Flutter

概述

简介

Flutter 是谷歌开发的一款开源、免费的，基于 **Dart 语言**的移动 UI 框架，可以快速在 iOS 和 Android 上构建高质量的原生应用。 它最大的特点就是**跨平台**和**高性能**。

- Dart 语言

  Dart 是由谷歌开发的计算机编程语言，它可以被用于 Web、服务器、移动应用和物联网等领域的开发。Dart 诞生于 2011 年，号称要取代 JavaScript。

- 跨平台

  良好的跨平台性，直接带来的好处就是减少开发成本。

  - 移动端
    - Android
    - iOS
  - Web
    - 各种浏览器
  - 桌面
    - Windows
    - Mac
  - 嵌入式平台
    - Linux
    - Fuchsia

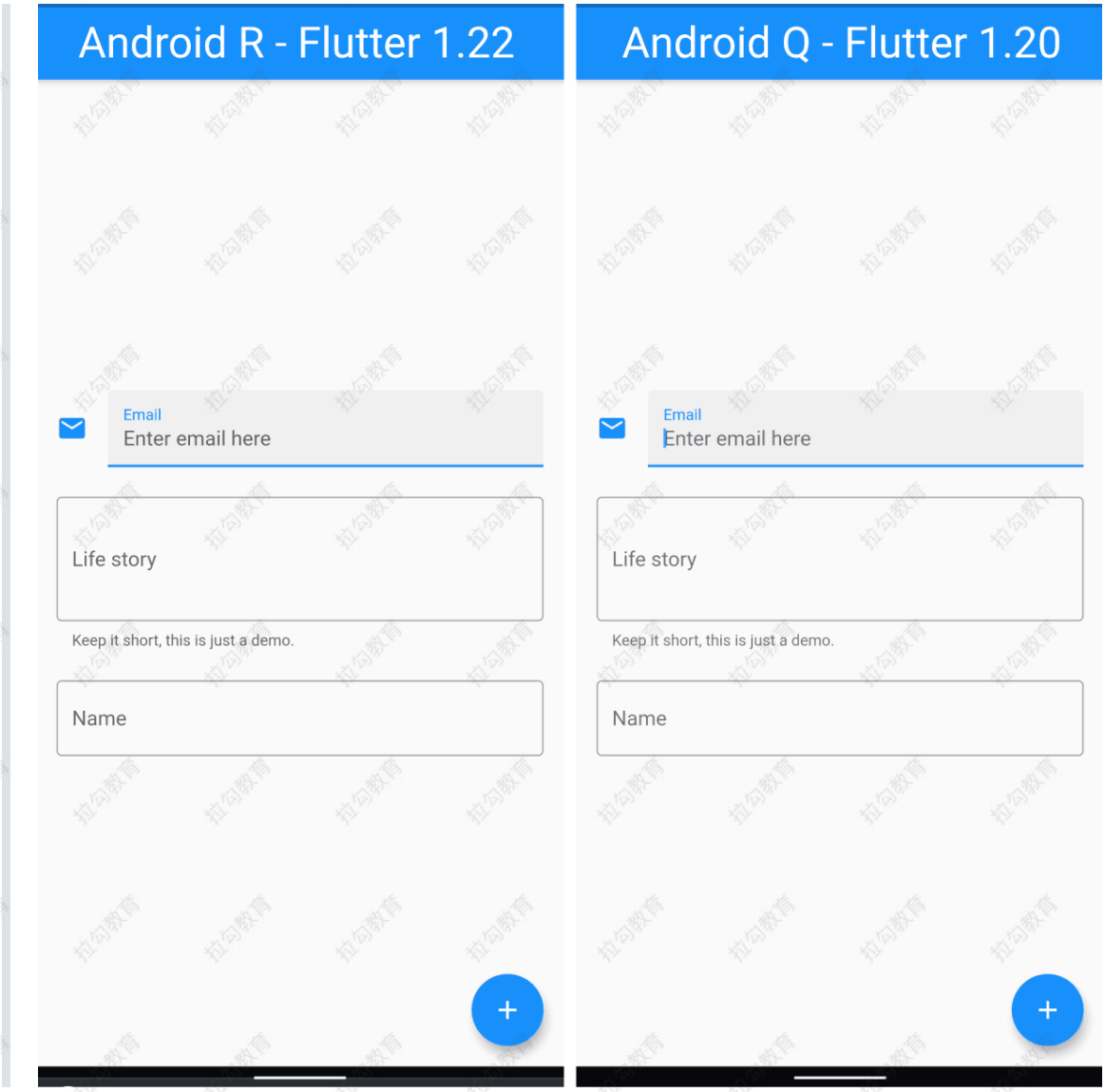
- 高性能

  Flutter 采用 GPU （图形显示）渲染技术，所以性能极高。Flutter 编写的应用是可以达到 120 fps（每秒传输帧数），这也就是说，它完全可以胜任游戏的制作。而 React Native 的性能只能达到 60 fps。

发展历程

- 2015，Flutter 在 Dart 开发者峰会上亮相
- 2018-6，Flutter 发布了首个预览版本
- 2018-12，Flutter 1.0 发布
- 2019-9，Flutter 1.9 发布，添加 Web 端支持
- 2020-9，Flutter 1.22 发布，带来了对 iOS 14 和 Android 11 的支持

> Flutter 1.22 与 Flutter 1.20 的效果展示



主流框架对比

移动应用的三种开发模式

开发模式	原生开发	混合开发	Web 开发
运行环境	Android、iOS	Android、iOS	浏览器、WebView
编程语言	Java、Objective-C	JavaScript、Dart	HTML、CSS、Javascript
可移植性	差	一般	好
开发速度	慢	一般	快
性能	快	较慢	慢
学习成本	高	一般	低

混合开发框架对比

框架	React Native	Weex	Flutter
所属公司	Facebook	Alibaba	Google
编程语言	JavaScript（React）	JavaScript（Vue）	Dart
引擎	JSCore	V8	Flutter engine
支持系统	Android、iOS	Android、iOS、Web	Android、iOS、Fuchsia
性能	一般	较快	较快
适用场景	整体 App	单页面	整体 App
学习成本	难	易	一般

成功案例

除了大量为适应快速发展节奏和节省人力成本的中小型企业，很多一线互联网企业也在重要项目中落地了 Flutter 技术。第一个吃 Flutter 这只螃蟹的头部企业是阿里巴巴。阿里巴巴旗下的多款应用都使用了 Flutter。例如：咸鱼、淘宝特价版、盒马、优酷、飞猪等。另外，字节跳动内部的很多移动端应用，也选中 Flutter。

资源网站

官网：https://flutter.dev/

Github：https://github.com/flutter/flutter

中文网站：

- https://flutterchina.club/
- https://flutter.cn/

环境搭建

Windows 环境搭建

详情参考：Flutter 之 Windows 环境搭建.md

Mac 环境搭建

详情参考：Flutter 之 Mac 环境搭建.md

Dart

简介

Dart 是谷歌开发的，类型安全的，面向对象的编程语言，被应用于 Web、服务器、移动应用和物联网等领域。它的语法类似 C 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。

Dart 诞生于 2011 年 10 月 10 日

Dart 的运行方式有两种

- 原生虚拟机
- JavaScript 引擎（Dart 代码可以转成 JS 代码，然后在浏览器运行）

Dart 是**类型安全**的编程语言：Dart 使用静态类型检查和 运行时检查 的组合来确保，变量的值始终与变量的静态类型或其他安全类型相匹配。

官网：

- 英文：https://dart.dev/
- 中文：https://dart.cn/

如果想要在线运行 Dart 代码，可以使用 DartPad

- https://dartpad.dartlang.org/
- https://dartpad.cn/

Dart 语言概览：https://dart.cn/samples

Dart 速查表：https://dart.cn/codelabs/dart-cheatsheet

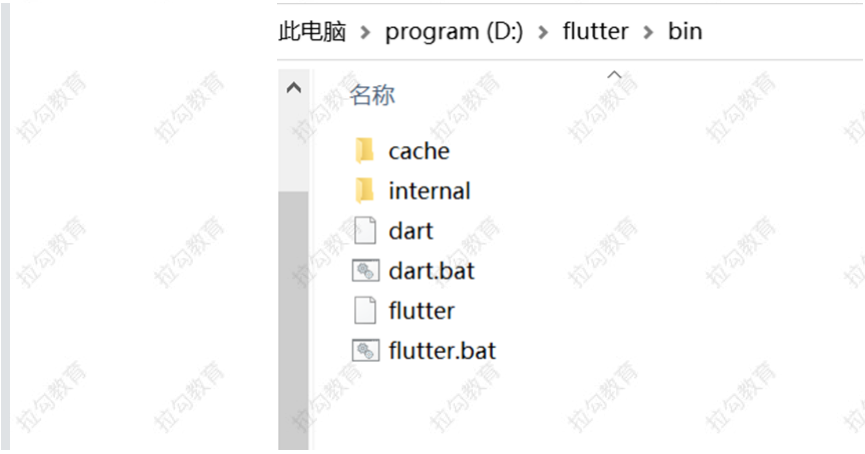
环境搭建

Dart 的环境搭建主要有两种方式

- 跟随 Flutter SDK 一起安装（推荐）

  从 Flutter 1.21 版本开始，Flutter SDK 会同时包含完整的 Dart SDK。

  > 即：如果你已经安装了 Flutter，就无需再下载 Dart SDK 了。
  >
  > 例如，我在 flutter SDK 的 bin 目录下，也找到了 dart.bat



