Dart 语言中文教程

翻译自 Dart 官网，查看原文。欢迎提供修改意见。

另见：Dart 库教程。

当前版本：2.4.0 (stable)

完成度：100%

这个页面展示如何使用 Dart 的各个主要特性，从变量、运算符到类和库，并且假定你已经会使用其他编程语言编写代码。
要详细了解 Dart 核心库相关内容，请查阅 Dart 库教程。当你想对一个语言特性深入了解时，无论何时都可以查阅 Dart 语言规范。

> 小说明：在 DartPad 上，你可以尝试 Dart 的大部分语言特性（了解更多）。
>
> 转到 DartPatd。

目录

一个基本的 Dart 程序

下面的代码使用了 Dart 的许多基本特性：

```
// 定义函数
printInteger(int aNumber) {
  print('The number is $aNumber.'); // 打印到控制台
}

// 这里是程序开始执行的地方
main() {
  var number = 42; // 定义并初始化变量
  printInteger(number); // 调用函数
}
```

下面是这个程序使用的可适用于所有 (almost) Dart 应用的特性：

```
// 这是一个单行注释
```

一个单行注释。Dart 也支持多行和文档注释。详情请参阅 注释。

```
int
```

一个类型。其他的一些 内置类型 包括**字符串**、**List** 和**布尔**。

```
42
```

一个数值字面量。数值字面量是一种编译期常量。

```
print()
```

一个方便的展示输出方式。

```
'...' (or "...")
```

一个字符串字面量。

```
$variableName (or ${expression})
```

字符串插值：插入一个变量或者表达式的字符串值到一个字符串字面量里。详情请参阅 字符串。

```
main()
```

应用开始执行的特定的、必须的、顶级的函数。详情请参阅 main() 函数。

```
var
```

一种声明变量但是不指定类型的方法。

> 说明：本篇文章的代码遵从 Dart 风格指南 中的公约。

重要概念

当你学习 Dart 语言的时候，请记住以下事实和概念：

- 所有可以放在一个变量里面的东西都是*对象*，而且所有对象都是*类*的实例。每一个数值、函数和 **null** 都是对象。所有的对象都继承自 Object 类。
- 尽管 Dart 是强类型的，但是 Dart 支持类型推断所以类型声明是可选的。在上面的代码中，**number** 被推断为类型 **int**。当你想要显式声明没有预期的类型时，**使用特殊的 dynamic 类型。**
- Dart 支持泛型，像是 **List<int>**（包含整数的列表）或者 **List<dynamic>**（一个包含任意类型对象的列表）。
- 除了绑定在类和对象上的函数（分别为静态方法和实例方法）以外，Dart 还支持顶级函数（像 **main()**）。你还可以在函数中创建函数（嵌套函数或局部函数）。
- 不像 Java，Dart 没有这些关键词：**pubilc**，**protected**，**private**。如果一个标识符以下划线 (_) 开头，那么对于它的库来说是私有的。详情请参阅 库和可见性。
- 标识符可以以下划线 (_) 开头，后面跟上任意字母和数字的组合。
- Dart 既有“表达式”（具有运行时的值）也有“语句”（没有运行时的值）。例如，条件表达式 **condition ? expr1 : expr2** 有 **expr1** 或 **expr2** 的值。相对的一个 if-else 语句，是没有值的。一个语句经常包含一个或多个表达式，但是一个表达式不能直接包含一个语句。
- Dart 开发工具会报告两种类型的问题："警告"和"错误"。警告只是表明你的代码可能无法正常工作，但是并不会禁止你执行程序。错误可能是编译期或者运行期的。一个编译期错误完全禁止程序的执行；而运行期错误会在代码执行到这里时抛出一个 异常。

关键词

下面的表格列出了 Dart 语言特殊对待的关键词。

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

避免使用这些单词作为标识符。然而，如果必要，带角标的关键词可以作为标识符：

- 带角标 *1* 的是 **上下文关键词**，它们只在特定的地方有有意义。除此之外他们在所有地方都是合法的关键词。
- 带角标 *2* 的是 **内置标识符**。为了简化将JavaScript代码移植到Dart的任务，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，也不能用作导入前缀。
- 带角标 *3* 的是新的，与 异步支持 相关的限制性关键词，在 Dart 1.0 发布后才被加入。在以 **async**、**async*** 或 **yield** 标识的函数体中，你不能使用 **async**、**await** 或者 **yield** 作为标识符。

关键词表里的其他所有单词都是**保留词**。你不能使用它们作为标识符。

变量

这里是创建并初始化一个变量的例子：

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

变量保存的是引用。名字是 **name** 的变量包含一个指向值为 "Bob" 的**字符串**对象的引用。

名字为 **name** 的变量类型被推断为 **String**，但是你可以通过显示指定类型来改变这个行为。如果对象不限于一个单一类型，指定它为 **Object** 或 **dynamic** 类型。

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

另一个选择是显式指定类型为它将会被推断的类型：

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

> 说明：对于局部变量，本篇文章遵守 代码风格推荐 使用 var，而不是类型声明。

默认值

未初始话的变量有一个初始值 **null**。即使是数值类型的变量初始值也是 null，因为数值——和 Dart 中其他所有类型一样——都是对象。

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

> 说明：生产环境的代码会忽略 assert() 调用。在开发时，如果 *condition* 的结果是 false，**assert(*condition*)** 会抛出一个异常。详情请参阅 断言。

Final 和 const

如果你从不打算改变一个变量，请使用 **final** 和 **const**，而不是 **var** 或者一个类型名。Final 变量只可以被设置一次；而 const 变量是编译期常量。（Const 变量是隐式 final 的。）一个 final 的顶级变量或者类变量在首次被使用时初始化。

> 说明：实例变量只可以是 **final** 的，不可以是 **const** 的。Final 实例变量必须在构造函数体开始前被初始化——在变量声明时、通过构造函数参数或者在构造函数的 初始化列表 中。

这里是创建并设置一个 final 变量的例子：

```
final name = 'Bob'; // 没有类型声明
final String nickname = 'Boddy';
```

你不可以改变一个 final 变量的值：

```
name = 'Alice'; // 错误：一个 final 变量只可以被设置一次
```

对那些你想要作为**编译期常量**的变量使用 **const**。如果这个 const 变量是类级别的，使用 **static const** 标识它。在你声明的时候，设置变量的值为编译期常量比如数字、字符串字面量、另一个常量或者常量数值的算术运算结果。

```
const bar = 1000000; // 压力单位（达因/cm2）
const double atm = 1.01325 * bar; // 标准大气压
```

Const 关键词不仅可以声明常量。你还可以使用它创建常量值，也可以声明创建常量值的构造函数。任何变量都可以拥有一个常量值。

```
var foo = const [];
final bar = const [];
const baz = []; // 等同于 `const []`
```

你可以忽略常量声明中初始化表达式中的 **const**，像上面的 **baz** 一样。详情请参阅 不要重复使用 const。

你可以改变一个非 final 且非 const 变量的值，即使它有一个常量值。

```
foo = [1, 2, 3]; // 之前是 const []
```

你不可以改变一个常量的值：

```
baz = [42]; // 错误：常量不可以被赋值
```

要了解更多使用 **const** 创建常量值的内容，请参阅 List、Map 和类。

内置类型

Dart 语言对以下类型有特殊的支持：

- numbers（数值）
- strings（字符串）
- booleans（布尔）
- lists（列表，也称为数组）
- maps（映射）
- runes（在字符串中表示一个Unicode字符）
- symbols

你可以使用字面量初始化以上任意类型。比如，**'this is a string'** 就是一个字符串字面量，而 **true** 是一个 boolean 字面量。

由于 Dart 中的所有变量都是对象的引用——一个类的实例——所有你通常可以使用"构造函数"来初始化变量。某些内置类型有它们自己的构造函数。比如，你可以使用 **Map()** 构造函数创建一个 map。

数值

Dart 中的数值有两种类型：

int

小于等于64位的整数值，实际长度依赖运行平台。在 Dart 虚拟机上，可以是 -2^{63} 到 2^{63} 次方 -1。编译到 JavaScript 的 Dart 使用 JavaScript的数值，允许从 -2^{53} 到 2^{53} - 1的值。

double

64位（双精度）浮点数值，如 IEE 754 标准中所规定的。

Int 和 **double** 都是 num 的子类。Num 类型包含像 +，-，/ 和 * 这样的基本运算符，也是 **abs()**、**ceil()** 和 **floor()** 适用的类型。（像 >> 这样的位运算符定义在 int 类中。）如果你从 num 和它的子类中找不到你想要的，试着看看 dart:math 库。

整数是没有小数点的数字。下面是一些定义整数字面量的例子：

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

如果一个数字包含小数点，那么它是一个浮点数。下面是一些定义浮点数字面量的例子：

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

在 Dart 2.1 中，数字字面量在必要时会自动转换为 double：

```
double z = 1; // 等效于 double z = 1.0
```

