环境搭建

- SDK 下载和配置环境变量
  - https://dart.dev/get-dart
  - https://www.gekorm.com/dart-windows
- 安装 Dart 插件

  打开 IDEA, File -> Settings -> Plugins, 搜索 Dart 点击安装后重新启动 IDEA 后，点击新建 Dart 项目，并设置已经安装的 Dart SDK 的路径。

Dart 关键字

参考链接：https://dart.dev/guides/language/language-tour

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

> 避免使用这些单词作为标识符。但是，如有必要，标有上标的关键字可以用作标识符：
>
> - 带有 **1** 上标的单词为 **上下文关键字**，仅在特定位置具有含义。他们在任何地方都是有效的标识符。
> - 带有 **2** 上标的单词为 **内置标识符**，为了简化将 JavaScript 代码移植到 Dart 的工作，这些关键字在大多数地方都是有效的标识符，但它们不能用作类或类型名称，也不能用作 import 前缀。
> - 带有 **3** 上标的单词是与 Dart 1.0 发布后添加的异步支持相关的更新，作为限制类保留字。不能在标记为 async ， async* 或 sync* 的任何函数体中使用 await 或 yield 作为标识符。
>
> 关键字表中的剩余单词都是**保留字**。 不能将保留字用作标识符。

基础语法

- 变量声明

```
var name = "Simplation";
var year = 2020;
var antennaDiameter = 3.1415926;
var flybyObjects = ["Jupyter", "Simplation", "Name", true];
var imageDict = {"name": "Simplation", "age": 18};
```

- 变量

  Dart 代码是类型安全的，支持类型推断，因此大多数变量不需要显式指定类型。

- 默认值

  未初始化的任何类型的变量默认值都是 null。

- final 和 const

使用过程中从来不会被修改的变量可以使用 final 或 const 修饰，final 变量的值只能被设置一次；const 变量在编译时就已经固定最高级 final 变量或类变量在第一次使用时被初始化。

```
final name = "Simplation";
final age = 18;

name = "Simplation.WANG";  // Error：一个 final 变量只能被设置一次

const pressure = 100000;
const double atm = 1.01325 * pressure;

pressure = 50000;   // Error: 常量变量不能赋值修改。
```

数据类型

Dart 语言支持以下内建类型：Number、String、Boolean、List、Map、Set、Rune、Symbol

- Number

  int 和 double 都是 num 的亚类型。num 类型包括基本运算 +， -， /， 和 *， 以及 abs()、ceil() 和 floor() 等函数方法。 （按位运算符，例如»，定义在 int 类中。）如果 num 及其亚类型找不到你想要的方法，尝试查找使用 dart:math 库。

- int

  整数值不大于64位， 具体取决于平台。 在 Dart VM 上，值的范围从 -263 到 263 - 1. Dart 被编译为 JavaScript 时，使用 JavaScript numbers, 值的范围从 -253 到 253 - 1.

- Double

  64位（双精度）浮点数，依据 IEEE 754 标准。

```
// 定义整数类形
var x = 1;
var hex = 0xD82940;

// 定义 double 类型
var y = 3.14;
var exponents = 1.42e5;

// 从 Dart 2.1 开始，必要的时候 int 字面量会自动转换成 double 类型。
double z = 1; // 相当于 double z = 1.0.

print("------------------- 分割线 ----------------------");

// 数据类型转换
// String -> int
var num = int.parse("12306");
assert(num == 12306);

// String -> double
var dob = double.parse("3.1415926");
assert(dob == 3.1415926);

// int -> String
var int2str = 18.toString();
assert(int2str == "18");

// double -> String
```

```
var dob2str = 3.1415.toString();
assert(dob2str == "3.1415");

// 按位操作:移位（<<， >>），按位与（&）以及 按位或（|）
assert((3 << 1) == 6); // 0011 << 1 == 0110
assert((3 >> 1) == 1); // 0011 >> 1 == 0001
assert((3 | 4) == 7); // 0011 | 0100 == 0111
```

