链式操作

同步或异步

要同步链操作，通常一个接一个地编写语句，使用前面语句的结果来执行新语句:

```
// Build an animate action.
Animate animate = ...;
// Run the animate action synchronously.
animate.run();
// Build a listen action.
Listen listen = ...;
// Run the listen action synchronously.
listen.run();
```

在这里，我们运行一个 Animate 然后运行 Listen，此时均是同步的方式

这些操作也可以异步执行:

```
// Run the animate action asynchronously.
animate.async().run();
```

```
// Run the listen action asynchronously.
listen.async().run();
```

这些代码段执行异步操作，但它们没有链在一起：两个actions都将同时启动.

我们想要的是在第一个操作提供返回结果时执行第二个操作（这里是当动animation结束后listen）.我们将在下面看到如何异步链接这些操作.

Futures

什么是一个 future

`Future` 包装异步操作的对象.

它允许您在顺序方式编写代码时执行异步操作.

`Future` 类主要是用于异步执行以下操作:

- 创建actions和资源等对象,
- 处理action执行（结果，取消和错误),
- 链一些action执行资源创建.

future的状态

一个 `Future` 有3种状态：它可以提供结果，可以取消或者失败。这些状态对应于wrapped操作的不同最终状态.

如果一个wrapped操作:

- 返回一个值， `Future` 将在可用时提供此值,
- 被取消, `Future` 将会被取消,
- 遇到错误, `Future` 将会失败.

`Future<T>` 是一个通用类, 这里的 `T` 是他能提供的值的类型. 比如 `Future<String>` 能提供一个 `String` 类型的值.

注意

如果wrapped的操作没有返回任何东西，那么future的类型就是 `Future<Void>` . 在这种情况下，如果操作成功 `Future<Void>` 则将以结果状态结束.

`Future` 类提供的 `get` 方法可以同步访问 `Future` 的值:

```
Future<String> stringFuture = ...;
String value = stringFuture.get(); // Synchronous call to get the value.
```

调用 `get` 方法会阻塞当前线程：此调用是同步的.

提醒

同步调用会阻塞，而异步调用则立即返回值.

如果在wrapped作中发生错误, `get` 调用将抛出一个 `ExecutionException` 如果操作被取消，则 `get` 调用将抛出一个 `CancellationException` .

使用一个 `try-catch` 来确定 `Future` 完成的状态:

```
Future<String> stringFuture = ...;
try {
    String value = stringFuture.get();
    // The future finished with value.
} catch (ExecutionException e) {
    // The future finished with error.
} catch (CancellationException e) {
    // The future was cancelled.
}
```

