第五章 服务通信

基于异步消息模式的通信

异步&线程池

初始化线程的 4 种方式

1）、继承 Thread

2）、实现 Runnable 接口

3）、实现 Callable 接口 + FutureTask （可以拿到返回结果，可以处理异常）

4）、线程池

方式 1 和方式 2：主进程无法获取线程的运算结果。不适合电子商城这类微服务应用场景

方式 3：主进程可以获取线程的运算结果，但是不利于控制服务器中的线程资源。可以导致服务器资源耗尽。

1. 继承 Thread

```
public class ThreadTest {
    public static void main(String[] args) {
        System.out.println("main.....start.....");
        Thread thread = new Thread01();
        thread.start();
        System.out.println("main.....end.....");

    }

    public static class Thread01 extends Thread {
        @Override
        public void run() {
            System.out.println("当前线程: " + Thread.currentThread().getId());
            int i = 10 / 2;
            System.out.println("运行结果: " + i);
        }
```

运行结果

```
main.....start.....
main.....end.....
当前线程: 9
运行结果: 5
```

2. 实现 Runnable 接口

```
public class ThreadTest {
    public static void main(String[] args) {
        System.out.println("main......start.....");
        Runable01 runable01 = new Runable01();
        new Thread(runable01).start();
        System.out.println("main......end.....");
    }
    public static class Runable01 implements Runnable {
        @Override
        public void run() {
            System.out.println("当前线程: " + Thread.currentThread().getId());
            int i = 10 / 2;
            System.out.println("运行结果: " + i);
```

运行结果

```
main......start.....
main......end.....
当前线程: 9
运行结果: 5
```

3. 实现 Callable 接口 + FutureTask

```java
public class ThreadTest {
    public static void main(String[] args) {
        System.out.println("main......start.....");
        FutureTask<Integer> futureTask = new FutureTask<>(new Callable01());
        new Thread(futureTask).start();
        System.out.println("main......end.....");
    }
    public static class Callable01 implements Callable<Integer> {
        @Override
        public Integer call() throws Exception {
            System.out.println("当前线程：" + Thread.currentThread().getId());
            int i = 10 / 2;
            System.out.println("运行结果：" + i);
            return i;
        }
```

运行结果

```
main......start.....
main......end.....
当前线程：9
运行结果：5
```

3. 实现 Callable 接口 + FutureTask

```
public class ThreadTest {
    public static void main(String[] args) throws ExecutionException, Interru
        System.out.println("main......start.....");
        FutureTask<Integer> futureTask = new FutureTask<>(new Callable01());
        new Thread(futureTask).start();
        //等待整个线程执行完成，获取返回结果
        Integer integer = futureTask.get();
        System.out.println("main......end....."+ integer);
    }
```

运行结果

```
main......start.....
当前线程：9
运行结果：5
main......end.....5
```

2. 线程池

问题？只要有一个任务就开一个线程，最终耗尽资源

```
new Thread (()->System.out.println("hello")).start();
```

业务代码里，前面三种启动线程的方式都不用，应该将所有的多线程异步任务都交给线程池执行

//当前系统中池只有一两个，每个异步任务，提交给线程池

```
public  static ExecutorService service = Executors.newFixedThreadPool(10);
```

```
service.submit(Runable task);
service.submit( Callable<T>  task);
service.execute(Runable command);
```

submit有返回值

execute无返回值

```java
public class ThreadTest {
    //当前系统中池只有一两个，每个异步任务，提交给线程池
    public  static ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        System.out.println("main......start.....");
        service.execute(new Runable01());
        System.out.println("main......end.....");
    }
```

运行结果

```
main......start.....
main......end.....
当前线程：9
运行结果：5
```

总结区别：方式 1 和方式 2：不能得到返回值
方式 3：可以获得返回值
1，2，3都不能控制资源
4可以控制资源，性能稳定

线程池创建方法:

1) Executors

2) ThreadPoolExecutor

ThreadPoolExecutor七大参数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

1）corePoolSize：核心线程数；线程池创建好之后，池中一直保持的线程的数量，即使线程空闲。除非设置allowCoreThreadTimeOut允许核心线程超时回收。

2）maximumPoolSize：池中允许的最大的线程数

3）keepAliveTime：存活时间；当线程数大于核心线程数的时候，线程在最大多长时间没有接到新任务就会终止释放（可释放的是maximumPoolSize-corePoolSize这部分）

