目录

[flutter生命周期： 7](#_Toc40282407)

[initState： 7](#_Toc40282408)

[didChangeDependencies(): 8](#_Toc40282409)

[build(): 8](#_Toc40282410)

[reassemble(): 8](#_Toc40282411)

[didChangeDependencies(): 9](#_Toc40282412)

[deactivate(): 9](#_Toc40282413)

[dispose(): 9](#_Toc40282414)

[基础组件 9](#_Toc40282415)

[Material组件 10](#_Toc40282416)

[Cupertino组件 10](#_Toc40282417)

[文本及样式 10](#_Toc40282418)

[Text : 用于显示简单样式文本，它包含一些控制文本显示样式的一些属性： 10](#_Toc40282419)

[TextStyle：用于指定文本显示的样式如颜色、字体、粗细、背景等。 11](#_Toc40282420)

[TextSpan：Text的所有文本内容只能按同一种样式，如果我们需要对一个Text内容的不同部分按照不同的样式显示，这时就可以使用TextSpan，它代表文本的一个“片段” 12](#_Toc40282421)

[图文混排：通过 Text.rich 接入 TextSpan 和 WidgetSpan 14](#_Toc40282422)

[DefaultTextStyle：在Widget树中，文本的样式默认是可以被继承的（子类文本类组件未指定具体样式时可以使用Widget树中父级设置的默认样式），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，而DefaultTextStyle正是用于设置默认文本样式的。 16](#_Toc40282423)

[字体：可以在Flutter应用程序中使用不同的字体。例如，我们可能会使用设计人员创建的自定义字体，或者其它第三方的字体，如Google Fonts中的字体。本节将介绍如何为Flutter应用配置字体，并在渲染文本时使用它们。 17](#_Toc40282424)

[按钮 19](#_Toc40282425)

[Material组件库中的按钮 19](#_Toc40282426)

[RaisedButton 19](#_Toc40282427)

[FlatButton 20](#_Toc40282428)

[OutlineButton 20](#_Toc40282429)

[IconButton 21](#_Toc40282430)

[带图标的按钮 21](#_Toc40282431)

[3.4.2 自定义按钮外观 22](#_Toc40282432)

[图片 24](#_Toc40282433)

[从asset中加载图片 24](#_Toc40282434)

[从网络加载图片 24](#_Toc40282435)

[ICON 27](#_Toc40282436)

[使用Material Design字体图标 28](#_Toc40282437)

[使用自定义字体图标 29](#_Toc40282438)

[单选开关和复选框 30](#_Toc40282439)

[CheckboxListTile 31](#_Toc40282440)

[输入框及表单 32](#_Toc40282441)

[TextField 32](#_Toc40282442)

[表单Form 43](#_Toc40282443)

[进度指示器 48](#_Toc40282444)

[LinearProgressIndicator：是一个线性、条状的进度条，定义如下： 48](#_Toc40282445)

[CircularProgressIndicator是一个圆形进度条，定义如下： 49](#_Toc40282446)

[布局类组件 49](#_Toc40282447)

[线性布局Row、Column 49](#_Toc40282448)

[Row 50](#_Toc40282449)

[Column 51](#_Toc40282450)

[弹性布局Flex 51](#_Toc40282451)

[流式布局Wrap、Flow 53](#_Toc40282452)

[Wrap 53](#_Toc40282453)

[Flow 54](#_Toc40282454)

[层叠布局Stack、Positioned 55](#_Toc40282455)

[Stack 55](#_Toc40282456)

[Position 56](#_Toc40282457)

[对齐与相对定位 Align 57](#_Toc40282458)

[Alignment 58](#_Toc40282459)

[FractionalOffset 58](#_Toc40282460)

[Center组件 59](#_Toc40282461)

[容器类组件 59](#_Toc40282462)

[填充（Padding） 59](#_Toc40282463)

[尺寸限制类容器（ConstrainedBox等） 60](#_Toc40282464)

[ConstrainedBox 60](#_Toc40282465)

[SizedBox 61](#_Toc40282466)

[重点说这么一件事： 61](#_Toc40282467)

[其他： 62](#_Toc40282468)

[装饰容器DecoratedBox 62](#_Toc40282469)

[BoxDecoration 63](#_Toc40282470)

[变换（Transform） 63](#_Toc40282471)

[Transform实操性比较强的部分，直接撸代码： 63](#_Toc40282472)

[RotatedBox 65](#_Toc40282473)

[Container 66](#_Toc40282474)

[Scaffold、TabBar、底部导航 68](#_Toc40282475)

[剪裁（Clip） 70](#_Toc40282476)

[CustomClipper 71](#_Toc40282477)

[可滚动组件 72](#_Toc40282478)

[Scrollbar 74](#_Toc40282479)

[CupertinoScrollbar 74](#_Toc40282480)

[ViewPort视口 74](#_Toc40282481)

[基于Sliver的延迟构建 74](#_Toc40282482)

[主轴和纵轴 75](#_Toc40282483)

[SingleChildScrollView 75](#_Toc40282484)

[ListView（最常用之一） 77](#_Toc40282485)

[ListView.builder 79](#_Toc40282486)

[ListView.separated 80](#_Toc40282487)

[GridView 81](#_Toc40282488)

[SliverGridDelegateWithFixedCrossAxisCount 82](#_Toc40282489)

[SliverGridDelegateWithMaxCrossAxisExtent 83](#_Toc40282490)

[CustomScrollView 85](#_Toc40282491)

[滚动监听及控制ScrollController 88](#_Toc40282492)

[滚动监听 88](#_Toc40282493)

[功能型组件： 92](#_Toc40282494)

[导航返回拦截（WillPopScope） 92](#_Toc40282495)

[举个例子： 92](#_Toc40282496)

[数据共享（InheritedWidget） 93](#_Toc40282497)

[didChangeDependencies 93](#_Toc40282498)

[深入了解InHeritedWidget 97](#_Toc40282499)

[跨组件状态共享（Provider） 99](#_Toc40282500)

[Provider的原理图： 99](#_Toc40282501)

[Provider的使用： 99](#_Toc40282502)

[颜色和主题 101](#_Toc40282503)

[颜色： 101](#_Toc40282504)

[Theme 105](#_Toc40282505)

[异步UI更新(FutureBuilder、StreamBuilder) 106](#_Toc40282506)

[FutureBuilder 106](#_Toc40282507)

[StreamBuilder 109](#_Toc40282508)

[对话框详解 110](#_Toc40282509)

[AlertDialog 110](#_Toc40282510)

[SimpleDialog 112](#_Toc40282511)

[Dialog 113](#_Toc40282512)

[对话框打开动画及遮罩 114](#_Toc40282513)

[底部菜单列表 116](#_Toc40282514)

[Loading框 118](#_Toc40282515)

[原始指针事件处理： 119](#_Toc40282516)

[忽略PointerEvent 122](#_Toc40282517)

[手势识别 122](#_Toc40282518)

[点击： 123](#_Toc40282519)

[双击： 123](#_Toc40282520)

[长按： 123](#_Toc40282521)

[垂直拖动： 123](#_Toc40282522)

[水平拖动： 123](#_Toc40282523)

[缩放： 123](#_Toc40282524)

[GestureDetector 124](#_Toc40282525)

[点击、双击、长按（直接看代码） 124](#_Toc40282526)

[拖动、滑动 125](#_Toc40282527)

[GestureRecognizer 129](#_Toc40282528)

[手势竞争与冲突 130](#_Toc40282529)

[竞争： 130](#_Toc40282530)

[竞争结果： 131](#_Toc40282531)

[冲突： 131](#_Toc40282532)

[**冲突解决方案：** 131](#_Toc40282533)

[事件总线 131](#_Toc40282534)

[问题：虽然可以监听和通知，但是当状态改变，会进行多次累计通知。是否需要再页面销毁时进行销毁？如何实现？ 133](#_Toc40282535)

[通知（Notification） 133](#_Toc40282536)

[自定义通知： 134](#_Toc40282537)

[阻止冒泡 136](#_Toc40282538)

[通知冒泡原理 137](#_Toc40282539)

[动画 138](#_Toc40282540)

[Flutter动画介绍 138](#_Toc40282541)

[Animation 138](#_Toc40282542)

[Curve 139](#_Toc40282543)

[AnimationController 140](#_Toc40282544)

[Tween 142](#_Toc40282545)

[动画结构 143](#_Toc40282546)

[实现一个AnimatedWidget和AnimatedBuilder对比的弹出动画： 143](#_Toc40282547)

[动画状态监听 146](#_Toc40282548)

[关于ios端flutter支付宝支付 147](#_Toc40282549)

[关于微信支付问题： 147](#_Toc40282550)

[关于flutterSDK更新 148](#_Toc40282551)

[疑问： 148](#_Toc40282552)

# Flutter简单介绍

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。

Flutter既不使用WebView，也不使用操作系统的原生控件,Flutter使用自己的高性能渲染引擎来绘制widget.

## Flutter高性能主要靠两点来保证:

1、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。

2、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。

# 图1-1Flutter框架结构

**注解：**

**Flutter Framework**

这是一个纯 Dart实现的 SDK，它实现了一套基础库，自底向上，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，会计算出有变化的部分，然后更新UI树，最终将UI树绘制到屏幕上，这个过程类似于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。

# 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)

**注：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

* 官网：https://flutter.dev/docs/development/tools/sdk/releases
* Github：https://github.com/flutter/flutter/releases

**特别提醒：**

1、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\）。

2、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。

# Android设置

**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。

# Dart语言

## 变量声明

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。

dynamic和Object

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(包括Function和Null)，所以任何类型的数据都可以赋值给Object声明的对象.dynamic与var一样都是关键词,声明的变量可以赋值任意对象。 而dynamic与Object相同之处在于,他们声明的变量可以在后期改变赋值类型。

dynamic t;

Object x;

t = "hi world";

x = 'Hello Object';

//下面代码没有问题

t = 1000;

x = 1000;

dynamic与Object不同的是,dynamic声明的对象编译器会提供所有可能的组合, 而Object声明的对象只能使用Object的属性与方法, 否则编译器会报错。如:

dynamic a;

Object b;

main() {

a = "";

b = "";

printLengths();

}

printLengths() {

// no warning

print(a.length);

// warning:

// The getter 'length' is not defined for the class 'Object'

print(b.length);

}

变量a不会报错, 变量b编译器会报错

dynamic的这个特性与Objective-C中的id作用很像.dynamic的这个特点使得我们在使用它时需要格外注意,这很容易引入一个运行时错误.

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 String 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。

函数声明

bool isNoble(int atomicNumber) {

return \_nobleGases[atomicNumber] != null;

}

Dart函数声明如果没有显式声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);

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 ;

函数作为变量

var say = (str){

print(str);

};

say("hi world");

函数作为参数传递

void execute(var callback) {

callback();

}

execute(() => print("xxx"))

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，并放在参数列表的最后面：

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

可选的命名参数

定义函数时，使用{param1, param2, …}，放在参数列表的最后面，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。

注意，不能同时使用可选的位置参数和可选的命名参数

## 异步支持

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。

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

# 用例子总体简单了解fLutter

通过Android Studio或VS Code创建一个新的Flutter工程，命名为"first\_flutter\_app"。创建好后，就会得到一个计数器应用的Demo。

我们先运行创建的工程，效果如图所示：

该计数器示例中，每点击一次右下角带“+”号的悬浮按钮，屏幕中央的数字就会加1。

在这个示例中，主要Dart代码是在**lib/main.dart**文件中，下面是它的源码：

import 'package:flutter/material.dart';  
  
void main() => runApp(new MyApp());  
  
class MyApp extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return new MaterialApp(  
 title: 'Flutter Demo',  
 theme: new ThemeData(  
 primarySwatch: Colors.*blue*,  
 ),  
 home: new MyHomePage(title: 'Flutter Demo Home Page'),  
 );  
 }  
}  
  
class MyHomePage extends StatefulWidget {  
 MyHomePage({Key key, this.title}) : super(key: key);  
 final String title;  
  
 @override  
 \_MyHomePageState createState() => new \_MyHomePageState();  
}  
  
class \_MyHomePageState extends State<MyHomePage> {  
 int \_counter = 0;  
  
 void \_incrementCounter() {  
 setState(() {  
 \_counter++;  
 });  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return new Scaffold(  
 appBar: new AppBar(  
 title: new Text(widget.title),  
 ),  
 body: new Center(  
 child: new Column(  
 mainAxisAlignment: MainAxisAlignment.center,  
 children: <Widget>[  
 new Text(  
 'You have pushed the button this many times:',  
 ),  
 new Text(  
 '$\_counter',  
 style: Theme.*of*(context).textTheme.headline4,  
 ),  
 ],  
 ),  
 ),  
 floatingActionButton: new FloatingActionButton(  
 onPressed: \_incrementCounter,  
 tooltip: 'Increment',  
 child: new Icon(Icons.*add*),  
 ), // This trailing comma makes auto-formatting nicer for build methods.  
 );  
 }  
}

## 分析：

1.导入包

import 'package:flutter/material.dart';

此行代码作用是导入了Material UI组件库。[Material](https://material.io/guidelines/)是一种标准的移动端和web端的视觉设计语言， Flutter默认提供了一套丰富的Material风格的UI组件。

2.应用入口

void main() => runApp(MyApp());

* 与C/C++、Java类似，Flutter 应用中main函数为应用程序的入口。main函数中调用了runApp 方法，它的功能是启动Flutter应用。runApp它接受一个Widget参数，在本示例中它是一个MyApp对象，MyApp()是Flutter应用的根组件。
* main函数使用了(=>)符号，这是Dart中单行函数或方法的简写。

3.应用结构

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return new MaterialApp(

//应用名称

title: 'Flutter Demo',

theme: new ThemeData(

//蓝色主题

primarySwatch: Colors.blue,

),

//应用首页路由

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

);

}

}

* MyApp类代表Flutter应用，它继承了StatelessWidget类，这也就意味着应用本身也是一个widget。
* 在Flutter中，大多数东西都是widget（后同“组件”或“部件”），包括对齐(alignment)、填充(padding)和布局(layout)等，它们都是以widget的形式提供。
* Flutter在构建页面时，会调用组件的build方法，widget的主要工作是提供一个build()方法来描述如何构建UI界面（通常是通过组合、拼装其它基础widget）。
* MaterialApp是Material库中提供的Flutter APP框架，通过它可以设置应用的名称、主题、语言、首页及路由列表等。MaterialApp也是一个widget。
* home为Flutter应用的首页，它也是一个widget。

## 首页

class MyHomePage extends StatefulWidget {

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

final String title;

@override

\_MyHomePageState createState() => new \_MyHomePageState();

}

class \_MyHomePageState extends State<MyHomePage> {

...

}

MyHomePage是Flutter应用的首页，它继承自StatefulWidget类，表示它是一个有状态的组件（Stateful widget）。关于Stateful widget我们将在第三章“Widget简介”一节仔细介绍，现在我们只需简单认为有状态的组件（Stateful widget） 和无状态的组件（Stateless widget）有两点不同：

* Stateful widget可以拥有状态，这些状态在widget生命周期中是可以变的，而Stateless widget是不可变的。
* Stateful widget至少由两个类组成：
  + 一个StatefulWidget类。
  + 一个State类；StatefulWidget类本身是不变的，但是State类中持有的状态在widget生命周期中可能会发生变化。

\_MyHomePageState类是MyHomePage类对应的状态类。MyHomePage类中并没有build方法，取而代之的是，build方法被挪到了\_MyHomePageState方法中.

### State类

\_MyHomePageState中都包含哪些东西：

1.该组件的状态。由于我们只需要维护一个点击次数计数器，所以定义一个\_counter状态：

int \_counter = 0; //用于记录按钮点击的总次数

\_counter为保存屏幕右下角带“+”号按钮点击次数的状态。

2. 设置状态的自增函数。

void \_incrementCounter() {

setState(() {

\_counter++;

});

}

当按钮点击时，会调用此函数，该函数的作用是先自增\_counter，然后调用setState 方法。setState方法的作用是通知Flutter框架，有状态发生了改变，Flutter框架收到通知后，会执行build方法来根据新的状态重新构建界面， Flutter 对此方法做了优化，使重新执行变的很快，所以你可以重新构建任何需要更新的东西，而无需分别去修改各个widget。

3. 构建UI界面

构建UI界面的逻辑在build方法中，当MyHomePage第一次创建时，\_MyHomePageState类会被创建，当初始化完成后，Flutter框架会调用Widget的build方法来构建widget树，最终将widget树渲染到设备屏幕上。所以，我们看看\_MyHomePageState的build方法中都干了什么事：

Widget build(BuildContext context) {

return new Scaffold(

appBar: new AppBar(

title: new Text(widget.title),

),

body: new Center(

child: new Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

new Text(

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

),

new Text(

'$\_counter',

style: Theme.of(context).textTheme.headline4,

),

],

),

),

floatingActionButton: new FloatingActionButton(

onPressed: \_incrementCounter,

tooltip: 'Increment',

child: new Icon(Icons.add),

),

);

}

* Scaffold是 Material 库中提供的页面脚手架，它提供了默认的导航栏、标题和包含主屏幕widget树（后同“组件树”或“部件树”）的body属性，组件树可以很复杂。后面示例中，路由默认都是通过Scaffold创建。
* body的组件树中包含了一个Center组件，Center可以将其子组件树对齐到屏幕中心。此例中，Center子组件是一个Column组件，Column的作用是将其所有子组件沿屏幕垂直方向依次排列； 此例中Column子组件是两个Text，第一个Text显示固定文本 “You have pushed the button this many times:”，第二个Text显示\_counter状态的数值。
* floatingActionButton是页面右下角的带“+”的悬浮按钮，它的onPressed属性接受一个回调函数，代表它被点击后的处理器，本例中直接将\_incrementCounter方法作为其处理函数。

现在，我们将整个计数器执行流程串起来：当右下角的floatingActionButton按钮被点击之后，会调用\_incrementCounter方法。在\_incrementCounter方法中，首先会自增\_counter计数器（状态），然后setState会通知Flutter框架状态发生变化，接着，Flutter框架会调用build方法以新的状态重新构建UI，最终显示在设备屏幕上。

## 为什么要将build方法放在State中，而不是放在StatefulWidget中？

其实主要就是为了提高开发的灵活性。如果将build()方法放在StatefulWidget中则会有两个问题：

1.状态访问不便

试想一下，如果我们的StatefulWidget有很多状态，而每次状态改变都要调用build方法，由于状态是保存在State中的，如果build方法在StatefulWidget中，那么build方法和状态分别在两个类中，那么构建时读取状态将会很不方便！试想一下，如果真的将build方法放在StatefulWidget中的话，由于构建用户界面过程需要依赖State，所以build方法将必须加一个State参数，大概是下面这样：

Widget build(BuildContext context, State state){

//state.counter

...

}

这样的话就只能将State的所有状态声明为公开的状态，这样才能在State类外部访问状态！但是，将状态设置为公开后，状态将不再具有私密性，这就会导致对状态的修改将会变的不可控。但如果将build()方法放在State中的话，构建过程不仅可以直接访问状态，而且也无需公开私有状态，这会非常方便。

2.继承StatefulWidget不便

例如，Flutter中有一个动画widget的基类AnimatedWidget，它继承自StatefulWidget类。AnimatedWidget中引入了一个抽象方法build(BuildContext context)，继承自AnimatedWidget的动画widget都要实现这个build方法。现在设想一下，如果StatefulWidget类中已经有了一个build方法，正如上面所述，此时build方法需要接收一个state对象，这就意味着AnimatedWidget必须将自己的State对象(记为\_animatedWidgetState)提供给其子类，因为子类需要在其build方法中调用父类的build方法，代码可能如下：

class MyAnimationWidget extends AnimatedWidget{

@override

Widget build(BuildContext context, State state){

//由于子类要用到AnimatedWidget的状态对象\_animatedWidgetState，

//所以AnimatedWidget必须通过某种方式将其状态对象\_animatedWidgetState

//暴露给其子类

super.build(context, \_animatedWidgetState)

}

}

这样很显然是不合理的，因为：

·····1. AnimatedWidget的状态对象是AnimatedWidget内部实现细节，不应该暴露给外部。

·····2. 如果要将父类状态暴露给子类，那么必须得有一种传递机制，而做这一套传递机制是无意义的，因为父子类之间状态的传递和子类本身逻辑是无关的。

# 路由管理：

**作用：**管理页面之间如何跳转，通常也可被称为导航管理。Flutter中的路由管理和原生开发类似，无论是Android还是iOS，导航管理都会维护一个路由栈，路由入栈(push)操作对应打开一个新页面，路由出栈(pop)操作对应页面关闭操作，而路由管理主要是指如何来管理路由栈。

1.一个简单的例子：（对上面的默认demo做些修改）

①创建一个新路由，命名为“NewRouter”

class NewRoute extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text("New route"),

),

body: Center(

child: Text("This is new route"),

),

);

}

}

//新路由继承自StatelessWidget，界面很简单，在页面中间显示一句"This is new route"。

②在\_MyHomePageState.build方法中的Column的子widget中添加一个按钮（FlatButton）:

Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

... //省略无关代码

FlatButton(

child: Text("open new route"),

textColor: Colors.blue,

onPressed: () {

//导航到新路由

Navigator.push( context,

MaterialPageRoute(builder: (context) {

return NewRoute();

}));

},

),

],

)

2. MaterialPageRoute

MaterialPageRoute继承自PageRoute类，PageRoute类是一个抽象类，表示占有整个屏幕空间的一个模态路由页面，它还定义了路由构建及切换时过渡动画的相关接口及属性。MaterialPageRoute是Material组件库提供的组件，它可以针对不同平台，实现与平台页面切换动画风格一致的路由切换动画：

* 对于Android，当打开新页面时，新的页面会从屏幕底部滑动到屏幕顶部；当关闭页面时，当前页面会从屏幕顶部滑动到屏幕底部后消失，同时上一个页面会显示到屏幕上。
* 对于iOS，当打开页面时，新的页面会从屏幕右侧边缘一致滑动到屏幕左边，直到新页面全部显示到屏幕上，而上一个页面则会从当前屏幕滑动到屏幕左侧而消失；当关闭页面时，正好相反，当前页面会从屏幕右侧滑出，同时上一个页面会从屏幕左侧滑入。

**MaterialPageRoute构造函数的各个参数的意义：**

MaterialPageRoute({

WidgetBuilder builder,

RouteSettings settings,

bool maintainState = true,

bool fullscreenDialog = false,

})

* builder是一个WidgetBuilder类型的回调函数，它的作用是构建路由页面的具体内容，返回值是一个widget。我们通常要实现此回调，返回新路由的实例。
* settings包含路由的配置信息，如路由名称、是否初始路由（首页）。
* maintainState：默认情况下，当入栈一个新路由时，原来的路由仍然会被保存在内存中，如果想在路由没用的时候释放其所占用的所有资源，可以设置maintainState为false。
* fullscreenDialog表示新的路由页面是否是一个全屏的模态对话框，在iOS中，如果fullscreenDialog为true，新页面将会从屏幕底部滑入（而不是水平方向）。

3.Navigator

Navigator是一个路由管理的组件，它提供了打开和退出路由页方法。Navigator通过一个栈来管理活动路由集合。通常当前屏幕显示的页面就是栈顶的路由。Navigator提供了一系列方法来管理路由栈，在此我们只介绍其最常用的两个方法：

Future push(BuildContext context, Route route)

将给定的路由入栈（即打开新的页面），返回值是一个Future对象，用以接收新路由出栈（即关闭）时的返回数据。

bool pop(BuildContext context, [ result ])

将栈顶路由出栈，result为页面关闭时返回给上一个页面的数据。

Navigator还有很多其它方法，如Navigator.replace、Navigator.popUntil等，详情请参考API文档或SDK源码注释，下面我们还需要介绍一下路由相关的另一个概念“命名路由”。

实例方法

Navigator类中第一个参数为context的**静态方法**都对应一个Navigator的**实例方法**， 比如Navigator.push(BuildContext context, Route route)等价于Navigator.of(context).push(Route route)，下面命名路由相关的方法也是一样的。

路由传值

很多时候，在路由跳转时我们需要带一些参数，比如打开商品详情页时，我们需要带一个商品id，这样商品详情页才知道展示哪个商品信息；又比如我们在填写订单时需要选择收货地址，打开地址选择页并选择地址后，可以将用户选择的地址返回到订单页等等。下面我们通过一个简单的示例来演示新旧路由如何传参。

示例：非命名路由的传值方式：

我们创建一个TipRoute路由，它接受一个提示文本参数，负责将传入它的文本显示在页面上，另外TipRoute中我们添加一个“返回”按钮，点击后在返回上一个路由的同时会带上一个返回参数，下面我们看一下实现代码。

class TipRoute extends StatelessWidget {

TipRoute({

Key key,

@required this.text, // 接收一个text参数

}) : super(key: key);

final String text;

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text("提示"),

),

body: Padding(

padding: EdgeInsets.all(18),

child: Center(

child: Column(

children: <Widget>[

Text(text),

RaisedButton(

onPressed: () => Navigator.pop(context, "我是返回值"),

child: Text("返回"),

)

],

),

),

),

);

}

}

下面是打开新路由TipRoute的代码：

class RouterTestRoute extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Center(

child: RaisedButton(

onPressed: () async {

// 打开`TipRoute`，并等待返回结果

var result = await Navigator.push(

context,

MaterialPageRoute(

builder: (context) {

return TipRoute(

// 路由参数

text: "我是提示xxxx",

);

},

),

);

//输出`TipRoute`路由返回结果

print("路由返回值: $result");

},

child: Text("打开提示页"),

),

);

}

}

****运行上面代码，点击RouterTestRoute页的“打开提示页”按钮，会打开TipRoute页，运行效果如图所示下：

**需要说明：**

1.提示文案“我是提示xxxx”是通过TipRoute的text参数传递给新路由页的。我们可以通过等待Navigator.push(…)返回的Future来获取新路由的返回数据。

2.在TipRoute页中有两种方式可以返回到上一页；第一种方式时直接点击导航栏返回箭头，第二种方式是点击页面中的“返回”按钮。这两种返回方式的区别是前者不会返回数据给上一个路由，而后者会。下面是分别点击页面中的返回按钮和导航栏返回箭头后，RouterTestRoute页中print方法在控制台输出的内容：

I/flutter (27896): 路由返回值: 我是返回值

I/flutter (27896): 路由返回值: null

命名路由：

所谓“命名路由”（Named Route）即有名字的路由，我们可以先给路由起一个名字，然后就可以通过路由名字直接打开新的路由了，这为路由管理带来了一种直观、简单的方式。

**路由表：**

要想使用命名路由，我们必须先提供并注册一个路由表（routing table），这样应用程序才知道哪个名字与哪个路由组件相对应。其实注册路由表就是给路由起名字，路由表的定义如下：

Map<String, WidgetBuilder> routes;

它是一个Map，key为路由的名字，是个字符串；value是个builder回调函数，用于生成相应的路由widget。我们在通过路由名字打开新路由时，应用会根据路由名字在路由表中查找到对应的WidgetBuilder回调函数，然后调用该回调函数生成路由widget并返回。

注册路由表：

路由表的注册方式很简单，我们回到之前“计数器”的示例，然后在MyApp类的build方法中找到MaterialApp，添加routes属性，代码如下：

MaterialApp(

title: 'Flutter Demo',

theme: ThemeData(

primarySwatch: Colors.blue,

),

//注册路由表

routes:{

"new\_page":(context) => NewRoute(),

... // 省略其它路由注册信息

} ,

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

);

现在我们就完成了路由表的注册。上面的代码中home路由并没有使用命名路由，如果我们也想将home注册为命名路由应该怎么做呢？其实很简单，直接看代码：

MaterialApp(

title: 'Flutter Demo',

initialRoute:"/", //名为"/"的路由作为应用的home(首页)

theme: ThemeData(

primarySwatch: Colors.blue,

),

//注册路由表

routes:{

"new\_page":(context) => NewRoute(),

"/":(context) => MyHomePage(title: 'Flutter Demo Home Page'), //注册首页路由

}

);

可以看到，我们只需在路由表中注册一下MyHomePage路由，然后将其名字作为MaterialApp的initialRoute属性值即可，该属性决定应用的初始路由页是哪一个命名路由。

通过路由名打开新路由页

要通过路由名称来打开新路由，可以使用Navigator 的pushNamed方法：

Future pushNamed(BuildContext context, String routeName,{Object arguments})

通过路由名来打开新的路由页，修改FlatButton的onPressed回调代码，改为：

onPressed: () {

Navigator.pushNamed(context, "new\_page");

//Navigator.push(context,

// MaterialPageRoute(builder: (context) {

// return NewRoute();

//}));

},

热重载应用，再次点击“open new route”按钮，依然可以打开新的路由页。

命名路由参数传递

我们先注册一个路由：

routes:{

"new\_page":(context) => EchoRoute(),

} ,

在路由页通过RouteSetting对象获取路由参数：

class EchoRoute extends StatelessWidget {

@override

Widget build(BuildContext context) {

//获取路由参数

var args=ModalRoute.of(context).settings.arguments;

//...省略无关代码

}

}

在打开路由时传递参数

Navigator.of(context).pushNamed("new\_page", arguments: "hi");

适配：

假设我们也想将上面路由传参示例中的TipRoute路由页注册到路由表中，以便也可以通过路由名来打开它。但是，由于TipRoute接受一个text参数，我们如何在不改变TipRoute源码的前提下适配这种情况？其实很简单：

MaterialApp(

... //省略无关代码

routes: {

"tip2": (context){

return TipRoute(text: ModalRoute.of(context).settings.arguments);

},

},

);

路由生成钩子

假设我们要开发一个电商APP，当用户没有登录时可以看店铺、商品等信息，但交易记录、购物车、用户个人信息等页面需要登录后才能看。为了实现上述功能，我们需要在打开每一个路由页前判断用户登录状态！如果每次打开路由前我们都需要去判断一下将会非常麻烦，那有什么更好的办法吗？答案是有！

MaterialApp有一个onGenerateRoute属性，它在打开命名路由时可能会被调用，之所以说可能，是因为当调用Navigator.pushNamed(...)打开命名路由时，如果指定的路由名在路由表中已注册，则会调用路由表中的builder函数来生成路由组件；如果路由表中没有注册，才会调用onGenerateRoute来生成路由。onGenerateRoute回调签名如下：

Route<dynamic> Function(RouteSettings settings)

有了onGenerateRoute回调，要实现上面控制页面权限的功能就非常容易：我们放弃使用路由表，取而代之的是提供一个onGenerateRoute回调，然后在该回调中进行统一的权限控制，如：

MaterialApp(

... //省略无关代码

onGenerateRoute:(RouteSettings settings){

return MaterialPageRoute(builder: (context){

String routeName = settings.name;

// 如果访问的路由页需要登录，但当前未登录，则直接返回登录页路由，

// 引导用户登录；其它情况则正常打开路由。

}

);

}

);

注意，onGenerateRoute只会对命名路由生效。

**写在后面的话：**

路由MaterialApp中还有navigatorObservers和onUnknownRoute两个回调属性，前者可以监听所有路由跳转动作，后者在打开一个不存在的命名路由时会被调用。

可自行参考API。

# 包管理

flutter使用配置文件pubspec.yaml（位于项目根目录）来管理第三方依赖包。

Flutter项目默认的配置文件是pubspec.yaml，我们看一个简单的示例：

name: flutter\_in\_action

description: First Flutter application.

version: 1.0.0+1

dependencies:

flutter:

sdk: flutter

cupertino\_icons: ^0.1.2

dev\_dependencies:

flutter\_test:

sdk: flutter

flutter:

uses-material-design: **true**

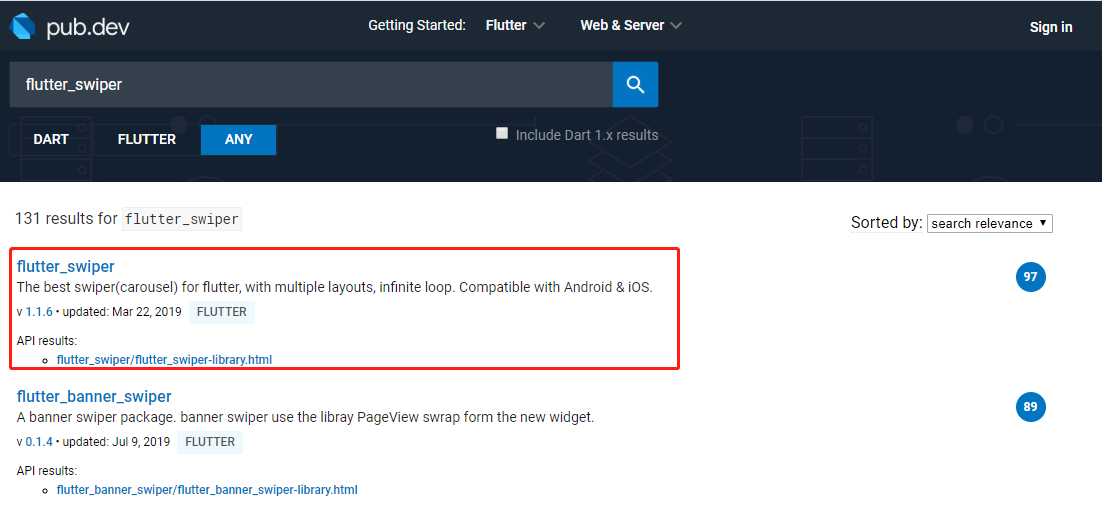
* name：应用或包名称。
* description: 应用或包的描述、简介。
* version：应用或包的版本号。
* dependencies：应用或包依赖的其它包或插件。（将所依赖的包添加到这里）
* dev\_dependencies：开发环境依赖的工具包（而不是flutter应用本身依赖的包）。
* flutter：flutter相关的配置选项。

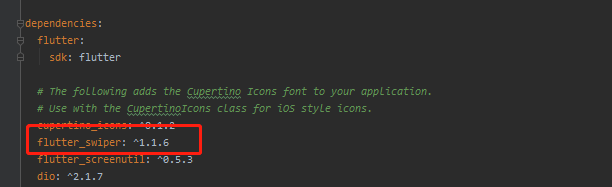
Pub仓库

Pub（<https://pub.dev/>）是Google官方的Dart Packages仓库，类似于node中的npm仓库，android中的jcenter。我们可以在Pub上面查找我们需要的包和插件，也可以向Pub发布我们的包和插件。