- 单独安装 Dart SDK

  参考官网文档：https://dart.dev/get-dart

基础

- 所有变量引用的都是 **对象**

  每个对象都是一个类的实例。数字、函数以及 `null` 都是对象。所有的类都继承于 Object 类。

- Dart 支持泛型

  比如 `List<int>`（表示一组由 int 对象组成的列表）或 `List<dynamic>`（表示一组由任何类型对象组成的列表）。

- Dart 支持顶级函数（例如 `main` 方法），同时还支持定义属于类或对象的函数（即 **静态** 和 **实例方法**）。你还可以在函数中定义函数（**嵌套** 或 **局部函数**）。

- Dart 支持顶级 **变量**，以及定义属于类或对象的变量（静态和实例变量）。实例变量有时称之为域或属性。

- Dart 没有类似于 Java 那样的 `public`、`protected` 和 `private` 成员访问限定符。如果一个标识符以下划线 (_) 开头则表示该标识符在库内是私有的。详情查看：库和可见性 。

- **标识符** 可以以字母或者下划线 (_) 开头，其后可跟字符和数字的组合。

- Dart 中 **表达式** 和 **语句** 是有区别的，表达式有值而语句没有。

  比如条件表达式 `expression condition ? expr1 : expr2` 中含有值 `expr1` 或 `expr2`。与 if-else 分支语句相比，`if-else` 分支语句则没有值。一个语句通常包含一个或多个表达式。

- Dart 工具可以显示 **警告** 和 **错误** 两种类型的问题。警告表明代码可能有问题但不会阻止其运行。错误分为编译时错误和运行时错误；编译时错误代码无法运行；运行时错误会在代码运行时导致异常。

- main()

  main() 是入口方法：一个特殊且 **必须的** 顶级函数，Dart 应用程序总是会从该函数开始执行。

```dart
// main() 方法是入口方法 - 必须声明 main() 方法程序才能运行
void main() {
  var number = 42;
  print(number); // dart 用打印内容，使用 print() 方法
}
```

- `void`

一种特殊的类型，以 `void` 声明的函数，并不会返回值。

- `int`

  另一种数据类型，表示一个整型数字。 Dart 中一些其他的内置类型包括 `String`、`List` 和 `bool`。

- `print()`

  一种便利的将信息输出显示的方式。

注释

注释方式与 JavaScript 一致

```
// 单行注释

/**
 * 多行注释
 */
```

文档注释可以让 dartdoc 来为你生成代码 API 文档

```
/// 这是文档注释
```

> 由于历史原因，dartdoc 支持两种格式的文档注释：
>
> ```
> /// (“C# 格式”)
> ```
>
> ```
> /** (“JavaDoc 格式”) */
> ```

文档注释允许使用大多数的 MarkDown 格式，并根据 markdown package 进行解析。

```
/// Returns the lesser of two numbers.
///
/// ```dart
/// min(5, 3) == 3
/// ```
```

变量

变量是一个引用，根据Dart中**万物皆对象**原则，即变量存储的都是对象的引用，或者说它们都是指向对象。

声明变量

Dart 是类型安全的，它结合静态类型检查和运行时检查，来保证变量的值总是和变量的静态类型相匹配。

Dart 中声明变量有两种方式：

- 明确指定类型

```
// 声明语法：变量类型 变量名 = 变量值
String name = 'dart'; // 显式声明为字符串。
```

- 不指定类型

```
// 如果未声明变量类型，而是通过 var 关键字声明，Dart 会自动推断变量的类型。
var otherName = 'Dart'; // 自动推动为字符串类型
```

> 在变量类型并不明确的情况下，可以使用 dynamic 关键字
>
> ```
> dynamic name = 'Bob';
> ```

变量名大小写敏感

```
// 变量的名字是区分大小写的
var age = 20;
var Age = 30;

print(age);
print(Age);
```

默认值

如果只声明变量，但是没有给变量赋值。**Dart 中变量的默认是为 null**

```
// JavaScript 变量的默认值是 undefined
var name; // == undefined

// Dart 变量的默认值是 null
var name; // == null
```

null 在 Dart 中是的确存在的，官网上是这样解释的，null 是弱类型 object 的子类型，并非基础数据类型。所有数据类型，如果被初始化后没有赋值的话都将会被赋值 null 类型。

在 Dart 中，只有布尔类型值 `true` 才是 true。因此，对于只为 null 或为 0 的变量，Dart 中不能通过 true 来判断

```
// JavaScript 中的变量会自动地进行类型转换
var myNull = null;
if (!myNull) {
  console.log('null is treated as false');
}

// Dart 中只有布尔类型值 `true` 才是 true。
var myNull = null;
if (myNull == null) {
  print('use "== null" to check null');
}
```

final 和 const

如果你不想更改一个变量，可以使用关键字 `final` 或者 `const` 修饰变量。

> 如果使用 const 修饰类中的变量，则必须加上 static 关键字，即 `static const`（注意：顺序不能颠倒）

```
final name = 'Bob';
// 不能修改 final 声明的变量
name = 'Alice'; // Error: a final variable can only be set once
```

final 与 const 的区别?

- 变量值的分配时机不同

  const 变量的值是编译时分配；而 final 变量的值是运行时分配。

  ```
  const time = DateTime.now(); // 报错 - 无法将运行时的值分配给 const 变量

  final time = DateTime.now(); // 成功 - 可以将运行时的值分配给 final 变量
  ```

- 不可变的是否递归

  const 变量的值不可变是深度递归的，其内部成员的值也都是不可变的

  ```
  void main() {
    const list = [1, 2, 3];
    list[0] = 10; // 尝试修改内部成员的值
    print(list); // 报错
  }
  ```

  final 变量其内部的成员是可变的，即其不可变性不是递归的

  ```
  void main() {
    final list = [1, 2, 3];
    list[0] = 10; // 尝试修改内部成员的值
    print(list); // 不报错
  }
  ```

数据类型

Dart 目前支持一下数据类型

- Number
  - num
  - int
  - double
- Boolean
- String
- List (也被称为 Array)
- Set
- Map

- Rune (用于在字符串中表示 Unicode 字符)
- Symbol

数值

dart:core 库定义了 num ， int 以及 double 类，这些类拥有一定的工具方法来处理数字。

- num 是数值类型，既包含整数，也包含浮点数（小数）
  - int 整数类型（必须是整数）
  - double 浮点数类型（既可以是整数，也可以是小数）

使用 int 和 double 的 `parse()` 方法将字符串转换为整型或双浮点型对象

```
assert(int.parse('42') == 42);
assert(int.parse('0x42') == 66);
assert(double.parse('0.50') == 0.5);
```

或者使用 num 的 parse() 方法，该方法可能会创建一个整型，否则为浮点型对象：

```
assert(num.parse('42') is int);
assert(num.parse('0x42') is int);
assert(num.parse('0.50') is double);
```

通过添加 `radix` 参数，指定整数的进制基数：

```
assert(int.parse('42', radix: 16) == 66);
```

使用 `toString()` 方法将整型或双精度浮点类型转换为字符串类型。

使用 toStringAsFixed(). 指定小数点右边的位数。

使用 toStringAsPrecision(): 指定字符串中的有效数字的位数。

```
// 将 int 转成 string
assert(42.toString() == '42');

// 将 double 转成 string.
assert(123.456.toString() == '123.456');

// 指定小数位数
assert(123.456.toStringAsFixed(2) == '123.46');

// 转换科学计数法
assert(123.456.toStringAsPrecision(2) == '1.2e+2');
assert(double.parse('1.2e+2') == 120.0);
```

int 参考：https://api.dart.dev/stable/dart-core/int-class.html

double 参考：https://api.dart.dev/stable/dart-core/double-class.html

num 参考：https://api.dart.dev/stable/dart-core/num-class.html

字符串

声明字符串

```
// 方式 1: 用 var 声明，然后 Dart 自动推断字符串类型
var str1 = 'this is str1';
var str2 = "this is str2";

// 方式 2: 用 String 声明
String str1 = 'this is str1';
String str2 = "this is str2";

// 方式 3: 三个引号声明可换行的字符串
String str1='''this is str1
this is str1

this is str1
''';
String str1="""
this is str1

this is str1
""";
```

字符串拼接

```
String str1 = 'Hello';
String str2 = 'Dart';

print("$str1 $str2");
print(str1 + " " + str2);
```

在 Dart 中一个字符串是一个固定不变的 UTF-16 编码单元序列。使用正则表达式 (RegExp 对象) 可以在字符串内搜索和替换部分字符串。

> String 定义了例如 `split()`， `contains()`， `startsWith()`， `endsWith()` 等方法。

```
// 包含
assert('Never odd or even'.contains('odd'));

// 匹配开头
assert('Never odd or even'.startsWith('Never'));

// 匹配结尾
assert('Never odd or even'.endsWith('even'));

// 查找指定字符的索引（位置）
assert('Never odd or even'.indexOf('odd') == 6);
```

