24 | CompletableFuture：异步编程没那么难

2019-04-23 王宝令

前面我们不止一次提到，用多线程优化性能，其实不过就是将串行操作变成并行操作。如果仔细观察，你还会发现在串行转换成并行的过程中，一定会涉及到异步化，例如下面的示例代码，现在是串行的，为了提升性能，我们得把它们并行化，那具体实施起来该怎么做呢？

```
//以下两个方法都是耗时操作
doBizA();
doBizB();
```

还是挺简单的，就像下面代码中这样，创建两个子线程去执行就可以了。你会发现下面的并行方案，主线程无需等待doBizA()和doBizB()的执行结果，也就是说doBizA()和doBizB()两个操作已经被异步化了。

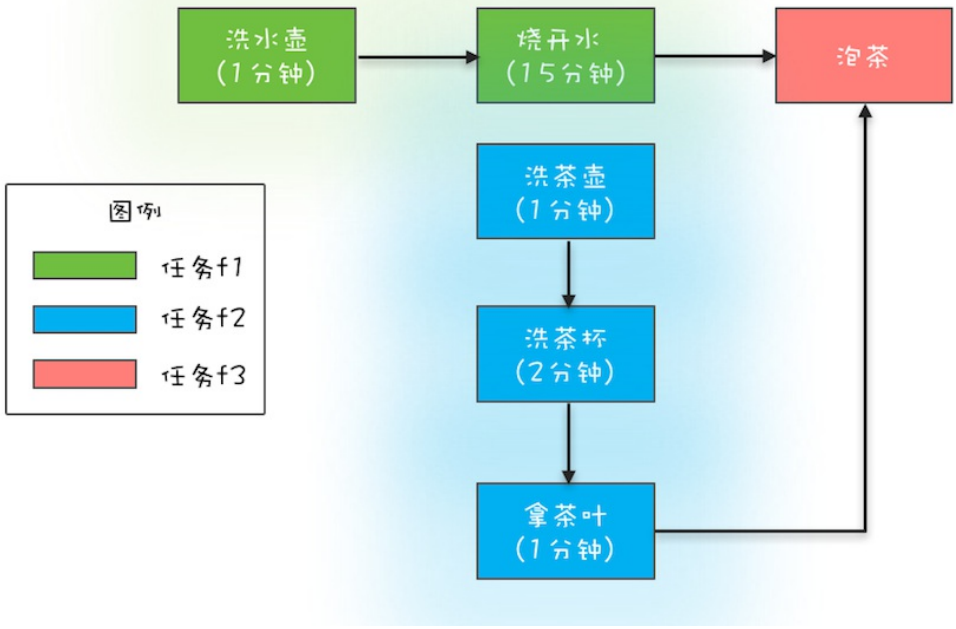
```
new Thread(()->doBizA())
  .start();
new Thread(()->doBizB())
  .start();
```

异步化，是并行方案得以实施的基础，更深入地讲其实就是：**利用多线程优化性能这个核心方**

案得以实施的基础。看到这里，相信你应该就能理解异步编程最近几年为什么会大火了，因为优化性能是互联网大厂的一个核心需求啊。Java在1.8版本提供了CompletableFuture来支持异步编程，CompletableFuture有可能是你见过的最复杂的工具类了，不过功能也着实让人感到震撼。

CompletableFuture的核心优势

为了领略CompletableFuture异步编程的优势，这里我们用CompletableFuture重新实现前面曾提及的烧水泡茶程序。首先还是需要先完成分工方案，在下面的程序中，我们分了3个任务：任务1负责洗水壶、烧开水，任务2负责洗茶壶、洗茶杯和拿茶叶，任务3负责泡茶。其中任务3要等待任务1和任务2都完成后才能开始。这个分工如下图所示。



