# 【多线程】

## 主要内容

1. 进程和线程
2. 线程的定义和创建
3. 线程生命周期
4. 线程控制
5. 线程同步

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 程序、进程和线程 | 理解 |
| 线程定义和创建1：继承Thread类 | 掌握 |
| 线程定义和创建2：实现Runable接口 | 掌握 |
| 线程定义和创建3：实现Callable接口 | 掌握 |
| 线程生命周期 | 掌握 |
| 线程控制 | 了解 |
| 线程同步及其作用 | 掌握 |
| 线程同步1-同步代码块 | 掌握 |
| 线程同步2-同步方法 | 掌握 |
| 线程同步3-Lock锁 | 掌握 |
| ReentrantLock | 掌握 |
| ReentrantReadWriteLock | 掌握 |

## 第一节 线程的定义和创建

### 1.1 进程和线程

* 程序Program
  + 程序是一段静态的代码，它是应用程序执行的蓝本
* 进程Process
  + 进程是指一种正在运行的程序，有自己的地址空间



* 进程的特点
  + 动态性
  + 并发性
  + 独立性
  + 并发和并行的区别
    - 并行：多个CPU同时执行多个任务
    - 并发：一个CPU（采用时间片）同时执行多个任务

|  |
| --- |
| **生活案例：并发和并行的区别**   * 并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行。所以无论从微观还是从宏观来看，二者都是一起执行的。      * 并发(concurrency)：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的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行 |

* 线程Thread
  + 进程内部的一个执行单元，它是程序中一个单一的顺序控制流程。
  + 线程又被称为轻量级进程(lightweight process)
  + 如果在一个进程中同时运行了多个线程，用来完成不同的工作，则称之为多线程
* 线程特点
  + 轻量级进程
  + 独立调度的基本单位
  + 共享进程资源
  + 可并发执行



* 线程和进程的区别

|  |  |  |
| --- | --- | --- |
| **区别** | **进程** | **线程** |
| 根本区别 | 作为资源分配的单位 | 调度和执行的单位 |
| 开 销 | 每个进程都有独立的代码和数据空间(进程上下文)，进程间的切换会有较大的开销。 | 线程可以看成时轻量级的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换的开销小。 |
| 所处环境 | 在操作系统中能同时运行多个任务(程序) | 在同一应用程序中有多个顺序流同时执行 |
| 分配内存 | 系统在运行的时候会为每个进程分配不同的内存区域 | 除了CPU外，不会为线程分配内存（线程所使用的资源是它所属的进程的资源），线程组只能共享资源 |
| 包含关系 | 没有线程的进程是可以被看作单线程的，如果一个进程内拥有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的。 | 线程是进程的一部分，所以线程有的时候被称为是轻权进程或者轻量级进程。 |

### 1.2 线程定义和创建1：继承Thread类

* Thread类是Java提供的线程顶级类，继承Thread类可快速定义线程。

#### 【示例1】 使用多线程实现龟兔赛跑

|  |
| --- |
| **package** com.bjsxt.threadDemo1;  **public class** Test1 {  **public static void** main(String[] args) {  */\*  \* 目标:通过多线程形式实现龟兔赛跑  \* 步骤: 1准备乌龟线程  \* 2准备兔子线程  \* 3启动两个线程 调用start方法 不要直接调用run方法  \* \*/    /\*Thread.currentThread方法获得当前线程对象  \* 哪个线程在执行这一行代码 返回的对象就是这个线程  \*  \* \*/* Thread thread = Thread.*currentThread*();  System.***out***.println(thread.getName());  System.***out***.println(thread.getPriority());     RunnerThread wugui =**new** RunnerThread();  RunnerThread tuzi =**new** RunnerThread();   *// 设置线程的优先级* wugui.setPriority(1);  tuzi.setPriority(10);   *//获得线程的优先级* System.***out***.println(wugui.getPriority());  System.***out***.println(tuzi.getPriority());   *//设置线程名字* wugui.setName(**"乌龟"**);  tuzi.setName(**"兔子"**);   *// 启动线程* wugui.start();  tuzi.start();     } } */\* \* 1继承Thread类 \* 2重写run方法 定义线程任务的方法 将线程要执行的工作放在run方法中即可 \* \* \*/* **class** RunnerThread **extends** Thread{   **public** RunnerThread(){}  **public** RunnerThread(String name){  **super**(name);  }   @Override  **public void** run() {  Thread thread = Thread.*currentThread*();  System.***out***.println(thread.getName());  System.***out***.println(thread.getPriority());   **for** (**int** i = 1; i <=100 ; i++) {  System.***out***.println(getName() +**"跑到了第"**+i+**"米"**);  }  } } |

* run() 线程体，线程要完成的任务
* start() 启动线程，线程进入就绪队列，等待获取CPU并执行
* 之前讲解的程序都是单线程的

### 1.3 线程定义和创建2：实现Runnable接口

#### 【示例2】使用多线程实现龟兔赛跑2

|  |
| --- |
| **package** com.bjsxt.threadDemo1;  **public class** Test2 {  */\*  \* 将任务和线程对象分开  \* 任务 子弹  \* 线程对象 枪 Thread  \*  \* \*/* **public static void** main(String[] args) {  *// 准备任务对象* RunnerRunnable game=**new** RunnerRunnable();  *// 将任务对象放入线程对象中* Thread wuguiThread =**new** Thread(game);  Thread tuziThread=**new** Thread(game);   *// 设置线程对象的名字* wuguiThread.setName(**"乌龟"**);  tuziThread.setName(**"兔子"**);   *// 启动线程* wuguiThread.start();  tuziThread.start();  }  } **class** RunnerRunnable **implements** Runnable{  @Override  **public void** run() {  Thread thread = Thread.*currentThread*();  **for** (**int** i = 1; i <=100 ; i++) {  System.***out***.println(thread.getName()+**"跑到了第"**+i+**"米"**);  };  } } |

