本问参考[《Flutter实战》](https://book.flutterchina.club/)

# 起步

## 移动开发技术简介

### 原始开发与跨平台技术

#### 原始开发

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

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

主要缺点：

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

但近几年，随着物联网时代到来、移动互联网高歌猛进，日新月异，在很多业务场景中，传统的纯原生开发已经不能满足日益增长的业务需求。主要表现在：

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

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

#### 跨平台技术

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

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

在接下来的章节中我们逐个来看看这三类框架的原理及优缺点。

### Hybrid技术简介

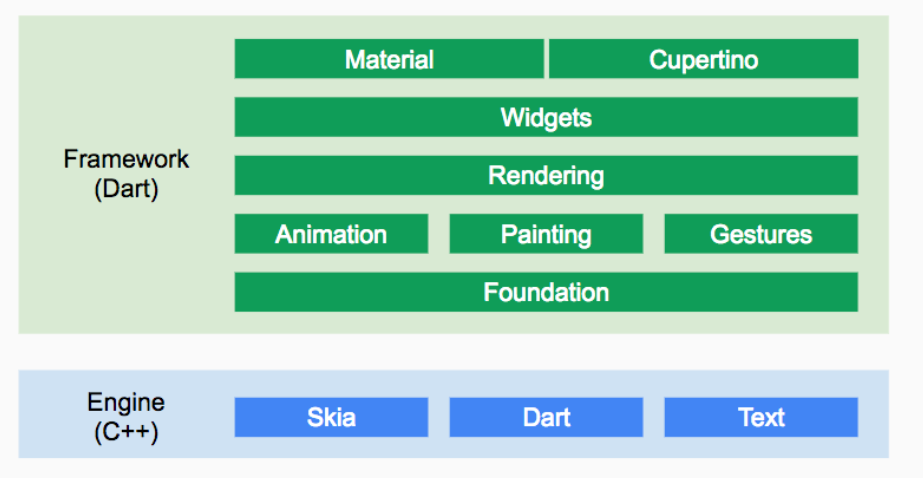
### React Native 和 Weex

### QT Moblie 与 Flutter

## Flutter简介

初识Flutter

### Flutter框架结构



## 搭建Flutter开发环境

## 常见配置问题

## Dart语言简介

# 第一个Flutter应用

## 计数器示例

## 路由管理

路由(Route)在移动开发中通常指页面（Page），这跟web开发中单页应用的Route概念意义是相同的，Route在Android中通常指一个Activity，在iOS中指一个ViewController。所谓路由管理，就是管理页面之间如何跳转，通常也可被称为导航管理。这和原生开发类似，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。

### 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组件库的一个Widget，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：

（1）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。

（2）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。

### Navigator

Navigator是一个路由管理的widget，它通过一个栈来管理一个路由widget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：

### 命名路由

所谓命名路由（Named Route）即给路由起一个名字，然后可以通过路由名字直接打开新的路由。这为路由管理带来了一种直观、简单的方式。

## 包管理

一个完整的应用程序往往会依赖很多第三方包，正如在原生开发中，Android使用Gradle来管理依赖，iOS用Cocoapods或Carthage来管理依赖，而Flutter也有自己的依赖管理工具，本节我们主要介绍一下flutter如何使用配置文件pubspec.yaml（位于项目根目录）来管理第三方依赖包。

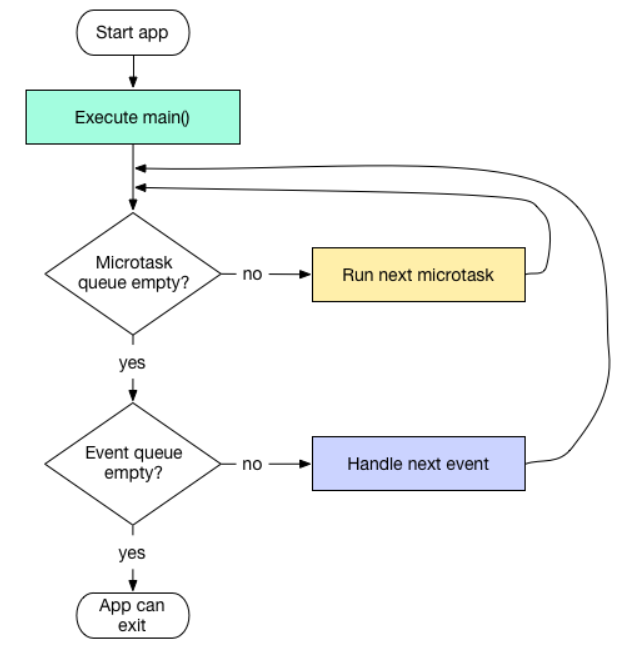
## 资源管理

## 调试Flutter APP

## Dart线程模型及异常处理

### Dart单线程模型

在Java和OC中，如果程序发生异常且没有被捕获，那么程序将会终止，但在Dart或JavaScript中则不会，究其原因，这和它们的运行机制有关系，Java和OC都是多线程模型的编程语言，任意一个线程触发异常且没被捕获时，整个进程就退出了。但Dart和JavaScript不会，它们都是单线程模型，运行机制很相似(但有区别)，下面我们通过Dart官方提供的一张图来看看dart大致运行原理：



Dart 在单线程中是以消息循环机制来运行的，其中包含两个任务队列，一个是“微任务队列” microtask queue，另一个叫做“事件队列” event queue。从图中可以发现，微任务队列的执行优先级高于事件队列。

现在我们来介绍一下Dart线程运行过程，如上图中所示，入口函数 main() 执行完后，消息循环机制便启动了。首先会按照先进先出的顺序逐个执行微任务队列中的任务，当所有微任务队列执行完后便开始执行事件队列中的任务，事件任务执行完毕后再去执行微任务，如此循环往复，生生不息。

在Dart中，所有的外部事件任务都在事件队列中，如IO、计时器、点击、以及绘制事件等，而微任务通常来源于Dart内部，并且微任务非常少，之所以如此，是因为微任务队列优先级高，如果微任务太多，执行时间总和就越久，事件队列任务的延迟也就越久，对于GUI应用来说最直观的表现就是比较卡，所以必须得保证微任务队列不会太长。值得注意的是，我们可以通过Future.microtask(…)方法向微任务队列插入一个任务。

在事件循环中，当某个任务发生异常并没有被捕获时，程序并不会退出，而直接导致的结果是当前任务的后续代码就不会被执行了，也就是说一个任务中的异常是不会影响其它任务执行的。

### Flutter异常捕获

# 基础Widgets

本节介绍一下Flutter中常用的一些基础widget.

