本章目录

移动开发技术简介

原生开发与跨平台技术

原生开发

原生应用程序是指某一个移动平台（比如iOS或安卓）所特有的应用，使用相应平台支持的开发工具和语言，并直接调用系统提供的SDK API。比如Android原生应用就是指使用Java或Kotlin语言直接调用Android SDK开发的应用程序；而iOS原生应用就是指通过Objective-C或Swift语言直接调用iOS SDK开发的应用程序。原生开发有以下主要优势：

- 可访问平台全部功能（GPS、摄像头）；
- 速度快、性能高、可以实现复杂动画及绘制，整体用户体验好；

主要缺点：

- 平台特定，开发成本高；不同平台必须维护不同代码，人力成本随之变大；
- 内容固定，动态化弱，大多数情况下，有新功能更新时只能发版；

在移动互联网发展初期，业务场景并不复杂，原生开发还可以应对产品需求迭代。但近几年，随着物联网时代到来、移动互联网高歌猛进，日新月异，在很多业务场景中，传统的纯原生开发已经不能满足日益增长的业务需求。主要表现在：

- 动态化内容需求增大；当需求发生变化时，纯原生应用需要通过版本升级来更新内容，但应用上架、审核是需要周期的，这对高速变化的互联网时代来说是很难接受的，所以，对应用动态化(不发版也可以更新应用内容)的需求就变的迫在眉睫。
- 业务需求变化快，开发成本变大；由于原生开发一般都要维护Android、iOS两个开发团队，版本迭代时，无论人力成本，还是测试成本都会变大。

总结一下，纯原生开发主要面临动态化和开发成本两个问题，而针对这两个问题，诞生了一些跨平台的动态化框架。

跨平台技术简介

针对原生开发面临问题，人们一直都在努力寻找好的解决方案，而时至今日，已经有很多跨平台框架(注意，本书中所指的“跨平台”若无特殊说明，即特指Android和iOS两个平台)，根据其原理，主要分为三类：

- H5+原生（Cordova、Ionic、微信小程序）
- JavaScript开发+原生渲染 （React Native、Weex、快应用）
- 自绘UI+原生(QT for mobile、Flutter)

在接下来的章节中我们逐个来看看这三类框架的原理及优缺点。

Hybird技术简介

H5+原生混合开发

这类框架主要原理就是将APP的一部分需要动态变动的内容通过H5来实现，通过原生的网页加载控件WebView (Android)或WKWebView（ios）来加载（以后若无特殊说明，我们用WebView来统一指代android和ios中的网页加载控件）。这样一来，H5部分是可以随时改变而不用发版，动态化需求能满足；同时，由于h5代码只需要一次开发，就能同时在Android和iOS两个平台运行，这也可以减小开发成本，也就是说，h5部分功能越多，开发成本就越小。我们称这种h5+原生的开发模式为**混合开发 ** ，采用混合模式开发的APP我们称之为**混合应用**或**Hybrid APP** ，如果一个应用的大多数功能都是H5实现的话，我们称其为**Web APP** 。

目前混合开发框架的典型代表有：Cordova、Ionic 和微信小程序，值得一提的是微信小程序目前是在webview中渲染的，并非原生渲染，但将来有可能会采用原生渲染。

混合开发技术点

如之前所述，原生开发可以访问平台所有功能，而混合开发中，h5代码是运行在WebView中，而WebView实质上就是一个浏览器内核，其JavaScript依然运行在一个权限受限的沙箱中，所以对于大多数系统能力都没有访问权限，如无法访问文件系统、不能使用蓝牙等。所以，对于H5不能实现的功能，都需要原生去做。而混合框架一般都会在原生代码中预先实现一些访问系统能力的API， 然后暴露给WebView以供JavaScript调用，这样一来，WebView就成为了JavaScript与原生API之间通信的桥梁，主要负责JavaScript与原生之间传递调用消息，而消息的传递必须遵守一个标准的协议，它规定了消息的格式与含义，我们把依赖于WebView的用于在JavaScript与原生之间通信并实现了某种消息传输协议的工具称之为**WebView JavaScript Bridge**, 简称 **JsBridge**，它也是混合开发框架的核心。

示例：JavaScript调用原生API获取手机型号

下面我们以Android为例，实现一个获取手机型号的原生API供JavaScript调用。在这个示例中将展示JavaScript调用原生API的流程，读者可以直观的感受一下调用流程。我们选用笔者在Github上开源的dsBridge作为JsBridge来进行通信。dsBridge是一个支持同步调用的跨平台的JsBridge，此示例中只使用其同步调用功能。

1. 首先在原生中实现获取手机型号的API `getPhoneModel`

```
class JSAPI{
 @JavascriptInterface
 public Object getPhoneModel(Object msg){
   return Build.MODEL;
 }
}
```

2. 将原生API通过WebView注册到JsBridge中

```
import wendu.dsbridge.DWebView
...
//DWebView继承自WebView，由dsBridge提供
DWebView dwebView= (DWebView) findViewById(R.id.dwebview);
//注册原生API到JsBridge
dwebView.addJavascriptObject(new JsAPI(), null);
```

3. 在JavaScript中调用原生API

```
var dsBridge=require("dsbridge")
//直接调用原生API `getPhoneModel`
var model=dsBridge.call("getPhoneModel");
//打印机型
console.log(model);
```

上面示例演示了JavaScript调用原生API的过程，同样的，一般来说优秀的JsBridge也支持原生调用JavaScript，dsBridge也是支持的，如果您感兴趣，可以去github dsBridge项目主页查看。

现在，我们回头来看一下，混合应用无非就是在第一步中预先实现一系列API供JavaScript调用，让JavaScript有访问系统的能力，看到这里，我相信你也可以自己实现一个混合开发框架了。

总结

混合应用的优点是动态内容是H5，web技术栈，社区及资源丰富，缺点是性能不好，对于复杂用户界面或动画，webview不堪重任。

React Native和Weex

本篇主要介绍一下 **JavaScript开发+原生渲染**的跨平台框架原理。

React Native (简称RN)是Facebook于2015年4月开源的跨平台移动应用开发框架，是Facebook早先开源的JS框架 React 在原生移动应用平台的衍生产物，目前支持iOS和Android两个平台。RN使用Javascript语言，类似于HTML的JSX，以及CSS来开发移动应用，因此熟悉Web前端开发的技术人员只需很少的学习就可以进入移动应用开发领域。

由于RN和React原理相通，并且Flutter也是受React启发，很多思想也都是相通的，万丈高楼平地起，我们有必要深入了解一下React原理。React是一个响应式的Web框架，我们先了解一下两个重要的概念：Dom树与响应式编程。

DOM树与控件树

文档对象模型（Document Object Model，简称DOM），是W3C组织推荐的处理可扩展标志语言的标准编程接口，一种独立于平台和语言的方式访问和修改一个文档的内容和结构。换句话说，这是表示和处理一个HTML或XML文档的标准接口。简单来说，DOM就是文档树，与用户界面控件树对应，在前端开发中通常指HTML对应的渲染树，但广义的DOM也可以指Android中的XML布局文件对应的控件树，而术语**DOM操作**就是指直接来操作渲染树（或控件树）， 因此，可以看到其实DOM树和控件树是等价的概念，只不过前者常用语Web开发中，而后者常用于原生开发中。

响应式编程

React中提出一个重要思想：状态改变则UI随之自动改变，而React框架本身就是响应用户状态改变的事件而执行重新构建用户界面的工作，这就是典型的**响应式**编程范式，下面我们总结一下React中响应式原理：

- 开发者只需关注状态转移（数据），当状态发生变化，React框架会自动根据新的状态重新构建UI。
- React框架在接收到用户状态改变通知后，会根据当前渲染树，结合最新的状态改变，通过Diff算法，计算出树中变化的部分，然后只更新变化的部分（DOM操作），从而避免整棵树重构，提高性能。

值得注意的是，在第二步中，状态变化后React框架并不会立即去计算并渲染DOM树的变化部分，相反，React会在DOM的基础上建立一个抽象层，即**虚拟DOM**树，对数据和状态所做的任何改动，都会被自动且高效的同步到虚拟DOM，最后再批量同步到真实DOM中，而不是每次改变都去操作一下DOM。为什么不能每次改变都直接去操作DOM树？这是因为在浏览器中每一次DOM操作都有可能引起浏览器的重绘或回流：

1. 如果DOM只是外观风格发生变化，如颜色变化，会导致浏览器重绘界面。
2. 如果DOM树的结构发生变化，如尺寸、布局、节点隐藏等导致，浏览器就需要回流（及重新排版布局）。

而浏览器的重绘和回流都是比较昂贵的操作，如果每一次改变都直接对DOM进行操作，这会带来性能问题，而批量操作只会触发一次DOM更新。

> 思考题：Diff操作和DOM批量更新难道不应该是浏览器的职责吗？第三方框架中去做合不合适?