烧水泡茶分工方案

下面是代码实现，你先略过runAsync()、supplyAsync()、thenCombine()这些不太熟悉的方法，从大局上看，你会发现：

1. 无需手工维护线程，没有繁琐的手工维护线程的工作，给任务分配线程的工作也不需要我们关注；
2. 语义更清晰，例如 f3 = f1.thenCombine(f2, ()->{}) 能够清晰地表述“任务3要等待任务1和任务2都完成后才能开始”；
3. 代码更简练并且专注于业务逻辑，几乎所有代码都是业务逻辑相关的。

```
//任务1：洗水壶->烧开水
CompletableFuture<Void> f1 =
  CompletableFuture.runAsync(()->{
```

```
  System.out.println("T1:洗水壶...");
  sleep(1, TimeUnit.SECONDS);

  System.out.println("T1:烧开水...");
  sleep(15, TimeUnit.SECONDS);
});
//任务2：洗茶壶->洗茶杯->拿茶叶
CompletableFuture<String> f2 =
  CompletableFuture.supplyAsync(()->{
  System.out.println("T2:洗茶壶...");
  sleep(1, TimeUnit.SECONDS);

  System.out.println("T2:洗茶杯...");
  sleep(2, TimeUnit.SECONDS);

  System.out.println("T2:拿茶叶...");
  sleep(1, TimeUnit.SECONDS);
  return "龙井";
});
//任务3：任务1和任务2完成后执行：泡茶
CompletableFuture<String> f3 =
  f1.thenCombine(f2, (__, tf)->{
    System.out.println("T1:拿到茶叶:" + tf);
    System.out.println("T1:泡茶...");
    return "上茶:" + tf;
  });
//等待任务3执行结果
System.out.println(f3.join());

void sleep(int t, TimeUnit u) {
  try {
    u.sleep(t);
  }catch(InterruptedException e){}
}
// 一次执行结果：
T1:洗水壶...
T2:洗茶壶...
T1:烧开水...
```

```
T1:烧开水...
T2:洗茶杯...
T2:拿茶叶...
T1:拿到茶叶:龙井
T1:泡茶...
上茶:龙井
```

领略CompletableFuture异步编程的优势之后，下面我们详细介绍CompletableFuture的使用，首先是如何创建CompletableFuture对象。

创建CompletableFuture对象

创建CompletableFuture对象主要靠下面代码中展示的这4个静态方法，我们先看前两个。在烧水泡茶的例子中，我们已经使用了runAsync(Runnable runnable)和supplyAsync(Supplier<U> supplier)，它们之间的区别是：Runnable 接口的run()方法没有返回值，而Supplier接口的get()方法是有返回值的。

前两个方法和后两个方法的区别在于：后两个方法可以指定线程池参数。

默认情况下CompletableFuture会使用公共的ForkJoinPool线程池，这个线程池默认创建的线程数是CPU的核数（也可以通过JVM option:-Djava.util.concurrent.ForkJoinPool.common.parallelism来设置ForkJoinPool线程池的线程数）。如果所有CompletableFuture共享一个线程池，那么一旦有任务执行一些很慢的I/O操作，就会导致线程池中所有线程都阻塞在I/O操作上，从而造成线程饥饿，进而影响整个系统的性能。所以，强烈建议你要**根据不同的业务类型创建不同的线程池，以避免互相干扰**。

```
//使用默认线程池
static CompletableFuture<Void>
  runAsync(Runnable runnable)
static <U> CompletableFuture<U>
  supplyAsync(Supplier<U> supplier)
//可以指定线程池
static CompletableFuture<Void>
  runAsync(Runnable runnable, Executor executor)
static <U> CompletableFuture<U>
  supplyAsync(Supplier<U> supplier, Executor executor)
```

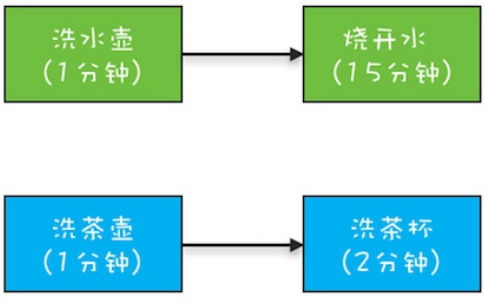
创建完CompletableFuture对象之后，会自动地异步执行runnable.run()方法或者supplier.get()方法，对于一个异步操作，你需要关注两个问题：一个是异步操作什么时候结束，另一个是如何获

