# Java中的进程与线程

学习目标：

1、进程和线程

2、线程的创建和启动

3、线程的状态

4、线程的调度

5、线程的同步

6、线程安全的类型

## 一：进程与线程

概述：几乎任何的操作系统都支持运行多个任务，通常一个任务就是一个程序，而一个程序就是一个进程。当一个进程运行时，内部可能包括多个顺序执行流，每个顺序执行流就是一个线程。

进程：进程是指处于运行过程中的程序，并且具有一定的独立功能。进程是系统进行资源分配和调度的一个单位。当程序进入内存运行时，即为进程。

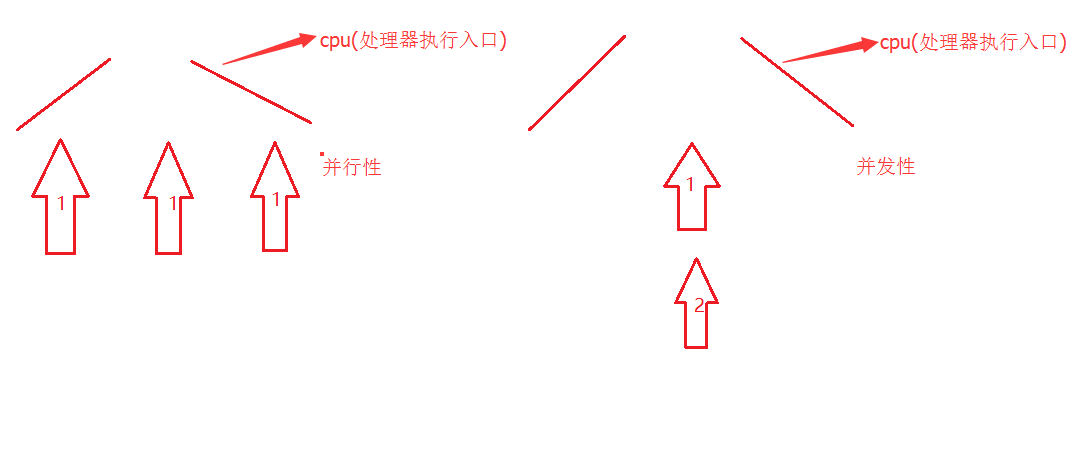
进程的三个特点：

1：独立性：进程是系统中独立存在的实体，它可以独立拥有资源，每一个进程都有自己独立的地址空间，没有进程本身的运行，用户进程不可以直接访问其他进程的地址空间。

2：动态性：进程和程序的区别在于进程是动态的，进程中有时间的概念，进程具有自己的生命周期和各种不同的状态。

3：并发性：多个进程可以在单个处理器上并发执行，互不影响。

并发性和并行性是不同的概念：并行是指同一时刻，多个命令在多个处理器上同时执行；并发是指在同一时刻，只有一条命令是在处理器上执行的，但多个进程命令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果



线程：线程是进程的组成部分，一个进程可以拥有多个线程，而一个线程必须拥有一个父进程。线程可以拥有自己的堆栈，自己的程序计数器和自己的局部变量，但不能拥有系统资源。它与父进程的其他线程共享该进程的所有资源。

线程的特点：

**线程可以完成一定任务，可以和其它线程共享父进程的共享变量和部分环境，相互协作来完成任务。**

**线程是独立运行的，其不知道进程中是否还有其他线程存在。**

**线程的执行是抢占式的，也就是说，当前执行的线程随时可能被挂起，以便运行另一个线程。**

**一个线程可以创建或撤销另一个线程，一个进程中的多个线程可以并发执行。**

## ****二：线程的创建及使用****

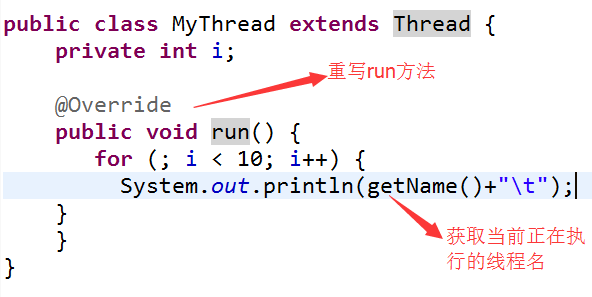
**java使用Thread类代表线程，所有的线程对象都必须是Thread或者其子类的实例，每个线程的作用是完成一定任务，实际上是就是执行一段程序流（一段顺序执行的代码）**

