1. Dart 变量与类型

在 Dart 中，我们可以用 `var` 或者具体的类型来声明一个变量。当使用 `var` 定义变量时，表示类型是交由编译器推断决定的，当然你也可以用静态类型去定义变量。

在默认情况下，未初始化的变量的值都是 null。

Dart 是类型安全的语言，并且所有类型都是对象类型，都继承自顶层类型 `Object`，因此一切变量的值都是类的实例（即对象），甚至数字、布尔值、函数和 null 也都是继承自 Object 的对象。

2. num、bool 与 String

Dart 的数值类型 `num`，只有两种子类：即 64 位 `int` 和符合 IEEE 754 标准的 64 位 `double`。

+、-、*、/ 等常见的运算符也是继承自 num。

为了表示布尔值，Dart 使用了一种名为 `bool` 的类型。

Dart 的 `String` 由 UTF-16 的字符串组成。构造字符串字面量时既能使用单引号也能使用双引号，还能在字符串中嵌入变量或表达式：你可以使用 `${express}` 把一个表达式的值放进字符串。

3. List、Map

在 Dart 中的数组和字典类型对应实现是 `List` 和 `Map`，统称为集合类型。

```
var arr1 = ["Tom", "Andy", "Jack"];
var arr2 = List.of([1,2,3]);
arr2.forEach((v) => print('${v}'));
var map2 = new Map();
map2['name'] = 'Tom';
map2['sex'] = 'male';
map2.forEach((k,v) => print('${k}: ${v}'));
```

4. 常量定义

定义不可变的变量，则需要在定义变量前加上 `final` 或 `const` 关键字：

- const，表示变量在编译期间即能确定的值；
- final 则不太一样，用它定义的变量可以在运行时确定值，而一旦确定后就不可再变。

在定义const常量时，你必须直接赋一个字面量，而不能是一个变量或者公式； 在定义final常量时，如何赋值就无所谓了，但赋值后就不能再改了。

5. 函数

在 Dart 中，所有类型都是对象类型，函数也是对象，它的类型叫作 `Function`。这意味着函数也可以被定义为变量，甚至可以被定义为参数传递给另一个函数。

```
bool isZero(int number) { //判断整数是否为0 return number == 0; }
Function f = isZero;
void printInfo(int number,Function check) => print("$number is Zero:
${isZero(number)}");
```

6. 函数重载

Dart 认为重载会导致混乱，因此从设计之初就不支持重载，而是提供了 **可选命名参数和 可选参数**。

Dart 不支持函数重载，但提供了可选命名参数和可选参数的方式，从而解决了函数声明时需要传递多个参数的可维护性。

在声明函数时：

- 给参数增加{}，以 paramName: value 的方式指定调用参数，也就是可选命名参数；
- 给参数增加[]，则意味着这些参数是可以忽略的，也就是可选参数。

在 Flutter 中会大量用到可选命名参数的方式，你一定要记住它的用法。

```
void enable1Flags({bool bold, bool hidden}) => print("$bold , $hidden");
void enable3Flags(bool bold, [bool hidden]) => print("$bold ,$hidden");
enable1Flags(bold: true, hidden: false); //true, false
enable3Flags(true, false); //true, false
```

7. 类

定义类：

```
class Point {
  num x, y;
  static num factor = 0;
  //语法糖，等同于在函数体内：this.x = x;this.y = y;
  Point(this.x,this.y);
  void printInfo() => print('($x, $y)');
  static void printZValue() => print('$factor');
}
```

