第 5 章

Dart 语言

2011 年 10 月，在丹麦召开的 GOTO 大会上，谷歌发布了一种新的编程语言 Dart。Dart 语言的诞生主要是要解决 JavaScript 存在的、在语言设计层面上无法修复的缺陷。

但是，Dart 语言由于缺少顶级项目的使用，一直并没有流行起来。2015 年，在听取了大量开发者的反馈后，谷歌决定将内置的 Dart VM 引擎从 Chrome 移除。

2018 年 12 月，谷歌正式发布了跨平台开发框架 Flutter 1.0 版本。Flutter 随即成为全球开发者最受欢迎的跨平台开发框架，Flutter 在众多谷歌内部研发的语言中选择了 Dart 语言作为开发语言。

同时，谷歌在全新研发的下一代操作系统 Fuchsia OS 中，Dart 被指定为官方的开发语言。

5.1 Dart 语言介绍

Dart 语言是谷歌开发的计算机编程语言，Logo 如图 5-1 所示，被广泛应用于 Web、服务器、移动应用和物联网等领域的开发。Dart 是面向对象、类定义的、单继承的语言。它的语法类似 Java 语言，可以转译为 JavaScript，支持接口（interfaces）、混入（mixins）、抽象类（abstract classes）、具体化泛型（reified generics）、可选类型（optional typing）和 sound type system。

图 5-1 Dart 语言 Logo

1. Dart 的特性

Dart 的特性主要有以下几点：

（1）执行速度快，Dart 是采用 AOT（Ahead Of Time）编译的，可以编译成快速的、可预测的本地代码，也可以采用 JIT（Just In Time）编译。

(2) 易于移植,Dart 可编译成 ARM 和 x86 代码,这样 Dart 可以在 Android、iOS 和其他系统运行。

(3) 容易上手,Dart 充分吸收了高级语言的特性,如果开发者已经熟悉 C++、C、Java 等其中的一种开发语言,基本上就可以快速上手 Dart 开发。

(4) 易于阅读,Dart 使 Flutter 不需要单独的声明式布局语言(XML 或 JSX),或者单独的可视化界面构建器,这是因为 Dart 的声明式编程布局易于阅读。

(5) 避免抢占式调度,Dart 可以在没有锁的情况下进行对象分配和垃圾回收,和 JavaScript 一样,Dart 避免了抢占式调度和共享内存,因此不需要锁。

2. Dart 的重要概念

Dart 的重要概念有以下几点:

(1) 在 Dart 中,一切都是对象,每个对象都是一个类的实例,所有对象都继承自 Object。

(2) Dart 在运行前解析所有的代码,指定数据类型和编译时常量,可以使代码运行得更快。

(3) 与 Java 不同,Dart 不具备关键字 public、protected、private。如果一个标识符以下画线开始,则它和它的库都是私有的。

(4) Dart 支持顶级的函数,如 main(),也支持类或对象的静态和实例方法,还可以在函数内部创建函数。

(5) Dart 支持顶级的变量,也支持类或对象的静态变量和实例变量,实例变量有时称为字段或属性。

(6) Dart 支持泛型类型,如 List<int>(整数列表)或 List<dynamic>(任何类型的对象列表)。

(7) Dart 工具可以报告两种问题:警告和错误。警告只是说明代码可能无法正常工作,但不会阻止程序执行。错误可以是编译时或运行时的。编译时错误会阻止代码执行;运行时错误会导致代码执行时报出异常。

5.2 安装与配置

Dart SDK 包含开发 Web、命令行和服务器端应用所需要的库和命令行工具。

从 Flutter 1.21 版本开始,Flutter SDK 会同时包含完整的 Dart SDK,因此如果已经安装了 Flutter,就无须再特别下载 Dart SDK 了。

这里推荐安装时先安装 Flutter,学习 Dart 语言的主要目的还是使用 Flutter 框架开发可以跨平台的应用 App。

Flutter 的安装配置可参考本书 13.2 节,详细介绍了安装 Flutter 的步骤,在这里不进行介绍。

5.3 第 1 个 Dart 程序

Dart 文件名以.dart 结尾，文件名使用英文小写加下画线的命名方式。

新建文件，命名为 hello_world.dart，如代码示例 5-1 所示。

代码示例 5-1

```
main() {
   print("Hello Dart!");
}
```

main()方法是 Dart 语言预定义的方法，此方法作为程序的入口方法。print()方法能够将字符串输出到标准输出流上(终端)。

Dart 语言中的语句以分号结尾。Dart 语言会忽略程序中出现的空格、制表符和换行符，因此可以在程序中自由使用空格、制表符和换行符，并且可以自由地以简洁一致的方式格式化和缩进程序，使代码易于阅读和理解。

上述代码的输出结果如下：

```
Hello Dart!
```

5.4 变量与常量

和其他语言一样，Dart 语言有变量和常量，下面介绍 Dart 的变量和常量的定义和用法。

1. 变量

变量可以分为不指定类型和指定类型。前者就像用 JavaScript 一样，后者则像用 Java 一样。

不指定类型有两种方法，如代码示例 5-2 所示。

代码示例 5-2　不指定类型

```
//1. 用关键字 var 定义并且没有初始值
var a;
a = 'a is string';
a = 123;
print(a);

//2. 用关键字 dynamic 或者 Object 定义，无所谓有没有初始值
dynamic b;
b = 'test';
```

```
b = 123;
print(b);

Object c = 'test';
c = 123;
print(c);
```

不指定类型的变量只是一个容器，什么数据都可以往里面装，因此用于存储一些过渡的临时值非常方便。

指定类型也有两种方案，需要注意的是采用关键字 var 定义变量时是否在初始化时赋值，这会导致在后续能不能修改这个变量的类型。

代码示例 5-3 指定类型

```
//类似传统 Java 的定义方式
String d;
d = "test";
//d = 1;              //错误,string 类型不能赋值 int
print(d);

//采用关键字 var 定义并且有初始值:自动推断类型
var e = "test";
//e = 1;              //错误,string 类型不能赋值 int
print(d);
```

和其他语言的初始值不一样，Dart 语言中的所有变量的默认值都是 null。例如一个 bool，在其他语言中初始值一般是 false，而在 Dart 语言中，它是 null。所幸的是，最新版本会有 non-nullable 功能，没赋值时会告诉开发者需要去初始化。

2. 常量

如果不打算更改变量的值，则可以使用 final 或者 const 定义。一个 final 变量只能被设置一次，而 const 变量是编译时常量，定义时必须赋值。

1）const

如果之前使用 JavaScript 进行开发，对于 const 还是有些需要注意的地方，因为它是真正的不变，如代码示例 5-4 所示。

代码示例 5-4 chapter05/01/const.dart

```
//const String a;
const String a = 'test';
//a = "test2";              //常量不能再改变它的值
print(a);
const List list = [1, 2, 3];
//和 JavaScript 不一样,常量的数组也是不能修改的
//list[1] = 2;              //编辑器不会报错,但是运行时会报错
```

```
print(list);

//同值的常量指向同一块内存
const String b = "test";
print(identical(a, b)); //是否指向同一块内存位置,true
```

2）final

final 相对来讲就比较简单了，除了只能赋值一次的要求，它更像 JavaScript 下的 const，而且比它还宽松（没有强制要求定义时赋值），如代码示例 5-5 所示。

