多线程

线程和进程

- 几乎所有的操作系统都支持同时运行多个任务，一个任务通常就是一个程序，每个运行中的程序就是一个进程。当一个程序运行时，内部可能包含了多个顺序执行流，每个顺序执行流就是一个线程。
- 进程是系统进行资源分配和调度的一个独立单位。

- 进程特征：

独立性：进程是系统中独立存在的实体，它可以拥有自己独立的资源，每个进程都拥有自己私有的地址空间。在没有经过进程本身允许的情况下，一个用户进程不可以直接访问其他进程的地址空间。

动态性：进程与程序的区别在于，程序只是一个静态的指令集合，而进程是一个正在系统中活动的指令集合。在进程中加入了时间的概念。进程具有自己的生命周期和各种不同的状态，这些概念在程序中都是不具备的。

并发性：多个进程可以在单个处理器上并发执行，多个进程之间不会互相影响。

线程和进程

- 对于CPU而言，它在某个时间点只能执行一个程序，也就是说，只能运行一个进程，CPU不断在这些进程之间轮换执行。因为CPU的执行速度相对人的感觉来说实在是太快了（如果启动的程序足够多，用户依然可以感觉到程序的运行速度下降），所以虽然CPU在多个进程之间轮换执行，但用户感觉好像有多个进程在同时执行。
- 注意：并发性和并行性是两个概念，并行指在同一时刻，有多条指令在多个处理器上同时执行；并发指在同一时刻只能有一条指令执行，但多个进程指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。

线程和进程

- 多线程则扩展了多进程的概念，使得同一个进程可以同时并发处理多个任务。线程也被称作轻量级进程，线程是进程的执行单元。就像进程在操作系统中的地位一样，线程在程序中是独立的、并发的执行流。当进程被初始化后，主线程就被创建了。对于绝大多数的应用程序来说，通常仅要求有一个主线程，但也可以在该进程内创建多条顺序执行流，这些顺序执行流就是线程，每个线程也是相互独立的。
- 线程是进程的组成部分，一个进程可以拥有多个线程，一个线程必须有一个父进程。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，但不拥有系统资源。它与父进程的其他线程共享该进程所拥有的全部资源。因为多个线程共享父进程的全部资源，因此编程更加方便；但必须更加小心，要确保线程不会妨碍同一进程里的其他线程。
- 线程是独立运行的，它并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，当前运行的线程在任何时候都可能被挂起，以便另外一个线程可以运行。
- 一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。
- 归纳：一个程序运行后，至少有一个进程，一个进程里可以包含多个线程，但至少要包含一个线程。操作系统可以同时执行多个任务，每个任务就是进程；进程可以同时执行多个任务，每个任务就是线程。

线程和进程

- 多线程编程的优点：
- 进程之间不能共享内存，但线程之间共享内存非常容易。
- 系统创建进程时，需要为该进程重新分配系统资源，但创建线程则代价小的多。因此，使用多线程来实现多任务并发比多进程的效率高。
- Java语言内置了多线程功能支持，而不是单纯地作为底层操作系统的调度方式，从而简化了Java多线程编程。

- 实际应用：

一个浏览器必须能同时下载多个图片；一个Web服务器必须能同时响应多个用户请求；Java虚拟机本身就在后台提供了一个超级线程来进行垃圾回收……

线程的创建与启动

- 继承Thread类创建线程类（**FirstThread**）

1. 定义Thread类的子类，并重写该类的run()方法,run()方法的方法体就代表了线程需要完成的任务。因此把run()方法称为线程执行体。
2. 创建Thread子类的实例，即创建了线程对象。
3. 调用线程对象的start()方法来启动该线程。

线程的创建和启动

- 关于FirstThread:

  程序的主方法中包含了一个循环，当循环变量i等于20时，创建并启动两个新线程。

  两个线程的循环变量不连续，表明他们没有共享数据。

  虽然上面程序只显式地创建并启动了2个线程，但实际上程序有3个线程，即程序显式创建的2个子线程和主线程。当Java程序开始运行后，程序至少会创建一个主线程，主线程的线程执行体不是由run()方法确定的，而是由main()方法确定的—main()方法的方法体代表主线程的线程执行体。

  程序可以通过setName(String name)方法为线程设置名字，也可以通过getName()方法返回指定线程的名字。在默认情况下，主线程的名字为main，用户启动的多个线程的名字依次为Thread-0、Thread-1、Thread-2……Thread-n等。

