# 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：

[

参考代码：deadlock\DeadLockingDiningPhilosophers.java

执行的结果：

执行结束

哲学家的编号：2开始思考

哲学家的编号：4开始思考

哲学家的编号：1开始思考

哲学家的编号：0开始思考

哲学家的编号：3开始思考

当前线程被中断了

当前线程被中断了

当前线程被中断了

当前线程被中断了

当前线程被中断了

]

这个程序表示每一个哲学家都有可能要表示进餐，从而等待其临近的Philosopher放下他们的Chopstick。这将会使得程序死锁。

要修正死锁必须明白，当以下四个条件同时满足时，就会发生死锁：

1. 互斥条件。任务使用的资源中至少有一个是不能共享的。
2. 至少有一个任务必须持有一个资源，并且正在等待获取一个当前被别的任务持有的资源。也就是说必须是拿着一根筷子等待另一个筷子。
3. 资源不能被任务抢占，任务必须把资源释放当做普通事件。你不能抢别人手里的筷子。
4. 必须有循环等待，这时一个任务等待其他任务所持有的资源，后者又在等待另一个任务所持有的资源，这样循环下去直到有一个任务等待第一个任务所持有的资源，使得大家都被锁住。

因为要发生死锁所有以上这些条件必须满足，所以要防止死锁的话只需要破坏其中一个就可以。在程序中防止死锁的最容易的办法就是破坏第四个循环条件。

[

参考代码：deadlock\FixedDiningPhilosophers.java

执行结果：

哲学家的编号：2开始拿左边的筷子

哲学家的编号：4开始思考

哲学家的编号：1开始思考

哲学家的编号：0开始拿左边的筷子

哲学家的编号：0开始拿右边的筷子

哲学家的编号：0开始就餐

哲学家的编号：3开始就餐

哲学家的编号：2开始拿右边的筷子

/.....

]

通过确保最后一个哲学家先拿起和放下左边的筷子，我们可以移除死锁，从而使得程序运行。Java并没有对死锁提供类库上的支持，能否通过仔细的程序设计避免死锁需要我们自己努力。

# 线程之间的协作

当你使用多线程来同时运行多个任务时，可以通过使用锁来同步两个任务的行为，从而使得一个任务不会干涉另一个任务的资源。也就是说，如果两个任务交替的不如某项共享资源，你可以使用互斥来保证任何时刻只有一个任务可以访问这项资源。

解决了多任务对共享资源的访问后，下一步是如何使得任务彼此之间可以协作，使得多个任务可以一起工作去解决某个问题。现在的问题不是彼此之间的干涉，而是彼此之间的协作。解决这类问题的关键是某些部分必须在其他部分被解决之前解决。

当任务协作时，关键问题是这些任务之间的握手。为了实现握手，我们使用了相同的基础特性：互斥。在这种情况下，互斥能够确保只有一个任务可以响应某个信号，这样就能根除任何可能的竞争条件。在互斥上，我们为任务添加了一种途径，可以将自身挂起，直至某些外部条件发生变化，表示是时候让这个任务开始为止。

## wait()与notifyAll()

wait()可以使你等待某个条件发生变化，而改变这个条件通常是由另一个任务来改变。你肯定不想在你的任务测试这个条件的同时，不断的进行空循环，这被称为忙等待，是一种不良的cpu使用方式。因此wait()会在外部条件发生变化的时候将任务挂起，并且只有在notify()或notifyAll()发生时，这个任务才会被唤醒并取检查所发生的变化。因此，wait()提供了一种在任务之间对活动同步的方式。

调用sleep()的时候锁并没有被释放，调用yield()也是一样。当一个任务在方法里遇到对wait()调用时，线程执行被挂起，对象的锁被释放，这就意味着另一个任务可以获得锁，因此在该对象中的其他synchronized方法可以在wait()期间被调用。因此，当你在调用wait()时，就是在声明：“我已经做完了所有的事情，但是我希望其他的synchronized操作再条件合适的情况下能够被执行”。

有两种形式的wait():

* 第一种接收毫秒作为参数：指再次暂停的时间。

在wait()期间对象锁是被释放的。

可以通过notify()或notifyAll()，或者指令到期，从wait()中恢复执行。

* 第二种不接受参数的wait()

这种wait()将无限等待下去，直到线程接收到notify()或notifyAll()。

wait()、notify()以及notifyAll()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类。仅仅作为线程的功能却成为了通用基类的一部分。原因是这些方法操作的锁，也是所有对象的一部分。所以你可以将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Runnable。实际上，只能在同步方法或者同步代码块里调用wait()、notify()或者notifyAll()。如果在非同步代码块里操作这些方法，程序可以通过编译，但是在运行时会得到IllegalMonitorStateException异常。意思是，在调用wait()、notify()或者notifyAll()之前必须拥有获取对象的锁。

比如，如果向对象x发送notifyAll()，那就必须在能够得到x的锁的同步控制块中这么做：

synchronized(x){

x.notifyAll();

}

我们看一个示例：一个是将蜡涂到Car上，一个是抛光它。抛光任务在涂蜡任务完成之前，是不能执行其工作的，而涂蜡任务在涂另一层蜡之前必须等待抛光任务完成。

[

参考代码: wait\Car.java、wait\WaxOn.java、wait\WaxOff.java、wait\WaxOmatic.java

执行结果：

/.....

