**一14一 一14一 一12一 一11一** 一10.5一 一10.5一 一10一

**目 录**

[1. Flutter介绍 2](#_Toc32677)

[1.1. Flutter是什么 2](#_Toc10580)

[1.2. Flutter的优势 2](#_Toc14801)

[1.2.1. 高效率 2](#_Toc19371)

[1.2.2. 创建美观、高度定制的用户体验 2](#_Toc18522)

[1.3. Flutter的核心内容 2](#_Toc1801)

[1.3.1. 一切都是控件 2](#_Toc2680)

[1.3.2. 组合大于继承 2](#_Toc26789)

[1.3.3. 层次结构 3](#_Toc24548)

[1.3.4. 构建控件 3](#_Toc9032)

[1.3.5. 处理用户交互 4](#_Toc615)

[1.4. 跟React Native、Weex等有什么不同 4](#_Toc15125)

[1.4.1. 原生App 4](#_Toc22805)

[1.4.2. WebViews 4](#_Toc2152)

[1.4.3. React Native 5](#_Toc31269)

[1.4.4. Flutter 5](#_Toc4795)

[2. 环境配置 5](#_Toc30917)

[3. Drat资料一：为 Java 准备的 Dart 教程 6](#_Toc2999)

[3.1. 创建一个简单的Dart类 6](#_Toc18015)

[3.2. 使用可选参数（而不是使用重载） 8](#_Toc13280)

[3.3. 创建工厂模式 10](#_Toc27426)

[3.4. 接口的实现 11](#_Toc32119)

[3.5. 函数式编程 11](#_Toc14297)

[4. Dart资料二：Dart语言下的Flutter 12](#_Toc10114)

[4.1.  基本类型 12](#_Toc25416)

[4.2. 变量 12](#_Toc6694)

[4.3. 方法 13](#_Toc22442)

[4.4. Flutter 13](#_Toc22193)

[5. Flutter Widget 14](#_Toc22802)

[5.1. 无状态StatelessWidget 14](#_Toc8451)

[5.2. 有状态StatefulWidget 14](#_Toc13225)

[6. Flutter 布局 15](#_Toc30009)

[6.1. Container 15](#_Toc22831)

[6.2. Colum、Row 16](#_Toc13261)

[6.3. Expanded 16](#_Toc20259)

[6.4. 使用 16](#_Toc32641)

[7. Flutter 页面 19](#_Toc7032)

# Flutter介绍

## Flutter是什么

Flutter是一个移动应用程序的软件开发工具包（SDK），用一个代码库构建高性能、高保真的iOS和Android应用程序。目标是使开发人员能够为Android和iOS提供自然的高质量的应用，在滚动行为、排版、图标等方面实现零差异。Flutter程序使用Dart语言编写

## Flutter的优势

### 高效率

* 用一套代码库就能开发iOS和Android应用
* 使用新型的、表现力强的语言和声明式的方法，用更少的代码来做更多的事情
* 开发调试更容易方便
  + 可以在应用程序运行时更改代码并重新加载查看效果，也就是热重新加载
  + 修复崩溃时可以从应用程序停止的位置继续调试

### 创建美观、高度定制的用户体验

* Flutter框架内置了一组丰富的质感设计控件
* 实现定制、美观、品牌驱动的设计，而不受OEM控件集的限制

## Flutter的核心内容

Flutter包括功能反应框架、2D渲染引擎、现成的控件和开发工具，这些组件协同工作，帮助您设计、构建、测试和调试应用程序。

### 一切都是控件

控件是每个Flutter应用程序的基本构建块，与分离视图、控制器、布局和其他属性的框架不同，Flutter具有一致的统一对象模型：控件。

一个控件可以定义：

* 结构元素（比如按钮或菜单）
* 风格元素（比如字体或颜色方案）
* 布局的方面（比如填充）
* 一些业务逻辑
* 等等…

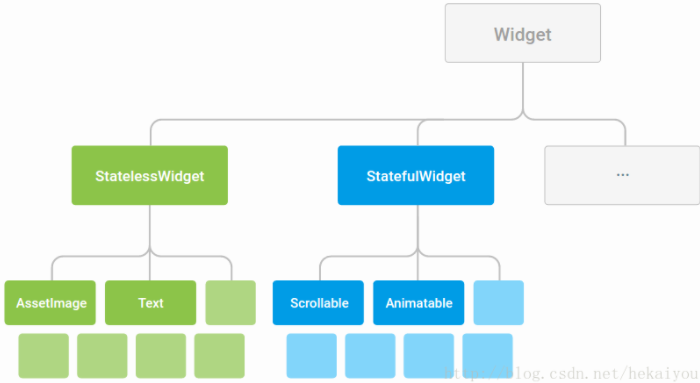
控件是基于构图形成层次结构，每个控件嵌套在其中，并从其父代继承属性，没有单独的“应用程序”对象，只有根控件。

您可以通过告知框架用另一个控件替换层次结构中的控件来响应事件，比如用户交互，然后框架会对比新的控件和旧的控件，并有效的更新用户界面，即更新有变化的控件。

### 组合大于继承

控件本身通常由许多小型、单用途的控件组成，结合起来产生强大的效果，例如，Container是一种常用的控件，由负责布局、绘画、定位和大小调整的几个控件组成，具体来说，Container是由LimitedBox、ConstrainedBox、 Align、Padding、DecoratedBox和Transform控件组成，而不是将Container子类化来产生自定义效果，您可以用这种新颖的方式组合这些以及其他简单的控件。

