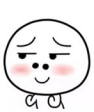
Broadview®
www.broadview.com.cn

Java 并发编程之美

翟陆续 薛宾田 著

中国工信出版集团

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



Java 并发编程之美

翟陆续 薛宾田 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

并发编程相比 Java 中其他知识点的学习门槛较高，从而导致很多人望而却步。但无论是职场面试，还是高并发 / 高流量系统的实现，却都离不开并发编程，于是能够真正掌握并发编程的人成为了市场迫切需求的人才。

本书通过图文结合、通俗易懂的讲解方式帮助大家完成多线程并发编程从入门到实践的飞跃！全书分为三部分，第一部分为 Java 并发编程基础篇，主要讲解 Java 并发编程的基础知识、线程有关的知识和并发编程中的其他相关概念，这些知识在高级篇都会有所使用，掌握了本篇的内容，就为学习高级篇奠定了基础；第二部分为 Java 并发编程高级篇，讲解了 Java 并发包中核心组件的实现原理，让读者知其然，也知其所以然，熟练掌握本篇内容，对我们在日常开发高并发、高流量的系统时会大有裨益；第三部分为 Java 并发编程实践篇，主要讲解并发组件的使用方法，以及在使用过程中容易遇到的问题和解决方法。

本书适合 Java 初级、中高级研发工程师，对 Java 并发编程感兴趣，以及希望探究 JUC 包源码原理的人员阅读。

图书在版编目（CIP）数据

Java 并发编程之美 / 翟陆续，薛宾田著 . —北京：电子工业出版社，2018.11
ISBN 978-7-121-34947-8

Ⅰ . ① J… Ⅱ . ①翟… ②薛… Ⅲ . ① JAVA 语言 – 程序设计 Ⅳ . ① TP312

中国版本图书馆 CIP 数据核字 (2018) 第 199378 号

策划编辑：刘　皎
责任编辑：牛　勇
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
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：787×980　1/16　印张：22.5　字数：432 千字
版　　次：2018 年 11 月第 1 版
印　　次：2018 年 11 月第 1 次印刷
定　　价：89.00 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）88254888，88258888。

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。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

业界好评

Java 的并发编程太重要，又太迷人，所以自 Goetz 的 Java Concurrency in Practice 在 2006 年出版，2012 年重译以来，国内的众多作者又陆陆续续出版了若干本相关主题的书籍。那我手上的这本，是又一本 Java 并发编程 (Yet Another ...) 吗？为了找一个大家再次购买的理由，我快速翻完了全书。

显而易见，作者是一位喜欢用代码说话的同学，第一部分基础知识中的每个知识点都伴随一段简短的示例及证明的代码，代码不撒谎。

作者对代码的爱，也带到了第二部分。书中针对 Java 并发库中的主要组件，进行了代码级的原理讲解，而且紧贴时代脉搏，涵盖了 JDK 8 的内容。如果你能耐下心来，跟随作者进行一番代码级的探究，所产生的印象比阅读文章、死记结论，无疑要深刻得多。

到了最后的实践部分，依然没有模式、架构之类的宏大叙事，而是作者自己一个个的实践例子。

所以，如果要简单概括，这就是一本有好奇心的 Coder，写给另一位有好奇心的 Coder 的 Java 并发编程书。

——肖桦（江南白衣），唯品会资深架构师，公众号“春天的旁边”

JDK 1.5 之前，我们必须自己编写代码实现一些并发编程的逻辑；之后到了 JDK 1.5，Doug Lea 解救了广大 Java 用户，在 JDK 里特意设计并实现了一套 JUC 的框架，给大家提供了非常好的并发编程体验。本书作者在阿里经历过大量并发的场景，积攒了不少并发编程的经验，并毫无保留地写入本书。通过书中对 JUC 源码的解读，读者可以揭开 JUC 的神秘面纱。这是一本值得仔细品读的好书。

——你假笨 / 寒泉子，PerfMa CEO，公众号“你假笨”

Java 并发编程所涉及的知识点比较多，多线程编程所考虑的场景相对比较复杂，包括线程间的资源共享、竞争、死锁等问题。并发编程相比 Java 中其他知识点，学习起来门槛相对较高，学习难度较大，从而导致很多人望而却步。加多的《Java 并发编程之美》这本书刚好填补了这个空缺，作者在并发编程领域深耕多年。本书用浅显易懂的文字为大家系统地介绍了 Java 并发编程的相关内容，推荐大家关注学习。

——纯洁的微笑，第三方支付公司技术总监，公众号“纯洁的微笑”

Java 并发编程无处不在，Java 多线程、并发处理是深入学习 Java 必须要掌握的技术。本书涵盖了 Java 并发包中的核心类、API 以及框架等内容，并辅以详尽的案例讲解，帮助读者快速学习、迅速掌握。如果你希望成长为一名优秀的 Java 程序员，有必要读一读本书。

——许令波，《深入分析 Java Web 技术内幕》作者

第一作者加多是一位非常勤奋的技术人员，经常发布各种技术文章，有时候甚至能做到每天一篇，在并发编程网已经累计发布了近百篇文章。本书是他多年的积累，厚积薄发，从并发编程的基础知识一直到实战娓娓道来，希望读者喜欢。

——方腾飞，并发编程网创始人

前　言

本书特色

不像其他并发类书籍那样晦涩难懂，本书的特色之一是通俗易懂，对 Java 有一定基础的开发人员都可以看懂。本书在基础篇专门讲解并发编程基础，笔者根据在项目实践中对这些知识的理解，总结了并发编程中常用的基础知识以及常用的概念，并通过图文结合的方式降低理解的难度，使用少量的代码讲解就可以让读者轻松掌握并发编程的基础知识，让读者逐步建立起自信。在高级篇主要讲解 JUC 并发包下并发组件的实现原理，首先介绍 JUC 里面最简单的原子类，让读者学会使用在基础篇里介绍的最简单的 CAS 操作，再逐步加大难度让读者慢慢适应：比如一开始打算把并发 List 放到锁后面讲解，因为并发 List 里面使用了锁，但是锁的理解难度比 List 大太多，所以最终还是坚持从易入难的原则，先讲解 List，再讲解锁。在实践篇中首先讲解并发组件在开源框架或者项目中的运用，让读者不仅可以知道并发组件的原理，而且可以了解怎么使用这些组件。最后总结笔者在项目中或者其他同事在项目中经常遇到的并发编程问题，并对其进行分析，给出解决方案。

