Flutter大合集，进阶Flutter高级工程师

视频：

1.透过Skia底层渲染应用，解决项目中的各种疑难杂症

2.闲鱼Flutter页面解析（一）如何区分Flutter页面和原生页面，闲鱼反编译源码分析

3.闲鱼Flutter页面解析（二）Flutter3.0 市场如何，如果是你，你如何设计Flutter架构

4.闲鱼Flutter页面解析（三）Android原生界面渲染流程，跟Flutter中的skia引擎关系

5.闲鱼Flutter页面解析（四）Flutter3.0 Framework中的UI绘制原理，UI事件处理

6.闲鱼Flutter页面解析（五）一步一步教你手写Flutter3.0

1.Flutter之Flutter环境搭建

Flutter环境搭建

1. 去Flutter官网进行下载最新可用的安装包，官网地址
2. 将下载的内容进行解压到想要安装的目录
3. 将刚刚解压完的路径添加到环境变量的path中

```
export PATH=$PATH:/storepath/flutter/bin
```

运行flutter doctor，第一次会下载相关配置，可能会较慢
通过上面的几步我们就可以将Flutter安装完成，现在我们需要进行开发工具的选择，Flutter现在流行的是使用VSCode或者是使用Intellij(Xcode)两种工具进行开发，鉴于我们一般在真实的项目中很少有机会重新开一个大的项目来完全使用Flutter进行开发，所以建议还是一开始使用Intellij(Xcode)。

确定好了编译器之后我们需要给对应的编译器安装插件，以Android Studio为例，我们需要安装两个插件

- Flutter插件 为了支持Flutter的工作流程
- Dart插件 提供Dart代码分析

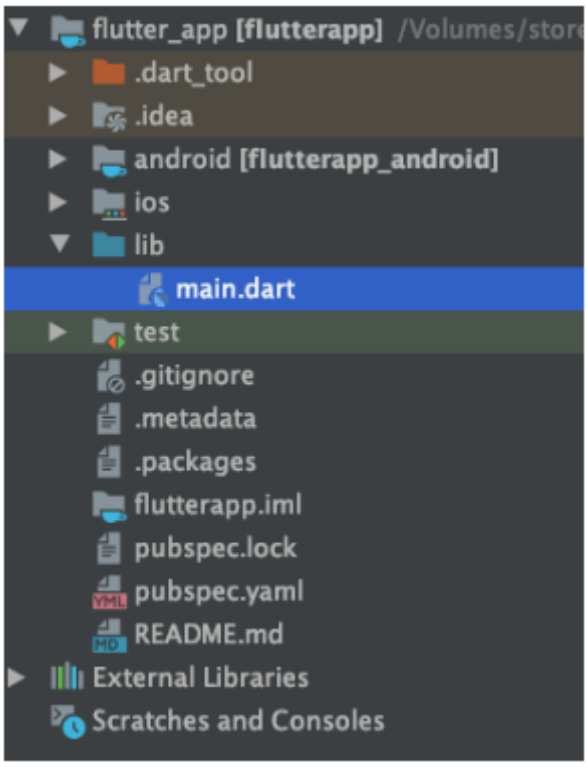
安装完成之后重启编译器生效。

补充:
这时候第一个坑出现了，以为可以开心的去学习Flutter了，结果发现无法新建Flutter项目，在File->New之后只有New Project，没有找到New Flutter Project，后一番搜索发现需要在Preferences->Plugins勾选上 Android APK Support、Android NDK Support这两个插件。

重启完之后我们可以看到New Flutter Project的选项了，赶紧新建一个来试一试。

2.Flutter之初始Dart

上篇介绍了Flutter环境的搭建，并且创建了第一个Flutter项目，我们可以先运行一下万一也有一个Hello World等着呢，运行之后真的有一个示例项目，点击下面的按钮还能给我们看到数字的增加，那回来看一下Flutter项目的结构。



android，ios对应两个平台的包，lib是我们主要写Flutter的包，pubspec.yaml是Flutter的配置文件，我们以后要添加的library都是需要在这里进行配置。
先可以不用管这些项目的结构，我们直接打开lib里面的main.dart看下我们最关心的Flutter到底长什么样子。说实话我第一次打开的时候我是蒙的，这都是什么，淡定一下心神，一起来来揭开它的神秘面纱。

第一印象

先不用管它dart的语法，我们先来大概看一眼，可以简单总结一下看到的内容。

1. 看到了熟悉的main函数，这里应该就是程序的入口了。还有个跟Lambda表达式很像的符号=>，估计是一样的作用。

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

2.通过下面的代码可以看到跟Java一样，也是通过Class来声明类，通过extends来声明继承，这里跟Java很像了，也是一样熟悉的Override，熟悉的方法格式，熟悉的返回格式，一开始害怕太难的顾虑消失了一半，通过上面的runApp里的参数我们差不多可以了解到这里build就是构建了我们一开始运行的页面，`MaterialApp()`里面的参数格式可能就陌生了，怎么参数都还带有名称呢？先记下来。

```
class MyApp extends StatelessWidget{
    @override
    Widget build(BuildContext context){
        return MaterialApp(
        title: 'Flutter Demo',
        ....
        );
    }
}
```

3.接着往后面看，这里的参数为什么会有一个大括号，super前面为什么是:呢？这里也是我们以后需要多加注意的地方。

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

其他的基本上只剩一些控件的使用了，当我们学会dart语法后再来看这些控件就会简单很多了，一开始的我有一些疑问，给大家总结一下：

- 我们原来布局用的xml和xib都去哪里了？
  Flutter没有再使用xml和xib来进行布局的编写，而是我们现在看到的一个个Widget来实现的，在Flutter中所有的都是组件，这个概念会一直伴随我们对Flutter的学习。
- 是否可以实时查看布局的样式呢？
  现在还没有，看到网上一些截图说是支持了Hot UI，可能是测试版本的IDE，不过应该很快就会看到，现在支持的Hot Reload也能让我们快速的看到运行效果，比原生的每次重新build再运行看到结果快了不知道多少倍。

接下来踏踏实实的去学习Dart。

3.Flutter之Dart语法基础

今天的内容是后面学习Flutter的基础。如果把开发Flutter应用比喻成盖楼的话，今天的知识就是那些砖。一定要把今天的内容掌握牢，后面学起来才会更轻松。

一、变量

在开始讲变量之前,要先给大家看下变量的声明，举三个例子。

```
var name = 'Cyy'
```

这行代码的意思就是，我声明了一个变量，名字是name,然后给他了一个初始值Cyy。

再看下一个

```
dynamic name = 'Cyy';
```

这行代码的意思也是，我声明了一个变量，名字是name,然后给他了一个初始值Cyy。

我们继续看下一个

```
String name = 'Cyy';
```

这行代码的意思还是声明了一个变量，名字是name，然后赋值为Cyy。

但是！他与上面两个的区别是，我在这里给他声明了变量的类型，就是我指定了这个name是字符串类型的。上面两个都没有给变量指明类型，那么问题来了！

二、var和dynamic的区别是什么呢?

var 初始化确定类型后不可更改类型，dynamic 可以更改类型

简单说，就是如果你一开始给var的变量初始化了一个值，Dart会去推断这个值，如果是什么类型，那这个变量以后就是什么类型了，不能再改变了。但是 dynamic还是可以更改滴。

现在再来看看这三个东西吧～

默认值

在Dart中,所有的类型都是对象，未初始化的变量的初始值为null。甚至Number类型的变量最初都为null，怎么理解呢，就像java里的包装类，Integer , Double 等等。

final

```
final name = 'Cyy';
final String nickname = 'XiaoFo';
```

简单来说，被final声明过的变量，后面你就无法再更改它的值了,所以final修饰的变量必须初始化，且只能在初始化时赋值一次.

举个例子:

```
void main() {
  final name = 'Cyy';
  final String nickname = 'XiaoFo';
  name = 'Cyy513'
}
```

运行结果: Error: Can't assign to the final variable 'name'.

它不会让我再去给他赋值,因为我声明它是final的变量，只允许被进行一次赋值。

const

在基础课，我们只需要知道const和final一样也只允许被赋值一次，后面不能再更改。他主要用来创建常量值、声明创建常量值的构造函数。后面用到的时候，会给大家细讲，基础阶段先暂时知道这些就可以。

三、常用的变量类型

Number

Number对象包含两个子对象，分别是int和double。int在不同平台位数不同,但是最大就64位。在Dart VM上，他的值从-263 到 263 - 1。这个数字啥意思，就是说int允许的最大值和最小值。double学名是双精度浮点数，初学者的话，可以这么理解，整数是不带小数点儿的，double是带小数点儿的。

怎么声明一个int型变量和double型变量呢？这样就可以了

```
int a = 2;
double b = 1.1;
String
```

就是你声明的变量是一个字符串,举个例子应该就都明白了。

```
 String cyy  = "hello Cyy"
```

也是非常简单,但是他有很多种用法：

```
  var sOne = 'Hello';
  var stwo = 'Cyy';
  var sThree = sOne + stwo; // 输出结果：HelloCyy
  var sFour = '${sOne}ABC'; //HelloABC
  var sFive = '$sOne $stwo'; //Hello Cyy
  var sSix = '''
  You can create
  multi-line strings like this one.
  ''';
  sSix的输出结果为:
  You can create
  multi-line strings like this one.
  ------------------------------------
  var sSeven= """This is also a
  multi-line string.""";
  sSeven的输出结果为:
  This is also a
    multi-line string
```

会这几个就够用了~

Boolean

布尔类型,只允许两种值，true和false.

List

任何编程语言都有集合，而Dart 的集合是List。

```
  //创建一个int类型的list
  List list = [1, 2, 3];
  // 输出[1, 2 3]
  print(list);
  有两种方式可以创建List
  方法一:
  // 使用List的构造函数
  var listOne = new List();
  // 添加元素
  listOne.add('Mario');
  // 添加多个元素
  listOne.addAll(['chun-li', 'jack']);
```

```
 ----------------------------------------
 //也可在括号里添加数字，表示List固定长度，但是不能进行添加 删除操作
  var listTwo = new List(10);
 如果执行了:listTwo.add('Mario');
 输出结果:Uncaught Error: Unsupported operation: add
 ----------------------------------------
 List listThree = ['Cyy', 'XiaoFo', 'Bob'];
 // 添加多个元素
 listThree.addAll(listOne);
 // 输出：[Cyy, XiaoFo, Bob, Mario, chun-li, banans]
 print(listThree);
 // 获取List的长度
 print(listThree.length);
 // 获取第一个元素
 print(listThree.first);
 // 获取元素最后一个元素
 print(listThree.last);
 // 利用索引获取元素
 print(listThree[0]);
----------------------------------------
List还有其他几种有趣的玩儿法
第一种:您可以使用运算符（...）将列表的所有元素插入另一个列表
var list = [1, 2, 3];
var list2 = [0, ...list];
print(list2);
输出结果:[0, 1, 2, 3]
第二种:
/*
  如果扩展运算符右边的表达式可能为空，
  则可以使用可识别空值的扩展运算符（...? ）避免出现异常
*/
var list;
var list2 = [0, ...?list];
print(list2);
输出结果:[0]
第三种:还可以使用for循环添加元素到集合中
var nav = [
  'Home',
  'Furniture',
  'Plants',
  if (true) 'Outlet'
];
print(nav);
输出结果:[Home, Furniture, Plants, Outlet]
Set
List 是有序的集合，那Set就是无序的集合。
var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};
//创建一个空Set集合
var names = <String>{};
//添加单个元素
names.add('fluorine');
//添加多个元素
names.addAll(halogens);
```

Map

map是将键和值相关联的对象。键和值都可以是任何类型的对象，Map的键是唯一的。

```
//键可以是整型也可以是字符串
var gifts = Map();
gifts['first'] = 'partridge';
gifts['second'] = 'turtledoves';
gifts['fifth'] = 'golden rings';
var nobleGases = Map();
nobleGases[2] = 'helium';
nobleGases[10] = 'neon';
nobleGases[18] = 'argon';
//使用.length来获取映射中的键值对的数量
var gifts = {'first': 'partridge'};
print(gifts.length)
// 指定键值对的参数类型
var myMap = new Map<int, String>();
// 检索Map是否含有某Key
myMap.containsKey(1);
//删除某个键值对
myMap.remove(1);
```

四、int、double、string相互转换

```
// String -> int
var one = int.parse('1');
// String -> double
var onePointOne = double.parse('1.1');
// int -> String
String oneAsString = 1.toString();
// double -> String
String piAsString = 3.14159.toStringAsFixed(2);
```

方法

参数

举一个函数的例子,你应该就知道什么是参数了。

```
// 调用函数时，可以使用paramName：value指定命名参数，例如:
enableFlags(bold: true, hidden: false);
// 定义函数时，使用{param1，param2，…}指定命名参数，
// bold和hidden就是参数，bool是他们的类型
void enableFlags({bool bold, bool hidden}) {...}
// 尽管命名参数是一种可选参数，但是您可以使用@required对其进行注释
// 以指示该参数是强制性的-用户必须为该参数提供一个值
// 如果用户不传，就会报错!
const Scrollbar({Key key, @required Widget child})
//在[]中包装一组功能参数会将其标记为可选参数，就是可以传，也可以不传
// 例如:
String say(String from, String msg, [String device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
可以这么调用:say('Bob', 'Howdy', 'smoke signal')
也可以这么调用：say('Bob', 'Howdy')
```

默认参数

您的函数可以使用=来定义命名参数和位置参数的默认值。默认值必须是编译时常量。如果未提供默认值，则默认值为null。举个栗子：

```
void enableFlags({bool bold = false, bool hidden = false}) {
}
调用：enableFlags(bold: true); //bold是true hidden是fals
可选参数使用默认值:
String say(String from, String msg,
    [String device = 'carrier pigeon', String mood]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  if (mood != null) {
    result = '$result (in a $mood mood)';
  }
  return result;
}
调用:say('Bob', 'Howdy')
输出结果为:Bob says Howdy with a carrier pigeon
-----------------------------------------------
您还可以将列表或地图作为默认值传递,例如:
void doStuff(
    {List<int> list = const [1, 2, 3],
    Map<String, String> gifts = const {
      'first': 'paper',
      'second': 'cotton',
      'third': 'leather'
    }}) {
  print('list:  $list');
  print('gifts: $gifts');
}
调用:doStuff()
输出结果:
list:  [1, 2, 3]
gifts: {first: paper, second: cotton, third: leather}
是不是很神奇,希望大家回去后，都自己去亲自敲一遍。
```

main函数

每个应用程序都必须有一个顶层main()函数，它可以作为应用程序的入口点。该main()函数返回void，视频中也说了，void就是不返回任何值。并具有List参数的可选参数。

```
void main() {
  //在这里写你的逻辑
}
```

匿名函数

正常的函数是这样的

```
void setName(String name){
}
```

是这样的

```
String getName(){
}
```

匿名函数是这样的

```
(){
}
```

这是啥？像不像一个人没有头？

但是他跟普通函数一样，能传各种参数，普通函数能传啥参数就，他也能传。

那他平时咋用呢？再举个例子:

```
void main() {
  var list = ['apples', 'bananas', 'oranges'];
  list.forEach((item) {
    print('${list.indexOf(item)}: $item');
  });
}
```

打印结果:

```
0: apples
1: bananas
2: oranges
```

这个item叫啥都行，你传个a进去，效果是一样的。

老师在视频中也说了，如果该函数仅包含一个语句，则可以使用箭头符号将其缩短。那么上面那个代码就可以改成:

```
void main() {
  var list = ['apples', 'bananas', 'oranges'];
  list.forEach(
    (item) => print('${list.indexOf(item)}: $item'));
}
```

效果是一样的。

返回值

```
foo(){
}
```

所有函数都返回一个值。如果未指定返回值，则语句返回null;

void就是没有返回值

词法范围

Dart 是静态作用域语言，变量的作用域在写代码的时候就确定过了。基本上大括号里面定义的变量就只能在大括号里面访问，和 Java 作用域 类似。举个例子:

```
var topLevel = true;
main() {
  var insideMain = true;
  myFunction() {
    var insideFunction = true;
    nestedFunction() {
      var insideNestedFunction = true;
    }
  }
}
nestedFunction可以访问:
topLevel,insideMain,insideFunction,insideNestedFunction 4个变量
他拥有最高权限。
myFunction可以访问:
topLevel,insideMain,insideFunction 3个变量
main可以访问:
topLevel,insideMain 2个变量
```

思考一下：如果main访问了insideFunction 会提示什么?

答案：会提示"找不到insideFunction";

五、词法闭包

概念太抽象了，但是我还是要写一下，哈哈, 我好幽默。

> 维基百科上对闭包的解释就很经典：在计算机科学中，闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。

函数可以关闭周围范围中定义的变量。

在以下示例中，makeAdder（）捕获变量addBy。

无论返回的函数到哪里，它都会记住addBy。

```
Function makeAdder(num addBy) {
  return (num i) => addBy + I;
}
void main() {
  // 引用了自由变量2
  var add2 = makeAdder(2);
  // 引用了自由变量4
  var add4 = makeAdder(4);
  print(add2(3));
  print(add4(3));
}
运行结果:
5
7
这个要多品品~不懂的一定要来群里问，大家一起讨论，这个一定要懂!
```

六、控制流和语句

if和else

Dart支持if语句和可选的else语句，条件判断语句中必须是布尔值，举个例子。

```
if (isRaining()) {
  //如果isRaining()返回true,则走这里
  you.bringRainCoat();
} else if (isSnowing()) {
  //如果isRaining()返回false,isSnowing()返回true,则走这里
  you.wearJacket();
} else {
  //如果isRaining()返回false,isSnowing()返回false,则走这里
  car.putTopDown();
}
```

for循环

```
var message = StringBuffer('Dart is fun');
for (var i = 0; i < 5; i++) {
  message.write('!');
}
pritn(message);
输出结果:Dart is fun!!!!
List和Set之类的可迭代类也支持for-in形式的迭代
var collection = [0, 1, 2];
for (var x in collection) {
  print(x); //输出结果 0 1 2
}
```

while 和do-while

while:只有测试条件成立，才会去执行循环体中的语句，否则跳出循环。

```
// 只有isDone()返回false时，才会去执行doSomething
while (!isDone()) {
  doSomething();
}
```

do-while:

只有循环体中的语句被执行后，才去测试循环条件，

只有循环条件成立，就继续执行下去，不成立就跳出循环。

```
//先去执行一次printLine()，再去判断atEndOfPage()是否返回false
do {
  printLine();
} while (!atEndOfPage());

break和continue

使用break终止循环
while (true) {
  if (shutDownRequested()) break;
  processIncomingRequests();
}
使用continue跳出当前循环，进行下一次循环
for (int i = 0; i < candidates.length; i++) {
  var candidate = candidates[i];
  if (candidate.yearsExperience < 5) {
```

```
    // 如果candidate.yearsExperience<5
    // 不执行candidate.interview();继续开始下一次循环
    continue;
  }
  candidate.interview();
}
```

switch和case

看个例子就明白了。

```
var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'PENDING':
    executePending();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'DENIED':
    executeDenied();
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown();
}
相当于:
if(command=='OPEN'){
}else if(command=='CLOSED'){
}...后面省略
但是要强调几点:
1. case后面如果省略了break，他就跳不出这个循环，会走到下一个case下面.
2. case子句可以声明局部变量，这些局部变量仅在该子句的范围内可见
3. Dart的case里面，逻辑可以是空，例如:
var command = 'CLOSED';
switch (command) {
  case 'CLOSED': // Empty case falls through.
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}
```

assert

在开发过程中，使用断言语句— assert(); 如果布尔条件为false，则中断执行

七、类

类的使用

对象具有由函数和数据（分别为方法和实例变量）组成。调用方法时，您可以在对象上调用它，该方法可以访问该对象的功能和数据。

```
var p = Point(2, 2);//Point是一个类
// 给Point类中的y赋值为3
p.y = 3;
// 打印p.y
print(p.y);// 3
采用 ？.代替 .为了避免p为null时发生异常
p?.y = 4;
构造函数的使用

您可以使用构造函数创建对象。构造函数名称可以是ClassName或ClassName.identifier。
class Point {
  double x, y;
  //这个就是Point这个对象的构造函数
  Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
this关键字指向了当前类的实例，上面的代码可以简化为
 class Point {
    num x;
    num y;
    // 语法棒棒糖
    Point(this.x, this.y);
 }
-----------------命名构造函数--------------------------------
命名构造函数:是在初始化时可直接使用类调用
class User {
    String name;
    int    age;
    //声明一个命名构造函数
    User.getInfo(String name, int age){
        this.name = name;
        this.age  = age;
        print("命名构造函数");
    }
}
//调用命名构造函数
User u = User.getInfo();

-----------------使用构造函数创建对象--------------------------
例如，以下代码使用Point（）和Point.fromJson（）构造函数创建Point对象
var p1 = Point(2, 2);
var p2 = Point.fromJson({'x': 1, 'y': 2});
也在构造函数名称之前使用可选的new关键字创建对象
var p1 = new Point(2, 2);
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

创建构造函数还有很多种方法，这里就不一一列举了，这里只列举了两种，总共大概有八九种，大家可以去网上看下，很多这样的文章，写的都很好。我们继续看下面的。

获取对象的类型

要在运行时获取对象的类型，可以使用Object的runtimeType属性，会返回Type对象。

```
print('The type of a is ${a.runtimeType}');
```

八、方法

方法是提对象行为的函数，可以通过他访问对象的变量。

```
class Rectangle {
   num left;
   num top;
   num width;
   num height;

   Rectangle(this.left, this.top, this.width, this.height);