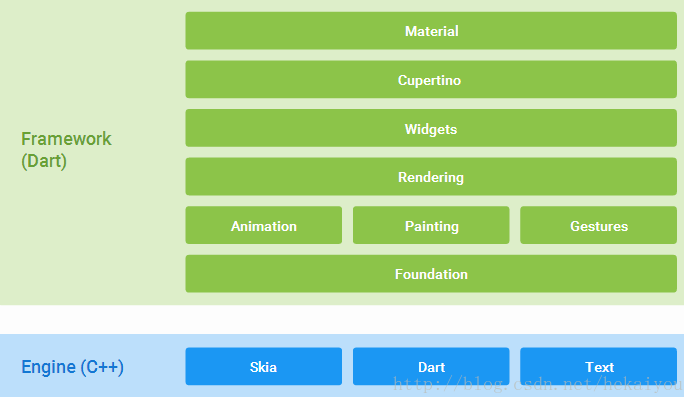
类的层次结构是扁平的，以最大化可能的组合数量



您还可以通过与其他控件组合来控制控件的布局，例如，要居中一个控件，您可以将其包装到Center控件中，有一些填充、对齐、行、列和网格的控件，这些布局控件没有自己的视觉表示，相反，它们的唯一目的是控制另一个控件布局的某些方面，当你想了解为什么一个控件以某种方式呈现，检查邻近的控件通常就可以知道原因

### 层次结构

Flutter框架被组织成一系列的层次，每一层都建立在底层之上



上图显示的框架上层，比下层使用的更频繁，这个设计的目标是帮助你用更少的代码来做更多的事情。例如，通过从Widgets层（控件层）组成基本控件来构建Material层（质感设计层），并且Widgets层（控件层）本身是通过从Rendering层（渲染层）编排下层对象来构建的。

这些层级提供了许多构建应用程序的选项，选择一种自定义的方法来发挥框架的完整表现力，或者从Widgets层（控件层）使用构建块，或混合搭配。您可以使用Flutter提供的现成控件，或使用Flutter团队用于构建框架的相同工具和技术来创建自己的自定义控件。

这就是Flutter，您可以获得高层次、统一的控件概念的生产力优势，同时还能深入到较低层次自由探索。

### 构建控件

您可以通过实现返回控件的树（或层次结构）的构建函数来定义控件的独特特征，这个树更具体地表示控件的用户界面的一部分。例如，工具栏控件可能具有返回一些文本和各种按钮的水平布局的构建函数，框架会递归地询问每个这些控件的构建，直到该过程在完全具体的控件中出现，然后框架再将其拼接成树。

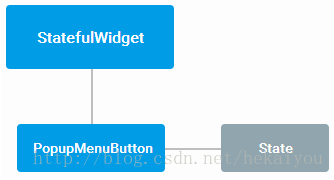
控件的构建功能应该没有副本的，无论何时被要求构建，控件应该返回一个新控件，而不管控件先前返回的是什么。但是框架会将先前版本与当前版本进行比较，并确定需要对用户界面进行哪些修改。

这种自动比较是非常有效的，可以实现高性能、互动的应用程序。构建函数的设计通过专注于声明控件是什么来简化代码，而不是将用户界面从一个状态更新到另一个状态的复杂化。

### 处理用户交互

如果控件的独特特征需要根据用户交互或其他因素进行更改，则该控件是有状态的。例如，如果控件具有每当用户点击按钮时递增的计数器，则计数器的值是该控件的状态，当该值更改时，需要重新构建控件以更新UI。

这些控件子类StatefulWidget（而不是StatelessWidget），并将其可变状态存储在State的子类中



每当你突然变化一个状态对象（例如增加计数器）时，你必须调用setState()来通知调用状态再次创建框架来更新用户界面。

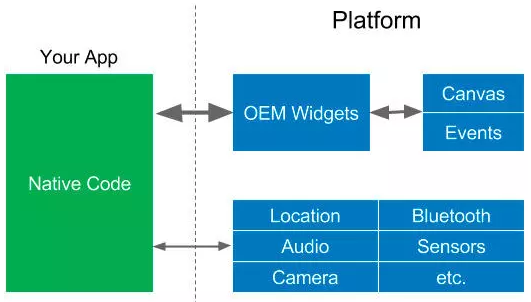
具有单独的状态和控件对象可以让其他控件以相同的方式处理无状态和有状态控件，而不用担心丢失状态。父类可以自由地创建一个新的子类实例，也不会失去子类的持续状态，更不需要坚持一个子类来保持自己的状态。该框架在适当时执行查找和重用现有状态对象的所有工作。

## 跟React Native、Weex等有什么不同

React Native、Weex等一直存在一个问题，就是性能跟原生App存在很大的差异。这跟它们的原理有很大的关系，下面从原生App，RN、Weex，Flutter的简单原理说一下它们的不同。

### 原生App

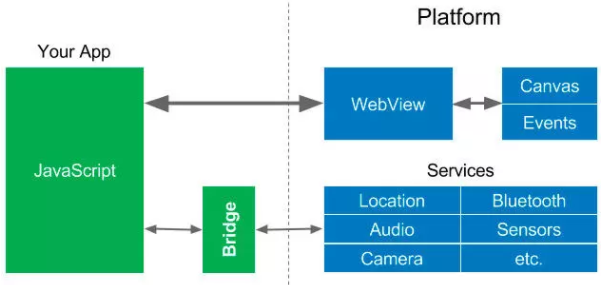
苹果2008年发布iOS，Google 2009年发布Android，它们的SDK是基于两种不同的编程语言Objective-C 和 Java,现在又有了Swift和Kotlin



