

起步:安装Flutter

[编辑本页](#)

[提Issue](#)

请选择您要安装Flutter的操作系统：

- WINDOWS
- MACOS
- LINUX



入门: 在Windows上搭建Flutter开发环境

[编辑本页](#) [提Issue](#)

- [使用镜像](#)
- [系统要求](#)
- [获取Flutter SDK](#)
 - [更新环境变量](#)
 - [运行 flutter doctor](#)
- [编辑器设置](#)
- [Android设置](#)
 - [安装Android Studio](#)
 - [设置您的Android设备](#)
 - [设置Android模拟器](#)
- [下一步](#)

使用镜像

由于在国内访问Flutter有时可能会受到限制, Flutter官方为中国开发者搭建了临时镜像, 大家可以将如下环境变量加入到用户环境变量中:

```
export PUB_HOSTED_URL=https://pub.flutter-io.cn
export FLUTTER_STORAGE_BASE_URL=https://storage.flutter-io.cn
```

注意: 此镜像为临时镜像, 并不能保证一直可用, 读者可以参考详情请参考 [Using Flutter in China](#) 以获得有关镜像服务器的最新动态。

系统要求

要安装并运行Flutter, 您的开发环境必须满足以下最低要求:

- 操作系统: Windows 7 或更高版本 (64-bit)
- 磁盘空间: 400 MB (不包括Android Studio的磁盘空间).
- 工具: Flutter 依赖下面这些命令行工具.
 - [Git for Windows](#) (Git命令行工具)

如果已安装Git for Windows, 请确保命令提示符或PowerShell中运行 `git` 命令, 不然在后面运行 `flutter doctor` 时将出现 `Unable to find git in your PATH` 错误, 此时需要手动添加 `C:\Program Files\Git\bin` 至 `Path` 系统环境变量中。如果是32位的Windows系统, 请将 `Program Files` 换成 `Program File (X86)`

获取Flutter SDK

1. 去flutter官网下载其最新可用的安装包, [点击下载](#);

注意, Flutter的渠道版本会不停变动, 请以Flutter官网为准。另外, 在中国大陆地区, 要想正常获取安装包列表或下载安装包, 可能需要翻墙, 读者也可以去Flutter github项目下去[下载安装包](#)。

2. 将安装包zip解压到你想安装Flutter SDK的路径 (如: `C:\src\flutter`; 注意, 不要将flutter安装到需要一些高权限的路径如 `C:\Program Files\`) 。
3. 在Flutter安装目录的 `flutter` 文件下找到 `flutter_console.bat`, 双击运行并启动flutter命令行, 接下来, 你就可以在Flutter命令行运行flutter命令了。

注意: 由于一些flutter命令需要联网获取数据, 如果您是在国内访问, 由于众所周知的原因, 直接访问很可能不会成功。上面的PUB_HOSTED_URL和FLUTTER_STORAGE_BASE_URL是google为国内开发者搭建的临时镜像。详情请参考 [Using Flutter in China](#)

上述命令为当前终端窗口临时设置PATH变量。要将Flutter永久添加到路径中, 请参阅[更新路径](#)。

要更新现有版本的Flutter, 请参阅[升级Flutter](#)。

更新环境变量

要在终端运行 `flutter` 命令, 你需要添加以下环境变量到系统PATH:

- 转到 “控制面板>用户帐户>用户帐户>更改我的环境变量”
- 在“用户变量”下检查是否有名为“Path”的条目:
 - 如果该条目存在, 追加 `flutter\bin` 的全路径, 使用 `;` 作为分隔符.
 - 如果条目不存在, 创建一个新用户变量 `Path`, 然后将 `flutter\bin` 的全路径作为它的值.
- 在“用户变量”下检查是否有名为“PUB_HOSTED_URL”和“FLUTTER_STORAGE_BASE_URL”的条目, 如果没有, 也添加它们。

重启Windows以应用此更改

运行 flutter doctor

打开一个新的命令提示符或PowerShell窗口并运行以下命令以查看是否需要安装任何依赖项来完成安装:

```
flutter doctor
```

在命令提示符或PowerShell窗口中运行此命令。目前, Flutter不支持像Git Bash这样的第三方shell。

该命令检查您的环境并在终端窗口中显示报告。Dart SDK已经在捆绑在Flutter里了, 没有必要单独安装Dart。仔细检查命令行输出以获取可能需要安装的其他软件或进一步需要执行的任务 (以粗体显示)

例如:

```
[~] Android toolchain - develop for Android devices
• Android SDK at D:\Android\sdk
  Android SDK is missing command line tools; download from https://goo.gl/XxQghQ
• Try re-installing or updating your Android SDK,
  visit https://flutter.io/setup/#android-setup for detailed instructions.
```

第一次运行一个flutter命令（如flutter doctor）时，它会下载它自己的依赖项并自行编译。以后再运行就会快得多。

以下各部分介绍如何执行这些任务并完成设置过程。你会看到在 flutter doctor 输出中，如果你选择使用IDE，我们提供了，IntelliJ IDEA，Android Studio和VS Code的插件，请参阅[编辑器设置](#) 以了解安装Flutter和Dart插件的步骤。

一旦你安装了任何缺失的依赖，再次运行 flutter doctor 命令来验证你是否已经正确地设置了。

该flutter工具使用Google Analytics匿名报告功能使用情况统计信息和基本崩溃报告。这些数据用于帮助改进Flutter工具。Analytics不是一运行或在运行涉及 flutter config 的任何命令时就发送，因此您可以在发送任何数据之前退出分析。要禁用报告，请执行 flutter config --no-analytics 并显示当前设置，然后执行 flutter config 。请参阅[Google的隐私政策](#)。

编辑器设置

使用 flutter 命令行工具，您可以使用任何编辑器来开发Flutter应用程序。输入 flutter help 在提示符下查看可用的工具。

我们建议使用我们的插件来获得丰富的IDE体验，支持编辑，运行和调试Flutter应用程序。请参阅[编辑器设置](#)了解详细步骤

Android设置

安装Android Studio

要为Android开发Flutter应用，您可以使用Mac，Windows或Linux（64位）机器。

Flutter需要安装和配置Android Studio:

1. 下载并安装 [Android Studio](#).
2. 启动Android Studio，然后执行“Android Studio安装向导”。这将安装最新的Android SDK，Android SDK平台工具和Android SDK构建工具，这是Flutter为Android开发时所必需的

设置您的Android设备

要准备在Android设备上运行并测试您的Flutter应用，您需要安装Android 4.1（API level 16）或更高版本的Android设备。

1. 在您的设备上启用 **开发人员选项** 和 **USB调试** 。详细说明可在[Android文档](#)中找到。
2. 使用USB将手机插入电脑。如果您的设备出现提示，请授权您的计算机访问您的设备。
3. 在终端中，运行 `flutter devices` 命令以验证Flutter识别您连接的Android设备。
4. 运行启动您的应用程序 `flutter run` 。

默认情况下，Flutter使用的Android SDK版本是基于你的 `adb` 工具版本。 如果您想让Flutter使用不同版本的Android SDK，则必须将该 `ANDROID_HOME` 环境变量设置为SDK安装目录。

设置Android模拟器

要准备在Android模拟器上运行并测试您的Flutter应用，请按照以下步骤操作：

1. 在您的机器上启用 [VM acceleration](#) 。
2. 启动 **Android Studio>Tools>Android>AVD Manager** 并选择 **Create Virtual Device**.
3. 选择一个设备并选择 **Next**。
4. 为要模拟的Android版本选择一个或多个系统映像，然后选择 **Next**. 建议使用 `x86` 或 `x86_64 image` 。
5. 在 **Emulated Performance**下, 选择 **Hardware - GLES 2.0** 以启用 [硬件加速](#).
6. 验证AVD配置是否正确，然后选择 **Finish**。

有关上述步骤的详细信息，请参阅 [Managing AVDs](#).

7. 在 **Android Virtual Device Manager**中, 点击工具栏的 **Run**。模拟器启动并显示所选操作系统版本或设备的启动画面。
8. 运行 `flutter run` 启动您的设备. 连接的设备名是 `Android SDK built for <platform>` ,其中 *platform* 是芯片系列, 如 `x86`.

下一步

[下一步: 配置编辑器](#)



起步: 配置编辑器

[编辑本页](#) [提Issue](#)

您可以使用任何文本编辑器与命令行工具来构建Flutter应用程序。不过，我们建议使用我们的编辑器插件之一，以获得更好的体验。通过我们的编辑器插件，您可以获得代码补全、语法高亮、**widget**编辑辅助、运行和调试支持等等。

按照下面步骤为Android Studio、IntelliJ或VS Code添加编辑器插件。如果你想使用其他的编辑器，那没关系，直接跳到 [下一步:创建并运行你的第一个应用程序](#)。

Android Studio

VS Code

Android Studio 安装

Android Studio: 为Flutter提供完整的IDE体验

安装Android Studio

- [Android Studio](#), 3.0或更高版本.

或者，您也可以使用IntelliJ：

- [IntelliJ IDEA Community](#), version 2017.1或更高版本.
- [IntelliJ IDEA Ultimate](#), version 2017.1 或更高版本.

安装Flutter和Dart插件

需要安装两个插件：

- `Flutter` 插件： 支持Flutter开发工作流 (运行、调试、热重载等).
- `Dart` 插件： 提供代码分析 (输入代码时进行验证、代码补全等).

要安装这些：

1. 启动Android Studio.
2. 打开插件首选项 (**Preferences>Plugins** on macOS, **File>Settings>Plugins** on Windows & Linux).
3. 选择 **Browse repositories...**, 选择 Flutter 插件并点击 `install` .
4. 重启Android Studio后插件生效.

下一步

让我们来体验一下Flutter：创建第一个项目，运行它，并体验“热重载”。

下一步: [体验 Flutter](#)



起步: 体验

[编辑本页](#) [提Issue](#)

本页介绍如何“试驾”Flutter: 从我们的模板创建一个新的Flutter应用程序, 运行它, 并学习如何使用Hot Reload进行更新重载

Flutter是一个灵活的工具包, 所以请首先选择您的开发工具来编写、构建和运行您的Flutter应用程序。

Android Studio

VS Code

Terminal + 编辑器

Android Studio: 为Flutter提供完整的IDE体验.

创建新应用

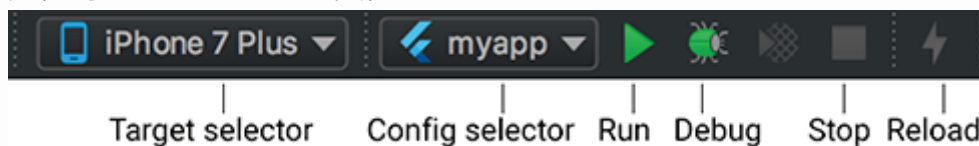
1. 选择 **File>New Flutter Project**
2. 选择 **Flutter application** 作为 project 类型, 然后点击 **Next**
3. 输入项目名称 (如 `myapp`), 然后点击 **Next**
4. 点击 **Finish**
5. 等待Android Studio安装SDK并创建项目.

上述命令创建一个Flutter项目, 项目名为myapp, 其中包含一个使用Material 组件的简单演示应用程序。

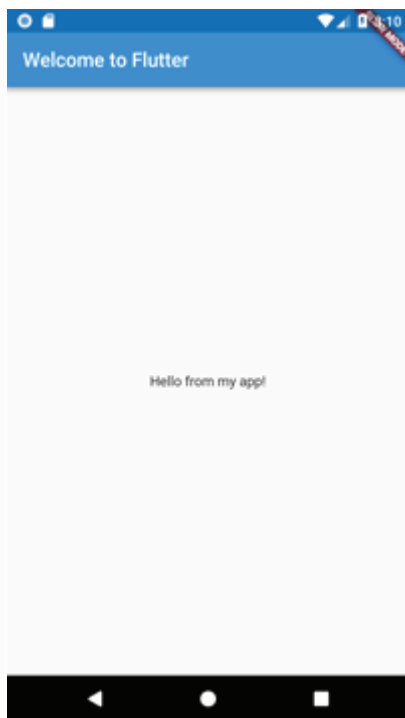
在项目目录中, 您应用程序的代码位于 `lib/main.dart` .

运行应用程序

1. 定位到Android Studio 工具栏:



2. 在 **target selector** 中, 选择一个运行该应用的Android设备. 如果没有列出可用, 请选择 **Tools>Android>AVD Manager** 并在那里创建一个
3. 在工具栏中点击 **Run**图标, 或者调用菜单项 **Run > Run**.
4. 如果一切正常, 您应该在您的设备或模拟器上看到启动的应用程序:



体验热重载

Flutter 可以通过 热重载 (*hot reload*) 实现快速的开发周期, 热重载就是无需重启应用程序就能实时加载修改后的代码, 并且不会丢失状态 (译者语: 如果是一个web开发者, 那么可以认为这和webpack的热重载是一样的)。简单的对代码进行更改, 然后告诉IDE或命令行工具你需要重新加载 (点击reload按钮), 你就会在你的设备或模拟器上看到更改。

1. 将字符串

`'You have pushed the button this many times:'` 更改为

`'You have clicked the button this many times:'`

2. 不要按“Stop”按钮; 让您的应用继续运行。

3. 要查看您的更改, 只需调用 **Save All** (`cmd-s` / `ctrl-s`), 或点击 热重载按钮 (带有闪电 图标的按钮)。

你就会立即看到更新后的字符串。

下一步

让我们通过创建一个小应用来学习一些Flutter的核心的概念。

[下一步: 编写您的第一个Flutter应用程序](#)



编写您的第一个 Flutter App

[编辑本页](#)

[提Issue](#)

这是创建您的第一个Flutter应用程序的指南。如果您熟悉面向对象和基本编程概念（如变量、循环和条件控制），则可以完成本教程，您无需要了解Dart或拥有移动开发的经验。

- [第1步: 创建 Flutter app](#)
- [第2步: 使用外部包\(package\)](#)
- [第3步: 添加一个 有状态的部件 \(Stateful widget\)](#)
- [第4步: 创建一个无限滚动ListView](#)
- [第5步: 添加交互](#)
- [第6步: 导航到新页面](#)
- [第7步: 使用主题更改UI](#)
- [做得好!](#)

你将会构建什么？

您将完成一个简单的移动应用程序，功能是：为一个创业公司生成建议的名称。用户可以选择和取消选择的名称、保存（收藏）喜欢的名称。该代码一次生成十个名称，当用户滚动时，会生成一新批名称。用户可以点击导航栏右边的列表图标，以打开到仅列出收藏名称的新页面。

这个 GIF 图展示了最终实现的效果

你会学到什么：

- Flutter应用程序的基本结构.
- 查找和使用packages来扩展功能.
- 使用热重载加快开发周期.
- 如何实现有状态的widget.
- 如何创建一个无限的、延迟加载的列表.
- 如何创建并导航到第二个页面.
- 如何使用主题更改应用程序的外观.

你会用到什么？

您需要安装以下内容：

- Flutter SDK

Flutter SDK包括Flutter的引擎、框架、widgets、工具和Dart SDK。此示例需要v0.1.4或更高版本

- **Android Studio IDE**

此示例使用的是Android Studio IDE，但您可以使用其他IDE，或者从命令行运行

- **Plugin for your IDE**

您必须为您的IDE单独安装Flutter 和 Dart插件，我们也提供了 [VS Code](#) 和 [IntelliJ](#) 的插件。

有关如何设置环境的信息，请参阅[Flutter 安装和设置](#)

第1步: 创建 Flutter app

创建一个简单的、基于模板的Flutter应用程序，按照[创建您的第一个Flutter应用](#)中的指南的步骤，然后将项目名称命名为startup_namer（而不是myapp），接下来你将会修改这个应用来完成最终的APP。

在这个示例中，你将主要编辑Dart代码所在的 **lib/main.dart** 文件，

提示: 将代码粘贴到应用中时，缩进可能会变形。您可以使用Flutter工具自动修复此问题:

- **Android Studio / IntelliJ IDEA:** 右键单击Dart代码，然后选择 **Reformat Code with dartfmt**.
- **VS Code:** 右键单击并选择 **Format Document**.
- **Terminal:** 运行 `flutter format <filename>`.

1. 替换 lib/main.dart.

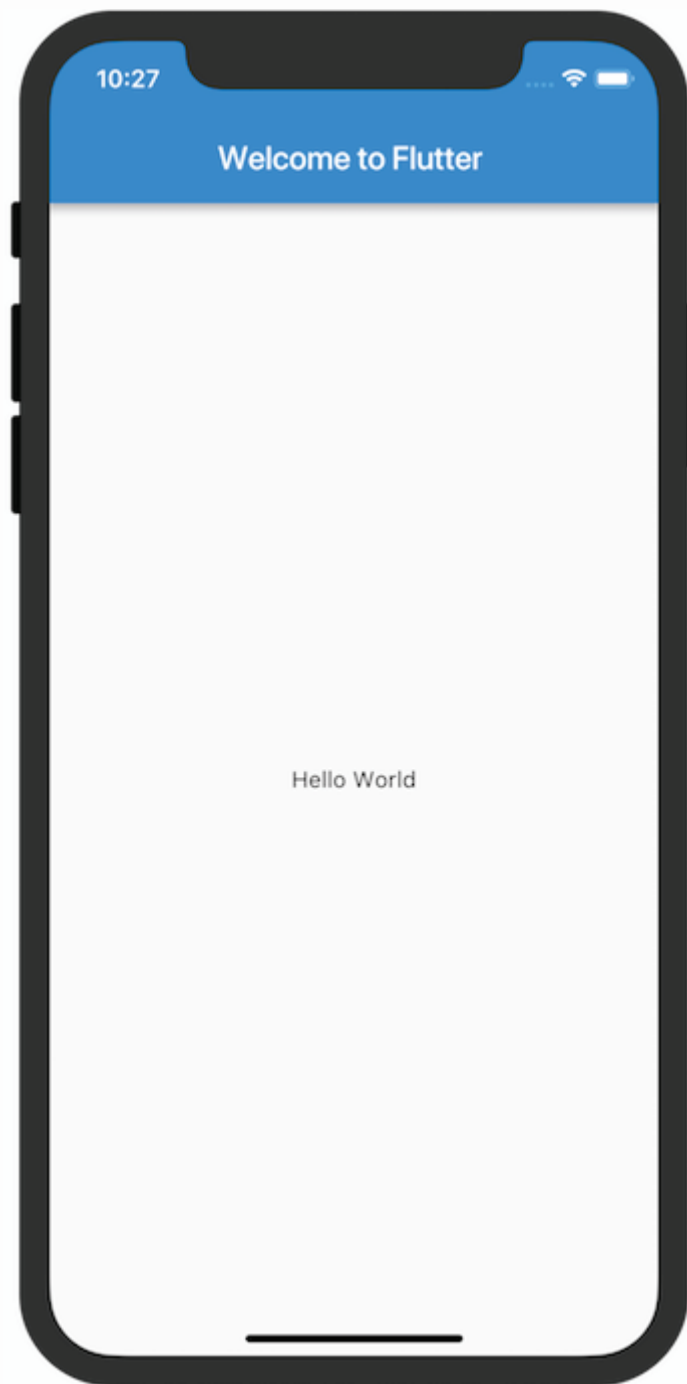
删除lib / main.dart中的所有代码，然后替换为下面的代码，它将在屏幕的中心显示“Hello World”。

```
import 'package:flutter/material.dart';

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Welcome to Flutter',
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text('Welcome to Flutter'),
        ),
        body: new Center(
          child: new Text('Hello World'),
        ),
      ),
    );
  }
}
```

2. 运行应用程序，你应该看到如下界面.



分析

- 本示例创建一个Material APP。**Material**是一种标准的移动端和web端的视觉设计语言。Flutter提供了一套丰富的Material widgets。
- main函数使用了(`=>`)符号, 这是Dart中单行函数或方法的简写。
- 该应用程序继承了 `StatelessWidget`, 这将会使应用本身也成为一個widget。在Flutter中, 大多数东西都是widget, 包括对齐(alignment)、填充(padding)和布局(layout)
- Scaffold 是 Material library 中提供的一个widget, 它提供了默认的导航栏、标题和包含主屏幕widget树的body属性。widget树可以很复杂。
- widget的主要工作是提供一个build()方法来描述如何根据其他较低级别的widget来显示自己。

本示例中的body的widget树中包含了一个Center widget, Center widget又包含一个 Text 子widget。 Center widget可以将其子widget树对其到屏幕中心。

第2步: 使用外部包(package)

在这一步中, 您将开始使用一个名为english_words的开源软件包, 其中包含数千个最常用的英文单词以及一些实用功能.

您可以在pub.dartlang.org上找到english_words软件包以及其他许多开源软件包

1. pubspec文件管理Flutter应用程序的assets(资源, 如图片、package等)。在pubspec.yaml中, 将english_words (3.1.0或更高版本) 添加到依赖项列表, 如下面高亮显示的行:

```
dependencies:  
  flutter:  
    sdk: flutter  
  
  cupertino_icons: ^0.1.0  
  english_words: ^3.1.0
```

2. 在Android Studio的编辑器视图中查看pubspec时, 单击右上角的 **Packages get**, 这会将依赖包安装到您的项目。您可以在控制台中看到以下内容:

```
flutter packages get  
Running "flutter packages get" in startup_namer...  
Process finished with exit code 0
```

3. 在 lib/main.dart 中, 引入 english_words, 如高亮显示的行所示:

```
import 'package:flutter/material.dart';  
import 'package:english_words/english_words.dart';
```

在您输入时, Android Studio会为您提供有关库导入的建议。然后它将呈现灰色的导入字符串, 让您知道导入的库尚未使用 (到目前为止)

4. 使用 English words 包生成文本来替换字符串“Hello World”.

Tip: “驼峰命名法” (称为 “upper camel case” 或 “Pascal case”), 表示字符串中的每个单词 (包括第一个单词) 都以大写字母开头。所以, “uppercamelcase” 变成 “UpperCamelCase”

进行以下更改, 如高亮部分所示:

```
import 'package:flutter/material.dart';
import 'package:english_words/english_words.dart';

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final wordPair = new WordPair.random();
    return new MaterialApp(
      title: 'Welcome to Flutter',
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text('Welcome to Flutter'),
        ),
        body: new Center(
          //child: new Text('Hello World'),
          child: new Text(wordPair.asPascalCase),
        ),
      ),
    );
  }
}
```

5. 如果应用程序正在运行，请使用热重载按钮 (⚡) 更新正在运行的应用程序。每次单击热重载或保存项目时，都会在正在运行的应用程序中随机选择不同的单词对。这是因为单词对是在 `build` 方法内部生成的。每次 `MaterialApp` 需要渲染时或者在 `Flutter Inspector` 中切换平台时 `build` 都会运行。



遇到问题？

如果您的应用程序运行不正常，请查找是否有拼写错误。如果需要，使用下面链接中的代码来对比更正。

- [pubspec.yaml](#) (The **pubspec.yaml** file won't change again.)
- [lib/main.dart](#)

第3步: 添加一个 有状态的部件 (Stateful widget)

Stateless widgets 是不可变的, 这意味着它们的属性不能改变 - 所有的值都是最终的.

Stateful widgets 持有的状态可能在**widget**生命周期中发生变化. 实现一个 **stateful widget** 至少需要两个类:

1. 一个 **StatefulWidget**类。

2. 一个 **State**类。 **StatefulWidget**类本身是不变的，但是 **State**类在**widget**生命周期中始终存在。

在这一步中，您将添加一个有状态的**widget-RandomWords**，它创建其**State**类**RandomWordsState**。**State**类将最终为**widget**维护建议的和喜欢的单词对。

1. 添加有状态的 **RandomWords widget** 到 **main.dart**。它也可以在**MyApp**之外的文件的任何位置使用，但是本示例将它放到了文件的底部。**RandomWords widget**除了创建**State**类之外几乎没有其他任何东西

```
class RandomWords extends StatefulWidget {
  @override
  createState() => new RandomWordsState();
}
```

2. 添加 **RandomWordsState** 类.该应用程序的大部分代码都在该类中，该类持有**RandomWords widget**的状态。这个类将保存随着用户滚动而无限增长的生成的单词对，以及喜欢的单词对，用户通过重复点击心形图标来将它们从列表中添加或删除。

你会一步一步地建立这个类。首先，通过添加高亮显示的代码创建一个最小类

```
class RandomWordsState extends State<RandomWords> {
}
```

3. 在添加状态类后，IDE会提示该类缺少**build**方法。接下来，您将添加一个基本的**build**方法，该方法通过将生成单词对的代码从**MyApp**移动到**RandomWordsState**来生成单词对。

将**build**方法添加到**RandomWordState**中，如下面高亮代码所示

```
class RandomWordsState extends State<RandomWords> {
  @override
  Widget build(BuildContext context) {
    final wordPair = new WordPair.random();
    return new Text(wordPair.asPascalCase);
  }
}
```

4. 通过下面高亮显示的代码，将生成单词对代的码从**MyApp**移动到**RandomWordsState**中

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final wordPair = new WordPair.random(); // 删除此行

    return new MaterialApp(
      title: 'Welcome to Flutter',
      home: new Scaffold(
```

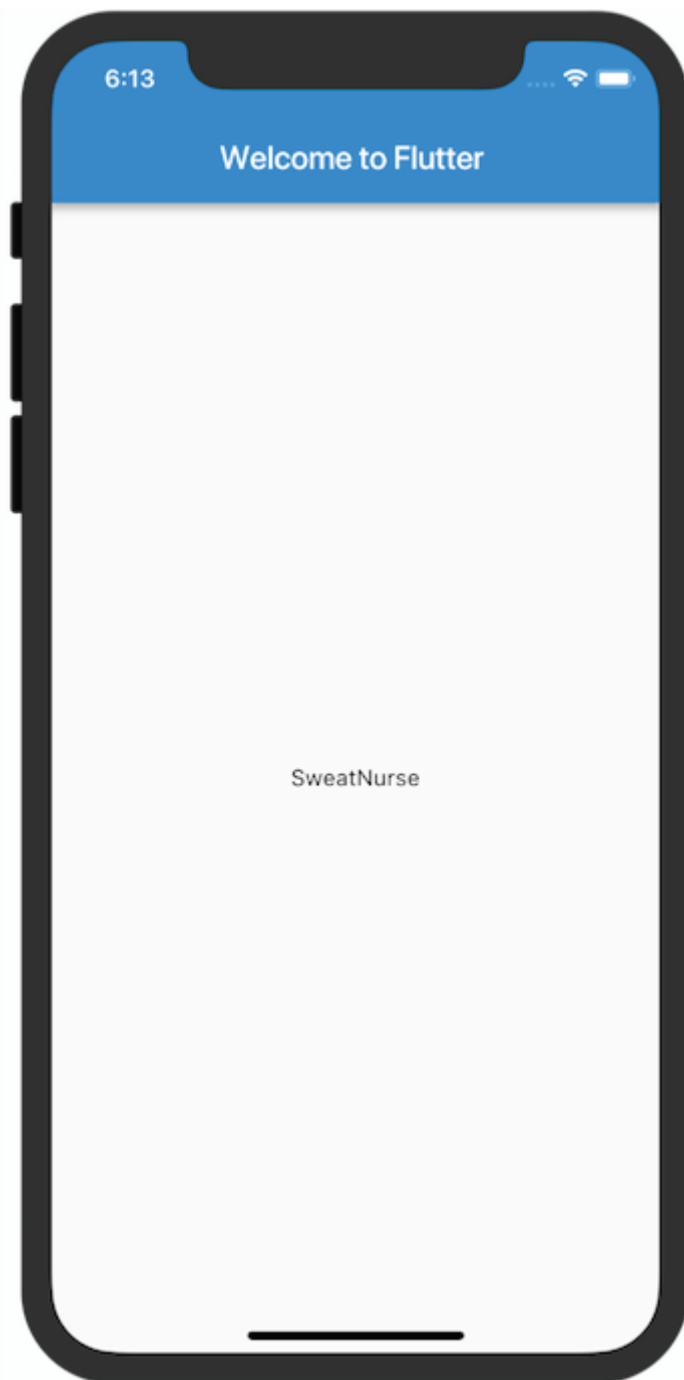
```
    appBar: new AppBar(  
      title: new Text('Welcome to Flutter'),  
    ),  
    body: new Center(  
      //child: new Text(wordPair.asPascalCase),  
      child: new RandomWords(),  
    ),  
  ),  
);  
}
```

重新启动应用程序。如果您尝试热重载，则可能会看到一条警告：

```
Reloading...  
Not all changed program elements ran during view reassembly; consider  
restarting.
```

这可能是误报，但考虑到重新启动可以确保您的更改在应用界面中生效。

应用程序应该像之前一样运行，每次热重载或保存应用程序时都会显示一个单词对。



遇到问题？

如果您的应用程序运行不正常，可以使用下面链接中的代码来对比更正。

- [lib/main.dart](#)

第4步: 创建一个无限滚动ListView

在这一步中，您将扩展（继承）`RandomWordsState`类，以生成并显示单词对列表。当用户滚动时，`ListView`中显示的列表将无限增长。`ListView`的 `builder` 工厂构造函数允许您按需建立一个懒加载的列表视图。

1. 向`RandomWordsState`类中添加一个 `_suggestions` 列表以保存建议的单词对。该变量以下划线（`_`）开

头，在Dart语言中使用下划线前缀标识符，会强制其变成私有的。

另外，添加一个 `biggerFont` 变量来增大字体大小

```
class RandomWordsState extends State<RandomWords> {
  final _suggestions = <WordPair>[];

  final _biggerFont = const TextStyle(fontSize: 18.0);
  ...
}
```

2. 向RandomWordsState类添加一个 `_buildSuggestions()` 函数. 此方法构建显示建议单词对的ListView。

ListView类提供了一个builder属性， `itemBuilder` 值是一个匿名回调函数， 接受两个参数-BuildContext和行迭代器 `i`。迭代器从0开始， 每调用一次该函数， `i` 就会自增1， 对于每个建议的单词对都会执行一次。该模型允许建议的单词对列表在用户滚动时无限增长。

添加如下高亮的行：

```
class RandomWordsState extends State<RandomWords> {
  ...
  Widget _buildSuggestions() {
    return new ListView.builder(
      padding: const EdgeInsets.all(16.0),
      // 对于每个建议的单词对都会调用一次itemBuilder，然后将单词对添加到ListTile行中
      // 在偶数行，该函数会为单词对添加一个ListTile row.
      // 在奇数行，该函数会添加一个分割线widget，来分隔相邻的词对。
      // 注意，在小屏幕上，分割线看起来可能比较吃力。
      itemBuilder: (context, i) {
        // 在每一列之前，添加一个1像素高的分隔线widget
        if (i.isOdd) return new Divider();

        // 语法 "i ~/ 2" 表示i除以2，但返回值是整形（向下取整），比如i为：1, 2, 3, 4, 5
        // 时，结果为0, 1, 1, 2, 2， 这可以计算出ListView中减去分隔线后的实际单词对数量
        final index = i ~/ 2;
        // 如果是建议列表中最后一个单词对
        if (index >= _suggestions.length) {
          // ...接着再生成10个单词对，然后添加到建议列表
          _suggestions.addAll(generateWordPairs().take(10));
        }
        return _buildRow(_suggestions[index]);
      }
    );
  }
}
```

3. 对于每一个单词对， `_buildSuggestions` 函数都会调用一次 `_buildRow`。这个函数在ListTile中显示每个新词对，这使您在下一步中可以生成更漂亮的显示行

在RandomWordsState中添加一个 `_buildRow` 函数：

```
class RandomWordsState extends State<RandomWords> {
  ...

  Widget _buildRow(WordPair pair) {
    return new ListTile(
      title: new Text(
        pair.asPascalCase,
        style: _biggerFont,
      ),
    );
  }
}
```

4. 更新RandomWordsState的build方法以使用 `_buildSuggestions()`，而不是直接调用单词生成库。更改后如下面高亮部分：

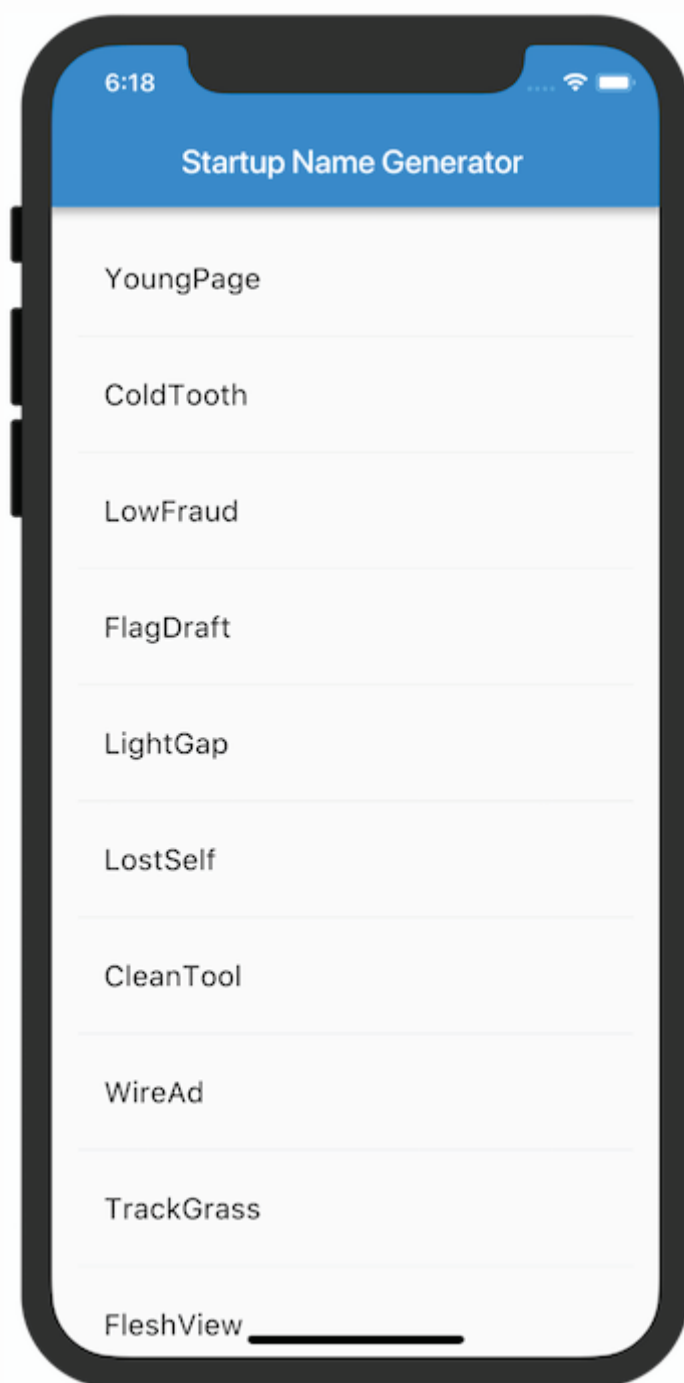
```
class RandomWordsState extends State<RandomWords> {
  ...
  @override
  Widget build(BuildContext context) {
    final wordPair = new WordPair.random(); // 删除这两行
    return new Text(wordPair.asPascalCase);
    return new Scaffold (
      appBar: new AppBar(
        title: new Text('Startup Name Generator'),
      ),
      body: _buildSuggestions(),
    );
  }
  ...
}
```

5. 更新MyApp的build方法。从MyApp中删除Scaffold和AppBar实例。这些将由RandomWordsState管理，这使得用户在下一步中从一个屏幕导航到另一个屏幕时，可以更轻松地更改导航栏中的路由名称。

用下面高亮部分替换最初的build方法：

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Startup Name Generator',
      home: new RandomWords(),
    );
  }
}
```

重新启动应用程序。你应该看到一个单词对列表。尽可能地向下滚动，您将继续看到新的单词对。



遇到问题？

如果你的应用没有正常运行，你可以使用一下链接中的代码对比更正。

- [lib/main.dart](#)

第5步: 添加交互

在这一步中，您将为每一行添加一个可点击的心形图标。当用户点击列表中的条目，切换其“收藏”状态时，将该词对添加到或移除“收藏夹”。

1. 添加一个 `_saved` **Set**(集合) 到 `RandomWordsState`。这个集合存储用户喜欢（收藏）的单词对。在这里，**Set**比**List**更合适，因为**Set**中不允许重复的值。

```
class RandomWordsState extends State<RandomWords> {
  final _suggestions = <WordPair>[];

  final _saved = new Set<WordPair>();

  final _biggerFont = const TextStyle(fontSize: 18.0);
  ...
}
```

2. 在 `_buildRow` 方法中添加 `alreadySaved` 来检查确保单词对还没有添加到收藏夹中。

```
Widget _buildRow(WordPair pair) {
  final alreadySaved = _saved.contains(pair);
  ...
}
```

3. 同时在 `_buildRow()` 中，添加一个心形 图标到 `ListTiles`以启用收藏功能。接下来，你就可以给心形图标添加交互能力了。

添加下面高亮的行：

```
Widget _buildRow(WordPair pair) {
  final alreadySaved = _saved.contains(pair);
  return new ListTile(
    title: new Text(
      pair.asPascalCase,
      style: _biggerFont,
    ),
    trailing: new Icon(
      alreadySaved ? Icons.favorite : Icons.favorite_border,
      color: alreadySaved ? Colors.red : null,
    ),
  );
}
```

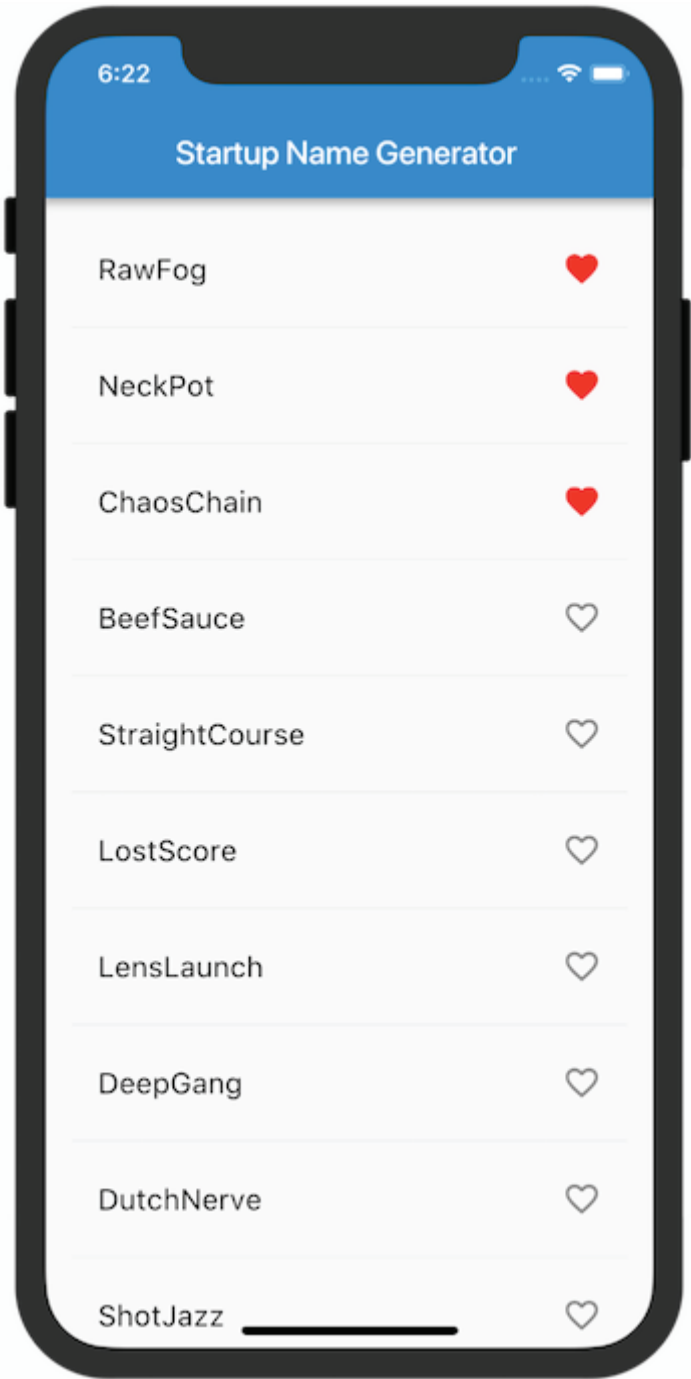
4. 重新启动应用。你现在可以在每一行看到心形 图标，但它们还没有交互。
5. 在 `_buildRow` 中让心形 图标变得可以点击。如果单词条目已经添加到收藏夹中，再次点击它将其从收藏夹中删除。当心形 图标被点击时，函数调用 `setState()` 通知框架状态已经改变。

添加如下高亮的行：

```
Widget _buildRow(WordPair pair) {  
  final alreadySaved = _saved.contains(pair);  
  return new ListTile(  
    title: new Text(  
      pair.asPascalCase,  
      style: _biggerFont,  
    ),  
    trailing: new Icon(  
      alreadySaved ? Icons.favorite : Icons.favorite_border,  
      color: alreadySaved ? Colors.red : null,  
    ),  
    onTap: () {  
      setState(() {  
        if (alreadySaved) {  
          _saved.remove(pair);  
        } else {  
          _saved.add(pair);  
        }  
      });  
    },  
  );  
}
```

提示: 在Flutter的响应式风格的框架中，调用setState() 会为State对象触发build() 方法，从而导致对UI的更新

热重载你的应用。你就可以点击任何一行收藏或删除。请注意，点击一行时会生成从心形 图标发出的水波动画



遇到了问题？

如果您的应用没有正常运行，请查看下面链接处的代码，对比更正。

- [lib/main.dart](#)

第6步: 导航到新页面

在这一步中，您将添加一个显示收藏夹内容的新页面（在Flutter中称为路由(route)）。您将学习如何在主路由和新路由之间导航（切换页面）。

在Flutter中，导航器管理应用程序的路由栈。将路由推入（push）到导航器的栈中，将会显示更新为该路由页面。从导航器的栈中弹出（pop）路由，将显示返回到前一个路由。

1. 在RandomWordsState的build方法中为AppBar添加一个列表图标。当用户点击列表图标时，包含收藏夹的新路由页面入栈显示。

提示: 某些widget属性需要单个widget (child) , 而其它一些属性, 如action, 需要一组widgets(children) , 用方括号[]表示。

将该图标及其相应的操作添加到build方法中:

```
class RandomWordsState extends State<RandomWords> {
  ...
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Startup Name Generator'),
        actions: <Widget>[
          new IconButton(icon: new Icon(Icons.list), onPressed: _pushSaved),
        ],
      ),
      body: _buildSuggestions(),
    );
  }
  ...
}
```

2. 向RandomWordsState类添加一个 _pushSaved() 方法.

```
class RandomWordsState extends State<RandomWords> {
  ...
  void _pushSaved() {
  }
}
```

