变量

你可以像 JavaScript 那样声明一个变量：

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

编译器会推导出 `name` 的类型是 `String` 类型，等价于：

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

我们可以从下面代码窥见 Dart 是强类型语言的特性：

```
var name = 'Bob';

// 调用 String 的方法
print(name.toLowerCase());

// 编译错误
// name = 1;
```

前面我们说过，Dart 除了具备简洁的特点，而且也可以是非常灵活的，如果你想变换一个变量的类型，你也可以使用 `dynamic` 来声明变量，这就跟 JavaScript 一样了：

```
dynamic name = 'Bob'; //String 类型
name = 1;// int 类型
print(name);
```

上面的代码可以正常编译和运行，但除非你有足够的理由，请不要轻易使用。

`final` 的语义和 Java 的一样，表示该变量是不可变的：

```
// String 可以省略
final String name = 'Bob';

// 编译错误
// name = 'Mary';
```

其中 `String` 可以省略，Dart 编译器足够聪明地知道变量 `name` 的类型。

如果要声明常量，可以使用 `const` 关键词：

```
const PI = '3.14';

class Person{
  static const name = 'KK';
}
```

如果类变量，则需要声明为 `static const` 。

内置类型

数值类型

Dart 内置支持两种数值类型，分别是`int`和`double`，它们的大小都是64位。

```
var x = 1;
// 0x开头为16进制整数
var hex = 0xDEADBEEF;


var y = 1.1;
// 指数形式
var exponents = 1.42e5;
```

需要注意的是，**在Dart中，所有变量值都是一个对象，`int`和`double`类型也不例外**，它们都是`num`类型的子类，这点和`Java`和`JavaScript`都不太一样：

```
// 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');
```

字符串

Dart 字符串使用的是UTF-16编码。

```
var s = '中';
s.codeUnits.forEach((ch) => print(ch));
// 输出为UNICODE值
20013
```

Dart 采用了 JavaScript 中类似模板字符串的概念，可以在字符串通过`${expression}`语法插入变量：

```
var s = "hello";

print('${s}, world!');

//可以简化成:
print('$s, world!');

//调用方法
print('${s.toUpperCase()}, world!');
```

Dart 可以直接通过 `==` 来比较字符串:

```
var s1 = "hello";
var s2 = "HELLO";
assert(s1.toUpperCase() == s2);
```

布尔类型

Dart 布尔类型对应为 `bool` 关键词，它有 `true` 和 `false` 两个值，这点和其他语言区别不大。值得一提的是，在Dart的条件语句 `if` 和 `assert` 表达式里面，它们的值必须是 `bool` 类型，这点和 JavaScript 不同。

```
var s = '';
assert(s.isEmpty);

if(s.isNotEmpty){
// do something
}

//编译错误，在JavaScript常用来判断undefined
if(s){
}
```

Lists

你可以把Dart中的 `List` 对应到 JavaScript 的数组或者 Java 中的 `ArrayList`，但 Dart 的设计更为精巧。

你可以通过类似 JavaScript 一样声明一个数组对象:

```
var list = [];
list.add('Hello');
list.add(1);
```

这里 `List` 容器接受的类型是 `dynamic`，你可以往里面添加任何类型的对象，但如果像这样声明:

```
var iList = [1,2,3];
iList.add(4);

//编译错误 The argument type 'String' can't be assigned to the parameter type 'int'
//iList.add('Hello');
```

那么Dart就会推导出这个 `List` 是个 `List<int>`，从此这个 `List` 就只能接受 `int` 类型数据了，你也可以显式声明 `List` 的类型:

```
var sList = List<String>();

//在Flutter类库中，有许多这样的变量声明:
List<Widget> children = const <Widget>[];
```

上面右边那个 `const` 的意思表示常量数组，在这里你可以理解为一个给 `children` 赋值了一个编译期常量空数组，这样的做法可以很好的节省内存，下面的例子可以让大家更好的理解常量数组的概念：

```
var constList = const <int>[1,2];

constList[0] = 2; //编译通过，  运行错误
constList.add(3); //编译通过，  运行错误
```

Dart2.3 增加了扩展运算符 (spread operator) `...` 和 `...?`，通过下面的例子你很容易就明白它们的用法：

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

如果扩展对象可能是 `null`，可以使用 `...?`：

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

你可以直接在元素内进行判断，决定是否需要某个元素：

```
var promoActive = true;
var nav = [
    'Home',
    'Furniture',
    'Plants',
    promoActive? 'About':'Outlet'
];
```

