25 Netty的Future是如何做到简捷易用的

更新时间：2020-08-17 09:52:51

> 受苦的人，没有悲观的权利。——尼采

前言

你好，我是彤哥。

上一节，我们一起学习了 Netty 中使用到的队列，它是对 Java 原生队列的增强，其实，Netty 对 Java 原生的增强还有很多，比如，本节将要讨论的 Future。

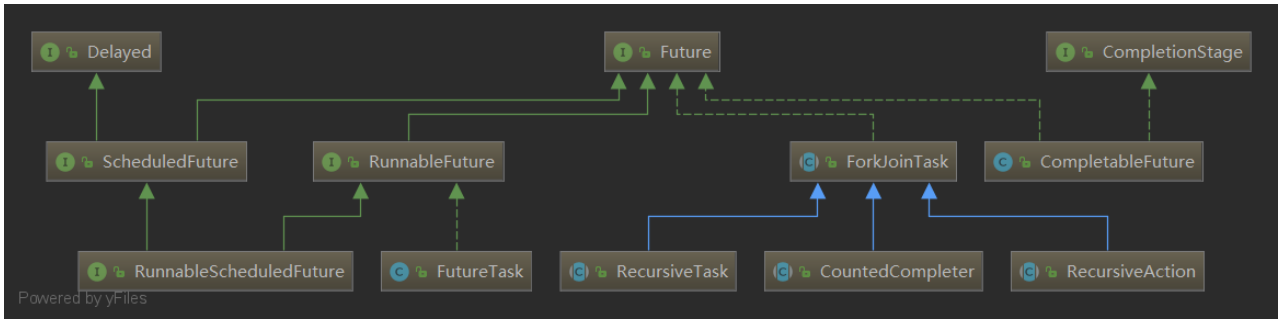
Java 原生提供了一个 Future 接口，可以用来获取线程池中任务执行的结果，但是，它只提供了简单的功能，不能实现诸如判断任务是否成功执行完毕、任务执行完后来个回调、手动设置任务执行的结果等功能。

为了实现这些功能，Netty 扩展了 Java 原生的 Future 接口，并增加了一些实用的方法，但是，一不小心，Netty 也搞出了一个小 bug，而且，到目前为止，这个 Bug 依然还未修复，你知道这个 bug 是什么吗？

今天，就让我们一起来学习 Netty 增强的 Future。

Java 原生 Future

首先，让我们来回顾下 Java 原生的 Future。



所谓 Future，是未来、将来的意思，在 Java 中，它是指在未来做些什么，它跟 Runnable 组合就变成了 RunnableFuture，它跟 Scheduled 组合就变成了 ScheduledFuture，它跟 ForkJoin 组合就变成了 ForkJoinTask，它跟 Completable 组合就变成了 CompletableFuture，在 JUC 包下面主要提供了上图中这几种实现，其实，远远不止这些，还有很多在内部类里面，我没有列出来，本节，我们主要看两个简单的实现：FutureTask 和 CompletableFuture。

> RecursiveAction 表示无返回值任务，RecursiveTask 表示有返回值任务，CountedCompleter 表示有回调的无返回值任务，它们三个都是用在 ForkJoinPool 中的，有兴趣的同学可以自己看下相关的源码。

Future 接口

首先，我们先看看 Future 接口提供了哪些能力：

```
package java.util.concurrent;

public interface Future<V> {
    /**
     * <p>After this method returns, subsequent calls to {@link #isDone} will
     * always return {@code true}.  Subsequent calls to {@link #isCancelled}
     * will always return {@code true} if this method returned {@code true}.
     */
    // 取消，请记住上面的话：
    //     在这个方法返回之后，后面的isDone()方法总是返回true。
    //     如果这个方法返回true，后面的isCancelled()方法总是返回true。
    boolean cancel(boolean mayInterruptIfRunning);
    // 判断是否已取消
    boolean isCancelled();
    // 判断是否已完成，已完成有多层含义：已正常完成、已异常完成、已取消
    boolean isDone();
    // 获取结果，可以设置等待时间
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

Future 接口除了提供获取任务执行结果的方法外，还提供了取消任务、判断任务是否已取消、是否已完成的方法，不过，这里的 isDone () 方法其实代表了三种状态：已正常完成、已异常完成、已取消，这是有点混乱的地方。

FutureTask

FutureTask，用于在未来获取线程池中任务执行的结果，它的主要方法就是 get ()，调用 get () 时，如果任务未执行完毕会阻塞当前线程直到任务执行完毕并返回，如果任务已执行完毕则会立即返回任务执行的结果。它的实现原理很简单，是很多异步框架实现同步的基本原理，比如，Dubbo 中同步调用的实现方式就是参考 FutureTask 的实现的。

那么，FutureTask 该如何使用呢？请看下面的例子：

```
public class FutureTaskTest {

    // 创建一个线程池
    private static final ExecutorService THREAD_POOL = Executors.newFixedThreadPool(8);

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        // 执行第一个任务
        Future<Integer> future1 = THREAD_POOL.submit(() -> {
            // 阻塞1秒
            LockSupport.parkNanos(TimeUnit.SECONDS.toNanos(1));
            // 返回1
            return 1;
        });
        // 执行第二个任务
        Future<Integer> future2 = THREAD_POOL.submit(() -> {
            // 阻塞2秒
            LockSupport.parkNanos(TimeUnit.SECONDS.toNanos(2));
            // 返回2
            return 2;
        });

