

# Dart 编程语言概览

这里将介绍 Dart 主要功能，从变量和运算符到类和库，假设您已经知道如何使用其他语言编程。

学习更多 Dart 核心库, 参考 [Dart 库概览](#). 想了解更多有关语言功能的详细信息, 参考 [Dart 编程语言规范](#).

**提示:** 使用 DartPad 可以体验 Dart 的大部分语言功能 ([了解更多](#)).

[打开 DartPad](#)

## 一个简单 Dart 程序

下面的代码用到了很多 Dart 的基本功能：

```
// 定义一个函数
printInteger(int aNumber) {
  print('The number is $aNumber.');
```

```
// 应用从这里开始执行。
main() {
  var number = 42; // 声明并初始化一个变量。
  printInteger(number); // 调用函数。
}
```

以下是此程序使用的代码，这些代码适用于所有（或几乎所有）的 Dart 应用：

**// 代码注释。**  
单行注释。 Dart 同样支持多行注释和文档注释。 有关更多信息，参考 [注释](#).

**int**  
数据类型。一些其他 [内置类型](#) 包括 `String` , `List` , 和 `bool` 。

**42**  
字面量。字面量是一种编译型常量。

**print()**  
便利输出方式。

**'...'** (or **"..."**)  
字符串常量。

**`$variableName` (或 `${expression}`)**  
字符串插值： 包括字符串文字内部的变量或表达式的字符串。 有关更多信息，参考 [Strings](#).

**main()**  
程序开始执行函数，该函数是特定的、*必须的*、 顶级函数。 有关更多信息，参考[The main\(\) function](#).

**var**  
定义变量，通过这种方式定义变量不需要指定变量类型。

**提示：** 本站的代码遵循 Dart 风格指南中的约定。 [Dart 风格指南](#).

## 重要的概念

在学习 Dart 语言时, 应该基于以下事实和概念：

- 任何保存在变量中的都是一个 *对象*， 并且所有的对象都是对应一个 *类* 的实例。 无论是数字，函数和 `null` 都是对象。所有对象继承自 [Object](#) 类。

- 尽管 Dart 是强类型的，但是 Dart 可以推断类型，所以类型注释是可选的。在上面的代码中，`number` 被推断为 `int` 类型。如果要明确说明不需要任何类型，[需要使用特殊类型 `dynamic`](#)。
- Dart 支持泛型，如 `List <int>`（整数列表）或 `List <dynamic>`（任何类型的对象列表）。
- Dart 支持顶级函数（例如 `main ()`），同样函数绑定在类或对象上（分别是 *静态函数* 和 *实例函数*）。以及支持函数内创建函数（*嵌套* 或 *局部函数*）。
- 类似地，Dart 支持顶级 *变量*，同样变量绑定在类或对象上（静态变量和实例变量）。实例变量有时称为字段或属性。
- 与 Java 不同，Dart 没有关键字 “public”，“protected” 和 “private”。如果标识符以下划线（`_`）开头，则它相对于库是私有的。有关更多信息，参考 [库和可见性](#)。
- *标识符* 以字母或下划线（`_`）开头，后跟任意字母和数字组合。
- Dart 语法中包含 *表达式*（*expressions*）（有运行时值）和 *语句*（*statements*）（没有运行时值）。例如，[条件表达式 `condition ? expr1 : expr2`](#) 的值可能是 `expr1` 或 `expr2`。将其与 [if-else 语句](#) 相比较，if-else 语句没有值。一条语句通常包含一个或多个表达式，相反表达式不能直接包含语句。
- Dart 工具提示两种类型问题：*警告\_* 和 *错误\_*。警告只是表明代码可能无法正常工作，但不会阻止程序的执行。错误可能是编译时错误或者运行时错误。编译时错误会阻止代码的执行; 运行时错误会导致代码在执行过程中引发 [异常]（`#exception`）。

## 关键字

Dart 语言关键字列表。

<a href="#">abstract</a> <sup>2</sup>	<a href="#">dynamic</a> <sup>2</sup>	<a href="#">implements</a> <sup>2</sup>	<a href="#">show</a> <sup>1</sup>
<a href="#">as</a> <sup>2</sup>	<a href="#">else</a>	<a href="#">import</a> <sup>2</sup>	<a href="#">static</a> <sup>2</sup>
<a href="#">assert</a>	<a href="#">enum</a>	<a href="#">in</a>	<a href="#">super</a>
<a href="#">async</a> <sup>1</sup>	<a href="#">export</a> <sup>2</sup>	<a href="#">interface</a> <sup>2</sup>	<a href="#">switch</a>
<a href="#">await</a> <sup>3</sup>	<a href="#">extends</a>	<a href="#">is</a>	<a href="#">sync</a> <sup>1</sup>
<a href="#">break</a>	<a href="#">external</a> <sup>2</sup>	<a href="#">library</a> <sup>2</sup>	<a href="#">this</a>
<a href="#">case</a>	<a href="#">factory</a> <sup>2</sup>	<a href="#">mixin</a> <sup>2</sup>	<a href="#">throw</a>
<a href="#">catch</a>	<a href="#">false</a>	<a href="#">new</a>	<a href="#">true</a>
<a href="#">class</a>	<a href="#">final</a>	<a href="#">null</a>	<a href="#">try</a>
<a href="#">const</a>	<a href="#">finally</a>	<a href="#">on</a> <sup>1</sup>	<a href="#">typedef</a> <sup>2</sup>
<a href="#">continue</a>	<a href="#">for</a>	<a href="#">operator</a> <sup>2</sup>	<a href="#">var</a>
<a href="#">covariant</a> <sup>2</sup>	<a href="#">Function</a> <sup>2</sup>	<a href="#">part</a> <sup>2</sup>	<a href="#">void</a>
<a href="#">default</a>	<a href="#">get</a> <sup>2</sup>	<a href="#">rethrow</a>	<a href="#">while</a>
<a href="#">deferred</a> <sup>2</sup>	<a href="#">hide</a> <sup>1</sup>	<a href="#">return</a>	<a href="#">with</a>
<a href="#">do</a>	<a href="#">if</a>	<a href="#">set</a> <sup>2</sup>	<a href="#">yield</a> <sup>3</sup>

避免使用这些单词作为标识符。但是，如有必要，标有上标的关键字可以用作标识符：

- 带有 **1** 上标的单词为 **上下文关键字**，仅在特定位置具有含义。他们在任何地方都是有效的标识符。
- 带有 **2** 上标的单词为 **内置标识符**，为了简化将 JavaScript 代码移植到 Dart 的工作，这些关键字在大多数地方都是有效的标识符，但它们不能用作类或类型名称，也不能用作 import 前缀。

- 带有 **3** 上标的单词是与 Dart 1.0 发布后添加的[异步支持](#)相关的更新，作为限制类保留字。不能在标记为 `async`，`async*` 或 `sync*` 的任何函数体中使用 `await` 或 `yield` 作为标识符。

关键字表中的剩余单词都是**保留字**。不能将保留字用作标识符。

## 变量

创建一个变量并进行初始化：

```
var name = 'Bob';
```

变量仅存储对象引用，这里的变量是 `name` 存储了一个 `String` 类型的对象引用。“Bob”是这个 `String` 类型对象的值。

`name` 变量的类型被推断为 `String`。但是也可以通过指定类型的方式，来改变变量类型。如果对象不限定为单个类型，可以指定为 [对象类型](#) 或 [动态类型](#)，参考 [设计指南](#)。

```
dynamic name = 'Bob';
```

另一种方式是显式声明可以推断出的类型：

```
String name = 'Bob';
```

**提示：** 本页局部变量遵守 [风格建议指南](#) 使用 `var`。没有使用指定类型的方式。

## 默认值

未初始化的变量默认值是 `null`。即使变量是数字 类型默认值也是 `null`，因为在 Dart 中一切都是对象，数字类型 也不例外。

```
int lineCount;  
assert(lineCount == null);
```

**提示：** 在生产环境代码中 `assert()` 函数会被忽略，不会被调用。 在开发过程中, `assert(condition)` 会在非 `true` 的条件下抛出异常.有关更多信息，参考 [Assert](#)。

## Final 和 Const

使用过程中从来不会被修改的变量， 可以使用 `final` 或 `const`, 而不是 `var` 或者其他类型， `Final` 变量的值只能被设置一次； `Const` 变量在编译时就已经固定 (`Const` 变量 是隐式 `Final` 的类型.) 最高级 `final` 变量或类变量在第一次使用时被初始化。

**提示：** 实例变量可以是 `final` 类型但不能是 `const` 类型。 必须在构造函数体执行之前初始化 `final` 实例变量 —— 在变量声明中，参数构造函数中或构造函数的[初始化列表](#)中进行初始化。

创建和设置一个 `Final` 变量：

```
final name = 'Bob'; // Without a type annotation  
final String nickname = 'Bobby';
```

`final` 不能被修改：

```
name = 'Alice'; // Error: 一个 final 变量只能被设置一次。
```

X static analysis: error/warning

如果需要在**编译时**就固定变量的值，可以使用 `const` 类型变量。如果 `Const` 变量是类级别的，需要标记为 `static const`。在这些地方可以使用在编译时就已经固定不变的值，字面量的数字和字符串， 固定的变量，或者是用于计算的固定数字：

```
const bar = 1000000; // 压力单位 (dynes/cm2)
const double atm = 1.01325 * bar; // 标准气压
```

Const 关键字不仅可以用于声明常量变量。 还可以用来创建常量值，以及声明创建常量值的构造函数。 任何变量都可以拥有常量值。

```
var foo = const [];  
final bar = const [];  
const baz = []; // Equivalent to `const []`
```

声明 `const` 的初始化表达式中 `const` 可以被省略。 比如上面的 `baz`。有关更多信息，参考 [DON'T use const redundantly](#)。

非 Final， 非 const 的变量是可以被修改的，即使这些变量 曾经引用过 const 值。

```
foo = [1, 2, 3]; // 曾经引用过 const [] 常量值。
```

Const 变量的值不可以修改：

```
baz = [42]; // Error: 常量变量不能赋值修改。
```

X static analysis: error/warning

更多关于使用 `const` 创建常量值，参考 [Lists](#)， [Maps](#)， 和 [Classes](#)。

## 内建类型

Dart 语言支持以下内建类型：

- Number
- String
- Boolean
- List (也被称为 *Array*)
- Map
- Set
- Rune (用于在字符串中表示 Unicode 字符)
- Symbol

这些类型都可以被初始化为字面量。 例如, `'this is a string'` 是一个字符串的字面量， `true` 是一个布尔的字面量。

因为在 Dart 所有的变量终究是一个对象（一个类的实例）， 所以变量可以使用 *构造函数* 进行初始化。 一些内建类型拥有自己的构造函数。 例如， 通过 `Map()` 来构造一个 map 变量。

## Number

Dart 语言的 Number 有两种类型:

### int

整数值不大于64位， 具体取决于平台。 在 Dart VM 上， 值的范围从  $-2^{63}$  到  $2^{63} - 1$ . Dart 被编译为 JavaScript 时，使用 [JavaScript numbers](#), 值的范围从  $-2^{53}$  到  $2^{53} - 1$ .

