# Dart语言概览

极客学院出版

# 前言

Dart 是谷歌在 2011 年推出的编程语言，是一种结构化 Web 编程语言，允许用户通过 Chromium 中所整合的虚拟机（Dart VM）直接运行 Dart 语言编写的程序，免去了单独编译的步骤。以后这些程序将从 Dart VM 更快的性能与较低的启动延迟中受益。Dart 从设计之初就为配合现代 web 整体运作而考虑，开发团队也同时在持续改进 Dart 向 JavaScript 转换的快速编译器。Dart VM 以及现代 JavaScript 引擎（V8 等）都是 Dart 语言的首选目标平台。

本教程是 Dart 官方文档 A Tour of the Dart Language (https://www.dartlang.org/docs/dart-up-and-running/ch02.html) 的中文翻译版本。

## 适用人群

这个教程设计为了让对 Dart 感兴趣的软件专业人士更加完整的了解 Dart 的基本语法结构。完成这个教程，你将充分了解 Dart 让自己获得更高的水平的专业知识。

## 学习前提

在你继续本教程之前，你必须对简单的术语有一定的了解，比如源码，文档等等。因为在你的组织下处理各级软件项目，如果你有软件工作的知识在软件开发和软件测试流程那是最好的。同时，你需要能够使用 Dart 提供的 IDE 工具进行 Dart 程序的开发。

# 目录

# 1

# 一个基础的 Dart 程序

下面的代码运用了 Dart 的许多基础功能：

```
//定义一个函数
printNumber(num aNumber) {
  print('The number is $aNumber.'); // 输出至控制台.
}

// 此处是APP开始执行的地方
main() {
  var number = 42; // 声明并初始化变量
  printNumber(number); // 调用函数
}
```

下面是这个程序的使用，适用于所有（或者说几乎所有）的Dart apps;

```
// 这是注释
```

使用 `//` 来表示剩下的行是一个注释，也可以使用 `/*...*/` 来注释，细节请见 注释（Comments）()；

num

一种变量类型。其他一些内置类型有 String, int 和 bool;

42

一个数字，数字是一种编译常量。

print()

一个手动的方式来展示输出。

'...'(或者"...")

一个字符串。

$variableName (或者 ${expression})

字符串插值：包括一个变量或表达式的字符串，相当于一个字符串内。欲了解更多信息，请参见 字符串 (built-in-types.md#strings) 。

main()

是一个特殊的、必要、顶级的函数，应用程序开始执行的地方。欲了解更多信息，请参阅 main()函数 (functions.md#main) 。

var 一种特殊的方式来声明变量，不用特指变量类型。

> 注意：我们的代码参照 Dart 风格指南（Dart Style Guide (https://www.dartlang.org/articles/style-guide/)，例如我们使用了两个空格缩进。

# 2

# 重要的概念

当学习Dart语言的时候，把这些事实和概念记在脑子里：

- 每个变量都是一个对象，每个对象是一个类的实例。甚至数字，函数，和null都是对象。所有对象都继承自Object (http://api.dartlang.org/dart_core/Object.html) 类
- 指定静态类型（如num前面的例子中）讲清意图，用 tools 开启静态检查，但它是可选的。（可能注意到当你调试代码，没有指定类型的变量会得到一个特殊的类型： dynamic ）
- Dart解析所有的代码运行之前。可以对Dart提供提示，例如，通过使用类型或编译时间常数来捕获错误或帮助代码运行更快。
- Dart支持顶级函数（如main()）也支持类或者对象（静态和实例方法分别支持）里的函数。还可以在函数里创建函数（嵌套或局部功能）。
- 类似的，Dart支持顶级变量，以及依赖于类或对象（静态变量和实例变量）变量。实例变量有时被称为域或属性。
- 与Java不同，Dart不具备关键字public，protected和private。如果一个标识符以下划线（_）开始，那么它和它的库都是私有的。有关详细信息，请参阅 Libraries and visibility (https://www.dartlang.org/docs/dart-up-and-running/ch02.html#libraries-and-visibility) 。
- 标识符可以字母或（_）开始，或者是字符加数字的组合开头。
- 有时，判断是一个表达式还是一个语句会很重要，所以我们要准确了解这两个单词。
- Dart tools可报告两类问题：警告(warning)和错误(error)。警告只是迹象表明，代码可能无法正常工作，但他们不会阻止程序的执行。错误可以是编译时或运行时,编译时的错误阻止代码执行;当代码执行时一个运行时的错误会导致一个 异常（exception）() 被抛出。
- Dart有两种运行模式：生产 (production) 和检查 (checked) 。我们建议在检查模式开发和调试，并将其部署到生产模式。
- Production mode是Dart程序一个速度优化的默认运行模式。Production mode忽略 断言语句（assert statements）(control-flow-statements.md#assert) 和静态类型。
- Checked mode 是开发人员友好的方式，可以帮助你在运行时捕捉一些类型的错误。例如，如果分配一个非数字来声明为一个 num 变量，然后在检查模式会抛出异常。

# 3

# 关键字

下面表格列出了 Dart 语言的关键字：

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

- 上标1的单词是内置的标识符(built-in identifiers)。避免使用表格内的标识作为符标识，而且从来不使用它们作为类(class)或类型(type)的名称。内置标识符存在，以方便从 JavaScript 到 Dart 的移植。例如，如果一些JavaScript代码中有一个名为工厂的变量，当你将代码移植到 Datr 中，你不必重新命名它。
- 上标2的单词是的Dart1.0版本之后添加异步支持较新的、有限的保留字。不能使用async，await，或yield作为在标有async，或sync的任何函数体的标识符。欲了解更多信息，请参见 异步性支持(Asynchrony support) () 。
- 在关键字表中的所有单词都是保留字。不能使用保留字作为标识符。

# 4

# 变量

下面是创建变量并对其赋值的一个例子：

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

变量都是引用，变量name包含对一个 String 对象值是 “Bob” 的引用。

## 默认值

未初始化的变量具有 null 的初始值。即使数字类型变量最初为 null ，因为数字是对象。

```
int lineCount;
assert(lineCount == null);
// 变量 (尽管会赋值数值)被初始化为 null.
```

> 注：assert() 调用在生产模式(production mode)下是被忽略的。在检查模式下，aeesrt(condition) 抛出一个异常，除非条件为真。有关详细信息，请参阅 Assert 部分。

## 可选类型

你必须在你的变量声明中添加静态类型选项：

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

添加类型是一种清晰地表达你意图的方式。工具如编译器和编辑器可以使用这些类型来帮助你，通过为错误和代码完成提供漏洞预警和代码补全。

> 注：本章参照风格指南推荐的使用var,而不是为局部变量做类型标注。

## final 和 const

如果从不打算改变一个变量，使用 final 或者 const 代替 var 或者其他类型。一个 final 变量只能被设置一次；一个 const 变量是一个编译时常数。

被声明为 final 的顶层或类变量第一次使用时被初始化：

```
final name = 'Bob'; // Or: final String name = 'Bob';
// name = 'Alice';  // 取消注释会产生一个错误
```

> 注：延迟初始化变量最终有助于应用程序启动更快。

使用常量作为要编译的常数变量。如果 const 的变量是在类级别，将其标记为静态常量。（实例变量不能是const。）如果你声明的变量，设置该值为编译时常数设置，如文字，一个 const 变量，或常数算术运算的结果：

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

# 5

# 内置类型

Dart语言对以下类型的特殊支持：

- 数字 number
- 字符串 strings
- 布尔 booleans
- 列表 lists（也称为数组arrays）
- 图 maps
- 符号 symbols

可以初始化的任何使用文字这些特殊类型的一个对象。例如，*'this is a string'*是字符串常量，而*ture*的是一个布尔值。

因为在 Dart 中，每一个变量是指一个对象，即一个类的实例，通常可以使用构造函数初始化变量的对象实例。一些内置的类型都有自己的构造函数。例如，你可以使用 Map() 构造函数来创建一个映射，使用代码，如 new Map()。