        // 把两个任务执行的结果相加
        int result = future1.get() + future2.get();
        // 打印
        System.out.println("result=" + result);
    }
}
```

在这个例子中，我们创建了一个固定大小的线程池，提交了两个任务，一个阻塞 1 秒后返回 1，一个阻塞 2 秒后返回 2，在主线程中，我们调用两个 Future 的 get () 方法获取任务执行的结果，并把它们相加，最后，打印输出结果。

运行这段代码，我们会发现，大概会在 2 秒后打印结果，也就是主线程一共等待了 2 秒钟，具体来说，是在调用两个 Future.get () 的时候其中一个等待了 2 秒钟，这两个 get () 方法有个短板效应（好像不太准备，长板效应是不是好点 ^^），以最晚返回的为准。

在这两例子中，其实是有两次阻塞，第一次是 future1.get () 阻塞了 1 秒，返回之后，调用 future2.get ()，又阻塞了 1 秒，才返回。如果把 future1 和 future2 换下位置会怎样呢？那就只有一次阻塞了，因为调用 future2.get () 的时候阻塞了 2 秒，此时，再调用 future1.get () 的时候，future1 对应的任务早都执行完了，所以会立即返回，不会再次阻塞。

那么，这个 get () 方法的阻塞是如何实现的呢？

让我们先看看 FutureTask 有哪些属性：

```
public class FutureTask<V> implements RunnableFuture<V> {
    // 状态
    private volatile int state;
    private static final int NEW          = 0;
    private static final int COMPLETING   = 1;
    private static final int NORMAL       = 2;
    private static final int EXCEPTIONAL  = 3;
    private static final int CANCELLED    = 4;
    private static final int INTERRUPTING = 5;
    private static final int INTERRUPTED  = 6;

    // 任务
    private Callable<V> callable;
    // 返回值（正常的返回值或者异常）
    private Object outcome;
    // 运行任务的线程（防止多个线程同时运行了这个任务）
    private volatile Thread runner;
    // 等待队列，即调用者，多个调用者组成队列
    private volatile WaitNode waiters;
}
```

FutureTask 主要有五个重要的属性：

- state，状态，状态有七种，对于一个正常结束的任务，它的状态会从 NEW 到 COMPLETING，最后变成 NORMAL，表示正常结束；对于一个异常结束的任务，它的状态会从 NEW 到 COMPLETING，再到 EXCEPTIONAL，表示异常结束；
- callable，任务，外部传入的待执行的任务，任务执行完了会把这个字段置空；
- outcome，任务执行的结果，如果任务正常执行完毕则返回值会存储在里面，如果任务异常执行完成则保存的是异常信息，当调用 get () 方法时，如果任务已结束则根据状态变量的值判断是抛出异常还是正常返回值；另外，可以看到这个变量没有使用 volatile 关键字修饰，因为它总是在 state 调用之后被调用，所以，state 的 volatile 相当于对它也起到了保护作用；
- runner，运行任务的线程，当有一个线程运行任务时，设置该值，当有其它线程也想运行该任务时，则会直接返回；
- waiters，等待队列，在任务未结束之前，调用 get () 方法的所有线程都将进入这个队列中等待，当任务完成时，会依次唤醒这些等待的线程。

根据这五个属性，让我们脑补一下 FutureTask 正常执行结束的流程：

1. 初始化时，state 变量的值为 NEW；
2. 此时，如果调用 get () 方法，检测到此时 state 为 NEW，新建一个 WaitNode 节点并入队，当然了，此时 waiters 只有一个节点，如果另一个线程也调用了 get () 方法，那个线程也会入队；
3. 轮到任务运行了，运行之前检查状态是不是 NEW，并把运行的线程设置到 runner 变量中，这样下一个线程如果要运行检测到 runner 中有值了，就不会再重复执行任务了；
4. 当任务运行完毕，将 state 设置为 COMPLETING，并把任务执行的结果设置到 outcome 中；
5. 因为是正常运行结束，所以，最后，将 state 修改为 NORMAL 状态；
6. 任务运行完毕，将 waiters 中阻塞的线程全部唤醒；
7. get () 方法返回结果，调用着获取到结果做后续业务处理；

> 上面只是我们脑补的过程，实际情况到底是不是这样呢？这不在本节的范围之内，有兴趣的同学可以参考下这篇文章：死磕 java 线程系列之线程池深入解析 —— 未来任务执行流程

好了，FutureTask 看完了，我们再来看看 CompletableFuture。

CompletableFuture

CompletableFuture，使用它你可以在任务执行完毕之后执行一系列的回调，不管是正常还是异常结束。

那么，它该怎么使用呢？让我们来看一个例子：

```
public class CompletableFutureTest {

    // 创建一个线程池
    private static Executor threadPool = Executors.newFixedThreadPool(5);

    public static void main(String[] args) throws IOException, ExecutionException, InterruptedException {
        // 执行一个有返回值的任务
        CompletableFuture<?> future = CompletableFuture.supplyAsync(() -> {
            LockSupport.parkNanos(TimeUnit.SECONDS.toNanos(2));
            System.out.println("inner thread: " + Thread.currentThread().getName());
//            throw new RuntimeException();
            return "aaaaa";
            // 如果执行异常，会进入下面这个回调
        }, threadPool).exceptionally(e -> {
            System.out.println("exception: " + e + ", thread: " + Thread.currentThread().getName());
            return "bbbbb";
            // 执行一个有返回值的回调
        }).thenApplyAsync(s -> {
            System.out.println("result: " + s + ", thread: " + Thread.currentThread().getName());
            return "ccccc";
            // 执行一个无返回值的回调
        }, threadPool).thenAcceptAsync(s -> {
            System.out.println("result2: " + s + ", thread: " + Thread.currentThread().getName());
            // 再执行一个有返回值的回调
        }, threadPool).thenApplyAsync(s -> {
            System.out.println("result3: " + s + ", thread: " + Thread.currentThread().getName());
            return "ddddd";
        }, threadPool);