注意：使用继承Thread类的方法来创建线程类时，多个线程之间无法共享线程类的实例变量。i变量是FirstThread的实例变量，而不是局部变量，但因为程序每次创建线程对象时都需要创建一个FirstThread对象，所以Thread-0和Thread-1不能共享该实例变量。

线程的创建和启动

- 实现Runnable接口创建线程类（**SecondThread**）

1. 定义Runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。
2. 创建Runnable实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。
3. 调用线程对象的start()方法来启动该线程。

线程的创建和启动

- 关于SecondThread：

FirstThread和SecondThread创建线程对象的方式有所区别：前者直接创建Thread子类即可代表线程对象；后者创建的Runnable对象只能作为线程对象的target。

通过SecondThread可以看到，两个子线程的i变量是连续的，也就是采用Runnable接口的方式创建的多线程是可以共享线程类的实例变量。这是因为在这种方式下，程序所创建的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线程类（实际上应该是线程的target类）的实例变量。

线程的创建和启动

- 使用Callable和Future创建线程（**ThirdThread**）
- Callable接口提供了一个call()方法可以作为线程执行体，但call()方法比run()方法功能更强大。（call()方法可以有返回值，可以声明抛出异常）

1. 创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，且该call()方法有返回值，再创建Callable实现类的实例。从Java8开始，可以直接使用Lambda表达式创建Callable对象。
2. 使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。
3. 使用FutureTask对象作为Thread对象的target创建并启动新线程。
4. 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

线程的创建和启动

- 关于ThirdThread:

ThirdThread使用Lambda表达式直接创建了Callable对象，这样就无需先创建Callable实现类，再创建Callable对象了。实现Callable接口与实现Runnable接口并没有太大的差别，只是Callable的call()方法允许声明抛出异常，而且允许带返回值。

程序先使用Lambda表达式创建了一个Callable对象，然后将该实例包装成一个FutureTask对象。主线程中当循环变量i等于20时，程序启动以FutureTask对象为target的线程。程序最后调用FutureTask对象的get()方法来返回call()方法的返回值--该方法将导致主线程被阻塞，直到call()方法结束并返回为止。