如果你只想使用并发包，那么可以阅读本书，因为本书在讲解代码时基本都是用的实例；如果你想研究源码却一筹莫展——不知道如何下手或者感觉吃力，也可以阅读本书，因为本书对核心代码进行了讲解；如果你想了解并发编程中的常见问题，增加对并发的认识，也可以阅读本书，因为本书对这类问题进行了总结。

如何阅读本书

本书分为基础篇、高级篇和实践篇，其中基础篇讲解线程的知识和并发编程中的基本概念以及基础知识，高级篇则介绍并发包下常用的并发组件的原理，实践篇讲解并发组件的具体使用方法和在并发编程中会遇到的一些并发问题及解决方法。

阅读开源框架源码的一点心得

为什么要看源码

我们在做项目的时候一般会遇到下面的问题：

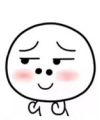
（1）不知道如何去设计。比如刚入职场时，来一个需求需做概要设计，不知如何下手，不得不去看当前系统类似需求是如何设计的，然后仿照去设计。

（2）设计的时候，考虑问题不周全。相比职场新手，这类人对一个需求依靠自己的经验已经能够拿出一个概要设计，但是设计中经常会遗漏一些异常细节，比如使用多线程有界队列执行任务，遇到机器宕机了，如果队列里面的任务不存盘的话，那么机器下次启动的时候这些任务就丢失了。

对于这些问题，说到底主要还是因为经验不够，而经验主要从项目实践中积累，所以招聘单位一般都会限定工作时间大于 3 年，因为这些人的项目经验相对较丰富，在项目中遇到的场景相对较多。工作经验的积累来自于年限与实践，然而看源码可以扩展我们的思路，这是变相增加我们经验的不错方法。虽然不能在短时间内通过时间积累经验，但是可以通过学习开源框架、开源项目来获取经验。

另外，进职场后一般都要先熟悉现有系统，如果有文档还好，没文档的话就得自己去翻代码研究。如果之前对阅读源码有经验，那么在研究新系统的代码逻辑时就不会那么费劲了。

还有一点就是，当你使用框架或者工具做开发时，如果你对它的实现有所了解，就能最大化地减少出故障的可能。比如并发队列 ArrayBlockingQueue 里面关于元素入队有个 offer 方法 和 put 方法，虽然某个时间点你知道使用 offer 方法时，当队列满了就会丢弃要入队的元素，之后 offer 方法会返回 false，而不会阻塞当前线程；而使用 put 方法时，当队列满了，则会挂起当前线程，直到队列有空闲，元素入队成功后才返回。但是人是善忘的，一段时间不使用，就会忘记它们的区别，当你再去使用时，需进入 offer 和 put 方法的内部，看它们的源码实现。进入 offer 方法一看，哦，原来队列满后直接返回了 false；进入 put 方法一看，哦，原来队列满后，直接使用条件变量的 await 方法挂起了当前线程。知道了它们的区别，你就可以根据自己的需求来选择了。



看源码最大的好处是可以开阔思维，提升架构设计能力。有些东西仅靠书本和自己思考是很难学到的，必须通过看源码，看别人如何设计，然后思考为何这样设计才能领悟到。能力的提高不在于你写了多少代码，做了多少项目，而在于给你一个业务场景时，你是否能拿出几种靠谱的解决方案，并且说出各自的优缺点。而如何才能拿出来，一来靠经验，二来靠归纳总结，而看源码可以快速增加你的经验。

如何看源码

那么如何阅读源码呢？在你看某一个框架的源码前，先去 Google 查找这个开源框架的官方介绍，通过资料了解该框架有几个模块，各个模块是做什么的，之间有什么联系，每个模块都有哪些核心类，在阅读源码时可以着重看这些类。

然后对哪个模块感兴趣就去写个小 demo，先了解一下这个模块的具体作用，然后再 debug 进入看具体实现。在 debug 的过程中，第一遍是走马观花，简略看一下调用逻辑，都用了哪些类；第二遍需有重点地 debug，看看这些类担任了架构图里的哪些功能，使用了哪些设计模式。如果第二遍有感觉了，便大致知道了整体代码的功能实现，但是对整体代码结构还不是很清晰，毕竟代码里面多个类来回调用，很容易遗忘当前断点的来处；那么你可以进行第三遍 debug，这时候你最好把主要类的调用时序图以及类图结构画出来，等画好后，再对着时序图分析调用流程，就可以清楚地知道类之间的调用关系，而通过类图可以知道类的功能以及它们相互之间的依赖关系。

另外，开源框架里面每个功能类或者方法一般都有注释，这些注释是一手资料，比如 JUC 包里的一些并发组件的注释，就已经说明了它们的设计原理和使用场景。

在阅读源码时，最好画出时序图和类图，因为人总是善忘的。如果隔一段时间你再去看之前看过的源码，虽然有些印象，但当你想去看某个模块的逻辑时，又需根据 demo 再从头 debug 了。而如果有了这俩图，就可以从这俩图里面直接找，并且看一眼时序图就知道整个模块的脉络了。

此外，查框架使用说明最好去官网查（这些信息是源头，是没有经过别人翻译的），虽然是英文，但是看久了就好了，毕竟还有 Google 翻译呐！

当然研究代码时不一定非要 debug 三遍，其实这里说的是三种掌握程度，如果你 debug 一遍就能掌握，那自然更好啦。

目　录

第一部分　Java 并发编程基础篇

第三部分　Java 并发编程实践篇

第一部分

Java并发编程基础篇

本篇主要介绍并发编程的基础知识，包含两章内容，分别为并发编程线程基础以及并发编程的其他概念与原理解析。

第1章

并发编程线程基础

1.1 什么是线程

在讨论什么是线程前有必要先说下什么是进程，因为线程是进程中的一个实体，线程本身是不会独立存在的。进程是代码在数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，线程则是进程的一个执行路径，一个进程中至少有一个线程，进程中的多个线程共享进程的资源。

操作系统在分配资源时是把资源分配给进程的，但是 CPU 资源比较特殊，它是被分配到线程的，因为真正要占用 CPU 运行的是线程，所以也说线程是 CPU 分配的基本单位。

在 Java 中，当我们启动 main 函数时其实就启动了一个 JVM 的进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程。