## Numbers

Dart 数字有两种形式：

- int 整数值，通常应在范围-2(53次方)到2(53次方)
- double 64位（双精度）浮点数，作为指定由IEEE 754制定标准；

无论int和double是num的子类型。该数字类型包括基本运算符，如+， －，/和*，以及位运算符，例如 >>。 num的类型也是在那里你会发现 abs()，ceil()，和 floor()，以及其他的方法。如果num及其亚型没有你要找的内容，Math库可能有。

> 注意：-253到253范围之外的整数，目前在从 Dart 代码生成 JavaScript 中比同样的 Dart 代码在 Dart 虚拟机运行效果不同。其原因是，Dart 中数字可以被指定到具有任意精度的整数，但 JavaScript不可以。详情请参阅issue 1533 (http://dartbug.com/1533) 。

整数是没有小数点的数字。以下是定义整数常量的一些例子：

```
var x = 1;
var hex = 0xDEADBEEF;
var bigInt = 346534658346524376592384765923749587398457 29;
```

如果一个号码包括一个小数，这是一个Double类型的。以下是定义Double文字的一些例子：

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

如何把字符串转换成一个数字，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 类型规定了传统按位移位（<<，>>），与（&）和或（|）运算符。例如：

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

()

## 字符串 Strings

Dart字符串是UTF-16编码单元的序列。可以使用单或双引号来创建一个字符串：

```
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.";
```

你可以通过使用 ${expression} 把一个表达式的值放进字符串。如果表达式是一个标识符，你可以跳过{}。为了获得相应对象的字符串，Dart 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!');
```

注：运算符==测试两个对象是否是等价的。两个字符串是等价的，如果它们包含的代码单元相同的序列。

可以利用相邻字符串或+运算符连接字符串：

```
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.');
```

另一种方法来创建一个多行字符串：使用三引号用单或双引号：

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

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

可以通过带有 r 的前缀来创建一个“raw”的字符串：

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

可以使用 Unicode 转义代替字符串：

```
// Unicode 转义工作：[heart]
// 转义: [heart]
print('Unicode escapes work: \u2665');
```

有关使用字符串的更多信息，请参见 字符串和正则表达式 (https://www.dartlang.org/docs/dart-up-and-running/ch03.html#strings-and-regular-expressions) 。

()

## 布尔值 booleans

为了表示布尔值，Dart 有一个名为bool的类型。只有两个对象具有 bool 类型：布尔文字，true 和 false。

当Dart需要一个布尔值，仅值 true 被视为真。所有其他值被视为假。不像在 JavaScript 中，值，如1，“aString”，someObject 都视为假的。

例如，请考虑下面的代码，这是有效 JavaScript 代码也是有效的Dart代码：

```
var name = 'Bob';
if (name) {
  // JavaScript中会产生打印，而Dart中不会
  print('You have a name!');
}
```

如果将上面代码作为JavaScript代码运行，它将打印 “You have a name!” ，因为名字是一个非空的对象。然而，在Dart在生产模式下运行时，上面的代码不会打印，因为在所有的名字被转换为假（因为 name！= true ）。在 DART 运行检查模式，前面的代码抛出一个异常，因为该名称变量不是一个 bool。

下面的代码行为不同的 JavaScript 和 Dart 另一个例子：

```
if (1) {
  print('JS prints this line.');
} else {
  print('Dart in production mode prints this line.');
  // 但在checked模式下会抛出一个异常，因为1不是一个boolean
}
```

注意：只有在生产模式，而不是检查模式，前面的两个示例工作。在检查模式下，一个例外是，当需要一个boolean值，而此时一个非布尔时使用，异常会被抛出。 Dart针对布尔值的设计是为了避免可能出现的时候许多值可以被视为真而产生怪异的行为。这意味着，不能使用代码像if（nonbooleanValue），应该明确检查值。例如：

```
// 检查一个空字符串.
var fullName = '';
assert(fullName.isEmpty);

// 检查为零.
var hitPoints = 0;
assert(hitPoints <= 0);

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

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

## List 列表

几乎每一个编程语言中最常见的集合就是数组或有序组对象了。在Dart，数组是列表对象，所以我们通常只是将其称为lists。

Dart列表书面上看起来像JavaScript数组常量。 这里有一个简单的Dart列表：

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

列表使用从零开始的索引，其中0为第一个元素,list.length-1是最后一个元素的索引。你可以得到一个列表的长度，就像在JavaScript中一样，参阅列表元素：

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

列表类型有很多方法，方便操纵名单。有关列表的详细信息，请参见 泛型(Generics) () 和 集合(Collections) (https://www.dartlang.org/docs/dart-up-and-running/ch03.html#collections) 。

## Maps

在一般情况下，一个 map 是一个对象，相关联的键和值。这两个键和值可以是任何类型的对象。每个键只发生一次，但是可以使用相同的值多次。对于 mapDart 提供 map 文字和 Map 类型支持。

这里有几个简单的Dart图的例子，图用文字创建：

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

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

可以使用一个Map构造相同的对象：

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

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

添加新键值对到现有的map就像你在JavaScript中：

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds'; // 增加键值对
```

检索map的值(以在 JavaScript 中相同的方式)：

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

如果你寻找的关键词不在图里，则返回空（null）：

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

使用 .length，以获得map中键值对的数目：

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

有关映射的更多信息，请参见泛型和图。

## 符号 symbols

一个符号对象表示在Dart程序中声明的操作符或标识。你可能不会需要使用这些符号，但他们对于由名字指向的API是很有用的，因为时常要改变的是标识符的名字，而不是标识符的符号。

为了得到符号的标识，使用符号的文本，只是 # 后跟标识符：

```
#radix
#bar
```

有关符号的详细信息，请参阅 Dart：镜子 - 反射(dart:mirrors-reflection) (https://www.dartlang.org/docs/dart-up-and-running/ch03.html#dartmirrors---reflection) 。

# 6

# 函数

下面是一份关于如何使用函数的示例：

```
void  printNumber(num number) {
    print('The number is $number.');
}
```

尽管上面这个指导的格式声明了参数和返回值的类型，实际上你可以不这么做：

```
printNumber(num number) { // 没有声明类型也是可以的
    print('The number is $number.');
}
```

对于仅含有一个表达式的方法，你可以使用一种简写的语法：

```
void printNumber(num number) =>
    print('The number is $number.');
```

`=> 表达式;` 语法是 `{ return 表达式 }` 的简写。在 `printNumber()` 方法中，这个表达式调用了顶级函数 `print()` 。

> 注意：只有一个表达式能够在箭头符（ => ）和分号（ ; ）之间出现，语句是不可以这样使用的。比如，你不能把 if 语句放在这两个符号之间，但是一个三元运算符（ ?: ）是可以的。

调用函数的方法如下：

```
printNumber(123)
```

一个函数可以有两种类型的参数：必要参数和可选参数。所有的必要参数都应放在可选参数之前，当必要参数已经全部列出时，才能在后面加入可选参数。

## 可选参数

可选参数可以是可选位置参数或者可选命名参数，但不能既是可选位置参数又是可选命名参数。

这两种可选参数都可以定义默认值。但是默认值必须是编译时的常量，比如字面值。如果没有为之提供默认值，那么该参数的默认值将会是 null。

### 可选命名参数

在调用函数的时候，你可以通过 `paraName: Value` 的形式使用有命名的参数。比如：

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

当定义一个函数的时候，使用 `{param1, param2, ...}` 来声明有命名的参数：

```
/// 将 bold 和 hidden 作为你声明的参数的值
enableFlages({bool bold, bool hidden}) {
    // ...
}
```

使用冒号来设置默认值：

```
/// 把 bold 和 hidden 作为参数的值，并将默认值设为 false
enableFlags({bool bold: false, bool hidden: false}) {
    // ...
}