甚至使用 `for` 来动态添加多个元素：

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

这种动态的能力使得 Flutter 在构建 Widget 树的时候非常方便。

Sets

`Set` 的语意和其他语言的是一样的，都是表示在容器中对象唯一。在Dart中， `Set` 默认是 `LinkedHashSet` 实现，表示元素按添加先后顺序排序。

声明 `Set` 对象：

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

遍历`Set`，遍历除了上面提到的`for...in`，你还可以使用类似 Java 的 lambada 中的`forEach`形式：

```
halogens.add('bromine');
halogens.add('astatine');
halogens.forEach((el) => print(el));
```

输出结果：

```
fluorine
chlorine
bromine
iodine
astatine
```

除了容器的对象唯一特性之外，其他基本和`List`是差不多的。

```
// 添加类型声明:
var elements = <String>{};

var promoActive = true;
// 动态添加元素
final navSet = {'Home', 'Furniture', promoActive? 'About':'Outlet'};
```

Maps

`Map`对象的声明方式保持了 JavaScript 的习惯，Dart 中`Map`的默认实现是`LinkedHashMap`，表示元素按添加先后顺序排序。

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

assert(gifts['first'] == 'partridge');
```

添加一个键值对：

```
gifts['fourth'] = 'calling birds';
```

遍历`Map`：

```
gifts.forEach((key,value) => print('key: $key, value: $value'));
```

函数

在 Dart 中，函数本身也是个对象，它对应的类型是`Function`，这意味着函数可以当做变量的值或者作为一个方法入传参数值。

```
void sayHello(var name){
  print('hello, $name');
}
```

```
void callHello(Function func, var name){
  func(name);
}

void main(){
  // 函数变量
  var helloFuc = sayHello;
  // 调用函数
  helloFuc('Girl');
  // 函数参数
  callHello(helloFuc,'Boy');
}
```

输出：

```
hello, Girl
hello, Boy
```

对于只有一个表达式的简单函数，你还可以通过`=>`让函数变得更加简洁，`=> expr`在这里相当于`{ return expr; }`，我们来看一下下面的语句：

```
String hello(var name ) => 'hello, $name';
```

相当于：

```
String hello(var name ){
  return 'hello, $name';
}
```

参数

在Flutter UI库里面，命名参数随处可见，下面是一个使用了命名参数（Named parameters）的例子：

```
void enableFlags({bool bold, bool hidden}) {...}
```

调用这个函数：

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

命名参数默认是可选的，如果你需要表达该参数必传，可以使用`@required`：

```
void enableFlags({bool bold, @required bool hidden}) {}
```

当然，Dart 对于一般的函数形式也是支持的：

```
void enableFlags(bool bold, bool hidden) {}
```

和命名参数不一样，这种形式的函数的参数默认是都是要传的：

```
enableFlags(false, true);
```

你可以使用`[]`来增加非必填参数：

```
void enableFlags(bool bold, bool hidden, [bool option]) {}
```

另外，Dart 的函数还支持设置参数默认值：

```
void enableFlags({bool bold = false, bool hidden = false}) {...}

String say(String from, [String device = 'carrier pigeon', String mood]) {}
```

匿名函数

顾名思意，匿名函数的意思就是指没有定义函数名的函数。你应该对此不陌生了，我们在遍历`List`和`Map`的时候已经使用过了，通过匿名函数可以进一步精简代码：

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

闭包

Dart支持闭包。没有接触过JavaScript的同学可能对闭包（closure）比较陌生，这里给大家简单解释一下闭包。

闭包的定义比较拗口，我们不去纠结它的具体定义，而是打算通过一个具体的例子去理解它：

```
Function closureFunc() {
  var name = "Flutter"; // name 是一个被 init 创建的局部变量
  void displayName() { // displayName() 是内部函数,一个闭包
    print(name); // 使用了父函数中声明的变量
  }
  return displayName;
}

void main(){
  //myFunc是一个displayName函数
  var myFunc = closureFunc(); // (1)

  // 执行displayName函数
  myFunc(); // (2)
}
```

结果如我们所料的那样打印了`Flutter`。

在（1）执行完之后，`name`作为一个函数的局部变量，引用的对象不是应该被回收掉了吗？但是当我们在内函数调用外部的`name`时，它依然可以神奇地被调用，这是为什么呢？

这是因为Dart在运行内部函数时会形成闭包，**闭包是由函数以及创建该函数的词法环境组合而成，这个环境包含了这个闭包创建时所能访问的所有局部变量**。

我们简单变一下代码：

