链滴

Dart 语法《二》

作者：yc654084303

原文链接：https://ld246.com/article/1571715065185

来源网站：链滴

由于篇幅原因，有一部分基础语法在Dart语法《一》中已经说明啦，不明白的请查看Dart语法《一》

本篇主要归纳函数、类、库和可见性、异步支持

函数 Function

● 以下是一个实现函数的例子：

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

1. **main()函数**

● 每个应用程序都必须有一个顶层main()函数，它可以作为应用程序的入口点。该main()函数返回voi并具有List<String>参数的可选参数。

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

● 级联符号..允许您在同一个对象上进行一系列操作。除了函数调用之外，还可以访问同一对象上的段。这通常会为您节省创建临时变量的步骤，并允许您编写更流畅的代码。

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

● 上述例子相对于：

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

● 级联符号也可以嵌套使用。 例如：

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

- 当返回值是void时不能构建级联。 例如，以下代码失败：

```
var sb = StringBuffer();
sb.write('foo') // 返回void
  ..write('bar'); // 这里会报错
```

- **注意： 严格地说，级联的..符号不是操作符。它只是Dart语法的一部分。**

2.**可选参数**

- 可选的命名参数, 定义函数时，使用{param1, param2, ...}，用于指定命名参数。例如：

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

- 可选的位置参数，用[]它们标记为可选的位置参数：

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

- 下面是一个不带可选参数调用这个函数的例子：

```
  say('Bob', 'Howdy'); //结果是： Bob says Howdy
```

- 下面是用第三个参数调用这个函数的例子：

```
  say('Bob', 'Howdy', 'smoke signal'); //结果是：Bob says Howdy with a smoke signal
```

3.**默认参数**

- 函数可以使用=为命名参数和位置参数定义默认值。默认值必须是编译时常量。如果没有提供默认，则默认值为null。
- 下面是为命名参数设置默认值的示例:

```
// 设置 bold 和 hidden 标记的默认值都为false
void enableFlags2({bool bold = false, bool hidden = false}) {
 // ...
}
// 调用的时候：bold will be true; hidden will be false.
 enableFlags2(bold: true);
```

- 下一个示例显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;
 }

 //调用方式:
 say('Bob', 'Howdy'); //结果为:  Bob says Howdy with a carrier pigeon;
```

● 您还可以将list或map作为默认值传递。下面的示例定义一个函数doStuff()，该函数指定列表参数默认list和gifts参数的默认map。

```
// 使用list 或者map设置默认值
 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');
 }
```

4. **作为一个类对象的功能**

● 您可以将一个函数作为参数传递给另一个函数。

```
void printElement(int element) {
print(element);
}
 var list = [1, 2, 3];
// 把 printElement函数作为一个参数传递进来
 list.forEach(printElement);
```

● 您也可以将一个函数分配给一个变量。

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

5.**匿名函数**

● 大多数函数都能被命名为匿名函数，如 main() 或 printElement()。您还可以创建一个名为匿名函的无名函数，有时也可以创建lambda或闭包。您可以为变量分配一个匿名函数，例如，您可以从集中添加或删除它。

● 一个匿名函数看起来类似于一个命名函数 - 0或更多的参数，在括号之间用逗号和可选类型标注分。

● 下面的代码块包含函数的主体:

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

● 下面的示例定义了一个具有无类型参数的匿名函数item，该函数被list中的每个item调用，输出一字符串，该字符串包含指定索引处的值。

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

● 如果函数只包含一条语句，可以使用箭头符号=>来缩短它, 比如上面的例2可以简写成：

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

6.**返回值**

● 所有函数都返回一个值，如果没有指定返回值，则语句return null，隐式地附加到函数体。

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

类（Classes）

1. **对象**

● Dart 是一种面向对象的语言，并且支持基于mixin的继承方式。

● Dart 语言中所有的对象都是某一个类的实例,所有的类有同一个基类--Object。

● 基于mixin的继承方式具体是指：一个类可以继承自多个父类。

● 使用new语句来构造一个类，构造函数的名字可能是ClassName，也可以是ClassName.identifier，例如：

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

  // Create a Point using Point().
  var p1 = new Point(2, 2);

  // Create a Point using Point.fromJson().
  var p2 = new Point.fromJson(jsonData);
```

● 使用.（dot）来调用实例的变量或者方法。

```
 var p = new 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(new Point(4, 4));
```

- 使用 ?.来确认前操作数不为空, 常用来替代. , 避免左边操作数为null引发异常。

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

- 使用const替代new来创建编译时的常量构造函数。

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

- 使用runtimeType方法，在运行中获取对象的类型。该方法将返回Type 类型的变量。

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

2. **实例化变量(Instance variables)**

- 在类定义中，所有没有初始化的变量都会被初始化为null。

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

- 类定义中所有的变量, Dart语言都会隐式的定义 setter 方法，针对非空的变量会额外增加 getter 方法。

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

 main() {
   var point = new 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.
 }
```

3. **构造函数(Constructors)**

- 声明一个和类名相同的函数，来作为类的构造函数。

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

