整理 Flutter 相关面试题全解析

Dart 部分

1. Dart 语言的特性？

- Productive（生产力高，Dart 的语法清晰明了，工具简单但功能强大）
- Fast（执行速度快，Dart 提供提前优化编译，以在移动设备和 Web 上获得可预测的高性能和快速启动。）
- Portable（易于移植，Dart 可编译成 ARM 和 X86 代码，这样 Dart 移动应用程序可以在 iOS、Android 和其他地方运行）
- Approachable（容易上手，充分吸收了高级语言特性，如果你已经知道 C++，C 语言，或者 Java，你可以在短短几天内用 Dart 来开发）
- Reactive（响应式编程）

2.Dart 的一些重要概念？

- 在 Dart 中，一切都是对象，所有的对象都是继承自 Object
- Dart 是强类型语言，但可以用 var 或 dynamic 来声明一个变量，Dart 会自动推断其数据类型,dynamic 类似 c#
- 没有赋初值的变量都会有默认值 null
- Dart 支持顶层方法，如 main 方法，可以在方法内部创建方法
- Dart 支持顶层变量，也支持类变量或对象变量
- Dart 没有 public protected private 等关键字，如果某个变量以下划线（_）开头，代表这个变量在库中是私有的

3.Dart 当中的 「..」表示什么意思？

- Dart 当中的 「..」意思是 「级联操作符」，为了方便配置而使用。
- 「..」和「.」不同的是 调用「..」后返回的相当于是 this，而「.」返回的则是该方法返回的值 。
- **2. Dart 的作用域**
- Dart 没有 「public」「private」等关键字，默认就是公开的，私有变量使用 下划线 _开头。

- **3. Dart 是不是单线程模型？是如何运行的？**
- Dart 是单线程模型，如何运行的看这张图：
- 
- 引用《Flutter 中文网》里的话：
- Dart 在单线程中是以消息循环机制来运行的，其中包含两个任务队列，一个是“微任务队列” microtask queue，另一个叫做“事件队列” event queue。
- 入口函数 main() 执行完后，消息循环机制便启动了。首先会按照先进先出的顺序逐个执行微任务队列中的任务，当所有微任务队列执行完后便开始执行事件队列中的任务，事件任务执行完毕后再去执行微任务，如此循环往复，生生不息。

4. Dart 多任务如何并行的？

- 刚才也说了，既然 Dart 不存在多线程，那如何进行多任务并行？
- Dart 当中提供了一个 **类似于新线程，但是不共享内存的独立运行的 worker - isolate**。
- 那他们是如何交互的？
- 这里引用 flutter 入门之 dart 中的并发编程、异步和事件驱动详解 中的一部分答案：
- 
- 在 dart 中，一个 Isolate 对象其实就是一个 isolate 执行环境的引用，一般来说我们都是通过当前的 isolate 去控制其他的 isolate 完成彼此之间的交互，而当我们想要创建一个新的 Isolate 可以使用 Isolate.spawn 方法获取返回的一个新的 isolate 对象，两个 isolate 之间使用 SendPort 相互发送消息，而 isolate 中也存在了一个与之对应的 ReceivePort 接受消息用来处理，但是我们需要注意的是，ReceivePort 和 SendPort 在每个 isolate 都有一对，只有同一个 isolate 中的 ReceivePort 才能接受到当前类的 SendPort 发送的消息并且处理。

5.dart 是值传递还是引用传递？

dart 是值传递。我们每次调用函数，传递过去的都是对象的内存地址，而不是这个对象的复制。
先来看段代码

```
main(){

  Test a = new Test(5);
```

```
  print("a 的初始值为：${a.value}");

  setValue(a);

  print("修改后 a 的值为：${a.value}");

}



class Test{

  int value = 1;

  Test(int newValue){

    this.value = newValue;

  }

}



setValue(Test s){

  print("修改 value 为 100");

  s.value = 100;

}
```

输出结果为：

```
a 的初始值为：5

修改 value 为 100

修改后 a 的值为:100
```

从这里可以看出是值传递，如果只是复制了一个对象，然后把这个新建的对象地址传递到函数里面的话，setValue()函数中的修改是不会影响到 main 函数中的 a 的，因为二者所引用的内存地址是不一样。
有些人可能会以以下代码反驳我：

```
main(){

  int s = 6;

  setValue(s);
```

```
  print(s); //输出 6，而不是 7
}

class Test{
  int value = 1;
  Test(int newValue){
    this.value = newValue;
  }
}

setValue(int s){
  s += 1;
}
```

你看，这输出的不是 6 吗，在 dart 中一切皆为对象，如果是引用传递，那为什么是 6 啊。答案是这样的，在 setValue()方法中，参数 s 实际上和我们初始化 int s = 6 的 s 不是一个对象，只是他们现在指的是同一块内存区域，然后在 setValue()中调用 s += 1 的时候，这块内存区域的对象执行+1 操作，然后在堆(类比 java)中产生了一个新的对象，s 再指向这个对象。所以 s 参数只是把 main 函数中的 s 的内存地址复制过去了，就比如 java 中的：

```
public class Test {
    public static void main(String[] args) {
        Test a = new Test();
        Test b = a;
        b = new Test();
    }
}
```

我们只要记住一点，参数是把内存地址传过去了，如果对这个内存地址上的对象修改，那么其他位置的引用该内存地址的变量值也会修改。千万要记住 dart 中一切都是对象。

6. Dart 属于是强类型语言 ，但可以用 var 来声明变量，Dart 会自推导出数据类型，var 实际上是编译期的“语法糖”。dynamic 表示动态类型，被编译后，实际是一个 object 类型，在编译期间不进行任何的类型检查，而是在运行期进行类型检查。

7. Dart 中 if 等语句只支持 bool 类型，switch 支持 String 类型。

8.Dart 中数组和 List 是一样的。

9.Dart 中，Runes 代表符号文字 ，是 UTF-32 编码的字符串，用于如 Runes input = new Runes('\u{1f596} \u{1f44d}');

10.Dart 支持闭包。

11.Dart 中 number 类型分为 int 和 double ，没有 float 类型。

13.Dart 中 级联操作符 可以方便配置逻辑，如下代码：

```
event
  ..id = 1
  ..type = ""
  ..actor = "";
```

14. 说一下 Future？

- Future，字面意思「未来」，是用来处理异步的工具。
- 刚才也说过：
- Dart 在单线程中是以消息循环机制来运行的，其中包含两个任务队列，一个是“微任务队列” microtask queue，另一个叫做“事件队列” event queue。
- Future 默认情况下其实就是往「事件队列」里插入一个事件，当有空余时间的时候就去执行，当执行完毕后会回调 `Future.then(v)` 方法。
- 而我们也可以通过使用 `Future.microtask` 方法来向 「微任务队列」中插入一个任务，这样就会提高他执行的效率。
- 因为在 Dart 每一个 isolate 当中，执行优先级为 ： Main > MicroTask > EventQueue

15. 说一下 Stream？

- Stream 和 Feature 一样，都是用来处理异步的工具。
- 但是 Stream 和 Feature 不同的地方是 Stream 可以接收多个异步结果，而 Feature 只有一个。
- Stream 的创建可以使用 `Stream.fromFuture`，也可以使用 `StreamController` 来创建和控制。
- 还有一个注意点是：普通的 Stream 只可以有一个订阅者，如果想要多订阅的话，要使用 `asBroadcastStream()`。

16. 说一下 mixin？

- 关于什么是 mixin，引用 张风捷特烈 文章中的：
- 首先 mixin 是一个定义类的关键字。直译出来是混入，混合的意思 Dart 为了支持多重继承，引入了 mixin 关键字，它最大的特殊处在于： mixin 定义的类不能有构造方法，这样可以避免继承多个类而产生的父类构造方法冲突
-

16.Widget 和 element 和 RenderObject 之间的关系

首先我详细说下当时的情景，面试官问我 `Widget` 和 `Element` 之间是不是一对多的关系，如果是增加一个 `Widget` 之后，这个关系又是什么。 这部分还是没有很好地答案，现在只是一个猜想，如果添加了一个 `widget`，`Element` 树遍历后面所有的 `Element` 看类型是否发生改变，有的话再重建 `RenderObject`。`Element` 和 `Widget` 之间应该还是一对一的关系，因为每个 `Widget` 的 `context` 都是独一无二的。等想好了再写上去吧。

