**CompletableFuture 总结**

**简单介绍**

CompletableFuture是java8的新特性之一，用于异步编程，对于非阻塞的代码可以从串行改为并行，使用这种并行方式，可以极大的提高程序的性能。

CompletableFuture 实现了 JAVA中原本的Future 和 CompletionStage接口，前者是对后者的一个扩展，增加了异步回调、流式处理、多个Future组合处理的能力，使Java在处理多任务的协同工作时更加顺畅便利。

下面从创建、返回、回调、组合来介绍如何使用CompletableFuture

**创建**

**创建CompletableFuture实例的方法为 supplyAsync 和 runAsync**

两个方法都支持lambda表达式，其主要区别为是否有返回值，如下所示

* supplyAsync 返回 CompletableFuture<T>
* runAsync 返回 CompletableFuture<Void>

|  |
| --- |
| 注意：两个方法都有各自的重载版本，可以指定线程池，如果不指定线程池，则会使用默认的ForkJoinPool.commonPool()来获得线程执行任务。 |

代码举例：

|  |
| --- |
| Java ExecutorService executorService = Executors.newFixedThreadPool(3); CompletableFuture<String> completableFuture1 = CompletableFuture.supplyAsync(() -> {  return "test1"; },executorService); CompletableFuture<Void> completableFuture2 = CompletableFuture.runAsync(() -> {  log.info("test2"); },executorService); |

**返回**

**获取CompletableFuture实例返回的方法为 join 和 get**

两者都是获得任务的返回值，且都会阻塞任务，其区别为，

1. join抛出的是未受检查的异常，而get抛出的是经过检查的异常，需要调用方手动处理(抛出或try catch)
2. get有一个重载方法，可以传入超时时间

因为两者的区别，所以join更适合在链中调用，无须处理异常，而get适合在最外围调用，使用try catch处理异常并返回数据

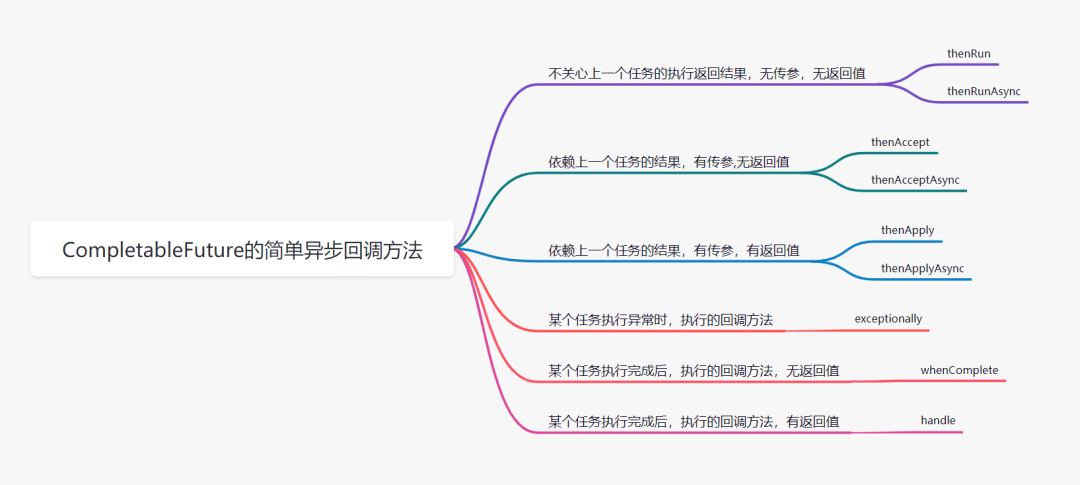
代码举例：

|  |
| --- |
| Java CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> {  return 101; },executorService); try {  completableFuture1.get(); } catch (InterruptedException | ExecutionException e) {  throw new RuntimeException(e); }  CompletableFuture<Integer> completableFuture2 = CompletableFuture.runAsync(() -> {  return 102; },executorService); CompletableFuture<Integer> completableFuture3 = completableFuture2.thenApplyAsync(a -> {  return completableFuture2.join() + 3; }, executorService); |

**回调**

**回调方法总览**

回调方法是completeableFuture中链式调用的重要一步，在很多时候，我们希望把线程A的结果作为线程B的入参，来执行线程B，这种情况下就是使用回调方法的时机



