目录

Flutter 进阶学习笔记

第一章 为什么 Flutter 是跨平台开发的终极之选

跨平台开发是当下最受欢迎、应用最广泛的框架之一。能实现跨平台开发的框架也五花八门，让人眼花缭乱。

最流行的跨平台框架有 Xamarin、PhoneGap、Ionic、Titanium、Monaca、Sencha、jQuery Mobile、React native、Flutter 等等。但这些工具的表现也是高低有别，各有千秋。

在这些流行的框架中，有很多也已经消失在了历史的长河中被人渐渐遗忘了。但 React native 和 Flutter 这俩框架地位依旧坚挺，备受欢迎。

01 这是为什么？

因为它们俩分别由最强大的科技巨头 Facebook 和谷歌背书支持。本文将讨论谷歌 Flutter 这个万千瞩目的框架。

你想知道什么是 Flutter 应用开发吗？你是否经常查询这些问题：Flutter 在 iOS 开发环境中好用吗？它比 React native 更好吗？

本文会告诉你，为什么 Flutter 是一个值得信赖的跨平台应用开发解决方案。下面就跟我一起探究答案，深入了解这一跨平台开发最佳工具吧。

02 跨平台开发

新手可能会问这个问题：什么是跨平台开发呢？

本质上来说，跨平台开发就是"一石二鸟"的开发技术。下面简单解释一下。早期没有跨平台框架的时候，开发者必须为同一应用的各个平台（比如 Android、iOS、Windows 等）分别编写代码。这对开发者以及投资开发该应用的企业而言都是费时费力又花钱的工作。

那么跨平台框架解决了什么问题呢？就是用跨平台框架可以只用一份代码就适配所有平台，省钱又省时。

下面回到主题，谈谈为什么 Flutter 能用来开发最优秀的移动应用，为什么它是跨平台开发的首选。

03 什么是 Flutter

简而言之，Flutter 是一个软件开发工具包（SDK）。它包含众多小部件、框架和工具，能帮助开发者无缝构建跨平台应用。

介绍 Flutter 的功能之前，我们先来看看它的优势和不足。

Flutter 的优势：

它完全免费，彻底开源
可以用来更快地创建应用

出色的用户界面（UI）

节省代码量
可接入平台原生功能

最适合 MVP 开发（最小化可行产品）

较老的设备也使用相同 UI 运行应用

减少测试工作量
更丰富的社区支持

较低的维护难度

内置来自 Dart 的包管理器

Flutter 的不足：

Flutter 仅适用于移动设备平台，浏览器不支持 Flutter（最新的 Flutter 1.5 提供了 Flutter for Web，开始解决这个问题）。

Flutter 框架诞生不久，可能欠缺很多功能。

Flutter 不支持开发 Apple TV 或 Android TV 上的应用。

相比 JS/TS，Flutter 可选的包较少。

04 Flutter 的特性

谷歌现已发布 Flutter 的最新重大更新版本，Flutter 1.2 版本。Flutter 新版主要的改进包括：

为开发者提供跨平台应用开发的最前沿工具。
新版为原有的小部件增加了许多新功能。
新版还增强了核心框架的稳定性、质量和性能。

1. 原生 ARM 代码

Flutter 有一个名为原生 ARM 的功能，对初创企业和科技公司而言很有意义。它可以帮助开发者更轻松地实现自己的想法，为应用项目带来最显著的优势。

2. Web 视图组件

这一功能使用户可以轻松地在移动应用中查看 Web 内容。此外，Flutter 还让应用中的页面跳转和稳定更加容易。

3. Dart 2.2

最近更新的 Dart.2.2 确实是一个变革性的角色。它提升了 AOT 编译代码的性能。此外，dart 库提供了很多用来建立映射、列表和对象集合的类。

Dart2.2 的其他功能包括：

映射是键值对的集合。
Flutter 列表是有序的值序列，其中每个值可以通过索引访问，并且可以多次出现。
它更新了所有 dart 语言规范以适配改动。

开发者可以使用 Dart 通用前端（CFE）构建新的语言功能。

4. 应用内购买

当用户在 App store 中启动应用内购买时，这些功能可以让你的应用正常完成交易。

Google Play 和 App Store 的开发者都需要对带有应用内购买项目的应用做好配置，正常调用它们的应用内购买 API。

5. Android 应用包

Flutter 支持 Android 应用包，这是一种新的上传格式，包含应用程序的所有编译代码和资源。这种格式可以加快 APK 的打包和向 Google Play 发布的流程。

6. 无需手动管理多个 APK

这些功能让用户可以下载更小、更优化的 APK。开发者也不需要为了支持多种设备而构建、发布和管理多个 APK 了。

7. 减小 APK 大小

Android 应用包使用的 APK 拆分机制可以缩减应用的大小，并支持 Android 应用程序的动态交付等新功能。

8. 动态功能模块

此功能允许开发者将某些功能和资源与应用程序的基础模块分离开来，并将前者添加到应用程序包中。

例如，如果你的应用包含相机功能，则可以将其设为动态模块。之后当用户想要下载并安装这个功能时就可以按需操作了。

05 Flutter 构建应用的工具

Flutter 框架支持很多工具，例如 Android Studio 和 visual studio code。还有的工具允许用户从命令行和 Dart DevTools 构建应用以进行调试。

此外，它还允许开发者查看日志、调试应用，并检查 Flutter 应用开发的小部件。

以下是最适合移动应用开发的 Flutter 工具。

时间线视图：它可以帮助你逐帧监控应用，观察应用的呈现和计算工作。

小部件检查器：此工具支持可视化和浏览 Flutter 小部件树层级结构。

日志视图：它显示来自应用程序、网络、框架和垃圾回收事件的活动日志。

源代码级调试器：用户可以用它一步步执行代码、标记断点并检查调用堆栈。

06 使用 Flutter 构建的热门应用

了解过 Flutter 的最新功能之后，我们来看看哪些初创公司和知名品牌使用 Flutter 进行跨平台开发。下面是 Flutter 的 应用案例。

阿里巴巴（电子商务）：这家电子商务巨头无人不知无人不晓。阿里巴巴在淘宝中就用了谷歌 Flutter 开发。

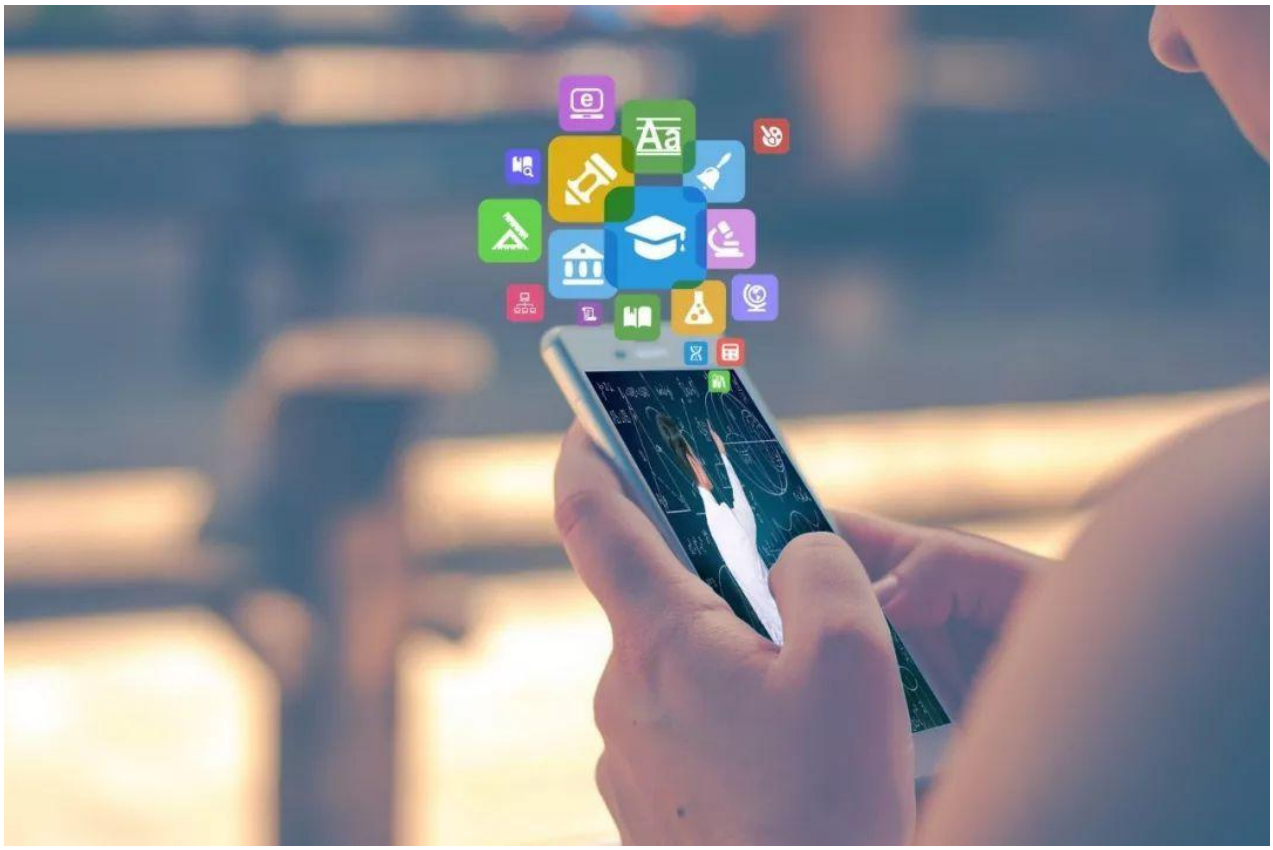
谷歌广告（实用程序）：这是付费营销的终极工具。这个跨平台的应用就是用 Flutter 制作的，可以用来监控企业的网络广告投放。此外，它还允许谷歌专家添加 / 修改 / 删除关键字，改进你的广告计划。

Birch Finance（金融）：Birch Finance 是一个信用卡积分兑换应用，可以帮助用户管理并优化自己的信用卡。用户可以用它一站式管理所有信用卡账户，它还提供了多种赚取和兑换奖励的途径。

腾讯（游戏等应用）：这家中国科技巨头也使用 Flutter 开发即时通讯软件服务和游戏，诸如绝地求生、QQ 音乐、电商应用等。

Watermaniac（健康与健身）：医疗保健行业也在开发跨平台应用程序。

Watermaniac 已决定使用 Flutter 构建其应用。该应用能帮助用户监控他们摄取的水量。



07 构建 Flutter 应用的成本

构建应用当然需要花费时间和资源。项目花费的时间越久，需要投入的各种资源也就越多。因此开发应用所需的成本主要取决于时间和资源这两个因素。

如果你正在考虑使用 Flutter 开发应用，以下是应用开发的成本计算。

成本计算器：

总成本 = SDLC 流程所需的小时数 * 每小时所需资源的费用

SDLC 流程包括构建、测试、部署、更改和维护应用程序的详细计划。它包括 UI / UX 设计、前端与后端开发、质量保证和生产发布。

最重要的一点在于，假设一个具备基础功能的小型应用的开发投入在 10,000 美元到 50,000 美元的水平上，那么使用 Flutter 之后成本能减少一半。

08 结论

总的来说，跨平台开发对初创公司和业务都有很大的好处。而 Flutter 作为新诞生的开发框架有着明显的优势和好处。

如果你想快速构建跨平台应用，Flutter 非常适合你。无论是要创建最小可行产品（MVP）还是成熟的企业应用程序，Flutter 都是最佳解决方案。

第二章 在 Windows 上搭建 Flutter 开发环境

01 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。

02 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。

03 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 instructi
ons.
```

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。

04 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

05Android 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.

起步：配置编辑器

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

1、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.

- VS Code

3、Visual Studio Code (VS Code) 安装

VS Code: 轻量级编辑器，支持 Flutter 运行和调试.

安装 VS Code

- VS Code，安装 1.20.1 或更高版本.

安装 Flutter 插件

1. 启动 VS Code
2. 调用 **View>Command Palette…**
3. 输入 'install'，然后选择 **Extensions: Install Extension** action
4. 在搜索框输入 `flutter`，在搜索结果列表中选择 'Flutter'，然后点击 **Install**
5. 选择 'OK' 重新启动 VS Code

4、通过 Flutter Doctor 验证您的设置

1. 调用 **View>Command Palette…**
2. 输入 'doctor'，然后选择 **'Flutter: Run Flutter Doctor'** action
3. 查看"OUTPUT"窗口中的输出是否有问题

5、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".

起步：体验

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

Android Studio: 为 Flutter 提供完整的 IDE 体验.

1. 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`.

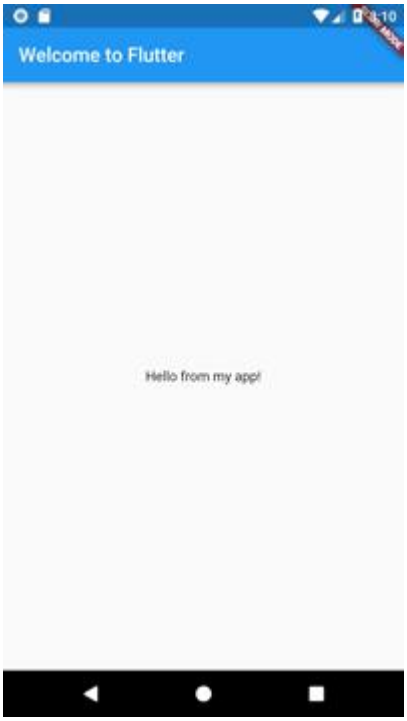
2. 运行应用程序

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会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。

- VS Code

VS Code: 轻量级编辑器，支持 Flutter 运行和调试.

1. 创建新的应用

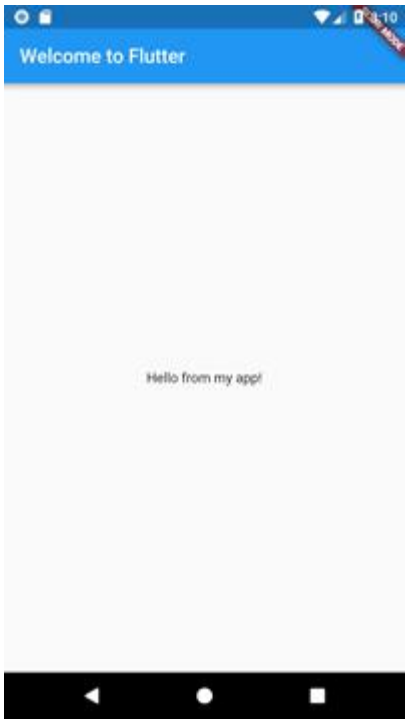
1. 启动 VS Code
2. 调用 **View>Command Palette…**
3. 输入 ‘flutter’，然后选择 **‘Flutter: New Project’** action
4. 输入 Project 名称 (如 `myapp`)，然后按回车键
5. 指定放置项目的位置，然后按蓝色的确定按钮
6. 等待项目创建继续，并显示 main.dart 文件

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 组件的简单的演示应用程序。

在项目目录中，您的应用程序的代码位于 `lib/main.dart`.

2. 运行应用程序

1. 确保在 VS Code 的右下角选择了目标设备
2. 按 F5 键或调用 **Debug>Start Debugging**
3. 等待应用程序启动
4. 如果一切正常，在应用程序建成功后，您应该在您的设备或模拟器上看到应用程序:



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. 用你喜欢的编辑器打开文件 `lib/main.dart`

2. 将字符串 `'You have pushed the button this many times:'` 更改为 `'You have clicked the button this many times:'`

3. 不要按“停止”按钮；让您的应用继续运行.

4. 要查看您的更改，请调用 **Save** (`cmd-s` / `ctrl-s`)，或者点击 **热重载按钮** (绿色圆形箭头按钮).

你会立即在运行的应用程序中看到更新的字符串

Terminal + 编辑器: 您的编辑选择与 Flutter 的终端工具结合运行和构建

创建新的应用

1. 使用 `flutter create` 命令创建一个 project:

```
$ flutter create myapp$ cd myapp
```

上述命令创建一个 Flutte 让项目，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，您的应用程序的代码位于 `lib/main.dart`.

运行应用程序

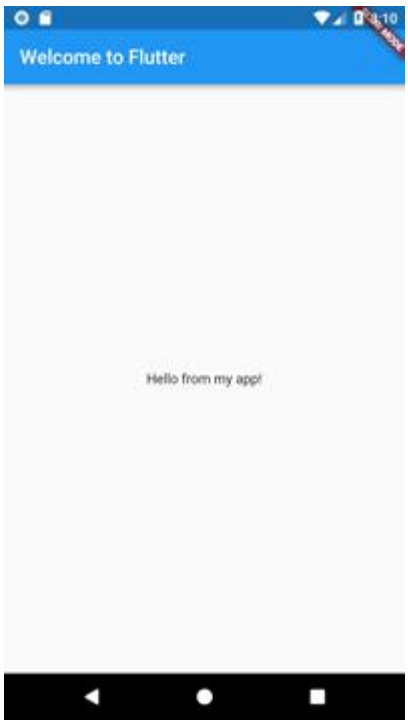
- 检查 Android 设备是否在运行。如果没有显示，请参照 设置.

```
$ flutter devices
```

- 运行 `flutter run` 命令来运行应用程序:

```
$ flutter run
```

- 如果一切正常，在应用程序建成功后，您应该在您的设备或模拟器上看到应用程序:



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. 打开文件 `lib/main.dart`
2. 将字符串 `'You have pushed the button this many times:'` 更改为 `'You have clicked the button this many times:'`
3. 不要按“停止”按钮; 让您的应用继续运行.
4. 要查看您的更改，请调用 **Save** (`cmd-s` / `ctrl-s`), 或者点击 **热重载按钮** (带有闪电图标的按钮).

你会立即在运行的应用程序中看到更新的字符串

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。

第三章 编写您的第一个 Flutter App

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。

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.

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

- 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
你必须为您的 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**.

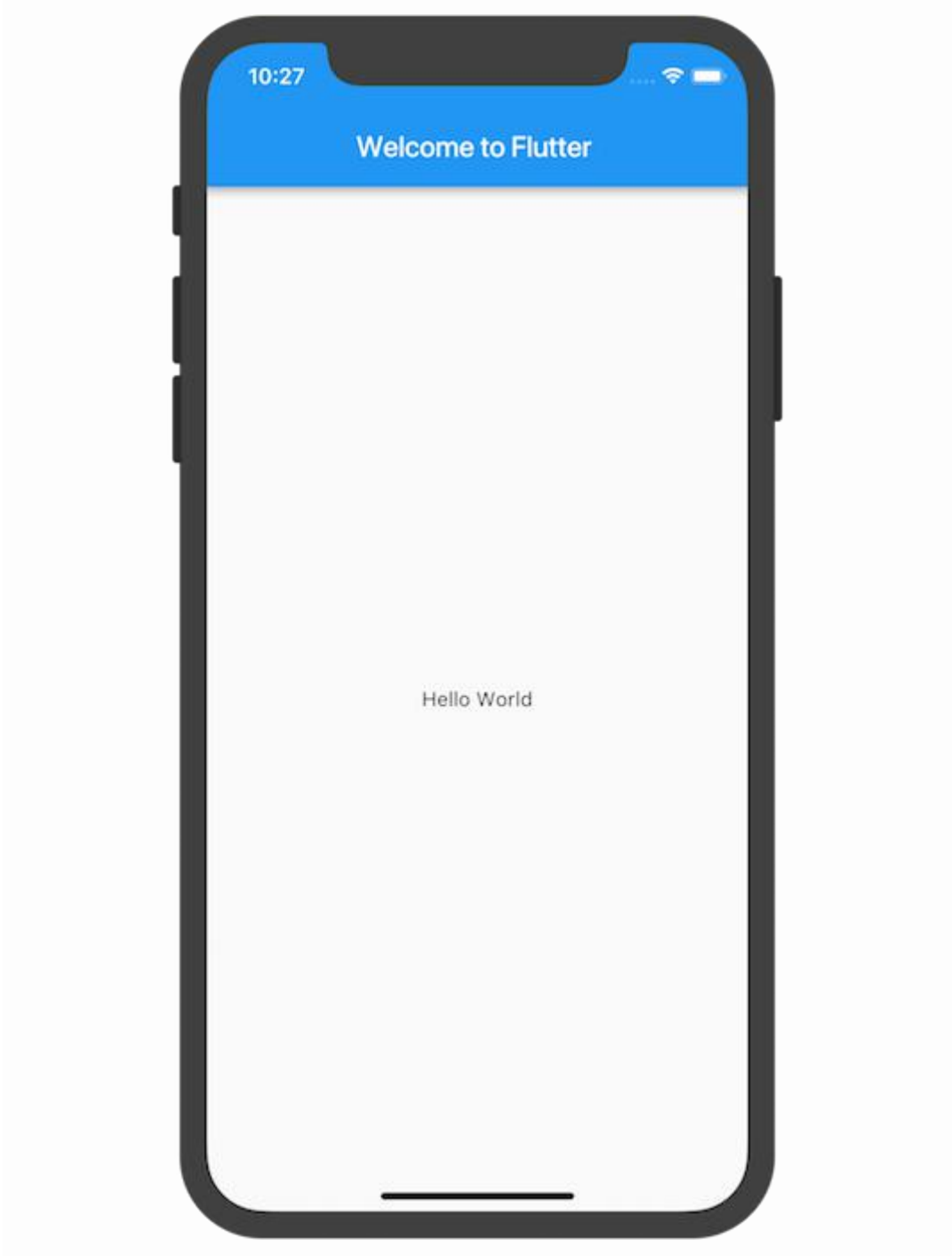
VS Code: 右键单击并选择 **Format Document**.

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';
voidmain()=>runApp(newMyApp());
classMyAppextendsStatelessWidget{
@override
Widgetbuild(BuildContextcontext){
returnnewMaterialApp(
title:'Welcome to Flutter',
home:newScaffold(
appBar:newAppBar(
title:newText('Welcome to Flutter'),
),
body:newCenter(
child:newText('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个 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';

voidmain()=>runApp(newMyApp());

classMyAppextendsStatelessWidget{

@override

  Widget build(BuildContext context){

final wordPair =new WordPair.random();

returnnewMaterialApp(

     title:'Welcome to Flutter',

     home:newScaffold(

       appBar:newAppBar(

         title:newText('Welcome to Flutter'),

),

       body:newCenter(

//child: new Text('Hello World'),

         child:newText(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

```
classRandomWordsextendsStatefulWidget{

@override

createState()=>newRandomWordsState();}
```

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

```
classRandomWordsStateextendsState<RandomWords>{
@override
Widget build(BuildContext context){
final wordPair =new WordPair.random();
returnnew 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

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

returnnewMaterialApp(
     title:'Welcome to Flutter',
     home:newScaffold(
       appBar:newAppBar(
         title:newText('Welcome to Flutter'),
),
       body:newCenter(
//child: new Text(wordPair.asPascalCase),
         child:newRandomWords(),
```

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

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

```
classRandomWordsStateextendsState<RandomWords>{

final _suggestions =<WordPair>[];


final _biggerFont =constTextStyle(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:

```
classRandomWordsStateextendsState<RandomWords>{

...

Widget _buildSuggestions(){

returnnew 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)returnnew 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 ：

```
classRandomWordsStateextendsState<RandomWords>{
...

Widget _buildRow(WordPair pair){
returnnew 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：

```
classRandomWordsStateextendsState<RandomWords>{
```

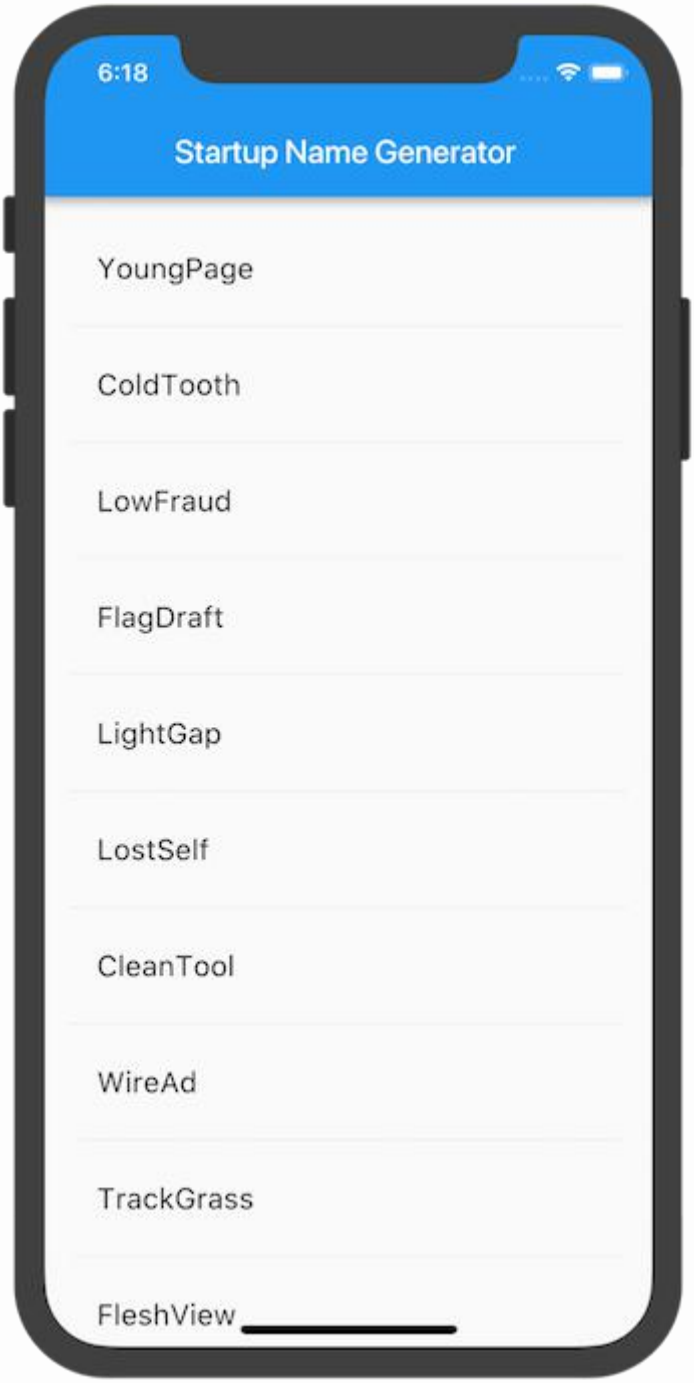
```
...
@override
  Widget build(BuildContext context){
final wordPair =new WordPair.random();// 删除这两行
returnnew Text(wordPair.asPascalCase);
returnnew Scaffold (
appBar:new AppBar(
title:new Text('Startup Name Generator'),
),
body:_buildSuggestions(),
);
}
...}
```

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的路由名称。

用下面高亮部分替换最初的 build 方法：

```
classMyAppextendsStatelessWidget{
@override
Widget build(BuildContext context){
returnnew 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出“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。

```
classRandomWordsStateextendsState<RandomWords>{
final _suggestions =<WordPair>[];

final _saved =new Set<WordPair>();

final _biggerFont =constTextStyle(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);
returnnewListTile(
    title:newText(
      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。

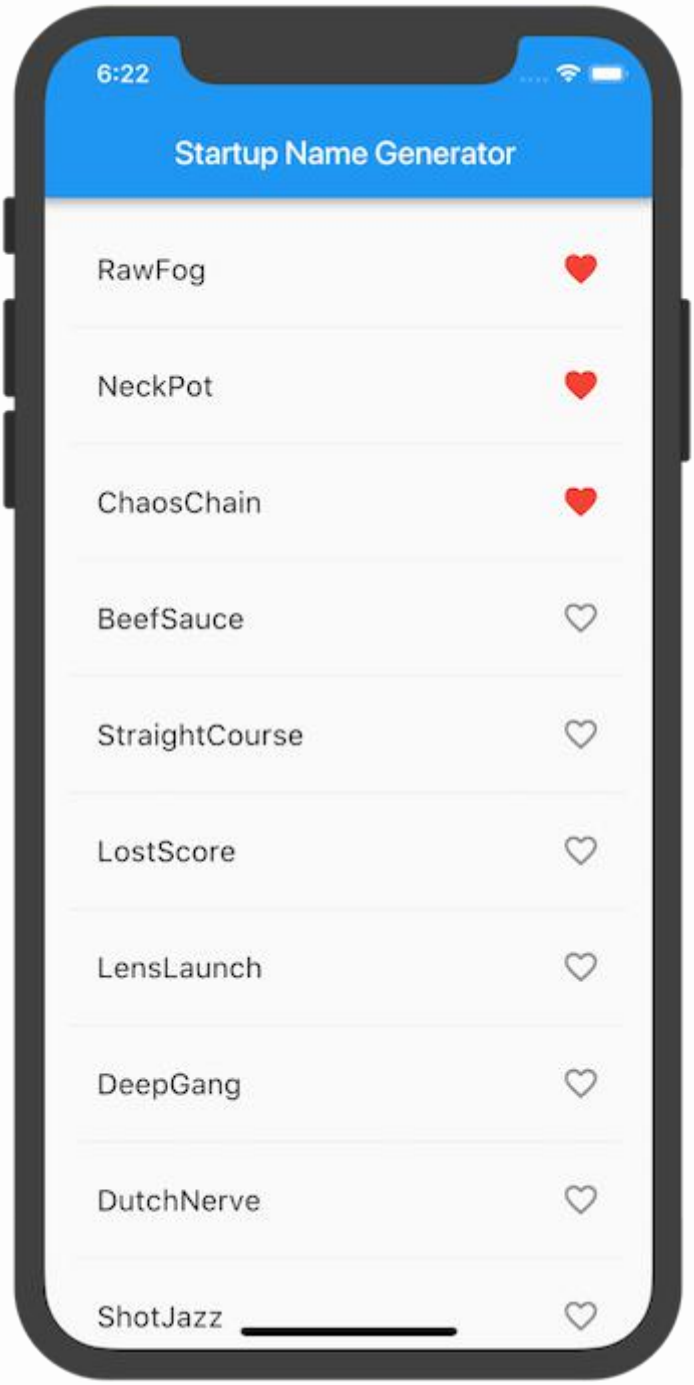
在 `_buildRow` 中让心形❤ 图标变得可以点击。如果单词条目已经添加到收藏夹中，再次点击它将其从收藏夹中删除。当心形❤ 图标被点击时，函数调用 `setState()` 通知框架状态已经改变。

添加如下高亮的行:

```
Widget _buildRow(WordPair pair){
final alreadySaved = _saved.contains(pair);
returnnewListTile(
    title:newText(
      pair.asPascalCase,
      style: _biggerFont,
),
    trailing:newIcon(
      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移除。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：

```
classRandomWordsStateextendsState<RandomWords>{

...

@override

  Widget build(BuildContext context){

returnnewScaffold(

     appBar:newAppBar(

       title:newText('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.

```
classRandomWordsStateextendsState<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){

returnnew ListTile(

title:new Text(

pair.asPascalCase,

style: _biggerFont,

),

);

},

);

final divided = ListTile

.divideTiles(
```

```
context: context,
tiles: tiles,
)
.toList();
},
),
);}
```

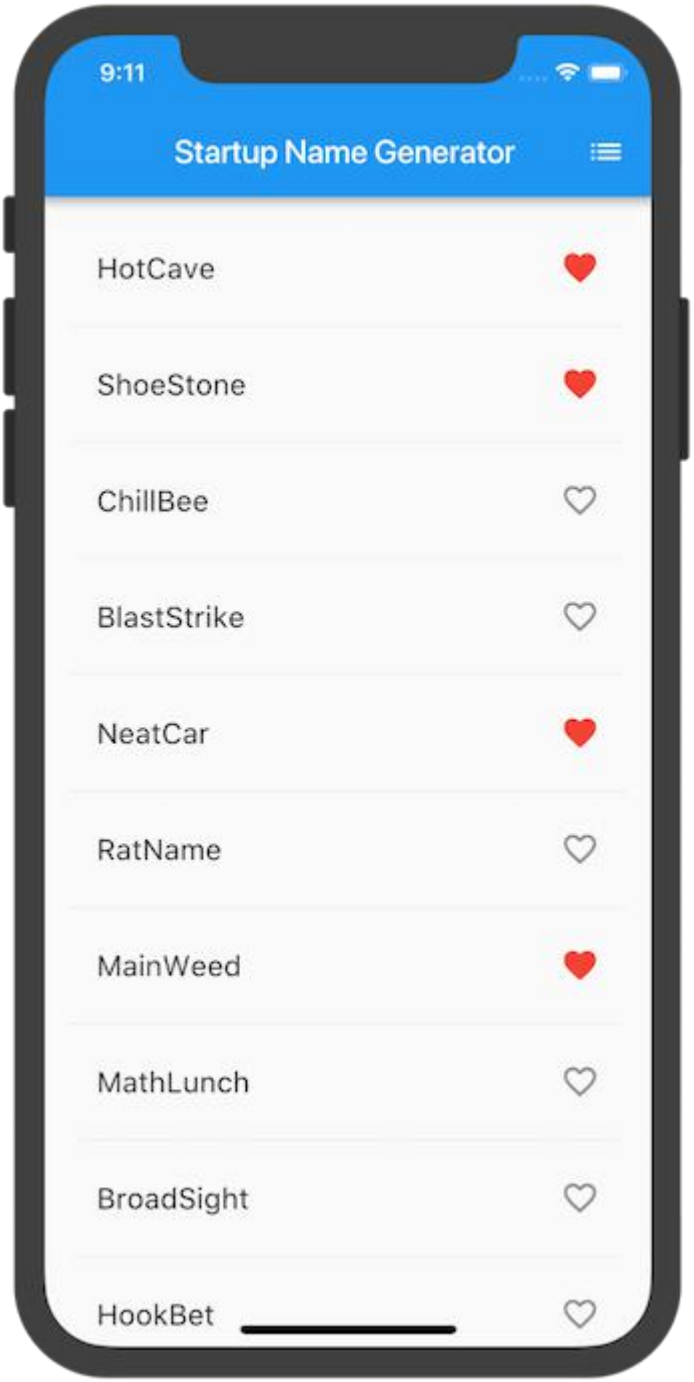
5.builder 返回一个 Scaffold，其中包含名为“Saved Suggestions”的新路由的应用栏。新路由的 body 由包含 ListTiles 行的 ListView 组成；每行之间通过一个分隔线分隔。

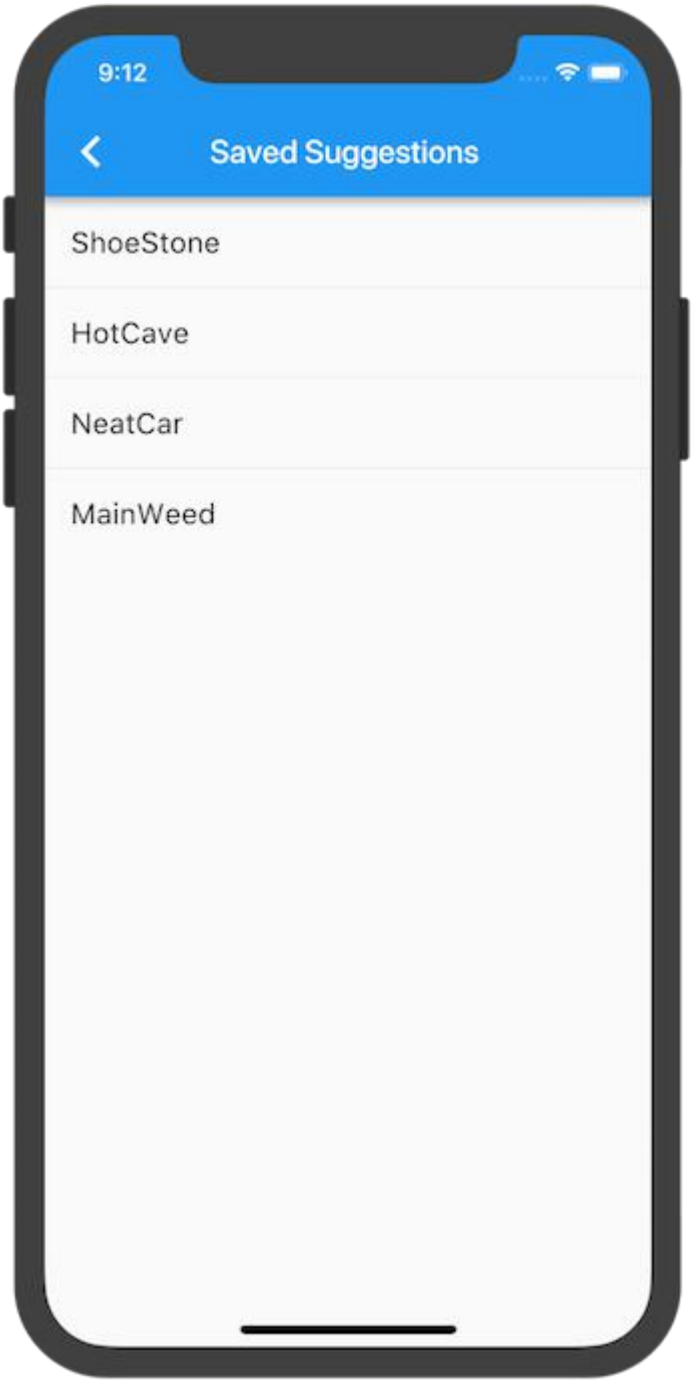
添加如下高亮的代码:

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

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

5. 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。

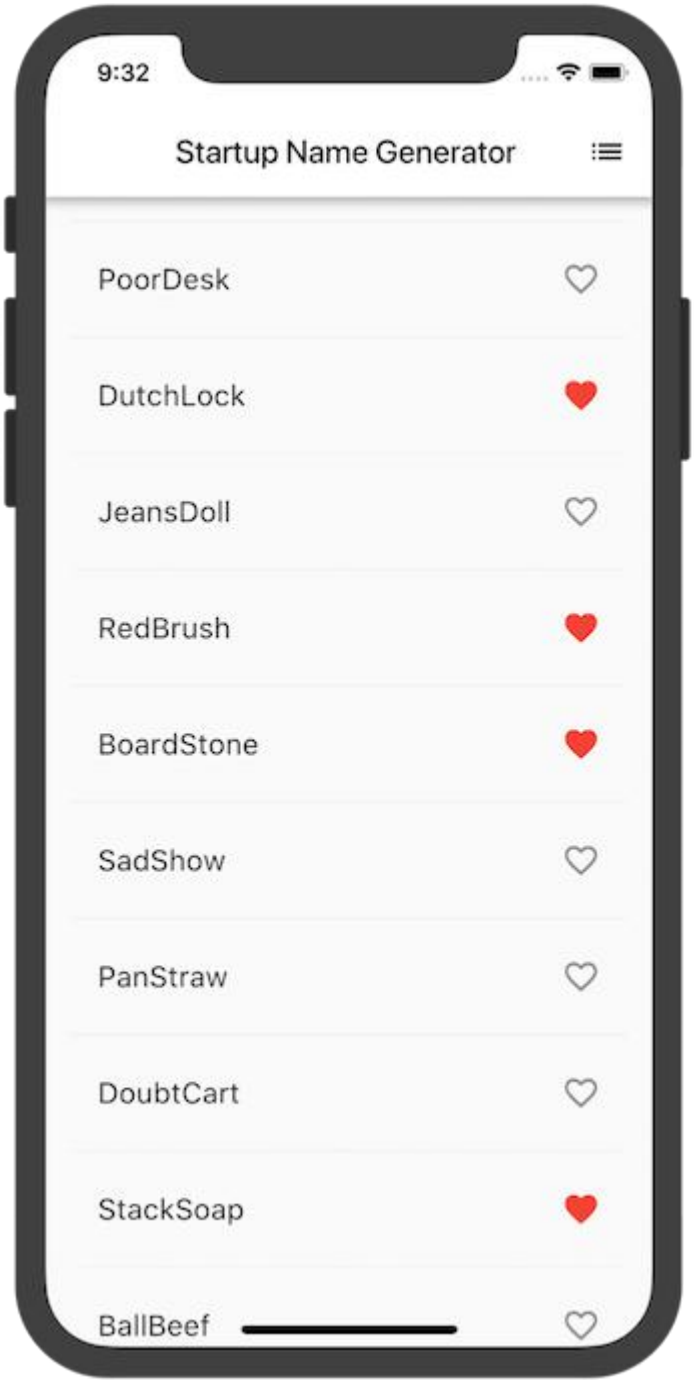
1.您可以通过配置 ThemeData 类轻松更改应用程序的主题。 您的应用程序目前使用默认主题，下面将更改 primary color 颜色为白色。

通过如下高亮部分代码，将应用程序的主题更改为白色：

```
classMyAppextendsStatelessWidget{
@override
  Widget build(BuildContext context){
returnnewMaterialApp(
     title:'Startup Name Generator',
theme:new ThemeData(
primaryColor: Colors.white,
),
     home:newRandomWords(),
);
}}
```

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**

第四章 Flutter 开发环境搭建和调试

1.开发环境的搭建

1. 下载 Flutter SDK
2. 配置环境变量
3. 安装 Visual Studio Code 所需插件
4. 创建 Flutter 项目

2.模拟器的安装与调试

Flutter 开发工具很多，有很多支持 Flutter 开发的 IDE。比如 Android Studio、Visual Studio Code、InteIIiJ IDEA、Atom、Komodo 等。这里将使用 Visual Studio Code 作为主要开发工具，因为 Visual Studio Code 占用内存和 CPU 比较低，非常的流畅，体验也比较的好。模拟器的话，这里推荐使用 Android 官方的模拟器，也就是 Android Studio SDK 里带的模拟器。不过，这里的模拟器我们使用单独启动的，无需从 Android Studio 启动，当然也可以用真机运行调试。接下来，我们就开始 Flutter 开发环境的搭建吧。注意：本文是在 Windows 环境下安装的开发环境。本文将主要介绍：

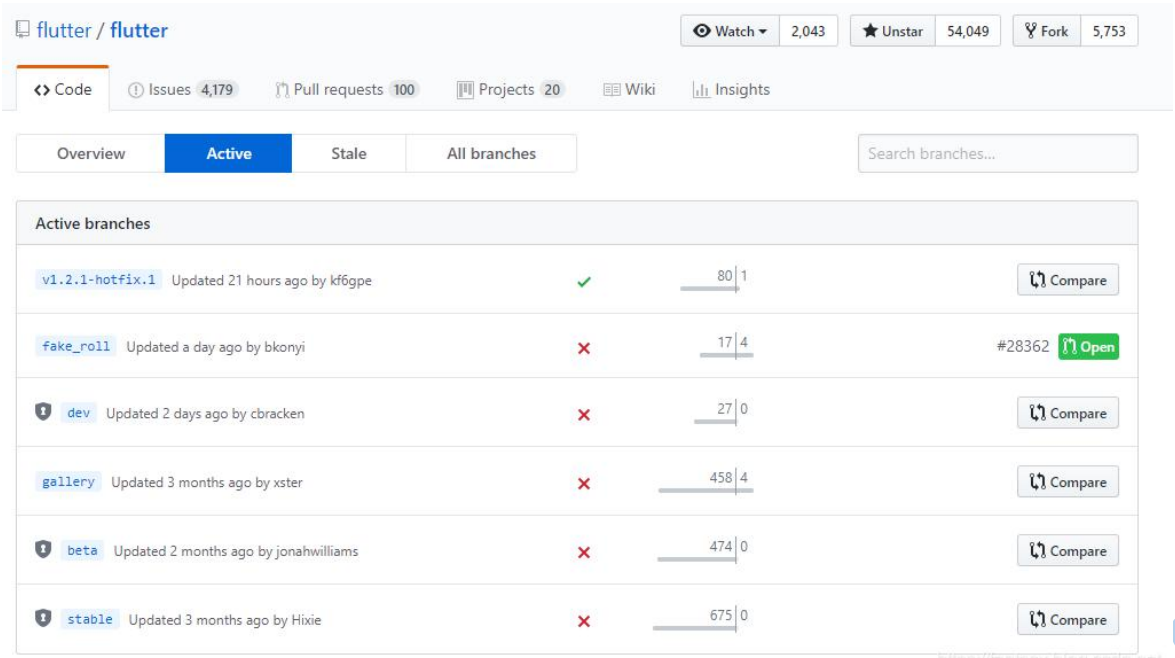
- Flutter 下载与环境变量配置
- Visual Studio Code 插件安装与新建 Flutter 项目
- 模拟器的安装
- 运行 Flutter 项目到模拟器和真机
- Flutter 常用命令

3..开发环境的搭建

1. 下载 Flutter SDK

Flutter SDK 由两部分构成，一个是 Dart SDK，另一个就是 Flutter SDK，因为 Flutter 是基于 Dart 的。可以通过两种方式下载：一种是 Git 下载；另一种是直接下载 SDK 压缩包即可。

Git 方式我们可以通过拉取官方 Github 上的 flutter 分支来下载。分支分类如下图：



可以看到主要有 dev、beta 和 stable 三个官方分支使，这里正式开发的话可以下载 stable 稳定版本。用 Git 命令进行下载 stable 分支：

```
git clone-b stable https://github.com/flutter/flutter.git
```

另一种是直接官网下载 SDK 压缩包，官方下载地址为：

https://storage.googleapis.com/flutter_infra/releases/stable/windows/flutter_windows_v1.0.0-stable.zip

2. 配置环境变量

下载完 SDK 后我们可以把它解压放到指定文件夹里，接下来就是配置 SDK 环境变量量，这样我们就可以在需要的目录执行相关命令了。如果在官网更新下载 SDK 慢的话，可以设置国内的镜像代理地址，这样下载会快一些。可以将如下的国内下载镜像地址加入到环境变量中：

```
变量名：PUB_HOSTED_URL，变量值：https://pub.flutter-io.cn

变量名：FLUTTER_STORAGE_BASE_URL，变量值：https://storage.flutter-io.cn
```

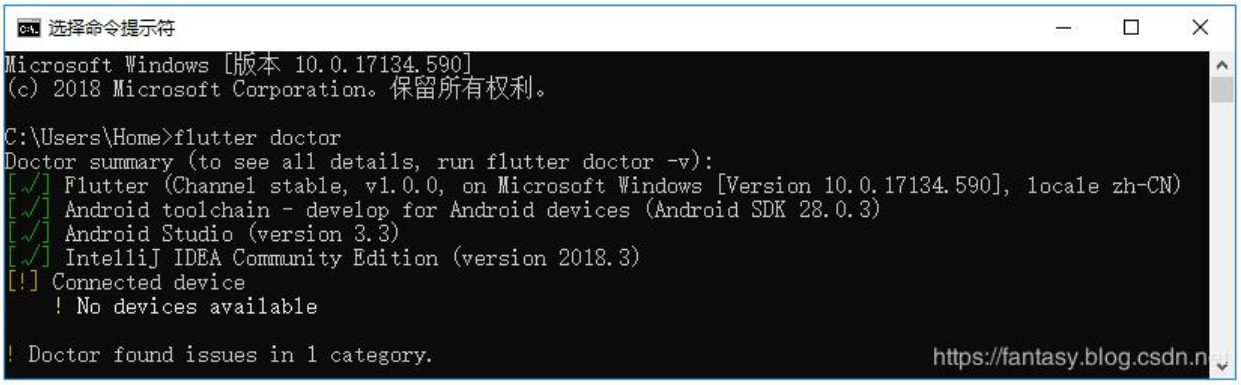
Flutter SDK 环境变量，讲 flutter 的 bin 目录加入环境变量即可：

```
[你的 Flutter 文件夹路径]\flutter\bin
```

这样我们的 Flutter SDK 的环境变量就配置完毕了。接下来在命令提示符窗口中输入命令：

```
flutter doctor
```

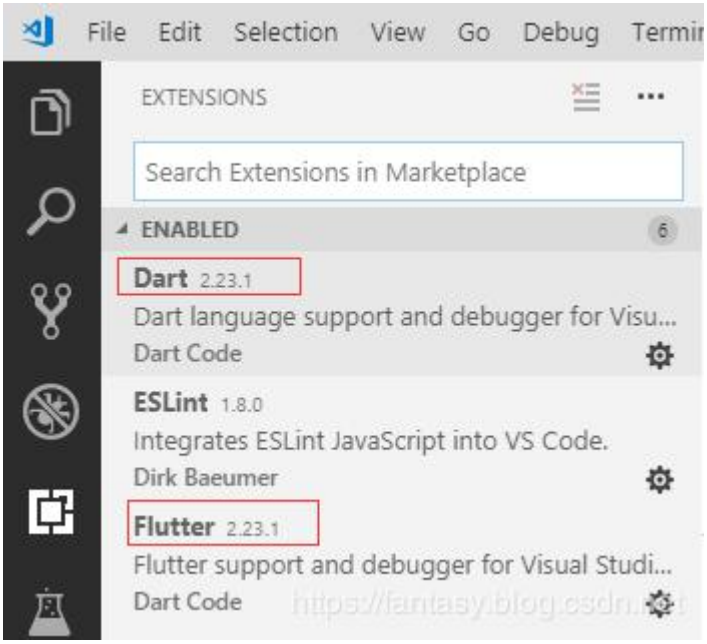
它可以帮助我们检查 Flutter 环境变量是否设置成功，Android SDK 是否下载以及配置好环境变量等等。如果有相关的错误提示，根据提示进行修复和安装、设置即可。每次运行这个命令，都会帮你检查是否缺失了必要的依赖。通过运行 flutter doctor 命令来验证你是否已经正确地设置了，并且可以自动更新和下载相关的依赖。如果全部配置正确的话，会出现如下类似的检测信息：



主要检测信息为：Flutter、Android toolchain、Connected device。

3. 安装 Visual Studio Code 所需插件

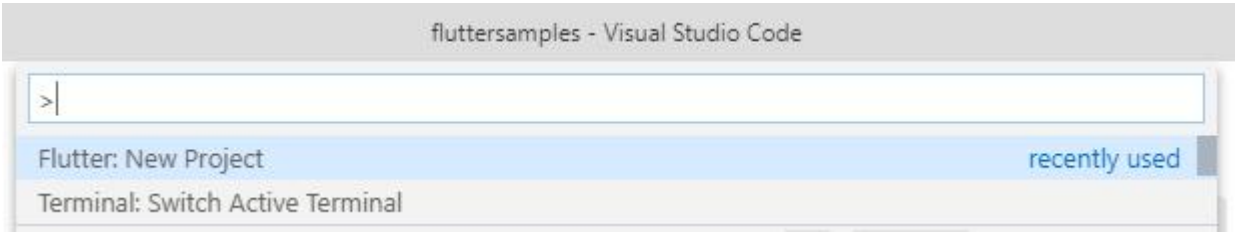
在 Visual Studio Code 的 Extensions 里搜索安装 Dart 和 Flutter 扩展插件：



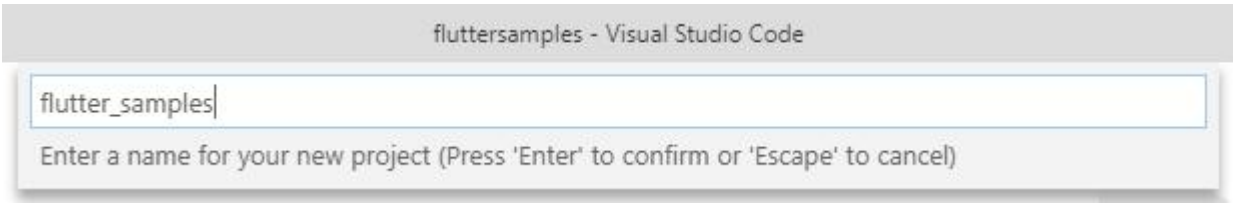
安装完成插件后，重启 Visual Studio Code 编辑器即可。

4. 创建 Flutter 项目

接下来进行 Flutter 项目的新建，我们可以通过命令面板或者快捷键 Ctrl+Shif+P 打开命令面板，找到 Flutter：New Project：



点击 New Project，接下来进入项目名称输入：



回车，然后选择好项目的存储位置即可，这样就完成了 Flutter 项目的新建。整个的创建流程日志如下：

```
[undefined] flutter create .

Waiting for another flutter command to release the startup lock...

Creating project ....

  .gitignore (created)

  .idea\libraries\Dart_SDK.xml (created)

  .idea\libraries\Flutter_for_Android.xml (created)

  .idea\libraries\KotlinJavaRuntime.xml (created)

  .idea\modules.xml (created)
```

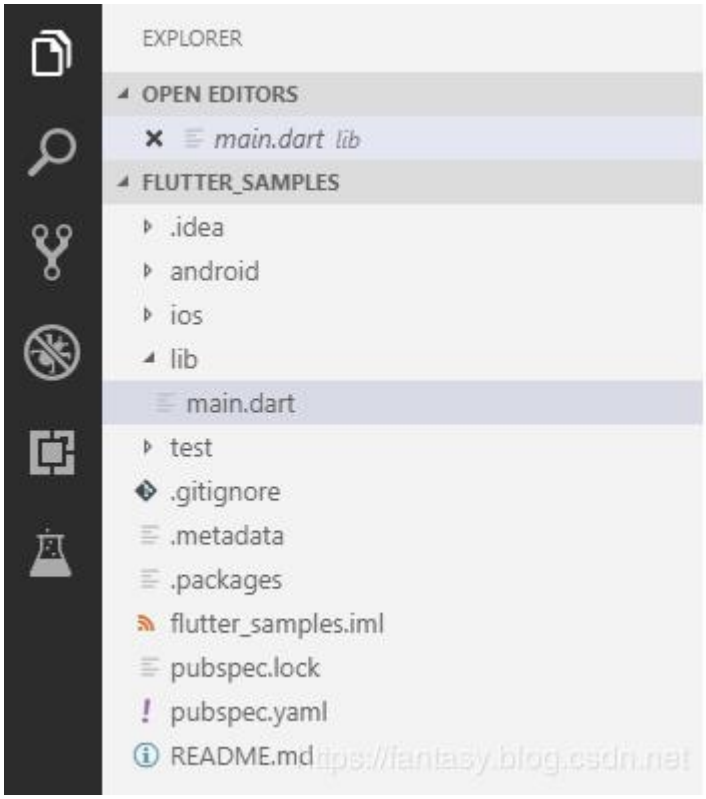
```
 .idea\runConfigurations\main_dart.xml (created)

 .idea\workspace.xml (created)

 .metadata (created)

 android\app\build.gradle (created)

 android\app\src\main\java\com\example\fluttersamples\MainActivity.java
(created)

 android\build.gradle (created)

 android\flutter_samples_android.iml (created)

 android\app\src\main\AndroidManifest.xml (created)

 android\app\src\main\res\drawable\launch_background.xml (created)

 android\app\src\main\res\mipmap-hdpi\ic_launcher.png (created)

 android\app\src\main\res\mipmap-mdpi\ic_launcher.png (created)

 android\app\src\main\res\mipmap-xhdpi\ic_launcher.png (created)

 android\app\src\main\res\mipmap-xxhdpi\ic_launcher.png (created)

 android\app\src\main\res\mipmap-xxxhdpi\ic_launcher.png (created)

 android\app\src\main\res\values\styles.xml (created)

 android\gradle\wrapper\gradle-wrapper.properties (created)

 android\gradle.properties (created)

 android\settings.gradle (created)

 ios\Runner\AppDelegate.h (created)

 ios\Runner\AppDelegate.m (created)

 ios\Runner\main.m (created)

 ios\Runner.xcodeproj\project.pbxproj (created)
```

```
  ios\Runner.xcodeproj\xcshareddata\xcschemes\Runner.xcscheme (created)

  ios\Flutter\AppFrameworkInfo.plist (created)

  ios\Flutter\Debug.xcconfig (created)

  ios\Flutter\Release.xcconfig (created)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Contents.json (created)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-1024x1024@1x.pn
g (created)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-20x20@1x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-20x20@2x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-20x20@3x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-29x29@1x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-29x29@2x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-29x29@3x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-40x40@1x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-40x40@2x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-40x40@3x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-60x60@2x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-60x60@3x.png (c
reated)
```

```
  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-76x76@1x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-76x76@2x.png (c
reated)

  ios\Runner\Assets.xcassets\AppIcon.appiconset\Icon-App-83.5x83.5@2x.pn
g (created)

  ios\Runner\Assets.xcassets\LaunchImage.imageset\Contents.json (create
d)

  ios\Runner\Assets.xcassets\LaunchImage.imageset\LaunchImage.png (creat
ed)

  ios\Runner\Assets.xcassets\LaunchImage.imageset\LaunchImage@2x.png (cr
eated)

  ios\Runner\Assets.xcassets\LaunchImage.imageset\LaunchImage@3x.png (cr
eated)

  ios\Runner\Assets.xcassets\LaunchImage.imageset\README.md (created)

  ios\Runner\Base.lproj\LaunchScreen.storyboard (created)

  ios\Runner\Base.lproj\Main.storyboard (created)

  ios\Runner\Info.plist (created)

  ios\Runner.xcodeproj\project.xcworkspace\contents.xcworkspacedata (cre
ated)

  ios\Runner.xcworkspace\contents.xcworkspacedata (created)

  lib\main.dart (created)

  flutter_samples.iml (created)

  pubspec.yaml (created)

  README.md (created)

  test\widget_test.dart (created)

Running "flutter packages get" in flutter_samples...           11.8s
```

```
Wrote 64 files.



All done!

[√] Flutter is fully installed. (Channel stable, v1.0.0, on Microsoft Wi
ndows [Version 10.0.17134.590], locale zh-CN)

[√] Android toolchain - develop for Android devices is fully installed.
(Android SDK 28.0.3)

[√] Android Studio is fully installed. (version 3.3)

[√] IntelliJ IDEA Community Edition is fully installed. (version 2018.3)

[!] Connected device is not available.



Run "flutter doctor" for information about installing additional compone
nts.



In order to run your application,type:



  $ cd .

  $ flutter run



Your application code is in .\lib\main.dart.



exit code 0
```

Flutter 项目结构如下：



其中，Android 相关的修改和配置在 android 目录下，结构和 Android 应用项目结构一样；IOS 相关修改和配置在 ios 目录下，结构和 IOS 应用项目结构一样。

最重要的 flutter 代码文件是在 lib 目录下，类文件以.dart 结尾，语法结构为 Dart 语法结构。大致如下：

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

voidmain()=>runApp(MyApp());

classMyAppextendsStatelessWidget{

// This widget is the root of your application.

@override

Widgetbuild(BuildContext context){

returnMaterialApp(

     title:'Flutter Demo',
```

```
      theme:ThemeData(

// This is the theme of your application.

//

// Try running your application with "flutter run". You'll see the

// application has a blue toolbar. Then, without quitting the app, try

// changing the primarySwatch below to Colors.green and then invoke

// "hot reload" (press "r" in the console where you ran "flutter run",

// or simply save your changes to "hot reload" in a Flutter IDE).

// Notice that the counter didn't reset back to zero; the application

// is not restarted.

        primarySwatch:Colors.blue,

),

      home:MyHomePage(title:'Flutter Demo Home Page'),

);

}}

classMyHomePageextendsStatefulWidget{

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



// This widget is the home page of your application. It is stateful, mean

ing

// that it has a State object (defined below) that contains fields that a

ffect

// how it looks.
```

```
// This class is the configuration for the state. It holds the values (in
 this

// case the title) provided by the parent (in this case the App widget) a
nd

// used by the build method of the State. Fields in a Widget subclass are

// always marked "final".



finalString title;



@override

  _MyHomePageState createState()=>_MyHomePageState();}

class _MyHomePageState extendsState<MyHomePage>{

int _counter =0;



void_incrementCounter(){

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++;

});
```

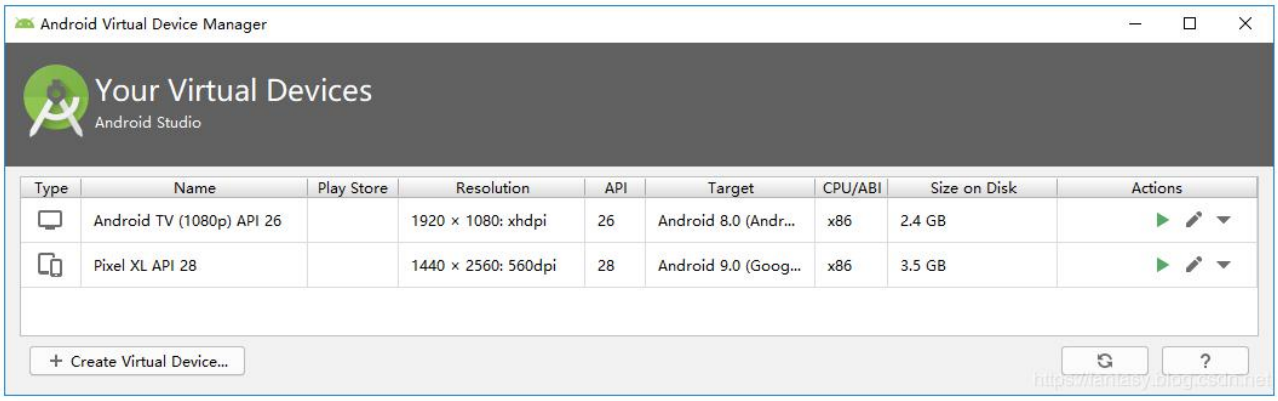
```
}



@override

Widgetbuild(BuildContext context){

// This method is rerun every time setState is called, for instance as do
ne

// by the _incrementCounter method above.

//

// The Flutter framework has been optimized to make rerunning build meth
ods

// fast, so that you can just rebuild anything that needs updating rather

// than having to individually change instances of widgets.

returnScaffold(

     appBar:AppBar(

// Here we take the value from the MyHomePage object that was created by

// the App.build method, and use it to set our appbar title.

       title:Text(widget.title),

),

     body:Center(

// Center is a layout widget. It takes a single child and positions it

// in the middle of the parent.

       child:Column(

// Column is also layout widget. It takes a list of children and

// arranges them vertically. By default, it sizes itself to fit its
```

```
// children horizontally, and tries to be as tall as its parent.

//

// Invoke "debug painting" (press "p" in the console, choose the

// "Toggle Debug Paint" action from the Flutter Inspector in Android

// Studio, or the "Toggle Debug Paint" command in Visual Studio Code)

// to see the wireframe for each widget.

//

// Column has various properties to control how it sizes itself and

// how it positions its children. Here we use mainAxisAlignment to

// center the children vertically; the main axis here is the vertical

// axis because Columns are vertical (the cross axis would be

// horizontal).

          mainAxisAlignment:MainAxisAlignment.center,

          children:<Widget>[

Text(

'You have pushed the button this many times:',

),

Text(

'$_counter',

              style:Theme.of(context).textTheme.display1,

),

],

),
```

```
),

     floatingActionButton:FloatingActionButton(

       onPressed: _incrementCounter,

       tooltip:'Increment',

       child:Icon(Icons.add),

),// This trailing comma makes auto-formatting nicer for build methods.

);

}}
```

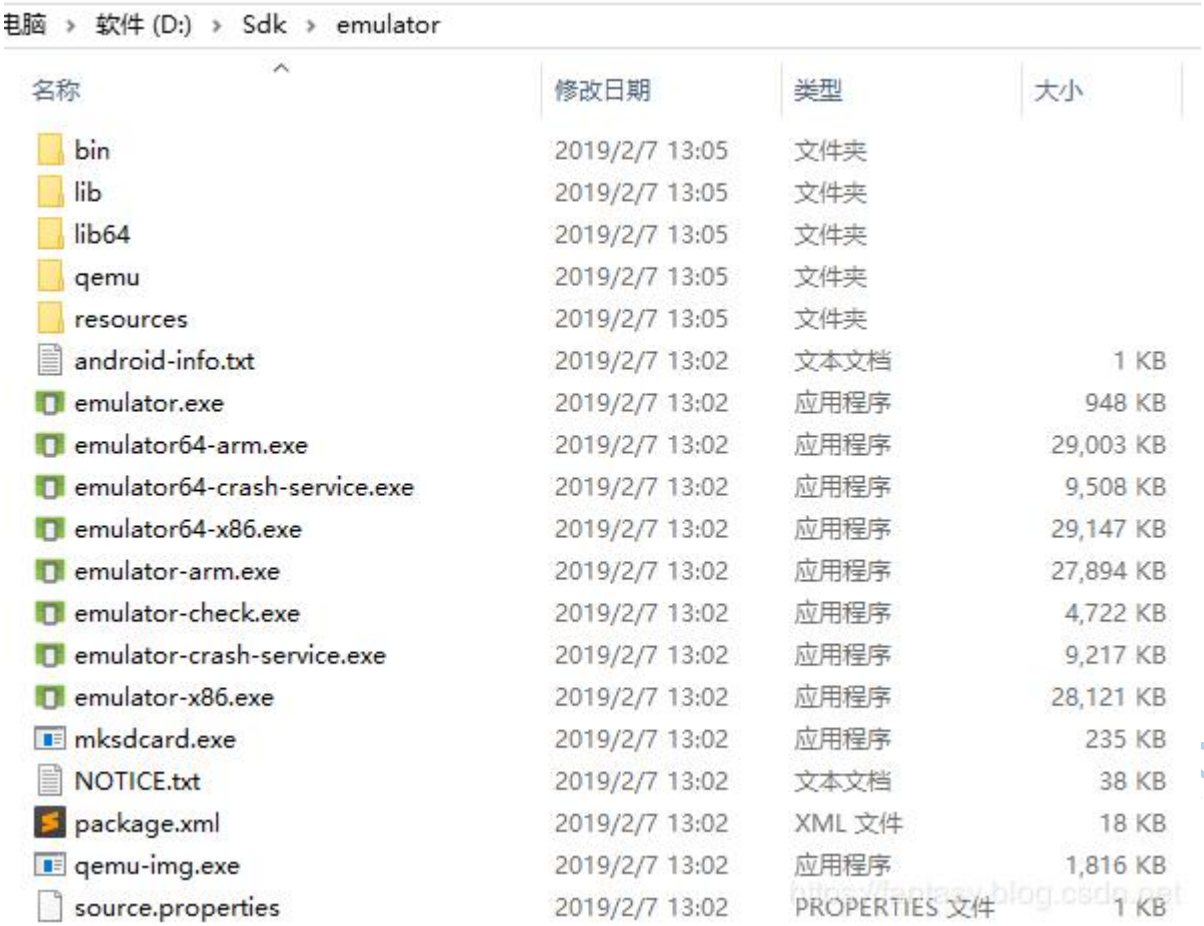
4.模拟器的安装与调试

项目新建完毕了，接下来就是编译运行 Flutter 项目到真机或者模拟器了。先说模拟器，模拟器在我们下载的 Android SDK 的目录里，可以通过两种方法创建模拟器，推荐在 Android Studio 里新建一个模拟器，点击进入 AVD Manager，如果没有模拟器的话，就创建一个即可，可以选择最新的 SDK：



创建完毕后，我们就可以在电脑的模拟器目录看到我们创建的模拟器里：

对应的模拟器 AVD Manager 相关也在 Android SDK 目录下：



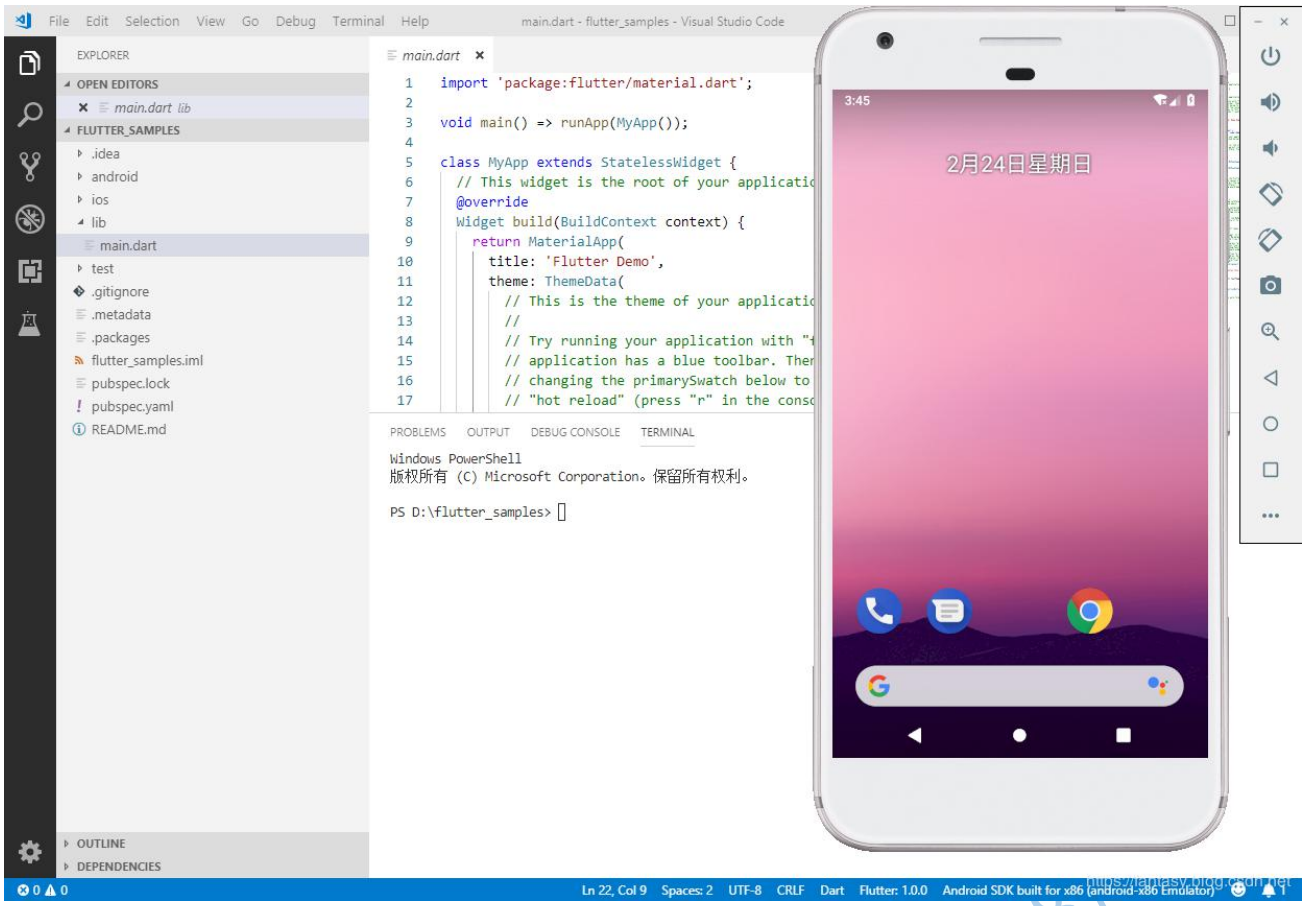
接下来我们就可以关闭相关窗口了，建立一个 bat 文件，写入启动模拟器的命令，

这样每次启动模拟器直接运行这个 bat 文件即可：

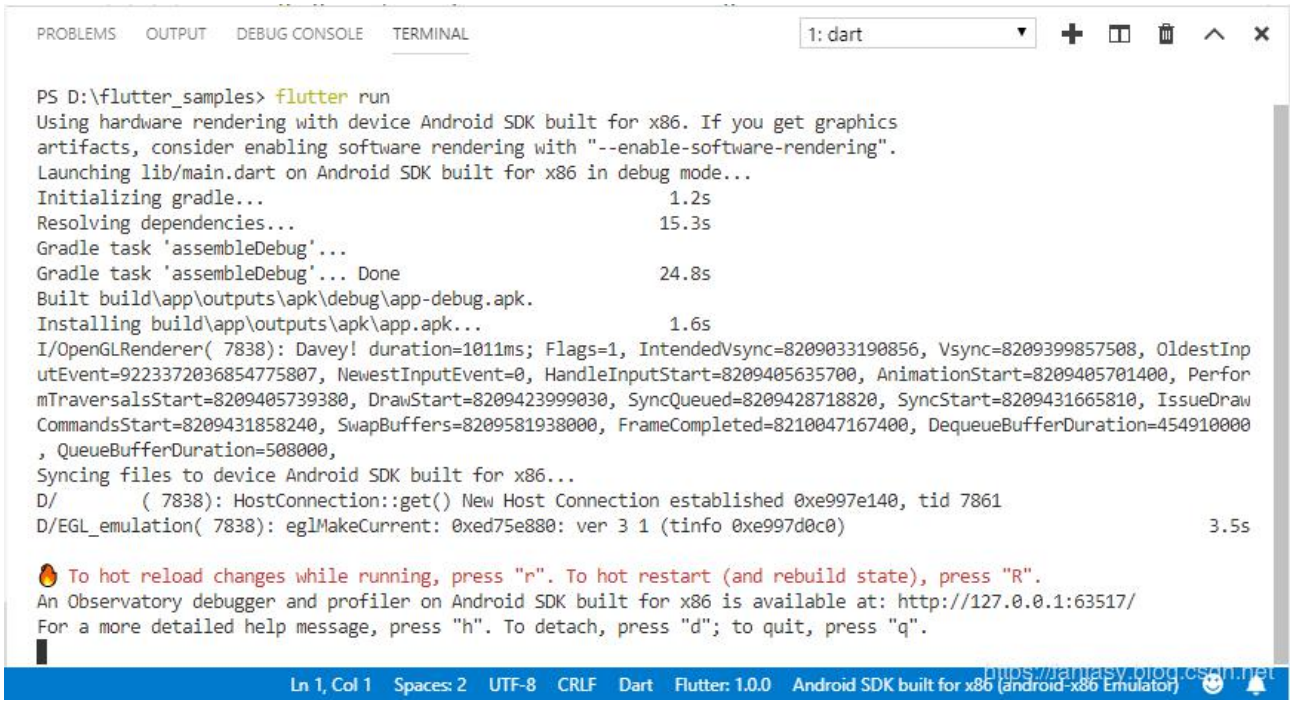
```
D:\Sdk\emulator\emulator.exe -avd Pixel_XL_API_28
```

模拟器所在的 SDK 目录根据你的实际情况位置修改即可。

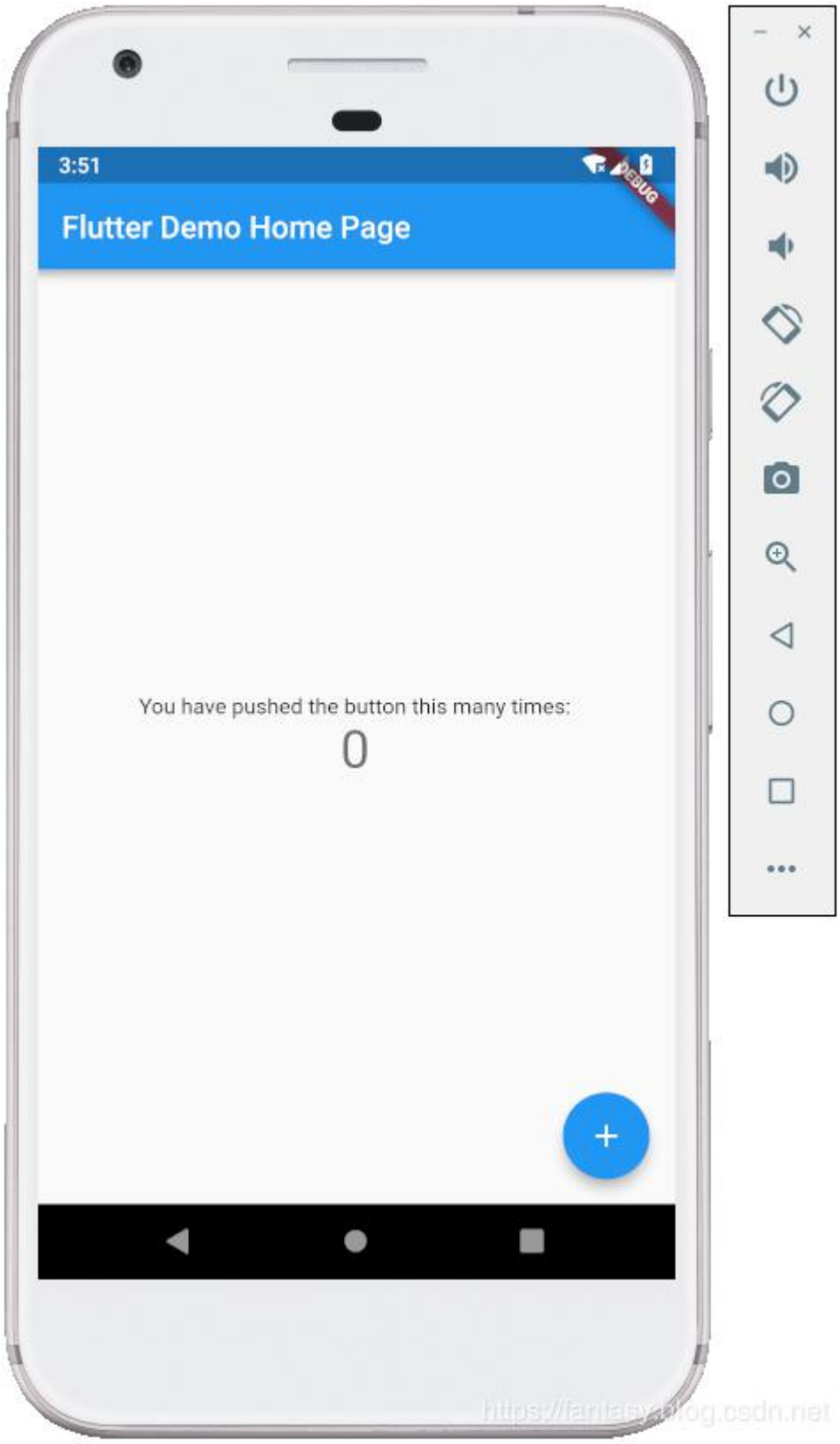
接下来，双击这个 bat 文件运行模拟器：



接着在项目所在目录运行 flutter run 命令即可编译运行 flutter 项目到模拟器上：



运行效果如下图：



运行成功后，后续运行调试只要不退出应用界面，就可以进行热重载，输入 r 进行热重载当前页面，输入 R 进行整个应用的热重启，输入 h 弹出帮助信息，输入 d 解除关联，输入 q 退出应用调试。如果遇到有多个模拟器或者模拟器和真机同时存在的话，可以通过-d 参数加设备 ID 指定要运行的设备，例如：

