链滴

实例带你获取多线程 Thread 的返回值之 (贰) - Callable 配合线程池返回数据

作者：adlered

原文链接：https://ld246.com/article/1552643240583

来源网站：链滴

前言

阅读本篇文章，你需要先理解以下知识：

- 第一章：Callable的使用（ 点我跳转）
- 多线程Thread的基本使用 （ 点我跳转）
- 线程池基本知识 （ 点我跳转）
- extends和implements
- 重写Override
- try catch错误处理
- Java基础知识

回顾

在上一章（点我跳转）我们了解了Callable的基本使用，本次我们将把Callable运用到线程池（点我转）中。

拷贝

用你的IDE新建一个项目或类，并将类命名为TestThreadPool，然后将下面的代码替换到类中：

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class TestThreadPool {
    ExecutorService executorService = Executors.newFixedThreadPool(2);
    public static void main(String[] args) {
        //实例化类
        TestThreadPool testThreadPool = new TestThreadPool();
```

```
        //调用动态方法
        testThreadPool.threadPool();
    }

    public void threadPool() {
        Thread1 thread1 = new Thread1();
        Thread2 thread2 = new Thread2();
        //将Future包装进List，实现添加结果
        List<Future> resultList = new ArrayList<Future>();
        for (int i = 0; i < 3; i++) {
            System.out.println("线程池已提交：" + i);
            Future res1 = executorService.submit(thread1);
            Future res2 = executorService.submit(thread2);
            //将获取的结果添加进List
            resultList.add(res1);
            resultList.add(res2);
        }
        System.out.println("正在关闭线程池...");
        executorService.shutdown();
        System.out.println("线程池已关闭.");
        //executorService.shutdownNow();
        //线程池运行结束，打印结果
        for (int i = 0; i < resultList.size(); i++) {
            Future future = resultList.get(i);
            try {
                System.out.println(future.get());
            } catch (InterruptedException | ExecutionException e) {}
        }
    }
}

/**
 * 线程1
 */
class Thread1 implements Callable {
    @Override
    public Object call() throws Exception {
        try {
            Thread.sleep(500);
        } catch (Exception e) {}
        return "本条数据来自线程1";
    }
}

/**
 * 线程2
 */
class Thread2 implements Callable {
    @Override
    public Object call() throws Exception {
        try {
            Thread.sleep(500);
        } catch (Exception e) {}
        return "本条数据来自线程2";
```

```
    }
}
```

对比

和线程池第二章的文章（点我跳转）中的实例代码对比一下，你会发现它们大概是相同的，此时再回一下Callable中的实例代码（点我跳转），你会发现这篇是这两篇的结合。

区别

定义的两个线程类：

```
/**
 * 线程1
 */
class Thread1 implements Callable {
    @Override
    public Object call() throws Exception {
        try {
            Thread.sleep(500);
        } catch (Exception e) {}
        return "本条数据来自线程1";
    }
}

/**
 * 线程2
 */
class Thread2 implements Callable {
    @Override
    public Object call() throws Exception {
        try {
            Thread.sleep(500);
        } catch (Exception e) {}
        return "本条数据来自线程2";
    }
}
```

你会发现它不再使用Runnable了，而是使用了Callable以支持返回数据。并且重写的方法不再是run()而是call()。我们使用return返回了String类型的字符串。

调用方法

```
public void threadPool() {
        Thread1 thread1 = new Thread1();
        Thread2 thread2 = new Thread2();
        //将Future包装进List，实现添加结果
        List<Future> resultList = new ArrayList<Future>();
        for (int i = 0; i < 3; i++) {
            System.out.println("线程池已提交：" + i);
            Future res1 = executorService.submit(thread1);
            Future res2 = executorService.submit(thread2);
```

```
            //将获取的结果添加进List
            resultList.add(res1);
            resultList.add(res2);
        }
        System.out.println("正在关闭线程池...");
        executorService.shutdown();
        System.out.println("线程池已关闭.");
        //executorService.shutdownNow();
        //线程池运行结束，打印结果
        for (int i = 0; i < resultList.size(); i++) {
            Future future = resultList.get(i);
            try {
                System.out.println(future.get());
            } catch (InterruptedException | ExecutionException e) {}
        }
    }
```

该方法仍是使用了同样的线程池，但执行方法使用了submit()而不是execute()。因为execute()方法支持Runnable，请注意。

我们将Future套入了一个List中，以便异步循环写入每个线程执行后返回的结果。

请仔细阅读调试，这并不难理解。

运行!

现在，运行你的代码，你会看到以下结果：

```
线程池已提交：0
线程池已提交：1
线程池已提交：2
正在关闭线程池...
线程池已关闭.
本条数据来自线程1
本条数据来自线程2
本条数据来自线程1
本条数据来自线程2
本条数据来自线程1
本条数据来自线程2
```

后语

Java线程与线程池的知识点实际上是很多的。使用多线程是为了拥有更强的性能和更灵活的调用能力同时也是每个合格的程序员必会的知识点。