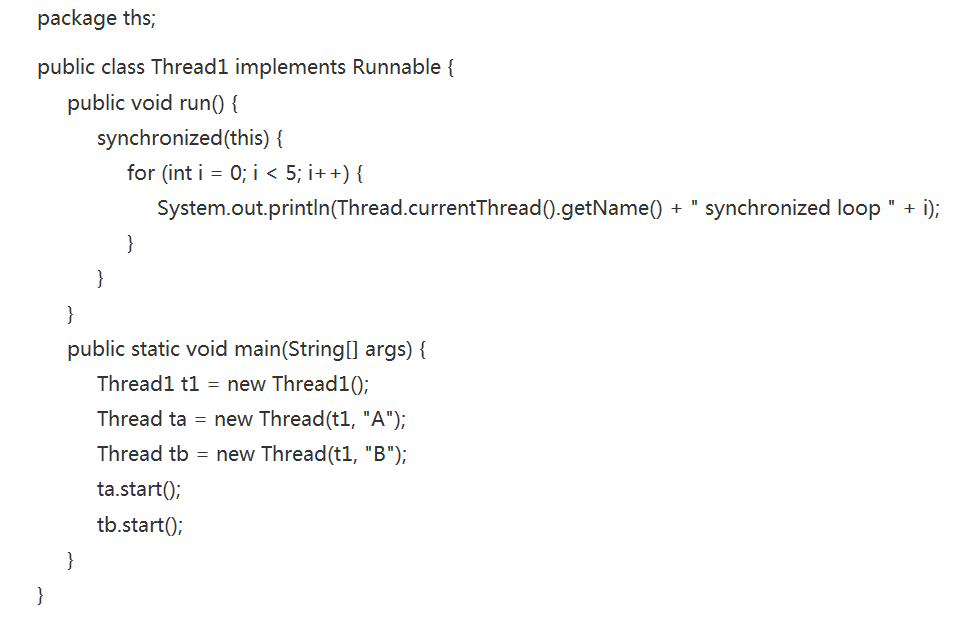
**原子性**意味着一个线程一次只能执行由一个指定监控对象（lock）保护的代码，从而防止多个线程在更新共享状态时相互冲突

# [java synchronized详解](http://www.cnblogs.com/GnagWang/archive/2011/02/27/1966606.html)

Java语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码

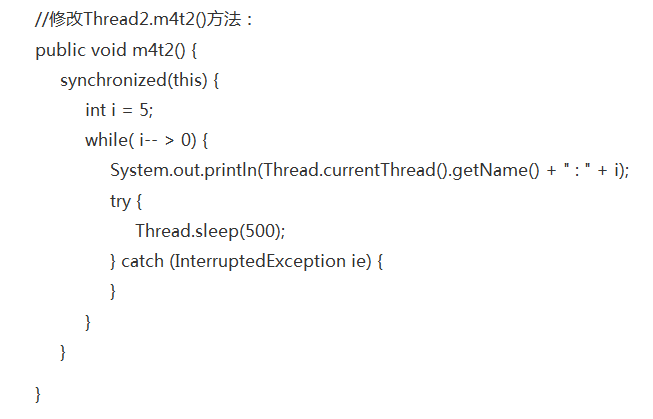
1. 当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块



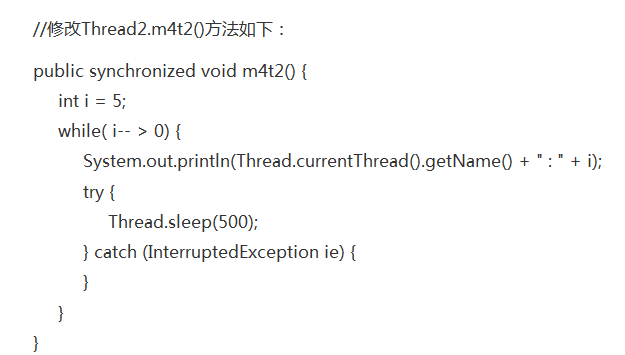
1. 当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块



3．当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞



4. 当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞



打个比方：一个object就像一个大房子，大门永远打开。房子里有 很多房间（也就是方法）。

这些房间有上锁的（synchronized方法）， 和不上锁之分（普通方法）。房门口放着一把钥匙（key），这把钥匙可以打开所有上锁的房间。

另外我把所有想调用该对象方法的线程比喻成想进入这房子某个 房间的人。所有的东西就这么多了，下面我们看看这些东西之间如何作用的。

在此我们先来明确一下我们的前提条件。该对象至少有一个synchronized方法，否则这个key还有啥意义。当然也就不会有我们的这个主题了。

一个人想进入某间上了锁的房间，他来到房子门口，看见钥匙在那儿（说明暂时还没有其他人要使用上锁的 房间）。于是他走上去拿到了钥匙

，并且按照自己 的计划使用那些房间。注意一点，他每次使用完一次上锁的房间后会马上把钥匙还回去。即使他要连续使用两间上锁的房间，

中间他也要把钥匙还回去，再取回来。

因此，普通情况下钥匙的使用原则是：“随用随借，用完即还。”

这时其他人可以不受限制的使用那些不上锁的房间，一个人用一间可以，两个人用一间也可以，没限制。但是如果当某个人想要进入上锁的房

间，他就要跑到大门口去看看了。有钥匙当然拿了就走，没有的话，就只能等了。

要是很多人在等这把钥匙，等钥匙还回来以后，谁会优先得到钥匙？Not guaranteed。象前面例子里那个想连续使用两个上锁房间的家伙，他

中间还钥匙的时候如果还有其他人在等钥匙，那么没有任何保证这家伙能再次拿到。 （JAVA规范在很多地方都明确说明不保证，象

Thread.sleep()休息后多久会返回运行，相同优先权的线程那个首先被执行，当要访问对象的锁被 释放后处于等待池的多个线程哪个会优先得

到，等等。我想最终的决定权是在JVM，之所以不保证，就是因为JVM在做出上述决定的时候，绝不是简简单单根据 一个条件来做出判断，而是

根据很多条。而由于判断条件太多，如果说出来可能会影响JAVA的推广，也可能是因为知识产权保护的原因吧。SUN给了个不保证 就混过去了

。无可厚非。但我相信这些不确定，并非完全不确定。因为计算机这东西本身就是按指令运行的。即使看起来很随机的现象，其实都是有规律

可寻。学过 计算机的都知道，计算机里随机数的学名是伪随机数，是人运用一定的方法写出来的，看上去随机罢了。另外，或许是因为要想弄

的确定太费事，也没多大意义，所 以不确定就不确定了吧）

**ReentrantLock 类**

java.util.concurrent.lock 中的 Lock 框架是锁定的一个抽象，它允许把锁定的实现作为 Java 类，而不是作为语言的特性来实现。这就为 Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。 ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上。）

