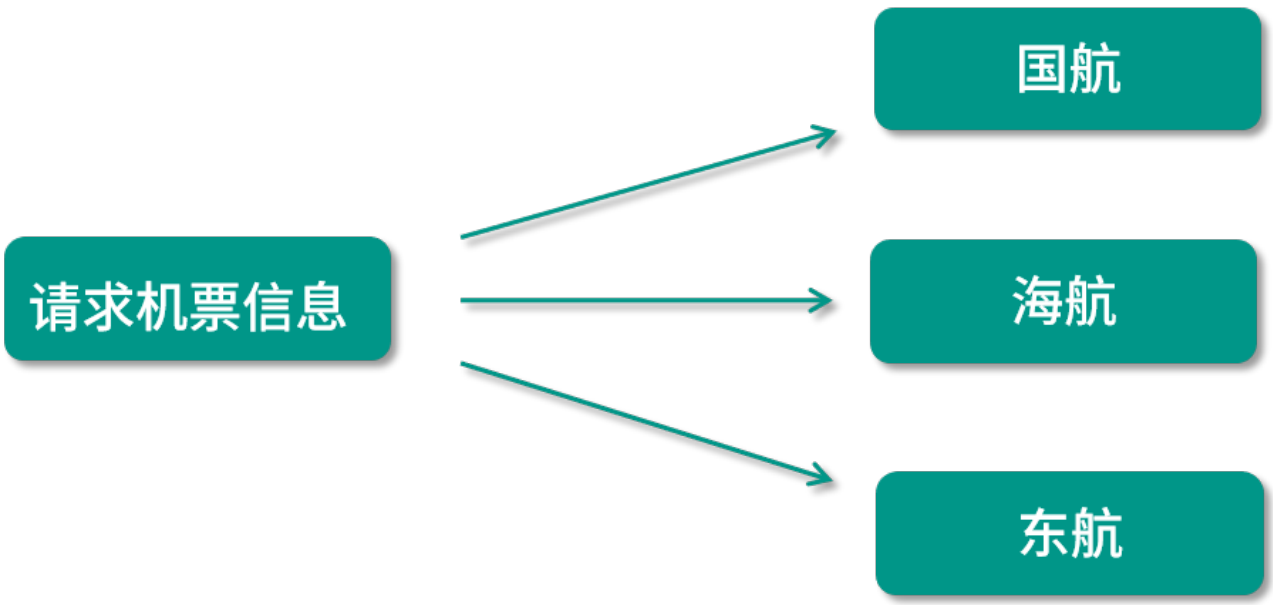
二

51 如何利用 CompletableFuture 实现“旅游平台”问题?