- String

  Dart 字符串是一组 UTF-16 单元序列。字符串通过单引号或者双引号创建。字符串可以通过 ${*expression*} 的方式内嵌表达式。

```
// 字符串创建
var s1 = "create s1 string";
var s2 = "create s2 string";
var s3 = "create s3 string";

var s = "Hello Dart, test";
assert("create s1 string $s" == s1 + " Hello Dart, test");

// 字符串拼接
var str = s1 + s2 + s3;
print(str);

// 创建多行字符串
var multipleStr = """
 三里清风三里路，
 步步清风再无你。
""";
print(multipleStr);

// 创建原始字符串：使用 r 作为前缀
var strs = "http://www.baidu. \ncom";
var originalStr = r"http://www.baidu. \ncom";
print(strs);
print(originalStr);
```

- Boolean

  Dart 使用 bool 类型表示布尔值。Dart 只有字面量 true and false 是布尔类型，这两个对象都是编译时常量。

```
// 检查空字符串/非空字符串
var emptyStr = "";
assert(emptyStr.isEmpty);

// 判断 0 值
var hintPoint = 0;
assert(hintPoint <= 0);

// 检查 null 值
var unicon;
assert(unicon == null);

// 检查 NaN
var iMeantToDoThis = 0 / 0;
```

```
assert(iMeantToDoThis.isNaN);
```

- List

  Dart 中的 *Array*就是 List 对象，通常称之为 *List* 。

```
// 创建列表
var list = [1, 2, 3, 4, 5];
print(list);

// 获取 list 的长度
print(list.length);

// 访问列表的元素
print(list[1] == 2);

// 在 List 字面量之前添加 const 关键字，可以定义 List 类型的编译时常量
var constantList = const [1, 2, 3, true];
// constantList[1] = 1; // 取消注释会引起错误。

// 添加元素
list.insert(5, 6);
list.add(7);
print(list);

// 删除元素
list.removeAt(6);
list.remove(6);
print(list);

// 清除元素
list.clear();
print(list);
```

- Map

  通常来说，Map 是用来关联 keys 和 values 的对象。keys 和 values 可以是任何类型的对象。在一个 Map 对象中一个 *key*只能出现一次。但是 *value*可以出现多次。Dart 中 Map 通过 Map 字面量和 Map 类型来实现。

```
// 创建泛型为<String, String>
var gifts = {
 // Key: Value
 "first": "Apple",
 "second": "ring",
 "third": "videos"
};

// 创建泛型为<int, String>
var nodeCase = {0: "apple", 1: "banana", 2: "car", 3: "dot"};

// 添加
var gift = Map(); // 等价于 var gift = new Map();
gift['first'] = 'Apple'; // add key - value
gift['second'] = 'Banana';
gift['third'] = 'Car';
```

```
var node = Map();
node[0] = 'apple';
node[1] = 'banana';
node[2] = 'car';
node[3] = 'dot';

assert(gift['first'] == 'Apple');

// 当 Map 中不存在 key 的时候会返回 null
assert(gift['fourth'] == null);

// 获取 Map 的长度
print(gift.length);

// 创建 Map 类型运行时常量，要在 Map 字面量前加上关键字 const。
final constantMap = const {
 2: 'helium',
 10: 'neon',
 18: 'argon',
};
// constantMap[2] = 'Helium'; // Error
```

- Set

  在 Dart 中 Set 是一个元素唯一且无需的集合。Dart 为 Set 提供了 Set 字面量和 Set 类型。

```
// 创建
var createSet = {"one", "two", "three", "four"}; // 自动推断泛型为 String
var nullSet = <String>{}; // 创建空的 Set 对象
// Set<String> names = {}; // 这样也是可以的。
// {} 默认是 Map 类型
// var names = {}; // 这样会创建一个 Map ，而不是 Set 。
print(createSet);

// Set 的相关操作
// 添加
var elements = <String>{};
elements.add("Apple");
elements.addAll(createSet);
print(elements);

// 获取 Set 长度
print(elements.length);

// 创建一个编译时 Set 常量
final constantSet = const {
 'Apple',
 'Banana',
 'Carrot',
 'Dot',
 'Earth',
};
// constantSet.add('Dot'); // Uncommenting this causes an error.
print(constantSet);
```

