[Java多线程 3](#_Toc88488711)

[1. 多线程的思想和原理 3](#_Toc88488712)

[1.1 进程和线程思想引入 3](#_Toc88488713)

[1.2 多线程技术原理 3](#_Toc88488714)

[2. JVM中的多线程和垃圾回收机制 3](#_Toc88488715)

[2.1 创建新的线程 5](#_Toc88488716)

[2.1.1 继承Thread类创建新线程 6](#_Toc88488717)

[1. 学习创建新线程 6](#_Toc88488718)

[2. 深入理解调用start方法的内核 8](#_Toc88488719)

[3. 多线程的内存图 9](#_Toc88488720)

[4. 多线程的运行状态 11](#_Toc88488721)

[5. 多线程的一个实例——售票 11](#_Toc88488722)

[2.1.2 实现Runnable接口创建新线程 12](#_Toc88488723)

[1. 售票实例——实现Runnable接口 13](#_Toc88488724)

[2. 实现Runnable接口来创建新线程的好处 13](#_Toc88488725)

[2.2 多线程的安全问题 14](#_Toc88488726)

[2.2.1 synchronized同步代码块 15](#_Toc88488727)

[2.2.2 同步的好坏及前提 16](#_Toc88488728)

[2.2.3 同步函数 16](#_Toc88488729)

[2.2.4 static修饰的同步函数 19](#_Toc88488730)

[2.2.5 单例模式的并发访问 20](#_Toc88488731)

[2.2.6 同步函数和同步代码块的区别 21](#_Toc88488732)

[2.2.7 死锁 22](#_Toc88488733)

[2.3 多线程之间的通信 23](#_Toc88488734)

[2.3.1先不考虑多线程，只考虑需求 23](#_Toc88488735)

[2.3.2单生产者单消费者——消费者不消费 24](#_Toc88488736)

[2.3.3 等待唤醒机制 25](#_Toc88488737)

[2.3.4单生产者单消费者——解决系统hang on 27](#_Toc88488738)

[2.3.5多个生产者和消费者 27](#_Toc88488739)

[2.3.3 多线程之多生产多消费——JDK1.5 Lock 接口 28](#_Toc88488740)

[2.3.4 多线程之多生产多消费——JDK1.5 Condition 接口 31](#_Toc88488741)

# Java多线程

# 1. 多线程的思想和原理

## 1.1 进程和线程思想引入

线程是一个我们之前没有听过的东西，认识它之前我们先来说另一个你可能知道的概念：进程。

什么是进程？直译过来就是正在进行的程序，你打开windows的任务管理器，就会发现里面有一个进程模块，里面显示的都是正在运行的程序。

进一步解释就是：进程就是应用程序运行时在内存中分配的空间。进程不负责执行。

那么一个程序分配完空间后，要想运行程序，必须要按照固定的起点和顺序进行执行，这个执行过程就是线程来控制的。进一步解释就是：线程就是进程中负责程序执行的执行单元，也叫执行路径。一个进程中至少有一个线程在负责该进程的运行。如果一个进程中出现了多个线程，就称这个程序时多线程程序。

举个例子来类比的话：

比如一个开辟一块地方建一个游泳池，相当于是一个进程，游泳考试的时候可以一个人先进来游然后计分，他结束后另一个人再进来考，这就是单线程。那么如果在泳池里设置多个赛道，多个人可以同时考试，这就是多线程。

## 1.2 多线程技术原理

多线程这种技术：可以解决多部分代码同时执行的需求。可以合理的使用CPU的资源。

我们首先要理解，内存只是一片存储空间，它本身是不能进行任何运算的，计算机中所有应用程序的运算都是由中央处理器（CPU）执行的。你同时打开多个应用程序，对于你而言它们是可以同时进行运算的，比如你播放音乐的同时还能用微信跟别人聊天，你就感觉好像是CPU在同一时间同时进行播放音乐和微信聊天的线程，但是实际上，CPU在某个时间只能进行一个计算，但是它的这种计算会在不同的应用程序之间以时间片进行非常快速的切换，快速到你根本感觉不到，所以看起来像是同时运行。

那么如何实现理论上的同时运行呢？现在我们使用的计算机一般都是多核的，相当于一个CPU里面有几个核，就可以同时进行几个计算。比如之前单核CPU相当于一个老师，开启的多个应用程序相当于多个等待被喂食的小朋友，老师就轮流以非常快的速度给他们喂食。多核CPU相当于有了多个老师。

Windows本身就是多线程的，因为它可以同时开启多个应用程序，相当于开启了多进程，而每个进程中又可以有多个线程。

# 2. JVM中的多线程和垃圾回收机制

考虑下面的代码执行过程：

public static void main(String[] args){

new Demo();

new Demo();

new Demo();

new Demo();

new Demo();

new Demo();

System,out.println(“Hello world!”);

}

我们都知道在程序执行的过程中有一个主线程，它在不断的new对象出来，因为创建的是匿名对象，意味着它们一旦创建就变成了垃圾，会由java的垃圾回收机制进行回收，那么考虑这种情况，当主线程创建到第4个对象时，java的垃圾回收机制启动了要回收前3个对象，垃圾回收是一个独立的线程，它不影响主线程继续创建新的对象。

Java中的垃圾回收线程是不定时启动的，这是由CPU的切换决定的。但是一旦垃圾回收线程启动，它会去看堆内存中的内存量，然后视情况清除垃圾。

与垃圾回收线程类似，如果你开启了多线程，其他的线程运行也具有随机性，这是由CPU的快速切换造成的。

所以到这里我们知道，JVM中至少有两个线程：

1. 一个负责代码运行的主线程
2. 另一个负责垃圾回收的线程。

堆内的每个对象都会被回收，怎么回收是由对象自己决定的，所以java把这个回收方法定义在Object类中，查看文档会发现Object类有一个finalize()方法，当垃圾回收器确定系统中不再需要这个对象时，垃圾回收器就会调用这个方法回收对象。

我们可以用下面的代码试着看一下垃圾回收的过程：

class Sample {

public void finalize() { // 重写了Object的方法

System.out.println("finalize sample");

}

}

public class FinalizeDemo {

public static void main(String[] args) {

new Sample();

new Sample();

new Sample();

new Sample();

new Sample();

System.out.println("main function finish");

}

}

运行之后发现finalize方法并没有被执行，这是因执行为java的垃圾回收器在主线程运行期间都没有动作。因为垃圾回收器什么时候启动是CPU说了算的。

但是java也帮我们提供了一个方法来主动启动垃圾回收器，那就是System类中的gc()方法。所以让我们主动启动垃圾回收器再运行试试：

class Sample {

public void finalize() { // 重写了Object的方法

System.out.println("finalize sample");

}

}

public class FinalizeDemo {

public static void main(String[] args) {

new Sample();

new Sample();

new Sample();

new Sample();

new Sample();

System.gc(); // 启动垃圾回收器

System.out.println("main function finish");

}

}

这时就会发现finalize方法被执行了，意味着垃圾回收器回收了垃圾。并且多执行几次可能会发现finalize被调用的次数不同，这是因为启动垃圾回收器后程序运行过程中有两个线程，CPU在主线程和垃圾回收线程之间来回切换运算，当主线程完成后，垃圾回收线程也会跟着结束无论垃圾回收完与否。这是由CPU决定的。