> 版本说明：在 Dart 2.1 之前，在 double 上下文中使用一个整数字面量会引发一个错误。

下面是一些数值和字符串互相转换的例子：

```
// 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(oneSAsString == '1');

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

整数类型支持传统的位运算符，比如移位 (<<, >>) 、按位与 (&) 和按位或 (|)。下面是一个例子：

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

以字面量定义的数值是编译期常量。许多算术表达式也同样是编译器常量，只要它们的操作数是编译期常数且最后得到一个数值。

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

字符串

Dart 的字符串是以 UTF-16 编码单元组成的序列。你可以使用单引号或者双引号来创建字符串：

```
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.";
```

你可以使用 **${*expression*}** 将一个表达式插入到字符串中。如果这个表达式是一个标识符，你可以省略 {}。为了得到一个对象的字符串表示，Dart 会调用对象的 **toString()** 方法。

```
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!');
```

> 说明：== 运算符测试两个对象是否相等。两个字符串相等的条件是它们包含同样的编码单位序列。

你可以通过并排字符串字面量或者使用 + 运算符来串联字符串：

```
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.');
```

另一种方式是创建一个多行字符串：使用三个引号（单引号或双引号）来标记：

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

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

你可以通过 **r** 前缀来创建一个"原始的"字符串：

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

要详细了解 Unicode 字符在字符串中是怎样表示的，请参阅 Runes。

只要所有的插值表达式是编译期常量，计算结果为 null 或者 数值、字符串、布尔值，那么这个字符串字面量就是编译期常量。

```
// 可作为常量字符串的组成部分
const aConstNum = 0;
const aConstBool = true;
const aConstString = 'a constant string';

// 不可作为常量字符串的组成部分
var aNum = 0;
var aBool = true;
var aString = 'a string';
const aConstList = [1, 2, 3];

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

要了解更多使用字符串的信息，请参阅 字符串和正则表达式。

布尔

为了表示布尔值，Dart 内置了一个名字为 **bool** 的类型。只有两个对象拥有布尔类型：布尔字面量 **true** 和 **false**，它们两个都是编译期常量。

Dart 的类型安全意味着你不能使用像 **if (*nonbooleanValue*)** 这样的代码。取而代之，你需要明确地检查值，像是：

```
// 检查是否是空字符串
var fullName = '';
assert(fullName.isEmpty);

// 检查是否为0
var hitPoints = 0;
assert(hitPoints <= 0);

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

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

Lists

“数组”或者有序的对象组，也许是大部分编程语言中最常用的集合类型了。在 Dart 中，数组是类型为 List 的对象，所以人们通常称之为“列表”。

Dart 的列表字面量看起来就像 JavaScript 的数组字面量。下面是一个简单的 Dart 列表：

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

> 说明：Dart 推断上面的 **list** 类型是 **List<int>**。如果你试图添加一个非整数值对象到这个列表中，分析器或者运行时会报告一个错误。要了解详细信息，请参阅 类型推断。

列表使用基于0的索引，也就是说 0 是列表中第一个元素的索引，而 **list.length - 1** 是最后一个元素的索引。你可以像 JavaScript 一样获取 list 的长度和它的元素：

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

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

要创建一个作为编译期常量的列表，在列表字面量前加上 **const**：

```
var constantList = const [1, 2, 3];
// constantList[1] = 1; // 这一行会引发一个错误
```

Dart 2.3 引入了 **扩展运算符 (...)** 和 **空感知的扩展运算符 (...?)**，它们提供了一个简洁的方法来向集合中插入多个元素。

举例来说，你可以使用扩展运算符 (...) 来向一个列表中插入另一个列表的所有元素。

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

如果扩展运算符右边的表达式可能为空，你可以使用空感知的扩展运算符来 (...?) 避免异常：

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

要了解更多关于扩展运算符的详情和例子，请参阅 扩展运算符提案。

Dart 2.3 也引入了 **集合 if** 和 **集合 for**，它们提供了使用条件 (if) 和循环 (for) 构建结合的方法。

下面是一个使用 **集合 if** 来创建一个包含3个或4个项目的列表的例子：

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

下面是一个使用 **集合 for** 在把一个集合的元素插入到另一个集合前操纵它们的例子：

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

要了解更多关于集合 if 和 for 的详情和例子，请参阅 控制流集合提案。

列表类型有许多方便的方法可以用来操作列表。要了解更多内容，请参阅 泛型 和 集合。

Sets

Dart 中的 set 是无序且唯一的元素集合。Dart 通过 set 字面量和 Set 类型来支持 set。

> 版本说明：尽管 Set 类型一直是 Dart 核心的一部分，set 字面量却是 Dart 2.2 中新引入的。

下面是一个简单的 Dart set，使用字面量创建：

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

> 说明：Dart 推断 **halogens** 的类型为 **Set<String>**。如果你试图添加错误的类型到这个 set 中，分析器或运行时会抛出一个错误。要了解更多信息，请参阅 类型推断。

要创建一个空的 set，使用 **{}** 并提供一个类型参数，或者使用 **{}** 指向带类型的 **Set**：

```
var names = <String>{};
// Set<String> names = {}; // 这样也可以
// var names = {}; // 创建一个 map，而不是 set
```

> **Set 还是 map?** Map 的字面量语法和 set 的字面量语法很相似。因为 map 字面量的优先级更高，**{}** 默认表示 **Map** 类型。如果你忘记了 **{}** 的类型注解或者它指向的变量，Dart 会创建一个类型为 **Map<dynamic, dynamic>** 的对象。

使用 **add()** 或 **addAll()** 来向已存在的 set 中添加元素：

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

使用 **.length** 来获取 set 中元素的数量：

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

要创建一个 set 作为编译期常量，在 set 字面量前使用 **const**。

```
final constantSet =
    const {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};
// constantSet.add('helium'); // 取消这一行的注释会引发一个错误
```

对于 Dart 2.3，set 支持扩展运算符 （... 和 ...?） 还有集合 if 和集合 for，就像列表那样。要了解更多信息，请参阅 列表 中的相关讨论。

要了解更多关于 set 的信息，请参阅 泛型 和 Set。

Maps

通常来说，映射是一个关联了键和值的对象。键和值都可以是任意类型的对象。“键”是唯一的，但是你可以多次使用相同的“值”。Dart 通过映射字面量和 Map 类型来支持映射。

下面是几个简单的 Dart 映射，使用字面量创建：

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

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

> 说明：Dart 推断 **gifts** 拥有类型 **Map<String, String>**，而 **nobleGases** 拥有类型 **Map<int, String>**。如果你试图添加错误的类型到上面的映射中，分析器或者运行时会报告一个错误。要了解更多信息，请参阅 类型推断。

你可以通过 Map 的构造函数创建同样的对象：

```
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';
```

> 说明：你可能对 **new Map()** 这样的形式会更熟悉。在 Dart 2 中，关键词 **new** 是可选的。详情请参阅 使用构造函数。

添加一个新的键值对到已存在的映射，方法和 JavaScript 一样：

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds'; // 添加一个键值对
```

从映射中取得一个值也和 JavaScript 的写法一样：

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

如果你查找一个不存在的键，会得到 null：

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

使用 **.length** 去获得映射中键值对的数量：

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

要创建一个作为编译期常量的映射，在映射字面量前加上 **const**：

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

// constantMap[2] = 'Helium'; // 这一行会引发一个错误
```

对于 Dart 2.3，映射支持扩展运算符 （... 和 ...?） 还有集合 if 和集合 for，就像列表那样。要了解更多信息，请参阅 列表 中的相关讨论。

要了解更多关于映射的内容，请参阅 泛型 和 Map。

Runes

在 Dart 中，runes 表示字符串中 UTF-32 编码的码位。

Unicode 为世界上所有的书写系统中的每个字母、数字和符号都定义了一个唯一数值。因为 Dart 的字符串是 UTF-16 码位的序列，在字符串中表示32位 Unicode 值需要特殊的语法。

表示一个 Unicode 码位的通常方式是 **\uXXXX**，其中 XXXX 是一个4位16进制数字。比如，心形符号 (♥) 表示为 **\u2665**。要指定多于或少于4位的16进制数字，将数字放到花括号里。比如，哈哈笑的 emoji (😆) 表示为 **\u{1f600}**。

字符串 类中有几个属性可以用来提取 rune 信息。**codeUnitAt** 和 **codeUnit** 属性返回16位编码单元。使用 **runes** 属性来获取一个字符串的 runes。

下面的例子展示了 runes、16位编码单元和32位编码单元的关系：

```
main() {
  var clapping = '\u{1f44f}';
  print(clapping);
  print(clapping.codeUnits);
  print(clapping.runes.toList());