进程和线程的关系如图 1-1 所示。

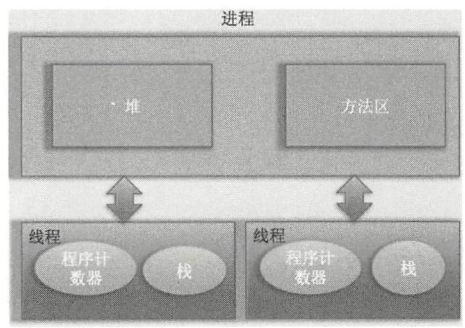


图 1-1

由图 1-1 可以看到，一个进程中有多个线程，多个线程共享进程的堆和方法区资源，但是每个线程有自己的程序计数器和栈区域。

程序计数器是一块内存区域，用来记录线程当前要执行的指令地址。那么为何要将程序计数器设计为线程私有的呢？前面说了线程是占用 CPU 执行的基本单位，而 CPU 一般是使用时间片轮转方式让线程轮询占用的，所以当前线程 CPU 时间片用完后，要让出 CPU，等下次轮到自己的时候再执行。那么如何知道之前程序执行到哪里了呢？其实程序计数器就是为了记录该线程让出 CPU 时的执行地址的，待再次分配到时间片时线程就可以从自己私有的计数器指定地址继续执行。另外需要注意的是，如果执行的是 native 方法，那么 pc 计数器记录的是 undefined 地址，只有执行的是 Java 代码时 pc 计数器记录的才是下一条指令的地址。

另外每个线程都有自己的栈资源，用于存储该线程的局部变量，这些局部变量是该线程私有的，其他线程是访问不了的，除此之外栈还用来存放线程的调用栈帧。

堆是一个进程中最大的一块内存，堆是被进程中的所有线程共享的，是进程创建时分配的，堆里面主要存放使用 new 操作创建的对象实例。

方法区则用来存放 JVM 加载的类、常量及静态变量等信息，也是线程共享的。

1.2　线程创建与运行

Java 中有三种线程创建方式，分别为实现 Runnable 接口的 run 方法，继承 Thread 类并重写 run 的方法，使用 FutureTask 方式。

首先看继承 Thread 类方式的实现。

```
public class ThreadTest {

    //继承Thread类并重写run方法
    public static class MyThread extends Thread {

        @Override
        public void run() {

            System.out.println("I am a child thread");

        }
```

```
    }

    public static void main(String[] args) {

        // 创建线程
        MyThread thread = new MyThread();

        // 启动线程
        thread.start();
    }
}
```

如上代码中的 MyThread 类继承了 Thread 类，并重写了 run() 方法。在 main 函数里面创建了一个 MyThread 的实例，然后调用该实例的 start 方法启动了线程。需要注意的是，当创建完 thread 对象后该线程并没有被启动执行，直到调用了 start 方法后才真正启动了线程。

其实调用 start 方法后线程并没有马上执行而是处于就绪状态，这个就绪状态是指该线程已经获取了除 CPU 资源外的其他资源，等待获取 CPU 资源后才会真正处于运行状态。一旦 run 方法执行完毕，该线程就处于终止状态。

使用继承方式的好处是，在 run() 方法内获取当前线程直接使用 this 就可以了，无须使用 Thread.currentThread() 方法；不好的地方是 Java 不支持多继承，如果继承了 Thread 类，那么就不能再继承其他类。另外任务与代码没有分离，当多个线程执行一样的任务时需要多份任务代码，而 Runable 则没有这个限制。下面看实现 Runnable 接口的 run 方法方式。

```
    public static class RunableTask implements Runnable{

        @Override
        public void run() {
            System.out.println("I am a child thread");
        }

    }
public static void main(String[] args) throws InterruptedException{

        RunableTask task = new RunableTask();
        new Thread(task).start();
        new Thread(task).start();
}
```

如上面代码所示，两个线程共用一个 task 代码逻辑，如果需要，可以给 RunableTask 添加参数进行任务区分。另外，RunableTask 可以继承其他类。但是上面介绍的两种方式都有一个缺点，就是任务没有返回值。下面看最后一种，即使用 FutureTask 的方式。

```
//创建任务类，类似Runable
public static class CallerTask implements Callable<String>{

        @Override
        public String call() throws Exception {

            return "hello";
        }

    }

    public static void main(String[] args) throws InterruptedException {
    // 创建异步任务
        FutureTask<String> futureTask  = new FutureTask<>(new CallerTask());
        //启动线程
        new Thread(futureTask).start();
        try {
           //等待任务执行完毕，并返回结果
            String result = futureTask.get();
            System.out.println(result);
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
}
```

如上代码中的 CallerTask 类实现了 Callable 接口的 call() 方法。在 main 函数内首先创建了一个 FutrueTask 对象（构造函数为 CallerTask 的实例），然后使用创建的 FutrueTask 对象作为任务创建了一个线程并且启动它，最后通过 futureTask.get() 等待任务执行完毕并返回结果。

小结：使用继承方式的好处是方便传参，你可以在子类里面添加成员变量，通过 set 方法设置参数或者通过构造函数进行传递，而如果使用 Runnable 方式，则只能使用主线程里面被声明为 final 的变量。不好的地方是 Java 不支持多继承，如果继承了 Thread 类，那么子类不能再继承其他类，而 Runable 则没有这个限制。前两种方式都没办法拿到任务的返回结果，但是 Futuretask 方式可以。

1.3 线程通知与等待

Java 中的 Object 类是所有类的父类，鉴于继承机制，Java 把所有类都需要的方法放到了 Object 类里面，其中就包含本节要讲的通知与等待系列函数。

1. wait() 函数

当一个线程调用一个共享变量的 wait() 方法时，该调用线程会被阻塞挂起，直到发生下面几件事情之一才返回：(1)其他线程调用了该共享对象的 notify() 或者 notifyAll() 方法；(2) 其他线程调用了该线程的 interrupt() 方法，该线程抛出 InterruptedException 异常返回。

另外需要注意的是，如果调用 wait() 方法的线程没有事先获取该对象的监视器锁，则调用 wait() 方法时调用线程会抛出 IllegalMonitorStateException 异常。

那么一个线程如何才能获取一个共享变量的监视器锁呢？

(1) 执行 synchronized 同步代码块时，使用该共享变量作为参数。

```
synchronized (共享变量) {
        //doSomething
    }
```

(2) 调用该共享变量的方法，并且该方法使用了 synchronized 修饰。

```
synchronized void add(int a,int b){
        //doSomething
}
```

另外需要注意的是，一个线程可以从挂起状态变为可以运行状态（也就是被唤醒），即使该线程没有被其他线程调用 notify()、notifyAll() 方法进行通知，或者被中断，或者等待超时，这就是所谓的虚假唤醒。

虽然虚假唤醒在应用实践中很少发生，但要防患于未然，做法就是不停地去测试该线程被唤醒的条件是否满足，不满足则继续等待，也就是说在一个循环中调用 wait() 方法进行防范。退出循环的条件是满足了唤醒该线程的条件。

```
synchronized (obj) {
        while (条件不满足){
          obj.wait();
        }
}
```

