包与插件

- 开发package
- 插件开发：平台通道简介
- 插件开发：实现Android端API
- 插件开发：实现IOS端API

Package

使用package可以创建共享的模块化代码。一个最小的package包括：

- 一个`pubspec.yaml`文件：声明了package的名称、版本、作者等的元数据文件。
- 一个`lib`文件夹：包括包中公开的(public)代码，最少应有一个`<package-name>.dart`文件

Flutter Packages分为两类：

- Dart包：其中一些可能包含Flutter的特定功能，因此对Flutter框架具有依赖性，这种包仅用于Flutter，例如`fluro`包。
- 插件包：一种专用的Dart包，其中包含用Dart代码编写的API，以及针对Android（使用Java或Kotlin）和针对iOS（使用OC或Swift）平台的特定实现，也就是说插件包括原生代码，一个具体的例子是`battery`插件包。

注意，虽然Flutter的Dart运行时和Dart VM运行时不是完全相同，但是如果Package中没有涉及这些存在差异的部分，那么这样的包可以同时支持Flutter和Dart VM，如Dart http网络库dio.

开发Dart包

第一步：创建Dart包

您可以通过Android Studio：File>New>New Flutter Project 来创建：

您也可以通过使用`--template=package`来执行`flutter create`命令来创建：

```
flutter create --template=package hello
```

这将在`hello/`文件夹下创建一个具有以下专用内容的package工程：

- `lib/hello.dart`： Package的Dart代码
- `test/hello_test.dart`：Package的单元测试代码。

实现package

对于纯Dart包，只需在主`lib/<package name>.dart`文件内或`lib`目录中的文件中添加功能即可 。要测试软件包，请在`test`目录中添加unit tests。下面我们看看如何组织Package包的代码，我们以shelf Package为例，它的目录结构如下：

在lib根目录下的shelf.dart中，导出了多个lib/src目录下的dart文件：

```
export 'src/cascade.dart';
export 'src/handler.dart';
export 'src/handlers/logger.dart';
export 'src/hijack_exception.dart';
export 'src/middleware.dart';
export 'src/pipeline.dart';
export 'src/request.dart';
export 'src/response.dart';
export 'src/server.dart';
export 'src/server_handler.dart';
```

而Package中主要的功能的源码都在src目录下。shelf Package也导出了一个迷你库: shelf_io，它主要是处理HttpRequest的。

导入包

当需要使用这个Package时，我们可以通过"package:"指令来指定包的入口文件：

```
import 'package:utilities/utilities.dart';
```

同一个包中的源码文件之间也可以使用相对路径来导入。

生成文档

可以使用 dartdoc 工具来为Package生成文档，开发者需要做的就是遵守文档注释语法在代码中添加文档注释，最后使用dartdoc可以直接生成API文档（一个静态网站）。文档注释是使用三斜线"///"开始，如：

```
/// The event handler responsible for updating the badge in the UI.
void updateBadge() {
  ...
}
```

详细的文档语法请参考dartdoc 。

处理包的相互依赖

如果我们正在开发一个`hello`包，它依赖于另一个包，则需要将该依赖包添加到`pubspec.yaml`文件的`dependencies`部分。 下面的代码使`url_launcher`插件的API在`hello`包中是可用的：

在`hello/pubspec.yaml`中:

```
dependencies:
  url_launcher: ^0.4.2
```

现在可以在`hello`中`import 'package:url_launcher/url_launcher.dart'`然后调用`launch()`方法了。

这与在Flutter应用程序或任何其他Dart项目中引用软件包没有什么不同。

但是，如果`hello`碰巧是一个插件包，其平台特定的代码需要访问`url_launcher`公开的特定于平台的API，那么我们还需要为特定于平台的构建文件添加合适的依赖声明，如下所示。

Android

在 `hello/android/build.gradle`:

```
android {
    // lines skipped
    dependencies {
        provided rootProject.findProject(":url_launcher")
    }
}
```

