Dart基本语法

哆啦_ 关注

1 2019.08.27 17:12:32 字数 7,825 阅读 12,166

重要概念

- 可以放在变量中的都是**对象**,所有对象都是类的实例,包括数字,函数,null都是对象, 所有对象都是继承自 Object类
- Dart类型是强类型语言,但是Dart也支持类型推断,如果要明确说明不需要任何类型，请使用特殊类型dynamic。
- Dart支持泛型,比如 `List<int>` (包含int的数组), `List<dynamic>` (包含任意类型对象的数组)
- Dart支持top-level函数(比如main()),以及方法(静态方法和实例方法). 也可以在函数中创建函数(内嵌函数或本地函数)
- 同样的,Dart支持top-level变量,以及绑定到类或对象的变量(静态变量和实例变量),实例变量有时候被称为 字段或者属性
- 不像Java,Dart没有 `public` , `protected` , `private` 关键字.如果标识符以下划线(_)开头,则它就是私有变量
- 标识符可以以字母或下划线(_)开头

变量

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

变量存储引用, name变量包含了一个String对象的引用,该对象的值是'Bob'.

name被推断为String类型,但是可以通过指定来改变其类型,如果对象不局限于单一类型,可以指定为Object或者dynamic

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

默认值

未初始化的变量的初始值为 `null` ,即使数字类型的初始值也是 `null` ,因为在Dart中, everything is object!

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

> **重点:** 生产环境下,代码会忽略 assert()调用, 开发环境中, 断言会在条件为false时抛出异常

final 和 const

如果永远不会修改变量,使用 *final* 或者 *const*。而不是使用*var*或者其他类型。

final 变量只会被设置一次，一个 *const* 变量是编译时常量（const是隐式的final），final 修饰的top-level变量和类变量在第一次使用时初始化

> **Note:** 实例变量可以是final，但不能是const， final 实例变量必须在构造函数体开始之前初始化(可以在变量声明，构造函数参数，或者在构造函数的初始化列表)

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

不能修改final变量的值

```
name = 'Alice'; // Error: a final variable can only be set once.

```

const 用于编译时常量。 如果*const*在类中使用， 使用*static const*标识。在声明该变量的地方，将值设置为编译时常量，比如数字或者字符串，

```

const bar = 1000000; // Unit of pressure (dynes/cm2)
const double atm = 1.01325 * bar; // Standard atmosphere

```

const 关键字并不只是用来声明常量变量，也可以用来创建常量值，以及声明创建常量值的构造函数。任意变量都可以有一个常量值

```

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

```

可以从*const*声明的初始化表达式中省略*const*，就像上面的 `baz` 。

可以改变 non-final,non-const变量的值，即使该变量有过一个 const 值：

```
foo = [1,2,3];// 之前是 const[]

```

但是不能修改const变量的值：

```
baz = [42];// Error

```

> 同样的， `var foo = const [];` foo这个数组的值也不能再改变，即foo不能添加/移除元素
>
> `foo.add(1);` error
>
> `Unsupported operation: Cannot add to an unmodifiable list`

内置类型

Dart语言特别支持以下类型：

- numbers
- strings
- booleans
- lists (也就是数组)
- sets
- maps

- runes (为了在字符串中表示Unicode字符)
- symbols

可以使用字面量来初始化这些类型。

Number

Dart数字有两种类型

int

不超过64位的整数，具体取决于平台。在Dart VM上，值的范围： -2^{63} 到 $2^{63}-1$。编译为JavaScript的Dart使用的是Javascript number，值的范围是 -2^{53} 到 $2^{53}-1$

double

64位(双精度)浮点型数字。由IEEE 754标准规定

int和double都是num的子类。 num类包含基础运算符，比如+，-，*，/，以及abs(),ceil(),floor(),(在int类里有位运算符，比如<<)

在Dart2.1中，整型会在需要时自动转为double类型

```
double z = 1;// 等同于 double z = 1.0;
```

下面是字符串转为数字，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');

```

String

Dart字符串是一系列UTF-16代码单元。 可以使用单引号或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.";

```

可以使用*${expression}*将表达式的值放在字符串中，如果表达式是标识符，可以省略{}, 要获得与对象对应的字符串，可以调用对象的 `toString()` 方法