4）unit：时间单位;

5）BlockingQueue<Runnable> workQueue:阻塞队列，用来存储等待执行的任务，如果当前对线程的需求超过了corePoolSize大小，就会放在这里等待空闲线程执行。

6）threadFactory：创建线程的工厂，比如指定线程名等

7） handler：拒绝策略，如果线程满了，线程池就会使用拒绝策略。

运行流程：

1、线程池创建，准备好 core 数量的核心线程，准备接受任务

2、新的任务进来，用 core 准备好的空闲线程执行。

(1) 、core 满了，就将再进来的任务放入阻塞队列中。空闲的 core 就会自己去阻塞队列获取任务执行

(2) 、阻塞队列满了，就直接开新线程执行，最大只能开到 max 指定的数量

(3) 、max 都执行好了。Max-core 数量空闲的线程会在keepAliveTime 指定的时间后自动销毁。最终保持到 core 大小

(4) 、如果线程数开到了 max 的数量，还有新任务进来，就会使用 reject 指定的拒绝策略进行处理

3、所有的线程创建都是由指定的 factory 创建的。

```
ThreadPoolExecutor executor = new ThreadPoolExecutor( corePoolSize: 5,
        maximumPoolSize: 200,                          maximumPoolSize: 200,
        keepAliveTime: null,                           keepAliveTime: 10,
        unit: null,                                    TimeUnit.SECONDS,
        workQueue: null,                               new LinkedBlockingQueue<>( capacity: 100000),
        threadFactory: null,                           Executors.defaultThreadFactory(),
        handler: null                                  new ThreadPoolExecutor.AbortPolicy());
);
```

问题:

一个线程池 core 7； max 20 ， queue：50，100 并发进来怎么分配的;

先有 7 个能直接得到执行，接下来 50 个进入阻塞队列排队，再多开 13 个继续执行。现在 70 个被安排上了。剩下 30 个默认拒绝策略。

常见的 4 种线程池

1) newCachedThreadPool：core 是0，所有都可回收

Executors.newCachedThreadPool(); 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程。

2) newFixedThreadPool：固定大小，core=max，都不可回收

Executors.newFixedThreadPool();创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

3) newScheduledThreadPool：定时任务线程池

Executors.newScheduledThreadPool();创建一个定长线程池，支持定时及周期性任务执行。

4) newSingleThreadExecutor：单个线程池，core=max=1

Executors.newSingleThreadExecutor(); 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

开发中为什么使用线程池

- 降低资源的消耗

通过重复利用已经创建好的线程降低线程的创建和销毁带来的损耗

- 提高响应速度

因为线程池中的线程数没有超过线程池的最大上限时，有的线程处于等待分配任务的状态，当任务来时无需创建新的线程就能执行

- 提高线程的可管理性

线程池会根据当前系统特点对池内的线程进行优化处理，减少创建和销毁线程带来的系统开销。无限的创建和销毁线程不仅消耗系统资源，还降低系统的稳定性，使用线程池进行统一分配。

CompletableFuture 异步编排

复杂异步调用场景

三个异步请求:

A　A需要等异步任务C的结果再执行

B　可以直接执行

C

异步任务之间有关系组合

业务场景：

查询商品详情页的逻辑比较复杂，有些数据还需要远程调用，必然需要花费更多的时间。

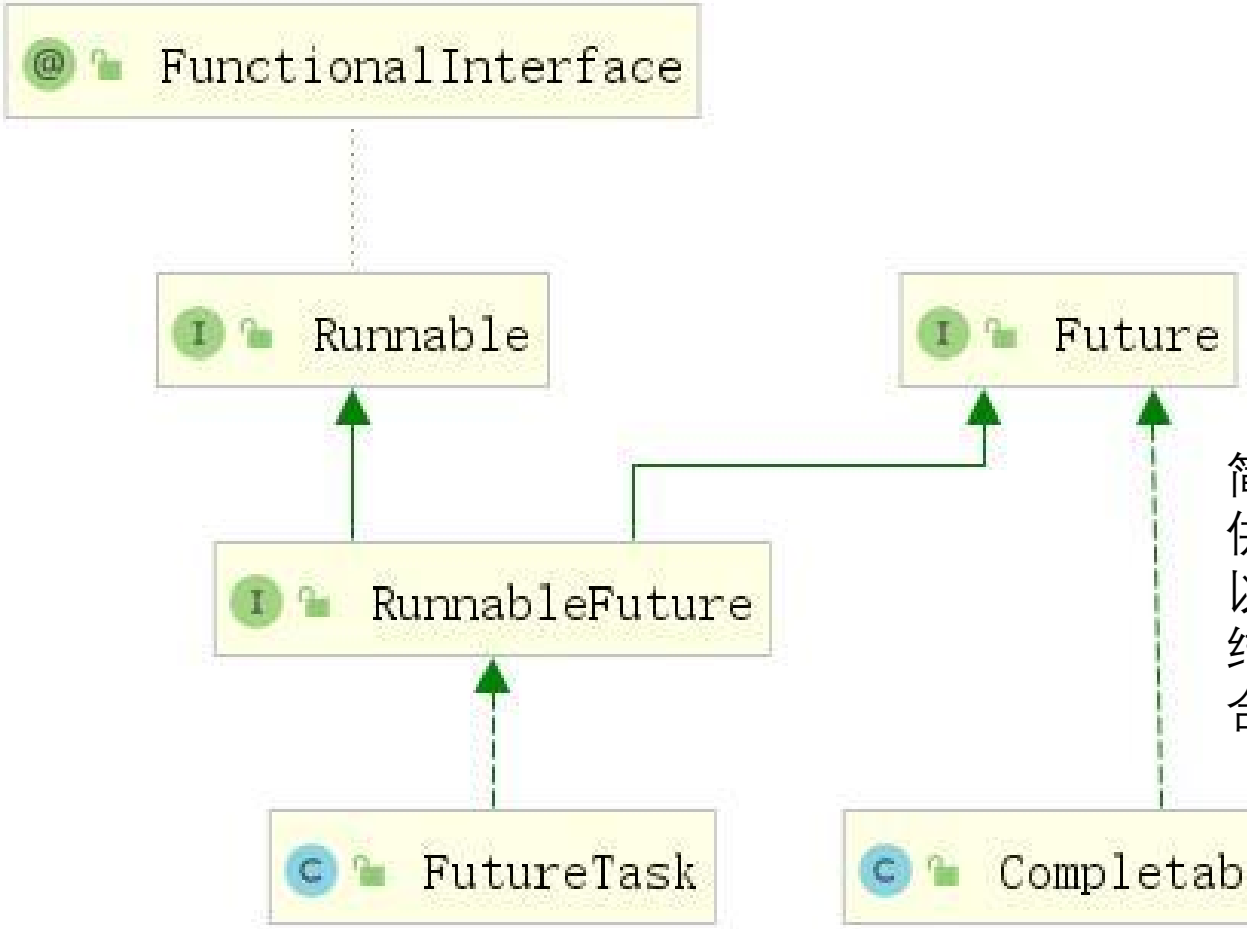
```
// 1. 获取sku的基本信息     0.5s

// 2. 获取sku的图片信息     0.5s

// 3. 获取sku的促销信息     1s

// 4. 获取spu的所有销售属性    1s

// 5. 获取规格参数组及组下的规格参数    1.5s

// 6. spu详情      1s
```

假如商品详情页的每个查询，需要如下标注的时间才能完成

那么，用户需要 5.5s 后才能看到商品详情页的内容。很显然是不能接受的。

如果有多个线程同时完成这 6 步操作，也许只需要 1.5s 即可完成响应。

CompletableFuture 和 FutureTask 同属于 Future 接口的实现类，都可以获取线程的执行结果。



简化异步编程的复杂性，提供了函数式编程的能力，可以通过回调的方式处理计算结果，并且提供了转换和组合 CompletableFuture 的方法

CompletableFuture 启动异步任务

创建异步操作对象

CompletableFuture 提供了四个静态方法来创建一个异步操作。

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

1、runXxxx 都是没有返回结果的，supplyXxx 都是可以获取返回结果的

2、可以传入自定义的线程池，否则就用默认的线程池；

```java
public class ThreadTest {
    //当前系统中池只有一两个，每个异步任务，提交给线程池
    public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
    public static void main(String[] args) {
        System.out.println("main......start.....");
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            System.out.println("当前线程：" + Thread.currentThread().getId());
            int i = 10 / 2;
            System.out.println("运行结果：" + i);
        }, executor);
        System.out.println("main......end.....");
```

