# 第一部分Flutter基础篇

## 一 新建Flutter项目

略，可以参考我的博客

## 二 Dart语言基础

### 1 Dart数据类型

#### 1 Dart语言关键字

关键字如下



#### 2 变量（Variable）

变量赋值的例子

var name = 'Bob';

没有初始化的变量都会被赋予默认值 null.

即使是数字也是如此， 因为在Dart 中数字也是一个对象

可选类型

也可以在定义的时候指定变量的类型。

String name = 'Bob';

指定数据类型可以更好的辨明自己的使用意图，编译器和IDE 工具可以根据这些类型信息来做检查，更早的发现问题。

如前文所说，通过指定类型，也可以减少编译和运行时间。

#### 3 常量和固定值

如果定义的变量不会变化，可以使用final 或 const来指明。

也可以使用final 或 const来代替类型声明。

final name = 'Bob'; // Or: final String name = 'Bob';

// name = 'Alice'; // Uncommenting this causes an error

final的值只能被设定一次。

const 是一个编译时的常量。( Const variables are implicitly final.)

const name = "hello";

通过对const类型做四则运算将自动得到一个const类型的值。

const bar = 1000000; // Unit of pressure (dynes/cm2)

const atm = 1.01325 \* bar; // Standard atmosphere

可以通过const来创建常量的值

就是说const[] 本身是构造函数。

// Note: [] creates an empty list.

// const [] creates an empty, immutable list (EIA).

var foo = const []; // foo is currently an EIA.

final bar = const []; // bar will always be an EIA.

const baz = const []; // baz is a compile-time constant EIA.

// You can change the value of a non-final, non-const variable,

// even if it used to have a const value.

foo = [];

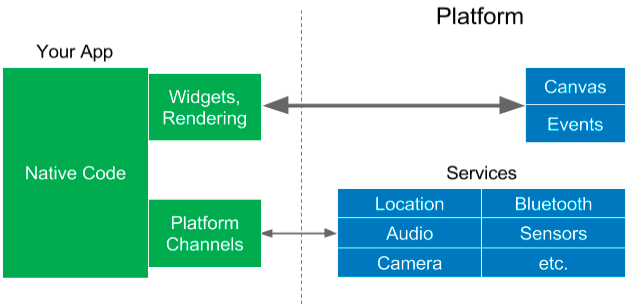
// You can't change the value of a final or const variable.

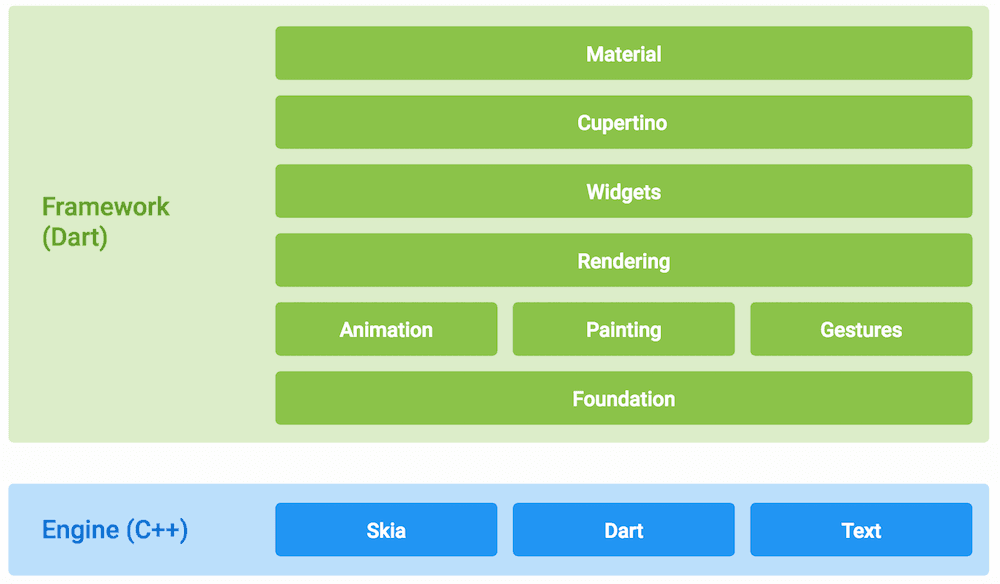
// bar = []; // Unhandled exception.

