# 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 = [];

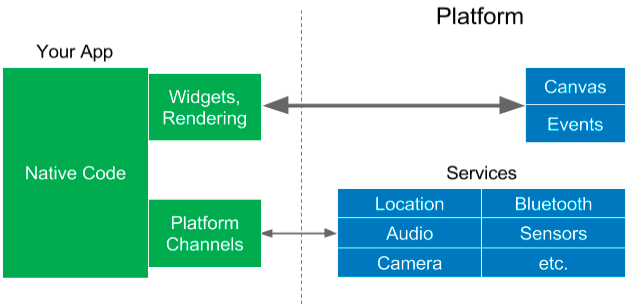
// You can't change the value of a final or const variable.

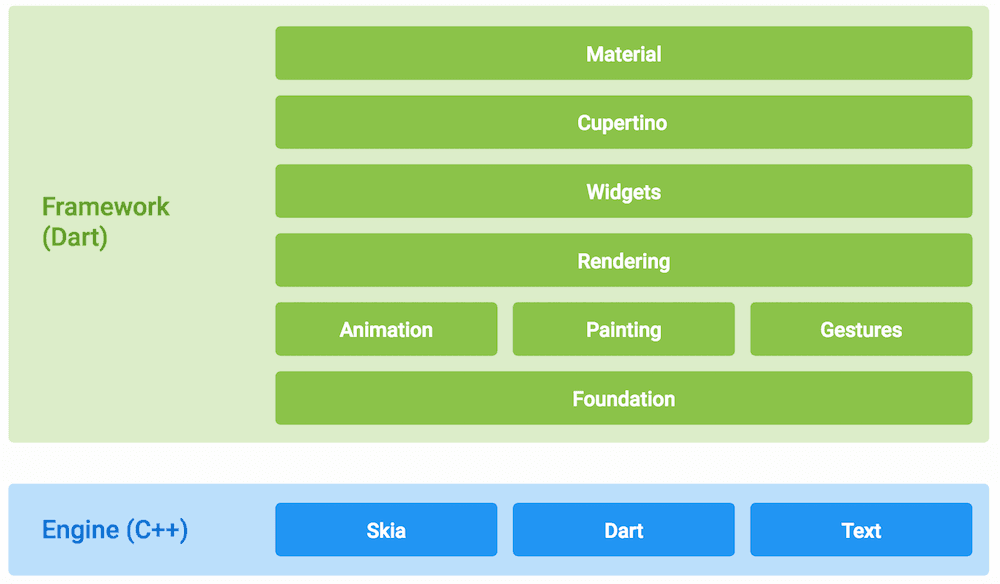
// bar = []; // Unhandled exception.

// 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

#### 1 Padding

Padding在Flutter中用的也挺多的，作为一个基础的控件，功能非常单一，给子节点设置padding属性。写过其他端的都了解这个属性，就是设置内边距属性，内边距的空白区域，也是widget的一部分

1.2 布局行为

Padding的布局分为两种情况：

当child为空的时候，会产生一个宽为left+right，高为top+bottom的区域；

当child不为空的时候，Padding会将布局约束传递给child，根据设置的padding属性，缩小child的布局尺寸。然后Padding将自己调整到child设置了padding属性的尺寸，在child周围创建空白区域。

示例代码

new Padding(

padding: new EdgeInsets.all(8.0),

child: const Card(child: const Text('Hello World!')),

)

padding：padding的类型为EdgeInsetsGeometry，EdgeInsetsGeometry是EdgeInsets以及EdgeInsetsDirectional的基类。在实际使用中不涉及到国际化，例如适配阿拉伯地区等，一般都是使用EdgeInsets。EdgeInsetsDirectional光看命名就知道跟方向相关，因此它的四个边距不限定上下左右，而是根据方向来定的。

使用场景

Padding本身还是挺简单的，基本上需要间距的地方，它都能够使用。如果在单一的间距场景，使用Padding比Container的成本要小一些，毕竟Container里面包含了多个widget。Padding能够实现的，Container都能够实现，只不过，Container更加的复杂。