运行结果

```
main......start.....
main......end.....
当前线程：9
运行结果：5
```

```java
public class ThreadTest {
    //当前系统中池只有一两个，每个异步任务，提交给线程池
    public  static ExecutorService executor = Executors.newFixedThreadPool( n
    public static void main(String[] args) throws ExecutionException, Interru
        System.out.println("main......start.....");
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(()->
            System.out.println("当前线程：" + Thread.currentThread().getId());
            int i = 10 / 2;
            System.out.println("运行结果：" + i);
            return i;
        }, executor);
        Integer integer = future.get();
        System.out.println("main......end....." + integer);
```

运行结果

```
main......start.....
当前线程：9
运行结果：5
main......end.....5
```

计算完成时回调方法

```
public CompletableFuture<T> whenComplete(BiConsumer<? super T,? super Throwable> action);
public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T,? super Throwable>
action);
public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T,? super Throwable>
action, Executor executor);

public CompletableFuture<T> exceptionally(Function<Throwable,? extends T> fn);
```

whenComplete 可以处理正常和异常的计算结果，exceptionally 处理异常情况。

whenComplete 和 whenCompleteAsync 的区别：

whenComplete：是执行当前任务的线程继续执行 whenComplete 的任务。

whenCompleteAsync：是把 whenCompleteAsync 这个任务继续提交给线程池来进行执行。

方法不以 Async 结尾，意味着 Action 使用相同的线程执行，而 Async 可能会使用其他线程执行（如果是使用相同的线程池，也可能会被同一个线程选中执行）

```java
public class ThreadTest {
    //当前系统中池只有一两个，每个异步任务，提交给线程池
    public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
    public static void main(String[] args) throws ExecutionException, InterruptedException
        System.out.println("main......start.....");
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            System.out.println("当前线程：" + Thread.currentThread().getId());
            int i = 10 / 2;
            System.out.println("运行结果：" + i);
            return i;
        }, executor).whenComplete((res,exception) -> {
            System.out.println("异步任务成功完成了...结果是：" + res + "异常是：" + exception);
```

运行结果

```
main......start.....
当前线程：9
运行结果：5
异步任务成功完成了...结果是：5异常是：null
```

```java
public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
public static void main(String[] args) throws ExecutionException, InterruptedException
    System.out.println("main......start.....");
    CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
        System.out.println("当前线程：" + Thread.currentThread().getId());
        int i = 10 / 0;
        System.out.println("运行结果：" + i);
        return i;
    }, executor).whenComplete((res,exception) -> {
        System.out.println("异步任务成功完成了...结果是：" + res + "异常是：" + exception);
```

运行结果

```
main......start.....
当前线程：9
异步任务成功完成了...结果是：null异常是：java.util.concurrent.CompletionException: java.lang.ArithmeticException: / by zero
```

```java
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
    System.out.println("当前线程：" + Thread.currentThread().getId());
    int i = 10 / 0;
    System.out.println("运行结果：" + i);
    return i;
}, executor).whenComplete((res,exception) -> {
    //虽然能得到异常信息，但是没法修改返回数据
    System.out.println("异步任务成功完成了...结果是：" + res + "异常是：" + exception);
}).exceptionally(throwable -> {
    //可以感知异常，同时返回默认值
    return 10;
});
Integer integer= future.get();
System.out.println("main......end....." + integer);
```

运行结果

```
main......start.....
当前线程：9
异步任务成功完成了...结果是：null异常是：java.util.concurrent.CompletionException: java.lang.ArithmeticException: / by zero
main......end.....10
```