## Widget简介

### 概念

Flutter中几乎所有的对象都是一个Widget。

在前面的介绍中，我们知道Flutter中几乎所有的对象都是一个Widget，与原生开发中“控件”不同的是，Flutter中的widget的概念更广泛，它不仅可以表示UI元素，也可以表示一些功能性的组件如：用于手势检测的 GestureDetector widget、用于应用主题数据传递的Theme等等。而原生开发中的控件通常只是指UI元素。在后面的内容中，我们在描述UI元素时，我们可能会用到“控件”、“组件”这样的概念，读者心里需要知道他们就是widget，只是在不同场景的不同表述而已。由于Flutter主要就是用于构建用户界面的，所以，在大多数时候，读者可以认为widget就是一个控件，不必纠结于概念。

### Widget与Element

在Flutter中，Widget的功能是“描述一个UI元素的配置数据”，它就是说，**Widget其实并不是表示最终绘制在设备屏幕上的显示元素，而只是显示元素的一个配置数据。** 实际上，Flutter中真正代表屏幕上显示元素的类是Element，也就是说Widget只是描述Element的一个配置。**Widget只是UI元素的一个配置数据，并且一个Widget可以对应多个Element，**这是因为同一个Widget对象可以被添加到UI树的不同部分，而真正渲染时，UI树的每一个Element节点都会对应一个Widget对象。总结一下：

（1）Widget实际上就是Element的配置数据，Widget树实际上是一个配置树，而真正的UI渲染树是由Element构成；不过，由于Element是通过Widget生成，所以它们之间有对应关系，所以在大多数场景，我们可以宽泛地认为Widget树就是指UI控件树或UI渲染树。

（2）一个Widget对象可以对应多个Element对象。这很好理解，根据同一份配置（Widget），可以创建多个实例（Element）。

### 主要接口

我们先来看一下Widget类的声明：

@immutable

abstract class Widget extends DiagnosticableTree {

const Widget({ this.key });

final Key key;

@protected

Element createElement();

@override

String toStringShort() {

return key == null ? '$runtimeType' : '$runtimeType-$key';

}

@override

void debugFillProperties(DiagnosticPropertiesBuilder properties) {

super.debugFillProperties(properties);

properties.defaultDiagnosticsTreeStyle = DiagnosticsTreeStyle.dense;

}

static bool canUpdate(Widget oldWidget, Widget newWidget) {

return oldWidget.runtimeType == newWidget.runtimeType

&& oldWidget.key == newWidget.key;

}

}

（1）Widget类继承自DiagnosticableTree，DiagnosticableTree即“诊断树”，主要作用是提供调试信息。

（2）Key: 这个key属性类似于React/Vue中的key，主要的作用是决定是否在下一次build时复用旧的widget，决定的条件在canUpdate()方法中。

（3）createElement()：正如前文所述“一个Widget可以对应多个Element”；Flutter Framework在构建UI树时，会先调用此方法生成对应节点的Element对象。此方法是Flutter Framework隐式调用的，在我们开发过程中基本不会调用到。

（4）debugFillProperties(...) 复写父类的方法，主要是设置诊断树的一些特性。

（5） canUpdate(...)是一个静态方法，它主要用于在Widget树重新build时复用旧的widget，其实具体来说，应该是：是否用新的Widget对象去更新旧UI树上所对应的Element对象的配置；通过其源码我们可以看到，只要newWidget与oldWidget的runtimeType和key同时相等时就会用newWidget去更新Element对象的配置，否则就会创建新的Element。

有关Key和Widget复用的细节将会在本书后面高级部分深入讨论，读者现在只需知道，为Widget显式添加key的话可能（但不一定）会使UI在重新构建时变的高效，读者目前可以先忽略此参数。本书后面的示例中，我们只在构建列表项UI时会显式指定Key。

另外Widget类本身是一个抽象类，其中最核心的就是定义了createElement()接口，在Flutter开发中，我们一般都不用直接继承Widget类来实现Widget，相反，我们通常会通过继承StatelessWidget和StatefulWidget来间接继承Widget类来实现，而Stateless Widget和Stateful Widget都是直接继承自Widget类，而这两个类也正是Flutter中非常重要的两个抽象类，它们引入了两种Widget模型，接下来我们将重点介绍一下这两个类。

### Stateless Widget

StatelessWidget用于不需要维护状态的场景，它通常在build方法中通过嵌套其它Widget来构建UI，在构建过程中会递归的构建其嵌套的Widget。

|  |
| --- |
| **abstract class** StatelessWidget **extends** Widget {  */// Initializes [key] for subclasses.* **const** StatelessWidget({ Key key }) : **super**(key: key);   */// Creates a [StatelessElement] to manage this widget's location in the tree.  ///  /// It is uncommon for subclasses to override this method.* @override  StatelessElement createElement() => StatelessElement(**this**);  ………. |

### Stateful Widget

和StatelessWidget一样，StatefulWidget也是继承自widget类，并重写了createElement()方法，不同的是返回的Element 对象并不相同；另外StatefulWidget类中添加了一个新的接口createState()，下面我们看看StatefulWidget的类定义：

abstract class StatefulWidget extends Widget {

const StatefulWidget({ Key key }) : super(key: key);

@override

StatefulElement createElement() => new StatefulElement(this);

@protected

State createState();

}

（1）StatefulElement 间接继承自Element类，与StatefulWidget相对应（作为其配置数据）。StatefulElement中可能会多次调用createState()来创建状态(State)对象。

（2）createState() 用于创建和Stateful widget相关的状态，它在Stateful widget的生命周期中可能会被多次调用。例如，当一个Stateful widget同时插入到widget树的多个位置时，Flutter framework就会调用该方法为每一个位置生成一个独立的State实例，其实，本质上就是一个StatefulElement对应一个State实例。

### state

一个StatefulWidget类会对应一个State类，State表示与其对应的StatefulWidget要维护的状态，State中的保存的状态信息可以：

（1）在widget build时可以被同步读取。

（2）在widget生命周期中可以被改变，当State被改变时，可以手动调用其setState()方法通知Flutter framework状态发生改变，Flutter framework在收到消息后，会重新调用其build方法重新构建widget树，从而达到更新UI的目的。

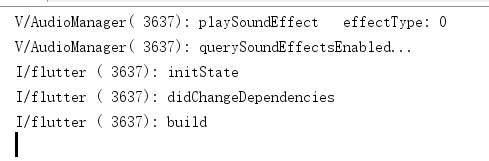
State中有两个常用属性：

（1）widget，它表示与该State实例关联的widget实例，由Flutter framework动态设置。注意，这种关联并非永久的，因为在应用声明周期中，UI树上的某一个节点的widget实例在重新构建时可能会变化，但State实例只会在第一次插入到树中时被创建，当在重新构建时，如果widget被修改了，Flutter framework会动态设置State.widget为新的widget实例。

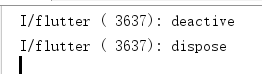
（2）context，它是BuildContext类的一个实例，表示构建widget的上下文，它是操作widget在树中位置的一个句柄，它包含了一些查找、遍历当前Widget树的一些方法。每一个widget都有一个自己的context对象。

#### State生命周期

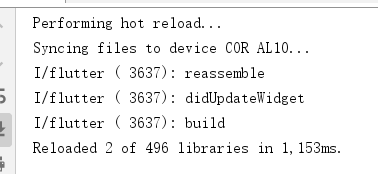
打开一个路由界面



返回到上一级：



点击热重载：



下面我们来看看 各个回调函数：

（1）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。不能在该回调中调用BuildContext.inheritFromWidgetOfExactType（该方法用于在Widget树上获取离当前widget最近的一个父级InheritFromWidget，关于InheritedWidget我们将在后面章节介绍），原因是在初始化完成后，Widget树中的InheritFromWidget也可能会发生变化，所以正确的做法应该在在build（）方法或didChangeDependencies()中调用它。

（2）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InheritedWidget，然后在之后的build() 中InheritedWidget发生了变化，那么此时InheritedWidget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。