我们实验过发现，主线程就负责main函数中代码的执行，垃圾回收线程就负责垃圾回收代码的执行，他俩互不干扰，所以每一个线程都有运行的代码内容，这叫做线程任务。

我们后面也会主动去创建线程，每一个线程的目的都是为了去运行指定的任务代码。而线程的任务代码都封装在特定的区域中，比如主线程运行的任务都定义在main方法中，垃圾回收线程在收垃圾时都会运行finalize方法。

垃圾回收线程是jvm内置的线程，正常情况下我们做开发时不需要太关注它，也不需要主动开启它，因为它会在需要的时候自动开启。

## 2.1 创建新的线程

初步了解多线程后，我们就会想，我们在java开发中如何能主动的开启一个新的线程呢？

这里需要明确的是，线程本身不是应用程序创建的，而是通过调用所在计算机的操作系统来创建的，所以从最底层来讲，线程是由操作系统创建的。

Java应用程序也是一样，它也可以调用操作系统暴露的接口来创建新线程，java内部封装了这个创建线程的功能，在java的API文档中可以查到，有2种方式可以创建新线程：

1. 将类声明为Thread类的子类，该类要重写run方法，然后创建该类的对象，调用start方法来启动线程并自动运行run方法中的任务代码。

### 2.1.1 继承Thread类创建新线程

#### 1. 学习创建新线程

我们通过文档可以知道，这种方式需要两步：1. 继承Thread类，2.重写run方法。

继承Thread类我们之前已经学过了，直接继承，但是重写run方法？怎么重写，那就需要先了解run方法是干嘛的，查询发现，run方法啥也没干，就是个返回值为void的空方法。

那么我们就按着做：

class MyThread extends Thread{

public void run(){

}

}

然后接着看文档，发现通过new 创建的这个类的对象就是新线程对象，然后用.start()启动线程。那么.start()是干啥的？接着查文档，就发现它的解释是：start() 方法使该线程开始执行，Java虚拟机调用该线程的run方法。哦到这里我们就明白了，上面重写run方法是有用的。

start()方法是Thread类默认的方法，调用它会做两个事情：

1. 启动线程
2. 执行run()方法。

所以我们就知道了，run方法中放的就是线程要执行的代码，所以记住，自定义线程的任务代码都存储在run方法中。

例子：下面代码中，运行时默认有一个main线程，然后我们又主动创建了t1,t2两个新的线程，而执行时打印内容的顺序是不确定的，这是因为CPU运算的不确定性。

class MyThread extends Thread {  
  
 private String name;  
  
 public MyThread(String name) {  
 this.name = name;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(name + "......" + i);  
 }  
 }  
}  
  
public class ThreadDemo {  
  
 public static void main(String[] args) {  
 MyThread t1 = new MyThread("张三");  
 MyThread t2 = new MyThread("李四");  
 t1.start();  
 t2.start();  
  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println("main......" + i);  
 }  
 }  
}

在上述运行结果中，我们是用name成员变量来区分不同的线程，但是实际上Thread类提供了方法来获取创建的线程的名称：

String getName();

因为我们创建的MyThread类继承了Thread类，所以它可以直接调用父类的这个方法来拿到创建的线程的名字：

public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(getName() + "......" + name + "......" + i);  
 }  
}

这样我们就会打印出当前对象的默认线程名。

但是如果我们想让main方法中的打印内容也显示线程名字，应该怎么办？main方法所在的类是ThreadDemo类，它可没有继承Thread类所以不能调用getName()方法，那么怎么办呢？Thread类除了getName方法外，还提供了另一个静态方法来获取当前正在运行的线程对象：

static Thread currentThread();

所以main方法里可以这么写：

public static void main(String[] args) {  
 MyThread t1 = new MyThread("张三");  
 MyThread t2 = new MyThread("李四");  
 t1.start();  
 t2.start();  
  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "......" + i);  
 }  
}

#### 2. 深入理解调用start方法的内核

到这里我们大概了解了如果通过继承Thread类来创建并启动一个线程。那么我们现在来思考下面两个面试中常遇到的问题：

1. 如果MyThread类中重写了start()方法的话，运行过程和不重写有什么不同？

重写start()方法后的代码如下：

class MyThread extends Thread {  
 private String name;  
 public MyThread(String name) {  
 this.name = name;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(getName() + "......" + name + "......" + i);  
 }  
 }  
  
 public void start() {  
 this.run();  
 }  
}  
  
public class ThreadDemo {  
 public static void main(String[] args) {  
 MyThread t1 = new MyThread("张三");  
 MyThread t2 = new MyThread("李四");  
 t1.start();  
 t2.start();  
  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "......" + i);  
 }  
 }  
}

我们运行这个main方法会发现打印结果和之前一样，但是打印结果相同不意味着执行过程也相同，当MyThread类重写start() 方法后，t1.start()和t2.start()相当于是运行的子类的方法，t1和t2虽然是已经创建的线程对象，但是start()方法被重写后失去了原来启动线程的特性，这就意味着虽然还是运行了run()方法，但是t1和t2线程并没有启动，实际上是main线程在执行它们的代码。

那么你可能会想，既然是main线程在执行代码，为什么t1和t2的getName()打印的不是main呢？其实这也很好理解，你调用getName()是调用的对象的方法，t1和t2虽然没被启动，但是仍然是线程对象。

所以如果是为了查看当前运行的线程，应该是使用Thread.currentThread().getName()，而不是getName():

public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "......" + name + "......" + i);  
 }  
}  
  
public void start() {  
 this.run();  
}

这个问题需要深刻理解，new一个线程类是创建出了一个线程对象，此时它默认就有了一个线程名称，但是必须通过调用父类的start()方法来启动线程，它才会真正的去执行代码。

在实际开发中，不要去重写start()方法。

1. 调用start方法和直接调用run方法有什么不同？

这个问题跟前面那个问题其实是一样的，调用start()方法是指调用Thread类的静态start()方法，它做两件事：启动线程并用该线程去执行run()方法中的内容。

而直接调用run()方法实际上是main线程在执行run()方法中的内容。

#### 3. 多线程的内存图

想要更深刻的了解多线程的运行，还是要理解多线程运行时内存中的变化。

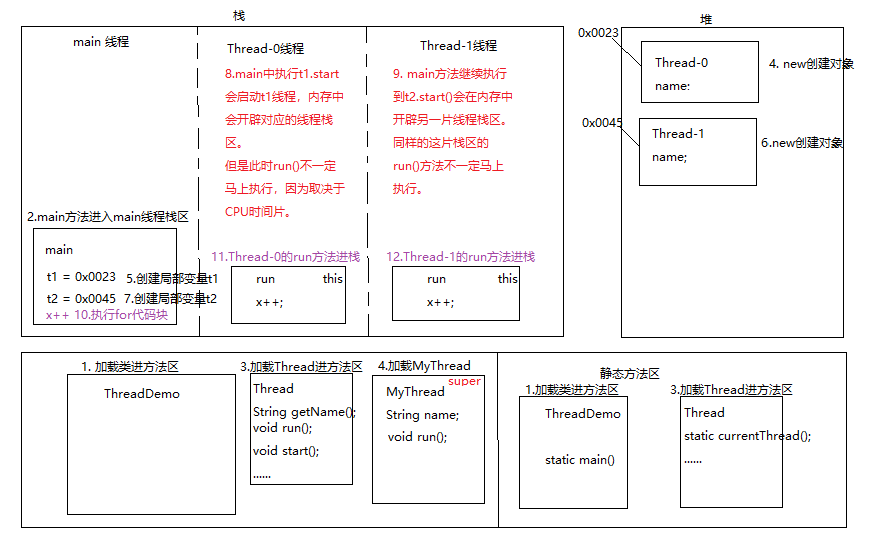
还是以下面的代码为例：

class MyThread extends Thread {  
  
 private String name;  
  
 public MyThread(String name) {  
 this.name = name;  
 }  
  
 public void run() {  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "......" + name + "......" + i);  
 }  
 }  
}  
  
public class ThreadDemo {  
  
 public static void main(String[] args) {  
 MyThread t1 = new MyThread("张三");  
 MyThread t2 = new MyThread("李四");  
 t1.start();  
 t2.start();  
  
 for (int i = 0; i < 10; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "......" + i);  
 }  
 }  
}

