# 一、线程

## 1、进程

1、什么是进程？

* 正在执行的程序称作一个进程。进程负责规划内存空间的划分。

2、windows号称是多任务的操作系统，那么windows是同时运行多个应用程序吗？

* 从宏观角度讲：windows确实是在同时运行多个应用程序。
* 从微观角度讲：CPU是做了一个快速切换的动作，由于速度太快，所以我们感觉不到在切换而已。

3、注意：

* 单核的CPU在一个时间片中只能执行一个应用程序。各个应用程序其实是在做CPU的的资源争夺战而已，CPU做了飞快的切换动作。
* 与其说是 进程 在做CPU的资源争夺战，不如说是线程在做CPU的资源争夺战。

## 2、线程

1、什么是线程？

* 线程在一个进程中负责代码的执行，就是进程中一个执行代码的路径。（即程序中的一个功能）

2、什么是多线程？

* 在一个进程中有多个线程同时在执行不同的任务。

3、疑问？线程负责了代码的执行，我们之前没有学习过线程，那么代码为什么可以执行呢？

* 运行任何一个java程序，Java虚拟机都会创建一个主main线程来执行main方法中的所有代码。

4、一个Java应用程序，至少有几个线程？

* 至少有两个线程，一个main主线程负责代码的执行，一个垃圾回收器线程，负责垃圾回收。

5、多线程的好处？

* 解决了一个进程可以同时执行多个任务的问题。
* 提高了资源的利用率。

6、多线程的弊端：

* 增加cup的负担
* 降低了一个进程中线程的执行概率
* 引发了线程安全问题
* 出现死锁现象

# 二、多线程

## 1、如何创建线程？

## （1）方式一：继承Thread类

1、第一步：自定义一个类继承Thread类

2、第二步：重写Thread类的run() 方法，把自定义线程的任务代码写在run()方法中。

3、第三步：创建Thread的子类对象，并且**调用start()方法**启动线程。

**1）疑问：**重写run方法的目的是什么？

每个线程都有自己的任务代码，Java虚拟机创建的主线程的任务代码就是main方法中的所有代码

自定义线程的任务代码就写在run()方法中，自定义线程负责了run()方法中的代码。

2）注意：

一个线程一旦开启（调用了start()方法），那么线程就会执行run()方法中的代码。

run方法千万不能直接调用，直接调用run()方法，就相当于调用了一个普通的run方法而已，并没有开启新的线程。

### （2）方式二：实现Runnable接口

1、第一步：实现Runnable接口

2、第二步：重写run() 方法

3、第三步：创建 Runnable实现类的对象

4、第四步：创建Thread类的对象，并接受Runnable实现类的对象

5、第五步：使用Thread线程对象调用Thread类的start()方法，启动一个线程

|  |
| --- |
| *// 1、实现 Runnable 接口，并实现其中的方法* **class** CreatRunnable **implements** Runnable {  *// 2、重写 run()方法* @Override  **public void** run() {  **for** (**int** i = 0; i < 5; i++) {  System.***out***.println(Thread.*currentThread*().getName() + i);  }  } }  **public class** Dome3 {  **public static void** main(String[] args) {  *// 3、创建 Runnable 接口的实现类对象* CreatRunnable creatRunnable = **new** CreatRunnable();  *// 4、创建 Thread 类的对象，并接受 RUnnable的实现类对象* Thread thread = **new** Thread(creatRunnable, **"creatRunnable线程"**);  *// 5、调用 Thread 的 start()方法启动线程* thread.start();  } } |

**疑问：**

**问题1 ：请问Runnable实现类的对象是线程对象吗？**

* 不是线程对象，只是一个实现了Runnable接口的实现类对象而已。
* 只有Thrad或者Thread类的子类才是线程对象

**问题2：为什么要把Runnable实现类对象作为实参传递给Thread对象呢？作用是什么？**

* 解析Thread类的源码

|  |
| --- |
| // Thread thread = new Thread（createRunnable，”张三”）;  // 使用了target变量接受Runnable实现类的对象  **public** Thread(Runnable target, String name) {  init(**null**, target, name, 0); }  // thread.start(); 调用start()方法启动一个线程。  // 线程启动后，执行run()方法中的任务代码，首先判断是否传入了target变量。  // 如果传入，调用target对象的run方法作为线程任务执行。实际上执行的是Runnable实现类对象的run方法。  @Override **public void** run() {  **if** (**target** != **null**) {  **target**.run();  } } |