#### 2 Align

在其他端的开发，Align一般都是当做一个控件的属性，并没有拿出来当做一个单独的控件。Align本身实现的功能并不复杂，设置child的对齐方式，例如居中、居左居右等，并根据child尺寸调节自身尺寸。

布局行为

Align的布局行为分为两种情况：

当widthFactor和heightFactor为null的时候，当其有限制条件的时候，Align会根据限制条件尽量的扩展自己的尺寸，当没有限制条件的时候，会调整到child的尺寸；

当widthFactor或者heightFactor不为null的时候，Aligin会根据factor属性，扩展自己的尺寸，例如设置widthFactor为2.0的时候，那么，Align的宽度将会是child的两倍。

Align为什么会有这样的布局行为呢？原因很简单，设置对齐方式的话，如果外层元素尺寸不确定的话，内部的对齐就无法确定。因此，会有宽高因子、根据外层限制扩大到最大尺寸、外层不确定时调整到child尺寸这些行为。

示例代码

new Align(

alignment: Alignment.center,

widthFactor: 2.0,

heightFactor: 2.0,

child: new Text("Align"),

)

属性解析

alignment：对齐方式，一般会使用系统默认提供的9种方式，但是并不是说只有这9种，例如如下的定义。系统提供的9种方式只是预先定义好的。

/// The top left corner.

static const Alignment topLeft = const Alignment(-1.0, -1.0);

Alignment实际上是包含了两个属性的，其中第一个参数，-1.0是左边对齐，1.0是右边对齐，第二个参数，-1.0是顶部对齐，1.0是底部对齐。根据这个规则，我们也可以自定义我们需要的对齐方式，例如

/// 居右高于底部1/4处.

static const Alignment rightHalfBottom = alignment: const Alignment(1.0, 0.5),

widthFactor：宽度因子，如果设置的话，Align的宽度就是child的宽度乘以这个值，不能为负数。

heightFactor：高度因子，如果设置的话，Align的高度就是child的高度乘以这个值，不能为负数。

使用场景

一般在对齐场景下使用，例如需要右对齐或者底部对齐之类的。它能够实现的功能，Container都能实现

Center

Center继承自Align，只不过是将alignment设置为Alignment.center，其他属性例如widthFactor、heightFactor，布局行为，都与Align完全一样，在这里就不再单独做介绍了。Center源码如下，没有设置alignment属性，是因为Align默认的对齐方式就是居中。

class Center extends Align {

/// Creates a widget that centers its child.

const Center({ Key key, double widthFactor, double heightFactor, Widget child })

: super(key: key, widthFactor: widthFactor, heightFactor: heightFactor, child: child);

}

### 5 FittedBox、AspectRatio、ConstrainedBox

#### 1 FittedBox

简介

按照其官方的介绍，它主要做了两件事情，缩放（Scale）以及位置调整（Position）。

FittedBox会在自己的尺寸范围内缩放并且调整child位置，使得child适合其尺寸。做过移动端的，可能会联想到ImageView控件，它是将图片在其范围内，按照规则，进行缩放位置调整。FittedBox跟ImageView是有些类似的，可以猜测出，它肯定有一个类似于ScaleType的属性。

布局行为

FittedBox的布局行为还算简单，官方没有给出说明，我在这里简单说一下。由于FittedBox是一个容器，需要让其child在其范围内缩放，因此其布局行为分两种情况：