17. widget 树的 root 节点

还是没能理解面试官的意思。。有能够理解的同学请评论告知我一下。 现在理解了，面试官的意思应该指是 runApp()方法中的那个的 Widget。我当时也想说的，不过忘记这个方法名是啥了。。。

18. mixin extends implement 之间的关系

- 继承（关键字 extends）

Flutter 中的继承是单继承，构造函数不能继承，子类重写超类的方法，要用**@override**
子类调用超类的方法，要用 super
Flutter 中的子类可以访问父类中的所有变量和方法，因为 Flutter 中没有公有、私有的区别

-

混入 mixins （关键字 with）
mixins Dart 赋予的多继承特性，类似 C#的多继承，而 Java 是不支持类的多继承
混合的对象是类，可以混合多个

-
-

接口实现（关键字 implements）

-

Flutter 是没有 interface 的，但是 Flutter 中的每个类都是一个隐式的接口，这个接口包含类里的所有成员变量，以及定义的方法。
当 class 被当做 interface 用时，class 中的成员变量也需要在子类里重新实现。在成员变量前加**@override**

这三种关系可以同时存在，但是有前后顺序：
extends -> mixins -> implements

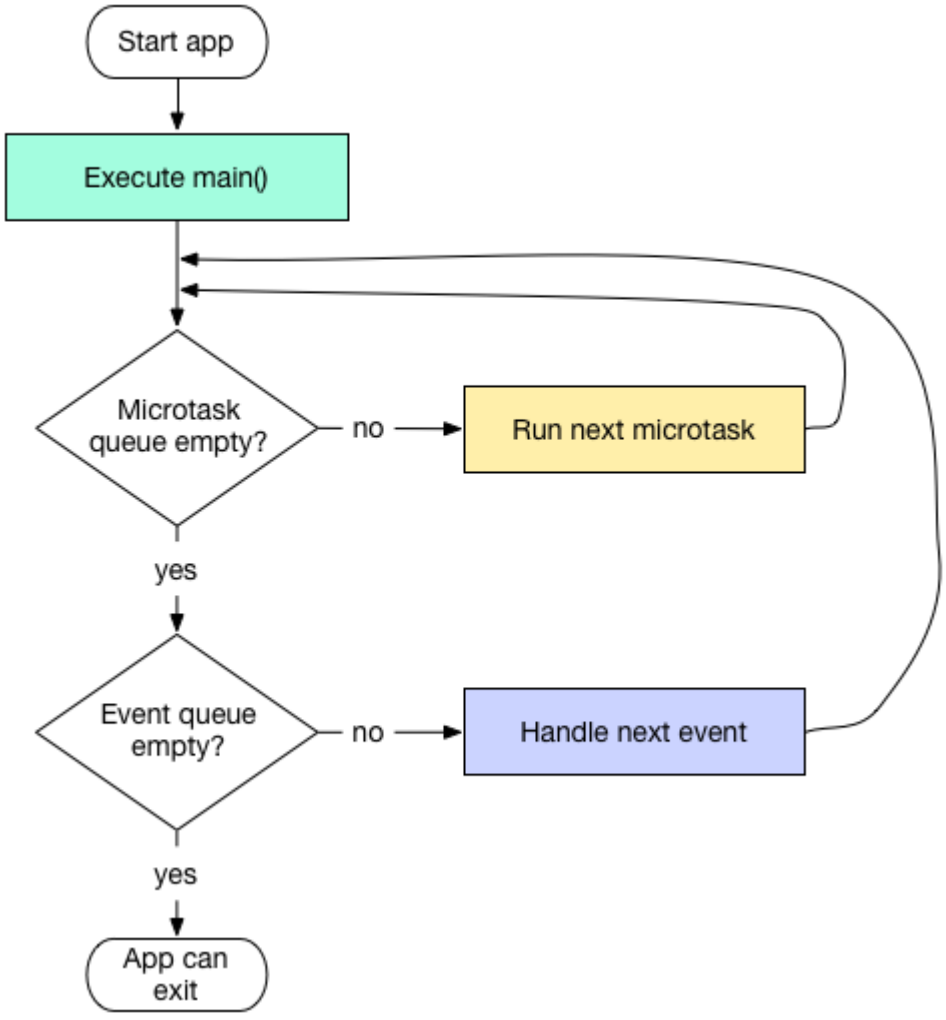
19.Future 和 microtask 执行顺序

MricroTask

Dart 中事件机制的实现 ：Main isolate 中有一个 Looper，但存在两个 Queue:Event Queue 和 Microtask Queue 。

Dart 中事件的执行顺序： Main > MicroTask > EventQueue。

在 Main 中写代码将最先执行；
执行完 Main 中的代码，然后会检查并执行 Microtask Queue 中的任务，
通常使用 scheduleMicrotask 将事件添加到 MicroTask Queue 中；
最后执行 EventQueue 队列中的代码，通常使用 Future 向 EventQueue 加入时间，也可以使用 async 和 await 向 EventQueue 加入事件。



Future 先进先出

Future 提供链式调用

```
new Future (() => print('拆分任务_1'))

    .then((i) => print('拆分任务_2'))

    .then((i) => print('拆分任务_3'))

    .whenComplete(()=>print('任务完成'));
```

多 Future 和 多 micTask 的执行顺序

```
void testScheduleMicrotatsk() {

  scheduleMicrotask(() => print('Mission_1'));



//注释 1

  new Future.delayed(new Duration(seconds: 1), () => print('Mission_2'));



//注释 2

  new Future(() => print('Mission_3')).then((_) {

    print('Mission_4');

    scheduleMicrotask(() => print('Mission_5'));

  }).then((_) => print('Mission_6'));



//注释 3

  new Future(() => print('Mission_7'))

      .then((_) => new Future(() => print('Mission_8')))
```

```
      .then((_) => print('Mission_9'));



//注释 4

  new Future(() => print('Mission_10'));



  scheduleMicrotask(() => print('Mission_11'));



  print('Mission_12');

}
```

输出结果

```
I/flutter (19025): Mission_12
I/flutter (19025): Mission_1
I/flutter (19025): Mission_11
I/flutter (19025): Mission_3
I/flutter (19025): Mission_4
I/flutter (19025): Mission_6
I/flutter (19025): Mission_5
I/flutter (19025): Mission_7
I/flutter (19025): Mission_10
I/flutter (19025): Mission_8
I/flutter (19025): Mission_9
Syncing files to device MIX 3...
I/flutter (19025): Mission_2
```

19.await for 的使用方式

代码如下

```
String data;

getData() async {
```

```
  data = await http.get(Uri.encodeFull(url), headers: {"Accept":
"application/json"});     //延迟执行后赋值给 data

}
```

- await 关键字必须在 async 函数内部使用
- 调用 async 函数必须使用 await 关键字
- 返回默认是一个 future 对象

```
//定义了返回结果值为 String 类型

Future<String> getDatas(String category) async {

    var request = await _httpClient.getUrl(Uri.parse(url));

    var response = await request.close();

    return await response.transform(utf8.decoder).join();

}



run() async{

    int data = await getDatas('keji');    //因为类型不匹配，IDE 会报错

}
```

Furture 的作用是为了链式调用

```
//案例 3
 funA(){
  ...set an important variable...    //设置变量
}

funB(){
  ...use the important variable...   //使用变量
}
main(){
  new Future.then(funA()).then(funB());   // 明确表现出了后者依赖前者设置的变量
值

  new Future.then(funA()).then((_) {new Future(funB())});    //还可以这样用

  //链式调用，捕获异常
```

```
  new Future.then(funA(),onError: (e) { handleError(e); }).then(funB(),onError:
(e) { handleError(e); })
}
```

20.赋值操作符

比较有意思的赋值操作符有：

```
AA ?? ″999″ ///表示如果 AA 为空，返回 999
AA ??= ″999″ ///表示如果 AA 为空，给 AA 设置成 999
AA ~/999 ///AA 对于 999 整除
```