线程的创建和启动

- 实现Runnable、Callable接口的方式创建线程优缺点：

优：

- 线程只是实现了接口，还可以继承其他类。
- 在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣：

- 编程稍稍复杂，如果需要访问当前线程，则必须使用Thread.currentThread()方法。

- 继承Thread类的方式创建多线程的优缺点：

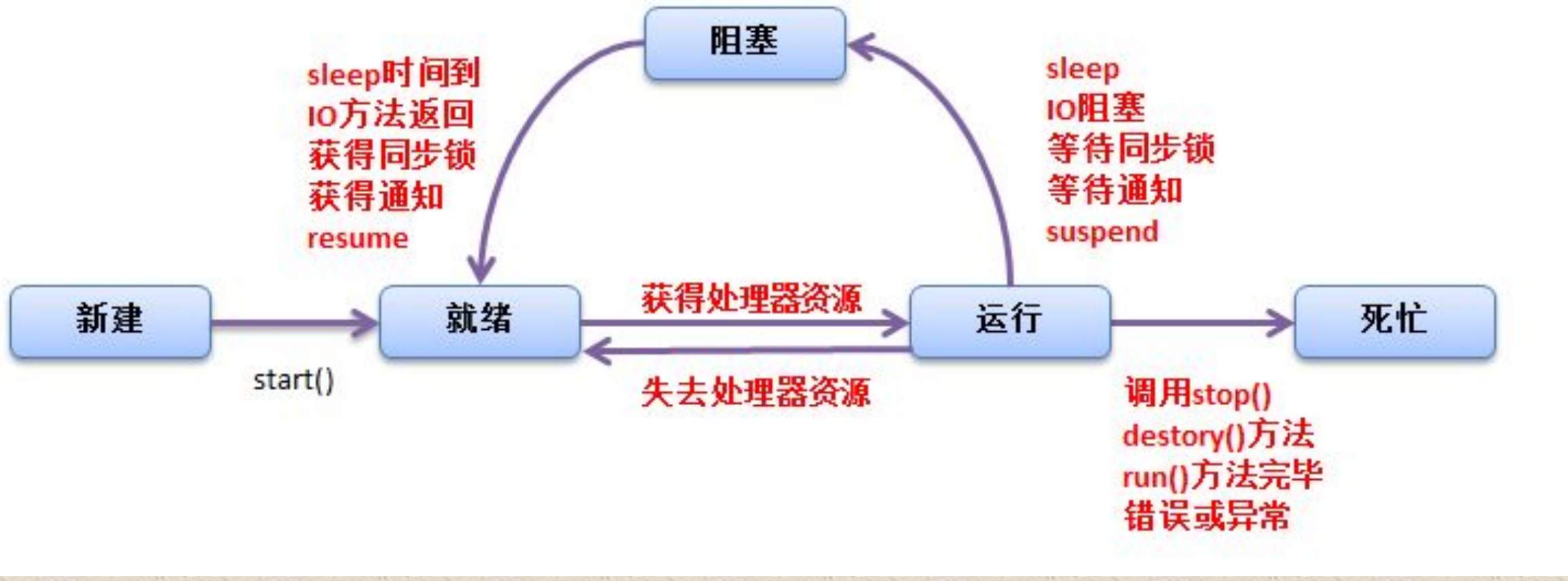
优：

- 编写简单，如果需要访问当前线程，直接使用this即可获得当前线程。

劣：

- 因为线程类已经继承了Thread类，所以不能再继承其他父类。

线程的生命周期


阻塞
sleep时间到
IO方法返回
获得同步锁
获得通知
resume
sleep
IO阻塞
等待同步锁
等待通知
suspend
新建
就绪
获得处理器资源
运行
死忙
start()
失去处理器资源
调用stop()
destory()方法
run()方法完毕
错误或异常

线程的生命周期

- 新建（New）

  当程序使用new关键字创建了一个线程之后，该线程就处于新建状态，此时它和其他的Java对象一样，仅仅由虚拟机为其分配内存，并初始化其成员变量的值。此时的线程对象没有表现出任何线程的动态特征，程序也不会执行线程的线程执行体。

- 就绪（Runnable）

  当线程对象调用了start()方法之后，该线程处于就绪状态。Java虚拟机会为其创建方法调用栈和程序计数器，处于这个状态中的线程并没有开始运行，只是表示该线程可以运行了。至于该线程何时开始运行，取决于JVM里线程调度器的调度。

注意：（**InvokeRun**）

启动线程使用start()方法，而不是run()方法。调用start()方法来启动线程，系统会把该run()方法当成线程执行体来处理；但如果直接调用线程对象的run()方法，则run()方法立即会被执行，而且在run()方法返回之前，其他线程无法并发执行。即，如果直接调用线程对象的run()方法，系统把线程对象当成一个普通对象，而run()方法也是一个普通方法，而不是线程执行体。

线程的生命周期

- 关于InvokeRun:

  整个程序只有一个线程，主线程。

线程的生命周期

- 运行（Running）

  如果处于就绪状态的线程获得了CPU，开始执行run()方法的线程执行体，则该线程处于运行状态，如果计算机只有一个CPU，那么在任何时刻只有一个线程处于运行状态。当然，在一个多处理器的机器上，将会有多个线程并行执行；当线程数大于处理器数时，依然会存在多个线程在同一个CPU上轮换的现象。

- 阻塞（Blocked）

  当一个线程开始运行后，它不可能一直处于运行状态（除非它的线程执行体足够短，瞬间就执行结束了），线程在运行过程中需要被中断，目的是使其他线程获得执行的机会，线程调度的细节取决于底层平台所采用的策略。对于采用抢占式策略的系统而言，系统会给每个可执行的线程一个小时间段来处理任务；当该时间段用完后，系统就会剥夺该线程所占的资源，让其他线程获得执行的机会。在选择下一个线程时，系统会考虑线程的优先级。

线程的生命周期

- 当发生如下情况时，线程会进入阻塞状态。

1. 线程调用sleep()方法，主动放弃所占用的处理器资源。
2. 线程调用了一个阻塞式IO方法，在该方法返回之前，该线程被阻塞。
3. 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所持有。
4. 县城在等待某个通知（notify）。
5. 程序调用了线程的suspend()方法将该程序挂起。但这个方法容易导致死锁。所以应该尽量避免使用该方法。

当前正在执行的线程被阻塞之后，其他线程就可以获得执行的机会，被阻塞的线程会在合适的时候重新进入就绪状态。被阻塞线程的阻塞解除后，必须重新等待线程调度器再次调度他。

