Please visit website: http://cxyroad.com

深入理解 Future, CompletableFuture, ListenableFuture，回调机制

深入理解 Future, CompletableFuture, ListenableFuture，回调机制

本文禁止转载。

本文从设计思想、具体实现等角度分析了 Future、CompletableFuture、ListenableFuture等接口或类，提出了一些最佳实践，精华内容为示例代码。耐心看完，相信你一定会有所收获。

理想的 Future

1. 函数式思想：值类型，和类型，结果延迟获取，最终状态有两种，正常获取结果 v 或者异常结果 ex。
2. 只有读逻辑，写逻辑由其他类实现（如 Scala 中的 Promise）
3. 支持回调，链式调用
4. 对异步支持良好，但是也支持同步调用
5. 可取消任务，取消的结果视为异常结果

现实的 Future 接口

1. Future 封装了任务(写逻辑），提供了阻塞 get()方法，但是结果为四种状态，而且是以受检异常形式抛出，每次处理时都需要 try-catch，无法分别处理。受检异常的问题是即使知道代码不会抛异常，还是要写模版代码 catch 处理。

```
V get() throws InterruptedException, ExecutionException;
V get(long timeout, TimeUnit unit) throws InterruptedException,
ExecutionException, TimeoutException;
```

2. 非强制的取消任务机制，调用 cancel 方法，若任务处于 NEW 状态，可以

保证取消；若任务处于运行状态，需要依赖于任务支持中断机制，Java 的中断机制是设立标志位，无法直接传输数据。

```
boolean cancel(boolean mayInterruptIfRunning);
```

3. Future 接口不支持回调以及链式调用，CompletableFuture 实现了此功能，但是自身也有一些问题。
4. Future 接口可以当做值类型使用，也推荐这么使用，不过很多情况下会出现对于 Future 结果的修改。note: Java 底层不支持值类型，其值类型官方说法为 value-based。需要注意区分值类型和可变数据容器 Wrapper 类，值类型是不可变的，比如 Integer 和 AtomicInteger 的区别。
5. JDK19 的改进

在 JDK19 终于提供了以下一系列方法，其基本思想就是值类型，绕过 get 阻塞调用的弊端。当已知结果已经得出后，直接取值，无需进行受检异常处理。美中不足：中断机制还是不视为异常的一部分，exceptionNow 在中断状态时，还是会中断当前线程。

```
default V resultNow();
default Throwable exceptionNow();
default State state();
enum State {
    /**
     * The task has not completed.
     */
    RUNNING,
    /**
     * The task completed with a result.
     * @see Future#resultNow()
     */
    SUCCESS,
    /**
     * The task completed with an exception.
     * @see Future#exceptionNow()
     */
    FAILED,
    /**
     * The task was cancelled.
     * @see #cancel(boolean)
```

```
     */
    CANCELLED
}
```

总之，新增的默认方法使用需要先进行状态判断，然后调用获取结果方法。以下是代码示例：

```
// 处理已运算结果
results = futures.stream()
            .filter(f -> f.state() == Future.State.SUCCESS)
            .map(Future::resultNow)
            .toList();
// allOf 运算结果处理
CompletableFuture.allOf(c1, c2, c3).join();
foo(c1.resultNow(), c2.resultNow(), c3.resultNow());
```

不过，默认方法对于第三方库来说可能会出现运行问题，需要确认第三方库支持再使用默认方法。

CompletableFuture 大幅改进

岁月史书：Future, ExecutorService, BlockingQueue, AQS 等在 JDK1.5 推出，随后 Google Guava 类库提供了 ListenableFuture 补全了 Future 的回调功能，CompletableFuture 借鉴了 ListenableFuture 功能，结合函数式支持，在 JDK1.8 推出。

1. 链式调用

```
public class CompletableFuture<T> implements Future<T>,
CompletionStage<T>
```