取异步操作的执行结果。因为CompletableFuture类实现了Future接口，所以这两个问题你都可以通过Future接口来解决。另外，CompletableFuture类还实现了CompletionStage接口，这个接口内容实在是太丰富了，在1.8版本里有40个方法，这些方法我们该如何理解呢？

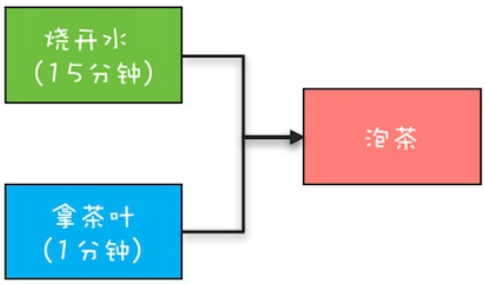
如何理解CompletionStage接口

我觉得，你可以站在分工的角度类比一下工作流。任务是有时序关系的，比如有**串行关系、并行关系、汇聚关系**等。这样说可能有点抽象，这里还举前面烧水泡茶的例子，其中洗水壶和烧开水就是串行关系，洗水壶、烧开水和洗茶壶、洗茶杯这两组任务之间就是并行关系，而烧开水、拿茶叶和泡茶就是汇聚关系。

串行关系



并行关系



汇聚关系

CompletionStage接口可以清晰地描述任务之间的这种时序关系，例如前面提到的 f3 = f1.thenCombine(f2, ()->{}) 描述的就是一种汇聚关系。烧水泡茶程序中的汇聚关系是一种 AND 聚合关系，这里的AND指的是所有依赖的任务（烧开水和拿茶叶）都完成后才开始执行当前任务（泡茶）。既然有AND聚合关系，那就一定还有OR聚合关系，所谓OR指的是依赖的任务只要有一个完成就可以执行当前任务。

在编程领域，还有一个绕不过去的山头，那就是异常处理，CompletionStage接口也可以方便地描述异常处理。

下面我们就来一一介绍，CompletionStage接口如何描述串行关系、AND聚合关系、OR聚合关系以及异常处理。

1. 描述串行关系

CompletionStage接口里面描述串行关系，主要是thenApply、thenAccept、thenRun和thenCompose这四个系列的接口。

thenApply系列函数里参数fn的类型是接口Function<T, R>，这个接口里与CompletionStage相关的方法是 R apply(T t)，这个方法既能接收参数也支持返回值，所以thenApply系列方法返回的是CompletionStage<R>。

而thenAccept系列方法里参数consumer的类型是接口Consumer<T>，这个接口里与CompletionStage相关的方法是 void accept(T t)，这个方法虽然支持参数，但却不支持回值，所以thenAccept系列方法返回的是CompletionStage<Void>。

thenRun系列方法里action的参数是Runnable，所以action既不能接收参数也不支持返回值，所以thenRun系列方法返回的也是CompletionStage<Void>。

这些方法里面Async代表的是异步执行fn、consumer或者action。其中，需要你注意的是thenCompose系列方法，这个系列的方法会新创建出一个子流程，最终结果和thenApply系列是相同的。

```
CompletionStage<R> thenApply(fn);
CompletionStage<R> thenApplyAsync(fn);
CompletionStage<Void> thenAccept(consumer);
CompletionStage<Void> thenAcceptAsync(consumer);
CompletionStage<Void> thenRun(action);
CompletionStage<Void> thenRunAsync(action);
CompletionStage<R> thenCompose(fn);
CompletionStage<R> thenComposeAsync(fn);
```

通过下面的示例代码，你可以看一下thenApply()方法是如何使用的。首先通过supplyAsync()启动一个异步流程，之后是两个串行操作，整体看起来还是挺简单的。不过，虽然这是一个异步流程，但任务①②③却是串行执行的，②依赖①的执行结果，③依赖②的执行结果。

```
CompletableFuture<String> f0 =
  CompletableFuture.supplyAsync(
    () -> "Hello World")      //①
  .thenApply(s -> s + " QQ")  //②
  .thenApply(String::toUpperCase);//③

System.out.println(f0.join());
//输出结果
HELLO WORLD QQ
```