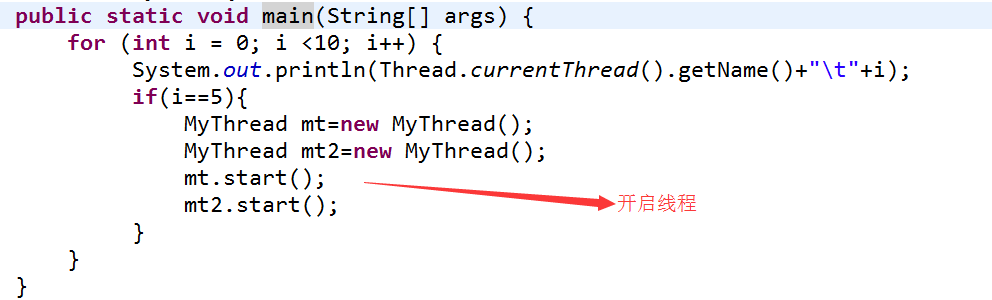
### ****方案一：继承Thread类创建线程类****

步骤：1.定义Thread类的子类 并重写该类的Run方法，该run方法的方法体就代表了该线程需要完成的任务

2.创建Thread类的实例，即创建了线程对象

3.调用线程的start方法来启动线程





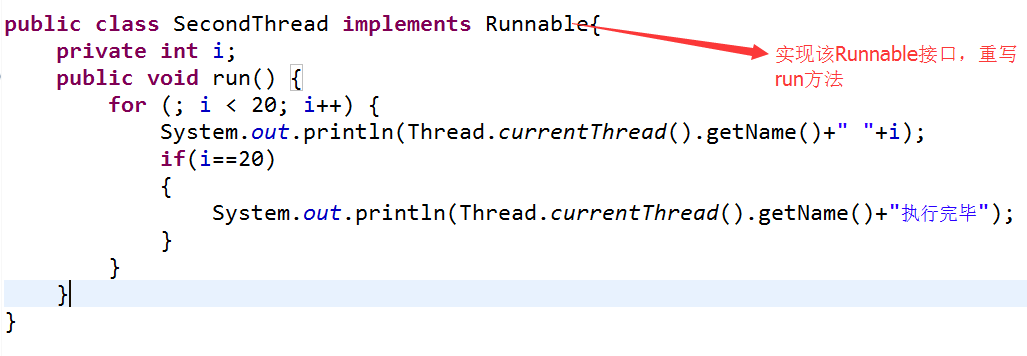
结论：使用继承子Thread类的子类来创建线程类时，多个线程无法共享线程类的实例变量(比如上面的i)

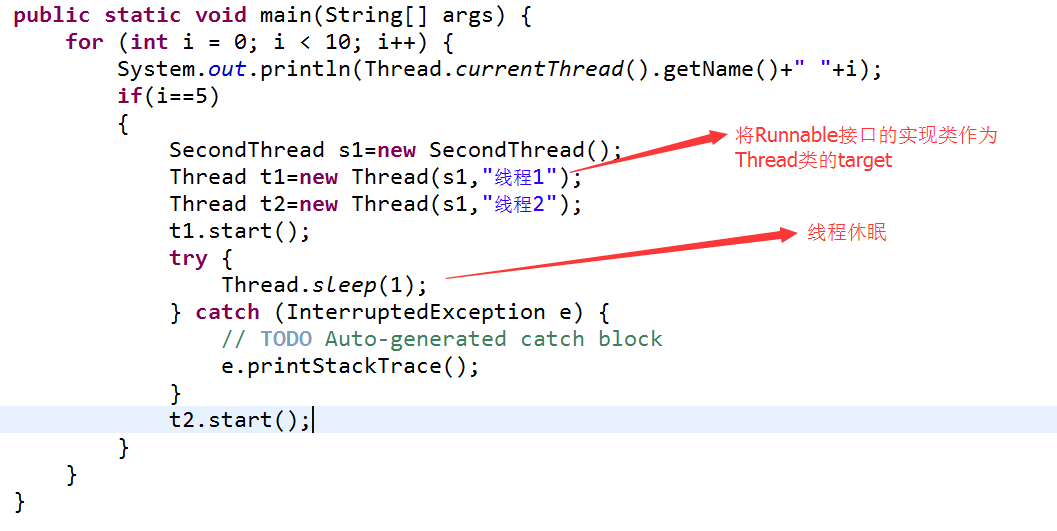
### 方案二：实现Runnable接口

1：定义Runnable接口的实现类，并重写它的Run方法，run方法同样是该线程的执行体！

2：创建Runnable实现类的实例，并将此实例作为Thread的target创建一个Thread对象，该Thread对象才是真正的线程对象！

3：调用start方法启动该线程





结论：采用Ruunable接口的方式创建多个线程可以共享线程类的实例变量，这是因为在这种方式下，程序创建的Runnable对象只是线程的target，而多个线程可以共享一个target，所以多个线程可以共享一个实例变量

