https://mp.weixin.qq.com/s/Z7Yi2t09s3SnliFlu1ob1Q

# JUC中工具类CompletableFuture

**CompletableFuture**是java8中新增的一个类，算是对Future的一种增强，用起来很方便，也是会经常用到的一个工具类，熟悉一下。

## CompletionStage接口

* CompletionStage代表异步计算过程中的某一个阶段，一个阶段完成以后可能会触发另外一个阶段
* 一个阶段的计算执行可以是一个Function，Consumer或者Runnable。比如：stage.thenApply(x -> square(x)).thenAccept(x -> System.out.print(x)).thenRun(() -> System.out.println())
* 一个阶段的执行可能是被单个阶段的完成触发，也可能是由多个阶段一起触发。

## CompletableFuture类

* 在Java8中，CompletableFuture提供了非常强大的Future的扩展功能，可以帮助我们简化异步编程的复杂性，并且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，也提供了转换和组合 CompletableFuture 的方法。
* 它可能代表一个明确完成的Future，也有可能代表一个完成阶段（ CompletionStage ），它支持在计算完成以后触发一些函数或执行某些动作。
* 它实现了Future和CompletionStage接口

**常见的方法，熟悉一下：**

### runAsync 和 supplyAsync方法

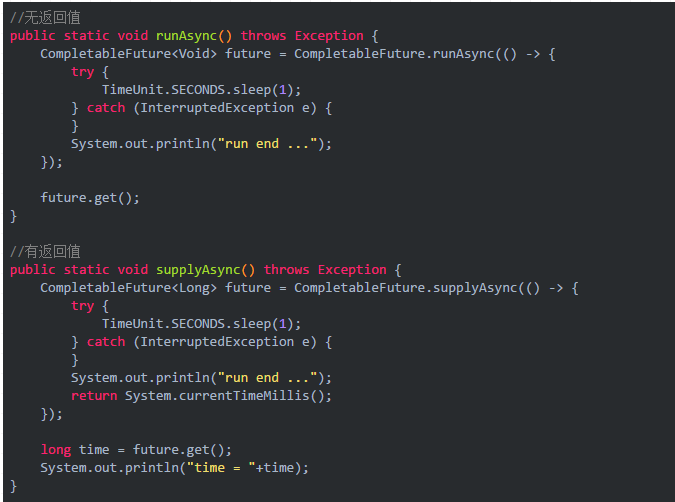
CompletableFuture 提供了四个静态方法来创建一个异步操作。

public 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)

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。以下所有的方法都类同。

* runAsync方法不支持返回值。
* supplyAsync可以支持返回值。

**示例代码**



### 计算结果完成时的回调方法

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

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)

可以看到Action的类型是BiConsumer它可以处理正常的计算结果，或者异常情况。

**whenComplete 和 whenCompleteAsync 的区别：**

* whenComplete：是执行当前任务的线程执行继续执行 whenComplete 的任务。
* whenCompleteAsync：是执行把 whenCompleteAsync 这个任务继续提交给线程池来进行执行。

**示例代码**



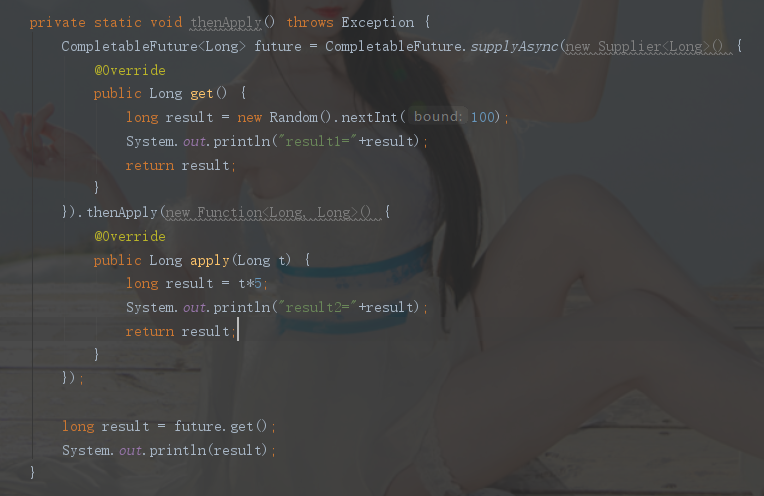
### thenApply 方法

当一个线程依赖另一个线程时，可以使用 thenApply 方法来把这两个线程串行化。

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)