21.可选方法参数

Dart 方法可以设置 **参数默认值** 和 **指定名称** 。

比如： `getDetail(Sting userName, reposName, {branch = "master"}){}` 方法，这里 branch 不设置的话，默认是“master”。参数类型 可以指定或者不指定。调用效果： `getRepositoryDetailDao("aaa","bbbb",branch:"dev");` 。

22.作用域

Dart **没有关键词** public 、private **等修饰符，** _ **下横向直接代表** private **，但是有** @protected **注解** 。

23.构造方法

Dart 中的多构造方法，可以通过命名方法实现。

默认构造方法只能有一个，而通过 `Model.empty()` 方法可以创建一个空参数的类，其实方法名称随你喜欢，而变量初始化值时，只需要通过 this.name 在构造方法中指定即可：

```
class ModelA {
  String name;
  String tag;

  //默认构造方法，赋值给 name 和 tag
  ModelA(this.name, this.tag);
```

```
  //返回一个空的 ModelA
  ModelA.empty();

  //返回一个设置了 name 的 ModelA
  ModelA.forName(this.name);
}
```

24.getter setter 重写

Dart 中所有的基础类型、类等都继承 Object ，默认值是 NULL， 自带 getter 和 setter ，而如果是 final 或者 const 的话，那么它只有一个 getter 方法，Object 都支持 getter、setter 重写：

```
  @override
  Size get preferredSize {
return Size.fromHeight(kTabHeight + indicatorWeight);
  }
```

25.Assert(断言)

assert 只在检查模式有效，在开发过程中，assert(unicorn == null); 只有条件为真才正常，否则直接抛出异常，一般用在开发过程中，某些地方不应该出现什么状态的判断。

26.重写运算符，如下所示重载 operator 后对类进行 +/- 操作。

```
class Vector {
  final int x, y;

  Vector(this.x, this.y);

  Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
  Vector operator -(Vector v) => Vector(x - v.x, y - v.y);

  ...
}

void main() {
  final v = Vector(2, 3);
  final w = Vector(2, 2);
```

```
  assert(v + w == Vector(4, 5));
  assert(v - w == Vector(0, 1));
}
```

支持重载的操作符 ：

<	+	|	[]
>	/	^	[]=
<=	~/	&	~
>=	*	<<	==
–	%	>>	

类、接口、继承

Dart 中没有接口，类都可以作为接口，把某个类当做接口实现时，只需要使用 implements ，然后复写父类方法即可。

Dart 中支持 mixins ，按照出现顺序应该为 extends 、 mixins 、implements 。

Zone

Dart 中可通过 Zone 表示指定代码执行的环境，类似一个沙盒概念，在 Flutter 中 **C++** 运行 Dart 也是在 _runMainZoned 内执行 runZoned 方法启动，而我们也可以通过 Zone ，在运行环境内捕获全局异常等信息:

```
  runZoned(() {
    runApp(FlutterReduxApp());
  }, onError: (Object obj, StackTrace stack) {
print(obj);
print(stack);
  });
```

同时你可以给 runZoned 注册方法，在需要时执行回调，如下代码所示，这样的在一个 Zone 内任何地方，只要能获取 onData 这个 ZoneUnaryCallback，就都可以调用到 handleData

```
///最终需要处理的地方
handleData(result) {
print("VVVVVVVVVVVVVVVVVVVVVVVVVV");
print(result);
}
```

```
///返回得到一个 ZoneUnaryCallback
var onData = Zone.current.registerUnaryCallback<dynamic,
int>(handleData);

///执行 ZoneUnaryCallback 返回数据
Zone.current.runUnary(onData, 2);
```

异步逻辑可以通过 scheduleMicrotask 可以插入异步执行方法：

```
Zone.current.scheduleMicrotask((){
  //todo something
});
```

更多可参看 ： 《Flutter 完整开发实战详解(十一、全面深入理解 Stream)》

Future

Future 简单了说就是对 Zone 的封装使用。

比如 Future.microtask 中主要是执行了 Zone 的 scheduleMicrotask ，而 result._complete 最后调用的是 _zone.runUnary 等等。

```
  factory Future.microtask(FutureOr<T> computation()) {
    _Future<T> result = new _Future<T>();
    scheduleMicrotask(() {
      try {
        result._complete(computation());
      } catch (e, s) {
        _completeWithErrorCallback(result, e, s);
      }
    });
return result;
  }
```

Dart 中可通过 async/await 或者 Future 定义异步操作，而事实上 async/await 也只是语法糖，最终还是通过编译器转为 Future。

有兴趣看这里 ：

generators

code_generator.dart

Flutter 完整开发实战详解(十一、全面深入理解 Stream)

Stream

Stream 也是有对 Zone 的另外一种封装使用。

Dart 中另外一种异步操作， async* / yield **或者** Stream **可定义** Stream **异步，** async* / yield **也只是语法糖，最终还是通过编译器转为** Stream**。Stream 还支持同步操作。**

1）、Stream 中主要有 Stream 、 StreamController 、StreamSink 和 StreamSubscription 四个关键对象，大致可以总结为：

StreamController ：如类名描述，用于整个 Stream 过程的控制，提供各类接口用于创建各种事件流。

StreamSink ：一般作为事件的入口，提供如 add ， addStream 等。

Stream ：事件源本身，一般可用于监听事件或者对事件进行转换，如 listen 、where 。

StreamSubscription ：事件订阅后的对象，表面上用于管理订阅过等各类操作，如 cacenl 、pause ，同时在内部也是事件的中转关键。

2)、一般通过 StreamController **创建** Stream**；通过** StreamSink **添加事件；通过** Stream **监听事件；通过** StreamSubscription **管理订阅。**

3）、Stream 中支持各种变化，比如 map 、expand 、where 、take 等操作，同时支持转换为 Future 。

更多可参看 ：《Flutter 完整开发实战详解(十一、全面深入理解 Stream)》

Flutter 部分

1. Flutter 是什么？

Flutter 是谷歌的移动 UI 框架，可以快速在 iOS 和 Android 上构建高质量的原生用户界面。 Flutter 可以与现有的代码一起工作。在全世界，Flutter 正在被越来越多的开发者和组织使用，并且 Flutter 是完全免费、开源的。

2. Flutter 特性有哪些？

快速开发（毫秒级热重载）

- 绚丽 UI（内建漂亮的质感设计 Material Design 和 Cupertino Widget 和丰富平滑的动画效果和平台感知）
- 响应式(Reactive，用强大而灵活的 API 解决 2D、动画、手势、效果等难题)
- 原生访问功能
- 堪比原生性能

3.基础知识

Flutter 和 React Native 不同主要在于 **Flutter UI 是直接通过 skia 渲染的 ，而 React Native 是将 js 中的控件转化为原生控件，通过原生去渲染的** ，相关更多可查看：《移动端跨平台开发的深度解析》。

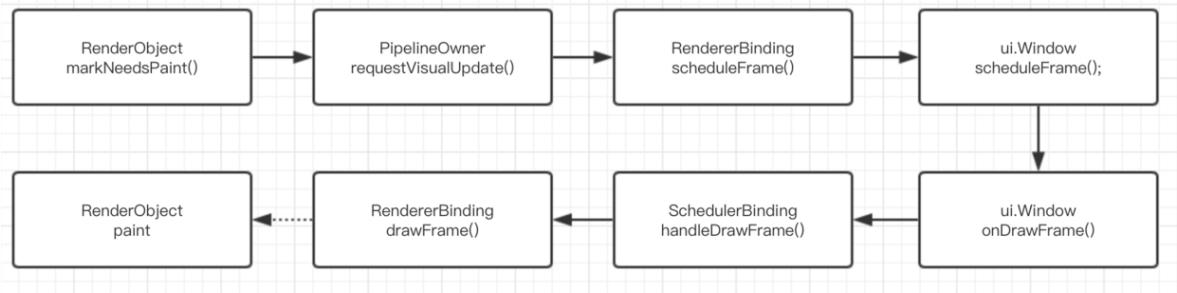
> Flutter 中存在 `Widget` 、 `Element` 、`RenderObject` 、`Layer` 四棵树，其中 `Widget` **与** `Element` **是一对多的关系** ，
>
> `Element` 中持有 `Widget` 和 `RenderObject` ， 而 `Element` **与** `RenderObject` **是一一对应的关系（除去** `Element` **不存在** `RenderObject` **的情况，如** `ComponentElement` **是不具备** `RenderObject`） ，
>
> 当 `RenderObject` 的 `isRepaintBoundary` 为 `true` 时，那么个区域形成一个 `Layer`，所以**不是每个** `RenderObject` **都具有** `Layer` **的，因为这受** `isRepaintBoundary` **的影响。**
>
> 更多相关可查阅 《Flutter 完整开发实战详解(九、 深入绘制原理)》