**thenRun / thenRunAsync**

这两个方法表示当指定的CompletableFuture实例1执行完之后，才会执行当前CompletableFuture实例2，其中两个CompletableFuture实例不涉及参数传递，且这两个方法也不会返回参数。

这两个方法的区别是

* 调用thenRun方法执行第二个任务时，则第二个任务和第一个任务是**共用同一个线程池中的线程**。
* 调用thenRunAsync执行第二个任务时，则第一个任务使用原本的线程池，**第二个任务使用的是ForkJoin线程池，但如果你传入一个线程池，会使用你传入的线程池中的线程**

总结：就是说不带Async的方法是由触发该任务的线程执行该任务，带Async的方法是由触发该任务的线程将任务提交到线程池，执行任务的线程跟触发任务的线程不一定是同一个。

|  |
| --- |
| 注意：thenAccept和thenAcceptAsync，thenApply和thenApplyAsync区别同上。 |

|  |
| --- |
| Java public CompletableFuture<Void> thenRun(Runnable action); public CompletableFuture<Void> thenRunAsync(Runnable action); |

代码举例：

|  |
| --- |
| SQL ExecutorService executorService = Executors.newFixedThreadPool(3); CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> {  return 101; },executorService); CompletableFuture<Void> completableFuture3 = completableFuture1.thenRunAsync(() -> {  log.info("103"); }, executorService); |

**thenAccept / thenAcceptAsync**

这两个方法表示当指定的CompletableFuture实例1执行完之后，才会执行当前CompletableFuture实例2，其中会将CompletableFuture实例1的执行结果，作为入参，传递到回调方法中，但这两个回调方法也不会返回参数。

两者区别同thenRun 和 thenRunAsync，请向上查阅

|  |
| --- |
| Java public CompletionStage<Void> thenAccept(Consumer<? super T> action); public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action); |

代码举例：

|  |
| --- |
| SQL CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> {  return 101; },executorService); CompletableFuture<Void> completableFuture3 = completableFuture1.thenAcceptAsync(a -> {  log.info("这是completableFuture1的结果: {}", a); }, executorService); |

**thenApply / thenApplyAsync**

这两个方法表示当指定的CompletableFuture实例1执行完之后，才会执行当前CompletableFuture实例2，其中会将CompletableFuture实例1的执行结果，作为入参，传递到回调方法中，且这两个回调方法会返回参数。

两者区别同thenRun 和 thenRunAsync，请向上查阅

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

代码举例：

|  |
| --- |
| Java ExecutorService executorService = Executors.newFixedThreadPool(3); CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> {  return 101; },executorService); CompletableFuture<Integer> completableFuture3 = completableFuture1.thenApplyAsync(a -> {  return a + 1; }, executorService); |

**exceptionally**

|  |
| --- |
| Java public CompletionStage<T> exceptionally(Function<Throwable, ? extends T> fn); |

该方法主要用于处理异常，只有当前CompletableFuture实例异常时，才会执行exceptionally，如果当前CompletableFuture实例正常执行，则不会执行exceptionally其中的逻辑

|  |
| --- |
| *注意：exceptionally是有返回值的*   1. *如果当前CompletableFuture实例正常执行，则exceptionally的返回值就是当前CompletableFuture实例的返回值* 2. *如果当前CompletableFuture实例异常，则会运行exceptionally中的逻辑来计算返回值* 3. *链式调用中，出现异常即会调用该方法的exceptionally，后续链路不会再调用* |

代码举例1：

|  |
| --- |
| Java ExecutorService executorService = Executors.newFixedThreadPool(5); CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> {  if (true){  ThrowableUtil.commonException(ErrorCodeEnum.ILLEGAL\_PARAMETERS, "test");  }  log.info("原本的值：101");  return 101; },executorService).exceptionally(a -> {  log.error("测试异常: ", a);  log.info("更改后的值：102");  return 102; }); System.out.println("completableFuture1: " + completableFuture1.join()); |

此时输出为：

|  |
| --- |
| Bash 2023-08-16 15:57:15.368 -ERROR 24372 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 测试异常:   java.util.concurrent.CompletionException: com.mi.info.miim.fs.common.exception.CommonException: test  at java.util.concurrent.CompletableFuture.encodeThrowable(CompletableFuture.java:273)  at java.util.concurrent.CompletableFuture.completeThrowable(CompletableFuture.java:280)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1606)  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)  at java.lang.Thread.run(Thread.java:750) Caused by: com.mi.info.miim.fs.common.exception.CommonException: test  at com.mi.info.miim.fs.common.util.ThrowableUtil.commonException(ThrowableUtil.java:25)  at com.mi.info.miim.fs.WarehouseSeatClientTest.lambda$test1$0(WarehouseSeatClientTest.java:57)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1604)  ... 3 common frames omitted  2023-08-16 15:57:15.369 - INFO 24372 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 更改后的值：102 completableFuture1: 102 |