热重载应用，列表图标将会出现在导航栏中。现在点击它不会有任何反应，因为 _pushSaved 函数还是空的。

3. 当用户点击导航栏中的列表图标时，建立一个路由并将其推入到导航管理器栈中。此操作会切换页面以显示新路由。

新页面的内容在在MaterialPageRoute的 builder 属性中构建， builder 是一个匿名函数。

添加Navigator.push调用，这会使路由入栈（以后路由入栈均指推入到导航管理器的栈）

```
void _pushSaved() {
  Navigator.of(context).push(
  );
}
```

}

4. 添加`MaterialPageRoute`及其`builder`。现在，添加生成`ListTile`行的代码。`ListTile`的`divideTiles()`方法在每个`ListTile`之间添加1像素的分割线。该`divided`变量持有最终的列表项。

```
void _pushSaved() {
  Navigator.of(context).push(
    new MaterialPageRoute(
      builder: (context) {
        final tiles = _saved.map(
          (pair) {
            return new ListTile(
              title: new Text(
                pair.asPascalCase,
                style: _biggerFont,
              ),
            );
          },
        );
        final divided = ListTile
          .divideTiles(
            context: context,
            tiles: tiles,
          )
          .toList();
      },
    );
}
```

5. `builder`返回一个`Scaffold`，其中包含名为“Saved Suggestions”的新路由的应用栏。新路由的`body`由包含`ListTiles`行的`ListView`组成；每行之间通过一个分隔线分隔。

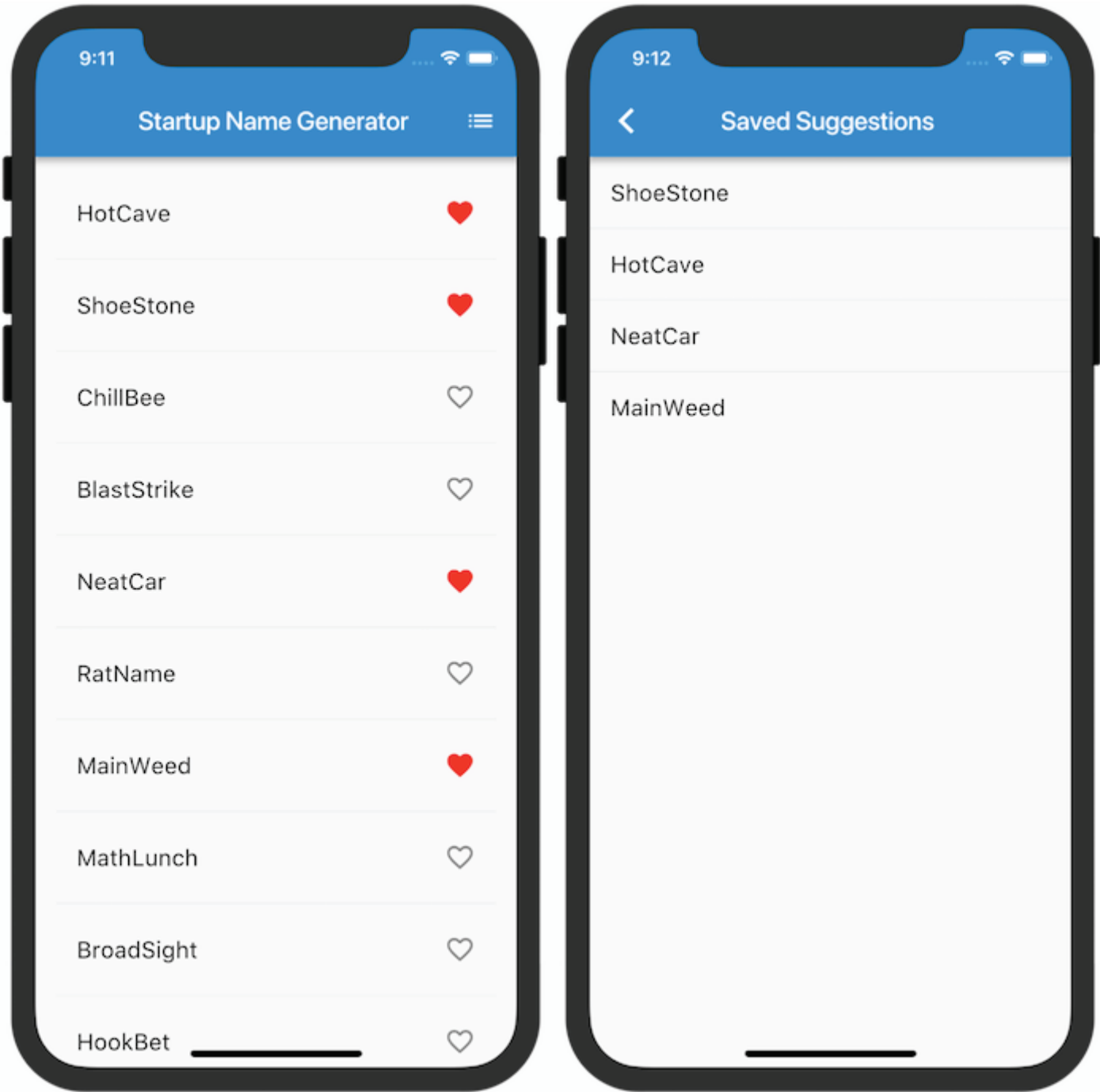
添加如下高亮的代码：

```
void _pushSaved() {
  Navigator.of(context).push(
    new MaterialPageRoute(
      builder: (context) {
        final tiles = _saved.map(
          (pair) {
            return new ListTile(
              title: new Text(
                pair.asPascalCase,
                style: _biggerFont,
              ),
            );
          },
        );
      },
    );
}
```

```
final divided = ListTile
  .divideTiles(
    context: context,
    tiles: tiles,
  )
  .toList();

return new Scaffold(
  appBar: new AppBar(
    title: new Text('Saved Suggestions'),
  ),
  body: new ListView(children: divided),
);
},
),
);
}
```

6. 热重载应用程序。收藏一些选项，并点击应用栏中的列表图标，在新路由页面中显示收藏的内容。请注意，导航器会在应用栏中添加一个“返回”按钮。你不必显式实现`Navigator.pop`。点击后退按钮返回到主页路由。



遇到了问题？

如果您的应用不能正常工作，请参考下面链接处的代码，对比并更正。

- [lib/main.dart](#)

第7步：使用主题更改UI

在这最后一步中，您将会使用主题。主题控制您应用程序的外观和风格。您可以使用默认主题，该主题取决于物理设备或模拟器，也可以自定义主题以适应您的品牌。

1. 您可以通过配置ThemeData类轻松更改应用程序的主题。 您的应用程序目前使用默认主题，下面将更改primary color颜色为白色。

通过如下高亮部分代码，将应用程序的主题更改为白色：

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Startup Name Generator',
      theme: new ThemeData(
        primaryColor: Colors.white,
      ),
      home: new RandomWords(),
    );
  }
}
```

- 2. 热重载应用。 请注意，整个背景将会变为白色，包括应用栏。
- 3. 作为读者的一个练习，使用 **ThemeData** 来改变UI的其他方面。 **Material library**中的 **Colors**类提供了许多可以使用的颜色常量， 你可以使用热重载来快速简单地尝试、实验。



遇到了问题？

如果你遇到了问题，请查看以下链接中应用程序的最终代码。

- [lib/main.dart](#)

做的好!

你已经编写了一个可以在iOS和Android上运行的交互式Flutter应用程序。在这个例子中，你已经做了下面这些事：

- 从头开始创建一个Flutter应用程序.
- 编写 Dart 代码.
- 利用外部的第三方库.
- 使用热重载加快开发周期.
- 实现一个有状态的widget，为你的应用增加交互.
- 用ListView和ListTiles创建一个延迟加载的无限滚动列表.
- 创建了一个路由并添加了在主路由和新路由之间跳转逻辑
- 了解如何使用主题更改应用UI的外观.

下一步

[下一步: 了解更多](#)



Flutter Widget框架概述

[编辑本页](#) [提Issue](#)

- [介绍](#)
- [Hello World](#)
- [基础 Widget](#)
- [使用 Material 组件](#)
- [处理手势](#)
- [根据用户输入改变widget](#)
- [整合所有](#)
- [响应widget生命周期事件](#)
- [Key](#)
- [全局 Key](#)

介绍

Flutter Widget采用现代响应式框架构建，这是从 [React](#) 中获得的灵感，中心思想是用widget构建你的UI。Widget描述了他们的视图在给定其当前配置和状态时应该看起来像什么。当widget的状态发生变化时，widget会重新构建UI，Flutter会对比前后变化的不同，以确定底层渲染树从一个状态转换到下一个状态所需的最小更改（译者语：类似于React/Vue中虚拟DOM的diff算法）。

注意: 如果您想通过代码来深入了解Flutter，请查看 [构建Flutter布局](#) 和 [为Flutter App添加交互功能](#)。

Hello World

一个最简单的Flutter应用程序，只需一个widget即可！如下面示例：将一个widget传给 `runApp` 函数即可：

```
import 'package:flutter/material.dart';

void main() {
  runApp(
    new Center(
      child: new Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```


该 `runApp` 函数接受给定的 `Widget` 并使其成为widget树的根。在此示例中，widget树由两个`widget:Center`(及其子widget)和`Text`组成。框架强制根widget覆盖整个屏幕，这意味着文本“Hello, world”会居中显示在屏幕上。文本显示的方向需要在`Text`实例中指定，当使用`MaterialApp`时，文本的方向将自动设定，稍后将进行演示。

在编写应用程序时，通常会创建新的widget，这些widget是无状态的 `StatelessWidget` 或者是有状态的 `StatefulWidget`，具体的选择取决于您的widget是否需要管理一些状态。widget的主要工作是实现一个 `build` 函数，用以构建自身。一个widget通常由一些较低级别widget组成。Flutter框架将依次构建这些widget，直到构建到最底层的子widget时，这些最低层的widget通常为 `RenderObject`，它会计算并描述widget的几何形状。

基础 Widget

主要文章: [widget概述-布局模型](#)

Flutter有一套丰富、强大的基础widget，其中以下是很常用的：

- `Text`：该 widget 可让创建一个带格式的文本。
- `Row`、`Column`：这些具有弹性空间的布局类Widget可让您在水平（Row）和垂直（Column）方向上创建灵活的布局。其设计是基于web开发中的Flexbox布局模型。
- `Stack`：取代线性布局 (译者语：和Android中的LinearLayout相似)，`Stack` 允许子 widget 堆叠，您可以使用 `Positioned` 来定位他们相对于 `Stack` 的上下左右四条边的位置。Stacks是基于Web开发中的绝度定位（absolute positioning）布局模型设计的。
- `Container`：`Container` 可让您创建矩形视觉元素。container 可以装饰为一个 `BoxDecoration`，如 background、一个边框、或者一个阴影。`Container` 也可以具有边距（margins）、填充(padding)和应用于其大小的约束(constraints)。另外，`Container` 可以使用矩阵在三维空间中对其进行变换。

以下是一些简单的Widget，它们可以组合出其它的Widget：

```
import 'package:flutter/material.dart';

class MyAppBar extends StatelessWidget {
  MyAppBar({this.title});

  // Widget子类中的字段往往会定义为"final"

  final Widget title;

  @override
  Widget build(BuildContext context) {
    return new Container(
      height: 56.0, // 单位是逻辑上的像素（并非真实的像素，类似于浏览器中的像素）
      padding: const EdgeInsets.symmetric(horizontal: 8.0),
      decoration: new BoxDecoration(color: Colors.blue[500]),
    );
  }
}
```

```

// Row 是水平方向的线性布局 (linear layout)
child: new Row(
  //列表项的类型是 <Widget>
  children: <Widget>[
    new IconButton(
      icon: new Icon(Icons.menu),
      tooltip: 'Navigation menu',
      onPressed: null, // null 会禁用 button
    ),
    // Expanded expands its child to fill the available space.
    new Expanded(
      child: title,
    ),
    new IconButton(
      icon: new Icon(Icons.search),
      tooltip: 'Search',
      onPressed: null,
    ),
  ],
),
);
}

class MyScaffold extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Material 是UI呈现的“一张纸”
    return new Material(
      // Column is 垂直方向的线性布局.
      child: new Column(
        children: <Widget>[
          new MyAppBar(
            title: new Text(
              'Example title',
              style: Theme.of(context).primaryTextTheme.title,
            ),
          ),
          new Expanded(
            child: new Center(
              child: new Text('Hello, world!'),
            ),
          ),
        ],
      ),
    );
  }
}

void main() {
  runApp(new MaterialApp(
    title: 'My app', // used by the OS task switcher
    home: new MyScaffold(),
  ));
}

```

请确保在pubspec.yaml文件中，将 flutter 的值设置为： uses-material-design: true 。这允许我们可以使用一组预定义Material icons。

```
name: my_app
flutter:
  uses-material-design: true
```

为了继承主题数据，widget需要位于 MaterialApp 内才能正常显示， 因此我们使用 MaterialApp 来运行该应用。

在 MyAppBar 中创建一个 Container ， 高度为56像素（像素单位独立于设备，为逻辑像素）， 其左侧和右侧均有8像素的填充。在容器内部， MyAppBar 使用 Row 布局来排列其子项。中间的 title widget被标记为 Expanded , , 这意味着它会填充尚未被其他子项占用的的剩余可用空间。Expanded可以拥有多个children， 然后使用 flex 参数来确定他们占用剩余空间的比例。

MyScaffold 通过一个 Column widget，在垂直方向排列其子项。在 Column 的顶部，放置了一个 MyAppBar 实例， 将一个Text widget作为其标题传递给应用程序栏。将widget作为参数传递给其他widget是一种强大的技术，可以让您创建各种复杂的widget。最后， MyScaffold 使用了一个 Expanded 来填充剩余的空间，正中间包含一条message。

使用 Material 组件

主要文章: [Widgets 总览 - Material 组件](#)

Flutter提供了许多widgets，可帮助您构建遵循Material Design的应用程序。Material应用程序以 MaterialApp widget开始， 该widget在应用程序的根部创建了一些有用的widget，其中包括一个 Navigator ， 它管理由字符串标识的Widget栈（即页面路由栈）。 Navigator 可以让您的应用程序在页面之间的平滑的过渡。 是否使用 MaterialApp 完全是可选的，但是使用它是一个很好的做法。

```
import 'package:flutter/material.dart';

void main() {
  runApp(new MaterialApp(
    title: 'Flutter Tutorial',
    home: new TutorialHome(),
  ));
}

class TutorialHome extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    //Scaffold是Material中主要的布局组件。
```

```

return new Scaffold(
  appBar: new AppBar(
    leading: new IconButton(
      icon: new Icon(Icons.menu),
      tooltip: 'Navigation menu',
      onPressed: null,
    ),
    title: new Text('Example title'),
    actions: <Widget>[
      new IconButton(
        icon: new Icon(Icons.search),
        tooltip: 'Search',
        onPressed: null,
      ),
    ],
  ),
  //body占屏幕的大部分
  body: new Center(
    child: new Text('Hello, world!'),
  ),
  floatingActionButton: new FloatingActionButton(
    tooltip: 'Add', // used by assistive technologies
    child: new Icon(Icons.add),
    onPressed: null,
  ),
);
}
}

```

现在我们已经从 `MyAppBar` 和 `MyScaffold` 切换到了 `AppBar` 和 `Scaffold` widget，我们的应用程序现在看起来已经有一些“Material”了！例如，应用栏有一个阴影，标题文本会自动继承正确的样式。我们还添加了一个浮动操作按钮，以便进行相应的操作处理。

请注意，我们再次将widget作为参数传递给其他widget。该 `Scaffold` widget 需要许多不同的widget的作为命名参数，其中的每一个被放置在 `Scaffold` 布局中相应的位置。同样，`AppBar` 中，我们给参数 `leading`、`actions`、`title` 分别传一个widget。这种模式在整个框架中会经常出现，这也可能是您在设计自己的widget时会考虑到一点。

处理手势

主要文章: [Flutter中的手势](#)

大多数应用程序包括某种形式与系统的交互。构建交互式应用程序的第一步是检测输入手势。让我们通过创建一个简单的按钮来了解它的工作原理：

```

class MyButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new GestureDetector(

```

```

    onTap: () {
      print('MyButton was tapped!');
    },
    child: new Container(
      height: 36.0,
      padding: const EdgeInsets.all(8.0),
      margin: const EdgeInsets.symmetric(horizontal: 8.0),
      decoration: new BoxDecoration(
        borderRadius: new BorderRadius.circular(5.0),
        color: Colors.lightGreen[500],
      ),
      child: new Center(
        child: new Text('Engage'),
      ),
    ),
  );
}

```

该 `GestureDetector` widget 并不具有显示效果，而是检测由用户做出的手势。当用户点击 `Container` 时，`GestureDetector` 会调用它的 `onTap` 回调，在回调中，将消息打印到控制台。您可以使用 `GestureDetector` 来检测各种输入手势，包括点击、拖动和缩放。

许多widget都会使用一个 `GestureDetector` 为其他widget提供可选的回调。例如，`IconButton`、`RaisedButton`、和 `FloatingActionButton`，它们都有一个 `onPressed` 回调，它会在用户点击该widget时被触发。

根据用户输入改变widget

主要文章: `StatefulWidget`，`State.setState`

到目前为止，我们只使用了无状态的widget。无状态widget从它们的父widget接收参数，它们被存储在 `final` 型的成员变量中。当一个widget被要求构建时，它使用这些存储的值作为参数来构建widget。

为了构建更复杂的体验 - 例如，以更有趣的方式对用户输入做出反应 - 应用程序通常会携带一些状态。Flutter使用StatefulWidgets来满足这种需求。StatefulWidgets是特殊的widget，它知道如何生成State对象，然后用它来保持状态。思考下面这个简单的例子，其中使用了前面提到 `RaisedButton`：

```

class Counter extends StatefulWidget {
  // This class is the configuration for the state. It holds the
  // values (in this nothing) provided by the parent and used by the build
  // method of the State. Fields in a Widget subclass are always marked "final".

  @override
  _CounterState createState() => new _CounterState();
}

class _CounterState extends State<Counter> {

```

```

int _counter = 0;

void _increment() {
  setState(() {
    // This call to setState tells the Flutter framework that
    // something has changed in this State, which causes it to rerun
    // the build method below so that the display can reflect the
    // updated values. If we changed _counter without calling
    // setState(), then the build method would not be called again,
    // and so nothing would appear to happen.
    _counter++;
  });
}

@override
Widget build(BuildContext context) {
  // This method is rerun every time setState is called, for instance
  // as done by the _increment method above.
  // The Flutter framework has been optimized to make rerunning
  // build methods fast, so that you can just rebuild anything that
  // needs updating rather than having to individually change
  // instances of widgets.
  return new Row(
    children: <Widget>[
      new RaisedButton(
        onPressed: _increment,
        child: new Text('Increment'),
      ),
      new Text('Count: $_counter'),
    ],
  );
}

```

您可能想知道为什么 **StatefulWidget** 和 **State** 是单独的对象。在 **Flutter** 中，这两种类型的对象具有不同的生命周期：**Widget** 是临时对象，用于构建当前状态下的应用程序，而 **State** 对象在多次调用 `build()` 之间保持不变，允许它们记住信息(状态)。

上面的例子接受用户点击，并在点击时使 `_counter` 自增，然后直接在其 `build` 方法中使用 `_counter` 值。在更复杂的应用程序中，**widget** 结构层次的不同部分可能有不同的职责；例如，一个 **widget** 可能呈现一个复杂的用户界面，其目标是收集特定信息（如日期或位置），而另一个 **widget** 可能会使用该信息来更改整体的显示。

在 **Flutter** 中，事件流是“向上”传递的，而状态流是“向下”传递的（译者语：这类似于 **React/Vue** 中父子组件通信的方式：子 **widget** 到父 **widget** 是通过事件通信，而父到子是通过状态），重定向这一流程的共同父元素是 **State**。让我们看看这个稍微复杂的例子是如何工作的：

```

class CounterDisplay extends StatelessWidget {
  CounterDisplay({this.count});

```

```

    final int count;

    @override
    Widget build(BuildContext context) {
      return new Text('Count: $count');
    }
  }

class CounterIncrementor extends StatelessWidget {
  CounterIncrementor({this.onPressed});

  final VoidCallback onPressed;

  @override
  Widget build(BuildContext context) {
    return new RaisedButton(
      onPressed: onPressed,
      child: new Text('Increment'),
    );
  }
}

class Counter extends StatefulWidget {
  @override
  _CounterState createState() => new _CounterState();
}

class _CounterState extends State<Counter> {
  int _counter = 0;

  void _increment() {
    setState(() {
      ++_counter;
    });
  }

  @override
  Widget build(BuildContext context) {
    return new Row(children: <Widget>[
      new CounterIncrementor(onPressed: _increment),
      new CounterDisplay(count: _counter),
    ]);
  }
}

```

注意我们是如何创建了两个新的无状态widget的！我们清晰地分离了 显示 计数器（CounterDisplay）和 更改计数器（CounterIncrementor）的逻辑。 尽管最终效果与前一个示例相同，但责任分离允许将复杂性逻辑封装在各个widget中，同时保持父项的简单性。

整合所有

让我们考虑一个更完整的例子，将上面介绍的概念汇集在一起。我们假设一个购物应用程序，该应用程序显示

出售的各种产品，并维护一个购物车。我们先来定义 `ShoppingListItem`：

```
class Product {
  const Product({this.name});
  final String name;
}

typedef void CartChangedCallback(Product product, bool inCart);

class ShoppingListItem extends StatelessWidget {
  ShoppingListItem({Product product, this.inCart, this.onCartChanged})
    : product = product,
      super(key: new ObjectKey(product));

  final Product product;
  final bool inCart;
  final CartChangedCallback onCartChanged;

  Color _getColor(BuildContext context) {
    // The theme depends on the BuildContext because different parts of the tree
    // can have different themes. The BuildContext indicates where the build is
    // taking place and therefore which theme to use.

    return inCart ? Colors.black54 : Theme.of(context).primaryColor;
  }

  TextStyle _getTextStyle(BuildContext context) {
    if (!inCart) return null;

    return new TextStyle(
      color: Colors.black54,
      decoration: TextDecoration.lineThrough,
    );
  }

  @override
  Widget build(BuildContext context) {
    return new ListTile(
      onTap: () {
        onCartChanged(product, !inCart);
      },
      leading: new CircleAvatar(
        backgroundColor: _getColor(context),
        child: new Text(product.name[0]),
      ),
      title: new Text(product.name, style: _getTextStyle(context)),
    );
  }
}
```

该 `ShoppingListItem` widget是无状态的。它将其在构造函数中接收到的值存储在 `final` 成员变量中，然后

在 `build` 函数中使用它们。例如，`inCart` 布尔值表示在两种视觉展示效果之间切换：一个使用当前主题的主色，另一个使用灰色。

当用户点击列表项时，`widget`不会直接修改其 `inCart` 的值。相反，`widget`会调用其父`widget`给它的 `onCartChanged` 回调函数。此模式可让您在`widget`层次结构中存储更高的状态，从而使状态持续更长的时间。在极端情况下，存储传给 `runApp` 应用程序的`widget`的状态将在整个生命周期中持续存在。

当父项收到 `onCartChanged` 回调时，父项将更新其内部状态，这将触发父项使用新 `inCart` 值重建 `ShoppingListItem` 新实例。虽然父项 `ShoppingListItem` 在重建时创建了一个新实例，但该操作开销很小，因为Flutter框架会将新构建的`widget`与先前构建的`widget`进行比较，并仅将差异部分应用于底层 `RenderObject`。

我们来看看父`widget`存储可变状态的示例：

```
class ShoppingList extends StatefulWidget {
  ShoppingList({Key key, this.products}) : super(key: key);

  final List<Product> products;

  // The framework calls createState the first time a widget appears at a given
  // location in the tree. If the parent rebuilds and uses the same type of
  // widget (with the same key), the framework will re-use the State object
  // instead of creating a new State object.

  @override
  _ShoppingListState createState() => new _ShoppingListState();
}

class _ShoppingListState extends State<ShoppingList> {
  Set<Product> _shoppingCart = new Set<Product>();

  void _handleCartChanged(Product product, bool inCart) {
    setState(() {
      // When user changes what is in the cart, we need to change _shoppingCart
      // inside a setState call to trigger a rebuild. The framework then calls
      // build, below, which updates the visual appearance of the app.

      if (inCart)
        _shoppingCart.add(product);
      else
        _shoppingCart.remove(product);
    });
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Shopping List'),
      ),
```

```

    body: new ListView(
      padding: new EdgeInsets.symmetric(vertical: 8.0),
      children: widget.products.map((Product product) {
        return new ShoppingListItem(
          product: product,
          inCart: _shoppingCart.contains(product),
          onCartChanged: _handleCartChanged,
        );
      }).toList(),
    ),
  );
}

void main() {
  runApp(new MaterialApp(
    title: 'Shopping App',
    home: new ShoppingList(
      products: <Product>[
        new Product(name: 'Eggs'),
        new Product(name: 'Flour'),
        new Product(name: 'Chocolate chips'),
      ],
    ),
  ));
}

```

`ShoppingList` 类继承自 `StatefulWidget`，这意味着这个 `widget` 可以存储状态。当 `ShoppingList` 首次插入到树中时，框架会调用其 `createState` 函数以创建一个新的 `_ShoppingListState` 实例来与该树中的相应位置关联（请注意，我们通常命名 `State` 子类时带一个下划线，这表示其是私有的）。当这个 `widget` 的父级重建时，父级将创建一个新的 `ShoppingList` 实例，但是 **Flutter** 框架将重用已经在树中的 `_ShoppingListState` 实例，而不是再次调用 `createState` 创建一个新的。

要访问当前 `ShoppingList` 的属性，`_ShoppingListState` 可以使用它的 `widget` 属性。如果父级重建并创建一个新的 `ShoppingList`，那么 `_ShoppingListState` 也将用新的 `widget` 值重建（译者语：这里原文档有错误，应该是 `_ShoppingListState` 不会重新构建，但其 `widget` 的属性会更新为新构建的 `widget`）。如果希望在 `widget` 属性更改时收到通知，则可以覆盖 `didUpdateWidget` 函数，以便将旧的 `oldWidget` 与当前 `widget` 进行比较。

处理 `onCartChanged` 回调时，`_ShoppingListState` 通过添加或删除产品来改变其内部 `_shoppingCart` 状态。为了通知框架它改变了它的内部状态，需要调用 `setState`。调用 `setState` 将该 `widget` 标记为“dirty”（脏的），并且计划在下次应用程序需要更新屏幕时重新构建它。如果在修改 `widget` 的内部状态后忘记调用 `setState`，框架将不知道您的 `widget` 是“dirty”（脏的），并且可能不会调用 `widget` 的 `build` 方法，这意味着用户界面可能不会更新以展示新的状态。

通过以这种方式管理状态，您不需要编写用于创建和更新子 `widget` 的单独代码。相反，您只需实现可以处理这两种情况的 `build` 函数。

响应 widget 生命周期事件

主要文章: [State](#)

在`StatefulWidget`调用 `createState` 之后，框架将新的状态对象插入树中，然后调用状态对象的 `initState`。子类化`State`可以重写 `initState`，以完成仅需要执行一次的工作。例如，您可以重写 `initState` 以配置动画或订阅`platform services`。`initState` 的实现中需要调用 `super.initState`。

当一个状态对象不再需要时，框架调用状态对象的 `dispose`。您可以覆盖该 `dispose` 方法来执行清理工作。例如，您可以覆盖 `dispose` 取消定时器或取消订阅`platform services`。`dispose` 典型的实现是直接调用 `super.dispose`。

Key

主要文章: [Key](#)

您可以使用`key`来控制框架将在`widget`重建时与哪些其他`widget`匹配。默认情况下，框架根据它们的 `runtimeType` 和它们的显示顺序来匹配。使用 `key` 时，框架要求两个`widget`具有相同的 `key` 和 `runtimeType`。

`Key`在构建相同类型`widget`的多个实例时很有用。例如，`ShoppingList` 构建足够的 `ShoppingListItem` 实例以填充其可见区域：

- 如果没有`key`，当前构建中的第一个条目将始终与前一个构建中的第一个条目同步，即使在语义上，列表中的第一个条目如果滚动出屏幕，那么它将不会再在窗口中可见。
- 通过给列表中的每个条目分配为“语义” `key`，无限列表可以更高效，因为框架将同步条目与匹配的语义`key`并因此具有相似（或相同）的可视外观。此外，语义上同步条目意味着在有状态子`widget`中，保留的状态将附加到相同的语义条目上，而不是附加到相同数字位置上的条目。

全局 Key

主要文章: [GlobalKey](#)

您可以使用全局`key`来唯一标识子`widget`。全局`key`在整个`widget`层次结构中必须是全局唯一的，这与局部`key`不同，后者只需要在同级中唯一。由于它们是全局唯一的，因此可以使用全局`key`来检索与`widget`关联的状态。



Cookbook

[编辑本页](#) [提Issue](#)

Cookbook包含了在编写Flutter应用程序时常见问题及示例。每个主题都是独立的，可以当做作参考文档来帮助您构建应用程序

设计基础

- [使用主题共享颜色和字体样式](#)

Images

- [显示来自网上的图片](#)
- [用占位符淡入图片](#)
- [使用缓存图](#)

Lists

- [创建一个基本list](#)
- [创建一个水平list](#)
- [使用长列表](#)
- [创建不同类型子项的List](#)
- [创建一个 grid List](#)

处理手势

- [处理点击](#)
- [添加Material触摸水波效果](#)
- [实现滑动关闭](#)

导航

- [导航到新页面并返回](#)
- [给新页面传值](#)
- [从新页面返回数据给上一个页面](#)

网络

- [从网上获取数据](#)
- [进行认证请求](#)

使用WebSockets



使用主题共享颜色和字体样式

[编辑本页](#) [提Issue](#)

为了在整个应用中共享颜色和字体样式，我们可以使用主题。定义主题有两种方式：全局主题或使用 `Theme` 来定义应用程序局部的颜色和字体样式。事实上，全局主题只是由应用程序根 `MaterialApp` 创建的 `Theme` ！

定义一个主题后，我们可以在我们自己的 `Widgets` 中使用它。另外，Flutter 提供的 `Material Widgets` 将使用我们的主题为 `AppBar`、`Buttons`、`Checkboxes` 等设置背景颜色和字体样式。

创建应用主题

为了在整个应用程序中共享包含颜色和字体样式的主题，我们可以提供 `ThemeData` 给 `MaterialApp` 的构造函数。

如果没有提供 `theme`，Flutter 将创建一个默认主题。

```
new MaterialApp(  
  title: title,  
  theme: new ThemeData(  
    brightness: Brightness.dark,  
    primaryColor: Colors.lightBlue[800],  
    accentColor: Colors.cyan[600],  
  ),  
);
```

请参阅 [ThemeData](#) 文档以查看您可以定义的所有颜色和字体。

局部主题

如果我们想在应用程序的一部分中覆盖应用程序的全局的主题，我们可以将要覆盖得部分封装在一个 `Theme Widget` 中。

有两种方法可以解决这个问题：创建特有的 `ThemeData` 或扩展父主题。

创建特有的 `ThemeData`

如果我们不想继承任何应用程序的颜色或字体样式，我们可以通过 `new ThemeData()` 创建一个实例并将其传递给 `Theme Widget`。

```
new Theme (  
  // Create a unique theme with "new ThemeData"  
  data: new ThemeData(  
    accentColor: Colors.yellow,  
  ),  
  child: new FloatingActionButton(  
    onPressed: () {},  
    child: new Icon(Icons.add),  
  ),  
);
```

扩展父主题

扩展父主题时无需覆盖所有的主题属性，我们可以通过使用 `copyWith` 方法来实现。

```
new Theme (  
  // Find and Extend the parent theme using "copyWith". Please see the next  
  // section for more info on `Theme.of`.  
  data: Theme.of(context).copyWith(accentColor: Colors.yellow),  
  child: new FloatingActionButton(  
    onPressed: null,  
    child: new Icon(Icons.add),  
  ),  
);
```

使用主题

现在我们已经定义了一个主题，我们可以在Widget的 `build` 方法中通过 `Theme.of(context)` 函数使用它！

`Theme.of(context)` 将查找Widget树并返回树中最近的 `Theme`。如果我们的Widget之上有一个单独的 `Theme` 定义，则返回该值。如果不是，则返回App主题。事实上，`FloatingActionButton` 真是通过这种方式找到 `accentColor` 的！

下面看一个简单的示例：

```
new Container(  
  color: Theme.of(context).accentColor,  
  child: new Text(  
    'Text with a background color',  
    style: Theme.of(context).textTheme.title,  
  ),  
);
```

完整的例子

```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';

void main() {
  runApp(new MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final appName = 'Custom Themes';

    return new MaterialApp(
      title: appName,
      theme: new ThemeData(
        brightness: Brightness.dark,
        primaryColor: Colors.lightBlue[800],
        accentColor: Colors.cyan[600],
      ),
      home: new MyHomePage(
        title: appName,
      ),
    );
  }
}

class MyHomePage extends StatelessWidget {
  final String title;

  MyHomePage({Key key, @required this.title}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(title),
      ),
      body: new Center(
        child: new Container(
          color: Theme.of(context).accentColor,
          child: new Text(
            'Text with a background color',
            style: Theme.of(context).textTheme.title,
          ),
        ),
      ),
      floatingActionButton: new Theme(
        data: Theme.of(context).copyWith(accentColor: Colors.yellow),
        child: new FloatingActionButton(
```



```
        onPressed: null,
        child: new Icon(Icons.add),
      ),
    ),
  );
}
```



显示来自网上的图片

[编辑本页](#) [提Issue](#)

显示图片是大多数移动应用程序的基础。Flutter提供了 `Image` Widget来显示不同类型的图片。

为了处理来自URL的图像，请使用 `Image.network` 构造函数。

```
new Image.network(  
  'https://raw.githubusercontent.com/flutter/website/master/_includes/code/layout/lakes  
/images/lake.jpg',  
)
```

Bonus: GIF动画

`Image` Widget的一个惊艳的功能是：支持GIF动画！

```
new Image.network(  
  'https://github.com/flutter/plugins/raw/master/packages/video_player/doc/demo_ipod.gif?raw=true',  
);
```

占位图和缓存

`Image.network` 默认不能处理一些高级功能，例如在在下载完图片后加载或缓存图片到设备中后，使图片渐隐渐显。要实现这种功能，请参阅以下内容：

- [用占位符淡入图片](#)
- [使用缓存图片](#)

完整的例子

```
import 'package:flutter/material.dart';  
  
void main() => runApp(new MyApp());
```

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    var title = 'Web Images';  
  
    return new MaterialApp(  
      title: title,  
      home: new Scaffold(  
        appBar: new AppBar(  
          title: new Text(title),  
        ),  
        body: new Image.network(  
          'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes/i  
images/lake.jpg?raw=true',  
        ),  
      ),  
    );  
  }  
}
```



用占位符淡入图片

[编辑本页](#) [提Issue](#)

当使用默认 `Image` widget显示图片时，您可能会注意到它们在加载完成后会直接显示到屏幕上。这可能会让用户产生视觉突兀。

相反，如果你最初显示一个占位符，然后在图像加载完显示时淡入，那么它会不会更好？我们可以使用 `FadeInImage` 来达到这个目的！

`FadeInImage` 适用于任何类型的图片：内存、本地**Asset**或来自网上的图片。

在这个例子中，我们将使用`transparent_image`包作为一个简单的透明占位图。您也可以考虑按照**[Assets和图](#)****[片](#)**指南使用本地资源来做占位图。

```
new FadeInImage.memoryNetwork(  
  placeholder: kTransparentImage,  
  image: 'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes/images/lake.jpg?raw=true',  
);
```

完整的例子

```
import 'package:flutter/material.dart';  
import 'package:transparent_image/transparent_image.dart';  
  
void main() {  
  runApp(new MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    final title = 'Fade in images';  
  
    return new MaterialApp(  
      title: title,  
      home: new Scaffold(  
        appBar: new AppBar(  
          title: new Text(title),  
        ),  
        body: new Stack(  

```

```
children: <Widget>[
  new Center(child: new CircularProgressIndicator()),
  new Center(
    child: new FadeInImage.memoryNetwork(
      placeholder: kTransparentImage,
      image:
        'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes/images/lake.jpg?raw=true',
    ),
  ),
],
);
```



使用缓存图片

[编辑本页](#) [提Issue](#)

在某些情况下，在从网上下载图片后缓存图片可能会很方便，以便它们可以脱机使用。为此，我们可以使用 `cached_network_image` 包来达到目的。

除了缓存之外，`cached_image_network`包在加载时还支持占位符和淡入淡出图片！

```
new CachedNetworkImage(  
  imageUrl: 'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes  
/images/lake.jpg?raw=true',  
);
```

添加一个占位符

`cached_network_image` 包允许我们使用任何Widget作为占位符！在这个例子中，我们将在图片加载时显示一个进度圈。

```
new CachedNetworkImage(  
  placeholder: new CircularProgressIndicator(),  
  imageUrl: 'https://github.com/flutter/website/blob/master/_includes/code/layout/lakes  
/images/lake.jpg?raw=true',  
);
```

完整的例子

```
import 'package:flutter/material.dart';  
import 'package:cached_network_image/cached_network_image.dart';  
  
void main() {  
  runApp(new MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {
```

```
final title = 'Cached Images';

return new MaterialApp(
  title: title,
  home: new Scaffold(
    appBar: new AppBar(
      title: new Text(title),
    ),
    body: new Center(
      child: new CachedNetworkImage(
        placeholder: new CircularProgressIndicator(),
        imageUrl:
          'https://github.com/flutter/website/blob/master/_includes/code/layout/1
akes/images/lake.jpg?raw=true',
      ),
    ),
  ),
);
}
```



基本List

[编辑本页](#) [提Issue](#)

显示数据列表是移动应用程序常见的需求。Flutter包含的 `ListView` Widget，使列表变得轻而易举！

创建一个ListView

使用标准 `ListView` 构造函数非常适合仅包含少量条目的列表。我们使用内置的 `ListTile` Widget来作为列表项。

```
new ListView(  
  children: <Widget>[  
    new ListTile(  
      leading: new Icon(Icons.map),  
      title: new Text('Maps'),  
    ),  
    new ListTile(  
      leading: new Icon(Icons.photo_album),  
      title: new Text('Album'),  
    ),  
    new ListTile(  
      leading: new Icon(Icons.phone),  
      title: new Text('Phone'),  
    ),  
  ],  
);
```

完整的例子

```
import 'package:flutter/material.dart';  
  
void main() => runApp(new MyApp());  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    final title = 'Basic List';  
  
    return new MaterialApp(  
      title: title,  
      home: new Scaffold(  

```



```

    appBar: new AppBar(
      title: new Text(title),
    ),
    body: new ListView(
      children: <Widget>[
        new ListTile(
          leading: new Icon(Icons.map),
          title: new Text('Map'),
        ),
        new ListTile(
          leading: new Icon(Icons.photo),
          title: new Text('Album'),
        ),
        new ListTile(
          leading: new Icon(Icons.phone),
          title: new Text('Phone'),
        ),
      ],
    ),
  ),
);
}
```



创建一个水平list

[编辑本页](#)[提Issue](#)

有时，您可能想要创建一个水平滚动（而不是垂直滚动）的列表。`ListView` 本身就支持水平list。

在创建`ListView`时，设置 `scrollDirection` 为水平方向以覆盖默认的垂直方向。

```
new ListView(  
  // This next line does the trick.  
  scrollDirection: Axis.horizontal,  
  children: <Widget>[  
    new Container(  
      width: 160.0,  
      color: Colors.red,  
    ),  
    new Container(  
      width: 160.0,  
      color: Colors.blue,  
    ),  
    new Container(  
      width: 160.0,  
      color: Colors.green,  
    ),  
    new Container(  
      width: 160.0,  
      color: Colors.yellow,  
    ),  
    new Container(  
      width: 160.0,  
      color: Colors.orange,  
    ),  
  ],  
)
```

完整的例子

```
import 'package:flutter/material.dart';  
  
void main() => runApp(new MyApp());  
  
class MyApp extends StatelessWidget {  
  @override
```

```
Widget build(BuildContext context) {  
  final title = 'Horizontal List';  
  
  return new MaterialApp(  
    title: title,  
    home: new Scaffold(  
      appBar: new AppBar(  
        title: new Text(title),  
      ),  
      body: new Container(  
        margin: new EdgeInsets.symmetric(vertical: 20.0),  
        height: 200.0,  
        child: new ListView(  
          scrollDirection: Axis.horizontal,  
          children: <Widget>[  
            new Container(  
              width: 160.0,  
              color: Colors.red,  
            ),  
            new Container(  
              width: 160.0,  
              color: Colors.blue,  
            ),  
            new Container(  
              width: 160.0,  
              color: Colors.green,  
            ),  
            new Container(  
              width: 160.0,  
              color: Colors.yellow,  
            ),  
            new Container(  
              width: 160.0,  
              color: Colors.orange,  
            ),  
          ],  
        ),  
      ),  
    ),  
  );  
}
```



使用长列表

[编辑本页](#) [提Issue](#)

标准的 `ListView` 构造函数适用于小列表。为了处理包含大量数据的列表，最好使用 `ListView.builder` 构造函数。

`ListView` 的构造函数需要一次创建所有项目，但 `ListView.builder` 的构造函数不需要，它将在列表项滚动到屏幕上时创建该列表项。

1. 创建一个数据源

首先，我们需要一个数据源来。例如，您的数据源可能是消息列表、搜索结果或商店中的产品。大多数情况下，这些数据将来自互联网或数据库。

在这个例子中，我们将使用 `List.generate` 构造函数生成拥有10000个字符串的列表

```
final items = new List<String>.generate(10000, (i) => "Item $i");
```

2. 将数据源转换成Widgets

为了显示我们的字符串列表，我们需要将每个字符串展现为一个Widget！

这正是 `ListView.builder` 发挥作用的地方。在我们的例子中，我们将每一行显示一个字符串：

```
new ListView.builder(  
  itemCount: items.length,  
  itemBuilder: (context, index) {  
    return new ListTile(  
      title: new Text('${items[index]}'),  
    );  
  },  
);
```

完整的例子

```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';

void main() {
  runApp(new MyApp(
    items: new List<String>.generate(10000, (i) => "Item $i"),
  ));
}

class MyApp extends StatelessWidget {
  final List<String> items;