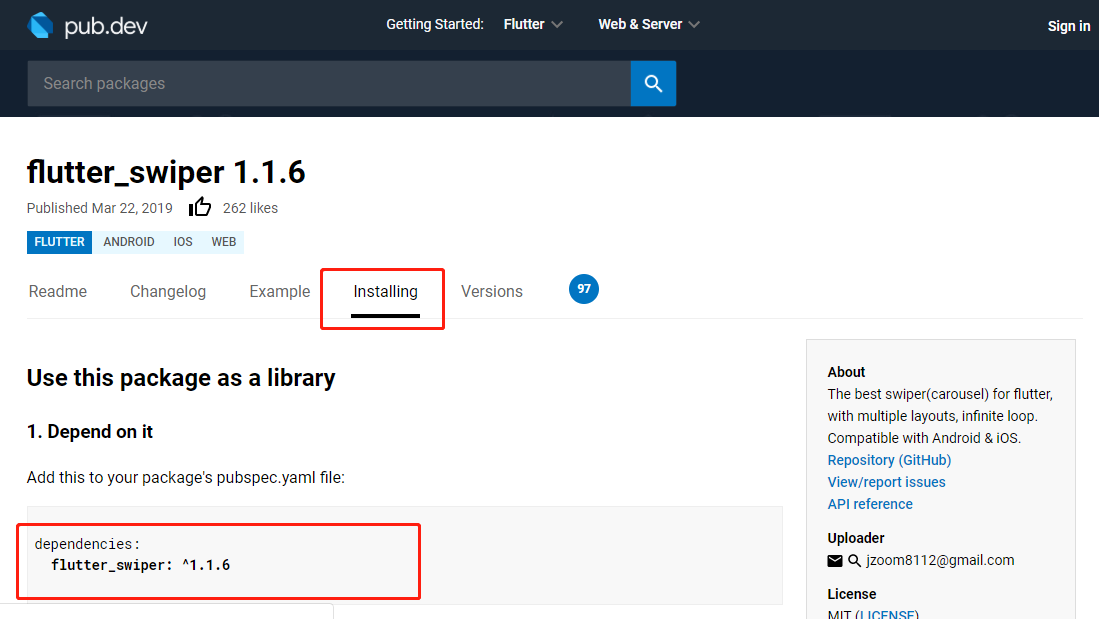
用法：

1.搜索对应插件包





2.在pubspec.yaml的dependencies中配置包名



3.保存自动下载对应包或者执行flutter packages get 命令来下载依赖包。

需要注意dependencies和dev\_dependencies的区别，前者的依赖包将作为APP的源码的一部分参与编译，生成最终的安装包。而后者的依赖包只是作为开发阶段的一些工具包，主要是用于帮助我们提高开发、测试效率，比如flutter的自动化测试包等。

4. 引入下载包



**写在后面的话：其他依赖方式（依赖本地包，依赖Git包）等可查找相关资料学习**（<https://www.dartlang.org/tools/pub/dependencies>）**。**

# 资源管理

Flutter APP安装包中会包含代码和 assets（资源）两部分。Assets是会打包到程序安装包中的，可在运行时访问。常见类型的assets包括静态数据（例如JSON文件）、配置文件、图标和图片（JPEG，WebP，GIF，动画WebP / GIF，PNG，BMP和WBMP）等。

指定 assets

和包管理一样，Flutter也使用[pubspec.yaml](https://www.dartlang.org/tools/pub/pubspec)文件来管理应用程序所需的资源，举个例子:

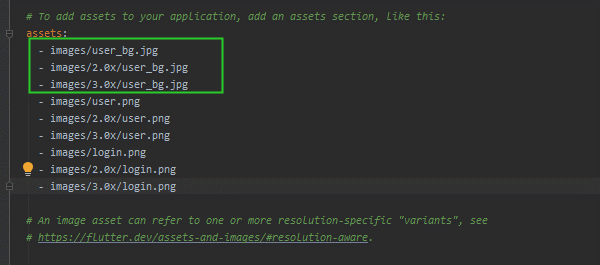
flutter:

assets:

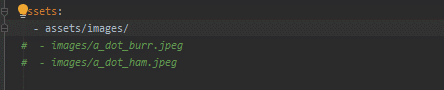
- assets/my\_icon.png

- assets/background.png

理论上讲：assets指定应包含在应用程序中的文件，每个asset都通过相对于pubspec.yaml文件所在的文件系统路径来标识自身路径，还可以设置多倍图，flutter会根据网络、手机分辨率等进行自动选择（Asset 变体（variant））。asset的声明顺序没有关系，asset的实际目录可以是任意文件夹。



也可以统一声明（亲测可以这么用，但是在官方目前没有找到此描述）：



这样代表一次性声明images目录下所有的文件。

Asset 变体（variant）

构建过程支持“asset变体”的概念：不同版本的asset可能会显示在不同的上下文中。 在pubspec.yaml的assets部分中指定asset路径时，构建过程中，会在相邻子目录中查找具有相同名称的任何文件。这些文件随后会与指定的asset一起被包含在asset bundle中。

按照上面提到的例子，那么这两个images/user\_bg.jpg和他的2倍图和3倍图都将包含在您的asset bundle中。前者被认为是main asset（主资源），后者被认为是一种变体（variant）。

加载 assets

应用可以通过[AssetBundle](https://docs.flutter.io/flutter/services/AssetBundle-class.html)对象访问其asset 。有两种主要方法允许从Asset bundle中加载字符串或图片（二进制）文件。

加载文本assets

1.通过[rootBundle](https://docs.flutter.io/flutter/services/rootBundle.html)对象加载：每个Flutter应用程序都有一个[rootBundle](https://docs.flutter.io/flutter/services/rootBundle.html)对象， 通过它可以轻松访问主资源包，直接使用package:flutter/services.dart中全局静态的rootBundle对象来加载asset即可。

2.通过[DefaultAssetBundle](https://docs.flutter.io/flutter/widgets/DefaultAssetBundle-class.html)加载：建议使用[DefaultAssetBundle](https://docs.flutter.io/flutter/widgets/DefaultAssetBundle-class.html)来获取当前BuildContext的AssetBundle。 这种方法不是使用应用程序构建的默认asset bundle，而是使父级widget在运行时动态替换的不同的AssetBundle，这对于本地化或测试场景很有用。

通常，可以使用DefaultAssetBundle.of()在应用运行时来间接加载asset（例如JSON文件），而在widget上下文之外，或其它AssetBundle句柄不可用时，可以使用rootBundle直接加载这些asset，例如：

import 'dart:async' show Future;

import 'package:flutter/services.dart' show rootBundle;

Future<String> loadAsset() async {

return await rootBundle.loadString('assets/config.json');

}

加载图片

类似于原生开发，Flutter也可以为当前设备加载适合其分辨率的图像。

##### 声明分辨率相关的图片 assets

[AssetImage](https://docs.flutter.io/flutter/painting/AssetImage-class.html)可以将asset的请求逻辑映射到最接近当前设备像素比例（dpi）的asset。为了使这种映射起作用，必须根据特定的目录结构来保存asset：

* …/image.png
* …/**M**x/image.png
* …/**N**x/image.png
* …etc.

其中M和N是数字标识符，对应于其中包含的图像的分辨率，也就是说，它们指定不同设备像素比例的图片。

主资源默认对应于1.0倍的分辨率图片。看一个例子：

* …/my\_icon.png
* …/2.0x/my\_icon.png
* …/3.0x/my\_icon.png

在设备像素比率为1.8的设备上，.../2.0x/my\_icon.png 将被选择。对于2.7的设备像素比率，.../3.0x/my\_icon.png将被选择。

如果未在Image widget上指定渲染图像的宽度和高度，那么Image widget将占用与主资源相同的屏幕空间大小。 也就是说，如果.../my\_icon.png是72px乘72px，那么.../3.0x/my\_icon.png应该是216px乘216px; 但如果未指定宽度和高度，它们都将渲染为72像素×72像素（以逻辑像素为单位）。

pubspec.yaml中asset部分中的每一项都应与实际文件相对应，但主资源项除外。当主资源缺少某个资源时，会按分辨率从低到高的顺序去选择 ，也就是说1x中没有的话会在2x中找，2x中还没有的话就在3x中找。

##### 加载图片

要加载图片，可以使用[AssetImage](https://docs.flutter.io/flutter/painting/AssetImage-class.html)类。例如，我们可以从上面的asset声明中加载背景图片：

Widget build(BuildContext context) {

return new DecoratedBox(

decoration: new BoxDecoration(

image: new DecorationImage(

image: new AssetImage('graphics/background.png'),

),

),

);

}

注意，AssetImage并非是一个widget， 它实际上是一个ImageProvider，有些时候你可能期望直接得到一个显示图片的widget，那么你可以使用Image.asset()方法，如：

Widget build(BuildContext context) {

return Image.asset('graphics/background.png');

}

##### 依赖包中的资源图片

要加载依赖包中的图像，必须给AssetImage提供package参数。

例如，假设应用程序依赖于一个名为“my\_icons”的包，它具有如下目录结构：

* …/pubspec.yaml
* …/icons/heart.png
* …/icons/1.5x/heart.png
* …/icons/2.0x/heart.png
* …etc.

然后加载图像，使用: new AssetImage('icons/heart.png', package: 'my\_icons')或者new Image.asset('icons/heart.png', package: 'my\_icons')

##### 打包包中的 assets

如果在pubspec.yaml文件中声明了期望的资源，它将会打包到相应的package中。特别是，包本身使用的资源必须在pubspec.yaml中指定。

包也可以选择在其lib/文件夹中包含未在其pubspec.yaml文件中声明的资源。在这种情况下，对于要打包的图片，应用程序必须在pubspec.yaml中指定包含哪些图像。 例如，一个名为“fancy\_backgrounds”的包，可能包含以下文件：

* …/lib/backgrounds/background1.png
* …/lib/backgrounds/background2.png
* …/lib/backgrounds/background3.png

要包含第一张图像，必须在pubspec.yaml的assets部分中声明它：

flutter:

assets:

- packages/fancy\_backgrounds/backgrounds/background1.png

lib/是隐含的，所以它不应该包含在资产路径中。

特定平台 assets

上面的资源都是flutter应用中的，这些资源只有在Flutter框架运行之后才能使用，如果要给我们的应用设置APP图标或者添加启动图，那我们必须使用特定平台的assets。

##### 设置APP图标

更新Flutter应用程序启动图标的方式与在本机Android或iOS应用程序中更新启动图标的方式相同。

* Android

在Flutter项目的根目录中，导航到.../android/app/src/main/res目录，里面包含了各种资源文件夹。 只需按照Android开发人员指南（<https://developer.android.com/guide/practices/ui_guidelines/icon_design_launcher.html#size>）中的说明， 将其替换为所需的资源，并遵守每种屏幕密度（dpi）的建议图标大小标准。

**注意:**如果您重命名.png文件，则还必须在您AndroidManifest.xml的<application>标签的android:icon属性中更新名称。

* Ios

在Flutter项目的根目录中，导航到.../ios/Runner。该目录中Assets.xcassets/AppIcon.appiconset已经包含占位符图片， 只需将它们替换为适当大小的图片，保留原始文件名称。

##### 更新启动页

在Flutter框架加载时，Flutter会使用本地平台机制绘制启动页。此启动页将持续到Flutter渲染应用程序的第一帧时。

**注意:**这意味着如果不在应用程序的main()方法中调用[runApp](https://docs.flutter.io/flutter/widgets/runApp.html)函数 （或者更具体地说，如果不调用[window.render](https://docs.flutter.io/flutter/dart-ui/Window/render.html)去响应[window.onDrawFrame](https://docs.flutter.io/flutter/dart-ui/Window/onDrawFrame.html)）的话， 启动屏幕将永远持续显示。

* Android

要将启动屏幕（splash screen）添加到您的Flutter应用程序， 请导航至.../android/app/src/main。在res/drawable/launch\_background.xml，通过自定义drawable来实现自定义启动界面（你也可以直接换一张图片）。

* iOS

要将图片添加到启动屏幕（splash screen）的中心，请导航至.../ios/Runner。在Assets.xcassets/LaunchImage.imageset， 拖入图片，并命名为LaunchImage.png、LaunchImage@2x.png、LaunchImage@3x.png。 如果你使用不同的文件名，那还必须更新同一目录中的Contents.json文件，图片的具体尺寸可以查看苹果官方的标准。

也可以通过打开Xcode完全自定义storyboard。在Project Navigator中导航到Runner/Runner然后通过打开Assets.xcassets拖入图片，或者通过在LaunchScreen.storyboard中使用Interface Builder进行自定义。

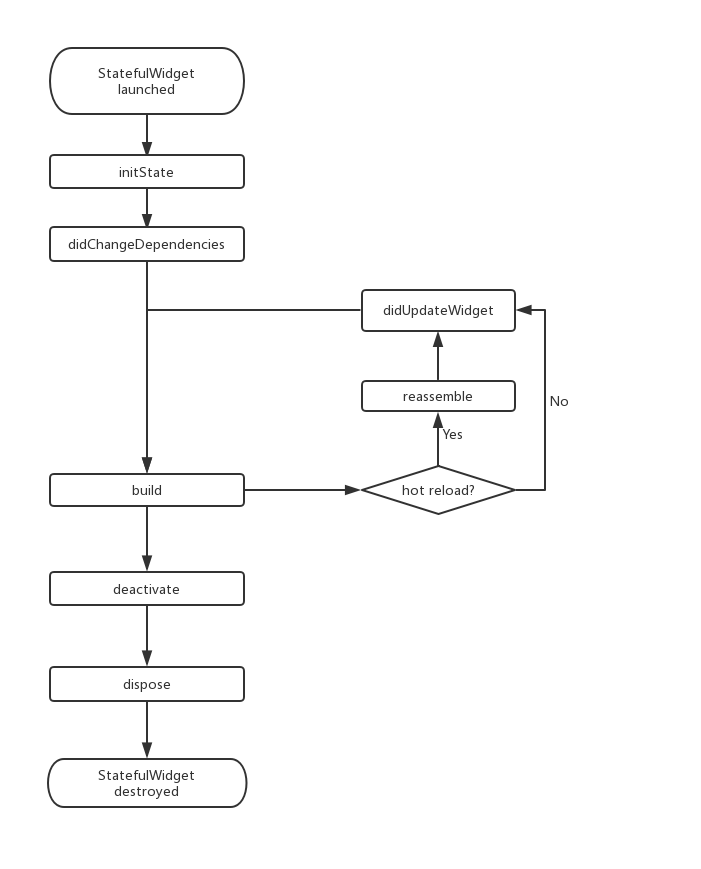
# 调试Flutter应用

debugger() 断点

print() 打印

debugPrint() 封装print，它将输出限制在一个级别，避免被Android内核丢弃。

# flutter生命周期：



## initState：

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只会调用一次该回调，所以，通常在该回调中做一些一次性的操作。如状态初始化、订阅子树的事件通知等。不能在该回调中用buildContext.dependOnInheritedWidgetOfExacType(该方法用于在Widget树上获取离当前widget最近的一个父级InheriFromWidget)，原因是在初始化完成后，Widget树中的InheriFromWidget也可能发生变化，所以正确的做法应该在build()方法或disChangeDependencies()中调用它。

## didChangeDependencies():

当State对象的依赖发生变化时会被调用；例如：在之前build()中包含了一个InheritdWidget，然后在之后的build()中InheritdWidget发生了变化，那么此时inheritdwidget的子widget的didChangeDependencies()回调都会被调用。典型的场景是当系统语言Locale或应用主题改变时，Flutter framework会通知widget调用此回调。

## build():

主要用于构建Widget子树的，会在如下场景被调用：

1.在调用initState()之后。

2.在调用didUpdateWidget()之后。

3.在调用setState()之后。

4.在调用didChangeDependencies()之后。

5.在State对象从树中一个位置移除后(会调用deactivate)有重新插入到树的其他位置之后。

## reassemble():

此回调是专门为了开发调试而提供的，在热重载时会被调用，此回调在Release模式下永远不会被调用。

## didChangeDependencies():

在widget重新构建时，Flutter framework会调用Widget.canUpdate 来检测Widget树中同一位置的新旧节点，然后决定是否需要更新，如果Widget.canUpdate返回true则会调用此回调。Widget.canUpdate会在新旧widget的key和runtimeType同时相等时会返回true，也就是说在新旧widget的key和runtimeType同时相等时didUpdateWidget()就会被调用。

## deactivate():

当State对象从树中被移除时，会调用此回调。在一些场景下，Flutter framework会将State对象重新插到树中，如包含此State对象的子树在树的一个位置移动到另一个位置时(可以通过GlobalKey来实现)。如果移除后没有重新插入到树中则紧接着会调用dispose()方法。

## dispose():

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。

# 基础组件

首先引入：import 'package:flutter/widgets.dart';

* [Text](https://docs.flutter.io/flutter/widgets/Text-class.html)：该组件可让您创建一个带格式的文本。
* [Row](https://docs.flutter.io/flutter/widgets/Row-class.html)、[Column](https://docs.flutter.io/flutter/widgets/Column-class.html)： 这些具有弹性空间的布局类Widget可让您在水平（Row）和垂直（Column）方向上创建灵活的布局。其设计是基于Web开发中的Flexbox布局模型。
* [Stack](https://docs.flutter.io/flutter/widgets/Stack-class.html)： 取代线性布局 (译者语：和Android中的FrameLayout相似)，[Stack](https://docs.flutter.io/flutter/widgets/Stack-class.html)允许子 widget 堆叠， 你可以使用[Positioned](https://docs.flutter.io/flutter/widgets/Positioned-class.html)来定位他们相对于Stack的上下左右四条边的位置。Stacks是基于Web开发中的绝对定位（absolute positioning )布局模型设计的。
* [Container](https://docs.flutter.io/flutter/widgets/Container-class.html)：[Container](https://docs.flutter.io/flutter/widgets/Container-class.html)可让您创建矩形视觉元素。container 可以装饰一个[BoxDecoration](https://docs.flutter.io/flutter/painting/BoxDecoration-class.html), 如 background、一个边框、或者一个阴影。[Container](https://docs.flutter.io/flutter/widgets/Container-class.html)也可以具有边距（margins）、填充(padding)和应用于其大小的约束(constraints)。另外，[Container](https://docs.flutter.io/flutter/widgets/Container-class.html)可以使用矩阵在三维空间中对其进行变换。

## Material组件

Flutter提供了一套丰富的Material组件，它可以帮助我们构建遵循Material Design设计规范的应用程序。Material应用程序以[MaterialApp](https://docs.flutter.io/flutter/material/MaterialApp-class.html)组件开始， 该组件在应用程序的根部创建了一些必要的组件，比如Theme组件，它用于配置应用的主题。 是否使用[MaterialApp](https://docs.flutter.io/flutter/material/MaterialApp-class.html)完全是可选的，但是使用它是一个很好的做法。在之前的示例中，我们已经使用过多个Material 组件了，如：Scaffold、AppBar、FlatButton等。要使用Material 组件，需要先引入它：

import 'package:flutter/material.dart';

## Cupertino组件

Flutter也提供了一套丰富的Cupertino风格的组件，尽管目前还没有Material 组件那么丰富，但是它仍在不断的完善中。值得一提的是在Material 组件库中有一些组件可以根据实际运行平台来切换表现风格，比如MaterialPageRoute，在路由切换时，如果是Android系统，它将会使用Android系统默认的页面切换动画(从底向上)；如果是iOS系统，它会使用iOS系统默认的页面切换动画（从右向左）。

## 文本及样式

### Text : 用于显示简单样式文本，它包含一些控制文本显示样式的一些属性：

Text("Hello world",

textAlign: TextAlign.left,

);

Text("Hello world! I'm Jack. "\*4,

maxLines: 1,

overflow: TextOverflow.ellipsis,

);

Text("Hello world",

textScaleFactor: 1.5,

);

·textAlign：文本的对齐方式；可以选择左对齐、右对齐还是居中。注意，对齐的参考系是Text widget本身。本例中虽然是指定了居中对齐，但因为Text文本内容宽度不足一行，Text的宽度和文本内容长度相等，那么这时指定对齐方式是没有意义的，只有Text宽度大于文本内容长度时指定此属性才有意义。下面我们指定一个较长的字符串。

·maxLines、overflow：指定文本显示的最大行数，默认情况下，文本是自动折行的，如果指定此参数，则文本最多不会超过指定的行。如果有多余的文本，可以通过overflow来指定截断方式，默认是直接截断，本例中指定的截断方式TextOverflow.ellipsis，它会将多余文本截断后以省略符“...”表示；TextOverflow的其它截断方式请参考SDK文档。

·textScaleFactor：代表文本相对于当前字体大小的缩放因子，相对于去设置文本的样式style属性的fontSize，它是调整字体大小的一个快捷方式。该属性的默认值可以通过MediaQueryData.textScaleFactor获得，如果没有MediaQuery，那么会默认值将为1.0。

### TextStyle：用于指定文本显示的样式如颜色、字体、粗细、背景等。

Text("Hello world",

style: TextStyle(

color: Colors.blue,

fontSize: 18.0,

height: 1.2,

fontFamily: "Courier",

background: new Paint()..color=Colors.yellow,

decoration:TextDecoration.underline,

decorationStyle: TextDecorationStyle.dashed

),

);

·height：该属性用于指定行高，但它并不是一个绝对值，而是一个因子，具体的行高等于fontSize\*height。

·fontFamily ：由于不同平台默认支持的字体集不同，所以在手动指定字体时一定要先在不同平台测试一下。

·fontSize：该属性和Text的textScaleFactor都用于控制字体大小。但是有两个主要区别：

~fontSize可以精确指定字体大小，而textScaleFactor只能通过缩放比例来控制。

~textScaleFactor主要是用于系统字体大小设置改变时对Flutter应用字体进行全局调整，而fontSize通常用于单个文本，字体大小不会跟随系统字体大小变化。

### TextSpan：Text的所有文本内容只能按同一种样式，如果我们需要对一个Text内容的不同部分按照不同的样式显示，这时就可以使用TextSpan，它代表文本的一个“片段”

const TextSpan({

TextStyle style,

Sting text,

List<TextSpan> children,

GestureRecognizer recognizer,

});

3-8其中style和text属性代表该文本片段的样式和内容。children是一个TextSpan的数组，也就是说TextSpan可以包括其他TextSpan。而recognizer用于对该文本片段上用于手势进行识别处理。效果如下图，然后用TextSpan实现它。

源码：

Text.rich(TextSpan(

children: [

TextSpan(

text: "Home: "

),

TextSpan(

text: "https://flutterchina.club",

style: TextStyle(

color: Colors.blue

),

recognizer: \_tapRecognizer

),

]

))

·通过TextSpan实现了一个基础文本片段和一个链接片段，然后通过Text.rich方法将TextSpan添加到Text中，之所以可以这样做，是因为Text其实就是RichText的一个包装，而RichText是可以显示多种样式(富文本)的widget。

·\_tapRecognizer，它是点击链接后的一个处理器（代码已省略）

### 图文混排：通过 Text.rich 接入 TextSpan 和 WidgetSpan

Text.rich(TextSpan(

children: <InlineSpan>[

TextSpan(text: 'Flutter is'),

WidgetSpan(

child: SizedBox(

width: 120,

height: 50,

child: Card(

color: Colors.blue,

child: Center(child: Text('Hello World!'))),

)),

WidgetSpan(

child: SizedBox(

width: size > 0 ? size : 0,

height: size > 0 ? size : 0,

child: new Image.asset(

"static/gsy\_cat.png",

fit: BoxFit.cover,

),

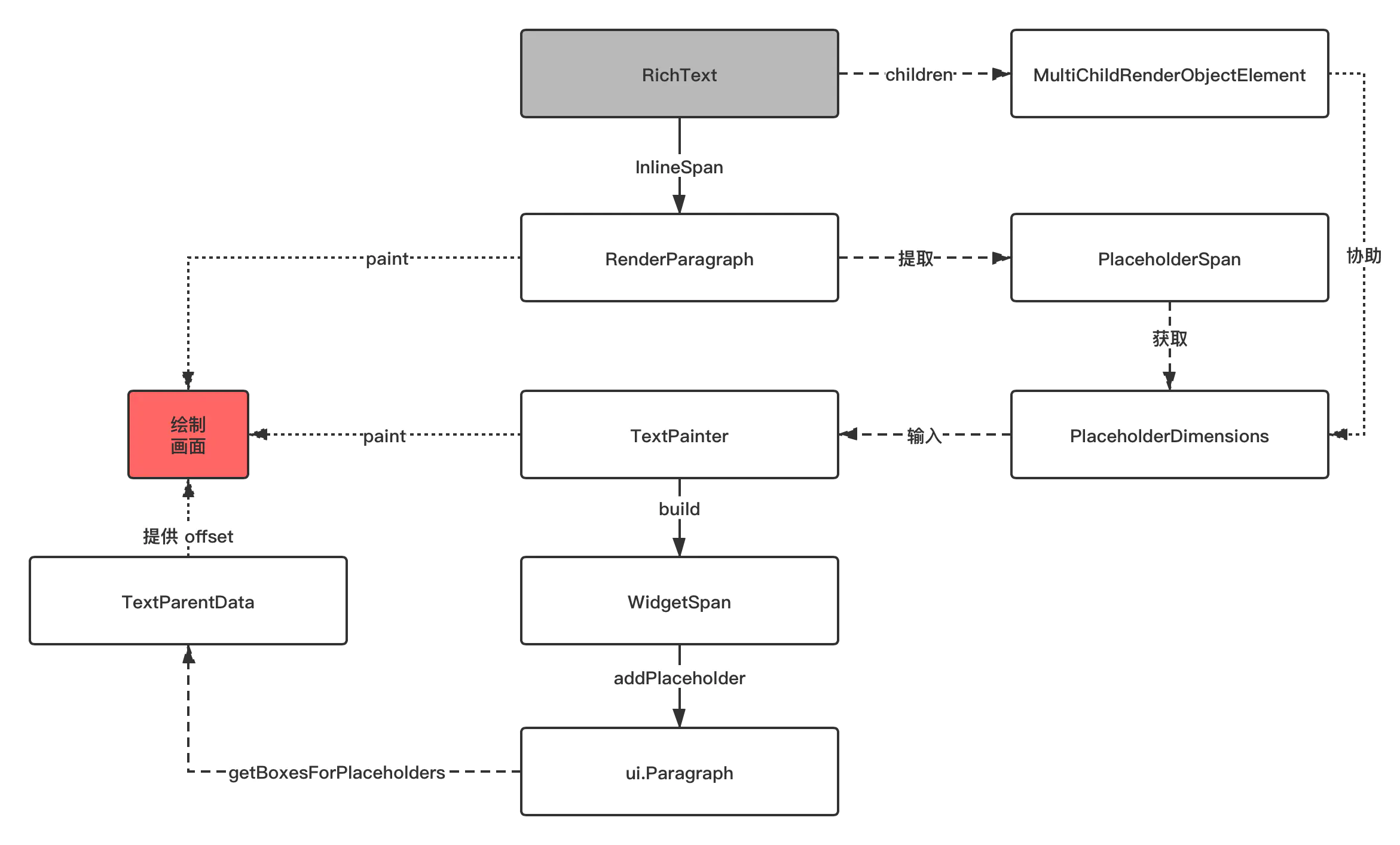
)),

TextSpan(text: 'the best!'),

],

)

通过 Text.rich 接入 TextSpan 和 WidgetSpan 就可以快速实现图文混排的需求，并且可以看出 WidgetSpan 不止支持图片控件，它可以接入任何你需要的 Widget ，比如 Card 、InkWell 等等。

基本原理图如下：

 RichText 中传入 TextSpan ， 在 TextSpan 的 children 中使用 WidgetSpan ，WidgetSpan 里的 Widget 们会转成 MultiChildRenderObjectElement 的 children， 处理后得到一个 child 链表结构；

 之后 TextSpan 进入 RenderParagrash ，会抽取出对应 PlaceholderSpan（WidgetSpan），然后通过转化为 PlaceholderDimensions 保存大小等信息；

 之后进去 TextPainter 会触发 InlineSpan 的 build 方法，从而将前面得到的 PlaceholderDimensions 传递到 WidgetSpan 中；

 WidgetSpan 中的控件信息通过 addPlaceholder 会被传递到 Paragraph；

 之后 TextPainter 中通过 addPlaceholder 的信息获取，调用 \_paragraph.getBoxesForPlaceholders() 获取去控件绘制需要的 offset ；

 有了大小和位置，最终文本中插入的控件，会在 RenderParagrash 的 paint 方法被绘制。

### DefaultTextStyle：在Widget树中，文本的样式默认是可以被继承的（子类文本类组件未指定具体样式时可以使用Widget树中父级设置的默认样式），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，而DefaultTextStyle正是用于设置默认文本样式的。

DefaultTextStyle(

//1.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 world"),

Text("I am Jack"),

Text("I am Jack",

style: TextStyle(

inherit: false, //2.不继承默认样式

color: Colors.grey

),

),

],

),

);

直接看效果：



### 字体：可以在Flutter应用程序中使用不同的字体。例如，我们可能会使用设计人员创建的自定义字体，或者其它第三方的字体，如[Google Fonts](https://fonts.google.com/)中的字体。本节将介绍如何为Flutter应用配置字体，并在渲染文本时使用它们。

在Flutter中使用字体分两步完成。首先在pubspec.yaml中声明它们，以确保它们会打包到应用程序中。然后通过[TextStyle](https://docs.flutter.io/flutter/painting/TextStyle-class.html)属性使用字体。

#### 在asset中声明

要将字体文件打包到应用中，和使用其它资源一样，要先在pubspec.yaml中声明它。然后将字体文件复制到在pubspec.yaml中指定的位置。如：

flutter:

fonts:

- family: Raleway

fonts:

- asset: assets/fonts/Raleway-Regular.ttf

- asset: assets/fonts/Raleway-Medium.ttf

weight: 500

- asset: assets/fonts/Raleway-SemiBold.ttf

weight: 600

- family: AbrilFatface

fonts:

- asset: assets/fonts/abrilfatface/AbrilFatface-Regular.ttf

#### 使用字体

// 声明文本样式

const textStyle = const TextStyle(

fontFamily: 'Raleway',

);

// 使用文本样式

var buttonText = const Text(

"Use the font for this text",

style: textStyle,

);

#### Package中的字体

要使用Package中定义的字体，必须提供package参数。例如，假设上面的字体声明位于my\_package包中。然后创建TextStyle的过程如下：

const textStyle = const TextStyle(

fontFamily: 'Raleway',

package: 'my\_package', //指定包名

);

如果在package包内部使用它自己定义的字体，也应该在创建文本样式时指定package参数，如上例所示。

一个包也可以只提供字体文件而不需要在pubspec.yaml中声明。 这些文件应该存放在包的lib/文件夹中。字体文件不会自动绑定到应用程序中，应用程序可以在声明字体时有选择地使用这些字体。假设一个名为my\_package的包中有一个字体文件：

lib/fonts/Raleway-Medium.ttf

然后，应用程序可以声明一个字体，如下面的示例所示：

flutter:

fonts:

- family: Raleway

fonts:

- asset: assets/fonts/Raleway-Regular.ttf

- asset: packages/my\_package/fonts/Raleway-Medium.ttf

weight: 500

lib/是隐含的，所以它不应该包含在asset路径中。

在这种情况下，由于应用程序本地定义了字体，所以在创建TextStyle时可以不指定package参数：

const textStyle = const TextStyle(

fontFamily: 'Raleway',

);

## 按钮

### Material组件库中的按钮

Material 组件库中提供了多种按钮组件如RaisedButton、FlatButton、OutlineButton等，它们都是直接或间接对RawMaterialButton组件的包装定制，所以他们大多数属性都和RawMaterialButton一样。在介绍各个按钮时我们先介绍其默认外观，而按钮的外观大都可以通过属性来自定义，我们在后面统一介绍这些属性。另外，所有Material 库中的按钮都有如下相同点：

1．按下时都会有“水波动画”（又称“涟漪动画”，就是点击时按钮上会出现水波荡漾的动画）。

2．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

RaisedButton 即"漂浮"按钮，它默认带有阴影和灰色背景。按下后，阴影会变大，如图所示：



使用RaisedButton非常简单，如：

RaisedButton(

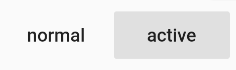
child: Text("normal"),

onPressed: () {},

);

### FlatButton

FlatButton即扁平按钮，默认背景透明并不带阴影。按下后，会有背景色，如图所示：



使用FlatButton也很简单，代码如下：

FlatButton(

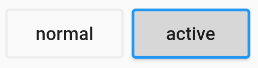
child: Text("normal"),

onPressed: () {},

)

### OutlineButton

OutlineButton默认有一个边框，不带阴影且背景透明。按下后，边框颜色会变亮、同时出现背景和阴影(较弱)，如图所示：



使用OutlineButton也很简单，代码如下：

OutlineButton(

child: Text("normal"),

onPressed: () {},

)

### IconButton

IconButton是一个可点击的Icon，不包括文字，默认没有背景，点击后会出现背景，如图所示：



代码如下：

IconButton(

icon: Icon(Icons.thumb\_up),

onPressed: () {},

)

### 带图标的按钮

RaisedButton、FlatButton、OutlineButton都有一个icon 构造函数，通过它可以轻松创建带图标的按钮，如图3-14所示：



代码如下：

RaisedButton.icon(

icon: Icon(Icons.send),

label: Text("发送"),

onPressed: \_onPressed,

),

OutlineButton.icon(

icon: Icon(Icons.add),

label: Text("添加"),

onPressed: \_onPressed,

),

FlatButton.icon(

icon: Icon(Icons.info),

label: Text("详情"),

onPressed: \_onPressed,

),

### 3.4.2 自定义按钮外观

按钮外观可以通过其属性来定义，不同按钮属性大同小异，我们以FlatButton为例，介绍一下常见的按钮属性，详细的信息可以查看API文档。

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

})

其中大多数属性名都是自解释的，我们不赘述。下面我们通过一个示例来看看如何自定义按钮。

##### 示例

图3-15定义一个背景蓝色，两边圆角的按钮。效果如图所示：

代码如下：

FlatButton(

color: Colors.blue,

highlightColor: Colors.blue[700],

colorBrightness: Brightness.dark,

splashColor: Colors.grey,

child: Text("Submit"),

shape:RoundedRectangleBorder(borderRadius: BorderRadius.circular(20.0)),

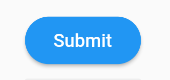
onPressed: () {},

)

很简单吧，在上面的代码中，我们主要通过shape来指定其外形为一个圆角矩形。因为按钮背景是蓝色(深色)，我们需要指定按钮主题colorBrightness为Brightness.dark，这是为了保证按钮文字颜色为浅色。

Flutter 中没有提供去除背景的设置，假若我们需要去除背景，则可以通过将背景颜色设置为全透明来实现。对应上面的代码，便是将 color: Colors.blue 替换为 color: Color(0x000000)。

细心的读者可能会发现这个按钮没有阴影(点击之后也没有)，这样会显得没有质感。其实这也很容易，将上面的FlatButton换成RaisedButton就行，其它代码不用改（这里 color 也不做更改），换了之后的效果如图所示：



是不是有质感了！之所以会这样，是因为RaisedButton默认有配置阴影：

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

...