通过Runnable实现多线程其实就是将run包装成线程的执行体，但是目前java无法将任意方法包装成线程执行体

### 方案三：使用callable和future创建线程

**从Java5开始，Java提供 Callable接口,Callable接口提供了一个call（）方法可以作为线程执行体，看起来和Runnable很像，但call（）方法更强大——call（）方法可以有返回值、call（）方法可以抛出异常**

Java5提供了Future接口来代表Callable接口的call（）方法的返回值，并为Future接口提供了一个FutureTask实现类，该实现类实现类Future接口，也实现了Runnable接口——可以作为Thread的target。

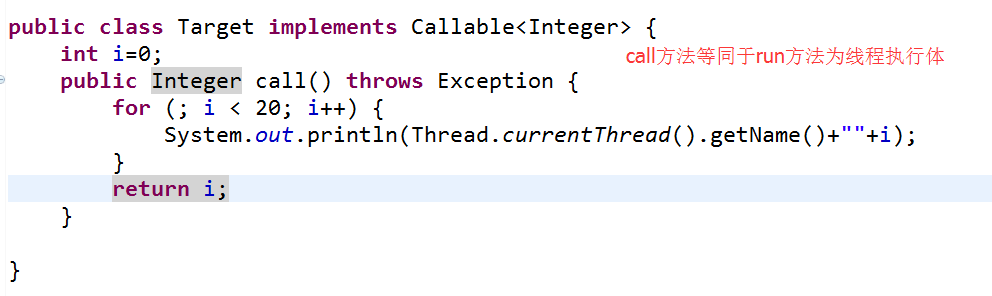
实现步骤：

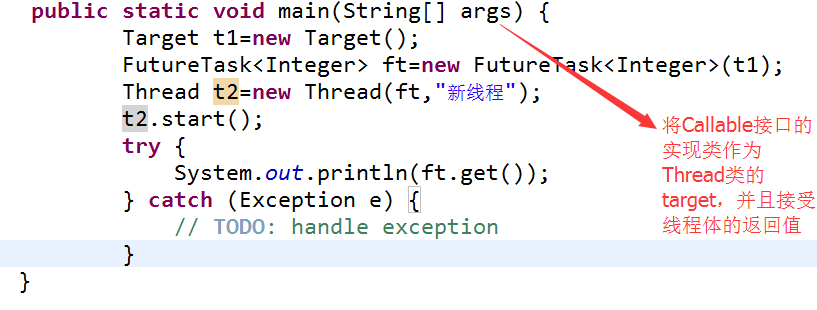
1：创建Callable接口的实现类，并实现call方法，该call方法会成为线程执行体，且call方法具有返回值，在创建callable接口的实现类！

2：使用FutrueTask类来包装Callable对象，该FutrueTask封装类Callable的call方法的返回值

3：使用FutrueTask对象作为Thread的target创建并启动新线程！

4：使用FutrueTask的get方法获取执行结束后的返回值





结论：采取Runnable、Callable的优势在于——线程类只是实现了Runnable或Callable接口，还可以继承其它类；在这种方法下，多个线程可以共享一个target对象，因此非常适合多个相同线程处理同一份资源的情况，从而将CPU、代码和数据分开，形参清晰的模型，体现了面对对象的编程思想。劣势在于编程复杂度略高。

## 三：线程的状态

当线程被创建并被启动时，它既不是一启动就进入了执行状态，在线程的生命周期中，它要经过new(新建)，就绪（Runnable），运行（Running），阻塞(Blocked)，dead(死亡)。

当线程启动之后，它不可能一直霸占着cpu独自运行，所有cpu需要在多条线程轮流切换，于是线程就也会多次在运行.就绪之间切换

### 1：新建和就绪状态

--新建状态：

当程序使用new关键字创建了一个线程时，该线程就处于新建状态。

此时的它和其它java对象一样，仅有虚拟机分配内存，并初始化成员变量的值。此时的线程对象并没有表现出线程的任何动态特征，程序也不会执行线程的线程执行体

--就绪状态：

当线程对象调用了start()方法后，该线程就处于就绪状态，java虚拟机会为其创建方法调用栈和程序计数器，处于该状态的线程并没有开始执行，只是表明该线程可以运行了，至于该线程何时运行，取决于JVM的调度。