操作（Operators）

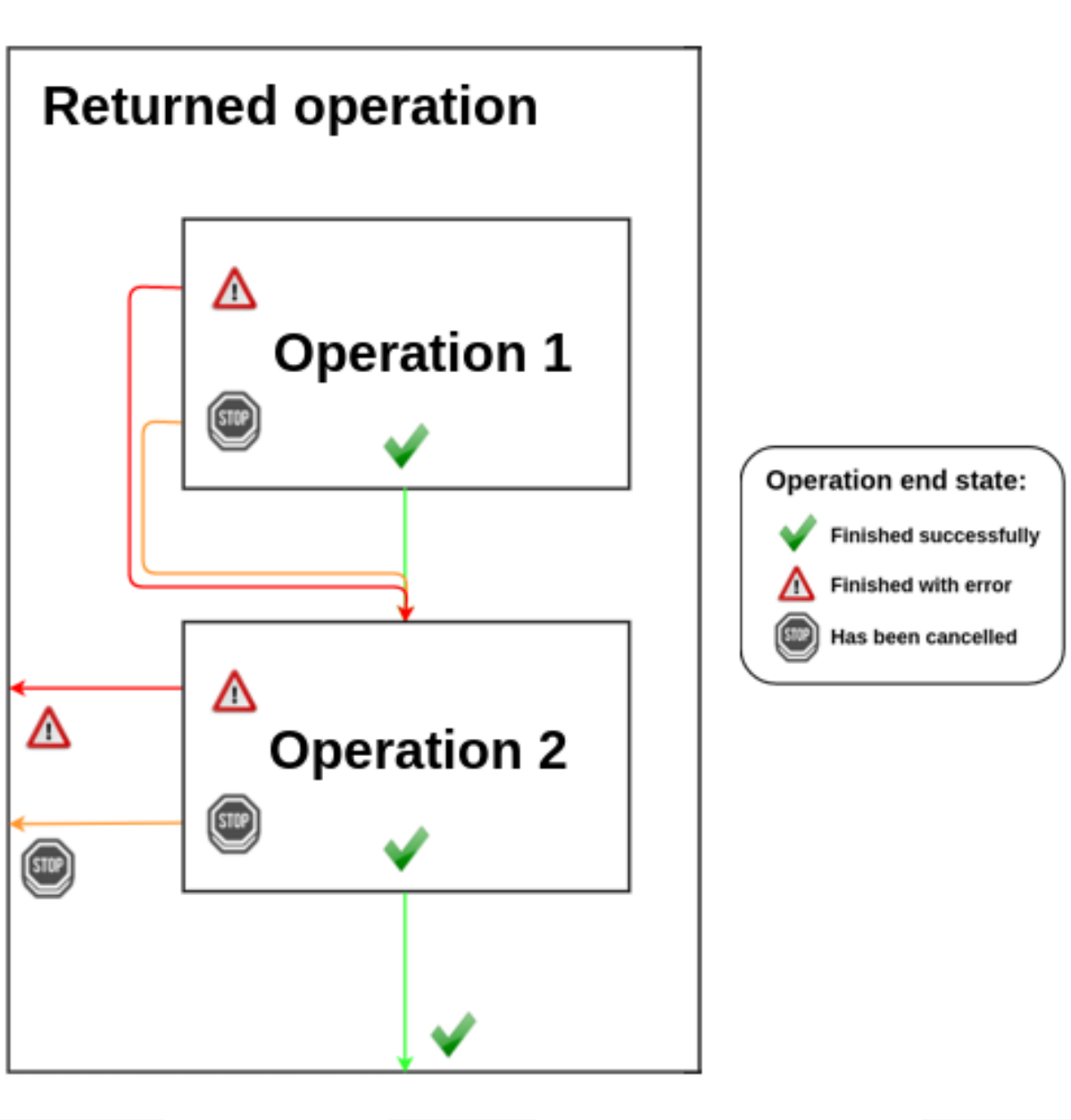
`Future` 类包含若干方法，让您链起几个异步操作:

- `thenConsume` ,
- `thenApply` ,
- `thenCompose` ,
- `andThenConsume` ,
- `andThenApply` ,
- `andThenCompose` .

`then` / `andThen`

和 “then…”:

`then...` 方法允许无论第一个操作结束状态如何，都让您链接2个操作。当您想要链相互独立的操作时，它们非常有用.