   // 定义两个计算属性: right and bottom.
   num get right => left + width;
   set right(num value) => left = value - width;
   num get bottom => top + height;
   set bottom(num value) => top = value - height;
}

main() {
   var rect = new Rectangle(3, 4, 20, 15);
   print(rect.left);//调用了隐含的getLeft 所以是3
   rect.right = 12;//执行了上面right方法
   print(rect.left);//这里left = 12-20 这块儿是个匿名函数
}
运行结果:3 -8
这个例子多看一会儿，思考一下。
```

抽象方法

Instance， getter 和 setter 方法可以是抽象的，也就是定义一个接口，但是把实现交给其他的类。要创建一个抽象方法，使用分号（；）代替方法体。

```
 abstract class Doer {
    // 定义实例变量和方法
    void doSomething(); // 定义一个抽象方法。
 }

 class EffectiveDoer extends Doer {
     void doSomething() {
        // 提供一个实现
     }
 }
 ! *****抽象类是不能被实例化的****! 这句话很重要，需要被注意到，还需要细品
 下面的类不是抽象类，因此它可以被实例化，即使定义了一个抽象方法
  class SpecializedContainer extends AbstractContainer {
    // 定义更多构造函数，域，方法

    void updateChildren() {
      // 实现 updateChildren()
    }
   // 抽象方法造成一个警告，但是不会阻止实例化。
   void doSomething();
 }
```

隐式接口

每个类都隐式地定义一个接口，该接口包含类的所有实例成员及其实现的任何接口。如果您想创建一个类A，它支持类B的API而不继承B的实现，那么类A应该实现B接口。

简单的说，当我们定义了一个类的时候，同时就会产生一个和此类同名的接口，而且此接口包含了我们定义的类中所有的方法，以及它的成员变量。

```
//定义一个父类Vehicle
class Person{
  num age  = 0  ;
  Person(){
    print("super Constructor") ;
  }
  Person.get(){
    print("super Run") ;
  }
  Person.create(){
    print("super create") ;
  }
  void run(){
    print("person run") ;
  }
  void printAge(){
    print("age = > $age") ;
  }
}
怎么用?
class Cyy implements Person{
  @override
  void run() {
    //重写了run方法
    print("Cyy running");
  }
  @override
  void printAge() {
    print("Cyy age = $age") ;
  }
  //覆盖(实现)成员变量
  @override
  num age;
}
```

继承

使用 extends 创建一个子类，同时 supper 将指向父类

```
 class Television {
    void turnOn() {
        _illuminateDisplay();
        _activateIrSensor();
    }
 }

 class SmartTelevision extends Television {

    void turnOn() {
       super.turnOn();
       _bootNetworkInterface();
```

```
      _initializeMemory();
      _upgradeApps();
    }
}
当实例化SmartTelevision这个类后，调用turnOn()方法时会执行:
_illuminateDisplay(),_activateIrSensor(),
_bootNetworkInterface(),_initializeMemory(),
_upgradeApps() 5个方法
```

枚举类型

枚举类型，通常被称为 enumerations 或 enums ，是一种用来代表一个固定数量的常量的特殊类。

声明一个枚举类型需要使用关键字 enum

```
enum Color {
   Mario,
   chun-li,
   Rose
}
要得到枚举列表的所有值，可使用枚举的 values 常量
例如:Color.values
```

Mixins

因为我们课程是基础课程，所以Mixins这里就暂时先不讲了，如果后面有需要，我们会单独写一片文章来讲解。但是我会把我觉得写的比较好的文章，放到今日推荐里，感兴趣的同学可以去看看。

九、常用的操作符

算数运算符

算数运算符，这里就不一一列出来了，大家看下方推荐文章，官方文档里有写，如果有不懂的，可以下方留言或者加群，问我们，肯定知无不言。

等号和关系运算符

要测试两个对象x和y是否表示同一事物，请使用==运算符。（在极少数情况下，您需要知道两个对象是否是完全相同的对象，请改用same()函数）

as is is!

主要是在运行过程中，用来检查类型的。

```
当且仅当您确定对象属于该类型时，才使用as运算符将对象转换为特定类型
(emp as Person).firstName = 'Bob';
也就是说，只有你确定emp是Person类型时，才能这么用。
不然会报错:TypeError: "emp": type 'JSString' is not a subtype of type 'Person'
---------------------------------------------------------------------------
如果您不确定该对象的类型为T，使用该对象之前使用is检查类型。
if (emp is Person) {
  // Type check
  emp.firstName = 'Bob';
}
```

赋值运算符

```
// 把value 赋值给 a
a = value;
// 如果b为null，则为b赋值；否则，b保持不变
b ??= value;
```

条件表达式

condition ? expr1 : expr2

如果条件为true，则返回expr1否则返回expr2。

expr1 ?? expr2

如果expr1不为null，则返回expr1否则返回expr2。

级联符号

```
看一个例子就明白了,暂时知道怎么用就行.
class Person {
  var name;
  var age;
  Person(this.name, this.age);
  getUserInfo() {
    print("${this.name},${this.age}");
  }
}

void main() {
  var p = new Person('Cyy', 20);
  p.getInfo();
  //..为级联操作,可以同时赋值执行方法
  p
    ..name = "Mario"
    ..age = 30
    ..getInfo();
}
```

运行结果:

```
Cyy,20
Mario,30
```

好啦，今天的内容实在太多了，但我认为只要好好读，一定能学懂的

4.Flutter之Dart的集合和控制流程

Dart的集合和控制流程

Dart跟我们常用的语言也是有List，Map，Set这些类型的集合类型，只是有的用法跟我们以前接触的略有不同。

1.List
一个集合的初始化可以通过三种形式，第一行创建了一个固定长度的列表，如果出现了超过长度则会直接报错，后两种都是可以创建一个可变长度的集合，在后面通过add和insert方法添加单个元素，通过addAll添加一组元素。

```
List sizeList = List(3);
List emptyList = List();
List list = ["11","22","33","44"];
emptyList.add("55");
list.addAll(emptyList);
// sizeList.addAll(list);Cannot add to a fixed-length list
list.insert(0, "00");
print(list);//[00, 11, 22, 33, 44, 55]
```

2.List遍历
在集合中我们最常用到的就是对集合的遍历，给大家看几种比较常用的遍历方式

```
  //方式一
for (var i = 0;i < list.length;i++) {
  print("item is "+list[i]);
}
  //方式二
for (var item in list) {
  print("item is $item");
}
  //方式三
list.forEach((item){
  print("item is $item");
});
```

还有一种常用的遍历是使用list的every方法，特别之处在于它返回的是一个bool类型，在碰到不满足条件的元素后会返回False，并停止循环，如果所有元素都满足条件则返回True。

```
var result = list.every((item){
  var s = item as String;
  print("刚刚进来的数据是$s");
  /*
  刚刚进来的数据是00
  刚刚进来的数据是11
  刚刚进来的数据是22
  刚刚进来的数据是33
  刚刚进来的数据是44 */
  return s != '44';
});
print("result is $result");//result is false
```

3. Set
Set的初始化只有两种形式，Set跟其他语言的Set一样都是自带了去重特性，只是它在去重完了还是保持着一个有序的集合

```
var setList = {"a","b","c","d","a","b"};
print(setList);//{a, b, c, d}
var set = Set();
set.add("a");
set.add("b");
set.add("c");
print(set);//{a, b, c}
```

4.Set遍历
Set的循环也可以使用上面List中介绍的循环方式，还介绍一种也是常用的适用于所有集合的一种遍历所有元素的方法，map方法。特别之处在于在遍历的同时我们可以修改元素的内容，获得到一个修改完的集合。

```
var setMap = set.map((item){
  if(item is String)//数据类型的判断，is操作符判断是否后面指定的类型，非操作为is!
    return item.toUpperCase();
});
print(setMap);//(A, B, C)
```

5.Map
Map的初始化也是有两种形式，跟Set一样它也是在有相同Key的时候去重完了保持一个有序的状态。可以通过使用containsKey和containsValue方法来查询想要找的元素是否存在。

```
var mapList = {'a':'11','b':'22','c':'33','c':'44'};
print(mapList);//{a: 11, b: 22, c: 44}
var map = Map();
map["aa"] = "11";
map["bb"] = "22";
map["cc"] = "33";
map["aa"] = "44";
print(map);//{aa: 44, bb: 22, cc: 33}
print(map.containsKey("aa"));//true
print(map.containsValue("11"));//false
```

7.控制语句
上面在各种集合的介绍里顺便加入了基本的几种控制流程的语句，有if...else...有三种for循环的写法，和forEach的用法，继续说一下switch...case...和do...while的用法。
乍一看感觉switch跟原来也没有区别，但是细看会发现在case2上多了一个label，而且这个label还出现在了我们调用之后，是不是感受到了状态机的味道，以后的多状态处理就可以使用它了。

```
switch (1) {
   case 0:
     print("0");
     break;
   case 1:
     print("1");// 1
     continue nums;
   nums:
   case 2:
     print("2");// 2
     break;
   default:
 }
```

do...while和while...do的用法跟其他语言的用法是一样的，没有其他需要注意的地方了。

```
  List list = ["11","22","33","44"];
  var i = 0;
  do {
    i++;
  } while (i<list.length);
  while(i > 0){
    i--;
  }
```

以上是Dart中的集合和控制流程语句基本用法，还有一些高级的用法可以在以后的学习中慢慢加进来。

接下来学习Dart中的函数

5.Flutter之Dart的函数

Dart的函数

与有些语言有些区别的是Dart的函数也是作为对象存在的，也就是说可以作为方法中的参数，也可以赋值给变量，举个例子：

```
void main() {
  var fun = (str){
      print("str is $str");
  };//将一个匿名函数作为参数进行赋值
  fun("aaa");//str is aaa  调用这个匿名函数并传入参数为"aaa"
  var nFun = printStr;//将函数作为变量进行赋值，注意：赋值函数的时候函数名后面没有括号！！
  fun("hello nFun");//str is hello nFun 调用函数并传入参数
  sendMsg(printStr);//将刚刚赋值给变量的函数作为参数传给sendMsg方法
}
//传入一个函数
void sendMsg(void f(dat)){
  for(int i =0;i<5;i++){
    f(i);//执行传入的函数
  }
}
//创建一个打印字符串的函数
Function printStr(str){
  print("str is $str");
}
```

乍看起来还是比较难以理解的，我们还是先简单说一下在Dart中基本的函数定义和调用方法。

```
返回类型  方法名(形参){
  return 返回值
}
```

这样的结构是一个标准的函数的样式，在Dart中还扩展了一些更加方便我们操作的写法，在Dart中一个方法最简化可以怎么写?

```
nunFun(str)=>str;
//可以翻译为
String nunFun(String str){
    return str;
}
```

返回的数据类型可以省略，在返回后动态判断，传入的数据变量也是可以省略，这样只需要一个变量名就写好了一个方法。
在很多语言中我们有重载来定义多个同名方法，但是Dart中是不允许同一个类中有同名函数的，但是有很多脚本语言中的可选参数来代替，个人感觉真是很爽的体验。可选参数有两种方式，一种是根据名称可选，另一种根据位置可选。来个例子：

```
//根据名称可选
nameParams("Dart",param3:"Java");//不是可选的直接传参，可选的参数需要用 形参名:value 的方式传入
void nameParams(String param1,{String param2,String param3}){//使用{}来包裹名称可选参数
  print("param1->$param1,param2->$param2,param3->$param3");//param1->Dart,param2->null,param3->Java   因为param2没有传入，所以这里的值为null
}
//根据位置可选
positionParams("Dart","","Java");//根据参数位置依次传入值，如果只想传入第一个和第三个那也必须传入第二个，个人感觉不如名称可选灵活。
void positionParams(String param1,[String param2,String param3]){//使用[]来包裹位置可选参数
  print("param1->$param1,param2->$param2,param3->$param3");//param1->Dart,param2->,param3->Java//因为param2传了空字符串，所以这里打印没有null了。
}
```

可以使用@required关键字来声明该参数是需要传的，但是不传我在实际体验中也没有感受到有强制的效果。

```
void nameParams(String param1,{String param2,@required String param3}){...}
//使用required必须要导入import 'package:meta/meta.dart';
```

在Dart中还有一个比较赞的是跟很多脚本语言一样可以直接给形参赋值，这样我们在没有传入这个参数的时候也就有了一个默认值。

```
nameParams("Dart",param3:"Java"); //这里并没有给param2进行传参操作
void nameParams(String param1,{String param2="Object-c",@required String param3}){
  print("param1->$param1,param2->$param2,param3->$param3");//param1->Dart,param2->Object-c,param3->Java  通过给定的param2的默认值，我们在打印的时候看到了param2是有值的。
}
```

在Dart中也有函数嵌套，内层的函数可以获取到外层函数的变量，变量的声明周期在该方法内都是有效的。

```
layerFun(){
  var index = 3;
  print("layerFun");
  layer1Fun(){
    var index1 = 4;
    index = index1;
    print("layerFun1");
    layer2Fun(){
      var index2 = 5;
      index1 = index2;
      print("index->$index,index1->$index1,index2->$index2");
```

```
    }
  }
}
```

在Dart中也有函数嵌套，内层的函数可以获取到外层函数的变量，变量的声明周期在该方法内都是有效的。

```
layerFun(){
  var index = 3;
  print("layerFun");
  layer1Fun(){
    var index1 = 4;
    index = index1;
    print("layerFun1");
    layer2Fun(){
      var index2 = 5;
      index1 = index2;
      print("index->$index,index1->$index1,index2->$index2");
    }
  }
}
```

现在再回过头来看一开始写的将函数作为变量值和方法的参数就好理解了一些，Flutter是万物皆组件，Dart是万物皆对象

接下来学习Dart的类

6.Flutter之Dart的类

在我们一开始创建了第一个项目的时候已经大概看了一眼Dart中的类的样子，接下来我们需要好好研究一下它，首先先写一个朴素的类:

```
class Person{
  String name;
}
```

这是一个单纯的类的构成，构造方法为隐藏的，属性只有一个name属性。在Dart中只能有个构造方法，构造方法中可以使用可选参数的方式来实现原来我们有多个构造函数的需求，当然也有其他的方式我们一起看下：

```
void main() {
  var hanMeimei = Person(name:"HanMeimei",sex:"female");//使用带有参数的构造方法创建一个韩梅梅
  var liLei = Person.withName("LiLei");//使用只传名字的方式创建一个李雷
  print(hanMeimei);//通过重写了Object的toString方法来打印了韩梅梅的信息
  print(liLei);//通过重写了Object的toString方法来打印了李雷的信息
}

class Person{
  String name;
  String sex;
  int age;
  //有参数的构造方法
```

```
  Person({String name,String sex,int age = 15}){//参数可以是用可选参数的形式，解决了需要传不同参数的问题
    this.name = name;
    this.sex = sex;
    this.age = age;
  }
  // Person({this.name,this.sex,this.age = 15});  这是对上面有参数的构造方法的简写，这也是以后要写的形式
  //创建一个只需要传入Name的构造方法
  Person.withName(String name){
    this.name = name;
  }
  @override
  toString(){
    return "name is $name,sex is $sex age is $age";
  }
}
```

在上面的例子中我们使用修改toString的方式来打印出了想要的信息，还有没有其他的方式呢，是有的，在Dart中我们可以实现call()函数来让对象作为方法去使用。在上面的代码中添加：

```
  void call(){
      print("name is ${this.name},sex is ${this.sex} age is ${this.age}");
  }
```

在main中添加下面的代码，可以看到直接得到了打印结果，当然call方法也是可以添加参数的，可以自己试一下。

```
  hanMeimei();//name is HanMeimei,sex is female age is 15
  liLei();//name is LiLei,sex is null age is null
```

在Dart里类里的属性是在该library可见的，也就是我们在类中的私有属性是可以在该dart文件中都可以正常访问的，可以通过修改get，set方法来访问私有属性，属性也需要注意不要保存可以通过计算获得的属性。我们新建一个rect.dart文件来看一下私有属性和计算属性。

```
class Rect{
  num _width;
  num _height;
  //计算属性，不需要存储
  num get area => _width*_height;
  set area(num value){
      _width = value/10;
  }
  num get width => _width;//无法直接访问私有属性_width,需要通过get方法获取
  set width(num value) => _width = value;//无法直接设置_width的值，需要通过set方法设置

  num get height => _height;//无法直接访问私有属性_height,需要通过get方法获取
  set height(num value) => _height = value;//无法直接设置_heigh的值，需要通过set方法设置
}
```

在main函数的dart文件中引入该library

```
import 'rect.dart';
```

然后在main中调用该Rect类，发现无法直接通过对象使用属性，只能通过set方法，获取到area属性跟别的属性一样的方式，然而我们并没有在rect.dart中单独去定义area属性。

```
  var rect = Rect();
  rect.height = 10;
  rect.width = 5;
  print(rect.area);//50
```

Dart中的类的基本使用和类中属性的介绍就算是结束了，可以看到和我们其他语言有很多共通之处，只是在写法上需要多注意一些细节。

接下来学习Dart的类继承

7.Flutter之Dart的类继承

Dart的类继承

前面介绍了在Dart中的类和属性的写法已经特点，接下来学习它的继承、接口、抽象类
Dart也是单继承的方式，我们如果要继承其他的类需要在类名后面添加extends关键字。
在上节介绍的类里的Person作为父类，添加一个学生类:

```
class Student extends Person{//学生类继承了Person类的所有公有属性和方法
  String grader;
  Student({String name,String sex,int age =
15,this.grader}):super(name:name,sex:sex,age:age); //子类的构造方法，子类的构造方法里没有办法再添加新的代码了，只能给父类和自己的属性赋值了。
  //Student({String name,String sex,int age = 15,String grader}):this.grader =
grader,super(name:name,sex:sex,age:age);  构造方法还可以这样写
  //覆写了Person类的call方法
  @override
  void call(){
    print("student name is $name,age is $age,grader is ${this.grader}");
  }
}
```

在main方法中的使用跟直接使用Person类的时候是一样的:

```
 var student = Student(name:"LiLei",grader: "七年级二班");
 student();//student name is LiLei,age is 15,grader is 七年级二班
```

在Dart也是有有抽象类的存在，与其他语音一样抽象类是无法直接被实例化的，我们需要创建一个类来实现抽象类中的方法。

```
class MathTeacher extends Teacher{
  MathTeacher({String name,String sex,int
age}):super(name:name,sex:sex,age:age);
  @override
  void teach(){
    print("math teacher name is $name");
  }
}

abstract class Teacher extends Person{
  Teacher({String name,String sex,int age}):super(name:name,sex:sex,age:age);
  void teach();
}
```

不过有一个特别的使用，在Dart是没有interface的，可以使用抽象类来代替，还是用老师的例子，数学老师的工作是教数学。

```
abstract class Work{//创建一个抽象类来代替的interface
  void doWork();
}

class MathTeacher extends Teacher implements Work{//抽象类可以用来继承也可以用
implements
  MathTeacher({String name,String sex,int
age}):super(name:name,sex:sex,age:age);
  @override
  void teach(){
    print("math teacher name is $name");
  }
  void doWork(){//实现implements里的方法
    print("math teachers teach math");
  }
}
```

然后我们在main方法中调用老师work的方法。

```
Work work = MathTeacher(name:"张",sex:"female");
work.doWork();//math teachers teach math
```

当然我们使用普通的任何类都是可以当做接口类来使用的，但是我们需要重写原来类的所有属性和方法，所以一般情况我们还是使用一个抽象类来作为接口类。

在Dart中还有一个概念是Mixins，我们开始说了Dart也是单继承的，但是我们可以通过with关键字来将多个类的属性和方法合并到一个类中，如果出现同名方法则会根据with后跟的类的先后属性进行覆盖，with也是可以单独使用省略掉extends关键字。with的类当中不能有显示声明的构造方法并且只能是继承自Object。

```
class Test extends Test1 with Test2,Test3{//with多个类时中间用逗号分隔。

}
//class Test = Test1 with Test2,Test3;上面的代码可以简写成这样，看起来像把他们组合在了一
起

class Test1{
  void test(){
```

```
    print("test1");
  }
}

class Test2{
  void test(){
    print("test2");
  }
}

class Test3{
  void test(){
    print("test3");
  }
}
```

在main函数中我们调用Test的test方法试一下

```
var test = Test();
test.test();//test3
```

看到结果是打印输出了我们最后一个with的方法。

接下来学习Dart的异步操作

8.Flutter之Dart的异步操作

我们经常在开发的时候面临上传下载等耗时操作，这时候需要请我们的异步模块出场了，使用异步的目的是为了在我们在执行一段耗时代码的时候不至于UI卡主用户无法操作。
首先我们可以使用Future来帮助我们创建一个简单的异步请求。

```
  Future<String> getData(){
    return Future<String>((){//返回一个Future实例。
      sleep(Duration(seconds: 3));//这是是用到了延时操作，让程序睡3秒。
      return "获取到了数据";
    });
  }
```

在我们拿到这个Future之后就可以使用then方法来获取到里面的值。

```
  print("testSyn start");
  getData().then((value){ //这里调用getData()获取到的是Future对象，需要通过then方法来获取里面的值
    print("value is $value");//打印获取到的值
  });
  print("testSyn end");
  /*输出结果
    testSyn start
    testSyn end
    value is 获取到了数据*/
```

这是一个简单的异步操作，如果在异步操作中出现了异常，我们还需要用catchError方法来捕获产生的异常，然后会异常进行处理操作。

```
 Future<String> getNetData(){
    return Future<String>((){
      sleep(Duration(seconds: 3));
      throw Exception("timeout exception");//过了三秒钟之后抛出一个连接超时的异常。
    });
  }
```

在调用出我们需要捕捉该异常：

```
print("testSyn start");
getNetData().then((value){
    print("value is $value");
  }).catchError((error){//在then方法后面直接抓取异常
    print("error is $error");
  });
print("testSyn end");
```

Future可以使用链式,每一次都是返回的一个Future来让后面的方法进行调用。