可以使用相邻的字符串或者 + 号来连接两个字符串

```
// 相邻的字符串
var s1 = 'String '
    'concatenation'
    " works even over line breaks.";
```

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

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

```

创建多行字符串的方法：使用单引号/双引号的三重引号

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

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

```

可以添加一个前缀 r来创建一个'raw'字符串

```
var s = r'In a raw string, not even \n gets special treatment.';
```

bool

Dart中的布尔类行为bool，有两个值：true,false;

Dart是类型安全的，也就意味着不会像OC那样有非0即真的情况。条件表达式中必须明确传递一个布尔值。

```
// Check for an empty string.
var fullName = '';
assert(fullName.isEmpty);

// Check for zero.
var hitPoints = 0;
assert(hitPoints <= 0);

// Check for null.
var unicorn;
assert(unicorn == null);

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

List

Dart中的数组，有序集合。

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

> 注意： Dart会类型推断list的类型为 List<int>. 如果之后向其中添加其他非int的对象，编译器会抛出错误

跟其他语言的数组一样，List的下标索引从0开始，

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

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

要创建编译时常量的列表，请在列表前添加const:

```
var constantList = const [1,2,3];
// constantList[1] = 1; 会有错误，因为列表是常量 不能再修改了
```

Dart2.3扩展运算符(...)和空值感知运算符(...?)，它提供了一种将多个元素插入到集合的简洁方法。

比如，可以使用扩展运算符将一个列表中的所有元素插入到另一个列表中

```
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有可能为null，可以使用空值感知运算符来避免异常。

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

Dart2.3也引入了**collection if**和**collection for**来创建集合。

下面是使用**collection if**的例子，列表包含三个/四个 item：

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

使用**collection for**来操作列表item,然后将它们添加到另一个列表：

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

```

Set

Dart中的无序集合是Set,

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

```

创建一个空的Set,请使用前面带有类型参数的{},或者将{}赋给类行为Set的变量

```
var names = <String>{};
// Set<String> names = {}; // This works, too.
// var names = {}; // Creates a map, not a set.

```

> **Set还是Map?** Map和Set的字面亮语法类似。由于Map首先出现，所以{}默认是Map类型。如果忘记了{}的类型注释，Dart会创建一个Map<dynamic, dynamic>类型的对象，也就是说{}默认会是Map类型

`add()`，`addAll()` 方法来为Set添加元素：

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

```

使用 `.length` 来获取Set的元素个数：

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

要创建一个编译时常量的Set，在Set前添加const

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

类似于List， Set也支持扩展运算符(...)和空值感知运算符(...?);

Map

类似于iOS中的字典。key和value都可以是**任意类型**的对象，key只能出现一次，value可以出现多次。

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

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

也可以使用Map的构造函数来创建Map：

```
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';

```

向一个Map中添加键值对，类似于JS：

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

如果查询一个不存在的key，会返回null：

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

使用 `.length` 来获取Map元素的个数：

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

```

创建一个编译时常量的Map，在Map字面量前使用const

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

// constantMap[2] = 'Helium'; //error

```

Dart2.3之后，Map也支持 `...` 和 `...?`

Runes

在Dart中，Runes是字符串的UTF-32代码点。

Unicode为世界上所有书写系统中的每个字母，数字和符号定义了唯一的数值。由于Dart字符串是 UTF-16编码的序列，因此在字符串中表示32位的Unicode值需要特殊的语法。

表达Unicode代码点的常用方法是 `\uXXXX`，其中XXXX是4位十六进制值。 例如，心脏字符（❤）是 `\u2665` 。 要指定多于或少于4个十六进制数字，请将值放在大括号中。 例如，笑的表情符号（😆）是 `\u{1f600}` 。

String类有几个属性可用于提取rune信息。**codeUnitAt**和**codeUnit**属性返回16位的code unit。使用**runes**属性获取字符串的runes

以下示例说明了runes(符文)、16位代码单元和32位代码点之间的关系：

```
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));
}
```

打印信息：

```
flutter: 👏
flutter: [55357, 56399]
flutter: [128079]
flutter: ♥ 😅 😎 👻 🖖 👍
```

Symbol

Symbol对象表示Dart程序中声明的运算符或标识符。可能永远也用不到Symbol。。。

要获取标识符的符号，可以使用symbol字面量， `#` 后跟标识符：