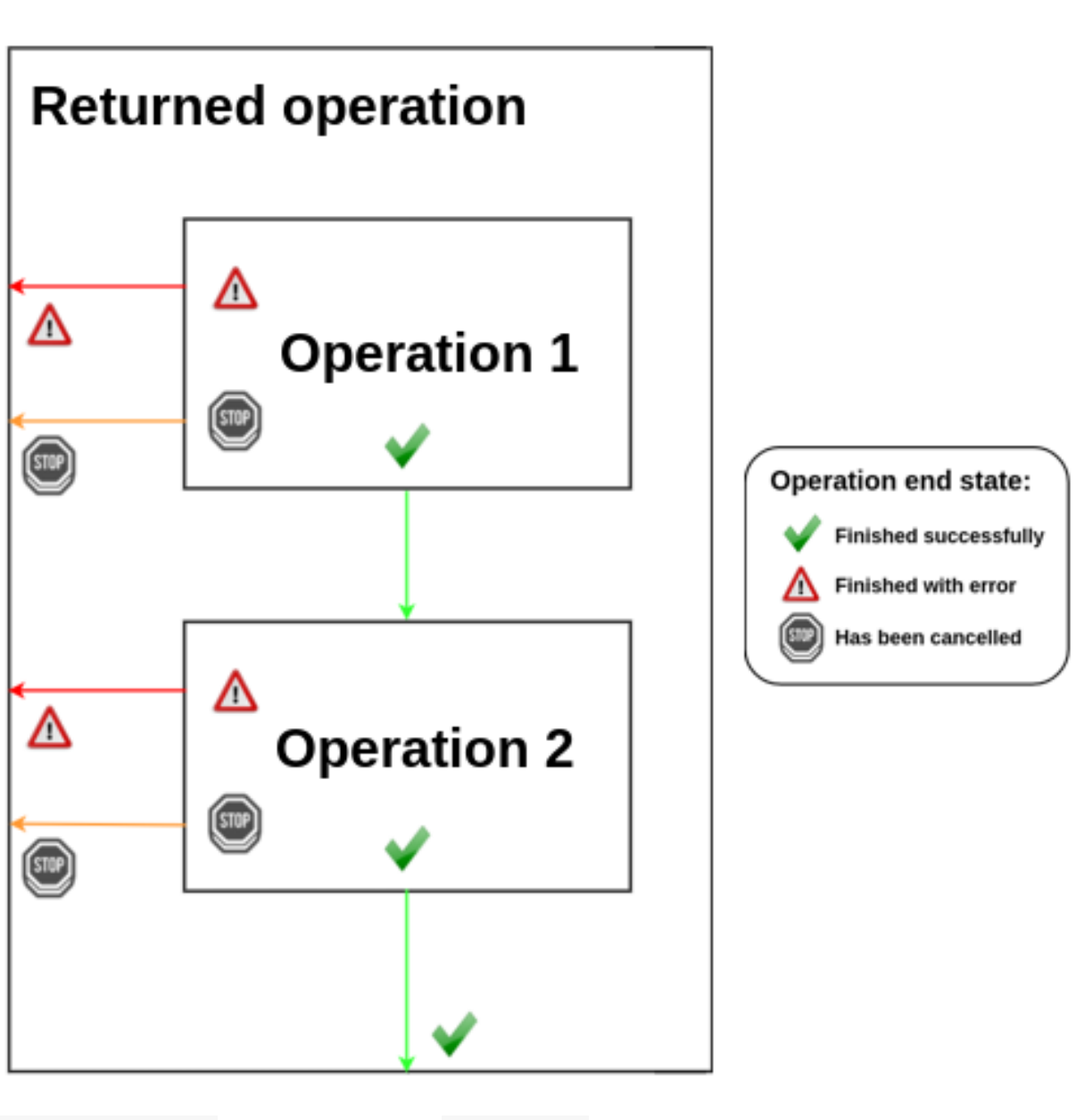


`then...` 使您可以访问 `Future` 包装的第一个操作. 使用 `isSuccess` , `hasError` 和 `isCancelled` 方法来确定 `Future` 的状态:

```
Future<String> future = ...;
future.thenConsume(stringFuture -> {
    if (stringFuture.isSuccess()) {
        // Handle success state.
        // Access the value with stringFuture.get().
    } else if (stringFuture.isCancelled()) {
        // Handle cancelled state.
    } else {
        // Handle error state.
        // Access the error with stringFuture.getError().
    }
});
```

和“andThen…”:

`andThen...` 当且仅当第一个以结果状态结束时，该方法允许您链2个操作。当某些操作取决于链中的前一个操作时，它们会很有用.



