



Netty 中的异步编程 Future 和 Promise

Netty 中大量 I/O 操作都是异步执行，本篇博文来聊聊 Netty 中的异步编程。

Java Future 提供的异步模型

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，可以用来

对于异步编程，我们想要的实现是：提交一个任务，在任务执行期间提交者可以做别的事情，这个任务是在异步执行的，当任务执行完毕时，提交者任务完成获取结果。

那么在 Future 中是怎么实现的呢？我们先看接口定义：

CONTENTS	
">	0.0.1. Java Future 提...
">	0.0.2. CompletableF...
">	0.0.3. Netty 中的异...

```
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;  
}
```

Copy

我们看一个示例：

```
public class FutureTest {  
  
    public static void main(String[] args) {  
        ExecutorService executorService = Executors.newFixedThreadPool(2);  
        System.out.println("start");  
        Future<Integer> submit = executorService.submit(() -> {  
            try {  
                Thread.sleep(3000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            return 1;  
        });  
        Integer value = null;  
        try {  
            value = submit.get();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        System.out.println(value);  
        System.out.println("end");  
    }  
}
```

Copy

Future 的使用方式是：投递一个任务到 Future 中执行，操作完之后调用 `Future#get()` 或者 `Future#isDone()` 方法判断是否执行完毕。从这个逻辑上看，Future 提供的功能是：用户线程需要主动轮询 Future 线程是否完成当前任务，如果不通过轮询是否完成而是同步等待获取，则会阻塞直到执行完毕为止。所以从这里看，Future 并不是真正的异步，因为它少了一个 **回调**，充其量只能算是一个同步非阻塞模式。

`executorService.submit()` 方法获取带返回值的 Future 结果有两种方式：

1. 一种是通过实现 `Callable` 接口；
2. 第二种是中间变量返回。继承 Future 的子类：FutureTask，通过 FutureTask 返回异步结果而不是在主线程中获取（FutureTask 是使用 `Callable` 进行创建）。

面两种方式的代码就变为这样:

```
public class FutureTest {
```

Copy

```
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);
        System.out.println("start");
        //方式1 通过 executorService 提交一个异步线程
        //Future<Integer> submit = executorService.submit(new NewCallableTask());

        //方式2 通过 FutureTask 包装异步线程的返回, 返回结果在 FutureTask 中获取而不是 在提交线程中
        FutureTask<Integer> task = new FutureTask<>(new NewCallableTask());
        executorService.submit(task);
        //-----方式2-----

        Integer value = null;
        try {
            value = task.get();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(value);
        System.out.println("end");
    }

    /**
     * 通过实现 Callable 接口
     */
    static class NewCallableTask implements Callable<Integer> {

        @Override
        public Integer call() throws Exception {
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return 1;
        }
    }
}
```

≡ CONTENTS

×

"> 0.0.1. Java Future 提...
"> 0.0.2. CompletableF...
"> 0.0.3. Netty 中的异...

一般在使用线程池创建线程执行任务的时候会有两种方式, 要么实现 Runnable 接口, 要么实现 Callable 接口, 它们的区别在于:

1. Callable 可以在任务结束的时候提供一个返回值, Runnable 无法提供这个功能;
2. Callable 的 call 方法可以抛出异常, 而 Runnable 的 run 方法不能抛出异常。

而我们的异步返回自然是使用 Callable 方式。那么 Callable 是如何实现的呢?

从 Callable 被提交的地方入手: `executorService.submit(task)`, ExecutorService 是一个接口, 他的默认实现类是: AbstractExecutorService, 我们看这里的 `submit()` 实现方式:

```
public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
```

Copy

可以看到将 Callable 又包装成了 RunnableFuture。而这个 `RunnableFuture` 就比较神奇, 它同时继承了 Runnable 和 Future, 既有线程的能力又有可携带返回值的功能。

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    /**
```

Copy



```
* Sets this Future to the result of its computation
* unless it has been cancelled.
*/
void run();
}
```

所以再看 `submit()` 方法，其实是将 `RunnableFuture` 线程送入线程池执行，执行是一个新线程，只是这个执行的对象是 `FutureTask`，所以 `submit()` 方法返回的是 `FutureTask` 对象，而不是 `Future` 对象。取执行结果。

那么 `Callable` 优势如何变为 `RunnableFuture` 的呢？我们看 `newTaskFor(task)` 方法：

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
    return new FutureTask<T>(callable);
}
```

