# 1.4 Dart语言简介

在之前我们已经介绍过Dart语言的相关特性，读者可以翻看一下，如果读者已经熟悉Dart语法，可以跳过本节，如果你还不了解Dart，也不用担心，按照笔者经验，如果你有过其他编程语言经验（尤其是Java和JavaScript）的话会非常容易上手Dart。当然，如果你是iOS开发者，也不用担心，Dart中也有一些与Swift比较相似的特性，如命名参数等，笔者当时学习Dart时，只是花了一个小时，看完Dart官网的Language Tour，就开始动手写Flutter了。

在笔者看来，Dart的设计目标应该是同时借鉴了Java和JavaScript。Dart在静态语法方面和Java非常相似，如类型定义、函数声明、泛型等，而在动态特性方面又和JavaScript很像，如函数式特性、异步支持等。除了融合Java和JavaScript语言之所长之外，Dart也具有一些其它具有表现力的语法，如可选命名参数、..（级联运算符）和?.（条件成员访问运算符）以及??（判空赋值运算符）。其实，对编程语言了解比较多的读者会发现，在Dart中其实看到的不仅有Java和JavaScript的影子，它还具有其它编程语言中的身影，如命名参数在Objective-C和Swift中早就很普遍，而??操作符在PHP 7.0语法中就已经存在了，因此我们可以看到Google对Dart语言给予厚望，是想把Dart打造成一门集百家之所长的编程语言。

接下来，我们先对Dart语法做一个简单的介绍，然后再将Dart与JavaScript和Java做一个简要的对比，方便读者更好的理解。

注意：由于本书并非专门介绍Dart语言的书籍，所以本章主要会介绍一下在Flutter开发中常用的语法特性，如果想更多了解Dart，读者可以去Dart官网学习，现在互联网上Dart相关资料已经很多了。另外Dart 2.0已经正式发布，所以本书所有示例均采用Dart 2.0语法。

## 1.4.1 变量声明

1. **var**

* 类似于JavaScript中的var，它可以接收任何类型的变量，但最大的不同是Dart中var变量一旦赋值，类型便会确定，则不能再改变其类型，如：
* var t;  
  t = "hi world";  
  // 下面代码在dart中会报错，因为变量t的类型已经确定为String，  
  // 类型一旦确定后则不能再更改其类型。  
  t = 1000;
* 上面的代码在JavaScript是没有问题的，前端开发者需要注意一下，之所以有此差异是因为Dart本身是一个强类型语言，任何变量都是有确定类型的，在Dart中，当用var声明一个变量后，Dart在编译时会根据第一次赋值数据的类型来推断其类型，编译结束后其类型就已经被确定，而JavaScript是纯粹的弱类型脚本语言，var只是变量的声明方式而已。

1. **dynamic**和**Object**

* Object 是Dart所有对象的根基类，也就是说所有类型都是Object的子类(包括Function和Null)，所以任何类型的数据都可以赋值给Object声明的对象. dynamic与var一样都是关键词,声明的变量可以赋值任意对象。 而dynamic与Object相同之处在于,他们声明的变量可以在后期改变赋值类型。
* dynamic t;  
  Object x;  
  t = "hi world";  
  x = 'Hello Object';  
  //下面代码没有问题  
  t = 1000;  
  x = 1000;
* dynamic与Object不同的是,dynamic声明的对象编译器会提供所有可能的组合, 而Object声明的对象只能使用Object的属性与方法, 否则编译器会报错。如:
* dynamic a;  
   Object b;  
   main() {  
   a = "";  
   b = "";  
   printLengths();  
   }   
    
   printLengths() {  
   // no warning  
   print(a.length);  
   // warning:  
   // The getter 'length' is not defined for the class 'Object'  
   print(b.length);  
   }
* 变量a不会报错, 变量b编译器会报错
* dynamic的这个特性与Objective-C中的id作用很像. dynamic的这个特点使得我们在使用它时需要格外注意,这很容易引入一个运行时错误.

