Java中的线程及线程间通信

# Runnable接口

## Runnable接口

**@FunctionalInterface**

public interface Runnable

Runnable接口存在于**java.lang**包中，是一个功能接口(**Functional Interface**)。

## 功能

Runnable接口是一个功能接口，线程中执行的实例都必须实现该接口，且该实例必须实现无参数的run()方法。

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

Functional Interface:This is a functional interface and can therefore be used as **the assignment target for a lambda expression or method reference**.

This interface is designed **to provide a common protocol for objects** that wish to execute code while they are active. For example, Runnable is implemented by class **Thread**. Being active simply means that a thread has been started and has not yet been stopped.

该接口主要是规定了线程执行任务的标准。

In addition, **Runnable provides the means for a class to be active while not subclassing Thread**. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

要想创建一个新的线程，必须**直接或间接的实现**Runnable接口且重写其中的run方法，其中**run方法**是用来封装线程所要的任务。

## 方法

Runnable接口中只有一个方法：**run()**。

**void run()**

When an object implementing interface Runnable **is used to create a thread**, starting the thread causes the object's run method to be called in that separately executing thread. The general contract of the method run is that it may take any action whatsoever.

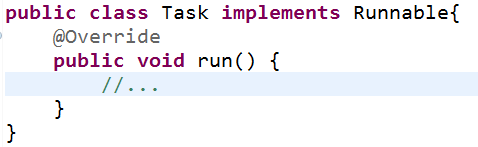
作用：run()方法用于**封装新线程要执行的任务**。

## 实现类

**Runnable接口**有许多直接实现类，其中最重要的是**Thread类**。

All Known Implementing Classes:AsyncBoxView.ChildState, ForkJoinWorkerThread, **FutureTask**, RenderableImageProducer, SwingWorker, **Thread**, **TimerTask**

**另外，**创建线程的常用方式，创建一个封装任务的Task类，即实现Runnable接口的一个类。



## 继承的子接口

All Known Subinterfaces: **RunnableFuture<V>**, **RunnableScheduledFuture<V>。**

# Thread类

## 继承关系：

public class Thread extends **Object** implements **Runnable**

Thread类：存在于**java.lang包**中，直接继承于Object，实现了java.lang.Runnable接口，并覆写了Runnable接口中的**run()方法**。

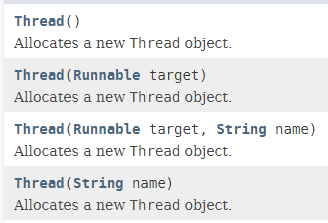
## 构造方法

一个Thread线程初始化参数：①线程名字String name;②**线程任务:Runnable task**；③线程所在组：ThreadGroup；④栈大小：long stackSize。

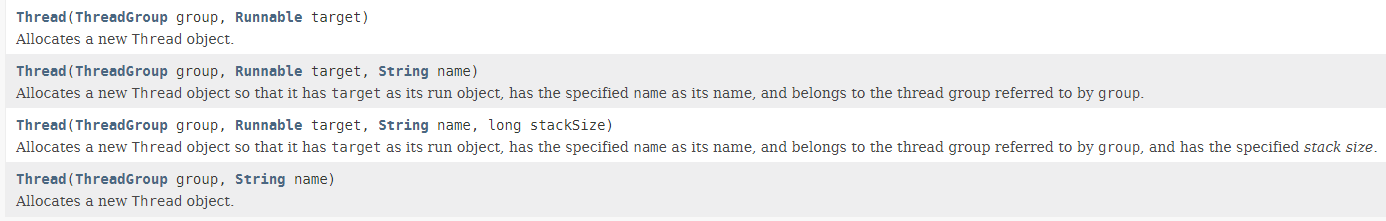
注意：线程是存在**栈**中的。

### 常用的

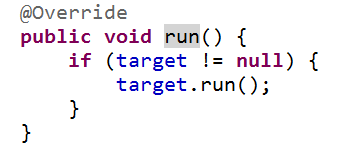
创建一个线程，Runnable task一般不可缺少；给线程命名有利于辨别该线程。



### 规定线程组ThreadGroup和栈大小



## Thread类中的run()方法

如果创建Thread对象时没有传入Runnable的实现类对象，则该Thread对象的run方法什么也不执行。所以，要创建Thread对象，一般需要传入Runnable的实现类对象。