```
flutter run -d emulator-5556
```

可以通过 flutter devices 或 adb devices 命令查看目前已连接的设备信息。
还有一种命令方式创建模拟器，输入如下命令可以查看当前可用的模拟器：

```
flutter emulator
```

输入以下命令可以创建指定名称的模拟器，默认创建的模拟器 Android 版本号为已安装的最新的 SDK 版本号：

```
flutter emulators --create --name xyz
```

运行以下命令可以启动模拟器：

```
flutter emulators --launch <emulator id>
```

替换为你的模拟器 ID 名称即可。

真机设备运行调试和模拟器的过程基本一样，手机和电脑通过 USB 连接，手机开启开发人员选项和 USB 调试，最后运行 flutter run 命令即可。
其他常用的命令如下：

```
flutter build apk;//打包 Android 应用

flutter build apk -release;

flutter install;//安装应用

flutter build ios;//打包 IOS 应用
```

```
flutter build ios -release;

flutter clean;//清理重新编译项目

flutter upgrade;//升级 Flutter SDK 和依赖包

flutter channel;//查看 Flutter 官方分支列表和当前项目使用的 Flutter 分支

flutter channel <分支名>;//切换分支
```

好了，Flutter 开发环境搭建和调试就为大家讲解到这里了。

第五章 Dart 语法篇之基础语法(一)

简述：

又是一段新的开始，Dart 这门语言相信很多人都是通过 Flutter 这个框架才了解的，因为 Flutter 相比 Dart 更被我们所熟知。很多人迟迟不愿尝试 Flutter 原因大多数是因为学习成本高，显然摆在面前的是需要去重新学习一门新的语言 dart，然后再去学习一个开发框架 Flutter,再加上很多莫名奇妙的坑,不说多的就从 Github 上 Flutter 项目 issue 数来看坑是着实不少，所以很多人也就望而却步了。当然这个问题是一个新的技术框架刚开始都可能存在的，但我们更需要看到 Flutter 框架跨平台技术思想先进性。

为什么要开始一系列 Dart 相关的文章?

很简单，就是为了更好地开发 Flutter，其实开发 Flutter 使用 Dart 的核心知识点并不需要太过于全面，有些东西根本用不到，所以该系列文章也将是有选择性选取 Dart 一些常用技术点讲解。另一方面，读过我文章的小伙伴就知道我是 Kotlin 的狂热忠实粉，其实语言都是相通，你可以从 Dart 身上又能看到 Kotlin 的身影，所以我上手 Dart 非常快，可以对比着学习。所以后期 Dart 文章我会将 Dart 与 Kotlin 作为对比讲解，所以如果你学过 Kotlin 那么恭喜你，上手 Dart 将会非常快。

该系列 Dart 文章会讲哪些内容呢?

本系列文章主要会涉及以下内容: dart 基本语法、变量常量和类型推导、集合、函数、面向对象的 Mixins、泛型、生成器函数、Async 和 Await、Stream 和 Future、Isolate 和 EventLoop 以及最后基本介绍下 DartVM 的工作原理。

一、Hello Dart

这是第一个 Hello Dart 程序，很多程序入口都是从 main 函数开始，所以 dart 也不例外，一起来看下百变的 main 函数

```
//main 标准写法 voidmain(){

print('Hello World!');//注意：Dart 和 Java 一样表达式以分号结尾，写习惯 Kotlin 的小伙伴需要注意了，这可能是你从 Kotlin 转 Dart 最大不适之一。}

//dart 中所有函数的类型都是可以省略的，省略的时候返回值类型为 dynamicmain(){

print('Hello World!');}

//如果函数内部只有一个表达式，可以省略大括号，使用"=>"箭头函数；//而对于 Kotlin 则是如果只有一个表达式，可以省略大括号，使用"="连接，类似 fun main(args: Array<String>) = println('Hello World!')voidmain()=>print('Hello World!');

//最简写形式 main()=>print('Hello World!');
```

二、数据类型

在 dart 中的一切皆是对象，包括数字、布尔值、函数等，它们和 Java 一样都继承于 Object，所以它们的默认值也就是 null. 在 dart 主要有：布尔类型 bool、数字类型 num(数字类型又分为 int，double，并且两者父类都是 num)、字符串类型 String、集合类型(List, Set, Map)、Runes 类和 Symbols 类型(后两个用的并不太多)

1、布尔类型(bool)

在 dart 中和 C 语言一样都是使用 bool 来声明一个布尔类型变量或常量，而在 Kotlin 则是使用 Boolean 来声明，但是一致的是它对应的值只有两个 true 和 false.

```
main(){
```

```
bool isClosed =true;//注意，dart 还是和 Java 类似的 [类型][变量名]方式声明，这个和 Kotlin 的 [变量名]:[类型]不一样.

bool isOpened =false;}
```

2、数字类型(num、int、double)

在 dart 中 num、int、double 都是类,然后 int、double 都继承 num 抽象类，这点和 Kotlin 很类似，在 Kotlin 中 Number、Int、Double 都是类，然后 Int、Double 都继承于 Number. 注意，但是在 dart 中没有 float, short, long 类型

```
main(){

    double pi =3.141592653;

    int width =200;

    int height =300;

print(width / height);//注意:这里和 Kotlin、Java 都不一样，两个 int 类型相除是 double 类型小数，而不是整除后的整数。

print(width ~/ height);//注意: 这才是 dart 整除正确姿势}
```

此外和 Java、Kotlin 一样，dart 也拥有一些数字常用的函数:

```
main(){

print(3.141592653.toStringAsFixed(3));//3.142 保留有效数字

print(6.6.floor());//6 向下取整

print((-6.6).ceil());//-6 向上取整

print(9.9.ceil());//10 向上取整

print(666.6.round());//667 四舍五入

print((-666.6).abs());// 666.6 取绝对值

print(666.6.toInt());//666 转化成 int,这中 toInt、toDouble 和 Kotlin 类似
```

```
print(999.isEven);//false 是否是偶数

print(999.isOdd);//true 是否是奇数

print(666.6.toString());//666.6 转化成字符串}
```

3、字符串类型(String)

在 Dart 中支持单引号、双引号、三引号以及$字符串模板用法和 Kotlin 是一模一样的。

```
main(){

    String name ='Hello Dart!';//单引号

    String title ="'Hello Dart!'";//双引号

    String description ="""

          Hello Dart! Hello Dart!

          Hello Dart!

          Hello Dart! Hello Dart!

""";//三引号

    num value =2;

    String result ="The result is $value";//单值引用

    num width =200;

    num height =300;

    String square ="The square is ${width * height}";//表达式的值引用}
```

和 Kotlin 一样，dart 中也有很多字符串操作的方法，比如字符串拆分、子串等

```
main(){

String url ="https://mrale.ph/dartvm/";
```

```
print(url.split("://")[0]);//字符串分割 split 方法，类似 Java 和 Kotlin

print(url.substring(3,9));//字符串截取 substring 方法，类似 Java 和 Kotlin

print(url.codeUnitAt(0));//取当前索引位置字符的 UTF-16 码

print(url.startsWith("https"));//当前字符串是否以指定字符开头，类似 Java 和 K
otlin

print(url.endsWith("/"));//当前字符串是否以指定字符结尾，类似 Java 和 Kotlin

print(url.toUpperCase());//大写，类似 Java 和 Kotlin

print(url.toLowerCase());//小写，类似 Java 和 Kotlin

print(url.indexOf("ph"));//获取指定字符的索引位置，类似 Java 和 Kotlin

print(url.contains("http"));//字符串是否包含指定字符，类似 Java 和 Kotlin

print(url.trim());//去除字符串的首尾空格，类似 Java 和 Kotlin

print(url.length);//获取字符串长度
```

```
print(url.replaceFirst("t","A"));//替换第一次出现 t 字符位置的字符

print(url.replaceAll("m","M"));//全部替换，类似 Java 和 Kotlin}
```

4、类型检查(is 和 is!)和强制类型转换(as)

和 Kotlin 一样，dart 也是通过 is 关键字来对类型进行检查以及使用 as 关键字对类型进行强制转换，如果判断不是某个类型 dart 中使用 is!，而在 Kotlin 中正好相反则用 !is 表示。

```
main(){
    int number =100;
    double distance =200.5;
    num age =18;
print(number is num);//true
print(distance is! int);//true
print(age as int);//18}
```

5、Runes 和 Symbols 类型

在 Dart 中的 Runes 和 Symbols 类型使用并不多，这里做个简单的介绍, Runes 类型是 UTF-32 字节单元定义的 Unicode 字符串，Unicode 可以使用数字表示字母、数字和符号，然而在 dart 中 String 是一系列的 UTF-16 的字节单元，所以想要表示 32 位的 Unicode 的值，就需要用到 Runes 类型。我们一般使用\uxxxx 这种形式来表示一个 Unicode 码，xxxx 表示 4 个十六进制值。当十六进制数据多余或者少于 4 位时，将十六进制数放入到花括号中，例如，微笑表情（

```
main(){

var clapping ='\u{1f44f}';

print(clapping);

print(clapping.codeUnits);//返回十六位的字符单元数组

print(clapping.runes.toList());



  Runes input =newRunes(

'\u2665  \u{1f605}  \u{1f60e}  \u{1f47b}  \u{1f596}  \u{1f44d}');

print(newString.fromCharCodes(input));}
```

6、Object 类型

在 Dart 中所有东西都是对象，都继承于 Object，所以可以使用 Object 可以定义任何的变量，而且赋值后，类型也可以更改。

```
main(){

   Object color ='black';

   color =0xff000000;//运行正常，0xff000000 类型是 int，int 也继承于 Object
 }
```

7、dynamic 类型

在 Dart 中还有一个和 Object 类型非常类似的类型那就是 dynamic 类型，下面讲到的 var 声明的变量未赋值的时候就是 dynamic 类型，它可以像 Object 一样可以改变类型。dynamic 类型一般用于无法确定具体类型，注意：建议不要滥用 dynamic，一般尽量使用 Object，如果你对 Flutter 和 Native 原生通信 PlatformChannel 代码熟悉的话，你会发现里面大量使用了 dynamic，因为可能 native 数据类型无法对应 dart 中的数据类型,此时 dart 接收一般就会使用

dynamic.

Object 和 dynamic 区别在于: Object 会在编译阶段检查类型，而 dynamic 不会在编译阶段检查类型。

```
main(){

dynamic color ='black';

    color =0xff000000;//运行正常，0xff000000 类型是 int，int 也继承于 Object}
```

三、变量和常量

1、var 关键字

在 dart 中可以使用 var 来替代具体类型的声明，会自动推导变量的类型，这是因为 var 并不是直接存储值，而是存储值的对象引用，所以 var 可以声明任何变量。这一点和 Kotlin 不一样，在 Kotlin 中声明可变的变量都必须需要使用 var 关键字，而 Kotlin 的类型推导是默认行为和 var 并没有直接关系。注意: 在 Flutter 开发一般会经常使用 var 声明变量，以便于可以自动推导变量的类型。

```
main(){

int colorValue =0xff000000;

var colorKey ='black';//var 声明变量 自动根据赋值的类型，推导为 String 类型

// 使用 var 声明集合变量

var colorList =['red','yellow','blue','green'];

var colorSet ={'red','yellow','blue','green'};

var colorMap ={'white':0xffffffff,'black':0xff000000};}
```

但是在使用 var 声明变量的时候，需要注意的是：如果 var 声明的变量开始不初始化，不仅值可以改变它的类型也是可以被修改的，但是一旦开始初始化赋值后，它的类型就确定了，后续不能被改变。

```
main(){

var color;// 仅有声明未赋值的时候，这里的 color 的类型是 dynamic，所以它的类型是可以变的

  color ='red';

print(color is String);//true

  color =0xffff0000;

print(color is int);//true



var colorValue =0xffff0000;//声明时并赋值，这里 colorValue 类型已经推导出为 int，并且确定了类型

  colorValue ='red';//错误，这里会抛出编译异常，String 类型的值不能赋值给 int 类型

print(colorValue is int);//true}
```

2、常量(final 和 const)

在 dart 中声明常量可以使用 const 或 final 两个关键字，注意：这两者的区别在于如果常量是编译期就能初始化的就用 const(有点类似 Kotlin 中的 const val) 如果常量是运行时期初始化的就用 final(有点类似 Kotlin 中的 val)

```
main(){

const PI =3.141592653;//const 定义常量

final nowTime = DateTime.now();//final 定义常量}
```

四、集合(List、Set、Map)

1、集合 List

在 dart 中的 List 和 Kotlin 还是很大的区别，换句话说 Dart 整个集合类型系统的划分都和 Kotlin 都不一样，比如 Dart 中集合就没有严格区分成可变集合(Kotlin 中 MutableList)和不变集合(Kotlin 中的 List)，在使用方式上你会感觉它更像数组，但是它是可以随意对元素增删改成的。

```
List 初始化方式 main(){

    List<String> colorList =['red','yellow','blue','green'];//直接使用[]形式初始化

var colorList =<String>['red','yellow','blue','green'];}
```

List 常用的函数

```
main(){

    List<String> colorList =['red','yellow','blue','green'];

    colorList.add('white');//和 Kotlin 类似通过 add 添加一个新的元素

print(colorList[2]);//可以类似 Kotlin 一样，直接使用数组下标形式访问元素

print(colorList.length);//获取集合的长度，这个 Kotlin 不一样，Kotlin 中使用的是 size

    colorList.insert(1,'black');//在集合指定 index 位置插入指定的元素

    colorList.removeAt(2);//移除集合指定的 index=2 的元素，第 3 个元素

    colorList.clear();//清除所有元素

print(colorList.sublist(1,3));//截取子集合

print(colorList.getRange(1,3));//获取集合中某个范围元素

print(colorList.join('<--->'));//类似 Kotlin 中的 joinToString 方法，输出：red<--->yellow<--->blue<--->green

print(colorList.isEmpty);
```

```
print(colorList.contains('green'));}
```

List 的遍历方式

```
main(){

    List<String> colorList =['red','yellow','blue','green'];

//for-i 遍历

for(var i =0; i < colorList.length; i++){//可以使用 var 或 int

print(colorList[i]);

}

//forEach 遍历

    colorList.forEach((color)=>print(color));//forEach 的参数为 Function.
=>使用了箭头函数

//for-in 遍历

for(var color in colorList){

print(color);

}

//while+iterator 迭代器遍历，类似 Java 中的 iteator

while(colorList.iterator.moveNext()){

print(colorList.iterator.current);

}}
```

2、集合 Set

集合 Set 和列表 List 的区别在于 集合中的元素是不能重复 的。所以添加重复的元素时会返回 false,表示添加不成功.

Set 初始化方式

```
main(){

    Set<String> colorSet={'red','yellow','blue','green'};//直接使用{}形式初始化

var colorList =<String>{'red','yellow','blue','green'};}
```

集合中的交、并、补集，在 Kotlin 并没有直接给到计算集合交、并、补的 API

```
main() {

    var colorSet1 = {'red', 'yellow', 'blue', 'green'};

    var colorSet2 = {'black', 'yellow', 'blue', 'green', 'white'};

    print(colorSet1.intersection(colorSet2));//交集-->输出: {'yellow', 'blue', 'green'}

    print(colorSet1.union(colorSet2));//并集--->输出: {'black', 'red', 'yellow', 'blue', 'green', 'white'}

    print(colorSet1.difference(colorSet2));//补集--->输出: {'red'}

}
```

Set 的遍历方式(和 List 一样)

```
main(){

    Set<String> colorSet ={'red','yellow','blue','green'};

//for-i 遍历

for(var i =0; i < colorSet.length; i++){

//可以使用 var 或 int

print(colorSet[i]);

}
```

```
//forEach 遍历

    colorSet.forEach((color)=>print(color));//forEach 的参数为 Function.
=>使用了箭头函数

//for-in 遍历

for(var color in colorSet){

print(color);

}

//while+iterator 迭代器遍历，类似 Java 中的 iteator

while(colorSet.iterator.moveNext()){

print(colorSet.iterator.current);

}

}
```

3、集合 Map

集合 Map 和 Kotlin 类似，key-value 形式存储，并且 Map 对象的中 key 是不能重复的

```
Map 初始化方式 main(){

    Map<String, int> colorMap ={'white':0xffffffff,'black':0xff000000};/
/使用{key:value}形式初始化

var colorMap =<String, int>{'white':0xffffffff,'black':0xff000000};}
```

Map 中常用的函数

```
main(){

    Map<String, int> colorMap ={'white':0xffffffff,'black':0xff000000};

print(colorMap.containsKey('green'));//false
```

```
print(colorMap.containsValue(0xff000000));//true

print(colorMap.keys.toList());//['white','black']

print(colorMap.values.toList());//[0xffffffff, 0xff000000]

    colorMap['white']=0xfffff000;//修改指定 key 的元素

    colorMap.remove('black');//移除指定 key 的元素}
```

Map 的遍历方式

```
main(){

    Map<String, int> colorMap ={'white':0xffffffff,'black':0xff000000};

//for-each key-value

    colorMap.forEach((key, value)=>print('color is $key, color value is
$value'));}
```

Map.fromIterables 将 List 集合转化成 Map

```
main(){

    List<String> colorKeys =['white','black'];

    List<int> colorValues =[0xffffffff,0xff000000];

    Map<String, int> colorMap = Map.fromIterables(colorKeys, colorValue
s);}
```

4、集合常用的操作符

dart 对于集合操作的也非常符合现代语言的特点，含有丰富的集合操作符 API，可以让你处理结构化的数据更加简单。

```
main(){

  List<String> colorList =['red','yellow','blue','green'];

//forEach 箭头函数遍历
```

```
  colorList.forEach((color)=>{print(color)});

  colorList.forEach((color)=>print(color));//箭头函数遍历，如果箭头函数内部只有一个表达式可以省略大括号



//map 函数的使用

print(colorList.map((color)=>'$color_font').join(","));



//every 函数的使用，判断里面的元素是否都满足条件，返回值为 true/false

print(colorList.every((color)=> color =='red'));



//sort 函数的使用

  List<int> numbers =[0,3,1,2,7,12,2,4];

  numbers.sort((num1, num2)=> num1 - num2);//升序排序

  numbers.sort((num1, num2)=> num2 - num1);//降序排序

print(numbers);



//where 函数使用，相当于 Kotlin 中的 filter 操作符，返回符合条件元素的集合

print(numbers.where((num)=> num >6));



//firstWhere 函数的使用，相当于 Kotlin 中的 find 操作符，返回符合条件的第一个元素，如果没找到返回 null

print(numbers.firstWhere((num)=> num ==5, orElse:()=>-1));//注意：如果没有找到，执行 orElse 代码块，可返回一个指定的默认值
```

```
//singleWhere 函数的使用，返回符合条件的第一个元素，如果没找到返回 null，但是前提是集合中只有一个符合条件的元素，否则就会抛出异常

print(numbers.singleWhere((num)=> num ==4, orElse:()=>-1));//注意：如果没有找到，执行 orElse 代码块，可返回一个指定的默认值



//take(n)、skip(n)函数的使用，take(n)表示取当前集合前 n 个元素，skip(n)表示跳过前 n 个元素，然后取剩余所有的元素

print(numbers.take(5).skip(2));



//List.from 函数的使用，从给定集合中创建一个新的集合,相当于 clone 一个集合

print(List.from(numbers));



//expand 函数的使用，将集合一个元素扩展成多个元素或者将多个元素组成二维数组展开成平铺一个一位数组

var pair =[

[1,2],

[3,4]

];

print('flatten list: ${pair.expand((pair) => pair)}');



var inputs =[1,2,3];

print('duplicated list: ${inputs.expand((number)=>[

    number,

    number,

    number
```

```
])}');}
```

五、流程控制

1、for 循环

```
main(){

    List<String> colorList =['red','yellow','blue','green'];

for(var i =0; i < colorList.length; i++){//可以用 var 或 int

print(colorList[i]);

}}
```

2、while 循环

```
main(){

    List<String> colorList =['red','yellow','blue','green'];

var index =0;

while(index < colorList.length){

print(colorList[index++]);

}}
```

3、do-while 循环

```
main(){

    List<String> colorList =['red','yellow','blue','green'];

var index =0;

do{

print(colorList[index++]);
```

```
}while(index < colorList.length);}
```

4、break 和 continue

```
main(){

    List<String> colorList =['red','yellow','blue','green'];

for(var i =0; i < colorList.length; i++){//可以用 var 或 int

if(colorList[i]=='yellow'){

continue;

}

if(colorList[i]=='blue'){

break;

}

print(colorList[i]);

}}
```

5、if-else

```
voidmain(){

var numbers =[1,2,3,4,5,6,7,8,9,10,11];

for(var i =0; i < numbers.length; i++){

if(numbers[i].isEven){

print('偶数: ${numbers[i]}');

}elseif(numbers[i].isOdd){

print('奇数: ${numbers[i]}');

}else{
```

```
print('非法数字');

}

}}
```

6、三目运算符(?:)

```
voidmain(){

var numbers =[1,2,3,4,5,6,7,8,9,10,11];

for(var i =0; i < numbers.length; i++){

      num targetNumber = numbers[i].isEven ? numbers[i]*2: numbers[i]+4;

print(targetNumber);

}}
```

7、switch-case 语句

```
Color getColor(String colorName){

  Color currentColor = Colors.blue;

switch(colorName){

case"read":

      currentColor = Colors.red;

break;

case"blue":

      currentColor = Colors.blue;

break;

case"yellow":

      currentColor = Colors.yellow;
```

```
break;

}

return currentColor;}
```

8、Assert(断言)

在 dart 中如果条件表达式结果不满足条件，则可以使用 assert 语句中断代码的执行。特别是在 Flutter 源码中随处可见都是 assert 断言的使用。注意: 断言只在检查模式下运行有效，如果在生产模式运行，则断言不会执行。

```
assert(text !=null);//text 为 null,就会中断后续代码执行 assert(urlString.sta
rtsWith('https'));
```

六、运算符

1、算术运算符

名称	运算符	例子
加	+	var result = 1 + 1;
减	-	var result = 5 - 1;
乘	*	var result = 3 * 5;
除	/	var result = 3 / 5; //0.6
整除	~/	var result = 3 ~/ 5; //0
取余	%	var result = 5 % 3; //2

2、条件运算符

名称	运算符	例子
大于	>	2 > 1
小于	<	1 < 2
等于	==	1 == 1
不等于	!=	3 != 4
大于等于	>=	5 >= 4
小于等于	<=	4 <= 5

3、逻辑运算符

名称	运算符	例子
或	||	2 > 1 || 3 < 1
与	&&	2 > 1 && 3 < 1
非	!	!(2 > 1)

4、位运算符

名称	运算符
位与	&
位或	|
位非	~
异或	^
左移	<<
右移	>>

5、三目运算符

condition ? expr1 : expr2

```
var isOpened =(value==1)?true:false;
```

6、空安全运算符

（1）result = expr1 ?? expr2

如果发现 expr1 为 null,就返回 expr2 的值，否则就返回 expr1 的值，这个类似于 Kotlin 中的 result = expr1 ?: expr2

```
main(){

var choice = question.choice ??'A';

//等价于
```

```
var choice2;

if(question.choice ==null){

        choice2 ='A';

}else{

        choice2 = question.choice;

}}
```

（2）expr1 ??= expr2 等价于 expr1 = expr1 ?? expr2 (转化成第一种)

```
main(){

var choice ??='A';

//等价于

if(choice ==null){

        choice ='A';

}}
```

（3）result = expr1?.value

如果 expr1 不为 null 就返回 expr1.value，否则就会返回 null，类似 Kotlin 中的 ?.

如果 expr1 不为 null,就执行后者

```
var choice = question?.choice;//等价于 if(question ==null){

returnnull;}else{

return question.choice;}



question?.commit();//等价于 if(question ==null){

return;//不执行 commit()}else{
```

```
    question.commit();//执行 commit 方法     }
```

7、级联操作符(..)

级联操作符是 .., 可以让你对一个对象中字段进行链式调用操作，类似 Kotlin 中的 apply 或 run 标准库函数的使用。

```
question

..id ='10001'

..stem ='第一题: xxxxxx'

..choices =<String>['A','B','C','D']

..hint ='听音频做题';

复制代码 Kotlin 中的 run 函数实现对比

question.run {

    id ='10001'

    stem ='第一题: xxxxxx'

    choices =listOf('A','B','C','D')

    hint ='听音频做题'}
```

8、运算符重载

在 dart 支持运算符自定义重载,使用 operator 关键字定义重载函数

```
classVip{

final int level;

final int score;



constVip(this.level,this.score);
```

```
  bool operator>(Vip other)=>

      level > other.level ||(level == other.level && score > other.scor
e);



  bool operator<(Vip other)=>

      level < other.level ||(level == other.level && score < other.scor
e);



  bool operator==(Vip other)=>

      level == other.level &&

      score == other.level;//注意：这段代码可能在高版本的 Dart 中会报错，在低
版本是 OK 的

//上述代码，在高版本 Dart 中，Object 中已经重载了==,所以需要加上 covariant 关键
字重写这个重载函数。

@override

  bool operator==(covariant Vip other)=>

(level == other.level && score == other.score);



@override

  int get hashCode =>super.hashCode;//伴随着你还需要重写 hashCode，至于什么
原因大家应该都知道}



main(){

var userVip1 =Vip(4,3500);
```

```
var userVip2 =Vip(4,1200);

if(userVip1 > userVip2){

print('userVip1 is super vip');

}elseif(userVip1 < userVip2){

print('userVip2 is super vip');

}}
```

七、异常

dart 中的异常捕获方法和 Java,Kotlin 类似，使用的也是 try-catch-finally; 对特定异常的捕获使用 on 关键字. dart 中的常见异常有: NoSuchMethodError(当在一个对象上调用一个该对象没有 实现的函数会抛出该错误)、ArgumentError (调用函数的参数不合法会抛出这个错误)

```
main(){

  int num =18;

  int result =0;

try{

    result = num ~/0;

}catch(e){//捕获到 IntegerDivisionByZeroException

print(e.toString());

}finally{

print('$result');

}}

//使用 on 关键字捕获特定的异常 main(){
```

```
  int num =18;

  int result =0;

try{

    result = num ~/0;

} on IntegerDivisionByZeroException catch(e){//捕获特定异常

print(e.toString());

}finally{

print('$result');

}}
```

八、函数

在 dart 中函数的地位一点都不亚于对象，支持闭包和高阶函数，而且 dart 中的函数也会比 Java 要灵活的多，而且 Kotlin 中的一些函数特性，它也支持甚至比 Kotlin 支持的更全面。比如支持默认值参数、可选参数、命名参数等.

1、函数的基本用法

```
main(){

print('sum is ${sum(2, 5)}');}



num sum(num a, num b){

return a + b;}
```

2、函数参数列表传参规则

```
//num a, num b, num c, num d 最普通的传参：调用时，参数个数和参数顺序必须固定
add1(num a, num b, num c, num d){
```

```
print(a + b + c + d);}

//[num a, num b, num c, num d]传参：调用时，参数个数不固定，但是参数顺序需要一一对应，不支持命名参数 add2([num a, num b, num c, num d]){

print(a + b + c + d);}

//{num a, num b, num c, num d}传参：调用时，参数个数不固定，参数顺序也可以不固定，支持命名参数,也叫可选参数，是 dart 中的一大特性，这就是为啥 Flutter 代码那么多可选属性，大量使用可选参数 add3({num a, num b, num c, num d}){

print(a + b + c + d);}

//num a, num b, {num c, num d}传参：调用时，a,b 参数个数固定顺序固定，c,d 参数个数和顺序也可以不固定 add4(num a, num b,{num c, num d}){

print(a + b + c + d);}

main(){

add1(100,100,100,100);//最普通的传参：调用时，参数个数和参数顺序必须固定

add2(100,100);//调用时，参数个数不固定，但是参数顺序需要一一对应，不支持命名参数(也就意味着顺序不变)

add3(

      b:200,

      a:200,

      c:100,

      d:100);//调用时，参数个数不固定，参数顺序也可以不固定，支持命名参数(也就意味着顺序可变)

add4(100,100, d:100, c:100);//调用时，a,b 参数个数固定顺序笃定，c,d 参数个数和顺序也可以不固定}
```

3、函数默认参数和可选参数(以及与 Kotlin 对比)

dart 中函数的默认值参数和可选参数和 Kotlin 中默认值参数和命名参数一致，只是写法上不同而已

```
add3({num a, num b, num c, num d =100}){//d 就是默认值参数，给的默认值是 100

print(a + b + c + d);}

main(){

add3(b:200, a:100, c:800);}
```

与 Kotlin 对比

```
funadd3(a: Int, b: Int, c: Int, d: Int =100){

println(a + b + c + d)}

funmain(args: Array<String>){

add3(b =200, a =100, c =800)}
```

4、函数类型与高阶函数

在 dart 函数也是一种类型 Function,可以作为函数参数传递，也可以作为返回值。

类似 Kotlin 中的 FunctionN 系列函数

```
main(){

Function square =(a){

return a * a;

};



Function square2 =(a){

return a * a * a;

};



add(3,4, square, square2)}
```

```
num add(num a, num b,[Function op,Function op2]){

//函数作为参数传递

returnop(a)+op2(b);}
```

5、函数的简化以及箭头函数

在 dart 中的如果在函数体内只有一个表达式，那么就可以使用箭头函数来简化代码，这点也和 Kotlin 类似，只不过在 Kotlin 中人家叫 lambda 表达式，只是写法上不一样而已。

```
add4(num a, num b,{num c, num d}){

print(a + b + c + d);}

add5(num a, num b,{num c, num d})=>print(a + b + c + d);

复制代码九、面向对象

在 dart 中一切皆是对象，所以面向对象在 Dart 中依然举足轻重，下面就先通过一个简单的例子认识下 dart 的面向对象，后续会继续深入。1、类的基本定义和使用 abstractclassPerson{

    String name;

    int age;

    double height;

Person(this.name,this.age,this.height);//注意，这里写法可能大家没见过， 这点和 Java 是不一样，这里实际上是一个 dart 的语法糖。但是这里不如 Kotlin，Kotlin 是直接把 this.name 传值的过程都省了。

//与上述的等价代码,当然这也是 Java 中必须要写的代码

Person(String name, int age, double height){

this.name = name;

this.age = age;
```

```
this.height = height;

}

//然而 Kotlin 很彻底只需要声明属性就行,下面是 Kotlin 实现代码

abstractclassPerson(val name: String, val age: Int, val height: Double)}

classStudentextendsPerson{//和 Java 一样同时使用 extends 关键字表示继承

Student(String name, int age, double height, double grade):super(name, a
ge, height);//在 Dart 里：类名(变量，变量,...) 是构造函数的写法，:super()表示
该构造调用父类，这里构造时传入三个参数}
```

2、类中属性的 getter 和 setter 访问器(类似 Kotlin)

```
abstractclassPerson{

  String _name;////相当于 kotlin 中的 var 修饰的变量有 setter、getter 访问器，
在 dart 中没有访问权限，默认_下划线开头变量表示私有权限，外部文件无法访问

final int _age;//相当于 kotlin 中的 val 修饰的变量只有 getter 访问器

Person(this._name,this._age);//这是上述简写形式



//使用 set 关键字 计算属性 自定义 setter 访问器

setname(String name)=> _name = name;

//使用 get 关键字 计算属性 自定义 getter 访问器

  bool get isStudent => _age >18;}
```

总结

这是 dart 的第一篇文章，主要就是从整体上介绍了下 dart 的语法，当然里面还有一些东西需要深入，后续会继续深入探讨。整体看下有没有觉得 Kotlin 和 dart

语法很像，其实里面有很多特性都是现代编程语言的特性，包括你在其他语言中同样能看到比如 swift 等。就到这里，后面继续聊 dart 和 flutter...

第六章 Dart 语法篇之集合的使用与源码解析(二)

简述:

我们将继续 Dart 语法的第二篇集合，虽然集合在第一篇中已经介绍的差不多，但是在这篇文章中将会更加全面介绍有关 Dart 中的集合，因为之前只是介绍了 dart:core 包中的 List、Set、Map，实际上在 dart 中还提供一个非常丰富的 dart:collection 包，看过集合源码小伙伴都知道 dart:core 包中的集合实际上是委托到 dart:collection 包中实现的，所以下面我也会从源码的角度去把两者联系起来。当然这里也只会选择几个常用的集合作为介绍。

一、List

在 dart 中的 List 集合是具有长度的可索引对象集合，它没有委托 dart:collection 包中集合实现，完全由内部自己实现。

初始化

```
main(){

//初始化一:直接使用[]形式初始化

     List<String> colorList1 =['red','yellow','blue','green'];



//初始化二: var + 泛型

var colorList2 =<String>['red','yellow','blue','green'];



//初始化三: 初始化定长集合

     List<String> colorList3 =List(4);//初始化指定大小为 4 的集合,

     colorList3.add('deepOrange');//注意: 一旦指定了集合长度，不能再调用 add 方法，否则会抛出 Cannot add to a fixed-length list。也容易理解因为一个定长的集合不能再扩展了。

print(colorList3[2]);//null,此外初始化 4 个元素默认都是 null



//初始化四: 初始化空集合且是可变长的

    List<String> colorList4 =List();//相当于 List<String> colorList4 =
  []

    colorList4[2]='white';//这里会报错，[]=实际上就是一个运算符重载，表示修改指定 index 为 2 的元素为 white，然而它长度为 0 所以找不到 index 为 2 元素，所以会抛出 IndexOutOfRangeException

}
```

遍历

```
main(){

     List<String> colorList =['red','yellow','blue','green'];
```

```
//for-i 遍历

for(var i =0; i < colorList.length; i++){//可以使用 var 或 int

print(colorList[i]);

}

//forEach 遍历

      colorList.forEach((color)=>print(color));//forEach 的参数为 Functi
on. =>使用了箭头函数

//for-in 遍历

for(var color in colorList){

print(color);

}

//while+iterator 迭代器遍历，类似 Java 中的 iteator

while(colorList.iterator.moveNext()){

print(colorList.iterator.current);

}}
```

常用的函数

```
main(){

      List<String> colorList =['red','yellow','blue','green'];

      colorList.add('white');//和 Kotlin 类似通过 add 添加一个新的元素

      List<String> newColorList =['white','black'];

      colorList.addAll(newColorList);//addAll 添加批量元素

print(colorList[2]);//可以类似 Kotlin 一样，直接使用数组下标形式访问元素

print(colorList.length);//获取集合的长度，这个 Kotlin 不一样，Kotlin 中使
用的是 size
```

```
      colorList.insert(1,'black');//在集合指定 index 位置插入指定的元素

      colorList.removeAt(2);//移除集合指定的 index=2 的元素，第 3 个元素

      colorList.clear();//清除所有元素

print(colorList.sublist(1,3));//截取子集合

print(colorList.getRange(1,3));//获取集合中某个范围元素

print(colorList.join('<--->'));//类似 Kotlin 中的 joinToString 方法，输出:
 red<--->yellow<--->blue<--->green

print(colorList.isEmpty);

print(colorList.contains('green'));}
```

构造函数源码分析

dart 中的 List 有很多个构造器，一个主构造器和多个命名构造器。主构造器中有个 length 可选参数.

```
externalfactoryList([int length]);//主构造器，传入 length 可选参数，默认为
0

externalfactory List.filled(int length, E fill,{bool growable =fals
e});//filled 命名构造器，只能声明定长的数组

externalfactory List.from(Iterable elements,{bool growable =true});

factory List.of(Iterable<E> elements,{bool growable =true})=>

      List<E>.from(elements, growable: growable);//委托给 List.from 构造
器来实现

externalfactory List.unmodifiable(Iterable elements);
```

exteranl 关键字(插播一条内容)

注意: 问题来了，可能大家看到 List 源码的时候一脸懵逼，构造函数没有具体的实现。不知道有没有注意 到 **exteranl** 关键字。**external 修饰的函数具有一种实现函数声明和实现体分离的特性**。这下应该就明白了，也就是对应实现在别的地方。实际上你可以在 DartSDK 中的源码找到，以 List 举例，对应的是 `sdk/sdk_nnbd/lib/_internal/vm/lib/array_patch.dart`，此外对应的 external 函数实现会有一个 **@patch 注解** 修饰.