练习

* 两种方式的优缺点  
   方式1：继承Thread类  
   缺点：Java单继承，无法继承其他类； 优点：代码稍微简单  
   方式2：实现Runnable接口  
   优点 还可以去继承其他类 便于多个线程共享同一个资源；缺点：代码略有繁琐  
   实际开发中，方式2使用更多一些
* 可以使用匿名内部类来创建Runnable对象；更可以使用lambda表达式来完成。

|  |
| --- |
| Runnable runnable= ()->{   **while**(**true**){  System.**out**.println(**"乌龟领先了............"**+Thread.*currentThread*().getName());  } }; |

* 已经学习的线程Thread的属性和方法

|  |  |
| --- | --- |
| **字段摘要** | |
| static int | **[MAX\_PRIORITY](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "MAX_PRIORITY)**        线程可以具有的最高优先级。 |
| static int | **[MIN\_PRIORITY](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "MIN_PRIORITY)**         线程可以具有的最低优先级。 |
| static int | **[NORM\_PRIORITY](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "NORM_PRIORITY)**     分配给线程的默认优先级。 |

|  |  |
| --- | --- |
| **方法摘要** | |
| static [Thread](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \o "java.lang 中的类) | **[currentThread](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "currentThread())**()       返回对当前正在执行的线程对象的引用。 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[getName](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "getName())**()     返回该线程的名称。 |
| int | **[getPriority](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "getPriority())**()   返回线程的优先级。 |
| void | **[run](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "run())**() 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。 |
| void | **[setName](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "setName(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) name)        改变线程名称，使之与参数 name 相同。 |
| void | **[setPriority](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "setPriority(int))**(int newPriority)        更改线程的优先级。 |
| void | **[start](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html" \l "start())**()        使该线程开始执行；Java 虚拟机调用该线程的 run 方法。 |

### 1.4 线程定义和创建3：实现Callable接口

JDK1.5后推出了第三种定义线程的方式，实现Callable接口。该方式最大的变化是可以有返回值，并且可以抛出检查异常。

#### 【示例3】使用多线程获取随机数

|  |
| --- |
| **package** com.bjsxt.threadDemo1;  **import** java.util.Random; **import** java.util.concurrent.Callable; **import** java.util.concurrent.ExecutionException; **import** java.util.concurrent.Future; **import** java.util.concurrent.FutureTask;  **public class** Test4 {  **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  *// 1准备任务对象* RandomCallable rc =**new** RandomCallable();  *// 2将任务对象装入多个不同的线程对象* FutureTask<Integer> future1=**new** FutureTask(rc);  FutureTask<Integer> future2=**new** FutureTask(rc);  Thread t1=**new** Thread(future1,**"线程1"**);  Thread t2=**new** Thread(future2,**"线程2"**);  *// 3启动线程* t1.start();  t2.start();  *// 4获取线程运行结果* System.***out***.println(future1.get());  System.***out***.println(future2.get());    *// 取消线程任务  /\* System.out.println("线程是否取消"+future.isCancelled());  future.cancel(true);  System.out.println("线程是否取消"+future.isCancelled());  System.out.println("线程是否完成"+future.isDone());  if(!future.isCancelled()){  Integer integer = future.get();  System.out.println(integer);  System.out.println("线程是否完成"+future.isDone());  }\*/* } } *// 定义任务 定义一个线程 返回一个0-10随机生成整数* **class** RandomCallable **implements** Callable<Integer>{  @Override  **public** Integer call() **throws** Exception {  **int** i = **new** Random().nextInt(10);  System.***out***.println(Thread.*currentThread*().getName()+**"生成的随机数为:"**+i);  **return** i;  } } |

和Runnable一样，同样可以使用Lambda表达式来实现，更加简洁。

|  |
| --- |
| Callable<Integer> callable = ()->{  Thread.*sleep*(5000);  **return new** Random().nextInt(10); }; |

**第三种方式：实现Callable接口**

与实行Runnable相比， Callable功能更强大些

* + 方法名不同
  + 可以有返回值，支持泛型的返回值
  + 可以抛出检查异常
  + 需要借助FutureTask，比如获取返回结果

**Future接口**

* + 可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。
  + FutrueTask是Futrue接口的唯一的实现类
  + FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值

