Java中创建线程的两个方法

1. 简单描述创建线程的方法：
   1. **继承Thread类+覆盖run方法**

启动：创建子类对象+对象.start方法。

* 1. 实现Runnable接口+实现run方法

启动：采用静态代理Thread

* + 1. 创建真是角色对象：任务对象；
    2. 创建代理角色对象+传入真实角色对象；
    3. 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。

* 1. 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。**

1. 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trycatch或抛出Runtime异常。
2. **创建一个新的线程有两个方法**：
3. **继承Thread类**，并**重写Thread类中的run方法**：

创建步骤：① 定义一个类并继承**Thread类**，② 重写Thread类中的**run方法，**将要执行的代码写到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**。

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**，只有调用**start方法**才会启动一个线程，启动后**自动执行run方法中的任务**。

1. **实现接口Runnable，并覆写其中的run方法**：

**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。

1. 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。
2. 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**

1. 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();

1. **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();

1. 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();

1. 两种方法的比较：
2. **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。
3. **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。
4. **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。

1. 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**。

1. 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。

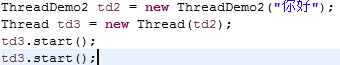
1. **抛出异常**要看四部分信息：①异常所属的线程，②异常的名称，③异常的提示信息，④异常发生的位置。
2. 对于一个**线程**，不能**多次启动start**；特别是**当线程结束后不能再次启动**。否则会抛出异常 IllegalThreadStateException。

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

* 1. **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 }

1. 面试题：

class Test3 implements Runnable{

public void run(Thread t) {

}

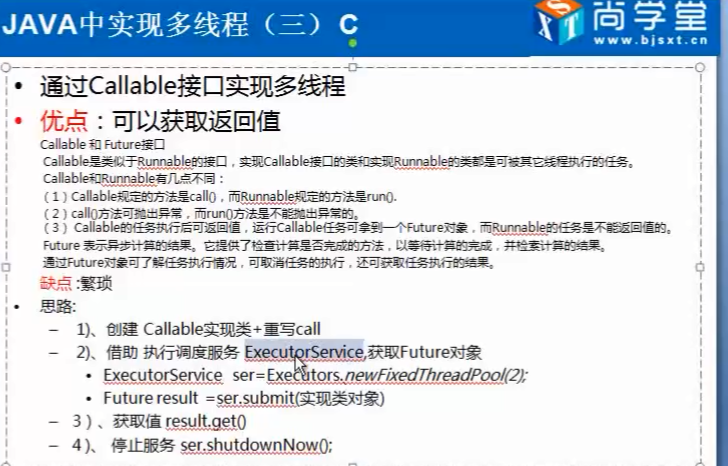
}

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**）

1. 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。



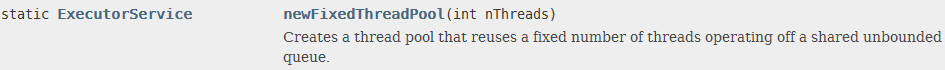
* 1. 使用的是java.util.Collable<V>接口。

只有一个方法call（），返回V类型。

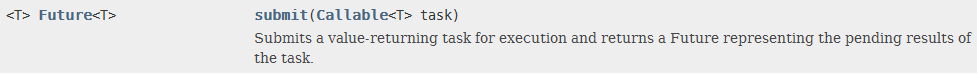


* 1. 类Executors：位于java.util.concurrent包中。

利用其静态方法：获取一个[ExecutorService](mk:@MSITStore:E:\JAVA\JAVA%20JIHE\API帮助文档\CHM帮助文档\jdk-8u60-docs-all_API.chm::/api/java/util/concurrent/ExecutorService.html)对象。



* 1. 利用[ExecutorService](mk:@MSITStore:E:\JAVA\JAVA%20JIHE\API帮助文档\CHM帮助文档\jdk-8u60-docs-all_API.chm::/api/java/util/concurrent/ExecutorService.html) 实现对象的submit方法获取Future接口的实现对象。



* 1. 再次调用Future实现对象的get方法即可获取call方法的返回值。
  2. 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();

}

}