```
#radix
#bar
```

Function

Dart是真面向对象的语言，所以即使是函数也是一个对象，类型为Function。这意味着函数可以分配给变量或者作为参数传递给其他函数。

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

或者

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

=> expr是{ return expr; }的缩写，

函数可以有两种类型的参数：必需和可选。必需参数放在首位，后面跟着一些可选参数，

可选参数

可选参数可以是位置参数，也可以是命名参数

可选命名参数

当调用一个函数的时候，可以使用*paramName: value*的形式指定命名参数：

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

当定义一个函数时，使用*{param1, param2, ...}*的形式来制定命名参数：

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

```

Flutter实例创建表达式可能变得复杂，因此Widget构造函数仅使用命名参数。 这使得实例创建表达式更易于阅读。

你可以在任何Dart代码(不仅是Flutter)中使用 `@required` 来注释一个命名参数，来表明该参数是必须的：

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

当构建Scrollbar时，如果 `child` 参数缺失，就会报一个错误

可选位置参数

把一组函数参数包括在 `[]` ,来标记这些参数是可选位置参数：

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

调用该函数-不传可选参数

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

调用该函数-传递可选参数

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

```

参数默认值

可以使用 = 来为命名参数或者位置参数设置默认值。默认值必须是**编译时常量**，如果没有提供默认值，那默认值就是null。

举个例子：

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

// bold will be true; hidden will be false.
enableFlags(bold: true);

```

下面示范了如何为位置参数设置默认值：

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

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

```

也可以传一个List或者Map作为默认值：

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

main()函数

每一个app都有一个顶层的main()函数，作为应用的入口点。该函数返回值为void，并接收一个可选的 `List<String>` 参数。

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

```dart
// Run the app like this: 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');
}

```

函数作为first-class对象

函数本身可以作为一个参数传递给另一个函数，比如：

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

var list = [1, 2, 3];

// Pass printElement as a parameter.
list.forEach(printElement);
```

也可以把函数赋值给一个变量：

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

匿名函数

大多数函数是有名字的，比如 `main()` 或者 `printElement ()`,我们也可以创建一个没有名字的函数-匿名函数，或者创建lambda以及闭包。你可以把匿名函数赋值给一个变量，方便添加到一个集合中，或者从集合中删除。

匿名函数看起来类似于命名函数-零个或多个参数，

以下示例定义了一个匿名函数，有一个无类型参数item。

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

词汇范围

Dart是一种词法范围的语言，这意味着变量的范围是静态确定的(只需通过代码的布局)。

```dart
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);
    }
  }
}

```

闭包

闭包是一个函数对象，它可以访问其词法范围中的变量，即使该函数在其原始范围之外使用也是如此。

函数可以关闭周围范围中定义的变量。 在以下示例中，makeAdder（）捕获变量addBy。 无论返回的函数在哪里，它都会记住addBy。

```
/// Returns a function that adds [addBy] to the
/// function's argument.
Function makeAdder(num addBy) {
  return (num i) => addBy + i;
}

void main() {
  // Create a function that adds 2.
  var add2 = makeAdder(2);

  // Create a function that adds 4.
  var add4 = makeAdder(4);

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

函数相等

这里有个例子来测试顶层函数，静态函数，以及实例函数的相等性：

```
void foo() {} // A top-level function

class A {
  static void bar() {} // A static method
  void baz() {} // An instance method
}

void main() {
  var x;

  // Comparing top-level functions.
  x = foo;
  assert(foo == x);

  // Comparing static methods.
  x = A.bar;
  assert(A.bar == x);

  // Comparing instance methods.
  var v = A(); // Instance #1 of A
  var w = A(); // Instance #2 of A
  var y = w;
  x = w.baz;

  // These closures refer to the same instance (#2),
  // so they're equal.
  assert(y.baz == x);

  // These closures refer to different instances,
  // so they're unequal.
  assert(v.baz != w.baz);
}

```

返回值

所有的函数都有**返回值**，如果没有指定返回值，则返回null。

