



Java并发编程

极客学院出版

前言

全面记录了 Java 并发编程的相关知识，包括 Java 5 新增加的并发包内的相关类，分析了并发编程中的常见问题，并深入 Java 内存模型，对底层并发机制的实现做了一些分析。如果你正在编写、设计、调试、维护以及分析多线程的 Java 程序，那么本书正是你需要的。

| 适用人群

本书适合 Java 程序开发人员阅读。

| 前提准备

熟悉 Java 语法，了解 Java 面向对象编程思想。

目录

前言	1
第 1 章 并发性与多线程介绍	4
第 2 章 多线程的实现方法	6
第 3 章 线程中断	11
第 4 章 线程挂起、恢复与终止	19
第 5 章 守护线程与线程阻塞	27
第 6 章 Volatile 关键字（上）	29
第 7 章 Volatile 关键字（下）	35
第 8 章 synchronized 关键字	40
第 9 章 synchronized 的另一个重要作用：内存可见性	43
第 10 章 实现内存可见性的两种方法比较：synchronized 和 Volatile	46
第 11 章 多线程环境下安全使用集合 API	48
第 12 章 死锁	51
第 13 章 可重入内置锁	55
第 14 章 线程间协作：wait、notify、notifyAll	57
第 15 章 notify 通知的遗漏	60
第 16 章 notifyAll 造成的早期通知问题	67
第 17 章 生产者—消费者模型	72
第 18 章 深入 Java 内存模型（1）	77
第 19 章 深入 Java 内存模型（2）	84
第 20 章 并发新特性—Executor 框架与线程池	88

第 21 章	并发新特性—Lock 锁与条件变量	99
第 22 章	并发新特性—阻塞队列与阻塞栈.....	113
第 23 章	并发新特性—障碍器 CyclicBarrier	120
第 24 章	并发新特性—信号量 Semaphore	123



1

并发性与多线程介绍



在过去单 CPU 时代，单任务在一个时间点只能执行单一程序。之后发展到多任务阶段，计算机能在同一时间点并行执行多任务或多进程。虽然并不是真正意义上的“同一时间点”，而是多个任务或进程共享一个 CPU，并由操作系统来完成多任务间对 CPU 的运行切换，以使得每个任务都有机会获得一定的时间片运行。

随着多任务对软件开发者带来的新挑战，程序不再能假设独占所有的 CPU 时间、所有的内存和其他计算机资源。一个好的程序榜样是在其不再使用这些资源时对其进行释放，以使得其他程序能有机会使用这些资源。

再后来发展到多线程技术，使得在一个程序内部能拥有多个线程并行执行。一个线程的执行可以被认为是一个 CPU 在执行该程序。当一个程序运行在多线程下，就好像有多个 CPU 在同时执行该程序。

多线程比多任务更加有挑战。多线程是在同一个程序内部并行执行，因此会对相同的内存空间进行并发读写操作。这可能是在单线程程序中从来不会遇到的问题。其中的一些错误也未必会在单 CPU 机器上出现，因为两个线程从来不会得到真正的并行执行。然而，更现代的计算机伴随着多核 CPU 的出现，也就意味着不同的线程能被不同的 CPU 核得到真正意义的并行执行。

如果一个线程在读一个内存时，另一个线程正向该内存进行写操作，那进行读操作的那个线程将获得什么结果呢？是写操作之前旧的值？还是写操作成功之后的新值？或是一半新一半旧的值？或者，如果是两个线程同时写同一个内存，在操作完成后将会是什么结果呢？是第一个线程写入的值？还是第二个线程写入的值？还是两个线程写入的一个混合值？因此如没有合适的预防措施，任何结果都是可能的。而且这种行为的发生甚至不能预测，所以结果也是不确定性的。

Java 的多线程和并发性

Java 是最先支持多线程的开发的语言之一，Java 从一开始就支持了多线程能力，因此 Java 开发者能常遇到上面描述的问题场景。这也是我想为 Java 并发技术而写这篇系列的原因。作为对自己的笔记，和对其他 Java 开发的追随者都可获益的。

该系列主要关注 Java 多线程，但有些在多线程中出现的问题会和多任务以及分布式系统中出现的存在类似，因此该系列会将多任务和分布式系统方面作为参考，所以叫法上称为“并发性”，而不是“多线程”。



2

多线程的实现方法



Java 中实现多线程有两种方法：继承 Thread 类、实现 Runnable 接口，在程序开发中只要是多线程，肯定永远以实现 Runnable 接口为主，因为实现 Runnable 接口相比继承 Thread 类有如下优势：

- 可以避免由于 Java 的单继承特性而带来的局限；
- 增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的；
- 适合多个相同程序代码的线程区处理同一资源的情况。

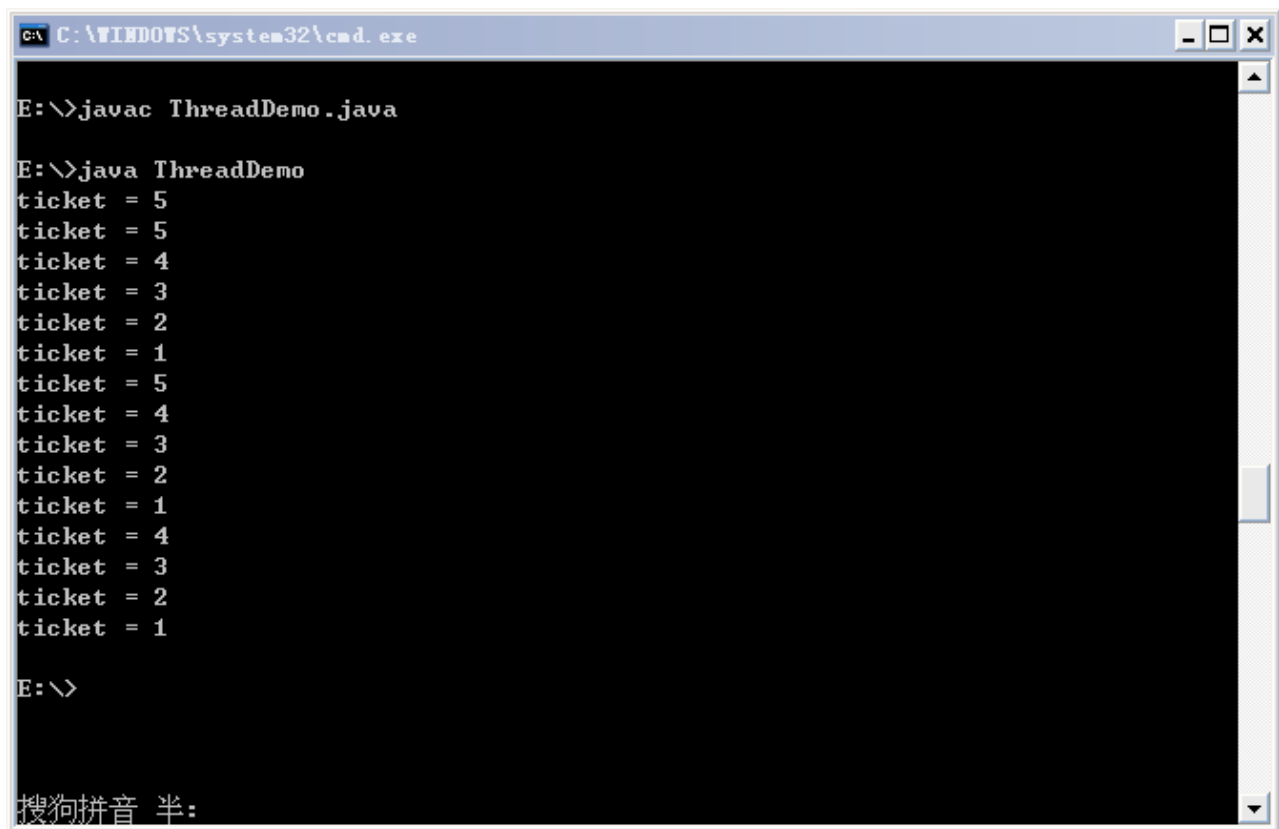
下面以典型的买票程序（基本都是以这个为例子）为例，来说明二者的区别。

首先通过继承 Thread 类实现，代码如下：

```
class MyThread extends Thread{
    private int ticket = 5;
    public void run(){
        for (int i=0;i<10;i++)
        {
            if(ticket > 0){
                System.out.println("ticket = " + ticket--);
            }
        }
    }
}

public class ThreadDemo{
    public static void main(String[] args){
        new MyThread().start();
        new MyThread().start();
        new MyThread().start();
    }
}
```

执行结果如下：



```

C:\WINDOWS\system32\cmd.exe

E:\>javac ThreadDemo.java

E:\>java ThreadDemo
ticket = 5
ticket = 5
ticket = 4
ticket = 3
ticket = 2
ticket = 1
ticket = 5
ticket = 4
ticket = 3
ticket = 2
ticket = 1
ticket = 4
ticket = 3
ticket = 2
ticket = 1

E:\>

```

从结果中可以看出，每个线程单独卖了 5 张票，即独立地完成了买票的任务，但实际应用中，比如火车站售票，需要多个线程去共同完成任务，在本例中，即多个线程共同买 5 张票。

下面是通过实现 Runnable 接口实现的多线程程序，代码如下：

```

class MyThread implements Runnable{
    private int ticket = 5;
    public void run(){
        for (int i=0;i<10;i++){
            {
                if(ticket > 0){
                    System.out.println("ticket = " + ticket--);
                }
            }
        }
    }
}

public class RunnableDemo{
    public static void main(String[] args){
        MyThread my = new MyThread();
        new Thread(my).start();
        new Thread(my).start();
        new Thread(my).start();
    }
}

```

```
}
}
```

执行结果如下：

```
C:\WINDOWS\system32\cmd.exe

E:\>java RunnableDemo
ticket = 5
ticket = 4
ticket = 3
ticket = 1
ticket = 2

E:\>
```

从结果中可以看出，三个线程一共卖了 5 张票，即它们共同完成了买票的任务，实现了资源的共享。

针对以上代码补充三点：

- 在第二种方法（Runnable）中，ticket 输出的顺序并不是 54321，这是因为线程执行的时机难以预测，ticket--并不是原子操作。
- 在第一种方法中，我们 new 了 3 个 Thread 对象，即三个线程分别执行三个对象中的代码，因此便是三个线程去独立地完成卖票的任务；而在第二种方法中，我们同样也 new 了 3 个 Thread 对象，但只有一个 Runnable 对象，3 个 Thread 对象共享这个 Runnable 对象中的代码，因此，便会出现 3 个线程共同完成卖票任务的结果。如果我们 new 出 3 个 Runnable 对象，作为参数分别传入 3 个 Thread 对象中，那么 3 个线程便会独立执行各自 Runnable 对象中的代码，即 3 个线程各自卖 5 张票。
- 在第二种方法中，由于 3 个 Thread 对象共同执行一个 Runnable 对象中的代码，因此可能会造成线程的不安全，比如可能 ticket 会输出 -1（如果我们 System.out....语句前加上线程休眠操作，该情况将很有可能出现），这种情况的出现是由于，一个线程在判断 ticket 为 1>0 后，还没有来得及减 1，另一个线程已经将 ticket 减 1，变为了 0，那么接下来之前的线程再将 ticket 减 1，便得到了 -1。这就需要加入同步操作（即互

斥锁)，确保同一时刻只有一个线程在执行每次 for 循环中的操作。而在第一种方法中，并不需要加入同步操作，因为每个线程执行自己 Thread 对象中的代码，不存在多个线程共同执行同一个方法的情况。



线程中断



使用 interrupt()中断线程

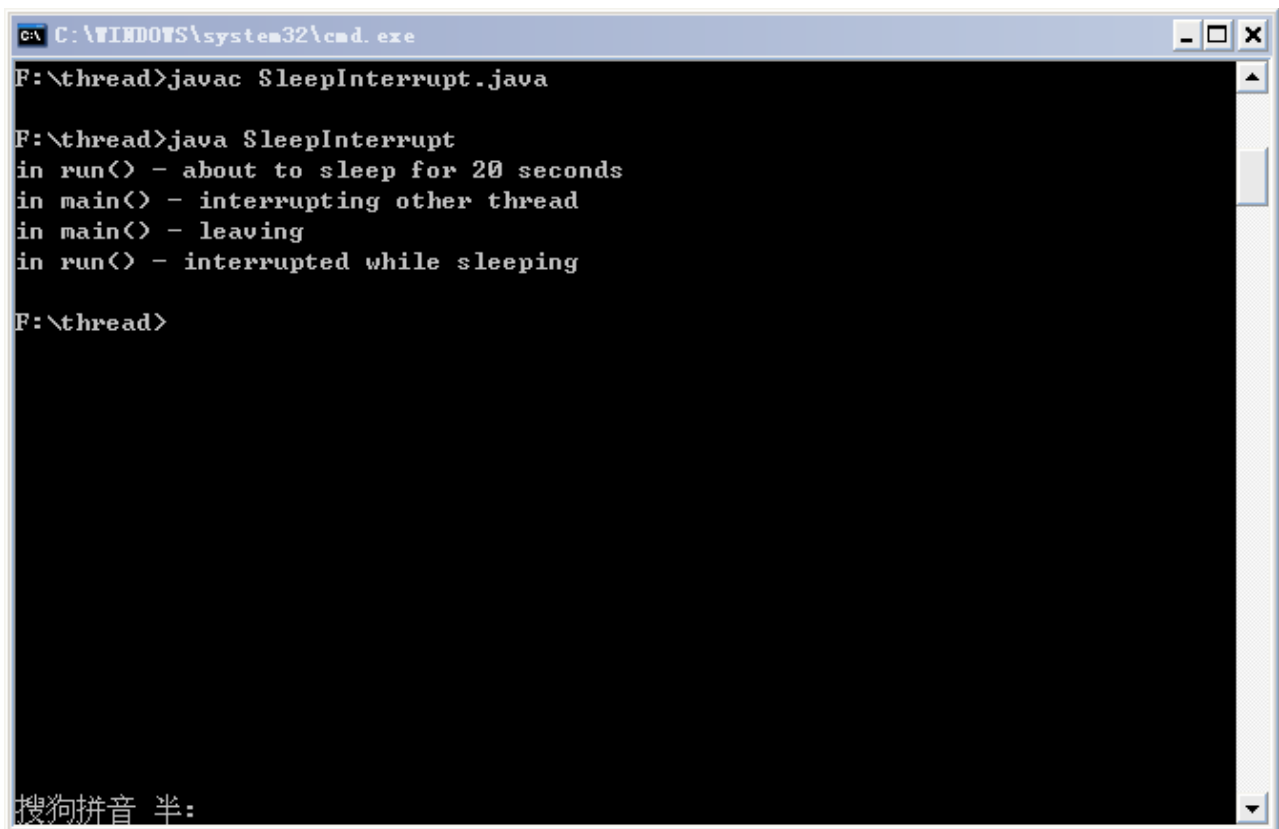
当一个线程运行时，另一个线程可以调用对应的 Thread 对象的 interrupt()方法来中断它，该方法只是在目标线程中设置一个标志，表示它已经被中断，并立即返回。这里需要注意的是，如果只是单纯的调用 interrupt()方法，线程并没有实际被中断，会继续往下执行。

下面一段代码演示了休眠线程的中断：

```
public class SleepInterrupt extends Object implements Runnable{
    public void run(){
        try{
            System.out.println("in run() – about to sleep for 20 seconds");
            Thread.sleep(20000);
            System.out.println("in run() – woke up");
        }catch(InterruptedException e){
            System.out.println("in run() – interrupted while sleeping");
            //处理完中断异常后，返回到run ( ) 方法入口，
            //如果没有return，线程不会实际被中断，它会继续打印下面的信息
            return;
        }
        System.out.println("in run() – leaving normally");
    }

    public static void main(String[] args) {
        SleepInterrupt si = new SleepInterrupt();
        Thread t = new Thread(si);
        t.start();
        //主线程休眠2秒，从而确保刚才启动的线程有机会执行一段时间
        try {
            Thread.sleep(2000);
        }catch(InterruptedException e){
            e.printStackTrace();
        }
        System.out.println("in main() – interrupting other thread");
        //中断线程t
        t.interrupt();
        System.out.println("in main() – leaving");
    }
}
```

运行结果如下：



```

C:\WINDOWS\system32\cmd.exe
F:\thread>javac SleepInterrupt.java

F:\thread>java SleepInterrupt
in run() - about to sleep for 20 seconds
in main() - interrupting other thread
in main() - leaving
in run() - interrupted while sleeping

F:\thread>

```

搜狗拼音 半:

主线程启动新线程后，自身休眠 2 秒钟，允许新线程获得运行时间。新线程打印信息 `about to sleep for 20 seconds` 后，继而休眠 20 秒钟，大约 2 秒钟后，main 线程通知新线程中断，那么新线程的 20 秒的休眠将被打断，从而抛出 `InterruptedException` 异常，执行跳转到 catch 块，打印出 `interrupted while sleeping` 信息，并立即从 `run()` 方法返回，然后消亡，而不会打印出 catch 块后面的 `leaving normally` 信息。

请注意：由于不确定的线程规划，上图运行结果的后两行可能顺序相反，这取决于主线程和新线程哪个先消亡。但前两行信息的顺序必定如上图所示。

另外，如果将 catch 块中的 `return` 语句注释掉，则线程在抛出异常后，会继续往下执行，而不会被中断，从而会打印出 `leaving normally` 信息。

待决中断

在上面的例子中，`sleep()`方法的实现检查到休眠线程被中断，它会相当友好地终止线程，并抛出 `InterruptedException` 异常。另外一种情况，如果线程在调用 `sleep()`方法前被中断，那么该中断称为待决中断，它会在刚调用 `sleep()`方法时，立即抛出 `InterruptedException` 异常。

下面的代码演示了待决中断：

```

public class PendingInterrupt extends Object {
    public static void main(String[] args){
        //如果输入了参数，则在main线程中中断当前线程（亦即main线程）
        if( args.length > 0 ){
            Thread.currentThread().interrupt();
        }
        //获取当前时间
        long startTime = System.currentTimeMillis();
        try{
            Thread.sleep(2000);
            System.out.println("was NOT interrupted");
        }catch(InterruptedException x){
            System.out.println("was interrupted");
        }
        //计算中间代码执行的时间
        System.out.println("elapsedTime=" + ( System.currentTimeMillis() - startTime));
    }
}

```

如果 PendingInterrupt 不带任何命令行参数，那么线程不会被中断，最终输出的时间差距应该在 2000 附近（具体时间由系统决定，不精确），如果 PendingInterrupt 带有命令行参数，则调用中断当前线程的代码，但 main 线程仍然运行，最终输出的时间差距应该远小于 2000，因为线程尚未休眠，便被中断，因此，一旦调用 sleep()方法，会立即打印出 catch 块中的信息。

执行结果如下：

```

C:\ 选定 C:\WINDOWS\system32\cmd.exe
F:\thread>java PendingInterrupt
was NOT interrupted
elapsedTime=2000

F:\thread>java PendingInterrupt yes
was interrupted
elapsedTime=0

F:\thread>

```

搜狗拼音 半:

这种模式下，main 线程中断它自身。除了将中断标志（它是 Thread 的内部标志）设置为 true 外，没有其他任何影响。线程被中断了，但 main 线程仍然运行，main 线程继续监视实时时钟，并进入 try 块，一旦调用 sleep() 方法，它就会注意到待决中断的存在，并抛出 InterruptedException。于是执行跳转到 catch 块，并打印出线程被中断的信息。最后，计算并打印出时间差。

使用 isInterrupted() 方法判断中断状态

可以在 Thread 对象上调用 isInterrupted() 方法来检查任何线程的中断状态。这里需要注意：线程一旦被中断，isInterrupted() 方法便会返回 true，而一旦 sleep() 方法抛出异常，它将清空中断标志，此时 isInterrupted() 方法将返回 false。

下面的代码演示了 isInterrupted() 方法的使用：

```

public class InterruptCheck extends Object{
    public static void main(String[] args){
        Thread t = Thread.currentThread();
        System.out.println("Point A: t.isInterrupted()=" + t.isInterrupted());
        //待决中断，中断自身
        t.interrupt();
        System.out.println("Point B: t.isInterrupted()=" + t.isInterrupted());
        System.out.println("Point C: t.isInterrupted()=" + t.isInterrupted());
    }
}

```



```

try{
    Thread.sleep(2000);
    System.out.println("was NOT interrupted");
}catch( InterruptedException x){
    System.out.println("was interrupted");
}
//抛出异常后，会清除中断标志，这里会返回false
System.out.println("Point D: t.isInterrupted()=" + t.isInterrupted());
}
}

```

运行结果如下：

```

C:\WINDOWS\system32\cmd.exe
F:\thread>javac InterruptCheck.java
F:\thread>java InterruptCheck
Point A: t.isInterrupted()=false
Point B: t.isInterrupted()=true
Point C: t.isInterrupted()=true
was interrupted
Point D: t.isInterrupted()=false
F:\thread>

```

搜狗拼音 半:

使用 Thread.interrupted()方法判断中断状态

可以使用 `Thread.interrupted()` 方法来检查当前线程的中断状态（并隐式重置为 `false`）。又由于它是静态方法，因此不能在特定的线程上使用，而只能报告调用它的线程的中断状态，如果线程被中断，而且中断状态尚不清楚，那么，这个方法返回 `true`。与 `isInterrupted()` 不同，它将自动重置中断状态为 `false`，第二次调用 `Thread.interrupted()` 方法，总是返回 `false`，除非中断了线程。

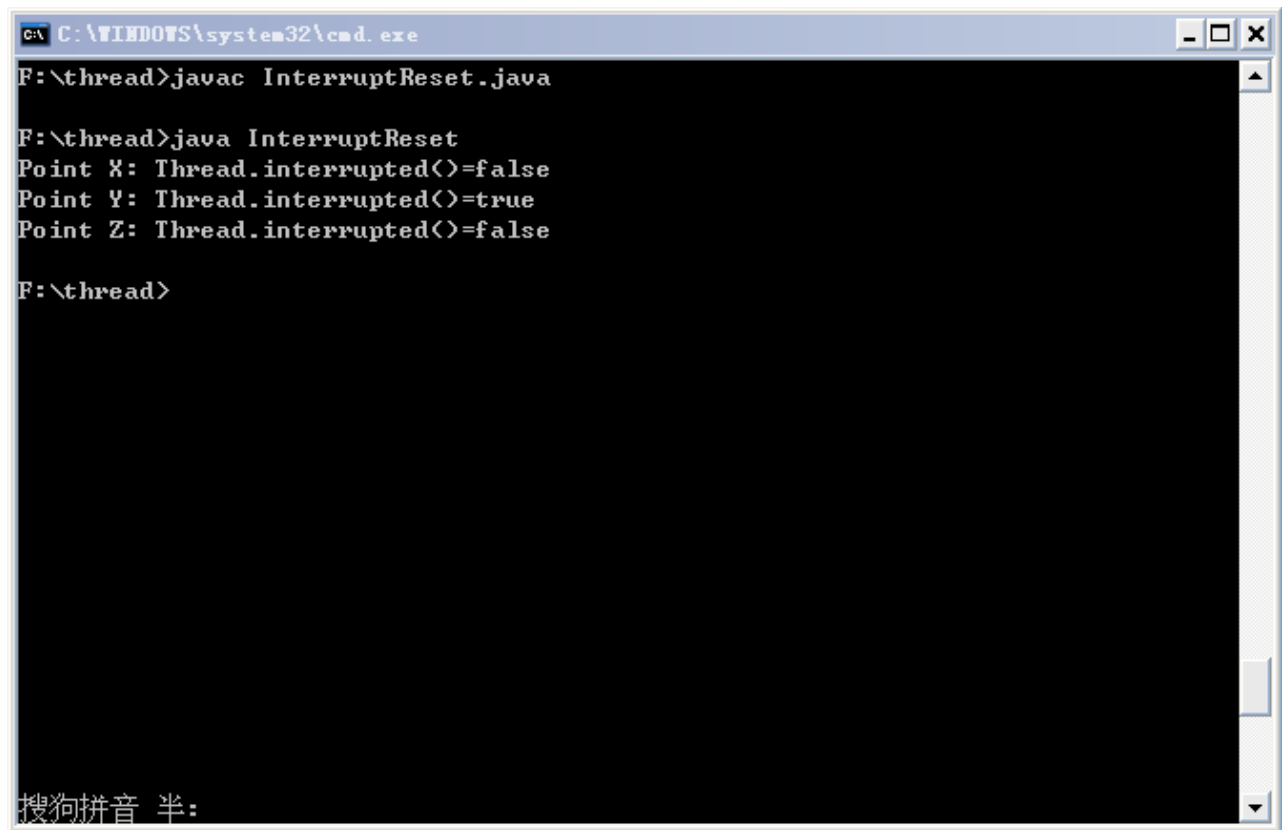
如下代码演示了 `Thread.interrupted()` 方法的使用：

```

public class InterruptReset extends Object {
    public static void main(String[] args) {
        System.out.println(
            "Point X: Thread.interrupted()=" + Thread.interrupted());
        Thread.currentThread().interrupt();
        System.out.println(
            "Point Y: Thread.interrupted()=" + Thread.interrupted());
        System.out.println(
            "Point Z: Thread.interrupted()=" + Thread.interrupted());
    }
}