将 `Callable` 包装为 `FutureTask` 对象，看到这里又关联到 `FutureTask`，：

```
public class FutureTask<V> implements RunnableFuture<V> {
    ...
}
```

可以看到 `FutureTask` 是 `RunnableFuture` 的子类，这也就解释了上面的示例为什么在线程池中可以提交 `FutureTask` 实例。

更详细的执行过程这里就不再分析，重点剖析 `Future` 的实现过程，它并不是真正的异步，没有实现回调。所以在 Java8 中又新增了一个真正的异步函数：`CompletableFuture`。

CompletableFuture 非阻塞异步编程模型

Java 8 中新增加了一个类：`CompletableFuture`，它提供了非常强大的 `Future` 的扩展功能，最重要的是实现了回调的功能。

使用示例：

```
public class CallableFutureTest {

    public static void main(String[] args) {
        System.out.println("start");
        /**
         * 异步非阻塞
         */
        CompletableFuture.runAsync(() -> {
            try {
                Thread.sleep(3000);
                System.out.println("sleep done");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("done");
    }
}
```

`CompletableFuture.runAsync()` 方法提供了异步执行无返回值任务的功能。

```
ExecutorService executorService = Executors.newFixedThreadPool(100);

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // do something
})
```

CONTENTS

- "> 0.0.1. Java Future 提...
- "> 0.0.2. CompletableFuture...
- "> 0.0.3. Netty 中的异...

Copy

Copy

Copy

```
return "result";
}, executorService);
```

`CompletableFuture.supplyAsync()` 方法提供了异步执行有返回值任务的功能。

`CompletableFuture`源码中有四个静态方法用来执行异步任务：

```
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier){..}

public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier,Executor executor){..}

public static CompletableFuture<Void> runAsync(Runnable runnable){..}

public static CompletableFuture<Void> runAsync(Runnable runnable,
Executor executor){..}
```

CONTENTS	
">	0.0.1. Java Future 提...
">	0.0.2. CompletableFuture...
">	0.0.3. Netty 中的异...

前面两个可以看到是带返回值的方法，后面两个是不带返回值的方法。同时支持传入自定义的线程池，如果不传入线程池的话默认是使用 `ForkJoinPool.commonPool()` 作为它的线程池执行异步代码。

合并两个异步任务

如果有两个任务需要异步执行，且后面需要对这两个任务的结果进行合并处理，`CompletableFuture` 也支持这种处理：

```
ExecutorService executorService = Executors.newFixedThreadPool(100);

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {
    return "Task1";
}, executorService);
CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {
    return "Task2";
}, executorService);
CompletableFuture<String> future = future1.thenCombineAsync(future2, (task1, task2) -> {
    return task1 + task2; // return "Task1Task2" String
});
```

Copy

通过 `CompletableFuture.thenCombineAsync()` 方法获取两个任务的结果然后进行相应的操作。

下一个依赖上一个的结果

如果第二个任务依赖第一个任务的结果：

```
ExecutorService executorService = Executors.newFixedThreadPool(100);

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {
    return "Task1";
}, executorService);
CompletableFuture<String> future = future1.thenComposeAsync(task1 -> {
    return CompletableFuture.supplyAsync(() -> {
        return task1 + "Task2"; // return "Task1Task2" String
    });
}, executorService);
```

Copy

`CompletableFuture.thenComposeAsync()` 支持将第一个任务的结果传入第二个任务中。

常用 API 介绍

1. 拿到上一个任务的结果做后续操作，上一个任务完成后的动作

```
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)
```

Copy

上面四个方法表示在当前阶段任务完成之后下一步要做什么。`whenComplete` 表示在当前线程内继续做下一步，带 `Async` 后缀的表示使用新线程去执行。

2. 拿到上一个任务的结果做后续操作，使用 `handler` 来处理逻辑，可以返回与第一阶段处理的返回类型不一样的返回类型。



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

[Copy](#)

