在Pepper上运行actions

Actions 是您将用于为Pepper创建应用程序的主要组件.通过actions, Pepper可以:

说话

听

移动

执行动画

吸引人类

聊天

视角调整

环境中定位

照相

预备知识

为了理解如何使用actions，您应该熟悉异步性.

创建 action

使用构建器创建同步或异步的actions.

以下是构建操作的步骤:

- 使用 `QiContext` ，创建一个action构建器，
- 将action参数传递给构建器,
- 调用 `build` 或 `buildAsync` 方法来创建action.

同步

调用在builder上的 `build` 方法来创建一个同步action :

```
// Build an action synchronously.
Say say = SayBuilder.with(qiContext) // Create a builder with the QiContext.
                    .withText("Hello!") // Specify the action parameters.
                    .build();
```

警告

不要在UI线程上创建同步actions，否则会抛出一个 `NetworkOnMainThreadException` . 此机制可防止UI线程被阻塞.

异步

调用builder上 `buildAsync` 方法来创建一个异步的action :

```
// Build an action asynchronously.
Future<Say> sayActionFuture = SayBuilder.with(qiContext) // Create a builder with the QiCont
                                        .withText("Hello!") // Specify the action parameters
                                        .buildAsync();
```

引用的是action上的 `Future` 实例.

执行 action

同步

要执行 **同步action**, 要使用 `run` 方法:

```
// Execute the action synchronously.
say.run();
```

警告

不要调用 `run` 方法在 on UI 线程中否则它会抛出 `NetworkOnMainThreadException` . 此机制可防止UI线程阻塞.

异步

每个action都提供一个 `Async` 界面 允许您创建 **异步 action**. 要创建异步action, 要调用action上的 `async` 方法:

```
// Create an asynchronous action.
Say.Async sayAsync = say.async();
```

当你调用 `run` 方法时，执行异步action:

```
// Execute the asynchronous action.
sayAsync.run();
```

或者一行代码:

```
// Execute the action asynchronously.
say.async().run();
```

结果是:

- 当action运行时**线程不会被阻塞**
- 在 `run` 调用后，action的执行 **不会立刻结束.**

异步性和同步性之间的选择取决于您的用例。下面将详细介绍一些用例.

管理action执行

如上所述，可以同步或异步执行action.

因此，这允许您执行以下操作:

- 在action执行成功时执行一些代码,
- 处理action执行期间的潜在错误,
- 处理action的取消.

获得action/action成功后的操作

既然您知道如何执行actions，您将知道在执行成功时如何操作.

同步

使用 `run` 方法执行一个同步action 并获得action结果.

```
// Run the action synchronously and get the result.
Void ignore = say.run();
// Act on success.
Log.i(TAG, "Say action finished with success.");
```

注意

这里的结果类型是 `Void` 因为 Say 没有任何返回值. 其他的actions会有返回值，为之后所用.

异步

操作一个异步（asynchronous的action执行, 你需要保持一个引用reference. 对于一个异步（asynchronous）action, `run` 方法返回 reference 给你:

```
Future<Void> sayFuture = say.async().run();
```

`Future` 类提供的 `andThenConsume` 方法，允许你获得保持在 `Future` 的值并使用它:

```
// Run the action asynchronously.
Future<Void> sayFuture = say.async().run();

// Chain a lambda to the future.
sayFuture.andThenConsume(ignore -> {
    // Act on success.
    Log.i(TAG, "Say action finished with success.");
});
```

注意

如果您在UI线程上，则可能需要异步运行你的actions.例如，当您单击按钮时，您希望Pepper同时说话.

这里的lambda**不在UI线程中执行**, 因此如果要执行在lambda中的一些代码与UI交互，使用 `Qi.onUiThread` 方法.这方法简单的将lamda的执行转移到UI上.

```
sayFuture.andThenConsume(Qi.onUiThread((Consumer<Void>) ignore -> {
    Toast.makeText(MainActivity.this, "Say action finished with success.", Toast.LENGTH_SHOR
}));
```

错误处理

执行一个action时，请注意 **错误可能会产生**. 处理它的方式取决于您执行action的方式.

同步

当你在同步action上调用 `run` 方法时，可能抛出一个 `RuntimeException` . 如果这个发生在 `onRobotFocusGained` 方法时, 它的执行会停止在 `run` 方法的调用上.

用 `try-catch` 管理这个错误:

```
try {
    say.run();
    Log.i(TAG, "Success");
} catch (Exception exception) {
    Log.e(TAG, "Error", exception);
}
```

异步

如果选择异步执行action,使用 `thenConsume` 方法处理潜在错误.

`Future` 类提供的 `hasError` 方法来检查是否 `Future` 包含的错误:

```
Future<Void> sayFuture = say.async().run();
sayFuture.thenConsume(future -> {
    if (future.isSuccess()) {
        Log.i(TAG, "Success");
    } else if (future.hasError()) {
        Log.e(TAG, "Error", future.getError());
    }
});
```

取消 actions

需要取消一个action，要调用 `requestCancellation` 方法在对应的 `Future` :

```
// Execute the action asynchronously.
Future<Void> sayFuture = say.async().run();
// Cancel the action asynchronously.
sayFuture.requestCancellation();
```

如果你用 `thenConsume` 方法来链接 `Future` ,你可以查看 `Future` 的取消是否用了 `isCancelled` 方法:

```
// Execute the action asynchronously.
Future<Void> sayFuture = say.async().run();

// Chain the future with a lambda.
sayFuture.thenConsume(future -> {
    if (future.isSuccess()) {
        Log.i(TAG, "Success");
    } else if (future.isCancelled()) {
        Log.i(TAG, "Cancelled");
    }
});

// Cancel the action asynchronously.
sayFuture.requestCancellation();
```

使用多个 actions

既然您已经了解了如何操作对action的执行，我们将介绍一个常见的用例：如何一起使用多个actions.

链多个actions

同步

要同步链接动作，请一个接一个使用同步方法执行它们:

```
// Create the first action.
Say sayHello = SayBuilder.with(qiContext)
                         .withText("Hello!")
                         .build();

// Create the second action.
Say sayPepper = SayBuilder.with(qiContext)
                          .withText("I'm Pepper!")
                          .build();

// Run the first action synchronously.
sayHello.run();
// Run the second action synchronously.
sayPepper.run();
```

如果第一个action成功，则执行第二个action.

异步

`Future` 类提供两种方法来异步链多个动作: `thenCompose` 和 `andThenCompose` :

```
// Create the first action.
Say sayHello = SayBuilder.with(qiContext)
                         .withText("Hello!")
                         .build();

// Run the first action asynchronously.
Future<Void> sayHelloFuture = sayHello.async().run();

// Chain a lambda to the future.
sayHelloFuture.andThenCompose(ignore -> {
    // Create the second action.
    Say sayPepper = SayBuilder.with(qiContext)
                              .withText("I'm Pepper!")
                              .build();

    // Run the second action asynchronously.
    return sayPepper.async().run();
});
```

同步运行actions

如果您希望Pepper同时执行多个action, 请一个接一个地异步执行它们:

```
// Create the first action.
Say say = SayBuilder.with(qiContext)
                    .withText("Hello!")
                    .build();

// Create the second action.
Animate animate = AnimateBuilder.with(qiContext)
                                .withAnimation(animation)
                                .build();

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

注意

某些actions无法同时执行。例如，机器人无法同时运行多个Say 实例