可以看到只有异常时，才会执行，且exceptionally有返回值，如果将true改成false，则输出为：

|  |
| --- |
| SQL 2023-08-16 15:59:45.366 - INFO 25680 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 原本的值：101 completableFuture1: 101 |

代码举例2：

|  |
| --- |
| Java ExecutorService executorService = Executors.newFixedThreadPool(5); CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> {  if (true){  ThrowableUtil.commonException(ErrorCodeEnum.ILLEGAL\_PARAMETERS, "test");  }  log.info("这是CompletableFuture实例1的值：101");  return 101; },executorService);  CompletableFuture<Integer> completableFuture2 = completableFuture1.thenApply(a -> {  log.info(String.format("这是CompletableFuture实例2的值：%d", a + 1));  return a + 1; });  CompletableFuture<Integer> completableFuture3 = completableFuture1.exceptionally(a -> {  log.error("这是异常测试: ", a);  log.info("更改后的值：103");  return 103; });  System.out.println("completableFuture3: " + completableFuture2.join()); |

输出：

|  |
| --- |
| Bash 2023-08-16 17:19:18.377 -ERROR 25840 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 这是异常测试:   java.util.concurrent.CompletionException: com.mi.info.miim.fs.common.exception.CommonException: test  at java.util.concurrent.CompletableFuture.encodeThrowable(CompletableFuture.java:273)  at java.util.concurrent.CompletableFuture.completeThrowable(CompletableFuture.java:280)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1606)  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)  at java.lang.Thread.run(Thread.java:750) Caused by: com.mi.info.miim.fs.common.exception.CommonException: test  at com.mi.info.miim.fs.common.util.ThrowableUtil.commonException(ThrowableUtil.java:25)  at com.mi.info.miim.fs.WarehouseSeatClientTest.lambda$test1$0(WarehouseSeatClientTest.java:55)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1604)  ... 3 common frames omitted  2023-08-16 17:19:18.400 - INFO 25840 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 更改后的值：103  java.util.concurrent.CompletionException: com.mi.info.miim.fs.common.exception.CommonException: test   at java.util.concurrent.CompletableFuture.encodeThrowable(CompletableFuture.java:273)  at java.util.concurrent.CompletableFuture.completeThrowable(CompletableFuture.java:280)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1606)  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)  at java.lang.Thread.run(Thread.java:750) Caused by: com.mi.info.miim.fs.common.exception.CommonException: test  at com.mi.info.miim.fs.common.util.ThrowableUtil.commonException(ThrowableUtil.java:25)  at com.mi.info.miim.fs.WarehouseSeatClientTest.lambda$test1$0(WarehouseSeatClientTest.java:55)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1604)  ... 3 more |

true改false的输出：

|  |
| --- |
| SQL 2023-08-16 17:21:35.847 - INFO 1640 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 这是CompletableFuture实例1的值：101 2023-08-16 17:21:35.847 - INFO 1640 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 这是CompletableFuture实例2的值：102 completableFuture3: 102 |

**whenComlete**

whenComplete会将上一个CompletableFuture实例的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。

whenComlete方法并没有返回值，它对应的返回值是原CompletableFuture实例的返回值

whenComlete有重载方法，可以传入线程池，如不传入则默认使用原CompletableFuture实例的线程池

PS：因为该方法并没有返回值，所以只适合日志记录，资源清理等不会对原结果做更改的操作

|  |
| --- |
| Java public CompletionStage<T> whenComplete(BiConsumer<? super T, ? super Throwable> action); public CompletionStage<T> whenCompleteAsync(BiConsumer<? super T, ? super Throwable> action, Executor executor); |