（3）build()：此回调读者现在应该已经相当熟悉了，它主要是用于构建Widget子树的，会在如下场景被调用：

在调用initState()之后。

在调用didUpdateWidget()之后。

在调用setState()之后。

在调用didChangeDependencies()之后。

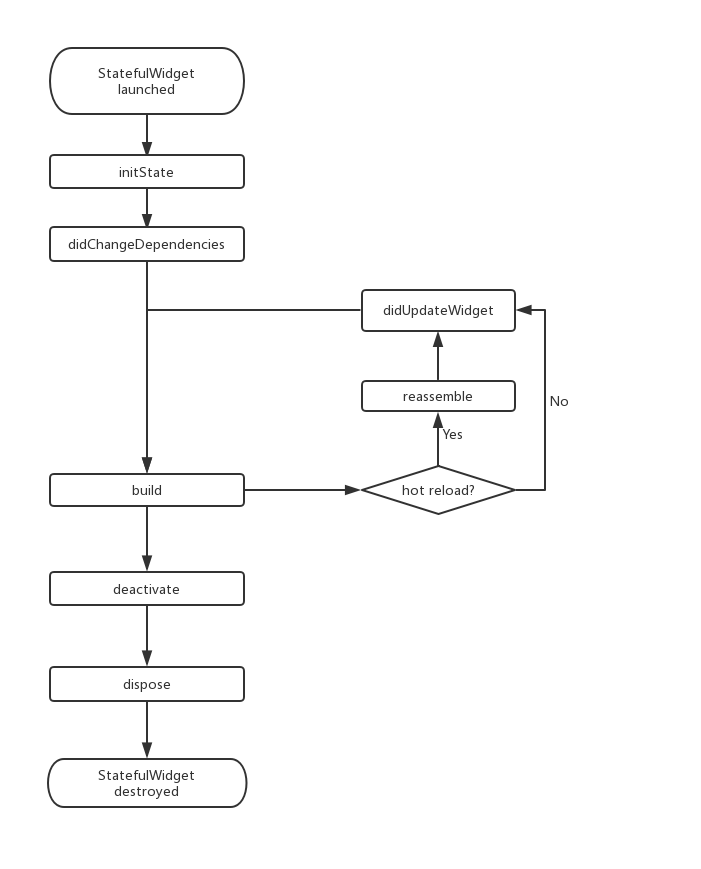
在State对象从树中一个位置移除后（会调用deactivate）又重新插入到树的其它位置之后。

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

（5）didUpdateWidget()：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。正如之前所述，Widget.canUpdate会在新旧widget的key和runtimeType同时相等时会返回true，也就是说在在新旧widget的key和runtimeType同时相等时didUpdateWidget()就会被调用。

（6）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。

（7）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。



### state管理

言归正传，我们想一个问题，stateful widget的状态应该被谁管理？widget本身？父widget？都会？还是另一个对象？答案是取决于实际情况！以下是管理状态的最常见的方法：

Widget管理自己的state。

父widget管理子widget状态。

混合管理（父widget和子widget都管理状态）。

如何决定使用哪种管理方法？以下原则可以帮助你决定：

如果状态是用户数据，如复选框的选中状态、滑块的位置，则该状态最好由父widget管理。

如果状态是有关界面外观效果的，例如颜色、动画，那么状态最好由widget本身来管理。

如果某一个状态是不同widget共享的则最好由它们共同的父widget管理。

在widget内部管理状态封装性会好一些，而在父widget中管理会比较灵活。有些时候，如果不确定到底该怎么管理状态，那么推荐的首选是在父widget中管理（灵活会显得更重要一些）。

#### widget管理自身

#### 父widget管理子widget的state

#### 混合管理

#### 全局状态管理

当应用中包括一些跨widget（甚至跨路由）的状态需要同步时，上面介绍的方法很难胜任了。比如，我们有一个设置页，里面可以设置应用语言，但是我们为了让设置实时生效，我们期望在语言状态发生改变时，我们的APP Widget能够重新build一下，但我们的APP Widget和设置页并不在一起。正确的做法是通过一个全局状态管理器来处理这种“相距较远”的widget之间的通信。目前主要有两种办法：

（1）实现一个全局的事件总线，将语言状态改变对应为一个事件，然后在APP Widget所在的父widgetinitState 方法中订阅语言改变的事件，当用户在设置页切换语言后，我们触发语言改变事件，然后APP Widget那边就会收到通知，然后重新build一下即可。

（2）使用redux这样的全局状态包，读者可以在pub上查看其详细信息。

### widget库的介绍

Flutter提供了一套丰富、强大的基础widget，在基础widget库之上Flutter又提供了一套Material风格（Android默认的视觉风格）和一套Cupertino风格（iOS视觉风格）的widget库。要使用基础widget库，需要先导入：

import 'package:flutter/widgets.dart';

下面我们介绍一下常用的widget。

#### 基础widget

（1）Text：该 widget 可让您创建一个带格式的文本。

（2）Row、 Column： 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。

（3）Stack： 取代线性布局 (译者语：和Android中的FrameLayout相似)，Stack允许子 widget 堆叠， 你可以使用 Positioned 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。

（4）Container： Container 可让您创建矩形视觉元素。container 可以装饰一个BoxDecoration, 如 background、一个边框、或者一个阴影。 Container 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可以使用矩阵在三维空间中对其进行变换。

#### Material widget

Flutter提供了一套丰富的Material widget，可帮助您构建遵循Material Design的应用程序。Material应用程序以MaterialApp widget开始， 该widget在应用程序的根部创建了一些有用的widget，比如一个Theme，它配置了应用的主题。 是否使用MaterialApp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 widget了，如：Scaffold、AppBar、FlatButton等。要使用Material widget，需要先引入它：

import 'package:flutter/material.dart';

#### Cupertino widget

### 总结

Flutter提供了丰富的widget，在实际的开发中你可以随意使用它们，不要怕引入过多widget库会让你的应用安装包变大，这不是web开发，dart在编译时只会编译你使用了的代码。由于Material和Cupertino都是在基础widget库之上的，所以如果你的应用中引入了这两者之一，则不需要再引入flutter/widgets.dart了，因为它们内部已经引入过了。

## 文本，字体样式

### Text

Text("Hello world",

textAlign: TextAlign.center,

);

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

maxLines: 1,

overflow: TextOverflow.ellipsis,

);

Text("Hello world",

textScaleFactor: 1.5,

);

### TextStyle

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

),

);

### TextSpan

const TextSpan({

TextStyle style,

Sting text,

List<TextSpan> children,

GestureRecognizer recognizer,

});

Text.rich(TextSpan(

children: [

TextSpan(

text: "Home: "

),

TextSpan(

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

style: TextStyle(

color: Colors.blue

),

recognizer: \_tapRecognizer

),

]

))

### DefaultTextStyle

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

),

),

],

),

);

### 使用字体

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属性使用字体。

（1）要先在pubspec.yaml中声明它。

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

（2）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,

);

## 按钮

Material widget库中提供了多种按钮Widget如RaisedButton、FlatButton、OutlineButton等，它们都是直接或间接对RawMaterialButton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”。

（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

### FlatButton

### OutlineButton

### IconButton

### 自定义按钮外观

## 图片和Icon

#### 图片

Flutter中，我们可以通过Image来加载并显示图片，Image的数据源可以是asset、文件、内存以及网络。

#### ImageProvider

ImageProvider 是一个抽象类，主要定义了图片数据获取的接口load()，从不同的数据源获取图片需要实现不同的ImageProvider ，如AssetImage是实现了从Asset中加载图片的ImageProvider，而NetworkImage实现了从网络加载图片的ImageProvider。

#### ICON

Flutter中，可以像web开发一样使用iconfont，iconfont即“字体图标”，它是将图标做成字体文件，然后通过指定不同的字符而显示不同的图片。

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。

## 单选框和复选框

Material widgets库中提供了Material风格的单选开关Switch和复选框Checkbox，它们都是继承自StatelessWidget，所以它们本身不会保存当前选择状态，所以一般都是在父widget中管理选中状态。当用户点击Switch或Checkbox时，它们会触发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) {  print(**"build"**);  **return** Scaffold(  appBar: AppBar(  title: Text(**"Image Widget"**),  ),  body: Center(  child: Column(  mainAxisAlignment: MainAxisAlignment.**center**,  children: <Widget>[  Text(**"Switch and Check 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;  });  },  ),  ]),  ),  );  } } |

## 输入框和表单

Material widget库中提供了丰富的输入框及表单Widget。

### TextField

TextField用于文本输入，它提供了很多属性，我们先简单介绍一下主要属性的作用。

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,

...

