34 谁都不能偷懒-通过 CompletableFuture 组装你的异步计算单元

更新时间：2019-12-24 10:04:04

> 不想当将军的士兵，不是好士兵。
>
> ——拿破仑

本节是我在写专栏过程中临时决定加入的，之前考虑 CompletableFuture 的使用需要结合 lambda 表达式以及 stream 的思想，对于初学者有些困难。但是 CompletableFuture 自 java 8 引入后，实际开发中使用还是比较多的，还是决定写一节 CompletableFuture 的使用。

一些比较复杂的异步计算场景，尤其是需要串联多个异步计算单元的场景，可以考虑使用 CompletableFuture 来实现。如果你熟悉 Stream 以及 lambda，学习使用 CompletableFuture 会比较简单。如果没有接触过 Stream 可能理解上会有一点困难。不过没有关系，我们集中注意力在 CompletableFuture 本身上，跟着本节讲解的思路，自己多做练习，相信你肯定能够融会贯通，灵活运用。

1、CompletableFuture 介绍

CompletableFuture 作为 Java 8 的新特性被引入。任何工具的出现肯定带着自己的使命，那么它是用来解决什么问题的呢？

在现实世界中，我们需要解决的复杂问题都是要分为若干步骤。就像我们的代码一样，一个复杂的逻辑方法中，会调用多个方法来一步一步实现。

设想如下场景，植树节要进行植树，分为下面几个步骤：

1、挖坑 10 分钟

2、拿树苗 5 分钟

3、种树苗 20 分钟

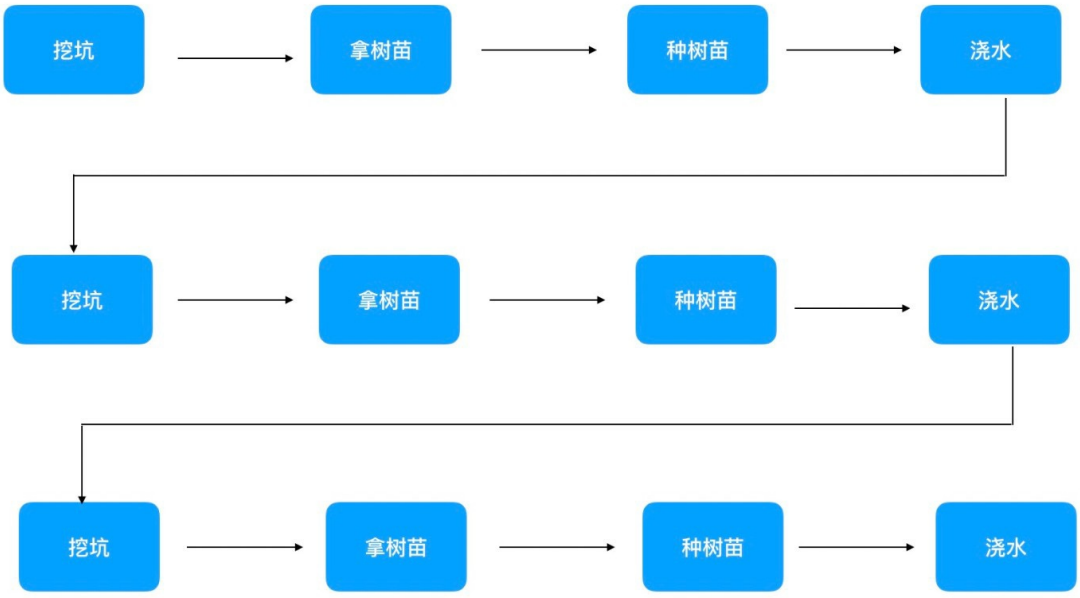
4、浇水 5 分钟

其中 1 和 2 可以并行，1 和 2 都完成了才能进行步骤 3，然后才能进行步骤 4。

我们有如下几种实现方式：

1、只有一个人种树

如果现在只有一个人植树，要种 100 棵树，那么只能按照如下顺序执行：



图中仅列举种三棵树示意。可以看到串行执行，只能种完一棵树再种一棵，那么种完 100 棵树需要 40 * 100 = 4000 分钟。

