

## 24 | CompletableFuture: 异步编程没那么难

2019-04-23 王宝令

Java并发编程实战

[进入课程 >](#)



讲述: 王宝令

时长 10:53 大小 9.98M




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？

复制代码

```
1 // 以下两个方法都是耗时操作
2 doBizA();
3 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。

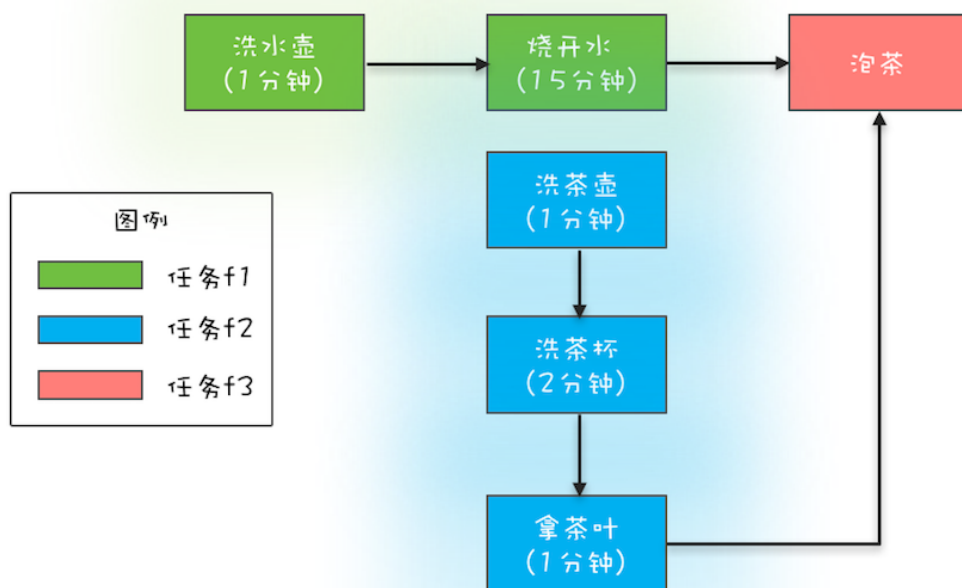
 复制代码

```
1 new Thread(()->doBizA())
2     .start();
3 new Thread(()->doBizB())
4     .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。

 复制代码

```
1 // 任务 1: 洗水壶 -> 烧开水
2 CompletableFuture<Void> f1 =
3     CompletableFuture.runAsync(()->{
4         System.out.println("T1: 洗水壶...");
5         sleep(1, TimeUnit.SECONDS);
6
7         System.out.println("T1: 烧开水...");
8         sleep(15, TimeUnit.SECONDS);
9     });
10 // 任务 2: 洗茶壶 -> 洗茶杯 -> 拿茶叶
11 CompletableFuture<String> f2 =
12     CompletableFuture.supplyAsync(()->{
13         System.out.println("T2: 洗茶壶...");
14         sleep(1, TimeUnit.SECONDS);
15
16         System.out.println("T2: 洗茶杯...");
17         sleep(2, TimeUnit.SECONDS);
18
19         System.out.println("T2: 拿茶叶...");
20         sleep(1, TimeUnit.SECONDS);
21         return " 龙井 ";
22     });
23 // 任务 3: 任务 1 和任务 2 完成后执行: 泡茶
24 CompletableFuture<String> f3 =
25     f1.thenCombine(f2, (_, tf)->{
26         System.out.println("T1: 拿到茶叶:" + tf);
27         System.out.println("T1: 泡茶...");
28         return " 上茶:" + tf;
29     });
30 // 等待任务 3 执行结果
31 System.out.println(f3.join());
32
33 void sleep(int t, TimeUnit u) {
34     try {
```

```
35     u.sleep(t);
36 }catch(InterruptedException e){}
37 }
38 // 一次执行结果:
39 T1: 洗水壶...
40 T2: 洗茶壶...
41 T1: 烧开水...
42 T2: 洗茶杯...
43 T2: 拿茶叶...
44 T1: 拿到茶叶: 龙井
45 T1: 泡茶...
46 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**根据不同的业务类型创建不同的线程池，以避免互相干扰**。

 复制代码

```
1 // 使用默认线程池
2 static CompletableFuture<Void>
3     runAsync(Runnable runnable)
4 static <U> CompletableFuture<U>
5     supplyAsync(Supplier<U> supplier)
```