本课时我们主要讲解如何利用 CompletableFuture 实现旅游平台问题。

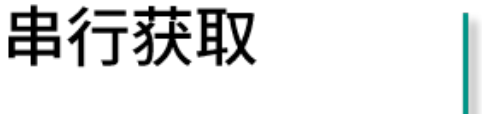
旅游平台问题

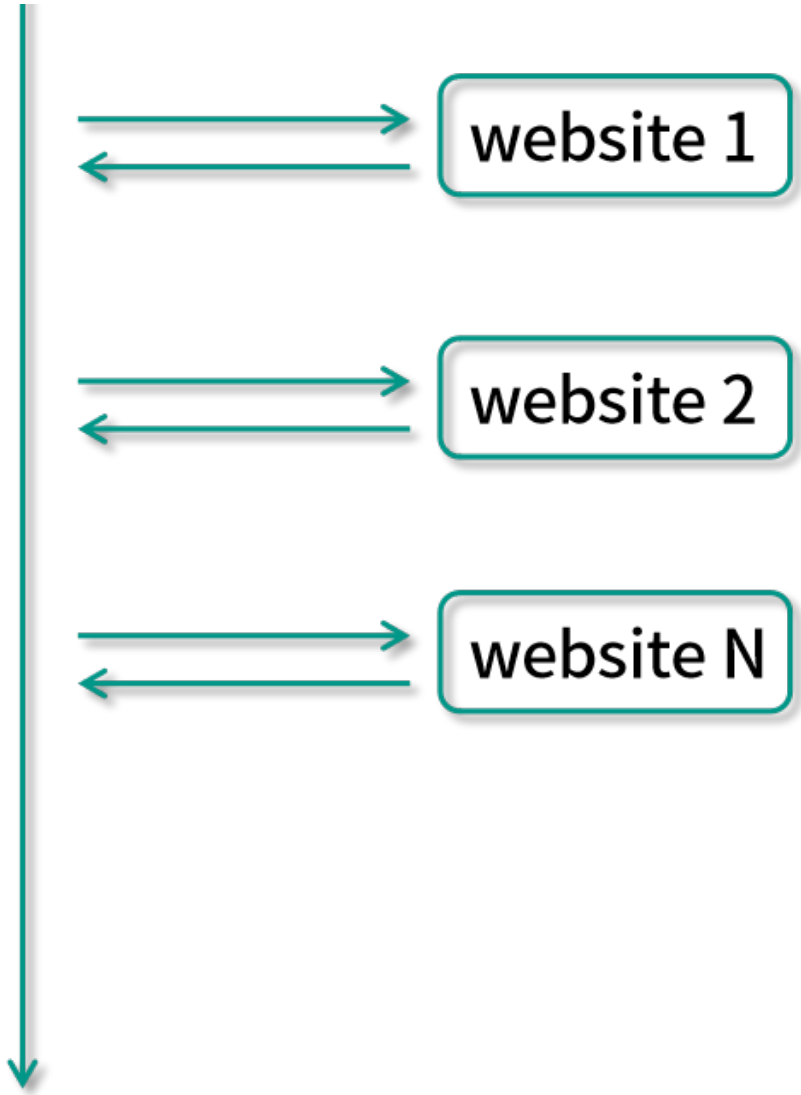
什么是旅游平台问题呢？如果想要搭建一个旅游平台，经常会有这样的需求，那就是用户想同时获取多家航空公司的航班信息。比如，从北京到上海的机票钱是多少？有很多家航空公司都有这样的航班信息，所以应该把所有航空公司的航班、票价等信息都获取到，然后再聚合。由于每个航空公司都有自己的服务器，所以分别去请求它们的服务器就可以了，比如请求国航、海航、东航等，如下图所示：



串行

一种比较原始的方式是用串行的方式来解决这个问题。



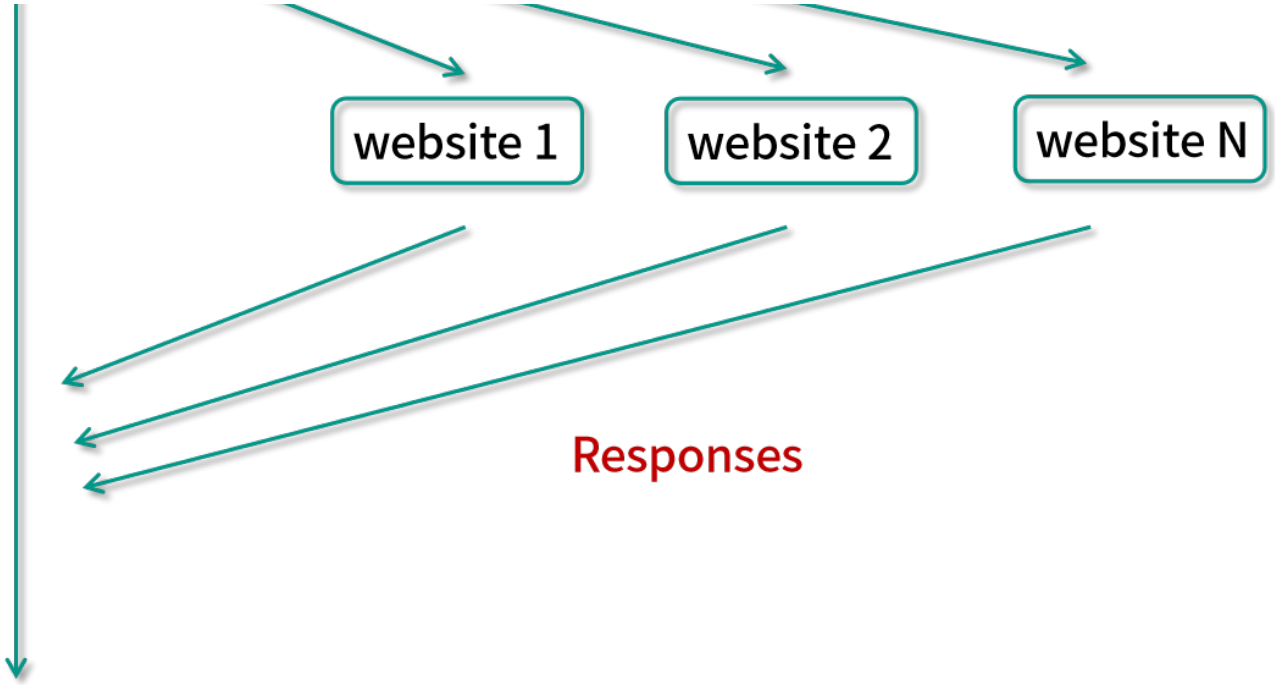


比如我们想获取价格，要先去访问国航，在这里叫作 website 1，然后再去访问海航 website 2，以此类推。当每一个请求发出去之后，等它响应回来以后，我们才能去请求下一个网站，这就是串行的方式。