> 此处需要有一张插图

React Native

上文已经提到React Native 是React 在原生移动应用平台的衍生产物，那两者主要的区别是什么呢？其实，主要的区别在于虚拟DOM映射的对象是什么？React中虚拟DOM最终会映射为浏览器DOM树，而RN中虚拟DOM会通过JavaScript Core 映射为原生控件树。

JavaScript Core 是一个JavaScript解释器，它在React Native中主要有两个作用：

1. 为JavaScript提供运行环境。
2. 是JavaScript与原生应用之间通信的桥梁，作用和JsBridge一样，事实上，在iOS中，很多JsBridge的实现都是基于JavaScript Core 。

而RN中将虚拟DOM映射为原生控件的过程中分两步：

1. 布局消息传递； 将虚拟DOM布局信息传递给原生；
2. 原生根据布局信息通过对应的原生控件渲染控件树；

至此，React Native 便实现了跨平台。 相对于混合应用，由于React Native是原生控件渲染，所以性能会比混合应用中H5好很多，同时React Native是Web开发技术栈，也只需维护一份代码，同样是跨平台框架。

Weex

Weex是阿里巴巴于2016年发布的跨平台移动端开发框架，思想及原理和React Native类似，最大的不同是语法层面，Weex支持Vue语法和Rax语法，Rax 的 DSL 语法是基于 React JSX 语法而创造。与 React 不同，在 Rax 中 JSX 是必选的，它不支持通过其它方式创建组件，所以学习 JSX 是使用 Rax 的必要基础。而React Native只支持JSX语法。

快应用

快应用是华为、小米、OPPO、魅族等国内9大主流手机厂商将共同制定的轻量级应用标准，目标直指微信小程序。它也是采用JavaScript语言开发，原生控件渲染，与React Native和Weex相比主要有两点不同：

1. 快应用自身不支持Vue或React语法，其采用原生JavaScript开发，其开发框架和微信小程序很像，值得一提的是小程序目前已经可以使用Vue语法开发（mpvue），从原理上来讲，Vue的语法也可以移植到快应用上。
2. React Native和Weex的渲染/排版引擎是集成到框架中的，每一个APP都需要打包一份，安装包体积较大；而快应用渲染/排版引擎是集成到ROM中的，应用中无需打包，安装包体积小，正因如此，快应用才能在保证性能的同时做到快速分发。

总结

JavaScript开发+原生渲染的方式主要优点如下：

1. 采用Web开发技术栈，社区庞大、上手快、开发成本相对较低。
2. 原生渲染，性能相比H5提高很多。
3. 动态化较好，支持热更新。

不足：

1. 渲染时需要JavaScript和原生之间通信，在有些场景如拖动可能会因为通信频繁导致卡顿。
2. JavaScript为脚本语言，执行时需要JIT，执行效率和AOT代码仍有差距。
3. 由于渲染依赖原生控件，不同平台的控件需要单独维护，并且当系统更新时，社区控件可能会滞后；除此之外，其控件系统也要受到原生UI系统限制，例如，在Android中，手势冲突消歧规则是固定的，这在使用不同人写的控件嵌套时手势冲突非常棘手。

QT Moblie与Flutter

在本篇中，我们看看最后一种跨平台技术：自绘UI+原生。这种技术的思路是，通过在不同平台实现一个统一接口的渲染引擎来绘制UI，而不依赖系统原生控件，所以可以做到不同平台UI的一致性。注意，自绘引擎解决的是UI的跨平台问题，如果涉及其它系统能力调用，依然要涉及原生开发。这种平台技术的优点如下：

1. 性能高；由于自绘引擎是直接调用系统API来绘制UI，所以性能和原生控件接近。

2. 灵活、组件库易维护、UI外观保真度和一致性高；由于UI渲染不依赖原生控件，也就不需要根据不同平台的控件单独维护一套组件库，所以代码容易维护。由于组件库是同一套代码、同一个渲染引擎，所以在不同平台，组件显示外观可以做到高保真和高一致性；另外，由于不依赖原生控件，也就不会受原生布局系统的限制，这样布局系统会非常灵活。

不足：

1. 动态性不足；为了保证UI绘制性能，自绘UI系统一般都会采用AOT模式编译其发布包，所以应用发布后，不能像Hybird和RN那些使用JavaScript（JIT）作为开发语言的框架那样动态下发代码。

也许你已经猜到Flutter就属于这一类跨平台技术，没错，Flutter正是实现一套自绘引擎，并拥有一套自己的UI布局系统。不过，自绘制引擎的思路并不是什么新概念，Flutter并不是第一个尝试这么做的，在它之前有一个典型的代表，即大名鼎鼎的QT。

QT简介

Qt是一个1991年由Qt Company开发的跨平台C图形用户界面应用程序开发框架。2008年，Qt Company科技被诺基亚公司收购，Qt也因此成为诺基亚旗下的编程语言工具。2012年，Qt被Digia收购。2014年4月，跨平台集成开发环境Qt Creator 3.1.0正式发布，实现了对于iOS的完全支持，新增WinRT、Beautifier等插件，废弃了无Python接口的GDB调试支持，集成了基于Clang的C/C代码模块，并对Android支持做出了调整，至此实现了全面支持iOS、Android、WP，它提供给应用程序开发者构建图形用户界面所需的所有功能。但是，QT虽然在PC端获得了巨大成功，备受社区追捧，然而其在移动端却表现不佳，在近几年，虽然偶尔能听到QT的声音，但一直很弱，无论QT本身技术如何、设计思想如何，但事实上终究是败了，究其原因，笔者认为主要有四：

第一：QT移动开发社区太小，学习资料不足，生态不好。

第二：官方推广不利，支持不够。

第三：移动端发力较晚，市场已被其它动态化框架占领（Hybird和RN）。

第四：在移动开发中，C++开发和Web开发栈相比有着先天的劣势。

基于此四点，尽管QT是移动端开发跨平台自绘引擎的先驱，但却成为了烈士。

Flutter简介

“千呼万唤始出来”，铺垫这么久，现在终于等到本书的主角出场了！

Flutter是Google发布的一个用于创建跨平台、高性能移动应用的框架。Flutter和QT mobile一样，都没有使用原生控件，相反都实现了一个自绘引擎，使用自身的布局、绘制系统。那么，我们会担心，QT mobile面对的问题Flutter是否也一样，Flutter会不会步入QT mobile后尘，成为另一个烈士？要回到这个问题，我们先来看看Flutter诞生过程：

- 2017 年 Google I/O 大会上，Google 首次推出了一款新的用于创建跨平台、高性能的移动应用框架——Flutter。
- 2018年2月，Flutter发布了第一个Beta版本，同年五月， 在2018年Google I/O大会上，Flutter 更新到了 beta 3 版本。
- 2018年6月，Flutter发布了首个预览版本，这意味着 Flutter 进入了正式版（1.0）发布前的最后阶段。

观其发展，在2018年5月份，Flutter 进入了 GitHub stars 排行榜前 100 名，已有 27k star。而今天(2018年8月16日)，已经有35K的Star。经历了短短一年多的时间，Flutter 生态系统得以快速增长，由此可见，Flutter在开发者中受到了热烈的欢迎，其未来发展值得期待！

现在，我们来和QT mobile做一个对比：

1. 生态；从Github上来看，目前Flutter活跃用户正在高速增长。从Stackoverflow上提问来看，Flutter社区现在已经很庞大。Flutter的文档、资源也越来越丰富，开发过程中遇到的很多问题都可以在Stackoverflow或其github issue中找到答案。
2. 技术支持；现在Google正在大力推广Flutter，Flutter的作者中很多人都是来自Chromium团队，并且github上活跃度很高。另一个角度，从今年上半年Flutter频繁的版本发布也可以看出Google对Flutter的投入的资源不小，所以在官方技术支持这方面，大可不必担心。
3. 开发效率；Flutter的热重载可帮助开发者快速地进行测试、构建UI、添加功能并更快地修复错误。在iOS和Android模拟器或真机上可以实现毫秒级热重载，并且不会丢失状态。这真的很棒，相信我，如果你是一名原生开发者，体验了Flutter开发流后，很可能就不想重新回去做原生了，毕竟很少有人不吐槽原生开发的编译速度。

基于以上三点，相信读者和笔者一样，flutter未来如何，心中自有定论。到现在为止，我们已经对移动端开发技术有了一个全面的了解，接下来我们便要进入本书的主题，你准备好了吗！

本章总结

本章主要介绍了目前移动开发中三种跨平台技术，现在我们从框架角度对比一下：

技术类型	UI渲染方式	性能	开发效率	动态化	框架代表
H5+原生	WebView渲染	一般	高	✔	Cordova、Ionic
JavaScript+原生渲染	原生控件渲染	好	一般	✔	RN、Weex
自绘UI+原生	调用系统API渲染	好	高	默认不支持	QT、Flutter

上表中**开发语言**主要指UI的开发语言，**动态化**主要指是否支持动态下发代码和是否支持热更新。值得注意的是Flutter的Release包默认是使用Dart AOT模式编译的，所以不支持动态化，但Dart还有JIT或snapshot运行方式，这些模式都是支持动态化的，后续会介绍。