        System.out.println("completableFuture running!");

        // 调用get()方法
        System.out.println("get: " + future.get());

    }
}
```

在这个例子中，我有两个问题：

1. 这个 get () 方法获取的返回值到底是什么？
2. 如果把任务体中的抛出异常注释打开，并把下面一行注释掉，这个 get () 获取到的返回值又是什么？

运行一下这个程序，你会发现最终 get () 方法获取的返回值都是 `ddddd`，也就是说原始任务的返回值被吃掉了，这是这个类比较迷的一点。

另外，CompletableFuture 中的很多方法都有同步和异步两种方式，针对异步方式，你还可以设置使用哪个线程池来运行，如果没有指定线程池，默认使用的是 `ForkJoinPool.commonPool()` 来运行。

这个类里面大量使用了跟 ForkJoinPool 相关的代码，比如对任务或者回调的包装都是继承自 ForkJoinTask，它已经不是一个纯粹的类了，跟 ForkJoinPool 的耦合特别严重，这不是一个良好的编程范式，所以，这个类我不打算深入讲解。

既然，Java 原生提供了 CompletableFuture 这个可以支持回调的 Future，那么，Netty 为什么还要实现自己的 Future 呢？总结起来，原因大概有以下两点：

1. CompletableFuture 是 Java8 新加入的类， 对于早期的 Netty 无法吃到这个福利；
2. CompletableFuture 的实现过于复杂，且强耦合于 ForkJoinPool，这不是良好的设计，所以，Netty 并不打算使用它；

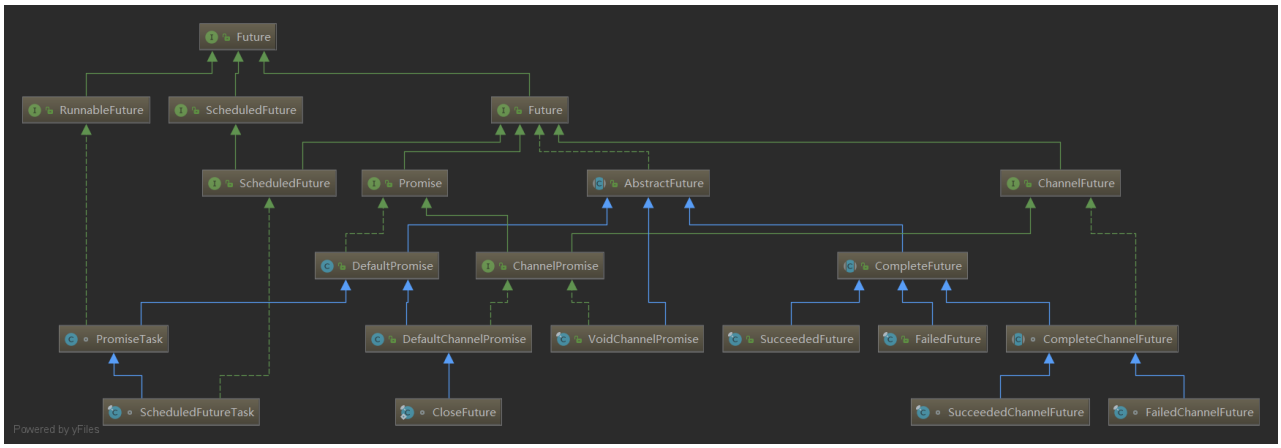
所以，下面就让我们看看 Netty 是如何实现自己的 Future 的。

Netty 的 Future

Netty 的 Future 扩展自 Java 原生的 Future，而且又提供了一种可以修改结果的增强，叫作 Promise。

针对 Future 的学习，我们同样使用从宏观到微观的分析方法。

继承体系



其实，Netty 中的 Future 远远不止这些，这个图中只列举了一部分 Future，还有诸如带进度条的 Future 等不常用的没有列进来。

上面这个图，我们可以分成几个部分来看：

- 左边，跟 Runnable 相关，表示的是任务，主要实现类有 PromiseTask、ScheduledFutureTask 等；
- 中间，主要是 Promise 相关，经典实现 DefaultPromise；
- 右边，跟 Complete 相关，表示完成，Netty 将完成分成两种状态，即 Succeeded 和 Failed，非常清晰；
- 最后一部分，跟 Channel 相关，它可以跟 Promise 和 Complete 任意组合；

可以看到，Netty 中 Future 的继承体系非常清晰，而且看一眼名字就大概知道是干什么的，除了结构清晰以外，还有另外一点值得说明的是，Netty 中所有 Future 子类的代码也是比较接近的，看懂了一个，其它的基本很容易就能看懂，而 Java 原生的 Future 体系，可以说，每一个 Future 的实现都不一样，这让人感到很痛苦。

好了，既然 Netty 的 Future 是对 Java 原生 Future 的增强，那我们就来看看它到底增强了哪些方法吧。

增强的方法

Netty 中 Future 的增强主要分成两级，一级是 Future，一级是 Promise。

我们先来看看 Future 这个接口：