Flutter 中 Widget 不可变，每次保持在一帧，如果**发生改变是通过** State **实现跨帧状态保存**，而**真实完成布局和绘制数组的是** RenderObject ，Element 充当两者的桥梁， State **就是保存在** Element **中。**

Flutter 中的 BuildContext **只是接口，而** Element **实现了它。**

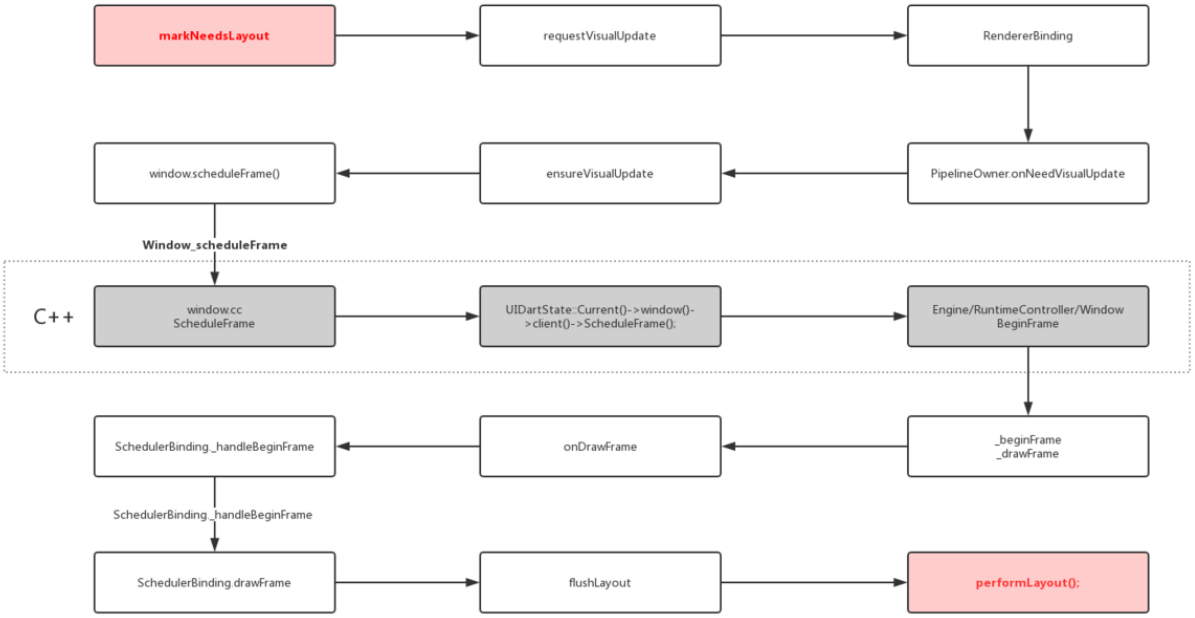
Flutter 中 setState **其实是调用了** markNeedsBuild ，该方法内部**标记此** Element **为** Dirty ，然后在下一帧 WidgetsBinding.drawFrame 才会被绘制，这可以看出 setState **并不是立即生效的。**

Flutter 中 RenderObject 在 attch/layout 之后会通过 markNeedsPaint(); 使得页面重绘，流程大概如下：



通过 isRepaintBoundary 往上确定了更新区域，通过 requestVisualUpdate 方法触发更新往下绘制。

正常情况 RenderObject 的布局相关方法调用顺序是 ： layout ->performResize ->performLayout ->markNeedsPaint，但是用户一般不会直接调用 layout，而是**通过** markNeedsLayout ，具体流程如下：



Flutter 中一般 **json** 数据从 `String` 转为 `Object` 的过程中都需要先经过 `Map` 类型。

Flutter 中 `InheritedWidget` 一般用于**状态共享**，如 `Theme` 、`Localizations` 、 `MediaQuery` 等，都是通过它实现共享状态，这样我们可以通过 `context` 去获取共享的状态，比如 `ThemeData theme = Theme.of(context);`

在 `Element` 的 `inheritFromWidgetOfExactType` 方法实现里，有一个 `Map<Type, InheritedElement> _inheritedWidgets` 的对象。

`_inheritedWidgets` 一般情况下是空的，只有当父控件是 `InheritedWidget` 或者本身是 `InheritedWidgets` 时才会有被初始化，而当父控件是 `InheritedWidget` 时，这个 `Map` 会被一级一级往下传递与合并 。

所以当我们通过 context 调用

inheritFromWidgetOfExactType 时，就可以往上查找到父

控件的 Widget 。

Flutter 中默认主要通过 runtimeType 和 key 判断更新：

```
static bool canUpdate(Widget oldWidget, Widget newWidget) {
return oldWidget.runtimeType == newWidget.runtimeType
&& oldWidget.key == newWidget.key;
  }
}
```

4.Flutter 中的生命周期

initState() 表示当前 State 将和一个 BuildContext 产生关联，但是此时 BuildContext 没有完全装载完成，如果你需要在该方法中获取 BuildContext ，可以 new Future.delayed(const Duration(seconds: 0, (){//context}); 一下。

didChangeDependencies() 在 initState() 之后调用，当 State 对象的依赖关系发生变化时，该方法被调用，初始化时也会调用。

deactivate() 当 State 被暂时从视图树中移除时，会调用这个方法，同时页面切换时，也会调用。

dispose() Widget 销毁了，在调用这个方法之前，总会先调用 deactivate()。

didUpdateWidge 当 widget 状态发生变化时，会调用。

通过 StreamBuilder 和 FutureBuilder 我们可以快速使用 Stream 和 Future 快速构建我们的异步控件: 《Flutter 完整开发实战详解(十一、全面深入理解 Stream)》

Flutter 中 runApp 启动入口其实是一个 WidgetsFlutterBinding ，它主要是通过 BindingBase 的子类 GestureBinding 、ServicesBinding 、SchedulerBinding 、PaintingBinding 、SemanticsBinding 、RendererBinding 、WidgetsBinding 等，通过 mixins 的组合而成的。

Flutter 中的 Dart 的线程是以**事件循环和消息队列**的形式存在，包含两个任务队列，**一个是 microtask 内部队列，一个是 event 外部队列，而 microtask 的优先级又高于 event 。**

因为 microtask 的优先级又高于 event， 同时会阻塞 event 队列，所以如果 microtask 太多就可能会对触摸、绘制等外部事件造成阻塞卡顿哦。

Flutter 中存在**四大线程，分别为** UI Runner、GPU Runner、IO Runner， Platform Runner **（原生主线程）**，同时在 Flutter 中可以通过 isolate 或者 compute 执行真正的跨线程异步操作。

5.PlatformView

Flutter 中通过 PlatformView 可以嵌套原生 View 到 Flutter UI 中，这里面其实是使用了 Presentation + VirtualDisplay + Surface 等实现的，大致原理就是：

使用了类似副屏显示的技术，VirtualDisplay 类代表一个虚拟显示器，调用 DisplayManager 的 createVirtualDisplay() 方法，将虚拟显示器的内容渲染在一个 Surface 控件上，然后将 Surface 的 id 通知给 Dart，让 engine 绘制时，在内存中找到对应的 Surface 画面内存数据，然后绘制出来。em... **实时控件截图渲染显示技术。**

Flutter 的 Debug 下是 JIT 模式，release 下是 AOT 模式。

Flutter 中可以通过 mixins AutomaticKeepAliveClientMixin ，然后重写 wantKeepAlive 保持住页面，记得在被保持住的页面 build 中调用 super.build 。（因为 mixins 特性）。