您现在可以在`hello/android/src`源码中`import io.flutter.plugins.urllauncher.UrlLauncherPlugin`访问`UrlLauncherPlugin`类。

iOS

在`hello/ios/hello.podspec`:

```
Pod::Spec.new do |s|
  # lines skipped
  s.dependency 'url_launcher'
```

您现在可以在`hello/ios/Classes`源码中`#import "UrlLauncherPlugin.h"`然后访问`UrlLauncherPlugin`类。

解决依赖冲突

假设我们想在我们的`hello`包中使用`some_package`和`other_package`，并且这两个包都依赖`url_launcher`，但是依赖的是`url_launcher`的不同的版本。 那我们就有潜在的冲突。避免这种情况的最好方法是在指定依赖关系时，程序包作者使用版本范围而不是特定版本。

```
dependencies:
  url_launcher: ^0.4.2    # 这样会较好, 任何0.4.x(x >= 2)都可.
  image_picker: '0.1.1'   # 不是很好, 只有0.1.1版本.
```

如果`some_package`声明了上面的依赖关系,`other_package`声明了`url_launcher`版本像'0.4.5'或'^0.4.0'，pub将能够自动解决问题。

即使`some_package`和`other_package`声明了不兼容的`url_launcher`版本，它仍然可能会和`url_launcher`以兼容的方式正常工作。 你可以通过向`hello`包的`pubspec.yaml`文件中添加依赖性覆盖声明来处理冲突，从而强制使用特定版本：

强制使用`0.4.3`版本的`url_launcher`，在`hello/pubspec.yaml`中:

```
dependencies:
  some_package:
  other_package:
dependency_overrides:
  url_launcher: '0.4.3'
```

如果冲突的依赖不是一个包，而是一个特定于Android的库，比如`guava`，那么必须将依赖重写声明添加到Gradle构建逻辑中。

强制使用`23.0`版本的`guava`库，在`hello/android/build.gradle`中：

```
configurations.all {
    resolutionStrategy {
        force 'com.google.guava:guava:23.0-android'
    }
}
```

Cocoapods目前不提供依赖覆盖功能。

发布Package

一旦实现了一个包，我们可以在Pub上发布它，这样其他开发者就可以轻松使用它。

在发布之前，检查`pubspec.yaml`、`README.md`以及`CHANGELOG.md`文件，以确保其内容的完整性和正确性。然后，运行 dry-run 命令以查看是否都准备OK了：

```
flutter packages pub publish --dry-run
```

验证无误后，我们就可以运行发布命令了：

```
flutter packages pub publish
```

> 如果你遇到包发布失败的情况，先检查是否因为众所周知的网络原因，如果是网络问题，可以使用VPN，这里需要注意的是一些代理只会代理部分APP的网络请求，如浏览器的，它们可能并不能代理dart的网络请求，所以在这种情况下，即使开了代理也依然无法连接到Pub，因此，在发布Pub包时使用全局代理或全局VPN会保险些。如果网络没有问题，以管理员权限(sudo)运行发布命令重试。

插件开发：平台通道简介

所谓“平台特定”或“特定平台”，平台指的就是指Flutter运行的平台，如Android或IOS，可以认为就是应用的原生部分。所以，平台通道正是Flutter和原生之间通信的桥梁，它也是Flutter插件的底层基础设施。

Flutter使用了一个灵活的系统，允许您调用特定平台的API，无论在Android上的Java或Kotlin代码中，还是iOS上的ObjectiveC或Swift代码中均可用。

Flutter与原生之间的通信依赖灵活的消息传递方式：

- 应用的Flutter部分通过平台通道（platform channel）将消息发送到其应用程序的所在的宿主（iOS或Android）应用（原生应用）。
- 宿主监听平台通道，并接收该消息。然后它会调用该平台的API，并将响应发送回客户端，即应用程序的Flutter部分。

平台通道

使用平台通道在Flutter(client)和原生(host)之间传递消息，如下图所示：

当在Flutter中调用原生方法时，调用信息通过平台通道传递到原生，原生收到调用信息后方可执行指定的操作，如需返回数据，则原生会将数据再通过平台通道传递给Flutter。值得注意的是消息传递是异步的，这确保了用户界面在消息传递时不会被挂起。