注意！！！

启动线程要调用start方法，而不是run方法，永远不要调用线程的run方法，如果调用run方法，系统会把线程对象当作普通的对象，会吧线程的执行体当作普通方法来调用！在调用了run方法之后，该线程就不在处于新建状态，不要再调用该线程的start方法！java中只能对处于新建状态的线程使用start方法，否则将会引发IllegalThreadStateException异常！

### 2：运行状态和阻塞状态

当发生如下的几种情况时，将会进入阻塞状态：

当线程调用sleep方法主动放弃所占用的处理器资源

线程调用了一个阻塞时的IO方法，在该方法返回之前，线程会被阻塞

线程试图获得一个同步监视器，但该同步监视器正被其他线程锁持有

线程正在等待某个通知（notify）

程序调用了线程的suspend方法将该线程挂起

当以上几个情况，当发生如下的情况将会重新进入就绪状态

**调用sleep（）方法过了指定时间**

**线程调用的阻塞时IO方法依旧返回**

**线程成功地获得了试图获得的同步监视器**

**现在正在等待某个通知，而其它线程发出一个通知**

**处于挂起状态的线程被调用了resume（）方法**

**注意！！！**

线程从阻塞状态只能进入就绪状态，无法直接进入运行状态。就绪和运行状态之间的转换通常不受程序控制，而是系统线程的调度决定的。

调用yield()方法可以让处于运行时的线程转入就绪状态。

### 3：线程死亡

线程会以以下三种方式结束，结束后处于死亡状态

run或call方法执行完成，程序结束

线程抛出一个未捕获的Exception或者Error

直接调用该线程的stop方法来结束线程

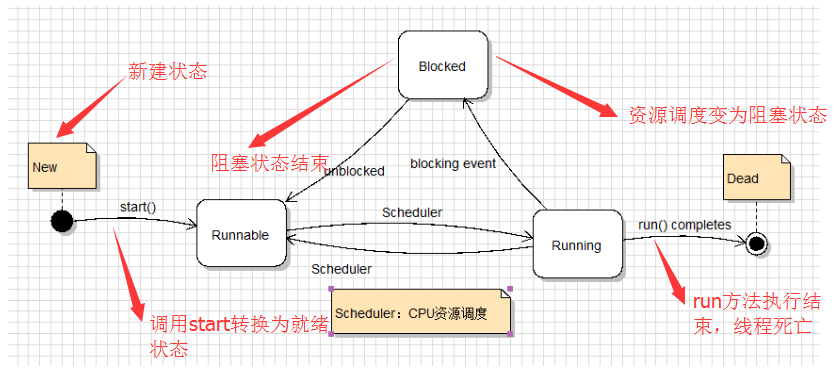
当主线程结束时，其它线程不受任何影响，并不会随之结束。一旦子线程启动起来后，他就会拥有和主线程相同的地位，它不会受主线程影响。

为了测试某个线程是否死亡，可以调用该线程的isAlive方法，当线程处于就绪，运行，阻塞三种状态时，将返回true；当线程处于新建，死亡两种状态时返回为false。