上面是原生App的一个简单架构，开发人员直接调用平台SDK进行UI开发。由于语言及SDK的不同，所以开发人员必须为两个平台分别开发App。

### WebViews

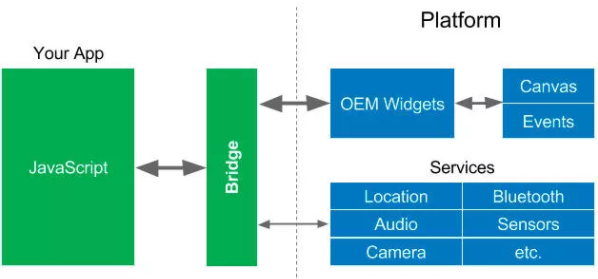
最早的跨平台方案是基于JaveScript 和 WebView的，像PhoneGap、Cordova、Ionic等。



UI通过WebView来显示html代码，系统服务则通过一个中间层桥接到JaveScript中去。

### React Native

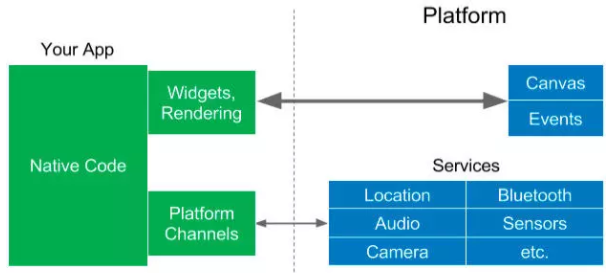
RN不仅桥接系统服务，也将系统UI也桥接到了JaveScript中，这样写出来的UI最终也会渲染成原生的控件



如上图这样，UI的渲染是很频繁的，要使UI不卡顿，必须达到60Fps。但是桥接会花一定的时间。所以这样的架构有时候会有性能问题。

### Flutter

Flutter使用Dart语言开发，Dart可以被编译（AOT）成不同平台的本地代码，让Flutter可以直接和平台通讯而不需要一个中间的桥接过程，从而提高了性能



# 环境配置

* 安装Git、Android Studio、Android SDK，配置好所有
* 在用户环境变量中配置PUB\_HOSTED\_URL = <https://pub.flutter-io.cn和FLUTTER_STORAGE_BASE_URL> = https://storage.flutter-io.cn
* cmd中切换到想要放flutter sdk的路径，比如H:\adt-bundle-windows
* 通过git clone -b beta <https://github.com/flutter/flutter.git克隆，将推荐的beta稳定版本克隆到本地，此时flutter> sdk的路径为H:\adt-bundle-windows\flutter
* 在用户环境变量中配置PATH = H:\adt-bundle-windows\flutter\bin,重启windows以应用此更改
* 在cmd中运行flutter doctor，首次运行flutter命令时，它会下载它自己的依赖项并自行编译
* 在Android Studio中配置plugins -> Browse Reponsitories -> 搜索Flutter的插件并安装，它会自动下载Dart插件

# Drat资料一：为 Java 准备的 Dart 教程

## 创建一个简单的Dart类

从创建一个简单的 Bicycle 类来开始学习 Dart，这个Dart类和Java所创建的类拥有同样的功能

java代码

Bicycle.java

public class Bicycle {  
 private int cadence;  
 private int gear;  
 private int speed;  
 public Bicycle(int startCadence, int startSpeed, int startGear) {  
 gear = startGear;  
 cadence = startCadence;  
 speed = startSpeed;  
 }  
 public int getCadence() {  
 return cadence;  
 }  
 public void setCadence(int newValue) {  
 cadence = newValue;  
 }  
 public int getGear() {  
 return gear;  
 }  
 public void setGear(int newValue) {  
 gear = newValue;  
 }  
 public int getSpeed() {  
 return speed;  
 }  
 public void applyBrake(int decrement) {  
 speed -= decrement;  
 }  
 public void speedUp(int increment) {  
 speed += increment;  
 }  
}

定义 Bicycle 类

在 main() 函数的上方，增加一个 Bicycle 类，它有三个实例变量，然后移除 main()函数中的内容，如下：

**class** Bicycle {  
 int cadence;  
 int speed;  
 int gear;  
}  
  
**void** main() {  
}

* main() 方法是 Dart 的主方法，如果你需要访问命令行传递过来的参数，可以使用 main(List<String> args 方法。
* main() 方法存在于最外层的作用域，在 Dart 中你可以在类之外编写代码，变量、方法、存取方法都可以独立于类之外维持生命周期。
* 无论是 main() 还是 Bicycle 类都声明为 public 的，默认都是 public 的，在 Dart 中没有诸如 public、private、protected 这样的关键词。

定义一个 Bicycle 的构造函数

在 Bicycle 类中增加以下的构造函数：

Bicycle(**this**.cadence, **this**.speed, **this**.gear);

* 这个构造函数没有方法体，这个在 Dart 中是合法的。
* 在没有方法体的构造函数后必须有分号“;” ，否则DartPad 将会显示一个错误，告诉你"A function body must be provided."
* 在构造函数的参数中使用 this 可以直接对实例变量进行赋值，不用再编写多余的代码。
* 上述代码同下列的代码是相同功效的。

Bicycle(int cadence, int speed, int gear) {

this.cadence = cadence;

this.speed = speed;

this.gear = gear;

}

实例化并打印 bicycle

向 main() 方法中增加下列代码：

void main() {

var bike = new Bicycle(2, 0, 1);

print(bike); //Instance of 'Bicycle'

}

移除可选的 new 关键字：

var bike = Bicycle(2, 0, 1);

