二

49 Future 的主要功能是什么?

在本课时我们将讲解 Future 的主要功能是什么。

Future 类

Future 的作用

Future 最主要的作用是，比如当做一定运算的时候，运算过程可能比较耗时，有时会去查数据库，或是繁重的计算，比如压缩、加密等，在这种情况下，如果我们一直在原地等待方法返回，显然是不明智的，整体程序的运行效率会大大降低。我们可以把运算的过程放到子线程去执行，再通过 Future 去控制子线程执行的计算过程，最后获取到计算结果。这样一来就可以把整个程序的运行效率提高，是一种异步的思想。

Callable 和 Future 的关系

接下来我们介绍下 Callable 和 Future 的关系，前面讲过，Callable 接口相比于 Runnable 的一大优势是可以有返回结果，那这个返回结果怎么获取呢？就可以用 Future 类的 get 方法来获取 。因此，Future 相当于一个存储器，它存储了 Callable 的 call 方法的任务结果。除此之外，我们还可以通过 Future 的 isDone 方法来判断任务是否已经执行完毕了，还可以通过 cancel 方法取消这个任务，或限时获取任务的结果等，总之 Future 的功能比较丰富。有了这样一个从宏观上的概念之后，我们就来具体看一下 Future 类的主要方法。

Future 的方法和用法

首先看一下 Future 接口的代码，一共有 5 个方法，代码如下所示：

```
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, TimeoutExceptio

}
```

其中，第 5 个方法是对第 4 个方法的重载，方法名一样，但是参数不一样。

get() 方法：获取结果

get 方法最主要的作用就是获取任务执行的结果，该方法在执行时的行为取决于 Callable 任务的状态，可能会发生以下 5 种情况。

（1）最常见的就是**当执行 get 的时候，任务已经执行完毕了**，可以立刻返回，获取到任务执行的结果。

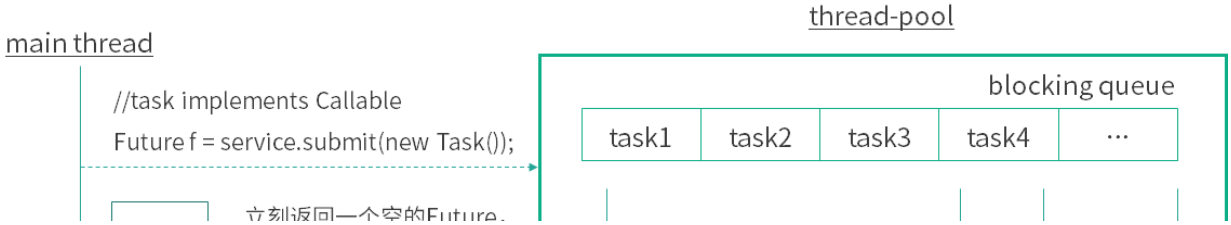
（2）**任务还没有结果**，这是有可能的，比如我们往线程池中放一个任务，线程池中可能积压了很多任务，还没轮到我去执行的时候，就去 get 了，在这种情况下，相当于任务还没开始；还有一种情况是**任务正在执行中**，但是执行过程比较长，所以我去 get 的时候，它依然在执行的过程中。无论是任务还没开始或在进行中，我们去调用 get 的时候，都会把当前的线程阻塞，直到任务完成再把结果返回回来。

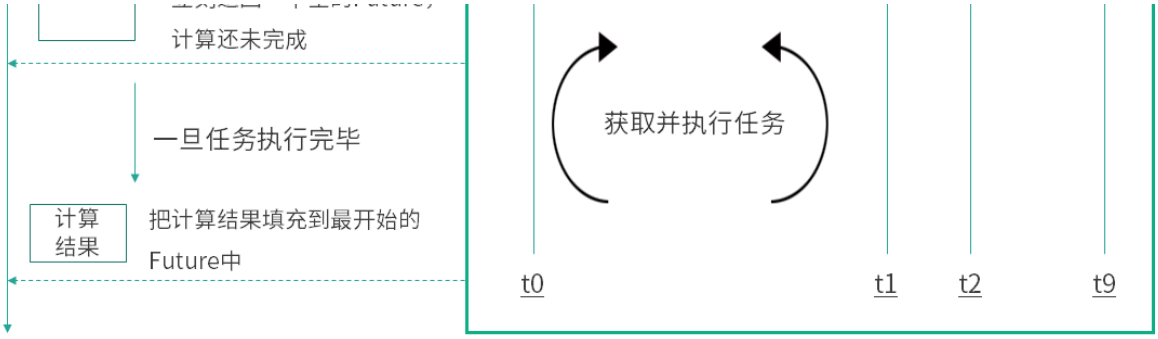
（3）**任务执行过程中抛出异常**，一旦这样，我们再去调用 get 的时候，就会抛出 ExecutionException 异常，不管我们执行 call 方法时里面抛出的异常类型是什么，在执行 get 方法时所获得的异常都是 ExecutionException。