Handler 与 whenComplete 的区别是 handler 是可以返回一个新的 CompletableFuture 类型的。

```
CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(() -> {
    return "hahaha";
}).handle((r, e) -> {
    return 1;
});
```

[Copy](#)

3. 拿到上一个任务的结果做后续操作， thenApply方法

```
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)
```

注意到 thenApply 方法的参数中是没有 Throwable，这就意味着如有异常就会立即失败，不能在处理逻辑内处理。且 thenApply 返回的也是新的 CompletableFuture。这就是它与前面两个的区别。

4. 拿到上一个任务的结果做后续操作，可以不返回任何值， thenAccept方法

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

[Copy](#)

看这里的示例：

```
CompletableFuture.supplyAsync(() -> {
    return "result";
}).thenAccept(r -> {
    System.out.println(r);
}).thenAccept(r -> {
    System.out.println(r);
});
```

[Copy](#)

执行完毕是不会返回任何值的。

CompletableFuture 的特性体现在执行完 runAsync 或者 supplyAsync 之后的操作上。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。

另外当你依赖 CompletableFuture 的计算结果才能进行下一步的时候，无需手动判断当前计算是否完成，可以通过 CompletableFuture 的事件监听自动去完成。

Netty 中的异步编程

说 Netty 中的异步编程之前先说一个异步编程模型：Future/Promise异步模型。

future和promise起源于函数式编程和相关范例（如逻辑编程），目的是将值（future）与其计算方式（promise）分离，从而允许更灵活地进行计算，特别是通过并行化。

Future 表示目标计算的返回值，Promise 表示计算的方式，这个模型将返回结果和计算逻辑分离，目的是为了让计算逻辑不影响返回结果，从而抽象出一套异步编程模型。那计算逻辑如何与结果关联呢？它们之间的纽带就是 callback。

引用自：<https://zh.wikipedia.org/wiki/Future%E4%B8%8Epromise>

在 Netty 中的异步编程就是基于该模型来实现。Netty 中非常多的异步调用，最简单的例子就是我们 Server 和 Client 端启动的例子：

Server：

```
public void start(){
    EventLoopGroup bossGroup = new NioEventLoopGroup();
    EventLoopGroup workGroup = new NioEventLoopGroup();

    ServerBootstrap server = new ServerBootstrap().group(bossGroup,workGroup)
        .channel(NioServerSocketChannel.class)
        .childHandler(new ServerChannelInitializer());

    try {
        ChannelFuture future = server.bind(port).sync();
        future.channel().closeFuture().sync();
    } catch (InterruptedException e) {
        log.error("server start fail",e);
    } finally {
        bossGroup.shutdownGracefully();
        workGroup.shutdownGracefully();
    }
}
```



ent:

```
public void start(){
    EventLoopGroup group = new NioEventLoopGroup();

    Bootstrap bootstrap = new Bootstrap();
    bootstrap.group(group)
        .channel(NioSocketChannel.class)
        .handler(new ClientChannelInitializer());

    try {
        ChannelFuture future = bootstrap.connect(address,port).sync();
        future.channel().writeAndFlush(msg: "Hello world, i'm online");
        future.channel().closeFuture().sync();
    } catch (Exception e) {
        log.error("client start fail",e);
    } finally {
        group.shutdownGracefully();
    }
}
```

CONTENTS	
">	0.0.1. Java Future 提...
">	0.0.2. CompletableF...
">	0.0.3. Netty 中的异...

Netty 中使用了一个 ChannelFuture 来实现异步操作，看似与 Java 中的 Future 相似，我们看一下代码：

```
public interface ChannelFuture extends Future<Void> {
}
```