### 本节作业

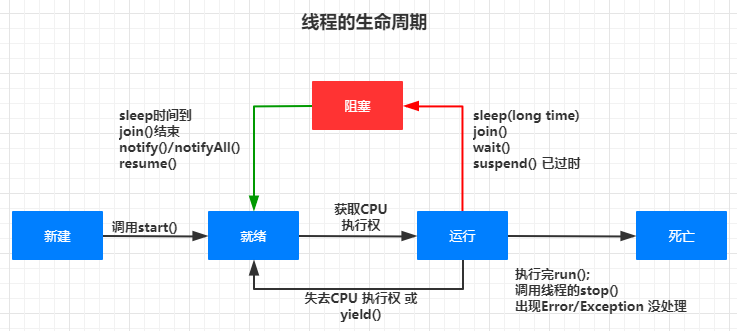
1. 进程和线程的联系和区别
2. 通过继承Thread类定义和创建线程
3. 通过实现Runnable接口来定义线程实现龟兔赛跑
4. 通过实现Callable接口来定义线程实现获取随机数
5. 通过两种方式实现多个窗口卖100张票

1Thread

|  |
| --- |
| **package** com.bjsxt.threadDemo;   **public class** Test {  **public static void** main(String[] args) {  TickedThread tt1=**new** TickedThread();  tt1.setName(**"窗口1"**);  TickedThread tt2=**new** TickedThread();  tt2.setName(**"窗口2"**);   tt1.start();  tt2.start();  } } **class** TickedThread **extends** Thread{  **static int** *t* =100;  @Override  **public void** run() {   **while**(*t*>0){  System.***out***.println(getName()+**"售出了第"**+*t*+**"张票"**);  *t*--;  }  } } |

## 第二节 线程控制

### 2.1 线程的生命周期



* 新生状态：
  + 用new关键字建立一个线程对象后，该线程对象就处于新生状态。
  + 处于新生状态的线程有自己的内存空间，通过调用start进入就绪状态
* 就绪状态：
  + 处于就绪状态线程具备了运行条件，但还没分配到CPU，处于线程就绪队列，等待系统为其分配CPU
  + 当系统选定一个等待执行的线程后，它就会从就绪状态进入执行状态，该动作称之为“cpu调度”。
* 运行状态：
  + 在运行状态的线程执行自己的run方法中代码，直到等待某资源而阻塞或完成任务而死亡。
  + 如果在给定时间片内没执行结束，就会被系统给换下来回到等待执行状态。
* 阻塞状态：
  + 处于运行状态的线程在某些情况下，如执行了sleep（睡眠）方法，或等待I/O设备等资源，将让出CPU并暂时停止自己的运行，进入阻塞状态。
  + 在阻塞状态的线程不能进入就绪队列。只有当引起阻塞的原因消除时，如睡眠时间已到，或等待的I/O设备空闲下来，线程便转入就绪状态，重新到就绪队列中排队等待，被系统选中后从原来停止的位置开始继续运行。
* 死亡状态：
  + 死亡状态是线程生命周期中最后一个阶段。线程死亡原因有三个。一个是正常运行的线程完成了它的全部工作；另一个是线程被强制性地终止，如通过执行stop方法来终止一个线程[不推荐使用】，三是线程抛出未捕获的异常

|  |
| --- |
| **总结：线程生命周期**   * 运行状态的出口：阻塞、就绪、终止 * 就绪状态的入口：新生、运行、阻塞 * 讲解了线程同步、线程通信后会增加更多的状态：增加了两种阻塞状态 * CPU的执行权 占用CPU运行的权利 运行状态 * CPU的等待权 等待CPU执行的权利 就绪状态 |

### 2.2 线程控制

理解了线程生命周期的基础上，可以使用Java提供的线程控制命令对线程的生命周期进行干预。

* **join ()**

阻塞指定线程等到另一个线程完成以后再继续执行

* **sleep ()**

使线程停止运行一段时间，让出CPU，将处于阻塞状态

如果调用了sleep方法之后，没有其他等待执行的线程，这个时候当前线程不会马上恢复执行！

实际开发中经常使用Thread.sleep()来模拟线程切换，暴露线程安全问题。

* **yield ()**

让当前正在执行线程暂停，不是阻塞线程，而是将线程转入就绪状态

如果调用了yield方法之后，没有其他等待执行的线程，这个时候当前线程就会马上恢复执行！

* **setDaemon()**

可以将指定的线程设置成后台线程

创建后台线程的线程结束时，后台线程也随之消亡

只能在线程启动之前把它设为后台线程

* **interrupt()**

并没有直接中断线程，而是需要被中断线程自己处理

* **stop()**

结束线程，不推荐使用

#### 【示例4】使用join()阻塞当前线程

|  |
| --- |
| **package** com.bjsxt.threadDemo3;  **public class** Test1 {  **public static void** main(String[] args) **throws** InterruptedException {  RunnerThread runnerThread = **new** RunnerThread();  runnerThread.setName(**"兔子"**);  RunnerThread runnerThread2 = **new** RunnerThread();  runnerThread2.setName(**"乌龟"**);  runnerThread.start();  runnerThread2.start();  *// 让其他线程加入当前线程  // 让乌龟和兔子线程加入主方法线程* runnerThread.join();  runnerThread2.join();  System.***out***.println(**"比赛结束"**);    } } **class** RunnerThread **extends** Thread{  **public** RunnerThread(){   }  **public** RunnerThread(String name){  *// 调用父类构造方法,设置名字* **super**(name);  }   @Override  **public void** run() {  **for** (**int** i = 1; i <= 100; i++) {  System.***out***.println(getName()+**"跑到了第"**+i+**"米"**);  }  } } |