  MyApp({Key key, @required this.items}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final title = 'Long List';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new ListView.builder(
          itemCount: items.length,
          itemBuilder: (context, index) {
            return new ListTile(
              title: new Text('${items[index]}'),
            );
          },
        ),
      ),
    );
  }
}
```



使用不同类型的子项创建列表

[编辑本页](#) [提Issue](#)

我们经常需要创建显示不同类型内容的列表。例如，我们可能正在制作一个列表，其中显示一个标题，后面跟着与该标题相关的几个子项，再后面是另一个标题，等等。

我们如何用Flutter创建这样的结构？

步骤

1. 使用不同类型的数据创建数据源
2. 将数据源转换为Widgets列表

1. 使用不同类型的数据创建数据源

条目(子项)类型

为了表示列表中的不同类型的条目，我们需要为每个类型的条目定义一个类。

在这个例子中，我们将在一个应用程序上显示一个标题，后面跟着五条消息。因此，我们将创建三个类：`ListItem`、`HeadingItem`、和 `MessageItem`。

```
// The base class for the different types of items the List can contain
abstract class ListItem {}

// A ListItem that contains data to display a heading
class HeadingItem implements ListItem {
  final String heading;

  HeadingItem(this.heading);
}

// A ListItem that contains data to display a message
class MessageItem implements ListItem {
  final String sender;
  final String body;

  MessageItem(this.sender, this.body);
}
```

创建Item列表

大多数时候，我们会从互联网或本地数据库中读取数据，并将该数据转换成item的列表。

对于这个例子，我们将生成一个Item列表来处理。该列表将包含一个标题、后跟五条消息，然后重复。

```
final items = new List<ListItem>.generate(
  1200,
  (i) => i % 6 == 0
    ? new HeadingItem("Heading $i")
    : new MessageItem("Sender $i", "Message body $i"),
);
```

2. 将数据源转换为Widgets列表

为了将每个item转换为Widget，我们将使用 `ListView.builder` 构造函数。

通常，我们需要提供一个 `builder` 函数来检查我们正在处理的item类型，并返回该item类型对应的Widget。

在这个例子中，使用 `is` 关键字来检查我们正在处理的item的类型，这个速度很快，并会自动将每个item转换为适当的类型。但是，如果您更喜欢另一种模式，也有不同的方法可以解决这个问题！

```
new ListView.builder(
  // Let the ListView know how many items it needs to build
  itemCount: items.length,
  // Provide a builder function. This is where the magic happens! We'll
  // convert each item into a Widget based on the type of item it is.
  itemBuilder: (context, index) {
    final item = items[index];

    if (item is HeadingItem) {
      return new ListTile(
        title: new Text(
          item.heading,
          style: Theme.of(context).textTheme.headline,
        ),
      );
    } else if (item is MessageItem) {
      return new ListTile(
        title: new Text(item.sender),
        subtitle: new Text(item.body),
      );
    }
  },
);
```

完整的例子

```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';

void main() {
  runApp(new MyApp(
    items: new List<ListItem>.generate(
      1000,
      (i) => i % 6 == 0
        ? new HeadingItem("Heading $i")
        : new MessageItem("Sender $i", "Message body $i"),
    ),
  ));
}

class MyApp extends StatelessWidget {
  final List<ListItem> items;

  MyApp({Key key, @required this.items}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final title = 'Mixed List';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new ListView.builder(
          // Let the ListView know how many items it needs to build
          itemCount: items.length,
          // Provide a builder function. This is where the magic happens! We'll
          // convert each item into a Widget based on the type of item it is.
          itemBuilder: (context, index) {
            final item = items[index];

            if (item is HeadingItem) {
              return new ListTile(
                title: new Text(
                  item.heading,
                  style: Theme.of(context).textTheme.headline,
                ),
              );
            } else if (item is MessageItem) {
              return new ListTile(
                title: new Text(item.sender),
                subtitle: new Text(item.body),
              );
            }
          }
        )
      )
    );
  }
}
```



```
        },
      ),
    ),
  );
}

// The base class for the different types of items the List can contain
abstract class ListItem {}

// A ListItem that contains data to display a heading
class HeadingItem implements ListItem {
  final String heading;

  HeadingItem(this.heading);
}

// A ListItem that contains data to display a message
class MessageItem implements ListItem {
  final String sender;
  final String body;

  MessageItem(this.sender, this.body);
}
```



创建一个 Grid List

[编辑本页](#) [提Issue](#)

在某些情况下，您可能希望将item显示为网格，而不是一个普通列表。对于这需求，我们可以使用 `GridView` Widget。

使用网格的最简单方法是使用 `GridView.count` 构造函数，它允许我们指定行数或列数。

在这个例子中，我们将生成一个包含100个widget的list，在网格中显示它们的索引。这将帮助我们观察 `GridView` 如何工作。

```
new GridView.count(
  // Create a grid with 2 columns. If you change the scrollDirection to
  // horizontal, this would produce 2 rows.
  crossAxisCount: 2,
  // Generate 100 Widgets that display their index in the List
  children: new List.generate(100, (index) {
    return new Center(
      child: new Text(
        'Item $index',
        style: Theme.of(context).textTheme.headline,
      ),
    );
  }),
);
```

完整的例子

```
import 'package:flutter/material.dart';

void main() {
  runApp(new MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Grid List';

    return new MaterialApp(
      title: title,
```

```
home: new Scaffold(  
  appBar: new AppBar(  
    title: new Text(title),  
  ),  
  body: new GridView.count(  
    // Create a grid with 2 columns. If you change the scrollDirection to  
    // horizontal, this would produce 2 rows.  
    crossAxisCount: 2,  
    // Generate 100 Widgets that display their index in the List  
    children: new List.generate(100, (index) {  
      return new Center(  
        child: new Text(  
          'Item $index',  
          style: Theme.of(context).textTheme.headline,  
        ),  
      );  
    }  
  ),  
);
```



处理Taps

[编辑本页](#) [提Issue](#)

我们不仅希望向用户展示信息，还希望我们的用户与我们的应用互动！那么，我们如何响应用户基本操作，如点击和拖动？在Flutter中我们可以使用 `GestureDetector` Widget！

假设我们想要创建一个自定义按钮，当点击时显示一个SnackBar。我们如何解决这个问题？

步骤

1. 创建一个button。
2. 把它包装在 `GestureDetector` 中并提供一个 `onTap` 回调。

```
// Our GestureDetector wraps our button
new GestureDetector(
  // When the child is tapped, show a snackbar
  onTap: () {
    final snackBar = new SnackBar(content: new Text("Tap"));

    Scaffold.of(context).showSnackBar(snackBar);
  },
  // Our Custom Button!
  child: new Container(
    padding: new EdgeInsets.all(12.0),
    decoration: new BoxDecoration(
      color: Theme.of(context).buttonColor,
      borderRadius: new BorderRadius.circular(8.0),
    ),
    child: new Text('My Button'),
  ),
);
```

注意

1. 如果您想将Material 水波效果添加到按钮中，请参阅[“添加Material 触摸水波”](#)。
2. 虽然我们已经创建了一个自定义按钮来演示这些概念，但Flutter也提供了一些其它开箱即用的按钮：[RaisedButton](#)、[FlatButton](#)和[CupertinoButton](#)。

完整的例子

```
import 'package:flutter/material.dart';

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'Gesture Demo';

    return new MaterialApp(
      title: title,
      home: new MyHomePage(title: title),
    );
  }
}

class MyHomePage extends StatelessWidget {
  final String title;

  MyHomePage({Key key, this.title}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(title),
      ),
      body: new Center(child: new MyButton()),
    );
  }
}

class MyButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Our GestureDetector wraps our button
    return new GestureDetector(
      // When the child is tapped, show a snackbar
      onTap: () {
        final snackBar = new SnackBar(content: new Text("Tap"));

        Scaffold.of(context).showSnackBar(snackBar);
      },
      // Our Custom Button!
      child: new Container(
        padding: new EdgeInsets.all(12.0),
        decoration: new BoxDecoration(
          color: Theme.of(context).buttonColor,
          borderRadius: new BorderRadius.circular(8.0),
        ),
        child: new Text('My Button'),
      ),
    );
  }
}
```

```
}

```



添加Material触摸水波效果

[编辑本页](#) [提Issue](#)

在设计应遵循Material Design指南的应用程序时，我们希望在点击时将水波动画添加到Widgets。

Flutter提供了 `InkWell` Widget来实现这个效果。

步骤

1. 创建一个可点击的Widget。
2. 将它包裹在一个 `InkWell` 中来管理点击回调和水波动画。

```
// The InkWell Wraps our custom flat button Widget
new InkWell(
  // When the user taps the button, show a snackbar
  onTap: () {
    Scaffold.of(context).showSnackBar(new SnackBar(
      content: new Text('Tap'),
    ));
  },
  child: new Container(
    padding: new EdgeInsets.all(12.0),
    child: new Text('Flat Button'),
  ),
);
```

完整的例子

```
import 'package:flutter/material.dart';

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'InkWell Demo';

    return new MaterialApp(
      title: title,
      home: new MyHomePage(title: title),
    );
  }
}
```

```
}  
}  
  
class MyHomePage extends StatelessWidget {  
  final String title;  
  
  MyHomePage({Key key, this.title}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return new Scaffold(  
      appBar: new AppBar(  
        title: new Text(title),  
      ),  
      body: new Center(child: new MyButton()),  
    );  
  }  
}  
  
class MyButton extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    // The InkWell Wraps our custom flat button Widget  
    return new InkWell(  
      // When the user taps the button, show a snackbar  
      onTap: () {  
        Scaffold.of(context).showSnackBar(new SnackBar(  
          content: new Text('Tap'),  
        ));  
      },  
      child: new Container(  
        padding: new EdgeInsets.all(12.0),  
        child: new Text('Flat Button'),  
      ),  
    );  
  }  
}
```



实现滑动关闭

[编辑本页](#) [提Issue](#)

“滑动删除”模式在移动应用中很常见。例如，如果我们正在编写一个电子邮件应用程序，我们希望允许我们的用户在列表中滑动电子邮件。当他们这样做时，我们需要将该条目从收件箱移至垃圾箱。

Flutter通过提供 `Dismissable` Widget 使这项任务变得简单。

步骤

- 1. 创建item列表。
- 2. 将每个item包装在一个 `Dismissable` Widget中。
- 3. 提供滑动背景提示。

1. 创建item列表

第一步是创建一个我们可以滑动的列表。有关如何创建列表的更多详细说明，请按照[使用长列表](#)进行操作。

创建数据源

在我们的例子中，我们需要20个条目。为了简单起见，我们将生成一个字符串列表。

```
final items = new List<String>.generate(20, (i) => "Item ${i + 1}");
```

将数据源转换为List

首先，我们将简单地在屏幕上的列表中显示每个项目(先不支持滑动)。

```
new ListView.builder(  
  itemCount: items.length,  
  itemBuilder: (context, index) {  
    return new ListTile(title: new Text('${items[index]}'));  
  },  
);
```

将每个item包装在Dismissible Widget中

现在我们希望让用户能够将条目从列表中移除，用户删除一个条目后，我们需要从列表中删除该条目并显示一个SnackBar。在实际的场景中，您可能需要执行更复杂的逻辑，例如从Web服务或数据库中删除条目。

这是我们就可以使用 Dismissible。在下面的例子中，我们将更新 itemBuilder 函数以返回一个 Dismissible Widget。

```
new Dismissible(  
  // Each Dismissible must contain a Key. Keys allow Flutter to  
  // uniquely identify Widgets.  
  key: new Key(item),  
  // We also need to provide a function that will tell our app  
  // what to do after an item has been swiped away.  
  onDismissed: (direction) {  
    // Remove the item from our data source  
    items.removeAt(index);  
  
    // Show a snackbar! This snackbar could also contain "Undo" actions.  
    Scaffold.of(context).showSnackBar(  
      new SnackBar(content: new Text('$item dismissed')));  
  },  
  child: new ListTile(title: new Text('$item')),  
);
```

3. 提供滑动背景提示

现在，我们的应用程序将允许用户从列表中滑动项目，但用户并不知道滑动后做了什么，所以，我们需要告诉用户滑动操作会移除条目。为此，我们将在滑动条目时显示指示。在下面的例子中，我们通过将背景设置为红色表示为删除操作。

为此，我们为 Dismissible 提供一个 background 参数。

```
new Dismissible(  
  // Show a red background as the item is swiped away  
  background: new Container(color: Colors.red),  
  key: new Key(item),  
  onDismissed: (direction) {  
    items.removeAt(index);  
  
    Scaffold.of(context).showSnackBar(  
      new SnackBar(content: new Text('$item dismissed')));  
  },  
  child: new ListTile(title: new Text('$item')),  
);
```

完整的例子

```
import 'package:flutter/foundation.dart';
import 'package:flutter/material.dart';

void main() {
  runApp(new MyApp(
    items: new List<String>.generate(20, (i) => "Item ${i + 1}"),
  ));
}

class MyApp extends StatelessWidget {
  final List<String> items;

  MyApp({Key key, @required this.items}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final title = 'Dismissing Items';

    return new MaterialApp(
      title: title,
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text(title),
        ),
        body: new ListView.builder(
          itemCount: items.length,
          itemBuilder: (context, index) {
            final item = items[index];

            return new Dismissible(
              // Each Dismissible must contain a Key. Keys allow Flutter to
              // uniquely identify Widgets.
              key: new Key(item),
              // We also need to provide a function that will tell our app
              // what to do after an item has been swiped away.
              onDismissed: (direction) {
                items.removeAt(index);

                Scaffold.of(context).showSnackBar(
                  new SnackBar(content: new Text("$item dismissed")));
              },
              // Show a red background as the item is swiped away
              background: new Container(color: Colors.red),
              child: new ListTile(title: new Text('$item')),
            );
          },
        ),
      ),
    );
  }
}
```

```
}

```



导航到新页面并返回

[编辑本页](#) [提Issue](#)

大多数应用程序包含多个页面。例如，我们可能有一个显示产品的页面，然后，用户可以点击产品，跳到该产品的详情页。

在Android中，页面对应的是Activity，在iOS中是ViewController。而在Flutter中，页面只是一个widget！

在Flutter中，我们那么我们可以使用 `Navigator` 在页面之间跳转。

步骤

1. 创建两个页面。
2. 调用 `Navigator.push` 导航到第二个页面。
3. 调用 `Navigator.pop` 返回第一个页面。

1. 创建两个页面

我们创建两个页面，每个页面包含一个按钮。点击第一个页面上的按钮将导航到第二个页面。点击第二个页面上的按钮将返回到第一个页面。页面结构如下：

```
class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('First Screen'),
      ),
      body: new Center(
        child: new RaisedButton(
          child: new Text('Launch new screen'),
          onPressed: () {
            // Navigate to second screen when tapped!
          },
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
```

```
    appBar: new AppBar(  
      title: new Text("Second Screen"),  
    ),  
    body: new Center(  
      child: new RaisedButton(  
        onPressed: () {  
          // Navigate back to first screen when tapped!  
        },  
        child: new Text('Go back!'),  
      ),  
    ),  
  );  
}
```

2. 调用 Navigator.push 导航到第二个页面

为了导航到新的页面，我们需要调用 `Navigator.push` 方法。该 `push` 方法将添加 `Route` 到由导航器管理的路由栈中！

该 `push` 方法需要一个 `Route`，但 `Route` 从哪里来？我们可以创建自己的，或直接使用 `MaterialPageRoute`。`MaterialPageRoute` 很方便，因为它使用平台特定的动画跳转到新的页面(Android和IOS屏幕切换动画会不同)。

在 `FirstScreen` Widget的 `build` 方法中，我们添加 `onPressed` 回调：

```
// Within the `FirstScreen` Widget  
onPressed: () {  
  Navigator.push(  
    context,  
    new MaterialPageRoute(builder: (context) => new SecondScreen()),  
  );  
}
```

3. 调用 Navigator.pop 返回第一个页面。

现在我们在第二个屏幕上，我们如何关闭它并返回到第一个屏幕？使用 `Navigator.pop` 方法！该 `pop` 方法将 `Route` 从导航器管理的路由栈中移除当前路径。代码如下：

```
// Within the SecondScreen Widget  
onPressed: () {  
  Navigator.pop(context);  
}
```

完整的例子

```
import 'package:flutter/material.dart';

void main() {
  runApp(new MaterialApp(
    title: 'Navigation Basics',
    home: new FirstScreen(),
  ));
}

class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('First Screen'),
      ),
      body: new Center(
        child: new RaisedButton(
          child: new Text('Launch new screen'),
          onPressed: () {
            Navigator.push(
              context,
              new MaterialPageRoute(builder: (context) => new SecondScreen()),
            );
          },
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Second Screen"),
      ),
      body: new Center(
        child: new RaisedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: new Text('Go back!'),
        ),
      ),
    );
  }
}
```

```
    }  
  }  
}
```



给新页面传值

[编辑本页](#) [提Issue](#)

通常，我们不仅要导航到新的页面，还要将一些数据传给页面。例如，我们想传一些关于我们点击的条目的信息。

请记住：页面只是 **Widgets™**。在这个例子中，我们将创建一个 **Todos** 列表。当点击一个 **todo** 时，我们将导航到一个显示关于待办事项信息的新页面（**Widget**）。

Directions

- 1. 定义一个 **Todo** 类。
- 2. 显示 **Todos** 列表。
- 3. 创建一个显示待办事项详情的页面。
- 4. 导航并将数据传递到详情页。

1. 定义一个 Todo 类

首先，我们需要一种简单的方法来表示 **Todos**(待办事项)。在这个例子中，我们将创建一个包含两部分数据的类：标题和描述。

```
class Todo {  
  final String title;  
  final String description;  
  
  Todo(this.title, this.description);  
}
```

2. 创建一个Todos列表

其次，我们要显示一个 **Todos** 列表。在这个例子中，我们将生成 20 个待办事项并使用 **ListView** 显示它们。有关使用列表的更多信息，请参阅 [基础 List](#)。

生成Todos列表

```
final todos = new List<Todo>.generate(  
  20,  
  (i) => new Todo(  

```

```
        'Todo $i',  
        'A description of what needs to be done for Todo $i',  
      ),  
    );
```

使用ListView显示Todos列表

```
new ListView.builder(  
  itemCount: todos.length,  
  itemBuilder: (context, index) {  
    return new ListTile(  
      title: new Text(todos[index].title),  
    );  
  },  
);
```

到现在为止，我们生成了20个Todo并将它们显示在ListView中！

3. 创建一个显示待办事项(todo)详情的页面

现在，我们将创建我们的第二个页面。页面的标题将包含待办事项的标题，页面正文将显示说明。

由于这是一个普通的 `StatelessWidget`，我们只需要在创建页面时传递一个**Todo**！然后，我们将使用给定的**Todo**来构建新的页面。

```
class DetailScreen extends StatelessWidget {  
  // Declare a field that holds the Todo  
  final Todo todo;  
  
  // In the constructor, require a Todo  
  DetailScreen({Key key, @required this.todo}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    // Use the Todo to create our UI  
    return new Scaffold(  
      appBar: new AppBar(  
        title: new Text("${todo.title}"),  
      ),  
      body: new Padding(  
        padding: new EdgeInsets.all(16.0),  
        child: new Text('${todo.description}'),  
      ),  
    );  
  }  
}
```

```
}
```

4. 导航并将数据传递到详情页

接下来，当用户点击我们列表中的待办事项时我们将导航到 `DetailScreen`，并将`Todo`传递给 `DetailScreen`。

为了实现这一点，我们将实现 `ListTile` 的 `onTap` 回调。在我们的 `onTap` 回调中，我们将再次调用 `Navigator.push` 方法。

```
new ListView.builder(  
  itemCount: todos.length,  
  itemBuilder: (context, index) {  
    return new ListTile(  
      title: new Text(todos[index].title),  
      // When a user taps on the ListTile, navigate to the DetailScreen.  
      // Notice that we're not only creating a new DetailScreen, we're  
      // also passing the current todo to it!  
      onTap: () {  
        Navigator.push(  
          context,  
          new MaterialPageRoute(  
            builder: (context) => new DetailScreen(todo: todos[index]),  
          ),  
        );  
      },  
    );  
  },  
);  
};
```

完整的例子

```
import 'package:flutter/foundation.dart';  
import 'package:flutter/material.dart';  
  
class Todo {  
  final String title;  
  final String description;  
  
  Todo(this.title, this.description);  
}  
  
void main() {  
  runApp(new MaterialApp(  

```

```

    title: 'Passing Data',
    home: new TodosScreen(
      todos: new List.generate(
        20,
        (i) => new Todo(
          'Todo $i',
          'A description of what needs to be done for Todo $i',
        ),
      ),
    ),
  ));
}

class TodosScreen extends StatelessWidget {
  final List<Todo> todos;

  TodosScreen({Key key, @required this.todos}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Todos'),
      ),
      body: new ListView.builder(
        itemCount: todos.length,
        itemBuilder: (context, index) {
          return new ListTile(
            title: new Text(todos[index].title),
            // When a user taps on the ListTile, navigate to the DetailScreen.
            // Notice that we're not only creating a new DetailScreen, we're
            // also passing the current todo through to it!
            onTap: () {
              Navigator.push(
                context,
                new MaterialPageRoute(
                  builder: (context) => new DetailScreen(todo: todos[index]),
                ),
              );
            },
          );
        },
      ),
    );
  }
}

class DetailScreen extends StatelessWidget {
  // Declare a field that holds the Todo
  final Todo todo;

  // In the constructor, require a Todo
  DetailScreen({Key key, @required this.todo}) : super(key: key);

  @override

```

```
Widget build(BuildContext context) {  
  // Use the Todo to create our UI  
  return new Scaffold(  
    appBar: new AppBar(  
      title: new Text("${todo.title}"),  
    ),  
    body: new Padding(  
      padding: new EdgeInsets.all(16.0),  
      child: new Text('${todo.description}'),  
    ),  
  );  
}
```



从新页面返回数据给上一个页面

[编辑本页](#) [提Issue](#)

在某些情况下，我们可能想要从新页面返回数据。例如，假设我们导航到了一个新页面，向用户呈现两个选项。当用户点击某个选项时，我们需要将用户选择通知给第一个页面，以便它能够处理这些信息！

我们如何实现？使用 `Navigator.pop` ！

步骤

1. 定义主页。
2. 添加一个打开选择页面的按钮。
3. 在选择页面上显示两个按钮。
4. 点击一个按钮时，关闭选择的页面。
5. 主页上弹出一个snackbar以显示用户的选择。

1. 定义主页

主页将显示一个按钮。点击后，它将打开选择页面！

```
class HomeScreen extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return new Scaffold(  
      appBar: new AppBar(  
        title: new Text('Returning Data Demo'),  
      ),  
      // We'll create the SelectionButton Widget in the next step  
      body: new Center(child: new SelectionButton()),  
    );  
  }  
}
```

2. 添加一个打开选择页面的按钮。

现在，我们将创建我们的SelectionButton。我们的选择按钮将会：

1. 点击时启动SelectionScreen
2. 等待SelectionScreen返回结果

```
class SelectionButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new RaisedButton(
      onPressed: () {
        _navigateAndDisplaySelection(context);
      },
      child: new Text('Pick an option, any option!'),
    );
  }

  // A method that launches the SelectionScreen and awaits the result from
  // Navigator.pop
  _navigateAndDisplaySelection(BuildContext context) async {
    // Navigator.push returns a Future that will complete after we call
    // Navigator.pop on the Selection Screen!
    final result = await Navigator.push(
      context,
      // We'll create the SelectionScreen in the next step!
      new MaterialPageRoute(builder: (context) => new SelectionScreen()),
    );
  }
}
```

3. 在选择页面上显示两个按钮。

现在，我们需要构建一个选择页面！它将包含两个按钮。当用户点击按钮时，应该关闭选择页面并让主页知道哪个按钮被点击！

现在，我们将定义UI，并确定如何在下一步中返回数据。

```
class SelectionScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Pick an option'),
      ),
      body: new Center(
        child: new Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new Padding(
              padding: const EdgeInsets.all(8.0),
              child: new RaisedButton(
                onPressed: () {
                  // Pop here with "Yep"...
                },
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

```
        child: new Text('Yep!'),
      ),
    ),
    new Padding(
      padding: const EdgeInsets.all(8.0),
      child: new RaisedButton(
        onPressed: () {
          // Pop here with "Nope"
        },
        child: new Text('Nope.'),
      ),
    ),
  ),
],
),
),
),
);
}
```

4. 点击一个按钮时，关闭选择的页面。

现在，我们完成两个按钮的 `onPressed` 回调。为了将数据返回到第一个页面，我们需要使用 `Navigator.pop` 方法。

`Navigator.pop` 接受一个可选的(第二个)参数 `result`。如果我们返回结果，它将返回到一个 `Future` 到主页的 `SelectionButton` 中！

Yep 按钮

```
new RaisedButton(
  onPressed: () {
    // Our Yep button will return "Yep!" as the result
    Navigator.pop(context, 'Yep!');
  },
  child: new Text('Yep!'),
);
```

Nope 按钮

```
new RaisedButton(
  onPressed: () {
    // Our Nope button will return "Nope!" as the result
    Navigator.pop(context, 'Nope!');
  },
);
```



```
child: new Text('Nope!'),  
);
```

5. 主页上弹出一个snackbar以显示用户的选择。

既然我们正在启动一个选择页面并等待结果，那么我们会想要对返回的信息进行一些操作！

在这种情况下，我们将显示一个显示结果的SnackBar。为此，我们将更新 `SelectionButton` 中的 `_navigateAndDisplaySelection` 方法。

```
_navigateAndDisplaySelection(BuildContext context) async {  
  final result = await Navigator.push(  
    context,  
    new MaterialPageRoute(builder: (context) => new SelectionScreen()),  
  );  
  
  // After the Selection Screen returns a result, show it in a SnackBar!  
  Scaffold  
    .of(context)  
    .showSnackBar(new SnackBar(content: new Text("$result")));  
}
```

完整的例子

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(new MaterialApp(  
    title: 'Returning Data',  
    home: new HomeScreen(),  
  ));  
}  
  
class HomeScreen extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return new Scaffold(  
      appBar: new AppBar(  
        title: new Text('Returning Data Demo'),  
      ),  
      body: new Center(child: new SelectionButton()),  
    );  
  }  
}
```

```
class SelectionButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new RaisedButton(
      onPressed: () {
        _navigateAndDisplaySelection(context);
      },
      child: new Text('Pick an option, any option!'),
    );
  }

  // A method that launches the SelectionScreen and awaits the result from
  // Navigator.pop!
  _navigateAndDisplaySelection(BuildContext context) async {
    // Navigator.push returns a Future that will complete after we call
    // Navigator.pop on the Selection Screen!
    final result = await Navigator.push(
      context,
      new MaterialPageRoute(builder: (context) => new SelectionScreen()),
    );

    // After the Selection Screen returns a result, show it in a Snackbar!
    Scaffold
      .of(context)
      .showSnackBar(new SnackBar(content: new Text("$result")));
  }
}

class SelectionScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text('Pick an option'),
      ),
      body: new Center(
        child: new Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            new Padding(
              padding: const EdgeInsets.all(8.0),
              child: new RaisedButton(
                onPressed: () {
                  // Close the screen and return "Yep!" as the result
                  Navigator.pop(context, 'Yep!');
                },
                child: new Text('Yep!'),
              ),
            ),
            new Padding(
              padding: const EdgeInsets.all(8.0),
              child: new RaisedButton(
                onPressed: () {
                  // Close the screen and return "Nope!" as the result
```

```
        Navigator.pop(context, 'Nope.');
```

```
      },
```

```
      child: new Text('Nope.'),
```

```
    ),
```

```
  ),
```

```
],
```

```
),
```

```
),
```

```
);
```

```
}
```

```
}
```



从互联网上获取数据

[编辑本页](#) [提Issue](#)

从大多数应用程序都需要从互联网上获取数据，**Dart**和**Flutter**提供了相关工具！

注：本篇示例官方使用的是用 `http` package发起简单的网络请求，但是 `http` package功能较弱，很多常用功能都不支持。我们建议您使用[dio](#) 来发起网络请求，它是一个强大易用的dart http请求库，支持Restful API、FormData、拦截器、请求取消、Cookie管理、文件上传/下载.....详情请查看[github dio](#) .

步骤

1. 添加`http` package依赖
2. 使用`http` package发出网络请求
3. 将响应转为自定义的Dart对象
4. 获取并显示数据

1. 添加 http package

`http` package提供了从互联网获取数据的最简单方法。

```
dependencies:  
  http: <latest_version>
```

2. 发起网络请求

在这个例子中，我们将使用 `http.get` 从JSONPlaceholder REST API中获取示例文章。

```
Future<http.Response> fetchPost() {  
  return http.get('https://jsonplaceholder.typicode.com/posts/1');  
}
```

`http.get` 方法返回一个包含一个 `Response` 的 `Future` 。

- `Future` 是与异步操作一起工作的核心Dart类。它用于表示未来某个时间可能会出现可用值或错误。

- `http.Response` 类包含一个成功的HTTP请求接收到的数据

3. 将响应转换为自定义Dart对象

虽然发出网络请求很简单，但如果要使用原始的 `Future<http.Response>` 并不简单。为了让我们可以开开心心的写代码，我们可以将 `http.Response` 转换成我们自己的Dart对象。

创建一个 `Post` 类

首先，我们需要创建一个 `Post` 类，它包含我们网络请求的数据。它还将包括一个工厂构造函数，它允许我们可以通过json创建一个 `Post` 对象。

手动转换JSON只是一种选择。有关更多信息，请参阅关于[JSON和序列化](#)的完整文章。

```
class Post {
  final int userId;
  final int id;
  final String title;
  final String body;

  Post({this.userId, this.id, this.title, this.body});

  factory Post.fromJson(Map<String, dynamic> json) {
    return new Post(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
      body: json['body'],
    );
  }
}
```

将 `http.Response` 转换成一个 `Post` 对象

现在，我们将更新 `fetchPost` 函数以返回一个 `Future<Post>`。为此，我们需要：

1. 使用 `dart:convert` package将响应内容转化为一个json `Map`。
2. 使用 `fromJson` 工厂函数，将json `Map` 转化为一个 `Post` 对象。

```
Future<Post> fetchPost() async {
  final response = await http.get('https://jsonplaceholder.typicode.com/posts/1');
  final json = JSON.decode(response.body);

  return new Post.fromJson(json);
}
```

Hooray! 现在我们有一个函数，我们可以调用它从互联网上获取一篇文章！

4. 获取并显示数据

为了获取数据并将其显示在屏幕上，我们可以使用 `FutureBuilder` widget! `FutureBuilder` Widget可以很容易与异步数据源一起工作。

我们需要提供两个参数：

1. `future` 参数是一个异步的网络请求，在这个例子中，我们传递调用 `fetchPost()` 函数的返回值(一个future)。
2. 一个 `builder` 函数，这告诉Flutter在 `future` 执行的不同阶段应该如何渲染界面。

```
new FutureBuilder<Post>(
  future: fetchPost(),
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return new Text(snapshot.data.title);
    } else if (snapshot.hasError) {
      return new Text("${snapshot.error}");
    }

    // By default, show a loading spinner
    return new CircularProgressIndicator();
  },
);
```

译者注：其中 `snapshot.data` 即为Future的结果。

完整的例子

```
import 'dart:async';
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

Future<Post> fetchPost() async {
  final response =
    await http.get('https://jsonplaceholder.typicode.com/posts/1');
  final responseJson = json.decode(response.body);

  return new Post.fromJson(responseJson);
}
```

```

class Post {
  final int userId;
  final int id;
  final String title;
  final String body;

  Post({this.userId, this.id, this.title, this.body});

  factory Post.fromJson(Map<String, dynamic> json) {
    return new Post(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
      body: json['body'],
    );
  }
}

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Fetch Data Example',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text('Fetch Data Example'),
        ),
        body: new Center(
          child: new FutureBuilder<Post>(
            future: fetchPost(),
            builder: (context, snapshot) {
              if (snapshot.hasData) {
                return new Text(snapshot.data.title);
              } else if (snapshot.hasError) {
                return new Text("${snapshot.error}");
              }

              // By default, show a loading spinner
              return new CircularProgressIndicator();
            },
          ),
        ),
      ),
    );
  }
}

```



进行认证请求

[编辑本页](#) [提Issue](#)

为了从更多的Web服务获取数据，有些时候您需要提供授权。有很多方法可以做到这一点，但最常见的可能是使用HTTP header中的 `Authorization`。

注：本篇示例官方使用的是用 `http` package发起简单的网络请求，但是 `http` package功能较弱，很多功能都不支持。我们建议您使用[dio](#) 来发起网络请求，它是一个强大易用的dart http请求库，支持Restful API、FormData、拦截器、请求取消、Cookie管理、文件上传/下载.....详情请查看[github dio](#)。

添加 Authorization Headers

`http` package提供了一种方便的方法来为请求添加headers。您也可以使用 `dart:io` package来添加。

```
Future<http.Response> fetchPost() {  
  return http.get(  
    'https://jsonplaceholder.typicode.com/posts/1',  
    // Send authorization headers to your backend  
    headers: {HttpHeaders.AUTHORIZATION: "Basic your_api_token_here"},  
  );  
}
```

完整的例子

这个例子建立在[从互联网上获取数据](#)之上。

```
import 'dart:async';  
import 'dart:convert';  
import 'dart:io';  
import 'package:http/http.dart' as http;  
  
Future<Post> fetchPost() async {  
  final response = await http.get(  
    'https://jsonplaceholder.typicode.com/posts/1',  
    headers: {HttpHeaders.AUTHORIZATION: "Basic your_api_token_here"},  
  );  
  final json = JSON.decode(response.body);  
  
  return new Post.fromJson(json);  
}
```

```
class Post {  
  final int userId;  
  final int id;  
  final String title;  
  final String body;  
  
  Post({this.userId, this.id, this.title, this.body});  
  
  factory Post.fromJson(Map<String, dynamic> json) {  
    return new Post(  
      userId: json['userId'],  
      id: json['id'],  
      title: json['title'],  
      body: json['body'],  
    );  
  }  
}
```



使用WebSockets

[编辑本页](#) [提Issue](#)

除了正常的HTTP请求外，我们还可以使用WebSocket连接到服务器。WebSocket允许与服务器进行双向通信而无需轮询。

在这个例子中，我们将连接到由websocket.org提供的测试服务器。服务器将简单地返回我们发送给它的相同消息！

步骤

1. 连接到WebSocket服务器。
2. 监听来自服务器的消息。
3. 将数据发送到服务器。
4. 关闭WebSocket连接。

1. 连接到WebSocket服务器

`web_socket_channel` package 提供了我们需要连接到WebSocket服务器的工具。

该package提供了一个 `WebSocketChannel` 允许我们既可以监听来自服务器的消息，又可以将消息发送到服务器的方法。

在Flutter中，我们可以创建一个 `WebSocketChannel` 连接到一台服务器：

```
final channel = new IOWebSocketChannel.connect('ws://echo.websocket.org');
```

2. 监听来自服务器的消息

现在我们建立了连接，我们可以监听来自服务器的消息，在我们发送消息给测试服务器之后，它会返回相同的消息。

我们如何收取消息并显示它们？在这个例子中，我们将使用一个 `StreamBuilder` Widget来监听新消息，并用一个Text Widget来显示它们。

```
new StreamBuilder(  
  stream: widget.channel.stream,  
  builder: (context, snapshot) {
```

```
return new Text (snapshot.hasData ? '${snapshot.data}' : '');  
},  
);
```

这是如何工作的?

`WebSocketChannel` 提供了一个来自服务器的消息`Stream`。

该 `Stream` 类是 `dart:async` 包中的一个基础类。它提供了一种方法来监听来自数据源的异步事件。与 `Future` 返回单个异步响应不同, `Stream` 类可以随着时间推移传递很多事件。

该 `StreamBuilder` `Widget`将连接到一个`Stream`, 并在每次收到消息时通知Flutter重新构建界面。

3. 将数据发送到服务器

为了将数据发送到服务器, 我们会 `add` 消息给 `WebSocketChannel` 提供的`sink`。

```
channel.sink.add('Hello!');
```

这是如何工作的?

`WebSocketChannel` 提供了一个 `StreamSink`, 它将消息发给服务器。

`StreamSink` 类提供了给数据源同步或异步添加事件的一般方法。

4. 关闭WebSocket连接

在我们使用 `WebSocket` 后, 要关闭连接:

```
channel.sink.close();
```

完整的例子

```
import 'package:flutter/foundation.dart';  
import 'package:web_socket_channel/io.dart';  
import 'package:flutter/material.dart';  
import 'package:web_socket_channel/web_socket_channel.dart';
```

```

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final title = 'WebSocket Demo';
    return new MaterialApp(
      title: title,
      home: new MyHomePage(
        title: title,
        channel: new IOWebSocketChannel.connect('ws://echo.websocket.org'),
      ),
    );
  }
}

class MyHomePage extends StatefulWidget {
  final String title;
  final WebSocketChannel channel;

  MyHomePage({Key key, @required this.title, @required this.channel})
    : super(key: key);

  @override
  _MyHomePageState createState() => new _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  TextEditingController _controller = new TextEditingController();

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new Padding(
        padding: const EdgeInsets.all(20.0),
        child: new Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: <Widget>[
            new Form(
              child: new TextFormField(
                controller: _controller,
                decoration: new InputDecoration(labelText: 'Send a message'),
              ),
            ),
            new StreamBuilder(
              stream: widget.channel.stream,
              builder: (context, snapshot) {
                return new Padding(
                  padding: const EdgeInsets.symmetric(vertical: 24.0),
                  child: new Text(snapshot.hasData ? '${snapshot.data}' : ''),
                );
              },
            ),
          ],
        ),
      ),
    );
  }
}

```

```
        ),
      ],
    ),
  ),
  floatingActionButton: new FloatingActionButton(
    onPressed: _sendMessage,
    tooltip: 'Send message',
    child: new Icon(Icons.send),
  ), // This trailing comma makes auto-formatting nicer for build methods.
);
}

void _sendMessage() {
  if (_controller.text.isNotEmpty) {
    widget.channel.sink.add(_controller.text);
  }
}

@override
void dispose() {
  widget.channel.sink.close();
  super.dispose();
}
}
```





在Flutter中构建布局

[编辑本页](#) [提Issue](#)

你会学到:

- Flutter的布局机制如何工作.
- 如何垂直和水平布局widget.
- 如何构建一个Flutter布局.

这是在Flutter中构建布局的指南。首先，您将构建以下屏幕截图的布局。然后回过头， 本指南将解释Flutter的布局方法，并说明如何在屏幕上放置一个widget。在讨论如何水平和垂直放置widget之后，会介绍一些最常见的布局widget：



Oeschinen Lake Campground

Kandersteg, Switzerland

★ 41



CALL



ROUTE



SHARE

Lake Oeschinen lies at the foot of the Blüemlisalp in the Bernese Alps. Situated 1,578 meters above sea level, it is one of the larger Alpine Lakes. A gondola ride from Kandersteg, followed by a half-hour walk through pastures and pine forest, leads you to the lake, which warms to 20 degrees Celsius in the summer. Activities enjoyed here include rowing, and riding the summer toboggan run.

- 构建布局
 - 第0步: 设置
 - 第一步: 绘制布局图
 - 第二步: 实现标题行
 - 第三步: 实现button行
 - 第四步: 实现文本部分
 - 第五步: 实现图片部分
 - 第六步: 整合
- Flutter的布局方法

- [放置一个widget](#)
- [水平或垂直排列多个widget](#)
 - [对齐 widget](#)
 - [调整 widget](#)
 - [打包 widget](#)
 - [嵌套行和列](#)
- [通用布局 widget](#)
 - [标准 widget](#)
 - [Material Components](#)
- [资料](#)

构建布局

如果你想对布局机制有一个“全貌”的理解，请参考[Flutter的布局方法](#)

第0步: 设置

首先, 获取代码:

- 确保您已经安装好了 [set up](#) 您的Flutter环境.
- [创建一个基本的Flutter应用程序](#).

接下来, 将图像添加到示例中:

- 在工程根目录创建一个 `images` 文件夹.
- 添加 `lake.jpg`. (请注意, `wget`不能保存此二进制文件。)
- 更新 `pubspec.yaml` 文件以包含 `assets` 标签. 这样才会使您的图片在代码中可用。


第一步: 绘制布局图

第一步是将布局拆分成基本的元素:

- 找出行和列.
- 布局包含网格吗?
- 有重叠的元素吗?
- 是否需要选项卡?
- 注意需要对齐、填充和边框的区域.

首先, 确定更大的元素。在这个例子中, 四个元素排列成一行: 一个图像, 两个行和一个文本块

Column



Oeschinen Lake Campground

Kandersteg, Switzerland

★ 41

CALL

ROUTE

SHARE

Lake Oeschinen lies at the foot of the Blüemlisalp in the Bernese Alps. Situated 1,578 meters above sea level, it is one of the larger Alpine Lakes. A gondola ride from Kandersteg, followed by a half-hour walk through pastures and pine forest, leads you to the lake, which warms to 20 degrees Celsius in the summer. Activities enjoyed here include rowing, and riding the summer toboggan run.

Text

Title section

Icon

Text

Row with 3 children

Text

Column of 2 children
Expanded to fill remaining space

Oeschinen Lake Campground

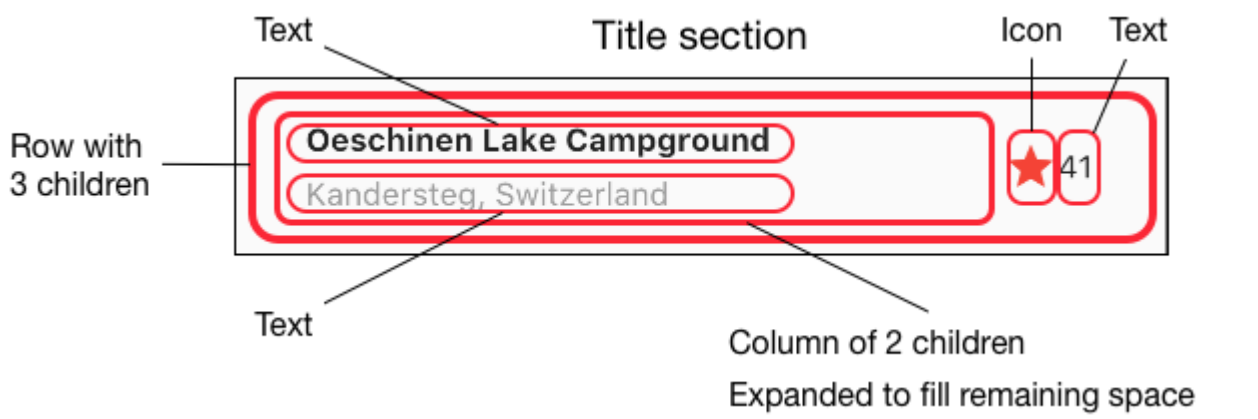
Kandersteg, Switzerland

★

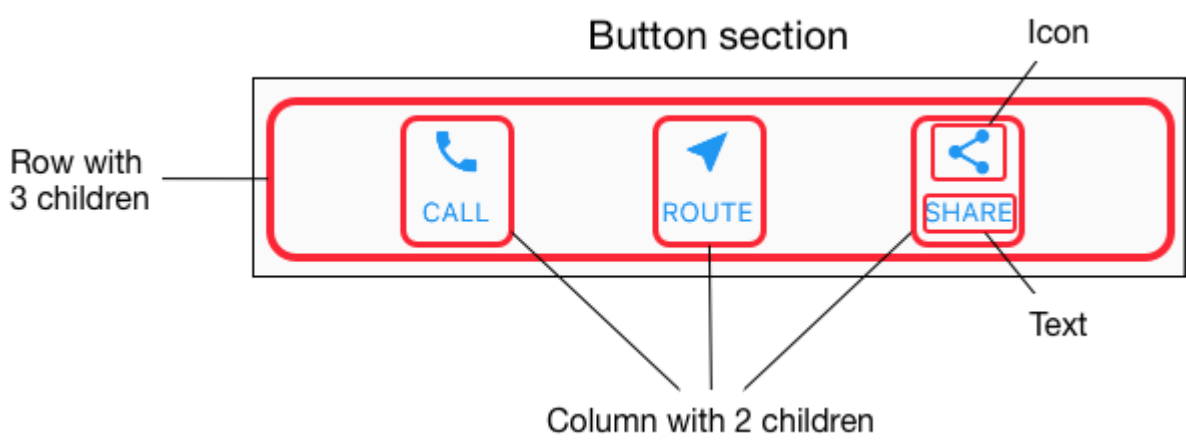
41

<https://flutterchina.club/tutorials/layout/>[2019/2/19 11:18:23]

接下来，绘制每一行。第一行称其为标题部分，有三个子项：一列文字，一个星形图标和一个数字。它的第一个子项，列，包含2行文字。第一列占用大量空间，所以它必须包装在Expanded widget中。



第二行称其为按钮部分，也有3个子项：每个子项都是一个包含图标和文本的列。



一旦拆分好布局，最简单的就是采取自下而上的方法来实现它。为了最大限度地减少深度嵌套布局代码的视觉混淆，将一些实现放置在变量和函数中。

Step 2: 实现标题行

首先，构建标题部分左边栏。将Column（列）放入Expanded中会拉伸该列以使用该行中的所有剩余空闲空间。设置crossAxisAlignment属性值为CrossAxisAlignment.start，这会将该列中的子项左对齐。

将第一行文本放入Container中，然后底部添加8像素填充。列中的第二个子项（也是文本）显示为灰色。

标题行中的最后两项是一个红色的星形图标和文字“41”。将整行放在容器中，并沿着每个边缘填充32像素

这是实现标题行的代码。

Note: If you have problems, you can check your code against [lib/main.dart](#) on GitHub.

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    Widget titleSection = new Container(
      padding: const EdgeInsets.all(32.0),
```

```

child: new Row(
  children: [
    new Expanded(
      child: new Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          new Container(
            padding: const EdgeInsets.only(bottom: 8.0),
            child: new Text(
              'Oeschinen Lake Campground',
              style: new TextStyle(
                fontWeight: FontWeight.bold,
              ),
            ),
          ),
          new Text(
            'Kandersteg, Switzerland',
            style: new TextStyle(
              color: Colors.grey[500],
            ),
          ),
        ],
      ),
    ),
    new Icon(
      Icons.star,
      color: Colors.red[500],
    ),
    new Text('41'),
  ],
),
);
//...
}

```

提示: 将代码粘贴到应用程序中时，缩进可能会变形。您可以通过右键单击，选择 **Reformat with dartfmt** 来在IntelliJ中修复此问题。或者，在命令行中，您可以使用 [dartfmt](#)。

提示: 为了获得更快的开发体验，请尝试使用Flutter的热重载功能。热重载允许您修改代码并查看更改，而无需完全重新启动应用程序。IntelliJ的Flutter插件支持热重载，或者您可以从命令行触发。有关更多信息，请参阅[Hot Reloads vs 应用程序重新启动](#)。

第3步: 实现按钮行

按钮部分包含3个使用相同布局的列 - 上面一个图标，下面一行文本。该行中的列平均分布行空间，文本和图标颜色为主题中的primary color，它在应用程序的build()方法中设置为蓝色：

```

class MyApp extends StatelessWidget {

```

```
@override
Widget build(BuildContext context) {
  //...

  return new MaterialApp(
    title: 'Flutter Demo',
    theme: new ThemeData(
      primarySwatch: Colors.blue,
    ),
    //...
  );
}
```

由于构建每个列的代码几乎是相同的，因此使用一个嵌套函数，如**buildButtonColumn**，它会创建一个颜色为**primary color**，包含一个**Icon**和**Text**的 **Widget** 列。

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    //...