大部分链式调用方法定义在 CompletionStage 中：

CompletionStage 接口方法	函数式接口	Stream 相似方法
`<U> CompletionStage<U> thenApply(Function<? super T,? extends U> fn);`	Function:: apply	map
`CompletionStage<Void> thenAccept(Consumer<? super T> action);`	Consumer: accept	forEach
`CompletionStage<Void> thenRun(Runnable action);`	Runnable:: run	forEach
`<U,V> CompletionStage<V> thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn);`	BiFunction:: apply	Guava#Streams# zip
`<U> CompletionStage<U> applyToEither(CompletionStage<? extends T> other, Function<? super T, U> fn);`	Function:: apply	None
`<U> CompletionStage<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> fn);`	Function:: apply	flatMap
`runAfterEither, acceptEither`	略	None

异常处理：

CompletionStage 接口方法	说明
`<U> CompletionStage<U> handle(BiFunction<? super T, Throwable, ? extends U> fn);`	和类型结果处理，对应 Stream:: map 方法
`CompletionStage<T> whenComplete(BiConsumer<? super T, ? super Throwable> action);`	和类型结果处理
`CompletionStage<T> exceptionally(Function<Throwable, ? extends T> fn);`	更常用，因为以上 handle 方法本质上是两个逻辑，拆分出来更合理
`default CompletionStage<T> exceptionallyCompose (Function<Throwable, ? extends CompletionStage<T>> fn)`	JDK12，异常处理的 flatMap 版本

以上方法均有异步（asnyc）方法，实际开发中通常使用异步调用，充分利用异步的能力，这也是异步回调的意义。

链式调用使用了函数式编程中的 [Railway Oriented Programming](http://cxyroad.com/ ”https://fsharpforfunandprofit.com/rop/”）模式，存在两种处理管道，正常

处理流程和异常处理流程，各个方法单独处理某个流程，比如 thenCompose 处理正常数据，exceptionallyCompose 处理异常数据，不能处理则返回自身。

> 最佳实践: 异步调用方法的返回值不要返回 null, 不要使用 handle, whenComplete 方法，可以使用 NullObject 模式或者 Optional 包装，这样可以最大限度地利用函数式的能力。

2. Future 接口语义优化

> CompletableFuture also implements Future with the following policies:
> Since (unlike FutureTask) this class has no direct control over the computation that causes it to be completed, cancellation is treated as just another form of exceptional completion. Method cancel has the same effect as completeExceptionally(new CancellationException()). Method isCompletedExceptionally can be used to determine if a CompletableFuture completed in any exceptional fashion.
> In case of exceptional completion with a CompletionException, methods get() and get(long, TimeUnit) throw an ExecutionException with the same cause as held in the corresponding CompletionException. To simplify usage in most contexts, this class also defines methods join() and getNow that instead throw the CompletionException directly in these cases.

总之，CompletableFuture 取消视为异常结果，提供快速获取结果的方法，获取值的方法不抛出受检异常。

join 为阻塞方法，getNow 不阻塞。

以下是值类型支持：

运行时设置结果：boolean complete(T value)， boolean completeExceptionally(Throwable ex)

拷贝方法（JDK9）：CompletableFuture copy()

拷贝方法属于防御性编程，尽量避免了对值的修改，虽然无法完全避免，但是

也说明了将 Future 当做值类型使用是源码所提倡的。

不好的方法：支持强制写结果，但是不保证回调使用的是重写的方法，所以为了避免代码出现意想不到的问题，这个方法完全不推荐使用。

```
public void obtrudeValue(T value) {
    result = (value == null) ? NIL : value;
    postComplete();
}

public void obtrudeException(Throwable ex) {
    if (ex == null) throw new NullPointerException();
    result = new AltResult(ex);
    postComplete();
}
```

3. 初始化(写入值)

支持三种创建方式：

1. 传入任务（Supplier 或者 Runnable)静态方法创建。
2. 传入空参数，表示暂未绑定任务，实现延迟绑定，搭配 completeAsync、orTimeout 等使用。
3. 直接绑定值。

```
public static <U> CompletableFuture<U> supplyAsync(Supplier<U>
supplier);
public static CompletableFuture<Void> runAsync(Runnable runnable);
// 直接绑定值
public static <U> CompletableFuture<U> completedFuture(U value)
public static <U> CompletableFuture<U> failedFuture(Throwable ex); //
JDK9

public CompletableFuture() {} // incompleteFuture
public boolean completeExceptionally(Throwable ex);
// JDK9 支持任务延迟绑定
public CompletableFuture<T> completeAsync(Supplier<? extends T>
```

supplier, Executor executor);
public CompletableFuture<T> orTimeout(long timeout, TimeUnit unit);
public CompletableFuture<T> completeOnTimeout(T value, long timeout, TimeUnit unit)

```

> 超时方法应该优先使用，避免出现长时间等待，预估好任务执行时间的上届，出现问题可以及时发现。

### 4. 任务编排与异常结果传播

#### allOf, anyOf