### double

64位（双精度）浮点数，依据 IEEE 754 标准。

`int` 和 `double` 都是 `num` 的亚类型。 `num` 类型包括基本运算 `+`， `-`， `/`， 和 `*`， 以及 `abs()`， `ceil()`， 和 `floor()`， 等函数方法。（按位运算符，例如`>>`，定义在 `int` 类中。） 如果 `num` 及其亚类型找不到你想要的方法， 尝试查找使用 [dart:math](#) 库。

整数类型不包含小数点。 下面是定义整数类型字面量的例子：

```
var x = 1;  
var hex = 0xDEADBEEF;
```

如果一个数字包含小数点，那么就是小数类型。 下面是定义小数类型字面量的例子：

```
var y = 1.1;
var exponents = 1.42e5;
```

从 Dart 2.1 开始，必要的时候 int 字面量会自动转换成 double 类型。

```
double z = 1; // 相当于 double z = 1.0.
```

**版本提示：** 在 2.1 之前，在 double 上下文中使用 int 字面量是错误的。

以下是将字符串转换为数字的方法，反之亦然：

```
// String -> int
var one = int.parse('1');
assert(one == 1);

// String -> double
var onePointOne = double.parse('1.1');
assert(onePointOne == 1.1);

// int -> String
String oneAsString = 1.toString();
assert(oneAsString == '1');

// double -> String
String piAsString = 3.14159.toStringAsFixed(2);
assert(piAsString == '3.14');
```

int 特有的传统按位运算操作，移位 (<<, >>)，按位与 (&) 以及 按位或 (|)。例如：

```
assert((3 << 1) == 6); // 0011 << 1 == 0110
assert((3 >> 1) == 1); // 0011 >> 1 == 0001
assert((3 | 4) == 7); // 0011 | 0100 == 0111
```

数字类型字面量是编译时常量。在算术表达式中，只要参与计算的因子是编译时常量，那么算术表达式的结果也是编译时常量。

```
const msPerSecond = 1000;
const secondsUntilRetry = 5;
const msUntilRetry = secondsUntilRetry * msPerSecond;
```

## String

Dart 字符串是一组 UTF-16 单元序列。字符串通过单引号或者双引号创建。

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.";
```

字符串可以通过 `${expression}` 的方式内嵌表达式。如果表达式是一个标识符，则 `{}` 可以省略。在 Dart 中通过调用就对象的 `toString()` 方法来得到对象相应的字符串。

```
var s = 'string interpolation';

assert('Dart has $s, which is very handy.' ==
    'Dart has string interpolation, ' +
    'which is very handy.');
```

```
assert('That deserves all caps. ' +
    '${s.toUpperCase()} is very handy!' ==
    'That deserves all caps. ' +
    'STRING INTERPOLATION is very handy!');
```

**提示：** `==` 运算符用来测试两个对象是否相等。在字符串中，如果两个字符串包含了相同的编码序列，那么这两个字符串相等。

units.

可以使用 `+` 运算符来把多个字符串连接为一个，也可以把多个字面量字符串写在一起来实现字符串连接：

```
var s1 = 'String '
    'concatenation'
    " works even over line breaks.";
assert(s1 ==
    'String concatenation works even over '
    'line breaks.');
```

```
var s2 = 'The + operator ' + 'works, as well.';
assert(s2 == 'The + operator works, as well.');
```

使用连续三个单引号或者三个双引号实现多行字符串对象的创建：

```
var s1 = '''
You can create
multi-line strings like this one.
''';
```

```
var s2 = """This is also a
multi-line string.""";
```

使用 `r` 前缀，可以创建“原始 raw”字符串：

```
var s = r"In a raw string, even \n isn't special.";
```

参考 [Runes](#) 来了解如何在字符串中表达 Unicode 字符。

一个编译时常量的字面量字符串中，如果存在插值表达式，表达式内容也是编译时常量，那么该字符串依旧是编译时常量。插入的常量值类型可以是 `null`，数值，字符串或布尔值。

```
// const 类型数据
const aConstNum = 0;
const aConstBool = true;
const aConstString = 'a constant string';

// 非 const 类型数据
var aNum = 0;
var aBool = true;
var aString = 'a string';
const aConstList = [1, 2, 3];

const validConstString = '$aConstNum $aConstBool $aConstString'; //const 类型数据
// const invalidConstString = '$aNum $aBool $aString $aConstList'; //非 const 类型数据
```

更多关于 `string` 的使用, 参考 [字符串和正则表达式](#).



## Boolean

Dart 使用 `bool` 类型表示布尔值。 Dart 只有字面量 `true` and `false` 是布尔类型， 这两个对象都是编译时常量。

Dart 的类型安全意味着不能使用 `if (nonbooleanValue)` 或者 `assert (nonbooleanValue)`。而是应该像下面这样，明确的进行值检查：

```
// 检查空字符串。
var fullName = '';
assert(fullName.isEmpty);

// 检查 0 值。
var hitPoints = 0;
assert(hitPoints <= 0);

// 检查 null 值。
var unicorn;
assert(unicorn == null);

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

## List

几乎每种编程语言中最常见的集合可能是 *array* 或有序的对象集合。 在 Dart 中的 *Array* 就是 [List](#) 对象， 通常称之为 *List* 。

Dart 中的 List 字面量非常像 JavaScript 中的 array 字面量。 下面是一个 Dart List 的示例：

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

**提示：** Dart 推断 `list` 的类型为 `List<int>`。如果尝试将非整数对象添加到此 List 中， 则分析器或运行时会引发错误。 有关更多信息，请阅读 [类型推断](#)。

Lists 的下标索引从 0 开始，第一个元素的索引是 0。`list.length - 1` 是最后一个元素的索引。 访问 List 的长度和元素与 JavaScript 中的用法一样：

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

list[1] = 1;
assert(list[1] == 1);
```

在 List 字面量之前添加 `const` 关键字，可以定义 List 类型的编译时常量：

```
var constantList = const [1, 2, 3];
// constantList[1] = 1; // 取消注释会引起错误。
```

List 类型包含了很多 List 的操作函数。 更多信息参考 [泛型](#) 和 [集合](#)。

## Set

在 Dart 中 Set 是一个元素唯一且无需的集合。 Dart 为 Set 提供了 Set 字面量和 [Set](#) 类型。

**版本提示：** 虽然 Set 类型一直是 Dart 的核心部分， 但在 Dart2.2 中才引入了 Set 字面量。

下面是通过字面量创建 Set 的一个简单示例：

```
var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};
```

**Note:** Dart 推断 `halogens` 类型为 `Set<String>`。如果尝试为它添加一个 错误类型的值，分析器或执行时会抛出错误。更多内容，参阅 [类型推断](#)。

要创建一个空集，使用前面带有类型参数的 `{}`，或者将 `{}` 赋值给 `Set` 类型的变量：

```
var names = <String>{};
// Set<String> names = {}; // 这样也是可以的。
// var names = {}; // 这样会创建一个 Map，而不是 Set。
```

**是 Set 还是 Map ？** Map 字面量语法同 Set 字面量语法非常相似。因为先有的 Map 字母量语法，所以 `{}` 默认是 Map 类型。如果忘记在 `{}` 上注释类型或赋值到一个未声明类型的变量上，那么 Dart 会创建一个类型为 `Map<dynamic, dynamic>` 的对象。

使用 `add()` 或 `addAll()` 为已有的 Set 添加元素：

```
var elements = <String>{};
elements.add('fluorine');
elements.addAll(halogens);
```

使用 `.length` 来获取 Set 中元素的个数：

```
var elements = <String>{};
elements.add('fluorine');
elements.addAll(halogens);
assert(elements.length == 5);
```

在 Set 字面量前增加 `const`，来创建一个编译时 Set 常量：

```
final constantSet = const {
  'fluorine',
  'chlorine',
  'bromine',
  'iodine',
  'astatine',
};
// constantSet.add('helium'); // Uncommenting this causes an error.
```

更多关于 Set 的内容，参阅 [Generic](#) 及 [Set](#)。

## Map

通常来说，Map 是用来关联 keys 和 values 的对象。keys 和 values 可以是任何类型的对象。在一个 Map 对象中一个 key 只能出现一次。但是 value 可以出现多次。Dart 中 Map 通过 Map 字面量和 [Map](#) 类型来实现。

下面是使用 Map 字面量的两个简单例子：

```
var gifts = {
  // Key:    Value
  'first': 'partridge',
  'second': 'turtledoves',
  'fifth': 'golden rings'
};

var nobleGases = {
  2: 'helium',
  10: 'neon',
  18: 'argon',
};
```



**提示：** Dart 会将 `gifts` 的类型推断为 `Map<String, String>`，`nobleGases` 的类型推断为 `Map<int, String>`。如果尝试在上面的 `map` 中添加错误类型，那么分析器或者运行时会引发错误。有关更多信息，请阅读[类型推断](#)。

以上 `Map` 对象也可以使用 `Map` 构造函数创建：

```
var gifts = Map();
gifts['first'] = 'partridge';
gifts['second'] = 'turtledoves';
gifts['fifth'] = 'golden rings';

var nobleGases = Map();
nobleGases[2] = 'helium';
nobleGases[10] = 'neon';
nobleGases[18] = 'argon';
```

**提示：** 这里为什么只有 `Map()`，而不是使用 `new Map()`。因为在 Dart 2 中，`new` 关键字是可选的。有关更多信息，参考[构造函数的使用](#)。

类似 JavaScript，添加 key-value 对到已有的 `Map` 中：

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds'; // Add a key-value pair
```

类似 JavaScript，从一个 `Map` 中获取一个 `value`：

```
var gifts = {'first': 'partridge'};
assert(gifts['first'] == 'partridge');
```

如果 `Map` 中不包含所要查找的 `key`，那么 `Map` 返回 `null`：

```
var gifts = {'first': 'partridge'};
assert(gifts['fifth'] == null);
```

使用 `.length` 函数获取当前 `Map` 中的 key-value 对数量：

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds';
assert(gifts.length == 2);
```

创建 `Map` 类型运行时常量，要在 `Map` 字面量前加上关键字 `const`。

```
final constantMap = const {
  2: 'helium',
  10: 'neon',
  18: 'argon',
};

// constantMap[2] = 'Helium'; // 取消注释会引起错误。
```

更多关于 `Map` 的内容，参考[Generics](#) and [Maps](#)。


## Rune

在 Dart 中，`Rune` 用来表示字符串中的 UTF-32 编码字符。

Unicode 定义了一个全球的书写系统编码，系统中使用的所有字母，数字和符号都对应唯一的数值编码。由于 Dart 字符串是一系列 UTF-16 编码单元，因此要在字符串中表示 32 位 Unicode 值需要特殊语法支持。

表示 Unicode 编码的常用方法是， `\uXXXX`, 这里 XXXX 是一个4位的16进制数。 例如， 心形符号 (♥) 是 `\u2665`。对于特殊的非 4 个数值的情况， 把编码值放到大括号中即可。 例如， emoji 的笑脸 (📄) 是 `\u{1f600}`。

`String` 类有一些属性可以获得 rune 数据。 属性 `codeUnitAt` 和 `codeUnit` 返回16位编码数据。 属性 `runes` 获取字符串中的 Rune 。

下面是示例演示了 Rune 、 16-bit code units、 和 32-bit code points 之间的关系。 点击运行按钮  查看 runes 结果。

Dart

Install SDKFormatResetplay\_arrow

