实战分析Java的异步编程，并通过CompletableFuture进行高效调优

一、写在开头

在我们一开始讲多线程的时候，提到过**异步**与**同步**的概念，这里面我们再回顾一下：

* **同步**：调用方在调用某个方法后，等待被调用方返回结果；调用方在取得被调用方的返回值后，再继续运行。调用方顺序执行，同步等待被调用方的返回值，这就是阻塞式调用；
* **异步**：调用方在调用某个方法后，直接返回，不需要等待被调用方返回结果；被调用方开启一个线程处理任务，调用方可以同时去处理其他工作。调用方和被调用方是异步的，这就是非阻塞式调用。

`适应场景` **同步**：如果数据存在线程间的共享，或竞态条件，需要同步。如多个线程同时对同一个变量进行读和写的操作，必须等前一个请求完成，后一个请求去调用前一个请求的结果，这时候就只能采用同步方式。 **异步**：当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就可以使用异步，提高效率、加快程序的响应。

而我们今天探讨的话题就是Java中的异步编程。

二、Future

为了提升Java程序的响应速度，在JDK1.5时引入了JUC包，里面包含了一个接口文件：Future，这是Java中实现异步编程的开端，我们可以将Future理解为一种异步思想或者一种设计模式；当我们执行某一耗时的任务时，可以将这个耗时任务交给一个子线程去异步执行，同时我们可以干点其他事情，不用傻傻等待耗时任务执行完成。等我们的事情干完后，我们再通过 `Future` 类获取到耗时任务的执行结果。

它的底层也是几个很容易理解的接口方法：

```
// V 代表了Future执行的任务返回值的类型
public interface Future<V> {
    // 取消任务执行
    // 成功取消返回 true，否则返回 false
    boolean cancel(boolean mayInterruptIfRunning);
    // 判断任务是否被取消
    boolean isCancelled();
    // 判断任务是否已经执行完成
    boolean isDone();
    // 获取任务执行结果
    V get() throws InterruptedException, ExecutionException;
    // 指定时间内没有返回计算结果就抛出 TimeOutException 异常
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutExceptio
 }
```

这些接口大致提供的服务是：我有一个任务分配给了Future，然后我可以继续去干其他的事情，然后我可以在这个过程中去看任务是否完成，也可以取消任务，一段时间后我也可以去获取到任务执行后的结果，也可以设置任务多久执行完，没执行完抛异常等。

对于Future的使用，我想大家应该并不陌生的，我们在学习线程池的时候就有涉及，看下面这个测试案例：

```
//这里使用Executors只是方便测试，正常使用时推荐使用ThreadPoolExecutor!
ExecutorService executorService = Executors.newFixedThreadPool(3);
Future<String> submit = executorService.submit(() -> {
    try {
        Thread.sleep(5000L);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "javabuild";
});
String s = submit.get();
System.out.println(s);
executorService.shutdown();
```

这里我们通过executorService.submit()方法去提交一个任务，线程池会返回一个 Future 类型的对象，通过这个 Future 对象可以判断任务是否执行成功，并且可以通过 Future 的 get()方法来获取返回值。

三、Future实战

经过了上面的学习了解，我们来根据案例场景进行实战使用Future，毕竟现在很多大厂除了问面试八股文之外，更多的会涉及到场景题！

`场景模拟`

> **假如你是一个12306的开发人员，为了在节假日满足大量用户的出行需要，请高效的完成：用户搜索一个目的地，推荐出所有的交通方案+酒店+耗时，并根据价格从低到高排序**

拿到这种场景题的时候，我们往往需要分步处理：

1. 根据目的地，搜索出所有的飞机、火车、客车路线，每个路线间隔30分钟；
2. 计算出每种路线的耗时；
3. 根据交通方案中最后一个到站点进行可用酒店匹配；
4. 根据不同交通方案+对应的酒店价格进行最终出行总价格计算；
5. 将所有组合的出行方案反馈给用户。

好了，分析完我们大概需要做的步骤，我们就来通过代码实现一下吧。

第一步： 我们先来创建一个固定10个线程的线程池，用来处理以上每一步的任务。

```
//这里使用Executors只是演示，正常使用时推荐使用ThreadPoolExecutor！
```