    Column buildButtonColumn(IconData icon, String label) {
      Color color = Theme.of(context).primaryColor;

      return new Column(
        mainAxisAlignment: MainAxisAlignment.min,
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          new Icon(icon, color: color),
          new Container(
            margin: const EdgeInsets.only(top: 8.0),
            child: new Text(
              label,
              style: new TextStyle(
                fontSize: 12.0,
                fontWeight: FontWeight.w400,
                color: color,
              ),
            ),
          ),
        ],
      );
    }
    //...
  }
}
```

构造函数将图标直接添加到列（**Column**）中。将文本放入容器以在文本上方添加填充，将其与图标分开。

通过调用函数并传递**icon**和文本来构建这些列。然后在行的主轴方向通过 `MainAxisAlignment.spaceEvenly` 平均的分配每个列占据的行空间。

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    //...

    Widget buttonSection = new Container(
      child: new Row(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: [
          buildButtonColumn(Icons.call, 'CALL'),
          buildButtonColumn(Icons.near_me, 'ROUTE'),
          buildButtonColumn(Icons.share, 'SHARE'),
        ],
      ),
    );
    //...
  }
}
```

第4步：实现文本部分

将文本放入容器中，以便沿每条边添加32像素的填充。`softwrap` 属性表示文本是否应在软换行符（例如句点或逗号）之间断开。

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    //...

    Widget textSection = new Container(
      padding: const EdgeInsets.all(32.0),
      child: new Text(
        '''
Lake Oeschinen lies at the foot of the Blüemlisalp in the Bernese Alps. Situated 1,578 me
ters above sea level, it is one of the larger Alpine Lakes. A gondola ride from Kanderste
g, followed by a half-hour walk through pastures and pine forest, leads you to the lake,
which warms to 20 degrees Celsius in the summer. Activities enjoyed here include rowing,
and riding the summer toboggan run.
        ''',
        softWrap: true,
      ),
    );
    //...
  }
}
```

第5步：实现图像部分

四列元素中的三个现在已经完成，只剩下图像部分。该图片可以在Creative Commons许可下[在线获得](#)，但是它非常大，且下载缓慢。在步骤0中，您已经将该图像包含在项目中并更新了pubspec文件，所以现在可以从代码中直接引用它：

```
body: new ListView(  
  children: [  
    new Image.asset(  
      'images/lake.jpg',  
      height: 240.0,  
      fit: BoxFit.cover,  
    ),  
    // ...  
  ],  
)
```

`BoxFit.cover` 告诉框架，图像应该尽可能小，但覆盖整个渲染框

Step 6: 整合

在最后一步，你将上面这些组装在一起。这些`widget`放置到`ListView`中，而不是列中，因为在小设备上运行应用程序时，`ListView`会自动滚动。

```
//...  
body: new ListView(  
  children: [  
    new Image.asset(  
      'images/lake.jpg',  
      width: 600.0,  
      height: 240.0,  
      fit: BoxFit.cover,  
    ),  
    titleSection,  
    buttonSection,  
    textSection,  
  ],  
)  
//...
```

Dart 代码: [main.dart](#)

Image: [images](#)

Pubspec: [pubspec.yaml](#)

结束了！当您热重载应用程序时，就会看到和截图中相同界面。您可以参考 [给Flutter APP 添加交互](#)来给您的应用添加交互。

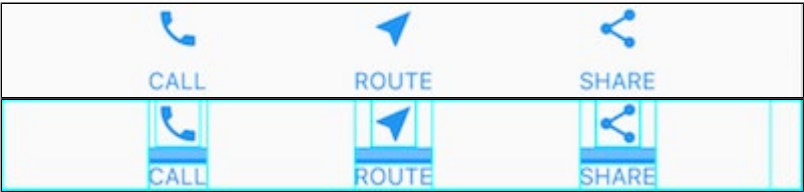
Flutter的布局方法

重点是什么？

- Widgets 是用于构建UI的类.
- Widgets 用于布局和UI元素.
- 通过简单的widget来构建复杂的widget

Flutter布局机制的核心就是widget。在Flutter中，几乎所有东西都是一个widget - 甚至布局模型都是widget。您在Flutter应用中看到的图像、图标和文本都是widget。甚至你看不到的东西也是widget，例如行（row）、列（column）以及用来排列、约束和对齐这些可见widget的网格（grid）。

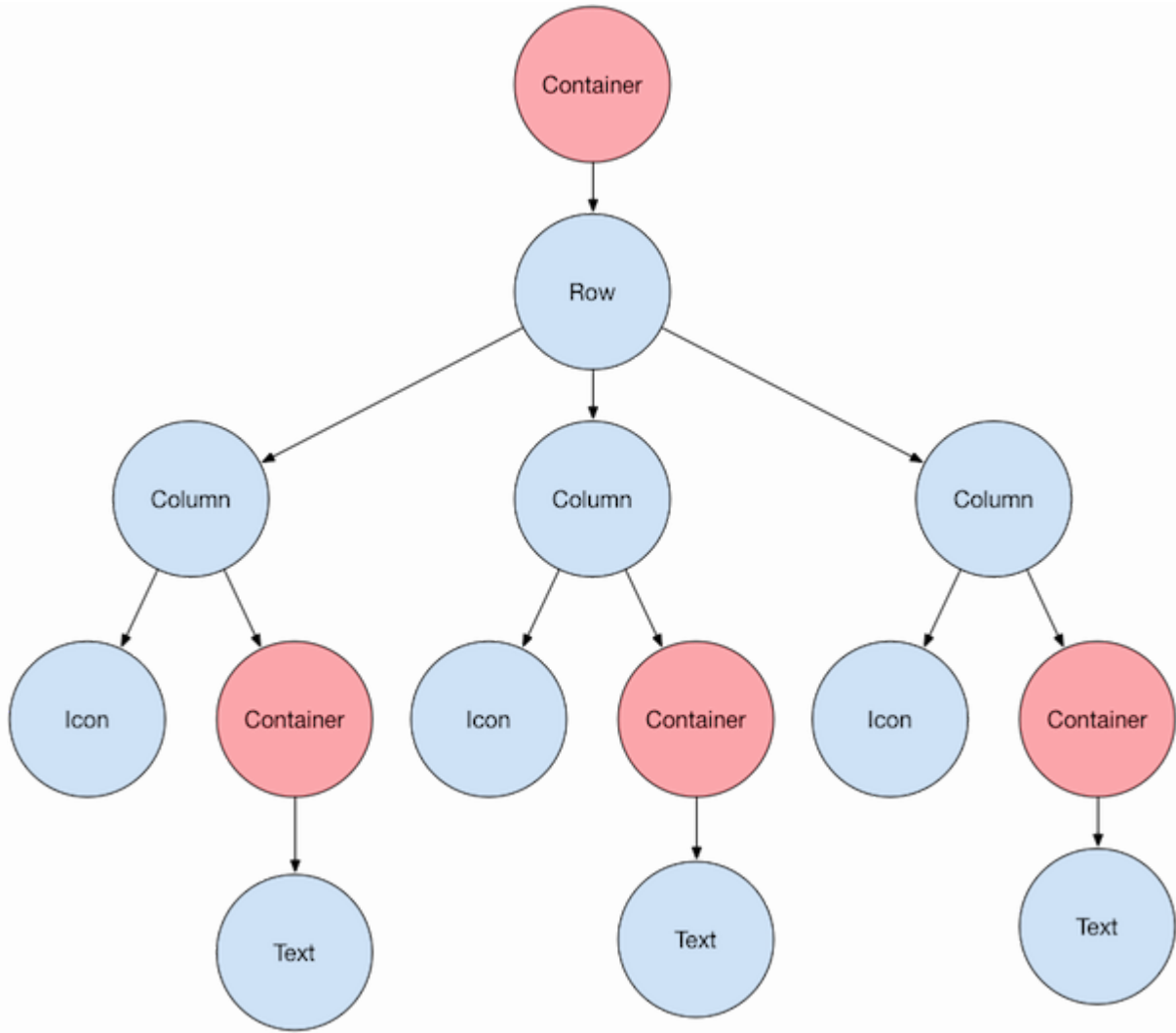
您可以通过构建widget来构建更复杂的widget。例如，下面左边的屏幕截图显示了3个图标，每个图标下有一个标签：



第二个屏幕截图显示布局结构，显示一个行包含3列，其中每列包含一个图标和一个标签。

注意: 本教程中的大多数屏幕截图都是在debugPaintSizeEnabled为true时显示的，因此您可以看到布局结构。 有关更多信息，请参阅[可视化调试](#)，这是[调试 Flutter Apps](#)中的一节。

以下是此UI的widget树示意图：



大部分应该看起来应该像您所期望的，但您可能想了解一下Container（以粉红色显示）。 Container也是一个widget，允许您自定义其子widget。如果要添加填充， 边距， 边框或背景色， 请使用Container来设置（译者语：只有容器有这些属性）。

在这个例子中， 每个Text放置在Container中以添加边距。 整个行也被放置在容器中以在行的周围添加填充。

本例UI中的其他部分也可以通过属性来控制。 使用其 color 属性设置图标的颜色。 使用Text的 style 属性来设置字体， 颜色， 粗细等。 列和行的属性允许您指定他们的子项如何垂直或水平对齐， 以及应该占据多少空间。

布局一个 widget

重点是什么？

- 即使应用程序本身也是一个 widget.
- 创建一个widget并将其添加到布局widget中是很简单的.
- 要在设备上显示widget， 请将布局widget添加到 app widget中。
- 使用Scaffold是最容易的， 它是 Material Components库中的一个widget， 它提供了一个默认banner， 背景颜色， 并且具有添加drawer， snack bar和底部sheet的API。
- widget widget

如果您愿意，可以构建仅使用标准 库中的 来构建您的应用程序

如何在Flutter中布局单个widget？本节介绍如何创建一个简单的widget并将其显示在屏幕上。它还展示了一个简单的Hello World应用程序的完整代码。

在Flutter中，只需几个步骤即可在屏幕上放置文本，图标或图像。

1. 选择一个widget来保存该对象。

根据您想要对齐或约束可见窗口小部件的方式，从各种布局widget中进行选择，因为这些特性通常会传递到所包含的widget中。这个例子使用Center，它可以将内容水平和垂直居中。

2. 创建一个widget来容纳可见对象

注意:Flutter应用程序是用Dart语言编写的。如果您了解Java或类似的面向对象编程语言，Dart会感到非常熟悉。 如果不了解的话，你可以试试 DartPad-一个可以在任何浏览器上使用的交互式Dart playground。 [Dart 语言之旅](#)是一篇介绍Dart语言特性的概述。

例如，创建一个Text widget:

```
new Text('Hello World', style: new TextStyle(fontSize: 32.0))
```

创建一个 Image widget:

```
new Image.asset('images/myPic.jpg', fit: BoxFit.cover)
```

创建一个 Icon widget:

```
new Icon(Icons.star, color: Colors.red[500])
```

3. 将可见widget添加到布局widget.

所有布局widget都有一个 child 属性（例如Center或Container），或者一个 children 属性，如果他们需要一个widget列表（例如Row，Column，ListView或Stack）。

将Text widget添加到Center widget:

```
new Center(  
  child: new Text('Hello World', style: new TextStyle(fontSize: 32.0))
```

4. 将布局widget添加到页面.

Flutter应用本身就是一个widget，大部分widget都有一个build()方法。在应用程序的build方法中创建会在设

备上显示的widget。对于Material应用程序，您可以将Center widget直接添加到 `body` 属性中

```
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new Center(
        child: new Text('Hello World', style: new TextStyle(fontSize: 32.0)),
      ),
    );
  }
}
```

Note: 在设计用户界面时，您可以使用标准widget库中的widget，也可以使用Material Components中的widget。您可以混合使用两个库中的widget，可以自定义现有的widget，也可以构建一组自定义的widget。

对于非Material应用程序，您可以将Center widget添加到应用程序的 `build()` 方法中：

```
// 这个App没有使用Material组件， 如Scaffold。
// 一般来说，app没有使用Scaffold的话，会有一个黑色的背景和一个默认为黑色的文本颜色。
// 这个app，将背景色改为了白色，并且将文本颜色改为了黑色以模仿Material app
import 'package:flutter/material.dart';

void main() {
  runApp(new MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Container(
      decoration: new BoxDecoration(color: Colors.white),
      child: new Center(
        child: new Text('Hello World',
          textDirection: TextDirection.ltr,
          style: new TextStyle(fontSize: 40.0, color: Colors.black87)),
      ),
    );
  }
}
```

请注意，默认情况下，非Material应用程序不包含AppBar，标题或背景颜色。如果您想在非Material应用程序中使用这些功能，您必须自己构建它们。此应用程序将背景颜色更改为白色，将文本更改为深灰色以模

仿Material应用程序。

好了! 当你运行这个应用时，你会看到:



Dart 代码 (Material app): [main.dart](#)
Dart 代码 (仅使用标准Widget的app): [main.dart](#)

垂直和水平放置多个widget

最常见的布局模式之一是垂直或水平排列widget。您可以使用行(Row)水平排列widget，并使用列 (Column) 垂直排列widget。

重点是什么？

- 行和列是两种最常用的布局模式。
- 行和列都需要一个子widget列表。
- 子widget本身可以是行、列或其他复杂widget。
- 您可以指定行或列如何在垂直或水平方向上对齐其子项
- 您可以拉伸或限制特定的子widget.

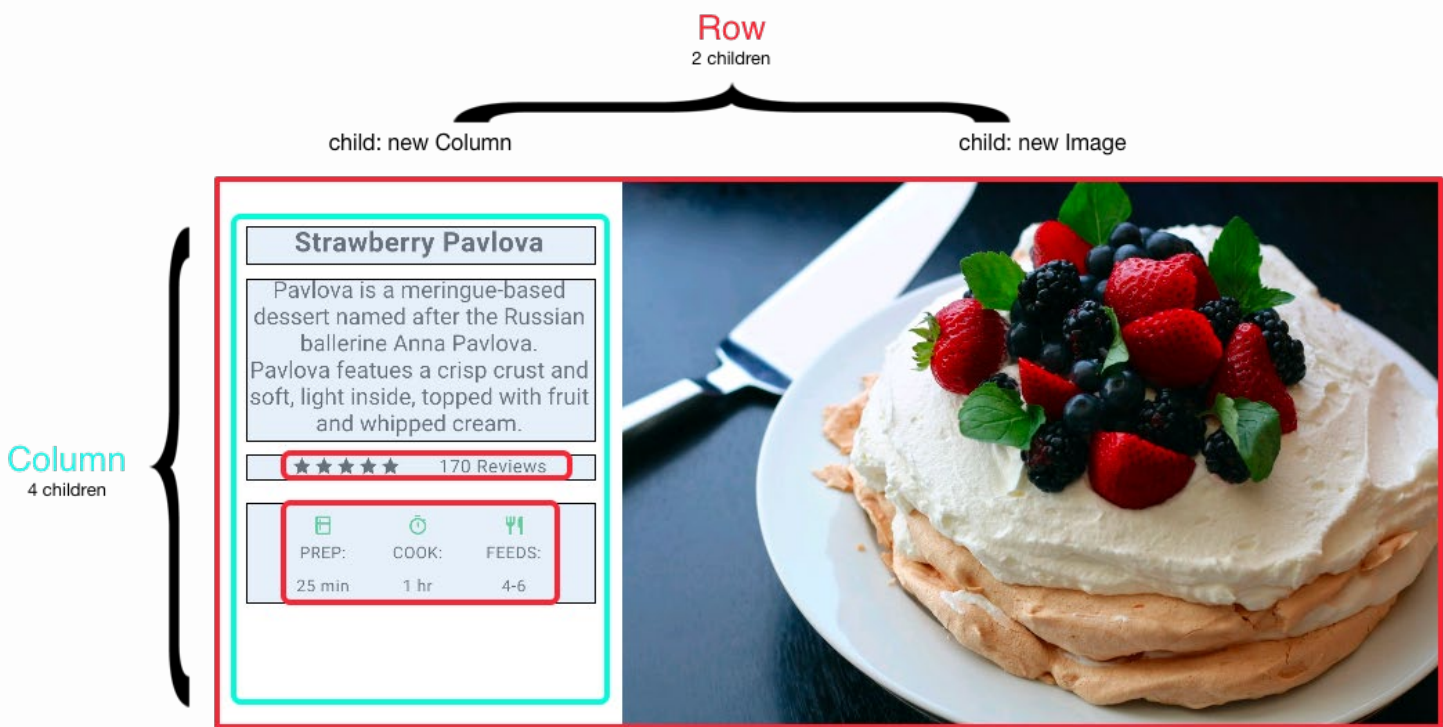
- 您可以指定子widget如何使用行或列的可用空间.

Contents

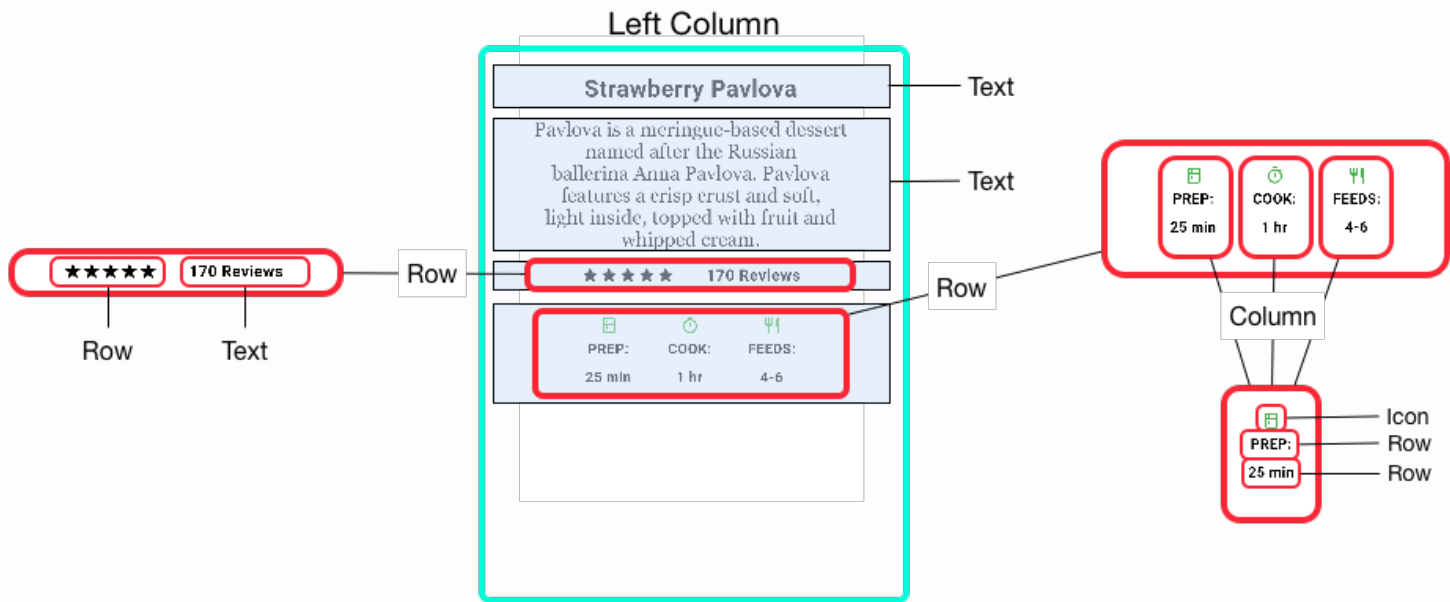
- 对齐 widgets
- 调整 widgets
- 聚集 widgets
- 嵌套行和列

要在Flutter中创建行或列，可以将一个widget列表添加到Row 或Column 中。同时，每个孩子本身可以是一个Row或一个Column，依此类推。以下示例显示如何在行或列内嵌套行或列。

此布局按行组织。该行包含两个孩子：左侧的一列和右侧的图片：



左侧的Column widget树嵌套行和列。

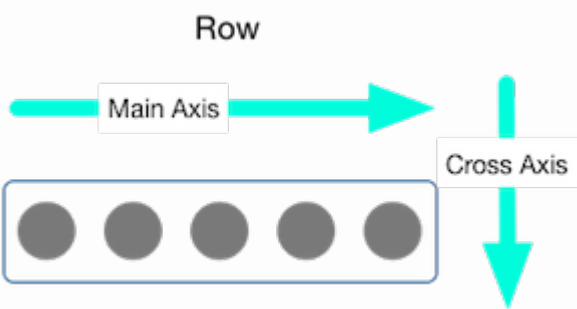


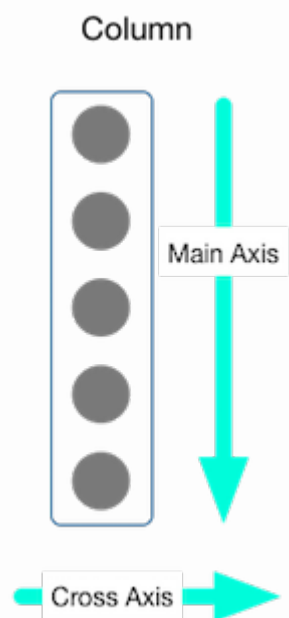
您将在[嵌套行和列](#)中实现一些Pavlova（图片中的奶油水果蛋白饼） 的布局代码

注意:行和列是水平和垂直布局的基本、低级widget - 这些低级widget允许最大化的自定义。Flutter还提供专门的, 更高级别的widget, 可能足以满足您的需求。 例如, 您可能更喜欢[ListTile](#)而不是Row, [ListTile](#)是一个易于使用的小部件, 具有前后图标属性以及最多3行文本。您可能更喜欢[ListView](#)而不是列, [ListView](#)是一种列状布局, 如果其内容太长而无法适应可用空间, 则会自动滚动。有关更多信息, 请参阅[通用布局widget](#)。

对齐 widgets

您可以控制行或列如何使用 `mainAxisAlignment` 和 `crossAxisAlignment` 属性来对齐其子项。对于行(Row)来说, 主轴是水平方向, 横轴垂直方向。对于列 (Column) 来说, 主轴垂直方向, 横轴水平方向。





`MainAxisAlignment` 和 `CrossAxisAlignment` 类提供了很多控制对齐的常量。

注意: 将图片添加到项目时，需要更新`pubspec`文件才能访问它们 - 此示例使用`Image.asset`显示图像。有关更多信息，请参阅此示例的[pubspec.yaml](#)文件，或在Flutter中添加资源和图像。如果您使用的是网上的图片，则不需要执行此操作，使用`Image.network`即可。

在以下示例中，3个图像中的每一个都是100像素宽。渲染盒（在这种情况下，整个屏幕）宽度超过300个像素，因此设置主轴对齐方式为 `spaceEvenly`，它会在每个图像之间，之前和之后均匀分配空闲的水平空间。

```
appBar: new AppBar(
  title: new Text(widget.title),
),
body: new Center(
  child: new Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
      new Image.asset('images/pic1.jpg'),
```



Dart code: [main.dart](#)

Images: [images](#)

Pubspec: [pubspec.yaml](#)

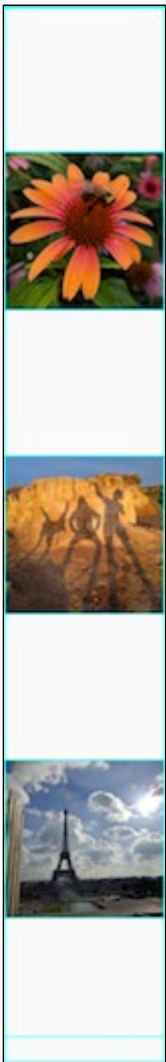
列的工作方式与行相同。以下示例显示了一列，包含3个图片，每个图片高100个像素。渲染盒（在这种情况下，整个屏幕）的高度大于300像素，因此设置主轴对齐方式为 `spaceEvenly`，它会在每个图像之间，上方和下方均匀分配空闲的垂直空间。

```
appBar: new AppBar(  
  title: new Text(widget.title),  
) ,  
body: new Center(  
  child: new Column(  
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
    children: [  
      new Image.asset('images/pic1.jpg'),
```

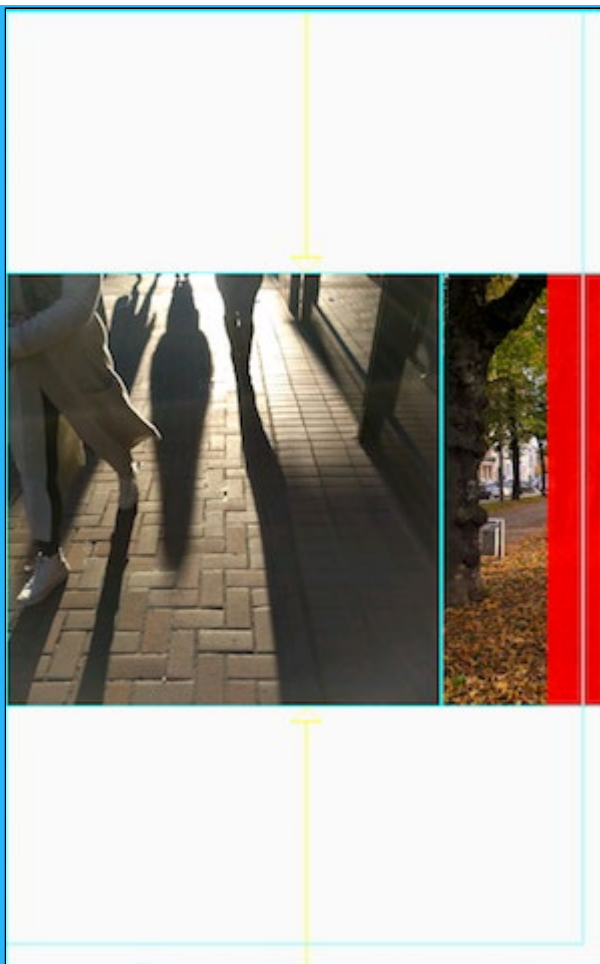
Dart code: [main.dart](#)

Images: [images](#)

Pubspec: [pubspec.yaml](#)



*注意: 如果布局太大而不适合设备, 则会在受影响的边缘出现红色条纹。例如, 以下截图中的行对于设备的屏幕来说太宽:



通过使用Expanded widget, 可以将widget的大小设置为适和行或列, 这在下面的[调整 widgets](#) 部分进行了描述。

调整 widget

也许你想要一个widget占据其兄弟widget两倍的空间。您可以将行或列的子项放置在Expandedwidget中, 以控制沿着主轴方向的widget大小。Expanded widget具有一个flex属性, 它是一个整数, 用于确定widget的弹性系数,默认弹性系数是1。

例如, 要创建一个由三个widget组成的行, 其中中间widget的宽度是其他两个widget的两倍, 将中间widget的弹性系数设置为2:

```
appBar: new AppBar(  
  title: new Text(widget.title),  
) ,  
body: new Center(  
  child: new Row(  
    crossAxisAlignment: CrossAxisAlignment.center,  
    children: [  
      new Expanded(  
        child: new Image.asset('images/pic1.jpg'),  
      ),  
      new Expanded(  
        flex: 2,  
        child: new Image.asset('images/pic2.jpg'),  
      ),  
    ],  
  ),  
)
```

```
),  
  new Expanded(
```



Dart code: [main.dart](#)

Images: [images](#)

Pubspec: [pubspec.yaml](#)

要修复上一节中的示例：其中一行有3张图片，行对于其渲染框太宽，并且导致右边出现红色条中的问题，可以使用Expanded widget来包装每个widget。默认情况下，每个widget的弹性系数为1，将行的三分之一分配给每个小部件。

```
appBar: new AppBar(  
  title: new Text(widget.title),  
) ,  
body: new Center(  
  child: new Row(  
    crossAxisAlignment: CrossAxisAlignment.center,  
    children: [  
      new Expanded(  
        child: new Image.asset('images/pic1.jpg'),  
      ),  
      new Expanded(  
        child: new Image.asset('images/pic2.jpg'),  
      ),  
      new Expanded(  
        child: new Image.asset('images/pic3.jpg'),  
      ),  
    ],  
  ),  
)
```



Dart code: [main.dart](#)

Images: [images](#)

Pubspec: [pubspec.yaml](#)

聚集 widgets

默认情况下，行或列沿着其主轴会尽可能占用尽可能多的空间，但如果要将孩子紧密聚集在一起，可以将mainAxisSize设置为MainAxisSize.min。 以下示例使用此属性将星形图标聚集在一起（如果不聚集，五张星形图标会分散开）。

```
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    var packedRow = new Row(
      mainAxisAlignment: MainAxisAlignment.min,
      children: [
        new Icon(Icons.star, color: Colors.green[500]),
        new Icon(Icons.star, color: Colors.green[500]),
        new Icon(Icons.star, color: Colors.green[500]),
        new Icon(Icons.star, color: Colors.black),
        new Icon(Icons.star, color: Colors.black),
      ],
    );

    // ...
  }
}
```



Dart code: [main.dart](#)
Icons: [Icons class](#)
Pubspec: [pubspec.yaml](#)

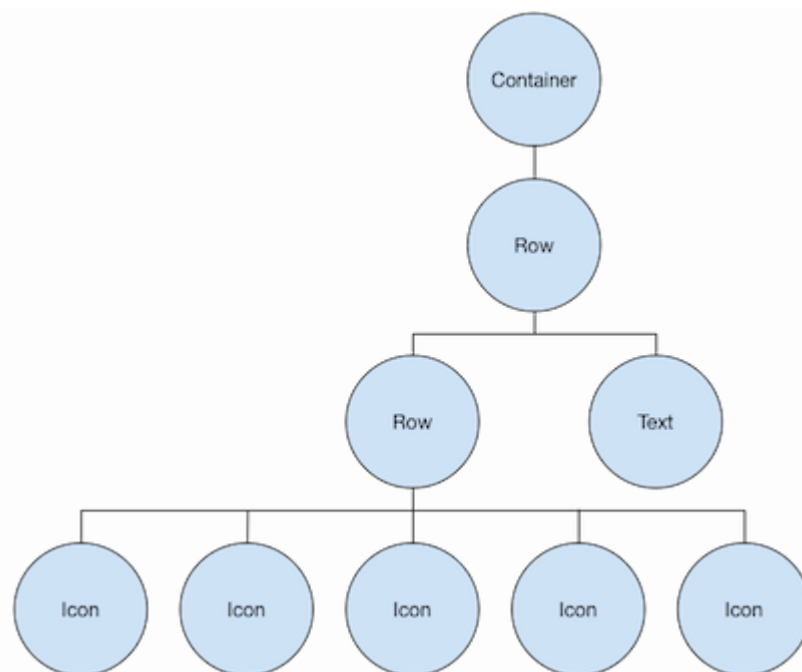
嵌套行和列

布局框架允许您根据需要在行和列内部再嵌套行和列。让我们看下面红色边框圈起来部分的布局代码：



红色边框部分的布局通过两个行来实现。评级行包含五颗星和评论数量。图标行包含三列图标和文本。

评级行的widget树:



该 `ratings` 变量创建一个包含5个星形图标和一个文本的行：

```

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    //...

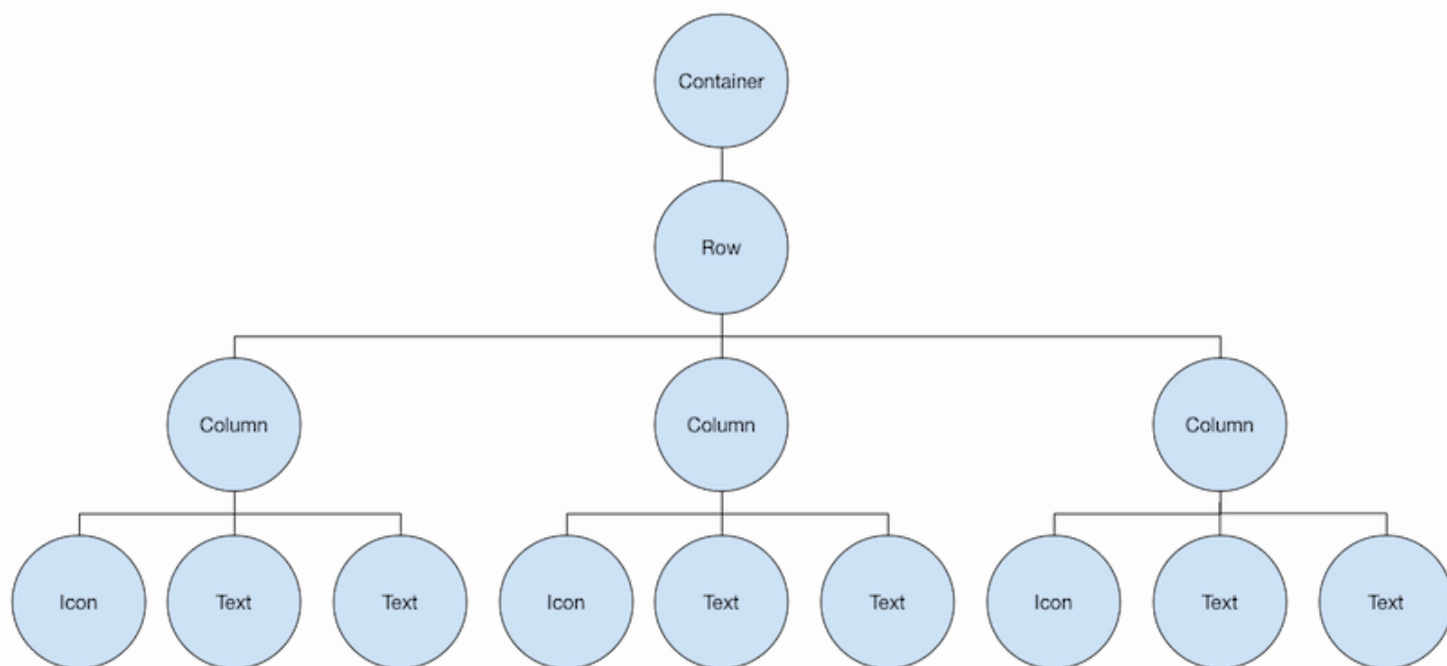
    var ratings = new Container(
      padding: new EdgeInsets.all(20.0),
      child: new Row(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: [
          new Row(
            mainAxisAlignment: MainAxisAlignment.min,
            children: [
              new Icon(Icons.star, color: Colors.black),
              new Icon(Icons.star, color: Colors.black),
              new Icon(Icons.star, color: Colors.black),
              new Icon(Icons.star, color: Colors.black),
              new Icon(Icons.star, color: Colors.black),
            ],
          ),
          new Text(
            '170 Reviews',
            style: new TextStyle(
              color: Colors.black,
              fontWeight: FontWeight.w800,
              fontFamily: 'Roboto',
              letterSpacing: 0.5,
              fontSize: 20.0,
            ),
          ),
        ],
      ),
    ),
  },
}

```

```
);  
//...  
}  
}
```

提示: 为了最大限度地减少由嵌套严重的布局代码导致的视觉混淆, 可以在变量和函数中实现UI的各个部分。

评级行下方的图标行包含3列; 每个列都包含一个图标和两行文本, 您可以在其widget树中看到:



该 `iconList` 变量定义了图标行:

```
class _MyHomePageState extends State<MyHomePage> {  
  @override  
  Widget build(BuildContext context) {  
    // ...  
  
    var descTextStyle = new TextStyle(  
      color: Colors.black,  
      fontWeight: FontWeight.w800,  
      fontFamily: 'Roboto',  
      letterSpacing: 0.5,  
      fontSize: 18.0,  
      height: 2.0,  
    );  
  
    // DefaultTextStyle.merge允许您创建一个默认的文本样式, 该样式会被其  
    // 所有的子节点继承  
    var iconList = DefaultTextStyle.merge(  
      style: descTextStyle,  
      child: new Container(  
        padding: new EdgeInsets.all(20.0),  

```

```

        child: new Row(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            new Column(
              children: [
                new Icon(Icons.kitchen, color: Colors.green[500]),
                new Text('PREP:'),
                new Text('25 min'),
              ],
            ),
            new Column(
              children: [
                new Icon(Icons.timer, color: Colors.green[500]),
                new Text('COOK:'),
                new Text('1 hr'),
              ],
            ),
            new Column(
              children: [
                new Icon(Icons.restaurant, color: Colors.green[500]),
                new Text('FEEDS:'),
                new Text('4-6'),
              ],
            ),
          ],
        ),
      ),
    ),
  );
  // ...
}

```

该 `leftColumn` 变量包含评分和图标行，以及描述Pavlova的标题和文字：

```

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    //...

    var leftColumn = new Container(
      padding: new EdgeInsets.fromLTRB(20.0, 30.0, 20.0, 20.0),
      child: new Column(
        children: [
          titleText,
          subTitle,
          ratings,
          iconList,
        ],
      ),
    );
    //...
  }
}

```

```
}
}
```

左列放置在容器中以约束其宽度。最后，用整个行（包含左列和图像）放置在一个Card内构建UI：

Pavlova图片来自 [Pixabay](#)，可以在Creative Commons许可下使用。 您可以使用Image.network直接从网上下载显示图片，但对于此示例，图像保存到项目中的图像目录中，添加到pubspec文件， 并使用Images.asset。 有关更多信息，请参阅[在Flutter中添加Asserts和图片](#)。

```
body: new Center(
  child: new Container(
    margin: new EdgeInsets.fromLTRB(0.0, 40.0, 0.0, 30.0),
    height: 600.0,
    child: new Card(
      child: new Row(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          new Container(
            width: 440.0,
            child: leftColumn,
          ),
          mainImage,
        ],
      ),
    ),
  ),
),
```

- Dart code: [main.dart](#)
- Images: [images](#)
- Pubspec: [pubspec.yaml](#)

提示: Pavlova示例在广泛的横屏设备（如平板电脑）上运行最佳。如果您在iOS模拟器中运行此示例， 则可以使用**Hardware > Device**菜单选择其他设备。对于这个例子，我们推荐iPad Pro。 您可以使用**Hardware > Rotate**将其方向更改为横向模式 。您还可以使用**Window > Scale**更改模拟器窗口的大小（不更改逻辑像素的数量）

常用布局widgets

Flutter拥有丰富的布局widget，但这里有一些最常用的布局widget。其目的是尽可能快地让您构建应用并运行，而不是让您淹没在整个完整的widget列表中。 有关其他可用widget的信息，请参阅[widget概述](#)，或使用[API 参考 docs](#)文档中的搜索框。 此外，API文档中的widget页面经常会推荐一些可能更适合您需求的类似widget。

以下widget分为两类：[widgets library](#)中的标准widget和[Material Components library](#)中的专用widget 。 任何应用程序都可以使用widgets library中的widget，但只有Material应用程序可以使用Material Components库。

标准 widgets

- **Container**

添加 padding, margins, borders, background color, 或将其他装饰添加到widget.

- **GridView**

将 widgets 排列为可滚动的网格.

- **ListView**

将widget排列为可滚动列表

- **Stack**

将widget重叠在另一个widget之上.

Material Components

- **Card**

将相关内容放到带圆角和投影的盒子中。

- **ListTile**

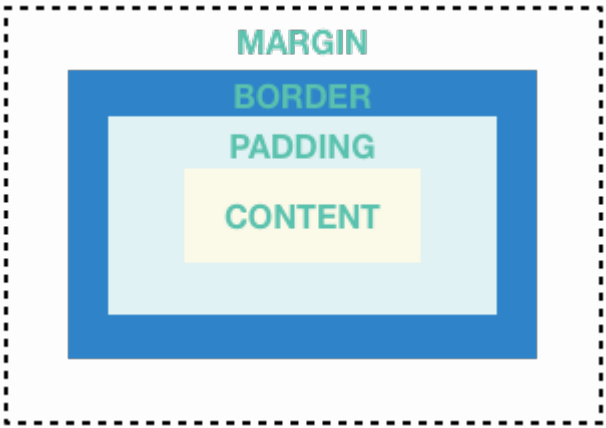
将最多3行文字，以及可选的行前和行尾的图标排成一行

Container

许多布局会自由使用容器来使用padding分隔widget，或者添加边框（border）或边距（margin）。您可以通过将整个布局放入容器并更改其背景颜色或图片来更改设备的背景。

Container 概要：

- 添加padding, margins, borders
- 改变背景颜色或图片
- 包含单个子widget，但孩子widget可以是Row，Column，甚至是widget树的根



Container 示例:

除了下面的例子之外，本教程中的许多示例都使用了Container。您还可以在[Flutter Gallery](#)中找到更多容器示例。

该布局中每个图像使用一个Container来添加一个圆形的灰色边框和边距。然后使用容器将列背景颜色更改为浅灰色。

Dart code: [main.dart](#), snippet below

Images: [images](#)

Pubspec: [pubspec.yaml](#)



```
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {

    var container = new Container(
      decoration: new BoxDecoration(
        color: Colors.black26,
      ),
      child: new Column(
        children: [
          new Row(
            children: [
              new Expanded(
                child: new Container(
                  decoration: new BoxDecoration(
                    border: new Border.all(width: 10.0, color: Colors.black38),
                    borderRadius:
                      const BorderRadius.all(const Radius.circular(8.0)),
                  ),
                  margin: const EdgeInsets.all(4.0),
                  child: new Image.asset('images/pic1.jpg'),
                ),
              ),
              new Expanded(
                child: new Container(
                  decoration: new BoxDecoration(
                    border: new Border.all(width: 10.0, color: Colors.black38),
                    borderRadius:
                      const BorderRadius.all(const Radius.circular(8.0)),
                  ),
                  margin: const EdgeInsets.all(4.0),
                  child: new Image.asset('images/pic2.jpg'),
                ),
              ),
            ],
          ),
          // ...
          // See the definition for the second row on GitHub:
          // https://raw.githubusercontent.com/flutter/website/master/_includes/code/lay
```

```
out/container/main.dart

    ],
  ),
);
//...
}
```

GridView

使用[GridView](#)将widget放置为二维列表。 [GridView](#)提供了两个预制list， 或者您可以构建自定义网格。 当[GridView](#)检测到其内容太长而不适合渲染框时， 它会自动滚动。

[GridView](#) 概览:

- 在网格中放置widget
- 检测列内容超过渲染框时自动提供滚动
- 构建您自己的自定义grid， 或使用一下提供的grid之一:
 - `GridView.count` 允许您指定列数
 - `GridView.extent` 允许您指定项的最大像素宽度

注意: 在显示二维列表时， 重要的是单元格占用哪一行和哪一列时， 应该使用[Table](#)或 [DataTable](#)。

[GridView](#) 示例:



使用 `GridView.extent` 创建最大宽度为150像素的网格

Dart code: [main.dart](#), snippet below

Images: [images](#)

Pubspec: [pubspec.yaml](#)



使用 `GridView.count` 在纵向模式下创建两个行的grid， 并在横向模式下创建3个行的网格。 通过为每个GridTile设置 `footer` 属性来创建标题。

Dart code: [grid_list_demo.dart](#) from the [Flutter Gallery](#)

```
// The images are saved with names pic1.jpg, pic2.jpg...pic30.jpg.
// The List.generate constructor allows an easy way to create
// a list when objects have a predictable naming pattern.

