CompletableFuture 工具类

介绍

Java 8 提供的工具类，实现了 Future 接口和 CompletionStage 接口。拥有 40 多种方法，为函数式编程中的流式调用而准备。

支持流式计算，函数式编程，完成通知，自定义异常处理等特性。

和 Future 一样，可以作为函数调用的契约。调用线程调用 CompletableFuture.supplyAsync()创建一个新线程，请求线程可以先处理其他任务。如果其他任务完成后调用 CompletableFuture.get()方法获取结果。如果数据还没有准备好，就会等待。

在 Java 8 之前，如果使用实现 Runnable 的 Run 方法实现多线程，问题是 Run()没有返回值。

如果使用 Callable 的 Call 方法，再用 Future 的 get()获取返回值，会造成主线程的阻塞。

CompletableFuture 的用法

以下带 Async 后缀的函数表示需要连接的后置任务会被单独提交到线程池中，从而相对前置任务来说是异步运行的。除此之外，两者没有其他区别。

创建 CompleteableFuture

构造函数创建

最简单的方式就是通过构造函数创建一个 CompletableFuture 实例。

```
CompletableFuture<String> future = new CompletableFuture();
String result = future.join();
System.out.println(result);
```

由于新创建的 CompletableFuture 还没有任何计算结果，这时调用 join，当前线程会一直阻塞在这里。

此时，如果在另外一个线程中，主动设置该 CompletableFuture 的值，则上面线程中的结果就能返回。

```
future.complete("test");
```

supplyAsync 创建

通过该函数创建的 CompletableFuture 实例会异步执行当前传入的计算任务。在调用端，则可以通过 get()或 join()获取最终计算结果。

```
CompletableFuture<String> future
        = CompletableFuture.supplyAsync(()->{
      System.out.println("compute test");
      return "test";
});

String result = future.join();
System.out.println("get result: " + result);
```

join()和 get()方法都是阻塞调用它们的线程（通常为主线程）来获取 CompletableFuture 异步之后的返回值。

CompletableFuture.get() 和 CompletableFuture.join() 这两个方法是获取异步守护线程的返回值的。

不同点：

get()方法会抛出**经检查的异常**，可被捕获，自定义处理或者直接抛出。

join()方法会抛出**未经检查的异常**。

runAsync 创建

与 supplyAsync()不同的是，runAsync()传入的任务要求是 Runnable 类型的，所以没有返回值。因此，runAsync 适合创建不需要返回值的计算任务。

```
CompletableFuture<Void> future = CompletableFuture.runAsync(()->{
            System.out.println("compute test");
        });

System.out.println("get result: " + future.join());
```

由于任务没有返回值， 所以最后的打印结果是"get result: null"。

简单用法 get() 与 complete()

get() 获取任务结果

在主线程中调用 get()方法会阻塞主线程。

get：获取任务返回值，没有则阻塞直到有返回值返回；抛出经过检查的异常 InterruptedException, ExecutionException

getNow：如果结果已经计算完则返回结果或者抛出异常，否则返回给定的 valueIfAbsent 值；

join：获取任务返回值，没有则阻塞直到有返回值返回；抛出未经检查的异常，不会强制抛出，会将异常包装成 CompletionException/CancellationException 异常；

join()与 get()区别

join()返回计算的结果或者抛出一个 unchecked 异常(CompletionException)，而 get()返回一个具体的异常。另外 get() 可以指定超时时间。

complete() 手动完成任务

手动完成任务。所有等待这个 Future 的客户端都将得到指定的结果。并且对 completableFuture.complete()的后续调用将被忽略。

提交任务 runAsync() 与 supplyAsync()

runAsync()

异步调用没有返回值。

supplyAsync()

异步调用有返回值。

任务编排

链式处理 thenRun()、thenAccept() 和 thenApply()