首先回顾一下我们之前画的内存图，包括栈，堆，方法区，然后栈是先进后出。但是学习多线程以后我们发现，main线程和别的线程同时运行（实际上是CPU在它们之间不停的做快速切换运算），我们之前学过在main方法内部调用别的方法时，必须一个一个按顺序来，但是我们的例子中main方法里调用了两个start()方法，但是这俩方法CPU不断切换，并没有按前后顺序来，这不是违背了栈内存的原理吗？

这里我们就要了解一些新知识了，栈内存首先有一个main线程区，main方法首先进栈，如果main方法中启动了新的线程，那么栈内存内会开辟新的线程区。你可以这么理解，栈内存中不同的线程路径对应不同的栈区，在各自的栈区内，先进后出。

那么上面代码的内存图就是：



注意，上面图中紫色的10, 11,12这三个执行过程顺序不确定，是由CPU来决定的。

#### 4. 多线程的运行状态

我们上面画内存图的时候就发现了，多线程的执行顺序是不确定的，这是由CPU决定，那么这样一来，线程就有很多种的状态。只有理解了线程的状态，我们才能更好的描述线程的执行。

其实现在我们自己就能猜测几个状态了：

1. 通过new 关键字来创建线程对象。此时线程状态是被创建。
2. 通过调用start()方法来启动线程，线程启动后未必马上执行，但是即使它此时在等待，但早晚会被执行的。所以此时，线程具备CPU的执行资格，它的状态是临时阻塞状态。
3. 当CPU切换到当前的线程执行时，线程的状态就是运行中。
4. 目前我们知道的就这么多，但是想也知道，一个线程最终肯定是要停止的，一个线程怎么样才会停止呢？线程启动的目的就是为了执行run()方法，那么run()方法结束后线程自然就结束了。线程结束后的状态叫消亡状态。

另外，Thread类还提供了一个 void stop()方法来主动结束一个线程，但是目前该方法已停用。

1. 上面的4种状态我们都比较好理解，但是线程还有另一种比较特殊的状态，就是冻结。冻结的意思是线程还存在，只是不会被CPU执行了。现在可能还无法理解为什么会有这么奇怪的状态，但是没关系，我们先了解一下这个概念就可以了。

Java中有些类提供了几种方法来让线程进入冻结状态：

* Thread类：static void sleep(long millis, int nanos); //导致正在执行的线程以指定的毫秒数加上指定的纳秒数来暂停（临时停止执行），这段时间里该线程会释放CPU的执行资格。时间过了之后，该线程会自动激活并重新获得CPU的执行资格。
* Object类：void wait(); // 导致当前线程等待，直到另一个线程调用 notify()方法或该对象的 notifyAll()方法唤醒它。

Object类：void wait(long timeout); // 导致当前线程等待，直到另一个线程调用 notify()方法或该对象的 notifyAll()方法唤醒它，或者指定的时间已过。

需要注意，一个线程调用wait()方法进入冻结状态是很危险的，因为如果没有别的线程调用notify()来唤醒它的话，它就会一直被冻结，即使别的线程都已经结束了就剩它自己，那么由于还有线程存在，进程无法停止，就会一直挂着，程序就死掉了。

#### 5. 多线程的一个实例——售票

需求：我们有100张票，一个窗口卖票的话很慢，现在想要开启4个窗口同时卖票。

我们最初的想法大概是这样：

class TicketSale extends Thread {  
 private int ticketNum = 100;  
 public void run() {  
 while (ticketNum > 0) {  
 System.*out*.println(Thread.*currentThread*().getName()+ “……” + ticketNum--);  
 }  
 }  
  
}  
  
public class ThreadTicketSampleDemo {  
 public static void main(String[] args) {  
 TicketSale t1 = new TicketSale();  
 TicketSale t2 = new TicketSale();  
 TicketSale t3 = new TicketSale();  
 TicketSale t4 = new TicketSale();  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 }  
}

但是运行会发现，会有好几个线程同时卖同一个票，总共卖了400张票，这是为什么呢？其实也很明显，因为在堆内存当中有四个对象，每个对象都持有100张票，每个线程都处理100张。

那么这时就要思考了，如何才能用我们已经学过的只是来解决这个问题？那么就要分析了，我们出现问题的原因是，堆内存中有四个对象，每个对象都有100张票，那么我们需要的是只有100张票，就有两种解决方案：

1. 还是有四个对象，只有100张票。这种方案就是把ticketNum设置为static，那么这个成员变量就会在方法去中的静态区中，且四个对象共享这个变量。
2. 只创建一个对象，这个对象有100张票。这种方案我们之前也学过，只创建一个对象那就是单例模式。

上面的两种解决方案都可以解决本例的问题，但是不建议这么做。因为实际生活中，可能不止有一种票，有火车票，动车票，高铁票，那么这种情况下，各种票都卖100张，用上面的两种方案解决起来都有点复杂。这时我们就要用到创建多线程的第二种方式。

### 2.1.2 实现Runnable接口创建新线程

查找JAVA api 文档来看Runnable接口的定义和方法。

Runnable接口应由任何类实现，其实例将由线程执行。 该类必须定义一个无参数的方法，称为run 。

需要注意的是：实现Runnable接口的类不是线程类，new对象时不会创建新的线程。创建新线程依然要通过new Thread类来实现。

而Thread类除了空参数构造方法以外，还提供了一些带参数的构造方法，有一个就是：

Thread(Runnable target)

它可以接收一个Runnable类型的对象作为参数，这时，new Thread(Runnable target)时就会创建一个线程，并把Runnable参数中的run()方法作为任务代码。

那么，Runnable接口的出现，就实现了线程任务和线程类的解耦。

#### 1. 售票实例——实现Runnable接口

解释的有点抽象，我们就通过实例来了解这种方式的应用，还是用上面售票的例子：

class TicketSale2 implements Runnable {  
  
 private int ticketNum = 100;  
  
 public void run() {  
 while (ticketNum > 0) {  
 System.*out*.println(Thread.*currentThread*().getName() + "......" + ticketNum--);  
 }  
 }  
}  
  
public class RunnableTicketSampleDemo {  
  
 public static void main(String[] args) {  
 TicketSale2 t = new TicketSale2();  
 Thread t1 = new Thread(t);  
 Thread t2 = new Thread(t);  
 Thread t3 = new Thread(t);  
 Thread t4 = new Thread(t);  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 }  
}

运行这个代码，就发现四个线程共同卖完了100张票，解决了我们的需求。

那么总结，如何用实现Runnable接口来实现创建新线程并执行呢？

1. 定义一个类实现Runnable接口，这个类不是线程类。
2. 在类中重写Runnable接口的run()方法，将希望多线程运行的任务代码放到这个方法里。
3. 创建上面类的对象，并将它作为参数传递给new Thread(Runnable target)来创建Thread线程对象，这时线程会自动将参数对象里的run()方法作为任务代码。
4. 调用start()方法启动线程。