List<Container> _buildGridTileList(int count) {

  return new List<Container>.generate(
    count,
```

```
(int index) =>
    new Container(child: new Image.asset('images/pic${index+1}.jpg')));
}

Widget buildGrid() {
  return new GridView.extent(
    maxCrossAxisExtent: 150.0,
    padding: const EdgeInsets.all(4.0),
    mainAxisSpacing: 4.0,
    crossAxisSpacing: 4.0,
    children: _buildGridTileList(30));
}

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new Center(
        child: buildGrid(),
      ),
    );
  }
}
```

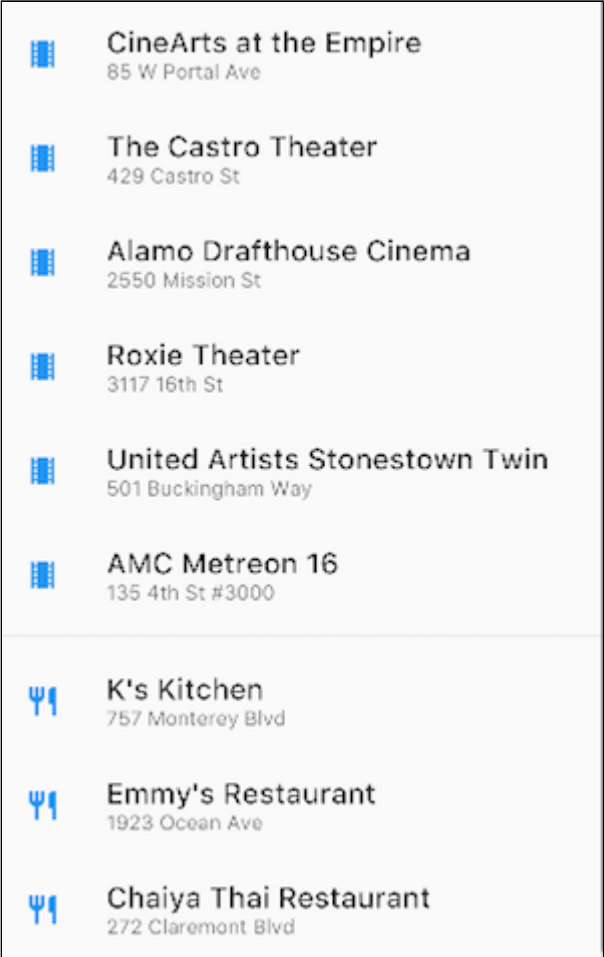
ListView

ListView是一个类似列的widget，它的内容对于其渲染框太长时会自动提供滚动。

ListView 摘要:

- 用于组织盒子中列表的特殊Column
- 可以水平或垂直放置
- 检测它的内容超过显示框时提供滚动
- 比Column配置少，但更易于使用并支持滚动

ListView 示例:



使用ListView显示多个ListTile的业务列表。分隔线将剧院与餐厅分开

Dart code: [main.dart](#), snippet below

Icons: [Icons class](#)

Pubspec: [pubspec.yaml](#)

DEEP PURPLE	INDIGO	BLUE	LIGHT BLUE	CYAN
50				#FFE3F2FD
100				#FFBBDEFB
200				#FF90CAF9
300				#FF64B5F6
400				#FF42A5F5
500				#FF2196F3
600				#FF1E88E5
700				#FF1976D2
800				#FF1565C0
900				#FF0D47A1
A100				#FF82B1FF
A200				#FF448AFF
A400				#FF2979FF

使用ListView控件来显示Material Design palette中的Colors

Dart code: Flutter Gallery中的 colors_demo.dart

```
List<Widget> list = <Widget>[
  new ListTile(
    title: new Text('CineArts at the Empire',
      style: new TextStyle(fontWeight: FontWeight.w500, fontSize: 20.0)),
    subtitle: new Text('85 W Portal Ave'),
    leading: new Icon(
      Icons.theaters,
      color: Colors.blue[500],
    ),
  ),
  new ListTile(
    title: new Text('The Castro Theater',
      style: new TextStyle(fontWeight: FontWeight.w500, fontSize: 20.0)),
    subtitle: new Text('429 Castro St'),
    leading: new Icon(
      Icons.theaters,
      color: Colors.blue[500],
    ),
  ),
  // ...
  // See the rest of the column defined on GitHub:
  // https://raw.githubusercontent.com/flutter/website/master/_includes/code/layout/list
  view/main.dart
```

```
];

class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      // ...
      body: new Center(
        child: new ListView(
          children: list,
        ),
      ),
    );
  }
}
```

Stack

使用Stack来组织需要重叠的widget。 widget可以完全或部分重叠底部widget。

Stack summary:

- 用于与另一个widget重叠的widget
- 子列表中的第一个widget是base widget; 随后的子widget被覆盖在基础widget的顶部
- Stack的内容不能滚动
- 您可以选择剪切超过渲染框的子项

Stack 示例:



使用Stack叠加Container（在半透明的黑色背景上显示其文本）， 放置在Circle Avatar的顶部。 Stack使用alignment属性和调整文本偏移。

Dart code: [main.dart](#), snippet below

Image: [images](#)

Pubspec: [pubspec.yaml](#)



使用Stack将gradient叠加到图像的顶部。 gradient确保工具栏的图标与图片不同。

Dart code: [contacts_demo.dart](#)

```
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    var stack = new Stack(
      alignment: const Alignment(0.6, 0.6),
      children: [
```

```
new CircleAvatar(  
  backgroundImage: new AssetImage('images/pic.jpg'),  
  radius: 100.0,  
) ,  
new Container(  
  decoration: new BoxDecoration(  
    color: Colors.black45,  
  ) ,  
  child: new Text(  
    'Mia B',  
    style: new TextStyle(  
      fontSize: 20.0,  
      fontWeight: FontWeight.bold,  
      color: Colors.white,  
    ) ,  
  ) ,  
) ,  
) ,  
],  
);  
// ...  
}
```

Card

Material Components 库中的**Card**包含相关内容块，可以由大多数类型的**widget**构成，但通常与**ListTile**一起使用。**Card**有一个子项，但它可以是支持多个子项的列，行，列表，网格或其他小部件。默认情况下，**Card**将其大小缩小为0像素。您可以使用**SizedBox**来限制**Card**的大小。

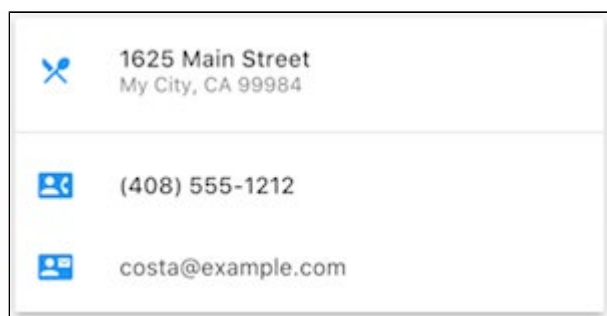
在Flutter中，**Card**具有圆角和阴影，这使它有一个3D效果。更改**Card**的 `elevation` 属性允许您控制投影效果。例如，将 `elevation` 设置为24.0，将会使**Card**从视觉上抬离表面并使阴影变得更加分散。有关支持的 `elevation` 值的列表，请参见**Material guidelines**中的**Elevation and Shadows**。如果指定不支持的值将会完全禁用投影。

Card 摘要:

- 实现了一个 **Material Design card**
- 接受单个子项，但该子项可以是Row，Column或其他包含子级列表的widget
- 显示圆角和阴影
- Card内容不能滚动
- **Material Components** 库的一个widget

Card 示例:

包含3个ListTiles并通过用SizedBox包装进行大小调整的Card。



分隔线分隔第一个和第二个ListTiles。

Dart code: [main.dart](#), snippet below

Icons: [Icons class](#)

Pubspec: [pubspec.yaml](#)



包含图像和文字的Card

Dart code: [cards_demo.dart](#) from the [Flutter Gallery](#)

```
class _MyHomePageState extends State<MyHomePage> {
  @override
  Widget build(BuildContext context) {
    var card = new SizedBox(
      height: 210.0,
      child: new Card(
        child: new Column(
          children: [
            new ListTile(
              title: new Text('1625 Main Street',
                style: new TextStyle(fontWeight: FontWeight.w500)),
              subtitle: new Text('My City, CA 99984'),
              leading: new Icon(
                Icons.restaurant_menu,
                color: Colors.blue[500],
              ),
            ),
            new Divider(),
            new ListTile(
              title: new Text('(408) 555-1212',
                style: new TextStyle(fontWeight: FontWeight.w500)),
```



```
        leading: new Icon(
          Icons.contact_phone,
          color: Colors.blue[500],
        ),
      ),
      new ListTile(
        title: new Text('costa@example.com'),
        leading: new Icon(
          Icons.contact_mail,
          color: Colors.blue[500],
        ),
      ),
    ],
  ),
);
//...
```

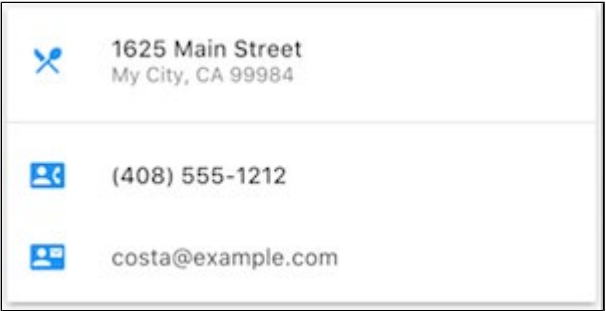
ListTile

ListTile是Material Components库中的一个专门的行级widget，用于创建包含最多3行文本和可选的行前和行尾图标的行。ListTile在Card或ListView中最常用，但也可以在别处使用。

ListTile 摘要:

- 包含最多3行文本和可选图标的专用行
- 比起Row不易配置，但更易于使用
- Material Components 库里的widget

ListTile 示例:



包含3个ListTiles的Card

Dart code: See [Card examples](#).



使用ListTile列出3个下拉按钮类型。

Dart code: [buttons_demo.dart](#) from the [Flutter Gallery](#)

资源

编写布局代码时以下资源可能会有所帮助。

- [Widget 概述](#)
介绍Flutter中提供的许多widgets。
- [HTML/CSS在Flutter中的模拟](#)
对于熟悉web开发的人员，本页将HTML / CSS功能映射到Flutter功能。
- [Flutter Gallery](#)
一个Demo应用，展示了许多Material Design widget和其他Flutter功能。
- [Flutter API documentation](#)
所有Flutter库的参考文档。
- [处理Flutter中的框约束](#)
讨论widgets如何受其渲染框限制。
- [在Flutter中添加 Assets 和 Images](#)
说明如何将图像和其他资源添加到应用程序包中。
- [从零到一开始 Flutter](#)
一个写他第一个Flutter应用程序的经验。



[关于本站](#) [加入我们](#) [Gitme](#) [Dart中文网](#) [Flutter中文网开源计划](#) [译者博客](#) | [Github](#)

京ICP备14014371号-3

为您的Flutter应用程序添加交互

[编辑本页](#)[提Issue](#)

你将会学到:

- 如何响应点击(`tap`).
- 如何创建自定义`widget`.
- `stateless` (无状态) 和 `stateful` (有状态) `widgets`的区别.

如何修改你的应用程序, 使其对用户动作做出反应? 在本教程中, 您将为`widget`添加交互。具体来说, 您将通过创建一个管理两个无状态`widget`的自定义有状态的`widget`来使图标可以点击。

内容

- [Stateful 和 stateless widgets](#)
- [创建一个stateful widget](#)
 - [第1步: 决定哪个widget管理widget的状态\(state\)](#)
 - [第2步: 创建StatefulWidget子类](#)
 - [第3步: 创建State子类](#)
 - [第4步: 将stateful widget 插入到 widget 树](#)
 - [遇到问题?](#)
- [管理状态](#)
 - [widget管理自己的state](#)
 - [父组件管理state](#)
 - [混合管理state](#)
- [其它交互式widgets](#)
 - [标砖widgets](#)
 - [Material Components](#)
- [资源](#)

准备

如果您已经根据[在Flutter中构建布局](#)一节的构建好了布局, 请跳过本块。

- 确保您已经[配置](#)好了Flutter开发环境.
- [创建一个基础的Flutter app](#).
- 用Github上的 `main.dart` 替换本地工程 `lib/main.dart`
- 用Github上的 `pubspec.yaml` 替换本地工程 `pubspec.yaml`
- 在你的工程中创建一个 `images` 文件夹, 并添加 `lake.jpg`.

一旦你有一个连接和启用的设备，或者你已经启动了[iOS模拟器](#)（Flutter安装一节介绍过），就会很容易开始！ </aside>

在[Flutter中构建布局](#)一节展示了如何构建下面截图所示的布局。



Oeschinen Lake Campground

Kandersteg, Switzerland

★ 41



CALL



ROUTE



SHARE

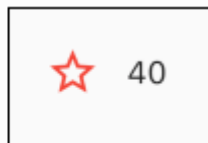
Lake Oeschinen lies at the foot of the Blüemlisalp in the Bernese Alps. Situated 1,578 meters above sea level, it is one of the larger Alpine Lakes. A gondola ride from Kandersteg, followed by a half-hour walk through pastures and pine forest, leads you to the lake, which warms to 20 degrees Celsius in the summer. Activities enjoyed here include rowing, and riding the summer toboggan run.

当应用第一次启动时，这颗星形图标是实心红色，表明这个湖以前已经被收藏了。星号旁边的数字表示41个人对此湖感兴趣。完成本教程后，点击星形图标将取消收藏状态，然后用轮廓线的星形图标代替实心的，并减

少计数。再次点击会重新收藏，并增加计数。



Favorited



Not favorited

为了实现这个，您将创建一个包含星号和计数的自定义widget，它们都是widget。因为点击星形图标会更改这两个widget的状态，所以同一个widget应该同时管理这两个widget。

Stateful（有状态）和 stateless（无状态） widgets

什么是重点？

- 有些widgets是有状态的, 有些是无状态的
- 如果用户与widget交互，widget会发生变化，那么它就是有状态的。
- widget的状态（state）是一些可以更改的值, 如一个slider滑动条的当前值或checkbox是否被选中。
- widget的状态保存在一个State对象中, 它和widget的布局显示分离。
- 当widget状态改变时, State 对象调用 `setState()`，告诉框架去重绘widget。

stateless widget 没有内部状态. `Icon`、`IconButton`, 和 `Text` 都是无状态widget, 他们都是 `StatelessWidget` 的子类。

stateful widget 是动态的. 用户可以和其交互 (例如输入一个表单、 或者移动一个slider滑块), 或者可以随时间改变 (也许是数据改变导致的UI更新). `Checkbox`, `Radio`, `Slider`, `InkWell`, `Form`, and `TextField` 都是 *stateful* widgets, 他们都是 `StatefulWidget` 的子类。

创建一个有状态的widget

重点：

- 要创建一个自定义有状态widget，需创建两个类：`StatefulWidget`和`State`
- 状态对象包含widget的状态和`build()` 方法。
- 当widget的状态改变时，状态对象调用 `setState()`，告诉框架重绘widget

在本节中，您将创建一个自定义有状态的widget。您将使用一个自定义有状态widget来替换两个无状态widget - 红色实心星形图标和其旁边的数字计数 - 该widget用两个子widget管理一行：`IconButton`和`Text`。

实现一个自定义的有状态widget需要创建两个类：

- 定义一个widget类，继承自`StatefulWidget`。
- 包含该widget状态并定义该widget `build()` 方法的类，它继承自`State`。

本节展示如何为Lakes应用程序构建一个名为`FavoriteWidget`的`StatefulWidget`。第一步是选择如何管

理FavoriteWidget的状态。

Step 1: 决定哪个对象管理widget的状态

Widget的状态可以通过多种方式进行管理，但在我们的示例中，widget本身（FavoriteWidget）将管理自己的状态。在这个例子中，切换星形图标是一个独立的操作，不会影响父窗口widget或其他用户界面，因此该widget可以在内部处理它自己的状态。

在[管理状态](#)中了解更多关于widget和状态的分离以及如何管理状态的信息。

Step 2: 创建StatefulWidget子类

FavoriteWidget类管理自己的状态，因此它重写 `createState()` 来创建状态对象。框架会在构建widget时调用 `createState()`。在这个例子中，`createState()` 创建 `_FavoriteWidgetState` 的实例，您将在下一步中实现该实例。

```
class FavoriteWidget extends StatefulWidget {  
  @override  
  _FavoriteWidgetState createState() => new _FavoriteWidgetState();  
}
```

注意: 以下划线 (`_`) 开头的成员或类是私有的。有关更多信息，请参阅[Dart语言参考](#)中的[库和可见性](#)部分。

Step 3: 创建State子类

自定义State类存储可变信息 - 可以在widget的生命周期内改变逻辑和内部状态。当应用第一次启动时，用户界面显示一个红色实心的星形图标，表明该湖已经被收藏，并有41个“喜欢”。状态对象存储这些信息在 `_isFavorited` 和 `_favoriteCount` 变量。

状态对象也定义了 `build` 方法。此 `build` 方法创建一个包含红色IconButton和Text的行。该widget使用 `IconButton`（而不是 `Icon`），因为它具有一个 `onPressed` 属性，该属性定义了处理点击的回调方法。 `IconButton` 也有一个 `icon` 的属性，持有 `Icon`。

按下 `IconButton` 时会调用 `_toggleFavorite()` 方法，然后它会调用 `setState()`。调用 `setState()` 是至关重要的，因为这告诉框架，widget的状态已经改变，应该重绘。`_toggleFavorite` 在: 1) 实心的星形图标和数字“41” 和 2) 虚心的星形图标和数字“40”之间切换UI。

```
class _FavoriteWidgetState extends State<FavoriteWidget> {  
  bool _isFavorited = true;  
  int _favoriteCount = 41;  
  
  void _toggleFavorite() {  
    setState(() {  
      // If the lake is currently favorited, unfavorite it.  
      if ( isFavorited) {
```

```

        _favoriteCount -= 1;
        _isFavorited = false;
        // Otherwise, favorite it.
    } else {
        _favoriteCount += 1;
        _isFavorited = true;
    }
  });
}

@override
Widget build(BuildContext context) {
  return new Row(
    mainAxisAlignment: MainAxisAlignment.min,
    children: [
      new Container(
        padding: new EdgeInsets.all(0.0),
        child: new IconButton(
          icon: (_isFavorited
            ? new Icon(Icons.star)
            : new Icon(Icons.star_border)),
          color: Colors.red[500],
          onPressed: _toggleFavorite,
        ),
      ),
      new SizedBox(
        width: 18.0,
        child: new Container(
          child: new Text('$favoriteCount'),
        ),
      ),
    ],
  );
}

```

提示: 当文本在40和41之间变化时, 将文本放在**SizedBox**中并设置其宽度可防止出现明显的“跳跃”, 因为这些值具有不同的宽度。

Step 4: 将有stateful widget插入widget树中

将您自定义stateful widget在build方法中添加到widget树中。首先, 找到创建图标和文本的代码, 并删除它:

```

// ...
new Icon(
  Icons.star,
  color: Colors.red[500],
),
new Text('41')
// ...

```


在相同的位置创建stateful widget:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    Widget titleSection = new Container(
      // ...
      child: new Row(
        children: [
          new Expanded(
            child: new Column(
              // ...
            ),
            new FavoriteWidget(),
          ],
        ),
      );

    return new MaterialApp(
      // ...
    );
  }
}
```

好了! 当您热重载应用程序后, 星形图标就会响应点击了.

遇到问题?

如果您的代码无法运行, 请在IDE中查找可能的错误。调试Flutter应用程序可能会有所帮助。如果仍然无法找到问题, 请根据GitHub上的示例检查代码。

- lib/main.dart
- pubspec.yaml —此文件没有变化
- lakes.jpg —此文件没有变化

本页面的其余部分介绍了可以管理widget状态的几种方式, 并列出了其他可用的可交互的widget。

管理状态

重点是什么?

- 有多种方法可以管理状态.
- 选择使用何种管理方法

如果不是很清楚时, 那就在父widget中管理状态吧.

谁管理着stateful widget的状态? widget本身? 父widget? 都会? 另一个对象? 答案是.....这取决于实际情况。有几种有效的方法可以给你的widget添加互动。作为小部件设计师。以下是管理状态的最常见的方法:

- widget管理自己的state
- 父widget管理 widget状态
- 混搭管理 (父widget和widget自身都管理状态)

如何决定使用哪种管理方法? 以下原则可以帮助您决定:

- 如果状态是用户数据, 如复选框的选中状态、滑块的位置, 则该状态最好由父widget管理
- 如果所讨论的状态是有关界面外观效果的, 例如动画, 那么状态最好由widget本身来管理.

如果有疑问, 首选是在父widget中管理状态

我们将通过创建三个简单示例来举例说明管理状态的不同方式: TapboxA、TapboxB和TapboxC。这些例子功能是相似的 - 每创建一个容器, 当点击时, 在绿色或灰色框之间切换。 `_active` 确定颜色: 绿色为true, 灰色为false。



这些示例使用[GestureDetector](#)捕获Container上的用户动作。

widget管理自己的状态

有时, widget在内部管理其状态是最好的。例如, 当[ListView](#)的内容超过渲染框时, [ListView](#)自动滚动。大多数使用[ListView](#)的开发人员不想管理[ListView](#)的滚动行为, 因此[ListView](#)本身管理其滚动偏移量。

`_TapboxAState` 类:

- 管理TapboxA的状态.
- 定义 `_active`: 确定盒子的当前颜色的布尔值.
- 定义 `_handleTap()` 函数, 该函数在点击该盒子时更新 `_active`, 并调用 `setState()` 更新UI.
- 实现widget的所有交互式行为.

```
// TapboxA 管理自身状态.
```

```
//----- TapboxA -----
```

```

class TapboxA extends StatefulWidget {
  TapboxA({Key key}) : super(key: key);

  @override
  _TapboxAState createState() => new _TapboxAState();
}

class _TapboxAState extends State<TapboxA> {
  bool _active = false;

  void _handleTap() {
    setState(() {
      _active = !_active;
    });
  }

  Widget build(BuildContext context) {
    return new GestureDetector(
      onTap: _handleTap,
      child: new Container(
        child: new Center(
          child: new Text(
            _active ? 'Active' : 'Inactive',
            style: new TextStyle(fontSize: 32.0, color: Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: new BoxDecoration(
          color: _active ? Colors.lightGreen[700] : Colors.grey[600],
        ),
      ),
    );
  }
}

//----- MyApp -----

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Flutter Demo',
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text('Flutter Demo'),
        ),
        body: new Center(
          child: new TapboxA(),
        ),
      ),
    );
  }
}

```

Dart code: `lib/main.dart`

父widget管理widget的state

对于父widget来说，管理状态并告诉其子widget何时更新通常是最有意义的。例如，`IconButton`允许您将图标视为可点按的按钮。`IconButton`是一个无状态的小部件，因为我们认为父widget需要知道该按钮是否被点击来采取相应的处理。

在以下示例中，`TapboxB`通过回调将其状态导出到其父项。由于`TapboxB`不管理任何状态，因此它的父类为`StatelessWidget`。

`ParentWidgetState` 类:

- 为`TapboxB` 管理 `_active` 状态.
- 实现 `_handleTapboxChanged()` , 当盒子被点击时调用的方法.
- 当状态改变时, 调用 `setState()` 更新UI.

`TapboxB` 类:

- 继承 `StatelessWidget` 类, 因为所有状态都由其父widget处理.
- 当检测到点击时, 它会通知父widget.

```
// ParentWidget 为 TapboxB 管理状态.

//----- ParentWidget -----

class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() => new _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return new Container(
      child: new TapboxB(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    ),
  }
}
```

```

    );
  }
}

//----- TapboxB -----

class TapboxB extends StatelessWidget {
  TapboxB({Key key, this.active: false, @required this.onChangeed})
    : super(key: key);

  final bool active;
  final ValueChanged<bool> onChangeed;

  void _handleTap() {
    onChangeed(!active);
  }

  Widget build(BuildContext context) {
    return new GestureDetector(
      onTap: _handleTap,
      child: new Container(
        child: new Center(
          child: new Text(
            active ? 'Active' : 'Inactive',
            style: new TextStyle(fontSize: 32.0, color: Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: new BoxDecoration(
          color: active ? Colors.lightGreen[700] : Colors.grey[600],
        ),
      ),
    );
  }
}

```

Dart code: `lib/main.dart`

提示: 在创建API时, 请考虑使用@required为代码所依赖的任何参数使用注解。 要使用@required注解, 请导入[foundation library](#) (该库重新导出Dart的[meta.dart](#))

```
'package: flutter/foundation.dart';
```

混合管理

对于一些widget来说，混搭管理的方法最有意义的。在这种情况下，有状态widget管理一些状态，并且父widget管理其他状态。

在TapboxC示例中，点击时，盒子的周围会出现一个深绿色的边框。点击时，边框消失，盒子的颜色改变。TapboxC将其 `_active` 状态导出到其父widget中，但在内部管理其 `_highlight` 状态。这个例子有两个状态对象 `_ParentWidgetState` 和 `_TapboxCState`。

`_ParentWidgetState` 对象:

- 管理 `_active` 状态.
- 实现 `_handleTapboxChanged()`，当盒子被点击时调用.
- 当点击盒子并且 `_active` 状态改变时调用 `setState()` 更新UI

`_TapboxCState` 对象:

- 管理 `_highlight` state.
- `GestureDetector` 监听所有tap事件。当用户点下时，它添加高亮（深绿色边框）；当用户释放时，会移除高亮。
- 当按下、抬起、或者取消点击时更新 `_highlight` 状态，调用 `setState()` 更新UI。
- 当点击时，将状态的改变传递给父widget.

```
//----- ParentWidget -----

class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() => new _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return new Container(
      child: new TapboxC(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    );
  }
}

//----- TapboxC -----

class TapboxC extends StatefulWidget {
```

```

TapboxC({Key key, this.active: false, @required this.onChangeed})
  : super(key: key);

final bool active;
final ValueChanged<bool> onChangeed;

_TapboxCState createState() => new _TapboxCState();
}

class _TapboxCState extends State<TapboxC> {
  bool _highlight = false;

  void _handleTapDown(TapDownDetails details) {
    setState(() {
      _highlight = true;
    });
  }

  void _handleTapUp(TapUpDetails details) {
    setState(() {
      _highlight = false;
    });
  }

  void _handleTapCancel() {
    setState(() {
      _highlight = false;
    });
  }

  void _handleTap() {
    widget.onChangeed(!widget.active);
  }

  Widget build(BuildContext context) {
    // This example adds a green border on tap down.
    // On tap up, the square changes to the opposite state.
    return new GestureDetector(
      onTapDown: _handleTapDown, // Handle the tap events in the order that
      onTapUp: _handleTapUp, // they occur: down, up, tap, cancel
      onTap: _handleTap,
      onTapCancel: _handleTapCancel,
      child: new Container(
        child: new Center(
          child: new Text(widget.active ? 'Active' : 'Inactive',
            style: new TextStyle(fontSize: 32.0, color: Colors.white)),
        ),
        width: 200.0,
        height: 200.0,
        decoration: new BoxDecoration(
          color:
            widget.active ? Colors.lightGreen[700] : Colors.grey[600],
          border: _highlight
            ? new Border.all(
                color: Colors.teal[700],

```

```
        width: 10.0,
      ),
      : null,
    ),
  ),
);
}
```

另一种实现可能会将高亮状态导出到父widget，同时保持 `_active` 状态为内部，但如果您要求某人使用该TapBox，他们可能会抱怨说没有多大意义。开发人员只会关心该框是否处于活动状态。开发人员可能不在乎高亮显示是如何管理的，并且倾向于让TapBox处理这些细节。

Dart code: `lib/main.dart`

其他交互式widgets

Flutter提供各种按钮和类似的交互式widget。这些widget中的大多数实现了[Material Design 指南](#)，它们定义了一组具有质感的UI组件。

如果你愿意，你可以使用[GestureDetector](#)来给任何自定义widget添加交互性。您可以在[管理状态](#)和[Flutter Gallery](#)中找到GestureDetector的示例。

注意: Flutter还提供了一组名为[Cupertino](#)的iOS风格的小部件。

When you need interactivity, it's easiest to use one of the prefabricated widgets. Here's a partial list: 当你需要交互性时，最容易的是使用预制的widget。这是预置widget部分列表:

标准 widgets:

- [Form](#)
- [FormField](#)

Material Components:

- [Checkbox](#)
- [DropDownButton](#)
- [FlatButton](#)
- [FloatingActionButton](#)
- [IconButton](#)
- [Radio](#)
- [RaisedButton](#)
- [Slider](#)
- [Switch](#)

- [TextField](#)

资源

给您的应用添加交互时，以下资源可能会有所帮助

- [处理手势](#), [Flutter Widget框架总览](#)的一节
如何创建一个按钮并使其响应用户动作.
- [Flutter中的手势](#)
Flutter手势机制的描述
- [Flutter API 文档](#)
所有Flutter库的参考文档.
- [Flutter Gallery](#)
一个Demo应用程序，展示了许多Material Components和其他Flutter功能
- [Flutter的分层设计 \(video\)](#)
此视频包含有关有状态和无状态widget的信息。由Google工程师Ian Hickson讲解。



[关于本站](#) [加入我们](#) [Gitme](#) [Dart中文网](#) [Flutter中文网开源计划](#) [译者博客](#) | [Github](#)

京ICP备14014371号-3

Flutter中的点击、拖动和其它手势

[编辑本页](#)[提Issue](#)

- [介绍](#)
- [Pointers](#)
- [手势](#)
 - [手势消歧](#)

介绍

本文档介绍了如何在Flutter中监听并响应手势（点击、拖动和缩放）。

Flutter中的手势系统有两个独立的层。第一层有原始指针(pointer)事件，它描述了屏幕上指针（例如，触摸，鼠标和触控笔）的位置和移动。第二层有手势，描述由一个或多个指针移动组成的语义动作。

Pointers

指针(Pointer)代表用户与设备屏幕交互的原始数据。有四种类型的指针事

- `PointerDownEvent` 指针接触到屏幕的特定位置
- `PointerMoveEvent` 指针从屏幕上的一个位置移动到另一个位置
- `PointerUpEvent` 指针停止接触屏幕
- `PointerCancelEvent` 指针的输入事件不再针对此应用（事件取消）

在指针按下时，框架对您的应用程序执行_命中测试_，以确定指针与屏幕相接的位置存在哪些widget。指针按下事件（以及该指针的后续事件）然后被分发到由_命中测试_发现的最内部的widget。从那里开始，这些事件会冒泡在widget树中向上冒泡，这些事件会从最内部的widget被分发到到widget根的路径上的所有小部件（译者语：这和web中事件冒泡机制相似），没有机制取消或停止冒泡过程(译者语：这和web不同，web中的时间冒泡是可以停止的)。

To listen to pointer events directly from the widgets layer, use a `Listener` widget. However, generally, consider using gestures (as discussed below) instead. 要直接从widget层监听指针事件，可以使用 `Listener` widget。但是，通常，请考虑使用手势（如下所述）

手势

手势表示可以从多个单独的指针事件（甚至可能是多个单独的指针）识别的语义动作（例如，轻敲，拖动和缩放）。完整的一个手势可以分派多个事件，对应于手势的生命周期（例如，拖动开始，拖动更新和拖动结束）：

- `Tap`
 - `onTapDown` 指针已经在特定位置与屏幕接触
 - `onTapUp`

指针停止在特定位置与屏幕接触

- `onTap` **tap**事件触发
- `onTapCancel` 先前指针触发的 `onTapDown` 不会在触发tap事件
- 双击
 - `onDoubleTap` 用户快速连续两次在同一位置轻敲屏幕.
- 长按
 - `onLongPress` 指针在相同位置长时间保持与屏幕接触
- 垂直拖动
 - `onVerticalDragStart` 指针已经与屏幕接触并可能开始垂直移动
 - `onVerticalDragUpdate` 指针与屏幕接触并已沿垂直方向移动.
 - `onVerticalDragEnd` 先前与屏幕接触并垂直移动的指针不再与屏幕接触，并且在停止接触屏幕时以特定速度移动
- 水平拖动
 - `onHorizontalDragStart` 指针已经接触到屏幕并可能开始水平移动
 - `onHorizontalDragUpdate` 指针与屏幕接触并已沿水平方向移动
 - `onHorizontalDragEnd` 先前与屏幕接触并水平移动的指针不再与屏幕接触，并在停止接触屏幕时以特定速度移动

要从widget层监听手势，请使用 `GestureDetector` .

如果您使用的是Material Components，那么大多数widget已经对tap或手势做出了响应。例如 `IconButton`和 `FlatButton` 响应presses（taps），`ListView` 响应滑动事件触发滚动。如果您不使用这些widget，但想要在点击时上出现“墨水飞溅”效果，可以使用 `InkWell` 。

手势消歧

在屏幕上的指定位置，可能会有多个手势检测器。所有这些手势检测器在指针事件流经过并尝试识别特定手势时监听指针事件流。`GestureDetector` widget决定是哪种手势。

当屏幕上给定指针有多个手势识别器时，框架通过让每个识别器加入一个“手势竞争场”来确定用户想要的手势。“手势竞争场”使用以下规则确定哪个手势胜出

- 在任何时候，识别者都可以宣布失败并离开“手势竞争场”。如果在“竞争场”中只剩下一个识别器，那么该识别器就是赢家
- 在任何时候，识别者都可以宣布胜利，这会导致胜利，并且所有剩下的识别器都会失败

例如，在消除水平和垂直拖动的歧义时，两个识别器在接收到指针向下事件时进入“手势竞争场”。识别器观察指针移动事件。如果用户将指针水平移动超过一定数量的逻辑像素，则水平识别器将声明胜利，并且手势将被解释为水平拖拽。类似地，如果用户垂直移动超过一定数量的逻辑像素，垂直识别器将宣布胜利。

当只有水平（或垂直）拖动识别器时，“手势竞争场”是有益的。在这种情况下，“手势竞争场”将只有一个识别器，并且水平拖动将被立即识别，这意味着水平移动的第二个像素可以被视为拖动，用户不需要等待进一步的手势消歧。



Flutter中的动画

[编辑本页](#) [提Issue](#)

- [动画类型](#)
 - [补间\(Tween\)动画](#)
 - [基于物理的动画](#)
- [常见的动画模式](#)
 - [动画列表或网格](#)
 - [共享元素转换](#)
 - [交错动画](#)
- [其它资源](#)

精心设计的动画会让用户界面感觉更直观、流畅，能改善用户体验。Flutter的动画支持可以轻松实现各种动画类型。许多widget，特别是[Material Design widgets](#)，都带有在其设计规范中定义的标准动画效果，但也可以自定义这些效果。

以下资源是开始学习Flutter动画框架的好地方。这些文档都详细的展示了如何编写动画代码。

- [教程: Flutter中的动画](#)

介绍了Flutter动画包（控制器、动画、曲线、监听器）中的基础类，它引导您使用不同的动画API逐步制作补间动画。
- [构建漂亮的用户界面](#)

Codelab 演示如何构建简单的聊天应用程序. [Step 7 \(Animate your app\)](#) 展示了如何对新消息进行动画处理 - 将它从输入区域滑动到消息列表。

动画类型

动画分为两类：基于tween或基于物理的。以下部分解释了这些术语的含义，并列出了一些相关的资源。 在一些情况下，我们最好的文档就是Flutter gallery中的示例代码。

补间(Tween)动画

“介于两者之间”的简称。在补间动画中，定义了开始点和结束点、时间线以及定义转换时间和速度的曲线。然后由框架计算如何从开始点过渡到结束点。

上面列出的文档[Flutter动画教程](#) 并不是专门介绍补间动画的，但在其示例中使用了补间动画。

基于物理的动画

在基于物理的动画中，运动被模拟为与真实世界的行为相似。例如，当你掷球时，它在何处落地，取决于抛球速度有多快、球有多重、距离地面有多远。类似地，将连接在弹簧上的球落下（并弹起）与连接到绳子上的球放下的方式也是不同。

- [Flutter Gallery](#)

在 **Material Components** , [Grid](#) 示例中演示了一个动画。从网格中选择其中一个图像并放大, 然后您可以甩手或拖动平移图像。

- 另外请参阅 [AnimationController.animateWith](#) 和 [SpringSimulation](#)的文档

常见的动画模式

大多数UX或交互设计师发现在设计UI时有一些会经常使用的动画模式。本节列出了一些常用的动画模式, 并告诉您可以在哪里了解更多。

动画列表或网格

此模式涉及在网格或列表中添加或删除元素时应用动画。

- [AnimatedList 示例](#)

此演示来自[示例程序目录](#), 演示如何将元素添加到列表或删除选定元素。 在用户使用加号 (+) 和减号 (-) 按钮时修该并同步列表。

共享元素转换

在这种模式中, 用户从页面中选择一个元素 (通常是一个图像), 然后打开所选元素的详情页面, 在打开详情页时使用动画。在Flutter中, 您可以使用[Hero widget](#) 轻松实现路由 (页面) 之间的共享元素过渡动画。

- [Hero 动画](#)

如何创建两种风格的 **Hero** 动画:

- 在改变位置和大小的同时, **hero**从一页飞到另一页
- **hero**的边界从一个圆形变成一个正方形, 同时它从一个页面飞到另一个页面

- [Flutter Gallery](#) 您可以自己构建Gallery应用程序, 也可以从Play商店下载(中国不行)。 其中 [Shrine](#)演示了包括hero动画的一个例子。

- 另外请参阅 [Hero, Navigator](#) 和 [PageRoute](#) 类的API文档。

交错动画

动画被分解为较小的动作, 其中一些动作被延迟。较小的动画可以是连续的, 或者可以部分或完全重叠。

- [交错动画 \(Staggered Animations\)](#)  NEW

其它资源

在以下链接中了解更多关于Flutter动画的信息:

- [动画: 技术概述](#)
查看动画库中的一些主要类，以及Flutter的动画架构。
- [动画和运动（Motion） Widgets](#)
Flutter提供的一些动画widget的目录



使用字体

[编辑本页](#) [提Issue](#)

介绍

你可以在你的Flutter应用程序中使用不同的字体。例如，您可能会使用您的设计人员创建的自定义字体，或者您可能会使用[Google Fonts\(需翻墙\)](#) 中的字体。

本页介绍如何为Flutter应用配置字体，并在渲染文本时使用它们。

使用字体

在Flutter应用程序中使用字体分两步完成。首先在 `pubspec.yaml` 中声明它们，以确保它们包含在应用程序中。然后通过 `TextStyle` 属性使用字体。

在asset中声明

要在应用中包含字体，请先在 `pubspec.yaml` 中声明它。然后将字体文件复制到您在 `pubspec.yaml` 中指定的位置。

```
flutter:
  fonts:
    - family: Raleway
      fonts:
        - asset: assets/fonts/Raleway-Regular.ttf
        - asset: assets/fonts/Raleway-Medium.ttf
          weight: 500
        - asset: assets/fonts/Raleway-SemiBold.ttf
          weight: 600
    - family: AbrilFatface
      fonts:
        - asset: assets/fonts/abrilfatface/AbrilFatface-Regular.ttf
```

使用字体

```
// declare the text style
const textStyle = const TextStyle(
  fontFamily: 'Raleway',
);
```



```
// use the text style
var buttonText = const Text(
  "Use the font for this text",
  style: textStyle,
);
```

依赖包中的字体 {#from-packages}

要使用依赖包中定义的字体，必须提供 `package` 参数。例如，假设上面的字体声明位于 `pubspec.yaml` 中声明的 `my_package` 包中。然后创建 `TextStyle` 的过程如下：

```
const textStyle = const TextStyle(
  fontFamily: 'Raleway',
  package: 'my_package',
);
```

如果包内部使用它定义的字体，则也应该在创建文本样式时指定 `package` 参数，如上例所示。

一个包也可以提供字体文件而不需要在 `pubspec.yaml` 中声明。这些文件应该包的 `lib/` 文件夹中。字体文件不会自动绑定到应用程序中，应用程序可以在声明字体时有选择地使用这些字体。假设一个名为 `my_package` 的包中有一个：

```
lib/fonts/Raleway-Medium.ttf
```

然后，应用程序可以声明一个字体，如下面的示例所示：

```
flutter:
  fonts:
    - family: Raleway
      fonts:
        - asset: assets/fonts/Raleway-Regular.ttf
        - asset: packages/my_package/fonts/Raleway-Medium.ttf
          weight: 500
```

这 `lib/` 是隐含的，所以它不应该包含在 `asset` 路径中。

在这种情况下，由于应用程序本地定义了字体，所以创建的 `TextStyle` 没有 `package` 参数：

```
const textStyle = const TextStyle(
```

```
fontFamily: 'Raleway',  
);
```

使用Material Design字体图标

如果要使用Material Design字体图标，可以通过向 `pubspec.yaml` 文件添加属性 `uses-material-design: true` 来简单包含它。

```
flutter:  
  # The following line ensures that the Material Icons font is  
  # included with your application, so that you can use the icons in  
  # the Icons class.  
  uses-material-design: true
```

pubspec.yaml 选项定义

`family` 是字体的名称, 你可以在 `TextStyle` 的 `fontFamily` 属性中使用.