  Runes input = new Runes(
      '\u2665  \u{1f605}  \u{1f60e}  \u{1f47b}  \u{1f596}  \u{1f44d}');
  print(new String.fromCharCodes(input));
}
```

> 说明：请谨慎使用列表操作来处理 runes。这些方法可能很容易失效，而且依赖特定的语言、字符集和具体的操作。要了解更多信息，请参阅 Stack Overflow 上的 How do I reverse a String in Dart?

Symbols

一个 Symbols 对象表示 Dart 程序中已声明的运算符或标识符。你可能永远都不需要用到 symbols，但是它们对于通过名称来标识引用的接口非常重要，因为缩写改变标识符的名字但不改变标识符的 symbols。

要获取一个标识符的 symbol，使用 symbol 字面量，语法是 **#** 后面跟上标识符：

```
#radix
#bar
```

Symbol 字面量是编译期常量。

函数

Dart 是一个完全的面向对象语言，所以甚至连函数也是对象，而且拥有一个类型 Function。这意味着函数可以被赋值给一个变量，或者作为参数传递给其他函数。你可以把一个 Dart 类实例作为函数来调用，只要它是一个函数。详情请参阅 可被调用的类。

下面的例子展示了如何实现一个函数：

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

尽管 Effective Dart 推荐 为公共API添加类型注释，但是如果你忽略了类型，函数依然是可用的：

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

对那些只包含一个表达式的函数，你可以使用简写的语法：

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

这里的 **=> *expre*** 语法是 **{ return *expr*; }** 的简写。符号 **=>** 有时被称为箭头语法。

> 说明：只有单个表达式——而不是语句——可以出现在箭头 (=>) 和 分号 (;) 的中间。比如，你不可以放 if 语句，但是可以使用 条件表达式。

函数有两种类型的参数：*必须参数*和*可选参数*。必须参数在参数列表的前面，可选参数跟在后面。可选参数可以是*命名参数*或*位置参数*。

> 说明：一些 API——特别是 Flutter 控件——只使用命名参数，即使这些参数是强制的。阅读下节以了解详情。

可选参数

可选参数可以是位置参数或者命名参数，但不可以两者兼是。

命名参数

当调用一个函数时，你可以通过 *参数名: 参数值* 的格式指定命名参数。比如：

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

当定义一个函数时，使用 *{参数1, 参数2, ...}* 的格式来指定命名参数：

```
/// 设置可选的"加粗"和"隐藏"标志
void enableFlags({bool bold, bool hidden}) {
  // ...
}
```

尽管命名参数是一种可选参数，你可以使用 @required 注解来声明这个参数是强制的——即用户必须为这个参数提供一个值。比如：

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

当某人试图创建 **Scrollbar** 而不指定 **child** 参数时，分析器会报告一个问题。

要使用 @required 注解，依赖 meta 包并且引入 **包：meta/meta.dart**。

Required 被定义在 meta 包中。可以直接导入 **package:meta/meta.dart**，也可以导入其他导出了 **meta** 的包，比如 Flutter 的 **package:flutter/material.dart**。

位置参数

包裹函数的参数集到 **[]** 中来表明它们是可选参数：

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

下面的例子展示调用该函数时不带可选参数：

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

下面的例子展示调用该函数时带上第三个参数：

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

参数默认值

你可以使用 = 来为函数参数定义默认值，可适用于命名参数和位置参数。默认值必须是编译期常量。如果没有提供默认值，默认值便是 **null**。

下面的例子展示为命名参数设置默认值：

```
/// 设置可选的 bold 和 hidden 标志
void enableFlags({bool bold = false, bool hidden = false}) {
  // ...
}

// bold 将会是true，hidden  将会是false
enableFlags(bold: true);
```

> 弃用说明：以前的代码可能会使用冒号 (:) 而不是 = 来设置命名参数的默认值。原因是之前只有 : 可以用来给命名参数设置默认值。而现在对 : 的支持可能会被废弃，所以我们推荐你 使用 = 来指定默认值。

下一个例子展示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');
```

你也可以使用列表或者映射作为默认值。下面的例子定义了一个函数 **doStuff()**，它为参数 **list** 指定了一个默认的列表，为参数 **gifts** 指定了一个默认的映射。

```
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

每一个应用都有一个顶级的 **main()** 函数作为这个应用的入口。**main()** 函数返回 **void** 而且有一个可选的参数，类型为 **List<String>**。

下面的例子是 web app 里面的 **main()** 函数：

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

> 说明：上面代码中的 .. 语法被称作 级联。使用级联，你可以对单个对象的成员们进行多次操作。

下面的例子是命令行程序中的 **main()** 函数，它接受命令行参数：

```
// 像这样运行该程序：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');
}
```

你可以使用 args 库 来定义和解析命令行参数。

函数作为一等对象

你可以把函数作为参数传递给其他函数。比如：

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

var list = [1, 2, 3];

// 把 printElement 作为参数
list.forEach(printElement);
```

你也可以把函数赋值给一个变量，比如：

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

这个例子使用了匿名函数。下一节会详细讲解匿名函数。

匿名函数

大部分函数都有一个名字，比如 **main()** 或者 **printElement()**。你也可以创建无名函数，它们叫做“匿名函数”，有时也被称为 "lambda" 或者 "闭包"。你可能会将匿名函数赋值给一个变量，以便后续使用，比如，你可以将它添加到一个集合，或者从集合移除。

匿名函数看起来就像一个命名函数——括号中0个或多个参数、逗号隔开、可选的类型注释。

之后的代码块包含了函数的主体：

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

下面的例子定义了一个匿名函数，包含一个无类型的参数 **item**。这个函数会从列表中的每一个元素调用，打印一个包含了在指定索引处的值的字符串。

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

如果这个函数只包含一个语句，你可以使用箭头符号简化它，它们的效果是等价的：

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

词法作用域

Dart 是词法作用域语言，意味着变量的作用域是静态确定的，简单地通过代码的布局来确定。你可以“沿着花括号向外走“来判断一个变量是否在作用域中。

下面是一个嵌套函数的例子，它包含了各个层级的作用域中的变量：

```
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()** 可以使用各个层级的变量，一直到最外层。

词法闭包

”闭包“指可以访问词法作用域中变量的一个函数对象，即使这个函数是在它原本作用域的外部被使用的。

函数可以捕获定义在它周围作用域中的变量。在下面的例子中，**makeAdder()** 捕获了变量 **addBy**。无论返回的函数到哪儿，它都记得 **addBy**。

```
/// 返回一个函数，该函数会添加 "addBy" 到
/// 这个函数的参数上并返回
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);
}
```

验证函数的相等性

下面的例子验证了顶级函数、静态函数和实例方法的相等性：

```
void foo() {} // 一个顶级函数

class A {
  static void bar() {} // 一个静态函数
  void baz() {} // 一个实例方法
}

void main() {
  var x;

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

  // 比较静态函数
  x = A.bar;
  assert(A.bar == x);

  // 比较实例方法
  var v = A(); // Instance #1 of A
```

```
  var w = A(); // Instance #2 of A
  var y = w;
  x = w.baz;

  // 闭包指向同一个实例，
  // 因此它们是相等的
  assert(y.baz == x);

  // 闭包指向不同的实例，
  // 因此它们是不相等的
  assert(v.baz != w.baz);
}
```

返回值

所有函数都有返回值。如果没有指定返回值，那么语句 **return null;** 会被隐式地添加到函数体上：

```
foo() {}

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

运算符

Dart 定义了下面表格中的这些运算符。你可以重写其中大部分的运算符，在 重载运算符 部分有更详细的描述。

描述	运算符
一元后缀	*expr++ expr--* () [] . ?.
一元前缀	*-expr !expr ~expr ++expr --expr*
乘除	* / % ~/
加减	+ -
移位	<< >>
按位与	&
按位异或	^
按位或	|
关系和类型检查	>= > <= < as is is!
相等性	== !=
逻辑与	&&
逻辑或	||
如果为空	??
条件	*expr1* ? *expr2* : *expr3*
级联	..
赋值	= *= /= ~/= %= += -= <<= >>= &= ^= |= ??=

使用运算符创建的是表达式。下面是一些表达式的例子：

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

在 运算符表 中，每一个表达式都比它下面的表达式优先级高。比如，取余运算符 **%** 的优先级就比等于运算符高（因此先执行），而等于运算符的优先级又比逻辑与运算符 **&&** 高。优先级意味着下面两行代码的执行方式相同：

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

// 可读性差，但是效果等价
if (n % i == 0 && d % i == 0) ...
```

> 警告：对于作用于两个操作数的操作符，最左边的操作数决定了哪个版本的运算符会被使用。比如，如果你有一个 Vector 对象和 Point 对象，**aVector + aPoint** 使用 Vector 版本的 +。

算术运算符

Dart 支持通常的算术运算符，如下表所示。

运算符	含义
+	加
-	减
-expr	取反（改变一个表达式的正负号）
*	乘
/	除
~/	整数除，返回整数结果
%	获取整数除的余数（取模）

例子：

```
assert(2 + 3 == 5);
assert(2 - 3 == -1);
assert(2 * 3 == 6);
assert(5 / 2 == 2.5); // 结果是 double
assert(5 ~/ 2 == 2); // 结果是 int
assert(5 % 2 == 1); // 取余