如上代码是经典的调用共享变量 wait() 方法的实例，首先通过同步块获取 obj 上面的监视器锁，然后在 while 循环内调用 obj 的 wait() 方法。

下面从一个简单的生产者和消费者例子来加深理解。如下面代码所示，其中 queue 为共享变量，生产者线程在调用 queue 的 wait() 方法前，使用 synchronized 关键字拿到了该共享变量 queue 的监视器锁，所以调用 wait() 方法才不会抛出 IllegalMonitorStateException 异常。如果当前队列没有空闲容量则会调用 queued 的 wait() 方法挂起当前线程，这里使用循环就是为了避免上面说的虚假唤醒问题。假如当前线程被虚假唤醒了，但是队列还是没有空余容量，那么当前线程还是会调用 wait() 方法把自己挂起。

```
//生产线程
synchronized (queue) {

    //消费队列满，则等待队列空闲
    while (queue.size() == MAX_SIZE) {
        try {
            //挂起当前线程，并释放通过同步块获取的queue上的锁，让消费者线程可以获取该锁，然后
              获取队列里面的元素
            queue.wait();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    //空闲则生成元素，并通知消费者线程
    queue.add(ele);
    queue.notifyAll();

    }
}
//消费者线程
synchronized (queue) {

    //消费队列为空
    while (queue.size() == 0) {
        try
            //挂起当前线程，并释放通过同步块获取的queue上的锁，让生产者线程可以获取该锁，将生
              产元素放入队列
            queue.wait();
```

```
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        //消费元素，并通知唤醒生产者线程
        queue.take();
        queue.notifyAll();

        }
}
```

在如上代码中假如生产者线程 A 首先通过 synchronized 获取到了 queue 上的锁，那么后续所有企图生产元素的线程和消费线程将会在获取该监视器锁的地方被阻塞挂起。线程 A 获取锁后发现当前队列已满会调用 queue.wait() 方法阻塞自己，然后释放获取的 queue 上的锁，这里考虑下为何要释放该锁？如果不释放，由于其他生产者线程和所有消费者线程都已经被阻塞挂起，而线程 A 也被挂起，这就处于了死锁状态。这里线程 A 挂起自己后释放共享变量上的锁，就是为了打破死锁必要条件之一的持有并等待原则。关于死锁后面的章节会讲。线程 A 释放锁后，其他生产者线程和所有消费者线程中会有一个线程获取 queue 上的锁进而进入同步块，这就打破了死锁状态。

另外需要注意的是，当前线程调用共享变量的 wait() 方法后只会释放当前共享变量上的锁，如果当前线程还持有其他共享变量的锁，则这些锁是不会被释放的。下面来看一个例子。

```
// 创建资源
private static volatile  Object resourceA = new Object();
private static volatile Object resourceB = new Object();

public static void main(String[] args) throws InterruptedException {

    // 创建线程
    Thread threadA = new Thread(new Runnable() {
        public void run() {

            try {

                // 获取resourceA共享资源的监视器锁
                synchronized (resourceA) {
                    System.out.println("threadA get resourceA lock");
```

```
                // 获取resourceB共享资源的监视器锁
                synchronized (resourceB) {
                    System.out.println("threadA get resourceB lock");

                    // 线程A阻塞，并释放获取到的resourceA的锁
                    System.out.println("threadA release resourceA lock");
                    resourceA.wait();

                }

            }

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

// 创建线程
Thread threadB = new Thread(new Runnable() {
    public void run() {

        try {

      //休眠1s
            Thread.sleep(1000);

            // 获取resourceA共享资源的监视器锁
            synchronized (resourceA) {
                System.out.println("threadB get resourceA lock");

                System.out.println("threadB try get resourceB lock...");

                // 获取resourceB共享资源的监视器锁
                synchronized (resourceB) {
                    System.out.println("threadB get resourceB lock");

                    // 线程B阻塞，并释放获取到的resourceA的锁
                    System.out.println("threadB release resourceA lock");
                    resourceA.wait();

                }
```

```
                }

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });

    // 启动线程
    threadA.start();
    threadB.start();

    // 等待两个线程结束
    threadA.join();
    threadB.join();

    System.out.println("main over");

}
```

输出结果如下：

```
WaitNotifyTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
threadA get resourceA lock
threadA get resourceB lock
threadA release resourceA lock
threadB get resourceA lock
threadB try get resourceB lock...
```

如上代码中，在 main 函数里面启动了线程 A 和线程 B，为了让线程 A 先获取到锁，这里让线程 B 先休眠了 1s，线程 A 先后获取到共享变量 resourceA 和共享变量 resourceB 上的锁，然后调用了 resourceA 的 wait() 方法阻塞自己，阻塞自己后线程 A 释放掉获取的 resourceA 上的锁。

线程 B 休眠结束后会首先尝试获取 resourceA 上的锁，如果当时线程 A 还没有调用 wait() 方法释放该锁，那么线程 B 会被阻塞，当线程 A 释放了 resourceA 上的锁后，线程 B 就会获取到 resourceA 上的锁，然后尝试获取 resourceB 上的锁。由于线程 A 调用的是 resourceA 上的 wait() 方法，所以线程 A 挂起自己后并没有释放获取到的 resourceB 上的锁，所以线程 B 尝试获取 resourceB 上的锁时会被阻塞。

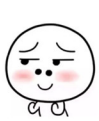
这就证明了当线程调用共享对象的 wait() 方法时，当前线程只会释放当前共享对象的

锁，当前线程持有的其他共享对象的监视器锁并不会被释放。

最后再举一个例子进行说明。当一个线程调用共享对象的 wait() 方法被阻塞挂起后，如果其他线程中断了该线程，则该线程会抛出 InterruptedException 异常并返回。

```
public class WaitNotifyInterupt {

    static Object obj = new Object();

    public static void main(String[] args) throws InterruptedException {

        //创建线程
        Thread threadA = new Thread(new Runnable() {
            public void run() {
                try {
                    System.out.println("---begin---");
                    //阻塞当前线程
                    synchronized (obj) {
                       obj.wait();
                    }
                    System.out.println("---end---");

                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        threadA.start();

        Thread.sleep(1000);

        System.out.println("---begin interrupt threadA---");
        threadA.interrupt();
        System.out.println("---end interrupt threadA---");
    }
}
```