```
x
1

1
```

file\_copylaunch

Console

no issues

**提示：** 谨慎使用 list 方式操作 Rune 。这种方法很容易引发崩溃， 具体原因取决于特定的语言， 字符集和操作。 有关更多信息， 参考 [How do I reverse a String in Dart? on Stack Overflow](#).

## Symbol

一个 Symbol 对象表示 Dart 程序中声明的运算符或者标识符。 你也许永远都不需要使用 Symbol ， 但要按名称引用标识符的 API 时， Symbol 就非常有用了。 因为代码压缩后会改变标识符的名称， 但不会改变标识符的符号。 通过字面量 Symbol ， 也就是标识符前面添加一个 `#` 号， 来获取标识符的 Symbol 。

```
#radix
#bar
```

Symbol 字面量是编译时常量。

## 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。 也可以把 Dart 类的实例当做方法来调用。 有关更多信息， 参考 [Callable classes](#).

已下是函数实现的示例：

```
bool isNoble(int atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}
```

虽然在 Effective Dart 中推荐 [公共API中声明类型](#), 但是省略了类型声明， 函数依旧是可以正常使用的：

```
isNoble(atomicNumber) {
  return _nobleGases[atomicNumber] != null;
}
```

如果函数中只有一句表达式， 可以使用简写语法：

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

`=> expr` 语法是 `{ return expr; }` 的简写。 `=>` 符号 有时也被称为 *箭头* 语法。

**提示：** 在箭头 (=>) 和分号 (;) 之间只能使用一个 *表达式*，不能是 *语句*。例如：不能使用 [if 语句](#)，但是可以用 [条件表达式](#)。

函数有两种参数类型: 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”。参考下一章节，了解更多细节。

## 可选参数

可选参数可以是命名参数或者位置参数，但一个参数只能选择其中一种方式修饰。

### 命名可选参数

调用函数时，可以使用指定命名参数 *paramName: value*。例如：

```
enableFlags(bold: true, hidden: false);
```

定义函数是，使用 *{param1, param2, ...}* 来指定命名参数：

```
/// Sets the [bold] and [hidden] flags ...  
void enableFlags({bool bold, bool hidden}) {...}
```

[Flutter](#) 创建实例的表达式可能很复杂， 因此窗口小部件构造函数仅使用命名参数。 这样创建实例的表达式更易于阅读。

使用 [@required](#) 注释表示参数是 *required* 性质的命名参数， 该方式可以在任何 Dart 代码中使用（不仅仅是Flutter）。

```
const Scrollbar({Key key, @required Widget child})
```

此时 [Scrollbar](#) 是一个构造函数， 当 *child* 参数缺少时，分析器会提示错误。

[Required](#) 被定义在 [meta](#) package。无论是直接引入 (import) `package:meta/meta.dart`， 或者引入了其他 package，而这个 package 输出 (export) 了 `meta`，比如 Flutter 的 `package:flutter/material.dart`。

### 位置可选参数

将参数放到 `[]` 中来标记参数是可选的：

```
String say(String from, String msg, [String device]) {  
  var result = '$from says $msg';  
  if (device != null) {  
    result = '$result with a $device';  
  }  
  return result;  
}
```

下面是不使用可选参数调用上面方法 的示例：

```
assert(say('Bob', 'Howdy') == 'Bob says Howdy');
```

下面是使用可选参数调用上面方法的示例：

```
assert(say('Bob', 'Howdy', 'smoke signal') ==  
       'Bob says Howdy with a smoke signal');
```

### 默认参数值

在定义方法的时候，可以使用 `=` 来定义可选参数的默认值。默认值只能是编译时常量。 如果没有提供默认值，则默认值为 null。

下面是设置可选参数默认值示例：

```
/// 设置 [bold] 和 [hidden] 标志 ...
void enableFlags({bool bold = false, bool hidden = false}) {...}

// bold 值为 true; hidden 值为 false.
enableFlags(bold: true);
```

**不推荐：**旧版本代码中可能使用的是冒号 (:) 而不是 = 来设置参数默认值。原因是起初命名参数只支持 :。这种支持可能会被弃用。建议 [使用 = 指定默认值。](#)

下面示例演示了如何为位置参数设置默认值：

```
String say(String from, String msg,
  [String device = 'carrier pigeon', String mood]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  if (mood != null) {
    result = '$result (in a $mood mood)';
  }
  return result;
}

assert(say('Bob', 'Howdy') ==
  'Bob says Howdy with a carrier pigeon');
```

list 或 map 可以作为默认值传递。下面的示例定义了一个方法 `doStuff()`，并分别指定参数 `list` 和 `gifts` 的默认值。

```
void doStuff(
  {List<int> list = const [1, 2, 3],
  Map<String, String> gifts = const {
    'first': 'paper',
    'second': 'cotton',
    'third': 'leather'
  }}) {
  print('list: $list');
  print('gifts: $gifts');
}
```

## main() 函数

任何应用都必须有一个顶级 `main()` 函数，作为应用服务的入口。 `main()` 函数返回值为空，参数为一个可选的 `List<String>`。

下面是 web 应用的 `main()` 函数：

```
void main() {
  querySelector('#sample_text_id')
    ..text = 'Click me!'
    ..onClick.listen(reverseText);
}
```

### 提示：

以上代码中的 `..` 语法为 [级联调用](#)（cascade）。使用级联调用，可以简化在一个对象上执行的多个操作。

下面是一个命令行应用的 `main()` 方法，并且使用了输入参数：

```
// 这样运行应用: dart args.dart 1 test
void main(List<String> arguments) {
  print(arguments);

  assert(arguments.length == 2);
  assert(int.parse(arguments[0]) == 1);
  assert(arguments[1] == 'test');
}
```

使用 [args library](#) 可以定义和解析命令行参数。

## 函数是一等对象

一个函数可以作为另一个函数的参数。 例如：

```
void printElement(int element) {
  print(element);
}

var list = [1, 2, 3];

// 将 printElement 函数作为参数传递。
list.forEach(printElement);
```

同样可以将一个函数赋值给一个变量，例如：

```
var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';
assert(loudify('hello') == '!!! HELLO !!!');
```

示例中使用了匿名函数。 下一章节会有更多介绍。

## 匿名函数

多数函数是有名字的， 比如 `main()` 和 `printElement()`。也可以创建没有名字的函数，这种函数被称为 *匿名函数*， 有时候也被称为 *lambda* 或者 *closure*。匿名函数可以赋值到一个变量中， 举个例子，在一个集合中可以添加或者删除一个匿名函数。

匿名函数和命名函数看起来类似— 在括号之间可以定义一些参数或可选参数，参数使用逗号分割。

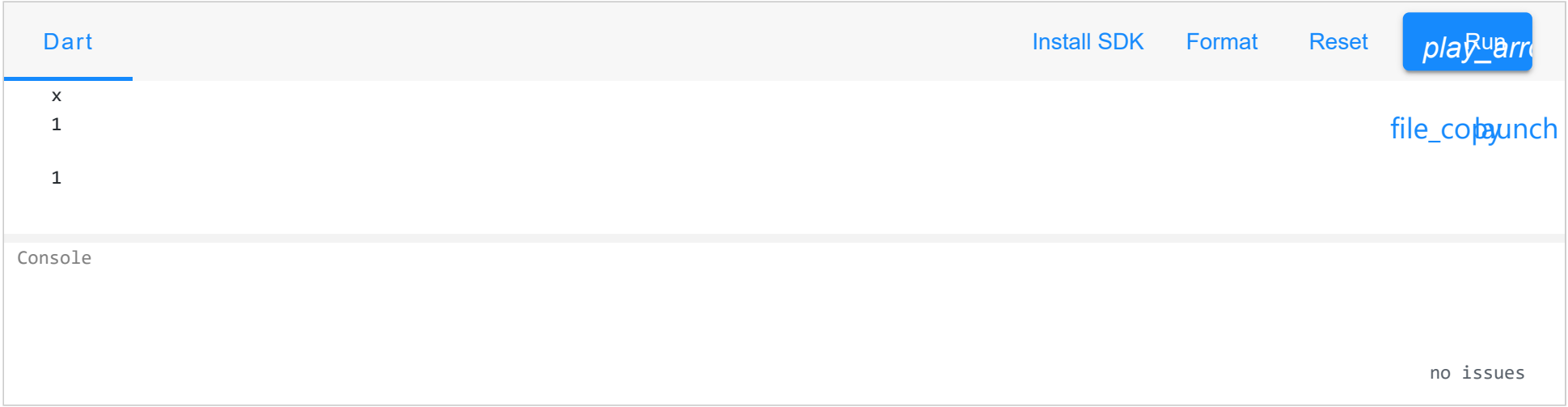
后面大括号中的代码为函数体：

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

下面例子中定义了一个包含一个无类型参数 `item` 的匿名函数。 `list` 中的每个元素都会调用这个函数，打印元素位置和值的字符串。

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

点击运行按钮  执行代码。



如果函数只有一条语句， 可以使用箭头简写。粘贴下面代码到 DartPad 中 并点击运行按钮， 验证两个函数是等价性。

```
list.forEach(  
  (item) => print('${list.indexOf(item)}: $item');
```

## 词法作用域

Dart 是一门词法作用域的编程语言， 就意味着变量的作用域是固定的， 简单说变量的作用域在编写代码的时候就已经确定了。 花括号内的是变量可见的作用域。

下面示例关于多个嵌套函数的变量作用域：

```
bool topLevel = true;  
  
void main() {  
  var insideMain = true;  
  
  void myFunction() {  
    var insideFunction = true;  
  
    void nestedFunction() {  
      var insideNestedFunction = true;  
  
      assert(topLevel);  
      assert(insideMain);  
      assert(insideFunction);  
      assert(insideNestedFunction);  
    }  
  }  
}
```

注意 `nestedFunction()` 可以访问所有的变量， 一直到顶级作用域变量。

## 词法闭包

*闭包* 即一个函数对象， 即使函数对象的调用在它原始作用域之外， 依然能够访问在它词法作用域内的变量。

函数可以封闭定义到它作用域内的变量。 接下来的示例中， `makeAdder()` 捕获了变量 `addBy`。无论在什么时候执行返回函数， 函数都会使用捕获的 `addBy` 变量。



```
/// 返回一个函数，返回的函数参数与 [addBy] 相加。
Function makeAdder(num addBy) {
  return (num i) => addBy + i;
}

void main() {
  // 创建一个加 2 的函数。
  var add2 = makeAdder(2);

  // 创建一个加 4 的函数。
  var add4 = makeAdder(4);

  assert(add2(3) == 5);
  assert(add4(3) == 7);
}
```

## 测试函数是否相等

下面是顶级函数，静态方法和示例方法相等性的测试示例：

```
void foo() {} // 顶级函数

class A {
  static void bar() {} // 静态方法
  void baz() {} // 示例方法
}

void main() {
  var x;

  // 比较顶级函数。
  x = foo;
  assert(foo == x);

  // 比较静态方法。
  x = A.bar;
  assert(A.bar == x);

  // 比较实例方法。
  var v = A(); // A的1号实例
  var w = A(); // A的2号实例
  var y = w;
  x = w.baz;

  // 两个闭包引用的同一实例（2号），
// 所以它们相等。
  assert(y.baz == x);

  // 两个闭包引用的非同一个实例，
// 所以它们不相等。
  assert(v.baz != w.baz);
}
```

## 返回值

所有函数都会返回一个值。如果没有明确指定返回值，函数体会被隐式的添加 `return null;` 语句。

