内容目录

《Android进阶之光》

git clone <https://github.com/henrymorgen/android-advanced-light>

•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

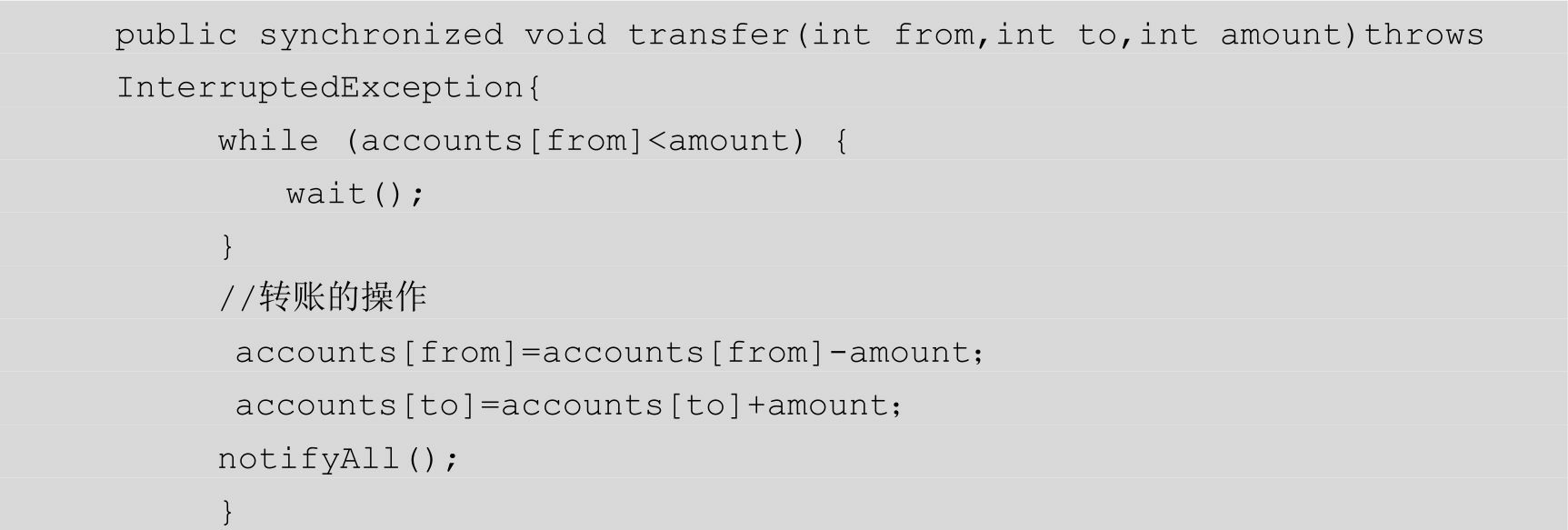


我们发现转账方余额不足；如果有其他线程给这个转账方再转足够的钱，就可以转账成功了。我们可以用newCondition方法获得一个条件对象,我们得到条件对象后调用await方法，当前线程就被阻塞了并放弃了锁。