输出如下。



```
<terminated> WaitNotifyInterupt [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2018年8月8日 下午2:01:14)
---begin---
---begin interrupt threadA---
---end interrupt threadA---
java.lang.InterruptedException
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Object.java:502)
        at com.gitchat.demo.netty_learn.WaitNotifyInterupt$1.run(WaitNotifyInterupt.java:16)
        at java.lang.Thread.run(Thread.java:745)
```

在如上代码中，threadA 调用共享对象 obj 的 wait() 方法后阻塞挂起了自己，然后主线程在休眠 1s 后中断了 threadA 线程，中断后 threadA 在 obj.wait() 处抛出 java.lang.InterruptedException 异常而返回并终止。

2. wait(long timeout) 函数

该方法相比 wait() 方法多了一个超时参数，它的不同之处在于，如果一个线程调用共享对象的该方法挂起后，没有在指定的 timeout ms 时间内被其他线程调用该共享变量的 notify() 或者 notifyAll() 方法唤醒，那么该函数还是会因为超时而返回。如果将 timeout 设置为 0 则和 wait 方法效果一样，因为在 wait 方法内部就是调用了 wait(0)。需要注意的是，如果在调用该函数时，传递了一个负的 timeout 则会抛出 IllegalArgumentException 异常。

3. wait(long timeout, int nanos) 函数

在其内部调用的是 wait(long timeout) 函数，如下代码只有在 nanos>0 时才使参数 timeout 递增 1。

```
public final void wait(long timeout, int nanos) throws InterruptedException {
        if (timeout < 0) {
            throw new IllegalArgumentException("timeout value is negative");
        }

        if (nanos < 0 || nanos > 999999) {
            throw new IllegalArgumentException(
                                "nanosecond timeout value out of range");
        }

        if (nanos > 0) {
            timeout++;
        }

        wait(timeout);
    }
```

4. notify() 函数

一个线程调用共享对象的 notify() 方法后，会唤醒一个在该共享变量上调用 wait 系列方法后被挂起的线程。一个共享变量上可能会有多个线程在等待，具体唤醒哪个等待的线程是随机的。

此外，被唤醒的线程不能马上从 wait 方法返回并继续执行，它必须在获取了共享对象的监视器锁后才可以返回，也就是唤醒它的线程释放了共享变量上的监视器锁后，被唤醒的线程也不一定会获取到共享对象的监视器锁，这是因为该线程还需要和其他线程一起竞争该锁，只有该线程竞争到了共享变量的监视器锁后才可以继续执行。

类似 wait 系列方法，只有当前线程获取到了共享变量的监视器锁后，才可以调用共享变量的 notify() 方法，否则会抛出 IllegalMonitorStateException 异常。

5. notifyAll() 函数

不同于在共享变量上调用 notify() 函数会唤醒被阻塞到该共享变量上的一个线程，notifyAll() 方法则会唤醒所有在该共享变量上由于调用 wait 系列方法而被挂起的线程。

下面举一个例子来说明 notify() 和 notifyAll() 方法的具体含义及一些需要注意的地方，代码如下。

```
// 创建资源
private static volatile Object resourceA = new Object();

public static void main(String[] args) throws InterruptedException {

    // 创建线程
    Thread threadA = new Thread(new Runnable() {
        public void run() {

            // 获取resourceA共享资源的监视器锁
            synchronized (resourceA) {

                System.out.println("threadA get resourceA lock");
                try {

                    System.out.println("threadA begin wait");
                    resourceA.wait();
                    System.out.println("threadA end wait");
```

```
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
    });

    // 创建线程
    Thread threadB = new Thread(new Runnable() {
        public void run() {

            synchronized (resourceA) {
                System.out.println("threadB get resourceA lock");
                try {

                    System.out.println("threadB begin wait");
                    resourceA.wait();
                    System.out.println("threadB end wait");

                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

    });

    // 创建线程
    Thread threadC = new Thread(new Runnable() {
        public void run() {

            synchronized (resourceA) {

                System.out.println("threadC begin notify");
                resourceA.notify();
            }
        }
    });

    // 启动线程
    threadA.start();
```

```
        threadB.start();

        Thread.sleep(1000);
        threadC.start();

        // 等待线程结束
        threadA.join();
        threadB.join();
        threadC.join();

        System.out.println("main over");

}
```

输出结果如下。

```
WaitNotifyAllTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
threadA get resourceA lock
threadA begin wait
threadB get resourceA lock
threadB begin wait
threadC begin notify
threadA end wait
```

如上代码开启了三个线程，其中线程 A 和线程 B 分别调用了共享资源 resourceA 的 wait() 方法，线程 C 则调用了 nofity() 方法。这里启动线程 C 前首先调用 sleep 方法让主线程休眠 1s，这样做的目的是让线程 A 和线程 B 全部执行到调用 wait 方法后再调用线程 C 的 notify 方法。这个例子试图在线程 A 和线程 B 都因调用共享资源 resourceA 的 wait() 方法而被阻塞后，让线程 C 再调用 resourceA 的 notify() 方法，从而唤醒线程 A 和线程 B。但是从执行结果来看，只有一个线程 A 被唤醒，线程 B 没有被唤醒：

从输出结果可知线程调度器这次先调度了线程 A 占用 CPU 来运行，线程 A 首先获取 resourceA 上面的锁，然后调用 resourceA 的 wait() 方法挂起当前线程并释放获取到的锁，然后线程 B 获取到 resourceA 上的锁并调用 resourceA 的 wait() 方法，此时线程 B 也被阻塞挂起并释放了 resourceA 上的锁，到这里线程 A 和线程 B 都被放到了 resourceA 的阻塞集合里面。线程 C 休眠结束后在共享资源 resourceA 上调用了 notify() 方法，这会激活 resourceA 的阻塞集合里面的一个线程，这里激活了线程 A，所以线程 A 调用的 wait() 方法返回了，线程 A 执行完毕。而线程 B 还处于阻塞状态。如果把线程 C 调用的 notify() 方法改为调用 notifyAll() 方法，则执行结果如下。

```
<terminated> WaitNotifyAllTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
threadA get resourceA lock
threadA begin wait
threadB get resourceA lock
threadB begin wait
threadc begin notify
threadB end wait
threadA end wait
main over
```

从输入结果可知线程 A 和线程 B 被挂起后，线程 C 调用 notifyAll() 方法会唤醒 resourceA 的等待集合里面的所有线程，这里线程 A 和线程 B 都会被唤醒，只是线程 B 先获取到 resourceA 上的锁，然后从 wait() 方法返回。线程 B 执行完毕后，线程 A 又获取了 resourceA 上的锁，然后从 wait() 方法返回。线程 A 执行完毕后，主线程返回，然后打印输出。