代码示例 5-5　chapter05/01/final.dart

```
final String c;
c = "test";
//c = "test2";
print(c);

//list 元素可以修改
final list2 = [1, 2, 3];
list2[1] = 2;
print(list2);
```

5.5　内置类型

Dart 的内置类型包括数组、字符串、布尔、列表、Set、Map、Runes、Symbols 类型。

Dart 是一门强类型编程语言，但是可以使用 var 进行变量类型推断。如果要明确说明不需要任何类型，则需要使用特殊类型 dynamic。dynamic 修饰定义的变量可以赋值任何类型，在运行中也可以随时赋值任何类型的变量值。

1. Numbers 数值

Numbers 数值类型包含 int 和 double 两种类型，没有像 Java 中的 float 类型，int 和 double 都是 num 的子类型，如代码示例 5-6 所示。

代码示例 5-6　chapter05/02/00_int.dart

```
int x = 10;
int y = 0xFFEEAA;
double z = 0.1;
var m = 5;
```

2. Strings 字符串

字符串代表了一系列的字符。Dart 字符串是一系列 UTF-16 代码单元。Dart 中的字符串变量使用 String 修饰定义。单引号或双引号包裹的字符组合表示字符串字面量，如代

码示例 5-7 所示。

代码示例 5-7 chapter05/02/01_string. dart

```
void main() {
  String a = "Hello";
  String b = 'Dart';
  var c = "Hello Dart";
}
```

3. Booleans 布尔值

要表示布尔值,可使用 Dart 中的 bool 类型。布尔类型只有两个值:true 和 false,它们都是编译时常量,如代码示例 5-8 所示。

代码示例 5-8 chapter05/02/02_bool. dart

```
void main() {
  bool d = false;
  bool e = true;
  var f = 10 > 15; //f = false
}
```

4. Lists 列表

Dart 语言中的数组被称作列表(List 对象)。Dart 语言中的列表类型的定义如代码示例 5-9 所示。

代码示例 5-9 chapter05/02/03_list. dart

```
void main() {
  List<int> list = [1, 2];
  List<String> list2 = ['hello', 'dart'];
  var list3 = [3, 4];

  list3[0] = 8;

  List<int> list4 = [];                    //未初始化,不定长列表
  List<int> list5 = List.filled(2, 5);     //未初始化,定长列表

  list4.add(7);                            //向列表添加元素
  #list5 的长度为 2,超出时会报错
  list5[0] = 1;                            //将列表的 0 号元素赋值为 1
  list5[1] = 2;                            //将列表的 1 号元素赋值为 2
  print(list3[0]);                         //打印数字 8
}
```

Dart 语言中的列表是有序的,像其他强类型编程语言中的有序集合,列表的类型定义

使用了泛型。

5. Set 集合

Dart 语言中的集合是指无序集合(Set),集合的创建如代码示例 5-10 所示。

代码示例 5-10　chapter05/02/04_set.dart

```
void main() {
  var dynamicSet = Set();
  dynamicSet.add('dart');
  dynamicSet.add('flutter');
  dynamicSet.add(1);
  dynamicSet.add(1);
  print('dynamicSet : ${dynamicSet}');
  //常用属性与 list 类似

  //常用方法,如增、删、改、查与 list 类似
  var set1 = {'dart', 'flutter'};
  print('set1 : ${set1}');
  var set2 = {'go', 'kotlin', 'dart'};
  print('set2 : ${set2}');
  var difference12 = set1.difference(set2);
  var difference21 = set2.difference(set1);
  print('set1 difference set2 : ${difference12}');
                                          //返回 set1 集合里有但 set2 里没有的元素集合
  print('set2 difference set1 : ${difference21}');
                                          //返回 set2 集合里有但 set1 里没有的元素集合
  var intersection = set1.intersection(set2);
  print('set1 set2 交集 : ${intersection}');      //返回 set1 和 set2 的交集
  var union = set1.union(set2);
  print('set1 set2 并集 : ${union}');             //返回 set1 和 set2 的并集
  set2.retainAll(['dart', 'flutter']);            //只保留(要保留的元素需在原 set 中存在)
  print('set2 只保留 dart flutter : ${set2}');
}
```

6. Map 集合

Dart 语言中的映射类型相当于 Python 中的字典类型,其中的元素都是以键-值对的形式存在的,映射的创建如代码示例 5-11 所示。

代码示例 5-11　chapter05/02/05_map.dart

```
void main() {
  //动态类型
  var dynamicMap = Map();
  dynamicMap['name'] = 'dart';
  dynamicMap[1] = 'android';
  print('dynamicMap : ${dynamicMap}');
```

```
  //强类型
  var map = Map<int, String>();
  map[1] = 'android';
  map[2] = 'flutter';
  print('map : ${map}');
  //也可以这样声明
  var map1 = {'name': 'dart', 1: 'android'};
  map1.addAll({'name': 'kotlin'});
  print('map1 : ${map1}');
  //常用属性
//print(map.isEmpty);                    //是否为空
//print(map.isNotEmpty);                 //是否不为空
//print(map.length);                     //键 - 值对的个数
//print(map.keys);                       //key 集合
//print(map.values);                     //value 集合
}
```

7. Runes 符号字符

在 Dart 中,符号是字符串的 UTF-32 代码单元,如代码示例 5-12 所示。

代码示例 5-12 chapter05/02/06_map. dart

```
void main() {
  Runes runes = new Runes('\u{1f605} \u6211');
  var str1 = String.fromCharCodes(runes);
  print(str1);
}
```

输出结果如图 5-2 所示。

图 5-2 输出结果

5.6 函数

Dart 是一种真正的面向对象语言,因此既是函数也是对象并且具有类型 Function。这意味着函数可以分配给变量或作为参数传递给其他函数。

1. 定义方法

和绝大多数编程语言一样,Dart 函数通常的定义方式如代码示例 5-13 所示。

代码示例 5-13　chapter05/03/01_func.dart

```
//函数定义
String getHello() {
  return "hello dart!";
}

void main() {
  //函数调用
  var str = getHello();
  print(str);
}
```

如果函数体中只包含一个表达式，则可以使用简写语法，代码如下。

```
String getHello() => "hello dart!";
```

2. 可选参数

Dart 函数可以设置可选参数，可以使用命名参数，也可以使用位置参数。

命名参数，定义格式如 {param1, param2, ...}，如代码示例 5-14 所示。

代码示例 5-14　chapter05/03/02_func_param1.dart

```
//函数定义
void showPerson({var name, var age}) {
  if (name != null) {
    print("name = $name");
  }
  if (age != null) {
    print("age = $age");
  }
}

void main() {
//函数调用
  showPerson(name: "leo");
}
```

位置参数，使用[]来标记可选参数，如代码示例 5-15 所示。

代码示例 5-15　chapter05/03/03_func_param2.dart

```
//函数定义
void showHello(var name, [var age]) {
  print("name = $name");
```

```
  if (age != null) {
    print("age = $age");
  }
}

//参数给定类型
String sayHello(String from, String msg, [String? device]) {
  var result = 'from dart';
  if (device != null) {
    result = 'result with a device';
  }
  return result;
}

void main(List<String> args) {
  //函数调用
  showHello("dart");
  showHello("dart", 18);
  sayHello("bj","hi","dart");
}
```

3. 默认值

函数的可选参数也可以使用等号(=)设置默认值,如代码示例 5-16 所示。

代码示例 5-16 chapter05/03/04_func_param4.dart

```
//函数定义
void showHello(var name, [var age = 18]) {
  print("name = $name");

  if (age != null) {
    print("age = $age");
  }
}

void main(List<String> args) {
  //函数调用
  showHello("dart");
}
```

