### 21.1.1 更快的执行

并发通常是提高运行在**单处理器**上的程序的性能。

表面上看来，将程序的所有部分当作单个任务运行好像是开销更小一点，并且可以节省上下文切换的代价

使这个问题变得有点不同的是阻塞。

在单处理器系统中的性能提高的常见示例是事件驱动的编程

### 21.2.1 Defining tasks

This call to the static method Thread.yield() inside run() is a suggestion to the thread scheduler(the part of the Java threading mechanism that moves the CPU from one thread to the next) that says, “I’ve done the important parts of my cycle and this would be a good time to switch to another task for a while”

|  |
| --- |
| public static void main(String args[]){  Thread t = new Thread(new LiftOff());  t.start();  System.out.println(“Waiting for lift off”);  } |

即使LiftOff中的run方法里包含有许多的输出语句，但往往是后面的”Waiting for lift off”先输出

### 21.2.3 Using Executors

Java SE5 java.util.concurrency executors simplify concurrent programming by managing Thread objects for you. Executors provide a layer of indirection between a client and the execution of a task; Executors allow you to manage the execution of asynchronous tasks without having to explicitly manage the lifecycle of threads. Executors are the preferred method for starting tasks in Java SE5/6.

An ExecutorService (an Executor with a service liftcycle-e.g., shutdown) knows how to build the appropriate context to execute Runnable objects. The CashedThreadPool creates one thread per task.

ExecutorService exec = Executors.newCachedThreadPool();

With the FixedThreadPool, you do expensive thread allocation once, up front and you thus limit the number of threads.

A CachedThreadPool will generally create as many threads as it needs during the execution of a program and then will stop creating new threads as it recycles the old ones, so it’s a reasonable first choice as an Executor. Only if this approach causes problems do you need to switch to a FixedThreadPool.

A SingleThreadExecutor is like a FixedThreadPool with a size of one thread. This is useful for anything you want to run in another thread continually(a long-lived task), such as a task that listens to incoming socket connections. It is also handy for short tasks that you want to run in a thread. For example, small tasks that update a local or remote log, or for an event dispatching thread.

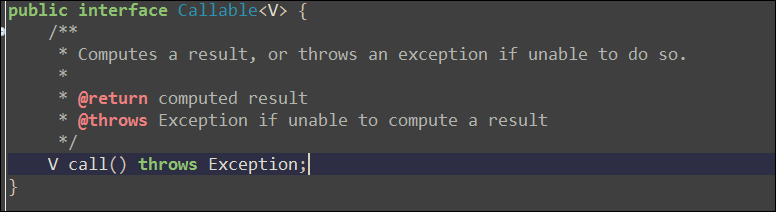
If more that one task is submitted to a SingleThreadExecutor, the tasks will be queued and eash task will run to completion before the next task is begun, all using the same thread. In the following example, you’ll see each task completed, in the order in which it was submitted, before the next one is begun. Thus, as singleThreadExecutor serializes the tasks that are submitted to it, and maintains its own (hidden) queue of pending tasks.

As another example, suppose you have a number of threads running tasks that use the file system. You can run these tasks with a SingleThreadExecutor to ensure that only one task at a time is running from any thread. This way , you don’t need to deal with synchronizing on the shared resource ( and you won’t clobber敲打 the file system in the meantime同时).