handle 方法

和 complete 一样，但handle可对结果做最后的处理（可处理异常），可改变返回值。

```
public <U> CompletionStage<U> handle(BiFunction<? super T, Throwable, ? extends U> fn);
public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ? extends U>
fn);
public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ? extends U>
fn,Executor executor);
```

```java
public class ThreadTest {
    //当前系统中池只有一两个，每个异步任务，提交给线程池
    public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
    public static void main(String[] args) throws ExecutionException, InterruptedExceptio
        System.out.println("main......start.....");
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            System.out.println("当前线程：" + Thread.currentThread().getId());
            int i = 10 / 0;
            System.out.println("运行结果：" + i);
            return i;
        }, executor).handle((res,thr) -> {
            if (res != null) {
                return res * 2;
            }
            if (thr != null) {
                System.out.println("异步任务成功完成了...结果是：" + res + "异常是：" + thr);
                return 0;
            }
            return 0;
        });
        Integer integer= future.get();
        System.out.println("main......end....." + integer);
```

```
main......start.....
当前线程：9
异步任务成功完成了...结果是：null异常是：java.util.concurrent.CompletionException: java.lang.ArithmeticException: / by zero
main......end.....0
```

```java
public static void main(String[] args) throws ExecutionException, InterruptedExc
    System.out.println("main......start.....");
    CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
        System.out.println("当前线程：" + Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("运行结果：" + i);
        return i;
    }, executor).handle((res,thr) -> {
        if (res != null) {
            return res * 2;
        }
        if (thr != null) {
            System.out.println("异步任务成功完成了...结果是：" + res + "异常是：" + thr);
            return 0;
        }
        return 0;
    });
    Integer integer= future.get();
    System.out.println("main......end....." + integer);
```

运行结果

```
main......start.....
当前线程：9
运行结果：2
main......end.....4
```

线程串行化方法

以下方法可以将两个异步任务串联起来:

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

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

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

thenApply 方法：当一个线程依赖另一个线程时，获取上一个任务返回的结果，并返回当前任务的返回值。

thenAccept 方法：消费处理结果。接收任务的处理结果，并消费处理，无返回结果。

thenRun 方法：只要上面的任务执行完成，就开始执行 thenRun，只是处理完任务后，执行thenRun 的后续操作。

带有 Async 默认是异步执行的。同之前。

以上都要前置任务成功完成。

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

```java
public class ThreadTest {
    //当前系统中池只有一两个，每个异步任务，提交给线程池
    public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        System.out.println("main......start.....");
        CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {
            System.out.println("当前线程：" + Thread.currentThread().getId());
            int i = 10 / 4;
            System.out.println("运行结果：" + i);
            return i;
        }, executor).thenRunAsync(() -> {
            System.out.println("任务2启动了..." );
        },executor);
        System.out.println("main......end....." );
    }
```

运行结果

```
main......start.....
当前线程：9
运行结果：2
main......end.....
任务2启动了...
```

```java
public class ThreadTest {
    //当前系统中池只有一两个，每个异步任务，提交给线程池
    public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
    public static void main(String[] args) throws ExecutionException, InterruptedExcepti
        System.out.println("main......start.....");
        CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {
            System.out.println("当前线程：" + Thread.currentThread().getId());
            int i = 10 / 4;
            System.out.println("运行结果：" + i);
            return i;
        }, executor).thenAcceptAsync((res) -> {
            System.out.println("任务2启动了..." + res );
        },executor);
        System.out.println("main......end....." );
```

运行结果

```
main......start.....
当前线程：9
运行结果：2
main......end.....
任务2启动了...2
```

```java
public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
public static void main(String[] args) throws ExecutionException, InterruptedException {
    System.out.println("main......start.....");
    CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
        System.out.println("当前线程：" + Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("运行结果：" + i);
        return i;
    }, executor).thenApplyAsync((res) -> {
        System.out.println("任务2启动了..." + res );
        return "hello" + res;
    },executor);
    System.out.println("main......end....." + future.get());
```

运行结果

```
main......start.....
当前线程：9
运行结果：2
任务2启动了...2
main......end.....hello2
```

两任务组合 - 都要完成

runAfterBoth：组合两个 future，不需要获取 future 的结果，只需两个 future 处理完任务后，再处理该任务。

```
public CompletableFuture<Void> runAfterBoth(CompletionStage<?> other,
                                            Runnable action);

public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other,
                                                 Runnable action);

public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other,
                                                 Runnable action,
                                                 Executor executor);
```

```java
public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
public static void main(String[] args) throws ExecutionException, InterruptedException {
    System.out.println("main......start.....");
    CompletableFuture<Integer> future01 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务1线程：" + Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("任务1结束：" );
        return i;
    }, executor);
    CompletableFuture<String> future02 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务2线程：" + Thread.currentThread().getId());
        System.out.println("任务2结束：" );
        return "hello";
    }, executor);
    future01.runAfterBothAsync(future02,()->{
        System.out.println("任务3开始：");
    },executor);
    System.out.println("main......end.....");
```

```
main......start.....
任务1线程：9
任务1结束：
任务2线程：10
任务2结束：
main......end.....
任务3开始：
```

thenAcceptBoth：组合两个 future，获取两个 future 任务的返回结果，然后处理任务，没有返回值。

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

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

public <U> CompletableFuture<Void> thenAcceptBothAsync(
    CompletionStage<? extends U> other,
    BiConsumer<? super T, ? super U> action, Executor executor);
```