4. main()函数

和其他编程语言一样,Dart 中每个应用程序都必须有一个顶级 main()函数,该函数作为应用程序的入口,代码如下:

```
void main() {
  print('Hello, World!');
}
void main(List<String> arguments) {
  print(arguments);
}
```

5. 函数作为参数

Dart 中的函数可以作为另一个函数的参数，如代码示例 5-17 所示。

代码示例 5-17　chapter05/03/05_func_fn.dart

```
//函数定义
void println(String name) {
  print("name =  $name");
}

void showSomething(var name, Function log) {
  log(name);
}

void main(List<String> args) {
  //函数调用
  showSomething("leo", println);
}
```

6. 匿名函数

在 Dart 中可以创建一个没有函数名称的函数，这种函数称为匿名函数，或者称为 lambda 函数、闭包函数，但是和其他函数一样，它也有形参列表，可以有可选参数，如代码示例 5-18 所示。

代码示例 5-18　chapter05/03/06_func_lambda.dart

```
//函数定义
void showLog(var name, Function log) {
  log(name);
}

void main(List<String> args) {
  //函数调用,匿名函数作为参数
  showLog("leo", (name) {
    print("name =  $name");
  });
}
```

匿名函数就是没有名字的函数，代码如下：

```
([[Type] param1[, ...]]) {
  codeBlock;
};
```

匿名函数通常用在不需要被其他场景调用的情况，例如遍历一个 list，代码如下：

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

其他的用法如下：

```
((num x) => x;                          //没有函数名,有必选的位置参数 x
(num x) {return x;}                     //等价于上面的形式
(int x, [int step]) => x + step;        //没有函数名,有可选的位置参数 step
(int x, {int step1, int step2}) => x + step1 + step2;
                                        //没有函数名,有可选的命名参数 step1、step2
```

7. 嵌套函数

Dart 支持嵌套函数，也就是函数中可以定义函数，如代码示例 5-19 所示。

代码示例 5-19　chapter05/03/07_func_loop.dart

```
//函数定义
void showLog(var name) {
  print("That is a nested function!");

  //函数中定义函数
  void println(var name) {
    print("name =  $name");
  }
  println(name);
}

void main(List<String> args) {
  //函数调用
  showLog("leo");
}
```

8. 函数闭包

闭包是一种方法(对象)，它定义在其他方法内部，闭包能够访问外部方法中的局部变量，并持有其状态，如代码示例 5-20 所示。

代码示例 5-20　chapter05/03/08_func_closer.dart

```
test() {
  int count = 0;
  return () {
    print(count++);
  };
}

void main(List<String> args) {
  var func = test();
  func();
  func();
  func();
  func();
}
```

5.7　运算符

Dart 中用到的运算符如表 5-1 所示。

表 5-1　Dart 运算符列表

操作符名称	描　　述
一元后缀	expr++ expr-- () [] . ?.
一元前缀	-expr ! expr ~expr ++expr --expr
乘除操作	* / % ~/
加减操作	+ -
移位	<< >>
按位与	&
按位异或	^
按位或	|
比较关系和类型判断	>= > <= < as is is!
等判断	== !=
逻辑与	&&
逻辑或	||
是否 null	??
条件语句操作	expr1 ? expr2: expr3
级联操作	..
分配赋值操作	= *= /= ~/= %= += -= <<= >>= &= ^= |= ??=

1. 级联

级联“..”可以实现对同一对象执行一系列操作。除了函数调用，还可以访问同一对象

上的字段。这通常会省去创建临时变量的步骤，并允许编写更多的级联代码。

如代码示例 5-21 所示。

代码示例 5-21 级联运算符

```
querySelector('#confirm')                    //获取一个对象
  ..text = '确认操作'                         //使用它的成员
  ..classes.add('confirm')
  ..onClick.listen((e) => window.alert('Confirmed!'));
```

第 1 种方法调用 querySelector()，返回一个 selector 对象。遵循级联符号的代码对这个 selector 对象进行操作，忽略任何可能返回的后续值。

上面的例子相当于下面的写法，如代码示例 5-22 所示。

代码示例 5-22

```
var button = querySelector('#confirm');
button.text = '确认操作';
button.classes.add('confirm');
button.onClick.listen((e) => window.alert('Confirmed!'));
```

注意：严格来讲，级联的“双点”符号不是运算符，这只是 Dart 语法的一部分。

2. 类型测试操作符

as、is 和 is! 操作符在运行时用于检查类型非常方便。使用 as 操作符可以把一个对象转换为特定类型。一般来讲，如果在 is 测试之后还有一些关于对象的表达式，则可以把 as 当作 is 测试的一种简写，代码如下：

```
if (emp is Person) {
  //Type check
  emp.firstName = 'Leo';
}
```

也可以通过 as 来简化代码，代码如下：

```
(emp as Person).firstName = 'Leo';
```

5.8 分支与循环

Dart 中的控制流语句和其他语言一样，包含以下方式：

(1) if 和 else。

(2) for 循环。

(3) while 和 do-while 循环。

(4) break 和 continue。

(5) switch…case 语句。

1. for 循环

可以使用循环的标准迭代,如代码示例 5-23 所示。

代码示例 5-23

```
void main() {
  var list = [1, 2, 3, 4, 5];
  //for 循环
  for (var index = 0; index < list.length; index++) {
    print(list[index]);
  }
  //当不需要使用下标时可以使用这种方法遍历列表的元素
  for (var item in list) {
    print(item);
  }
}
```

如果要迭代的对象是可迭代的,则可以使用 forEach()方法。如果不需要知道当前迭代计数器,则使用 forEach()是一个很好的选择,代码如下:

```
candidates.forEach((candidate) => candidate.interview());
```

2. switch…case 语句

以上控制流语句和其他编程语言的用法一样,switch…case 有一个特殊的用法,可以使用 continue 语句和标签来执行指定的 case 语句,如代码示例 5-24 所示。

代码示例 5-24 switch…case

```
void main() {
  String lan = 'Java';
  //switch…case,每个 case 后面要跟一个 break,默认为 default
  switch (lan) {
    case 'dart':
      print('dart is my fav');
      break;
    case 'Java':
      print('Java is my fav');
      break;
    default:
      print('none');
  }
  switch (lan) {
    D:
    case 'dart':
```

```
        print('dart is my fav');
        break;
      case 'Java':
        print('Java is my fav');
        //先执行当前 case 中的代码,然后跳转到 D 中的 case 继续执行
        continue D;
      //break;
      default:
        print('none');
    }
  }
```

5.9 异常处理

Dart 异常与传统原生平台异常很不一样,原生平台的任务采用多线程调度,当一个线程出现未捕获的异常时,会导致整个进程退出,而在 Dart 中是单线程的,任务采用事件循环调度,Dart 异常并不会导致应用程序崩溃,取而代之的是当前事件后续的代码不会被执行了。

这样带来的好处是一些无关紧要的异常不会导致闪退,用户还可以继续使用核心功能。坏处是这些异常可能没有明显的提示和异常表现,从而导致问题容易被隐藏,如果此时恰好是在核心流程上且链路较长的异常,则可能导致问题排查极难下手。

1. 抛出异常

使用 throw 抛出异常,异常可以是 Exception 或者 Error 类型的,也可以是其他类型的,但是不建议这么用。另外,throw 语句在 Dart 2 中也是一个表达式,因此可以是=>。