}

值得注意的是，在Material 组件库中，我们会在很多组件中见到elevation相关的属性，它们都是用来控制阴影的，这是因为阴影在Material设计风格中是一种很重要的表现形式，

## 图片

从asset中加载图片

1.**在工程根目录下创建一个images目录，并将图片avatar.png拷贝到该目录。**

2.**在pubspec.yaml中的flutter部分添加如下内容：**

assets:

- images/avatar.png

注意: 由于 yaml 文件对缩进严格，所以必须严格按照每一层两个空格的方式进行缩进，此处assets前面应有两个空格。

**3.加载该图片：**

Image(

image: AssetImage("images/avatar.png"),

width: 100.0

);

**Image也提供了一个快捷的构造函数Image.asset用于从asset中加载、显示图片：**

Image.asset("images/avatar.png",

width: 100.0,

)

从网络加载图片

Image(

image: NetworkImage(

"https://avatars2.githubusercontent.com/u/20411648?s=460&v=4"),

width: 100.0,

)

Image也提供了一个快捷的构造函数Image.network用于从网络加载、显示图片：

Image.network(

"https://avatars2.githubusercontent.com/u/20411648?s=460&v=4",

width: 100.0,

)

Image的主要参数：

const Image({

...

this.width, //图片的宽

this.height, //图片高度

this.color, //图片的混合色值

this.colorBlendMode, //混合模式

this.fit,//缩放模式

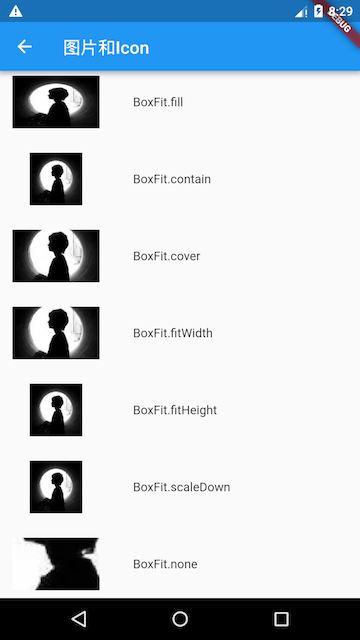
this.alignment = Alignment.center, //对齐方式

this.repeat = ImageRepeat.noRepeat, //重复方式

...

})

* width、height：用于设置图片的宽、高，当不指定宽高时，图片会根据当前父容器的限制，尽可能的显示其原始大小，如果只设置width、height的其中一个，那么另一个属性默认会按比例缩放，但可以通过下面介绍的fit属性来指定适应规则。
* fit：该属性用于在图片的显示空间和图片本身大小不同时指定图片的适应模式。适应模式是在BoxFit中定义，它是一个枚举类型，有如下值：
  + fill：会拉伸填充满显示空间，图片本身长宽比会发生变化，图片会变形。
  + cover：会按图片的长宽比放大后居中填满显示空间，图片不会变形，超出显示空间部分会被剪裁。
  + contain：这是图片的默认适应规则，图片会在保证图片本身长宽比不变的情况下缩放以适应当前显示空间，图片不会变形。
  + fitWidth：图片的宽度会缩放到显示空间的宽度，高度会按比例缩放，然后居中显示，图片不会变形，超出显示空间部分会被剪裁。
  + fitHeight：图片的高度会缩放到显示空间的高度，宽度会按比例缩放，然后居中显示，图片不会变形，超出显示空间部分会被剪裁。
  + none：图片没有适应策略，会在显示空间内显示图片，如果图片比显示空间大，则显示空间只会显示图片中间部分。



* color和colorBlendMode：在图片绘制时可以对每一个像素进行颜色混合处理，color指定混合色，而colorBlendMode指定混合模式，下面是一个简单的示例：

Image(

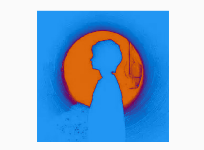
image: AssetImage("images/avatar.png"),

width: 100.0,

color: Colors.blue,

colorBlendMode: BlendMode.difference,

);



* repeat：当图片本身大小小于显示空间时，指定图片的重复规则。简单示例如下：

Image(

image: AssetImage("images/avatar.png"),

width: 100.0,

height: 200.0,

repeat: ImageRepeat.repeatY ,

)

Image缓存

Flutter框架对加载过的图片是有缓存的（内存），默认最大缓存数量是1000，最大缓存空间为100M。

## ICON

在字体文件中，每一个字符都对应一个位码，而每一个位码对应一个显示字形，不同的字体就是指字形不同，即字符对应的字形是不同的。而在iconfont中，只是将位码对应的字形做成了图标，所以不同的字符最终就会渲染成不同的图标。

**\*\*在Flutter开发中，iconfont和图片相比有如下优势：**

·体积小：可以减小安装包大小。

·矢量的：iconfont都是矢量图标，放大不会影响其清晰度。

·可以应用文本样式：可以像文本一样改变字体图标的颜色、大小对齐等。

·可以通过TextSpan和文本混用。

使用Material Design字体图标

Flutter默认包含了一套Material Design的字体图标，在pubspec.yaml文件中的配置如下

flutter:

uses-material-design: **true**

Material Design所有图标可以在其官网查看：<https://material.io/tools/icons/>

举个例子：

String icons = "";

// accessible: &#xE914; or 0xE914 or E914

icons += "\uE914";

// error: &#xE000; or 0xE000 or E000

icons += " \uE000";

// fingerprint: &#xE90D; or 0xE90D or E90D

icons += " \uE90D";

Text(icons,

style: TextStyle(

fontFamily: "MaterialIcons",

fontSize: 24.0,

color: Colors.green

),

);

运行效果如图：



Flutter封装了IconData和Icon来专门显示字体图标，上面的例子也可以用如下方式实现：

Row(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

Icon(Icons.accessible,color: Colors.green,),

Icon(Icons.error,color: Colors.green,),

Icon(Icons.fingerprint,color: Colors.green,),

],

)

Icons类中包含了所有Material Design图标的IconData静态变量定义。

使用自定义字体图标

我们也可以使用自定义字体图标iconfont.cn上有很多字体图标素材，我们可以选择自己需要的图标打包下载后，会生成一些不同格式的字体文件，在Flutter中，我们使用ttf格式即可。

**具体步骤：**

1.导入字体图标文件；这一步和导入字体文件相同，假设我们的字体图标文件保存在项目根目录下，路径为"fonts/iconfont.ttf"：

fonts:

- family: myIcon #指定一个字体名

fonts:

- asset: fonts/iconfont.ttf

2.为了使用方便，我们定义一个MyIcons类，功能和Icons类一样：将字体文件中的所有图标都定义成静态变量：

class MyIcons{

// book 图标

static const IconData book = const IconData(

0xe614,

fontFamily: 'myIcon',

matchTextDirection: true

);

// 微信图标

static const IconData wechat = const IconData(

0xec7d,

fontFamily: 'myIcon',

matchTextDirection: true

);

}

3.可以用啦

Row(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

Icon(MyIcons.book,color: Colors.purple,),

Icon(MyIcons.wechat,color: Colors.green,),

],

)

## 单选开关和复选框

Material 组件库中提供了Material风格的单选开关Switch和复选框Checkbox，虽然它们都是继承自StatefulWidget，但它们本身不会保存当前选中状态，选中状态都是由父组件来管理的。当Switch或Checkbox被点击时，会触发它们的onChanged回调，我们可以在此回调中处理选中状态改变逻辑。

class SwitchAndCheckBoxTestRoute extends StatefulWidget {

@override

\_SwitchAndCheckBoxTestRouteState createState() => new \_SwitchAndCheckBoxTestRouteState();

}

class \_SwitchAndCheckBoxTestRouteState extends State<SwitchAndCheckBoxTestRoute> {

bool \_switchSelected=true; //维护单选开关状态

bool \_checkboxSelected=true;//维护复选框状态

@override

Widget build(BuildContext context) {

return Column(

children: <Widget>[

Switch(

value: \_switchSelected,//当前状态

onChanged:(value){

//重新构建页面

setState(() {

\_switchSelected=value;

});

},

),

Checkbox(

value: \_checkboxSelected,

activeColor: Colors.red, //选中时的颜色

onChanged:(value){

setState(() {

\_checkboxSelected=value;

});

} ,

)

],

);

}

}

Switch和Checkbox属性比较简单，它们都有一个activeColor属性，用于设置激活态的颜色。至于大小，到目前为止，Checkbox的大小是固定的，无法自定义，而Switch只能定义宽度，高度也是固定的。值得一提的是Checkbox有一个属性tristate ，表示是否为三态，其默认值为false，这时Checkbox有两种状态即“选中”和“不选中”，对应的value值为true和false。如果tristate值为true时，value的值会增加一个状态null。

### CheckboxListTile

CheckboxListTile 是一个 Checkbox 的上层封装，它的外观是提供类似设置页的选择组件，可设置图标和文字。

new CheckboxListTile(

secondary: const Icon(Icons.shutter\_speed),

title: const Text('硬件加速'),

value: this.check,

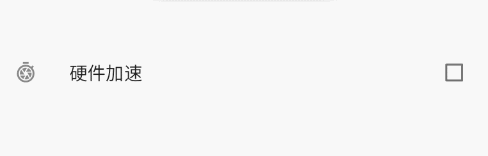
onChanged: (bool value) {

setState(() {

this.check = !this.check;

});

},

),

## 输入框及表单

Material组件库中提供了输入框组件TextField和表单组件Form。

TextField

TextField用于文本输入，并提供了很多属性：

const TextField({

...

TextEditingController controller,

FocusNode focusNode,

InputDecoration decoration = const InputDecoration(),

TextInputType keyboardType,

TextInputAction textInputAction,

TextStyle style,

TextAlign textAlign = TextAlign.start,

bool autofocus = false,

bool obscureText = false,

int maxLines = 1,

int maxLength,

bool maxLengthEnforced = true,

ValueChanged<String> onChanged,

VoidCallback onEditingComplete,

ValueChanged<String> onSubmitted,

List<TextInputFormatter> inputFormatters,

bool enabled,

this.cursorWidth = 2.0,

this.cursorRadius,

this.cursorColor,

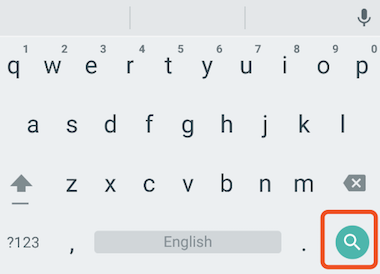
...

})

* controller：编辑框的控制器，通过它可以设置/获取编辑框的内容、选择编辑内容、监听编辑文本改变事件。大多数情况下我们都需要显式提供一个controller来与文本框交互。如果没有提供controller，则TextField内部会自动创建一个。
* focusNode：用于控制TextField是否占有当前键盘的输入焦点。它是我们和键盘交互的一个句柄（handle）。
* InputDecoration：用于控制TextField的外观显示，如提示文本、背景颜色、边框等。
* keyboardType：用于设置该输入框默认的键盘输入类型，取值如下：

| **TextInputType枚举值** | **含义** |
| --- | --- |
| text | 文本输入键盘 |
| multiline | 多行文本，需和maxLines配合使用(设为null或大于1) |
| number | 数字；会弹出数字键盘 |
| phone | 优化后的电话号码输入键盘；会弹出数字键盘并显示“\* #” |
| datetime | 优化后的日期输入键盘；Android上会显示“: -” |
| emailAddress | 优化后的电子邮件地址；会显示“@ .” |
| url | 优化后的url输入键盘； 会显示“/ .” |

* textInputAction：键盘动作按钮图标(即回车键位图标)，它是一个枚举值，有多个可选值，全部的取值列表可以查看API文档，下面是当值为TextInputAction.search时，原生Android系统下键盘样式如图所示：



* style：正在编辑的文本样式。
* textAlign: 输入框内编辑文本在水平方向的对齐方式。
* autofocus: 是否自动获取焦点。
* obscureText：是否隐藏正在编辑的文本，如用于输入密码的场景等，文本内容会用“•”替换。
* maxLines：输入框的最大行数，默认为1；如果为null，则无行数限制。
* maxLength和maxLengthEnforced ：maxLength代表输入框文本的最大长度，设置后输入框右下角会显示输入的文本计数。maxLengthEnforced决定当输入文本长度超过maxLength时是否阻止输入，为true时会阻止输入，为false时不会阻止输入但输入框会变红。
* onChange：输入框内容改变时的回调函数；注：内容改变事件也可以通过controller来监听。
* onEditingComplete和onSubmitted：这两个回调都是在输入框输入完成时触发，比如按了键盘的完成键（对号图标）或搜索键（🔍图标）。不同的是两个回调签名不同，onSubmitted回调是ValueChanged<String>类型，它接收当前输入内容做为参数，而onEditingComplete不接收参数。
* inputFormatters：用于指定输入格式；当用户输入内容改变时，会根据指定的格式来校验。
* enable：如果为false，则输入框会被禁用，禁用状态不接收输入和事件，同时显示禁用态样式（在其decoration中定义）。
* cursorWidth、cursorRadius和cursorColor：这三个属性是用于自定义输入框光标宽度、圆角和颜色的。

#### 示例：登录输入框

##### 布局

Column(

children: <Widget>[

TextField(

autofocus: true,

decoration: InputDecoration(

labelText: "用户名",

hintText: "用户名或邮箱",

prefixIcon: Icon(Icons.person)

),

),

TextField(

decoration: InputDecoration(

labelText: "密码",

hintText: "您的登录密码",

prefixIcon: Icon(Icons.lock)

),

obscureText: true,

),

],

);

运行后，效果如图所示：



##### 获取输入内容

获取输入内容有两种方式：

**1.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。**

**2.通过controller直接获取。**

定义一个controller：

//定义一个controller

TextEditingController \_unameController = TextEditingController();

然后设置输入框controller：

TextField(

autofocus: true,

controller: \_unameController, //设置controller

...

)

通过controller获取输入框内容

print(\_unameController.text)

##### 监听文本变化

**监听文本变化也有两种方式：**

设置onChange回调，如：

TextField(

autofocus: true,

onChanged: (v) {

print("onChange: $v");

}

)

通过controller监听，如：

@override

void initState() {

//监听输入改变

\_unameController.addListener((){

print(\_unameController.text);

});

}

两种方式相比，onChanged是专门用于监听文本变化，而controller的功能却多一些，除了能监听文本变化外，它还可以设置默认值、选择文本，下面我们看一个例子：

**1.创建一个controller:**

TextEditingController \_selectionController = TextEditingController();

**2.设置默认值，并从第三个字符开始选中后面的字符**

\_selectionController.text="hello world!";

\_selectionController.selection=TextSelection(

baseOffset: 2,

extentOffset: \_selectionController.text.length

);

**3.设置controller:**

TextField(

controller: \_selectionController,

)

运行效果如图所示：

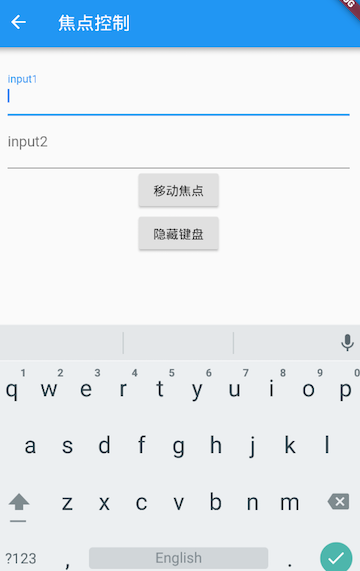
图3-26

##### 控制焦点

焦点可以通过FocusNode和FocusScopeNode来控制，默认情况下，焦点由FocusScope来管理，它代表焦点控制范围，可以在这个范围内可以通过FocusScopeNode在输入框之间移动焦点、设置默认焦点等。我们可以通过FocusScope.of(context) 来获取Widget树中默认的FocusScopeNode。下面看一个示例，在此示例中创建两个TextField，第一个自动获取焦点，然后创建两个按钮：

* 点击第一个按钮可以将焦点从第一个TextField挪到第二个TextField。
* 点击第二个按钮可以关闭键盘。

我们要实现的效果如图所示：



代码如下：

class FocusTestRoute extends StatefulWidget {

@override

\_FocusTestRouteState createState() => new \_FocusTestRouteState();

}

class \_FocusTestRouteState extends State<FocusTestRoute> {

FocusNode focusNode1 = new FocusNode();

FocusNode focusNode2 = new FocusNode();

FocusScopeNode focusScopeNode;

@override

Widget build(BuildContext context) {

return Padding(

padding: EdgeInsets.all(16.0),

child: Column(

children: <Widget>[

TextField(

autofocus: true,

focusNode: focusNode1,//关联focusNode1

decoration: InputDecoration(

labelText: "input1"

),

),

TextField(

focusNode: focusNode2,//关联focusNode2

decoration: InputDecoration(

labelText: "input2"

),

),

Builder(builder: (ctx) {

return Column(

children: <Widget>[

RaisedButton(

child: Text("移动焦点"),

onPressed: () {

//将焦点从第一个TextField移到第二个TextField

// 这是一种写法 FocusScope.of(context).requestFocus(focusNode2);

// 这是第二种写法

if(null == focusScopeNode){

focusScopeNode = FocusScope.of(context);

}

focusScopeNode.requestFocus(focusNode2);

},

),

RaisedButton(

child: Text("隐藏键盘"),

onPressed: () {

// 当所有编辑框都失去焦点时键盘就会收起

focusNode1.unfocus();

focusNode2.unfocus();

},

),

],

);

},

),

],

),

);

}

}

FocusNode和FocusScopeNode还有一些其它的方法，详情可以查看API文档。

##### 监听焦点状态改变事件

FocusNode继承自ChangeNotifier，通过FocusNode可以监听焦点的改变事件，如：

...

// 创建 focusNode

FocusNode focusNode = new FocusNode();

...

// focusNode绑定输入框

TextField(focusNode: focusNode);

...

// 监听焦点变化

focusNode.addListener((){

print(focusNode.hasFocus);

});

获得焦点时focusNode.hasFocus值为true，失去焦点时为false。

##### 自定义样式

虽然我们可以通过decoration属性来定义输入框样式，下面以自定义输入框下划线颜色为例来介绍一下：

TextField(

decoration: InputDecoration(

labelText: "请输入用户名",

prefixIcon: Icon(Icons.person),

// 未获得焦点下划线设为灰色

enabledBorder: UnderlineInputBorder(

borderSide: BorderSide(color: Colors.grey),

),

//获得焦点下划线设为蓝色

focusedBorder: UnderlineInputBorder(

borderSide: BorderSide(color: Colors.blue),

),

),

),

上面代码我们直接通过InputDecoration的enabledBorder和focusedBorder来分别设置了输入框在未获取焦点和获得焦点后的下划线颜色。另外，我们也可以通过主题来自定义输入框的样式，下面我们探索一下如何在不使用enabledBorder和focusedBorder的情况下来自定义下滑线颜色。

由于TextField在绘制下划线时使用的颜色是主题色里面的hintColor，但提示文本颜色也是用的hintColor， 如果我们直接修改hintColor，那么下划线和提示文本的颜色都会变。值得高兴的是decoration中可以设置hintStyle，它可以覆盖hintColor，并且主题中可以通过inputDecorationTheme来设置输入框默认的decoration。所以我们可以通过主题来自定义，代码如下：

Theme(

data: Theme.of(context).copyWith(

hintColor: Colors.grey[200], //定义下划线颜色

inputDecorationTheme: InputDecorationTheme(

labelStyle: TextStyle(color: Colors.grey),//定义label字体样式

hintStyle: TextStyle(color: Colors.grey, fontSize: 14.0)//定义提示文本样式

)

),

child: Column(

children: <Widget>[

TextField(

decoration: InputDecoration(

labelText: "用户名",

hintText: "用户名或邮箱",

prefixIcon: Icon(Icons.person)

),

),

TextField(

decoration: InputDecoration(

prefixIcon: Icon(Icons.lock),

labelText: "密码",

hintText: "您的登录密码",

hintStyle: TextStyle(color: Colors.grey, fontSize: 13.0)

),

obscureText: true,

)

],

)

)

运行效果如图所示：



我们成功的自定义了下划线颜色和提问文字样式，细心的读者可能已经发现，通过这种方式自定义后，输入框在获取焦点时，labelText不会高亮显示了，正如上图中的"用户名"本应该显示蓝色，但现在却显示为灰色，并且我们还是无法定义下划线宽度。另一种灵活的方式是直接隐藏掉TextField本身的下划线，然后通过Container去嵌套定义样式，如:

Container(

child: TextField(

keyboardType: TextInputType.emailAddress,

decoration: InputDecoration(

labelText: "Email",

hintText: "电子邮件地址",

prefixIcon: Icon(Icons.email),

border: InputBorder.none //隐藏下划线

)

),

decoration: BoxDecoration(

// 下滑线浅灰色，宽度1像素

border: Border(bottom: BorderSide(color: Colors.grey[200], width: 1.0))

),

)

运行效果：

image-20180904150511545

通过这种组件组合的方式，也可以定义背景圆角等。一般来说，优先通过decoration来自定义样式，如果decoration实现不了，再用widget组合的方式。

表单Form

实际业务中，在正式向服务器提交数据前，都会对各个输入框数据进行合法性校验，但是对每一个TextField都分别进行校验将会是一件很麻烦的事。还有，如果用户想清除一组TextField的内容，除了一个一个清除有没有什么更好的办法呢？为此，Flutter提供了一个Form组件，它可以对输入框进行分组，然后进行一些统一操作，如输入内容校验、输入框重置以及输入内容保存。

#### Form

Form继承自StatefulWidget对象，它对应的状态类为FormState。我们先看看Form类的定义：

Form({

@required Widget child,

bool autovalidate = false,

WillPopCallback onWillPop,

VoidCallback onChanged,

})

* autovalidate：是否自动校验输入内容；当为true时，每一个子FormField内容发生变化时都会自动校验合法性，并直接显示错误信息。否则，需要通过调用FormState.validate()来手动校验。
* onWillPop：决定Form所在的路由是否可以直接返回（如点击返回按钮），该回调返回一个Future对象，如果Future的最终结果是false，则当前路由不会返回；如果为true，则会返回到上一个路由。此属性通常用于拦截返回按钮。
* onChanged：Form的任意一个子FormField内容发生变化时会触发此回调。

#### FormField

Form的子孙元素必须是FormField类型，FormField是一个抽象类，定义几个属性，FormState内部通过它们来完成操作，FormField部分定义如下：

const FormField({

...

FormFieldSetter<T> onSaved, //保存回调

FormFieldValidator<T> validator, //验证回调

T initialValue, //初始值

bool autovalidate = false, //是否自动校验。

})

为了方便使用，Flutter提供了一个TextFormField组件，它继承自FormField类，也是TextField的一个包装类，所以除了FormField定义的属性之外，它还包括TextField的属性。

#### FormState

FormState为Form的State类，可以通过Form.of()或GlobalKey获得。我们可以通过它来对Form的子孙FormField进行统一操作。我们看看其常用的三个方法：

* FormState.validate()：调用此方法后，会调用Form子孙FormField的validate回调，如果有一个校验失败，则返回false，所有校验失败项都会返回用户返回的错误提示。
* FormState.save()：调用此方法后，会调用Form子孙FormField的save回调，用于保存表单内容
* FormState.reset()：调用此方法后，会将子孙FormField的内容清空。

#### 示例

我们修改一下上面用户登录的示例，在提交之前校验：

1.用户名不能为空，如果为空则提示“用户名不能为空”。

2.密码不能小于6位，如果小于6为则提示“密码不能少于6位”。

完整代码：

class FormTestRoute extends StatefulWidget {

@override

\_FormTestRouteState createState() => new \_FormTestRouteState();

}

class \_FormTestRouteState extends State<FormTestRoute> {

TextEditingController \_unameController = new TextEditingController();

TextEditingController \_pwdController = new TextEditingController();

GlobalKey \_formKey= new GlobalKey<FormState>();

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title:Text("Form Test"),

),

body: Padding(

padding: const EdgeInsets.symmetric(vertical: 16.0, horizontal: 24.0),

child: Form(

key: \_formKey, //设置globalKey，用于后面获取FormState

autovalidate: true, //开启自动校验

child: Column(

children: <Widget>[

TextFormField(

autofocus: true,

controller: \_unameController,

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(

controller: \_pwdController,

decoration: InputDecoration(

labelText: "密码",

hintText: "您的登录密码",

icon: Icon(Icons.lock)

),

obscureText: true,

//校验密码

validator: (v) {

return v

.trim()

.length > 5 ? null : "密码不能少于6位";

}

),

// 登录按钮

Padding(

padding: const EdgeInsets.only(top: 28.0),

child: Row(

children: <Widget>[

Expanded(

child: RaisedButton(

padding: EdgeInsets.all(15.0),

child: Text("登录"),

color: Theme

.of(context)

.primaryColor,

textColor: Colors.white,

onPressed: () {

//在这里不能通过此方式获取FormState，context不对

//print(Form.of(context));

// 通过\_formKey.currentState 获取FormState后，

// 调用validate()方法校验用户名密码是否合法，校验

// 通过后再提交数据。

if((\_formKey.currentState as FormState).validate()){

//验证通过提交数据

}

},

),

),

],

),

)

],

),

),

),

);

}

}

运行后效果如图所示：



注意，登录按钮的onPressed方法中不能通过Form.of(context)来获取，原因是，此处的context为FormTestRoute的context，而Form.of(context)是根据所指定context向根去查找，而FormState是在FormTestRoute的子树中，所以不行。正确的做法是通过Builder来构建登录按钮，Builder会将widget节点的context作为回调参数：

Expanded(

// 通过Builder来获取RaisedButton所在widget树的真正context(Element)

child:Builder(builder: (context){

return RaisedButton(

...

onPressed: () {

//由于本widget也是Form的子代widget，所以可以通过下面方式获取FormState

if(Form.of(context).validate()){

//验证通过提交数据

}

},

);

})

)

其实context正是操作Widget所对应的Element的一个接口，由于Widget树对应的Element都是不同的，所以context也都是不同的。Flutter中有很多“of(context)”这种方法，在使用时一定要注意context是否正确。

## 进度指示器

写在前面：LinearProgressIndicator和CircularProgressIndicator都是取父容器的尺寸作为绘制的边界的。

另：动画（初始化可以定义动画类型及方式等）和自定义样式（可以通过CustomPainter Widget 来自定义绘制逻辑）可参考API。

LinearProgressIndicator：是一个线性、条状的进度条，定义如下：

LinearProgressIndicator({

double value,

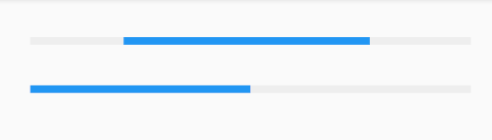
Color backgroundColor,

Animation<Color> valueColor,

...

})

效果如图：



* value：value表示当前的进度，取值范围为[0,1]；如果value为null时则指示器会执行一个循环动画（模糊进度）；当value不为null时，指示器为一个具体进度的进度条。
* backgroundColor：指示器的背景色。
* valueColor: 指示器的进度条颜色；值得注意的是，该值类型是Animation<Color>，这允许我们对进度条的颜色也可以指定动画。如果我们不需要对进度条颜色执行动画，换言之，我们想对进度条应用一种固定的颜色，此时我们可以通过AlwaysStoppedAnimation来指定。

CircularProgressIndicator是一个圆形进度条，定义如下：

CircularProgressIndicator({

double value,

Color backgroundColor,

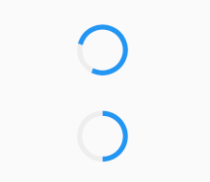
Animation<Color> valueColor,

this.strokeWidth = 4.0, //strokeWidth表示圆形进度条的粗细

...

})

效果如图：



# 布局类组件

## 线性布局Row、Column

**写在前面：实际上，Row和Column都只会在主轴方向占用尽可能大的空间，而纵轴的长度则取决于他们最大子元素的长度。**

Row

Row可以在水平方向排列其子widget。定义如下：

Row({

...

TextDirection textDirection,

MainAxisSize mainAxisSize = MainAxisSize.max,

MainAxisAlignment mainAxisAlignment = MainAxisAlignment.start,

VerticalDirection verticalDirection = VerticalDirection.down,

CrossAxisAlignment crossAxisAlignment = CrossAxisAlignment.center,

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

})

* textDirection：表示水平方向子组件的布局顺序(是从左往右还是从右往左)，默认为系统当前Locale环境的文本方向(如中文、英语都是从左往右，而阿拉伯语是从右往左)。
* mainAxisSize：表示Row在主轴(水平)方向占用的空间，默认是MainAxisSize.max，表示尽可能多的占用水平方向的空间，此时无论子widgets实际占用多少水平空间，Row的宽度始终等于水平方向的最大宽度；而MainAxisSize.min表示尽可能少的占用水平空间，当子组件没有占满水平剩余空间，则Row的实际宽度等于所有子组件占用的的水平空间；
* mainAxisAlignment：表示子组件在Row所占用的水平空间内对齐方式，如果mainAxisSize值为MainAxisSize.min，则此属性无意义，因为子组件的宽度等于Row的宽度。只有当mainAxisSize的值为MainAxisSize.max时，此属性才有意义，MainAxisAlignment.start表示沿textDirection的初始方向对齐，如textDirection取值为TextDirection.ltr时，则MainAxisAlignment.start表示左对齐，textDirection取值为TextDirection.rtl时表示从右对齐。而MainAxisAlignment.end和MainAxisAlignment.start正好相反；MainAxisAlignment.center表示居中对齐。读者可以这么理解：textDirection是mainAxisAlignment的参考系。
* verticalDirection：表示Row纵轴（垂直）的对齐方向，默认是VerticalDirection.down，表示从上到下。
* crossAxisAlignment：表示子组件在纵轴方向的对齐方式，Row的高度等于子组件中最高的子元素高度，它的取值和MainAxisAlignment一样(包含start、end、 center三个值)，不同的是crossAxisAlignment的参考系是verticalDirection，即verticalDirection值为VerticalDirection.down时crossAxisAlignment.start指顶部对齐，verticalDirection值为VerticalDirection.up时，crossAxisAlignment.start指底部对齐；而crossAxisAlignment.end和crossAxisAlignment.start正好相反；
* children：子组件数组。

Column

Column可以在垂直方向排列其子组件。参数和Row一样，不同的是布局方向为垂直，主轴纵轴正好相反.

