内容目录

《Android进阶之光》

•New: 新创建状态。线程被创建，还没有调用 start 方法，在线程运行之前还有一些基础工作要做。

•Runnable: 可运行状态。一旦调用start方法，线程就处于Runnable状态。一个可运行的线程可能正在运行也可能没有运行，这取决于操作系统给线程提供运行的时间。

•Blocked: 阻塞状态。表示线程被锁阻塞，它暂时不活动。

•Waiting: 等待状态。线程暂时不活动，并且不运行任何代码，这消耗最少的资源，直到线程调度器重新激活它。

•Timed waiting: 超时等待状态。和等待状态不同的是，它是可以在指定的时间自行返回的。

•Terminated: 终止状态。表示当前线程已经执行完毕。导致线程终止有两种情况：第一种就是run方法执行完毕正常退出；第二种就是因为一个没有捕获的异常而终止了run方法，导致线程进入终止状态。

# 第4章 多线程

## 4.1 线程基础

1. 继承Thread类，重写run()方法

2. 实现Runnable接口，并实现该接口的run()方法

3. 实现Callable接口，重写call()方法

Thread.currentThread().isInterrupted()

# 4.2 同步

### 4.2.1 重入锁与条件对象

重入锁ReentrantLock是JavaSE 5.0引入的，就是支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁

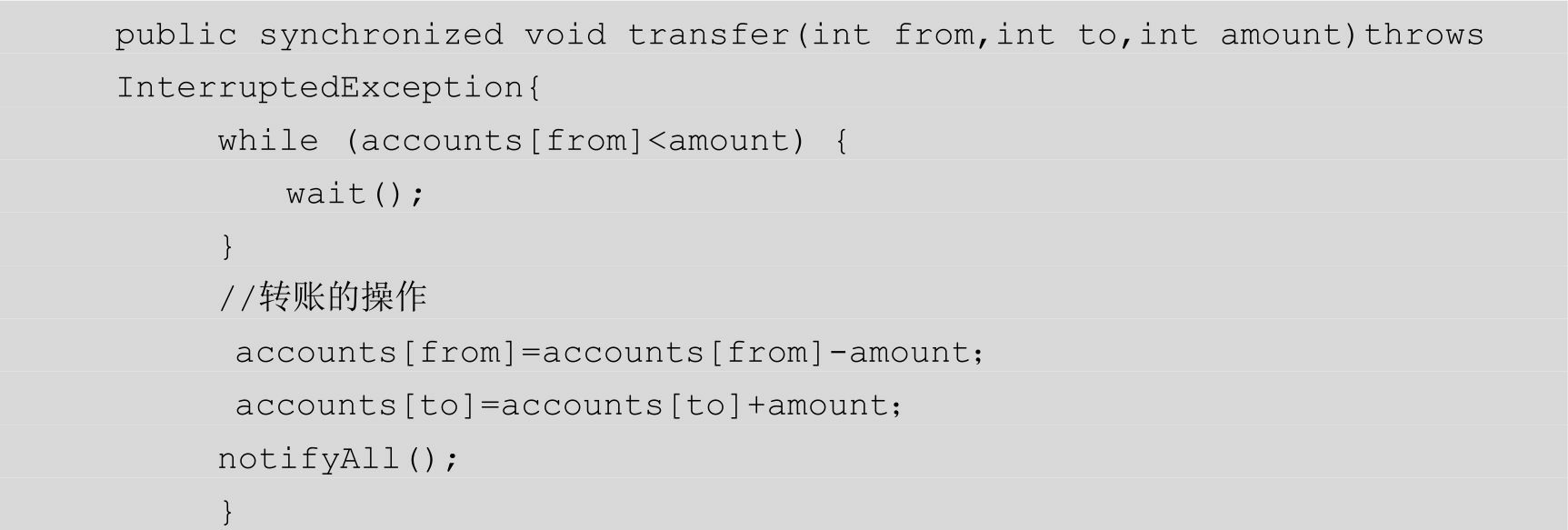


我们发现转账方余额不足；如果有其他线程给这个转账方再转足够的钱，就可以转账成功了。我们可以用newCondition方法获得一个条件对象,我们得到条件对象后调用await方法，当前线程就被阻塞了并放弃了锁。

当调用signalAll方法时并不是立即激活一个等待线程，它仅仅解除了等待线程的阻塞，以便这些线程能够在当前线程退出同步方法后，通过竞争实现对对象的访问。还有一个方法是signal，它则是随机解除某个线程的阻塞。如果该线程仍然不能运行，则再次被阻塞。如果没有其他线程再次调用signal，那么系统就死锁了。