```
public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

```

以上为最常见的两种任务编排方式，allOf 当所有任务成功时，返回值为 null，否则返回异常；anyOf 当任意一个任务执行结束后，返回子任务结果值，不区分正常结果和异常值。如果需要返回任意一个正确的返回值，可以自行对结果进行封装，可以参考如下代码：

```
@Slf4j
class AnyOfDemo {
    final static Executor executor = Executors.newCachedThreadPool();

    public static void main(String[] args) {
        var exCf = supplyAsync(AnyOfDemo::exceptionalTask, executor);
        var normalCf = supplyAsync(AnyOfDemo::normalTask, executor);
        var timedCf = supplyAsync(AnyOfDemo::timedTask, executor);
        var cfs = List.of(exCf, normalCf, timedCf);
       // 若结果为异常，转换为未完成的 CompletableFuture
        @SuppressWarnings("unchecked")
        var wrappedCfs = (CompletableFuture<Integer>[]) cfs.stream()
            .map(cf -> exceptionAsIncomplete(cf, executor))
```
```

```
            .toArray(CompletableFuture[]::new);
// 任务设置超时时间，提高容错
      // 对于任务执行情况，记录日志
      // 异常处理在 exceptionally 方法中，函数式的处理方式，避免向外抛出异常，不影响主线程运行
      // 为了这段代码包的饺子
        anyOf(wrappedCfs)
            .orTimeout(10, TimeUnit.SECONDS)
            .whenCompleteAsync((v, ex) -> log.info("取消子任务: {}", cancelAll(cfs)), executor)
            .thenAcceptAsync(x -> log.info("获取结果为: {}", x), executor)
            .exceptionallyAsync(ex -> {
                log.error("执行异常", ex);
                return null;
            })
            .join();
    }

    static Integer exceptionalTask() {
      // 仅做演示需要，多数情况下任务都需要做异常处理
        throw new RuntimeException("执行异常");
    }

    static Integer normalTask() {
        return ThreadLocalRandom.current().nextInt();
    }


    static Integer timedTask() {
        try {
            Thread.sleep(Duration.ofSeconds(15));
        } catch (InterruptedException e) {
            throw new RuntimeException("中断任务", e);
        }
        log.info("定时任务结束");
        return 15;
    }

    static <T> CompletableFuture<T> exceptionAsIncomplete(CompletableFuture<T> cf, Executor executor) {
        return cf.exceptionallyComposeAsync(ex -> {
            log.warn("子任务执行失败", ex);
            return new CompletableFuture<>();
        }, executor);
    }

    static <T> List<Boolean> cancelAll(List<CompletableFuture<T>> cfs) {
        return cfs.stream().map(cf -> cf.cancel(true)).toList();
```

```
    }
}
```

执行日志如下：

```
> Task :AnyOfDemo.main()
15:47:15.108 [pool-1-thread-3] INFO
com.example.demo.futures.AnyOfDemo -- 取消子任务: [false, false, true]
15:47:15.107 [pool-1-thread-2] WARN
com.example.demo.futures.AnyOfDemo -- 子任务执行失败
java.util.concurrent.CompletionException: java.lang.RuntimeException: 执行异常
at
java.base/java.util.concurrent.CompletableFuture.encodeThrowable(CompletableFuture.java:315)
at
java.base/java.util.concurrent.CompletableFuture.completeThrowable(CompletableFuture.java:320)
at
java.base/java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1770)
at
java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)
at
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:642)
at java.base/java.lang.Thread.run(Thread.java:1583)
Caused by: java.lang.RuntimeException: 执行异常
Caused by: java.lang.RuntimeException: 执行异常

at
com.example.demo.futures.AnyOfDemo.exceptionalTask(FutureDemo.java:99)
at
java.base/java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1768)
... 3 common frames omitted
15:47:15.112 [pool-1-thread-5] INFO
com.example.demo.futures.AnyOfDemo -- 获取结果为: -1390753190
15:47:15.108 [pool-1-thread-4] WARN
com.example.demo.futures.AnyOfDemo -- 子任务执行失败
java.util.concurrent.CancellationException: null
```

