> **想学Flutter，就请关注这个专栏** Flutter系列（一）——详细介绍 Flutter系列（二）——与React Native进行对比 Flutter系列（三）——环境搭建（Windows） Flutter系列（四）——HelloWorld Dart语言详解（一）——详细介绍 Dart语言详解（二）——基本语法 Dart语法详解（三）——进阶篇

文档归档： https://github.com/yang0range/flutterfile

前言

在上一篇文章，我们详细的介绍了Dart语法的一些基本语法，这一这篇文章，我们继续介绍Dart的语法的相关知识。

异常

不管是Java语言还是Dart语言，都有异常，以及异常的捕获，但是不同的是dart中的异常都是非检查异常，方法可以不声明可能抛出的异常，也不要求捕获任何异常。

Dart提供了Exception和Error类型以及一些子类型来定义异常。不过，还可以自定义异常，只要抛出非空对象作为异常即可，不要求必须是Exception和Error对象，但是一般来说都是抛出Exception和Error类型。

接下来我们详细介绍一下。

Exception类型

名称	说明
DeferredLoadException	延迟加载异常
FormatException	格式异常
IntegerDivisionByZeroException	整数除零异常
IOException	IO异常
IsolateSpawnException	隔离产生异常
TimeoutException	超时异常

Error类型

名称	说明	名称	说明
AbstractClassInstantiationError	抽象类实例化错误	NoSuchMethodError	没有这个方法错误
ArgumentError	参数错误	NullThrownError	Null 错误
AssertionError	断言错误	OutOfMemoryError	内存溢出错误
AsyncError	异步错误	RemoteError	远程错误
CastError	Cast 错误	StackOverflowError	堆栈溢出错误
ConcurrentModificationError	并发修改错误	StateError	状态错误
CyclicInitializationError	周期初始错误	UnimplementedError	未实现的错误
FallThroughError	Fall Through 错误	UnsupportedError	不支持错误
JsonUnsupportedObjectError	json 不支持错误		

异常抛出

异常的抛出和Java还是很相像的。

```
  //抛出Exception对象
  throw new FormatException('格式异常');

  //抛出Error对象
  throw new NullThrownError();

  //抛出任意非null对象
//  throw '这是一个异常';
```

异常捕获

```
try {
throw new NullThrownError();
//    throw new OutOfMemoryError();
  } on OutOfMemoryError {
    //on 指定异常类型
    print('没有内存了');
//    rethrow; //把捕获的异常给 重新抛出
  } on Error {
    //捕获Error类型
    print('Unknown error catched');
  } on Exception catch (e) {
    //捕获Exception类型
    print('Unknown exception catched');
  } catch (e, s) {
    //catch() 可以带有一个或者两个参数， 第一个参数为抛出的异常对象， 第二个为StackTrace对象堆栈信息
    print(e);
    print(s);
```

```
  }
}
```

类

类的概念和Java当中类似。
但是也有很大的区别。

- 类的构造

```
//java中写法
class P {
  double x;
  double y;

  P(int x, int y) {
    this.x = x;
    this.y = y;
  }
}


//dart建议写法
class P {
  num x;
  num y;
  Point(this.x, this.y);
```

- 重定向构造函数

```
class P {
  num x;
  num y;

  Point(this.x, this.y);

  //重定向构造函数，使用冒号调用其他构造函数
  P.alongXAxis(num x) : this(x, 0);
}
```