```
6 // 可以指定线程池
7 static CompletableFuture<Void>
8     runAsync(Runnable runnable, Executor executor)
9 static <U> CompletableFuture<U>
10    supplyAsync(Supplier<U> supplier, Executor executor)
```

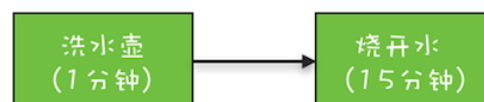
创建完 `CompletableFuture` 对象之后，会自动地异步执行 `runnable.run()` 方法或者 `supplier.get()` 方法，对于一个异步操作，你需要关注两个问题：一个是异步操作什么时候结束，另一个是如何获取异步操作的执行结果。因为 `CompletableFuture` 类实现了 `Future` 接口，所以这两个问题你都可以通过 `Future` 接口来解决。另外，`CompletableFuture` 类还实现了 `CompletionStage` 接口，这个接口内容实在是太丰富了，在 1.8 版本里有 40 个方法，这些方法我们该如何理解呢？

## 如何理解 `CompletionStage` 接口

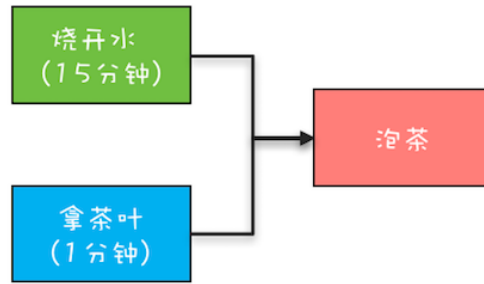
我觉得，你可以站在分工的角度类比一下工作流。任务是有时序关系的，比如有**串行关系**、**并行关系**、**汇聚关系**等。这样说可能有点抽象，这里还举前面烧水泡茶的例子，其中洗水壶和烧开水就是串行关系，洗水壶、烧开水和洗茶壶、洗茶杯这两组任务之间就是并行关系，而烧开水、拿茶叶和泡茶就是汇聚关系。



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，但却不支持返回值，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一个子流程，最终结果和 thenApply 系列是相同的。

 复制代码

```
1 CompletionStage<R> thenApply(fn);
2 CompletionStage<R> thenApplyAsync(fn);
3 CompletionStage<Void> thenAccept(consumer);
4 CompletionStage<Void> thenAcceptAsync(consumer);
5 CompletionStage<Void> thenRun(action);
6 CompletionStage<Void> thenRunAsync(action);
7 CompletionStage<R> thenCompose(fn);
8 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。

 复制代码

```
1 CompletableFuture<String> f0 =
2     CompletableFuture.supplyAsync(
3         () -> "Hello World") //①
4     .thenApply(s -> s + " QQ") //②
5     .thenApply(String::toUpperCase); //③
6
7 System.out.println(f0.join());
8 // 输出结果
9 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。

 复制代码

```
1 CompletionStage<R> thenCombine(other, fn);
2 CompletionStage<R> thenCombineAsync(other, fn);
3 CompletionStage<Void> thenAcceptBoth(other, consumer);
4 CompletionStage<Void> thenAcceptBothAsync(other, consumer);
5 CompletionStage<Void> runAfterBoth(other, action);
6 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。

 复制代码

```
1 CompletionStage applyToEither(other, fn);
2 CompletionStage applyToEitherAsync(other, fn);
3 CompletionStage acceptEither(other, consumer);
4 CompletionStage acceptEitherAsync(other, consumer);
5 CompletionStage runAfterEither(other, action);
6 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。

 复制代码

```
1 CompletableFuture<String> f1 =
2     CompletableFuture.supplyAsync(()->{
3         int t = getRandom(5, 10);
4         sleep(t, TimeUnit.SECONDS);
5         return String.valueOf(t);
6     });
7
8 CompletableFuture<String> f2 =
9     CompletableFuture.supplyAsync(()->{
10         int t = getRandom(5, 10);
11         sleep(t, TimeUnit.SECONDS);
12         return String.valueOf(t);
13     });
14
15 CompletableFuture<String> f3 =
16     f1.applyToEither(f2, s -> s);
17
```



```
18 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？

 复制代码

```
1 CompletableFuture<Integer>
2     f0 = CompletableFuture.
3         .supplyAsync(()->(7/0))
4         .thenApply(r->r*10);
5 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。

 复制代码