```
foo() {}

assert(foo() == null);
```

## 运算符

下表是 Dart 定义的运算符。多数运算符可以被重载，详情参考 [重写运算符](#)。

Description	Operator
unary postfix	<code>expr++</code> <code>expr--</code> <code>()</code> <code>[]</code> <code>.</code> <code>?.</code>
unary prefix	<code>-expr</code> <code>!expr</code> <code>~expr</code> <code>++expr</code> <code>--expr</code>
multiplicative	<code>*</code> <code>/</code> <code>%</code> <code>~/</code>
additive	<code>+</code> <code>-</code>
shift	<code>&lt;&lt;</code> <code>&gt;&gt;</code> <code>&gt;&gt;&gt;</code>
bitwise AND	<code>&amp;</code>
bitwise XOR	<code>^</code>
bitwise OR	<code> </code>
relational and type test	<code>&gt;=</code> <code>&gt;</code> <code>&lt;=</code> <code>&lt;</code> <code>as</code> <code>is</code> <code>is!</code>
equality	<code>==</code> <code>!=</code>
logical AND	<code>&amp;&amp;</code>
logical OR	<code>  </code>
if null	<code>??</code>
conditional	<code>expr1 ? expr2 : expr3</code>
cascade	<code>..</code>
assignment	<code>=</code> <code>*=</code> <code>/=</code> <code>+=</code> <code>-=</code> <code>&amp;=</code> <code>^=</code> <i>etc.</i>

**提示：** 上述表格中描述的运算符优先级近似于Dart 解析器实际行为。 更准确的描述，请参阅 [Dart language specification](#) 中的语法。

创建表达式的时候会用到运算符。 下面是一些运算符表达式的实例：

```
a++
a + b
a = b
a == b
c ? a : b
a is T
```

在 [运算符表](#) 中， 每一行的运算符优先级， 由上到下依次排列， 第一行优先级最高， 最后一行优先级最低。 例如 `%` 运算符优先级高于 `==`， 而 `==` 高于 `&&`。根据优先级规则， 那么意味着以下两行代码执行的方式相同：

```
// 括号可以提高可读性。
if ((n % i == 0) && (d % i == 0)) ...

// 可读性差，但是是等效的。
if (n % i == 0 && d % i == 0) ...
```

**警告：** 对于有两个操作数的运算符， 运算符的功能由左边的操作数决定。 例如, 如果有两个操作数 `Vector` 和 `Point`, `aVector + aPoint` 使用的是 `Vector` 中定义的 `+` 运算符。

# 算术运算符

Dart 支持常用的运算运算符，如下表所示：

Operator	Meaning
<code>+</code>	Add
<code>-</code>	Subtract
<code>-expr</code>	Unary minus, also known as negation (reverse the sign of the expression)
<code>*</code>	Multiply
<code>/</code>	Divide
<code>~/</code>	Divide, returning an integer result
<code>%</code>	Get the remainder of an integer division (modulo)

示例：

```
assert(2 + 3 == 5);
assert(2 - 3 == -1);
assert(2 * 3 == 6);
assert(5 / 2 == 2.5); // 结果是双浮点型
assert(5 ~/ 2 == 2); // 结果是整型
assert(5 % 2 == 1); // 余数

assert('5/2 = ${5 ~/ 2} r ${5 % 2}' == '5/2 = 2 r 1');
```

Dart 还支持前缀和后缀，自增和自减运算符。

Operator	Meaning
<code>++var</code>	<code>var = var + 1</code> (expression value is <code>var + 1</code> )
<code>var++</code>	<code>var = var + 1</code> (expression value is <code>var</code> )
<code>--var</code>	<code>var = var - 1</code> (expression value is <code>var - 1</code> )
<code>var--</code>	<code>var = var - 1</code> (expression value is <code>var</code> )

示例：

```
var a, b;

a = 0;
b = ++a; // a自加后赋值给b。
assert(a == b); // 1 == 1

a = 0;
b = a++; // a先赋值给b后, a自加。
assert(a != b); // 1 != 0

a = 0;
b = --a; // a自减后赋值给b。
assert(a == b); // -1 == -1

a = 0;
b = a--; // a先赋值给b后, a自减。
assert(a != b); // -1 != 0
```

# 关系运算符

下表列出了关系运算符及含义：

Operator	Meaning
==	Equal; see discussion below
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

要测试两个对象x和y是否表示相同的事物， 使用 == 运算符。(在极少数情况下， 要确定两个对象是否完全相同， 需要使用 [identical\(\)](#) 函数。) 下面给出 == 运算符的工作原理：

1. 如果 x 或 y 可以 null，都为 null 时返回 true ， 其中一个为 null 时返回 false。
2. 结果为函数 `x.==(y)` 的返回值。(如上所见, == 运算符执行的是第一个运算符的函数。 我们甚至可以重写很多运算符，包括 ==， 运算符的重写，参考 [重写运算符](#)。)

这里列出了每种关系运算符的示例：

```
assert(2 == 2);
assert(2 != 3);
assert(3 > 2);
assert(2 < 3);
assert(3 >= 3);
assert(2 <= 3);
```

# 类型判定运算符

as， is， 和 is! 运算符用于在运行时处理类型检查：

Operator	Meaning
as	Typecast (也被用于 <a href="#">指定库前缀</a> )
is	True if the object has the specified type
is!	False if the object has the specified type

例如， `obj is Object` 总是 true。但是只有 `obj` 实现了 `T` 的接口时， `obj is T` 才是 true。

使用 `as` 运算符将对象强制转换为特定类型。 通常，可以认为是 `is` 类型判定后， 被判定对象调用函数的一种缩写形式。 请考虑以下代码：

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

使用 `as` 运算符进行缩写：

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

**提示：** 以上代码并不是等价的。如果 `emp` 为 `null` 或者不是 `Person` 对象，那么第一个 `is` 的示例，后面将不回执行；第二个 `as` 的示例会抛出异常。

## 赋值运算符

使用 `=` 为变量赋值。使用 `??=` 运算符时，只有当被赋值的变量为 `null` 时才会赋值给它。

```
// 将值赋值给变量a
a = value;
// 如果b为空时，将变量赋值给b，否则，b的值保持不变。
b ??= value;
```

复合赋值运算符（如 `+=`）将算术运算符和赋值运算符组合在了一起。

<code>=</code>	<code>--=</code>	<code>/=</code>	<code>%=</code>	<code>&gt;&gt;=</code>	<code>^=</code>
<code>+=</code>	<code>*=</code>	<code>~/=</code>	<code>&lt;&lt;=</code>	<code>&amp;=</code>	<code> =</code>

以下说明复合赋值运算符的作用：

	Compound assignment	Equivalent expression
For an operator <i>op</i> :	<code>a op= b</code>	<code>a = a op b</code>
Example:	<code>a += b</code>	<code>a = a + b</code>

以下示例使用赋值和复合赋值运算符：

```
var a = 2; // 使用 = 复制
a *= 3; // 复制并做乘法运算： a = a * 3
assert(a == 6);
```

## 逻辑运算符

逻辑操作符可以反转或组合布尔表达式。

Operator	Meaning
<code>!expr</code>	inverts the following expression (changes false to true, and vice versa)
<code>  </code>	logical OR
<code>&amp;&amp;</code>	logical AND

下面是关于逻辑表达式的示例：

```
if (!done && (col == 0 || col == 3)) {
  // ...Do something...
}
```

## 按位和移位运算符

在 Dart 中，可以单独操作数字的某一位。通常情况下整数类型使用按位和移位运算符来操作。

Operator	Meaning
----------	---------

Operator	Meaning
&	AND
	OR
^	XOR
<code>~expr</code>	Unary bitwise complement (0s become 1s; 1s become 0s)
<<	Shift left
>>	Shift right

下面是关于按位和移位运算符的示例：

```
final value = 0x22;
final bitmask = 0x0f;

assert((value & bitmask) == 0x02); // AND
assert((value & ~bitmask) == 0x20); // AND NOT
assert((value | bitmask) == 0x2f); // OR
assert((value ^ bitmask) == 0x2d); // XOR
assert((value << 4) == 0x220); // Shift left
assert((value >> 4) == 0x02); // Shift right
```

## 条件表达式

Dart有两个运算符，有时可以替换 [if-else](#) 表达式， 让表达式更简洁：

**`condition ? expr1 : expr2`**

如果条件为 true, 执行 `expr1` (并返回它的值)； 否则, 执行并返回 `expr2` 的值。

**`expr1 ?? expr2`**

如果 `expr1` 是 non-null, 返回 `expr1` 的值； 否则, 执行并返回 `expr2` 的值。

如果赋值是根据布尔值， 考虑使用 `?:`。

```
var visibility = isPublic ? 'public' : 'private';
```

如果赋值是基于判定是否为 null， 考虑使用 `??`。

```
String playerName(String name) => name ?? 'Guest';
```

下面给出了其他两种实现方式， 但并不简洁：

```
// Slightly longer version uses ?: operator.
String playerName(String name) => name != null ? name : 'Guest';

// Very long version uses if-else statement.
String playerName(String name) {
  if (name != null) {
    return name;
  } else {
    return 'Guest';
  }
}
```

## 级联运算符 (..)



级联运算符 (`..`) 可以实现对同一个对像进行一系列的操作。除了调用函数， 还可以访问同一对象上的字段属性。这通常可以节省创建临时变量的步骤， 同时编写出更流畅的代码。

考虑一下代码：

```
querySelector('#confirm') // 获取对象。
  ..text = 'Confirm' // 调用成员变量。
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'));
```

第一句调用函数 `querySelector()`， 返回获取到的对象。获取的对象依次执行级联运算符后面的代码， 代码执行后的返回值会被忽略。

上面的代码等价于：

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

级联运算符可以嵌套，例如：

```
final addressBook = (AddressBookBuilder()
  ..name = 'jenny'
  ..email = 'jenny@example.com'
  ..phone = (PhoneNumberBuilder()
    ..number = '415-555-0100'
    ..label = 'home')
  .build())
.build());
```

在返回对象的函数中谨慎使用级联操作符。 例如，下面的代码是错误的：

```
var sb = StringBuffer();
sb.write('foo')
  ..write('bar'); // Error: 'void' 没哟定义 'write' 函数。
```

`sb.write()` 函数调用返回 `void`， 不能在 `void` 对象上创建级联操作。

**提示：** 严格的来讲，“两个点”的级联语法不是一个运算符。它只是一个 Dart 的特殊语法。

## 其他运算符

大多数剩余的运算符，已在示例中使用过：

Operator	Name	Meaning
<code>()</code>	Function application	Represents a function call
<code>[]</code>	List access	Refers to the value at the specified index in the list
<code>.</code>	Member access	Refers to a property of an expression; example: <code>foo.bar</code> selects property <code>bar</code> from expression <code>foo</code>
<code>?.</code>	Conditional member access	Like <code>.</code> , but the leftmost operand can be null; example: <code>foo?.bar</code> selects property <code>bar</code> from expression <code>foo</code> unless <code>foo</code> is null (in which case the value of <code>foo?.bar</code> is null)

更多关于 `.`, `?.` 和 `..` 运算符介绍，参考 [Classes](#).

# 控制流程语句

你可以通过下面任意一种方式来控制 Dart 程序流程：

- `if` and `else`
- `for` loops
- `while` and `do-while` loops
- `break` and `continue`
- `switch` and `case`
- `assert`

使用 `try-catch` 和 `throw` 也可以改变程序流程， 详见 [Exceptions](#)。

## if 和 else

Dart 支持 `if - else` 语句，其中 `else` 是可选的， 比如下面的例子， 另参考 [conditional expressions](#).