```

运行结果如下：



```

C:\WINDOWS\system32\cmd.exe
F:\thread>javac InterruptReset.java

F:\thread>java InterruptReset
Point X: Thread.interrupted()=false
Point Y: Thread.interrupted()=true
Point Z: Thread.interrupted()=false

F:\thread>

```

从结果中可以看出，当前线程中断自身后，在 Y 点，中断状态为 true，并由 Thread.interrupted() 自动重置为 false，那么下次调用该方法得到的结果便是 false。

补充

这里补充下 yield 和 join 方法的使用。

- join 方法用线程对象调用，如果在一个线程 A 中调用另一个线程 B 的 join 方法，线程 A 将会等待线程 B 执行完毕后再执行。
- yield 可以直接用 Thread 类调用，yield 让出 CPU 执行权给同等级的线程，如果没有相同级别的线程在等待 CPU 的执行权，则该线程继续执行。



4

线程挂起、恢复与终止



挂起和恢复线程

Thread 的 API 中包含两个被淘汰的方法，它们用于临时挂起和重启某个线程，这些方法已经被淘汰，因为它们是不安全的，不稳定的。如果在不合适的时候挂起线程（比如，锁定共享资源时），此时便可能会发生死锁条件——其他线程在等待该线程释放锁，但该线程却被挂起了，便会发生死锁。另外，在长时间计算期间挂起线程也可能导致问题。

下面的代码演示了通过休眠来延缓运行，模拟长时间运行的情况，使线程更可能在不适当的时候被挂起：

```
public class DeprecatedSuspendResume extends Object implements Runnable{

    //volatile关键字，表示该变量可能在被一个线程使用的同时，被另一个线程修改
    private volatile int firstVal;
    private volatile int secondVal;

    //判断二者是否相等
    public boolean areValuesEqual(){
        return ( firstVal == secondVal);
    }

    public void run() {
        try{
            firstVal = 0;
            secondVal = 0;
            workMethod();
        }catch(InterruptedException x){
            System.out.println("interrupted while in workMethod()");
        }
    }

    private void workMethod() throws InterruptedException {
        int val = 1;
        while (true){
            stepOne(val);
            stepTwo(val);
            val++;
            Thread.sleep(200); //再次循环钱休眠200毫秒
        }
    }

    //赋值后，休眠300毫秒，从而使线程有机会在stepOne操作和stepTwo操作之间被挂起
    private void stepOne(int newVal) throws InterruptedException{
        firstVal = newVal;
```

```

        Thread.sleep(300); //模拟长时间运行的情况
    }

    private void stepTwo(int newVal){
        secondVal = newVal;
    }

    public static void main(String[] args){
        DeprecatedSuspendResume dsr = new DeprecatedSuspendResume();
        Thread t = new Thread(dsr);
        t.start();

        //休眠1秒，让其他线程有机会获得执行
        try {
            Thread.sleep(1000);}
        catch (InterruptedException x){}
        for (int i = 0; i < 10; i++){
            //挂起线程
            t.suspend();
            System.out.println("dsr.areValuesEqual()=" + dsr.areValuesEqual());
            //恢复线程
            t.resume();
            try{
                //线程随机休眠0~2秒
                Thread.sleep((long)(Math.random()*2000.0));
            }catch (InterruptedException x){
                //略
            }
        }
        System.exit(0); //中断应用程序
    }
}

```

运行结果如下：

```

C:\WINDOWS\system32\cmd.exe

F:\thread>javac DeprecatedSuspendResume.java
注意: DeprecatedSuspendResume.java 使用或覆盖了已过时的 API。
注意: 要了解详细信息, 请使用 -Xlint:deprecation 重新编译。

F:\thread>java DeprecatedSuspendResume
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=false
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=false
dsr.areValuesEqual()=false
dsr.areValuesEqual()=true
dsr.areValuesEqual()=false
dsr.areValuesEqual()=true

F:\thread>

```

搜狗拼音 半:

从 `areValuesEqual()` 返回的值有时为 `true`，有时为 `false`。以上代码中，在设置 `firstVal` 之后，但在设置 `secondVal` 之前，挂起新线程会产生麻烦，此时输出的结果会为 `false`（情况 1），这段时间不宜挂起线程，但因为线程不能控制何时调用它的 `suspend` 方法，所以这种情况是不可避免的。

当然，即使线程不被挂起（注释掉挂起和恢复线程的两行代码），如果在 `main` 线程中执行 `asr.areValuesEqual()` 进行比较时，恰逢 `stepOne` 操作执行完，而 `stepTwo` 操作还没执行，那么得到的结果同样可能是 `false`（情况 2）。

下面我们给出不用上述两个方法来实现线程挂起和恢复的策略——设置标志位。通过该方法实现线程的挂起和恢复有一个很好的地方，就是可以在线程的指定位置实现线程的挂起和恢复，而不用担心其不确定性。

对于上述代码的改进代码如下：

```

public class AlternateSuspendResume extends Object implements Runnable {

    private volatile int firstVal;
    private volatile int secondVal;
    //增加标志位，用来实现线程的挂起和恢复
    private volatile boolean suspended;

    public boolean areValuesEqual() {
        return ( firstVal == secondVal );
    }
}

```

```

}

public void run() {
    try {
        suspended = false;
        firstVal = 0;
        secondVal = 0;
        workMethod();
    } catch ( InterruptedException x ) {
        System.out.println("interrupted while in workMethod()");
    }
}

private void workMethod() throws InterruptedException {
    int val = 1;

    while ( true ) {
        //仅当贤臣挂起时，才运行这行代码
        waitWhileSuspended();

        stepOne(val);
        stepTwo(val);
        val++;

        //仅当线程挂起时，才运行这行代码
        waitWhileSuspended();

        Thread.sleep(200);
    }
}

private void stepOne(int newVal)
    throws InterruptedException {

    firstVal = newVal;
    Thread.sleep(300);
}

private void stepTwo(int newVal) {
    secondVal = newVal;
}

public void suspendRequest() {
    suspended = true;
}

```



```

public void resumeRequest() {
    suspended = false;
}

private void waitWhileSuspended()
    throws InterruptedException {

    //这是一个“繁忙等待”技术的示例。
    //它是非等待条件改变的最佳途径，因为它会不断请求处理器周期地执行检查，
    //更佳的技术是：使用Java的内置“通知-等待”机制
    while ( suspended ) {
        Thread.sleep(200);
    }
}

public static void main(String[] args) {
    AlternateSuspendResume asr =
        new AlternateSuspendResume();

    Thread t = new Thread(asr);
    t.start();

    //休眠1秒，让其他线程有机会获得执行
    try { Thread.sleep(1000); }
    catch ( InterruptedException x ) {}

    for ( int i = 0; i < 10; i++ ) {
        asr.suspendRequest();

        //让线程有机会注意到挂起请求
        //注意：这里休眠时间一定要大于
        //stepOne操作对firstVal赋值后的休眠时间，即300ms，
        //目的是为了防止在执行asr.areValuesEqual ( ) 进行比较时，
        //恰逢stepOne操作执行完，而stepTwo操作还没执行
        try { Thread.sleep(350); }
        catch ( InterruptedException x ) {}

        System.out.println("dsr.areValuesEqual()=" +
            asr.areValuesEqual());

        asr.resumeRequest();

        try {
            //线程随机休眠0~2秒

```

```

        Thread.sleep(
            ( long ) (Math.random() * 2000.0) );
    } catch ( InterruptedException x ) {
        //略
    }
}

System.exit(0); //退出应用程序
}
}

```

运行结果如下：

```

c:\ 选定 C:\WINDOWS\system32\cmd.exe
F:\thread>javac AlternateSuspendResume.java

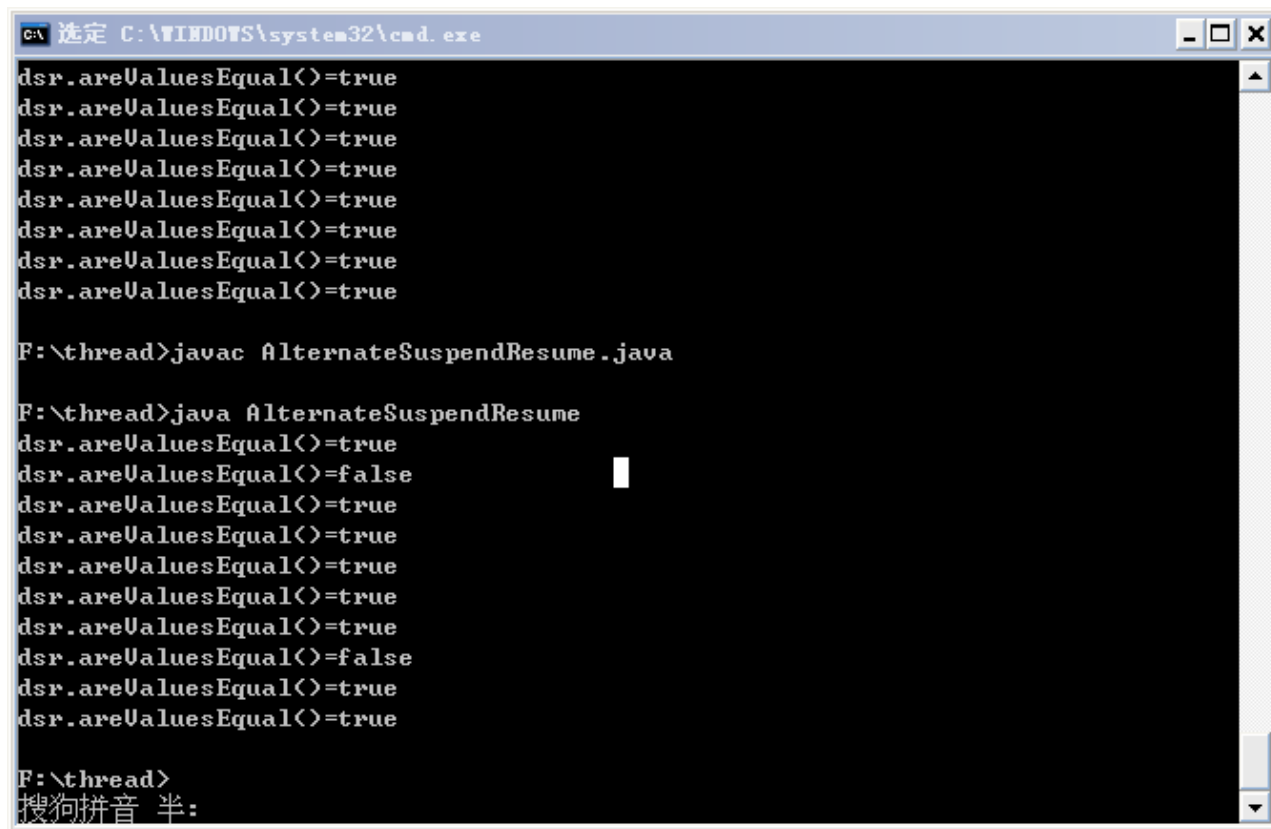
F:\thread>java AlternateSuspendResume
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true

F:\thread>java AlternateSuspendResume
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
搜狗拼音 半:

```

由结果可以看出，输出的所有结果均为 true。首先，针对情况 1（线程挂起的位置不确定），这里确定了线程挂起的位置，不会出现线程在 stepOne 操作和 stepTwo 操作之间挂起的情况；针对情况 2（main 线程中执行 asr.areValuesEqual() 进行比较时，恰逢 stepOne 操作执行完，而 stepTwo 操作还没执行），在发出挂起请求后，还没有执行 asr.areValuesEqual() 操作前，让 main 线程休眠 450ms（>300ms），如果挂起请求发出时，新线程正执行到或即将执行到 stepOne 操作（如果在其前面的话，就会响应挂起请求，从而挂起线程），那么在 stepTwo 操作执行前，main 线程的休眠还没结束，从而 main 线程休眠结束后执行 asr.areValuesEqual() 操作进行比较时，stepTwo 操作已经执行完，因此也不会出现输出结果为 false 的情况。

可以将 `ars.suspendRequest()` 代码后的 `sleep` 代码去掉，或将休眠时间改为 200（明显小于 300 即可）后，查看执行结果，会发现结果中依然会有出现 `false` 的情况。如下图所示：



```

C:\WINDOWS\system32\cmd.exe
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true

F:\thread>javac AlternateSuspendResume.java

F:\thread>java AlternateSuspendResume
dsr.areValuesEqual()=true
dsr.areValuesEqual()=false
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true
dsr.areValuesEqual()=false
dsr.areValuesEqual()=true
dsr.areValuesEqual()=true

F:\thread>
搜狗拼音 半:

```

总结：线程的挂起和恢复实现的正确方法是：通过设置标志位，让线程在安全的位置挂起。

终止线程

当调用 `Thread` 的 `start()` 方法，执行完 `run()` 方法后，或在 `run()` 方法中 `return`，线程便会自然消亡。另外 `Thread` API 中包含了一个 `stop()` 方法，可以突然终止线程。但它在 JDK1.2 后便被淘汰了，因为它可能导致数据对象的崩溃。一个问题是，当线程终止时，很少有机会执行清理工作；另一个问题是，当在某个线程上调用 `stop()` 方法时，线程释放它当前持有的所有锁，持有这些锁必定有某种合适的理由——也许是阻止其他线程访问尚未处于一致性状态的数据，突然释放锁可能使某些对象中的数据处于不一致状态，而且不会出现数据可能崩溃的任何警告。

终止线程的替代方法：同样是使用标志位，通过控制标志位来终止线程。



T



5

守护线程与线程阻塞



守护线程

Java 中有两类线程：User Thread(用户线程)、Daemon Thread(守护线程)

用户线程即运行在前台的线程，而守护线程是运行在后台的线程。守护线程作用是为其他前台线程的运行提供便利服务，而且仅在普通、非守护线程仍然运行时才需要，比如垃圾回收线程就是一个守护线程。当 VM 检测仅剩一个守护线程，而用户线程都已经退出运行时，VM 就会退出，因为没有如果没有了被守护这，也就没有继续运行程序的必要了。如果有非守护线程仍然存活，VM 就不会退出。

守护线程并非只有虚拟机内部提供，用户在编写程序时也可以自己设置守护线程。用户可以用 Thread 的 `setDaemon(true)` 方法设置当前线程为守护线程。

虽然守护线程可能非常有用，但必须小心确保其他所有非守护线程消亡时，不会由于它的终止而产生任何危害。因为你不可能知道在所有的用户线程退出运行前，守护线程是否已经完成了预期的服务任务。一旦所有的用户线程退出了，虚拟机也就退出运行了。因此，不要在守护线程中执行业务逻辑操作（比如对数据的读写等）。

另外有几点需要注意：

- `setDaemon(true)` 必须在调用线程的 `start()` 方法之前设置，否则会跑出 `IllegalThreadStateException` 异常。
- 在守护线程中产生的新线程也是守护线程。
- 不要认为所有的应用都可以分配给守护线程来进行服务，比如读写操作或者计算逻辑。

线程阻塞

线程可以阻塞于四种状态：

- 当线程执行 `Thread.sleep()` 时，它一直阻塞到指定的毫秒时间之后，或者阻塞被另一个线程打断；
- 当线程碰到一条 `wait()` 语句时，它会一直阻塞到接到通知（`notify()`）、被中断或经过了指定毫秒时间为止（若制定了超时值的话）
- 线程阻塞与不同 I/O 的方式有多种。常见的一种方式 `InputStream` 的 `read()` 方法，该方法一直阻塞到从流中读取一个字节的数据为止，它可以无限阻塞，因此不能指定超时时间；
- 线程也可以阻塞等待获取某个对象锁的排他性访问权限（即等待获得 `synchronized` 语句必须的锁时阻塞）。

注意，并非所有的阻塞状态都是可中断的，以上阻塞状态的前两种可以被中断，后两种不会对中断做出反应



T



6

volatile 关键字（上）



volatile 用处说明

在 JDK1.2 之前，Java 的内存模型实现总是从主存（即共享内存）读取变量，是不需要进行特别的注意的。而随着 JVM 的成熟和优化，现在在多线程环境下 volatile 关键字的使用变得非常重要。

在当前的 Java 内存模型下，线程可以把变量保存在本地内存（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成数据的不一致。

要解决这个问题，就需要把变量声明为 volatile，这就指示 JVM，这个变量是不稳定的，每次使用它都到主存中进行读取。一般说来，多任务环境下，各任务间共享的变量都应该加 volatile 修饰符。

Volatile 修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

Java 语言规范中指出：为了获得最佳速度，允许线程保存共享成员变量的私有拷贝，而且只当线程进入或者离开同步代码块时才将私有拷贝与共享内存中的原始值进行比较。

这样当多个线程同时与某个对象交互时，就必须注意到要让线程及时的得到共享成员变量的变化。而 volatile 关键字就是提示 JVM：对于这个成员变量，不能保存它的私有拷贝，而应直接与共享成员变量交互。

volatile 是一种稍弱的同步机制，在访问 volatile 变量时不会执行加锁操作，也就不会执行线程阻塞，因此 volatile 变量是一种比 synchronized 关键字更轻量级的同步机制。

使用建议：在两个或者更多的线程需要访问的成员变量上使用 volatile。当要访问的变量已在 synchronized 代码块中，或者为常量时，没必要使用 volatile。

由于使用 volatile 屏蔽掉了 JVM 中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。

示例程序

下面给出一段代码，通过其运行结果来说明使用关键字 volatile 产生的差异，但实际上遇到了意料之外的问题：

```
public class Volatile extends Object implements Runnable {  
    //value变量没有被标记为volatile  
    private int value;  
    //missedIt变量被标记为volatile  
    private volatile boolean missedIt;  
    //creationTime不需要声明为volatile，因为代码执行中它没有发生变化
```

```

private long creationTime;

public Volatile() {
    value = 10;
    missedIt = false;
    //获取当前时间，亦即调用Volatile构造函数时的时间
    creationTime = System.currentTimeMillis();
}

public void run() {
    print("entering run()");

    //循环检查value的值是否不同
    while ( value < 20 ) {
        //如果missedIt的值被修改为true，则通过break退出循环
        if ( missedIt ) {
            //进入同步代码块前，将value的值赋给currValue
            int currValue = value;
            //在一个任意对象上执行同步语句，目的是为了让该线程在进入和离开同步代码块时，
            //将该线程中的所有变量的私有拷贝与共享内存中的原始值进行比较，
            //从而发现没有用volatile标记的变量所发生的变化
            Object lock = new Object();
            synchronized ( lock ) {
                //不做任何事
            }
            //离开同步代码块后，将此时value的值赋给valueAfterSync
            int valueAfterSync = value;
            print("in run() - see value=" + currValue + ", but rumor has it that it changed!");
            print("in run() - valueAfterSync=" + valueAfterSync);
            break;
        }
    }
    print("leaving run()");
}

public void workMethod() throws InterruptedException {
    print("entering workMethod()");
    print("in workMethod() - about to sleep for 2 seconds");
    Thread.sleep(2000);
    //仅在此改变value的值
    value = 50;
    print("in workMethod() - just set value=" + value);
    print("in workMethod() - about to sleep for 5 seconds");
    Thread.sleep(5000);
    //仅在此改变missedIt的值

```



```

        missedIt = true;
        print("in workMethod() – just set missedIt=" + missedIt);
        print("in workMethod() – about to sleep for 3 seconds");
        Thread.sleep(3000);
        print("leaving workMethod()");
    }

/*
*该方法的功能是在要打印的msg信息前打印出程序执行到此所化去的时间，以及打印msg的代码所在的线程
*/
    private void print(String msg) {
        //使用java.text包的功能，可以简化这个方法，但是这里没有利用这一点
        long interval = System.currentTimeMillis() – creationTime;
        String tmpStr = "    " + ( interval / 1000.0 ) + "000";
        int pos = tmpStr.indexOf(".");
        String secStr = tmpStr.substring(pos – 2, pos + 4);
        String nameStr = "    " + Thread.currentThread().getName();
        nameStr = nameStr.substring(nameStr.length() – 8, nameStr.length());
        System.out.println(secStr + " " + nameStr + ": " + msg);
    }

    public static void main(String[] args) {
        try {
            //通过该构造函数可以获取实时时钟的当前时间
            Volatile vol = new Volatile();

            //稍停100ms，以让实时时钟稍稍超前获取时间，使print（）中创建的消息打印的时间值大于0
            Thread.sleep(100);

            Thread t = new Thread(vol);
            t.start();

            //休眠100ms，让刚刚启动的线程有时间运行
            Thread.sleep(100);
            //workMethod方法在main线程中运行
            vol.workMethod();
        } catch ( InterruptedException x ) {
            System.err.println("one of the sleeps was interrupted");
        }
    }
}

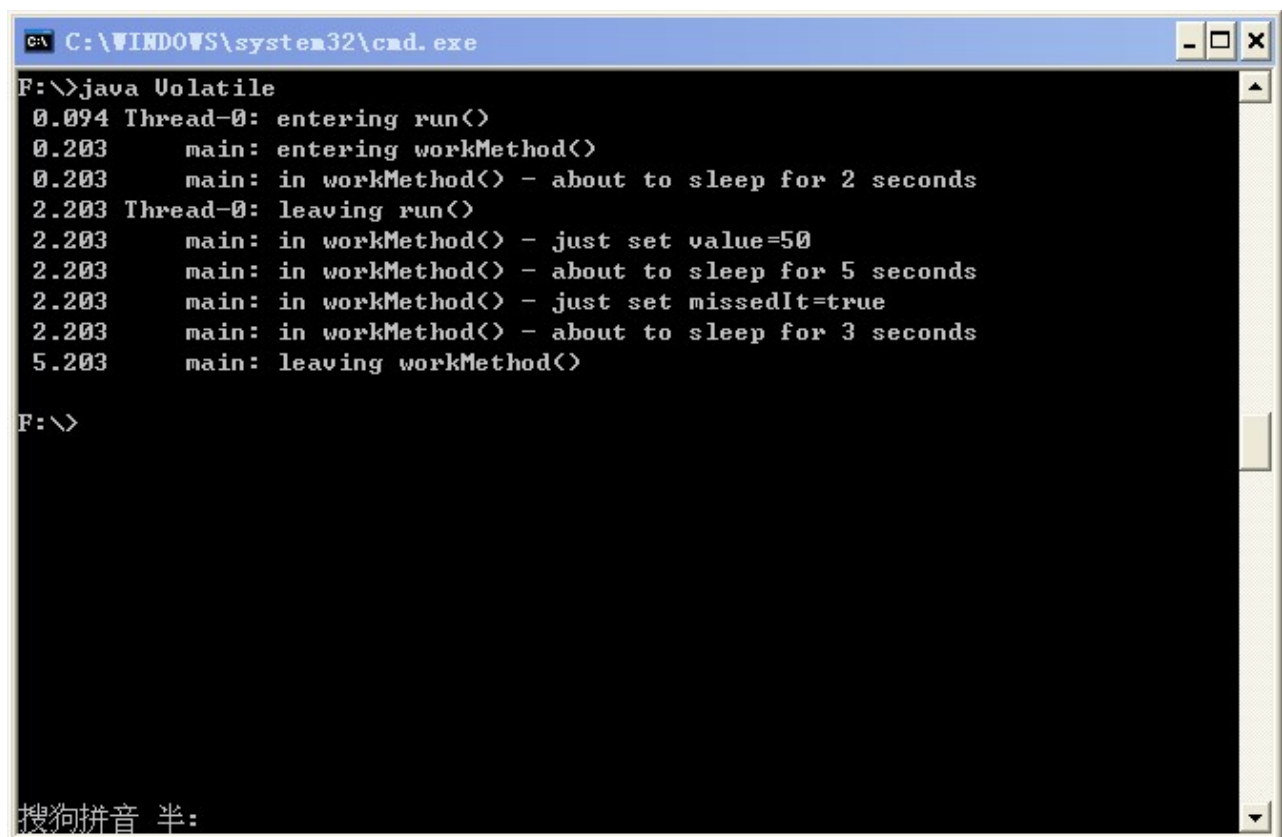
```