```
@patchclassList<E>{

//对应的是 List 主构造函数的实现

@patch

factoryList([int length]) native "List_new";//实际上这里是通过 native 层的 c
++数组来实现，具体可参考 runtime/lib/array.cc



//对应的是 List.filled 构造函数的实现,fill 是需要填充元素值，默认 growable 是 f
alse，默认不具有扩展功能

@patch

factory List.filled(int length, E fill,{bool growable:false}){

var result = growable ?new_GrowableList<E>(length):new_List<E>(length);/
/可以看到如果是可变长，就会创建一个_GrowableList，否则就创建内部私有的_List

if(fill !=null){//fill 填充元素值不为 null,就返回 length 长度填充值为 fill 的集
合

for(int i =0; i < length; i++){

        result[i]= fill;

}

}
```

```
return result;//否则直接返回相应长度的空集合

}



//对应的是 List.from 构造函数的实现,可将 Iterable 的集合加入到一个新的集合中,默认 growable 是 true,默认具备扩展功能

@patch

factory List.from(Iterable elements,{bool growable:true}){

if(elements is EfficientLengthIterable<E>){

      int length = elements.length;

var list = growable ?new_GrowableList<E>(length):new_List<E>(length);//如果是可变长，就会创建一个_GrowableList，否则就创建内部私有的_List

if(length >0){

//只有在必要情况下创建 iterator

        int i =0;

for(var element in elements){

          list[i++]= element;

}

}

return list;

}



//如果 elements 是一个 Iterable<E>,就不需要为每个元素做类型测试

//因为在一般情况下，如果 elements 是 Iterable<E>，在开始循环之前会用单个类型测试替换其中每个元素的类型测试。但是注意下：等等，我发现下面这段源码好像有点问题，难道是我眼神不好，if 和 else 内部执行代码一样。
```

```
if(elements is Iterable<E>){

//创建一个_GrowableList

     List<E> list =new_GrowableList<E>(0);

//遍历 elements 将每个元素重新加入到_GrowableList 中

for(E e in elements){

       list.add(e);

}

//如果是可变长的直接返回这个 list 即可

if(growable)return list;

//否则调用 makeListFixedLength 使得集合变为定长集合,实际上调用 native 层的 c++
实现

returnmakeListFixedLength(list);

}else{

     List<E> list =new_GrowableList<E>(0);

for(E e in elements){

       list.add(e);

}

if(growable)return list;

returnmakeListFixedLength(list);

}

}



//对应的是 List.unmodifiable 构造函数的实现
```

```
@patch

factory List.unmodifiable(Iterable elements){

final result =newList<E>.from(elements, growable:false);

//这里利用了 List.from 构造函数创建一个定长的集合 result

returnmakeFixedListUnmodifiable(result);

}

...}
```

对应的 `List.from` sdk 的源码解析

```
//sdk/lib/_internal/vm/lib/internal_patch.dart 中的 makeListFixedLength

@patchList<T> makeListFixedLength<T>(List<T> growableList)

 native "Internal_makeListFixedLength";

//runtime/lib/growable_array.cc 中的 Internal_makeListFixedLengthDEFINE_N
ATIVE_ENTRY(Internal_makeListFixedLength,0,1){

GET_NON_NULL_NATIVE_ARGUMENT(GrowableObjectArray,array,

 arguments->NativeArgAt(0));

returnArray::MakeFixedLength(array,/* unique = */true);//调用 Array::Make
FixedLength C++方法变为定长集合}

//runtime/vm/object.cc 中的 Array::MakeFixedLength 返回一个 RawArray

RawArray*Array::MakeFixedLength(const GrowableObjectArray& growable_arra
y, bool unique){



ASSERT(!growable_array.IsNull());

 Thread* thread = Thread::Current();
```

```
 Zone* zone = thread->zone();

 intptr_t used_len = growable_array.Length();

//拿到泛型类型参数，然后准备复制它们

const TypeArguments& type_arguments =

 TypeArguments::Handle(growable_array.GetTypeArguments());



//如果集合为空

if(used_len ==0){

//如果 type_arguments 是空，那么它就是一个原生 List，不带泛型类型参数的

if(type_arguments.IsNull()&&!unique){

//这是一个原生 List(没有泛型类型参数)集合并且是非 unique，直接返回空数组

return Object::empty_array().raw();

}

// 根据传入 List 的泛型类型参数，创建一个新的空的数组

    Heap::Space space = thread->IsMutatorThread()? Heap::kNew : Heap::kO
ld;//如果是 MutatorThread 就开辟新的内存空间否则复用旧的

Array&array=Array::Handle(zone,Array::New(0, space));//创建一个新的空数组
array

array.SetTypeArguments(type_arguments);//设置拿到的类型参数

returnarray.raw();//返回一个相同泛型参数的新数组

}



//如果集合不为空，取出 growable_array 中的 data 数组，且返回一个带数据新的数组 a
rray
```

```
constArray&array=Array::Handle(zone, growable_array.data());

ASSERT(array.IsArray());

array.SetTypeArguments(type_arguments);//设置拿到的类型参数

//这里主要是回收原来的 growable_array,数组长度置为 0，内部 data 数组置为空数组

    growable_array.SetLength(0);

    growable_array.SetData(Object::empty_array());

//注意：定长数组实现的关键点来了，会调用 Truncate 方法将 array 截断 used_len 长
度

array.Truncate(used_len);

returnarray.raw();//最后返回 array.raw()}
```

总结一下 `List.from` 的源码实现，首先传入 elements 的 `Iterate<E>`，如果 elements 不带泛型参数，也就是所谓的原生集合类型，并且是非 unique，直接返回空数组；如果带泛型参数空集合，那么会创建新的空集合并带上原来泛型参数返回；如果是带泛型参数非空集合，会取出其中 data 数组，来创建一个新的复制原来数据的集合并带上原来泛型参数返回，最后需要截断把数组截断成原始数组长度。

为什么需要 exteranl function

关键就是在于它能**实现声明和实现分离**，这样就**能复用同一套对外 API 的声明，然后对应多套多平台的实现**，如果对源码感兴趣的小伙伴就会发现相同 API 声明在 js 中也有另一套实现，这样不管是 dart for web 还是 dart for vm 对于上层开发而言都是一套 API，对于上层开发者是透明的。

二、Set

dart:core 包中的 **Set 集合**实际上是**委托到 dart:collection 中的 LinkedHashSet** 来实现的。集合 Set 和列表 List 的区别在于 **集合中的元素是不能重复** 的。所以添加重复的元素时会返回 false,表示添加不成功.

Set 初始化方式

```
main(){

   Set<String> colorSet={'red','yellow','blue','green'};//直接使用{}形式初始化

var colorList =<String>{'red','yellow','blue','green'};}
```

集合中的交、并、补集，在 Kotlin 并没有直接给到计算集合交、并、补的 API

```
main() {

   var colorSet1 = {'red', 'yellow', 'blue', 'green'};

   var colorSet2 = {'black', 'yellow', 'blue', 'green', 'white'};

   print(colorSet1.intersection(colorSet2));//交集-->输出: {'yellow', 'blue', 'green'}

   print(colorSet1.union(colorSet2));//并集--->输出: {'black', 'red', 'yellow', 'blue', 'green', 'white'}

   print(colorSet1.difference(colorSet2));//补集--->输出: {'red'}

}
```

Set 的遍历方式(和 List 一样)

```
main(){
```

```
    Set<String> colorSet ={'red','yellow','blue','green'};

//for-i 遍历

for(var i =0; i < colorSet.length; i++){

//可以使用 var 或 int

print(colorSet[i]);

}

//forEach 遍历

    colorSet.forEach((color)=>print(color));//forEach 的参数为 Function.
 =>使用了箭头函数

//for-in 遍历

for(var color in colorSet){

print(color);

}

//while+iterator 迭代器遍历，类似 Java 中的 iteator

while(colorSet.iterator.moveNext()){

print(colorSet.iterator.current);

}

}
```

构造函数源码分析

```
factorySet()= LinkedHashSet<E>;//主构造器委托到 LinkedHashSet 主构造器 fa
ctory Set.identity()= LinkedHashSet<E>.identity;//Set 的命名构造器 ident
ity 委托给 LinkedHashSet 的 identityfactory Set.from(Iterable elements)=
LinkedHashSet<E>.from;//Set 的命名构造器 from 委托给 LinkedHashSet 的 from
factory Set.of(Iterable<E> elements)= LinkedHashSet<E>.of;//Set 的命名
构造器 of 委托给 LinkedHashSet 的 of
```

对应 LinkedHashSet 的源码分析,篇幅有限感兴趣可以去深入研究

```
abstractclassLinkedHashSet<E>implementsSet<E>{

//LinkedHashSet 主构造器声明带了三个函数类型参数作为可选参数,同样是通过 ext
eranl 实现声明和实现分离，要深入可找到对应的@Patch 实现

externalfactoryLinkedHashSet(

{bool equals(E e1, E e2),

      int hashCode(E e),

      bool isValidKey(potentialKey)});



//LinkedHashSet 命名构造器 from

factory LinkedHashSet.from(Iterable elements){

//内部直接创建一个 LinkedHashSet 对象

    LinkedHashSet<E> result = LinkedHashSet<E>();

//并将传入 elements 元素遍历加入到 LinkedHashSet 中

for(final element in elements){

      result.add(element);

}

return result;

}



//LinkedHashSet 命名构造器 of，首先创建一个 LinkedHashSet 对象，通过级联操作
直接通过 addAll 方法将元素加入到 elements

factory LinkedHashSet.of(Iterable<E> elements)=>
```

```
     LinkedHashSet<E>()..addAll(elements);



voidforEach(voidaction(E element));



  Iterator<E>get iterator;}
```

对应的 `sdk/lib/_internal/vm/lib/collection_patch.dart` 中的`@Patch LinkedHashSet`

```
@patchclassLinkedHashSet<E>{

@patch

factoryLinkedHashSet(

{bool equals(E e1, E e2),

     int hashCode(E e),

     bool isValidKey(potentialKey)}){

if(isValidKey ==null){

if(hashCode ==null){

if(equals ==null){

returnnew_CompactLinkedHashSet<E>();//可选参数都为 null,默认创建_Compac
tLinkedHashSet

}

       hashCode = _defaultHashCode;

}else{

if(identical(identityHashCode, hashCode)&&

identical(identical, equals)){
```

```
returnnew_CompactLinkedIdentityHashSet<E>();//创建_CompactLinkedIdentityHashSet

}

       equals ??= _defaultEquals;

}

}else{

     hashCode ??= _defaultHashCode;

     equals ??= _defaultEquals;

}

returnnew_CompactLinkedCustomHashSet<E>(equals, hashCode, isValidKey);//可选参数 identical,默认创建_CompactLinkedCustomHashSet

}



@patch

factory LinkedHashSet.identity()=>new_CompactLinkedIdentityHashSet<E>();}
```

三、Map

dart:core 包中的 Map 集合 实际上是 委托到 dart:collection 中的 LinkedHashMap 来实现的。集合 Map 和 Kotlin 类似，key-value 形式存储，并且 Map 对象的中 key 是不能重复的

Map 初始化方式

```
main(){
```

```
    Map<String, int> colorMap ={'white':0xffffffff,'black':0xff000000};//使用{key:value}形式初始化

var colorMap =<String, int>{'white':0xffffffff,'black':0xff000000};

var colorMap = Map<String, int>();//创建一个空的 Map 集合

//实际上等价于下面代码，后面会通过源码说明

var colorMap = LinkedHashMap<String, int>();}
```

Map 中常用的函数

```
main(){

    Map<String, int> colorMap ={'white':0xffffffff,'black':0xff000000};

print(colorMap.containsKey('green'));//false

print(colorMap.containsValue(0xff000000));//true

print(colorMap.keys.toList());//['white','black']

print(colorMap.values.toList());//[0xffffffff, 0xff000000]

    colorMap['white']=0xfffff000;//修改指定 key 的元素

    colorMap.remove('black');//移除指定 key 的元素}
```

Map 的遍历方式

```
main(){

    Map<String, int> colorMap ={'white':0xffffffff,'black':0xff000000};

//for-each key-value

    colorMap.forEach((key, value)=>print('color is $key, color value is $value'));}
```

Map.fromIterables 将 List 集合转化成 Map

```
main(){

    List<String> colorKeys =['white','black'];

    List<int> colorValues =[0xffffffff,0xff000000];

    Map<String, int> colorMap = Map.fromIterables(colorKeys, colorValu
es);}
```

构造函数源码分析

```
externalfactoryMap();//主构造器交由外部@Patch 实现，实际上对应的@Patch 实
现还是委托给 LinkedHashMap

factory Map.from(Map other)= LinkedHashMap<K, V>.from;//Map 的命名构造
器 from 委托给 LinkedHashMap 的 from

factory Map.of(Map<K, V> other)= LinkedHashMap<K, V>.of;//Map 的命名构
造器 of 委托给 LinkedHashMap 的 of

externalfactory Map.unmodifiable(Map other);//unmodifiable 构造器交由外
部@Patch 实现

factory Map.identity()= LinkedHashMap<K, V>.identity;//Map 的命名构造器
identity 交由外部@Patch 实现

factory Map.fromIterable(Iterable iterable,

{K key(element), V value(element)})= LinkedHashMap<K, V>.fromIterable;
//Map 的命名构造器 fromIterable 委托给 LinkedHashMap 的 fromIterable

factory Map.fromIterables(Iterable<K> keys, Iterable<V> values)=

     LinkedHashMap<K, V>.fromIterables;//Map 的命名构造器 fromIterables
委托给 LinkedHashMap 的 fromIterables
```

对应 LinkedHashMap 构造函数源码分析

```
abstractclassLinkedHashMap<K, V>implementsMap<K, V>{

//主构造器交由外部@Patch 实现

externalfactoryLinkedHashMap(

{bool equals(K key1, K key2),

      int hashCode(K key),

      bool isValidKey(potentialKey)});



//LinkedHashMap 命名构造器 identity 交由外部@Patch 实现

externalfactory LinkedHashMap.identity();



//LinkedHashMap 的命名构造器 from

factory LinkedHashMap.from(Map other){

//创建一个新的 LinkedHashMap 对象

    LinkedHashMap<K, V> result = LinkedHashMap<K, V>();

//遍历 other 中的元素，并添加到新的 LinkedHashMap 对象

    other.forEach((k, v){

      result[k]= v;

});

return result;

}
```

```
//LinkedHashMap 的命名构造器 of,创建一个新的 LinkedHashMap 对象，通过级联操作符调用 addAll 批量添加 map 到新的 LinkedHashMap 中

factory LinkedHashMap.of(Map<K, V> other)=>

     LinkedHashMap<K, V>()..addAll(other);



//LinkedHashMap 的命名构造器 fromIterable，传入的参数是 iterable 对象、key 函数参数、value 函数参数两个可选参数

factory LinkedHashMap.fromIterable(Iterable iterable,

{K key(element), V value(element)}){

//创建新的 LinkedHashMap 对象，通过 MapBase 中的 static 方法_fillMapWithMappedIterable，给新的 map 添加元素

    LinkedHashMap<K, V> map = LinkedHashMap<K, V>();

    MapBase._fillMapWithMappedIterable(map, iterable, key, value);

return map;

}



//LinkedHashMap 的命名构造器 fromIterables

factory LinkedHashMap.fromIterables(Iterable<K> keys, Iterable<V> values){

//创建新的 LinkedHashMap 对象，通过 MapBase 中的 static 方法_fillMapWithIterables，给新的 map 添加元素

    LinkedHashMap<K, V> map = LinkedHashMap<K, V>();

    MapBase._fillMapWithIterables(map, keys, values);

return map;

}}
```

```
//MapBase 中的_fillMapWithMappedIterable

staticvoid_fillMapWithMappedIterable(

      Map map, Iterable iterable,key(element),value(element)){

    key ??= _id;

    value ??= _id;



for(var element in iterable){//遍历 iterable，给 map 对应复制

      map[key(element)]=value(element);

}

}

// MapBase 中的_fillMapWithIterables

staticvoid_fillMapWithIterables(Map map, Iterable keys, Iterable valu
es){

    Iterator keyIterator = keys.iterator;//拿到 keys 的 iterator

    Iterator valueIterator = values.iterator;//拿到 values 的 iterator



    bool hasNextKey = keyIterator.moveNext();//是否有 NextKey

    bool hasNextValue = valueIterator.moveNext();//是否有 NextValue



while(hasNextKey && hasNextValue){//同时遍历迭代 keys，values

      map[keyIterator.current]= valueIterator.current;

      hasNextKey = keyIterator.moveNext();

      hasNextValue = valueIterator.moveNext();
```

```
}



if(hasNextKey || hasNextValue){//最后如果其中只要有一个为 true,说明 key 与
value 的长度不一致，抛出异常

throwArgumentError("Iterables do not have same length.");

}

}
```

Map 的@Patch 对应实现，对应 `sdk/lib/_internal/vm/lib/map_patch.dart` 中

```
@patchclassMap<K, V>{

@patch

factory Map.unmodifiable(Map other){

returnnewUnmodifiableMapView<K, V>(newMap<K, V>.from(other));

}



@patch

factoryMap()=>newLinkedHashMap<K, V>();//可以看到 Map 的创建实际上最终还
是对应创建了 LinkedHashMap<K, V>}
```

四、Queue

Queue 队列顾名思义先进先出的一种数据结构，在 Dart 对队列也做了一定的支持，实际上 **Queue** 的实现是委托给 **ListQueue** 来实现。 Queue 继承于

`EfficientLengthIterable<E>`接口，然后 `EfficientLengthIterable<E>`接口又继承了 `Iterable<E>`.所以意味着 Queue 可以向 List 那样使用丰富的操作函数。并且由 Queue 派生出了 `DoubleLinkedQueue` 和 `ListQueue`

初始化

```
import 'dart:collection';//注意: Queue 位于 dart:collection 包中需要导包

main(){

//通过主构造器初始化

var queueColors =Queue();

  queueColors.addFirst('red');

  queueColors.addLast('yellow');

  queueColors.add('blue');

//通过 from 命名构造器初始化

var queueColors2 = Queue.from(['red','yellow','blue']);

//通过 of 命名构造器初始化

var queueColors3 = Queue.of(['red','yellow','blue']);}
```

常用的函数

```
import 'dart:collection';//注意: Queue 位于 dart:collection 包中需要导包 m
ain(){

var queueColors =Queue()

..addFirst('red')

..addLast('yellow')

..add('blue')
```

```
..addAll(['white','black'])

..remove('black')

..clear();}
```

遍历

```
import'dart:collection';//注意：Queue 位于 dart:collection 包中需要导包

main(){

  Queue<String> colorQueue = Queue.from(['red','yellow','blue','green
']);

//for-i 遍历

for(var i =0; i < colorQueue.length; i++){

//可以使用 var 或 int

print(colorQueue.elementAt(i));//注意：获取队列中的元素不用使用 colorQue
ue[i]，因为 Queue 内部并没有去实现[]运算符重载

}

//forEach 遍历

  colorQueue.forEach((color)=>print(color));//forEach 的参数为 Function.
 =>使用了箭头函数

//for-in 遍历

for(var color in colorQueue){

print(color);

}}
```

构造函数源码分析

```
factoryQueue()= ListQueue<E>;//委托给 ListQueue<E>主构造器
```

```
factory Queue.from(Iterable elements)= ListQueue<E>.from;//委托给 ListQueue<E>的命名构造器 from


factory Queue.of(Iterable<E> elements)= ListQueue<E>.of;//委托给 ListQueue<E>的命名构造器 of
```

对应的 ListQueue 的源码分析

```
classListQueue<E>extendsListIterable<E>implementsQueue<E>{

staticconst int _INITIAL_CAPACITY =8;//默认队列的初始化容量是 8

  List<E?> _table;

  int _head;

  int _tail;

  int _modificationCount =0;



ListQueue([int? initialCapacity])

: _head =0,

        _tail =0,

        _table = List<E?>(_calculateCapacity(initialCapacity));//有趣的是可以看到 ListQueque 内部实现是一个 List<E?>集合，E?还是一个泛型类型为可空类型，但是目前 dart 的可空类型特性还在实验中，不过可以看到它的源码中已经用起来了。



//计算队列所需要容量大小

static int _calculateCapacity(int? initialCapacity){
```

```
//如果 initialCapacity 为 null 或者指定的初始化容量小于默认的容量就是用默认的
容量大小

if(initialCapacity ==null|| initialCapacity < _INITIAL_CAPACITY){



return _INITIAL_CAPACITY;

}elseif(!_isPowerOf2(initialCapacity)){//容量大小不是 2 次幂

return_nextPowerOf2(initialCapacity);//找到大小是接近 number 的 2 次幂的数

}

assert(_isPowerOf2(initialCapacity));//断言检查

return initialCapacity;//最终返回 initialCapacity,返回的容量大小一定是 2
次幂的数

}



//判断容量大小是否是 2 次幂

static bool _isPowerOf2(int number)=>(number &(number -1))==0;



//找到大小是接近 number 的二次幂的数

static int _nextPowerOf2(int number){

assert(number >0);

    number =(number <<1)-1;

for(;;){

      int nextNumber = number &(number -1);

if(nextNumber ==0)return number;

      number = nextNumber;
```

```
}

}



//ListQueue 的命名构造函数 from

factory ListQueue.from(Iterable<dynamic> elements){

//判断 elements 是否是 List<dynamic>类型

if(elements is List<dynamic>){

      int length = elements.length;//取出长度

      ListQueue<E> queue = ListQueue<E>(length +1);//创建 length + 1 长
度的 ListQueue

assert(queue._table.length > length);//必须保证新创建的 queue 的长度大于
传入 elements 的长度

for(int i =0; i < length; i++){

        queue._table[i]= elements[i]as E;//然后就是给新 queue 中的元素赋
值，注意需要强转成泛型类型 E

}

      queue._tail = length;//最终移动队列的 tail 尾部下标，因为可能存在实际
长度大于实际元素长度

return queue;

}else{

      int capacity = _INITIAL_CAPACITY;

if(elements is EfficientLengthIterable){//如果是 EfficientLengthIterab
le 类型，就将 elements 长度作为初始容量不是就使用默认容量

        capacity = elements.length;

}
```

```
      ListQueue<E> result = ListQueue<E>(capacity);

for(final element in elements){

        result.addLast(element as E);//通过 addLast 从队列尾部插入

}

return result;//最终返回 result

}

}



//ListQueue 的命名构造函数 of

factory ListQueue.of(Iterable<E> elements)=>

      ListQueue<E>()..addAll(elements);//直接创建 ListQueue<E>()并通过 a
ddAll 把 elements 加入到新的 ListQueue 中

...}
```

五、LinkedList

在 dart 中 LinkedList 比较特殊，它不是一个带泛型集合，因为它泛型类型上界是 `LinkedListEntry`，内部的数据结构实现是一个双链表，链表的结点是 `LinkedListEntry` 的子类，且内部维护了 `_next` 和 `_previous` 指针。此外它并没有实现 List 接口

初始化

```
import'dart:collection';//注意: LinkedList 位于 dart:collection 包中需要
导包 main(){

var linkedList = LinkedList<LinkedListEntryImpl<int>>();

var prevLinkedEntry = LinkedListEntryImpl<int>(99);
```

```
var currentLinkedEntry = LinkedListEntryImpl<int>(100);

var nextLinkedEntry = LinkedListEntryImpl<int>(101);

  linkedList.add(currentLinkedEntry);

  currentLinkedEntry.insertBefore(prevLinkedEntry);//在当前结点前插入一个新的结点

  currentLinkedEntry.insertAfter(nextLinkedEntry);//在当前结点后插入一个新的结点

  linkedList.forEach((entry)=>print('${entry.value}'));}

//需要定义一个 LinkedListEntry 子类 classLinkedListEntryImpl<T>extendsLinkedListEntry<LinkedListEntryImpl<T>>{

final T value;



LinkedListEntryImpl(this.value);



@override

  String toString(){

return"value is $value";

}}
```

常用的函数

```
currentLinkedEntry.insertBefore(prevLinkedEntry);//在当前结点前插入一个新的结点

currentLinkedEntry.insertAfter(nextLinkedEntry);//在当前结点后插入一个新的结点

currentLinkedEntry.previous;//获取当前结点的前一个结点
```

```
currentLinkedEntry.next;//获取当前结点的后一个结点

currentLinkedEntry.list;//获取 LinkedList

currentLinkedEntry.unlink();//把当前结点 entry 从 LinkedList 中删掉
```

遍历

```
//forEach 迭代

 linkedList.forEach((entry)=>print('${entry.value}'));

//for-i 迭代

for(var i =0; i < linkedList.length; i++){

print('${linkedList.elementAt(i).value}');

}

//for-in 迭代

for(var element in linkedList){

print('${element.value}');

}
```

六、HashMap

初始化

```
import'dart:collection';//注意: HashMap 位于 dart:collection 包中需要导包
main(){

var hashMap =HashMap();//通过 HashMap 主构造器初始化

  hashMap['a']=1;

  hashMap['b']=2;
```

```
  hashMap['c']=3;

var hashMap2 = HashMap.from(hashMap);//通过 HashMap 命名构造器 from 初始化

var hashMap3 = HashMap.of(hashMap);//通过 HashMap 命名构造器 of 初始化

var keys =['a','b','c'];

var values =[1,2,3]

var hashMap4 = HashMap.fromIterables(keys, values);//通过 HashMap 命名构
造器 fromIterables 初始化



  hashMap2.forEach((key, value)=>print('key: $key  value: $value'));}
```

常用的函数

```
import'dart:collection';//注意: HashMap 位于 dart:collection 包中需要导包
main(){

var hashMap =HashMap();//通过 HashMap 主构造器初始化

   hashMap['a']=1;

   hashMap['b']=2;

   hashMap['c']=3;

print(hashMap.containsKey('a'));//false

print(hashMap.containsValue(2));//true

print(hashMap.keys.toList());//['a','b','c']

print(hashMap.values.toList());//[1, 2, 3]

   hashMap['a']=55;//修改指定 key 的元素

   hashMap.remove('b');//移除指定 key 的元素}
```

遍历

```
import'dart:collection';//注意: HashMap 位于 dart:collection 包中需要导包
main(){

var hashMap =HashMap();//通过 HashMap 主构造器初始化

   hashMap['a']=1;

   hashMap['b']=2;

   hashMap['c']=3;

//for-each key-value

   hashMap.forEach((key, value)=>print('key is $key, value is $value
'));}
```

构造函数源码分析

```
//主构造器交由外部@Patch 实现 externalfactoryHashMap(

{bool equals(K key1, K key2),

     int hashCode(K key),

     bool isValidKey(potentialKey)});

//HashMap 命名构造器 identity 交由外部@Patch 实现 externalfactory HashMap.
identity();

//HashMap 命名构造器 fromfactory HashMap.from(Map other){

//创建一个 HashMap 对象

    Map<K, V> result = HashMap<K, V>();

//遍历 other 集合并把元素赋值给新的 HashMap 对象
```

```
    other.forEach((k, v){

      result[k]= v;

});

return result;

}

//HashMap 命名构造器 of，把 other 添加到新创建 HashMap 对象 factory HashMap.
of(Map<K, V> other)=> HashMap<K, V>()..addAll(other);

//HashMap 命名构造器 fromIterablefactory HashMap.fromIterable(Iterable
iterable,

{K key(element), V value(element)}){

    Map<K, V> map = HashMap<K, V>();//创建一个新的 HashMap 对象

    MapBase._fillMapWithMappedIterable(map, iterable, key, value);//通
过 MapBase 中的_fillMapWithMappedIterable 赋值给新的 HashMap 对象

return map;}

//HashMap 命名构造器 fromIterablesfactory HashMap.fromIterables(Iterabl
e<K> keys, Iterable<V> values){

    Map<K, V> map = HashMap<K, V>();//创建一个新的 HashMap 对象

    MapBase._fillMapWithIterables(map, keys, values);//通过 MapBase 中的
_fillMapWithIterables 赋值给新的 HashMap 对象

return map;}
```

HashMap 对应的@Patch 源码实

现,`sdk/lib/_internal/vm/lib/collection_patch.dart`

```
@patchclassHashMap<K, V>{

@patch
```

```
factoryHashMap(

{bool equals(K key1, K key2),

      int hashCode(K key),

      bool isValidKey(potentialKey)}){

if(isValidKey ==null){

if(hashCode ==null){

if(equals ==null){

returnnew_HashMap<K, V>();//创建私有的_HashMap 对象

}

        hashCode = _defaultHashCode;

}else{

if(identical(identityHashCode, hashCode)&&

identical(identical, equals)){

returnnew_IdentityHashMap<K, V>();//创建私有的_IdentityHashMap 对象

}

        equals ??= _defaultEquals;

}

}else{

      hashCode ??= _defaultHashCode;

      equals ??= _defaultEquals;

}

returnnew_CustomHashMap<K, V>(equals, hashCode, isValidKey);//创建私有
的_CustomHashMap 对象
```

```
}


@patch
factory HashMap.identity()=>new_IdentityHashMap<K, V>();


  Set<K>_newKeySet();}
```

七、Map、HashMap、LinkedHashMap、SplayTreeMap 区别

在 Dart 中还有一个 SplayTreeMap，它的初始化、常用的函数和遍历方式和 LinkedHashMap、HashMap 使用类似。但是 Map、HashMap、LinkedHashMap、SplayTreeMap 有什么区别呢。

`Map`

Map 是 key-value 键值对集合。在 Dart 中的 Map 中的每个条目都可以迭代的。迭代顺序取决于 HashMap，LinkedHashMap 或 SplayTreeMap 的实现。如果您使用 Map 构造函数创建实例，则**默认情况下会创建一个 LinkedHashMap**。

`HashMap`

HashMap 不保证插入顺序。如果先插入 key 为 A 的元素，然后再插入具有 key 为 B 的另一个元素，则在遍历 `Map` 时，有可能先获得元素 B。

`LinkedHashMap`

LinkedHashMap 保证插入顺序。根据插入顺序对存储在 LinkedHashMap 中的数据进行排序。如果先插入 key 为 A 的元素，然后再插入具有 key 为 B 的另一个元素，则在遍历 `Map` 时，总是先取的 key 为 A 的元素，然后再取的 key 为 B 的元素。

`SplayTreeMap`

SplayTreeMap 是一个自平衡二叉树，它允许更快地访问最近访问的元素。基本操作如插入，查找和删除可以在 O（log（n））时间复杂度中完成。它通过使经常访问的元素靠近树的根来执行树的旋转。因此，如果需要更频繁地访问某些元素，则使用 SplayTreeMap 是一个不错的选择。但是，如果所有元素的数据访问频率几乎相同，则使用 SplayTreeMap 是没有用的。

八、命名构造函数 from 和 of 的区别以及使用建议

通过上述各个集合源码可以看到，基本上每个集合(List、Set、LinkedHashSet、LinkedHashMap、Map、HashMap 等)中都有 from 和 of 命名构造函数。可能有的人有疑问了，它们有什么区别，各自的应用场景呢。其实答案从源码中就看出一点了。以 List,Map 中的 from 和 of 为例。

```
main(){

var map ={'a':1,'b':2,'c':3};

var fromMap = Map.from(map);//返回类型是 Map<dynamic, dynamic>

var ofMap = Map.of(map);//返回类型是 Map<String, int>



varlist=[1,2,3,4];
```

```
var fromList =List.from(list);//返回类型是 List<dynamic>

var ofList =List.of(list);//返回类型是 List<int>}
```

从上述例子可以看出 List、Map 中的 from 函数返回对应的集合泛型类型是 `List<dynamic>` 和 `Map<dynamic, dynamic>` 而 of 函数返回对应集合泛型类型实际类型是 `List<int>` 和 `Map<String, int>`。我们都知道 dynamic 是一种无法确定的类型，在编译期不检查类型，只在运行器检查类型，而具体类型是在编译期检查类型。而且从源码中可以看到 from 函数往往会处理比较复杂逻辑比如需要重新遍历传入的集合然后把元素加入到新的集合中，而 of 函数只需要创建一个新的对象通过 addAll 函数批量添加传入的集合元素。

所以这里为了代码效率考虑给出建议是: **如果你传入的原有集合元素类型是确定的，请尽量使用 of 函数创建新的集合，否则就可以考虑使用 from 函数。**

总结

到这里我们 dart 语法系列第二篇就结束了，相信通过这篇文章大家对 dart 中的集合应该有了全面的了解，下面我们将继续研究 dart 和 Flutter 相关内容。

第七章 Dart 语法篇之集合操作符函数与源码分析(三)

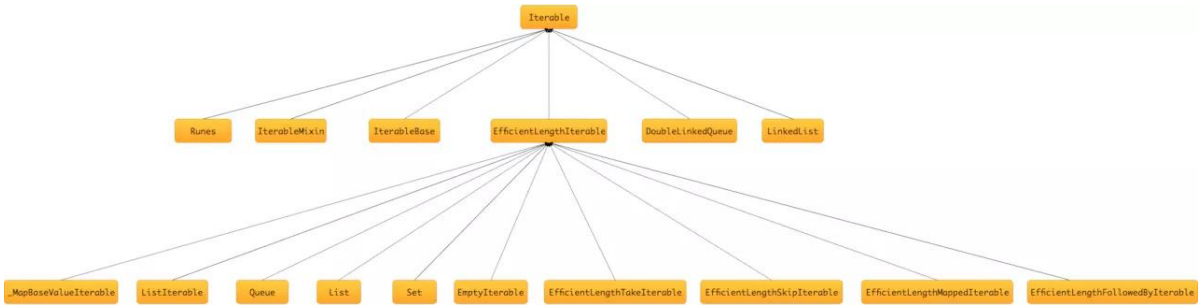
简述：

在上一篇文章中，我们全面地分析了常用集合的使用以及集合部分源码的分析。那么这一节讲点更实用的内容，绝对可以提高你的 Flutter 开发效率的函数，那就是集合中常用的操作符函数。这次说的内容的比较简单就是怎么用，以及源码内部是怎么实现的。

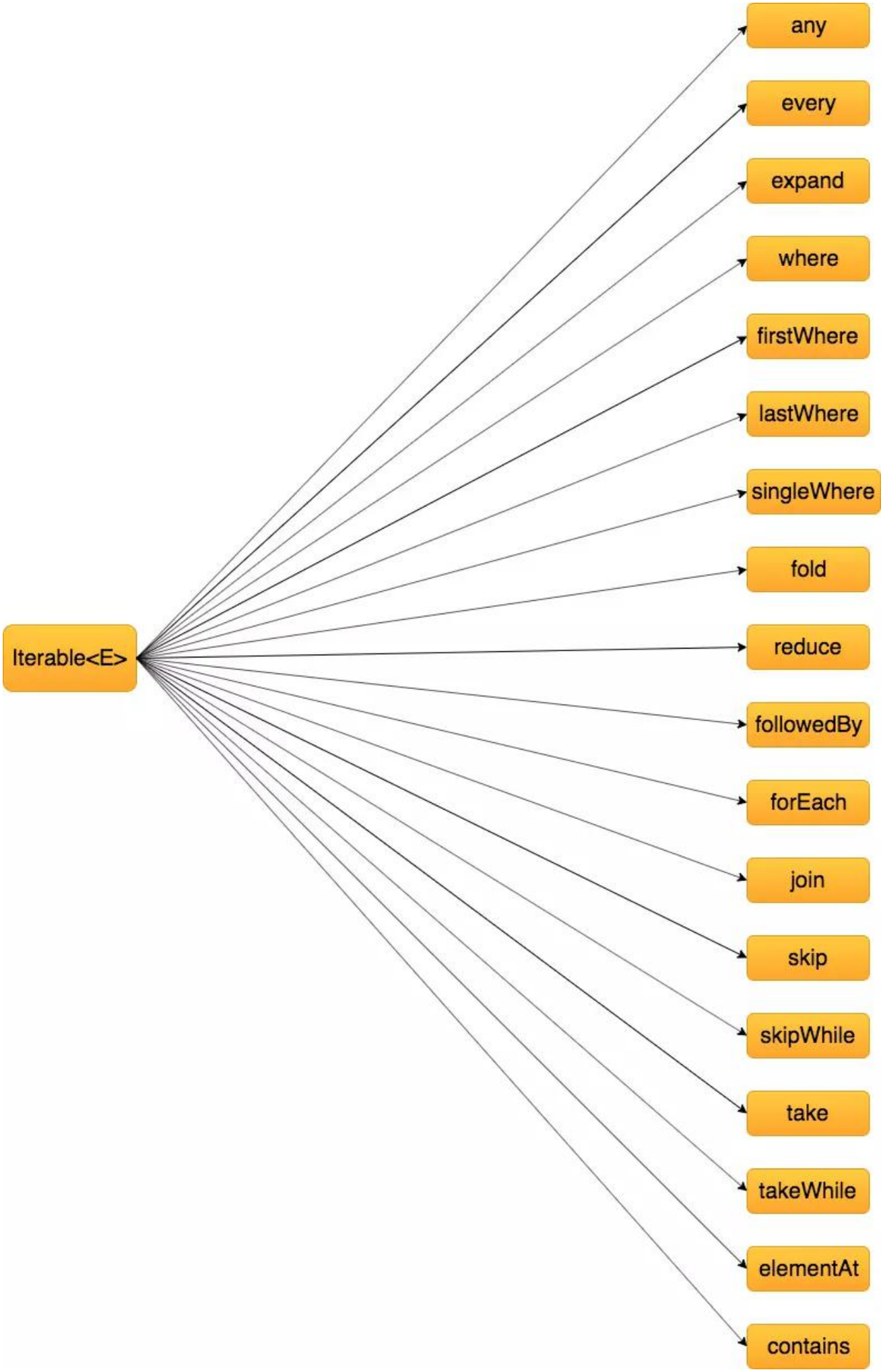
一、Iterable<E>

在 dart 中几乎所有集合拥有的操作符函数(例如: map、every、where、reduce 等)都是因为继承或者实现了 `Iterable`。

1、Iterable 类关系图



2、Iterable 类方法图


Iterable<E>
any
every
expand
where
firstWhere
lastWhere
singleWhere
fold
reduce
followedBy
forEach
join
skip
skipWhile
take
takeWhile
elementAt
contains

二、forEach

1、介绍

```
voidforEach(voidf(E element))
```

forEach 在 dart 中用于**遍历和迭代集合**，也是 dart 中操作集合最常用的方法之一。接收一个 `f(E element)`函数作为参数，返回值类型为空 void.

2、使用方式

```
main(){

var languages =<String>['Dart','Kotlin','Java','Javascript','Go','Python','Swift'];

  languages.forEach((language)=>print('The language is $language'));//由于只有一个表达式，所以可以直接使用箭头函数。

  languages.forEach((language){

if(language =='Dart'|| language =='Kotlin'){

print('My favorite language is $language');

}

});}
```

3、源码解析

```
voidforEach(voidf(E element)){
```

```
//可以看到在 forEach 内部实际上就是利用 for-in 迭代，每迭代一次就执行一次 f 函数，

//并把当前 element 回调出去

for(E element in this)f(element);

}
```

三、map

1、介绍

```
Iterable<T> map<T>(T f(E e))
```

map 函数主要用于**集合中元素的映射**，也可以映射转化成其他类型的元素。可以看到 map 接收一个 `T f(E e)`函数作为参数，最后返回一个泛型参数为 `T` 的 `Iterable`。实际上是返回了带有元素的一个新的**惰性** `Iterable`，然后通过迭代的时候，对每个元素都调用 `f` 函数。注意: `f` 函数是一个接收泛型参数为 `E` 的元素,然后返回一个泛型参数为 `T` 的元素，这就是 map 可以将原集合中每个元素映射成其他类型元素的原因。

2、使用方式

```
main(){

var languages =<String>['Dart','Kotlin','Java','Javascript','Go','Python
','Swift'];

print(languages.map((language)=>'develop language is ${language}').join
('---'));}
```

3、源码解析

以上面的例子为例，

1、首先，需要明确一点，`languages` 内部本质是一个`_GrowableList<T>`，我们都知道`_GrowableList<T>`是继承了 `ListBase<T>`,然后 `ListBase<E>`又 mixin with `ListMixin<E>`.所以 `languages.map` 函数调用就是调用 `ListMixin<E>`中的 map 函数，实际上还是相当于调用了自身的成员函数 map.