如果外部有约束的话，按照外部约束调整自身尺寸，然后缩放调整child，按照指定的条件进行布局；

如果没有外部约束条件，则跟child尺寸一致，指定的缩放以及位置属性将不起作用。

示例代码

class MyFittedBox extends StatelessWidget{

@override

Widget build(BuildContext context) {

// TODO: implement build

return Container(

color: Colors.amberAccent,

width: 300.0,

height: 400.0,

child: FittedBox(

fit:BoxFit.contain,

alignment: Alignment.topCenter,

child: Container(

color: Colors.red,

child: Text("FittedBox",

style: TextStyle(

color: Colors.blue,

fontSize: 12.0,

),),

),

),

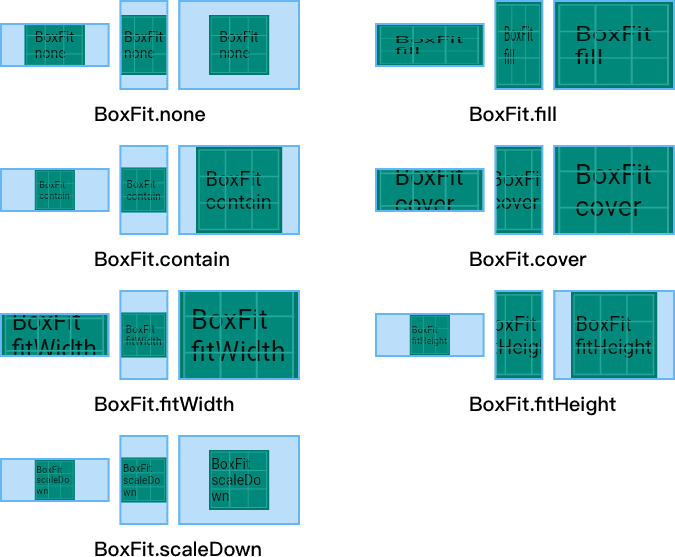
);

}

}

属性解析

fit：缩放的方式，默认的属性是BoxFit.contain，child在FittedBox范围内，尽可能的大，但是不超出其尺寸。这里注意一点，contain是保持着child宽高比的大前提下，尽可能的填满，一般情况下，宽度或者高度达到最大值时，就会停止缩放。



BoxFit布局表现

alignment：对齐方式，默认的属性是Alignment.center，居中显示child。

使用场景

FittedBox在目前的项目中还未用到过。对于需要缩放调整位置处理的，一般都是图片。笔者一般都是使用Container中的decoration属性去实现相应的效果。对于其他控件需要缩放以及调整位置的，目前还没有遇到使用场景，大家只需要知道有这么一个控件，可以实现这个功能即可。

#### 2 AspectRatio

AspectRatio的作用是调整child到设置的宽高比，这种控件在其他移动端平台上一般都不会提供，Flutter之所以提供，我想最大的原因，可能就是自定义起来特别麻烦吧。

布局行为

AspectRatio的布局行为分为两种情况：

* AspectRatio首先会在布局限制条件允许的范围内尽可能的扩展，widget的高度是由宽度和比率决定的，类似于BoxFit中的contain，按照固定比率去尽量占满区域。
* 如果在满足所有限制条件过后无法找到一个可行的尺寸，AspectRatio最终将会去优先适应布局限制条件，而忽略所设置的比率。

示例代码

new Container(

height: 200.0,

child: new AspectRatio(

aspectRatio: 1.5,

child: new Container(

color: Colors.red,

),

),

);

属性解析

aspectRatio：aspectRatio是宽高比，最终可能不会根据这个值去布局，具体则要看综合因素，外层是否允许按照这种比率进行布局，只是一个参考值。

使用场景

AspectRatio适用于需要固定宽高比的情景下。笔者最近使用这个控件的场景是相机，相机的预览尺寸都是固定的几个值，因此不能随意去设置相机的显示区域，必须按照比率进行显示，否则会出现拉伸的情况。除此之外，倒是用的不多。

#### 3 ConstrainedBox

这个控件的作用是添加额外的限制条件（constraints）到child上，本身挺简单的，可以被一些控件替换使用。Flutter的布局控件体系，梳理着发现确实有点乱，感觉总体思想是缺啥就造啥，哈哈