1. **final**和**const**

* 如果您从未打算更改一个变量，那么使用 final 或 const，不是var，也不是一个类型。 一个 final 变量只能被设置一次，两者区别在于：const 变量是一个编译时常量，final变量在第一次使用时被初始化。被final或者const修饰的变量，变量类型可以省略，如：
* //可以省略String这个类型声明  
  final str = "hi world";  
  //final String str = "hi world";   
  const str1 = "hi world";  
  //const String str1 = "hi world";

## 1.4.2 函数

Dart是一种真正的面向对象的语言，所以即使是函数也是对象，并且有一个类型**Function**。这意味着函数可以赋值给变量或作为参数传递给其他函数，这是函数式编程的典型特征。

1. 函数声明

* bool isNoble(int atomicNumber) {  
   return \_nobleGases[atomicNumber] != null;  
  }
* Dart函数声明如果没有显式声明返回值类型时会默认当做dynamic处理，注意，函数返回值没有类型推断：
* typedef bool CALLBACK();  
    
  //不指定返回类型，此时默认为dynamic，不是bool  
  isNoble(int atomicNumber) {  
   return \_nobleGases[atomicNumber] != null;  
  }  
    
  void test(CALLBACK cb){  
   print(cb());   
  }  
  //报错，isNoble不是bool类型  
  test(isNoble);

1. 对于只包含一个表达式的函数，可以使用简写语法

* bool isNoble (int atomicNumber)=> \_nobleGases [ atomicNumber ] ！= null ;

1. 函数作为变量

* var say = (str){  
   print(str);  
  };  
  say("hi world");

1. 函数作为参数传递

* void execute(var callback) {  
   callback();  
  }  
  execute(() => print("xxx"))

1. 可选的位置参数

* 包装一组函数参数，用[]标记为可选的位置参数，并放在参数列表的最后面：
* String say(String from, String msg, [String device]) {  
   var result = '$from says $msg';  
   if (device != null) {  
   result = '$result with a $device';  
   }  
   return result;  
  }
* 下面是一个不带可选参数调用这个函数的例子：
* say('Bob', 'Howdy'); //结果是： Bob says Howdy
* 下面是用第三个参数调用这个函数的例子：
* say('Bob', 'Howdy', 'smoke signal'); //结果是：Bob says Howdy with a smoke signal

1. 可选的命名参数

* 定义函数时，使用{param1, param2, …}，放在参数列表的最后面，用于指定命名参数。例如：
* //设置[bold]和[hidden]标志  
  void enableFlags({bool bold, bool hidden}) {  
   // ...   
  }
* 调用函数时，可以使用指定命名参数。例如：paramName: value
* enableFlags(bold: true, hidden: false);
* 可选命名参数在Flutter中使用非常多。
* **注意，不能同时使用可选的位置参数和可选的命名参数**

## 1.4.3 异步支持

Dart类库有非常多的返回Future或者Stream对象的函数。 这些函数被称为**异步函数**：它们只会在设置好一些耗时操作之后返回，比如像 IO操作。而不是等到这个操作完成。

async和await关键词支持了异步编程，允许您写出和同步代码很像的异步代码。

### Future

Future与JavaScript中的Promise非常相似，表示一个异步操作的最终完成（或失败）及其结果值的表示。简单来说，它就是用于处理异步操作的，异步处理成功了就执行成功的操作，异步处理失败了就捕获错误或者停止后续操作。一个Future只会对应一个结果，要么成功，要么失败。

由于本身功能较多，这里我们只介绍其常用的API及特性。还有，请记住，Future 的所有API的返回值仍然是一个Future对象，所以可以很方便的进行链式调用。

#### Future.then

为了方便示例，在本例中我们使用Future.delayed 创建了一个延时任务（实际场景会是一个真正的耗时任务，比如一次网络请求），即2秒后返回结果字符串“hi world!”，然后我们在then中接收异步结果并打印结果，代码如下：

Future.delayed(new Duration(seconds: 2),(){  
 return "hi world!";  
}).then((data){  
 print(data);  
});

#### Future.catchError

如果异步任务发生错误，我们可以在catchError中捕获错误，我们将上面示例改为：

Future.delayed(new Duration(seconds: 2),(){  
 //return "hi world!";  
 throw AssertionError("Error");   
}).then((data){  
 //执行成功会走到这里   
 print("success");  
}).catchError((e){  
 //执行失败会走到这里   
 print(e);  
});

在本示例中，我们在异步任务中抛出了一个异常，then的回调函数将不会被执行，取而代之的是 catchError回调函数将被调用；但是，并不是只有 catchError回调才能捕获错误，then方法还有一个可选参数onError，我们也可以它来捕获异常：

Future.delayed(new Duration(seconds: 2), () {  
 //return "hi world!";  
 throw AssertionError("Error");  
}).then((data) {  
 print("success");  
}, onError: (e) {  
 print(e);  
});

#### Future.whenComplete

有些时候，我们会遇到无论异步任务执行成功或失败都需要做一些事的场景，比如在网络请求前弹出加载对话框，在请求结束后关闭对话框。这种场景，有两种方法，第一种是分别在then或catch中关闭一下对话框，第二种就是使用Future的whenComplete回调，我们将上面示例改一下：