（4）**任务被取消了**，如果任务被取消，我们用 get 方法去获取结果时则会抛出 CancellationException。

（5）**任务超时**，我们知道 get 方法有一个重载方法，那就是带延迟参数的，调用了这个带延迟参数的 get 方法之后，如果 call 方法在规定时间内正常顺利完成了任务，那么 get 会正常返回；但是如果到达了指定时间依然没有完成任务，get 方法则会抛出 TimeoutException，代表超时了。

下面用图的形式让过程更清晰：





在图中，右侧是一个线程池，线程池中有一些线程来执行任务。重点在图的左侧，可以看到有一个 submit 方法，该方法往线程池中提交了一个 Task，这个 Task 实现了 Callable 接口，当我们去给线程池提交这个任务的时候，调用 submit 方法会立刻返回一个 Future 类型的对象，这个对象目前内容是空的，其中还不包含计算结果，因为此时计算还没有完成。

当计算一旦完成时，也就是当我们可以获取结果的时候，线程池便会把这个结果填入到之前返回的 Future 中去（也就是 f 对象），而不是在此时新建一个新的 Future。这时就可以利用 Future 的 get 方法来获取到任务的执行结果了。

我们来看一个代码示例：

```
/**
 * 描述：        演示一个 Future 的使用方法
 */
public class OneFuture {

    public static void main(String[] args) {

        ExecutorService service = Executors.newFixedThreadPool(10);

        Future<Integer> future = service.submit(new CallableTask());

        try {

            System.out.println(future.get());

        } catch (InterruptedException e) {

            e.printStackTrace();

        } catch (ExecutionException e) {

            e.printStackTrace();

        }

        service.shutdown();
```

```
    }
    static class CallableTask implements Callable<Integer> {
        @Override
        public Integer call() throws Exception {
            Thread.sleep(3000);
            return new Random().nextInt();
        }
    }
}
```

在这段代码中，main 方法新建了一个 10 个线程的线程池，并且用 submit 方法把一个任务提交进去。这个任务如代码的最下方所示，它实现了 Callable 接口，它所做的内容就是先休眠三秒钟，然后返回一个随机数。接下来我们就直接把 future.get 结果打印出来，其结果是正常打印出一个随机数，比如 100192 等。这段代码对应了我们刚才那个图示的讲解，这也是 Future 最常用的一种用法。

isDone() 方法：判断是否执行完毕

下面我们再接着看看 Future 的一些其他方法，比如说 isDone() 方法，该方法是用来判断当前这个任务是否执行完毕了。

需要注意的是，这个方法如果返回 true 则代表执行完成了；如果返回 false 则代表还没完成。但这里如果返回 true，并不代表这个任务是成功执行的，比如说任务执行到一半抛出了异常。那么在这种情况下，对于这个 isDone 方法而言，它其实也是会返回 true 的，因为对它来说，虽然有异常发生了，但是这个任务在未来也不会再被执行，它确实已经执行完毕了。所以 isDone 方法在返回 true 的时候，不代表这个任务是成功执行的，只代表它执行完毕了。

我们用一个代码示例来看一看，代码如下所示：

```
public class GetException {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(20);
        Future<Integer> future = service.submit(new CallableTask());
```

```
            try {
                for (int i = 0; i < 5; i++) {
                    System.out.println(i);
                    Thread.sleep(500);
                }
                System.out.println(future.isDone());
                future.get();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
        static class CallableTask implements Callable<Integer> {
            @Override
            public Integer call() throws Exception {
                throw new IllegalArgumentException("Callable抛出异常");
            }
        }
    }
```