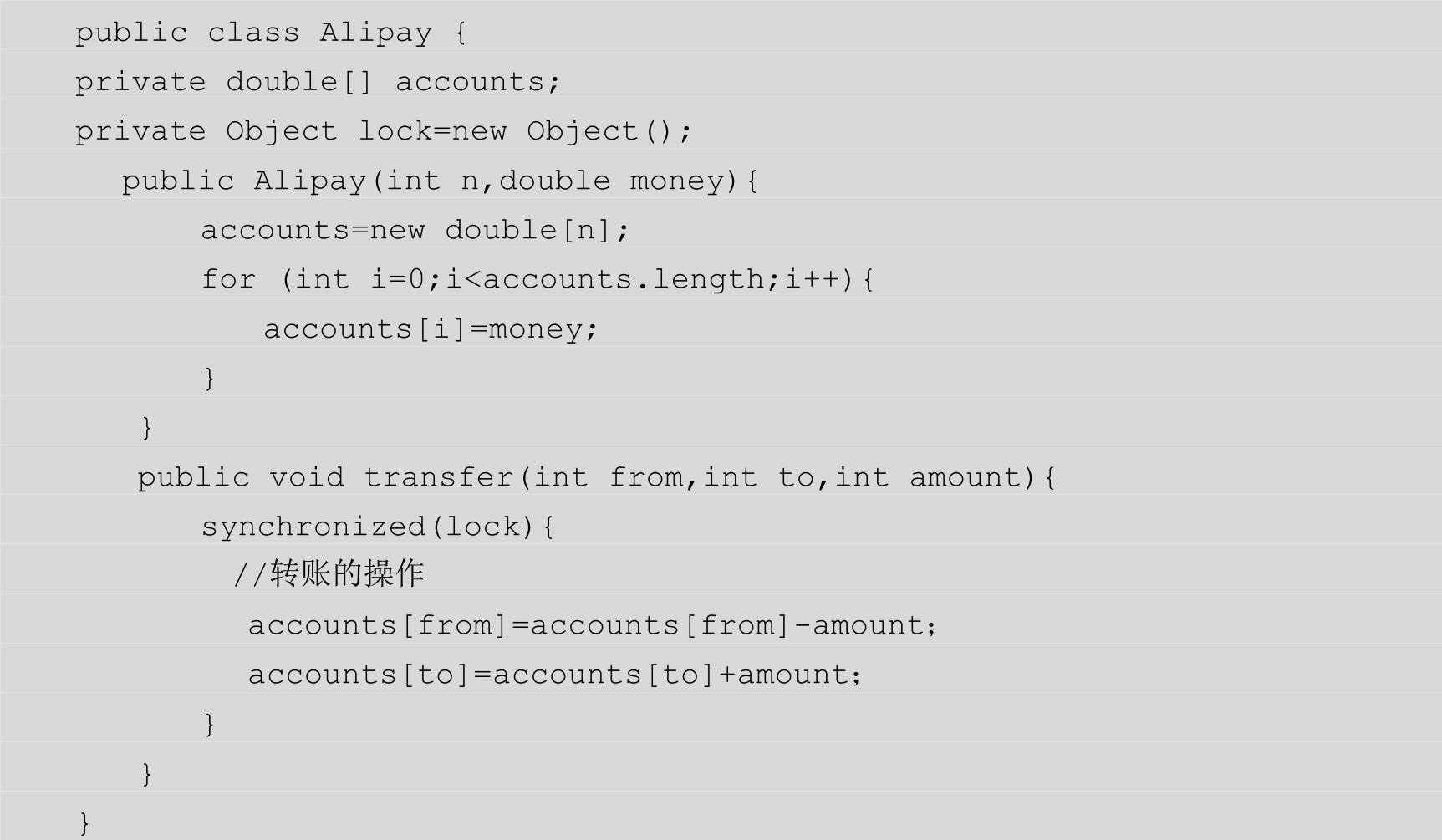
### 4.2.2 同步方法

使用synchronized关键字来可以替代ReentrantLock，来实现上一节中的方法



### 4.2.3 同步代码块

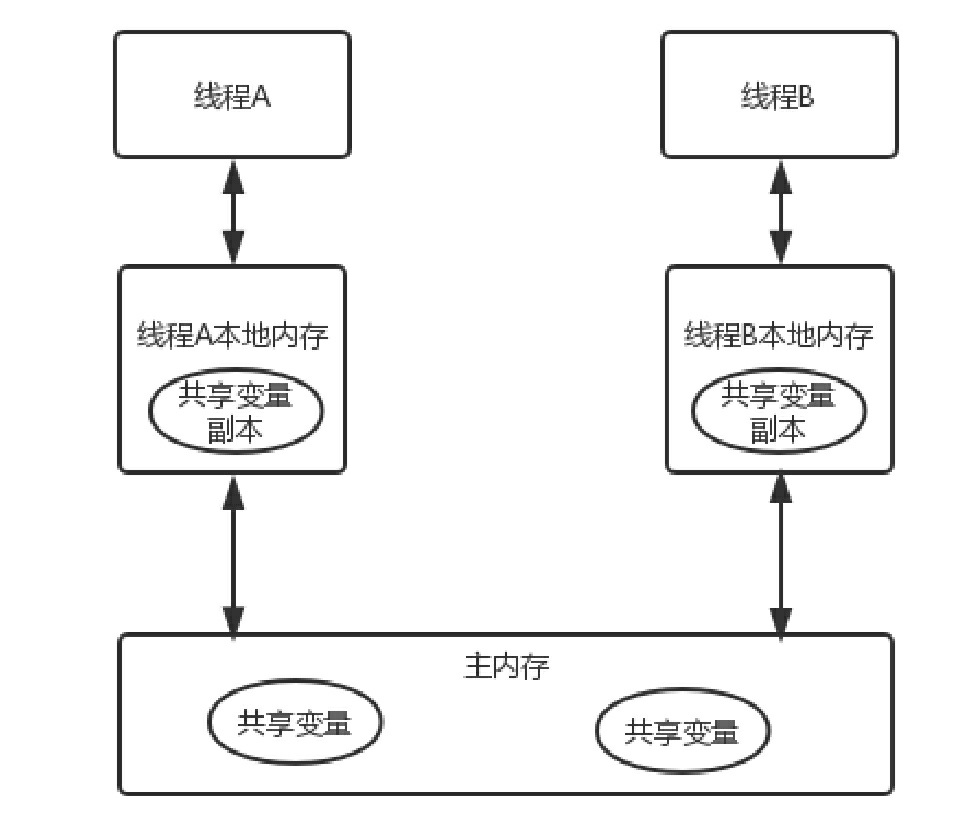
其获得了obj的锁，obj指的是一个对象。再来看看Alipay类，我们用同步代码块进行改写。

 在这里创建了一个名为Lock的Object类,为的是使用Object类所持有的锁。同步代码块是非常脆弱的，通常不推荐使用。一般实现同步最好用java.util.concurrent包下提供的类，比如阻塞队列。如果同步方法适合你的程序，那么请尽量使用同步方法，这样可以减少编写代码的数量，减少出错的概率。如果特别需要使用Lock/Condition结构提供的独有特性时，才使用Lock/Condition。

### 4.2.4 volatile

如果读写仅仅是一两个实例域的话，可以用volatile来代替同步(锁)，可以减小开销。

