# 认识多线程

并发编程的意义：并发编程可以使程序执行速度得到极大的提高，或者为设计某些类型的程序提供更简单的模型。

并发编程的几个方面内容：

并发编程令人困惑的一个重要原因是：使用并发时需要解决的问题有多个，而实现并发的方式也有多种，并且这两者之间没有明显的映射关系。

解决办法：我们必须理解所有的这些问题和特例，以便有效的使用并发。

具体办法

* 更快的执行

如果你想让一个程序运行的更快，那么可以将其断开为多个片段，在单独的处理器上运行每个片段。并发是用于多处理器编程的基本工具。当前速度的提高是以多核处理器的形式而不是更快的芯片的形式出现的。为了使程序运行的更快，你必须学习如何利用这些额外的处理器，而这正是并发赋予你的能力。

但是，并发通常是提高运行在单处理器上的程序的性能。

* 改进代码设计

Java的线程机制是抢占式的，这表示调度机制会周期性地中断线程，将上下文切换到另一个线程，从而为每个线程都提供时间片段，使得每个线程都分配到数量合理地时间去驱动它的任务。在协作式系统中，每个任务都会自动的放弃控制，这要求程序员有意识的在每个任务中插入让步语句。协作系统的优势是双重的：上下文切换的开销比抢占式要低廉的多，可以同时执行的线程数量理论上没有限制。当你处理大量的仿真元素时，这是一种理想的解决方案。但是注意，某些协作式系统并未设计为可以在多个处理器之间分配任务，这可能会非常有限。

并发需要付出代价，但这些代价与在程序设计、资源负载均衡以及用户方便方面的改进相比，就显得微不足道。通常，线程能够使我们创建更加松耦合的设计。

## 线程的基本机制

并发编程使得我们可以将程序划分为多个分离的、独立的任务。通过使用多线程机制，这些独立任务中的每一个都将由执行程序来驱动。一个线程就是进程中的一个单一的顺序控制流，因此，单个进程可以拥有多个并发执行的任务，但是你的程序使得每个任务都好像有其自己的CPU一样。其底层机制是切分CPU时间，但我们通常不需要考虑他。

线程模型为编程带来了便利。它简化了在单一程序中同时多个操作的处理。在使用线程时，CPU将轮流给每个任务分配其占用时间。每个任务都觉得自己在一直占用CPU，但事实上CPU时间是划分片段分配给了所有任务（也有可能是运行在多个CPU之上）。线程的一大好处是可以使你从这个层次抽身出来，即代码不需要知道它是运行在一个还是多个CPU上。所以，使用线程机制是一个建立透明的、可扩展的程序的方法，如果程序运行速度太慢，为机器增添一个CPU就很容易的增加程序运行的速度。多个任务、多个线程是使用多个处理器系统的最合理方式。

* 定义任务

线程可以驱动任务，因此需要一种描述任务的方式，这可以由Runnable接口来提供。要想定义任务，只需要实现Runnable接口并编写run()方法，使得该任务可以执行你的命令。

任务的run()方法通常会有某种形式的循环，使得任务一直运行下去直到不再需要，所以要设定跳出循环的条件。在run()方法中对静态方法Thread.yield()的调用是对线程调度器的一种建议，线程调度器是Java多线程机制的一部分，可以将CPU从一个线程转移到另一个线程。它声明了，我们已经执行完生命周期中最重要的一部分，此刻正是切换给其他任务执行的大好时机。

[

参考代码：runnable/LiftOff.java, runnable/MainThread.java

执行结果：#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),

]

Runnable结论：从Runnable导出一个类，他必须实现run()方法，他没有任何内在线程的能力。要实现线程的行为，必须显式的给一个任务赋予run()方法。

* Thread类

将Runnable对象转变为一个工作任务的方式是把它提交给一个Thread构造器。

[

参考代码：thread/BasicThreads.java、thread/LiftOff.java

执行结果：任务开始

#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),

]

Thread构造器只需要一个Runnable对象。调用start()方法为该线程执行提供必须的初始化操作，然后调用Runnable的run()方法，以便在这个线程中启动任务。我们看到输出语句先输出了，任务的语句后输出了。这表明start()语句直接返回了，它实际上只是产生了对LiftOff.run()方法的调用，并且这个方法还没有完成，但是由于run()方法是由不同的线程执行的，所以main()方法中的任务还可以继续执行。因此，程序会同时运行两个方法。