按照以上的理论来分析，由于 value 变量不是 volatile 的，因此它在 main 线程中的改变不会被 Thread-0 线程（在 main 线程中新开启的线程）马上看到，因此 Thread-0 线程中的 while 循环不会直接退出，它会继续判断 missedIt 的值，由于 missedIt 是 volatile 的，当 main 线程中改变了 missedIt 时，Thread-0 线程会立即

看到该变化, 那么 if 语句中的代码便得到了执行的机会, 由于此时 Thread-0 依然没有看到 value 值的变化, 因此, currValue 的值为 10, 继续向下执行, 进入同步代码块, 因为进入前后要将该线程内的变量值与共享内存中的原始值对比, 进行校准, 因此离开同步代码块后, Thread-0 便会察觉到 value 的值变为了 50, 那么后面的 valueAfterSync 的值便为 50, 最后从 break 跳出循环, 结束 Thread-0 线程。

意料之外的问题

但实际的执行结果如下:



```

C:\WINDOWS\system32\cmd.exe
F:\>java Volatile
0.094 Thread-0: entering run()
0.203      main: entering workMethod()
0.203      main: in workMethod() - about to sleep for 2 seconds
2.203 Thread-0: leaving run()
2.203      main: in workMethod() - just set value=50
2.203      main: in workMethod() - about to sleep for 5 seconds
2.203      main: in workMethod() - just set missedIt=true
2.203      main: in workMethod() - about to sleep for 3 seconds
5.203      main: leaving workMethod()

F:\>

```

从结果中可以看出, Thread-0 线程并没有进入 while 循环, 说明 Thread-0 线程在 value 的值发生变化后, missedIt 的值发生变化前, 便察觉到了 value 值的变化, 从而退出了 while 循环。这与理论上的分析不符, 我便尝试注释掉 value 值发生改变与 missedIt 值发生改变之间的线程休眠代码 Thread.sleep(5000), 以确保 Thread-0 线程在 missedIt 的值发生改变前, 没有时间察觉到 value 值的变化。但执行的结果与上面大同小异 (可能有一两行顺序不同, 但依然不会打印出 if 语句中的输出信息)。

问题分析

在 JDK1.7~JDK1.3 之间的版本上输出结果与上面基本大同小异，只有在 JDK1.2 上才得到了预期的结果，即 Thread-0 线程中的 while 循环是从 if 语句中退出的，这说明 Thread-0 线程没有及时察觉到 value 值的变化。

这里需要注意：volatile 是针对 JIT 带来的优化，因此 JDK1.2 以前的版本基本不用考虑，另外，在 JDK1.3.1 开始，开始运用 HotSpot 虚拟机，用来代替 JIT。因此，是不是 HotSpot 的问题呢？这里需要再补充一点：

JIT 或 HotSpot 编译器在 server 模式和 client 模式编译不同，server 模式为了使线程运行更快，如果其中一个线程更改了变量 boolean flag 的值，那么另外一个线程会看不到，因为另外一个线程为了使得运行更快所以从寄存器或者本地 cache 中取值，而不是从内存中取值，那么使用 volatile 后，就告诉不论是什么线程，被 volatile 修饰的变量都要从内存中取值。

对于非 volatile 修饰的变量，尽管 jvm 的优化，会导致变量的可见性问题，但这种可见性的问题也只是在短时间内高并发的情况下发生，CPU 执行时会很快刷新 Cache，一般的情况下很难出现，而且出现这种问题是不可预测的，与 jvm, 机器配置环境等都有关系。



7

volatile 关键字（下）



在《Volatile 关键字（上）》一文中遗留了一个问题，就是 volatile 只修饰了 missedIt 变量，而没修饰 value 变量，但是在线程读取 value 的值的时候，也读到的是最新的数据。

下面讲解问题出现的原因。

首先明确一点：假如有两个线程分别读写 volatile 变量时，线程 A 写入了某 volatile 变量，线程 B 在读取该 volatile 变量时，便能看到线程 A 对该 volatile 变量的写入操作，关键在这里，它不仅会看到对该 volatile 变量的写入操作，A 线程在写 volatile 变量之前所有可见的共享变量，在 B 线程读同一个 volatile 变量后，都将立即变得对 B 线程可见。

回过头来看文章中出现的问题，由于程序中 volatile 变量 missedIt 的写入操作在 value 变量写入操作之后，而且根据 volatile 规则，又不能重排序，因此，在线程 B 读取由线程 A 改变后的 missedIt 之后，它之前的 value 变量在线程 A 的改变也对线程 B 变得可见了。

我们颠倒一下 value=50 和 missedIt=true 这两行代码试下，即 missedIt=true 在前，value=50 在后，这样便会得到我们想要的结果：value 值的改变不会被看到。

这应该是 JDK1.2 之后对 volatile 规则做了一些修订的结果。

修改后的代码如下：

```
public class Volatile extends Object implements Runnable {
    //value变量没有被标记为volatile
    private int value;
    //missedIt变量被标记为volatile
    private volatile boolean missedIt;
    //creationTime不需要声明为volatile，因为代码执行中它没有发生变化
    private long creationTime;

    public Volatile() {
        value = 10;
        missedIt = false;
        //获取当前时间，亦即调用Volatile构造函数时的时间
        creationTime = System.currentTimeMillis();
    }

    public void run() {
        print("entering run()");

        //循环检查value的值是否不同
        while ( value < 20 ) {
            //如果missedIt的值被修改为true，则通过break退出循环
            if ( missedIt ) {
                //进入同步代码块前，将value的值赋给currValue
            }
        }
    }
}
```

```

    int currValue = value;
    //在一个任意对象上执行同步语句，目的是为了让该线程在进入和离开同步代码块时，
    //将该线程中的所有变量的私有拷贝与共享内存中的原始值进行比较，
    //从而发现没有用volatile标记的变量所发生的变化
    Object lock = new Object();
    synchronized ( lock ) {
        //不做任何事
    }
    //离开同步代码块后，将此时value的值赋给valueAfterSync
    int valueAfterSync = value;
    print("in run() - see value=" + currValue + ", but rumor has it that it changed!");
    print("in run() - valueAfterSync=" + valueAfterSync);
    break;
}
}
print("leaving run()");
}

public void workMethod() throws InterruptedException {
    print("entering workMethod()");
    print("in workMethod() - about to sleep for 2 seconds");
    Thread.sleep(2000);
    //仅在此改变value的值
    missedIt = true;
//    value = 50;
    print("in workMethod() - just set value=" + value);
    print("in workMethod() - about to sleep for 5 seconds");
    Thread.sleep(5000);
    //仅在此改变missedIt的值
//    missedIt = true;
    value = 50;
    print("in workMethod() - just set missedIt=" + missedIt);
    print("in workMethod() - about to sleep for 3 seconds");
    Thread.sleep(3000);
    print("leaving workMethod()");
}

/*
*该方法的功能是在要打印的msg信息前打印出程序执行到此所化去的时间，以及打印msg的代码所在的线程
*/
private void print(String msg) {
    //使用java.text包的功能，可以简化这个方法，但是这里没有利用这一点
    long interval = System.currentTimeMillis() - creationTime;
    String tmpStr = "  " + ( interval / 1000.0 ) + "000";
    int pos = tmpStr.indexOf(".");

```

```

String secStr = tmpStr.substring(pos - 2, pos + 4);
String nameStr = "    " + Thread.currentThread().getName();
nameStr = nameStr.substring(nameStr.length() - 8, nameStr.length());
System.out.println(secStr + " " + nameStr + ": " + msg);
}

public static void main(String[] args) {
    try {
        //通过该构造函数可以获取实时时钟的当前时间
        Volatile vol = new Volatile();

        //稍停100ms，以让实时时钟稍稍超前获取时间，使print（）中创建的消息打印的时间值大于0
        Thread.sleep(100);

        Thread t = new Thread(vol);
        t.start();

        //休眠100ms，让刚刚启动的线程有时间运行
        Thread.sleep(100);
        //workMethod方法在main线程中运行
        vol.workMethod();
    } catch ( InterruptedException x ) {
        System.err.println("one of the sleeps was interrupted");
    }
}
}

```

运行结果如下：

```

C:\ 选定 C:\WINDOWS\system32\cmd.exe - java Volatile
F:\>
F:\>javac Volatile.java
F:\>java Volatile
0.093 Thread-0: entering run()
0.203      main: entering workMethod()
0.203      main: in workMethod() - about to sleep for 2 seconds
2.203      main: in workMethod() - just set value=10
2.203      main: in workMethod() - about to sleep for 5 seconds
2.203 Thread-0: in run() - see value=10, but rumor has it that it changed!
2.218 Thread-0: in run() - valueAfterSync=10
2.218 Thread-0: leaving run()
7.203      main: in workMethod() - just set missedIt=true
18.109      main: in workMethod() - about to sleep for 3 seconds

```

搜狗拼音 半:

很明显，这其实并不符合使用 volatile 的第二个条件：该变量要没有包含在具有其他变量的不变式中。因此，在这里使用 volatile 是不安全的。

附上一篇讲述 volatile 关键字正确使用的很好的文章：<http://www.ibm.com/developerworks/cn/java/j-jtp06197.html>



synchronized 关键字



在并发编程中，多线程同时并发访问的资源叫做临界资源，当多个线程同时访问对象并要求操作相同资源时，分割了原子操作就有可能出现数据的不一致或数据不完整的情况，为避免这种情况的发生，我们会采取同步机制，以确保在某一时刻，方法内只允许有一个线程。

采用 synchronized 修饰符实现的同步机制叫做互斥锁机制，它所获得的锁叫做互斥锁。每个对象都有一个 monitor (锁标记)，当线程拥有这个锁标记时才能访问这个资源，没有锁标记便进入锁池。任何一个对象系统都会为其创建一个互斥锁，这个锁是为了分配给线程的，防止打断原子操作。每个对象的锁只能分配给一个线程，因此叫做互斥锁。

这里就使用同步机制获取互斥锁的情况，进行几点说明：

1. 如果同一个方法内同时有两个或更多线程，则每个线程有自己的局部变量拷贝。
2. 类的每个实例都有自己的对象级别锁。当一个线程访问实例对象中的 synchronized 同步代码块或同步方法时，该线程便获取了该实例的对象级别锁，其他线程这时如果要访问 synchronized 同步代码块或同步方法，便需要阻塞等待，直到前面的线程从同步代码块或方法中退出，释放掉了该对象级别锁。
3. 访问同一个类的不同实例对象中的同步代码块，不存在阻塞等待获取对象锁的问题，因为它们获取的是各自实例的对象级别锁，相互之间没有影响。
4. 持有一个对象级别锁不会阻止该线程被交换出来，也不会阻塞其他线程访问同一示例对象中的非 synchronized 代码。当一个线程 A 持有一个对象级别锁（即进入了 synchronized 修饰的代码块或方法中）时，线程也有可能被交换出去，此时线程 B 有可能获取执行该对象中代码的时间，但它只能执行非同步代码（没有用 synchronized 修饰），当执行到同步代码时，便会被阻塞，此时可能线程规划器又让 A 线程运行，A 线程继续持有对象级别锁，当 A 线程退出同步代码时（即释放了对象级别锁），如果 B 线程此时再运行，便会获得该对象级别锁，从而执行 synchronized 中的代码。
5. 持有对象级别锁的线程会让其他线程阻塞在所有的 synchronized 代码外。例如，在一个类中有三个 synchronized 方法 a, b, c，当线程 A 正在执行一个实例对象 M 中的方法 a 时，它便获得了该对象级别锁，那么其他的线程在执行同一实例对象（即对象 M）中的代码时，便会在所有的 synchronized 方法处阻塞，即在方法 a, b, c 处都要被阻塞，等线程 A 释放掉对象级别锁时，其他的线程才可以去执行方法 a, b 或者 c 中的代码，从而获得该对象级别锁。
6. 使用 synchronized (obj) 同步语句块，可以获取指定对象上的对象级别锁。obj 为对象的引用，如果获取了 obj 对象上的对象级别锁，在并发访问 obj 对象时时，便会在其 synchronized 代码处阻塞等待，直到获取到该 obj 对象的对象级别锁。当 obj 为 this 时，便是获取当前对象的对象级别锁。
7. 类级别锁被特定类的所有示例共享，它用于控制对 static 成员变量以及 static 方法的并发访问。具体用法与对象级别锁相似。

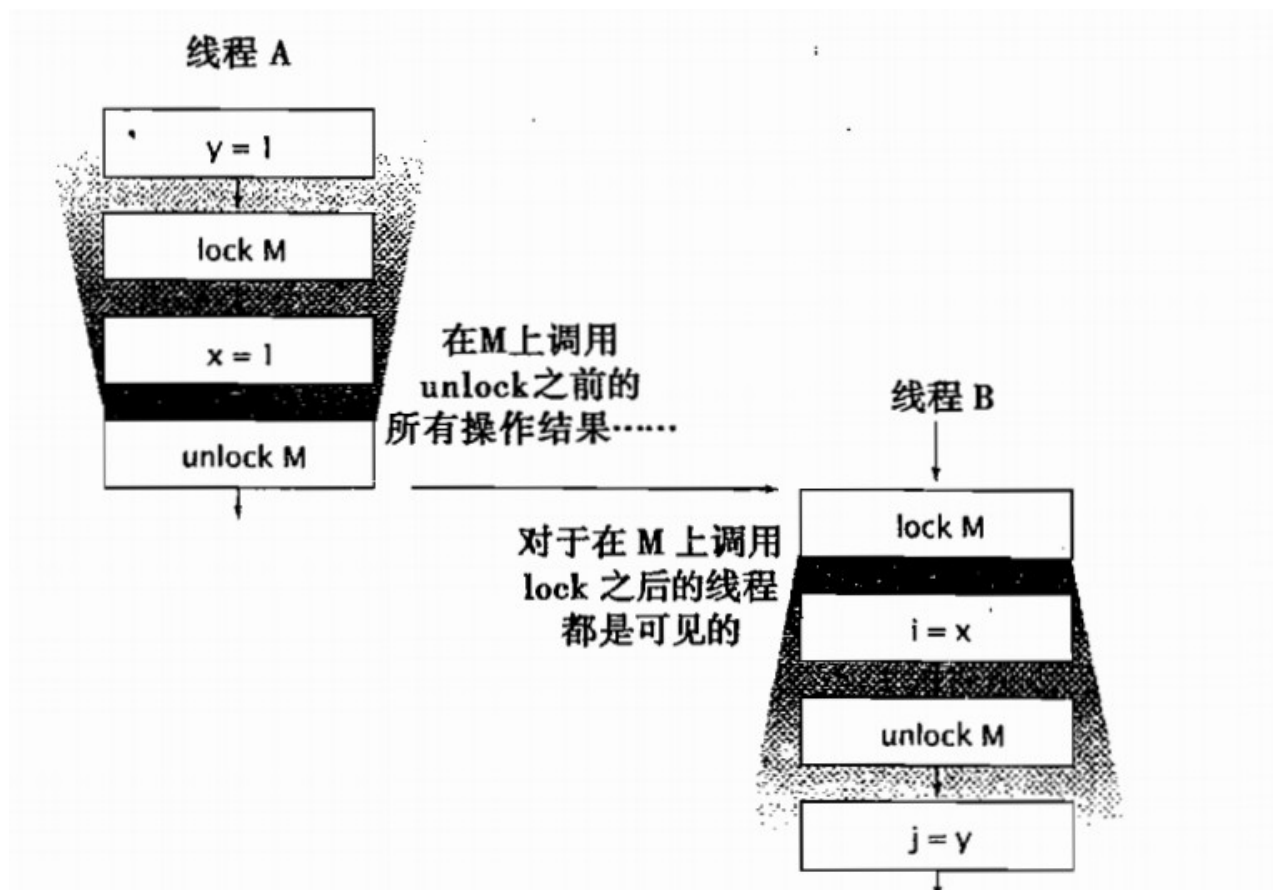
8. 互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现方式。synchronized 关键字经过编译后，会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令。根据虚拟机规范的要求，在执行 monitorenter 指令时，首先要尝试获取对象的锁，如果获得了锁，把锁的计数器加 1，相应地，在执行 monitorexit 指令时会将锁计数器减 1，当计数器为 0 时，锁便被释放了。由于 synchronized 同步块对同一个线程是可重入的，因此一个线程可以多次获得同一个对象的互斥锁，同样，要释放相应次数的该互斥锁，才能最终释放掉该锁。

9

synchronized 的另一个重要作用：内存可见性

加锁（synchronized 同步）的功能不仅仅局限于互斥行为，同时还存在另外一个重要的方面：内存可见性。我们不仅希望防止某个线程正在使用对象状态而另一个线程在同时修改该状态，而且还希望确保当一个线程修改了对象状态后，其他线程能够看到该变化。而线程的同步恰恰也能够实现这一点。

内置锁可以用于确保某个线程以一种可预测的方式来查看另一个线程的执行结果。为了确保所有的线程都能看到共享变量的最新值，可以在所有执行读操作或写操作的线程上加上同一把锁。下图示例了同步的可见性保证。



当线程 A 执行某个同步代码块时，线程 B 随后进入由同一个锁保护的同步代码块，这种情况下可以保证，当锁被释放前，A 看到的所有变量值（锁释放前，A 看到的变量包括 y 和 x ）在 B 获得同一个锁后同样可以由 B 看到。换句话说，当线程 B 执行由锁保护的同步代码块时，可以看到线程 A 之前在同一个锁保护的同步代码块中的所有操作结果。如果在线程 A `unlock M` 之后，线程 B 才进入 `lock M`，那么线程 B 都可以看到线程 A `unlock M` 之前的操作，可以得到 $i=1, j=1$ 。如果在线程 B `unlock M` 之后，线程 A 才进入 `lock M`，那么线程 B 就不一定能看到线程 A 中的操作，因此 j 的值就不一定是 1。

现在考虑如下代码：

```
public class MutableInteger
{
    private int value;

    public int get(){
```

```
    return value;
}
public void set(int value){
    this.value = value;
}
}
```

以上代码中，get 和 set 方法都在没有同步的情况下访问 value。如果 value 被多个线程共享，假如某个线程调用了 set，那么另一个正在调用 get 的线程可能会看到更新后的 value 值，也可能看不到。

通过对 set 和 get 方法进行同步，可以使 MutableInteger 成为一个线程安全的类，如下：

```
public class SynchronizedInteger
{
    private int value;

    public synchronized int get(){
        return value;
    }
    public synchronized void set(int value){
        this.value = value;
    }
}
```

对 set 和 get 方法进行了同步，加上了同一把对象锁，这样 get 方法可以看到 set 方法中 value 值的变化，从而每次通过 get 方法取得的 value 的值都是最新的 value 值。



10

实现内存可见性的两种方法比较：synchronized
和 Volatile



在《synchronized 的另一个重要作用：内存可见性》这篇文中，讲述了通过同步实现内存可见性的方法，在《Volatile 关键字（上）》这篇文中，讲述了通过 volatile 变量实现内存可见性的方法，这里比较下二者的区别。

- volatile 变量是一种稍弱的同步机制在访问 volatile 变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此 volatile 变量是一种比 synchronized 关键字更轻量级的同步机制。
- 从内存可见性的角度看，写入 volatile 变量相当于退出同步代码块，而读取 volatile 变量相当于进入同步代码块。
- 在代码中如果过度依赖 volatile 变量来控制状态的可见性，通常会比使用锁的代码更脆弱，也更难以理解。仅当 volatile 变量能简化代码的实现以及对同步策略的验证时，才应该使用它。一般来说，用同步机制会更安全些。
- 加锁机制（即同步机制）既可以确保可见性又可以确保原子性，而 volatile 变量只能确保可见性，原因是声明为 volatile 的简单变量如果当前值与该变量以前的值相关，那么 volatile 关键字不起作用，也就是说如下的表达式都不是原子操作：`count++`、`count = count+1`。

当且仅当满足以下所有条件时，才应该使用 volatile 变量：

- 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
- 该变量没有包含在其他变量的不变式中。

总结：在需要同步的时候，第一选择应该是 synchronized 关键字，这是最安全的方式，尝试其他任何方式都是有风险的。尤其在、jdk1.5 之后，对 synchronized 同步机制做了很多优化，如：自适应的自旋锁、锁粗化、锁消除、轻量级锁等，使得它的性能明显有了很大的提升。



11



多线程环境下安全使用集合 API



在集合 API 中，最初设计的 Vector 和 Hashtable 是多线程安全的。例如：对于 Vector 来说，用来添加和删除元素的方法是同步的。如果只有一个线程与 Vector 的实例交互，那么，要求获取和释放对象锁便是一种浪费，另外在不必要的时候如果滥用同步化，也有可能带来死锁。因此，对于更改集合内容的方法，没有一个是同步化的。集合本质上是非多线程安全的，当多个线程与集合交互时，为了使它多线程安全，必须采取额外的措施。

在 Collections 类中有多个静态方法，它们可以获取通过同步方法封装非同步集合而得到的集合：

- `public static Collection synchronizedCollection(Collection c)`
- `public static List synchronizedList(List l)`
- `public static Map synchronizedMap(Map m)`
- `public static Set synchronizedSet(Set s)`
- `public static SortedMap synchronizedSortedMap(SortedMap sm)`
- `public static SortedSet synchronizedSortedSet(SortedSet ss)`

这些方法基本上返回具有同步集合方法版本的新类。比如，为了创建多线程安全且由 ArrayList 支持的 List，可以使用如下代码：

```
List list = Collection.synchronizedList(new ArrayList());
```

注意，ArrayList 实例马上封装起来，不存在对未同步化 ArrayList 的直接引用（即直接封装匿名实例）。这是一种最安全的途径。如果另一个线程要直接引用 ArrayList 实例，它可以执行非同步修改。

下面给出一段多线程中安全遍历集合元素的示例。我们使用 Iterator 逐个扫描 List 中的元素，在多线程环境中，当遍历当前集合中的元素时，一般希望阻止其他线程添加或删除元素。安全遍历的实现方法如下：

```
import java.util.*;

public class SafeCollectionIteration extends Object {
    public static void main(String[] args) {
        //为了安全起见，仅使用同步列表的一个引用，这样可以确保控制了所有访问
        //集合必须同步化，这里是一个List
        List wordList = Collections.synchronizedList(new ArrayList());

        //wordList中的add方法是同步方法，会获取wordList实例的对象锁
        wordList.add("Iterators");
        wordList.add("require");
        wordList.add("special");
        wordList.add("handling");
    }
}
```

```
//获取wordList实例的对象锁，  
//迭代时，阻塞其他线程调用add或remove等方法修改元素  
synchronized ( wordList ) {  
    Iterator iter = wordList.iterator();  
    while ( iter.hasNext() ) {  
        String s = (String) iter.next();  
        System.out.println("found string: " + s + ", length=" + s.length());  
    }  
}  
}
```