布局行为

ConstrainedBox的布局行为非常简单，取决于设置的限制条件，而关于父子节点的限制因素生效优先级，可以查看之前的文章，在这里就不做具体叙述了

示例代码

class MyConstrainedBox extends StatelessWidget {

@override

Widget build(BuildContext context) {

return ConstrainedBox(

constraints: BoxConstraints(

minWidth: 100.0,

minHeight: 100.0,

maxHeight: 250.0,

maxWidth: 300.0

),

child: Container(

height: 150.0,

color: Colors.greenAccent,

),

);

}

}

例子也挺简单的，在一个宽高200.0的Container上添加一个约束最大最小宽高的ConstrainedBox，实际的显示中，则是一个宽高为150.0的区域。

属性解析

constraints：添加到child上的额外限制条件，其类型为BoxConstraints。BoxConstraints的作用是干啥的呢？其实很简单，就是限制各种最大最小宽高。说到这里插一句，double.infinity在widget布局的时候是合法的，也就说，例如想最大的扩展宽度，可以将宽度值设为double.infinity。

使用场景

需要添加额外的约束条件可以使用此控件，例如设置最小宽高，尽可能的占用区域等等。笔者在实际开发中使用的倒不是很多，倒不是说这个控件不好使用，而是好多约束因素是综合的，例如需要额外的设置margin、padding属性能，因此单独再套个这个就显得很繁琐了。

关于UnconstrainedBox

这个控件不做详细介绍了，它跟ConstrainedBox相反，是不添加任何约束条件到child上，让child按照其原始的尺寸渲染。

很神奇是吧，我也觉得，其实它的作用就是给child一个尽可能大的空间，不加约束的让其显示。用处我暂时木有想到。只能说Flutter生产Widget很随性

### 6 Baseline、FractionallySizedBox、IntrinsicHeight、IntrinsicWidth

### 7 LimitedBox、Offstage、OverflowBox、SizedBox

### 8 SizedOverflowBox、Transform、CustomSingleChildLayout

### 9 Row、Column

#### 1 Row

在Flutter中非常常见的一个多子节点控件，将children排列成一行。估计是借鉴了Web中Flex布局，所以很多属性和表现，都跟其相似。但是注意一点，自身不带滚动属性，如果超出了一行，在debug下面则会显示溢出的提示

布局行为

Row的布局有六个步骤，这种布局表现来自Flex（Row和Column的父类）：

首先按照不受限制的主轴（main axis）约束条件，对flex为null或者为0的child进行布局，然后按照交叉轴（ cross axis）的约束，对child进行调整；

按照不为空的flex值，将主轴方向上剩余的空间分成相应的几等分；

对上述步骤flex值不为空的child，在交叉轴方向进行调整，在主轴方向使用最大约束条件，让其占满步骤2所分得的空间；

Flex交叉轴的范围取自子节点的最大交叉轴；

主轴Flex的值是由mainAxisSize属性决定的，其中MainAxisSize可以取max、min以及具体的value值；

每一个child的位置是由mainAxisAlignment以及crossAxisAlignment所决定。

Row的布局行为表面上看有这么多个步骤，其实也还算是简单，可以完全参照web中的Flex布局，包括主轴、交叉轴等概念

源码：

class MyRow extends StatelessWidget {

@override

Widget build(BuildContext context) {

// TODO: implement build

return Container(

height: 150.0,

child: Row(

children: <Widget>[

Expanded(

child: Container(

color: Colors.red,

padding: EdgeInsets.all(5.0),

),

flex: 1,

),

Expanded(

child: Container(

color: Colors.green,

padding: EdgeInsets.all(5.0),

),

flex: 2,

),

Expanded(

child: Container(

color: Colors.blue,

padding: EdgeInsets.all(5.0),

),

flex: 1,

),

],

),

);