```
  getData().then((value){
    print("value is $value");
    return "return future first";
  }).then((value){
    print("value is $value");
    return "return future second";
  }).then((value){
    print("result is $value");
  }).catchError((error){
    print("error is $error");
  });
  print("testSyn end");
  /*运行结果
  value is 获取到了数据
  value is return future first
  result is return future second  */
```

在我们日常使用中我们异步的调用需要使用await、async这两个关键字，这两个关键字都是同时出现的，在调用了await的方法上需要加上async关键字。

```
 Future<String> getNetworkData() async {
    var response = await getData();//去调用的getData方法并获取到了里面返回的值。
    return "response is $response";
  }
```

我们在main函数里的调用:

```
  getNetworkData().then((value){
    print(value);//response is 获取到了数据
  });
```

再附上一段官网的示例，我们可以从中学到更多的用法： 再附上一段官网的示例，我们可以从中学到更多的用法：

```
void printOrderMessage () async {
```

```
  print('Awaiting user order...');
  var order = await fetchUserOrder();
  print('Your order is: $order');
}

Future<String> fetchUserOrder() {
  // Imagine that this function is more complex and slow.
  return Future.delayed(Duration(seconds: 4), () => 'Large Latte');
}

Future<void>main() async {
  countSeconds(4);
  await printOrderMessage();
}

// You can ignore this function - it's here to visualize delay time in this
example.
void countSeconds(s) {
  for( var i = 1 ; i <= s; i++ ) {
      Future.delayed(Duration(seconds: i), () => print(i));
   }
}
/**运行结果为:
Awaiting user order...
1
2
3
4
Your order is: Large Latte
*/
```

Dart 的基础知识差不多就介绍完了，遗漏的部分可以在以后Flutter的学习中慢慢的再加入进来。又可以回到我们的Flutter的学习了。

接下来讲解Flutter的基本控件

9.Flutter之基本布局

我们之前说过Flutter中都是由组件构成的，组件分成两种，一种是可变状态的Widget继承自`StatefulWidget`，一种是不可变的Widget继承自`StatelessWidget`，有什么区别呢？在`StatefulWidget`中可以通过`setState()`方法来通知组件来调用自己的`build()`方法来刷新页面。在`StatelessWidget`就缺失了`setState`方法，就固定状态不能改变了。
看下我们原来新建的项目这个页面里就包含了这两种组件的运用，这里的MyApp的整体结构不会发生改变所以直接使用了`Stateless`的widget来显示页面，但是页面还是会在我们点击按钮的时候点击次数

进行了刷新，所以我们需要对`MyHomePage`继承自`StatefulWidget`。

```
class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ), // ThemeData
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    ); // MaterialApp
  }
```

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;
  @override
  _MyHomePageState createState() => _MyHomePageState();
}
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    nameParams("Dart",param2:"Java");
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'You have pushed the button this many times:',
            ), // Text
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.display1,
            ), // Text
          ], // <Widget>[]
        ), // Column
      ), // Center
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ), // This trailing comma makes auto-formatting nicer for build me
    ); // Scaffold
```

可以看到在`StatefulWidget`里创建了一个State，在State里我们就包含了`setState()`和`build()`方法，当我们调用setState方法的时候就会去刷新一下页面，也就更新了Text里的数据。
了解了可变状态组件后就知道我们是如何进行页面交互的，现在可以看下这个页面如何通过一个个Widget来搭建的。

介绍一下我们常用的布局类，看我们新建项目中的已经包含的Column还有Row和Flex，Wrap，Stack。我们依次对他们进行一个介绍：

Column 子元素垂直排列
将所有子元素按照竖直方向进行排列。
`mainAxisAlignment`
排列的方式，默认值`MainAxisAlignment.start`
`MainAxisAlignment.start` 子元素在顶部垂直排列
`MainAxisAlignment.end` 子元素在底部垂直排列
`MainAxisAlignment.center` 子元素在居中垂直排列
`MainAxisAlignment.spaceBetween` 子元素在空间里均匀的放置
`MainAxisAlignment.spaceAround` 根据元素的数量平分空间，子元素分别居中。
`MainAxisAlignment.spaceEvenly` 根据元素的数量+1平分空间，子元素分别居中。

`crossAxisAlignment`
横向显示的方式，默认值`CrossAxisAlignment.center`
`CrossAxisAlignment.start` 子元素靠左边显示
`CrossAxisAlignment.center` 子元素居中显示
`CrossAxisAlignment.end` 子元素的靠右边显示
`CrossAxisAlignment.stretch` 子元素占满横向空间
`CrossAxisAlignment.baseline` 效果和start是一样的

`children`
我们需要给其添加的子元素。
其他的不常用属性就不多做介绍了，我们可以在需要的时候来看一下就可以。

Row子元素横向排列
用法与Column一致，只是子元素的排列是按横向排列。

Flex弹性布局
我们打开Column和Row的源码看一下发现这两个布局都是Flex的子类，区别在于给他们的direction设置了不同的方向，用法也是一致的，所以都是用Column和Row来做具体的事情。

Wrap流式布局
作为一个可以多行展示的布局，可以在我们添加元素显示超过一行时可以自动折行。

`direction`
子元素的排列方式，默认值Axis.horizontal，所有子元素横向排列。
`Axis.horizontal` 子元素横向排列
`Axis.vertical` 子元素纵向排列

alignment
子元素的排列方向，默认值`WrapAlignment.start`，所有子元素左对齐。
`WrapAlignment.start` 子元素靠左边显示
`WrapAlignment.center` 子元素居中显示
`WrapAlignment.end` 子元素的靠右边显示
`WrapAlignment.spaceBetween` 同`WrapAlignment.start`相同
`WrapAlignment.spaceAround` 同`WrapAlignment.center`相同
`WrapAlignment.spaceEvenly` 同`WrapAlignment.center`相同

`spacing`
两个子元素的间距，默认值是0

`runSpacing`
自动折行后两行之间的间距，默认值是0
其他的不常用属性就不多做介绍了，我们可以在需要的时候来看一下就可以。

`children`
我们需要给其添加的子元素。

Stack层叠布局

`alignment`
未指定定位的子元素默认定位方式，默认值`AlignmentDirectional.topStart`
`AlignmentDirectional.topStart` 所有子元素堆叠到左上角
`AlignmentDirectional.topCenter` 所有子元素堆叠到顶部中间
`AlignmentDirectional.topEnd` 所有子元素堆叠到顶部右边
`AlignmentDirectional.centerStart` 所有子元素堆叠到中间的左部
`AlignmentDirectional.center` 所有子元素堆叠到中间
`AlignmentDirectional.centerEnd` 所有子元素堆叠到中间的右边
`AlignmentDirectional.bottomStart` 所有子元素堆叠到底部左侧
`AlignmentDirectional.bottomCenter` 所有子元素堆叠到底部的中间
`AlignmentDirectional.bottomEnd` 所有子元素堆叠到底部的右边

`fit`
用于没有指定位置的子元素占用的空间,默认值`StackFit.loose`
`StackFit.expand` 所有子元素占满父空间
`StackFit.loose` 所有子元素根据指定的大小显示

`overflow`
如果子组件的大小超过了Stack的大小，是否进行裁切，默认值：`Overflow.clip`
`Overflow.clip` 超过的部分将进行裁切
`Overflow.visible` 超过的部分也会显示出来

children
我们需要给其添加的子元素。

一些基本的布局Widget就介绍的差不多了，可以多个组合连多试一下。

10.Flutter之基本容器

Flutter官方并没有对Widget进行官方分类，其实对于容器和布局类型的划分比较纠结，有些感觉不是很明确怎么去划分到哪个部分，先按容器下可以添加子控件的数量作为分类标准，添加多个子控件的为布局，对单个组件进行设置的为容器，如果以后有好的方法再进行调整。
这里分出来的容器有Padding，Align，Positioned，Container，Scaffold，Transform，ConstrainedBox，SizedBox，RotatedBox。容器比较多，来一个一个的学习他们的特点。

一、Padding

容器的内填充，刚一开始看到padding肯定会想到它的好兄弟margin，在Flutter中去除掉了margin的概念，我们只能是通过使用padding来代替margin的实现。

padding
我们需要内填充的大小，这里不是传入具体的数值，需要传入`EdgeInsetsGeometry`，通过它可以设置填充的位置以及大小。
如果我们只想填充某几个边，不是全部的时候，可以使用`EdgeInsets.only`，可以根据上下左右来设

定。

```
const EdgeInsets.only({
    this.left = 0.0,
    this.top = 0.0,
    this.right = 0.0,
    this.bottom = 0.0,
  });
```

如果我们要设定的是全部，则推荐使用EdgeInsets.all，可以直接设定全部边的内边距。

```
const EdgeInsets.all(double value)
    : left = value,
      top = value,
      right = value,
      bottom = value;
```

child
添加内边距的控件

二、Align

子控件相对于父控件的位置，可以回看上篇中的Stack，如果没有指定子控件的位置的话都是根据Stack中的alignment显示的，如果给子控件设定了Align则会摆脱它的束缚。

`alignment`
子控件位于父容器的位置，默认设置`Alignment.center`
`Alignment.topLeft` 子控件位于顶部左上角
`Alignment.topCenter` 子控件位于顶部中间
`Alignment.topRight` 子控件位于顶部右边
`Alignment.centerLeft` 子控件位于中间的左部
`Alignment.center` 子控件位于中间
`Alignment.centerRight` 子控件位于中间的右边
`Alignment.bottomLeft` 子控件位于底部左侧
`Alignment.bottomCenter` 子控件位于底部的中间
`Alignment.bottomRight` 子控件位于底部的右边

child
需要指定位置的子控件

三、Positioned子控件在Stack布局中的位置，相当于绝对定位。

left
距离屏幕左边的距离
top
距离屏幕顶部的距离
right
距离屏幕右边的距离
bottom
距离屏幕底部的距离
width
子控件的宽度

height
子控件的高度

注意： (left,right,width)(top,bottom,height)这两组属性都是只能设置其中的两个，不能三个全部设置！！

child
需要指定位置的子控件

四、Container

是不是觉得上面的各种设置会很麻烦，Container不仅包含了Align和Padding的功能还有一些更强大的功能。

alignment
可以参照Align的设置

padding
可以参照Padding的设置

color
Container的背景色
Flutter内置了丰富的颜色，我们可以直接Colors里面的颜色如：Colors.black，如果有自定义颜色的要求，我们可以Color(0x000000)

decoration
可以设置Container背景的渐变色，圆角和阴影颜色，和color属性之能选择一个设置。
需要传入BoxDecoration(),这个属性我们看基本控件的时候再去了解它。

constraints
设置容器的大小，如果设置了则widht和height的属性会失效
BoxConstraints.tightFor 可以只设置宽高某一个边的大小
BoxConstraints.tight 必须设置宽高的大小

transform
旋转视图通过Matrix4进行设置，常用的方法有：
Matrix4.translationValues` 控制控件的移动方向x :水平方向,y： 竖直方向, z：垂直于屏幕，所以看不出来效果，
Matrix4.rotationZ 控制控件的旋转。Matrix4.rotationX和Matrix4.rotationY不常用，都是以侧边为轴进行旋转的操作。rotationX以顶为轴旋转，rotationY以左侧边为轴旋转

foregroundDecoration
在Container上加的遮罩颜色，设置的方式与decoration相同。

width
组件的宽

height
组件的高

margin
这里Container帮助我们实现了margin功能，传值方法同padding的方法，他的值不算在宽高里。

child
需要指定位置的子控件

五、Scaffold

终于到它了，在我们的demo中就看到了它的身影，可以看到他负责了我们整个页面的结构。先了解一下它都给我们准备好了哪些功能。

appBar
就是我们的标题栏，我们需要添加一个AppBar组件，AppBar比较大我们也放到后面看基础组件的时候去了解。最基本的添加一个AppBar我们可以设置

```
AppBar(title: Text(widget.title),)
```

body
标题栏下面的整个空间的设置我们都要通过这里进行设置，这里可以添加的我们上一篇提到的布局类。

floatingActionButton
我们在页面上通过点击然后增加数量的按钮就是通过它来显示出来的，我们也将这个控件放到后面基础组件的时候去了解。

**floatingActionButtonLocation `设置悬浮按钮显示的位置`
FloatingActionButtonLocation.startTop,FloatingActionButtonLocation.miniStartTop `显示在左上角`
FloatingActionButtonLocation.endFloat `显示在右下角但是离底部有距离`
FloatingActionButtonLocation.endDocked `显示在右下角紧挨着底部`
FloatingActionButtonLocation.centerFloat `显示在底部中间但是离底部有距离`
FloatingActionButtonLocation.centerDocked `显示在底部中间紧挨着底部`
FloatingActionButtonLocation.endTop` 显示在右上角

`floatingActionButtonAnimator`

悬浮按钮的显示隐藏动画，系统内置了`FloatingActionButtonAnimator.scaling`动画，如果想要自定义动画可以参照`FloatingActionButtonAnimator.scaling`创建的示例去编写自己喜欢的动画。
`persistentFooterButtons`
在页面的最底部添加按钮，添加底部切换按钮我们使用`bottomNavigationBar`,所以一般不会去使用`persistentFooterButtons`。

drawer和endDrawer
抽屉效果的抽屉布局,只是一个是左抽屉，一个是右抽屉。
里面想要添加的布局可以是我们上一章说的布局类，也可以是容器类。

bottomNavigationBar
这里可以添加我们常用的布局样式，但是我们一般是使用BottomNavigationBar类来设置显示的底部按钮，具体的BottomNavigationBar的介绍我们等讲到基础控件的时候再去详细了解。

bottomSheet
这里的布局会持续的呆在底部，如果软键盘被唤起则跟随软键盘一起升高，常用的场景类似微信的文字输入的底部栏。

backgroundColor
设置底部栏的背景色

resizeToAvoidBottomInset
设置键盘弹起时是否会遮挡底部的布局，false则会进行遮挡，true则不会进行遮挡。

primary
布局开始计算的位置是否包括状态栏，设置为False则会从屏幕的最顶端开始计算，默认值True是从状态栏下开始计算。

drawerDragStartBehavior
处理拖动行为的开始方式，默认是DragStartBehavior.start，替换了DragStartBehavior.down也没有看出来改变是什么。

extendBody
控制body底部的可显示范围是否在bottomNavigationBar和persistentFooterButtons之上，如果设置为True则会直接显示到屏幕的底部，而不是bottomNavigationBar和persistentFooterButtons的上面。

extendBodyBehindAppBar
更上面的类似，这个是控制是否是从屏幕顶部开始显示，而不是从AppBar下面开始显示。

drawerScrimColor
设置当Drawer打开的时候下面内容的遮挡颜色

drawerEdgeDragWidth
设置Drawer可以展开的宽度
我们平时设计APP内的常用控件基本上都帮忙已经预留了位置，我们只需要按需填坑就可以了，还是挺不错的设计。

六、Transform

来设置控件的旋转和偏移。
transform
跟Container中介绍的transform一样，参考上面的设置。

alignment
相对于坐标系原点的对齐方式，对齐方式参照Align的alignment进行设置。

origin
相对于起始位置进行的一个偏移，传入Offset(偏移的X，偏移的Y)。

七、ConstrainedBox

尺寸限定容器，我们可以通过它设定一个控件的最小最大宽度，高度。
minWidth
最小宽度，默认0

maxWidth
最大宽度，默认double.infinity，没有限制

minHeight
最小高度，默认0

maxHeight
最大高度，默认double.infinity，没有限制

child
用于设定的子控件

八、SizedBox

尺寸固定的容器，设定完之后子控件的大小就固定了
width
子控件的宽度
height
子控件的高度
child
用于固定大小的子控件

九、RotatedBox

旋转子控件，功能于Matrix4.rotationZ一样
quarterTurns
沿着顺时针方向旋转90度的次数。
child
用于旋转的子控件

基本上我们常用的容器控件就介绍完了，这里内容还是比较多的，还是需要多去练习好好理解其中的意思。

接下来学习Flutter的基本控件

11.Flutter之基本组件1

Flutter给我们提供了丰富的组件来搭建UI，我们可以通过这些组件搭建出我们想要的APP，由于组件太多我们需要分成几次来学习这些组件了，先从已经见过的demo中出现的AppBar，FloatingActionButton，Text，BottomNavigationBar开始认识。

一、Text

先看我们使用频率最高也是最基本的控件Text，先看下平时我们怎么使用的：

```
Text createText(String str) => Text(str,
    style:TextStyle(color: Colors.red,fontSize: 14),
    textAlign:TextAlign.left,
    maxLines: 3);
```

结合我们在Dart中的类里面的内容，可以看到我们如果想要实现一个最基本的Text只需要传入一个data，也就是要显示的字符。其他的可选参数可以根据我们想要实现的效果来传参。

style
根据需要传入TextStyle示例，参数比较简单，基本上都是一眼看上去就知道是做什么的了。

```
const TextStyle({
    this.inherit = true, //控制文字显示，False为不显示
    this.color, //字体颜色
    this.backgroundColor, //背景色
    this.fontSize,//字体大小
    this.fontWeight,//字重，加粗字体可以设置
    this.fontStyle, // 文字样式，像斜体
    this.letterSpacing, //字符间距
    this.wordSpacing, // 字间距
    this.textBaseline, //设置文字的基线
    this.height,    //行高
    this.locale,    //设定一些特殊符号可以设置
    this.foreground,//前景色
    this.background,//背景色
    this.shadows, //设置阴影
    this.fontFeatures,//字体字形列表
    this.decoration, //给文字添加画线，比如下划线
    this.decorationColor, //划线的颜色
    this.decorationStyle, //划线的样式，实线，虚线
    this.decorationThickness,//划线的宽度
    this.debugLabel,//无实际用处
    String fontFamily, //字体
    List<String> fontFamilyFallback,//字体列表，如果没有设置fountFamily则从里面开始找
```

```
    String package,//用哪种字体
  })
```

strutStyle
如果没有设置文字样式，可以使用这里的设置，可以传入StrutStyle实体。传参的参数同TextStyle。

textAlign
文字相对于父容器的对齐方式。
TextAlign.left 左对齐
TextAlign.right 右对齐
TextAlign.center 居中对齐
TextAlign.justify 拉伸文字填满父容器
TextAlign.start 根据文字的方向，不管是从左到右还是从右到左，根据开始位置对齐
TextAlign.end 根据文字的方向，不管是从左到右还是从右到左，根据结束位置对齐

textDirection
文字的显示方向，可以是从左到右也可以从右到左

locale
设定一些特殊符号可以设置，比如日语和法语时的特殊符号是设定。

softWrap
如果设定为False为只有一行且水平方向无限长。

overflow
如果指定行数后超出的部分裁切方式。
TextOverflow.clip 直接裁断
TextOverflow.fade 会有一个渐变的裁断
TextOverflow.ellipsis 裁断后面的最后加上省略号
TextOverflow.visible 全部显示不裁断，但是溢出的无法看到

textScaleFactor
文字相对于当前设定的文字大小的缩放因子。

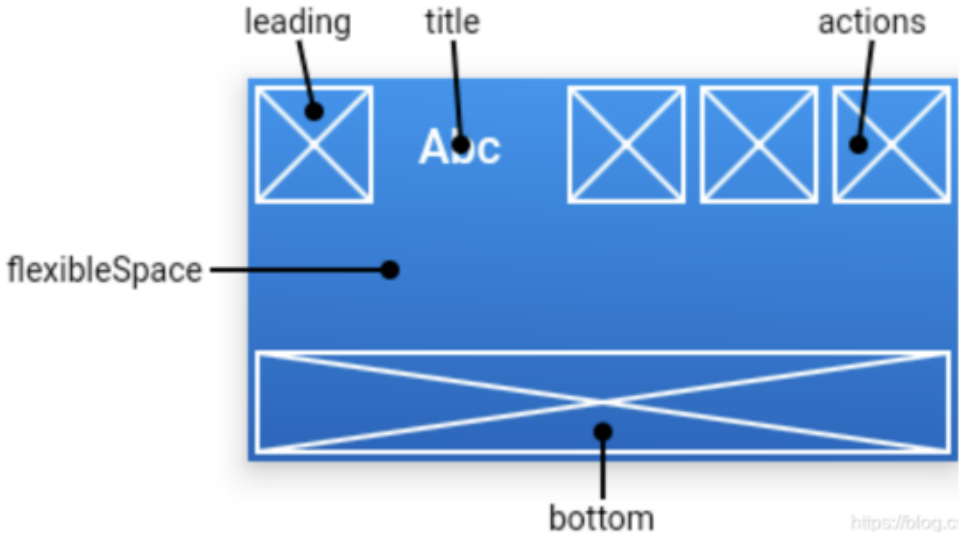
maxLines
文字最多显示行数

semanticsLabel
没有实际意义

textWidthBasis
一行或者多行文本宽度的不同形式，没有看到实际变化
可以看到Text比我们原生的多了很多属性，很多原来实现比较复杂的功能都有了好的解决方法。还是比较强大的。