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

第二步： 部分代码实例，方法就不贴了，太多太长了，大家需要对Future的用法理解即可

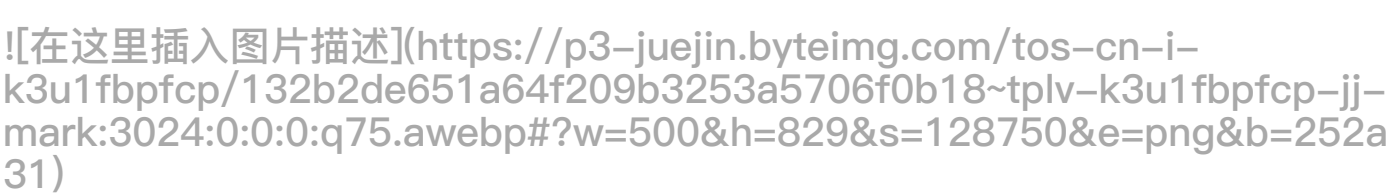
```
// 1. 根据传入的目的地查询所有出行方案，包括交通组合，价格，到站地点，出发时间，到站时间等
        Future<List<TripMethods>> tripMethods = executor.submit(() -> searchMethods(searchCondition));

        List<TripMethods> methods;
        try {
            methods = tripMethods.get();
        } catch (InterruptedException | ExecutionException e) {
            // 处理异常
        }

        // 2. 对每个出行方案的最终到站点查询酒店
        List<Future<List<Hotel>>> futureHotelsList = new ArrayList<>();
        for (TripMethods method : methods) {
            Future<List<Hotel>> futureHotels = executor.submit(() -> searchHotels(method));
            futureHotelsList.add(futureHotels);
        }
        // 出行方案=交通方案+酒店+耗时+价格
        List<Future<List<TravelPackage>>> futureTravelPackagesList = new ArrayList<>();
        for (Future<List<Hotel>> futureHotels : futureHotelsList) {
            List<Hotel> hotels;
            try {
                hotels = futureHotels.get();
            } catch (InterruptedException | ExecutionException e) {
                // 处理异常
            }

            // 3. 对每个交通方案的价格和其对应的酒店价格进行求和
            for (Hotel hotel : hotels) {
                Future<List<TravelPackage>> futureTravelPackages = executor.submit(() -> calculatePrices(hotel));
                futureTravelPackagesList.add(futureTravelPackages);
            }
        }
```

```
        List<TravelPackage> travelPackages = new ArrayList<>();
        for (Future<List<TravelPackage>> futureTravelPackages : futureTra
velPackagesList) {
            try {
                travelPackages.addAll(futureTravelPackages.get());
            } catch (InterruptedException | ExecutionException e) {
                // 处理异常
            }
        }

        // 4. 将所有出行方案按照价格排序
        travelPackages.sort(Comparator.comparing(TravelPackage::getPric
e));

        // 5. 返回结果
        return travelPackages;
```

```

我们在这里将每一步分任务，都作为一个future对象，处理完返回。但是这样会带来诸多问题，比如：我们调用future的get方法是阻塞操作，大大影响效率，并且在复杂的链路关系中，这种拆分式的写法，很难理清楚关联关系，先后关系等；

## 四、CompletableFuture 调优

在这种背景下，Java 8 时引入CompletableFuture 类，它的诞生是为了解决 Future 的这些缺陷。CompletableFuture 除了提供了更为好用和强大的 Future 特性之外，还提供了函数式编程、异步任务编排组合（可以将多个异步任务串联起来，组成一个完整的链式调用）等能力。

```
//CompletableFuture实现了Future的接口方法，CompletionStage 接口描述了一个异步计算的阶段。很多计算可以分成多个阶段或步骤，此时可以通过它将所有步骤组合起来，形成异步计算的流水线。
public class CompletableFuture<T> implements Future<T>, CompletionSt
age<T> {
}
```

```

在CompletableFuture类中通过CompletionStage提供了大量的接口方法，他们让CompletableFuture拥有了出色的函数式编程能力，方法太多，我们无法