初识Flutter

Flutter简介

Flutter 是 Google推出并开源的移动应用开发框架，主打跨平台、高保真、高性能。开发者可以通过 Dart语言开发 App，一套代码同时运行在 iOS 和 Android平台。 Flutter提供了丰富的组件、接口，开发者可以很快地为 Flutter添加 native扩展。同时 Flutter还使用 Native引擎渲染视图，这无疑能为用户提供良好的体验。

跨平台自绘引擎

Flutter与用于构建移动应用程序的其它大多数框架不同，因为Flutter既不使用WebView，也不使用操作系统的原生控件。 相反，Flutter使用自己的高性能渲染引擎来绘制widget。这样不仅可以保证在Android和iOS上UI的一致性，而且也可以避免对原生控件依赖而带来的限制及高昂的维护成本。

Flutter使用Skia作为其2D渲染引擎，Skia是Google的一个2D图形处理函数库，包含字型、坐标转换，以及点阵图都有高效能且简洁的表现，Skia是跨平台的，并提供了非常友好的API，目前Google Chrome浏览器和Android均采用Skia作为其绘图引擎，值得一提的是，由于Android系统已经内置了Skia，所以Flutter在打包APK(Android应用安装包)时，不需要再将Skia打入APK中，但iOS系统并未内置Skia，所以构建iPA时，也必须将Skia一起打包，这也是为什么Flutter APP的Android安装包比iOS安装包小的主要原因。

高性能

Flutter高性能主要靠两点来保证，首先，Flutter APP采用Dart语言开发。Dart在JIT（即时编译）模式下，速度与 JavaScript基本持平。但是 Dart支持 AOT，当以AOT模式运行时，JavaScript便远远追不上了。速度的提升对高帧率下的视图数据计算很有帮助。其次，Flutter使用自己的渲染引擎来绘制UI，布局数据等由Dart语言直接控制，所以在布局过程中不需要像RN那样要在JavaScript和Native之间通信，这在一些滑动和拖动的场景下具有明显优势，因为在滑动和拖动过程往往都会引起布局发生变化，所以JavaScript需要和Native之间不停的同步布局信息，这和在浏览器中要JavaScript频繁操作DOM所带来的问题是相同的，都会带来比较可观的性能开销。

采用Dart语言开发

这是一个很有意思，但也很有争议的问题，在了解Flutter为什么选择了 Dart而不是JavaScript之前我们先来介绍两个概念：JIT和AOT。

目前，程序主要有两种运行方式：静态编译与动态解释。静态编译的程序在执行前全部被翻译为机器码，通常将这种类型称为**AOT** （Ahead of time）即“提前编译”；而解释执行的则是一句一句边翻译边运行，通常将这种类型称为**JIT**（Just-in-time）即“即时编译”。AOT程序的典型代表是用C/C++开发的应用，它们必须在执行前编译成机器码，而JIT的代表则非常多，如JavaScript、python等，事实上，所有脚本语言都支持JIT模式。但需要注意的是JIT和AOT指的是程序运行方式，和编程语言并非强关联的，有些语言既可以以JIT方式运行也可以以AOT方式运行，如Java、Python，它们可以在第一次执行时编译成中间字节码、然后在之后执行时可以直接执行字节码，也许有人会说，中间字节码并非机器码，在程序执行时仍然需要动态将字节码转为机器码，是的，这没有错，不过通常我们区分是否为AOT的标准就是看代码在执行之前是否需要编译，只要需要编译，无论其编译产物是字节码还是机器码，都属于AOT。在此，读者不必纠结于概念，概念就是为了传达精神而发明的，只要读者能够理解其原理即可，得其神忘其形。

现在我们看看Flutter为什么选择Dart语言？笔者根据官方解释以及自己对Flutter的理解总结了以下几条（由于其它跨平台框架都将JavaScript作为其开发语言，所以主要将Dart和JavaScript做一个对比）：

1. **开发效率高**

   Dart运行时和编译器支持Flutter的两个关键特性的组合：

   基于JIT的快速开发周期：Flutter在开发阶段采用，采用JIT模式，这样就避免了每次改动都要进行编译，极大的节省了开发时间；

   基于AOT的发布包: Flutter在发布时可以通过AOT生成高效的ARM代码以保证应用性能。而JavaScript则不具有这个能力。

2. **高性能**

   Flutter旨在提供流畅、高保真的的UI体验。为了实现这一点，Flutter中需要能够在每个动画帧中运行大量的代码。这意味着需要一种既能提供高性能的语言，而不会出现会丢帧的周期性暂停，而Dart支持AOT，在这一点上可以做的比JavaScript更好。

3. **快速内存分配**

   Flutter框架使用函数式流，它很大程度上依赖于底层的内存分配器，因此，能

够有效地处理小的、短期的内存分配会非常重要，所以在缺乏此功能的语言中Flutter无法有效地工作；当然Chrome V8的JavaScript引擎在内存分配上也已经做的很好，事实上Dart开发团队很多成员都是来自Chrome团队的，所以在内存分配上Dart并不能作为超越JavaScript的优势，而对于Flutter来说，它需要这样的特性，而Dart也正好满足而已。

4. **类型安全**

   由于Dart是类型安全的语言，支持静态类型检测，可以在编译前发现一些类型错误，排除潜在问题。这一点对于前端开发者来说可能会更有说服力，由于JavaScript是一个弱类型语言，前端社区出现了很多给JavaScript代码添加静态类型检测的扩展语言和工具，如：Coffeescript、微软的TypeScript以及Facebook的Flow。相比之下，Dart本身支持静态类型，这是它的一个重要优势。

5. **Dart团队就在隔壁**

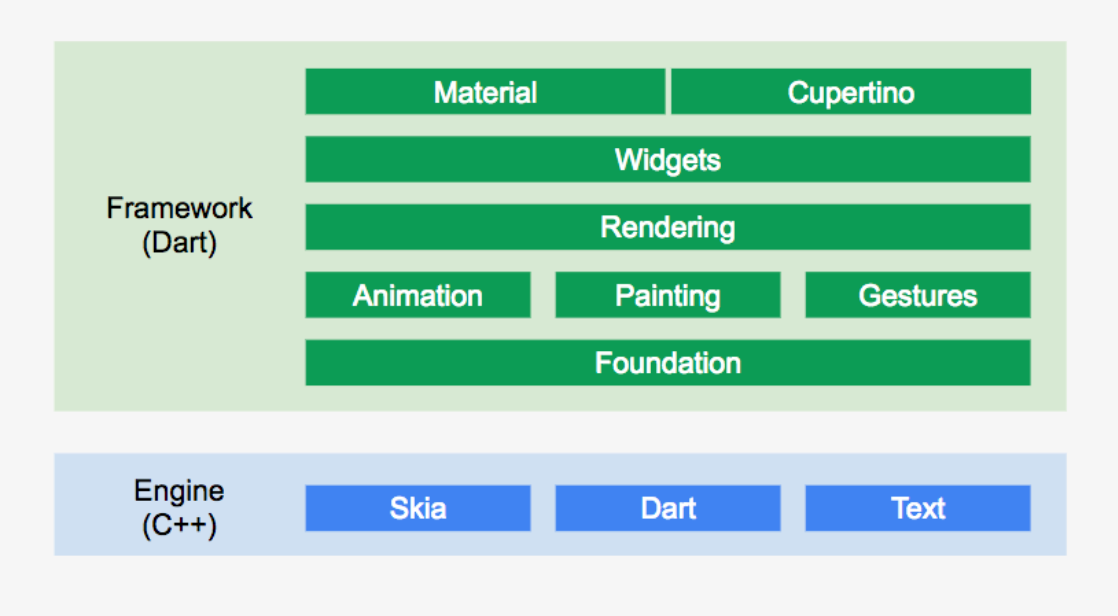
   我竟无言以对，看似不应该作为理由的理由，仔细想来却是非常重要的理由。如此一来，Flutter团队可以获得更多、更方便的支持，正如Flutter官网所述“我们有机会与Dart社区密切合作，Dart社区正在积极投入资源改进Dart在Flutter中的使用。例如，当我们采用Dart时，该语言没有提供生成原生二进制文件的工具链（这对于实现可预测的高性能是很有帮助的），但是现在它实现了，因为Dart团队为Flutter构建了它。同样，Dart VM之前已经针对吞吐量进行了优化，但团队现在正在优化VM的延迟时间，这对于Flutter的工作负载更为重要。”

总结

本节主要介绍了一下Flutter的特点，如果你感到有些点还不是很好理解，不用着急，随着日后对Flutter细节的了解，再回过头来看，相信你会有更深的体会。

Flutter框架结构

本节我们先对Flutter的框架做一个整体介绍，旨在让读者心中有一个整体的印象，这对初学者来说非常重要，如果一下子便深入到Flutter中，就像一个在沙漠中没有地图的人，即使可以找到一个绿洲，但是他也不会知道下一个绿洲在哪，所以，无论学什么技术，都要现有一张清晰的“地图”，而我们的学习过程就是“按图索骥”，这样我们才不会陷于细节而“目无全牛”。言归正传，我们看一下Flutter官方提供的一张框架图：