`andThen...` 直接让您访问 `Future` 包装的第一个操作的值:

```
Future<String> future = ...;
future.andThenConsume(string -> {
    // Here you have access to the String.
});
```

建议

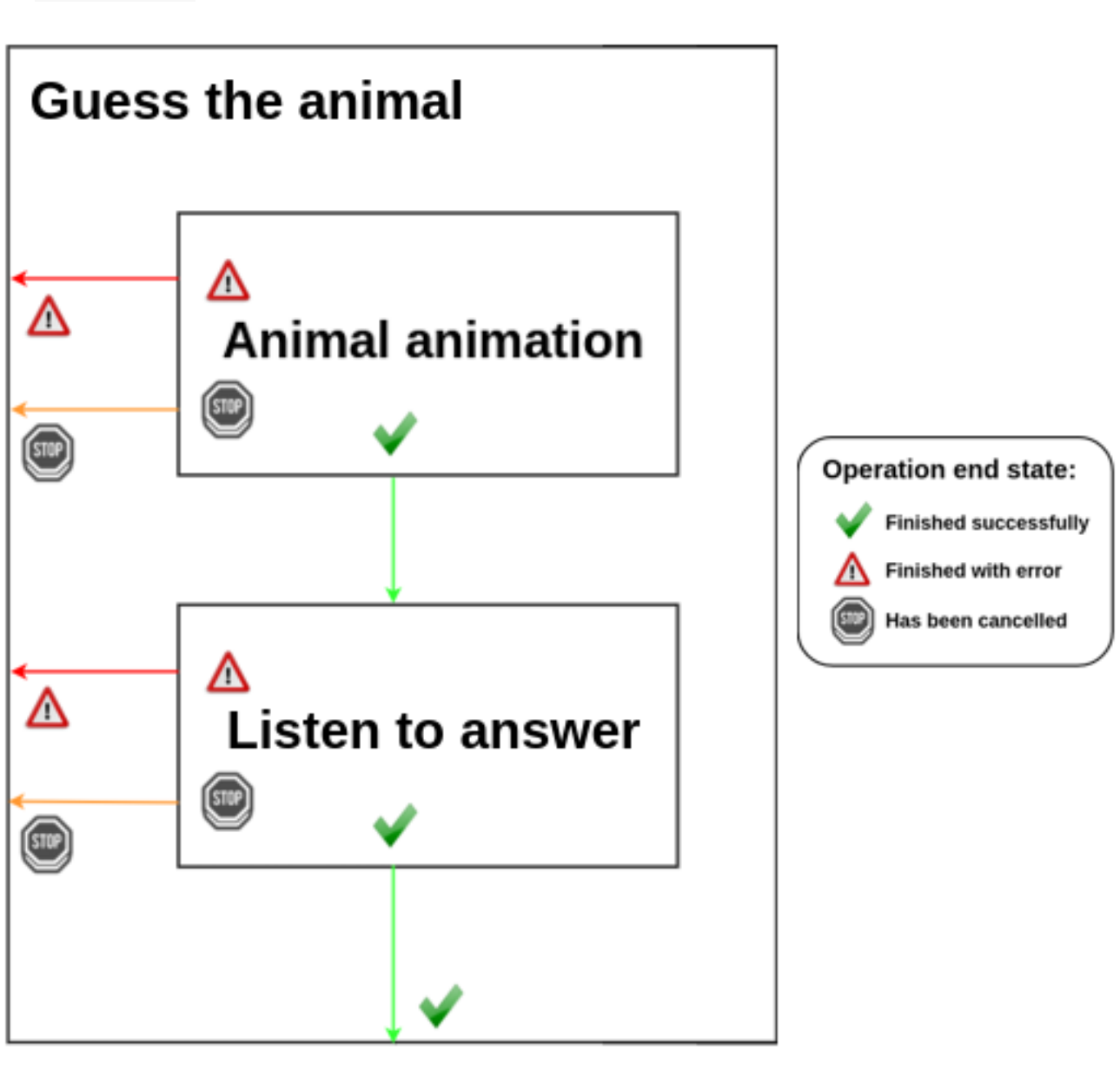
可以将 `andThen...` 视为布尔 `&` 运算符: 如果左侧为假，则不用看右侧.

Returned future:

返回的 `Future` 封装的操作等价于连续的2个操作.