#### 2. 实现Runnable接口来创建新线程的好处

既然我们已经了解了创建新线程的两种方式，那么自然需要对比一下这两种方式。

相比于实现接口而言，继承有很多的限制：

1. 一个类只能继承一个父类，如果它已经有了自己的父类，那么就没办法继承Thread了。
2. 另外多线程通常情况是希望多个线程访问唯一的共同资源，但是继承Thread类的话，资源会出现多份。

那么Runnable接口相比它而言，好处就很多了：

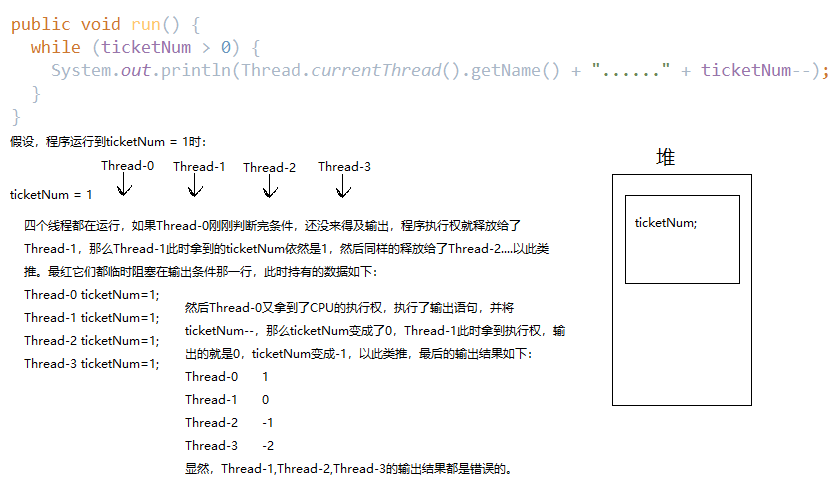
1. 避免了单继承的局限性
2. Runnable接口的出现更符合面向对象的思想，将线程任务单独进行了封装。
3. Runnable接口的出现降低了线程对象和线程任务的耦合性。

所以，我们以后都用实现Runnable接口的方式来创建多线程。

## 2.2 多线程的安全问题

一旦开启了多线程，那么多个线程的运行就有很大的不确定性，那么会不会有一些安全隐患呢？

我们分析一下上面的线程代码：



上述这种情况，你可能会觉得真的会发生吗？其实是会的，我们可以在输出语句之前暂停一下，就会发生：

public void run() {  
 while (ticketNum > 0) {  
 try {  
 Thread.*sleep*(10); // 让线程在这里暂停一下  
 } catch (InterruptedException err) {  
  
 }  
 System.*out*.println(Thread.*currentThread*().getName() + "......" + ticketNum--);  
 }  
}

上面问题的原因是：

1. 多线程处理了共享数据
2. 线程任务中有多条对共享数据的操作，一个线程在操作共享数据的过程中，其他线程参与了运算，造成了数据的错误。

### 2.2.1 synchronized同步代码块

既然知道了原因，我们怎么解决呢？只要保证多条操作共享数据的代码在某一时间段被一条线程所执行，在执行期间不允许其他线程参与运算。

在代码中如何体现？要用一个新的知识点，同步代码块：

synchronized(对象){

// 需要被同步的代码

}

代码如下：

class TicketSale2 implements Runnable {  
  
 private int ticketNum = 100;  
 Object obj = new Object();  
  
 public void run() {  
 synchronized (obj) { // 注意这里的obj和ticketNum一样是共享数据  
 while (ticketNum > 0) {  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException err) {  
 }  
  
 System.*out*.println(Thread.*currentThread*().getName() + "......" + ticketNum--);  
 }  
 }  
 }  
}

为什么synchronized代码块就可以解决问题呢？这就涉及到它的原理，这是个复杂的问题，我们现在只能用简单的方式了解，后面才会慢慢的深入体会：

synchronized(obj){} 这个代码块，每个线程执行到这里时，都会先做一个判断，obj对象是否已经被别的线程持有，如果不是，当前线程直接持有这个对象，然后带着对象一起进入到代码块内部，即使在代码块内部失去了执行权，cpu切换到了别的线程，那么别的线程执行到这里会发现obj已经被别的线程持有，所以进不来。 直到持有obj对象的线程执行完代码块的内容，释放了obj对象，别的线程才能重新持有obj进而进入代码块执行。

所以，synchronized(obj)代码块中的这个obj相当于一个锁，先来到的线程持有这个锁，除非它释放锁，否则别的线程无法执行代码块中的内容。

### 2.2.2 同步的好坏及前提

我们上面简单的解释了一下synchronized代码块的原理，就会发现它有好处也有坏处。

**好处：**解决了多线程的安全问题。

**坏处：**由于每个线程执行到synchronized代码块都要先进行obj锁的判断，如果obj被别的线程持有，则无法做任何事情，有点浪费资源，降低效率，但是这种降低是在可接受范围内的。

但是有时候你可能发现，即使你用了同步代码块，依然有问题，这是因为你的同步代码块写的有问题，必须记住一点：使用synchronized代码块，有一个前提：

多个线程在同步中必须使用同一个锁，这才是对多个线程同步。

所以，synchronized(obj)这里年的obj必须是多个线程共同的锁，相当于obj必须是线程共享对象，所以，如果你不小心写成了synchronized(new Object())，是肯定会有问题的，这相当于每个线程过来时，都会自己新建一个对象锁并持有，并不是真正的同步。

然后，可能你还会觉得，用了同步代码块之后，不就相当于是单线程了吗？那我们还创建多线程干嘛呢？这个问题必须得明白，对于我们的例子而言，run()方法中的内容比较简单，但是正常情况下，run()方法可以很复杂，这里面应当对共享数据的代码使用synchronized代码块，而不操作共享数据的部分就不要加了。

### 2.2.3 同步函数

了解同步函数之前，我们先做一个同步代码块的练习：

两个用户去同一家银行存钱，每个人存3次，每次存一百，实现这整个过程，并在每次存钱后都打印每个用户的银行账户里里钱的余额。

思路：1. 用户去银行存钱，分析需求发现有两类对象，用户和银行账户，所以有用户类和银行账户类。

2. 银行账户需要记录余额，所以有余额属性。存钱实际上是银行余额在增加，所以银行账户具有添加余额的行为。

用户去银行存款实际是往自己的银行账户里打钱，所以用户持有银行账户对象，用户存款的行为实际上是调用银行账户的存款方法。

3. 多个用户可以同时存款，所以用户的存款行为可以同时发生，应该定义为线程任务。

4. 银行账户的余额属性是多线程的共享数据，且多个线程都会对这个共享数据进行操作，所以会导致安全问题，那么要用同步代码块解决。

5. 哪部分代码要用同步呢？肯定是多线程的共享数据，并对共享数据进行操作的代码。所以，虽然是Customer实现Runnable接口并重写了run方法，但是实际上它是调用了BankAccount类的add方法，对共享数据的操作是在add方法内，故而，同步代码块在add方法内。

代码如下：

class BankAccount {  
 private int amount;  
 private Object object = new Object();  
  
 public void add(int num) {  
 synchronized (object) {  
 amount += num;  
 System.*out*.println(amount);  
 }  
 }  
}  
  
class Customer implements Runnable {  
 private BankAccount bankAccount = new BankAccount();  
  
 public void run() {  
 for (int i = 0; i < 3; i++) {  
 bankAccount.add(100);  
 }  
 }  
}  
  
public class BankSaveMoneyDemo {  
 public static void main(String[] args) {  
 Customer customer1 = new Customer();  
 Customer customer2 = new Customer();  
 Thread t1 = new Thread(customer1);  
 Thread t2 = new Thread(customer2);  
 t1.start();  
 t2.start();  
 }  
}