[

参考代码2：thread/MoreBasicThreads.java

执行结果：

Waiting for LiftOff

#0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8), #3(8), #4(8), #0(7), #1(7), #2(7), #3(7), #4(7), #0(6), #1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5), #3(5), #4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3), #2(3), #3(3), #4(3), #0(2), #1(2), #2(2), #3(2), #4(2), #0(1), #1(1), #2(1), #3(1), #4(1), #0(Liftoff!), #1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),

输出结果说明不同任务的

]

输出结果说明不同任务的执行被混在了一起。这种交换是由线程调度器自动控制的。如果你有多个处理器，线程调度器就会在这些处理器之间分发线程。当main()创建Thread对象时，它并没有捕获对这些对象的引用。在使用普通对象时，对于垃圾回收器是一种公平的游戏，但是在使用Thread时，情况就不同。每个Thread都注册了自己，存在一个对它的引用，而且在任务退出run()死亡之前，垃圾回收器无法清除它。

* 使用Executor

Java SE5的java.util.concurrent包中的执行器(Executor)将为你管理Thread对象，简化了并发编程。Executor在客户端和任务之间建立了一个中间层；与客户端直接执行任务不同，这个中介将直接执行任务。Executor允许你管理异步任务的执行，而无需显式的管理线程和生命周期。我们可以使用Executor来替代显式的创建Thread对象。

[

参考代码: executor/CachedThreadPool.java、executor/LiftOff.java

执行结果：

#2(9), #1(9), #0(9), #2(8), #2(7), #2(6), #2(5), #1(8), #1(7), #2(4), #0(8), #2(3), #2(2), #1(6), #2(1), #0(7), #2(Liftoff!), #1(5), #0(6), #1(4), #0(5), #0(4), #1(3), #0(3), #1(2), #0(2), #1(1), #0(1), #1(Liftoff!), #0(Liftoff!),

]

Shutdown()方法的调用可以防止新任务被提交给这个Executor，当前线程将继续运行在shutdown()被提交之前提交的所有任务。

我们可以使用不同类型的Executor。

FixedThreadPool使用了有限的线程集来执行提交的任务，你可以一次性预先执行代价高昂的线程分配，也可以限制线程的数量。这可以节省时间，因为你不用为每个任务都固定的去创建线程。注意：在任何线程池中，现有线程在可能的情况下都会复用。CachedThreadPool在程序执行过程中通常会创建与所需要数量相同的线程，然后在它回收旧线程时停止创建新的线程，因此它是首选。只有当这种方式引发问题时才需要切换到FixedThreadPool。

[

参考代码：executor/FixedThreadPool.java、executor/LiftOff.java

]

SingleThreadExecutor就像是线程数量为1的FixedThreadPool。如果向其中提交了多个任务，那么这些任务将排队，每个任务都会在下一个任务开始之前结束，所有的任务将使用相同的线程。

[

参考代码：executor/SingleThreadPool.java、executor/LiftOff.java

执行结果：

#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!), #1(9), #1(8), #1(7), #1(6), #1(5), #1(4), #1(3), #1(2), #1(1), #1(Liftoff!),

#2(9), #2(8), #2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1), #2(Liftoff!),

]

假如你有大量的任务将使用文件系统。你可以运用SingleThreadExecutor来运行这些线程，以确保任意时刻在任何线程中都只要唯一的任务在运行。这种方式你不需要在共享资源上同步。

* 从任务中产生返回值

Runnable是执行工作的独立任务，但是他不返回任何值。如果你希望在任务执行完成时能够返回值，那么可以实现Callable接口。它是具有类型参数的泛型，它的类型参数表示的是从方法call()中返回的值，并且必须使用ExecutorService.submit()方法调用它。

[

参考代码: callable/TaskWithResult.java

执行结果：

任务执行完毕0

任务执行完毕1

任务执行完毕2

]

submit方法会产生Future对象，它用Callable返回结果的特定类型进行了参数化。可以使用isDone()方法查询Future对象是否完成。当任务完成时可以调用get()方法获取结果。

* 休眠

影响任务行为的一种简单方法是调用sleep()，这将使任务终止执行给定的时间。

[

参考代码：sleep/SleepTask.java、sleep/LiftOff.java

]