```
foo() {}

assert(foo() == null);

```

操作符

描述	操作符
一元后缀	expr++ expr-- () [] . ?.
一元前缀	-expr !expr ~expr ++expr --expr
乘法	* / % ~/
加法	+ -
位移	<< >> >>>
按位与	&
按位异或	^
按位或	|
关系和类型测试	>= > <= < as is is!
相等	== !=
逻辑与	&&
逻辑或	||
if null	??
条件	expr1 ? expr2 : expr3
级联	..
赋值	= *= /= += -= &= ^= 等

这里有几个操作符的用法

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

在上述表格中，每个操作符都比其下一行的操作符有更高的优先级。比如，乘法运算符%比相等运算符==有更高的优先级(因此在==之前先执行%)。相等运算符==优先级高于逻辑与运算符&&。该优先级意味着以下两行代码以相同的方式执行:

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

// 比较难读，但跟上面是相等的
if (n % i == 0 && d % i == 0) ...

```

算数运算符

Dart支持以下的算数运算符

运算符	含义
+	加法
-	相减
-expr	取反
*	相乘
/	相除,返回的是double

运算符	含义
~/	相除，返回的是整数int
%	取余

比如：

```
assert(2 + 3 == 5);
assert(2 - 3 == -1);
assert(2 * 3 == 6);
assert(5 / 2 == 2.5); // Result is a double
assert(5 ~/ 2 == 2); // Result is an int
assert(5 % 2 == 1); // Remainder

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

```

Dart还支持自增自减运算：

```

var a, b;

a = 0;
b = ++a; // Increment a before b gets its value.
assert(a == b); // 1 == 1

a = 0;
b = a++; // Increment a AFTER b gets its value.
assert(a != b); // 1 != 0

a = 0;
b = --a; // Decrement a before b gets its value.
assert(a == b); // -1 == -1

a = 0;
b = a--; // Decrement a AFTER b gets its value.
assert(a != b); // -1 != 0
```

相等和关系运算符

跟其他语言的一样

类型判断运算符

as, is, 和 is! 操作符在运行时检查类型非常方便。

操作符	含义
as	类型转换(也用于指定库前缀)
is	如果对象具有指定的类型则返回true
is!	如果对象具有指定的类型返回false

如果 `obj` 实现了由 `T` 指定的接口， `obj is T` 的结果为true。比如， `obj is Object` 永远都是true。

使用 `as` 运算符将对象强制转换为特定的类型。 通常情况下应该将 `as` 作为 `is` 的简写，比如：

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

可以使用 `as` 来简写：

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

> **注意：** 该代码并不是等价的。如果 emp是null或者不是Person类的对象。使用is不会有什么影响，使用as的话会抛出异常。

赋值运算符

```
// 将value赋值给a
a = value;

// 如果b是null，将value赋值给b；否则，b将保持不变
b ??= value;
```

逻辑运算符

跟其他语言类似

位运算

与C一样

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

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

```

条件表达式

Dart有两个运算符，可以简明地计算可能需要if-else语句的表达式：

`condition ? expr1 : expr2`

三目运算符，跟其他语言一样

`expr1 ?? expr2`

如果expr1是 non-null,则返回它的值，否则，计算并返回expr2的值。

如果要基于布尔表达式来赋值的话使用三目运算

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

如果布尔表达式要测试null，请考虑使用??。

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

前面的例子至少可以用其他两种方式编写，但不够简洁：

```

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

// Very long version uses if-else statement.
String playerName(String name) {
```

```
  if (name != null) {
    return name;
  } else {
    return 'Guest';
  }
```

级联表示法(..)

级联（ .. ）允许对同一对象进行一系列操作。 除了函数调用，还可以访问同一对象上的字段。 这通常可以节省创建临时变量的步骤，并允许编写更多流畅的代码。

```

querySelector('#confirm') // Get an object.
  ..text = 'Confirm' // Use its members.
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'));

```

上面的代码等同于:

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

```

也可以内嵌我们的级联表达式，比如:

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

```

小心在返回实际对象的函数上构造级联。 例如，以下代码会失败:

```
var sb = StringBuffer();
sb.write('foo')
  ..write('bar'); // Error: method 'write' isn't defined for 'void'.

```

`sb.write()` 函数返回void，不能在void上构建级联。

> **注意：**严格来说，级联的“双点”符号不是运算符。 它只是Dart语法的一部分。

其他运算符

只介绍下 `?.` ： 最左边的操作数可以为null，比如： `foo?.bar` ,如果foo不为null，则从foo中选择bar属性，如果foo为null，则 `foo?.bar` 为null。

控制流语句

if-else,for循环,while/do-while跟其他语言一样

Dart中的Switch语句使用==来比较整数，字符串或者编译时常量。比较对象必须是同一个类的实例(而不是其子类)，并且该类不能覆盖==。

每个非空case子句以break语句结束。 结束非空case子句的其他有效方法是continue，throw或return语句。

```
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();
}
```

断言

跟其他语言的断言一样。Flutter中只有在debug模式下才开启断言。

异常

与Java相比，Dart的所有异常都是未经检查的异常。 方法不会声明它们可能引发的异常，并且不需要捕获任何异常。

Dart提供了Exception和Error类型，以及许多预定义的子类型。 当然，也可以自定义异常。 Dart程序可以抛出任何非null对象(不仅仅是Exception和Error对象)作为异常。

Throw

抛出异常：

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

也可以抛出任意对象：

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

Catch

捕获异常会阻止异常传播（除非重新抛出异常）,并有机会处理它：

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

要处理可能抛出多种类型异常的代码，可以指定多个catch子句。 与抛出对象的类型匹配的第一个catch子句处理异常。 如果catch子句未指定类型，则该子句可以处理任何类型的抛出对象：

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

```
  // A specific exception
  buyMoreLlamas();
} on Exception catch (e) {
  // Anything else that is an exception
  print('Unknown exception: $e');
} catch (e) {
  // No specified type, handles all
  print('Something really unknown: $e');
}
```

如前面的代码所示，可以使用 on 或 catch 或两者结合使用。 需要指定异常类型时使用 on , 在需要异常对象时使用 catch 。

可以指定两个参数到 catch() 。第一个参数是抛出的异常，第二个是堆栈跟踪信息(一个 StackTrace 对象)。

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

要部分处理异常，同时允许异常传播，可以使用 rethrow 关键字。

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

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

Finally

为了确保无论异常是否抛出都会执行一些代码，可以使用 finally 语句。如果 catch 语句没有匹配到该异常，则该异常会在执行 finally 语句的代码之后抛出。

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

```

finally 语句会在匹配异常的 catch 语句之后执行：

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

类

Dart是一种面向对象的语言，具有类和基于*mixin*的继承。 每个对象都是一个类的实例，所有类都来自Object。 基于*Mixin*的继承意味着虽然每个类（除了Object）只有一个超类，但是类的body可以在多个类层次结构中重用。

类的成员

类具有函数、方法以及实例变量等成员。

使用点语法(.)来引用实例变量或者方法：

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

// Set the value of the instance variable y.
p.y = 3;

// Get the value of y.
assert(p.y == 3);

// Invoke distanceTo() on p.
num distance = p.distanceTo(Point(4, 4));
```

使用 `?.` 来代替 `.` ，可以防止左边运算对象为null的异常。

```
// If p is non-null, set its y value to 4.
p?.y = 4;
```

使用构造函数

可以使用构造函数创建对象。 构造函数名称可以是 `ClassName` 或 `ClassName.identifier` 。 例如，以下代码使用 `Point()` 和 `Point.fromJson()` 构造函数创建 `Point` 对象：

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

一些类提供了编译时构造函数，创建一个编译时常量。在构造函数名称前加 `const` 关键字：

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

```

构造两个相同的编译时常量时，只会产生一个实例：

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

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

在常量上下文中，可以在构造函数或字面量之前省略 `const` ：

```
// Lots of const keywords here.
const pointAndLine = const {
  'point': const [const ImmutablePoint(0, 0)],
  'line': const [const ImmutablePoint(1, 10), const ImmutablePoint(-2, 11)],
};
```

除了第一个 `const` ,其他的都可以省略：

```
// 只有一个const，它建立了恒定的上下文。
const pointAndLine = {
  'point': [ImmutablePoint(0, 0)],
  'line': [ImmutablePoint(1, 10), ImmutablePoint(-2, 11)],
};

```

如果常量构造函数在常量上下文之外,并且在没有使用 `const` ， 则会创建一个非常量对象：

```
var a = const ImmutablePoint(1, 1); // Creates a constant
var b = ImmutablePoint(1, 1); // Does NOT create a constant

assert(!identical(a, b)); // NOT the same instance!
```

获取对象类型

在运行时获取对象的类型，可以使用对象的 `runtimeType` 属性，返回一个Type对象/

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

实例变量

声明实例变量：

```
class Point {
  num x; // Declare instance variable x, initially null.
  num y; // Declare y, initially null.
  num z = 0; // Declare z, initially 0.
}

```

所有未初始化的实例变量默认值都是 `null` .