- 类的Get和Set方法

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

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

  num get right => left + width;
  set right(num value) => left = value - width;
  num get bottom => top + height;
  set bottom(num value) => top = value - height;
}
```

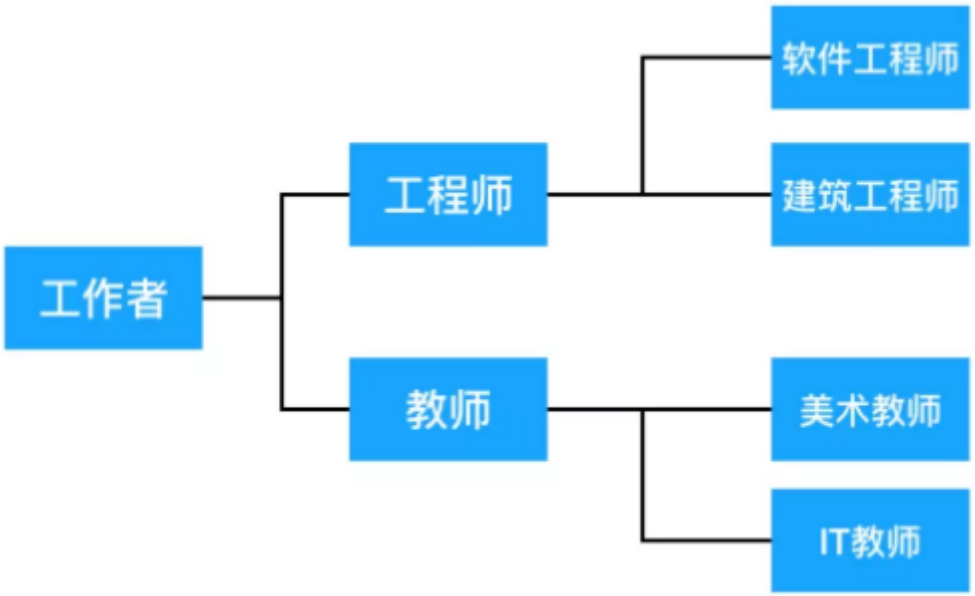
- mixins

Mixins是Dart里面的一个全新概念，简单来说，用来复用多个类之间的代码，减少耦合，换句话来说，可以从中扩展方法（或变量）而不扩展类。

不用不知道，一用起来才发现，真香！

下面具体举例说明一下：

例如，我们有这么一个职业关系图：



从图中可以梳理出以下关系：

- 工程师类目，有软件工程师和建筑工程师等，他们共同点都是工程师。
- 教师类目有美术教师，IT教师等，他们共同点都是教师。

如果我们用Java来实现的花

```
//工作者
abstract class Worker {
    public abstract void doWork();//工作者需要工作
}

//工程师
```

```
class Engineer extends Worker {
    @Override
    public void doWork() {
        System.out.println("工程师在工作");
    }
}

//教师
class Teacher extends Worker {
    @Override
    public void doWork() {
        System.out.println("教师在教学");
    }
}

//软件工程师
class SoftwareEngineer extends Engineer {

}

//建筑工程师
class BuildingEngineer extends Engineer {

}

//美术教师
class ArtTeacher extends Teacher {

}

//IT教师
class ITTeacher extends Teacher {

}
```

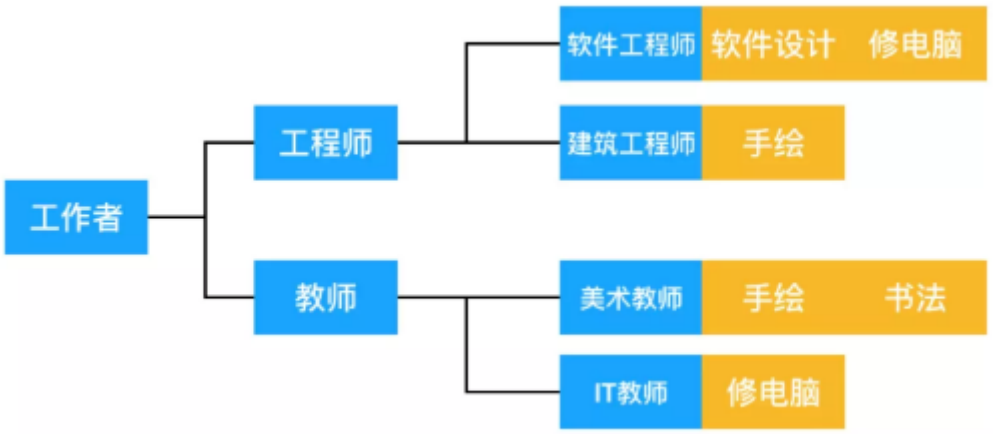
如果用Dart来实现

```
//工作者
abstract class Worker {
  void doWork();//工作者需要工作
}

//工程师
class Engineer extends Worker {
  void doWork() {
    print('工程师在工作');
  }
}