这里需要注意的是：在 Java 语言中，大部分的线程安全类都是相对线程安全的，它能保证对这个对象单独的操作时线程安全的，我们在调用的时候不需要额外的保障措施，但是对于一些特定的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。例如 Vector、HashTable、Collections的synchronizedXxx x()方法包装的集合等。



T



12

死锁



当线程需要同时持有多个锁时，有可能产生死锁。考虑如下情形：

线程 A 当前持有互斥锁 lock1，线程 B 当前持有互斥锁 lock2。接下来，当线程 A 仍然持有 lock1 时，它试图获取 lock2，因为线程 B 正持有 lock2，因此线程 A 会阻塞等待线程 B 对 lock2 的释放。如果此时线程 B 在持有 lock2 的时候，也在试图获取 lock1，因为线程 A 正持有 lock1，因此线程 B 会阻塞等待 A 对 lock1 的释放。二者都在等待对方所持有锁的释放，而二者却又都没释放自己所持有的锁，这时二者便会一直阻塞下去。这种情形称为死锁。

下面给出一个两个线程间产生死锁的示例，如下：

```
public class Deadlock extends Object {
    private String objID;

    public Deadlock(String id) {
        objID = id;
    }

    public synchronized void checkOther(Deadlock other) {
        print("entering checkOther()");
        try { Thread.sleep(2000); }
        catch ( InterruptedException x ) { }
        print("in checkOther() - about to " + "invoke 'other.action()'");

        //调用other对象的action方法，由于该方法是同步方法，因此会试图获取other对象的对象锁
        other.action();
        print("leaving checkOther()");
    }

    public synchronized void action() {
        print("entering action()");
        try { Thread.sleep(500); }
        catch ( InterruptedException x ) { }
        print("leaving action()");
    }

    public void print(String msg) {
        threadPrint("objID=" + objID + " - " + msg);
    }

    public static void threadPrint(String msg) {
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + ": " + msg);
    }
}
```

```

public static void main(String[] args) {
    final Deadlock obj1 = new Deadlock("obj1");
    final Deadlock obj2 = new Deadlock("obj2");

    Runnable runA = new Runnable() {
        public void run() {
            obj1.checkOther(obj2);
        }
    };

    Thread threadA = new Thread(runA, "threadA");
    threadA.start();

    try { Thread.sleep(200); }
    catch ( InterruptedException x ) {}

    Runnable runB = new Runnable() {
        public void run() {
            obj2.checkOther(obj1);
        }
    };

    Thread threadB = new Thread(runB, "threadB");
    threadB.start();

    try { Thread.sleep(5000); }
    catch ( InterruptedException x ) {}

    threadPrint("finished sleeping");

    threadPrint("about to interrupt() threadA");
    threadA.interrupt();

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

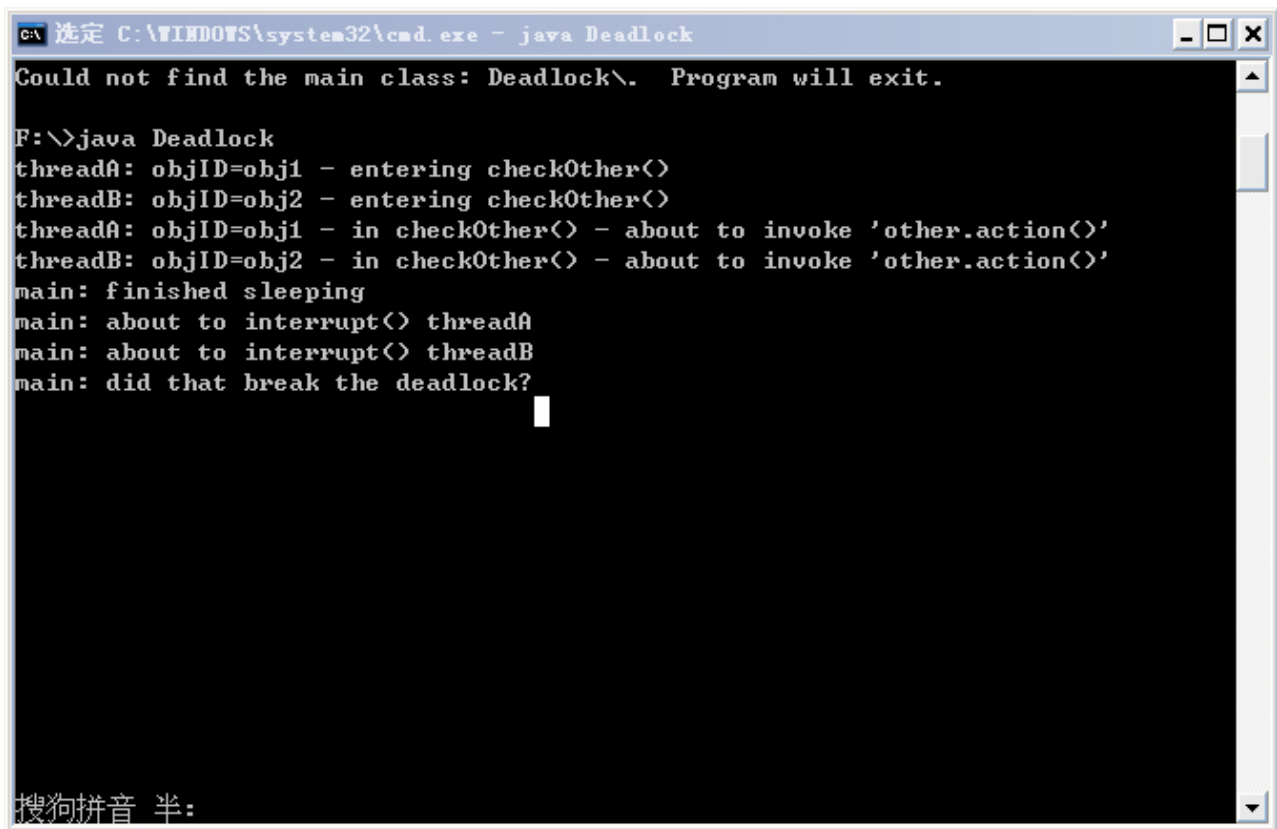
    threadPrint("about to interrupt() threadB");
    threadB.interrupt();

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

    threadPrint("did that break the deadlock?");
}
}

```

运行结果如下：



```
选定 C:\WINDOWS\system32\cmd.exe - java Deadlock
Could not find the main class: Deadlock\.. Program will exit.

F:\>java Deadlock
threadA: objID=obj1 - entering checkOther()
threadB: objID=obj2 - entering checkOther()
threadA: objID=obj1 - in checkOther() - about to invoke 'other.action()'
threadB: objID=obj2 - in checkOther() - about to invoke 'other.action()'
main: finished sleeping
main: about to interrupt() threadA
main: about to interrupt() threadB
main: did that break the deadlock?
```

从结果中可以看出，在执行到 `other.action()` 时，由于两个线程都在试图获取对方的锁，但对方都没有释放自己的锁，因而便产生了死锁，在主线程中试图中断两个线程，但都无果。

大部分代码并不容易产生死锁，死锁可能在代码中隐藏相当长的时间，等待不常见的条件地发生，但即使是很小的概率，一旦发生，便可能造成毁灭性的破坏。避免死锁是一件困难的事，遵循以下原则有助于规避死锁：

- 只在必要的最短时间内持有锁，考虑使用同步语句块代替整个同步方法；
- 尽量编写不在同一时刻需要持有多个锁的代码，如果不可避免，则确保线程持有第二个锁的时间尽量短暂；
- 创建和使用一个大锁来代替若干小锁，并把这个锁用于互斥，而不是用作单个对象的对象级别锁。



13

可重入内置锁



每个 Java 对象都可以用做一个实现同步的锁，这些锁被称为内置锁或监视器锁。线程在进入同步代码块之前会自动获取锁，并且在退出同步代码块时会自动释放锁。获得内置锁的唯一途径就是进入由这个锁保护的同步代码块或方法。

当某个线程请求一个由其他线程持有的锁时，发出请求的线程就会阻塞。然而，由于内置锁是可重入的，因此如果摸个线程试图获得一个已经由它自己持有的锁，那么这个请求就会成功。“重入”意味着获取锁的操作的粒度是“线程”，而不是调用。重入的一种实现方法是，为每个锁关联一个获取计数值和一个所有者线程。当计数值为 0 时，这个锁就被认为没有被任何线程所持有，当线程请求一个未被持有的锁时，JVM 将记下锁的持有者，并且将获取计数值置为 1，如果同一个线程再次获取这个锁，计数值将递增，而当线程退出同步代码块时，计数器会相应地递减。当计数值为 0 时，这个锁将被释放。

重入进一步提升了加锁行为的封装性，因此简化了面向对象并发代码的开发。分析如下程序：

```
public class Father
{
    public synchronized void doSomething(){
        .....
    }
}

public class Child extends Father
{
    public synchronized void doSomething(){
        .....
        super.doSomething();
    }
}
```

子类覆写了父类的同步方法，然后调用父类中的方法，此时如果没有可重入的锁，那么这段代码件产生死锁。

由于 Fither 和 Child 中的 doSomething 方法都是 synchronized 方法，因此每个 doSomething 方法在执行前都会获取 Child 对象实例上的锁。如果内置锁不是可重入的，那么在调用 super.doSomething 时将无法获得该 Child 对象上的互斥锁，因为这个锁已经被持有，从而线程会永远阻塞下去，一直在等待一个永远也无法获取的锁。重入则避免了这种死锁情况的发生。

同一个线程在调用本类中其他 synchronized 方法/块或父类中的 synchronized 方法/块时，都不会阻碍该线程地执行，因为互斥锁时可重入的。



14

线程间协作：wait、notify、notifyAll



在 Java 中，可以通过配合调用 Object 对象的 wait() 方法和 notify() 方法或 notifyAll() 方法来实现线程间的通信。在线程中调用 wait() 方法，将阻塞等待其他线程的通知（其他线程调用 notify() 方法或 notifyAll() 方法），在线程中调用 notify() 方法或 notifyAll() 方法，将通知其他线程从 wait() 方法处返回。

Object 是所有类的超类，它有 5 个方法组成了等待/通知机制的核心：notify()、notifyAll()、wait()、wait(long) 和 wait(long, int)。在 Java 中，所有的类都从 Object 继承而来，因此，所有的类都拥有这些共有方法可供使用。而且，由于他们都被声明为 final，因此在子类中不能覆写任何一个方法。

这里详细说明一下各个方法在使用中需要注意的几点。

wait()

```
public final void wait() throws InterruptedException,IllegalMonitorStateException
```

该方法用来将当前线程置入休眠状态，直到接到通知或被中断为止。在调用 wait() 之前，线程必须要获得该对象的对象级别锁，即只能在同步方法或同步块中调用 wait() 方法。进入 wait() 方法后，当前线程释放锁。在从 wait() 返回前，线程与其他线程竞争重新获得锁。如果调用 wait() 时，没有持有适当的锁，则抛出 IllegalMonitorStateException，它是 RuntimeException 的一个子类，因此，不需要 try-catch 结构。

notify()

```
public final native void notify() throws IllegalMonitorStateException
```

该方法也要在同步方法或同步块中调用，即在调用前，线程也必须要获得该对象的对象级别锁，如果调用 notify() 时没有持有适当的锁，也会抛出 IllegalMonitorStateException。

该方法用来通知那些可能等待该对象的对象锁的其他线程。如果有多个线程等待，则线程规划器任意挑选出其中一个 wait() 状态的线程来发出通知，并使它等待获取该对象的对象锁（notify 后，当前线程不会马上释放该对象锁，wait 所在的线程并不能马上获取该对象锁，要等到程序退出 synchronized 代码块后，当前线程才会释放锁，wait 所在的线程也才可以获取该对象锁），但不惊动其他同样在等待被该对象 notify 的线程们。当第一个获得了该对象锁的 wait 线程运行完毕以后，它会释放掉该对象锁，此时如果该对象没有再次使用 notify 语句，则即便该对象已经空闲，其他 wait 状态等待的线程由于没有得到该对象的通知，会继续阻塞在 wait 状态，直到这个对象发出一个 notify 或 notifyAll。这里需要注意：它们等待的是被 notify 或 notifyAll，而不是锁。这与下面的 notifyAll() 方法执行后的情况不同。

notifyAll()

```
public final native void notifyAll() throws IllegalMonitorStateException
```

该方法与 notify ()方法的工作方式相同，重要的一点差异是：

notifyAll 使所有原来在该对象上 wait 的线程统统退出 wait 的状态（即全部被唤醒，不再等待 notify 或 notifyAll，但由于此时还没有获取到该对象锁，因此还不能继续往下执行），变成等待获取该对象上的锁，一旦该对象锁被释放（notifyAll 线程退出调用了 notifyAll 的 synchronized 代码块的时候），他们就会去竞争。如果其中一个线程获得了该对象锁，它就会继续往下执行，在它退出 synchronized 代码块，释放锁后，其他的已经被唤醒的线程将会继续竞争获取该锁，一直进行下去，直到所有被唤醒的线程都执行完毕。

深入理解

如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。

优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。



15

notify 通知的遗漏



notify 通知的遗漏很容易理解，即 threadA 还没开始 wait 的时候，threadB 已经 notify 了，这样，threadB 通知是没有任何响应的，当 threadB 退出 synchronized 代码块后，threadA 再开始 wait，便会一直阻塞等待，直到被别的线程打断。

遗漏通知的代码

下面给出一段代码演示通知是如何遗漏的，如下：

```
public class MissedNotify extends Object {
    private Object proceedLock;

    public MissedNotify() {
        print("in MissedNotify()");
        proceedLock = new Object();
    }

    public void waitToProceed() throws InterruptedException {
        print("in waitToProceed() - entered");

        synchronized (proceedLock) {
            print("in waitToProceed() - about to wait()");
            proceedLock.wait();
            print("in waitToProceed() - back from wait()");
        }

        print("in waitToProceed() - leaving");
    }

    public void proceed() {
        print("in proceed() - entered");

        synchronized (proceedLock) {
            print("in proceed() - about to notifyAll()");
            proceedLock.notifyAll();
            print("in proceed() - back from notifyAll()");
        }

        print("in proceed() - leaving");
    }

    private static void print(String msg) {
        String name = Thread.currentThread().getName();
        System.out.println(name + ": " + msg);
    }
}
```

```

}

public static void main(String[] args) {
    final MissedNotify mn = new MissedNotify();

    Runnable runA = new Runnable() {
        public void run() {
            try {
                //休眠1000ms，大于runB中的500ms，
                //是为了后调用waitToProceed，从而先notifyAll，后wait，
                //从而造成通知的遗漏
                Thread.sleep(1000);
                mn.waitToProceed();
            } catch ( InterruptedException x ) {
                x.printStackTrace();
            }
        }
    };

    Thread threadA = new Thread(runA, "threadA");
    threadA.start();

    Runnable runB = new Runnable() {
        public void run() {
            try {
                //休眠500ms，小于runA中的1000ms，
                //是为了先调用proceed，从而先notifyAll，后wait，
                //从而造成通知的遗漏
                Thread.sleep(500);
                mn.proceed();
            } catch ( InterruptedException x ) {
                x.printStackTrace();
            }
        }
    };

    Thread threadB = new Thread(runB, "threadB");
    threadB.start();

    try {
        Thread.sleep(10000);
    } catch ( InterruptedException x ) {}

    //试图打断wait阻塞
    print("about to invoke interrupt() on threadA");
}

```

```

        threadA.interrupt();
    }
}

```

执行结果如下：

```

C:\WINDOWS\system32\cmd.exe

F:\chapter08>java MissedNotify
main: in MissedNotify()
threadB: in proceed() - entered
threadB: in proceed() - about to notifyAll()
threadB: in proceed() - back from notifyAll()
threadB: in proceed() - leaving
threadA: in waitToProceed() - entered
threadA: in waitToProceed() - about to wait()
main: about to invoke interrupt() on threadA
java.lang.InterruptedException
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:485)
    at MissedNotify.waitToProceed(MissedNotify.java:14)
    at MissedNotify$1.run(MissedNotify.java:45)
    at java.lang.Thread.run(Unknown Source)

F:\chapter08>
搜狗拼音 半:

```

分析：由于 threadB 在执行 mn.proceed() 之前只休眠了 500ms，而 threadA 在执行 mn.waitToProceed() 之前休眠了 1000ms，因此，threadB 会先苏醒，继而执行 mn.proceed()，获取到 proceedLock 的对象锁，继而执行其中的 notifyAll()，当退出 proceed() 方法中的 synchronized 代码块时，threadA 才有机会获取 proceedLock 的对象锁，继而执行其中的 wait() 方法，但此时 notifyAll() 方法已经执行完毕，threadA 便漏掉了 threadB 的通知，便会阻塞下去。后面主线程休眠 10 秒后，尝试中断 threadA 线程，使其抛出 InterruptedException。

修正后的代码

为了修正 MissedNotify，需要添加一个 boolean 指示变量，该变量只能在同步代码块内部访问和修改。修改后的代码如下：

```

public class MissedNotifyFix extends Object {
    private Object proceedLock;
    //该标志位用来指示线程是否需要等待

```



```

private boolean okToProceed;

public MissedNotifyFix() {
    print("in MissedNotifyFix()");
    proceedLock = new Object();
    //先设置为false
    okToProceed = false;
}

public void waitToProceed() throws InterruptedException {
    print("in waitToProceed() - entered");

    synchronized ( proceedLock ) {
        print("in waitToProceed() - entered sync block");
        //while循环判断，这里不用if的原因是为了防止早期通知
        while ( okToProceed == false ) {
            print("in waitToProceed() - about to wait()");
            proceedLock.wait();
            print("in waitToProceed() - back from wait()");
        }

        print("in waitToProceed() - leaving sync block");
    }

    print("in waitToProceed() - leaving");
}

public void proceed() {
    print("in proceed() - entered");

    synchronized ( proceedLock ) {
        print("in proceed() - entered sync block");
        //通知之前，将其设置为true，这样即使出现通知遗漏的情况，也不会使线程在wait出阻塞
        okToProceed = true;
        print("in proceed() - changed okToProceed to true");
        proceedLock.notifyAll();
        print("in proceed() - just did notifyAll()");

        print("in proceed() - leaving sync block");
    }

    print("in proceed() - leaving");
}

private static void print(String msg) {

```

```

String name = Thread.currentThread().getName();
System.out.println(name + ": " + msg);
}

public static void main(String[] args) {
    final MissedNotifyFix mnf = new MissedNotifyFix();

    Runnable runA = new Runnable() {
        public void run() {
            try {
                //休眠1000ms，大于runB中的500ms，
                //是为了后调用waitToProceed，从而先notifyAll，后wait，
                Thread.sleep(1000);
                mnf.waitToProceed();
            } catch ( InterruptedException x ) {
                x.printStackTrace();
            }
        }
    };

    Thread threadA = new Thread(runA, "threadA");
    threadA.start();

    Runnable runB = new Runnable() {
        public void run() {
            try {
                //休眠500ms，小于runA中的1000ms，
                //是为了先调用proceed，从而先notifyAll，后wait，
                Thread.sleep(500);
                mnf.proceed();
            } catch ( InterruptedException x ) {
                x.printStackTrace();
            }
        }
    };

    Thread threadB = new Thread(runB, "threadB");
    threadB.start();

    try {
        Thread.sleep(10000);
    } catch ( InterruptedException x ) {}

    print("about to invoke interrupt() on threadA");
    threadA.interrupt();
}

```

```
}
}
```

执行结果如下：

```
C:\WINDOWS\system32\cmd.exe

F:\chapter08>java MissedNotifyFix
main: in MissedNotify()
threadB: in proceed() - entered
threadB: in proceed() - entered sync block
threadB: in proceed() - changed okToProceed to true
threadB: in proceed() - just did notifyAll()
threadB: in proceed() - leaving sync block
threadB: in proceed() - leaving
threadA: in waitToProceed() - entered
threadA: in waitToProceed() - entered sync block
threadA: in waitToProceed() - leaving sync block
threadA: in waitToProceed() - leaving
main: about to invoke interrupt() on threadA

F:\chapter08>
```

注意代码中加了注释的部分，在 threadB 进行通知之前，先将 okToProceed 置为 true，这样如果 threadA 将通知遗漏，那么就不会进入 while 循环，也便不会执行 wait 方法，线程也就不会阻塞。如果通知没有被遗漏，wait 方法返回后，okToProceed 已经被置为 true，下次 while 循环判断条件不成立，便会退出循环。

这样，通过标志位和 wait、notifyAll 的配合使用，便避免了通知遗漏而造成的阻塞问题。

总结：在使用线程的等待/通知机制时，一般都要配合一个 boolean 变量值（或者其他能够判断真假的条件），在 notify 之前改变该 boolean 变量的值，让 wait 返回后能够退出 while 循环（一般都要在 wait 方法外围加一层 while 循环，以防止早期通知），或在通知被遗漏后，不会被阻塞在 wait 方法处。这样便保证了程序的正确性。



16

notifyAll 造成的早期通知问题



如果线程在等待时接到通知，但线程等待的条件还不满足，此时，线程接到的就是早期通知，如果条件满足的时间很短，但很快又改变了，而变得不再满足，这时也将发生早期通知。这种现象听起来很奇怪，下面通过一个示例程序来说明问题。

很简单，两个线程等待删除 List 中的元素，同时另外一个线程正要向其中添加项目。代码如下：

```
import java.util.*;

public class EarlyNotify extends Object {
    private List list;

    public EarlyNotify() {
        list = Collections.synchronizedList(new LinkedList());
    }

    public String removeItem() throws InterruptedException {
        print("in removeItem() - entering");

        synchronized ( list ) {
            if ( list.isEmpty() ) { //这里用if语句会发生危险
                print("in removeItem() - about to wait()");
                list.wait();
                print("in removeItem() - done with wait()");
            }

            //删除元素
            String item = (String) list.remove(0);

            print("in removeItem() - leaving");
            return item;
        }
    }

    public void addItem(String item) {
        print("in addItem() - entering");
        synchronized ( list ) {
            //添加元素
            list.add(item);
            print("in addItem() - just added: " + item + "");

            //添加后，通知所有线程
            list.notifyAll();
            print("in addItem() - just notified");
        }
        print("in addItem() - leaving");
    }
}
```

```

}

private static void print(String msg) {
    String name = Thread.currentThread().getName();
    System.out.println(name + ": " + msg);
}

public static void main(String[] args) {
    final EarlyNotify en = new EarlyNotify();

    Runnable runA = new Runnable() {
        public void run() {
            try {
                String item = en.removeItem();
                print("in run() - returned: '" +
                    item + "'");
            } catch ( InterruptedException ix ) {
                print("interrupted!");
            } catch ( Exception x ) {
                print("threw an Exception!!!\n" + x);
            }
        }
    };

    Runnable runB = new Runnable() {
        public void run() {
            en.addItem("Hello!");
        }
    };

    try {
        //启动第一个删除元素的线程
        Thread threadA1 = new Thread(runA, "threadA1");
        threadA1.start();

        Thread.sleep(500);

        //启动第二个删除元素的线程
        Thread threadA2 = new Thread(runA, "threadA2");
        threadA2.start();

        Thread.sleep(500);
        //启动增加元素的线程
        Thread threadB = new Thread(runB, "threadB");
        threadB.start();
    }
}

```

```

        Thread.sleep(10000); // wait 10 seconds

        threadA1.interrupt();
        threadA2.interrupt();
    } catch ( InterruptedException x ) {}
}
}