## 弹性布局Flex

**写在前面：Flutter中的弹性布局主要通过Flex和Expanded来配合实现。**

class FlexLayoutTestRoute extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Column(

children: <Widget>[

//Flex的两个子widget按1：2来占据水平空间

Flex(

direction: Axis.horizontal,

children: <Widget>[

Expanded(

flex: 1,

child: Container(

height: 30.0,

color: Colors.red,

),

),

Expanded(

flex: 2,

child: Container(

height: 30.0,

color: Colors.green,

),

),

],

),

Padding(

padding: const EdgeInsets.only(top: 20.0),

child: SizedBox(

height: 100.0,

//Flex的三个子widget，在垂直方向按2：1：1来占用100像素的空间

child: Flex(

direction: Axis.vertical,

children: <Widget>[

Expanded(

flex: 2,

child: Container(

height: 30.0,

color: Colors.red,

),

),

Spacer(

flex: 1,

),

Expanded(

flex: 1,

child: Container(

height: 30.0,

color: Colors.green,

),

),

],

),

),

),

],

);

}

}

**代码梳理：Flex为弹性布局，子组件为Expanded。可以都通过flex设置值进行弹性撑开，也可以固定其中某一个或多个组件的固定宽度或者高度，剩余组件进行自适应！**

## 流式布局Wrap、Flow

Wrap

Wrap({

...

this.direction = Axis.horizontal,

this.alignment = WrapAlignment.start,

this.spacing = 0.0,

this.runAlignment = WrapAlignment.start,

this.runSpacing = 0.0,

this.crossAxisAlignment = WrapCrossAlignment.start,

this.textDirection,

this.verticalDirection = VerticalDirection.down,

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

})

**Wrap特有的几个属性：**

* spacing：主轴方向子widget的间距
* runSpacing：纵轴方向的间距
* runAlignment：纵轴方向的对齐方式

**这里可以举个例子：**

Wrap(

spacing: 8.0, // 主轴(水平)方向间距

runSpacing: 4.0, // 纵轴（垂直）方向间距

alignment: WrapAlignment.center, //沿主轴方向居中

children: <Widget>[

new Chip(

avatar: new CircleAvatar(backgroundColor: Colors.blue, child: Text('A')),

label: new Text('Hamilton'),

),

new Chip(

avatar: new CircleAvatar(backgroundColor: Colors.blue, child: Text('M')),

label: new Text('Lafayette'),

),

new Chip(

avatar: new CircleAvatar(backgroundColor: Colors.blue, child: Text('H')),

label: new Text('Mulligan'),

),

new Chip(

avatar: new CircleAvatar(backgroundColor: Colors.blue, child: Text('J')),

label: new Text('Laurens'),

),

],

)



### Flow

极少使用，需要自己实现子widget的位置转换，主要用于一些需要自定义布局策略或性能要求较高(如动画中)的场景。

优点：

1.性能好；Flow是一个对子组件尺寸以及位置调整非常高效的控件，Flow用转换矩阵在对子组件进行位置调整的时候进行了优化：在Flow定位过后，如果子组件的尺寸或者位置发生了变化，在FlowDelegate中的paintChildren()方法中调用context.paintChild 进行重绘，而context.paintChild在重绘时使用了转换矩阵，并没有实际调整组件位置。

2.灵活；由于我们需要自己实现FlowDelegate的paintChildren()方法，所以我们需要自己计算每一个组件的位置，因此，可以自定义布局策略。

缺点：

1.使用复杂。

2.不能自适应子组件大小，必须通过指定父容器大小或实现TestFlowDelegate的getSize返回固定大小。

备注：具体使用规则可查看相关文档，哈哈哈，这个我也不熟

## 层叠布局Stack、Positioned

**子组件可以根据距父容器四个角的位置来确定自身的位置。绝对定位允许子组件堆叠起来（按照代码中声明的顺序）。Flutter中使用Stack和Positioned这两个组件来配合实现绝对定位。Stack允许子组件堆叠，而Positioned用于根据Stack的四个角来确定子组件的位置。**

Stack

Stack({

this.alignment = AlignmentDirectional.topStart,

this.textDirection,

this.fit = StackFit.loose,

this.overflow = Overflow.clip,

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

})

* alignment：此参数决定如何去对齐没有定位（没有使用Positioned）或部分定位的子组件。所谓部分定位，在这里**特指没有在某一个轴上定位：**left、right为横轴，top、bottom为纵轴，只要包含某个轴上的一个定位属性就算在该轴上有定位。
* textDirection：和Row、Wrap的textDirection功能一样，都用于确定alignment对齐的参考系，即：textDirection的值为TextDirection.ltr，则alignment的start代表左，end代表右，即从左往右的顺序；textDirection的值为TextDirection.rtl，则alignment的start代表右，end代表左，即从右往左的顺序。
* fit：此参数用于确定**没有定位**的子组件如何去适应Stack的大小。StackFit.loose表示使用子组件的大小，StackFit.expand表示扩伸到Stack的大小。
* overflow：此属性决定如何显示超出Stack显示空间的子组件；值为Overflow.clip时，超出部分会被剪裁（隐藏），值为Overflow.visible时则不会。

Position

const Positioned({

Key key,

this.left,

this.top,

this.right,

this.bottom,

this.width,

this.height,

@required Widget child,

})

left、top、right、bottom分别代表离Stack左、上、右、底四边的距离。width和height用于指定需要定位元素的宽度和高度。注意，Positioned的width、height和其它地方的意义稍微有点区别，此处用于配合left、top、right、bottom来定位组件，举个例子，在水平方向时，你只能指定left、right、width三个属性中的两个，如指定left和width后，right会自动算出(left+width)，如果同时指定三个属性则会报错，垂直方向同理。

一个简单的例子：

Stack(

alignment:Alignment.center ,

fit: StackFit.expand, //未定位widget占满Stack整个空间

children: <Widget>[

Positioned(

left: 18.0,

child: Text("I am Jack"),

),

Container(child: Text("Hello world",style: TextStyle(color: Colors.white)),

color: Colors.red,

),

Positioned(

top: 18.0,

child: Text("Your friend"),

)

],

),

## 对齐与相对定位 Align

**用来满足调整一个子元素在父元素中的位置。**

Align组件可以调整子组件的位置，并且可以根据子组件的宽高来确定自身的的宽高：

Align({

Key key,

this.alignment = Alignment.center,

this.widthFactor,

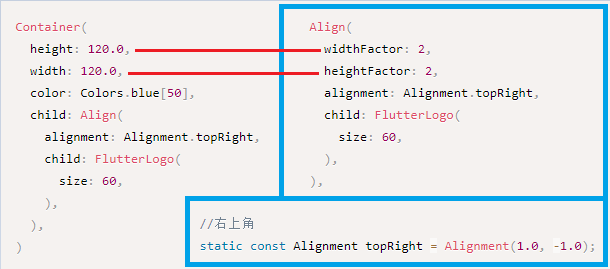
this.heightFactor,

Widget child,

})

**示例：**

\*FlutterLogo是Flutter SDK提供的一个组件，内容就是Flutter的商标。

* alignment: 需要一个AlignmentGeometry类型的值，表示子组件在父组件中的起始位置。AlignmentGeometry是一个抽象类，它有两个常用的子类：**Alignment**和**FractionalOffset**。

Alignment

Alignment继承自AlignmentGeometry，表示矩形内的一个点，他有两个属性x、y，分别表示在水平和垂直方向的偏移，Alignment定义如下：

Alignment(this.x, this.y)

Alignment Widget会以**矩形的中心点作为坐标原点**，即Alignment(0.0, 0.0)。x、y的值从-1到1分别代表矩形左边到右边的距离和顶部到底边的距离，因此2个水平（或垂直）单位则等于矩形的宽（或高），如Alignment(-1.0, -1.0)代表矩形的左侧顶点，而Alignment(1.0, 1.0)代表右侧底部终点，而Alignment(1.0, -1.0) 则正是右侧顶点，即Alignment.topRight。为了使用方便，矩形的原点、四个顶点，以及四条边的终点在Alignment类中都已经定义为了静态常量。

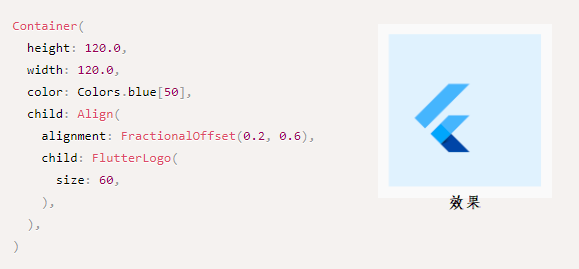
Alignment可以通过其**坐标转换公式**将其坐标转为子元素的具体偏移坐标：

**(Alignment.x\*childWidth/2+childWidth/2, Alignment.y\*childHeight/2+childHeight/2)**

其中childWidth为子元素的宽度，childHeight为子元素高度。

FractionalOffset

FractionalOffset继承自Alignment，它和Alignment唯一的区别就是坐标原点不同！FractionalOffset的坐标原点为矩形的左侧顶点，这和布局系统的一致。FractionalOffset的坐标转换公式为：

**实际偏移 = (FractionalOffse.x \* childWidth, FractionalOffse.y \* childHeight)**

* widthFactor和heightFactor是用于确定Align组件本身宽高的属性；它们是两个缩放因子，会分别乘以子元素的宽、高，最终的结果就是Align组件的宽高。如果值为null，则组件的宽高将会占用尽可能多的空间。

Center组件

class Center extends Align {

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

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

}

Center继承自Align，它比Align只少了一个alignment参数；由于Align的构造函数中alignment值为Alignment.center，所以，我们可以认为Center组件其实是对齐方式确定（Alignment.center）了的Align。

# 容器类组件

## 填充（Padding）

**直接上代码：**

Padding({

...

EdgeInsetsGeometry padding,

Widget child,

})

**EdgeInsets：**

* fromLTRB(double left, double top, double right, double bottom)：分别指定四个方向的填充。
* all(double value) : 所有方向均使用相同数值的填充。
* only({left, top, right ,bottom })：可以设置具体某个方向的填充(可以同时指定多个方向)。
* symmetric({ vertical, horizontal })：用于设置对称方向的填充，vertical指top和bottom，horizontal指left和right。

## 尺寸限制类容器（ConstrainedBox等）

ConstrainedBox

ConstrainedBox用于对子组件添加额外的约束:

ConstrainedBox(  
 constraints: BoxConstraints(  
 minWidth: double.*infinity*, //宽度尽可能大  
 minHeight: 50.0 //最小高度为50像素  
 ),  
 child: Container(  
 height: 5.0,//高度设置为5像素,但ConstrainedBox最小高度限制生效，显示效果依旧是50像素,  
 child: DecoratedBox(  
 decoration: BoxDecoration(color: Colors.*red*),  
 )  
 ),  
)

BoxConstraints

BoxConstraints用于设置限制条件，它的定义如下：

const BoxConstraints({

this.minWidth = 0.0, //最小宽度

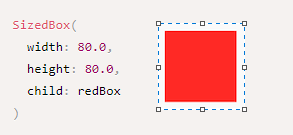
this.maxWidth = double.infinity, //最大宽度

this.minHeight = 0.0, //最小高度

this.maxHeight = double.infinity //最大高度

})

SizedBox

SizedBox用于给子元素指定固定的宽高.(可以用于快速撑开组件之间的距离，设置分割线等操作)

重点说这么一件事：

有多重限制时，对于minWidth和minHeight来说，是取父子中相应数值较大的。实际上，只有这样才能保证父限制与子限制不冲突。

另外，有时候父组件进行了限制，或者存在多重限制时，有子组件需要“脱离”这种限制，那么可以使用**UnconstrainedBox**来实现。

UnconstrainedBox对父组件限制的“去除”并非是真正的去除：父限制的条件仍然是生效的，只不过它不影响最终子元素的大小，但仍然还是占有相应的空间，可以认为此时的父ConstrainedBox是作用于子UnconstrainedBox上，而当前组件只受子ConstrainedBox限制.

其他：

AspectRatio：可以指定子组件的长宽比

LimitedBox：用于指定最大宽高

FractionallySizedBox：可以根据父容器宽高的百分比来设置子组件宽高

## 装饰容器DecoratedBox

DecoratedBox可以在其子组件绘制前(或后)绘制一些装饰（Decoration），如背景、边框、渐变等。DecoratedBox定义如下：

const DecoratedBox({

Decoration decoration,

DecorationPosition position = DecorationPosition.background,

Widget child

})

* decoration：代表将要绘制的装饰，它的类型为Decoration。Decoration是一个抽象类，它定义了一个接口createBoxPainter()，子类的主要职责是需要通过实现它来创建一个画笔，该画笔用于绘制装饰。
* position：此属性决定在哪里绘制Decoration，它接收DecorationPosition的枚举类型，该枚举类有两个值：
  + background：在子组件之后绘制，即背景装饰。
  + foreground：在子组件之上绘制，即前景。

BoxDecoration

我们通常会直接使用BoxDecoration类，它是一个Decoration的子类，实现了常用的装饰元素的绘制。

BoxDecoration({

Color color, //颜色

DecorationImage image,//图片

BoxBorder border, //边框

BorderRadiusGeometry borderRadius, //圆角

List<BoxShadow> boxShadow, //阴影,可以指定多个

Gradient gradient, //渐变

BlendMode backgroundBlendMode, //背景混合模式

BoxShape shape = BoxShape.rectangle, //形状

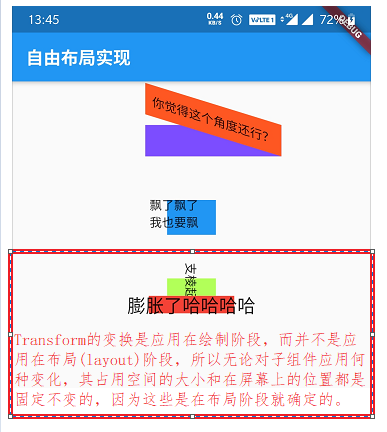
})

**直接撸代码：**

注意： 如果Container里面加上decoration属性，这个时候color属性必须得放在BoxDecoration

## 变换（Transform）

Transform实操性比较强的部分，直接撸代码：

body: Column(  
 children: <Widget>[  
 Container(  
 color: Colors.*deepPurpleAccent*,  
 margin: EdgeInsets.only(top: 50, left: 50),  
// height: 50,  
 child: new Transform(  
 alignment: Alignment.*topRight*, //相对于坐标系原点的对齐方式  
 transform: new Matrix4.skewY(0.3), //沿Y轴倾斜0.3弧度  
 child: new Container(  
 padding: const EdgeInsets.all(8.0),  
 color: Colors.*deepOrange*,  
 child: const Text('你觉得这个角度还行？'),  
 ),  
 ),  
 ),  
 Padding(  
 padding: EdgeInsets.all(50),  
 child: DecoratedBox(  
 decoration: BoxDecoration(color: Colors.*blue*),  
 //默认原点为左上角，左移20像素，向上平移5像素  
 child: Transform.translate(  
 offset: Offset(-20.0, -5.0),  
 child: Column(  
 children: <Widget>[  
 Text("飘了飘了"),  
 Text("我也要飘"),  
 ],  
 ),  
 ),  
// child: Text("原來在这里"),  
 ),  
 ),  
 Center(  
 child: DecoratedBox(  
 decoration: BoxDecoration(color: Colors.*lightGreenAccent*),  
 child: Transform.rotate(  
 //旋转90度  
 angle: math.pi / 2, //import 'dart:math' as math;  
 child: Text("支棱起来"),  
 ),  
 ),  
 ),  
 SizedBox(  
 height: 50,  
 ),  
 Row(  
 mainAxisAlignment: MainAxisAlignment.center,  
 children: <Widget>[  
 DecoratedBox(  
 decoration: BoxDecoration(color: Colors.*red*),  
 child: Transform.scale(  
 scale: 1.5, //放大到1.5倍  
 child: Text("膨胀了哈哈哈哈"))),  
 ],  
 ),  
 ],  
 ),

### RotatedBox

RotatedBox和Transform.rotate功能相似，它们都可以对子组件进行旋转变换，但是有一点不同：RotatedBox的变换是在layout阶段，会影响在子组件的位置和大小。

Row(  
 mainAxisAlignment: MainAxisAlignment.center,  
 children: <Widget>[  
 DecoratedBox(  
 decoration: BoxDecoration(color: Colors.*red*),  
 //将Transform.rotate换成RotatedBox  
 child: RotatedBox(  
 quarterTurns: 1, //旋转90度(1/4圈)  
 child: Text("我压死你"),  
 ),  
 ),  
 Text("你压不住我了！！", style: TextStyle(color: Colors.*green*, fontSize: 18.0),)  
 ],  
),

## Container

Container是一个组合类容器，它本身不对应具体的RenderObject，它是DecoratedBox、ConstrainedBox、Transform、Padding、Align等组件组合的一个多功能容器，所以我们只需通过一个Container组件可以实现同时需要装饰、变换、限制的场景。**一句话讲：有了它就可以设置直接设置padding、margin、width、height等等属性！**

Container({

this.alignment,

this.padding, //容器内补白，属于decoration的装饰范围

Color color, // 背景色

Decoration decoration, // 背景装饰

Decoration foregroundDecoration, //前景装饰

double width,//容器的宽度

double height, //容器的高度

BoxConstraints constraints, //容器大小的限制条件

this.margin,//容器外补白，不属于decoration的装饰范围

this.transform, //变换

this.child,

})



## Scaffold、TabBar、底部导航

AppBar({

Key key,

this.leading, //导航栏最左侧Widget，常见为抽屉菜单按钮或返回按钮。

this.automaticallyImplyLeading = true, //如果leading为null，是否自动实现默认的leading按钮

this.title,// 页面标题

this.actions, // 导航栏右侧菜单

this.bottom, // 导航栏底部菜单，通常为Tab按钮组

this.elevation = 4.0, // 导航栏阴影

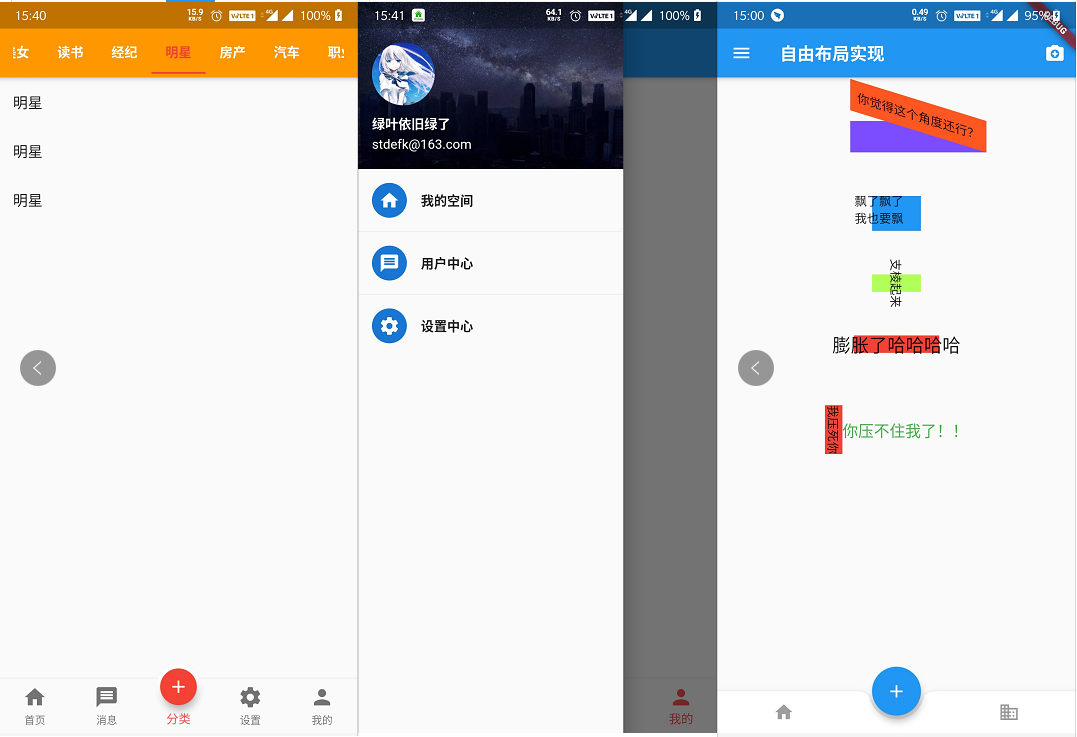
this.centerTitle, //标题是否居中

this.backgroundColor,

... //其它属性见源码注释

})

| **组件名称** | **解释** |
| --- | --- |
| AppBar | 一个导航栏骨架 |
| MyDrawer | 抽屉菜单 |
| BottomNavigationBar | 底部导航栏 |
| FloatingActionButton | 漂浮按钮 |

另外还有TabBar、TabBarView可以快速实现菜单导航，过于简单，就不专门写了，哈哈哈哈哈，直接看效果：

## 剪裁（Clip）

**常用语头像或者缩略图展示：**

import 'package:flutter/material.dart';

class ClipTestRoute extends StatelessWidget {

@override

Widget build(BuildContext context) {

// 头像

Widget avatar = Image.asset("imgs/avatar.png", width: 60.0);

return Center(

child: Column(

children: <Widget>[

avatar, //不剪裁

ClipOval(child: avatar), //剪裁为圆形

ClipRRect( //剪裁为圆角矩形

borderRadius: BorderRadius.circular(5.0),

child: avatar,

),

Row(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

Align(

alignment: Alignment.topLeft,

widthFactor: .5,//宽度设为原来宽度一半，另一半会溢出

child: avatar,

),

Text("你好世界", style: TextStyle(color: Colors.green),)

],

),

Row(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

ClipRect(//将溢出部分剪裁

child: Align(

alignment: Alignment.topLeft,

widthFactor: .5,//宽度设为原来宽度一半

child: avatar,

),

),

Text("你好世界",style: TextStyle(color: Colors.green))

],

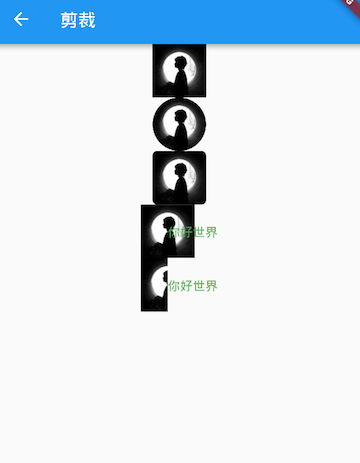
),

],

),

);

}

}

CustomClipper

CustomClipper可以自定义剪裁特定区域：

举个例子：截取图片中部40×30像素的范围

class MyClipper extends CustomClipper<Rect> {

@override

Rect getClip(Size size) => Rect.fromLTWH(10.0, 15.0, 40.0, 30.0);

@override

bool shouldReclip(CustomClipper<Rect> oldClipper) => false;

}

DecoratedBox(

decoration: BoxDecoration(

color: Colors.red

),

child: ClipRect(

clipper: MyClipper(), //使用自定义的clipper

child: avatar

),

)

* getClip()是用于获取剪裁区域的接口，由于图片大小是60×60，我们返回剪裁区域为Rect.fromLTWH(10.0, 15.0, 40.0, 30.0)，即图片中部40×30像素的范围。
* shouldReclip() 接口决定是否重新剪裁。如果在应用中，剪裁区域始终不会发生变化时应该返回false，这样就不会触发重新剪裁，避免不必要的性能开销。如果剪裁区域会发生变化（比如在对剪裁区域执行一个动画），那么变化后应该返回true来重新执行剪裁。

注意：剪裁是在layout完成后的绘制阶段进行的，所以不会影响组件的大小，这和Transform原理是相似的。

# 可滚动组件

说在前面的话：下面从Scrollbar到主轴和纵轴内容为概念性内容，做相对了解和理解即可

当组件内容超过当前显示视口(ViewPort)时，如果没有特殊处理，Flutter则会提示Overflow错误。为此，Flutter提供了多种可滚动组件（Scrollable Widget）用于显示列表和长布局。在本章中，我们先介绍一下常用的可滚动组件（如ListView、GridView等），然后介绍一下ScrollController。可滚动组件都直接或间接包含一个Scrollable组件，因此它们包括一些共同的属性，为了避免重复介绍，我们在此统一介绍一下：

Scrollable({

...

this.axisDirection = AxisDirection.down,

this.controller,

this.physics,

@required this.viewportBuilder, //后面介绍

})

* axisDirection滚动方向。
* physics：此属性接受一个ScrollPhysics类型的对象，它决定可滚动组件如何响应用户操作，比如用户滑动完抬起手指后，继续执行动画；或者滑动到边界时，如何显示。默认情况下，Flutter会根据具体平台分别使用不同的ScrollPhysics对象，应用不同的显示效果，如当滑动到边界时，继续拖动的话，在iOS上会出现弹性效果，而在Android上会出现微光效果。如果你想在所有平台下使用同一种效果，可以显式指定一个固定的ScrollPhysics，Flutter SDK中包含了两个ScrollPhysics的子类，他们可以直接使用：
  + ClampingScrollPhysics：Android下微光效果。
  + BouncingScrollPhysics：iOS下弹性效果。
* controller：此属性接受一个ScrollController对象。ScrollController的主要作用是控制滚动位置和监听滚动事件。默认情况下，Widget树中会有一个默认的PrimaryScrollController，如果子树中的可滚动组件没有显式的指定controller，并且primary属性值为true时（默认就为true），可滚动组件会使用这个默认的PrimaryScrollController。这种机制带来的好处是父组件可以控制子树中可滚动组件的滚动行为，例如，Scaffold正是使用这种机制在iOS中实现了点击导航栏回到顶部的功能。我们将在本章后面“滚动控制”一节详细介绍ScrollController。

### Scrollbar

Scrollbar是一个Material风格的滚动指示器（滚动条），如果要给可滚动组件添加滚动条，只需将Scrollbar作为可滚动组件的任意一个父级组件即可，如：

Scrollbar(

child: SingleChildScrollView(

...

),

);

Scrollbar和CupertinoScrollbar都是通过监听滚动通知来确定滚动条位置的。

### CupertinoScrollbar

CupertinoScrollbar是iOS风格的滚动条，如果你使用的是Scrollbar，那么在iOS平台它会自动切换为CupertinoScrollbar。

### ViewPort视口

在很多布局系统中都有ViewPort的概念，在Flutter中，术语ViewPort（视口），如无特别说明，则是指一个Widget的实际显示区域。例如，一个ListView的显示区域高度是800像素，虽然其列表项总高度可能远远超过800像素，但是其ViewPort仍然是800像素。

### 基于Sliver的延迟构建

通常可滚动组件的子组件可能会非常多、占用的总高度也会非常大；如果要一次性将子组件全部构建出将会非常昂贵！为此，Flutter中提出一个Sliver（中文为“薄片”的意思）概念，如果一个可滚动组件支持Sliver模型，那么该滚动可以将子组件分成好多个“薄片”（Sliver），只有当Sliver出现在视口中时才会去构建它，这种模型也称为“基于Sliver的延迟构建模型”。可滚动组件中有很多都支持基于Sliver的延迟构建模型，如ListView、GridView，但是也有不支持该模型的，如SingleChildScrollView。

### 主轴和纵轴

在可滚动组件的坐标描述中，通常将滚动方向称为主轴，非滚动方向称为纵轴。由于可滚动组件的默认方向一般都是沿垂直方向，所以默认情况下主轴就是指垂直方向，水平方向同理。

## SingleChildScrollView

SingleChildScrollView只能接收一个子组件。通常SingleChildScrollView只应在期望的内容不会超过屏幕太多时使用，这是因为SingleChildScrollView不支持基于Sliver的延迟实例化模型，所以如果预计视口可能包含超出屏幕尺寸太多的内容时，那么使用SingleChildScrollView将会非常昂贵（性能差），此时应该使用一些支持Sliver延迟加载的可滚动组件，如ListView。

定义如下：

SingleChildScrollView({

this.scrollDirection = Axis.vertical, //滚动方向，默认是垂直方向

this.reverse = false,

this.padding,

bool primary,

this.physics,

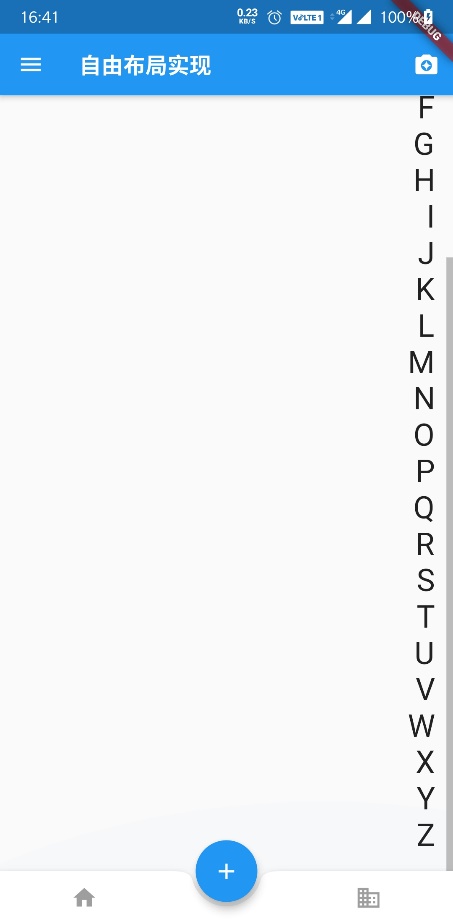
this.controller,

this.child,

})

* reverse：该属性API文档解释是：是否按照阅读方向相反的方向滑动，如：scrollDirection值为Axis.horizontal，如果阅读方向是从左到右(取决于语言环境，阿拉伯语就是从右到左)。reverse为true时，那么滑动方向就是从右往左。其实此属性本质上是决定可滚动组件的初始滚动位置是在“头”还是“尾”，取false时，初始滚动位置在“头”，反之则在“尾”，读者可以自己试验。
* primary：指是否使用widget树中默认的PrimaryScrollController；当滑动方向为垂直方向（scrollDirection值为Axis.vertical）并且没有指定controller时，primary默认为true.

body: Scrollbar(  
 // 显示进度条  
 child: SingleChildScrollView(  
 padding: EdgeInsets.all(16.0),  
 child: Container(  
 width: double.*infinity*,  
 child: Column(  
 crossAxisAlignment: CrossAxisAlignment.end ,  
 //动态创建一个List<Widget>  
 children: "ABCDEFGHIJKLMNOPQRSTUVWXYZ".split("").map((c) => Text(  
 c,  
 textScaleFactor: 2.0, //每一个字母都用一个Text显示,字体为原来的两倍  
 )).toList(),  
 ),  
 )  
 ),  
),



## ListView（最常用之一）

ListView({

...

//可滚动widget公共参数

Axis scrollDirection = Axis.vertical,

bool reverse = false,

ScrollController controller,

bool primary,

ScrollPhysics physics,

EdgeInsetsGeometry padding,

//ListView各个构造函数的共同参数

double itemExtent,

bool shrinkWrap = false,

bool addAutomaticKeepAlives = true,

bool addRepaintBoundaries = true,

double cacheExtent,

//子widget列表

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

})

* itemExtent：该参数如果不为null，则会强制children的“长度”为itemExtent的值；这里的“长度”是指滚动方向上子组件的长度，也就是说如果滚动方向是垂直方向，则itemExtent代表子组件的高度；如果滚动方向为水平方向，则itemExtent就代表子组件的宽度。在ListView中，指定itemExtent比让子组件自己决定自身长度会更高效，这是因为指定itemExtent后，滚动系统可以提前知道列表的长度，而无需每次构建子组件时都去再计算一下，尤其是在滚动位置频繁变化时（滚动系统需要频繁去计算列表高度）。
* shrinkWrap：该属性表示是否根据子组件的总长度来设置ListView的长度，默认值为false 。默认情况下，ListView的会在滚动方向尽可能多的占用空间。当ListView在一个无边界(滚动方向上)的容器中时，shrinkWrap必须为true。
* addAutomaticKeepAlives：该属性表示是否将列表项（子组件）包裹在AutomaticKeepAlive 组件中；典型地，在一个懒加载列表中，如果将列表项包裹在AutomaticKeepAlive中，在该列表项滑出视口时它也不会被GC（垃圾回收），它会使用KeepAliveNotification来保存其状态。如果列表项自己维护其KeepAlive状态，那么此参数必须置为false。
* addRepaintBoundaries：该属性表示是否将列表项（子组件）包裹在RepaintBoundary组件中。当可滚动组件滚动时，将列表项包裹在RepaintBoundary中可以避免列表项重绘，但是当列表项重绘的开销非常小（如一个颜色块，或者一个较短的文本）时，不添加RepaintBoundary反而会更高效。和addAutomaticKeepAlive一样，如果列表项自己维护其KeepAlive状态，那么此参数必须置为false。

**先来一个简单直白好吃的例子：默认构造函数构建的ListView没有应用基于Sliver的懒加载模型：（除非列表展示数据少的时候可以快速使用，一般不建议使用）**

ListView(

shrinkWrap: true,

padding: const EdgeInsets.all(20.0),

children: <Widget>[

const Text('I\'m dedicating every day to you'),

const Text('Domestic life was never quite my style'),

const Text('When you smile, you knock me out, I fall apart'),

const Text('And I thought I was so smart'),

],

);

ListView.builder

适合列表项比较多（或者无限）的情况，因为只有当子组件真正显示的时候才会被创建，也就说通过该构造函数创建的ListView是支持基于Sliver的懒加载模型的。

ListView.builder({

// ListView公共参数已省略

...

@required IndexedWidgetBuilder itemBuilder,

int itemCount,

...