代码举例：

|  |
| --- |
| Java ExecutorService executorService = Executors.newFixedThreadPool(5); CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> {  if (false){  ThrowableUtil.commonException(ErrorCodeEnum.ILLEGAL\_PARAMETERS, "test");  }  log.info("这是CompletableFuture实例1的值：101");  return 101; }, executorService);  CompletableFuture<Integer> completableFuture2 = completableFuture1.whenComplete((result, exception) -> {  if (null != exception){  log.info("CompletableFuture实例1把报错穿过来了");  ThrowableUtil.commonException(ErrorCodeEnum.ILLEGAL\_PARAMETERS, exception.getMessage());  }  log.info(String.format("CompletableFuture实例1把“%d”穿过来了", result)); });  System.out.println("completableFuture2: " + completableFuture2.join()); |

输出：

|  |
| --- |
| SQL 2023-08-16 18:28:48.465 - INFO 22588 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 这是CompletableFuture实例1的值：101 2023-08-16 18:28:48.465 - INFO 22588 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : CompletableFuture实例1把“101”穿过来了 completableFuture2: 101 |

把false改为true的输出：

|  |
| --- |
| Bash 2023-08-16 18:30:25.296 - INFO 25572 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : CompletableFuture实例1把报错穿过来了  java.util.concurrent.CompletionException: com.mi.info.miim.fs.common.exception.CommonException: test   at java.util.concurrent.CompletableFuture.encodeThrowable(CompletableFuture.java:273)  at java.util.concurrent.CompletableFuture.completeThrowable(CompletableFuture.java:280)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1606)  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)  at java.lang.Thread.run(Thread.java:750) Caused by: com.mi.info.miim.fs.common.exception.CommonException: test  at com.mi.info.miim.fs.common.util.ThrowableUtil.commonException(ThrowableUtil.java:25)  at com.mi.info.miim.fs.WarehouseSeatClientTest.lambda$test1$0(WarehouseSeatClientTest.java:55)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1604)  ... 3 more |

**handle**

该方法与whenComplete基本一致，他也会将上一个CompletableFuture实例的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。

但不同的是handle的回调方法有返回值，且返回结果可以是对原CompletableFuture实例的执行结果或者异常进行修改得到，也可以返回一个新的值。

代码举例：

|  |
| --- |
| Java ExecutorService executorService = Executors.newFixedThreadPool(5); CompletableFuture<Integer> completableFuture1 = CompletableFuture.supplyAsync(() -> {  if (false){  ThrowableUtil.commonException(ErrorCodeEnum.ILLEGAL\_PARAMETERS, "testError1");  }  log.info("这是CompletableFuture实例1的值：101");  return 101; }, executorService);  CompletableFuture<Integer> completableFuture2 = completableFuture1.handle((result, exception) -> {  if (null != exception){  log.info("CompletableFuture实例1把报错穿过来了:" + exception.getMessage());  log.info("修改CompletableFuture实例1的报错为: testError2");  // 这里可以不进行报错，返回一个新的值  ThrowableUtil.commonException(ErrorCodeEnum.ILLEGAL\_PARAMETERS, "testError2");  }  log.info(String.format("CompletableFuture实例1把“%d”穿过来了", result));  log.info("修改CompletableFuture实例1的值为 102");  return result + 1; });  System.out.println("completableFuture2: " + completableFuture2.join()); |

输出：

|  |
| --- |
| SQL 2023-08-16 19:03:42.298 - INFO 23960 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 这是CompletableFuture实例1的值：101 2023-08-16 19:03:42.299 - INFO 23960 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : CompletableFuture实例1把“101”穿过来了 2023-08-16 19:03:42.299 - INFO 23960 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 修改CompletableFuture实例1的值为 102 completableFuture2: 102 |