```java
package io.netty.util.concurrent;

public interface Future<V> extends java.util.concurrent.Future<V> {
    // 返回是否成功
    boolean isSuccess();
    // 返回是否可取消
    boolean isCancellable();
    // 返回异常，如果有异常的话
    Throwable cause();
    // 添加或移除回调的监听器
    Future<V> addListener(GenericFutureListener<? extends Future<? super V>> listener);
    Future<V> addListeners(GenericFutureListener<? extends Future<? super V>>... listeners);
    Future<V> removeListener(GenericFutureListener<? extends Future<? super V>> listener);
    Future<V> removeListeners(GenericFutureListener<? extends Future<? super V>>... listeners);
    // 阻塞直到完成（内部调用await()方法），如果有异常则会抛出异常
    Future<V> sync() throws InterruptedException;
    Future<V> syncUninterruptibly();
    // 等待完成
    Future<V> await() throws InterruptedException;
    Future<V> awaitUninterruptibly();
    boolean await(long timeout, TimeUnit unit) throws InterruptedException;
    boolean await(long timeoutMillis) throws InterruptedException;
    boolean awaitUninterruptibly(long timeout, TimeUnit unit);
    boolean awaitUninterruptibly(long timeoutMillis);
    // 立即获取结果，如果还没有完成则返回null，不会阻塞
    V getNow();
    // 取消
    @Override
    boolean cancel(boolean mayInterruptIfRunning);
}
```

可以看到，Netty 的 Future 接口增加了很多语义比较明确的方法，比如判断是否成功完成、是否可取消、返回任务执行异常、明确的阻塞或者等待方法、立即获取结果的方法等，另外，最有用的要数 Listener 回调机制了，这是 Java 原生 Future 所欠缺的，虽然 Java 中的 CompletableFuture 也提供了回调机制，但是，使用 CompletableFuture 的回调连原始任务执行的结果都不知道了，显然是有问题的，而且里面耦合了太多 ForkJoinPool 的代码，也不符合 Netty 的极简原则。

好了，我们再看看另一级增强 ——Promise：

```
public interface Promise<V> extends Future<V> {
    // 显式地将结果设置为成功，如果设置失败将抛出异常
    Promise<V> setSuccess(V result);
    // 尝试将结果设置为成功，如果设置失败将返回false
    boolean trySuccess(V result);
    // 显式地将结果设置为失败，如果设置失败将抛出异常
    Promise<V> setFailure(Throwable cause);
    // 尝试将结果设置为失败，如果设置失败将返回false
    boolean tryFailure(Throwable cause);
    // 设置为不可取消
    boolean setUncancellable();
    // 以下为Future接口方法的重新定义（重写）
    @Override
    Promise<V> addListener(GenericFutureListener<? extends Future<? super V>> listener);
    @Override
    Promise<V> addListeners(GenericFutureListener<? extends Future<? super V>>... listeners);
    @Override
    Promise<V> removeListener(GenericFutureListener<? extends Future<? super V>> listener);
    @Override
    Promise<V> removeListeners(GenericFutureListener<? extends Future<? super V>>... listeners);
    @Override
    Promise<V> await() throws InterruptedException;
    @Override
    Promise<V> awaitUninterruptibly();
    @Override
    Promise<V> sync() throws InterruptedException;
    @Override
    Promise<V> syncUninterruptibly();
}
```

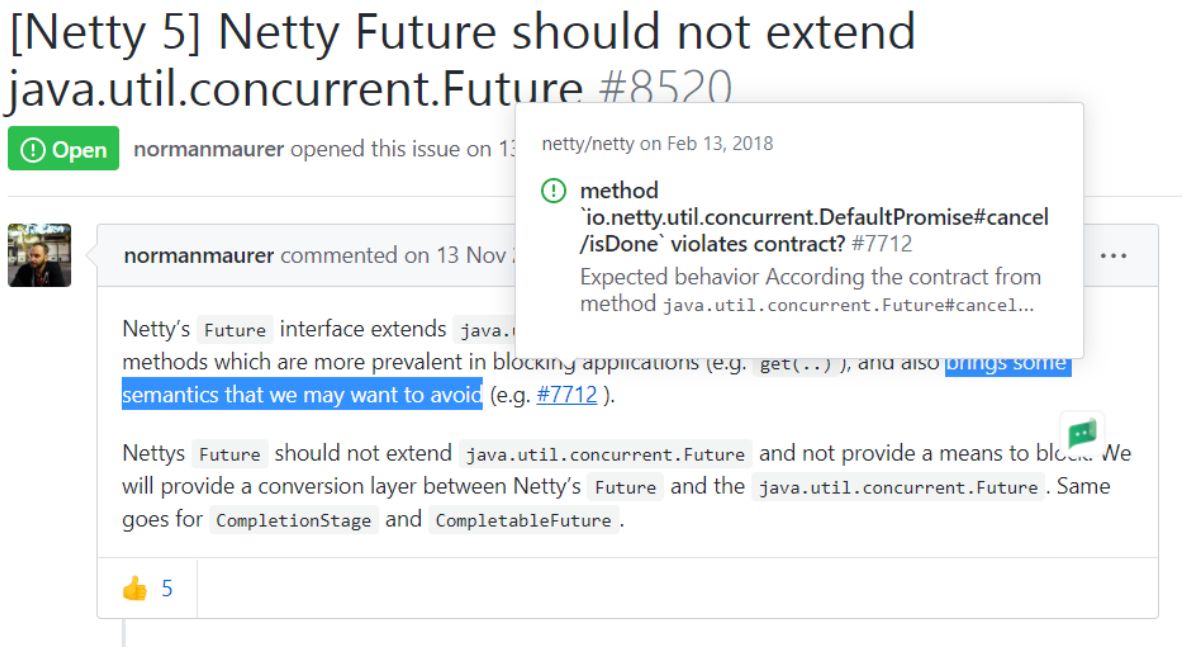
Promise 的主要作用就是提供了显式修改结果的方法，它可以把结果修改为成功或者失败，另外，还提供了设置不可取消的方法，我们开篇说的小 bug 就出在这个 `setUncancellable()` 方法上，为什么这么说呢？让我们看个例子：