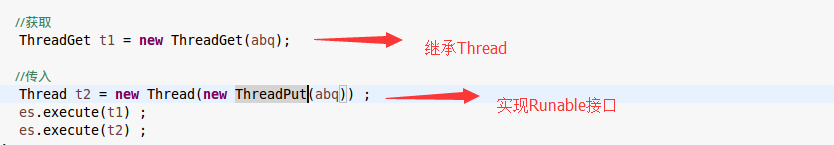
reentrant 锁意味着什么呢？简单来说，它有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放。这模仿了 synchronized 的语义；如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续） synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。

在查看清单 1 中的代码示例时，可以看到 Lock 和 synchronized 有一点明显的区别 —— lock 必须在 finally 块中释放。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放！这一点区别看起来可能没什么，但是实际上，它极为重要。忘记在 finally 块中释放锁，可能会在程序中留下一个定时炸弹，当有一天炸弹爆炸时，您要花费很大力气才有找到源头在哪。而使用同步，JVM 将确保锁会获得自动释放。

**什么时候选择用 ReentrantLock 代替 synchronized**

既然如此，我们什么时候才应该使用 ReentrantLock 呢？答案非常简单 —— 在确实需要一些 synchronized 所没有的特性的时候，比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者锁投票。 ReentrantLock 还具有可伸缩性的好处，应当在高度争用的情况下使用它，但是请记住，大多数 synchronized 块几乎从来没有出现过争用，所以可以把高度争用放在一边。我建议用 synchronized 开发，直到确实证明 synchronized 不合适，而不要仅仅是假设如果使用 ReentrantLock “性能会更好”。请记住，这些是供高级用户使用的高级工具。（而且，真正的高级用户喜欢选择能够找到的最简单工具，直到他们认为简单的工具不适用为止。）。一如既往，首先要把事情做好，然后再考虑是不是有必要做得更快。

Java之队列

队列离不开Thread类，有两种实现方法1.继承Thead类去重写run方法2.实现Runable接口，但是需要把实例传入进去

队列是一种“FIFO”先进先出的数据结构.可以想象每年在火车站中买票的人群所组成的"队列"。

  |队员1 队员2 队员3 队员3 队员4 队员5 队员6 队员7 队员8|

 想象两个指针 一个指向队头 一个指向队尾，加人到队列中是从队尾入的，出队是从队头出队的

**LinkedBlockingQueue**的容量是没有上限的（说的不准确，在不指定时容量为Integer.MAX\_VALUE，不要然的话在put时怎么会受阻呢），但是也可以选择指定其最大容量，它是基于链表的队列，此队列按 FIFO（先进先出）排序元素

**ArrayBlockingQueue**在构造时需要指定容量， 并可以选择是否需要公平性，如果公平参数被设置true，等待时间最长的线程会优先得到处理（其实就是通过将ReentrantLock设置为true来 达到这种公平性的：即等待时间最长的线程会先操作）。通常，公平性会使你在性能上付出代价，只有在的确非常需要的时候再使用它。它是基于数组的阻塞循环队 列，此队列按 FIFO（先进先出）原则对元素进行排序

**PriorityBlockingQueue**是一个带优先级的 队列，而不是先进先出队列。元素按优先级顺序被移除，该队列也没有上限（看了一下源码，PriorityBlockingQueue是对 PriorityQueue的再次包装，是基于堆数据结构的，而PriorityQueue是没有容量限制的，与ArrayList一样，所以在优先阻塞 队列上put时是不会受阻的。虽然此队列逻辑上是无界的，但是由于资源被耗尽，所以试图执行添加操作可能会导致 OutOfMemoryError），但是如果队列为空，那么取元素的操作take就会阻塞，所以它的检索操作take是受阻的。另外，往入该队列中的元 素要具有比较能力

**DelayQueue**（基于PriorityQueue来实现的）是一个存放Delayed 元素的无界阻塞队列，只有在延迟期满时才能从中提取元素。该队列的头部是延迟期满后保存时间最长的 Delayed 元素。如果延迟都还没有期满，则队列没有头部，并且poll将返回null。当一个元素的 getDelay(TimeUnit.NANOSECONDS) 方法返回一个小于或等于零的值时，则出现期满，poll就以移除这个元素了。此队列不允许使用 null 元素

[**精巧好用的DelayQueue**](http://www.cnblogs.com/jobs/archive/2007/04/27/730255.html)

我们谈一下实际的场景吧。我们在开发中，有如下场景  
  
a) 关闭空闲连接。服务器中，有很多客户端的连接，空闲一段时间之后需要关闭之。  
b) 缓存。缓存中的对象，超过了空闲时间，需要从缓存中移出。  
c) 任务超时处理。在网络协议滑动窗口请求应答式交互时，处理超时未响应的请求。  
  
一种笨笨的办法就是，使用一个后台线程，遍历所有对象，挨个检查。这种笨笨的办法简单好用，但是对象数量过多时，可能存在性能问题，检查间隔时间不好设置，间隔时间过大，影响精确度，多小则存在效率问题。而且做不到按超时的时间顺序处理。   
  