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

**可选位置参数**

把一组函数的参数放在 [] 之内可以把它们标记为可选位置参数：

```
Sring 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述方法的一个实例：

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

接下来则是使用可选参数调用的实例：

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

可选位置参数使用 = 来声明默认值：

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

```
}

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

()

## main 函数

每一个应用都必须有一个顶级函数 main() ，这个函数就是整个应用的入口函数。 main() 函数返回值类型为 void 并且有一个可选参数 List<String> 。

下面是一份关于网络应用的 main() 函数示例：

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

> 提示：代码前的 .. 操作符是级联操作符，用于允许用户在一个对象上实现多个操作。更多关于级联操作符的内容请见 类 () 。

下面是一个命令行应用的带参数的 main() 函数示例：

```
// 这个应用应该在命令行中这样启动：
// 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');
}
```

你可以在 args library (https://pub.dartlang.org/packages/args) 上了解如何定义和分析命令行参数。

## 作为对象的函数

你可以把一个函数当做参数传给另一个函数。比如：

```
printElement(element) {
    print(element);
```

```
}

var list = [1, 2, 3];

// 将 printElement 作为参数传递给其他函数
list.forEach(printElement);
```

你也可以把一个函数定义为一个变量，就像下面这样：

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

## 语法作用域

Dart 是语法作用域型的语言，也就是说变量的生命周期是genuine代码的布局静态决定的。你可以“在大括号之外追溯”来看一下变量是否存在于这个区域中。

下面是一份关于嵌套的带参函数在不同的作用域等级上的示例：

```
var topLevel = true;

main() {
    var indideMain = true;

    myFunction() {
        var insideFunction = true;

        nestedFunction() {
            var insideNestedFunction = true;

            assert(topLevel);
            assert(insideMain);
            assert(insideFunction);
            assert(insideNestedFunction);
        }
    }
}
```

请留意 `nestedFunction()` 是怎样使用不同作用域上的变量的，最好从最内层开始一直分析到最外层。

## 语法的封闭性

封闭指的是一个函数可以访问其语法作用域内的变量，即使这个函数是在变量本身的作用域之外被调用的。

函数内部会包含在临近作用域内所定义的变量。在下一个示例中， `makeAdder()` 捕获了变量 `addBy` 。不管返回的函数在哪里被调用，它都可以使用 `addBy` 。

```
/// 返回一个把 addBy 作为参数的函数
Function makeAdder(num addBy) {
    return (num i) => addBy + 1;
}

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);
}
```

## 函数的等价性测试

下面是关于顶级函数、静态方法和实例方法的等价性测试：

```
foo() {} // 一个顶级函数

class SomeClass {
    static void bar() {} // 一个静态方法
    void baz() {} // 一个示例方法
}

main() {
    var x;

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

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

    // 这些封闭引用了相同的示例对象（A的实例2）
```

```
    // 所以它们是等价的
    assert(y.baz == x);

    // 这些封闭引用的十不同实例，所以它们不等价
    aessert(v.baz != w.baz);
}
```

## 返回值

所有的函数都有返回值。如果没有定义返回值，那么语句 `return null` 将会隐式地加到函数体的最后。

# 7

# 操作符

Dart 定义了下表所示的操作符。你可以重写其中的一些操作符，具体请见 重载操作符 (classes.md#overridable-operators) 。

| 介绍 | 符号 |
| --- | --- |
| 一元后缀符 | expr++ expr-- () [] . |
| 一元前缀符 | -expr !expr ~expr ++expr --expr |
| 乘法类型 | * / % ~/ |
| 加法类型 | + - |
| 位操作符 | << >> |
| 按位与 | & |
| 按位异或 | ^ |
| 按为或 | \| |
| 比较和类型测试 | >= <= > < as is is! |
| 等价 | == != |
| 逻辑与 | && |
| 逻辑或 | \|\| |
| 条件运算符 | expr1 ? expr2 : expr3 |
| 级联运算符 | .. |
| 赋值 | = *= /= /= ~/= %= += -= <<= >>= &= ^= \|= |

当你是用这些运算符的时候，你就创建了表达式。这里有一些表达式的例子：

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

在之前的操作符表中，操作符的优先级由其所在行定义，上面行内的操作符优先级大于下面行内的操作符。例如，乘法类型操作符 `%` 的优先级比等价操作符 `==` 要高，而 `==` 操作符的优先级又比逻辑与操作符 `&&` 要高。这些操作符的优先级顺序将在下面的两行代码中体现出来：

```
// 1.使用括号来提高可读性
if ((n % i == 0) && (d % i == 0))

// 2.难以阅读，但是和上面等价
if (n % i == 0 && d % i == 0)
```

> 警告：对于二元运算符，其左边的操作数将会决定使用的操作符的种类。例如，当你使用一个 Vector 对象以及一个 Point 对象时， aVector + aPoint 使用的 + 是由Vector 所定义的。

## 算术运算符

Dart 支持一些常规的算术运算符，具体如下表所示：

| 操作符 | 含义 |
| --- | --- |
| + | 加 |
| – | 减 |
| –expr | 一元减号，也被命名为负号（使后面表达式的值反过来） |
| * | 乘 |
| / | 除 |
| ~/ | 返回一个整数值的除法 |
| % | 取余，除法剩下的余数 |

具体示例：

```
assert(2 + 3 == 5);
assert(2 – 3 == –1);
assert(2 * 3 == 6);
assert(5 / 2 == 2.5); // 结果是double类型
assert(5 ~/ 2 == 2); // 结果是一个整数
assert(5 % 2 == 1); // 余数

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

Dart 同时也支持前缀和后缀自增以及自减运算符。

| 操作符 | 含义 |
| --- | --- |
| ++var | var = var + 1（表达式中相当于var + 1） |
| var++ | var = var + 1（表达式中相当于var） |
| ––var | var = var – 1（表达式中相当于var – 1） |
| var–– | var = var – 1（表达式中相当于var） |

示例

```
var a, b;

a = 0;
b = ++a; // 在b获得其值前先自增a
assert(a == b); // 1 == 1

a = 0;
b = a++; // 在b获得其值后自增a
```

```
assert(a != b); // 1 != 0

a = 0;
b = --a; // 在b获得其值前自减a
assert(a == b); // -1 == -1

a = 0;
b = a--; // 在b获得其值后自减a
assert(a != b); // -1 != 0
```

## 等价和关系操作符

下表中包括的是等价以及关系操作符：

| 操作符 | 含义 |
| --- | --- |
| == | 等价；后面讨论中会有详细说明 |
| != | 不等价 |
| > | 大于 |
| < | 小于 |
| >= | 大于等于 |
| <= | 小于等于 |

如果想测试两个对象 x 和 y 是不是同一个对象，使用 `==` 运算符。（在少数情况下如果你想比较两个对象是否相等，需要使用 identical (https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core#id_identical) 来替代 `==` 。)接下来将会说明 `==` 操作符是怎样起作用的：

- 1. 如果 x 或者 y 为 null，两者都为 null 则返回 true， 只有其中一个为 null 则返回 false。
- 1. 返回一个函数调用的结果： `x.==(y)` 。（这个调用是正确的，像 `==` 这样的运算符实际上是由第一个操作数所调用的一个方法。你可以重写大部分运算符，关于这部分的内容你将在 重载操作符 (classes.md#overridable-operators) 中看到。

以下是关于等价和关系运算符的用法的示例：

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

## 类型测试操作符

as 、 is 和 is! 操作符在运行时用于检查类型非常便捷。

| 运算符 | 含义 |
| --- | --- |
| as | 类型转换 |
| is | 当对象是相应类型时返回 true |
| as | 当对象不是相应类型时返回 true |

如果 obj 实现了 T 所定义的借口，那么 obj is T 将返回 true。比如， obj is Object 必然返回 true。

使用 as 操作符可以把一个对象转换为特定类型。一般来说，如果在 is 测试之后还有一些关于对象的表达式，你可以把 as``当做是is 测试的一种简写。考虑下面这段代码：

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

你也可以通过 as 来简化代码：

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

> 注意：上面两段代码并不相等。如果 emp 的值为 null 或者不是 Person 的一个对象，第一段代码不会做任何事情，第二段代码将会报错 。

## 赋值运算符

就如你已经知道的，你可以使用 = 操作符来给变量赋值。同时，你也可以使用复合的赋值操作符，比如 += 。

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

接下来将会介绍复合的赋值运算符如何实现的：

| 说明 | 复合赋值 | 等价表达式 |
| --- | --- | --- |
| 对于操作符op | a op b | a = a op b |
| 具体例子 | a += b | a = a + b |

下面的代码同时使用了赋值和复合赋值操作符：