字符串常见操作

```
// 求子串
assert('Never odd or even'.substring(6, 9) == 'odd');
```

```
// 分割字符串
var parts = 'structured web apps'.split(' ');
assert(parts.length == 3);
assert(parts[0] == 'structured');

// 根据索引取值
assert('Never odd or even'[0] == 'N');

// 通过空字符将字符串分割
for (var char in 'hello'.split('')) {
  print(char);
}

// 获取字符串的 UTF-16 编码
var codeUnitList = 'Never odd or even'.codeUnits.toList();
assert(codeUnitList[0] == 78);
```

大小写转换

```
// 转成大写
assert('hello world'.toUpperCase() == 'HELLO WORLD');

// 转成小写
assert('HELLO WORLD'.toLowerCase() == 'hello world');
```

使用 `trim()` 移除首尾空格。使用 `isEmpty` 检查一个字符串是否为空（长度为0）。

```
// 修剪字符串
assert('  hello  '.trim() == 'hello');

// 判断字符串是否为空
assert(''.isEmpty);

// 判断字符串不为空
assert('  '.isNotEmpty);
```

替换字符串

> 字符串是不可变的对象，也就是说字符串可以创建但是不能被修改。没有一个方法实际的改变了字符串的状态。例如，方法 `replaceAll()` 返回一个新字符串，并没有改变原始字符串：

```
var s1 = 'Hello, NAME!';
var s2 = s1.replaceAll(RegExp('NAME'), 'Bob');

// s1 没有变
assert(s2 != s1);
```

构造字符串

要以代码方式生成字符串，可以使用 StringBuffer。 `writeAll()` 的第二个参数为可选参数，用来指定分隔符

```
var sb = StringBuffer();
sb
  ..write('Use a StringBuffer for ')
  ..writeAll(['efficient', 'string', 'creation'], ' ')
  ..write('.');

var fullString = sb.toString();

assert(fullString == 'Use a StringBuffer for efficient string creation.');
```

正则表达式

RegExp 类提供与 JavaScript 正则表达式相同的功能。使用正则表达式可以对字符串进行高效搜索和模式匹配。

```
// Here's a regular expression for one or more digits.
var numbers = RegExp(r'\d+');

var allCharacters = 'llamas live fifteen to twenty years';
var someDigits = 'llamas live 15 to 20 years';

// contains() can use a regular expression.
assert(!allCharacters.contains(numbers));
assert(someDigits.contains(numbers));

// Replace every match with another string.
var exedOut = someDigits.replaceAll(numbers, 'XX');
assert(exedOut == 'llamas live XX to XX years');
```

获取正则匹配结果

```
var numbers = RegExp(r'\d+');
var someDigits = 'llamas live 15 to 20 years';

// 是否匹配
assert(numbers.hasMatch(someDigits));

// 循环输出所有的匹配结果
for (var match in numbers.allMatches(someDigits)) {
  print(match.group(0)); // 15, 20
}
```

String 参考：https://api.dart.cn/stable/dart-core/String-class.html

布尔

Dart 使用 `bool` 关键字表示布尔类型，布尔类型只有两个对象 `true` 和 `false`，两者都是编译时常量。

Dart 的类型安全不允许你使用类似 `if (nonbooleanValue)` 或者 `assert (nonbooleanValue)` 这样的代码检查布尔值。相反，你应该总是显示地检查布尔值，比如像下面的代码这样：

```
// 检查是否为空字符串 (Check for an empty string).
var fullName = '';
```

```
assert(fullName.isEmpty);

// 检查是否小于等于零。
var hitPoints = 0;
assert(hitPoints <= 0);

// 检查是否为 null。
var unicorn;
assert(unicorn == null);

// 检查是否为 NaN。
var iMeantToDoThis = 0 / 0;
assert(iMeantToDoThis.isNaN);
```

List（列表）

在 Dart 中数组由 List 对象表示。

```
var list = [1, 2, 3];
```

> 这里 Dart 推断出 `list` 的类型为 `List<int>`，如果往该数组中添加一个非 int 类型的对象则会报错。

list 是参数化类型，因此可以指定 list 应该包含的元素类型：

```
// 下面的 list，只能包含字符串
var fruits = List<String>();

fruits.add('apples');
var fruit = fruits[0];
assert(fruit is String);
```

Dart 在 2.3 引入了 **扩展操作符**（`...`）和 **null-aware 扩展操作符**（`...?`），它们提供了一种将多个元素插入数组的简洁方法。

```
var list = [1, 2, 3];
var list2 = [0, ...list];
assert(list2.length == 4);
```

如果扩展操作符右边可能为 null，你可以使用 null-aware 扩展操作符（`...?`）来避免产生异常：

```
var list;
var list2 = [0, ...?list];
assert(list2.length == 1);
```

Dart 在 2.3 还同时引入了 **Collection If** 和 **Collection For**，在构建集合时，可以使用条件判断（`if`）和循环（`for`）。下面示例是使用 **Collection If** 来创建一个 List 的示例，它可能包含 3 个或 4 个元素：

```
var nav = [
  'Home',
  'Furniture',
  'Plants',
  if (promoActive) 'Outlet'
];
```

下面示例是使用 **Collection For** 将列表中的元素修改后添加到另一个列表中的示例：

```
var listOfInts = [1, 2, 3];
var listOfStrings = [
  '#0',
  for (var i in listOfInts) '#$i'
];
assert(listOfStrings[1] == '#1');
```

List 类中有许多用于操作 List 的便捷方法

```
// 构造函数方式声明
var vegetables = List();

// 字面量方式声明
var fruits = ['apples', 'oranges'];

// 添加一项
fruits.add('kiwis');

// 添加多项
fruits.addAll(['grapes', 'bananas']);

// 返回列表长度
assert(fruits.length == 5);

// 删除一项
var appleIndex = fruits.indexOf('apples');
fruits.removeAt(appleIndex);
assert(fruits.length == 4);

// 清空所有
fruits.clear();
assert(fruits.isEmpty);
```

其他 List 的 API，请参考：https://api.dart.dev/stable/dart-core/List-class.html

Set（集合）

在 Dart 中，**Set 是一个无序的，元素唯一的集合**。因为一个 set 是无序的，所以无法通过下标（位置）获取 set 中的元素。

```
var numbers = Set();
numbers.addAll([1, 2, 3]);
assert(numbers.length == 3);

// 添加一个重复的元素，没有任何效果
```

```
numbers.add(1);
assert(numbers.length == 3);

// 在 Set 中删除一个元素
numbers.remove(1);
assert(numbers.length == 2);

// 清空集合
numbers.clear(1);
```

使用 `contains()` 和 `containsAll()` 来检查一个或多个元素是否在 set 中：

```
var numbers = Set();
numbers.addAll([1, 2, 3]);

// 检查单个元素是否在集合中
assert(numbers.contains(1));

// 检查多个元素是否 都 在集合中
assert(numbers.containsAll([2, 3]));
```

交集是由两个集合中公共的元素组成的新集合。

```
var numbers = Set();
numbers.addAll([1, 2, 3]);

// 求两个集合的交集
var even = Set.from([2, 4]); // from 是 Set 的构造函数
var intersection = numbers.intersection(even);
assert(intersection.length == 1);
assert(intersection.contains(2));
```

集合的合并通过 union() 函数完成

```
void main() {
  // 合并
  var s2 = new Set();
  s2.addAll(['糜竺', '孙乾', '简雍', '刘备']);
  var s3 = new Set();
  s3.addAll(['云长', '翼德', '子龙', '刘备']);
  print(s3.union(s2));
  // 返回 {云长, 翼德, 子龙, 刘备, 糜竺, 孙乾, 简雍}
}
```

其他 Set 的 API，请参考：https://api.dart.dev/stable/dart-core/Set-class.html

Map（映射）

Map 是一个无序的 key-value （键值对）集合，就是大家熟知的 字典（*dictionary*） 或者 哈希（*hash*）。 map 将 kay 与 value 关联，以便于检索。和 JavaScript 不同，Dart 对象不是 map。

声明 map 可以使用简洁的字面量语法，也可以使用传统构造函数：

```
// 一般 Map 的键（key）是字符串类型.
var stars = {
  'boy': ['肖战', '王一博', '鹿晗'],
  'girl': ['杨幂', '李一桐', '王晓晨']
};

// 以构造函数的方式声明 Map
var myMap = Map();

// 我们可以指定 Map 的键（key）和值（value）的数据类型
var mapName = Map<int, String>();
```

通过大括号语法可以为 map 添加，获取，设置元素。使用 `remove()` 方法从 map 中移除键值对。

```
var myMap = {9527: '周星星'};

// 通过 key 取值
assert(myMap[9527] == '周星星');

// 检查指定 key 是否存在
assert(myMap.containsKey(9527));

// 删除 key 指定的值
myMap.remove(9527);
assert(!myMap.containsKey(9527));

// 清空
myMap.clear()
```