- Renu

  在字符串中标识 Unicode 字符

在 Dart 中， Rune 用来表示字符串中的 UTF-32 编码字符。Unicode 定义了一个全球的书写系统编码，系统中使用的所有字母，数字和符号都对应唯一的数值编码。由于 Dart 字符串是一系列 UTF-16 编码单元，因此要在字符串中表示 32 位 Unicode 值需要特殊语法支持。表示 Unicode 编码的常用方法是， \uXXXX, 这里 XXXX 是一个 4 位的 16 进制数。

```
// Rune 格式：\u{xxxx}
var simle = "\u{1f600}";
var heartShape = "\u{2665}";

print(simle + " -- " + heartShape);   // -- ♥
```

- Symbol

  一个 Symbol 对象表示 Dart 程序中声明的运算符或者标识符。你也许永远都不需要使用 Symbol，但要按名称引用标识符的 API 时， Symbol 就非常有用了。因为代码压缩后会改变标识符的名称，但不会改变标识符的符号。通过字面量 Symbol，也就是标识符前面添加一个 # 号，来获取标识符的 Symbol 。

```
var radix = #radix;
var bar = #bar;

print(radix);   // Symbol("radix")
print(bar);     // Symbol("bar")
```

函数

Dart 是一门真正面向对象的语言，甚至其中的函数也是对象，并且有它的类型 Function。这也意味着函数可以被赋值给变量或者作为参数传递给其他函数。也可以把 Dart 类的实例当做方法来调用。

- 可选参数

  可选参数可以是命名参数或者位置参数，但一个参数只能选择其中一种方式修饰。

  - 命名可选参数

```
void main() {
  getNumber(20);
}

// => expr 语法是 { return expr; } 的简写。 => 符号 有时也被称为 箭头 语法。
int getNumber(int number) {
  return number + 5;
} // 等价于 int getNumber(int number) => number + 5;

void getParams({bold: true, hidden: false}) {}

// 使用 @required 注释表示参数是 required 性质的命名参数， 该方式可以在任何 Dart 代码中使用。
```

  - 位置可选参数

    将参数放到 [] 中来标记参数是可选的。

```
void main() {
  // 未传递可选参数
  assert(say("Simplation", "Hello Dart") == "Simplation says Hello Dart");
```

```
  // 传递可选参数
  assert(say("Simplation.WANG", "Hello Dart!", "Say Hello") ==
"Simplation.WANG says Hello Dart! with a Say Hello");
}

// 定义可选参数方法
String say(String from, String msg, [String device]) {
  var result = "$from says $msg";
  if (device != null) {
    result = "$result with a $device";
  }
  return result;
}
```

- 默认参数值

  在定义方法的时候，可以使用 = 来定义可选参数的默认值。默认值只能是编译时常量。如果没有提供默认值，则默认值为 null。

```
void main() {
  enableFlags(bold: true);

  assert(say("Simplation", "哈哈哈...") == "Simplation says 哈哈哈... with
a Mobile Phone");

  // 调用方法
  doStuff();
}

/// 设置 [bold] 和 [hidden] 标志 ...
void enableFlags({bool bold = false, bool hidden = false}) {
}

// 示例
String say(String from, String msg,
    [String device = "Mobile Phone", String mode]) {
  var result = "$from says $msg";

  if (device != null) {
    result = "$result with a $device";
  }

  if (mode != null) {
    result = "$result (in a $mode mode)";
  }

  return result;
}

// list 或 map 可以作为默认值传递。
void doStuff(
    {List<int> list = const [1, 2, 3],
    Map<String, String> gifts = const {
      "first": "apple",
      "second": "banana",
      "third": "carrot"
```

```
    }}) {
  print("list: $list");
  print("gifts: $gifts");
}
```