```
var a = 2;          // 赋值运算符 =
a *= 3;             // 赋值并相乘： a = a * 3
assert(a == 6);
```

## 逻辑运算符

你可以使用逻辑运算符来转置或者结合布尔值的表达式。

| 运算符 | 含义 |
| --- | --- |
| !expr | 将后面的表达式转置（false 转 true） |
| \|\| | 逻辑或 |
| && | 逻辑与 |

下面是逻辑运算符的用法：

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

## 位操作与移位运算符

在 Dart 中，你可以直接操作数字的每一个字位。一般来说，你会使用位操作与移位运算符来对整数进行操作。

| 运算符 | 含义 |
| --- | --- |
| & | 与 |
| \| | 或 |
| ^ | 异或 |
| ~expr | 二元按位取补（1变成0，0变成1） |
| << | 左移 |
| >> | 右移 |

下面是关于位操作与移位运算符的说明：

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

assert((value & bitmask)  == 0x02);  // 与
assert((value & ~bitmask) == 0x20);  // 与非
assert((value | bitmask)  == 0x2f);  // 或
assert((value ^ bitmask)  == 0x2d);  // 异或
assert((value << 4)       == 0x220); // 左移
assert((value >> 4)       == 0x02);  // 右移
```

()

## 其他运算符

还有一些运算符，这些运算符的说明你已经在之前看过了

| 操作符 | 名称 | 含义 |
| --- | --- | --- |
| （ ） | 函数应用 | 表示函数调用 |
| [] | 目录 | 列表中制定索引处的值 |
| expr1 ? expr2 : expr3 | 条件运算符 | expr1 为真则返回 expr2，否则返回 expr3 |
| . | 成员访问 | 表达式的属性，例如：`foo.bar` 从 `foo` 中选择了 `bar` 属性 |
| .. | 级联 | 允许你在单个对象的成员上执行多个操作，具体可见 类 () |

# 8

# 控制流语句

你可以使用以下任何一种方式来控制你的Dart代码流：

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

你也可以通过使用try-catch和throw来改变控制流，具体说明请见 异常 () 部分。

## if 和 else

Dart支持If语句以及可选的else语句，如样例所示，同样可见包含了 其他运算符 (operators.md#other-operators) （请见上部分）的条件表达式(?:)

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

记住，它不同于JavaScript的地方是,Dart将所有不为True的值视作False，如想了解更多相关信息,请见 Booleans (built-in-types.md#booleans) 部分。

## for 循环

你可以通过For循环实现重复声明，如下例：

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

Dart的for循环中的闭包能获取循环变量的值，避免了像Javascript中的常见陷阱，如下例，请思考下列代码：

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

正如我们所期待的，输出结果分别是0和1，但相反的情况可以在Javascript中见到，输出分别为2和2。

如果正在重申的变量是可重申的，你可以使用 forEach()方法 (https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core.Iterable#id_forEach) 。如果你不需要知道重申计数器的值,使用forEach()是一个很好的选择：

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

可重申的类例如Lish以及Set也支持for-in形式的 iteration (https://www.dartlang.org/docs/dart-up-and-running/ch03.html#iteration) ：

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

## while 和 do-while

while循环在开始之前就对条件进行求值运算：

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

Do-while循环在循环开始之后才对条件进行求值运算：

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

## break 和 continue

使用break来结束循环：

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

使用continue来跳过本次循环，进入下次循环：

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

你或许可以写出一个不同的例子如果你正在使用例如List或者Set的 Iterable (https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core.Iterable) :

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

()

## switch 和 case

Dart中Switch语句通过使用 == 来比较整型，字符串或者编译时间常量。被比较的对象必须都是同一个类（不能是它的任何一个子类型）中的实例，并且这个类不允许覆盖 ==.枚举类型 (classes.md#enums) 很适用于在Switch语句。

> 提示：Dart中的Switch语句是为限制条件而设计的，比如翻译器或者扫描器。

作为规定，每一个不为空的case分句都以break语句结尾.其他正确的结尾方式也可以使用continue,throw或者return语句来实现。 当没有任何case分句的条件符合时，使用default分句来执行代码：

```
var command = ‘OPEN’;
switch (command){
case ‘CLOSE’:
executeClosed();
break;
case ‘PENDING’:
executePending();
break;
case ‘APPROVED’:
executeApproved():
break;
case ‘DENIED’:
executeDenied();
break;
case ‘OPEN’
executeOpen();
break;
default:
executeUnknown();
}
```

下列的例子漏写了case分句中的break语句，从而产生了一个错误：

```
var command = ‘OPEN’
switch(command){
case ‘OPEN’:
executeOpen();
//错误;缺少了 break 造成了一个异常!!

case ‘CLOSED’:
executeClosed();
break;
}
```

然而,Dart支持空的case分句，并允许使用从一个case到另一个case的贯穿形式：

```
var command = ‘CLOSED’;
switch (command){
case ‘CLOSED’://空的内容造成了贯穿.
case ‘NOW_CLOSED’:
// CLOSED 和 NOW_CLOSED两个都会运行.
executeNowClosed();
break;
}
```

如果你真的想用这种贯穿方式，你可以使用continue语句以及一个标签：

```
war command = ‘CLOSED’;
switch(command){
case ‘CLOSED’:
executeClosed();
continue nowClosed;
//继续在 nowClosed 标记的地方执行.

nowClosed:
case ‘NOW_CLOSED’:
//CLOSED 和 NOW_CLOSED两个都会运行.
executeNowClosed();
break;
}
```

一个case分句可以含有局部变量，该局部变量仅仅只在此分句范围内可见。

()

## Assert

如果一个布尔条件值为false，使用assert语句来中断正常执行的代码。你可以在本教程中找到一些assert语句的样例。这里有一些例子：

```
 //确保这个变量不为空值.
assert(text != null);

//确保这个变量小于100.
assert(number < 100);

//确保它是一个https协议类型的URL.
assert(urlString.startsWith(‘https’));
```

提示：Assert语句仅仅只能在调试模式下使用，在生产模式下没有任何作用。

在assert语句后面的括号中，你可以加入任何表示布尔值或者函数的表达式。如果表达式的值或者函数返回值true，则assert语句成功并继续执行代码。如果值为false，则assert语句失败并抛出一个异常 (an AssertionError (https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core.AssertionError) )。

# 9

# 异常

你的Dart代码可以抛出异常和捕获异常。异常就是出现预期之外的结果的错误。如果没有捕获异常, isolate 将会使异常挂起，往往会导致 isolate 和程序终止。

与java相反，Dart中所有异常都是不需检测的异常。方法并不会声明它将抛出哪些异常，而且你不需要去捕捉任何异常。

Dart 除了提供 异常 (https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core.Exception) 、错误 (https://api.dartlang.org/apidocs/channels/be/dartdoc-viewer/dart:core.Error) 类型以外还提供了众多预定义的子类型。当然，你可以定义你自己的异常类型。毕竟，Dart程序可以将任何非空对象作为异常抛出，不只局限与异常和错误对象。