#### 4.2.4.1 Java内存模型

 线程A与线程B之间若要通信的话，必须要经历下面两个步骤:

(1) 线程A把线程A本地内存中更新过的共享变量刷新到主存中去。

(2) 线程B到主存中去读取线程A之前已更新过的共享变量。由此可见，如果我们执行下面的语句:

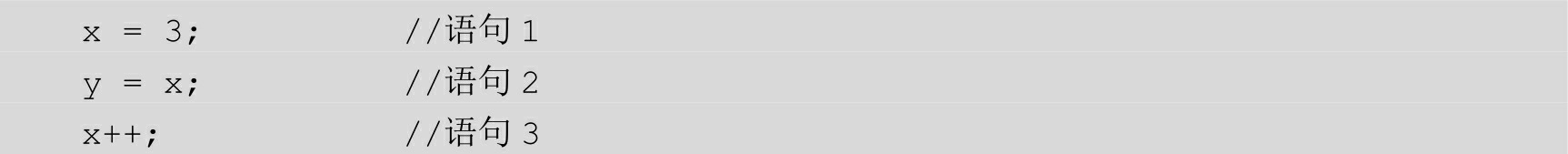
int i = 3;

执行线程必须先在自己的工作线程中对变量i所在的缓存行进行赋值操作，然后再写入主存当中，而不是直接将数值3写入主存当中。

#### 4.2.4.2 原子性、可见性和有序性

**(1) 原子性**

对基本数据类型变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的,要么执行完毕，要么就不执行。现在看一下下面的代码，如下所示:

 在上面3个语句中，只有语句1是原子性操作，其他两个语句都不是原子性操作。

语句2虽说很短，但它包含了两个操作，它先读取x的值，再将x的值写入工作内存。读取x的值以及将x的值写入工作内存这两个操作单拿出来都是原子性操作，但是合起来就不是原子性操作了。