这场景，使用DelayQueue最适合了。  
  
DelayQueue是java.util.concurrent中提供的一个很有意思的类。很巧妙，非常棒！但是java doc和Java SE 5.0的source中都没有提供Sample。我最初在阅读ScheduledThreadPoolExecutor源码时，发现DelayQueue的妙用。随后在实际工作中，应用在session超时管理，网络应答通讯协议的请求超时处理。  
  
本文将会对DelayQueue做一个介绍，然后列举应用场景。并且提供一个Delayed接口的实现和Sample代码。  
  
DelayQueue是一个BlockingQueue，其特化的参数是Delayed。（不了解BlockingQueue的同学，先去了解BlockingQueue再看本文）  
Delayed扩展了Comparable接口，比较的基准为延时的时间值，Delayed接口的实现类getDelay的返回值应为固定值（final）。DelayQueue内部是使用PriorityQueue实现的。  
  
DelayQueue = BlockingQueue + PriorityQueue + Delayed  
  
DelayQueue的关键元素BlockingQueue、PriorityQueue、Delayed。可以这么说，DelayQueue是一个使用优先队列（PriorityQueue）实现的BlockingQueue，优先队列的比较基准值是时间。

以下是Delayed的实现

import java.util.concurrent.Delayed;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicLong;

public class DelayItem<T> implements Delayed {

/\*\* Base of nanosecond timings, to avoid wrapping \*/

private static final long NANO\_ORIGIN = System.nanoTime();

/\*\*

\* Returns nanosecond time offset by origin

\*/

final static long now() {

return System.nanoTime() - NANO\_ORIGIN;

}

/\*\*

\* Sequence number to break scheduling ties, and in turn to guarantee FIFO order among tied

\* entries.

\*/

private static final AtomicLong sequencer = new AtomicLong(0);

/\*\* Sequence number to break ties FIFO \*/

private final long sequenceNumber;

/\*\* The time the task is enabled to execute in nanoTime units \*/

private final long time;

private final T item;

public DelayItem(T submit, long timeout) {

this.time = now() + timeout;

this.item = submit;

this.sequenceNumber = sequencer.getAndIncrement();

}

public T getItem() {

return this.item;

}

public long getDelay(TimeUnit unit) {

long d = unit.convert(time - now(), TimeUnit.NANOSECONDS);

return d;

}

public int compareTo(Delayed other) {

if (other == this) // compare zero ONLY if same object

return 0;

if (other instanceof DelayItem) {

DelayItem x = (DelayItem) other;

long diff = time - x.time;

if (diff < 0)

return -1;

else if (diff > 0)

return 1;

else if (sequenceNumber < x.sequenceNumber)

return -1;

else

return 1;

}

long d = (getDelay(TimeUnit.NANOSECONDS) - other.getDelay(TimeUnit.NANOSECONDS));

return (d == 0) ? 0 : ((d < 0) ? -1 : 1);

}

}

以下是Cache的实现，包括了put和get方法，还包括了可执行的main函数。

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.ConcurrentMap;

import java.util.concurrent.DelayQueue;

import java.util.concurrent.TimeUnit;

import java.util.logging.Level;

import java.util.logging.Logger;

public class Cache<K, V> {

private static final Logger LOG = Logger.getLogger(Cache.class.getName());

private ConcurrentMap<K, V> cacheObjMap = new ConcurrentHashMap<K, V>();

private DelayQueue<DelayItem<Pair<K, V>>> q = new DelayQueue<DelayItem<Pair<K, V>>>();

private Thread daemonThread;

public Cache() {

Runnable daemonTask = new Runnable() {

public void run() {

daemonCheck();

}

};

daemonThread = new Thread(daemonTask);

daemonThread.setDaemon(true);

daemonThread.setName("Cache Daemon");

daemonThread.start();

}

private void daemonCheck() {

if (LOG.isLoggable(Level.INFO))

LOG.info("cache service started.");

for (;;) {

try {

DelayItem<Pair<K, V>> delayItem = q.take();

if (delayItem != null) {

// 超时对象处理

Pair<K, V> pair = delayItem.getItem();

cacheObjMap.remove(pair.first, pair.second); // compare and remove

}

} catch (InterruptedException e) {

if (LOG.isLoggable(Level.SEVERE))

LOG.log(Level.SEVERE, e.getMessage(), e);

break;

}

}

if (LOG.isLoggable(Level.INFO))

LOG.info("cache service stopped.");

}

// 添加缓存对象

s

public V get(K key) {

return cacheObjMap.get(key);

}

// 测试入口函数

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

Cache<Integer, String> cache = new Cache<Integer, String>();

cache.put(1, "aaaa", 3, TimeUnit.SECONDS);

Thread.sleep(1000 \* 2);

{

String str = cache.get(1);

System.out.println(str);

}

Thread.sleep(1000 \* 2);

{

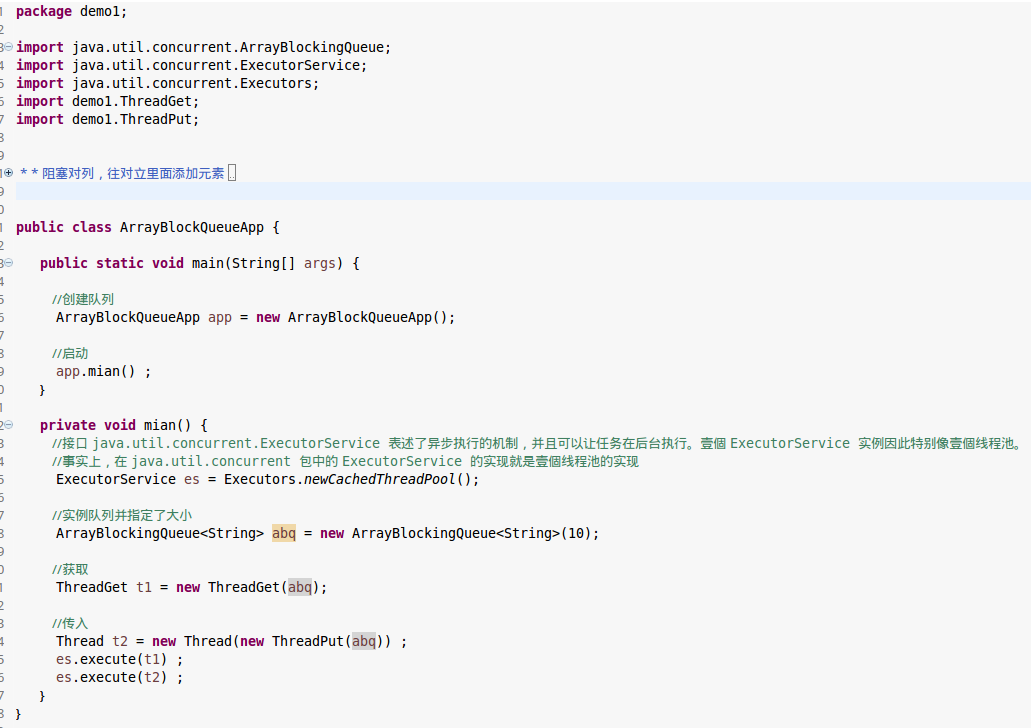
String str = cache.get(1);