Copy

这里 ChannelFuture 继承了一个 Future，这是 Java 中的 Future 吗？跟下去发现并不是 JDK 的，而是 Netty 自己实现的。该类位于：

io.netty.util.concurrent 包中：

```
public interface Future<V> extends java.util.concurrent.Future<V> {

    // 只有IO操作完成时才返回true
    boolean isSuccess();
    // 只有当cancel(boolean)成功取消时才返回true
    boolean isCancellable();
    // IO操作发生异常时，返回导致IO操作以此的原因，如果没有异常，返回null
    Throwable cause();
    // 向Future添加事件，future完成时，会执行这些事件，如果add时future已经完成，会立即执行监听事件
    Future<V> addListener(GenericFutureListener<? extends Future<? super V>> listener);
    Future<V> addListeners(GenericFutureListener<? extends Future<? super V>>... listeners);
    // 移除监听事件，future完成时，不会触发
    Future<V> removeListener(GenericFutureListener<? extends Future<? super V>> listener);
    Future<V> removeListeners(GenericFutureListener<? extends Future<? super V>>... listeners);
    // 等待future done
    Future<V> sync() throws InterruptedException;
    // 等待future done，不可打断
    Future<V> syncUninterruptibly();
    // 等待future完成
    Future<V> await() throws InterruptedException;
    // 等待future 完成，不可打断
    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);
    // 立刻获得结果，如果没有完成，返回null
    V getNow();
    // 如果成功取消，future会失败，导致CancellationException
    @Override
    boolean cancel(boolean mayInterruptIfRunning);
}
```

Copy



Netty 自己实现的 Future 继承了 JDK 的 Future，新增了 `sync()` 和 `await()` 用于阻塞等待，还加了 Listeners，只要任务结束去回调 Listener 就可以了，那么我们就不要主动调用 `isDone()` 来获取状态，或通过 `get()` 阻塞方法来获取值。

Netty 的 Future 与 Java 的 Future 虽然类名相同，但功能上略有不同，Netty 中引入了 Promise 机制。在 Java 的 Callable 或 Runnable 实现类，该类的 `call()` 或 `run()` 执行完毕意味着业务逻辑的完结，在 Promise 机制中，置业务逻辑的成功与失败，这样更加方便的监控自己的业务逻辑。

CONTENTS		×
">	0.0.1. Java Future 提...	
">	0.0.2. CompletableF...	
">	0.0.3. Netty 中的异...	

```
public interface Promise<V> extends Future<V> {
    // 设置future执行结果为成功
    Promise<V> setSuccess(V result);

    // 尝试设置future执行结果为成功,返回是否设置成功
    boolean trySuccess(V result);
    // 设置失败
    Promise<V> setFailure(Throwable cause);
    // 尝试设置future执行结果为失败,返回是否设置成功
    boolean tryFailure(Throwable cause);
    // 设置为不能取消
    boolean setUncancellable();

    // 源码中，以下为覆盖了Future的方法，例如：

    Future<V> addListener(GenericFutureListener<? extends Future<? super V>> listener);

    @Override
    Promise<V> addListener(GenericFutureListener<? extends Future<? super V>> listener);
}
```

Promise 接口继承自 Future 接口，重点添加了上述几个方法，可以人工设置 future 的执行成功与失败，并通知所有监听的 listener。

从 Future 和 Promise 提供的方法来看，Future 都是 get 类型的方法，主要用来判断当前任务的状态。而 Promise 中是 set 类型的方法，主要来对任务的状态来进行操作。这里就体现出来将 结果和操作过程分离的设计。

Promise 实现类是DefaultPromise类，该类十分重要，Future 的 listener 机制也是由它实现的，所以我们先来分析一下该类。先来看一下它的重要属性：

```
// 可以嵌套的Listener的最大层数，可见最大值为8
private static final int MAX_LISTENER_STACK_DEPTH = Math.min(8,
    SystemPropertyUtil.getInt("io.netty.defaultPromise

// result字段由使用RESULT_UPDATER更新
@SuppressWarnings("rawtypes")
private static final AtomicReferenceFieldUpdater<DefaultPromise, Object> RESULT_UPDATER;
private static final Signal SUCCESS = Signal.valueOf(DefaultPromise.class, "SUCCESS");
// 异步操作不可取消
private static final Signal UNCANCELLABLE = Signal.valueOf(DefaultPromise.class, "UNCANCELLABLE");
// 异步操作失败时保存异常原因
private static final CauseHolder CANCELLATION_CAUSE_HOLDER = new CauseHolder(ThrowableUtil.unknownStackTrace(
    new CancellationException(), DefaultPromise.class, "cancel(...)));
```