})

* itemBuilder：它是列表项的构建器，类型为IndexedWidgetBuilder，返回值为一个widget。当列表滚动到具体的index位置时，会调用该构建器构建列表项。
* itemCount：列表项的数量，如果为null，则为无限列表。

**上代码：**

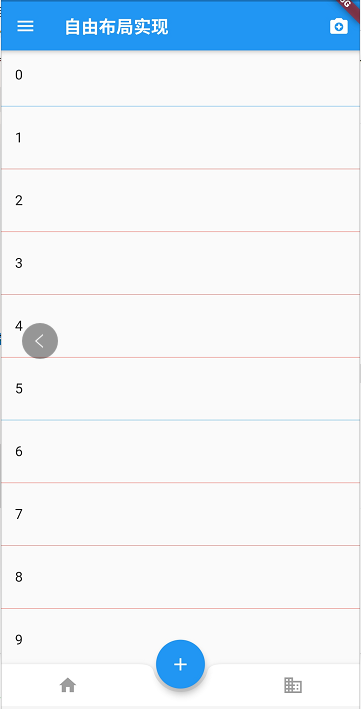
ListView.builder(  
 itemCount: 100,  
 itemExtent: 50.0, //强制高度为50.0  
 itemBuilder: (BuildContext context, int index) {  
 return ListTile(title: Text("$index"));  
 }  
),

ListView.separated

ListView.separated可以在生成的列表项之间添加一个分割组件，它比ListView.builder多了一个separatorBuilder参数，该参数是一个分割组件生成器。

**代码走起：**

ListView.separated(  
 itemCount: 100,  
 //列表项构造器  
 itemBuilder: (BuildContext context, int index) {  
 return ListTile(title: Text("$index"));  
 },  
 //分割器构造器  
 separatorBuilder: (BuildContext context, int index) {  
 return index%5==0?Divider(color: Colors.*blue*,):Divider(color: Colors.*red*,);  
 },  
),

效果配上：

## GridView

GridView({

Axis scrollDirection = Axis.vertical,

bool reverse = false,

ScrollController controller,

bool primary,

ScrollPhysics physics,

bool shrinkWrap = false,

EdgeInsetsGeometry padding,

@required SliverGridDelegate gridDelegate, //控制子widget layout的委托

bool addAutomaticKeepAlives = true,

bool addRepaintBoundaries = true,

double cacheExtent,

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

})

基本和ListView一致，单独看下gridDelegate参数，它的类型是SliverGridDelegate，用来控制GridView子组件如何排列(layout)。Flutter中提供了两个SliverGridDelegate的子类SliverGridDelegateWithFixedCrossAxisCount和SliverGridDelegateWithMaxCrossAxisExtent，我们可以直接使用

SliverGridDelegateWithFixedCrossAxisCount

SliverGridDelegateWithFixedCrossAxisCount({

@required double crossAxisCount,

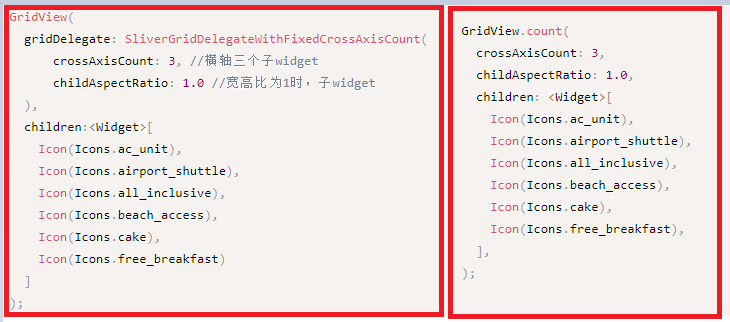
double mainAxisSpacing = 0.0,

double crossAxisSpacing = 0.0,

double childAspectRatio = 1.0,

})

* crossAxisCount：横轴子元素的数量。此属性值确定后子元素在横轴的长度就确定了，即ViewPort横轴长度除以crossAxisCount的商。
* mainAxisSpacing：主轴方向的间距。
* crossAxisSpacing：横轴方向子元素的间距。
* childAspectRatio：子元素在横轴长度和主轴长度的比例。由于crossAxisCount指定后，子元素横轴长度就确定了，然后通过此参数值就可以确定子元素在主轴的长度。

以上两种方案都实现了下面的效果，所以GirdView.count()就不单独说了！！！

SliverGridDelegateWithMaxCrossAxisExtent

**该子类实现了一个横轴子元素为固定最大长度的layout算法，其构造函数为：**

SliverGridDelegateWithMaxCrossAxisExtent({

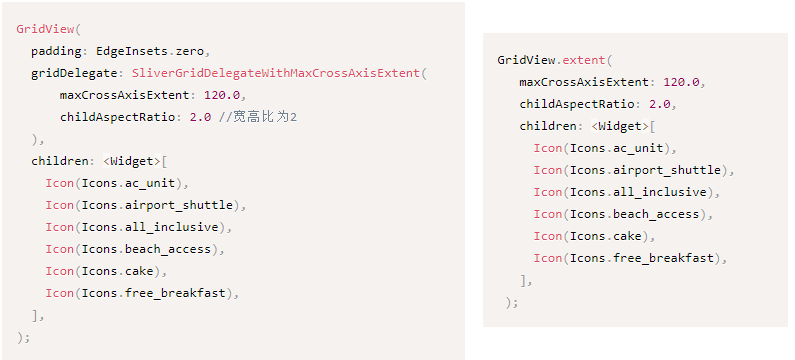
double maxCrossAxisExtent,

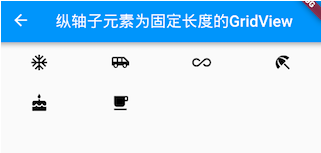
double mainAxisSpacing = 0.0,

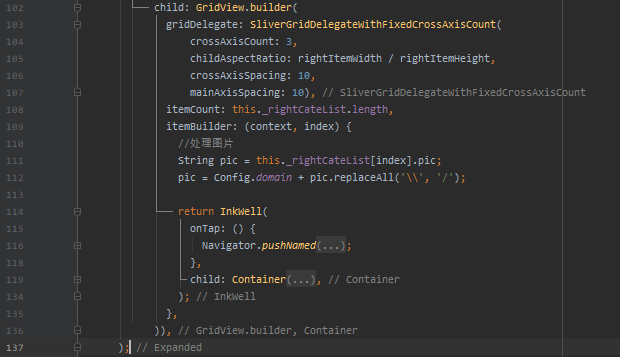
double crossAxisSpacing = 0.0,

double childAspectRatio = 1.0,

})

maxCrossAxisExtent为子元素在横轴上的最大长度，之所以是“最大”长度，是**因为横轴方向每个子元素的长度仍然是等分的**，举个例子，如果ViewPort的横轴长度是450，那么当maxCrossAxisExtent的值在区间[450/4，450/3)内的话，子元素最终实际长度都为112.5，而childAspectRatio所指的子元素横轴和主轴的长度比为**最终的长度比**。其它参数和SliverGridDelegateWithFixedCrossAxisCount相同。

以上两种方案都实现了下面的效果，所以GirdView.extent就不单独说了！！！

**再上一段代码：**

**是的，你想的对，一般用GridView.builder就是了！**

## CustomScrollView

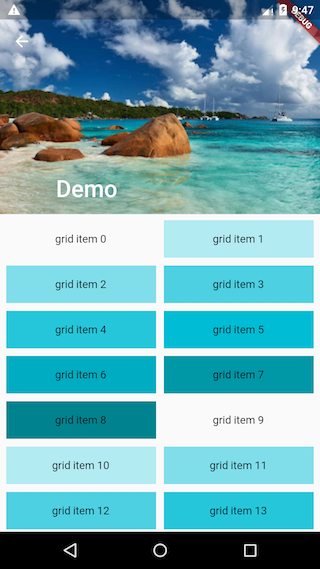
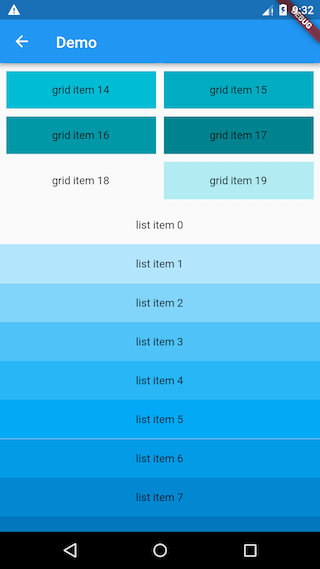
CustomScrollView是可以使用Sliver来自定义滚动模型（效果）的组件。它可以包含多种滚动模型，举个例子，假设有一个页面，顶部需要一个GridView，底部需要一个ListView，而要求整个页面的滑动效果是统一的，即它们看起来是一个整体。如果使用GridView+ListView来实现的话，就不能保证一致的滑动效果，因为它们的滚动效果是分离的，所以这时就需要一个"胶水"，把这些彼此独立的可滚动组件"粘"起来，而CustomScrollView的功能就相当于“胶水”。

**上代码再说话：**

import 'package:flutter/material.dart';  
  
class CustomScrollViewTestRoute extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 //因为本路由没有使用Scaffold，为了让子级Widget(如Text)使用  
 //Material Design 默认的样式风格,我们使用Material作为本路由的根。  
 return Material(  
 child: CustomScrollView(  
 slivers: <Widget>[  
 //AppBar，包含一个导航栏  
 SliverAppBar(  
 pinned: true,  
 expandedHeight: 250.0,  
 flexibleSpace: FlexibleSpaceBar(  
 title: const Text('Demo'),  
 background: Image.asset(  
 "./images/avatar.png", fit: BoxFit.cover,),  
 ),  
 ),  
  
 SliverPadding(  
 padding: const EdgeInsets.all(8.0),  
 sliver: new SliverGrid( //Grid  
 gridDelegate: new SliverGridDelegateWithFixedCrossAxisCount(  
 crossAxisCount: 2, //Grid按两列显示  
 mainAxisSpacing: 10.0,  
 crossAxisSpacing: 10.0,  
 childAspectRatio: 4.0,  
 ),  
 delegate: new SliverChildBuilderDelegate(  
 (BuildContext context, int index) {  
 //创建子widget   
 return new Container(  
 alignment: Alignment.*center*,  
 color: Colors.*cyan*[100 \* (index % 9)],  
 child: new Text('grid item $index'),  
 );  
 },  
 childCount: 20,  
 ),  
 ),  
 ),  
 //List  
 new SliverFixedExtentList(  
 itemExtent: 50.0,  
 delegate: new SliverChildBuilderDelegate(  
 (BuildContext context, int index) {  
 //创建列表项   
 return new Container(  
 alignment: Alignment.*center*,  
 color: Colors.*lightBlue*[100 \* (index % 9)],  
 child: new Text('list item $index'),  
 );  
 },  
 childCount: 50 //50个列表项  
 ),  
 ),  
 ],  
 ),  
 );  
 }  
}

**代码分为三部分：**

* 头部SliverAppBar：SliverAppBar对应AppBar，两者不同之处在于SliverAppBar可以集成到CustomScrollView。SliverAppBar可以结合FlexibleSpaceBar实现Material Design中头部伸缩的模型，具体效果，读者可以运行该示例查看。
* 中间的SliverGrid：它用SliverPadding包裹以给SliverGrid添加补白。SliverGrid是一个两列，宽高比为4的网格，它有20个子组件。
* 底部SliverFixedExtentList：它是一个所有子元素高度都为50像素的列表。

**上图：**

## 滚动监听及控制ScrollController

写在前面：此部分内容写记录了常用功能. ScrollMetrics有一些其他属性，包括滚动监听，定位，以及还原滚动位置等操作，可自行参考API

ScrollController({

double initialScrollOffset = 0.0, //初始滚动位置

this.keepScrollOffset = true,//是否保存滚动位置

...

})

ScrollController常用的属性和方法：

* offset：可滚动组件当前的滚动位置。
* jumpTo(double offset)、animateTo(double offset,...)：这两个方法用于跳转到指定的位置，它们不同之处在于，后者在跳转时会执行一个动画，而前者不会。

滚动监听

ScrollController间接继承自Listenable，我们可以根据ScrollController来监听滚动事件，如：

controller.addListener(()=>print(controller.offset))

**上代码：**

class ScrollControllerTestRoute extends StatefulWidget {

@override

ScrollControllerTestRouteState createState() {

return new ScrollControllerTestRouteState();

}

}

class ScrollControllerTestRouteState extends State<ScrollControllerTestRoute> {

//定义自己的controller

ScrollController \_controller = new ScrollController();

bool showToTopBtn = false; //是否显示“返回到顶部”按钮

@override

void initState() {

super.initState();

//监听滚动事件，打印滚动位置

\_controller.addListener(() {

print(\_controller.offset); //打印滚动位置

if (\_controller.offset < 1000 && showToTopBtn) {

setState(() {

showToTopBtn = false;

});

} else if (\_controller.offset >= 1000 && showToTopBtn == false) {

setState(() {

showToTopBtn = true;

});

}

});

}

@override

void dispose() {

//为了避免内存泄露，销毁时需要调用\_controller.dispose

\_controller.dispose();

super.dispose();

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text("滚动控制")),

body: Scrollbar(

child: ListView.builder(

itemCount: 100,

itemExtent: 50.0, //列表项高度固定时，显式指定高度是一个好习惯(性能消耗小)

controller: \_controller,

itemBuilder: (context, index) {

return ListTile(title: Text("$index"),);

}

),

),

floatingActionButton: !showToTopBtn ? null : FloatingActionButton(

child: Icon(Icons.arrow\_upward),

onPressed: () {

//返回到顶部时执行动画

\_controller.animateTo(.0,

duration: Duration(milliseconds: 200),

curve: Curves.ease

);

}

),

);

}

}

**再上代码：**

import 'package:flutter/material.dart';

class ScrollNotificationTestRoute extends StatefulWidget {

@override

\_ScrollNotificationTestRouteState createState() =>

new \_ScrollNotificationTestRouteState();

}

class \_ScrollNotificationTestRouteState

extends State<ScrollNotificationTestRoute> {

String \_progress = "0%"; //保存进度百分比

@override

Widget build(BuildContext context) {

return Scrollbar( //进度条

// 监听滚动通知

child: NotificationListener<ScrollNotification>(

onNotification: (ScrollNotification notification) {

double progress = notification.metrics.pixels /

notification.metrics.maxScrollExtent;

//重新构建

setState(() {

\_progress = "${(progress \* 100).toInt()}%";

});

print("BottomEdge: ${notification.metrics.extentAfter == 0}");

//return true; //放开此行注释后，进度条将失效

},

child: Stack(

alignment: Alignment.center,

children: <Widget>[

ListView.builder(

itemCount: 100,

itemExtent: 50.0,

itemBuilder: (context, index) {

return ListTile(title: Text("$index"));

}

),

CircleAvatar( //显示进度百分比

radius: 30.0,

child: Text(\_progress),

backgroundColor: Colors.black54,

)

],

),

),

);

}

}

在接收到滚动事件时，参数类型为ScrollNotification，它包括一个metrics属性，它的类型是ScrollMetrics，该属性包含当前ViewPort及滚动位置等信息：

* pixels：当前滚动位置。
* maxScrollExtent：最大可滚动长度。
* extentBefore：滑出ViewPort顶部的长度；此示例中相当于顶部滑出屏幕上方的列表长度。
* extentInside：ViewPort内部长度；此示例中屏幕显示的列表部分的长度。
* extentAfter：列表中未滑入ViewPort部分的长度；此示例中列表底部未显示到屏幕范围部分的长度。
* atEdge：是否滑到了可滚动组件的边界（此示例中相当于列表顶或底部）。

# 功能型组件：

## 导航返回拦截（WillPopScope）

WillPopScope的默认构造函数：

const WillPopScope({

...

@required WillPopCallback onWillPop,

@required Widget child

})

onWillPop是一个回调函数，当用户点击返回按钮时被调用（包括导航返回按钮以及Android物理返回按钮）。该返回需要返回一个Future对象，如果返回的Future最终值为false时，则当前路由不会出栈（不会返回）；最终值为true，当前路由出栈退出。我们需要提供这个回调来决定是否退出。

举个例子：

为了防止用户误触返回键退出，我们拦截返回事件。当用户在1秒内点击两次返回按钮时，则退出；如果间隔超过1秒则不退出，并重新记时。代码如下：

import 'package:flutter/material.dart';  
  
class WillPopScopeTestRoute extends StatefulWidget {  
 @override  
 WillPopScopeTestRouteState createState() {  
 return new WillPopScopeTestRouteState();  
 }  
}  
  
class WillPopScopeTestRouteState extends State<WillPopScopeTestRoute> {  
 DateTime \_lastPressedAt; //上次点击时间  
  
 @override  
 Widget build(BuildContext context) {  
 return new WillPopScope(  
 onWillPop: () async {  
 if (\_lastPressedAt == null ||  
 DateTime.now().difference(\_lastPressedAt) > Duration(seconds: 1)) {  
 //两次点击间隔超过1秒则重新计时  
 \_lastPressedAt = DateTime.now();  
 return false;  
 }  
 return true;  
 },  
 child: Container(  
 alignment: Alignment.*center*,  
 child: Text("1秒内连续按两次返回键退出"),  
 )  
 );  
 }  
}

## 数据共享（InheritedWidget）

InheritedWidget 是Flutter 中非常重要的一个功能型组件，他提供了一种数据在widget树中从上到下的传递、共享的方式。比如我们在应用的根Widget中通过inheritedWidget共享了一个数据，那么我们便可以在任意子widget中来获取该共享数据！这个特性正如Flutter SDK中通过InheritedWidget来共享应用主题（Theme）和Locale（当前语言环境）信息的。

InheritedWidget 和React中的context功能类似，和逐级传递数据相比，它们能实现组件跨级传递数据。InheritedWidget的在 Widget树中数据传递的方向是从上到下的，这和通知Notification（状态改变通知）的传递方向正好相反。

didChangeDependencies

didChangeDependencies会在“依赖”发生变化时被Flutter Framework调用。而这个“依赖”指的就是子widget是否使用了父widget中InheritedWidget的数据。

**下面我们看一下之前“计数器”示例应用程序的InheritedWidget版本。需要说明的是，本示例主要是为了演示InheritedWidget的功能特性，并不是计数器的推荐实现方式。**

1. 首先，我们通过继承InheritedWidget，将当前计数器点击次数保存在ShareDataWidget的data属性中：

import 'package:flutter/material.dart';  
  
class ShareDataWidget extends InheritedWidget {  
 ShareDataWidget({@required this.data, Widget child}) : super(child: child);  
  
 final int data; //需要在子树中共享的数据，保存点击次数  
  
 //定义一个便捷方法，方便子树中的widget获取共享数据  
 static ShareDataWidget *of*(BuildContext context) {  
 return context.dependOnInheritedWidgetOfExactType<ShareDataWidget>();  
 }  
  
 //该回调决定当data发生变化时，是否通知子树中依赖data的Widget  
 @override  
 bool updateShouldNotify(ShareDataWidget old) {  
 //如果返回true，则子树中依赖(build函数中有调用)本widget的子widget的`state.didChangeDependencies`会被调用  
 return old.data != data;  
 }  
}

1. 然后我们实现一个子组件\_TestWidget，在其build方法中引用ShareDataWidget中的数据。同时，在其didChangeDependencies() 回调中打印日志：

class \_TestWidget extends StatefulWidget {  
 @override  
 \_\_TestWidgetState createState() => new \_\_TestWidgetState();  
}  
  
class \_\_TestWidgetState extends State<\_TestWidget> {  
 @override  
 Widget build(BuildContext context) {  
 //使用InheritedWidget中的共享数据  
 return Text(ShareDataWidget.*of*(context).data.toString());  
 }  
  
 @override  
 void didChangeDependencies() {  
 super.didChangeDependencies();  
 //父或祖先widget中的InheritedWidget改变(updateShouldNotify返回true)时会被调用。  
 //如果build中没有依赖InheritedWidget，则此回调不会被调用。  
 print("Dependencies change");  
 }  
}

1. 最后，我们创建一个按钮，每点击一次，就将ShareDataWidget的值自增：

class InheritedWidgetTestRoute extends StatefulWidget {  
 @override  
 \_InheritedWidgetTestRouteState createState() => new \_InheritedWidgetTestRouteState();  
}  
  
class \_InheritedWidgetTestRouteState extends State<InheritedWidgetTestRoute> {  
 int count = 0;  
  
 @override  
 Widget build(BuildContext context) {  
 return Center(  
 child: ShareDataWidget( //使用ShareDataWidget  
 data: count,  
 child: Column(  
 mainAxisAlignment: MainAxisAlignment.center,  
 children: <Widget>[  
 Padding(  
 padding: const EdgeInsets.only(bottom: 20.0),  
 child: \_TestWidget(),//子widget中依赖ShareDataWidget  
 ),  
 RaisedButton(  
 child: Text("Increment"),  
 //每点击一次，将count自增，然后重新build,ShareDataWidget的data将被更新   
 onPressed: () => setState(() => ++count),  
 )  
 ],  
 ),  
 ),  
 );  
 }  
}

课件依赖发生变化后，其didChangeDependencies()会被调用。值得注意的是，**如果\_TestWidget的build方法中没有使用ShareDataWidget的数据，那么它的didChangeDependencies()将不会被调用，因为它并没有依赖ShareDataWidget。**例如，我们将\_\_TestWidgetState代码改为下面这样，didChangeDependencies()将不会被调用：

class \_\_TestWidgetState extends State<\_TestWidget> {  
 @override  
 Widget build(BuildContext context) {  
 // 使用InheritedWidget中的共享数据  
// return Text(ShareDataWidget.of(context).data.toString());  
 return Text("text");  
 }  
  
 @override  
 void didChangeDependencies() {  
 super.didChangeDependencies();  
 // build方法中没有依赖InheritedWidget，此回调不会被调用。  
 print("Dependencies change");  
 }  
}

上面的代码中，我们将build()方法中依赖ShareDataWidget的代码注释掉了，然后返回一个固定Text，这样一来，当点击Increment按钮后，ShareDataWidget的data虽然发生变化，但由于\_\_TestWidgetState并未依赖ShareDataWidget，所以\_\_TestWidgetState的didChangeDependencies方法不会被调用。

didChangeDependencies()中适合做什么？

一般来说，子widget很少会重写此方法，因为在依赖改变后framework也都会调用build()方法。但是，如果你需要在依赖改变后执行一些昂贵的操作，比如网络请求，这时最好的方式就是在此方法中执行，这样可以避免每次build()都执行这些昂贵操作。

深入了解InHeritedWidget

如果我们只想在\_\_TestWidgetState中引用ShareDataWidget数据，但却不希望在ShareDataWidget发生变化时调用\_\_TestWidgetState的didChangeDependencies()方法应该怎么办？其实答案很简单，我们只需要将ShareDataWidget.of()的实现改一下即可：

//定义一个便捷方法，方便子树中的widget获取共享数据  
static ShareDataWidget of(BuildContext context) {  
//return context.dependOnInheritedWidgetOfExactType<ShareDataWidget>();  
return context.getElementForInheritedWidgetOfExactType<ShareDataWidget>().widget;  
}

唯一的改动就是获取ShareDataWidget对象的方式，把dependOnInheritedWidgetOfExactType()方法换成了context.getElementForInheritedWidgetOfExactType<ShareDataWidget>().widget，那么他们到底有什么区别呢，我们看一下这两个方法的源码

@override

InheritedElement getElementForInheritedWidgetOfExactType<T extends InheritedWidget>() {

assert(\_debugCheckStateIsActiveForAncestorLookup());

final InheritedElement ancestor = \_inheritedWidgets == null ? null : \_inheritedWidgets[T];

return ancestor;

}

@override

InheritedWidget dependOnInheritedWidgetOfExactType({ Object aspect }) {

assert(\_debugCheckStateIsActiveForAncestorLookup());

final InheritedElement ancestor = \_inheritedWidgets == null ? null : \_inheritedWidgets[T];

//多出的部分

if (ancestor != null) {

assert(ancestor is InheritedElement);

return dependOnInheritedElement(ancestor, aspect: aspect) as T;

}

\_hadUnsatisfiedDependencies = true;

return null;

}

我们可以看到，dependOnInheritedWidgetOfExactType()比getElementForInheritedWidgetOfExactType()多调了dependOnInheritedElement方法，dependOnInheritedElement源码如下：

@override

InheritedWidget dependOnInheritedElement(InheritedElement ancestor, { Object aspect }) {

assert(ancestor != null);

\_dependencies ??= HashSet<InheritedElement>();

\_dependencies.add(ancestor);

ancestor.updateDependencies(this, aspect);

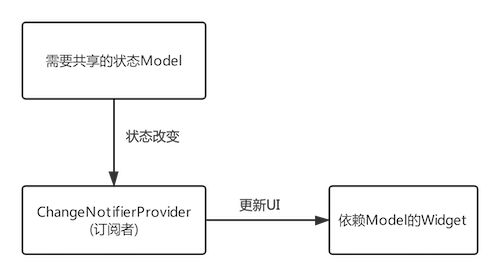
return ancestor.widget;

}

可以看到dependOnInheritedElement方法中主要是注册了依赖关系！看到这里也就清晰了，**调用dependOnInheritedWidgetOfExactType()和getElementForInheritedWidgetOfExactType()的区别就是前者会注册依赖关系，而后者不会**，所以在调用dependOnInheritedWidgetOfExactType()时，InheritedWidget和依赖它的子孙组件关系便完成了注册，之后当InheritedWidget发生变化时，就会更新依赖它的子孙组件，也就是会调这些子孙组件的didChangeDependencies()方法和build()方法。而当调用的是getElementForInheritedWidgetOfExactType()时，由于没有注册依赖关系，所以之后当InheritedWidget发生变化时，就不会更新相应的子孙Widget。

注意：如果将上面示例中ShareDataWidget.of()方法实现改成调用getElementForInheritedWidgetOfExactType()，运行示例后，点击"Increment"按钮，会发现\_\_TestWidgetState的didChangeDependencies()方法确实不会再被调用，但是其build()仍然会被调用！造成这个的原因其实是，点击"Increment"按钮后，会调用\_InheritedWidgetTestRouteState的setState()方法，此时会重新构建整个页面，由于示例中，\_\_TestWidget 并没有任何缓存，所以它也都会被重新构建，所以也会调用build()方法。

## 跨组件状态共享（Provider）

Provider的原理图：

Model变化后会自动通知ChangeNotifierProvider（订阅者），ChangeNotifierProvider内部会重新构建InheritedWidget，而依赖该InheritedWidget的子孙Widget就会更新。

关于原理详解，可参阅《Flutter实战》第7章第3节：<https://book.flutterchina.club/chapter7/provider.html>

Provider的使用：

1.安装：

dependencies:

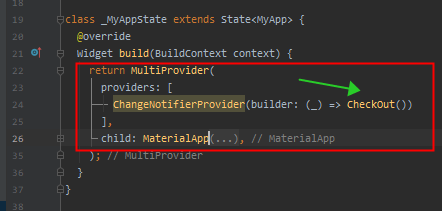
**provider: ^4.1.1**

2.引入

**import** 'package:provider/provider.dart';

3.创建需要共享的类组件

import 'package:flutter/material.dart';  
  
class CheckOut with ChangeNotifier {  
 List \_checkOutListData = []; //购物车数据  
 List get checkOutListData => this.\_checkOutListData;  
  
 changeCheckOutListData(data){  
 this.\_checkOutListData=data;  
 notifyListeners();//数据更新后进行通知  
 }  
  
}

4.在main.Dart中引用并注册：

5.在需要数据共享的页面引入、定义、使用：

引入provider包以及定义的数据共享类组件

import 'package:provider/provider.dart';  
import '../provider/CheckOut.dart';

定义并使用：

## 颜色和主题

颜色：

Flutter中的Color类中颜色以一个int值保存，我们知道显示器颜色是由红、绿、蓝三基色组成，每种颜色占8比特，存储结构如下：

| **Bit（位）** | **颜色** |
| --- | --- |
| 0-7 | 蓝色 |
| 8-15 | 绿色 |
| 16-23 | 红色 |
| 24-31 | Alpha (不透明度) |

****·如何将颜色字符串转成Color对象****

如Web开发中的色值通常是一个字符串如"#dc380d"，它是一个RGB值，我们可以通过下面这些方法将其转为Color类：

Color(0xffdc380d); //如果颜色固定可以直接使用整数值

//颜色是一个字符串变量

var c = "dc380d";

Color(int.parse(c,radix:16)|0xFF000000) //通过位运算符将Alpha设置为FF

Color(int.parse(c,radix:16)).withAlpha(255) //通过方法将Alpha设置为FF

·颜色亮度

假如，我们要实现一个背景颜色和Title可以自定义的导航栏，并且背景色为深色时我们应该让Title显示为浅色；背景色为浅色时，Title显示为深色。要实现这个功能，我们就需要来计算背景色的亮度，然后动态来确定Title的颜色。Color类中提供了一个computeLuminance()方法，它可以返回一个[0-1]的一个值，数字越大颜色就越浅，我们可以根据它来动态确定Title的颜色，下面是导航栏NavBar的简单实现：

import 'package:flutter/material.dart';  
  
class NavBar extends StatelessWidget {  
 final String title;  
 final Color color; //背景颜色  
  
 NavBar({  
 Key key,  
 this.color,  
 this.title,  
 });  
  
 @override  
 Widget build(BuildContext context) {  
 return Container(  
 constraints: BoxConstraints(  
 minHeight: 52,  
 minWidth: double.*infinity*,  
 ),  
 decoration: BoxDecoration(  
 color: color,  
 boxShadow: [  
 //阴影  
 BoxShadow(  
 color: Colors.*black26*,  
 offset: Offset(0, 3),  
 blurRadius: 3,  
 ),  
 ],  
 ),  
 child: Text(  
 title,  
 style: TextStyle(  
 fontWeight: FontWeight.*bold*,  
 //根据背景色亮度来确定Title颜色  
 color: color.computeLuminance() < 0.5 ? Colors.*white* : Colors.*black*,  
 ),  
 ),  
 alignment: Alignment.*center*,  
 );  
 }  
}

使用封装好的NavBar：

Column(  
 children: <Widget>[  
 //背景为蓝色，则title自动为白色  
 NavBar(color: Colors.blue, title: "标题"),  
 //背景为白色，则title自动为黑色  
 NavBar(color: Colors.white, title: "标题"),  
 ]  
)

 效果：

·MaterialColor

MaterialColor是实现Material Design中的颜色的类，它包含一种颜色的10个级别的渐变色。MaterialColor通过"[]"运算符的索引值来代表颜色的深度，有效的索引有：50，100，200，…，900，数字越大，颜色越深。MaterialColor的默认值为索引等于500的颜色。举个例子，Colors.blue是预定义的一个MaterialColor类对象，定义如下：

static const MaterialColor blue = MaterialColor(  
 \_bluePrimaryValue,  
 <int, Color>{  
 50: Color(0xFFE3F2FD),  
 100: Color(0xFFBBDEFB),  
 200: Color(0xFF90CAF9),  
 300: Color(0xFF64B5F6),  
 400: Color(0xFF42A5F5),  
 500: Color(\_bluePrimaryValue),  
 600: Color(0xFF1E88E5),  
 700: Color(0xFF1976D2),  
 800: Color(0xFF1565C0),  
 900: Color(0xFF0D47A1),  
 },  
);  
static const int \_bluePrimaryValue = 0xFF2196F3;

Colors.blue[50]到Colors.blue[100]的色值从浅蓝到深蓝渐变，效果如图：

Theme

Theme组件可以为Material APP定义主题数据（ThemeData）。Material组件库里很多组件都使用了主题数据，如导航栏颜色、标题字体、Icon样式等。Theme内会使用InheritedWidget来为其子树共享样式数据。

·ThemeData

ThemeData用于保存是Material 组件库的主题数据，Material组件需要遵守相应的设计规范，而这些规范可自定义部分都定义在ThemeData中了，所以我们可以通过ThemeData来自定义应用主题。在子组件中，我们可以通过Theme.of方法来获取当前的ThemeData。

注意：Material Design 设计规范中有些是不能自定义的，如导航栏高度，ThemeData只包含了可自定义部分。

ThemeData部分数据定义：

ThemeData({

Brightness brightness, //深色还是浅色

MaterialColor primarySwatch, //主题颜色样本，见下面介绍

Color primaryColor, //主色，决定导航栏颜色

Color accentColor, //次级色，决定大多数Widget的颜色，如进度条、开关等。

Color cardColor, //卡片颜色

Color dividerColor, //分割线颜色

ButtonThemeData buttonTheme, //按钮主题

Color cursorColor, //输入框光标颜色

Color dialogBackgroundColor,//对话框背景颜色

String fontFamily, //文字字体

TextTheme textTheme,// 字体主题，包括标题、body等文字样式

IconThemeData iconTheme, // Icon的默认样式

TargetPlatform platform, //指定平台，应用特定平台控件风格

...

})

上面只是ThemeData的一小部分属性，完整的数据定义读者可以查看SDK。上面属性中需要说明的是primarySwatch，它是主题颜色的一个"样本色"，通过这个样本色可以在一些条件下生成一些其它的属性，例如，如果没有指定primaryColor，并且当前主题不是深色主题，那么primaryColor就会默认为primarySwatch指定的颜色，还有一些相似的属性如accentColor 、indicatorColor等也会受primarySwatch影响。

写在后面的提示：全局设置主题后，可在单页和所需要的地方进行单独设置并覆盖全局主题影响。

# 异步UI更新(FutureBuilder、StreamBuilder)

作用：flutter专门提供此方法来保证快速实现动态更新UI操作，比如数据加载等待，进度显示等功能。

FutureBuilder

FutureBuilder会依赖一个Future，它会根据所依赖的Future的状态来动态构建自身。FutureBuilder构造函数：

FutureBuilder({

this.future,

this.initialData,

@required this.builder,

})

* future：FutureBuilder依赖的Future，通常是一个异步耗时任务。
* initialData：初始数据，用户设置默认数据。
* builder：Widget构建器；该构建器会在Future执行的不同阶段被多次调用，构建器签名如下：

Function (BuildContext context, AsyncSnapshot snapshot)

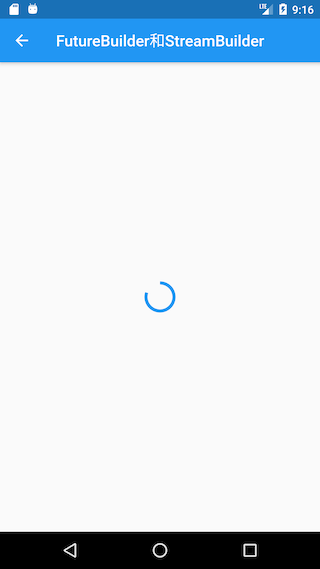
* snapshot会包含当前异步任务的状态信息及结果信息 ，比如我们可以通过snapshot.connectionState获取异步任务的状态信息、通过snapshot.hasError判断异步任务是否有错误等等，完整的定义读者可以查看AsyncSnapshot类定义。

另外，FutureBuilder的builder函数签名和StreamBuilder的builder是相同的。

**一个例子说明怎么使用：**

//模拟数据请求  
Future<String> mockNetworkData() async {  
 return Future.delayed(Duration(seconds: 2), () => "我是从互联网上获取的数据");  
}  
//FutureBuilder  
Widget build(BuildContext context) {  
 return Center(  
 child: FutureBuilder<String>(  
 future: mockNetworkData(), //请求是否结束  
 builder: (BuildContext context, AsyncSnapshot snapshot) {  
 // 请求已结束  
 if (snapshot.connectionState == ConnectionState.done) {  
 if (snapshot.hasError) {  
 // 请求失败，显示错误  
 return Text("Error: ${snapshot.error}");  
 } else {  
 // 请求成功，显示数据  
 return Text("Contents: ${snapshot.data}");  
 }  
 } else {  
 // 请求未结束，显示loading  
 return CircularProgressIndicator();  
 }  
 },  
 ),  
 );  
}

看下效果：



上面代码中我们在builder中根据当前异步任务状态ConnectionState来返回不同的widget。ConnectionState是一个枚举类，定义如下：

enum ConnectionState {

/// 当前没有异步任务，比如[FutureBuilder]的[future]为null时

none,

/// 异步任务处于等待状态

waiting,

/// Stream处于激活状态（流上已经有数据传递了），对于FutureBuilder没有该状态。

active,

/// 异步任务已经终止.

done,

}

注意，ConnectionState.active只在StreamBuilder中才会出现。

StreamBuilder

在Dart中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，它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。StreamBuilder正是用于配合Stream来展示流上事件（数据）变化的UI组件。看一下StreamBuilder的默认构造函数：

StreamBuilder({

Key key,

this.initialData,

Stream<T> stream,

@required this.builder,

})

**凡是UI会依赖多个异步数据而发生变化的场景都可以使用StreamBuilder。直接上代码：**

//创建一个计时器的示例：每隔1秒，计数加1。这里，我们使用Stream来实现每隔一秒生成一个数字:  
Stream<int> counter() {  
 return Stream.periodic(Duration(seconds: 1), (i) {  
 return i;  
 });  
}  
//StreamBuilder使用  
Widget build(BuildContext context) {  
 return StreamBuilder<int>(  
 stream: counter(), //异步结果  
 //initialData: ,// a Stream<int> or null  
 builder: (BuildContext context, AsyncSnapshot<int> snapshot) {  
 if (snapshot.hasError)  
 return Text('Error: ${snapshot.error}');  
 switch (snapshot.connectionState) {  
 case ConnectionState.none:  
 return Text('没有Stream');  
 case ConnectionState.waiting:  
 return Text('等待数据...');  
 case ConnectionState.active:  
 return Text('active: ${snapshot.data}');  
 case ConnectionState.done:  
 return Text('Stream已关闭');  
 }  
 return null; // unreachable  
 },  
 );  
}

# 对话框详解

## AlertDialog

构造函数定义如下：

const AlertDialog({

Key key,

this.title, //对话框标题组件

this.titlePadding, // 标题填充

this.titleTextStyle, //标题文本样式

this.content, // 对话框内容组件

this.contentPadding = const EdgeInsets.fromLTRB(24.0, 20.0, 24.0, 24.0), //内容的填充

this.contentTextStyle,// 内容文本样式

this.actions, // 对话框操作按钮组

this.backgroundColor, // 对话框背景色

this.elevation,// 对话框的阴影

this.semanticLabel, //对话框语义化标签(用于读屏软件)

this.shape, // 对话框外形

})

**例子：**

import 'package:flutter/material.dart';  
  
class MyDemo extends StatefulWidget {  
 @override  
 \_MyDemoState createState() => \_MyDemoState();  
}  
  
class \_MyDemoState extends State<MyDemo> {  
 @override  
 Widget build(BuildContext context) {  
 return RaisedButton(  
 child: Text("点我刪除"),  
 onPressed: () async {  
 //弹出对话框并等待其关闭  
 bool delete = await showDeleteConfirmDialog1(context);  
 if (delete == null) {  
 print("取消删除");  
 } else {  
 print("已确认删除");  
 //... 删除文件  
 }  
 },  
 );  
 }  
}  
// 弹出对话框  
Future<bool> showDeleteConfirmDialog1(context) {  
 return showDialog<bool>(  
 context: context,  
 builder: (context) {  
 return AlertDialog(  
 title: Text("提示"),  
 content: Text("您确定要删除当前文件吗?"),  
 actions: <Widget>[  
 FlatButton(  
 child: Text("取消"),  
 // 关闭对话框

onPressed: () => Navigator.*of*(context).pop(),  
 ),  
 FlatButton(  
 child: Text("删除"),  
 onPressed: () {  
 //关闭对话框并返回true  
 Navigator.*of*(context).pop(true);  
 },  
 ),  
 ],  
 );  
 },  
 );  
}