- main 函数

  任何应用都必须有一个顶级 main() 函数，作为应用服务的入口 main() 函数返回值为空，参数为一个可选的 List 。

```
void main() {
  getNumber(20);

  // 未传递可选参数
  // assert(say("Simplation", "Hello Dart") == "Simplation says Hello
Dart");

  // 传递可选参数
  // assert(say("Simplation.WANG", "Hello Dart!", "Say Hello") ==
  //    "Simplation.WANG says Hello Dart! with a Say Hello");

  enableFlags(bold: true);

  assert(say("Simplation", "哈哈哈...") ==
      "Simplation says 哈哈哈... with a Mobile Phone");

  doStuff();

  print(arguments);
}

// => expr 语法是 { return expr; } 的简写。 => 符号 有时也被称为 箭头 语法。
int getNumber(int number) {
  return number + 5;
} // 等价于 int getNumber(int number) => number + 5;

void getParams({bold: true, hidden: false}) {}

// 使用 @required 注释表示参数是 required 性质的命名参数， 该方式可以在任何 Dart 代码
中使用。

/// 设置 [bold] 和 [hidden] 标志 ...
void enableFlags({bool bold = false, bool hidden = false}) {}


// list 或 map 可以作为默认值传递。
void doStuff(
    {List<int> list = const [1, 2, 3],
    Map<String, String> gifts = const {
      "first": "apple",
      "second": "banana",
      "third": "carrot"
    }}) {
  print("list: $list");
  print("gifts: $gifts");
}
```

```
String say(String from, String msg,
    [String device = "Mobile Phone", String mode]) {
  var result = "$from says $msg";

  if (device != null) {
    result = "$result with a $device";
  }

  if (mode != null) {
    result = "$result (in a $mode mode)";
  }

  return result;
}
```

- 函数是一等对象

```
void main() {
 var list = [1, 2, 3];
 // 进行遍历操作
 list.forEach(printElement);

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

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

- 匿名函数

  匿名函数和命名函数看起来类似— 在括号之间可以定义一些参数或可选参数，参数使用逗号分割。

```
// 匿名函数格式
([[*Type*] *param1*[, …]]) {
*codeBlock*;
};

// 示例
void main() {
 var fruits = ["apple", "banana", "orange"];
 fruits.forEach((item) {
   print("${fruits.indexOf(item)}:$item");
});

 // 等价于 fruits.forEach((item) =>
print("${fruits.indexOf(item)}:$item"));
}
```

- 词法作用域

  Dart 是一门词法作用域的编程语言，就意味着变量的作用域是固定的，简单说变量的作用域在编写代码的时候就已经确定了。花括号内的是变量可见的作用域。

```
```

```
bool topLevel = true;

void main() {
 var insideMain = true;

 void myFunction() {
   var insideFunction = true;

   void nestedFunction() {
     var insideNestedFunction = true;

     assert(topLevel);
     assert(insideMain);
     assert(insideFunction);
     assert(insideNestedFunction);
   }
}
}
// 注意 nestedFunction() 可以访问所有的变量， 一直到顶级作用域变量。
```

- 语法闭包

  闭包即一个函数对象，即使函数对象的调用在它原始作用域之外，依然能够访问在它词法作用域内的变量。

```
void main() {
 var add2 = makeAdd(2);
 var add4 = makeAdd(4);

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

// 返回一个函数，返回的函数参数与 [addBy] 相加。
Function makeAdd(num addBy) {
 return (num i) => addBy + i;
}
```

- 测试函数是否相等

```
// 顶级函数，静态方法和示例方法相等性的测试示例：
void foo() {} // 顶级函数

class A {
 static void bar() {} // 静态方法
 void baz() {} // 示例方法
}

void main() {
 var x;

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

 // 比较静态方法。
 x = A.bar;
```

```
  assert(A.bar == x);

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

  // 两个闭包引用的同一实例（2号），
  // 所以它们相等。
  assert(y.baz == x);

  // 两个闭包引用的非同一个实例，
  // 所以它们不相等。
  assert(v.baz != w.baz);
}
```

- 返回值

  所有的函数都会返回一个值，如果没有明确的返回值，函数体会被隐式的添加 *return null*；语句

运算符

Dart 定义的运算符表，优先级从上到下依次减弱。

Description	Operator
unary postfix	*expr*++ *expr*-- () [] . ?.
unary prefix	-*expr* !*expr* ~*expr* ++*expr* --*expr*
multiplicative	* / % ~/
additive	+ -
shift	<< >> >>>
bitwise AND	&
bitwise XOR	^
bitwise OR	|
relational and type test	>= > <= < as is is!
equality	== !=
logical AND	&&
logical OR	||
if null	??
conditional	*expr1* ? *expr2* : *expr3*
cascade	..
assignment	= *= /= += -= &= ^= *etc.*

- 算术运算符
  - Dart 常用运算符： +、-、*、/、~/、%