结论：Thread类默认实现的run方法相当于是空的。可以通过继承Thread类或传入Runnable实现对象来覆盖Thread中默认的run()方法。

## Thread中线程的优先级Priority

**对线程优先级的理解**：对线程赋予不同的优先级，只是优先级大的线程执行到的**概率大**些，优先级小的线程执行到的**概率小**些。但是，优先级大的线程不一定比优先级小的线程先执行，优先级低的线程有可能先执行。**优先级并不能代表执行的顺序，只是一种概率大小的含义，优先级高的线程先执行的概率较大，但是低优先级的线程也有可能先执行**。

**Java虚拟机**中通过**线程调度器**来管理线程，调度器把线程分成10个等级；Thread中规定线程优先级范围为1—10；默认为5。把优先级为1、5、10封装成了字段。线程（包括主线程）被创建时，默认的优先级级别都是5。

static int **MAX\_PRIORITY 值为10**

The maximum priority that a thread can have.

static int **MIN\_PRIORITY 值为1**

The minimum priority that a thread can have.

static int **NORM\_PRIORITY 值为5**

The default priority that is assigned to a thread.

Thread中也定义了get和set方法，获取和设置线程的优先级：

**int getPriority()** Returns this thread's priority.

**void setPriority(int newPriority)** Changes the priority of this thread.

## Thread的静态方法

**一个类的静态方法**体现了该类的具有的共性，获取该类的共同性质。

Thread类的静态方法有：

### static Thread currentThread() 获取当前的线程对象。

Returns a reference to **the currently executing thread object**.

### static int activeCount() 获取当前线程组中active的线程个数，返回int值。

Returns an estimate of the number of active threads in the current thread's thread group and its subgroups.

### static boolean holdsLock(Object obj) 判断当前线程是否持有obj锁

Returns true **if and only if** the current thread holds the monitor lock on the specified object.

### static boolean interrupted() 判断当前线程是否已经中断。

Tests whether the current thread has been interrupted.

### static void yield()

A hint to the scheduler that the current thread is willing to yield its current use of a processor.

Thread.yield() 线程让步。

## Thread.yield()方法详解

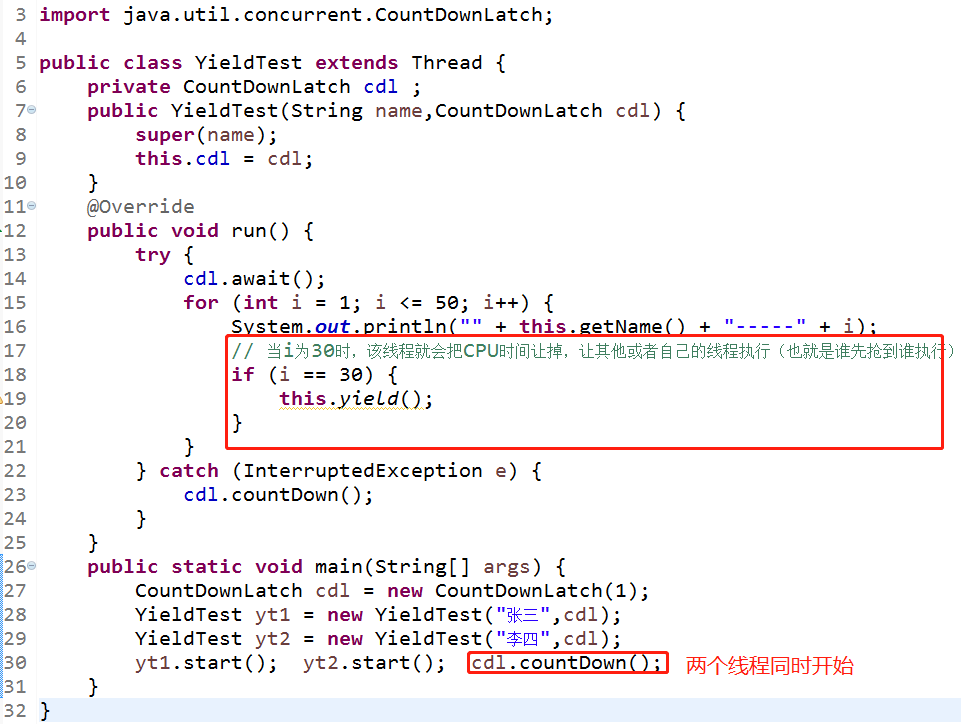
**yield**美[jild]vt.屈服，投降，让步; 不再反对;vi.放弃，屈服; 退让，退位;