语句3包括3个操作: 读取x的值、对x的值进行加1、向工作内存写入新值。

通过这3个语句我们得知，一个语句含有多个操作时，就不是原子性操作，只有简单地读取和赋值(将数字赋值给某个变量)才是原子性操作。java.util.concurrent.atomic 包中有很多类使用了很高效的机器级指令(而不是使用锁)来保证其他操作的原子性。例如AtomicInteger类提供了方法incrementAndGet和decrementAndGet，它们分别以原子方式将一个整数自增和自减。可以安全地使用AtomicInteger类作为共享计数器而无须同步。另外这个包还包含AtomicBoolean、AtomicLong和AtomicReference这些原子类，这仅供开发并发工具的系统程序员使用，应用程序员不应该使用这些类。

**(2) 可见性**

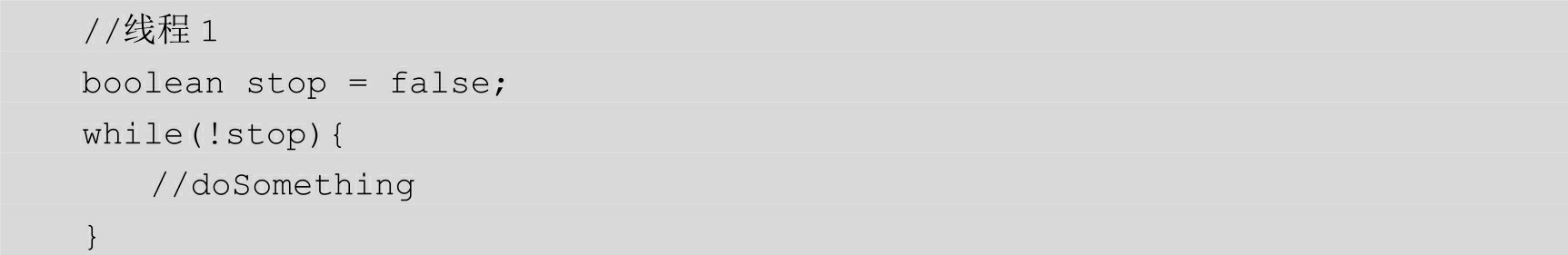
可见性是指线程之间的可见性，一个线程修改的结果，另一个线程马上就能看到。当一个共享变量被volatile修饰时，它会保证修改的值立即被更新到主存，所以对其他线程是可见的。当有其他线程需要读取该值时，其他线程会去主存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，并不会立即被写入主存，何时被写入主存也是不确定的。当其他线程去读取该值时，此时主存中可能还是原来的旧值，这样就无法保证可见性。

**(3) 有序性**

Java内存模型中允许编译器和处理器对指令进行重排序，虽然重排序过程不会影响到单线程执行的正确性，但是会影响到多线程并发执行的正确性。这时可以通过volatile来保证有序性，除了volatile，也可以通过synchronized和Lock来保证有序性。我们知道，synchronized和Lock保证每个时刻只有一个线程执行同步代码，这相当于是让线程顺序执行同步代码，从而保证了有序性。

#### 4.2.4.3 volatile关键字

volatile保证可见性，当一个共享变量被volatile修饰之后，一个是线程修改了变量的值时,变量的新值对其他线程是立即可见的；另一个是禁止使用指令重排序。



volatile可以保证更新到主内存中。

volatile不保证原子性，线程1将变量读入主存中进行加一，如果此时阻塞，线程2也进行这个操作，最后的结果就是线程1和线程2读取的都是一样的值，进行了加1覆盖，结果该变量只进行了一次加1。