Flutter 手势事件主要是通过竞技判断的：

主要有 hitTest 把所有需要处理的控件对应的 RenderObject ， 从 child 到 parent 全部组合成列表，从最里面一直添加到最外层。

然后从队列头的 child 开始 for 循环执行 handleEvent 方法，执行 handleEvent 的过程不会被拦截打断。

一般情况下 Down 事件不会决出胜利者，大部分时候是在 MOVE 或者 UP 的时候才会决出胜利者。

竞技场关闭时只有一个的就直接胜出响应，没有胜利者就拿排在队列第一个强制胜利响应。

同时还有 `didExceedDeadline` 处理按住时的 Down 事件额外处理，同时手势处理一般在 `GestureRecognizer` 的子类进行。

> 更多详细请查看：《Flutter 完整开发实战详解(十三、全面深入触摸和滑动原理)》
>
> Flutter 中 `ListView` 滑动其实都是通过改变 `ViewPort` 中的 `child` 布局来实现显示的。
>
> 常用状态管理的:目前有 `scope_model` 、`flutter_redux` 、`fish_redux` 、`bloc + Stream` 等几种模式，具体可见：《Flutter 完整开发实战详解(十二、全面深入理解状态管理设计)》

6.Platform Channel

Flutter 中可以通过 `Platform Channel` 让 Dart 代码和原生代码通信的：

- `BasicMessageChannel` ：用于传递字符串和半结构化的信息。
- `MethodChannel` ：用于传递方法调用（method invocation）。
- `EventChanne l`: 用于数据流（event streams）的通信。

同时 `Platform Channel` **并非是线程安全的**，更多详细可查阅闲鱼技术的《深入理解 Flutter Platform Channel》

其中基础数据类型映射如下：

7.Android 启动页

Android 中 `Flutter` 默认启动时会在 `FlutterActivityDelegate.java` 中读取 AndroidManifset.xml 内 `meta-data` 标签，其中 `io.flutter.app.android.SplashScreenUntilFirstFrame` 标志位如果为 ture ，就会启动 Splash 画面效果（类似 IOS 的启动页面）。

启动时原生代码会读取 android.R.attr.windowBackground 得到指定的 Drawable ， 用于显示启动闪屏效果，之后并且通过 flutterView.addFirstFrameListener，在 onFirstFrame 中移除闪屏。

8. Flutter 和 Dart 的关系是什么？

Flutter 是一个使用 Dart 语言开发的跨平台移动 UI 框架，通过自建绘制引擎，能高性能、高保真地进行移动开发。Dart 囊括了多数编程语言的优点，它更符合 Flutter 构建界面的方式。

9. Widget 和 element 和 RenderObject 之间的关系？

- Widget 是用户界面的一部分,并且是不可变的。
- Element 是在树中特定位置 Widget 的实例。
- RenderObject 是渲染树中的一个对象，它的层次结构是渲染库的核心。

Widget 会被 inflate（填充）到 Element，并由 Element 管理底层渲染树。Widget 并不会直接管理状态及渲染,而是通过 State 这个对象来管理状态。Flutter 创建 Element 的可见树，相对于 Widget 来说，是可变的，通常界面开发中，我们不用直接操作 Element,而是由框架层实现内部逻辑。就如一个 UI 视图树中，可能包含有多个 TextWidget(Widget 被使用多次)，但是放在内部视图树的视角，这些 TextWidget 都是填充到一个个独立的 Element 中。Element 会持有 renderObject 和 widget 的实例。记住，Widget 只是一个配置，RenderObject 负责管理布局、绘制等操作。

在第一次创建 Widget 的时候，会对应创建一个 Element， 然后将该元素插入树中。如果之后 Widget 发生了变化，则将其与旧的 Widget 进行比较，并且相应地更新 Element。重要的是，Element 不会被重建，只是更新而已。

10. mixin extends implement 之间的关系?

继承（关键字 extends）、混入 mixins（关键字 with）、接口实现（关键字 implements）。这三者可以同时存在，前后顺序是 extends -> mixins -> implements。

Flutter 中的继承是单继承，子类重写超类的方法要用@Override，子类调用超类的方法要用 super。

在 Flutter 中，Mixins 是一种在多个类层次结构中复用类代码的方法。mixins 的对象是类，mixins 绝不是继承，也不是接口，而是一种全新的特性，可以 mixins 多个类，mixins 的使用需要满足一定条件。

11. 使用 mixins 的条件是什么？

因为 mixins 使用的条件，随着 Dart 版本一直在变，这里讲的是 Dart2.1 中使用 mixins 的条件：

mixins 类只能继承自 object
mixins 类不能有构造函数
一个类可以 mixins 多个 mixins 类
可以 mixins 多个类，不破坏 Flutter 的单继承

12. mixin 怎么指定异常类型？

on 关键字可用于指定异常类型。 on 只能用于被 mixins 标记的类，例如 mixins X on A，意思是要 mixins X 的话，得先接口实现或者继承 A。这里 A 可以是类，也可以是接口，但是在 mixins 的时候用法有区别.

on 一个类：

```
class A {
void a(){
    print("a");
  }
}


mixin X on A{
void x(){
    print("x");
  }
}

class mixinsX extends A with X{
}
```

on 的是一个接口： 得首先实现这个接口，然后再用 mix

```
class A {
void a(){
   print("a");
  }
}

mixin X on A{
void x(){
   print("x");
  }
}
class implA implements A{
  @override
void a() {}
}
class mixinsX2 extends implA with X{
}
```

13. Flutter main future mirotask 的执行顺序?

普通代码都是同步执行的，结束后会开始检查 microtask 中是否有任务，若有则执行，执行完继续检查 microtask，直到 microtask 列队为空。最后会去执行 event 队列（future）。

14. Future 和 Isolate 有什么区别？

future 是异步编程，调用本身立即返回，并在稍后的某个时候执行完成时再获得返回结果。在普通代码中可以使用 await 等待一个异步调用结束。

isolate 是并发编程，Dartm 有并发时的共享状态，所有 Dart 代码都在 isolate 中运行，包括最初的 main()。每个 isolate 都有它自己的堆内存，意味着其中所有内存数据，包括全局数据，都仅对该 isolate 可见，它们之间的通信只能通过传递消息的机制完成，消息则通过端口(port)收发。isolate 只是一个概念，具体取决于如何实现，比如在 Dart VM 中一个 isolate 可能会是一个线程，在 Web 中可能会是一个 Web Worker。

15. Stream 与 Future 是什么关系？

Stream 和 Future 是 Dart 异步处理的核心 API。Future 表示稍后获得的一个数据，所有异步的操作的返回值都用 Future 来表示。但是 Future 只能表示一次异步获得的数据。而 Stream 表示多次异步获得的数据。比如界面上的按钮可能会被用户点击多次，所以按钮上的点击事件（onClick）就是一个 Stream 。简单地说，Future 将返回一个值，而 Stream 将返回多次值。Dart 中统一使用 Stream 处理异步事件流。Stream 和一般的集合类似，都是一组数据，只不过一个是异步推送，一个是同步拉取。

16. Stream 两种订阅模式？

Stream 有两种订阅模式：**单订阅(single)** 和 **多订阅（broadcast）**。单订阅就是只能有一个订阅者，而广播是可以有多个订阅者。这就有点类似于消息服务（Message Service）的处理模式。单订阅类似于点对点，在订阅者出现之前会持有数据，在订阅者出现之后就才转交给它。而广播类似于发布订阅模式，可以同时有多个订阅者，当有数据时就会传递给所有的订阅者，而不管当前是否已有订阅者存在。
Stream 默认处于单订阅模式，所以同一个 stream 上的 listen 和其它大多数方法只能调用一次，调用第二次就会报错。但 Stream 可以通过 transform() 方法（返回另一个 Stream）进行连续调用。通过 Stream.asBroadcastStream() 可以将一个单订阅模式的 Stream 转换成一个多订阅模式的 Stream，isBroadcast 属性可以判断当前 Stream 所处的模式。

17. await for 如何使用?