}

}

属性解析

MainAxisAlignment：主轴方向上的对齐方式，会对child的位置起作用，默认是start。

其中MainAxisAlignment枚举值：

* center：将children放置在主轴的中心；
* end：将children放置在主轴的末尾；
* spaceAround：将主轴方向上的空白区域均分，使得children之间的空白区域相等，但是首尾child的空白区域为1/2；
* spaceBetween：将主轴方向上的空白区域均分，使得children之间的空白区域相等，首尾child都靠近首尾，没有间隙；
* spaceEvenly：将主轴方向上的空白区域均分，使得children之间的空白区域相等，包括首尾child；
* start：将children放置在主轴的起点；

其中spaceAround、spaceBetween以及spaceEvenly的区别，就是对待首尾child的方式。其距离首尾的距离分别是空白区域的1/2、0、1。

MainAxisSize：在主轴方向占有空间的值，默认是max。

MainAxisSize的取值有两种：

* max：根据传入的布局约束条件，最大化主轴方向的可用空间；
* min：与max相反，是最小化主轴方向的可用空间；

CrossAxisAlignment：children在交叉轴方向的对齐方式，与MainAxisAlignment略有不同。

CrossAxisAlignment枚举值有如下几种：

* baseline：在交叉轴方向，使得children的baseline对齐；
* center：children在交叉轴上居中展示；
* end：children在交叉轴上末尾展示；
* start：children在交叉轴上起点处展示；
* stretch：让children填满交叉轴方向；
* TextDirection：阿拉伯语系的兼容设置，一般无需处理。

VerticalDirection：定义了children摆放顺序，默认是down。

VerticalDirection枚举值有两种：

down：从top到bottom进行布局；

up：从bottom到top进行布局。

top对应Row以及Column的话，就是左边和顶部，bottom的话，则是右边和底部。

TextBaseline：使用的TextBaseline的方式，有两种，前面已经介绍过。

使用场景

Row和Column都是非常常用的布局控件。一般情况下，比方说需要将控件在一行或者一列显示的时候，都可以使用。但并不是说只能使用Row或者Column去布局，也可以使用Stack，看具体的场景选择。

#### 2 Column

在讲解Row的时候，其实是按照Flex的一些布局行为来进行的，包括源码分析，也都是在用Flex进行分析的。Row和Column都是Flex的子类，只是direction参数不同。Column各方面同Row，因此在这里不再另行讲解。

在讲解Flex的时候，也说过是参照了web的Flex布局，如果有相关开发经验的同学，完全可以参照着去理解，这样子更容易去理解它们的用法和原理。

### 10 Stack、IndexedStack、GridView

#### 1 Stack

简介

Stack可以类比web中的absolute，绝对布局。绝对布局一般在移动端开发中用的较少，但是在某些场景下，还是有其作用。当然，能用Stack绝对布局完成的，用其他控件组合也都能实现。

布局行为

Stack的布局行为，根据child是positioned还是non-positioned来区分。

对于positioned的子节点，它们的位置会根据所设置的top、bottom、right以及left属性来确定，这几个值都是相对于Stack的左上角；

对于non-positioned的子节点，它们会根据Stack的aligment来设置位置

对于绘制child的顺序，则是第一个child被绘制在最底端，后面的依次在前一个child的上面，类似于web中的z-index。如果想调整显示的顺序，则可以通过摆放child的顺序来进行。

示例

class MyStack extends StatelessWidget{

@override

Widget build(BuildContext context) {

// TODO: implement build

return Stack(

//对齐方式

alignment: Alignment(0.6, 0.6),

children: <Widget>[

//原型头像

CircleAvatar(

backgroundImage: AssetImage("assets/images/pic.jpg"),

//半径

radius: 100.0,

),

Container(

decoration: BoxDecoration(

color: Colors.blue,

),

child: Text(

"Mia B",

style: TextStyle(

fontSize: 20.0,

fontWeight: FontWeight.bold,

color: Colors.white,

),

),

),

],

);