```
@pragma("vm:entry-point")class_GrowableList<T> extends ListBase<T>{//_Gr
owableList<T>是继承了 ListBase<T>

...}



abstract classListBase<E> extends Object with ListMixin<E>{//ListBase mi
xin with ListMixin<E>

...}
```

2、然后可以看到 `ListMixin<E>`实际上实现了 `List<E>`,然后 `List<E>`继承了`EfficientLengthIterable<E>`，最后 `EfficientLengthIterable<E>`继承 `Iterable<E>`，所以最终的 map 函数来自于 `Iterable<E>`但是具体的实现定义在 `ListMinxin<E>`中。

```
abstractclassListMixin<E>implementsList<E>{

...
```

```
//可以看到这里是直接返回一个 MappedListIterable，它是一个惰性 Iterable

    Iterable<T> map<T>(T f(E element))=> MappedListIterable<E, T>(this,
 f);

...}
```

3、为什么是惰性的呢，**可以看到它并不是直接返回转化后的集合，而是返回一个带有值的 MappedListIterable 的，如果不执行 `elementAt` 方法，是不会触发执行 map 传入的 `f` 函数**，所以它是惰性的。

```
classMappedListIterable<S, T>extendsListIterable<T>{

final Iterable<S> _source;//_source 存储了所携带的原集合

final _Transformation<S, T> _f;//_f 函数存储了 map 函数传入的闭包，



MappedListIterable(this._source,this._f);



  int get length => _source.length;

//注意：只有 elementAt 函数执行的时候，才会触发执行_f 方法，然后通过_source 的 e
lementAt 函数取得原集合中的元素，

//最后针对_source 中的每个元素执行_f 函数处理。

  T elementAt(int index)=>_f(_source.elementAt(index));}
```

4、一般不会单独使用 map 函数，因为单独使用 map 的函数时，仅仅返回的是惰性的 `MappedListIterable`。由上面的源码可知，仅仅在 elementAt 调用的时候才

会触发 map 中的闭包。所以我们一般使用完 map 后会配合 `toList()、toSet()`函数或者触发 `elementAt` 函数的函数(例如这里的 `join`)一起使用。

```
languages.map((language)=>'develop language is ${language}').toList();//toList()方法调用才会真正去执行 map 中的闭包。



languages.map((language)=>'develop language is ${language}').toSet();//toSet()方法调用才会真正去执行 map 中的闭包。



languages.map((language)=>'develop language is ${language}').join('---');//join()方法调用才会真正去执行 map 中的闭包。



  List<E>toList({bool growable =true}){

    List<E> result;

if(growable){

      result =<E>[]..length = length;

}else{

      result = List<E>(length);

}

for(int i =0; i < length; i++){

      result[i]=this[i];//注意：这里的 this[i]实际上是运算符重载了[]，最终就是调用了 elementAt 函数，这里才会真正的触发 map 中的闭包，

}

return result;

}
```

四、any

1、介绍

```
boolany(booltest(E element))
```

any 函数主要用于**检查是否存在任意一个满足条件的元素，只要匹配到第一个就返回 true，如果遍历所有元素都不符合才返回 false**. any 函数接收一个 `bool test(E element)` 函数作为参数，`test` 函数回调一个 `E` 类型的 `element` 并返回一个 `bool` 类型的值。

2、使用方式

```
main(){

    bool isDartExisted = languages.any((language)=> language =='Dart');}
```

3、源码解析

```
  bool any(bool test(E element)){

    int length =this.length;//获取到原集合的 length

//遍历原集合，只要找到符合 test 函数的条件，就返回 true

for(int i =0; i < length; i++){

if(test(this[i]))returntrue;

if(length !=this.length){

throwConcurrentModificationError(this);
```

```
}

}

//遍历完集合后，未找到符合条件的集合就返回 false

returnfalse;

}
```

五、every

1、介绍

```
boolevery(booltest(E element))
```

every 函数主要用于**检查是否集合所有元素都满足条件，如果都满足就返回 true，只要存在一个不满足条件的就返回 false.** every 函数接收一个 `bool test(E element)` 函数作为参数，`test` 函数回调一个 `E` 类型的 `element` 并返回一个 `bool` 类型的值。

2、使用方式

```
main(){

    bool isDartAll = languages.every((language)=> language =='Dart');}
```

3、源码解析

```
boolevery(booltest(E element)){

//利用 for-in 遍历集合，只要找到不符合 test 函数的条件，就返回 false.

for(E element in this){
```

```
if(!test(element))returnfalse;

}//遍历完集合后，找到所有元素符合条件就返回 true

returntrue;

}
```

六、where

1、介绍

```
Iterable<E> where(bool test(E element))
```

where 函数主要用于**过滤符合条件的元素，类似 Kotlin 中的 filter 的作用，最后返回符合条件元素的集合。** where 函数接收一个 `bool test(E element)`函数作为参数，最后返回一个泛型参数为 `E` 的 `Iterable`。类似 map 一样，where 这里也是返回一个惰性的 `Iterable<E>`，然后对它的 `iterator` 进行迭代，对每个元素都执行 `test` 方法。

2、使用方式

```
main(){

   List<int> numbers =[0,3,1,2,7,12,2,4];

print(numbers.where((num)=> num >6));//输出：(7,12)

//注意：这里是 print 的内容实际上输出的是 Iterable 的 toString 方法返回的内容。}
```

3、源码解析

1、首先，需要明确一点 `numbers` 实际上是一个 `_GrowableList<T>` 和 `map` 的分析原理类似，最终还是调用了 `ListMixin` 中的 `where` 函数。

```
//可以看到这里是直接返回一个 WhereIterable 对象，而不是返回过滤后元素集合，所以它返回的 Iterable 也是惰性的。

Iterable<E>where(bool test(E element))=> WhereIterable<E>(this, test);
```

2、然后，继续深入研究下 `WhereIterable` 是如何实现的

```
classWhereIterable<E>extendsIterable<E>{

final Iterable<E> _iterable;//传入的原集合

final _ElementPredicate<E> _f;//传入的 where 函数中闭包参数



WhereIterable(this._iterable,this._f);



//注意：这里 WhereIterable 的迭代借助了 iterator，这里是直接创建一个 WhereIterator，并传入元集合_iterable 中的 iterator 以及过滤操作函数。

  Iterator<E>get iterator =>newWhereIterator<E>(_iterable.iterator, _f);



// Specialization of [Iterable.map] to non-EfficientLengthIterable.

  Iterable<T> map<T>(T f(E element))=>newMappedIterable<E, T>._(this, f);}
```

- 3、然后，继续深入研究下 `WhereIterator` 是如何实现的

```
classWhereIterator<E>extendsIterator<E>{
```

```
final Iterator<E> _iterator;//存储集合中的 iterator 对象

final _ElementPredicate<E> _f;//存储 where 函数传入闭包函数



WhereIterator(this._iterator,this._f);



//重写 moveNext 函数

  bool moveNext(){

//遍历原集合的_iterator

while(_iterator.moveNext()){

//注意：这里会执行_f 函数，如果满足条件就会返回 true，不符合条件的直接略过，迭代下一个元素；

//那么外部迭代时候，就可以通过 current 获得当前元素，这样就实现了在原集合基础上过滤拿到符合条件的元素。

if(_f(_iterator.current)){

returntrue;

}

}

//迭代完_iterator 所有元素后返回 false,以此来终止外部迭代。

returnfalse;

}

//重写 current 的属性方法

  E get current => _iterator.current;}
```

4、一般在使用的 `WhereIterator` 的时候，外部肯定还有一层 `while` 迭代，但是这个 `WhereIterator` 比较特殊，`moveNext()`的返回值由 `where` 中闭包函数参数返回值决定的，符合条件元素 `moveNext()`就返回 true,不符合就略过，迭代检查下一个元素，直至整个集合迭代完毕，`moveNext()`返回 false,以此也就终止了外部的迭代循环。

5、上面分析，`WhereIterable` 是惰性的，那它啥时候触发呢？没错就是在迭代它的 `iterator` 时候才会触发，以上面例子为例

```
print(numbers.where((num)=> num >6));//输出：(7,12)，最后会调用 Iterable 的
toString 方法返回的内容。

//看下 Iterable 的 toString 方法实现

String toString()=> IterableBase.iterableToShortString(this,'(',')');//
这就是为啥输出样式是 (7,12)//继续查看 IterableBase.iterableToShortString

static String iterableToShortString(Iterable iterable,

[String leftDelimiter ='(', String rightDelimiter =')']){

if(_isToStringVisiting(iterable)){

if(leftDelimiter =="("&& rightDelimiter ==")"){

// Avoid creating a new string in the "common" case.

return"(...)";

}

return"$leftDelimiter...$rightDelimiter";

}

    List<String> parts =<String>[];

    _toStringVisiting.add(iterable);

try{
```

```
_iterablePartsToStrings(iterable, parts);//注意:这里实际上就是通过将 itera
ble 转化成 List,内部就是通过迭代 iterator,以此会触发 WhereIterator 中的_f 函数。

}finally{

assert(identical(_toStringVisiting.last, iterable));

      _toStringVisiting.removeLast();

}

return(StringBuffer(leftDelimiter)

..writeAll(parts,", ")

..write(rightDelimiter))

.toString();

}



/// Convert elements of [iterable] to strings and store them in [parts].
这个函数代码实现比较多，这里给出部分代码 void_iterablePartsToStrings(Iterabl
e iterable, List<String> parts){

...

  int length =0;

  int count =0;

  Iterator it = iterable.iterator;

// Initial run of elements, at least headCount, and then continue until

// passing at most lengthLimit characters.

//可以看到这是外部迭代 while

while(length < lengthLimit || count < headCount){

if(!it.moveNext())return;//这里实际上调用了 WhereIterator 中的 moveNext 函数,
经过_f 函数处理的 moveNext()
```

```
    String next ="${it.current}";//获取 current.

    parts.add(next);

    length += next.length + overhead;

    count++;

}

...}
```

七、firstWhere 和 singleWhere 和 lastWhere

1、介绍

```
EfirstWhere(booltest(E element),{EorElse()})ElastWhere(booltest(E elemen
t),{EorElse()})EsingleWhere(booltest(E element),{EorElse()})
```

首先从源码声明结构上来看，firstWhere、lastWhere 和 singleWhere 是一样，它们都是接收两个参数，一个是必需参数:`test` 筛选条件闭包函数，另一个是可选参数:`orElse` 闭包函数。

但是它们用法却不同，firstWhere 主要是**用于筛选顺序第一个符合条件的元素，可能存在多个符合条件元素**；lastWhere 主要是**用于筛选顺序最后一个符合条件的元素，可能存在多个符合条件元素**；singleWhere 主要是**用于筛选顺序唯一一个符合条件的元素，不可能存在多个符合条件元素，存在的话就会抛出异常 IterableElementError.tooMany()，** 所以使用它的使用需要谨慎注意下

2、使用方式

```
main(){

var numbers =<int>[0,3,1,2,7,12,2,4];

//注意：如果没有找到，执行 orElse 代码块，可返回一个指定的默认值-1

print(numbers.firstWhere((num)=> num ==5, orElse:()=>-1));

//注意：如果没有找到，执行 orElse 代码块，可返回一个指定的默认值-1

print(numbers.lastWhere((num)=> num ==2, orElse:()=>-1));

//注意：如果没有找到，执行 orElse 代码块，可返回一个指定的默认值，前提是集合中只有一个符合条件的元素，否则就会抛出异常

print(numbers.singleWhere((num)=> num ==4, orElse:()=>-1));}
```

3、源码解析

```
//firstWhere

EfirstWhere(booltest(E element),{EorElse()}){

for(E element inthis){//直接遍历原集合，只要找到第一个符合条件的元素就直接返回，终止函数

if(test(element))return element;

}

if(orElse !=null)returnorElse();//遍历完集合后，都没找到符合条件的元素并且外部传入了 orElse 就会触发 orElse 函数

//否则找不到元素，直接抛出异常。所以这里需要注意下，如果不想抛出异常，可能你需要处理下 orElse 函数。

throw IterableElementError.noElement();

}
```

```
//lastWhere

ElastWhere(booltest(E element),{EorElse()}){

E result;//定义 result 来记录每次符合条件的元素

bool foundMatching =false;//定义一个标志位是否找到符合匹配的。

for(E element inthis){

if(test(element)){//每次找到符合条件的元素，都会重置 result，所以 result 记录了最新的符合条件元素，那么遍历到最后，它也就是最后一个符合条件的元素

        result = element;

        foundMatching =true;//找到后重置标记位

}

}

if(foundMatching)return result;//如果标记位为 true 直接返回 result 即可

if(orElse !=null)returnorElse();//处理 orElse 函数

//同样，找不到元素，直接抛出异常。可能你需要处理下 orElse 函数。

throw IterableElementError.noElement();

}



//singleWhere

EsingleWhere(booltest(E element),{EorElse()}){

E result;

bool foundMatching =false;

for(E element inthis){

if(test(element)){
```

```
if(foundMatching){//主要注意这里，只要 foundMatching 为 true,说明已经找到一个符合条件的元素，如果触发这条逻辑分支，说明不止一个元素符合条件就直接抛出 IterableElementError.tooMany()异常

throw IterableElementError.tooMany();

}

        result = element;

        foundMatching =true;

}

}

if(foundMatching)return result;

if(orElse !=null)returnorElse();

//同样，找不到元素，直接抛出异常。可能你需要处理下 orElse 函数。

throw IterableElementError.noElement();

}
```

八、join

1、介绍

```
 String join([String separator =""])

复制代码
```

join 函数主要是用于**将集合所有元素值转化成字符串，中间用指定的** `separator` **连接符连接**。 可以看到 join 函数比较简单，接收一个 `separator` 分隔符的可选参数，可选参数默认值是空字符串，最后返回一个字符串。

2、使用方式

```
main(){

var numbers =<int>[0,3,1,2,7,12,2,4];

print(numbers.join('-'));//输出: 0-3-1-2-7-12-2-4}
```

3、源码解析

```
//接收 separator 可选参数，默认值为""

  String join([String separator =""]){

    Iterator<E> iterator =this.iterator;

if(!iterator.moveNext())return"";

//创建 StringBuffer

    StringBuffer buffer =StringBuffer();

//如果分隔符为空或空字符串

if(separator ==null|| separator ==""){

//do-while 遍历 iterator,然后直接拼接元素

do{

        buffer.write("${iterator.current}");

}while(iterator.moveNext());

}else{

//如果分隔符不为空

//先加入第一个元素

     buffer.write("${iterator.current}");
```

```
//然后 while 遍历 iterator

while(iterator.moveNext()){

        buffer.write(separator);//先拼接分隔符

        buffer.write("${iterator.current}");//再拼接元素

}

}

return buffer.toString();//最后返回最终的字符串。

}
```

九、take

1、介绍

```
  Iterable<E>take(int count)

复制代码
```

take 函数主要是用于**截取原集合前 count 个元素组成的集合**，take 函数接收一个 `count` 作为函数参数，最后返回一个泛型参数为 `E` 的 `Iterable`。类似 where 一样，take 这里也是返回一个惰性的 `Iterable<E>`，然后对它的 `iterator` 进行迭代。

takeWhile 函数主要用于

2、使用方式

```
main(){

   List<int> numbers =[0,3,1,2,7,12,2,4];
```

```
print(numbers.take(5));//输出(0, 3, 1, 2, 7)}
```

3、源码解析

1、首先，需要明确一点 `numbers.take` 调用了 `ListMixin` 中的 `take` 函数，可以看到并没有直接返回集合前 `count` 个元素，而是返回一个 `TakeIterable<E>` 惰性 `Iterable`。

```
  Iterable<E>take(int count){

returnTakeIterable<E>(this, count);

}
```

2、然后，继续深入研究 `TakeIterable`

```
classTakeIterable<E>extendsIterable<E>{

final Iterable<E> _iterable;//存储原集合

final int _takeCount;//take count



factoryTakeIterable(Iterable<E> iterable, int takeCount){

    ArgumentError.checkNotNull(takeCount,"takeCount");

    RangeError.checkNotNegative(takeCount,"takeCount");

if(iterable is EfficientLengthIterable){//如果原集合是 EfficientLengthIte
rable，就返回创建 EfficientLengthTakeIterable

returnnewEfficientLengthTakeIterable<E>(iterable, takeCount);

}

//否则就返回 TakeIterable
```

```
returnnewTakeIterable<E>._(iterable, takeCount);

}


  TakeIterable._(this._iterable,this._takeCount);

//注意：这里是返回了 TakeIterator，并传入原集合的 iterator 以及_takeCount

  Iterator<E>get iterator {

returnnewTakeIterator<E>(_iterable.iterator, _takeCount);

}}
```

3、然后，继续深入研究 `TakeIterator`.

```
classTakeIterator<E>extendsIterator<E>{

final Iterator<E> _iterator;//存储原集合中的 iterator

  int _remaining;//存储需要截取的前几个元素的数量


TakeIterator(this._iterator,this._remaining){

assert(_remaining >=0);

}


  bool moveNext(){

    _remaining--;//通过_remaining 作为游标控制迭代次数

if(_remaining >=0){//如果_remaining 大于等于 0 就会继续执行 moveNext 方法

return _iterator.moveNext();
```

```
}

    _remaining =-1;

returnfalse;//如果_remaining 小于 0 就返回 false,终止外部循环

}



  E get current {

if(_remaining <0)returnnull;

return _iterator.current;

}}
```

4、所以上述例子中最终还是调用 `Iterable` 的 `toString` 方法，方法中会进行 `iterator` 的迭代，最终会触发惰性 `TakeIterable` 中的 `TakeIterator` 的 `moveNext` 方法。

十、takeWhile

1、介绍

```
  Iterable<E>takeWhile(booltest(Evalue))
```

takeWhile 函数主要用于**依次选择满足条件的元素，直到遇到第一个不满足的元素，并停止选择**。takeWhile 函数接收一个 `test` 条件函数作为函数参数，然后返回一个惰性的 `Iterable<E>`。

2、使用方式

```
main(){
```

```
  List<int> numbers =[3,1,2,7,12,2,4];

print(numbers.takeWhile((number)=> number >2).toList());//输出: [3] 遇到1 的时候就不满足大于 2 条件就终止筛选。}
```

3、源码解析

1、首先，因为 `numbers` 是 `List<int>`所以还是调用 `ListMixin` 中的 `takeWhile` 函数

```
  Iterable<E>takeWhile(booltest(E element)){

returnTakeWhileIterable<E>(this, test);//可以看到它仅仅返回的是 TakeWhileIterable，而不是筛选后符合条件的集合，所以它是惰性。

}
```

2、然后，继续看下 `TakeWhileIterable<E>`的实现

```
classTakeWhileIterable<E>extendsIterable<E>{

final Iterable<E> _iterable;

final _ElementPredicate<E> _f;



TakeWhileIterable(this._iterable,this._f);



  Iterator<E>get iterator {

//重写 iterator，创建一个 TakeWhileIterator 对象并返回。

returnnewTakeWhileIterator<E>(_iterable.iterator, _f);

}}
```

```
//TakeWhileIteratorclassTakeWhileIterator<E>extendsIterator<E>{

final Iterator<E> _iterator;

final _ElementPredicate<E> _f;

  bool _isFinished =false;



TakeWhileIterator(this._iterator,this._f);



  bool moveNext(){

if(_isFinished)returnfalse;

//原集合_iterator 遍历结束或者原集合中的当前元素 current 不满足_f 条件，就返回 f
alse 以此终止外部的迭代。

//进一步说明了只有 moveNext 调用，才会触发_f 的执行，此时惰性的 Iterable 才得以
执行。

if(!_iterator.moveNext()|||!_f(_iterator.current)){

      _isFinished =true;//迭代结束重置_isFinished 为 true

returnfalse;

}

returntrue;

}



  E get current {

if(_isFinished)returnnull;//如果迭代结束，还取 current 就直接返回 null 了

return _iterator.current;

}}
```

十、skip

1、介绍

```
Iterable<E>skip(int count)
```

skip 函数主要是用于**跳过原集合前 count 个元素后，剩下元素组成的集合**，skip 函数接收一个 `count` 作为函数参数，最后返回一个泛型参数为 `E` 的 `Iterable`。类似 where 一样，skip 这里也是返回一个惰性的 `Iterable<E>`，然后对它的 `iterator` 进行迭代。

2、使用方式

```
main(){

  List<int> numbers =[3,1,2,7,12,2,4];

print(numbers.skip(2).toList());//输出: [2, 7, 12, 2, 4] 跳过前两个元素 3, 1 直接从第 3 个元素开始    }
```

3、源码解析

1、首先，因为 `numbers` 是 `List<int>`所以还是调用 `ListMixin` 中的 `skip` 函数

```
Iterable<E>skip(int count)=> SubListIterable<E>(this, count,null);//返回的是一个 SubListIterable 惰性 Iterable，传入原集合和需要跳过的 count 大小
```

2、然后，继续看下 `SubListIterable<E>`的实现,这里只看下 `elementAt` 函数实现

```
classSubListIterable<E>extendsListIterable<E>{
```

```
final Iterable<E> _iterable;// Has efficient length and elementAt.

final int _start;//这是传入的需要 skip 的 count

final int _endOrLength;//这里传入为 null

...

  int get _endIndex {

    int length = _iterable.length;//获取原集合长度

if(_endOrLength ==null|| _endOrLength > length)return length;//_endIndex
为原集合长度

return _endOrLength;

}



  int get _startIndex {//主要看下_startIndex 的实现

    int length = _iterable.length;//获取原集合长度

if(_start > length)return length;//如果 skip 的 count 超过集合自身长度，_star
tIndex 为原集合长度

return _start;//否则返回 skip 的 count

}



  E elementAt(int index){

    int realIndex = _startIndex + index;//相当于把原集合中每个元素原来 index,
整体向后推了_startIndex,最后获取真实映射的 realIndex

if(index <0|| realIndex >= _endIndex){//如果 realIndex 越界就会抛出异常

thrownewRangeError.index(index,this,"index");

}
```

```
return _iterable.elementAt(realIndex);//否则就取对应 realIndex 在原集合中的元素。

}

...}
```

十一、skipWhile

1、介绍

```
Iterable<E>skipWhile(bool test(E element))
```

skipWhile 函数主要用于**依次跳过满足条件的元素，直到遇到第一个不满足的元素，并停止筛选**。skipWhile 函数接收一个 `test` 条件函数作为函数参数，然后返回一个惰性的 `Iterable<E>`。

2、使用方式

```
main(){

  List<int> numbers =[3,1,2,7,12,2,4];

print(numbers.skipWhile((number)=> number <4).toList());//输出: [7, 12, 2, 4]

//因为 3、1、2 都是满足小于 4 的条件，所以直接 skip 跳过，直到遇到 7 不符合条件停止筛选，剩下的就是[7, 12, 2, 4]}
```

3、源码解析

1、首先，因为 `numbers` 是 `List<int>`所以还是调用 `ListMixin` 中的 `skipWhile` 函数

```
  Iterable<E>skipWhile(booltest(E element)){

returnSkipWhileIterable<E>(this, test);//可以看到它仅仅返回的是 SkipWhileI
terable，而不是筛选后符合条件的集合，所以它是惰性的。

}
```

2、然后，继续看下 `SkipWhileIterable<E>`的实现

```
classSkipWhileIterable<E>extendsIterable<E>{

final Iterable<E> _iterable;

final _ElementPredicate<E> _f;



SkipWhileIterable(this._iterable,this._f);

//重写 iterator，创建一个 SkipWhileIterator 对象并返回。

  Iterator<E>get iterator {

returnnewSkipWhileIterator<E>(_iterable.iterator, _f);

}}

//SkipWhileIteratorclassSkipWhileIterator<E>extendsIterator<E>{

final Iterator<E> _iterator;//存储原集合的 iterator

final _ElementPredicate<E> _f;//存储 skipWhile 中筛选闭包函数

  bool _hasSkipped =false;//判断是否已经跳过元素的标识，默认为 false



SkipWhileIterator(this._iterator,this._f);

//重写 moveNext 函数

  bool moveNext(){
```

```
if(!_hasSkipped){//如果是最开始第一次没有跳过任何元素

      _hasSkipped =true;//然后重置标识为 true,表示已经进行了第一次跳过元素的操作

while(_iterator.moveNext()){//迭代原集合中的 iterator

if(!_f(_iterator.current))returntrue;//只要找到符合条件的元素，就略过迭代下一个元素，不符合条件就直接返回 true 终止当前 moveNext 函数，而此时外部迭代循环正式从当前元素开始迭代，

}

}

return _iterator.moveNext();//那么遇到第一个不符合条件元素之后所有元素就会通过_iterator.moveNext()正常返回

}



  E get current => _iterator.current;}
```

十二、follwedBy

1、介绍

```
Iterable<E>followedBy(Iterable<E> other)
```

followedBy 函数主要用于**在原集合后面追加拼接另一个** `Iterable<E>` **集合**，

followedBy 函数接收一个 `Iterable<E>` 参数，最后又返回一个 `Iterable<E>` 类型的值。

2、使用方式

```
main(){
```

```
var languages =<String>['Kotlin','Java','Dart','Go','Python'];

print(languages.followedBy(['Swift','Rust','Ruby','C++','C#']).toList
());//输出: [Kotlin, Java, Dart, Go, Python, Swift, Rust, Ruby, C++, C#]}
```

3、源码解析

1、首先，还是调用 `ListMixin` 中的 `followedBy` 函数

```
  Iterable<E>followedBy(Iterable<E> other)=>

      FollowedByIterable<E>.firstEfficient(this, other);//这里实际上还是返
回一个惰性的 FollowedByIterable 对象，这里使用命名构造器 firstEfficient 创建对
象
```

2、然后，继续看下 `FollowedByIterable` 中的 `firstEfficient` 实现

```
factory FollowedByIterable.firstEfficient(

      EfficientLengthIterable<E> first, Iterable<E> second){

if(second is EfficientLengthIterable<E>){//List 肯定是一个 EfficientLength
Iterable，所以会创建一个 EfficientLengthFollowedByIterable，传入的参数 first
是当前集合，second 是需要在后面拼接的集合

returnnewEfficientLengthFollowedByIterable<E>(first, second);

}

returnnewFollowedByIterable<E>(first, second);

}
```

3、然后，继续看下 `EfficientLengthFollowedByIterable` 的实现,这里只具体看下 `elementAt` 函数的实现

```
classEfficientLengthFollowedByIterable<E>extendsFollowedByIterable<E>

implementsEfficientLengthIterable<E>{

EfficientLengthFollowedByIterable(

EfficientLengthIterable<E> first,EfficientLengthIterable<E> second)

:super(first, second);

...

EelementAt(int index){//elementAt 在迭代过程会调用

int firstLength = _first.length;//取原集合的长度

if(index < firstLength)return _first.elementAt(index);//如果 index 小于原集合长度就从原集合中获取元素

return _second.elementAt(index - firstLength);//否则就通过 index - firstLength 计算新的下标从拼接的集合中获取元素。

}

...}
```

十三、expand

1、介绍

```
Iterable<T> expand<T>(Iterable<T> f(E element))
```

expand 函数主要用于**将集合中每个元素扩展为零个或多个元素或者将多个元素组成二维数组展开成平铺一个一维数组**。 expand 函数接收一个 `Iterable<T> f(E element)` 函数作为函数参数。这个闭包函数比较特别，特别之处在于 `f` 函数返回

的是一个 `Iterable<T>`,那么就意味着可以将原集合中每个元素扩展成多个相同元素。注意 expand 函数最终还是返回一个惰性的 `Iterable<T>`

2、使用方式

```
main(){

var pair =[

[1,2],

[3,4]

];

print('flatten list: ${pair.expand((pair) => pair).toList()}');//输出: flatten list: [1, 2, 3, 4]

var inputs =[1,2,3];

print('duplicated list: ${inputs.expand((number) => [number, number, number]).toList()}');//输出: duplicated list: [1, 1, 1, 2, 2, 2, 3, 3, 3]}
```

3、源码解析

1、首先还是调用 `ListMixin` 中的 `expand` 函数。

```
  Iterable<T> expand<T>(Iterable<T>f(E element))=>

      ExpandIterable<E, T>(this, f);//可以看到这里并没有直接返回扩展的集合，而是创建一个惰性的 ExpandIterable 对象返回，
```

2、然后继续深入 `ExpandIterable`

```
typedef Iterable<T> _ExpandFunction<S, T>(S sourceElement);
```

```
classExpandIterable<S, T>extendsIterable<T>{

final Iterable<S> _iterable;

final _ExpandFunction<S, T> _f;



ExpandIterable(this._iterable,this._f);



  Iterator<T>get iterator =>newExpandIterator<S, T>(_iterable.iterator,
_f);//注意：这里 iterator 是一个 ExpandIterator 对象，传入的是原集合的 iterato
r 和 expand 函数中闭包函数参数_f}

//ExpandIterator 的实现 classExpandIterator<S, T>implementsIterator<T>{

final Iterator<S> _iterator;

final _ExpandFunction<S, T> _f;

//创建一个空的 Iterator 对象_currentExpansion

  Iterator<T> _currentExpansion =constEmptyIterator();

  T _current;



ExpandIterator(this._iterator,this._f);



  T get current => _current;//重写 current

//重写 moveNext 函数，只要当迭代的时候，moveNext 执行才会触发闭包函数_f 执行。

  bool moveNext(){

//如果_currentExpansion 返回 false 终止外部迭代循环

if(_currentExpansion ==null)returnfalse;

//开始_currentExpansion 是一个空的 Iterator 对象，所以 moveNext()为 false
```

```
while(!_currentExpansion.moveNext()){

      _current =null;

//迭代原集合中的_iterator

if(_iterator.moveNext()){

//如果_f 抛出异常，先重置_currentExpansion 为 null，遇到 if (_currentExpansi
on == null) return false;就会终止外部迭代

       _currentExpansion =null;

       _currentExpansion =_f(_iterator.current).iterator;//执行_f 函数

}else{

returnfalse;

}

}

    _current = _currentExpansion.current;

returntrue;

}}
```

十四、reduce

1、介绍

```
E reduce(E combine(E previousValue, E element))

T fold<T>(T initialValue, T combine(T previousValue, E element))
```

reduce 函数主要用于集合中元素依次归纳(combine)，每次归纳后的结果会和下一个元素进行归纳，它可以用来累加或累乘，具体取决于 combine 函数中操作，combine 函数中会回调上一次归纳后的值和当前元素值，reduce 提供的是获取累积迭代结果的便利条件. fold 和 reduce 几乎相同，唯一区别是 fold 可以指定初始值。 但是需要注意的是，combine 函数返回值的类型必须和集合泛型类型一致。

2、使用方式

```
main(){

  List<int> numbers =[3,1,2,7,12,2,4];

print(numbers.reduce((prev, curr)=> prev + curr));//累加

print(numbers.fold(2,(prev, curr)=>(prev as int)+ curr));//累加

print(numbers.reduce((prev, curr)=> prev + curr)/ numbers.length);//求平均数

print(numbers.fold(2,(prev, curr)=>(prev as int)+ curr)/ numbers.length);//求平均数

print(numbers.reduce((prev, curr)=> prev * curr));//累乘

print(numbers.fold(2,(prev, curr)=>(prev as int)* curr));//累乘



var strList =<String>['a','b','c'];

print(strList.reduce((prev, curr)=>'$prev*$curr'));//拼接字符串

print(strList.fold('e',(prev, curr)=>'$prev*$curr'));//拼接字符串}
```

3、源码解析

```
Ereduce(Ecombine(E previousValue,E element)){

int length =this.length;

if(length ==0)throw IterableElementError.noElement();

Evalue=this[0];//初始值默认取第一个

for(int i =1; i < length; i++){//从第二个开始遍历

value=combine(value,this[i]);//combine 回调 value 值和当前元素值，然后把 com
bine 的结果归纳到 value 上，依次处理。

if(length !=this.length){

throwConcurrentModificationError(this);//注意：在操作过程中不允许删除和添
加元素否则就会出现 ConcurrentModificationError

}

}

returnvalue;

}



  T fold<T>(T initialValue,Tcombine(T previousValue,E element)){

varvalue= initialValue;//和 reduce 唯一区别在于这里 value 初始值是外部指定的

int length =this.length;

for(int i =0; i < length; i++){

value=combine(value,this[i]);

if(length !=this.length){

throwConcurrentModificationError(this);

}

}
```

```
returnvalue;

}
```

十五、elementAt

1、介绍

```
E elementAt(int index)
```

elementAt 函数用于**获取对应 index 下标的元素**，传入一个 index 参数，返回对应泛型类型 `E` 的元素。

2、使用方式

```
main(){

print(numbers.elementAt(3));//elementAt 一般不会直接使用，更多是使用[]，运算符重载的方式间接使用。 }
```

3、源码解析

```
EelementAt(int index){

    ArgumentError.checkNotNull(index,"index");

    RangeError.checkNotNegative(index,"index");

int elementIndex =0;

//for-in 遍历原集合，找到对应 elementIndex 元素并返回

for(E element inthis){

if(index == elementIndex)return element;
```

```
      elementIndex++;

}

//找不到抛出 RangeError

throw RangeError.index(index,this,"index",null, elementIndex);

