Future 掌控未来

Callable 和 Runnable 的不同：

Runnable 接口的定义：

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

Runnable 的缺陷：

1. 不会有返回值
2. 不能抛出Checked Exception

为什么Runnable有这样的缺陷?

答：因为java就是这样写的，改不了，想修复以上缺陷使用Callable 。

Callable接口这么强大，那Callable是不是用来代替Runnable的?

答：不是的，它们并存，例如Thread类初始化时不接受Callable作为参数。

```
public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum(), 0);
    }
```

想用Callable类型的话怎么办呢，有个FutureTask可以提供帮助，先看看它的部分源码：？？？

最后总结一下 Callable 和 Runnable 的不同之处：

- **方法名**，Callable 规定的执行方法是 call()，而 Runnable 规定的执行方法是 run()；
- **返回值**，Callable 的任务执行后有返回值，而 Runnable 的任务执行后是没有返回值的；
- **抛出异常**，call() 方法可抛出异常，而 run() 方法是不能抛出受检查异常的；
- 和 Callable 配合的有一个 Future 类，通过 Future 可以了解任务执行情况，或者取消任务的执行，还可获取任务执行的结果，这些功能都是 Runnable 做不到的，Callable 的功能要比 Runnable 强大。

Future 的5 个方法：

- **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：

Future 相当于一个存储器，它存储了 Callable 的 call 方法的任务结果。

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
}
```

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。

- ## isDone() 方法：判断是否执行完毕

这个方法如果返回 true 则代表执行完成了（注意：如果任务执行到一半抛出了异常也算是执行完成，所以返回也是true）；如果返回 false 则代表还没完成。

- ## cancel 方法：取消任务的执行

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：

1. 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。
2. 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。
3. 这个任务正在执行，需根据mayInterruptIfRunning参数判断是否取消，如果该参数传入的是 true，则中断处理， cancel 方法返回 true，该参数传入的是 false，则不中断处理， cancel 方法返回 false。
    - 那么如何选择传入 true 还是 false 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。

- **isCancelled() 方法：判断是否被取消**

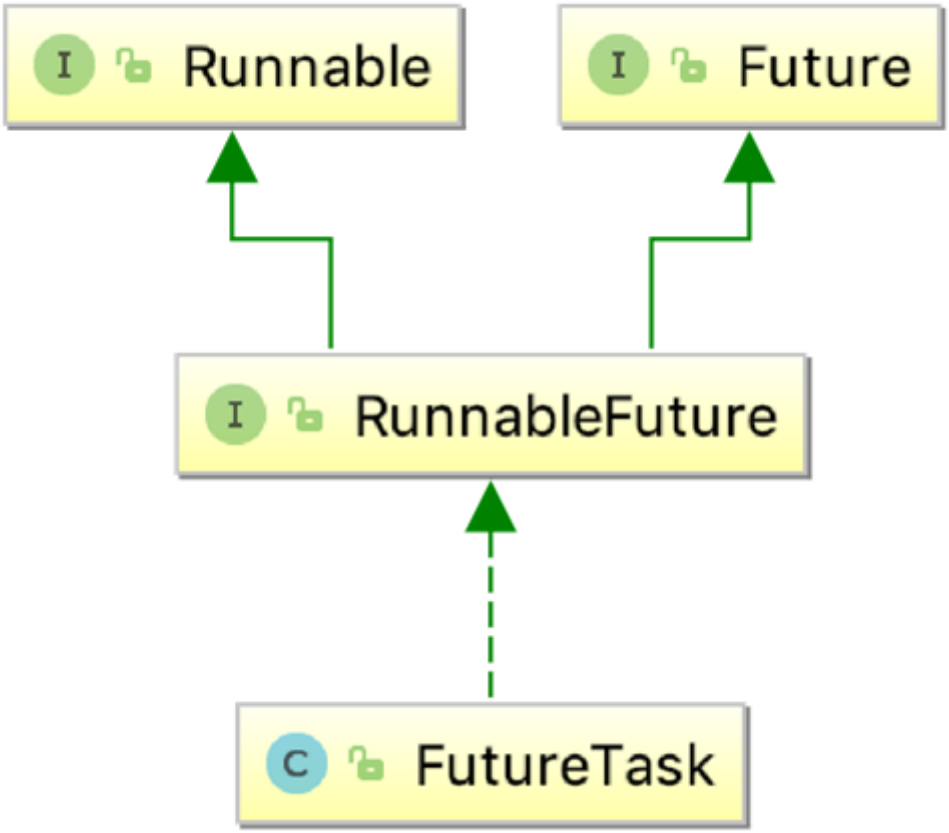
最后一个方法是 isCancelled 方法，判断是否被取消，它和 cancel 方法配合使用，比较简单。

Future的使用示例：