* 通过对Thread类的源码进行解析可知，将Runnable实现类对象作为实参传入Thread对象的作用：

就是将Runnable实现类的对象的run()方法作为线程的任务代码去执行。

**推荐使用：第二种 Runnable**

**原因：因为java是单继承、多实现的。**

## 2、线程的生命周期

（1）新建期

* + new 一个线程（创建一个线程对象）

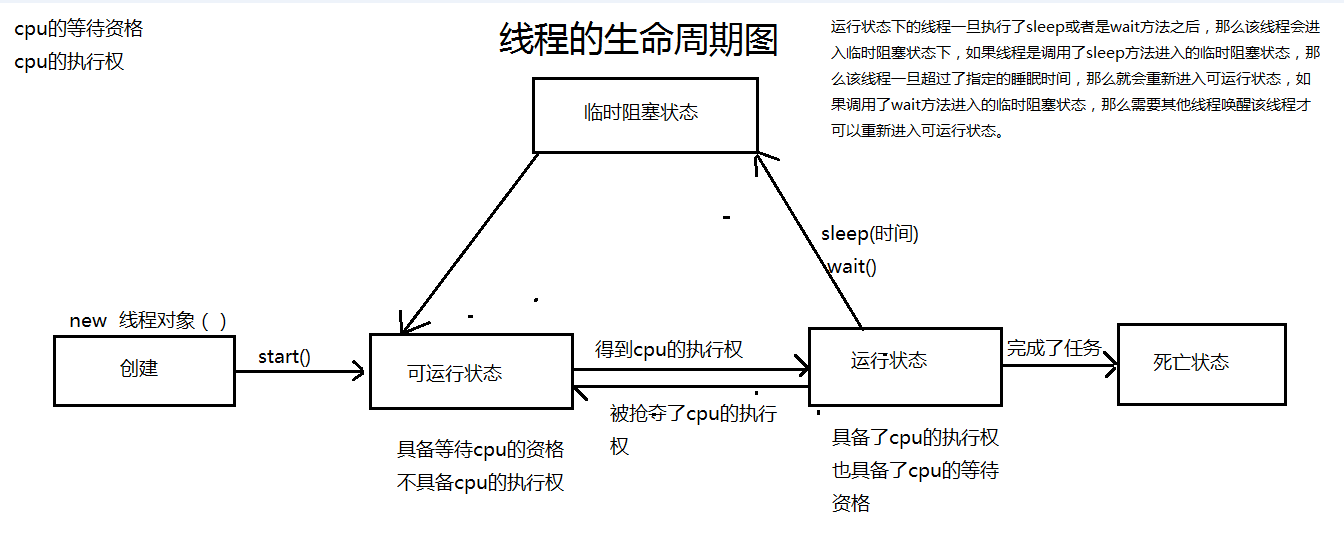
（2）等待执行期

* + 调用了 start()方法，此时具备了等待

（3）执行期

（4）临时阻塞期

（5）死亡期



## 3、线程安全问题

### （1）根本原因（线程安全问题）：

* + 同时存在两个或两个以上的线程对象共享一个资源
  + 多线程操作共享资源的代码有多句

### （2）解决方案（线程安全）

#### 1）同步代码块（方式一）

|  |
| --- |
| **Synchronized(锁对象){**  **需要被同步的代码块**  **}** |

##### 同步代码块 的注意事项

* + 所对象可以是任意一个对象
  + 一个线程在同步代码块中 sleep()了并不会释放锁对象
  + 如果不存在着线程安全问题，那么千万不要使用同步代码块，因为会降低效率
  + 锁对象必须是多线程共享的一个资源，否则锁不住。