Wax one

Wax off

Wax one

Wax off

Wax one

Wax off

Exiting via interrupt

Wxtiing via interrupt

Ending wax on task

Ending wax off task

]

在waitForWaxing()中检查WaxOn标志，如果它是false，那么这个调用任务将会被挂起。这个行为发生在synchronized方法中这一点很重要。因为这个方法中任务已经获得了锁。当你调用wait()时，线程被挂起，而锁被释放。释放锁是本质所在，因为为了安全地改变对象的状态，其他某个任务就必须能够获得这个锁。

WaxOn.run()表示给汽车打蜡的第一个步骤，它执行它的操作：调用sleep()模拟打蜡的时间，然后告知汽车打蜡结束，并且调用waitForWaxing()，这个方法会调用wait()挂起当前打蜡的任务。直到WaxOff任务调用这辆车的buffed()，从而改变状态并且调用notifyAll()重新唤醒为止。反过来也是一样的，在运行程序时，你可以看到控制权在两个任务之间来回的传递，这两个步骤过程在不断地重复。

### 错失的信号

当两个线程使用notify() / wait()或者notifyAll() / wait()进行协作时，有可能会错过某个信号。假设线程T1是通知T2的线程，而这两个线程都使用下面的方式实现：

T1：

synchronized(X){

//设置T2的一个条件

<setup condition for T2>

X.notify();

}

T2:

while( someCondition ){

//Potit

synchronized( X ){

X.wait();

}

}

以上的例子假设T2对someCondition发现其为true()。在执行Potit其中线程调度器可能切换到了T1。而T1将会执行重新设置condition，并且调用唤醒。当T2继续执行时，以至于不能意识到条件已经发生变化，因此会盲目的进入wait()。此时唤醒在之前已经调用过了，而T2将无限的等待下去唤醒的信号。

解决该问题的方案是防止someCondition变量上产生竞争条件：

synchronized(x){

while(someCondition){

x.wait();

}

}

## notify()与notifyAll()

可能有多个任务在单个Car对象上被挂起处于wait()状态，因此调用notifyAll()比调用notify()更安全。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。另外使用notify()，所有任务都必须等待相同的条件，因为如果你有多个任务在等待不同的条件，那你就不会知道是否唤醒了恰当的任务。如果使用notify()，当条件发生变化时，必须只有一个任务能从中受益。最后，这些限制对所有可能存在的子类都必须总起作用。如果这些规则任何一条不满足都必须使用notifyAll()。

在Java的线程机制中，有一个描述是这样的：notifyAll()将唤醒所有正在等待的任务。这是否意味着在程序中任何地方、任何处于wait()状态中的任务都将被任何对notifyAll()的调用唤醒呢？在下面的实例中说明了情况并非如此，当notifyAll()因某个特定锁被调用时，只有等待这个锁的任务才会被唤醒：

[

参考代码：wait\Blocker.java、wait\Task.java、wait\Task2.java、wait\NotifyVsNotifyAll.java

测试结果：

notify

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

Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-1,5,main] notify

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

Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-3,5,main] notify

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

Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-1,5,main] notify

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

Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-3,5,main] notify

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

Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-1,5,main] notify

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

Thread[pool-1-thread-2,5,main] Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-3,5,main] Time cancle

Task2.blocker.prodAll()

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

Shutting down

]

从上面输出的结果可以看出，我们启动了三个Task任务线程，一个Task2线程。使用timer做了一个定时器，每间隔4毫秒就轮换启动Task.blocker的notify()和notifyAll()方法。我们看到Task和Task2都有Blocker对象，他们调用Blocker对象的时候都会被阻塞。我们看到当调用Task.prod()的时候只有一个在等待锁的任务被唤醒，其余两个继续挂起。当调用Task.prodAll()的时候等待的三个线程都会被唤醒。当调用Task2.prodAll()的时候只有Task2的线程任务被唤醒。其余的三个Task任务继续挂起。

### 生产者与消费者

请考虑这样一种情况，在饭店有一个厨师和一个服务员。这个服务员必须等待厨师做好膳食。当厨师准备好时会通知服务员，之后服务员上菜，然后返回继续等待。这是一个任务协作示例：厨师代表生产者，而服务员代表消费者。两个任务必须在膳食被生产和消费时进行握手，而系统必须是以有序的方式关闭。

[

参考代码：wait\Meal.java、wait\WaitPerson.java、wait\Chef.java、wait\Restaurant.java

执行结果：

Order up! Waitperson got Meal 1

Order up! Waitperson got Meal 2

Order up! Waitperson got Meal 3

Order up! Waitperson got Meal 4

Order up! Waitperson got Meal 5

Order up! Waitperson got Meal 6

Order up! Waitperson got Meal 7

Order up! Waitperson got Meal 8

Order up! Waitperson got Meal 9

Out of food, closing

Order up! WaitPerson interrupted

Chef interrupted

]