可以从一个 map 中检索出所有的 key 或所有的 value：

```
// 一般 Map 的键（key）是字符串类型.
var stars = {
  'boy': ['肖战', '王一博', '鹿晗'],
  'girl': ['杨幂', '李一桐', '王晓晨']
};

// 获取 Map 中的所有 key
var keys = stars.keys;
assert(keys.length == 2);
assert(Set.from(keys).contains('boy'));

// 获取 Map 中的所有 value
var values = stars.values;
assert(values.length == 2);
assert(values.any((v) => v.contains('肖战')));
```

使用 `containsKey()` 方法检查一个 map 中是否包含某个key 。

> 因为 map 中的 value 可能会是 null
>
> 通过 key 获取 value，并通过判断 value 是否为 null 来判断 key 是否存在是不可靠的。

```
var stars = {
  'boy': ['肖战', '王一博', '鹿晗'],
  'girl': ['杨幂', '李一桐', '王晓晨']
};

assert(stars.containsKey('boy'));
assert(!stars.containsKey('baby'));
```

如果当且仅当该 key 不存在于 map 中，且要为这个 key 赋值，可使用`putIfAbsent()`方法。该方法需要一个方法返回这个 value。

```
var myMap = {};
myMap.putIfAbsent('Catcher', () => pickToughestKid());
assert(myMap['Catcher'] != null);
```

其他 Map 的 API，请参考：https://api.dart.dev/stable/dart-core/Map-class.html

公共集合方法

List, Set, 和 Map 共享集合中的常用 API。其中一些常见功能由 Iterable 类定义，这些 API 由 List 和 Set 实现。

> 虽然 Map 没有实现 Iterable，但可以使用 Map `keys` 和 `values` 属性从中获取 Iterable 对象。

使用 `isEmpty` 和 `isNotEmpty` 方法可以检查 list， set 或 map 对象中是否包含元素：

```
var coffees = [];
var teas = ['green', 'black', 'chamomile', 'earl grey'];
assert(coffees.isEmpty);
assert(teas.isNotEmpty);
```

使用 `forEach()` 可以让 List， Set 或 Map 对象中的每个元素都使用一个方法。

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

teas.forEach((tea) => print('I drink $tea'));
```

当在 Map 对象上调用 forEach() 方法时，函数必须带两个参数（key 和 value）：

```
hawaiianBeaches.forEach((k, v) {
  print('I want to visit $k and swim at $v');
});
```

Iterable 提供 `map()` 方法，这个方法将所有结果返回到一个对象中。

```
var teas = ['green', 'black', 'chamomile', 'earl grey'];

var loudTeas = teas.map((tea) => tea.toUpperCase());
loudTeas.forEach(print);
```

> `map()` 方法返回的对象是一个懒求值（lazily evaluated）对象：只有当访问对象里面的元素时，函数才会被调用。

使用 `map().toList()` 或 `map().toSet()`，可以强制在每个项目上立即调用函数。

```
var loudTeas = teas.map((tea) => tea.toUpperCase()).toList();
```

使用 Iterable 的 `where()` 方法可以获取所有匹配条件的元素。

使用 Iterable 的 `any()` 和 `every()` 方法可以检查部分或者所有元素是否匹配某个条件。

```
var stars = ['肖战', '王一博', '李一桐', '鹿晗'];

// 李一桐是 Girl
bool isGirl(String starName) => starName == '李一桐';

// 使用 where() 返回唯一的女明星
var girl = stars.where((star) => isGirl(star)); // 或者 stars.where(isGirl)

// 使用 any() 来检测是否有女明星（至少一个）
assert(stars.any(isGirl));

// 使用 every() 是否都是女明星
assert(!stars.every(isGirl));
```

Symbol

在 JavaScript 中，Symbol 是将基础数据类型转换为唯一标识符，核心应用是可以将复杂引用数据类型转换为对象数据类型的键名。

在 Dart 中，Symbol 是不透明的动态字符串名称，用于反映库中的元数据。用 Symbol 可以获得或引用类的一个镜像，概念比较复杂，但其实和 JavaScript 的用法基本上是一致的。例如，下面代码首先 new 了一个 test 为 Map 数据类型，设置一个属性 #t（Symbol 类型），然后分别打印 test、test 的 #t、test 的 Symbol("t") 和 #t。

```
void main() {
  Map test = new Map();
  test[#t] = 'symbol test';
  print(test);
  print(test[#t]);
  print(test[Symbol('t')]);
  print(#t);
}
```

运行代码结果如下：

```
flutter: {Symbol("t"): symbol test}
flutter: symbol test
flutter: symbol test
flutter: Symbol("t")
```

其中，test 包含了一个 Symbol 类型的 Key，value 为 symbol test 字符串的对象。test 的 #t 与 Symbol("t") 打印结果一致，#t 则与 Symbol("t') 是同一形式。

在上面的代码示例中，两者的核心在使用上基本是一致的，只是在理解方面相对不一样。**Symbol 在 Dart 中是一种反射概念，而在 JavaScript 中则是创建唯一标识的概念。**

运算符

地板除

先进行除法运算，然后向下取证

```
assert(5 ~/ 2 == 2); // 结果是一个整数
```

> 学过 Python 的小伙伴肯定很熟悉，这就是 Python 中的 **地板除** 啊

类型判断运算符

操作符	描述
`is`	如果对象是指定类型则返回 true，相当于 JS 中的 instanceof
`is!`	如果对象是指定类型则返回 false，注意：叹号在后面

```
if (emp is Person) {
  // 类型检查
  emp.firstName = 'Bob';
}
```

你可以使用 `as` 运算符进行缩写：

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

> 上述两种方式是有区别的：如果 `emp` 为 null 或者不为 Person 类型，则第一种方式将会抛出异常，而第二种不会。

避空运算符

`??`，左侧表达式为空时，返回右边的表达式。

```
print(1 ?? 3); // 返回 1;
print(null ?? 12); // 输出 12
var value;
print(value ?? 0); // 如果 value 是 null，则返回 0
```

`??=` 变量为空时，才为其赋值：

```
int a; // 任何对象的初始值都是 null
a ??= 3;
print(a); // 输出 3

a ??= 5; // 因为 a 的值不为空，所以此处赋值失败
print(a); // 仍然输出 3
```

条件属性访问

要保护可能会为空的属性的正常访问，请在点（`.`）之前加一个问号（`?`）。

```
var obj;
// print(obj.length); // 警告：The getter 'length' was called on null
print(obj?.length); // 若 obj 不为null，才访问 length
```

级联运算符

要对同一对象执行一系列操作，请使用级联（`..`）。我们都看到过这样的表达式：

```
// 调用 myObject 的 someMethod 方法，返回 someMethod 的返回值
myObject.someMethod()

// 调用 myObject 的 someMethod 方法，返回 myObject 对象的引用
myObject..someMethod()
```

级联可以让你在同一个对象上连续调用多个对象的变量或方法。例如：

```
List listInt = [];
listInt.add(0);
listInt.add(1);
listInt.add(2);
listInt.removeAt(1);
print(listInt); // output: [0, 2]
```

使用级联：

```
List listInt = []
  ..add(0)
  ..add(1)
  ..add(2)
  ..removeAt(1);
print(listInt); // output: [0, 2]
```

流程控制语句

Dart 支持一下流程控制语句

- `if` 和 `else`
- `for` 循环
- `while` 和 `do-while` 循环
- `break` 和 `continue`
- `switch` 和 `case`
- `assert`

Dart 中的流程控制语句与 JavaScript 中一致，这里不再展开。

函数

有些函数是 Dart 提供的。我们称为系统函数/内置函数。例如，我们用过的 print()。这里，我们主要讨论自定义函数。

Dart 是一门真正面向对象的语言， 甚至其中的函数也是对象，并且有它的类型 Function 。 这也意味着函数可以被赋值给变量或者作为参数传递给其他函数。

Dart 中的函数与 JavaScript 中的函数类似。箭头函数、函数闭包、匿名函数、高阶函数、参数可选等基本上都一样。Dart 由于是强类型，因此在声明函数的时候，可以增加一个返回类型，这点在 TypeScript 中的用法是一致的，因此，对于前端开发人员来说，Dart 函数很容易掌握。

声明函数

Dart 中声明函数，**不需要 function 关键字**，声明语法：

```
返回类型 方法名称（参数1，参数2,...）{
  方法体
  return 返回值;
}
```

- 返回类型
  - 表示函数返回结果的类型，可以是 Dart 中合法的数据类型，例如：String，int，List 等。
  - 返回类型可以省略，默认是 void，表示函数没有返回值（只执行代码，没有 return）
- 方法名称
  - 方法名称的约定与 JavaScript 一致，不能出现特殊字符，建议使用小驼峰方式命名
- 参数
  - 多个参数之间用逗号分隔

具体来看：

```
// JavaScript 声明函数
function fn() {
  return true;
}

// Dart 声明函数
fn() {
  return true;
}
// 或指定返回类型
bool fn() {
  return true;
}
```

Dart 支持箭头函数

普通函数：

```
(s) { return s.isEmpty; }
```

箭头函数：

```
(s) => s.isEmpty;
```

函数参数

按照函数参数是否必须。函数参数分类两种: required 和 optional。 required 类型参数在参数最前面，随后是 optional 类型参数。 命名的可选参数也可以标记为 “@ required”

Dart 的函数中有两种传参方法：

- 位置参数