}
```

总结

到这里，有关 dart 中集合操作符函数相关内容就结束了，关于集合操作符函数使用在 Flutter 中开发非常有帮助，特别在处理集合数据中，可以让你的代码实现更优雅，不要再是一上来就 for 循环直接开干，虽然也能实现，但是如果能适当使用操作符函数，将会使代码更加简洁。欢迎继续关注，下一篇 Dart 中的函数的使用...

第八章 Dart 语法篇之函数的使用(四)

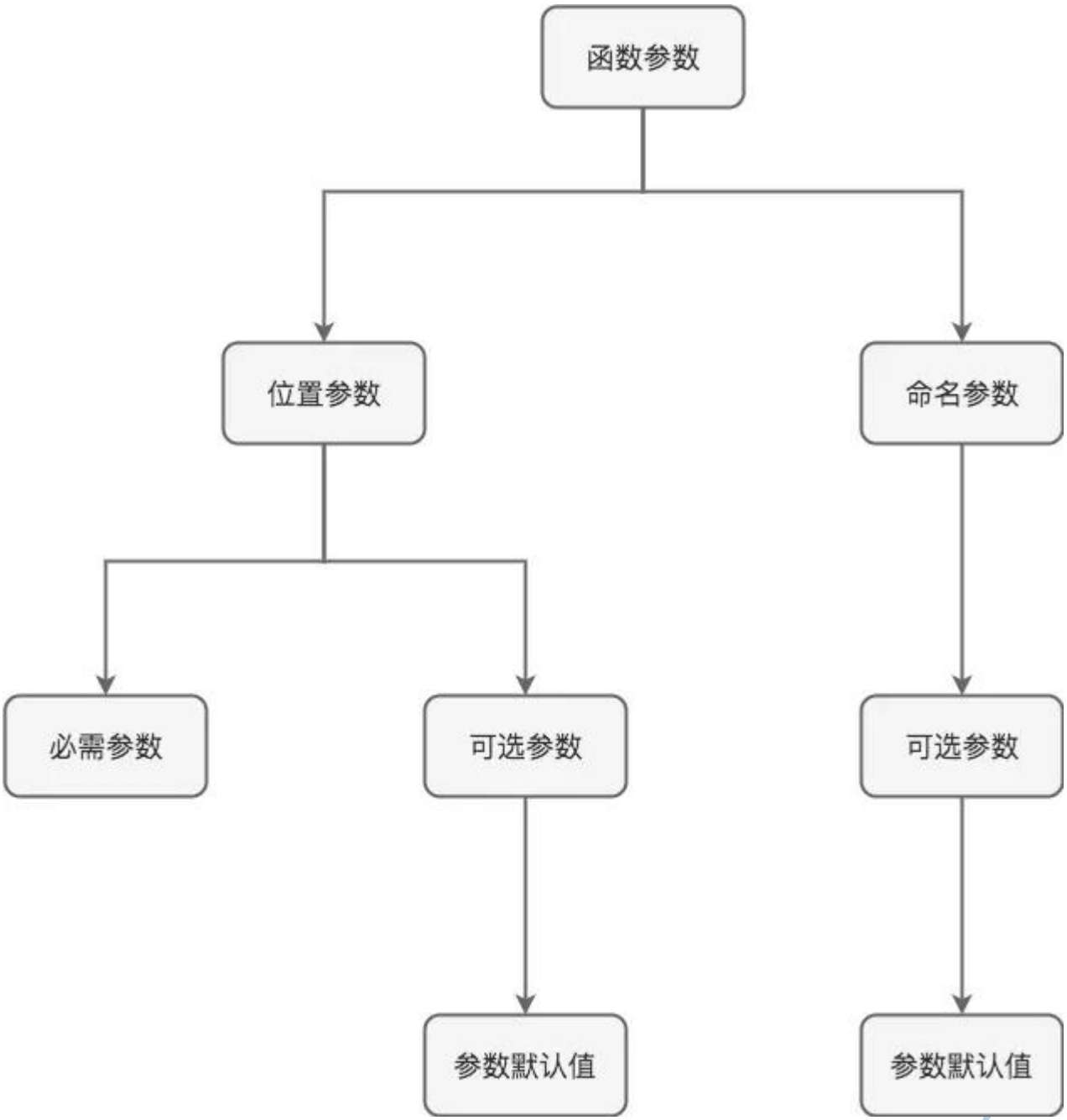
简述：

在上一篇文章中我们详细地研究了一下集合有关内容，包括集合的操作符的使用甚至我们还深入到源码实现原理，从原理上掌握集合的使用。那么这篇文章来研究一下 Dart 的另一个重要语法: **函数**。

这篇主要会涉及到: 函数命名参数、可选参数、参数默认、闭包函数、箭头函数以及函数作为对象使用。

一、函数参数

在 Dart 函数参数是一个比较重要的概念，此外它涉及到概念的种类比较多，比如位置参数、命名参数、可选位置参数、可选命名参数等等。函数总是有一个所谓形参列表，虽然这个参数列表可能为空，比如 **getter 函数就是没有参数列表的**. 此外在 Dart 中函数参数大致可分为两种: **位置参数和命名参数**，来一张图理清它们的概念关系



1、位置参数

位置参数可以必需的也可以是可选。

无参数

```
//无参数类型-这是不带函数参数或者说参数列表为空
```

```
String getDefaultErrorMsg()=>'Unknown Error!';//无参数类型-等价于上面函数形式，同样是参数列表为空 get getDefaultErrorMsg =>'Unknown Error!';
```

必需位置参数

```
//必需位置参数类型-这里的 exception 是必需的位置参数

String getErrorMsg(Exception exception)=> exception.toString();
```

可选位置参数

```
//注意：可选位置参数是中括号括起来表示，例如[String error]

String getErrorMsg([String error])=> error ??'Unknown Error!';
```

必需位置参数和可选位置参数混合

```
//注意：可选位置参数必须在必需位置参数的后面

String getErrorMsg(Exception exception,[String extraInfo])=>'${exception.toString()}---$extraInfo';
```

2、命名参数

命名参数**始终是可选参数**。为什么是命名参数，这是因为在调用函数时可以任意指定参数名来传参。

可选命名参数

```
//注意：可选命名参数是大括号括起来表示，例如{num a, num b, num c, num d}

num add({num a, num b, num c, num d}){

return a + b + c + d;}//调用 main(){
```

```
print(add(d:4, b:3, a:2, c:1));//这里的命名参数就是可以任意顺序指定参数名传值,例如 d: 4, b: 3, a: 2, c: 1}
```

必需位置参数和可选命名参数混合

```
//注意：可选命名参数必须在必需位置参数的后面

num add(num a, num b,{num c, num d}){

return a + b + c + d;}//调用 main(){

print(add(4,5, d:3, c:1));//这里的命名参数就是可以任意顺序指定参数名传值,例如 d: 3, c: 1,但是必需参数必须按照顺序传参。}
```

注意：可选位置参数和可选命名参数不能混合在一起使用，因为可选参数列表只能位于整个函数形参列表的最后。

```
voidadd7([num a, num b],{num c, num d}){//非法声明，想想也没有必要两者一起混合使用场景。所以

...}
```

3、关于可选位置参数`[num a, num b]`和可选命名参数`{num a, num b}`使用场景

可能问题来了，啥时候使用可选位置参数，啥时候使用可选命名参数呢?

这里给个建议：**首先，参数是非必需的也就是可选的，如果可选参数个数只有一个建议直接使用可选位置参数`[num a, num b]`；如果可选参数个数是多个的话建议用可选命名参数`{num a, num b}`**. 因为多个参数可选，指定参数名传参对整体代码可读性有一定的增强。

4、参数默认值(针对可选参数)

首先，需要明确一点，参数默认值只针对可选参数才能添加的。可以使用 = 来定义命名和位置参数的默认值。**默认值必须是编译时常量。如果没有提供默认值，则默认值为 null。**

可选位置参数默认值

```
num add(num a, num b, num c,[num d =5]}){//使用=来赋值默认值

return a + b + c + d;}main(){

print(add(1,2,3));//有默认值参数可以省略不传 实际上求和结果是: 1 + 2 + 3 + 5(默认值)

print(add(1,2,3,4));//有默认值参数指定传入 4，会覆盖默认值，所以求和结果是: 1 + 2 + 3 + 4}
```

可选命名参数默认值

```
num add({num a, num b, num c =3, num d =4}){

return a + b + c + d;}main(){

print(add(100,100, d:100, c:100));}
```

二、匿名函数(闭包，lambda)

在 Dart 中可以创建一个没有函数名称的函数，这种函数称为**匿名函数，或者 lambda 函数或者闭包函数**。但是和其他函数一样，它也有形参列表，可以有可选参数。

```
(num x)=> x;//没有函数名，有必需的位置参数 x(num x){return x;}//等价于上面形式(int x,[int step])=> x + step;//没有函数名，有可选的位置参数 step(int x,{i
```

```
nt step1, int step2})=> x + step1 + step2;////没有函数名，有可选的命名参数 s
tep1、step2
```

闭包在 dart 中的应用

闭包函数在 dart 用的特别多，单从集合中操作符来说就有很多。

```
main(){

  List<int> numbers =[3,1,2,7,12,2,4];

//reduce 函数实现累加,reduce 函数中接收的(prev, curr) => prev + curr 就是一个
闭包

print(numbers.reduce((prev, curr)=> prev + curr));

//还可以不用闭包形式来写，但是这并不是一个好的方案,不建议下面这样使用。

plus(prev, curr)=> prev + curr;

print(numbers.reduce(plus));}//reduce 函数定义

Ereduce(Ecombine(Evalue,E element)){//combine 闭包函数

    Iterator<E> iterator =this.iterator;

if(!iterator.moveNext()){

throw IterableElementError.noElement();

}

Evalue= iterator.current;

while(iterator.moveNext()){

value=combine(value, iterator.current);//执行 combine 函数

}

returnvalue;
```

```
}
```

三、箭头函数

在 Dart 中还有一种函数的简写形式，那就是**箭头函数**。箭头函数**是只能包含一行表达式**的函数，会注意到它**没有花括号，而是带有箭头**的。箭头函数更有助于代码的可读性，类似于 Kotlin 或 Java 中的 lambda 表达式`->`的写法。

```
main(){

  List<int> numbers =[3,1,2,7,12,2,4];

print(numbers.reduce((prev, curr){//闭包简写形式

return prev + curr;

}));

print(numbers.reduce((prev, curr)=> prev + curr));//等价于上述形式，箭头函数简写形式}
```

四、局部函数

在 Dart 中还有一种可以直接定义在函数体内部的函数，可以把称为**局部函数或者内嵌函数**。我们知道函数声明可以出现顶层，比如常见的 main 函数等等。局部函数的好处就是从作用域角度来看，它可以访问外部函数变量，并且还能避免引入一个额外的外部函数，使得整个函数功能职责统一。

```
//定义外部函数 fibonacciintfibonacci(int n){

//定义局部函数 lastTwo

    List<int>lastTwo(int n){

if(n <1){
```

```
return<int>[0,1];

}else{

var p =lastTwo(n -1);

return<int>[p[1], p[0]+ p[1]];

}

}

returnlastTwo(n)[1];}
```

五、顶层函数和静态函数

在 Dart 中有一种特别的函数，我们知道在面向对象语言中比如 Java，并不能直接定义一个函数的，而是需要定义一个类，然后在类中定义函数。但是在 Dart 中可以不用在类中定义函数，而是直接基于 dart 文件顶层定义函数，这种函数我们一般称为**顶层函数**。最常见就是 main 函数了。而静态函数就和 Java 中类似，依然使用 **static 关键字来声明**，然后必须是**定义在类的内部**的。

```
//顶层函数，不定义在类的内部 main(){

print('hello dart');}

classNumber{

static int getValue()=>100;//static 修饰定义在类的内部。}
```

六、main 函数

每个应用程序都有一个顶级的 main()函数，它作为应用程序的入口点。main()函数返回 void，所以在 dart 可以直接省略 void，并有一个可选的列表参数作为参数。

```
//你一般看到的 main 是这样的 main(){

print('hello dart');}//实际上它和 Java 类似可以带个参数列表 main(List<String>
 args){

print('hello dart: ${args[0]}, ${args[1]}');//用 dart command 执行的时候: d
art test.dart arg0 arg1 =>输出:hello dart: arg0, arg1    }
```

七、Function 函数对象

在 Dart 中一切都是对象，函数也不例外，函数可以作为一个参数传递。其中 `Function` 类是代表所有函数的公共顶层接口抽象类。`Function` 类中并没有声明任何实例方法。但是它有一个非常重要的静态类函数 `apply`. 该函数接收一个 `Function` 对象 `function`，一个 `List` 的参数 `positionalArguments` 以及一个可选参数 `Map<Symbol, dynamic>` 类型的 `namedArguments`。大家似乎明白了什么？知道为啥 dart 中函数支持位置参数和命名参数吗？没错就是它们两个参数功劳。实际上，**apply()函数提供一种使用动态确定的参数列表来调用函数的机制，通过它我们就能处理在编译时参数列表不确定的情况**。

```
abstractclassFunction{

externalstaticapply(Function function, List positionalArguments,

[Map<Symbol,dynamic> namedArguments]);//可以看到这是 external 声明，我们需要
找到对应的 function_patch.dart 实现



  int get hashCode;
```

```
  bool operator==(Object other);}
```

在 sdk 源码中找到 `sdk/lib/_internal/vm/lib/function_patch.dart` 对应的 `function_patch` 的实现

```
@patchclassFunction{

// TODO(regis): Pass type arguments to generic functions. Wait for API s
pec.

//可以看到内部私有的_apply 函数，最终接收两个 List 原生类型的参数 arguments,nam
es 分别代表着我们使用函数时

//定义的所有参数 List 集合 arguments(包括位置参数和命名参数)以及命名参数名 List
集合 names，不过它是委托到 native 层的 Function_apply C++函数实现的。

static _apply(List arguments, List names) native "Function_apply";



@patch

staticapply(Function function, List positionalArguments,

[Map<Symbol,dynamic> namedArguments]){

//计算外部函数位置参数的个数

    int numPositionalArguments =1+// 默认同时会传入 function 参数，所以默认+
1

(positionalArguments !=null? positionalArguments.length :0);//位置参数的
集合不为空就返回集合长度否则返回 0

//计算外部函数命名参数的个数

    int numNamedArguments = namedArguments !=null? namedArguments.length
 :0;;//命名参数的集合不为空就返回集合长度否则返回 0

//计算所有参数个数总和: 位置参数个数 + 命名参数个数

    int numArguments = numPositionalArguments + numNamedArguments;
```

```
//创建一个定长为所有参数个数大小的 List 集合 arguments

    List arguments =newList(numArguments);

//集合第一个元素默认是传入的 function 对象

    arguments[0]= function;

//然后从 1 的位置开始插入所有的位置参数到 arguments 参数列表中

    arguments.setRange(1, numPositionalArguments, positionalArguments);

//然后再创建一个定长为命名参数长度的 List 集合

    List names =newList(numNamedArguments);

    int argumentIndex = numPositionalArguments;

    int nameIndex =0;

//遍历命名参数 Map 集合

if(numNamedArguments >0){

      namedArguments.forEach((name, value){

        arguments[argumentIndex++]= value;//把命名参数对象继续插入到 argume
nts 集合中

        names[nameIndex++]= internal.Symbol.getName(name);//并把对应的参数
名标识存入 names 集合中

});

}

return_apply(arguments, names);//最后调用_apply 函数传入所有参数对象集合以及
命名参数名称集合

}}
```

不妨再来瞅瞅 C++层中的 `Function_apply` 的实现

```
DEFINE_NATIVE_ENTRY(Function_apply,0,2){

const int kTypeArgsLen =0;// TODO(regis): Add support for generic functi
on.

constArray& fun_arguments =

Array::CheckedHandle(zone, arguments->NativeArgAt(0));//获取函数的所有参
数对象数组 fun_arguments

constArray& fun_arg_names =

Array::CheckedHandle(zone, arguments->NativeArgAt(1));//获取函数的命名参
数参数名数组 fun_arg_names

constArray& fun_args_desc =Array::Handle(

      zone, ArgumentsDescriptor::New(kTypeArgsLen, fun_arguments.Length
(),

                                     fun_arg_names));//利用 fun_arg_names 生
成对应命名参数描述符集合

//注意：这里会调用 DartEntry 中的 InvokeClosure 函数，传入了所有参数对象数组 f
un_arguments 和 fun_arg_names 生成对应命名参数描述符集合//最后返回 result

const Object& result = Object::Handle(

      zone, DartEntry::InvokeClosure(fun_arguments, fun_args_desc));

if(result.IsError()){

    Exceptions::PropagateError(Error::Cast(result));

}

return result.raw();}
```

总结

到这里有关 Dart 中的函数就说完，有了这篇文章相信大家对 dart 函数应该有个全面的了解了。欢迎持续，下一篇我们将进入 Dart 中的面向对象...

第九章 Dart 语法篇之面向对象基础(五)

简述：

从这篇文章开始，我们继续 Dart 语法篇的第五讲, dart 中的面向对象基础。我们知道在 Dart 中一切都是对象，所以面向对象在 Dart 开发中是非常重要的。此外它还和其他有点不一样的地方，比如多继承 mixin、构造器不能被重载、setter 和 getter 的访问器函数等。

一、属性访问器(accessor)函数 setter 和 getter

在 Dart 类的属性中有一种为了方便访问它的值特殊函数，那就是 setter,getter 属性访问器函数。**实际上，在 dart 中每个实例属性始终有与之对应的 setter,getter 函数(若是 final 修饰只读属性只有 getter 函数，而可变属性则有 setter，getter 两种函数)。而在给实例属性赋值或获取值时，实际上内部都是对 setter 和 getter 函数的调用。**

1、属性访问器函数 setter

setter 函数名前面添加**前缀** `set`，并只接收一个参数。setter 调用语法于传统的变量赋值是一样的。如果一个实例属性是可变的，那么一个 setter 属性访问器函数就会为它自动定义，所有实例属性的赋值实际上都是对 setter 函数的调用。这一点和 Kotlin 中的 setter，getter 非常相似。

```
classRectangle{

  num left, top, width, height;
```

```
Rectangle(this.left,this.top,this.width,this.height);

setright(num value)=> left =value- width;//使用 set 作为前缀，只接收一个参数 value

setbottom(num value)=> top =value- height;//使用 set 作为前缀，只接收一个参数 value}

main(){

var rect =Rectangle(3,4,20,15);

  rect.right =15;//调用 setter 函数时，可以直接使用类似属性赋值方式调用 right 函数。

  rect.bottom =12;//调用 setter 函数时，可以直接使用类似属性赋值方式调用 bottom 函数。}
```

对比 Kotlin 中的实现

```
classRectangle(var left: Int,var top: Int,var width: Int,var height: Int)
{

var right: Int =0//在 kotlin 中表示可变使用 var,只读使用 val

set(value){//kotlin 中定义 setter

            field = value

            left = value - width

}

var bottom: Int =0

set(value){//kotlin 中定义 setter

            field = value

            top = value - height
```

```
}}

funmain(args: Array<String>){

val rect =Rectangle(3,4,20,15);

    rect.right =15//调用 setter 函数时，可以直接使用类似属性赋值方式调用 right
函数。

    rect.bottom =12//调用 setter 函数时，可以直接使用类似属性赋值方式调用 bott
om 函数。 }
```

2、属性访问器函数 getter

Dart 中所有实例属性的访问都是通过调用 getter 函数来实现的。每个实例数额行始终都有一个与之关联的 getter,由 Dart 编译器提供的。

```
classRectangle{

  num left, top, width, height;



Rectangle(this.left,this.top,this.width,this.height);

get square => width * height;;//使用 get 作为前缀，getter 来计算面积.

setright(num value)=> left = value - width;//使用 set 作为前缀，只接收一个参
数 value

setbottom(num value)=> top = value - height;//使用 set 作为前缀，只接收一个
参数 value}

main(){

var rect =Rectangle(3,4,20,15);

  rect.right =15;//调用 setter 函数时，可以直接使用类似属性赋值方式调用 right
函数。
```

```
  rect.bottom =12;//调用 setter 函数时，可以直接使用类似属性赋值方式调用 botto
m 函数。

print('the rect square is ${rect.square}');//调用 getter 函数时，可以直接使
用类似读取属性值方式调用 square 函数。}
```

对比 kotlin 实现

```
classRectangle(var left: Int,var top: Int,var width: Int,var height: Int)
{

var right: Int =0

set(value){

            field = value

            left = value - width

}

var bottom: Int =0

set(value){

            field = value

            top = value - height

}



val square: Int//因为只涉及到了只读，所以使用 val

get()= width * height//kotlin 中定义 getter}

funmain(args: Array<String>){

val rect =Rectangle(3,4,20,15);

    rect.right =15
```

```
    rect.bottom =12

println(rect.square)//调用 getter 函数时，可以直接使用类似读取属性值方式调用 s
quare 函数。}
```

3、属性访问器函数使用场景

其实，上面 `setter`，`getter` 函数实现的目的，普通函数也能做到的。但是如果用 `setter,getter` 函数形式更符合编码规范。既然普通函数也能做到，那具体什么时候使用 `setter,getter` 函数，什么时候使用普通函数呢。这不得不把这个问题和另一问题转化一下成为: 哪种场景该定义属性还是定义函数的问题(关于这个问题，记得很久之前在讨论 Kotlin 的语法详细介绍过)。我们都知道**函数一般描述动作行为，而属性则是描述状态数据结构(状态可能经过多个属性值计算得到)。**如果类中需要向外暴露类中某个状态那么更适合使用 `setter,getter` 函数；如果是触发类中的某个行为操作，那么普通函数更适合一点。

比如下面这个例子，`draw` 绘制矩形动作更适合使用普通函数来实现，`square` 获取矩形的面积更适合使用 getter 函数来实现,可以仔细体会下。

```
classRectangle{

  num left, top, width, height;



Rectangle(this.left,this.top,this.width,this.height);



setright(num value)=> left = value - width;//使用 set 作为前缀，只接收一个参
数 value

setbottom(num value)=> top = value - height;//使用 set 作为前缀，只接收一个
参数 value
```

```
get square => width * height;//getter 函数计算面积，描述 Rectangle 状态特性

  bool draw(){

print('draw rect');//draw 绘制函数，触发是动作行为

returntrue;

}}

main(){

var rect =Rectangle(3,4,20,15);

  rect.right =15;//调用 setter 函数时，可以直接使用类似属性赋值方式调用 right 函数。

  rect.bottom =12;//调用 setter 函数时，可以直接使用类似属性赋值方式调用 botto m 函数。

print('the rect square is ${rect.square}');

  rect.draw();}
```

二、面向对象中的变量

1、实例变量

实例变量实际上就是类的成员变量或者称为成员属性，当声明一个实例变量时，它会确保每一个对象实例都有自己唯一属性的拷贝。如果要表示实例私有属性的话就直接在属性名前面加下划线 _,例如`_width`和`_height`

```
classRectangle{

  num left, top, _width, _height;//声明了 left,top,_width,_height 四个成员属性，未初始化时，它们的默认值都是 null}
```

上述例子中的 `left, top, width, height` 都是会自动引入一个 `getter` 和 `setter`.事实上，在 dart 中属性都不是直接访问的，所有对字段属性的引用都是对属性访问器函数的调用，只有访问器函数才能直接访问它的状态。

2、类变量(static 变量)与顶层变量

类变量实际上就是 `static` 修饰的变量，属于类的作用域范畴；**顶层变量**就是**定义的变量不在某个具体类体内**，而是处于整个代码文件中，相当于文件顶层，和顶层函数差不多意思。`static` 变量更多人愿意把它称为**静态变量**，但是在 **Dart 中静态变量不仅仅包括 static 变量还包括顶层变量**。

其实对于**类变量和顶层变量的访问都还是通过调用它的访问器函数来实现的**，但是类变量和顶层变量有点特殊，**它们是延迟初始化的**，在 `getter` 函数第一次被调用时类变量或顶层变量才执行初始化，也即是第一次引用类变量或顶层变量的时候。如果类变量或顶层变量没有被初始化默认值还是 `null`.

```
classAnimal{}classDogextendsAnimal{}classCatextendsAnimal{

Cat(){

print("I'm a Cat!");

}}//注意,这里变量不定义在任何具体类体内，所以这个 animal 是一个顶层变量。//虽然看似创建了 Cat 对象，但是由于顶层变量延迟初始化的原因，这里根本就没有创建 Cat 对象

Animal animal =Cat();main(){

    animal =Dog();//然后又将 animal 引用指向了一个新的 Dog 对象，}
```

顶层变量是具有延迟初始化过程，所以 `Cat` 对象并没有创建，因为整个代码执行中并没有去访问 `animal`，所以无法触发第一次 `getter` 函数，也就导致 `Cat` 对象没有创建，直接表现是根本就不会输出 `I'm a Cat!`这句话。这就是为什么顶层变量是延迟初始化的原因，`static` 变量同理。

3、final 变量

在 Dart 中使用 `final` 关键字修饰变量，表示该变量初始化后不能再被修改。

`final` 变量只有 `getter` 访问器函数，没有 `setter` 函数。类似于 Kotlin 中的 `val` 修饰的变量。声明成 `final` 的变量必须在实例方法运行前进行初始化，所以初始化 `final` 变量有很多中方法。注意: 建议尽量使用 `final` 来声明变量

```
classPerson{

final String gender ='男';//直接在声明的时候初始化，这种方式比较局限，针对基本数据类型还可以，但如果是一个对象类型就显示不合适了。

final String name;

final int age;

Person(this.name,this.age);//利用构造函数为 final 变量初始化。

//上述代码等价于下面实现

Person(String name, int age){

this.name = name;

this.age = age;

}}
```

`final` 与 `const` **的区别**，就好比 Kotlin 中的 `val` 与 `const val` 之间的区别，`const` **是编译期就进行了初始化**，而 `final` **则是运行期进行初始化**。

4、常量对象

在 dart 有些**对象是在编译期就可以计算的常量**，所以在 dart 中支持常量对象的定义，**常量对象**的创建需要使用 `const` 关键字。**常量对象的创建也是调用类的构**

造函数，但是注意必须是常量构造函数，该构造函数是用 `const` 关键字修饰的。常量构造函数**必须是数字、布尔值或字符串，此外常量构造函数不能有函数体**，但是它可以有初始化列表。

```
classPoint{

finaldouble x, y;

constPoint(this.x,this.y);//常量构造函数，使用 const 关键字且没有函数体}

main(){

const defaultPoint =constPoint(0,0);//创建常量对象}
```

三、构造函数

1、主构造函数

主构造函数是 Dart 中创建对象最普通一种构造函数，而且**主构造函数只能有一个，如果没有指定主构造函数，那么会默认自动分配一个默认无参的主构造函数**。此外 **dart 中构造函数不支持重载**

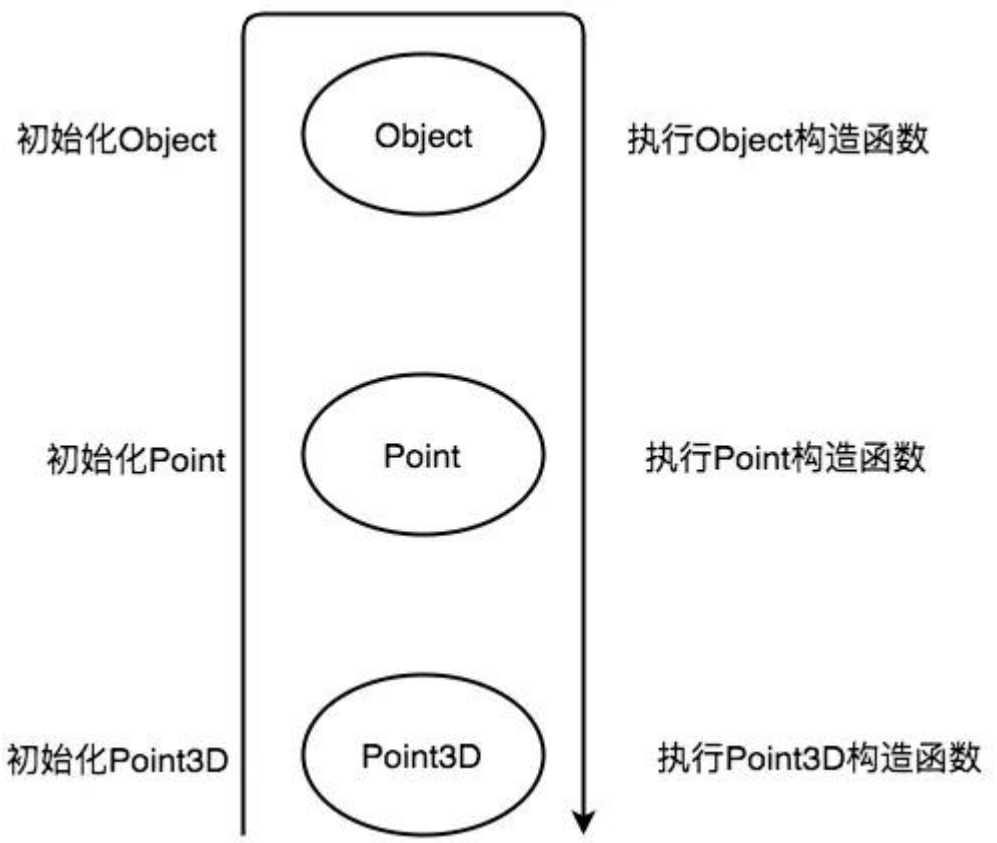
```
classPerson{

var name;

//隐藏了默认的无参构造函数 Person();}//等价于:classPerson{

var name;

Person();//一般把与类名相同的函数称为主构造函数}//等价于 classPerson{

var name;

Person(){}}

classPerson{
```

```
final String name;

final int age;

Person(this.name,this.age);//显式声明有参主构造函数

Person();//编译异常，注意：dart 不支持同时重载多个构造函数。}
```

构造函数初始化列表：

```
classPoint3DextendsPoint{

    double z;

Point3D(a, b, c): z = c /2,super(a, b);//初始化列表，多个初始化步骤用逗号分隔；先初始化 z ，然后执行 super(a, b)调用父类的构造函数}//等价于 classPoint3DextendsPoint{

    double z;

Point3D(a, b, c): z = c /2;//如果初始化列表没有调用父类构造函数，

//那么就会存在一个隐含的父类构造函数 super 调用将会默认添加到初始化列表的尾部}
```

初始化的顺序如下图:



初始化实例变量几种方式

```
//方式一：通过实例变量声明时直接赋默认值初始化 classPoint{

double x =0, y =0;}

//方式二：使用构造函数初始化方式 classPoint{

double x, y;

Point(this.x,this.y);}

//方式三：使用初始化列表初始化 classPoint{

double x, y;

Point(double a,double b): x = a, y = b;//:后跟初始化列表}

//方式四：在构造函数中初始化，注意这个和方式二还是有点不一样的。classPoint{

double x, y;

Point(double a,double b){
```

```
        x = a;

        y = b;

}}
```

2、命名构造函数

通过上面主构造函数我们知道在 **Dart 中的构造函数是不支持重载的，实际上 Dart 中连基本的普通函数都不支持函数重载。** 那么问题来了，我们经常会遇到构造函数重载的场景，有时候需要指定不同的构造函数形参来创建不同的对象。所以为了解决不同参数来创建对象问题，**虽然抛弃了函数重载，但是引入命名构造函数的概念**。它可以指定任意参数列表来构建对象，只不过的是需要给构造函数指定特定的名字而已。

```
class Person {

final String name;

  int age;



Person(this.name,this.age);



  Person.withName(this.name);//通过类名.函数名形式来定义命名构造函数 withNam
e。只需要 name 参数就能创建对象，

//如果没有命名构造函数，在其他语言中，我们一般使用函数重载的方式实现。}

main(){

var person =Person('mikyou',18);//通过主构造函数创建对象

var personWithName = Person.withName('mikyou');//通过命名构造函数创建对象}
```

3、重定向构造函数

有时候需要将**构造函数重定向到同一个类中的另一个构造函数，重定向构造函数的主体为空，构造函数的调用出现在冒号(:)之后。**

```
classPoint{

    double x, y;

Point(this.x,this.y);

    Point.withX(double x):this(x,0);//注意这里使用 this 重定向到 Point(doubl
e x, double y)主构造函数中。}//或者 import'dart:math';

classPoint{

  double distance;



  Point.withDistance(this.distance);



Point(double x, double y):this.withDistance(sqrt(x * x + y * y));//注意:
这里是主构造函数重定向到命名构造函数 withDistance 中。}
```

4、factory 工厂构造函数

一般来说，构造函数总是会创建一个新的实例对象。但是有时候会遇到并不是每次都需要创建新的实例，可能需要使用缓存，如果仅仅使用上面普通构造函数是很难做到的。那么这时候就需要 **factory 工厂构造函数。它使用 `factory` 关键字来修饰构造函数，并且可以从缓存中的返回已经创建实例或者返回一个新的实例。**在 dart 中任意构造函数都可以被替换成工厂方法， 它看起来和普通构造函数没什么区别，可能没有初始化列表或初始化参数，但是它**必须有一个返回对象的函数体。**

```
classLogger{
```

```
//实例属性

final String name;

  bool mute =false;



// _cache is library-private, thanks to

// the _ in front of its name.

staticfinal Map<String, Logger> _cache =

<String, Logger>{};//类属性



factoryLogger(String name){//使用 factory 关键字声明工厂构造函数，

if(_cache.containsKey(name)){

return _cache[name]//返回缓存已经创建实例

}else{

final logger = Logger._internal(name);//缓存中找不到对应的 name logger，调用_internal 命名构造函数创建一个新的 Logger 实例

     _cache[name]= logger;//并把这个实例加入缓存中

return logger;//注意：最后返回这个新创建的实例

}

}



  Logger._internal(this.name);//定义一个命名私有的构造函数_internal



voidlog(String msg){//实例方法
```

```
if(!mute)print(msg);

}}
```

四、抽象方法、抽象类和接口

抽象方法就是**声明一个方法而不提供它的具体实现**。任何实例的方法都可以是抽象的，包括 getter,setter,操作符或者普通方法。含有抽象方法的类本身就是一个抽象类，抽象类的声明使用关键字 `abstract`.

```
abstractclassPerson{//abstract 声明抽象类

  String name();//抽象普通方法

get age;//抽象 getter}

classStudentextendsPerson{//使用 extends 继承

@override

  String name(){

// TODO: implement name

returnnull;

}



@override

// TODO: implement age

get age =>null;}
```

在 Dart 中并没有像其他语言一样有个 `interface` 的关键字修饰。因为 **Dart 中每个类都默认隐含地定义了一个接口**。

```
abstractclassSpeaking{//虽然定义的是抽象类，但是隐含地定义接口 Speaking
  String speak();}
abstractclassWriting{//虽然定义的是抽象类，但是隐含地定义接口 Writing
  String write();}
classStudentimplementsSpeaking, Writing {//使用 implements 关键字实现接口
@override
  String speak(){//重写 speak 方法
// TODO: implement speak
returnnull;
}

@override
  String write(){//重写 write 方法
// TODO: implement write
returnnull;
}}
```

五、类函数

类函数顾名思义就是**类的函数，它不属于任何一个实例，所以它也就不能被继承**。类函数使用 `static` 关键字修饰，调用时可以直接使用类名.函数名的方式调用。

```
classPoint{

    double x,y;

Point(this.x,this.y);

static double distance(Point p1, Point p2){//使用 static 关键字，定义类函数。

var dx = p1.x - p2.x;

var dy = p1.y - p2.y;

returnsqrt(dx * dx + dy * dy);

}}main(){

var point1 =Point(2,3);

var point2 =Point(3,4);

print('the distance is ${Point.distance(point1, point2)}');//使用 Point.d
istance => 类名.函数名方式调用}
```

总结

到这里有关 dart 中面向对象基础部分已经介绍完毕，这篇文章主要介绍了 dart 中常用的构造函数以及一些面向对象基础知识，下一篇我们将继续 dart 中面向对象一些高级的东西，比如面向对象的继承和 mixin.

第十章 Dart 语法篇之面向对象继承和 Mixins(六)

简述：

上一篇文章中我们详细地介绍了 Dart 中的面向对象的基础，这一篇文章中我们继续探索 Dart 中面向对象的重点和难点(继承和 mixins). mixins(混合)特性是很多语言中都是没有的。这篇文章主要涉及到 Dart 中的普通继承、mixins 多继承的形式(实际上本质并不是真正意义的多继承)、mixins 线性化分析、mixins 类型、mixins 使用场景等。

一、类的单继承

1、基本介绍

Dart 中的单继承和其他语言中类似，都是通过使用 `extends` 关键字来声明。例如

```
classStudentextendsPerson{//Student 类称为子类或派生类，Person 类称为父类或基类或超类。这一点和 Java 中是一致的。

...}
```

2、继承中的构造函数

- 子类中构造函数会默认调用父类中无参构造函数(一般为主构造函数)。

```
classPerson{

  String name;
```

```
  String age;



Person(){

print('person');

}}

classStudentextendsPerson{

  String classRoom;

Student(){

print('Student');

}}

main(){

var student =Student();//构造 Student()时会先调用父类中无参构造函数，再调用子类中无参构造函数}
```

输出结果:

```
person

Student



Process finished with exit code 0
```

- 若父类中没有默认无参的构造函数，则需要显式调用父类的构造函数(可以是命名构造函数也可以主构造函数或其他)，并且在初始化列表的尾部显式调用父类中构造函数，也即是类构造函数 :后面列表的尾部。

```
classPerson{

  String name;

  int age;



Person(this.name,this.age);//指定了带参数的构造函数为主构造函数，那么父类中就没有默认无参构造函数

//再声明两个命名构造函数

  Person.withName(this.name);



  Person.withAge(this.age);}

classStudentextendsPerson{

  String classRoom;



Student(String name, int age):super(name, age){//显式调用父类主构造函数

print('Student');

}



  Student.withName(String name):super.withName(name){}//显式调用父类命名构造函数 withName
```

```
  Student.withAge(int age):super.withAge(age){}//显式调用父类命名构造函数 w
ithAge}

main(){

var student1 =Student('mikyou',18);

var student2 = Student.withName('mikyou');

var student3 = Student.withAge(18);}
```

- 父类的构造函数在子类构造函数体开始执行的位置调用，如果有初始化列表，初始化列表会在父类构造函数执行之前执行。

```
classPerson{

  String name;

  int age;



Person(this.name,this.age);//指定了带参数的构造函数为主构造函数，那么父类中就
没有默认无参构造函数}

classStudentextendsPerson{

final String classRoom;



Student(String name, int age, String room): classRoom = room,super(name,
 age){//注意 super(name, age)必须位于初始化列表尾部

print('Student');