不要试图对一个已经死亡的线程调用start方法让它重新启动，死亡后的线程无法作为线程使用。

如果处于非新建状态的线程使用start方法，就会引发IllegalThreadStateException异常。

### 4：线程基本状态图

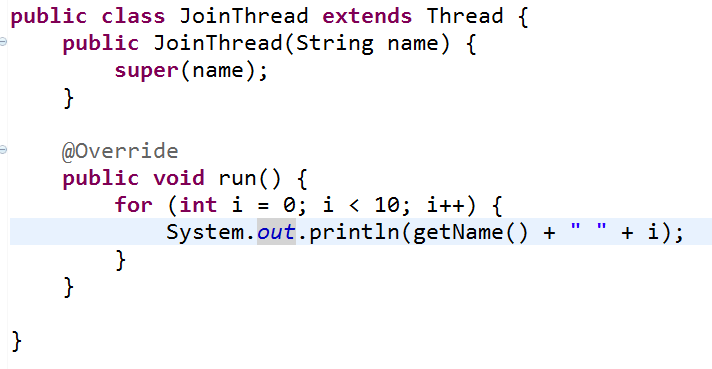


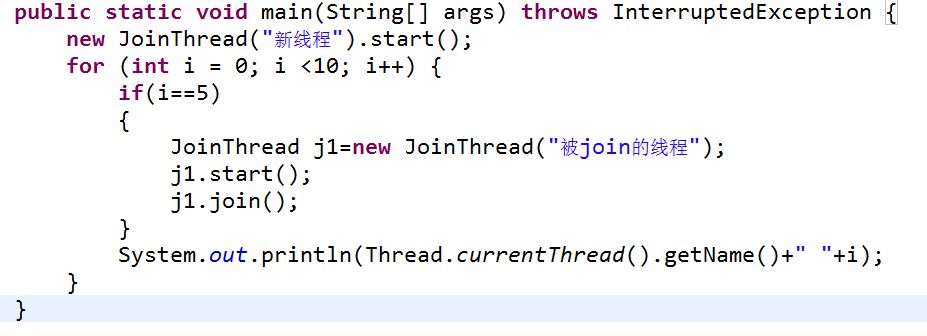
## 四：线程的调度

### join线程

Thread提供了让一个线程等待另一个线程完成的方法--join方法，当在某个程序执行流中调用其他线程的join方法，调用线程将被阻塞，直到被join方法加入的join线程执行完毕为止。

join方法应用场景：通常由使用线程的程序调用，以将大问题划为许多小问题，每个小问题分配一个线程，当所有的小问题都被处理之后，再调用主线程进行下一步操作！





### 后台线程

有一种线程，是在后台运行的，其任务是为其他线程提供服务，这种线程称之为后台线程（Daemon Thread）,又称之为守护线程。jvm的垃圾回收器就是典型的后台进程。

后台线程的特点：

1.当前台线程全部死亡，后台线程会自动死亡

2.调用Thread的setDaemon(ture)方法可以将指定线程设置成为后台线程。

3.当整个虚拟机只剩下后台线程时，程序就没有运行的必要了，所有虚拟机将退出

4.Thread类还提供了一个isDaemon方法，用于指定该线程是否是后台线程！

5.前台创建的线程默认为前台线程，而后台创建的线程默认为后台线程。

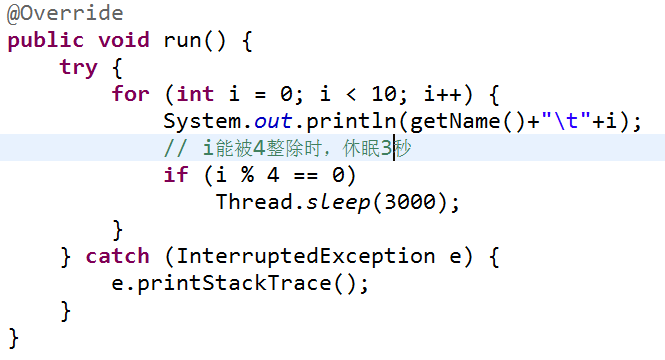
6.前台线程死亡时，jvm会通知后台线程死亡，但它从接受指令到做出响应需要一段时间 此外，如果要将某个线程设置为后台线程，必须要在该线程启动之前设置，也就是setDaemon(true)必须在start方法之前调用，否则会引发IllegalThreadStateException异常。

Jvm垃圾回收机制：<http://blog.csdn.net/u012152619/article/details/46981643>

### 3.线程休眠(sleep)

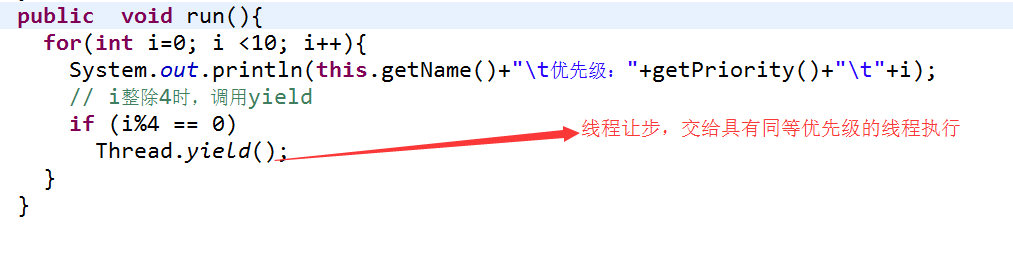
当前线程调用sleep方法进入阻塞状态时，在其睡眠时间内，该线程不会获得执行的机会