Copy

第一个套 listener，是指在 listener 的 `operationComplete()` 方法中，可以再次使用 `future.addListener()` 继续添加 listener，Netty 限制的最大层数是8，用户可使用系统变量 `io.netty.defaultPromise.maxListenerStackDepth` 设置。

为了更好的说明，先写了一个示例，Netty 中的 Future/Promise模型是可以单独拿出来使用的。

```
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.util.concurrent.DefaultPromise;
import io.netty.util.concurrent.Promise;

import java.util.concurrent.TimeUnit;

/**
 * @author rickiyang
```

Copy



```
* @date 2020-04-19
* @Desc TODO
*/
```

```
public class PromiseTest {

    public static void main(String[] args) {
        PromiseTest testPromise = new PromiseTest();
        Promise<String> promise = testPromise.doSomething("哈哈");
        promise.addListener(future -> System.out.println(promise.get()+" , something is done");

    }

    /**
     * 创建一个DefaultPromise并返回，将业务逻辑放入线程池中执行
     * @param value
     * @return
     */
    private Promise<String> doSomething(String value) {
        NioEventLoopGroup loop = new NioEventLoopGroup();
        DefaultPromise<String> promise = new DefaultPromise<>(loop.next());
        loop.schedule(() -> {
            try {
                Thread.sleep(1000);
                promise.setSuccess("执行成功。" + value);
                return promise;
            } catch (InterruptedException ignored) {
                promise.setFailure(ignored);
            }
            return promise;
        }, 0, TimeUnit.SECONDS);
        return promise;
    }

}
```

CONTENTS		×
">	0.0.1. Java Future 提...	
">	0.0.2. CompletableF...	
">	0.0.3. Netty 中的异...	

通过这个例子可以看到，Promise 能够在业务逻辑线程中通知 Future 成功或失败，由于 Promise 继承了 Netty 的 Future，因此可以加入监听事件。而 Future 和 Promise 的好处在于，获取到 Promise 对象后可以为其设置异步调用完成后的操作，然后立即继续去做其他任务。

来看一下 addListener() 方法：

```
@Override
public Promise<V> addListener(GenericFutureListener<? extends Future<? super V>> listener) {
    checkNotNull(listener, "listener");
    //并发控制，保证多线程情况下只有一个线程执行添加操作
    synchronized (this) {
        addListener0(listener);
    }
    // 操作完成，通知监听者
    if (isDone()) {
        notifyListeners();
    }

    return this;
}

private void addListener0(GenericFutureListener<? extends Future<? super V>> listener) {
    if (listeners == null) {
        listeners = listener;
    } else if (listeners instanceof DefaultFutureListeners) {
        // 如果当前Promise实例持有listeners的是DefaultFutureListeners类型，则调用它的add()方法进行添加
        ((DefaultFutureListeners) listeners).add(listener);
    } else {
        // 步入这里说明当前Promise实例持有listeners为单个GenericFutureListener实例，需要转换为DefaultFutureListeners实例
        listeners = new DefaultFutureListeners((GenericFutureListener<? extends Future<V>>) listeners, listener);
    }
}
```

Copy



这里看到有一个全局变量 `listeners`，我们看到他的定义：

```
private Object listeners;
```

为啥会是一个 `Object` 类型的对象呢，不是应该是 `List` 或者是数组才对嘛。Netty之所以这样设计，是因为大多数情况下用集合和数组都会造成浪费。当只有一个 `listener` 时，该字段为一个 `GenericFutureListener` 对象；当多于一个 `listener` 时，为 `DefaultFutureListeners`，可以储存多个 `listener`。

我们再来看 `notifyListeners()` 方法：

```
private void notifyListeners() {
    EventExecutor executor = executor();
    //当前EventLoop线程需要检查listener嵌套
    if (executor.inEventLoop()) {
        final InternalThreadLocalMap threadLocals = InternalThreadLocalMap.get();
        //这里是当前listener的嵌套层数
        final int stackDepth = threadLocals.futureListenerStackDepth();
        if (stackDepth < MAX_LISTENER_STACK_DEPTH) {
            threadLocals.setFutureListenerStackDepth(stackDepth + 1);
            try {
                notifyListenersNow();
            } finally {
                threadLocals.setFutureListenerStackDepth(stackDepth);
            }
            return;
        }
    }
    //外部线程直接提交给新线程执行
    safeExecute(executor, new Runnable() {
        @Override
        public void run() {
            notifyListenersNow();
        }
    });
}
```