at
java.base/java.util.concurrent.CompletableFuture.cancel(CompletableFut
ure.java:2510)
at
com.example.demo.futures.AnyOfDemo.lambda$cancelAll$6(FutureDemo
.java:125)
at
java.base/java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeli
ne.java:197)
at
java.base/java.util.AbstractList$RandomAccessSpliterator.forEachRemai
ning(AbstractList.java:722)
at
java.base/java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java
:509)
at
java.base/java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPip
eline.java:499)
at
java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java
:575)
at
java.base/java.util.stream.AbstractPipeline.evaluateToArrayNode(Abstrac
tPipeline.java:260)
at
java.base/java.util.stream.ReferencePipeline.toArray(ReferencePipeline.ja
va:616)
at
java.base/java.util.stream.ReferencePipeline.toArray(ReferencePipeline.ja
va:622)
at
java.base/java.util.stream.ReferencePipeline.toList(ReferencePipeline.jav
a:627)
at
com.example.demo.futures.AnyOfDemo.cancelAll(FutureDemo.java:125)
at
com.example.demo.futures.AnyOfDemo.lambda$main$2(FutureDemo.java
:89)
at
java.base/java.util.concurrent.CompletableFuture.uniWhenComplete(Com
pletableFuture.java:863)
at
java.base/java.util.concurrent.CompletableFuture$UniWhenComplete.tryF
ire(CompletableFuture.java:841)
at
java.base/java.util.concurrent.CompletableFuture$Completion.run(Compl
etableFuture.java:482)
at

java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPool
Executor.java:1144)
at
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPo
olExecutor.java:642)
at java.base/java.lang.Thread.run(Thread.java:1583)
15:47:30.110 [pool-1-thread-1] INFO
com.example.demo.futures.AnyOfDemo -- 定时任务结束

```

分析执行日志，可以看出：

* `15:47:15.112 [pool-1-thread-5] INFO
com.example.demo.futures.AnyOfDemo -- 获取结果为: -1390753190`

任务执行成功
* `15:47:15.107 [pool-1-thread-2] WARN
com.example.demo.futures.AnyOfDemo -- 子任务执行失败`
`java.util.concurrent.CompletionException: java.lang.RuntimeException:
执行异常`

日志记录了子任务执行失败情况，此为 exceptionalTask 抛出异常。

*Caused by: java.lang.RuntimeException: 执行异常* 显示了两次，一次是日志，另一次是执行线程抛出的，因为没有 try-catch 异常处理。
* `15:47:15.108 [pool-1-thread-3] INFO
com.example.demo.futures.AnyOfDemo -- 取消子任务: [false, false, true]`

`15:47:15.108 [pool-1-thread-4] WARN
com.example.demo.futures.AnyOfDemo -- 子任务执行失败`
`java.util.concurrent.CancellationException: null`

执行成功后中断了超时的子任务
* `15:47:30.110 [pool-1-thread-1] INFO
com.example.demo.futures.AnyOfDemo -- 定时任务结束`

**CompletableFuture 并不会控制任务执行过程**，15 秒后，timedTask 才完全中断。
```