在上面的代码中，我们看到，add方法内部有一个synchronized(){} 代码块，锁是一个object对象。分析这个代码块，发现它和普通的代码块在使用上并没有很大的区别，只是有了synchronized关键字。

那么，我们就可以考虑把synchronized关键字提出到方法层次，这样，就可以省略掉一个代码块，这种方法其实就是同步方法，又叫同步函数：

public synchronized void add(int num) {  
 amount += num;  
 System.*out*.println(amount);  
}

这样写可能你会有疑问，同步代码块有个重要的部分就是锁，之前我们用一个object对象来作为锁，那么改成同步函数之后，锁在哪里？我们知道，一个非静态函数一定要被对象调用，那么执行到这个函数时，必然会有一个当前对象，那就是函数所持有的this。

注意，同步函数的锁就是函数所持有的this对象。

那我们来用一个例子验证一下，同步函数的锁究竟是不是this。

需求：

依然以卖票为例，启动两个线程，一个线程执行同步代码块，使用明锁，另一个线程执行同步函数，使用this锁。可以用一个boolean类型的flag来实现线程切换。

两个线程执行的任务是一样的，只是不断进行切换。代码如下：

class TicketSeller implements Runnable {  
  
 private int ticketNum = 10000;  
 private final Object obj = new Object();  
 private boolean flag = true;  
  
 public void setFlag(boolean flag) {  
 this.flag = flag;  
 }  
  
 public void run() {  
 if (flag) {  
 while (true) {  
 synchronized (obj) {  
 if (ticketNum > 0) {  
 System.*out*.println(Thread.*currentThread*().getName() + "... 同步代码块..." + ticketNum--);  
 }  
 }  
 }  
 } else {  
 while (true) {  
 sale();  
 }  
 }  
  
 }  
  
 private synchronized void sale() {  
 if (ticketNum > 0) {  
 System.*out*.println(Thread.*currentThread*().getName() + "... 同步函数..." + ticketNum--);  
 }  
 }  
}  
  
public class ThisLockDemo {  
  
 public static void main(String[] args) throws InterruptedException {  
 TicketSeller seller = new TicketSeller();  
 Thread t0 = new Thread(seller);  
 Thread t1 = new Thread(seller);  
 t0.start();  
 Thread.*sleep*(10);  
 seller.setFlag(false);  
 t1.start();  
 }  
}

多运行几次，观察两个进行都运行的结果，你可能会发现，还会有卖0号票的情况，这就意味着，虽然有同步代码块和同步函数，但是并不是真正意义上的同步，因为同步代码块和同步函数使用的不是同一个锁，这时，我们将同步代码块使用的锁改成this试试看，无论运行多少次，都不会再有0号票。

### 2.2.4 static修饰的同步函数

我们都知道，static函数是不会持有this的，因为static函数是基于类的，在创建对象之前就可以访问static函数了。那么static修饰的同步函数的锁必然不会是this，那么它的锁是什么？

这里我们要学习一点新知识。

我们之前学过，在程序执行时：

1. 首先加载main方法所在的类进入方法区。

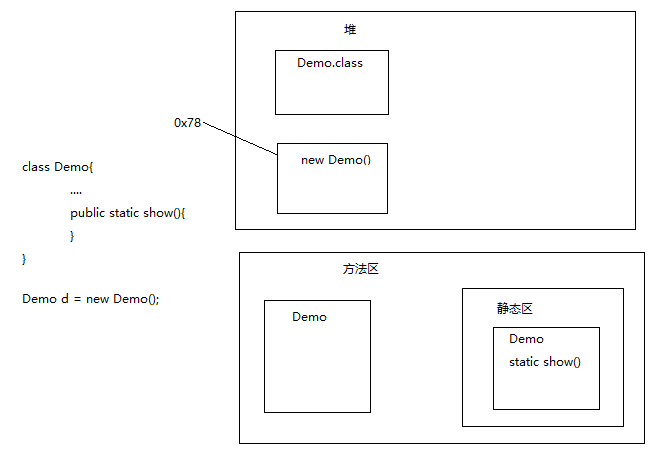
2. 然后main方法进栈。

3. 继续执行main方法中的内容，如果有用到别的类，那么把类加载进入方法区。

4. 如果需要new 对象的话，就在堆内存中分配地址，创建对象。

实际上，在3之后，也就是加载类进入方法区之后，会立即在堆内存中创建一个该类的字节码文件对象。无论是否用new创建对象。该字节码文件对象的表示就是：类名.class

虽然以.class结尾，但是它实际上也是一个堆内存中的对象，只是是字节码文件对象。后面如果用new创建对象的话，都要根据这个字节码文件对象来创建。



实际上，以static修饰的同步函数，它的锁就是对应的字节码文件对象。

### 2.2.5 单例模式的并发访问

还记得我们之前学的单例模式吗？如果有多线程并发访问的话，会不会有问题？

单例模式分为饿汉模式和懒汉模式，我们逐个分析。

1. 饿汉单例模式：

class SingletonHungry {  
  
 private static final SingletonHungry *singleton* = new SingletonHungry();  
  
 private SingletonHungry() {  
 }  
  
 public static SingletonHungry getInstance() {  
 return *singleton*;  
 }  
}

这种情况下，如果有多个线程同时访问，也没有安全问题，最后所有的线程拿到的都是唯一的共同实例。

1. 懒汉单例模式：

class SingletonLazy {  
  
 private static SingletonLazy *singleton* = null;  
  
 private SingletonLazy() {  
 }  
  
 public static SingletonLazy getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new SingletonLazy();  
 }  
 return *singleton*;  
 }  
}

如果多线程同时访问getInstance，假设线程1执行进入if方法后释放资源给线程2，那么线程2也能进入if方法，这种情况下，线程1和线程2都会创建实例，就破坏了单例的本意。那么如何解决问题呢？使用同步，同步函数或者同步代码块都可以。

同步函数：

public static synchronized SingletonLazy getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new SingletonLazy();  
 }  
 return *singleton*;  
}

同步代码块：

public static SingletonLazy getInstance() {  
 synchronized (SingletonLazy.class) {  
 if (*singleton* == null) {  
 *singleton* = new SingletonLazy();  
 }  
 }  
 return *singleton*;  
}

这两种方式效果是一样的，但是它们有一个问题，那就是会降低效率，所有的线程进来后，都要先判断锁，判断锁的代价相对而言是比较高的，那么有没有一种方式，既能解决安全问题又可以稍微提高点效率呢？有的，那就是双重判断if，在判断锁之前，再加一次判断，不满足条件就无需判断锁。

public static SingletonLazy getInstance() {  
 if (*singleton* == null) {  
 synchronized (SingletonLazy.class) {  
 if (*singleton* == null) {  
 *singleton* = new SingletonLazy();  
 }  
 }  
 }  
 return *singleton*;  
}