### 使用显式的Lock和Condition对象

在Java SE5的类库中还有额外的工具。使用互斥并允许任务挂起的基本类是Condition，你可以通过在Condition上调用await()来挂起一个任务。当外部条件发生变化时，意味着某个任务应该继续执行，你可以通过调用signal()来通知这个任务，从而唤醒一个任务，或者调用signalAll()来唤醒所有在这个Condition上被挂起的任务。signalAll()比notifyAll()是更安全的方式。

下面我们来重写我们的打蜡和抛光类：

[

参考代码：wait\WaxOMatic2.java

]

在Car的构造器中单个的Lock将产生一个Condition对象，这个对象被用来管理任务之间的通信。但是这个Condition不包含任何有关处理状态的信息，因此你需要额外的表示处理状态的信息，即Boolean waxOn()。

### 生产者消费者与队列

wait()和notifyAll()方法以一种非常低级的方式解决了任务的互操作的问题，即每次交互时都握手。许多时候我们可以使用同步队列来解决协作的问题，同步队列在任何时刻只允许一个任务插入或移除元素。在java.util.concurrent.BlockingQueue接口中提供了这个队列，这个接口有大量的标准实现。可以使用LinkedBlockingQueue，它是一个无界队列，还可以使用ArrayBlockingQueue，它具有固定的尺子，可以在它被阻塞之前向其中放置有限数量的元素。

如果消费者任务试图从队列中获取对象，而该队列为空时，那么这些队列就可以挂起这些任务，并且当有更多的元素可用时恢复这些消费任务。阻塞队列可以解决非常大的问题，而其方式与wait()和notifyAll()相比，则简单且可靠。

下面是一个简单的测试，它将多个LiftOff对象执行串行化。消费者LiftOffRunner将每个LiftOff对象从BlockingQueue中推出并直接运行。它通过显式地调用run()而使用自己的线程来运行，而不是为每个任务启动一个线程。

[

参考代码：queue\LiftOff.java、queue\LiftOffRunner.java、queue\TestBlockingQueues.java

]

### 吐司BlockingQueue

下面是一个示例，每一台机器都有三个任务：一个只做吐司、一个给吐司抹黄油、另一个在涂抹黄油的吐司上抹果酱。我们来示例如果使用BlockingQueue来运行这个示例：

[

参考代码：queue\ToastOMatic.java

]

这个示例中没有任何显式的同步，因为同步队列和系统的设计隐式地管理了每片Toast，在任何时刻都只有一个任务在操作。因为队列的阻塞，使得处理过程将被自动挂起和恢复。

# 终结任务

有时候我们需要任务被突然终止。

## 装饰性花园

这是一个让我们观察的示例，它不仅演示了终止问题，而且还演示了资源共享。下面的仿真程序中，花园委员会希望知道每天进入公园的总人数。每个公园门口都有一个计数器，并且任何一个门口的计数值递增时，就表示公园中的总人数的共享数值也会递增。

[

参考代码: shutdown/ Count.java、shutdown/ Entrance.java、shutdown/ OrnamentalGraden.java

测试结果：

/....

Entrance0: 2092Total10425

Entrance1: 2082Total10426

Entrance3: 2082Total10427

Entrance2: 2085Total10428

Entrance4: 2088Total10429

结束了

结束了

结束了

结束了

结束了

总人数10429

所有的 Entrances:10429

]

我们使用单个的count对象来跟踪参观者的主计数器，并且将其作为Entrance的静态域存储。Count.increment()和Count.value()都是synchronized的，用来控制对count域的访问。

每个Entrance任务都维护着一个本地值number, 它包含通过某个特定入口进入的参观者数量。这提供了双重检查，以确保记录的参观者数量是正确的。

因为canceled是volatile布尔值，而且它只会被读取和赋值，所以不需要同步对其访问，就可以安全的操作它。

执行任务3秒之后，main()向Entrance发送static cancel()消息，然后调用exec对象的shutdown()方法，之后调用了exec的awaitTermination()方法。这个方法表示等待每个任务结束，如果所有的任务在超时时间未结束，则返回true，否则返回false，表示不是所有的任务都已经结束。

当执行run()方法时，你会看到输出了通过每个门口的人数和此时的总人数。如果移除Count.increment()的synchronized的声明，你会看到总人数与你的期望会有差异，只要用互斥来同步对Count的访问，问题就可以解决。Count.increment()通过使用temp()和yield()，增加了失败的可能性。在真正的线程问题中失败的可能性在统计学角度看非常小，因此可能很容易的掉进陷阱。

## 在阻塞时中介

前面示例中的run()方法在其循环中包含对sleep()的调用。但是sleep()会使任务从执行状态变为阻塞状态，而有时你必须终止被阻塞的任务。

### 线程状态

一个线程可以处于以下四种状态之一：

* + 新建（new）：

当线程被创建时，它只会短暂处于这种状态。此时已经分配了必要的资源，并执行初始化。此刻线程已经有资格获得cpu时间了，之后调度器会把这个线程转变为可运行状态或阻塞状态。

* + 就绪（Runnable）：

在这种状态下只要调度器把时间片分配给线程，线程就可以运行。在任意时刻线程可以运行也可以不运行，取决于调度器是否分配给线程时间片。

* + 阻塞（Blocked）:

线程能够运行，但有个条件阻止它运行。当线程处于阻塞状态时，调度器将忽略线程，不会分配给它任何cpu时间。

* + 死亡（Dead）:

处于死亡和终止状态的线程将不再可被调度，并且再也不会得到cpu时间，它的任务已经结束，或不再是可运行的。任务死亡的方式通常是从run()方法返回，但是任务的线程还可以被中断。

### 进入阻塞状态

一个任务进入阻塞状态可能有如下原因：

* 通过调用sleep()使任务进入休眠状态，在这种情况下任务在指定的时间内不会运行。
* 你通过调用wait()使线程挂起。直到线程得到了notify()或notifyAll()消息，线程才会进入就绪状态。
* 任务在等待某个输入或输出完成。
* 任务试图在某个对象上调用其同步控制方法，但是对象锁不可用，因为另一个任务已经获取了锁。
* stop()方法，已经被废弃，因为它不释放线程获得的锁，并且线程处于不一致的状态，其他任务在这种状态下可以浏览并修改它们。这所产生的问题很难发现。

现在我们需要解决的问题是：有时我们希望能够终止处于阻塞状态的任务。我们决定让其主动终止，那么必须强制这个任务跳出阻塞状态。

### 中断

在Runnable.run()方法中间打断任务是非常棘手的。当你打断被阻塞的任务时可能需要清理资源。正是由于这一点，在run()方法中间打断更像是抛出异常。为了以此方式终止任务并且返回良好的状态，你必须仔细考虑代码的执行路径，并正确编写catch子句正确清理所有的事务。

Thread类包含interrupt()方法，因此你可以终止被阻塞的任务，这个方法将设置线程的中断状态。如果一个线程被阻塞或者试图执行一个阻塞操作，那么设置这个线程的阻塞状态将抛出InterruptedException。当抛出该异常或者该任务调用Thread.interrupt()时，中断状态将被复位。Thread.interrupt()提供了离开run()循环而不抛出异常的办法。

Java的新类库尽量避免我们直接操作Thread对象，而是尽量使用Executor来执行所有操作。如果你再Executor上调用shutdownNow()，那么将会发送一个interrupt()给由它启动的所有线程。然而有时我们需要的是中断其中的某一个任务，此时如果我们使用Executor调用submit()来启动任务，就可以持有该任务的上下文对象。submit()将返回一个泛型的Future<?>，其中有一个未修饰的参数，持有这个Future的关键是可以在其上调用cancel()。那么就可以拥有在该线程上调用interrupt()以停止这个线程的权限。

下面的示例用Executor展示了基本的interrupt()用法：

[

参考代码：interrupt\ Interrupting.java

执行结果：

Interrupting SleepBlocked

InterruptedException

Exiting SleepBlocked.run()

Interrupt sent to SleepBlocked

Waiting for read():

Interrupting IOBlocked

Interrupt sent to IOBlocked

Trying to call f()

Interrupting SynchronizedBlocked

Interrupt sent to SynchronizedBlocked

Aborting with System.exit(0)

]

上面的任务展示了三种不同的阻塞类型。SleepBlock是可中断的阻塞类型，而IOBlocked和SynchronizedBlocked是不可中断的阻塞实例。从输出中可以看出，能够中断对sleep()的调用（任何抛出InterruptedException的调用）。但是不能中断试图获取synchronized锁或者试图执行I/O操作的线程。

对于这类问题我们有一个行之有效的解决方案，即关闭任务在其发生阻塞的底层资源：

[

参考代码：interrupt\ CloseResource.java

执行结果：

Waiting for read():

Waiting for read():

Shutting down all threads

Closing java.net.SocketInputStream

Interrupted from blocked I/O

Exiting IOBlocked.run()

Closing java.io.BufferedInputStream

Exiting IOBlocked.run()

]

在shutdownNow()被调用之后和两个输入流被调用close()之前的延迟强调的是一旦底层资源被关闭，任务将解除阻塞。

### 被互斥所阻塞

如果你尝试在一个对象上调用其synchronized方法，而这个对象的锁已经被其他任务获得，那么调用任务将会被挂起，直至这个锁被获得。下面的示例说明了同一个互斥可以被同一个任务多次获得：

[

参考代码: interrupt\ MultiLock.java

执行结果：

f1 调用 f24

f2 调用 f13

f1 调用 f22

f2 调用 f11

f1 调用 f20

]

同一个任务先获得了f1()中的锁，又获得了f2()中的锁。这么做是有意义的，因为一个任务能够调用同一个对象中的其他的synchronized方法，而这个任务已经持有了锁。

前面I/O的例子我们观察到，只要任务以不可中断的方式被阻塞，那么都有潜在的锁住程序的可能。Java SE5类库中添加了一个特性，即在ReentrantLock上阻塞的任务具备可以被中断的能力。

[

参考代码：interrupt\ BlockedMutex.java、interrupt\ Blocked2.java、interrupt\ Interrupting2.java

执行结果：

开始线程执行

线程调用 interrupt(

BlockMutex 被中断异常

被中断了

]

### 检查中断

注意，当你在线程上调用interrupt()时，中断发生的唯一时刻是在任务要进入到阻塞操作中，或者已经在阻塞操作内部时。如果你的run()方法没有产生任何阻塞调用的情况下，调用interrupt()将无法停止某个任务。你的任务此时需要第二种方式退出。