DAG

任务编排可以表示为有向无环图（DAG）

这个图是笔者从 wiki 百科上找的，其 DAG 任务编排的简单实现如下：

```
@Slf4j
class DagDemo {
  // 代码仅作演示 DAG 用，不考虑 executor，异常处理等情况
    public static void main(String[] args) {
        var a = supplyAsync(DagDemo::randomTask)
            .thenApply(v -> log("a", v));
        var b = a.thenApply(x -> x + 1)
            .thenApply(v -> log("b", v));
        var c = a.thenApply(x -> x * 2)
            .thenApply(v -> log("c", v));
      // flatMap 语义，此技巧可以实现多参数方法调用
        var d = a.thenCompose(x -> b.thenCompose(y -> c.thenApply(z ->
x + y + z)))
            .thenApply(v -> log("d", v));
      // 使用 allOf 实现相同语义
        var e = allOf(a, c, d).thenApply(x -> List.of(a.join(), c.join(), d.join()))
            .thenApply(v -> log("e", v));
        e.join();
    }

    static int randomTask() {
        return ThreadLocalRandom.current().nextInt(10);
    }

    static <T> T log(String taskName, T v) {
        log.info("{}: {}", taskName, v);
        return v;
    }
}
```

```
// 运行结果如下
17:16:46.737 [main] INFO com.example.demo.futures.DagDemo -- a: 6
17:16:46.740 [main] INFO com.example.demo.futures.DagDemo -- b: 7
17:16:46.741 [main] INFO com.example.demo.futures.DagDemo -- c: 12
17:16:46.742 [main] INFO com.example.demo.futures.DagDemo -- d: 25
17:16:46.743 [main] INFO com.example.demo.futures.DagDemo -- e: [6,
12, 25]

```

Promise 写功能

以下是一个简单的单一生产者消费者实现，从 [Scala Promise 示例代码](http://cxyroad.com/ "https://docs.scala-lang.org/overviews/core/futures.html#projections") 借鉴的 CompletableFuture 实现。

简单来说，Promise 实现了 Future 的写功能

```
import scala.concurrent.{ Future, Promise }
import scala.concurrent.ExecutionContext.Implicits.global

val p = Promise[T]()
val f = p.future

val producer = Future {
  val r = produceSomething()
  p.success(r)
  continueDoingSomethingUnrelated()
}

val consumer = Future {
  startDoingSomething()
  f.foreach { r =>
    doSomethingWithResult()
  }
}

```

不妨将以上抽象代码具体化，如下示例代码中生产者读取文件信息，消费者对

文件进行单词计数：

```
@Slf4j
class PromiseDemo {
    public static void main(String[] args) {
        var executorService = Executors.newCachedThreadPool();
        var cf = new CompletableFuture<String>();
        var producer = runAsync(() -> {
            log.info("开始加载文件");
            String s = loadFile();
            cf.complete(s);
            log.info("读取文件成功");
        }, executorService);
        var consumer = runAsync(() -> {
            log.info("开始计数");
            String s = cf.join();
            int count = count(s);
            log.info("count = " + count);
        }, executorService);
        // 安全结束，避免异常抛出
        allOf(producer, consumer)
            .orTimeout(30, TimeUnit.SECONDS)
            .exceptionallyAsync(ex -> {
                log.error("执行异常", ex);
                return null;
            })
            .join();
    }

    static String loadFile() {
        return "abandon ability";
    }

    static int count(String s) {
        return (int) Arrays.stream(s.split("\\s+")).distinct().count();
    }
}
```

执行结果如下：

```
```

```
17:29:52.956 [pool-1-thread-1] INFO
com.example.demo.futures.PromiseDemo -- 开始加载文件
17:29:52.956 [pool-1-thread-2] INFO
com.example.demo.futures.PromiseDemo -- 开始计数
17:29:52.959 [pool-1-thread-1] INFO
com.example.demo.futures.PromiseDemo -- 读取文件成功
17:29:52.965 [pool-1-thread-2] INFO
com.example.demo.futures.PromiseDemo -- count = 2
```

5. 继承

学习一个类，首先需要学习怎么使用 API，使用了什么算法，其次需要了解其对继承的支持，如何拓展，最后才是源码。

CompletableFuture 中没有 protected 方法，文档或源码中搜索 subclass、override 等关键字即可找到相关拓展点。

最重要的拓展点是 defaultExecutor()方法，默认使用的是 commonPool，适用于计算密集型任务，但是不同的任务最好创建不同的线程池，子类可以重写这个方法，不用在每次调用 async 方法时都显式传 executor 参数。

6. 其他

新增了一些延迟任务支持，本文不作分析。

只有两个字段，结果和回调 stack。

```
volatile Object result;       // Either the result or boxed AltResult
volatile Completion stack;
```