## throw 抛出异常

这里抛出了一个异常：

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

你也可以将任意对象作为异常抛出：

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

由于抛出异常是单个表达式，所以你可以将 in=> 语句作为异常抛出，也可以在其他任何地方抛出异常。

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

## catch 捕获异常

捕获了一个异常后，就停止了捕获异常过程。捕获一个异常，你就有机会去处理它：

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

为了处理含有多种类型异常的代码，你可以选择多个catch子句。第一个匹配抛出对象类型的catch子句将会处理这个异常。如果catch子句未说明所捕获的异常类型，这个子句就可处理任何被抛出的对象。

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

```
  // 一个具体异常
  buyMoreLlamas();
} on Exception catch (e) {
  // 任意一个异常
  print('Unknown exception: $e');
} catch (e) {
  // 非具体类型
  print('Something really unknown: $e');
}
```

像上面展示的代码一样，你可以用 `on` 或者 `catch` ，或者两者都用。当你需要指定异常类型的时候用 `on` ，当你的异常处理者需要异常对象时用 `catch` 。

## finally

为了确保不论是否抛出异常，代码都正常运行，请使用 `finally` 子句。如果没有 `catch` 匹配子句的异常， `finally` 子句运行以后异常将被传播：

```
try {
  breedMoreLlamas();
} finally {
  // 即使抛出一个异常时也会进行清理
  cleanLlamaStalls();
}
```

在匹配了所有 `catch` 之后，子句 `finally` 运行了。

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

通过阅读 Exceptions (https://www.dartlang.org/docs/dart-up-and-running/ch03.html#exceptions) 部分可了解更多。

# 10

# 类

Dart 是一种面向对象语言，包含类和基于 mixin 的继承两部分。每个对象是一个类的实例，并且 Object (https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core.Object) 是所有类的父类。基于 mixin 的继承指的是每个类（除了 Object ）都只有一个父类，类体还可以在多个类继承中被重用。

要创建一个对象，你可以使用 `new` 关键词并在其后跟上一个构造函数。构造函数可以写成 `类名` ，或者 `类名.标识符` 形式。例如：

```
var jsonData = JSON.decode('{"x":1, "y":2}');

//用 Point() 创建一个点。
var p1 = new Point(2, 2);

// 用 Point().fromJson() 创建一个点。
var p2 = new Point.fromJson(jsonData);
```

对象的成员分为函数和数据两类（各自的方法和实例变量）。当你调用一个方法时，你是通过一个对象来调用它的：该方法可访问该对象的方法和数据。用 . 指向对象的方法和数据成员。

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

// 给 y赋值。
p.y = 3;

// 获取 y 的值。
assert(p.y == 3);

// 用 p 对象调用 distanceTo() 。
num distance = p.distanceTo(new Point(4, 4));
```

当你想对一个对象的成员进行一系列操作时，用串联操作（ cascade ）。

```
querySelector('#button') // 获取一个对象。
  ..text = 'Confirm'   // 调用他的成员。
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'));
```

一些类提供常量构造函数，要创建一个编译时用的常量构造函数，使用 `const` 关键字代替 `new` ：

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

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

assert(identical(a, b)); // 他们是相同的实例!
```

下面的部分来讨论如何实现类。

## 实例变量

这里是你如何声明实例变量：

```
class Point {
  num x; // 声明实例变量 x，默认值为 null 。
  num y; // 声明实例变量 y，默认值为 null 。
  num z = 0; // 声明实例变量 z，初始化为 0 。
}
```

所有未初始化的实例变量的值为 `null` 。

所有的实例变量会自动生成一个隐式的 getter 方法。 Non-final 实例变量也会自动生成一个隐式的 setter 方法。有关详细信息，参见 getter and setter (classes.md#getters-and-setters) 。

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

main() {
  var point = new Point();
  point.x = 4;        // 用 setter 方法得到 x 。
  assert(point.x == 4); // 用 getter 方法得到 x 。
  assert(point.y == null); // 值为 null 。
}
```

如果你在声明实例变量时进行了初始化（而不是在构造函数或方法），在实例创建时该值就确定了，它是在构造函数初始化列表中执行的。

## 构造函数

要声明一个构造函数，只需创建一个与类同名的方法（或者加上一个额外的标识符命名构造函数的描述）。构造函数最常见的形式，就是自动生成的构造函数，下面创建一个类的新实例：

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

  Point(num x, num y) {
```

```
    // 有个更好的方法来实现。
    this.x = x;
    this.y = y;
  }
}
```

this 关键字是指当前实例。

**注意**：只有当名字冲突时才能使用 this。否则，Dart 会忽略 this 。

为一个实例变量分配一个构造函数参数的模式是很常见的，Dart有语法糖使它使用起来更容易：

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

  // 用语法糖来设置 x ， y 。
  // 在构造函数运行之前。
  Point(this.x, this.y);
}
```

## 默认构造函数

如果你不声明一个构造函数，系统会提供默认构造函数。默认构造函数没有参数，它将调用父类的无参数构造函数。

## 构造函数不能继承

子类不继承父类的构造函数。子类只有默认构造函数。（ 无参数，没有名字的构造函数 ）。

## 命名构造函数

使用命名构造函数可以为一个类声明多个构造函数，或者说是提供额外的声明：

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

  Point(this.x, this.y);

  // 命名构造函数
```

```
  Point.fromJson(Map json) {
    x = json['x'];
    y = json['y'];
  }
}
```

记住，构造函数不能继承，这意味着子类不会继承父类的构造函数。如果你希望子类在创建之后能够拥有在父类中声明的命名构造函数，你就必须在子类中实现该构造函数。

## 调用非默认的父类的构造函数

默认情况下，在子类的构造函数将会调用父类的无参数构造函数。如果父类没有构造函数，则必须手动调用父类的构造函数中的一个。在冒号（ : ）之后、构造函数之前指定父类的构造函数（如果有的话）。

```
class Person {
  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({});

  // Prints:
  // in Person
  // in Employee
}
```

在调用父类构造函数前会检测参数，这个参数可以是一个表达式，如函数调用：

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

**警告**：父类构造函数的参数不能访问 `this` 。例如，参数可调用静态方法但是不能调用实方法。

## 初始化列表

除了调用父类构造函数，你也可以在构造函数体运行之前初始化实例变量。用逗号隔开使其分别初始化。

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

  Point(this.x, this.y);

  // 初始化列表在构造函数运行前设置实例变量。

  Point.fromJson(Map jsonMap)
      : x = jsonMap['x'],
        y = jsonMap['y'] {
    print('In Point.fromJson(): ($x, $y)');
  }
}
```

**警告**：右手边的初始化程序无法访问 `this` 关键字。

## 重定向构造函数

有时一个构造函数的目的只是重定向到同一个类中的另一个构造函数。如果一个重定向的构造函数的主体为空，那么调用这个构造函数的时候，直接在冒号后面调用这个构造函数即可。

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

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

  // 主构造函数的代表。
  Point.alongXAxis(num x) : this(x, 0);
}
```

## 静态构造函数

如果你的类产生的对象永远不会改变，你可以让这些对象成为编译时常量。为此，需要定义一个 `const` 构造函数并确保所有的实例变量都是 `final` 的。

```
class ImmutablePoint {
  final num x;
  final num y;
  const ImmutablePoint(this.x, this.y);
  static final ImmutablePoint origin =
      const ImmutablePoint(0, 0);
}
```

()

## 工厂构造函数

当实现一个使用 `factory` 关键词修饰的构造函数时，这个构造函数不必创建类的新实例。例如，工厂构造函数可能从缓存返回实例，或者它可能返回子类型的实例。 下面的示例演示一个工厂构造函数从缓存返回的对象：

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

  // _cache 是一个私有库,幸好名字前有个 _ 。
  static final Map<String, Logger> _cache =
      <String, Logger>{};

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

  Logger._internal(this.name);

  void log(String msg) {
```

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

**注**：工厂构造函数不能用 this。

调用一个工厂构造函数，你需要使用 `new` 关键字：

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

## 方法

方法就是为对象提供行为的函数。

## 实例方法

对象的实例方法可以访问实例变量和 `this` 。以下示例中的 `distanceTo()` 方法是实例方法的一个例子：

