1.9 wait() 方法：立即释放锁.....	202
3.1.10 sleep() 方法：不释放锁.....	203
3.1.11 notify() 方法：不立即释放锁.....	204
3.1.12 interrupt() 方法遇到 wait() 方法.....	206
3.1.13 notify() 方法：只通知一个 线程.....	208
3.1.14 notifyAll() 方法：通知所有 线程.....	211
3.1.15 wait(long) 方法的基本使用.....	212
3.1.16 wait(long) 方法自动向下运行 需要重新持有锁.....	214
3.1.17 通知过早问题与解决方法.....	217
3.1.18 wait 条件发生变化与使用 while 的必要性.....	220
3.1.19 生产者 / 消费者模式的实现.....	224
3.1.20 通过管道进行线程间通信—— 字节流.....	250
3.1.21 通过管道进行线程间通信—— 字符流.....	253
3.1.22 实现 wait/notify 的交叉备份.....	256
3.2 join() 方法的使用.....	259
3.2.1 学习 join() 方法前的铺垫.....	259
3.2.2 join() 方法和 interrupt() 方法 出现异常.....	261
3.2.3 join(long) 方法的使用.....	263
3.2.4 join(long) 方法与 sleep(long) 方法的区别.....	264
3.2.5 join() 方法后面的代码提前 运行——出现意外.....	268
3.2.6 join() 方法后面的代码提前 运行——解释意外.....	270
3.2.7 join(long millis, int nanos) 方法 的使用.....	273
3.3 类 ThreadLocal 的使用.....	273
3.3.1 get() 方法与 null.....	274
3.3.2 类 ThreadLocal 存取数据 流程分析.....	275
3.3.3 验证线程变量的隔离性.....	277
3.3.4 解决 get() 方法返回 null 的问题.....	282
3.3.5 验证重写 initialValue() 方法的 隔离性.....	283
3.4 类 InheritableThreadLocal 的使用.....	284
3.4.1 类 ThreadLocal 不能实现值 继承.....	285
3.4.2 使用 InheritableThreadLocal 体现值继承特性.....	286
3.4.3 值继承特性在源代码中的执行 流程.....	288
3.4.4 父线程有最新的值，子线程仍 是旧值.....	291
3.4.5 子线程有最新的值，父线程仍 是旧值.....	293
3.4.6 子线程可以感应对象属性值 的变化.....	294
3.4.7 重写 childValue() 方法实现对 继承的值进行加工.....	297
3.5 本章小结.....	298

第 4 章 Lock 对象的使用	299	4.1.19 public boolean isLocked() 方法 的使用	334
4.1 使用 ReentrantLock 类	299	4.1.20 public void lockInterruptibly() 方法的使用	335
4.1.1 使用 ReentrantLock 实现同步	299	4.1.21 public boolean tryLock() 方法 的使用	336
4.1.2 验证多代码块间的同步性	301	4.1.22 public boolean tryLock (long timeout, TimeUnit unit) 方法 的使用	338
4.1.3 await() 方法的错误用法与更正	304	4.1.23 public boolean await (long time, TimeUnit unit) 方法 的使用	339
4.1.4 使用 await() 和 signal() 实现 wait/notify 机制	307	4.1.24 public long awaitNanos (long nanosTimeout) 方法 的使用	341
4.1.5 await() 方法暂停线程运行的 原理	309	4.1.25 public boolean awaitUntil (Date deadline) 方法的使用	342
4.1.6 通知部分线程——错误用法	312	4.1.26 public void awaitUninterru- ptibly() 方法的使用	344
4.1.7 通知部分线程——正确用法	314	4.1.27 实现线程按顺序执行业务	346
4.1.8 实现生产者 / 消费者模式一对一 交替输出	317	4.2 使用 ReentrantReadWriteLock 类	349
4.1.9 实现生产者 / 消费者模式多对多 交替输出	319	4.2.1 ReentrantLock 类的缺点	349
4.1.10 公平锁与非公平锁	321	4.2.2 ReentrantReadWriteLock 类的 使用——读读共享	351
4.1.11 public int getHoldCount() 方法 的使用	324	4.2.3 ReentrantReadWriteLock 类的 使用——写写互斥	352
4.1.12 public final int getQueue Length() 方法的使用	325	4.2.4 ReentrantReadWriteLock 类的 使用——读写互斥	352
4.1.13 public int getWaitQueue- Length (Condition condition) 方法的使用	327	4.2.5 ReentrantReadWriteLock 类的 使用——写读互斥	354
4.1.14 public final boolean has- QueuedThread (Thread thread) 方法的使用	328	4.3 本章小结	355
4.1.15 public final boolean has- QueuedThreads() 方法的使用	329	第 5 章 定时器 Timer	356
4.1.16 public boolean hasWaiters (Con- dition condition) 方法的使用	331	5.1 定时器 Timer 的使用	356
4.1.17 public final boolean isFair() 方法 的使用	332		
4.1.18 public boolean isHeldBy- CurrentThread() 方法的使用	333		