JDK9 提供了转换到 CompletionStage 的方法，至此两者可以相关转换。但是这种设计严重违反接口隔离原则，如果我只想使用 CompletaionStage 中的方法，直接声明类型为 CompletionStage 即可，为什么还要舍近求远搞一个类似 Collections.unmodifiableList 的实现。而且接口里 toCompletableFuture 方法也是不明所以，接口竟然可以返回子类实现。

```
 // 以下代码请速读, 实际上子类很难继承实现
/**
  * Returns a new CompletionStage that is completed normally with
  * the same value as this CompletableFuture when it completes
  * normally, and cannot be independently completed or otherwise
  * used in ways not defined by the methods of interface {@link
  * CompletionStage}.  If this CompletableFuture completes
  * exceptionally, then the returned CompletionStage completes
  * exceptionally with a CompletionException with this exception as
  * cause.
  *
  * <p> Unless overridden by a subclass, a new non-minimal
  * CompletableFuture with all methods available can be obtained from
  * a minimal CompletionStage via {@link #toCompletableFuture()}.
  * For example, completion of a minimal stage can be awaited by
  *
  * <pre> {@code minimalStage.toCompletableFuture().join(); }</pre>
  *
  * @return the new CompletionStage
  * @since 9
  */
 public CompletionStage<T> minimalCompletionStage() {
     return uniAsMinimalStage();
 }

 private MinimalStage<T> uniAsMinimalStage() {
     Object r;
     if ((r = result) != null)
         return new MinimalStage<T>(encodeRelay(r));
     MinimalStage<T> d = new MinimalStage<T>();
     unipush(new UniRelay<T,T>(d, this));
     return d;
 }

 /**
  * A subclass that just throws UOE for most non-CompletionStage
methods.
  */
 static final class MinimalStage<T> extends CompletableFuture<T> {
     MinimalStage() { }
     MinimalStage(Object r) { super(r); }
     @Override public <U> CompletableFuture<U>
newIncompleteFuture() {
         return new MinimalStage<U>(); }
```

```
@Override public T get() {
    throw new UnsupportedOperationException(); }
@Override public T get(long timeout, TimeUnit unit) {
    throw new UnsupportedOperationException(); }
@Override public T getNow(T valueIfAbsent) {
    throw new UnsupportedOperationException(); }
@Override public T join() {
    throw new UnsupportedOperationException(); }
@Override public T resultNow() {
    throw new UnsupportedOperationException(); }
@Override public Throwable exceptionNow() {
    throw new UnsupportedOperationException(); }
@Override public boolean complete(T value) {
    throw new UnsupportedOperationException(); }
@Override public boolean completeExceptionally(Throwable ex) {
    throw new UnsupportedOperationException(); }
@Override public boolean cancel(boolean mayInterruptIfRunning) {
    throw new UnsupportedOperationException(); }
@Override public void obtrudeValue(T value) {
    throw new UnsupportedOperationException(); }
@Override public void obtrudeException(Throwable ex) {
    throw new UnsupportedOperationException(); }
@Override public boolean isDone() {
    throw new UnsupportedOperationException(); }
@Override public boolean isCancelled() {
    throw new UnsupportedOperationException(); }
@Override public boolean isCompletedExceptionally() {
    throw new UnsupportedOperationException(); }
@Override public State state() {
    throw new UnsupportedOperationException(); }
@Override public int getNumberOfDependents() {
    throw new UnsupportedOperationException(); }
@Override public CompletableFuture<T> completeAsync
    (Supplier<? extends T> supplier, Executor executor) {
    throw new UnsupportedOperationException(); }
@Override public CompletableFuture<T> completeAsync
    (Supplier<? extends T> supplier) {
    throw new UnsupportedOperationException(); }
@Override public CompletableFuture<T> orTimeout
    (long timeout, TimeUnit unit) {
    throw new UnsupportedOperationException(); }
@Override public CompletableFuture<T> completeOnTimeout
    (T value, long timeout, TimeUnit unit) {
    throw new UnsupportedOperationException(); }
@Override public CompletableFuture<T> toCompletableFuture() {
    Object r;
    if ((r = result) != null)
        return new CompletableFuture<T>(encodeRelay(r));
```

```
            else {
                CompletableFuture<T> d = new CompletableFuture<>();
                unipush(new UniRelay<T,T>(d, this));
                return d;
            }
        }
    }

