06 | CompletableFuture：如何理解 Java 8 新引入的异步编程类?

今天，我们一起来看下如何理解 Java8 引入的新异步编程类，CompletableFuture。

在第 05 时，我们直接用“线程”和“阻塞队列”构建实现了一个简单的流计算框架。这个框架帮助我们理解了流计算系统的基本实现原理，但是它用起来不是非常方便，需要配合框架写一些业务无关的代码。

所以，今天我们的目标就是对这个框架进行改造。我们不再用原始的“线程”和“阻塞队列”，而是使用 Java 8 中引入的 CompletableFuture 类。你将看到，用 CompletableFuture 这个异步编程类，实现的流计算框架是多么灵活好用。

Java 8 为啥引入 CompletableFuture 类

在 Java 8 之前，我们写异步代码的时候，主要还是依靠 ExecutorService 类和 Future 类。Future 类提供了 get 方法，用于在任务完成时获取任务结果。但是，Future 类的 get 方法有个缺点，它是阻塞的，需要同步等待结果返回。这就在事实上让原本异步执行的过程，重新退化成了同步的过程，失去了异步的作用。

为了避免这种问题，不同的第三方库提供了不同的解决方案，比如 Guava 库中的SettableFuture/ListenableFuture、Netty 中的 Future 和 ChannelFuture 等。这些解决方案都是通过注册监听或回调的方式，形成回调链，从而实现了真正意义上的异步执行。

与此形成鲜明对比的是，JDK 自己却一直没有真正的异步编程工具类。

所以，在被 JavaScript、C# 等诸多语言嫌弃后，Java 8 终于推出了自己的异步编程方案，这就是 CompletableFuture 类。

CompletableFuture 类采用回调的方式实现异步执行，并提供了大量有关构建异步调用链的API。这些 API 使得 Java 异步编程变得无比灵活和方便，极大程度地解放了 Java 异步编程的生产力。可以说，**CompletableFuture 类仅凭一己之力，将 Java 异步编程提升到了一个全新的境界**。

所以，接下来我们就先看看 CompletableFuture 类都有哪些神奇的方法。

常用的 CompletableFuture 类方法

如果你查看 Java 8 中 CompletableFuture 类的源码，你会发现这个类有 80 多个可以公共访问的方法。并且这些方法中，很多方法的名字相似，它们的注释说明也似乎大同小异。再加上本身这个类是新引入的异步编程工具类。所以，对于初次接触这个类，以及对异步编程并不熟悉的开发人员来说，很容易没有头绪，不知道具体怎么使用这些方法，也不知道从哪个方法开始。

但爽哥想说的是，不用担心，当你理解这个类后，你会发现 CompletableFuture 类的所有这些方法之间，是有非常强的逻辑性的。通过这些方法，你可以构建出各种各样的异步调用链过程。

所以，为了帮助你更具逻辑性地理解 CompletableFuture 类。我将通过“产品在流水线上被一步一步加工，直到产品加工完成后被装入仓库”的过程，来依次讲解 CompletableFuture 类中最主要的几个类的使用逻辑。在模块一时，我们已经讲过“流”和“异步”是相通的，所以这里借助流水线的方式来讲解 CompletableFuture 类也是十分合适的。

1. 既然要生产产品，那么首先就是将毛坯产品放在流水线的“起点”处。这个过程，是通过 supplyAsync 方法来完成的。supplyAsync 方法的定义如下：

```
public static <U> CompletableFuture<U> supplyAsync(
    Supplier<U> supplier, Executor executor)
```

supplyAsync 是开启 CompletableFuture 异步调用链的方法之一。使用这个方法，会将supplier 封装为一个任务提交给 executor 执行，然后返回一个记录任务执行状态和结果的 CompletableFuture 对象。之后可以在这个 CompletableFuture 对象上挂接各种回调动作。
所以说，supplyAsync 可以作为“流”的起点。

