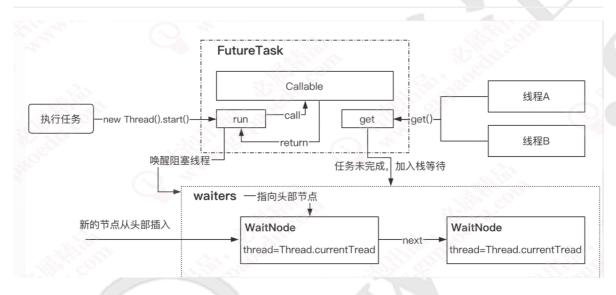
Future/Callable

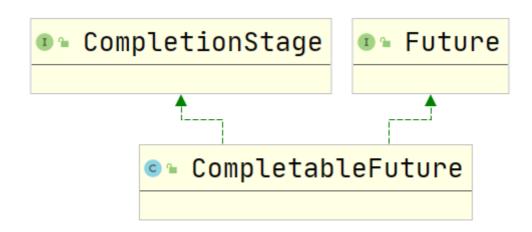
JAVA5

提供了一个带有返回值的线程。

Future/Callable 应该如何实现?



CompletableFuture



- 通过Future同步等待执行结果
- CompletionStage,增强异步回调的功能。

构建方法

构建一个CompletableFuture

- supplyAsync 异步执行一个任务,提供返回值
- supplyAsync(runnable,Executor executor) 提供返回值
- runAsync(runnable,Executor executor) -> 异步执行一个任务,但是可以自定义线程池,没有返回值
- runAsync(runnable) -> 异步执行一个任务, 默认用ForkJoinPool.commonPool();, 没有返回值

CompletionStage

• 纯消费类型的方法

纯消费类型的方法,指依赖上一个异步任务的结果作为当前函数的参数进行下一步计算,它的特点是不返回新的计算值,这类的方法都包含 Accept 这个关键字。在CompletionStage中包含9个 Accept关键字的方法,这9个方法又可以分为三类:依赖单个CompletionStage任务完成,依赖两个CompletionStage任务都完成,依赖两个CompletionStage中的任何一个完成。

```
//当前线程同步执行
public CompletionStage<Void> thenAccept(Consumer<? super T> action);
//使用ForkJoinPool.commonPool线程池执行action
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action);
//使用自定义线程池执行action
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T>
action,Executor executor);
public <U> CompletionStage<Void> thenAcceptBoth(CompletionStage<? extends U>
other,BiConsumer<? super T, ? super U> action);
public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<?</pre>
extends U> other,BiConsumer<? super T, ? super U> action);
public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<?</pre>
extends U> other,BiConsumer<? super T, ? super U> action,Executor executor);
public CompletionStage<Void> acceptEither(CompletionStage<? extends T>
other,Consumer<? super T> action);
public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T>
other,Consumer<? super T> action);
public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T>
other,Consumer<? super T> action,Executor executor);
```

• 有返回值类型的方法

有返回值类型的方法,就是用上一个异步任务的执行结果进行下一步计算,并且会产生一个新的有返回值的CompletionStage对象。

在CompletionStage中,定义了9个带有返回结果的方法,同样也可以分为三个类型:依赖单个CompletionStage任务完成,依赖两个CompletionStage任务都完成,依赖两个CompletionStage中的任何一个完成。

```
public <U> CompletionStage<U> thenApply(Function<? super T,? extends U> fn);
public <U> CompletionStage<U> thenApplyAsync(Function<? super T,? extends U>
fn);
public <U> CompletionStage<U> thenApplyAsync(Function<? super T,? extends U>
fn,Executor executor);
public <U,V> CompletionStage<V> thenCombine(CompletionStage<? extends U>
other, BiFunction<? super T,? super U,? extends V> fn);
public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends</pre>
U> other,BiFunction<? super T,? super U,? extends V> fn);
public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends</pre>
U> other,BiFunction<? super T,? super U,? extends V> fn,Executor executor);
public <U> CompletionStage<U> applyToEither(CompletionStage<? extends T>
other, Function<? super T, U> fn);
public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends</pre>
T> other, Function<? super T, U> fn);
public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends</pre>
T> other,Function<? super T, U> fn,Executor executor);
```

```
public CompletionStage<Void> thenRun(Runnable action);
public CompletionStage<Void> thenRunAsync(Runnable action);
public CompletionStage<Void> thenRunAsync(Runnable action, Executor
executor);
public CompletionStage<Void> runAfterBoth(CompletionStage<?> other,Runnable
action);
public CompletionStage<Void> runAfterBothAsync(CompletionStage<?>
other, Runnable action);
public CompletionStage<Void> runAfterBothAsync(CompletionStage<?>
other, Runnable action, Executor executor);
public CompletionStage<Void> runAfterEither(CompletionStage<?>
other, Runnable action);
public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?>
other, Runnable action);
public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?>
other, Runnable action, Executor executor);
```

多任务组合

```
public <U> CompletionStage<U> thenCompose(Function<? super T, ? extends
CompletionStage<U>> fn);
public <U> CompletionStage<U> thenComposeAsync(Function<? super T, ? extends
CompletionStage<U>> fn);
public <U> CompletionStage<U> thenComposeAsync(Function<? super T, ? extends
CompletionStage<U>> fn, Executor executor);
```

异常处理

whenComplete

whenComplete表示当任务执行完成后,会触发的方法,它的特点是,不论前置的 CompletionStage任务是正常执行结束还是出现异常,都能够触发特定的 action 方法,主要方法 如下。

handle

handle表示前置任务执行完成后,不管前置任务执行状态是正常还是异常,都会执行handle中的 fn 函数,它和whenComplete的作用几乎一致,不同点在于,handle是一个有返回值类型的方法。

exceptionally

exceptionally接受一个 fn 函数,当上一个CompletionStage出现异常时,会把该异常作为参数传递到 fn 函数

原理分析

基于Treiber stack结构,实现任务的存储。