#### 【示例5】使用sleep()让线程休眠

|  |
| --- |
| **package** com.bjsxt.threadDemo3;  **public class** Test1 {  **public static void** main(String[] args) **throws** InterruptedException {  RunnerThread runnerThread = **new** RunnerThread();  runnerThread.setName(**"兔子"**);  RunnerThread runnerThread2 = **new** RunnerThread();  runnerThread2.setName(**"乌龟"**);  runnerThread.start();  runnerThread2.start();  *// 让其他线程加入当前线程  // 让乌龟和兔子线程加入主方法线程* runnerThread.join();  runnerThread2.join();  System.***out***.println(**"比赛结束"**);   } } **class** RunnerThread **extends** Thread{  **public** RunnerThread(){   }  **public** RunnerThread(String name){  *// 调用父类构造方法,设置名字* **super**(name);  }   @Override  **public void** run() {  **for** (**int** i = 1; i <= 100; i++) {  System.***out***.println(getName()+**"跑到了第"**+i+**"米"**);  **try** {// 让当前线程停止\*\*\*毫秒  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } } |

#### 【示例6】使用yield()礼让线程

|  |
| --- |
| **package** com.bjsxt.threadDemo3;  **public class** Test1 {  **public static void** main(String[] args) **throws** InterruptedException {  RunnerThread runnerThread = **new** RunnerThread();  runnerThread.setName(**"兔子"**);  runnerThread.start();   **for** (**int** i = 1; i <= 100; i++) {  *// 让当前线程从运行状态 进入就绪状态,等待调度后再运行* Thread.*yield*();  System.***out***.println(**"乌龟跑到了第"**+i+**"米"**);  }    } } **class** RunnerThread **extends** Thread{  **public** RunnerThread(){   }  **public** RunnerThread(String name){  *// 调用父类构造方法,设置名字* **super**(name);  }   @Override  **public void** run() {  **for** (**int** i = 1; i <= 100; i++) {  System.***out***.println(getName()+**"跑到了第"**+i+**"米"**);  }  } } |

#### 【示例7】使用setDaemon()设置守护线程

|  |
| --- |
| **package** com.bjsxt.threadDemo3;  **public class** Test1 {  **public static void** main(String[] args) **throws** InterruptedException {  RunnerThread runnerThread = **new** RunnerThread();  runnerThread.setName(**"兔子"**);  *// 设置伴随线程* runnerThread.setDaemon(**true**);  runnerThread.start();  **for** (**int** i = 1; i <= 100; i++) {  System.***out***.println(**"乌龟跑到了第"**+i+**"米"**);  }  } } **class** RunnerThread **extends** Thread{  **public** RunnerThread(){   }  **public** RunnerThread(String name){  *// 调用父类构造方法,设置名字* **super**(name);  }   @Override  **public void** run() {  **for** (**int** i = 1; **true**; i++) {  System.***out***.println(getName()+**"跑到了第"**+i+**"米"**);  }  } } |

#### 【示例8】中断线程

|  |
| --- |
| **package** com.bjsxt.threadDemo3;  **public class** Test1 {  **public static void** main(String[] args) {  Runner runner=**new** Runner();  runner.start();   **for** (**int** i = 1; i <=200 ; i++) {  System.***out***.println(**"兔子跑到了第"**+i+**"米"**);  }  *// 停止线程 强制的  //runner.stop();  // 打断线程 征得线程同意的* runner.interrupt();  } } **class** Runner **extends** Thread{  @Override  **public void** run() {  **int** i=1;  **while**(!isInterrupted()){  System.***out***.println(**"乌龟跑到了第"**+i+++**"米"**);  }  } } |

### 本节作业

1. 线程的生命周期（就绪状态的三个来源，运行状态的三个去处）
2. sleep()和yield()方法的异同之处
3. 使用interrupt()中断线程

4.电影院有两个售票窗口,同时卖票

1各自卖100张

|  |
| --- |
| **package** com.bjsxt.threadDemo1;  **public class** Test3 {  **public static void** main(String[] args) {  TicketRunnable tr =**new** TicketRunnable();  Thread t1 =**new** Thread(tr,**"窗口1"**);  Thread t2 =**new** Thread(tr,**"窗口2"**);  t1.start();  t2.start();  } } *// 卖票的任务* **class** TicketRunnable **implements** Runnable{  @Override  **public void** run() {  **for** (**int** i =1 ; i <=100 ; i++) {  System.***out***.println(Thread.*currentThread*().getName()+**"卖出了第"**+i+**"张票"**);  }  } } |

2共卖100张

|  |
| --- |
| **public class** Test3 {  **public static void** main(String[] args) {  TicketRunnable tr1 =**new** TicketRunnable();  TicketRunnable tr2 =**new** TicketRunnable();  Thread t1 =**new** Thread(tr1,**"窗口1"**);  Thread t2 =**new** Thread(tr2,**"窗口2"**);  t1.start();  t2.start();  } } *// 卖票的任务* **class** TicketRunnable **implements** Runnable{  **static int** *i* =1;  @Override  **public void** run() {  **for** ( ; *i* <=100 ; *i*++) {  System.***out***.println(Thread.*currentThread*().getName()+**"卖出了第"**+*i*+**"张票"**);  }  } } |