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。

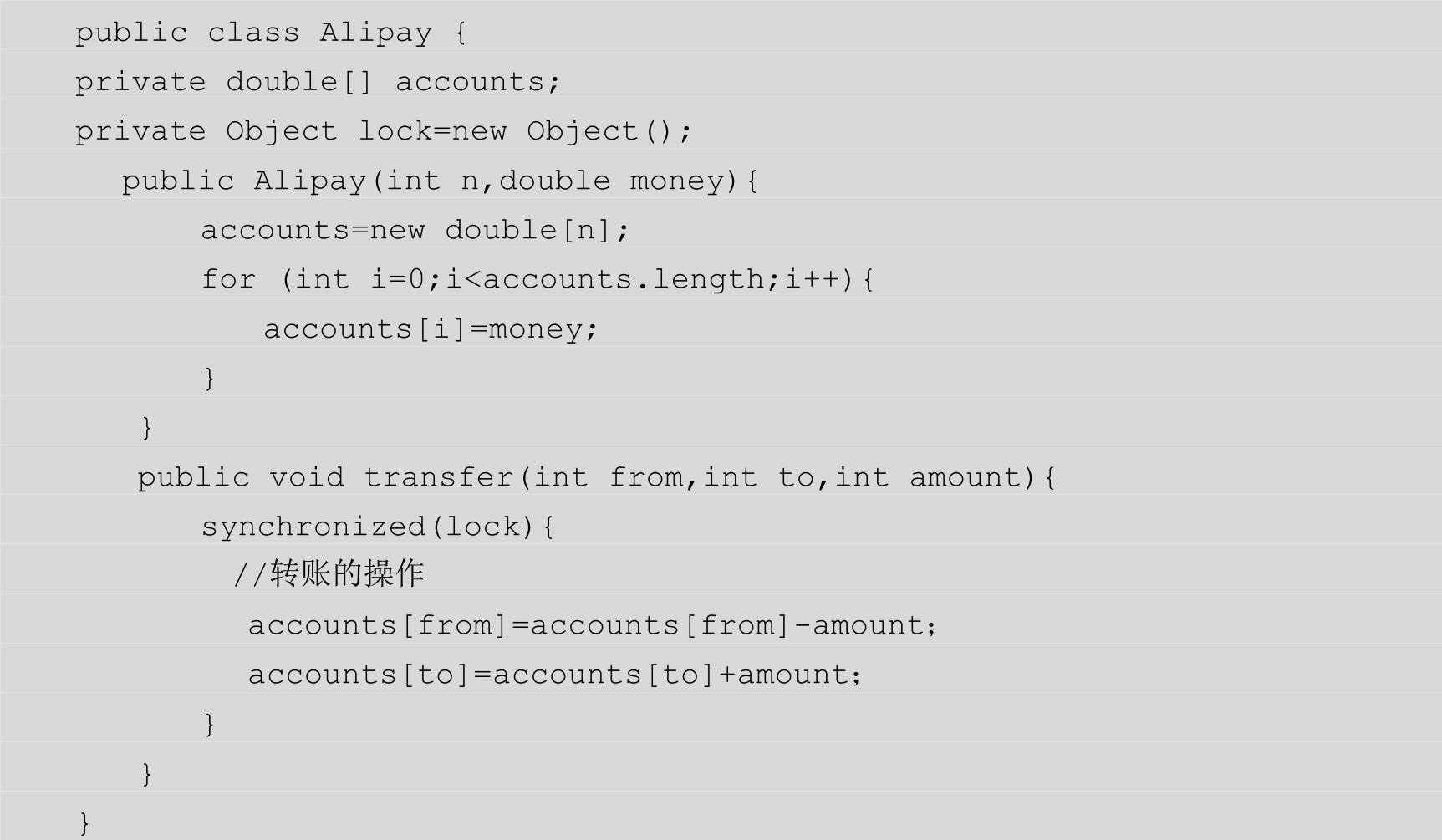
### 4.2.2 同步方法

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

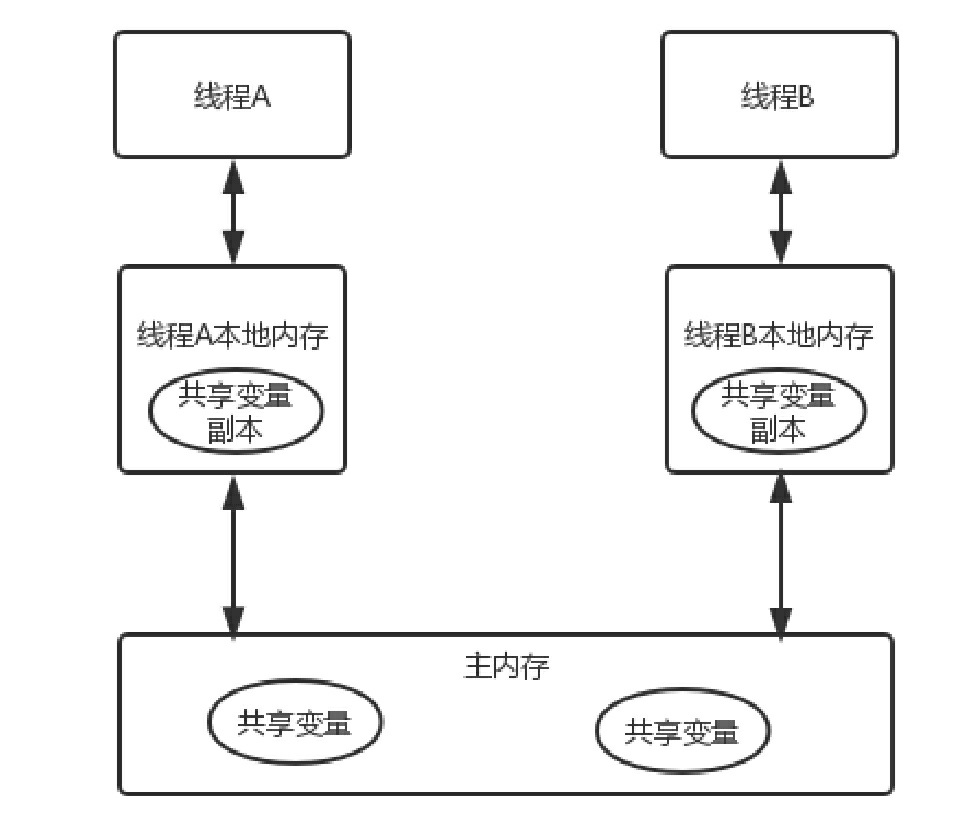
其获得了obj的锁，obj指的是一个对象。再来看看Alipay类，我们用同步代码块进行改写。

 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:

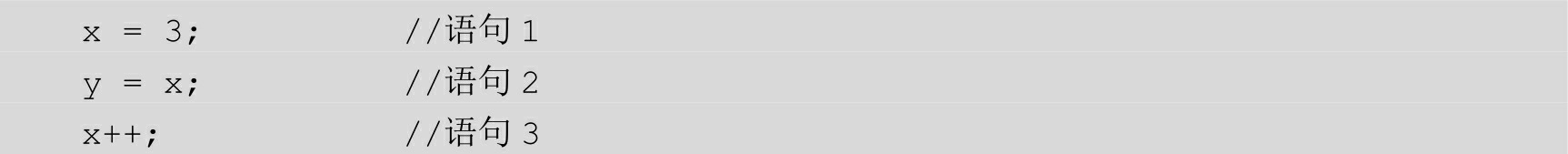
