1. 线程初见
2. 概述

进程是受操作系统管理的基本运行单元，线程是 进程中独立运行的子任务。使用多线程技术，可以在同一时间内运行更多不同种类的任务。一个进程运行时至少会有一个线程在运行，如main方法的线程至少有一个main线程在运行。

1. 实现多线程的方式

实现多线程有两种方式，一种是继承Thread类，一种是实现Runnable接口。使用继承Thread类的方式创建新线程时，最大的局限就是不支持多继承，因此，Runnable解决了这个问题，可以一边实现该接口同时继承其他的类。

1. 继承Thread类

创建一个自定义的线程类，继承Thread类，重写run方法，该方法中填写的是线程要执行的任务。

package thread;  
  
public class MyThread extends Thread {  
  
 @Override  
 public void run() {  
 super.run();  
 System.*out*.println("MyThread");  
 }  
}

再创建一个测试类，启动自定义线程，可以看到线程中的run方法的操作虽然先被调用，但是仍然在后面输出，这就是线程的随机调用。

package thread;  
  
public class Run {  
 public static void main(String[] args) {  
 MyThread myThread = new MyThread();  
 myThread.start();  
 System.*out*.println("结束！");  
 }  
}

start()方法通知“线程规划器”，这个线程准备好了，可以调用该线程对象中的run方法了，之后，系统可以在某一个时间点来调用run方法，线程会开始运行，启动，这就让多线程有了异步执行的效果。如果直接调用run方法的话，就只是同步了，该线程对象不会交给“线程规划器”来处理，而是由main主线程调用run方法。

1. 实现Runnable接口

创建一个实现了Runnable接口的类MyRunnable，同样是重写run方法。

package thread;  
  
public class MyRunnable implements Runnable {  
 @Override  
 public void run() {  
 System.*out*.println("ing");  
 }  
}

创建一个测试类，测试能够传入Runnable接口的Thread构造器：

package thread;  
  
public class Run {  
 public static void main(String[] args) {  
 Runnable myRunnable = new MyRunnable();  
 Thread myThread = new Thread(myRunnable);  
 myThread.start();  
 System.*out*.println("结束！");  
 }  
}

1. 多线程
2. 问题概述

当多个线程要同时访问一个变量时，一定会出现变量的状态异常，即非线程安全问题。要使得多个线程之间进行同步，依然按照顺序进行处理，就需要将该变量上锁，当该变量被使用时，就将该变量锁上，其他变量发现该变量已上锁，就进行等待。只有该变量的锁被取消时，其他的线程才能够访问该变量。

1. synchronized关键字

非线程安全的问题可以用synchronized关键字解决，在需要上锁的变量，方法上添加这个关键字，多线程的访问就不会出现非线程安全问题，加锁的这段代码称为“互斥区”。

1. 线程方法介绍
2. currentThread()

该方法返回代码段正在被哪个线程调用，注意，这个线程并不是该代码段所在的线程，而是它被调用的线程。

1. isAlive()

判断当前的线程是否处于活动状态，所谓的活动状态指的是线程已经启动但是还没有终止，只要正在运行或者准备开始运行都是存活的。

1. sleep()

在指定的毫秒数内让当前正在执行的线程暂停执行，注意！！！这个线程指的是this.currentThread()返回的线程，即正在进行调用操作的线程。而不是我们想当然的sleep所处的代码段的线程！！！

1. getId()

该方法用于取得线程的唯一标识。

1. 停止线程
2. 问题概述

停止一个线程意味着停止正在做的操作，这可能会出现很大的风险，如数据的丢失，必须要做好防范措施。

1. interrupt()

通常使用的是Thread.interrupt()方法，该方法在当前线程中打了一个停止的标记，但是并没有真正地停止线程，还需要加上一个判断才能停止线程。

1. this.interrupted()

该方法可以得知运行该方法的线程是否中断，太拗口了！！！即一个线程运行了该方法，该方法可以测试出该线程是否中断。并不是说哪个线程调用了该方法，就是测试这个线程是否中断。如下所示，即是我们新建了一个线程并且中断该线程，用该线程调用interrupted()方法，该方法依然返回的是false，因为该方法是由main线程运行的，main线程并没有中断。

package thread;  
  
public class Run2 {  
 public static void main(String[] args) {  
 try {  
 MyThread2 myThread2 = new MyThread2();  
 myThread2.start();  
 Thread.*sleep*(1000);  
 myThread2.interrupt();  
 System.*out*.println(myThread2.*interrupted*());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("end");  
 }  
}

总结，interrupted()方法测试运行该方法的线程是否中断，可以理解为该方法在哪个线程中运行就是判断该线程的状态；并且执行后可以将状态清除为false，即如果执行两次该方法，如果第一次为true，那么第一次执行后，该状态就被清楚为了false，第二次执行就返回false了。

1. this.isInterrupted()

该方法不是static方法，因此需要使用一个线程对象调用它，因此该方法返回的就是调用的线程对象是否中断，并且不会清楚状态标志。

1. 停止线程

在run()方法中通过判断当前的线程是否中断，来结束当前线程的执行。

package thread;  
  
public class MyThread3 extends Thread {  
  
 @Override  
 public void run() {  
 super.run();  
 for(int i = 0; i < 500000; ++i){  
 if(this.*interrupted*()){  
 System.*out*.println("停止！！！");  
 break;  
 }  
 System.*out*.println("i = " + i+1);  
 }  
 }  
}

注意，上面的示例中虽然停止了线程，但是如果for语句下面还有语句，还是会继续运行的，这个可以用抛出异常解决，将break语句替换为抛出异常，这样，代码就会进入catch中，不会继续执行for语句下面的语句了。

package thread;  
  
public class MyThread3 extends Thread {  
  
 @Override  
 public void run() {  
 super.run();  
 try {  
 for(int i = 0; i < 500000; ++i){  
 if(this.*interrupted*()){  
 System.*out*.println("停止！！！");  
 throw new InterruptedException();  
 }  
 System.*out*.println("i = " + i+1);  
 }  
 System.*out*.println("?????");  
 } catch (InterruptedException e) {  
 System.*out*.println("catch");  
 e.printStackTrace();  
 }  
 }  
}

1. 在sleep中停止线程

在sleep状态下停止某一线程，会进入catch语句中，并且清除线程的停止状态值，使之变为false。

1. 使用return停止线程

可以将break语句替换为return语句，同样可以达到停止线程的目的，但是这样的话代码中就会有很多的return语句，因此不建议这样的操作。最好的方式还是使用抛出异常的方式，在catch块中可以对异常的信息进行相关的处理。