* 在 Dart 2 中，new 关键字变成了可选的。
* 如果你确信某个变量的值不会再发生改变，你可以使用 final 来代替 var

复写toString方法

刚刚得到的输出 "Instance of 'Bicycle' 虽然是正确的，但是并没有显示出更具体的信息。所有的 Dart 类中都有一个 toString() 方法，你可以复写这个方法。在 Bicycle 类中的任意位置增加 toString()方法：

@override

String toString() => 'Bicycle: $speed mph';

* 修饰符 @override 会告诉分析器你当前是在复写某个成员方法，如果该复写不成功，分析器就会报错。
* Dart 可以使用单引号或者双引号进行字符串的声明。
* 可以在字符串内使用 ${expression} 的方式来实现字符串模板的效果，如果该表达式仅仅是一个标识符，还可以去掉花括号 $variableName。
* 针对只有一行的方法可以使用 => 来简化方法的书写。

增加一个只读的变量

之前的 Java 示例定义了一个只读的变量 speed，并将其声明为 private 的，提供了一个 getter 方法。接下来，你将使用 Dart 来实现同样的方法。

在变量名前增加下划线 \_ 来标记为它是私有的，也就是说可以仅仅通过改变变量名来实现将 speed 标记为只读的。

将 speed 标记为私有、只读的成员变量

在 Bicycle 构造函数中，移除 speed 参数：

Bicycle(this.cadence, this.gear);

在 main() 方法中，移除 Bicycle 构造函数调用中的第二个参数 (speed) ：

var bike = Bicycle(2, 1);

初始化 \_speed 为 0：

int \_speed = 0;

在 Bicycle 类中增加下列的 getter 方法

int get speed => \_speed;

* 未初始化的变量（即使是数字类型的变量）的值都为 null
* 所有名字以下划线开头的变量，Dart 的编译器都会将其强制标记为私有的
* 默认情况下，Dart 会为所有公开的变量提供getter、setter方法。因此在之前 Java 例子中，cadence 和 gear 都有自己的getter、setter方法，而在dart中可以直接通过 bike.gear 或者 bike.cadence 访问。
* 你先直接通过 bike.cadence 来进行实例变量的访问，然后再去通过定义存取方法来进行重构。
* 若要为speed提供setter方法的，代码如下

set speed(int newSpeed) => \_speed = newSpeed;

bick.speed = 10086;

最终的 Dart 类如下：

**class** Bicycle {  
 int cadence;  
 int \_speed;  
 int gear;  
 Bicycle(**this**.cadence, **this**.gear);  
 int **get** speed => \_speed;  
 **void** applyBrake(int decrement) => \_speed -= decrement;  
 **void** speedUp(int increment) => \_speed += increment;  
 @override  
 String toString() => 'Bicycle speed is $\_speed';  
}  
**void** main() {  
 **var** bick = Bicycle(1, 1);  
 print(bick);  
}

## 使用可选参数（而不是使用重载）

Java 代码中，常常使用重载构造函数的方法，重载的构造函数和之前的构造函数具有相同的方法名，但是在方法的参数个数或者参数类型上有些许不同。Dart 并不支持构造函数的重载，而采用了另外一种方法来处理这种情况，我们将在本章接下来的地方看到它是如何实现的

java代码

Point.java

public class Point {  
 public int x = 0;  
 public int y = 0;  
 //constructor  
 public Point(int a, int b) {  
 x = a;  
 y = b;  
 }  
}

Rectangle.java

public class Rectangle {  
 public int width = 0;  
 public int height = 0;  
 public Point origin;  
 // four constructors  
 public Rectangle() {  
 origin = new Point(0, 0);  
 }  
 public Rectangle(Point p) {  
 origin = p;  
 }  
 public Rectangle(int w, int h) {  
 origin = new Point(0, 0);  
 width = w;  
 height = h;  
 }  
 public Rectangle(Point p, int w, int h) {  
 origin = p;  
 width = w;  
 height = h;  
 }  
 // a method for moving the rectangle  
 public void move(int x, int y) {  
 origin.x = x;  
 origin.y = y;  
 }  
 // a method for computing the area of the rectangle  
 public int getArea() {  
 return width \* height;  
 }  
}

增加 Rectangle 构造方法、优化文本输出

可以直接添加下列的一个没有方法体的构造方法来替代 Java 代码中的四个构造方法。

**class** Rectangle {  
 int other;

int width;  
 int height;

Point origin;  
 Rectangle(**this**.ohter, {**this**.origin = **const** Point(0, 0), **this**.width = 0, **this**.height = 0});  
 @override  
 String toString() => 'other: $other, Origin: (${origin.x}, ${origin.y}), width: $width, height: $height';  
}

* 可选命名参数必须写在闭合的花括号中 “{}”
* 可选命名参数必须写在非默认参数的后边
* 可选命名参数必须是在编译期就可以确定的常量，例如this.origin = const Point(0, 0) 这样的代码表明给实例变量 origin 提供了默认的值 Point(0,0)，

使用构造方法

main() {  
 print(Rectangle(100, origin: **const** Point(10, 20), width: 100, height: 200));  
 print(Rectangle(100, origin: **const** Point(10, 10)));  
 print(Rectangle(100, width: 200));  
 print(Rectangle(100));  
}