需要注意的是，通过 thenApply / thenAccept / thenRun 连接的任务，当且仅当前置任务计算完成时，才会开始后置任务的计算。因此，这组函数主要用于连接前后有依赖的任务链。

thenRun() 无入参无返回

thenRun 提交的任务类型需遵从 Runnable 签名，即没有入参也没有返回值。

thenAccept() 有入参无返回

thenAccept 提交的任务类型需遵从 Consumer 签名，也就是有入参但是没有返回值，其中入参为前置任务的结果。

thenApply() 有入参有返回

thenApply 提交的任务类型需遵从 Function 签名，也就是有入参和返回值，其中入参为前置任务的结果。

thenApply 和 thenCompose 的区别

public <U> CompletionStage<U> thenApply(Function<? super T,? extends U> fn);

public <U> CompletionStage<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> fn);

Function<? super T,? extends U>

T：上一个任务返回结果的类型

U：当前任务的返回值类型

两个方法的返回值都是 CompletionStage<U>，不同之处在于它们的传入参数 fn.

对于 thenApply，fn 函数是一个对一个已完成的 stage 或者说 CompletableFuture 的返回值进行计算、操作。

对于 thenCompose，fn 函数是对另一个 CompletableFuture 进行计算、操作。

```
CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() ->
123).thenApply(num -> "thenApply 任务:" + num);
CompletableFuture<String> f2 = CompletableFuture.supplyAsync(() ->
456).thenCompose(num -> CompletableFuture.supplyAsync(() -> "thenCompose 任
务:" + num));

System.out.println(f1.join());
System.out.println(f2.join());
```

组合任务 thenCompose() 与 thenCombine() - 都完成

thenCompose()

thenCompose 主要用于有前后依赖关系之间的任务进行连接。

合并两个有依赖关系的 CompletableFutures 的执行结果。组合两个 future，并将前一个任务的返回结果作为下一个任务的参数，存在先后顺序。

```
private static Integer num = 10;

    public static void main(String[] args) throws Exception {
        System.out.println("主线程开始");
        //第一步加 10
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(()
-> {
            System.out.println("加 10 任务开始");
            num += 10;
            return num;
```

```
        });
          //合并
        CompletableFuture<Integer> future1 = future.thenCompose(i ->
        //再来一个 CompletableFuture
                CompletableFuture.supplyAsync(() -> {
                    return i + 1;
                }));
        System.out.println(future.get());
        System.out.println(future1.get());
    }
```

thenCombine() (组合两个 future，获取它们的返回值，有返回值)

thenCombine 主要用于没有前后依赖关系之间的任务进行连接。

组合两个 future，获取它们的返回值，并返回当前任务的返回值。

```
private static Integer num = 10;

    public static void main(String[] args) throws Exception {
        System.out.println("主线程开始");
        CompletableFuture<Integer> job1 = CompletableFuture.supplyAsync(() ->
{
            System.out.println("加 10 任务开始");
            num += 10;
            return num;
        });
        CompletableFuture<Integer> job2 = CompletableFuture.supplyAsync(() ->
{
            System.out.println("乘以 10 任务开始");
            num = num * 10;
            return num;
        });
        //合并两个结果
        CompletableFuture<Object> future = job1.thenCombine(job2, new
                BiFunction<Integer, Integer, List<Integer>>() {
                    @Override
                    public List<Integer> apply(Integer a, Integer b) {
                        List<Integer> list = new ArrayList<>();
                        list.add(a);
```

```
                    list.add(b);
                    return list;
                }
            });
        System.out.println("合并结果为:" + future.get());
    }
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
20:46:56.557 [main] INFO com.txw.ms.sample.service.TestCompletableFuture - main thread start...
20:46:56.688 [ForkJoinPool.commonPool-worker-2] INFO com.txw.ms.sample.service.TestCompletableFuture - jobB start...
20:46:56.688 [ForkJoinPool.commonPool-worker-1] INFO com.txw.ms.sample.service.TestCompletableFuture - jobA start...
20:46:56.689 [main] INFO com.txw.ms.sample.service.TestCompletableFuture - thenCombine return: [150, 15]
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
20:47:34.731 [main] INFO com.txw.ms.sample.service.TestCompletableFuture - main thread start...
20:47:34.877 [ForkJoinPool.commonPool-worker-1] INFO com.txw.ms.sample.service.TestCompletableFuture - jobA start...
20:47:34.878 [ForkJoinPool.commonPool-worker-1] INFO com.txw.ms.sample.service.TestCompletableFuture - jobB start...
20:47:34.878 [main] INFO com.txw.ms.sample.service.TestCompletableFuture - thenCombine return: [50, 60]
```