System.out.println(str);

}

}

}

再来个例子





### Java并发编程

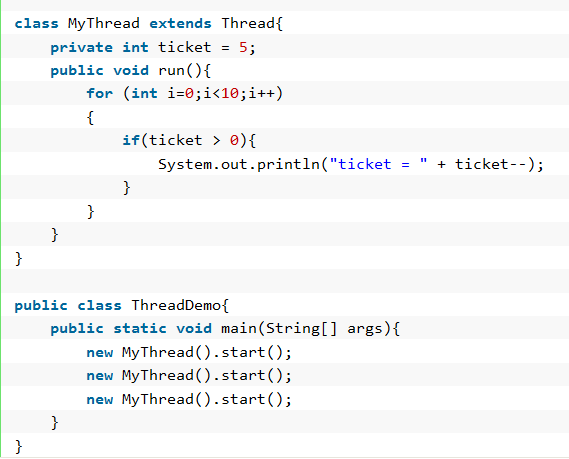
##### Runnable和Thread实现多线程的区别

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Runnable接口相比继承Thread类有如下优势：

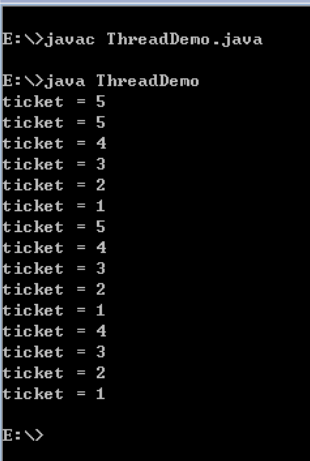
    1、可以避免由于Java的单继承特性而带来的局限；

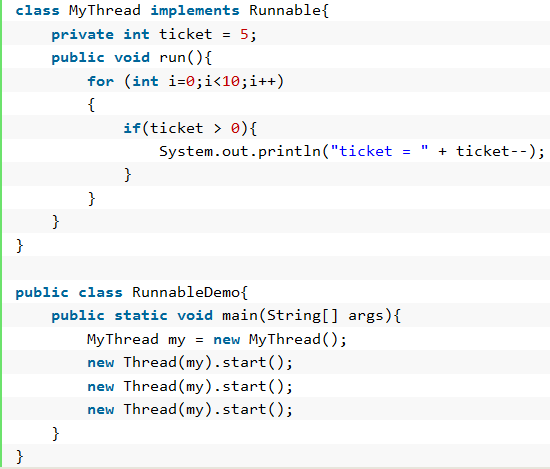
    2、增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的；

    3、适合多个相同程序代码的线程区处理同一资源的情况。

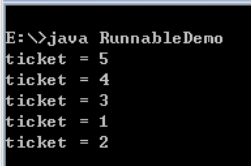


结果





结果



 针对以上代码补充三点：

    1、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。

    2、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。

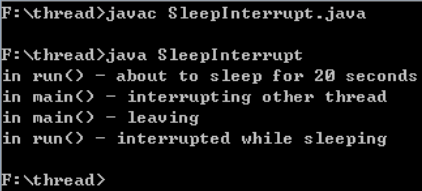
    3、在第二种方法中，由于3个Thread对象共同执行一个Runnable对象中的代码，因此可能会造成线程的不安全，比如可能ticket会输出-1（如果我们System.out....语句前加上线程休眠操作，该情况将很有可能出现），这种情况的出现是由于，一个线程在判断ticket为1>0后，还没有来得及减1，另一个线程已经将ticket减1，变为了0，那么接下来之前的线程再将ticket减1，便得到了-1。这就需要加入同步操作（即互斥锁），确保同一时刻只有一个线程在执行每次for循环中的操作。而在第一种方法中，并不需要加入同步操作，因为每个线程执行自己Thread对象中的代码，不存在多个线程共同执行同一个方法的情况

##### 线程中断

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。



结果

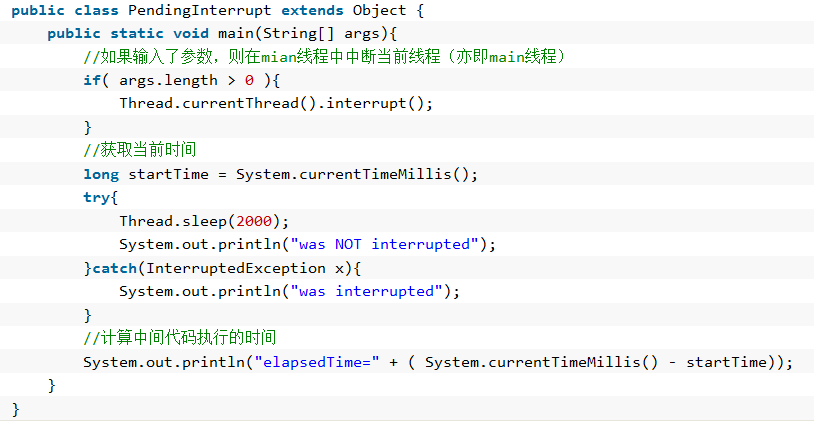


主线程启动新线程后，自身休眠2秒钟，允许新线程获得运行时间。新线程打印信息“about to sleep for 20 seconds”后，继而休眠20秒钟，大约2秒钟后，main线程通知新线程中断，那么新线程的20秒的休眠将被打断，从而抛出InterruptException异常，执行跳转到catch块，打印出“interrupted while sleeping”信息，并立即从run（）方法返回，然后消亡，而不会打印出catch块后面的“leaving normally”信息。