5.1.1	schedule(TimerTask task, Date time) 方法的测试	356	TERMINATED	410	
5.1.2	schedule(TimerTask task, Date firstTime, long period) 方法的测试	366	7.1.2	验证 TIMED_WAITING	411
5.1.3	schedule(TimerTask task, long delay) 方法的测试	374	7.1.3	验证 BLOCKED	412
5.1.4	schedule(TimerTask task, long delay, long period) 方法的测试	374	7.1.4	验证 WAITING	414
5.1.5	scheduleAtFixedRate (TimerTask task, Date firstTime, long period) 方法的测试	375	7.2	线程组	415
5.2	本章小结	384	7.2.1	线程对象关联线程组： 一级关联	416
第 6 章	单例模式与多线程	385	7.2.2	线程对象关联线程组： 多级关联	417
6.1	立即加载 / 饿汉模式	385	7.2.3	线程组自动归属特性	418
6.2	延迟加载 / 懒汉模式	387	7.2.4	获取根线程组	419
6.2.1	延迟加载 / 懒汉模式解析	387	7.2.5	线程组中加线程组	420
6.2.2	延迟加载 / 懒汉模式的缺点	388	7.2.6	组内的线程批量停止	421
6.2.3	延迟加载 / 懒汉模式的解决方案	390	7.2.7	递归取得与非递归取得 组内对象	422
6.3	使用静态内置类实现单例模式	399	7.3	Thread.activeCount() 方法的 使用	423
6.4	序列化与反序列化的单例模式 实现	400	7.4	Thread.enumerate(Thread tarray[]) 方法的使用	423
6.5	使用 static 代码块实现单例模式	402	7.5	再次实现线程执行有序性	424
6.6	使用 enum 枚举数据类型实现单例 模式	404	7.6	SimpleDateFormat 非线程安全	426
6.7	完善使用 enum 枚举数据类型实现 单例模式	405	7.6.1	出现异常	426
6.8	本章小结	407	7.6.2	解决异常的方法 1	428
第 7 章	拾遗增补	408	7.6.3	解决异常的方法 2	430
7.1	线程的状态	408	7.7	线程中出现异常的处理	431
7.1.1	验证 NEW、RUNNABLE 和		7.7.1	线程出现异常的默认行为	431
			7.7.2	使用 setUncaughtException- Handler() 方法进行异常处理	432
			7.7.3	使用 setDefaultUncaughtExce- ptionHandler() 方法进行异常 处理	433
			7.8	线程组内处理异常	434
			7.9	线程异常处理的优先性	437
			7.10	本章小结	442



Java 多线程技能

作为本书的第 1 章，重点是让读者快速进入 Java 多线程的学习，所以本章主要介绍 Thread 类的核心方法。Thread 类的核心方法较多，读者应该着重掌握如下技术点：