经过上面的分析，我们可以知道，延迟加载模式在多线程下的代码是比较麻烦的，所以如果写单例，推荐使用饿汉模式。

不过延迟加载在面试中会经常遇到下面问题：

1. 没有多线程时延迟加载的代码。

2. 有多线程时如何解决安全问题。

3. 多线程时同步的锁是什么。

4. 多线程时既解决安全问题，效率又比较高的代码。

### 2.2.6 同步函数和同步代码块的区别

我们已经学了同步代码块和同步函数，那么它们有什么区别呢？分别在什么情况下使用呢？

区别：

同步代码块可以使用任意的对象作为锁，同步函数只能使用this作为锁（非静态情况下）。

根据区别我们就可以分析：

如果一个类中只需要一个锁，就可以使用同步函数，写法比较简单。

如果一个类中要使用多个锁，或者多个类中要使用同一个锁，这时要用同步代码块。

### 2.2.7 死锁

我们知道，无论是同步代码块还是同步函数，都有锁，用来阻塞别的线程。那么什么叫死锁呢？

死锁就是进程中所有的线程都卡在锁上了，无法向下进行。很多种场景都会导致死锁：

场景一：同步嵌套：有多个锁，比如a和b，并且a锁中包含b锁，b锁中包含a锁。

来试着写一个同步嵌套的死锁程序吧，面试当中可能会遇到：

class Task implements Runnable {  
  
 private boolean flag;  
  
 public Task(boolean flag) {  
 this.flag = flag;  
 }  
  
 public void run() {  
 if (flag) {  
 while (true) {  
 synchronized (MyLock.*LOCK\_A*) {  
 System.*out*.println("if......LOCK\_A");  
 synchronized (MyLock.*LOCK\_B*) {  
 System.*out*.println("if......LOCK\_A");  
 }  
 }  
 }  
 } else {  
 while (true) {  
 synchronized (MyLock.*LOCK\_B*) {  
 System.*out*.println("else......LOCK\_B");  
 synchronized (MyLock.*LOCK\_A*) {  
 System.*out*.println("else......LOCK\_A");  
 }  
 }  
 }  
 }  
 }  
}  
  
class MyLock {  
  
 public static final Object *LOCK\_A* = new Object();  
 public static final Object *LOCK\_B* = new Object();  
}  
  
public class DeadLockDemo {  
  
 public static void main(String[] args) {  
 Task t1 = new Task(true);  
 Task t2 = new Task(false);  
 new Thread(t1).start();  
 new Thread(t2).start();  
 }  
}

## 2.3 多线程之间的通信

我们之前学习的多线程的内容，是多个线程在执行同一个线程任务（都在执行同一个run方法）中处理共享资源。

但是，有更多的情况是，多个线程依然是处理同一个资源，但是它们处理的任务不同了。

一个非常典型的例子，就是生产者和消费者。生产者生产商品，消费者来买商品，它们都在处理商品资源，但是处理的方式不同。

分析：

有一种共享资源要描述：商品：名称和编号。

有一个生产者要描述：持有资源，包括生产任务。

有一个消费者要描述：持有资源，包括消费任务。

### 2.3.1先不考虑多线程，只考虑需求

我们先不考虑多线程的问题，来实现代码试试：threadscommunicate/CreatorAndConsumer.java

public class CreatorAndConsumer {  
  
 public static void main(String[] args) {  
 List<Product> products = new ArrayList<>();  
 Creator c1 = new Creator(products);  
 c1.createProduct("bread");  
 c1.createProduct("bread");  
  
 Creator c2 = new Creator(products);  
 c1.createProduct("bread");  
 c1.createProduct("bread");  
  
 Consumer consume = new Consumer(products);  
 consume.consume();  
 consume.consume();  
 consume.consume();  
  
 new Consumer(products).consume();  
 new Consumer(products).consume();  
 new Consumer(products).consume();  
 new Consumer(products).consume();  
 new Consumer(products).consume();  
 new Consumer(products).consume();  
  
 }  
  
}  
  
  
class Creator {  
  
 private static int *num* = 1;  
 private final List<Product> products;  
  
 public Creator(List<Product> products) {  
 this.products = products;  
 }  
  
 public void createProduct(String name) {  
 System.*out*.println("Create new product. id: " + *num* + ", name: " + name);  
 products.add(new Product(*num*++, name));  
 }  
}  
  
@AllArgsConstructor  
class Consumer {  
  
 private List<Product> products;  
  
 public void consume() {  
 if (products != null && products.size() > 0) {  
 System.*out*.println(  
 "Consume a product, id: " + products.get(0).getId() + ", name: " + products.get(0)  
 .getName());  
 products.remove(0);  
 } else {  
 System.*out*.println("No product to consume");  
 }  
 }

上面代码运行没问题，为什么我们先不考虑多线程？这是因为要先理清生产者和消费者以及商品之间的关系。

### 2.3.2单生产者单消费者——消费者不消费

思路清晰后，我们就可以着手创建多线程了，先从创建一个生产者线程和一个消费者线程开始：

threadscommunicate/CreatorAndConsumer2.java

**运行结果：**

运行发现不符合预期啊，生产者生产完一批商品后直接结束了，消费者没进行消费。

**分析问题：**为什么消费者会不消费？

这是因为程序中现在有三个线程：主线程、生产者线程、消费者线程（这里忽略垃圾回收线程），而多线程的运行取决于CPU的时间片分配，消费者没有消费是因为生产者生产一批产品后，消费者线程还没来得及启动主线程就结束了。

**解决思路：**那么如何解决这个问题呢？

我们要让生产者生产后进入休眠状态，同时能够唤醒消费者线程。如何做到？这就涉及了一个新的知识点，那就是等待唤醒机制。

### 2.3.3 等待唤醒机制

我们前面学过线程的状态，有一种状态叫做冻结，那么我们可以让生产者生产之后就冻结，释放执行权给消费者，消费者消费后唤醒生产者，然后冻结自己。如此循环往复就可以了。这就是多线程通信中最常见的等待唤醒机制。

如何让线程进入冻结状态？可以用sleep(),也可以用wait()。但是我们冻结线程后还要主动唤醒它，所以最合适的就是wait()冻结，notify()唤醒。

wait(): 让线程进入冻结状态，并将线程临时存储在线程池中。

notify():唤醒指定线程池中的任意一个线程，是无序的，并且任意一个。

notifyAll(): 唤醒指定线程池中的所有线程。

但是，当我们在synchronized代码块中使用线程锁时，要注意使用锁.wait()和锁.notify()来冻结和唤醒线程。

public class CreatorAndConsumer3 {  
  
 public static void main(String[] args) {  
 List<Product> products = new ArrayList<>();  
 Creator3 creator = new Creator3(products);  
 Consumer3 consumer = new Consumer3(products);  
  
 Thread t1 = new Thread(creator);  
 Thread t2 = new Thread(consumer);  
 t1.start();  
 t2.start();  
 }  
  
}  
  
class Creator3 implements Runnable {  
  
 private int num = 1;  
 private final List<Product> products;  
  
 public Creator3(List<Product> products) {  
 this.products = products;  
 }  
  
 @Override  
 public void run() {  
 createProduct();  
 }  
  
 private void createProduct() {  
 synchronized (products) {  
 try {  
 while (products.size() < 10) {  
 System.*out*.println(*currentThread*().getName() + " - Create id: " + num);  
 products.add(new Product(num++, "bread"));  
 }  
 products.notify();  
 products.wait();  
 System.*out*.println("Creator wait..." + products.size());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}  
  
@AllArgsConstructor  
class Consumer3 implements Runnable {  
  
 private final List<Product> products;  
  
 @Override  
 public void run() {  
 consume();  
 }  
  
 private void consume() {  
 synchronized (products) {  
 try {  
 while (products.size() > 0) {  
 System.*out*.println(  
 *currentThread*().getName() + " - Consume id: " + products.get(0).getId());  
 products.remove(0);  
 }  
 products.notify();  
 products.wait();  
 System.*out*.println("Consumer wait...");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