})

#### 示例：登录输入框

#### 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。

### 表单Form

# 布局类Widgets

## 布局类Widgets简介

布局类Widget都会包含一个或多个子widget，不同的布局类Widget对子widget排版(layout)方式不同。我们在前面说过Element树才是最终的绘制树，Element树是通过widget树来创建的（通过Widget.createElement()），widget其实就是Element的配置数据。Flutter中，根据Widget是否需要包含子节点将Widget分为了三类，分别对应三种Element，如下表：

| **Widget** | **对应的Element** | **用途** |
| --- | --- | --- |
| LeafRenderObjectWidget | LeafRenderObjectElement | Widget树的叶子节点，用于没有子节点的widget，通常基础widget都属于这一类，如Text、Image。 |
| SingleChildRenderObjectWidget | SingleChildRenderObjectElement | 包含一个子Widget，如：ConstrainedBox、DecoratedBox等 |
| MultiChildRenderObjectWidget | MultiChildRenderObjectElement | 包含多个子Widget，一般都有一个children参数，接受一个Widget数组。如Row、Column、Stack等 |

布局类Widget就是指直接或间接继承(包含)MultiChildRenderObjectWidget的Widget，它们一般都会有一个children属性用于接收子Widget。我们看一下继承关系 Widget > RenderObjectWidget > (Leaf/SingleChild/MultiChild)RenderObjectWidget 。RenderObjectWidget类中定义了创建、更新RenderObject的方法，子类必须实现他们，关于RenderObject我们现在只需要知道它是最终布局、渲染UI界面的对象即可，也就是说，对于布局类Widget来说，其布局算法都是通过对应的RenderObject对象来实现的，所以读者如果对接下来介绍的某个布局类Widget原理感兴趣，可以查看其RenderObject的实现，而在本章中，为了让读者对布局类Widget有个快速的认识，所以我们不会深入到RenderObject的细节中去。在学习本章时，读者的重点是掌握不同布局类Widget的布局特点，具体原理和细节我们会在后面高级部分介绍。

## 线性布局Row，Column

所谓线性布局，即指沿水平或垂直方向排布子Widget。Flutter中通过Row和Column来实现线性布局，类似于Android中的LinearLayout控件。Row和Column都继承自Flex，我们将在弹性布局一节中详细介绍Flex。

### 主轴和纵轴

对于线性布局，有主轴和纵轴之分，如果布局是沿水平方向，那么主轴就是指水平方向，而纵轴即垂直方向；如果布局沿垂直方向，那么主轴就是指垂直方向，而纵轴就是水平方向。在线性布局中，有两个定义对齐方式的枚举类MainAxisAlignment和CrossAxisAlignment，分别代表主轴对齐和纵轴对齐。

### 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>[],

})

### Column

Column可以在垂直方向排列其子widget。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，读者可类比Row来理解，在此不再赘述。

## 弹性布局Flex

**弹性布局允许子widget按照一定比例来分配父容器空间**，弹性布局的概念在其UI系统中也都存在，如H5中的弹性盒子布局，Android中的FlexboxLayout。Flutter中的弹性布局主要通过Flex和Expanded来配合实现。

### Flex

Flex可以沿着水平或垂直方向排列子widget，如果你知道主轴方向，使用Row或Column会方便一些，因为Row和Column都继承自Flex，参数基本相同，所以能使用Flex的地方一定可以使用Row或Column。Flex本身功能是很强大的，它也可以和Expanded配合实现弹性布局，接下来我们只讨论Flex和弹性布局相关的属性(其它属性已经在介绍Row和Column时介绍过了)。

Flex({

...

@required this.direction, //弹性布局的方向, Row默认为水平方向，Column默认为垂直方向

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

})

### Expanded

可以按比例“扩伸”Row、Column和Flex子widget所占用的空间。

const Expanded({

int flex = 1,

@required Widget child,

})

flex为弹性系数，如果为0或null，则child是没有弹性的，即不会被扩伸占用的空间。如果大于0，所有的Expanded按照其flex的比例来分割主轴的全部空闲空间。下面我们看一个例子：

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,

),

),

],

),

),

),

],

);

}

}

## 流式布局Wrap，Flow

### Wrap

在介绍Row和Colum时，如果子widget超出屏幕范围，则会报溢出错误，如：

Row(

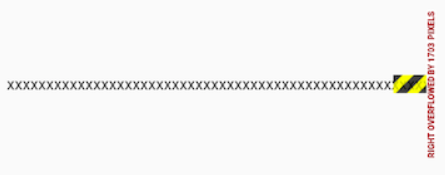
children: <Widget>[

Text("xxx"\*100)

],

);

运行：



可以看到，右边溢出部分报错。这是因为Row默认只有一行，如果超出屏幕不会折行。我们把超出屏幕显示范围会自动折行的布局称为**流式布局**。Flutter中通过Wrap和Flow来支持流式布局，将上例中的Row换成Wrap后溢出部分则会自动折行。下面是Wrap的定义:

### Flow

我们一般很少会使用Flow，因为其过于复杂，需要自己实现子widget的位置转换，在很多场景下首先要考虑的是Wrap是否满足需求。Flow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。Flow有如下优点：

（1）性能好；Flow是一个对child尺寸以及位置调整非常高效的控件，Flow用转换矩阵（transformation matrices）在对child进行位置调整的时候进行了优化：在Flow定位过后，如果child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（transformation matrices），并没有实际调整Widget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widget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widget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。

## 层叠布局Stack，Positioned

层叠布局和Web中的绝对定位、Android中的Frame布局是相似的，子widget可以根据到父容器四个角的位置来确定本身的位置。绝对定位允许子widget堆叠（按照代码中声明的顺序）。Flutter中使用Stack和Positioned来实现绝对定位，Stack允许子widget堆叠，而Positioned可以给子widget定位（根据Stack的四个角）。

### Stack

Stack({

this.alignment = AlignmentDirectional.topStart,

this.textDirection,

this.fit = StackFit.loose,

this.overflow = Overflow.clip,

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

})

### positioned

const Positioned({

Key key,

this.left,

this.top,

this.right,

this.bottom,

this.width,

this.height,

@required Widget child,

})

# 容器类Widgets

容器类Widget和布局类Widget都作用于其子Widget，不同的是：

（1）布局类Widget一般都需要接收一个widget数组（children），他们直接或间接继承自（或包含）MultiChildRenderObjectWidget ；而容器类Widget一般只需要接收一个子Widget（child），他们直接或间接继承自（或包含）SingleChildRenderObjectWidget。

（2）布局类Widget是按照一定的排列方式来对其子Widget进行排列；而容器类Widget一般只是包装其子Widget，对其添加一些修饰（补白或背景色等）、变换(旋转或剪裁等)、或限制(大小等)。

注意，Flutter官方并没有对Widget进行官方分类，我们对其分类主要是为了方便讨论和对Widget功能区分的记忆。

## Padding

Padding可以给其子节点添加补白（填充），我们在前面很多示例中都已经使用过它了，现在来看看它的定义：

Padding({

...