线程的生命周期

- 当发生如下指定的情况时可以解除阻塞，让该线程重新进入就绪状态。

1. 调用sleep()方法的线程经过了指定的时间。
2. 线程调用的阻塞式IO方法已经返回。
3. 线程成功地获得了试图获取的同步监视器。
4. 线程正在等待某个通知时，其他线程发出了一个通知。
5. 处于挂起状态的线程被调用了resume()恢复方法。

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。而就绪和运行状态之间的转换，通常不受程序控制，而是由系统线程调度所决定。当处于就绪状态的线程获得处理器资源时，该线程进入运行状态；当处于运行状态的线程失去处理器资源时，该线程进入就绪状态。但有一个方法例外，调用yield()方法可以让运行状态的线程转入就绪状态。

线程的生命周期

- 死亡（Dead）

1. run()或call()方法执行完成，线程正常结束。
2. 线程抛出一个未捕获的Exception或Error。
3. 直接调用该线程的stop()方法来结束该线程--该方法容易导致死锁，通常不推荐使用。

不要试图对一个已经死亡的线程调用start()方法使它重新启动，死亡就是死亡，该线程将不可再次作为线程执行。（**StartDead**）

线程的生命周期

- 关于StartDead:

该代码试图在线程已死亡的情况下再次调用start()方法来启动该线程，运行时会引发IllegalThreadStateException异常，这表明处于死亡状态的线程无法再次运行了。

程序只能对新建状态的线程调用start()方法，对新建状态的线程两次调用start()方法也是错误的，这都会引发IllegalThreadStateException异常。

控制线程

- Join线程（**JoinThread**）

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()方法时，调用线程被阻塞，直到被join()方法加入的join线程执行完为止。

join()方法通常由使用线程的程序调用，以将大问题划分为许多小问题，每个小问题分配一个线程。当所有的小问题得到处理后，再调用主线程来进一步操作。

控制线程

- 关于JoinThread

程序中一共有3个线程，主方法开始时就启动了名为“新线程”的子线程，该子线程将会和main线程并发执行。当主线程的循环变量i等于20时，启动了名为“被join的线程”的线程，该线程不会和main线程并发执行，main线程必须等该线程执行结束后才可以向下执行。在名为“被join的线程”的线程执行时，实际上只有2个子线程并发执行，而主线程处于等待状态。

控制线程

- 后台线程（**DaemonThread**）

有一种线程，它是在后台运行的，它的任务是为其他线程提供服务。这种线程被称为“后台线程”，又被称为“守护线程”或“精灵线程”。JVM的垃圾回收线程就是典型的后台线程。

后台线程有个特征：如果所有的前台线程都死亡，后台线程会自动死亡。

调用Thread对象的setDaemon(true)方法可将指定线程设置为后台线程。当所有的前台线程死亡时，后台线程随之死亡。当整个虚拟机中只剩下后台线程时，程序就没有继续运行的必要了，所以，虚拟机也就退出了。

控制线程

- 关于DaemonThread:

先将t线程设置成后台线程，然后启动该线程，本来该线程应该执行到i等于999时才会结束，但运行程序时不难发现该后台线程无法运行到999，因为当主线程也就是程序中唯一的前台线程运行结束后，JVM会主动退出，因而后台线程也就被结束了。

前台线程死亡后，JVM会通知后台线程死亡，但从它接收指令到做出响应，需要一定时间。而且要将某个线程设置为后台线程，必须在该线程启动之前设置，也就是说，setDaemon(true)在start()方法之前调用，否则会引发IllegalThreadStateException异常。

Thread类还提供了一个isDaemon()方法，用于判断指定线程是否为后台线程。

控制线程

- 线程睡眠：sleep（**SleepTest**）

如果需要让当前正在执行的线程暂停一段时间，并进入阻塞状态。则可以通过调用Thread类的静态sleep()方法来实现。

当当前线程调用sleep()方法进入阻塞状态后，在其睡眠时间段内，该线程不会获得执行的机会，即使系统中没有其他可执行的线程，处于sleep()中的线程也不会执行，因此sleep()方法常用来暂停程序的执行。

控制线程

- 线程让步：yield（**YieldThread**）

该方法可以让当前正在执行的线程暂停，但它不会阻塞该线程，它只是将该线程转入就绪状态。yield()方法只是让当前线程暂停一下，让系统的线程调度器重新调度一次。完全可能的情况是：当某个线程调用了yield()方法暂停之后，线程调度器又将其调度出来重新执行。