二、APPBar

顶部的AppBar，我们在每个页面都会见到的控件，在Flutter中也进行了比较好的封装。



leading
在title之前的控件，我们可以根据需要做自定义设置，也可以使用系统提供给我们的样式。

automaticallyImplyLeading
在没有自定义leading的时候如果有侧边栏False则不会显示leading，True会显示默认的leading并相应点击事件。

title
显示的标题。

actions
一组右边的按钮，可以是文字也可以是图片按钮。

flexibleSpace
处于appbar和tabbar之下的一个控件，与appbar、tabbar一样的高度，可以通过它来设置appbar和tabbar的背景

bottom
设置在AppBar底部的控件显示。一般会设置tabbar。

elevation
设置阴影的大小

shape
AppBar和它的影子的形状。

backgroundColor
设置背景色，与flexibleSpace的区别在于这里只能设置为一个color，上面有很多可以的操作。

brightness
设定状态栏的亮度，状态栏上的文字依据设备信息的图标也会跟着变化。
Brightness.light 为亮色，则上面的文字和图标都会变成黑色
Brightness.dark 为深色，则上面的文字和图标都会变成白色

iconTheme
设定图标的大小，透明度以及颜色。通过IconThemeData实例进行设置。

actionsIconTheme
同理设置的是actionIcon。

textTheme
设定title的文字属性。

primary
AppBar开始的位置，True是从状态栏下面开始，False是从屏幕顶部开始。

centerTitle
标题是否居中显示

titleSpacing
title和leading之间的距离，默认是0。

toolbarOpacity
导航栏内容的透明度，取值0-1，0为隐藏，1为显示。

bottomOpacity
在bottom 设置的底部布局的透明度，取值0-1，0为隐藏，1为显示。
AppBar差不多就结束了，可以看到我们基本上常用的功能都已经有了，我们只要按需填空就可以得到我们想要的AppBar了。

三、FloatingActionButton
在底部的悬浮按钮，我们可以控制他的显示位置。

child
可以添加的按钮上的元素，可以是图片也可以使文字。

tooltip
长按按钮时我们要显示的提示。

foregroundColor
前景色

backgroundColor
背景色

elevation
阴影的大小。

onPressed
在按下的时候我们需要处理的响应，可以参照demo中按下后调用数字加一。

mini
显示一个小的按钮，True为显示小按钮，False显示默认大按钮。

shape
控制按钮的显示样子。

focusNode
用以控制焦点的监听，暂时还不知道怎么使用。

autofocus
是否开启自动获取焦点，默认值False。

materialTapTargetSize
设置目标大小。
`MaterialTapTargetSize.padded` 设定最小的目标大小为48x48，
`MaterialTapTargetSize.shrinkWrap` 设定为目标指定的最小大小。
但是实际体验后感觉没有变化。

isExtended
是否是被扩展的button，默认是False，如果我们调用的是extended的指定构造方法则会被赋值为True。

四、BottomNavigationBar

底部导航栏，一般也是我们程序首页的时候最常用的控件，切换底部的导航栏来在首页显示多个页面。

```
BottomNavigationBar({
    Key key,
    @required this.items,//参照下面详解
    this.onTap,//参照下面详解
    this.currentIndex = 0, //可以指定刚进入的时候默认选中的位置。
    this.elevation = 8.0,//阴影大小。
    BottomNavigationBarType type,//参照下面详解
    Color fixedColor,//与selectedItemColor类似，不可以同时设置
    this.backgroundColor,//整体的背景色
    this.iconSize = 24.0, //每个item里的图片大小
    Color selectedItemColor, //选中时的文字以及icon的颜色
    this.unselectedItemColor,//未选中时文字以及icon的颜色
    this.selectedIconTheme = const IconThemeData(),
    this.unselectedIconTheme = const IconThemeData(),
    this.selectedFontSize = 14.0,//选中时的字体大小
    this.unselectedFontSize = 12.0,//未选中时的字体大小
    this.selectedLabelStyle,//设置选中时的字体大小等样式
    this.unselectedLabelStyle,//设置未选中时的字体大小等样式
    this.showSelectedLabels = true,//item被选中时是否显示底下的文字
    bool showUnselectedLabels,//在未选中的时候是否显示底下的文字
  })
```

items

最关键的内容，我们需要添加的NavigationBar的元素，可以添加一组的`BottomNavigationBarItem`。`BottomNavigationBarItem`也是比较简单：

```
const BottomNavigationBarItem({
    @required this.icon,//上面添加的icon
    this.title, //需要显示的文字
    Widget activeIcon, //当该item被点击选中的时候显示的icon
    this.backgroundColor, //背景色
  })
```

onTap

`BottomNavigationBar`中某一个item被点击后会触发方法被调用，返回被点击的位置。

```
onTap: (position){},
```

type
设置显示的样式
`BottomNavigationBarType.fixed` 小于四个item的时候系统默认会设置为fix，这时会给他自动加上主题颜色。
`BottomNavigationBarType.shifting` 大于等于四个item的时候系统设置会shifting，需要自己加上设定颜色。

先了解一下以上控件的基本属性，日常使用基本上都是满足我们的需求的。

接下来还要学习Flutter基本组件

12.Flutter之基本组件2

上一篇我们了解了AppBar，FloatingActionButton，Text，BottomNavigationBar的基本使用，这篇将会介绍TabBar，TabBarView，Image

一、TabBar

也是我们经常使用的控件，在现在的APP中越来越多的信息需要展示的时候，tabbar的出现可以解决页面冗长无序的问题，可以根据内容的分类进行划分tab。

```
const TabBar({
    Key key,
    @required this.tabs,//参照下面详解
    this.controller,//参照下面详解
    this.isScrollable = false,//如果tab很多是否可以滚动显示，如果是False会平分宽度以全部显示，如果是True则会居中显示全部的tab
    this.indicatorColor,//在选中的tab下方的指示器颜色
    this.indicatorWeight = 2.0, //指示器的高度
    this.indicatorPadding = EdgeInsets.zero,//指示器底部添加的padding
    this.indicator,//设定指示器的样式，默认的为UnderlineTabIndicator，我们可以仿照它写自己想要的指示器的样式
    this.indicatorSize,//参照下面详解
    this.labelColor,//选中label的颜色
    this.labelStyle,//选中label的样式，传入的是TextStyle
    this.labelPadding,//可以给label加padding，padding的添加方式也是通过EdgeInsets类来添加。
    this.unselectedLabelColor,//没有选中的label的颜色
    this.unselectedLabelStyle,//没有选中的label的样式，传入的是TextStyle
    this.dragStartBehavior = DragStartBehavior.start,
    this.onTap,//点击事件响应，调用给返回点击的位置position
  })
```

tabs

需要添加的tab元素，也就是根据需要设定我们要显示的头部。会添加一组Tab信息，先看下单个的Tab。

```
const Tab({
    Key key,
    this.text,//想要显示的标题文字
    this.icon,//可以自己添加Icon
    this.child,//扩展的其他的想要添加的内容
  })
```

controller

我们需要传入TabController对象，根据名字也看得出来是控制Tab的一个控制器，看下如何创建该对象。

```
TabController({
int initialIndex = 0,//初始化时默认选中的位置
@required this.length, //Tab要显示的数量
@required TickerProvider vsync //控制Tab和TabView的同步，我们可以直接让创建Controller的类继承或者with SingleTickerProviderStateMixin
})
```

创建一个`controller_controller = TabController(vsync: this, length: tabs.length);`这里的this也就是当前类对象，我们还需要让当前类继承自`SingleTickerProviderStateMixin`，我们可以这样写：`class _MyHomePageState extends State<MyHomePage> with SingleTickerProviderStateMixin`还记得这个with关键字吧，如果忘记了可以回去看下Dart的类继承 里面的介绍。
indicatorSize
指示器的宽度
`TabBarIndicatorSize.tab` 和tab整个的宽度一致
`TabBarIndicatorSize.label` 和tab上面文字的宽度一致

二、TabBarView

是和TabBar一起搭配使用的，可以根据选择的tab显示不同的页面

```
const TabBarView({
    Key key,
    @required this.children,
    this.controller,
    this.physics,
    this.dragStartBehavior = DragStartBehavior.start,
  })
```

children

也即是我们需要添加的TabBarView的布局页面，这里可以根据需要添加我们前面说到的容器等组件。给出一个简单的示例：

```
TabBarView(
   controller: _controller,
   children: tabs.map(
             (Tab tab) => Container(
                         child: Center(
                       child: Text(tab.text),
              )),
              ).toList())
```

用到的是我们前面说到的map关键字，通过map我们将tab转为一个个的Center容器。然后直接将整个数组用来显示。

controller
同TabBar的Controller，一般将他们的Controller用一个变量来设置，来统一行为。

physics
控制tabBarView在滑动的操作，默认为ScrollPhysics，如果想要修改滑动操作可以仿照该类进行重写。

dragStartBehavior
拖动的滑动行为，在默认滑动的时候默认值为`DragStartBehavior.start`，自测的在修改为`DragStartBehavior.down`后也没有看到变化。

三、Image

Image可以是在我们日常使用中最频繁的了，也给我们提供了多种显示图片的方法，我们可以根据需要来进行图片的显示。先看下最基础的Image是需要我们如何做的

```
 const Image({
    Key key,
    @required this.image,//需要自己实现的获取图片的方式。
    this.frameBuilder,//参照下面详细解释
    this.loadingBuilder,//参照下面详细解释
    this.semanticLabel,//关于图片的描述，不重要
    this.excludeFromSemantics = false, //排除image的语义，True则会忽略semanticLabel的内容
    this.width,//设置显示的宽度
    this.height,//设置显示的高度，需要注意的是如果宽高小于图片的宽高则会以最短边为基准等比例的裁剪，居中显示，如果宽高大于图片的宽高则图片不拉伸居中显示
    this.color, //设置图片的前景色，图片的主色会根据设定变化
    this.colorBlendMode,//图片的混合模式，一般与color配合使用
    this.fit,//参照下面详细解释
    this.alignment = Alignment.center,//对齐方式,前面列举了很多，不再赘述
    this.repeat = ImageRepeat.noRepeat,//参照下面详细解释
    this.centerSlice,//参照下面详细解释
    this.matchTextDirection = false,//参照下面详细解释
    this.gaplessPlayback = false,//图片的路径变化时是否保留显示原图片，True是保留，False则不会保留，重新加载的时候会空白一直等到图片完全加载完
    this.filterQuality = FilterQuality.low,//图片的质量控制
  })
```

frameBuilder

可以给Image添加边框以及间距，也可以做出堆叠其他的控件的效果，列举一个添加边框的例子：

```
frameBuilder: (BuildContext context, Widget child, int frame,
                      bool wasSynchronouslyLoaded) {
                    return Padding(
                      padding: EdgeInsets.only(left: 5, top: 5),
                      child: child,
                    );
                  },
```

loadingBuilder

在图片加载的时候，还没有加载完成之前的显示，当加载完成之后就会被image覆盖掉。注意是覆盖不是替换，如果加载的是一个透明或者半透的图则会显示出来loadingBuilder的组件。列举一个简单实用：

```
loadingBuilder: (BuildContext context, Widget child,
                      ImageChunkEvent loadingProgress) {
                    return Container(
                        color: Colors.yellow,
                        child: child);
                  },
```

fit

在如果宽高都大于图片的时候可以通过fit属性填充多出来的空间。
`BoxFit.none` 不设置
`BoxFit.fill` 宽高填满，会引起图片的拉伸
`BoxFit.contain` 以最大边为基准缩放图片，完全显示出图片，会有空白
`BoxFit.cover` 以最小边为基准缩放图片，填充满图片，会被裁切
`BoxFit.fitWidth` 图片比例不变，宽填充满，会有空白
`BoxFit.fitHeight` 图片比例不变，高填充满，会有空白
`BoxFit.scaleDown` 在不大于原图尺寸的时候会等比例缩小，大于图片尺寸则会原图尺寸根据alignment设置

repeat
对于没有填充满的区域是否以原图片重复添加的方式填充
`ImageRepeat.repeat` X轴和Y轴的空白空间都进行填充
`ImageRepeat.repeat` X X轴的空白空间进行填充
`ImageRepeat.repeat` Y Y轴的空白空间进行填充
`ImageRepeat.noRepeat` 不进行填充

centerSlice
我们如果对一个图片的局部进行拉伸，则可以通过该属性进行设置，需要传入一个Rect的对象，我们可以根据需要创建该实例，举个简单的例子：

```
centerSlice: Rect.fromLTRB(0,20,40,59),
```

用图片的宽为0-40，高为20-59的部分进行图片的拉伸，并填充满整个image。

matchTextDirection
控制图片的显示方向是否跟文字的显示方向一致，需要配合Directionality容器使用，Directionality容器可以控制子控件的显示方向，如果设置为false则不跟随Directionality设置的方向显示，True则会跟随一样显示。举个简单例子：

```
 Directionality(
                textDirection: TextDirection.rtl,
                child: Image.network(
                  "https://img.jpg",
                  matchTextDirection: true,
                  ...})
```

上面刚刚介绍的是图片的默认构造方法，这里使用的时候我们还需要自己来写一个加载方式，否则无法显示出来图片，但是Flutter已经帮我们创建了几种加载方式，我们只需要根据场景选择合适的加载方式即可。

方式一：

```
Image.asset()
```

在工程的根目录下创建assets/images目录，将要显示的本地图片放到该目录下。
在根目录下的`pubspec.yaml`文件中添加assets的引用：

```
flutter:
  uses-material-design: true
  assets:
    - assets/images/
```

这样我们才可以在程序中引用到该图片。在使用中与基本的Image会有下面几点不同：

```
Image.asset(
    String name, {
    AssetBundle bundle,//设定要读取的资源Bundle，如果不设置则会从rootBundle里获取
    String package,//在取别的library的资源时候需要添加包名
    int cacheWidth,//指定的缓存的宽
    int cacheHeight,//指定的缓存的高
    ...
  })
```

方式二：

```
Image.network();
```

从网络上获取图片，需要注意添加网络权限，在android的manifest文件中添加：

```
    <uses-permission android:name="android.permission.INTERNET"/>
```

将图片的地址直接传入即可，其他参数与assets没有区别。

方式三：

```
Image.file();
```

需要传入一个File对象，File对象的需要我们导入IO包，`import 'dart:io' show File;`
File的创建也很简单，我们只需要传入文件的路径即可创建成功。其他参数与assets没有区别。

方式四：

```
Image.memory();
```

从内存中的bytes集合中获取，将其传入参数中进行显示，其他参数与assets没有区别。

基本上我们通过这两篇的学习可以写一个简单的小demo跑起来了，还是需要多加练习，单看这些属性有些无法理解的很透彻还是需要都实现一遍来看看效果的。

接下来学习Flutter基础组件的第三部分

13.Flutter之基本组件3

上一篇我们一起学习了TabBar，TabBarView，Image，接下来一起来学习Button，TextField和Card。

一、Button

在Flutter中的Button从大的风格上来划分，可以划分为MaterialButton(Google推荐的风格)和CupertinoButton(IOS推荐风格)，这篇主要是介绍MaterialButton，CupertinoButton是比较简单的，可以直接参照它的构造进行了解。

MaterialButton是RaisedButton，FlatButton的父类，还有很多种类的Button，例如：`DropdownButton，FloatingActionButton，IconButton，OutlineButton`，我们一个个来介绍它们。
首先我们可以先了解下`MaterialButton`：

`MaterialButton`

我们先看下它的构造方法，发现它有着很多的参数：

```
const MaterialButton({
    Key key,
    @required this.onPressed,//按钮的点击事件被触发调用此方法
    this.onLongPress,//按钮长按事件被触发调用此方法
    this.onHighlightChanged,//高亮状态的变化通知，会返回是否高亮的bool值，一般在按下会返回True，松开会返回False
    this.textTheme,//参照下面详解
    this.textColor,//设置的文字的颜色
    this.disabledTextColor,//当文字不可用时文字的颜色
    this.color,//设置的按钮的颜色
    this.disabledColor,//按钮不可用的时候的颜色
    this.focusColor,//按钮获取焦点时的颜色
    this.hoverColor,//是在H5的时候鼠标悬停的颜色
    this.highlightColor,//按下高亮时的颜色
    this.splashColor,//水波纹效果的颜色
    this.colorBrightness,//关于主题的亮度，Brightness.dark深色，Brightness.light浅色的
    this.elevation,//阴影的长度
    this.focusElevation,//获取焦点时阴影的长度
    this.hoverElevation,//H5的鼠标悬浮焦点时阴影的长度
    this.highlightElevation,//高亮时的阴影高度
    this.disabledElevation,//禁用时的阴影高度
    this.padding,//内部填充空白大小
    this.shape,//设定的形状,例如我们要创建一个圆形的按钮：shape: CircleBorder()
    this.clipBehavior = Clip.none,//参照下面详解，
    this.focusNode,//焦点的关联组件
    this.autofocus = false,//是否自动获取焦点
    this.materialTapTargetSize,//配置最小尺寸，参照前面的FloatingActionButton的配置
    this.animationDuration,//如果形状和阴影发生变化的时候的持续时间
    this.minWidth,//按钮的最小宽度
    this.height,//按钮的高度
    this.enableFeedback = true,//是否提供辅助功能
    this.child,//可以在按钮中的子控件，我们可以添加文字或者图片作为按钮的样式
  })
```

textTheme
根据主题里的颜色设置文字颜色，默认值是 `ButtonTextTheme.normal` 。
`ButtonTextTheme.normal` 是黑色或者白色，根据 `ThemeData.brightness` 的设置不同有变化。
`ButtonTextTheme.accent` 文字的颜色同 `ThemeData.accentColor` 的设置一样。
B`uttonTextTheme.primary` 文字的颜色同 `ThemeData.primaryColor` 的设置一样。
可以看到上面的按钮设置中虽然属性众多但是都是一些比较好理解的一些简单属性，所以不再多加赘述了。只需要根据需要进行设定就可以了。

`clipBehavior`

Clip.none 不进行设置
Clip.hardEdge 裁剪速度快，但容易失真，会有锯齿。
Clip.antiAliasWithSaveLayer 裁剪后具有抗锯齿特性并分配屏幕缓冲区，所有后续操作在缓冲区进行，然后再进行裁剪和合成，很少使用。
Clip.antiAlias 裁剪边缘抗锯齿，裁剪更平滑，裁剪速度比 `Clip.antiAliasWithSaveLayer` 快，但是比 Clip.hardEdge 慢，常用于圆形和弧形之类的形状裁剪。

RaisedButton

因为它是`MaterialButton`的子类，只是样式略有区别，有了默认的样式，所以不再多加赘述它的区别。

FlatButton

扁平化的按钮，也是基于`MaterialButton`的子类，用法与`MaterialButton`一致。

DropdownButton

```
DropdownButton({
    Key key,
    @required this.items,//参照下面的详解
    this.selectedItemBuilder,//参照下面的详解
    this.value,//在items定义的value值中的一条，如果没有符合的则会报错
    this.hint,//提示文字
    this.disabledHint,//按钮被禁用时的提示文字
    @required this.onChanged,//参照下面的详解
    this.elevation = 8,//下方阴影的高度
    this.style,//设置显示的文字样式
    this.underline,//可以自己设置DropDownButton中的下划线样式
    this.icon,//右边的下拉箭头可以通过icon参数进行替换
    this.iconDisabledColor,//当button被禁用了的时候icon的颜色
    this.iconEnabledColor,//当button可以使用的时候icon的颜色
    this.iconSize = 24.0,//icon的大小
    this.isDense = false,//是否减少高度，如果是True则button里的文字和下划线之间没有间隙，False则有间隙
    this.isExpanded = false,//是否要从当前位置开始横向填充满控件，True则会一直填充到父容器右边
    this.itemHeight = kMinInteractiveDimension,//每一条目的高度，默认值为48
    this.focusColor,//获取焦点后的颜色
    this.focusNode,//焦点绑定
    this.autofocus = false,//自动获取焦点
  })
```

items

需要传入`DropdownMenuItem`的集合，我们先看下`DropdownMenuItem`能为我们做什么。

```
const DropdownMenuItem({
    Key key,
    this.value,//我们给这个条目定义的Value值
    @required Widget child,//每个条目里添加的控件
  })
```

如果我们要做一个简单的控件，则可以使用下面的方式来创建一个`DropdownMenuItem`的集合：

```
items: <String>['One', 'Two', 'Free', 'Four']
                    .map<DropdownMenuItem<String>>((String value) {
                  return DropdownMenuItem<String>(
                    value: value,
                    child: Text(value),
                  );
                }).toList(),
```

我们在以前的控件中也使用过map函数来帮忙我们快速的创建一组控件，这里也是通过map函数来帮我们来快速创建了每一个`DropdownMenuItem`。

`selectedItemBuilder`
这里返回的是选中后`DropdownButton`的显示样式，如果我们想要换一种样式我们可以自定义每一个条目选中后的按钮状态。例如：

```
selectedItemBuilder: (BuildContext context) {
                  return items.map<Widget>((String item) {
                    return Text(item);
                  }).toList();
                }
```

onChanged

在下拉框出现后进行选择完后的回调，会返回我们传入的value的值，可以根据这里返回的值来刷新`DropdownButton`显示的item内容。例如：

```
 onChanged: (String newValue) {
                  setState(() {
                    dropdownValue = newValue;
                  });
                },
```

IconButton

带有图片的button，基本用法与其他的Button差不多，我们只看下它不同的地方。

```
const IconButton({
    Key key,
    this.iconSize = 24.0,//图片按钮的尺寸
    @required this.icon,//传入进来的按钮图片
    @required this.onPressed,//点击事件的触发，如果没有传入该方法则认为该按钮不可用
    this.color,//只有在按钮可用的时候才会生效，也就是onPressed必须有值传入
    ...
  })
```

outlineButton

一个自带边框的按钮，其他的属性和普通的button差不多，我们只看它的差异部分。

```
const OutlineButton({
    Key key,
    this.borderSide,//参照下面的详细解释
    this.disabledBorderColor,//当按钮不可用的时候的颜色
    this.highlightedBorderColor,//按钮按下时的边框颜色
    ...
  })
```

borderside

用来设置边框的颜色以及粗细，我们需要传入BorderSide的对象

```
const BorderSide({
    this.color = const Color(0xFF000000),//边框的颜色
    this.width = 1.0,//边框的粗细程度
    this.style = BorderStyle.solid,//边框的样式
  })
```