EdgeInsetsGeometry padding,

Widget child,

})

EdgeInsetsGeometry是一个抽象类，开发中，我们一般都使用EdgeInsets，它是EdgeInsetsGeometry的一个子类，定义了一些设置补白的便捷方法。

### EdgeInsets

我们看看EdgeInsets提供的便捷方法：

fromLTRB(double left, double top, double right, double bottom)：分别指定四个方向的补白。

all(double value) : 所有方向均使用相同数值的补白。

only({left, top, right ,bottom })：可以设置具体某个方向的补白(可以同时指定多个方向)。

symmetric({ vertical, horizontal })：用于设置对称方向的补白，vertical指top和bottom，horizontal指left和right。

## 布局限制类容器ConstrainedBox和SizedBox

ConstrainedBox和SizedBox都是通过RenderConstrainedBox来渲染的。SizedBox只是ConstrainedBox的一个定制，本节把他们放在一起讨论。

### ConstrainedBox

ConstrainedBox用于对子widget添加额外的约束。例如，如果你想让子widget的最小高度是80像素，你可以使用const BoxConstraints(minHeight: 80.0)作为子widget的约束。

#### 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widget指定固定的宽高，

### 多重限制

如果某一个widget有多个父ConstrainedBox限制，那么最终会是哪个生效？ 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。

### UnconstrainedBox

UnconstrainedBox不会对子Widget产生任何限制，它允许其子Widget按照其本身大小绘制。一般情况下，我们会很少直接使用此widget，但在"去除"多重限制的时候也许会有帮助，我们看下下面的代码：

ConstrainedBox(

constraints: BoxConstraints(minWidth: 60.0, minHeight: 100.0), //父

child: UnconstrainedBox( //“去除”父级限制

child: ConstrainedBox(

constraints: BoxConstraints(minWidth: 90.0, minHeight: 20.0),//子

child: redBox,

),

)

)

上面代码中，如果没有中间的UnconstrainedBox，那么根据上面所述的多重限制规则，那么最终将显示一个90×100的红色框。但是由于 UnconstrainedBox “去除”了父ConstrainedBox的限制，则最终会按照子ConstrainedBox的限制来绘制redBox，即90×20。

## 装饰容器DecoratedBox

DecoratedBox可以在其子widget绘制前(或后)绘制一个装饰Decoration（如背景、边框、渐变等）。DecoratedBox定义如下：

const DecoratedBox({

Decoration decoration,

DecorationPosition position = DecorationPosition.background,

Widget child

})

### BoxDecoration

我们通常会直接使用BoxDecoration，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

})

## 变换Transform

Transform可以在其子Widget绘制时对其应用一个矩阵变换（transformation），Matrix4是一个4D矩阵，通过它我们可以实现各种矩阵操作。

注意：

（1）Transform的变换是应用在绘制阶段，而并不是应用在布局(layout)阶段，所以无论对子widget应用何种变化，其占用空间的大小和在屏幕上的位置都是固定不变的，因为这些是在布局阶段就确定的。

（2）由于矩阵变化只会作用在绘制阶段，所以在某些场景下，在UI需要变化时，可以直接通过矩阵变化来达到视觉上的UI改变，而不需要去重新触发build流程，这样会节省layout的开销，所以性能会比较好。如之前介绍的Flow widget，它内部就是用矩阵变换来更新UI，除此之外，Flutter的动画widget中也大量使用了Transform以提高性能。

## Container容器

Container是我们要介绍的最后一个容器类widget，它本身不对应具体的RenderObject，它是DecoratedBox、ConstrainedBox、Transform、Padding、Align等widget的一个组合widget。所以我们只需通过一个Container可以实现同时需要装饰、变换、限制的场景。下面是Container的定义：

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

Material库提供了很多Widget，本节介绍一些常用的Widget，其余的读者可以查看文档或Flutter Gallery中Material组件部分的示例。注意，笔者强烈建议用户将Flutter Gallery示例跑起来，它是一个很全面的Flutter示例，是非常好的参考Demo。

### Scaffold

大多数路由页都会包含一个导航栏，有些路由页可能会有抽屉菜单(Drawer)以及底部Tab导航菜单等。如果每个页面都需要开发者自己手动去实现，这会是一件非常无聊的事。幸运的是，我们前面提到过，Flutter Material库提供了一个Scaffold Widget，它是一个路由页的骨架，可以非常容易的拼装出一个完整的页面。

### AppBar

AppBar是一个Material风格的导航栏，它可以设置标题、导航栏菜单、底部Tab等。下面我们看看AppBar的定义：

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

})

# 可滚动Widgets

## 可滚动Widgets简介

当内容超过显示视口(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widget（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widget（如ListView、GridView等），然后介绍一下Scrollable与可滚动widget的原理。可滚动Widget都直接或间接包含一个Scrollable widget，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

})

### Scrollbar

Scrollbar是一个Material风格的滚动指示器（滚动条），如果要给可滚动widget添加滚动条，只需将Scrollbar作为可滚动widget的父widget即可，如：

Scrollbar(

child: SingleChildScrollView(

...

),

);

### 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。

### 主轴和纵轴

在可滚动widget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widget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类似于Android中的ScrollView，它只能接收一个子Widget。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,

})

## ListView

ListView是最常用的可滚动widget，它可以沿一个方向线性排布所有子widget。我们看看ListView的默认构造函数定义：

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>[],

})

### 构造函数

#### 默认构造函数

默认构造函数有一个children参数，它接受一个Widget列表（List）。这种方式适合只有少量的子widget的情况，因为这种方式需要将所有children都提前创建好（这需要做大量工作），而不是等到子widget真正显示的时候再创建。

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Widget真正显示的时候才会被创建。下面看一下ListView.builder的核心参数列表：

ListView.builder({

// ListView公共参数已省略

...

@required IndexedWidgetBuilder itemBuilder,

int itemCount,

...

})

#### ListView.separated

ListView.separated可以生成列表项之间的分割器，它除了比ListView.builder多了一个separatorBuilder参数，该参数是一个分割器生成器。下面我们看一个例子：奇数行添加一条蓝色下划线，偶数行添加一条绿色下划线。

class ListView3 extends StatelessWidget {

@override

Widget build(BuildContext context) {

//下划线widget预定义以供复用。

Widget divider1=Divider(color: Colors.blue,);

Widget divider2=Divider(color: Colors.green);

return 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%2==0?divider1:divider2;

},

);

}

}

## GridView

GridView可以构建一个二维网格列表，其默认构造函数定义如下：

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>[],

})

我们可以看到，GridView和ListView的大多数参数都是相同的，它们的含义也都相同，如有疑惑读者可以翻阅ListView一节，在此不再赘述。我们唯一需要关注的是gridDelegate参数，类型是SliverGridDelegate，它的作用是控制GridView子widget如何排列(layout)，SliverGridDelegate是一个抽象类，定义了GridView Layout相关接口，子类需要通过实现它们来实现具体的布局算法，Flutter中提供了两个SliverGridDelegate的子类SliverGridDelegateWithFixedCrossAxisCount和SliverGridDelegateWithMaxCrossAxisExtent，下面我们分别介绍：

### SliverGridDelegateWithFixedCrossAxisCount

该子类实现了一个横轴为固定数量子元素的layout算法，其构造函数为：

SliverGridDelegateWithFixedCrossAxisCount({

@required double crossAxisCount,

double mainAxisSpacing = 0.0,

double crossAxisSpacing = 0.0,

double childAspectRatio = 1.0,

})

### 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。

### 构造函数

#### GridView.builder

上面我们介绍的GridView都需要一个Widget数组作为其子元素，这些方式都会提前将所有子widget都构建好，所以只适用于子Widget数量比较少时，当子widget比较多时，我们可以通过GridView.builder来动态创建子Widget。GridView.builder 必须指定的参数有两个：

GridView.builder(

...