```
int sumUp(int a, int b, int c) {
  return a + b + c;
}

// 中括号内的是可选参数
int sumUp(int a, int b, [int c = 0]) {
  int sum = a + b;
  if (c != 0) sum += c; // 如果 c 存在，则进行累加
  return sum;
}
```

> 1. 可选参数放在中括号中
> 2. 可选参数放在必选参数的后面
> 3. 可以给可选参数指定默认值

- 命名参数

  命名参数通过大括号指定

```
// 这里的 op 是命名参数
void operate(int a, int b, {String op = '+'}) {
  print('$a ${op} $b');
}

// 调用时，命名参数的实参名称，必须与形参名称一致
operate(1, 2, op: '-') // 输出 1 - 2
```

异步函数

Dart 支持异步函数。JavaScript 中的异步调用可以通过 Promise 来实现。Dart 中，通过 Future 来实现异步操作。Dart 中的异步函数与 JavaScript 中类似。支持两种写法：

- 回调写法（then）

  @ JavaScript

```
// JavaScript
function funcName(url) {
  return fetch(url)
    .then(res => res.json())
    .then(resJson => {
      return resJson;
```

```
    });
}

// 调用
const url = "https://httpbin.org/ip"
funcName(url)
  .then(res => console.log(res.origin))
  .catch(err => console.log(err))
```

@ Dart

```
// Dart
import 'dart:convert';
import 'package:http/http.dart' as http;

Future<String> getIPAddress() {
  final url = 'https://httpbin.org/ip';
  return http.get(url).then((response) {
    String ip = jsonDecode(response.body)['origin'];
    return ip;
  });
}

main() {
  getIPAddress()
    .then((ip) => print(ip))
    .catchError((error) => print(error));
}
```

> **http 是第三方包。引入之前需要先安装。**
>
> 1. 在项目的根目录添加 pubspec.yaml 文件
> 2. 在 pubspec.yaml 声明依赖
>
> ```
> // 项目名称
> name: dart
>
> // Dart 环境
> environment:
>   sdk: ">=2.7.0 <3.0.0"
>
> // 项目依赖
> dependencies:
>   http: ^0.12.2
> ```
>
> 3. 执行安装
>   1. VS Code 中，保存时，自动安装
>   2. 或者，在命令行中（项目根目录下），运行 pub get 命令
> 4. 然后运行上面的代码

- 同步写法（async ... await）

  在 JavaScript 中，`async` 函数返回一个 `Promise`。`await` 操作符用于等待 `Promise`

```
```

```javascript
// JavaScript
async function funcName(url) {
  const res await (await fetch(url).json());
  return res;
}

// 调用
const url = "https://httpbin.org/ip"
try {
  const ip = await funcName(url)
  console.log(ip.origin);
} catch (error) {
  console.error(error);
}
```

在 Dart 中，`async` 函数返回一个 `Future`，而函数体会在未来执行。`await` 操作符用于等待 `Future`。

```dart
// Dart
import 'dart:convert'; // 包含 jsonDecode
import 'package:http/http.dart' as http;

Future<String> getIPAddress() async {
  final url = 'https://httpbin.org/ip';
  final response = await http.get(url);
  String ip = jsonDecode(response.body)['origin'];
  return ip;
}

main() async {
  try {
    final ip = await getIPAddress();
    print(ip);
  } catch (error) {
    print(error);
  }
}
```

Future 详情：https://api.dart.dev/stable/dart-async/Future-class.html

对象与类

Dart 是一种基于类和 Mixin（混入） 继承机制的面向对象的语言。 每个对象都是一个类的实例，所有的类都继承于 Object。 Dart 支持单继承， 但可以通过 Mixin 实现代码重用（Mixin 是模拟实现多继承的方案）。

类

类是包含特定功能的代码段。通过 class 关键字声明，类中包含两大内容

- 属性
- 方法（函数在类中称为方法）

也可以说，类是属性和方法的集合，例如：

```
class Person {
  String name = 'Tom';

  say() {
    print('Hello'+name);
  }
}
```

> 与 Java 不同，Dart 中的方法不允许重载（即一个类中不能出现同名方法）

对象

对象是类的实例化结果：

```
var p = new Person(); // 实例化
p.say();
```

> 比喻：类是**建造图纸**，而对象就是根据建造图纸造出来的**房子**。
>
> 没有实例化之前，类不占或少占内存，
>
> 实例化后，对象的内容真实在内存中占用空间（就像建房子需要占地一样）

构造器（构造函数）

构造器是类中的方法

- 默认构造器（Default Constructors）

  默认构造器是一个与类同名的方法，在实例化时，自动被调用。

  ```
  class Point {
    num x, y;

    // 构造器是一个与类名同名的方法
    Point(num x, num y) {
      this.x = x; // 这里的this关键字表示当前实例。
      this.y = y; // 这里的this关键字表示当前实例。
    }
  }

  // 使用
  void main() {
    Point p = new Point(2, 2);
  }
  ```

  > 注意：**默认情况下 Dart 中会省略 this 关键字，只有在命名有歧义时才会使用 this 标识。**

  构造器有简写方式（即，构造器语法糖）

```
class Point {
  num x, y;

  // 构造器语法糖
  Point(this.x, this.y);
}
```

如果没有声明构造器，会提供一个默认的构造器。默认构造器是无参的，并且会调用父类的无参构造器。

子类不会从父类继承构造器。子类如果不声明构造器的话，那么就只会有默认的无参构造器。

- 命名构造器（ Named Constructors ）

  在类中使用命名构造器实现多个构造器，可以提供额外的清晰度：

```
class Point {
  num x, y;

  // 默认构造器
  Point(this.x, this.y);

  // 命名构造器 - 默认坐标
  Point.origin() {
    x = 0;
    y = 0;
  }

  // 命名构造器 - 来自 JSON 的坐标
  Point.fromJson({x:0, y:0}) {
    this.x = x;
    this.y = y;
  }
}

void main() {
  Point p1 = new Point.origin();
  print(p1.x);

  Point p2 = new Point.fromJson(x: 12, y: 18);
  print(p2.x);
}
```

- 常量构造器（Constant Constructors）

  如果生成类的对象不会变，可以定义常量构造函数。常量构造函数可以创建一个**规范化**的实例。

  声明为常量构造函数后，初始化对象时，避免重复开辟新的内存空间。普通的实例化操作，每次实例化对象时，都需要在内存中开辟一块新的空间。例如：

```
class Point {
  num x;
  num y;
  Point(this.x, this.y);
}

void main() {
  var p1 = new Point(1, 2);
  var p2 = new Point(1, 2);
  print(p1 == p2); // 返回 false。因为每次实例化，都会重新开辟内存空间
}
```

关于常量构造函数，有以下几个特点：

- 常量构造函数必须用 **const** 关键词修饰；
- 常量构造函数必须用于成员变量都是 **final** 的类；

  类中不允许有普通类型的属性，因此其变量在构造函数完成之后不允许变更；
- final 属性不设置初始值；
- 常量构造函数不允许有函数体；
- 构建常量实例必须使用定义的常量构造函数；
- 实例化时需要加 **const** 关键字，否则实例化的对象仍然可以修改变量值；

```
class ImmutablePoint {
  final num x;
  final num y;
  const ImmutablePoint(this.x, this.y);
}

void main() {
  // 使用构造器创建两个相等的编译时常量，它们会生成同一个实例
  var p3 = const ImmutablePoint(1, 2);
  var p4 = const ImmutablePoint(1, 2);
  print(p3 == p4); // 返回 true。因为第一次开辟内存空间后，再次通过常量构造函数创建对象，如果对象的属性完全一样，Dart 会指向相同的内存空间。而不是再次开辟新的内存空间
}
```

内部的 const 可以省略

```
// 省略前：
const pointAndLine = const {
  'point': const [const ImmutablePoint(0, 0)],
  'line': const [const ImmutablePoint(1, 10), const ImmutablePoint(-2, 11)],
};

const pointAndLine = {
  'point': [ImmutablePoint(0, 0)],
  'line': [ImmutablePoint(1, 10), ImmutablePoint(-2, 11)],
};
```

- 工厂构造函数（Factory Constructors）

当执行构造函数并不总是创建这个类的一个新实例时。可以考虑使用工厂构造函数。工厂函数不会自动生成实例（因此，工厂构造函数中，不能使用 this 关键字），而是通过代码来决定返回的实例。工厂构造函数通过 factory 关键字声明。

```
class Person{
  String name;

  static Person instance;

  factory Person([String name = '刘备']) {
    if (Person.instance == null) {
      Person.instance = new Person.newSelf(name);
    }
    // print(this.name); // 工厂函数中不能使用 this
    return Person.instance;
  }

  Person.newSelf(this.name);
}

void main(){
  // 工厂函数的调用，与其他构造函数一样
  Person p1 = new Person('关羽');
  print(p1.name); // 输出 关羽

