Java 并发编程：Callable、Future 和 FutureTask

Java 并发编程：Callable、Future 和 FutureTask

在前面的文章中我们讲述了创建线程的 2 种方式，一种是直接继承 Thread，另外一种就是实现 Runnable 接口。

这 2 种方式都有一个缺陷就是：在执行完任务之后无法获取执行结果。

如果需要获取执行结果，就必须通过共享变量或者使用线程通信的方式来达到效果，这样使用起来就比较麻烦。

而自从 Java 1.5 开始，就提供了 Callable 和 Future，通过它们可以在任务执行完毕之后得到任务执行结果。

今天我们就来讨论一下 Callable、Future 和 FutureTask 三个类的使用方法。以下是本文的目录大纲：

一.Callable 与 Runnable

二.Future

三.FutureTask

四.使用示例

若有不正之处请多多谅解，并欢迎批评指正。

请尊重作者劳动成果，转载请标明原文链接：

http://www.cnblogs.com/dolphin0520/p/3949310.html

一.Callable 与 Runnable

先说一下 java.lang.Runnable 吧，它是一个接口，在它里面只声明了一个 run()方法：

```
1
2
3
public interface Runnable {
    public abstract void run();
}
```

由于 run()方法返回值为 void 类型，所以在执行完任务之后无法返回任何结果。

Callable 位于 java.util.concurrent 包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做 call()：

```
1
2
3
4
5
6
7
8
9
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

可以看到，这是一个泛型接口，call()函数返回的类型就是传递进来的 V 类型。

那么怎么使用 Callable 呢？一般情况下是配合 ExecutorService 来使用的，在 ExecutorService 接口中声明了若干个 submit 方法的重载版本：

```
1
2
3
<T> Future<T> submit(Callable<T> task);
<T> Future<T> submit(Runnable task, T result);
Future<?> submit(Runnable task);
```

第一个 submit 方法里面的参数类型就是 Callable。

暂时只需要知道 Callable 一般是和 ExecutorService 配合来使用的，具体的使用方法讲在后面讲述。

一般情况下我们使用第一个 submit 方法和第三个 submit 方法，第二个 submit 方法很少使用。

二.Future

Future 就是对于具体的 Runnable 或者 Callable 任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过 get 方法获取执行结果，该方法会阻塞直到任务返回结果。

Future 类位于 java.util.concurrent 包下，它是一个接口：

```
1
2
3
4
5
6
7
8
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

在 Future 接口中声明了 5 个方法，下面依次解释每个方法的作用：

- cancel 方法用来取消任务，如果取消任务成功则返回 true，如果取消任务失败则返回 false。参数 mayInterruptIfRunning 表示是否允许取消正在执行却没有执行完毕的任务，如果设置 true，则表示可以取消正在执行过程中的任务。如果任务已经完成，则无论 mayInterruptIfRunning 为 true 还是 false，此方法肯定返回 false，即如果取消已经完成的任务会返回 false；如果任务正在执行，若 mayInterruptIfRunning 设置为 true，则返回 true，若 mayInterruptIfRunning 设置为 false，则返回 false；如果任务还没有执行，则无论 mayInterruptIfRunning 为 true 还是 false，肯定返回 true。
- isCancelled 方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。
- isDone 方法表示任务是否已经完成，若任务完成，则返回 true；

- get()方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；
- get(long timeout, TimeUnit unit)用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回 null。

也就是说 Future 提供了三种功能：

1）判断任务是否完成；

2）能够中断任务；

3）能够获取任务执行结果。

因为 Future 只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的 FutureTask。

三.FutureTask

我们先来看一下 FutureTask 的实现：

```
1  public class FutureTask<V> implements RunnableFuture<V>
```

FutureTask 类实现了 RunnableFuture 接口，我们看一下 RunnableFuture 接口的实现：

```
1  public interface RunnableFuture<V> extends Runnable, Future<V> {
2      void run();
3  }
```

可以看出 RunnableFuture 继承了 Runnable 接口和 Future 接口，而 FutureTask 实现了 RunnableFuture 接口。所以它既可以作为 Runnable 被线程执行，又可以作为 Future 得到 Callable 的返回值。

FutureTask 提供了 2 个构造器：

```
public FutureTask(Callable<V> callable) {
}
public FutureTask(Runnable runnable, V result) {
}
```

事实上，FutureTask 是 Future 接口的一个唯一实现类。

四.使用示例

1.使用 Callable+Future 获取执行结果

```
public class Test {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        Task task = new Task();
        Future<Integer> result = executor.submit(task);
        executor.shutdown();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }

        System.out.println("主线程在执行任务");

        try {
            System.out.println("task 运行结果"+result.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("所有任务执行完毕");
    }
}
class Task implements Callable<Integer>{
    @Override
    public Integer call() throws Exception {
        System.out.println("子线程在进行计算");
        Thread.sleep(3000);
        int sum = 0;
```

```
            for(int i=0;i<100;i++)
                sum += i;
            return sum;
        }
    }
```

执行结果：

```
子线程在进行计算
主线程在执行任务
task 运行结果 4950
所有任务执行完毕
```

2.使用 Callable+FutureTask 获取执行结果

```
public class Test {
    public static void main(String[] args) {
        //第一种方式
        ExecutorService executor = Executors.newCachedThreadPool();
        Task task = new Task();
        FutureTask<Integer> futureTask = new FutureTask<Integer>(task);
        executor.submit(futureTask);
        executor.shutdown();

        //第二种方式，注意这种方式和第一种方式效果是类似的，只不过一个使用的是 ExecutorServ
        /*Task task = new Task();
        FutureTask<Integer> futureTask = new FutureTask<Integer>(task);
        Thread thread = new Thread(futureTask);
        thread.start();*/

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e1) {
            e1.printStackTrace();
        }

        System.out.println("主线程在执行任务");

        try {
            System.out.println("task 运行结果"+futureTask.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
27          } catch (ExecutionException e) {
28              e.printStackTrace();
29          }
30
31          System.out.println("所有任务执行完毕");
32      }
33  }
34  class Task implements Callable<Integer>{
35      @Override
36      public Integer call() throws Exception {
37          System.out.println("子线程在进行计算");
38          Thread.sleep(3000);
39          int sum = 0;
40          for(int i=0;i<100;i++)
41              sum += i;
42          return sum;
43      }
44  }
45
```

如果为了可取消性而使用 Future 但又不提供可用的结果，则可以声明 Future<?> 形式类型、并返回 null 作为底层任务的结果。