}}

main(){

var student =Student('mikyou',18,'三年级八班');}
```

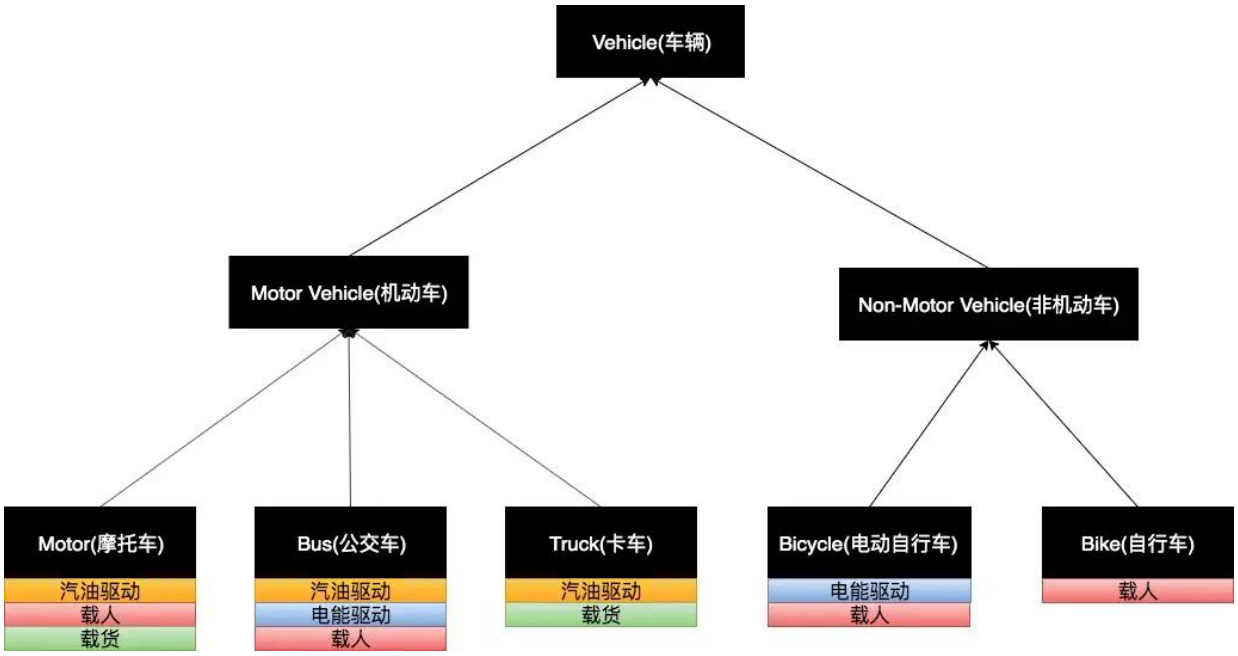
二、基于 Mixins 的多继承

除了上面和其他语言类似的单继承外，在 Dart 中还提供了另一继承的机制就是基于 **Mixins 的多继承**，但是它**不是真正意义上类的多继承，它始终还是只能有一个超类(基类)**。

1、为什么需要 Mixins?

为什么需要 Mixins 多继承？它实际上为了解决单继承所带来的问题，我们很多语言中都是采用**了单继承+接口多实现**的方式。但是这种方式并不能很好适用于所有场景。

假设一下下面场景，我们把车进行分类，然后下面的颜色条表示各种车辆具有的能力。



我们通过上图就可以看到，这些车辆都有一个共同的父类 `Vehicle`,然后它又由两个抽象的子类: `MotorVehicle` 和 `NonMotorVehicle` 。有些类是具有相同的行为和能力，但是有的类又有自己独有的行为和能力。比如公交车 `Bus` 和摩托车 `Motor` 都能使用汽油驱动，但是摩托车 `Motor` 还能载货公交车 `Bus` 却不可以。

如果**仅仅是单继承模型下，无法把部分子类具有相同行为和能力抽象放到基类，因为对于不具有该行为和能力的子类来说是不妥的，所以只能在各自子类另外实现**。那么就问题来了，部分具有相同能力和行为的子类中都要保留一份相同的代码实现。这就是产生冗余，突然觉得单继承模型有点鸡肋，食之无味弃之可惜。

```
//单继承模型的普通实现 abstractclassVehicle{}

abstractclassMotorVehicleextendsVehicle{}

abstractclassNonMotorVehicleextendsVehicle{}

classMotorextendsMotorVehicle{

voidpetrolDriven()=>print("汽油驱动");



voidpassengerService()=>print('载人');



voidcarryCargo()=>print('载货');}

classBusextendsMotorVehicle{

voidpetrolDriven()=>print("汽油驱动");



voidelectricalDriven()=>print("电能驱动");



voidpassengerService()=>print('载人');}

classTruckextendsMotorVehicle{
```

```
voidpetrolDriven()=>print("汽油驱动");



voidcarryCargo()=>print('载货');}

classBicycleextendsNonMotorVehicle{

voidelectricalDriven()=>print("电能驱动");



voidpassengerService()=>print('载人');}

classBikeextendsNonMotorVehicle{

voidpassengerService()=>print('载人');}
```

可以从上述实现代码来看发现，有很多相同冗余代码实现，请注意这里所说的相同代码是连具体实现都相同的。很多人估计想到一个办法那就是将各个能力提升成接口，然后各自的选择去实现相应能力。但是我们知道即使抽成了接口，各个实现类中还是需要写对应的实现代码，冗余还是无法摆脱。不妨我们来试试用接口:

```
//单继承+接口多实现 abstractclassVehicle{}

abstractclassMotorVehicleextendsVehicle{}

abstractclassNonMotorVehicleextendsVehicle{}

//将各自的能力抽成独立的接口，这样的好处就是可以从抽象角度对不同的实现类赋予不同接口能力，// 职责更加清晰,但是这个只是一方面的问题，它还是无法解决相同能力实现代码冗余的问题 abstractclassPetrolDriven{

voidpetrolDriven();}

abstractclassPassengerService{

voidpassengerService();}

abstractclassCargoService{
```

```
voidcarryCargo();}

abstractclassElectricalDriven{

voidelectricalDriven();}

//对于 Motor 赋予了 PetrolDriven、PassengerService、CargoService 能力 classM
otorextendsMotorVehicleimplementsPetrolDriven, PassengerService, CargoSe
rvice {

@override

voidcarryCargo()=>print('载货');//仍然需要重写 carryCargo



@override

voidpassengerService()=>print('载人');//仍然需要重写 passengerService



@override

voidpetrolDriven()=>print("汽油驱动");//仍然需要重写 petrolDriven}

//对于 Bus 赋予了 PetrolDriven、ElectricalDriven、PassengerService 能力 clas
sBusextendsMotorVehicleimplementsPetrolDriven, ElectricalDriven, Passeng
erService {

@override

voidelectricalDriven()=>print("电能驱动");//仍然需要重写 electricalDriven



@override

voidpassengerService()=>print('载人');//仍然需要重写 passengerService



@override

voidpetrolDriven()=>print("汽油驱动");//仍然需要重写 petrolDriven}
```

```
//对于 Truck 赋予了 PetrolDriven、CargoService 能力 classTruckextendsMotorVe
hicleimplementsPetrolDriven, CargoService {

@override

voidcarryCargo()=>print('载货');//仍然需要重写 carryCargo



@override

voidpetrolDriven()=>print("汽油驱动");//仍然需要重写 petrolDriven}

//对于 Bicycle 赋予了 ElectricalDriven、PassengerService 能力 classBicycleex
tendsNonMotorVehicleimplementsElectricalDriven, PassengerService {

@override

voidelectricalDriven()=>print("电能驱动");//仍然需要重写 electricalDriven



@override

voidpassengerService()=>print('载人');//仍然需要重写 passengerService}

//对于 Bike 赋予了 PassengerService 能力 classBikeextendsNonMotorVehicleimpl
ementsPassengerService{

@override

voidpassengerService()=>print('载人');//仍然需要重写 passengerService}
```

针对相同实现代码冗余的问题，使用 Mixins 就能很好的解决。**它能复用类中某个行为的具体实现，而不是像接口仅仅从抽象角度规定了实现类具有哪些能力，至于具体实现接口方法都必须重写，也就意味着即使是相同的实现还得重新写一遍**。一起看下 Mixins 改写后代码:

```
//mixins 多继承模型实现 abstractclassVehicle{}
```

```
abstractclassMotorVehicleextendsVehicle{}

abstractclassNonMotorVehicleextendsVehicle{}

//将各自的能力抽成独立的 Mixin 类

mixin PetrolDriven {//使用 mixin 关键字代替 class 声明一个 Mixin 类

voidpetrolDriven()=>print("汽油驱动");}



mixin PassengerService {//使用 mixin 关键字代替 class 声明一个 Mixin 类

voidpassengerService()=>print('载人');}



mixin CargoService {//使用 mixin 关键字代替 class 声明一个 Mixin 类

voidcarryCargo()=>print('载货');}



mixin ElectricalDriven {//使用 mixin 关键字代替 class 声明一个 Mixin 类

voidelectricalDriven()=>print("电能驱动");}

classMotorextendsMotorVehiclewith PetrolDriven, PassengerService, CargoS
ervice {}//利用 with 关键字使用 mixin 类

classBusextendsMotorVehiclewith PetrolDriven, ElectricalDriven, Passenge
rService {}//利用 with 关键字使用 mixin 类

classTruckextendsMotorVehiclewith PetrolDriven, CargoService {}//利用 wit
h 关键字使用 mixin 类

classBicycleextendsNonMotorVehiclewith ElectricalDriven, PassengerServic
e {}//利用 with 关键字使用 mixin 类

classBikeextendsNonMotorVehiclewith PassengerService {}//利用 with 关键字
使用 mixin 类
```

可以对比发现 Mixins 类能真正地解决相同代码冗余的问题，并能实现很好的复用；所以使用 Mixins 多继承模型可以很好地解决单继承模型所带来冗余问题。

2、Mixins 是什么?

用 dart 官网一句话来概括: **Mixins 是一种可以在多个类层次结构中复用类代码的方式。**

基本语法

方式一: Mixins 类使用关键字 `mixin` 声明定义

```
mixin PetrolDriven {//使用 mixin 关键字代替 class 声明一个 Mixin 类

voidpetrolDriven()=>print("汽油驱动");}

classMotorextendsMotorVehiclewith PetrolDriven {//使用 with 关键字来使用 mi
xin 类

...}

classPetrolextendsPetrolDriven{//编译异常，注意:mixin 类不能被继承

...}

main(){

var petrolDriven =PetrolDriven()//编译异常，注意:mixin 类不能实例化}
```

方式二: Dart 中的普通类当作 Mixins 类使用

```
classPetrolDriven{

factory PetrolDriven._()=>null;//主要是禁止 PetrolDriven 被继承以及实例化

voidpetrolDriven()=>print("汽油驱动");}
```

```
classMotorextendsMotorVehiclewith PetrolDriven {//普通类也可以作为 Mixins 类使用

...}
```

3、使用 Mixins 多继承的场景

那么问题来了，什么时候去使用 Mixins 呢？

当想要在不同的类层次结构中多个类之间共享相同的行为时或者无法合适抽象出部分子类共同的行为到基类中时. 比如说上述例子中在 `MotorVehicle` (机动车)和 `Non-MotorVehicle` (非机动车)两个不同类层次结构中，其中 `Bus` (公交车)和 `Bicycle` (电动自行车)都有相同行为 `ElectricalDriven` 和 `PassengerService`. 但是很明显你无法把这个两个共同的行为抽象到基类 `Vehicle` 中，因为这样的话 `Bike` (自行车)继承 `Vehicle` 会自动带有一个 `ElectricalDriven` 行为就比较诡异。所以这种场景下 **mixins 就是一个不错的选择，可以跨类层次之间复用相同行为的实现**。

4、Mixins 的线性化分析

在说 Mixins 线性化分析之前，一起先来看个例子

```
classA{

voidprintMsg()=>print('A');}

mixin B {

voidprintMsg()=>print('B');}

mixin C {

voidprintMsg()=>print('C');}
```

```
classBCextendsAwith B, C {}classCBextendsAwith C, B {}

main(){

var bc =BC();

    bc.printMsg();



var cb =CB();

    cb.printMsg();}
```

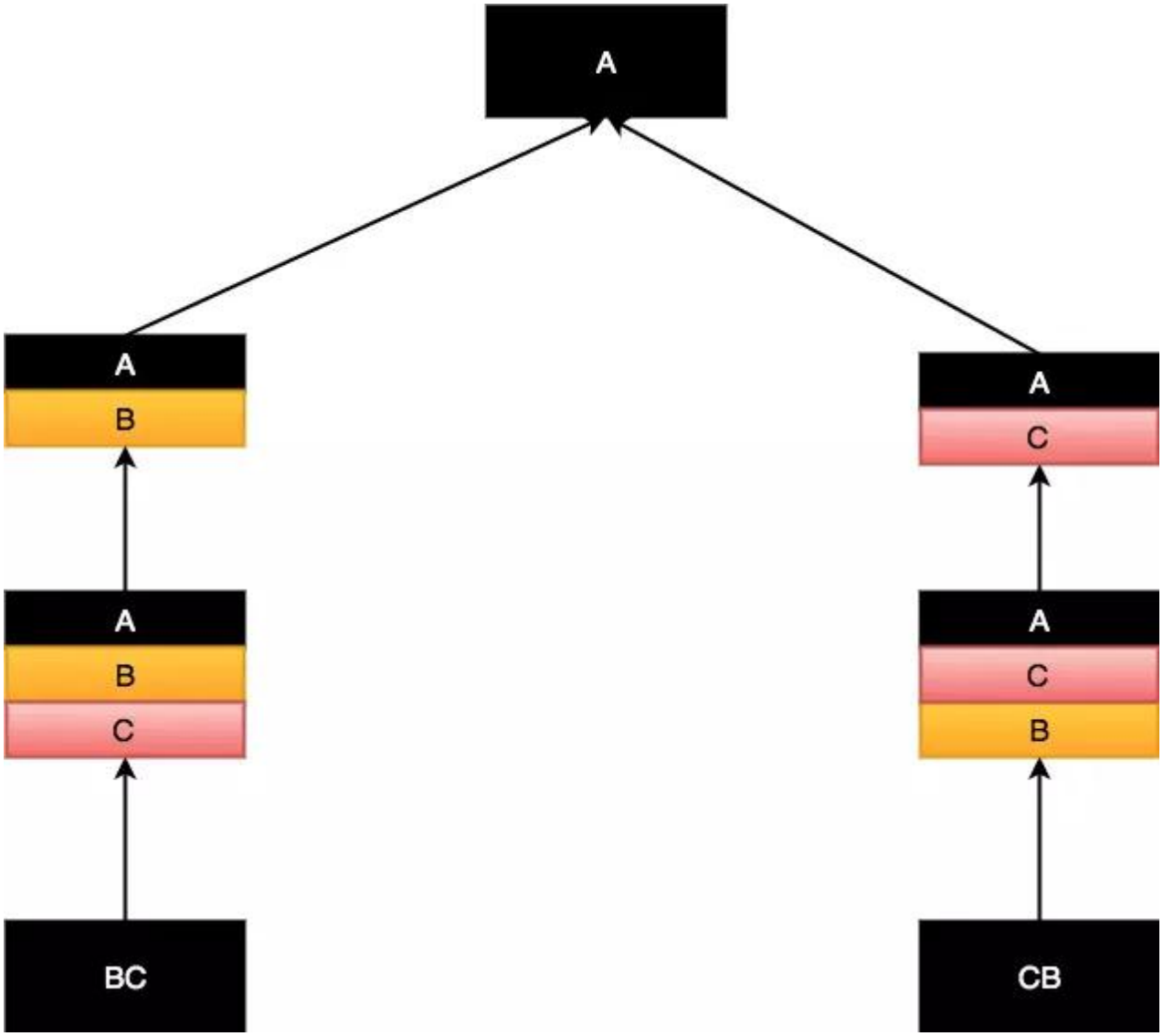
不妨考虑下上述例子中应该输出啥呢?

输出结果:

```
C

B



Process finished with exit code 0
```

为什么会是这样的结果？实际上可以通过线性分析得到输出结果。理解 Mixin 线性化分析有一点很重要就是: **在 Dart 中 Mixins 多继承并不是真正意义上的多继承，实际上还是单继承；而每次 Mixin 都是会创建一个新的中间类。并且这个中间类总是在基类的上层**。

关于上述结论可能有点难以理解，下面通过一张 mixins 继承结构图就能清晰明白了:



通过上图，我们可以很清楚发现 **Mixins 并不是经典意义上获得多重继承的方法。Mixins 是一种抽象和复用一系列操作和状态的方式，而是生成多个中间的 mixin 类**(比如生成 ABC 类，AB 类，ACB 类，AC 类)。它类似于从扩展类获得的复用，但**由于它是线性的，因此与单继承兼容**。

上述 mixins 代码在语义理解上可以转化成下面形式:

```
//A with B 会产生 AB 类混合体，B 类中 printMsg 方法会覆盖 A 类中的 printMsg 方法，那么 AB 中间类保留是 B 类中的 printMsg 方法 classAB=AwithB;//AB with C 会产生 ABC 类混合体,C 类中 printMsg 方法会覆盖 AB 混合类中的 printMsg 方法，那么 ABC 中间
```

```
类保留 C 类中 printMsg 方法 classABC=ABwithC;//最终 BC 类相当于继承的是 ABC 类混合体，最后调用的方法是 ABC 中间类保留 C 类中 printMsg 方法，最后输出 CclassBCextendsABC{}

//A with C 会产生 AC 类混合体，C 类中 printMsg 方法会覆盖 A 类中的 printMsg 方法，那么 AC 中间类保留是 C 类中的 printMsg 方法 classAC=AwithC;//AC with B 会产生 ACB 类混合体,B 类中 printMsg 方法会覆盖 AC 混合类中的 printMsg 方法，那么 ACB 中间类保留 B 类中 printMsg 方法 classACB=ACwithB;//最终 CB 类相当于继承的是 ACB 类混合体，最后调用的方法是 ACB 中间类保留 B 类中 printMsg 方法，最后输出 BclassCBextendsACB{}
```

5、Mixins 中的类型

有了上面的探索，不妨再来考虑一个问题，mixin 的实例对象是什么类型呢？我们都知道它肯定是它基类的子类型。 一起来看个例子：

```
classA{

voidprintMsg()=>print('A');}

mixin B {

voidprintMsg()=>print('B');}

mixin C {

voidprintMsg()=>print('C');}

classBCextendsAwith B, C {}classCBextendsAwith C, B {}

main(){

var bc =BC();

print(bc is A);

print(bc is B);

print(bc is C);
```

```
var cb =CB();

print(cb is A);

print(cb is B);

print(cb is C);}
```

输出结果:

```
true

true

true

true

true

true


Process finished with exit code 0
```

可以看到输出结果全都是 `true`，这是为什么呢？

其实通过上面那张图就能找到答案，我们知道 mixin with 后就会生成新的中间类，比如中间类 `AB、ABC`. 同时也会生成 一个新的接口 `AB、ABC`(因为所有 Dart 类都定义了对应的接口, Dart 类是可以直接当做接口实现的)。新的中间类 `AB` 继承基类 `A` 以及 `A` 类和 `B` 类混合的成员(方法和属性)副本，但它也实现了 `A` 接口和 `B` 接口。那么就很容易理解， `ABC` 混合类最后会实现了 `A`、`B`、`C` 三个接口， 最后 `BC` 类继承

ABC 类实际上也相当于间接实现了 A、B、C 三个接口，所以 BC 类肯定是 A、B、C 的子类型，故所有输出都是 true。

其实一般情况 mixin 中间类和接口都是不能直接引用的，比如这种情况:

```
classBCextendsAwith B, C {}//这种情况我们无法直接引用中间 AB 混合类和接口、ABC 混合类和接口
```

但是如果这么写，就能直接引用中间混合类和接口了:

```
classA{

voidprintMsg()=>print('A');}

mixin B {

voidprintMsg()=>print('B');}

mixin C {

voidprintMsg()=>print('C');}

classAB= A with B;

classABC= AB with C;

classDextendsAB{}

classEimplementsAB{

@override

voidprintMsg(){

// TODO: implement printMsg

}}

classFextendsABC{}

classGimplementsABC{
```

```
@override

voidprintMsg(){

// TODO: implement printMsg

}}

main(){

var ab =AB();

print(ab is A);

print(ab is B);



var e =E();

print(e is A);

print(e is B);

print(e is AB);



var abc =ABC();

print(abc is A);

print(abc is B);

print(abc is C);



var f =F();

print(f is A);

print(f is B);

print(f is C);
```

```
print(f is AB);

print(f is ABC);}
```

输出结果:

```
true

true

true

true

true

true

true

true

true

true

true

true

true


Process finished with exit code 0
```

参考资料

Dart: What are mixins?

总结

到这里，有关 dart 中面向对象以及 mixins 的内容就结束。这篇文章已经对 Mixins 原理进行了详细以及全面的分析，再不用担心啥时候使用 mixins 以及用起 mixin 来心里没底。面向对象结束，下一篇文章我们将进入 Dart 中的类型系统和泛型以及 Dart 未来版本将要支持的非空和可空类型。

第十二章 Dart 语法篇之类型系统与泛型(七)

简述：

下面开始 Dart 语法篇的第七篇类型系统和泛型，上一篇我们用了一篇 Dart 中可空和非空类型译文做了铺垫。实际上，Dart 中的类型系统是不够严格，这当然和它的历史原因有关。在 dart 最开始诞生之初，它的定位是一门像 javascript 一样的动态语言，动态语言的类型系统是比较松散的，所以在 Dart 类型也是可选的。然后动态语言类型系统松散对开发者并不是一件好事，程序逻辑一旦复杂，松散的类型可能就变得混乱，分析起来非常痛苦，但是有静态类型检查可以在编译的时候就快速定位问题所在。

其实，dart 类型系统不够严格，这一点不仅仅体现在可选类型上和还没有划分可空与非空类型上，甚至还体现 dart 中的泛型类型安全上，这一点我会通过对比 Kotlin 和 Dart 中泛型实现。你会发现 Dart 和 Kotlin 泛型安全完全走不是一个路子，而且 dart 泛型安全是不可靠的，但是也会发现 dart2.0 之后对这块做很大的改进。

一、可选类型

在 Dart 中的类型实际上是可选的，也就是在 Dart 中函数类型，参数类型，变量类型是可以直接省略的。

```
sum(a, b, c, d){//函数参数类型和返回值类型可以省略
return a + b + c + d;}
main(){
print('${sum(10, 12, 14, 12)}');//正常运行}
```

```
```

上述的 `sum` 函数既没有返回值类型也没有参数类型，可能有的人会疑惑如果 `sum` 函数最后一个形参传入一个 `String` 类型会是怎么样。

答案是: 静态类型检查分析正常但是编译运行异常。

```
sum(a, b, c, d){

return a + b + c + d;}

main(){

print('${sum(10, 12, 14, "12312")}');//静态检查类型检查正常，运行异常}

//运行结果

Unhandled exception:

type 'String'is not a subtype of type 'num' of 'other'//请先记住这个子类型不匹配异常问题，因为在后面会详细分析子类型的含义，而且 Dart、Flutter 开发中会经常看到这个异常。



Process finished with exit code 255
```

虽然，可选类型从一方面使得整个代码变得简洁以及具有动态性，但是从另一方面它会使得静态检查类型难以分析。但是这也使得 dart 中失去了基于类型**函数重载**特性。我们都知道函数重载是静态语言中比较常见的语法特性，可是在 dart 中是不支持的。比如在其他语言我们一般使用构造器重载解决多种方式构造对象的场景，但是 dart 不支持构造器重载，所以为了解决这个问题，Dart 推出了命名构造器的概念。那可选类型语法特性为什么会和函数重载特性冲突呢?

我们可以使用反证法，假设 dart 支持函数重载，那么可能就会有以下这段代码:

```
classIllegalCode{
```

```
overloaded(num data){



}

overloaded(List data){//假设支持函数重载，实际上这是非法的



}}

main(){

var data1 =100;

var data2 =["100"];

//由于 dart 中的类型是可选的，以下函数调用，根本就无法分辨下面代码实际上调用哪个
overloaded 函数。

overloaded(data1);

overloaded(data2);}
```

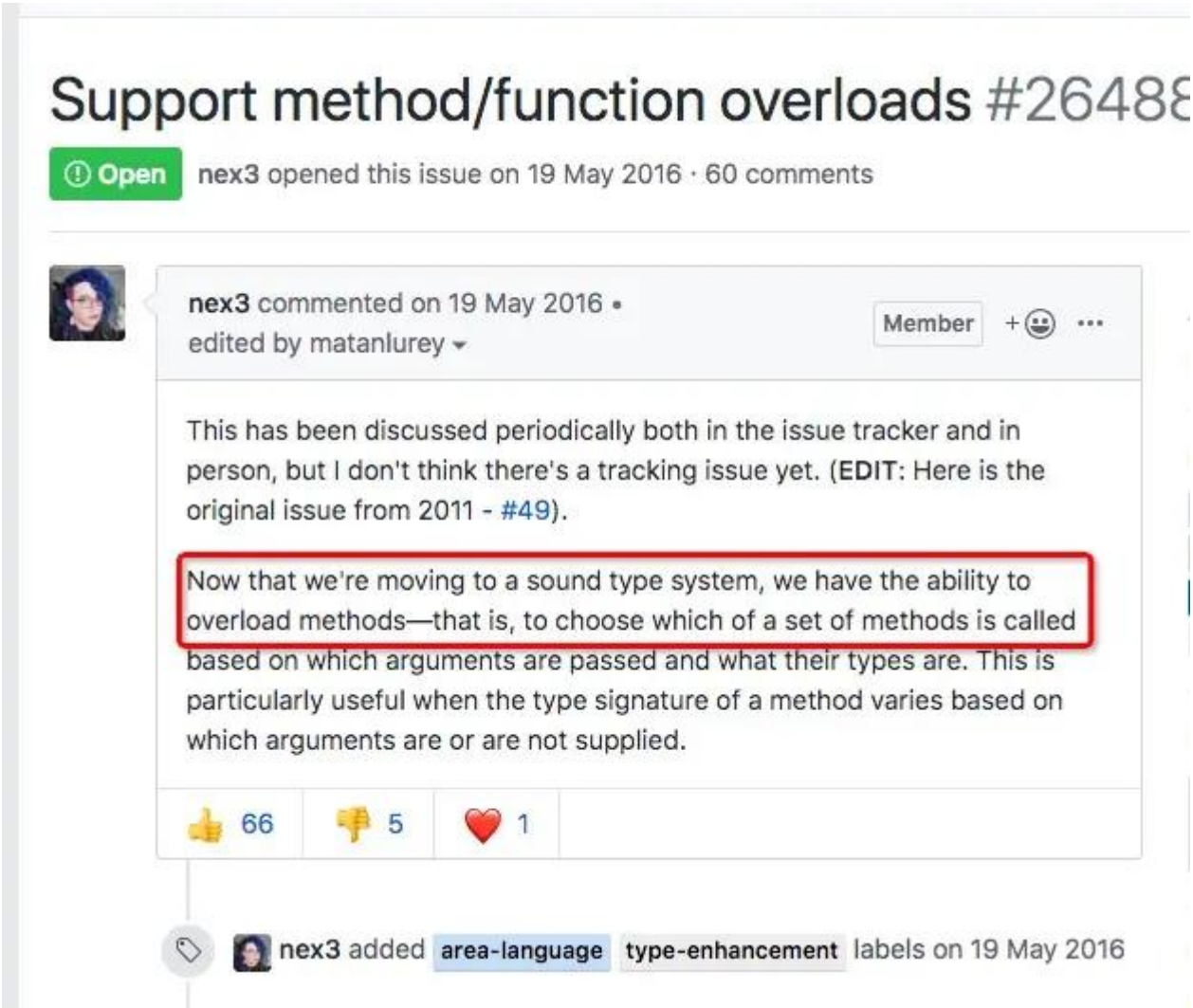
个人一些想法，如果仅从可选类型角度去考虑的话，实际上 dart 现在是可以支持基于类型的函数重载的，因为 Dart 有类型推导功能。如果 dart 能够推导出上述 data1 和 data2 类型，那么就可以根据推导出的类型去匹配重载的函数。Kotlin 就是这样做的，以 Kotlin 为例:

```
funoverloaded(data: Int){

//....}

funoverloaded(data: List<String>){

//....}

funmain(args: Array<String>){
```

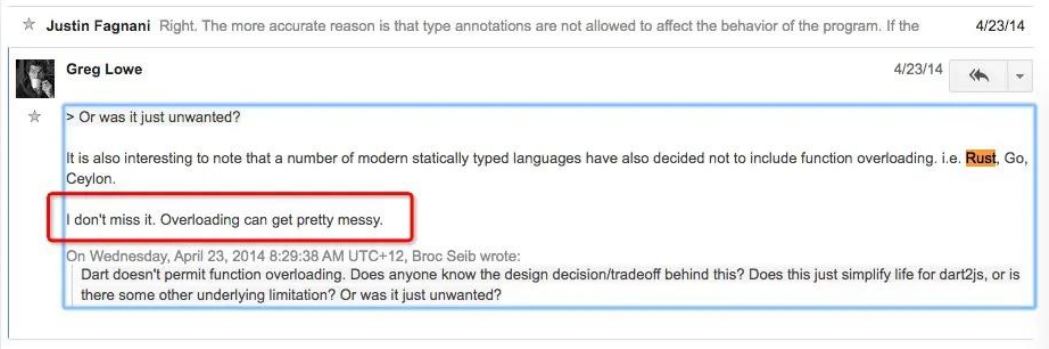
```
val data1 =100//这里 Kotlin 也是采用类型推导为 Int

val data2 =listOf("100")//这里 Kotlin 也是采用类型推导为 List<String>

//所以以下重载函数的调用在 Kotlin 中是合理的

overloaded(data1)

overloaded(data2)}
```

实际上，Dart 官方在 Github 提到过 Dart 迁移到新的类型系统中，Dart 是有能力支持函数重载的 。具体可以参考这个 dartlang 的

issue: github.com/dart-lang/s...



但是，dart 为什么不支持函数重载呢？其实，不是没有能力支持，而是没有必要的。其实在很多的现代语言比如 GO，Rust 中的都是没有函数重载。Kotlin 中也推荐使用默认值参数替代函数重载,感兴趣的可以查看我之前的一篇文章 juejin.im/post/5ac0da...。然而在 dart 中函数也是支持默认值参数的，其实函数重载更容易让人困惑，就比如 Java 中的 `Thread` 类中 7，8 个构造函数重载放在一起，让人就感到困惑。具体参考这个讨论: groups.google.com/a/dartlang....



二、接口类型

在 Dart 中没有直接显示声明接口的方法，没有类似 `interface` 的关键字来声明接口，而是隐性地通过类声明引入。所以每个类都存在一个对应名称隐性的接口，dart 中的类型也就是接口类型。

```
//定义一个抽象类 Person,同时它也是一个隐性的 Person 接口 abstractclassPerson{

final String name;

final int age;



Person(this.name,this.age);



get description =>"My name is $name, age is $age";}

//定义一个 Student 类，使用 implements 关键字实现 Person 接口 classStudentimpl
ementsPerson{

@override

// TODO: implement age

  int get age =>null;//重写 age getter 函数，由于在 Person 接口中是 final 修饰，
所以它只有 getter 访问器函数，作为接口实现就是需要重写它所有的函数，包括它的 get
ter 或 setter 方法。
```

```
@override

// TODO: implement description

get description =>null;//重写定义 description 方法



@override

// TODO: implement name

  String get name =>null;//重写 name getter 函数，由于在 Person 接口中是 fina
l 修饰，所以它只有 getter 访问器函数，作为接口实现就是需要重写它所有的函数，包括
它的 getter 或 setter 方法。}

//定义一个 Student2 类，使用 extends 关键字继承 Person 抽象类 classStudent2exte
ndsPerson{

Student2(String name, int age):super(name, age);//调用父类中的构造函数



@override

get description ="Student: ${super.description}";//重写父类中的descripti
on 方法}
```

三、泛型

1、泛型的基本介绍

Dart 中的泛型和其他语言差不多，但是 Dart 中的类型是可选的，使用泛型可以限定类型；使用泛型可以减少很多模板代码。

一起来看个例子:

```
//这是一个打印 int 类型 msg 的 PrintMsgclassPrintMsg{

int _msg;



  set msg(int msg){

this._msg = msg;

}



voidprintMsg(){

print(_msg);

}}

//现在又需要支持 String，double 甚至其他自定义类的 Msg，我们可能这么加 classMsg
{

  @override

  String toString(){

return"This is Msg";

}}

classPrintMsg{

int _intMsg;

  String _stringMsg;

double _doubleMsg;

  Msg _msg;



  set intMsg(int msg){
```

```
this._intMsg = msg;

}



  set stringMsg(String msg){

this._stringMsg = msg;

}



  set doubleMsg(double msg){

this._doubleMsg = msg;

}



  set msg(Msg msg){

this._msg = msg;

}



voidprintIntMsg(){

print(_intMsg);

}



voidprintStringMsg(){

print(_stringMsg);

}
```

```
voidprintDoubleMsg(){

print(_doubleMsg);

}



voidprintMsg(){

print(_msg);

}}

//但是有了泛型以后，我们可以把上述代码简化很多:classPrintMsg<T>{

  T _msg;



  set msg(T msg){

this._msg = msg;

}



voidprintMsg(){

print(_msg);

}}
```

补充一点 Dart 中可以指定实际的泛型参数类型，也可以省略。省略实际上就相当于指定了泛型参数类型为 `dynamic` 类型。

```
classTest{

  List<int> nums =[1,2,3,4];

  Map<String, int> maps ={'a':1,'b':2,'c':3,'d':4};
```

```
//上述定义可简写成如下形式,但是不太建议使用这种形式，仅在必要且适当的时候使用

  List nums =[1,2,3,4];

  Map maps ={'a':1,'b':2,'c':3,'d':4};

//上述定义相当于如下形式

  List<dynamic> nums =[1,2,3,4];

  Map<dynamic,dynamic> maps ={'a':1,'b':2,'c':3,'d':4};}
```

2、泛型的使用

类泛型的使用

```
//定义类的泛型很简单，只需要在类名后加：<T>；如果需要多个泛型类型参数，可以在尖括号中追加，用逗号分隔 classList<T>{

  T element;


voidadd(T element){

//...

}}
```

函数泛型的使用

```
//定义函数的泛型 voidadd(T elememt){//函数参数类型为泛型类型

//...}


T elementAt(int index){//函数参数返回值类型为泛型类型
```

```
//...}



E transform(R data){//函数参数类型和函数参数返回值类型均为泛型类型

//... }
```

集合泛型的使用

```
var list =<int>[1,2,3];//相当于如下形式

List<int> list =[1,2,3];

var map =<String, int>{'a':1,'b':2,'c':3};//相当于如下形式

Map<String, int> map ={'a':1,'b':2,'c':3};
```

泛型的上界限定

```
//和 Java 一样泛型上界限定可以使用 extends 关键字来实现 classList<Textends n
um>{

T element;

voidadd(T element){

//...