这样做的效率非常低下，比如航空公司比较多，假设每个航空公司都需要 1 秒钟的话，那么用户肯定等不及，所以这种方式是不可取的。

并行

接下来我们就对刚才的思路进行改进，最主要的思路就是把串行改成并行，如下图所示：

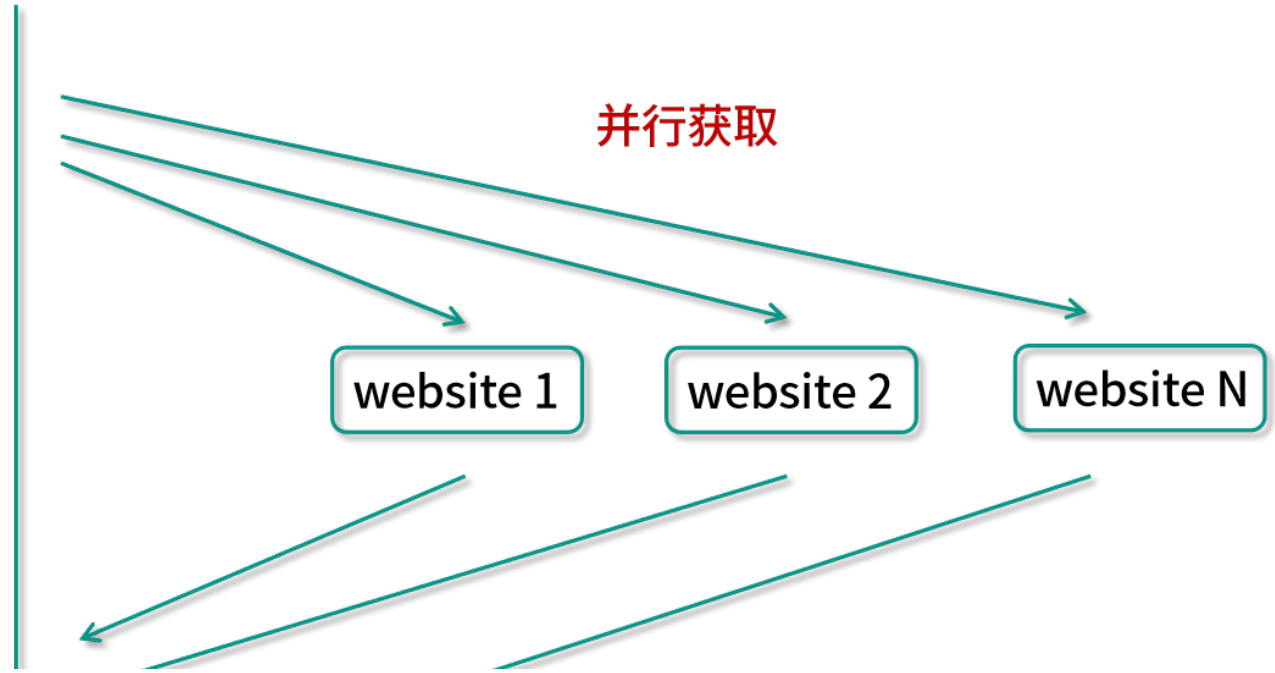


我们可以并行地去获取这些机票信息，然后再把机票信息给聚合起来，这样的话，效率会成倍的提高。

这种并行虽然提高了效率，但也有一个缺点，那就是会“一直等到所有请求都返回”。如果有一个网站特别慢，那么你不应该被那个网站拖累，比如说某个网站打开需要二十秒，那肯定是等不了这么长时间的，所以我们需要一个功能，那就是有超时的获取。

有超时的并行获取

下面我们就来看看下面这种有超时的并行获取的情况。



在这种情况下，就属于有超时的并行获取，同样也在并行的去请求各个网站信息。但是我们规定了一个时间的超时，比如 3 秒钟，那么到 3 秒钟的时候如果都已经返回了那当然最好，把它们收集起来即可；但是如果还有些网站没能及时返回，我们就把这些请求给忽略掉，这样一来用户体验就比较好了，它最多只需要等固定的 3 秒钟就能拿到信息，虽然拿到的可能不是最全的，但是总比一直等更好。