实际上，当某个线程调用了yield()方法暂停之后，只有优先级与当前线程相同，或者优先级比当前线程更高的、处于就绪状态的线程才会获得执行的机会。

控制线程

- 关于YieldThread:

程序调用yield()静态方法让当前正在执行的线程暂停，让系统线程调度器重新调度。两个线程的优先级完全一样时，当一个线程使用方法暂停后，另一个线程就会开始执行。如果将两个线程分别设置不同的优先级，线程会将执行机会让给优先级要高的线程。

控制线程

- sleep()方法和yield()方法的区别：
- sleep()方法暂停当前线程后，会给其他线程执行机会，不会理会其他线程的优先级；但yield()方法只会给优先级相同、或优先级更高的线程执行机会。
- sleep()方法会将线程转入阻塞状态，直到经过阻塞时间才会转入就绪状态；而yield()方法不会将线程转入阻塞状态，它只是强制当前线程进入就绪状态。因此完全有可能某个线程调用yield()方法暂停之后，立即再次获得处理器资源被执行。
- sleep()方法声明抛出了IllegalThreadStateException异常，所以调用sleep()方法时，要么捕捉该异常，要么显式声明抛出该异常；而yield()方法则没有声明抛出任何异常。
- sleep()方法比yield()方法有更好的可移植性，通常不建议使用yield()方法来控制并发线程的执行。

线程同步

- 线程安全问题

银行取钱的基本流程：

1. 用户输入账户、密码，系统判断用户的账户密码是否匹配。
2. 用户输入值款金额。
3. 系统判断账户余额是否大于取款金额。
4. 如果余额大于取款金额，则取款成功；如果余额小于取款金额，则取款失败。

线程同步

- 线程安全（**DrawTest**）

模拟两个人使用同一个账户并取钱的问题。

先定义一个账户类，该账户类封装了账户编号和余额两个实例。

接下来提供一个取钱的线程类，该线程类根据执行账户、取钱数量进行取钱操作。取钱的逻辑是，当余额不足时无法提取现金；当余额足够时，系统吐出钞票，余额减少。

线程同步

- 同步代码块

之所以出现账户余额负值的结果，是因为run()方法的方法体不具有同步安全性—程序中有两个并发线程在修改Account对象。(当有两个进程并发修改同一个文件时就有可能造成异常)

为了解决这个问题，Java的多线程支持引入了同步监视器来解决这个问题，使用同步监视器的通用方法就是同步代码块。语法如下：

```
synchronized(obj){
    //此处的代码就是同步代码块
}
```

此处，obj就是同步监视器。代码的含义是：线程开始执行同步代码块之前，必须先获得对同步监视器的锁定。

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

线程同步

- 同步代码块（**DrawThread2**）

虽然Java程序允许使用任何对象作为同步监视器,但同步监视器的目的是，阻止两个线程对同一个共享资源进行并发访问。因此，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。对于取钱模拟程序，应该考虑使用账户（Account）作为同步监视器。

线程同步

- 同步方法（**AccountTest、DrawThreadTest**）

与同步代码块对应多线程安全支持，还提供了同步方法。能不方法就是使用关键字来修饰某个方法。则该方法称为同步方法。

例如上面的Account就是一个可变类，它的accountNo和balance两个成员变量都可以被改变。当两个线程同时修改Account对象的balance成员变量的值时，程序就出现了异常。如果要将Account类对balance的访问设置成线程安全的，那么只要把修改balance的方法变成同步方法即可。

线程同步

- 同步锁（Lock）

从Java5开始，Java提供了一种功能更强大的线程同步机制--通过显式定义同步锁对象来实现同步。在这种机制下，同步锁由Lock对象充当。

Lock是控制多个线程对共享资源进行访问的工具。通常，锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。

在实现线程安全的控制中，比较常用的是ReentrantLock(可重入锁)。使用该Lock对象可以显式的加锁、释放锁。

线程同步

- 同步锁（Lock）（**AccountLock**）

使用ReentrantLock对象来进行同步，加锁和释放锁出现在不同的作用范围内时，通常建议使用finally块来确保在必要时释放锁。

代码格式如下：

```
class X {
    //定义锁对象
    private final ReentrantLock lock = new
ReentrantLock();
    //定义需要保证线程安全的方法
    public void m(){
        //加锁
        lock.lock();
        try {
            //需要保证线程安全的代码
            //...method body
        }
        //使用finally块来保证释放锁
        finally{
            lock.unlock();
        }
    }
}
```