}}
```

3、子类、子类型和子类型化关系

泛型类与非泛型类

我们可以把 Dart 中的类可分为两大类: **泛型类**和**非泛型类**

先说**非泛型类**也就是开发中接触最多的一般类，一般的类去定义一个变量的时候，它的**类**实际就是这个变量的类型. 例如定义一个 Student 类，我们会得到一个 Student 类型

泛型类比非泛型类要更加复杂，实际上**一个泛型类可以对应无限种类型**。为什么这么说，其实很容易理解。我们从前面文章知道，在定义泛型类的时候会定义泛型形参，要想拿到一个合法的泛型类型就需要在外部使用地方传入具体的类型实参替换定义中的类型形参。我们知道在 Dart 中 `List` 是一个类，它不是一个类型。由它可以衍生成无限种泛型类型。例如 `List<String>`、`List<int>`、`List<List<num>>`、`List<Map<String,int>>`

何为子类型

我们可能会经常在 Flutter 开发中遇到 subtype 子类型的错误: `type 'String' is not a subtype of type 'num' of 'other'`. 到底啥是子类型呢？它和子类是一个概念吗?

首先给出一个数学归纳公式:

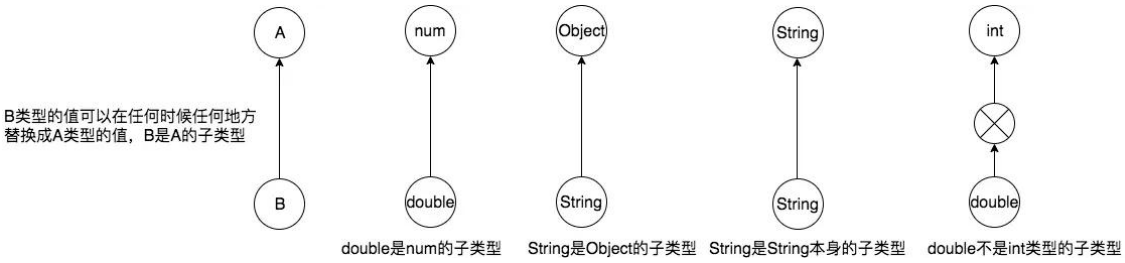
如果 G 是一个有 n 个类型参数的泛型类，而 A[i]是 B[i]的子类型且属于 1..n 的范围，那么可表示为 G<A[1],...,A[n]> * G<B[1],...,B[n]>的子类型，其中 A * B 可表示 A 是 B 的子类型。

上述是不是很抽象，其实 Dart 中的子类型概念和 Kotlin 中子类型概念极其相似。

我们一般说**子类**就是派生类，该类一般会继承它的父类(也叫基类)。例如: `class Student extends Person{//...}`,这里的 Student 一般称为 Person 的**子类**。

而**子类型**则不一样，我们从上面就知道一个类可以有很多类型，那么子类型不仅仅是想子类那样继承关系那么严格。子类型定义的规则一般是这样的: **任何时候**

如果需要的是 A 类型值的任何地方，都可以使用 B 类型的值来替换的，那么就可以说 B 类型是 A 类型的子类型或者称 A 类型是 B 类型的超类型。可以明显看出子类型的规则会比子类规则更为宽松。那么我们可以一起分析下面几个例子:



注意: 某个类型也是它自己本身的子类型，很明显 String 类型的值任意出现地方，String 肯定都是可以替换的。属于子类关系的一般也是子类型关系。像 double 类型值肯定不能替代 int 类型值出现的地方，所以它们不存在子类型关系.

子类型化关系:

如果 A 类型的值在任何时候任何地方出现都能被 B 类型的值替换，B 类型就是 A 类型的子类型，那么 B 类型到 A 类型之间这种映射替换关系就是子类型化关系

4、协变(covariant)

一提到协变，可能我们还会对应另外一个词那就是逆变，实际上在 Dart1.x 的版本中是既支持协变又支持逆变的，但是在 Dart2.x 版本仅仅支持协变的。有了子类型化关系的概念，那么协变就更好理解了，协变实际上就是**保留子类型化关系**，首先，我们需要去明确一下这里所说的保留子类型化关系是针对谁而言的呢?

比如说 `int` 是 `num` 的子类型，因为**在 Dart 中所有泛型类都默认是协变**的，所以 `List<int>` 就是 `List<num>` 的子类型，这就是保留了子类型化关系，保留的是泛型参数(`int` 和 `num`)类型的子类型化关系.

一起看个例子:

```
classFruit{

final String color;



Fruit(this.color);}

classAppleextendsFruit{

Apple():super("red");}

classOrangeextendsFruit{

Orange():super("orange");}

voidprintColors(List<Fruit> fruits){

for(var fruit in fruits){

print('${fruit.color}');

}}

main(){

  List<Apple> apples =<Apple>[];

  apples.add(Apple());

printColors(apples);//Apple 是 Fruit 的子类型，所以 List<Apple>是 List<Fruit>
子类型。

// 所以 printColors 函数接收一个 List<Fruit>类型，可以使用 List<Apple>类型替代

  List<Orange> oranges =<Orange>[];
```

```
  oranges.add(Orange());

printColors(oranges);//同理



  List<Fruit> fruits =<Fruit>[];

  fruits.add(Fruit('purple'));

printColors(fruits);//Fruit 本身也是 Fruit 的子类型，所以 List<Fruit>肯定是 L
ist<Fruit>子类型}
```

5、协变在 Dart 中的应用

实际上，在 Dart 中协变默认用于泛型类型实际上还有用于另一种场景**协变方法参数类型**. 可能对专业术语有点懵逼，先通过一个例子来看下:

```
//定义动物基类 classAnimal{

final String color;



Animal(this.color);}

//定义 Cat 类继承 AnimalclassCatextendsAnimal{

Cat():super('black cat');}

//定义 Dog 类继承 AnimalclassDogextendsAnimal{

Dog():super('white dog');}

//定义一个装动物的笼子类 classAnimalCage{

voidputAnimal(Animal animal){

print('putAnimal: ${animal.color}');
```

```
}}

//定义一个猫笼类 classCatCageextendsAnimalCage{

@override

voidputAnimal(Animal animal){//注意：这里方法参数是 Animal 类型

super.putAnimal(animal);

}}

//定义一个狗笼类 classDogCageextendsAnimalCage{

@override

voidputAnimal(Animal animal){//注意：这里方法参数是 Animal 类型

super.putAnimal(animal);

}}
```

我们需要去重写 putAnimal 方法，由于是继承自 AnimalCage 类，所以方法参数类型是 Animal.这会造成什么问题呢？一起来看下:

```
main(){

//创建一个猫笼对象

var catCage =CatCage();

//然后却可以把一条狗放进去，如果按照设计原理应该猫笼子只能 put 猫。

  catCage.putAnimal(Dog());//这行静态检查以及运行都可以通过。



//创建一个狗笼对象

var dogCage =DogCage();
```

```
//然后却可以把一条猫放进去，如果按照设计原理应该狗笼子只能 put 狗。

  dogCage.putAnimal(Cat());//这行静态检查以及运行都可以通过。}
```

其实对于上述的出现问题，我们更希望 putAnimal 的参数更具体些，为了解决上述问题你需要使用 `covariant` 协变关键字。

```
//定义一个猫笼类 classCatCageextendsAnimalCage{

@override

voidputAnimal(covariant Cat animal){//注意：这里使用 covariant 协变关键字 表示 CatCage 对象中的 putAnimal 方法只接收 Cat 对象

super.putAnimal(animal);

}}

//定义一个狗笼类 classDogCageextendsAnimalCage{

@override

voidputAnimal(covariant Dog animal){//注意：这里使用 covariant 协变关键字 表示 DogCage 对象中的 putAnimal 方法只接收 Dog 对象

super.putAnimal(animal);

}}//调用 main(){

//创建一个猫笼对象

var catCage =CatCage();

  catCage.putAnimal(Dog());//这时候这样调用就会报错，报错信息：Error: The argument type 'Dog' can't be assigned to the parameter type 'Cat'.}
```

为了进一步验证结论，可以看下这个例子:

```
typedefvoidPutAnimal(Animal animal);

classTestFunction{

voidputCat(covariant Cat animal){}//使用 covariant 协变关键字



voidputDog(Dog animal){}



voidputAnimal(Animal animal){}}

main(){

var function =TestFunction()

print(function.putCat is PutAnimal);//true 因为使用协变关键字

print(function.putDog is PutAnimal);//false

print(function.putAnimal is PutAnimal);//true 本身就是其子类型}
```

6、为什么 Kotlin 比 Dart 的泛型型变更安全

实际上 Dart 和 Java 一样，泛型型变都存在安全问题。以及 `List` 集合为例，`List` 在 Dart 中既是**可变的，又是协变的**，这样就会存在安全问题。然而 Kotlin 却不一样，在 Kotlin 把集合分为可变集合 `MutableList<E>`和只读集合 `List<E>`，其中 `List<E>`在 Kotlin 中就是**不可变的，协变的**，这样就不会存在安全问题。下面这个例子将对比 Dart 和 Kotlin 的实现:

Dart 中的实现

```
classFruit{
```

```
final String color;



Fruit(this.color);}

classAppleextendsFruit{

Apple():super("red");}

classOrangeextendsFruit{

Orange():super("orange");}

voidprintColors(List<Fruit> fruits){//实际上这里 List 是不安全的。

for(var fruit in fruits){

print('${fruit.color}');

}}

main(){

  List<Apple> apples =<Apple>[];

  apples.add(Apple());

printColors(apples);//printColors 传入是一个 List<Apple>，因为是协变的}
```

为什么说 printColors 函数中的 `List<Fruit>`是不安全的呢，外部 `main` 函数中传入的是一个 `List<Apple>`.所以 printColors 函数中的 `fruits` 实际上是一个 `List<Apple>`.

可是 `printColors` 这样改动呢?

```
voidprintColors(List<Fruit> fruits){//实际上这里 List 是不安全的。

  fruits.add(Orange());//静态检查都是通过的,Dart1.x 版本中运行也是可以通过的，但是好在 Dart2.x 版本进行了优化，
```

```
// 在 2.x 版本中运行是会报错的:type 'Orange' is not a subtype of type 'Apple
' of 'value'

// 由于在 Dart 中 List 都是可变的，在 fruits 中添加 Orange(),实际上是在 List<App
le>中添加 Orange 对象，这里就会出现安全问题。

for(var fruit in fruits){

print('${fruit.color}');

}}
```

Kotlin 中的实现

然而在 Kotlin 中的不会存在上面那种问题，Kotlin 对集合做了很细致的划分，分为可变与只读。只读且协变的泛型类型更具安全性。一起看下 Kotlin 怎么做到的。

```
openclassFruit(val color: String)

class Apple :Fruit("red")

class Orange :Fruit("orange")

funprintColors(fruits: List<Fruit>){

    fruits.add(Orange())//此处编译不通过，因为在 Kotlin 中只读集合 List<E>，没
有 add，remove 之类修改集合的方法只有读的方法，

//所以它不会存在 List<Apple>中还添加一个 Orange 的情况出现。

for(fruit in fruits){

println(fruit.color)

}}

funmain(){

val apples =listOf(Apple())
```

```
printColors(apples)}
```

四、类型具体化

1、类型检测

在 Dart 中一般使用 `is` 关键字做类型检测，这一点和 Kotlin 中是一致的，如果判断不是某个类型 dart 中使用 **is!**，而在 Kotlin 中正好相反则用 **!is** 表示。类型检测就是对表达式结果值的动态类型与目标类型做对比测试。

```
main(){

var apples =[Apple()];

print(apples is List<Apple>);}
```

2、强制类型转化

强制类型转换在 Dart 中一般使用 **as** 关键字，这一点也和 Kotlin 中是一致的。强制类型转换是对一个表达式的值转化目标类型，如果转化失败就会抛出 `CastError` 异常。

```
Object o =[1,2,3];

o as List;

o as Map;//抛出异常
```

五、总结

到这里我们就把 Dart 中的类型系统和泛型介绍完毕了，相信这篇文章将会使你对 Dart 中的类型系统有一个更全面的认识。其实通过 Dart 中泛型，就会发现 Dart2.x 真的优化很多东西，比如泛型安全的问题，虽然静态检查能通过但是运行无法通过，换做 Dart1.x 运行也是可以通过的。Dart2.x 将会越来越严谨越来越完善，说明 Dart 在改变这是一件好事，一起期待它的更多特性。

第十三章 Flutter 中的 widget

01.Flutter 页面-基础 Widget

在 Flutter 中，几乎所有的对象都是一个 Widget，与原生开发中的控件不同的是，Flutter 中的 widget 的概念更广泛，它不仅可以表示 UI 元素，也可以表示一些功能性的组件如：用于手势检测的 GestureDetector widget、用于应用主题数据传递的 Theme 等等。由于 Flutter 主要就是用于构建用户界面的，所以，在大多数时候，可以认为 widget 就是一个控件，不必纠结于概念。

Widget 的功能是“描述一个 UI 元素的配置数据”，Widget 其实并不是表示最终绘制在设备屏幕上的显示元素，而只是显示元素的一个配置数据。实际上，Flutter 中真正代表屏幕上显示元素的类是 Element，

也就是说 Widget 只是描述 Element 的一个配置。一个 Widget 可以对应多个 Element，这是因为同一个 Widget 对象可以被添加到 UI 树的不同部分，而真正渲染时，UI 树的每一个节点都会对应一个 Element 对象。

02.Widget

StatelessWidget 和 StatefulWidget 是 flutter 的基础组件，日常开发中自定义 Widget 都是选择继承这两者之一。也是在往后的开放中，我们最多接触的 Widget：

StatelessWidget :无状态的，展示信息，面向那些始终不变的 UI 控件；

StatefulWidget：有状态的，可以通过改变状态使得 UI 发生变化，可以包含用户交互(比如弹出一个 dialog)。

在实际使用中，Stateless 与 Stateful 的选择需要取决于这个 Widget 是有状态还是无状态，简单来说看界面是否需要更新。

03.StatelessWidget

StatelessWidget 用于不需要维护状态的场景，它通常在 build 方法中通过嵌套其它 Widget 来构建 UI，在构建过程中会递归的构建其嵌套的 Widget。

BuildContext 表示构建 widget 的上下文，它是操作 widget 在树中位置的一个句柄，它包含了一些查找、遍历当前 Widget 树的一些方法。每一个 widget 都有一个自己的 context 对象。

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



void main()=> runApp(StatelessApp());



class StatelessApp extends StatelessWidget{



///在 build 方法中通过嵌套其它 Widget 来构建 UI，在构建过程中会递归的构建其嵌套的 Widget



@override



Widget build(BuildContext context){



//嵌套 MaterialApp：封装了应用程序实现 Material Design 所需要的一些 widget
```

```
returnMaterialApp(title:"Widget 演示",//标题,显示在 recent 时候的标题



//主页面



//Scaffold : Material Design 布局结构的基本实现。

    home:Scaffold(

        //ToolBar/ActionBar

        appBar:AppBar(title:Text("Widget")),

        body:Text("Hello,Flutter!"),

      )

   );

 }

}
```

Material Design:

一种设计语言，Material Design 于 2014 年的 Google I/O 首次亮相，是谷歌推出的全新的设计语言。说白了，就是一种设计风格。

StatefulWidget

StatefulWidget 是动态的，添加了一个新的接口 createState()用于创建和 Stateful widget 相关的状态 State，它在 Stateful widget 的生命周期中可能会被多次调用。

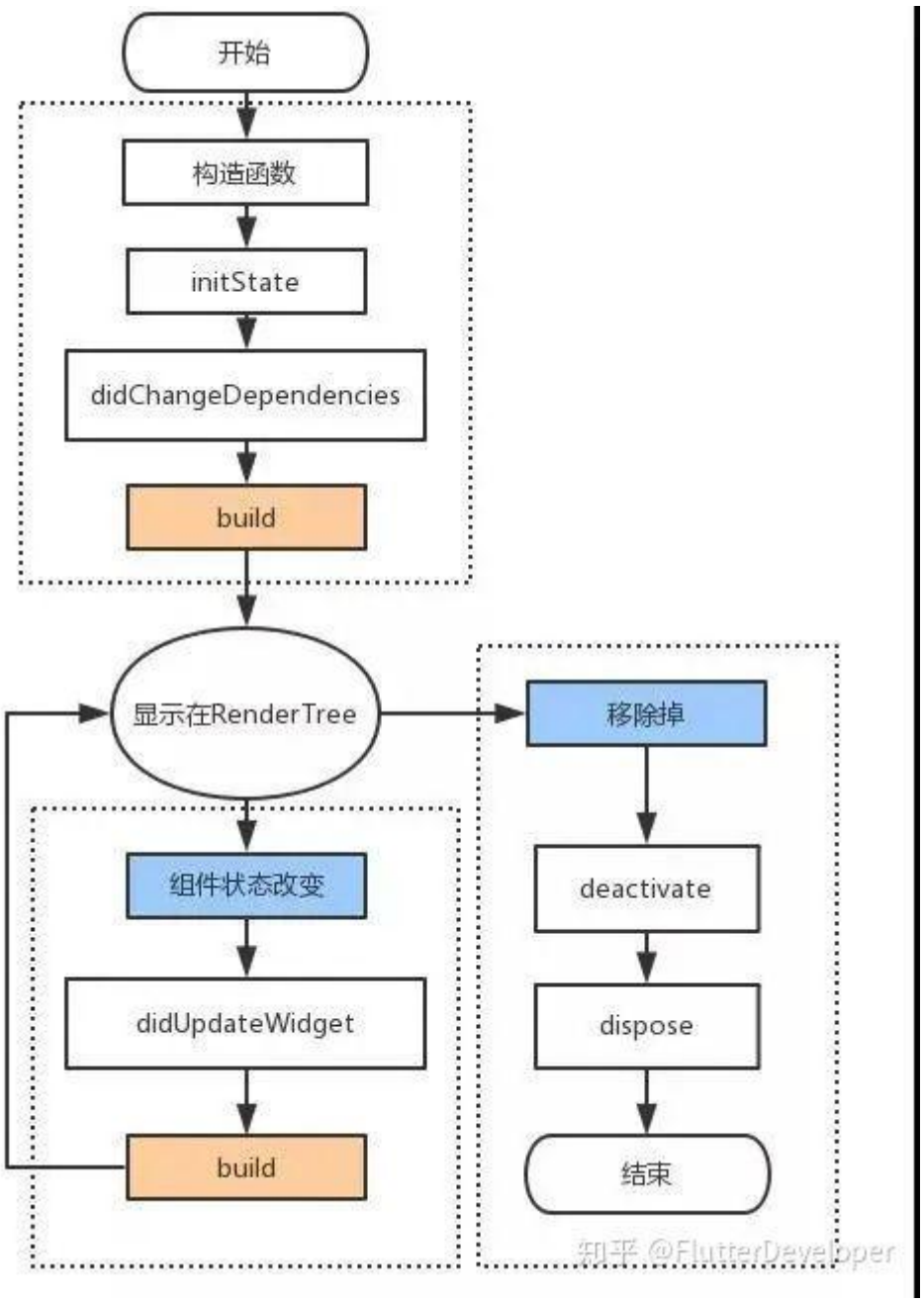
当 State 被改变时，可以手动调用其 setState()方法通知 Flutter framework 状态发生改变，Flutter framework 在收到消息后，会重新调用其 build 方法重新构建 widget 树，从而达到更新 UI 的目的。

```dart
class StatefulState extends State<StatefulApp> {
  int _i;

  ///当Widget第一次插入到Widget树时会被调用，对于每一个State对象，Flutter framework只会调用一
  @override
  void initState() {
    super.initState();
    _i = 1;
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        title: "Widget演示",
        theme: ThemeData(),
        home: Scaffold(
          appBar: AppBar(title: Text("Widget")),
          body: RaisedButton(
            onPressed: () {
              //修改状态，setState会重新调用build更新ui
              setState(() {
                _i++;
              });
            },
            child: Text("Hello,Flutter! $_i"),
          ),
        ));
  }
}
```

知乎 @FlutterDeveloper

04. State 生命周期

State 的生命周期为:



State 类除了 build 之外还提供了很多方法能够让我们重写，这些方法会在不同的状态下由 Flutter 调起执行，所以这些方法我们就称之为生命周期方法。在这里我们用 statefulwidget 点击按钮后移除子 statefulwidget。

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

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

class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  bool isShowChild;

  ///当Widget第一次插入到Widget树时会被调用，对于每一个State对象，Flutter framework只会调用
  @override
  void initState() {
    super.initState();
    isShowChild = true;
    debugPrint("parent initState......");
  }

  ///初始化时，在initState()之后立刻调用
  ///当依赖的InheritedWidget rebuild,会触发此接口被调用
  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    debugPrint("parent didChangeDependencies......");
  }

  ///绘制界面，当setState触发的时候会再次被调用
  @override
  Widget build(BuildContext context) {
    debugPrint("parent build......");
    return MaterialApp(
      home: Scaffold(
        body: Center(
            child: RaisedButton(
          onPressed: () {
            setState(() {
              isShowChild = !isShowChild;
            });
          },
          child: isShowChild ? Child() : Text("演示移除Child"),
        )),
      ),
    );
  }
```


更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```
  ///状态改变的时候会调用该方法,比如调用了setState
  @override
  void didUpdateWidget(MyApp oldWidget) {
    super.didUpdateWidget(oldWidget);
    debugPrint("parent didUpdateWidget......");
  }

  ///当State对象从树中被移除时，会调用此回调
  @override
  void deactivate() {
    super.deactivate();
    debugPrint('parent deactivate......');
  }

  ///当State对象从树中被永久移除时调用；通常在此回调中释放资源
  @override
  void dispose() {
    super.dispose();
    debugPrint('parent dispose......');
  }
}

class Child extends StatefulWidget {
  @override
  _ChildState createState() => _ChildState();
}

class _ChildState extends State<Child> {
  @override
  Widget build(BuildContext context) {
    debugPrint("child build......");
    return Text('lifeCycle');
  }

  @override
  void initState() {
    super.initState();
    debugPrint("child initState......");
  }

  ///初始化时，在initState()之后立刻调用
  ///当依赖的InheritedWidget rebuild,会触发此接口被调用
  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    debugPrint("child didChangeDependencies......");
  }
```


更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```dart
  ///父widget状态改变的时候会调用该方法,比如父节点调用了setState
  @override
  void didUpdateWidget(Child oldWidget) {
    super.didUpdateWidget(oldWidget);
    debugPrint("child didUpdateWidget.....");
  }

  ///当State对象从树中被移除时，会调用此回调
  @override
  void deactivate() {
    super.deactivate();
    debugPrint('child deactivate.....');
  }

  ///当State对象从树中被永久移除时调用；通常在此回调中释放资源
  @override
  void dispose() {
    super.dispose();
    debugPrint('child dispose.....');
  }
}
```

知乎 @FlutterDeveloper

执行的输出结果显示为:

运行到显示

```
I/flutter (22218): parent initState.....
I/flutter (22218): parent didChangeDependencies.....
I/flutter (22218): parent build.....
I/flutter (22218): child initState.....
I/flutter (22218): child didChangeDependencies.....
I/flutter (22218): child build.....
```

知乎 @FlutterDeveloper

点击按钮会移除 Child

```
I/flutter (22218): parent build......
I/flutter (22218): child deactivate......
I/flutter (22218): child dispose......
```

将 MyApp 的代码由 child:isShowChild?Child():Text("演示移除 Child")，改为 child:Child()，点击按钮时

```
I/flutter (22765): parent build......
I/flutter (22765): child didUpdateWidget......
I/flutter (22765): child build......
```

05.基础 widget

文本显示

Text

Text 是展示单一格式的文本 Widget(Android TextView)。

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

///
/// main方法 调用runApp传递Widget，这个Widget成为widget树的根
void main() => runApp(TextApp());

///
/// 1、单一文本Text
///
//创建一个无状态的Widget
class TextApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    //封装了应用程序实现Material Design所需要的一些widget
    return MaterialApp(
      title: "Text演示", //标题,显示在recent时候的标题
      //主页面
      //Scaffold : Material Design布局结构的基本实现。
      home: Scaffold(
        //ToolBar/ActionBar
        appBar: AppBar(title: Text("Text")),
        body: Text("Hello,Flutter"),
      ),
    );
  }
}
```

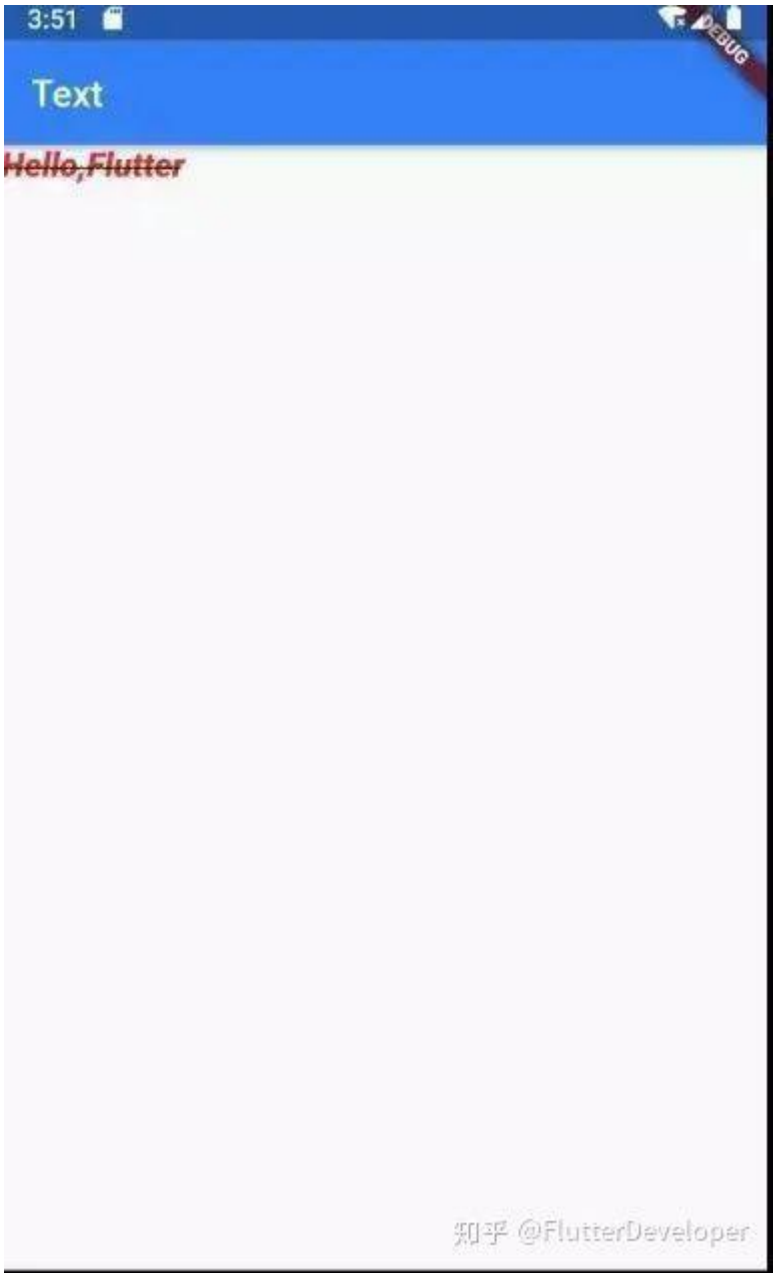
知乎 @FlutterDeveloper

在使用 Text 显示文字时候，可能需要对文字设置各种不同的样式，类似 Android 的 android:textColor/Size 等

在 Flutter 中也拥有类似的属性

```dart
Widget _TextBody() {
  return Text(
    "Hello,Flutter",
    style: TextStyle(
        //颜色
        color: Colors.red,
        //字号 默认14
        fontSize: 18,
        //粗细
        fontWeight: FontWeight.w800,
        //斜体
        fontStyle: FontStyle.italic,
        //underline: 下划线，overline: 上划线，lineThrough: 删除线
        decoration: TextDecoration.lineThrough,
        decorationColor: Colors.black,
        //solid: 实线，double: 双线，dotted: 点虚线，dashed: 横虚线，wavy: 波浪线
        decorationStyle: TextDecorationStyle.wavy),
  );
}

class TextApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Text演示",
      home: Scaffold(
        appBar: AppBar(title: Text("Text")),
        body: _TextBody(),
      ),
    );
  }
}
```

知乎 @FlutterDeveloper



RichText

如果需要显示更为丰富样式的文本(比如一段文本中文字不同颜色),可以使用 RichText 或者 Text.rich

```dart
Widget _RichTextBody() {
  var textSpan = TextSpan(
    text: "Hello",
    style: TextStyle(color: Colors.red),
    children: [
      TextSpan(text: "Flu", style: TextStyle(color: Colors.blue)),
      TextSpan(text: "uter", style: TextStyle(color: Colors.yellow)),
    ],
  );
  //Text.rich(textSpan);
  return RichText(text: textSpan);
}
```



```dart
Widget _RichTextBody() {
  var textSpan = TextSpan(
    text: "Hello",
    style: TextStyle(color: Colors.red),
    children: [
      TextSpan(text: "Flu", style: TextStyle(color: Colors.blue)),
      TextSpan(text: "uter", style: TextStyle(color: Colors.yellow)),
    ],
  );
  //Text.rich(textSpan);
  return RichText(text: textSpan);
}
```

知乎 @FlutterDeveloper

06DefaultTextStyle

在 widget 树中 ,文本的样式默认是可以被继承的 ,因此 如果在 widget 树的某一个节点处设置一个默认的文本样式 , 那么该节点的子树中所有文本都会默认使用这个样式。相当于在 Android 中定义 Theme

```dart
Widget _DefaultStyle(){
  DefaultTextStyle(
    //设置文本默认样式
    style: TextStyle(
      color:Colors.red,
      fontSize: 20.0,
    ),
    textAlign: TextAlign.start,
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: <Widget>[
        Text("Hello Flutter!"),
        Text("Hello Flutter!"),
        Text("Hello Flutter!",
          style: TextStyle(
              inherit: false, //不继承默认样式
              color: Colors.grey
          ),
        ),
      ],
    ),
  );
}
```


07FlutterLogo

这个 Widget 用于显示 Flutter 的 logo.....

```dart
Widget flutterLogo() {
  return FlutterLogo(
    //大小
    size: 100,
    //logo颜色 默认为 Colors.blue
    colors: Colors.red,
    //markOnly：只显示logo，horizontal：logo右边显示flutter文字，stacked：logo下面显示文字
    style: FlutterLogoStyle.stacked,
    //logo上文字颜色
    textColor: Colors.blue,
  );
}
```


08. Icon

主要用于显示内置图标的 Widget

```dart
Widget icon() {
  return Icon(
      //使用预定义Material icons
      // https://docs.flutter.io/flutter/material/Icons-class.html
      Icons.add,
      size: 100,
      color: Colors.red);
}
```


显示图片的 Widget。图片常用的格式主要有 bmp,jpg,png,gif,webp 等，Android 中并不是天生支持 gif 和 webp 动图，但是这一特性在 flutter 中被很好的支持了。

方式	解释
Image()	使用ImageProvider提供图片，如下方法本质上也是使用的这个方法
Image.asset	加载资源图片
Image.file	加载本地图片文件
Image.network	加载网络图片
Image.memory	加载内存图片

09. Iamge.asset

在工程目录下创建目录，如：assets，将图片放入此目录。打开项目根目录：pubspec.yaml

```
flutter:

  # The following line ensures that the Material Icons font i
  # included with your application, so that you can use the i
  # the material Icons class.
  uses-material-design: true

  # To add assets to your application, add an assets section,
  # assets:
  #   - images/a_dot_burr.jpeg
  #   - images/a_dot_ham.jpeg
  assets:
    - assets/
```



```
return MaterialApp(
      title: "Image演示",
      home: Scaffold(
        appBar: AppBar(title: Text("Image")),
        body: Image.asset("assets/banner.jpeg"),
      ),
    );
```

Image.file

在 sd 卡中放入一张图片。然后利用 path_provider 库获取 sd 卡根目录(Dart 库版本可以在：https://pub.dartlang.org/packages 查询)。

```
environment:
  sdk: ">=2.0.0-dev.68.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your applicatio
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^0.1.2
  path_provider: ^0.4.1

dev_dependencies:
  flutter_test:
    sdk: flutter

  build_runner: ^1.0.0
  json_serializable: ^2.0.0
```

知乎 @FlutterDeveloper

注意权限

```dart
class ImageState extends State<ImageApp> {
  Image image;

  @override
  void initState() {
    super.initState();
    getExternalStorageDirectory().then((path) {
      setState(() {
        image = Image.file(File("${path.path}${Platform.pathSeparator}banner.jpeg"));
      });
    });
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Image演示",
      home: Scaffold(
        appBar: AppBar(title: Text("Image")),
        body: image,
      ),
    );
  }
}
```


Image.network

直接给网络地址即可。

Flutter 1.0 ,加载 https 时候经常出现证书错误。必须断开 AS 打开 app

Image.memory

```dart
Future<List<int>> _imageByte() async {
  String path = (await getExternalStorageDirectory()).path;
  return await File("$path${Platform.pathSeparator}banner.jpeg").readAsBytes();
}

class ImageState extends State<ImageApp> {
  Image image;

  @override
  void initState() {
    super.initState();
    _imageByte().then((bytes) {
      setState(() {
        image = Image.memory(bytes);
      });
    });
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Image演示",
      home: Scaffold(
        appBar: AppBar(title: Text("Image")),
        body: image,
      ),
    );
  }
}
```


09. CircleAvatar

主要用来显示用户的头像，任何图片都会被剪切为圆形。

```dart
CircleAvatar(
      //图片提供者 ImageProvider
      backgroundImage: AssetImage("assets/banner.jpeg"),
      //半径，控制大小
      radius: 50.0,
    );
```

010. FadeInImage

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。如果最初显示一个占位符，然后在图像加载完显示时淡入，我们可以使用 FadeInImage 来达到这个目的！

```
image =  FadeInImage.memoryNetwork(
     placeholder: kTransparentImage,
     image: 'https://flutter.io/images/homepage/header-illustration.png',
   );
```

011 按钮

Material widget 库中提供了多种按钮 Widget 如 RaisedButton、FlatButton、OutlineButton 等，它们都是直接或间接对 RawMaterialButton 的包装定制，所以他们大多数属性都和 RawMaterialButton 一样。所有 Material 库中的按钮都有如下相同点：

按下时都会有“水波动画”。

有一个 onPressed 属性来设置点击回调，当按钮按下时会执行该回调，如果不提供该回调则按钮会处于禁用状态，禁用状态不响应用户点击。

RaisedButton

"漂浮"按钮，它默认带有阴影和灰色背景

```
RaisedButton(
          child: Text("normal"),
          onPressed: () => {},
        )
```

012. FlatButton

扁平按钮，默认背景透明并不带阴影

```
FlatButton(
  child: Text("normal"),
  onPressed: () => {},
)
```

013. OutlineButton

默认有一个边框，不带阴影且背景透明。

```
OutlineButton(
  child: Text("normal"),
  onPressed: () => {},
)
```

IconButton

可点击的 Icon

```
IconButton(
  icon: Icon(Icons.thumb_up),
  onPressed: () => {},
)
```

按钮外观可以通过其属性来定义，不同按钮属性大同小异

```
const FlatButton({
  ...
  @required this.onPressed, //按钮点击回调
  this.textColor, //按钮文字颜色
  this.disabledTextColor, //按钮禁用时的文字颜色
  this.color, //按钮背景颜色
  this.disabledColor,//按钮禁用时的背景颜色
  this.highlightColor, //按钮按下时的背景颜色
  this.splashColor, //点击时，水波动画中水波的颜色
  this.colorBrightness,//按钮主题，默认是浅色主题
  this.padding, //按钮的填充
  this.shape, //外形
  @required this.child, //按钮的内容
})

FlatButton(
  onPressed: () => {},
  child: Text("Raised"),
  //蓝色
  color: Colors.blue,
  //水波
  splashColor: Colors.yellow,
  //深色主题，这样文字颜色会变成白色
  colorBrightness: Brightness.dark,
   //圆角按钮
  shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(50)
  ),
)
```

知乎 @FlutterDeveloper

而 RaisedButton，默认配置有阴影，因此在配置 RaisedButton 时，拥有一系列 elevation 属性的配置

```
const RaisedButton({
  ...
  this.elevation = 2.0, //正常状态下的阴影
  this.highlightElevation = 8.0,//按下时的阴影
  this.disabledElevation = 0.0,// 禁用时的阴影
  ...
}
```

知乎 @FlutterDeveloper

输入框

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

void main() => runApp(Demo1());

class Demo1 extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: "Demo1",
      home: Scaffold(
        appBar: AppBar(
          title: Text("登录"),
        ),
        //线性布局，垂直方向
        body: Column(
          children: <Widget>[
            TextField(
              //自动获得焦点
              autofocus: true,
              decoration: InputDecoration(
                  labelText: "用户名",
                  hintText: "用户名或邮箱",
                  prefixIcon: Icon(Icons.person)),
            ),
            TextField(
              //隐藏正在编辑的文本
              obscureText: true,
              decoration: InputDecoration(
                  labelText: "密码",
                  hintText: "您的登录密码",
                  prefixIcon: Icon(Icons.lock)),
            ),
          ],
        ),
      ),
    );
  }
}
```

这个效果非常的“系统”，我们可能大多数情况下需要将下划线更换为矩形边框，这时候可能就需要组合 widget 来完成:

```dart
//容器 设置一个控件的尺寸、背景、margin
Container(
    margin: EdgeInsets.all(32),
    child: TextField(
        keyboardType: TextInputType.emailAddress,
        decoration: InputDecoration(
            labelText: "用户名",
            hintText: "用户名或邮箱",
            prefixIcon: Icon(Icons.person),
            border: InputBorder.none //隐藏下划线
            )),
    //装饰
    decoration: BoxDecoration(
      // 边框浅灰色，宽度1像素
      border: Border.all(color: Colors.red[200], width: 1.0),
      //圆角
      borderRadius: BorderRadius.circular(5.0),
    ),
  )
```



```dart
//容器 设置一个控件的尺寸、背景、margin
Container(
    margin: EdgeInsets.all(32),
    child: TextField(
        keyboardType: TextInputType.emailAddress,
        decoration: InputDecoration(
            labelText: "用户名",
            hintText: "用户名或邮箱",
            prefixIcon: Icon(Icons.person),
            border: InputBorder.none //隐藏下划线
            )),
    //装饰
    decoration: BoxDecoration(
      // 边框浅灰色，宽度1像素
      border: Border.all(color: Colors.red[200], width: 1.0),
      //圆角
      borderRadius: BorderRadius.circular(5.0),
    ),
  )
```

知乎 @FlutterDeveloper

焦点控制

FocusNode: 与 Widget 绑定，代表了这个 Widget 的焦点

FocusScope: 焦点控制范围

FocusScopeNode：控制焦点

```dart
class _TextFocusState extends State<TextFocusWidget> {
  FocusNode focusNode1 = new FocusNode();
  FocusNode focusNode2 = new FocusNode();

  void _listener() {
    debugPrint("用户名输入框焦点:${focusNode1.hasFocus}");
  }

  @override
  void initState() {
    super.initState();
    //监听焦点状态改变事件
    focusNode1.addListener(_listener);
  }

  @override
  void dispose() {
    super.dispose();
    focusNode1.dispose();
    focusNode2.dispose();
  }
```

知乎 @FlutterDeveloper

更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246

```dart
@override
Widget build(BuildContext context) {
  return Column(
    children: <Widget>[
      TextField(
        autofocus: true,
        //关联焦点
        focusNode: focusNode1,
        //设置键盘动作为：下一步
        textInputAction: TextInputAction.next,
        //点击下一步执行回调
        onEditingComplete: () {
          //获得 context对应UI树的焦点范围 的焦点控制器
          FocusScopeNode focusScopeNode = FocusScope.of(context);
          //将焦点交给focusNode2
          focusScopeNode.requestFocus(focusNode2);
        },
        decoration: InputDecoration(
            labelText: "用户名",
            hintText: "用户名或邮箱",
            prefixIcon: Icon(Icons.person)),
      ),
      TextField(
        //隐藏正在编辑的文本
        obscureText: true,
        focusNode: focusNode2,
        decoration: InputDecoration(
            labelText: "密码",
            hintText: "您的登录密码",
            prefixIcon: Icon(Icons.lock)),
      ),
      custom(),
    ],
  );
}
}
```

知乎 @FlutterDeveloper

获取输入内容

获取输入内容有两种方式：

定义两个变量，用于保存用户名和密码，然后在 onChange 触发时，各自保存一下输入内容。

通过 controller 直接获取。

onChange 获得输入内容:

```
TextField(
      onChanged: (s) => debugPrint("ssss:$s"),
    )
```

controller 获取:

定义一个 controller：

```
//定义一个controller
TextEditingController _unameController=new TextEditingController();
```

然后设置输入框 controller：

```
TextField(
    controller: _unameController, //设置controller
    ...
)
```

通过 controller 获取输入框内容

```
debugPrint(_unameController.text)
```

014.TextFormField

TextFormField 比 TextField 多了一些属性，其中 validator 用于设置验证回调。在单独使用时与 TextField 没有太大的区别。当结合 From，利用 From 可以对输入框进行分组，然后进行一些统一操作(验证)

```dart
class _TextFocusState extends State<TextFocusWidget> {
  //全局key
  GlobalKey<FormState> _key = GlobalKey<FormState>();

  @override
  Widget build(BuildContext context) {
    return Form(
        //类似 id
        key: _key,
        child: Column(
          children: <Widget>[
            TextFormField(
                autofocus: true,
                decoration: InputDecoration(
                    labelText: "用户名",
                    hintText: "用户名或邮箱",
                    icon: Icon(Icons.person)),
                // 校验用户名
                validator: (v) {
                  return v.trim().length > 0 ? null : "用户名不能为空";
                }),
            TextFormField(
                decoration: InputDecoration(
                    labelText: "密码",
                    hintText: "您的登录密码",
                    icon: Icon(Icons.lock)),
                // 校验用户名
                validator: (v) {
                  return v.trim().length > 0 ? null : "密码不能为空";
                }),
            RaisedButton(
              onPressed: () {
                //Form所有TextFormField成功 返回true
                if (_key.currentState.validate()) {

                }
              },
              child: Text("提交"),
            )
          ],
        ));
  }
}
```


更多录播视频+架构学习资料免费领取请加 Android 开发高级技术交流群 QQ 群：892872246