  Person p2 = new Person('张飞');
  print(p2.name); // 输出 关羽

  print(p1 == p2); // 返回 true
}
```

工厂构造函数可以实现单例：

```
class Singleton {
  static final Singleton _singleton = Singleton.internal();

  factory Singleton() => _singleton;

  Singleton.internal();
}
```

访问修饰

Dart 没有类似于 Java 那样的 `public`、`protected` 和 `private` 成员访问限定符。如果一个标识符以下划线 (_) 开头，则表示该标识符是私有的

> 以下划线 (_) 开头的属性和方法，只在当前库内私有，即跨库私有。

- 私有属性
  - 以下划线开头
  - 只能在当前类中访问
- 私有方法
  - 以下划线开头

- 只能在当前类中访问

getter 与 setter

- getter

  getter 声明的函数通过 get 关键字修饰，函数没有小括号，访问时也没有小括号（像访问属性一样访问方法）

```
class Rect {
  num height;
  num width;

  Rect(this.height, this.width);

  // 普通方法的声明
  num area() {
    return this.height * this.width;
  }

  // getter 方法的声明（类型关键字放在 get 前面）
  num get area {
    return this.height * this.width;
  }
}

void main() {
  Rect r = new Rect(10,4);
  // print("面积:${r.area()}"); // 访问普通方法
  print(r.area); // 访问 getter 方法
}
```

- setter

  setter 声明的函数通过 set 关键字声明，访问时，像设置属性一样给函数传参

```
class Rect {
  num height;
  num width;

  Rect(this.height, this.width);

  num get area {
    return this.height * this.width;
  }

  // 声明 setter 方法
  set myHeight(value) {
    this.height = value;
  }
}

void main() {
  Rect r = new Rect(10,4);
  r.myHeight = 6; // 访问 setter 方法
```

```
  print(r.area);
}
```

初始化列表

初始化列表的作用是，在构造函数中给定属性的默认值

```
// Dart 中，我们也可以在构造函数运行之前初始化实例变量
class Rect {
  int height;
  int width;

  // Rect(this.height, this.weight)
  // 调用默认构造函数时，给参数赋值
  Rect() : height = 2, width = 10 {
    // 实例化之前，给属性赋值
    print("${this.height}---${this.width}");
  }

  getArea(){
    return this.height * this.width;
  }
}

void main() {
  Rect r = new Rect(); // 实例化时调用构造函数之前，通过初始化列表赋值
  print(r.getArea());
}
```

```
class Point {
  num x, y;

  Point(this.x, this.y);

  // 命名构造器
  Point.origin() {
    x = 0;
    y = 0;
  }

  // 命名构造器
  Point.fromJson(Map<String, num> json)
    : x = json['x'],
      y = json['y'] {
    print('In Point.fromJson(): ($x, $y)');
  }
}

void main() {
  Point p = Point.fromJson({'x': 2, 'y': 3});
}
```

static

使用 `static` 关键字实现类范围的变量和方法。

- 静态属性
  - 通过 static 关键字声明的属性是静态属性
  - 静态属性可以通过类名称直接访问，即无需实例化过程（类名.属性名）
  - 静态变量只有被使用的时候才会初始化（不使用，不初始化，不占空间）
- 静态方法
  - 通过 static 关键字声明的方法是静态方法
  - 静态方法可以通过类名称直接访问，即无需实例化过程（类名.方法名()）
  - 静态方法（类方法）不能在实例上使用，因此，不能通过 this 访问静态方法
  - 静态方法不能访问非静态成员，非静态方法可以访问静态成员

继承

类可以根据先后关系，分为 **父类** 和 **子类**。一般在父类中声明公共的内容，在子类中写具体的实现，子类通过继承的方式，获取父类中声明的公共的内容（属性和方法）

Dart 支持单继承。

```
// 这里 Orbiter 可以看作子类，Spacecraft 是父类，Orbiter 继承 Spacecraft
class Orbiter extends Spacecraft {
  double altitude;
  Orbiter(String name, DateTime launchDate, this.altitude) : super(name,
launchDate);
}
```

类的继承通过 **extends** 关键字实现。子类中调用父类中的方法，通过 **super** 关键字来实现

元数据

使用元数据可以为代码增加一些额外的信息。元数据注解以 `@` 开头，其后紧跟一个编译时常量（比如 `deprecated`）或者调用一个常量构造函数。

元数据可以在 library、class、typedef、type parameter、 constructor、factory、function、field、parameter 或者 variable 声明之前使用，也可以在 import 或 export 之前使用。可使用反射在运行时获取元数据信息。

比较常用的元数据有：

@override

当父类和子类中出现同名方法后，子类中的方法可以覆盖父类中的同名方法。此时，我们可以在子类的同名方法前加上注解 @override，用来标记当前方法是一个覆盖的方法。

```
class Person {
  void work() {
    print("我是一个人");
  }
}

class Student extends Person {
```

```
  @override // @override 可以写也可以不写，建议加上
  work() {
    // super.work(); // 在子类中调用父类中的方法
    print("我是学生，我的工作是学习");
  }
}

void main() {
  Student w = new Student();
  w.work();
}
```

@required

如果某参数是必填参数，则可以通过 @required 来修饰

```
MyHomePage({Key key, @required this.title}) : super(key:key);
```

@deprecated

若某类或某方法加上该注解之后，表示此方法或类不再建议使用。

例如：下面的代码中，不建议使用 activate 方法，而是推荐使用 trunOn 方法

```
class Television {
  @deprecated   // 表明该方法即将被废弃
  void activate(){
    turnOn();
  }

  void turnOn(){
    print('Television Turn On!');
  }
}
```

另外，我们还可以自定义元数据注解

```
library todo;

class Todo {
  final String who;
  final String what;

  const Todo(this.who, this.what);
}
```

使用 @todo 注解的示例：

```
import 'todo.dart';

@Todo('seth', 'make this do something')
void doSomething() {
  print('do something');
}
```

抽象类

抽象类可以看作类的模板：用来约定子类中，必须出现的属性或方法。抽象类有如下特点：

- 抽象类是通过 abstract 关键字来定义的类
- 抽象类不能被实例化
- 抽象方法是指不含方法体的方法，Dart 中，抽象方法不能通过 abstract 关键字来声明
- 抽象类中可以有抽象方法，也可以没有抽象方法（一般都会有抽象方法）
- 如果子类继承抽象类，子类必须得实现（覆写）抽象类中所有的抽象方法
- 如果把抽象类当做接口实现的话，必须得实现抽象类里面定义的所有属性和方法

例如：手机都有芯片和摄像头。但是不同手机品牌，使用的芯片和摄像头是不一样的。此时，我们可以声明一个抽象类 Phone，约定必须包含芯片方法 chip() 和摄像头方法 camera()，具体的芯片和摄像头信息，可以通过子类来实现。代码如下：

```
abstract class Phone {
  void chip(); // 抽象方法
  void camera(); // 抽象方法

  void name() {
    print('我是一个抽象类里面的普通方法');
  }
}

class Xiaomi extends Phone{
  @override
  void chip() {
    print('骁龙888');
  }

  @override
  void camera() {
    print('三星摄像头');
  }
}

class Huawei extends Phone {
  @override
  void chip() {
    print('麒麟990');
  }

  @override
  void camera() {
    print('徕卡摄像头');
  }
}

void main() {
  // Phone p = new Phone();  // 抽象类不能被实例化

  Xiaomi m = new Xiaomi();
  m.chip();
  m.camera();
  m.name();
```

```
  Huawei h = new Huawei();
  h.chip();
  h.camera();
  h.name();
}
```

> 多态：多态就是父类定义一个方法不去实现，让继承他的子类去实现，每个子类有不同的表现。
>
> 例如：Phone 中有 chip() 方法，但没有实现（chip() 是一个抽象方法，没写具体代码逻辑）
>
> Xiaomi 继承了 Phone 并实现了 chip() 方法，小米用的芯片是骁龙888
>
> Huawei 继承了 Phone 并实现了 chip() 方法，华为用的芯片是麒麟990
>
> **可以用一句话来概括多态：龙生九子，各有不同。**

接口

在 Dart 中，类和接口是统一的，类就是接口（这一点与 TypeScript 不同，TypeScript 中通过 interface 关键字来声明接口）。**只是在具体用法上不同。**接口可以看作是类或抽象类的模板。用来约定一些必须实现的属性和方法。

Dart 中每个类都隐式地定义了一个包含所有实例成员的接口。因此，任意类都可以作为接口被实现。

> 因为接口中约定的内容，必须要在类中实现，所以，一般在 Dart 中，使用抽象类作为接口

类实现接口，通过 **implements** 关键字

```
// 一个类可以实现一个接口
class A implements B {
  // ...
}

// 一个类可以实现多个接口，多个接口之间用逗号分隔
class Phone implements Processor, Camera {
  // ...
}
```

接口可以看成一个个小零件。类实现接口就相当于组装零件。例如：手机中都有处理器和摄像头，而处理器有自己的参数，摄像头也有自己的参数。生产手机时，需要将特定参数的零件组装起来。

```
/**
 * 手机的处理器
 */
abstract class Processor {
  String cores; // 内存核数：2核，4核，8核
  arch(String name); // 芯片制程：7nm，5nm
}