这种方式对应到程序，就是单线程同步执行。

2、三个人同时种树，每个人负责种一棵树

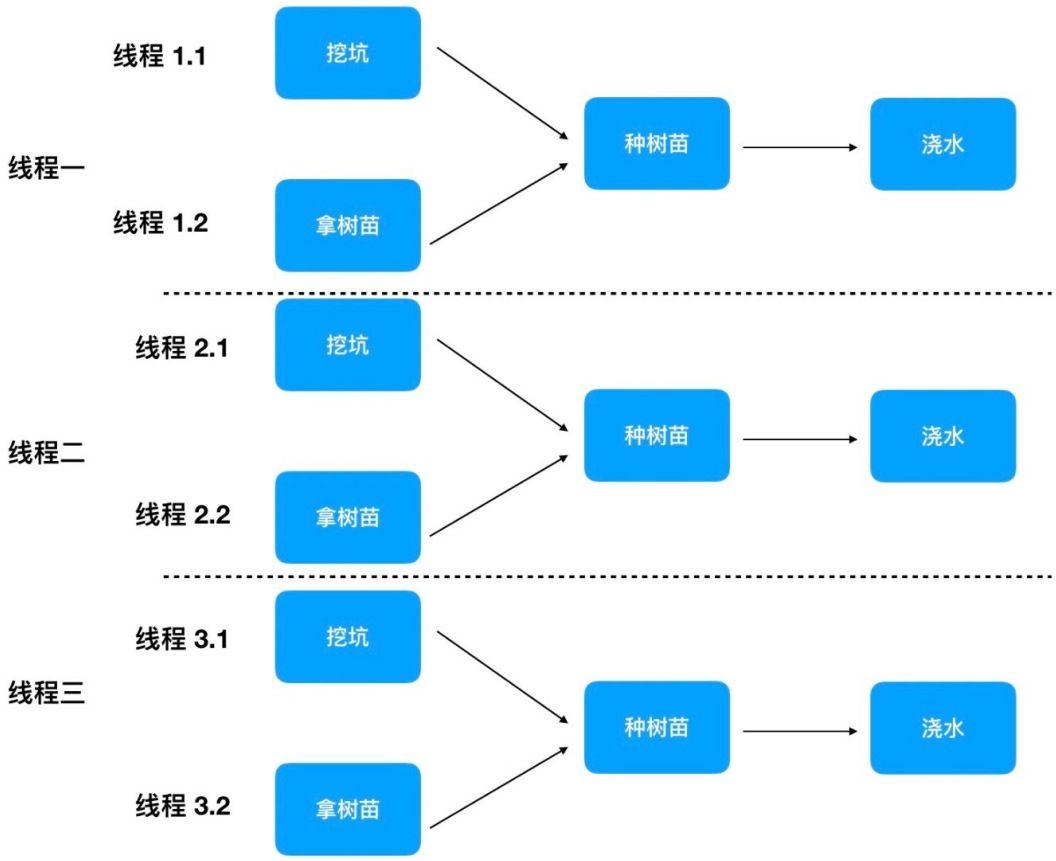
如何缩短种树时长呢？你肯定想这还不好办，学习了这么久的并发，这肯定难不倒我。不是要种 100 棵树吗？那我找 100 个人一块种，每个人种一棵。那么只需要 40 分钟就可以种完 100 棵树了。

没错，如果你的程序有个方法叫做 plantTree，里面包含了如上四部，那么你起 100 个线程就可以了。但是，请注意，100 个线程的创建和销毁需要消耗大量的系统资源。并且创建和销毁线程都有时间消耗。此外CPU的核数并不能真的支持100个线程并发。如果我们要种1万棵树呢？总不能起一万个线程吧？

所以这只是理想情况，我们一般是通过线程池来执行，并不会真的启动100个线程。

3、多个人同时种树。种每一棵树的时候，不依赖的步骤可以分不同的人并行干

这种方式可以进一步缩短种树的时长，因为第一步挖坑和第二步拿树苗可以两个人并行去做，所以每棵树只需要35分钟。如下图：



如果程序还是 100 个主线程并发运行 plantTree 方法，那么只需要 35 分钟种完 100 颗树。

这里需要注意每个线程中，由于还要并发两个线程去做 1，2 两个步骤。实际运行中会又 100*3 = 300 个线程参与植树。但是负责 1，2 步骤的线程只会短暂参与，然后就闲置了。

这种方法和第二种方式也存在大量创建线程的问题。所以也只是理想情况。

4、假如只有 4 个人植树，每个人只负责自己的步骤，那么执行如下图

可以看到一开始小王挖完第一个坑后，小李已经取回两个树苗，但此时小张才能开始种第一个树苗。此后小张就可以一个接一个的去种树苗了，并且在他种下一棵树苗的时候，小赵可以并行浇水。按照这个流程走下来，种完 100 颗树苗需要 10+20x100+5=2015 分钟。比单线程的4000分钟好了很多，但是远远比不上 100 个线程并发种树的速度。不过不要忘记 100 个线程并发只是理想情况，而本方法只用了 4 个线程。