一一讲解，只能通过对上面测试源码进行调优时，去使用，使用到的解释一下哈。



在这里插入图片描述

【CompletableFuture优化代码】

```
CompletableFuture.supplyAsync(() -> searchMethods()) // 1. 根据传入的目的地查询所有出行方案，包括交通组合，价格，到站地点，出发时间，到站时间等
                .thenCompose(methods -> { // 2. 对每个出行方案的最终到站点查询酒店
                    List<CompletableFuture<List<TravelPackage>>> travelPackageFutures = methods.stream()
                            .map(method -> CompletableFuture.supplyAsync(() -> searchHotels(method)) // 查询酒店
                                    .thenCompose(hotels -> { // 3. 对每个交通方案的价格和其对应的酒店价格进行求和
                                        List<CompletableFuture<TravelPackage>> packageFutures = hotels.stream()
                                                .map(hotel -
> CompletableFuture.supplyAsync(() -
> new TravelPackage(method, hotel)))
                                                .collect(Collectors.toList());

                                        return CompletableFuture.allOf(packageFutures.toArray(new CompletableFuture[0]))
                                                .thenApply(v -
> packageFutures.stream()
                                                        .map(CompletableFuture::join)
                                                        .collect(Collectors.toList()));
                                    }))
                            .collect(Collectors.toList());

                    return CompletableFuture.allOf(travelPackageFutures.toArray(new CompletableFuture[0]))
                            .thenApply(v -> travelPackageFutures.stream()
```

```
                        .flatMap(future -> future.join().stream())
                        .collect(Collectors.toList()));
            })
            .thenApply(travelPackages -> {  // 4. 将所有出行方案按照价格排序
                return travelPackages.stream()
                        .sorted(Comparator.comparing(TravelPackage::getPrice))
                        .collect(Collectors.toList());
            })
            .exceptionally(e -> {  // 处理所有的异常
                // 处理异常
                return null;
            });
```

在这里我们将整个实现都以一种函数链式调用的方式完成了，看似冗长，实则各个关系的先后非常明确，对于复杂的业务逻辑实现更加容易进行问题的排查与理解。

【解析】

1）在这段代码的开头，我们通过CompletableFuture 自带的静态工厂方法 supplyAsync() 进行对象的创建，平时还可以用以new关键字或者runAsync()方法创建实例；

```
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier);
// 使用自定义线程池(推荐)
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor);
static CompletableFuture<Void> runAsync(Runnable runnable);
// 使用自定义线程池(推荐)
static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor);
```

2）thenCompose()：用 thenCompose() 按顺序链接两个 CompletableFuture 对象，实现异步的任务链。它的作用是将前一个任务的返回结果作为下一个任务的输入参数，从而形成一个依赖关系。注意：这个方法是非阻塞的，即查询酒店的操作会立即开始，而不需要等待查询交通方案的操作完成。

3）thenApply()：thenApply() 方法接受一个 Function 实例，用它来处理结果；

4）allOf()：方法会等到所有的 CompletableFuture 都运行完成之后再返回；

5） 调用 join() 可以让程序等待都运行完了之后再继续执行。

6）exceptionally()：这个方法用于处理CompletableFuture的异常情况，如果CompletableFuture的计算过程中抛出异常，那么这个方法会被调用。

五、总结

好了，今天就讲这么多，其实在Java中通过条用CompletableFuture实现异步编排的工作还是稍微有点难度的，大量的API支持，需要我们在一次次的实战中去熟悉，并灵活使用。推荐大家去看看京东的asyncTool这个框架，里面就大量使用了CompletableFuture。

结尾彩蛋

如果本篇博客对您有一定的帮助，大家记得**留言+点赞+收藏**呀。原创不易，转载请联系Build哥！ ![在这里插入图片描述](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/d888d1e7ea3e417ba07b958b031bbb4b~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=260&h=202&s=37960&e=png&b=fcfafa)

如果您想与Build哥的关系更近一步，还可以“JavaBuild888”，在这里除了看到《Java成长计划》系列博文，还有提升工作效率的小笔记、读书心得、大厂面经、人生感悟等等，欢迎您的加入！ ![在这里插入图片描述](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/deac7f3f68d54f49a8ddb03d739f7b69~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=2394&h=259&s=61904&e=png&b=fefefe)

原文链接: https://juejin.cn/post/7377658332843409419