# 2.4 资源管理

Flutter APP安装包中会包含代码和 assets（资源）两部分。Assets是会打包到程序安装包中的，可在运行时访问。常见类型的assets包括静态数据（例如JSON文件）、配置文件、图标和图片（JPEG，WebP，GIF，动画WebP / GIF，PNG，BMP和WBMP）等。

## 指定 assets

和包管理一样，Flutter也使用[pubspec.yaml](https://www.dartlang.org/tools/pub/pubspec)文件来管理应用程序所需的资源，举个例子:

flutter:  
 assets:  
 - assets/my\_icon.png  
 - assets/background.png

assets指定应包含在应用程序中的文件， 每个asset都通过相对于pubspec.yaml文件所在的文件系统路径来标识自身的路径。asset的声明顺序是无关紧要的，asset的实际目录可以是任意文件夹（在本示例中是assets文件夹）。

在构建期间，Flutter将asset放置到称为 *asset bundle* 的特殊存档中，应用程序可以在运行时读取它们（但不能修改）。

## Asset 变体（variant）

构建过程支持“asset变体”的概念：不同版本的asset可能会显示在不同的上下文中。 在pubspec.yaml的assets部分中指定asset路径时，构建过程中，会在相邻子目录中查找具有相同名称的任何文件。这些文件随后会与指定的asset一起被包含在asset bundle中。

例如，如果应用程序目录中有以下文件:

* …/pubspec.yaml
* …/graphics/my\_icon.png
* …/graphics/background.png
* …/graphics/dark/background.png
* …etc.

然后pubspec.yaml文件中只需包含:

flutter:  
 assets:  
 - graphics/background.png

那么这两个graphics/background.png和graphics/dark/background.png 都将包含在您的asset bundle中。前者被认为是\_main asset\_ （主资源），后者被认为是一种变体（variant）。

在选择匹配当前设备分辨率的图片时，Flutter会使用到asset变体（见下文），将来，Flutter可能会将这种机制扩展到本地化、阅读提示等方面。

## 加载 assets

您的应用可以通过[AssetBundle](https://docs.flutter.io/flutter/services/AssetBundle-class.html)对象访问其asset 。有两种主要方法允许从Asset bundle中加载字符串或图片（二进制）文件。

### 加载文本assets

* 通过[rootBundle](https://docs.flutter.io/flutter/services/rootBundle.html) 对象加载：每个Flutter应用程序都有一个[rootBundle](https://docs.flutter.io/flutter/services/rootBundle.html)对象， 通过它可以轻松访问主资源包，直接使用package:flutter/services.dart中全局静态的rootBundle对象来加载asset即可。
* 通过 [DefaultAssetBundle](https://docs.flutter.io/flutter/widgets/DefaultAssetBundle-class.html) 加载：建议使用 [DefaultAssetBundle](https://docs.flutter.io/flutter/widgets/DefaultAssetBundle-class.html) 来获取当前BuildContext的AssetBundle。 这种方法不是使用应用程序构建的默认asset bundle，而是使父级widget在运行时动态替换的不同的AssetBundle，这对于本地化或测试场景很有用。

通常，可以使用DefaultAssetBundle.of()在应用运行时来间接加载asset（例如JSON文件），而在widget上下文之外，或其它AssetBundle句柄不可用时，可以使用rootBundle直接加载这些asset，例如：

import 'dart:async' show Future;  
import 'package:flutter/services.dart' show rootBundle;  
  
Future<String> loadAsset() async {  
 return await rootBundle.loadString('assets/config.json');  
}

### 加载图片

类似于原生开发，Flutter也可以为当前设备加载适合其分辨率的图像。

#### 声明分辨率相关的图片 assets

[AssetImage](https://docs.flutter.io/flutter/painting/AssetImage-class.html) 可以将asset的请求逻辑映射到最接近当前设备像素比例（dpi）的asset。为了使这种映射起作用，必须根据特定的目录结构来保存asset：

* …/image.png
* …/**M**x/image.png
* …/**N**x/image.png
* …etc.

其中M和N是数字标识符，对应于其中包含的图像的分辨率，也就是说，它们指定不同设备像素比例的图片。

主资源默认对应于1.0倍的分辨率图片。看一个例子：

* …/my\_icon.png
* …/2.0x/my\_icon.png
* …/3.0x/my\_icon.png

在设备像素比率为1.8的设备上，.../2.0x/my\_icon.png 将被选择。对于2.7的设备像素比率，.../3.0x/my\_icon.png将被选择。

如果未在Image widget上指定渲染图像的宽度和高度，那么Image widget将占用与主资源相同的屏幕空间大小。 也就是说，如果.../my\_icon.png是72px乘72px，那么.../3.0x/my\_icon.png应该是216px乘216px; 但如果未指定宽度和高度，它们都将渲染为72像素×72像素（以逻辑像素为单位）。

pubspec.yaml中asset部分中的每一项都应与实际文件相对应，但主资源项除外。当主资源缺少某个资源时，会按分辨率从低到高的顺序去选择 ，也就是说1x中没有的话会在2x中找，2x中还没有的话就在3x中找。