**使当前线程从执行状态（运行状态）变为可执行态（就绪状态）**。cpu会从众多的可执行态里选择，也就是说，当前也就是刚刚的那个线程还是有可能会被再次执行到的，并不是说一定会执行其他线程而该线程在下一次中不会执行到了。

Java线程中有一个Thread.yield( )方法，很多人翻译成**线程让步**。顾名思义，就是说当一个线程使用了这个方法之后，它就会把自己CPU执行的时间让掉，让自己或者其它的线程运行。

打个比方：现在有很多人在排队上厕所，好不容易轮到这个人上厕所了，突然这个人说：“我要和大家来个竞赛，看谁先抢到厕所！”，然后所有的人在同一起跑线冲向厕所，有可能是别人抢到了，也有可能他自己有抢到了。我们还知道线程有个优先级的问题，那么手里有优先权的这些人就一定能抢到厕所的位置吗? 不一定的，他们只是概率上大些，也有可能没特权的抢到了。

示例：



运行结果：

第一种情况：李四（线程）当执行到30时会CPU时间让掉，这时张三（线程）抢到CPU时间并执行。

第二种情况：李四（线程）当执行到30时会CPU时间让掉，这时李四（线程）抢到CPU时间并执行。

## Thread.join()方法

**join方法:线程联合**。作用：当前运行的线程进入阻塞状态，等待**调用join方法的线程**运行完毕后，再继续执行，注意主线程只等调用join方法的线程运行完毕，调用join方法也会抛出异常。临时加入一个线程可以使用join方法。

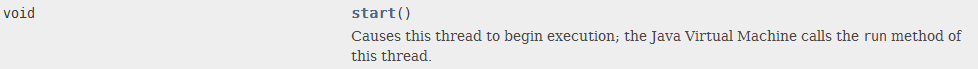
## 守护线程daemon

**守护线程**： **setDaemon(boolean b)**方法可以将此线程设置为守护线程或用户线程。This method must be invoked before the thread is started.

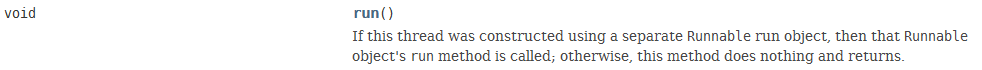
守护线程与一般线程运行、等待都一样，只不过结束的时候不一样，只要一般线程消失，守护线程会自动消失，无论其处于何种状态。当所有的线程都是守护线程时，此时程序会自动退出。

## 一般方法

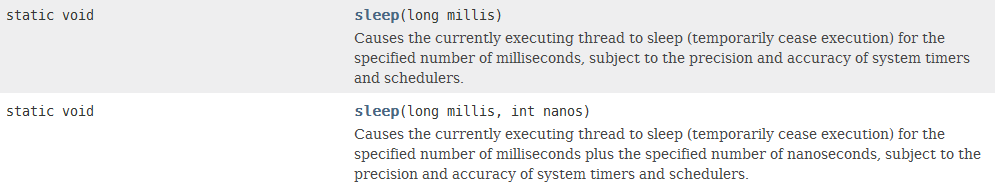
### start方法：启动此线程。



### run方法：封装线程执行的任务。



### sleep方法：



### checkAccess方法：



### getXxxx方法：

1. **getId方法**：获取线程的ID。



1. **getName()方法**：获取线程的名字。



1. **getPriority()方法**：获取线程的优先级。



### setXxx方法：

1. **setName()**



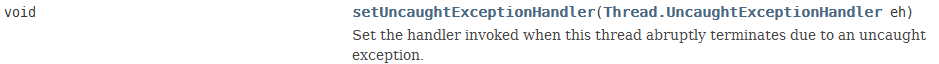
1. **setPriority()方法**: **设置优先级，数值只有1—10；默认的值是5.**