```java
public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
public static void main(String[] args) throws ExecutionException, InterruptedException {
    System.out.println("main......start.....");
    CompletableFuture<Integer> future01 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务1线程：" + Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("任务1结束：" );
        return i;
    }, executor);
    CompletableFuture<String> future02 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务2线程：" + Thread.currentThread().getId());
        System.out.println("任务2结束：" );
        return "hello";
    }, executor);
    future01.thenAcceptBothAsync(future02,(f1,f2)->{
        System.out.println("任务3开始...之前的结果："+ f1+"-->"+f2);
    },executor);
    System.out.println("main......end.....");
```

```
任务1线程：9
任务1结束：
任务2线程：10
任务2结束：
main......end.....
任务3开始...之前的结果：2-->hello
```

thenCombine：组合两个 future，获取两个 future 的返回结果，并返回当前任务的返回值。

```
public <U,V> CompletableFuture<V> thenCombine(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn);

public <U,V> CompletableFuture<V> thenCombineAsync(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn);

public <U,V> CompletableFuture<V> thenCombineAsync(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn, Executor executor);
```

```java
public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
public static void main(String[] args) throws ExecutionException, InterruptedException {
    System.out.println("main......start.....");
    CompletableFuture<Integer> future01 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务1线程： " + Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("任务1结束： " );
        return i;
    }, executor);
    CompletableFuture<String> future02 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务2线程： " + Thread.currentThread().getId());
        System.out.println("任务2结束： " );
        return "hello";
    }, executor);
    CompletableFuture<String> future=future01.thenCombineAsync(future02,(f1,f2)->{
        return f1+",,"+f2+"hahaha" ;
    },executor);
    System.out.println("main......end....." + future.get());
```

```
main......start.....
任务1线程： 9
任务1结束：
任务2线程： 10
任务2结束：
main......end.....2,,hello   hahaha
```

两任务组合 - 一个完成

runAfterEither：两个任务只要有一个执行完成，就处理后续任务，不需要获取 future 的结果，也没有返回值。

```
public CompletableFuture<Void> runAfterEither(CompletionStage<?> other,
                                              Runnable action);

public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other,
                                                   Runnable action);

public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other,
                                                   Runnable action,
                                                   Executor executor);
```

```java
public static void main(String[] args) throws ExecutionException, InterruptedEx
    System.out.println("main......start.....");
    CompletableFuture<Integer> future01 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务1线程： " + Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("任务1结束： " );
        return i;
    }, executor);
    CompletableFuture<String> future02 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务2线程： " + Thread.currentThread().getId());
        try {
            Thread.sleep( millis: 3000);
            System.out.println("任务2结束： " );
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return "hello";
    }, executor);
    future01.runAfterEitherAsync(future02,()->{
        System.out.println("任务3开始： " );
    },executor);
    System.out.println("main......end.....");
```

```
main......start.....
任务1线程： 9
任务1结束：
任务2线程： 10
main......end.....
任务3开始：
任务2结束：
```

acceptEither：两个任务只要有一个执行完成，获取它的返回值，处理任务，没有新的返回值。

```
public CompletableFuture<Void> acceptEither(
    CompletionStage<? extends T> other, Consumer<? super T> action);

public CompletableFuture<Void> acceptEitherAsync(
    CompletionStage<? extends T> other, Consumer<? super T> action);

public CompletableFuture<Void> acceptEitherAsync(
    CompletionStage<? extends T> other, Consumer<? super T> action,
    Executor executor);
```

```java
public static void main(String[] args) throws ExecutionException, InterruptedException {
    System.out.println("main......start.....");
    CompletableFuture<Object> future01 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务1线程： " + Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("任务1结束： " );
        return i;
    }, executor);
    CompletableFuture<Object> future02 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务2线程： " + Thread.currentThread().getId());
        try {
            Thread.sleep( millis: 3000);
            System.out.println("任务2结束： " );
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return "hello";
    }, executor);
    future01.acceptEitherAsync(future02,(res)->{
        System.out.println("任务3开始： " + res);
    },executor);
    System.out.println("main......end.....");
```