这种机会是由中断状态来表示的，其状态可以通过调用interrupt()来设置。你可以调用interrupted()来检查中断状态，这不仅可以告诉你interrupt()是否被调用过，而且还可以清楚中断状态。清楚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来得到通知。如果想再次检查了解是否被中断，则可以调用Thread.interrupted()时将结果保存起来。

下面展示了典型的检查中断方法：

[

参考代码：interrupt\ NeedsCleanup.java、interrupt\ Blocked3.java、interrupt\ InterruptingIdiom.java

测试结果：

NeedsCleanup:1

java.lang.InterruptedException: sleep interrupted

cleanup()1

NeedsCleanup:1

at java.lang.Thread.sleep(Native Method)

at java.lang.Thread.sleep(Thread.java:340)

at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)

at concurrency.Blocked3.run(Blocked3.java:13)

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

NeedsCleanup:2

计算完毕

cleanup()2

cleanup()1

NeedsCleanup:1

NeedsCleanup:2

/....

]

注意看第一条的异常，NeedsCleanup类强调在你经由异常离开循环时，正确清理资源的必要性。注意：所有的run()方法中创建的资源都必须在其后紧跟try-finally子句，以确保nCleanup()方法总会被调用。

# SE5新特性

Java SE5的类库中引入了大量的新设计的用来解决并发问题的新类。学习它们将有助于编写更加简单而健壮的并发程序。

## CountDownLatch

它被用来同步一个或多个任务，强制它们等待由其他的任务执行的一组操作完成。

你可以向CountDownLatch对象设置一个初始计数值，任何在这个对象上调用wait()的方法都将阻塞，直至这个计数值到达0。其他任务在结束其工作时，可以在该对象上调用CountDown()来减小这个计数值。CountDownLatch被设计为只触发一次，计数值不能被重置。如果需要重置计数值的版本，则可以使用CyclicBarrier版本。调用countDown()的任务在产生这个调用时并没有被阻塞，只有对await()的调用会被阻塞，直至计数值到达0。

CountDownLatch的典型用法是将一个任务分为n个互相独立的可解决任务，并创建值为0的CountDownLatch。当每个任务完成时，都会在这个锁存器上调用countDown()。等待问题被解决的任务在这个锁存器上调用await()，将它们自己拦住，直至锁存器计数结束。

[

参考代码: interrupt\ CountDownLatchDemo.java

]

所有的任务都使用在main()中定义的同一个单一的CountDownLatch。

## 类库的线程安全

类中包含了一个静态的Random对象，这意味着多个任务可能会同时调用Random.nextInt()。在这种情况下，可以通过向TaskPortion提供自己的Random对象来解决。也就是说通过移除static限定符的方式解决。对于Java标准类库来说哪些是线程安全的、哪些是线程不安全的，JDK文档并没有指出。要理解这一点必须逐个的去查看源码，恰好Random.nextInt()是线程安全的。

## CyclicBarrier

你希望创建一组任务，它们并行地执行工作，然后在进行下一个步骤之前等待，直至所有任务都完成。这使得所有的任务都在栅栏处等待，因此可以一致地向前移动。

下面是模仿赛马游戏的一个仿真版本：

[

参考代码：concurrent\ HorseRace.java

]

可以向CyclicBarrier提供一个栅栏动作，它是一个Runnable，当计数值到达0时自动执行。这里栅栏动作是作为匿名内部类来创建的，它被提交给CyclicBarrier的构造器。CyclicBarrier使得每匹马都执行了向前移动所必须执行的工作，然后等待栅栏处所有的马都准备完毕。当所有的马向前移动时，CyclicBarrier将自动调用Runnable栅栏动作任务，按顺序显示马和终点线的位置。一旦所有的任务越过了栅栏，它就会自动地为下一回合比赛做好准备。

## DelayQueue

这是一个无界的BlockingQueue，用于放置实现DelayQueue接口的对象，其中的对象只能在其到期时才能从队列中取走。这种队列是有序的，即队列对象的延迟到期的时间最长。如果没有任何延迟到期时间，那么就不会有任何头元素，并且poll()将返回null。

下面示例：其中的Delayed对象自身就是任务，而DelayedTaskConsumer将最紧急的任务从（到期时间最长的任务）队列中去除，然后运行它。

[

参考代码：concurrent\ DelayQueueDemo.java

]

DelayedTask包含一个称为sequence的List<DelayedTask>，它保存了任务被创建的顺序，因此我们看到排序是按照实际发生的顺序执行的。Delsyed接口有一个方法名叫getDealay()，它可以用来告知延迟到期多长时间，或者延迟在多长时间以前已经到期。这个方法将强制我们使用TimeUnit类，因为这就是参数类型。

## PriorityBlockingQueue

这是一个很基础的优先级队列，它具有可阻塞的读取操作。下面是一个示例，其中在优先级队列中的对象是按照优先级顺序从队列中出现的任务。PrioritizedTask被赋予一个优先级数字，以此来提供顺序：

[

参考代码：concurrent\ PriorityBlockingQueueDemo.java

]