Dart 不像 Java 一样，没有 public、private 等关键字，声明变量和方法时，在前面加上 “_” 可作为 private 方法使用。

“_”的限制范围并不是类访问级别的，而是库访问级别。

类的实例化需要根据参数提供多种初始化方式。除了可选命名参数和可选参数之外，Dart 还提供了命名构造函数的方式，使得类的实例化过程语义更清晰。

8. 复用

继承父类和实现接口。

在 Dart 中，你可以对同一个父类进行继承或接口实现：

- 继承父类意味着，子类由父类派生，会自动获取父类的成员变量和方法实现，子类可以根据需要覆写构造函数及父类方法；
- 接口实现则意味着，子类获取到的仅仅是接口的成员变量符号和方法符号，需要重新实现成员变量，以及方法的声明和初始化，否则编译器会报错。

```
class Point {
  num x = 0, y = 0;
  void printInfo() => print('($x,$y)');
}

//Vector继承自Point
class Vector extends Point{
  num z = 0;
  @override
  void printInfo() => print('($x,$y,$z)'); //覆写了printInfo实现
}

//Coordinate是对Point的接口实现
class Coordinate implements Point {
  num x = 0, y = 0; //成员变量需要重新声明
  void printInfo() => print('($x,$y)'); //成员函数需要重新声明实现
}

var xxx = Vector();
xxx
  ..x = 1
  ..y = 2
  ..z = 3; //级联运算符，等同于xxx.x=1; xxx.y=2;xxx.z=3;
xxx.printInfo(); //输出(1,2,3)
```

子类 Coordinate 采用接口实现的方式，仅仅是获取到了父类 Point 的一个“空壳子”，只能从语义层面当成接口 Point 来用，但并不能复用 Point 的原有实现。那么，我们是否能够找到方法去复用 Point 的对应方法实现呢?

也许你很快就想到了，我可以让 Coordinate 继承 Point，来复用其对应的方法。但，如果 Coordinate 还有其他的父类，我们又该如何处理呢?

其实，除了继承和接口实现之外，Dart 还提供了另一种机制来实现类的复用，即“混入”（Mixin）。混入鼓励代码重用，可以被视为具有实现方法的接口。这样一来，不仅可以解决 Dart 缺少对多重继承的支持问题，还能够避免由于多重继承可能导致的歧义（菱形问题）。

要使用混入，只需要 with 关键字即可。

```
class Coordinate with Point {
}

var yyy = Coordinate();
```

```
print (yyy is Point); //true
print(yyy is Coordinate); //true
```

通过混入，一个类里可以以非继承的方式使用其他类中的变量与方法。

9. 运算符

在 Dart 中，一切都是对象，就连运算符也是对象成员函数的一部分。

Dart 多了几个额外的运算符，用于简化处理变量实例缺失（即 null）的情况。

- ?. 运算符：假设 Point 类有 printInfo() 方法，p 是 Point 的一个可能为 null 的实例。那么，p 调用成员方法的安全代码，可以简化为 p?.printInfo() ，表示 p 为 null 的时候跳过，避免抛出异常。
- ??= 运算符：如果 a 为 null，则给 a 赋值 value，否则跳过。这种用默认值兜底的赋值语句在 Dart 中我们可以用 a ??= value 表示。
- ?? 运算符：如果 a 不为 null，返回 a 的值，否则返回 b。在 Java 或者 C++ 中，我们需要通过三元表达式 (a != null)? a : b 来实现这种情况。而在 Dart 中，这类代码可以简化为 a ?? b。

Dart 提供了类似 C++ 的运算符覆写机制，使得我们不仅可以覆写方法，还可以覆写或者自定义运算符。

operator 是 Dart 的关键字，与运算符一起使用，表示一个类成员运算符函数。在理解时，我们应该把 operator 和运算符作为整体，看作是一个成员函数名。

```
class Vector {
  num x, y;
  Vector(this.x, this.y);
  // 自定义相加运算符，实现向量相加
  Vector operator +(Vector v) =>  Vector(x + v.x, y + v.y);
  // 覆写相等运算符，判断向量相等
  bool operator == (dynamic v) => x == v.x && y == v.y;
}

final x = Vector(3, 3);
final y = Vector(2, 2);
final z = Vector(1, 1);
print(x == (y + z)); //  输出true
```