非 Exception 或者 Error 类型是可以抛出的,但是不建议这么用,代码如下:

```
testException(){
   throw "this is exception";
}
testException2(){
   throw Exception("this is exception");
}
```

也可以用 =>箭头函数的用法,代码如下:

```
void testException3() => throw Exception("test exception");
```

2. 捕获异常

on 可以捕获到某一类的异常,但是无法获取异常对象; catch 可以捕获到异常对象。这两个关键字可以组合使用; rethrow 可以重新抛出捕获的异常,如代码示例 5-25 所示。

代码示例 5-25

```
testException(){
   throw FormatException("this is exception");
}

main(List < String > args) {
  try{
     testException();
   } on FormatException catch(e){       //如果匹配不到 FormatException,则会继续匹配
     print("catch format exception");
     print(e);
     rethrow;                           //重新抛出异常
  } on Exception{                       //匹配不到 Exception,会继续匹配
     print("catch exception") ;
  }catch(e, r){                         //匹配所有类型的异常。e 是异常对象,r 是 StackTrace
                                        //对象,异常的堆栈信息
  print(e);
  }
}
```

3. finally

finally 内部的语句,无论是否有异常,都会执行,如代码示例 5-26 所示。

代码示例 5-26 finally

```
testException(){
   throw FormatException("this is exception");
}

main(List < String > args) {
  try{
    testException();
  } on FormatException catch(e){
    print("catch format exception");
    print(e);
    rethrow;
  } on Exception{
    print("catch exception") ;
  }catch(e, r){
    print(e);
  }finally{
    print("this is finally");              //在 rethrow 之前执行
  }
}
```

5.10 面向对象编程

面向对象编程包括以下特性。

(1) 封装：封装是将数据和代码捆绑到一起，避免外界的干扰和不确定性。对象的某些数据和代码是私有的，不能被外界访问，以此实现对数据和代码不同级别的访问权限。

(2) 继承：继承是让某种类型的对象获得另一种类型的对象的特征。通过继承可以实现代码的重用，从已存在的类派生出的一个新类将自动具有原来那个类的特性，同时，它还可以拥有自己的新特性。

(3) 多态：多态是指不同事物具有不同表现形式的能力。多态机制使具有不同内部结构的对象可以共享相同的外部接口，通过这种方式减少代码的复杂度。

Dart 是一种面向对象的语言，具有类和基于 mixin 的继承。同 Java 一样，Dart 的所有类也都继承自 Object。

5.10.1 类与对象

类是具有相同类型的对象的抽象。一个对象所包含的所有数据和代码可以通过类来构造。

对象是运行期的基本实体，也是一个包括数据和操作这些数据的代码的逻辑实体，如图 5-3 所示。

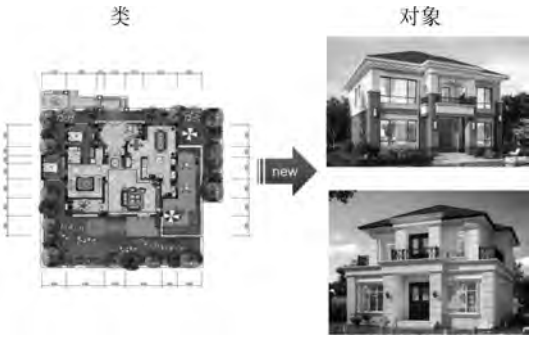

图 5-3 类与对象的关系

1. 类的定义

类可以看成创建具体对象的模板，一个类模板包括类的实例属性和方法，以及类属性和类方法。

Dart 的类与其他语言都有很大的区别，例如在 Dart 的类中可以有无数个构造函数，可以重写类中的操作符，有默认的构造函数，由于 Dart 没有接口，所以 Dart 的类也是接口，因此可以将类作为接口来重新实现。

下面介绍类的定义，如代码示例 5-27 所示。

代码示例 5-27　chapter05/04/01_class.dart

```
class Person {
  //实例属性
  String name;
  int age;
  //私有属性
  String _address;

  //构造函数:与类同名,不支持构造方法重载
  Person(this.name, this.age, this._address);
}
```

创建类的实例对象,代码如下:

代码示例 5-28

```
void main(List<String> args) {
  var p = new Person("leo", 20, "beijing");
}
```

注意:从 Dart 2 开始,new 关键字是可选的。

2. 构造函数

可以使用构造函数来创建一个对象。构造函数的命名方式可以为类名(ClassName)或类名.标识符(ClassName.identifier)的形式,例如下述代码分别使用 Person()和 Person.fromJson()两种构造器创建了 Person 对象。

```
var p1 = Person("leo", 20, "beijing");
var p2 = Person.fromJson();
```

Dart 中不支持构造函数的重载,所有采用 ClassName.构造方法名的方法实现构造方法的重载。

如果没有声明构造函数,则默认有构造函数,默认的构造函数没有参数,可调用父类的无参构造函数。子类不能继承父类的构造函数。

构造函数就是一个与类同名的函数,关键字 this 是指当前的,只有在命名冲突时有效,否则 Dart 会忽略处理。

1）常量构造函数

想让类生成的对象永远不会改变,可以让这些对象变成编译时常量,定义一个 const 构造函数并确保所有实例变量是 final 的,如代码示例 5-29 所示。

代码示例 5-29

```
void main() {
  const point = Point(7, 8);
```

```
}

class Point {
  final int x;
  final int y;
  const Point(this.x, this.y);
}
```

常量构造函数有以下几点特性：

(1) 常量构造函数需以 const 关键字修饰。

(2) const 构造函数必须用于成员变量都是 final 的类。

(3) 构建常量实例必须使用定义的常量构造函数。

(4) 如果实例化时不加 const 修饰符，则即使调用的是常量构造函数，实例化的对象也不是常量实例。

2) 工厂构造函数

使用 factory 关键字实现构造函数时不一定要创建一个类的新实例，例如，一个工厂的构造函数可能从缓存中返回一个实例，或者返回一个子类的实例，如代码示例 5-30 所示。

代码示例 5-30　工厂构造函数

```
void main(){
   var logger = new Logger("Button");
   logger.log("单击了按钮!");
}

class Logger {
   final String name;
   bool mute = false;

   static final Map<String, Logger> _cache = <String, Logger>{};

   factory Logger(String name) {
      if (_cache.containsKey(name)) {
         return _cache[name];
      } else {
         final logger = new Logger._internal(name);
         _cache[name] = logger;
         return logger;
      }
   }

   Logger._internal(this.name);

   void log(String msg) {
```

```
        if (!mute) {
          print(msg);
        }
      }
}
```

3. 实例变量和方法

实例对象可以访问实例变量和方法，如代码示例 5-31 所示。

代码示例 5-31

```
class Person {
  //实例属性
  String name;
  int age;
  String job;
  //私有属性
  String _address;

  //构造函数:与类同名,不支持构造方法重载
  Person(this.name, this.age, this.job, this._address);

  //实例方法
  void say() {
    print("$name say");
  }

  void study() {
    print("$name study");
  }

  //私有实例方法
  void _run() {
    print("$name run");
  }
}

void main(List<String> args) {
  var p = Person("leo", 20, "worker", "beijing");
  p.study();
}
```