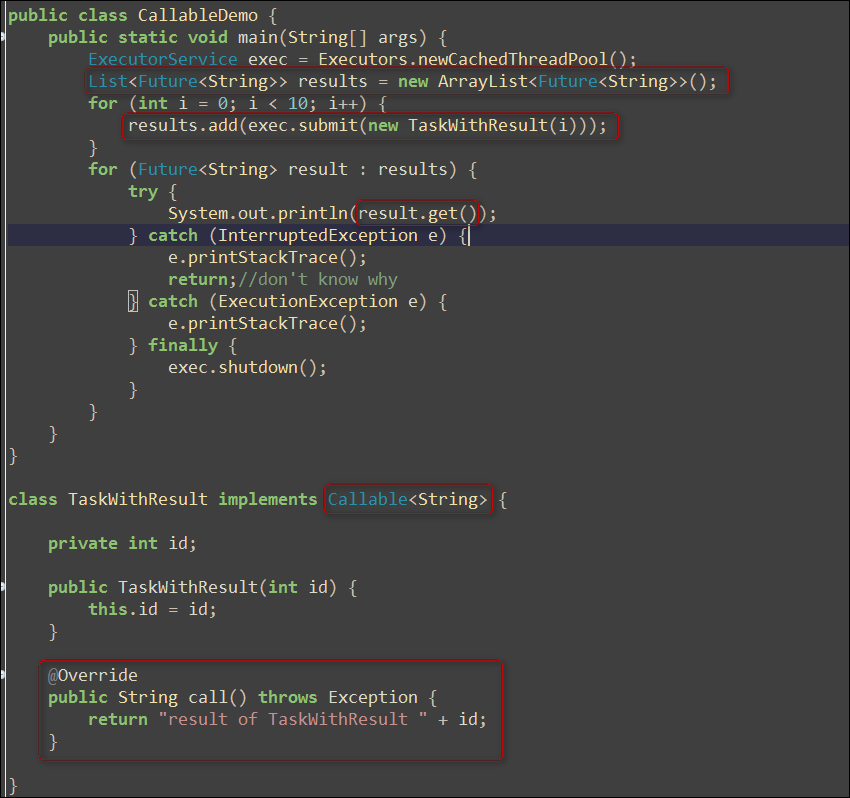
### Producing return values from tasks

A runnable is a separate task that performs work, but it doesn’t return a value. If you want the task to produce a value when it’s done, you can implement the Callable interface rather than the Runnable interface. Callable, introduced in Java SE5, is a generic with a type parameter representing the return value from the method call() (instead of run()), and must be invoked using an ExecutorService submit() method. Here’s a simple example:

The interface of Callable:



Demo:



### Sleeping

The call sleep() can throw an InterruptedException, and you can see that this is caught in run(). Because exceptions won’t propagate传送 across threads back to main(), you must locally handle any exceptions that arise within a task.

JavaSE5 introduced the more explicit version of sleep() as part of the TimeUnit can also be used to perform conversions.



### Priority

However, this doesn’t mean that threads with lower priority aren’t run (so you can’t get deadlocked because of priorities). Lower-priority threads just tend to run less often.

The vast(广阔的；巨大的；大量的；巨额的) majority(多数) of the time all threads should run at the default priority. Trying to manipulate thread priorities is usually a mistake.

优先级的数字越大，优先级越高，最小为1

Although the JDK has 10 priority levels, this doesn’t map well to many operating systems. For example, Windows has 7 priority levels that are not fixed, so the mapping is indeterminate. Sun’s Solaris has 231 levels. The only portable approach is to stick to MAX\_PRIORITY, NORM\_PRIORITY, and MIN\_PRIORITY when you’re adjusting priority levels

### Yielding

This hint (and it is a hint-there’s no guarantee your implementation will listen to it) takes the form of the yield() method. When you call yield(), you are suggesting that other threads of the same priority might be run

In general, however, you can’t rely on yield() for any serious control or tuning of your application. Indeed, yield() is often used incorrectly.

### Daemon threads

A “daemon” thread is intended to provide a general service in the background as long as the program is running, but is not part of the essence(本质，实质；精华；香精) of the program. Thus, when all of the non-daemon threads complete, the program is terminated, killing all daemon threads in the process. Conversely(相反地), if there are any non-daemon threads still running, the program doesn’t terminate. There is, for instance, a non-daemon thread that runs main().

You must set the thread to be a daemon by calling setDaemon() before it is started.

如果一个线程是Daemon线程，那么它创建的任何线程将被自动设置成后台线程。

一旦main()退出，JVM就会立即关闭所有的后台线程，所以它们几乎不是一种好的思想。非后台的Executor通常是一种更好的方式，因为Executor控制的所有任务可以同时被关闭。正如你将要在本章稍后看到的，在这种情况下，关闭将以有序的方式执行。