注意：上例代码可能输出不同结果，因为 jobA 和 jobB 并发执行且顺序随机，而且操作了同一个静态数据。

thenAcceptBoth（组合两个 future，获取它们的执行结果，没有返回值）

```
private static void test11thenAcceptBoth() throws ExecutionException,
InterruptedException {
    CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {
        int number1 = new Random().nextInt(3) + 1;
        try {
            TimeUnit.SECONDS.sleep(number1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("第一阶段：" + number1);
        return number1;
    });

    CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {
        int number2 = new Random().nextInt(3) + 1;
        try {
            TimeUnit.SECONDS.sleep(number2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("第二阶段：" + number2);
        return number2;
```

```
    });

    CompletableFuture job = future1.thenAcceptBoth(future2, (num1, num2) -> {
        System.out.println("最终结果：" + (num1 + num2));
    });

    job.get();
}
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
第一阶段：1
第二阶段：2
最终结果：3
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
第二阶段：1
第一阶段：2
最终结果：3
```

注意：如果 future1.get()则只会运行 future1。

future1 和 future2 哪个先执行是随机的。

CompletableFuture 使用详解 - 周文豪 - 博客园 (cnblogs.com)

runAfterBoth（组合两个 future，不获取它们的执行结果，没有返回值）

```
private static void test12runAfterBoth() throws ExecutionException,
InterruptedException {
    CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> {
        int number1 = new Random().nextInt(3) + 1;
        System.out.println("第一阶段：" + number1);
        try {
            TimeUnit.SECONDS.sleep(number1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return number1;
    });

    CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> {
```

```
        int number2 = new Random().nextInt(3) + 1;
        System.out.println("第二阶段：" + number2);
        try {
            TimeUnit.SECONDS.sleep(number2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return number2;
    });

    CompletableFuture job = future1.runAfterBoth(future2, () -> {
        System.out.println("两个任务完成。");
    });

    job.get();
}
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
第一阶段：2
第二阶段：2
两个任务完成。
```

thenCombine / thenAcceptBoth / runAfterBoth 都是组合的任务互相不依赖

这三个方法都是将两个 CompletableFuture 组合起来，只有这两个都正常执行完了才会执行某个任务。

区别在于，

thenCombine 会将两个任务的执行结果作为方法入参传递到指定方法中，且该方法有返回值；

thenAcceptBoth 同样将两个任务的执行结果作为方法入参，但是无返回值；

runAfterBoth 没有入参，也没有返回值。注意两个任务中只要有一个执行异常，则将该异常信息作为指定任务的执行结果。

组合任务 applyToEither, acceptEither, runAfterEither - 任意一个完成

applyToEither / acceptEither / runAfterEither

这三个方法都是将两个 CompletableFuture 组合起来，只要其中一个执行完了就会执行某个任务（其他线程依然会继续执行）。

其区别在于

applyToEither 会将已经执行完成的任务的执行结果作为方法入参，并有返回值；

acceptEither 同样将已经执行完成的任务的执行结果作为方法入参，但是没有返回值；

runAfterEither 没有方法入参，也没有返回值。注意两个任务中只要有一个执行异常，则将该异常信息作为指定任务的执行结果。

多任务组合 allOf()与 anyOf()

allOf()