Flutter Framework

这是一个纯 Dart实现的 SDK，它实现了一套基础库，从低向上，我们来简单介绍一下：

- 底下两层（Foundation和Animation、Painting、Gestures）在Google的一些视频中被合并为一个dart UI层，对应的是Flutter中的`dart:ui`包，它是Flutter引擎暴露的底层UI库，提供动画、手势及绘制能力。

- Rendering层，这一层是一个抽象的布局层，它依赖于dart UI层，Rendering层会构建一个UI树，当UI树有变化时，会计算出变化部分，然后再更新UI树，最终将UI树绘制到屏幕上，这个类似于React中的虚拟DOM。Rendering层可以说是Flutter UI框架最核心的部分，它除了确定每个UI元素的位置、大小之外还要进行坐标变换、绘制(调用底层dart:ui)。

- Widgets层是Flutter提供的的一套基础组件库，在基础组件库之上，Flutter还提供了 Material 和Cupertino两种视觉风格的组件库。而**我们Flutter开发的大多数场景，只是和这两层打交道**。

Flutter Engine

这是一个纯 C++实现的 SDK，其中包括了 Skia引擎、Dart运行时、文字排版引擎等。在代码调用 `dart:ui`库时，调用最终会走到Engine层，然后实现真正的绘制逻辑。

总结

Flutter框架本身有着良好的分层设计，本节旨在让读者对Flutter整体框架有个大概的印象，相信到现在为止，读者已经对Flutter有一个初始印象，在我们正式动手之前，我们还需要了解一下Flutter的开发语言Dart。

如何学习Flutter

本节给大家一些学习建议，分享一下笔者在学习Flutter中的一些心得，希望可以帮助你提高学习效率，避免不必要的坑。

资源

- **官网**：阅读Flutter官网的资源是快速入门的最佳方式，同时官网也是了解最新Flutter发展动态的地方，由于目前Flutter仍然处于快速发展阶段，所以建议读者还是时不时的去官网看看有没有新的动态。

- **源码及注释**：源码注释应作为学习Flutter的第一文档，Flutter SDK的源码是开源的，并且注释非常详细，也有很多示例，实际上，Flutter官方的SDK文档就是通过注释生成的。源码结合注释可以帮你解决大多数问题。

- **Github**：如果遇到的问题在StackOverflow上也没有找到答案，可以去github flutter 项目下提issue。

- **Gallery源码**：Gallery是Flutter官方示例APP，里面有丰富的示例，读者可以在网上下载安装。Gallery的源码在Flutter源码“examples”目录下。

社区

- **StackOverflow**：如果你还没听过StackOverflow，这是目前全球最大的程序员问答社区，现在也是活跃度最高的Flutter问答社区。StackOverflow上面除了世界各地的Flutter使用者会在上面交流之外，Flutter开发团队的成员也经常会在上面回答问题。
- **Flutter中文网社区**：Flutter中文网(https://flutterchina.club)是笔者维护中文网站，目前也是最大的中文资源社区，上面提供了Flutter官网的文档翻译、开源项目、及案例，还有申请加入组织的入口哦。
- **掘金**：掘金的Flutter主页目前也收集了不少的博客文章，读者可以自行浏览。

总结

有了资料和社区后，对于我们学习者自身来说，最重要的还是要多动手、多实践，在本书后面的章节中，希望读者能够亲自动手写一下示例。准备好了吗，下一章中，我们将正式进入Flutter世界！

安装Flutter

工欲善其事必先利其器，本节首先会分别介绍一下在Windows和macOS下Flutter SDK的安装，然后再介绍一下配IDE和模拟器的使用。

搭建Flutter开发环境

由于Flutter会同时构建Android和IOS两个平台的发布包，所以Flutter同时依赖Android SDK和iOS SDK，在安装Flutter时也需要安装响应平台的构建工具和SDK。下面我们分别介绍一下Windows和macOS下的环境搭建。

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 https://flutter.io/community/china 以获得有关镜像服务器的最新动态。

在Windows上搭建Flutter开发环境

系统要求

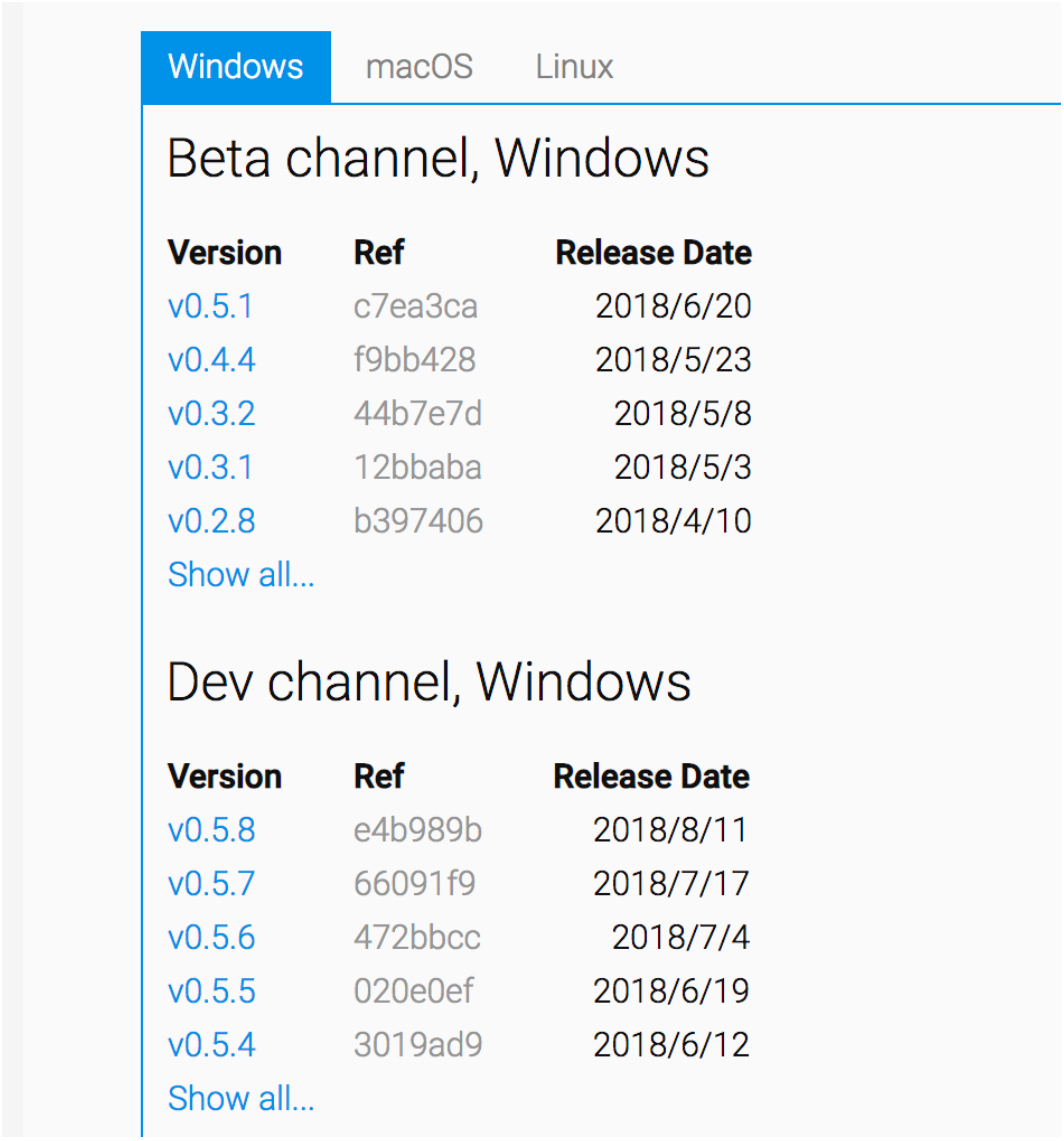
要安装并运行Flutter，您的开发环境必须满足以下最低要求:

- 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.
    - PowerShell 5.0 或更新的版本
    - Git for Windows（Git命令行工具）;

    如果已安装Git for Windows，请确保可以在命令提示符或PowerShell中运行 git 命令

获取Flutter SDK

1. 去flutter官网下载其最新可用的安装包，官网地址：https://flutter.io/sdk-archive/#windows



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，地址：https://github.com/flutter/flutter/releases 。

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。

更新环境变量

如果你想在Windows系统自带命令行（而不是）运行flutter命令，需要添加以下环境变量到用户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.
  - 如果该条目不存在，创建一个新用户变量 Path ，然后将 `flutter\bin` 的全路径作为它的值.

重启Windows以应用此更改.