```

执行结果如下：

```

C:\WINDOWS\system32\cmd.exe

F:\chapter08>java EarlyNotify
threadA1: in removeItem() - entering
threadA1: in removeItem() - about to wait()
threadA2: in removeItem() - entering
threadA2: in removeItem() - about to wait()
threadB: in addItem() - entering
threadB: in addItem() - just added: 'Hello!'
threadB: in addItem() - just notified
threadB: in addItem() - leaving
threadA2: in removeItem() - done with wait()
threadA2: in removeItem() - leaving
threadA1: in removeItem() - done with wait()
threadA2: in run() - returned: 'Hello!'
threadA1: threw an Exception!!!
java.lang.IndexOutOfBoundsException: Index: 0, Size: 0

F:\chapter08>

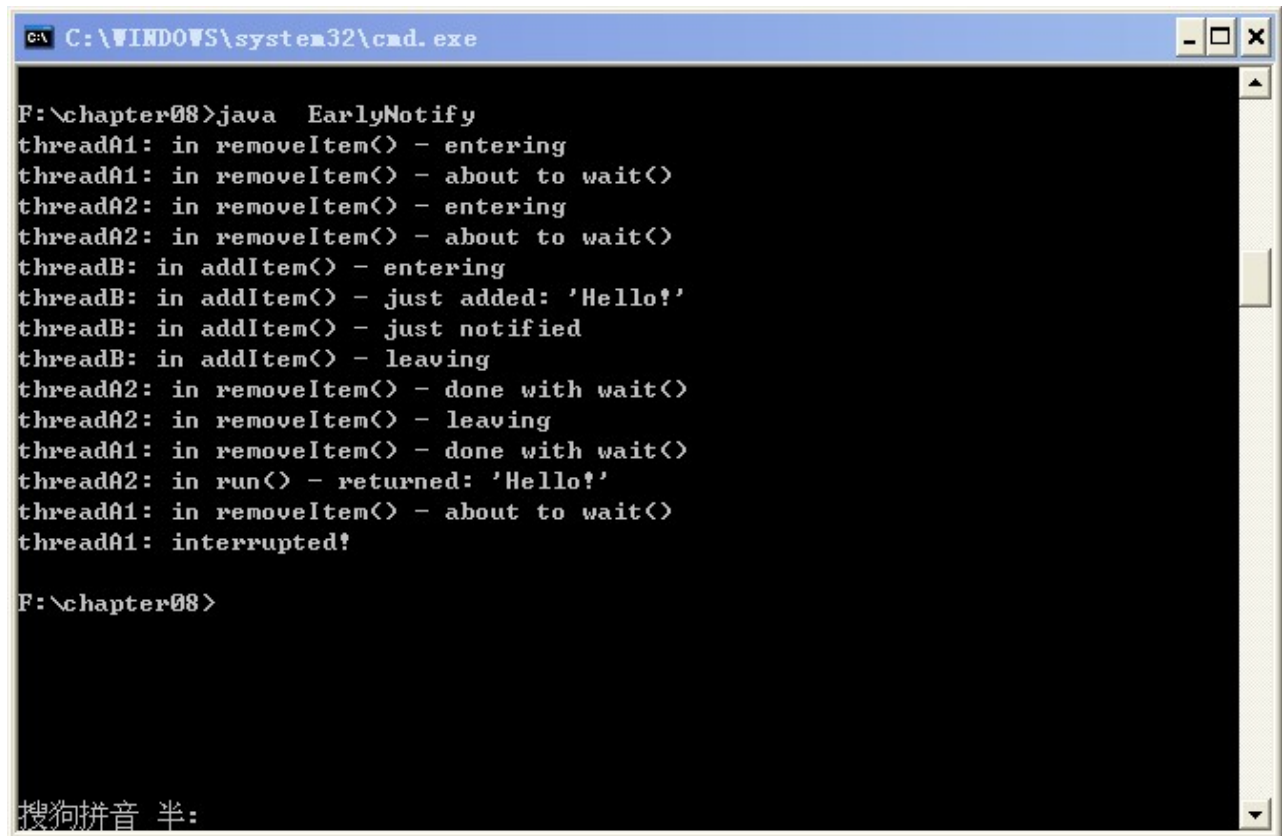
```

分析：首先启动 threadA1，threadA1 在 removeItem() 中调用 wait()，从而释放 list 上的对象锁。再过 500ms，启动 threadA2，threadA2 调用 removeItem()，获取 list 上的对象锁，也发现列表为空，从而在 wait() 方法处阻塞，释放 list 上的对象锁。再过 500ms 后，启动 threadB，并调用 addItem，获得 list 上的对象锁，并在 list 中添加一个元素，同时用 notifyAll 通知所有线程。

threadA1 和 threadA2 都从 wait() 返回，等待获取 list 对象上的对象锁，并试图从列表中删除添加的元素，这就就会产生麻烦，只有其中一个操作能成功。假设 threadA1 获取了 list 上的对象锁，并删除元素成功，在退出 synchronized 代码块时，它便会释放 list 上的对象锁，此时 threadA2 便会获取 list 上的对象锁，会继续删除 list 中的元素，但是 list 已经为空了，这便会抛出 IndexOutOfBoundsException。

要避免以上问题只需将 wait 外围的 if 语句改为 while 循环即可，这样当 list 为空时，线程便会继续等待，而不会继续去执行删除 list 中元素的代码。

修改后的执行结果如下：



```
C:\WINDOWS\system32\cmd.exe

F:\chapter08>java EarlyNotify
threadA1: in removeItem() - entering
threadA1: in removeItem() - about to wait()
threadA2: in removeItem() - entering
threadA2: in removeItem() - about to wait()
threadB: in addItem() - entering
threadB: in addItem() - just added: 'Hello!'
threadB: in addItem() - just notified
threadB: in addItem() - leaving
threadA2: in removeItem() - done with wait()
threadA2: in removeItem() - leaving
threadA1: in removeItem() - done with wait()
threadA2: in run() - returned: 'Hello!'
threadA1: in removeItem() - about to wait()
threadA1: interrupted!

F:\chapter08>
```

搜狗拼音 半:

总结：在使用线程的等待/通知机制时，一般都要在 while 循环中调用 wait()方法，满足条件时，才让 while 循环退出，这样一般也要配合使用一个 boolean 变量（或其他能判断真假的条件，如本文中的 list.isEmpty()），满足 while 循环的条件时，进入 while 循环，执行 wait()方法，不满足 while 循环的条件时，跳出循环，执行后面的代码。



17

生产者-消费者模型



生产者消费者问题是线程模型中的经典问题：生产者和消费者在同一时间段内共用同一存储空间，生产者向空间里生产数据，而消费者取走数据。

这里实现如下情况的生产—消费模型：

生产者不断交替地生产两组数据“姓名--1 --> 内容--1”，“姓名--2--> 内容--2”，消费者不断交替地取得这两组数据，这里的“姓名--1”和“姓名--2”模拟为数据的名称，“内容--1”和“内容--2”模拟为数据的内容。

由于本程序中牵扯到线程运行的不确定性，因此可能会出现以下问题：

- 假设生产者线程刚向数据存储空间添加了数据的名称，还没有加入该信息的内容，程序就切换到了消费者线程，消费者线程将把信息的名称和上一个信息的内容联系在一起；
- 生产者生产了若干次数据，消费者才开始取数据，或者是，消费者取完一次数据后，还没等生产者放入新的数据，又重复取出了已取过的数据。

问题 1 很明显要靠同步来解决，问题 2 则需要线程间通信，生产者线程放入数据后，通知消费者线程取出数据，消费者线程取出数据后，通知生产者线程生产数据，这里用 wait/notify 机制来实现。

详细的实现代码如下：

```
class Info{ // 定义信息类
    private String name = "name";//定义name属性，为了与下面set的name属性区别开
    private String content = "content" ;// 定义content属性，为了与下面set的content属性区别开
    private boolean flag = true ; // 设置标志位,初始时先生产
    public synchronized void set(String name,String content){
        while(!flag){
            try{
                super.wait() ;
            }catch(InterruptedException e){
                e.printStackTrace() ;
            }
        }
        this.setName(name) ; // 设置名称
        try{
            Thread.sleep(300) ;
        }catch(InterruptedException e){
            e.printStackTrace() ;
        }
        this.setContent(content) ; // 设置内容
        flag = false ; // 改变标志位，表示可以取走
        super.notify();
    }
}
```

```

public synchronized void get(){
    while(flag){
        try{
            super.wait() ;
        }catch(InterruptedException e){
            e.printStackTrace() ;
        }
    }
    try{
        Thread.sleep(300) ;
    }catch(InterruptedException e){
        e.printStackTrace() ;
    }
    System.out.println(this.getName() +
        " --> " + this.getContent());
    flag = true ; // 改变标志位，表示可以生产
    super.notify();
}
public void setName(String name){
    this.name = name ;
}
public void setContent(String content){
    this.content = content ;
}
public String getName(){
    return this.name ;
}
public String getContent(){
    return this.content ;
}
}

class Producer implements Runnable{ // 通过Runnable实现多线程
    private Info info = null ;    // 保存Info引用
    public Producer(Info info){
        this.info = info ;
    }
    public void run(){
        boolean flag = true ; // 定义标记位
        for(int i=0;i<10;i++){
            if(flag){
                this.info.set("姓名--1","内容--1") ; // 设置名称
                flag = false ;
            }else{
                this.info.set("姓名--2","内容--2") ; // 设置名称
                flag = true ;
            }
        }
    }
}

```

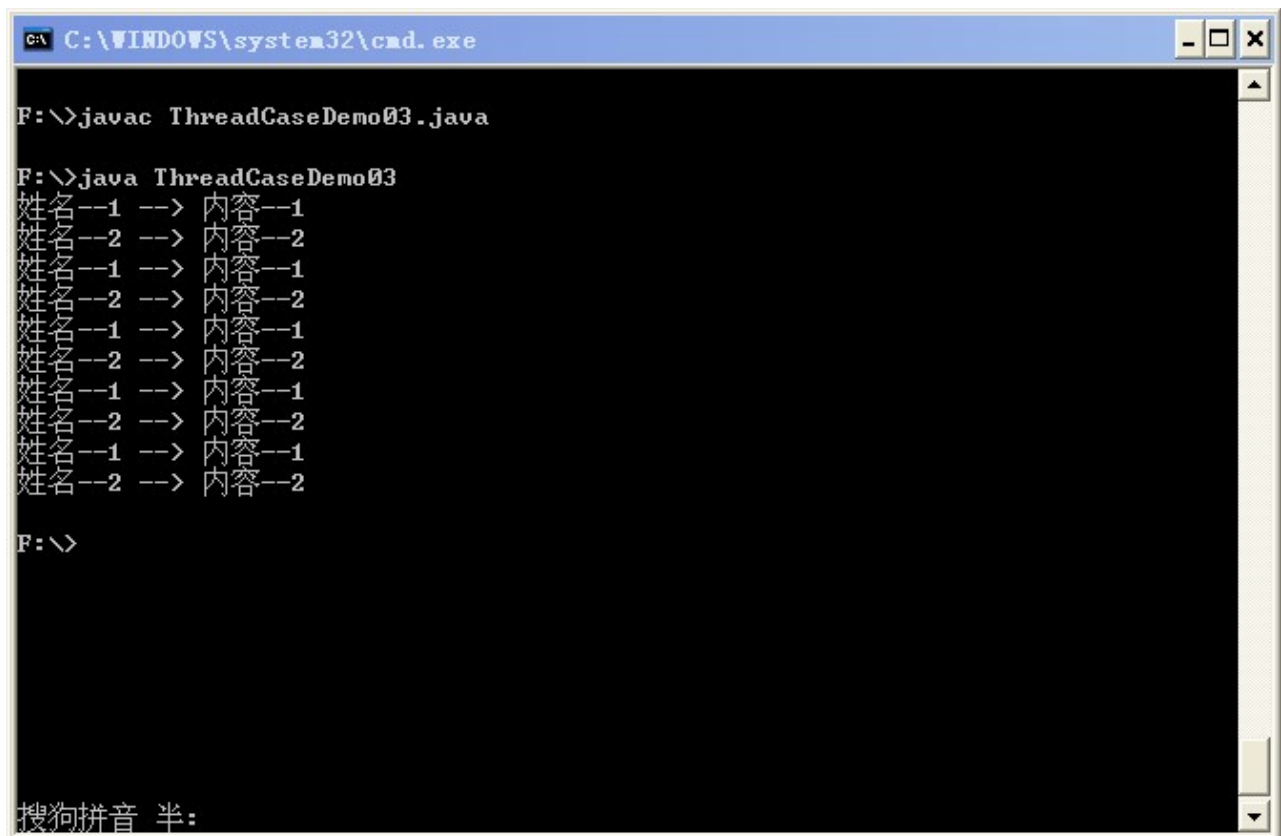
```

    }
}
}
}
class Consumer implements Runnable{
    private Info info = null ;
    public Consumer(Info info){
        this.info = info ;
    }
    public void run(){
        for(int i=0;i<10;i++){
            this.info.get() ;
        }
    }
}
}
public class ThreadCaseDemo03{
    public static void main(String args[]){
        Info info = new Info(); // 实例化Info对象
        Producer pro = new Producer(info) ; // 生产者
        Consumer con = new Consumer(info) ; // 消费者
        new Thread(pro).start() ;
        //启动了生产者线程后，再启动消费者线程
        try{
            Thread.sleep(500) ;
        }catch(InterruptedException e){
            e.printStackTrace() ;
        }

        new Thread(con).start() ;
    }
}

```

执行结果如下：



```
C:\WINDOWS\system32\cmd.exe

F:\>javac ThreadCaseDemo03.java

F:\>java ThreadCaseDemo03
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2

F:\>

搜狗拼音 半:
```

另外，在 run 方法中，二者循环的次数要相同，否则，当一方的循环结束时，另一方的循环依然继续，它会阻塞在 wait()方法处，而等不到对方的 notify 通知。



18

深入 Java 内存模型（1）



happen-before 规则介绍

Java 语言中有一个“先行发生”（happen-before）的规则，它是 Java 内存模型中定义的两项操作之间的偏序关系，如果操作 A 先行发生于操作 B，其意思就是说，在发生操作 B 之前，操作 A 产生的影响都能被操作 B 观察到，“影响”包括修改了内存中共享变量的值、发送了消息、调用了方法等，它与时间上的先后发生基本没有太大关系。这个原则特别重要，它是判断数据是否存在竞争、线程是否安全的主要依据。

举例来说，假设存在如下三个线程，分别执行对应的操作：

- 线程 A 中执行如下操作：i=1
- 线程 B 中执行如下操作：j=i
- 线程 C 中执行如下操作：i=2

假设线程 A 中的操作“i=1”happen-before 线程 B 中的操作“j=i”，那么就可以保证在线程 B 的操作执行后，变量 j 的值一定为 1，即线程 B 观察到了线程 A 中操作“i=1”所产生的影响；现在，我们依然保持线程 A 和线程 B 之间的 happen-before 关系，同时线程 C 出现在了线程 A 和线程 B 的操作之间，但是 C 与 B 并没有 happen-before 关系，那么 j 的值就不确定了，线程 C 对变量 i 的影响可能会被线程 B 观察到，也可能不会，这时线程 B 就存在读取到不是最新数据的风险，不具备线程安全性。

下面是 Java 内存模型中的八条可保证 happen-before 的规则，它们无需任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随机地重排序。

1. 程序次序规则：在一个单独的线程中，按照程序代码的执行流顺序，（时间上）先执行的操作 happen-before（时间上）后执行的操作。
2. 管理锁定规则：一个 unlock 操作 happen-before 后面（时间上的先后顺序，下同）对同一个锁的 lock 操作。
3. volatile 变量规则：对一个 volatile 变量的写操作 happen-before 后面对该变量的读操作。
4. 线程启动规则：Thread 对象的 start() 方法 happen-before 此线程的每一个动作。
5. 线程终止规则：线程的所有操作都 happen-before 对此线程的终止检测，可以通过 Thread.join() 方法结束 Thread.isAlive() 的返回值等手段检测到线程已经终止执行。
6. 线程中断规则：对线程 interrupt() 方法的调用 happen-before 发生于被中断线程的代码检测到中断时事件的发生。

7. 对象终结规则：一个对象的初始化完成（构造函数执行结束）happen—before 它的 finalize()方法的开始。
8. 传递性：如果操作 A happen—before 操作 B，操作 B happen—before 操作 C，那么可以得出 A happen—before 操作 C。

时间上先后顺序和 happen—before 原则

”时间上执行的先后顺序“与”happen—before“之间有何不同呢？

首先来看操作 A 在时间上先与操作 B 发生，是否意味着操作 A happen—before 操作 B？

一个常用来分析的例子如下：

```
private int value = 0;

public int get(){
    return value;
}

public void set(int value){
    this.value = value;
}
```

假设存在线程 A 和线程 B，线程 A 先（时间上的先）调用了 setValue(3)操作，然后（时间上的后）线程B调用了同一对象的 getValue()方法，那么线程B得到的返回值一定是3吗？

对照以上八条 happen—before 规则，发现没有一条规则适合于这里的 value 变量，从而我们可以判定线程 A 中的 setValue(3)操作与线程 B 中的 getValue()操作不存在 happen—before 关系。因此，尽管线程 A 的 setValue(3)在操作时间上先于操作 B 的 getvalue()，但无法保证线程 B 的 getValue()操作一定观察到了线程 A 的 setValue(3)操作所产生的结果，也即是 getValue()的返回值不一定为 3（有可能是之前 setValue 所设置的值）。这里的操作不是线程安全的。

因此，”一个操作时间上先发生于另一个操作“并不代表”一个操作 happen—before 另一个操作“。

解决方法：可以将 setValue（int）方法和 getValue()方法均定义为 synchronized 方法，也可以把 value 定义为 volatile 变量（value 的修改并不依赖 value 的原值，符合 volatile 的使用场景），分别对应 happen—before 规则的第 2 和第 3 条。注意，只将 setValue（int）方法和 getvalue()方法中的一个定义为 synchronized 方法是不行的，必须对同一个变量的所有读写同步，才能保证不读取到陈旧的数据，仅仅同步读或写是不够的。

其次来看，操作 A happen—before 操作 B，是否意味着操作 A 在时间上先与操作 B 发生？

看有如下代码：


```
x = 1;
y = 2;
```

假设同一个线程执行上面两个操作：操作 A：x=1 和操作 B：y=2。根据 happen-before 规则的第 1 条，操作 A happen-before 操作 B，但是由于编译器的指令重排序（Java 语言规范规定了 JVM 线程内部维持顺序化语义，也就是说只要程序的最终结果等同于它在严格的顺序化环境下的结果，那么指令的执行顺序就可能与代码的顺序不一致。这个过程通过叫做指令的重排序。指令重排序存在的意义在于：JVM 能够根据处理器的特性（CPU 的多级缓存系统、多核处理器等）适当的重新排序机器指令，使机器指令更符合 CPU 的执行特点，最大限度的发挥机器的性能。在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整）等原因，操作 A 在时间上有可能后于操作 B 被处理器执行，但这并不影响 happen-before 原则的正确性。

因此，”一个操作 happen-before 另一个操作“并不代表”一个操作时间上先发生于另一个操作“。

最后，一个操作和另一个操作必定存在某个顺序，要么一个操作或者是先于或者是后于另一个操作，或者与两个操作同时发生。同时发生是完全可能存在的，特别是在多 CPU 的情况下。而两个操作之间却可能没有 happen-before 关系，也就是说有可能发生这样的情况，操作 A 不 happen-before 操作 B，操作 B 也不 happen-before 操作 A，用数学上的术语 happen-before 关系是个偏序关系。两个存在 happen-before 关系的操作不可能同时发生，一个操作 A happen-before 操作 B，它们必定在时间上是完全错开的，这实际上也是同步的语义之一（独占访问）。

利用 happen-before 规则分析 DCL

DCL 即双重检查加锁，下面是一个典型的在单例模式中使用 DCL 的例子：

```
public class LazySingleton {
    private int someField;

    private static LazySingleton instance;

    private LazySingleton() {
        this.someField = new Random().nextInt(200)+1;    // (1)
    }

    public static LazySingleton getInstance() {
        if (instance == null) {                          // (2)
            synchronized(LazySingleton.class) {         // (3)
                if (instance == null) {                  // (4)
                    instance = new LazySingleton();      // (5)
                }
            }
        }
    }
}
```

```

    }
    return instance;                // (6)
}

public int getSomeField() {
    return this.someField;          // (7)
}
}

```

这里得到单一的 instance 实例是没有问题的，问题的关键在于尽管得到了 Singleton 的正确引用，但是却有可能访问到其成员变量的不正确值。具体来说 Singleton.getInstance().getSomeField() 有可能返回 someField 的默认值 0。如果程序行为正确的话，这应当是不可能发生的事，因为在构造函数里设置的 someField 的值不可能为 0。为也说明这种情况理论上有可能发生，我们只需要说明语句(1)和语句(7)并不存在 happen-before 关系。

假设线程 I 是初次调用 getInstance() 方法，紧接着线程 II 也调用了 getInstance() 方法和 getSomeField() 方法，我们要说明的是线程 I 的语句(1)并不 happen-before 线程 II 的语句(7)。线程 II 在执行 getInstance() 方法的语句(2)时，由于对 instance 的访问并没有处于同步块中，因此线程 II 可能观察到也可能观察不到线程 I 在语句(5)时对 instance 的写入，也就是说 instance 的值可能为空也可能为非空。我们先假设 instance 的值非空，也就观察到了线程 I 对 instance 的写入，这时线程 II 就会执行语句(6)直接返回这个 instance 的值，然后对这个 instance 调用 getSomeField() 方法，该方法也是在没有任何同步情况被调用，因此整个线程 II 的操作都是在没有同步的情况下调用，这时我们便无法利用上述 8 条 happen-before 规则得到线程 I 的操作和线程 II 的操作之间的任何有效的 happen-before 关系（主要考虑规则的第 2 条，但由于线程 II 没有在进入 synchronized 块，因此不存在 lock 与 unlock 锁的问题），这说明线程 I 的语句(1)和线程 II 的语句(7)之间并不存在 happen-before 关系，这就意味着线程 II 在执行语句(7)完全有可能观测不到线程 I 在语句(1)处对 someFiled 写入的值，这就是 DCL 的问题所在。很荒谬，是吧？DCL 原本是为了逃避同步，它达到了这个目的，也正是因为如此，它最终受到惩罚，这样的程序存在严重的 bug，虽然这种 bug 被发现的概率绝对比中彩票的概率还要低得多，而且是转瞬即逝，更可怕的是，即使发生了你也不会想到是 DCL 所引起的。

前面我们说了，线程 II 在执行语句(2)时也有可能观察空值，如果是种情况，那么它需要进入同步块，并执行语句(4)。在语句(4)处线程 II 还能够读到 instance 的空值吗？不可能。这里因为这时对 instance 的写和读都是发生在同一个锁确定的同步块中，这时读到的数据是最新的数据。为也加深印象，我再用 happen-before 规则分析一遍。线程 II 在语句(3)处会执行一个 lock 操作，而线程 I 在语句(5)后会执行一个 unlock 操作，这两个操作都是针对同一个锁——Singleton.class，因此根据第 2 条 happen-before 规则，线程 I 的 unlock 操作 happen-before 线程 II 的 lock 操作，再利用单线程规则，线程 I 的语句(5) -> 线程 I 的 unlock 操作，线程 II 的 lock 操作 -> 线程 II 的语句(4)，再根据传递规则，就有线程 I 的语句(5) -> 线程 II 的语句(4)，也就是说线程 II 在执行语句(4)时能够观测到线程 I 在语句(5)时对 Singleton 的写入值。接着对返回的 instance 调用 getSomeField() 方法时，我们也能得到线程 I 的语句(1) -> 线程 II 的语句(7)（由于线程 II 有进入 synchronized 块，根据规

则 2 可得)，这表明这时 `getSomeField` 能够得到正确的值。但是仅仅是这种情况的正确性并不妨碍 DCL 的不正确性，一个程序的正确性必须在所有的情况下的行为都是正确的，而不能有时正确，有时不正确。

对 DCL 的分析也告诉我们一条经验原则：对引用（包括对象引用和数组引用）的非同步访问，即使得到该引用的最新值，却并不能保证也能得到其成员变量（对数组而言就是每个数组元素）的最新值。

解决方案

最简单而且安全的解决方法是使用 `static` 内部类的思想，它利用的思想是：一个类直到被使用时才被初始化，而类初始化的过程是非并行的，这些都有 JLS 保证。

如下述代码：

```
public class Singleton {

    private Singleton() {}

    // Lazy initialization holder class idiom for static fields
    private static class InstanceHolder {
        private static final Singleton instance = new Singleton();
    }

    public static Singleton getSingleton() {
        return InstanceHolder.instance;
    }
}
```