```
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。

FutureTask 和Future、 Runable关系：



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。

用 FutureTask 来创建 Future：

```
1  public class FutureTaskDemo {
2
3      public static void main(String[] args) {
4          Task task = new Task();
5          FutureTask<Integer> integerFutureTask = new FutureTask<>(task);
6          new Thread(integerFutureTask).start();
7          try {
8              System.out.println("task运行结果："+integerFutureTask.get());
9          } catch (InterruptedException e) {
10             e.printStackTrace();
11         } catch (ExecutionException e) {
12             e.printStackTrace();
13         }
14     }
15 }
16
17 class Task implements Callable<Integer> {
18
19     @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。

使用 Future 有哪些注意点？Future 产生新的线程了吗？

使用 Future 的注意点：

- **get 方法容易 block：**

```
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";
        }
    }
}
```

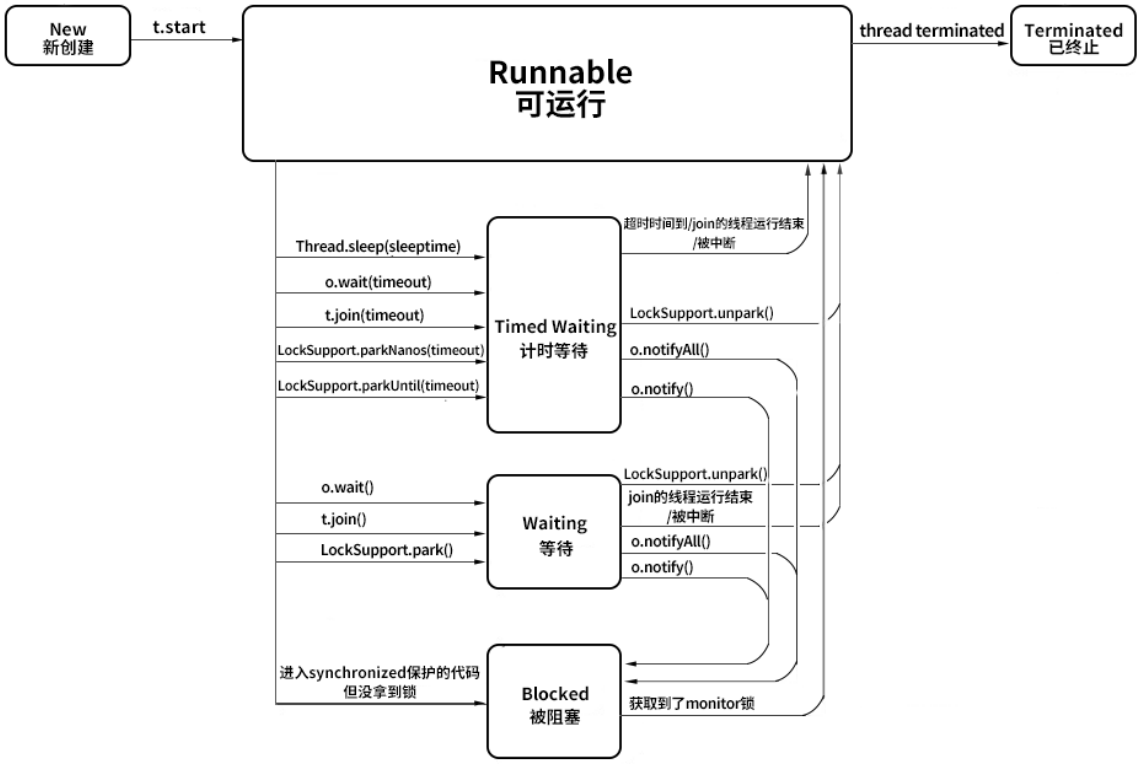
```
速度慢的任务
速度慢的任务
速度快的任务
速度快的任务
```

如上代码输出的结果，线程是按顺序执行的，先执行两个慢任务，如果慢任务等待时间很长的话那么线程一直处于阻塞状态，于是在调用 get 方法时，应该使用 timeout 来限制【使用 get(long timeout, TimeUnit unit) 而不是get()】，如果超时便会抛出一个 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，这样程序就不会一直卡着了。

- **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。

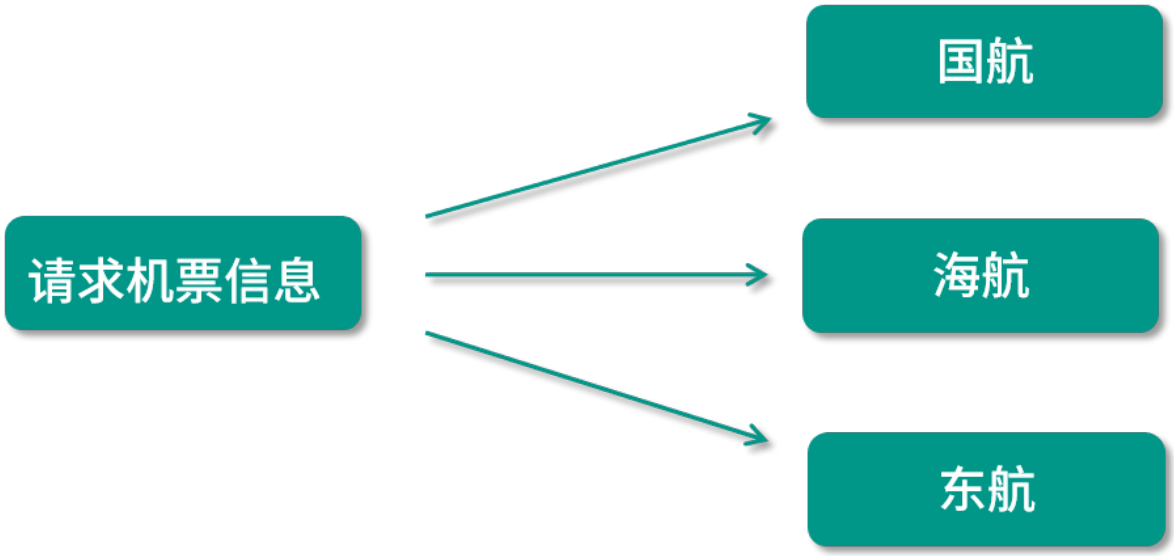


Future 产生新的线程了吗?

答：没有，它们需要借助其他的比如 Thread 类或者线程池才能执行任务。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。