运行 flutter doctor命令

在Flutter命令行运行如下命令来查看是否还需要安装其它依赖，如果需要，安装它们：

```
flutter doctor
```

该命令检查你的环境并在命令行窗口中显示报告。Dart SDK已经在打包在Flutter SDK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。

例如：

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

第一次运行一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。缺失的依赖需要安装一下，安装完成后再运行 `flutter doctor` 命令来验证是否安装成功。

Android设置

Flutter依赖于Android Studio的全量安装。Android Studio不仅可以管理Android 平台依赖、SDK版本等，而且它也是Flutter开发推荐的IDE之一（当然，你也可以使用其它编辑器或IDE，我们将会在后面讨论）。

安装Android Studio

1. 下载并安装 Android Studio，下载地址：https://developer.android.com/studio/index.html 。
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，这些是用Flutter进行Android开发所需要的。

安装遇到问题?

如果在安装过程中遇到问题，可以先去flutter官网查看一下安装方式是否发生变化，或者在网上搜索一下解决方案。

在macOS上搭建Flutter开发环境

在masOS下可以同时进行Android和iOS设备的测试。

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: macOS (64-bit)
- 磁盘空间: 700 MB (不包括Xcode或Android Studio的磁盘空间）.
- 工具: Flutter 依赖下面这些命令行工具.
  - `bash、mkdir、rm、git、curl、unzip、which`

获取Flutter SDK

1. 去flutter官网下载其最新可用的安装包，官网地址：https://flutter.io/sdk-archive/#macos

   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，地址：https://github.com/flutter/flutter/releases 。

2. 解压安装包到你想安装的目录，如：

   ```
   cd ~/development
   unzip ~/Downloads/flutter_macos_v0.5.1-beta.zip
   ```

3. 添加`flutter`相关工具到path中：

   ```
   export PATH=`pwd`/flutter/bin:$PATH
   ```

   此代码只能暂时针对当前命令行窗口设置PATH环境变量，要想永久将Flutter添加到PATH中请参考下面**更新环境变量** 部分。

运行 flutter doctor命令

这一步和Windows下步骤一致，不再赘述。

更新环境变量

将Flutter添加到PATH中，可以在任何终端会话中运行`flutter`命令。

对于所有终端会话永久修改此变量的步骤是和特定计算机系统相关的。通常，您会在打开新窗口时将设置环境变量的命令添加到执行的文件中。例如

1. 确定您Flutter SDK的目录记为“FLUTTER_INSTALL_PATH”，您将在步骤3中用到。
2. 打开(或创建) `$HOME/.bash_profile` 。文件路径和文件名可能在你的电脑上不同.
3. 添加以下路径:

   ```
   export PATH=[FLUTTER_INSTALL_PATH]/flutter/bin:$PATH
   ```

   例如笔者Flutter 安装目录是“~/code/flutter_dir”，那么代码为：

   ```
   export PATH=~/code/flutter_dir/flutter/bin:$PATH
   ```

4. 运行 `source $HOME/.bash_profile` 刷新当前终端窗口。

   > **注意:** 如果你使用终端是zsh，终端启动时 `~/.bash_profile` 将不会被加载，解决办法就是修改 `~/.zshrc` ，在其中添加：source ~/.bash_profile

5. 验证“flutter/bin”是否已在PATH中：

   ```
   echo $PATH
   ```

安装 Xcode

要为iOS开发Flutter应用程序，您需要Xcode 9.0或更高版本:

1. 安装Xcode 9.0或更新版本(通过链接下载或苹果应用商店).
2. 配置Xcode命令行工具以使用新安装的Xcode版本 `sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer` 对于大多数情况，当您想要使用最新版本的Xcode时，这是正确的路径。如果您需要使用不同的版本，请指定相应路径。
3. 确保Xcode许可协议是通过打开一次Xcode或通过命令 `sudo xcodebuild -license` 同意过了.

使用Xcode，您可以在iOS设备或模拟器上运行Flutter应用程序。

安装Android Studio

和Window一样，要在Android设备上构建并运行Flutter程序都需要先安装Android Studio，读者可以先自行下载并安装Android Studio，在此不再赘述。

升级 Flutter

Flutter SDK分支

Flutter SDK有多个分支，如beta、dev、master，其中beta分支为稳定分支（日后有新的稳定版本发布后可能也会有新的稳定分支，如1.0.0），dev和master为开发分支，安装flutter后，你可以运行`flutter channel`查看所有分支，如笔者本地运行后，结果如下：

```
Flutter channels:
  beta
  dev
* master
```

带"*"号的分支即你本地的Flutter SDK 跟踪的分支，要切换分支，可以使用`flutter channel beta`或`flutter channel master`，Flutter官方建议跟踪稳定分支，但你也可以跟踪`master`分支，这样可以查看最新的变化，但这样稳定性要低的多。

升级Flutter SDK和依赖包

要升级flutter sdk，只需一句命令：

```
flutter upgrade
```

该命令会同时更新Flutter SDK和你的flutter项目依赖包。如果你只想更新项目依赖包（不包括Flutter SDK），可以使用如下命令：

- `flutter packages get`获取项目所有的依赖包。
- `flutter packages upgrade`获取项目所有依赖包的最新版本。

IDE配置与使用

理论上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，Flutter官方建议使用Android Studio和VS Code之一以获得更好的开发体验。Flutter官方提供了这两款编辑器插件，通过IDE和插件可获得代码补全、语法高亮、widget编辑辅助、运行和调试支持等功能，可以帮助我们极大的提高开发效率。下面我们分别介绍一下Android Studio和VS Code的配置及使用（Android Studio和VS Code读者可以在其官网获得最新的安装，由于安装比较简单，故不再赘述）。

Android Studio 配置与使用

由于Android Studio是基于IntelliJ IDEA开发的，所以读者也可以使用IntelliJ IDEA。

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)。

安装步骤：

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

接下来，让我们用Android Studio创建一个Flutter项目，然后运行它，并体验“热重载”。

创建Flutter应用

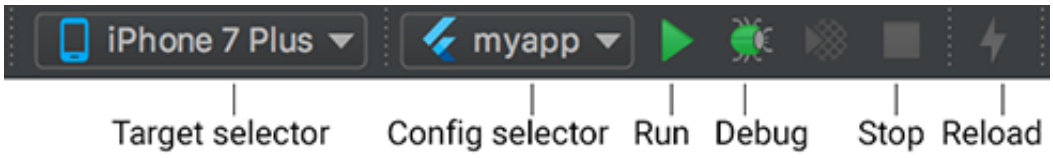
1. 选择 **File>New Flutter Project** 。
2. 选择 **Flutter application** 作为 project 类型, 然后点击 Next。
3. 输入项目名称 (如 `myapp`)，然后点击 Next。
4. 点击 **Finish**。
5. 等待Android Studio安装SDK并创建项目。