我们已经用等待唤醒机制来解决了消费者不消费的问题，现在我们来深入的理解一下它的原理。

wait(), notify() 和notifyAll() 必须在同步代码（同步代码块或同步函数）里，因为它们是用来操作同步锁上的线程的状态的。同时在使用这些方法时，必须标识它们所属于的锁，标识方式就是锁对象.wait()，锁对象.notify()，锁对象.notifyAll()。

相同锁的notify()可以唤醒相同锁的wait()。

也许你还记得，wait()和notify()并不是线程类的方法，而是Object类的对象。这就是因为任意一个对象都可能是锁对象，所有的锁对象可能会调用wait()和notify()。所以它们是Object类的对象。

代码示例：threadscommunicate/ CreatorAndConsumer3.java

我们用等待唤醒机制解决了消费者不消费的问题，那么来运行它试试看吧。

**运行结果：**生产者生产，消费者也消费，但过一会儿系统一直挂在那，不运行也不停止

**分析问题：**为什么会有系统hang on？

生产者生产后唤醒消费者同时自己休眠，消费者消费后唤醒生产者同时自己休眠，但是此时，由于生产者的判断条件是：while(lists.size() < 10){}已经在上次生产后判断结束了，也就是说生产者被唤醒时所在的代码是while()之外的，它已经不会再进入循环了，而是直接运行结束，生产者线程结束。但是此时消费者线程处于休眠状态，且再也不会被唤醒，所以最终的状态就是系统一直挂在那里。

**解决思路：**如果解决这个系统一直hang on的问题呢？

我们让生产者被唤醒后继续进入循环，而不是直接结束。那么循环判断条件就一直为true。也就是把while(lists.size() < 10)改成while(true)，至于lists.size()<10的判断则在循环内进行。

### 2.3.4单生产者单消费者——解决系统hang on

根据上面的解决思路，创建新代码：threadscommunicate/ CreatorAndConsumer4.java

**运行结果：**系统运行正常

### 2.3.5多个生产者和消费者

如果我们有多个生产者和多个消费者的话呢？你要怎么实现？我们一步步来分析解决。直接将CreatorAndConsumer4.java中的单个生产者单个消费者改为创建多个生产者线程和多个消费者线程试试。

示例代码：threadscommunicate/ MultipleCreatorsAndConsumers.java

**运行结果：**死锁

**分析问题：**由于有多个生产者了，锁了生产者notify的也许是另一个生产者线程，但是我们希望的是它能notify消费者线程。

**解决思路：**目前没有一个办法能唤醒对方线程，但是可以直接notifyAll()来唤醒所有线程。

根据上面的解决思路，创建新代码：threadscommunicate/ MultipleCreatorsAndConsumers2.java。将notify()改成notifyAll()。

我们运行发现没有问题，至此多线程中的多生产者和多消费者问题解决。

不过，我们分析代码看看有没有浪费资源降低效率的地方？

（1）唤醒所有线程真的有必要吗？我们只需要一个对方线程。

上面两这个问题，我们现在还无能为力，因为目前我们学的知识还不能解决。但是我们马上会学到一种新的实现多线程的方式。

### 2.3.6 多线程之多生产多消费——JDK1.5 Lock 接口

我们在前一节学会了多生产线程和多消费线程中的解决方案，但是到jdk1.4为止，还是有遗留的效率问题：我们没有必要唤醒全部线程，只需要唤醒一个对方线程即可。

既然我们已经明确了要使用jdk1.5中的Lock接口，那就要先查询JDK文档了解一下Lock接口的内容。结论是：Lock接口的出现提供了比synchronized更多的操作。Lock接口中有几个方法，我们这里先用两个：

lock(): 获取锁

unlock(): 释放锁

我们知道同步代码块和同步函数持有锁，一个线程进来后会获取锁然后执行代码，然后释放锁，但是同步代码块和同步函数对锁的获取和释放都是隐式的。这样其实并不利于我们对锁和同步的理解。

而JDK1.5中Lock接口的出现，将锁单独封装成为对象，并提供了对锁的显示操作。根本上来说，Lock接口就是同步的替代：

将线程中的同步更换为Lock接口的形式：注意：lock()和unlock()之间的代码都是同步代码。

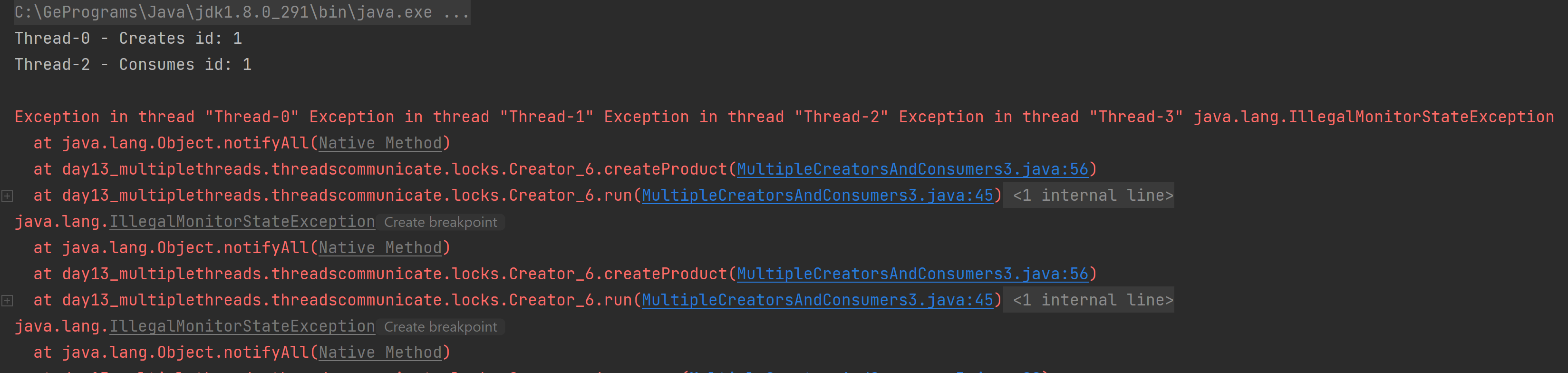
注意，在使用JDK中的接口时，重写自己的方法之前，先了解一下JDK已经提供的实现类。Lock接口的已知实现类就有：

ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock.

那我们就直接使用ReentrantLock试试。将之前的多生产多消费的终结代码用Lock接口替代，示例代码：threadscommunicate/locks/MultipleCreatorsAndConsumers3.java

public class MultipleCreatorsAndConsumers3 {  
  
 public static void main(String[] args) {  
 List<Product> products = new ArrayList<>();  
 Lock lock = new ReentrantLock();  
 Creator\_6 creator = new Creator\_6(products, lock);  
 Consumer\_6 consumer = new Consumer\_6(products, lock);  
  
 Thread t11 = new Thread(creator);  
 Thread t12 = new Thread(creator);  
 Thread t21 = new Thread(consumer);  
 Thread t22 = new Thread(consumer);  
 t11.start();  
 t12.start();  
 t21.start();  
 t22.start();  
 }  
}  
  
  
class Creator\_6 implements Runnable {  
  
 private int num = 1;  
 private final List<Product> products;  
 private final Lock lock;  
  
 public Creator\_6(List<Product> products, Lock lock) {  
 this.products = products;  
 this.lock = lock;  
 }  
  