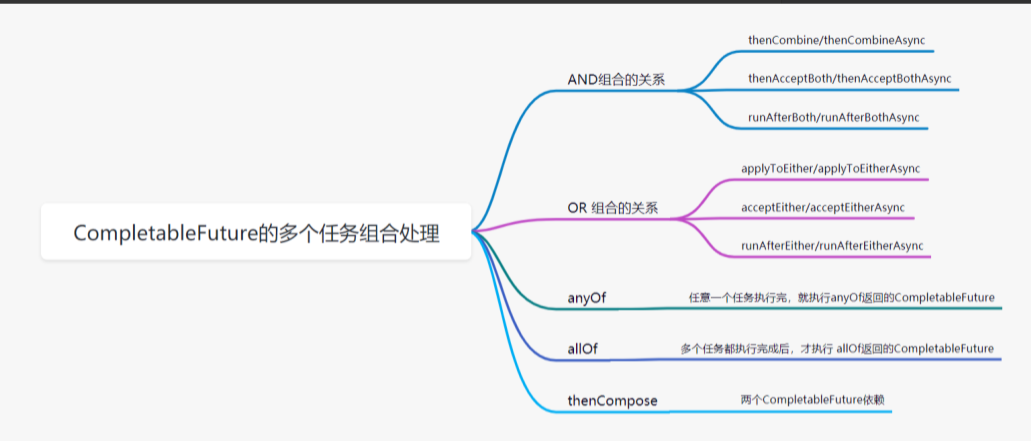
将false改为true后的输出：

|  |
| --- |
| Bash 2023-08-18 10:10:18.355 - INFO 8356 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : CompletableFuture实例1把报错穿过来了:com.mi.info.miim.fs.common.exception.CommonException: testError1 2023-08-18 10:10:18.356 - INFO 8356 --- [pool-5-thread-1] [] c.m.i.miim.fs.WarehouseSeatClientTest : 修改CompletableFuture实例1的报错为: testError2  java.util.concurrent.CompletionException: com.mi.info.miim.fs.common.exception.CommonException: testError2   at java.util.concurrent.CompletableFuture.encodeThrowable(CompletableFuture.java:273)  at java.util.concurrent.CompletableFuture.completeThrowable(CompletableFuture.java:280)  at java.util.concurrent.CompletableFuture.uniHandle(CompletableFuture.java:838)  at java.util.concurrent.CompletableFuture$UniHandle.tryFire(CompletableFuture.java:811)  at java.util.concurrent.CompletableFuture.postComplete(CompletableFuture.java:488)  at java.util.concurrent.CompletableFuture$AsyncSupply.run$$$capture(CompletableFuture.java:1609)  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java)  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)  at java.lang.Thread.run(Thread.java:750) Caused by: com.mi.info.miim.fs.common.exception.CommonException: testError2  at com.mi.info.miim.fs.common.util.ThrowableUtil.commonException(ThrowableUtil.java:25)  at com.mi.info.miim.fs.WarehouseSeatClientTest.lambda$test1$1(WarehouseSeatClientTest.java:55)  at java.util.concurrent.CompletableFuture.uniHandle(CompletableFuture.java:836)  ... 7 more |

**组合**

**组合总览**

组合是多个异步关系之间的联动处理，可以对多个CompletableFuture实例进行组合来达到我们想要的结果



**AND关系 thenCombine / thenAcceptBoth / runAfterBoth**

这三个方法都表示：**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**。且这三个方法都有一个可以传入的线程池的重载方法，在我个人的测试中，如果不传入线程，当前CompletableFuture实例会使用组合的两个CompletableFuture实例最后执行完的线程。

* 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**有返回值**
* thenAcceptBoth: 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**无返回值**
* runAfterBoth **不会把执行结果当做方法入参**，且没有返回值。

|  |
| --- |
| Java public <U,V> CompletionStage<V> thenCombine(CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn); public <U> CompletionStage<Void> thenAcceptBoth(CompletionStage<? extends U> other,BiConsumer<? super T, ? super U> action); public CompletionStage<Void> runAfterBoth(CompletionStage<?> other,Runnable action); |

代码举例：

|  |
| --- |
| Java ForkJoinPool pool=new ForkJoinPool(); // 创建异步执行任务: CompletableFuture<Double> cf = CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job1,time->"+System.currentTimeMillis());  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job1,time->"+System.currentTimeMillis());  return 1.2; }); CompletableFuture<Double> cf2 = CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job2,time->"+System.currentTimeMillis());  try {  Thread.sleep(1500);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job2,time->"+System.currentTimeMillis());  return 3.2; }); //cf和cf2的CompletableFuture实例都执行完成后，会将其执行结果作为方法入参传递给cf3,且有返回值 CompletableFuture<Double> cf3=cf.thenCombine(cf2,(a,b)->{  System.out.println(Thread.currentThread()+" start job3,time->"+System.currentTimeMillis());  System.out.println("job3 param a->"+a+",b->"+b);  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job3,time->"+System.currentTimeMillis());  return a+b; });  //cf和cf2的CompletableFuture实例都执行完成后，会将其执行结果作为方法入参传递给cf3,无返回值 CompletableFuture cf4=cf.thenAcceptBoth(cf2,(a,b)->{  System.out.println(Thread.currentThread()+" start job4,time->"+System.currentTimeMillis());  System.out.println("job4 param a->"+a+",b->"+b);  try {  Thread.sleep(1500);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job4,time->"+System.currentTimeMillis()); });  //cf4和cf3都执行完成后，执行cf5，无入参，无返回值 CompletableFuture cf5=cf4.runAfterBoth(cf3,()->{  System.out.println(Thread.currentThread()+" start job5,time->"+System.currentTimeMillis());  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  }  System.out.println("cf5 do something");  System.out.println(Thread.currentThread()+" exit job5,time->"+System.currentTimeMillis()); });  System.out.println("main thread start cf.get(),time->"+System.currentTimeMillis()); //等待子任务执行完成 System.out.println("cf run result->"+cf.get()); System.out.println("main thread start cf5.get(),time->"+System.currentTimeMillis()); System.out.println("cf5 run result->"+cf5.get()); System.out.println("main thread exit,time->"+System.currentTimeMillis()); |