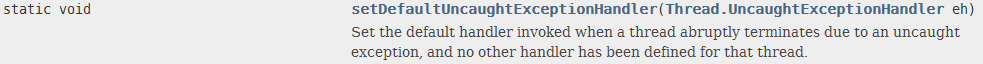


1. **setDaemon(Boolean on)**：**守护线程**。



1. 其他的setXxxx（）



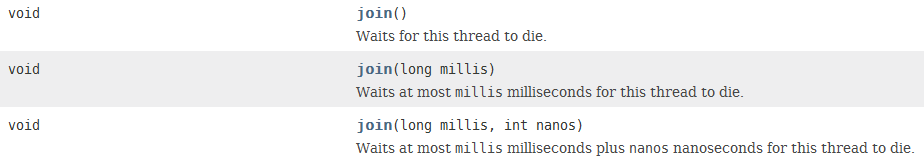




### interrupt（）：中断此线程。



### join（）方法：



### 判断方法：

1. interrupted：静态方法，判断此线程是否是被中断了。



1. isInterrupted：



1. isAlive：

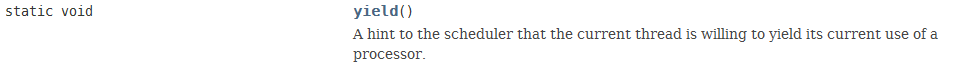


1. isDaemon():是否是后台运行。

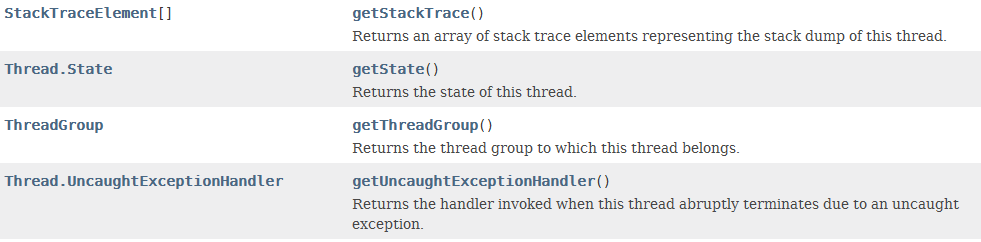
**daemon ['diːmən]n. 守护进程；后台程序**



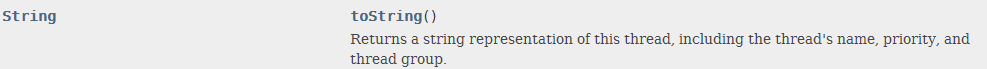
### yield：释放cpu执行资格，线程让步。



### getXxxx（）：



### toString:返回线程的表现形式，包括线程名、优先级和线程组。



# Java创建线程的三个方法

## 简单描述创建线程的方法：

### 继承Thread类+覆盖run方法

启动线程：创建子类对象+**对象.start()**方法；

### 实现Runnable接口构建Task类+实现run方法

启动线程：创建Thread对象，传入**Runnable接口**的实现类

### Callable接口

（3） 由于run方法返回类型为void且不能抛出异常，所以后来又引入了第三种方法，但是比较繁琐，使用较少，见最后介绍。

了解即可。

## 创建一个新的线程有三个方法：（详解）

### 创建一个继承Thread的子类，并重写Thread中的run方法：(很少使用该方式)

创建步骤：① 定义一个类并继承Thread类extends Thread，② **重写(**Override) Thread类中的**run方法**，将线程所要执行的代码写到run方法中；③在main方法中，创建此类的一个对象，即**创建线程成功**；④调用此**对象的start方法**，启动此线程。

### 实现接口Runnable，并实现run()方法： (一般使用这种方式)

静态代理设计模式：Thread类作为一个代理。

创建步骤：i. 定义一个类并实现接口Runnable；ii. 重写接口中的run方法，将线程的任务代码封装到run方法中去；iii .对新创建的类new出一个对象，并作为Thread类构造方法的参数传递进去；从而就可以创建一个Thread类的线程对象。iiii．调用线程对象（Thread类的对象）的start方法启动线程。

方法2的原理：（为什么这么做？）

因为线程的任务都封装在线程类中的run方法中，所以要在创建线程对象（Thread对象）时，必须明确此线程要执行的任务，即需要把创建类的对象作为参数传进去。

任务只是任务，但是要想执行，需要找个代理，Thread就充当代理的角色。

采用静态代理Thread

a． 创建真实角色对象：任务对象；

b． 创建代理角色对象+传入真实角色对象；

c． 代理角色.start();