```
if (isRaining()) {
  you.bringRainCoat();
} else if (isSnowing()) {
  you.wearJacket();
} else {
  car.putTopDown();
}
```

和 JavaScript 不同， Dart 的判断条件必须是布尔值，不能是其他类型。 更多信息，参考 [Booleans](#) 。

## for 循环

进行迭代操作，可以使用标准 `for` 语句。 例如：

```
var message = StringBuffer('Dart is fun');
for (var i = 0; i < 5; i++) {
  message.write('!');
}
```

闭包在 Dart 的 `for` 循环中会捕获循环的 `index` 索引值， 来避免 JavaScript 中常见的陷阱。 请思考示例代码：

```
var callbacks = [];
for (var i = 0; i < 2; i++) {
  callbacks.add(() => print(i));
}
callbacks.forEach((c) => c());
```

和期望一样，输出的是 `0` 和 `1`。但是示例中的代码在 JavaScript 中会连续输出两个 `2`。

如果要迭代一个实现了 `Iterable` 接口的对象， 可以使用 [forEach\(\)](#) 方法， 如果不需要使用当前计数值， 使用 `forEach()` 是非常棒的选择；

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

实现了 `Iterable` 的类（比如， `List` 和 `Set`）同样也支持使用 `for-in` 进行迭代操作 [iteration](#)：

```
var collection = [0, 1, 2];
for (var x in collection) {
  print(x); // 0 1 2
}
```

## while 和 do-while

`while` 循环在执行前判断执行条件：

```
while (!isDone()) {  
    doSomething();  
}
```

`do-while` 循环在执行后判断执行条件：

```
do {  
    printLine();  
} while (!atEndOfPage());
```

## break 和 continue

使用 `break` 停止程序循环：

```
while (true) {  
    if (shutdownRequested()) break;  
    processIncomingRequests();  
}
```

使用 `continue` 跳转到下一次迭代：

```
for (int i = 0; i < candidates.length; i++) {  
    var candidate = candidates[i];  
    if (candidate.yearsExperience < 5) {  
        continue;  
    }  
    candidate.interview();  
}
```

如果对象实现了 [Iterable](#) 接口（例如，`list` 或者 `set`）。那么上面示例完全可以用另一种方式来实现：

```
candidates  
    .where((c) => c.yearsExperience >= 5)  
    .forEach((c) => c.interview());
```

## switch 和 case

在 Dart 中 `switch` 语句使用 `==` 比较整数，字符串，或者编译时常量。比较的对象必须都是同一个类的实例（并且不可以是子类），类必须没有对 `==` 重写。 [枚举类型](#) 可以用于 `switch` 语句。

**提示：** 在 Dart 中 `Switch` 语句仅适用于有限的情况下，例如在 `interpreter` 或 `scanner` 中。

在 `case` 语句中，每个非空的 `case` 语句结尾需要跟一个 `break` 语句。除 `break` 以外，还有可以使用 `continue`, `throw`, 者 `return`。

当没有 `case` 语句匹配时，执行 `default` 代码：

```

var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'PENDING':
    executePending();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'DENIED':
    executeDenied();
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown();
}

```

下面的 `case` 程序示例中缺省了 `break` 语句，导致错误：

```

var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // ERROR: 丢失 break

  case 'CLOSED':
    executeClosed();
    break;
}

```

但是，Dart 支持空 `case` 语句，允许程序以 fall-through 的形式执行。

```

var command = 'CLOSED';
switch (command) {
  case 'CLOSED': // Empty case falls through.
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}

```

在非空 `case` 中实现 fall-through 形式，可以使用 `continue` 语句结合 `lable` 的方式实现：

```

var command = 'CLOSED';
switch (command) {
  case 'CLOSED':
    executeClosed();
    continue nowClosed;
  // Continues executing at the nowClosed label.

  nowClosed:
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
}

```

`case` 语句可以拥有局部变量，这些局部变量只能在这个语句的作用域中可见。

## assert

如果 `assert` 语句中的布尔条件为 `false`，那么正常的程序执行流程会被中断。在本章中包含部分 `assert` 的使用，下面是一些示例：

```
// 确认变量值不为空。  
assert(text != null);  
  
// 确认变量值小于100。  
assert(number < 100);  
  
// 确认 URL 是否是 https 类型。  
assert(urlString.startsWith('https'));
```

**提示：** `assert` 语句只在开发环境中有效，在生产环境是无效的；Flutter 中的 `assert` 只在 [debug 模式](#) 中有效。开发用的工具，例如 [dartdevc](#) 默认是开启 `assert` 功能。其他的一些工具，例如 [dart](#) 和 [dart2js](#)，支持通过命令行开启 `assert`：`--enable-asserts`。

`assert` 的第二个参数可以为其添加一个字符串消息。

```
assert(urlString.startsWith('https'),  
      'URL ($urlString) should start with "https".');
```

`assert` 的第一个参数可以是解析为布尔值的任何表达式。如果表达式结果为 `true`，则断言成功，并继续执行。如果表达式结果为 `false`，则断言失败，并抛出异常 ([AssertionError](#))。

## 异常

Dart 代码可以抛出和捕获异常。异常表示一些未知的错误情况。如果异常没有被捕获，则异常会抛出，导致抛出异常的代码终止执行。

和 Java 有所不同，Dart 中的所有异常是非检查异常。方法不会声明它们抛出的异常，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。当然也可以定义自己的异常类型。但是，此外 Dart 程序可以抛出任何非 `null` 对象，不仅限 `Exception` 和 `Error` 对象。

## throw

下面是关于抛出或者 引发 异常的示例：

```
throw FormatException('Expected at least 1 section');
```

也可以抛出任意的对象：

```
throw 'Out of llamas!';
```

**提示：** 高质量的生产环境代码通常会实现 [Error](#) 或 [Exception](#) 类型的异常抛出。

因为抛出异常是一个表达式，所以可以在 `=>` 语句中使用，也可以在其他使用表达式的地方抛出异常：

```
void distanceTo(Point other) => throw UnimplementedError();
```

## catch

捕获异常可以避免异常继续传递（除非重新抛出（`rethrow`）异常）。可以通过捕获异常的机会来处理该异常：

```
try {
    breedMoreLlamas();
} on OutOfLlamasException {
    buyMoreLlamas();
}
```

通过指定多个 catch 语句，可以处理可能抛出多种类型异常的代码。与抛出异常类型匹配的第一个 catch 语句处理异常。如果 catch 语句未指定类型，则该语句可以处理任何类型的抛出对象：

```
try {
    breedMoreLlamas();
} on OutOfLlamasException {
    // 一个特殊的异常
    buyMoreLlamas();
} on Exception catch (e) {
    // 其他任何异常
    print('Unknown exception: $e');
} catch (e) {
    // 没有指定的类型，处理所有异常
    print('Something really unknown: $e');
}
```

如上述代码所示，捕获语句中可以同时使用 on 和 catch，也可以单独分开使用。使用 on 来指定异常类型，使用 catch 来捕获异常对象。

catch() 函数可以指定1到2个参数，第一个参数为抛出的异常对象，第二个为堆栈信息（一个 [StackTrace](#) 对象）。

```
try {
    // ...
} on Exception catch (e) {
    print('Exception details:\n $e');
} catch (e, s) {
    print('Exception details:\n $e');
    print('Stack trace:\n $s');
}
```

如果仅需要部分处理异常，那么可以使用关键字 **rethrow** 将异常重新抛出。

```
void misbehave() {
    try {
        dynamic foo = true;
        print(foo++); // Runtime error
    } catch (e) {
        print('misbehave() partially handled ${e.runtimeType}.');
        rethrow; // Allow callers to see the exception.
    }
}

void main() {
    try {
        misbehave();
    } catch (e) {
        print('main() finished handling ${e.runtimeType}.');
    }
}
```

## finally

不管是否抛出异常，finally 中的代码都会被执行。如果 catch 没有匹配到异常，异常会在 finally 执行完成后，再次被抛出：



```
try {
  breedMoreLlamas();
} finally {
  // Always clean up, even if an exception is thrown.
  cleanLlamaStalls();
}
```

任何匹配的 `catch` 执行完成后，再执行 `finally`：

```
try {
  breedMoreLlamas();
} catch (e) {
  print('Error: $e'); // Handle the exception first.
} finally {
  cleanLlamaStalls(); // Then clean up.
}
```

更多详情，请参考 [Exceptions](#) 章节。

## 类

Dart 是一种基于类和 mixin 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](#)。基于 \*Mixin 继承\* 意味着每个类（除 Object 外）都只有一个超类，一个类中的代码可以在其他多个继承类中重复使用。

### 使用类的成员变量

对象的由函数和数据（即方法和实例变量）组成。方法的调用要通过对象来完成：调用的方法可以访问其对象的其他函数和数据。

使用 `(.)` 来引用实例对象的变量和方法：

```
var p = Point(2, 2);

// 为实例的变量 y 设置值。
p.y = 3;

// 获取变量 y 的值。
assert(p.y == 3);

// 调用 p 的 distanceTo() 方法。
num distance = p.distanceTo(Point(4, 4));
```

使用 `?.` 来代替 `.`，可以避免因为左边对象可能为 `null`，导致的异常：

```
// 如果 p 为 non-null, 设置它变量 y 的值为 4。
p?.y = 4;
```

### 使用构造函数

通过 `构造函数` 创建对象。构造函数的名字可以是 `ClassName` 或者 `ClassName.identifier`。例如，以下代码使用 `Point` 和 `Point.fromJson()` 构造函数创建 `Point` 对象：

```
var p1 = Point(2, 2);
var p2 = Point.fromJson({'x': 1, 'y': 2});
```

以下代码具有相同的效果，但是构造函数前面的 `new` 关键字是可选的：

```
var p1 = new Point(2, 2);
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

**版本提示：** 在 Dart 2 中 `new` 关键字变成了可选的。

一些类提供了[常量构造函数](#)。使用常量构造函数，在构造函数名之前加 `const` 关键字，来创建编译时常量时：

```
var p = const ImmutablePoint(2, 2);
```

构造两个相同的编译时常量会产生一个唯一的， 标准的实例：

```
var a = const ImmutablePoint(1, 1);
var b = const ImmutablePoint(1, 1);

assert(identical(a, b)); // 它们是同一个实例。
```

在 *常量* 上下文中， 构造函数或者字面量前的 `const` 可以省略。 例如，下面代码创建了一个 `const` 类型的 `map` 对象：

```
// 这里有很多的 const 关键字。
const pointAndLine = const {
  'point': const [const ImmutablePoint(0, 0)],
  'line': const [const ImmutablePoint(1, 10), const ImmutablePoint(-2, 11)],
};
```

保留第一个 `const` 关键字，其余的全部省略：

```
// 仅有一个 const，由该 const 建立常量上下文。
const pointAndLine = {
  'point': [ImmutablePoint(0, 0)],
  'line': [ImmutablePoint(1, 10), ImmutablePoint(-2, 11)],
};
```

