

主讲老师: Fox

课前须知:

1. 关于Future的使用, 务必掌握, 掌握了Future就掌握了未来
2. CompletableFuture的API非常丰富, 不用全部掌握, 大概了解有哪些功能, 使用时会查API就行

有道云链接: <http://note.youdao.com/noteshare?id=0e961b20b4e7a0b21fab4ed9f88c1ac5&sub=DF1632E020FB4B82BE3FB7E8F92FAC48>

Callable&Future&FutureTask介绍

Callable和Runnable的区别

Future 的主要功能

利用 FutureTask 创建 Future

如何使用

使用案例: 促销活动中商品信息查询

Future 注意事项

Future的局限性

CompletionService

CompletionService原理

使用案例

询价应用: 向不同电商平台询价, 并保存价格

实现类似 Dubbo 的 Forking Cluster场景

应用场景总结

CompletableFuture使用详解

应用场景

创建异步操作

`runAsync&supplyAsync`

获取结果

`join&get`

结果处理

`whenComplete&exceptionally`

结果转换

`thenApply`

`thenCompose`

结果消费

`thenAccept`

`thenRun`

结果组合

`thenCombine`

任务交互

`applyToEither`

`acceptEither`

`runAfterEither`

`runAfterBoth`

`anyOf`

`allOf`

CompletableFuture常用方法总结

使用案例：实现最优的“烧水泡茶”程序

基于Future实现

基于CompletableFuture实现

Callable&Future&FutureTask介绍

直接继承Thread或者实现Runnable接口都可以创建线程，但是这两种方法都有一个问题就是：没有返回值，也就是不能获取执行完的结果。因此java1.5就提供了Callable接口来实现这一场景，而Future和FutureTask就可以和Callable接口配合起来使用。

Callable和Runnable的区别

思考：为什么需要 Callable?

```
1 @FunctionalInterface
2 public interface Runnable {
3     public abstract void run();
4 }
5 @FunctionalInterface
6 public interface Callable<V> {
7     V call() throws Exception;
8 }
```

Runnable 的缺陷：

- 不能返回一个返回值
- 不能抛出 checked Exception

Callable的call方法可以有返回值，可以声明抛出异常。和 Callable 配合的有一个 Future 类，通过 Future 可以了解任务执行情况，或者取消任务的执行，还可获取任务执行的结果，这些功能都是 Runnable 做不到的，Callable 的功能要比 Runnable 强大。

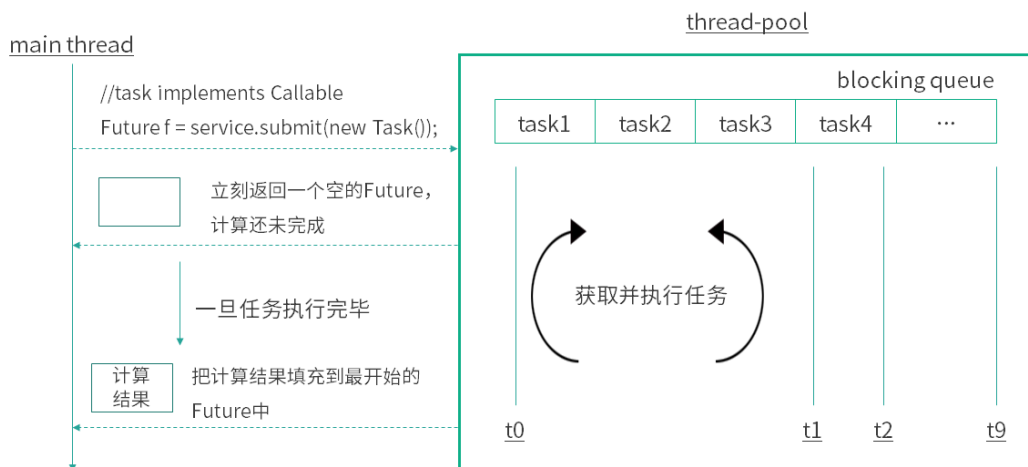
```
1 new Thread(new Runnable() {
2     @Override
3     public void run() {
4         System.out.println("通过Runnable方式执行任务");
5     }
6 }).start();
7
8 FutureTask task = new FutureTask(new Callable() {
9     @Override
10    public Object call() throws Exception {
11        System.out.println("通过Callable方式执行任务");
12        Thread.sleep(3000);
13        return "返回任务结果";
14    }
15 });
16 new Thread(task).start();
17 System.out.println(task.get());
```

Future 的主要功能

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。

必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

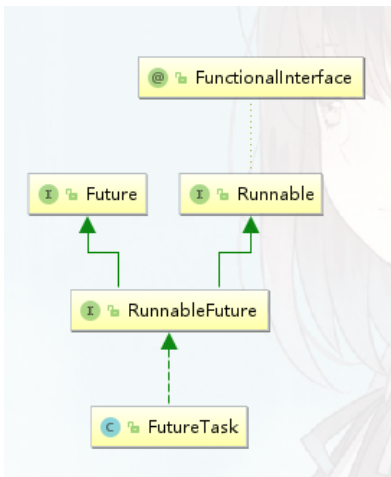
- `boolean cancel (boolean mayInterruptIfRunning)` 取消任务的执行。参数指定是否立即中断任务执行，或者等等任务结束
- `boolean isCancelled ()` 任务是否已经取消，任务正常完成前将其取消，则返回true
- `boolean isDone ()` 任务是否已经完成。需要注意的是如果任务正常终止、异常或取消，都将返回true
- `V get () throws InterruptedException, ExecutionException` 等待任务执行结束，然后获得V类型的结果。InterruptedException 线程被中断异常，ExecutionException任务执行异常，如果任务被取消，还会抛出CancellationException
- `V get (long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException` 同上面的get功能一样，多了设置超时时间。参数timeout指定超时时间，unit指定时间的单位，在枚举类TimeUnit中有相关的定义。如果计算超时，将抛出TimeoutException