`asset` 是相对于 `pubspec.yaml` 文件的路径.这些文件包含字体中字形的轮廓。在构建应用程序时，这些文件会包含在应用程序的`asset`包中。

可以给字体设置粗细、倾斜等样式

- `weight` 属性指定字体的粗细，取值范围是100到900之间的整百数(100的倍数). 这些值对应 `FontWeight`，可以用于 `TextStyle` 的 `fontWeight` 属性
- `style` 指定字体是倾斜还是正常，对应的值为 `italic` 和 `normal`. 这些值对应 `FontStyle` 可以用于 `TextStyle` 的 `fontStyle` `TextStyle` 属性

TextStyle

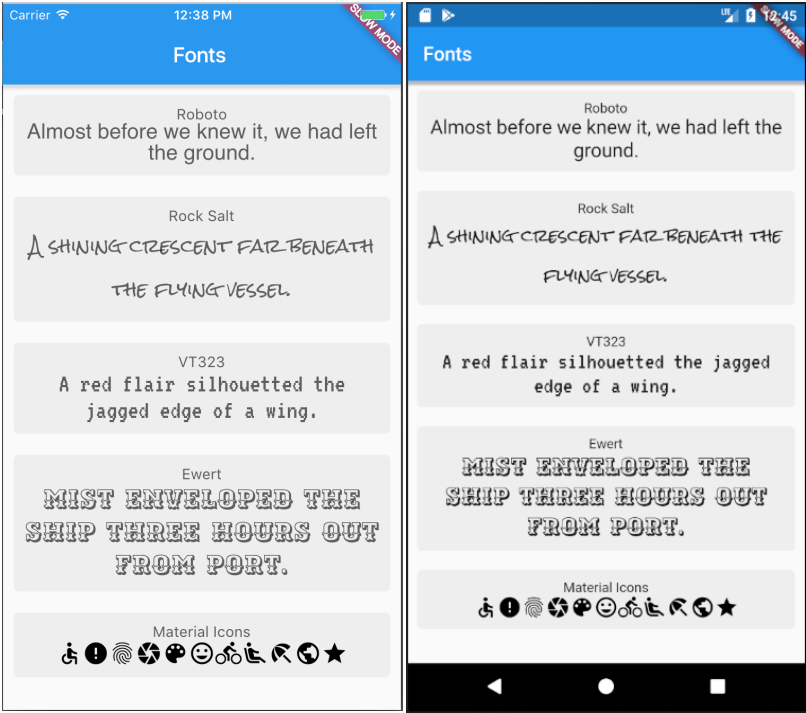
如果一个 `TextStyle` 对象指定了一个没有确切字体文件的`weight`或`style`，那么引擎会为该字体使用一个通用的文件，并尝试为指定的`weight`和`style`推断一个轮廓。

例子

以下是在应用程序中声明和使用字体的示例。

iOS

Android



在pubsec.yaml中声明字体。

```
name: my_application
description: A new Flutter project.

dependencies:
  flutter:
    sdk: flutter

flutter:
  # Include the Material Design fonts.
  uses-material-design: true

  fonts:
    - family: Rock Salt
      fonts:
        # https://fonts.google.com/specimen/Rock+Salt
        - asset: fonts/RockSalt-Regular.ttf
    - family: VT323
      fonts:
        # https://fonts.google.com/specimen/VT323
        - asset: fonts/VT323-Regular.ttf
    - family: Ewert
      fonts:
        # https://fonts.google.com/specimen/Ewert
        - asset: fonts/Ewert-Regular.ttf
```

源码是：

```
import 'package:flutter/material.dart';
```

```

const String words1 = "Almost before we knew it, we had left the ground.";
const String words2 = "A shining crescent far beneath the flying vessel.";
const String words3 = "A red flair silhouetted the jagged edge of a wing.";
const String words4 = "Mist enveloped the ship three hours out from port.";

void main() {
  runApp(new MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Flutter Fonts',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new FontsPage(),
    );
  }
}

class FontsPage extends StatefulWidget {
  @override
  _FontsPageState createState() => new _FontsPageState();
}

class _FontsPageState extends State<FontsPage> {
  @override
  Widget build(BuildContext context) {
    // Rock Salt - https://fonts.google.com/specimen/Rock+Salt
    var rockSaltContainer = new Container(
      child: new Column(
        children: <Widget>[
          new Text(
            "Rock Salt",
          ),
          new Text(
            words2,
            textAlign: TextAlign.center,
            style: new TextStyle(
              fontFamily: "Rock Salt",
              fontSize: 17.0,
            ),
          ),
        ],
      ),
      margin: const EdgeInsets.all(10.0),
      padding: const EdgeInsets.all(10.0),
      decoration: new BoxDecoration(
        color: Colors.grey.shade200,
        borderRadius: new BorderRadius.all(new Radius.circular(5.0)),
      ),
    );
  }
}

```

```
// VT323 - https://fonts.google.com/specimen/VT323
var vt323Container = new Container(
  child: new Column(
    children: <Widget>[
      new Text(
        "VT323",
      ),
      new Text(
        words3,
        textAlign: TextAlign.center,
        style: new TextStyle(
          fontFamily: "VT323",
          fontSize: 25.0,
        ),
      ),
    ],
  ),
  margin: const EdgeInsets.all(10.0),
  padding: const EdgeInsets.all(10.0),
  decoration: new BoxDecoration(
    color: Colors.grey.shade200,
    borderRadius: new BorderRadius.all(new Radius.circular(5.0)),
  ),
);

// https://fonts.google.com/specimen/Ewert
var ewertContainer = new Container(
  child: new Column(
    children: <Widget>[
      new Text(
        "Ewert",
      ),
      new Text(
        words4,
        textAlign: TextAlign.center,
        style: new TextStyle(
          fontFamily: "Ewert",
          fontSize: 25.0,
        ),
      ),
    ],
  ),
  margin: const EdgeInsets.all(10.0),
  padding: const EdgeInsets.all(10.0),
  decoration: new BoxDecoration(
    color: Colors.grey.shade200,
    borderRadius: new BorderRadius.all(new Radius.circular(5.0)),
  ),
);

// Material Icons font - included with Material Design
String icons = "";
```

```

// https://material.io/icons/#ic_accessible
// accessible: &#xE914; or 0xE914 or E914
icons += "\u{E914}";

// https://material.io/icons/#ic_error
// error: &#xE000; or 0xE000 or E000
icons += "\u{E000}";

// https://material.io/icons/#ic_fingerprint
// fingerprint: &#xE90D; or 0xE90D or E90D
icons += "\u{E90D}";

// https://material.io/icons/#ic_camera
// camera: &#xE3AF; or 0xE3AF or E3AF
icons += "\u{E3AF}";

// https://material.io/icons/#ic_palette
// palette: &#xE40A; or 0xE40A or E40A
icons += "\u{E40A}";

// https://material.io/icons/#ic_tag_faces
// tag faces: &#xE420; or 0xE420 or E420
icons += "\u{E420}";

// https://material.io/icons/#ic_directions_bike
// directions bike: &#xE52F; or 0xE52F or E52F
icons += "\u{E52F}";

// https://material.io/icons/#ic_airline_seat_recline_extra
// airline seat recline extra: &#xE636; or 0xE636 or E636
icons += "\u{E636}";

// https://material.io/icons/#ic_beach_access
// beach access: &#xEB3E; or 0xEB3E or EB3E
icons += "\u{EB3E}";

// https://material.io/icons/#ic_public
// public: &#xE80B; or 0xE80B or E80B
icons += "\u{E80B}";

// https://material.io/icons/#ic_star
// star: &#xE838; or 0xE838 or E838
icons += "\u{E838}";

var materialIconsContainer = new Container(
  child: new Column(
    children: <Widget>[
      new Text(
        "Material Icons",
      ),
      new Text(
        icons,
        textAlign: TextAlign.center,
        style: new TextStyle(
          inherit: false,

```

```
        fontFamily: "MaterialIcons",
        color: Colors.black,
        fontStyle: FontStyle.normal,
        fontSize: 25.0,
      ),
    ),
  ],
),
margin: const EdgeInsets.all(10.0),
padding: const EdgeInsets.all(10.0),
decoration: new BoxDecoration(
  color: Colors.grey.shade200,
  borderRadius: new BorderRadius.all(new Radius.circular(5.0)),
),
);

return new Scaffold(
  appBar: new AppBar(
    title: new Text("Fonts"),
  ),
  body: new ListView(
    children: <Widget>[
      rockSaltContainer,
      v2t323Container,
      ewertContainer,
      materialIconsContainer,
    ],
  ),
);
}
```



Flutter中的盒约束

[编辑本页](#) [提Issue](#)

盒约束是指`widget`可以按照指定限制条件来决定自身如何占用布局空间，所谓的“盒”即指自身的渲染框。

- [介绍](#)
- [无边界约束](#)
- [Flex](#)

介绍

在Flutter中，`widget`由其底层的 `RenderBox` 对象渲染。渲染框由它们的父级给出约束，并且在这些约束下调整自身大小。约束由最小宽度、最大宽度和高度组成；尺寸由特定的宽度和高度组成。

通常，按照`widget`如何处理他们的约束来看，有三种类型的盒子：

- 尽可能大。例如 `Center` 和 `ListView` 的渲染盒
- 跟随子`widget`大小。例如，`Transform` 和 `Opacity` 的渲染盒。
- 指定尺寸。例如，`Image` 和 `Text` 的渲染盒

一些`widget`，例如 `Container`，会根据构造函数参数的不同而不同。`Container` 默认是尽可能大的占用空间，但是如果你给它指定一个`width`，那它就会采用指定的值。

其他一些，例如 `Row` 和 `Column` (`flex boxes`) 会根据给定它们的约束的不同而不同，如下面的“**Flex**”部分所述。

这些约束有时是“tight”，这意味着它们没有留下让渲染框自己决定大小的空间（例如，如果最小和最大宽度相同，也就是说有一个tight宽度）。其中的主要例子是 `App` `widget`，它是`RenderView`类包含的一个`widget`：由应用程序`build`函数返回的子`widget`的渲染框被指定了一个约束，强制它精确填充应用程序的内容区域（通常是整个屏幕）。Flutter中的许多盒子，特别是那些只包含一个子`widget`的，都会将其约束传递给他们的孩子。这意味着如果您在应用程序的渲染树的根部嵌套一些盒子，那么子节点都要受到这些渲染盒的约束。

有些箱子放松了约束，有“最大”约束，但没有“最小”约束。例如，`Center`。

无边界约束

在某些情况下，渲染盒的约束将是无边界（`unbounded`）的或无限的。这意味着最大宽度或最大高度会设置为 `double.INFINITY`。

一个本身试图占用尽可能大的渲染盒在给定无边界约束时不会有用，在检查模式下（译者语：指Dart的checked模式），会抛出异常。

渲染盒具有无边界约束的最常见情况是在自身处于弹性盒（`Row` 和 `Column`）内以及可滚动区域

（ `ListView` 和其他 `ScrollView` 的子类）内。

特别是， `ListView` 试图扩充以适应其横向可用空间（即，如果它是一个垂直滚动块，它将尝试与其父项一样宽）。 如果您`ListView`在水平滚动的情况下嵌套垂直滚动的`ListView`，则内部滚动区域会尽可能宽，这是无限宽的，因为外部滚动区域可以在水平方向上一直滚动。

Flex

弹性盒自身（ `Row` 和 `Column` ） 的行为有所不同，这取决于它们在给定的方向上是处于有边界的限制还是无边界的限制下。

在有边界的限制下，他们在这个方向上会尽可能大。

在无边界的限制下，他们试图让自己的子节点在这个方向自适应。 在这种情况下，您不能将子节点的 `flex` 属性设置为0以外的任何值（默认值为0）。 在`widget`库中，这意味着一个弹性盒位于另一个弹性盒或可滚动的盒子内部时，你不能使用 `Expanded` 。 如果你这样做，你会得到一个异常消息。

在 交叉 方向上, 例如在 `Column` 的宽度和在 `Row` 的高度上，它们绝不能是无界的，否则它们将无法合理地对齐他们的子节点。



在Flutter中添加资源和图片

[编辑本页](#)[提Issue](#)

- [介绍](#)
- [指定 assets](#)
 - [Asset 变体 \(variant\)](#)
- [加载 assets](#)
 - [加载文本assets](#)
 - [加载 images](#)
 - [声明分辨率相关的图片 assets](#)
 - [加载图片](#)
 - [依赖包中的资源图片](#)
 - [打包 package assets](#)
- [平台 assets](#)
 - [更新app 图标](#)
 - [Android](#)
 - [iOS](#)
 - [更新启动页](#)
 - [Android](#)
 - [iOS](#)

介绍

Flutter应用程序可以包含代码和 **assets**（有时称为资源）。**asset**是打包到程序安装包中的，可在运行时访问。常见类型的**asset**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）。

指定 assets

Flutter使用 `pubspec.yaml` 文件（位于项目根目录），来识别应用程序所需的**asset**。

这里是一个例子：

```
flutter:
  assets:
    - assets/my_icon.png
    - assets/background.png
```

该 `assets` 部分的 `flutter` 部分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

件所在位置的显式路径进行标识。`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`）。

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。

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，如果您的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.`

...然后您的 `pubspec.yaml` 文件包含：

```
flutter:  
  assets:  
    - graphics/background.png
```

.....那么这两个 `graphics/background.png` 和 `graphics/dark/background.png` 将包含在您的`asset bundle`中。前者被认为是`_main asset_`，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使用`asset`变体；见下文。将来，这种机制可能会扩展到本地化、阅读提示等方面。

加载 assets

您的应用可以通过 `AssetBundle` 对象访问其`asset`。

有两种主要方法允许从`Asset bundle`中加载字符串/text（`loadString`）或图片/二进制（`load`）。

加载文本assets

每个Flutter应用程序都有一个 `rootBundle` 对象，可以轻松访问主资源包。可以直接使用 `package:flutter/services.dart` 中全局静态的 `rootBundle` 对象来加载`asset`。

但是，建议使用 `DefaultAssetBundle` 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()` 从应用运行时间接加载asset（例如JSON文件）。

在Widget上下文之外，或AssetBundle的句柄不可用时，您可以使用 `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 images

Flutter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` 了解如何将逻辑请求asset映射到最接近当前设备像素比例的asset。为了使这种映射起作用，应该根据特定的目录结构来保存asset

- .../image.png
- .../Mx/image.png
- .../Nx/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，它们指定不同素设备像比例的图片

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控件上指定渲染图像的宽度和高度，以便它将占用与主资源相同的屏幕空间量（并不是相同的物理像素），只是分辨率更高。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，会按分辨率从低到的顺序去选择（译者语：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，请在widget的build方法中使用 `AssetImage` 类。

例如，您的应用可以从上面的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'),
        // ...
      ),
      // ...
    ),
  );
  // ...
}
```

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。（如果您使用一些更低级别的类，如 `ImageStream` 或 `ImageCache`，您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')
```

包使用的本身的资源也应该加上 `package` 参数来获取。

打包 package assets

如果在 `pubspec.yaml` 文件中声明了期望的资源，它将会打包到响应的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
```

The `lib/` is implied, so it should not be included in the asset path.

`lib/` 是隐含的，所以它不应该包含在资产路径中。

平台 assets

也有时候可以直接在平台项目中使用`asset`。以下是在Flutter框架加载并运行之前使用资源的两种常见情况。

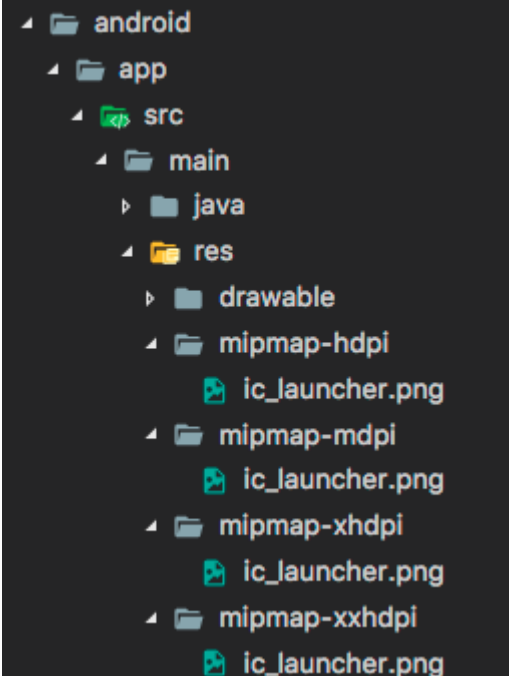
更新app 图标

更新您的Flutter应用程序的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` 。各种位图资源文件夹（如 `mipmap-hdpi` 已包含占位符图像“`ic_launcher.png`”）。只需按照[Android开发人员指南](#)中的说明，将其替换为所需的资源，并遵守每种屏幕密度的建议图标大小标准。

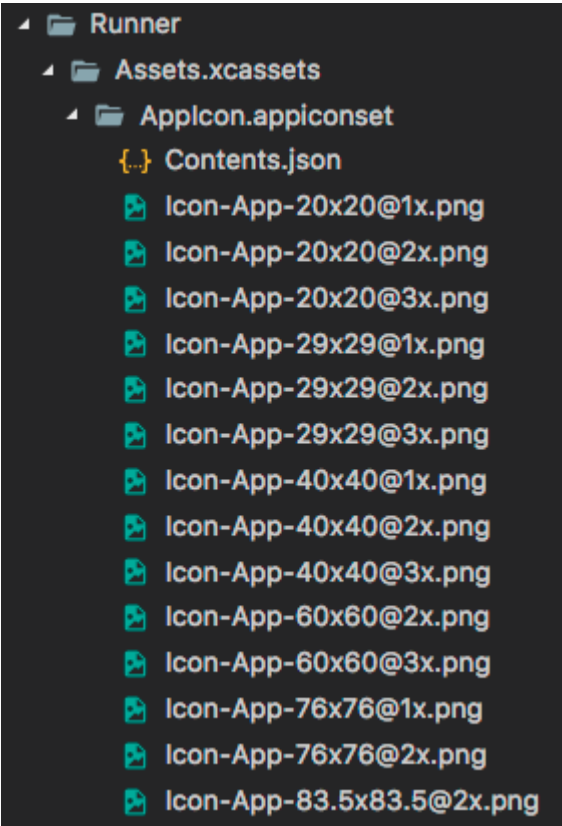


注意: 如果您重命名.png文件, 则还必须在您AndroidManifest.xml的<application>标签的android:icon属性中更新名称。

iOS

在你的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。只需将它们替换为适当大小的图片。保留

原始文件名称。



更新启动页



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`函数（或者更具体地说，如果您不调用`window.render`去响应`window.onDrawFrame`）的话，启动屏幕将永远持续显示。

Android

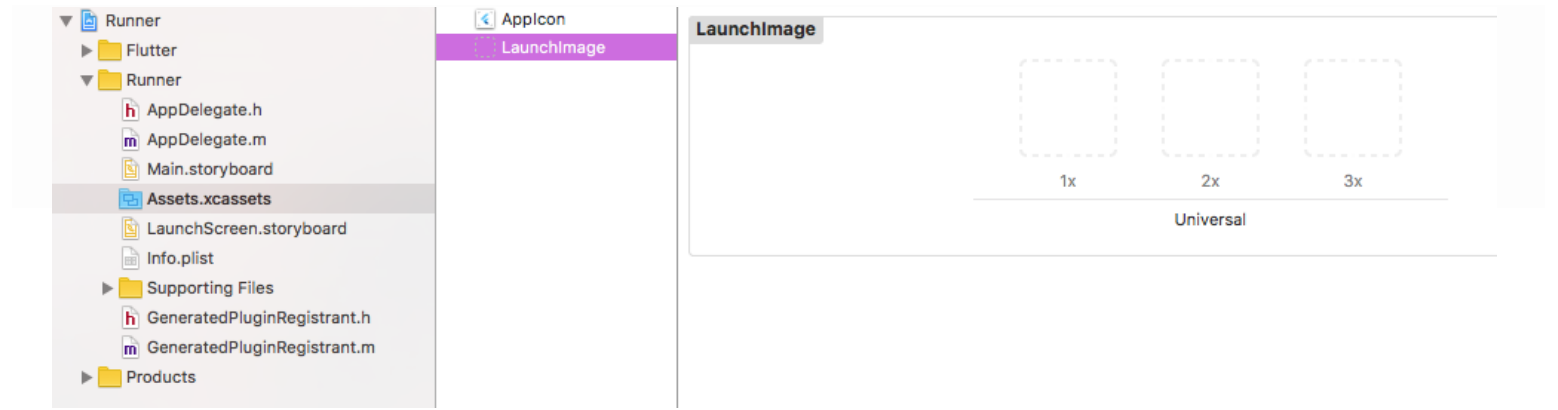
要将启动屏幕（**splash screen**）添加到您的Flutter应用程序，请导航至 `.../android/app/src/main`。在 `res/drawable/launch_background.xml`，通过自定义**drawable**来实现自定义启动界面。

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 `images` `LaunchImage.png`、`LaunchImage@2x.png`、`LaunchImage@3x.png`。如果您使用不同的文件名，那您还必须更新同一目录中的 `Contents.json` 文件。

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。



处理文本输入

[编辑本页](#) [提Issue](#)

- [介绍](#)
- [选择一个widget](#)
- [获取用户输入](#)
 - [onChanged](#)
 - [TextEditingController](#)
- [例子](#)

介绍

本页面介绍如何在Flutter应用程序中输入文本

选择一个widget

`TextField` 是最常用的文本输入widget.

默认情况下, `TextField` 有一个下划线装饰 (`decoration`)。您可以通过提供给 `decoration` 属性设置一个 `InputDecoration` 来添加一个标签、一个图标、提示文字和错误文本。要完全删除装饰 (包括下划线和为标签保留的空间), 将`decoration`明确设置为空即可。

`TextFormField` 包裹一个 `TextField` 并将其集成在 `Form` 中。你要提供一个验证函数来检查用户的输入是否满足一定的约束 (例如, 一个电话号码) 或当你想将 `TextField` 与其他 `FormField` 集成时, 使用 `TextFormField`。

获取用户输入

有两种获取用户输入的主要方法: :

- 处理 `onChanged` 回调
- 提供一个 `TextEditingController` .

onChanged

每当用户输入时, `TextField`会调用它的 `onChanged` 回调。 您可以处理此回调以查看用户输入的内容。例如, 如果您正在输入搜索字段, 则可能需要在用户输入时更新搜索结果。

TextEditingController

一个更强大 (但更精细) 的方法是提供一个 `TextEditingController` 作为 `TextField` 的 `controller` 属性。 在用户输入时, `controller`的 `text` 和 `selection` 属性不断的更新。要在这些属性更改时得到通知, 请使

用controller的 `addListener` 方法监听控制器。（如果你添加了一个监听器，记得在你的State对象的`dispose`方法中删除监听器）。

该 `TextEditingController` 还可以让您控制 `TextField` 的内容。如果修改controller的 `text` 或 `selection` 的属性，`TextField` 将更新，以显示修改后的文本或选中区间。例如，您可以使用此功能来实现推荐内容的自动补全。

例子

以下是使用一个 `TextEditingController` 读取`TextField`中用户输入的值的示例：

```
/// Opens an [AlertDialog] showing what the user typed.
class ExampleWidget extends StatefulWidget {
  ExampleWidget({Key key}) : super(key: key);

  @override
  _ExampleWidgetState createState() => new _ExampleWidgetState();
}

/// State for [ExampleWidget] widgets.
class _ExampleWidgetState extends State<ExampleWidget> {
  final TextEditingController _controller = new TextEditingController();

  @override
  Widget build(BuildContext context) {
    return new Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        new TextField(
          controller: _controller,
          decoration: new InputDecoration(
            hintText: 'Type something',
          ),
        ),
        new RaisedButton(
          onPressed: () {
            showDialog(
              context: context,
              child: new AlertDialog(
                title: new Text('What you typed'),
                content: new Text(_controller.text),
              ),
            );
          },
          child: new Text('DONE'),
        ),
      ],
    );
  }
}
```



路由和导航

[编辑本页](#) [提Issue](#)

大多数应用程序具有多个页面或视图，并且希望将用户从页面平滑过渡到另一个页面。Flutter的路由和导航功能可帮助您管理应用中屏幕之间的命名和过渡。

- [核心概念](#)
- [例子](#)

核心概念

管理多个页面时有两个核心概念和类：[Route](#)和 [Navigator](#)。一个route是一个屏幕或页面的抽象，Navigator是管理route的Widget。Navigator可以通过route入栈和出栈来实现页面之间的跳转。

要开始使用，我们建议您阅读[Navigator的API文档](#)。在那里，您将了解各种路由、命名路由、路由过度等等。

例子

[stocks](#) 示例是一个[MaterialApp](#)，其中有如何声明命名route的一个简单的例子

[Navigator API 文档](#) 包含了一些路由跳转、命名路由、路由返回数据等多个示例



国际化Flutter App

[编辑本页](#) [提Issue](#)

你将会学到什么：

- 如何跟踪设备的区域(locale)设置（用户的首选语言）
- 如何管理特定于区域环境(locale)的应用程序值.
- 如何使用应用程序支持多区域环境

如果您的应用可能会给另一种语言的用户使用，那么您需要“国际化”它。这意味着您在编写应用程序时需要为应用程序支持的每种语言环境，设置“本地化”的一些值，如文本和布局。Flutter提供一些widgets和类，以帮助实现国际化，而Flutter的库本身也是国际化的。

接下来的教程主要是根据Flutter MaterialApp类编写的，因为大多数应用程序都是这样编写的。根据较低级别的WidgetsApp类编写的应用程序也可以使用相同的类和逻辑进行国际化。

内容

- [设置一个国际化的应用程序: flutter_localizations package](#)
- [跟踪区域设置: Locale类和Localizations widget](#)
- [加载和获取本地化的值](#)
- [使用打包好的LocalizationsDelegates](#)
- [为应用的本地化资源定义一个类](#)
- [指定应用的supportedLocales参数](#)
- [指定本地化资源的另一个类](#)
- [附录: 使用 Dart intl 工具](#)

国际化APP的例子

如果您想通过阅读国际化Flutter应用程序的代码开始，下面是两个小例子。第一个旨在尽可能简单，第二个使用intl包提供的API和工具。如果您没用过Dart的intl包，请参阅[使用Dart intl 工具](#)。

- [简单的国际化应用](#)
- [基于 intl 包的国际化应用](#)

设置一个国际化的应用程序: the flutter_localizations package

默认情况下，Flutter仅提供美国英语本地化。要添加对其他语言的支持，应用程序必须指定其他MaterialApp属性，并包含一个名为的单独包-“flutter_localizations”。截至2017年10月，该软件包支持15种语言。

要使用flutter_localizations，请将该包作为依赖项添加到您的 pubspec.yaml 文件中：

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
```

接下来，导入flutter_localizations库，并指定MaterialApp的 localizationsDelegates 和 supportedLocales ：

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

new MaterialApp(
  localizationsDelegates: [
    // ... app-specific localization delegate[s] here
    GlobalMaterialLocalizations.delegate,
    GlobalWidgetsLocalizations.delegate,
  ],
  supportedLocales: [
    const Locale('en', 'US'), // English
    const Locale('he', 'IL'), // Hebrew
    // ... other locales the app supports
  ],
  // ...
)
```

基于WidgetsApp的应用程序类似，只是不需要 GlobalMaterialLocalizations.delegate 。

localizationsDelegates 列表中的元素是生成本地化值集合的工厂。 GlobalMaterialLocalizations.delegate 为Material Components库提供了本地化的字符串和其他值。 GlobalWidgetsLocalizations.delegate 定义widget默认的文本方向，从左到右或从右到左。

有关这些应用程序属性的更多信息，它们所依赖的类型以及如何国际化Flutter应用程序，这些都可以在下面找到。

跟踪区域设置: Locale类和Localizations widget

该 Locale 类是用来识别用户的语言环境。 移动设备支持通过系统设置菜单为所有应用程序设置区域。国际化应用程序通过显示区域设置特定的值进行响应。 例如，如果用户将设备的语言环境从英语切换到法语，则显示“Hello World”的文本widget将用“Bonjour le monde”重建。

Localizations 小部件定义其子项的区域设置以及子项依赖的本地化资源。 如果系统的语言环境发生变化， WidgetsApp将创建一个Localizations widget并重建它。

您始终可以通过以下方式查找应用的当前区域设置：

```
Locale myLocale = Localizations.localeOf(context);
```

加载和获取本地化的值

`Localizations` widget用于加载和查找包含本地化值的集合的对象。应用程序通过 `Localizations.of(context, type)` 来引用这些对象。如果设备的区域设置发生更改，则`Localizations` widget会自动加载新区域设置的值，然后重新构建使用它们的widget。发生这种情况是因为`Localizations`像`InheritedWidget`一样工作。当`build`函数引用了继承的widget时，会创建对继承的widget的隐式依赖关系。当继承的widget发生更改（`Localizations` widget的区域设置发生更改时），将重建其依赖的上下文。

本地化值由`Localizations` widget的 `LocalizationsDelegates` 列表加载。每个委托必须定义一个异步`load()`方法，以生成封装了一系列本地化值的对象。通常这些对象为每个本地化值定义一个方法。

在大型应用程序中，不同的模块或软件包可能会与自己的本地化捆绑在一起。这就是`Localizations` widget管理对象表的原因，每个`LocalizationsDelegate`都有一个(对象表)。要检索由`LocalizationsDelegate` `load` 方法之一产生的对象，可以指定一个`BuildContext`和对象的类型。

例如，`Material Component` widgets的本地化字符串由`MaterialLocalizations`类定义。此类的实例由`MaterialApp`类提供的`LocalizationDelegate`创建。它们可以通过 `Localizations.of` 被获取到：

```
Localizations.of<MaterialLocalizations>(context, MaterialLocalizations);
```

这个特殊的 `Localizations.of()` 表达式经常使用，所以`MaterialLocalizations`类提供了一个方便的简写：

```
static MaterialLocalizations of(BuildContext context) {
  return Localizations.of<MaterialLocalizations>(context, MaterialLocalizations);
}

/// References to the localized values defined by MaterialLocalizations
/// are typically written like this:

tooltip: MaterialLocalizations.of(context).backButtonTooltip,
```

使用打包好的LocalizationsDelegates

为了尽可能小而且简单，`flutter`软件包中仅提供美国英语值的`MaterialLocalizations`和`WidgetsLocalizations`接口的实现。这些实现类分别称为`DefaultMaterialLocalizations`和`DefaultWidgetsLocalizations`。除非与应用程序的`localizationsDelegates`参数指定了相同基本类型的不同`delegate`，否则它们会自动包含。

`flutter_localizations`软件包包含称为`GlobalMaterialLocalizations`和`GlobalWidgetsLocalizations`的本地化接口的多语言实现。国际化的应用程序必须按照[设置国际化应用程序](#)中的说明为这些类指定本地化代理。


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

new MaterialApp(
  localizationsDelegates: [
    // ... app-specific localization delegate[s] here
    GlobalMaterialLocalizations.delegate,
    GlobalWidgetsLocalizations.delegate,
  ],
  supportedLocales: [
    const Locale('en', 'US'), // English
    const Locale('fr', 'CA'), // canadian French
    // ... other locales the app supports
  ],
  // ...
)
```

全局本地化delegates构造相应类的特定于语言环境的实例。例如， `GlobalMaterialLocalizations.delegate` 是一个产生`GlobalMaterialLocalizations`实例的`LocalizationsDelegate`。

截至2017年10月，国际化代理的类支持约[15种语言](#)

为应用的本地化资源定义一个类

将所有这些放在一起用于国际化应用程序通常从封装应用程序本地化值的类开始。下面的例子是这些类的典型例子。

这个示例的[完整源代码](#)

本示例基于[intl](#)包提供的API和工具。[指定本地化资源的另一个类](#)描述了一个不依赖于intl包的示例。

`DemoLocalizations`类包含应用程序的字符串（仅用于示例），该字符串被翻译为应用程序支持的语言环境。使用Dart的[intl 包](#)生成的函数 `initializeMessages()` 来加载翻译的字符串，并使用 `Intl.message()` 查找它们。

```
class DemoLocalizations {
  static Future<DemoLocalizations> load(Locale locale) {
    final String name = locale.countryCode.isEmpty ? locale.languageCode : locale.toString();
    final String localeName = Intl.canonicalizedLocale(name);
    return initializeMessages(localeName).then((Null _) {
      Intl.defaultLocale = localeName;
      return new DemoLocalizations();
    });
  }

  static DemoLocalizations of(BuildContext context) {
    return Localizations.of<DemoLocalizations>(context, DemoLocalizations);
  }
}
```

```
String get title {
  return Intl.message(
    'Hello World',
    name: 'title',
    desc: 'Title for the Demo application',
  );
}
```

基于intl包的类导入生成的message目录，该目录提供 `initializeMessages()` 函数和每个语言环境的后备存储 `Intl.message()`。message目录由一个[intl工具](#)生成，该工具分析包含`Intl.message()`调用的类的源代码。在这个示例中，这是DemoLocalizations类。

指定应用程序的supportedLocales参数

虽然Flutter的Material Components library包含对大约16种语言的支持，但默认情况下仅提供英文。开发人员需要决定支持哪种语言，因为工具库支持与应用程序不同的一组语言环境是没有意义的。

`MaterialApp` `supportedLocales` 参数限制语言环境更改。当用户更改其设备上的区域设置时，新区域如果是列表的成员，则应用程序的 `Localizations` widget 就适用于该区域。如果找不到设备区域设置的精确匹配项（译者语：指语言和地区同时匹配，如中文，中国），则使用第一个匹配区域设置 `languageCode`（译者语：只设置语言，而不指定地区）。如果失败，则 `supportedLocales` 使用列表的第一个元素。

就之前的DemoApp示例而言，该应用只接受美国英语或加拿大法语语言环境，并将美国英语（列表中的第一个语言环境）替换为其他任何内容。

可以提供一个 `localeResolutionCallback`，在应用获取用户设置的语言区域时会被回调。例如，要让您的应用程序无条件接受用户选择的任何区域设置：

```
class DemoApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      localeResolutionCallback(Locale locale, Iterable<Locale> supportedLocales) {
        return locale;
      },
      // ...
    );
  }
}
```

指定本地化资源的另一个类

之前的DemoApp示例是根据Dart `intl` 包定义的。为了简单起见，开发人员可以选择自己的方法来管理本地化值，也可以与不同的i18n框架进行集成。这个示例的[完整源代码](#)

在此版本的**DemoApp**中，包含应用程序本地化的类**DemoLocalizations**将直接在每种语言**Map**中包含其所有的翻译：

```
class DemoLocalizations {
  DemoLocalizations(this.locale);

  final Locale locale;

  static DemoLocalizations of(BuildContext context) {
    return Localizations.of<DemoLocalizations>(context, DemoLocalizations);
  }

  static Map<String, Map<String, String>> _localizedValues = {
    'en': {
      'title': 'Hello World',
    },
    'es': {
      'title': 'Hola Mundo',
    },
  };

  String get title {
    return _localizedValues[locale.languageCode]['title'];
  }
}
```

在**简单的国际化应用**中， **DemoLocalizationsDelegate**略有不同。它的load方法返回一个**SynchronousFuture**， 因为不需要进行异步加载。

```
class DemoLocalizationsDelegate extends LocalizationsDelegate<DemoLocalizations> {
  const DemoLocalizationsDelegate();

  @override
  bool isSupported(Locale locale) => ['en', 'es'].contains(locale.languageCode);

  @override
  Future<DemoLocalizations> load(Locale locale) {
    return new SynchronousFuture<DemoLocalizations>(new DemoLocalizations(locale));
  }

  @override
  bool shouldReload(DemoLocalizationsDelegate old) => false;
}
```

附录: 使用Dart intl工具

在使用Dart **intl** 包构建API之前， 您需要查看 **intl** 包的文档。以下是根据intl软件包本地化应用程序的过程摘要。

示例程序依赖于一个生成的源文件 `l10n/messages_all.dart`，它定义了应用程序使用的所有本地化字符串。

重新构建 `l10n/messages_all.dart` 需要两个步骤.

1. 将应用程序的根目录作为当前目录，从 `lib/main.dart` 生成 `l10n/intl_messages.arb`

```
$ flutter pub run intl_translation:extract_to_arb --output-dir=lib/l10n lib/main.dart
```

该 `intl_messages.arb` 文件是一个JSON格式的map，拥有一个在 `main.dart` 中定义的 `Intl.message()` 函数入口。此文件作为英语和西班牙语翻译的一个模板，`intl_en.arb` 和 `intl_es.arb`。这些翻译是由您，开发人员创建的。

2. 使用应用程序的根目录作为当前目录，为每个 `lib/l10n/intl_*.arb` 文件生成 `intl_messages_<locale>.dart`，并在 `intl_messages_all.dart` 中导入所有message文件：

```
$ flutter pub run intl_translation:generate_from_arb --output-dir=lib/l10n \
  --no-use-deferred-loading lib/main.dart lib/l10n/intl_*.arb
```

`DemoLocalizations`类使用生成的 `initializeMessages()` 函数（定义在 `intl_messages_all.dart`）来加载本地化的message并使用 `Intl.message()` 来查找它们。



使用 packages

[编辑本页](#) [提Issue](#)

Flutter支持使用由其他开发者贡献给Flutter和Dart生态系统的共享软件包。这使您可以快速构建应用程序，而无需从头开始开发所有应用程序。

现有的软件包支持许多使用场景，例如，网络请求（[http](#)），自定义导航/路由处理（[fluro](#)），集成设备API（如[url_launcher](#) & [battery](#)）以及使用第三方平台SDK（如[Firebase](#)(需翻墙)）。

如果您正打算开发新的软件包，请参阅[开发软件包](#)。

如果您希望添加资源、图片或字体，无论是存储在文件还是包中，请参阅[资源和图片](#)。

- [使用包](#)
 - [搜索packages](#)
 - [将包依赖项添加到应用程序](#)
- [开发新的packages](#)
- [管理包依赖和版本](#)
 - [Package versions](#)
 - [更新依赖包](#)
 - [依赖未发布的packages](#)
- [例子](#)
 - [例子: 使用 CSS Colors package](#)
 - [Example: 使用URL Launcher package to 启动浏览器](#)

使用包

搜索packages

Packages会被发布到了 [Pub](#) 包仓库.

[Flutter landing 页面](#) 显示了与Flutter兼容的包（即声明依赖通常与扑兼容）。所有已发布的包都支持搜索。

将包依赖项添加到应用程序

要将包'[css_colors](#)'添加到应用中，请执行以下操作

1. 依赖它
 - 打开 `pubspec.yaml` 文件，然后在 `dependencies` 下添加 `css_colors:`
2. 安装它
 - 在 terminal中: 运行 `flutter packages get`
 - 或者
 - 在 IntelliJ中: 点击 `pubspec.yaml` 文件顶部的'Packages Get'

3. 导入它

- 在您的Dart代码中添加相应的 `import` 语句.

有关完整示例，请参阅下面的[CSS Colors example](#) below.

开发新的packages

如果某个软件包不适用于您的特定需求，则可以开发新的[自定义package](#)。

管理包依赖和版本

Package versions

所有软件包都有一个版本号，在他们的 `pubspec.yaml` 文件中指定。Pub会在其名称旁边显示软件包的当前版本（例如，请参阅[url_launcher](#)软件包）以及所有先前版本的列表。

当 `pubspec.yaml` 使用速记形式添加包时，`plugin1:` 这被解释为 `plugin1: any`，即可以使用任何版本的包。为了确保某个包在更新后还可以正常使用，我们建议使用以下格式之一指定版本范围：

- 范围限制: 指定一个最小和最大的版本号,如:

```
dependencies:
  url_launcher: '>=0.1.2 <0.2.0'
```

- 范围限制使用 [caret 语法](#): 与常规的范围约束类似（译者语：这和node下npm的版本管理类似）

```
dependencies:
  collection: '^0.1.2'
```

有关更多详细信息，请参阅 [Pub 版本管理指南](#).

