Java回调函数

回调函数是在调用某个函数（通常是API函数时）,将自己的函数（这个函数称为回调函数）的地址作为参数传递给那个函数，那个函数在需要的时候，利用传递的地址调用回调函数。可以利用回调函数处理消息或完成一定操作。

# 简单示例

定义三个类，分别是主函数类，callback函数的接口类，业务处理类，在业务处理类中，处理完业务后，执行一个callback函数

1）定义callback函数接口类

*public interface ComputeCallback {*

*public void onComputeEnd();*

*}*

实现如下：

*public class ComputeCallbackImpl implements ComputeCallback{*

*@Override*

*public void onComputeEnd() {*

*System.out.println("End, callback");*

*}}*

2）业务类

*public class TestCallback {*

*public void compute(int n, ComputeCallback callback) {*

*for(int i = 0 ;i < n ;i++) {*

*System.out.println(i);*

*}*

*callback.onComputeEnd();*

*}}*

3）主函数类

*public class TestMain {*

*public static void main(String[] args) {*

*new TestCallback().compute(1000, new ComputeCallbackImpl());*

*}}*

4）测试结果如下：

*1 0,1,2,3,4,5,6,7,8,9,*

*2 End, callback*

在主类中，新建业务类的时候，传递进去的第二个参数是一个实现了回调接口的类对象，在业务类中，调用这个对象的onComputeEnd方法，JVM找到这个对象的函数实现并调用，然后输出结果。

# Java多线程

创建线程的两种方式是继承Thread和实现Runnable接口，但是缺陷是在执行完任务后无法获取执行结果。但是在Java多线程时，执行的线程可能依赖其他线程的执行结果，这就需要在线程中获取另一个线程的信息，获取方式有轮询和回调，轮询效率很低，因此主要使用的是回调的方法。实现方式有三种：Callable,Future和FutureTask

## Callable和Runnable

java.lang.Runnable接口，只声明了一个run方法，实现如下：

*public interface Runnable {*

*public abstract void run();*

*}*

由于run方法返回值为void，所以在执行完任务之后无法返回任何结果。

Callable位于java.util.concurrent包下，它只声明一个方法，这个方法为call，如下所示：

*public interface Callable<V> {*

*V call() throws Exception;*

*}*

这是一个泛型接口，call方法返回的类型是传递进来的V类型。一般在ExecutorService中声明submit方法重载，通常调用的方法为：

*<T> Future<T> submit(Callable<T> task);*

*<T> Future<T> submit(Runnable task, T result);*

*Future<?> submit(Runnable task);*

第一个submit方法里面的参数类型是Callable。Callable实际上是属于Executor框架中的功能类。

## Future

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果，必要是通过get方法获取执行结果，该方法会阻塞直到任务返回结果。Future类位于java.util.concurrent包下，接口如下：

*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;*

*}*

在Future中声明了5个方法：

* cancel，取消任务，如果取消成功则返回true，否则false。参数mayInterruptIfRunning表示允许取消正在执行却没有执行完毕的任务，如果设置true，则表示可以取消正在执行的任务。如果任务已经完成，则无论mayInterruptIfRunning为true还是flase，此方法肯定返回false，即如果取消已经完成的任务会返回false。如果任务正在执行，若设置为true，则返回true，若mayInterruptIfRunning设置为false，则返回false。如果任务没有执行，若无论mayInterruptIfRunning为true还是false，肯定返回true。
* isCancelled，表示任务是否被取消成功，如果正常取消则返回true
* isDone，表示任务是否已经完成，若完成则返回true
* get方法，用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回
* get(long timeout,TimeUnit unit)，用来获取执行结果，如果在指定时间内还没有获取结果则返回null