## 第三节 线程同步

### 3.1 线程安全问题

* **应用场景：**
  + 多个窗口同时销售100张票,售完为止
* **分析**
  + 使用多线程解决
  + 开发一个售票线程类
  + 因为多线程共享同100张票
* **思路**
  + 创建售票任务对象,内部准备成员变量,存储票数
  + 创建创建售票线程
  + 创建测试类,让多个线程同时售票

#### 【示例9】引入线程同步

|  |
| --- |
| **package** com.bjsxt.threadDemo4;  **public class** Test1{  **public static void** main(String[] args) {  TicketRunnable tr1 =**new** TicketRunnable();  TicketRunnable tr2 =**new** TicketRunnable();  Thread t1 =**new** Thread(tr1,**"窗口1"**);  Thread t2 =**new** Thread(tr2,**"窗口2"**);  t1.start();  t2.start();  } } */\* \* 问题:线程安全问题 \* 原因:多个线程共享相同的资源 \* \*/ // 卖票的任务* **class** TicketRunnable **implements** Runnable{    @Override  **public void** run() {  **while** ( *i* <=1000 ) {  **if**(*i*<=1000){  String info =Thread.*currentThread*().getName()+**"卖出了第"**+*i*+**"张票"**;  System.***out***.println(info);  *i*++;  }**else**{  System.***out***.println(**"卖光了"**);  }  }  } } |



分析：使用Thread.sleep()的目的在于模拟线程切换，在一个线程判断完余额后，不是立刻取款，而是让出CPU，这样另外一个线程获取CPU，并且进行数据判断。线程安全问题就这么产生了。如果保证安全，必须判断余额和取款的语句必须被一个线程执行完才能让另外一个线程执行。当多个线程共享同一个资源时,就容易出现线程安全问题

**当多个线程访问同一个数据时，容易出现线程安全问题。需要让线程同步，保证数据安全。**

* **线程同步（thread synchronized）**
  + 当两个或两个以上线程访问同一资源时，需要某种方式来确保资源在某一时刻只被一个线程使用
* **线程同步的实现方案**

1. 同步代码块锁
   * + synchronized (obj){ }
2. 同步方法锁
   * + private synchronized void makeWithdrawal(int amt) {}
3. Lock锁
   * + ReentrantLock、ReentrantReadWriteLock
4. volatile+CAS无锁化方案

### 3.2 同步代码块

#### 【示例10】使用同步代码块实现线程同步

|  |
| --- |
| **package** com.bjsxt.threadDemo4;  **public class** Test1{  **public static void** main(String[] args) {  TicketRunnable tr1 =**new** TicketRunnable();  TicketRunnable tr2 =**new** TicketRunnable();  Thread t1 =**new** Thread(tr1,**"窗口1"**);  Thread t2 =**new** Thread(tr2,**"窗口2"**);  t1.start();  t2.start();  } } */\* \* 问题:线程安全问题 \* 原因:多个线程共享相同的资源 \* 解决:让线程同步 \* 方法: \* 1synchronized 同步 \* 1.同步代码块 \* 使用synchronized (suo){}将代码进行同步保护 \* 任何对象都可以是同步锁,任何对象在内存上都可以记录一个 on off 的状态 \* 多个线程必须使用同一个锁 \* \* 2.同步代码函数(同步方法) \*  \* 2Lock锁 \* \*/ // 卖票的任务* **class** TicketRunnable **implements** Runnable{  **static int** *i* =1;  **static** Object *suo*=**new** Object();  @Override  **public void** run() {  **while** ( *i* <=1000 ) {  **synchronized** (**"锁"**){  **if**(*i*<=1000){  String info =Thread.*currentThread*().getName()+**"卖出了第"**+*i*+**"张票"**;  System.***out***.println(info);  *i*++;  }**else**{  System.***out***.println(**"卖光了"**);  }   }  }  } } |

**总结1：认识同步监视器（锁子）**

   synchronized(同步监视器){ }

1. 必须是引用数据类型，不能是基本数据类型
2. 在同步代码块中可以改变同步监视器对象的值，不能改变其引用
3. 尽量不要String和包装类Integer做同步监视器.如果使用了，只要保证代码块中不对其进行任何操作也没有关系
4. 一般使用共享资源做同步监视器即可
5. 也可以创建一个专门的同步监视器，没有任何业务含义
6. 建议使用final修饰同步监视器

**总结2：同步代码块的执行过程**

1. 第一个线程来到同步代码块，发现同步监视器open状态，需要close,然后执行其中的代码
2. 第一个线程执行过程中，发生了线程切换（阻塞 就绪），第一个线程失去了cpu，但是没有开锁open
3. 第二个线程获取了cpu，来到了同步代码块，发现同步监视器close状态，无法执行其中的代码，第二个线程也进入阻塞状态
4. 第一个线程再次获取CPU,接着执行后续的代码；同步代码块执行完毕，释放锁open
5. 第二个线程也再次获取cpu，来到了同步代码块，发现同步监视器open状态，重复第一个线程的处理过程（加锁）

**强调：同步代码块中能发生CPU切换吗？能！！！ 但是后续的被执行的线程也无法执行同步代码块（锁仍旧close）**



**总结3：线程同步 优点和缺点**

   优点：安全

   缺点：效率低下 ,可能出现死锁



**总结4：其他**

1. 多个代码块使用了同一个同步监视器（锁），锁住一个代码块的同时，也锁住所有使用该锁的所有代码块，其他线程无法访问其中的任何一个代码块
2. 多个代码块使用了同一个同步监视器（锁），锁住一个代码块的同时，也锁住所有使用该锁的所有代码块， 但是没有锁住使用其他同步监视器的代码块，其他线程有机会访问其他同步监视器的代码块