二、TextField

用户输入框，属性比较多，我们来先看看他们都代表什么：

```
const TextField({
    Key key,
    this.controller,//参照下面的详解
    this.focusNode,//关联的光标下一个组件
    this.decoration = const InputDecoration(),//文本框的装饰器，默认提供一个带有下划线的样式，我们可以修改颜色，添加icon或者其他
    TextInputType keyboardType,//参照下面的详解
    this.textInputAction,//参照下面的详解
    this.textCapitalization = TextCapitalization.none,//设置大小写键盘
    this.style,//设置文字的样式
    this.strutStyle,//设置一个统一的文本样式
    this.textAlign = TextAlign.start,//文字对齐方式
    this.textAlignVertical,//文字竖直方向的显示方式，靠顶部，底部，居中
    this.textDirection,//文字的显示方向
    this.readOnly = false,//是否可以进行编辑，True就是只读文字，不可更改
    ToolbarOptions toolbarOptions, //参照下面的详解
    this.showCursor,//是否显示编辑的光标
    this.autofocus = false,//自动获取焦点
    this.obscureText = false,//是否隐藏显示内容，密码格式可以设置为True
    this.autocorrect = true,//是否开启自动纠正功能
    this.enableSuggestions = true,//是否开启输入推荐
    this.maxLines = 1,//最大输入行数
    this.minLines,//最小输入行数
    this.expands = false,//不理解意思
    this.maxLength,//允许输入的最长字符长度
    this.maxLengthEnforced = true,//是否允许可以输入的字符数超过最大长度
    this.onChanged,//文字变化的时候的回调
    this.onEditingComplete,//提交内容时候的回调，一般是点击回车按钮的时候回调
    this.onSubmitted,//在提交时候回调，不可与onEditingComplete同时使用,区别于onEditingComplete的是回调带有参数。
    this.inputFormatters,//输入的内容的格式验证，可以传入多组格式
    this.enabled,//是否启用该输入框
    this.cursorWidth = 2.0,//闪烁的光标的宽度
    this.cursorRadius,//光标的圆角弧度
    this.cursorColor,//输入光标的颜色
    this.keyboardAppearance,//键盘的亮度
    this.scrollPadding = const EdgeInsets.all(20.0),//滚动时的内边距,实际测试没有发现有用
    this.dragStartBehavior = DragStartBehavior.start,//拖拽行为的开始点,实际测试没有发现有用
    this.enableInteractiveSelection = true,//长按的时候是否显示复制，剪切，粘贴这些功能
    this.onTap,//在点击的时候的回调
    this.buildCounter,//如果我们设置了maxlength，则可以通过它来在右下角显示字数
    this.scrollController,//设置内容滚动的监听
    this.scrollPhysics,//
  })
```

controller

文本框内容变化需要回调，通过Controller来抓取监听，Controller里添加addListener来获取内容的变化，例如：

```
controller.addListener((){
       print("text is ${controller.text}");//获取文本输入框内容
    });
```

keyboardType

选择弹出的输入框的类型。
text 普通的文本输入类型
multiline 跟普通键盘一样，只是允许输入多行，也就是左下角的确认按钮会变成回车按钮。
number 数字键盘
phone 拨号键盘，带有"*"和 "#".
datetime 在安卓手机上显示数字键盘加上":"和"-"，ios上显示默认键盘
emailAddress 已经加上"@" 和"."的普通键盘
url 已经加上"/" 和"."的普通键盘
visiblePassword 普通键盘

textInputAction
这里控制最后回车按钮的触发事件，比如回车和发送按钮。
none 默认状态，回车符号
unspecified 安卓显示 回车符号
done 完成按钮
go 显示GO按钮
search 显示搜索按钮
send 显示发送按钮
next 显示下一步按钮
previous iOS显示一个左转箭头，安卓显示回车符号
continueAction android 不支持
join 显示join按钮
route 显示Route按钮
emergencyCall android 不支持
newline 回车按钮

toolbarOptions
设置长按文字后出现的工具选择框都有哪些功能，需要传入ToolbarOptions实体。

```
const ToolbarOptions({
    this.copy = false,//是否显示复制按钮
    this.cut = false,//是否显示剪切按钮，如果设置不可编辑则不显示
    this.paste = false,//是否显示粘贴按钮，如果设置不可编辑则不显示
    this.selectAll = false,//是否显示选择全部按钮
  })
```

三、Card

卡片控件，可以设置显示的阴影效果，只能传入单一控件。我们先看下他需要的参数：

```
const Card({
    Key key,
    this.color, //卡片的颜色
    this.elevation, //设置阴影的大小
    this.shape, //参照下面的详细说明
    this.borderOnForeground = true, //从文档上来看是shape的边是否要遮挡child，默认是遮挡的。实际没有看出来效果
    this.margin, //设置外边距
    this.clipBehavior, //参照Button 的clipBehavior属性说明
    this.child, //需要填充到里面的子控件
    this.semanticContainer = true,//判断是否是语义容器，作为系统的辅助功能的时候使用。
  })
```

shape

控制card的形状，如果我们想要一个带圆角的Card需要下面这样设置：

```
shape: const RoundedRectangleBorder(borderRadius:
BorderRadius.all(Radius.circular(14.0))),
```

今天我们又学习了这几个组件，主要是Button的部分比较多，按钮的样式比较复杂，还需要多加练习查看运行后的结果 .

14.Flutter之基本组件4

前面已经学习了很多容器和组件，基本上可以为我们搭建一个简单的APP了，这篇作为基础组件的最后一篇，其他的组件可以等到我们使用到的时候再去进行查漏就可以，就不再一一分析了。
这篇我们一起来学习ToggleButtons，Checkbox，CheckboxListTile，Switch，Slider，RangeSlide。

一、ToggleButtons

一组水平摆放的切换按钮，可以设置多个状态的切换选择。还是跟以前一样先看下它的构造：

```
const ToggleButtons({
    Key key,
    @required this.children,//每个ToggleButton的布局样式，我们可以传入Icon或者Text来作为子Button
    @required this.isSelected,//需要为每一个Button设置，必须数量与给出的按钮个数一致，我们在每次点击触发后再去刷新State来更改状态
    this.onPressed,//参照下面详解
    this.textStyle,//如果我们设置的是Text则可以在这里设置统一的样式
    this.constraints,//我们可以控制这一组按钮里的每个按钮的大小，需要传入BoxConstraints对象，可以设置宽高
    this.color,//设置里面的按钮默认颜色
    this.selectedColor,//选中状态的颜色
    this.disabledColor,//不可用的颜色
    this.fillColor,//选中状态时的按钮底色
    this.focusColor,//获取焦点的颜色
    this.highlightColor,//按下的时候高亮的颜色
    this.hoverColor,//悬浮鼠标的颜色
    this.splashColor,//点击时的过度色
    this.focusNodes,//焦点的关联点
    this.renderBorder = true,//是否显示边框，False则不显示边框
    this.borderColor,//默认边框的颜色
    this.selectedBorderColor,//被选中时的边框颜色
    this.disabledBorderColor,//被禁用时的边框颜色
```

```
    this.borderRadius,//边框的圆角度数
    this.borderWidth,//边框线条的粗细
  })
```

onpressed

点击每个button的回调，回调时会返回点击的按钮的位置，如果没有给定onPress的值那么默认的会设置ToggleButton的状态为disable，我们可以根据位置来更换button的状态，举个例子：

```
 onPressed: (position){
                setState(() {
                  selected[position] = !selected[position];
                });
              },
```

二、Checkbox

我们通常使用的复选框按钮，我们可以一般用来作为设置的状态按钮。

```
const Checkbox({
    Key key,
    @required this.value, //是否选中的状态
    this.tristate = false, //当value的值为空时是否要显示一个破折号，也就是一个横杠。True表示显示。
    @required this.onChanged,//在点击按钮的时候状态变化的监听，回调时会返回当时的状态
    this.activeColor,//选中时按钮的颜色
    this.checkColor,//选中时里面对勾的颜色
    this.focusColor,//获取光标时的颜色
    this.hoverColor,//鼠标悬停时的颜色
    this.materialTapTargetSize,//设置大小，是否是固定大小还是根据主题的大小
    this.focusNode,//焦点的关联点
    this.autofocus = false,//是否允许自动获取焦点
  })
```

看着还是比较简单的，我们可以根据需要进行参数的配置

三、CheckboxListTile

既然提到了CheckBox顺便看一下CheckboxListTile，这是一个已经帮我们封装好了的一条选择框，包含了图片和文字描述的位置还有选择框，省去了我们自己构造这样条目的麻烦，先了解下它的构造：

```
 const CheckboxListTile({
    Key key,
    @required this.value,//是否选中的状态
    @required this.onChanged,//按钮点击的变化回调
    this.activeColor,//选中时按钮的颜色
    this.checkColor,//选中时里面对勾的颜色
    this.title,//添加条目的Title文字描述
    this.subtitle,//添加的子标题描述
    this.isThreeLine = false,//是否允许标题的多行显示，true允许多行，false在子标题为空时标题只能显示一行，不为空时可以显示两行，但是实际测试没有发现效果
    this.dense,//是否密集的显示，设置为True后title的字体会变小
    this.secondary,//在最左边显示的控件，一般用来放置一个图片
    this.selected = false,//如果设置为True则文字和图片会显示被按下的颜色
    this.controlAffinity = ListTileControlAffinity.platform,//图片和控制按钮的位置，leading则会让图片和控制按钮调换位置。
  })
```

四、Switch

我们常见的开关按钮，用来做是否开关某个功能。先来看下它的构造方法：

```
const Switch({
    Key key,
    @required this.value,//设置的是否开启的状态
    @required this.onChanged,//在点击按钮的时候状态变化的监听，回调时会返回当时的状态
    this.activeColor,//打开时上面的圆扭颜色
    this.activeTrackColor,//打开时下面的横条颜色
    this.inactiveThumbColor,//在关闭时圆钮的颜色
    this.inactiveTrackColor,//在关闭时下面的横条颜色
    this.activeThumbImage,//打开时的按钮可以替换为图片
    this.inactiveThumbImage,//关闭状态的按钮可以用图片进行替换
    this.materialTapTargetSize,//设置大小，是否是固定大小还是根据主题的大小
    this.dragStartBehavior = DragStartBehavior.start,//开始识别手势的点
    this.focusColor,//获取光标时的颜色
    this.hoverColor,//鼠标悬停时的颜色
    this.focusNode,//焦点的关联点
    this.autofocus = false,//是否允许自动获取焦点
  })
```

看着按钮也是比较简单的，不再多加赘述。

五、Slider

滑块组件，我们可以拖拽来显示特定的值。

```
const Slider({
    Key key,
    @required this.value,//当前选定的值
    @required this.onChanged,//拖动滑块的变化事件，会将拖动变化的值回传
    this.onChangeStart,//在开始拖动时的回调，会将开始时点的值回传
    this.onChangeEnd,//在停止拖动时的回调，将停止时的点的值回传
    this.min = 0.0,//slide的开始点
    this.max = 1.0,//slide的结束点
    this.divisions,//将开始到结束的点划分为多少段
    this.label,//设置之后拖动圆钮上在拖动时会有小气泡出现，显示label定义的文字
    this.activeColor,//拖动圆钮和已经经过的部分会变成该颜色
    this.inactiveColor,//没有拖到到的部分会显示该颜色
    this.semanticFormatterCallback,//根据slide的值创建一个自定义语义的值,还是留给了辅助
功能使用的
  })
```

六、Rangeslide

跟Slide很相似，只是由由原来的选择的是某个值改为了某个区间段的值。我们只看下差异的部分：

```
 RangeSlider({
    Key key,
    @required this.values,//需要给定一段区间的设置。例如values:
RangeValues(startIndex,endIndex),
    @required this.onChanged,//由原来的返回单值变成返回RangeValues的对象，RangeValues
可以返回开始和结束的点的值
    this.onChangeStart,//由原来的返回单值变成返回RangeValues的对象，RangeValues可以返回
开始和结束的点的值
    this.onChangeEnd,//由原来的返回单值变成返回RangeValues的对象，RangeValues可以返回开
始和结束的点的值
    ...
  })
```

这篇文章介绍的几个组件都是比较简单，我们可以自己多加练习，看看实际的效果。

接下来来学习Flutter的列表

15.Flutter之列表

基本上的组件学的差不多了，我们开始一起来学习日常用的最多的列表功能。列表我们一般常用的有ScrollView，ListView，和GridView，在Flutter中也是有一一对应的组件，在Flutter里ScrollView是抽象类无法直接被实例化，我们可以使用它的子类，如NestedScrollView，BoxScrollView，在Flutter比较特殊ListView,GridView也是ScrollView的子类。我们依次看下。

Nestedscrollview

这里的NestedScrollView跟Android中的意义不一样，在Android中是直接一个可以滑动的列表，我们将子View放到ScrollView下的ViewGroup中就可以实现一个页面的滚动，而这里的NestedScrollView是做了一个包含TabBar的一个封装，我们可以利用它轻松的实现向上滑动的时候顶部的折叠

```
const NestedScrollView({
    Key key,
    this.controller,//滚动操作的控制器，给我们一个默认值_NestedScrollController，如果需
要修改可以模仿来做更改
    this.scrollDirection = Axis.vertical, //滚动的方向，Axis.vertical竖向滚动，
horizontal横向滚动
    this.reverse = false,//是否页面做翻转，如果是True则bar在下面，滚动的在上面
    this.physics,//当用户停止滑动，滑动到顶部时的动作
    @required this.headerSliverBuilder,//参照下面的详细说明
    @required this.body,//参照下面的详细说明
    this.dragStartBehavior = DragStartBehavior.start,//拖拽开始的状态
  })
```

headerSliverBuilder

向上滑动的时候的折叠区域，可以传入一组的控件 List，如果我们在向上滑动的时候可以折叠的话我们需要返回一个`SliverOverlapAbsorber`，`SliverOverlapAbsorber`稍后我们做详细解释。

bady

传入一个可以滑动的布局，比如我们可以传入一个`CustomScrollView`，我们则可以通过`CustomScrollView`来实现页面的滑动，也可以传入`SliverList`,`SliverGrid`让页面滚动起来。
先来看下上面提到的`SliverOverlapAbsorber`。

SliveroverlapAbsorber

使用`SliverOverlapAbsorber`包裹`SliverAppBar`让在滑动的时候可以进行通知头部，控制AppBar的收放。还是先看下构造函数：

```
const SliverOverlapAbsorber({
    Key key,
    @required this.handle, //参照下面的详细说明
    Widget child,//需要控制的组件，一般使用SliverAppBar
  })
```

handle

作为一个必选参数来控制堆叠时的控件高度，我们一般使用下面`NestedScrollView.sliverOverlapAbsorberHandleFor`来创建一个默认的操作。

```
handle:NestedScrollView.sliverOverlapAbsorberHandleFor(context)
```

SliverAppBar

跟AppBar有很多共通的参数，我们可以拿出来它与AppBar的不同之处来作解释：

```
const SliverAppBar({
    Key key,
    this.flexibleSpace,//我们需要显示的标题栏里的内容，可以根据需要进行自己的设置，比如一个Text
    this.forceElevated = false,//是否保留SliverAppBar下面的阴影，True不管是滚动还是停止都有阴影，False则都不显示
    this.expandedHeight,//标题栏和文字说明直接的间距
    this.floating = false,//和snap配合使用
    this.pinned = false,//顶部的Title栏是否是固定的，False则不显示title栏，True则会滑动时保留title栏
    this.snap = false,//和floating配合使用，两个都是True则滑动到顶再下向滑动时首先是标题栏先滑下来，然后再是列表的向下滚动。
    this.stretch = false,//是否拉伸以充满滑动区域，没有试出来它的效果
    this.stretchTriggerOffset = 100.0,//设定调用onStretchTrigger的滑动距离
    this.onStretchTrigger,//设定的回调
    ...
  })
```

设置还是比较简单，可以根据我们是否需要滚动，是否滚动的时候是否固定头部来进行设置。

Listview

作为一个线性排布所有子控件的容器，它其实比较像在Android中认识的ScrollView，但是我们可以通过itemBuilder来实现Android中的ListView的形式。先看下ListView的构造方法，先省略NestedScrollView里面共通的参数：

```
ListView({
    Key key,
    bool shrinkWrap = false,//是否根据子组件的总长度来设置ScrollView的长度，如果ScrollView的父容器是一个无边际的的容器则设置为True。
    EdgeInsetsGeometry padding,//设置ListView的内填充大小
    this.itemExtent,//如果不为空则会强制children的长度为itemExtent的值。
    bool addAutomaticKeepAlives = true,//是否将列表项放到AutomaticKeepAlive组件中，在懒加载的时候如果设置为True则滑出窗口视图的时候不会被回收，由KeepAliveNotification来保存状态
    bool addRepaintBoundaries = true,//是否将列表项放到RepaintBoundary组件中，设置为True在列表滚动的时候可以避免重绘，但是如果每个Item都很简单比较小的时候如果设置为True会更高效。
    bool addSemanticIndexes = true,//设置是否将列表项添加到用位置来表示的语义控件中，还是原来说到的辅助功能使用的
    double cacheExtent,//缓存的长度
    List<Widget> children = const <Widget>[],//参照下面的详细说明
    int semanticChildCount,//语义表达的列表项的数量，还是辅助功能使用。
    ...
  })
```

children

这里用来存放滑动的内容，我们可以创建一组Text为了测试滑动比如：

```
ListView(
          children: <Widget>[
            Text("Hello"),
            Text("Hello"),
            Text("Hello"),
            Text("Hello"),
          ],
        )
```

如果要是显示一组内容相同的列表我们可以使用ListView的建造者模式的方法`ListView.builder()`，其他的参数同ListView只是多了一个

```
 @required IndexedWidgetBuilder itemBuilder,
```

我们可以通过它来创建一组数据的列表比如：

```
ListView.builder(
          itemBuilder: (BuildContext context, int index) {
            return Text("Hello" + index.toString());
          },
          padding: EdgeInsets.all(30),
        )
```

常用的还有`ListView.separated()`只是比`ListView.builder()`多了一个分割线的设置，不再赘述，可以自己试一下。

GridView

跟ListView的参数有很大的相似度，它的功能就是将竖向列表在横向也可以按照指定的块数显示。我们看下它的构造方法，跟ListView类似的不再说明：

```
GridView({
    Key key,
    @required this.gridDelegate,//参照下面的详细说明
   ...
  })
```

gridDelegate

我们看到需要传入一个`SliverGridDelegate`，但是该类是一个抽象类，我们需要传入一个实现类，找到了`SliverGridDelegateWithFixedCrossAxisCount`，我们将其传入，来看下它有哪些参数 .

```
const SliverGridDelegateWithFixedCrossAxisCount({
    @required this.crossAxisCount,//要划分的块数
    this.mainAxisSpacing = 0.0,//跟主要滑动方向一致的每块间隔大小
    this.crossAxisSpacing = 0.0,//另外一个方向上每块间隔大小
    this.childAspectRatio = 1.0,//滑动方向和另一方向块的大小比例
  })
```

一个简单的使用：

```
GridView(
          gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
              crossAxisCount:3,
              mainAxisSpacing:10,
            crossAxisSpacing: 10,
          ),
          children: <Widget>[
            Text("Hello",style: TextStyle(backgroundColor: Colors.green),),
            Text("Hello"),
            Text("Hello"),
            Text("Hello"),
          ],
        ),
```

在GridView中也有`builder()`和`separated()`方法，跟ListView的方法一致。

SliverList

可以理解为我们可以将多个布局按照一个顺序来进行的排列，每个都是单独的个体。它的构造比较简单，我们看下：

```
const SliverList({
    Key key,
    @required SliverChildDelegate delegate,//参照下面的详细说明
  })
```

delegate

就只需要传入一个代理，`SliverChildDelegate`是一个抽象类，我们还需要找到它的实现`SliverChildListDelegate`，来看下`SliverChildListDelegate`的构造：

```
SliverChildListDelegate(
    this.children, {
    this.addAutomaticKeepAlives = true,
    this.addRepaintBoundaries = true,
    this.addSemanticIndexes = true,
    this.semanticIndexCallback = _kDefaultSemanticIndexCallback,
    this.semanticIndexOffset = 0,
  })
```

跟List基本上一样的参数，只是children是一组组件，我们需要一个个的实现，`SliverChildDelegate`下面还有一个实现是`SliverChildBuilderDelegate`，这个可以帮助我们快速的创建一组列表与ListView相同的略过：

```
const SliverChildBuilderDelegate(
    this.builder, //这里创建了一个构造器，下面详细说明
    {this.findChildIndexCallback,//构造一个返回当前Item的Index的方法
    this.childCount,//想要显示的Item的数量
    ...
  })
```

builder

可以看到它的定义是比较有意思的，是定义了一个方法：

```
typedef IndexedWidgetBuilder = Widget Function(BuildContext context, int index);
```

我们需要传一个函数进去，可以简单的试一下：

```
delegate: SliverChildBuilderDelegate(
                        (BuildContext context, int index) {
                      return Container(
                        alignment: Alignment.center,
                        color: Colors.red,
                        child: Text("position is $index",style:
TextStyle(fontSize: 14,color: Colors.black),),
                      );
                    }
                )
```

这样就构造了一个我想要显示的每一个条目的样式，然后控制数量和缓存机制就可以了。

SliverGrid

跟GridView是一样的显示形式，只是也是进行了分块，我们还是看下构造方法：

```
  const SliverGrid({
    Key key,
    @required SliverChildDelegate delegate,
    @required this.gridDelegate,
  })
```

只是比上面的SliverList多了一个gridDelegate，我们只看下gridDelegate，看到它需要传入`SliverGridDelegate`，但是也是一个抽象类，还是看下它的实现`SliverGridDelegateWithFixedCrossAxisCount`，嗯，是的，跟GridView中的是一样的，我们可以直接去看GridView中的说明。这样进行设置后一个可以分割的Grid就完成了。