volatile保证有序性，不会被编译器优化，比如volatile变量之后的语句不能在volatile变量前面执行。

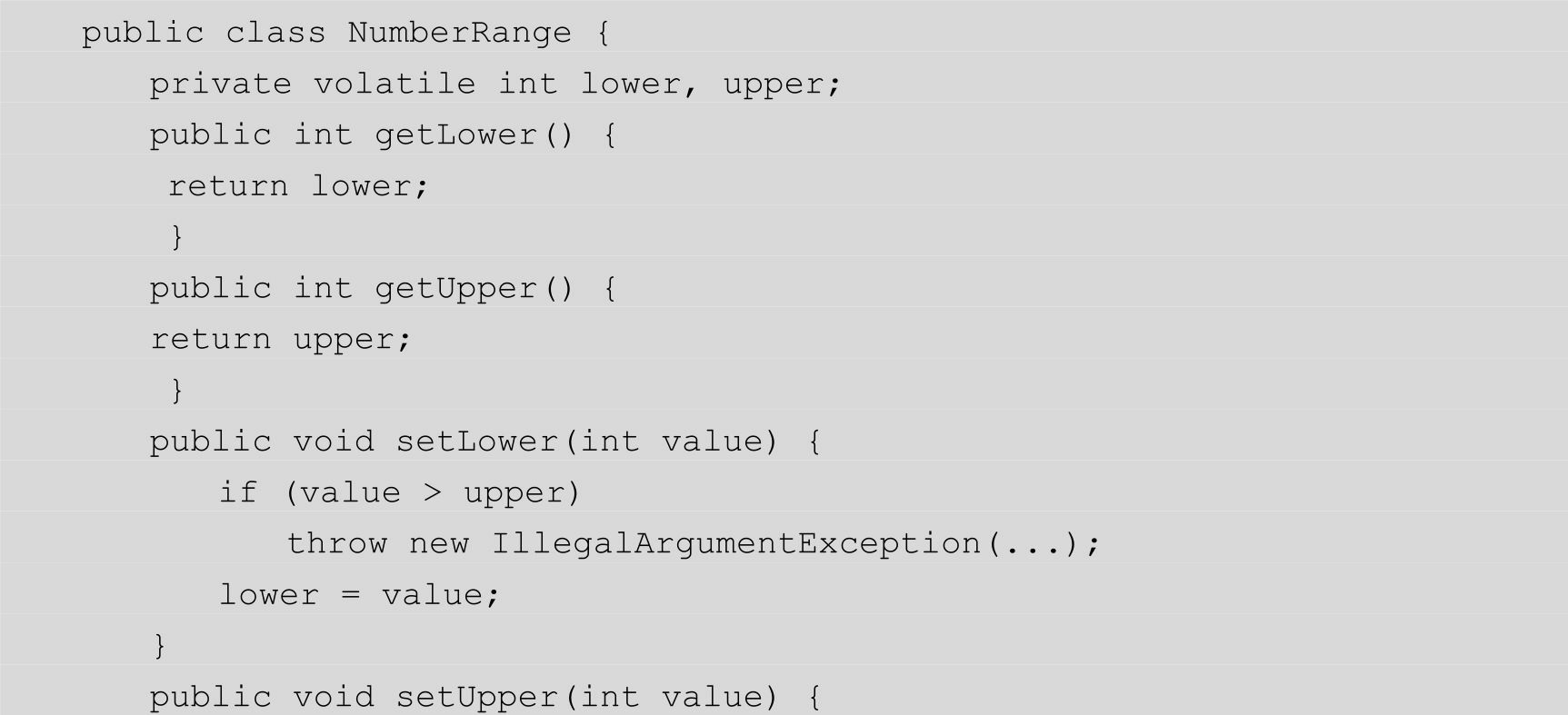
#### 4.2.4.4 正确使用volatile关键字

注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。使用volatile必须具备以下两个条件:

(1) 对变量的写操作不会依赖于当前值。

(2) 该变量没有包含在具有其他变量的不变式中。

第一个条件就是不能是自增、自减等操作，上文已经提到volatile不保证原子性。关于第二个条件，我们来举一个例子，它包含了一个不变式: 下界总是小于或等于上界，代码如下所示:

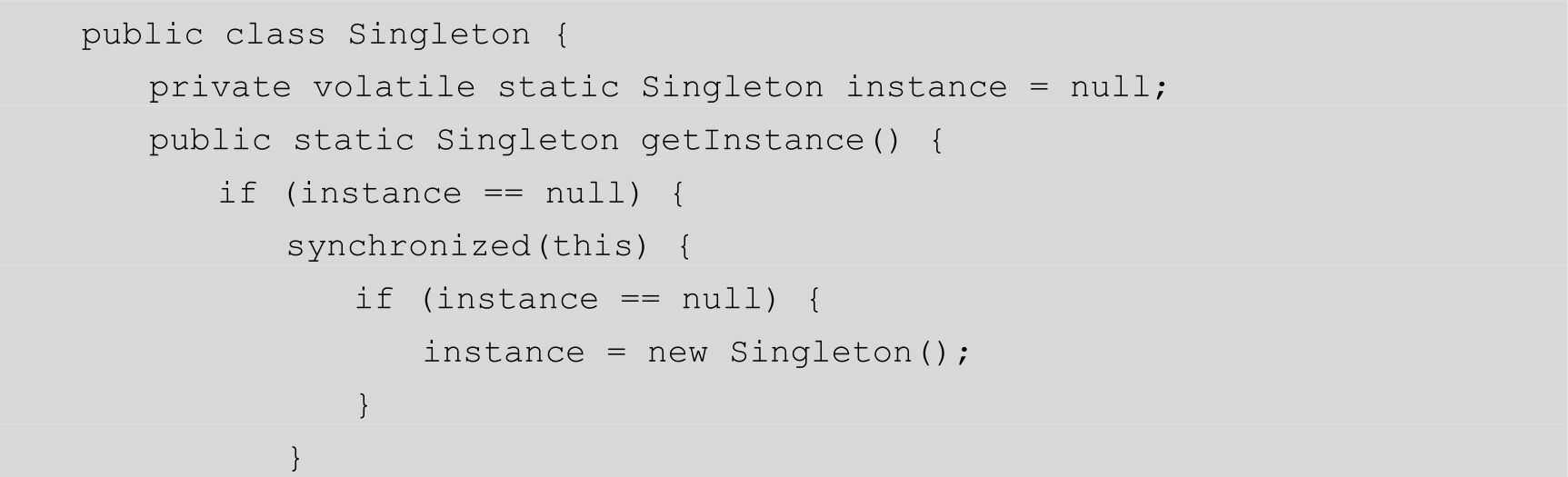
 在同一时间内，线程A调用setLower(4)并且线程B调用setUpper(3)，虽然这两个操作交叉存入的值是不符合条件的，但是这两个线程都会通过用于保护不变式的检查，使得最后的范围值是(4, 3)，这显然是不对的。