利用 FutureTask 创建 Future

Future实际采用**FutureTask**实现，该对象相当于是消费者和生产者的桥梁，消费者通过FutureTask 存储任务的处理结果，更新任务的状态：未开始、正在处理、已完成等。而生产者拿到的 FutureTask 被转型为 Future 接口，可以阻塞式获取任务的处理结果，非阻塞式获取任务处理状态。

FutureTask既可以被当做Runnable来执行，也可以被当做Future来获取Callable的返回结果。



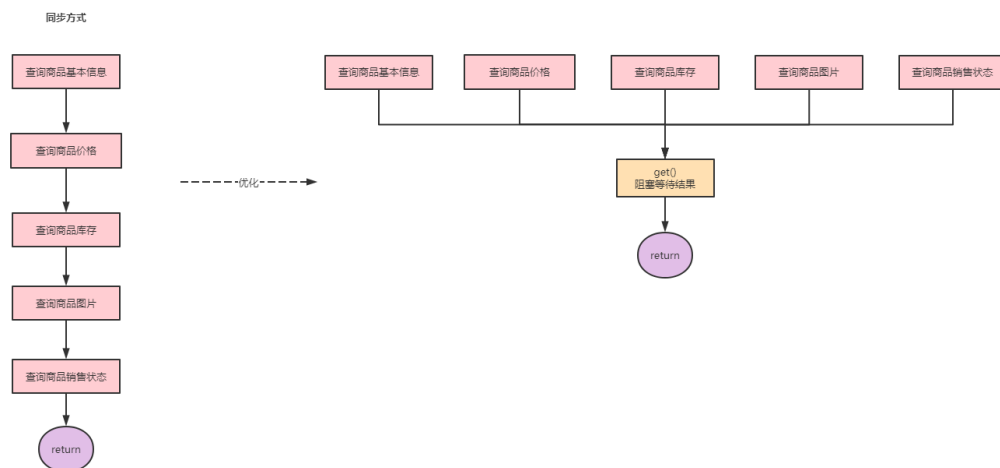
如何使用

把 Callable 实例当作 FutureTask 构造函数的参数，生成 FutureTask 的对象，然后把这个对象当作一个 Runnable 对象，放到线程池中或另起线程去执行，最后还可以通过 FutureTask 获取任务执行的结果。

```
1 public class FutureTaskDemo {
2
3     public static void main(String[] args) throws ExecutionException, Interrupt
4         edException {
5         Task task = new Task();
6         //构建futureTask
7         FutureTask<Integer> futureTask = new FutureTask<>(task);
8         //作为Runnable入参
9         new Thread(futureTask).start();
10
11         System.out.println("task运行结果: "+futureTask.get());
12     }
13
14     static class Task implements Callable<Integer> {
15
16         @Override
17         public Integer call() throws Exception {
18             System.out.println("子线程正在计算");
19             int sum = 0;
20             for (int i = 0; i < 100; i++) {
21                 sum += i;
22             }
23             return sum;
24         }
25     }
```

使用案例：促销活动中商品信息查询

在维护促销活动时需要查询商品信息（包括商品基本信息、商品价格、商品库存、商品图片、商品销售状态等）。这些信息分布在不同的业务中心，由不同的系统提供服务。如果采用同步方式，假设一个接口需要50ms，那么一个商品查询下来就需要200ms-300ms，这对于我们来说是不满意的。如果使用Future改造则需要的就是最长耗时服务的接口，也就是50ms左右。



```
1 public class FutureTaskDemo2 {
2
3     public static void main(String[] args) throws ExecutionException, Interrupt
4         edException {
5
6         FutureTask<String> ft1 = new FutureTask<>(new T1Task());
7         FutureTask<String> ft2 = new FutureTask<>(new T2Task());
8         FutureTask<String> ft3 = new FutureTask<>(new T3Task());
9         FutureTask<String> ft4 = new FutureTask<>(new T4Task());
10        FutureTask<String> ft5 = new FutureTask<>(new T5Task());
11
12        //构建线程池
13        ExecutorService executorService = Executors.newFixedThreadPool(5);
14        executorService.submit(ft1);
15        executorService.submit(ft2);
16        executorService.submit(ft3);
17        executorService.submit(ft4);
18        executorService.submit(ft5);
19        //获取执行结果
20        System.out.println(ft1.get());
21        System.out.println(ft2.get());
22        System.out.println(ft3.get());
23        System.out.println(ft4.get());
24        System.out.println(ft5.get());
25    }
26 }
```

```
23 System.out.println(ft5.get());
24
25 executorService.shutdown();
26
27 }
28
29 static class T1Task implements Callable<String> {
30 @Override
31 public String call() throws Exception {
32 System.out.println("T1:查询商品基本信息...");
33 TimeUnit.MILLISECONDS.sleep(50);
34 return "商品基本信息查询成功";
35 }
36 }
37
38 static class T2Task implements Callable<String> {
39 @Override
40 public String call() throws Exception {
41 System.out.println("T2:查询商品价格...");
42 TimeUnit.MILLISECONDS.sleep(50);
43 return "商品价格查询成功";
44 }
45 }
46
47 static class T3Task implements Callable<String> {
48 @Override
49 public String call() throws Exception {
50 System.out.println("T3:查询商品库存...");
51 TimeUnit.MILLISECONDS.sleep(50);
52 return "商品库存查询成功";
53 }
54 }
55
56 static class T4Task implements Callable<String> {
57 @Override
58 public String call() throws Exception {
59 System.out.println("T4:查询商品图片...");
60 TimeUnit.MILLISECONDS.sleep(50);
61 return "商品图片查询成功";
62 }
63 }
64
```

```
65  static class T5Task implements Callable<String> {
66  @Override
67  public String call() throws Exception {
68  System.out.println("T5:查询商品销售状态...");
69  TimeUnit.MILLISECONDS.sleep(50);
70  return "商品销售状态查询成功";
71  }
72  }
73
74 }
```