我们再对分工做下调整。每个人不只干自己的工作，一旦自己的工作做完了就看有没有其他工作可以做。比如小王挖坑完后，发现可以种树苗，那么他就去种树苗。小李拿树苗完成后也可以去挖坑或者种树苗。这样整体的效率就会更高了。如果基于这种思想，那么我们实际上把任务分成了 4 类，每类 100 件，一共 400 件任务。400 件任务全部完成，意味着整个任务就完成了。那么任务的参与者只需要知道任务的依赖，然后不断领取可以执行的任务去执行。这样的效率将会是最高的。

前文说到我们不可能通过100个线程并发来执行任务，所以一般情况下我们都会使用线程池，这和上面的设计思想不谋而合。使用线程池后，由于第四种方式把步骤拆的更细，提高了并发的可能性。因此速度会比第二种方式更快。那么和第三种比起来，哪种更快呢？如果线程数量可以无穷大，这两个方法能达到的最短时间是一样的，都是 35 分钟。不过在线程有限的情况下，第四种方式对线程的使用率会更高，因为每个步骤都可以并行执行（参与种树的人完成自己的工作后，都可以去帮助其他人），线程的调度更为灵活，所以线程池中的线程很难闲下来，一直保持在运转之中。是的，谁都不能偷懒。而第三种由于只能并发在 plantTree 方法及挖坑和拿树苗，所以不如第四种方式灵活。

上文讲了这么多，主要是要说明 CompletableFuture 出现的原因。他用来把复杂任务拆解为一个个衔接的异步执行步骤，从而提升整体的效率。我们回一下小节题目：谁都不能偷懒。没错，这就是 CompletableFuture 要达到的效果，通过对计算单元的抽象，让线程能够高效的并发参与每一个步骤。同步的代码通过 CompletableFuture 可以完全改造为异步代码。下面我们就来看看如何使用 CompletableFuture。

2、CompletableFuture 介绍

CompletableFuture 实现了 Future 接口并且实现了 CompletionStage 接口。Future 接口我们已经很熟悉了，而 CompletionStage 接口定了异步计算步骤之间的规范，这样确保一步一步能够衔接上。CompletionStage 定义了38个 public 的方法用于异步计算步骤间的衔接。接下来我们会挑选一些常用的，相对使用频率较高的方法，来看看如何使用。

2.1 已知计算结果

如果你已经知道 CompletableFuture 的计算结果，可以使用静态方法 completedFuture。传入计算结果，声明 CompletableFuture 对象。在调用 get 方法时会立即返回传入的计算结果，不会被阻塞，如下代码：

```
public static void noComputation() throws ExecutionException, InterruptedException {
    CompletableFuture<String> completableFuture
            = CompletableFuture.completedFuture("hello world");

    System.out.println("result is " + completableFuture.get());
}

public static void main(String[] args) throws ExecutionException, InterruptedException {
    noComputation();
}
```

输出为：

```
result is hello world
```

是不是觉得这种用法没有什么意义？既然知道计算结果了，直接使用就好了，为什么还要通过 CompletableFuture 进行包装？这是因为异步计算单元需要通过 CompletableFuture 进行衔接，所以有的时候我们即使已经知道计算结果，也需要包装为 CompletableFuture，才能融入到异步计算的流程之中。

2.2 封装有返回值的异步计算逻辑

这是我们最常用的方式。把需要异步计算的逻辑封装为一个计算单元，交由 CompletableFuture 去运行。如下面的代码：

```
public static void supplyAsync() throws ExecutionException, InterruptedException {
    CompletableFuture<String> completableFuture
            = CompletableFuture.supplyAsync(() -> "挖坑完成");

    System.out.println("result is " + completableFuture.get());
}


public static void main(String[] args) throws ExecutionException, InterruptedException {
    supplyAsync();
}
```

这里我们使用了 CompletableFuture 的 supplyAsync 方法，以 lambda 表达式的方式向其传递了一个 supplier 接口的实现。supplier 是只有一个方法的函数接口，这里使用的就是常说的函数式编程。关于函数式编程并不在本专栏讨论范围内，这里你只需要知道我们为 supplyAsync 方法传入了一个可执行的函数，而 “Hello world” 就是这段函数的返回值。我们运行后结果如下：

```
result is 挖坑完成
```