```
import 'dart:math';

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

()

## setters 和 Getters

是一种提供对方法属性读和写的特殊方法。每个实例变量都有一个隐式的 getter 方法，合适的话可能还会有 setter 方法。你可以通过实现 getters 和 setters 来创建附加属性，也就是直接使用 `get` 和 `set` 关键词：

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

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

  // 定义两个计算属性: right and 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;
}

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

借助于 getter 和 setter ，你可以直接使用实例变量，并且在不改变客户代码的情况下把他们包装成方法。

**注**：不论是否显式地定义了一个 getter，类似增量（ ++ ）的操作符，都能以预期的方式工作。为了避免产生任何向着不期望的方向的影响，操作符一旦调用 getter ，就会把他的值存在临时变量里。

()

## 抽象方法

Instance ， getter 和 setter 方法可以是抽象的，也就是定义一个接口，但是把实现交给其他的类。要创建一个抽象方法，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，所以这里的方法不是抽象的...
```

```
  }
}
```

调用抽象方法会导致运行时错误。

详情见 抽象类 (classes.md#abstract-classes) 。

()

## 重载操作符

你可以重写在下表中列出的操作符。例如，如果你定义了一个向量类，你可以定义一个 `+` 方法来加两个向量。

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

以下是一个类中重写 `+` 和 `-` 操作符的例子：

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

  /// Overrides + (a + b).
  Vector operator +(Vector v) {
    return new Vector(x + v.x, y + v.y);
  }

  /// Overrides - (a - b).
  Vector operator -(Vector v) {
    return new Vector(x - v.x, y - v.y);
  }
}

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

  // v == (2, 3)
  assert(v.x == 2 && v.y == 3);
```

```
  // v + w == (4, 5)
  assert((v + w).x == 4 && (v + w).y == 5);

  // v − w == (0, 1)
  assert((v − w).x == 0 && (v − w).y == 1);
}
```

如果你重写了 `==` ，你也应该重写对象中 `hashCode` 的 getter 方法。对于重写 `==` 和 `hashCode` 例子，参见实现 Implementing map keys (https://www.dartlang.org/docs/dart-up-and-running/ch03.html#implementing-map-keys) 。

想要知道更多关于重载的信息，参见 扩展一个类 (classes.md#extending-a-class) 。

()

## 抽象类

使用 `abstract` 修饰符来定义一个抽象类，该类不能被实例化。抽象类在定义接口的时候非常有用，实际上抽象中也包含一些实现。如果你想让你的抽象类被实例化，请定义一个 工厂构造函数 (classes.md#factory-constructors) 。

抽象类通常包含 抽象方法 (classes.md#abstract-methods) 。下面是声明一个含有抽象方法的抽象类的例子：

```
// 这个类是抽象类，因此不能被实例化。
abstract class AbstractContainer {
  // ...定义构造函数，域，方法...

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

下面的类不是抽象类，因此它可以被实例化，即使定义了一个抽象方法：

```
class SpecializedContainer extends AbstractContainer {
  // ...定义更多构造函数，域，方法...

  void updateChildren() {
    // ...实现 updateChildren()...
  }

  // 抽象方法造成一个警告，但是不会阻止实例化。
  void doSomething();
}
```

## 隐式接口

每个类隐式的定义了一个接口，含有类的所有实例和它实现的所有接口。如果你想创建一个支持类 B 的 API 的类 A，但又不想继承类 B ，那么，类 A 应该实现类 B 的接口。

一个类实现一个或更多接口通过用 `implements` 子句声明，然后提供 API 接口要求。例如：

```
// 一个 person ，包含 greet() 的隐式接口。
class Person {
  // 在这个接口中，只有库中可见。
  final _name;

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

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

//  Person 接口的一个实现。
class Imposter implements Person {
  // 我们不得不定义它，但不用它。
  final _name = "";

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

greetBob(Person person) => person.greet('bob');

main() {
  print(greetBob(new Person('kathy')));
  print(greetBob(new Imposter()));
}
```

这里是具体说明一个类实现多个接口的例子：

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

()

## 扩展一个类

使用 `extends` 创建一个子类，同时 `supper` 将指向父类：

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

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

子类可以重载实例方法， getters 方法， setters 方法。下面是个关于重写 Object 类的方法 `noSuchMethod()` 的例子,当代码企图用不存在的方法或实例变量时，这个方法会被调用。

```
class A {
  // 如果你不重写 noSuchMethod 方法, 就用一个不存在的成员，会导致 NoSuchMethodError 错误。
  void noSuchMethod(Invocation mirror) {
    print('You tried to use a non-existent member:' +
        '${mirror.memberName}');
  }
}
```

你可以使用 `@override` 注释来表明你重写了一个成员。

```
class A {
  @override
  void noSuchMethod(Invocation mirror) {
    // ...
  }
}
```

如果你用 noSuchMethod() 实现每一个可能的 getter 方法，setter 方法和类的方法，那么你可以使用 @proxy 标注来避免警告。

```
@proxy
class A {
  void noSuchMethod(Invocation mirror) {
    // ...
  }
}
```

关于注释的更多信息，请参 元数据 () 。

()

## 枚举类型

枚举类型，通常被称为 enumerations 或 enums ，是一种用来代表一个固定数量的常量的特殊类。

## 使用枚举

声明一个枚举类型需要使用关键字 enum ：

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

在枚举中每个值都有一个 index getter 方法，它返回一个在枚举声明中从 0 开始的位置。例如，第一个值索引值为 0 ，第二个值索引值为 1 。

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

要得到枚举列表的所有值，可使用枚举的 values 常量。

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

你可以在 switch 语句 (control-flow-statements.md#switch-and-case) 中使用枚举。如果 e 在 switch (e) 是显式类型的枚举，那么如果你不处理所有的枚举值将会弹出警告：

```
enum Color {
  red,
  green,
  blue
}
// ...
Color aColor = Color.blue;
switch (aColor) {
  case Color.red:
    print('Red as roses!');
    break;
  case Color.green:
    print('Green as grass!');
    break;
  default: // Without this, you see a WARNING.
    print(aColor);  // 'Color.blue'
}
```

## 枚举类型有以下限制

- 你不能在子类中混合或实现一个枚举。
- 你不能显式实例化一个枚举。

更多信息，见 Dart Language Specification (https://www.dartlang.org/docs/spec/) 。

## 为类添加特征：mixins

mixins 是一种多类层次结构的类的代码重用。

要使用 mixins ，在 with 关键字后面跟一个或多个 mixin 的名字。下面的例子显示了两个使用mixins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 mixin ，就创建一个继承 Object 类的子类，不声明任何构造函数，不调用 `super` 。例如：

```
abstract class 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');
    }
  }
}
```

更多信息，见文章 Mixins in Dart (https://www.dartlang.org/articles/mixins/) 。

## 类的变量和方法

使用 `static` 关键字来实现类变量和类方法。

## 静态变量

静态变量（类变量）对于类状态和常数是有用的：

```
class Color {
  static const red =
      const Color('red'); // 一个恒定的静态变量
  final String name;      // 一个实例变量。
  const Color(this.name); // 一个恒定的构造函数。
}

main() {
  assert(Color.red.name == 'red');
}
```

只有当静态变量被使用时才被初始化。

**注**：本章内容依据代码风格指南推荐的 lowerCamelCase 来为常量命名。

## 静态方法

静态方法（类方法）不在一个实例上进行操作，因而不必访问 `this` 。例如：