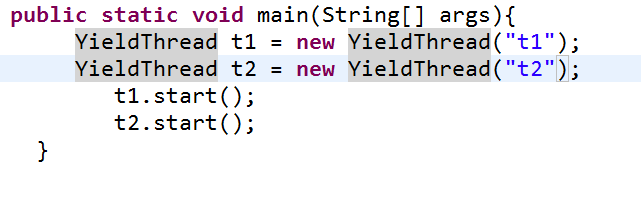
即便系统中没有其它可执行的线程，处于sleep的线程也不会执行，因此sleep方法常用于暂停程序的执行！



### 4:线程让步

yield()的作用是让步。它能让当前线程由“运行状态”进入到“就绪状态”，从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保证在当前线程调用yield()之后，其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！





在Cpu并行执行的情况下，效果不明显

### sleep()方法和yield方法的区别：

1.sleep方法暂停当前线程后会给其它线程执行机会，不会理会其它线程的优先级，但yield方法之后给优先级相同，或优先级更高的线程执行机会。

2.sleep方法会将线程转入阻塞状态，直到经过阻塞时间才会转为就绪状态，而yield方法不会转入阻塞状态，只是强制将当前线程转入就绪状态

3.sleep方法声明抛出了InterruptedException异常，所有调用sleep方法就要捕获此异常，而yield方法则没有

4.sleep方法比yield方法有更好的执行！

### 6：改变线程的优先级：

每个线程都有一定的优先级，优先级更高的线程将会有更多的执行机会

每个线程默认的优先级都与创建它的父进程的优先级相同，默认情况下，main进程具有普通优先级

Thread类提供setPriority(int newPriority)和getPriority()方法来设置和返回线程的优先级其中setPriority参数是int类型，范围0到10之间