#### 加载图片

要加载图片，可以使用 [AssetImage](https://docs.flutter.io/flutter/painting/AssetImage-class.html)类。例如，我们可以从上面的asset声明中加载背景图片：

Widget build(BuildContext context) {  
 return new DecoratedBox(  
 decoration: new BoxDecoration(  
 image: new DecorationImage(  
 image: new AssetImage('graphics/background.png'),  
 ),  
 ),  
 );  
}

注意，AssetImage 并非是一个widget， 它实际上是一个ImageProvider，有些时候你可能期望直接得到一个显示图片的widget，那么你可以使用Image.asset()方法，如：

Widget build(BuildContext context) {  
 return Image.asset('graphics/background.png');  
}

使用默认的 asset bundle 加载资源时，内部会自动处理分辨率等，这些处理对开发者来说是无感知的。 (如果使用一些更低级别的类，如 [ImageStream](https://docs.flutter.io/flutter/painting/ImageStream-class.html)或 [ImageCache](https://docs.flutter.io/flutter/painting/ImageCache-class.html) 时你会注意到有与缩放相关的参数)

#### 依赖包中的资源图片

要加载依赖包中的图像，必须给AssetImage提供package参数。

例如，假设您的应用程序依赖于一个名为“my\_icons”的包，它具有如下目录结构：

* …/pubspec.yaml
* …/icons/heart.png
* …/icons/1.5x/heart.png
* …/icons/2.0x/heart.png
* …etc.

然后加载图像，使用:

new AssetImage('icons/heart.png', package: 'my\_icons')

或

new Image.asset('icons/heart.png', package: 'my\_icons')

**注意：包在使用本身的资源时也应该加上package参数来获取**。

##### 打包包中的 assets

如果在pubspec.yaml文件中声明了期望的资源，它将会打包到相应的package中。特别是，包本身使用的资源必须在pubspec.yaml中指定。

包也可以选择在其lib/文件夹中包含未在其pubspec.yaml文件中声明的资源。在这种情况下，对于要打包的图片，应用程序必须在pubspec.yaml中指定包含哪些图像。 例如，一个名为“fancy\_backgrounds”的包，可能包含以下文件：

* …/lib/backgrounds/background1.png
* …/lib/backgrounds/background2.png
* …/lib/backgrounds/background3.png

要包含第一张图像，必须在pubspec.yaml的assets部分中声明它：

flutter:  
 assets:  
 - packages/fancy\_backgrounds/backgrounds/background1.png

lib/是隐含的，所以它不应该包含在资产路径中。

### 特定平台 assets

上面的资源都是flutter应用中的，这些资源只有在Flutter框架运行之后才能使用，如果要给我们的应用设置APP图标或者添加启动图，那我们必须使用特定平台的assets。

#### 设置APP图标

更新Flutter应用程序启动图标的方式与在本机Android或iOS应用程序中更新启动图标的方式相同。

* Android
* 在Flutter项目的根目录中，导航到.../android/app/src/main/res目录，里面包含了各种资源文件夹（如mipmap-hdpi已包含占位符图像“ic\_launcher.png”，见图2-8）。 只需按照[Android开发人员指南](https://developer.android.com/guide/practices/ui_guidelines/icon_design_launcher.html#size)中的说明， 将其替换为所需的资源，并遵守每种屏幕密度（dpi）的建议图标大小标准。
* 图2-8
* **注意:** 如果您重命名.png文件，则还必须在您AndroidManifest.xml的<application>标签的android:icon属性中更新名称。
* iOS
* 在Flutter项目的根目录中，导航到.../ios/Runner。该目录中Assets.xcassets/AppIcon.appiconset已经包含占位符图片（见图2-9）， 只需将它们替换为适当大小的图片，保留原始文件名称。
* 图2-9

#### 更新启动页

图2-10

在Flutter框架加载时，Flutter会使用本地平台机制绘制启动页。此启动页将持续到Flutter渲染应用程序的第一帧时。

**注意:** 这意味着如果您不在应用程序的main()方法中调用[runApp](https://docs.flutter.io/flutter/widgets/runApp.html) 函数 （或者更具体地说，如果您不调用[window.render](https://docs.flutter.io/flutter/dart-ui/Window/render.html)去响应[window.onDrawFrame](https://docs.flutter.io/flutter/dart-ui/Window/onDrawFrame.html)）的话， 启动屏幕将永远持续显示。

##### Android

要将启动屏幕（splash screen）添加到您的Flutter应用程序， 请导航至.../android/app/src/main。在res/drawable/launch\_background.xml，通过自定义drawable来实现自定义启动界面（你也可以直接换一张图片）。

##### iOS

要将图片添加到启动屏幕（splash screen）的中心，请导航至.../ios/Runner。在Assets.xcassets/LaunchImage.imageset， 拖入图片，并命名为LaunchImage.png、LaunchImage@2x.png、LaunchImage@3x.png。 如果你使用不同的文件名，那您还必须更新同一目录中的Contents.json文件，图片的具体尺寸可以查看苹果官方的标准。

您也可以通过打开Xcode完全自定义storyboard。在Project Navigator中导航到Runner/Runner然后通过打开Assets.xcassets拖入图片，或者通过在LaunchScreen.storyboard中使用Interface Builder进行自定义，如图2-11所示。

图2-11