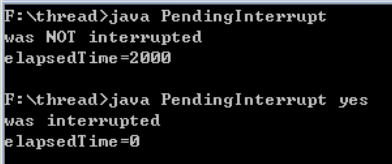
    请注意：由于不确定的线程规划，上图运行结果的后两行可能顺序相反，这取决于主线程和新线程哪个先消亡。但前两行信息的顺序必定如上图所示。

    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。

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

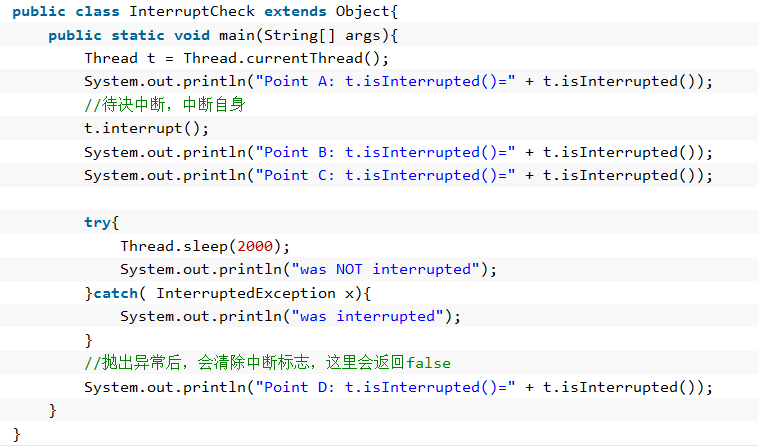


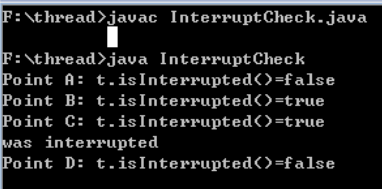
如果PendingInterrupt不带任何命令行参数，那么线程不会被中断，最终输出的时间差距应该在2000附近（具体时间由系统决定，不精确），如果PendingInterrupt带有命令行参数，则调用中断当前线程的代码，但main线程仍然运行，最终输出的时间差距应该远小于2000，因为线程尚未休眠，便被中断，因此，一旦调用sleep（）方法，会立即打印出catch块中的信息。执行结果如下:



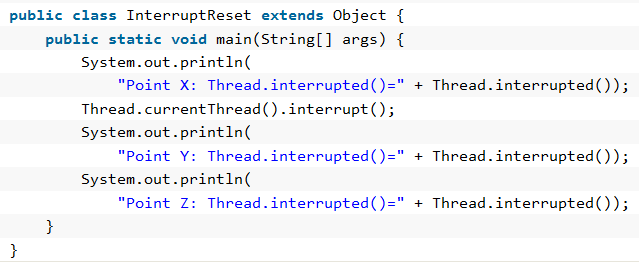
这种模式下，main线程中断它自身。除了将中断标志（它是Thread的内部标志）设置为true外，没有其他任何影响。线程被中断了，但main线程仍然运行，main线程继续监视实时时钟，并进入try块，一旦调用sleep（）方法，它就会注意到待决中断的存在，并抛出InterruptException。于是执行跳转到catch块，并打印出线程被中断的信息。最后，计算并打印出时间差。

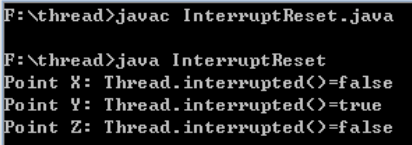
可以在Thread对象上调用isInterrupted（）方法来检查任何线程的中断状态。这里需要注意：线程一旦被中断，isInterrupted（）方法便会返回true，而一旦sleep（）方法抛出异常，它将清空中断标志，此时isInterrupted（）方法将返回false。





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





**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。

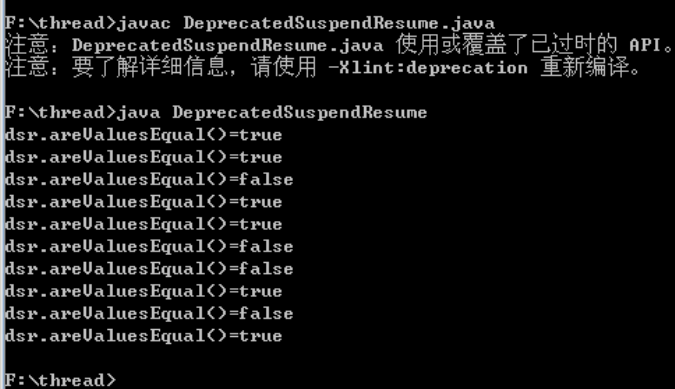
yield可以直接用Thread类调用，yield让出CPU执行权给同等级的线程，如果没有相同级别的线程在等待CPU的执行权，则该线程继续执行。

##### 线程挂起、恢复与终止的正确方法

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







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），这段时间不适宜挂起线程，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

**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**

##### 守护线程与线程阻塞的四种情况

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：

    1、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。

    2、在守护线程中产生的新线程也是守护线程。    
    3、 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。

线程可以阻塞于四种状态：

    1、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；

    2、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）

    3、线程阻塞与不同I/O的方式有多种。常见的一种方式是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；

    4、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

##### volatile变量修饰符—意料之外的问题

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](http://baike.baidu.com/view/390935.htm)的使用变得非常重要。