### Coding variations

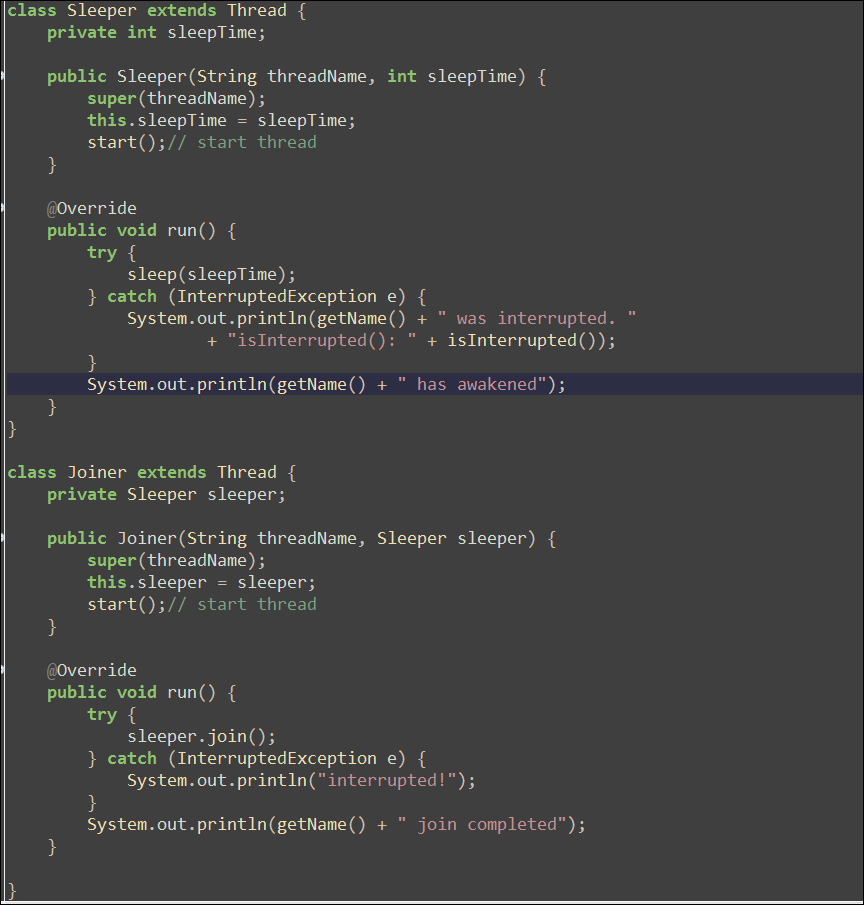
Notice the start() is called within the constructor. This example is quite simple and therefore probably safe, but you should be aware that starting threads inside a constructor can be quite problematic, because another task might start executing before the constructor has completed, which means the task may be able to access the object in an unstable state. This is yet another reason to prefer the use of Executors to the explicit creation of Thread objects.

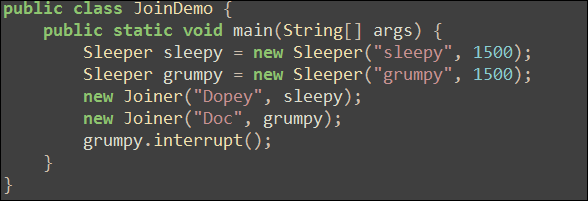
### Joining a thread

One thread may call join() on another thread to wait for the second thread to complete before proceeding. If a thread calls t.join() on another thread t, then the calling thread is suspended until the target thread t finishes ( when t.isAlive() is false).

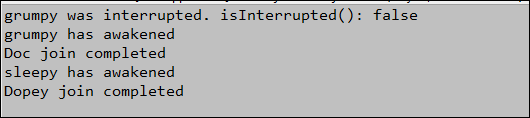
You may also call join() with a timeout argument ( in either MICROSECONDS or milliseconds and nanoseconds) so that if the target thread doesn’t finish in that period of time, the call to join() returns anyway.

The call to join() may be aborted by calling interrupt() on the calling thread, so a try-catch clause is required.





Output is :



A sleeper is a thread that goes to sleep for a time specified in its constructor. In run(), the call to sleep() may terminate when the time expires, but it may also be interrupted. Inside the catch clause, the interruption is reported, along with the value of isInterrupted(). When another thread calls interrupt() on this thread, a flag is set to indicate that the thread has been interrupted. However, this flag is cleared when the exception is caught, so the result will always be false inside the catch clause. The flag is used for other situations where a thread may examine its interrupted state apart from the exception.

A joiner is a task that waits for a Sleeper to wake up by calling join() on the Sleeper object. In main(), each Sleeper has a Joiner, and you can see in the output that if the Sleeper either is interrupted or ends normally, the Joiner completes in conjuction with the Sleeper.

Note that the Java SE5 java.util.concurrent libraries contain tools such as CyclicBarries(demonstrated later in this capter) that may be more appropriate than join(), which was part of the original threading library.