2. 描述AND汇聚关系

CompletionStage接口里面描述AND汇聚关系，主要是thenCombine、thenAcceptBoth和runAfterBoth系列的接口，这些接口的区别也是源自fn、consumer、action这三个核心参数不同。它们的使用你可以参考上面烧水泡茶的实现程序，这里就不赘述了。

```
CompletionStage<R> thenCombine(other, fn);
CompletionStage<R> thenCombineAsync(other, fn);
CompletionStage<Void> thenAcceptBoth(other, consumer);
CompletionStage<Void> thenAcceptBothAsync(other, consumer);
CompletionStage<Void> runAfterBoth(other, action);
CompletionStage<Void> runAfterBothAsync(other, action);
```

3. 描述OR汇聚关系

CompletionStage接口里面描述OR汇聚关系，主要是applyToEither、acceptEither和runAfterEither系列的接口，这些接口的区别也是源自fn、consumer、action这三个核心参数不同。

```
CompletionStage applyToEither(other, fn);
CompletionStage applyToEitherAsync(other, fn);
CompletionStage acceptEither(other, consumer);
CompletionStage acceptEitherAsync(other, consumer);
CompletionStage runAfterEither(other, action);
CompletionStage runAfterEitherAsync(other, action);
```

下面的示例代码展示了如何使用applyToEither()方法来描述一个OR汇聚关系。

```
CompletableFuture<String> f1 =
  CompletableFuture.supplyAsync(()->{
    int t = getRandom(5, 10);
    sleep(t, TimeUnit.SECONDS);
    return String.valueOf(t);
});

CompletableFuture<String> f2 =
  CompletableFuture.supplyAsync(()->{
    int t = getRandom(5, 10);
    sleep(t, TimeUnit.SECONDS);
    return String.valueOf(t);
});

CompletableFuture<String> f3 =
  f1.applyToEither(f2,s -> s);

System.out.println(f3.join());
```

4. 异常处理

虽然上面我们提到的fn、consumer、action它们的核心方法都**不允许抛出可检查异常，但是却无法限制它们抛出运行时异常**，例如下面的代码，执行 7/0 就会出现除零错误这个运行时异常。非异步编程里面，我们可以使用try{}catch{}来捕获并处理异常，那在异步编程里面，异常该如何处理呢？

```
CompletableFuture<Integer>
  f0 = CompletableFuture.
    .supplyAsync(()->(7/0))
    .thenApply(r->r*10);
System.out.println(f0.join());
```

CompletionStage接口给我们提供的方案非常简单，比try{}catch{}还要简单，下面是相关的方法，使用这些方法进行异常处理和串行操作是一样的，都支持链式编程方式。

```
CompletionStage exceptionally(fn);
CompletionStage<R> whenComplete(consumer);
CompletionStage<R> whenCompleteAsync(consumer);
CompletionStage<R> handle(fn);
CompletionStage<R> handleAsync(fn);
```

下面的示例代码展示了如何使用exceptionally()方法来处理异常，exceptionally()的使用非常类似于try{}catch{}中的catch{}，但是由于支持链式编程方式，所以相对更简单。既然有try{}catch{}，那就一定还有try{}finally{}，whenComplete()和handle()系列方法就类似于try{}finally{}中的finally{}，无论是否发生异常都会执行whenComplete()中的回调函数consumer和handle()中的回调函数fn。whenComplete()和handle()的区别在于whenComplete()不支持返回结果，而handle()是支持返回结果的。

```
CompletableFuture<Integer>
  f0 = CompletableFuture
    .supplyAsync(()->7/0))
    .thenApply(r->r*10)
    .exceptionally(e->0);
System.out.println(f0.join());
```

总结

曾经一提到异步编程，大家脑海里都会随之浮现回调函数，例如在JavaScript里面异步问题基本上都是靠回调函数来解决的，回调函数在处理异常以及复杂的异步任务关系时往往力不从心，对此业界还发明了个名词：**回调地狱**（Callback Hell）。应该说在前些年，异步编程还是声名狼藉的。

不过最近几年，伴随着ReactiveX的发展（Java语言的实现版本是RxJava），回调地狱已经被完美解决了，异步编程已经慢慢开始成熟，Java语言也开始官方支持异步编程：在1.8版本提供了CompletableFuture，在Java 9版本则提供了更加完备的Flow API，异步编程目前已经完全工业化。因此，学好异步编程还是很有必要的。