```

ListenableFuture

鉴于本文前面已经分析了 Future 实现的基本思想，同时 Guava 提供了[详细文档](http://cxyroad.com/ "https://guava.dev/releases/21.0-rc1/api/docs/com/google/common/util/concurrent/ListenableFuture.html")说明，下面仅总结 ListenableFuture 的某些关键实现：

* 遵循替换原则，子类（子接口）不改变父类（父接口）语义，ListenableFuture 拓展了 Future 接口，支持回调。
* 保留了原本的 cancel 机制，可以取消线程池中的任务。
* CompletableFuture 和 ListenableFuture 可以实现的功能相似，命名方式大多不同。
* 推荐 ListeningExecutorService 和 ListenableFuture 搭配使用，ListeningExecutorService 可以无缝替换 ExecutorService，提交的任务返回 ListenableFuture, 还新增了一些定时任务相关方法。
* 支持链式调用（使用 FluentFuture）
* Futures 提供了很多易用的方法，如 allAsList, whenAllComplete, whenAllSucceed

```
// 定义
@DoNotMock("Use the methods in Futures (like immediateFuture) or SettableFuture")
@ElementTypesAreNonnullByDefault
public interface ListenableFuture<V extends @Nullable Object> extends Future<V> {
  void addListener(Runnable listener, Executor executor);
}

// 官方示例，链式调用方法少但简洁，所有的 CompletableFuture 方法都可以实现，本质上是 map, flatMap
ListenableFuture<Boolean> adminIsLoggedIn =
FluentFuture.from(usersDatabase. getAdminUser())
```

```
  .transform(User::getId, directExecutor())
  .transform(ActivityService::isLoggedIn, threadPool)
  .catching(RpcException.class, e -> false, directExecutor());
```

我们可以使用 ListenableFuture 再次实现之前的 anyOf 语义：

```
@Slf4j
class AnyOfListenableDemo {
  // 装饰器模式实现功能新增
    final static ListeningExecutorService executor =
MoreExecutors.listeningDecorator(newCachedThreadPool());

    public static void main(String[] args) {
      // Futures 没有 anyOf 的直接实现，如果有实现的话请告诉我，这里使用
SettableFuture 实现
      // 这里展示了两种添加回调的方式，一种是使用 Futures, 另一种是使用
FluentFuture 链式调用
        var normalLf =
executor.submit(AnyOfListenableDemo::normalTask);
        var exLf =
executor.submit(AnyOfListenableDemo::exceptionalTask);
        var timedLf = executor.submit(AnyOfListenableDemo::timedTask);
        var lfs = List.of(normalLf, exLf, timedLf);
      // SettableFuture 实现了单次设置值，set 方法只能成功一次
        var anyLf = SettableFuture.<Integer>create();
        Futures.addCallback(normalLf, getCallback("normal", anyLf),
executor);
        Futures.addCallback(exLf, getCallback("exceptionally", anyLf),
executor);
        Futures.addCallback(timedLf, getCallback("timed", anyLf),
executor);
        anyLf.addListener(() -> lfs.forEach(f -> f.cancel(true)), executor);

        var fluentFut = FluentFuture.from(anyLf)
            .withTimeout(Duration.ofSeconds(10),
newScheduledThreadPool(1))
            .transform(x -> {
                log.info("任务执行成功,结果: {}", x);
                return x;
            }, executor)
            .catching(Throwable.class, t -> {
                log.error("任务执行失败", t);
                return 0;
```

```
            }, executor);
        Futures.getUnchecked(fluentFut);
    }

    static FutureCallback<Integer> getCallback(String taskName,
                                               SettableFuture<Integer> future) {
        return new FutureCallback<>() {
            @Override
            public void onSuccess(Integer result) {
                future.set(result);
            }

            @Override
            public void onFailure(Throwable t) {
                log.info("{}任务执行失败", taskName, t);
            }
        };
    }

    static Integer exceptionalTask() {
        throw new RuntimeException("执行异常");
    }

    static Integer normalTask() {
        return ThreadLocalRandom.current().nextInt();
    }

    static Integer timedTask() {
        try {
            Thread.sleep(Duration.ofSeconds(15));
        } catch (InterruptedException e) {
            log.info("中断任务", e);
            throw new RuntimeException(e);
        }
        log.info("定时任务结束");
        return 15;
    }
}