- ❑ 线程的启动；
- ❑ 如何使线程暂停；
- ❑ 如何使线程停止；
- ❑ 线程的优先级；
- ❑ 线程安全相关的问题。

以上内容也是本章学习的重点与思路，掌握这些内容是进入 Java 多线程学习的必经之路。

1.1 进程和多线程概述

本书主要介绍在 Java 语言中使用的多线程技术，但讲到多线程技术时不得不提及“进程”这个概念，“百度百科”对“进程”的解释如图 1-1 所示。

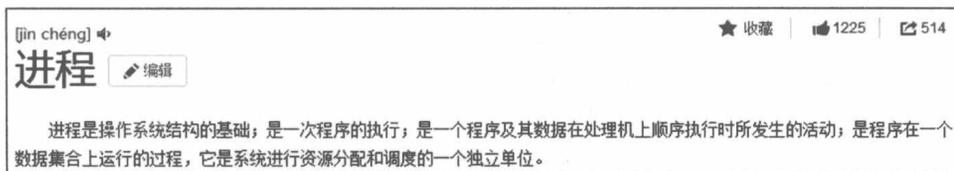


图 1-1 进程的定义

初看这段文字十分抽象，难以理解，那么再来看如图 1-2 所示的内容。



图 1-2 Windows 7 系统中的进程列表

难道一个正在操作系统中运行的 exe 程序可以理解成一个“进程”？没错！

通过查看“Windows 任务管理器”窗口中的列表，完全可以将运行在内存中的 exe 文件理解成进程——进程是受操作系统管理的基本运行单元。

程序是指令序列，这些指令可以让 CPU 完成指定的任务。*.java 程序经编译后形成 *.class 文件，在 Windows 中启动一个 JVM 虚拟机相当于创建了一个进程，在虚拟机中加载 class 文件并运行，在 class 文件中通过执行创建新线程的代码来执行具体的任务。创建测试用的代码如下：

```
public class Test1 {
    public static void main(String[] args) {
        try {
            Thread.sleep(Integer.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

在没有运行这个类之前，任务管理器中以“j”开头的进程列表如图 1-3 所示。

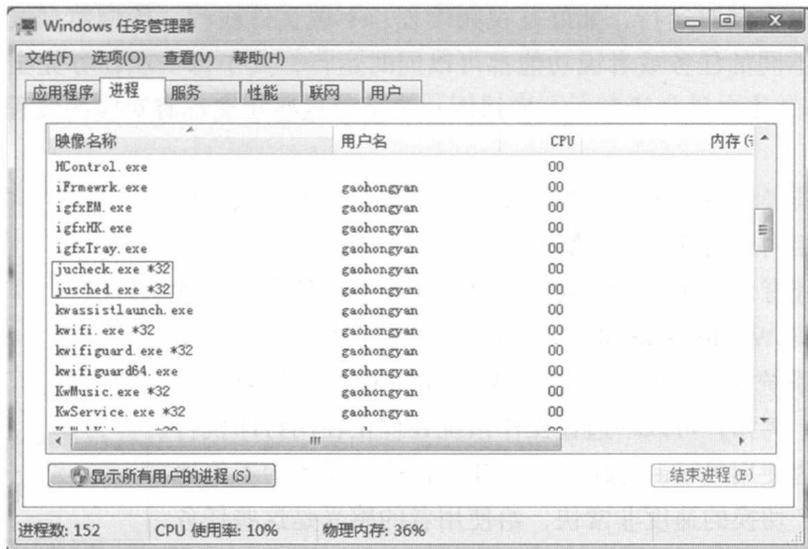


图 1-3 任务管理器中以“j”开头的进程

Test1 类重复运行 3 次后的进程列表如图 1-4 所示。可以看到，在任务管理器中创建了 3 个 javaw.exe 进程，说明每执行一次 main() 方法就创建一个进程，其本质上就是 JVM 虚拟机进程。

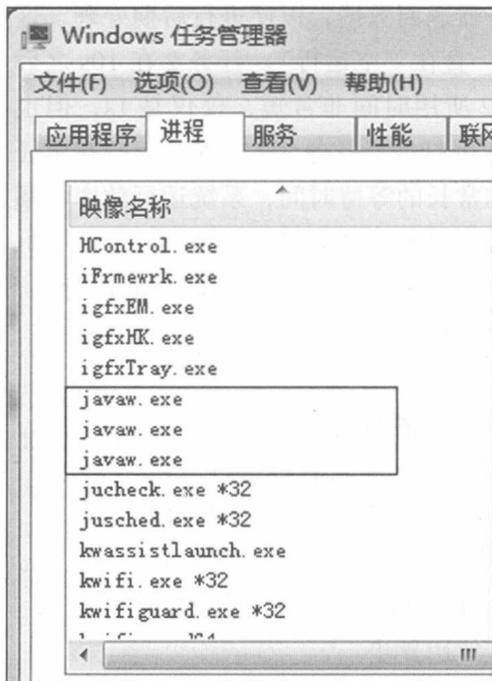


图 1-4 创建了 3 个 javaw.exe 进程

那什么是线程呢？线程可以理解为在进程中独立运行的子任务，例如，QQ.exe 运行时，

很多的子任务也在同时运行，如好友视频线程、下载文件线程、传输数据线程、发送表情线程等，这些不同的任务或者说功能都可以同时运行，其中每一项任务完全可以理解成是“线程”在工作，传文件、听音乐、发送图片表情等这些功能都有对应的线程在后台默默地运行。

进程负责向操作系统申请资源。在一个进程中，多个线程可以共享进程中相同的内存或文件资源。