输出：

Origin: (10, 20), width: 100, height: 200

Origin: (10, 10), width: 0, height: 0

Origin: (0, 0), width: 200, height: 0

Origin: (0, 0), width: 0, height: 0

* 非默认参数，不可忽略
* 可选命名参数需要使用格式“xx:常量”的写法

## 创建工厂模式

在 Java 中工厂类是一个广泛使用的设计模式，它相比较直接对类进行实例化来说，具有诸多优势，比如：

* 隐藏实例化的具体细节
* 提供可以配置为其他类的能力
* 直接返回一个已有的对象而不是直接返回一个新的对象。

使用一个 Shapes 实例来实例化形状的类，并输出打印出他们的面积

**import** 'dart:math';  
**abstract class** Shape{  
 num **get** area;  
}  
**class** Circle **implements** Shape{  
 **final** num radius;  
 Circle(**this**.radius);  
 @override  
 num **get** area => pi \* pow(radius, 2);  
}  
**class** Square **implements** Shape{  
 **final** num side;  
 Square(**this**.side);  
 @override  
 num **get** area => pow(side, 2);  
}  
main() {  
 **final** circle = Circle(2);  
 **final** square = Square(2);  
 print(circle.area);  
 print(square.area);  
}

* Dart 支持抽象类
* 你可以在一个文本中定义多个类
* dart.math 是一个 Dart 核心库，其余的还有诸如 dart:core，dart:async，dart:convert 和 dart:collection 这样的核心库
* 在 Dart 1.x 的版本中，核心库中的常量是大写的（比如 PI）；在 Dart 2 的版本中，都是小写的(pi)。

选项 1：创建一个顶层的方法

在最外层作用域中（在所有类的作用域之外）实现一个工厂方法

Shape shapeFactory(String type){  
 **if**(type == 'circle') **return** Circle(2);  
 **if**(type == 'suqare') **return** Square(2);  
 **throw** 'Cant\'t creat $type';  
}

**final** circle = shapeFactory('circle');  
**final** square = shapeFactory('square');

* Dart SDK 中已经定义了许多常见的异常，也提供了通过扩展 Exception 类来进行异常的自定义，或者你也可以像上述代码中一样直接抛出字符串用来描述所出现的问题。
* 当出现异常后，DartPad 会直接抛出 Uncaught，你可以将代码封装在 try-catch 语句中来捕获异常
* 如果一个字符串中包含单引号，那么如果该字符串使用单引号声明的，则需要在字符串内部的单引号之前，增加转移符('Can\'t create $type')，如果字符串是使用双引号进行声明的("Can't create $type")，则什么都不需要做。

选项2 ：创建一个工厂模式的构造方法

使用 Dart 的 factory 关键字来创建一个工厂模式的构造方法

**abstract class** Shape{  
 **factory** Shape(String type){  
 **if**(type == 'circle') **return** Circle(2);  
 **if**(type == 'square') **return** Circle(2);  
 **throw** 'Can\'t creat $type';  
 }  
 num **get** area;  
}

**final** circle = Shape('circle');  
**final** square = Shape('square');

## 接口的实现

Dart 语言并没有提供 interface 关键字，但是每一个类都隐式地定义了一个接口。

扩展 Circle 类，增加一个 CircleMock:

class Circle implements Shape {  
 final num radius;  
 Circle(this.radius);  
 @override  
 num get area => pi \* pow(radius, 2);  
}

class CircleMock implements Circle {}

你将会看到一个"Missing concrete implementations" 的错误，添加两个实例变量 area 和 radius 即可修复这个问题

class CircleMock implements Circle {

num area;

num radius;

}

虽然 CircleMock 并没有定义任何行为，但是在 Dart 中这是完全合法的，不会有任何报错。

## 函数式编程

在函数式编程中，你可以做到：

* 将函数当做参数进行传递
* 将函数直接赋值给变量
* 对函数进行解构，只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数（也被称为柯里化）
* 创建一个可以被当做为常量的匿名函数（也被称为 lambda 表达式）

Dart 支持所有的这些特性，在 Dart 中，每个函数都是一个对象，并且每个函数都有它的类型 Function，这意味着所有函数都可以支持赋值操作，以及都可以作为参数传递给其他的函数

下列代码使用了命令式编程的方法，并没有使用函数式：

// ${'a' \* length} 这样的代码执行效果是将字符 'a' 重复 length 次

String scream(int length) => "A${'a' \* length}h!";

main() {

final values = [1, 2, 3, 5, 10, 50];

for (var length in values) {

print(scream(length));

}

}

你将会看到类似于下列的输出：

Aah!

Aaah!

Aaaah!

Aaaaaah!

Aaaaaaaaaaah!

Aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaah!

使用为函数式

values.skip(1).take(3).map(scream).forEach(print);

将会看到下列的输出：

Aaah!

Aaaah!

Aaaaaah!

* skip(1) 会忽略迭代中的第一个值
* take(3)会获取接下来的 3 个值，也就是 2，3 和 5，剩下的值都会被忽略掉

# Dart资料二：Dart语言下的Flutter

 在跨平台开领域被 JS 一统天下的今天，Dart 语言的出现无疑是一股清流。作为后来者，Dart语言有着不少Java、kotlin 和 JS 的影子，所以对于 Android 原生开发者、前端开发者而言无疑是非常友好的。

## 基本类型

var 可以定义变量，如 var tag = "666" ，这和 JS 、 Kotlin 等语言类似，同时 Dart 属于动态类型语言，支持闭包。

Dart 中 number 类型分为 int 和 double ，其中 java 中的 long 对应的也是 Dart 中的 int 类型。Dart 中没有 float 类型。

Dart 下只有 bool 型可以用于 if 等判断，不同于 JS 这种使用方式是不合法的 var g = "null"; if(g){} 。

DART中，switch 支持 String 类型。

## 变量

Dart 不需要给变量设置 setter getter方法， 这和 kotlin 等类似。Dart 中所有的基础类型、类等都继承 Object ，默认值是 NULL，自带 getter 和 setter ，而如果是 final 或者 const 的话，那么它只有一个 getter 方法。