```
main......start...
任务1线程： 9
任务1结束：
任务2线程： 10
main......end.....
任务3开始： 2
任务2结束：
```

applyToEither：两个任务有一个执行完成，获取它的返回值，处理任务，并有新的返回值。

```
public <U> CompletableFuture<U> applyToEither(
    CompletionStage<? extends T> other, Function<? super T, U> fn);

public <U> CompletableFuture<U> applyToEitherAsync(
    CompletionStage<? extends T> other, Function<? super T, U> fn);

public <U> CompletableFuture<U> applyToEitherAsync(
    CompletionStage<? extends T> other, Function<? super T, U> fn,
    Executor executor);
```

```java
public static void main(String[] args) throws ExecutionException, InterruptedException {
    System.out.println("main......start.....");
    CompletableFuture<Object> future01 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务1线程： " + Thread.currentThread().getId());
        int i = 10 / 4;
        System.out.println("任务1结束： " );
        return i;
    }, executor);
    CompletableFuture<Object> future02 = CompletableFuture.supplyAsync(() -> {
        System.out.println("任务2线程： " + Thread.currentThread().getId());
        try {
            Thread.sleep( millis: 3000);
            System.out.println("任务2结束： " );
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return "hello";
    }, executor);
    CompletableFuture future = future01.applyToEitherAsync(future02,(res)->{
        return res.toString()+"--》任务3开始";
    },executor);
    System.out.println("main......end....." + future.get());
```

```
main......start.....
任务1线程： 9
任务1结束：
任务2线程： 10
main......end.....2--》任务3开始
任务2结束：
```

多任务组合

allOf：等待所有任务完成
anyOf：只要有一个任务完成

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

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

```
CompletableFuture<String> futureImg = CompletableFuture.supplyAsync(()->{
    System.out.println("查询商品的图片信息");
    return "hello.jsp";
});
CompletableFuture<String> futureAttr = CompletableFuture.supplyAsync(()->{
    System.out.println("查询商品的属性");
    return "黑色+256G";
});
CompletableFuture<String> futureDesc = CompletableFuture.supplyAsync(()->{
    System.out.println("查询商品介绍");
    return "华为";
});
futureImg.get();
futureAttr.get();
futureDesc.get();
```

```java
public class ThreadTest {
    //当前系统中池只有一两个，每个异步任务，提交给线程池
    public  static ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        CompletableFuture<String> futureImg = CompletableFuture.supplyAsync(()->{
            System.out.println("查询商品的图片信息");
            return "hello.jsp";
        });
        CompletableFuture<String> futureAttr = CompletableFuture.supplyAsync(()->{
            System.out.println("查询商品的属性");
            return "黑色+256G";
        });
        CompletableFuture<String> futureDesc = CompletableFuture.supplyAsync(()->{
            System.out.println("查询商品介绍");
            return "华为";
        });
        CompletableFuture<Void> allof = CompletableFuture.allOf(futureImg,futureAttr,futureDesc);
        allof.get();//等待所有结果完成
        System.out.println("main......end....."+ futureImg.get()+"-->"
                +futureAttr.get()+ "-->"+futureDesc.get());
```

```
查询商品的图片信息
查询商品的属性
查询商品介绍
main......end.....hello.jsp-->黑色+256G-->华为
```

```java
public static void main(String[] args) throws ExecutionException, InterruptedException {
    CompletableFuture<Object> futureImg = CompletableFuture.supplyAsync(()->{
        System.out.println("查询商品的图片信息");
        return "hello.jsp";
    },executor);
    CompletableFuture<Object> futureAttr = CompletableFuture.supplyAsync(()->{
        System.out.println("查询商品的属性");
        return "黑色+256G";
    },executor);
    CompletableFuture<Object> futureDesc = CompletableFuture.supplyAsync(()->{
        try {
            Thread.sleep( millis: 3000);
            System.out.println("查询商品介绍");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return "华为";
    },executor);
    CompletableFuture<Object> anyof = CompletableFuture.anyOf(futureImg,futureAttr,futureDesc);
    anyof.get();//等待所有结果完成
    System.out.println("main......end....."+ anyof.get());
```

```
查询商品的图片信息
查询商品的属性
main......end.....hello.jsp
查询商品介绍
```