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

Dart 也支持前缀和后缀的自增、自减运算符：

运算符	含义
++var	var = var + 1（表达式的值是 var + 1）
var--	var = var + 1（表达式的值是 var）
--var	var = var -1（表达式的值是 var - 1）
var--	var = var - 1（表达式的值是 var）

例子：

```
var a, b;

a = 0;
b = ++a; // 在过取值之前自增
assert(a == b); // 1 == 1

a = 0;
b = a++; // 在取值之后自增
assert(a != b); // 1 != 0

a = 0;
b = --a; // 在取值之前自减
assert(a == b); // -1 == -1
```

```
a = 0;
b = a--; // 在取值之后自减
assert(a != b); // -1 != 0
```

相等和关系运算符

下表列出了相等和关系运算符的含义。

运算符	含义
==	相等；请参阅下面的讨论
!=	不等
>	大于
<	小于
>=	大于等于
<=	小于等于

要判断两个对象 x 和 y 是否代表相同的东西，使用 **==** 运算符。（在少数情况下，当你想知道两个对象是否是同一个对象，使用 identical()。）下面是 **==** 运算符的工作原理：

1. 如果 *x* 或者 *y* 是空，仅当两者都是空时返回 true，否则返回 false。
2. 返回方法 *x.==(y)* 的调用结果。（是的，像 **==** 这样的运算符就是在它第一个操作数上调用的方法。你甚至可以重载许多运算符，包括 **==**，详情请参阅 重载运算符。

这里是一些使用相等和关系运算符的例子：

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

类型检查运算符

运算符 **as**、**is** 和 **is!** 可以在运行期方便地进行类型检查。

运算符	含义
as	类型转换
is	如果对象拥有指定类型返回 true
is!	如果对象拥有指定类型返回 false

如果 **obj** 实现了接口 **T**，那么表达式 **obj is T** 的结果是 true。比如，**obj is Object** 总是返回 true。

使用 **as** 运算符将对象转换成指定类型。通常的，你应该使用它作为跟随在 **is** 后面使用该对象作为表达式这种情况的简写。比如，考虑以下代码：

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

你可以使用 **as** 运算符使代码更简洁：

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

> 说明：上面的代码并不是完全等价的。如果 **emp** 是空或者不是 Person，第一个例子（使用 is）什么都不做，而第二个（使用 as）会抛出一个异常。

赋值运算符

如你所见，你可以使用 **=** 运算符进行赋值。要在仅当被赋值的变量为空时才进行赋值，使用 **??=** 运算符。

```
// 赋值到 a
a = value;
// 如果 b 为空才赋值到 b；否则 b 保留原始值
b ??= value;
```

复合赋值运算符比如 **+=** 组合了一个其他运算符到赋值运算符。

=	**-=**	**/=**	**%=**	**>>=**	**^=**
+=	*=	~/	<<=	&=	|=

下面解释复合赋值运算符的原理：

	复合赋值	**等价的表达式**
对运算符 *op*：	a *op*= b	a = a *op* b
例子：	a += b	a = a + b

下面的例子同时使用了赋值运算符和复合赋值运算符：

```
var a = 2; // 使用 = 赋值
a *= 3; // 赋值并进行乘法：a = a * 3
assert(a == 6);
```

逻辑运算符

你可以使用逻辑运算符反转或者组合布尔表达式。

运算符	**含义**
!*expr*	反转后面表达式的值（false 变 true，true 变 false）
||	逻辑或
&&	逻辑与

下面是一个使用逻辑运算符的例子：

```
if (!done && (col == 0 || col == 3)) {
  // ...一些操作...
}
```

按位和移位运算符

在 Dart 中你可以操作数值中的各个位。通常你需要在整数上使用这些按位和移位运算符。

运算符	含义
&	按位与
|	按位或
^	按位异或
~expr	一元按位补码（0s 变成 1s；1s 变成 0s）
<<	左移位
>>	右移位

下面是使用按位和移位操作符的例子：

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

assert((value & bitmask) == 0x02); // 按位与
assert((value & ~bitmask) == 0x20); // 按位与、按位补码
assert((value | bitmask) == 0x2f); // 按位或
assert((value ^ bitmask) == 0x2d); // 按位异或
assert((value << 4) == 0x220); // 左移位
assert((value >> 4) == 0x02); // 右移位
```

条件表达式

Dart 有两个表达式可以让你简明地计算可能需要 if-else 语句的表达式：

condition ? expr1 : expr2

如果 *condition* 是 true，计算 *expr1*（并且返回它的值）；否则，计算并返回 *expr2* 的值。

expr1 ?? expr 2

如果 *expr1* 非空，返回它的值；否则，计算并返回 *expr2* 的值。

当你需要根据一个布尔表达式进行赋值时，考虑使用 **?:**。

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

如果布尔表达式进行的是空值判断，考虑使用 **??**。

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

上一个例子至少可以改写为以下两种形式，但是没有那么简洁：

```
// 比使用 ?: 运算符的长一点
String playerName(String name) => name != null ? name : 'Guest';

// 使用 if-else 语句的长版本
String playerName(String name) {
  if (name != null) {
    return name;
  } else {
    return 'Guest';
  }
}
```

级联符号

级联 (..) 允许你在同一个对象上进行一系列操作。除了方法调用，你也可以访问同一个对象的属性。这样通常可以让你免于创建临时变量并且写出更加顺畅的代码。

考虑以下代码：

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

第一个方法调用，**querySelector()**，返回一个选择器对象。作用在这个选择器对象上的级联标记后面的代码，会忽略所有随后可能会有的返回值。

前面的代码等同于：

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

你也可以嵌套级联。比如：

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

请小心地在那些返回真实对象的方法上构造级联。比如，下面的调用会失败：

```
var sb = StringBuffer();
sb.write('foo')
  ..write('bar'); // 错误：void 没有定义 'write' 方法
```

代码 **sb.write()** 返回 void，你不能在 void 上构造级联。

> 说明：严格地说，级联的"两点"标记不是一个运算符。它是 Dart 语法的一部分。

其他运算符

你已经在其他例子中看到过剩下的大部分运算符了：

运算符	名称	含义
()	函数调用	代表一个函数调用
[]	列表访问	引用列表中指定索引处的值
.	成员访问	引用一个表达式的属性；比如：**foo.bar** 选取了表达式 **foo** 的 **bar** 属性
?.	有条件的成员访问	类似 **.**，但是左操作数可以为空；比如：**foo.bar** 选取了表达式 **foo** 的 **bar** 属性除非 **foo** 是空（在这种情况下 **foo?.bar** 的值是空）

要了解更多关于 **.**、**?.** 和 **..** 的内容，请参阅 类。

控制流语句

你可以在 Dart 中使用下面任意方式控制代码的执行流程：

- **if** 和 **else**
- **for** 循环
- **while** 和 **do-while** 循环
- **break** 和 **continue**
- **switch** 和 **case**
- **断言**

你也可以使用 **try-catch** 和 **throw** 控制流程， 如 异常 中所述。

If 和 else

Dart 支持带 **else** 语句的 **if** 语句，如下面例子所展示的。另见 条件表达式。

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

不像 JavaScript，条件必须使用布尔值，而不是其他的。要了解更多内容，请参阅 布尔。

For 循环

你可以使用标准的 **for** 循环进行迭代。比如：

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

Dart for 循环中的闭包会捕获 index 的**值**，避免 JavaScript 中常见的一个坑。比如，考虑：

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

输出是 **0** 然后是 **1**，如期望一样。对比下来，在 JavaScript 中上面的例子会输出 **2** 和 **2**。

如果你要迭代的对象是一个 Iterable，你可以使用 forEach() 方法。如果你不需要知道当前迭代的计数，使用 **forEach()** 是一个不错的选择。

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

像列表和集合这样的 Iterable 也支持 **for-in** 形式的 迭代。

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

While 和 do-while 循环

While 循环在循环之前计算条件：

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

而 **do-while** 循环在循环后计算条件：

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

Break 和 continue

使用 **break** 中止循环：

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

使用 **continue** 跳到下一个循环：

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

如果你在用 Iterable，比如列表或者集合，你可能会以不同的方式编写上面的例子:

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

Switch 和 case

Dart 中的 switch 语句使用 **==** 比较整数、字符串或其他编译期常量。被比较的对象必须全部是相同的类的实例（而不是它的任何子类），而且该类必须不能重载 **==**。**枚举类** 在 **switch** 语句中可以良好地运行。

> 说明：Dart 中的 switch 语句只适用于有限的情况，就像翻译员和扫描仪。

作为规定，每一个非空的 case 子句都以 **break** 结束。结束一个非空 **case** 子句的其他方式是 **continue**、**throw** 或者 **return** 语句。

在没有条件匹配的情况下，使用 **default** 字句执行代码：

```
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();
}
```

下面的例子遗漏了 **case** 子句中的 **break**，这会产生一个错误：