使用多线程有什么优点呢？其实如果有使用“多任务操作系统”的经验，如 Windows 系列，大家应该都有这样的体会：使用多任务操作系统 Windows，可以大幅利用 CPU 的空闲时间来处理其他任务，例如，可以一边让操作系统处理正在用打印机打印的数据，一边使用 Word 编辑文档。CPU 在这些任务中不停地进行切换，由于切换的速度非常快，给使用者的感受是这些任务在同时运行，所以使用多线程技术可以在同一时间内执行更多不同的任务。

为了更加有效地理解多线程的优势，下面先来看如图 1-5 所示的单任务运行环境。

在图 1-5 中，任务 1 和任务 2 是两个完全独立、不相关的任务。任务 1 在等待远程服务器返回数据，以便进行后期处理，这时 CPU 一直呈等待状态，一直在“空运行”。任务 2 在 10s 之后被运行，虽然执行完任务 2 所用时间非常短，仅仅是 1s，但也必须等任务 1 运行结束后才可以运行任务 2，本程序运行在单任务环境中，所以任务 2 有非常长的等待时间，系统运行效率大幅降低。单任务的特点就是排队执行，即同步，就像在 cmd 中输入一条命令后，必须等待这条命令执行完才可以执行下一条命令。在同一时间只能执行一个任务，CPU 利用率大幅降低，这就是单任务运行环境的缺点。

多任务运行环境如图 1-6 所示。

在图 1-6 中，CPU 完全可以在任务 1 和任务 2 之间来回切换，使任务 2 不必等到 10s 之后再运行，系统和 CPU 的运行效率大大提升，这就是为什么要使用多线程技术、为什么要学习多线程。多任务的特点是在同一时间可以执行多个任务，这也是多线程技术的优点。使用多线程也就是在使用异步。

在通常情况下，单任务和多任务的实现与操作系统有关。例如，在一台计算机上使用同一个 CPU，安装 DOS 磁盘操作系统只能实现单任务运行环境，而

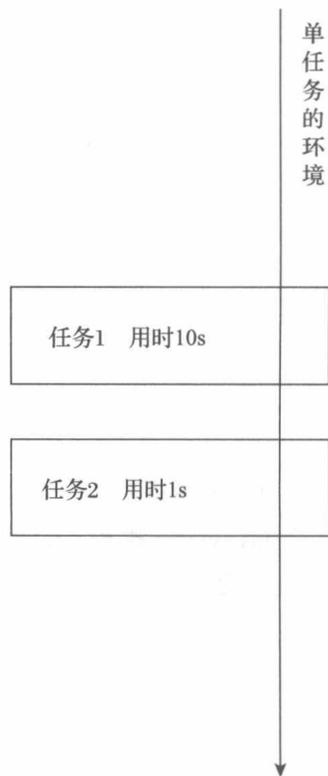


图 1-5 单任务运行环境

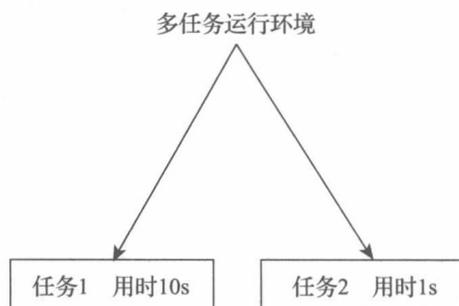


图 1-6 多任务运行环境

安装 Windows 操作系统则可以实现多任务运行环境。

在什么场景下使用多线程技术？笔者总结了两点。

1) 阻塞。一旦系统中出现了阻塞现象，则可以根据实际情况来使用多线程技术提高运行效率。

2) 依赖。业务分为两个执行过程，分别是 A 和 B。当 A 业务发生阻塞情况时，B 业务的执行不依赖 A 业务的执行结果，这时可以使用多线程技术来提高运行效率；如果 B 业务的执行依赖 A 业务的执行结果，则可以不使用多线程技术，按顺序进行业务的执行。

在实际的开发应用中，不要为了使用多线程而使用多线程，要根据实际场景决定。



多线程是异步的，所以千万不要把 Eclipse 代码的顺序当作线程执行的顺序，线程被调用的时机是随机的。

1.2 使用多线程

想学习一个技术就要“接近”它，所以本节首先通过一个示例来接触一下线程。

一个进程正在运行时至少会有一个线程在运行，这种情况在 Java 中也是存在的，这些线程在后台默默地执行，例如，调用 `public static void main()` 方法的线程就是这样的，而且它由 JVM 创建。

创建示例项目 `callMainMethodMainThread`，并创建 `Test.java` 类，代码如下：