```

```
20:46:46.381 [pool-1-thread-1] INFO
com.example.demo.futures.AnyOfListenableDemo -- 任务执行成功,结果:
-1967447682
20:46:46.369 [pool-1-thread-2] INFO
com.example.demo.futures.AnyOfListenableDemo -- exceptionally任务执
```

行失败
java.lang.RuntimeException: 执行异常
at
com.example.demo.futures.AnyOfListenableDemo.exceptionalTask(PromiseDemo.java:152)
at
com.google.common.util.concurrent.TrustedListenableFutureTask$TrustedFutureInterruptibleTask.runInterruptibly(TrustedListenableFutureTask.java:131)
at
com.google.common.util.concurrent.InterruptibleTask.run(InterruptibleTask.java:76)
at
com.google.common.util.concurrent.TrustedListenableFutureTask.run(TrustedListenableFutureTask.java:82)
at
java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)
at
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:642)
at java.base/java.lang.Thread.run(Thread.java:1583)
20:46:46.386 [pool-1-thread-3] INFO
com.example.demo.futures.AnyOfListenableDemo -- 中断任务
java.lang.InterruptedException: sleep interrupted
at java.base/java.lang.Thread.sleep0(Native Method)
at java.base/java.lang.Thread.sleep(Thread.java:592)
at
com.example.demo.futures.AnyOfListenableDemo.timedTask(PromiseDemo.java:161)
at
com.google.common.util.concurrent.TrustedListenableFutureTask$TrustedFutureInterruptibleTask.runInterruptibly(TrustedListenableFutureTask.java:131)
at
com.google.common.util.concurrent.InterruptibleTask.run(InterruptibleTask.java:76)
at
com.google.common.util.concurrent.TrustedListenableFutureTask.run(TrustedListenableFutureTask.java:82)
at
java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)
at
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:642)
at java.base/java.lang.Thread.run(Thread.java:1583)
20:46:46.386 [pool-1-thread-2] INFO

```
com.example.demo.futures.AnyOfListenableDemo -- timed任务执行失败
java.util.concurrent.CancellationException: Task was cancelled.
at
com.google.common.util.concurrent.AbstractFuture.cancellationExceptio
nWithCause(AbstractFuture.java:1572)
at
com.google.common.util.concurrent.AbstractFuture.getDoneValue(Abstra
ctFuture.java:592)
at
com.google.common.util.concurrent.AbstractFuture.get(AbstractFuture.ja
va:553)
at
com.google.common.util.concurrent.FluentFuture$TrustedFuture.get(Flue
ntFuture.java:91)
at
com.google.common.util.concurrent.Uninterruptibles.getUninterruptibly(U
ninterruptibles.java:247)
at
com.google.common.util.concurrent.Futures.getDone(Futures.java:1180)
at
com.google.common.util.concurrent.Futures$CallbackListener.run(Future
s.java:1128)
at
java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPool
Executor.java:1144)
at
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPo
olExecutor.java:642)
at java.base/java.lang.Thread.run(Thread.java:1583)

```

从执行日志看，相比使用 CompletableFuture 的实现，可以实现完全的取消机制。

总结

1. 使用 ListenableFuture 还是 CompletableFuture?

各有利弊，以下是我的个人看法：如果你使用的是 JDK8 的话，不推荐使用 CompletableFuture，因为有很多方法支持不足，比如设置超时时间，虽然可以使用一些工具类实现相关功能；不如直接使用 Guava，代码不易出错，文档齐全。 JDK8 以上版本，首选 CompletableFuture，需要取消机制时，可以使用 ListenableFuture 搭配 ListeningExectuorService。

2. 潜龙勿用。只有在真正需要使用异步处理时（如性能、响应时间要求）再使用异步处理。
3. 很多情况下，我们需要的可能不是回调，而是配置良好的线程池。
 原文链接: https://juejin.cn/post/7388332481739882547