使用volatile有很多种场景，这里介绍其中的两种。

(1) 状态标志

就是之前的那个利用一个变量来退出while循环。

(2) 双重检查模式(DCL)

 getInstance方法中对Singleton进行了两次判空，第一次是为了不必要的同步，第二次是只有在Singleton等于null的情况下才创建实例。在这里用到了volatile关键字会或多或少地影响性能，但考虑到程序的正确性，牺牲这点性能还是值得的。DCL的优点是资源利用率高，第一次执行getInstance方法时单例对象才被实例化，效率高。其缺点是第一次加载时反应稍慢一些，在高并发环境下也有一定的缺陷(虽然发生的概率很小)。

## 4.3 阻塞队列

4.3.1 阻塞队列简介

(1) 当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞(挂起)，直到有数据放入队列。

(2) 当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞(挂起)，直到队列中有空的位置，线程被自动唤醒。

4.3.2 Java中的阻塞队列

• ArrayBlockingQueue: 由数组结构组成的有界阻塞队列。

• LinkedBlockingQueue: 由链表结构组成的有界阻塞队列。

• PriorityBlockingQueue: 支持优先级排序的无界阻塞队列。

• DelayQueue: 使用优先级队列实现的无界阻塞队列。

• SynchronousQueue: 不存储元素的阻塞队列。

• LinkedTransferQueue: 由链表结构组成的无界阻塞队列。

• LinkedBlockingDeque: 由链表结构组成的双向阻塞队列。

4.3.3 阻塞队列的实现原理

4.3.4 阻塞队列的使用场景

## 4.4 线程池

在Java 1.5中提供了Executor框架用于把任务的提交和执行解耦,任务的提交交给Runnable或者Callable，而Executor框架用来处理任务。Executor框架中最核心的成员就是 ThreadPoolExecutor，它是线程池的核心实现类。

## 4.5 AsyncTask的原理

# 第5章 网络编程与网络框架

# 第6章 设计模式

GoF提出的设计模式总共有23种，根据目的准则分类，分为三大类。

• 创建型设计模式，共5种: 单例模式、工厂方法模式、抽象工厂模式、建造者模式、原型模式。

• 结构型设计模式，共7种: 适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

• 行为型设计模式，共11种: 策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

代码在FramworkFromAndroidOin2018/BaseKnowledge/DesignIdea/

## 6.3 创建型设计模式

### 6.3.1单例模式

**1.单例模式的6种写法**

**(1)饿汉模式**

这种方式在类加载时就完成了初始化，所以类加载较慢，但获取对象的速度快。这种方式基于类加载机制，避免了多线程的同步问题。在类加载的时候就完成实例化，没有达到懒加载的效果。如果从始至终未使用过这个实例，则会造成内存的浪费。

**(2)懒汉模式(线程不安全)**

懒汉模式声明了一个静态对象，在用户第一次调用时初始化。这虽然节约了资源，但第一次加载时需要实例化，反应稍慢一些，而且在多线程时不能正常工作。

**(3)懒汉模式(线程安全)**

这种写法能够在多线程中很好地工作，但是每次调用getInstance方法时都需要进行同步。这会造成不必要的同步开销，而且大部分时候我们是用不到同步的。所以，不建议用这种模式。