Function  
T：上一个任务返回结果的类型  
U：当前任务的返回值类型

**示例代码**



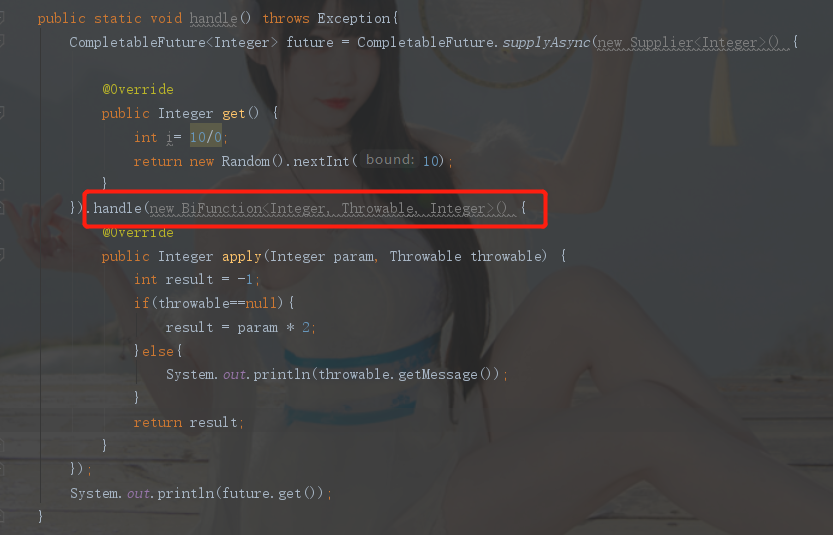
第二个任务依赖第一个任务的结果。

### handle 方法

handle 是执行任务完成时对结果的处理。  
handle 方法和 thenApply 方法处理方式基本一样。不同的是 handle 是在任务完成后再执行，还可以处理异常的任务。thenApply 只可以执行正常的任务，任务出现异常则不执行 thenApply 方法。

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);

**示例代码**



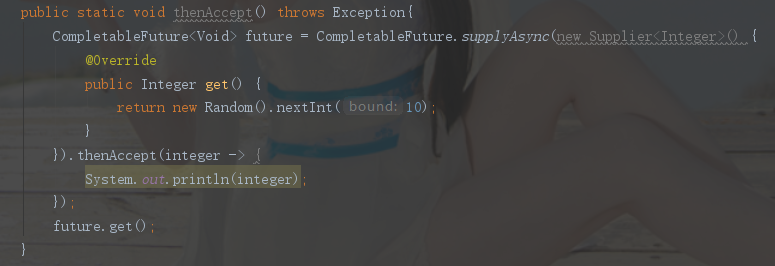
从示例中可以看出，在 handle 中可以根据任务是否有异常来进行做相应的后续处理操作。而 thenApply 方法，如果上个任务出现错误，则不会执行 thenApply 方法。

### thenAccept 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。

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);

**示例代码**



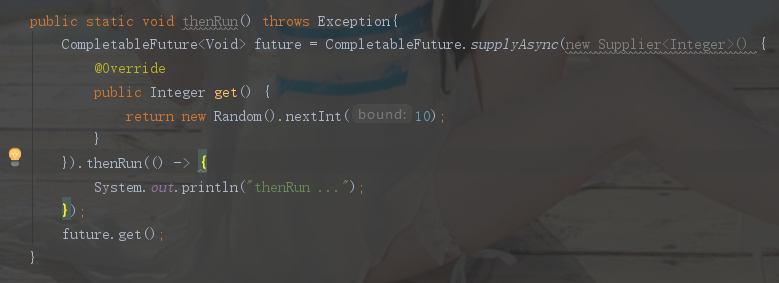
从示例代码中可以看出，该方法只是消费执行完成的任务，并可以根据上面的任务返回的结果进行处理。并没有后续的输错操作。