## SimpleDialog

**例子：**

RaisedButton(  
 child: Text("点我切换语言"),  
 onPressed: () async {  
 //弹出对话框并等待其关闭  
 await changeLanguage(context);  
 },  
),

Future<void> changeLanguage(context) async {  
 int i = await showDialog<int>(  
 context: context,  
 builder: (BuildContext context) {  
 return SimpleDialog(  
 title: const Text('请选择语言'),  
 children: <Widget>[  
 SimpleDialogOption(  
 onPressed: () {  
 // 返回1  
 Navigator.*pop*(context, 1);  
 },  
 child: Padding(  
 padding: const EdgeInsets.symmetric(vertical: 6),  
 child: const Text('中文简体'),  
 ),  
 ),  
 SimpleDialogOption(  
 onPressed: () {  
 // 返回2  
 Navigator.*pop*(context, 2);  
 },  
 child: Padding(  
 padding: const EdgeInsets.symmetric(vertical: 6),  
 child: const Text('美国英语'),  
 ),  
 ),  
 ],  
 );  
 });  
  
 if (i != null) {  
 print("选择了：${i == 1 ? "中文简体" : "美国英语"}");  
 }  
}

## Dialog

**例子：**

RaisedButton(  
 child: Text("点我切换语言"),  
 onPressed: () async {  
 //弹出对话框并等待其关闭  
 await showListDialog(context);  
 },  
),

Future<void> showListDialog(context) async {  
 int index = await showDialog<int>(  
 context: context,  
 builder: (BuildContext context) {  
 var child = Column(  
 children: <Widget>[  
 ListTile(title: Text("请选择")),  
 Expanded(  
 child: ListView.builder(  
 itemCount: 30,  
 itemBuilder: (BuildContext context, int index) {  
 return ListTile(  
 title: Text("$index"),  
 onTap: () => Navigator.*of*(context).pop(index),  
 );  
 },  
 )),  
 ],  
 );  
 //使用AlertDialog会报错  
 //return AlertDialog(content: child);  
 return Dialog(child: child);  
 },  
 );  
 if (index != null) {  
 print("点击了：$index");  
 }  
}

**三个例子的效果依次是：**

## C:\Users\admin\AppData\Local\Temp\WeChat Files\a721a2d7493b22381e9d3198fd575e1.jpgC:\Users\admin\AppData\Local\Temp\WeChat Files\708c99f7ba52fd4fde47a18480e5fd4.jpgC:\Users\admin\AppData\Local\Temp\WeChat Files\78f573f92a35fff7222deae080b93a8.jpg对话框打开动画及遮罩

Future<T> showGeneralDialog<T>({

@required BuildContext context,

@required RoutePageBuilder pageBuilder, //构建对话框内部UI

bool barrierDismissible, //点击遮罩是否关闭对话框

String barrierLabel, // 语义化标签(用于读屏软件)

Color barrierColor, // 遮罩颜色

Duration transitionDuration, // 对话框打开/关闭的动画时长

RouteTransitionsBuilder transitionBuilder, // 对话框打开/关闭的动画

})

**直接上代码：**

RaisedButton(  
 child: Text("点我动画删除"),  
 onPressed: () async {  
 //弹出对话框并等待其关闭  
 await showMyDialog(context);  
 },  
),

Future<bool> showMyDialog(context) {  
 showCustomDialog<bool>(  
 context: context,  
 builder: (context) {  
 return AlertDialog(  
 title: Text("提示"),  
 content: Text("您确定要删除当前文件吗?"),  
 actions: <Widget>[  
 FlatButton(  
 child: Text("取消"),  
 onPressed: () => Navigator.*of*(context).pop(),  
 ),  
 FlatButton(  
 child: Text("删除"),  
 onPressed: () {  
// 执行删除操作  
 Navigator.*of*(context).pop(true);  
 },  
 ),  
 ],  
 );  
 },  
 );  
}  
  
Future<T> showCustomDialog<T>({  
 @required BuildContext context,  
 bool barrierDismissible = true,  
 WidgetBuilder builder,  
}) {  
 final ThemeData theme = Theme.*of*(context, shadowThemeOnly: true);  
 return showGeneralDialog(  
 context: context,  
 pageBuilder: (BuildContext buildContext, Animation<double> animation,  
 Animation<double> secondaryAnimation) {  
 final Widget pageChild = Builder(builder: builder);  
 return SafeArea(  
 child: Builder(builder: (BuildContext context) {  
 return theme != null  
 ? Theme(data: theme, child: pageChild)  
 : pageChild;  
 }),  
 );  
 },  
 barrierDismissible: barrierDismissible,  
 barrierLabel: MaterialLocalizations.*of*(context).modalBarrierDismissLabel,  
 barrierColor: Colors.*black87*,  
 // 自定义遮罩颜色  
 transitionDuration: const Duration(milliseconds: 150),  
 transitionBuilder: \_buildMaterialDialogTransitions,  
 );  
}  
  
Widget \_buildMaterialDialogTransitions(  
 BuildContext context,  
 Animation<double> animation,  
 Animation<double> secondaryAnimation,  
 Widget child) {  
 // 使用缩放动画  
 return ScaleTransition(  
 scale: CurvedAnimation(  
 parent: animation,  
 curve: Curves.*easeInCirc*,  
 ),  
 child: child,  
 );  
}

## 底部菜单列表

**可以用于分享，购物车等等常见情况！**

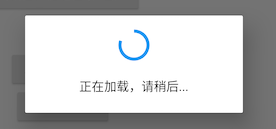
RaisedButton(  
 child: Text("底部弹出"),  
 onPressed: () async {  
 //弹出对话框并等待其关闭  
 await \_showModalBottomSheet(context);  
 },  
),

// 弹出底部菜单列表模态对话框  
Future<int> \_showModalBottomSheet(context) {  
 return showModalBottomSheet<int>(  
 context: context,  
 builder: (BuildContext context) {  
 return ListView.builder(  
 itemCount: 30,  
 itemBuilder: (BuildContext context, int index) {  
 return ListTile(  
 title: Text("$index"),  
 onTap: () => Navigator.*of*(context).pop(index),  
 );  
 },  
 );   
 },  
 );  
}

## Loading框

RaisedButton(  
 child: Text("等待加载"),  
 onPressed: () async {  
 //弹出对话框并等待其关闭  
 await showLoadingDialog(context);  
 },  
),

showLoadingDialog(context) {  
 showDialog(  
 context: context,  
 barrierDismissible: false, //点击遮罩不关闭对话框  
 builder: (context) {  
 return AlertDialog(  
 content: Column(  
 mainAxisSize: MainAxisSize.min,  
 children: <Widget>[  
 CircularProgressIndicator(),  
 Padding(  
 padding: const EdgeInsets.only(top: 26.0),  
 child: Text("正在加载，请稍后..."),  
 )  
 ],  
 ),  
 );  
 },  
 );  
}



**写在后面的话：还可以实现日历、城市选择等功能，可享用三方包成果：**<https://pub.dev/>

# 原始指针事件处理：

在移动端，各个平台或UI系统的原始指针事件模型基本都是一致的：一次完整的事件分为三个阶段：**手指按下、手指移动、手指抬起**。更高级别的手势（点击、双击、拖动等）都是基于原始事件的。

当指针按下时，flutter会对应用程序执行命中测试（Hit Test），以确定指针与屏幕接触的位置存在哪些组件。和Web一样，被命中的组件发生的事件会有**事件冒泡**，**不过要注意的是Flutter中并没有机制取消或者停止冒泡。**

**Flutter中可以使用Listener来监听原始触摸事件，其构造函数定义如下：**

Listener({

Key key,

this.onPointerDown, //手指按下回调

this.onPointerMove, //手指移动回调

this.onPointerUp,//手指抬起回调

this.onPointerCancel,//触摸事件取消回调

this.behavior = HitTestBehavior.deferToChild, //在命中测试期间如何表现

Widget child

})

**PointerDownEvent、PointerMoveEvent、PointerUpEvent都是PointerEvent的一个子类，PointerEvent类中包括当前指针的一些信息，常用的有：**

* position：它是鼠标相对于当对于全局坐标的偏移。
* delta：两次指针移动事件（PointerMoveEvent）的距离。
* pressure：按压力度，如果手机屏幕支持压力传感器(如iPhone的3D Touch)，此属性会更有意义，如果手机不支持，则始终为1。
* orientation：指针移动方向，是一个角度值。

**behavior属性决定子组件如何响应命中测试，它的值类型为HitTestBehavior，这是一个枚举类，有三个枚举值：**

* deferToChild：子组件会一个接一个的进行命中测试，如果子组件中有测试通过的，则当前组件通过，这就意味着，如果指针事件作用于子组件上时，其父级组件也肯定可以收到该事件。
* opaque：在命中测试时，将当前组件当成不透明处理(即使本身是透明的)，最终的效果相当于当前Widget的整个区域都是点击区域。

Listener(  
 child: ConstrainedBox(  
 constraints: BoxConstraints.tight(Size(300.0, 150.0)),  
 child: Center(child: Text("Box A")),  
 ),  
 //behavior: HitTestBehavior.opaque,  
 onPointerDown: (event) => print("down A")  
),

上例中，只有点击文本内容区域才会触发点击事件，因为deferToChild会去子组件判断是否命中测试，而该例中子组件就是 Text("Box A")。 如果我们想让整个300×150的矩形区域都能点击我们可以将behavior设为HitTestBehavior.opaque。注意，该属性并不能用于在组件树中拦截（忽略）事件，它只是决定命中测试时的组件大小。

* translucent：当点击组件透明区域时，可以对自身边界内及底部可视区域都进行命中测试，这意味着点击顶部组件透明区域时，顶部组件和底部组件都可以接收到事件，例如：

Stack(  
 children: <Widget>[  
 Listener(  
 child: ConstrainedBox(  
 constraints: BoxConstraints.tight(Size(300.0, 200.0)),  
 child: DecoratedBox(  
 decoration: BoxDecoration(color: Colors.*blue*)),  
 ),  
 onPointerDown: (event) => print("down0"),  
 ),  
 Listener(  
 child: ConstrainedBox(  
 constraints: BoxConstraints.tight(Size(200.0, 100.0)),  
 child: Center(child: Text("左上角200\*100范围内非文本区域点击")),  
 ),  
 onPointerDown: (event) => print("down1"),  
 //behavior: HitTestBehavior.translucent, //放开此行注释后可以"点透"  
 )  
 ],  
),

上例中，当注释掉最后一行代码后，在左上角200\*100范围内非文本区域点击时（顶部组件透明区域），控制台只会打印“down0”，也就是说顶部组件没有接收到事件，而只有底部接收到了。当放开注释后，再点击时顶部和底部都会接收到事件，此时会打印：

I/flutter ( 3039): down1

I/flutter ( 3039): down0

如果behavior值改为HitTestBehavior.opaque，则只会打印"down1"。

## 忽略PointerEvent

假如我们不想让某个子树响应PointerEvent的话，我们可以使用IgnorePointer和AbsorbPointer，这两个组件都能阻止子树接收指针事件，不同之处在于AbsorbPointer本身会参与命中测试，而IgnorePointer本身不会参与，这就意味着AbsorbPointer本身是可以接收指针事件的(但其子树不行)，而IgnorePointer不可以。一个简单的例子如下：

Listener(  
 child: AbsorbPointer(  
 child: Listener(  
 child: Container(  
 color: Colors.*red*,  
 width: 200.0,  
 height: 100.0,  
 ),  
 onPointerDown: (event)=>print("in"),  
 ),  
 ),  
 onPointerDown: (event)=>print("up"),  
)

点击Container时，由于它在AbsorbPointer的子树上，所以不会响应指针事件，所以日志不会输出"in"，但AbsorbPointer本身是可以接收指针事件的，所以会输出"up"。如果将AbsorbPointer换成IgnorePointer，那么两个都不会输出。

# 手势识别

**Flutter中用于处理手势：GestureDetector和GestureRecognizer。**

## 点击：

**onTapDown：可能导致点击的指针已联系到屏幕的特定位置。**

**onTapUp：触发点按的指针已停止在特定位置与屏幕联系。**

**onTap：发生了点击。**

**onTapCancel：以前触发onTapDown的指针最终没有导致点击。**

## 双击：

**onTapDown：onDoubleTap：用户已经快速连续地在同一位置点击屏幕两次。**

## 长按：

**onLongPress：指针在相同的位置与屏幕保持的长时间接触。**

## 垂直拖动：

**onVerticalDragStart：指针已经接触到屏幕，而且可能开始垂直移动。**

**onVerticalDragUpdate：与屏幕接触并垂直移动的指针沿垂直方向移动。**

**onVerticalDragEnd：以前与屏幕接触并垂直移动的指针不再与屏幕接触，并且当其停止接触屏幕时以特定速度移动。**

## 水平拖动：

**onHorizontalDragStart：指针已经接触到屏幕，而且可能开始水平移动。**

**onHorizontalDragUpdate：与屏幕接触并水平移动的指针沿水平方向移动。**

**onHorizontalDragEnd：以前与屏幕接触并水平移动的指针不再与屏幕接触，并且当其停止接触屏幕时以特定速度移动**

## 缩放：

**onScaleStart：指针已经接触到屏幕，而且可能反向或相向移动。**

**onScaleUpdate：与屏幕接触并双向移动的指针沿反向或相向移动。**

**onScaleEnd：以前与屏幕接触并双向移动的指针不再与屏幕接触，并且停止双向移动**

## GestureDetector

点击、双击、长按（直接看代码）

import 'package:flutter/material.dart';  
class GestureDetectorTestRoute extends StatefulWidget {  
 @override  
 MyDemo createState() =>  
 new MyDemo();  
}  
  
class MyDemo extends State<GestureDetectorTestRoute> {  
 String \_operation = "No Gesture detected!"; //保存事件名  
 @override  
 Widget build(BuildContext context) {  
 return Center(  
 child: GestureDetector(  
 child: Container(  
 alignment: Alignment.*center*,  
 color: Colors.*blue*,  
 width: 200.0,  
 height: 100.0,  
 child: Text(\_operation,  
 style: TextStyle(color: Colors.*white*),  
 ),  
 ),  
 onTap: () => updateText("Tap"),//点击  
 onDoubleTap: () => updateText("DoubleTap"), //双击  
 onLongPress: () => updateText("LongPress"), //长按  
 ),  
 );  
 }  
  
 void updateText(String text) {  
 //更新显示的事件名  
 setState(() {  
 \_operation = text;  
 });  
 }  
}

**注意**： 当同时监听onTap和onDoubleTap事件时，当用户触发tap事件时，会有200毫秒左右的延时，这是因为当用户点击完之后很可能会再次点击以触发双击事件，所以GestureDetector会等一段时间来确定是否为双击事件。如果用户只监听了onTap（没有监听onDoubleTap）事件时，则没有延时。

拖动、滑动

**下面代码实现一个悬浮的按钮场景，可以在屏幕上随意拖拽：**

import 'package:flutter/material.dart';  
  
class Drag extends StatefulWidget {  
 @override  
 \_DragState createState() => new \_DragState();  
}  
  
class \_DragState extends State<Drag> with SingleTickerProviderStateMixin {  
 double \_top = 0.0; //距顶部的偏移  
 double \_left = 0.0;//距左边的偏移  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("测试"),  
 ),  
 body: Stack(  
 children: <Widget>[  
 Positioned(  
 top: \_top,  
 left: \_left,  
 child: GestureDetector(  
 child: CircleAvatar(child: Text("A")),  
 //手指按下时会触发此回调  
 onPanDown: (DragDownDetails e) {  
 //打印手指按下的位置(相对于屏幕)  
 print("用户手指按下：${e.globalPosition}");  
 },  
 //手指滑动时会触发此回调  
 onPanUpdate: (DragUpdateDetails e) {  
 //用户手指滑动时，更新偏移，重新构建  
 setState(() {  
 \_left += e.delta.dx;  
 \_top += e.delta.dy;  
 });  
 },  
 onPanEnd: (DragEndDetails e){  
 //打印滑动结束时在x、y轴上的速度  
 print(e.velocity);  
 },  
 ),  
 )  
 ],  
 ),  
 );  
 }  
}

//日志输出

2020-05-13 10:55:28.039 2392-2521/com.example.flutter\_test01 I/flutter: 用户手指按下：Offset(146.7, 768.4)

2020-05-13 10:55:28.502 2392-2521/com.example.flutter\_test01 I/flutter: Velocity(1838.0, -1854.3)

2020-05-13 10:55:29.778 2392-2521/com.example.flutter\_test01 I/flutter: 用户手指按下：Offset(271.6, 387.0)

2020-05-13 10:55:31.759 2392-2521/com.example.flutter\_test01 I/flutter: Velocity(-816.0, 638.6)

2020-05-13 10:55:32.201 2392-2521/com.example.flutter\_test01 I/flutter: 用户手指按下：Offset(315.8, 710.5)

2020-05-13 10:55:32.327 2392-2521/com.example.flutter\_test01 I/flutter: Velocity(1962.2, -3481.0)

2020-05-13 10:55:37.295 2392-2521/com.example.flutter\_test01 I/flutter: 用户手指按下：Offset(367.2, 453.3)

2020-05-13 10:55:37.787 2392-2521/com.example.flutter\_test01 I/flutter: Velocity(3429.4, -3008.3)

* DragDownDetails.globalPosition：当用户按下时，此属性为用户按下的位置相对于**屏幕**（而非父组件）原点(左上角)的偏移。
* DragUpdateDetails.delta：当用户在屏幕上滑动时，会触发多次Update事件，delta指一次Update事件的滑动的偏移量。
* DragEndDetails.velocity：该属性代表用户抬起手指时的滑动速度(包含x、y两个轴的），示例中并没有处理手指抬起时的速度，常见的效果是根据用户抬起手指时的速度做一个减速动画。

单一方向拖动：（直接看代码应该就很清晰了）

import 'package:flutter/material.dart';  
  
class Drag extends StatefulWidget {  
 @override  
 \_DragState createState() => new \_DragState();  
}  
  
class \_DragState extends State<Drag> with SingleTickerProviderStateMixin {  
 double \_top = 1.0; //距顶部的偏移  
 double \_left = 0.0;//距左边的偏移  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("测试"),  
 ),  
 body: Stack(  
 children: <Widget>[  
 Positioned(  
 left: \_left,  
 child: GestureDetector(  
 child: CircleAvatar(child: Text("A")),  
 //垂直方向拖动事件  
 onHorizontalDragUpdate: (DragUpdateDetails details) {  
 setState(() {  
 \_left += details.delta.dx;  
 });  
 }  
 ),  
 ),  
 Positioned(  
 top: \_top,  
 child: GestureDetector(  
 child: CircleAvatar(child: Text("B")),  
 //垂直方向拖动事件  
 onVerticalDragUpdate: (DragUpdateDetails details) {  
 setState(() {  
 \_top += details.delta.dy;  
 });  
 }  
 ),  
 ),  
 ],  
 ),  
 );  
 }  
}

简易缩放功能实现：

import 'package:flutter/material.dart';  
  
class ScaleTestRoute extends StatefulWidget {  
 @override  
 \_ScaleTestRouteState createState() => new \_ScaleTestRouteState();  
}  
  
class \_ScaleTestRouteState extends State<ScaleTestRoute> {  
 double \_width = 200.0; //通过修改图片宽度来达到缩放效果  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("测试"),  
 ),  
 body: Center(  
 child: GestureDetector(  
 //指定宽度，高度自适应  
 child: Image.network("https://goss.veer.com/creative/vcg/veer/800water/veer-310891172.jpg", width: \_width),  
 onScaleUpdate: (ScaleUpdateDetails details) {  
 setState(() {  
 //缩放倍数在0.8到10倍之间  
 \_width=200\*details.scale.clamp(.8, 10.0);  
 });  
 },  
 ),  
 ),  
 );  
 }  
}

## GestureRecognizer

GestureDetector内部是使用一个或多个GestureRecognizer来识别各种手势的，而GestureRecognizer的作用就是通过Listener来将原始指针事件转换为语义手势，GestureDetector直接可以接收一个子widget。GestureRecognizer是一个抽象类，一种手势的识别器对应一个GestureRecognizer的子类，Flutter实现了丰富的手势识别器，我们可以直接使用。

**一个实用的例子：**

假设我们要给一段富文本（RichText）的不同部分分别添加点击事件处理器，但是TextSpan并不是一个widget，这时我们不能用GestureDetector，但TextSpan有一个recognizer属性，它可以接收一个GestureRecognizer。假设我们需要在点击时给文本变色:

import 'package:flutter/material.dart';  
import 'package:flutter/gestures.dart';  
  
class GestureRecognizerTestRoute extends StatefulWidget {  
 @override  
 \_GestureRecognizerTestRouteState createState() =>  
 new \_GestureRecognizerTestRouteState();  
}  
  
class \_GestureRecognizerTestRouteState  
 extends State<GestureRecognizerTestRoute> {  
 TapGestureRecognizer \_tapGestureRecognizer = new TapGestureRecognizer();  
 bool \_toggle = false; //变色开关  
  
 @override  
 void dispose() {  
 //用到GestureRecognizer的话一定要调用其dispose方法释放资源  
 \_tapGestureRecognizer.dispose();  
 super.dispose();  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("测试"),  
 ),  
 body: Center(  
 child: Text.rich(TextSpan(children: [  
 TextSpan(text: "你好世界"),  
 TextSpan(  
 text: "点我变色",  
 style: TextStyle(  
 fontSize: 30.0, color: \_toggle ? Colors.*blue* : Colors.*red*),  
 recognizer: \_tapGestureRecognizer  
 ..onTap = () {  
 setState(() {  
 \_toggle = !\_toggle;  
 });  
 },  
 ),  
 TextSpan(text: "你好世界"),  
 ])),  
 ),  
 );  
 }  
}

## 手势竞争与冲突

竞争：

1.同时有两个事件，一个是水平滑动，一个是垂直滑动，就形成了竞争。

2.父子组件同时拥有事件，也会形成竞争。

竞争结果：

1.取决于第一次移动时两个轴上的位移分量，哪个轴的大，哪个轴在本次滑动事件竞争中就胜出。也就是执行那个事件。

2.如果在子组件操作，子组件拥有优先处理权，所以子组件执行事件，父组件不执行。

冲突：

由于手势竞争最终只有一个胜出者，所以，当有多个手势识别器时，可能会产生冲突。

冲突解决方案：

**手势冲突只是手势级别的，而手势是对原始指针的语义化的识别，所以在遇到复杂的冲突场景时，都可以通过Listener直接识别原始指针事件来解决冲突。**

# 事件总线

在APP中，我们经常会需要一个广播机制，用以跨页面事件通知，比如一个需要登录的APP中，页面会关注用户登录或注销事件，来进行一些状态更新。这时候，一个事件总线便会非常有用，事件总线通常实现了订阅者模式，订阅者模式包含发布者和订阅者两种角色，可以通过事件总线来触发事件和监听事件，本节我们实现一个简单的全局事件总线，我们使用单例模式，代码如下：

//订阅者回调签名  
typedef void EventCallback(arg);  
  
class EventBus {  
 //私有构造函数  
 EventBus.\_internal();  
  
 //保存单例  
 static EventBus *\_singleton* = new EventBus.\_internal();  
  
 //工厂构造函数  
 factory EventBus()=> *\_singleton*;  
  
 //保存事件订阅者队列，key:事件名(id)，value: 对应事件的订阅者队列  
 var \_emap = new Map<Object, List<EventCallback>>();  
  
 //添加订阅者  
 void on(eventName, EventCallback f) {  
 if (eventName == null || f == null) return;  
 \_emap[eventName] ??= new List<EventCallback>();  
 \_emap[eventName].add(f);  
 }  
  
 //移除订阅者  
 void off(eventName, [EventCallback f]) {  
 var list = \_emap[eventName];  
 if (eventName == null || list == null) return;  
 if (f == null) {  
 \_emap[eventName] = null;  
 } else {  
 list.remove(f);  
 }  
 }  
  
 //触发事件，事件触发后该事件所有订阅者会被调用  
 void emit(eventName, [arg]) {  
 var list = \_emap[eventName];  
 if (list == null) return;  
 int len = list.length - 1;  
 //反向遍历，防止订阅者在回调中移除自身带来的下标错位  
 for (var i = len; i > -1; --i) {  
 list[i](arg);  
 }  
 }  
}  
  
//定义一个top-level（全局）变量，页面引入该文件后可以直接使用bus  
var bus = new EventBus();

//定义一个top-level（全局）变量，页面引入该文件后可以直接使用bus  
var bus = new EventBus();

//使用  
//页面A中  
...  
//监听登录事件  
bus.on("login", (arg) {  
// do something  
});  
  
//登录页B中  
...  
//登录成功后触发登录事件，页面A中订阅者会被调用  
bus.emit("login", userInfo);

Dart中实现单例模式的标准做法就是使用static变量+工厂构造函数的方式，这样就可以保证new EventBus()始终返回都是同一个实例，读者应该理解并掌握这种方法。

问题：虽然可以监听和通知，但是当状态改变，会进行多次累计通知。是否需要再页面销毁时进行销毁？如何实现？

# 通知（Notification）

通知（Notification）是Flutter中一个重要的机制，在widget树中，每一个节点都可以分发通知，通知会沿着当前节点向上传递，所有父节点都可以通过NotificationListener来监听通知。Flutter中将这种由子向父的传递通知的机制称为**通知冒泡**（Notification Bubbling）。通知冒泡和用户触摸事件冒泡是相似的，但有一点不同：通知冒泡可以中止，但用户触摸事件不行。

**NotificationListener定义如下：**

class NotificationListener<T extends Notification> extends StatelessWidget {

const NotificationListener({

Key key,

@required this.child,

this.onNotification,

}) : super(key: key);

...//省略无关代码

}

1. NotificationListener继承自StatelessWidget类，所以它可以直接嵌套到Widget树中。
2. NotificationListener可以指定一个模板参数，该模板参数类型必须是继承自Notification；当显式指定模板参数时，NotificationListener便只会接收该参数类型的通知。
3. onNotification回调为通知处理回调，其函数签名如下：

typedef NotificationListenerCallback<T extends Notification> = bool Function(T notification);

它的返回值类型为布尔值，当返回值为true时，阻止冒泡，其父级Widget将再也收不到该通知；当返回值为false时继续向上冒泡通知。

## 自定义通知：

1. 定义一个通知类，要继承自Notification类

class MyNotification extends Notification {

MyNotification(this.msg);

final String msg;

}

1. 分发通知。

Notification有一个dispatch(context)方法，它是用于分发通知的，我们说过context实际上就是操作Element的一个接口，它与Element树上的节点是对应的，通知会从context对应的Element节点向上冒泡。

**这是一个简单的完整例子：**

import 'package:flutter/material.dart';  
  
class NotificationRoute extends StatefulWidget {  
 @override  
 NotificationRouteState createState() {  
 return new NotificationRouteState();  
 }  
}  
  
class NotificationRouteState extends State<NotificationRoute> {  
 String \_msg = "";  
  
 @override  
 Widget build(BuildContext context) {  
 //监听通知  
 return Scaffold(  
 body: NotificationListener<MyNotification>(  
 onNotification: (notification) {  
 setState(() {  
 \_msg += notification.msg + "~";  
 });  
 return true;  
 },  
 child: Center(  
 child: Column(  
 mainAxisSize: MainAxisSize.min,  
 children: <Widget>[  
// RaisedButton(  
// onPressed: () => MyNotification("Hi").dispatch(context),  
// child: Text("Send Notification"),  
// ),  
 Builder(  
 builder: (context) {  
 return RaisedButton(  
 //按钮点击时分发通知  
 onPressed: () => MyNotification("Hi").dispatch(context),  
 child: Text("Send Notification"),  
 );  
 },  
 ),  
 Text(\_msg)  
 ],  
 ),  
 ),  
 ),  
 );  
 }  
}  
  
class MyNotification extends Notification {  
 MyNotification(this.msg);  
  
 final String msg;  
}

注意：代码中注释的部分是不能正常工作的，因为这个context是根Context，而NotificationListener是监听的子树，所以我们通过Builder来构建RaisedButton，来获得按钮位置的context。

## 阻止冒泡

上面例子稍微做些改动：

class NotificationRouteState extends State<NotificationRoute> {

String \_msg="";

@override

Widget build(BuildContext context) {

//监听通知

return NotificationListener<MyNotification>(

onNotification: (notification){

print(notification.msg); //打印通知

return false;

},

child: NotificationListener<MyNotification>(

onNotification: (notification) {

setState(() {

\_msg+=notification.msg+" ";

});

return false;

},

child: ...//省略重复代码

),

);

}

}

此时两个NotificationListener进行了嵌套，子NotificationListener的onNotification回调返回了false，表示不阻止冒泡，所以父NotificationListener仍然会受到通知，所以控制台会打印出通知信息；如果将子NotificationListener的onNotification回调的返回值改为true，则父NotificationListener便不会再打印通知了，因为子NotificationListener已经终止通知冒泡了。

## 通知冒泡原理

Flutter框架中是如何实现通知冒泡的？为了搞清楚这个问题，就必须看一下源码，我们从通知分发的的源头出发，然后再顺藤摸瓜。由于通知是通过Notification的dispatch(context)方法发出的，那我们先看看dispatch(context)方法中做了什么，下面是相关源码：

void dispatch(BuildContext target) {

target?.visitAncestorElements(visitAncestor);

}

dispatch(context)中调用了当前context的visitAncestorElements方法，该方法会从当前Element开始向上遍历父级元素；visitAncestorElements有一个遍历回调参数，在遍历过程中对遍历到的父级元素都会执行该回调。遍历的终止条件是：已经遍历到根Element或某个遍历回调返回false。源码中传给visitAncestorElements方法的遍历回调为visitAncestor方法，我们看看visitAncestor方法的实现：

//遍历回调，会对每一个父级Element执行此回调

bool visitAncestor(Element element) {

//判断当前element对应的Widget是否是NotificationListener。

//由于NotificationListener是继承自StatelessWidget，

//故先判断是否是StatelessElement

if (element is StatelessElement) {

//是StatelessElement，则获取element对应的Widget，判断

//是否是NotificationListener 。

final StatelessWidget widget = element.widget;

if (widget is NotificationListener<Notification>) {

//是NotificationListener，则调用该NotificationListener的\_dispatch方法

if (widget.\_dispatch(this, element))

return false;

}

}

return true;

}

visitAncestor会判断每一个遍历到的父级Widget是否是NotificationListener，如果不是，则返回true继续向上遍历，如果是，则调用NotificationListener的\_dispatch方法，我们看看\_dispatch方法的源码：

bool \_dispatch(Notification notification, Element element) {

// 如果通知监听器不为空，并且当前通知类型是该NotificationListener

// 监听的通知类型，则调用当前NotificationListener的onNotification

if (onNotification != null && notification is T) {

final bool result = onNotification(notification);

// 返回值决定是否继续向上遍历

return result == true;

}

return false;

}

我们可以看到NotificationListener的onNotification回调最终是在\_dispatch方法中执行的，然后会根据返回值来确定是否继续向上冒泡。上面的源码实现其实并不复杂，通过阅读这些源码，一些额外的点读者可以注意一下：

1. Context上也提供了遍历Element树的方法。
2. 我们可以通过Element.widget得到element节点对应的widget；我们已经反复讲过Widget和Element的对应关系，读者通过这些源码来加深理解。

# 动画

## Flutter动画介绍

Flutter中对动画进行了抽象，主要涉及Animation、Curve、Controller、Tween这四个角色。

Animation

Animation是一个抽象类，它本身和UI渲染没有任何关系，而它主要的功能是保存动画的插值和状态；其中一个比较常用的Animation类是Animation<double>。Animation对象是一个在一段时间内依次生成一个区间(Tween)之间值的类。Animation对象在整个动画执行过程中输出的值可以是线性的、曲线的、一个步进函数或者任何其他曲线函数等等，这由Curve来决定。 根据Animation对象的控制方式，动画可以正向运行（从起始状态开始，到终止状态结束），也可以反向运行，甚至可以在中间切换方向。Animation还可以生成除double之外的其他类型值，如：Animation<Color> 或Animation<Size>。在动画的每一帧中，我们可以通过Animation对象的value属性获取动画的当前状态值。

动画通知：

我们可以通过Animation来监听动画每一帧以及执行状态的变化，Animation有如下两个方法：

1. addListener()；它可以用于给Animation添加帧监听器，在每一帧都会被调用。帧监听器中最常见的行为是改变状态后调用setState()来触发UI重建。
2. addStatusListener()；它可以给Animation添加“动画状态改变”监听器；动画开始、结束、正向或反向（见AnimationStatus定义）时会调用状态改变的监听器。

Curve

动画过程可以是匀速的、匀加速的或者先加速后减速等。Flutter中通过Curve（曲线）来描述动画过程，我们把匀速动画称为线性的(Curves.linear)，而非匀速动画称为非线性的。

我们可以通过CurvedAnimation来指定动画的曲线，如：

final CurvedAnimation curve =

new CurvedAnimation(parent: controller, curve: Curves.easeIn);

CurvedAnimation和AnimationController（下面介绍）都是Animation<double>类型。CurvedAnimation可以通过包装AnimationController和Curve生成一个新的动画对象 ，我们正是通过这种方式来将动画和动画执行的曲线关联起来的。我们指定动画的曲线为Curves.easeIn，它表示动画开始时比较慢，结束时比较快。[Curves](https://docs.flutter.io/flutter/animation/Curves-class.html)类是一个预置的枚举类，定义了许多常用的曲线，下面列几种常用的：

| **Curves曲线** | **动画过程** |
| --- | --- |
| linear | 匀速的 |
| decelerate | 匀减速 |
| ease | 开始加速，后面减速 |
| easeIn | 开始慢，后面快 |
| easeOut | 开始快，后面慢 |
| easeInOut | 开始慢，然后加速，最后再减速 |

除了上面列举的，[Curves](https://docs.flutter.io/flutter/animation/Curves-class.html)类中还定义了许多其它的曲线，可以自行查看Curves类定义。

当然我们也可以创建自己Curve，例如我们定义一个正弦曲线：

class ShakeCurve extends Curve {

@override

double transform(double t) {

return math.sin(t \* math.PI \* 2);