```
public class DefaultPromiseIsDoneBugTest {

    public static void main(String[] args) {
        // 创建一个promise
        Promise<?> promise = GlobalEventExecutor.INSTANCE.newPromise();
        // 设置为不可取消
        promise.setUncancellable();
        // 调用取消的方法
        boolean cancel = promise.cancel(false);
        // 调用是否完成的方法
        boolean isDone = promise.isDone();
        // 打印
        System.out.println(cancel);
        System.out.println(isDone);
        System.out.println(promise.isCancelled());
    }

}
```

相信你也可以预测到这段代码运行的结果，没错，它返回的结果是三个 false，乍看之下，似乎没什么问题。确实没什么大问题，让我们再回顾一下 Java 原生的 Future 接口中 cancel () 方法的注释：

```java
public interface Future<V> {
    /**
     * <p>After this method returns, subsequent calls to {@link #isDone} will
     * always return {@code true}.  Subsequent calls to {@link #isCancelled}
     * will always return {@code true} if this method returned {@code true}.
     */
    // 取消，请记住上面的话：
    //     在这个方法返回之后，后面的isDone()方法总是返回true。
    //     如果这个方法返回true，后面的isCancelled()方法总是返回true。
    boolean cancel(boolean mayInterruptIfRunning);
}
```

Netty 的实现虽然没有太大问题，但明显违反了 Java 原来这个方法的语义。针对这个问题，Netty 官方也给出了回应：



根据 Norman 的回复，在下个大版本中，Netty 的 Future 接口将不再继承自 Java 原生的 Future 接口，但是，会提供一些转换的方法。

好了，这只是一个小插曲，下面我们从源码层面分析一个 Future 家族的经典实现 ——DefaultPromise。

DefaultPromise

主要属性

关于源码解析，我们首先来看一下 DefaultPromise 的属性：

```java
public class DefaultPromise<V> extends AbstractFuture<V> implements Promise<V> {
    // ...省略常量
    // 任务执行的结果
    private volatile Object result;
    // 执行的线程
    private final EventExecutor executor;
    // 监听器，用于执行回调
    private Object listeners;
    // 等待者的数量，即阻塞的线程数量
    private short waiters;
    // 用于控制通知监听，因为是在synchronized内部使用，所以不用加volatile关键字
    private boolean notifyingListeners;
}
```

可以看到，与 Java 原生 FutureTask 不一样的地方主要有以下几点：

1. 没有 state 变量，只有一个 volatile 修饰的 result 用于存放结果，那么，它是怎么维护这些状态的呢？
2. 没有了阻塞线程组成的队列，只存储了阻塞线程的数量，那么，它是怎么通知等待线程的呢？
3. 可以绑定了一系列的监听器，但是，它却是个 Object 类型，是不是有点奇怪？
4. 绑定了一个 EventExecutor，它是在什么时候使用的呢？

让我们带着这些问题，继续前进。

调试用例

其实，DefaultPromise 的调试用例还是比较难写的，需要有前面服务启动和数据流向的源码剖析的底子，然后，从中找出一些端倪，才能比较容易地写出来，下面是我写的调试用例：

```
public class DefaultPromiseTest {

    public static void main(String[] args) throws InterruptedException, IOException {
        // 相当于一个线程池中的一个线程（单线程）
        EventExecutor eventExecutor = new DefaultEventExecutor();
        // 使用EventExecutor的newPromise()方法创建
        Promise<Integer> promise1 = eventExecutor.newPromise();
        // 使用构造方法创建，推荐使用上面那种方式创建Promise
        Promise<Integer> promise2 = new DefaultPromise<>(eventExecutor);

        GenericFutureListener<Future<? super Integer>> listener = future -> {
            // 成功执行完毕
            if (future.isSuccess()) {
                System.out.println("done success, result=" + future.get()+", thread=" + Thread.currentThread().getName());
            } else {
                // 失败了
                System.out.println("done exception, e=" + future.cause()+", thread=" + Thread.currentThread().getName());
            }
        };
        promise1.addListener(listener);
        promise2.addListener(listener);

        // 执行两个任务
        eventExecutor.execute(() -> calc(2, promise1));
        eventExecutor.execute(() -> calc(0, promise2));

        // sync，阻塞，如果有异常会抛出异常
        promise1.sync();
        // await，阻塞，不会抛出异常
        promise2.await();

        System.out.println("finish");