/**
 * 手机的摄像头
 */
abstract class Camera {
  String resolution; // 分辨率：1000万，3000万
  brand(String name); // 品牌：徕卡，蔡司，三星
}
```

```
/**
 * 手机实现 Processor 和 Camera，必须实现接口中的属性和方法
 */
class Phone implements Processor, Camera {
  @override
  String cores;
  @override
  String resolution;

  Phone(this.cores, this.resolution);

  @override
  arch(name) {
    print('芯片制程:'+name);
  }

  @override
  brand(name) {
    print('相机品牌:'+name);
  }
}

void main() {
  Phone p = new Phone('4核', '3000万');
  p.arch('5nm');
  p.arch('徕卡');
}
```

混入（Mixin）

mixin 关键字在 Dart 2.1 中被引用支持。

Mixin 是一段公共代码

Mixin （混入）可以看作类的公共代码。一般，我们可以将一些公共的属性和方法，放在在 Mixin 中。然后类通过引入 Mixin。实现多个类复用一段公共代码（Mixin 是一种在单继承环境中，重用代码的方法）。

> **C++ 支持多继承，可以最大程度地复用代码。Dart 仅支持单继承，通过 Mixin 来弥补代码复用的缺陷。**

Mixin 只能继承 Object

Mixin 只能继承 Object 类。继承其他类会报错。

Mixin 是没有构造函数的类

> **Dart 中的既可以把类当成 Mixin，也可以单独声明 Mixin。**

- 把类用作 Mixin

  如果把类用作 Mixin，则类中不能写构造函数。

  ```
  class MyMixin {
    // 公共属性或方法
  }
  ```

- 单独声明混入（Mixin）

```
mixin MyMixin {
  // 公共属性或方法
}
```

> 注意：通过 mixin 关键字声明和混入，不能作为常规类使用

通过 with 关键字引入 Mixin

类可以通过 with 关键字引入 Mixin，一个类可以使用一个 Mixin，也可以使用多个 Mixin（这一点与接口类似）

```
// 通过 with 关键字引入 Mixin
class MyClass with MyMixin {
  // ...
}

// 也可以先继承，再引入 Mixin
class MyClassA extends MyClassB with MyMixin {
  // ...
}

// 引入 多个 Mixins
class MyClass with MyMixinA, MyMixinB {
  // ...
}
```

同名覆盖

引入多个 Mixins 时，如果多个 Mixins 中，包含同名属性或方法。则最后引入的会覆盖之前引入的。如下：

```
mixin MyMixinA {
  String name = 'MyMixinA';

  void run() {
    print('A');
  }
}

mixin MyMixinB {
  String name = 'MyMixinB';

  void run() {
    print('B');
  }
}

// 引入 多个 Mixins；先引入 A，后引入 B
class MyClass with MyMixinA, MyMixinB {
  // ...
}

void main() {
  MyClass c = new MyClass();
```

```
  c.run(); // 此处返回 B
  print(c.name); // 此处返回 MyMixinB
}
```

> **继承与混入的区别?**
>
> 继承是亲爹（Dart 支持单继承），只能有一个；混入（Mixin）是干爹，可以引入多个。
>
> **接口与混入的区别?**
>
> 接口强制其子类实现其属性和方法，相当于是子类的模板；混入只提供公共内容，不强制使用。

泛型（Generics）

泛型是用来解决类，接口，函数中，关于数据类型的校验问题。通常情况下，使用一个字母来代表类型参数， 例如 E, T, S, K, 和 V 等。

- 泛型函数

```
// 返回类型 函数名<输入类型>（参数类型 参数） { // ... }
T getData<T>(T value){
  return value;
}

void main() {
  // 指定类型：int
  print(getData<int>(12));
}
```

如果不需要指定数据类型，可以不指定。例如，下面的函数未指定返回结果的数据类型

```
// 没有执行返回结果的类型
getData<T>(T value){
  return value;
}
```

如果不适用泛型，我们的代码可能写很多

```
int getData(int value){
  return value;
}

String getData(String value){
  return value;
}

List getData(List value){
  return value;
}

// 更多代码...
```

因此，泛型也可以看成是节省代码的一种方案。使用泛型可以减少重复的代码。

- 泛型类

泛型类与泛型函数类似。可以让类中数据的类型动态变更。例如：

```
class GenericsClass<T>{
  // List list = new List<T>();
  List list = <T>[];

  void add(T value){
    this.list.add(value);
  }

  void printInfo(){
    for (var i = 0; i < this.list.length; i++) {
      print(this.list[i]);
    }
  }
}

void main() {
  GenericsClass g = new GenericsClass<int>();
  g.add(12);
  g.add(23);
  g.printInfo();
}
```

- 泛型接口

  泛型接口是将泛型用在接口中，使得接口中的数据类型可变。

  例如，我要写一个缓存类，缓存的数据可能是字符串，也可能是对象。如果没有泛型。我可能会这样写：

```
abstract class ObjectCache {
  getByKey(String key);
  void setByKey(String key, Object value);
}

abstract class StringCache {
  getByKey(String key);
  void setByKey(String key, String value);
}
```

  使用泛型后的写法，如下：

```
abstract class Cache<T> {
  getByKey(String key);
  void setByKey(String key, T value);
}
```

  有了泛型接口以后，我们可以基于接口，实现一些具体的代码。例如，我要写两个缓存类，一个文件缓存，一个内存缓存。这两个缓存类都是先上面的缓存接口。代码如下：

```
// 泛型接口
abstract class Cache<T> {
  getByKey(String key);
  void setByKey(String key, T value);
}
```

```
// 文件缓存类
class FlieCache<T> implements Cache<T> {
  @override
  getByKey(String key) {
    return null;
  }

  @override
  void setByKey(String key, T value) {
    print("文件缓存: key=${key} value=${value}");
  }
}

// 内存缓存类
class MemoryCache<T> implements Cache<T> {
  @override
  getByKey(String key) {
    return null;
  }

  @override
  void setByKey(String key, T value) {
    print("内存缓存: key=${key} value=${value}");
  }
}

void main() {
  // 文件缓存 - 缓存字符串
  // FlieCache fc = new FlieCache<String>();
  // fc.setByKey('hello', 'world');

  // 文件缓存 - 缓存 Map
  FlieCache fc = new FlieCache<Map>();
  fc.setByKey('index', {"name": "张三丰", "age": 148});

  // 内存缓存 - 缓存字符串
  // MemoryCache mc = new MemoryCache<String>();
  // mc.setByKey('hello', 'world');

  // 内存缓存 - 缓存 Map
  MemoryCache mc = new MemoryCache<Map>();
  mc.setByKey('index', {"name": "张三丰", "age": 148});
}
```

- 字面量形式中使用泛型

  List , Set 和 Map 字面量也是可以参数化的。 参数化字面量和之前的字面量定义类似。

  对于 List 或 Set 只需要在声明语句前加 `<type>` 前缀，

  对于 Map 只需要在声明语句前加 `<keyType, valueType>` 前缀，

  例如：

```
var scores = <double>[67.5, 87, 75];
var names = <String>{'刘备', '关羽', '张飞'};
var routes = <String, String>{
  'home': 'Homepage',
  'user': 'UserPage',
};
```

- 构造函数中使用泛型

  在调用构造函数的时，在类名字后面使用尖括号（`<...>`）来指定泛型类型。 例如：

  ```
  var nameSet = Set<String>.from(names);
  ```

  下面代码创建了一个 key 为 integer， value 为 View 的 map 对象：

  ```
  var views = Map<int, View>();
  ```

- 限制泛型类型

  使用泛型类型的时候， 可以使用 `extends` 实现参数类型的限制。

  ```
  class Foo<T extends SomeBaseClass> {
    // Implementation goes here...
    String toString() => "Instance of 'Foo<$T>'";
  }

  class Extender extends SomeBaseClass {...}
  ```

  可以使用 `SomeBaseClass` 或其任意子类作为通用参数：

  ```
  var someBaseClassFoo = Foo<SomeBaseClass>();
  var extenderFoo = Foo<Extender>();
  ```

  也可以不指定泛型参数：

  ```
  var foo = Foo();
  print(foo); // Instance of 'Foo<SomeBaseClass>'
  ```

  指定任何非 `SomeBaseClass` 类型会导致错误：

  ```
  var foo = Foo<Object>(); // 报错 'Object' doesn't extend 'SomeBaseClass'.
  ```

枚举类型

枚举类型也称为 *enumerations* 或 *enums* ， 是一种特殊的类，用于表示数量固定的常量值。

使用 `enum` 关键字定义一个枚举类型：

```
enum Color { red, green, blue }
```

枚举中的每个值都有一个 `index` getter 方法， 该方法返回值所在枚举类型定义中的位置（从 0 开始）。

例如，第一个枚举值的索引是 0 ， 第二个枚举值的索引是 1。

```
assert(Color.red.index == 0);
assert(Color.green.index == 1);
assert(Color.blue.index == 2);
```

使用枚举的 `values` 常量， 获取所有枚举值列表（ list ）。

```
List<Color> colors = Color.values;
assert(colors[2] == Color.blue);
```

可以在 switch 语句中使用枚举， 如果不处理所有枚举值，会收到警告：