Future 注意事项

- 当 for 循环批量获取 Future 的结果时容易 block，get 方法调用时应使用 timeout 限制
- Future 的生命周期不能后退。一旦完成了任务，它就永久停在了“已完成”的状态，不能从头再来

思考：使用Callable 和Future 产生新的线程了吗？


Future的局限性






从本质上说，**Future表示一个异步计算的结果**。它提供了isDone()来检测计算是否已经完成，并且在计算结束后，可以通过get()方法来获取计算结果。在异步计算中，Future确实是个非常优秀的接口。但是，它的本身也确实存在着许多限制：

- **并发执行多任务**：Future只提供了get()方法来获取结果，并且是阻塞的。所以，除了等待你别无他法；
- **无法对多个任务进行链式调用**：如果你希望在计算任务完成后执行特定动作，比如发邮件，但Future却没有提供这样的能力；
- **无法组合多个任务**：如果你运行了10个任务，并期望在它们全部执行结束后执行特定动作，那么在Future中这是无能为力的；
- **没有异常处理**：Future接口中没有关于异常处理的方法；

CompletionService

Callable+Future 可以实现多个task并行执行，但是如果遇到前面的task执行较慢时需要阻塞等待前面的task执行完后面task才能取得结果。而**CompletionService的主要功能就是一边生成任务,一边获取任务的返回值。让两件事分开执行,任务之间不会互相阻塞，可以实现先执行完的先取结果，不再依赖任务顺序了。**

 **CompletionService**

-  **poll(): Future<V>**
-  **poll(long, TimeUnit): Future<V>**
-  **submit(Callable<V>): Future<V>**
-  **submit(Runnable, V): Future<V>**
-  **take(): Future<V>**

CompletionService原理

内部通过阻塞队列+FutureTask，实现了任务先完成可优先获取到，即结果按照完成先后顺序排序，内部有一个先进先出的阻塞队列，用于保存已经执行完成的Future，通过调用它的take方法或poll方法可以获取到一个已经执行完成的Future，进而通过调用Future接口实现类的get方法获取最终的结果

使用案例

询价应用：向不同电商平台询价，并保存价格

- 采用“ThreadPoolExecutor+Future”的方案：异步执行询价然后再保存

```
1 // 创建线程池
2 ExecutorService executor = Executors.newFixedThreadPool(3);
3 // 异步向电商S1询价
4 Future<Integer> f1 = executor.submit(()->getPriceByS1());
5 // 异步向电商S2询价
6 Future<Integer> f2= executor.submit(()->getPriceByS2());
7 // 获取电商S1报价并异步保存
8 executor.execute(()->save(f1.get()));
9 // 获取电商S2报价并异步保存
10 executor.execute(()->save(f2.get()));
```

如果获取电商S1报价的耗时很长，那么即便获取电商S2报价的耗时很短，也无法让保存S2报价的操作先执行，因为这个主线程都阻塞在了f1.get()操作上。

- 使用CompletionService实现先获取的报价先保存到数据库

```
1 //创建线程池
2 ExecutorService executor = Executors.newFixedThreadPool(10);
3 //创建CompletionService
4 CompletionService<Integer> cs = new ExecutorCompletionService<>(executor);
5 //异步向电商S1询价
6 cs.submit(() -> getPriceByS1());
7 //异步向电商S2询价
8 cs.submit(() -> getPriceByS2());
9 //异步向电商S3询价
10 cs.submit(() -> getPriceByS3());
```

```
11 //将询价结果异步保存到数据库
12 for (int i = 0; i < 3; i++) {
13     Integer r = cs.take().get();
14     executor.execute(() -> save(r));
15 }
```

实现类似 Dubbo 的 Forking Cluster场景

Dubbo 中有一种叫做 Forking 的集群模式，这种集群模式下，支持并行地调用多个服务实例，只要有一个成功就返回结果。

```
1
2 geocoder(addr) {
3     //并行执行以下3个查询服务，
4     r1=geocoderByS1(addr);
5     r2=geocoderByS2(addr);
6     r3=geocoderByS3(addr);
7     //只要r1,r2,r3有一个返回
8     //则返回
9     return r1|r2|r3;
10 }
```

```
1
2 // 创建线程池
3 ExecutorService executor = Executors.newFixedThreadPool(3);
4 // 创建CompletionService
5 CompletionService<Integer> cs = new ExecutorCompletionService<>(executor);
6 // 用于保存Future对象
7 List<Future<Integer>> futures = new ArrayList<>(3);
8 //提交异步任务，并保存future到futures
9 futures.add(cs.submit(()->geocoderByS1()));
10 futures.add(cs.submit(()->geocoderByS2()));
11 futures.add(cs.submit(()->geocoderByS3()));
12 // 获取最快返回的任务执行结果
13 Integer r = 0;
14 try {
15     // 只要有一个成功返回，则break
16     for (int i = 0; i < 3; ++i) {
17         r = cs.take().get();
18         //简单地通过判空来检查是否成功返回
19         if (r != null) {
20             break;
21         }
```

```
22     }
23 } finally {
24     //取消所有任务
25     for(Future<Integer> f : futures)
26         f.cancel(true);
27 }
28 // 返回结果
29 return r;
```