### thenRun 方法

跟 thenAccept 方法不一样的是，不关心任务的处理结果。只要上面的任务执行完成，就开始执行 thenAccept 。

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

**示例代码**



该方法同 thenAccept 方法类似。不同的是上个任务处理完成后，并不会把计算的结果传给 thenRun 方法。只是处理完任务后，执行 thenAccept 的后续操作。

### thenCombine  合并任务

thenCombine 会把 两个 CompletionStage 的任务都执行完成后，把两个任务的结果一块交给 thenCombine 来处理。

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

**示例代码**



### thenAcceptBoth

当两个CompletionStage都执行完成后，把结果一块交给thenAcceptBoth来进行消耗

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

**示例代码**

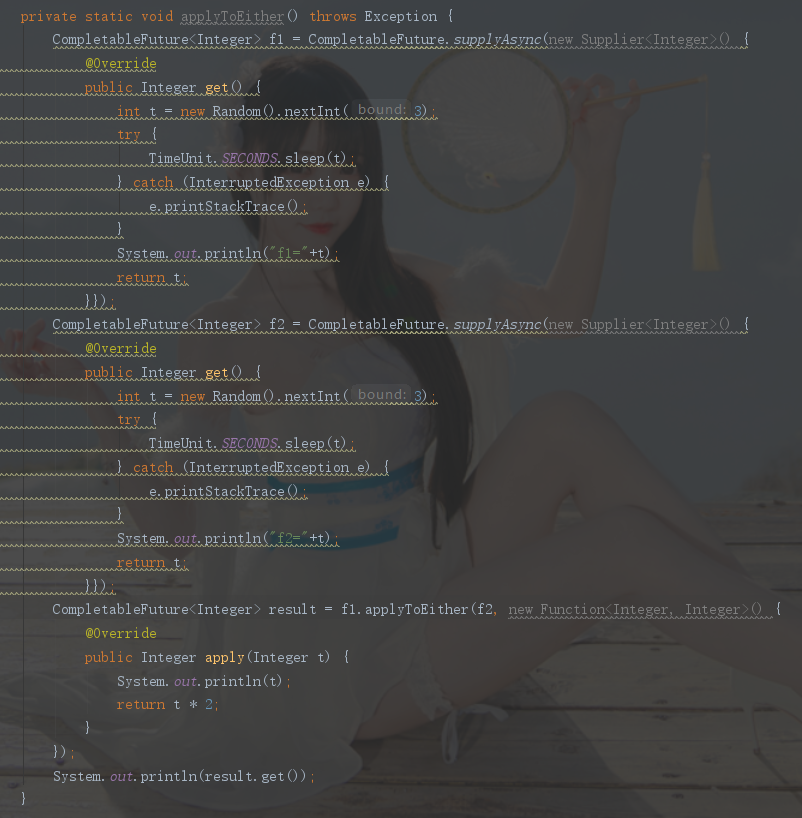


### applyToEither 方法

两个CompletionStage，谁执行返回的结果快，我就用那个CompletionStage的结果进行下一步的转化操作。

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

