32 请按到场顺序发言—Completion Service详解

更新时间：2019-12-17 09:48:40

> 时间像海绵里的水，只要你愿意挤，总还是有的。
>
> ——鲁迅

讲解 CompletionService 之前，我们先回忆一下 ExcutorSevice。ExcutorService 实现了通过线程池来并发执行任务。其中有一种方式是通过线程池执行 Callable 任务，然后通过 Future 获取异步执行的结果，如下面的代码：

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
    ExecutorService executor = Executors.newFixedThreadPool(5);

    Callable callable1 = () -> {
        Thread.sleep(10000);
        return "任务1完成";
    };

    Callable callable2 = () -> {
        Thread.sleep(5000);
        return "任务2完成";
    };


    Future future1 = executor.submit(callable1);
    Future future2 = executor.submit(callable2);

    System.out.println(future1.get());
    System.out.println(future2.get());
}
```

任务一执行需要 10 秒，任务二执行只需要 5 秒。但是当执行到 future1.get () 时，主线程会被阻塞。等待 10 秒后第一个任务执行完才会去获取第二个任务。然后执行和第二个任务相关的打印操作。大家有没有看出问题？任务 2 明明在 5 秒前就已经执行完成，却不能立刻打印。主线程阻塞在任务一结果的获取上。这样程序执行的效率并不高。如果任务完成后能够立刻被取得执行结果，然后执行后面的逻辑，效率就会有显著的提升。今天我们要讲解的 CompletionService 就是用来做这个事情的。CompletionService 可以按照执行完成结果的到场顺序，被主线程获取到，从而继续执行后面逻辑。

1、了解 CompletionService

了解一个类最好、最快的方法就是阅读源代码的注解。而大多数人通常的做法却是去百度或者 google。这样有两个弊端，一是效率并不一定高，可能搜出来很多无用的内容。二是看到的文章并不权威，甚至可能是错的。有的同学可能觉得英文阅读费劲，其实作为开发人员，英语阅读已经是必备技能。这就如同你要熟知 IDE 的快捷键一样，所以如果觉得英文阅读困难，可以刻意练习。其实多读一些技术文档，会发现用词基本都是类似的。

扯的有点远，我们收回来，先看看源代码中对 CompletionService 的解释：

对异步任务执行和执行结果消费解藕。生产者提交任务执行。消费者则获取完成的任务，然后按照完成任务的顺序对任务结果进行处理。

官方的解释是不是十分简洁明了？

2、使用 CompletionService

下面我们使用 CompletionService 实现一个吃苹果的程序。首先我声明一个流，里面是一些水果，每个水果会对应一个洗干净的任务。然后主线程拿到洗干净的水果再一个个吃掉。代码如下：

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
    ExecutorService pool = Executors.newFixedThreadPool(5);
    CompletionService<String> service = new ExecutorCompletionService<String>(pool);

  Stream.of("苹果", "梨", "葡萄", "桃")
          .forEach(fruit -> service.submit(() -> {
                    if(fruit.equals("苹果")){
                        TimeUnit.SECONDS.sleep(6);
                    }else if(fruit.equals("梨")){
                        TimeUnit.SECONDS.sleep(1);
                    }else if(fruit.equals("葡萄")){
                        TimeUnit.SECONDS.sleep(10);
                    }else if(fruit.equals("桃")){
                        TimeUnit.SECONDS.sleep(3);
                    }
                    return "洗干净的"+fruit;
                }))
          );

    String result;
    while((result=service.take().get())!=null){
        System.out.println("吃掉"+result);
    }
}
```

可以看到有四种水果。会为每个水果启一个洗水果的任务。每种水果洗的时间不同，其中葡萄最不好洗要 10 秒，而梨最好洗，只需要 1 秒。等待水果洗好后，主线程通过 service.take () 取得执行完成的 Future，然后从里面 get 出返回值，把洗干净的水果吃掉。

我们可以看到输出如下：

```
吃掉洗干净的梨
吃掉洗干净的桃
吃掉洗干净的苹果
吃掉洗干净的葡萄
```

可以看到哪个水果先洗干净就会先被吃掉。这也证明了 service.take () 的顺序是任务的完成顺序，而不是任务提交的顺序。

通过 CompletionService 我们就可以一端生产，另一端按照完成的顺序进行消费。这避免提交大量任务时，不知道哪个任务先完成，从而在调用 Future 的 get 方法时产生阻塞。使用 CompletionService，永远都是完成一个返回一个，然后消费一个。这样你的程序才更为高效。

主线程收到返回后，可以再继续使用 CompletionService 来异步执行下一步的逻辑，这和非阻塞的编程方式异曲同工。

3、CompletionService 源码分析

3.1 CompletionService 构造方法

我们先看如何初始化 CompletionService：

```
ExecutorService pool = Executors.newFixedThreadPool(5);
CompletionService<String> service = new ExecutorCompletionService<String>(pool);
```

首先初始化 ExecutorService，在构造 ExecutorCompletionService 时作为参数传入。其实 CompletionService 对任务的执行其实就是借助于 ExecutorService 来完成的。接下来我们进入它的构造函数：

```
public ExecutorCompletionService(Executor executor) {
    if (executor == null)
        throw new NullPointerException();
    this.executor = executor;
    this.aes = (executor instanceof AbstractExecutorService) ?
        (AbstractExecutorService) executor : null;
    this.completionQueue = new LinkedBlockingQueue<Future<V>>();
}
```