PrioritizedTask对象的创建序列被记录在sequence List中，用于和实际的执行顺序比较。PrioritizedTaskProducer和PrioritizedTaskConsumer通过PriorityBlockingQueue彼此连接。因为这种队列的阻塞特性提供了所有必须的同步，所以你应该注意到，这里不需要任何显式的同步，不必考虑当你从这种队列读取时，其中是否还有元素，因为这种队列在没有元素时将直接阻塞读取者。

## 使用ScheduledThreadPoolExecutor的温室控制器

假定温室控制系统的示例，它可以控制各种设施的开关，或者是对他们进行调节。这可以被看做是一种并发问题，每个期望的事件都是一个预定时间运行的任务。通过使用schedule()运行一次任务或者使用scheduleAtFixedRate()每隔规则的时间重复执行任务，你可以将Runnable对象设置为在将来的某个时刻执行。

[

参考代码：concurrent\ GreenhouseScheduler.java

]

DataPoint可以持有并显示单个的数据段，而CollectData是被调度的任务，它在每次运行时，都可以产生仿真数据，并将其添加到greenhouse的List<DataPoint>中。注意：volatile和synchronized在适当的场合都得到了应用，以防止任务之间的互相干涉。在持有DataPoint的List中的所有方法都是synchronized的，这是因为List被创建时使用了synchronizedList()。

## 总结

SE5中的新类库包含了许多的新特性，为我们解决问题提供了许多的方便。现在JDK已经更新到了8。要学习更多的新内容还需不断地学习新版本的JDK。

# 性能优化

在Java SE5类库中的java.util.concurrent类库中存在着数量庞大的用于性能提高的类。当细细读取这个类库时很难辨认哪些类适用于哪些常规应用，而哪些类适用于提高性能。

## 比较各类互斥技术

Java包括老式的synchronized关键字和Java SE5中新的Lock和Atomic类，那么比较这些不同的方式，更多的理解他们的各自价值和使用范围，就会显得很有意义。

[

参考代码：optimi\ Incrementable.java, optimi\ SynchronizingTest.java、optimi\ LockingTest.java、optimi\ SimpleMicroBenchmark.java

测试结果：

Synchronizing: 231087528

lockTime: 199864257

]

使用Lock要比使用synchronized高效的多，而且synchronized在不同次数运行过程中的的开销看起来变化范围太大，而Lock相对比较一致。

这是否意味着你永远都不应该使用synchronized关键字呢？这里有两个因素要考虑：首先要看互斥方法体的大小，在实际开发中互斥部分可能会非常大，因此在方法体中所花费的时间的的百分比可能会明显大于进入和退出互斥的开销，这样就淹没了提高互斥速度所带来的好处。当然对于这一点我们需要在性能调优时尝试各种不同的方法观察它们的影响。其次，很明显synchronized关键字所产的代码要比Lock要少的多，可读性也提高了许多。代码被阅读的次数远高于被编写的次数。在编程时与其他人的交流相对于与计算机的交流要重要的多，因此代码的可读性至关重要。

## 免锁容器

容器是所有编程的基础工具，这其中也包括并发编程。早期的容器类Vector和Hashtable具有许多的synchronized方法，当它们用于非多线程程序时，便会导致不可接受的开销。在Java 1.2中，新的容器类库是不同步的，并且Collections类提供了各种static的同步修饰方法，从而来同步不同类型的容器。尽管这是一种改进，因为它可以使你选择在你的容器中是否要使用同步，但是这种开销仍旧是基于synchronized加锁的机制。Java SE5特别添加了新的容器，通过使用更灵巧的技巧来消除加锁，从而提高线程安全的性能。

这些免锁容器背后的通用策略是：对容器的修改可以和读取操作同时发生，只要读取者只能看到完成修改后的结果即可。修改是在容器数据结构的某个部分的一个单独的副本上执行的，并且这个副本在修改过程中是不可见的。只有当修改完成时，被修改的结构才会自动地与主数据结构交换，之后读取者就可以看到修改之后的结果了。

在CopyOnWriteArrayList中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组再被修改时，读取操作可以安全地执行。当修改完成时，一个原子性的操作将把新的数组换入，使得新的读取操作可以看到这个新的修改。CopyOnWriteArrayList的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出ConcurrentModificationException。CopyOnWriteArraySet将使用CopyOnWriteArrayList来实现其免锁行为。ConcurrentHashMap和ConcurrentLinkedQueue使用了类似的技术，允许并发的读取和写入，但是容器中只有部分内容而不是整个容器可以被复制和修改。然而，任何修改在完成之前，读取者仍旧不能看到它们。

## 乐观锁

只要你主要是从免锁容器中读取，那么就会比其对应的synchronized快许多，因为获取和释放锁的开销被省掉了。如果要向免锁容器执行少量的写入，那么情况也是如此。下面我们将用一个测试的例子来展示在各种不同的情况下，这些容器在性能方面差异的大致概念。