优点：1）避免了单继承的局限性；2）便于共享数据资源。

## 注意：run()方法和start()方法的区别：

**run()方法**只是个普通的方法，调用run()方法并不能启动此线程，和普通方法执行一样，只有调用start方法才会启动该线程，启动后线程自动执行run()方法中的任务。

## 任何方法都会隐式地抛出一个RuntimeException异常。由于Runnable的run方法没有抛出异常，所以创建的新的线程出现了异常只能try…catch…或者抛出Runtime异常。

## 创建线程的这两个方法中，以第二种方法常见，实现Runnable接口，对任务进行单独封装。

## 第一种方法：利用继承Thread类：

（注意Thread是一个线程类，不是接口，存在于java.lang包中）

创建一个线程类(如 MyThread)继承java.lang.Thread类，并重写run方法，调用线程类对象的start方法即可启动此线程。这种方法是根据具体的任务封装成一个线程类，也就是这个类创建出的线程对象执行的任务完全相同，不能实现多个线程同时执行同一个任务。

示例：

（1） 首先创建一个线程类MyThread并重写run方法，new出一个线程对象，调用start方法即可开启线程。

package threadx.test;

public class MyThread extends Thread {

@Override

public void run() {

System.out.println("创建一个线程类，通过线程类new出线程对象");

}

}

main函数：

MyThread myThread = new MyThread();

myThread.start();//即可开启线程

（2） 利用父类引用变量指向子类的对象，采用匿名类1的方式：

Thread th1 = new Thread() {

public void run() {

System.out.println("这是第一个线程");

}

};

th1.start();

（3） 匿名类方式2：

new Thread() {

@Override

public void run() {

System.out.println("第一种方法创建线程");

}

}.start();

## 第二种方法：把任务进行封装成对象，然后把任务对象传入Thread对象中。

（1）任务封装的方法：创建一个类，需要实现Runnable接口，且必须实现其中的run() 方法。

package threadx.test;

public class Target1 implements Runnable {

@Override

public void run() {

System.out.println("任务封装，构造线程");

} }

main方法中：

Target1 target = new Target1();

Thread th = new Thread(target);

th.start();

简化代码：(new Thread(new Target1())).start();

（2） 匿名类方式创建任务：

(new Thread(new Runnable() {

@Override

public void run() {

System.out.println("匿名类封装任务对象");

}

})).start();

## 两种方法的比较：

（1） 共同点：要想创建一个线程必须实现Runnable接口，不过Thread类已经实现了Runnable接口，且Java在设计时已经把关于线程的属性和行为封装到了Thread类中，故要想创建线程必须基于Thread类。

（2） 不同点：第一种方法直接把Thread类继承下来，并需要把Thread类的任务进行重写，出现了线程子类，但是线程子类的任务是固定的，且不能实现多个线程共同完成同一个任务，因为这样每个线程子类都具有自己的任务。第二种方法是把任务单独封装成一个类，然后创建任务对象，就可以实现同一个任务对象，把此任务对象传入多个线程中，就可以实现多个线程共同完成同一个任务。

（3） 两种方法的比较：

（1）第一个是继承类Thread，重写Thread类中的run方法，第二个是实现接口Runnable，实现接口的run方法，并同时可以继承其他的一个类；

（2）采用继承Thread类的方法创建一个新的线程类，不仅仅是覆写了run方法，而且会把Thread类中的所有方法继承下来，而实现接口Runnable，只需要实现run方法，封装的仅仅只是线程的任务，且可以实现多个线程执行同一个任务，这种方式可以解决很多实际问题如多窗口售票。

（3） 实现Runnable接口的方法的好处:

①将线程的任务从线程的子类中分离出来，进行了单独的封装，按照面向对象的思想，就是把线程的任务封装成对象；②避免了Java单继承的局限性：由于Java中的类是单继承的，所以如果一个类已经继承了另外一个类，此时又想作为一个线程类，那么此类就不能继承Thread类了，只能采用实现Runnable接口的方法。

因此，利用实现Runnable接口的方法创建线程比较常用，所以以后需要创建新线程时，就去实现Runnable接口，然后作为参数传进Thread的构造方法中。

## 综合两种创建线程的方法：

new Thread(new Runnable() {

@Override

public void run() {

System.out.println("任务封装的匿名对象类");

}

}) {

public void run() {

System.out.println("子类线程的任务");

}

}.start();

解析： 首先是利用匿名实现接口的方式创建一个匿名任务类，传进Thread类中，又利用匿名继承的方式创建了一个子类，对父类Thread的run方法覆盖，所以最后调用的是子类线程的run方法。