### 3.3 同步方法

#### 【示例11】使用同步方法实现线程同步

同步实例方法

|  |
| --- |
| **package** com.bjsxt.threadDemo4;  **public class** Test1{  **public static void** main(String[] args) {  TicketRunnable tr1 =**new** TicketRunnable();  Thread t1 =**new** Thread(tr1,**"窗口1"**);  Thread t2 =**new** Thread(tr1,**"窗口2"**);  t2.start();  t1.start();  } } */\* \* 问题:线程安全问题 \* 原因:多个线程共享相同的资源 \* 解决:让线程同步 \* 方法: \* 1synchronized 同步 \* 1.同步代码块 \* 使用synchronized (suo){}将代码进行同步保护 \* 任何对象都可以是同步锁,任何对象在内存上都可以记录一个 on off 的状态 \* 多个线程必须使用同一个锁 \* \* 2.同步代码函数(同步方法) \* 同步实例方法时 锁是this \*  \* \* 2Lock锁 \* \*/ // 卖票的任务* **class** TicketRunnable **implements** Runnable{  **static int** *i* =1;  @Override  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName()+**"开始卖票"**);  **while** (*i* <=1000) {  sellTicket();  }  }   **public synchronized void** sellTicket(){  **if**(*i*<=1000){  String info =Thread.*currentThread*().getName()+**"卖出了第"**+*i*+**"张票"**;  System.***out***.println(info);  *i*++;  }**else**{  System.***out***.println(**"售罄了"**);  }  } } |

同步静态方法

|  |
| --- |
| **package** com.bjsxt.threadDemo4;  **public class** Test1{  **public static void** main(String[] args) {  TicketRunnable tr1 =**new** TicketRunnable();  TicketRunnable tr2 =**new** TicketRunnable();  Thread t1 =**new** Thread(tr1,**"窗口1"**);  Thread t2 =**new** Thread(tr2,**"窗口2"**);  t2.start();  t1.start();  } } */\* \* 问题:线程安全问题 \* 原因:多个线程共享相同的资源 \* 解决步骤: \* 1确定要同步的代码 \* 2同步处理 \* 3保证多个线程共享一个锁 \* 方法: \* 1 synchronized 同步 \* 1.同步代码块 \* 使用synchronized (suo){}将代码进行同步保护 \* 任何对象都可以是同步锁,任何对象在内存上都可以记录一个 on off 的状态 \* 多个线程必须使用同一个锁 \* \* 2.同步代码函数(同步方法) \* 同步实例方法时 锁是this \* 同步静态方法时 锁是当前类的字节码对象 \* \* 2Lock锁 \* \*/ // 卖票的任务* **class** TicketRunnable **implements** Runnable{  **static int** *i* =1;  @Override  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName()+**"开始卖票"**);  **while** (*i* <=1000) {  *sellTicket*();  }  }   **public synchronized static void** sellTicket() {   **if**(*i*<=1000){  String info =Thread.*currentThread*().getName()+**"卖出了第"**+*i*+**"张票"**;  System.***out***.println(info);  *i*++;  **try** {  Thread.*sleep*(5000);*// 不会释放锁* } **catch** (InterruptedException e) {  e.printStackTrace();  }  }**else**{  System.***out***.println(**"售罄了"**);  }  } } |

**总结：关于同步方法**

1. 不要将run()定义为同步方法
2. 同步实例方法的同步监视器(锁)是this；同步静态方法的监视器是类名.class
3. 同步代码块的效率要高于同步方法
   * + 同步方法的锁是this，一旦锁住一个方法，就锁住了所有的同步方法；同步代码块只是锁住使用该同步监视器的代码块，而没锁住使用其他监视器的代码块
     + 同步方法是将线程锁在了方法的外部，而同步代码块锁将线程锁在了代码块的外部，但是却是方法的内部
4. **对于synchronized锁（同步代码块和同步方法），如果正常执行完毕，会释放锁。如果线程执行异常，JVM也会让线程自动释放锁。所以不用担心锁不会释放。**
5. **synchronized锁的缺点：**
   1. 如果获取锁的线程由于要等待IO或其他原因（如调用sleep方法）被阻塞了，但又没有释放锁，其他线程只能干巴巴地等待，此时会影响程序执行效率。甚至造成死锁;
   2. 只要获取了synchronized锁，不管是读操作还是写操作，都要上锁，都会独占。如果希望多个读操作可以同时运行，但是一个写操作运行，无法实现。

### 死锁

死锁产生的原因:1多个线程共享多个资源 2多个线程都需要其他线程的资源,每个线程又不愿或者无法放弃自己的资源(锁的开关无法人为控制)