```
var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // 错误：缺少 break

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

然而，Dart 确实支持空的 **case** 子句，来允许 fall-through 形式：

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED': // 空子句 fall-through
  case 'NOW_CLOSED':
    // CLOSED 和 NOW_CLOSED 都会执行
    executeNowClosed();
    break;
}
```

如果你真的想 fall-through，你可以使用 **continue** 语句并加上一个标签：

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED':
    executeClosed();
    continue nowClosed;
  // 在 nowClosed 标签处继续执行

  nowClosed:
  case 'NOW_CLOSED':
    // CLOSED 和 NOW_CLOSED 都会执行
    executeNowClosed();
    break;
}
```

一个 **case** 子句可以有局部变量，这些变量仅在该子句中可见。

断言

在开发时，如果一个布尔值为 false，使用 **断言** 语句——**assert(*condition*, *optionalMessage*)**——来中止正常的执行。你可以在本教程中找到断言语句的例子。下面就是一些：

```
// 确保变量是非空的值
assert(text != null);

// 确保变量小于100
assert(number < 100);

// 确保变量是一个 https 的 URL
assert(urlString.startsWith('https'));
```

要添加信息到一个断言，使用一个字符串作为 **assert** 的第二个参数。

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

传递给 **assert** 的第一个参数只能是可解析为布尔值的表达式。如果这个表达式的值是 true，断言成功并且继续执行。如果它是 false，断言失败并且抛出一个异常（一个 AssertionError）。

断言到底何时起作用？这取决于你所使用的工具和框架：

- Flutter 在调试模式下启动断言。
- 开发专用工具像是 dartdevc 通常默认支持断言。
- 还有一些工具，像是 dart 和 dart2js，通过命令行标志来支持断言：**--enable-asserts**。

在生产环境的代码中，断言会被忽略，而且 **assert** 的参数也不会被求值。

异常

你的 Dart 代码可以抛出和捕获异常。异常是指发生了未意料的错误。如果异常没被捕获，抛出异常的 isolate 会被挂起，一般情况下这会导致 isolate 和 应用程序终止。

与 Java 相反，Dart 中所有的异常都是未检查异常。方法不声明它们可能会抛出的异常，而且你没有被要求捕获任何异常。

Dart 提供了 Exception 和 Error 类型，以及众多预定义的子类。当然，你可以定义自己的异常。然而，Dart 程序可以抛出任何非空的对象——而不仅仅是 Exception 和 Error 对象——作为异常。

Throw

下面是抛出一个异常的例子：

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

你也可以抛出任意对象：

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

> 说明：符合生产质量的代码通常抛出实现了 Error 或 Exception 的类型。

因为抛出异常是一个表达式，你可以在 => 语句中抛出异常，以及任何允许表达式的地方抛出异常：

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

Catch

捕获异常会中止异常的传播（除非你又重新抛出这个异常）。捕获异常给了你处理它的机会：

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

要处理抛出超过一种类型异常的代码，你可以指定多个 catch 子句。第一个匹配抛出对象类型的子句处理这个异常。如果 catch 子句没有指定类型，那么这个子句可以处理任意类型的抛出对象：

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  // 一个指定的异常
  buyMoreLlamas();
} on Exception catch (e) {
  // 任何是异常的对象
  print('Unknown exception: $e');
} catch (e) {
  // 未指定类型，处理所有对象
  print('Something really unknown: $e');
}
```

如前面代码所示，你可以使用 **on** 或 **catch** 或两者同时使用。当你需要指定类型时使用 **on**。当你的异常处理需要异常对象时使用 **catch**。

你可以为 **catch()** 指定一个或两个参数。第一个参数是被抛出的对象，第二个是调用栈（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');
}
```

要部分处理一个异常，而允许它继续传播，使用 **rethrow** 关键词。

```
void misbehave() {
  try {
    dynamic foo = true;
    print(foo++); // 运行期异常
  } catch (e) {
    print('misbehave() partially handled ${e.runtimeType}.');
    rethrow; // 允许调用者看到这个异常
  }
}

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

Finally

要确保一些代码无论是否有异常抛出都会执行，使用 **finally** 子句。如果没有 **catch** 子句匹配异常，异常在 **finally** 子句执行后继续传播：

```
try {
  breedMoreLlamas();
} finally {
  // 总是执行清理，即使有异常抛出
  cleanLlamaStalls();
}
```

Finally 子句在任意匹配的 **catch** 子句后执行：

```
try {
  breedMoreLlamas();
} catch (e) {
  print('Error: $e'); // 先处理异常
} finally {
  cleanLlamaStalls(); // 然后执行清理
}
```

阅读库教程中的 Exceptions 章节来了解更多内容。

类

Dart 是一门面向对象的编程语言，具备类和基于混入的继承。

每一个对象都是一个类的实例，而所有的类都派生自 Object。“基于混入的继承”意味着虽然每个类（除了 Object）都只有一个父类，但类的主体可以在多个类层级中被复用。

使用类成员

对象包含由函数和数据（分别是“方法”和“实例变量”）组成的“成员”。当你调用一个方法时，你在一个对象上“调用”：这个方法可以访问该对象的函数和数据：

使用一个点 (.) 来引用实例变量或方法：

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

// 设置实例变量 y 的值
p.y = 3;

// 获取 y 的值
assert(p.y == 3);

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

使用 **?.** 代替 **.** 来避免当左操作数为空时会引发的异常：

```
// 如果 p 是非空值，设置它的 y 值为 4
p?.y = 4;
```

使用构造函数

你可以使用”构造函数“创建一个对象。构造函数的名字可以是 ***ClassName*** 或 ***ClassName.identifier***。比如，下面的代码使用 **Point()** 和 **Point.fromJson()** 构造函数创建了 **Point** 对象：

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

下面的代码具有相同的效果，但是在构造函数前使用了可选的 **new** 关键词：

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

> 版本说明：关键词 **new** 在 Dart 2 中变成了可选的。

一些类提供 常量构造函数。要使用构造函数创建一个编译期常量，在构造函数名前面加上 **const** 关键词：

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

构造两个相同的编译期常量结果会是同一个、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
```

在一个"常量上下文"中，你可以省略构造函数或字面量前的 **const**。比如，下面代码中，创建常量映射：

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

除了第一个以外，你可以省略其他所有的 **const** 关键词：

```
// 只有一个 const，它创建了常量上下文
const pointAndLine = {
  'point': [ImmutablePoint(0, 0)],
  'line': [ImmutablePoint(1, 10), ImmutablePoint(-2, 11)],
};
```

如果一个常量构造函数在常量上下文之外并且没有使用 **const** 来调用，它会创建一个 **非常量对象**：

```
var a = const ImmutablePoint(1, 1); // 创建一个常量
var b = ImmutablePoint(1, 1); // 不会创建一个常量

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

> 版本说明：常量上下文中的 **const** 关键词在 Dart 2 中变成了可选的。

获取对象类型

要获取一个对象的类型，你可以使用对象的 **runtimeType** 属性，会返回一个 Type 对象。

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

到此为止，你已经看到了如何"使用"类。本章余下的内容会展示如何"实现"类。

实例变量

你可以使用如下方式声明实例变量：

```
class Point {
  num x; // 声明一个实例变量，初始值为 null
  num y; // 声明 y，初始值为 null
  num z = 0; // 声明 z，初始值为 0
}
```

所有未初始化的实例变量值都为 **null**。

所有的实例变量都生成隐式的 *getter* 方法。非 final 的实例变量同时生产一个隐式的 *setter* 方法。详情请参阅 Getter 和 setter。

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

void main() {
  var point = Point();
  point.x = 4; // 使用 x 的 setter 方法
  assert(point.x == 4); // 使用 x 的 getter 方法
  assert(point.y == null); // 默认值为 null
}
```

如果你在声明的时候初始化实例变量（而不是在构造函数或者方法里），值会在实例创建的时候被设置，在构造函数和它的初始化列表执行前。

构造函数

通过创建一个和类名一样（或者类名加上一个可选的、额外的标识符作为命名构造函数）的方法，来声明一个构造函数。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) {
    // 有更好的实现方式，请看下文分解
    this.x = x;
    this.y = y;
  }
}
```

关键词 **this** 引用当前实例。

> 说明：仅当有命名冲突时使用 **this**。否则，Dart 的风格是省略 **this**。

将构造函数的参数赋值给一个实例变量，这种模式是如此常见，因此，Dart 有语法糖来简化操作：

```
class Point {
  num x, y;

  // 设置 x 和 y 的语法糖
  // 在构造函数体之前执行
  Point(this.x, this.y);
}
```

默认构造函数

如果你没有声明构造函数，一个默认构造函数会提供给你。默认构造函数没有参数，并且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。一个没有声明构造函数的子类只拥有默认的（无参、无名字的）构造函数。

命名构造函数

使用命名构造函数来实现多个构造函数或者让代码更清晰：

```
class Point {
  num x, y;

  Point(this.x, this.y);