await for 是不断获取 stream 流中的数据，然后执行循环体中的操作。它一般用在直到 stream 什么时候完成，并且必须等待传递完成之后才能使用，不然就会一直阻塞。

```
Stream<String> stream = new Stream<String>.fromIterable(['不开心', '面试', '没', '过']);
main() async{
  print('上午被开水烫了脚');
  await for(String s in stream){
    print(s);
  }
  print('晚上还没吃饭');
}
```

18. Flutter 中的 Widget、State、Context 的核心概念？是为了解决什么问题？

Widget：在 Flutter 中，几乎所有东西都是 Widget。将一个 Widget 想象为一个可视化的组件（或与应用可视化方面交互的组件），当你需要构建与布局直接或间接相关的任何内容时，你正在使用 Widget。

Widget 树：Widget 以树结构进行组织。包含其他 Widget 的 widget 被称为父 Widget(或 widget 容器)。包含在父 widget 中的 widget 被称为子 Widget。

Context：仅仅是已创建的所有 Widget 树结构中的某个 Widget 的位置引用。简而言之，将 context 作为 widget 树的一部分，其中 context 所对应的 widget 被添加到此树中。一个 context 只从属于一个 widget，它和 widget 一样是链接在一起的，并且会形成一个 context 树。

State：定义了 StatefulWidget 实例的行为，它包含了用于"**交互/干预**"Widget 信息的行为和布局。应用于 State 的任何更改都会强制重建 Widget。

这些状态的引入，主要是为了解决多个部件之间的交互和部件自身状态的维护。

19. Widget 的两种类型是什么？

StatelessWidget：一旦创建就不关心任何变化，在下次构建之前都不会改变。它们除了依赖于自身的配置信息（在父节点构建时提供）外不再依赖于任何其他信息。比如典型的 Text、Row、Column、Container 等，都是 StatelessWidget。它的生命周期相当简单：初始化、通过 build()渲染。

StatefulWidget：在生命周期内，该类 Widget 所持有的数据可能会发生变化，这样的数据被称为 **State**，这些拥有动态内部数据的 Widget 被称为 StatefulWidget。比如复选框、Button 等。State 会与 Context 相关联，并且此关联是永久性的，State 对象将永远不会改变其 Context，即使可以在树结构周围移动，也仍将与该 context 相关联。当 **state 与 context 关联时**，state 被视为**已挂载**。StatefulWidget 由两部分组成，在初始化时必须要在 createState()时初始化一个与之相关的 State 对象。

20. State 对象的初始化流程？

initState()：一旦 State 对象被创建，initState 方法是第一个（构造函数之后）被调用的方法。可通过重写来执行额外的初始化，如初始化动画、控制器等。重写该方法时，应该首先调用 super.initState()。在 initState 中，无法真正使用 context，因为框架还没有完全将其与 state 关联。initState 在该 State 对象的生命周期内将不会再次调用。

didChangeDependencies()：这是第二个被调用的方法。在这一阶段，context 已经可用。如果你的 Widget 链接到了一个 InheritedWidget 并且/或者你需要初始化一些 listeners（基于 context），通常会重写该方法。

build(BuildContext context)：此方法在 didChangeDependencies()、didUpdateWidget()之后被调用。每次 State 对象更新（或当 InheritedWidget 有新的通知时）都会调用该方法！我们一般都在 build 中来编写真正的功能代码。为了强制重建，可以在需要的时候调用 setState((){...})方法。

dispose()：此方法在 Widget 被废弃时调用。可重写该方法来执行一些清理操作（如解除 listeners），并在此之后立即调用 super.dispose()。

21. Widget 唯一标识 Key 有那几种？

在 flutter 中，每个 widget 都是被唯一标识的。这个唯一标识在 build 或 rendering 阶段由框架定义。该标识对应于可选的 Key 参数，如果省略，Flutter 将会自动生成一个。

在 flutter 中，主要有 4 种类型的 Key：GlobalKey（确保生成的 Key 在整个应用中唯一，是很昂贵的，允许 element 在树周围移动或变更父节点而不会丢失状态）、LocalKey、UniqueKey、ObjectKey。

22. 什么是 Navigator? MaterialApp 做了什么？

Navigator 是在 Flutter 中负责管理维护页面堆栈的导航器。MaterialApp 在需要的时候，会自动为我们创建 Navigator。Navigator.of(context)，会使用 context 来向上遍历 Element 树，找到 MaterialApp 提供的_NavigatorState 再调用其 push/pop 方法完成导航操作。

23.flutter 与 React Native 有什么不同？

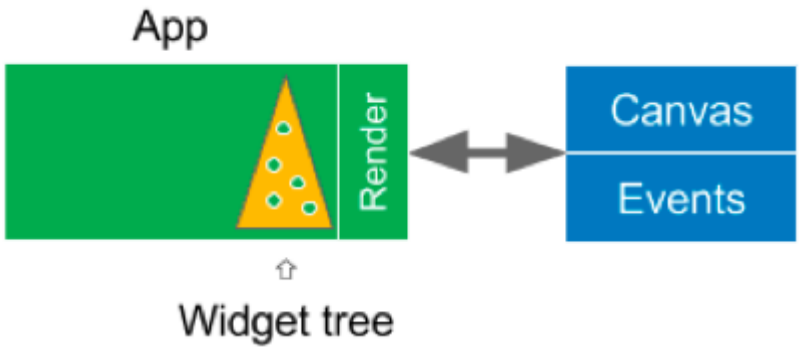
React Native 利用 JavaScript 桥将其小部件转换为 OEM 小部件。而且由于它不断地进行这种转换（比较和更新周期），因此会产生瓶颈并导致性能下降。

虽然仍然使用反应式视图的优势，但 Flutter 并没有使用这种桥将其自己的小部件转换为 OEM 小部件。除了快速和流畅的 UI 性能和可预测性之外，作为此项的另一个优势，您在 Android KitKat 设备上看到的内容与您在 Android Pie 上获得的内容相同。这种兼容性是显而易见的，因为 Flutter 不使用 OEM 小部件，并且不受不同 Android 版本之间的 UI / UX 更改的影响。

24.为什么说 flutter 是原生的

 Flutter 应用程序与平台的交互

Flutter 使用名为 Skia 的图形引擎在应用程序端执行所有 UI 呈现。这意味着它不依赖于平台提供的 OEM 小部件。它只需要平台的画布来绘制自己的渲染。这确保了可预测性和开发人员对小部件和布局的完全控制。



Flutter 内部小部件树

除此之外，Flutter 将其结构保持为小部件树。顺便说一下，Flutter 中的几乎所有东西都是一个小部件，它使您能够在小部件内部的小部件结构中构建您的应用程序。此内部树结构允许 Skia 仅呈现需要更新的小部件，并从缓存中检索未更改的甚至移动的小部件。

25.讲一下 flutter 的几个特点/优缺点

Dart 是用于开发 Flutter 应用程序的面向对象，垃圾收集的编程语言。它也是由谷歌创建的，但它是开源的，因此它在 Google 内外都有社区。

除了 Google 的起源之外，Dart 还被选为 Flutter 的语言，原因如下：它是极少数可以同时编译 AOT（提前）和 JIT（即时）的语言之一。

在应用程序开发过程中使用 `JIT` 编译，因为它可以通过动态编译代码来实现热重新加载（我将在下一个问题中详细讨论）和快速开发周期。
完成开发并准备发布后，将使用 `AOT` 编译。然后将代码 AOT 编译为本机代码，从而实现应用程序的快速启动和高性能执行。
就个人而言，我对 Dart 的经验是，如果您是熟悉 Java 或类似语言的开发人员，只需要几天的时间就可以习惯它。因此，如果您是 Android 开发人员，那么这种语言的学习曲线应该非常低。