//教师
class Teacher extends Worker {
  void doWork() {
    print('教师在教学');
  }
}
```

```
//软件工程师
class SoftwareEngineer extends Engineer {

}

//建筑工程师
class BuildingEngineer extends Engineer {

}

//美术教师
class ArtTeacher extends Teacher {

}

//IT教师
class ITTeacher extends Teacher {

}
```

从以上来看，似乎Java和Dart没有什么特别大的区别，因为Dart也是单继承。

下面，我们再把场景丰富一下。



通过图形或表格可以看出，软件工程师和IT教师都具备修电脑的能力，建筑工程师和美术教师都具备手绘的能力，但是这些能力都是他们特有的，不是工程师或者教师具备的能力，所以不能在他们的父类中实现。

如果使用Java，那么我们自然就想到了使用interface。

```
interface CanFixComputer {
    void fixComputer();
}

interface CanDesignSoftware {
    void designSoftware();
}
```

```java
//软件工程师
class SoftwareEngineer extends Engineer implements CanFixComputer, CanDesignSoftware {

    @Override
    public void fixComputer() {
        System.out.println("修电脑");
    }

    @Override
    public void designSoftware() {
        System.out.println("设计软件");
    }
}

//IT教师
class ITTeacher extends Teacher implements CanFixComputer {

    @Override
    public void fixComputer() {
        System.out.println("修电脑");
    }
}
```

但是，我们知道Dart当中没有interface的概念，但并不意味着这门语言没有接口，事实上，Dart任何一个类都是接口，你可以实现任何一个类，只需要重写那个类里面的所有具体方法。

如果要用Dart来实现的化，只需要将interface修改成abstract class。

但是我们发现，fixComputer这个接口被继承了两次，并且两次的实现都是一样的，这里就出现了代码重复和冗余的问题。怎么办呢？在java中我们有接口的default实现来解决这个问题(这是一个java8出现的不得已的方案。)

但是，有了mixins之后，这件事就变得不那么难了。

```dart
abstract class CanFixComputer {
  void fixComputer() {
    print('修电脑');
  }
}

abstract class CanDesignSoftware {
  void designSoftware() {
    print('设计软件');
  }
}

//软件工程师
class SoftwareEngineer extends Engineer
    with CanFixComputer, CanDesignSoftware {

}

//IT教师
class ITTeacher extends Teacher with CanFixComputer {
```

```
}

main() {
  ITTeacher itTeacher = new ITTeacher();
  itTeacher.doWork();
  itTeacher.fixComputer();
  SoftwareEngineer softwareEngineer = new SoftwareEngineer();
  softwareEngineer.doWork();
  softwareEngineer.fixComputer();
  softwareEngineer.designSoftware();
}
```

通过以上代码，我们可以看到这里不用implements，更不是extends，而是with。

每个具有某项特性的类不再需要具体去实现同样的功能，接口是没法实现功能的，而通过继承的方式虽然能实现功能，但已经有父类，同时不是一个父类，又不能多继承，所以这个时候，Dart的Mixin机制就比Java的接口会高效，开发上层的只需要关心当前需要什么特性，而开发功能模块的关心具体要实现什么功能。

关于顺序的理解

既然是with，那应该也会有顺序的区别。