构造函数中构造了三个属性：

```
private final Executor executor;
private final AbstractExecutorService aes;
private final BlockingQueue<Future<V>> completionQueue;
```

executor 就是你传入的 ExecutorService，用来执行任务。

aes 的作用是创建新的 task。它的初始化过程比较有意思，判断了是否为 AbstractExecutorService 的实例。至于为什么这么做，我们后面再详细讲解。

completionQueue 是一个存放 Future 的阻塞队列，并且是无界的。这意味着如果源头不断的产生 Future，但是没有去消费，就会造成内存泄漏。

executor 执行完成的 Future 会被放入 completionQueue 中，take 方法将会从

completionQueue 中取得最新的 future 对象（最近执行完的 task 的结果）。

3.2 CompletionService 的 submit 方法

```
public Future<V> submit(Callable<V> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<V> f = newTaskFor(task);
    executor.execute(new QueueingFuture(f));
    return f;
}
```

首先将 Callable 类型的 task 转为 RunnableFuture 类型。RunnableFuture 是个接口，FutureTask 是其一种实现。

然后通过 new QueueingFuture (f)，再将 RunnableFuture 包装为 QueueingFuture 类型的对象。QueueingFuture 的作用就是在 Future 完成时，加入到 completionQueue 中。

我们先看 newTaskFor 的源码：

```
private RunnableFuture<V> newTaskFor(Callable<V> task) {
    if (aes == null)
        return new FutureTask<V>(task);
    else
        return aes.newTaskFor(task);
}
```

如果 aes 为空，那么直接 new FutureTask。如果不为空则调用 aes 的 newTaskFor 方法。什么情况 aes 会为空呢？我们再看下 aes 初始化的代码：

```
this.aes = (executor instanceof AbstractExecutorService) ?
(AbstractExecutorService) executor : null;
```

当传入的 executor 为 AbstractExecutorService 类型时，那么 aes 不为空。否则 aes 为空。这两处逻辑处理是相关联的，这么做的原因如下：

1、如果 executor 是 AbstractExecutorService 的子类，有可能会重写 newTaskFor 方法，所以这里优先使用 executo r 的方法来创建 Task，这样后面通过 executor 执行 task 才能正确。比如 ForkJoinPool 就对 newTaskFor 方法进行了重写；

2、如果 executor 不是继承自 AbstractExecutorService。那么它可能并没有 newTaskFor 方法。所以需要 CompletionService 自己来创建 FutureTask。

这样看来 aes 的存在，只是为了尽量使用 executor 提供的 newTaskFor 方法来创建 task，以使后面 excute 方法能够正常运行。

接下来我们分析 QueueingFuture 方法：

```
private class QueueingFuture extends FutureTask<Void> {
    QueueingFuture(RunnableFuture<V> task) {
        super(task, null);
        this.task = task;
    }
    protected void done() { completionQueue.add(task); }
    private final Future<V> task;
}
```

QueueingFuture 是内部静态类，并且是 FutureTask 的子类。他只是重写了 done 方法。大家回忆上一节对 Future 的分析，应该还记得 done 方法在任务执行结果返回后被调用，但是留给子类来实。这里就用上了这个特性。done 方法里面做的就是把 task 加入阻塞队列中。这意味着，先完成的 task 会先把自己的 Future 放入队列中。那么当然也会被 take 方法先取到。而由于是阻塞队列，所以 take 方法取不到 task 时，就会阻塞。但由于能被 take 到的 task 肯定已经有了返回值，所以调用 task 的 get 方法时就不会再次阻塞了。也就是说 client 代码中的下面一行只会在 take 时发生阻塞：

```
while((result=service.take().get())!=null){
    System.out.println("吃掉"+result);
}
```

executor 执行任务的代码就不用再次分析了，这在之前学习 Executor 的时候已经详细分析过了。submit 方法分析完后我们再来看看 take 方法。

3.3 CompletionService 的 take 方法

相比较 submit 方法，take 方法就更为简单了，如下：

```
public Future<V> take() throws InterruptedException {
    return completionQueue.take();
}
```

只有一行代码，就是从 completionQueue 中取得 Futrue 对象。由于 completionQueue 是阻塞队列，当没有 Future 时，就会阻塞在此。而 completionQueue 中保存 Future 的顺序是完成顺序。

4、 总结

CompletionService 给我们提供了一种非阻塞的异步执行方式。让程序更为高效。他的实现非常的简单和巧妙，值得我们借鉴。其实我们学习到这里，不知道你是否有这种体会，这些工具实际上就是我们之前学习内容的组合运用，如果前面你掌握的很牢固，学习起来一点也不费劲。如果前面就似懂非懂，那么就会越看越糊涂。其实我们在学习上至少有一半的时间都是在打基础，但这个过程必不可少，并且受益更为深远。

}

31 凭票取餐—Future模式详解

33 分阶段执行你的任务-学习使用 Phaser运行多阶段任务