一个需要注意的地方是，在共享变量上调用 notifyAll() 方法只会唤醒调用这个方法前调用了 wait 系列函数而被放入共享变量等待集合里面的线程。如果调用 notifyAll() 方法后一个线程调用了该共享变量的 wait() 方法而被放入阻塞集合，则该线程是不会被唤醒的。尝试把主线程里面休眠 1s 的代码注释掉，再运行程序会有一定概率输出下面的结果。

```
WaitNotifyAllTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/jav
threadA get resourceA lock
threadA begin wait
threadC begin notify
threadB get resourceA lock
threadB begin wait
threadA end wait
```

也就是在线程 B 调用共享变量的 wait() 方法前线程 C 调用了共享变量的 notifyAll 方法，这样，只有线程 A 被唤醒，而线程 B 并没有被唤醒，还是处于阻塞状态。

1.4 等待线程执行终止的 join 方法

在项目实践中经常会遇到一个场景，就是需要等待某几件事情完成后才能继续往下执行，比如多个线程加载资源，需要等待多个线程全部加载完毕再汇总处理。Thread 类中有一个 join 方法就可以做这个事情，前面介绍的等待通知方法是 Object 类中的方法，而 join 方法则是 Thread 类直接提供的。join 是无参且返回值为 void 的方法。下面来看一个简单的例子。

```
public static void main(String[] args) throws InterruptedException {
```

```
Thread threadOne = new Thread(new Runnable() {

    @Override
    public void run() {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("child threadOne over!");

    }
});

Thread threadTwo = new Thread(new Runnable() {

    @Override
    public void run() {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("child threadTwo over!");


    }
});


//启动子线程
threadOne.start();
threadTwo.start();

System.out.println("wait all child thread over!");

//等待子线程执行完毕，返回
threadOne.join();
threadTwo.join();
```

```
        System.out.println("all child thread over!");
    }
```

如上代码在主线程里面启动了两个子线程，然后分别调用了它们的 join() 方法，那么主线程首先会在调用 threadOne.join() 方法后被阻塞，等待 threadOne 执行完毕后返回。threadOne 执行完毕后 threadOne.join() 就会返回，然后主线程调用 threadTwo.join() 方法后再次被阻塞，等待 threadTwo 执行完毕后返回。这里只是为了演示 join 方法的作用，在这种情况下使用后面会讲到的 CountDownLatch 是个不错的选择。

另外，线程 A 调用线程 B 的 join 方法后会被阻塞，当其他线程调用了线程 A 的 interrupt() 方法中断了线程 A 时，线程 A 会抛出 InterruptedException 异常而返回。下面通过一个例子来加深理解。

```
public static void main(String[] args) throws InterruptedException {

    //线程one
    Thread threadOne = new Thread(new Runnable() {

        @Override
        public void run() {

            System.out.println("threadOne begin run!");
            for (;;) {
            }

        }
    });
    //获取主线程
    final Thread mainThread = Thread.currentThread();

    //线程two
    Thread threadTwo = new Thread(new Runnable() {

        @Override
        public void run() {
            //休眠1s
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
            //中断主线程
            mainThread.interrupt();

        }
    });

    // 启动子线程
    threadOne.start();

    //延迟1s启动线程
    threadTwo.start();


    try{//等待线程one执行结束
        threadOne.join();

    }catch(InterruptedException e){
        System.out.println("main thread:" + e);
    }

}
```

输出结果如下。

```
JoinInterruptedExceptionTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Co
threadOne begin run!
main thread:java.lang.InterruptedException
```

如上代码在threadOne线程里面执行死循环，主线程调用threadOne的join方法阻塞自己等待线程threadOne执行完毕，待threadTwo休眠1s后会调用主线程的interrupt()方法设置主线程的中断标志，从结果看在主线程中的threadOne.join()处会抛出InterruptedException异常。这里需要注意的是，在threadTwo里面调用的是主线程的interrupt()方法，而不是线程threadOne的。

1.5　让线程睡眠的sleep方法

Thread类中有一个静态的sleep方法，当一个执行中的线程调用了Thread的sleep方法后，调用线程会暂时让出指定时间的执行权，也就是在这期间不参与CPU的调度，但是该线程所拥有的监视器资源，比如锁还是持有不让出的。指定的睡眠时间到了后该函数

会正常返回，线程就处于就绪状态，然后参与 CPU 的调度，获取到 CPU 资源后就可以继续运行了。如果在睡眠期间其他线程调用了该线程的 interrupt() 方法中断了该线程，则该线程会在调用 sleep 方法的地方抛出 InterruptedException 异常而返回。

下面举一个例子来说明，线程在睡眠时拥有的监视器资源不会被释放。

```
public class SleepTest2 {

    // 创建一个独占锁
    private static final Lock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {

        // 创建线程A
        Thread threadA = new Thread(new Runnable() {
            public void run() {
                // 获取独占锁
                lock.lock();
                try {
                    System.out.println("child threadA is in sleep");

                    Thread.sleep(10000);

                    System.out.println("child threadA is in awaked");

                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    // 释放锁
                    lock.unlock();
                }
            }
        });

        // 创建线程B
        Thread threadB = new Thread(new Runnable() {
            public void run() {
                // 获取独占锁
                lock.lock();
                try {
                    System.out.println("child threadB is in sleep");

                    Thread.sleep(10000);
```

```
                System.out.println("child threadB is in awaked");

            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                // 释放锁
                lock.unlock();
            }
        }
    });

    // 启动线程
    threadA.start();
    threadB.start();

  }

}
```

执行结果如下。

```
<terminated> SleepTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
child threadA is in sleep
child threadA is in awaked
child threadB is in sleep
child threadB is in awaked
```

如上代码首先创建了一个独占锁，然后创建了两个线程，每个线程在内部先获取锁，然后睡眠，睡眠结束后会释放锁。首先，无论你执行多少遍上面的代码都是线程 A 先输出或者线程 B 先输出，不会出现线程 A 和线程 B 交叉输出的情况。从执行结果来看，线程 A 先获取了锁，那么线程 A 会先输出一行，然后调用 sleep 方法让自己睡眠 10s，在线程 A 睡眠的这 10s 内那个独占锁 lock 还是线程 A 自己持有，线程 B 会一直阻塞直到线程 A 醒来后执行 unlock 释放锁。下面再来看一下，当一个线程处于睡眠状态时，如果另外一个线程中断了它，会不会在调用 sleep 方法处抛出异常。

```
public static void main(String[] args) throws InterruptedException {

    //创建线程
    Thread thread = new Thread(new  Runnable() {
        public void run() {

            try {
```

```
                System.out.println("child thread is in sleep");

                Thread.sleep(10000);
                System.out.println("child thread is in awaked");

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });

    //启动线程
    thread.start();

    //主线程休眠2s
    Thread.sleep(2000);

    //主线程中断子线程
    thread.interrupt();
}
```