@required SliverGridDelegate gridDelegate,

@required IndexedWidgetBuilder itemBuilder,

)

其中itemBuilder为子widget构建器。

## CustomScrollView

CustomScrollView是可以使用sliver来自定义滚动模型（效果）的widget。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widget（Sliver）"粘"起来，而CustomScrollView的功能就相当于“胶水”。

## 滚动监听及控制ScrokkController

在前几节中，我们介绍了Flutter中常用的可滚动Widget，也说过可以用ScrollController来控制可滚动widget的滚动位置，本节先介绍一下ScrollController，然后以ListView为例，展示一下ScrollController的具体用法。最后，再介绍一下路由切换时如何来保存滚动位置。

### ScrollController

构造函数：

ScrollController({

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

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

...

})

### 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))

### 滚动位置恢复

PageStorage是一个用于保存页面(路由)相关数据的Widget，它并不会影响子树的UI外观，其实，PageStorage是一个功能型Widget，它拥有一个存储桶（bucket），子树中的Widget可以通过指定不同的PageStorageKey来存储各自的数据或状态。

### ScrollController控制原理

我们来介绍一下ScrollController的另外三个方法：

ScrollPosition createScrollPosition(

ScrollPhysics physics,

ScrollContext context,

ScrollPosition oldPosition);

void attach(ScrollPosition position) ;

void detach(ScrollPosition position) ;

当ScrollController和Scrollable Widget关联时，Scrollable Widget首先会调用ScrollController的createScrollPosition()方法来创建一个ScrollPosition来存储滚动位置信息，接着，Scrollable Widget会调用attach()方法，将创建的ScrollPosition添加到ScrollController的positions属性中，这一步称为“注册位置”，只有注册后animateTo() 和 jumpTo()才可以被调用。当Scrollable Widget销毁时，会调用ScrollController的detach()方法，将其ScrollPosition对象从ScrollController的positions属性中移除，这一步称为“注销位置”，注销后animateTo() 和 jumpTo() 将不能再被调用。

### 滚动监听

Flutter Widget树中子Widget可以通过发送通知（Notification）与父(包括祖先)Widget通信。父Widget可以通过NotificationListener Widget来监听自己关注的通知，这种通信方式类似于Web开发中浏览器的事件冒泡，我们在Flutter中沿用“冒泡”这个术语。Scrollable Widget在滚动时会发送ScrollNotification类型的通知，ScrollBar正是通过监听滚动通知来实现的。通过NotificationListener监听滚动事件和通过ScrollController有两个主要的不同：

（1）通过NotificationListener可以在从Scrollable Widget到Widget树根之间任意位置都能监听。而ScrollController只能和具体的Scrollable Widget关联后才可以。

（2）收到滚动事件后获得的信息不同；NotificationListener在收到滚动事件时，通知中会携带当前滚动位置和ViewPort的一些信息，而ScrollController只能获取当前滚动位置。

# 功能型Widgets

功能型Widget指的是不会影响UI布局及外观的Widget，它们通常具有一定的功能，如事件监听、数据存储等，我们之前介绍过的FocusScope（焦点控制）、PageStorage（数据存储）、NotificationListener（事件监听）都属于功能型Widget。由于Widget是Flutter的一等公民，功能型Widget非常多，我们不会去一一介绍，本章中主要介绍几种常用的功能型Widget。

## 导航返回拦截-WillOpoScope

为了避免用户误触返回按钮而导致APP退出，在很多APP中都拦截了用户点击返回键的按钮，当用户在某一个时间段内点击两次时，才会认为用户是要退出（而非误触）。Flutter中可以通过WillPopScope来实现返回按钮拦截，我们看看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调用（包括导航返回按钮及Android物理返回按钮），该回调需要返回一个Future对象，如果返回的Future最终值为false时，则当前路由不出栈(不会返回)，最终值为true时，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。

## 数据共享-InheritedWidget

InheritedWidget是Flutter中非常重要的一个功能型Widget，它可以高效的将数据在Widget树中向下传递、共享，这在一些需要在Widget树中共享数据的场景中非常方便，如Flutter中，正是通过InheritedWidget来共享应用主题(Theme)和Locale(当前语言环境)信息的。

### didChangeDependencies

在介绍StatefulWidget时，我们提到State对象有一个回调didChangeDependencies，它会在“依赖”发生变化时被Flutter Framework调用。而这个“依赖”指的就是是否使用了父widget中InheritedWidget的数据，如果使用了，则代表有依赖，如果没有使用则代表没有依赖。这种机制可以使子组件在所依赖的主题、locale等发生变化时有机会来做一些事情。

## 主题-Theme

Theme Widget可以为Material APP定义主题数据（ThemeData），Material组件库里很多Widget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Widget共享样式数据。

### ThemeData

ThemeData是Material Design Widget库的主题数据，Material库的Widget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，所以我们可以通过ThemeData来自定义应用主题。我们可以通过Theme.of方法来获取当前的ThemeData。

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

...

})

# 事件处理与通知

Flutter中的手势系统有两个独立的层。第一层为原始指针(pointer)事件，它描述了屏幕上指针（例如，触摸、鼠标和触控笔）的位置和移动。 第二层为手势，描述由一个或多个指针移动组成的语义动作，如拖动、缩放、双击等。本章将先分别介绍如何处理这两种事件，最后再介绍一下Flutter中重要的Notification机制。

## 原始指针事件处理

本节先来介绍一下原始指针事件(Pointer Event，在移动设备上通常为触摸事件)，下一节再介绍手势处理。

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，即：一次完整的事件分为三个阶段：手指按下、手指移动、和手指抬起，而更高级别的手势（如点击、双击、拖动等）都是基于这些原始事件的。

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widget， 指针按下事件（以及该指针的后续事件）然后被分发到由命中测试发现的最内部的widget，然后从那里开始，事件会在widget树中向上冒泡，这些事件会从最内部的widget被分发到widget根的路径上的所有Widget，

Flutter中可以使用Listener widget来监听原始触摸事件，它也是一个功能性widget。

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

})

## 手势识别

GestureDetector是一个用于手势识别的功能性Widget，我们通过它可以来识别各种手势，它是指针事件的语义化封装，接下来我们详细介绍一下各种手势识别：

### 点击、双击、长按

我们通过GestureDetector对Container进行手势识别

### 拖动、滑动

一次完整的手势过程是指用户手指按下到抬起的整个过程，期间，用户按下手指后可能会移动，也可能不会移动。GestureDetector对于拖动和滑动事件是没有区分的，他们本质上是一样的。GestureDetector会将要监听的widget的原点（左上角）作为本次手势的原点，当用户在监听的widget上按下手指时，手势识别就会开始。

### 单一方向拖动

在本示例中，是可以朝任意方向拖动的，但是在很多场景，我们只需要沿一个方向来拖动，如一个垂直方向的列表，GestureDetector可以只识别特定方向的手势事件，我们将上面的例子改为只能沿垂直方向拖动：

### 放缩

GestureDetector可以监听缩放事件，下面示例演示了一个简单的图片缩放效果：

### 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。

### 手势竞争与冲突

#### 手势竞争

如果在上例中我们同时监听水平和垂直方向的拖动事件，那么我们斜着拖动时哪个方向会生效？实际上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。实际上Flutter中的手势识别引入了一个Arena的概念，Arena直译为“竞技场”的意思，每一个手势识别器（GestureRecognizer）都是一个“竞争者”（GestureArenaMember），当发生滑动事件时，他们都要在“竞技场”去竞争本次事件的处理权，而最终只有一个“竞争者”会胜出(win)。