}

}

AnimationController

AnimationController用于控制动画，它包含动画的启动forward()、停止stop()、反向播放reverse()等方法。AnimationController会在动画的每一帧，就会生成一个新的值。默认情况下，AnimationController在给定的时间段内线性的生成从0.0到1.0（默认区间）的数字。 例如，下面代码创建一个Animation对象（但不会启动动画）：

final AnimationController controller = new AnimationController(

duration: const Duration(milliseconds: 2000), vsync: this);

AnimationController生成数字的区间可以通过lowerBound和upperBound来指定，如：

final AnimationController controller = new AnimationController(

duration: const Duration(milliseconds: 2000),

lowerBound: 10.0,

upperBound: 20.0,

vsync: this

);

AnimationController派生自Animation<double>，因此可以在需要Animation对象的任何地方使用。但是，AnimationController具有控制动画的其他方法，例如forward()方法可以启动正向动画，reverse()可以启动反向动画。在动画开始执行后开始生成动画帧，屏幕每刷新一次就是一个动画帧，在动画的每一帧，会随着根据动画的曲线来生成当前的动画值（Animation.value），然后根据当前的动画值去构建UI，当所有动画帧依次触发时，动画值会依次改变，所以构建的UI也会依次变化，所以最终我们可以看到一个完成的动画。 另外在动画的每一帧，Animation对象会调用其帧监听器，等动画状态发生改变时（如动画结束）会调用状态改变监听器。

duration表示动画执行的时长，通过它我们可以控制动画的速度。

**注意**： 在某些情况下，动画值可能会超出AnimationController的[0.0，1.0]的范围，这取决于具体的曲线。例如，fling()函数可以根据我们手指滑动（甩出）的速度(velocity)、力量(force)等来模拟一个手指甩出动画，因此它的动画值可以在[0.0，1.0]范围之外 。也就是说，根据选择的曲线，CurvedAnimation的输出可以具有比输入更大的范围。例如，Curves.elasticIn等弹性曲线会生成大于或小于默认范围的值。

Ticker

当创建一个AnimationController时，需要传递一个vsync参数，它接收一个TickerProvider类型的对象，它的主要职责是创建Ticker，定义如下：

abstract class TickerProvider {

//通过一个回调创建一个Ticker

Ticker createTicker(TickerCallback onTick);

}

Flutter应用在启动时都会绑定一个SchedulerBinding，通过SchedulerBinding可以给每一次屏幕刷新添加回调，而Ticker就是通过SchedulerBinding来添加屏幕刷新回调，这样一来，每次屏幕刷新都会调用TickerCallback。使用Ticker(而不是Timer)来驱动动画会防止屏幕外动画（动画的UI不在当前屏幕时，如锁屏时）消耗不必要的资源，因为Flutter中屏幕刷新时会通知到绑定的SchedulerBinding，而Ticker是受SchedulerBinding驱动的，由于锁屏后屏幕会停止刷新，所以Ticker就不会再触发。

Tween

默认情况下，AnimationController对象值的范围是[0.0，1.0]。如果我们需要构建UI的动画值在不同的范围或不同的数据类型，则可以使用Tween来添加映射以生成不同的范围或数据类型的值。例如，像下面示例，Tween生成[-200.0，0.0]的值：

final Tween doubleTween = new Tween<double>(begin: -200.0, end: 0.0);

Tween构造函数需要begin和end两个参数。Tween的唯一职责就是定义从输入范围到输出范围的映射。输入范围通常为[0.0，1.0]，但这不是必须的，我们可以自定义需要的范围。

Tween继承自Animatable<T>，而不是继承自Animation<T>，Animatable中主要定义动画值的映射规则。

下面我们看一个ColorTween将动画输入范围映射为两种颜色值之间过渡输出的例子：

final Tween colorTween =

new ColorTween(begin: Colors.transparent, end: Colors.black54);

Tween对象不存储任何状态，相反，它提供了evaluate(Animation<double>animation)方法，它可以获取动画当前映射值。Animation对象的当前值可以通过value()方法取到。evaluate函数还执行一些其它处理，例如分别确保在动画值为0.0和1.0时返回开始和结束状态。

Tween.animate

要使用Tween对象，需要调用其animate()方法，然后传入一个控制器对象。例如，以下代码在500毫秒内生成从0到255的整数值。

final AnimationController controller = new AnimationController(

duration: const Duration(milliseconds: 500), vsync: this);

Animation<int> alpha = new IntTween(begin: 0, end: 255).animate(controller);

注意animate()返回的是一个Animation，而不是一个Animatable。

以下示例构建了一个控制器、一条曲线和一个Tween：

final AnimationController controller = new AnimationController(

duration: const Duration(milliseconds: 500), vsync: this);

final Animation curve =

new CurvedAnimation(parent: controller, curve: Curves.easeOut);

Animation<int> alpha = new IntTween(begin: 0, end: 255).animate(curve);

## 动画结构

实现一个AnimatedWidget和AnimatedBuilder对比的弹出动画：

import 'package:flutter/material.dart';  
  
class ScaleAnimationRoute extends StatefulWidget {  
 @override  
 \_ScaleAnimationRouteState createState() => new \_ScaleAnimationRouteState();  
}  
  
//需要继承TickerProvider，如果有多个AnimationController，则应该使用TickerProviderStateMixin。  
class \_ScaleAnimationRouteState extends State<ScaleAnimationRoute>  
 with SingleTickerProviderStateMixin {  
 Animation<double> animation;  
 AnimationController controller;  
  
 initState() {  
 super.initState();  
 controller = new AnimationController(  
 duration: const Duration(seconds: 3), vsync: this);  
 //使用弹性曲线  
 animation = CurvedAnimation(parent: controller, curve: Curves.*bounceIn*);  
 //图片宽高从0变到300  
 animation = new Tween(begin: 0.0, end: 300.0).animate(animation)  
 ..addListener(() {  
 setState(() {});  
 });  
 //启动动画  
 controller.forward();  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return new Scaffold(  
 appBar: AppBar(  
 title: Text("动画演示"),  
 ),  
 body: Column(  
 children: <Widget>[  
 AnimatedBuilder(  
 animation: animation,  
 child: Image.network(  
 "https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1589369956883&di=9a38ed223a0b5592b57481ce1d9e1627&imgtype=0&src=http%3A%2F%2Fimage.biaobaiju.com%2Fuploads%2F20181111%2F19%2F1541937392-UxafVIlCQi.jpg"),  
 builder: (BuildContext ctx, Widget child) {  
 return new Center(  
 child: Container(  
 height: animation.value,  
 width: animation.value,  
 child: child,  
 ),  
 );  
 },  
 ),  
 GrowTransition(  
 child: Image.network("https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1589369956883&di=9a38ed223a0b5592b57481ce1d9e1627&imgtype=0&src=http%3A%2F%2Fimage.biaobaiju.com%2Fuploads%2F20181111%2F19%2F1541937392-UxafVIlCQi.jpg"),  
 animation: animation,  
 ),  
 ],  
 )  
 );  
 }  
  
 dispose() {  
 //路由销毁时需要释放动画资源  
 controller.dispose();  
 super.dispose();  
 }  
}  
  
class AnimatedImage extends AnimatedWidget {  
 AnimatedImage({Key key, Animation<double> animation})  
 : super(key: key, listenable: animation);  
  
 Widget build(BuildContext context) {  
 final Animation<double> animation = listenable;  
 return new Center(  
 child: Image.network(  
 "https://timgsa.baidu.com/timg?image&quality=80&size=b9999\_10000&sec=1589369956883&di=9a38ed223a0b5592b57481ce1d9e1627&imgtype=0&src=http%3A%2F%2Fimage.biaobaiju.com%2Fuploads%2F20181111%2F19%2F1541937392-UxafVIlCQi.jpg",  
 width: animation.value,  
 height: animation.value),  
 );  
 }  
}

//通过封装一个GrowTransition来说明，它可以对子widget实现放大动画：  
class GrowTransition extends StatelessWidget {  
 GrowTransition({this.child, this.animation});  
  
 final Widget child;  
 final Animation<double> animation;  
  
 Widget build(BuildContext context) {  
 return new Center(  
 child: new AnimatedBuilder(  
 animation: animation,  
 builder: (BuildContext context, Widget child) {  
 return new Container(  
 height: animation.value,  
 width: animation.value,  
 child: child  
 );  
 },  
 child: child  
 ),  
 );  
 }  
}

写在后面：上面的例子中，child看起来像被指定了两次。但实际发生的事情是：将外部引用child传递给AnimatedBuilder后AnimatedBuilder再将其传递给匿名构造器， 然后将该对象用作其子对象。最终的结果是AnimatedBuilder返回的对象插入到widget树中。

**说三点优势：**

1. 不用显式的去添加帧监听器，然后再调用setState() 了，这个好处和AnimatedWidget是一样的。
2. 动画构建的范围缩小了，如果没有builder，setState()将会在父组件上下文中调用，这将会导致父组件的build方法重新调用；而有了builder之后，只会导致动画widget自身的build重新调用，避免不必要的rebuild。
3. 通过AnimatedBuilder可以封装常见的过渡效果来复用动画。

动画状态监听

我们可以通过Animation的addStatusListener()方法来添加动画状态改变监听器。Flutter中，有四种动画状态，在AnimationStatus枚举类中定义:

| **枚举值** | **含义** |
| --- | --- |
| dismissed | 动画在起始点停止 |
| forward | 动画正在正向执行 |
| reverse | 动画正在反向执行 |
| completed | 动画在终点停止 |

用例子来感受下：

我们将上面图片放大的示例改为先放大再缩小再放大……这样的循环动画。要实现这种效果，我们只需要监听动画状态的改变即可，即：在动画正向执行结束时反转动画，在动画反向执行结束时再正向执行动画。代码如下：

//只需要将initSatte中的动画逻辑修改即可实现

initState() {  
 super.initState();  
 controller = new AnimationController(  
 duration: const Duration(seconds: 1), vsync: this);  
 //图片宽高从0变到300  
 animation = new Tween(begin: 0.0, end: 300.0).animate(controller);  
 animation.addStatusListener((status) {  
 if (status == AnimationStatus.completed) {  
 //动画执行结束时反向执行动画  
 controller.reverse();  
 } else if (status == AnimationStatus.dismissed) {  
 //动画恢复到初始状态时执行动画（正向）  
 controller.forward();  
 }  
 });  
  
 //启动动画（正向）  
 controller.forward();  
}

## 自定义路由切换动画

Material组件库中提供了一个MaterialPageRoute组件，它可以使用和平台风格一致的路由切换动画，如在iOS上会左右滑动切换，而在Android上会上下滑动切换。

那如何在Android上也实现左右切换的风格呢？

##### 方案一: 使用CupertinoPageRoute

CupertinoPageRoute是Cupertino组件库提供的iOS风格的路由切换组件，它实现的就是左右滑动切换。

Navigator.push(context, CupertinoPageRoute(

builder: (context)=>PageB(),

));

##### 方案二：使用PageRouteBuilder自定义路由切换动画

Navigator.push(

context,

PageRouteBuilder(

transitionDuration: Duration(milliseconds: 500), //动画时间为500毫秒

pageBuilder: (BuildContext context, Animation animation,

Animation secondaryAnimation) {

return new FadeTransition(

//使用渐隐渐入过渡,

opacity: animation,

child: PageB(), //路由B

);

},

),

);

##### 方案三：自定义路由类FadeRoute

class FadeRoute extends PageRoute {

FadeRoute({

@required this.builder,

this.transitionDuration = const Duration(milliseconds: 300),

this.opaque = true,

this.barrierDismissible = false,

this.barrierColor,

this.barrierLabel,

this.maintainState = true,

});

final WidgetBuilder builder;

@override

final Duration transitionDuration;

@override

final bool opaque;

@override

final bool barrierDismissible;

@override

final Color barrierColor;

@override

final String barrierLabel;

@override

final bool maintainState;

@override

Widget buildPage(BuildContext context, Animation<double> animation,

Animation<double> secondaryAnimation) => builder(context);

@override

Widget buildTransitions(BuildContext context, Animation<double> animation,

Animation<double> secondaryAnimation, Widget child) {

return FadeTransition(

opacity: animation,

child: builder(context),

);

}

}

//调用自定义类

Navigator.push(context, FadeRoute(builder: (context) {

return PageB();

}));

## Hero动画

Hero指的是可以在路由(页面)之间“飞行”的widget，简单来说Hero动画就是在路由切换时，有一个共享的widget可以在新旧路由间切换。由于共享的widget在新旧路由页面上的位置、外观可能有所差异，所以在路由切换时会从旧路逐渐过渡到新路由中的指定位置，这样就会产生一个Hero动画。

实现一个点击头像，显示头像原图的功能。先看代码：

import 'package:flutter/material.dart';  
  
class ScaleAnimationRoute extends StatefulWidget {  
 @override  
 \_ScaleAnimationRouteState createState() => new \_ScaleAnimationRouteState();  
}  
  
//需要继承TickerProvider，如果有多个AnimationController，则应该使用TickerProviderStateMixin。  
class \_ScaleAnimationRouteState extends State<ScaleAnimationRoute>  
 with SingleTickerProviderStateMixin {  
  
 initState() {  
 super.initState();  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return new Scaffold(  
 appBar: AppBar(  
 title: Text("动画演示"),  
 ),  
 body: HeroAnimationRoute(),  
 );  
 }  
  
 dispose() {  
 super.dispose();  
 }  
}  
  
// 路由A  
class HeroAnimationRoute extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return Container(  
 alignment: Alignment.*center*,  
 child: InkWell(  
 child: Hero(  
 tag: "avatar", //唯一标记，前后两个路由页Hero的tag必须相同  
 child: ClipOval(  
 child: Image.network("https://ss2.bdstatic.com/70cFvnSh\_Q1YnxGkpoWK1HF6hhy/it/u=1556243087,2666503779&fm=26&gp=0.jpg",  
 width: 100.0,  
 height: 100,  
 ),  
 ),  
 ),  
 onTap: () {  
 //打开B路由  
 Navigator.*push*(context, PageRouteBuilder(  
 pageBuilder: (BuildContext context, Animation animation,  
 Animation secondaryAnimation) {  
 return new FadeTransition(  
 opacity: animation,  
 child: Scaffold(  
 appBar: AppBar(  
 title: Text("原图"),  
 ),  
 body: HeroAnimationRouteB(),  
 ),  
 );  
 })  
 );  
 },  
 ),  
 );  
 }  
}  
  
class HeroAnimationRouteB extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return Center(  
 child: Hero(  
 tag: "avatar", //唯一标记，前后两个路由页Hero的tag必须相同  
 child: Image.network("https://ss2.bdstatic.com/70cFvnSh\_Q1YnxGkpoWK1HF6hhy/it/u=1556243087,2666503779&fm=26&gp=0.jpg"),  
 ),  
 );  
 }  
}

实现Hero动画只需要用Hero组件将要共享的widget包装起来，并提供一个相同的tag即可，中间的过渡帧都是Flutter Framework自动完成的。必须要注意， 前后路由页的共享Hero的tag必须是相同的，Flutter Framework内部正是通过tag来确定新旧路由页widget的对应关系的。

Hero动画的原理比较简单，Flutter Framework知道新旧路由页中共享元素的位置和大小，所以根据这两个端点，在动画执行过程中求出过渡时的插值（中间态）即可。而且Flutter都已经完成，我们无需自己开发。

## 交织动画：

**比如说屏幕左上角A点到屏幕右下角B点的动画移动是由X轴移动动画和Y轴移动动画组成，像这样由多动画组成的复杂动画就是交织动画（**Stagger Animation**）。**

使用交织动画需要注意：

1. 要创建交织动画，需要使用多个动画对象（Animation）。
2. 一个AnimationController控制所有的动画对象。
3. 给每一个动画对象指定时间间隔（Interval）

所有动画都由同一个[AnimationController](https://docs.flutter.io/flutter/animation/AnimationController-class.html)驱动，无论动画需要持续多长时间，控制器的值必须在0.0到1.0之间，而每个动画的间隔（Interval）也必须介于0.0和1.0之间。对于在间隔中设置动画的每个属性，需要分别创建一个[Tween](https://docs.flutter.io/flutter/animation/Tween-class.html) 用于指定该属性的开始值和结束值。也就是说0.0到1.0代表整个动画过程，我们可以给不同动画指定不同的起始点和终止点来决定它们的开始时间和终止时间。

看一个例子：

import 'package:flutter/material.dart';  
  
class StaggerRoute extends StatefulWidget {  
 @override  
 \_StaggerRouteState createState() => \_StaggerRouteState();  
}  
  
class \_StaggerRouteState extends State<StaggerRoute> with TickerProviderStateMixin {  
 AnimationController \_controller;  
  
 @override  
 void initState() {  
 super.initState();  
 \_controller = AnimationController(  
 duration: const Duration(milliseconds: 2000),  
 vsync: this  
 );  
 }  
  
  
 Future<Null> \_playAnimation() async {  
 try {  
 //先正向执行动画  
 await \_controller.forward().orCancel;  
 //再反向执行动画  
 await \_controller.reverse().orCancel;  
 } on TickerCanceled {  
 // the animation got canceled, probably because we were disposed  
 }  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("交织动画"),  
 ),  
 body: GestureDetector(  
 behavior: HitTestBehavior.opaque,  
 onTap: () {  
 \_playAnimation();  
 },  
 child: Center(  
 child: Container(  
 width: 300.0,  
 height: 300.0,  
 decoration: BoxDecoration(  
 color: Colors.*black*.withOpacity(0.1),  
 border: Border.all(  
 color: Colors.*black*.withOpacity(0.5),  
 ),  
 ),  
 //调用我们定义的交织动画Widget  
 child: StaggerAnimation(  
 controller: \_controller  
 ),  
 ),  
 ),  
 ),  
 );  
 }  
}  
  
class StaggerAnimation extends StatelessWidget {  
 StaggerAnimation({ Key key, this.controller }): super(key: key){  
 //高度动画  
 height = Tween<double>(  
 begin:.0 ,  
 end: 300.0,  
 ).animate(  
 CurvedAnimation(  
 parent: controller,  
 curve: Interval(  
 0.0, 0.6, //间隔，前60%的动画时间  
 curve: Curves.*ease*,  
 ),  
 ),  
 );  
  
 color = ColorTween(  
 begin:Colors.*green* ,  
 end:Colors.*red*,  
 ).animate(  
 CurvedAnimation(  
 parent: controller,  
 curve: Interval(  
 0.0, 0.6,//间隔，前60%的动画时间  
 curve: Curves.*ease*,  
 ),  
 ),  
 );  
  
 padding = Tween<EdgeInsets>(  
 begin:EdgeInsets.only(left: .0),  
 end:EdgeInsets.only(left: 100.0),  
 ).animate(  
 CurvedAnimation(  
 parent: controller,  
 curve: Interval(  
 0.6, 1.0, //间隔，后40%的动画时间  
 curve: Curves.*ease*,  
 ),  
 ),  
 );  
 }  
  
  
 final Animation<double> controller;  
 Animation<double> height;  
 Animation<EdgeInsets> padding;  
 Animation<Color> color;  
  
 Widget \_buildAnimation(BuildContext context, Widget child) {  
 return Container(  
 alignment: Alignment.*bottomCenter*,  
 padding:padding.value ,  
 child: Container(  
 color: color.value,  
 width: 50.0,  
 height: height.value,  
 ),  
 );  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return AnimatedBuilder(  
 builder: \_buildAnimation,  
 animation: controller,  
 );  
 }  
}

## 通用“动画切换”组件（AnimatedSwitcher）

AnimatedSwitcher

AnimatedSwitcher可以同时对其新、旧子元素添加显示、隐藏动画。也就是说在AnimatedSwitcher的子元素发生变化时，会对其旧元素和新元素， AnimatedSwitcher的定义：

const AnimatedSwitcher({

Key key,

this.child,

@required this.duration, // 新child显示动画时长

this.reverseDuration,// 旧child隐藏的动画时长

this.switchInCurve = Curves.linear, // 新child显示的动画曲线

this.switchOutCurve = Curves.linear,// 旧child隐藏的动画曲线

this.transitionBuilder = AnimatedSwitcher.defaultTransitionBuilder, // 动画构建器

this.layoutBuilder = AnimatedSwitcher.defaultLayoutBuilder, //布局构建器

})

当AnimatedSwitcher的child发生变化时（类型或Key不同），旧child会执行隐藏动画，新child会执行执行显示动画。究竟执行何种动画效果则由transitionBuilder参数决定，该参数接受一个AnimatedSwitcherTransitionBuilder类型的builder，定义如下：

typedef AnimatedSwitcherTransitionBuilder =

Widget Function(Widget child, Animation<double> animation);

该builder在AnimatedSwitcher的child切换时会分别对新、旧child绑定动画：

1. 对旧child，绑定的动画会反向执行（reverse）
2. 对新child，绑定的动画会正向指向（forward）

这样一下，便实现了对新、旧child的动画绑定。AnimatedSwitcher的默认值是AnimatedSwitcher.defaultTransitionBuilder：默认情况为渐隐渐现

Widget defaultTransitionBuilder(Widget child, Animation<double> animation) {

return FadeTransition(

opacity: animation,

child: child,

);

}

看一个简单的例子：

import 'package:flutter/material.dart';  
  
class AnimatedSwitcherCounterRoute extends StatefulWidget {  
 const AnimatedSwitcherCounterRoute({Key key}) : super(key: key);  
  
 @override  
 \_AnimatedSwitcherCounterRouteState createState() => \_AnimatedSwitcherCounterRouteState();  
}  
  
class \_AnimatedSwitcherCounterRouteState extends State<AnimatedSwitcherCounterRoute> {  
 int \_count = 0;  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("看看效果"),  
 ),  
 body: Center(  
 child: Column(  
 mainAxisAlignment: MainAxisAlignment.center,  
 children: <Widget>[  
 AnimatedSwitcher(  
 duration: const Duration(milliseconds: 500),  
 transitionBuilder: (Widget child, Animation<double> animation) {  
 //执行旋转动画  
 return RotationTransition(child: child, turns: animation);  
 },  
 child: Text(  
 '$\_count',  
 //显示指定key，不同的key会被认为是不同的Text，这样才能执行动画  
 key: ValueKey<int>(\_count),  
 style: Theme.*of*(context).textTheme.headline,  
 ),  
 ),  
 RaisedButton(  
 child: const Text('+1',),  
 onPressed: () {  
 setState(() {  
 \_count += 1;  
 });  
 },  
 ),  
 ],  
 ),  
 ),  
 );  
 }  
}

## Flutter预置的动画过渡组件

| **组件名** | **功能** |
| --- | --- |
| AnimatedPadding | 在padding发生变化时会执行过渡动画到新状态 |
| AnimatedPositioned | 配合Stack一起使用，当定位状态发生变化时会执行过渡动画到新的状态。 |
| AnimatedOpacity | 在透明度opacity发生变化时执行过渡动画到新状态 |
| AnimatedAlign | 当alignment发生变化时会执行过渡动画到新的状态。 |
| AnimatedContainer | 当Container属性发生变化时会执行过渡动画到新的状态。 |
| AnimatedDefaultTextStyle | 当字体样式发生变化时，子组件中继承了该样式的文本组件会动态过渡到新样式。 |

# 自绘组件 （CustomPaint与Canvas）

写在前面的话：自定义组件大致可以分为三种：通过组合其它组件、自绘和实现RenderObject。组合其他组件就是把各个组件进行合理组合使用，这里就不做赘述。实现RenderObject有些复杂，在后面的Flutter再具体阐述。

几乎所有的UI系统都会提供一个自绘UI的接口，这个接口通常会提供一块2D画布Canvas，Canvas内部封装了一些基本绘制的API，开发者可以通过Canvas绘制各种自定义图形。在Flutter中，提供了一个CustomPaint组件，它可以结合画笔CustomPainter来实现自定义图形绘制。

CustomPaint

CustomPaint 构造函数：

CustomPaint({

Key key,

this.painter, //背景画笔，会显示在子节点后面;

this.foregroundPainter, //前景画笔，会显示在子节点前面

this.size = Size.zero, //当child为null时，代表默认绘制区域大小，如果有child则忽略此参数，画布尺寸则为child尺寸。如果有child但是想指定画布为特定大小，可以使用SizeBox包裹CustomPaint实现。

this.isComplex = false, //是否复杂的绘制，如果是，Flutter会应用一些缓存策略来减少重复渲染的开销。

this.willChange = false, //和isComplex配合使用，当启用缓存时，该属性代表在下一帧中绘制是否会改变。

Widget child, //子节点，可以为空

})

注意：

如果CustomPaint有子节点，为了避免子节点不必要的重绘并提高性能，通常情况下都会将子节点包裹在RepaintBoundary组件中，这样会在绘制时就会创建一个新的绘制层（Layer），其子组件将在新的Layer上绘制，而父组件将在原来Layer上绘制，也就是说RepaintBoundary子组件的绘制将独立于父组件的绘制，RepaintBoundary会隔离其子节点和CustomPaint本身的绘制边界。示例如下：

CustomPaint(

size: Size(300, 300), //指定画布大小

painter: MyPainter(),

child: RepaintBoundary(child:...)),

)

CustomPainter

CustomPainter中提定义了一个虚函数paint：

void paint(Canvas canvas, Size size);

paint有两个参数:

* Canvas：一个画布，包括各种绘制方法，我们列出一下常用的方法：

|  |  |
| --- | --- |
| **API名称** | **功能** |
| drawLine | 画线 |
| drawPoint | 画点 |
| drawPath | 画路径 |
| drawImage | 画图像 |
| drawRect | 画矩形 |
| drawCircle | 画圆 |
| drawOval | 画椭圆 |
| drawArc | 画圆弧 |

* Size：当前绘制区域大小。

画笔Paint

Flutter提供了Paint类来实现画笔。在Paint中，我们可以配置画笔的各种属性如粗细、颜色、样式等。如：

var paint = Paint() //创建一个画笔并配置其属性

..isAntiAlias = true //是否抗锯齿

..style = PaintingStyle.fill //画笔样式：填充

..color=Color(0x77cdb175);//画笔颜色

综上，我们实现一个五子棋盘的demo

import 'package:flutter/material.dart';  
import 'dart:math';  
  
class CustomPaintRoute extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("五子棋盘"),  
 ),  
 body:Center(  
 child: CustomPaint(  
 size: Size(300, 300), //指定画布大小  
 painter: MyPainter(),  
 ),  
 )  
 );  
 }  
}  
  
class MyPainter extends CustomPainter {  
 @override  
 void paint(Canvas canvas, Size size) {  
 double eWidth = size.width / 15;  
 double eHeight = size.height / 15;  
  
 //画棋盘背景  
 var paint = Paint()  
 ..isAntiAlias = true  
 ..style = PaintingStyle.fill //填充  
 ..color = Color(0x77cdb175); //背景为纸黄色  
 canvas.drawRect(Offset.*zero* & size, paint);  
  
 //画棋盘网格  
 paint  
 ..style = PaintingStyle.stroke //线  
 ..color = Colors.*black87* ..strokeWidth = 1.0;  
  
 for (int i = 0; i <= 15; ++i) {  
 double dy = eHeight \* i;  
 canvas.drawLine(Offset(0, dy), Offset(size.width, dy), paint);  
 }  
  
 for (int i = 0; i <= 15; ++i) {  
 double dx = eWidth \* i;  
 canvas.drawLine(Offset(dx, 0), Offset(dx, size.height), paint);  
 }  
  
 //画一个黑子  
 paint  
 ..style = PaintingStyle.fill  
 ..color = Colors.*black*;  
 canvas.drawCircle(  
 Offset(size.width / 2 - eWidth / 2, size.height / 2 - eHeight / 2),  
 min(eWidth / 2, eHeight / 2) - 2,  
 paint,  
 );  
  
 //画一个白子  
 paint.color = Colors.*white*;  
 canvas.drawCircle(  
 Offset(size.width / 2 + eWidth / 2, size.height / 2 - eHeight / 2),  
 min(eWidth / 2, eHeight / 2) - 2,  
 paint,  
 );  
 }  
  
 //在实际场景中正确利用此回调可以避免重绘开销，本示例我们简单的返回true  
 @override  
 bool shouldRepaint(CustomPainter oldDelegate) => true;  
}

性能

绘制是比较昂贵的操作，所以我们在实现自绘控件时应该考虑到性能开销，下面是两条关于性能优化的建议：

* 尽可能的利用好shouldRepaint返回值；在UI树重新build时，控件在绘制前都会先调用该方法以确定是否有必要重绘；假如我们绘制的UI不依赖外部状态，那么就应该始终返回false，因为外部状态改变导致重新build时不会影响我们的UI外观；如果绘制依赖外部状态，那么我们就应该在shouldRepaint中判断依赖的状态是否改变，如果已改变则应返回true来重绘，反之则应返回false不需要重绘。
* 绘制尽可能多的分层；在上面五子棋的示例中，我们将棋盘和棋子的绘制放在了一起，这样会有一个问题：由于棋盘始终是不变的，用户每次落子时变的只是棋子，但是如果按照上面的代码来实现，每次绘制棋子时都要重新绘制一次棋盘，这是没必要的。优化的方法就是将棋盘单独抽为一个组件，并设置其shouldRepaint回调值为false，然后将棋盘组件作为背景。然后将棋子的绘制放到另一个组件中，这样每次落子时只需要绘制棋子。
  + 简单优化
* import 'package:flutter/material.dart';  
  import 'dart:math';  
    
  class CustomPaintRoute extends StatelessWidget {  
   @override  
   Widget build(BuildContext context) {  
   return Scaffold(  
   appBar: AppBar(  
   title: Text("五子棋盘"),  
   ),  
   body: Center(  
   child: CustomPaint(  
   size: Size(300, 300), //指定画布大小  
   painter: MyPainter(),  
   foregroundPainter: Mypiece(),  
   ),  
   ));  
   }  
  }  
    
  class MyPainter extends CustomPainter {  
   @override  
   void paint(Canvas canvas, Size size) {  
   double eWidth = size.width / 15;  
   double eHeight = size.height / 15;  
    
   //画棋盘背景  
   var paint = Paint()  
   ..isAntiAlias = true  
   ..style = PaintingStyle.fill //填充  
   ..color = Color(0x77cdb175); //背景为纸黄色  
   canvas.drawRect(Offset.*zero* & size, paint);  
    
   //画棋盘网格  
   paint  
   ..style = PaintingStyle.stroke //线  
   ..color = Colors.*black87* ..strokeWidth = 1.0;  
    
   for (int i = 0; i <= 15; ++i) {  
   double dy = eHeight \* i;  
   canvas.drawLine(Offset(0, dy), Offset(size.width, dy), paint);  
   }  
    
   for (int i = 0; i <= 15; ++i) {  
   double dx = eWidth \* i;  
   canvas.drawLine(Offset(dx, 0), Offset(dx, size.height), paint);  
   }  
   }  
    
   //在实际场景中正确利用此回调可以避免重绘开销，本示例我们简单的返回true  
   @override  
   bool shouldRepaint(CustomPainter oldDelegate) => false;  
  }  
    
  class Mypiece extends CustomPainter {  
   @override  
   void paint(Canvas canvas, Size size) {  
   double eWidth = size.width / 15;  
   double eHeight = size.height / 15;  
    
   //画一个黑子  
   var paint = Paint()  
   ..style = PaintingStyle.fill  
   ..color = Colors.*black*;  
   canvas.drawCircle(  
   Offset(size.width / 2 - eWidth / 2, size.height / 2 - eHeight / 2),  
   min(eWidth / 2, eHeight / 2) - 2,  
   paint,  
   );  
    
   //画一个白子  
   paint.color = Colors.*white*;  
   canvas.drawCircle(  
   Offset(size.width / 2 + eWidth / 2, size.height / 2 - eHeight / 2),  
   min(eWidth / 2, eHeight / 2) - 2,  
   paint,  
   );  
   }  
    
   //在实际场景中正确利用此回调可以避免重绘开销，本示例我们简单的返回true  
   @override  
   bool shouldRepaint(CustomPainter oldDelegate) => true;  
  }

# 自绘实例：圆形背景渐变进度条

第一步：绘制环形组件

import 'dart:math';  
import 'package:flutter/material.dart';  
  
class GradientCircularProgressIndicator extends StatelessWidget {  
 GradientCircularProgressIndicator({  
 this.strokeWidth = 2.0,  
 @required this.radius,  
 @required this.colors,  
 this.stops,  
 this.strokeCapRound = false,  
 this.backgroundColor = const Color(0xFFEEEEEE),  
 this.totalAngle = 2 \* pi,  
 this.value  
 });  
  
 *///粗细* final double strokeWidth;  
  
 */// 圆的半径* final double radius;  
  
 *///两端是否为圆角* final bool strokeCapRound;  
  
 */// 当前进度，取值范围 [0.0-1.0]* final double value;  
  
 */// 进度条背景色* final Color backgroundColor;  
  
 */// 进度条的总弧度，2\*PI为整圆，小于2\*PI则不是整圆* final double totalAngle;  
  
 */// 渐变色数组* final List<Color> colors;  
  
 */// 渐变色的终止点，对应colors属性* final List<double> stops;  
  
 @override  
 Widget build(BuildContext context) {  
 double \_offset = .0;  
 // 如果两端为圆角，则需要对起始位置进行调整，否则圆角部分会偏离起始位置  
 // 下面调整的角度的计算公式是通过数学几何知识得出，读者有兴趣可以研究一下为什么是这样  
 if (strokeCapRound) {  
 \_offset = asin(strokeWidth / (radius \* 2 - strokeWidth));  
 }  
 var \_colors = colors;  
 if (\_colors == null) {  
 Color color = Theme  
 .*of*(context)  
 .accentColor;  
 \_colors = [color, color];  
 }  
 return Transform.rotate(  
 angle: -pi / 2.0 - \_offset,  
 child: CustomPaint(  
 size: Size.fromRadius(radius),  
 painter: \_GradientCircularProgressPainter(  
 strokeWidth: strokeWidth,  
 strokeCapRound: strokeCapRound,  
 backgroundColor: backgroundColor,  
 value: value,  
 total: totalAngle,  
 radius: radius,  
 colors: \_colors,  
 )  
 ),  
 );  
 }  
}  
  
//实现画笔  
class \_GradientCircularProgressPainter extends CustomPainter {  
 \_GradientCircularProgressPainter({  
 this.strokeWidth: 10.0,  
 this.strokeCapRound: false,  
 this.backgroundColor = const Color(0xFFEEEEEE),  
 this.radius,  
 this.total = 2 \* pi,  
 @required this.colors,  
 this.stops,  
 this.value  
 });  
  
 final double strokeWidth;  
 final bool strokeCapRound;  
 final double value;  
 final Color backgroundColor;  
 final List<Color> colors;  
 final double total;  
 final double radius;  
 final List<double> stops;  
  