#### 【示例12】演示死锁

|  |
| --- |
| **package** com.bjsxt.threadDemo6;  **public class** Test2 {  **public static void** main(String[] args) {  **new** Thread(**new** XiaobaiRunn()).start();  **new** Thread(**new** XiaomingRunn()).start();  } }  **class** XiaomingRunn **implements** Runnable{  @Override  **public void** run() {  **synchronized** (**"遥控器"**){  System.***out***.println(**"小明抢到了遥控器,正在准备抢电池"**);  **synchronized** (**"电池"**){  System.***out***.println(**"小明抢到了电池,打开空调爽歪歪"**);  }  }  } } **class** XiaobaiRunn **extends** Thread{  @Override  **public void** run() {  **synchronized** (**"电池"**){  System.***out***.println(**"小白抢到了电池,正在准备抢遥控器"**);  **synchronized** (**"遥控器"**){  System.***out***.println(**"小白抢到了遥控器,打开空调爽歪歪"**);  }  }  } } |

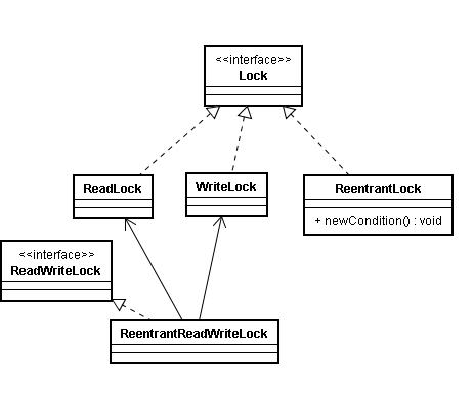
### 本节作业

1. 线程同步的四种方式是什么
2. 同步代码块的执行过程
3. 使用线程同步的synchronized两种方式完成多个用户对同一个账户取款

## 第四节 线程同步

### 4.1 Lock锁

基于synchronized锁的一些缺点，JDK1.5中推出了新一代的线程同步方式：Lock锁。更强大、更灵活、效率也更高。其核心API如图所示。



#### 【示例12】使用Lock锁实现线程同步

|  |
| --- |
| **package** com.bjsxt.lockDemo;  **import** java.util.concurrent.locks.Lock; **import** java.util.concurrent.locks.ReentrantLock;  **public class** Test1{  **public static void** main(String[] args) {  TicketRunnable tr=**new** TicketRunnable();  Thread t1=**new** Thread(tr,**"窗口1"**);  Thread t2=**new** Thread(tr,**"窗口2"**);  t1.start();  t2.start();   }  }  **class** TicketRunnable **implements** Runnable{  **static int** *i* =1;  */\*  \* 公平锁 非公平锁  \* 公平锁:当多个线程共同尝试上锁的时候,lock会倾向于让等待时间最长的线程优先获得  \* 乐观锁 乐观锁认为线程安全问题很少出现,即是上锁,也锁不住  \* 悲观锁 悲观锁任务线程安全问题随时放生,上了锁就能锁住  \* 目前所学的所有的锁都是悲观锁  \* \*/* Lock **lock**=**new** ReentrantLock(**true**);  @Override  **public void** run() {  **while** (*i* <=1000) {  **lock**.lock();*//上锁* **try** {  **if**(*i*<=1000){  String info =Thread.*currentThread*().getName()+**"卖出了第"**+*i*+**"张票"**;  System.***out***.println(info);  *i*++;  }**else**{  System.***out***.println(**"售罄"**);  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **lock**.unlock();*// 开锁* }   }  }   } |

**Lock接口**

下面介绍Lock锁的API。首先要说明的就是Lock接口，通过查看Lock的源码可知，Lock接口有6个方法。 下面来逐个讲述Lock接口中每个方法的使用，lock()、tryLock()、tryLock(long time, TimeUnit unit)和lockInterruptibly()是用来获取锁的。unLock()方法是用来释放锁的。newCondition()在后面的线程通信中使用。

|  |
| --- |
| **public interface** Lock {  **void** lock();  **void** lockInterruptibly() **throws** InterruptedException;  **boolean** tryLock();  **boolean** tryLock(**long** time, TimeUnit unit)  **throws** In terruptedException;  **void** unlock();  Condition newCondition(); } |

在Lock中声明了四个方法来获取锁，那么这四个方法有何区别呢？

* **lock()**

首先lock()方法是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。

如果采用Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。因此一般来说，使用Lock必须在try{}catch{}块中进行，并且将释放锁的操作放在finally块中进行，以保证锁一定被被释放，防止死锁的发生。通常使用Lock来进行同步的话，是以下面这种形式去使用的：

|  |
| --- |
| Lock l = ...;  l.lock();  try {  // access the resource protected by this lock  } finally {  l.unlock();  } |

* **tryLock()**

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false，也就说这个方法无论如何都会立即返回。拿不到锁时不会一直在那等待。

* **tryLock(long time, TimeUnit unit)**

tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回true。

* **lockInterruptibly()**

lockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。

**ReentrantLock类**

ReentrantLock，意思是“可重入锁”。ReentrantLock是唯一实现了Lock接口的非内部类，并且ReentrantLock提供了更多的方法。

ReentrantLock锁在同一个时间点只能被一个线程锁持有；

而可重入的意思是，ReentrantLock锁，可以被单个线程多次获取。

ReentrantLock分为“公平锁”和“非公平锁”。它们的区别体现在获取锁的机制上是否公平。“锁”是为了保护竞争资源，防止多个线程同时操作线程而出错，ReentrantLock在同一个时间点只能被一个线程获取(当某线程获取到“锁”时，其它线程就必须等待)；ReentraantLock是通过一个FIFO的等待队列来管理获取该锁所有线程的。在“公平锁”的机制下，线程依次排队获取锁；而“非公平锁”在锁是可获取状态时，不管自己是不是在队列的开头都会获取锁。