应用场景总结

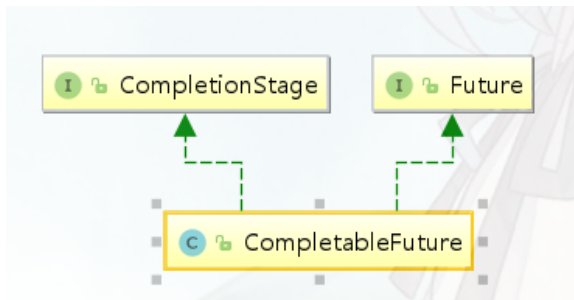
- 当需要批量提交异步任务的时候建议你使用CompletionService。
CompletionService将线程池Executor和阻塞队列BlockingQueue的功能融合在了一起，能够让批量异步任务的管理更简单。
- CompletionService能够让异步任务的执行结果有序化。先执行完的先进入阻塞队列，利用这个特性，你可以轻松实现后续处理的有序性，避免无谓的等待，同时还可以快速实现诸如Forking Cluster这样的需求。
- 线程池隔离。CompletionService支持自己创建线程池，这种隔离性能避免几个特别耗时的任务拖垮整个应用的风险。

CompletableFuture使用详解

简单的任务，用Future获取结果还好，但我们并行提交的多个异步任务，往往并不是独立的，很多时候业务逻辑处理存在串行[依赖]、并行、聚合的关系。如果要我们手动用Future实现，是非常麻烦的。

CompletableFuture是Future接口的扩展和增强。CompletableFuture实现了Future接口，并在此基础上进行了丰富地扩展，完美地弥补了Future上述的种种问题。更为重要的是，**CompletableFuture实现了对任务的编排能力。**借助这项能力，我们可以轻松地组织不同任务的运行顺序、规则以及方式。从某种程度上说，这项能力是它的核心能力。而在以往，虽然通过CountDownLatch等工具类也可以实现任务的编排，但需要复杂的逻辑处理，不仅耗费精力且难以维护。

jdk8 API文档: <https://docs.oracle.com/javase/8/docs/api/>



CompletionStage接口: 执行某一个阶段，可向下执行后续阶段。异步执行，默认线程池是ForkJoinPool.commonPool()

应用场景

描述依赖关系:

1. thenApply() 把前面异步任务的结果，交给后面的Function
2. thenCompose()用来连接两个有依赖关系的任务，结果由第二个任务返回

描述and聚合关系:

1. thenCombine:任务合并，有返回值
2. thenAcceptBoth:两个任务执行完成后，将结果交给thenAcceptBoth消耗，无返回值。
3. runAfterBoth:两个任务都执行完成后，执行下一步操作（Runnable）。

描述or聚合关系:

1. applyToEither:两个任务谁执行的快，就使用那一个结果，有返回值。
2. acceptEither: 两个任务谁执行的快，就消耗那一个结果，无返回值。
3. runAfterEither: 任意一个任务执行完成，进行下一步操作(Runnable)。

并行执行:

CompletableFuture类自己也提供了anyOf()和allOf()用于支持多个CompletableFuture并行执行

创建异步操作

CompletableFuture 提供了四个静态方法来创建一个异步操作:

```
1 public static CompletableFuture<Void> runAsync(Runnable runnable)
2 public static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
3 public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
4 public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)
```

这四个方法区别在于:

- `runAsync` 方法以 `Runnable` 函数式接口类型为参数，没有返回结果，`supplyAsync` 方法 `Supplier` 函数式接口类型为参数，返回结果类型为 `U`；**`Supplier` 接口的 `get()` 方法是有返回值的（会阻塞）**
- 没有指定 `Executor` 的方法会使用 `ForkJoinPool.commonPool()` 作为它的线程池执行异步代码。如果指定线程池，则使用指定的线程池运行。
- 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**

runAsync&supplyAsync

```

1 Runnable runnable = () -> System.out.println("执行无返回结果的异步任务");
2 CompletableFuture.runAsync(runnable);
3
4 CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
5     System.out.println("执行有返回值的异步任务");
6     try {
7         Thread.sleep(5000);
8     } catch (InterruptedException e) {
9         e.printStackTrace();
10    }
11    return "Hello World";
12 });
13 String result = future.get();
14 System.out.println(result);

```

执行无返回结果的异步任务

执行有返回值的异步任务

获取结果

join&get

`join()`和`get()`方法都是用来获取`CompletableFuture`异步之后的返回值。`join()`方法抛出的是 `unchecked` 异常（即未经检查的异常），不会强制开发者抛出。`get()`方法抛出的是经过检查的异常，`ExecutionException`, `InterruptedException` 需要用户手动处理（抛出或者 `try catch`）

结果处理

当CompletableFuture的计算结果完成，或者抛出异常的时候，我们可以执行特定的 Action。主要是下面的方法：

```
1 public CompletableFuture<T> whenComplete(BiConsumer<? super T, ? super Throwable> action)
2 public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T, ? super Throwable> action)
3 public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T, ? super Throwable> action, Executor executor)
4 public CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn)
```

- Action的类型是BiConsumer<? super T, ? super Throwable>，它可以处理正常的计算结果，或者异常情况。
- 方法不以Async结尾，意味着Action使用相同的线程执行，而Async可能会使用其它的线程去执行(如果使用相同的线程池，也可能被同一个线程选中执行)。
- 这几个方法都会返回CompletableFuture，当Action执行完毕后它的结果返回原始的CompletableFuture的计算结果或者返回异常

whenComplete&exceptionally

```
1 CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
2     try {
3         TimeUnit.SECONDS.sleep(1);
4     } catch (InterruptedException e) {
5     }
6     if (new Random().nextInt(10) % 2 == 0) {
7         int i = 12 / 0;
8     }
9     System.out.println("执行结束！");
10    return "test";
11 });
12
13 future.whenComplete(new BiConsumer<String, Throwable>() {
14     @Override
15     public void accept(String t, Throwable action) {
16         System.out.println(t+" 执行完成！");
17     }
18 });
19
20 future.exceptionally(new Function<Throwable, String>() {
21     @Override
22     public String apply(Throwable t) {
23         System.out.println("执行失败： " + t.getMessage());
```

```
24     return "异常xxx";
25 }
26 }).join();
```

执行结束!

test 执行完成!