如果常量构造函数在常量上下文之外， 且省略了 `const` 关键字， 此时创建的对象是非常量对象：

```
var a = const ImmutablePoint(1, 1); // 创建一个常量对象
var b = ImmutablePoint(1, 1); // 创建一个非常量对象

assert(!identical(a, b)); // 两者不是同一个实例!
```

**版本提示：** 在 Dart 2 中，一个常量上下文中的 `const` 关键字可以被省略。

## 获取对象的类型

使用对象的 `runtimeType` 属性， 可以在运行时获取对象的类型， `runtimeType` 属性会返回一个 [Type](#) 对象。

```
print('The type of a is ${a.runtimeType}');
```

到目前为止，我们已经解了如何\_使用\_类。 本节的其余部分将介绍如何\_实现\_一个类。

## 实例变量

下面是声明实例变量的示例：

```
class Point {
  num x; // 声明示例变量 x，初始值为 null。
  num y; // 声明示例变量 y，初始值为 null。
  num z = 0; // 声明示例变量 z，初始值为 0。
}
```

未初始化实例变量的默认值为“null”。

所有实例变量都生成隐式 *getter* 方法。非 `final` 的实例变量同样会生成隐式 *setter* 方法。有关更多信息，参考 [Getters 和 setters](#)。

```
class Point {
  num x;
  num y;
}

void main() {
  var point = Point();
  point.x = 4; // Use the setter method for x.
  assert(point.x == 4); // Use the getter method for x.
  assert(point.y == null); // Values default to null.
}
```

如果在声明时进行了实例变量的初始化，那么初始化值会在实例创建时赋值给变量，该赋值过程在构造函数及其初始化列表执行之前。

## 构造函数

通过创建一个与其类同名的函数来声明构造函数（另外，还可以附加一个额外的可选标识符，如 [命名构造函数](#) 中所述）。下面通过最常见的构造函数形式，即生成构造函数，创建一个类的实例：

```
class Point {
  num x, y;

  Point(num x, num y) {
    // 还有更好的方式来实现下面代码，敬请关注。
    this.x = x;
    this.y = y;
  }
}
```

使用 `this` 关键字引用当前实例。

**提示：** 近当存在命名冲突时，使用 `this` 关键字。否则，按照 Dart 风格应该省略 `this`。

通常模式下，会将构造函数传入的参数的值赋值给对应的实例变量，Dart 自身的语法糖精简了这些代码：

```
class Point {
  num x, y;

  // 在构造函数体执行前，
  // 语法糖已经设置了变量 x 和 y。
  Point(this.x, this.y);
}
```

## 默认构造函数

在没有声明构造函数的情况下，Dart 会提供一个默认的构造函数。默认构造函数没有参数并会调用父类的无参构造函数。

## 构造函数不被继承

子类不会继承父类的构造函数。子类不声明构造函数，那么它就只有默认构造函数（匿名，没有参数）。

## 命名构造函数

使用命名构造函数可为一个类实现多个构造函数，也可以使用命名构造函数来更清晰的表明函数意图：

```
class Point {
  num x, y;

  Point(this.x, this.y);

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


切记，构造函数不能够被继承， 这意味着父类的命名构造函数不会被子类继承。 如果希望使用父类中定义的命名构造函数创建子类， 就必须在子类中实现该构造函数。

### 调用父类非默认构造函数

默认情况下，子类的构造函数会自动调用父类的默认构造函数（匿名，无参数）。 父类的构造函数在子类构造函数体开始执行的位置被调用。 如果提供了一个 [initializer list](#) （初始化参数列表）， 则初始化参数列表在父类构造函数执行之前执行。 总之，执行顺序如下：

- 1. initializer list （初始化参数列表）
- 2. superclass's no-arg constructor （父类的无名构造函数）
- 3. main class's no-arg constructor （主类的无名构造函数）

如果父类中没有匿名无参的构造函数， 则需要手工调用父类的其他构造函数。 在当前构造函数冒号 (:) 之后，函数体之前，声明调用父类构造函数。

下面的示例中，Employee 类的构造函数调用了父类 Person 的命名构造函数。 点击运行按钮  执行示例代码。

Dart

Install SDKFormatResetplay\_arrow

```
x
1

1
```

file\_copy

play\_arrow

Console

no issues

由于父类的构造函数参数在构造函数执行之前执行， 所以参数可以是一个表达式或者一个方法调用：

```
class Employee extends Person {
  Employee() : super.fromJson(getDefaultData());
  // ...
}
```

警告： 调用父类构造函数的参数无法访问 this 。例如， 参数可以为静态函数但是不能是实例函数。

### 初始化列表


除了调用超类构造函数之外， 还可以在构造函数体执行之前初始化实例变量。 各参数的初始化用逗号分隔。

```
// 在构造函数体执行之前，
// 通过初始列表设置实例变量。
Point.fromJson(Map<String, num> json)
  : x = json['x'],
    y = json['y'] {
  print('In Point.fromJson(): ($x, $y)');
}
```

**警告：** 初始化程序的右侧无法访问 `this`。

在开发期间， 可以使用 `assert` 来验证输入的初始化列表。

```
Point.withAssert(this.x, this.y) : assert(x >= 0) {
  print('In Point.withAssert(): ($x, $y)');
}
```

使用初始化列表可以很方便的设置 `final` 字段。 下面示例演示了， 如何使用初始化列表初始化设置三个 `final` 字段。 点击运行按钮  执行示例代码。

Dart

Install SDKFormatResetplay\_arrow

```
x
1

1
```

file\_copylaunch

Console

no issues

### 重定向构造函数

有时构造函数的唯一目的是重定向到同一个类中的另一个构造函数。 重定向构造函数的函数体为空， 构造函数的调用在冒号 (:) 之后。

```
class Point {
  num x, y;

  // 类的主构造函数。
  Point(this.x, this.y);

  // 指向主构造函数
  Point.alongXAxis(num x) : this(x, 0);
}
```

### 常量构造函数

如果该类生成的对象是固定不变的， 那么就可以把这些对象定义为编译时常量。 为此， 需要定义一个 `const` 构造函数， 并且声明所有实例变量为 `final`。

```
class ImmutablePoint {
    static final ImmutablePoint origin =
        const ImmutablePoint(0, 0);

    final num x, y;

    const ImmutablePoint(this.x, this.y);
}
```

常量构造函数创建的实例并不总是常量。 更多内容，查看 [使用构造函数](#) 章节。

## 工厂构造函数

当执行构造函数并不总是创建这个类的一个新实例时，则使用 `factory` 关键字。 例如，一个工厂构造函数可能会返回一个 `cache` 中的实例， 或者可能返回一个子类的实例。

以下示例演示了从缓存中返回对象的工厂构造函数：

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

    // 从命名的 _ 可以知,
    // _cache 是私有属性。
    static final Map<String, Logger> _cache =
        <String, Logger>{};

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

    Logger._internal(this.name);

    void log(String msg) {
        if (!mute) print(msg);
    }
}
```

**提示：** 工厂构造函数无法访问 `this`。

工厂构造函数的调用方式与其他构造函数一样：

```
var logger = Logger('UI');
logger.log('Button clicked');
```

## 方法

方法是对象提供行为的函数。

### 实例方法

对象的实例方法可以访问 `this` 和实例变量。 以下示例中的 `distanceTo()` 方法就是实例方法：



```
import 'dart:math';

class Point {
  num x, y;

  Point(this.x, this.y);

  num distanceTo(Point other) {
    var dx = x - other.x;
    var dy = y - other.y;
    return sqrt(dx * dx + dy * dy);
  }
}
```

## Getter 和 Setter

Getter 和 Setter 是用于对象属性读和写的特殊方法。回想之前的例子，每个实例变量都有一个隐式 Getter，通常情况下还会有一个 Setter。使用 `get` 和 `set` 关键字实现 Getter 和 Setter，能够为实例创建额外的属性。

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

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

  // 定义两个计算属性: right 和 bottom。
  num get right => left + width;
  set right(num value) => left = value - width;
  num get bottom => top + height;
  set bottom(num value) => top = value - height;
}

void main() {
  var rect = Rectangle(3, 4, 20, 15);
  assert(rect.left == 3);
  rect.right = 12;
  assert(rect.left == -8);
}
```

最开始实现 Getter 和 Setter 也许是直接返回成员变量；随着需求变化，Getter 和 Setter 可能需要进行计算处理而使用方法来实现；但是，调用对象的代码不需要做任何修改。

**提示：**类似 `(++)` 之类操作符不管是否定义了 `getter` 方法，都能够正确的执行。为了避免一些问题，操作符只调用一次 `getter` 方法，然后把值保存到一个临时的变量中。

## 抽象方法

实例方法，`getter`，和 `setter` 方法可以是抽象的，只定义接口不进行实现，而是留给其他类去实现。抽象方法只存在于 [抽象类](#) 中。

定义一个抽象函数，使用分号 `(;)` 来代替函数体：

```
abstract class Doer {
  // 定义实例变量和方法 ...

  void doSomething(); // 定义一个抽象方法。
}

class EffectiveDoer extends Doer {
  void doSomething() {
    // 提供方法实现，所以这里的方法就不是抽象方法了...
  }
}
```

调用抽象方法会导致运行时错误。

# 抽象类

使用 `abstract` 修饰符来定义 *抽象类* — 抽象类不能实例化。 抽象类通常用来定义接口，以及部分实现。 如果希望抽象类能够被实例化，那么可以通过定义一个 [工厂构造函数](#) 来实现。

抽象类通常具有 [抽象方法](#)。下面是一个声明具有抽象方法的抽象类示例：

```
// 这个类被定义为抽象类，
// 所以不能被实例化。
abstract class AbstractContainer {
    // 定义构造行数，字段，方法...

    void updateChildren(); // 抽象方法。
}
```

# 隐式接口

每个类都隐式的定义了一个接口，接口包含了该类所有的实例成员及其实现的接口。 如果要创建一个 A 类，A 要支持 B 类的 API，但是不需要继承 B 的实现， 那么可以通过 A 实现 B 的接口。

一个类可以通过 `implements` 关键字来实现一个或者多个接口， 并实现每个接口要求的 API。例如：

```
// person 类。 隐式接口里面包含了 greet() 方法声明。
class Person {
    // 包含在接口里，但只在当前库中可见。
    final _name;

    // 不包含在接口里，因为这是一个构造函数。
    Person(this._name);

    // 包含在接口里。
    String greet(String who) => 'Hello, $who. I am $_name.';
}

// person 接口的实现。
class Impostor implements Person {
    get _name => '';

    String greet(String who) => 'Hi $who. Do you know who I am?';
}

String greetBob(Person person) => person.greet('Bob');

void main() {
    print(greetBob(Person('Kathy')));
    print(greetBob(Impostor()));
}
```

下面示例演示一个类如何实现多个接口： Here’s an example of specifying that a class implements multiple interfaces:

```
class Point implements Comparable, Location {...}
```

# 扩展类（继承）

使用 `extends` 关键字来创建子类， 使用 `super` 关键字来引用父类：

```
class Television {
    void turnOn() {
        _illuminateDisplay();
        _activateIrSensor();
    }
    // ...
}

class SmartTelevision extends Television {
    void turnOn() {
        super.turnOn();
        _bootNetworkInterface();
        _initializeMemory();
        _upgradeApps();
    }
    // ...
}
```

重写类成员

子类可以重写实例方法，getter 和 setter。可以使用 `@override` 注解指出想要重写的成员：

