50 使用 Future 有哪些注意点？Future 产生新的线程了吗？

在本课时我们将讲解使用 Future 有哪些注意点，以及 Future 产生新的线程了吗？

Future 的注意点

1. 当 for 循环批量获取 Future 的结果时容易 block，get 方法调用时应使用 timeout 限制

对于 Future 而言，第一个注意点就是，当 for 循环批量获取 Future 的结果时容易 block，在调用 get 方法时，应该使用 timeout 来限制。

下面我们具体看看这是一个什么情况。

首先，假设一共有四个任务需要执行，我们都把它放到线程池中，然后它获取的时候是按照从 1 到 4 的顺序，也就是执行 get() 方法来获取的，代码如下所示：

```
public class FutureDemo {

    public static void main(String[] args) {

        //创建线程池

        ExecutorService service = Executors.newFixedThreadPool(10);

        //提交任务，并用 Future 接收返回结果

        ArrayList<Future> allFutures = new ArrayList<>();

        for (int i = 0; i < 4; i++) {

            Future<String> future;

            if (i == 0 || i == 1) {

                future = service.submit(new SlowTask());

            } else {

                future = service.submit(new FastTask());
```

```
            }
            allFutures.add(future);
        }
        for (int i = 0; i < 4; i++) {
            Future<String> future = allFutures.get(i);
            try {
                String result = future.get();
                System.out.println(result);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
        service.shutdown();
    }
    static class SlowTask implements Callable<String> {
        @Override
        public String call() throws Exception {
            Thread.sleep(5000);
            return "速度慢的任务";
        }
    }
    static class FastTask implements Callable<String> {
        @Override
        public String call() throws Exception {
            return "速度快的任务";
        }
```

```
    }
}
```

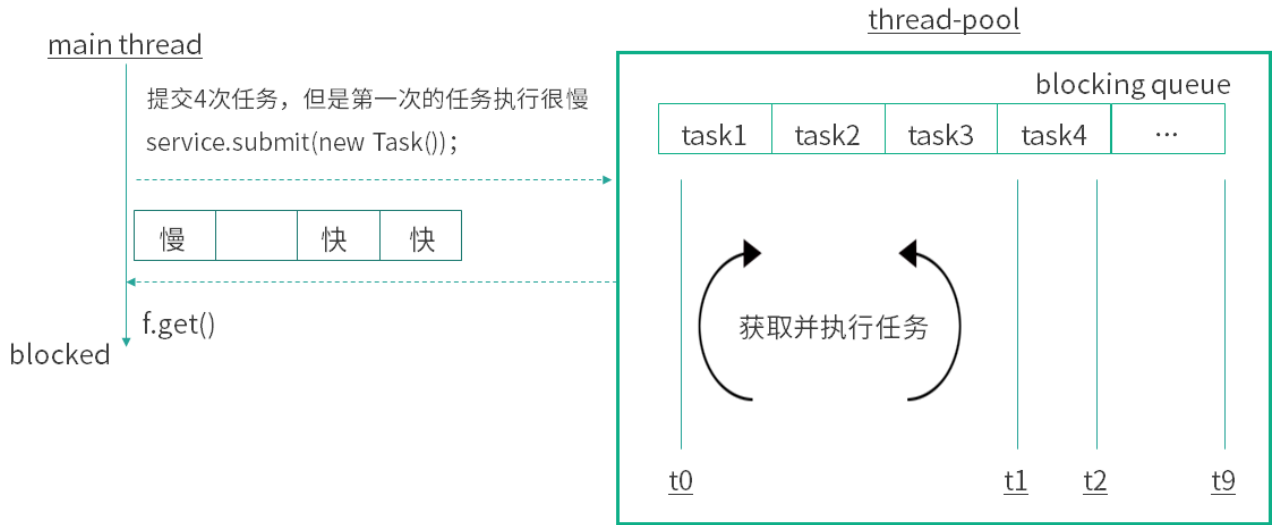
可以看出，在代码中我们新建了线程池，并且用一个 list 来保存 4 个 Future。其中，前两个 Future 所对应的任务是慢任务，也就是代码下方的 SlowTask，而后两个 Future 对应的任务是快任务。慢任务在执行的时候需要 5 秒钟的时间才能执行完毕，而快任务很快就可以执行完毕，几乎不花费时间。

在提交完这 4 个任务之后，我们用 for 循环对它们依次执行 get 方法，来获取它们的执行结果，然后再把这个结果打印出来。

执行结果如下：

```
速度慢的任务
速度慢的任务
速度快的任务
速度快的任务
```

可以看到，这个执行结果是打印 4 行语句，前面两个是速度慢的任务，后面两个是速度快的任务。虽然结果是正确的，但实际上在执行的时候会先等待 5 秒，然后再很快打印出这 4 行语句。



这里有一个问题，即第三个的任务量是比较小的，它可以很快返回结果，紧接着第四个任务也会返回结果。但是由于前两个任务速度很慢，所以我们在利用 get 方法执行时，会卡在第一个任务上。也就是说，虽然此时第三个和第四个任务很早就得到结果了，但我们在此时使用这种 for 循环的方式去获取结果，依然无法及时获取到第三个和第四个任务的结果。直到