// baz = []; // Unhandled exception

### 2 面向对象

# 第二部分 Flutter原理





# 第三部分 Flutter基础控件

## 1 Flutter UI控件

### 1 MaterialApp

Android MD设计所必须要的组件，一般作为顶层widget使用。

继承关系

Inheritance

Object->Diagnosticable ->DiagnosticableTree ->Widget ->StatefulWidget ->MaterialApp

一般与以下widget一起使用

Scaffold: Material Design布局结构的基本实现。此类提供了用于显示drawer、snackbar和底部sheet的API。

Navigator，用于管理应用程序的页面堆栈。

MaterialPageRoute，它定义以特定于材料的方式转换的应用页面。

WidgetsApp，它定义基本的app元素但不依赖于材质库。

### 2 Scaffold

Material Design布局结构的基本实现。此类提供了用于显示drawer、snackbar和底部sheet的API。

简单来说，Scanold就是一个提供MD设计中基本布局的widget,包括最上面的appBar,body，以及下部的drawer,snackbar等

继承关系

Object –>Diagnosticable ->DiagnosticableTree ->Widget ->StatefulWidget ->Scaffold

常用元素

appBar 标题栏

backgroundColor 背景色

body 内容区

示例

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(

title: Text("测试Demo"),

centerTitle: true,

),

body: MyContainer(),

backgroundColor: Colors.grey,

));

}

}

### 3 Container

定义：是一个结合了绘制（painting）、定位（positioning）以及尺寸（sizing）widget的widget。

可以得出几个信息，它是一个组合的widget，内部有绘制widget、定位widget、尺寸widget。后续看到的不少widget，都是通过一些更基础的widget组合而成的。

#### 1.1 组成

Container的组成如下：

最里层的是child元素；

child元素首先会被padding包着；

然后添加额外的constraints限制；

最后添加margin。

Container的绘制的过程如下：

首先会绘制transform效果；

接着绘制decoration；

然后绘制child；

最后绘制foregroundDecoration。

Container自身尺寸的调节分两种情况：

Container在没有子节点（children）的时候，会试图去变得足够大。除非constraints是unbounded限制，在这种情况下，Container会试图去变得足够小。

带子节点的Container，会根据子节点尺寸调节自身尺寸，但是Container构造器中如果包含了width、height以及constraints，则会按照构造器中的参数来进行尺寸的调节。

#### 1.2 布局行为

由于Container组合了一系列的widget，这些widget都有自己的布局行为，因此Container的布局行为有时候是比较复杂的。

一般情况下，Container会遵循如下顺序去尝试布局：

对齐（alignment）；

调节自身尺寸适合子节点；

采用width、height以及constraints布局；

扩展自身去适应父节点；

调节自身到足够小。

进一步说：

如果没有子节点、没有设置width、height以及constraints，并且父节点没有设置unbounded的限制，Container会将自身调整到足够小。

如果没有子节点、对齐方式（alignment），但是提供了width、height或者constraints，那么Container会根据自身以及父节点的限制，将自身调节到足够小。

如果没有子节点、width、height、constraints以及alignment，但是父节点提供了bounded限制，那么Container会按照父节点的限制，将自身调整到足够大。

如果有alignment，父节点提供了unbounded限制，那么Container将会调节自身尺寸来包住child；

如果有alignment，并且父节点提供了bounded限制，那么Container会将自身调整的足够大（在父节点的范围内），然后将child根据alignment调整位置；

含有child，但是没有width、height、constraints以及alignment，Container会将父节点的constraints传递给child，并且根据child调整自身。

另外，margin以及padding属性也会影响到布局

#### 1.3 属性解析

key：Container唯一标识符，用于查找更新。

alignment：控制child的对齐方式，如果container或者container父节点尺寸大于child的尺寸，这个属性设置会起作用，有很多种对齐方式。

padding：decoration内部的空白区域，如果有child的话，child位于padding内部。padding与margin的不同之处在于，padding是包含在content内，而margin则是外部边界，设置点击事件的话，padding区域会响应，而margin区域不会响应。

color：用来设置container背景色，如果foregroundDecoration设置的话，可能会遮盖color效果。

decoration：绘制在child后面的装饰，设置了decoration的话，就不能设置color属性，否则会报错，此时应该在decoration中进行颜色的设置。

foregroundDecoration：绘制在child前面的装饰。

width：container的宽度，设置为double.infinity可以强制在宽度上撑满，不设置，则根据child和父节点两者一起布局。

height：container的高度，设置为double.infinity可以强制在高度上撑满。

constraints：添加到child上额外的约束条件。

margin：围绕在decoration和child之外的空白区域，不属于内容区域。

transform：设置container的变换矩阵，类型为Matrix4。

child：container中的内容widget。

#### 1.4 使用示例

class MyContainer extends StatelessWidget{

String \_image\_url = 'http://h.hiphotos.baidu.com/zhidao/wh%3D450%2C600/sign=0d023672312ac65c67506e77cec29e27/9f2f070828381f30dea167bbad014c086e06f06c.jpg';

