Путеводитель по методам класса java.util.concurrent.CompletableFuture

Появившийся в Java8 класс [CompletableFuture](http://download.java.net/lambda/b88/docs/api/java/util/concurrent/CompletableFuture.html) — средство для передачи информации между параллельными потоками исполнения. По существу это блокирующая очередь, способная передать только одно ссылочное значение. В отличие от обычной очереди, передает также исключение, если оно возникло при вычислении передаваемого значения.  
  
Класс содержит несколько десятков методов, в которых легко потеряться. Данная статья классифицирует эти методы по нескольким признакам, чтобы в них было легко ориентироваться.  
  
Для разминки познакомимся с новыми интерфейсами из пакета java.util.Function, которые используются как типы параметров во многих методах.

*// два параметра, возвращает результат*

BiFunction<T, U,R> { R apply(T t, U u); }

*// два параметра, не возвращает результат*

BiConsumer<T,U> { void accept(T t, U u) }

*// один параметр, возвращает результат*

Function<T, R> { R apply(T t); }

*// один параметр, не возвращает результат*

Consumer<T> { void accept(T t); }

*// Без параметров, возвращает результат*

Supplier<T> { T get(); }

Вспомним также старый добрый Runnable:

*// Без параметров, не возвращает результат*

Runnable { void run(); }

Эти интерфейсы являются функциональными, то есть, значения этого типа могут быть заданы как ссылками на объекты, так и ссылками на методы или лямбда-выражениями.  
  
Как средство передачи данных, класс CompletableFuture имеет два суб-интерфейса — для записи и для чтения, которые в свою очередь делятся на непосредственные (синхронные) и опосредованные (асинхронные). Программно выделен только суб-интерфейс непосредственного чтения (java.util.concurrent.Future, существующий со времен java 5), но в целях классификации полезно мысленно выделять и остальные. Кроме этого разделения по суб-интерфейсам, я также буду стараться отделять базовые методы и методы, реализующие частные случаи.  
  
Для краткости вместо “объект типа CompletableFuture” будем говорить “фьючерс”. «Данный фьючерс» означает фьючерс, к которому применятся описываемый метод.

1. Интерфейс непосредственной записи

Базовых методов, понятно, два — записать значение и записать исключение:

boolean complete(T value)

boolean completeExceptionally(Throwable ex)

с очевидной семантикой.  
  
Прочие методы:

boolean cancel(boolean mayInterruptIfRunning)

эквивалентен completeExceptionally(new CancellationException). Введен для совместимости с java.util.concurrent.Future.

static <U> CompletableFuture<U> completedFuture(U value)

эквивалентен CompletableFuture res=new CompletableFuture(); res.complete(value).

void obtrudeValue(T value)

void obtrudeException(Throwable ex)

Насильно перезаписывают хранящееся значение. Верный способ выстрелить себе в ногу.

2. Интерфейс непосредственного чтения

boolean isDone()

Проверяет, был ли уже записан результат в данный фьючерс.

T get()

Ждет, если результат еще не записан, и возвращает значение. Если было записано исключение, бросает ExecutionException.  
  
Прочие методы:

boolean isCancelled()

проверяет, было ли записано исключение с помощью метода cancel().

T join()

То же, что get(), но бросает CompletionException.

T get(long timeout, TimeUnit unit)

get() с тайм-аутом.

T getNow(T valueIfAbsent)

возвращает результат немедленно. Если результат еще не записан, возвращает значение параметра valueIfAbsent.

int getNumberOfDependents()

примерное число других CompletableFuture, ждущих заполнения данного.

3. Интерфейс опосредованной записи

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

Запускается задача с функцией supplier, и результат выполнения записывается во фьючерс. Запуск задачи производится на стандартном пуле потоков.

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)

То же самое, но запуск на пуле потоков, указанном параметром executor.

static CompletableFuture<Void> runAsync(Runnable runnable)

static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)

То же самое, что и supplyAsync, но акция типа Runnable и, соответственно, результат будет типа Void.

4. Интерфейс опосредованного чтения

Предписывает выполнить заданное действие (реакцию) немедленно по заполнению этого (и/или другого) фьючерса. Самый обширный суб-интерфейс. Классифицируем его составляющие по двум признакам:  
  
а) способ запуска реакции на заполнение: возможно запустить ее синхронно как метод при заполнении фьючерса, или асинхронно как задачу на пуле потоков. В случае асинхронного запуска используются методы с суффиксом Async (в двух вариантах — запуск на общем потоке ForkJoinPool.commonPool(), либо на потоке, указанном дополнительным параметром). Далее будут описываться только методы для синхронного запуска.  
  
б) топология зависимости между данным фьючерсом и реакцией на его заполнение: типа “линейная”, типа “any“ и типа ”all”.  
  
— линейная зависимость: один фьючерс поставляет одно значение в реакцию  
  
— способ “any” — на входе два или более фьючерса; первый (по времени) результат, появившийся в одном из фьючерсов, передается в реакцию; остальные результаты игнорируются  
  
— способ “all” — на входе два или более фьючерса; результаты всех фьючерсов накапливаются и затем передаются в реакцию.

4.1 Выполнить реакцию по заполнению данного фьючерса (линейная)

Эти методы имеют имена, начинающиеся с префикса then, имеют один параметр — реакцию, и возвращают новый фьючерс типа CompletableFuture для доступа к результату исполнения реакции. Различаются по типу реакции.