```
// 示例
void main() {
 assert(2 + 3 == 5);
 assert(2 - 3 == -1);
 assert(2 * 3 == 6);
 assert(5 / 2 == 2.5); // 结果是双浮点型
 assert(5 ~/ 2 == 2); // 结果是整型
 assert(5 % 2 == 1); // 余数

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

  - Dart 的前缀和后缀，自增和自减运算符。*++var*、*var++*、*--var*、*var--*

```
// 示例
void main() {
  var a, b;

  a = 0;
  b = ++a; // a自加后赋值给b。
  assert(a == b); // 1 == 1

  a = 0;
  b = a++; // a先赋值给b后，a自加。
  assert(a != b); // 1 != 0

  a = 0;
  b = --a; // a自减后赋值给b。
  assert(a == b); // -1 == -1

  a = 0;
  b = a--; // a先赋值给b后，a自减。
  assert(a != b); // -1 != 0
}
```

- 关系运算符

  Dart 中的关系运算符：==、!=、>、<、>=、<=

  要测试两个对象 x 和 y 是否表示相同的事物，使用 == 运算符。(在极少数情况下，要确定两个对象是否完全相同，需要使用 identical() 函数) 下面给出 == 运算符的工作原理：

  - 如果 *x* 或 *y* 可以 null，都为 null 时返回 true，其中一个为 null 时返回 false。
  - 结果为函数 *x*.==(*y*) 的返回值。(如上所见, == 运算符执行的是第一个运算符的函数。我们甚至可以重写很多运算符，包括 ==，运算符的重写，参考 重写运算符)

```
// 示例
void main() {
  assert(2 == 2);
  assert(2 != 3);
  assert(3 > 2);
  assert(2 < 3);
  assert(3 >= 3);
  assert(2 <= 3);
}
```

- 类型判定运算符

  类型判定运算符包括 as，is，和 is! 主要用于在运算时处理类型检查。

  使用 as 运算符将对象强制转换为特定类型。

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

- 赋值运算符

  使用 = 为变量赋值。使用 ??= 运算符时，只有当被赋值的变量为 null 时才会赋值给它。

```
void main() {
  // 将值赋值给变量a
  a = value;

  // 如果b为空时，将变量赋值给b，否则，b的值保持不变。
  b ??= value;

  var a = 3;
  a *= 3; // 等价于 a = a * 3;
  assert(a == 9);
}
```

- 逻辑运算符

  &&、||、!*expr*

- 按位和移位运算符

  &、|、^、~*expr*、<<、>>

```
// 示例
void main() {
  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，有时可以替换 if-else 表达式，让表达式更简洁：

  - *condition* ? *expr1*: *expr2*

    如果条件为 true, 执行 *expr1*(并返回它的值)：否则，执行并返回 *expr2*的值。

  - *expr1***??** *expr2*

    如果 *expr1*是 non-null，返回 *expr1*的值；否则，执行并返回 *expr2*的值。

- 级联运算符

  级联运算符 (..) 可以实现对同一个对像进行一系列的操作。除了调用函数，还可以访问同一对象上的字段属性。

  > ..并不是一个运算符，而是 Dart 的特殊语法。

- 其它运算符

控制流程语句

- if and else

  Dart 支持 if-else 语句，其中 else 是可选的

```
// 示例
void main() {
  var weather = "raining";

  if (weather == "raining") {
    print("you can bring umbrella");
  } else {
    print("you don't bring umbrella, weather is not rain");
  }
}
```

- for loops

  进行迭代操作，可以使用标准 for 语句。

```
void main() {
  var sf = StringBuffer("Hello Dart");
  for (int i = 0; i < 5; i++) {
    sf.write("!");
  }
  print(sf); // Hello Dart!!!!!
}
```

- while and do-while loops
  - while：循环在执行前判断执行条件

```
void main() {
  int i = 0;
  while (i < 10) {
    i ++;
    print("先判断条件再执行循环后的结果: $i");
  }
}
```

  - do-while：循环在执行后判断执行条件