```
import 'dart:math';

class Point {
  num x;
  num 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);
  }
}

main() {
  var a = new Point(2, 2);
  var b = new Point(4, 4);
  var distance = Point.distanceBetween(a, b);
  assert(distance < 2.9 && distance > 2.8);
}
```

**注**：考虑到使用高阶层的方法而不是静态方法，是为了常用或者广泛使用的工具和功能。

你可以将静态方法作为编译时常量。例如，你可以把静态方法作为一个参数传递给静态构造函数。

# 11

# 泛型

如果你在API文档寻找基本数组类型或者 List (https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core.List) 类型，你将会看到该类型实际上为List <E> ,其中<...>标记表示此表为一个泛型类型（或为参数化结构）—— 一种含有正规类型参数的类型。按照惯例，类型变量通常为单字符名称，例如E,T,S,K,以及V。

## 为何要使用泛型?

因为在Dart中类型是可选的，你不一定使用泛型。或许你想用，可是，因为一些相同的原因你会想在代码中使用其他的类型：这些类型（泛型或者其他类型）可以注释你的文档以及代码，使你的意图更加清晰。

比如，如果你打算使用一个仅仅包含字符串的List，你可以声明它为List <String> （可理解为“字符串类型组成的List”），通过这种方式，你的程序员同事，以及你的工具（比如Dart编辑器和调试模式下的Dart虚拟机）能检测到将一个非字符串的变量分配到List中很可能是错误的，这里给出一个样例：

```
var names = new List<String>();
names.addAll([‘Seth’,’Kathy’,’Lars’]);
//...
names.add(42); //在调试模式中失败 (在生产模式中成功).
```

另外一个使用泛型的原因是为了减少代码的重复。泛型可以让你能共享多个类型的一个接口和实现方式，它在调试模式以及静态分析的错误预警中仍然很有优势。举个例子，当你在创建一个接口来缓存一个对象时：

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

你发现你想要一个字符串专用的接口，所以你创建了另外一个接口：

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

接下来，你决定你想要一个这种接口的数字专用的接口...你想到了这个方法.
泛型类型可以减少你创建这些接口的困难。取而代之的是，你只需要创建一个带有一个类型参数的接口即可：

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

在这个代码中，T是一个替代类型，即占位符，你可以将他视为后续被开发者定义的类型。

## 使用集合常量

Lis常量以及map常量都能被参数化，参数常量就像你已经见过的常量那样，除非你在左 方括号之前添加 <type> (对于List ) 或者 <keyType,valuetype> (对于map ) 。当你需要避免调试模式下的类型警告，你或许可以使用参数常量。这里有一个使用常量类型的例子：

```
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’
};
```

## 使用带构造器的参数化类型

为了在使用构造器时详细说明一个或多个类型，将类型放入类名后的三角括号（ <...> ）中，举个例子：

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

下列代码创建了一个含有整型的键以及值为View的map：

~~~Dart var views = new Map<int,view>(); ~~~

## 泛型集合及其包含的类型

Dart泛型类型是被具体化的，意思就是它们在整个运行时间中都携带着类型信息。举个例子，你可以测试一个集合中的类型甚至是在生产模式中：

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

然而，上述表达式检查的仅仅是集合中的类型--并不是其中的对象。在生产模式下，一个List <String> 中可能含有一些非字符项，解决方法可以是逐项检查其类型或者在异常处理程序中加入数据项操作代码(查看 异常 () )。
~~~

**提示**：相反的，在Java中泛型使用erasure，意思就是泛型类型的参数在运行中将会被抹除。在Java中你可以测试一个对象是否是一个表，但是你不能测试它是否是一个List <String> .

更多关于泛型的信息，请见 Optional Types in Dart (https://www.dartlang.org/articles/optional-types/)

12

# 库和可见性

import,part,library指令可以帮助创建一个模块化的，可共享的代码库。库不仅提供了API，还提供隐私单元：以下划线（_）开头的标识符只对内部库可见。每个Dartapp就是一个库，即使它不使用库指令。

库可以分布式使用包。见 Pub Package and Asset Manager (https://www.dartlang.org/tools/pub/) 中有关pub(SDK中的一个包管理器）。

## 使用库

使用 import 来指定如何从一个库命名空间用于其他库的范围。

例如，Dart Web应用一般采用这个库 dart:html (http://api.dartlang.org/html.html)，可以这样导入：

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

唯一需要 import 的参数是一个指向库的 URI。对于内置库，URI中具有特殊dart:scheme。对于其他库，你可以使用文件系统路径或package:scheme。包 *package*：scheme specifies libraries，如pub工具提供的软件包管理器库。例如：

```
import 'dart:io';
import 'package:mylib/mylib.dart';
import 'package:utils/utils.dart';
```

> 注：URI代表统一资源标识符。网址（统一资源定位器）是一种常见的URI的。

## 指定库前缀

如果导入两个库是有冲突的标识符，那么你可以指定一个或两个库的前缀。例如，如果 library1 和 library2 都有一个元素类，那么你可能有这样的代码：

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;
// ...
var element1 = new Element(); // 使用lib1里的元素
var element2 =
    new lib2.Element();       // 使用lib2里的元素
```

## 导入部分库

如果想使用的库一部分，你可以选择性导入库。例如：

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

```
//导入所有除了foo
import 'package:lib2/lib2.dart' hide foo;
```

## 延迟加载库

延迟(deferred)加载（也称为延迟(lazy)加载）允许应用程序按需加载库。下面是当你可能会使用延迟加载某些情况：

- 为了减少应用程序的初始启动时间；
- 执行A / B测试–尝试的算法的替代实施方式中；
- 加载很少使用的功能，例如可选的屏幕和对话框。

> 警告：延迟加载是先在1.6实现的新功能。如果你使用它，你发现任何执行问题，请让我们知道 (http://dartbug.com/) 。

为了延迟加载一个库，你必须使用 deferred as 先导入它。

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

当需要库时，使用该库的调用标识符调用 LoadLibrary（）。

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

在前面的代码，在库加载好之前，await关键字都是暂停执行的。有关 async 和 await 见 asynchrony support 的更多信息。

您可以在一个库调用 LoadLibrary（）多次都没有问题。该库也只被加载一次。

当您使用延迟加载，请记住以下内容：

- 延迟库的常量在其作为导入文件时不是常量。记住，这些常量不存在，直到迟库被加载完成。
- 你不能在导入文件中使用延迟库常量的类型。相反，考虑将接口类型移到同时由延迟库和导入文件导入的库。
- Dart隐含调用LoadLibrary（）插入到定义deferred as namespace。在调用LoadLibrary（）函数返回一个Future。

## 库的实现

用 library 来来命名库，用part来指定库中的其他文件。 注意：不必在应用程序中（具有顶级main（ ）函数的文件）使用library，但这样做可以让你在多个文件中执行应用程序。

### 声明库

利用library identifier（库标识符）指定当前库的名称：

```
// 声明库，名ballgame
library ballgame;

// 导入html库
import 'dart:html';

// ...代码从这里开始...
```

### 关联文件与库

添加实现文件，把part fileUri放在有库的文件，其中fileURI是实现文件的路径。然后在实现文件中，添加部分标识符（part of identifier），其中标识符是库的名称。下面的示例使用的一部分，在三个文件来实现部分库。

第一个文件，ballgame.dart，声明球赛库，导入其他需要的库，并指定ball.dart和util.dart是此库的部分：

```
library ballgame;

import 'dart:html';
// ...其他导入在这里...

part 'ball.dart';
part 'util.dart';

// ...代码从这里开始...
```

第二个文件ball.dart，实现了球赛库的一部分：

```
part of ballgame;

// ...代码从这里开始...
```

第三个文件，util.dart，实现了球赛库的其余部分：

```
part of ballgame;

// ...Code goes here...
```

### 重新导出库(Re-exporting libraries)

可以通过重新导出部分库或者全部库来组合或重新打包库。例如，你可能有实现为一组较小的库集成为一个较大库。或者你可以创建一个库，提供了从另一个库方法的子集。

```
// In french.dart:
library french;

hello() => print('Bonjour!');
goodbye() => print('Au Revoir!');

// In togo.dart:
library togo;

import 'french.dart';
export 'french.dart' show hello;

// In another .dart file:
import 'togo.dart';

void main() {
  hello();   //print bonjour
  goodbye(); //FAIL
}
```