或者

执行失败: java.lang.ArithmeticException: / by zero

null 执行完成!

结果转换

所谓结果转换，就是将上一段任务的执行结果作为下一阶段任务的入参参与重新计算，产生新的结果。

thenApply

thenApply 接收一个函数作为参数，使用该函数处理上一个CompletableFuture调用的结果，并返回一个具有处理结果的Future对象。

```
1 public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U>
fn)
2 public <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends
U> fn)
3 public <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends
U> fn, Executor executor)
```

```
1 CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
2     int result = 100;
3     System.out.println("一阶段: " + result);
4     return result;
5 }).thenApply(number -> {
6     int result = number * 3;
7     System.out.println("二阶段: " + result);
8     return result;
9 });
10
11 System.out.println("最终结果: " + future.get());
```

一阶段: 100

二阶段: 300

最终结果: 300

thenCompose

thenCompose 的参数为一个返回 CompletableFuture 实例的函数，该函数的参数是先前计算步骤的结果。

```
1 public <U> CompletableFuture<U> thenCompose(Function<? super T, ? extends Co
mpletionStage<U>> fn);
```

```

2 public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn) ;
3 public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn, Executor executor) ;

```

```

1 CompletableFuture<Integer> future = CompletableFuture
2   .supplyAsync(new Supplier<Integer>() {
3     @Override
4     public Integer get() {
5       int number = new Random().nextInt(30);
6       System.out.println("第一阶段: " + number);
7       return number;
8     }
9   })
10  .thenCompose(new Function<Integer, CompletionStage<Integer>>() {
11    @Override
12    public CompletionStage<Integer> apply(Integer param) {
13      return CompletableFuture.supplyAsync(new Supplier<Integer>() {
14        @Override
15        public Integer get() {
16          int number = param * 2;
17          System.out.println("第二阶段: " + number);
18          return number;
19        }
20      });
21    }
22  });
23 System.out.println("最终结果: " + future.get());

```

第一阶段: 10

第二阶段: 20

最终结果: 20

thenApply 和 thenCompose的区别

- thenApply 转换的是泛型中的类型，返回的是同一个CompletableFuture；
- thenCompose 将内部的 CompletableFuture 调用展开来并使用上一个 CompletableFuture 调用的结果在下一步的 CompletableFuture 调用中进行运算，是生成一个新的CompletableFuture。

```

1 CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");
2
3 CompletableFuture<String> result1 = future.thenApply(param -> param + " World");

```



```

4 CompletableFuture<String> result2 = future
5   .thenCompose(param -> CompletableFuture.supplyAsync(() -> param + "
World"));
6
7 System.out.println(result1.get());
8 System.out.println(result2.get());

```

```

Hello World
Hello World

```

结果消费

与结果处理和结果转换系列函数返回一个新的 `CompletableFuture` 不同，结果消费系列函数只对结果执行Action，而不返回新的计算值。

根据对结果的处理方式，结果消费函数又分为：

- `thenAccept`系列：对单个结果进行消费
- `thenAcceptBoth`系列：对两个结果进行消费
- `thenRun`系列：不关心结果，只对结果执行Action

thenAccept

通过观察该系列函数的参数类型可知，它们是函数式接口Consumer，这个接口只有输入，没有返回值。

```

1 public CompletionStage<Void> thenAccept(Consumer<? super T> action);
2 public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action);
3 public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action, Executor executor);

```

```

1 CompletableFuture<Void> future = CompletableFuture
2   .supplyAsync(() -> {
3     int number = new Random().nextInt(10);
4     System.out.println("第一阶段: " + number);
5     return number;
6   }).thenAccept(number ->
7     System.out.println("第二阶段: " + number * 5));
8
9 System.out.println("最终结果: " + future.get());

```

```

第一阶段: 8
第二阶段: 40
最终结果: null

```

thenAcceptBoth

`thenAcceptBoth` 函数的作用是，当两个 `CompletionStage` 都正常完成计算的时候，就会执行提供的action消费两个异步的结果。

```

1 public <U> CompletionStage<Void> thenAcceptBoth(CompletionStage<? extends U>
other, BiConsumer<? super T, ? super U> action);

2 public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<? exten
ds U> other, BiConsumer<? super T, ? super U> action);

3 public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<? exten
ds U> other, BiConsumer<? super T, ? super U> action, Executor executor);

```

```

1 CompletableFuture<Integer> futrue1 = CompletableFuture.supplyAsync(new Suppl
ier<Integer>() {
2     @Override
3     public Integer get() {
4         int number = new Random().nextInt(3) + 1;
5         try {
6             TimeUnit.SECONDS.sleep(number);
7         } catch (InterruptedException e) {
8             e.printStackTrace();
9         }
10        System.out.println("第一阶段: " + number);
11        return number;
12    }
13 });

14

15 CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(new Suppl
ier<Integer>() {
16     @Override
17     public Integer get() {
18         int number = new Random().nextInt(3) + 1;
19         try {
20             TimeUnit.SECONDS.sleep(number);
21         } catch (InterruptedException e) {
22             e.printStackTrace();
23         }
24        System.out.println("第二阶段: " + number);
25        return number;
26    }
27 });

28

29 futrue1.thenAcceptBoth(future2, new BiConsumer<Integer, Integer>() {
30     @Override
31     public void accept(Integer x, Integer y) {
32         System.out.println("最终结果: " + (x + y));
33     }
34 }).join();