CustomScrollview

跟ScrollView也不是一样的，它比较像是一个粘合剂的作用，如果我们的列表显示多个列表的组合那么可以放置到CustomScrollView里面，来保证它们的滑动事件的统一，里面也可以放置像ListView和GridView这样子的组件，但是需要注意如果我们需要让他们联合滚动需要不能使用ListView和GridView，我们需要一种已经帮我们做好分割的SliverList和SliverGrid，还是先看下他的构造方法，有部分同NestedScrollView直接去掉了：

```
const CustomScrollView({
    Key key,
    bool primary,//如果为True则会让它的Controller创建一个PrimaryScrollController的实体，不能再自定义Controller。
    bool shrinkWrap = false,//是否根据子组件的总长度来设置ScrollView的长度，如果ScrollView的父容器是一个无边际的的容器则设置为True。
    Key center,
    this.slivers = const <Widget>[],//参照下面的详细说明
    ...
  })
```

slivers

这里我们就要放入可分裂的组件了，我们可以将上面的SliverList或者SliverGrid放入其中，可以混合放入，这样就成了一个混合列表。
滚动页面的内容比较复杂，不是很好理解，还是建议多去试一试，这样可以加深我们的记忆。

接下来学习Flutter中的弹框

16.Flutter之弹框

在Flutter中我们也有着丰富的弹出框控件，Flutter给我们准备好了SimpleDialog，AlertDialog，AboutDialog，CupertinoAlertDialog这些弹出框，我们可以依次来认识它们。
在认识它们之前还需要先学习一下如果让这些组件显示在屏幕上，这些组件创建完之后是否直接显示的，我们还需要调用一个show方法，比如`showDialog()`；`showAboutDialog()`；`showCupertinoDialog()`;这样的显示方法。

Show方法

show方法是根据想要显示的对话框的样式不同进行了封装，showDialog可以放下我们想要的所有dialog，只是针对不同的dialog会有时比较繁琐，我们先看下showDialog方法中需要传入哪些参数：

```
Future<T> showDialog<T>({
  @required BuildContext context, //传入上下文
  bool barrierDismissible = true,  //点击遮罩层是否对话框消失
  @Deprecated(
    'Instead of using the "child" argument, return the child from a closure '
    'provided to the "builder" argument. This will ensure that the BuildContext
'
    'is appropriate for widgets built in the dialog. '
    'This feature was deprecated after v0.2.3.'
  )
  Widget child,//过时了，不再使用
  WidgetBuilder builder, //需要构建的对话框
  bool useRootNavigator = true, //是否将dialog放到根的Navigator中，默认是添加的
})
```

如果我们想要一个AboutDialog那么我们可以直接使用`showAboutDialog()`方法来直接创建一个AboutDialog。AboutDialog我们一般用来显示应用的基本信息，用的比较少，我们看下`showAboutDialog()`:

```
void showAboutDialog({
  @required BuildContext context,//传入上下文
  String applicationName,//传入应用的名字
  String applicationVersion,//应用的版本号
  Widget applicationIcon,//应用的图标
  String applicationLegalese,//添加一些应用的文字说明
  List<Widget> children,//添加一些自定义的子控件，根据大家的需要自行补充就行
  bool useRootNavigator = true,//是否将dialog放到根的Navigator中，默认是添加的
})
```

如果我们想要创建一个iOS风格的dialog，那么我们可以使用`showCupertinoDialog();`来直接创建，可以看下`showCupertinoDialog()`:

```
Future<T> showCupertinoDialog<T>({
  @required BuildContext context, //传入上下文
  @required WidgetBuilder builder,//需要构建的对话框
  bool useRootNavigator = true,//是否将dialog放到根的Navigator中，默认是添加的
})
```

可以看到跟showDialog基本上是一致的，我们只需要在builder里传入CupertinoDialog或者CupertinoAlertDialog就可以了。
如果我们想要给弹出框的时候增加一个动画我们可以使用showGeneralDialog，我们可以根据自己需要制定旋转缩放等动画，先看下它的构造：

```
Future<T> showGeneralDialog<T>({
  @required BuildContext context,//必填
  @required RoutePageBuilder pageBuilder,//必填，如果没有要传入的builder需要传入一个空的，例如 (context, anim1, anim2) {},
  bool barrierDismissible,//控制弹出框是否显示的，如果为True那么barrierLabel必须设置，如果设置为False则弹出框不会显示
  String barrierLabel,//在barrierDismissible为True的时候必须有值，否则弹出框不会显示
  Color barrierColor,//动画的背景
  Duration transitionDuration,//dialog显示和消失的时间，对应这里就是显示的动画时间
  RouteTransitionsBuilder transitionBuilder,//我们可以组件自己的显示动画
  bool useRootNavigator = true,//是否将dialog放到根的Navigator中，默认是添加的
})
```

我们对上面的show方法举个例子，一个旋转的对话框：

```
 showGeneralDialog(
              context: context,
              pageBuilder: (context, anim1, anim2) {},
              barrierColor: Colors.grey.withOpacity(.4),
              barrierDismissible: true,
              barrierLabel:"旋转框" ,
              transitionDuration: Duration(milliseconds: 400),
              transitionBuilder: (context, anim1, anim2, child) {
                return Transform.rotate(
                  angle: anim1.value * 360,
                  child: AlertDialog(
                    shape: RoundedRectangleBorder(
                        borderRadius: BorderRadius.all(Radius.circular(20))),
                    title: Text("Dialog"),
                    content: Text("Hello world"),
                  ),
                );
              },
            );
```

SimpleDialog

一般我们使用SimpleDialog用于多项条目的选择框，我们可以先看下它的构造方法：

```
 const SimpleDialog({
    Key key,
    this.title,//弹出框的标题栏
    this.titlePadding = const EdgeInsets.fromLTRB(24.0, 24.0, 24.0, 0.0),//标题的上下左右间距
    this.children,//这里可以添加选择条目,这里添加的不是普通的组件了，是需要传入SimpleDialogOption
    this.contentPadding = const EdgeInsets.fromLTRB(0.0, 12.0, 0.0, 16.0),//内容与边框之间的间距
    this.backgroundColor,//弹出框的背景颜色
    this.elevation,//弹出框的阴影大小
    this.semanticLabel,//辅助功能中的语音提示
    this.shape,//弹出框的形状。比如设置圆角边框
  })
```

SimpleDialogoption

作为SimpleDialog的children项，我们看下它可以为我们做些什么，先看下它的构造方法：

```
const SimpleDialogOption({
    Key key,
    this.onPressed,//点击的响应事件
    this.child,//添加的子组件
  })
```

这里我们就可以把每个条目的点击事件和每个条目要显示的内容来做显示了。例如：

```
SimpleDialogOption(
       child: Text("条目一"),
       onPressed: (){
            Navigator.of(context).pop();
        },)
```

这里我们就显示了一个条目并且点击后整个弹出框消失。

AlertDialog

用于展示一些提示信息的dialog，我们可以为dialog添加多个按钮，我们先看下它的构造，和SimpleDialog里面相同的就省略了：

```
const AlertDialog({
    Key key,
    this.titleTextStyle,//给标题增加字体的样式，样式的设置同Text的样式设置方法
    this.content,//我们用来展示要显示的内容，需要注意的是如果内容过长我们需要使用可滚动的布局列表来展示，否则无法全部显示
    this.contentTextStyle,//给展示的内容添加文字的全局样式
    this.actions,//这里我们可以添加确定和取消等按钮，如果能横向全部展示则会显示全部，如果不能全部显示则全部按钮竖向显示
    ...
  })
```

AboutDialog

一般用来显示一些应用的基本信息，会默认带着两个按钮，一个ViewLicences跳转到应用的Licences的页面,一个Close用来关闭dialog，我们来看下它的构造：

```
const AboutDialog({
    Key key,
    this.applicationName,//应用名称
    this.applicationVersion,//应用的版本
    this.applicationIcon,//应用的icon
    this.applicationLegalese,//放一些应用的说明
    this.children,//可以添加自定义控件
  })
```

CupertinoAlertDialog

用来显示iOS风格的dialog，基本用法与AlertDialog一致，在使用`CupertinoAlertDialog`时我们需要修改show方法为`showCupertinoDialog`，我们主要看下差异的部分，先看下它的构造：

```
 const CupertinoAlertDialog({
    Key key,
    this.scrollController,//给内容的滚动添加一个控制器，我们可以参照前面
NestedScrollView所介绍的Controller
    this.actionScrollController,//给action的滚动添加一个控制器，action在传入多个按钮的
时候会呈现竖直排列
    this.insetAnimationDuration = const Duration(milliseconds: 100),//显示弹框的动
画时间
    this.insetAnimationCurve = Curves.decelerate,//进入动画的曲线，也就是进入的路径
    ...
  })
```

基本上我们常用的弹出框就介绍完了，我们可以结合项目选择合适的对话框来显示，如果需要自定义弹框的可以根据上面的介绍来创建自定义对话框。

接下来学习Flutter网络请求

17.Flutter之网络请求

网络请求在一个APP的分量还是很重的，我们需要熟练的掌握Http请求部分，在Flutter中我们有三种方式，一种是使用Dart给我们提供的HttpClient，第二种是使用第三方的Http请求库，第三种是使用Flutter给我们准备好的Dio。我们主要是看后面两种，第一种比较繁琐一般也不会使用，如果想要了解的可以自行搜索。

HTTP库

这是一个基础的网络请求库，我们在使用之前需要先引入这个类库，在`pubspec.yaml`中添加：

```
dependencies:
  http: ^0.12.1
```

添加完成后在右上角会出现一个pub get的按钮，点击后可以帮助我们下载这个类库，然后在我们需要的地方加载该类：

```
import 'package:http/http.dart';
```

这样我们的准备工作就算是完成了，接下来我们开始试一下如果使用Http库进行网络请求了：

```
void doHttpRequest() async{//里面包含异步请求的时候需要加上asyn关键字
    var url = "http://www.baidu.com";
    var postResponse = await http.post(url,//使用Post请求，在需要等待的时候加上await
关键字
        headers: {"": "", "": ""},//添加请求头
        body: {"": "", "": "",});//添加请求的参数

    var response = await http.get(url);//使用get方法，直接请求该地址
    if(response.statusCode == 200){//如果返回的状态码是200则有正确的返回结果
      print("response body = ${response.body}");
    }else{//其他的异常状态码
      print("请求异常${response.statusCode}");
    }
  }
```

这是比较简单进行一次网络请求，我们也可以根据自己的需要进行一定的封装，这里不多介绍，我们主要是使用Flutter给我们准备好的Dio来进行网络请求。

DIO

套用一下dio中的介绍：
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、自定义适配器等
可以做的事情还是挺多的，我们一点点的进行学习吧。
首先我们需要将它引入到我们项目中：

```
dependencies:
  dio: ^3.0.9  //目前的最新版本是3.0.9
```

发送一个get请求，我们可以使用两种形式：

```
    var host = "https://wanandroid.com/wxarticle";
    var getResponse = await Dio().get("$host/list/408/1/json?id=404&page=1");//传统的我们使用的直接请求的方式
    var getResponse1 = await Dio().get("$host/list/408/1/json",
            queryParameters: {"id":"404","page":"1"});//给我们扩展出来了新的写法，这样可以更好的查看参数
```

我们看下返回信息Response里都给我们带来了什么：

```
 Response({
    this.data,//返回回来的数据
    this.headers,//响应头信息
    this.request,//请求时带的信息也返回回来了
    this.isRedirect,//是否是重定向，这个字段的是否可用取决于适配器是否支持(不是很理解这个字段)
    this.statusCode,//返回的响应码，成功请求返回200
    this.statusMessage,//返回的响应码的状态描述，如果是失败的时候有个简短的状态信息
    this.redirects,//会将每次重定向的地址以及请求状态放到这里，我们可以跟踪每次的重定向请求
    this.extra,//自定义字段，我们可以在发送请求的时候将想要在返回处理的时候需要处理的信息进行存放，在请求响应完成时可以拿到附加信息
  })
```

上面就是在完成一次请求后我们可以获取到的响应内容，我们看下通过post方式会是怎样的请求形式：

```
var postResponse = await Dio().post(host,data: {"name":"403"});
```

很简单，只是上面的get关键字换成了post，然后传入我们想要的参数就可以了。我们不仅可以通过get和post这两个函数进行请求，我们也可以使用Dio给我们准备的request方法，我们可以根据自己想要的灵活配置：

```
Future<Response<T>> request<T>(
    String path, {//请求的地址
    data,//post需要传入的参数
    Map<String, dynamic> queryParameters,//get方法的时候可以添加的参数
    CancelToken cancelToken,//可以添加网络取消，网络取消的监听都在这里了
    Options options,//网络的配置，参照下面的详解
    ProgressCallback onSendProgress,//发送网络请求的进度监听回调
    ProgressCallback onReceiveProgress,//接收数据返回的进度监听回调
  });
```

options

网络请求的配置，我们可以通过它来设置请求头和连接超时的设置，看下我们怎么配置一个网络请求的：

```
RequestOptions({
    String method,//网络请求的方法
    int sendTimeout,//发送的超时时间
    int receiveTimeout,//接受超时时间
    this.connectTimeout,//链接超时的时间
    this.data,//post请求数据时传递的参数
    this.path,//如果是我们用了http或者https开头则会忽略baseUrl的设置，如果没有，则拼接到baseUrl后面
    this.queryParameters,//使用get方法时候的参数拼接
    this.baseUrl,//请求的根网址
    this.onReceiveProgress,//在收到消息时的进度
    this.onSendProgress,//在发送消息时的进度
    this.cancelToken,//可以添加网络取消，网络取消的监听都在这里了
    Map<String, dynamic> extra,//额外的参数配置，可以在返回的数据中获取到
    Map<String, dynamic> headers,//请求头
    ResponseType responseType,//响应类型,可以是json，可以是数据流，可以是json
    String contentType,//设置内容的类型
    ValidateStatus validateStatus,//给设定一个响应码，如果是返回了True则表示请求成功
    bool receiveDataWhenStatusError = true,//是否在Http的响应码是失败的时候接收响应数据
    bool followRedirects = true,//是否可以重定向
    int maxRedirects,//最多可以重定向的次数
    RequestEncoder requestEncoder,//我们可以设置在发送请求时对请求进行编码，默认是utf-8的编码，可以根据需求进行修改
    ResponseDecoder responseDecoder,//当我们接收返回消息时对内容进行解码，默认是utf-8，可以根据需求进行修改
  })
```

这是在我们使用dio的request方法的时候传入的option，在get和post方法的时候我们也看到有一个option，这两个传入的option还有一些小区别，我们看下在get和post的时候我们传入的option：

```
BaseOptions({
    String method,//网络请求的方法
    this.connectTimeout,//链接超时的时间
    int receiveTimeout,//接受超时时间
    int sendTimeout,//发送的超时时间
    this.baseUrl,//设置一个根地址
    this.queryParameters,//在get方法的时候作为拼接的参数
    Map<String, dynamic> extra,//在请求时的携带的额外信息
    Map<String, dynamic> headers,//请求头
    ResponseType responseType = ResponseType.json,//响应类型,可以是json，可以是数据流
    String contentType,//设置内容的类型
    ValidateStatus validateStatus,//给设定一个响应码，如果是返回了True则表示请求成功
    bool receiveDataWhenStatusError = true,//是否在Http的响应码是失败的时候接收响应数据
    bool followRedirects = true,//是否可以重定向
    int maxRedirects = 5,//最多可以重定向的次数
    RequestEncoder requestEncoder,//我们可以设置在发送请求时对请求进行编码，默认是utf-8的编码，可以根据需求进行修改
    ResponseDecoder responseDecoder,//当我们接收返回消息时对内容进行解码，默认是utf-8，可以根据需求进行修改
  })
```

再看下我们如何来做消息的拦截:

```
 dio.interceptors.add(InterceptorsWrapper(
        onRequest:(options){//发送请求时的过滤
            if(options.extra["data"] == 1)
                options.cancelToken.cancel("");//我们可以根据自己的设定过滤来取消请求
        },
        onResponse:(response){//在接收到数据响应的时候过滤
        },
        onError:(error){//在接收到错误信息的时候的拦截
        }
    ));
```

在网络请求过程中我们有时候还遇到过链式请求，需要在一次请求完成后进行下一次请求，我们看下如何实现这样的一个链式请求：

```
await Future.wait([dio.get(""),dio.post("")],);//我们可以直接写入多个网络请求，Future可以在一次请求完成后进行下一次的请求
```

使用Dio我们可以进行文件的上传下载，使用也是比较简单：

```
//对文件进行下载
 dio.download("下载地址", "保存在本地的地址",onReceiveProgress: (count,total){
        print("已经完成的下载量${count},需要下载的文件大小${total}");
    });
//对文件进行上传
 var file = MultipartFile.fromFile("本地文件路径",filename: "文件名");//先获取到File的对象
 var formData = FormData.fromMap({
      "file1":file,//将File对象传入到待上传的map中
    });
 dio.post(host,data:formData);//开始文件的上传
```

Dio的基本使用就差不多了，我们可以对它进行封装，这样每次在进行请求不用每个请求单独去做设置了，网上有很多大神给我们提供的封装案例，在这里不多描述，以后可以做一个通用的封装给大家。

接下来学习FLutter数据持久化

18.Flutter之数据持久化

在移动端存储数据我们有多种方法有sharePreferences，SQLite，本地文件，网络存储这几种方法，后面的两种在前面的文章中有部分介绍，这里不多做说明，这篇主要是跟大家一起来学习sharePreferences，SQLite。

sharePreference

这个在原来安卓的开发中经常用到，来看下在Flutter中是否还是我们原来熟悉的sharePreference么。首先介绍一下sharePreference的使用，由于是第三方库我们先将它引入到我们项目中：

```
dependencies:
  shared_preferences: ^0.5.7
```

引入完成后我们可以在使用时import该类库

```
import 'package:shared_preferences/shared_preferences.dart';
```

引入完成我们就可以进行使用了，先从常规的增删改查来看下：
增：

```
//在实例化的时候需要读取本地已经持久化的数据来读到内存中，有可能会耗时比较多，这里它采用了异步的方式，所以需要加await关键字，在引用它的方法声明上添加async关键字
SharedPreferences sharePerence = await SharedPreferences.getInstance();
sharePerence.setString("key1", "value1");
//这里存储了一个字符串，还可以存储其他类型的数据：
sharePerence.setBool(key, value);
sharePerence.setDouble(key, value);
sharePerence.setInt(key, value);
sharePerence.setStringList(key, value);
```

删：

```
sharePerence.remove(key);//删除比较简单
```

改：
改的方法同增加的方式，如果发现了重复的key则会对原来的值进行覆盖。
查：

```
sharePerence.containsKey(key);//是否包含指定key的值
```

它是怎么在本地进行存储的呢？它会生成一个xml放置到本地的沙盒中，打开里面看到存储的数据结构是：

```
<map>
    <string name="flutter.key1">value1</string>
</map>
```

也就是说我们所有的shareperfence都是存储在这一个xml中的，联想到xml的读取和写入方式，在存储到大量的数据后读取速度就会有比较大的影响，这里不建议存放过多的数据。

SQLite

存储大量数据的时候它就比较合适了，我们可以看下它是如何帮我们存储大量的数据的。
我们先引入SQLite的第三方库：

```
dependencies:
  sqflite: ^1.3.0
```

在需要使用的地方import该类库：

```
import 'package:sqflite/sqflite.dart';
```

完成引用后就可以开始使用了，先看下如何操作一个数据库，直接使用openDatabase方法就可以直接打开一个数据库，一起看下如何创建和升级表：

```
Future<Database> openDatabase(String path,//需要打开的数据库名称
    {int version,//最新的数据库版本
    OnDatabaseConfigureFn onConfigure,//参照下面的详解
    OnDatabaseCreateFn onCreate,//参照下面的详解
    OnDatabaseVersionChangeFn onUpgrade,//参照下面的详解
    OnDatabaseVersionChangeFn onDowngrade,//参照下面的详解
    OnDatabaseOpenFn onOpen,//参照下面的详解
    bool readOnly = false,//是否是只读模式
    bool singleInstance = true})//是否数据库是单例的，如果是单例的我们传入地址是一个的时候则会返回刚刚实例的数据库
```

onConfigure

在回调onCreate/onUpdate/onOpen之前会调用该方法，我们可以在这里做一些初始化的工作。onConfigure的类型为`OnDatabaseConfigureFn`，我们看下`OnDatabaseConfigureFn`是什么：

```
typedef OnDatabaseConfigureFn = FutureOr<void> Function(Database db);//这里我们需要传入一个带有Database参数的函数来处理初始需要做的事情
```

onCreate

在完成了onConfigure的回调后开始调用OnCreate方法，在这里我们可以创建表，还是看下`OnDatabaseCreateFn`是什么：

```
typedef OnDatabaseCreateFn = FutureOr<void> Function(Database db, int version);//这里我们需要一个传入一个带有DataBase和version的函数来处理初始化，可以根据version来创建表
```

我们先在这步进行一个数据库的创建为了以后我们好看接下里的实例：

```
onCreate: (dataBase, version) {
      dataBase.execute("create table my_table (id integer primary key,username text,password text)");//创建一个my_table的表
    }
```

onUpgrade

在版本号我们更新了以后会调用onUpgrade的方法，如果发生了表内字段的变更我们可以在这里根据版本号进行判断，针对不同的版本设定字段更新语句。看下OnDatabaseVersionChangeFn是什么：

```
typedef OnDatabaseVersionChangeFn = FutureOr<void> Function(
    Database db, int oldVersion, int newVersion);//这里我们需要一个传入一个带有DataBase、oldversion和newVersion的函数来处理初始化，可以根据oldVersion，newVersion来修改表结构
```

onDowngrade

如果出现newVersion低于了oldVersion的版本会调用onDowngrade的方法，调用的时候可以根据降级的版本进行数据表的恢复。传入的也是OnDatabaseVersionChangeFn，可以参照上面的。
打开了一个数据库之后我们就可以对数据库进行操作了，我们还是从常规的增删改查来看下：
增：

```
dataBase.execute(
          "insert into my_table(username,password) values(?,?);",
["flutter","pwd1"]);//增加一条数据
```

