并发基础篇（5）： 创建线程的四种方式

概述

线程的创建一共有四种方式：

- 继承于 Thread 类，重写 run（ ）方法；
- 实现 Runable 接口，实现里面的 run（ ）方法；
- 使用 FutureTask 实现有返回结果的线程
- 使用 ExecutorService、Executors 线程池。

在详细了解这四种方法之前，先来理解一下为什么线程要这样创建：形象点来说，Thread 是一个工人，run（ ）方法里面的便是他的任务栏，这个任务栏默认是空的。当你想要这个线程做点什么时，你可以重写 Thread 里面的 run 方法，重写这个工人的任务栏；也可以通过 runable、callable 接口 ,从外部赋予这个工人任务。还可以将任务交给一堆工人 ,谁有空就谁就承担这个任务(线程池)。

一、四种方式的详细介绍

1、继承于 Thread 类，重写 run（ ）方法

```
//继承 Thread
class MyThread extends Thread{
    //重写 run 方法
    @Override
    public void run() {
        //任务内容....
        System.out.println("当前线程是："+Thread.currentThread().getName());
    }
}

运行结果：
   当前线程是：Thread-0
```

如果线程类使用的很少，那么可以使用匿名内部类，请看下面的例子：

```
Thread thread = new Thread(){
        @Override
        public void run() {
            //任务内容....
```

```
            System.out.println("当前线程是："+Thread.currentThread().getName());
        }
};
```

2、实现 Runable 接口，实现里面的 run（ ）方法：

第一种方法- -继承 Thread 类的方法，一般情况下是不建议用的，因为 java 是单继承结构，一旦继承了 Thread 类，就无法继承其他类了。所以建议使用 实现 Runable 接口 的方法；

```
Thread thread = new Thread(new MyTask());
    //线程启动
thread.start();
```

MyTask 类：

```
//实现 Runnable 接口
class MyTask implements Runnable{
    //重写 run 方法
    public void run() {
        //任务内容....
        System.out.println("当前线程是："+Thread.currentThread().getName());
    }
}
```

同样，如果这个任务类（MyTask ）用的很少，也可以使用匿名内部类：

```
Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
            //任务内容....
            System.out.println("当前线程是："+Thread.currentThread().getName());
        }
});
```

3、使用 FutureTask 实现有返回结果的线程

FutureTask 是一个可取消的异步计算任务 ,是一个独立的类 ,实现了 Future、Runnable 接口。**FutureTask 的出现是为了弥补 Thread 的不足而设计的,可以让程序员跟踪、获取任务的执行情况、计算结果 。**

因为 FutureTask 实现了 Runnable ,所以 FutureTask 可以作为参数来创建一个新的线程来执行，也可以提交给 Executor 执行。FutureTask 一旦计算完成，就不能再重新开始或取消计算。

FutureTask 的构造方法

可以接受 Runnable,Callable 的子类实例。

```
//创建一个 FutureTask，一旦运行就执行给定的 Callable。
public FutureTask(Callable<V> callable);
//创建一个 FutureTask，一旦运行就执行给定的 Runnable，并安排成功完成时 get 返回给定的结果 。
public FutureTask(Runnable runnable, V result)
```

FutureTask 的简单例子

```
public class Test {
  public static void main(String[] args) throws InterruptedException, ExecutionException
{
    FutureTask<Double> task = new FutureTask(new MyCallable());
    //创建一个线程，异步计算结果
    Thread thread = new Thread(task);
    thread.start();
    //主线程继续工作
    Thread.sleep(1000);
    System.out.println("主线程等待计算结果...");
    //当需要用到异步计算的结果时，阻塞获取这个结果
    Double d = task.get();
    System.out.println("计算结果是："+d);
    //用同一个 FutureTask 再起一个线程
    Thread thread2 = new Thread(task);
    thread2.start();
}
}
class MyCallable implements Callable<Double>{
    @Override
    public Double call() {
         double d = 0;
         try {
             System.out.println("异步计算开始.......");
              d = Math.random()*10;
              d += 1000;
            Thread.sleep(2000);
             System.out.println("异步计算结束.......");
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        return d;
    }
}
```

运行结果：

异步计算开始…….

主线程等待计算结果…

异步计算结束…….

计算结果是：1002.7806590582911

4、使用线程池 ExecutorSerice、Executors

前面三种方法，都是显式地创建一个线程，可以直接控制线程，如线程的优先级、线程是否是守护线程，线程何时启动等等。而第四种方法，则是创建一个线程池，池中可以有 1 个或多个线程，这些线程都是线程池去维护 ,控制程序员不需要关心这些细节 ,只需要将任务提交给线程池去处理便可，非常方便。

创建线程池的前提最好是你的任务量大，因为创建线程池的开销比创建一个线程大得多。

创建线程池的方式

ExecutorService 是一个比较重要的接口，实现这个接口的子类有两个 ThreadPoolExecutor (普通线程池)、ScheduleThreadPoolExecutor (定时任务的线程池)。你可以通过这两个类来创建一个线程池，但要传入各种参数，不太方便。

为了方便用户，JDK 中提供了工具类 Executors，提供了几个创建常用的线程池的工厂方法。由于篇幅原因，不细说，可参考我的并发系列文章。

Executors 创建单线程的线程池

```
public class MyTest {
    public static void main(String[] args) {
        //创建一个只有一个线程的线程池
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        //创建任务，并提交任务到线程池中
        executorService.execute(new MyRunable("任务 1"));
        executorService.execute(new MyRunable("任务 2"));
        executorService.execute(new MyRunable("任务 3"));
    }
}
class MyRunable implements Runnable{
```

```
    private String taskName;
    public MyRunable(String taskName) {
        this.taskName = taskName;
    }
    @Override
    public void run() {
        System.out.println("线程池完成任务："+taskName);
    }
}
```

二、关于 run（ ）方法的思考

看看下面这种情况：线程类 Thread 接收了外部任务，同时又用匿名内部类的方式重写了内部的 run（ ）方法，这样岂不是有两个任务，那么究竟会执行那个任务呢？还是两个任务一起执行呢？

```
Thread thread = new Thread(new MyTask()){
        @Override
        public void run() {//重写 Thread 类的 run 方法
            System.out.println("Thread 类的 run 方法");
        }
    };
    //线程启动
    thread.start();

//实现 Runnable 接口
class MyTask implements Runnable{
    //重写 run 方法
    @Override
    public void run() {
        //任务内容....
        System.out.println("这是 Runnable 的 run 方法");
    }
}

运行结果：
   Thread 类的 run 方法
```

通过上面的结果，可以看出：线程最后执行的是 Thread 类内部的 run（ ）方法，这是为什么呢？我们先来分析一下 JDK 的 Thread 源码：

```
private Runnable target;
```

```
public void run() {
    if (target != null) {
        target.run();
    }
}
```

一切都清晰明了了，Thread 类的 run 方法在没有重写的情况下，是判断一下是否有 Runnable 对象传进来，如果有，那么就调用 Runnable 对象里的 run 方法；否则，就什么都不干，线程结束。所以，针对上面的例子，一旦你继承重写了 Thread 类的 run（）方法，而你又想可以接收 Runable 类的对象，那么就要加上 super.run（），执行没有重写时的 run 方法，改造的例子如下：

```
Thread thread = new Thread(new MyTask()){
        @Override
        public void run() {//重写 Thread 类的 run 方法
           //调用父类 Thread 的 run 方法，即没有重写时的 run 方法
            super.run();
            System.out.println("Thread 类的 run 方法");
        }
    };

运行结果：
   这是 Runnable 的 run 方法
   Thread 类的 run 方法
```