#### 2）同步函数（方式二）

* + 同步函数：就是使用 synchronized修饰一个函数

##### 同步函数的注意事项

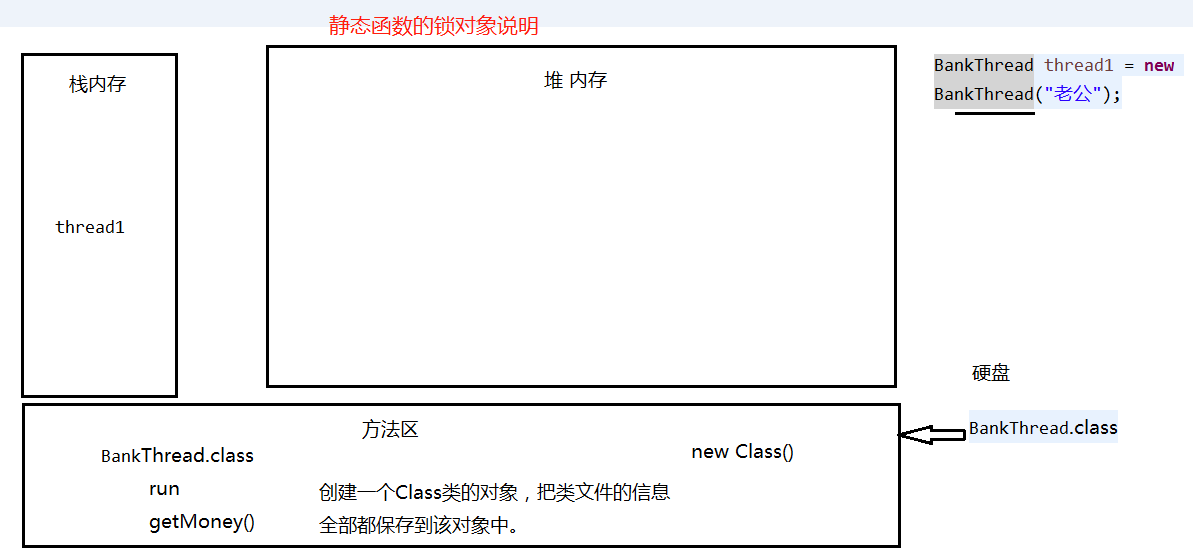
1、如果是一个非静态的同步函数，它的锁对象是this对象；如果是一个静态的同步函数，它的锁对象是当前函数所在类的字节码文件（Class对象）

2、同步函数的琐对象是固定的，不是由你来指定的。

### 3）推荐：使用同步代码块

原因：

1. 同步代码块的琐可以由我们来指定，方便控制；同步函数的琐是固定的 ，不能由我们来控制。
2. 同步代码块可以很方便的控制需要被同步代码的范围；同步函数必须是整个函数的代码都被同步。



## 4、死锁

* java中同步机制解决了线程安全问题，但是也同时引发了死锁问题。

### 1、根本原因(死锁现象出现)

* 1. 存在两个或两个以上的线程对象
  2. 存在两个或两个以上的共享资源

### 2、解决方案(死锁现象)

* 没有解决方案，只能尽量去避免死锁现象的发生。

3、案例

|  |
| --- |
| **package** thread2;  **public class** Dome2 {  **public static void** main(String[] args) {  DeadLock zhangsan = **new** DeadLock(**"张三"**);  DeadLock ergouzi = **new** DeadLock(**"二狗子"**);  zhangsan.start();  ergouzi.start();  } }  **class** DeadLock **extends** Thread {  **public** DeadLock(String name) {  **super**(name);  }   @Override  **public void** run() {  **if** (**"张三"**.equals(Thread.*currentThread*().getName())) {  **synchronized** (**"遥控器"**) {  System.***out***.println(**"张三拿到了遥控器，准备去拿电池..."**);  **synchronized** (**"电池"**) {  System.***out***.println(**"张三拿到了电池，组装成遥控器，爽歪歪的吹着空调..."**);  }  }  } **else if** (**"二狗子"**.equals(Thread.*currentThread*().getName())) {  **synchronized** (**"电池"**) {  System.***out***.println(**"二狗子拿到了电池，准备去拿遥控器..."**);  **synchronized** (**"遥控器"**) {  System.***out***.println(**"二狗子拿到了遥控器，组装成遥控器，爽歪歪的吹着空调..."**);  }  }  }  } } |

**运行结果：**

二狗子拿到了电池，准备去拿遥控器...

张三拿到了遥控器，准备去拿电池...

该结果出现了死锁现象。

运行结果：

张三拿到了遥控器，准备去拿电池...

张三拿到了电池，组装成遥控器，爽歪歪的吹着空调...

二狗子拿到了电池，准备去拿遥控器...

二狗子拿到了遥控器，组装成遥控器，爽歪歪的吹着空调...