```
1 CompletionStage exceptionally(fn);
2 CompletionStage<R> whenComplete(consumer);
3 CompletionStage<R> whenCompleteAsync(consumer);
4 CompletionStage<R> handle(fn);
5 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。`

 复制代码

```
1 CompletableFuture<Integer>
2     f0 = CompletableFuture
3         .supplyAsync(()->7/0))
4         .thenApply(r->r*10)
5         .exceptionally(e->0);
6 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？

 复制代码

```
1 // 采购订单
2 PurchersOrder po;
3 CompletableFuture<Boolean> cf =
4     CompletableFuture.supplyAsync(()->{
5         // 在数据库中查询规则
6         return findRuleByJdbc();
7     }).thenApply(r -> {
8         // 规则校验
9         return check(po, r);
```

```
10 });  
11 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。



# Java 并发编程实战

全面系统提升你的并发编程能力

王宝令  
资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | Future：如何用多线程实现最优的“烧水泡茶”程序？

下一篇 25 | CompletionService：如何批量执行异步任务？

## 精选留言 (35)

写留言



袁阳

2019-04-23

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

展开 ▾

作者回复: ㊗️ ㊗️

◀ ▶



**密码123456**

2019-04-23

👍 8

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？

作者回复: 这个简单场景没必要用

◀ ▶



**青莲**

2019-04-23

👍 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

展开 ▾

作者回复: ㊗️ ㊗️

◀ ▶



**刘晓林**

2019-04-23

👍 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。...

展开 ▾

作者回复: 思考题考虑的很全面 ㊗️

◀ ▶



tyul

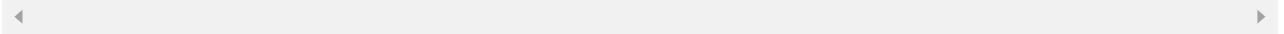
2019-04-23

👍 2

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。

展开 ▾

作者回复: 是的，复杂场景就能体现出优势了



笃行之

2019-04-29

👍 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？

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。



发条橙子 ...

2019-04-24

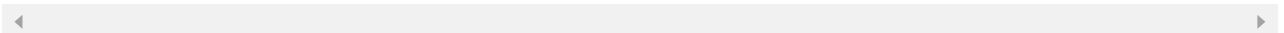
👍 1

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...

展开 ▾

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



易儿易

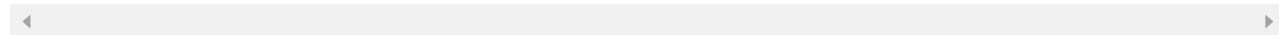
2019-04-23

👍 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.....

展开 ∨

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。



**刘晓林**

2019-04-23

👍 1

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

展开 ∨

作者回复: 后面有介绍



**linqw**

2019-04-23

👍 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 = ...
```

展开 ∨



**木木匠**

2019-04-23

👍 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?



**Michael**

2019-05-23

👍

老师 你好, 对文章点赞这种功能异步如何实现?

展开 ∨

作者回复: 喊一嗓子, 让朋友点





大卫

2019-05-19



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？...

展开 ∨



xuery

2019-05-15



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

展开 ∨



佑儿

2019-05-10



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

展开 ∨

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



Sunqc

2019-04-30



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completion service

3.就是这节的主题completion future...

展开 ∨



aroll

2019-04-29



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

作者回复: 找到原因就好



aroll

2019-04-27

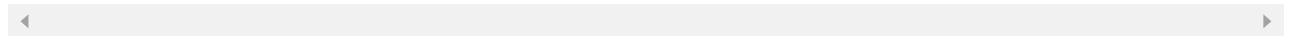


是的，启动前设置成守护线程了，就像这样

```
public static void main(String[] args){  
    Thread thread = new Thread(new Runnable() {  
        @Override  
        public void run() {...
```

展开 ∨

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？



aroll

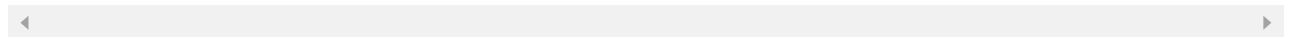
2019-04-26



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。

展开 ∨

作者回复: 启动之前设置成守护线程了？



Zach\_

2019-04-25



很喜欢这个专栏！

但是，老师说 教好学生，饿死师傅。我.....😂😂😂

展开 ✓