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

在项目目录中，您应用程序的代码位于 `lib/main.dart`。

运行应用程序

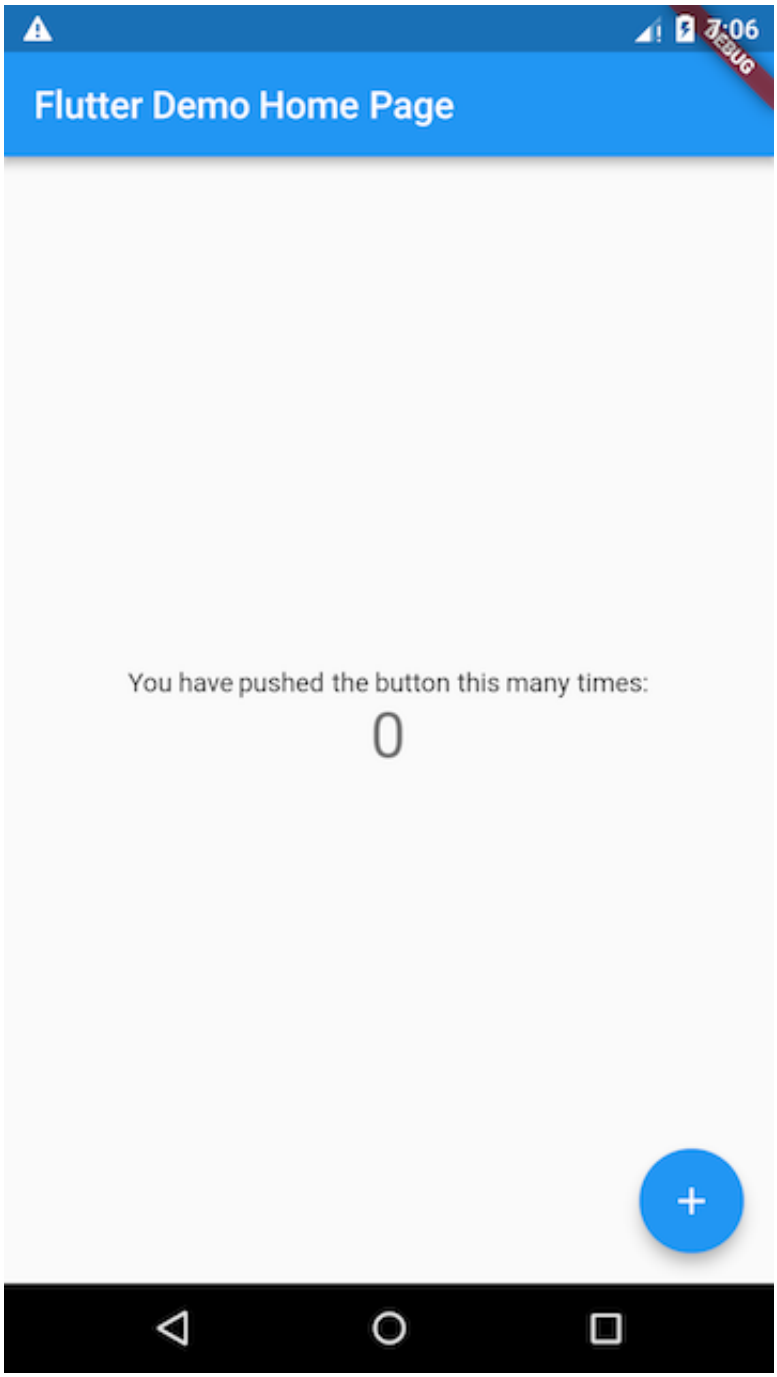
1. 定位到Android Studio工具栏：



2. 在 **target selector** 中, 选择一个运行该应用的Android设备。如果没有列出可用，请选择 **Tools>Android>AVD Manager** 并在那里创建一个。

3. 在工具栏中点击 **Run图标**。

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。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`lib/main.dart`文件
  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。

VS Code的配置与使用

VS Code是一个轻量级编辑器，支持Flutter运行和调试。

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。
6. 验证配置
    - 调用 **View>Command Palette...**
    - 输入 ‘doctor’, 然后选择 **‘Flutter: Run Flutter Doctor’** action。
    - 查看“OUTPUT”窗口中的输出是否有问题

创建Flutter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

体验热重载

1. 打开 `lib/main.dart` 文件。
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。

连接设备运行Flutter应用

Window下只支持为Android设备构建并运行Flutter应用，而macOS同时支持为iOS和Android设备。下面分别介绍如何连接Android和iOS设备来运行flutter应用。

连接Android模拟器

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

1. 启动 **Android Studio>Tools>Android>AVD Manager** 并选择 **Create Virtual Device**.

2. 选择一个设备并选择 **Next**。

3. 为要模拟的Android版本选择一个或多个系统印象，然后选择 **Next**. 建议使用 *x86* 或 *x86_64* image .

4. 在 “Emulated Performance”下, 选择 **Hardware - GLES 2.0** 以启用 硬件加速.

5. 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.

6. 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。

7. 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。

连接Android真机设备

要准备在Android设备上运行并测试Flutter应用，需要Android 4.1（API level 16）或更高版本的Android设备.

1. 在Android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。如果设备出现调试授权提示，请授权你的电脑可以访问该设备。
3. 在命令行运行 `flutter devices` 命令以验证Flutter识别您连接的Android设备。
4. 运行启动你应用程序 `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` 环境变量设置为响应的SDK安装目录。

连接iOS模拟器

要准备在iOS模拟器上运行并测试Flutter应用，请按以下步骤操作：

1. 在你的MAC上，通过 Spotlight 或以下命令找到模拟器：

   ```
   open -a Simulator
   ```

2. 通过检查模拟器 **Hardware > Device** 菜单中的设置，确保模拟器正在使用64位设备（iPhone 5s或更高版本）。

3. 根据你电脑屏幕大小，模拟高清屏iOS设备可能会溢出屏幕。可以在模拟器的 **Window> Scale** 菜单下设置设备比例。

4. 运行 `flutter run` 启动flutter应用程序。

连接iOS真机设备

要将Flutter应用安装到iOS真机设备，需要一些额外的工具和一个Apple帐户，还需要在Xcode中进行一些设置。

1. 安装 homebrew （如果已经安装了brew,跳过此步骤）。

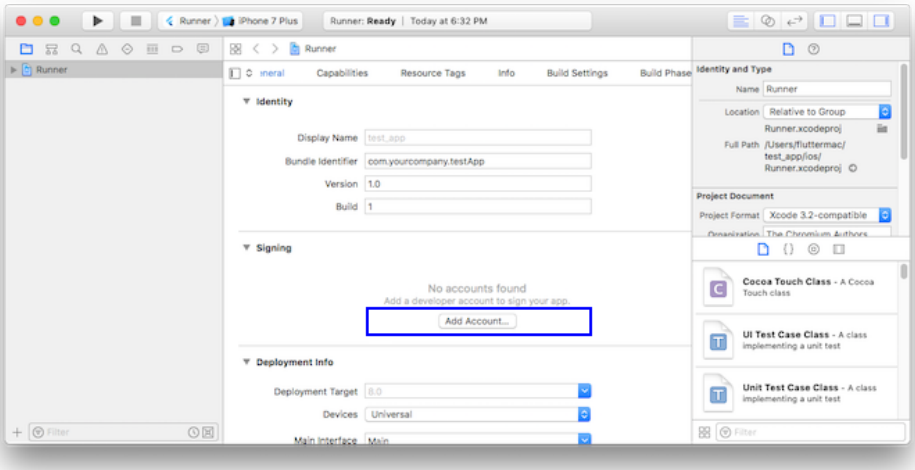
2. 打开终端并运行如下这些命令：

```
brew update
brew install --HEAD libimobiledevice
brew install ideviceinstaller ios-deploy cocoapods
pod setup
```

如果这些命令中的任何一个失败并出现错误，请运行brew doctor并按照说明解决问题.

3. 遵循Xcode签名流程来配置您的项目:

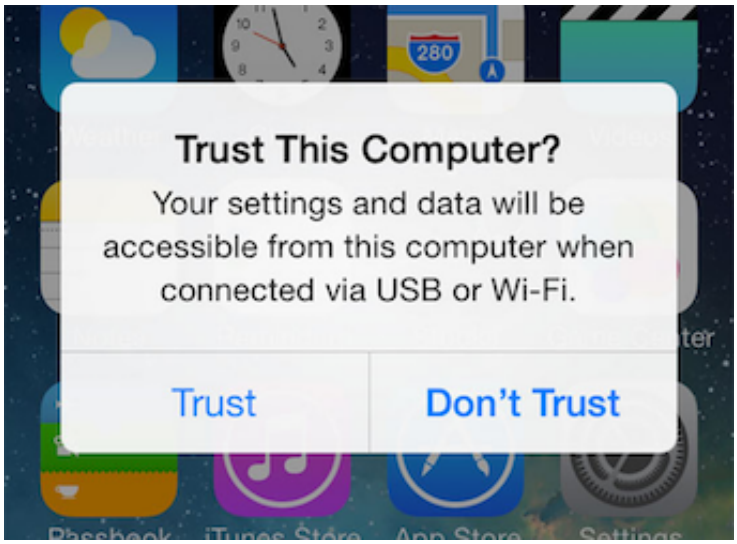
   - 在你Flutter项目目录中通过 `open ios/Runner.xcworkspace` 打开默认的Xcode workspace.
   - 在Xcode中，选择导航面板左侧中的`Runner`项目。
   - 在 `Runner` target设置页面中，确保在 **General > Signing > Team** 下选择了你的开发团队。当你选择一个团队时，Xcode会创建并下载开发证书，向你的设备注册你的帐户，并创建和下载配置文件（如果需要）。
   - 要开始您的第一个iOS开发项目，您可能需要使用您的Apple ID登录Xcode。



   任何Apple ID都支持开发和测试，但若想将应用分发到App Store，就必须注册Apple开发者计划，有关详情读者可以自行了解。

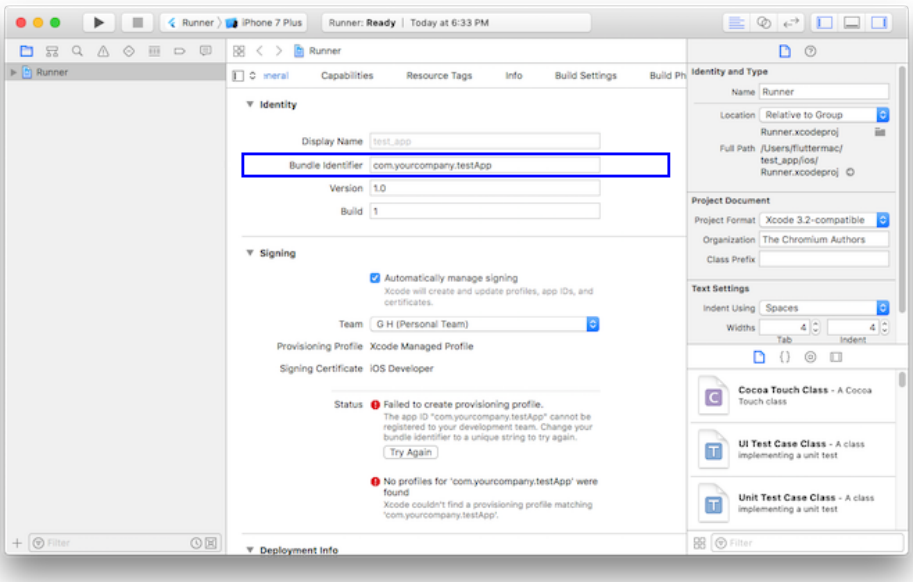
iv. 当您第一次attach真机设备进行iOS开发时，需要同时信任你的Mac和该设备上的开发证书。首次将iOS设备连接到Mac时，请在对话框中选