Dart 中 final 和 const 表示常量，比如 final name = 'GSY'; const value= 1000000; 同时 static const 组合代表了静态常量。其中 const 的值在编译期确定，final 的值要到编译时才确定。（ps： Flutter 在 Release 下是 AOT 模式。）

Dart 下的数值，在作为字符串使用时，是需要显式指定的。比如：int i = 0; print("aaaa" + i); 这样并不支持，需要 print("aaaa" + i.toString()); 这样使用。这和 Java 与 JS 存在差异。所以在使用动态类型时，需要注意不要把 number 类型当做 String 使用。

DART 中数组等于列表，所以 var list = []; 和 List list = new List() 可以简单看做一样。

## 方法

  Dart 下 ?? 、??= 属于操作符，如: AA ?? "999" 表示如果 AA 为空，返回999；AA ??= "999" 表示如果 AA 为空，给 AA 设置成 999。

Dart 方法可以设置 参数默认值 和 指定名称 。比如： getDetail(Sting userName, reposName, {branch = "master"}){} 方法，这里 branch 不设置的话，默认是 “master” 。参数类型 可以指定或者不指定。调用效果： getRepositoryDetailDao(“aaa", "bbbb", branch: "dev");

  Dart 不像 Java ，没有关键词 public 、private 等修饰符，\_下横向直接代表 private ，但是有 @protected 注解。

  Dart 中多构造函数，可以通过如下代码实现的。默认构造方法只能有一个，而通过Model.xx() 方法可以创建一个其他构造方法的MoDel类，方法名称随你喜欢。而变量初始化值时，只需要通过 this.name 在构造方法中指定即可：

**class** ModelA {  
 String name;  
 String tag;  
 //默认构造方法，赋值给name和tag  
 ModelA(**this**.name, **this**.tag);  
 //返回一个空的ModelA  
 ModelA.empty();  
 //返回一个设置了name的ModelA  
 ModelA.forName(**this**.name);  
}

**class** ModelA {  
 String name;  
 String tag;  
 //默认构造方法，空的ModelA  
 ModelA.empty();  
 //返回一个设置了name和tag的ModelA  
 ModelA(**this**.name, **this**.tag);  
 //返回一个设置了name的ModelA  
 ModelA.forName(**this**.name);  
}

## Flutter

Flutter 中支持 async/await 。这一点和 ES7 很像，如下代码所示，只是定义的位置不同。同时异步操作也和 ES6 中的Promise 很像，只是 Flutter 中返回的是 Future 对象，通过 then 可以执行下一步。如果返回的还是 Future 便可以 then().then.() 的流式操作了

*///模拟等待两秒，返回OK*request() **async** {  
 **await** Future.delayed(Duration(seconds: 1));  
 **return** "ok!";  
}  
*///得到"ok!"后，将"ok!"修改为"ok from request"*doSomeThing() **async** {  
 String data = **await** request();  
 data = "ok from request";  
 **return** data;  
}  
*///打印结果*renderSome() {  
 doSomeThing().then((value) {  
 print(value);  
 *///输出ok from request* });  
}

Flutter 中 setState 很有 React Native 的既视感，Flutter 中也是通过 state 跨帧实现管理数据状态的，这个后面会详细讲到。

Flutter 中一切皆 Widget 呈现，通过 build方法返回 Widget，这也是和 React Native 中，通过 render 函数返回需要渲染的 component 一样的模式

# Flutter Widget

在 Flutter 中，一切的显示都是 Widget 。Widget 是一切的基础，作为响应式的渲染，属于 MVVM 的实现机制。我们可以通过修改数据，再用setState 设置数据，Flutter 会自动通过绑定的数据更新 Widget 。所以你需要做的就是实现 Widget 界面，并且和数据绑定起来。

Widget 分为 有状态 和 无状态 两种，在 Flutter 中每个页面都是一帧。无状态就是保持在那一帧。而有状态的 Widget 当数据更新时，其实是绘制了新的 Widget，只是 State 实现了跨帧的数据同步保存。

## 无状态StatelessWidget

继承自 StatelessWidget，通过 build 方法返回一个布局好的控件。

Widget 和 Widget 之间通过 child: 进行嵌套。其中有的 Widget 只能有一个 child，比如下方的 Container ；有的 Widget 可以多个 child ，也就是children:，比如` Colum 布局。下方代码便是 Container Widget 嵌套了 Text Widget

**import** 'package:flutter/material.dart';  
**class** DEMOWidget **extends** StatelessWidget {  
 **final** String text;  
 //数据可以通过构造方法传递进来  
 DEMOWidget(**this**.text);  
 @override  
 Widget build(BuildContext context) {  
 //这里返回你需要的控件  
 //这里末尾有没有的逗号，对于格式化代码而言是不一样的。  
 **return** Container(  
 //白色背景  
 color: Colors.*white*,  
 //Dart语法中，?? 表示如果text为空，就返回尾号后的内容。  
 child: Text(text ?? "这就是无状态DMEO"),  
 );  
 }  
}

## 有状态StatefulWidget

需要创建管理的是主要是 State ， 通过 State 的 build 方法去构建控件。在 State 中，可以动态改变数据，这类似 MVVM 实现，在 setState 之后，改变的数据会触发 Widget 重新构建刷新。而下方代码中，是通过延两秒之后，让文本显示为 "这就变了数值"。State 中主要的声明周期有 ：

* initState ：初始化，理论上只有初始化一次，第二篇中会说特殊情况下。
* didChangeDependencies：在 initState 之后调用，此时可以获取其他 State 。
* dispose ：销毁，只会调用一次。

Flutter 其实就是这么简单！你的关注点只要在：创建你的 StatelessWidget 或者 StatefulWidget 而已。你需要的就是在 build 中堆积你的布局，然后把数据添加到 Widget 中，最后通过 setState 改变数据，从而实现画面变化