### Thread groups

线程组持有一个线程集合。线程组的价值可以引用Joshua Bloch的话来总结，它在Sun时是软件架构师，订正并极大地改善了JDK1.2的Java集合类库：

“最好把线程组看成是一次不成功的尝试，你只要忽略它就可以了”

### Catching exceptions

### Sharing resources

基本上所有的并发模式在解决线程冲突问题的时候，都是采用序列化访问共享资源的方案。

|  |
| --- |
| synchronized void f(){}  synchronized void g(){} |

If f() is called for an object by one task, a different task cannot call f() or g() for the same object until f() is completed and releases the lock. Thus, there is a single lock that is shared by all the synchronized methods of a particular object, and this lock can be used to prevent object memory from being written by more than one task at a time.

Note that it’s especially important to make field private when working with concurrency; otherwise the synchronized keyword cannot prevent another task from accessing a field directly, and thus producing collections.

One task may acquire an object’s lock multiple times. This happens if one method calls a second method on the same object, which in turn calls another method on the same object, etc. The JVM keeps track of the number of times the object has been locked. If the object is unlocked, it has a count of zero. As a task acquires another lock on the same object, the count is incremented. Naturally, multiple lock acquisition is only allowed for the task that acquired the lock in the first place. Each time the task leaves a synchronized method, the count is decremented, until the count goes to zero, releasing the lock entirely for use by other tasks.

There’s also a single lock per class (as part of the Class object for the class), so that synchronized static methods can lock each other out from simultaneous(同时的；联立的；同时发生的) access of static data on a class-wide basis.

When should you synchronize? Apply Brian’s Rule of synchronization:

“If you are writing a variable that might next be read by another thread, or reading a variable that might have last been writtern by another thread, you must use synchronization, and further, both the reader and writer must synchronize using the same monitor lock.”

This is an important point: Every method that accesses a critical shared resource must be synchronized or it won’t work right

### 使用的显式的Lock对象

Lock对象必须被显式地创建，锁定和释放。因此，它与内建的锁形式相比，代码缺乏优雅性。但是，对于解决某些问题来说，它更加灵活。

尽管try-finally所需的代码比synchronized关键字要多，某些事物失败了，那么就会抛出一个异常。但是你没有机会去做任何清理的工作，以维护系统使其处于良好状态。有了显式的Lock对象，你就可以使用Finally子句将系统维护在正确的状态了。

大体上，当你使用Synchroized关键字时，需要写的代码量更少，并且用户错误出现的可能性也会降低，因此通常只有在解决特殊问题时，才使用显式的Lock对象。例如，用synchronized关键字不能尝试着获取锁且最终取锁会失败，或者尝试着获取锁一段时间，然后放弃它，要实现这些，你必须使用concurrncy类库。

### 原子性和易变性

原子性可以应用于除了long和double之外的所有基本类型之上的“简单操作”。对于读取和写入除long和double之外的基本类型变量这样的操作，可以保证它们会被当作不可分（原子）的操作来操作内存。但是JVM可以将64位(long和double变量)的读取和写入当作两个分离的32操作来执行，这就产生了在一个读取和写入操作中音发生上下文切换，从而导致不同的任务可以看到不正确结果的可能性（这有时被称为字撕裂，因为你可能会看到部分被修改过的数值）。但是当你定义long或double变量时，如果使用volatile关键字，就会获得（简单的同仁与返回操作的）原子性（注意，在Java SE5之前，这个关键字一直没能正确地工作）

Volatile关键字还确保了应用中的可视性。如果你将一个域声明为volatile的，那么只要对这个域产生了写操作，那么所有的读操作就都可以看到这个修改。即使使用了本地缓存，情况也确定如此，volatile域会立即被写入到主存中，而读取操作就发生在主存中。

如果一个域完全由synchorized方法或语句来防护，那就不必将其设置为volatile的。

Java递增/递减操作不是原子性的，并且涉及一个读操作和一个写操作，所以即使是在这么简单的操作中，也为产生线程问题留下了空间。

如果你将一个域定义为volatile，那么它就会告诉编译器不要执行任何移除读取和写入操作的优化，这些操作的目的是用线程中的局部变量维护对这个域的精确同步。

### 原子类

Java SE5引入了诸如AtomicInteger, AtomicLong, AtomicReference等特殊的原子性变量类，它们操作了下面形式的原子性条件更新操作

Boolean compareAndSet(expectedValue, updateValue);

### 临界区

|  |
| --- |
| synchronized(syncObject){  //something need to be done  } |

这也被称为同步控制块；在进入此段代码前，必须得到syncObject对象的锁。如果其它线程已经得到这个锁，那么就得等到锁被释放以后，才能进入临界区。

通过使用同步控制块，而不是对整个方法进行同步控制，可以使多个任务访问对象的时间性能得到显著提高。

### 在其它对象上同步

如果获得synchronized块上的锁，那么该对象其它的synchronized方法和临界区就不能被调用了。因此，如果在this上同步，临界区的效果就会直接缩小在同步的范围内。