CompletableFuture已经能够满足简单的异步编程需求，如果你对异步编程感兴趣，可以重点关注RxJava这个项目，利用RxJava，即便在Java 1.6版本也能享受异步编程的乐趣。

课后思考

创建采购订单的时候，需要校验一些规则，例如最大金额是和采购员级别相关的。有同学利用CompletableFuture实现了这个校验的功能，逻辑很简单，首先是从数据库中把相关规则查出

来，然后执行规则校验。你觉得他的实现是否有问题呢？

```
//采购订单
PurchersOrder po;
CompletableFuture<Boolean> cf =
  CompletableFuture.supplyAsync(()->{
    //在数据库中查询规则
    return findRuleByJdbc();
  }).thenApply(r -> {
    //规则校验
    return check(po, r);
});
Boolean isOk = cf.join();
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言

袁阳

12

思考题:

1，读数据库属于io操作，应该放在单独线程池，避免线程饥饿

2，异常未处理

2019-04-23

作者回复

2019-04-23

密码123456 9

我在想一个问题，明明是串行过程，直接写就可以了。为什么还要用异步去实现串行？

2019-04-23

作者回复

这个简单场景没必要用

2019-04-23

青莲 6

1.查数据库属于io操作，用定制线程池

2.查出来的结果做为下一步处理的条件，若结果为空呢，没有对应处理

3.缺少异常处理机制

2019-04-23

作者回复

2019-04-23

tyul 3

回答「密码123456」：CompletableFuture 在执行的过程中可以不阻塞主线程，支持 runAsync、anyOf、allOf 等操作，等某个时间点需要异步执行的结果时再阻塞获取。

2019-04-23

作者回复

是的，复杂场景就能体现出优势了

2019-04-23

刘晓林 3

思考题：

1.没有进行异常处理，

2.要指定专门的线程池做数据库查询

3.如果检查和查询都比较耗时，那么应该像之前的对账系统一样，采用生产者和消费者模式，让上一次的检查和下一次的查询并行起来。

另外，老师把javadoc里那一堆那一堆方法进行了分类，分成串行、并行、AND聚合、OR聚合，简直太棒了，一下子就把这些方法纳入到一个完整的结构体系里了。简直棒

2019-04-23

作者回复

思考题考虑的很全面

2019-04-23

发条橙子 。 2

老师，我有个疑问。 completableFuture 中各种关系（并行、串行、聚合），实际上就覆盖了各种需求场景。 例如： 线程A 等待 线程B 或者 线程C 等待 线程A和B 。

我们之前讲的并发包里面 countdownLatch , 或者 threadPoolExecutor 和future 就是来解决这些关系场景的， 那有了 completableFuture 这个类， 是不是以后有需求都优先考虑用 completableFuture ？ 感觉这个类就可以解决前面所讲的类的问题了

2019-04-24

作者回复

我觉得可以优先使用CompletableFuture，当然前提是你的jdk是1.8

2019-04-24

刘晓林 2

我觉得既然都讲到CompletableFuture了，老师是不是有必要不一章ForkJoinPool呀？毕竟，ForkJoinPool和ThreadPoolExecutor还是有很多不一样的。谢谢老师

2019-04-23

作者回复

后面有介绍

2019-04-23

笃行之 1

”如果所有 CompletableFuture 共享一个线程池，那么一旦有任务执行一些很慢的 I/O 操作，就会导致线程池中所有线程都阻塞在 I/O 操作上，从而造成线程饥饿，进而影响整个系统的性能。”老师，阻塞在io上和是不是在一个线程池没关系吧？

2019-04-29

作者回复

有关系，如果系统就一个线程池，里面的线程都阻塞在io上，那么系统其他的任务都需要等待。如果其他任务有自己的线程池，就没有问题。

2019-04-29

易儿易 1

老师我有一个问题：在描述串行关系时，为什么参数没有other？这让我觉得并不是在描述两个子任务的串行关系，而是给第一个子任务追加了一个类似“回调方法”fn等.....而并行关系和汇聚关系则很明确的出现了other.....

2019-04-23

作者回复

你也可以理解成给第一个子任务追加了一个类似“回调方法”。回调不也是在第一个任务执行完才回调吗？所以也是串行的。都是一回事，你怎么理解起来顺手就怎么理解就可以了。

2019-04-24

linqw 1

课后习题，规则校验依赖于数据库中的规则，如果规则不是共用的，直接放在一个内部，如果规则是共用，可以在主线程进行规则获取，异步校验规则。thenApply会重新创建一个CompletableFuture