        System.in.read();
    }

    private static void calc(int num, Promise<Integer> promise) {
        try {
            // 阻塞1秒
            LockSupport.parkNanos(TimeUnit.SECONDS.toNanos(1));
            // 注意除数
            int result = 100 / num;
            System.out.println("success, thread=" + Thread.currentThread().getName());
            // 成功执行完毕
            promise.setSuccess(result);
        } catch (Exception e) {
            System.out.println("exception caught, e=" + e + ", thread=" + Thread.currentThread().getName());
            // 执行异常
            promise.setFailure(e);
        }
    }
}
```

首先，我们要明确一个概念，不管是 Future 还是 Promise，它们本身什么都代表不了，它们只是用来存储任务运行的结果（正常结果或者异常结果），所以，它们必须和任务一起使用才有意义，至于怎么一起使用，主要有两种形式，一种是同时实现 Future 接口（或 Promise 接口）和 Runnabe 接口，这样的实现类一般都是以 `XxxTask` 命名，比如 `FutureTask`、`PromiseTask`，一种是不实现 Runnable 接口，这样的类一般不具有 run () 方法，必须以参数的形式传递到任务的 run () 方法里面，这样在任务执行完毕才能把结果赋值给它。

可以看到，在我们的调试用例中，DefaultPromise 没有实现 Runnable 接口，所以，它只能以参数的形式传递给任务的 run () 方法，在任务执行完毕把结果再赋值给它。而任务一般在哪里运行呢？任务必须在线程池中运行，更确切地说，是需要有一个线程来执行任务，所以，这里我们需要创建一个线程，而在 Netty 中，线程又是以 EventExecutor 的形式存在的（可以把 EventExecutor 理解为 Java 原生线程池中的 Worker），因此，我们这里使用了一个 EventExecutor 的子类来作为线程使用。另外，查看 DefaultPromise 的构造方法，也可以知道，必须传一个 EventExecutor 类型的参数。

OK，在这个调试用例中，我们先创建了一个 DefaultEventExecutor 来作为任务执行的线程使用，随后使用不同的方式创建了两个 Promise，然后，创建了一个监听器并绑定到两个 Promise 中，接着使用上面创建的 EventExecutor 分别执行两个任务，最后，使用两种不同的方式阻塞等待任务执行完毕。在这个用例中，两个任务完全执行完毕需要多长时间？1 秒吗？2 秒吗？答案是 2 秒，因为这两个任务是放到同一个线程中来执行的，变成了串行，每个任务阻塞 1 秒，所以，一共是 2 秒。

好了，调试用例解释完毕，下面就来深入部署 DefaultPromise 的源码。

源码剖析

前面创建 DefaultEventExecutor 和 DefaultPromise 的部分直接跳过，到创建监听器这里。

这里使用 lambda 表达式的形式创建了一个 GenericFutureListener 的匿名类，这个类里面我们对任务执行的结果进行监听，这个结果当然就是存储在 Future 里面的，通过 Future 的 isSuccess () 方法判断任务是否执行成功，然后针对成功和失败分别打印不同的语句。

然后，把这个监听器分别添加到两个 Promise 中，跟踪到 addListener () 方法中：

```
// io.netty.util.concurrent.DefaultPromise#addListener
@Override
public Promise<V> addListener(GenericFutureListener<? extends Future<? super V>> listener) {
    checkNotNull(listener, "listener");
    // 添加监听器的时候加锁
    synchronized (this) {
        addListener0(listener);
    }
    // 如果任务已经执行完毕，直接触发监听器
    if (isDone()) {
        notifyListeners();
    }

    return this;
}

private void addListener0(GenericFutureListener<? extends Future<? super V>> listener) {
    // listeners即我们前面说的那个Object对象
    if (listeners == null) {
        // 1. 如果为空，则把listener直接赋值给它
        listeners = listener;
    } else if (listeners instanceof DefaultFutureListeners) {
        // 3. 如果是DefaultFutureListeners，则调用其add()方法
        ((DefaultFutureListeners) listeners).add(listener);
    } else {
        // 2. 否则创建一个DefaultFutureListeners把两个listener都添加进去
        listeners = new DefaultFutureListeners((GenericFutureListener<?>) listeners, listener);
    }
}
```

可以看到，对于 `private Object listeners;`，这个 listeners 可能有两种类型，一种是 GenericFutureListener，一种是 DefaultFutureListeners。试想一下，连续添加多个 listener 时的逻辑：

- 添加第一个 listener 时，listeners 为空，直接把这个 listener 赋值给 listeners；
- 添加第二个 listener 时，listeners 不为空，且不为 DefaultFutureListeners 类型，创建一个 DefaultFutureListeners 对象，并把当前添加这个和上一个添加的 listener 都通过构造方法传递给 DefaultFutureListeners；
- 添加第三个 listener 时，listeners 不为空，且为 DefaultFutureListeners 类型，直接调用 DefaultFutureListeners 的 add () 方法把 listener 添加进去。

查看 DefaultFutureListeners 的内容，可以发现，DefaultFutureListeners 里面维护一了个 GenericFutureListener 数组，那么，Netty 为什么要如此不嫌麻烦地这么玩呢？直接把这个数组放到 DefaultPromise 中，它不香么？

我猜测 Netty 这么玩的主要原因是，如果把数组直接放到 DefaultPromise 中，则针对这个数组的添加元素、删除元素、扩容的方法都要写在 DefaultPromise 中，DefaultPromise 类就变得不再单纯了，所以，另外拿一个类维护这些信息会比较好一些。至于添加第一个元素的时候为什么不使用 DefaultFutureListeners，可能是 DefaultFutureListeners 比较重（相当于一个 ArrayList），而且大部场景下可能都只有一个 Listener，所以针对一个 Listener 的时候单独处理一下可能要好一些，就像 HashMap 中针对 Key 为 String 类型时会单独处理一样。

好了，经历过两次 addListener ()，我们创建的 Listener 已经成功添加到两个 Promise 中了，下面就是执行任务 `eventExecutor.execute(() -> calc(2, promise1));` 了，这里就不用跟踪进去了，直接在 calc () 方法中打一个断点，按 F9，程序自然会走到 calc () 方法中：

```
private static void calc(int num, Promise<Integer> promise) {
    try {
        // 阻塞1秒
        LockSupport.parkNanos(TimeUnit.SECONDS.toNanos(1));
        // 注意除数
        int result = 100 / num;
        System.out.println("success, thread=" + Thread.currentThread().getName());
        // 成功执行完毕
        promise.setSuccess(result);
    } catch (Exception e) {
        System.out.println("exception caught, e=" + e + ", thread=" + Thread.currentThread().getName());
        // 执行异常
        promise.setFailure(e);
    }
}
```

在 calc () 方法中，我们主要做了一个除法运算，除数是传进来的参数 num，大家应该知道，如果 num 为 0 的话，这里肯定会抛出异常，所以，这里我使用一个 try…catch… 分别处理成功和失败的结果，对于成功的结果，调用 Promise 的 setSuccess () 方法将结果设置进去，对于失败的结果，那就是异常喽，使用 Promise 的 setFailure () 将异常信息设置进去。

继续跟进到 setSuccess () 方法内部：