Future.delayed(new Duration(seconds: 2),(){  
 //return "hi world!";  
 throw AssertionError("Error");  
}).then((data){  
 //执行成功会走到这里   
 print(data);  
}).catchError((e){  
 //执行失败会走到这里   
 print(e);  
}).whenComplete((){  
 //无论成功或失败都会走到这里  
});

#### Future.wait

有些时候，我们需要等待多个异步任务都执行结束后才进行一些操作，比如我们有一个界面，需要先分别从两个网络接口获取数据，获取成功后，我们需要将两个接口数据进行特定的处理后再显示到UI界面上，应该怎么做？答案是Future.wait，它接受一个Future数组参数，只有数组中所有Future都执行成功后，才会触发then的成功回调，只要有一个Future执行失败，就会触发错误回调。下面，我们通过模拟Future.delayed 来模拟两个数据获取的异步任务，等两个异步任务都执行成功时，将两个异步任务的结果拼接打印出来，代码如下：

Future.wait([  
 // 2秒后返回结果   
 Future.delayed(new Duration(seconds: 2), () {  
 return "hello";  
 }),  
 // 4秒后返回结果   
 Future.delayed(new Duration(seconds: 4), () {  
 return " world";  
 })  
]).then((results){  
 print(results[0]+results[1]);  
}).catchError((e){  
 print(e);  
});

执行上面代码，4秒后你会在控制台中看到“hello world”。

### Async/await

Dart中的async/await 和JavaScript中的async/await功能和用法是一模一样的，如果你已经了解JavaScript中的async/await的用法，可以直接跳过本节。

#### 回调地狱(Callback Hell)

如果代码中有大量异步逻辑，并且出现大量异步任务依赖其它异步任务的结果时，必然会出现Future.then回调中套回调情况。举个例子，比如现在有个需求场景是用户先登录，登录成功后会获得用户ID，然后通过用户ID，再去请求用户个人信息，获取到用户个人信息后，为了使用方便，我们需要将其缓存在本地文件系统，代码如下：

//先分别定义各个异步任务  
Future<String> login(String userName, String pwd){  
 ...  
 //用户登录  
};  
Future<String> getUserInfo(String id){  
 ...  
 //获取用户信息   
};  
Future saveUserInfo(String userInfo){  
 ...  
 // 保存用户信息   
};

接下来，执行整个任务流：

login("alice","\*\*\*\*\*\*").then((id){  
 //登录成功后通过，id获取用户信息   
 getUserInfo(id).then((userInfo){  
 //获取用户信息后保存   
 saveUserInfo(userInfo).then((){  
 //保存用户信息，接下来执行其它操作  
 ...  
 });  
 });  
})

可以感受一下，如果业务逻辑中有大量异步依赖的情况，将会出现上面这种在回调里面套回调的情况，过多的嵌套会导致的代码可读性下降以及出错率提高，并且非常难维护，这个问题被形象的称为**回调地狱（Callback Hell）**。回调地狱问题在之前JavaScript中非常突出，也是JavaScript被吐槽最多的点，但随着ECMAScript6和ECMAScript7标准发布后，这个问题得到了非常好的解决，而解决回调地狱的两大神器正是ECMAScript6引入了Promise，以及ECMAScript7中引入的async/await。 而在Dart中几乎是完全平移了JavaScript中的这两者：Future相当于Promise，而async/await连名字都没改。接下来我们看看通过Future和async/await如何消除上面示例中的嵌套问题。

##### 使用Future消除Callback Hell

login("alice","\*\*\*\*\*\*").then((id){  
 return getUserInfo(id);  
}).then((userInfo){  
 return saveUserInfo(userInfo);  
}).then((e){  
 //执行接下来的操作   
}).catchError((e){  
 //错误处理   
 print(e);  
});

正如上文所述， *“Future 的所有API的返回值仍然是一个Future对象，所以可以很方便的进行链式调用”* ，如果在then中返回的是一个Future的话，该future会执行，执行结束后会触发后面的then回调，这样依次向下，就避免了层层嵌套。

##### 使用async/await消除callback hell

通过Future回调中再返回Future的方式虽然能避免层层嵌套，但是还是有一层回调，有没有一种方式能够让我们可以像写同步代码那样来执行异步任务而不使用回调的方式？答案是肯定的，这就要使用async/await了，下面我们先直接看代码，然后再解释，代码如下：

task() async {  
 try{  
 String id = await login("alice","\*\*\*\*\*\*");  
 String userInfo = await getUserInfo(id);  
 await saveUserInfo(userInfo);  
 //执行接下来的操作   
 } catch(e){  
 //错误处理   
 print(e);   
 }   
}