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

记住构造函数不被继承，意味着父类的命名构造函数不会被子类继承。如果你希望用父类中的命名构造函数创建子类，你必须在子类中实现该构造函数。

调用父类的非默认构造函数

默认地，子类的构造函数会调用父类的无名、无参构造函数。父类的构造函数会在构造函数体的一开始被调用。如果 初始化列表 也被使用了，它在父类被调用之前调用。总结下来，执行的顺序如下：

1. 初始化列表
2. 父类的无参构造函数
3. 主类的无参构造函数

如果父类没有无名、无参的构造函数，那么你必须手动调用父类的其中一个构造函数。在冒号 (:) 后面，构造函数体之前（如果有的话）指定父类的构造函数。

下面的例子中，Employee 类的构造函数调用了它父类 Person 的命名构造函数。

```
class Person {
  String firstName;

  Person.fromJson(Map data) {
    print('in Person');
  }
```

```
}

class Employee extends Person {
  // Person 没有默认构造函数
  // 你必须调用 super.fromJson(data)
  Employee.fromJson(Map data) : super.fromJson(data) {
    print('in Employee');
  }
}

main() {
  var emp = new Employee.fromJson({});

  // 打印:
  // in Person
  // in Employee
  if (emp is Person) {
    // 类型检查
    emp.firstName = 'Bob';
  }
  (emp as Person).firstName = 'Bob';
}
```

由于父类构造函数的参数在构造函数调用前被计算，参数可以是一个表达式比如一个函数调用：

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

> 警告：父类的构造函数不能访问 **this**。因此，参数可以是静态方法但是不能是实例方法。

初始化列表

调用父类构造函数的同时，你也可以在构造函数体执行之前初始化实例变量。使用逗号分隔初始化器。

```
// 初始化列表在构造函数体执行前设置实例变量的值
Point.fromJson(Map<String, num> json)
    : x = json['x'],
      y = json['y'] {
  print('In Point.fromJson(): ($x, $y)');
}
```

> 警告：初始化器右边不能访问 **this**。

在开发阶段，你可以在初始化列表中使用 **assets** 验证输入。

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

初始化列表是设置 final 属性的方便方法。下面的例子在初始化列表中初始了三个 final 属性。

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

```
class Point {
  final num x;
  final num y;
  final num distanceFromOrigin;

  Point(x, y)
      : x = x,
        y = y,
        distanceFromOrigin = sqrt(x * x + y * y);
}

main() {
  var p = new Point(2, 3);
  print(p.distanceFromOrigin);
}
```

重定向构造函数

有时候一个构造函数的唯一目的是重定向到同一个类的另一个构造函数。一个重定向构造函数的函数体是空的，构造函数的调用在冒号 (:) 后面。

```
class Point {
  num x, y;

  // 该类的主调用函数
  Point(this.x, this.y);

  // 代理到主构造函数
  Point.alongXAxis(num x) : this(x, 0);
}
```

常量构造函数

如果你的类生成的对象从不改变，你可以让这些对象变成编译期常量。要想这样，定义一个**常量**构造函数并确保所有实例变量都是 **final** 的。

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

  final num x, y;

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

常量构造函数并不总是会创建常量。要了解详情，请看 使用构造函数 章节。

工厂构造函数

当要实现一个不总是创建这个类新实例的构造函数时，使用 **factory** 关键词。比如，一个工厂构造函数可能从缓存中返回一个实例，或者可能返回子类的一个实例。

下面的代码展示了一个工厂构造函数从缓存中返回对象：

```
class Logger {
  final String name;
```

```
  bool mute = false;

  // _cache 是库内私有的，多亏了它名字前的 _
  static final Map<String, Logger> _cache =
      <String, Logger>{};

  factory Logger(String name) {
    return _cache.putIfAbsent(
        name, () => Logger._internal(name));
  }

  Logger._internal(this.name);

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

> 说明：工厂构造函数无法访问 **this**。

调用工厂构造函数的方式和其他构造函数一样：

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

方法

方法指那些为一个对象提供行为的函数。

实例方法

对象上的实例方法可以访问实例变量和 **this**。下面代码中的 **distanceTo()** 方法就是一个实例方法的例子：

```
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);
  }
}
```

Getters 和 setters

Getters 和 setters 是为一个对象的属性提供读写权限的特殊方法。回想每一个实例变量都有一个隐式的 getter，符合条件的还会有一个 setter。你可以通过实现 getters 和 setters 创建额外的属性，使用 **get** 和 **set** 关键词：

```
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);
}
```

通过 getters 和 setters，你可以从实例变量起步，之后使用方法封装它们，整个过程不需要修改代码。

> 说明：无论是否明确定义 getter，像自增 (++) 这样的表达式都会以预期的方式执行。为避免任何非预期的副作用，运算符只会调用 getter 一次，然后保存它的值到一个临时变量中。

抽象方法

实例方法、getter 和 setter 都可以是抽象的，这样即定义了一个接口但是把它的实现留给其他类。抽象方法只能存在于 抽象类 中。

要使一个方法变得抽象，使用分号 (;) 代替方法体：

```
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() {
    // 提供一个实现，所以该方法在这里不是抽象的
  }
}
```

抽象类

使用 **abstract** 修饰符定义一个“抽象”类——一个不能被实例化的类。抽象类在定义接口时是有用的，通常附带一些实现。如果你想让你的抽象类变成可实例化的，定义一个 工厂构造函数。

抽象类经常包含 抽象方法。下面是一个定义抽象类的例子，它包含一个抽象方法：

```
// 该类定义为抽象的，因此无法被实例化
abstract class AbstractContainer {
  // 定义构造函数、属性、方法...

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

隐式接口

每一个类都隐式地定义了一个包括它所有实例成员和它实现的任意接口的接口。如果你希望创建一个类 A 来支持类 B 的 API 但是又不继承 B 的实现，类 A 应该实现 B 接口。

一个类要实现一个或多个接口，在 **implements** 子句中定义它们然后提供这些接口需要的 API 的实现。比如：

```
// Person 类。隐式接口包含 greet()
class Person {
  // 在接口中，但是只在这个库中可见
  final _name;

  // 不在接口中，因为这是一个构造函数
  Person(this._name);

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

// 一个 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()));
}
```

下面是一个类实现多个接口的例子：

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

继承类

使用 **extends** 来创建一个子类，使用 **super** 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。你可以使用 **@override** 注解来表明你想要重写一个成员：

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

要缩小一个方法参数或者实例变量的类型并写出 类型安全 的代码，你可以使用 **covariant** 关键词。

重载运算符

你可以重载下表中展示的运算符。比如，如果你定义一个 Vector（矢量）类，你可能会定义一个 **+** 方法来加两个矢量。

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

> 说明：你可能注意到 **!=** 不是一个可重载的运算符。表达式 **e1 != e2** 仅仅是 **!(e1 == e2)** 的语法糖。

下面是一个重载 **+** 和 **-** 运算符的例子：

```
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 hasCode 没有展示。详情请看下面的说明
  // ···
}

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

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

如果你重载 **==**，你也需要重载对象的 **hashCode** getter。有关重载 **==** 和 **hasCode** 的例子，请参阅 实现映射的键。

要了解更多关于重载的内容，一般来说，可以参阅 继承类。

noSuchMethod()

要检测或响应代码试图使用不存在的方法或实例变量的情况，你可以重写 **noSuchMethod()**：

```
class A {
  // 除非你重写 noSuchMethod，不然使用
  // 不存在的成员会引发 NoSuchMethodError
  @override
  void noSuchMethod(Invocation invocation) {
    print('You tried to use a non-existent member: ' +
        '${invocation.memberName}');
  }
}
```

你**不可以调用**一个未实现的方法除非以下情况有**一个**成立：

- 接收者有静态类型 **dynamic**。
- 接收者有一个定义了该未实现方法的静态类型（抽象也可），而且接收者的动态类型有不同于 **Object** 类 **noSuchMethod()** 的实现。

要了解更多信息，请参阅 noSuchMethod 跳转规范。

枚举类型

枚举类型，通常被称作 *enumerations* 或 *enums*（枚举），是用来表示有固定数量的常量值的一种特殊类。

使用枚举

使用 **enum** 关键词声明一个枚举类型：

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

枚举中的每一个值都有一个 **index** getter，返回枚举声明中基于0的位置索引。比如，第一个值有索引 0，而第二个值有索引 1.

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

要获取枚举中所有值的列表，使用枚举的 **values** 常量。

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

你可以在 switch 语句 中使用枚举，而且如果你没有处理所有的枚举值，你会得到一个警告：

```
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，你会看到一个警告
    print(aColor); // 'Color.blue'
}
```

枚举类型有以下限制：

- 你不可以继承、混入或实现一个枚举。
- 你不可以显式实例化一个枚举。

要了解更多内容，请参阅 Dart 语言规范。

为类添加特性：混入

混入 (mixin) 是在类的多继承中复用类代码的一种方式。

要“使用”混入，使用 **with** 关键词跟着一个或多个混入的名字。下面的例子展示了两个使用混入的类：

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

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

要实现一个混入，创建一个类继承 Object，不声明构造函数。除非你希望 mixin 可用作常规类，否则请使用 mixin 关键字代替 class。比如：