删：

```
int deleteCount = await dataBase.rawDelete("delete from my_table where
username=?",["flutter"]);//返回影响的行数
```

改：

```
int updateCount = await dataBase.rawUpdate("update my_table set password = ?
where username = ?",["pwd2","flutter"]);//返回影响的行数
```

查：

```
List<Map> dataList = await dataBase.rawQuery("select * from my_table where
username= ?",["flutter"]);//返回所有的查询结果，以刚刚的插入数据的SQL为例，返回的结果为：
[{id: 1, username: flutter, password: pwd1}]
```

在我们使用完数据库之后一定要记得关闭数据库，避免资源的浪费，关闭数据库：

```
dataBase.close();
```

如果我们想要删除一个表就可以根在OnCreate里的操作一样使用execute方法：

```
dataBase.execute("drop table my_table");
```

如果我们想要执行多条数据库指令，需要使用到SQL中的事务来帮助我们保证数据的一致性：

```
 dataBase.transaction((txn) async {
        var batch = txn.batch();
        batch.rawInsert("insert into my_table(username,password) values(?,?)",
["flutter1","pwd2"]);
        batch.rawInsert("insert into my_table(username,password) values(?,?)",
["flutter2","pwd3"]);
        batch.rawInsert("insert into my_table(username,password) values(?,?)",
["flutter3","pwd4"]);
        batch.commit(noResult: true);//一并插入三条数据，并不需要返回结果
      });
```

数据库的基本操作就介绍完了，在实际使用的过程中这样的使用是不太符合我们日常习惯的，最好还是根据面向对象的方式将增删改查的方式封装到对象里。

接下来学习一下Flutter的Json解析

19.Flutter之JSON解析：

Json是我们最常用的数据传输格式，我们需要在发送数据的时候将对象转为Json，在收到数据后将Json转为对象，这需要我们的转化，我们可以自己手解这些Json字符串，也可以利用一些第三方库帮我们自动转化。
手动转化的我们引入转化的类库：

```
    import 'dart:convert';
```

引入之后我们就可以直接使用json库来帮助我们进行json和实体的转化了。
json解析：

```
    String jsonStr = '{"username":"flutter","password":"pwd"}';
    Map<String,dynamic> decode = json.decode(jsonStr);//这里帮助我们将json的字符串转为了Map集合
    print("username is ${decode['username']}");//这里使用Map的读取方式来获取username作为key的值
```

我们再把上面的decode转为json看是否可以：

```
    String codeStr = json.encode(decode);//直接将需要转的对象encode就可以了
    print("codeStr is ${codeStr}");//打印已经转为json的字符串
```

这是比较简单的json解析方式，我们再看下我们在项目中最常用的使用Json转对象的方法，我们可以在其他的平台看到有很多的转化的类库，我们这里使用官方提供给我们的json_serializable。
引入第三方的自动转化库：

```
dependencies:
  json_annotation: ^3.0.1
  json_serializable: ^3.3.0
```

创建一个实体类：

```
import 'package:json_annotation/json_annotation.dart';//引入第三方的类库
part 'user.g.dart';//在我们编译时会自动生成
@JsonSerializable()//这里告诉编译器它是一个需要解析json的类
class User{
  User(this.userName,this.password);
  @JsonKey(name:"name")//给字段在通过json转化的时候提供了一个别名
  String userName;
  String password;
//以下是固定写法，在写其他实体类时记得替换里面的名字就可以了
  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

在写完这些会发现有些地方还标红，不能正常使用，我们还需要编译一下自动生成没有的类和代码。我们需要在项目的根目录下运行：

```
flutter packages pub run build_runner build
```

或者：

```
flutter packages pub run build_runner watch
```

第一种方式是手动构建，如果我们创建了需要转换的实体后需要进行一次构建。第二种是启动观察器自动帮助我们构建，当发现有需要转换的实体后会自动编译帮助我们构建缺少的文件。
如果运行上面的命令出现了下面的错误提示还需要我们引入自动构建的类库：

```
Could not find package "build_runner". Did you forget to add a dependency?
pub finished with exit code 65
```

添加类库：

```
dependencies:
    build_runner: ^1.10.0
```

然后就可以正常使用了，实体类的创建看起来比较繁琐，其实已经有很多的第三方已经帮助我们做好了创建实体类的工具，比如：
1、在线生成工具：JsonToDart
2、以Android Studio为例可以安装FlutterJsonBeanFactory插件，安装成功后，我们选择菜单栏的File->New的时候就多出了一个JsonToDartBeanAction，输入类名和想要解析的Json就得到了想要的Bean。
准备工作做好了，在使用的时候我们看下如何使用，还是以上一个例子中的json为例：

```
    //序列化为User对象
    User user = User.fromJson(json.decode(jsonStr));
    //将对象转化为String类型
    var userStr = User("flutter","pwd11").toJson();
```

面对复杂的对象类型还需要好好看下如果做好合理的结构，本篇的基本Json解析就学到这了。

接下来学习Flutter的动画

20.Flutter之动画

为了让我们的场景切换更加流畅和一些元素更能抓住用户的眼球，我们在APP中都会添加多种动画来装扮我们的APP，在以往的开发中对于普通的旋转位移，缩放动画我们都有一些简单的处理方式，但是面对复杂的动画我们都会耗费我们大量的时间来进行计算，我们一起来针对Flutter中的动画进行学习，看下在Flutter中我们如何处理我们日常使用的动画。
在Flutter中也为我们提供了多种动画示例，我们稍后看下，我们先看下如何不使用这些动画示例来创建一个我们想要的动画。
我们首先需要一个动画的控制单元，在Flutter中AnimationController来充当这个角色，我们先认识一下AnimationController：

`AnimationController`

```
AnimationController({
    double value, //给定动画的初始状态
    this.duration,//动画执行的时间间隔，如果没有设置reverseDuration则返回时的持续时间也是duration指定的时间
    this.reverseDuration,//返回原来状态的动画运行时间
    this.debugLabel,//调试的标签
    this.lowerBound = 0.0,//动画消失时的值
    this.upperBound = 1.0,//动画完成时的值
    this.animationBehavior = AnimationBehavior.normal,//如果设置为Normal如果设置的disableAnimations为True则不会浪费时间，如果设置为 AnimationBehavior.preserve则会消耗同样的时间
    @required TickerProvider vsync,//防止屏幕外动画，消耗不必要的资源
  })
```

范围控制我们看到在AnimationController为我们提供了0-1的控制，但是在实际使用中很多情况是无法满足我们需求的，我们这时候需要Tween来帮助我们做一个范围的设定，可以设定动画变化的范围。Tween提供了多种样式的变化，如：IntTween，ColorTween,RectTween等变化的范围设定。我们简单来了解下他们，然后可以根据需要在使用的时候进行选择。

```
IntTween({ int begin, int end }) : super(begin: begin, end: end);//开始的数值，和结束的数值，都是提供的整数
```

```
ColorTween({ Color begin, Color end }) : super(begin: begin, end: end);//开始的颜色，和结束的颜色，提供的是颜色值
```

我们把他们结合起来看可以做些什么：

```
    controller = AnimationController(
        duration: Duration(milliseconds: 1000), vsync: this);//创建一个动画的控制器
    Tween tween = Tween<double>(begin: 100.0, end: 400.0);//创建一个动画调整大小的限定器
    animation = tween.animate(controller);//将限定器与控制器进行绑定，就可以得到一个Animation对象
    animation.addListener(() {//当屏幕上每一帧变化的时候都会在这里进行刷新通知，每次刷新都调用我们build方法重绘当前控制的元素
      setState(() {});
    });
    controller.forward();//开始执行动画
```

如果我们想要监听动画的开始和结束需要给animation添加状态监听：

```
    animation.addStatusListener((status) {
      if(status == AnimationStatus.completed){//如果当前状态是已经完成了动画则往下执行
        controller.reverse(from:1.0);//将动画反向执行一遍
      }
    });
```

有了动画的控制我们只需要根据返回的状态值，控制build里面的元素显示就可以了，比如我们控制Flutter图标的一个放大动画：

```
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        margin: EdgeInsets.symmetric(vertical: 10.0),
        height: animation.value,//这里根据返回的限定值进行刷新，将logo显示为指定的大小
        width: animation.value,//这里根据返回的限定值进行刷新，将logo显示为指定的大小
        child: FlutterLogo(),显示的Flutter图标的组件
      ),
    );
  }
```

其他的动画比如位移和旋转我们可以控制margin和transform的设置，具体的属性设置可以参照之前我们介绍容器时的参数说明。
这样虽然能帮助我们实现动画，如果我们需要重复利用这个动画则需要实现多处，我们这时候可以使用AnimatedWidget来帮助我们做一些简化，Flutter为我们提供了一些AnimatedWidget的示例，比如FadeTransition、PositionedTransition、RelativePositionedTransition、RotationTransition、

ScaleTransition、SizeTransition、SlideTransition。这些是已经帮我们做好的了动画模式。我们帮上面的例子通过AnimatedWidget来改造一下。

```
class ImageAnimationWidget extends AnimatedWidget {//创建一个ImageAnimationWidget继承自AnimatedWidget来实现动画

  ImageAnimationWidget({Key key, Animation<double> animation})
      : super(key: key, listenable: animation);

  @override
  Widget build(BuildContext context) {
    final Animation<double> animation = listenable;
    return Center(
      child: Container(
        margin: EdgeInsets.symmetric(vertical: 10.0),
        height: animation.value,
        width: animation.value,
        child: FlutterLogo(),
      ),
    );
  }
}
```

这块的代码就是我们刚刚State里的build代码，我们把动画部分提取了出来，这样我们在以后需要多个该动画时直接引用即可，原来的build代码，现在已经被替换为了：

```
@override
  Widget build(BuildContext context) {
    return ImageAnimationWidget(animation: animation,);
  }
```

还是上面的例子，如果我们想要使用Flutter为我们提供的示例来实现怎么做呢：

```
  @override
  Widget build(BuildContext context) {
    return ScaleTransition(scale: animation,child: FlutterLogo(),);
  }
```

就是把State里的build方法替换为了ScaleTransition，然后我们需要修改animation中的Tween的开始结束大小，方便了很多，其他的动画我们可以自己进行实验一下。

通过本篇我们学习了一些简单的动画，Flutter的基础学习也告一段落，接下来需要开一个新的专题，一起学习一个项目的0开始。

21.Flutter之Row使用

1、解决平分宽度，并各自从左开始布局

```
                        Row(
                          mainAxisAlignment: MainAxisAlignment.start,
                          mainAxisSize: MainAxisSize.max,
                          crossAxisAlignment: CrossAxisAlignment.center,
                          //元素与空白互相间隔
```

```
                            children: [
                              Expanded(
                                flex: 1,//设置权重
                                child: Row(
                                  mainAxisAlignment: MainAxisAlignment.start,
                                  textDirection: TextDirection.ltr,
                                  children: [
                                    Text(
                                      "企业生产状态:",
                                      style: TextStyle(color: Colors.black26),
                                    ),
                                    Text(
                                      "正常",
                                      style: TextStyle(color: Colors.black87),
                                    ),
                                  ],
                                ),
                              ),
                              Expanded(
                                  flex: 1,//设置权重
                                  child: Row(
                                    mainAxisAlignment: MainAxisAlignment.start,
                                    mainAxisSize: MainAxisSize.max,
                                    children: [
                                      Text(
                                        "行业:",
                                        style: TextStyle(color: Colors.black26),
                                      ),
                                      Expanded(//解决Text文本过长超出屏幕错误
                                        child: Text(
                                          mList[index].industryClass,
                                          maxLines: 1,
                                          overflow: TextOverflow.ellipsis,
                                          style:
                                              TextStyle(color: Colors.black87),
                                        ),
                                      ),
                                    ],
                                  )),
                            ],
                          ),
```

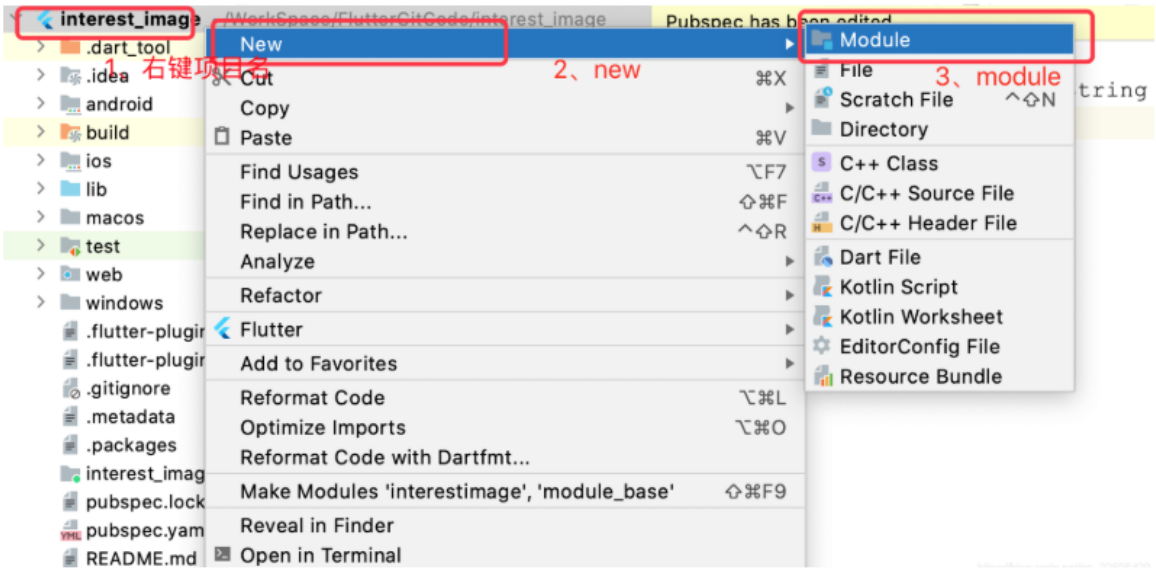
2、解决Row中Text文本过长超出屏幕错误

```
Row(
   mainAxisAlignment: MainAxisAlignment.start,
   mainAxisSize: MainAxisSize.max,
   children: [
           Text("行业:",
                style: TextStyle(color: Colors.black26),
                ),
           Expanded(//解决Text文本过长超出屏幕错误
                    child: Text("很长很长的文本很长很长的文本一行装不下",
                           maxLines: 1,
                           overflow: TextOverflow.ellipsis,
                           style:TextStyle(color: Colors.black87),),
                    ),
```

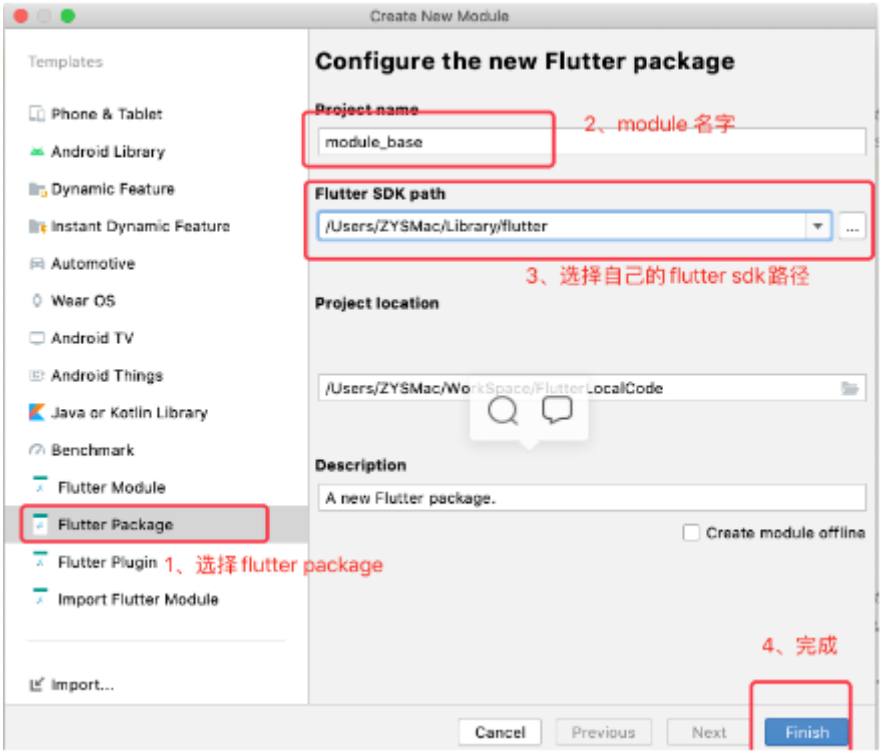
```
            ],
    )
```

22.Flutter 主工程引入包，模块化

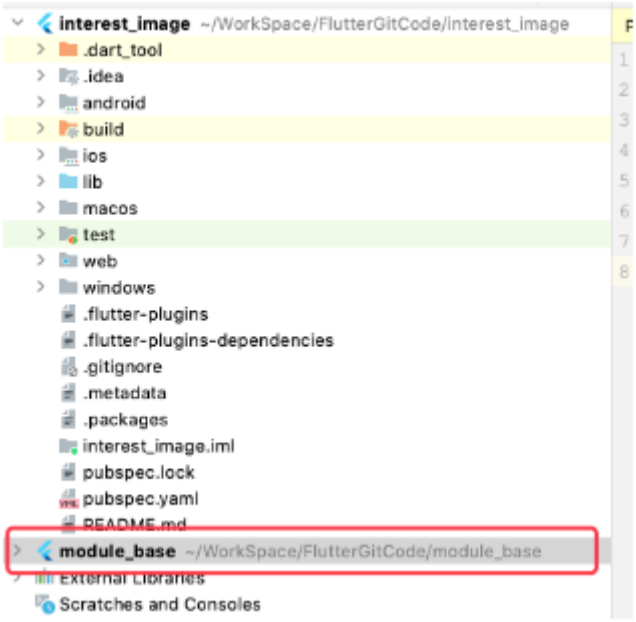
1、右键项目名，新建module



2、选择module类型

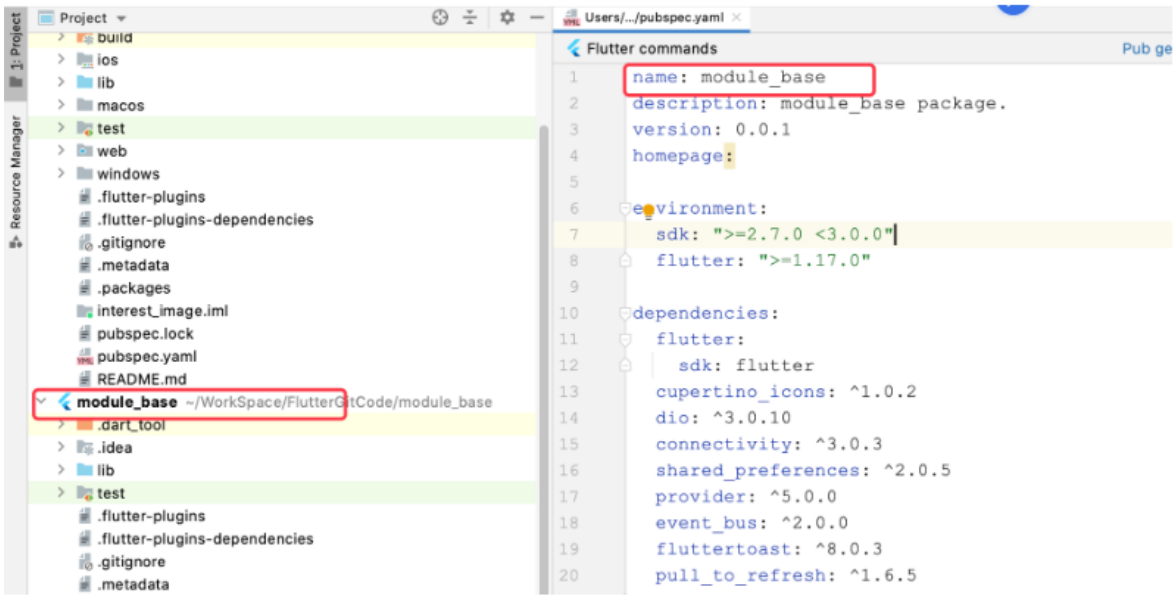


完成后，不管module路径在哪，都会自动添加都项目下

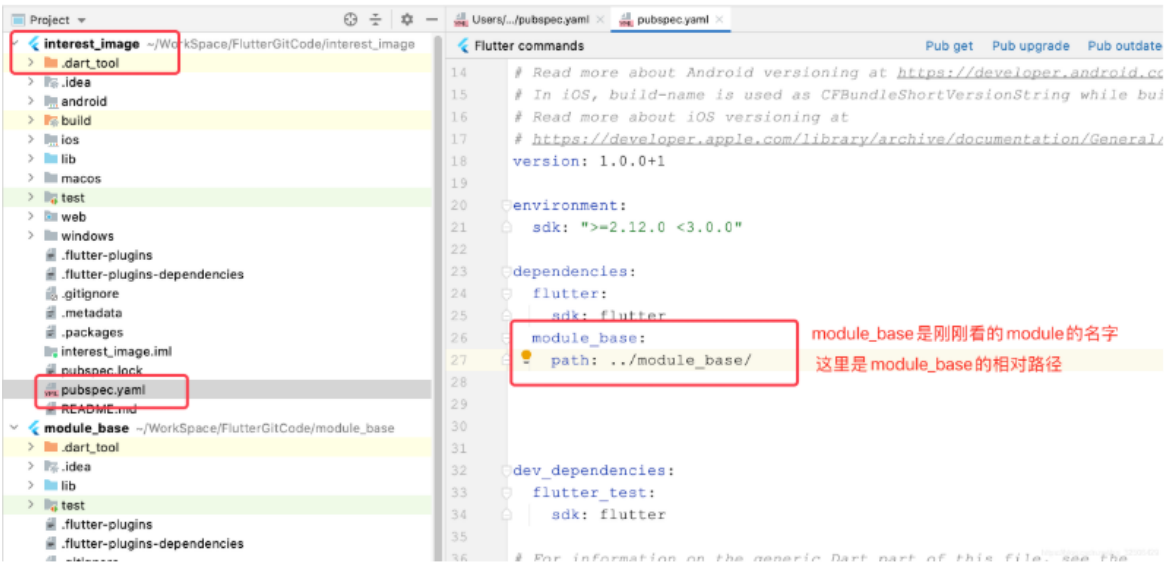


3、主工程配置module

（1）、查看module名字

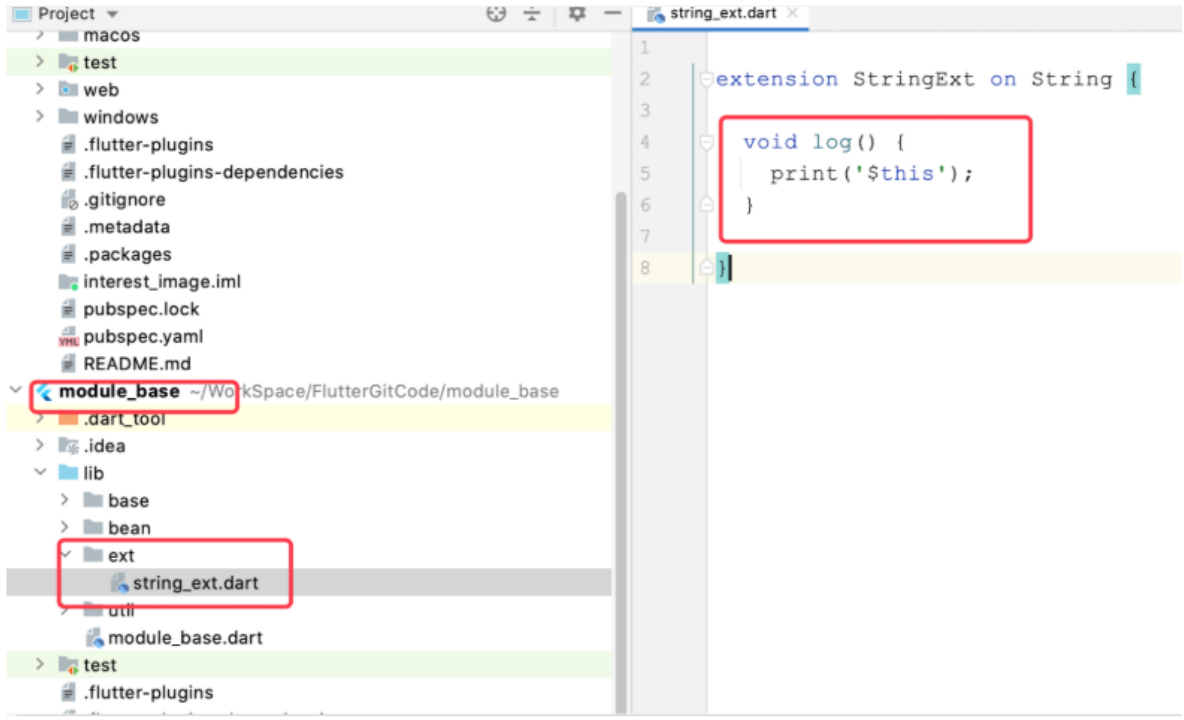


（2）、进入主工程配置文件

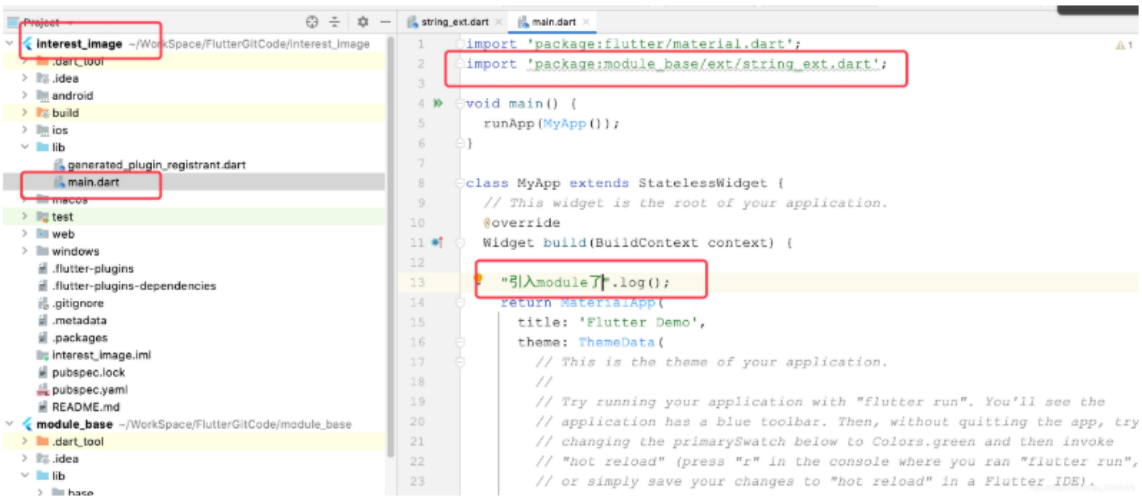


4、测试

（1）、在module_base中新建扩展函数



（2）、在主工程中使用



23.Flutter项目实战（上）

作为Flutter实战的开篇，我们需要介绍下我们要做的内容以及我们的准备工作，为了能让我们的实战顺利进行需要一个开放的API接口服务平台，选择了半天最终选择了一个开发者平台，感谢玩安卓的开发者提供的开放API，可以让我们在练习一些项目的时候使用。

确定了我们要做什么我们就可以着手准备了，我们先把项目的结构分为哪些模块进行设计。