```

第二阶段: 1
第一阶段: 2
最终结果: 3

thenRun

thenRun 也是对线程任务结果的一种消费函数, 与thenAccept不同的是, thenRun 会在上一阶段 CompletableFuture 计算完成的时候执行一个Runnable, Runnable并不使用该 CompletableFuture 计算的结果。

```
1 public CompletionStage<Void> thenRun(Runnable action);
2 public CompletionStage<Void> thenRunAsync(Runnable action);
3 public CompletionStage<Void> thenRunAsync(Runnable action,Executor
  executor);
```

```
1 CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {
2   int number = new Random().nextInt(10);
3   System.out.println("第一阶段: " + number);
4   return number;
5 }).thenRun(() ->
6   System.out.println("thenRun 执行"));
7 System.out.println("最终结果: " + future.get());
```

第一阶段: 2
thenRun 执行
最终结果: null

结果组合

thenCombine

thenCombine 方法, 合并两个线程任务的结果, 并进一步处理。

```
1 public <U,V> CompletionStage<V> thenCombine(CompletionStage<? extends U> oth
  er,BiFunction<? super T,? super U,? extends V> fn);
2 public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends
  U> other,BiFunction<? super T,? super U,? extends V> fn);
3 public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends
  U> other,BiFunction<? super T,? super U,? extends V> fn,Executor executor);
```

```
1 CompletableFuture<Integer> future1 = CompletableFuture
2   .supplyAsync(new Supplier<Integer>() {
3     @Override
4     public Integer get() {
5       int number = new Random().nextInt(10);
6       System.out.println("第一阶段: " + number);
7       return number;
```

```

8  }
9  });
10 CompletableFuture<Integer> future2 = CompletableFuture
11     .supplyAsync(new Supplier<Integer>() {
12         @Override
13         public Integer get() {
14             int number = new Random().nextInt(10);
15             System.out.println("第二阶段: " + number);
16             return number;
17         }
18     });
19 CompletableFuture<Integer> result = future1
20     .thenCombine(future2, new BiFunction<Integer, Integer, Integer>() {
21         @Override
22         public Integer apply(Integer x, Integer y) {
23             return x + y;
24         }
25     });
26 System.out.println("最终结果: " + result.get());

```

第一阶段: 9

第二阶段: 5

最终结果: 14

任务交互

所谓线程交互，是指将两个线程任务获取结果的速度相比较，按一定的规则进行下一步处理。

applyToEither

两个线程任务相比较，先获得执行结果的，就对该结果进行下一步的转化操作。

```

1 public <U> CompletionStage<U> applyToEither(CompletionStage<? extends T> oth
er,Function<? super T, U> fn);
2 public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends
T> other,Function<? super T, U> fn);
3 public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends
T> other,Function<? super T, U> fn,Executor executor);

```

```

1 CompletableFuture<Integer> future1 = CompletableFuture
2     .supplyAsync(new Supplier<Integer>() {
3         @Override
4         public Integer get() {
5             int number = new Random().nextInt(10);
6             System.out.println("第一阶段start: " + number);
7             try {
8                 TimeUnit.SECONDS.sleep(number);

```

```

9  } catch (InterruptedException e) {
10  e.printStackTrace();
11  }
12  System.out.println("第一阶段end: " + number);
13  return number;
14  }
15  });
16  CompletableFuture<Integer> future2 = CompletableFuture
17  .supplyAsync(new Supplier<Integer>() {
18  @Override
19  public Integer get() {
20  int number = new Random().nextInt(10);
21  System.out.println("第二阶段start: " + number);
22  try {
23  TimeUnit.SECONDS.sleep(number);
24  } catch (InterruptedException e) {
25  e.printStackTrace();
26  }
27  System.out.println("第二阶段end: " + number);
28  return number;
29  }
30  });
31
32  future1.applyToEither(future2, new Function<Integer, Integer>() {
33  @Override
34  public Integer apply(Integer number) {
35  System.out.println("最快结果: " + number);
36  return number * 2;
37  }
38  }).join();

```

```

第一阶段start: 6
第二阶段start: 5
第二阶段end: 5
最快结果: 5

```

acceptEither

两个线程任务相比较，先获得执行结果的，就对该结果进行下一步的消费操作。

```

1  public CompletionStage<Void> acceptEither(CompletionStage<? extends T>
other, Consumer<? super T> action);
2  public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T>
other, Consumer<? super T> action);
3  public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T>
other, Consumer<? super T> action, Executor executor);

```

```

1 CompletableFuture<Integer> future1 = CompletableFuture
2   .supplyAsync(new Supplier<Integer>() {
3     @Override
4     public Integer get() {
5       int number = new Random().nextInt(10) + 1;
6       try {
7         TimeUnit.SECONDS.sleep(number);
8       } catch (InterruptedException e) {
9         e.printStackTrace();
10      }
11      System.out.println("第一阶段: " + number);
12      return number;
13    }
14  });
15
16 CompletableFuture<Integer> future2 = CompletableFuture
17   .supplyAsync(new Supplier<Integer>() {
18     @Override
19     public Integer get() {
20       int number = new Random().nextInt(10) + 1;
21       try {
22         TimeUnit.SECONDS.sleep(number);
23       } catch (InterruptedException e) {
24         e.printStackTrace();
25      }
26      System.out.println("第二阶段: " + number);
27      return number;
28    }
29  });
30
31 future1.acceptEither(future2, new Consumer<Integer>() {
32   @Override
33   public void accept(Integer number) {
34     System.out.println("最快结果: " + number);
35   }
36 }).join();