4. getter 和 setter

getter 和 setter(也称为访问器和更改器)允许程序分别初始化和检索类字段的值。使用 get 关键字定义 getter(访问器)。setter(更改器)是使用 set 关键字定义的。默认的

getter/setter与每个类相关联，但是，可以通过显式定义setter/getter来覆盖默认值。getter没有参数并返回一个值，setter只有一个参数但不返回值，如代码示例5-32所示。

代码示例5-32 chapter05\04\02_class.dart

```
class Person {
  //实例属性
  String name;
  int age;
  //私有属性
  String _address;

  //setter、getter
  String get address => this._address;
  set address(String addr) => _address = addr;
}
```

5. 重写运算符

在软件开发过程中，运算符重载(Operator Overloading)是多态的一种。运算符重载通常只是一种语法糖，这种语法对语言的功能没有影响，但是更方便程序员使用。让程序更加简洁，有更高的可读性。

可以覆盖的运算符：<、+、|、[]、>、/、^、[]=、<=、~/、&、~、>=、*、<<、==、-、%、>>，如代码示例5-33所示。

代码示例5-33 chapter05\04\03_class.dart

```
class Role {
  final String name;
  final int _accessLevel;

  const Role(this.name, this._accessLevel);
  bool operator >(Role Other) {
    return this._accessLevel > Other._accessLevel;
  }

  bool operator <(Role Other) {
    return this._accessLevel < Other._accessLevel;
  }
}

main() {
  var adminRole = new Role('管理员', 3);
  var editorRole = new Role('编辑', 2);
  var userRole = new Role('用户', 1);
  if (adminRole > editorRole ) {
    print("管理员的权限大于编辑");
```

```
    }
    if (editorRole > userRole) {
      print("编辑的权限大于用户");
    }
  }
```

6. 类的变量和方法

使用 static 关键字实现类的变量和方法。静态变量在其首次被使用时才被初始化。静态方法(类方法)不能被一个类的实例访问,同样地,静态方法内也不可以使用关键字 this,如代码示例 5-34 所示。

代码示例 5-34　chapter05\04\04_class_static.dart

```
class Person {
  //实例属性
  String name;
  int age;

  //类属性 [类型属性]
  static String language = "han";

  //类方法 [类型方法]
  static void work() {
    print("说 $language 的是中国人");
    print("人类需要工作!");
  }

  //构造函数
  Person(this.name, this.age);

  //实例方法
  void say() {
    print(" $name say");
  }

  void study() {
    print(" $name study");
  }
}

void main(List < String > args) {
  //类变量和类方法只能通过类名访问
  Person.language = "中文";
  Person.work();
}
```

5.10.2 类的继承

继承格式和Java的类似，使用extends关键字。继承是复用的一种手段，当子类继承父类时，子类会继承父类的所有公开属性和公开方法（包括计算属性），而私有的属性和方法则不会被继承。子类可以覆写父类的公开方法，如代码示例5-35所示。

代码示例5-35 chapter05\04\05_extends.dart

```
class People {
  say() {
    print("people can say!");
  }
}

class Man extends People {
  @override
  say() {
    print("我是中国男人");
  }
}

class Woman extends People {
  @override
  say() {
    print("我是中国女人");
  }
}

void main(List<String> args) {
  var man = Man();
  man.say();
  var women = Woman();
  women.say();
}
```

Dart中的类的继承特点如下：

（1）子类使用extends关键字来继承父类。

（2）子类会继承父类里可见的属性和方法，但是不会继承构造函数。

（3）子类能复写父类的方法getter和setter。

5.10.3 抽象类

使用abstract修饰符定义的抽象类不能被实例化，抽象类用于定义接口，常用于实现，抽象类里通常有抽象方法，但有抽象方法的不一定是抽象类。

Dart 中的抽象类主要用于定义标准，子类可以继承抽象类，也可以实现抽象类接口：

(1) 抽象类用 abstract 关键字声明。

(2) 抽象类中没有方法体的方法是抽象方法。

(3) 抽象类中可以定义普通方法。

(4) 抽象方法不能使用 abstract 关键字。

(5) 抽象类作为接口使用时必须实现所有的属性和方法。

(6) 抽象类不能被实例化。

(7) 继承抽象类的子类可以实例化。

(8) Dart 中没有 interface 关键字。

抽象类的作用是定义标准，子类继承并实现标准，如代码示例 5-36 所示。

代码示例 5-36　chapter05\04\06_class_abstract.dart

```
abstract class Animal {
  //抽象方法,只有方法声明
  //不需要实现,由子类重写实现
  eat();
  run();
  //普通方法,子类可以选择性地实现
  showInfo() {
    print('我是一个抽象类里的普通方法');
  }
}

class Dog extends Animal {
  @override
  eat() {
    print('小狗在啃骨头');
  }

  @override
  run() {
    //TODO: implement run
    print('小狗在跑');
  }
}

class Cat extends Animal {
  @override
  eat() {
    //TODO: implement eat
    print('小猫在吃老鼠');
  }
```

```
  @override
  run() {
    //TODO: implement run
    print('小猫在跑');
  }
}

main() {
  //Animal a = new Animal();                    //和 Java 类似,抽象类无法直接被实例化
  Dog d = Dog();
  d.eat();
  d.showInfo();

  Cat c = Cat();
  c.eat();
  c.showInfo();
}
```

5.10.4 多态

Dart 中多态的特征如下：

(1) 子类实例化赋值给父类引用。

(2) 多态就是父类定义一种方法,让继承的子类实现其方法,并且每个子类都有自己独有的方法。

(3) 父类引用无法调用子类独有的方法。

多态如代码示例 5-37 所示。

代码示例 5-37 chapter05\04\07_duotai.dart

```
class Animal {
  eat() {
    print('Animal eat');
  }
}

class Dog extends Animal {
  @override
  eat() {
    print("小狗吃");
  }
}

class Cat extends Animal {
  @override
```

```
  eat() {
    print("小猫吃");
  }
}

main(List<String> args) {
  Animal a1 = Dog();
  a1.eat();                //小狗吃

  Animal a2 = Cat();
  a2.eat();                //小猫吃
}
```

5.10.5 隐式接口

Dart 中没有 interface 关键字来定义接口，但是普通类和抽象类都可以作为接口被实现，使用 implements 关键字进行实现。

如果实现的类是普通类，则需要将普通类和抽象类中的属性及方法全重写。抽象类可以定义抽象方法，而普通类则不可以，所以如果要实现接口方式，则一般使用抽象类定义接口。

隐式接口如代码示例 5-38 所示。

代码示例 5-38　chapter05\04\08_interface1.dart

```
abstract class DoSomething {
  start() {
    print("这里是常规开始");
  }

  step1();
  step2();
  step3();
  end() {
    print("这里是常规结束");
  }
}

class DoSubject implements DoSomething {
  @override
  end() {
    //TODO: implement end
    throw UnimplementedError();
  }
```

```
  @override
  start() {
    //TODO: implement start
    throw UnimplementedError();
  }

  @override
  step1() {
    //TODO: implement step1
    throw UnimplementedError();
  }

  @override
  step2() {
    //TODO: implement step2
    throw UnimplementedError();
  }

  @override
  step3() {
    //TODO: implement step3
    throw UnimplementedError();
  }
}
```