这里有个疑问就是为什么要设置当前的调用栈深度+1。

接着看真正执行通知的方法：

```
private void notifyListenersNow() {
    Object listeners;
    synchronized (this) {
        // 正在通知或已没有监听者（外部线程删除）直接返回
        if (notifyingListeners || this.listeners == null) {
            return;
        }
        notifyingListeners = true;
        listeners = this.listeners;
        this.listeners = null;
    }
    for (;;) {
        //只有一个listener
        if (listeners instanceof DefaultFutureListeners) {
            notifyListener0((DefaultFutureListeners) listeners);
        } else {
            //有多个listener
            notifyListener0(this, (GenericFutureListener<? extends Future<V>>) listeners);
        }
    }
    synchronized (this) {
        if (this.listeners == null) {
            // 执行完毕且外部线程没有再添加监听者
            notifyingListeners = false;
        }
    }
}
```

CONTENTS

- "> 0.0.1. Java Future 提...
- "> 0.0.2. CompletableF...
- "> 0.0.3. Netty 中的异...

Copy

Copy



```

        return;
    }
    //外部线程添加了新的监听者继续执行
    listeners = this.listeners;
    this.listeners = null;
}
}
}

```

Netty 中 DefaultPromise 是一个非常常用的类，这是 Promise 实现的基础。DefaultChannelPromise DefaultPromise channel 这个属性。

Promise 目前支持两种类型的监听器：

- GenericFutureListener：支持泛型的 Future ；
- GenericProgressiveFutureListener：它是 GenericFutureListener 的子类，支持进度表示和支持泛型的Future 监听器（有些场景需要多个步骤实现，类似于进度条那样）。

为了让 Promise 支持多个监听器，Netty 添加了一个默认修饰符修饰的 DefaultFutureListeners 类用于保存监听器实例数组：

```
final class DefaultFutureListeners {
```

Copy

```

    private GenericFutureListener<? extends Future<?>>[] listeners;
    private int size;
    private int progressiveSize; // the number of progressive listeners

```

```

// 这个构造相对特别，是为了让Promise中的listeners（Object类型）实例由单个GenericFutureListener实例转换为DefaultFutureListeners
@SuppressWarnings("unchecked")

```

```

DefaultFutureListeners(GenericFutureListener<? extends Future<?>> first, GenericFutureListener<? extends
    listeners = new GenericFutureListener[2];
    listeners[0] = first;
    listeners[1] = second;
    size = 2;
    if (first instanceof GenericProgressiveFutureListener) {
        progressiveSize ++;
    }
    if (second instanceof GenericProgressiveFutureListener) {
        progressiveSize ++;
    }
}

```

```

public void add(GenericFutureListener<? extends Future<?>> l) {
    GenericFutureListener<? extends Future<?>>[] listeners = this.listeners;
    final int size = this.size;
    // 注意这里，每次扩容数组长度是原来的2倍
    if (size == listeners.length) {
        this.listeners = listeners = Arrays.copyOf(listeners, size << 1);
    }
    // 把当前的GenericFutureListener加入数组中
    listeners[size] = l;
    // 监听器总数量加1
    this.size = size + 1;
    // 如果为GenericProgressiveFutureListener，则带进度指示的监听器总数量加1
    if (l instanceof GenericProgressiveFutureListener) {
        progressiveSize ++;
    }
}

```

```

public void remove(GenericFutureListener<? extends Future<?>> l) {
    final GenericFutureListener<? extends Future<?>>[] listeners = this.listeners;
    int size = this.size;
    for (int i = 0; i < size; i ++) {
        if (listeners[i] == l) {
            // 计算需要需要移动的监听器的下标
            int listenersToMove = size - i - 1;
            if (listenersToMove > 0) {
                // listenersToMove后面的元素全部移动到数组的前端
            }
        }
    }
}