另外，可以将 `instance` 声明为 `volatile`，即

```
private volatile static LazySingleton instance;
```

这样我们便可以得到，线程 I 的语句(5) → 语线程 II 的句(2)，根据单线程规则，线程 I 的语句(1) → 线程 I 的语句(5)和语线程 II 的句(2) → 语线程 II 的句(7)，再根据传递规则就有线程 I 的语句(1) → 语线程 II 的句(7)，这表示线程 II 能够观察到线程 I 在语句(1)时对 `someFiled` 的写入值，程序能够得到正确的行为。

注：1、`volatile` 屏蔽指令重排序的语义在 JDK1.5 中才被完全修复，此前的 JDK 中及时将变量声明为 `volatile`，也仍然不能完全避免重排序所导致的问题（主要是 `volatile` 变量前后的代码仍然存在重排序问题），这点也是在 JDK1.5 之前的 Java 中无法安全使用 DCL 来实现单例模式的原因。

2、把 `volatile` 写和 `volatile` 读这两个操作综合起来看，在读线程 B 读一个 `volatile` 变量后，写线程 A 在写这个 `volatile` 变量之前，所有可见的共享变量的值都将立即变得对读线程 B 可见。

3、在 java5 之前对 final 字段的同步语义和其它变量没有什么区别，在 java5 中，final 变量一旦在构造函数中设置完成（前提是在构造函数中没有泄露 this 引用），其它线程必定会看到在构造函数中设置的值。而 DCL 的问题正好在于看到对象的成员变量的默认值，因此我们可以将 LazySingleton 的 someField 变量设置成 final，这样在 java5 中就能够正确运行了。



19

深入 Java 内存模型（2）



主内存与工作内存

Java 内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量主要是指共享变量，存在竞争问题的变量。Java 内存模型规定所有的变量都存储在主内存中，而每条线程还有自己的工作内存，线程的工作内存中保存了该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量（根据 Java 虚拟机规范的规定，volatile 变量依然有共享内存的拷贝，但是由于它特殊的操作顺序性规定——从工作内存中读写数据前，必须先将主内存中的数据同步到工作内存中，所有看起来如同直接在主内存中读写访问一般，因此这里的描述对于 volatile 也不例外）。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值得传递均需要通过主内存来完成。

内存间交互操作

Java 内存模型中定义了以下 8 种操作来完成主内存与工作内存之间交互的实现细节：

- lock（锁定）：作用于主内存的变量，它把一个变量标示为一条线程独占的状态。
- unlock（解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
- read（读取）：作用于主内存的变量，它把一个变量的值从主内存传输到工作内存中，以便随后的 load 动作使用。
- load（载入）：作用于工作内存的变量，它把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。
- use（使用）：作用于工作内存的变量，它把工作内存中的一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值得字节码指令时将会执行这个操作。
- assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
- store（存储）：作用于工作内存的变量，它把工作内存中的一个变量的值传递到主内存中，以便随后的 write 操作使用。
- write（写入）：作用于主内存的变量，它把 store 操作从工作内存中得到的变量值放入主内存的变量中。

Java 内存模型还规定了执行上述 8 种基本操作时必须满足如下规则：

- 不允许 read 和 load、store 和 write 操作之一单独出现，以上两个操作必须按顺序执行，但没有保证必须连续执行，也就是说，read 与 load 之间、store 与 write 之间是可插入其他指令的。
- 不允许一个线程丢弃它的最近的 assign 操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。
- 不允许一个线程无原因地（没有发生过任何 assign 操作）把数据从线程的工作内存同步回主内存中。
- 一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load 或 assign）的变量，换句话说就是对一个变量实施 use 和 store 操作之前，必须先执行过了 assign 和 load 操作。
- 一个变量在同一个时刻只允许一条线程对其执行 lock 操作，但 lock 操作可以被同一个线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁。
- 如果对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行 load 或 assign 操作初始化变量的值。
- 如果一个变量实现没有被 lock 操作锁定，则不允许对它执行 unlock 操作，也不允许去 unlock 一个被其他线程锁定的变量。
- 对一个变量执行 unlock 操作之前，必须先把此变量同步回主内存（执行 store 和 write 操作）。

volatile 型变量的特殊规则

Java 内存模型对 volatile 专门定义了一些特殊的访问规则，当一个变量被定义成 volatile 之后，他将具备两种特性：

- 保证此变量对所有线程的可见性。这里不具体解释了。需要注意，volatile 变量的写操作除了对它本身的读操作可见外，volatile 写操作之前的所有共享变量均对 volatile 读操作之后的操作可见，另外注意其适用场景。
- 禁止指令重排序优化。普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获得正确的结果，而不能保证变量赋值操作的顺序与程序中的执行顺序一致，在单线程中，我们是无法感知这一点的。
- 补充：Java 语言规范规定了 JVM 线程内部维持顺序化语义，也就是说只要程序的最终结果等同于它在严格的顺序化环境下的结果，那么指令的执行顺序就可能与代码的顺序不一致，这个过程通过叫做指令的重排序。指令重排序存在的意义在于：JVM 能够根据处理器的特性（CPU 的多级缓存系统、多核处理器等）适当的重新排序机器指令，使机器指令更符合 CPU 的执行特点，最大限度的发挥机器的性能。在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整。

final 域

final 类型的域是不能修改的，除了这一点外，在 Java 内存模型中，final 域还有着特殊的语义，final 域能确保初始化过程的安全性，从而可以不受限制地访问不可变对象，并在共享这些对象时无须同步。具体而言，就是被 final 修饰的字段在构造器中一旦被初始化完成，并且构造器没有把“this”的引用传递出去（this 引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象），那么在其他线程中就能看到 final 字段的值，而且其外、外部可见状态永远也不会改变。它所带来的安全性是最简单最纯粹的。

long 和 double 型变量的特殊规则

Java 内存模型要求 lock、unlock、read、load、assign、use、store 和 write 这 8 个操作都具有原子性，但是对于 64 位的数据类型 long 和 double，在模型中特别定义了一条宽松的规定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作划分为两次 32 位的操作来进行。这样，如果有多个线程共享一个未被声明为 volatile 的 long 或 double 类型的变量，并且同时对它们进行读取和修改操作，那么某些线程可能会读到一个既非原值，也非其他线程修改值得代表了“半个变量”的数值。不过这种读取到“半个变量”的情况非常罕见，因为 Java 内存模型虽然允许虚拟机不把 long 和 double 变量的读写实现成原子操作，但允许虚拟机选择把这些操作实现为具有原子性的操作，而且还“强烈建议”虚拟机这样实现。目前各种平台下的商用虚拟机几乎都选择把 64 位数据的读写操作作为原子操作来对待，因此在编码时，不需要将 long 和 double 变量专门声明为 volatile。



20

并发新特性—Executor 框架与线程池



Executor 框架简介

在 Java 5 之后，并发编程引入了一堆新的启动、调度和管理线程的 API。Executor 框架便是 Java 5 中引入的，其内部使用了线程池机制，它在 `java.util.concurrent` 包下，通过该框架来控制线程的启动、执行和关闭，可以简化并发编程的操作。因此，在 Java 5 之后，通过 Executor 来启动线程比使用 Thread 的 `start` 方法更好，除了更易管理，效率更好（用线程池实现，节约开销）外，还有关键的一点：有助于避免 this 逃逸问题——如果我们在构造器中启动一个线程，因为另一个任务可能会在构造器结束之前开始执行，此时可能会访问到初始化了一半的对象用 Executor 在构造器中。

Executor 框架包括：线程池，Executor，Executors，ExecutorService，CompletionService，Future，Callable 等。

Executor 接口中定义了一个方法 `execute (Runnable command)`，该方法接收一个 Runnable 实例，它用来执行一个任务，任务即一个实现了 Runnable 接口的类。ExecutorService 接口继承自 Executor 接口，它提供了更丰富的实现多线程的方法，比如，ExecutorService 提供了关闭自己的方法，以及可为跟踪一个或多个异步任务执行状况而生成 Future 的方法。可以调用 ExecutorService 的 `shutdown ()` 方法来平滑地关闭 ExecutorService，调用该方法后，将导致 ExecutorService 停止接受任何新的任务且等待已经提交的任务执行完成(已经提交的任务会分两类：一类是已经在执行的，另一类是还没有开始执行的)，当所有已经提交的任务执行完毕后将会关闭 ExecutorService。因此我们一般用该接口来实现和管理多线程。

ExecutorService 的生命周期包括三种状态：运行、关闭、终止。创建后便进入运行状态，当调用了 `shutdown ()` 方法时，便进入关闭状态，此时意味着 ExecutorService 不再接受新的任务，但它还在执行已经提交的任务，当素有已经提交的任务执行完后，便到达终止状态。如果不调用 `shutdown ()` 方法，ExecutorService 会一直处在运行状态，不断接收新的任务，执行新的任务，服务器端一般不需要关闭它，保持一直运行即可。

Executors 提供了一系列工厂方法用于创先线程池，返回的线程池都实现了 ExecutorService 接口。

```
public static ExecutorService newFixedThreadPool(int nThreads)
```

创建固定数目线程的线程池。

```
public static ExecutorService newCachedThreadPool()
```

创建一个可缓存的线程池，调用 `execute` 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

```
public static ExecutorService newSingleThreadExecutor()
```

创建一个单线程化的 Executor。

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
```

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

这四种方法都是用的 Executors 中的 ThreadFactory 建立的线程，下面就以上四个方法做个比较：

newCachedThreadPool()

- 缓存型池子，先查看池中有没有以前建立的线程，如果有，就 reuse 如果没有，就建一个新的线程加入池中
- 缓存型池子通常用于执行一些生存期很短的异步型任务 因此在一些面向连接的 daemon 型 SERVER 中用得不多。但对于生存期短的异步任务，它是 Executor 的首选。
- 能 reuse 的线程，必须是 timeout IDLE 内的池中线程，缺省 timeout 是 60s,超过这个 IDLE 时长，线程实例将被终止及移出池。

注意，放入 CachedThreadPool 的线程不必担心其结束，超过 TIMEOUT 不活动，其会自动被终止。

newFixedThreadPool(int)

- newFixedThreadPool 与 cacheThreadPool 差不多，也是能 reuse 就用，但不能随时建新的线程。
- 其独特之处:任意时间点，最多只能有固定数目的活动线程存在，此时如果有新的线程要建立，只能放在另外的队列中等待，直到当前的线程中某个线程终止直接被移出池子。
- 和 cacheThreadPool 不同，FixedThreadPool 没有 IDLE 机制（可能也有，但既然文档没提，肯定非常长，类似依赖上层的 TCP 或 UDP IDLE 机制之类的），所以 FixedThreadPool 多数针对一些很稳定很固定的正规并发线程，多用于服务器。
- 从方法的源代码看，cache池和fixed 池调用的是同一个底层 池，只不过参数不同：
 - fixed 池线程数固定，并且是0秒IDLE（无IDLE）。
 - cache 池线程数支持 0-Integer.MAX_VALUE(显然完全没考虑主机的资源承受能力)，60 秒 IDLE。

newScheduledThreadPool(int)

- 调度型线程池
- 这个池子里的线程可以按 schedule 依次 delay 执行，或周期执行

SingleThreadExecutor()

- 单例线程，任意时间池中只能有一个线程
- 用的是和 cache 池和 fixed 池相同的底层池，但线程数目是 1-1,0 秒 IDLE（无 IDLE）

一般来说，CachedThreadPool 在程序执行过程中通常会创建与所需数量相同的线程，然后在它回收旧线程时停止创建新线程，因此它是合理的 Executor 的首选，只有当这种方式会引发问题时（比如需要大量长时间面向连接的线程时），才需要考虑用 FixedThreadPool。（该段话摘自《Thinking in Java》第四版）

Executor 执行 Runnable 任务

通过 Executors 的以上四个静态工厂方法获得 ExecutorService 实例，而后调用该实例的 execute（Runnable command）方法即可。一旦 Runnable 任务传递到 execute（）方法，该方法便会自动在一个线程上执行。下面是 Executor 执行 Runnable 任务的示例代码：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestCachedThreadPool{
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        // ExecutorService executorService = Executors.newFixedThreadPool(5);
        // ExecutorService executorService = Executors.newSingleThreadExecutor();
        for (int i = 0; i < 5; i++){
            executorService.execute(new TestRunnable());
            System.out.println("***** a" + i + " *****");
        }
        executorService.shutdown();
    }
}

class TestRunnable implements Runnable{
    public void run(){
        System.out.println(Thread.currentThread().getName() + "线程被调用了。");
    }
}
```

执行后的结果如下：

```

C:\WINDOWS\system32\cmd.exe

F:\>java TestCachedThreadPool
***** a0 *****
pool-1-thread-1线程被调用了。
***** a1 *****
***** a2 *****
pool-1-thread-3线程被调用了。
pool-1-thread-2线程被调用了。
***** a3 *****
pool-1-thread-1线程被调用了。
***** a4 *****
pool-1-thread-2线程被调用了。

F:\>

```

从结果中可以看出，pool-1-thread-1 和 pool-1-thread-2 均被调用了两次，这是随机的，execute 会首先在线程池中选择一个已有空闲线程来执行任务，如果线程池中没有空闲线程，它便会创建一个新的线程来执行任务。

Executor 执行 Callable 任务

在 Java 5 之后，任务分两类：一类是实现了 Runnable 接口的类，一类是实现了 Callable 接口的类。两者都可以被 ExecutorService 执行，但是 Runnable 任务没有返回值，而 Callable 任务有返回值。并且 Callable 的 call() 方法只能通过 ExecutorService 的 submit(Callable task) 方法来执行，并且返回一个 Future，是表示任务等待完成的 Future。

Callable 接口类似于 Runnable，两者都是为那些其实例可能被另一个线程执行的类设计的。但是 Runnable 不会返回结果，并且无法抛出经过检查的异常而 Callable 又返回结果，而且当获取返回结果时可能会抛出异常。Callable 中的 call() 方法类似 Runnable 的 run() 方法，区别同样是有返回值，后者没有。

当将一个 Callable 的对象传递给 ExecutorService 的 submit 方法，则该 call 方法自动在一个线程上执行，并且会返回执行结果 Future 对象。同样，将 Runnable 的对象传递给 ExecutorService 的 submit 方法，则该 run 方法自动在一个线程上执行，并且会返回执行结果 Future 对象，但是在该 Future 对象上调用 get 方法，将返回 null。

下面给出一个 Executor 执行 Callable 任务的示例代码：

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class CallableDemo{
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<String>> resultList = new ArrayList<Future<String>>();

        //创建10个任务并执行
        for (int i = 0; i < 10; i++){
            //使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中
            Future<String> future = executorService.submit(new TaskWithResult(i));
            //将任务执行结果存储到List中
            resultList.add(future);
        }

        //遍历任务的结果
        for (Future<String> fs : resultList){
            try{
                while(!fs.isDone()); //Future返回如果没有完成，则一直循环等待，直到Future返回完成
                System.out.println(fs.get()); //打印各个线程（任务）执行的结果
            }catch(InterruptedException e){
                e.printStackTrace();
            }catch(ExecutionException e){
                e.printStackTrace();
            }finally{
                //启动一次顺序关闭，执行以前提交的任务，但不接受新任务
                executorService.shutdown();
            }
        }
    }
}

class TaskWithResult implements Callable<String>{
    private int id;

    public TaskWithResult(int id){
        this.id = id;
    }

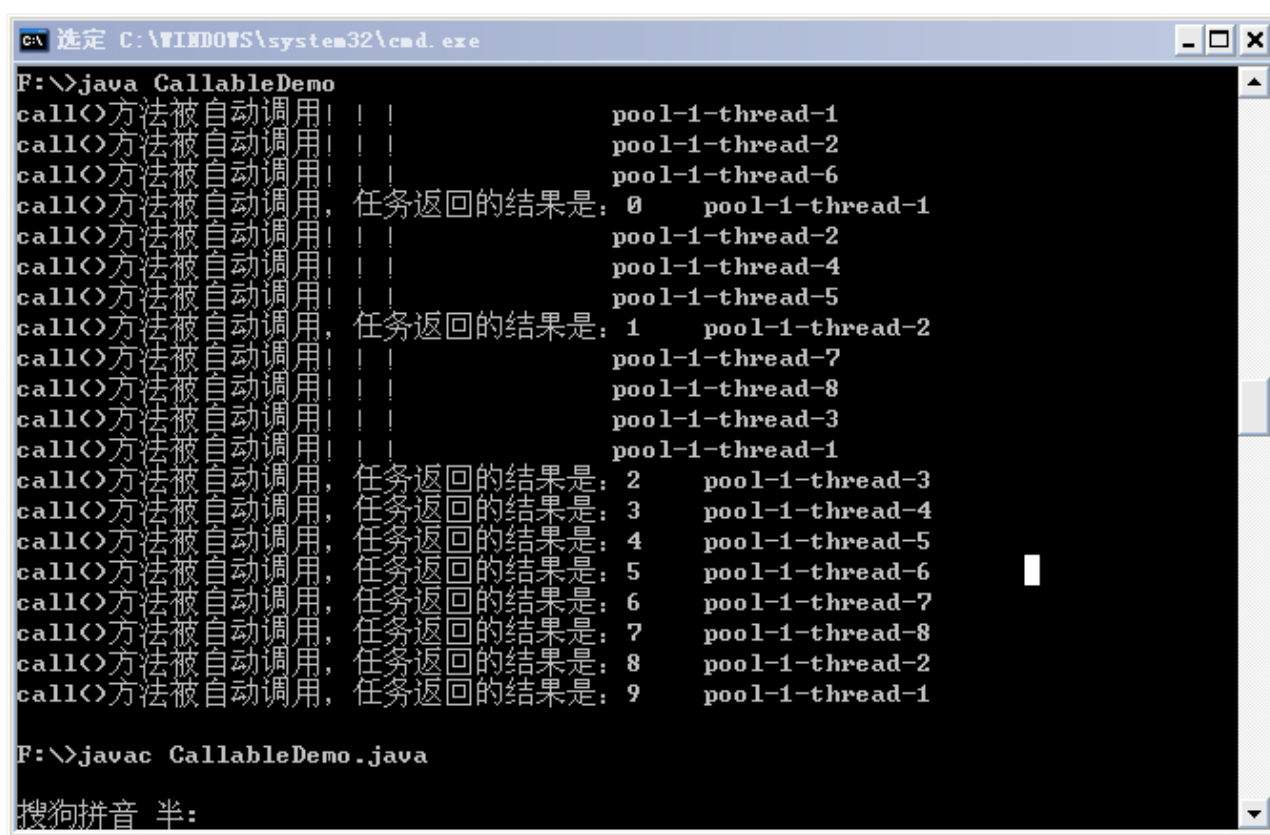
    /**
```

```

* 任务的具体过程，一旦任务传给ExecutorService的submit方法，
* 则该方法自动在一个线程上执行
*/
public String call() throws Exception {
    System.out.println("call()方法被自动调用!!! " + Thread.currentThread().getName());
    //该返回结果将被Future的get方法得到
    return "call()方法被自动调用，任务返回的结果是：" + id + " " + Thread.currentThread().getName();
}
}

```

执行结果如下：



```

C:\> 选定 C:\WINDOWS\system32\cmd.exe
F:\>java CallableDemo
call()方法被自动调用!!!          pool-1-thread-1
call()方法被自动调用!!!          pool-1-thread-2
call()方法被自动调用!!!          pool-1-thread-6
call()方法被自动调用，任务返回的结果是: 0    pool-1-thread-1
call()方法被自动调用!!!          pool-1-thread-2
call()方法被自动调用!!!          pool-1-thread-4
call()方法被自动调用!!!          pool-1-thread-5
call()方法被自动调用，任务返回的结果是: 1    pool-1-thread-2
call()方法被自动调用!!!          pool-1-thread-7
call()方法被自动调用!!!          pool-1-thread-8
call()方法被自动调用!!!          pool-1-thread-3
call()方法被自动调用!!!          pool-1-thread-1
call()方法被自动调用，任务返回的结果是: 2    pool-1-thread-3
call()方法被自动调用，任务返回的结果是: 3    pool-1-thread-4
call()方法被自动调用，任务返回的结果是: 4    pool-1-thread-5
call()方法被自动调用，任务返回的结果是: 5    pool-1-thread-6
call()方法被自动调用，任务返回的结果是: 6    pool-1-thread-7
call()方法被自动调用，任务返回的结果是: 7    pool-1-thread-8
call()方法被自动调用，任务返回的结果是: 8    pool-1-thread-2
call()方法被自动调用，任务返回的结果是: 9    pool-1-thread-1

F:\>javac CallableDemo.java
搜狗拼音 半:

```

从结果中同样可以看出，submit 也是首先选择空闲线程来执行任务，如果没有，才会创建新的线程来执行任务。另外，需要注意：如果 Future 的返回尚未完成，则 get () 方法会阻塞等待，直到 Future 完成返回，可以通过调用 isDone () 方法判断 Future 是否完成了返回。

自定义线程池

自定义线程池，可以用 ThreadPoolExecutor 类创建，它有多个构造方法来创建线程池，用该类很容易实现自定义的线程池，这里先贴上示例程序：

```

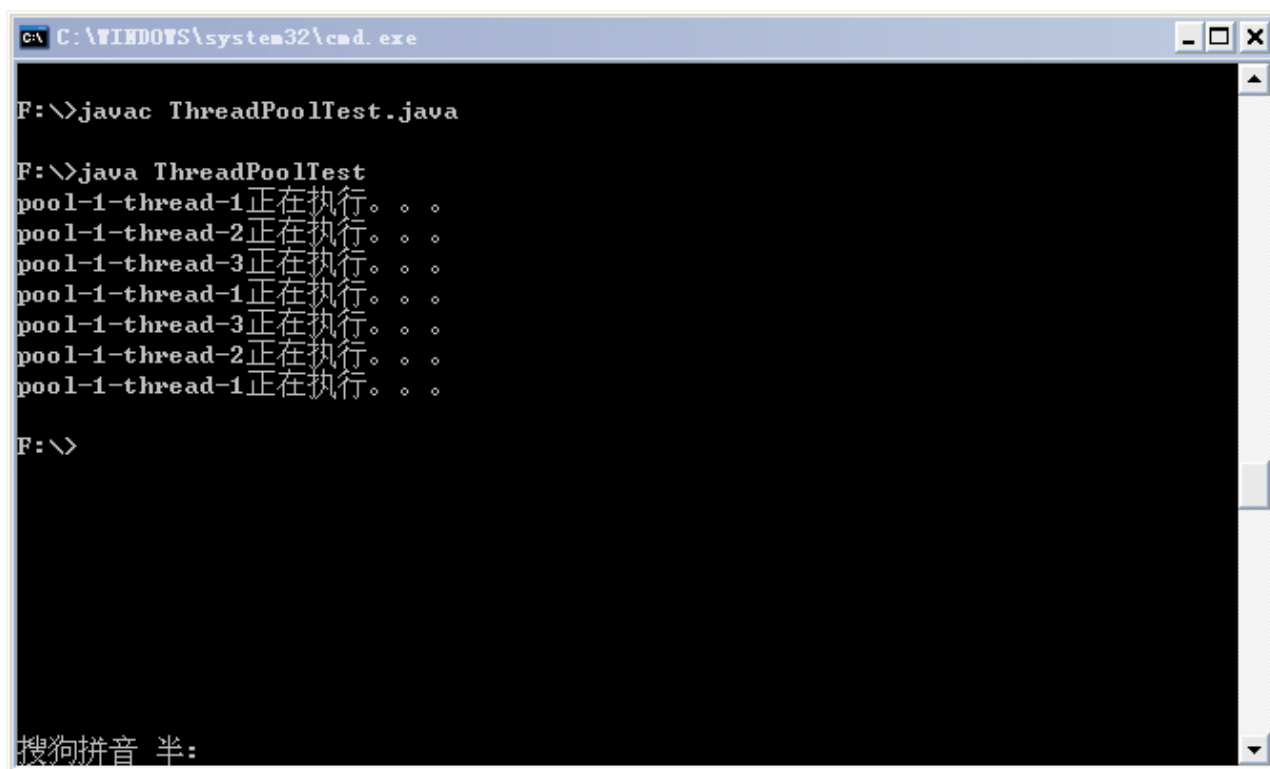
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolTest{
    public static void main(String[] args){
        //创建等待队列
        BlockingQueue<Runnable> bqueue = new ArrayBlockingQueue<Runnable>(20);
        //创建线程池，池中保存的线程数为3，允许的最大线程数为5
        ThreadPoolExecutor pool = new ThreadPoolExecutor(3,5,50,TimeUnit.MILLISECONDS,bqueue);
        //创建七个任务
        Runnable t1 = new MyThread();
        Runnable t2 = new MyThread();
        Runnable t3 = new MyThread();
        Runnable t4 = new MyThread();
        Runnable t5 = new MyThread();
        Runnable t6 = new MyThread();
        Runnable t7 = new MyThread();
        //每个任务会在一个线程上执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        pool.execute(t6);
        pool.execute(t7);
        //关闭线程池
        pool.shutdown();
    }
}

class MyThread implements Runnable{
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName() + "正在执行。。。");
        try{
            Thread.sleep(100);
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}

```

运行结果如下：