**(4)双重检查模式(DCL)**

这种写法在getSingleton方法中对Singleton进行了两次判空: 第一次是为了不必要的同步，第二次是在Singleton等于null的情况下才创建实例。在这里使用volatile会或多或少地影响性能，但考虑到程序的正确性，牺牲这点性能还是值得的。DCL的优点是资源利用率高。第一次执行getInstance时单例对象才被实例化，效率高。其缺点是第一次加载时反应稍慢一些，在高并发环境下也有一定的缺陷。DCL虽然在一定程度上解决了资源的消耗和多余的同步、线程安全等问题，但其还是在某些情况会出现失效的问题，也就是DCL失效。这里建议用静态内部类单例模式来替代DCL。

**(5)静态内部类单例模式**

第一次加载Singleton类时并不会初始化sInstance，只有第一次调用getInstance方法时虚拟机加载SingletonHolder 并初始化 sInstance。这样不仅能确保线程安全，也能保证Singleton类的唯一性。所以，推荐使用静态内部类单例模式。

**(6)枚举单例**

默认枚举实例的创建是线程安全的，并且在任何情况下都是单例。在上面讲的几种单例模式实现中，有一种情况下其会重新创建对象，那就是反序列化: 将一个单例实例对象写到磁盘再读回来，从而获得了一个实例。反序列化操作提供了readResolve方法，这个方法可以让开发人员控制对象的反序列化。在上述几个方法示例中，如果要杜绝单例对象被反序列化时重新生成对象，就必须加入如下方法:

枚举单例的优点就是简单，但是大部分应用开发很少用枚举，其可读性并不是很高。到这里单例模式的6种写法都介绍完了。至于选择用哪种形式的单例模式，则取决于你的项目本身情况: 是否为复杂的并发环境，或者是否需要控制单例对象的资源消耗。

**2.单例模式的使用场景**

在一个系统中，要求一个类有且仅有一个对象，它的具体使用场景如下:

• 整个项目需要一个共享访问点或共享数据。

• 创建一个对象需要耗费的资源过多，比如访问I/O或者数据库等资源。

• 工具类对象。

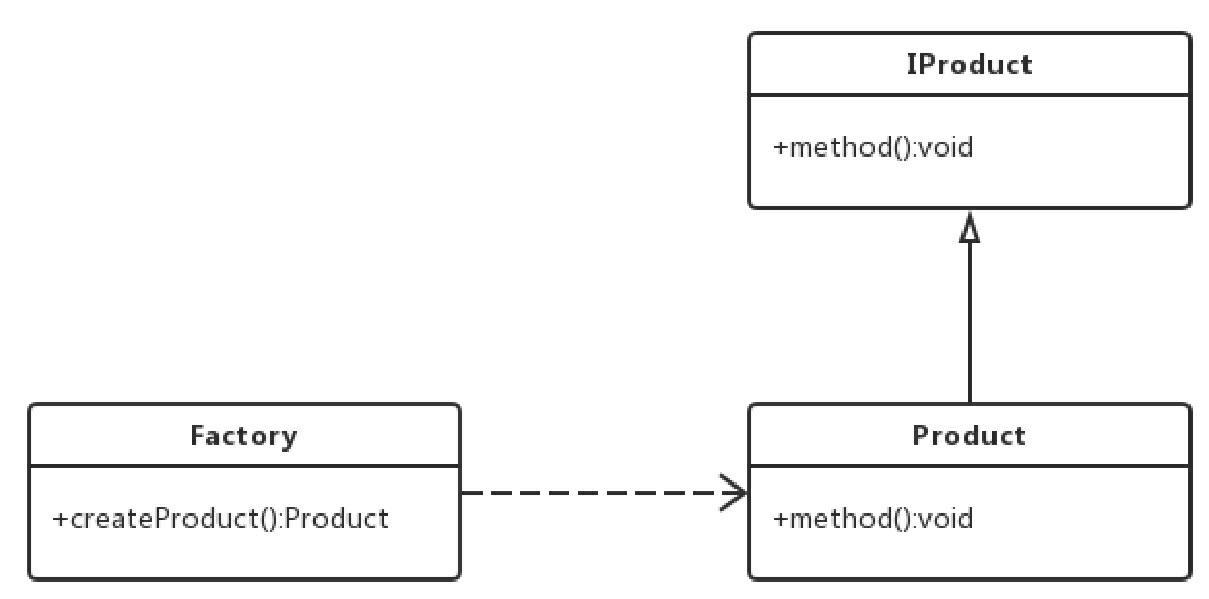
### 6.3.2简单工厂模式

在简单工厂模式中有如下角色。

•Factory: 工厂类，这是简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类的创建产品类的方法可以被外界直接调用，创建所需的产品对象。