更新依赖包

当你在添加一个包后首次运行（IntelliJ中的'Packages Get'） `flutter packages get`，Flutter将找到包的版本保存在pubspec.lock。这确保了如果您或您的团队中的其他开发人员运行 `flutter packages get` 后回获取相同版本的包。

如果要升级到软件包的新版本，例如使用该软件包中的新功能，请运行 `flutter packages upgrade`（在IntelliJ中点击 `Upgrade dependencies`）。这将根据您在pubspec.yaml中指定的版本约束下载所允许的最高可用版本。

依赖未发布的packages

即使未在Pub上发布，软件包也可以使用。对于不用于公开发布的专用插件，或者尚未准备好发布的软件包，可以使用其他依赖项选项：

- **路径 依赖:** 一个Flutter应用可以依赖一个插件通过文件系统的 `path:` 依赖。路径可以是相对的, 也可以是绝对的。例如, 要依赖位于应用相邻目录中的插件'plugin1', 请使用以下语法

```
dependencies:
  plugin1:
    path: ../plugin1/
```

- **Git 依赖:** 你也可以依赖存储在Git仓库中的包。如果软件包位于仓库的根目录中, 请使用以下语法:

```
dependencies:
  plugin1:
    git:
      url: git://github.com/flutter/plugin1.git
```

- **Git 依赖于文件夹中的包:** 默认情况下, Pub假定包位于Git存储库的根目录中。如果不是这种情况, 您可以使用`path`参数指定位置, 例如:

```
dependencies:
  package1:
    git:
      url: git://github.com/flutter/packages.git
      path: packages/package1
```

最后, 您可以使用`ref`参数将依赖关系固定到特定的git commit, branch或tag。有关更多详细信息, 请参阅 [Pub Dependencies article](#).

例子

例子: 使用 CSS Colors package

该 `css_colors` 包为CSS颜色定义颜色常量, 允许您在Flutter中需要 `Color` 类型的任何位置使用它们

要使用这个包:

1. 创建一个名为 'cssdemo'的新项目
2. 打开 `pubspec.yaml`, 并将:

```
dependencies:
  flutter:
    sdk: flutter
```

替换为:

```
dependencies:
  flutter:
    sdk: flutter
  css_colors: ^1.0.0
```

3. 在terminal中运行 `flutter packages get` , 或者在IntelliJ中点击'Packages get'

4. 打开 `lib/main.dart` 并替换其全部内容:

```
import 'package:flutter/material.dart';
import 'package:css_colors/css_colors.dart';

void main() {
  runApp(new MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      home: new DemoPage(),
    );
  }
}

class DemoPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      body: new Container(color: CSSColors.orange)
    );
  }
}
```

5. 运行应用程序

Example: 使用URL Launcher package to 启动浏览器

URL Launcher可以使您打开移动平台上的默认浏览器显示给定的URL。 它演示了软件包如何包含特定于平台的代码（我们称这些软件包为插件）。它在Android和iOS上均受支持。

使用这个插件:

1. 创建一个名为'launchdemo'的新项目

2. 打开 `pubspec.yaml`, 并将:

```
dependencies:  
  flutter:  
    sdk: flutter
```

替换为:

```
dependencies:  
  flutter:  
    sdk: flutter  
  url_launcher: ^0.4.1
```

3. 在terminal中运行 `flutter packages get`, 或者在IntelliJ中点击'Packages get'

4. 打开 `lib/main.dart` 并替换其全部内容:

```
import 'package:flutter/material.dart';  
import 'package:url_launcher/url_launcher.dart';  
  
void main() {  
  runApp(new MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return new MaterialApp(  
      home: new DemoPage(),  
    );  
  }  
}  
  
class DemoPage extends StatelessWidget {  
  launchURL() {  
    launch('https://flutter.io');  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    return new Scaffold(  
      body: new Center(  
        child: new RaisedButton(  
          onPressed: launchURL,  
          child: new Text('Show Flutter homepage'),  
        ),  
      ),  
    );  
  }  
};
```

```
}  
}
```

- 5. 运行应用程序。当您点击“Show Flutter homepage”时，您应该看到手机的默认浏览器打开，并出现Flutter主页



开发Packages和插件

[编辑本页](#)[提Issue](#)

- [Package 介绍](#)
 - [Package 类型](#)
- [Developing Dart packages](#)
 - [Step 1: 开发Dart包](#)
 - [Step 2: 实现package](#)
- [开发插件包](#)
 - [Step 1: 创建 package](#)
 - [Step 2: 实现包 package](#)
 - [Step 2a: 定义包API \(.dart\)](#)
 - [Step 2b: 添加Android平台代码 \(.java / .kt\)](#)
 - [Step 2c: 添加iOS平台代码 \(.h+.m/.swift\)](#)
 - [Step 2d: 连接API和平台代码](#)
- [添加文档](#)
 - [API documentation](#)
- [发布 packages](#)
- [处理包的相互依赖](#)
 - [Android](#)
 - [iOS](#)
 - [解决冲突](#)

Package 介绍

使用package可以创建可轻松共享的模块化代码。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

Package 类型

Packages可以包含多种内容：

- **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，仅将其用于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/或针对iOS（使用ObjC或Swift）平台的特定实现。一个具体的例子是 `battery` 插件包。

Developing Dart packages

Step 1: 开发Dart包

要创建Dart包，请使用 `--template=package` 来执行 `flutter create`

```
$ 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.

Step 2: 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。

要测试软件包，请在 `test` 目录中添加unit tests。

有关如何组织包内容的更多详细信息，请参阅[Dart library package](#)文档。

开发插件包

如果你想开发一个调用特定平台API的包，你需要开发一个插件包，插件包是Dart包的专用版本。插件包包含针对Android（Java或Kotlin代码）或iOS（Objective-C或Swift代码）编写的特定于平台的实现（可以同时包含Android和ios原生的代码）。API使用[platform channels](#)连接到特定平台（Android或IOS）。

Step 1: 创建 package

要创建插件包，请使用 `--template=plugin` 参数执行 `flutter create`

使用 `--org` 选项指定您的组织，并使用反向域名表示法。该值用于生成的Android和iOS代码中的各种包和包标识符。

```
$ flutter create --org com.example --template=plugin hello
```

这将在 `hello/` 文件夹下创建一个具有以下专用内容的插件工程：

- `lib/hello.dart` :
 - 插件包的Dart API.
- `android/src/main/java/com/yourcompany/hello/HelloPlugin.java` :
 - 插件包API的Android实现.
- `ios/Classes/HelloPlugin.m` :
 - 插件包API的ios实现.
- `example/` :

- 一个依赖于该插件的Flutter应用程序，来说明如何使用它

默认情况下，插件项目针对iOS代码使用Objective-C，Android代码使用Java。如果您更喜欢Swift或Kotlin，则可以使用 `-i` 或 `-a` 为iOS或Android指定语言。例如：

```
$ flutter create --template=plugin -i swift -a kotlin hello
```

Step 2: 实现包 package

由于插件包中包含用多种编程语言编写的多个平台的代码，因此需要一些特定的步骤来确保顺畅的体验。

Step 2a: 定义包API (.dart)

插件包的API在Dart代码中定义。打开主文件夹 `hello/`。找到 `lib/hello.dart`

Step 2b: 添加Android平台代码 (.java / .kt)

我们建议您使用Android Studio编辑Android代码。

在Android Studio中编辑Android平台代码之前，首先确保代码至少已经构建过一次（例如，从IntelliJ运行示例应用程序或在终端执行 `cd hello/example ; flutter build apk`）

接下来

1. 启动Android Studio
2. 在'Welcome to Android Studio'对话框选择 'Import project'，或者在菜单栏 'File > New > Import Project...'，然后选择 `hello/example/android/build.gradle` 文件。
3. 在'Gradle Sync' 对话框, 选择 'OK'.
4. 在'Android Gradle Plugin Update' 对话框, 选择 'Don't remind me again for this project'.

您插件的Android平台代码位于 `hello/java/com.yourcompany.hello/HelloPlugin` .

您可以通过按下 按钮从Android Studio运行示例应用程序.

Step 2c: 添加iOS平台代码 (.h+.m/.swift)

我们建议您使用Xcode编辑iOS代码。

在编辑Xcode中的iOS平台代码之前，首先确保代码至少已经构建过一次（例如，从Xcode中运行示例应用程序或终端执行 `cd hello/example ; flutter build ios --no-codesign`）。

接下来

1. 启动 Xcode
2. 选择 'File > Open'，然后选择 `hello/example/ios/Runner.xcworkspace` 文件。

您插件的iOS平台代码位于 `Pods/DevelopmentPods/hello/Classes/` 中。

您可以通过按下按钮来运行示例应用程序

Step 2d: 连接API和平台代码

最后，您需要将用Dart代码编写的API与平台特定的实现连接起来。这是通过[platform channels](#)完成的。

添加文档

建议将以下文档添加到所有软件包：

1. `README.md` :介绍包的文件
2. `CHANGELOG.md` 记录每个版本中的更改
3. `LICENSE` 包含软件包许可条款的文件
4. 所有公共API的API文档 (详情见下文)

API documentation

在发布软件包时，API文档会自动生成并发布到[dartdocs.org](#)，示例请参阅[device_info docs](#)

如果您希望在本地生成API文档，请使用以下命令：

1. 将目录更改为您的软件包的位置：

```
cd ~/dev/mypackage
```

2. 告诉文档工具Flutter SDK的位置：

```
export FLUTTER_ROOT=~/.dev/flutter (on macOS or Linux)
```

```
set FLUTTER_ROOT=~/.dev/flutter (on Windows)
```

3. 运行 `dartdoc` 工具 (它是Flutter SDK的一部分)：

```
$FLUTTER_ROOT/bin/cache/dart-sdk/bin/dartdoc (on macOS or Linux)
```

```
%FLUTTER_ROOT%\bin\cache\dart-sdk\bin\dartdoc (on Windows)
```

有关如何编写API文档的提示，请参阅[Effective Dart: Documentation](#)

发布 packages

一旦你实现了一个包，你可以在[Pub](#)上发布它，这样其他开发人员就可以轻松使用它

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
```

最后，运行发布命令：

```
$ flutter packages pub publish
```

有关发布的详细信息，请参阅[Pub publishing docs](#)

处理包的相互依赖

如果您正在开发一个 `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 **Dart API**，这在 `hello` 包中是可用的：

在 `hello/pubspec.yaml`：

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

现在你可以在 `hello` 中 `import 'package:url_launcher/url_launcher.dart'` 然后 `launch(someUrl)` 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**，那么您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

解决冲突

假设你想在你的 `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    # Good, any 0.4.x with x >= 2 will do.
  image_picker: '0.1.1'  # Not so good, only 0.1.1 will do.
```

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。类似的注释适用于插件包对 `Gradle` 模块和 `Cocoa pods` 的平台特定的依赖关系。

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。



使用平台通道编写平台特定的代码

[编辑本页](#) [提Issue](#)

译者语：所谓“平台特定”或“特定平台”，平台指的就是原生Android或iOS，本文主要讲原生和Flutter之间如何通信、如何进行功能互调。

本指南介绍如何编写自定义平台特定的代码。一些平台特定的功能可通过现有软件包获得; 请参阅[使用 packages](#)。

- [框架概述: 平台通道](#)
 - [平台通道数据类型支持和解码器](#)
- [示例: 使用平台通道调用iOS和Android代码](#)
 - [Step 1: 创建一个新的应用程序项目](#)
 - [Step 2: 创建Flutter平台客户端](#)
 - [Step 3a: 使用Java添加Android平台特定的实现](#)
 - [Step 3b: 使用Kotlin添加Android平台特定的实现](#)
 - [Step 4a: 使用Objective-C添加iOS平台特定的实现](#)
 - [Step 4b: 使用Swift添加一个iOS平台的实现](#)
- [从UI代码中分离平台特定的代码](#)
- [将平台特定的代码作为一个包发布](#)
- [自定义平台通道和编解码器](#)

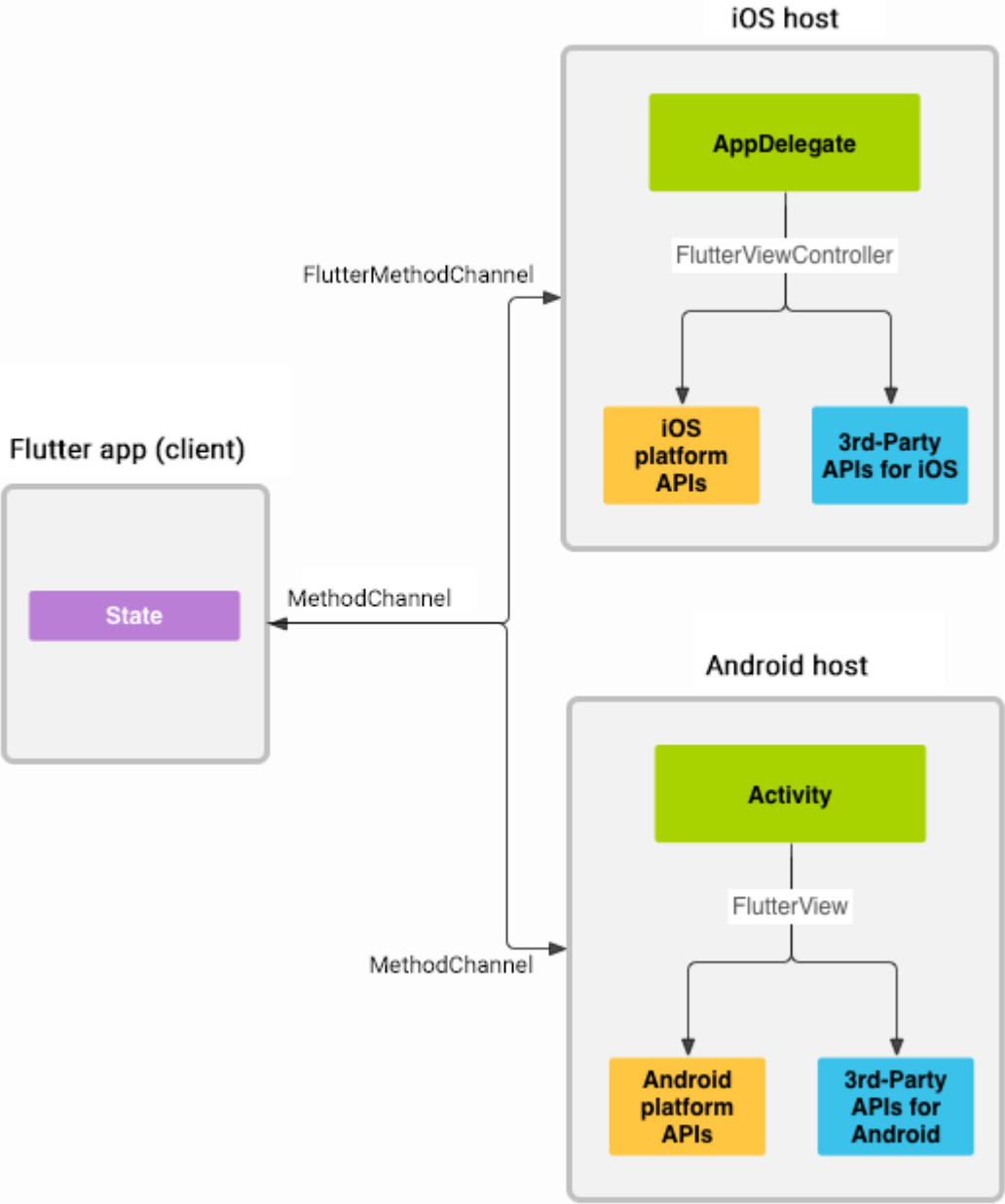
Flutter使用了一个灵活的系统，允许您调用特定平台的API，无论在Android上的Java或Kotlin代码中，还是iOS上的ObjectiveC或Swift代码中均可用。

Flutter平台特定的API支持不依赖于代码生成，而是依赖于灵活的消息传递的方式：

- 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）。
- 宿主监听的平台通道，并接收该消息。然后它会调用特定于该平台的API（使用原生编程语言） - 并将响应发送回客户端，即应用程序的Flutter部分。

框架概述: 平台通道

使用平台通道在客户端（Flutter UI）和宿主（平台）之间传递消息，如下图所示：



消息和响应是异步传递的，以确保用户界面保持响应(不会挂起)。

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。这些类允许您用很少的“脚手架”代码开发平台插件。

注意: 如果需要，方法调用也可以反向发送，宿主作为客户端调用Dart中实现的API。 这个[quick_actions](#)插件就是一个具体的例子

平台通道数据类型支持和解码器

标准平台通道使用标准消息编解码器，以支持简单的类似JSON值的高效二进制序列化，例如 `booleans`, `numbers`, `Strings`, `byte buffers`, `List`, `Maps`（请参阅 `StandardMessageCodec` 了解详细信息）。当您发送和接收值时，这些值在消息中的序列化和反序列化会自动进行。

下表显示了如何在宿主上接收Dart值，反之亦然：

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

示例: 使用平台通道调用iOS和Android代码

以下演示如何调用平台特定的API来获取和显示当前的电池电量。它通过一个平台消息 `getBatteryLevel` 调用Android `BatteryManager` API和iOS `device.batteryLevel` API。

该示例在应用程序内添加了特定于平台的代码。如果您想开发一个通用的平台包，可以在其它应用中也使用的话，你需要开发一个插件，则项目创建步骤稍有不同（请参阅[开发 packages](#)），但平台通道代码仍以相同方式编写。

注意: 此示例的完整的可运行源代码位于: [/examples/platform_channel/](#)，这个示例Android是用Java, IOS用的是Objective-C，IOS Swift版本请参阅 [/examples/platform_channel_swift/](#)

Step 1: 创建一个新的应用程序项目

首先创建一个新的应用程序:

- 在终端运行中: `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/-a标志:

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

Step 2: 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 - 例如，如果平台不支持平台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),  
],  
);  
}
```

Step 3a: 使用Java添加Android平台特定的实现

注意: 以下步骤使用Java。如果您更喜欢Kotlin, 请跳到步骤3b.

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`。确保使用与在Flutter客户端使用的通道名称相同。

```
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。此代码与您在原生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_CAPACITY);  
    } 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，所以我们在call参数中进行检测是否为 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，并使用response参数返回成功和错误情况的响应。如果调用未知的方法，我们也会通知返回：

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

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

您现就可以在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

Step 3b: 使用Kotlin添加Android平台特定的实现

注意: 以下步骤与步骤3a类似，只是使用Kotlin而不是Java。

此步骤假定您在step 1.中 使用该 `-a kotlin` 选项创建了项目

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. 定位到您 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` . (注意: 如果您使用Android Studio 2.3进行编辑, 请注意'kotlin'文件夹将显示为'java'.)

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。此代码与您在原生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.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_CAPACITY)  
    } 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，所以我们在call参数中进行检测是否为 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，并使用response参数返回成功和错误情况的响应。如果调用未知的方法，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

Step 4a: 使用Objective-C添加iOS平台特定的实现

注意: 以下步骤使用Objective-C。如果您喜欢Swift，请跳到步骤4b

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ObjectiveC代码，使用iOS电池API来获取电池电量。此代码与您在本机iOS应用程序中编写的代码完全相同。

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参数中进行检测是否为 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response参数返回成功和错误情况的响应。如果调用未知的方法，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 errorCode:@"UNAVAILABLE"
                    message:@"Battery info unavailable"
                    details:nil]);
        } else {
            result(@(batteryLevel));
        }
    } else {
        result(FlutterMethodNotImplemented);
    }
}];
```

您现在可以在iOS上运行应用程序。如果您使用的是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”。

Step 4b: 使用Swift添加一个iOS平台的实现

注意: 以下步骤与步骤4a类似，只不过是使用Swift而不是Objective-C.

此步骤假定您在步骤1中 使用 -i swift 选项创建了项目。

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! FlutterView
    Controller;
    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。此代码与您在本机iOS应用程序中编写的代码完全相同。

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: "Battery info unavailable",
                             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` 参数中进行检测是否为 `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 `response` 参数返回成功和错误情况的响应。如果调用未知的方法，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);  
  }  
});
```

您现在可以在 iOS 上运行应用程序。如果您使用的是 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”。

从 UI 代码中分离平台特定的代码

如果您希望在多个 Flutter 应用程序中使用特定于平台的代码，将代码分离为位于主应用程序之外的目录中，做一个平台插件会很有用。详情请参阅[开发 packages](#)。

将平台特定的代码作为一个包发布

如果您希望与 Flutter 生态系统中的其他开发人员分享您的特定平台的代码，请参阅[发布 packages](#) ([/developing-packages/#publish](#))以了解详细信息。

自定义平台通道和编解码器

除了上面提到的 `MethodChannel`，您还可以使用 `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，您可以使用专门的 `BinaryCodec`，`StringCodec` 和 `JSONMessageCodec` 类，或创建自己的编解码器。



读写文件

[编辑本页](#) [提Issue](#)

本指南介绍如何使用 `PathProvider` 插件和Dart的 `IO` 库在Flutter中读写文件。

- [介绍](#)
- [读写文件的示例](#)

介绍

`PathProvider` 插件提供了一种平台透明的方式来访问设备文件系统上的常用位置。该类当前支持访问两个文件系统位置：

- **临时目录:** 系统可随时清除的临时目录（缓存）。在iOS上，这对应于 `NSTemporaryDirectory()` 返回的值。在Android上，这是 `getCacheDir()` 返回的值。
- **文档目录:** 应用程序的目录，用于存储只有自己可以访问的文件。只有当应用程序被卸载时，系统才会清除该目录。在iOS上，这对应于 `NSDocumentDirectory`。在Android上，这是 `AppData` 目录。

一旦你的Flutter应用程序有一个文件位置的引用，你可以使用 `dart:io` API来执行对文件系统的读/写操作。有关使用Dart处理文件和目录的更多信息，请参阅此[概述](#) 和这些[示例](#)。

读写文件的示例

以下示例展示了如何统计应用程序中按钮被点击的次数(关闭重启数据不丢失)：

1. 通过 `flutter create` 或在IntelliJ中 **File > New Project** 创建一个新Flutter App.
2. 在 `pubspec.yaml` 文件中[声明依赖](#) `PathProvider` 插件
3. 用一下代码替换 `lib/main.dart` 中的:

```
import 'dart:io';
import 'dart:async';
import 'package:flutter/material.dart';
import 'package:path_provider/path_provider.dart';

void main() {
  runApp(
    new MaterialApp(
      title: 'Flutter Demo',
      theme: new ThemeData(primarySwatch: Colors.blue),
      home: new FlutterDemo(),
    ),
  ),
}
```

```

    );
}

class FlutterDemo extends StatefulWidget {
  FlutterDemo({Key key}) : super(key: key);

  @override
  _FlutterDemoState createState() => new _FlutterDemoState();
}

class _FlutterDemoState extends State<FlutterDemo> {
  int _counter;

  @override
  void initState() {
    super.initState();
    _readCounter().then((int value) {
      setState(() {
        _counter = value;
      });
    });
  }

  Future<File> _getLocalFile() async {
    // get the path to the document directory.
    String dir = (await getApplicationDocumentsDirectory()).path;
    return new File('$dir/counter.txt');
  }

  Future<int> _readCounter() async {
    try {
      File file = await _getLocalFile();
      // read the variable as a string from the file.
      String contents = await file.readAsString();
      return int.parse(contents);
    } on FileSystemException {
      return 0;
    }
  }

  Future<Null> _incrementCounter() async {
    setState(() {
      _counter++;
    });
    // write the variable as a string to the file
    await (await _getLocalFile()).writeAsString('$_counter');
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(title: new Text('Flutter Demo')),
      body: new Center(
        child: new Text('Button tapped $_counter time${
          counter == 1 ? '' : 's'
        }')
      )
    );
  }
}

```

```
        }.'),  
      ),  
      floatingActionButton: new FloatingActionButton(  
        onPressed: _incrementCounter,  
        tooltip: 'Increment',  
        child: new Icon(Icons.add),  
      ),  
    );  
  }  
}
```



在Flutter中发起HTTP网络请求

[编辑本页](#) [提Issue](#)

本页介绍如何在Flutter中创建HTTP网络请求。对于socket，请参阅[dart:io](#)。

注：本篇文档官方使用的是用dart io中的 `HttpClient` 发起的请求，但 `HttpClient` 本身功能较弱，很多常用功能都不支持。我们建议您使用[dio](#) 来发起网络请求，它是一个强大易用的dart http请求库，支持Restful API、FormData、拦截器、请求取消、Cookie管理、文件上传/下载.....详情请查看[github dio](#) 。

- [发起HTTP请求](#)
- [处理异步](#)
- [解码和编码JSON](#)
- [示例: 解码 HTTPS GET请求的JSON](#)
- [API 文档](#)

发起HTTP请求

http支持位于 `dart:io`，所以要创建一个HTTP client，我们需要添加一个导入：

```
import 'dart:io';

var httpClient = new HttpClient();
```

该 client 支持常用的HTTP操作, such as `GET` , `POST` , `PUT` , `DELETE` .

处理异步

注意，HTTP API 在返回值中使用了[Dart Futures](#)。我们建议使用 `async` / `await` 语法来调用API。

网络调用通常遵循如下步骤：

1. 创建 client.
2. 构造 Uri.
3. 发起请求, 等待请求，同时您也可以配置请求headers、 body。
4. 关闭请求, 等待响应.
5. 解码响应的内容.

Several of these steps use Future based APIs. Sample APIs calls for each step above are: 其中的几个步骤使用基于Future的API。上面步骤的示例：

```
get() async {  
  var httpClient = new HttpClient();  
  var uri = new Uri.http(  
    'example.com', '/path1/path2', {'param1': '42', 'param2': 'foo'});  
  var request = await httpClient.getUrl(uri);  
  var response = await request.close();  
  var responseBody = await response.transform(UTF8.decoder).join();  
}
```

有关完整的代码示例，请参阅下面的“示例”。

解码和编码JSON

使用 `dart:convert` 库可以简单解码和编码JSON。 有关其他的JSON文档，请参阅[JSON和序列化](#)。

解码简单的JSON字符串并将响应解析为Map：

```
Map data = JSON.decode(responseBody);  
// Assume the response body is something like: ['foo', { 'bar': 499 }]  
int barValue = data[1]['bar']; // barValue is set to 499
```

要对简单的JSON进行编码，请将简单值（字符串，布尔值或数字字面量）或包含简单值的Map，list等传给`encode`方法：

```
String encodedString = JSON.encode([1, 2, { 'a': null }]);
```

示例: 解码 HTTPS GET请求的JSON

以下示例显示了如何在Flutter应用中对HTTPS GET请求返回的JSON数据进行解码

It calls the [httpbin.com](#) web service testing API, which then responds with your local IP address. Note that secure networking (HTTPS) is used. 它调用[httpbin.com](#)Web service测试API，请注意，使用安全网络请求 (HTTPS)

1. 运行 `flutter create`，创建一个新的Flutter应用。
2. 将 `lib/main.dart` 替换为一下内容：

```
import 'dart:convert';
```

```

import 'dart:io';

import 'package:flutter/material.dart';

void main() {
  runApp(new MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      home: new MyHomePage(),
    );
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({Key key}) : super(key: key);

  @override
  _MyHomePageState createState() => new _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  var _ipAddress = 'Unknown';

  _getIPAddress() async {
    var url = 'https://httpbin.org/ip';
    var httpClient = new HttpClient();

    String result;
    try {
      var request = await httpClient.getUrl(Uri.parse(url));
      var response = await request.close();
      if (response.statusCode == HttpStatus.OK) {
        var json = await response.transform(UTF8.decoder).join();
        var data = JSON.decode(json);
        result = data['origin'];
      } else {
        result =
          'Error getting IP address:\nHttp status ${response.statusCode}';
      }
    } catch (exception) {
      result = 'Failed getting IP address';
    }

    // If the widget was removed from the tree while the message was in flight,
    // we want to discard the reply rather than calling setState to update our
    // non-existent appearance.
    if (!mounted) return;

    setState(() {
      _ipAddress = result;
    });
  }
}

```

```
    ));  
  }  
  
  @override  
  Widget build(BuildContext context) {  
    var spacer = new SizedBox(height: 32.0);  
  
    return new Scaffold(  
      body: new Center(  
        child: new Column(  
          mainAxisAlignment: MainAxisAlignment.center,  
          children: <Widget>[  
            new Text('Your current IP address is:'),  
            new Text('$_ipAddress.'),  
            spacer,  
            new RaisedButton(  
              onPressed: _getIPAddress,  
              child: new Text('Get IP address'),  
            ),  
          ],  
        ),  
      ),  
    );  
  }  
}
```

API 文档

有关完整的API文档，请参阅：

- [库 dart:io](#)
- [库 dart:convert](#)



JSON和序列化

[编辑本页](#) [提Issue](#)

很难想象移动应用程序不需要与Web服务器通信或在某个时刻存储结构化数据。在编写需要联网的应用程序时，我们很可能迟早需要使用JSON。

在本教程中，我们将探讨如何在Flutter中使用JSON。我们回顾一下在不同情况下使用的JSON解决方案以及原因。

- [哪种JSON序列化方法适合我？](#)
 - [小项目手动序列化](#)
 - [在大中型项目中使用代码生成](#)
- [Flutter中是否有GSON / Jackson / Moshi？](#)
- [使用 dart:convert手动序列化JSON](#)
 - [内连序列化JSON](#)
 - [在模型类中序列化JSON](#)
- [使用代码生成库序列化JSON](#)
 - [在项目中设置json_serializable](#)
 - [以json_serializable的方式创建model类](#)
 - [运行代码生成程序](#)
 - [一次性生成](#)
 - [持续生成](#)
 - [使用json_serializable模型](#)
- [一些参考](#)

哪种JSON序列化方法适合我？

本文介绍了使用JSON的两个常规策略：

- [手动序列化和反序列化](#)
- [通过代码生成自动序列化和反序列化](#)

不同的项目具有不同的复杂度和场景。对于较小项目，使用代码生成器可能会过度。对于具有多个JSON model的复杂应用程序，手动序列化可能会比较重复，并会很容易出错。

小项目手动序列化

手动JSON序列化是指使使用 `dart:convert` 中内置的JSON解码器。它将原始JSON字符串传递给 `JSON.decode()` 方法，然后在返回的 `Map<String, dynamic>` 中查找所需的值。它没有外部依赖或其它的设置，对于小项目很方便。

当您的项目变大时，手动编写序列化逻辑可能变得难以管理且容易出错。如果您在访问未提供的JSON字段时

输入了一个错误的字段，则您的代码将会在运行时引发错误。

如果您的项目中JSON model并不多，并且希望快速测试一下，那么手动序列化可能会很方便。有关手动序列化的示例，请参考[这里](#)。

在大中型项目中使用代码生成

代码生成功能的JSON序列化是指通过外部库为您自动生成序列化模板。它需要一些初始设置，并运行一个文件观察器，从您的model类生成代码。例如，[json_serializable](#)和[built_value](#)就是这样的库。

这种方法适用于较大的项目。不需要手写，如果访问JSON字段时拼写错误，这会在编译时捕获的。代码生成的不利之处在于它涉及到一些初始设置。另外，生成的源文件可能会在项目导航器会显得混乱。

当您有一个中型或大型项目时，您可能想要使用代码生成JSON序列化。要查看基于代码生成JSON序列化的示例，请参见[这里](#)。

Flutter中是否有GSON / Jackson / Moshi?

简单的回答是没有。

这样的库需要使用运行时反射，这在Flutter中是禁用的。运行时反射会干扰Dart的`_tree shaking`。使用`_tree shaking`，我们可以在发版时“去除”未使用的代码。这可以显着优化应用程序的大小。

由于反射会默认使用所有代码，因此`_tree shaking`会很难工作。这些工具无法知道哪些widget在运行时未被使用，因此冗余代码很难剥离。使用反射时，应用尺寸无法轻松的进行优化。

dartson呢?

[dartson](#) 使用了运行时反射 `runtime`，所以不能在Flutter中使用它。

虽然我们不能在Flutter中使用运行时反射，但有些库为我们提供了类似易于使用的API，但它们是基于代码生成的。这种方法在[代码生成库](#)部分有更详细的介绍。

使用 dart:convert手动序列化JSON

Flutter中基本的JSON序列化非常简单。Flutter有一个内置 `dart:convert` 库，其中包含一个简单的JSON编码器和解码器。

以下是一个简单的user model的示例JSON。

```
{
  "name": "John Smith",
  "email": "john@example.com"
}
```

`dart:convert`

有了 `JSON` ，我们可以用两种方式来序列化这个 `JSON model`。我们来看看这两种方法：

内连序列化JSON

通过查看 [dart: 转换JSON文档](#)，我们发现可以通过调用 `JSON.decode` 方法来解码JSON，使用JSON字符串作为参数。

```
Map<String, dynamic> user = JSON.decode(json);

print('Howdy, ${user['name']}!');
print('We sent the verification link to ${user['email']}');
```

不幸的是，`JSON.decode()`仅返回一个 `Map<String, dynamic>`，这意味着我们直到运行时才知道值的类型。通过这种方法，我们失去了大部分静态类型语言特性：类型安全、自动补全和最重要的编译时异常。这样一来，我们的代码可能会变得非常容易出错。

例如，当我们访问 `name` 或 `email` 字段时，我们输入的很快，导致字段名打错了。但由于这个JSON在map结构中，所以编译器不知道这个错误的字段名(译者语：所以编译时不会报错)。

在模型类中序列化JSON

我们可以通过引入一个简单的模型类(model class)来解决前面提到的问题，我们称之为 `User`。在 `User` 类内部，我们有：

- 一个 `User.fromJson` 构造函数, 用于从一个map构造出一个 `User` 实例 map structure
- 一个 `toJson` 方法, 将 `User` 实例转化为一个map.

这样，调用代码现在可以具有类型安全、自动补全字段（`name`和`email`）以及编译时异常。如果我们将拼写错误或字段视为 `int` 类型而不是 `String`，那么我们的应用程序就不会通过编译，而不是在运行时崩溃。

user.dart

```
class User {
  final String name;
  final String email;

  User(this.name, this.email);

  User.fromJson(Map<String, dynamic> json)
    : name = json['name'],
      email = json['email'];

  Map<String, dynamic> toJson() =>
  {
    'name': name,
    'email': email,
  }
}
```

```
};  
}
```

现在，序列化逻辑移到了模型本身内部。采用这种新方法，我们可以非常容易地反序列化`user`。

```
Map userMap = JSON.decode(json);  
var user = new User.fromJson(userMap);  
  
print('Howdy, ${user.name}!');  
print('We sent the verification link to ${user.email}.');
```

要序列化一个`user`，我们只是将该 `User` 对象传递给该 `JSON.encode` 方法。我们不需要手动调用 `toJson` 这个方法，因为 `JSON.encode` 已经为我们做了。

```
String json = JSON.encode(user);
```

这样，调用代码就不用担心JSON序列化了。但是，`model`类还是必须的。在生产应用程序中，我们希望确保序列化正常工作。在实践中，`User.fromJson` 和 `User.toJson` 方法都需要单元测试到位，以验证正确的行为。

另外，实际场景中，JSON对象很少会这么简单，嵌套的JSON对象并不罕见。

如果有什么能为我们自动处理JSON序列化，那将会非常好。幸运的是，有！

使用代码生成库序列化JSON

尽管还有其他库可用，但在本教程中，我们使用了[json_serializable package](#)包。它是一个自动化的源代码生成器，可以为我们生成JSON序列化模板。

由于序列化代码不再由我们手写和维护，我们将运行时产生JSON序列化异常的风险降至最低。

在项目中设置json_serializable

要包含 `json_serializable` 到我们的项目中，我们需要一个常规和两个开发依赖项。简而言之，开发依赖项是不包含在我们的应用程序源代码中的依赖项。

通过[此链接](#)可以查看这些所需依赖项的最新版本。

pubspec.yaml

```
dependencies:
```



```
# Your other regular dependencies here
json_annotation: ^2.0.0

dev_dependencies:
  # Your other dev_dependencies here
  build_runner: ^1.0.0
  json_serializable: ^2.0.0
```

在您的项目根文件夹中运行 `flutter packages get` (或者在编辑器中点击 “Packages Get”) 以在项目中使用这些新的依赖项.

以json_serializable的方式创建model类

让我们看看如何将我们的 `User` 类转换为一个 `json_serializable`。为了简单起见，我们使用前面示例中的简化JSON model。

user.dart

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

// user.g.dart 将在我们运行生成命令后自动生成
part 'user.g.dart';

///这个标注是告诉生成器，这个类是需要生成Model类的
@JsonSerializable()

class User{
  User(this.name, this.email);

  String name;
  String email;
  //不同的类使用不同的mixin即可
  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

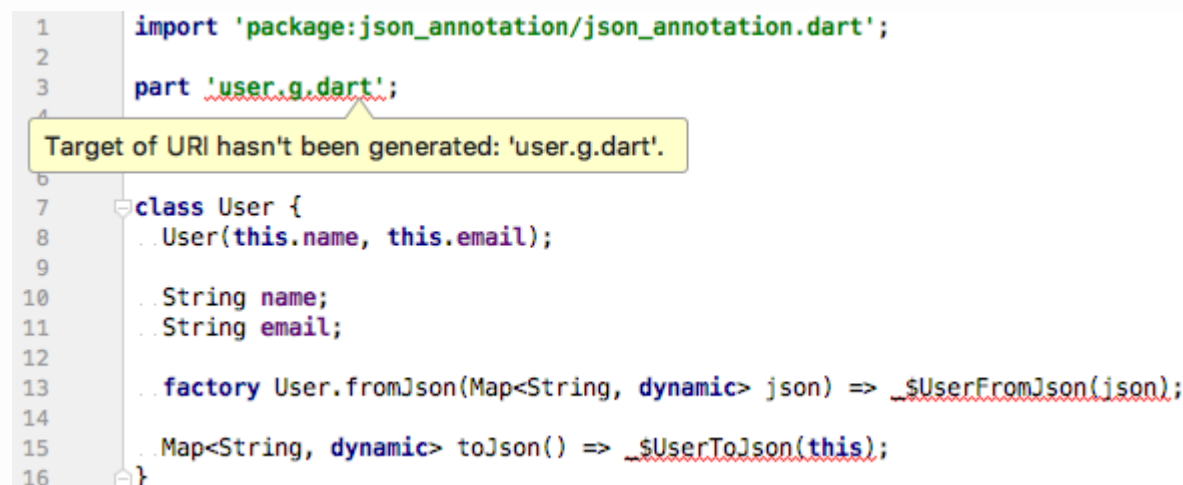
有了这个设置，源码生成器将生成用于序列化 `name` 和 `email` 字段的JSON代码。

如果需要，自定义命名策略也很容易。例如，如果我们正在使用的API返回带有 `_snake_case_` 的对象，但我们在我们的模型中使用 `_lowerCamelCase_`，那么我们可以使用 `@JsonKey` 标注：

```
/// Tell json_serializable that "registration_date_millis" should be
/// mapped to this property.
@JsonKey(name: 'registration_date_millis')
final int registrationDateMillis;
```

运行代码生成程序

`json_serializable` 第一次创建类时，您会看到与下图类似的错误。



这些错误是完全正常的，这是因为model类的生成代码还不存在。为了解决这个问题，我们必须运行代码生成器来为我们生成序列化模板。

There are two ways of running the code generator. 有两种运行代码生成器的方法：

一次性生成

通过在我们的项目根目录下运行 `flutter packages pub run build_runner build`，我们可以在需要时为我们的model生成json序列化代码。这触发了一次性构建，它通过我们的源文件，挑选相关的并为它们生成必要的序列化代码。

虽然这非常方便，但如果我们不需要每次在model类中进行更改时都要手动运行构建命令的话会更好。

持续生成

使用 `_watcher_` 可以使我们的源代码生成的过程更加方便。它会监视我们项目中文件的变化，并在需要时自动构建必要的文件。我们可以通过 `flutter packages pub run build_runner watch` 在项目根目录下运行来启动 `_watcher_`。

只需启动一次观察器，然后并让它在后台运行，这是安全的。

使用json_serializable模型

要通过 `json_serializable` 方式反序列化JSON字符串，我们不需要对先前的代码进行任何更改。

```

Map userMap = JSON.decode(json);
var user = new User.fromJson(userMap);

```

序列化也一样。调用API与之前相同。