对sleep()的调用会抛出异常，并且可以看到，它在run()中被捕获。Java SE5中引入更显式版本的sleep()方法，作为TimeUnit类的一部分。这个新版本的sleep()方法允许你指定sleep延迟的时间单元，因此可以提供更好的可阅读性。TimeUnit还可以被用来执行转换。

* 优先级

线程的优先级将线程的重要性传递给调度器。尽管CPU处理线程集的顺序是不确定的，但是调度器将倾向于让优先权高的线程先执行。然而并不意味着优先级低的线程得不到执行。优先级低的线程仅仅意味着执行的频率较低。

在绝大多数时间所有的线程都应该以默认的优先级运行。试图操纵优先级并不提倡。

[

参考代码: priority/SimplePriorities.java

执行结果：

Thread[pool-1-thread-6,10,main]: 5

Thread[pool-1-thread-1,1,main]: 5

Thread[pool-1-thread-5,1,main]: 5

Thread[pool-1-thread-2,1,main]: 5

Thread[pool-1-thread-6,10,main]: 4

Thread[pool-1-thread-4,1,main]: 5

Thread[pool-1-thread-3,1,main]: 5

Thread[pool-1-thread-6,10,main]: 3

Thread[pool-1-thread-6,10,main]: 2

Thread[pool-1-thread-1,1,main]: 4

Thread[pool-1-thread-5,1,main]: 4

Thread[pool-1-thread-2,1,main]: 4

Thread[pool-1-thread-4,1,main]: 4

Thread[pool-1-thread-3,1,main]: 4

Thread[pool-1-thread-6,10,main]: 1

Thread[pool-1-thread-5,1,main]: 3

Thread[pool-1-thread-1,1,main]: 3

。。。

]

我们使用了大量的运算来测试，观察到优先级为MAX\_PRIORITY的线程被线程调度器优先选择。注意：JDK有10个优先等级，但是与大多数操作系统的映射不好。比如, windows有7个优先级且不固定，所以这种映射关系也是不确定的。

* 让步

如果你已经知道你的一次循环过程中的工作已经完成，就可以给线程调度机制一个暗示: 你的工作完成的差不多了，可以让别的线程使用CPU了。这个暗示将通过调用yield()来完成。注意，这只是一种暗示，没有任何机制保证它将会被采纳。当调用yield()时，你也是在建议具有相同优先级的其他线程可以运行。

* 后台线程

后台线程就是指在程序运行的时候在后台提供一种通用服务的线程，这种线程不是程序必须的一部分。因此，当所有的非后台线程结束时，程序也就终止了，同时会杀死进程中的所有后台线程。反过来说，只要有任何非后台线程还在运行，程序就不会被终止。比如main()就是一个非后台线程。

[

参考代码：daemon/SimpleDaemons.java

执行结果：

All daemons started

Thread[Thread-8,5,main] concurrency.SimpleDaemons@790d3283

Thread[Thread-5,5,main] concurrency.SimpleDaemons@7e00f258

Thread[Thread-4,5,main] concurrency.SimpleDaemons@ca4a1b4

Thread[Thread-0,5,main] concurrency.SimpleDaemons@fcc3aac

Thread[Thread-9,5,main] concurrency.SimpleDaemons@1c4b4746

。。。

]

必须在线程启动之前调用setDaemon()方法，才能把它设置为后台线程。

可以通过isDaemon()方法来确定线程是否是一个后台线程。如果是一个后台线程，那么它创建的任何线程都将被自动设置为后台线程。

[

参考代码：daemon/Daemon.java、daemon/Daemons.java

执行结果：

DaemonSpawn 0 started, d.isDaemon() = true, DaemonSpawn 1 started, DaemonSpawn 2 started, DaemonSpawn 3 started, DaemonSpawn 4 started。。。。

]

Daemon被设置为了后台线程，然后派生出很多子线程，这些线程并没有被显式的设置为后台模式，不过他们却是后台线程。

[

参考代码：daemon/DaemonsDontRunFinally.java

运行结果：ADaemon中的finally子句不会执行，但是如果注释掉main()中的setDaemon()方法，就会看到finally子句被执行。这是因为当最后一个非后台线程终止时，后台线程会突然终止。因此一旦main()方法退出，JVM就会立即关闭所有的后台线程。

]

* 编码的变体

定义一个线程任务除了直接实现Runnable接口，我们也可以直接从Thread继承。

[

参考代码: thread/SimpleThread.java

执行结果：

#2(5)

#1(5)

#2(4)

#1(4)