执行结果如下。

```
<terminated> SleepTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
child thread is in sleep
java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
        at com.zlx.con.program.example.SleepTest$1.run(SleepTest.java:14)
        at java.lang.Thread.run(Thread.java:745)
```

子线程在睡眠期间，主线程中断了它，所以子线程在调用 sleep 方法处抛出了 InterruptedException 异常。

另外需要注意的是，如果在调用 Thread.sleep(long millis) 时为 millis 参数传递了一个负数，则会抛出 IllegalArgumentException 异常，如下所示。

```
<terminated> SleepTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (2017年8月27
Exception in thread "main" java.lang.IllegalArgumentException: timeout value is negative
        at java.lang.Thread.sleep(Native Method)
        at com.zlx.con.program.example.SleepTest.main(SleepTest.java:23)
```

1.6 让出 CPU 执行权的 yield 方法

Thread 类中有一个静态的 yield 方法，当一个线程调用 yield 方法时，实际就是在暗示线程调度器当前线程请求让出自己的 CPU 使用，但是线程调度器可以无条件忽略这个暗示。我们知道操作系统是为每个线程分配一个时间片来占有 CPU 的，正常情况下当一个线程把分配给自己的时间片使用完后，线程调度器才会进行下一轮的线程调度，而当一个线程调用了 Thread 类的静态方法 yield 时，是在告诉线程调度器自己占有的时间片中还没有使用完的部分自己不想使用了，这暗示线程调度器现在就可以进行下一轮的线程调度。

当一个线程调用 yield 方法时，当前线程会让出 CPU 使用权，然后处于就绪状态，线程调度器会从线程就绪队列里面获取一个线程优先级最高的线程，当然也有可能会调度到刚刚让出 CPU 的那个线程来获取 CPU 执行权。下面举一个例子来加深对 yield 方法的理解。

```
public class YieldTest implements Runnable {

    YieldTest() {

        //创建并启动线程
        Thread t = new Thread(this);
       t.start();
   }

   public void run() {

      for (int i = 0; i < 5; i++) {
         //当i=0时让出CPU执行权，放弃时间片，进行下一轮调度
         if ((i % 5) == 0) {
            System.out.println(Thread.currentThread() + "yield cpu...");

            //当前线程让出CPU执行权，放弃时间片，进行下一轮调度
            // Thread.yield();
         }
      }

      System.out.println(Thread.currentThread() + " is over");
   }

   public static void main(String[] args) {
      new YieldTest();
      new YieldTest();
```

```
        new YieldTest();
    }
}
```

输出结果如下。

```
<terminated> YieldTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/jav
Thread[Thread-0,5,main]yield cpu...
Thread[Thread-0,5,main] is over
Thread[Thread-1,5,main]yield cpu...
Thread[Thread-1,5,main] is over
Thread[Thread-2,5,main]yield cpu...
Thread[Thread-2,5,main] is over
```

如上代码开启了三个线程，每个线程的功能都一样，都是在 for 循环中执行 5 次打印。运行多次后，上面的结果是出现次数最多的。解开 Thread.yield() 注释再执行，结果如下。

```
<terminated> YieldTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[Thread-0,5,main]yield cpu...
Thread[Thread-2,5,main]yield cpu...
Thread[Thread-1,5,main]yield cpu...
Thread[Thread-0,5,main] is over
Thread[Thread-2,5,main] is over
Thread[Thread-1,5,main] is over
```

从结果可知，Thread.yield() 方法生效了，三个线程分别在 i=0 时调用了 Thread.yield() 方法，所以三个线程自己的两行输出没有在一起，因为输出了第一行后当前线程让出了 CPU 执行权。

一般很少使用这个方法，在调试或者测试时这个方法或许可以帮助复现由于并发竞争条件导致的问题，其在设计并发控制时或许会有用途，后面在讲解 java.util.concurrent.locks 包里面的锁时会看到该方法的使用。

总结：sleep 与 yield 方法的区别在于，当线程调用 sleep 方法时调用线程会被阻塞挂起指定的时间，在这期间线程调度器不会去调度该线程。而调用 yield 方法时，线程只是让出自己剩余的时间片，并没有被阻塞挂起，而是处于就绪状态，线程调度器下一次调度时就有可能调度到当前线程执行。

1.7 线程中断

Java 中的线程中断是一种线程间的协作模式，通过设置线程的中断标志并不能直接终

止该线程的执行，而是被中断的线程根据中断状态自行处理。

- **void interrupt() 方法**：中断线程，例如，当线程 A 运行时，线程 B 可以调用线程 A 的 interrupt() 方法来设置线程 A 的中断标志为 true 并立即返回。设置标志仅仅是设置标志，线程 A 实际并没有被中断，它会继续往下执行。如果线程 A 因为调用了 wait 系列函数、join 方法或者 sleep 方法而被阻塞挂起，这时候若线程 B 调用线程 A 的 interrupt() 方法，线程 A 会在调用这些方法的地方抛出 InterruptedException 异常而返回。
- **boolean isInterrupted() 方法**：检测当前线程是否被中断，如果是返回 true，否则返回 false。

```
public boolean isInterrupted() {
      //传递false，说明不清除中断标志
      return isInterrupted(false);
  }
```

- **boolean interrupted() 方法**：检测当前线程是否被中断，如果是返回 true，否则返回 false。与 isInterrupted 不同的是，该方法如果发现当前线程被中断，则会清除中断标志，并且该方法是 static 方法，可以通过 Thread 类直接调用。另外从下面的代码可以知道，在 interrupted() 内部是获取当前调用线程的中断标志而不是调用 interrupted() 方法的实例对象的中断标志。

```
public static boolean interrupted() {
  //清除中断标志
  return currentThread().isInterrupted(true);
}
```

下面看一个线程使用 Interrupted 优雅退出的经典例子，代码如下。

```
public void run(){
    try{
         ....
         //线程退出条件
         while(!Thread.currentThread().isInterrupted()&& more work to do){
                // do more work;
         }
    }catch(InterruptedException e){
                // thread was interrupted during sleep or wait
    }
    finally{
               // cleanup, if required
```

```
    }
}
```

