# **Java8新的异步编程方式 CompletableFuture(一)**

# **一. Future**

JDK 5引入了Future模式。Future接口是Java多线程Future模式的实现，在java.util.concurrent包中，可以来进行异步计算。

Future模式是多线程设计常用的一种设计模式。Future模式可以理解成：我有一个任务，提交给了Future，Future替我完成这个任务。期间我自己可以去做任何想做的事情。一段时间之后，我就便可以从Future那儿取出结果。

Future的接口很简单，只有五个方法。

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();

V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException;}

Future接口的方法介绍如下：

* 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，uint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

一般情况下，我们会结合Callable和Future一起使用，通过ExecutorService的submit方法执行Callable，并返回Future。

ExecutorService executor = Executors.newCachedThreadPool();

Future<String> future = executor.submit(() -> { //Lambda 是一个 callable， 提交后便立即执行，这里返回的是 FutureTask 实例

System.out.println("running task");

Thread.sleep(10000);

return "return task";

});

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

}

System.out.println("do something else"); //前面的的 Callable 在其他线程中运行着，可以做一些其他的事情

try {

System.out.println(future.get()); //等待 future 的执行结果，执行完毕之后打印出来

} catch (InterruptedException e) {

} catch (ExecutionException e) {

} finally {

executor.shutdown();

}

比起future.get()，其实更推荐使用get (long timeout, TimeUnit unit) 方法，设置了超时时间可以防止程序无限制的等待future的结果。

# **二. CompletableFuture介绍**

## **2.1 Future模式的缺点**

1、Future虽然可以实现获取异步执行结果的需求，但是它没有提供通知的机制，我们无法得知Future什么时候完成。

2、要么使用阻塞，在future.get()的地方等待future返回的结果，这时又变成同步操作。要么使用isDone()轮询地判断Future是否完成，这样会耗费CPU的资源。

## **2.2 CompletableFuture介绍**

Netty、Guava分别扩展了Java 的 Future 接口，方便异步编程。

Java 8新增的CompletableFuture类正是吸收了所有Google Guava中ListenableFuture和SettableFuture的特征，还提供了其它强大的功能，让Java拥有了完整的非阻塞编程模型：Future、Promise 和 Callback(在Java8之前，只有无Callback 的Future)。

CompletableFuture能够将回调放到与任务不同的线程中执行，也能将回调作为继续执行的同步函数，在与任务相同的线程中执行。它避免了传统回调最大的问题，那就是能够将控制流分离到不同的事件处理器中。

CompletableFuture弥补了Future模式的缺点。在异步的任务完成后，需要用其结果继续操作时，无需等待。可以直接通过thenAccept、thenApply、thenCompose等方式将前面异步处理的结果交给另外一个异步事件处理线程来处理。

# **三. CompletableFuture特性**

## **3.1 CompletableFuture的静态工厂方法**

| **方法名** | **描述** |
| --- | --- |
| runAsync(Runnable runnable) | 使用ForkJoinPool.commonPool()作为它的线程池执行异步代码。 |
| runAsync(Runnable runnable, Executor executor) | 使用指定的thread pool执行异步代码。 |
| supplyAsync(Supplier<U> supplier) | 使用ForkJoinPool.commonPool()作为它的线程池执行异步代码，异步操作有返回值 |
| supplyAsync(Supplier<U> supplier, Executor executor) | 使用指定的thread pool执行异步代码，异步操作有返回值 |

runAsync 和 supplyAsync 方法的区别是runAsync返回的CompletableFuture是没有返回值的。

CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {

System.out.println("Hello");

});

try {

future.get();

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

System.out.println("CompletableFuture");

而supplyAsync返回的CompletableFuture是由返回值的，下面的代码打印了future的返回值。

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");

try {

System.out.println(future.get());

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

System.out.println("CompletableFuture");

## **3.2 Completable**

| **方法名** | **描述** |
| --- | --- |
| complete(T t) | 完成异步执行，并返回future的结果 |
| completeExceptionally(Throwable ex) | 异步执行不正常的结束 |

future.get()在等待执行结果时，程序会一直block，如果此时调用complete(T t)会立即执行。

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");

future.complete("World");

try {

System.out.println(future.get());

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

执行结果：

World

可以看到future调用complete(T t)会立即执行。但是complete(T t)只能调用一次，后续的重复调用会失效。

如果future已经执行完毕能够返回结果，此时再调用complete(T t)则会无效。

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

future.complete("World");

try {

System.out.println(future.get());

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

执行结果：

Hello

如果使用completeExceptionally(Throwable ex)则抛出一个异常，而不是一个成功的结果。

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");

future.completeExceptionally(new Exception());

try {

System.out.println(future.get());

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

执行结果：

java.util.concurrent.ExecutionException: java.lang.Exception...