#2(3)

#1(3)

#2(2)

#1(2)

#1(1)

#2(1)

]

定义线程任务另外一种常用的方法是自管理的Runnable

[

参考代码：thread/SelfManaged.java

执行结果：

Thread-1(5), Thread-0(5), Thread-1(4), Thread-0(4), Thread-1(3), Thread-0(3), Thread-1(2), Thread-0(2), Thread-1(1), Thread-0(1),

]

注意：start()是在构造器中被调用的。但是应该意识到，在构造器中启动线程可能会变得有问题，因为另一个任务可能在构造器结束之前开始执行，这意味着该任务能够访问处于不稳定状态的对象。这也是我们优先选择Executor而不是显式的创建Thread的原因。

* 加入一个线程

一个线程可以在其他线程之上调用join()方法，其效果是等待一段时间直到第二个线程结束才继续执行。如果某个线程在另一个线程t上调用join()方法，此线程将会被挂起，直到目标线程t结束才恢复。也可以在调用join()时带上一个超时参数，这样如果目标线程在这段时期没有完成结束，join()方法总能返回。对join()方法的调用可以被中断，做法是在调用线程上调用interrupt()方法。

[

参考代码：join/Sleeper.java、join/Joiner.java、join/Joining.java

执行结果：

Grumpy was interrupted. isInterrupted(): false

Doc join completed

Sleepy has awakened

Dopey join completed

]

执行结果先输出了前两句，当Doc被执行时，此时Sleeper里的join()方法被挂起。休眠时间结束调用了interrupt()结束挂起之后线程又开始执行。

* 捕获异常

由于线程的本质特征，使得你不能捕获从线程中逃逸的异常。一旦异常逃出任务的run()方法，它就会向外传播到控制台，除非你采取特殊的步骤捕获这种错误的异常。在Java SE5之后，可以用Executor来解决这个问题。

[

参考代码： exception/ ExceptionThread.java

执行结果：

Exception in thread "pool-1-thread-1" java.lang.RuntimeException

at concurrency.ExceptionThread.run(ExceptionThread.java:11)

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)

at java.lang.Thread.run(Thread.java:745)

]

我们把调用语句加入到try-catch语句块中，产生的结果不会发生变化：未捕获异常。

为了能捕获异常，我们需要修改Executor产生线程的方式。Thread.UncaughtExceptionHandler是Java SE5中的新的接口，它允许你再每个Thread对象上都附着一个异常处理器。它的uncaughtException()会在线程因未捕获异常面临死亡时调用。

我们创建一个ThreadFactory，它将在每一个新创建的Thread对象上附着一个Thread.UncaughtExceptionHandler。

[

参考代码：exception/MyUncaughtExceptionHandler.java、exception/HandlerThreadFactory.java、exception/CaptureUncaughtException.java

执行结果：

创建一个新的线程

添加异常捕获结束

创建一个新的线程

添加异常捕获结束

自定义的异常捕获java.lang.RuntimeException

]

可以看到未捕获的异常是通过uncaughtException来捕获的。

如果你要在很多代码中使用相同的异常处理器，那么更简单的方法是在Thread类中设置一个静态域，并将这个处理器设置为默认的异常捕获处理器。

[

参考代码：exception/SettingDefaultHandler.java

]

注意：默认的异常处理器只有在线程未设置专有的异常处理器情况下才会被调用。

# 共享资源

每个线程每次只能做一件事情。但是多线程会在同时访问一个资源，比如会出现两个线程同时访问一个银行账户，向同一个打印机打印，改变同一个值等问题。

不正确的访问资源

我们先做一个实验：多个任务，一个任务产生一个偶数，其他的任务检验偶数的有效性。

[

参考代码：sync/simple/IntGenerator.java, sync/simple/EvenChecker.java，sync/simple/EvenGenerator.java

执行结果：

1537不是偶数

1541不是偶数

1539不是偶数

EvenChecker总是读取和测试IntGenerator的返回值。如果isCanceled()返回值为true, 则run()返回，这将告知test()中的Executor该任务完成了。任何EvenChecker任务都可以在与其关联的IntGenerator上调用cancel()，这将导致所有其他使用该IntGenerator的EvenChecker得到关闭。

]

一个任务可能在另一个任务执行第一个递增操作之后，但是没有执行第二个递增操作之前，调用next()方法。这将使这个值处于不恰当状态。为了证明这是可能发生的，test()方法创建了一组EvenChecker对象，以用来连续地读取并输出同一个EvenGenerator，并检测每个数值是否都是偶数。如果不是就报错终止。

