**Java多线程实现的四种方式**

**1.继承Thread类，重写run方法**

**2.实现Runnable接口，重写run方法，实现Runnable接口的实现类的实例对象作为Thread构造函数的target**

**3.通过Callable和FutureTask创建线程**

**4.通过线程池创建线程**

前面两种可以归结为一类：**无返回值**，原因很简单，通过重写run方法，run方式的返回值是void，所以没有办法返回结果

后面两种可以归结成一类：**有返回值**，通过Callable接口，就要实现call方法，这个方法的返回值是Object，所以返回的结果可以放在Object对象中

**方式1：继承Thread类的线程实现方式如下：**

public class ThreadDemo01 **extends Thread**{

public ThreadDemo01(){

//编写子类的构造方法，可缺省

}

**public void run()**{

//编写自己的线程代码

System.out.println(Thread.currentThread().getName());

}

public static void main(String[] args){

ThreadDemo01 threadDemo01 = new ThreadDemo01();

threadDemo01.setName("我是自定义的线程1");

threadDemo01.start();

System.out.println(Thread.currentThread().toString());

}

}

程序结果：

Thread[main,5,main]

我是自定义的线程1

线程实现方式2：**通过实现Runnable接口，实现run方法，接口的实现类的实例作为Thread的target作为参数传入带参的Thread构造函数，通过调用start()方法启动线程**

public class ThreadDemo02 {

public static void main(String[] args){

System.out.println(Thread.currentThread().getName());

**Thread t1 = new Thread(new MyThread());**

**t1.start();**

}

}

class MyThread **implements Runnable**{

@Override

**public void run()** {

// TODO Auto-generated method stub

System.out.println(Thread.currentThread().getName()+"-->我是通过实现接口的 线程实现方式！");

}

}

程序运行结果：

main

Thread-0–>我是通过实现接口的线程实现方式！

线程实现方式3：**通过Callable和FutureTask创建线程**

a:**创建Callable接口的实现类 ，并实现Call方法**

b:创建Callable实现类的实现，**使用FutureTask类包装Callable对象**，**该FutureTask对象封装了Callable对象的Call方法的返回值**

c:使用**FutureTask对象**作为Thread对象的target创建并启动线程

d:调用FutureTask对象的get()来获取子线程执行结束的返回值

public class ThreadDemo03 {

public static void main(String[] args) {

**Callable<Object> oneCallable = new Tickets<Object>();**

**FutureTask<Object> oneTask = new FutureTask<Object>(oneCallable);**

**Thread t = new Thread(oneTask);**

System.out.println(Thread.currentThread().getName());

**t.start();**

}

}

**class Tickets<Object> implements Callable<Object>**{

//重写call方法

@Override

**public Object call() throws Exception** {

System.out.println(Thread.currentThread().getName()+"-->我是通过实现Callable接口 通过FutureTask包装器来实现的线程");

return null;

}

}

程序运行结果：

main

Thread-0–>我是通过实现Callable接口通过FutureTask包装器来实现的线程

线程实现方式4：**通过线程池创建线程**

public class ThreadDemo05{

private static int POOL\_NUM = 10; //线程池数量

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

**ExecutorService executorService = Executors.newFixedThreadPool(5);**

for(int i = 0; i<POOL\_NUM; i++)

{

RunnableThread thread = new RunnableThread();

//Thread.sleep(1000);

**executorService.execute(thread);**

}

//关闭线程池

**executorService.shutdown();**

}

}

class RunnableThread **implements Runnable**{

@Override

**public void run()**{

System.out.println("通过线程池方式创建的线程：" + Thread.currentThread().getName() + " ");

}

}

程序运行结果：

通过线程池方式创建的线程：pool-1-thread-3

通过线程池方式创建的线程：pool-1-thread-4

通过线程池方式创建的线程：pool-1-thread-1

通过线程池方式创建的线程：pool-1-thread-5

通过线程池方式创建的线程：pool-1-thread-2

通过线程池方式创建的线程：pool-1-thread-5

通过线程池方式创建的线程：pool-1-thread-1

通过线程池方式创建的线程：pool-1-thread-4

通过线程池方式创建的线程：pool-1-thread-3

通过线程池方式创建的线程：pool-1-thread-2

ExecutorService、Callable都是属于Executor框架。返回结果的线程是在JDK1.5中引入的新特征，还有Future接口也是属于这个框架，有了这种特征得到返回值就很方便了。

通过分析可以知道，他同样也是实现了Callable接口，实现了Call方法，所以有返回值。这也就是正好符合了前面所说的两种分类

**执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。get方法是阻塞的，即：线程无返回结果，get方法会一直等待。**

再介绍Executors类：提供了一系列工厂方法用于**创建线程池**，返回的线程池都实现了ExecutorService接口。

**public static ExecutorService newFixedThreadPool(int nThreads)**

创建固定数目线程的线程池。

public static ExecutorService newCachedThreadPool()

创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static ExecutorService newSingleThreadExecutor()

创建一个单线程化的Executor。

public static ScheduledExecutorService newScheduledThreadPool(int

corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

ExecutoreService提供了**submit()**方法，传递一个Callable，或Runnable，返回Future。如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。