```
  }
}
```

● this关键字指向了当前类的实例, 上面的代码可以简化为：

```
class Point {
num x;
num y;
// Syntactic sugar for setting x and y
// before the constructor body runs.
 Point(this.x, this.y);
 }
```

4. **构造函数不能继承(Constructors aren’t inherited)**

Dart 语言中，子类不会继承父类的命名构造函数。如果不显式提供子类的构造函数，系统就提供默认构造函数

5. **命名的构造函数(Named constructors)**

● 使用命名构造函数从另一类或现有的数据中快速实现构造函数。

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

  Point(this.x, this.y);

  // 命名构造函数Named constructor
  Point.fromJson(Map json) {
    x = json['x'];
    y = json['y'];
  }
}
```

● 构造函数不能被继承，父类中的命名构造函数不能被子类继承。如果想要子类也拥有一个父类一样字的构造函数，必须在子类是实现这个构造函数。

6. **调用父类的非默认构造函数**

● 默认情况下，子类只能调用父类的无名，无参数的构造函数; 父类的无名构造函数会在子类的构造数前调用; 如果initializer list 也同时定义了，则会先执行initializer list 中的内容，然后在执行父类的名无参数构造函数，最后调用子类自己的无名无参数构造函数。即下面的顺序：

1. initializer list（初始化列表）
2. super class’s no-arg constructor（父类无参数构造函数）
3. main class’s no-arg constructor （主类无参数构造函数）

● 如果父类不显示提供无名无参数构造函数的构造函数，在子类中必须手打调用父类的一个构造函数这种情况下，调用父类的构造函数的代码放在子类构造函数名后，子类构造函数体前，中间使用:(colo) 分割。

```
class Person {
```

```
  String firstName;

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

class Employee extends Person {
  // 父类没有无参数的非命名构造函数，必须手动调用一个构造函数
  super.fromJson(data)
  Employee.fromJson(Map data) : super.fromJson(data) {
    print('in Employee');
  }
}

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

  // Prints:
  // in Person
  // in Employee
  if (emp is Person) {
    // Type check
    emp.firstName = 'Bob';
  }
  (emp as Person).firstName = 'Bob';
}
```

7.**初始化列表**

● 除了调用父类的构造函数，也可以通过初始化列表在子类的构造函数体前（大括号前）来初始化实的变量值，使用逗号,分隔。如下所示：

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

注意：上述代码，初始化程序无法访问 this 关键字。

8.**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 co st 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);
}
```

9.**重定向构造函数**

● 有时候构造函数的目的只是重定向到该类的另一个构造函数。重定向构造函数没有函数体，使用冒号分隔。

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

    // 主构造函数
    Point(this.x, this.y) {
        print("Point($x, $y)");
    }

    // 重定向构造函数，指向主构造函数，函数体为空
    Point.alongXAxis(num x) : this(x, 0);
}

void main() {
    var p1 = new Point(1, 2);
    var p2 = new Point.alongXAxis(4);
}
```

10. **常量构造函数**

● 如果类的对象不会发生变化，可以构造一个编译时的常量构造函数。定义格式如下：

- 定义所有的实例变量是final。
- 使用const声明构造函数。

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

11. **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);
    }
  }

}
```

注意：工厂构造函数不能用 this。

方法

● 方法就是为对象提供行为的函数。

1.**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：

```
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);
  }
}
```

2.**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 et 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);
}
```

1.

● 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。

2. **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...
    }
}
```

4. **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。

● 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 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。
```

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mixis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');
    }
  }
}
```

类的变量和方法

- 使用 static 关键字来实现类变量和类方法。
- 只有当静态变量被使用时才被初始化。
- 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');
}
```

- 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

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 。
- 抽象类通常包含 抽象方法。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();
}
```

类-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 {
  // ...
}
```

类-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
。
  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，那么你可以用 @proxy 标注来避免警告。

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

库和可见性

1. port,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。

2. 库可以分布式使用包。见 Pub Package and Asset Manager 中有关pub(SDK中的一个包管理）。

3. **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，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。对于其库，你可以使用文件系统路径或package:scheme。包 package：scheme specifies libraries，如pb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';
```

4. **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
```

5.**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;
```

6.**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。
- 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 asynchro y 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件导入的库。
    - Dart隐含调用LoadLibrary（）插入到定义deferred as namespace。在调用LoadLibrary（）数返回一个Future。

7.**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。

8. **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...
```

9.**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...
```

10. **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
}
```

异步的支持

1. 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。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（比如 I/O 操作）之后返回，而无需等待操作完成

2. 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

3. 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

4. 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()
```

5. 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.
      }
 }
```

6.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...
 }
```

7. **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**

8. **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：

  ● 在 stream 发出一个值之前等待

● 执行 for 循环的主体，把变量设置为发出的值。

  ● 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]

作者：GuoDongW

链接：https://www.jianshu.com/p/9e5f4c81cc7d

来源：简书