### 线程本地存储

防止任务在共享资源上产生冲突的第二种方式是根除对变量的共享。线程本地存储是一种自动化机制，可以为使用相同的变量的每个不同的线程都创建不同的存储。因此，如果你有5个线程都使用变量所表示的对象，那线程本地存储变会生成5个用于X的不同的存储块。主要是：它们使得你可以将状态与线程关联起来。

所以使用ThreadLocal定义的变量可以不需要对读写这个变量的方法加synchorized，因为ThreadLocal保证不会出现竞争条件。

### 终结任务

ExecutorService.awaitTermination()等待每个任务结束，如果所有的任务在超时时间达到之前全部结束，则返回true，否则返回false，表示不是所有的任务都已经结束了。尽管这会导致每个任务都退出其run()方法，并因此作为任务而终止，但是Entrance对象仍旧是有效的。

一个线程可以处于以下四种状态之一：

* 新建(new)
* 就绪(Runnable)
* 阻塞(Blocked)
* 死亡(Dead)

可能你已经注意到了，concurrency类库似乎在避免对Thread对象的直接操作，转而尽量通过Executor来执行所有的操作。如果你在Executor上调用shutdownNow()，那么它将发送一个interrupt()调用给它启动的所有的线程。这么做是有意义的，因为当你完成工程中的某个部分或者整个程序时，通常会希望同时关闭某个特定Executor的所有任务。然而，你也可能只希望中断某个单一任务。如果使用Executor，那么通过调用submit()而不是executor()来启动任务，就可以持有该任务的上下文。Submit()将返回一个泛型的Future<?>，其中有一个未修饰的参数，因为你永远都不会在其上调用get()—持有这种Future的关键在于你可以在其上调用cancel()，并因此可以使用它来中断某个特定任务。如果你将true传递给cancel()，那么它就会拥有在该线程上调用interrupt()以停止这个线程的权限。因此，cancel()是一种中断由Executor启动的单个线程的方式。

你能够中断对sleep()的调用（或者任何要求抛出InterruptedException的调用）。但是你不能中断正在试图获取synchronized锁或者试图执行I/O操作的线程。

### 检查中断

清除中断状态可以确保并发结构不会就某个任务被中断这个问题通知你两次，你可以经由单一的InterruptedException或单一的成功的Thread.interrupted()测试来得到这种通知。如果想要再次检查以了解是否被中断，则可以在调用Thread.interrupted()时将结果存储起来。

### 线程之间的协作

调用sleep()的时候锁并没有被释放，调用yield()也属于这种情况，理解这一点很重要。另一个方面，当一个任务在方法里遇到了对wait()的调用的时候，线程的执行被挂起，对象上的锁被释放。

Wait(), notify()和notifyAll()也有一个比较特殊的方面，那就是这些方法是基类Object的一部分，而不是属于Thread中的一部分。尽管开始看起来有点奇怪---仅仅针对线程的功能却作为通用基类的一部分实现，不过这是有道理的，因为这些方法操作的锁也是所有对象的一部分。所以，你可以把wait()放进任何同步控制方法里，而不用考虑这个类是继承自Thread还是实现了Runnable接口。实际上，只能在同步控制方法或同步控制块里调用wait(), notify()和notifyAll()（因为不用操作锁，所以sleep()可以在非同步控制方法里调用）。如果在非同步控制方法里调用这些方法，程序能通过编译，但运行的时候，将得到IllegalMonitorStateException异常，并伴随着一些含糊的信息，比如“当前线程不是拥有者”。消息的意思是，调用wait(), notify()和notifyAll()的任务在调用这些方法前必须“拥有”（获取）对象的锁。

当你调用某个ExcecutorService的shutdownNow()时，它会调用所有由它控制的线程的interrupte()。

前面的示例强调你必须用一个检查感兴趣的条件的while循环包围wait()。这很重要，因为：

你可能有多个任务出于相同的原因在等待同一个锁，而第一个唤醒任务可能会改变这种状态（即使你没有这么做，有人也会通过继承你的类去这么做）。如果属于这种情况，那么这个任务应该被再次挂起，直至其感兴趣的条件发生变化。