int i = 3;

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**

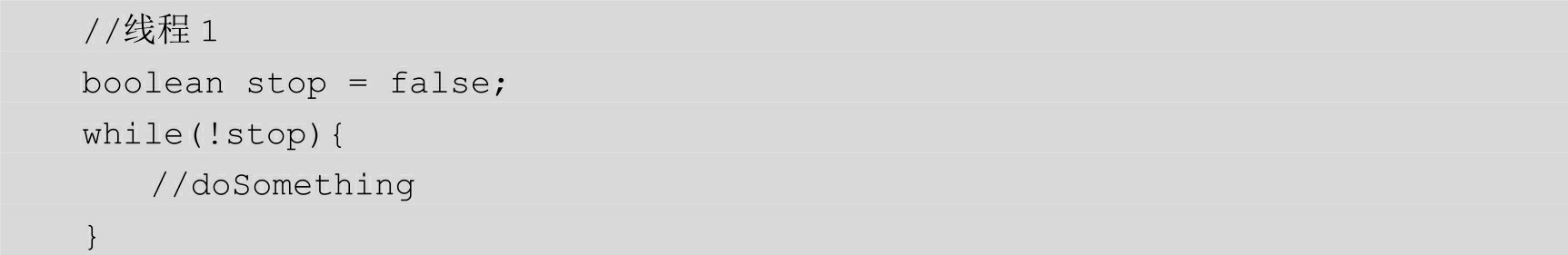
可见性是指线程之间的可见性，一个线程修改的结果，另一个线程马上就能看到。当一个共享变量被volatile修饰时，它会保证修改的值立即被更新到主存，所以对其他线程是可见的。当有其他线程需要读取该值时，其他线程会去主存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，并不会立即被写入主存，何时被写入主存也是不确定的。当其他线程去读取该值时，此时主存中可能还是原来的旧值，这样就无法保证可见性。