2. 当毛坯产品放在流水线上后，它就在流水线上传动起来。之后，当毛坯产品传动到加工位置时，就需要对其进行“加工”了。而“加工”就是通过 thenApplyAsync 方法来完成的。

```
public <U> CompletableFuture<U> thenApplyAsync(
    Function<? super T,? extends U> fn, Executor executor)
```

thenApplyAsync 用于在 CompletableFuture 对象上挂接一个转化函数。当 CompletableFuture 对象完成时，将它的结果作为输入参数调用转化函数。转化函数在执行各种逻辑后，返回另一种类型的数据作为输出。
这么一看，thenApplyAsync 的作用就是对“流”上的数据进行处理。

3. 当毛坯产品在流水线上最终被加工完成后，就变成了一个成品。所以，接下来我们就需要将这个成品装箱入库。而“装箱入库”的动作，就是通过 thenAcceptAsync 方法来完成的。

```
public CompletableFuture<Void> thenAcceptAsync(
    Consumer<? super T> action, Executor executor)
```

thenAcceptAsync 用于在 CompletableFuture 对象上挂接一个接收函数。当CompletableFuture 对象完成时，将它的结果作为输入参数调用接收函数。与 thenApplyAsync 类似，接收函数可以执行各种逻辑，但不同的是，接收函数不会返回任何类型数据，或者说返回类型是 void。

所以，thenAcceptAsync可以作为“流”的终点。

4. 现在需要对流水线进行升级改造，将流水线的其中一段，改造成“另外一条”流水线。那这个工作，就是通过 thenComposeAsync 方法来实现的。

```
public <U> CompletableFuture<U> thenComposeAsync(
    Function<? super T, ? extends CompletionStage<U>> fn, Executor executor)
```

thenComposeAsync 理解起来会复杂些，但它真的是一个非常重要的方法，请你务必理解它。
thenComposeAsync 在 API 形式上与 thenApplyAsync 类似，但是它的转化函数返回的不是一般类型的对象，而是一个 CompletionStage 对象，或者说得更具体点，实际中通常就是一个 CompletableFuture 对象。这意味着，我们可以在原来的 CompletableFuture 调用链上，插入另外一个调用链，从而形成一个新的调用链。这正是 compose(组成、构成)的含义所在。

所以，thenComposeAsync 的作用，就像是在“流”的某个地方，插入了另外一条“流”。

5. 现在老板为了鼓励工人更加努力的工作，就告诉大家谁先完成作业就给谁发奖金。那这个“谁先完成就由谁领奖金”的机制，是通过 applyToEither 方法来实现的。

```
public <U> CompletableFuture<U> applyToEither(
    CompletionStage<? extends T> other, Function<? super T, U> fn)
```

使用 applyToEither 可以实现两个 CompletableFuture 谁先完成，就由谁执行回调函数的功能。这也是一个非常有用的方法，爽哥经常用它来实现定时超期的功能。

6. 有一天车间运来了一个大货箱，一个人搬不动，所以老板就让“大家忙完手头的事后一起来搬运这个大货箱”。那这种“大家一起完成后再执行某个动作”的过程，就是由 allOf 方法来实现的。

```
public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
```

CompletableFuture.allOf 的作用是将多个 CompletableFuture 合并成一个CompletableFuture。这又是一个非常有用的方法，我们可以用它实现类似于 Map/Reduce 或 Fork/Join 的功能。

7. 流水线上工人在加工产品时，总会时不时地发生些意外情况，那发生意外情况后该怎么办呢？这就是由 exceptionally 方法来处理的。

```
public CompletableFuture<T> exceptionally(
        Function<Throwable, ? extends T> fn)
```