```
String json = JSON.encode(user);
```

有了 `json_serializable`，我们可以在 `User` 类上忘记任何手动的JSON序列化。源代码生成器创建一个名为 `user.g.dart` 的文件，它具有所有必需的序列化逻辑。现在，我们不必编写自动化测试来确保序列化的正常工作 - 这个库会确保序列化工作正常。

一些参考

- [Json编解码文档](#)
- [The json_serializable package in Pub](#)
- [json_serializable examples in GitHub](#)
- [Discussion about dart:mirrors in Flutter](#)



Flutter for Android 开发者

[编辑本页](#) [提Issue](#)

本文档适用于Android开发人员，可以将您现有的Android知识应用于使用Flutter构建移动应用程序。如果您了解Android框架的基础知识，那么您可以将此文档用作Flutter开发的一个开端。

使用Flutter构建应用时，您的Android知识和技能非常宝贵，因为Flutter依赖移动操作系统提供众多功能和配置。Flutter是一种为移动设备构建用户界面的新方式，但它有一个插件系统可与Android（和iOS）进行非UI的任务通信。如果您是Android专家，则不必重新学习使用Flutter的所有内容。

可以将此文档作为cookbook，通过跳转并查找与您的需求最相关的问题。

- [Views](#)
 - [Flutter和Android中的View](#)
 - [如何更新widget](#)
 - [如何布局？XML layout 文件跑哪去了？](#)
 - [如何在布局中添加或删除组件](#)
 - [在Android中，可以通过View.animate\(\)对视图进行动画处理，那在Flutter中怎样才能对Widget进行处理](#)
 - [如何使用Canvas draw/paint](#)
 - [如何构建自定义 Widgets](#)
- [Intents](#)
 - [Intent在Flutter中等价于什么？](#)
 - [如何在Flutter中处理来自外部应用程序传入的Intents](#)
 - [startActivityForResult 在Flutter中等价于什么](#)
- [异步UI](#)
 - [runOnUiThread 在Flutter中等价于什么](#)
 - [AsyncTask和IntentService在Flutter中等价于什么](#)
 - [OkHttp在Flutter中等价于什么](#)
 - [如何在Flutter中显示进度指示器](#)
- [项目结构和资源](#)
 - [在哪里存储分辨率相关的图片文件？HDPI/XXHDPI](#)
 - [在哪里存储字符串？如何存储不同的语言](#)
 - [Android Gradle vs Flutter pubspec.yaml](#)
- [Activities 和 Fragments](#)
 - [Activity和Fragment 在Flutter中等价于什么](#)
 - [如何监听Android Activity生命周期事件](#)
- [Layouts](#)
 - [LinearLayout在Flutter中相当于什么](#)
 - [RelativeLayout在Flutter中等价于什么](#)
 - [ScrollView在Flutter中等价于什么](#)
- [手势检测和触摸事件处理](#)

- 如何将一个onClick监听器添加到Flutter中的widget
- 如何处理widget上的其他手势
- [ListView & Adapter](#)
 - [ListView在Flutter中相当于什么](#)
 - [怎么知道哪个列表项被点击](#)
 - [如何动态更新ListView](#)
- [使用 Text](#)
 - [如何在 Text widget上设置自定义字体](#)
 - [如何在Text上定义样式](#)
- [表单输入](#)
 - [Input的”hint”在flutter中相当于什么](#)
 - [如何显示验证错误](#)
- [Flutter 插件](#)
 - [如何使用 GPS sensor](#)
 - [如何访问相机](#)
 - [如何使用Facebook登陆](#)
 - [如何构建自定义集成Native功能](#)
 - [如何在我的Flutter应用程序中使用NDK](#)
- [主题](#)
 - [如何构建Material主题风格的app](#)
- [数据库和本地存储](#)
 - [如何在Flutter中访问Shared Preferences ?](#)
 - [如何在Flutter中访问SQLite](#)
- [通知](#)
 - [如何设置推送通知](#)

Views

Flutter和Android中的View

在Android中，View是屏幕上显示的所有内容的基础，按钮、工具栏、输入框等一切都是View。

在Flutter中，View相当于Widget。然而，与View相比，Widget有一些不同之处。首先，Widget仅支持一帧，并且在每一帧上，Flutter的框架都会创建一个Widget实例树(译者语：相当于一次性绘制整个界面)。相比之下，在Android上View绘制结束后，就不会重绘，直到调用invalidate时才会重绘。

与Android的视图层次系统不同（在framework改变视图），而在Flutter中的widget是不可变的，这允许widget变得超级轻量。

如何更新widget

在Android中，您可以通过直接对view进行改变来更新视图。然而，在Flutter中Widget是不可变的，不会直接更新，而必须使用Widget的状态。

这是Stateful和Stateless widget的概念的来源。一个Stateless Widget就像它的名字，是一个没有状态信息的widget。

当您所需要的用户界面不依赖于对象配置信息以外的其他任何内容时，StatelessWidgets非常有用。

例如，在Android中，这与将logo图标放在ImageView中很相似。logo在运行时不会更改，因此您可以在Flutter中使用StatelessWidget。

如果您希望通过HTTP动态请求的数据更改用户界面，则必须使用StatefulWidget，并告诉Flutter框架该widget的状态已更新，以便可以更新该widget。

这里要注意的重要一点是无状态和有状态 widget的核心特性是相同的。每一帧它们都会重新构建，不同之处在于StatefulWidget有一个State对象，它可以跨帧存储状态数据并恢复它。

如果你有疑问，那么要记住这个规则：如果一个widget发生了变化（例如用户与它交互），它就是有状态的。但是，如果一个子widget对变化做出反应，而其父widget对变化没有反应，那么包含的父widget仍然可以是无状态的widget。

我们来看看你如何使用一个StatelessWidget。一个常见的StatelessWidget是Text。如果你看看Text Widget的实现，你会发现它是一个StatelessWidget的子类：

```
new Text(  
  'I like Flutter!',  
  style: new TextStyle(fontWeight: FontWeight.bold),  
);
```

正如你所看到的，Text 没有与之关联的状态信息，它呈现了构造函数中传递的内容，仅此而已。

但是，如果你想让“I Like Flutter”动态变化，例如点击一个FloatingActionButton？这可以通过将Text包装在StatefulWidget中并在点击按钮时更新它来实现，如：

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(new SampleApp());  
}  
  
class SampleApp extends StatelessWidget {  
  // This widget is the root of your application.  
  @override  
  Widget build(BuildContext context) {  
    return new MaterialApp(  
      title: 'Sample App',  
      theme: new ThemeData(  
        primarySwatch: Colors.blue,
```

```

    ),
    home: new SampleAppPage(),
  );
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  // Default placeholder text
  String textToShow = "I Like Flutter";

  void _updateText() {
    setState(() {
      // update the text
      textToShow = "Flutter is Awesome!";
    });
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Sample App"),
      ),
      body: new Center(child: new Text(textToShow)),
      floatingActionButton: new FloatingActionButton(
        onPressed: _updateText,
        tooltip: 'Update Text',
        child: new Icon(Icons.update),
      ),
    );
  }
}

```

如何布局？ XML layout 文件跑哪去了？

在Android中，您通过XML编写布局，但在Flutter中，您可以使用widget树来编写布局。

这里是一个例子，展示了如何在屏幕上显示一个简单的Widget并添加一些padding。

```

@override
Widget build(BuildContext context) {
  return new Scaffold(
    appBar: new AppBar(
      title: new Text("Sample App"),

```

```

    ),
    body: new Center(
      child: new MaterialButton(
        onPressed: () {},
        child: new Text('Hello'),
        padding: new EdgeInsets.only(left: 10.0, right: 10.0),
      ),
    ),
  );
}

```

您可以查看Flutter所提供的所有布局: [Flutter widget layout](#)

如何在布局中添加或删除组件

在Android中，您可以从父级控件调用addChild或removeChild以动态添加或删除View。在Flutter中，因为widget是不可变的，所以没有addChild。相反，您可以传入一个函数，该函数返回一个widget给父项，并通过布尔值控制该widget的创建。

例如，当你点击一个FloatingActionButton时，如何在两个widget之间切换：

```

import 'package:flutter/material.dart';

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  // Default value for toggle
  bool toggle = true;
}

```



```

void _toggle() {
  setState(() {
    toggle = !toggle;
  });
}

_getToggleChild() {
  if (toggle) {
    return new Text('Toggle One');
  } else {
    return new MaterialButton(onPressed: () {}, child: new Text('Toggle Two'));
  }
}

@override
Widget build(BuildContext context) {
  return new Scaffold(
    appBar: new AppBar(
      title: new Text("Sample App"),
    ),
    body: new Center(
      child: _getToggleChild(),
    ),
    floatingActionButton: new FloatingActionButton(
      onPressed: _toggle,
      tooltip: 'Update Text',
      child: new Icon(Icons.update),
    ),
  );
}

```

在Android中，可以通过View.animate()对视图进行动画处理，那在Flutter中怎样才能对Widget进行处理

在Flutter中，可以通过动画库给widget添加动画。

在Android中，您可以通过XML创建动画或在视图上调用.animate()。在Flutter中，您可以将widget包装到Animation中。

与Android相似，在Flutter中，您有一个AnimationController和一个Interpolator，它是Animation类的扩展，例如CurvedAnimation。您将控制器和动画传递到AnimationWidget中，并告诉控制器启动动画。

让我们来看看如何编写一个FadeTransition，当您按下时会淡入一个logo：

```

import 'package:flutter/material.dart';

void main() {
  runApp(new FadeAppTest());
}

```

```

class FadeAppTest extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Fade Demo',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new MyFadeTest(title: 'Fade Demo'),
    );
  }
}

class MyFadeTest extends StatefulWidget {
  MyFadeTest({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyFadeTest createState() => new _MyFadeTest();
}

class _MyFadeTest extends State<MyFadeTest> with TickerProviderStateMixin {
  AnimationController controller;
  CurvedAnimation curve;

  @override
  void initState() {
    controller = new AnimationController(duration: const Duration(milliseconds: 2000), vsync: this);
    curve = new CurvedAnimation(parent: controller, curve: Curves.easeIn);
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ),
      body: new Center(
        child: new Container(
          child: new FadeTransition(
            opacity: curve,
            child: new FlutterLogo(
              size: 100.0,
            )),
        ),
      floatingActionButton: new FloatingActionButton(
        tooltip: 'Fade',
        child: new Icon(Icons.brush),
        onPressed: () {
          controller.forward();
        },
      ),
    );
  }
}

```

}

See <https://flutter.io/widgets/animation/> and <https://flutter.io/tutorials/animation> for more specific details.

如何使用Canvas draw/paint

在Android中，您可以使用Canvas在屏幕上绘制自定义形状。

Flutter有两个类可以帮助您绘制画布，CustomPaint和CustomPainter，它们实现您的算法以绘制到画布。

在这个人气较高的StackOverFlow答案中，您可以看到签名painter是如何实现的。

请参阅<https://stackoverflow.com/questions/46241071/create-signature-area-for-mobile-app-in-dart-flutter>

```
import 'package:flutter/material.dart';
class SignaturePainter extends CustomPainter {
  SignaturePainter(this.points);
  final List<Offset> points;
  void paint(Canvas canvas, Size size) {
    var paint = new Paint()
      ..color = Colors.black
      ..strokeCap = StrokeCap.round
      ..strokeWidth = 5.0;
    for (int i = 0; i < points.length - 1; i++) {
      if (points[i] != null && points[i + 1] != null)
        canvas.drawLine(points[i], points[i + 1], paint);
    }
  }
  bool shouldRepaint(SignaturePainter other) => other.points != points;
}
class Signature extends StatefulWidget {
  SignatureState createState() => new SignatureState();
}
class SignatureState extends State<Signature> {
  List<Offset> _points = <Offset>[];
  Widget build(BuildContext context) {
    return new GestureDetector(
      onPanUpdate: (DragUpdateDetails details) {
        setState(() {
          RenderBox referenceBox = context.findRenderObject();
          Offset localPosition =
            referenceBox.globalToLocal(details.globalPosition);
          _points = new List.from(_points)..add(localPosition);
        });
      },
      onPanEnd: (DragEndDetails details) => _points.add(null),
      child: new CustomPaint(painter: new SignaturePainter(_points)),
    );
  }
}
class DemoApp extends StatelessWidget {
```

```
Widget build(BuildContext context) => new Scaffold(body: new Signature());
}
void main() => runApp(new MaterialApp(home: new DemoApp()));
```

如何构建自定义 Widgets

在Android中，您通常会继承View或已经存在的某个控件，然后覆盖其绘制方法来实现自定义View。

在Flutter中，一个自定义widget通常是通过组合其它widget来实现的，而不是继承。

我们来看看如何构建持有一个label的CustomButton。这是通过将Text与RaisedButton组合来实现的，而不是扩展RaisedButton并重写其绘制方法实现：

```
class CustomButton extends StatelessWidget {
  final String label;
  CustomButton(this.label);

  @override
  Widget build(BuildContext context) {
    return new RaisedButton(onPressed: () {}, child: new Text(label));
  }
}
```

Then you can use this CustomButton just like you would with any other Widget:

```
@override
Widget build(BuildContext context) {
  return new Center(
    child: new CustomButton("Hello"),
  );
}
```

Intents

Intent在Flutter中等价于什么？

在Android中，Intents主要有两种使用场景：在Activity之间切换，以及调用外部组件。Flutter不具有Intents的概念，但如果需要的话，Flutter可以通过Native整合来触发Intents。

要在Flutter中切换屏幕，您可以访问路由以绘制新的Widget。管理多个屏幕有两个核心概念和类：Route 和 Navigator。Route是应用程序的“屏幕”或“页面”的抽象（可以认为是Activity），Navigator是管理Route的Widget。Navigator可以通过push和pop route以实现页面切换。

和Android相似，您可以在AndroidManifest.xml中声明您的Activities，在Flutter中，您可以将具有指定Route的Map传递到顶层MaterialApp实例

```
void main() {
  runApp(new MaterialApp(
    home: new MyAppHome(), // becomes the route named '/'
    routes: <String, WidgetBuilder> {
      '/a': (BuildContext context) => new MyPage(title: 'page A'),
      '/b': (BuildContext context) => new MyPage(title: 'page B'),
      '/c': (BuildContext context) => new MyPage(title: 'page C'),
    },
  ));
}
```

然后，您可以通过**Navigator**来切换到命名路由的页面。

```
Navigator.of(context).pushNamed('/b');
```

Intents的另一个主要的用途是调用外部组件，如**Camera**或**File picker**。为此，您需要和**native**集成（或使用现有的库）

参阅 [Flutter Plugins] 了解如何与**native**集成.

如何在Flutter中处理来自外部应用程序传入的Intents

Flutter可以通过直接与Android层通信并请求共享的数据来处理来自Android的Intents

在这个例子中，我们注册文本共享**intent**，所以其他应用程序可以共享文本到我们的Flutter应用程序

这个应用程序的基本流程是我们首先处理**Android**端的共享文本数据，然后等待Flutter请求数据，然后通过**MethodChannel**发送。

首先在在**AndroidManifest.xml**中注册我们想要处理的**intent**

```
<activity
  android:name=".MainActivity"
  android:launchMode="singleTop"
  android:theme="@style/LaunchTheme"
  android:configChanges="orientation|keyboardHidden|keyboard|screenSize|locale|layoutDirection"
  android:hardwareAccelerated="true"
  android:windowSoftInputMode="adjustResize">
  <!-- This keeps the window background of the activity showing
        until Flutter renders its first frame. It can be removed if
        there is no splash screen (such as the default splash screen
        defined in @style/LaunchTheme). -->
  <meta-data
    android:name="io.flutter.app.android.SplashScreenUntilFirstFrame"
    android:value="true" />
```

```

<intent-filter>
  <action android:name="android.intent.action.MAIN"/>
  <category android:name="android.intent.category.LAUNCHER"/>
</intent-filter>
<intent-filter>
  <action android:name="android.intent.action.SEND" />
  <category android:name="android.intent.category.DEFAULT" />
  <data android:mimeType="text/plain" />
</intent-filter>
</activity>

```

然后，在MainActivity中，您可以处理intent，一旦我们从intent中获得共享文本数据，我们就会持有它，直到Flutter在完成准备就绪时请求它。

```

package com.yourcompany.shared;

import android.content.Intent;
import android.os.Bundle;

import java.nio.ByteBuffer;

import io.flutter.app.FlutterActivity;
import io.flutter.plugin.common.ActivityLifecycleListener;
import io.flutter.plugin.common.MethodCall;
import io.flutter.plugin.common.MethodChannel;
import io.flutter.plugins.GeneratedPluginRegistrant;

public class MainActivity extends FlutterActivity {
  String sharedText;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    GeneratedPluginRegistrant.registerWith(this);
    Intent intent = getIntent();
    String action = intent.getAction();
    String type = intent.getType();

    if (Intent.ACTION_SEND.equals(action) && type != null) {
      if ("text/plain".equals(type)) {
        handleSendText(intent); // Handle text being sent
      }
    }

    new MethodChannel(getFlutterView(), "app.channel.shared.data").setMethodCallHandler(
      new MethodChannel.MethodCallHandler() {
        @Override
        public void onMethodCall(MethodCall methodCall, MethodChannel.Result result)
        {
          if (methodCall.method.equals("getSharedText")) {
            result.success(sharedText);
            sharedText = null;

```

```

    }

    }

  });
}

void handleSendText(Intent intent) {
  sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
}
}

```

最后，在Flutter中，您可以在渲染Flutter视图时请求数据。

```

import 'package:flutter/material.dart';
import 'package:flutter/services.dart';

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample Shared App Handler',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  static const platform = const MethodChannel('app.channel.shared.data');
  String dataShared = "No data";

  @override
  void initState() {
    super.initState();
    getSharedText();
  }

  @override

```

```
Widget build(BuildContext context) {
  return new Scaffold(body: new Center(child: new Text(dataShared)));
}

getSharedText() async {
  var sharedData = await platform.invokeMethod("getSharedText");
  if (sharedData != null) {
    setState(() {
      dataShared = sharedData;
    });
  }
}
```

startActivityForResult 在Flutter中等价于什么

处理Flutter中所有路由的Navigator类可用于从已经push到栈的路由中获取结果。这可以通过等待push返回的Future来完成。例如，如果您要启动让用户选择其位置的地理位置的路由，则可以执行以下操作：

```
Map coordinates = await Navigator.of(context).pushNamed('/location');
```

然后在你的位置路由中，一旦用户选择了他们的位置，你可以将结果”pop”出栈

```
Navigator.of(context).pop({"lat":43.821757,"long":-79.226392});
```

异步UI

runOnUiThread 在Flutter中等价于什么

Dart是单线程执行模型，支持Isolates（在另一个线程上运行Dart代码的方式）、事件循环和异步编程。除非您启动一个Isolate，否则您的Dart代码将在主UI线程中运行，并由事件循环驱动（译者语：和JavaScript一样）。

例如，您可以在UI线程上运行网络请求代码而不会导致UI挂起(译者语：因为网络请求是异步的)：

```
loadData() async {
  String dataURL = "https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = JSON.decode(response.body);
  });
}
```

要更新UI，您可以调用setState，这会触发build方法再次运行并更新数据。

以下是异步加载数据并将其显示在ListView中的完整示例：

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();

    loadData();
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Sample App"),
      ),
      body: new ListView.builder(
        itemCount: widgets.length,
        itemBuilder: (BuildContext context, int position) {
          return getRow(position);
        }
      ));
  }
}
```

```

Widget getRow(int i) {
  return new Padding(
    padding: new EdgeInsets.all(10.0),
    child: new Text("Row ${widgets[i]["title"]}"),
  );
}

loadData() async {
  String dataURL = "https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = JSON.decode(response.body);
  });
}
}

```

AsyncTask和IntentService在Flutter中等价于什么

在Android中，当你想访问一个网络资源时，你通常会创建一个AsyncTask，它将在UI线程之外运行代码来防止你的UI被阻塞。AsyncTask有一个线程池，可以为你管理线程。

由于Flutter是单线程的，运行一个事件循环（如Node.js），所以您不必担心线程管理或者使用AsyncTasks、IntentServices。

要异步运行代码，可以将函数声明为异步函数，并在该函数中等待这个耗时任务

```

loadData() async {
  String dataURL = "https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = JSON.decode(response.body);
  });
}

```

这就是典型的进行网络或数据库调用的方式

在Android上，当您继承AsyncTask时，通常会覆盖3个方法，OnPreExecute、doInBackground和onPostExecute。在Flutter中没有这种模式的等价物，因为您只需等待一个长时间运行的函数，而Dart的事件循环将负责其余的事情。

但是，有时您可能需要处理大量数据，导致UI可能会挂起。

在这种情况下，与AsyncTask一样，在Flutter中，可以利用多个CPU内核来执行耗时或计算密集型任务。这是通过使用Isolates来完成的。

是一个独立的执行线程，它运行时不会与主线程共享任何内存。这意味着你不能从该线程访问变量或通过调用setState来更新你的UI。

我们来看一个简单的`Isolate`的例子，以及如何与主线程通信和共享数据以更新UI：

```
loadData() async {
  ReceivePort receivePort = new ReceivePort();
  await Isolate.spawn(dataLoader, receivePort.sendPort);

  // The 'echo' isolate sends it's SendPort as the first message
  SendPort sendPort = await receivePort.first;

  List msg = await sendReceive(sendPort, "https://jsonplaceholder.typicode.com/posts");

  setState(() {
    widgets = msg;
  });
}

// the entry point for the isolate
static dataLoader(SendPort sendPort) async {
  // Open the ReceivePort for incoming messages.
  ReceivePort port = new ReceivePort();

  // Notify any other isolates what port this isolate listens to.
  sendPort.send(port.sendPort);

  await for (var msg in port) {
    String data = msg[0];
    SendPort replyTo = msg[1];

    String dataURL = data;
    http.Response response = await http.get(dataURL);
    // Lots of JSON to parse
    replyTo.send(JSON.decode(response.body));
  }
}

Future sendReceive(SendPort port, msg) {
  ReceivePort response = new ReceivePort();
  port.send([msg, response.sendPort]);
  return response.first;
}
```

“`dataLoader`”是在它自己的独立执行线程中运行的隔离区，您可以在其中执行CPU密集型任务，例如解析大于1万的JSON或执行计算密集型数学计算。

下面是一个可以运行的完整示例：

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:async';
```

```
import 'dart:isolate';

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }

  showLoadingDialog() {
    if (widgets.length == 0) {
      return true;
    }

    return false;
  }

  getBody() {
    if (showLoadingDialog()) {
      return getProgressDialog();
    } else {
      return getListView();
    }
  }

  getProgressDialog() {
    return new Center(child: new CircularProgressIndicator());
  }
}
```

```

@override
Widget build(BuildContext context) {
  return new Scaffold(
    appBar: new AppBar(
      title: new Text("Sample App"),
    ),
    body: getBody());
}

ListView getListView() => new ListView.builder(
  itemCount: widgets.length,
  itemBuilder: (BuildContext context, int position) {
    return getRow(position);
  });

Widget getRow(int i) {
  return new Padding(padding: new EdgeInsets.all(10.0), child: new Text("Row ${widgets[
i]["title"]}"));
}

loadData() async {
  ReceivePort receivePort = new ReceivePort();
  await Isolate.spawn(dataLoader, receivePort.sendPort);

  // The 'echo' isolate sends it's SendPort as the first message
  SendPort sendPort = await receivePort.first;

  List msg = await sendReceive(sendPort, "https://jsonplaceholder.typicode.com/posts");

  setState(() {
    widgets = msg;
  });
}

// the entry point for the isolate
static dataLoader(SendPort sendPort) async {
  // Open the ReceivePort for incoming messages.
  ReceivePort port = new ReceivePort();

  // Notify any other isolates what port this isolate listens to.
  sendPort.send(port.sendPort);

  await for (var msg in port) {
    String data = msg[0];
    SendPort replyTo = msg[1];

    String dataURL = data;
    http.Response response = await http.get(dataURL);
    // Lots of JSON to parse
    replyTo.send(JSON.decode(response.body));
  }
}

Future sendReceive(SendPort port, msg) {

```

```

ReceivePort response = new ReceivePort();
port.send([msg, response.sendPort]);
return response.first;
}

}

```

OkHttp在Flutter中等价于什么

当使用受欢迎的“http”package时，Flutter进行网络信非常简单。

虽然“http” package 没有实现OkHttp的所有功能，但“http” package 抽象出了许多常用的API，可以简单有效的发起网络请求。

<https://pub.dartlang.org/packages/http>

您可以通过在pubspec.yaml中添加依赖项来使用它

```

dependencies:
  ...
  http: '>=0.11.3+12'

```

然后就可以进行网络调用，例如请求GitHub上的这个JSON GIST：

```

import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
[...]
loadData() async {
  String dataURL = "https://jsonplaceholder.typicode.com/posts";
  http.Response response = await http.get(dataURL);
  setState(() {
    widgets = JSON.decode(response.body);
  });
}
}

```

一旦获得结果后，您可以通过调用setState来告诉Flutter更新其状态，setState将使用网络调用的结果更新您的UI。

如何在Flutter中显示进度指示器

在Android中，当您执行耗时任务时，通常会显示进度指示器。

在Flutter中，这可以通过渲染Progress Indicator widget来实现。您可以通过编程方式显示Progress Indicator

，通过布尔值通知Flutter在耗时任务发起之前更新其状态。

在下面的例子中，我们将build函数分解为三个不同的函数。如果showLoadingDialog为true（当widgets.length == 0时），那么我们展示ProgressIndicator，否则我们将展示包含所有数据的ListView。

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    loadData();
  }

  showLoadingDialog() {
    if (widgets.length == 0) {
      return true;
    }

    return false;
  }

  getBody() {
    if (showLoadingDialog()) {
```

```

        return getProgressDialog();
    } else {
        return getListView();
    }
}

getProgressDialog() {
    return new Center(child: new CircularProgressIndicator());
}

@override
Widget build(BuildContext context) {
    return new Scaffold(
        appBar: new AppBar(
            title: new Text("Sample App"),
        ),
        body: getBody());
}

ListView getListView() => new ListView.builder(
    itemCount: widgets.length,
    itemBuilder: (BuildContext context, int position) {
        return getRow(position);
    });

Widget getRow(int i) {
    return new Padding(padding: new EdgeInsets.all(10.0), child: new Text("Row ${widgets[i]["title"]}"));
}

loadData() async {
    String dataURL = "https://jsonplaceholder.typicode.com/posts";
    http.Response response = await http.get(dataURL);
    setState(() {
        widgets = JSON.decode(response.body);
    });
}
}

```

项目结构和资源

在哪里存储分辨率相关的图片文件? HDPI/XXHDPI

Flutter遵循像iOS这样简单的3种分辨率格式: 1x, 2x, and 3x.

创建一个名为images的文件夹, 并为每个图像文件生成一个@2x和@3x文件, 并将它们放置在如下这样的文件夹中

- .../my_icon.png
- .../2.0x/my_icon.png
- .../3.0x/my_icon.png

然后，您需要在`pubspec.yaml`文件中声明这些图片

```
assets:  
  - images/a_dot_burr.jpeg  
  - images/a_dot_ham.jpeg
```

然后您可以使用`AssetImage`访问您的图像

```
return new AssetImage("images/a_dot_burr.jpeg");
```

在哪里存储字符串? 如何存储不同的语言

目前，最好的做法是创建一个名为`Strings`的类

```
class Strings{  
  static String welcomeMessage = "Welcome To Flutter";  
}
```

然后在你的代码中，你可以像访问你的字符串一样：

```
new Text(Strings.welcomeMessage)
```

Flutter对Android的可访问性提供了基本的支持，虽然这个功能正在进行中。

鼓励Flutter开发者使用[intl package](#) 进行国际化和本地化

Android Gradle vs Flutter pubspec.yaml

在Android中，您可以在Gradle文件来添加依赖项。

在Flutter中，虽然在Flutter项目中的Android文件夹下有Gradle文件，但只有在添加平台相关所需的依赖关系时才使用这些文件。 否则，应该使用`pubspec.yaml`声明用于Flutter的外部依赖项。

发现好的flutter packages的一个好地方 [Pub](#)

Activities 和 Fragments

Activity和Fragment 在Flutter中等价于什么

在Android中，`Activity`代表用户可以完成的一项重点工作。`Fragment`代表了一种模块化代码的方式，可以为大屏幕设备构建更复杂的用户界面，可以在小屏幕和大屏幕之间自动调整UI。 在Flutter中，这两个概念都等同

于Widget。

如何监听Android Activity生命周期事件

在Android中，您可以覆盖Activity的方法来捕获Activity的生命周期回调。

在Flutter中您可以通过挂接到WidgetsBinding观察并监听didChangeAppLifecycleState更改事件来监听生命周期事件

您可以监听到的生命周期事件是

- resumed - 应用程序可见并响应用户输入。这是来自Android的onResume
- inactive - 应用程序处于非活动状态，并且未接收用户输入。此事件在Android上未使用，仅适用于iOS
- paused - 应用程序当前对用户不可见，不响应用户输入，并在后台运行。这是来自Android的暂停
- suspending - 该应用程序将暂时中止。这在iOS上未使用

```
import 'package:flutter/widgets.dart';

class LifecycleWatcher extends StatefulWidget {
  @override
  _LifecycleWatcherState createState() => new _LifecycleWatcherState();
}

class _LifecycleWatcherState extends State<LifecycleWatcher> with WidgetsBindingObserver {
  AppLifecycleState _lastLifecycleState;

  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this);
  }

  @override
  void dispose() {
    WidgetsBinding.instance.removeObserver(this);
    super.dispose();
  }

  @override
  void didChangeAppLifecycleState(AppLifecycleState state) {
    setState(() {
      _lastLifecycleState = state;
    });
  }

  @override
  Widget build(BuildContext context) {
    if (_lastLifecycleState == null)
      return new Text('This widget has not observed any lifecycle changes.', textDirection: TextDirection.ltr);
  }
}
```

```

        return new Text('The most recent lifecycle state this widget observed was: $_lastLifecycleState.',
            textDirection: TextDirection.ltr);
    }
}

void main() {
    runApp(new Center(child: new LifecycleWatcher()));
}

```

Layouts

LinearLayout在Flutter中相当于什么

在Android中，使用LinearLayout来使您的控件呈水平或垂直排列。在Flutter中，您可以使用Row或Column来实现相同的结果。

```

@override
Widget build(BuildContext context) {
    return new Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            new Text('Row One'),
            new Text('Row Two'),
            new Text('Row Three'),
            new Text('Row Four'),
        ],
    );
}

```

```

@override
Widget build(BuildContext context) {
    return new Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
            new Text('Column One'),
            new Text('Column Two'),
            new Text('Column Three'),
            new Text('Column Four'),
        ],
    );
}

```

RelativeLayout在Flutter中等价于什么

RelativeLayout用于使widget相对于彼此位置排列。在Flutter中，有几种方法可以实现相同的结果

Column Row Stack

RelativeLayout

widget

您可以通过使用 `Row`、`Column` 和 `Stack` 的组合来实现 `RelativeLayout` 的效果。您可以为 `Widget` 构造函数指定相对于父组件的布局规则。

一个在Flutter中构建RelativeLayout的好例子，请参考在StackOverflow上 <https://stackoverflow.com/questions/44396075/equivalent-of-relativelayout-in-flutter>

ScrollView在Flutter中等价于什么

在Android中，`ScrollView`允许您包含一个子控件，以便在用户设备的屏幕比控件内容小的情况下，使它们可以滚动。

在Flutter中，最简单的方法是使用`ListView`。但在Flutter中，一个`ListView`既是一个`ScrollView`，也是一个Android `ListView`。

```
@override
Widget build(BuildContext context) {
  return new ListView(
    children: <Widget>[
      new Text('Row One'),
      new Text('Row Two'),
      new Text('Row Three'),
      new Text('Row Four'),
    ],
  );
}
```

手势检测和触摸事件处理

如何将一个onClick监听器添加到Flutter中的widget

在Android中，您可以通过调用方法 `setOnClickListener` 将`OnClick`绑定到按钮等view上。

在Flutter中，添加触摸监听器有两种方法：

1. 如果Widget支持事件监听，则可以将一个函数传递给它并进行处理。例如，`RaisedButton`有一个`onPressed`参数

```
@override
Widget build(BuildContext context) {
  return new RaisedButton(
    onPressed: () {
      print("click");
    },
    child: new Text("Button"));
}
```

如果 `FlutterLogo` 不支持事件监听，则可以将该 `FlutterLogo` 包装到 `GestureDetector` 中，并将处理函数传递给 `FlutterLogo` 参数。

```
class SampleApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return new Scaffold(  
      body: new Center(  
        child: new GestureDetector(  
          child: new FlutterLogo(  
            size: 200.0,  
          ),  
          onTap: () {  
            print("tap");  
          },  
        ),  
      ),  
    );  
  }  
}
```

如何处理widget上的其他手势

使用 `GestureDetector`，可以监听多种手势，例如

- Tap
 - `onTapDown`
 - `onTapUp`
 - `onTap`
 - `onTapCancel`
- Double tap
 - `onDoubleTap` 用户快速连续两次在同一位置轻敲屏幕。
- 长按
 - `onLongPress`
- 垂直拖动
 - `onVerticalDragStart`
 - `onVerticalDragUpdate`
 - `onVerticalDragEnd`
- 水平拖拽
 - `onHorizontalDragStart`
 - `onHorizontalDragUpdate`
 - `onHorizontalDragEnd`

For example here is a `GestureDetector` for double tap on the `FlutterLogo` that will make it rotate 例如，这里

是一个`GestureDetector`，用于监听FlutterLogo的双击事件，双击时使其旋转

```

AnimationController controller;
CurvedAnimation curve;

@override
void initState() {
  controller = new AnimationController(duration: const Duration(milliseconds: 2000), vsync: this);
  curve = new CurvedAnimation(parent: controller, curve: Curves.easeIn);
}

class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      body: new Center(
        child: new GestureDetector(
          child: new RotationTransition(
            turns: curve,
            child: new FlutterLogo(
              size: 200.0,
            ),
          ),
          onTap: () {
            if (controller.isCompleted) {
              controller.reverse();
            } else {
              controller.forward();
            }
          },
        ),
      ),
    );
  }
}

```

Listview & Adapter

ListView在Flutter中相当于什么

在Flutter中，ListView就是一个ListView！

在Android ListView中，您可以创建一个适配器，然后您可以将它传递给ListView，该适配器将使用适配器返回的内容来展示每一行。然而，你必须确保在合适的时机回收行，否则，你会得到各种疯狂的视觉和内存问题。

在Flutter中，由于Flutter的不可变的widget模型，将一个Widgets列表传递给ListView，而Flutter将负责确保它们快速平滑地滚动。

```
import 'package:flutter/material.dart';
```

```

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Sample App"),
      ),
      body: new ListView(children: _getListData()),
    );
  }

  _getListData() {
    List<Widget> widgets = [];
    for (int i = 0; i < 100; i++) {
      widgets.add(new Padding(padding: new EdgeInsets.all(10.0), child: new Text("Row $i"
    )));
    }
    return widgets;
  }
}

```

怎么知道哪个列表项被点击

在Android中，ListView有一个方法'onItemClickListener'来确定哪个列表项被点击。Flutter中可以更轻松地通过您传入的处理回调来进行操作。

```
import 'package:flutter/material.dart';

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Sample App"),
      ),
      body: new ListView(children: _getListData()),
    );
  }

  _getListData() {
    List<Widget> widgets = [];
    for (int i = 0; i < 100; i++) {
      widgets.add(new GestureDetector(
        child: new Padding(
          padding: new EdgeInsets.all(10.0),
          child: new Text("Row $i"),
        ),
        onTap: () {
          print('row tapped');
        },
      ));
    }
    return widgets;
  }
}
```


如何动态更新ListView

需要更新适配器并调用`notifyDataSetChanged`。在Flutter中，如果`setState()`中更新widget列表，您会发现没有变化，这是因为当`setState`被调用时，Flutter渲染引擎会遍历所有的widget以查看它们是否已经改变。当遍历到你的ListView时，它会做一个`==` 运算，以查看两个ListView是否相同，因为没有任何改变，因此没有更新数据。

要更新您的ListView，然后在`setState`中创建一个新的List（）并将所有旧数据复制到新列表中。这是实现更新的简单方法（译者语：此时状态改变，ListView被重新构建）

```
import 'package:flutter/material.dart';

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    for (int i = 0; i < 100; i++) {
      widgets.add(getRow(i));
    }
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
```

```

    appBar: new AppBar(
      title: new Text("Sample App"),
    ),
    body: new ListView(children: widgets),
  );
}

Widget getRow(int i) {
  return new GestureDetector(
    child: new Padding(
      padding: new EdgeInsets.all(10.0),
      child: new Text("Row $i")),
    onTap: () {
      setState(() {
        widgets = new List.from(widgets);
        widgets.add(getRow(widgets.length + 1));
        print('row $i');
      });
    },
  );
}
}

```

然而，推荐的方法是使用`ListView.Builder`。当您拥有动态列表或包含大量数据的列表时，此方法非常有用。这实际上相当于在Android上使用`RecyclerView`，它会自动为您回收列表元素：

```

import 'package:flutter/material.dart';

void main() {
  runApp(new SampleApp());
}

class SampleApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: new SampleAppPage(),
    );
  }
}

class SampleAppPage extends StatefulWidget {
  SampleAppPage({Key key}) : super(key: key);

  @override
  _SampleAppPageState createState() => new _SampleAppPageState();
}

```

```

class _SampleAppPageState extends State<SampleAppPage> {
  List widgets = [];

  @override
  void initState() {
    super.initState();
    for (int i = 0; i < 100; i++) {
      widgets.add(getRow(i));
    }
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Sample App"),
      ),
      body: new ListView.builder(
        itemCount: widgets.length,
        itemBuilder: (BuildContext context, int position) {
          return getRow(position);
        }
      ));

    Widget getRow(int i) {
      return new GestureDetector(
        child: new Padding(
          padding: new EdgeInsets.all(10.0),
          child: new Text("Row $i")),
        onTap: () {
          setState(() {
            widgets.add(getRow(widgets.length + 1));
            print('row $i');
          });
        },
      );
    }
  }
}

```

我们不是创建一个“新的ListView”，而是创建一个新的ListView.builder，它接受两个参数，即列表的初始长度和一个ItemBuilder函数。

ItemBuilder函数非常类似于Android适配器中的getView函数，它需要一个位置并返回要为该位置渲染的行。

最后，但最重要的是，如果您注意到onTap函数，在里面，我们不会再重新创建列表，而只是添加新元素到列表。

使用 Text

如何在 Text widget上设置自定义字体

在Android SDK（从Android O开始）中，创建一个Font资源文件并将其传递到TextView的FontFamily参数中。

在Flutter中，首先你需要把你的字体文件放在项目文件夹中（最好的做法是创建一个名为assets的文件夹）

接下来在pubspec.yaml文件中，声明字体：

```
fonts:
  - family: MyCustomFont
    fonts:
      - asset: fonts/MyCustomFont.ttf
      - style: italic
```

最后，将字体应用到Text widget:

```
@override
Widget build(BuildContext context) {
  return new Scaffold(
    appBar: new AppBar(
      title: new Text("Sample App"),
    ),
    body: new Center(
      child: new Text(
        'This is a custom font text',
        style: new TextStyle(fontFamily: 'MyCustomFont'),
      ),
    ),
  );
}
```

如何在Text上定义样式

除了自定义字体，您还可在Text上自定义很多不同的样式

Text的样式参数需要一个TextStyle对象，您可以在其中自定义许多参数，如

- color
- decoration
- decorationColor
- decorationStyle
- fontFamily
- fontSize
- fontStyle
- fontWeight
- hashCode
- height

- inherit
- letterSpacing
- textBaseline
- wordSpacing

表单输入

Input的”hint”在flutter中相当于什么

在Flutter中，您可以通过向Text Widget的装饰构造函数参数添加InputDecoration对象，轻松地输入框显示占位符文本

```
body: new Center(  
  child: new TextField(  
    decoration: new InputDecoration(hintText: "This is a hint"),  
  )  
)
```

如何显示验证错误

就像您如何使用“hint”一样，您可以将InputDecoration对象传递给Text的装饰构造函数。

但是，您不希望首先显示错误，并且通常会在用户输入一些无效数据时显示该错误。这可以通过更新状态并传递一个新的InputDecoration对象来完成

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(new SampleApp());  
}  
  
class SampleApp extends StatelessWidget {  
  // This widget is the root of your application.  
  @override  
  Widget build(BuildContext context) {  
    return new MaterialApp(  
      title: 'Sample App',  
      theme: new ThemeData(  
        primarySwatch: Colors.blue,  
      ),  
      home: new SampleAppPage(),  
    );  
  }  
}  
  
class SampleAppPage extends StatefulWidget {  
  SampleAppPage({Key key}) : super(key: key);
```

```

@override
_SampleAppPageState createState() => new _SampleAppPageState();
}

class _SampleAppPageState extends State<SampleAppPage> {
  String _errorText;

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text("Sample App"),
      ),
      body: new Center(
        child: new TextField(
          onSubmitted: (String text) {
            setState(() {
              if (!isEmail(text)) {
                _errorText = 'Error: This is not an email';
              } else {
                _errorText = null;
              }
            });
          },
          decoration: new InputDecoration(hintText: "This is a hint", errorText: _getErrorText()),
        ),
      ),
    );
  }

  _getErrorText() {
    return _errorText;
  }

  bool isEmail(String em) {
    String emailRegexp =
      r'^((^[<>() []\.\.,;:\s@""]+(\.[^<>() []\.\.,;:\s@""]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\)|((\[a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$';

    RegExp regExp = new RegExp(p);

    return regExp.hasMatch(em);
  }
}

```

Flutter 插件

如何使用 GPS sensor

要访问GPS传感器，您可以使用社区插件 <https://pub.dartlang.org/packages/location>

如何访问相机

访问相机的流行社区插件是 https://pub.dartlang.org/packages/image_picker

如何使用Facebook登陆

要访问Facebook Connect功能，您可以使用 https://pub.dartlang.org/packages/flutter_facebook_connect .

如何构建自定义集成Native功能

如果有Flutter或其社区插件缺失的平台特定功能，那么您可以自己按照以下教程构建 [开发packages](#) .

简而言之，Flutter的插件架构就像在Android中使用Event bus一样：您可以发出消息并让接收者进行处理并将结果返回给您，在这种情况下，接收者将是iOS或Android。

如何在我的Flutter应用程序中使用NDK

自定义插件首先会与Android应用程序通信，您可以在其中调用native标记的函数。一旦Native完成了相应操作，就可以将响应消息发回给Flutter并呈现结果。

主题

如何构建Material主题风格的app

Flutter很好的实现了一个美丽的Material Design，它会满足很多样式和主题的需求。与Android中使用XML声明主题不同，在Flutter中，您可以通过顶层widget声明主题。

MaterialApp是一个方便的widget，它包装了许多Material Design应用通常需要的widget，它通过添加Material特定功能构建在WidgetsApp上。

如果你不想使用Material Components，那么你可以声明一个顶级widget-WidgetsApp，它是一个便利的类，它包装了许多应用程序通常需要的widget

要自定义Material Components的颜色和样式，您可以将ThemeData对象传递到MaterialApp widget中，例如在下面的代码中，您可以看到主色板设置为蓝色，并且所有选择区域的文本颜色都应为红色。

```
class SampleApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Sample App',
      theme: new ThemeData(
        primarySwatch: Colors.blue,
        textSelectionColor: Colors.red
      ),
      home: new SampleAppPage(),
    );
  }
}
```

```

}
}

```

数据库和本地存储

如何在Flutter中访问Shared Preferences ?

在Android中，您可以使用SharedPreferences API存储一些键值对

在Flutter中，您可以通过使用插件[Shared_Preferences](#)来访问此功能

这个插件包装了Shared Preferences和NSUserDefaults（与iOS相同）的功能

```

import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';

void main() {
  runApp(
    new MaterialApp(
      home: new Scaffold(
        body: new Center(
          child: new RaisedButton(
            onPressed: _incrementCounter,
            child: new Text('Increment Counter'),
          ),
        ),
      ),
    ),
  );
}

_incrementCounter() async {
  SharedPreferences prefs = await SharedPreferences.getInstance();
  int counter = (prefs.getInt('counter') ?? 0) + 1;
  print('Pressed $counter times. ');
  prefs.setInt('counter', counter);
}

```

如何在Flutter中访问SQLite

在Android中，您可以使用SQLite存储，通过SQL查询的结构化数据。

在Flutter中，您可以使用[SQFlite](#)插件来访问SQFlite此功能

通知

如何设置推送通知

在Android中，您可以使用Firebase云消息传递为您的应用设置推送通知。

在Flutter中，您可以使用[Firebase_Messaging](#)插件访问此功能

注意：在中国无法使用Firebase服务。