**(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。

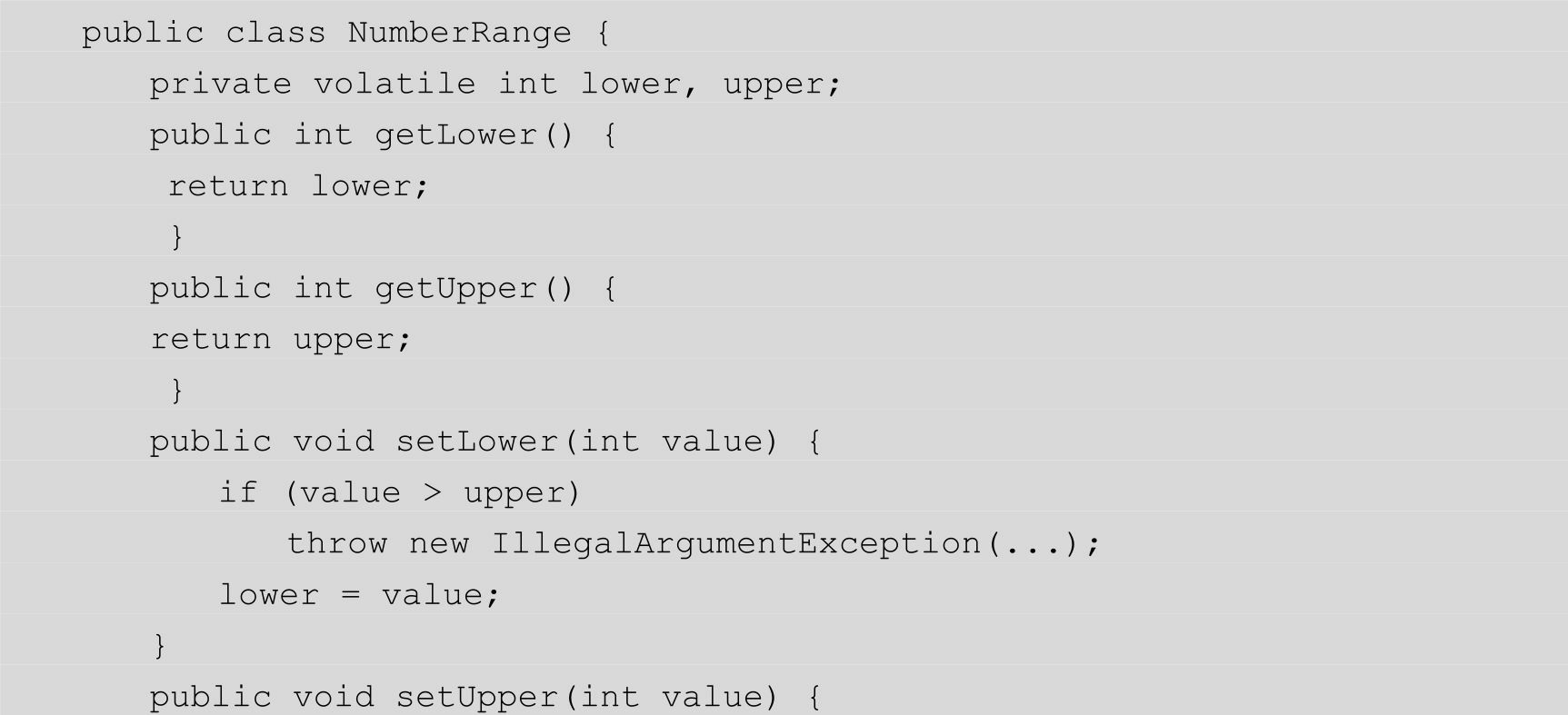
#### 4.2.4.4 正确使用volatile关键字

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:

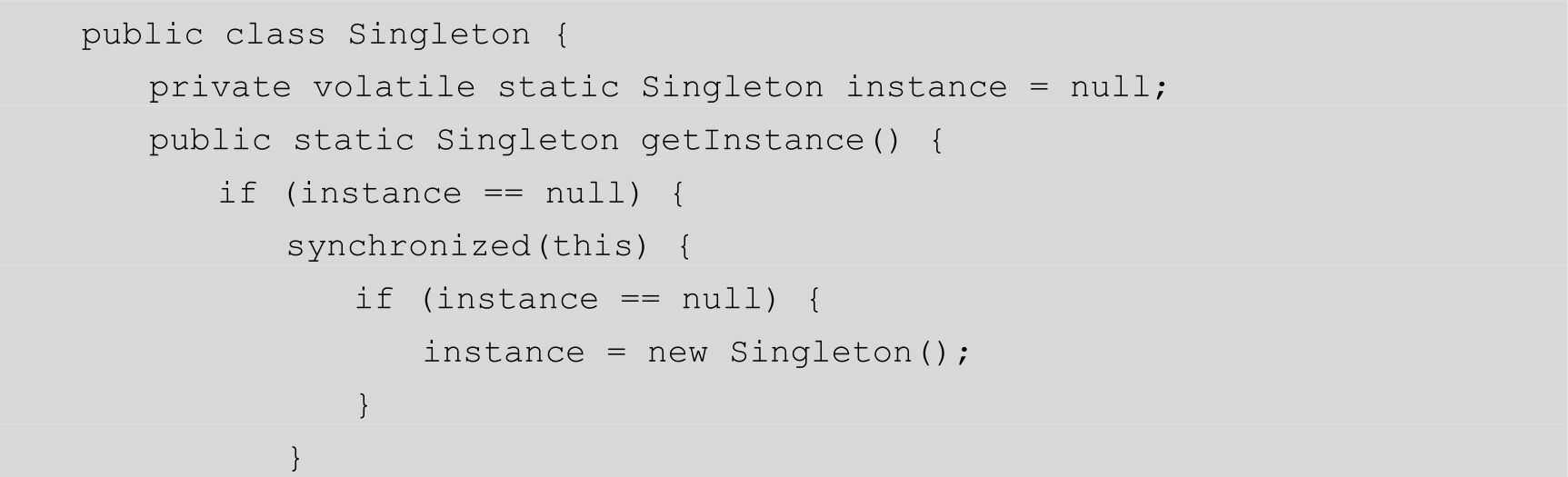
 在同一时间内，线程A调用setLower(4)并且线程B调用setUpper(3)，虽然这两个操作交叉存入的值是不符合条件的，但是这两个线程都会通过用于保护不变式的检查，使得最后的范围值是(4, 3)，这显然是不对的。

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/

# 第7章 事件总线

为了简化并且更加高质量地在Activity、Fragment、Thread和Service等之间的通信，同时解决组件之间高耦合的同时仍能继续高效地通信，事件总线设计出现了。提到事件总线我们会想到Eventbus和otto，所以本章就来讲解它们的使用方法以及原理。

## 7.1 解析EventBus

EventBus是一款针对Android优化的发布-订阅事件总线。它简化了应用程序内各组件间、组件与后台线程间的通信。其优点是开销小，代码更优雅，以及将发送者和接收者解耦。如果Activity和Activity进行交互还好说，但如果Fragment和Fragment进行交互则着实令人头疼。这时我们会使用广播来处理，但是使用广播略嫌麻烦并且效率也不高。如果传递的数据是实体类，需要序列化，那么传递的成本会有点高。