```
class SmartTelevision extends Television {
    @override
    void turnOn() {...}
    // ...
}
```

To narrow the type of a method parameter or instance variable in code that is [type safe](#), you can use the [covariant keyword](#).

重写运算符

下标的运算符可以被重写。 例如，想要实现两个向量对象相加，可以重写 `+` 方法。

<	+		[]
>	/	^	[]=
<=	~/	&	~
>=	*	<<	==
-	%	>>	

**提示：** 你可能会被提示 `!=` 运算符为非可重载运算符。 因为 `e1 != e2` 表达式仅仅是 `!(e1 == e2)` 的语法糖。

下面示例演示一个类重写 `+` 和 `-` 操作符：

```

class Vector {
    final int x, y;

    Vector(this.x, this.y);

    Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
    Vector operator -(Vector v) => Vector(x - v.x, y - v.y);

    // 运算符 == 和 hashCode 部分没有列出。 有关详情，请参考下面的注释。
    // ...
}

void main() {
    final v = Vector(2, 3);
    final w = Vector(2, 2);

    assert(v + w == Vector(4, 5));
    assert(v - w == Vector(0, 1));
}

```

如果要重写 `==` 操作符，需要重写对象的 `hashCode` getter 方法。 重写 `==` 和 `hashCode` 的实例，参考 [Implementing map keys](#).

有关重写的更多介绍，请参考 [扩展类 \(继承\)](#).

## noSuchMethod()

当代码尝试使用不存在的方法或实例变量时， 通过重写 `noSuchMethod()` 方法，来实现检测和应对处理：

```

class A {
    // 如果不重写 noSuchMethod，访问
    // 不存在的实例变量时会导致 NoSuchMethodError 错误。
    @override
    void noSuchMethod(Invocation invocation) {
        print('You tried to use a non-existent member: ' +
            '${invocation.memberName}');
    }
}

```

除非符合下面的任意一项条件， 否则没有实现的方法不能够被调用：

- receiver 具有 `dynamic` 的静态类型。
- receiver 具有静态类型，用于定义为实现的方法 (可以是抽象的), 并且 receiver 的动态类型具有 `noSuchMethod()` 的实现， 该实现与 `Object` 类中的实现不同。

有关更多信息，参考 [noSuchMethod forwarding specification](#).

## 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。

## 使用枚举

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，混合或实现。
- 枚举不能被显式实例化。

有关更多信息，参考 [Dart language specification](#) 。

## 为类添加功能：Mixin

Mixin 是复用类代码的一种途径， 复用的类可以在不同层级， 之间可以不存在继承关系。

通过 `with` 后面跟一个或多个混入的名称， 来 使用 Mixin， 下面的示例演示了两个使用 Mixin 的类：

```
class Musician extends Performer with Musical {
  // ...
}

class Maestro extends Person
  with Musical, Aggressive, Demented {
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

通过创建一个继承自 `Object` 且没有构造函数的类， 来 实现 一个 Mixin。如果 Mixin 不希望作为常规类被使用， 使用关键字 `mixin` 替换 `class`。例如：

```
mixin Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}
```

指定只有某些类型可以使用的 Mixin - 比如，Mixin 可以调用 Mixin 自身没有定义的方法 - 使用 `on` 来指定可以使用 Mixin 的父类类型：

```
mixin MusicalPerformer on Musician {  
  // ...  
}
```

**版本提示：** `mixin` 关键字在 Dart 2.1 中被引用支持。早期版本中的代码通常使用 `abstract class` 代替。更多有关 Mixin 在 2.1 中的变更信息，请参见 [Dart SDK changelog](#) 和 [2.1 mixin specification](#)。

**提示：** 对 Mixin 的一些限制正在被移除。关于更多详情，参考 [proposed mixin specification](#).

有关 Dart 中 Mixin 的理论演变，参考 [A Brief History of Mixins in Dart](#).

## 类变量和方法

使用 `static` 关键字实现类范围的变量和方法。

### 静态变量

静态变量（类变量）对于类级别的状态是非常有用的：

```
class Queue {  
  static const initialCapacity = 16;  
  // ...  
}  
  
void main() {  
  assert(Queue.initialCapacity == 16);  
}
```

静态变量只到它们被使用的时候才会初始化。

**提示：** 代码准守[风格推荐指南](#) 中的命名规则，使用 `lowerCamelCase` 来命名常量。

### 静态方法

静态方法（类方法）不能在实例上使用，因此它们不能访问 `this`。例如：

```
import 'dart:math';  
  
class Point {  
  num x, y;  
  Point(this.x, this.y);  
  
  static num distanceBetween(Point a, Point b) {  
    var dx = a.x - b.x;  
    var dy = a.y - b.y;  
    return sqrt(dx * dx + dy * dy);  
  }  
}  
  
void main() {  
  var a = Point(2, 2);  
  var b = Point(4, 4);  
  var distance = Point.distanceBetween(a, b);  
  assert(2.8 < distance && distance < 2.9);  
  print(distance);  
}
```

**提示：** 对于常见或广泛使用的工具和函数， 应该考虑使用顶级函数而不是静态方法。

静态函数可以当做编译时常量使用。 例如，可以将静态方法作为参数传递给常量构造函数。

## 泛型

在 API 文档中你会发现基础数组类型 `List` 的实际类型是 `List<E>`。`<...>` 符号将 `List` 标记为 *泛型* (或 *参数化*) 类型。 这种类型具有形式化的参数。 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。

## 为什么使用泛型

在类型安全上通常需要泛型支持， 它的好处不仅仅是保证代码的正常运行：

- 正确指定泛型类型可以提高代码质量。
- 使用泛型可以减少重复的代码。

如果能让 `List` 仅仅支持字符串类型， 可以将其声明为 `List<String>`（读作“字符串类型的 `list` ”）。那么，当一个非字符串被赋值给了这个 `list` 时，开发工具就能够检测到这样的做法可能存在错误。 例如：

```
var names = List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
names.add(42); // 错误
```

X static analysis: error/warning

另外一个使用泛型的原因是减少重复的代码。 泛型可以在多种类型之间定义同一个实现， 同时还可以继续使用检查模式和静态分析工具提供的代码分析功能。 例如，假设你创建了一个用于缓存对象的接口：

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

后来发现需要一个相同功能的字符串类型接口，因此又创建了另一个接口：

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

后来，又发现需要一个相同功能的数字类型接口 ... 这里你应该明白了。

泛型可以省去创建所有这些接口的麻烦。 通过创建一个带有泛型参数的接口，来代替上述接口：

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

在上面的代码中，`T` 是一个备用类型。 这是一个类型占位符，在开发者调用该接口的时候会指定具体类型。

## 使用集合字面量

`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， 下面是参数化字面量的示例：



```
var names = <String>['Seth', 'Kathy', 'Lars'];
var uniqueNames = <String>{'Seth', 'Kathy', 'Lars'};
var pages = <String, String>{
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

## 使用泛型类型的构造函数

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>();
```

## 运行时中的泛型集合

Dart 中泛型类型是 *固化的*，也就是说它们在运行时是携带着类型信息的。 例如， 在运行时检测集合的类型：

```
var names = List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
print(names is List<String>); // true
```

**提示：** 相反，Java中的泛型会被 *擦除*，也就是说在运行时泛型类型参数的信息是不存在的。 在Java中，可以测试对象是否为 List 类型， 但无法测试它是否为 List<String>。

## 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>();
```

X static analysis: error/warning

## 使用泛型函数

最初，Dart 的泛型只能用于类。新语法\_泛型方法\_，允许在方法和函数上使用类型参数：

```
T first<T>(List<T> ts) {
  // Do some initial work or error checking, then...
  T tmp = ts[0];
  // Do some additional checking or processing...
  return tmp;
}
```

这里的 `first` (`<T>`) 泛型可以在如下地方使用参数 `T`：

- 函数的返回值类型 (`T`).
- 参数的类型 (`List<T>`).
- 局部变量的类型 (`T tmp`).

关于泛型的更多信息，参考 [使用泛型函数](#)

## 库和可见性

`import` 和 `library` 指令可以用来创建一个模块化的，可共享的代码库。库不仅提供了 API，而且对代码起到了封装的作用：以下划线 (`_`) 开头的标识符仅在库内可见。每个 *Dart 应用程序都是一个库*，虽然没有使用 `library` 指令。

库可以通过包来分发。有关 pub（集成在SDK中的包管理器）的信息，请参考 [Pub Package 和 Asset Manager](#)。

## 使用库

通过 `import` 指定一个库命名空间中的内如如何在另一个库中使用。例如，Dart Web应用程序通常使用 [dart:html](#) 库，它们可以像这样导入：

```
import 'dart:html';
```

`import` 参数只需要一个指向库的 URI。对于内置库，URI 拥有自己特殊的`dart:` 方案。对于其他的库，使用系统文件路径或者 `package:` 方案。`package:` 方案指定由包管理器（如 pub 工具）提供的库。例如：

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

**提示：** *URI* 代表统一资源标识符。*URL*（统一资源定位符）是一种常见的URI。

## 指定库前缀

如果导入两个存在冲突标识符的库，则可以为这两个库，或者其中一个指定前缀。例如，如果 `library1` 和 `library2` 都有一个 `Element` 类，那么可以通过下面的方式处理：

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

// 使用 lib1 中的 Element。
Element element1 = Element();

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

## 导入库的一部分

如果你只使用库的一部分功能，则可以选择需要导入的 内容。例如：

```
// Import only foo.
import 'package:lib1/lib1.dart' show foo;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

## 延迟加载库

*Deferred loading* (也称之为 *lazy loading*) 可以让应用在需要的时候再加载库。 下面是一些使用延迟加载库的场景：

- 减少 APP 的启动时间。
- 执行 A/B 测试，例如 尝试各种算法的 不同实现。
- 加载很少使用的功能，例如可选的屏幕和对话框。

要延迟加载一个库，需要先使用 `deferred as` 来导入：

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

当需要使用的时候，使用库标识符调用 `loadLibrary()` 函数来加载库：

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

在前面的代码，使用 `await` 关键字暂停代码执行一直到库加载完成。 关于 `async` 和 `await` 的更多信息请参考 [异步支持](#)。

在一个库上你可以多次调用 `loadLibrary()` 函数。但是该库只是载入一次。

使用延迟加载库的时候，请注意一下问题：

- 延迟加载库的常量在导入的时候是不可用的。 只有当库加载完毕的时候，库中常量才可以使用。
- 在导入文件的时候无法使用延迟库中的类型。 如果你需要使用类型，则考虑把接口类型移动到另外一个库中， 让两个库都分别导入这个接口库。
- Dart 隐含的把 `loadLibrary()` 函数导入到使用 `deferred as` 的命名空间中。 `loadLibrary()` 方法返回一个 [Future](#)。

**Dart VM difference:** The Dart VM allows access to members of deferred libraries even before the call to `loadLibrary()`. This behavior might change, so **don't depend on the current VM behavior**. For details, see [issue #33118](#).