allOf 返回的 CompletableFuture 是多个任务都执行完成后才会执行，只要有一个任务执行异常，则返回的 CompletableFuture 执行 get 方法时会抛出异常，如果都是正常执行，则 get 返回 null。

```
private static void test9allOf() throws ExecutionException,
InterruptedException {
    CompletableFuture<Integer> jobA = CompletableFuture.supplyAsync(() -> {
        log.info("Start job: " + Thread.currentThread().getName());
        log.info("End job: " + Thread.currentThread().getName());
        return num;
    });
    CompletableFuture<Integer> jobB = CompletableFuture.supplyAsync(() -> {
        log.info("Start job: " + Thread.currentThread().getName());
        log.info("End job: " + Thread.currentThread().getName());
        return num;
    });
    CompletableFuture<Integer> jobC = CompletableFuture.supplyAsync(() -> {
        log.info("Start job: " + Thread.currentThread().getName());
        log.info("End job: " + Thread.currentThread().getName());
        return num;
    });

    CompletableFuture job = CompletableFuture.allOf(jobA, jobB, jobC)
            .whenComplete((result, ex) -> {
```

```
                if (ex != null) {
                    log.info("exception: " + ex.getMessage());
                } else {
                    log.info("result: " + result);
                }
            });

    log.info("return: " + jobA.get());
}
```

注意：allOf()本身并不会触发任务执行，调用 jobA.get()或者 job.get()都会触发所有任务执行。

anyOf()

anyOf 返回的 CompletableFuture 是多个任务只要其中一个执行完成就会执行，其 get 返回的是已经执行完成的任务的执行结果，如果该任务执行异常，则抛出异常。

```
private static void test10anyOf() throws ExecutionException,
InterruptedException {
    CompletableFuture<Integer> jobA = CompletableFuture.supplyAsync(() -> {
        log.info("Start job: " + Thread.currentThread().getName());
        log.info("End job: " + Thread.currentThread().getName());
        return 5;
    });
    CompletableFuture<Integer> jobB = CompletableFuture.supplyAsync(() -> {
        log.info("Start job: " + Thread.currentThread().getName());
        log.info("End job: " + Thread.currentThread().getName());
        return 6;
    });
    CompletableFuture<Integer> jobC = CompletableFuture.supplyAsync(() -> {
        log.info("Start job: " + Thread.currentThread().getName());
        log.info("End job: " + Thread.currentThread().getName());
        return 7;
    });

    CompletableFuture job = CompletableFuture.anyOf(jobA, jobB, jobC)
            .whenComplete((result, ex) -> {
                if (ex != null) {
                    log.info("exception: " + ex.getMessage());
```

```
            } else {
                log.info("result: " + result);
            }
        });

    log.info("return: " + job.get());
}
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
21:31:46.625 [ForkJoinPool.commonPool-worker-2] INFO com.txw.ms.sample.service.TestCompletableFuture - Start job: ForkJoinPool.commonPool-worker-2
21:31:46.629 [ForkJoinPool.commonPool-worker-2] INFO com.txw.ms.sample.service.TestCompletableFuture - End job: ForkJoinPool.commonPool-worker-2
21:31:46.629 [ForkJoinPool.commonPool-worker-2] INFO com.txw.ms.sample.service.TestCompletableFuture - result: 6
21:31:46.624 [ForkJoinPool.commonPool-worker-1] INFO com.txw.ms.sample.service.TestCompletableFuture - Start job: ForkJoinPool.commonPool-worker-1
21:31:46.629 [ForkJoinPool.commonPool-worker-1] INFO com.txw.ms.sample.service.TestCompletableFuture - End job: ForkJoinPool.commonPool-worker-1
21:31:46.629 [main] INFO com.txw.ms.sample.service.TestCompletableFuture - return: 6
```

任务编排总结

串行任务

thenRun，任务 A 完成后继续任务 B，任务 B 不处理任务 A 的结果，无返回。

thenAccept，任务 A 完成后继续任务 B，任务 B 处理任务 A 的结果，无返回。

thenApply，任务 A 完成后继续任务 B，任务 B 处理任务 A 的结果，并返回。

组合任务-都完成

thenCompose，任务 A 和任务 B 有依赖关系，任务 A 完成后继续任务 B，任务 B 处理任务 A 的结果，并返回。

thenCombine，任务 A 和任务 B 没有依赖关系，**同时进行**，两个任务的执行结果作为 thenCombine 定义的 apply 方法的入参，有返回。

thenAcceptBoth，任务 A 和任务 B 没有依赖关系，**同时进行**，获取任务的执行结果，没有返回值。

runAfterBoth ，任务 A 和任务 B 没有依赖关系，**同时进行**，不需要获取执行结果，没有返回值。

组合任务-任一完成

applyToEither，任务 A 和任务 B 没有依赖关系，**同时进行**，已经执行完成的任务的执行结果作为方法入参，并有返回值；

acceptEither，任务 A 和任务 B 没有依赖关系，**同时进行**，将已经执行完成的任务的执行结果作为方法入参，没有返回值；

runAfterEither，任务 A 和任务 B 没有依赖关系，**同时进行**，没有方法入参，也没有返回值。注意两个任务中只要有一个执行异常，则将该异常信息作为指定任务的执行结果。