* async用来表示函数是异步的，定义的函数会返回一个Future对象，可以使用then方法添加回调函数。
* await 后面是一个Future，表示等待该异步任务完成，异步完成后才会往下走；await必须出现在 async 函数内部。

可以看到，我们通过async/await将一个异步流用同步的代码表示出来了。

其实，无论是在JavaScript还是Dart中，async/await都只是一个语法糖，编译器或解释器最终都会将其转化为一个Promise（Future）的调用链。

## 1.4.4 Stream

Stream 也是用于接收异步事件数据，和Future 不同的是，它可以接收多个异步操作的结果（成功或失败）。 也就是说，在执行异步任务时，可以通过多次触发成功或失败事件来传递结果数据或错误异常。 Stream 常用于会多次读取数据的异步任务场景，如网络内容下载、文件读写等。举个例子：

Stream.fromFutures([  
 // 1秒后返回结果  
 Future.delayed(new Duration(seconds: 1), () {  
 return "hello 1";  
 }),  
 // 抛出一个异常  
 Future.delayed(new Duration(seconds: 2),(){  
 throw AssertionError("Error");  
 }),  
 // 3秒后返回结果  
 Future.delayed(new Duration(seconds: 3), () {  
 return "hello 3";  
 })  
]).listen((data){  
 print(data);  
}, onError: (e){  
 print(e.message);  
},onDone: (){  
  
});

上面的代码依次会输出：

I/flutter (17666): hello 1  
I/flutter (17666): Error  
I/flutter (17666): hello 3

代码很简单，就不赘述了。

思考题：既然Stream可以接收多次事件，那能不能用Stream来实现一个订阅者模式的事件总线？

## 1.4.5 Dart和Java及JavaScript对比

通过上面介绍，相信你对Dart应该有了一个初步的印象，由于笔者平时也使用Java和JavaScript，下面笔者根据自己的经验，结合Java和JavaScript，谈一下自己的看法。

之所以将Dart与Java和JavaScript对比，是因为，这两者分别是强类型语言和弱类型语言的典型代表，并且Dart 语法中很多地方也都借鉴了Java和JavaScript。

### Dart vs Java

客观的来讲，Dart在语法层面确实比Java更有表现力；在VM层面，Dart VM在内存回收和吞吐量都进行了反复的优化，但具体的性能对比，笔者没有找到相关测试数据，但在笔者看来，只要Dart语言能流行，VM的性能就不用担心，毕竟Google在Go（没用VM但有GC）、JavaScript（v8）、Dalvik（Android上的Java VM）上已经有了很多技术积淀。值得注意的是Dart在Flutter中已经可以将GC做到10ms以内，所以Dart和Java相比，决胜因素并不会是在性能方面。而在语法层面，Dart要比Java更有表现力，最重要的是Dart对函数式编程支持要远强于Java(目前只停留在Lambda表达式)，而Dart目前真正的不足是**生态**，但笔者相信，随着Flutter的逐渐火热，会回过头来反推Dart生态加速发展，对于Dart来说，现在需要的是时间。

### Dart vs JavaScript

JavaScript的弱类型一直被抓短，所以TypeScript、CoffeeScript甚至是Facebook的flow（虽然并不能算JavaScript的一个超集，但也通过标注和打包工具提供了静态类型检查）才有市场。就笔者使用过的脚本语言中（笔者曾使用过Python、PHP），JavaScript无疑是**动态化**支持最好的脚本语言，比如在JavaScript中，可以给任何对象在任何时候动态扩展属性，对于精通JavaScript的高手来说，这无疑是一把利剑。但是，任何事物都有两面性，JavaScript的强大的动态化特性也是把双刃剑，你可经常听到另一个声音，认为JavaScript的这种动态性糟糕透了，太过灵活反而导致代码很难预期，无法限制不被期望的修改。毕竟有些人总是对自己或别人写的代码不放心，他们希望能够让代码变得可控，并期望有一套静态类型检查系统来帮助自己减少错误。正因如此，在Flutter中，Dart几乎放弃了脚本语言动态化的特性，如不支持反射、也不支持动态创建函数等。并且Dart在2.0强制开启了类型检查（Strong Mode），原先的检查模式（checked mode）和可选类型（optional type）将淡出，所以在类型安全这个层面来说，Dart和TypeScript、CoffeeScript是差不多的，所以单从这一点来看，Dart并不具备什么明显优势，但综合起来看，Dart既能进行服务端脚本、APP开发、web开发，这就有优势了！

综上所述，笔者还是很看好Dart语言的将来，之所以表这个态，是因为在新技术发展初期，很多人可能还有所摇摆，有所犹豫，所以有必要给大家打一剂强心针，当然，这是一个见仁见智的问题，大家可以各抒己见。