**import** 'dart:async';  
**import** 'package:flutter/material.dart';  
**class** DemoStateWidget **extends** StatefulWidget {  
 **final** String text;  
 DemoStateWidget(**this**.text);*///通过构造方法传值*  
 *///主要是负责创建state* @override  
 \_DemoStateWidgetState createState() => \_DemoStateWidgetState(text);  
}  
**class** \_DemoStateWidgetState **extends** State<DemoStateWidget> {  
 String text;  
 \_DemoStateWidgetState(**this**.text);  
 @override  
 **void** initState() {  
 **super**.initState();*///初始化，这个函数在生命周期中只调用一次*  
 *///定时2秒* **new** Future.delayed(**const** Duration(seconds: 1), () {  
 setState(() {  
 text = "这就变了数值";  
 });  
 });  
 }  
 @override  
 **void** dispose() {**super**.dispose();*///销毁*  
 }  
 @override  
 **void** didChangeDependencies() {  
 *///在initState之后调 Called when a dependency of this [State] object changes.* **super**.didChangeDependencies();  
 }  
 @override  
 Widget build(BuildContext context) {  
 **return** Container(  
 child: Text(text ?? "这就是有状态DMEO"),  
 );  
 }  
}

# Flutter 布局

Flutter 中拥有需要将近30种内置的 布局Widget，其中常用有 Container、Padding、Center、Flex、Stack、Row、Colum、ListView 等，下面简单讲解它们的特性和使用。

| **类型** | **作用特点** |
| --- | --- |
| Container | 一个子 Widget。默认充满，包含了padding、margin、color、宽高、decoration 等配置。 |
| Padding | 一个子 Widget。只用于设置Padding，常用于嵌套child，给child设置padding。 |
| Center | 一个子 Widget。只用于居中显示，常用于嵌套child，给child设置居中。 |
| Stack | 多个子 Widget。 子Widget堆叠在一起。 |
| Colum | 多个子 Widget。垂直布局。 |
| Row | 多个子 Widget。水平布局。 |
| Expanded | 一个子 Widget。在 Colum 和 Row 中充满。 |
| ListView | 多个子 Widget。ListView。 |

## Container

最常用的默认布局！只能包含一个child:，支持配置 padding,margin,color,宽高,decoration（一般配置边框和阴影）等配置，在 Flutter 中，不是所有的控件都有 宽高、padding、margin、color 等属性，所以才会有 Padding、Center 等 Widget 的存在

**var** x = **new** Container(  
 *///四周10大小的maring* margin: EdgeInsets.all(10.0),  
 height: 120.0,  
 width: 500.0,  
 *///透明黑色遮罩* decoration: **new** BoxDecoration(  
 *///弧度为4.0* borderRadius: BorderRadius.all(Radius.circular(4.0)),  
 *///设置了decoration的color，就不能设置Container的color。* color: Colors.*black*,  
 *///边框* border:  
 **new** Border.all(color: Color(GSYColors.subTextColor), width: 0.3)),  
 child: **new** Text("666666"));

## Colum、Row

绝对是必备布局， 横竖布局也是日常中最常见的场景。它们常用的有这些属性配置：主轴方向是 start 或 center 等；副轴方向方向是 start 或 center 等；mainAxisSize 是充满最大尺寸，或者只根据子 Widget 显示最小尺寸

//主轴方向，Colum的竖向、Row我的横向

mainAxisAlignment: MainAxisAlignment.start,

//默认是最大充满、还是根据child显示最小大小

mainAxisSize: MainAxisSize.max,

//副轴方向，Colum的横向、Row我的竖向

crossAxisAlignment :CrossAxisAlignment.center,

## Expanded

在 Colum 和 Row 中代表着平均充满，当有两个存在的时候默认均分充满。同时页可以设置 flex 属性决定比例

**var** x = **new** Column(  
 *///主轴居中,即是竖直向居中* mainAxisAlignment: MainAxisAlignment.center,  
 *///大小按照最小显示* mainAxisSize: MainAxisSize.min,  
 *///横向也居中* crossAxisAlignment: CrossAxisAlignment.center,  
 children: <Widget>[  
 *///flex默认为1* **new** Expanded(  
 child: **new** Text("1111"),  
 flex: 2,  
 ),  
 **new** Expanded(child: **new** Text("2222")),  
 ],  
);

## 使用

接下来我们来写一个复杂一些的控件。布局内主要是现实一个居中的Icon图标和文本，中间间隔5.0的 padding。首先我们创建一个私有方法\_getBottomItem，返回一个 Expanded Widget，因为后面我们需要将这个方法返回的 Widget 在 Row 下平均充满。

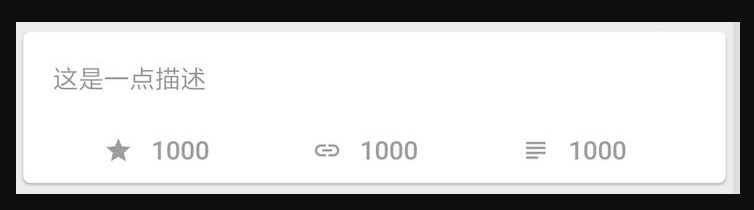
*///返回一个居中带图标和文本的Item*\_getBottomItem(IconData icon, String text) {  
 *///充满 Row 横向的布局* **return new** Expanded(  
 flex: 1,  
 *///居中显示* child: **new** Center(  
 *///横向布局* child: **new** Row(  
 *///主轴居中,即是横向居中* mainAxisAlignment: MainAxisAlignment.center,  
 *///大小按照最大充满* mainAxisSize: MainAxisSize.max,  
 *///竖向也居中* crossAxisAlignment: CrossAxisAlignment.center,  
 children: <Widget>[  
 *///一个图标，大小16.0，灰色* **new** Icon(  
 icon,  
 size: 16.0,  
 color: Colors.*grey*,  
 ),  
 *///间隔* **new** Padding(padding: **new** EdgeInsets.only(left: 5.0)),  
 *///显示文本* **new** Text(  
 text,  
 //设置字体样式：颜色灰色，字体大小14.0  
 style: **new** TextStyle(color: Colors.*grey*, fontSize: 14.0),  
 //超过的省略为...显示  
 overflow: TextOverflow.ellipsis,  
 //最长一行  
 maxLines: 1,  
 ),  
 ],  
 ),  
 ),  
 );  
}