这个程序最终会失败终止，因为每个EvenChecker任务在EvenGenerator处于不恰当的状态时，仍能够访问其中的信息。但是根据不同的操作系统和实现细节这个问题在循环多次之后也可能不会被探测到。有一点很重要，那就是递增程序自身也需要多个步骤，并且在递增过程中任务可能被挂起。也就是说递增在Java中不是原子性操作。因此，如果不保护任务，即使单一的递增也不是安全的。

## 解决共享资源竞争

使用线程有一个基本问题：你永远不知道一个线程何时在运行。对于并发操作，你需要某种方式来防止两个任务访问相同的资源，至少在关键阶段不能出现这种情况。防止这种冲突的方法是当资源被一个任务使用时，在其上加锁。第一个访问某项资源的任务必须锁定这个资源，使其他任务在其被解锁前无法访问它，而在其解锁之时，另一个任务就可以锁定并使用它，以此类推。

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。这意味着在给定时刻只允许一个任务访问共享资源。通常这种是通过在代码前面加上一句锁语句来实现的，这就使得在一段时间内只有一个任务可以运行这段代码。因为锁语句产生一种相互排斥的效果，这种机制称为互斥量。

另外，当一个锁被解锁的时候，我们并不能确定下一个使用锁的任务，因为线程调度机制并不是确定性的。可以通过yield()和setPriorit()来给线程调度器提供建议。

Java以提供关键字synchronized的形式，为防止资源冲突提供了内在支持。当任务要执行被synchronized关键字保护的代码片段的时候，它将检查锁是否可用，然后获取锁、执行代码、释放锁。共享资源一般是以对象形式存在于内存片段，可以是文件、输入输出端口。要控制对共享资源的访问，得先把它包装进一个对象。然后把所有要访问这个资源的方法标记为synchronized。

synchronized void f(){};

synchronized void g();

声明synchronized方法的方式

所有对象都自动含有单一的锁（监视器）。当在对象上调用其任意synchronized方法的时候，此对象被加锁，这时这个对象上的其它synchronized方法只有等到前一个方法调用完毕并释放了锁之后才能被调用。对于某个特定对象来说，其所有synchronized方法共享同一个锁，这可以被用来防止多个任务同时访问被编码为对象内存。

注意：使用并发时将对象设置为private是非常重要的，否则，synchronized关键字就不能防止其他的任务直接访问域，这样就会产生冲突。

针对每个类也有一个锁，所以synchronized修饰static方法可以在类的范围内防止对static数据的并发访问。

同步的选择：

如果你正在写一个变量，它可能接下来被另一个线程读取，或者正在读取一个上一次被另一个线程写过的变量，那么你必须使用同步，并且，读写线程都必须用相同的监视器锁同步。

通过在EvenGenerator中加入synchronized关键字，可以防止不希望的线程访问：

[

参考代码：sync/sync/IntGenerator.java、sync/sync/EvenGenerator.java、sync/sync/EvenChecker.java

]

对Thread.yield()的调用被插入到两个线程之间，以提高奇数的可能性。因为互斥可以防止多个任务同时进入临界区，所以上面不会产生任何的失败。第一个进入next()的任务获得锁，任何其他试图获取锁的任务都将被阻塞，直到第一个任务释放锁。

### 使用显式的Lock对象

Java SE5的类库中还包含定义在java.util.concurrent.locks中的显式的互斥机制。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。但是对于解决某些类型的问题时更加的灵活。

[

参考代码: sync/lock/IntGenerator.java、sync/lock/EvenGenerator.java、sync/lock/EvenChecker.java

]

当你在使用lock对象时，一些惯用写法很重要：

对unlock()方法的调用必须放在try-finally语句中。注意，上面代码中的return语句必须在try子句中出现，以确保unlock()不会过早地发生，从而将数据暴露在第二个任务。尽管try-finally子句比synchronized关键字要多，但显式的lock的优点也是显而易见的。如果在使用synchronized关键字时某些事务失败了，那么就会抛出一个异常。但是我们并没有机会去处理，以维护系统良好的状态。而使用显式的lock对象，你就可以使用finally子句维护系统的正确状态。大体上我们使用synchronized的情况更多，只有遇到解决特殊问题时才使用显式的lock对象。