```
class First {
  void doPrint() {
    print('First');
  }
}

class Second {
  void doPrint() {
    print('Second');
  }
}

class Father {
  void doPrint() {
    print('Father');
  }
}

class Son1 extends Father with First,Second {
  void doPrint() {
    print('Son1');
  }
}

class Son2 extends Father with First implements Second {
  void doPrint() {
    print('Son2');
  }
}
```

```
main() {
  Son1 son1 = new Son1();
  son1.doPrint();
  Son2 son2 = new Son2();
  son2.doPrint();
}
```

输出的内容是：

```
Son1
Son2
```

通过这里，我们可以看到 可以看到，无论是extends、implements还是mixin，优先级最高的是在具体类中的方法。

下一个例子：

```
class First {
  void doPrint() {
    print('First');
  }
}

class Second {
  void doPrint() {
    print('Second');
  }
}

class Father {
  void doPrint() {
    print('Father');
  }
}

class Son1 extends Father with First,Second {

}

class Son2 extends Father with First implements Second {

}

main() {
  Son1 son1 = new Son1();
  son1.doPrint();
  Son2 son2 = new Son2();
  son2.doPrint();
}
```

输出的结果：

```
Second
First
```

其实在Son2中implements只是说要实现他的doPrint()方法，这个时候其实具体实现是First中Mixin了具体实现。 而Mixin的具体顺序也是可以从代码倒过来看的，最后mixin的优先级是最高的。

泛型

在Dart当中，有很多的容器对象，在创建对象时都可以定义泛型类型，这一点和Java是一样的。
例如：

```
var list = List<String>();
 list.add('aaa');
 list.add('bbb');
 list.add('ccc');
 print(list);

 var map = Map<int, String>();
 map[1] = 'aaaa';
 map[2] = 'bbbb';
 map[3] = 'cccc';
 print(map);
```

和Java的区别

Java中的泛型信息是编译时的，泛型信息在运行时是不存在的。

Dart的泛型类型是固化的，在运行时也有可以判断的具体类型。

异步

Future

说到异步就不得不说到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。

因为Flutter返回的都是一个Fluter对象，自然就可以采用链式方法。

1. Future.then 任务执行完后的子任务
2. Future.delayed 延迟执行
3. Future.catchError 如果异步任务发生错误，我们可以在catchError中捕获错误。
4. Future.whenComplete 完成的时候调用。

5. Future.wait 等待多个异步任务都执行结束后才进行一些操作。

Async/await

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）**。

这个问题在JS当中十分的突出，Dart几乎是同样的问题，然后把相关方法平移过来。所以功能和用法也几乎是相同的。

- 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。

只有async方法才能使用await关键字调用方法 如果调用别的async方法必须使用await关键字。

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。

```
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: (){

});
```

输出

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

```
I/flutter (17666): hello 3
```

Isolates-隔离

所有Dart代码都在隔离区内运行，而不是线程。每个隔离区都有自己的内存堆，确保不会从任何其他隔离区访问隔离区的状态。

最后

通过这三篇文章，我们基本上把Dart语言的所涉及，所涵盖的内容都讲述了一遍。

这些还是偏理论多一些，语法还是在多实践，多写，多练的过程当中来找到其中的真谛。

接下来，我们就开始详细的展开Flutter的介绍了！ **Flutter**已经是**Top20**的软件库，通过接下来的一系列的文章，希望我和大家一起来学习**Flutter**，一起进步，一起有所收获，掌握未来技术主流的主动权！

有什么好的建议，意见，想法欢迎给我留言！

欢迎关注公共号

关注公众号会有更多收获！

Flutter系列（一）——详细介绍

Flutter系列（二）——与React Native进行对比

Flutter系列（三）——环境搭建（Windows

原创:Yang

Flutter系列（四）——HelloWorld

原创:Yang

协同开发利器——Git Submodule

原创:Yang

动动小手指点赞，收藏，转发一键三连走一波吧！

参考文档

https://kevinwu.cn/p/ae2ce64/ https://www.jianshu.com/p/06604077d843
《Flutter实战》