 @Override  
 public void run() {  
 createProduct();  
 }  
  
 private void createProduct() {  
 lock.lock();  
 try {  
 while (true) {  
 if (products.size() < 1) {  
 System.*out*.println(*currentThread*().getName() + " - Creates id: " + num);  
 products.add(new Product(num++, "bread"));  
 } else {  
 products.notifyAll();  
 products.wait();  
 }  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
}  
  
@AllArgsConstructor  
class Consumer\_6 implements Runnable {  
  
 private final List<Product> products;  
 private final Lock lock;  
  
 @Override  
 public void run() {  
 consume();  
 }  
  
 private void consume() {  
 lock.lock();  
 try {  
 while (true) {  
 if (products.size() > 0) {  
 System.*out*.println(  
 *currentThread*().getName() + " - Consumes id: " + products.get(0).getId());  
 System.*out*.println();  
 products.remove(0);  
 } else {  
 products.notifyAll();  
 products.wait();  
 }  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
  
 }  
}

运行时发现，会有错误：



这是因为，我们使用了wait和notify方法，也就是等待唤醒机制，但是我们知道它们是必须使用在同步代码块中的，我们使用了lock以后，就没有同步代码块了，这就是错误原因。

### 2.3.7 多线程之多生产多消费——JDK1.5 Condition 接口

我们上面遇到的问题是，wait和notify必须写在同步代码块或者同步函数中，使用了Lock之后，我们没有同步代码块了，会报错。

为什么会报错呢？分析一下，这是因为同步代码块或同步函数有给定的同步锁，但是现在使用Lock之后就没有了，wait就失去了所属的同步锁。

同步升级成Lock后，其中的锁不再是任意对象，而是Lock类型的对象，那么和任意锁绑定的监视器方法是不是也会有对应的升级呢？

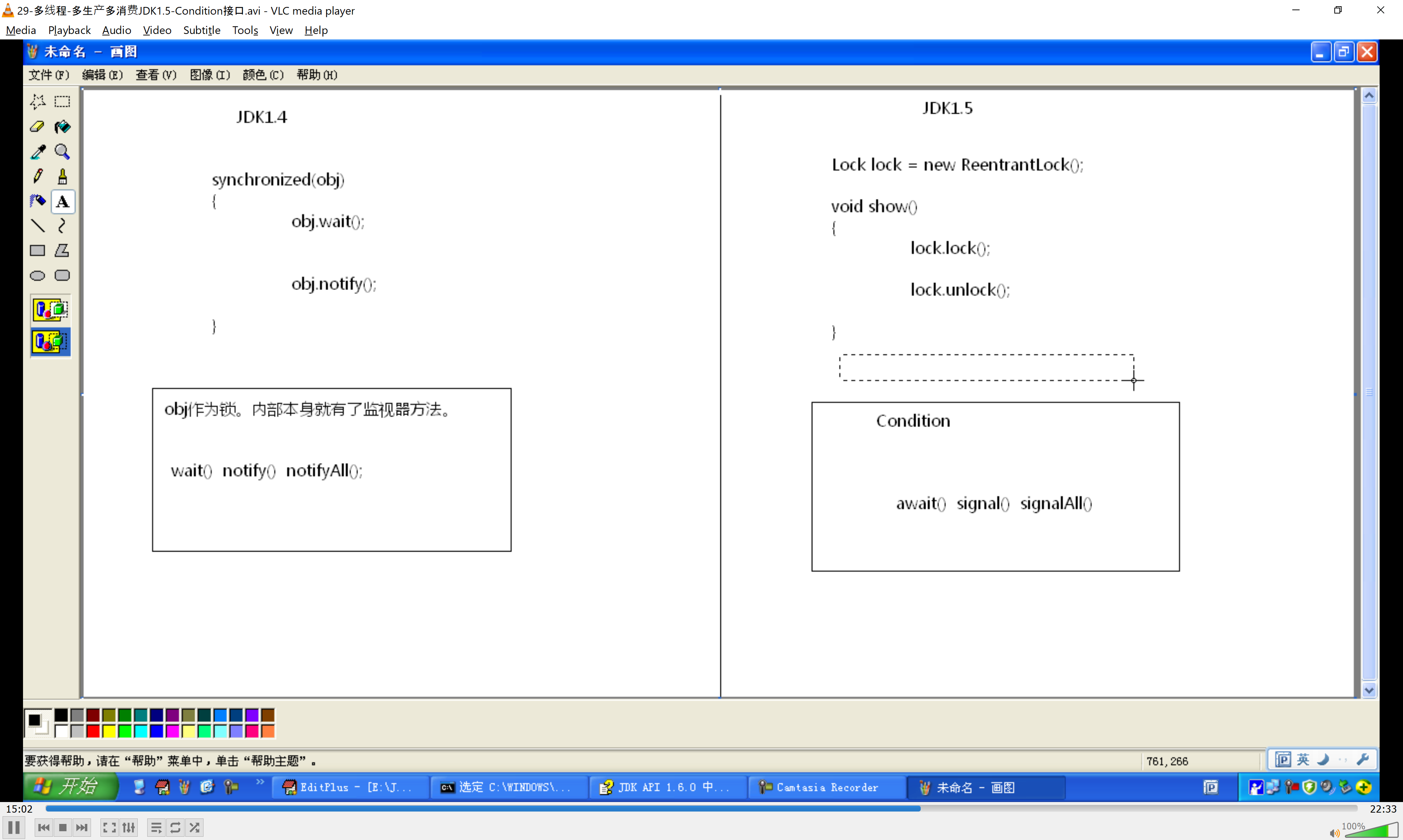
是的，这就是Condition接口。Condition接口将Object的监视器方法（wait(), notify(), notifyAll()）分解成了截然不同的对象，以便将这些对象与任意Lock对象组合使用。总而言之，Condition接口就替代了Object的监视器方法。

Await()方法：使当前线程处于等待状态

Signal()：唤醒等待线程中的一个线程。

SignalAll(): 唤醒所有在等待的线程。

那么我们来对比一下，原来的同步和现在的Lock接口之间实现同步的方式：



通过上图我们知道了，现在使用Lock对象可以很容易的显示获取锁、执行同步代码、以及释放锁，而它对应的监视器方法则由Condition对象提供，那么问题来了，Lock对象和Condition对象怎么才能建立联系然后一起组合使用呢？

实际上，Lock接口有一个方法，newCondition()可以创建一个Condition对象。

那么这个condition对象就是监视器对象，它和锁对象是独立的，而且它有自己的监视器方法，await(), signal(), signalAll(),且由于它是锁对象创建出来的对象，所以它的监视器方法对锁有效。

由此可知，我们可以将snychronized代码换成Lock.lock()….和Lock.unlock()，然后同步代码中使用的wait()换成await(), notify()换成signal(), notifyAll()换成signalAll().这就是JDK1.5提供的替换JDK1.4 中同步代码块或同步函数的方式。它们的作用是完全一致的，只是表现形式不同。

创建新代码：threadscommunicate/locks/ MultipleCreatorsAndConsumersLock.java

运行结果：一切正常

这时你就会想，Lock只是改变了表现形式吗？难道就没有别的好处？当然是有的.

JDK1.4中使用的监视器方法和锁是绑定的，也就是所锁对象只有一组监视器方法。但是JDK1.5中锁对象和监视器对象分离了，也就是说，一个所可以有多个监视器对象了，那么我们可以实现一个锁，对应两个监视器，一个生产者监视器，一个消费者监视器，那么在使用时，生产者可以直接唤醒消费者线程。

示例代码：threadscommunicate/locks/ MultipleCreatorsAndConsumersLock2.java