在当前的Java内存模型下，[线程](http://baike.baidu.com/view/1053.htm)可以把[变量](http://baike.baidu.com/view/296689.htm)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](http://baike.baidu.com/view/296689.htm)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](http://baike.baidu.com/view/296689.htm)声明为volatile（也可以使用同步，参见<http://blog.csdn.net/ns_code/article/details/17288243>），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](http://baike.baidu.com/view/684821.htm)在每次被[线程](http://baike.baidu.com/view/1053.htm)访问时，都强迫从[共享内存](http://baike.baidu.com/view/120892.htm)中重读该成员变量的值。而且，当[成员变量](http://baike.baidu.com/view/684821.htm)发生变化时，强迫线程将变化值回写到[共享内存](http://baike.baidu.com/view/120892.htm)。这样在任何时刻，两个不同的线程总是看到某个[成员变量](http://baike.baidu.com/view/684821.htm)的同一个值。

[Java语言](http://baike.baidu.com/view/229611.htm)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](http://baike.baidu.com/view/684821.htm)的私有拷贝，而且只当线程进入或者离开[同步代码块](http://baike.baidu.com/view/8314374.htm)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](http://baike.baidu.com/view/684821.htm)的变化。而volatile[关键字](http://baike.baidu.com/view/390935.htm)就是提示JVM：对于这个[成员变量](http://baike.baidu.com/view/684821.htm)，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volatilei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](http://baike.baidu.com/view/684821.htm)上使用volatile。当要访问的[变量](http://baike.baidu.com/view/296689.htm)已在synchronized代码块中，或者为[常量](http://baike.baidu.com/view/346799.htm)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](http://baike.baidu.com/view/2136208.htm)，所以在效率上比较低，因此一定在必要时才使用此[关键字](http://baike.baidu.com/view/390935.htm)

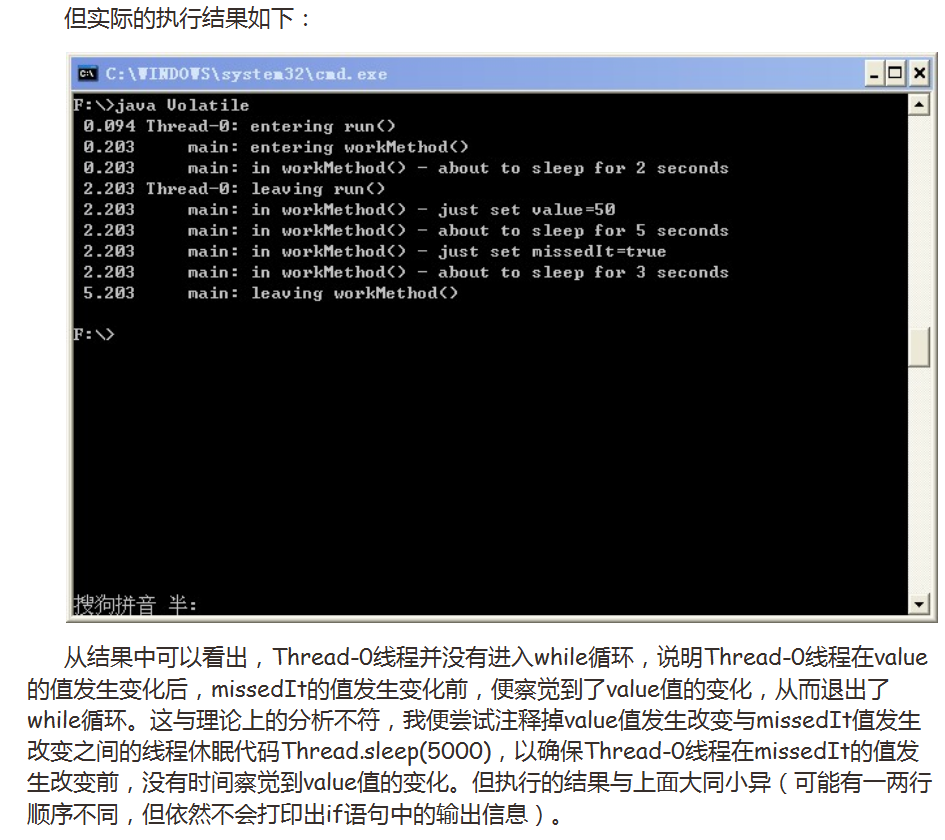






结果

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



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

##### 使用synchronized获取互斥锁的几点说明

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。

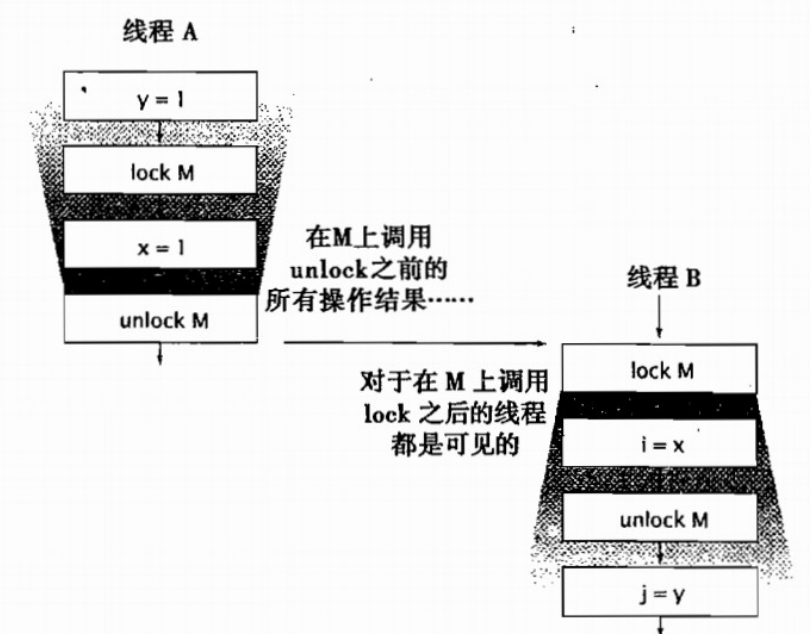
7、类级别锁被特定类的所有示例共享，它用于控制对static成员变量以及static方法的并发访问。具体用法与对象级别锁相似。