### 7.1.1 使用EventBus

EventBus的4种ThreadMode(线程模型)：

**POSTING(默认):** 发布事件和接收事件(事件处理)在同一个线程中，在线程模型为POSTING的事件处理函数中尽量避免执行耗时操作，因为它会阻塞事件的传递，甚至有可能会引起ANR。

**MAIN:** 事件的处理会在UI线程中执行。事件处理的时间不能太长,长了会导致ANR。

**BACKGROUND:** 如果事件是在UI线程中发布出来的，那么该事件处理函数就会在新的线程中运行; 如果事件本来就是在子线程中发布出来的，那么该事件处理函数直接在发布事件的线程中执行。在此事件处理函数中禁止进行UI更新操作。

**ASYNC:** 无论事件在哪个线程中发布，该事件处理函数都会在新建的子线程中执行; 同样，此事件处理函数中禁止进行UI更新操作。

**1. EventBus基本用法**

EventBus使用起来分为以下5个步骤

**(1)自定义一个事件类**

public class MessageEvent {

...

}

**(2)在需要订阅事件的地方注册事件**

EventBus.getDefault().register(this);

**(3)发送事件**

EventBus.getDefault().post(messageEvent);

**(4)处理事件**

@Subscribe (threadMode = ThreadMode.MAIN)

public void XXX(MessageEvent messageEvent) {

...

}

消息处理的方法可以随便取名，但是需要添加一个注解@Subscribe，并且要指定线程模型(默认为POSTING)。

**(5)取消事件订阅**

EventBus.getDefault().unregister(this);

**2.EventBus应用举例**

**(1)添加依赖库首先配置gradle:**

compile'org.greenrobot:eventbus:3.0.0'

**(2)定义消息事件类**

MessageEvent.java

**package** com.example.liuwangshu.mooneventbus;

/\*\*

\* Created by Administrator on 2016/8/15 0015.

\*/

**public class MessageEvent** {

**private String** message;

**public** MessageEvent(**String** message) {

**this**.message = message;

}

**public String** getMessage() {

**return** message;

}

**public void** setMessage(**String** message) {

**this**.message = message;

}

}

**(3)注册和取消订阅事件**

在 MainActivity 中定义了两个 Button: 一个用来注册事件，另一个用来跳转到SecondActivity。

bt\_message.setOnClickListener(**new** View.OnClickListener() {//跳转button

@Override

**public void** onClick(**View** v) {

startActivity(**new** Intent(**MainActivity**.**this**,**SecondActivity**.**class**));

}

});

bt\_subscription.setOnClickListener(**new** View.OnClickListener() {//注册事件button

@Override

**public void** onClick(**View** v) {

**if**(!**EventBus**.getDefault().isRegistered(**MainActivity**.**this**)) {

//注册事件

**EventBus**.getDefault().register(**MainActivity**.**this**);

}**else**{

**Toast**.makeText(**MainActivity**.**this**,"请勿重复注册事件",**Toast**.LENGTH\_SHORT).show();

}

}

});

**(4)事件订阅者处理事件**

在MainActivity中自定义方法来处理事件，在这里ThreadMode设置为MAIN，事件的处理会在UI线程中执行，用TextView来展示收到的事件消息:

**@Subscribe(threadMode = ThreadMode.MAIN)**

**public void** onMoonEvent(**MessageEvent** messageEvent){

tv\_message.setText(messageEvent.getMessage());

}

**(5)事件发布者发布事件**

创建了SecondActivity来发布消息，代码如下所示:

bt\_message=(**Button**)**this**.findViewById(**R.id**.bt\_message);

bt\_message.setText("发送事件");

bt\_message.setOnClickListener(**new** View.OnClickListener() {

@Override

**public void** onClick(**View** v) {

**EventBus**.getDefault().post(**new** MessageEvent("欢迎关注刘望舒的博客"));

finish();

}

});

**3.EventBus的黏性事件**

除了上面讲的普通事件外，EventBus还支持发送黏性事件，就是在发送事件之后再订阅该事件也能收到该事件，这跟黏性广播类似。代码参考源代码。

### 7.1.2 源码解析EventBus

前面讲到了EventBus的用法，本节我们将讲解EventBus的源码。

## 7.2 解析otto