综上所述，Future提供三种功能：

* 判断任务是否完成
* 能够中断任务
* 能够获取任务执行结果

## FutureTask

Future只是一个借口，所以无法直接用来创建对象，因此就有了FutureTask，其类继承关系如下：

*public class FutureTask<V> implements RunnableFuture<V>*

FutureTask类实现了RunnableFuture接口，RunnableFuture接口实现，如下：

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

*void run();*

*}*

可以看出FutureTask既可以作为Runnable线程执行，又可以作为Future得到Callable的返回值。FutureTask的2个构造器如下：

*public FutureTask(Callable<V> callable) {*

*if (callable == null) throw new NullPointerException();*

*this.callable = callable;*

*this.state = NEW; // ensure visibility of callable*

*}*

*public FutureTask(Runnable runnable, V result) {*

*this.callable = Executors.callable(runnable, result);*

*this.state = NEW; // ensure visibility of callable*

*}*

## Callable + Future获取执行结果

定义Task，继承Callable：

*public class Task implements Callable<Integer> {*

*@Override*

*public Integer call() throws Exception {*

*System.out.println("child thread computing");*

*Thread.sleep(3000);*

*int sum = 0;*

*for(int i= 0 ;i < 100 ;i++){*

*sum += i;*

*}*

*return sum;*

*}}*

定义主测试程序：

*public class TestCallable {*

*public static void main(String[] args) {*

*ExecutorService executor = Executors.newCachedThreadPool();*

*Task task = new Task();*

*Future<Integer> result = executor.submit(task);*

*try {*

*Thread.sleep(1000);*

*} catch(InterruptedException e) {*

*e.printStackTrace();*

*}*

*System.out.println("Main Thread is running");*

*try {*

*System.out.println("task running result:"+result.get());*

*} catch(InterruptedException| ExecutionException e) {*

*e.printStackTrace();*

*}*

*executor.shutdownNow();*

*System.out.println("All Task run successful");*

*}*

*}*

执行结果如下：

*child thread computing*

*Main Thread is running*

*task running result:4950*

*All Task run successful*

## 使用Callable和FutureTask获取执行结果

子线程使用上例中的Task，定义使用FutureTask的主程序：

*public class TestFutureTask {*

*public static void main(String[] args) {*

*ExecutorService executor = Executors.newCachedThreadPool();*

*Task task = new Task();*

*FutureTask<Integer> futureTask = new FutureTask<Integer>(task);*

*executor.submit(futureTask);*

*try {*

*Thread.sleep(1000);*

*} catch(InterruptedException e) {*

*e.printStackTrace();*

*}*

*System.out.println("Main Thread running");*

*try {*

*System.out.println("task result:"+futureTask.get());*

*} catch(InterruptedException | ExecutionException e) {*

*e.printStackTrace();*

*}*

*System.out.println("All Task running finished");*

*executor.shutdownNow();*

*}}*

如果为了可取消性而使用Future，但又不提供可用的结果，则可以声明Future<?>形式类型，并返回null作为底层任务的结果。

# Netty ChannelFuture

在Netty中，所有的IO操作都是异步的，可以等操作完成或者直接注册一个监听，具体的显示是通过Future和ChannelFuture。当注册一个监听，当操作执行成功或失败时监听会自动触发，总之所有的操作都会返回一个ChannelFuture，包括bind、write、connect等操作会简单的返回一个ChannelFuture，调用者不能立刻获得结果。

示例如下所示：

*ChannelFuture f = serverBootStrap.bind(port).sync();*

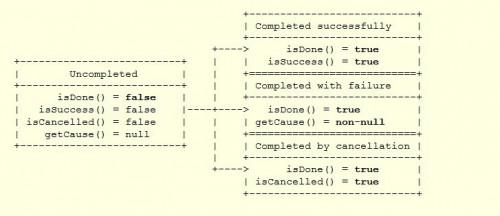
ChannelFuture继承Java Future接口，并增加Listener管理，当事件发生时，会触发监听器指定的响应的操作，

*public interface ChannelFuture extends Future<Void> {*

*@Override*

*ChannelFuture addListener(GenericFutureListener<? extends Future<? super Void>> listener);}*

ChannelFuture对象状态只有uncompleted和completed，当一个I/O操作开始时，一个ChannelFuture实例被创建：



当做一个IO操作并有任何后续任务的时候，推荐使用addListener(GenericFutureListner)方式获取通知，该方式是非阻塞的，把特定的ChannelFutureListener添加到ChannelFuture中，然后I/O线程在I/O操作相关的Future完成的时候通知监听器。如下所示：

*// BAD - NEVER DO THIS*

*@Override*

*public void channelRead(ChannelHandlerContext ctx, GoodByeMessage msg) {*

*ChannelFuture future = ctx.channel().close();*

*future.awaitUninterruptibly();*

*// Perform post-closure operation*

*// ...*

*}*

*// GOOD*

*@Override*

*public void channelRead(ChannelHandlerContext ctx, GoodByeMessage msg) {*

*ChannelFuture future = ctx.channel().close();*

*future.addListener(new ChannelFutureListener() {*

*public void operationComplete(ChannelFuture future) {*

*// Perform post-closure operation*

*// ...*

*}*

*});*

*}*

http://blog.csdn.net/javazejian/article/details/50890554

http://blog.csdn.net/zhuang\_wk/article/details/52807818

http://blog.csdn.net/javazejian/article/details/50896505

http://blog.csdn.net/xiaanming/article/details/8703708

https://www.cnblogs.com/sunfie/p/5259340.html