在这个任务从其wait()中被唤醒的时刻，有可能会有某个其它的任务已经做出了改变，从而使得这个任务在此时不能执行，或者执行其操作已经显得无关紧要。此时，应该通过再次调用wait()来将其重新挂起

也有可能某些任务出于不同的原因在等待你的对象上的锁（在这种情况下必须使用notifyAll()）。在这种情况下，你需要检查是否已经由正确的原因唤醒，如果不是，就再次调用wait()

因此，其本质就是要检查所感兴趣的特定条件，并在条件不满足的情况下返回到wait()中，惯用的方法就是使用while来编写这种代码

### 错失的信号

|  |
| --- |
| T1:  Synchronized(sharedMonitor){  <setup condition for T2>  sharedMonitor.notify();  }  T2:  While(someconditions){  //Point 1  Synchronized(sharedMonitor){  sharedMonitor.wait();//另外wait()方法也是在Object中定义的  }  } |

<setup condition for T2>是防止T2调用wait()的一个动作，当然前提是T2还没有调用wait()。

假设T2对someConditions求值并发现为true。在Point1，线程调用器可能切换到了T1。而T1将执行其设置，然后调用notify()。当T2得以继续执行时，此时对于T2来说，时机已经太晚了，以至于不能意识到这个条件已经发生了变化，因此盲目进入wait()。此时notify()将错失，而T2也将无法限地等待这个已经发送过的信号，从而生产死锁。

使用notify()而不是notifyAll()是一种优化。使用notify()时，在众多等待同一个锁的任务中只有一个会被唤醒，因此如果你希望使用notify()，就必须保证被唤醒的是恰当的任务。

在有关java的线程机制的讨论中，有一个令人困惑的描述：notifyAll()将唤醒“所有正在等待的任务”。这是否意味着在程序中的任何地方，任何处于wait()状态中的任务都将被任何可notifyAll的调用唤醒呢？在下面的示例中，与Task2相关的代码说明了情况并非如此—事实上，当notifyAll()因某个特定锁而被调用时，只有等待这个锁的任务才会被唤醒。

## ITCAST视频学习

进程：是一个正在执行中的程序。

每一个进程执行都有一个执行顺序。该顺序是一个执行路径，或者叫一个控制单元

线程：就是进程中的一个独立的控制单元，线程在控制着进程的执行。

一个进程中至少有一个线程

运行一个程序，JVM不单只启动一个主线程，同时至少也启动了垃圾回收的线程.

如果同步函数被静态修饰后，使用的锁是什么呢？

通过验证，发现不再是this。因为静态方法中也不可以定义this。

静态内存时，内存中没有本类对象，但是一定有该类对应的字节码文件对象类名.class 该对象的类型是Class

|  |
| --- |
| Public class Ticket implements Runnable  private static int ticks = 100;  public void run(){  synchronized(Ticket.class){  //do something  }  } |

改错题：

|  |
| --- |
| **public** **class** CorrectThreadCodes {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ThreadDemo t = **new** ThreadDemo();  Thread thread1 = **new** Thread(t);  thread1.start();  // TimeUnit.SECONDS.sleep(1);  // System.out.println("sleep over in main()");  t.flag = **false**;// code1  Thread thread2 = **new** Thread(t);  thread2.start();  }  }  **class** ThreadDemo **implements** Runnable {  **boolean** flag = **true**;  @Override  **public** **void** run() {  **if** (flag) {  **while** (**true**) {  **try** {  TimeUnit.MILLISECONDS.sleep(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.out.println("from flag = true");  }  } **else** {  **while** (**true**) {  **try** {  TimeUnit.MILLISECONDS.sleep(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.out.println("from flag = false");  }  }  }  } |

这个程序的所有输出都是为"from flag = false"，这是因为虽然有code1处将flag置为false, 但主线程在thread1还没有启动完的时候就已经执行了code1部分的代码，从而有这样的输出。 可以在code1前面加个sleep就可以达到所要的目的了.

所谓的懒汉式单例实现模式：

|  |
| --- |
| **public** **class** LazySingleDemo {  **private** **static** LazySingleDemo instance = **new** LazySingleDemo();  **private** LazySingleDemo() {  }  **public** **static** LazySingleDemo getInstance() {  **if** (instance == **null**) {// 此处的代码只为增加性能，尽量少地检查锁  **synchronized** (LazySingleDemo.**class**) {// 静态变量只能以.class来加锁，不能使用this  **if** (instance == **null**) {  instance = **new** LazySingleDemo();  }  }  }  **return** instance;  }  } |