```
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');
    }
  }
}
```

要指定只有某些类型可以使用这个混入——比如，这样你的混入就可以调用它没有定义的方法——使用 on 来指定所需的父类：

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

> 版本说明：对于 **mixin** 关键词的支持在 Dart 2.1 中被引入。之前版本的代码通常使用 **abstract class** 代替。要了解更多关于混入在 2.1 中的改变，请参阅 Dart SDK 变更日志 和 2.1 混入规范。

类变量和方法

使用 **static** 关键词来实现类级别的变量和方法。

静态变量

静态变量（类变量）对类级别的状态和常数是很有用的。

```
class Queue {
  static const initialCapacity = 16;
  // ···
}

void main() {
  assert(Queue.initialCapacity == 16);
}
```

静态变量直到它们被使用才会初始化。

> 说明：该页面遵守 代码规范推荐 倾向于使用 "小驼峰"来作为常量名。

静态方法

静态方法（类方法）不操作实例，因此不能访问 **this**。比如：

```
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);
}
```

> 说明：对常见或者广泛使用的实用工具和功能，考虑使用顶级函数，而不是静态方法。

你可以使用静态方法作为编译期常量。比如，你可以把静态方法作为一个常量构造函数的参数。

泛型

如果你查看基本数组类型 List 的 API 文档，你会发现它的类型其实是 **List<E>**。<...> 标记表示 List 是一个"泛型"（或带参数的）类——具有形式上的类型参数的类型。按照惯例，类型变量的名字是单个字母，比如 E，T，S，K，和 V。

为什么用泛型?

泛型通常是类型安全的要求，但它们除了让你的代码可以运行外还有诸多益处：

- 正确地指定泛型类型会产生更好的代码。
- 你可以使用泛型来减少代码重复。

如果你只想让一个列表包含字符串，你可以指定它为 **List<String>**（读作"字符串列表"）。这样一来，你、你的同事和你的工具可以检测到将一个非字符串对象添加到该列表是错误的。下面是一个例子：

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

使用泛型的另一个原因是为了减少代码重复。泛型使你在多个不同类型间共享同一个接口和实现，而依然享受静态分析的优势。比如说，你要创建一个缓存对象的接口：

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

你发现需要一个此接口的字符串版本，所以你创建了另一个接口：

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

之后，你觉得你需要一个该接口的数值版本……你应该可以理解了。

泛型可以让你省去创建所有这些接口的麻烦。取而代之，你可以创建一个单一的接口并接受一个类型参数：

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

在这段代码中，T 是替身类型。它是一个占位符，你可以将其视为开发者稍后定义的类型。

使用集合字面量

列表和映射字面量可以是参数化的。参数化的字面量就像你之前见过的字面量，只是在左括号前加上了 ***<type>***（对于列表）或 ***<keyType, valueType>***（对于映射）。下面是一个使用类型字面量的例子：

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

在构造函数中使用参数类型

使用构造函数时要指定一个或多个类型，可以将类型放在类名后面的尖括号 (**<...>**) 中。比如：

```
var names = List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
var nameSet = Set<String>.from(names);
```

下面的代码创建了一个有整数键和 View 类型值的映射：

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

泛型集合和它们包含的类型

Dart 的泛型类是“实体化”的，这意味着它们在运行期携带了自己的类型信息。因此，你可以检测一个集合的类型：

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

> 说明：作为对照，Java 中的泛型使用“擦除”，意味着泛型信息在运行时被移除。在 Java 中，可以检测一个对象是否是一个 List，但是你不能检测它是否是一个 List<String>。

限制参数类型

当实现一个泛型时，你可能想要限制它的参数类型。你可以使用 **extends** 做到这点。

```
class Foo<T extends SomeBaseClass> {
  // 下面是实现
  String toString() => "Instance of 'Foo<$T>'";
}

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

使用 **SomeBaseClass** 或者它的子类作为泛型参数是可以的：

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

不使用泛型参数也是可以的：

```
var foo = Foo();
print(foo); // 'Foo<SomeBaseClass>' 的实例
```

指定任意非 **SomeBaseClass** 类型会得到一个错误：

```
var foo = Foo<Object>();
```

使用泛型方法

起初，Dart 的泛型支持仅限于类。一个新的语法，称为"泛型方法"，允许在方法上使用类型参数：

```
T first<T>(List<T> ts) {
  // 做一些初始化工作或者错误检查
  T tmp = ts[0];
  // 做一些额外的检查或处理
  return tmp;
}
```

这里 **first** (**<T>**) 中的泛型参数允许你在以下几个地方使用类型参数 **T**：

- 在函数的返回类型中 (**T**)。
- 在参数的类型中 (**List<T>**)。
- 在局部变量的类型中 (**T tmp**)。

要了解关于泛型的更多信息，请参阅 使用泛型方法。

库和可见性

指令 **import** 和 **library** 可以帮你创建一个模块化和可共享的代码库。库不仅提供 API，也是一个隐私单位：以下划线 (_) 开头的标识符只在库中可见。"每个 Dart 应用都是一个库"，即使它没有使用 **library** 指令。

库可以通过 包 来发布。

使用库

使用 **import** 指令来指定一个库在其他库的作用域内如何被使用。

比如，Dart 网页应用通常使用 dart:html 库，它可以像这样被引入：

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

指令 import 唯一需要的参数是指定了这个库的 URI。对于内置库，URI 有特殊的 **dart:** 格式。对于其他更多的库，你可以使用一个文件系统路径或者 **package:** 格式。**package:** 格式指定由包管理器比如 pub 工具提供的库。比如：

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

> 说明：*URI* 指统一资源标识符。**URL** （统一资源定位符）是一种常见的 URI。

指定库前缀

如果你导入两个有标识符冲突的库，那么你可以为其中一个或两个指定前缀。比如，如果库1和库2都有一个 Element 类，那么你的代码可能是这样的：

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

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

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

只导入一个库的部分

如果你只想要使用一个库的部分，你可以选择性地导入库。比如：

```dart
// 只导入 foo
import 'package:lib1/lib1.dart' show foo;

// 导入除 foo 之外的所有名称
import 'package:lib2/lib2.dart' hide foo;
```

懒加载一个库

”延迟加载“（也称为”懒加载“）允许一个 web 应用按需加载一个库，仅当这个库被需要时。下面是一些你可能会使用延迟加载的情况：

- 为了减少 web 应用的初始启动时间。
- 为了执行 A/B 测试——尝试一个算法的替代实现。
- 为了加载很少使用的功能，比如可选的页面和对话框。

> 说明：仅 dart2js 支持懒加载。Fultter 和 Dart VM，还有 dartdevc 不支持懒加载。要了解更多信息，请参阅 issue #33118 和 issue #27776。

要懒加载一个库，你必须先使用 **deferred as** 导入它：

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

当你需要这个库时，使用这个库的标识符调用 **loadLibrary()** 。

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

在上面的代码中，**await** 关键字暂停执行直到这个库加载完成。要了解更多关于 **async** 和 **await** 的信息，请参阅 异步支持。

你可以在一个库上多次调用 **loadLibrary()** 而不用担心会有问题。这个库只会被加载一次。

当使用延迟加载时记住以下几点：

- 一个延迟加载库中的常量在导入的文件中的不是常量。记住，这些常量直到库被加载前都是不存在的。
- 你不能使用延迟加载库中的类型。取而代之地，考虑将接口类型移到另一个库中，而这个库被延迟加载的库和导入的文件导入。
- Dart 隐式地将 **loadLibrary()** 插入到你使用 **deferred as *namespace*** 定义的命名空间中。函数 **loadLibrary()** 返回一个 Future。

实现库

要获取关于如何实现一个库包的建议，请参阅 创建一个库包，包括：

- 如果组织库中的源代码。
- 如果使用 **export** 指令。
- 何时使用 **part** 指令。
- 何时使用 **library** 指令。

异步支持

Dart 的库充满了返回 Future 或 Stream 对象的函数。这些函数是“异步的”：它们在设置一个可能比较耗时的操作（比如 I/O）后返回，而不去等待操作完成。

关键字 **async** 和 **await** 支持异步编程，可以使你用看起来像同步的方式编写异步代码。

处理 Futures

当你需要一个已完成的 Future 的结果时，你有两个选择：

- 使用 **async** 和 **await**。
- 使用 Future API，如 Dart 库教程 中所述。

使用 **async** 和 **await** 的代码是异步的，但看起来像同步代码。比如，下面的代码使用 **await** 来等待一个异步函数的返回：

```
await lookUpVersion();
```

要使用 **await**，代码必须在一个“异步函数”中——一个标记为 **async** 的函数：

```
Future checkVersion() async {
  var version = await lookUpVersion();
  // 使用 version 做一些操作
}
```

> 说明：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，异步函数只在遇到第一个 await 表达式时执行（详情）。然后它返回一个 Future 对象，仅在 await 表达式完成后才恢复执行。