[

参考代码：sync/lock/AttemptLocking.java

执行结果：

tryLock(): true

tryLock(2, TimeUnit.SECONDS): true

tryLock(): true

tryLock(2, TimeUnit.SECONDS): true

acquired

]

ReentrantLock允许我们尝试着获取锁但是最终未获取锁，这样如果其他人已经获取了锁，那么你就可以决定离开做一些其他的事情，而不是一直等待这个锁被释放。显式的Lock对象在加锁和释放锁方面，相对于内建的synchronized锁来说，还赋予了你更细粒度的控制力。

### 原子性和易变性

在Java线程中，常常我们会认为原子操作不需要进行同步控制。原子操作是不能被线程调度机制中断的。这样的想法是错误的，依赖于原子性是危险的。原子性在Java的类库中已经实现了一些更加巧妙的构建。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”。但是JVM会把64位的long和double操作当做两个分离的32位的操作来执行，这就产生了一个读取和写入操作之间产生上下文切换，从而导致了不同的任务产生不正确结果的可能性。但是如果我们使用volatile关键字就会获得原子性（在Java SE5之前一直未能正确工作）。

因此，原子操作可由线程机制来保证其不可中断，但是即便这样，这也是一种简化的机制。有时看起来很安全的原子性操作实际上也可能不安全。

在多核处理器上，可视性问题远比原子性问题多得多。一个任务做出的修改可能对其他任务是不可见的。因为每个任务都会暂时把信息存储在缓存中。同步机制强制在处理器中一个任务做出的修改必须是可见的。volatile关键字确保了这种可视性。一个任务修改了被volatile修饰的对象，那么其他的任务读写操作都能看到这个修改，即使是用了缓存也能被看到。因为volatile会被立即写入主存，而读写操作就发生在主存中。同步也会导致向主存中刷新，所以如果一个对象是synchronized保护的那么就不必使用volatile修饰。使用volatile而不是synchronized的唯一安全的情况是类中只有一个可变的域。我们的第一选择应该是synchronized关键字，这是最安全的方式。

### 什么是原子性操作

对域中的值做赋值和返回操作通常都是原子性的。但是递增和递减并不是。

[

参考代码：sync/atom/Atomicity.java

]

递增或递减操作在编译后每个操作都会产生一个get和一个put指令，get和put指令之间还有一些其他的指令，因此在获取和修改之间，另一个任务可能会修改这个域。所以，这些操作不是原子性的。

再看一个例子：

[

参考代码:sync/atom/AtomicityTest.java

测试结果：

1

]

该程序找到奇数并终止。尽管“return i”是原子性操作，但是缺少同步使得其数值可以在不稳定的中间状态时被读取。还有由于i不是volatile的也存在可视性的问题。getValue()和evenIncrement()必须都是synchronized的。对于基本类型的读取和赋值操作被认为是安全的原子性操作。但是当对象处于不稳定状态时，仍旧很有可能使用原子性操作得到访问。最明智的做法是遵循同步的规则。

### 原子类

Java SE5中引入了诸如AtomicInteger、AtomicLong、AtomicReference等特殊的原子性变量类，它们提供下面形式的原子性条件更新操作：

boolean compareAndSet( expectedValue, updateValue);

这些类被调整为可以使用在现代处理器上，并且是机器级别的原子性，因此在使用他们时不需要担心。常规来说很少使用他们，但是对于性能调优来说，他们就大有用武之地了。

[

参考代码：sync/atom/AtomicInteger.java

]

Atomic类被设计为构建java.util.concurrent中的类，因此只有在特殊情况下才在代码中使用它们。使用Atomic类型的原子类可以在没有使用任何加锁机制的情况下也能得到很好的同步。但是通常依赖于锁对我们来说更安全一点。

### 临界区

有时我们需要防止多个线程同时访问方法内部的部分代码而不是防止访问整个方法。通过这种方式分离出来的代码被称为临界区，也是使用synchronized关键字修饰。语法是：synchronized被用来指定某个对象，此对象的锁被用来对括号内的代码进行同步控制：

synchronized( syncObject){

//被同步控制的代码块

}

这被称之为同步代码块；在进入此段代码之前，必须得到syncObject对象的锁。如果其他线程已经得到锁，那么就得等到锁被释放之后才能进入临界区。通过使用同步控制块，而不是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得到显著提高。