该结果并未出现死锁现象。

## 5、线程通讯

* 一个线程完成了自己的任务时，要通知另一个线程去完成另一个任务。

### （1）线程的常用方法

* wait(): 等待 如果线程执行了wait方法，那么该线程会进入等待的状态，等待状态下的线程必须要被其他线程调用notify方法才能唤醒。
* notify()： 唤醒 唤醒线程池等待线程其中的一个。
* notifyAll() : 唤醒线程池所有等待 线程。

### （2）wait与notify方法要注意的事项：

1. wait方法与notify方法是属于Object对象 的。

2. wait方法与notify方法必须要在同步代码块或者是同步函数中才能 使用。

3. wait方法与notify方法必需要由锁对象调用。

### （3）生产者与消费者案例集

|  |
| --- |
| // 测试类  **package** thread2; **public class** Dome5 {  **public static void** main(String[] args) {  *// 产品对象* Product product = **new** Product();  *// 生产者线程* Thread producer = **new** Thread(**new** Producer(product), **"生产者"**);   *// 消费者线程* Thread cunsumer = **new** Thread(**new** Cunsumer(product), **"消费者"**);   producer.start();*// 启动生产者线程*  cunsumer.start();*// 启动消费者线程*  }  } |

|  |
| --- |
| *// 产品类* **class** Product {  **private** String **name**;*//产品名* **private double price**;*//产品价格* **public boolean** isFlag() {  **return flag**;  }   **public void** setFlag(**boolean** flag) {  **this**.**flag** = flag;  }   **private boolean flag** = **false**;*//生产者是否生产了产品的标识，默认没有* **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public double** getPrice() {  **return price**;  }   **public void** setPrice(**double** price) {  **this**.**price** = price;  } } |

|  |
| --- |
| *// 消费者* **class** Cunsumer **implements** Runnable {  Product **p**;   **public** Cunsumer(Product p) {  **this**.**p** = p;  }   @Override  **public void** run() {  **while** (**true**) {  **synchronized** (**p**) {  **if** (**p**.isFlag() == **true**) {*// 产品已经生产完毕* System.***out***.println(**"消费者消费了"** + **p**.getName() + **" 价格："** + **p**.getPrice());  **p**.setFlag(**false**);*// 消费完后标识改为false* **p**.notifyAll();*//唤醒生产者线程* } **else** {  **try** {  *// 如果没有产品，则消费者等待* **p**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  } } |

|  |
| --- |
| *// 生产者类* **class** Producer **implements** Runnable {  Product **p**;    **public** Producer(Product p) {  **this**.**p** = p;  }   @Override  **public void** run() {  **int** count = 0;*//生产数量计数器* **while** (**true**) {  **synchronized** (**p**) {  *// 没有生产，进行生产* **if** (**p**.isFlag() == **false**) {  **if** (count % 2 == 0) {  **p**.setName(**"苹果"**);  **p**.setPrice(6.5);  } **else** {  **p**.setName(**"香蕉"**);  **p**.setPrice(2.0);  }  System.***out***.println(**"生产者生产出了："** + **p**.getName() + **" 价格是："** + **p**.getPrice());  count++;  **p**.setFlag(**true**);*// 生产后标识符修改为true* **p**.notifyAll();*// 生产后唤醒消费者线程* } **else** {  **try** {  *// 如果存在产品，则生产者等待* **p**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  } } |

## 6、wait方法与notify方法

### （1）方法详解

1）wait()方法：等待线程。

* 如果一个线程调用了wait()方法，那么该线程就会进入一个以**锁对象为标识符的线程池**中进行等待，直到其他线程调用了notify()方法才能被唤醒。

2）notify()方法：唤醒线程。

* 如果一个线程调用了notify()方法，那么就会唤醒以**锁对象为标识符的线程**中，等待状态线程（wait()）中的其中一个（随机唤醒，不能指定唤醒某个线程）。

3）notifyall()方法：唤醒全部线程。

* 如果一个线程调用了notifyall()方法，那么就会唤醒以 锁对象为标识符的全部线程。



### （2）notify()与wait()方法的注意事项：

1、wait()与notify()是属于Object类的。

2、wait()与notify()必须在同步代码块或同步函数中才能使用。（因为这两个方法必须由锁对象调用，否则出错）

3、wait()与notify()必须由锁对象调用。

## 7、线程的停止

### 1、停止线程的两种方法

* 停止一个线程，我们一般都会使用一个变量去控制的
* 如果需要停止一个出于等待状态的线程，那么我们需要通过变量来配合notify()，或者使用interrupt()方法来使用。