相比同步编程方式，异步程序发生异常时的问题会更加复杂。使用 exceptionally 方法，可以对异步调用链在执行过程中抛出的异常进行处理。

所以，通过这种“流水线上加工商品”的过程，我们很容易将 CompletableFuture 类中最主要的几个方法的功能和使用逻辑串起来。

后面你在使用 CompletableFuture 类构建异步调用链时，遇到难以理解的地方，可以时不时地联想下上面的流水线场景。

CompletableFuture 工作原理

前面介绍了 CompletableFuture 的几个常用 API，但光知道这些 API 还不足以体会到 CompletableFuture 的奥义和乐趣所在。我们最好还需要理解 CompletableFuture 类的内部工作原理。

所以接下来，我们借助于下面这段代码（完整代码参考）来详细分析下CompletableFuture类的工作原理：

```
CompletableFuture<String> cf1 = CompletableFuture.supplyAsync(Tests::source, executor1);
CompletableFuture<String> cf2 = cf1.thenApplyAsync(Tests::echo, executor2);
CompletableFuture<String> cf3_1 = cf2.thenApplyAsync(Tests::echo1, executor3);
CompletableFuture<String> cf3_2 = cf2.thenApplyAsync(Tests::echo2, executor3);
CompletableFuture<String> cf3_3 = cf2.thenApplyAsync(Tests::echo3, executor3);
CompletableFuture<Void> cf3 = CompletableFuture.allOf(cf3_1, cf3_2, cf3_3);
CompletableFuture<Void> cf4 = cf3.thenAcceptAsync(x -> print("world"), executor4);
```

通过阅读 JDK 源码，并借助于 IDE 的断点（比如 Mac 环境下 IntelliJ IDEA 的 F7 和 F8 功能键逐步调试）调试功能，可以追踪出以上代码生成 CompletableFuture 异步调用链的过程，如下图 1 所示。

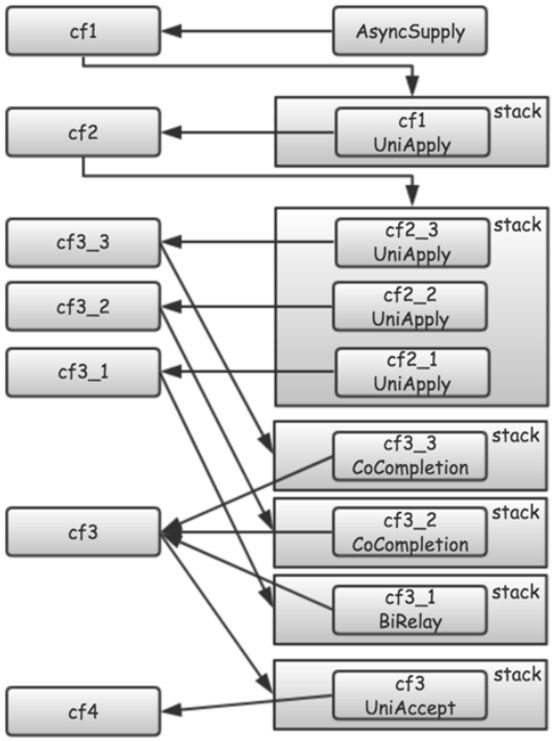

图1 CompletableFuture执行过程

具体来说，CompletableFuture 的异步调用链是这样形成的。

首先，通过 CompletableFuture.supplyAsync 创建了一个任务 Tests::source，并交给executor1 异步执行。用 cf1 来记录该任务在执行过程中的状态和结果。

然后，通过 cf1.thenApplyAsync，指定当 cf1(Tests::source) 完成时，需要回调的任务Tests::echo。cf1 使用 stack 来管理这个后续要回调的任务Tests::echo。用 cf2 来记录回调任务 Tests::echo 的执行状态和结果。

再然后，通过连续三次调用 cf2.thenApplyAsync，指定当 cf2(Tests::echo) 完成时，需要回调的后续三个任务：Tests::echo1、Tests::echo2 和 Tests::echo3。cf2 也是用 stack 来管理这三个后续需要执行的任务。