从一个泛型的框架入手，专门用于在任何类型的容器上执行测试，包括各种Map在内，其中泛型参数c表示容器的类型：

[

参考代码: lock\ Tester.java

]

abstract方法containerInitializer()返回将被测试的初始化后的容器，它被存储在testContainer域内。另一个abstract方法startReadersAndWriters()启动读取和写入任务，他们将读取和修改待测容器。不同的测试在运行时将具有数量变化的读取者和写入者，这样就可以观察到锁竞争和写入的效果。我们向构造器提供了各种有关的测试的信息，然后回调runTest()方法repetions次。runTest()将创建一个CountDownLatch、初始化容器，然后调用startReadersAndWriters()，并等待它们全部完成。

每个reader和writer类都基于TestTask，它可以度量其抽象方法test()的执行时间，然后在一个synchronized块中调用putResults()去存储度量结果。

为了使用这个框架，我们必须让想要测试的特定类型容器集成Tester，并提供适合的Reader和Writer类：

[

参考代码：optimi\lock\ ListComparisons.java

执行结果：

Type Read time Write time

Synched ArrayList 10r 0w 232158294700 0

Synched ArrayList 9r 1w 198947618203 24918613399

readTime + writeTime = 223866231602

Synched ArrayList 5r 5w 117367305062 132176613508

readTime + writeTime = 249543918570

CopyOnWriteArrayList 10r 0w 758386889 0

CopyOnWriteArrayList 9r 1w 741305671 136145237

readTime + writeTime = 877450908

CopyOnWriteArrayList 5r 5w 212763075 67967464300

readTime + writeTime = 68180227375

]

从输出结果来看，synchronized ArrayList无论读取者和写入者的数量是多少，都具有大致相同的性能。读取者与其他读取者竞争锁的方式与写入者相同。但是，CopyOnWriteArrayList在没有写入者时速度会更快，并且在有多个写入者时速度仍然很快。看起来你应该尽量使用CopyOnWriteArrayList，对列表写入的影响并没有超过短期同步整个列表的影响。当然，不同的环境下测试是不同的，你必须在你的具体应用中尝试两种不同的方式，以了解哪个到底更好。

## 比较各种Map实现

我们来比较一下synchronizedHashMap和ConcurrentHashMap在性能方面的差异：

[

参考代码: optimi\ MapComparisons.java

测试结果：

Type Read time Write time

Synched HashMap 10r 0w 306052025049 0

Synched HashMap 9r 1w 428319156207 47697347568

readTime + writeTime = 476016503775

Synched HashMap 5r 5w 243956877760 244012003202

readTime + writeTime = 487968880962

ConcurrentHashMap 10r 0w 23352654318 0

ConcurrentHashMap 9r 1w 18833089400 1541853224

readTime + writeTime = 20374942624

ConcurrentHashMap 5r 5w 12037625732 11850489099

readTime + writeTime = 23888114831

]

向ConcurrentHashMap添加写入的影响甚至还不如CopyOnWriteArrayList明显，这是因为ConcurrentHashMap使用了一种不同的技术，它可以明显地最小化写入所造成的影响。

## 乐观加锁

尽管Atomic对象执行像decrementAndGet()这样的原子操作，但是某些Atomic类还允许你执行所谓的乐观加锁。这意味着当你执行某项计算时，实际上没有使用互斥，但是在这个对象计算完成并且准备更新这个对象时，你需要使用一个compareAndSet()的方法。你将旧值和新值一起提交给这个方法，如果不一样，那么这个操作失败。这意味着某个地方的任务在这个操作期间修改了这个对象。但是我们是乐观的，因为我们保持数据在未锁定的状态，并希望没有任何其他的任务插入修改它。通过使用Atomic来替代synchronized或lock，可以获得性能上的好处。

如果compareAndSet()操作失败会怎么样呢？这是一个棘手的问题，也是应用这项技术受限的条件，即只能针对处理相同条件下的问题。如果失败就必须决定做些什么。因为不能执行某些恢复操作，那么你就不能使用这项技术。

下面看一个示例，一旦你运行该程序会发现它变慢，并开始应用性能调优技术：

[

参考代码: optimi\ FastSimulation.java

]

所有的元素都被置于数组内，这被认为有助于提高性能。每个Evolver对象会用它前一个元素和后一个元素来平均它的值，如果在更新时失败，那么将直接打印这个值并继续执行。注意，在上面的程序中并没有出现任何互斥。

## ReadWriteLock

ReadWriteLock对于那种向数据结构中不频繁的写入，但是有多个任务要经常读取这个数据结构的情况进行了优化。ReadWriteLock使得你可以同时有多个读取者，只要它们都不试图写入即可。如果写锁已经被其它任务持有，那么任何读取者都不能访问，直至这个写锁被释放。

ReadWriteLock这样就能提高程序的性能吗？这是不确定的，这取决于数据被读的频率和修改的频率比较的结果、读取和写入操作的时间、有多少线程竞争者、是否是多处理器上运行等因素。只有实验才能证明你的程序是否被优化了。

下面是ReadWriteLock的最基本用法示例：

[

参考代码: optimi\ ReaderWriterList.java

]

ReaderWriterList可以持有固定数量的任何类型的对象。你必须向构造器提供所希望的列表尺寸和组装这个列表时所用的初始对象。set()方法要获取一个写锁，以调用底层的arrayList.set()，而get()方法要获取一个读锁，以调用底层的arrayList.get()。另外，get()方法将检查是否已经有多个读者获取了锁，如果是，则将显示这种读取者的数量，以证明可以有多个读取者获取锁。