可见 completableFuture.get() 拿到的计算结果就是你传入函数执行后 return 的值。那么如果你有需要异步计算的逻辑，那么就可以放到 supplyAsync 传入的函数体中。这段函数是如何被异步执行的呢？如果你跟入代码可以看到其实 supplyAsync 是通过 Executor，也就是线程池来运行这段函数的。completableFuture 默认使用的是 ForkJoinPool，当然你也可以通过为 supplyAsync 指定其他 Excutor，通过第二个参数传入 supplyAsync 方法。

supplyAsync 使用场景非常多，举个简单的例子，主程序需要调用多个微服务的接口请求数据，那么就可以启动多个 CompletableFuture，调用 supplyAsync，函数体中是关于不同接口的调用逻辑。这样不同的接口请求就可以异步同时运行，最后再等全部接口返回时，执行后面的逻辑。

2.3 封装无返回值的异步计算逻辑

supplyAsync 接收的函数是有返回值的。有些情况我们只是一段计算过程，并不需要返回值。这就像 Runnable 的 run 方法，并没有返回值。这种情况我们可以使用 runAsync方法，如下面的代码：

```
public static void runAsync() throws ExecutionException, InterruptedException {
    CompletableFuture<Void> completableFuture
            = CompletableFuture.runAsync(() -> System.out.println("挖坑完成"));

    completableFuture.get();
}


public static void main(String[] args) throws ExecutionException, InterruptedException {
    supplyAsync();
}
```

runAsync 接收 runnable 接口的函数。所以并无返回值。栗子中的逻辑只是打印“挖坑完成”。

2.4 进一步处理异步返回的结果，并返回新的计算结果

当我们通过 supplyAsync 完成了异步计算，返回 CompletableFuture，此时可以继续对返回结果进行加工，如下面的代码：

```
public static void thenApply() throws ExecutionException, InterruptedException {
    CompletableFuture<String> completableFuture
            = CompletableFuture.supplyAsync(() -> "挖坑完成")
            .thenApply(s->s+",并且归还铁锹")
            .thenApply(s->s+"，全部完成。");

    System.out.println("result is " + completableFuture.get());
}


public static void main(String[] args) throws ExecutionException, InterruptedException {
    thenApply();
}
```

在调用 supplyAsync 后，我们两次链式调用 thenApply 方法。s 是前一步 supplyAsync 返回的计算结结果，我们对结算结果进行了两次再加工，输出如下：

```
result is 挖坑完成,并且归还铁锹，全部完成。
```

我们可以通过 thenApply 不断对计算结果进行加工处理。

如果想异步运行 thenApply 的逻辑，可以使用 thenApplyAsync。使用方法 xiangtong1，只不过会通过线程池异步运行.

2.5 进一步处理异步返回的结果，无返回

这种场景你可以使用thenApply。这个方法可以让你处理上一步的返回结果，但无返回值。参照如下代码：

```
public static void thenAccept() throws ExecutionException, InterruptedException {
    CompletableFuture<Void> completableFuture
            = CompletableFuture.supplyAsync(() -> "挖坑完成")
            .thenAccept(s-> System.out.println(s+",并且归还铁锹"));
    completableFuture.get();
}


public static void main(String[] args) throws ExecutionException, InterruptedException {
    thenAccept();
}
```

这里可以看到 thenAccept 接收的函数没有返回值，只有业务逻辑。处理后返回 CompletableFuture 类型对象。

2.6 既不需要返回值，也不需要上一步计算结果，只想在执行结束后再执行一段代码

此时你可以使用 thenRun 方法，他接收 Runnable 的函数，没有输入也没有输出，仅仅是在异步计算结束后回调一段逻辑，比如记录 log 等。参照下面代码：

```
public static void thenRun() throws ExecutionException, InterruptedException {
    CompletableFuture<Void> completableFuture
            = CompletableFuture.supplyAsync(() -> "挖坑完成")
            .thenAccept(s-> System.out.println(s+",并且归还铁锹"))
            .thenRun(()-> System.out.println("挖坑工作已经全部完成"));

    completableFuture.get();
}

public static void main(String[] args) throws ExecutionException, InterruptedException {
    thenRun();
}
```

可以看到在 thenAccept 之后继续调用了 thenRun，仅仅是打印了日志而已，输出如下：

```
挖坑完成,并且归还铁锹
挖坑工作已经全部完成
```