在客户端，MethodChannel API 可以发送与方法调用相对应的消息。 在宿主平台上，`MethodChannel` 在Android API 和 FlutterMethodChannel iOS API可以接收方法调用并返回结果。这些类可以帮助我们用很少的代码就能开发平台插件。

> **注意**: 如果需要，方法调用(消息传递)可以是反向的，即宿主作为客户端调用Dart中实现的API。 `quick_actions` 插件就是一个具体的例子。

平台通道数据类型支持

平台通道使用标准消息编/解码器对消息进行编解码，它可以高效的对消息进行二进制序列化与反序列化。由于Dart与原生平台之间数据类型有所差异，下面我们列出数据类型之间的映射关系。

Dart	Android	iOS
null	null	nil (NSNull when nested)
bool	java.lang.Boolean	NSNumber numberWithBool:
int	java.lang.Integer	NSNumber numberWithInt:
int, if 32 bits not enough	java.lang.Long	NSNumber numberWithLong:
int, if 64 bits not enough	java.math.BigInteger	FlutterStandardBigInteger
double	java.lang.Double	NSNumber numberWithDouble:
String	java.lang.String	NSString
Uint8List	byte[]	FlutterStandardTypedData typedDataWithBytes:
Int32List	int[]	FlutterStandardTypedData typedDataWithInt32:
Int64List	long[]	FlutterStandardTypedData typedDataWithInt64:
Float64List	double[]	FlutterStandardTypedData typedDataWithFloat64:
List	java.util.ArrayList	NSArray
Map	java.util.HashMap	NSDictionary

当在发送和接收值时，这些值在消息中的序列化和反序列化会自动进行。

开发Flutter插件

使用平台通道调用原生代码

下面我们通过一个获取电池电量的插件来介绍一下Flutter插件的开发流程。该插件中我们在Dart中通过`getBatteryLevel`调用Android `BatteryManager` API和iOS `device.batteryLevel` API。

创建一个新的应用程序项目

首先创建一个新的应用程序:

- 在终端中运行：`flutter create batterylevel`

默认情况下，模板支持使用Java编写Android代码，或使用Objective-C编写iOS代码。要使用Kotlin或Swift，请使用-i和/或-a标志:

- 在终端中运行: `flutter create -i swift -a kotlin batterylevel`

创建Flutter平台客户端

该应用的`State`类拥有当前的应用状态。我们需要延长这一点以保持当前的电量

首先，我们构建通道。我们使用`MethodChannel`调用一个方法来返回电池电量。

通道的客户端和宿主通过通道构造函数中传递的通道名称进行连接。单个应用中使用的所有通道名称必须是唯一的; 我们建议在通道名称前加一个唯一的“域名前缀”，例如`samples.flutter.io/battery`。

```
import 'dart:async';

import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
...
class _MyHomePageState extends State<MyHomePage> {
  static const platform = const
MethodChannel('samples.flutter.io/battery');

  // Get battery level.
}
```

接下来，我们调用通道上的方法，指定通过字符串标识符调用方法`getBatteryLevel`。 该调用可能失败(平台不支持平台API，例如在模拟器中运行时)，所以我们将invokeMethod调用包装在try-catch语句中。

我们使用返回的结果，在`setState`中来更新用户界面状态`batteryLevel`。

```
  // Get battery level.
  String _batteryLevel = 'Unknown battery level.';

  Future<Null> _getBatteryLevel() async {
    String batteryLevel;
    try {
      final int result = await
platform.invokeMethod('getBatteryLevel');
      batteryLevel = 'Battery level at $result % .';
    } on PlatformException catch (e) {
      batteryLevel = "Failed to get battery level:
'${e.message}'.";
    }

    setState(() {
      _batteryLevel = batteryLevel;
    });
  }
```

最后，我们在build创建包含一个小字体显示电池状态和一个用于刷新值的按钮的用户界面。

```
@override
Widget build(BuildContext context) {
  return new Material(
    child: new Center(
      child: new Column(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: [
          new RaisedButton(
            child: new Text('Get Battery Level'),
            onPressed: _getBatteryLevel,
          ),
          new Text(_batteryLevel),
        ],
      ),
    ),
  );
}
```