5 秒后，第一个任务出结果了，我们才能获取到，紧接着也可以获取到第二个任务的结果，然后才轮到第三、第四个任务。

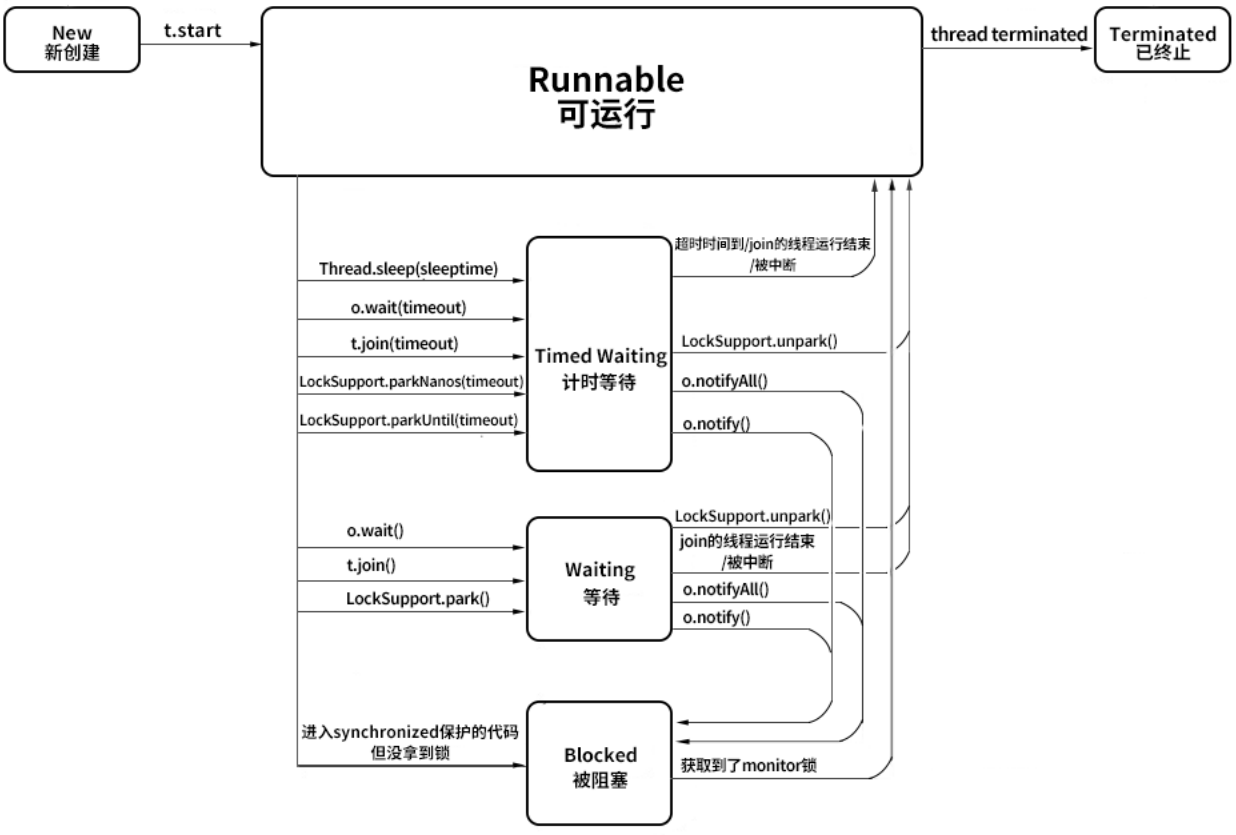
假设由于网络原因，第一个任务可能长达 1 分钟都没办法返回结果，那么这个时候，我们的主线程会一直卡着，影响了程序的运行效率。

此时我们就可以用 Future 的带超时参数的 get(long timeout, TimeUnit unit) 方法来解决这个问题。这个方法的作用是，如果在限定的时间内没能返回结果的话，那么便会抛出一个 TimeoutException 异常，随后就可以把这个异常捕获住，或者是再往上抛出去，这样就不会一直卡着了。

2. Future 的生命周期不能后退

Future 的生命周期不能后退，一旦完成了任务，它就永久停在了“已完成”的状态，不能从头再来，也不能让一个已经完成计算的 Future 再次重新执行任务。

这一点和线程、线程池的状态是一样的，线程和线程池的状态也是不能后退的。关于线程的状态和流转路径，第 03 讲已经讲过了，如图所示。



这个图也是我们当时讲解所用的图，如果有些遗忘，可以回去复习一下当时的内容。这一讲，我推荐你采用看视频的方式，因为视频中会把各个路径都标明清楚，看起来会更加清晰。

Future 产生新的线程了吗

最后我们再来回答这个问题：Future 是否产生新的线程了？

有一种说法是，除了继承 Thread 类和实现 Runnable 接口之外，还有第三种产生新线程的方式，那就是采用 Callable 和 Future，这叫作有返回值的创建线程的方式。这种说法是不正确的。

其实 Callable 和 Future 本身并不能产生新的线程，它们需要借助其他的比如 Thread 类或者线程池才能执行任务。例如，在把 Callable 提交到线程池后，真正执行 Callable 的其实还是线程池中的线程，而线程池中的线程是由 ThreadFactory 产生的，这里产生的新线程与 Callable、Future 都没有关系，所以 Future 并没有产生新的线程。

以上就是本讲的内容了。首先介绍了 Future 的两个注意点：第一个，在 get 的时候应当使用超时限制；第二个，Future 生命周期不能后退；然后又讲解了 Callable 和 Future 实际上并不是新建线程的第三种方式。

上一页 下一页