13

# 异步的支持

Dart 添加了一些新的语言特性用于支持异步编程。最通常使用的特性是 `async` 方法和 `await` 表达式。Dart 库大多方法返回 `Future` 和 `Stream` 对象。这些方法是异步的：它们在设置一个可能的耗时操作（比如 `I/O` 操作）之后返回，而无需等待操作完成

当你需要使用 `Future` 来表示一个值时，你有两个选择。

- 使用 `async` 和 `await`
- 使用 `Future API`

同样的，当你需要从 `Stream` 获取值的时候，你有两个选择。

- 使用 `async` 和一个异步的 for 循环（`await for`）
- 使用 `Stream API`

使用 `async` 和 `await` 的代码是异步的，不过它看起来很像同步的代码。比如这里有一段使用 `await` 等待一个异步函数结果的代码：

`await lookUpVersion()`

要使用 `await`，代码必须用 `await` 标记

```
checkVersion() async {
  var version = await lookUpVersion();
  if (version == expectedVersion) {
    // Do something.
  } else {
    // Do something else.
  }
}
```

你可以使用 `try`，`catch`，和 `finally` 来处理错误并精简使用了 `await` 的代码。

```
try {
  server = await HttpServer.bind(InternetAddress.LOOPBACK_IP_V4, 4044);
} catch (e) {
  // React to inability to bind to the port...
}
```

## 声明异步函数

一个异步函数是一个由 `async` 修饰符标记的函数。虽然一个异步函数可能在操作上比较耗时，但是它可以立即返回–在任何方法体执行之前。

```
checkVersion() async {
  // ...
}

lookUpVersion() async => /* ... */;
```

在函数中添加关键字 `async` 使得它返回一个 `Future`，比如，考虑一下这个同步函数，它将返回一个字符串。

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

如果你想更改它成为异步方法-因为在以后的实现中将会非常耗时-它的返回值是一个 `Future` 。

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

请注意函数体不需要使用 `Future API`，如果必要的话 Dart 将会自己创建 `Future` 对象。

## 使用带 future 的 await 表达式

一个 `await` 表达式具有以下形式

```
await expression
```

在异步方法中你可以使用 `await` 多次。比如，下列代码为了得到函数的结果一共等待了三次。

```
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，那么这个值会自动转为 `Future` 。这个 `Future` 对象表明了表达式应该返回一个对象。`await 表达式` 的值就是返回的一个对象。在对象可用之前， `await` 表达式将会一直处于暂停状态。

**如果 `await` 没有起作用，请确认它是一个异步方法。**比如，在你的 `main()` 函数里面使用 `await`， `main()` 的函数体必须被 `async` 标记：

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

## 结合 streams 使用异步循环

一个异步循环具有以下形式：

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

`表达式` 的值必须有Stream 类型（流类型）。执行过程如下：

1. 在 stream 发出一个值之前等待
2. 执行 for 循环的主体，把变量设置为发出的值。
3. 重复 1 和 2，直到 Stream 关闭

如果要停止监听 stream ，你可以使用 `break` 或者 `return` 语句，跳出循环并取消来自 `stream` 的订阅 。

**如果一个异步 for 循环没有正常运行，请确认它是一个异步方法。**比如，在应用的 `main()` 方法中使用异步的 for 循环时，main() 的方法体必须被 `async` 标记。

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

更多关于异步编程的信息，请看 `dart:async` 库部分的介绍。你也可以看文章 Dart Language Asynchrony Support: Phase 1 (https://www.dartlang.org/articles/await-async/) 和 Dart Language Asynchrony Support: Phase 2 (https://www.dartlang.org/articles/beyond-async/) , 和 the Dart language specification (https://www.dartlang.org/docs/spec/)

# Isolates

现在的网页浏览器，甚至是移动平台上的，运行在多核 CPU 之上。为了充分利用多核心的优势，开发人员通常对共享内存的线程采取并行策略。然而，在共享状态下使用并发容易出错并且会使代码复杂化。

Dart 在代码中使用 isolates 来替代线程。每个 isolate 有自己的内存堆，以确保 isolate 的状态不能被其他任何 isolate 访问。

# 15

# Typedefs

在 Dart 中，方法是对象，就像字符串和数字也是对象。typedef ,又被称作函数类型别名，让你可以为函数类型命名，并且该命名可以在声明字段和返回类型的时候使用。当一种函数类型被分配给一个变量的时候，typedef 会记录原本的类型信息。

考虑下面的代码，哪一个没有使用 typedef。

```
class SortedCollection {
  Function compare;

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

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

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

  // All we know is that compare is a function,
  // but what type of function?
  assert(coll.compare is Function);
}
```

当 `f` 分配到 `compare` 的时候类型信息丢失了。`f` 的类型是 `(Object, Object)` → `int` (→ 意味着返回的)，然而 `compare` 的类型是方法。如果我们使用显式的名字更改代码并保留类型信息，则开发者和工具都可以使用这些信息。

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

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

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

main() {
  SortedCollection coll = new SortedCollection(sort);
  assert(coll.compare is Function);
```

```
  assert(coll.compare is Compare);
}
```

**请注意** 目前 typedefs 仅限于函数类型，我们期望这一点能有所改变。

因为 typedefs 是简单的别名，所以它提供了一种方法来检查任何函数的类型。比如：

```
typedef int Compare(int a, int b);

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

main() {
  assert(sort is Compare); // True!
}
```

# 16

# 元数据

使用元数据来给你的代码提供附加信息。

元数据注解以 *@* 字符开头，后面跟一个编译时的常量引用（例如 *deprecated*）或者调用常量构造器的语句。

所有的 Dart 代码中支持三个注解：*@deprecated*，*@override* 和*@proxy*。*@override* 和*@proxy*的用法示例，请查看类的继承 (页 1)。以下是 *@deprecated* 用法的示例：

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

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

你可以定义你自己的元数据注解。下面的例子定义了一个有两个参数的 *@todo* 注解：

```
library todo;

class todo {
  final String who;
  final String what;

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

下面是使用 *@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、构造器、工厂、函数、属性、参数、变量声明、*import* 或 *export* 指令之前。你可以在运行时通过反射来取回元数据。

# 17

# 注释

Dart 支持单行注释、多行注释和文档注释。

## 单行注释

单行注释由 `//` 开始。每一行中 `//` 到行尾之间的内容会被 Dart 编译器忽略。

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

## 多行注释

一段多行注释由 `/*` 开始，由 `*/` 结束。在 `/*` 和 `*/` 之间的内容会被 Dart 编译器忽略（除非他们是文档注释；请看下面的部分）。多行注释可以嵌套。

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

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

## 文档注释

文档注释是由 `///` 或 `/**` 开始的多行或单行注释。在连续的行上使用 `///` 的效果等同于多行注释。

在一段文档注释中，Dart 编译器忽略所有除括号内的文本。你可以使用括号来引用类、方法、属性、顶级变量、函数和参数。括号中的名字会在被文档化程序元素的词法范围内解析。

下面是一个引用了其它类和参数的文档注释的例子：

```
/// 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]` 变成了指向 Food 类的 API 文档连接。

为了转换 Dart 代码并生成 HTML 文档，你可以使用 SDK 的 文档生成器 (https://www.dartlang.org/tools/dartdocgen/) 。生成文档的示例，请参阅 Dart API 文档 (http://api.dartlang.org/) 。关于如何组织你的文档，请参阅 文档注释准则 (https://www.dartlang.org/articles/doc-comment-guidelines/) 。

# 18

# 总结

本章总结了 Dart 最常用的语言特性。更多的特性正在实现当中，但我们希望不会破坏原有的代码。更多信息请参阅 Dart 语言规范 (https://www.dartlang.org/docs/spec/) 和 惯用 Dart (https://www.dartlang.org/articles/idiomatic-dart/) 之类的 文章 (https://www.dartlang.org/articles/) 。