```

第二阶段: 3

最快结果: 3

runAfterEither

两个线程任务相比较，有任何一个执行完成，就进行下一步操作，不关心运行结果。

```
1 public CompletionStage<Void> runAfterEither(CompletionStage<?> other, Runnable action);
2 public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?> other, Runnable action);
3 public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?> other, Runnable action, Executor executor);
```

```
1 CompletableFuture<Integer> future1 = CompletableFuture
2   .supplyAsync(new Supplier<Integer>() {
3   @Override
4   public Integer get() {
5     int number = new Random().nextInt(5);
6     try {
7       TimeUnit.SECONDS.sleep(number);
8     } catch (InterruptedException e) {
9       e.printStackTrace();
10    }
11    System.out.println("第一阶段: " + number);
12    return number;
13  }
14 });
15
16 CompletableFuture<Integer> future2 = CompletableFuture
17   .supplyAsync(new Supplier<Integer>() {
18   @Override
19   public Integer get() {
20     int number = new Random().nextInt(5);
21     try {
22       TimeUnit.SECONDS.sleep(number);
23     } catch (InterruptedException e) {
24       e.printStackTrace();
25     }
26     System.out.println("第二阶段: " + number);
27     return number;
28   }
29 });
30
31 future1.runAfterEither(future2, new Runnable() {
32 @Override
33 public void run() {
34   System.out.println("已经有一个任务完成了");
35 }
```

```
35 }  
36 }).join();
```

第一阶段: 3
已经有一个任务完成了

runAfterBoth

两个线程任务相比较，两个全部执行完成，才进行下一步操作，不关心运行结果。

```
1 public CompletionStage<Void> runAfterBoth(CompletionStage<?> other, Runnable  
   action);  
2 public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other, Runn  
   able action);  
3 public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other, Runn  
   able action, Executor executor);
```

```
1 CompletableFuture<Integer> future1 = CompletableFuture  
2   .supplyAsync(new Supplier<Integer>() {  
3     @Override  
4     public Integer get() {  
5       try {  
6         TimeUnit.SECONDS.sleep(1);  
7       } catch (InterruptedException e) {  
8         e.printStackTrace();  
9       }  
10      System.out.println("第一阶段: 1");  
11      return 1;  
12    }  
13  });  
14  
15 CompletableFuture<Integer> future2 = CompletableFuture  
16   .supplyAsync(new Supplier<Integer>() {  
17     @Override  
18     public Integer get() {  
19       try {  
20         TimeUnit.SECONDS.sleep(2);  
21       } catch (InterruptedException e) {  
22         e.printStackTrace();  
23       }  
24       System.out.println("第二阶段: 2");  
25       return 2;  
26     }  
27  });  
28
```



```

29 future1.runAfterBoth(future2, new Runnable() {
30     @Override
31     public void run() {
32         System.out.println("上面两个任务都执行完成了。");
33     }
34 }).get();

```

第一阶段: 1

第二阶段: 2

上面两个任务都执行完成了。

anyOf

anyOf 方法的参数是多个给定的 CompletableFuture，当其中的任何一个完成时，方法返回这个 CompletableFuture。

```

1 public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

```

```

1 Random random = new Random();
2 CompletableFuture<String> future1 = CompletableFuture
3     .supplyAsync(() -> {
4         try {
5             TimeUnit.SECONDS.sleep(random.nextInt(5));
6         } catch (InterruptedException e) {
7             e.printStackTrace();
8         }
9         return "hello";
10    });
11
12 CompletableFuture<String> future2 = CompletableFuture
13     .supplyAsync(() -> {
14         try {
15             TimeUnit.SECONDS.sleep(random.nextInt(1));
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19         return "world";
20    });
21 CompletableFuture<Object> result = CompletableFuture.anyOf(future1,
22     future2);
23 System.out.println(result.get());

```

world

allOf

allOf方法用来实现多 CompletableFuture 的同时返回。

```
1 public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
```

```
1 CompletableFuture<String> future1 = CompletableFuture
2   .supplyAsync(() -> {
3     try {
4       TimeUnit.SECONDS.sleep(2);
5     } catch (InterruptedException e) {
6       e.printStackTrace();
7     }
8     System.out.println("future1完成! ");
9     return "future1完成! ";
10  });
11
12 CompletableFuture<String> future2 = CompletableFuture
13   .supplyAsync(() -> {
14     System.out.println("future2完成! ");
15     return "future2完成! ";
16  });
17
18 CompletableFuture<Void> combineFuture = CompletableFuture
19   .allOf(future1, future2);
20 try {
21   combineFuture.get();
22 } catch (InterruptedException e) {
23   e.printStackTrace();
24 } catch (ExecutionException e) {
25   e.printStackTrace();
26 }
27 System.out.println("future1: " + future1.isDone() + ", future2: " +
  future2.isDone());
```

future2完成!

future1完成!