在接下来的两节中，我们会分别介绍Android和iOS端API的实现。

自定义平台通道和编解码器

除了上面提到的`MethodChannel`，还可以使用`BasicMessageChannel`，它支持使用自定义消息编解码器进行基本的异步消息传递。 此外，可以使用专门的`BinaryCodec`、`StringCodec`和`JSONMessageCodec`类，或创建自己的编解码器。

插件开发： Android端API实现

本节我们接着上一节"获取电池电量"插件的示例，来完成Android端API的实现。以下步骤是使用Java的示例，如果您更喜欢Kotlin，可以直接跳到后面Kotlin部分。

首先在Android Studio中打开您的Flutter应用的Android部分：

1. 启动 Android Studio
2. 选择 File > Open...
3. 定位到您 Flutter app目录, 然后选择里面的 `android` 文件夹，点击 OK
4. 在`java`目录下打开 `MainActivity.java`

接下来，在`onCreate`里创建MethodChannel并设置一个`MethodCallHandler`。确保使用和Flutter客户端中使用的通道名称相同的名称。

```
import io.flutter.app.FlutterActivity;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import io.flutter.plugin.common.MethodChannel.MethodCallHandler;
import io.flutter.plugin.common.MethodChannel.Result;

public class MainActivity extends FlutterActivity {
    private static final String CHANNEL =
"samples.flutter.io/battery";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        new MethodChannel(getFlutterView(),
CHANNEL).setMethodCallHandler(
          new MethodCallHandler() {
             @Override
             public void onMethodCall(MethodCall call, Result
result) {
                 // TODO
             }
          });
    }
}
```

接下来，我们添加Java代码，使用Android电池API来获取电池电量。此代码和在原生Android应用中编写的代码完全相同。

首先，添加需要导入的依赖。

```
import android.content.ContextWrapper;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.BatteryManager;
import android.os.Build.VERSION;
import android.os.Build.VERSION_CODES;
import android.os.Bundle;
```

然后，将下面的新方法添加到activity类中的，位于onCreate 方法下方：

```
private int getBatteryLevel() {
  int batteryLevel = -1;
  if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
    BatteryManager batteryManager = (BatteryManager)
getSystemService(BATTERY_SERVICE);
    batteryLevel =
batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPAC
ITY);
  } else {
    Intent intent = new ContextWrapper(getApplicationContext()).
        registerReceiver(null, new
IntentFilter(Intent.ACTION_BATTERY_CHANGED));
    batteryLevel = (intent.getIntExtra(BatteryManager.EXTRA_LEVEL,
-1) * 100) /
        intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
  }

  return batteryLevel;
}
```

最后，我们完成之前添加的`onMethodCall`方法。我们需要处理平台方法名为`getBatteryLevel`的调用消息，所以我们需要先在call参数判断调用的方法是否为`getBatteryLevel`。 这个平台方法的实现只需调用我们在前一步中编写的Android代码，并通过result参数返回成功或错误情况的响应信息。如果调用了未定义的API，我们也会通知返回：

```
@Override
public void onMethodCall(MethodCall call, Result result) {
    if (call.method.equals("getBatteryLevel")) {
        int batteryLevel = getBatteryLevel();

        if (batteryLevel != -1) {
            result.success(batteryLevel);
        } else {
            result.error("UNAVAILABLE", "Battery level not
available.", null);
        }
    } else {
        result.notImplemented();
    }
}
```

现在就可以在Android上运行该应用程序了，如果使用的是Android模拟器，则可以通过工具栏中的"..."按钮访问Extended Controls面板中的电池电量。

使用Kotlin添加Android平台特定的实现

使用Kotlin和使用Java的步骤类似，首先在Android Studio中打开您的Flutter应用的Android部分：

1. 启动 Android Studio。
2. 选择 the menu item "File > Open..."。
3. 定位到 Flutter app目录, 然后选择里面的 `android` 文件夹，点击 OK。
4. 在 `kotlin` 目录中打开 `MainActivity.kt` 。