[

参考代码：sync/block/Pair.java、sync/block/PairChecker.java、sync/block/PairManager.java、sync/block/PairManager1.java、sync/block/PairManager2.java、sync/block/CriticalSection.java、sync/block/PairManipulator.java

最后的测试结果：

pm1: Pair: x: 11, y: 11 checkCounter = 2183

pm2: Pair: x: 12, y: 12 checkCounter = 24600386

]

尽管每次运行的结果可能会不同，但一般情况下PairChecker的检查频率PariManager1比PairManager2少。后者采用同步代码块进行控制，所以对象不加锁的时间更长。使得其他线程能够更多地访问。

### 在其他对象上同步

synchronized块必须给定一个在其上同步的对象，并且合理的方式是，使用其方法正在被调用的当前对象：synchronized(this)，在这种方式中如果获得了synchronized块上的锁，那么该对象其他的synchronized方法和临界区就不能被调用了。

有时必须在另外一个对象上同步，但是如果你这样做，就必须确保所有相关的任务都是在同一个对象上同步的。

下面的例子演示了两个任务可以同时进入同一个对象，只要这个对象上的方法是在不同的锁上同步的即可：

[

参考代码：sync/sync/SyncObject.java

执行结果：

g()

f()

g()

f()...

]

其中，f()是在this上同步的，而g()是在一个syncObject上同步的synchronized块。因此，这两个同步是相互独立的。通过在main()中的方法调用可以看到，这两个方法并没有阻塞。

## 线程本地存储

防止任务在共享资源上产生冲突的第二中方式是根除对变量内存的共享。线程本地存储是一种自动化机制，可以使依赖相同变量的每个不同的线程创建不同的存储。因此，如果你有5个线程，那么线程会在本地生成5个不同的存储块。它们使得你可以将状态和线程关联起来。

创建和管理线程本地存储可以由java.lang.ThreadLocal类来实现：

[

参考代码: local/Accessor.java、local/ThreadLocalVariableHolder.java

测试结果：

#0:712564

#0:712565

#0:712566

#0:712567

#0:712568/...

]

ThreadLocal对象通常当做静态存储域。创建ThreadLocal方法时只能通过get()和set()方法来访问内容，其中，get()方法返回与对象相关联的副本，而set()将会把参数插入到为其线程存储的对象中，并返回存储中原有对象。运行这个程序的时候会发现每个单独的线程都分配了自己的存储，因为它们每个都要跟踪自己的计数值。

死锁

一个对象可以有synchronized方法或其他形式的加锁机制来防止别的任务在互斥还没有释放的时候就访问这个对象，这样就产生了阻塞任务。任务有可能变成阻塞状态，所以就可能发生这样的情况：某个任务在等待另一个任务，而后者又在等待别的任务，这样一直下去，直到这个链条上的任务又在等待第一个任务释放锁。这就形成了一个相互等待的循环，没有哪个线程能够继续。这被称之为死锁。

我们真正需要解决的问题是程序看起来可能工作良好，但是具有潜在的死锁风险。这时，死锁可能发生，而事先却没有任何征兆，所以缺陷会潜伏在你的程序里，知道被人意外的发现了。因此，在编写并发编程的时候，进行仔细的程序设计以防止死锁是非常关键的。

下面引入一个问题，一共有5位哲学家，这些哲学家将花部分时间思考，花部分时间就餐。作为哲学家他们很穷，所以他们只能买5根筷子。他们围坐在桌子的周围，每人之间放一根筷子。当一个哲学家要就餐的时候，这个哲学家必须同时得到左边和右边的筷子。如果一个哲学家左边或者右边已经得到筷子，那么这个哲学家就必须等待，直至可得到必须的筷子。

[

参考代码：deadlock\Chopstick.java

]

任何两个哲学家都不能使用同一根筷子。也就是不能同时taken()同一个筷子。另外，如果一个Chopstick被一个哲学家获得，那么另一个哲学家可以wait()，直到当前的这根筷子的持有者调用drop()结束使用。

[

参考代码：deadlock\Philosopher.java

]

在哲学家的任务中，每个哲学家都是不断地思考和吃饭。如果ponderFactor不为0，则pause()就会休眠一会儿。通过这样的方法你会看到哲学家会思考一段时间。然后尝试着去获取左边和右边的筷子，随后再在吃饭上花掉一段随机的时间，之后重复此过程。

现在我们来建立这个程序的死锁版本：