输出结果：

|  |
| --- |
| SQL Thread[ForkJoinPool.commonPool-worker-9,5,main] start job1,time->1692338889644 Thread[ForkJoinPool.commonPool-worker-2,5,main] start job2,time->1692338889645 main thread start cf.get(),time->1692338889646 Thread[ForkJoinPool.commonPool-worker-2,5,main] exit job2,time->1692338891156 Thread[ForkJoinPool.commonPool-worker-9,5,main] exit job1,time->1692338891659 Thread[ForkJoinPool.commonPool-worker-9,5,main] start job4,time->1692338891659 Thread[main,5,main] start job3,time->1692338891659 job4 param a->1.2,b->3.2 job3 param a->1.2,b->3.2 Thread[ForkJoinPool.commonPool-worker-9,5,main] exit job4,time->1692338893163 Thread[main,5,main] exit job3,time->1692338893666 Thread[main,5,main] start job5,time->1692338893666 cf5 do something Thread[main,5,main] exit job5,time->1692338894681 cf run result->1.2 main thread start cf5.get(),time->1692338894681 cf5 run result->null main thread exit,time->1692338894681 |

**OR关系 applyToEither / acceptEither / runAfterEither**

这三个方法都表示：将两个CompletableFuture组合起来，只要其中一个执行完了，就会开始执行调用的CompletableFuture实例

区别在于：

* applyToEither：会将已经执行完成的任务，作为方法入参，传递到指定方法中，且有返回值
* acceptEither: 会将已经执行完成的任务，作为方法入参，传递到指定方法中，且无返回值
* runAfterEither：不会把执行结果当做方法入参，且没有返回值。

关于这里的异常传递，是正常的调用传递，即被调用方法报错，则调用方法也会报错，但是如果出现其中一个方法已经执行完，那么另一个方法并不会继续执行，所以即使后面有报错，代码也并不会执行

代码举例：

|  |
| --- |
| Java *// 创建异步执行任务:*  CompletableFuture<Double> cf = CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job1,time->"+System.currentTimeMillis());  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job1,time->"+System.currentTimeMillis());  return 1.2;  });  CompletableFuture<Double> cf2 = CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job2,time->"+System.currentTimeMillis());  try {  Thread.sleep(1500);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job2,time->"+System.currentTimeMillis());  return 3.2;  });  *//cf和cf2的CompletableFuture实例都执行完成后，会将其执行结果作为方法入参传递给cf3,且有返回值*  CompletableFuture<Double> cf3=cf.applyToEither(cf2,(result)->{  System.out.println(Thread.currentThread()+" start job3,time->"+System.currentTimeMillis());  System.out.println("job3 param result->"+result);  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job3,time->"+System.currentTimeMillis());  return result;  });    *//cf和cf2的CompletableFuture实例都执行完成后，会将其执行结果作为方法入参传递给cf3,无返回值*  CompletableFuture cf4=cf.acceptEither(cf2,(result)->{  System.out.println(Thread.currentThread()+" start job4,time->"+System.currentTimeMillis());  System.out.println("job4 param result->"+result);  try {  Thread.sleep(1500);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job4,time->"+System.currentTimeMillis());  });    *//cf4和cf3都执行完成后，执行cf5，无入参，无返回值*  CompletableFuture cf5=cf4.runAfterEither(cf3,()->{  System.out.println(Thread.currentThread()+" start job5,time->"+System.currentTimeMillis());  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  }  System.out.println("cf5 do something");  System.out.println(Thread.currentThread()+" exit job5,time->"+System.currentTimeMillis());  });    System.out.println("main thread start cf.get(),time->"+System.currentTimeMillis());  *//等待子任务执行完成*  System.out.println("cf run result->"+cf.get());  System.out.println("main thread start cf5.get(),time->"+System.currentTimeMillis());  System.out.println("cf5 run result->"+cf5.get());  System.out.println("main thread exit,time->"+System.currentTimeMillis()); |