多任务组合

allOf：多个任务**同时进行**，任务全部完成后返回 CompletableFuture 对象，调用该对象的 get 方法返回 null。

anyOf：多个任务**同时进行**，其中一个任务完成后返回 CompletableFuture 对象。

异常处理 exceptionall，whenComplete，handle

exceptionally 有返回

异常处理，任务出现异常时触发。

exceptionally 方法指定某个任务执行异常时执行的回调方法，会将抛出异常作为参数传递到回调方法中。

```
    private static Integer num = 10;

    public static void main(String[] args) throws Exception {
        System.out.println("主线程开始");
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(()
-> {
            int i = 1 / 0;
            System.out.println("加 10 任务开始");
            num += 10;
            return num;
        }).exceptionally(ex -> {
            //异常内容输出
            System.out.println(ex.getMessage());
            return -1;
        });
        System.out.println(future.get());
    }
```

使用方式类似于 try catch 中的 catch 代码块中异常处理；当某个任务执行异常时将抛出的异常作为参数传递到回调方法中。

比如 supplyAsync().thenApply().exceptionally()，如果 sypplyAsync()发生异常，会被 exceptionally()捕获处理，而且 thenApply().不再执行。

whenComplete()

whenComplete 主要用于注入任务完成时的回调通知逻辑。这个解决了传统 future 在任务完成时，无法主动发起通知的问题。前置任务会将计算结果或者抛出的异常作为入参传递给回调通知函数。

不论上一个阶段是正常/异常完成(即不会对阶段的原来结果产生影响)；**类似于 try..catch..finanlly 中 finally 代码块**，无论是否发生异常都将会执行的。

当某个任务执行完成后，会将执行结果或者执行期间抛出的异常传递给回调方法：

如果是正常执行则异常为 null，回调方法对应的 CompletableFuture 的 result 和该任务一致，

如果该任务正常执行，则 get 方法返回执行结果，如果是执行异常，则 get 方法抛出异常。

```
public static void main(String[] args) throws Exception {
    CompletableFuture<Integer> futureA = CompletableFuture.supplyAsync(() ->
10 / 0)
 .thenApply(s -> {
 System.out.println("执行了");
                return s + 1;
            })
 .whenComplete((r, e) -> {
 if (r != null && e == null) {
 System.out.println("正常执行");
                }
 if (e != null) {
 System.out.println("执行异常：" + e.getMessage());
                }
 });
    System.out.println(futureA.get());
}
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
执行异常：java.lang.ArithmeticException: / by zero
Exception in thread "main" java.util.concurrent.ExecutionException Create breakpoint : java.lang.ArithmeticException: / by zero <2 internal lines>
    at com.txw.ms.sample.service.TestCompletableFuture.main(TestCompletableFuture.java:33)
Caused by: java.lang.ArithmeticException Create breakpoint : / by zero
    at com.txw.ms.sample.service.TestCompletableFuture.lambda$main$0(TestCompletableFuture.java:20) <6 internal lines>

Process finished with exit code 1
```