```
@Override
public Promise<V> setSuccess(V result) {
    if (setSuccess0(result)) {
        return this;
    }
    throw new IllegalStateException("complete already: " + this);
}
private boolean setSuccess0(V result) {
    return setValue0(result == null ? SUCCESS : result);
}
private boolean setValue0(Object objResult) {
    if (RESULT_UPDATER.compareAndSet(this, null, objResult) ||
        RESULT_UPDATER.compareAndSet(this, UNCANCELLABLE, objResult)) {
        if (checkNotifyWaiters()) {
            notifyListeners();
        }
        return true;
    }
    return false;
}
```

这里就比较简单了，不过知识点还是挺多的。首先，通过 CAS 的方式设置 result 的值，然后检查有没有等待者，有的话就通知它们，最后检查有没有 listeners，有的话就通知它们。

这里的 CAS 使用了一种特殊的技巧，我们可以看下 `RESULT_UPDATER` 的定义：

```
private static final AtomicReferenceFieldUpdater<DefaultPromise, Object> RESULT_UPDATER =
        AtomicReferenceFieldUpdater.newUpdater(DefaultPromise.class, Object.class, "result");
private volatile Object result;
```

上面这两条语句在运行结果上等于下面这一条语句：

```
private AtomicReference<Object> resultReference = new AtomicReference<>();
```

但是上面的写法更优，原因在于上面的语句使用 一个静态变量 + 一个非静态变量 的形式更节约内存，我们举个例子，假如创建 10 个 DefaultPromise，使用上面的语句内存中一共有 一个静态变量 + 10 个 result 变量，而如果使用下面的语句内存中会存储 10 个 resultReference 变量，同时，每个 resultReference 变量内部还维护了一个 Object 变量，所以一共是 20 个变量，下面的语句更耗内存，这种用法在 Netty 中随处可见。

然后，就是通知等待者：

```
private synchronized boolean checkNotifyWaiters() {
    if (waiters > 0) {
        notifyAll();
    }
    return listeners != null;
}
```

这里比较恶心的一点是，明明 waiters 和 listeners 没有半毛线关系，但是，却把它们俩的判断放在同一个方法中，而把监听器的通知放在了这个方法之外，这样写的目的是为了共用同一个 `synchronized`。这里通知等待者使用的就是 Object 对象的 notifyAll () 方法，它只能使用在 `synchronized` 关键字内部，没错，Netty 对于等待者的处理就是使用的 Java 原生的 `synchronized`，并没有使用其它任何的技巧，简单唯美。

最后，就是通知监听器，这里有点小复杂：

```
private void notifyListeners() {
    EventExecutor executor = executor();
    // 判断运行任务的线程跟执行监听回调的线程是不是同一个
    if (executor.inEventLoop()) {
        // 这里也是有点迷的地方，为什么要把stackDepth直接放到FastThreadLocal的InternalThreadLocalMap类中？
        // 这种耦合有点过分了
        final InternalThreadLocalMap threadLocals = InternalThreadLocalMap.get();
        final int stackDepth = threadLocals.futureListenerStackDepth();
        if (stackDepth < MAX_LISTENER_STACK_DEPTH) {
            threadLocals.setFutureListenerStackDepth(stackDepth + 1);
            try {
                // 立即通知监听器
                notifyListenersNow();
            } finally {
                threadLocals.setFutureListenerStackDepth(stackDepth);
            }
            // 注意，这里有个return
            return;
        }
    }
    // 如果不是同一个线程，走这个逻辑，使用监听器线程执行回调
    safeExecute(executor, new Runnable() {
        @Override
        public void run() {
            notifyListenersNow();
        }
    });
}
private void notifyListenersNow() {
    Object listeners;
    // 加锁，拿到listeners，赋值给局部变量，并清空之
    synchronized (this) {
        if (notifyingListeners || this.listeners == null) {
            return;
        }
        // 设置标志，说明正在进行回调的执行
        notifyingListeners = true;
        listeners = this.listeners;
        this.listeners = null;
    }
    for (;;) {
        // 执行回调
        if (listeners instanceof DefaultFutureListeners) {
            notifyListeners0((DefaultFutureListeners) listeners);
        } else {
            notifyListener0(this, (GenericFutureListener<?>) listeners);
        }
        // 加锁，做了两件事：
        // 如果执行回调期间添加了新的监听器，则赋值给局部变量，再次执行回调
        // 如果没有，则退出for循环并重置标志位
        synchronized (this) {
            if (this.listeners == null) {
                notifyingListeners = false;
                return;
            }
            listeners = this.listeners;
            this.listeners = null;
        }
    }
}
```

这里使用了锁拆分的技术，将锁拆分为两段，并把比较耗时的回调执行的过程剥离出去，能很大程度上提高效率。

真正执行回调这里就比较简单了，直接调用我们上面定义的 lambda 表达式：

```
// 有多个监听器
private void notifyListeners0(DefaultFutureListeners listeners) {
    GenericFutureListener<?>[] a = listeners.listeners();
    int size = listeners.size();
    for (int i = 0; i < size; i ++) {
        notifyListener0(this, a[i]);
    }
}
// 单个监听器
@SuppressWarnings({ "unchecked", "rawtypes" })
private static void notifyListener0(Future future, GenericFutureListener l) {
    try {
        l.operationComplete(future);
    } catch (Throwable t) {
        if (logger.isWarnEnabled()) {
            logger.warn("An exception was thrown by " + l.getClass().getName() + ".operationComplete()", t);
        }
    }
}
```

好了，最后，还有一个 await () 方法：