#### 手势冲突

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。

## 全局事件总线

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，

## 通知Notification

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（context）向上传递，所有父节点都可以通过NotificationListener来监听通知，Flutter中称这种通知由子向父的传递为“通知冒泡”（Notification Bubbling），这个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。

# 动画

精心设计的动画会让用户界面感觉更直观、流畅，能改善用户体验。 Flutter可以轻松实现各种动画类型，对于许多widget，特别是Material Design widgets，都带有在其设计规范中定义的标准动画效果(但也可以自定义这些效果)。本章将详细介绍Flutter的动画系统，并会通过几个小实例来演示，以帮助开发者迅速理解并掌握动画的开发流程与原理。

## Flutter动画简介

在任何系统的UI框架中，动画实现的原理都是相同的，即：在一段时间内，快速地多次改变UI外观，由于人眼会产生视觉暂留，最终看到的就是一个“连续”的动画，这和电影的原理是一样的，而UI的一次改变称为一个动画帧，对应一次屏幕刷新，而决定动画流畅度的一个重要指标就是帧率FPS（Frame Per Second），指每秒的动画帧数。很明显，帧率越高则动画就会越流畅。一般情况下，对于人眼来说，动画帧率超过16FPS，就比较流畅了，超过32FPS就会非常的细腻平滑，而超过32FPS基本就感受不到差别了。由于动画的每一帧都是要改变UI输出，所以在一个时间段内连续的改变UI输出是比较耗资源的，对设备的软硬件系统要求都较高，所以在UI系统中，动画的平均帧率是重要的性能指标，而在Flutter中，理想情况下是可以实现60FPS的，这和原生应用动画基本是持平的。

### Flutter中动画抽象

为了方便开发者创建动画，不同的UI系统对动画都进行了一些抽象，比如在Android中可以通过XML来描述一个动画然后设置给View。Flutter中也对动画进行了抽象，主要涉及Tween、Animation、Curve、Controller这些角色。

#### Animation

Animation对象本身和UI渲染没有任何关系。Animation是一个抽象类，它用于保存动画的插值和状态；其中一个比较常用的Animation类是Animation。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对象的输出值可以是线性的、曲线的、一个步进函数或者任何其他曲线函数。 根据Animation对象的控制方式，动画可以反向运行，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\ 或 Animation\。可以通过Animation对象的value属性获取动画的当前值。

#### 动画通知

我们可以通过Animation来监听动画的帧和状态变化：

（1）addListener()可以给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()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StatusListener。

在后面的章节中我们将会举例说明。

#### Curve

动画过程可以是匀速的、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，Curve可以是线性的(Curves.linear)，也可以是非线性的。

CurvedAnimation 将动画过程定义为一个非线性曲线.

final CurvedAnimation curve =

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

注: Curves 类定义了许多常用的曲线，也可以创建自己的，例如：

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，但不会启动它运行：

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，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()方法可以启动动画。数字的产生与屏幕刷新有关，因此每秒钟通常会产生60个数字(即60fps)，在动画的每一帧，生成新的数字后，每个Animation对象会调用其Listener对象回调，等动画状态发生改变时（如动画结束）会调用StatusListeners监听器。

#### 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。

通过将SingleTickerProviderStateMixin添加到State的定义中，然后将State对象作为vsync的值，这在后面的例子中可以见到。

#### Tween

默认情况下，AnimationController对象值的范围是0.0到1.0。如果我们需要不同的范围或不同的数据类型，则可以使用Tween来配置动画以生成不同的范围或数据类型的值。例如，像下面示例，Tween生成从-200.0到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到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，而不是继承自Animation。Animatable与Animation相似，不是必须输出double值。例如，ColorTween指定两种颜色之间的过渡。

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

### 使用AnimatedWidget简化

细心的读者可能已经发现上面示例中通过addListener()和setState() 来更新UI这一步其实是通用的，如果每个动画中都加这么一句是比较繁琐的。AnimatedWidget类封装了调用setState()的细节，并允许我们将Widget分离出来，重构后的代码如下：

### 用AnimatedBuilder重构

用AnimatedWidget可以从动画中分离出widget，而动画的渲染过程（即设置宽高）仍然在AnimatedWidget中，假设如果我们再添加一个widget透明度变化的动画，那么我们需要再实现一个AnimatedWidget，这样不是很优雅，如果我们能把渲染过程也抽象出来，那就会好很多，而AnimatedBuilder正是将渲染逻辑分离出来, 上面的build方法中的代码可以改为：

@override

Widget build(BuildContext context) {

//return AnimatedImage(animation: animation,);

return AnimatedBuilder(

animation: animation,

child: Image.asset("images/avatar.png"),

builder: (BuildContext ctx, Widget child) {

return new Center(

child: Container(

height: animation.value,

width: animation.value,

child: child,

),

);

},

);

}

### 动画状态监听

上面说过，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，下面我们逐个说明：

| **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 |

## 自定义路由过渡动画

Material库中提供了一个MaterialPageRoute，它可以使用和平台风格一致的路由切换动画，

## 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逐渐过渡，这样就会产生一个Hero动画。

你可能多次看到过 hero 动画。例如，一个路由中显示待售商品的缩略图列表，选择一个条目会将其跳转到一个新路由，新路由中包含该商品的详细信息和“购买”按钮。 在Flutter中将图片从一个路由“飞”到另一个路由称为**hero动画**，尽管相同的动作有时也称为 共享元素转换。

## 交错动画

有些时候我们可能会需要一些复杂的动画，这些动画可能由一个动画序列或重叠的动画组成，比如：有一个柱状图，需要在高度增长的同时改变颜色，等到增长到最大高度后，我们需要在X轴上平移一段距离。这时我们就需要使用交错动画（Stagger Animation）。交错动画需要注意以下几点：

（1）要创建交错动画，需要使用多个动画对象

（2）一个AnimationController控制所有动画

（3）给每一个动画对象指定间隔（Interval）

所有动画都由同一个AnimationController驱动，无论动画实时持续多长时间，控制器的值必须介于0.0和1.0之间，而每个动画的间隔（Interval）介于0.0和1.0之间。对于在间隔中设置动画的每个属性，请创建一个Tween。 Tween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起始点和终止点来决定它们的开始时间和终止时间。

# 自定义Widget

## 自定义Widget方法简介

当Flutter提供的现有Widget无法满足我们的需求，或者我们为了共享代码需要封装一些通用Widget，这时我们就需要自定义Widget。在Flutter中自定义Widget有三种方式：通过组合其它Widget、自绘和实现RenderObject，本节我们先分别介绍一下这三种方式的特点，后面章节中则详细介绍它们的细节。

### 组合其它Widget

这种方式是通过拼装其它低级别的Widget来组合成一个高级别的Widget，例如我们之前介绍的Container就是一个组合Widget，它是由DecoratedBox、ConstrainedBox、Transform、Padding、Align等组成。

在Flutter中，组合的思想非常重要，Flutter提供了非常多的基础Widget，而我们的界面开发都是按照需要组合这些Widget来实现各种不同的布局。

### 自绘

如果遇到无法通过系统提供的现有Widget实现的UI时，如我们需要一个渐变圆形进度条，而Flutter提供的CircularProgressIndicator并不支持在显示精确进度时对进度条应用渐变色（其valueColor 属性只支持执行旋转动画时变化Indicator的颜色），这时最好的方法就是通过自定义Widget绘制逻辑来画出我们期望的外观。Flutter中提供了CustomPaint和Canvas供我们自绘UI外观。