接着我们把上方的方法，放到新的布局里。如下流程和代码：

* 首先是 Container包含了Card，用于快速简单的实现圆角和阴影。
* 然后接下来包含了FlatButton实现了点击，通过Padding实现了边距。
* 接着通过Column垂直包含了两个子Widget，一个是Container、一个是Row。
* Row 内使用的就是\_getBottomItem方法返回的 Widget

@override  
Widget build(BuildContext context) {  
 **return new** Container(  
 *///卡片包装* child: **new** Card(  
 *///增加点击效果* child: **new** FlatButton(  
 onPressed: () {  
 print("点击了哦");  
 },  
 child: **new** Padding(  
 padding: **new** EdgeInsets.only(  
 left: 0.0, top: 10.0, right: 10.0, bottom: 10.0),  
 child: **new** Column(  
 mainAxisSize: MainAxisSize.min,  
 children: <Widget>[  
 *///文本描述* **new** Container(  
 child: **new** Text(  
 "这是一点描述",  
 style: TextStyle(  
 color: Color(GSYColors.subTextColor),  
 fontSize: 14.0,  
 ),  
 *///最长三行，超过 ... 显示* maxLines: 3,  
 overflow: TextOverflow.ellipsis,  
 ),  
 margin: **new** EdgeInsets.only(top: 6.0, bottom: 2.0),  
 alignment: Alignment.*topLeft*),  
 **new** Padding(padding: EdgeInsets.all(10.0)),  
 *///三个平均分配的横向图标文字* **new** Row(  
 crossAxisAlignment: CrossAxisAlignment.start,  
 children: <Widget>[  
 \_getBottomItem(Icons.*star*, "1000"),  
 \_getBottomItem(Icons.*link*, "1000"),  
 \_getBottomItem(Icons.*subject*, "1000"),  
 ],  
 ),  
 ],  
 ),  
 ))),  
 );  
}



# Flutter 页面

Flutter 中除了布局的 Widget，还有交互显示的 Widget 和完整页面呈现的Widget。其中常见的有 MaterialApp、Scaffold、Appbar、Text、Image、FlatButton等。下面简单介绍这些 Wdiget，并完成一个页面

| **类型** | **作用特点** |
| --- | --- |
| MaterialApp | 一般作为APP顶层的主页入口，可配置主题，多语言，路由等 |
| Scaffold | 一般用户页面的承载Widget，包含appbar、snackbar、drawer等material design的设定。 |
| Appbar | 一般用于Scaffold的appbar ，内有标题，二级页面返回按键等，tabbar等也会需要它 。 |
| Text | 显示文本，几乎都会用到，主要是通过style设置TextStyle来设置字体样式等。 |
| RichText | 富文本，通过设置TextSpan，可以拼接出富文本场景。 |
| TextField | 文本输入框 ：new TextField(controller: //文本控制器, obscureText: "hint文本"); |
| Image | 图片加载: new FadeInImage.assetNetwork( placeholder: "预览图", fit: BoxFit.fitWidth, image: "url"); |
| FlatButton | 按键点击: new FlatButton(onPressed: () {},child: new Container()); |

实现一个简单完整的页面试试：

* 首先我们创建一个StatefulWidget：DemoPage。
* 然后在 \_DemoPageState中，通过build创建了一个Scaffold。
* Scaffold内包含了一个AppBar和一个ListView。
* AppBar类似标题了区域，其中设置了 title为 Text Widget。
* body是ListView,返回了20个之前我们创建过的 DemoItem Widget。

**class** DemoPage **extends** StatefulWidget {  
 @override  
 \_DemoPageState createState() => \_DemoPageState();  
}  
**class** \_DemoPageState **extends** State<DemoPage> {  
 @override  
 Widget build(BuildContext context) {  
 *///一个页面的开始  
 ///如果是新页面，会自带返回按键* **return new** Scaffold(  
 *///背景样式* backgroundColor: Colors.*blue*,  
 *///标题栏，当然不仅仅是标题栏* appBar: **new** AppBar(  
 *///这个title是一个Widget* title: **new** Text("Title"),  
 ),  
 *///正式的页面开始  
 ///一个ListView，20个Item* body: **new** ListView.builder(  
 itemBuilder: (context, index) {  
 **return new** DemoItem();  
 },  
 itemCount: 20,  
 ),  
 );  
 }  
}

最后我们创建一个StatelessWidget作为入口文件，实现一个MaterialApp将上方的DemoPage设置为home页面，通过main入口执行页面

**void** main() {  
 runApp(**new** DemoApp());  
}  
**class** DemoApp **extends** StatelessWidget {  
 DemoApp({Key key}) : **super**(key: key);  
 @override  
 Widget build(BuildContext context) {  
 **return new** MaterialApp(home: DemoPage());  
 }  
}