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

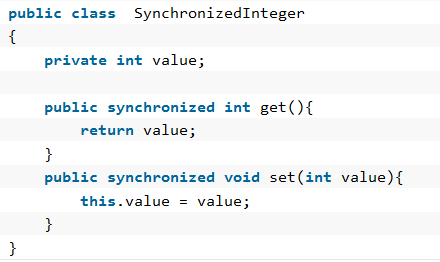
##### 图文讲述同步的另一个重要功能：内存可见性

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



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

##### 并发编程中实现内存可见的两种方法比较：加锁和volatile变量

1、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。

2、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。

3、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。

4、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：

1、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。

2、该变量没有包含在具有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。**

##### 多线程环境中安全使用集合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，也有可能会带来死锁。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 synchronizedCollention(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。



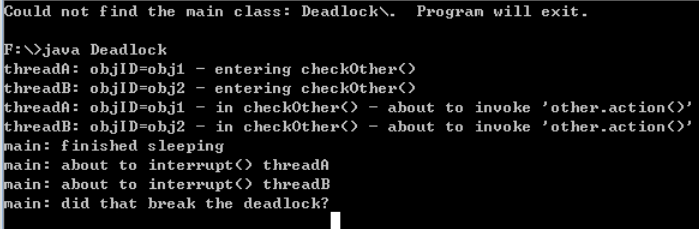
这里需要注意的是：在Java语言中，大部分的线程安全类都是相对线程安全的，它能保证对这个对象单独的操作时线程安全的，我们在调用的时候不需要额外的保障措施，但是对于一些特定的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。例如Vector、HashTable、Collections的synchronizedXxxx（）方法包装的集合等

##### 死锁

线程A当前持有互斥所锁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。







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：

1、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；

2、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；

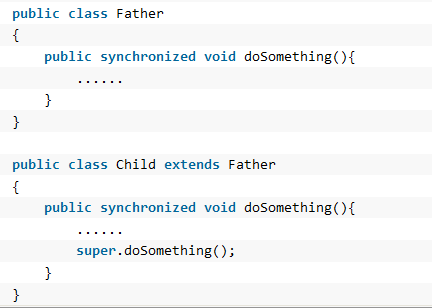
3、创建和使用一个大锁来代替若干小锁，并把这个锁用于互斥，而不是用作单个对象的对象级别锁；

##### 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，这个锁就被认为是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。

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Father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。

##### 使用wait/notify/notifyAll实现线程间通信的几点重要说明

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

1、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。

2、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，的如果调用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。

3、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。

4、wait（long）和wait（long,int）

显然，这两个方法是设置等待超时时间的，后者在超值时间上加上ns，精度也难以达到，因此，该方法很少使用。对于前者，如果在等待线程接到通知或被中断之前，已经超过了指定的毫秒数，则它通过竞争重新获得锁，并从wait（long）返回。另外，需要知道，如果设置了超时时间，当wait（）返回时，我们不能确定它是因为接到了通知还是因为超时而返回的，因为wait（）方法不会返回任何相关的信息。但一般可以通过设置标志位来判断，在notify之前改变标志位的值，在wait（）方法后读取该标志位的值来判断，当然为了保证notify不被遗漏，我们还需要另外一个标志位来循环判断是否调用wait（）方法。

**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。**

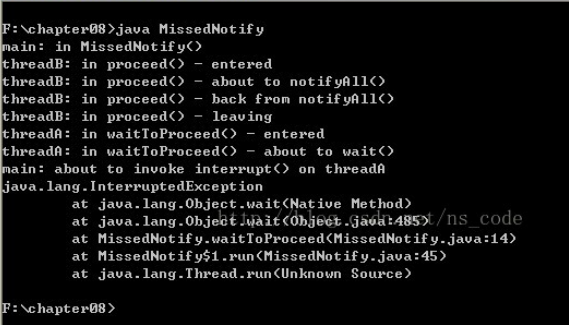
**优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用wait（）方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了synchronized代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。**

##### 线程间通信中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。





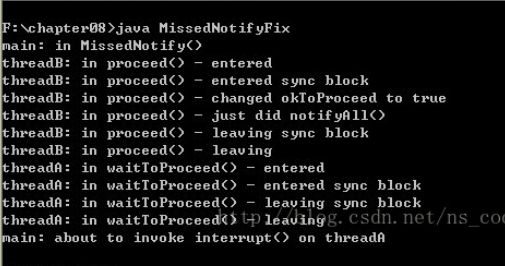


分析：由于threadB在执行mn.proceed（）之前只休眠了500ms，而threadA在执行mn.waitToProceed（）之前休眠了1000ms，因此，threadB会先苏醒，继而执行mn.proceed（），获取到proceedLock的对象锁，继而执行其中的notifyAll（），当退出proceed（）方法中的synchronized代码块时，threadA才有机会获取proceedLock的对象锁，继而执行其中的wait（）方法，但此时notifyAll（）方法已经执行完毕，threadA便漏掉了threadB的通知，便会阻塞下去。后面主线程休眠10秒后，尝试中断threadA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

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







 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。**

##### 线程间通信中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





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

**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。**

##### 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。

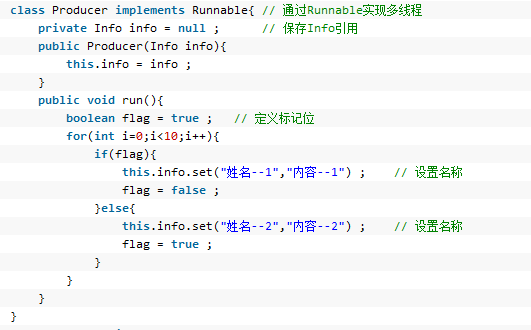
由于本程序中牵扯到线程运行的不确定性，因此可能会出现以下问题：

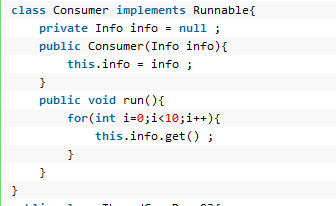
1、假设生产者线程刚向数据存储空间添加了数据的名称，还没有加入该信息的内容，程序就切换到了消费者线程，消费者线程将把信息的名称和上一个信息的内容联系在一起；