}

}

属性解析

alignment：对齐方式，默认是左上角（topStart）。

textDirection：文本的方向，绝大部分不需要处理。

fit：定义如何设置non-positioned节点尺寸，默认为loose。

其中StackFit有如下几种：

loose：子节点宽松的取值，可以从min到max的尺寸；

expand：子节点尽可能的占用空间，取max尺寸；

passthrough：不改变子节点的约束条件。

overflow：超过的部分是否裁剪掉（clipped）。

使用场景

Stack的场景还是比较多的，对于需要叠加显示的布局，一般都可以使用Stack。有些场景下，也可以被其他控件替代，我们应该选择开销较小的控件去实现。

#### 2 IndexedStack

简介

IndexedStack继承自Stack，它的作用是显示第index个child，其他child都是不可见的。所以IndexedStack的尺寸永远是跟最大的子节点尺寸一致。

例子

在此还是将Stack的例子稍加改造，将index设置为1，也就是显示含文本的Container的节点

class MyIndexedStack extends StatelessWidget{

@override

Widget build(BuildContext context) {

// TODO: implement build

return IndexedStack(

index: 1,

//对齐方式

alignment: Alignment(0.6, 0.6),

children: <Widget>[

//原型头像

CircleAvatar(

backgroundImage: AssetImage("assets/images/pic.jpg"),

//半径

radius: 100.0,

),

Container(

decoration: BoxDecoration(

color: Colors.blue,

),

child: Text(

"Mia B",

style: TextStyle(

fontSize: 20.0,

fontWeight: FontWeight.bold,

color: Colors.white,

),

),

),

Container(

decoration: BoxDecoration(

color: Colors.red,

),

child: Text(

"Mia A",

style: TextStyle(

fontSize: 20.0,

fontWeight: FontWeight.bold,

color: Colors.white,

),

),

),

],

);

}

}

使用场景

如果需要展示一堆控件中的一个，可以使用IndexedStack。有一定的使用场景，但是也有控件可以实现其功能，只不过操作起来可能会复杂一些。

#### 3 GridView

GridView在移动端上非常的常见，就是一个滚动的多列列表，实际的使用场景也非常的多

布局行为

GridView的布局行为不复杂，本身是尽量占满空间区域，布局行为上完全继承自ScrollView

示例

class MyGridView extends StatelessWidget{

@override

Widget build(BuildContext context) {

// TODO: implement build

return GridView.count(

crossAxisCount: 2,

children: List.generate(100, (index){

return Center(

child: Text(

"item $index",

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

),

);

}),

);

}

}

默认构造函数如下：

GridView({

Key key,

Axis scrollDirection = Axis.vertical,

bool reverse = false,

ScrollController controller,

bool primary,

ScrollPhysics physics,

bool shrinkWrap = false,

EdgeInsetsGeometry padding,

@required this.gridDelegate,

bool addAutomaticKeepAlives = true,

bool addRepaintBoundaries = true,

double cacheExtent,

List<Widget> children = const <Widget>[],

})

同时也提供了如下额外的四种构造方法，方便开发者使用。

GridView.builder

GridView.custom

GridView.count

GridView.extent

属性解析

scrollDirection：滚动的方向，有垂直和水平两种，默认为垂直方向（Axis.vertical）。

reverse：默认是从上或者左向下或者右滚动的，这个属性控制是否反向，默认值为false，不反向滚动。

controller：控制child滚动时候的位置。

primary：是否是与父节点的PrimaryScrollController所关联的主滚动视图。

physics：滚动的视图如何响应用户的输入。

shrinkWrap：滚动方向的滚动视图内容是否应该由正在查看的内容所决定。

padding：四周的空白区域。

gridDelegate：控制GridView中子节点布局的delegate。

cacheExtent：缓存区域。

使用场景

使用场景很多，非常常见的控件。也有控件可以实现其功能，例如官方说的，GridView实际上是一个silvers只包含一个SilverGrid的CustomScrollView

### 11 Flow、Table、Wrap

### 12 ListBody、ListView、CustomMultiChildLayout