利用 CompletableFuture 实现“旅游平台”问题：

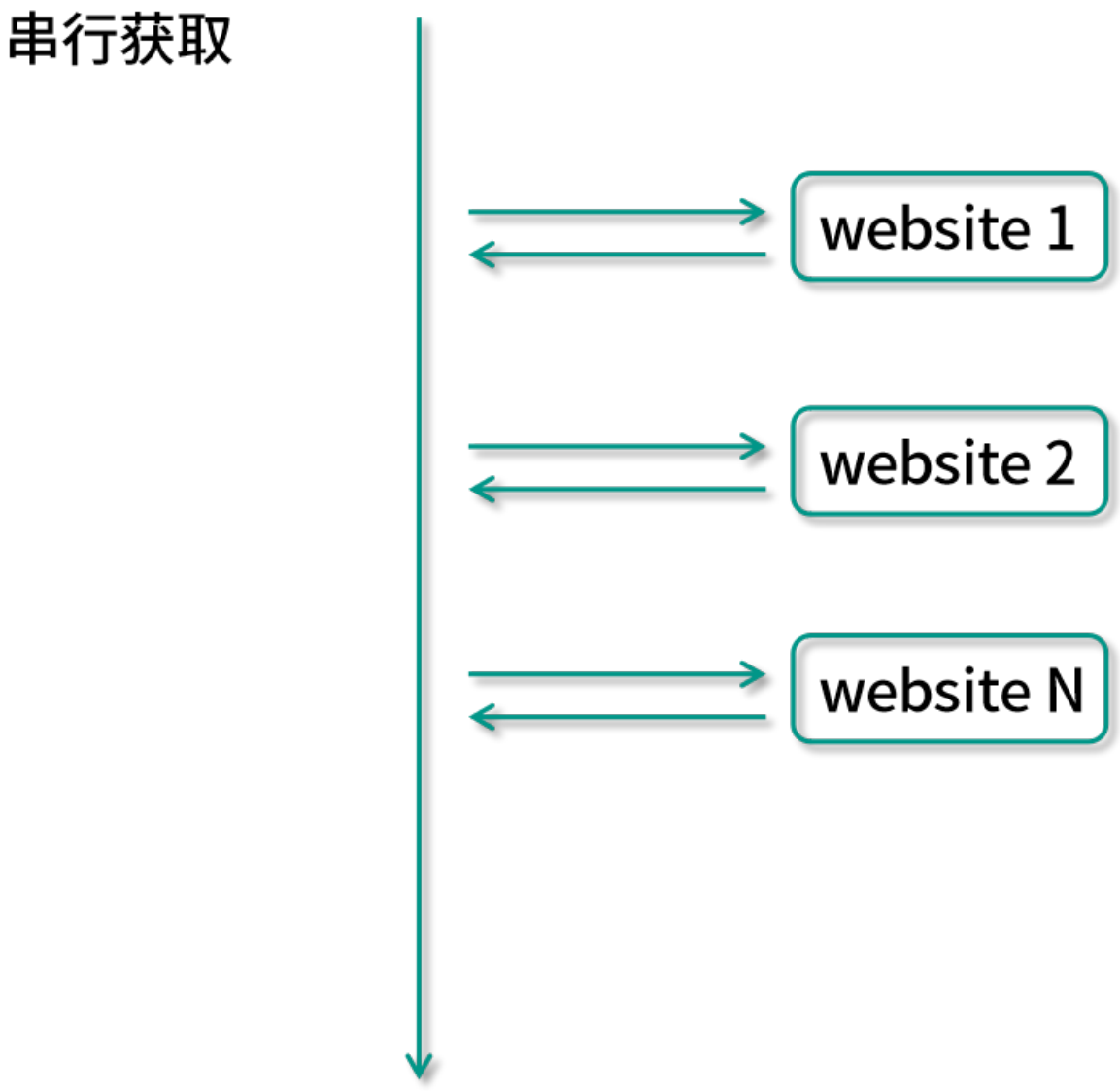
什么是旅游平台问题呢？如果想要搭建一个旅游平台，经常会有这样的需求，那就是用户想同时获取多家航空公司的航班信息。比如，从北京到上海的机票钱是多少？有很多家航空公司都有这样的航班信息，所以应该把所有航空公司的航班、票价等信息都获取到，然后再聚合。由于每个航空公司都有自己的服务器，所以分别去请求它们的服务器就可以了，比如请求国航、海航、东航等，如下图所示：



实现思路：

- **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。

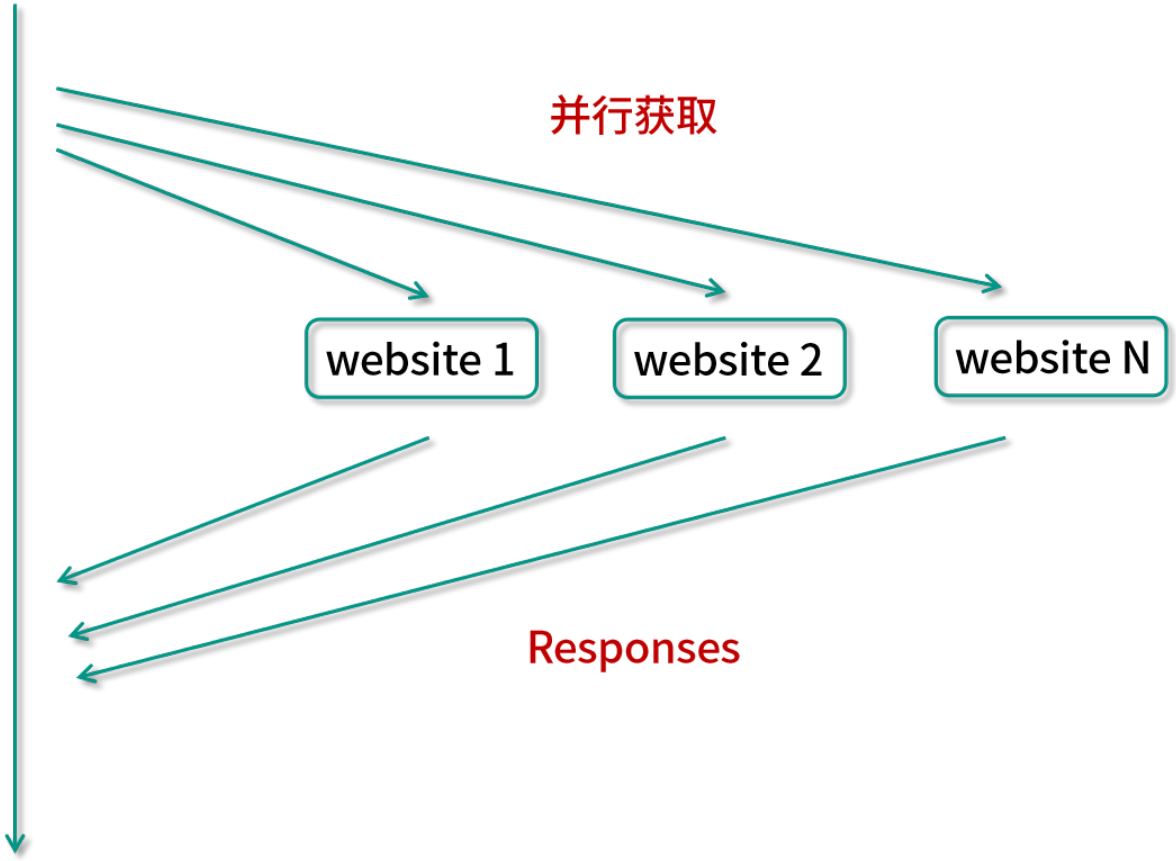


比如我们想获取价格，要先去访问国航，在这里叫作 website 1，然后再去访问海航 website 2，以此类推。当每一个请求发出去之后，等它响应回来以后，我们才能去请求下一个网站，这就是串行的方式。

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：

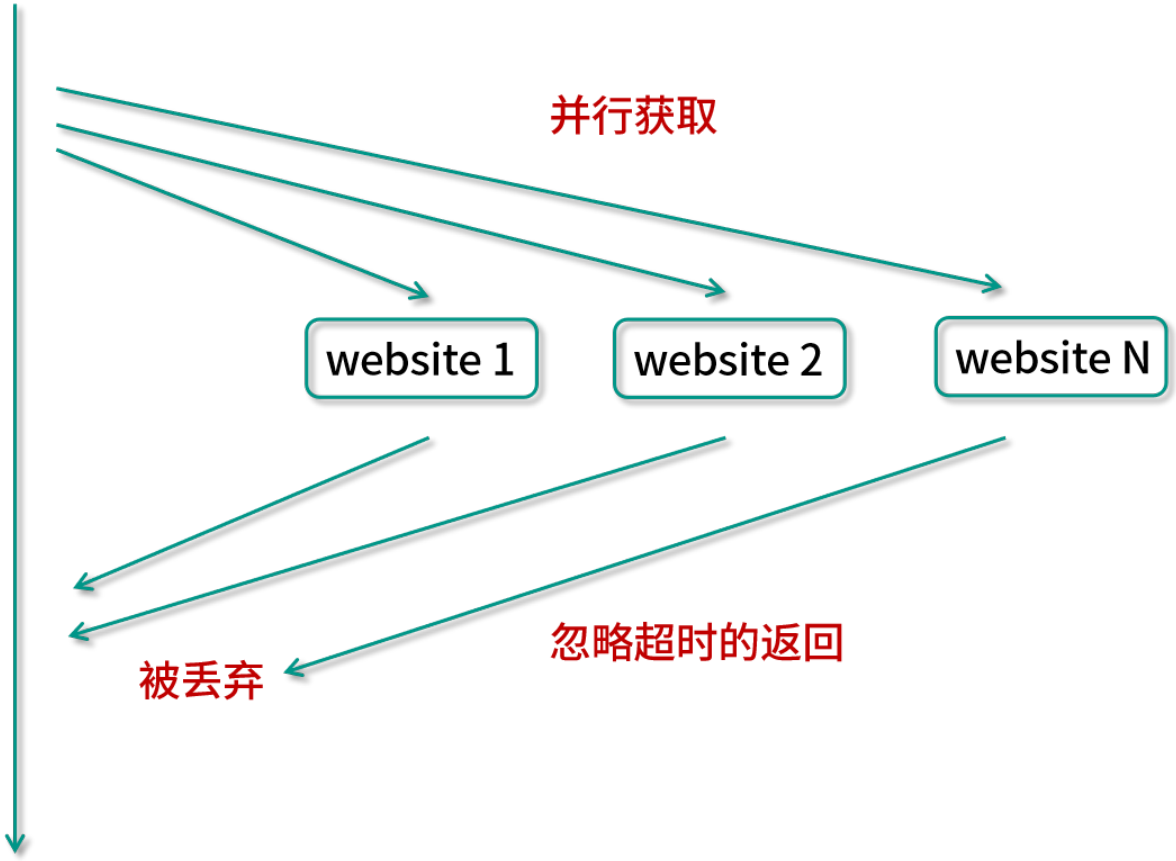


我们可以并行地去获取这些机票信息，然后再把机票信息给聚合起来，这样的话，效率会成倍的提高。

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（最佳方案）：**

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。