在使用 **await** 的代码中，可以使用 **try**、**catch** 和 **finally** 来处理错误和清理代码。

```
try {
  version = await lookUpVersion();
} catch (e) {
  // 处理查找版本号失败的情况
}
```

你可以在一个异步函数中多次使用 **await**。比如，下面的代码等待了三次函数的结果：

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

在 **await *expression*** 中，*expression* 的值通常是一个 Future；如果它不是，这个值会自动封装成一个 Future。这个 Future 对象表示返回一个对象的承诺。而 **await *expression*** 的值就是这个返回的对象。Await 表达式会导致执行暂停直到这个对象可用。

如果你在使用 await 时遇到了编译期错误，请确保 await 在异步函数内。比如，要在你的应用的 **main()** 函数中使用 **await**，**main()** 的函数体必须标记为 **async**：

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

声明异步函数

“异步函数”就是函数体被 **async** 修饰符标记的函数。

添加 **async** 关键词到一个函数会使它返回一个 Future。比如，考虑使用同步函数，返回一个字符串：

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

如果你修改它为一个异步函数——比如，因为之后的一个实现会是耗时的——它的返回值是一个 Future：

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

注意函数体并不一定要使用 Future API。Dart 会在必要的时候自动创建 Future。如果你的函数不返回有用的值，使它返回 **Future<void>**。

有关使用 future、async 和 await 的互动介绍，请参阅异步编程实验室。

处理 Streams

当你需要从一个 Stream 中获取值是，你有两个选择：

- 使用 **async** 和一个“异步 for 循环“ (**await for**)。
- 使用 Stream API，如 Dart 库教程 中所述。

> 说明：在使用 **await for** 前，请确保代码足够清晰并且你真的想要等待 stream 中所有的结果。比如，你通常**不**需要对 UI 事件监听器使用 **await for**，因为 UI 框架不停地发送事件的 stream。

一个异步 for 循环拥有下面的格式：

```
await for (varOrType identifier in expression) {
  // 每当 stream 发送一个值时执行一次
}
```

Expression 的值必须是 Stream 类型。按照如下步骤执行：

1. 等待 stream 发送一个值。
2. 执行 for 循环的循环体，使用发送的值作为变量值。
3. 重复 1 和 2 直到 stream 被关闭。

要停止监听这个 stream，你可以使用 **break** 或者 **return** 语句，它们会打断 for 循环并且取消对 stream 的订阅。

如果你在使用异步 for 循环时遇到了编译期错误，请确保 await for 在一个异步函数内。比如，要在你的应用的 **main()** 函数中使用异步 for 循环，**main()** 的函数体必须标记为 **async**：

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

要了解更多关于异步编程的信息，通常地，参阅 dart:async 部分的库文档。

生成器

当你需要懒惰地生成一系列的值时，考虑使用一个“生成器函数” (*generator function*)。Dart 对这类生成器函数有内置的支持：

- **同步的**生成器：返回一个 Iterable 对象。
- **异步的**生成器：返回一个 Stream 对象。

要实现一个**同步的**生成器函数，使用 **sync*** 标记函数体，并使用 **yield** 语句传递值：

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

要实现一个**异步的**生成器函数，使用 **async*** 标记函数体，并使用 **yield** 语句传递值：

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

如果你的生成器是递归的，可以使用 **yield*** 增进它的性能：

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

可被调用的类

要使你的 Dart 类实例像函数一样可以被调用，实现 **call()** 方法。

在下面的例子中，**WannabeFunction** 类定义了一个 call() 函数，它接受三个字符串参数并且连接它们，使用一个空格分隔每个字符串，最后附加一个感叹号。

```
class WannabeFunction {
  call(String a, String b, String c) => '$a $b $c!';
}

main() {
  var wf = new WannabeFunction();
  var out = wf("Hi","there,","gang");
  print('$out');
}
```

Isolates

大部分计算设备，即使在移动平台上，都拥有多核 CPU。要发挥所有这些核心的优势，开发者通常使用共享内存的线程来实现并发执行。然而，共享状态的并发容易出错并且导致复杂的代码。

Dart 的代码在 *isolates* 中执行，而不是线程。每个 isolate 都有它自己的内存栈，保证了没有其他的 isolate 可以访问。

要了解更多信息，请参阅以下内容：

- Dart 异步编程：Isolates 和事件循环
- dart:isolate API索引，包括 Isolate.spawn() 和 TransferableTypedData
- Flutter 网站上的后台解析教程

Typedefs

在 Dart 中，函数是对象，就像字符串和数字是对象一样。一个 *typedef*，或者叫做"函数类型别名"，给函数类型起了一个名字，使你可以在定义字段和返回值类型时使用。在将一个函数类型赋值给一个变量时，一个 typedef 保留了类型信息。

考虑以下代码，不使用 typedef 的情况：

```
class SortedCollection {
  Function compare;

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

// 初始化，部分实现
int sort(Object a, Object b) => 0;

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

  // 我们只知道 compare 是一个函数，
  // 但它是什么类型的函数呢？
  assert(coll.compare is Function);
}
```

在将 **f** 赋值给 **compare** 时类型信息丢失了。**f** 的类型是 (Object, Object) → int（箭头的意思是返回），而 **compare** 的类型是 Function。如果我们修改一下代码，来使用明确的名字并且保留类型信息，开发者和工具都可以使用这个类型信息。

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

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

// 初始化，部分实现
int sort(Object a, Object b) => 0;

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

> 说明：目前，typedefs 仅可用于函数类型。我们预计这会有所改变。

因为 typedef 只是简单的别名，所以它们提供了一种方式来检查任意函数的类型。比如：

```
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 (metadata) 来给你的代码提供额外的信息。一个元数据注解以字符 **@** 开头，后面跟着的要么是编译期常量（比如 **deprecated**），要么是常量构造函数的调用。

有两个注解可应用于所有的 Dart 代码：**@deprecated** 和 **@override**。使用 **@override** 的例子，请参阅 继承类。下面是一个使用 **@deprecated** 注解的例子：

```
class Television {
  /// _已废弃: 使用 [turnOn] 代替_
  @deprecated
  void activate() {
    turnOn();
  }

  /// 开启 TV 的电源
  void turnOn() {...}
}
```

你可以定义自己的元数据注解。下面是定义一个接受两个参数的 @todo 注解的例子：

```
library todo;

class Todo {
  final String who;
  final String what;

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

然后下面是一个使用 @todo 注解的例子：

```
import 'todo.dart';

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

元数据可以出现在库、类、typedef、类型参数、构造函数、工厂构造函数、函数、字段、参数或变量声明前以及导入导出指令前。你可以在运行期通过反射取回元数据。

注释

Dart 支持单行注释、多行注释和文档注释。

单行注释

一个单行注释以 **//** 开头。所有在 **//** 和行尾的东西都被 Dart 编译器所忽略。

```
void main() {
  // TODO: 重构成一个 AbstractLlamaGreetingFactory?
  print('Welcome to my Llama farm!');
}
```

多行注释

一个多行注释开始于 **/*** 结束于 ***/**。所有在 **/*** 与 ***/** 之间的东西都会被 Dart 编译器所忽略（除非这个注释是一个文档注释；请看下一节）。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

多行注释是以 /// 或 /** 开头的单行或多行注释。在连续的行上使用 /// 与多行文档注释有同意的效果。

在文档注释里，Dart 编译器会忽略所有不在括号中的文本。使用括号，你可以引用到类、方法、字段、顶级变量、函数和参数。括号中的名字会在文档程序元素所在的词法作用域内被解析。

下面是一个引用了其他类和参数的文档注释：

```
/// A domesticated South American camelid (Lama glama).
///
/// Andean cultures have used llamas as meat and pack
/// animals since pre-Hispanic times.
class Llama {
  String name;

  /// Feeds your llama [Food].
  ///
  /// The typical llama eats one bale of hay per week.
  void feed(Food food) {
    // ...
  }

  /// Exercises your llama with an [activity] for
  /// [timeLimit] minutes.
  void exercise(Activity activity, int timeLimit) {
    // ...
  }
}
```

在生成的文档中，**[Food]** 会变成指向 Food 类文档的链接。

要解析 Dart 代码并且生成 HTML 文档，你可以使用 SDK 中的 文档生成工具。要查找一个生成的文档的例子，请参阅 Dart API 文档。要获取关于如何组织注释的建议，请参阅 Dart 文档注释指南。

总结

该页面汇总了 Dart 语言常用的特性。更多的特性正在被实现，但我们希望它们不会破坏已有的代码。要了解更多信息，请参阅 Dart 语言规范 和 高效的 Dart。

要了解更多关于 Dart 核心库的信息，请参阅 Dart 库教程。

译者总结

该页面翻译了官方的"Dart 语言简明教程"，其中多次提到 Dart 库教程、Dart 语言规范 和 高效的 Dart，译者有意在之后翻译"Dart 库教程" 和 "高效的 Dart"，其中 Dart 库教程 已翻译完成。请关注该代码仓库。