2.7 组合 Future 处理逻辑

我们可以把两个 CompletableFuture 组合起来使用，如下面的代码：

```
public static void thenCompose() throws ExecutionException, InterruptedException {
    CompletableFuture<String> completableFuture
            = CompletableFuture.supplyAsync(() -> "挖坑完成")
            .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + " 并且归还铁锹"));

    System.out.println("result is " + completableFuture.get());
}
```

运行结果

```
result is 挖坑完成 并且归还铁锹
```

thenApply 和 thenCompose 的关系就像 stream中的 map 和 flatmap。从上面的例子来看，thenApply 和 thenCompose 都可以实现同样的功能。但是如果你使用一个第三方的库，有一个API返回的是CompletableFuture类型，那么你就只能使用 thenCompose方法。

2.8 组合Futurue结果

如果你有两个异步操作互相没有依赖，但是第三步操作依赖前两部计算的结果，那么你可以使用 thenCombine 方法来实现，如下面代码：

```
public static void thenCombine() throws ExecutionException, InterruptedException {
    CompletableFuture<String> completableFuture
            = CompletableFuture.supplyAsync(() -> "挖坑完成。")
            .thenCombine(CompletableFuture.supplyAsync(() -> "拿树苗完成。"),
                    (a,b)-> a+b+"植树完成。");

    System.out.println("result is " + completableFuture.get());
}

public static void main(String[] args) throws ExecutionException, InterruptedException {
    thenCombine();
}
```

挖坑和拿树苗可以同时进行，但是第三步植树则祖尧前两步完成后才能进行。执行结果如下：

```
result is 挖坑完成。拿树苗完成。植树完成。
```

可以看到符合我们的预期。使用场景之前也提到过。我们调用多个微服务的接口时，可以使用这种方式进行组合。处理接口调用间的依赖关系。

当你需要两个 Future 的结果，但是不需要再加工后向下游传递计算结果时，可以使用 thenAcceptBoth，用法一样，只不过接收的函数没有返回值。

2.9 并行处理多个 Future

假如我们对微服务接口的调用不止两个，并且还有一些其它可以异步执行的逻辑。主流程需要等待这些所有的异步操作都返回时，才能继续往下执行。此时我们可以使用 *CompletableFuture.allOf* 方法。它接收 n 个 CompletableFuture，返回一个 CompletableFuture。对其调用 get 方法后，只有所有的 CompletableFuture 全完成时才会继续后面的逻辑。我们看下面示例代码：

```
public static void allOf() throws ExecutionException, InterruptedException {
    CompletableFuture<Void> future1 = CompletableFuture.runAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("挖坑完成");
    });
    CompletableFuture<Void> future2 = CompletableFuture.runAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("取树苗完成");
    });
    CompletableFuture<Void> future3 = CompletableFuture.runAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("取肥料完成");
    });

    CompletableFuture.allOf(future1,future2,future3).get();

    System.out.println("植树准备工作完成！");

}

public static void main(String[] args) throws ExecutionException, InterruptedException {
    allOf();
}
```

输出结果为：

```
挖坑完成
取肥料完成
取树苗完成
植树准备工作完成!
```

可以看到三个 CompletableFuture 全部完成后，才会打印“植树准备工作完成！”。

2.10 异常处理

在异步计算链中的异常处理可以采用 handle 方法，它接收两个参数，第一个参数是计算及过，第二个参数是异步计算链中抛出的异常。使用方法如下：

```
public static void errorHandling() throws ExecutionException, InterruptedException {
    CompletableFuture<String> completableFuture
            = CompletableFuture
            .supplyAsync(() -> {
                if (1 == 1) {
                    throw new RuntimeException("Computation error!");
                }

                return "挖坑完成";
            })
            .handle((result, throwable) -> {
                if (result == null) {
                    return "挖坑异常";
                }
                return result;
            });

    System.out.println("result is " + completableFuture.get());
}

public static void main(String[] args) throws ExecutionException, InterruptedException {
    errorHandling();
}
```

代码中会抛出一个 RuntimeException，抛出这个异常时 result 为 null，而 throwable 不为null。根据这些信息你可以在 handle 中进行处理，如果抛出的异常种类很多，你可以判断 throwable 的类型，来选择不同的处理逻辑。

3、总结

本节我们学习了 CompletableFuture 的常见用法，它的方法远不止这些，其它的方法大家可以参照文档进行学习。在实际开发中，我推荐使用 CompletableFuture 进行异步计算，它更为灵活，并且可以采用 lambda 表达式进行函数式编程，代码更为简洁，可读性也更高。

}

33 分阶段执行你的任务-学习使用 Phaser运行多阶段任务

35拆分你的任务—学习使用 Fork/Join框架