## JVM创建的主线程的任务都定义在了主方法中；而自定义的线程的任务在哪里呢？为什么要重写run方法呢？

答：Thread类用于描述线程，线程是需要任务的，所以Thread类也对任务进行描述；这个任务就是通过Thread类中的run方法来体现，也就是说，run方法就是封装自定义线程运行任务的方法。run方法中定义的就是线程需要执行的任务代码。Runnable接口中就定义了run方法，Thread类实现了Runnable接口中的此方法。

## 10. 抛出异常要看四部分信息：

①异常所属的线程，②异常的名称，③异常的提示信息，④异常发生的位置。

## 11. 对于一个线程，不能多次启动start；特别是当线程结束后不能再次启动。否则会抛出异常 IllegalThreadStateException。

注意：

（1） 搞清楚什么是线程，对于第一种方法，新建的类的对象就是一个线程，因为与Thread类是继承关系，而第二种方法创建的类只是封装了线程任务，线程对象还是Thread对象。

如下面前两个都是错误的，而第三个是创建了两个线程，没有错误。

错误

错误

正确

（2） 弄清楚是哪个线程出现了异常。

一个线程开启了多次，但不是此线程出现了异常，而是第二次开启此线程的所处的线程发生了异常。如下面这个是在main方法中产生异常IllegalThreadStateException：

public static void main(String[] args) {

……

t1.start();t2.start();

t3.start();

t3.start();//此处产生异常，使main线程停止，后面的语句不再进行，即线程t4 便不会启动，但是前面三个线程仍然照常进行，只不过是main进程停止，并不影响t3进程。

t4.start(); //main线程异常，此语句不会执行 }

## 面试题：

class Test3 implements Runnable{

public void run(Thread t) {

}

}

请问有没有错误？有错误，错在第几行？

答：有错误，错在第一行，原因：要想实现接口必须实现接口中的所有方法，而接口Runnable的run方法是空参数的，所以此处并没有实现接口中的方法，如果没有实现接口中的方法，那么此类只能定义为抽象类，所以错在第一行，应该声明为abstract。（接口Runnable中只有一个空参数的run方法）

## 创建线程的第三种方法：call既可以返回值，也可以抛出异常。

这里用到的一些接口或类主要是用于服务器的线程编写，功能很强大。既可以返回值，也可以抛出异常，也可以终止线程等等。

（1） 使用的是java.util.Collable<V>接口。

只有一个方法call（），返回V类型。

（2） 类Executors：位于java.util.concurrent包中。

利用其静态方法：获取一个ExecutorService对象。

（3） 利用ExecutorService 实现对象的submit方法获取Future接口的实现对象。

（4） 再次调用Future实现对象的get方法即可获取call方法的返回值。

（5） 示例：演示龟兔赛跑。

import java.util.concurrent.Callable;

public class Race implements Callable<Integer> {

private String name;

private long time = 100;

private boolean flag = true;

private int step = 0;

public void setFlag(boolean flag) {

this.flag = flag;

}

public Race(String name) {

super();

this.name = name;

}

public Race(String name, long time) {

super();

this.name = name;

this.time = time;

}

@Override

public Integer call() throws Exception {//既可以返回数据，也可以抛出异常

while(flag) {

Thread.sleep(time);

this.step++;

}

return this.step;

}

}

测试：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class ThirdMethodCreation {

public static void main(String[] args) throws Exception {

Race rabbit = new Race("rabbit",50);//任务1

Race tortoise = new Race("tortoise",20);//任务2

ExecutorService es = Executors.newFixedThreadPool(2);//服务器

Future<Integer> future1 = es.submit(rabbit);//开启线程1

Future<Integer> future2 = es.submit(tortoise);//开启线程2

Thread.sleep(1000);

rabbit.setFlag(false);//停止线程

tortoise.setFlag(false); //停止线程

int stepOfRabbit = future1.get();//获取值

int stepOfTortoise = future2.get();//获取值等到线程结束后才可以获取结果，应该是一个阻塞式方法，直到线程结束，获取了结果才继续进行

System.out.println(stepOfRabbit+"\r\n"+stepOfTortoise);

es.shutdownNow();

}

}

# 开启一个线程必须是调用start方法，如果是调用run方法，就相当于调用一个普通方法一样，执行完run方法后，代码才会继续进行下面的。利用start方法开启线程后，就会直接和主线程同时随机的执行。