```

CONTENTS

- "> 0.0.1. Java Future 提...
- "> 0.0.2. CompletableF...
- "> 0.0.3. Netty 中的异...



```
        System.arraycopy(listeners, i + 1, listeners, i, listenersToMove);
    }
    // 当前监听器总量的最后一个位置设置为null, 数量减1
    listeners[-- size] = null;
    this.size = size;
    // 如果监听器是GenericProgressiveFutureListener, 则带进度指示的监听器总数量减1
    if (l instanceof GenericProgressiveFutureListener) {
        progressiveSize --;
    }
    return;
}

// 返回监听器实例数组
public GenericFutureListener<? extends Future<?>>[] listeners() {
    return listeners;
}

// 返回监听器总数量
public int size() {
    return size;
}

// 返回带进度指示的监听器总数量
public int progressiveSize() {
    return progressiveSize;
}
}
```

CONTENTS		×
">	0.0.1. Java Future 提...	
">	0.0.2. CompletableF...	
">	0.0.3. Netty 中的异...	

以上就是关于 Promise 和监听器相关的实现分析，再回到之前的启动类，是不是还有一个 sync() 方法：

```
@Override
public Promise<V> sync() throws InterruptedException {
    await();
    rethrowIfFailed();
    return this;
}

public Promise<V> await() throws InterruptedException {
    // 异步操作已经完成, 直接返回
    if (isDone()) {
        return this;
    }
    if (Thread.interrupted()) {
        throw new InterruptedException(toString());
    }
    // 死锁检测
    checkDeadLock();
    // 同步使修改waiters的线程只有一个
    synchronized (this) {
        while (!isDone()) { // 等待直到异步操作完成
            incWaiters(); // ++waiters;
            try {
                wait(); // JDK方法
            } finally {
                decWaiters(); // --waiters
            }
        }
    }
    return this;
}
}
```

Copy

这里其实就是一个同步检测当前事件是否完成的过程。



上就是 Netty 中实现的 Future/Promise 异步回调机制。实现并不是很难懂，代码很值得学习。除了 Netty 中实现了 Future/Promise 模型，在 Guava 中也有相关的实现，大家日常使用可以看看习惯引用相关的包。

Guava 实现：

```
<dependency>
<groupId>com.google.guava</groupId>
<artifactId>guava</artifactId>
<version>21.0</version>
</dependency>

ListeningExecutorService service = MoreExecutors.listeningDecorator(Executors.newSingleThreadExecutor());
ListenableFuture<Integer> future = service.submit(new Callable<Integer>() {
    public Integer call() throws Exception {
        TimeUnit.SECONDS.sleep(5);
        return 100;
    }
});
Futures.addCallback(future, new FutureCallback<Integer>() {
    public void onSuccess(Integer result) {
        System.out.println("success:" + result);
    }

    public void onFailure(Throwable throwable) {
        System.out.println("fail, e = " + throwable);
    }
});

Thread.currentThread().join();
```

Copy

≡ CONTENTS ×

">

0.0.1. Java Future 提...

">

0.0.2. CompletableF...

">

0.0.3. Netty 中的异...

分类: [Netty 源码分析](#)

推荐 5

赞赏

收藏

反对 0



<< 上一篇: [Netty 中的 handler 和 ChannelPipeline 分析](#)

>> 下一篇: [Servlet 和 Servlet容器](#)

posted @ 2020-04-21 07:45

rickiyang

阅读(9237)

评论(4)

编辑

收藏

举报

登录后才能查看或发表评论，立即 [登录](#) 或者 [逛逛](#) 博客园首页

- 编辑推荐:
- 深入探讨 Function Calling: 在 Semantic Kernel 中的应用实践

· Android 启动过程 – 万字长文(Android14)

· 我对微服务架构的简单理解

· 异构数据源同步之数据同步: datax 再改造, 开始触及源码

· 性能优化陷阱之 hash 真的比 strcmp 快吗
- 阅读排行:
- 我裸辞了!!!

· 盘点下华为大佬的技术拷问下我没招架住的一些问题

· .NET开源、跨平台、使用简单的面部识别库

· 初步搭建一个自己的对象存储服务---Minio

· 微服务实践Aspire项目发布到远程k8s集群