“then…” 例子:

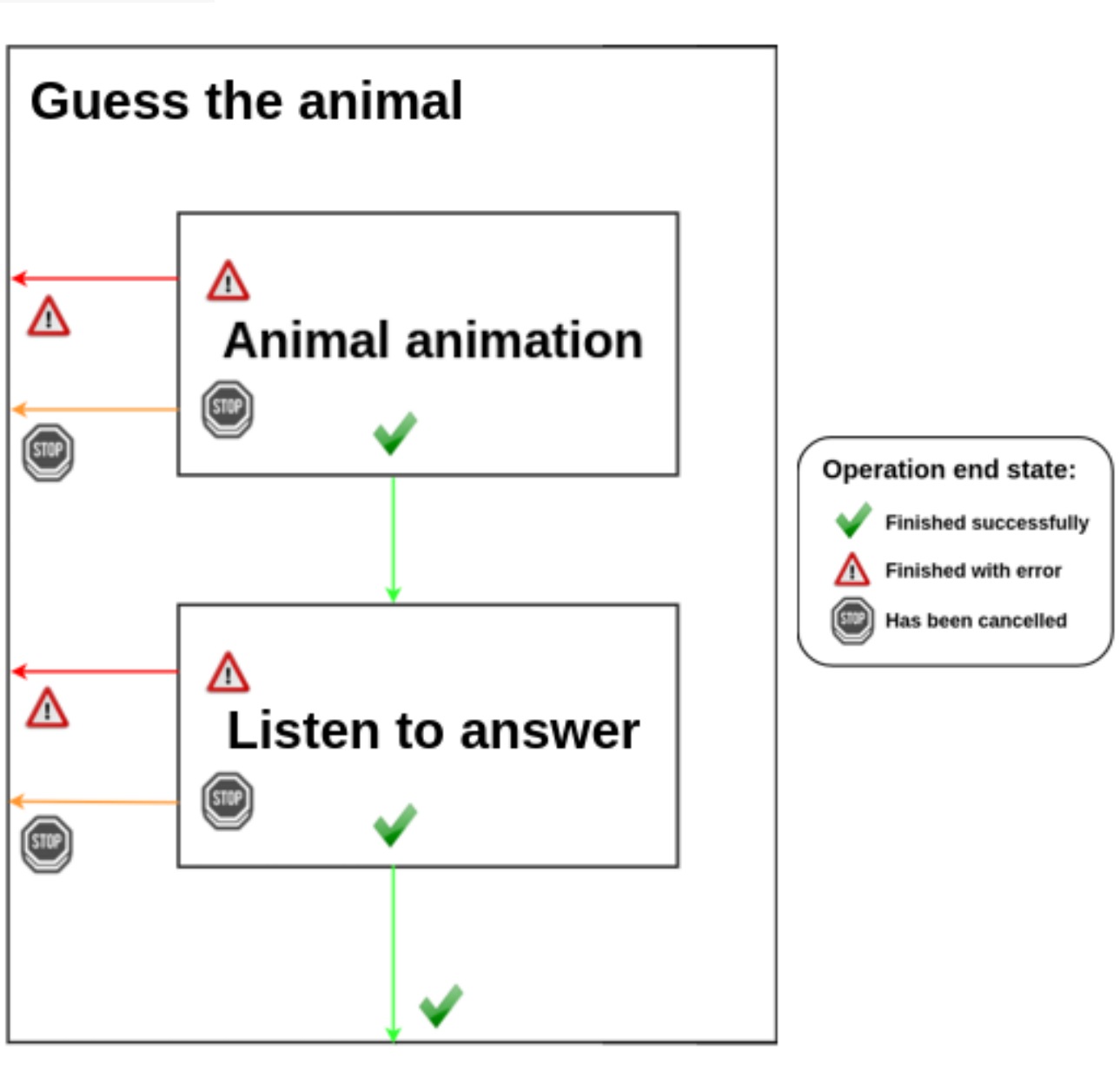
我们希望Pepper执行由几个animations组成的舞蹈。animations必须链在一起，即使一个animation失败，舞蹈也要继续：我们使用 `then...` .



```
// Build an animate action.
Animate firstAnimate = ...;
// Run the animate action asynchronously.
Future<Void> firstDance = firstAnimate.async().run();
// Chain several animations.
Future<Void> ballet = firstDance.thenCompose(dance -> {
    // Build an animate action.
    Animate secondAnimate = ...;
    // Run the animate action asynchronously.
    Future<Void> secondDance = secondAnimate.async().run();
    return secondDance;
});
```