择 `Trust`。



然后，转到iOS设备上的**设置**菜单，选择 **常规>设备管理** 并信任您的证书。

v. 如果Xcode中的自动签名失败，请验证项目的 **General > Identity > Bundle Identifier** 值是否唯一.



vi. 运行 `flutter run` 启动flutter应用程序。

Dart语言简介

在之前我们已经介绍过Dart语言的相关特性，读者可以翻看一下，如果你熟悉Dart语法，可以跳过本节，如果你还不了解Dart，不用担心，按照笔者经验，如果你有过其他编程语言经验，尤其是Java和JavaScript的话，所以，如果你是前端或Android开发者，那么将会非常容易上手Dart。当然，如果你是iOS开发者，也不用担心，dart中也有一些与swift比较相似的特性，如命名参数等，笔者当时学习Dart时，只是花了一个小时，看完Dart官网的Language Tour，就开始动手写Fluttter了。

在笔者看来，Dart的设计目标应该是既对标Java，也对标JavaScript，Dart在静态语法方面和Java非常相似，如类型定义、函数声明、泛型等，而在动态特性方面又和JavaScript很像，如函数式特性、异步支持等。除了融合Java和JavaScript语言之所长之外，Dart也具有一些其它具有表现力的语法，如可选命名参数、`..`（级联运算符）和`?.`（条件成员访问运算符）以及`??`（判空赋值运算符）。其实，对编程语言了解比较多的读者会发现，在Dart中其实看到的不仅有Java和JavaScript的影子，它还具有其它编程语言中的身影，如命名参数在Objective-C和Swift中早就很普遍，而`??`操作符在Php 7.0语法中就已经存在了，因此我们可以看到Google对Dart语言给予厚望，是想把Dart打造成一门集百家之所长的编程语言。

接下来，我们先对Dart语法做一个简单的介绍，然后再将Dart与JavaScript和Java做一个简要的对比，方便读者更好的理解。

> 注意：由于本书并非专门介绍Dart语言的书籍，所以本章主要会介绍一下在Flutter开发中常用的语法特性，如果想更多了解Dart，读者可以去Dart官网学习，现在互联网上Dart相关资料已经很多了。另外Dart 2.0已经正式发布，所以本书所有示例均采用Dart 2.0语法。

变量声明

1. **var**

   类似于JavaScript中的`var`，它可以接收任何类型的变量，但最大的不同是Dart中var变量一旦赋值，类型便会确定，则不能再改变其类型，如：

```
var t;
t="hi world";
// 下面代码在dart中会报错，因为变量t的类型已经确定为String，
// 类型一旦确定后则不能再更改其类型。
t=1000;
```

上面的代码在JavaScript是没有问题的，前端开发者需要注意一下，之所以有此差异是因为Dart本身是一个强类型语言，任何变量都是有确定类型的，在Dart中，当用`var`声明一个变量后，Dart在编译时会根据第一次赋值数据的类型来推断其类型，编译结束后其类型就已经被确定，而JavaScript是纯粹的弱类型脚本语言，var只是变量的声明方式而已。

2. **dynamic和Object**

`Dynamic`和`Object`与`var`功能相似，都会在赋值时自动进行类型推断，不同在于，赋值后可以改变其类型，如：

```
dynamic t;
t="hi world";
//下面代码没有问题
t=1000;
```

`Object`是dart所有对象的根基类，也就是说所有类型都是`Object`的子类，所以任何类型的数据都可以赋值给`Object`声明的对象，所以表现效果和`dynamic`相似。

3. **final和const**

如果您从未打算更改一个变量，那么使用`final`或`const`，不是`var`，也不是一个类型。 一个`final`变量只能被设置一次，两者区别在于：`const`变量是一个编译时常量，`final`变量在第一次使用时被初始化。被`final`或者`const`修饰的变量，变量类型可以省略，如：

```
//可以省略String这个类型声明
final str = "hi world";
//final str = "hi world";
const str1 = "hi world";
//const String str1 = "hi world";
```

函数

Dart是一种真正的面向对象的语言，所以即使是函数也是对象，并且有一个类型**Function**。这意味着函数可以赋值给变量或作为参数传递给其他函数，这是函数式编程的典型特征。

1. 函数声明

```
bool isNoble(int atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}
```

dart函数声明如果没有显示申明返回值类型时会默认当做`dynamic`处理，注意，函数返回值没有类型推断：

```
typedef bool CALLBACK();

//不指定返回类型，此时默认为dynamic，不是bool
isNoble(int atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}

void test(CALLBACK cb){
   print(cb());
}
//报错，isNoble不是bool类型
test(isNoble);
```

2. 对于只包含一个表达式的函数，可以使用简写语法

```
bool isNoble  (int atomicNumber ) => _nobleGases [ atomicNumber
] ! = null ;
```

3. 函数作为变量

```
var say= (str){
  print(str);
};
say("hi world");
```

4. 函数作为参数传递

```
void execute(var callback){
    callback();
}
execute(()=>print("xxx"))
```

5. 可选的位置参数

   包装一组函数参数，用[]标记为可选的位置参数：

```
String say(String from, String msg, [String device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

   下面是一个不带可选参数调用这个函数的例子：

```
say('Bob', 'Howdy'); //结果是：  Bob says Howdy
```

   下面是用第三个参数调用这个函数的例子：

```
say('Bob', 'Howdy', 'smoke signal'); //结果是：Bob says Howdy
with a smoke signal
```

6. 可选的命名参数

   定义函数时，使用{param1, param2, ...}，用于指定命名参数。例如：

```
//设置[bold]和[hidden]标志
void enableFlags({bool bold, bool hidden}) {
    // ...
}
```

调用函数时，可以使用指定命名参数。例如：`paramName: value`

```
enableFlags(bold: true, hidden: false);
```

可选命名参数在Flutter中使用非常多。

异步支持

Dart类库有非常多的返回`Future`或者`Stream`对象的函数。 这些函数被称为**异步函数**：它们只会在设置好一些耗时操作之后返回，比如像 IO操作。而不是等到这个操作完成。

`async`和`await`关键词支持了异步编程，运行您写出和同步代码很像的异步代码。

Future

`Future`与JavaScript中的`Promise`非常相似，表示一个异步操作的最终完成（或失败）及其结果值的表示。简单来说，它就是用于处理异步操作的，异步处理成功了就执行成功的操作，异步处理失败了就捕获错误或者停止后续操作。一个Future只会对应一个结果，要么成功，要么失败。

由于本身功能较多，这里我们只介绍其常用的API及特性。还有，请记住，`Future`的所有API的返回值仍然是一个`Future`对象，所以可以很方便的进行链式调用。

Future.then

为了方便示例，在本例中我们使用`Future.delayed`创建了一个延时任务（实际场景会是一个真正的耗时任务，比如一次网络请求），即2秒后返回结果字符串"hi world!"，然后我们在`then`中接收异步结果并打印结果，代码如下：

```
Future.delayed(new Duration(seconds: 2),(){
   return "hi world!";
}).then((data){
   print(data);
});
```

Future.catchError

如果异步任务发生错误，我们可以在`catchError`中捕获错误，我们将上面示例改为：

```
Future.delayed(new Duration(seconds: 2),(){
   //return "hi world!";
   throw AssertionError("Error");
}).then((data){
   //执行成功会走到这里
   print("success");
}).catchError((e){
   //执行失败会走到这里
   print(e);
});
```

在本示例中，我们在异步任务中抛出了一个异常，`then`的回调函数将不会被执行，取而代之的是`catchError`回调函数将被调用；但是，并不是只有`catchError`回调才能捕获错误，`then`方法还有一个可选参数`onError`，我们也可以它来捕获异常：

```
Future.delayed(new Duration(seconds: 2), () {
    //return "hi world!";
    throw AssertionError("Error");
}).then((data) {
    print("success");
}, onError: (e) {
    print(e);
});
```

Future.whenComplete

有些时候，我们会遇到无论异步任务执行成功或失败都需要做一些事的场景，比如在网络请求前弹出加载对话框，在请求结束后关闭对话框。这种场景，有两种方法，第一种是分别在`then`或`catch`中关闭一下对话框，第二种就是使用`Future`的`whenComplete`回调，我们将上面示例改一下：