 @override  
 void paint(Canvas canvas, Size size) {  
 if (radius != null) {  
 size = Size.fromRadius(radius);  
 }  
 double \_offset = strokeWidth / 2.0;  
 double \_value = (value ?? .0);  
 \_value = \_value.clamp(.0, 1.0) \* total;  
 double \_start = .0;  
  
 if (strokeCapRound) {  
 \_start = asin(strokeWidth/ (size.width - strokeWidth));  
 }  
  
 Rect rect = Offset(\_offset, \_offset) & Size(  
 size.width - strokeWidth,  
 size.height - strokeWidth  
 );  
  
 var paint = Paint()  
 ..strokeCap = strokeCapRound ? StrokeCap.round : StrokeCap.butt  
 ..style = PaintingStyle.stroke  
 ..isAntiAlias = true  
 ..strokeWidth = strokeWidth;  
  
 // 先画背景  
 if (backgroundColor != Colors.*transparent*) {  
 paint.color = backgroundColor;  
 canvas.drawArc(  
 rect,  
 \_start,  
 total,  
 false,  
 paint  
 );  
 }  
  
 // 再画前景，应用渐变  
 if (\_value > 0) {  
 paint.shader = SweepGradient(  
 startAngle: 0.0,  
 endAngle: \_value,  
 colors: colors,  
 stops: stops,  
 ).createShader(rect);  
  
 canvas.drawArc(  
 rect,  
 \_start,  
 \_value,  
 false,  
 paint  
 );  
 }  
 }  
  
 @override  
 bool shouldRepaint(CustomPainter oldDelegate) => true;  
  
}

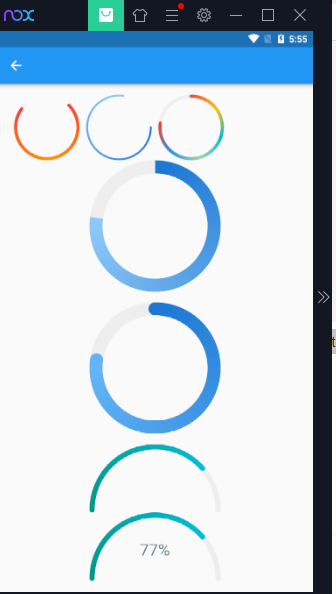
第二步：组合实例TurnBox

import 'package:flutter/widgets.dart';  
  
class TurnBox extends StatefulWidget {  
 const TurnBox({  
 Key key,  
 this.turns = .0, //旋转的“圈”数,一圈为360度，如0.25圈即90度  
 this.speed = 200, //过渡动画执行的总时长  
 this.child  
 }) :super(key: key);  
  
 final double turns;  
 final int speed;  
 final Widget child;  
  
 @override  
 \_TurnBoxState createState() => new \_TurnBoxState();  
}  
  
class \_TurnBoxState extends State<TurnBox>  
 with SingleTickerProviderStateMixin {  
 AnimationController \_controller;  
  
 @override  
 void initState() {  
 super.initState();  
 \_controller = new AnimationController(  
 vsync: this,  
 lowerBound: -double.*infinity*,  
 upperBound: double.*infinity* );  
 \_controller.value = widget.turns;  
 }  
  
 @override  
 void dispose() {  
 \_controller.dispose();  
 super.dispose();  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return RotationTransition(  
 turns: \_controller,  
 child: widget.child,  
 );  
 }  
  
 @override  
 void didUpdateWidget(TurnBox oldWidget) {  
 super.didUpdateWidget(oldWidget);  
 //旋转角度发生变化时执行过渡动画  
 if (oldWidget.turns != widget.turns) {  
 \_controller.animateTo(  
 widget.turns,  
 duration: Duration(milliseconds: widget.speed??200),  
 curve: Curves.*easeOut*,  
 );  
 }  
 }  
}

第三步：调用

import 'dart:math';  
import 'package:flutter/material.dart';  
import 'test.dart';  
import 'AAA.dart';  
  
class GradientCircularProgressRoute extends StatefulWidget {  
 @override  
 GradientCircularProgressRouteState createState() {  
 return new GradientCircularProgressRouteState();  
 }  
}  
  
class GradientCircularProgressRouteState  
 extends State<GradientCircularProgressRoute> with TickerProviderStateMixin {  
 AnimationController \_animationController;  
  
 @override  
 void initState() {  
 super.initState();  
 \_animationController =  
 new AnimationController(vsync: this, duration: Duration(seconds: 3));  
 bool isForward = true;  
 \_animationController.addStatusListener((status) {  
 if (status == AnimationStatus.forward) {  
 isForward = true;  
 } else if (status == AnimationStatus.completed ||  
 status == AnimationStatus.dismissed) {  
 if (isForward) {  
 \_animationController.reverse();  
 } else {  
 \_animationController.forward();  
 }  
 } else if (status == AnimationStatus.reverse) {  
 isForward = false;  
 }  
 });  
 \_animationController.forward();  
 }  
  
 @override  
 void dispose() {  
 \_animationController.dispose();  
 super.dispose();  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(),  
 body: SingleChildScrollView(  
 child: Center(  
 child: Column(  
 crossAxisAlignment: CrossAxisAlignment.center,  
 children: <Widget>[  
 AnimatedBuilder(  
 animation: \_animationController,  
 builder: (BuildContext context, Widget child) {  
 return Padding(  
 padding: const EdgeInsets.symmetric(vertical: 16.0),  
 child: Column(  
 children: <Widget>[  
 Wrap(  
 spacing: 10.0,  
 runSpacing: 16.0,  
 children: <Widget>[  
 GradientCircularProgressIndicator(  
 // No gradient  
 colors: [Colors.*blue*, Colors.*blue*],  
 radius: 50.0,  
 strokeWidth: 3.0,  
 value: \_animationController.value,  
 ),  
 GradientCircularProgressIndicator(  
 colors: [Colors.*red*, Colors.*orange*],  
 radius: 50.0,  
 strokeWidth: 3.0,  
 value: \_animationController.value,  
 ),  
 GradientCircularProgressIndicator(  
 colors: [Colors.*red*, Colors.*orange*, Colors.*red*],  
 radius: 50.0,  
 strokeWidth: 5.0,  
 value: \_animationController.value,  
 ),  
 GradientCircularProgressIndicator(  
 colors: [Colors.*teal*, Colors.*cyan*],  
 radius: 50.0,  
 strokeWidth: 5.0,  
 strokeCapRound: true,  
 value: CurvedAnimation(  
 parent: \_animationController,  
 curve: Curves.*decelerate*)  
 .value,  
 ),  
 TurnBox(  
 turns: 1 / 8,  
 child: GradientCircularProgressIndicator(  
 colors: [Colors.*red*, Colors.*orange*, Colors.*red*],  
 radius: 50.0,  
 strokeWidth: 5.0,  
 strokeCapRound: true,  
 backgroundColor: Colors.*red*[50],  
 totalAngle: 1.5 \* pi,  
 value: CurvedAnimation(  
 parent: \_animationController,  
 curve: Curves.*ease*)  
 .value),  
 ),  
 RotatedBox(  
 quarterTurns: 1,  
 child: GradientCircularProgressIndicator(  
 colors: [Colors.*blue*[700], Colors.*blue*[200]],  
 radius: 50.0,  
 strokeWidth: 3.0,  
 strokeCapRound: true,  
 backgroundColor: Colors.*transparent*,  
 value: \_animationController.value),  
 ),  
 GradientCircularProgressIndicator(  
 colors: [  
 Colors.*red*,  
 Colors.*amber*,  
 Colors.*cyan*,  
 Colors.*green*[200],  
 Colors.*blue*,  
 Colors.*red* ],  
 radius: 50.0,  
 strokeWidth: 5.0,  
 strokeCapRound: true,  
 value: \_animationController.value,  
 ),  
 ],  
 ),  
 GradientCircularProgressIndicator(  
 colors: [Colors.*blue*[700], Colors.*blue*[200]],  
 radius: 100.0,  
 strokeWidth: 20.0,  
 value: \_animationController.value,  
 ),  
  
 Padding(  
 padding: const EdgeInsets.symmetric(vertical: 16.0),  
 child: GradientCircularProgressIndicator(  
 colors: [Colors.*blue*[700], Colors.*blue*[300]],  
 radius: 100.0,  
 strokeWidth: 20.0,  
 value: \_animationController.value,  
 strokeCapRound: true,  
 ),  
 ),  
 //剪裁半圆  
 ClipRect(  
 child: Align(  
 alignment: Alignment.*topCenter*,  
 heightFactor: .5,  
 child: Padding(  
 padding: const EdgeInsets.only(bottom: 8.0),  
 child: SizedBox(  
 //width: 100.0,  
 child: TurnBox(  
 turns: .75,  
 child: GradientCircularProgressIndicator(  
 colors: [Colors.*teal*, Colors.*cyan*[500]],  
 radius: 100.0,  
 strokeWidth: 8.0,  
 value: \_animationController.value,  
 totalAngle: pi,  
 strokeCapRound: true,  
 ),  
 ),  
 ),  
 ),  
 ),  
 ),  
 SizedBox(  
 height: 104.0,  
 width: 200.0,  
 child: Stack(  
 alignment: Alignment.*center*,  
 children: <Widget>[  
 Positioned(  
 height: 200.0,  
 top: .0,  
 child: TurnBox(  
 turns: .75,  
 child: GradientCircularProgressIndicator(  
 colors: [Colors.*teal*, Colors.*cyan*[500]],  
 radius: 100.0,  
 strokeWidth: 8.0,  
 value: \_animationController.value,  
 totalAngle: pi,  
 strokeCapRound: true,  
 ),  
 ),  
 ),  
 Padding(  
 padding: const EdgeInsets.only(top: 10.0),  
 child: Text(  
 "${(\_animationController.value \* 100).toInt()}%",  
 style: TextStyle(  
 fontSize: 25.0,  
 color: Colors.*blueGrey*,  
 ),  
 ),  
 )  
 ],  
 ),  
 ),  
 ],  
 ),  
 );  
 },  
 ),  
 ],  
 ),  
 ),  
 ),  
 );  
 }  
}

上效果（代码可直接运行）：



# 文件操作

Android和iOS的应用存储目录不同，[PathProvider](https://pub.dartlang.org/packages/path_provider)插件提供了一种平台透明的方式来访问设备文件系统上的常用位置。该类当前支持访问两个文件系统位置：

* **临时目录:**可以使用getTemporaryDirectory()来获取临时目录； 系统可随时清除的临时目录（缓存）。在iOS上，这对应于[NSTemporaryDirectory()](https://developer.apple.com/reference/foundation/1409211-nstemporarydirectory)返回的值。在Android上，这是[getCacheDir()](https://developer.android.com/reference/android/content/Context.html#getCacheDir()返回的值。
* **文档目录:**可以使用getApplicationDocumentsDirectory()来获取应用程序的文档目录，该目录用于存储只有自己可以访问的文件。只有当应用程序被卸载时，系统才会清除该目录。在iOS上，这对应于NSDocumentDirectory。在Android上，这是AppData目录。
* **外部存储目录**：可以使用getExternalStorageDirectory()来获取外部存储目录，如SD卡；由于iOS不支持外部目录，所以在iOS下调用该方法会抛出UnsupportedError异常，而在Android下结果是android SDK中getExternalStorageDirectory的返回值。

# Http

## Http请求-HttpClient

Dart IO库中提供了用于发起Http请求的一些类，我们可以直接使用HttpClient来发起请求。使用HttpClient发起请求分为5步：

1.创建HttpClient：HttpClient = new HttpClient();

2.打开Http连接，设置请求头：

HttpClientRequest request = await httpClient.getUrl(uri)

这一步可以使用任意Http Method,如httpClient.post()、httpClient.delete()等。如果包含Query参数，可以在构建uri时添加，如：

Uri uri=Uri(scheme: "https", host: "flutterchina.club", queryParameters: {

"xx":"xx",

"yy":"dd"

});

通过HttpClientRequest可以直接设置请求header，如：

request.headers.add(“user-agent”,”test”)

如果是post或put等可以携带请求体方法，可以通过HttpClientRequest对象发送request body，如

String payload="...";

request.add(utf8.encode(payload));

//request.addStream(\_inputStream); //可以直接添加输入流

3．等待服务器连接

HttpClientResponse response = await request.close();

这一步完成后，请求信息就已经发送给服务器了，返回一个HttpClientResponse对象，它包含响应头（header）和响应流（响应体的Stream），接下来就可以通过读取响应流来获取响应内容。

4.读取响应内容

String responseBody = await response.transform(utf8.decoder).join();

我们通过读取响应流来获取服务器返回的数据，在读取时我们可以设置编码格式，这里是utf8。

5.请求结束，关闭HttpClient：

httpClient.close();

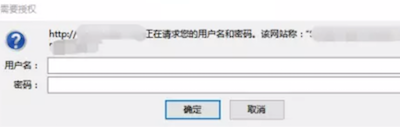
关闭client后，通过该client发起的所有请求都会中止。

HttpClient配置

| **属性** | **含义** |
| --- | --- |
| idleTimeout | 对应请求头中的keep-alive字段值，为了避免频繁建立连接，httpClient在请求结束后会保持连接一段时间，超过这个阈值后才会关闭连接。 |
| connectionTimeout | 和服务器建立连接的超时，如果超过这个值则会抛出SocketException异常。 |
| maxConnectionsPerHost | 同一个host，同时允许建立连接的最大数量。 |
| autoUncompress | 对应请求头中的Content-Encoding，如果设置为true，则请求头中Content-Encoding的值为当前HttpClient支持的压缩算法列表，目前只有"gzip" |
| userAgent | 对应请求头中的User-Agent字段。 |

HTTP请求认证

Http协议的认证（Authentication）机制可以用于保护非公开资源。如果Http服务器开启了认证，那么用户在发起请求时就需要携带用户凭据，如果你在浏览器中访问了启用Basic认证的资源时，浏览就会弹出一个登录框，如：



我们先看看Basic认证的基本过程：

1.客户端发送http请求给服务器，服务器验证该用户是否已经登录验证过了，如果没有的话， 服务器会返回一个401 Unauthozied给客户端，并且在响应header中添加一个 “WWW-Authenticate” 字段，例如：

WWW-Authenticate: Basic realm="admin"

其中"Basic"为认证方式，realm为用户角色的分组，可以在后台添加分组。

2.客户端得到响应码后，将用户名和密码进行base64编码（格式为用户名:密码），设置请求头Authorization，继续访问

Authorization: Basic YXXFISDJFISJFGIJIJG

服务器验证用户凭据，如果通过就返回资源内容。

注意，Http的方式除了Basic认证之外还有：Digest认证、Client认证、Form Based认证等，目前Flutter的HttpClient只支持Basic和Digest两种认证方式，这两种认证方式最大的区别是发送用户凭据时，对于用户凭据的内容，前者只是简单的通过Base64编码（可逆），而后者会进行哈希运算，相对来说安全一点点，但是为了安全起见，**无论是采用Basic认证还是Digest认证，都应该在Https协议下**，这样可以防止抓包和中间人攻击。

HttpClient关于Http认证的方法和属性：

1. addCredentials(Uri url, String realm, HttpClientCredentials credentials)

该方法用于添加用户凭据,如：

httpClient.addCredentials(\_uri,

"admin",

new HttpClientBasicCredentials("username","password"), //Basic认证凭据

);

如果是Digest认证，可以创建Digest认证凭据：

HttpClientDigestCredentials("username","password")

2.authenticate(Future<bool> f(Uri url, String scheme, String realm))

这是一个setter，类型是一个回调，当服务器需要用户凭据且该用户凭据未被添加时，httpClient会调用此回调，在这个回调当中，一般会调用addCredential()来动态添加用户凭证，例如：

httpClient.authenticate=(Uri url, String scheme, String realm) async{

if(url.host=="xx.com" && realm=="admin"){

httpClient.addCredentials(url,

"admin",

new HttpClientBasicCredentials("username","pwd"),

);

return true;

}

return false;

};

一个建议是，如果所有请求都需要认证，那么应该在HttpClient初始化时就调用addCredentials()来添加全局凭证，而不是去动态添加。

代理

可以通过findProxy来设置代理策略，例如，我们要将所有请求通过代理服务器（192.168.1.2:8888）发送出去：

client.findProxy = (uri) {

// 如果需要过滤uri，可以手动判断

return "PROXY 192.168.1.2:8888";

};

findProx回调返回值是一个遵循浏览器PAC脚本格式的字符串，详情可以查看API文档，如果不需要代理，返回"DIRECT"即可。

在APP开发中，很多时候我们需要抓包来调试，而抓包软件(如charles)就是一个代理，这时我们就可以将请求发送到我们的抓包软件，我们就可以在抓包软件中看到请求的数据了。

有时代理服务器也启用了身份验证，这和http协议的认证是相似的，HttpClient提供了对应的Proxy认证方法和属性：

set authenticateProxy(

Future<bool> f(String host, int port, String scheme, String realm));

void addProxyCredentials(

String host, int port, String realm, HttpClientCredentials credentials);

他们的使用方法和上面“HTTP请求认证”一节中介绍的addCredentials和authenticate相同，故不再赘述。

证书校验

Https中为了防止通过伪造证书而发起的中间人攻击，客户端应该对自签名或非CA颁发的证书进行校验。HttpClient对证书校验的逻辑如下：

1. 如果请求的Https证书是可信CA颁发的，并且访问host包含在证书的domain列表中(或者符合通配规则)并且证书未过期，则验证通过。
2. 如果第一步验证失败，但在创建HttpClient时，已经通过SecurityContext将证书添加到证书信任链中，那么当服务器返回的证书在信任链中的话，则验证通过。
3. 如果1、2验证都失败了，如果用户提供了badCertificateCallback回调，则会调用它，如果回调返回true，则允许继续链接，如果返回false，则终止链接。

综上所述，我们的证书校验其实就是提供一个badCertificateCallback回调，下面通过一个示例来说明。

示例：

假设我们的后台服务使用的是自签名证书，证书格式是PEM格式，我们将证书的内容保存在本地字符串中，那么我们的校验逻辑如下：

String PEM="XXXXX";//可以从文件读取

...

httpClient.badCertificateCallback=(X509Certificate cert, String host, int port){

if(cert.pem==PEM){

return true; //证书一致，则允许发送数据

}

return false;

};

X509Certificate是证书的标准格式，包含了证书除私钥外所有信息，读者可以自行查阅文档。另外，上面的示例没有校验host，是因为只要服务器返回的证书内容和本地的保存一致就已经能证明是我们的服务器了（而不是中间人），host验证通常是为了防止证书和域名不匹配。

对于自签名的证书，我们也可以将其添加到本地证书信任链中，这样证书验证时就会自动通过，而不会再走到badCertificateCallback回调中：

SecurityContext sc=new SecurityContext();

//file为证书路径

sc.setTrustedCertificates(file);

//创建一个HttpClient

HttpClient httpClient = new HttpClient(context: sc);

注意，通过setTrustedCertificates()设置的证书格式必须为PEM或PKCS12，如果证书格式为PKCS12，则需将证书密码传入，这样则会在代码中暴露证书密码，所以客户端证书校验不建议使用PKCS12格式的证书。

写在最后的话：值得注意的是，HttpClient提供的这些属性和方法最终都会作用在请求header里，我们完全可以通过手动去设置header来实现，之所以提供这些方法，只是为了方便开发者而已。另外，Http协议是一个非常重要的、使用最多的网络协议，每一个开发者都应该对http协议非常熟悉。

## Http请求-Dio http库

dio是一个强大的Dart Http请求库，支持Restful API、FormData、拦截器、请求取消、Cookie管理、文件上传/下载、超时等。dio的使用方式随着其版本升级可能会发生变化，如果本节所述内容和dio官方有差异，请以dio官方文档为准。

引入：

dependencies:

dio: ^x.x.x #请使用pub上的最新版本

导入并创建dio实例：

import 'package:dio/dio.dart';

Dio dio = Dio();

接下来就可以通过 dio实例来发起网络请求了，注意，一个dio实例可以发起多个http请求，一般来说，APP只有一个http数据源时，dio应该使用单例模式。

示例：

GET请求

Response response;

response=await dio.get("/test?id=12&name=wendu")

print(response.data.toString());

对于GET请求我们可以将query参数通过对象来传递，上面的代码等同于：

response=await dio.get("/test",queryParameters:{"id":12,"name":"wendu"})

print(response);

POST请求

response=await dio.post("/test",data:{"id":12,"name":"wendu"})

发起多个并发请求:

response= await Future.wait([dio.post("/info"),dio.get("/token")]);

下载文件:

response=await dio.download("https://www.google.com/",\_savePath);

发送 FormData:

FormData formData = new FormData.from({

"name": "wendux",

"age": 25,

});

response = await dio.post("/info", data: formData)

如果发送的数据是FormData，则dio会将请求header的contentType设为“multipart/form-data”。

通过FormData上传多个文件:

FormData formData = new FormData.from({

"name": "wendux",

"age": 25,

"file1": new UploadFileInfo(new File("./upload.txt"), "upload1.txt"),

"file2": new UploadFileInfo(new File("./upload.txt"), "upload2.txt"),

// 支持文件数组上传

"files": [

new UploadFileInfo(new File("./example/upload.txt"), "upload.txt"),

new UploadFileInfo(new File("./example/upload.txt"), "upload.txt")

]

});

response = await dio.post("/info", data: formData)

值得一提的是，dio内部仍然使用HttpClient发起的请求，所以代理、请求认证、证书校验等和HttpClient是相同的，我们可以在onHttpClientCreate回调中设置，例如：

(dio.httpClientAdapter as DefaultHttpClientAdapter).onHttpClientCreate = (client) {

//设置代理

client.findProxy = (uri) {

return "PROXY 192.168.1.2:8888";

};

//校验证书

httpClient.badCertificateCallback=(X509Certificate cert, String host, int port){

if(cert.pem==PEM){

return true; //证书一致，则允许发送数据

}

return false;

};

};

注意，onHttpClientCreate会在当前dio实例内部需要创建HttpClient时调用，所以通过此回调配置HttpClient会对整个dio实例生效，如果你想针对某个应用请求单独的代理或证书校验策略，可以创建一个新的dio实例即可。

除了这些基本的用法，dio还支持请求配置、拦截器等，官方资料比较详细，故不再赘述，详情可以参考dio主页：<https://github.com/flutterchina/dio>

实例

我们通过Github开放的API来请求flutterchina组织下的所有公开的开源项目，实现：

1. 在请求阶段弹出loading
2. 请求结束后，如果请求失败，则展示错误信息；如果成功，则将项目名称列表展示出来。

class \_FutureBuilderRouteState extends State<FutureBuilderRoute> {  
 Dio \_dio = new Dio();  
  
 @override  
 Widget build(BuildContext context) {  
  
 return new Container(  
 alignment: Alignment.*center*,  
 child: FutureBuilder(  
 future: \_dio.get("https://api.github.com/orgs/flutterchina/repos"),  
 builder: (BuildContext context, AsyncSnapshot snapshot) {  
 //请求完成  
 if (snapshot.connectionState == ConnectionState.done) {  
 Response response = snapshot.data;  
 //发生错误  
 if (snapshot.hasError) {  
 return Text(snapshot.error.toString());  
 }  
 //请求成功，通过项目信息构建用于显示项目名称的ListView  
 return ListView(  
 children: response.data.map<Widget>((e) =>  
 ListTile(title: Text(e["full\_name"]))  
 ).toList(),  
 );  
 }  
 //请求未完成时弹出loading  
 return CircularProgressIndicator();  
 }  
 ),  
 );  
 }  
}

## Http分块下载：

**原理**

Http协议定义了分块传输的响应header字段，但具体是否支持取决于Server实现，可以通过指定请求头的“range”字段来验证服务器是否支持分块传输。例如，我们可以李永峰curl命令来验证：

bogon:~ duwen$ curl -H "Range: bytes=0-10" http://download.dcloud.net.cn/HBuilder.9.0.2.macosx\_64.dmg -v

# 请求头

> GET /HBuilder.9.0.2.macosx\_64.dmg HTTP/1.1

> Host: download.dcloud.net.cn

> User-Agent: curl/7.54.0

> Accept: \*/\*

> Range: bytes=0-10

# 响应头

< HTTP/1.1 206 Partial Content

< Content-Type: application/octet-stream

< Content-Length: 11

< Connection: keep-alive

< Date: Thu, 21 Feb 2019 06:25:15 GMT

< Content-Range: bytes 0-10/233295878

我们在请求头中添加"Range: bytes=0-10"的作用是，告诉服务器本次请求我们只想获取文件0-10(包括10，共11字节)这块内容。如果服务器支持分块传输，则响应状态码为206，表示“部分内容”，并且同时响应头中包含“Content-Range”字段，如果不支持则不会包含。我们看看上面“Content-Range”的内容：

Content-Range: bytes 0-10/233295878

0-10表示本次返回的区块，233295878代表文件的总长度，单位都是byte, 也就是该文件大概233M多一点。

基于此，我们可以设计一个简单的多线程的文件分块下载器，实现的思路是：

1. 先检测是否支持分块传输，如果不支持，则直接下载；若支持，则将剩余内容分块下载。
2. 各个分块下载时保存到各自临时文件，等到所有分块下载完后合并临时文件。
3. 删除临时文件。

**实现**：

// 通过第一个分块请求检测服务器是否支持分块传输

Response response = await downloadChunk(url, 0, firstChunkSize, 0);

if (response.statusCode == 206) { //如果支持

//解析文件总长度，进而算出剩余长度

total = int.parse(

response.headers.value(HttpHeaders.contentRangeHeader).split("/").last);

int reserved = total -

int.parse(response.headers.value(HttpHeaders.contentLengthHeader));

//文件的总块数(包括第一块)

int chunk = (reserved / firstChunkSize).ceil() + 1;

if (chunk > 1) {

int chunkSize = firstChunkSize;

if (chunk > maxChunk + 1) {

chunk = maxChunk + 1;

chunkSize = (reserved / maxChunk).ceil();

}

var futures = <Future>[];

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

int start = firstChunkSize + i \* chunkSize;

//分块下载剩余文件

futures.add(downloadChunk(url, start, start + chunkSize, i + 1));

}

//等待所有分块全部下载完成

await Future.wait(futures);

}

//合并文件文件

await mergeTempFiles(chunk);

}

下面我们使用dio的downloadAPI 实现downloadChunk：

//start 代表当前块的起始位置，end代表结束位置

//no 代表当前是第几块

Future<Response> downloadChunk(url, start, end, no) async {

progress.add(0); //progress记录每一块已接收数据的长度

--end;

return dio.download(

url,

savePath + "temp$no", //临时文件按照块的序号命名，方便最后合并

onReceiveProgress: createCallback(no), // 创建进度回调，后面实现

options: Options(

headers: {"range": "bytes=$start-$end"}, //指定请求的内容区间

),

);

}

接下来实现mergeTempFiles:

Future mergeTempFiles(chunk) async {

File f = File(savePath + "temp0");

IOSink ioSink= f.openWrite(mode: FileMode.writeOnlyAppend);

//合并临时文件

for (int i = 1; i < chunk; ++i) {

File \_f = File(savePath + "temp$i");

await ioSink.addStream(\_f.openRead());

await \_f.delete(); //删除临时文件

}

await ioSink.close();

await f.rename(savePath); //合并后的文件重命名为真正的名称

}

**完整实现：**

/// Downloading by spiting as file in chunks

Future downloadWithChunks(

url,

savePath, {

ProgressCallback onReceiveProgress,

}) async {

const firstChunkSize = 102;

const maxChunk = 3;

int total = 0;

var dio = Dio();

var progress = <int>[];

createCallback(no) {

return (int received, \_) {

progress[no] = received;

if (onReceiveProgress != null && total != 0) {

onReceiveProgress(progress.reduce((a, b) => a + b), total);

}

};

}

Future<Response> downloadChunk(url, start, end, no) async {

progress.add(0);

--end;

return dio.download(

url,

savePath + "temp$no",

onReceiveProgress: createCallback(no),

options: Options(

headers: {"range": "bytes=$start-$end"},

),

);

}

Future mergeTempFiles(chunk) async {

File f = File(savePath + "temp0");

IOSink ioSink= f.openWrite(mode: FileMode.writeOnlyAppend);

for (int i = 1; i < chunk; ++i) {

File \_f = File(savePath + "temp$i");

await ioSink.addStream(\_f.openRead());

await \_f.delete();

}

await ioSink.close();

await f.rename(savePath);

}

Response response = await downloadChunk(url, 0, firstChunkSize, 0);

if (response.statusCode == 206) {

total = int.parse(

response.headers.value(HttpHeaders.contentRangeHeader).split("/").last);

int reserved = total -

int.parse(response.headers.value(HttpHeaders.contentLengthHeader));

int chunk = (reserved / firstChunkSize).ceil() + 1;

if (chunk > 1) {

int chunkSize = firstChunkSize;

if (chunk > maxChunk + 1) {

chunk = maxChunk + 1;

chunkSize = (reserved / maxChunk).ceil();

}

var futures = <Future>[];

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

int start = firstChunkSize + i \* chunkSize;

futures.add(downloadChunk(url, start, start + chunkSize, i + 1));

}

await Future.wait(futures);

}

await mergeTempFiles(chunk);

}

}

//分块下载

main() async {

var url = "http://download.dcloud.net.cn/HBuilder.9.0.2.macosx\_64.dmg";

var savePath = "./example/HBuilder.9.0.2.macosx\_64.dmg";

await downloadWithChunks(url, savePath, onReceiveProgress: (received, total) {

if (total != -1) {

print("${(received / total \* 100).floor()}%");

}

});

}

# WebSockets

WebSocket协议本质上是一个基于tcp的协议，它是先通过HTTP协议发起一条特殊的http请求进行握手后，如果服务端支持WebSocket协议，则会进行协议升级。WebSocket会使用http协议握手后创建的tcp链接，和http协议不同的是，WebSocket的tcp链接是个长链接（不会断开），所以服务端与客户端就可以通过此TCP连接进行实时通信。

webSocket.org提供的测试服务器：

<https://book.flutterchina.club/chapter11/websocket.html>

实现步骤：

1. 连接到WebSocket服务器。
2. 监听来自服务器的消息。
3. 将数据发送到服务器。
4. 关闭WebSocket连接。

完整例子：

import 'package:flutter/material.dart';  
import 'package:web\_socket\_channel/io.dart'; //安装下载：web\_socket\_channel

class WebSocketRoute extends StatefulWidget {  
 @override  
 \_WebSocketRouteState createState() => new \_WebSocketRouteState();  
}  
  
class \_WebSocketRouteState extends State<WebSocketRoute> {  
 TextEditingController \_controller = new TextEditingController();  
 IOWebSocketChannel channel;  
 String \_text = "";  
  
  
 @override  
 void initState() {  
 //创建websocket连接  
 channel = new IOWebSocketChannel.connect('ws://echo.websocket.org');  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return new Scaffold(  
 appBar: new AppBar(  
 title: new Text("WebSocket(内容回显)"),  
 ),  
 body: new Padding(  
 padding: const EdgeInsets.all(20.0),  
 child: new Column(  
 crossAxisAlignment: CrossAxisAlignment.start,  
 children: <Widget>[  
 new Form(  
 child: new TextFormField(  
 controller: \_controller,  
 decoration: new InputDecoration(labelText: 'Send a message'),  
 ),  
 ),  
 new StreamBuilder(  
 stream: channel.stream,  
 builder: (context, snapshot) {  
 //网络不通会走到这  
 if (snapshot.hasError) {  
 \_text = "网络不通...";  
 } else if (snapshot.hasData) {  
 \_text = "echo: "+snapshot.data;  
 }  
 return new Padding(  
 padding: const EdgeInsets.symmetric(vertical: 24.0),  
 child: new Text(\_text),  
 );  
 },  
 )  
 ],  
 ),  
 ),  
 floatingActionButton: new FloatingActionButton(  
 onPressed: \_sendMessage,  
 tooltip: 'Send message',  
 child: new Icon(Icons.*send*),  
 ),  
 );  
 }  
  
 void \_sendMessage() {  
 if (\_controller.text.isNotEmpty) {  
 channel.sink.add(\_controller.text);  
 }  
 }  
  
 @override  
 void dispose() {  
 channel.sink.close();  
 super.dispose();  
 }  
}

# Socket API

Http协议和WebSocket协议都属于应用层协议，除了它们，应用层协议还有很多如：SMTP、FTP等，这些应用层协议的实现都是通过Socket API来实现的。Flutter的Socket API在dart：io包中，下面我们看一个使用Socket实现简单http请求的示例，以请求百度首页为例：

\_request() async{

//建立连接

var socket=await Socket.connect("baidu.com", 80);

//根据http协议，发送请求头

socket.writeln("GET / HTTP/1.1");

socket.writeln("Host:baidu.com");

socket.writeln("Connection:close");

socket.writeln();

await socket.flush(); //发送

//读取返回内容

\_response =await socket.transform(utf8.decoder).join();

await socket.close();

}

可以看到，使用Socket需要我们自己实现Http协议（需要自己实现和服务器的通信过程），本例只是一个简单示例，没有处理重定向、cookie等。

# Json转Dart Model类

在实际项目中，后台接口往往会返回一些结构化数据，如JSON、XML等，为了方便我们在代码中操作JSON，我们先将JSON格式的字符串转为Dart对象，然后进行使用。

这里介绍我自己常用的两种方式，以供参考。更多方式可自行百度。

方法一：建议使用在简单小项目中

可以通过dart:convert中内置的JSON解码器json.decode() 来实现，该方法可以根据JSON字符串具体内容将其转为List或Map，这样我们就可以通过他们来查找所需的值.

//一个JSON格式的用户列表字符串

String jsonStr='[{"name":"Jack"},{"name":"Rose"}]';

//将JSON字符串转为Dart对象(此处是List)

List items=json.decode(jsonStr);

//输出第一个用户的姓名

print(items[0]["name"]);

方法二：在线生成<https://javiercbk.github.io/json_to_dart/>



# 关于ios端flutter支付宝支付

1.支付宝支付不能前端返回支付结果。需要异步通知服务器，或者前端通过多次请求，进行确认；

# 关于微信支付问题：

1.如果在前期微信支付审核中填写的平台信息BundleID或者Android的应用报名等信息有误，支付是回返回 -1 ；导致支付不成功

2.Android微信支付需要注意应用包名和应用签名的正确性，必须要正是打包后才能使用等细节

# 关于flutterSDK更新

在项目下执行命令：flutter upgrade

# 疑问：

1.．flutter中Container什么情况下宽度会失效？

2.．textField行高怎么控制?如何实现类textAlign效果？