```

C:\WINDOWS\system32\cmd.exe

F:\>javac ThreadPoolTest.java

F:\>java ThreadPoolTest
pool-1-thread-1正在执行。。。
pool-1-thread-2正在执行。。。
pool-1-thread-3正在执行。。。
pool-1-thread-1正在执行。。。
pool-1-thread-3正在执行。。。
pool-1-thread-2正在执行。。。
pool-1-thread-1正在执行。。。

F:\>

```

从结果中可以看出，七个任务是在线程池的三个线程上执行的。这里简要说明下用到的 `ThreadPoolExecutor` 类的构造方法中各个参数的含义。

```
public ThreadPoolExecutor (int corePoolSize, int maximumPoolSize, long    keepAliveTime, TimeUnit unit,BlockingQueue
```

- `corePoolSize`：线程池中所保存的核心线程数，包括空闲线程。
- `maximumPoolSize`：池中允许的最大线程数。
- `keepAliveTime`：线程池中的空闲线程所能持续的最长时间。
- `unit`：持续时间的单位。
- `workQueue`：任务执行前保存任务的队列，仅保存由 `execute` 方法提交的 `Runnable` 任务。

根据 `ThreadPoolExecutor` 源码前面大段的注释，我们可以看出，当试图通过 `execute` 方法将一个 `Runnable` 任务添加到线程池中时，按照如下顺序来处理：

1. 如果线程池中的线程数量少于 `corePoolSize`，即使线程池中有空闲线程，也会创建一个新的线程来执行新添加的任务；
2. 如果线程池中的线程数量大于等于 `corePoolSize`，但缓冲队列 `workQueue` 未滿，则将新添加的任务放到 `workQueue` 中，按照 FIFO 的原则依次等待执行（线程池中有线程空闲出来后依次将缓冲队列中的任务交付给空闲的线程执行）；

3. 如果线程池中的线程数量大于等于 `corePoolSize`，且缓冲队列 `workQueue` 已满，但线程池中的线程数量小于 `maximumPoolSize`，则会创建新的线程来处理被添加的任务；
4. 如果线程池中的线程数量等于了 `maximumPoolSize`，有 4 种才处理方式（该构造方法调用了含有 5 个参数的构造方法，并将最后一个构造方法为 `RejectedExecutionHandler` 类型，它在处理线程溢出时有 4 种方式，这里不再细说，要了解的，自己可以阅读下源码）。

总结起来，也即是说，当有新的任务要处理时，先看线程池中的线程数量是否大于 `corePoolSize`，再看缓冲队列 `workQueue` 是否满，最后看线程池中的线程数量是否大于 `maximumPoolSize`。

另外，当线程池中的线程数量大于 `corePoolSize` 时，如果里面有线程的空闲时间超过了 `keepAliveTime`，就将其移除线程池，这样，可以动态地调整线程池中线程的数量。

我们大致来看下 Executors 的源码，`newCachedThreadPool` 的不带 `RejectedExecutionHandler` 参数（即第五个参数，线程数量超过 `maximumPoolSize` 时，指定处理方式）的构造方法如下：

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
        60L, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>());
}
```

它将 `corePoolSize` 设定为 0，而将 `maximumPoolSize` 设定为了 `Integer` 的最大值，线程空闲超过 60 秒，将会从线程池中移除。由于核心线程数为 0，因此每次添加任务，都会先从线程池中找空闲线程，如果没有就会创建一个线程（`SynchronousQueue` 决定的，后面会说）来执行新的任务，并将该线程加入到线程池中，而最大允许的线程数为 `Integer` 的最大值，因此这个线程池理论上可以不断扩大。

再来看 `newFixedThreadPool` 的不带 `RejectedExecutionHandler` 参数的构造方法，如下：

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());
}
```

它将 `corePoolSize` 和 `maximumPoolSize` 都设定为了 `nThreads`，这样便实现了线程池的大小的固定，不会动态地扩大，另外，`keepAliveTime` 设定为了 0，也就是说线程只要空闲下来，就会被移除线程池，敢于 `LinkedBlockingQueue` 下面会说。

几种排队的策略

- 直接提交。缓冲队列采用 `SynchronousQueue`，它将任务直接交给线程处理而不保持它们。如果不存在可用于立即运行任务的线程（即线程池中的线程都在工作），则试图把任务加入缓冲队列将会失败，因此会构造一个新的线程来处理新添加的任务，并将其加入到线程池中。直接提交通常要求无界 `maximumPoolSize`（`Integer.MAX_VALUE`）以避免拒绝新提交的任务。`newCachedThreadPool` 采用的便是这种策略。
- 无界队列。使用无界队列（典型的便是采用预定义容量的 `LinkedBlockingQueue`，理论上是该缓冲队列可以对无限多的任务排队）将导致在所有 `corePoolSize` 线程都工作的情况下将新任务加入到缓冲队列中。这样，创建的线程就不会超过 `corePoolSize`，也因此，`maximumPoolSize` 的值也就无效了。当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列。`newFixedThreadPool` 采用的便是这种策略。
- 有界队列。当使用有限的 `maximumPoolSize` 时，有界队列（一般缓冲队列使用 `ArrayBlockingQueue`，并制定队列的最大长度）有助于防止资源耗尽，但是可能较难调整和控制，队列大小和最大池大小需要相互折衷，需要设定合理的参数。



21



并发新特性—Lock 锁与条件变量



简单使用 Lock 锁

Java 5 中引入了新的锁机制——`java.util.concurrent.locks` 中的显式的互斥锁：Lock 接口，它提供了比 `synchronized` 更加广泛的锁定操作。Lock 接口有 3 个实现它的类：`ReentrantLock`、`ReentrantReadWriteLock`、`ReadWriteLock` 和 `ReentrantReadWriteLock.WriteLock`，即重入锁、读锁和写锁。lock 必须被显式地创建、锁定和释放，为了可以使用更多的功能，一般用 `ReentrantLock` 为其实例化。为了保证锁最终一定会被释放（可能会有异常发生），要把互斥区放在 `try` 语句块内，并在 `finally` 语句块中释放锁，尤其当有 `return` 语句时，`return` 语句必须放在 `try` 语句中，以确保 `unlock()` 不会过早发生，从而将数据暴露给第二个任务。因此，采用 lock 加锁和释放锁的一般形式如下：

```
Lock lock = new ReentrantLock();//默认使用非公平锁，如果要使用公平锁，需要传入参数true
.....
lock.lock();
try {
    //更新对象的状态
    //捕获异常，必要时恢复到原来的不变约束
    //如果有return语句，放在这里
finally {
    lock.unlock();    //锁必须在finally块中释放
```

ReetrantLock 与 synchronized 比较

性能比较

在 JDK1.5 中，`synchronized` 是性能低效的。因为这是一个重量级操作，它对性能最大的影响是阻塞的是实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性带来了很大的压力。相比之下使用 Java 提供的 Lock 对象，性能更高一些。Brian Goetz 对这两种锁在 JDK1.5、单核处理器及双 Xeon 处理器环境下做了一组吞吐量对比的实验，发现多线程环境下，`synchronized` 的吞吐量下降的非常严重，而 `ReentrantLock` 则能基本保持在同一个比较稳定的水平上。但与其说 `ReentrantLock` 性能好，倒不如说 `synchronized` 还有非常大的优化余地，于是到了 JDK1.6，发生了变化，对 `synchronize` 加入了很多优化措施，有自适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在 JDK1.6 上 `synchronize` 的性能并不比 Lock 差。官方也表示，他们也更支持 `synchronize`，在未来的版本中还有优化余地，所以还是提倡在 `synchronized` 能实现需求的情况下，优先考虑使用 `synchronized` 来进行同步。

下面浅析以下两种锁机制的底层的实现策略。

互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题，因而这种同步又称为阻塞同步，它属于一种悲观的并发策略，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在 CPU 转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起 CPU 频繁的上下文切换导致效率很低。synchronized 采用的便是这种并发策略。

随着指令集的发展，我们有了另一种选择：基于冲突检测的乐观并发策略，通俗地讲就是先进性操作，如果没有其他线程争用共享数据，那操作就成功了，如果共享数据被争用，产生了冲突，那就再进行其他的补偿措施（最常见的补偿措施就是不断地重拾，直到试成功为止），这种乐观的并发策略的许多实现都不需要把线程挂起，因此这种同步被称为非阻塞同步。ReentrantLock 采用的便是这种并发策略。

在乐观的并发策略中，需要操作和冲突检测这两个步骤具备原子性，它靠硬件指令来保证，这里用的是 CAS 操作（Compare and Swap）。JDK1.5 之后，Java 程序才可以使用 CAS 操作。我们可以进一步研究 ReentrantLock 的源代码，会发现其中比较重要的获得锁的一个方法是 compareAndSetState，这里其实就是调用的 CPU 提供的特殊指令。现代的 CPU 提供了指令，可以自动更新共享数据，而且能够检测到其他线程的干扰，而 compareAndSet() 就用这些代替了锁定。这个算法称作非阻塞算法，意思是一个线程的失败或者挂起不应该影响其他线程的失败或挂起。

Java 5 中引入了注入 AtomicInteger、AtomicLong、AtomicReference 等特殊的原子性变量类，它们提供的如：compareAndSet()、incrementAndSet()和getAndIncrement()等方法都使用了 CAS 操作。因此，它们都是由硬件指令来保证的原子方法。

用途比较

基本语法上，ReentrantLock 与 synchronized 很相似，它们都具备一样的线程重入特性，只是代码写法上有点区别而已，一个表现为 API 层面的互斥锁（Lock），一个表现为原生语法层面的互斥锁（synchronized）。ReentrantLock 相对 synchronized 而言还是增加了一些高级功能，主要有以下三项：

- 等待可中断：当持有锁的线程长期不释放锁时，正在等待的线程可以选择放弃等待，改为处理其他事情，它对处理执行时间非常上的同步块很有帮助。而在等待由 synchronized 产生的互斥锁时，会一直阻塞，是不能被中断的。
- 可实现公平锁：多个线程在等待同一个锁时，必须按照申请锁的时间顺序排队等待，而非公平锁则不保证这点，在锁释放时，任何一个等待锁的线程都有机会获得锁。synchronized 中的锁时非公平锁，ReentrantLock 默认情况下也是非公平锁，但可以通过构造方法 ReentrantLock（ture）来要求使用公平锁。
- 锁可以绑定多个条件：ReentrantLock 对象可以同时绑定多个 Condition 对象（名曰：条件变量或条件队列），而在 synchronized 中，锁对象的 wait()和 notify()或 notifyAll()方法可以实现一个隐含条件，但如果要和多于一个的条件关联的时候，就不得不额外地添加一个锁，而 ReentrantLock 则无需这么做，只需要

多次调用 `newCondition()` 方法即可。而且我们还可以通过绑定 `Condition` 对象来判断当前线程通知的是哪些线程（即与 `Condition` 对象绑定在一起的其他线程）。

可中断锁

`ReentrantLock` 有两种锁：忽略中断锁和响应中断锁。忽略中断锁与 `synchronized` 实现的互斥锁一样，不能响应中断，而响应中断锁可以响应中断。

如果某一线程 A 正在执行锁中的代码，另一线程 B 正在等待获取该锁，可能由于等待时间过长，线程 B 不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，如果此时 `ReentrantLock` 提供的是忽略中断锁，则它不会去理会该中断，而是让线程 B 继续等待，而如果此时 `ReentrantLock` 提供的是响应中断锁，那么它便会处理中断，让线程 B 放弃等待，转而去处理其他事情。

获得响应中断锁的一般形式如下：

```
ReentrantLock lock = new ReentrantLock();
.....
lock.lockInterruptibly();//获取响应中断锁
try {
    //更新对象的状态
    //捕获异常，必要时恢复到原来的不变约束
    //如果有return语句，放在这里
}finally{
    lock.unlock();    //锁必须在finally块中释放
}
```

这里有一个不错的分析中断的示例代码（摘自网上）。

当用 `synchronized` 中断对互斥锁的等待时，并不起作用，该线程依然会一直等待，如下面的实例：

```
public class Buffer {

    private Object lock;

    public Buffer() {
        lock = this;
    }

    public void write() {
        synchronized (lock) {
            long startTime = System.currentTimeMillis();
            System.out.println("开始往这个buff写入数据...");
```

```

    for (;;) // 模拟要处理很长时间
    {
        if (System.currentTimeMillis()
            - startTime > Integer.MAX_VALUE) {
            break;
        }
    }
    System.out.println("终于写完了");
}

public void read() {
    synchronized (lock) {
        System.out.println("从这个buff读数据");
    }
}

public static void main(String[] args) {
    Buffer buff = new Buffer();

    final Writer writer = new Writer(buff);
    final Reader reader = new Reader(buff);

    writer.start();
    reader.start();

    new Thread(new Runnable() {

        @Override
        public void run() {
            long start = System.currentTimeMillis();
            for (;;) {
                //等5秒钟去中断读
                if (System.currentTimeMillis()
                    - start > 5000) {
                    System.out.println("不等了，尝试中断");
                    reader.interrupt(); //尝试中断读线程
                    break;
                }
            }
        }
    }).start();
    // 我们期待“读”这个线程能退出等待锁，可是事与愿违，一旦读这个线程发现自己得不到锁，

```



```

        // 就一直开始等待了，就算它等死，也得不到锁，因为写线程要21亿秒才能完成 T_T，即使我们中断它，
        // 它都不来响应下，看来真的要等死了。这个时候，ReentrantLock给了一种机制让我们来响应中断，
        // 让“读”能伸能屈，勇敢放弃对这个锁的等待。我们来改写Buffer这个类，就叫BufferInterruptibly吧，可中断缓存。
    }
}

class Writer extends Thread {

    private Buffer buff;

    public Writer(Buffer buff) {
        this.buff = buff;
    }

    @Override
    public void run() {
        buff.write();
    }
}

class Reader extends Thread {

    private Buffer buff;

    public Reader(Buffer buff) {
        this.buff = buff;
    }

    @Override
    public void run() {

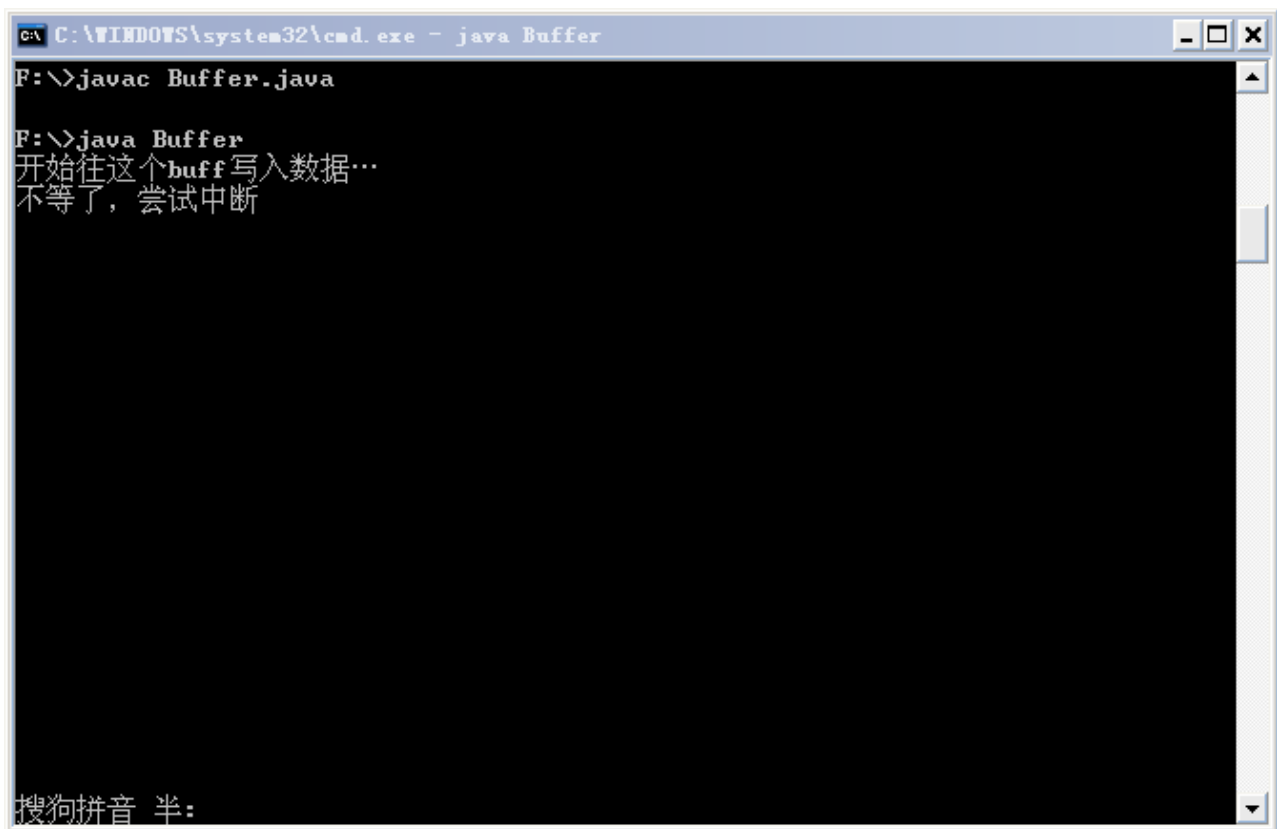
        buff.read();//这里估计会一直阻塞

        System.out.println("读结束");

    }
}

```

执行结果如下：



```

C:\WINDOWS\system32\cmd.exe - java Buffer
F:\>javac Buffer.java
F:\>java Buffer
开始往这个buff写入数据...
不等了，尝试中断
搜狗拼音 半:

```

我们等待了很久，后面依然没有输出，说明读线程对互斥锁的等待并没有被中断，也就是该户吃锁没有响应对读线程的中断。

我们再将上面代码中 `synchronized` 的互斥锁改为 `ReentrantLock` 的响应中断锁，即改为如下代码：

```

import java.util.concurrent.locks.ReentrantLock;

public class BufferInterruptibly {

    private ReentrantLock lock = new ReentrantLock();

    public void write() {
        lock.lock();
        try {
            long startTime = System.currentTimeMillis();
            System.out.println("开始往这个buff写入数据...");
            for (;;) // 模拟要处理很长时间
            {
                if (System.currentTimeMillis()
                    - startTime > Integer.MAX_VALUE) {
                    break;
                }
            }
        }
    }
}

```

```

        System.out.println("终于写完了");
    } finally {
        lock.unlock();
    }
}

public void read() throws InterruptedException {
    lock.lockInterruptibly();// 注意这里，可以响应中断
    try {
        System.out.println("从这个buff读数据");
    } finally {
        lock.unlock();
    }
}

public static void main(String args[]) {
    BufferInterruptibly buff = new BufferInterruptibly();

    final Writer2 writer = new Writer2(buff);
    final Reader2 reader = new Reader2(buff);

    writer.start();
    reader.start();

    new Thread(new Runnable() {

        @Override
        public void run() {
            long start = System.currentTimeMillis();
            for (;;) {
                if (System.currentTimeMillis()
                    - start > 5000) {
                    System.out.println("不等了，尝试中断");
                    reader.interrupt(); //此处中断读操作
                    break;
                }
            }
        }
    }).start();
}

}

class Reader2 extends Thread {

```

```
private BufferInterruptibly buff;

public Reader2(BufferInterruptibly buff) {
    this.buff = buff;
}

@Override
public void run() {

    try {
        buff.read();//可以收到中断的异常，从而有效退出
    } catch (InterruptedException e) {
        System.out.println("我不读了");
    }

    System.out.println("读结束");

}
}

class Writer2 extends Thread {

    private BufferInterruptibly buff;

    public Writer2(BufferInterruptibly buff) {
        this.buff = buff;
    }

    @Override
    public void run() {
        buff.write();
    }

}
```

执行结果如下：

```

选定 C:\WINDOWS\system32\cmd.exe - java BufferInterruptibly

F:\>javac BufferInterruptibly.java

F:\>java BufferInterruptibly
开始往这个buff写入数据...
不等了, 尝试中断
我不读了
读结束

搜狗拼音 半:

```

从结果中可以看出，尝试中断后输出了 catch 语句块中的内容，也输出了后面的“读结束”，说明线程对互斥锁的等待被中断了，也就是该互斥锁响应了对读线程的中断。

条件变量实现线程间协作

在生产者——消费者模型一文中，我们用 synchronized 实现互斥，并配合使用 Object 对象的 wait () 和 notify()或 notifyAll()方法来实现线程间协作。Java 5 之后，我们可以用 Reentrantlock 锁配合 Condition 对象上的 await()和 signal()或 signalAll()方法来实现线程间协作。在 ReentrantLock 对象上 newCondition()可以得到一个 Condition 对象，可以通过在 Condition 上调用 await()方法来挂起一个任务（线程），通过在 Condition 上调用 signal()来通知任务，从而唤醒一个任务，或者调用 signalAll()来唤醒所有在这个 Condition 上被其自身挂起的任务。另外，如果使用了公平锁，signalAll()的与 Condition 关联的所有任务将以 FIFO 队列的形式获取锁，如果没有使用公平锁，则获取锁的任务是随机的，这样我们便可以更好地控制处在 await 状态的任务获取锁的顺序。与 notifyAll()相比，signalAll()是更安全的方式。另外，它可以指定唤醒与自身 Condition 对象绑定在一起的任务。

下面将生产者——消费者模型一文中的代码改为用条件变量实现，如下：

```

import java.util.concurrent.*;
import java.util.concurrent.locks.*;

```

```

class Info{ // 定义信息类
    private String name = "name";//定义name属性，为了与下面set的name属性区别开
    private String content = "content" ;// 定义content属性，为了与下面set的content属性区别开
    private boolean flag = true ; // 设置标志位,初始时先生产
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition(); //产生一个Condition对象
    public void set(String name,String content){
        lock.lock();
        try{
            while(!flag){
                condition.await() ;
            }
            this.setName(name) ; // 设置名称
            Thread.sleep(300) ;
            this.setContent(content) ; // 设置内容
            flag = false ;// 改变标志位，表示可以取走
            condition.signal();
        }catch(InterruptedException e){
            e.printStackTrace() ;
        }finally{
            lock.unlock();
        }
    }

    public void get(){
        lock.lock();
        try{
            while(flag){
                condition.await() ;
            }
            Thread.sleep(300) ;
            System.out.println(this.getName() +
                " --> " + this.getContent()) ;
            flag = true ; // 改变标志位，表示可以生产
            condition.signal();
        }catch(InterruptedException e){
            e.printStackTrace() ;
        }finally{
            lock.unlock();
        }
    }

    public void setName(String name){
        this.name = name ;
    }
}

```

```

public void setContent(String content){
    this.content = content ;
}
public String getName(){
    return this.name ;
}
public String getContent(){
    return this.content ;
}
}

class Producer implements Runnable{ // 通过Runnable实现多线程
    private Info info = null ;    // 保存Info引用
    public Producer(Info info){
        this.info = info ;
    }
    public void run(){
        boolean flag = true ; // 定义标记位
        for(int i=0;i<10;i++){
            if(flag){
                this.info.set("姓名--1","内容--1") ; // 设置名称
                flag = false ;
            }else{
                this.info.set("姓名--2","内容--2") ; // 设置名称
                flag = true ;
            }
        }
    }
}

class Consumer implements Runnable{
    private Info info = null ;
    public Consumer(Info info){
        this.info = info ;
    }
    public void run(){
        for(int i=0;i<10;i++){
            this.info.get() ;
        }
    }
}

public class ThreadCaseDemo{
    public static void main(String args[]){
        Info info = new Info(); // 实例化Info对象
        Producer pro = new Producer(info) ; // 生产者
        Consumer con = new Consumer(info) ; // 消费者
        new Thread(pro).start() ;
    }
}

```