出现异常后，不再执行 thenApply，执行了 whenComplete，但是仍然抛出异常，因为没有 exceptionally。

```
private static void test4() throws ExecutionException, InterruptedException {
 CompletableFuture<Integer> futureA = CompletableFuture.supplyAsync(() ->
10/0)
 .thenApply(s -> {
 System.out.println("执行了");
                return s + 1;
            }).exceptionally(e -> {
 System.out.println(e.getMessage());
                return 100;
            })
 .whenCompleteAsync((r,e) -> {
 if(r != null && e == null){
 System.out.println("正常执行");
                }
 if(e != null){
 System.out.println("执行异常：" + e.getMessage());
                }
 });
    System.out.println(futureA.get());
}
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
java.lang.ArithmeticException: / by zero
正常执行
100

Process finished with exit code 0
```

handle() 有返回

类似于 thenAccept/thenRun，是最后一步的处理调用，但是同时可以处理异常。

handle 与 whenComplete 的作用有些类似，但是 handle 接收的处理函数有返回值，而且返回值会影响最终获取的计算结果。

handle 方法返回的 CompletableFuture 的 result 是回调方法的执行结果或者回调方法执行期间抛出的异常，与原始 CompletableFuture 的 result 无关了。

```
private static Integer num = 10;

    public static void main(String[] args) throws Exception {
        System.out.println("主线程开始");
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(()
-> {
          //  int i= 1/0;
            System.out.println("加 10 任务开始");
            num += 10;
            return num;
        }).handle((i, ex) -> {
            System.out.println("进入 handle 方法");
            if (ex != null) {
                System.out.println("发生了异常,内容为:" + ex.getMessage());
                return -1;
            } else {
                System.out.println("正常完成,内容为: " + i);
                return i;
            }
        });
        System.out.println(future.get());
    }
```

产出型方法，即可以对正常完成的结果进行转换，也可以对异常完成的进行补偿一个结果，即可以改变阶段的现状。

跟 whenComplete 基本一致，区别在于 handle 的回调方法有返回值，且 handle 方法返回的 CompletableFuture 的 result 是回调方法的执行结果或者回调方法执行期间抛出的异常，与原始 CompletableFuture 的 result 无关。

Java 多线程之 CompletableFuture_爱吃牛肉的大老虎的博客-CSDN 博客

java 异步编程 CompletableFuture_compleablefuture 异步编程_帅气的喵喵的博客-CSDN 博客

问题

1.runAsync 是否可以使用 exceptionally

可以。在 exceptionally 中返回 null。

```
private static void test5() throws ExecutionException, InterruptedException,
TimeoutException {
    final CompletableFuture completableFuture = CompletableFuture
            .runAsync(() -> log.info("" + 1 / 0))
            .exceptionally(ex -> {
                log.info("exception: " + ex.getMessage());
                return null;
            });

    log.info("result: " + completableFuture.get(1, TimeUnit.SECONDS));
}
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
10:35:00.320 [ForkJoinPool.commonPool-worker-1] INFO com.txw.ms.sample.service.TestCompletableFuture - exception: java.lang.ArithmeticException: / by zero
10:35:00.328 [main] INFO com.txw.ms.sample.service.TestCompletableFuture - result: null
```

2.trycatch 和 exceptionally 的区别

3.同一个任务，调用多次 get()，该任务会执行多次吗？

不会。任务只会执行一次，返回结果复用。

```
private static void test8supplyAsync() throws ExecutionException,
InterruptedException {
    CompletableFuture<Integer> job = CompletableFuture.supplyAsync(() -> {
       log.info("Start job: ");
       return num;
    });
    log.info("return: " + job.get() * job.get());
}
```

```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
20:33:35.370 [ForkJoinPool.commonPool-worker-1] INFO com.txw.ms.sample.service.TestCompletableFuture - Start job:
20:33:35.374 [main] INFO com.txw.ms.sample.service.TestCompletableFuture - return: 25
```

误区

1.没有设置超时时间

CompletableFuture.get(3, TimeUnit.SECONDS)

2. 没有使用自定义线程池

默认使用 ForkJoinPool。

3. 没有异常处理

4. 多个异步任务操作同一个对象

```
CompletableFuture<Integer> futureA = CompletableFuture.supplyAsync(() -> 10 / 0)
 .thenApply(s -> {
 System.out.println("执行了");
            return s + 1;
        })
 .whenComplete((r, e) -> {
 if (r != null && e == null) {
 System.out.println("正常执行");
            }
 if (e != null) {
 System.out.println("执行异常：" + e.getMessage());
            }
 });
System.out.println(futureA.get());
```