```
Future.delayed(new Duration(seconds: 2),(){
   //return "hi world!";
   throw AssertionError("Error");
}).then((data){
   //执行成功会走到这里
   print(data);
}).catchError((e){
   //执行失败会走到这里
   print(e);
}).whenComplete((){
   //无论成功或失败都会走到这里
});
```

Future.wait

有些时候，我们需要等待多个异步任务都执行结束后才进行一些操作，比如我们有一个界面，需要先分别从两个网络接口获取数据，获取成功后，我们需要将两个接口数据进行特定的处理后再显示到UI界面上，应该怎么做？答案是`Future.wait`，它接受一个`Future`数组参数，只有数组中所有`Future`都执行成功后，才会触发`then`的成功回调，只要有一个`Future`执行失败，就会触发错误回调。下面，我们通过模拟`Future.delayed`来模拟两个数据获取的异步任务，等两个异步任务都执行成功时，将两个异步任务的结果拼接打印出来，代码如下：

```
Future.wait([
  // 2秒后返回结果
  Future.delayed(new Duration(seconds: 2), () {
    return "hello";
  }),
  // 4秒后返回结果
  Future.delayed(new Duration(seconds: 4), () {
    return " world";
  })
]).then((results){
  print(results[0]+results[1]);
}).catchError((e){
  print(e);
});
```

执行上面代码，4秒后你会在控制台中看到“hello world”。

Async/await

Dart中的`async/await`和JavaScript中的`async/await`功能和用法是一模一样的，如果你已经了解JavaScript中的`async/await`的用法，可以直接跳过本节。

回调地狱(Callback hell)

如果代码中有大量异步逻辑，并且出现大量异步任务依赖其它异步任务的结果时，必然会出现`Future.then`回调中套回调情况。举个例子，比如现在有个需求场景是用户先登录，登录成功后会获得用户Id，然后通过用户Id，再去请求用户个人信息，获取到用户个人信息后，为了使用方便，我们需要将其缓存在本地文件系统，代码如下：

```
//先分别定义各个异步任务
Future<String> login(String userName, String pwd){
    ...
    //用户登录
};
Future<String> getUserInfo(String id){
    ...
    //获取用户信息
};
Future saveUserInfo(String userInfo){
    ...
    //  保存用户信息
};
```

接下来，执行整个任务流：

```
login("alice","******").then((id){
 //登录成功后通过，id获取用户信息
 getUserInfo(id).then((userInfo){
    //获取用户信息后保存
    saveUserInfo(userInfo).then((){
       //保存用户信息，接下来执行其它操作
         ...
    });
  });
})
```

可以感受一下，如果业务逻辑中有大量异步依赖的情况，将会出现上面这种在回调里面套回调的情况，过多的嵌套会导致的代码可读性下降以及出错率提高，并且非常难维护，这个问题被形象的称为**回调地狱（Callback hell）**。回调地狱问题在之前JavaScript中非常突出，也是JavaScript被吐槽最多的点，但随着ECMAScript6和ECMAScript7标准发布后，这个问题得到了非常好的解决，而解决回调地狱的两大神器正是ECMAScript6引入了`Promise`，以及ECMAScript7中引入的`async/await`。而在Dart中几乎是完全平移了JavaScript中的这两者：`Future`相当于`Promise`，而`async/await`连名字都没改。接下来我们看看通过`Future`和`async/await`如何消除上面示例中的嵌套问题。

使用Future消除callback hell

```
login("alice","******").then((id){
    return getUserInfo(id);
}).then((userInfo){
    return saveUserInfo(userInfo);
}).then((e){
   //执行接下来的操作
}).catchError((e){
  //错误处理
  print(e);
});
```

正如上文所述，*“`Future` 的所有API的返回值仍然是一个`Future`对象，所以可以很方便的进行链式调用”*，如果在then中返回的是一个`Future`的话，该`future`会执行，执行结束后会触发后面的`then`回调，这样依次向下，就避免了层层嵌套。

使用async/await消除callback hell

通过`Future`回调中再返回`Future`的方式虽然能避免层层嵌套，但是还是有一层回调，有没有一种方式能够让我们可以像写同步代码那样来执行异步任务而不使用回调的方式？答案是肯定的，这就要使用`async/await`了，下面我们先直接看代码，然后再解释，代码如下：

```
task() async {
   try{
    String id = await login("alice","******");
    String userInfo = await getUserInfo(id);
    await saveUserInfo(userInfo);
    //执行接下来的操作
   } catch(e){
    //错误处理
    print(e);
   }
}
```

- `async`用来表示函数是异步的，定义的函数会返回一个`Future`对象，可以使用then方法添加回调函数。
- `await` 后面是一个`Future`，表示等待该异步任务完成，异步完成后才会往下走；`await`必须出现在`async`函数内部。

可以看到，我们通过`async/await`将一个异步流用同步的代码表示出来了。

> 其实，无论是在JavaScript还是Dart中，`async/await`都只是一个语法糖，编译器或解释器最终都会将其转化为一个Promise（Future）的调用链。

Stream

`Stream`也是用于接收异步事件数据，和`Future`不同的是，它可以接收多个异步操作的结果（成功或失败）。 也就是说，在执行异步任务时，可以通过多次触发成功或失败事件来传递结果数据或错误异常。 `Stream`常用于会多次读取数据的异步任务场景，如网络内容下载、文件读写等。举个例子：

```
Stream.fromFutures([
  // 1秒后返回结果
  Future.delayed(new Duration(seconds: 1), () {
    return "hello 1";
  }),
  // 抛出一个异常
  Future.delayed(new Duration(seconds: 2),(){
    throw AssertionError("Error");
  }),
  // 3秒后返回结果
  Future.delayed(new Duration(seconds: 3), () {
    return "hello 3";
  })
]).listen((data){
   print(data);
}, onError: (e){
   print(e.message);
},onDone: (){

});
```

上面的代码依次会输出：

```
I/flutter (17666): hello 1
I/flutter (17666): Error
I/flutter (17666): hello 3
```

代码很简单，就不赘述了。

> 思考题：既然Stream可以接收多次事件，那能不能用Stream来实现一个订阅者模式的事件总线?

总结

通过上面介绍，相信你对Dart应该有了一个初步的印象，由于笔者平时也使用Java和JavaScript，下面笔者根据自己的经验，结合Java和JavaScript，谈一下自己的看法。

> 之所以将Dart与Java和JavaScript对比，是因为，这两者分别是强类型语言和弱类型语言的典型代表，并且Dart 语法中很多地方也都借鉴了Java和JavaScript。

Dart vs Java

客观的来讲，Dart在语法层面确实比Java更有表现力；在VM层面，Dart VM在内存回收和吞吐量都进行了反复的优化，但具体的性能对比，笔者没有找到相关测试数据，但在笔者看来，只要Dart语言能流行，VM的性能就不用担心，毕竟Google在go（没用vm但有GC）、javascript（v8）、dalvik（android上的java vm）上已经有了很多技术积淀。值得注意的是Dart在Flutter中已经可以将GC做到10ms以内，所以Dart和Java相比，决胜因素并不会是在性能方面。而在语法层面，Dart要比java更有表现力，最重要的是Dart对函数式编程支持要远强于Java(目前只停留在lamda表达式)，而Dart目前真正的不足是**生态**，但笔者相信，随着Flutter的逐渐火热，会回过头来反推Dart生态加速发展，对于Dart来说，现在需要的是时间。

Dart vs JavaScript

JavaScript的弱类型一直被抓短，所以TypeScript、Coffeescript甚至是Facebook的flow（虽然并不能算JavaScript的一个超集，但也通过标注和打包工具提供了静态类型检查）才有市场。就笔者使用过的脚本语言中（笔者曾使用过Python、PHP），JavaScript无疑是**动态化**支持最好的脚本语言，比如在JavaScript中，可以给任何对象在任何时候动态扩展属性，对于精通JavaScript的高手来说，这无疑是一把利剑。但是，任何事物都有两面性，JavaScript的强大的动态化特性也是把双刃剑，你可经常听到另一个声音，认为JavaScript的这种动态性糟糕透了，太过灵活反而导致代码很难预期，无法限制不被期望的修改。毕竟有些人总是对自己或别人写的代码不放心，他们希望能够让代码变得可控，并期望有一套静态类型检查系统来帮助自己减少错误。正因如此，在Flutter中，Dart几乎放弃了脚本语言动态化的特性，如不支持反射、也不支持动态创建函数等。并且Dart在2.0强制开启了类型检查（Strong Mode），原先的检查模式（checked mode）和可选类型（optional type）将淡出，所以在类型安全这个层面来说，Dart和TypeScript、Coffeescript是差不多的，所以单从这一点来看，Dart并不具备什么明显优势，但综合起来看，dart既能进行服务端脚本、APP开发、web开发，这就有优势了！

综上所述，笔者还是很看好Dart语言的将来，之所以表这个态，是因为在新技术发展初期，很多人可能还有所摇摆，有所犹豫，所以有必要给大家打一剂强心针，当然，这是一个见仁见智的问题，大家可以各抒己见。