```
PurchersOrder po;
CompletableFuture<Boolean> cf =
CompletableFuture.supplyAsync(()->{
// 在数据库中查询规则
r = findRuleByJdbc();
// 规则校验
return check(po, r);
});
Boolean isOk = cf.join();
```

CompletableFuture的写法和rxjava的使用很类似，一个结果作为下一个的参数，链式操作等

2019-04-23

木木匠 1

我觉得课后思考题中，既然是先查规则再校验，这本来就是一个串行化的动作，为什么要异步呢？用异步的意义在哪？

2019-04-23

Michael 0

老师 你好，对文章点赞这种功能异步如何实现？

2019-05-23

作者回复

喊一嗓子，让朋友点

2019-05-23

大卫 0

王老师，您好。

目前业务场景我觉得适合用completablefuture，一个详情页，动态接口，会调用多个上游接口做聚合，部分接口之间有依赖。

这些上游分别是不同业务线的，比如搜索、推荐、会员、用户、其他等。

问题1:您建议是每个业务线都是要建立独立的线程池？还是说几个业务线一个线程池?

问题2:这种一般是要按io密集型设置cpu大小吧，cpu核数 *（1-0.9），参考网上的0.9阻塞系数，合适么?

问题3:每个业务独立线程池，同一个jvm中会不会线程数量太大了，有什么额外影响吗?

谢谢

2019-05-19

xuery 0

Completable使用注意事项：1.不同的业务场景最好指定单独的线程池，避免相互影响

2.记得考虑异常处理

2019-05-15

佑儿 0

带有asyn的方法是异步执行，这里的异步是不在当前线程中执行？ 比较困惑

2019-05-10

作者回复

不是在调用方法的线程中执行的，这样是不是更容易理解

2019-05-12

Sunqc 0

评论区那个从多张表查数据然后验证保存到一张表。分页每次次读1000条数据的话

1.采用线程池+future，每次提交的任务结果保存到一个队列里，然后执行任务取队列结果执行保存；或者不采用队列

2.采用completionservice

3.就是这节的主题completionfuture

老师，您看重点前两个方案可行吗，compelerion处理批量操作，会不会大材小用的感觉啊哈哈这个线程池里线程数多大才合适呢

2019-04-30

aroll 0

嗯对，我以log的打印为准了，log打印结束并不代表主线程已经结束了，还是有个时间差，这个时候子线程还会运行一段时间，感谢老师

2019-04-29

作者回复

找到原因就好

2019-04-29

aroll 0

是的，启动前设置成守护线程了，就像这样

```
public static void main(String[] args){
Thread thread = new Thread(new Runnable() {
@Override
public void run() {
for(int i=0;i<10;i++){
try {
Thread.sleep(1);
} catch (InterruptedException e) {
e.printStackTrace();
}
log.info("子线程执行任务"+i);
}
}
```

```
});
thread.setDaemon(true);
thread.start();
for (int j=0;j<3;j++){
log.info("主线程执行任务"+j);
}
log.info("运行结束");
}
```

如果把sleep部分去掉，即使设成守护线程，主线程结束后子线程仍不会结束

2019-04-27

作者回复

我把sleep部分去掉，for改成while true，主线程结束，子线程还是能结束的。是不是log的锅？

2019-04-28

aroll 0

老师想请教您一个问题，我创建了一个用户线程然后将它设置为守护线程，为什么主线程结束时，它没有结束，需要在它的执行逻辑里调用sleep才会当主线程结束时结束。

2019-04-26

作者回复

启动之前设置成守护线程了？

2019-04-26

Zach_ 0

很喜欢这个专栏!

但是，老师说 教好学生，饿死师傅。 我.....