Thread类有三个静态常量：MAX\_PRIORITY :10  MIN\_PRIORITY :1 NORM\_PRIORITY:5

## 五：线程的同步

### 1.同步方法

obj:同步监视器，含义：线程开始执行同步代码块时，必须先获得对同步监听器的锁定

**任何时刻只能有一个线程可以获得对同步监视器的锁定，当同步代码块完成执行后，该线程会释放对该同步监视器的锁定。**

**虽然Java程序允许使用任何对象作为同步监听器，但通常推荐使用可能被并发访问的共享资源充当同步监视器。**

同步方法：

与同步代码块，Java的多线程安全支持还提供了同步方法，同步方法就是使用某个synchronized关键字修饰某个方法，则该方法被称为同步方法。

对于被synchroize修饰的方法(非static方法而言)，无需显示指定同步监视器，同步方法的同步监视器是this，也就是调用该方法调用的对象

**synchronized关键字可以修饰方法，可以修饰代码块，但不能修饰构造器、成员变量。**

线程安全的类具有如下特征：

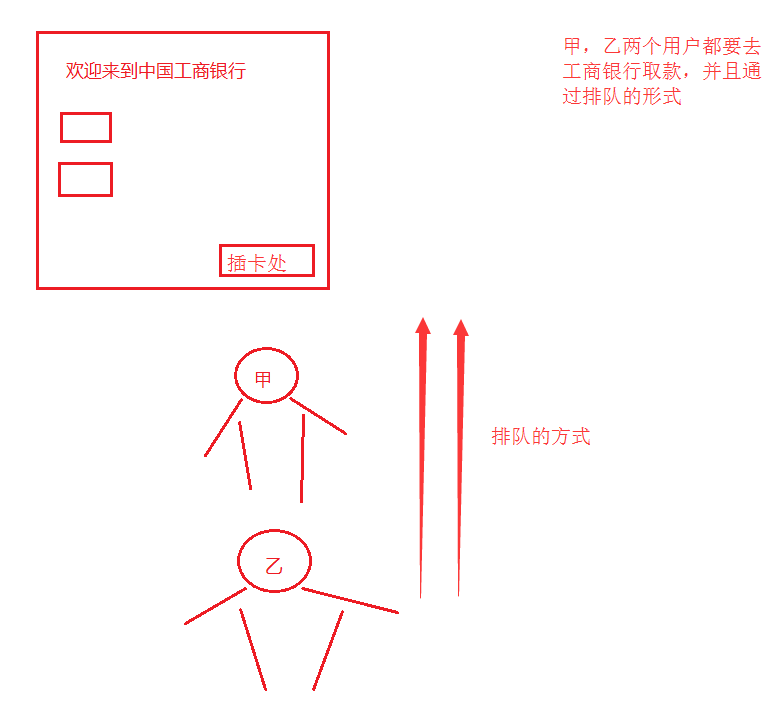
1：该类的对象可以被多个线程安全地访问

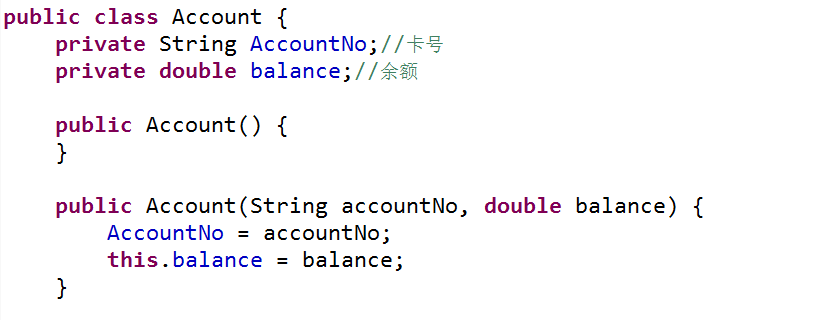
2：每个线程调用该对象的任一方法之后都将得到正确的结果

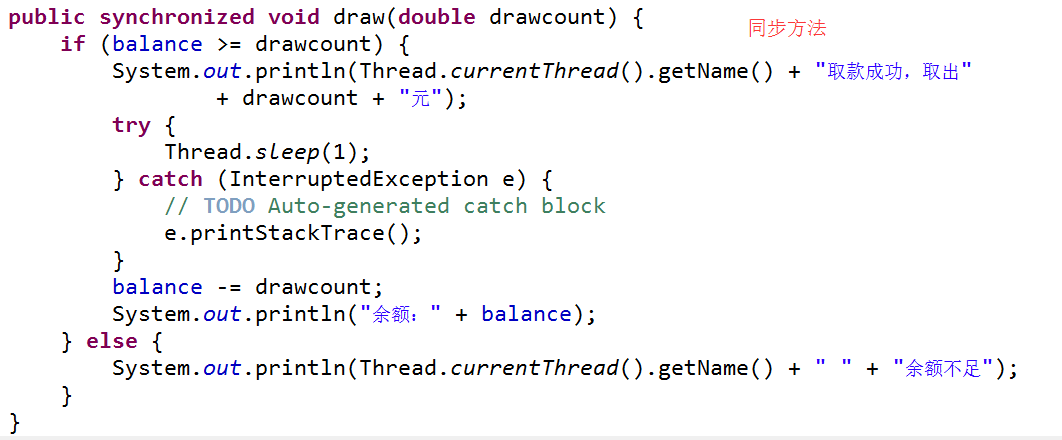
3：每个线程调用该对象的任一方法之后，该对象状态依然保持合理状态

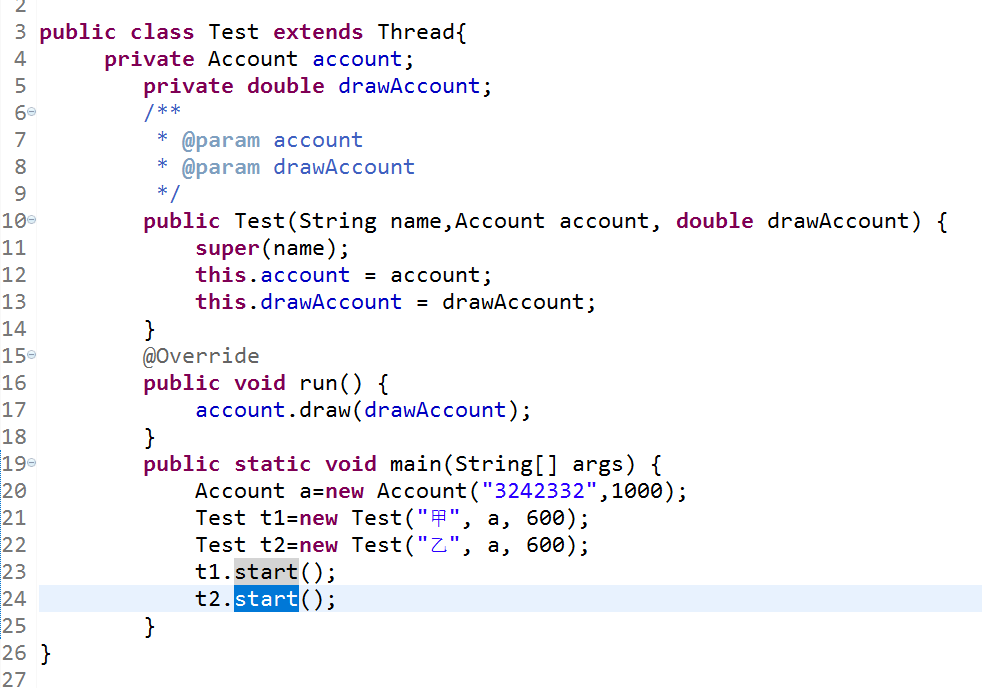
------取款案例：

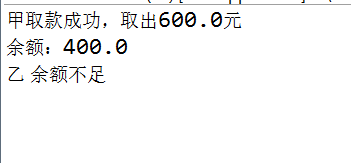
场景：









执行结果：

**可变类的线程安全是以降低运行效率为代价的，为减少线程安全锁带来的负面影响，可采用以下策略：**

**1：不要对线程安全类中的所有方法都同步，只对那些会改变竞争资源（竞争资源也就是共享资源）的方法进行同步。**

**2：如果可变类有单线程和多线程两种运行环境，那么要为该可变类提供两种版本（线程安全版和线程不安全版）**

例：