<U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)

Основной метод, в котором реакция получает значение из данного фьючерса и возвращаемое значения передается в результирующий фьючерс.

CompletableFuture<Void> thenAccept(Consumer<? super T> block)

Реакция получает значение из данного фьючерса, но не возвращает значения, так что  
значение результирующего фьючерса имеет тип Void.

CompletableFuture<Void> thenRun(Runnable action)

Реакция не получает и не возвращает значение.  
  
Пусть compute1..compute4 — это ссылки на методы. Линейная цепочка с передачей значений от шага к шагу может выглядит так:

supplyAsync(compute1).thenApply(compute2).thenApply(compute3).thenAccept(compute4);

что эквивалентно простому вызову

compute4(compute3(compute2(compute1())));

<U> CompletableFuture<U> thenCompose(Function<? super T, CompletableFuture<U>> fn)

То же, что thenApply, но реакция сама возвращает фьючерс вместо готового значения. Это может понадобиться, если нужно использовать реакцию сложной топологии.

4.2 Выполнить реакцию по заполнению любого из многих фьючерсов (any)

static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

Возвращает новый фьючерс, который заполняется когда заполняется любой из фьючерсов, переданных параметром cfs. Результат совпадает с результатом завершившегося фьючерса.

4.3 Выполнить реакцию по заполнению любого из двух фьючерсов (all)

Основной метод:

<U> CompletableFuture<U> applyToEither(CompletableFuture<? extends T> other, Function<? super T,U> fn)

Возвращает новый фьючерс, который заполняется когда заполняется данный фьючерс либо фьючерс, переданный параметром other. Результат совпадает с результатом завершившегося фьючерса.  
  
Метод эквивалентен выражению:

CompletableFuture.anyOf(this, other).thenApply(fn);

Остальные два метода отличаются лишь типом реакции:

CompletableFuture<Void> acceptEither(CompletableFuture<? extends T> other, Consumer<? super T> block)

CompletableFuture<Void> runAfterEither(CompletableFuture<?> other, Runnable action)

Непонятно, зачем было делать 3 метода \*Either (9 с учетом \*Async вариантов), когда достаточно было бы одного:

<T> CompletableFuture<T> either(CompletableFuture<? extends T> other) {

return CompletableFuture.anyOf(this, other);

}

тогда все эти методы можно было бы выразить как:

f1.applyToEither(other, fn) == f1.either(other).thenApply(fn);

f1.applyToEitherAsync(other, fn) == f1.either(other).thenApplyAsync(fn);

f1.applyToEitherAsync(other, fn, executor) == f1.either(other).thenApplyAsync(fn, executor);

f1.acceptEither(other, block) == f1.either(other).thenAccept(other);

f1.runAfterEither(other, action) == f1.either(other).thenRun(action);

и т.п. Кроме того, метод either можно было бы использовать и в других комбинациях.

4.4 Выполнить реакцию по заполнению двух фьючерсов

<U,V> CompletableFuture<V> thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)

Основной метод. Имеет на входе два фьючерса, результаты которых накапливаются и затем передаются в реакцию, являющейся функцией от двух параметров.  
  
Прочие методы отличаются типом реакции:

<U> CompletableFuture<Void> thenAcceptBoth(CompletableFuture<? extends U> other, BiConsumer<? super T,? super U> block)

реакция не возвращает значение

CompletableFuture<Void> runAfterBoth(CompletableFuture<?> other, Runnable action)

реакция не принимает параметров и не возвращает значение

4.5 Выполнить реакцию по заполнению многих фьючерсов

static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)

Возвращает CompletableFuture, завершающееся по завершению всех фьючерсов в списке параметров. Очевидный недостаток этого метода — в результирующий фьючерс не передаются значения, полученные во фьючерсах-параметрах, так что если они нужны, их нужно передавать каким-то другим способом.

4.6. Перехват ошибок исполнения

Если на каком-то этапе фьючерс завершается аварийно, исключение передается дальше по цепочке фьючерсов. Чтобы среагировать на ошибку и вернуться к нормальному исполнению, можно воспользоваться методами перехвата исключений.

CompletableFuture<T> exceptionally(Function<Throwable,? extends T> fn)

Если данный фьючерс завершился аварийно, то результирующий фьючерс завершится с результатом, выработанным функцией fn. Если данный фьючерс завершился нормально, то результирующий фьючерс завершится нормально с тем же результатом.

<U> CompletableFuture<U> handle(BiFunction<? super T,Throwable,? extends U> fn)

В этом методе реакция вызывается всегда, независимо от того, заваершился ли данный фьючерс нормально или аварийно. Если фьючерс завершился нормально с результатом r, то в реакцию будут переданы параметры (r, null), если аварийно с исключением ex, то в реакцию будут переданы параметры (null, ex). Результат реакции может быть другого типа, нежели результат данного фьючерса.  
  
Следующий пример взят из <http://nurkiewicz.blogspot.ru/2013/05/java-8-definitive-guide-to.html>:

CompletableFuture<Integer> safe = future.handle((r, ex) -> {

if (r != null) {

return Integer.parseInt(r);

} else {

log.warn("Problem", ex);

return -1;

}

});

Здесь future вырабатывает результат типа String либо ошибку, реакция переводит результат в целое число, а в случае ошибки выдает -1. Заметим, что вообще-то проверку надо начинать с if (ex!=null), так как r==null может быть как при аварийном, так и нормальном завершении, но в данном примере случай r==null рассматривается как ошибка.