想要实现这个目标有几种实现方案，我们一个一个的来看看。

线程池的实现

第一个实现方案是用线程池，我们来看一下代码。

```
public class ThreadPoolDemo {

    ExecutorService threadPool = Executors.newFixedThreadPool(3);

    public static void main(String[] args) throws InterruptedException {

        ThreadPoolDemo threadPoolDemo = new ThreadPoolDemo();

        System.out.println(threadPoolDemo.getPrices());

    }

    private Set<Integer> getPrices() throws InterruptedException {

        Set<Integer> prices = Collections.synchronizedSet(new HashSet<Integer>());

        threadPool.submit(new Task(123, prices));

        threadPool.submit(new Task(456, prices));

        threadPool.submit(new Task(789, prices));

        Thread.sleep(3000);

        return prices;

    }

    private class Task implements Runnable {
```

```
        Integer productId;
        Set<Integer> prices;
        public Task(Integer productId, Set<Integer> prices) {
            this.productId = productId;
            this.prices = prices;
        }
        @Override
        public void run() {
            int price=0;
            try {
                Thread.sleep((long) (Math.random() * 4000));
                price= (int) (Math.random() * 4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            prices.add(price);
        }
    }
}
```

在代码中，新建了一个线程安全的 Set，它是用来存储各个价格信息的，把它命名为 Prices，然后往线程池中去放任务。线程池是在类的最开始时创建的，是一个固定 3 线程的线程池。而这个任务在下方的 Task 类中进行了描述，在这个 Task 中我们看到有 run 方法，在该方法里面，我们用一个随机的时间去模拟各个航空网站的响应时间，然后再去返回一个随机的价格来表示票价，最后把这个票价放到 Set 中。这就是我们 run 方法所做的事情。

再回到 getPrices 函数中，我们新建了三个任务，productId 分别是 123、456、789，这里的 productId 并不重要，因为我们返回的价格是随机的，为了实现超时等待的功能，在这里调用了 Thread 的 sleep 方法来休眠 3 秒钟，这样做的话，它就会在这里等待 3 秒，之后直接返回 prices。

此时，如果前面响应速度快的话，prices 里面最多会有三个值，但是如果每一个响应时间都很慢，那么可能 prices 里面一个值都没有。不论你有多少个，它都会在休眠结束之后，也就是执行完 Thread 的 sleep 之后直接把 prices 返回，并且最终在 main 函数中把这个结果给打印出来。

我们来看一下可能的执行结果，一种可能性就是有 3 个值，即 [3815, 3609, 3819]（数字是随机的）；有可能是 1 个 [3496]、或 2 个 [1701, 2730]，如果每一个响应速度都特别慢，可能一个值都没有。

这就是用线程池去实现的最基础的方案。

CountDownLatch

在这里会有一个优化的空间，比如说网络特别好时，每个航空公司响应速度都特别快，你根本不需要等三秒，有的航空公司可能几百毫秒就返回了，那么我们也不应该让用户等 3 秒。所以需要进行一下这样的改进，看下面这段代码：

```
public class CountDownLatchDemo {

    ExecutorService threadPool = Executors.newFixedThreadPool(3);

    public static void main(String[] args) throws InterruptedException {

        CountDownLatchDemo countDownLatchDemo = new CountDownLatchDemo();

        System.out.println(countDownLatchDemo.getPrices());

    }

    private Set<Integer> getPrices() throws InterruptedException {

        Set<Integer> prices = Collections.synchronizedSet(new HashSet<Integer>());

        CountDownLatch countDownLatch = new CountDownLatch(3);

        threadPool.submit(new Task(123, prices, countDownLatch));

        threadPool.submit(new Task(456, prices, countDownLatch));

        threadPool.submit(new Task(789, prices, countDownLatch));

        countDownLatch.await(3, TimeUnit.SECONDS);

        return prices;

    }

    private class Task implements Runnable {
```

```
        Integer productId;
        Set<Integer> prices;
        CountDownLatch countDownLatch;
        public Task(Integer productId, Set<Integer> prices,
                CountDownLatch countDownLatch) {
            this.productId = productId;
            this.prices = prices;
            this.countDownLatch = countDownLatch;
        }
        @Override
        public void run() {
            int price = 0;
            try {
                Thread.sleep((long) (Math.random() * 4000));
                price = (int) (Math.random() * 4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            prices.add(price);
            countDownLatch.countDown();
        }
    }
}
```