所有的实例变量都会生成一个隐式的*getter*方法。非 `final` 实例变量也会隐式的生成*setter*方法。

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

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

构造函数

通过创建与其类同名的函数来声明构造函数。

```
class Point {
  num x, y;

  Point(num x, num y) {
    // There's a better way to do this, stay tuned.
    this.x = x;
    this.y = y;
  }
}
```

`this` 关键字代表着当前实例。在名称冲突时使用 `this` ,一般情况下，Dart会省略 `this` .

Dart具有语法糖，使其变得简单：

```
class Point {
  num x, y;

  // Syntactic sugar for setting x and y
  // before the constructor body runs.

```

```
  Point(this.x, this.y);
}
```

默认构造函数

如果没有声明构造函数，则会提供一个默认的构造函数。默认的构造函数没有参数，并且会调用其父类的无参数的构造函数。

构造函数不能继承

子类不能继承父类的构造函数！

命名构造函数

使用命名构造函数为类实现多个构造函数：

```
class Point {
  num x, y;

  Point(this.x, this.y);

  // Named constructor
  Point.origin() {
    x = 0;
    y = 0;
  }
}
```

重定向构造函数

有时构造函数的唯一目的是重定向到同一个类中的另一个构造函数。 重定向构造函数的body是空的，构造函数调用出现在冒号(:)之后。

```
class Point {
  num x, y;

  // The main constructor for this class.
  Point(this.x, this.y);

  // Delegates to the main constructor.
// 重定向到main函数
  Point.alongXAxis(num x) : this(x, 0);
}
```

常量构造函数

如果类生成的对象永远不会改变，则可以使这些变量为编译时常量。定义一个 `const` 构造函数来确保所有的实例变量都是 `final` 。

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

  final num x, y;

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

工厂(Factory)构造函数

当构造函数不需要每次都创建新的实例时，可以使用 `factory` 关键字。例如，一个工厂构造函数可能从缓存中返回实例，或者可能返回一个子类的实例。

下面的例子展示了工厂构造函数从缓存中返回实例：

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

  // _cache is library-private, thanks to
  // the _ in front of its name.
  static final Map<String, Logger> _cache =
      <String, Logger>{};

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

  Logger._internal(this.name);

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

> **注意:** 工厂构造函数不能访问 `this` .

调用工厂构造函数跟其他的构造函数一样:

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

```

方法

方法是为对象提供行为的函数。(函数是独立存在的，方法需要依赖对象，这就是函数与方法的区别)。

实例方法

实例方法可以访问实例变量和 `this` 。下面的 `distanceTo()` 就是一个实例方法:

```dart
import 'dart:math';

class Point {
  num x, y;

  Point(this.x, this.y);

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

```

getter & setter

每个实例变量都有一个隐式的getter，合适的话还有一个setter。可以通过 `set` 和 `get` 关键字实现setter和getter来创建其他的属性

```dart
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 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;
}

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

抽象方法

实例方法，setter和getter可以是抽象的，可以定义接口，但将其实现留给其他类。抽象方法只能存在于抽象类。

使用分号(;)而不是方法体来定义一个抽象方法：

```
abstract class Doer {
  // Define instance variables and methods...

  void doSomething(); // 定义抽象方法
}

class EffectiveDoer extends Doer {
  void doSomething() {
    // Provide an implementation, so the method is not abstract here...
  }
}

```

抽象类

使用 `abstract` 修饰符来定义抽象类(无法实例化的类)。抽象类对于定义接口非常有用，通常还有一些实现。如果希望抽象类看起来是可以实例化的，请定义工厂构造函数。