接下来，在 `onCreate` 里创建MethodChannel并设置一个 `MethodCallHandler` 。确保使用与在Flutter客户端使用的通道名称相同。

```
import android.os.Bundle
import io.flutter.app.FlutterActivity
import io.flutter.plugin.common.MethodChannel
import io.flutter.plugins.GeneratedPluginRegistrant

class MainActivity() : FlutterActivity() {
  private val CHANNEL = "samples.flutter.io/battery"

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    GeneratedPluginRegistrant.registerWith(this)

    MethodChannel(flutterView, CHANNEL).setMethodCallHandler {
call, result ->
      // TODO
    }
  }
}
```

接下来，我们添加Kotlin代码，使用Android电池API来获取电池电量，这和原生开发是一样的。

首先，添加需要导入的依赖。

```
import android.content.Context
import android.content.ContextWrapper
import android.content.Intent
import android.content.IntentFilter
import android.os.BatteryManager
import android.os.Build.VERSION
import android.os.Build.VERSION_CODES
```

然后，将下面的新方法添加到activity类中的，位于onCreate 方法下方：

```
  private fun getBatteryLevel(): Int {
    val batteryLevel: Int
    if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
      val batteryManager =
getSystemService(Context.BATTERY_SERVICE) as BatteryManager
      batteryLevel =
batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPAC
ITY)
    } else {
      val intent =
ContextWrapper(applicationContext).registerReceiver(null,
IntentFilter(Intent.ACTION_BATTERY_CHANGED))
      batteryLevel =
intent!!.getIntExtra(BatteryManager.EXTRA_LEVEL, -1) * 100 /
intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1)
    }

    return batteryLevel
  }
```

最后，我们完成之前添加的`onMethodCall`方法。我们需要处理平台方法名为`getBatteryLevel`的调用消息，所以我们需要先在call参数判断调用的方法是否为`getBatteryLevel`。 这个平台方法的实现只需调用我们在前一步中编写的Android代码，并通过result参数返回成功或错误情况的响应信息。如果调用了未定义的API，我们也会通知返回：

```
MethodChannel(flutterView, CHANNEL).setMethodCallHandler { call,
result ->
  if (call.method == "getBatteryLevel") {
     val batteryLevel = getBatteryLevel()
     if (batteryLevel != -1) {
       result.success(batteryLevel)
     } else {
       result.error("UNAVAILABLE", "Battery level not available.",
null)
     }
  } else {
      result.notImplemented()
  }
}
```

您现在就可以在Android上运行该应用程序。如果您使用的是Android模拟器，则可以通过工具栏中的"..."按钮访问Extended Controls面板中的电池电量。

插件开发：iOS端API实现

本节我们接着之前"获取电池电量"插件的示例，来完成iOS端API的实现。以下步骤使用Objective-C，如果您更喜欢Swift，可以直接跳到后面Swift部分。

首先打开Xcode中Flutter应用程序的iOS部分:

1. 启动 Xcode
2. 选择 File > Open...
3. 定位到您 Flutter app目录, 然后选择里面的 `iOS` 文件夹，点击 OK
4. 确保Xcode项目的构建没有错误。
5. 选择 Runner > Runner ， 打开 `AppDelegate.m`

接下来，在`application didFinishLaunchingWithOptions:`方法内部创建一个`FlutterMethodChannel`，并添加一个处理方法。 确保与在Flutter客户端使用的通道名称相同。

```
#import <Flutter/Flutter.h>

@implementation AppDelegate
- (BOOL)application:(UIApplication*)application
didFinishLaunchingWithOptions:(NSDictionary*)launchOptions {
  FlutterViewController* controller =
(FlutterViewController*)self.window.rootViewController;

  FlutterMethodChannel* batteryChannel = [FlutterMethodChannel

methodChannelWithName:@"samples.flutter.io/battery"

binaryMessenger:controller];

  [batteryChannel setMethodCallHandler:^(FlutterMethodCall* call,
FlutterResult result) {
    // TODO
  }];

  return [super application:application
didFinishLaunchingWithOptions:launchOptions];
}
```