```
Function closureFunc() {
  var name = "Flutter"; // name 是一个被 init 创建的局部变量
  void displayName() { // displayName() 是内部函数,一个闭包
    print(name); // 使用了父函数中声明的变量
  }
  name = 'Dart'; //重新赋值
  return displayName;
}
```

结果输出是`Dart`，可以看到内部函数访问外部函数的变量时，是在同一个词法环境中的。

返回值

在Dart中，所有的函数都必须有返回值，如果没有的话，那将自动返回`null`：

```
foo() {}

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

流程控制

这部分和大部分语言都一样，在这里简单过一下就行。

if-else

```
if(hasHause && hasCar){
    marry();
}else if(isHandsome){
    date();
}else{
    pass();
}
```

循环

各种`for`：

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

for(var i = 0; i != list.length; i++){}

for(var i in list){}
```

`while`和循环中断（中断也是在for中适用的）：

```
var i = 0;
  while(i != list.length){
    if(i % 2 == 0){
       continue;
    }
     print(list[i]);
```

```
  }

  i = 0;
  do{
    print(list[i]);
    if(i == 5){
      break;
    }
  }while(i != list.length);
```

如果对象是`Iterable`类型，你还可以像Java的 lambada 表达式一样：

```
list.forEach((i) => print(i));

list.where((i) =>i % 2 == 0).forEach((i) => print(i));
```

switch

`switch`可以用于`int`、`double`、`String`和`enum`等类型，`switch`只能在同类型对象中进行比较，进行比较的类不要覆盖`==`运算符。

```
var color = '';
  switch(color){
    case "RED":
      break;
    case "BLUE":
      break;
    default:

  }
```

assert

在Dart中，`assert`语句经常用来检查参数，它的完整表示是：`assert(condition, optionalMessage)`，如果`condition`为`false`，那么将会抛出`[AssertionError]`异常，停止执行程序。

```
assert(text != null);

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

`assert`通常只用于开发阶段，它在产品运行环境中通常会被忽略。在下面的场景中会打开`assert`：

1. Flutter 的`debug mode`。
2. 一些开发工具比如`dartdevc`默认会开启。
3. 一些工具，像`dart`和`dart2js`，可以通过参数`--enable-asserts`开启。

异常处理

Dart 的异常处理和Java很像，但是Dart中所有的异常都是非检查型异常（unchecked exception），也就是说，你不必像 Java 一样，被强制需要处理异常。

Dart 提供了`Exception`和`Error`两种类型的异常。一般情况下，你不应该对`Error`类型错误进行捕获处理，而是尽量避免出现这类错误。

比如`OutOfMemoryError`、`StackOverflowError`、`NoSuchMethodError`等都属于`Error`类型错误。

前面提到，因为 Dart 不像 Java 那样可以声明编译期异常，这种做法可以让代码变得更简洁，但是容易忽略掉异常的处理，所以我们在编码的时候，在可能会有异常的地方要注意阅读API文档，另外自己写的方法，如果有异常抛出，要在注释处进行声明。比如类库中的`File`类其中一个方法注释：

```
/**
   * Synchronously read the entire file contents as a list of bytes.
   *
   * Throws a [FileSystemException] if the operation fails.
   */
  Uint8List readAsBytesSync();
```

抛出异常

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

`throw`除了可以抛出异常对象，它还可以抛出任意类型对象，但建议还是使用标准的异常类作为最佳实践。

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

捕获异常

可以通过`on`关键词来指定异常类型：

```
var file = File("1.txt");
  try{
    file.readAsStringSync();
  } on FileSystemException {
     //do something
  }
```

使用`catch`关键词获取异常对象，`catch`有两个参数，第一个是异常对象，第二个是错误堆栈。

```
try{
    file.readAsStringSync();
} on FileSystemException catch (e){
    print('exception: $e');
} catch(e, s){ //其余类型
   print('Exception details:\n $e');
   print('Stack trace:\n $s');
}
```

使用`rethrow`抛给上一级处理：

```
try{
    file.readAsStringSync();
  } on FileSystemException catch (e){
    print('exception: $e');
  } catch(e){
     rethrow;
  }
```

finally

`finally`一般用于释放资源等一些操作，它表示最后一定会执行的意思，即便`try...catch`中有`return`，它里面的代码也会承诺执行。

```
try{
     print('hello');
     return;
  } catch(e){
     rethrow;
  } finally{
     print('finally');
}
```

输出：

```
hello
finally
```

面向对象

类

Dart 是一门面向对象的编程语言，所有对象都是某个类的实例，所有类继承了`Object`类。

一个简单的类：

```
class Point {
  num x, y;

  // 构造器
  Point(this.x, this.y);

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

类成员

Dart 通过 `.` 来调用类成员变量和方法的。