抽象类一般具有抽象方法：

```

// 这个类被定义为抽象类，因此它不能实例化
abstract class AbstractContainer {
  // 定义构造函数，字段，方法...

  void updateChildren(); //抽象方法
}
```

隐式接口

每个类都隐式定义一个接口，该接口包含该类的所有实例成员及其实现的所有接口。如果要在不继承class B实现的情况下，创建一个class A来支持class B的API，class A应该 `implements` B的接口。

```
// A person. The implicit interface contains greet().
class Person {
  // In the interface, but visible only in this library.
  final _name;

  // Not in the interface, since this is a constructor.
  Person(this._name);

  // In the interface.
  String greet(String who) => 'Hello, $who. I am $_name.';
}

// An implementation of the Person interface.
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 {...}
```

扩展类

使用 `extends` 创建子类，使用 `super` 引用父类:

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

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

override 成员

子类可以覆盖实例方法、 `getter` 和 `setter` 。可以使用 `@override` 注释来表示要覆盖一个成员:

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

覆盖运算符

您可以覆盖下表中显示的操作符。例如，如果您定义一个向量类，您可能定义一个+方法来添加两个向量。

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

> **注意：** `!=` 是不能覆盖的，因为 `e1 != e2` 是 `!(e1 == e2)` 的语法糖。

下面是一个覆盖 `+` 和 `-` 操作符的例子：

```
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);

  // Operator == and hashCode not shown. For details, see note below.
  // ...
}

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

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

```

如果你覆盖了 `==` ,你也应该覆盖对象的 `hashCode` getter方法。

noSuchMethod()

当代码试图调用不存在的方法或者实例变量，可以覆盖 `noSuchMethod()` 来检测或响应。

```
class A {
  // 除非覆盖了noSuchMethod。使用不存在的成员会导致NoSuchMethodError
  @override
  void noSuchMethod(Invocation invocation) {
    print('You tried to use a non-existent member: ' +
        '${invocation.memberName}');
  }
}
```

枚举类型

使用

使用 `enum` 关键字声明一个枚举类型：

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

每个枚举值都有一个 `index` getter,它返回枚举值的位置。比如，第一个值的 `index` 为0，第二个值的 `index` 为1.

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

要获取枚举中所有的值,可以使用枚举中的 `values` 常量。

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

枚举有以下限制：

- 不能子类化、 `mixin` 或者 `implment` 一个枚举
- 不能显式的实例化枚举

向类添加feature: mixin

mixin是一种在多个继承类中重用代码的一种方式。

若要使用mixin，请使用 `with` 关键字后跟一个或多个mixin名称。下面的例子显示了两个使用mixin的类:

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

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

要实现 `mixin` ，创建一个扩展 `Object` 的类，并且不声明构造函数，除非希望 `mixin` 像常规类一样使用。

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

比如，指定只有某些类型可以使用mixin，这样你的mixin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 {
  // ···
}
```

> Dart 2.1版本中引入了对mixin的支持，早期版本中通常使用抽象类

类变量和方法

使用 `static` 关键字来实现类范围的变量和方法。

静态变量（Static变量）

静态变量(类变量)对于类范围内的状态和常量很有用:

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

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

```

静态变量在使用之前不会初始化

静态方法

静态方法（类方法）不能操作实例，因为不能访问 `this`：

```
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);
}

```

可以使用静态方法作为编译时常量，比如，可以传递一个静态方法作为静态构造函数的参数

> **注意：** 对于通用的或者广泛使用的功能函数，考虑使用顶层函数，而不是静态方法

泛型

如果查看List的API文档，会发现List的实际类型是 `List<E>` 。 `<...>` 表示法将List标记为泛型(或者参数化)类型-具有正式类型参数的类型。按照惯例，大多数类型变量都有单字母名称，例如 E，T，S，K，V

为什么使用泛型

类型安全通常需要泛型，但有更多的好处：

- 正确指定泛型类型会产生更好的代码
- 使用泛型来减少代码重复

如果想要数组仅仅包含字符串，可以声明为 `List<String>` 。这样可非字符串插入到列表中的就会有错误：

```

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

```

使用泛型的另外一个理由是减少代码的重复。泛型允许在多个类型之间分享单个接口和实现。比如：创建一个用于缓存对象的接口：

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

然后发现想要一个特定于字符串的版本，然后可以这样实现：

```

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