下面有一个操作数据库的需求，需要开发一个数据库操作库，要求能够支持 MySQL、MS-SQL、MongoDB 三个数据库的操作，未来可能需要支持更多的数据库。

这里数据库的操作方式基本一样，但是不同数据库有不同的操作处理方式，而且需要考虑可扩展性，这里可以使用接口实现模式，如代码示例 5-39 所示。

代码示例 5-39 chapter05\04\09_interface2.dart

```
abstract class Db {
  String? uri;                    //数据库的链接地址
  add(String data);
  save();
  delete();
}

class Mysql implements Db {
  @override
  String? uri;

  Mysql(this.uri);
```

```
  @override
  add(data) {
    print('这是 MySQL 的 add 方法' + data);
  }

  @override
  delete() {
    return null;
  }

  @override
  save() {
    return null;
  }

  remove() {}
}

class MsSql implements Db {
  @override
  String? uri;

  MsSql(this.uri);

  @override
  add(String data) {
    print('这是 MS-SQL 的 add 方法' + data);
  }

  @override
  delete() {
    return null;
  }

  @override
  save() {
    return null;
  }
}

main() {
  Mysql mysql = new Mysql('MySQL:192.168.0.1');
  mysql.add('dart');
}
```

5.10.6 扩展类

在 Dart 中,扩展类(mixins)可以把自己的方法提供给其他类使用,但不需要成为其他类的父类。

因为 mixins 使用的条件随着 Dart 版本的变化一直在变,这里讲的是 Dart 2 中使用 mixins 的条件:

(1) 作为 mixins 的类只能继承自 Object,不能继承自其他类。

(2) 作为 mixins 的类不能有构造函数。

(3) 一个类可以混入多个 mixins 类。

(4) mixins 绝不是继承,也不是接口,而是一种全新的特性。

1. mixins 通过非继承的方式复用类中的代码

类 A 有一种方法 a(),类 B 需要使用 A 类中的 a()方法,而且不能用继承方式,这时就需要用到 mixins。类 A 就是 mixins 类(混入类),类 B 就是要被混入的类,如代码示例 5-40 所示。

代码示例 5-40 chapter05\04\10_mixins.dart

```
class A {
  String content = 'A Class';

  void a() {
    print("a");
  }
}

class B with A {}

void main(List<String> args) {
  B b = new B();
  print(b.content);
  b.a();
}
```

2. 一个类可以混入多个 mixins 类

虽然 Dart 不支持多重继承,但是可以使用 mixin 实现类似多重继承的功能,如代码示例 5-41 所示。

代码示例 5-41 chapter05\04\11_mixins.dart

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

```
}

class A1 {
  void a1() {
    print("a1");
  }
}

class B with A, A1 {}

void main(List<String> args) {
  B b = new B();
  b.a();
  b.a1();
}
```

3. on 关键字

on 只能用于被 mixins 标记的类，例如 mixin X on A，意思是要 mixins X，得先通过接口实现或者继承 A。这里 A 可以是类，也可以是接口，但是在混入时用法有区别。

on 一个类，用于继承，如代码示例 5-42 所示。

代码示例 5-42　chapter05\04\12_mixins_on.dart

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

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

class MixinsX extends A with X {}

void main(List<String> args) {
  var m = MixinsX();
  m.a();
}
```

on 一个接口，首先实现这个接口，然后用 mixin，如代码示例 5-43 所示。

代码示例 5-43 chapter05\04\13_mixins_on.dart

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

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

class implA implements A {
  @override
  void a() {
    print("implA a");
  }
}

class MixinsX2 extends implA with X {}

void main(List<String> args) {
  var m = MixinsX2();
  m.a();
}
```

5.11 泛型

泛型是程序设计语言的一种特性。允许程序员在强类型程序设计语言中编写代码时定义一些可变部分,这些可变部分在使用前必须进行指明。

1. 泛型方法

泛型方法可以约束一种方法使用同类型的参数、返回同类型的值,可以约束里面的变量类型,如代码示例 5-44 所示。

代码示例 5-44 chapter05\05\01_generic.dart

```
void setData<T>(String key, T value) {
  print("key = ${key}" + " value = ${value}");
}

T getData<T>(T value) {
```

```
  return value;
}

main(List<String> args) {
  setData("name", "hello dart!");                    //string 类型
  setData("name", 123);                              //int 类型

  print(getData("name"));                            //string 类型
  print(getData(123));                               //int 类型
  print(getData<bool>("hello"));                     //错误,约束类型是 bool,但是传入了 String 所
                                                     //以编译器会报错
}
```

2. 泛型类

声明泛型类,例如声明一个 Array 类,实际上就是 List 的别名,而 List 本身也支持泛型的实现,如代码示例 5-45 所示。

代码示例 5-45　chapter05\05\02_generic.dart

```
class Array<T> {
  List _list = [];
  Array();
  void add<T>(T value) {
    this._list.add(value);
  }

  get value {
    return this._list;
  }
}

main(List<String> args) {
  List l1 = [];
  l1.add("aa");
  l1.add("bb");
  print(l1);                //[aa, bb]

  Array arr = new Array<String>();
  arr.add("cc");
  arr.add("dd");
  print(arr.value);         //[cc, dd]

  Array arr2 = new Array<int>();
  arr2.add(1);
  arr2.add(2);
  print(arr2.value);        //[1, 2]
}
```

3. 泛型接口

下面声明一个 Storage 接口，然后 Cache 实现了此接口，能够约束存储的 value 的类型，如代码示例 5-46 所示。

代码示例 5-46 chapter05\05\03_generic.dart

```
abstract class Storage<T> {
  Map m = new Map();
  void set(String key, T value);
  void get(String key);
}

class Cache<T> implements Storage<T> {
  @override
  Map m = new Map();

  @override
  void get(String key) {
    print(m[key]);
  }

  @override
  void set(String key, T value) {
    m[key] = value;
    print("set success!");
  }
}

main(List<String> args) {
  Cache ch = new Cache<String>();
  ch.set("name", "123");
  ch.get("name");
  //ch.set("name", 1232); //type 'int' is not a subtype of type 'String' of 'value'x

  Cache ch2 = new Cache<Map>();
  ch2.set("hello", {"name": "dart", "age": 20});
  ch2.get("hello");
}
```

5.12 异步支持

Dart 和 JavaScript 都是单线程的，并且都提供了一些相似的特性来支持异步编程。在 Dart 中的异步函数返回 Future 或 Stream 对象，await 和 async 关键字用于异步编程，使编写异步代码就像同步代码一样。

5.12.1 Future 对象

Future 和 ECMAScript 6 的 Promise 的特性相似，它们是异步编程的解决方案，Future 是基于观察者模式的，它有 3 种状态：pending(进行中)、fulfilled(已成功)和 rejected(已失败)。

可以使用构造函数来实例化一个 Future 对象，如代码示例 5-47 所示。

代码示例 5-47　chapter05\06\01_future.dart

```
void main() {
  final request = Future<String>(() => 'request success');
  print(request);  //Instance of 'Future<String>'
}
```

Future 构造函数接收一个函数作为参数，泛型参数决定了返回值的类型，在上面的例子中，Future 返回值被规定为 String。

Future 实例生成后，可以用 then()方法指定成功状态的回调函数，如代码示例 5-48 所示。

代码示例 5-48

```
void main() {
  final request = Future<String>(() => 'request success');
  print(request);                    //Instance of 'Future<String>'
  request.then((e) => print(e));     //output: request success
}
```

then()方法还可以接收一个可选命名参数，参数的名称是 onError，即失败状态的回调函数，如代码示例 5-49 所示。

代码示例 5-49

```
void main() {
  final request = Future<String>(() {
    throw new FormatException('Expected at least 1 section');
  });
  final then = request.then((e) => print('success'), onError: (e) => print(e));
  print(then);

  /**
   * output:
   * Instance of 'Future<void>'
   * FormatException: Expected at least 1 section
   */
}
```