```
//创建对象, new 关键字可以省略
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));
```

你还可以通过 `.?` 来避免 `null` 对象。在Java 里面，经常需要大量的空判断来避免 `NullPonterException`，这是让人诟病Java的其中一个地方。而在Dart中，可以很方便地避免这个问题：

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

在 Dart 中，没有 `private`、`protected`、`public` 这些关键词，如果要**声明一个变量是私有的，则在变量名前添加下划线 `_`**，声明了私有的变量，只在本类库中可见。

```
class Point{
  num _x;
  num _y;
}
```

构造器（Constructor）

如果没有声明构造器，Dart 会给类生成一个默认的无参构造器，声明一个带参数的构造器，你可以像 Java这样：

```
class Person{
  String name;
  int sex;

  Person(String name, int sex){
    this.name = name;
    this.sex = sex;
  }
}
```

也可以使用简化版：

```
Person(this.name, this.sex);
```

或者命名式构造器：

```
Person.badGirl(){
    this.name = 'Bad Girl';
    this.sex = 1;
}
```

你还可以通过`factory`关键词来创建实例：

```
Person.goodGirl(){
    this.name = 'good Girl';
    this.sex = 1;
}

factory Person(int type){
    return type == 1 ? Person.badGirl(): Person.goodGirl();
}
```

`factory`对应到设计模式中工厂模式的语言级实现，在 Flutter 的类库中有大量的应用，比如`Map`：

```
// 部分代码
abstract class Map<K, V> {
    factory Map.from(Map other) = LinkedHashMap<K, V>.from;
}
```

如果一个对象的创建过程比较复杂，比如需要选择不同的子类实现或则需要缓存实例等，你就可以考虑通过这种方法。在上面`Map`例子中，通过声明`factory`来选择了创建子类`LinkedHashMap`（`LinkedHashMap.from`也是一个`factory`，里面是具体的创建过程）。

如果你想在对象创建之前的时候还想做点什么，比如参数校验，你可以通过下面的方法：

```
Person(this.name, this.sex): assert(sex == 1)
```

在构造器后面添加的一些简单操作叫做**initializer list**。

在Dart中，初始化的顺序如下：

1. 执行initializer list；
2. 执行父类的构造器；
3. 执行子类的构造器。

```
class Person{
  String name;
  int sex;

  Person(this.sex): name = 'a', assert(sex == 1){
    this.name = 'b';
    print('Person');
  }

}

class Man extends Person{
   Man(): super(1){
     this.name = 'c';
     print('Man');
   }
}

void main(){
  Person person = Man();
  print('name : ${person.name}');
```

```
}
```

上面的代码输出为：

```
Person
Man
name : c
```

如果子类构造器没有显式调用父类构造器，那么默认会调用父类的默认无参构造器。显式调用父类的构造器:

```
Man(height): this.height = height, super(1);
```

重定向构造器：

```
Man(this.height, this.age): assert(height > 0), assert(age > 0);

Man.old(): this(12, 60); //调用上面的构造器
```

Getter 和 Setter

在 Dart 中，对 `Getter` 和 `Setter` 方法有专门的优化。即便没有声明，每个类变量也会默认有一个 `get` 方法，在 `隐含接口` 章节会有体现。

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

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

  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);
}
```

抽象类

Dart 的抽象类和Java差不多，除了不可以实例化，可以声明抽象方法之外，和一般类没有区别。

```
abstract class AbstractContainer {
  num _width;

  void updateChildren(); // 抽象方法，强制继承子类实现该方法。

  get width => this._width;

  int sqrt(){
    return _width * _width;
  }
}
```

隐含接口

Dart 中的每个类都隐含了定义了一个接口，这个接口包含了这个类的所有成员变量和方法，你可以通过`implements`关键词来重新实现相关的接口方法：

```
class Person {
  //隐含了 get 方法
  final _name;

  Person(this._name);

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

class Impostor implements Person {
  // 需要重新实现
  get _name => '';

  // 需要重新实现
  String greet(String who) => 'Hi $who. Do you know who I am?';
}
```

实现多个接口：

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

继承

和Java基本一致，继承使用`extends`关键词：

```
class Television {
  void turnOn() {
    doSomthing();
  }
}

class SmartTelevision extends Television {

  @override
  void turnOn() {
    super.turnOn(); //调用父类方法
    doMore();
  }
```

```
}
```

重载操作符

比较特别的是，Dart 还允许重载操作符，比如`List`类支持的下标访问元素，就定义了相关的接口：

```
E operator [](int index);
```