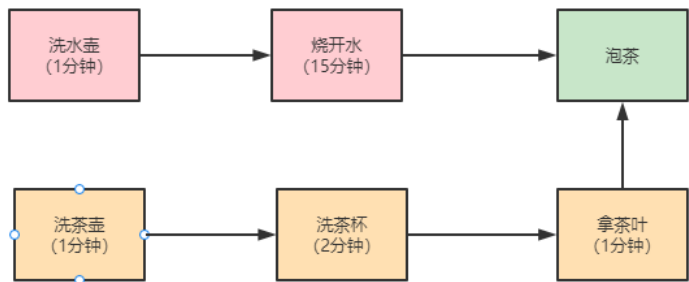
future1: true, future2: true

CompletableFuture常用方法总结

分类	方法	说明	返回值
异步执行一个线程	CompletableFuture.runAsync(Runnable)	默认使用ForkJoinPool.commonPool线程池	runAsync返回CompletableFuture<Void>:无返回值
	CompletableFuture.supplyAsync(Supplier)	也可指定Executor参数指定线程池 获取前一个线程的结果，转换。	supplyAsync返回CompletableFuture<T>:有返回值
2个线程依次执行	thenApply	thenApplySync:可指定线程池，其他方法类似	有返回值
	thenAccept	获取前一个线程的结果，消费。	无返回值
	thenRun	忽略前一个线程的结果，执行额外的逻辑	无返回值
	whenComplete	获取前一个线程的结果或异常，消费。	不影响上一线程返回值
	exceptionally	前面线程异常时，执行，一般跟whenComplete配合使用。 捕获异常范围跟前面所有异步线程 如thenApply().thenAccept().exceptionally()	有返回值
	handle	相当于whenComplete+exceptionally 根据是否产生异常内部if else分支处理业务逻辑	有返回值
等待2个线程都执行完	thenCombine	2个线程都要有返回值，等待都结束，结果合并转换。	有返回值
	thenAcceptBoth	2个线程都要有返回值，等待都结束，结果合并消费。	无返回值
	runAfterBoth	2个线程不需要有返回值，等待都结束，执行其他逻辑。	无返回值
等待2个线程任一执行完	applyToEither	2个线程都要有返回值，等待任一先结束，转换其结果。	有返回值
	acceptToEither	2个线程都要有返回值，等待任一先结束，消费其结果。	无返回值
	runAfterEither	2个线程不需要有返回值，等待任一结束，执行其他逻辑。	无返回值
多个线程等待	CompletableFuture.anyOf(cf1.cf2.cf3).join()	多个线程任一执行完即返回。	有返回值 Object
	CompletableFuture.allOf(cf1.cf2.cf3).join()	多个线程全部执行完返回。	无返回值

使用案例：实现最优的“烧水泡茶”程序

著名数学家华罗庚先生在《统筹方法》这篇文章里介绍了一个烧水泡茶的例子，文中提到最优的工序应该是下面这样：



对于烧水泡茶这个程序，一种最优的分工方案：用两个线程 T1 和 T2 来完成烧水泡茶程序，T1 负责洗水壶、烧开水、泡茶这三道工序，T2 负责洗茶壶、洗茶杯、拿茶叶三道工序，其中 T1 在执行泡茶这道工序时需要等待 T2 完成拿茶叶的工序。

基于Future实现

```
1 public class FutureTaskDemo3{
2
3     public static void main(String[] args) throws ExecutionException, Interrupt
edException {
4         // 创建任务T2的FutureTask
5         FutureTask<String> ft2 = new FutureTask<>(new T2Task());
6         // 创建任务T1的FutureTask
7         FutureTask<String> ft1 = new FutureTask<>(new T1Task(ft2));
8
9         // 线程T1执行任务ft1
10        Thread T1 = new Thread(ft1);
11        T1.start();
12        // 线程T2执行任务ft2
```

```
13 Thread T2 = new Thread(ft2);
14 T2.start();
15 // 等待线程T1执行结果
16 System.out.println(ft1.get());
17
18 }
19 }
20
21 // T1Task需要执行的任务:
22 // 洗水壶、烧开水、泡茶
23 class T1Task implements Callable<String> {
24     FutureTask<String> ft2;
25     // T1任务需要T2任务的FutureTask
26     T1Task(FutureTask<String> ft2){
27         this.ft2 = ft2;
28     }
29     @Override
30     public String call() throws Exception {
31         System.out.println("T1:洗水壶...");
32         TimeUnit.SECONDS.sleep(1);
33
34         System.out.println("T1:烧开水...");
35         TimeUnit.SECONDS.sleep(15);
36         // 获取T2线程的茶叶
37         String tf = ft2.get();
38         System.out.println("T1:拿到茶叶:"+tf);
39
40         System.out.println("T1:泡茶...");
41         return "上茶:" + tf;
42     }
43 }
44 // T2Task需要执行的任务:
45 // 洗茶壶、洗茶杯、拿茶叶
46 class T2Task implements Callable<String> {
47     @Override
48     public String call() throws Exception {
49         System.out.println("T2:洗茶壶...");
50         TimeUnit.SECONDS.sleep(1);
51
52         System.out.println("T2:洗茶杯...");
53         TimeUnit.SECONDS.sleep(2);
54
```

```
55 System.out.println("T2:拿茶叶...");
56 TimeUnit.SECONDS.sleep(1);
57 return "龙井";
58 }
59 }
```

基于CompletableFuture实现

```
1 public class CompletableFutureDemo2 {
2
3     public static void main(String[] args) {
4
5         //任务1: 洗水壶->烧开水
6         CompletableFuture<Void> f1 = CompletableFuture
7             .runAsync(() -> {
8                 System.out.println("T1:洗水壶...");
9                 sleep(1, TimeUnit.SECONDS);
10
11                 System.out.println("T1:烧开水...");
12                 sleep(15, TimeUnit.SECONDS);
13             });
14         //任务2: 洗茶壶->洗茶杯->拿茶叶
15         CompletableFuture<String> f2 = CompletableFuture
16             .supplyAsync(() -> {
17                 System.out.println("T2:洗茶壶...");
18                 sleep(1, TimeUnit.SECONDS);
19
20                 System.out.println("T2:洗茶杯...");
21                 sleep(2, TimeUnit.SECONDS);
22
23                 System.out.println("T2:拿茶叶...");
24                 sleep(1, TimeUnit.SECONDS);
25                 return "龙井";
26             });
27         //任务3: 任务1和任务2完成后执行: 泡茶
28         CompletableFuture<String> f3 = f1.thenCombine(f2, (_, tf) -> {
29             System.out.println("T1:拿到茶叶:" + tf);
30             System.out.println("T1:泡茶...");
31             return "上茶:" + tf;
32         });
33         //等待任务3执行结果
34         System.out.println(f3.join());
35     }
36 }
```

```
37  static void sleep(int t, TimeUnit u){  
38  try {  
39  u.sleep(t);  
40  } catch (InterruptedException e) {  
41  }  
42  }  
43  }
```