#### 1. dart是值传递还是引用传递

首先给个结论，dart是**值传递**。

**之前把引用传递理解错了，给各位读者报个歉，同时也感谢评论区的指正**

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的话，main函数中的a值是不会发生变化的。 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，如果是值传递，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) {

//相当于dart中main函数初始化

Test a = new Test();

//相当于setValue()中的s，并把参数代表的内存地址赋值给b

Test b = a;

//s指向一个新对象，即为dart中的s += 1

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。

偷偷说一句，我觉得面试官这个地方面试的不好，这种细节问题，如果不是遇到什么bug，业务忙的时候是没时间注意这个的，面试官可以把这两种情况展示下，然后问面试者原因是什么。。然后我就能回答出来了。。哭唧唧。。

#### 2. 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。

#### 3. 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。。。

#### 4. mixin extends implement之间的关系

这部分可以参考掘金的小德大佬的文章，高产似那啥。。

* [Flutter Dart mixins 探究](https://juejin.im/post/5c44382d51882523f0261bb5" \t "_blank)
* [Flutter Dart语法(1):extends 、 implements 、 with的用法与区别](https://juejin.im/post/5c4881dae51d45098e4d96cf" \t "_blank)

#### 6. Future和microtask执行顺序

同样参考小德的文章

* [深入了解Flutter的isolate(1) ---- 事件循环(event loop)及代码运行顺序](https://juejin.im/post/5c338bdb6fb9a04a01647eb9" \t "_blank)
* [深入了解Flutter的isolate(2) --- 创建自己的isolate](https://juejin.im/post/5c34ce1351882524b407232a" \t "_blank)
* [深入了解Flutter的isolate(3) --- Flutter的thread model（线程模型）](https://juejin.im/post/5c3844fae51d4551ec60988e" \t "_blank)
* [深入了解Flutter的isolate(4) --- 使用Compute写isolates](https://juejin.im/post/5c3844fae51d4551ec60988e" \t "_blank)

#### 7. dart中..是什么

级联符号 .. 可以让你连续操作相同的对象，不单可以连续地调用函数，还可以连续地访问方法，这样做可以避免创建临时变量，从而写出更流畅的代码，流式编程更符合现代编程习惯和编程风格:

main(){

Tree tree = new Tree(1);

tree..test1 = 1..test2 =5;

print(tree.test1);

print(tree.test2);

}

class Tree{

int value;

int test1 = 2;

int test2 = 3;

Tree(int a){

this.value = a;

}

}

#### 8. await for使用

先来一段官方文档

## await-for

As every Dart programmer knows, the **for-in** loop plays well with iterables. Similarly, the **await-for** loop is designed to play well with streams. Given a stream, one can loop over its values: Every time an element is added to the stream, the loop body is run. After each iteration, the function enclosing the loop suspends until the next element is available or the stream is done. Just like **await** expressions, **await-for** loops can only appear inside asynchronous functions.

大概意思就是await for是不断获取stream流中的数据，然后执行循环体中的操作。

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');

}复制代码

输出为

上午被开水烫了脚

不开心

面试

没

过

晚上还没吃饭

await for 和 listen的作用很相似，都是获取流中数据然后输出，但是正如await for中的await所示，如果stream没有传递完成，就会一直阻塞在这个位置，上面没吃饭是最后输出的，下面给个listen的实例，一看就懂。

Stream<String> stream = new Stream<String>.fromIterable(['不开心', '面试', '没', '过']);main(){

print('上午被开水烫了脚');

stream.listen((s) { print(s); });

print('晚上还没吃饭');

}

输出为

上午被开水烫了脚

晚上还没吃饭

不开心

面试

没

过

所以await for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，造成类似于Android ANR的问题。

#### 总结

其实面试官还是很nice的，第一次见到活的大佬。。大佬对flutter和dart的研究真的很深入，远不是我这种只会调api的人可以比拟的。 主要还是我一个半月没使用过flutter了，然后之前问其他大佬要不要准备Flutter，大佬们说不用，以前看的很多东西都忘的差不多了。 哎，还是自己准备不充分，或者开始大佬问我的时候直接回答忘得差不多了，应该就能过了吧。