如果查看JDK文档ReentrantReadWriteLock就会发现有大量的其他方法可用，涉及公平性和政策性等问题。这是一个相当复杂的工具，只有当你在考虑提高性能的方法时，才应该使用它。你的程序第一个方案应该考虑更直观的同步。

## 活动对象

多线程非常复杂并且难以正确使用，每个细节都很重要，我们需要处理每个细节，没有任何编译器检查形式的安全防护。是多线程模型出了问题吗？毕竟它来自于过程性编程世界，并且几乎没有任何改变。

有一种可替代的方式被称为活动者或行动者。之所以称这些对象是活动的，是因为每个对象都维护着它自己的工作器线程和消息队列，并且所有这种对象的请求都将进入队列排队，任何时刻都只能运行其中一个。因此，有了活动对象，我们就可以串行化消息而不是方法，这意味着不再需要防备一个任何在其循环的中间被中断这种问题。当你向一个活动对象发送消息时，这条消息将被转化为一个任务，该任务会被插入到这个对象队列中，等待在以后的某个时刻运行。Java SE5的Future在实现这种模式时将派上用场。

下面是一个示例，有两个方法，可以将方法调用排进队列：

[

参考代码：optimi\ ActiveObjectDemo.java

执行结果：

All asynch calls made

starting 0.0 + 0.0

0.0

starting 0.2 + 0.2

0.4

starting 0.4 + 0.4

0.8

starting 0.6 + 0.6

1.2

starting 0.8 + 0.8

1.6

staring0+0

0

staring1+1

staring2+2

2

4

staring3+3

6

staring4+4

8

]

newSingleThreadExecutor()的调用产生的单线程执行器维护着它自己的无界阻塞队列，并且只有一个线程从该队列中取走任务并执行他们直至完成。我们需要在calculateInt()和calculateFloat()中做的就是submit()提交一个新的Callable对象，以响应对这些方法的调用，这样就可以把方法转变为消息，而submit()的方法体包含在匿名内部类中。注意，每个活动对象的方法返回值都是一个具有泛型参数的Future，而这个泛型参数就是该方法的实际的返回类型。通过这种方式，方法调用可以立即返回，调用者可以通过Future来发现任务何时完成，并收集实际的返回值。

main()方法中创建了一个List<Future<?>>来捕获由活动对象的calculateInt()和calculateFloat()消息返回的Future对象。对于每个Future，都是用isDone()来从这个列表中抽取的，这种方式使得当Future完成并且其结果被处理之后，就会从List中移除。注意，使用CopyOnWriteArrayList可以移除为了防止ConcurrentModificationException异常而复制List的这种需求。

为了能够防止在不经意间出现线程的耦合，任何传递给活动对象的方法调用的参数必须都是只读的其它活动对象，或者是不连接对象，即没有连接其它任何任务的对象。

有了活动对象：

1. 每个对象都可以拥有自己的工作器线程
2. 每个对象都将维护对它自己的域的全部控制权
3. 所有的活动对象之间的通讯都将以这些对象之间的消息形式发生
4. 活动对象之前的所有消息都要排队

由于从一个活动对象到另一个活动对象的消息只能被排队时的延迟所阻塞，并且因为这个延迟总是非常短暂的而且独立于任何其他对象，所以发送消息实际上是不可阻塞的。由于一个活动对象系统只是经由消息来通信，所以两个对象在竞争调用另一个对象上的方法时，是不会被阻塞的，而这意味着不会发生死锁。这是一种巨大的进步，因为在活动对象的工作区线程在任何时刻只执行一个消息，所以不存在任何资源竞争，而你也不用操心如何同步方法。同步仍旧会发生，但是它通过将方法调用排队，使得任何时刻都只能发生一个调用，从而将同步控制在消息级别上发生。

# 总结

这个系列介绍了Java并发编程设计的基础知识，我们需要理解Java并发编程的内容：

1. 可以运行多个独立的任务
2. 必须考虑当这些任务关闭时，可能出现的所有问题。
3. 任务可能会在共享资源上彼此干涉。互斥（锁）是用来防止这种冲突的基本工具。
4. 如果任务设计的不够仔细，就可能会死锁。

明白什么时候使用并发、什么时候应该避免使用并发是非常关键的。使用它的主要原因是：

1. 要处理很多任务，它们交织在一起，应用并发能够有效地使用计算机
2. 要能够更好地组织代码
3. 要更便于用户使用

多线程的主要缺陷：

1. 等待共享资源的时候性能降低
2. 需要处理线程的额外CPU花费
3. 糟糕的程序设计导致不必要的复杂性
4. 有可能产生一些病态行为，如：饿死、竞争、死锁和活锁（多个任务各自运行的线程使得整体无法完成）
5. 不同平台导致的不一致性。

因为线程可能共享资源，比如一个对象的内存，而且你必须确定多个线程不会同时读取和改变这个资源，这就是线程锁产生的最大问题。这需要明智地使用可用的加锁机制，它们仅仅是工具，同时也可能引入潜在的死锁条件，所以要对它们理解透彻。