**示例代码**



### acceptEither 方法

两个CompletionStage，谁执行返回的结果快，我就用那个CompletionStage的结果进行下一步的消耗操作。

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

**示例代码**

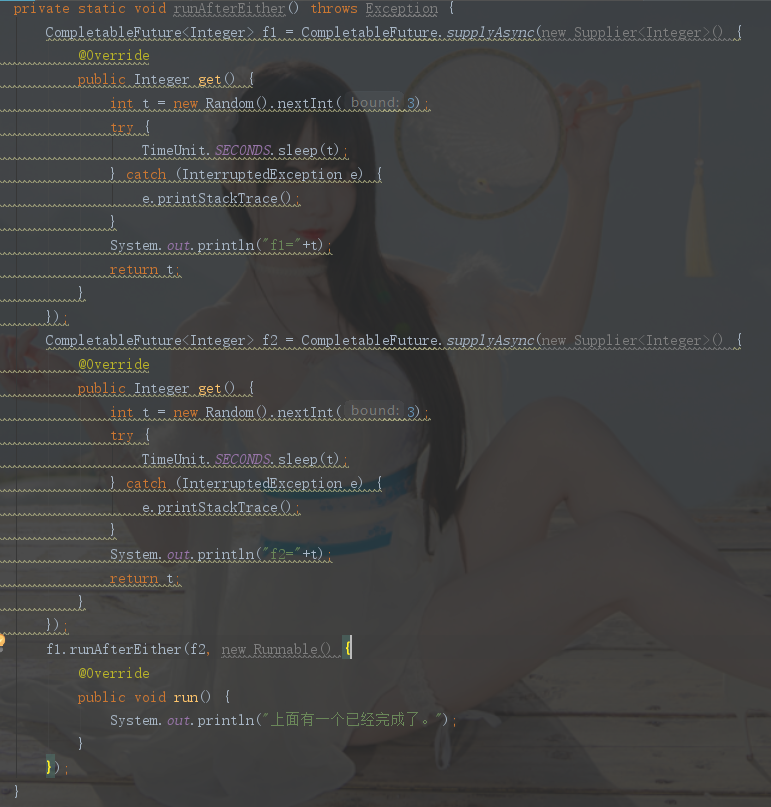


### runAfterEither 方法

两个CompletionStage，任何一个完成了都会执行下一步的操作（Runnable）

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

**示例代码**

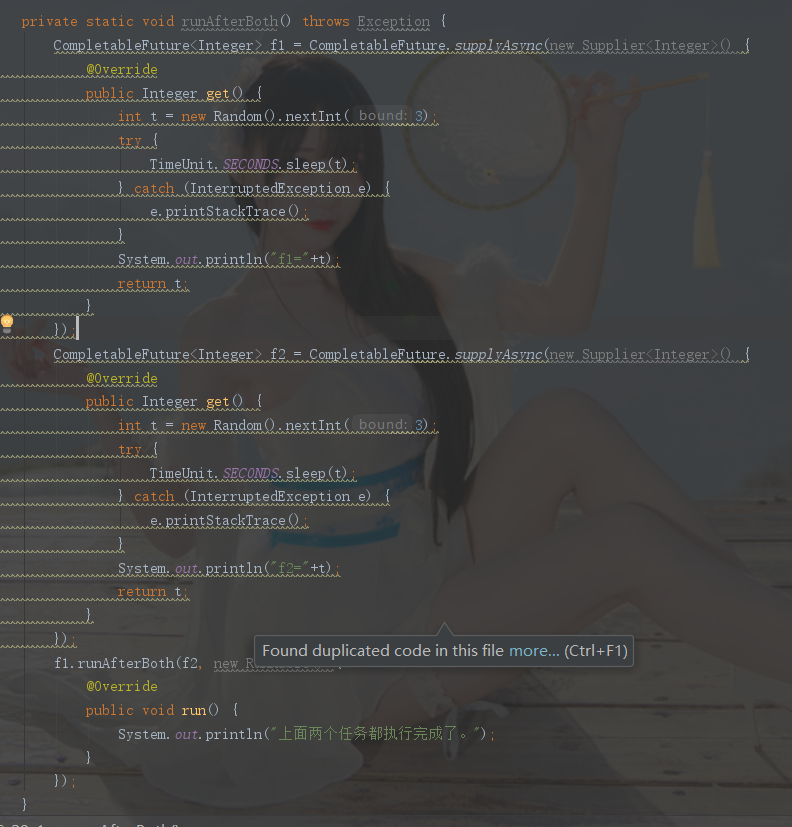


### runAfterBoth

两个CompletionStage，都完成了计算才会执行下一步的操作（Runnable）

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

**示例代码**



### thenCompose 方法

thenCompose 方法允许你对两个 CompletionStage 进行流水线操作，第一个操作完成时，将其结果作为参数传递给第二个操作。

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

**示例代码**