StringBuffer和StringBuilder就是这种情况，在单线程时应使用StringBuilder，多线程时使用StringBuffer。

**线程会在以下几种情况下释放对同步监听器的锁定：**

**当前线程的同步方法、同步代码块执行结束，当前线程释放了同步监听器。**

**当前线程在同步代码块、同步方法中遇到了break、continue，终止了该代码块、方法的运行，当前线程释放了同步监听器。**

**当前线程在同步代码块、同步方法中遇到了未处理的error、exception，导致该代码块、方法意外结束，当前线程释放了同步监听器。**

**当前线程执行同步代码块或同步方法时，程序执行了同步监听器对象的wait（）方法，则当前线程暂停，并释放同步监听器。**

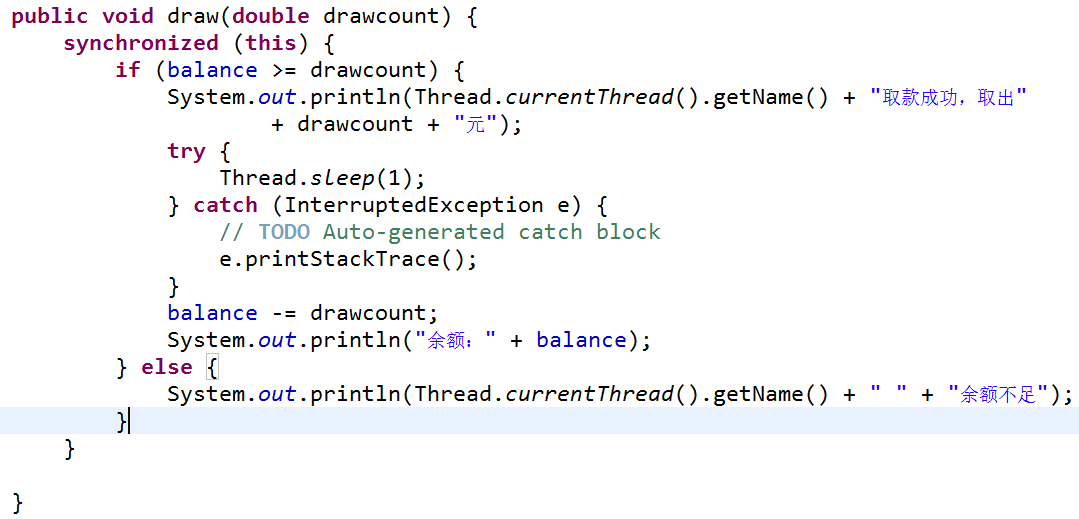
**在如下所示的情况下，线程不会释放同步监听器：**

**线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法来暂停当前线程的执行，当前线程不会释放同步监视器。**

**线程执行同步代码块时，其它线程调用了该线程的suspend（）方法将该方法挂起，该线程不会释放同步监听器。（所有程序应该避免使用suspend和resume来操控线程）**

### 2：同步锁

将取款案例的同步方法换为同步锁，效果是一样的！！



### 3:死锁

**1：当两线程相互等待对方释放同步监视器是就会发生死锁，JAVA虚拟机没有监测、处理死锁的措施，所以一定要避免死锁的出现。**

**2：一旦出现死锁，整个程序不会出现任何异常，也不会给出任何提示，只是所有线程处于阻塞状态，无法继续。**

**3：但是死锁很容易发生，尤其是系统中出现多个同步监听器的情况下。**

**4：由于Thread类的suspend（）方法也很容易导致死锁，所有JAVA不再推荐使用该方法来暂停线程的执行。**