接下来，我们添加Objective-C代码，使用iOS电池API来获取电池电量，这和原生是相同的。

在`AppDelegate`类中添加以下新的方法：

```
- (int)getBatteryLevel {
  UIDevice* device = UIDevice.currentDevice;
  device.batteryMonitoringEnabled = YES;
  if (device.batteryState == UIDeviceBatteryStateUnknown) {
    return -1;
  } else {
    return (int)(device.batteryLevel * 100);
  }
}
```

最后，我们完成之前添加的`setMethodCallHandler`方法。我们需要处理的平台方法名为`getBatteryLevel`，所以我们在call参数中需要先判断是否为`getBatteryLevel`。 这个平台方法的实现只需调用我们在前一步中编写的iOS代码，并使用result参数返回成功或错误的响应。如果调用了未定义的API，我们也会通知返回：

```
[batteryChannel setMethodCallHandler:^(FlutterMethodCall* call,
FlutterResult result) {
  if ([@"getBatteryLevel" isEqualToString:call.method]) {
    int batteryLevel = [self getBatteryLevel];

    if (batteryLevel == -1) {
      result([FlutterError errorWithCode:@"UNAVAILABLE"
                                 message:@"电池信息不可用"
                                 details:nil]);
    } else {
      result(@(batteryLevel));
    }
  } else {
    result(FlutterMethodNotImplemented);
  }
}];
```

现在可以在iOS上运行该应用程序了，如果使用的是iOS模拟器，请注意，它不支持电池API，因此应用程序将显示“电池信息不可用”。

使用Swift实现iOS API

以下步骤与上面使用Objective-C相似，首先打开Xcode中Flutter应用程序的iOS部分:

1. 启动 Xcode
2. 选择 File > Open...
3. 定位到您 Flutter app目录, 然后选择里面的 `ios` 文件夹，点击 OK
4. 确保Xcode项目的构建没有错误。
5. 选择 Runner > Runner ， 然后打开 `AppDelegate.swift`

接下来，覆盖application方法并创建一个 `FlutterMethodChannel` 绑定通道名称 `samples.flutter.io/battery`：

```
@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    GeneratedPluginRegistrant.register(with: self);

    let controller : FlutterViewController =
window?.rootViewController as! FlutterViewController;
    let batteryChannel = FlutterMethodChannel.init(name:
"samples.flutter.io/battery",
                                                   binaryMessenger:
controller);
    batteryChannel.setMethodCallHandler({
      (call: FlutterMethodCall, result: FlutterResult) -> Void in
      // Handle battery messages.
    });

    return super.application(application,
didFinishLaunchingWithOptions: launchOptions);
  }
}
```

接下来，我们添加Swift代码，使用iOS电池API来获取电池电量，这和原生开发是相同的。

将以下新方法添加到`AppDelegate.swift`底部:

```
private func receiveBatteryLevel(result: FlutterResult) {
  let device = UIDevice.current;
  device.isBatteryMonitoringEnabled = true;
  if (device.batteryState == UIDeviceBatteryState.unknown) {
    result(FlutterError.init(code: "UNAVAILABLE",
                             message: "电池信息不可用",
                             details: nil));
  } else {
    result(Int(device.batteryLevel * 100));
  }
}
```

最后，我们完成之前添加的`setMethodCallHandler`方法。我们需要处理的平台方法名为`getBatteryLevel`，所以我们在call参数中需要先判断是否为`getBatteryLevel`。 这个平台方法的实现只需调用我们在前一步中编写的iOS代码，并使用result参数返回成功或错误的响应。如果调用了未定义的API，我们也会通知返回：

```
batteryChannel.setMethodCallHandler({
  (call: FlutterMethodCall, result: FlutterResult) -> Void in
  if ("getBatteryLevel" == call.method) {
    receiveBatteryLevel(result: result);
  } else {
    result(FlutterMethodNotImplemented);
  }
});
```

现在可以在iOS上运行应用程序，如果使用的是iOS模拟器，请注意，它不支持电池API，因此应用程序将显示“电池信息不可用”。