### 实现RenderObject

Flutter提供的任何具有UI外观的Widget，如文本Text、Image都是通过相应的RenderObject渲染出来的，如Text是由RenderParagraph渲染，而Image是由RenderImage渲染。RenderObject是一个抽象类，它定义了一个抽象方法paint(...):

void paint(PaintingContext context, Offset offset)

PaintingContext代表Widget的绘制上下文，通过PaintingContext.canvas可以获得Canvas，绘制逻辑主要是通过Canvas API来实现。子类需要实现此方法以实现自身的绘制逻辑，如RenderParagraph需要实现文本绘制逻辑，而RenderImage需要实现图片绘制逻辑。

可以发现，RenderObject中最终也是通过Canvas来绘制的，那么通过实现RenderObject的方式和上面介绍的通过CustomPaint和Canvas自绘的方式有什么区别？其实答案很简单，CustomPaint只是为了方便开发者封装的一个代理类，它直接继承自SingleChildRenderObjectWidget，通过RenderCustomPaint的paint方法将Canvas和画笔Painter(需要开发者实现，后面章节介绍)连接起来实现了最终的绘制（绘制逻辑在Painter中）。

### 总结

组合是自定义组件最简单的方法，在任何需要自定义的场景下，都应该优先考虑是否能够通过组合来实现。而自绘和通过实现RenderObject的方法本质上是一样的，都需要开发者调用Canvas API手动去绘制UI，缺点是必须了解Canvas API，并且得自己去实现绘制逻辑，而优点是强大灵活，理论上可以实现任何外观的UI。

在本章接下来的小节中，我们将通过一些实例来详细介绍自定义UI的过程，由于后两种方法本质是相同的，后续我们只介绍CustomPaint和Canvas的方式，读者如果对自定义RenderObject的方法好奇，可以查看RenderParagraph或RenderImage源码。

## 通过组合现有Widget实现

在Flutter中页面UI通常都是由一些低阶别的Widget组合而成，当我们需要封装一些通用Widget时，应该首先考虑是否可以通过组合其它Widget来实现，如果可以则应优先使用组合，因为直接通过现有Widget拼装会非常方便、简单、高效。

## 实例：TurnBox

我们之前已经介绍过RotatedBox，但是它有两个缺点：一是只能将其子节点以90度的倍数旋转，二是当旋转的角度发生变化时，旋转角度更新过程没有动画。

本节我们将实现一个TurnBox，它可以以任意角度来旋转其子节点，并且在角度发生变化时可以执行一个动画过渡到新状态，同时，我们可以手动指定动画速度。

## CustomPaint与Canvas

对于一些复杂或不规则的UI，我们可能无法使用现有Widget组合的方式来实现，比如我们需要一个正六边形、一个渐变的圆形进度条、一个棋盘等，当然有时候我们可以使用图片来实现，但在一些需要动态交互的场景静态图片是实现不了的，比如要实现一个手写输入面板。这时，我们就需要来自己绘制UI外观。

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 Widget，它可以结合一个画笔CustomPainter来实现绘制自定义图形。

### CustomPaint

我们看看CustomPaint构造函数：

const CustomPaint({

Key key,

this.painter,

this.foregroundPainter,

this.size = Size.zero,

this.isComplex = false,

this.willChange = false,

Widget child, //子节点，可以为空

})

### 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:

（1）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 |

（2）Size：当前绘制区域大小。

### 画笔Paint

现在画布有了，我们最后还缺一个画笔，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

### 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：

（1）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。

（2）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Widget，并设置其shouldRepaint回调值为false，然后将棋盘Widget作为背景。然后将棋子的绘制放到另一个Widget中，这样落子时只需要绘制棋子。

## 实例：圆形渐变进度条（自绘）

本节我们实现一个圆形渐变进度条，它支持：

（1）支持多种渐变色。

（2）任意弧度；整个进度可以不是整圆。

（3）可以自定义粗细、两端是否圆角等样式。

可以发现要实现这样的一个进度条是无法通过现有组件组合而成的，所以我们通过自绘方式实现。

# 文件操作与网络请求

## 文件操作

Dart的IO库包含了文件读写的相关类，它属于Dart语法标准的一部分，所以通过Dart IO库，无论是Dart VM下的脚本还是Flutter，都是通过Dart IO库来操作文件的，不过和Dart VM相比，Flutter有一个重要差异是文件系统路径不同，这是因为Dart VM是运行在PC或服务器操作系统下，而Flutter是运行在移动操作系统中，他们的文件系统会有一些差异。

### APP目录

Android和iOS的应用存储目录不同，PathProvider 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：

（1）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() 返回的值。在Android上，这是getCacheDir())返回的值。

（2）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。

（3）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。

一旦你的Flutter应用程序有一个文件位置的引用，你可以使用dart:ioAPI来执行对文件系统的读/写操作。有关使用Dart处理文件和目录的详细内容可以参考Dart语言文档，下面我们看一个简单的例子。

## HTTP请求-HttpClient

### 通过HttpClient发起HTTP请求

Dart IO库中提供了Http请求的一些类，我们可以直接使用HttpClient来发起请求。使用HttpClient发起请求分为五步：

（1）创建一个HttpClient

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）等待连接服务器

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

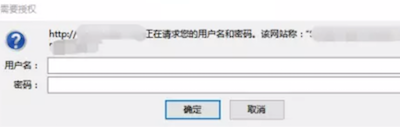
HttpClient有很多属性可以配置，常用的属性列表如下：

| **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。 |

可以发现，有些属性只是为了更方便的设置请求头，对于这些属性，你完全可以通过HttpClientRequest直接设置header，不同的是通过HttpClient设置的对整个httpClient都生效，而通过HttpClientRequest设置的只对当前请求生效。

### 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";

};

findProxy 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。

## Http请求-Dio package

### Dio http库

通过上一节介绍，我们可以发现直接使用HttpClient发起网络请求是比较麻烦的，很多事情得我们手动处理，如果再涉及到文件上传/下载、Cookie管理等就会非常繁琐。幸运的是，Dart社区有一些第三方http请求库，用它们来发起http请求将会简单的多，本节我们介绍一下目前人气较高的dio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。

## 实例：Http分块下载

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的实现，我们可以指定请求头的"range"字段来验证服务器是否支持分块传输

## WebSocket

Http协议是无状态的，只能由客户端主动发起，服务端再被动响应，服务端无法向客户端主动推送内容，并且一旦服务器响应结束，链接就会断开(见注解部分)，所以无法进行实时通信。WebSocket协议正是为解决客户端与服务端实时通信而产生的技术，现在已经被主流浏览器支持，所以对于Web开发者来说应该比较熟悉了，Flutter也提供了专门的包来支持WebSocket协议。

注意：Http协议中虽然可以通过keep-alive机制使服务器在响应结束后链接会保持一段时间，但最终还是会断开，keep-alive机制主要是用于避免在同一台服务器请求多个资源时频繁创建链接，它本质上是支持链接复用的技术，而并非用于实时通信，读者需要知道这两者的区别。

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。有关WebSocket协议细节，读者可以看RFC文档，下面我们重点看看Flutter中如何使用WebSocket。

## 使用Socket API

## Json转Model

# 包与插件

## 开发package

## 插件开发：平台通道简介

## 插件开发：实现Android端API

## 插件开发：实现IOS端API

## 系统能力调用

# 国际化

## 让App支持多语言

## 实现Localizations

## 使用Intl包

# Flutter核心原理

## Flutter UI系统

## Element和BuildContext

## RenderObject与RenderBox

## Flutter从启动到显示