凭借其干净但灵活的语法，Dart 可以被识别为仅包含任何高级编程语言中最需要的功能的语言。

26.什么是 ScopedModel / BLoC 模式？

ScopedModel 和 BLoC（业务逻辑组件）是常见的 Flutter 应用程序架构模式，可帮助将业务逻辑与 UI 代码分离，并使用更少的状态窗口小部件

27.什么是 stateWidget 和 statelessWidget？

内边距 margin 和外边距边距 padding

```
body: Center(
  child: Container(
    child:  new Text("hello zzl ",
    style: TextStyle(
      fontSize: 40.0,
    ),
      textAlign: TextAlign.center,
    ),
    alignment: Alignment.topCenter,
    width: 500.0,
    height: 400.0,
    color: Colors.blue,
    padding: const EdgeInsets.all(50.0),//外边距
   // margin: const EdgeInsets.all(100.0),//内边距
  ),
  ),
```

填充控件 Padding

Padding 的布局分为两种情况：

当 child 为空的时候，会产生一个宽为 left+right，高为 top+bottom 的区域；
当 child 不为空的时候，Padding 会将布局约束传递给 child，根据设置的 padding 属性，缩

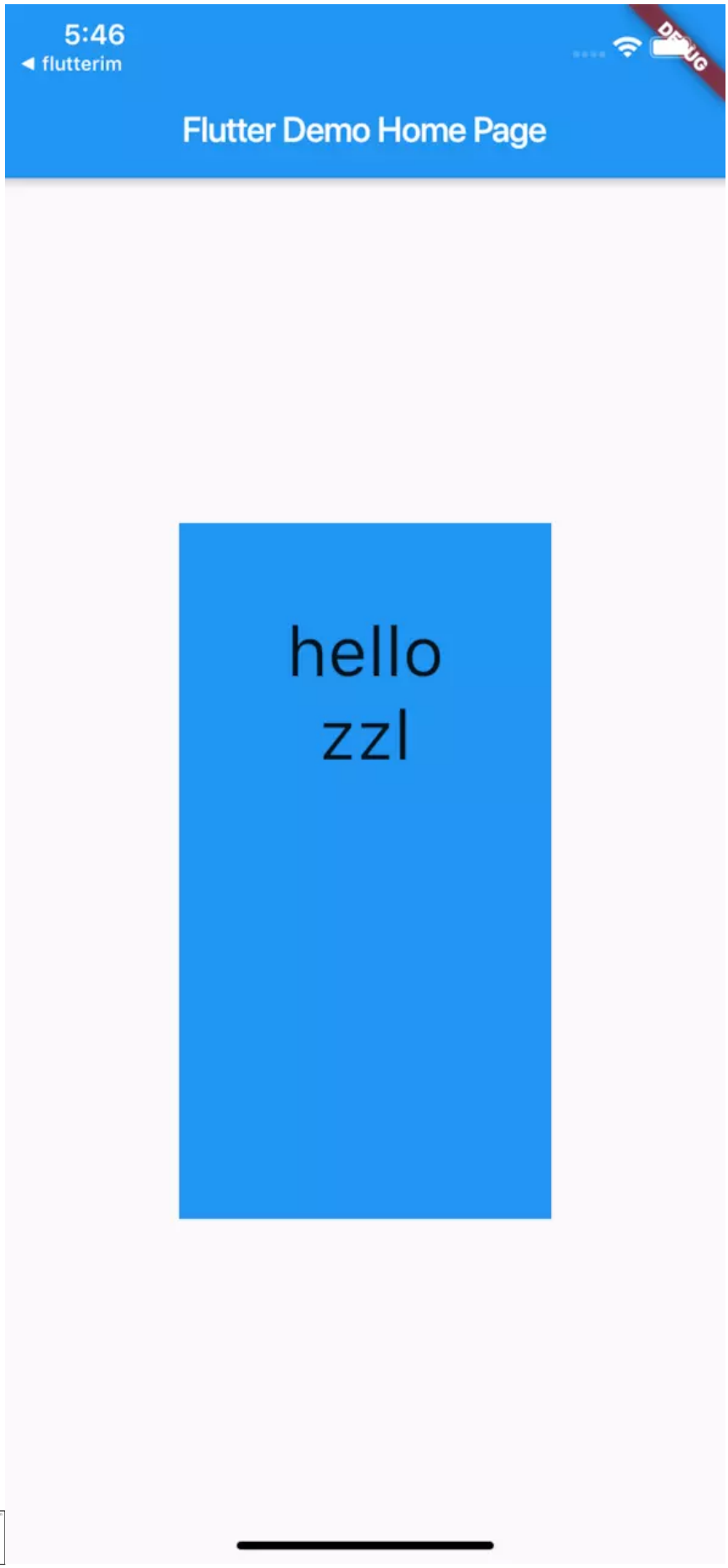
小 child 的布局尺寸。然后 Padding 将自己调整到 child 设置了 padding 属性的尺寸，在 child 周围创建空白区域。

```
import 'package:flutter/material.dart';

class PaddingDemo extends StatelessWidget{

  @override

  Widget build(BuildContext context) {

    return new Scaffold(

      appBar: new AppBar(

        title:new Text("padding 填充控件"),

      ),

      body: new Padding(

          padding: const EdgeInsets.all(8.0),

        child: new Image.asset("images/hua3.png"),

      ),

    );

  }

}

void main(){

  runApp(new MaterialApp(

    title: "padding 填充控件",

    home: new PaddingDemo(),

  ));
```

28.如何在 Flutter 中定义边距和填充？

内边距 margin 和外边距边距 padding

```
body: Center(
  child: Container(
    child:  new Text("hello zzl ",
    style: TextStyle(
      fontSize: 40.0,
    ),
      textAlign: TextAlign.center,
    ),
    alignment: Alignment.topCenter,
    width: 500.0,
    height: 400.0,
    color: Colors.blue,
    padding: const EdgeInsets.all(50.0),//外边距
   // margin: const EdgeInsets.all(100.0),//内边距
  ),
  ),
```