```
package test;

public class Test {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName());
    }

}
```

程序运行结果如图 1-7 所示。

在控制台输出的 `main` 其实就是一个名称为 `main` 的线程在执行 `main()` 方法中的代码。另外，需要说明一下，在控制台输出的 `main` 和 `main` 方法没有任何关系，它们仅仅是名字相同而已。

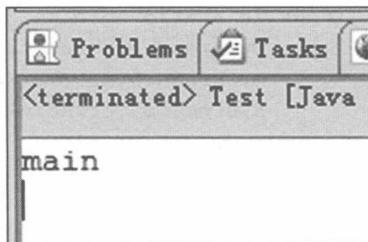


图 1-7 程序运行结果（主线程 `main` 出现）

1.2.1 继承 `Thread` 类

Java 的 JDK 开发包已经自带了对多线程技术的支持，通过它可以方便地进行多线程编程。实现多线程编程主要有两种方式：一种是继承

Thread 类，另一种是实现 Runnable 接口。

在学习如何创建新的线程前，先来看看 Thread 类的声明结构：

```
public class Thread implements Runnable
```

从上面的源代码中可以发现，Thread 类实现了 Runnable 接口，它们之间具有多态关系，多态结构的示例代码如下：

```
Runnable run1 = new Thread();
Runnable run2 = new MyThread();
Thread t1 = new MyThread();
```

其实使用继承 Thread 类的方式创建新线程时，最大的局限是不支持多继承，因为 Java 语言的特点是单根继承，所以为了支持多继承，完全可以实现 Runnable 接口，即一边实现一边继承，但这两种方式创建线程的功能是一样的，没有本质的区别。

本节主要介绍第一种方式。创建名称为 t1 的 Java 项目，创建一个自定义的线程类 MyThread.java，此类继承自 Thread，并且重写 run() 方法。在 run() 方法中添加线程要执行的任务代码如下：

```
package com.mythread.www;

public class MyThread extends Thread {
    @Override
    public void run() {
        super.run();
        System.out.println("MyThread");
    }
}
```

运行类代码如下：

```
package test;

import com.mythread.www.MyThread;

public class Run {

    public static void main(String[] args) {
        MyThread mythread = new MyThread();
        mythread.start();//耗时大
        System.out.println("运行结束!");//耗时小
    }
}
```

上面代码使用 start() 方法来启动一个线程，线程启动后会自动调用线程对象中的 run() 方法，run() 方法里面的代码就是线程对象要执行的任务，是线程执行任务的入口。

程序运行结果如图 1-8 所示。