我们通过下面的实例来进一步说明重载操作符：

```
class MyList{

  var list = [1,2,3];

  operator [](int index){
    return list[index];
  }
}

void main() {
  var list = MyList();
  print(list[1]); //输出 2
}
```

扩展方法

这个特性也是Dart让人眼前一亮的地方（Dart2.7之后才支持），可以对标到 JavaScript 中的 prototype。通过这个特性，你甚至可以给类库添加新的方法：

```
//通过关键词 extension 给 String 类添加新方法
extension NumberParsing on String {
  int parseInt() {
    return int.parse(this);
  }
}
```

后面`String`对象就可以调用该方法了：

```
print('42'.parseInt());
```

枚举类型

枚举类型和保持和Java的关键词一致：

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

在`switch`中使用:

```
// color 是 enmu Color 类型
switch(color){
    case Color.red:
      break;
    case Color.blue:
      break;
    case Color.green:
      break;
    default:
      break;
}
```

枚举类型还有一个`index`的getter，它是个连续的数字序列，从0开始：

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

新特性：Mixins

这个特性进一步增强了代码复用的能力，如果你有写过Android的布局XML代码或者Freemaker模板的话，那这个特性就可以理解为其中`inlclude`的功能。

声明一个`mixin`类：

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

通过`with`关键词进行复用：

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

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

`mixin`类甚至可以通过`on`关键词实现继承的功能：

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

类变量和类方法

```
class Queue {
  //类变量
  static int maxLength = 1000;
  // 类常量
  static const initialCapacity = 16;
  // 类方法
  static void modifyMax(int max){
    _maxLength = max;
  }
}

void main() {
  print(Queue.initialCapacity);
  Queue.modifyMax(2);
  print(Queue._maxLength);
}
```

泛型

在面向对象的语言中，泛型主要的作用有两点：

1、类型安全检查，把错误扼杀在编译期：

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

2、增强代码复用，比如下面的代码：

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

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

你可以通过泛型把它们合并成一个类：

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

在Java中，泛型是通过类型擦除来实现的，但在Dart中实打实的泛型：

```
var names = <String>[];
 names.addAll(['Tom',"Cat"]);
 // is 可以用于类型判断
 print(names is List<String>); // true
 print(names is List); // true
 print(names is List<int>); //false
```

你可以通过`extends`关键词来限制泛型类型，这点和Java一样：

```
abstract class Animal{}
class Cat extends Animal{}
class Ext<T extends Animal>{
  T data;
}

void main() {
  var e = Ext(); // ok
  var e1 = Ext<Animal>(); // ok
  var e2 = Ext<Cat>(); // ok
  var e3 = Ext<int>(); // compile error
}
```

使用类库

有生命力的编程语言，它背后都有一个强大的类库，它们可以让我们站在巨人的肩膀上，又免于重新造轮子。

导入类库

在Dart里面，通过`import`关键词来导入类库。

内置的类库使用`dart:`开头引入：

```
import 'dart:io';
```

了解更多内置的类库可以查看**这里**。

第三方类库或者本地的dart文件用`package:`开头：

比如导入用于网络请求的`dio`库：

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

Dart 应用本身就是一个库，比如我的应用名是`ccsys`，导入其他文件夹的类：

```
import 'package:ccsys/common/net_utils.dart';
import 'package:ccsys/model/user.dart';
```

如果你使用IDE来开发，一般这个事情不用你来操心，它会自动帮你导入的。

Dart 通过**pub.dev**来管理类库，类似Java世界的`Maven`或者Node.js的npm一样，你可以在里面找到非常多实用的库。

解决类名冲突

如果导入的类库有类名冲突，可以通过`as`使用别名来避免这个问题：

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

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

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

导入部分类

在一个dart文件中，可能会存在很多个类，如果你只想引用其中几个，你可以增加`show`或者`hide`来处理：

```
//文件: my_lib.dart
class One {}

class Two{}

class Three{}
```

使用`show`导入`One`和`Two`类：

```
//文件: test.dart
import 'my_lib.dart' show One, Two;

void main() {
  var one = One();
  var two = Two();
  //compile error
  var three = Three();
}
```

也可以使用`hide`排除`Three`，和上面是等价的：

```
//文件: test.dart
import 'my_lib.dart' hide Three;

void main() {
  var one = One();
  var two = Two();
}
```

延迟加载库

目前只有在web app（dart2js）中才支持延迟加载，Flutter、Dart VM是不支持的，我们这里仅做一下简单介绍。

你需要通过`deferred as`来声明延迟加载该类库：

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

当你需要使用的时候，通过`loadLibrary()`加载：

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

你可以多次调用`loadLibrary`，它不会被重复加载。