输出结果：

|  |
| --- |
| SQL Thread[ForkJoinPool.commonPool-worker-9,5,main] start job1,time->1692344280259 Thread[ForkJoinPool.commonPool-worker-2,5,main] start job2,time->1692344280259 main thread start cf.get(),time->1692344280260 Thread[ForkJoinPool.commonPool-worker-2,5,main] exit job2,time->1692344281767 Thread[ForkJoinPool.commonPool-worker-2,5,main] start job4,time->1692344281767 job4 param result->3.2 Thread[ForkJoinPool.commonPool-worker-9,5,main] exit job1,time->1692344282266 Thread[ForkJoinPool.commonPool-worker-9,5,main] start job3,time->1692344282266 cf run result->1.2 job3 param result->1.2 main thread start cf5.get(),time->1692344282266 Thread[ForkJoinPool.commonPool-worker-2,5,main] exit job4,time->1692344283278 Thread[ForkJoinPool.commonPool-worker-2,5,main] start job5,time->1692344283278 Thread[ForkJoinPool.commonPool-worker-9,5,main] exit job3,time->1692344284275 cf5 do something Thread[ForkJoinPool.commonPool-worker-2,5,main] exit job5,time->1692344284291 cf5 run result->null main thread exit,time->1692344284291 |

**批量关系 anyOf / allOf**

这两个方法可以见字识意，anyOf代表多个CompletableFuture实例只要其中一个执行完成就结束方法，anyOf表示多个CompletableFuture实例全部执行完成之后才结束方法。

allOf方法没有返回值，在最后会返回一个CompletableFuture<Void> ，该返回值没有任何实际指向，只是作为方法完成的标志，所以如果执行get会直接返回null

anyOf方法也没有返回值，在最后会返回一个CompletableFuture<Object>，这个对象表示的是已经执行完成的任务的执行结果

这里的异常传递也是正常的，即被调用方法报错，则调用方法也会报错，但是如果出现其中一个方法已经执行完，那么其他的方法并不会继续执行，因为anyOf会直接返回，所以即使后面有报错，代码也并不会执行

|  |
| --- |
| TypeScript public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs) {  return andTree(cfs, 0, cfs.length - 1); }  public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs) {  return orTree(cfs, 0, cfs.length - 1); } |

代码举例：

|  |
| --- |
| Go // 创建异步执行任务: CompletableFuture<Double> cf = CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job1,time->"+System.currentTimeMillis());  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job1,time->"+System.currentTimeMillis());  return 1.2; }); CompletableFuture<Double> cf2 = CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job2,time->"+System.currentTimeMillis());  try {  Thread.sleep(1500);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job2,time->"+System.currentTimeMillis());  return 3.2; }); CompletableFuture<Double> cf3 = CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job3,time->"+System.currentTimeMillis());  try {  Thread.sleep(1300);  } catch (InterruptedException e) {  }  //throw new RuntimeException("test");  System.out.println(Thread.currentThread()+" exit job3,time->"+System.currentTimeMillis());  return 2.2; }); //allof等待所有任务执行完成才执行cf4，如果有一个任务异常终止，则cf4.get时会抛出异常，都是正常执行，cf4.get返回null //anyOf是只有一个任务执行完成，无论是正常执行或者执行异常，都会执行cf4，cf4.get的结果就是已执行完成的任务的执行结果 CompletableFuture cf4=CompletableFuture.allOf(cf,cf2,cf3).whenComplete((a,b)->{  if(b!=null){  System.out.println("error stack trace->");  b.printStackTrace();  }else{  System.out.println("run succ,result->"+a);  } });  System.out.println("main thread start cf4.get(),time->"+System.currentTimeMillis()); //等待子任务执行完成 System.out.println("cf4 run result->"+cf4.get()); System.out.println("main thread exit,time->"+System.currentTimeMillis()); |