实现方案：

1. **使用线程池实现：**

```
1  public class ThreadPoolDemo {
2
3      ExecutorService threadPool = Executors.newFixedThreadPool(3);
4
5      public static void main(String[] args) throws InterruptedException {
6          ThreadPoolDemo threadPoolDemo = new ThreadPoolDemo();
7          System.out.println(threadPoolDemo.getPrices());
8      }
9
10     private Set<Integer> getPrices() throws InterruptedException {
11         Set<Integer> prices = Collections.synchronizedSet(new HashSet<Integer>
   ());
12         threadPool.submit(new Task(123, prices));
13         threadPool.submit(new Task(456, prices));
14         threadPool.submit(new Task(789, prices));
15         Thread.sleep(3000);
16         return prices;
17     }
18
19     private class Task implements Runnable {
20
21         Integer productId;
22         Set<Integer> prices;
23
24         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。

2. **使用CountDownLatch实现：**

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。

3. **使用CompletableFuture实现：**

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, prices));
        CompletableFuture<Void> task2 = CompletableFuture.runAsync(new Task(456, prices));
        CompletableFuture<Void> task3 = CompletableFuture.runAsync(new Task(789, prices));

        CompletableFuture<Void> allTasks = CompletableFuture.allOf(task1, task2, task3);
        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。