### 2、通过变量停止一个等待状态的线程

|  |
| --- |
| package thread2; **〈通过变量来停止一个等待状态的线程：当main线程中的i==80时，停止Thread1线程〉** public class Dome7 {  public static void main(String[] args) {  Thread1 thread1 = new Thread1();//Runnable实现类对象  Thread d = new Thread(thread1, "狗娃");//线程对象  d.setPriority(10);//线程的优先级 1-10  d.start();   for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + ":" + i);  if (i == 80) {  thread1.setFlag(false);  synchronized (thread1) {  thread1.notify();//唤醒线程  }  }  }  } }  class Thread1 implements Runnable {  private boolean flag = true;// 默认是true   public boolean isFlag() {  return flag;  }   public void setFlag(boolean flag) {  this.flag = flag;  }   @Override  public synchronized void run() {// 同步函数  int i = 0;  while (flag) { // 默认是true  System.out.println(Thread.currentThread().getName() + ":" + i);  i++;  if (i > 50) {  try {  this.wait();//将线程进行等待  } catch (InterruptedException e) {  System.out.println("接收到了异常信息。。。：");  }  }  }  } } |

### 3、通过变量来停止一个线程

|  |
| --- |
| **package** thread2;   *〈通过变量来停止一个线程〉*  **public class** Dome8 {   **public static void** main(String[] args) {  Thread2 thread2 = **new** Thread2();  Thread dog = **new** Thread(thread2, **"狗娃"**);  dog.setPriority(10);*//优先级* dog.start();   **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **":"** + i);  **if** (i == 80) {  thread2.setFlag(**false**);  }  }   } }  **class** Thread2 **implements** Runnable {  **private boolean flag** = **true**;   **public boolean** isFlag() {  **return flag**;  }   **public void** setFlag(**boolean** flag) {  **this**.**flag** = flag;  }   @Override  **public void** run() {  **int** i = 0;  **while** (**flag**) {  System.***out***.println(Thread.*currentThread*().getName() + **":"** + i);  i++;  }  } } |

## 8、守护线程

* 守护线程（后台线程）：在一个进程中，如果只剩下了守护线程，那么守护线程也会死亡。

### 1、守护线程的方法(都属于Thread类)

* boolean isDaemon(); 判断该线程是否是守护线程
* Thread setDaemon(Boolean flag); 设置该线程为守护线程

### 2、守护线程例子

|  |
| --- |
| **package** thread2; **public class** Dome9 {  **public static void** main(String[] args) {  Thread thread = **new** Thread(**new** DaemonThread(), **"后台线程"**);  thread.setDaemon(**true**);*//设置线程为守护线程* System.***out***.println(**"后台线程是守护线程吗？"** + thread.isDaemon());  thread.start();  **for** (**int** i = 0; i <= 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **":"** + i);  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } }  **class** DaemonThread **implements** Runnable {  **public void** run() {  **for** (**int** i = 0; i <= 100; i++) {  **if** (i == 0) {  System.***out***.println(**"开始下载QQ更新包..."**);  } **else** {  System.***out***.println(**"当前已下载"** + i + **"%"**);  }  **if** (i == 100) {  System.***out***.println(**"更新包下载完成，等待安装..."**);  }  **try** {  Thread.*sleep*(200);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } } |

## 9、join方法

* 一个线程如果执行了join语句，那么就会有一个新的线程加入到当前线程中。当前线程必须让步给新线程，直到新线程执行完任务，原线程才能继续执行剩余的任务。
* **注意：调用join()方法的线程必须是启动状态的（即必须先start()开启线程，再join加入线程）**

|  |
| --- |
| **package** thread2;  *join方法：一个线程如果执行了join语句，那么就会有一个新的线程加入当前线程，当前线程必须让步给新加入的线程，等新线程完成任务后才会继续执行原线程的任务 〉  注意：新加入的线程必须是启动状态的，即必须先start()开启线程，然后才能join加入线程* **public class** Dome10 {  **public static void** main(String[] args) {  Thread motherThread = **new** Thread(**new** Monther(), **"老妈线程"**);  motherThread.start();  }  } **class** Monther **implements** Runnable {  @Override  **public void** run() {  System.***out***.println(**"准备炒菜，发现没有酱油了..."**);  *// 等待儿子去买酱油* Thread sonThread = **new** Thread(**new** Son(), **"儿子线程"**);  **try** {  *// 如果一个线程(mother)执行了join方法，那么就会有一个新的线程(son)加入，原线程(mother)必须给新线程(son)让路，等待新线程(son)执行完任务后，原线程(mother)才会继续执行未完成的任务  // 注意：加入线程前，必须先启动线程，不能颠倒顺序* sonThread.start();  sonThread.join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"继续炒菜"**);  } } **class** Son **implements** Runnable {  @Override  **public void** run() {  System.***out***.println(**"儿子买回酱油，将酱油给老妈..."**);  } } |