## 实现库

有关如何实现库包的建议，请参考 [Create Library Packages](#) 这里面包括：

- 如何组织库的源文件。
- 如何使用 `export` 命令。
- 何时使用 `part` 命令。
- 何时使用 `library` 命令。

## 异步支持

Dart 库中包含许多返回 `Future` 或 `Stream` 对象的函数. 这些函数在设置完耗时任务（例如 I/O 曹组）后， 就立即返回了，不会等待耗任务完成。 使用 `async` 和 `await` 关键字实现异步编程。可以让你像编写同步代码一样实现异步操作。

## 处理 Future

可以通过下面两种方式，获得 `Future` 执行完成的结果：

- 使用 `async` 和 `await`。
- 使用 `Future API`，具体描述，参考 [库概览](#)。

使用 `async` 和 `await` 关键字的代码是异步的。 虽然看起来有点想同步代码。 例如，下面的代码使用 `await` 等待异步函数的执行结果。

```
await lookupVersion();
```

要使用 `await`，代码必须在 *异步函数*（使用 `async` 标记的函数）中：

```
Future checkVersion() async {  
  var version = await lookupVersion();  
  // Do something with version  
}
```

**提示：** 虽然异步函数可能会执行耗时的操作，但它不会等待这些操作。相反，异步函数只有在遇到第一个 `await` 表达式（[详情见](#)）时才会执行。也就是说，它返回一个 `Future` 对象，仅在`await`表达式完成后才恢复执行。

使用 `try`，`catch`，和 `finally` 来处理代码中使用 `await` 导致的错误。

```
try {  
  version = await lookupVersion();  
} catch (e) {  
  // React to inability to look up the version  
}
```

在一个异步函数中可以多次使用 `await`。例如，下面代码中等待了三次函数结果：

```
var entrypoint = await findEntrypoint();  
var exitCode = await runExecutable(entrypoint, args);  
await flushThenExit(exitCode);
```

在 `await 表达式` 中，`表达式` 的值通常是一个 `Future` 对象；如果不是，这是表达式的值会被自动包装成一个 `Future` 对象。`Future` 对象指明返回一个对象的承诺（promise）。`await 表达式` 执行的结果为这个返回的对象。`await` 表达式会阻塞代码的执行，直到需要的对象返回为止。

如果在使用 `await` 导致编译时错误，确认 `await` 是否在一个异步函数中。例如，在应用的 `main()` 函数中使用 `await`，`main()` 函数的函数体必须被标记为 `async`：

```
Future main() async {  
  checkVersion();  
  print('In main: version is ${await lookupVersion()}');  
}
```

## 声明异步函数

函数体被 `async` 标示符标记的函数，即是一个\_异步函数\_。将 `async` 关键字添加到函数使其返回`Future`。例如，考虑下面的同步函数，它返回一个 `String`：

```
String lookupVersion() => '1.0.0';
```

例如，将来的实现将非常耗时，将其更改为异步函数，返回值是 `Future`。

```
Future<String> lookupVersion() async => '1.0.0';
```

注意，函数体不需要使用`Future` API。如有必要，`Dart` 会创建 `Future` 对象。

如果函数没有返回有效值，需要设置其返回类型为 `Future<void>`。

## 处理 Stream

当需要从 `Stream` 中获取数据值时，可以通过一下两种方式：

- 使用 `async` 和一个 *异步循环* (`await for`)。
- 使用 Stream API, 更多详情, 参考 [in the library tour](#)。

**提示:** 在使用 `await for` 前, 确保代码清晰, 并且确实希望等待所有流的结果。例如, 通常不应该使用 `await for` 的UI事件侦听器, 因为UI框架会发送无穷无尽的事件流。

一下是异步for循环的使用形式:

```
await for (varOrType identifier in expression) {  
  // Executes each time the stream emits a value.  
}
```

上面 *表达式* 返回的值必须是 Stream 类型。 执行流程如下:

1. 等待, 直到流发出一个值。
2. 执行 for 循环体, 将变量设置为该发出的值
3. 重复1和2, 直到关闭流。

使用 `break` 或者 `return` 语句可以停止接收 `stream` 的数据, 这样就跳出了 `for` 循环, 并且从 `stream` 上取消注册。 \*\*如果在实现异步 `for` 循环时遇到编译时错误, 请检查确保 `await for` 处于异步函数中。 \*\* 例如, 要在应用程序的 `main()` 函数中使用异步 `for` 循环, `main()` 函数体必须标记为 `async`` :

```
Future main() async {  
  // ...  
  await for (var request in requestServer) {  
    handleRequest(request);  
  }  
  // ...  
}
```

有关异步编程的更多信息, 请参考 [dart:async](#) 部分。 同时也可参考文章 [Dart Language Asynchrony Support: Phase 1](#) 和 [Dart Language Asynchrony Support: Phase 2](#), 以及 [Dart language specification](#) 。

## 生成器

当您需要延迟生成( lazily produce )一系列值时, 可以考虑使用\_生成器函数\_。Dart 内置支持两种生成器函数:

- **Synchronous** 生成器: 返回一个 [Iterable](#) 对象。
- **Asynchronous** 生成器: 返回一个 [Stream](#) 对象。

通过在函数体标记 `sync*`, 可以实现一个**同步**生成器函数。使用 `yield` 语句来传递值:

```
Iterable<int> naturalsTo(int n) sync* {  
  int k = 0;  
  while (k < n) yield k++;  
}
```

通过在函数体标记 `async*`, 可以实现一个**异步**生成器函数。使用 `yield` 语句来传递值:

```
Stream<int> asynchronousNaturalsTo(int n) async* {  
  int k = 0;  
  while (k < n) yield k++;  
}
```


如果生成器是递归的, 可以使用 `yield*` 来提高其性能:

```
Iterable<int> naturalsDownFrom(int n) sync* {
  if (n > 0) {
    yield n;
    yield* naturalsDownFrom(n - 1);
  }
}
```

有关生成器的更多信息，请参考文章 [Dart Language Asynchrony Support: Phase 2](#)。

## 可调用类

通过实现类的 `call()` 方法，能够让类像函数一样被调用。

在下面的示例中，`WannabeFunction` 类定义了一个 `call()` 函数，函数接受三个字符串参数，函数体将三个字符串拼接，字符串间用空格分割，并在结尾附加了一个感叹号。单击运行按钮  执行代码。

Dart

Install SDKFormatReset

play\_arrow

file\_copy

launch

x

1

1

Console

no issues

有关把类当做方法使用的更多信息，请参考 [Emulating Functions in Dart](#)。

## Isolates

大多数计算机中，甚至在移动平台上，都在使用多核CPU。为了有效利用多核性能，开发者一般使用共享内存数据来保证多线程的正确执行。然而，多线程共享数据通常会导致很多潜在的问题，并导致代码运行出错。

所有 Dart 代码都在 *隔离区*（`isolates`）内运行，而不是线程。每个隔离区都有自己的内存堆，确保每个隔离区的状态都不会被其他隔离区访问。

有关更多信息，请参考 [dart:isolate library documentation](#)。

## Typedefs

在 Dart 中，函数也是对象，就想字符和数字对象一样。使用 *typedef*，或者 *function-type alias* 为函数起一个别名，别名可以用来声明字段及返回值类型。当函数类型分配给变量时，`typedef`会保留类型信息。

请考虑以下代码，代码中未使用 `typedef`：

```
class SortedCollection {
  Function compare;

  SortedCollection(int f(Object a, Object b)) {
    compare = f;
  }
}

// Initial, broken implementation. // broken ?
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = SortedCollection(sort);

  // 虽然知道 compare 是函数,
  // 但是函数是什么类型 ?
  assert(coll.compare is Function);
}
```



当把 `f` 赋值给 `compare` 的时候，类型信息丢失了。`f` 的类型是 `(Object, Object) → int` (这里 `→` 代表返回值类型)，但是 `compare` 得到的类型是 `Function`。如果我们使用显式的名字并保留类型信息，这样开发者和工具都可以使用这些信息：

```
typedef Compare = int Function(Object a, Object b);

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

// Initial, broken implementation.
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = SortedCollection(sort);
  assert(coll.compare is Function);
  assert(coll.compare is Compare);
}
```

**提示：** 目前，`typedefs` 只能使用在函数类型上，我们希望将来这种情况有所改变。

由于 `typedefs` 只是别名，他们还提供了一种方式来判断任意函数的类型。例如：

```
typedef Compare<T> = int Function(T a, T b);

int sort(int a, int b) => a - b;

void main() {
  assert(sort is Compare<int>); // True!
}
```

## 元数据

使用元数据可以提供有关代码的其他信息。元数据注释以字符 `@` 开头，后跟对编译时常量 (如 `deprecated`) 的引用或对常量构造函数的调用。

对于所有 Dart 代码有两种可用注解：`@deprecated` 和 `@override`。关于 `@override` 的使用，参考 [扩展类 \(继承\)](#)。下面是使用 `@deprecated` 注解的示例：

```
class Television {
  /// _Deprecated: Use [turnOn] instead._
  @deprecated
  void activate() {
    turnOn();
  }

  /// Turns the TV's power on.
  void turnOn() {...}
}
```

可以自定义元数据注解。下面的示例定义了一个带有两个参数的 `@todo` 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');
}
```

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 或者 export 指令之前使用。 使用反射可以在运行时获取元数据信息。

## 注释

Dart 支持单行注释、多行注释和文档注释。

### 单行注释

单行注释以 `//` 开始。 所有在 `//` 和改行结尾之间的内容被编译器忽略。

```
void main() {
  // TODO: refactor into an AbstractLlamaGreetingFactory?
  print('Welcome to my Llama farm!');
}
```

### 多行注释

多行注释以 `/*` 开始， 以 `*/` 结尾。 所有在 `/*` 和 `*/` 之间的内容被编译器忽略 （不会忽略文档注释）。 多行注释可以嵌套。

```
void main() {
  /*
   * This is a lot of work. Consider raising chickens.

   Llama larry = Llama();
   larry.feed();
   larry.exercise();
   larry.clean();
   */
}
```

### 文档注释

文档注释可以是多行注释，也可以是单行注释， 文档注释以 `///` 或者 `/**` 开始。 在连续行上使用 `///` 与多行文档注释具有相同的效果。

在文档注释中，除非用中括号括起来，否则Dart 编译器会忽略所有文本。使用中括号可以引用类、 方法、 字段、 顶级变量、 函数、 和 参数。 括号中的符号会在已记录的程序元素的词法域中进行解析。

下面是一个引用其他类和成员的文档注释：

```
/// A domesticated South American camelid (Lama glama).  
///  
/// 自从西班牙时代以来,  
/// 安第斯文化就将骆驼当做肉食类和运输类动物。  
class Llama {  
  String name;  
  
  /// 喂养骆驼 [Food].  
  ///  
  /// 典型的美洲驼每周吃一捆干草。  
  void feed(Food food) {  
    // ...  
  }  
  
  /// 使用 [activity] 训练骆驼  
  /// [timeLimit] 分钟。  
  void exercise(Activity activity, int timeLimit) {  
    // ...  
  }  
}
```

在生成的文档中，`[Food]` 会成为一个链接，指向 Food 类的 API 文档。

解析 Dart 代码并生成 HTML 文档，可以使用 SDK 中的 [documentation generation tool](#). 关于生成文档的实例，请参考 [Dart API documentation](#). 关于文档结构的建议，请参考 [Guidelines for Dart Doc Comments](#).

## 总结

本页概述了 Dart 语言中常用的功能。 还有更多特性有待实现，但我们希望它们不会破坏现有代码。 有关更多信息，请参考 [Dart language specification](#) 和 [Effective Dart](#).

要了解更多关于 Dart 核心库的内容，请参考 [A Tour of the Dart Libraries](#).