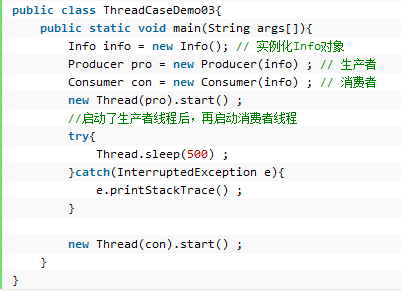
2、生产者生产了若干次数据，消费者才开始取数据，或者是，消费者取完一次数据后，还没等生产者放入新的数据，又重复取出了已取过的数据。

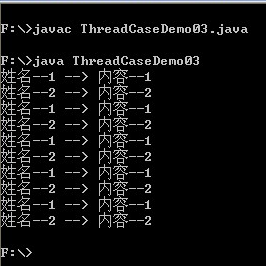
问题1很明显要靠同步来解决，问题2则需要线程间通信，生产者线程放入数据后，通知消费者线程取出数据，消费者线程取出数据后，通知生产者线程生产数据，这里用wait/notify机制来实现。











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

##### 深入Java内存模型——happen-before规则及其对DCL的分析

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。

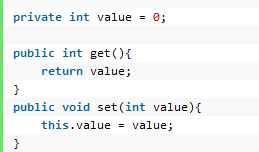
    7、对象终结规则：一个对象的初始化完成（构造函数执行结束）happen—before它的finalize（）方法的开始。

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？

1、**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？**



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

2、**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？**



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。

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）。



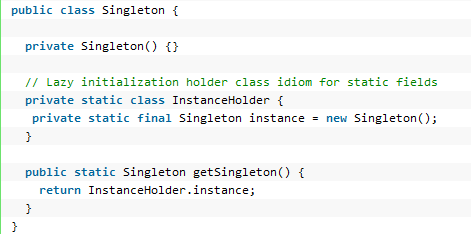
这里得到单一的instance实例是没有问题的，问题的关键在于尽管得到了Singleton的正确引用，但是却有可能访问到其成员变量的不正确值。具体来说Singleton.getInstance().getSomeField()有可能返回someField的默认值0。如果程序行为正确的话，这应当是不可能发生的事，因为在构造函数里设置的someField的值不可能为0。为也说明这种情况理论上有可能发生，我们只需要说明语句(1)和语句(7)并不存在happen-before关系。

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。

**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



2、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。

##### 深入Java内存模型—内存操作规则总结主内存与工作内存

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中操作来完成主内存与工作内存之间交互的实现细节：

1、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。

2、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。

3、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。

4、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。

5、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。

6、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。

7、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。

8、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：

1、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。

2、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。

3、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。

4、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。

5、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操作可以被同一个条线程重复执行多次，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。

6、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。

7、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。

8、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：

1、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**

2、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，但允许迅疾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吧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。

##### 并发新特性—Executor框架与线程池

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.cocurrent 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。

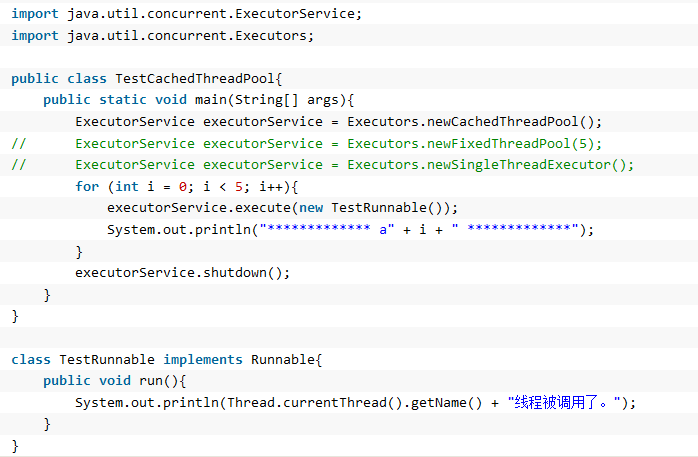
Executor接口中之定义了一个方法execute（Runnable command），该方法接收一个Runable实例，它用来执行一个任务，任务即一个实现了Runnable接口的类。ExecutorService接口继承自Executor接口，它提供了更丰富的实现多线程的方法，比如，ExecutorService提供了关闭自己的方法，以及可为跟踪一个或多个异步任务执行状况而生成 Future 的方法。 可以调用ExecutorService的shutdown（）方法来平滑地关闭 ExecutorService，调用该方法后，将导致ExecutorService停止接受任何新的任务且等待已经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了的任务，当素有已经提交了的任务执行完后，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。

一般来说，CachedTheadPool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。

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





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

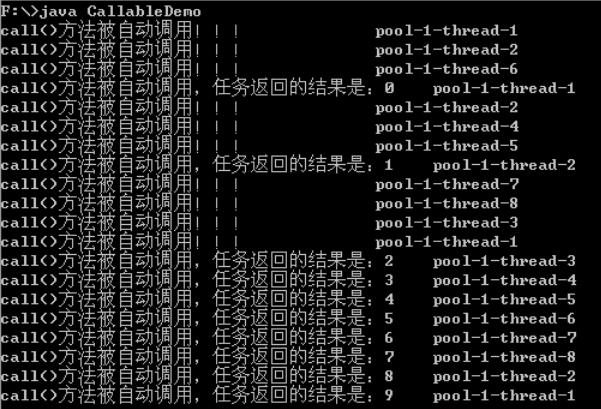
在Java 5之后，任务分两类：一类是实现了Runnable接口的类，一类是实现了Callable接口的类。两者都可以被ExecutorService执行，但是Runnable任务没有返回值，而Callable任务有返回值。并且Callable的call()方法只能通过ExecutorService的submit(Callable<T> task) 方法来执行，并且返回一个 <T>Future<T>，是表示任务等待完成的 Future。

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。







从结果中可以同样可以看出，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。