@override

Widget build(BuildContext context) {

// 新的Container布局

return Container(

constraints: BoxConstraints.expand(

height: Theme.of(context).textTheme.display1.fontSize \* 1.1 + 200.0,

),

decoration: BoxDecoration(

//边界

border: Border.all(width: 2.0, color: Colors.red),

//边角

borderRadius: BorderRadius.all(Radius.circular(20.0)),

//背景

image: DecorationImage(

image: NetworkImage(\_image\_url),

centerSlice: Rect.fromLTRB(270.0, 180.0, 1360.0, 730.0),

),

color: Colors.lightGreen

),

//padding

padding: EdgeInsets.all(8.0),

//对齐方式

alignment: Alignment.center,

//内容

child: Text("Hello World",

style: Theme.of(context).textTheme.display1.copyWith(color: Colors.black),

),

//矩阵变换属性

transform: Matrix4.rotationZ(0.5),

);

}

}

#### 1.5 使用场景

Container算是目前项目中，最经常用到的一个widget。在实际使用过程中，笔者在以下情况会使用到Container，当然并不是绝对的，也可以通过其他widget来实现。

需要设置间隔（这种情况下，如果只是单纯的间隔，也可以通过Padding来实现）；

需要设置背景色；

需要设置圆角或者边框的时候（ClipRRect也可以实现圆角效果）；

需要对齐（Align也可以实现）；

需要设置背景图片的时候（也可以使用Stack实现）。

#### 1.6 应用

自定义Button按钮

/// 定义一个Button

class MyButton extends StatefulWidget {

/// button 的三种状态

static const defaultBackgroundColor = const Color(0xFF8B5FFE);

static const defaultActiveBackgroundColor = const Color(0xB38B5FFE);

static const defaultDisabledBackgroundColor = Colors.grey;

final Widget title;

final Color backgroundColor;

final Color activeColor;

final Color disableColor;

final double height;

final double width;

final bool disable;

//点击事件

final VoidCallback onPress;

MyButton({

this.title,

this.backgroundColor = defaultBackgroundColor,

this.activeColor = defaultActiveBackgroundColor,

this.disableColor = defaultDisabledBackgroundColor,

this.height = 52.0,

this.width = double.infinity,

this.disable = false,

this.onPress});

@override

State<StatefulWidget> createState() {

// TODO: implement createState

return \_MyButtonState();

}

}

class \_MyButtonState extends State<MyButton> {

Color currentColor;

@override

void initState() {

// TODO: implement initState

super.initState();

//此处的widget就是上面的widget

if (widget.disable) {

currentColor = widget.disableColor;

} else {

currentColor = widget.backgroundColor;

}

}

//销毁时调用

@override

void deactivate() {

// TODO: implement deactivate

super.deactivate();

currentColor = widget.backgroundColor;

}

@override

Widget build(BuildContext context) {

return new GestureDetector(

onTap: () {

if (widget.onPress != null && !widget.disable) {

widget.onPress();

}

},

//按下，输入参数

onTapDown: (TapDownDetails details) {

print("x: ${details..toString()}");

if (!widget.disable) {

//更新状态

setState(() {

currentColor = widget.activeColor;

});

}

},

//松开

onTapUp: (TapUpDetails details) {

if (!widget.disable) {

//更新状态

setState(() {

currentColor = widget.backgroundColor;

});

}

},

//取消

onTapCancel: () {

if (!widget.disable) {

setState(() {

currentColor = widget.backgroundColor;

});

}

},

child: new Container(

decoration: BoxDecoration(

color: currentColor,

borderRadius: BorderRadius.all(Radius.circular(widget.height / 2)),

),

height: widget.height,

width: widget.width,

alignment: Alignment.center,

child: widget.title,

),

);

}

}

### 4 Padding、Align、Center

FittedBox、AspectRatio、ConstrainedBox

Baseline、FractionallySizedBox、IntrinsicHeight、IntrinsicWidth

LimitedBox、Offstage、OverflowBox、SizedBox

SizedOverflowBox、Transform、CustomSingleChildLayout

Row、Column

Stack、IndexedStack、GridView

Flow、Table、Wrap

ListBody、ListView、CustomMultiChildLayout