在上面的代码中，Future 实例的函数中抛出了异常，被 onError 回调函数捕获到，并且可以看出 then()方法返回的还是一个 Future 对象，所以还可以利用 Future 对象的 catchError 进行链式调用从而捕获异常，用法如代码示例 5-50 所示。

代码示例 5-50

```
void main() {
  final request = Future<String>(() {
    throw new FormatException('Expected at least 1 section');
  });
  request.then((e) => print('success'))
     .catchError((e) => print(e));   //output: FormatException: Expected at least 1 section
}
```

Dart 中也内置了很多方法会返回 Future 对象，例如，File 对象的 readAsString()方法，此方法是异步的，它用于读取文件，调用此方法将返回一个 Future 对象。

5.12.2 async 函数与 await 表达式

使用 async 关键字可以声明一个异步方法，并且该方法会返回一个 Future，如代码示例 5-51 所示。

代码示例 5-51

```
Future<String> getVersion() async {
  return 'v1.0';
}

checkVersion() async => true;

void main() {
  print(getVersion());     //output: Instance of 'Future<String>'
  print(checkVersion());   //output: Instance of 'Future<dynamic>'
}
```

await 表达式必须放入 async 函数体内才能使用，await 表达式会对代码造成阻塞，直至异步操作完成，如代码示例 5-52 所示。

代码示例 5-52

```
void main() async {
  await Future(() => print('request success'));
  print('test');

  /**
   * output:
```

```
    * request success
    * test
    */
}
```

await 表达式能够使异步操作变得更加方便，之前使用 Future 对象进行连续的异步操作时，类似代码示例 5-53 所示。

代码示例 5-53

```
void main() {
  Future<String>(() => 'request1')
    .then((res) {
      print(res);
      return Future<String>(() => 'request2');
    })
    .then((res) {
      print(res);
      return Future<String>(() => 'request3');
    })
    .then(print);

  /**
   * output:
   * request1
   * request2
   * request3
   */
}
```

在上面的代码中，每个异步操作都需要等待上个异步操作完成后才可进行，异步回调 then()方法是个链式操作，如果使用 await 表达式，则可以让这些连续的异步操作变得更加可读，看来起来就像是同步操作，并且拥有相同的效果，如代码示例 5-54 所示。

代码示例 5-54

```
void main() async {
  final res1 = await Future<String>(() => 'request1');
  print(res1);          //output: request1

  final res2 = await Future<String>(() => 'request2');
  print(res2);          //output: request2

  final res3 = await Future<String>(() => 'request3');
  print(res3);          //output: request3
}
```

因为 await 表达式后面是一个 Future 对象，所以可以使用 catchError 来捕获 Future 的异常，如代码示例 5-55 所示。

代码示例 5-55

```
void main() async {
  final res1 = await Future<String>(() => throw 'is error').catchError(print);
  print(res1);

  /**
   * output:
   * is error
   * null
   */
}
```

或者直接使用 try、catch 和 finally 来处理异常，如代码示例 5-56 所示。

代码示例 5-56

```
void main() async {
  try {
    final res = await Future<String>(() => throw 'is error');
  } catch(e) {
    print(e);     //output: is error
  }
}
```

5.13 库和库包

在 Dart 中，library 指令可以创建库，每个 Dart 文件都是一个库，库包(Library Package)是一组库(Library)文件的集合。

Dart 中的库主要有 3 种：自定义的库、系统内置库和 Pub 包管理系统中的库。

5.13.1 库

在 Dart 中，library 指令可以创建库(Library)，每个 Dart 文件都是一个库，即使没有使用 library 指令来指定，库在使用时也可通过 import 关键字引入。

1. 库创建与导出

Library 不仅可以提供 API，也是一个私有单元：以下画线开始的标识符仅仅在所在的 Library 中可见。每个 Dart 程序都是一个 Library，即使它没有使用 library 指令。

创建一个 Dart 文件，该 Dart 文件的名称就是库的名称，在库中编写业务代码，在库中定义的各种方法、变量、类等无须导出命令，其他库通过 import 导入后即可访问使用。

下面创建一个库模块：hello.dart，如代码示例 5-57 所示。

代码示例 5-57

```
//公开的方法,外部导入可用
void showHello() {
  print("hello lib ");
}

//私有方法,外部导入不可用
void _func1() {
  print("func1");
}
```

2. 库引用

模块引用的关键字是 import，import 模块的路径可以是相对路径，用于将其他文件导入当前文件中使用，避免多次复制。导入模块后，可用 show 关键字只对外提供某种方法，如 show log。

在 main.dart 模块中通过 import 导入这个模块，如代码示例 5-58 所示。

代码示例 5-58

```
import '../lib/hello.dart';

void main(List<String> args) {
  showHello();
}
```

3. 导入指定库的前缀

如果要导入两个有标识符冲突的库，则可以为其中一个或者两个指定前缀。例如：如果 hello.dart 和 world.dart 都有一个 showHello()方法，为了不冲突，如代码示例 5-59 所示。

代码示例 5-59

```
mport '../lib/hello.dart' as lib1;
import '../lib/world.dart' as lib2;

void main(List<String> args) {
  lib1.showHello();
  lib2.showHello();
}
```

4. 仅仅导入库的一部分

如果想要使用一个库的一部分，则可以有选择地导入一个库。这里需要使用 show 和

hide 关键字，多个变量用逗号隔开，如代码示例 5-60 所示。

代码示例 5-60

```
//只导入 foo 和 bar
import '../lib1.dart' show foo,bar;

//除了 foo 不导入，其他的都导入
import '../lib2.dart' hide foo;
```

5. 懒加载一个库

延迟加载（也称为懒加载）库允许一个应用程序在需要时才去加载一个库。这里是一些可能使用延迟加载的场景：要减少一个 App 的初始启动时间、A/B 测试和加载很少使用的功能。

要懒加载一个库，必须在第一次导入时使用 deferred as，代码如下：

```
import '../hello.dart' deferred as hello;
```

当需要使用延迟加载的库时，使用库的标识符调用 loadLibrary()，如代码示例 5-61 所示。

代码示例 5-61

```
Future greet() async {
  await hello.loadLibrary();
  hello.showHello();
}
```

可以在一个库上多次调用 loadLibrary()，这是不会有问题的，但该库仅仅会被加载一次。

5.13.2 自定义库包

在 Dart 中，有 pubspec.yaml 文件的应用可以被称为一个 Package，而自定义库包（Library Package）是一类特殊的 Package，这种包可以被其他的项目所依赖，也就是通常所讲的库包。