在这段代码中，可以看到有一个线程池，并且往线程池中去提交任务，这个任务会直接抛出一个异常。那么接下来我们就用一个 for 循环去休眠，同时让它慢慢打印出 0 ~ 4 这 5 个数字，这样做的目的是起到了一定的延迟作用。在这个执行完毕之后，再去调用 isDone() 方法，并且把这个结果打印出来，然后再去调用 future.get()。

这段代码的执行结果是这样的：

```
0
1
2
```

```
3

4

true

java.util.concurrent.ExecutionException: java.lang.IllegalArgumentException: Callab

...
```

这里要注意，我们知道这个异常实际上是在任务刚被执行的时候就抛出了，因为我们的计算任务中是没有其他逻辑的，只有抛出异常。我们再来看，控制台是什么时候打印出异常的呢？它是在 true 打印完毕后才打印出异常信息的，也就是说，在调用 get 方法时打印出的异常。

这段代码证明了三件事情：第一件事情，即便任务抛出异常，isDone 方法依然会返回 true；第二件事情，虽然抛出的异常是 IllegalArgumentException，但是对于 get 而言，它抛出的异常依然是 ExecutionException；第三个事情，虽然在任务执行一开始时就抛出了异常，但是真正要等到我们执行 get 的时候，才看到了异常。

cancel 方法：取消任务的执行

下面我们再来看一下 cancel 方法，如果不想执行某个任务了，则可以使用 cancel 方法，会有以下三种情况：

第一种情况最简单，那就是当任务还没有开始执行时，一旦调用 cancel，这个任务就会被正常取消，未来也不会被执行，那么 cancel 方法返回 true。

第二种情况也比较简单。如果任务已经完成，或者之前已经被取消过了，那么执行 cancel 方法则代表取消失败，返回 false。因为任务无论是已完成还是已经被取消过了，都不能再被取消了。

第三种情况比较特殊，就是这个任务正在执行，这个时候执行 cancel 方法是不会直接取消这个任务的，而是会根据我们传入的参数做判断。cancel 方法是必须传入一个参数，该参数叫作 **mayInterruptIfRunning**，它是什么含义呢？如果传入的参数是 true，执行任务的线程就会收到一个中断的信号，正在执行的任务可能会有一些处理中断的逻辑，进而停止，这个比较好理解。如果传入的是 false 则就代表不中断正在运行的任务，也就是说，本次 cancel 不会有任何效果，同时 cancel 方法会返回 false。

那么如何选择传入 true 还是 false 呢？

传入 true 适用的情况是，明确知道这个任务能够处理中断。

传入 false 适用于什么情况呢？

- 如果我们明确知道这个线程不能处理中断，那应该传入 false。
- 我们不知道这个任务是否支持取消（是否能响应中断），因为在大多数情况下代码是多人协作的，对于这个任务是否支持中断，我们不一定有十足的把握，那么在这种情况下也应该传入 false。
- 如果这个任务一旦开始运行，我们就希望它完全的执行完毕。在这种情况下，也应该传入 false。

这就是传入 true 和 false 的不同含义和选择方法。

isCancelled() 方法：判断是否被取消

最后一个方法是 isCancelled 方法，判断是否被取消，它和 cancel 方法配合使用，比较简单。

以上就是关于 Future 的主要方法的介绍了。

用 FutureTask 来创建 Future

除了用线程池的 submit 方法会返回一个 future 对象之外，同样还可以用 FutureTask 来获取 Future 类和任务的结果。

FutureTask 首先是一个任务（Task），然后具有 Future 接口的语义，因为它可以在将来（Future）得到执行的结果。

我们来看一下 FutureTask 的代码实现：

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

 ...