输出结果：

|  |
| --- |
| SQL Thread[ForkJoinPool.commonPool-worker-9,5,main] start job1,time->1692346334231 Thread[ForkJoinPool.commonPool-worker-2,5,main] start job2,time->1692346334231 Thread[ForkJoinPool.commonPool-worker-11,5,main] start job3,time->1692346334231 main thread start cf4.get(),time->1692346334232 Thread[ForkJoinPool.commonPool-worker-11,5,main] exit job3,time->1692346335538 Thread[ForkJoinPool.commonPool-worker-2,5,main] exit job2,time->1692346335744 Thread[ForkJoinPool.commonPool-worker-9,5,main] exit job1,time->1692346336242 run succ,result->null cf4 run result->null main thread exit,time->1692346336242 |

将allOf改为anyOf之后的输出：

|  |
| --- |
| SQL Thread[ForkJoinPool.commonPool-worker-9,5,main] start job1,time->1692346758268 Thread[ForkJoinPool.commonPool-worker-11,5,main] start job3,time->1692346758268 Thread[ForkJoinPool.commonPool-worker-2,5,main] start job2,time->1692346758268 main thread start cf4.get(),time->1692346758269 Thread[ForkJoinPool.commonPool-worker-11,5,main] exit job3,time->1692346759569 run succ,result->2.2 cf4 run result->2.2 main thread exit,time->1692346759569 |

**thenCompose**

thenCompose方法会把上个任务的结果作为入参，来执行新的逻辑，且在最后是返回一个新的CompletableFuture实例。

因为最后的结果是返回一个新的CompletableFuture实例，所以在方法主要用于两个CompletableFuture实例的关联，比如拿到上一个CompletableFuture实例的结果来生成一个新的CompletableFuture实例，简单来说，就是用于CompletableFuture实例的串行化，当然你也可以直接返回一个新的CompletableFuture，就是这样操作并没有什么意义罢了

代码举例：

|  |
| --- |
| Go // 创建异步执行任务: CompletableFuture<Double> cf = CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job1,time->"+System.currentTimeMillis());  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job1,time->"+System.currentTimeMillis());  return 1.2; }); CompletableFuture<String> cf2= cf.thenCompose((param)->{  System.out.println(Thread.currentThread()+" start job2,time->"+System.currentTimeMillis());  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job2,time->"+System.currentTimeMillis());  return CompletableFuture.supplyAsync(()->{  System.out.println(Thread.currentThread()+" start job3,time->"+System.currentTimeMillis());  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  }  System.out.println(Thread.currentThread()+" exit job3,time->"+System.currentTimeMillis());  return "job3 test";  }); }); System.out.println("main thread start cf.get(),time->"+System.currentTimeMillis()); //等待子任务执行完成 System.out.println("cf run result->"+cf.get()); System.out.println("main thread start cf2.get(),time->"+System.currentTimeMillis()); System.out.println("cf2 run result->"+cf2.get()); System.out.println("main thread exit,time->"+System.currentTimeMillis()); |

输出结果：

|  |
| --- |
| SQL Thread[ForkJoinPool.commonPool-worker-9,5,main] start job1,time->1692354584044 main thread start cf.get(),time->1692354584044 Thread[ForkJoinPool.commonPool-worker-9,5,main] exit job1,time->1692354586045 Thread[ForkJoinPool.commonPool-worker-9,5,main] start job2,time->1692354586045 cf run result->1.2 main thread start cf2.get(),time->1692354586045 Thread[ForkJoinPool.commonPool-worker-9,5,main] exit job2,time->1692354588054 Thread[ForkJoinPool.commonPool-worker-9,5,main] start job3,time->1692354588054 Thread[ForkJoinPool.commonPool-worker-9,5,main] exit job3,time->1692354590061 cf2 run result->job3 test main thread exit,time->1692354590061 |

---

参考文章：

1. [异步编程利器：CompletableFuture详解](https://www.cnblogs.com/ludongguoa/p/15316488.html)

2. [Java8 异步非阻塞做法：CompletableFuture 两万字详解！](https://blog.csdn.net/weixin\_45727359/article/details/128107773?spm=1001.2101.3001.6650.1&utm\_medium=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-128107773-blog-115525700.235%5Ev38%5Epc\_relevant\_default\_base&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-128107773-blog-115525700.235%5Ev38%5Epc\_relevant\_default\_base&utm\_relevant\_index=2)