### 4.2 ReadWriteLock锁

ReadWriteLock也是一个接口，在它里面只定义了两个方法：

|  |
| --- |
| **public interface** ReadWriteLock {   Lock readLock();   Lock writeLock();  } |

一个用来获取读锁，一个用来获取写锁。也就是说将文件的读写操作分开，分成2个锁来分配给线程，从而使得多个线程可以同时进行读操作。

ReadWriteLock是一个接口，ReentrantReadWriteLock是它的实现类，该类中包括两个内部类ReadLock和WriteLock，这两个内部类实现了Lock接口。

ReentrantReadWriteLock里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。

#### 【示例13】认识ReadWriteLock锁

|  |
| --- |
| **public class** TestLock {  **public static void** main(String[] args) {  //默认也是非公平锁 也是可重入锁  ReadWriteLock rwl = **new** ReentrantReadWriteLock();  //多次返回的都是同一把读锁 同一把写锁  Lock readLock = rwl.readLock();  Lock readLock2 = rwl.readLock();  Lock writeLock = rwl.writeLock();  readLock.lock();  readLock.unlock();  System.**out**.println(readLock==readLock2);  } } |

注意：从结果中看到，从一个ReadWriteLock中多次获取的ReadLock、WriteLock是同一把读锁，同一把写锁。

#### 【示例14】ReadWriteLock示例

|  |
| --- |
| **package** com.bjsxt.threadDemo8;  **import** java.util.concurrent.locks.Lock; **import** java.util.concurrent.locks.ReadWriteLock; **import** java.util.concurrent.locks.ReentrantReadWriteLock;  **public class** Test1 {  **public static void** main(String[] args) {    **for** (**int** i = 1; i <= 5; i++) {  **new** Thread(**new** Operator()::read,**"读取线程"**+i).start();  }  **for** (**int** i = 1; i <= 5; i++) {  **new** Thread(**new** Operator()::write,**"写入线程"**+i).start();  }  } } **class** Operator {  **static** ReadWriteLock *rwl*=**new** ReentrantReadWriteLock();  **public void** read(){  **for** (**int** i = 0; i < 100; i++) {  *rwl*.readLock().lock();  **try** {  System.***out***.println(Thread.*currentThread*().getName()+**"开始读取信息"**);  **try** {  Thread.*sleep*(5);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName()+**"信息读取结束"**);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  *rwl*.readLock().unlock();  }  }    }  **public void** write(){  **for** (**int** i = 0; i < 100; i++) {  *rwl*.writeLock().lock();  **try** {  System.***out***.println(Thread.*currentThread*().getName()+**"开始修改信息"**);  **try** {  Thread.*sleep*(5);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName()+**"信息修改结束"**);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  *rwl*.writeLock().unlock();  }  }  } } |

### 2.3 Lock锁和同步锁（synchronized）的选择

|  |  |  |
| --- | --- | --- |
| **区别** | 同步锁synchronized | Lock锁 |
| 可重入性 | 可重入锁 | 可重入锁 |
| 锁级别 | 是一个关键字，JVM级别的锁 | 是一个接口，JDK级别的锁 |
| 锁方案 | 取决于JVM底层的实现，不灵活 | 可以提供多种锁方案供选择，更灵活 |
| 异常处理 | 发生异常会自动释放锁 | 发生异常不会自动释放锁，需要在finally中释放锁 |
| 隐式/显式 | 隐式锁 | 显式锁 |
| 独占/共享 | 独占锁 | ReentrantLock、WriteLock 独占锁  ReadLock 共享锁 |
| 响应中断 | 不可响应中断，没有得到锁的线程会一直等待下去，直到获取到锁 | 等待的线程可以响应中断，提前结束等待 |
| 上锁内容 | 可以锁方法、可以锁代码块 | 只可以锁代码块 |
| 获取锁状态 | 不知道是否获取锁 | 可以知道是否获取锁（tryLock的返回值） |
| 性能高低 | 重量级锁，性能低(难道坐以待毙吗？改一下虚拟机底层如何？？) | 轻量级锁，性能高，尤其是竞争资源非常激烈时 |

|  |
| --- |
| **扩展：同步锁的底层优化：锁升级**   * 在 Java 语言中，使用Synchronized 是能够实现线程同步的，在操作同步资源的时候直接先加锁。加锁可以使一段代码在同一时间只有一个线程可以访问，在增加安全性的同时，牺牲掉的是程序的执行性能，被称为重量级锁。 * 为了在一定程度上减少获得锁和释放锁带来的性能消耗，在JDK6 之后引入了“偏向锁”和“轻量级锁”，所以总共有4种锁状态，级别由低到高依次为：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态。这几个状态会随着竞争情况逐渐升级。 * 锁升级的过程      * 在markword中标记锁的类型   普通对象在内存中的结构分为多部分，第一部分称为markword，共64位。在对应锁对象的markword字段的低位字段标记锁的类型。 |

### 本节作业

1. 使用Lock锁实现多个线程实现安全的银行取款操作
2. Lock锁和同步锁的异同之处
3. 实现三种线程同步方式实现多个窗口卖票，保证售票的安全性