•IProduct: 抽象产品类，这是简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。

•Product: 具体产品类，这是简单工厂模式的创建目标。

 Factory依赖传入Product参数，Product继承IProduct。

**1.简单工厂模式的简单实现**

**2.使用简单工厂模式的场景和优缺点**

• 优点: 使用户根据参数获得对应的类实例，避免了直接实例化类，降低了耦合性。

• 缺点: 可实例化的类型在编译期间已经被确定。如果增加新类型，则需要修改工厂，这违背了开放封闭原则。简单工厂需要知道所有要生成的类型其当子类过多或者子类层次过多时不适合使用。

### 6.3.3 工厂方法模式

定义: 定义一个用于创建对象的接口，让子类决定实例化哪个类。工厂方法使一个类的实例化延迟到其子类。

工厂方法模式的结构图如图6-3所示。

在工厂方法模式中有如下角色。

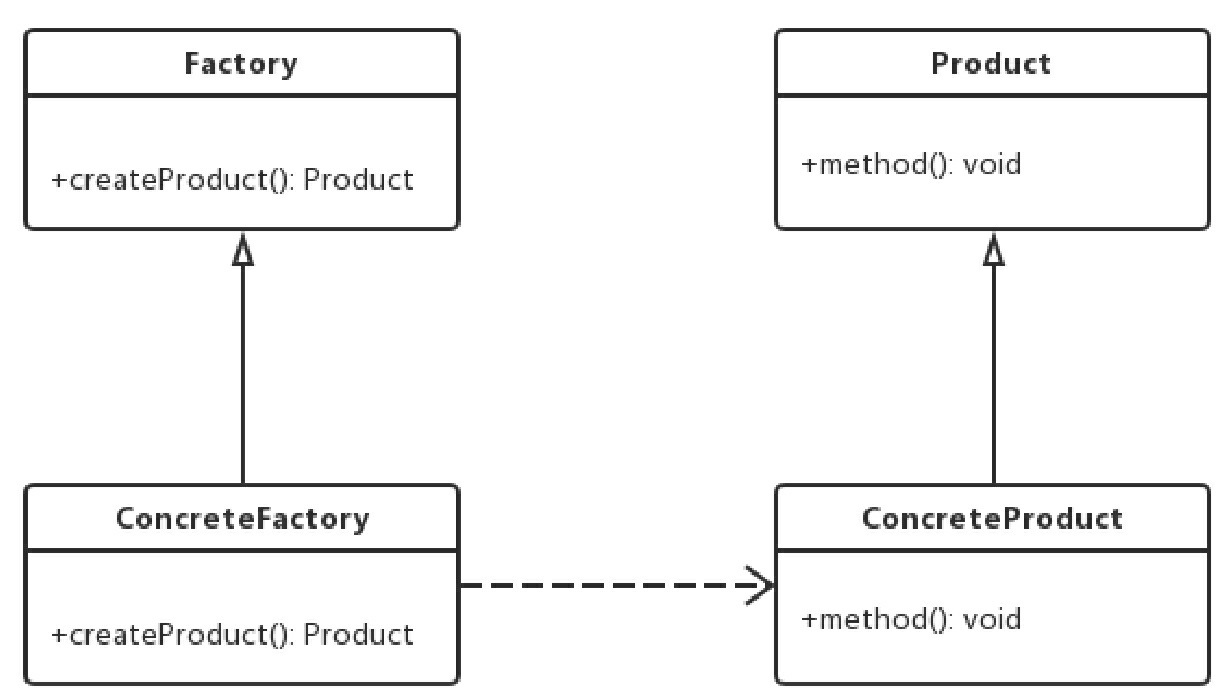
**1.工厂方法模式的简单实现**

• Product: 抽象产品类。

• ConcreteProduct: 具体产品类，实现Product接口。

• Factory: 抽象工厂类，该方法返回一个Product类型的对象。

• ConcreteFactory: 具体工厂类，返回ConcreteProduct实例。

**2.工厂方法与简单工厂**

对于简单工厂模式，我们都知道其在工厂类中包含了必要的逻辑判断，根据不同的条件来动态实例化相关的类。对客户端来说，这去除了与具体产品的依赖；但与此同时也会带来一个问题: 如果我们要增加产品，比如我们要生产苹果计算机，就需要在工厂类中添加一个Case分支条件，这违背了开放封闭原则，对修改也开放了。而工厂方法模式就没有违背这个开放封闭原则。如果我们需要生产苹果计算机，则无须修改工厂类，直接创建产品即可。