```
lib
    -constants    //存放一些常量
    -events       //跨界面的事件传递
    -model        //解析的数据和数据库数据
    -network      //网络请求以及网络配置
    -pages        //单独页面的集合
    -route        //页面的路由配置，避免多出配置导致的混乱
    -utils        //一些工具类
    -widgets      //小的工具组件，单独的widget
```

整体的一个结构就出来了，根据习惯我们还是先准备一些工具类，和网络的请求，先把与业务无关的工具准备好，这样让我们以后开始的时候更加的顺畅。网络的三方库配置在以前已经有介绍，不再多说了，我们当时也说在以后的实践中是需要进行一个封装的，我们一起来研究下如何能做一个合理的封装：
我选择的是创建一个单例的Http配置类，这样我们可以对Dio进行一次配置通用的效果：

```
class HttpConfig {
  static String BASE_URL = "";
  static HttpConfig _config;

  static HttpConfig get instance => _getInstance();
  Dio dio;

  HttpConfig._internal() {
    var baseOption = BaseOptions(
```

```
      connectTimeout: 6000,
      sendTimeout: 3000,
      baseUrl: BASE_URL,
      headers: HttpHeaderConfig.getHeaderConfig(),//这里使用的一个方法而不是一个变量来配置的header，因为有些header我们会放到sharePerence里，所以我们使用一个方法来可以随时进行读取
    );
    dio = Dio(baseOption);
    dio.interceptors.add(InterceptorsWrapper(//这里创建一个拦截器，拦截器在我们根据需要进行消息的拦截
        onRequest: (option) {},
        onResponse: (response) {
          if (response.statusCode == 400 || response.statusCode == 404) {}//这里拦截了错误的状态的返回
        },
        onError: (error) {}));
  }

  static HttpConfig _getInstance() {
    if (_config == null) {
      _config = HttpConfig._internal();
    }
    return _config;
  }

  //这里创建一个get请求的方法，不是直接把dio做成公用而是提供get方法是为了以后如果我们替换整个网络请求库的时候方便更换，还有一个原因，是我们在一般请求的时候需要对参数进行签名，这里可以帮助我们做统一的处理。
  void get<T>(String path,
      {Map<String, dynamic> params,
      Function(T t) onSuccess,          //这里对返回结果已经做了处理也是为了以后如果网络框架修改后能低成本迁移
      Function(int error) onError}) async {
    Response response;
    try {
      response = await dio.get(path, queryParameters: params);
      if (response.statusCode == 200 && onSuccess != null) {//先做过滤，这样我们在处理的时候就省去了很多重复代码
        onSuccess(response.data);
      } else if (onError != null) {
        onError(response.statusCode);   //如果状态码不对，则返回错误数据
      }
    } catch (e) {
      if (onError != null && response != null) {
        onError(-1);  //如果抛出了异常，则返回错误数据
      }
    }
  }
 //参照get的说明
  void post<T>(String path,
      {Map<String, dynamic> data,
      Function(T t) onSuccess,
      Function(int error) onError}) async {
    Response response;
    try {
      response = await dio.post(path, data: data);
      if (response.statusCode == 200 && onSuccess != null) {
        onSuccess(response.data);
```

```
      } else if (onError != null) {
        onError(response.statusCode);
      }
    } catch (e) {
      if (onError != null && response != null) {
        onError(-1);
      }
    }
  }
}
```

创建的HttpHeader的方法：

```
class HttpHeaderConfig {
  static Map<String, dynamic> getHeaderConfig() {
    Map<String,dynamic> config = Map<String,dynamic>();
    config[""] = "";//这里我们堆放我们的header，如果需要从数据库或者sharePerence中获取数据可以在这里获取
    return config;
  }
}
```

网络请求的方法我们就算是封装完了，然后我们可能还需要将网络返回的数据进行一个保存，我们先将sharePreferences进行一个封装。
sharePreference的第三方引入前面也有介绍过了，我们可以看下前面是如何进行引入的。然后我们对SharedPreferences进行封装：这里的封装参照了阿里开源的Flutter-Go，

```
//这里也是创建了一个单例的SharedPreferences
class SharedPrefrencesUtils {
  static SharedPrefrencesUtils _prefrencesUtils;
  static SharedPreferences _sharedPerence;

  static Future<SharedPrefrencesUtils> get instance async {
    return await _getInstance();
  }

    //因为SharedPreferences的初始化是耗时操作，需要在读取的时候需要添加await关键字，这也是这里的单例与网络请求有些许不同的原因。
  static Future<SharedPrefrencesUtils> _getInstance() async {
    if (_prefrencesUtils == null) {
      _prefrencesUtils = SharedPrefrencesUtils._internal();
    }
    if (_sharedPerence == null) {
      await _prefrencesUtils._init();
    }
    return _prefrencesUtils;
  }

  Future _init() async {
    _sharedPerence = await SharedPreferences.getInstance();
  }

  SharePrefrencesUtils._internal() {}

 //检查是否存在某个Key
  bool hasKey(String key) {
```

```
    if (_beforeCheck() || key == null) {
      return false;
    }
    return _sharedPerence.containsKey(key);
  }
  //存放字符串
  Future<bool> putString(String key, String value) {
    if (_beforeCheck()) return null;
    return _sharedPerence.setString(key, value);
  }

  Future<bool> putBool(String key, bool value) {
    if (_beforeCheck()) return null;
    return _sharedPerence.setBool(key, value);
  }

  Future<bool> putDouble(String key, double value) {
    if (_beforeCheck()) return null;
    return _sharePerence.setDouble(key, value);
  }

  Future<bool> putInt(String key, int value) {
    if (_beforeCheck()) return null;
    return _sharePerence.setInt(key, value);
  }

  String getString(String key) {
    if (_beforeCheck()) return null;
    return _sharedPerence.getString(key);
  }

  bool getBool(String key) {
    if (_beforeCheck()) return null;
    return _sharedPerence.getBool(key);
  }

  double getDouble(String key) {
    if (_beforeCheck()) return null;
    return _sharedPerence.getDouble(key);
  }

  int getInt(String key) {
    if (_beforeCheck()) return null;
    return _sharedPerence.getInt(key);
  }

  static bool _beforeCheck() {
    if (_sharedPerence == null) {
      return true;
    }
    return false;
  }
}
```

只是有SharedPreferences还不能满足我们现在APP的需要，我们还需要使用数据库来帮助我们存储更多的数据内容，数据库的引入在前面的介绍中我们也都已经进行了介绍，现在把数据库来进行一个封装：

```
class DataBaseUtils {
  static DataBaseUtils _dataBaseUtils;
  static Database _dataBase;

  static String _DATABASE_PATH = "exercise_db.db";
  static int _DATABASE_VERSION = 1;
  static String _CREATE_TABLE =
      "create table my_table (id integer primary key,userid text,password
text)";

  static Future<DataBaseUtils> get instance async {
    return await _getInstance();
  }

  //因为DataBase的初始化是耗时操作，需要在读取的时候需要添加await关键字，这也是这里的单例与网
络请求有些许不同的原因。
  static Future<DataBaseUtils> _getInstance() async {
    if (_dataBaseUtils == null) {
      _dataBaseUtils = DataBaseUtils._internal();
    }
    if (_dataBase == null) {
      await _dataBaseUtils._init();
    }
    return _dataBaseUtils;
  }

  Future _init() async {
    _dataBase = await openDatabase(_DATABASE_PATH, version: _DATABASE_VERSION,
        onCreate: (dataBase, version) {
      if (version == 1 && dataBase.isOpen) {
        dataBase.execute(_CREATE_TABLE);
      }
    }, onUpgrade: (dataBase, oldVersion, newVersion) {
      //这里预留了更新的操作，在version为1的时候并不会触发
      if (oldVersion == 1 && newVersion == 1 && dataBase.isOpen) {}
    });
  }

  ///插入数据
  ///tableName 需要插入的表名
  ///params 需要插入的数据
  Future<int> insert(String tableName, Map<String, dynamic> params) async {
    if (_dataBase != null && _dataBase.isOpen) {
      var dataId = await _dataBase.insert(tableName, params);
      return dataId;
    }
    return 0;
  }

  ///删除数据
  /// tableName 需要查的表名
  /// conditions 查询的条件
  /// mod 查询的模式，是否是用And连接起来的查询条件，可选['And','Or']
  Future<int> delete(String tableName,
      {Map<String, dynamic> conditions, String mod = "And"}) async {
    return await _dataBase.delete(tableName,
        where: _formatCondition(conditions, mod: mod));
```

```
}

///更新数据
/// tableName 需要查的表名
/// values 需要更新的数据
/// conditions 查询的条件
/// mod 查询的模式，是否是用And连接起来的查询条件，可选['And','Or']
Future<int> update(String tableName,
    {Map<String, dynamic> values,
    Map<String, dynamic> conditions,
    String mod = "And"}) async {
  return await _dataBase.update(tableName, values,
      where: _formatCondition(conditions, mod: mod));
}

/// tableName 需要查的表名
/// conditions 查询的条件
/// mod 查询的模式，是否是用And连接起来的查询条件，可选['And','Or']

Future<List> query(String tableName,
    {Map<String, dynamic> conditions, String mod = "And"}) async {
  if (_dataBase != null && _dataBase.isOpen) {
    return await _dataBase.query(tableName,
        where: _formatCondition(conditions, mod: mod));
  }
  return null;
}

DataBaseUtils._internal() {}


 ///格式化数据，将条件组合格式化为String
String _formatCondition(Map<String, dynamic> conditions,
    {String mod = "And"}) {
  var conditionStr = "";
  var index = 0;
  conditions.forEach((key, value) {
    if (value == null) {
      return;
    }
    if (value.runtimeType == String) {
      conditionStr = '$conditionStr $key like $value';
    }
    if (value.runtimeType == int ||
        value.runtimeType == double ||
        value.runtimeType == bool) {
      conditionStr = '$conditionStr $key = $value';
    }
    if (index >= 0 && index < conditions.length - 1) {
      conditionStr = '$conditionStr $mod';
    }
    index++;
  });
  return conditionStr;
}

void close(){
  if(_dataBase != null && _dataBase.isOpen){
```

```
      _dataBase.close();
    }
  }
}
```

现在与我们数据请求相关的内容就封装好了，从网络请求到数据的持久化都已经准备好了，下面我们准备一下资源的管理。

项目已经同步更新到了GitHub，有需要的同学可以看下。如有好的意见和建议可以提出共同探讨。

24.Flutter项目实战（下）

上篇文章中我们的项目结构以及基础的网络，存储以及配置完成了，接下来将所有页面的主题颜色，通用下拉刷新来做封装，写下来感觉有些地方的不是很好描述或者思路大于写博客的意义，所以在实战篇中不会把每个部分代码都再拿出来写了，感觉意义不大，Flutter的项目例子也找了看了一些没有看到特别好的分层结构，我也在想如何针对Flutter的多层嵌套的问题作出一个比较简单易做的分隔，减少多层级的嵌套，如果有好的思路可以一起讨论。
接下来我可能只会将遇到的问题和一些解决问题的思路整理一下来做分享，更多的内容可以去GitHub上去看下，如果有不妥的地方还请多多指点，大家共同学习。

在以前的APP上经常看到一键换肤的功能，在Flutter中会比较简单的实现，我们每个页面都是用统一的主题颜色，修改提供的统一颜色变量，然后刷新当前页面即可。我们先看下主题变量的设置：

```
class Style {
  //主题颜色的变量，为了以后如果出现换肤的预留
  static int THEME_COLOR = 0xFFCC1100;

  //全局的背景颜色
  static int BACKGROUND_COLOR = 0xFFFFFFFF;

  //设置全局的主题颜色
  static Color themeColor = Color(THEME_COLOR);
}
```

很是简单，然后在我们的页面的使用主题颜色的部分时直接使用：

```
color: Style.themeColor
```

其他的背景颜色，字体颜色都是类似的设置，不再多赘述。
下拉刷新的通用封装，这里借用了第三方的框架`pull_to_refresh`，我们一起看下如何使用下拉刷新：
首先我们先引入第三方的库：

```
dependencies:
    pull_to_refresh: ^1.5.8
```

在我们需要使用的类中引入下拉刷新的类：

```
import 'package:pull_to_refresh/pull_to_refresh.dart';
```

这里我做了一个全局的下拉刷新管理类，目的是为了让多个地方使用的时候不用逐个去进行初始化的操作，让它能更简单的实现我们的功能：

```
class CommonRefreshWidget extends SmartRefresher {
  ///这里的继承是为了能初始化一部分参数，
  ///这样以后使用的时候就可以拿到统一模板化的RefreshWidget，
  ///并且还可以根据自己的需要进行部分参数的修改
  CommonRefreshWidget(
      {Key key,
      @required controller,
      child,
      header:const ClassicHeader(
        height: 10,
      ),
      footer:const ClassicFooter(

      ),
      enablePullDown: true,
      enablePullUp: true,
      enableTwoLevel: false,
      onRefresh,
      onLoading,
      onTwoLevel,
      onOffsetChange,
      dragStartBehavior,
      primary,
      cacheExtent,
      semanticChildCount,
      reverse,
      physics,
      scrollDirection,
      scrollController})
      : assert(controller != null),
        super(
          key: key,
          controller: controller,
          child: child,
          header: header,
          footer: footer,
          enablePullDown: enablePullDown,
          enablePullUp: enablePullUp,
          enableTwoLevel: enableTwoLevel,
          onRefresh: onRefresh,
          onLoading: onLoading,
          onTwoLevel: onTwoLevel,
          onOffsetChange: onOffsetChange,
          dragStartBehavior: dragStartBehavior,
          primary: primary,
          cacheExtent: cacheExtent,
          semanticChildCount: semanticChildCount,
          reverse: reverse,
          physics: physics,
          scrollDirection: scrollDirection,
          scrollController: scrollController,
        );
}
```

这里比较简单，不用多加解释，其实很多地方我们都是可以做类似的操作的，这是我们可以做到的最低成本的封装，接下来我会将一部分有类似需求的功能进行同样封装，这里就不再占用更多的篇幅了。
接下来会去介绍项目中遇到的问题和一些自己的想法。可以持续关注。

25.Flutter问题汇总

一起从0开始Flutter实战!

项目开始了，我们把已经搭建的框架运行一下，果然有报错的出现，我们先看下第一个出现的异常：

```
 What went wrong:
 Could not determine the dependencies of task ':app:compileDebugJavaWithJavac'.
 Could not resolve all task dependencies for configuration
':app:debugCompileClasspath'.
 Could not resolve io.flutter:flutter_embedding_debug:1.0.0-
6bc433c6b6b5b98dcf4cc11aff31cdee90849f32.
    Required by:
    project :app
    project :app > project :shared_preferences
    project :app > project :sqflite
 > Skipped due to earlier error
 > Skipped due to earlier error
 > Skipped due to earlier error
```

这里的异常是因为我们在项目的配置没有配置完整，我们来看下解决方法：
打开android/build.gradle文件添加：

```
allprojects {
    repositories {
        google()
        jcenter()
        flatDir {
            dirs 'libs'
        }
        maven {
            url 'http://download.flutter.io'
        }
        maven { url "https://jitpack.io" }
    }
}
```

问题二：

```
* What went wrong:
 Execution failed for task ':app:stripDebugDebugSymbols'.
 No version of NDK matched the requested version 21.0.6113669. Versions
available locally: 20.0.5594570, 21.2.6472646
 Try:
 Run with --stacktrace option to get the stack trace. Run with --info or --debug
option to get more log output. Run with --scan to get full insights.
```

问题原因我们NDK本地有多个版本可选，所以我们可以指定版本，在android/app/build.gradle文件中修改：

```
android {
    compileSdkVersion 28
    ndkVersion "21.2.6472646"
    }
```

加指定版本。