“andThen…” 例子:

我们希望Pepper模仿动物，然后让某人猜出这种动物。当且仅当animation成功时，Pepper才听取答案：这时我们使用 `andThen...`



```
// Build an animate action.
Animate animate = ...;
// Run the animate action asynchronously.
Future<Void> imitation = animate.async().run();
// Chain the animation with a listen action.
Future<ListenResult> guess = imitation.andThenCompose(animateFuture -> {
    // Build a listen action.
    Listen listen = ...;
    // Run the listen action asynchronously.
    Future<ListenResult> answerListening = listen.async().run();
    return answerListening;
});
```

`consume` / `apply` / `compose`

根据第二个操作返回的结果，您将使用以下方法中的一个:

- `...Consume` ,
- `...Apply` ,
- `...Compose` .

consume:

用 `...Consume` 消耗掉 `Future` 并不返回任何值:

```
Future<String> future = ...;
Future<Void> returnedOperation = future.andThenConsume(string -> Log.i(TAG, "Success: " + st
```

apply:

用 `...Apply` 转化 `Future` 的值并返回特殊的类型:

```
Future<String> future = ...;
Future<Integer> returnedOperation = future.andThenApply(string -> string.length());
```

compose:

使用 `...Compose` 返回一个 `Future` . 这主要用于链actions:

```
Say say = ...;
Future<Void> sayFuture = say.async().run();
Future<Void> returnedOperation = sayFuture.andThenCompose(ignore -> {
    Animate animate = ...;
    Future<Void> animateFuture = animate.async().run();
    return animateFuture;
});
```

操作的状态:

lambda表示的操作具有不同的状态，具体取决于在lambda内部完成的操作:

Operation	State	
	If lambda throws	**Else**
Consume	Error	Success
Apply	Error	Success
Compose	Error	Inner `Future` ’s state*

内部的 `Future` 是在使用 `...Compose` lambda返回的

回调 Callbacks

每个链操作都将不同的回调作为参数:

Operator	andThen…	then…
…Consume	`Consumer<T>`	`Consumer<Future<T>>`
…Apply	`Function<T, R>`	`Function<Future<T>, R>`
…Compose	`Function<T, Future<R>>`	`Function<Future<T>, Future<R>>`

Consumer

`Consumer` 接口包含 `consume` 方法. 他用于消耗 `Future` .

注意

`consume` 作为 `Consumer` 的方法，其接口在**后台线程**上执行.

例子:

记录 `Future` 的结果:

```
// Java 8:
Future<String> future = ...;
future.andThenConsume(string -> Log.i(TAG, "Success: " + string));
```

```
// Java 7:
Future<String> future = ...;
future.andThenConsume(new Consumer<String>() {
    @Override
    public void consume(String string) throws Throwable {
        Log.i(TAG, "Success: " + string);
    }
});
```

Function

`Function` 接口包含 `execute` 方法. 它用于返回一个新的值/ `Future` .

注意

`Function` 的 `execute` 方法,其接口是在 **后台线程中**运行.

例子:

变换 `Future` 的结果:

```
// Java 8:
Future<String> future = ...;
future.andThenApply(string -> string.length());
```

```
// Java 7:
Future<String> future = ...;
future.andThenApply(new Function<String, Integer>() {
    @Override
    public Integer execute(String string) throws Throwable {
        return string.length();
    }
});
```

链接actions:

```
// Java 8:
Animate animate = ...;
animate.async().run().andThenCompose(ignore -> {
    Say say = ...;
    return say.async().run();
});
```

```
// Java 7:
Animate animate = ...;
animate.async().run().andThenCompose(new Function<Void, Future<Void>>() {
    @Override
    public Future<Void> execute(Void ignore) throws Throwable {
        Say say = ...;
        return say.async().run();
    }
});
```

See also

javadoc