如果想把自己编写的 Dart 程序上传到 pub.dev 上，或者提供给别人使用，就需要创建库包。

1. 创建 Library Package

在项目工程目录下，使用如下命令创建自定义库包，命令如代码示例 5-62 所示。

代码示例 5-62 创建自定义包的命令

```
flutter create -- template = package PACKAGENAME
```

命令执行后，自动创建一个自定义包目录，这里创建一个 hello 的库包，结构如图 5-4 所示。

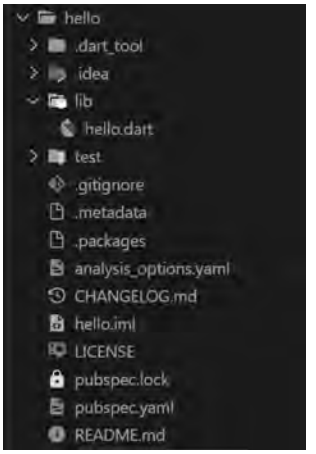

图 5-4　创建一个自定义包目录

2. Library Package 的结构

先看一下 Library Package 的结构，如代码示例 5-63 所示。

代码示例 5-63

```
PackageName
├── lib
|    └── main.dart
└── pubspce.yaml
```

上面是一个最简单的 Library Package 的结构，在 PackageName 目录下面创建一个 pubspce.yaml 文件。lib 目录存放的是 library 的代码。

lib 中的库可以供外部进行引用。如果是 Library 内部的文件，则可以放到 lib/src 目录下面，这里的文件表示是 private 的，不应该被别的程序引入。

如果想要将 src 中的包导出供外部使用，则可以在 lib 下面的 Dart 文件中使用 export，将需要用到的 lib 导出。这样其他用户只需导入这个文件。

3. library 指令

每个 Dart 应用程序默认都是一个 Library，只是没有使用 library 指令显式声明。如 main()方法所在的包，实际上默认隐藏了一个 main 的 library 的声明，如代码示例 5-64 所示。

代码示例 5-64

```
//main.dart
main() {                    //此 main 函数就是 main.dart 库中的顶层函数
```

```
    print('hello dart');
}

//实际上相当于
library main;                    //默认隐藏了一个 main 的 library 的声明
main() {
    print('hello dart');
}
```

创建一个自定义的 Library Package，需要在库文件上面添加 library 声明，如图 5-5 所示。

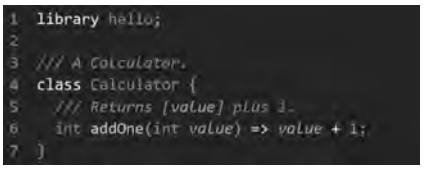

图 5-5 创建一个自定义的 Library Package

4. export 指令

和 JavaScript 中的模块导出不同的是，export 指令用于在包库中导出公开的单个 Dart 库文件。export 后面跟上需要导出的库的相对路径，代码如下：

```
export 'src/adapter.dart';
```

如开源 dio 库，在 dio 库的 lib 目录下的 dio.dart 文件中定义需要导出的公开库，当其他库需要引用这个库时，只需导入这个文件就可以调用所有导出的库了，如代码示例 5-65 所示。

代码示例 5-65

```
library dio;

export 'src/adapter.dart';
export 'src/cancel_token.dart';
export 'src/dio.dart';
export 'src/dio_error.dart';
export 'src/dio_mixin.dart' hide InterceptorState, InterceptorResultType;
export 'src/form_data.dart';
export 'src/headers.dart';
export 'src/interceptors/log.dart';
export 'src/multipart_file.dart';
export 'src/options.dart';
export 'src/parameter.dart';
export 'src/redirect_record.dart';
export 'src/response.dart';
export 'src/transformer.dart';
```

5. part 指令

Dart 中，通过 part、part of、library 指令实现拆分库，这样就可以将一个庞大的库拆分

成各种小库，只要引用主库即可，用法如下：

这里需要创建 3 个 Dart 文件，包括两个子库（calculator 和 logger）和一个主库（util）。子库 calculator.dart 的代码如代码示例 5-66 所示。

代码示例 5-66

```
//和主库建立连接
part of util;

int add(int i, int j) {
  return i + j;
}

int sub(int i, int j) {
  return i - j;
}

int random(int no) {
  return Random().nextInt(no);
}
```

子库 logger.dart 的代码如代码示例 5-67 所示。

代码示例 5-67

```
//和主库建立连接
part of util;

class Logger {
  String _app_name;
  Logger(this._app_name);
  void error(error) {
    print('${_app_name}Error: ${error}');
  }

  void warn(msg) {
    print('${_app_name}Error: ${msg}');
  }

  void deBug(msg) {
    print('${_app_name}Error: ${msg}');
  }
}
```

主库 util.dart 的代码如代码示例 5-68 所示。

代码示例 5-68

```
//给库命名
library util;

//导入 math,子库会用到
import 'dart:math';

//和子库建立联系
part 'logger.dart';
part 'calculator.dart';
```

在 main 中使用,如代码示例 5-69 所示。

代码示例 5-69

```
import './util.dart';

void main() {
  //使用 logger 库定义的类
  Logger logger = Logger('Demo');
  logger.deBug('这是 deBug 信息');

  //使用 calculator 库定义的方法
  print(add(1, 2));
}
```

5.13.3 系统库

Dart 为开发者提供了大量的基础库,这些基础库是开发者在开发中所需的一些基础开发库,如 I/O 操作、数据处理、网络请求、异步处理、文件操作等。

1. io、math 库

dart：math 库中提供了基础的数学函数的调用,如代码示例 5-70 所示。

代码示例 5-70

```
import 'dart:io';
import "dart:math";
main(){
    print(min(122,222));
    print(max(65,89));
}
```

2. 网络库(实现网络请求)

网络库的使用步骤如代码示例 5-71 所示。

代码示例 5-71

```
import 'dart:io';
import 'dart:convert';
void main() async{
  var result = await getInfoListApi();
  print(result);
}
//API:
getInfoListApi() async{
  //1. 创建 HttpClient 对象
  var httpClient = new HttpClient();
  //2. 创建 Uri 对象
  var uri = new Uri.http('www.51itcto.com','/api/3');
  //3. 发起请求,等待请求
  var request = await httpClient.getUrl(uri);
  //4. 关闭请求,等待响应
  var response = await request.close();
  //5. 解码响应的内容
  return await response.transform(utf8.decoder).join();
}
```

5.13.4 第三方库

如果开发应用的过程中需要某些特殊功能的库,但是系统库没有提供,此时就可以试着到第三方库市场搜索、安装及使用,下面介绍查找和安装第三方库的详细步骤。

1. 从下面网址找到要用的库

https://pub.dev/packages

https://pub.flutter-io.cn/packages

https://pub.dartlang.org/flutter/

pub.dev 是谷歌官方维护的一个 Dart 和 Flutter 的第三方代码库的上传下载网站,Dart 提供上传包和下载包的工具供开发者使用。

如需要使用一个强大的 HTTP 访问库,在 pub.dev 上就可以搜索,选择使用人数和排名高的库,如 dio 库,如图 5-6 所示。

2. 创建一个 pubspec.yaml 文件

pubspec.yaml 文件的 dependencies 用来配置需要下载的包名和版本号,然后在配置文件所在的目录命令行中执行 pub get 命令就可以获取远程库。

在 Visual Code 中保存该文件后就会自动把 dependencies 中配置的库包文件下载到本地 Flutter 安装目录下,如 C:\Flutter\.pub-cache\hosted\pub.flutter-io.cn。

图 5-6 DIO 库介绍

```
name: xxx
description: A new flutter module project.
dependencies:
   dio: ^4.0.4
   flutter:
     sdk: flutter
```

3. 查看引入库的使用文档

每个库的介绍页面都有简单的使用入门介绍，通过查看文档，在自己的项目中引用和使用，如获取 dio 库的文档使用说明，如代码示例 5-72 所示。

代码示例 5-72

```
import 'package:dio/dio.dart';
void getHttp() async {
  try {
    var response = await Dio().get('http://www.google.com');
    print(response);
  } catch (e) {
    print(e);
  }
}
```