```
}

```

后来，你可能需要更多的类型...

泛型可以省去所有这些接口的麻烦，创建一个带有类型参数的接口：

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

在这段代码中， `T` 是一个替身类型，一个占位符，

使用集合字面量

`List` , `Set` , `Map` 可以参数化：

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

跟构造函数一起使用参数化类型

要在使用构造函数时指定一个或多个类型，请将类型放在类名后面的尖括号中(`<...>`),比如：

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

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

```

泛型集合以及其所包含的类型

Dart泛型被具体化，这意味着它们在运行时携带类型信息。例如，你可以测试一个集合的类型:

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

限制参数化类型

当实现一个泛型时，可能想要限制参数的类型，可以使用 `extends` :

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

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

使用 `SomeBaseClass` 或其子类作为泛型参数是OK的：

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

```

不指定泛型参数也是可以的：

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

```

指定任意非 `SomeBaseClass` 类型会导致错误：

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

使用泛型方法

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

`first(<T>)` 上的泛型类型参数允许在几个地方使用类型参数T：

- 函数的返回类型(`T`)
- 参数的类型(`List<T>`)
- 局部变量(`T tmp`)

支持异步

Dart库充满了返回 `Future` 或者 `Stream` 对象的函数。这些函数是异步的：它们在一个可能非常耗时的操作(比如I/O)之后返回，而不需要等待操作完成。

`async` 和 `wait` 关键字支持异步编程，允许我们编写看起来类似同步的异步代码

处理Future

当需要一个完整Future的结果时，有两种选择：

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

使用 `async` 和 `await` 的代码是异步的，但看起来是同步的。比如：

```
await lookUpVersion();

```

要使用 `await` ，代码必须在 `async` 函数中:一个标记为 `async` 的函数:

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

> **注意：**虽然 `async` 函数可能会执行耗时操作，但并不需要等待这些操作。相反， `async` 函数只执行到第一个 `await` 表达式，然后返回一个 `Future` 对象，仅在 `await` 表达式完成后才恢复执行。

使用 `try` ， `catch` ,和 `finally` 在使用 `await` 的代码中来错误：

```
try {
  version = await lookUpVersion();
```

```
} catch (e) {
  // React to inability to look up the version
}
```

可以在 `async` 函数中多次使用 `await`：

```
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 `Future`，如果不是，该值也会自动包装在Future中，`await` 表达式会使执行暂停，知道该对象可用。

如果在使用 `await` 时遇到编译时错误，请确保 `await` 是在 `async` 函数中。
比如在app的 `main()` 函数中使用 `await`，则必须时main函数标记为 `aycnc`

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

处理Stream(流)

当需要从流中获取值的时候，有两个选项：

- 使用 `async` 和异步for循环(`await for`)
- 使用 Stream API

异步for循环：

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

表达式(expression)的值必须具有 `Stream` 类型。执行过程如下：

1. 等流发出一个值
2. 执行for循环的主体，将变量设置为流发出的值
3. 重复1和2，直到流关闭

停止监听流，可以使用 `break` 或者 `return` 语句，该语句会中断for循环并且取消订阅流

如果实现一个异步for循环时出现编译时错误，确保 `await for` 在异步函数中

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

Generators(生成器)

当想要懒加载一系列值时，可以考虑使用*generator函数*，dart内置两种该函数：

- 同步生成器：返回 `Iterable` 对象

- 异步生成器：返回 `Stream` 对象

要实现同步生成器函数，将函数主体标记为 `sync*` ，并使用 `yield` 语句来传递值：

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

```

要实现异步生成器函数，将函数标记为 `async*` ，并使用 `yield` 语句来传递值：

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

```

如果生成器是递归的，可以使用 `yield*` 来提高性能：

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

```

Isolates(隔离区)

大多数计算机，即使在移动平台上，也有多核CPU。 为了利用所有这些核心，开发人员传统上使用并发运行的共享内存线程。 但是，共享状态并发容易出错，并且可能导致代码复杂化。

所有Dart代码都在隔离区内运行，而不是线程。 每个隔离区都有自己的内存堆，确保不会从任何其他隔离区访问隔离区的状态。

文章来源：https://dart.dev/guides/language/language-tour