```
@Override
public Promise<V> await() throws InterruptedException {
    // 如果已经完成了，直接返回
    if (isDone()) {
        return this;
    }
    // 线程中断了，抛出异常
    if (Thread.interrupted()) {
        throw new InterruptedException(toString());
    }
    // 检查死锁，防止等待的方法在EventLoop中执行
    checkDeadLock();
    // 加锁
    synchronized (this) {
        while (!isDone()) {
            // 等待者数量加1
            incWaiters();
            try {
                // 等待，调用的是Object的wait()方法
                wait();
            } finally {
                // 等待者数量减1
                decWaiters();
            }
        }
    }
    return this;
}
```

await () 方法中使用的是就是 Java 原生关键字 `synchronized` 锁 + wait () 方法实现的，没有使用其它任何的技巧，简单唯美。

好了，到这里，DefaultPromise 的源码就分析完毕了，相比于 Java 原生的 FutureTask，DefaultPromise 的实现主要使用了大量的 synchronized 关键字锁来实现主要逻辑流的控制，虽然简单，但还是有很多知识点值得我们学习的，比如锁拆分、CAS 的使用，等等。

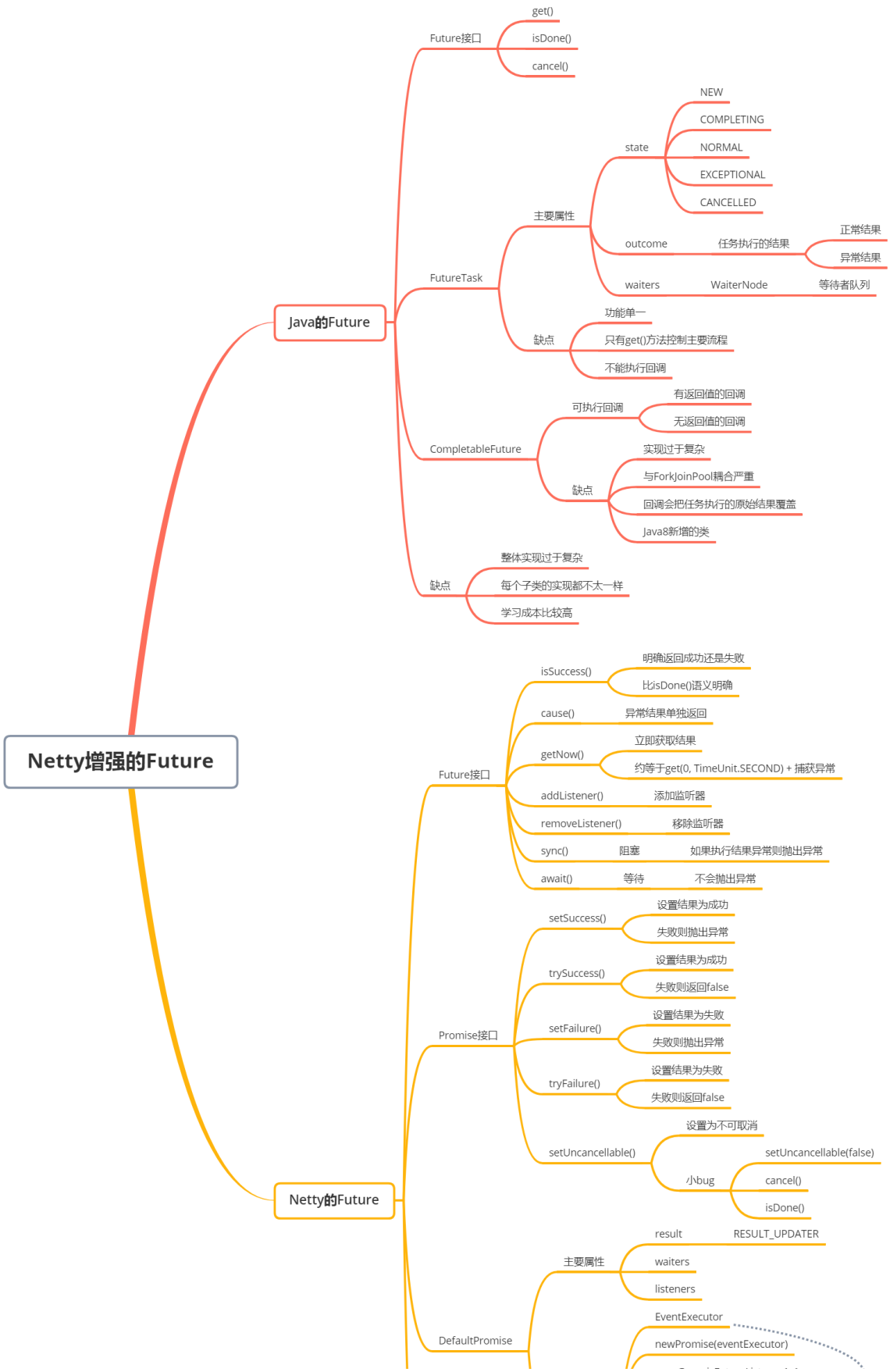
后记

本节，我们一起学习了 Netty 中增强的 Future，并对其主要实现类 ——DefaultPromise—— 进行了深入剖析。

看了本节的内容，你可能还是有点懵，这玩意在 Netty 中到底该如何使用呢？别急，下一节我们还会见到它的身影。

下一节，我们将对 Netty 中的线程池做一个深入的剖析，你准备好了吗？

思维导图



}

24 Netty的队列有何不一样

26 Netty的线程池有什么样的特性