下面看一个根据中断标志判断线程是否终止的例子。

```
public static void main(String[] args) throws InterruptedException {

    Thread thread = new Thread(new Runnable() {

        @Override
        public void run() {

            //如果当前线程被中断则退出循环
            while (!Thread.currentThread().isInterrupted())

                System.out.println(Thread.currentThread() + " hello");
        }
    });

    //启动子线程
    thread.start();

    //主线程休眠1s，以便中断前让子线程输出
    Thread.sleep(1000);

    //中断子线程
    System.out.println("main thread interrupt thread");
    thread.interrupt();

    //等待子线程执行完毕
    thread.join();
    System.out.println("main is over");

}
```

输出结果如下.

```
<terminated> MyThread [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[Thread-0,5,main] hello
main thread interrupt thread
Thread[Thread-0,5,main] hello
main is over
```

在如上代码中，子线程 thread 通过检查当前线程中断标志来控制是否退出循环，主线程在休眠 1s 后调用 thread 的 interrupt() 方法设置了中断标志，所以线程 thread 退出了循环。

下面再来看一种情况。当线程为了等待一些特定条件的到来时，一般会调用 sleep 函数、wait 系列函数或者 join() 函数来阻塞挂起当前线程。比如一个线程调用了 Thread.sleep(3000)，那么调用线程会被阻塞，直到 3s 后才会从阻塞状态变为激活状态。但是有可能在 3s 内条件已被满足，如果一直等到 3s 后再返回有点浪费时间，这时候可以调用该线程的 interrupt() 方法，强制 sleep 方法抛出 InterruptedException 异常而返回，线程恢复到激活状态。下面看一个例子。

```
public static void main(String[] args) throws InterruptedException {

        Thread threadOne = new Thread(new Runnable() {
            public void run() {

                try {
                    System.out.println("threadOne begin sleep for 2000 seconds");
                    Thread.sleep(2000000);
                    System.out.println("threadOne awaking");

                } catch (InterruptedException e) {
                    System.out.println("threadOne is interrupted while sleeping");
                    return;
                }

                System.out.println("threadOne-leaving normally");
            }
        });

        //启动线程
        threadOne.start();

        //确保子线程进入休眠状态
        Thread.sleep(1000);

        //打断子线程的休眠，让子线程从sleep函数返回
        threadOne.interrupt();

        //等待子线程执行完毕
        threadOne.join();

        System.out.println("main thread is over");

}
```

输出结果如下。

```
<terminated> SleepInterruptTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
threadOne begin sleep for 2000 seconds
threadOne is interrupted while sleeping
main thread is over
```

在如上代码中，threadOne 线程休眠了 2000s，在正常情况下该线程需要等到 2000s 后才会被唤醒，但是本例通过调用 threadOne.interrupt() 方法打断了该线程的休眠，该线程会在调用 sleep 方法处抛出 InterruptedException 异常后返回。

下面再通过一个例子来了解 interrupted() 与 isInterrupted() 方法的不同之处。

```
public static void main(String[] args) throws InterruptedException {

    Thread threadOne = new Thread(new Runnable() {
        public void run() {

            for(;;){

            }
        }
    });

    //启动线程
    threadOne.start();

    //设置中断标志
    threadOne.interrupt();

    //获取中断标志
    System.out.println("isInterrupted:" + threadOne.isInterrupted());

    //获取中断标志并重置
    System.out.println("isInterrupted:" + threadOne.interrupted());

    //获取中断标志并重置
    System.out.println("isInterrupted:" + Thread.interrupted());

    //获取中断标志
    System.out.println("isInterrupted:" + threadOne.isInterrupted());

    threadOne.join();
```

```
    System.out.println("main thread is over");

}
```

输出结果如下。

```
<terminated> SleepInterruptTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
isInterrupted:true
isInterrupted:false
isInterrupted:false
isInterrupted:true
```

第一行输出 true 这个大家应该都可以想到，但是下面三行为何是 false、false、true 呢，不应该是 true、false、false 吗？如果你有这个疑问，则说明你对这两个函数的区别还是不太清楚。上面我们介绍了在 interrupted() 方法内部是获取当前线程的中断状态，这里虽然调用了 threadOne 的 interrupted() 方法，但是获取的是主线程的中断标志，因为主线程是当前线程。threadOne.interrupted() 和 Thread.interrupted() 方法的作用是一样的，目的都是获取当前线程的中断标志。修改上面的例子为如下。

```
public static void main(String[] args) throws InterruptedException {

    Thread threadOne = new Thread(new Runnable() {
        public void run() {

            //中断标志为true时会退出循环，并且清除中断标志
            while (!Thread.currentThread().interrupted()) {

            }

            System.out.println("threadOne isInterrupted:" + Thread.currentThread().
isInterrupted());
        }
    });

    // 启动线程
    threadOne.start();

    // 设置中断标志
    threadOne.interrupt();

    threadOne.join();
    System.out.println("main thread is over");
```

```
}
```

输出结果如下。

```
<terminated> SleepInterruptTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
threadOne isInterrupted:false
main thread is over
```

由输出结果可知，调用 interrupted() 方法后中断标志被清除了。

1.8 理解线程上下文切换

在多线程编程中，线程个数一般都大于 CPU 个数，而每个 CPU 同一时刻只能被一个线程使用，为了让用户感觉多个线程是在同时执行的，CPU 资源的分配采用了时间片轮转的策略，也就是给每个线程分配一个时间片，线程在时间片内占用 CPU 执行任务。当前线程使用完时间片后，就会处于就绪状态并让出 CPU 让其他线程占用，这就是上下文切换，从当前线程的上下文切换到了其他线程。那么就有一个问题，让出 CPU 的线程等下次轮到自己占有 CPU 时如何知道自己之前运行到哪里了？所以在切换线程上下文时需要保存当前线程的执行现场，当再次执行时根据保存的执行现场信息恢复执行现场。

线程上下文切换时机有：当前线程的 CPU 时间片使用完处于就绪状态时，当前线程被其他线程中断时。

1.9 线程死锁

1.9.1 什么是线程死锁

死锁是指两个或两个以上的线程在执行过程中，因争夺资源而造成的互相等待的现象，在无外力作用的情况下，这些线程会一直相互等待而无法继续运行下去，如图 1-2 所示。

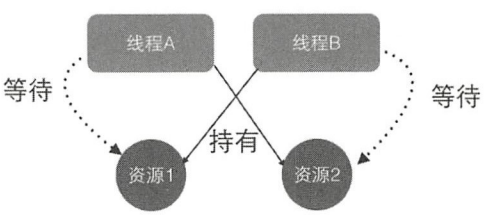


图 1-2