```
//启动了生产者线程后，再启动消费者线程
try{
    Thread.sleep(500);
}catch(InterruptedExceptio e){
    e.printStackTrace();
}

new Thread(con).start();
}
}
```

执行后，同样可以得到如下的结果：

```
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2
姓名--1 --> 内容--1
姓名--2 --> 内容--2
```

从以上并不能看出用条件变量的 `await()`、`signal()`、`signalAll()` 方法比用 `Object` 对象的 `wait()`、`notify()`、`notifyAll()` 方法实现线程间协作有多少优点，但它在处理更复杂的多线程问题时，会有明显的优势。所以，`Lock` 和 `Condition` 对象只有在更加困难的多线程问题中才是必须的。

读写锁

另外，`synchronized` 获取的互斥锁不仅互斥读写操作、写写操作，还互斥读读操作，而读读操作时不会带来数据竞争的，因此对读读操作也互斥的话，会降低性能。Java 5 中提供了读写锁，它将读锁和写锁分离，使得读操作不互斥，获取读锁和写锁的一般形式如下：

```
ReadWriteLock rwl = new ReentrantReadWriteLock();
rwl.writeLock().lock() //获取写锁
rwl.readLock().lock() //获取读锁
```

用读锁来锁定读操作，用写锁来锁定写操作，这样写操作和写操作之间会互斥，读操作和写操作之间会互斥，但读操作和读操作就不会互斥。

《Java 并发编程实践》一书给出了使用 `ReentrantLock` 的最佳时机：

当你需要以下高级特性时，才应该使用：可定时的、可轮询的与可中断的锁获取操作，公平队列，或者非块结构的锁。否则，请使用 `synchronized`。



T



22



并发新特性—阻塞队列与阻塞栈



阻塞队列

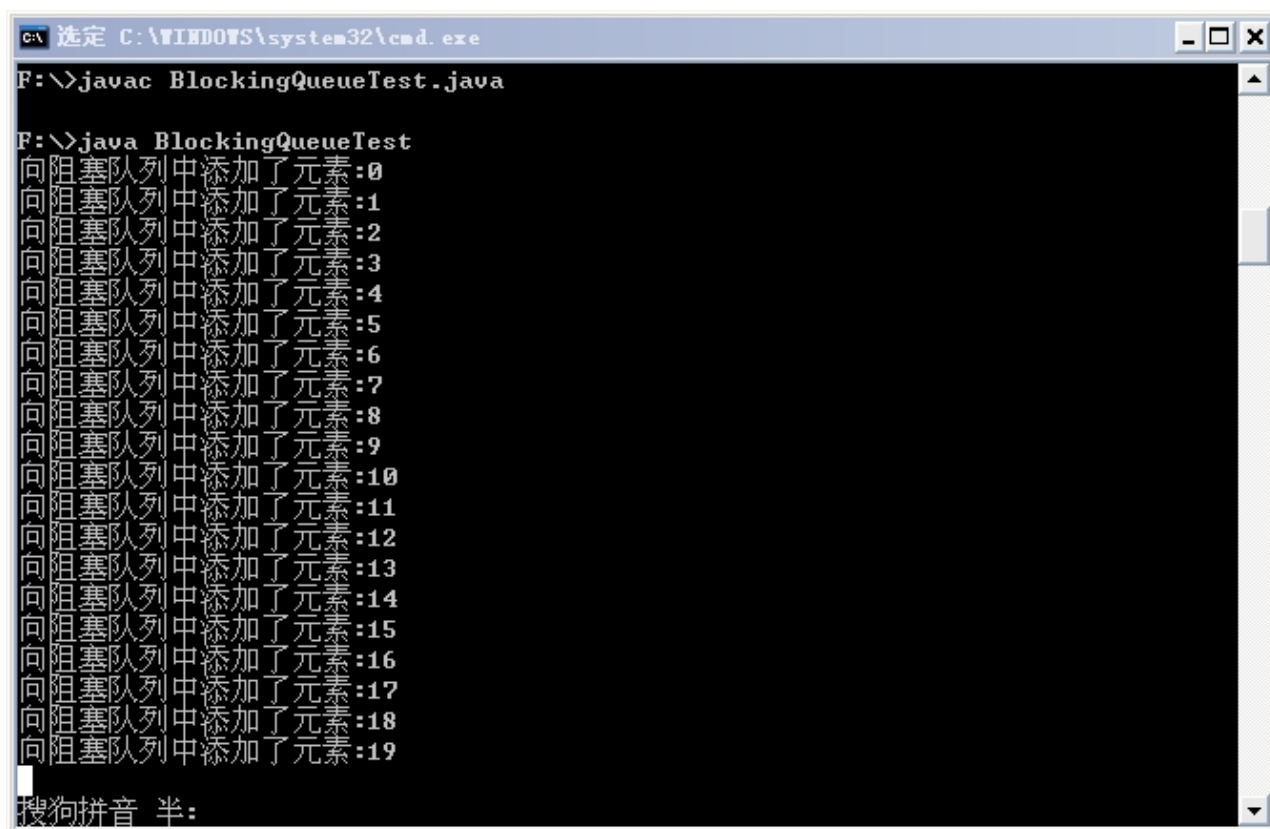
阻塞队列是 Java 5 并发新特性中的内容，阻塞队列的接口是 `java.util.concurrent.BlockingQueue`，它有多个实现类：`ArrayBlockingQueue`、`DelayQueue`、`LinkedBlockingQueue`、`PriorityBlockingQueue`、`SynchronousQueue` 等，用法大同小异，具体可查看 JDK 文档，这里简单举例看下 `ArrayBlockingQueue`，它实现了一个有界队列，当队列满时，便会阻塞等待，直到有元素出队，后续的元素才可以被加入队列。

看下面的例子：

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingQueueTest{
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<String> bqueue = new ArrayBlockingQueue<String>(20);
        for (int i = 0; i < 30; i++) {
            //将指定元素添加到此队列中
            bqueue.put("加入元素" + i);
            System.out.println("向阻塞队列中添加了元素:" + i);
        }
        System.out.println("程序到此运行结束，即将退出----");
    }
}
```

输出结果如下：



```

C:\ 选定 C:\WINDOWS\system32\cmd.exe
F:\>javac BlockingQueueTest.java

F:\>java BlockingQueueTest
向阻塞队列中添加了元素:0
向阻塞队列中添加了元素:1
向阻塞队列中添加了元素:2
向阻塞队列中添加了元素:3
向阻塞队列中添加了元素:4
向阻塞队列中添加了元素:5
向阻塞队列中添加了元素:6
向阻塞队列中添加了元素:7
向阻塞队列中添加了元素:8
向阻塞队列中添加了元素:9
向阻塞队列中添加了元素:10
向阻塞队列中添加了元素:11
向阻塞队列中添加了元素:12
向阻塞队列中添加了元素:13
向阻塞队列中添加了元素:14
向阻塞队列中添加了元素:15
向阻塞队列中添加了元素:16
向阻塞队列中添加了元素:17
向阻塞队列中添加了元素:18
向阻塞队列中添加了元素:19
搜狗拼音 半:

```

从执行结果中可以看出，由于队列中元素的数量限制在了 20 个，因此添加 20 个元素后，其他元素便在队列外阻塞等待，程序并没有终止。

如果队列已满后，我们将队首元素移出，并可以继续向阻塞队列中添加元素，修改代码如下：

```

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingQueueTest{
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<String> bqueue = new ArrayBlockingQueue<String>(20);
        for (int i = 0; i < 30; i++) {
            //将指定元素添加到此队列中
            bqueue.put("" + i);
            System.out.println("向阻塞队列中添加了元素:" + i);
            if(i > 18){
                //从队列中获取队头元素，并将其移出队列
                System.out.println("从阻塞队列中移除元素: " + bqueue.take());
            }
        }
        System.out.println("程序到此运行结束，即将退出----");
    }
}

```

执行结果如下：

```

F:\>java BlockingQueueTest
向阻塞队列中添加了元素:0
向阻塞队列中添加了元素:1
向阻塞队列中添加了元素:2
向阻塞队列中添加了元素:3
向阻塞队列中添加了元素:4
向阻塞队列中添加了元素:5
向阻塞队列中添加了元素:6
向阻塞队列中添加了元素:7
向阻塞队列中添加了元素:8
向阻塞队列中添加了元素:9
向阻塞队列中添加了元素:10
向阻塞队列中添加了元素:11
向阻塞队列中添加了元素:12
向阻塞队列中添加了元素:13
向阻塞队列中添加了元素:14
向阻塞队列中添加了元素:15
向阻塞队列中添加了元素:16
向阻塞队列中添加了元素:17
向阻塞队列中添加了元素:18
向阻塞队列中添加了元素:19
从阻塞队列中移除元素: 0
向阻塞队列中添加了元素:20
从阻塞队列中移除元素: 1
向阻塞队列中添加了元素:21
从阻塞队列中移除元素: 2
向阻塞队列中添加了元素:22
从阻塞队列中移除元素: 3
向阻塞队列中添加了元素:23
从阻塞队列中移除元素: 4
向阻塞队列中添加了元素:24
从阻塞队列中移除元素: 5
向阻塞队列中添加了元素:25
从阻塞队列中移除元素: 6
向阻塞队列中添加了元素:26
从阻塞队列中移除元素: 7
向阻塞队列中添加了元素:27
从阻塞队列中移除元素: 8
向阻塞队列中添加了元素:28
从阻塞队列中移除元素: 9
向阻塞队列中添加了元素:29
从阻塞队列中移除元素: 10
程序到此运行结束，即将退出----
搜狗拼音 半:

```

从结果中可以看出，当添加了第 20 个元素后，我们从队首移出一个元素，这样便可以继续向队列中添加元素，之后每添加一个元素，便从将队首元素移除，这样程序便可以执行结束。

阻塞栈

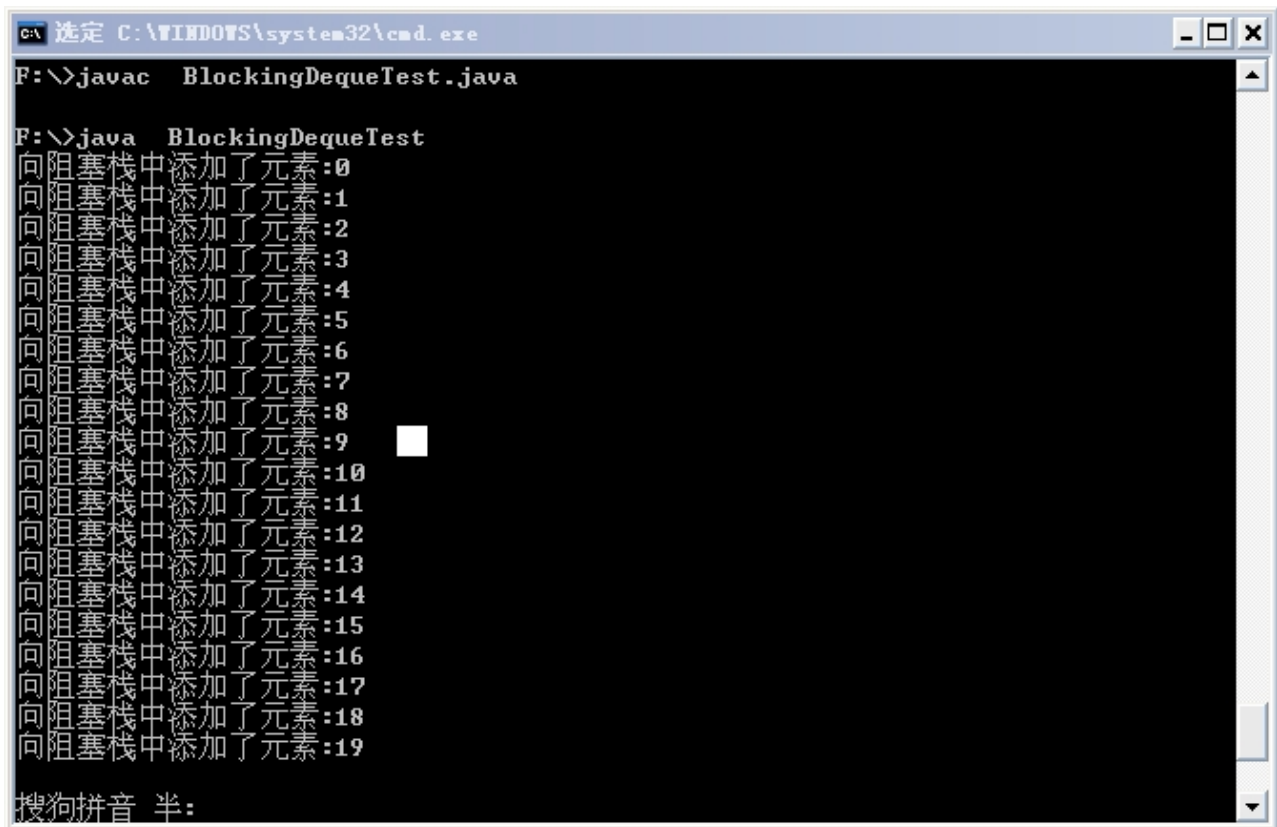
阻塞栈与阻塞队列相似，只是它是 Java 6 中加入的新特性，阻塞栈的接口 `java.util.concurrent.BlockingDeque` 也有很多实现类，使用方法也比较相似，具体查看 JDK 文档。

下面同样给出一个简单的例子：

```
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;

public class BlockingDequeTest {
    public static void main(String[] args) throws InterruptedException {
        BlockingDeque<String> bDeque = new LinkedBlockingDeque<String>(20);
        for (int i = 0; i < 30; i++) {
            //将指定元素添加到此阻塞栈中
            bDeque.putFirst("" + i);
            System.out.println("向阻塞栈中添加了元素:" + i);
        }
        System.out.println("程序到此运行结束，即将退出----");
    }
}
```

执行结果如下：



```

C:\>选定 C:\WINDOWS\system32\cmd.exe
F:\>javac  BlockingDequeTest.java
F:\>java  BlockingDequeTest
向阻塞栈中添加了元素:0
向阻塞栈中添加了元素:1
向阻塞栈中添加了元素:2
向阻塞栈中添加了元素:3
向阻塞栈中添加了元素:4
向阻塞栈中添加了元素:5
向阻塞栈中添加了元素:6
向阻塞栈中添加了元素:7
向阻塞栈中添加了元素:8
向阻塞栈中添加了元素:9
向阻塞栈中添加了元素:10
向阻塞栈中添加了元素:11
向阻塞栈中添加了元素:12
向阻塞栈中添加了元素:13
向阻塞栈中添加了元素:14
向阻塞栈中添加了元素:15
向阻塞栈中添加了元素:16
向阻塞栈中添加了元素:17
向阻塞栈中添加了元素:18
向阻塞栈中添加了元素:19
搜狗拼音 半:

```

程序依然会阻塞等待，我们改为如下代码：

```

import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;

public class BlockingDequeTest {
    public static void main(String[] args) throws InterruptedException {
        BlockingDeque<String> bDeque = new LinkedBlockingDeque<String>(20);
        for (int i = 0; i < 30; i++) {
            //将指定元素添加到此阻塞栈中
            bDeque.putFirst("" + i);
            System.out.println("向阻塞栈中添加了元素:" + i);
            if(i > 18){
                //从阻塞栈中取出栈顶元素，并将其移出
                System.out.println("从阻塞栈中移出了元素: " + bDeque.pollFirst());
            }
        }
        System.out.println("程序到此运行结束，即将退出-----");
    }
}

```

执行结果如下：

```

C:\> java BlockingDequeTest
向阻塞栈中添加了元素:0
向阻塞栈中添加了元素:1
向阻塞栈中添加了元素:2
向阻塞栈中添加了元素:3
向阻塞栈中添加了元素:4
向阻塞栈中添加了元素:5
向阻塞栈中添加了元素:6
向阻塞栈中添加了元素:7
向阻塞栈中添加了元素:8
向阻塞栈中添加了元素:9
向阻塞栈中添加了元素:10
向阻塞栈中添加了元素:11
向阻塞栈中添加了元素:12
向阻塞栈中添加了元素:13
向阻塞栈中添加了元素:14
向阻塞栈中添加了元素:15
向阻塞栈中添加了元素:16
向阻塞栈中添加了元素:17
向阻塞栈中添加了元素:18
向阻塞栈中添加了元素:19
从阻塞栈中移出了元素: 19
向阻塞栈中添加了元素:20
从阻塞栈中移出了元素: 20
向阻塞栈中添加了元素:21
从阻塞栈中移出了元素: 21
向阻塞栈中添加了元素:22
从阻塞栈中移出了元素: 22
向阻塞栈中添加了元素:23
从阻塞栈中移出了元素: 23
向阻塞栈中添加了元素:24
从阻塞栈中移出了元素: 24
向阻塞栈中添加了元素:25
从阻塞栈中移出了元素: 25
向阻塞栈中添加了元素:26
从阻塞栈中移出了元素: 26
向阻塞栈中添加了元素:27
从阻塞栈中移出了元素: 27
向阻塞栈中添加了元素:28
从阻塞栈中移出了元素: 28
向阻塞栈中添加了元素:29
从阻塞栈中移出了元素: 29
程序到此运行结束，即将退出----
搜狗拼音 半:

```

从结果中可以看出，当添加了第 20 个元素后，我们将栈顶元素移出，这样便可以继续向栈中添加元素，之后每添加一个元素，便将栈顶元素移出，这样程序便可以执行结束。



23



并发新特性—障碍器 CyclicBarrier



CyclicBarrier（又叫障碍器）同样是 Java 5 中加入的新特性，使用时需要导入 `java.util.concurrent.CyclicBarrier`。它适用于这样一种情况：你希望创建一组任务，它们并发地执行工作，另外的一个任务在这一组任务并发执行结束前一直阻塞等待，直到该组任务全部执行结束，这个任务才得以执行。这非常像 `CountDownLatch`，只是 `CountDownLatch` 是只触发一次的事件，而 `CyclicBarrier` 可以多次重用。

下面给出一个简单的实例来说明其用法：

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierTest {
    public static void main(String[] args) {
        //创建CyclicBarrier对象，
        //并设置执行完一组5个线程的并发任务后，再执行MainTask任务
        CyclicBarrier cb = new CyclicBarrier(5, new MainTask());
        new SubTask("A", cb).start();
        new SubTask("B", cb).start();
        new SubTask("C", cb).start();
        new SubTask("D", cb).start();
        new SubTask("E", cb).start();
    }
}

/**
 * 最后执行的任务
 */
class MainTask implements Runnable {
    public void run() {
        System.out.println(".....终于要执行最后的任务了.....");
    }
}

/**
 * 一组并发任务
 */
class SubTask extends Thread {
    private String name;
    private CyclicBarrier cb;

    SubTask(String name, CyclicBarrier cb) {
        this.name = name;
        this.cb = cb;
    }

    public void run() {
```

```

        System.out.println("[并发任务" + name + "] 开始执行");
        for (int i = 0; i < 999999; i++) ; //模拟耗时的任务
        System.out.println("[并发任务" + name + "] 开始执行完毕，通知障碍物");
        try {
            //每执行完一项任务就通知障碍物
            cb.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}

```

某次执行的结果如下：

```

[并发任务A] 开始执行
[并发任务B] 开始执行
[并发任务D] 开始执行
[并发任务E] 开始执行
[并发任务A] 开始执行完毕，通知障碍物
[并发任务E] 开始执行完毕，通知障碍物
[并发任务D] 开始执行完毕，通知障碍物
[并发任务C] 开始执行
[并发任务B] 开始执行完毕，通知障碍物
[并发任务C] 开始执行完毕，通知障碍物
.....终于要执行最后的任务了.....

```

从结果可以看出：MainTask 任务在一组中的 5 个任务执行完后才开始执行。



24



并发新特性—信号量 Semaphore



在操作系统中，信号量是个很重要的概念，它在控制进程间的协作方面有着非常重要的作用，通过对信号量的不同操作，可以分别实现进程间的互斥与同步。当然它也可以用于多线程的控制，我们完全可以通过使用信号量来自定义实现类似 Java 中的 synchronized、wait、notify 机制。

Java 并发包中的信号量 Semaphore 实际上是一个功能完毕的计数信号量，从概念上讲，它维护了一个许可集合，对控制一定资源的消费与回收有着很重要的意义。Semaphore 可以控制某个资源被同时访问的任务数，它通过 acquire() 获取一个许可，release() 释放一个许可。如果被同时访问的任务数已满，则其他 acquire 的任务进入等待状态，直到有一个任务被 release 掉，它才能得到许可。

下面给出一个采用 Semaphore 控制并发访问数量的示例程序：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
public class SemaphoreTest{
    public static void main(String[] args) {
        //采用新特性来启动和管理线程——内部使用线程池
        ExecutorService exec = Executors.newCachedThreadPool();
        //只允许5个线程同时访问
        final Semaphore semp = new Semaphore(5);
        //模拟10个客户端访问
        for (int index = 0; index < 10; index++){
            final int num = index;
            Runnable run = new Runnable() {
                public void run() {
                    try {
                        //获取许可
                        semp.acquire();
                        System.out.println("线程" +
                            Thread.currentThread().getName() + "获得许可: " + num);
                        //模拟耗时的任务
                        for (int i = 0; i < 999999; i++) ;
                        //释放许可
                        semp.release();
                        System.out.println("线程" +
                            Thread.currentThread().getName() + "释放许可: " + num);
                        System.out.println("当前允许进入的任务个数: " +
                            semp.availablePermits());
                    } catch (InterruptedException e){
                        e.printStackTrace();
                    }
                }
            };
            exec.execute(run);
        }
    }
}
```

```

    }
    //关闭线程池
    exec.shutdown();
  }
}

```

某次执行的结果如下：

```

线程pool-1-thread-1获得许可：0
线程pool-1-thread-1释放许可：0
当前允许进入的任务个数：5
线程pool-1-thread-2获得许可：1
线程pool-1-thread-6获得许可：5
线程pool-1-thread-4获得许可：3
线程pool-1-thread-8获得许可：7
线程pool-1-thread-2释放许可：1
当前允许进入的任务个数：2
线程pool-1-thread-5获得许可：4
线程pool-1-thread-8释放许可：7
线程pool-1-thread-3获得许可：2
线程pool-1-thread-4释放许可：3
线程pool-1-thread-10获得许可：9
线程pool-1-thread-6释放许可：5
线程pool-1-thread-10释放许可：9
当前允许进入的任务个数：2
线程pool-1-thread-3释放许可：2
当前允许进入的任务个数：1
线程pool-1-thread-5释放许可：4
当前允许进入的任务个数：3
线程pool-1-thread-7获得许可：6
线程pool-1-thread-9获得许可：8
线程pool-1-thread-7释放许可：6
当前允许进入的任务个数：5
当前允许进入的任务个数：3
当前允许进入的任务个数：3
当前允许进入的任务个数：3
线程pool-1-thread-9释放许可：8
当前允许进入的任务个数：5

```

可以看出，Semaphore 允许并发访问的任务数一直为 5，当然，这里还很容易看出一点，就是 Semaphore 仅仅是对资源的并发访问的任务数进行监控，而不会保证线程安全，因此，在访问的时候，要自己控制线程的安全访问。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/java-concurrency/>