这段代码使用 CountDownLatch 实现了这个功能，整体思路和之前是一致的，不同点在于我们新增了一个 CountDownLatch，并且把它传入到了 Task 中。在 Task 中，获取完机票信息并且把它添加到 Set 之后，会调用 countDown 方法，相当于把计数减 1。

这样一来，在执行 countDownLatch.await(3, TimeUnit.SECONDS) 这个函数进行等待时，如果三个任务都非常快速地执行完毕了，那么三个线程都已经执行了 countDown 方法，那么这个 await 方法就会立刻返回，不需要傻等到 3 秒钟。

如果有一个请求特别慢，相当于有一个线程没有执行 countDown 方法，来不及在 3 秒钟之内执行完毕，那么这个带超时参数的 await 方法也会在 3 秒钟到了以后，及时地放弃这一次等待，于是就把 prices 给返回了。所以这样一来，我们就利用 CountDownLatch 实现了这个需求，也就是说我们最多等 3 秒钟，但如果在 3 秒之内全都返回了，我们也可以快速地去返回，不会傻等，提高了效率。

CompletableFuture

我们再来看一下用 CompletableFuture 来实现这个功能的用法，代码如下所示：

```
public class CompletableFutureDemo {

    public static void main(String[] args)

            throws Exception {

        CompletableFutureDemo completableFutureDemo = new CompletableFutureDemo();

        System.out.println(completableFutureDemo.getPrices());

    }

    private Set<Integer> getPrices() {

        Set<Integer> prices = Collections.synchronizedSet(new HashSet<Integer>());

        CompletableFuture<Void> task1 = CompletableFuture.runAsync(new Task(123, pr

        CompletableFuture<Void> task2 = CompletableFuture.runAsync(new Task(456, pr

        CompletableFuture<Void> task3 = CompletableFuture.runAsync(new Task(789, pr

        CompletableFuture<Void> allTasks = CompletableFuture.allOf(task1, task2, ta

        try {

            allTasks.get(3, TimeUnit.SECONDS);

        } catch (InterruptedException e) {

        } catch (ExecutionException e) {

        } catch (TimeoutException e) {

        }

        return prices;

    }

    private class Task implements Runnable {
```

```
        Integer productId;
        Set<Integer> prices;
        public Task(Integer productId, Set<Integer> prices) {
            this.productId = productId;
            this.prices = prices;
        }
        @Override
        public void run() {
            int price = 0;
            try {
                Thread.sleep((long) (Math.random() * 4000));
                price = (int) (Math.random() * 4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            prices.add(price);
        }
    }
}
```

这里我们不再使用线程池了，我们看到 getPrices 方法，在这个方法中，我们用了 CompletableFuture 的 runAsync 方法，这个方法会异步的去执行任务。

我们有三个任务，并且在执行这个代码之后会分别返回一个 CompletableFuture 对象，我们把它们命名为 task 1、task 2、task 3，然后执行 CompletableFuture 的 allOf 方法，并且把 task 1、task 2、task 3 传入。这个方法的作用是把多个 task 汇总，然后可以根据需要去获取到传入参数的这些 task 的返回结果，或者等待它们都执行完毕等。我们就把这个返回值叫作 allTasks，并且在下面调用它的带超时时间的 get 方法，同时传入 3 秒钟的超时参数。

这样一来它的效果就是，如果在 3 秒钟之内这 3 个任务都可以顺利返回，也就是这个任务包括的那三个任务，每一个都执行完毕的话，则这个 get 方法就可以及时正常返回，并且往

下执行，相当于执行到 return prices。在下面的这个 Task 的 run 方法中，该方法如果执行完毕的话，对于 CompletableFuture 而言就意味着这个任务结束，它是以这个作为标记来判断任务是不是执行完毕的。但是如果有某一个任务没能来得及在 3 秒钟之内返回，那么这个带超时参数的 get 方法便会抛出 TimeoutException 异常，同样会被我们给 catch 住。这样一来它就实现了这样的效果：会尝试等待所有的任务完成，但是最多只会等 3 秒钟，在此之间，如及时完成则及时返回。那么所以我们利用 CompletableFuture，同样也可以解决旅游平台的问题。它的运行结果也和之前是一样的，有多种可能性。

最后做一下总结。在本课时中，我们先给出了一个旅游平台问题，它需要获取各航空公司的机票信息，随后进行了代码演进，从串行到并行，再到有超时的并行，最后到不仅有超时的并行，而且如果大家速度都很快，那么也不需要一直等到超时时间到，我们进行了这样的一步一步的迭代。

当然除了这几种实现方案之外，还会有其他的实现方案，你能想到哪些实现方案呢？不妨在下方留言告诉我，谢谢。

上一页 下一页