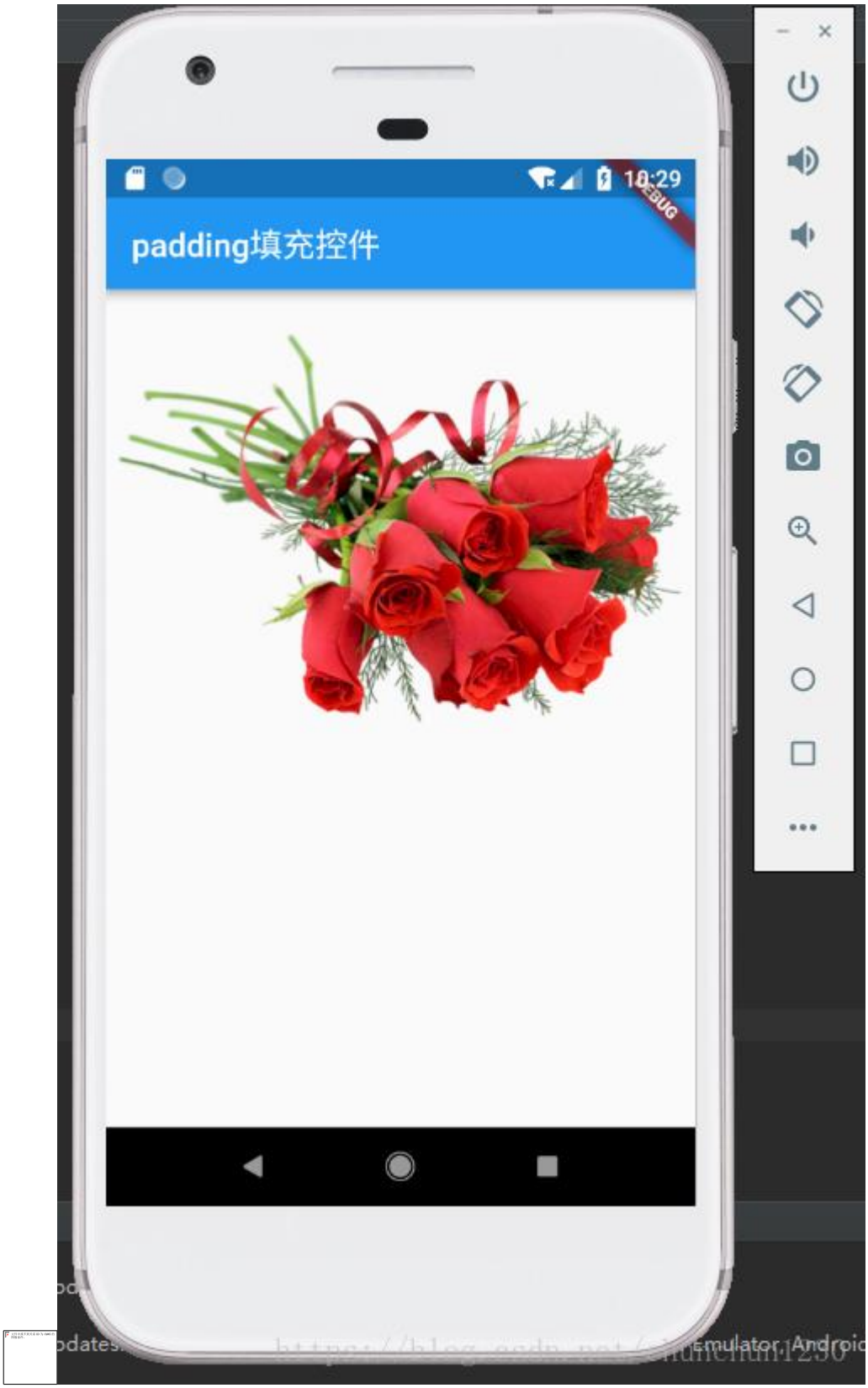

5:46
flutterim
Flutter Demo Home Page
hello
zzl

填充控件 Padding

Padding 的布局分为两种情况：

当 child 为空的时候，会产生一个宽为 left+right，高为 top+bottom 的区域；
当 child 不为空的时候，Padding 会将布局约束传递给 child，根据设置的 padding 属性，缩小 child 的布局尺寸。然后 Padding 将自己调整到 child 设置了 padding 属性的尺寸，在 child 周围创建空白区域。

```
import 'package:flutter/material.dart';

class PaddingDemo extends StatelessWidget{

  @override

  Widget build(BuildContext context) {

    return new Scaffold(

      appBar: new AppBar(

        title:new Text("padding 填充控件"),

      ),

      body: new Padding(

          padding: const EdgeInsets.all(8.0),

        child: new Image.asset("images/hua3.png"),

      ),

    );

  }

}

void main(){

  runApp(new MaterialApp(

    title: "padding 填充控件",

    home: new PaddingDemo(),

  ));
```


10:29
DEBUG
padding填充控件

29.谈一下 flutter state 的生命周期

State 的生命周期

从上面的例子中可以看到， 会要求提供一个含有视图树的 。

既然 能够控制一个视图的状态，那它肯定会有一系列的生命周期。

上图就是 State 的生命周期图。

1. `StatefulWidget.createState()`

Framework 调用会通过调用 `StatefulWidget.createState()` 来创建一个 State。

1. `initState()`

新创建的 State 会和一个 产生关联，此时认为 State 已经被安装好了，`initState()` 函数将会被调用。

通常，我们可以重写这个函数，进行初始化操作。

1. `didChangeDependencies()`

在 `initState()` 调用结束后，这个函数会被调用。

事实上，当 State 对象的依赖关系发生变化时，这个函数总会被 Framework 调用。

1. `build()`

经过以上步骤，系统认为一个 State 已经准备好了，就会调用 `build()` 来构建视图。

我们需要在这个函数中，返回一个 Widget。

1. `deactivate()`

当 State 被暂时从视图树中移除时，会调用这个函数。

页面切换时，也会调用它，因为此时 State 在视图树中的位置发生了变化，需要先暂时移除后添加。

⚠️注意，重写的时候必须要调用 `super.deactivate()`。

1. `dispose()`

当 State 被永久的从视图树中移除，Framework 会调用该函数。

在销毁前触发，我们可以在这里进行最终的资源释放。

在调用这个函数之前，总会先调用 `deactivate()`。

⚠️注意，重写的时候必须要调用 `super.dispose()`。

1. `didUpdateWidget(covariant T oldWidget)`

当 widget 的配置发生变化时，会调用这个函数。

比如， 的时候就会调用这个函数。

这个函数调用后，会调用 `build()`。

1. `setState()`

当我需要更新 State 的视图时，需要手动调用这个函数，它会触发 `build()` 。

31.Flutter 和 RN 的对比。

32.说一下 Hot Reload，Hot Restart，热更新三者的区别和原理。

33.Flutter 是如何做到一套 Dart 代码可以编译运行在 Android 和 iOS 平台的？所以说具体的原理。

34.Flutter 不具备反射，如果要使用反射，你应该如何使用？说一下大概的思路。

35.Flutter 在不使用 WebView 和 JS 方案的情况下。如何做到热更新？说一下大概思路。

36.如何让 Flutter 编译出来的 APP 的包大小尽可能的变小？

37.我们这个项目时一个综合系统的老项目，里面有 Android，iOS，还有 Web 代码，是一个混合开发的项目，现在需要迁移到 Flutter，加入你加入团队做这个项目的迁移工作，你觉得这个项目如何工程化、容器化以及架构演变应该从哪些维度思考？

38.APP 启动速度以及页面加载速度一直是我们比较关心的一个问题，特别是混合开发项目，谈谈你对 Flutter 渲染优化有哪些见解？

39.谈谈 Flutter 的内存回收管理机制，以及你平时是怎么处理内存的？内存泄漏和内存溢出你是怎么解决的？

40.再问一个简单一点的，你是如何把控混合项目开发时的生命周期（比如类似安卓的 onCreate、onResume 这种）和路由管理的？

41.Flutter for web 和 Flutter1.9 推出的 Flutter Web 有何本质上的区别？

42.谈谈你认为的 Flutter Web 应该如何改进？哪些内容可以改造之后可以用于平时的 Web 开发。谈谈你的改造方案。

43.谈谈如何打造低延迟的视频直播？为什么这样用？

44. StatefulWidget 的生命周期

- `initState()`：Widget 初始化当前 State，在当前方法中是不能获取到 Context 的，如想获取，可以试试 `Future.delayed()`
- `didChangeDependencies()`：在 `initState()` 后调用，State 对象依赖关系发生变化的时候也会调用。
- `deactivate()`：当 State 被暂时从视图树中移除时会调用这个方法，页面切换时也会调用该方法，和 Android 里的 `onPause` 差不多。
- `dispose()`：Widget 销毁时调用。
- `didUpdateWidget`：Widget 状态发生变化的时候调用。

借用 CoorChice 文章 里的一张图：

45. Flutter 如何与 Android iOS 通信？

Flutter 通过 `PlatformChannel` 与原生进行交互，其中 `PlatformChannel` 分为三种：

1. BasicMessageChannel：用于传递字符串和半结构化的信息。
2. MethodChannel：用于传递方法调用。Flutter 主动调用 Native 的方法，并获取相应的返回值。
3. EventChannel：用于数据流（event streams）的通信。

具体可以查看 闲鱼技术：深入理解 Flutter Platform Channel。

46. 什么是 Widgets、RenderObjects 和 Elements？

- `Widget` 仅用于存储渲染所需要的信息。
- `RenderObject` 负责管理布局、绘制等操作。
- `Element` 才是这颗巨大的控件树上的实体。

具体可以查看 [译] Flutter，什么是 Widgets、RenderObjects 和 Elements？

47. 说一下什么是状态管理，为什么需要它？

首先状态其实是一个概念上的东西，区分全局状态和局部状态。

局部状态比如说一个控件中输入的信息，全局状态比如是登陆后从后台请求回来的 userId。

当全局状态越来越多，多个页面共享一个状态时，我们就需要管理它。

常用的状态管理有：

- ScopedModel
- BLoC
- Redux / FishRedux
- Provider

48. 说一下 BLoC 模式？

具体可以查看： Vadaski - Flutter | 状态管理探索篇——BLoC(三)

这里引用一部分：

BLoC 是一种利用 reactive programming 方式构建应用的方法，这是一个由流构成的完全异步的世界。

49. 如何统一管理错误页面？

我们都知道，如果在 Flutter 当中出错的话，那就是一片红。

可以使用 `ErrorWidget.builder` 来自定义一个 Widget 就 ok 了。

具体可以看一下 小德 - 教你自定义 Flutter 错误页面