接着，通过 CompletableFuture.allOf，创建一个合并 cf3_1、cf3_2、cf3_3 的 cf3。这也意味着 cf3 只有在 cf3_1、cf3_2、cf3_3 都完成时才能完成。在 cf3 内部，是用一个**树（Tree）结构**来记录它和 cf3_1、cf3_2、cf3_3 的依赖关系。

最后，通过 cf3.thenAcceptAsync，指定了当 cf3 完成时，需要回调的任务，即 print。用 cf4来记录 print 任务的状态和结果。

所以总的来说，就是 CompletableFuture 用 stack 来管理它在完成时后续需要回调的任务。当任务完成时，再通过依赖关系，找到后续需要处理的 CompletableFuture，并继续调用执行。这样，就构成了一个调用链，所有任务将按照该调用链依次执行。

采用 CompletableFuture 实现流计算框架

现在，我们已经对 CompletableFuture 的功能、API 和工作原理都有了一定认识。接下来就是实践了，这里，我们用 CompletableFuture 对 05 课时中的流计算框架进行改进。

闲话少叙，咱们先直接上代码（完整代码参考）：

```
private final ExecutorService decoderExecutor = new BackPressureExecutor("decoderExecutor", 1, 2,
private final ExecutorService extractExecutor = new BackPressureExecutor("extractExecutor", 1, 4,
private final ExecutorService senderExecutor = new BackPressureExecutor("senderExecutor", 1, 2, 1
private final ExecutorService extractService = new BackPressureExecutor("extractService", 1, 16,
byte[] event = receiver.receive();
CompletableFuture
        .supplyAsync(() -> decoder.decode(event), decoderExecutor)
        .thenComposeAsync(extractor::extract, extractExecutor)
        .thenAcceptAsync(sender::send, senderExecutor)
        .exceptionally(e -> {
            logger.error("unexpected exception", e);
            return null;
        });
```

看，是不是非常简单！

上面的代码中，receiver 读取消息后，通过 supplyAsync 方法，将其交给 decoder 解码。消息在解码后，再交给 extractor 进行特征提取。

由于 extractor 内部有个独立的“流”用于特征的并行计算，故采用 thenComposeAsync 将这个内部“流”插入到整体的“流”中来。

“流”的最后一步是将消息发送到 Kafka，所以使用 thenAcceptAsync 作为“流”的终点。

另外，为了让整个异步调用链在执行过程中不会出现 OOM 问题，我们还使用了带反向压力功能的执行器 BackPressureExecutor。这个执行器的原理和实现方法，我们已经在 03 课时讨论过，这里重新提醒下，就不再赘述了。

从上面的改造过程可以看出，CompletableFuture 本身就是一个非常好用的流计算框架。短短几行代码，就实现了我们在 05 课时中的所有功能。

小结

今天，我们使用 CompletableFuture 这个异步编程框架，实现了一个流计算应用。

我们可以回顾下，在 05 课时中，我们实现的流计算框架，其主要工作原理是，线程从队列中读取数据进行处理，然后输出到下游队列。而在今天的课时中，我们使用 CompletableFuture 实现的流计算过程，也是使用队列在整个处理过程中传递数据。

这两种实现的工作原理在本质是完全一致的，但很明显，使用 CompletableFuture 更加灵活方便。令人开心的是，CompletableFuture 类的方法还在不断增强中。

所以在以后的开发中，当你遇到复杂的业务问题时，不妨从“流”的角度分析问题。这样你会发现自己的考虑重点，将更多地放在业务本身上。即使再复杂的业务流程，也只需要多分解几个步骤就可以了。

这时候你设计出的程序结构，将会变得更加直观清晰，性能提升也会变得更加容易。

所以，关于 CompletableFuture 类你还有什么疑问或想法呢？可以在课程的留言区将你的问题或想法写下来，我在看到后会进行分析和讲解，或者在后续的课程中进一步补充说明。

> 本课时精华：

点击此链接查看本课程所有课时的源码

PB 级企业大数据项目实战 + 拉勾硬核内推，5 个月全面掌握大数据核心技能。点击链接，全面赋能！