}
```

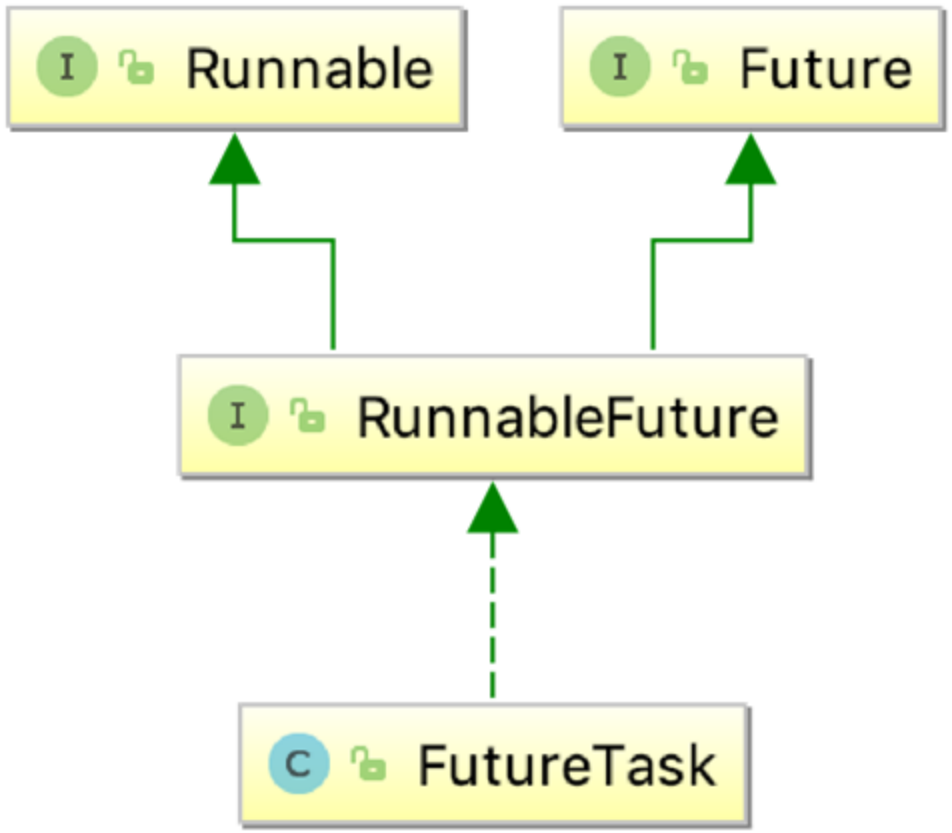
可以看到，它实现了一个接口，这个接口叫作 **RunnableFuture**。我们再来看一下 RunnableFuture 接口的代码实现：

```
public interface RunnableFuture<V> extends Runnable, Future<V> {

    void run();

}
```

可以看出，它是 extends Runnable 和 Future 这两个接口的，它们的关系如下图所示：



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

典型用法是，把 Callable 实例当作 FutureTask 构造函数的参数，生成 FutureTask 的对象，然后把这个对象当作一个 Runnable 对象，放到线程池中或另起线程去执行，最后还可以通过 FutureTask 获取任务执行的结果。

下面我们就用代码来演示一下：

```
/**
 * 描述：      演示 FutureTask 的用法
 */
```

```
public class FutureTaskDemo {

    public static void main(String[] args) {

        Task task = new Task();

        FutureTask<Integer> integerFutureTask = new FutureTask<>(task);

        new Thread(integerFutureTask).start();

        try {

            System.out.println("task运行结果：" +integerFutureTask.get());

        } catch (InterruptedException e) {

            e.printStackTrace();

        } catch (ExecutionException e) {

            e.printStackTrace();

        }

    }

}

class Task implements Callable<Integer> {

    @Override

    public Integer call() throws Exception {

        System.out.println("子线程正在计算");

        int sum = 0;

        for (int i = 0; i < 100; i++) {

            sum += i;

        }

        return sum;

    }

}
```

在这段代码中可以看出，首先创建了一个实现了 Callable 接口的 Task，然后把这个 Task 实例传入到 FutureTask 的构造函数中去，创建了一个 FutureTask 实例，并且把这个实例当作一个 Runnable 放到 new Thread() 中去执行，最后再用 FutureTask 的 get 得到结果，

并打印出来。

执行结果是 4950，正是任务里 0+1+2+...+99 的结果。

总结

最后对本课时进行一下总结，在本课时中，我们首先在宏观上讲解了 Future 的作用，然后讲解了 Callable 和 Future 的关系，接着对于 Future 的各个方法进行了详细介绍，最后还给出了 FutureTask 这种方法来创建 Future 的用法。

上一页 下一页