```
var aColor = Color.blue;

switch (aColor) {
  case Color.red:
    print('Red as roses!');
    break;
  case Color.green:
    print('Green as grass!');
    break;
  default: // 没有这个，会看到一个警告。
    print(aColor); // 'Color.blue'
}
```

枚举类型具有以下限制：

- 枚举不能被子类化，混入或实现。
- 枚举不能被显式实例化。

库

简介

Dart 中的库就是具有特定功能的模块。可能包含单个文件，也可能包含多个文件。

Dart 和 JavaScript 一样，有一个库管理网站（https://pub.dev/ - 对应 JavaScript 中的 https://npmjs.com）。你可以在这里搜索找到你想要的一些库，接下来只要在 Dart 的配置文件 pubspec.yaml 中增加该库即可。这点类似于在 JavaScript 的 package.json 中增加声明一样，同样也有 dependencies 和 dev_dependencies。

按照库的作者进行划分，可以将库分为三类

- 内置库：Dart 官方推出
- 第三方库：Dart 生态中的，由第三方作者（非官方，非己方-自己）提交的库
- 自定义库：由工程师自己写的库

引入库

引入方式

Dart 中导入库的方式也与 ES6 的 import 语法很相似。但是不同类型的库，在项目中的导入方式不同

- 引入核心库

```
import 'dart:math';
```

- 引入第三方库

```
import 'package:startup_namer/pages/homepage.dart';
```

> import 为关键词，package 为协议，可以使用 http 的方式，不过最好使用本地 package 方式，避免性能受影响。接下来的 startup_namer 为库名或者说是该项目名，pages 为 lib 下的一个文件夹，homepage.dart 则为具体需要引入的库文件名。

- 导入本地自定义文件

```
import 'path/to/my_other_file.dart';
```

这种方式最简单，在 import 后写上本地完成的路径就可以。

指定库前缀

如果两个代码库中，有冲突的标识符（重名），你可以为其中一个指定前缀的方式，来防止标识符冲突。

比如如果 library1 和 library2 都有 Element 类，那么可以这么处理：

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;

// 使用 lib1 的 Element 类。
Element element1 = Element();

// 使用 lib2 的 Element 类。
lib2.Element element2 = lib2.Element();
```

> Dart 中库前缀的作用于 Java 中的命名空间类似。

引入部分库

有时候，我们不需要整个库的内容。此时可以选择部分导入（即按需加载）

- 包含引入（show）

```
// 只导入 lib1 中的 foo。(Import only foo).
import 'package:lib1/lib1.dart' show foo;
```

- 排除引入（hide）

```
// 导入 lib2 中除了 foo 外的所有。
import 'package:lib2/lib2.dart' hide foo;
```

延时引入

延迟加载（也常称为 *懒加载*）允许应用在需要时再去加载代码库，下面是可能使用到延迟加载的场景：

- 为了减少应用的初始化时间。
- 处理 A/B 测试，比如测试各种算法的不同实现。
- 加载很少会使用到的功能，比如可选的屏幕和对话框。

使用 `deferred as` 关键字来标识需要延时加载的代码库：

```
import 'package:greetings/hello.dart' deferred as hello;
```

当实际需要使用到库中 API 时先调用 `loadLibrary` 函数加载库：

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

前面的代码，使用 `await` 关键字暂停代码执行，直到库加载完成。`loadLibrary` 函数可以调用多次也没关系，代码库只会被加载一次。

当你使用延迟加载的时候需要牢记以下几点：

- 延迟加载的代码库中的常量，需要在代码库被加载的时候才会导入，未加载时是不会导入的。
- 导入文件的时候无法使用延迟加载库中的类型。如果你需要使用类型，则考虑把接口类型转移到另一个库中，然后让两个库都分别导入这个接口库。
- Dart会隐式地将 `loadLibrary` 方法导入到使用了 `deferred as 命名空间` 的类中。`loadLibrary` 函数返回的是一个 Future。

组装库

通过 part 与 part of，可以构建功能负载的库。其中 part 用来引入分库，part of 用来在分库中，与主库建立联系

- 声明分库

  Camera.dart

```
// 与主库建立联系
part of phone; // 库名称就是通过 library 关键字声明的内容

class Camera {
  String name = "摄像头";

  void info() {
    print('我是摄像头');
  }
}
```

Processor.dart

```
// 与主库建立联系
part of phone;

class Processor {
  String name = "处理器";

  void info() {
    print('我是处理器');
  }
}
```

- 声明主库

```
library phone;

import 'dart:math';

// 与分库建立联系
part 'Camera.dart';
part 'Processor.dart';

void main() {
  Camera c = new Camera();
  c.info();

  Processor p = new Processor();
  p.info();

  print(pi);
}
```

- 使用主库

```
import 'lib/phone/main.dart' as phone;

void main() {
  phone.main();
}
```

自定义库（包）

自己写的库（包），使用方式最简单

声明库

```
class Person {
  String name;
  int age;

  // 默认构造函数的简写
  Person(this.name, this.age);

  void info() {
    print("Info: ${this.name} - ${this.age}");
  }
}
```

使用库

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

main(){
  var p = new Person('张三丰', 120);
  print(p.info());
}
```

当然，也可以把自己写的库发布到 pub 上。请参考 **第三方库（包）**

内置库（包）

对于 Dart 内置的库，使用 `dart:xxxxxx` 的形式

- dart:core

  内置类型，集合和其他核心功能。该库会被自动导入到所有的 Dart 程序。

- dart:async

  支持异步编程，包括 Future 和 Stream 等类。

- dart:math

  数学常数和函数，以及随机数生成器。

- dart:convert

  用于在不同数据表示之间进行转换的编码器和解码器，包括 JSON 和 UTF-8。

- dart:html

  用于基于浏览器应用的 DOM 和其他 API。

- dart:io

  服务器和命令行应用程序的 I/O 操作，包括 Flutter 应用，服务端应用，以及命令行脚本。

第三方库（包）

Dart 生态系统使用 包 来管理**共享软件**，比如：库和工具。我们使用 **Pub 包管理工具** 来获取 Dart 包。

https://pub.dev/ (相当于 Node.js 生态中的 https://npmjs.com)

第三方库的使用步骤

1. 在项目根目录下声明 pubspec.yaml 文件

2. 在 pubspec.yaml 中声明依赖（第三方包）
3. 通过 pub get 命令安装依赖（第三方包）
4. 引入第三方包

   通常以 `package:xxxx` 的形式导入。`package:xxxx` 指定的库，通过包管理器（比如 pub 工具）来提供：

   ```
   import 'package:test/test.dart';
   ```

Dart 也支持开发者自己开发一些库，并且发布到 pub.dev 上，这点基本上和 npm 管理一致，这里我只介绍 pub.dev 库的基本格式。

```
dart_string_manip
├── example
|   └── main.dart
├── lib
|   ├── dart_string_manip.dart
|   └── src
|       ├── classes.dart
|       └── functions.dart
├── .gitignore
├── .packages
├── LICENSE
├── README.md
├── pubspec.lock
└── pubspec.yaml
```

Dart 库目录中至少包含一个 pubspec 文件。 pubspec 文件记录一些关于库的元数据。此外，库还包含其他依赖项（在 pubspec 中列出）， Dart 库，应用，资源，测试，图片，以及示例。

- 创建 pubspec

  pubspec 是一个名为 `pubspec.yaml` 的文件，文件位于应用的根路径。最简单的 pubspec 只需要列出包名：

  ```
  name: my_app
  ```

  下面是一个 pubspec 的示例，示例中声明依赖了在 Pub 站点上托管的两个包（ `js` 和 `intl` ）：

  ```
  name: my_app
  dependencies:
    js: ^0.6.0
    intl: ^0.15.8
  ```

  发完成该库以后，如果需要发布到 pub.dev，则可以参照 官网的说明，按步骤进行即可。

- 获取包

  项目中一旦拥有了 pubspec 文件，就可以在项目根目录中执行 `pub get` 命令：

  ```
  cd <path-to-my_app>
  pub get
  ```

`pub get` 命令确定当前应用所依赖的包，并将它们保存到中央系统缓存（central system cache）中。如果当前应用依赖了一个公开包， Pub 会从 Pub 站点 该包。对于一个 Git 依赖， Pub 会 Clone 该 Git 仓库。同样包括包的相关依赖也会被下载。例如，如果 `js` 包依赖 `test` 包， `pub` 会同时获取 `js` 包和 `test` 包。

Pub 会创建一个 `.packages` 文件（位于应用程序的根路目录下），该文件将应用程序所依赖的每个包名相应的映射到系统缓存中的包。

- 从包中导入库

  使用 `package:` 前缀，导入包中的库：

```
import 'package:js/js.dart' as js;
import 'package:intl/intl.dart';
```

  Dart 运行时会抓取 `package:` 之后的内容，并在应用程序的 `.packages` 文件中查找它。

  也可以使用此方式从自己的包中导入库。思考下面的 pubspec 文件，该文件声明了对 `transmogrify` 包（虚构的包名）的依赖：

```
name: my_app
dependencies:
  transmogrify:
```