```
void main() {
  int i = 0;
  do {
    i ++;
    print("先执行再判断条件的结果：$i");
  } while (i < 10);
}
```

- break and continue
  - break：终止程序的循环操作

```
void main() {
  int i = 0;
  do {
    i ++;
    print("先执行再判断条件的结果：$i");
    break;
  } while (i < 10);
}
// 先执行再判断条件的结果：1
```

  - continue：终止本次循环操作，进行下一次循环操作

```
void main() {
  int i = 0;
  while (i < 10) {
    if (i == 3) {
      continue;
    }
    i++;
    print("先判断条件再执行循环后的结果 $i");
  }
}
```

- switch and case

  在 Dart 中 switch 语句使用 == 比较整数，字符串，或者编译时常量。比较的对象必须都是同一个类的实例（并且不可以是子类），类必须没有对 == 重写。枚举类型 可以用于 switch 语句。在 case 语句中，每个非空的 case 语句结尾需要跟一个 break 语句。除 break 以外，还有可以使用 continue, throw，或者 return。

  - 当没有匹配到 case 语句时，执行 default 语句
  - 当 case 语句中缺少 break 语句时，会导致程序报错
  - 支持空 case 语句，允许程序以 fall-through 的形式执行

```
void main() {
  String name;
  // String name = "Apple";
  switch (name) {
    case "Apple":
      print("name is $name");
      break;

    case "Banana":
      print("name is $name");
      break;

    default:
      print("name is null.");
      break;
  }
}
```

- assert

  如果 assert 语句中的布尔条件为 false，那么正常的程序执行流程会被中断。如果正常执行，控制台不会有任何输出；反之，则会出现 Error。

```
void main() {
  int number = 9;
  String name;
  String urlString = "https:www.baidu.com";

  // 确定变量 number 小于 10
  assert(number < 10);

  // 确定变量 name 为空
  assert(name == null);

  // 确定 url 是否是 https
```

```
  assert(urlString.startsWith("https"));
}
```

```
<br>

## 异常

Dart 代码可以抛出和捕获异常。 异常表示一些未知的错误情况。如果异常没有被捕获，则异常会抛出，导致抛出异常的代码终止执行。和 Java 有所不同，Dart 中的所有异常是非检查异常。方法不会声明它们抛出的异常，也不要求捕获任何异常。

Dart 提供了 [Exception](https://api.dartlang.org/stable/dart-core/Exception-class.html) 和 [Error](https://api.dartlang.org/stable/dart-core/Error-class.html) 类型 以及一些子类型。也可以定义自己的异常类型。

- throw：抛出异常

  throw xxxx（xxx 可以是 Dart 已定义的异常，也可以是自己定义的异常 ）

- catch：捕获异常

  捕获语句中可以同时使用 on 和 catch，也可以单独分开使用。使用 on 来指定异常类型，使用 catch 来捕获异常对象。catch() 函数可以指定 1 到 2 个参数，第一个参数为抛出的异常对象，第二个为堆栈信息(一个 [StackTrace](https://api.dartlang.org/stable/dart-core/StackTrace-class.html) 对象)。

  ```dart
  // 示例：伪代码仅用于展示捕获异常的操作
  void main() {
    try {
      breedMoreLlamas();
    } on OutOfLlamasException {
      // 一个特殊的异常
      buyMoreLlamas();
    } on Exception catch (e) {
      // 其他任何异常
      print('Unknown exception: $e');
    } catch (e) {
      // 没有指定的类型，处理所有异常
      print('Something really unknown: $e');
    }
  }
```
```

- finally：无论是抛出或者是补货异常，finally 中的代码都会被执行。

```
// 格式
try {
  // 方法
} catch (e) {
  // 要捕获的异常
} finally {
  // 最终都会被执行的代码
}
```

类

Dart 是一种基于类和 `mixin` 继承机制的面向对象的语言。每个对象都是一个类的实例，所有的类都继承于 Object。基于 * Mixin 继承* 意味着每个类（除 Object 外）都只有一个超类，一个类中的代码可以在其他多个继承类中重复使用。

泛型

<...> 符号将 List 标记为 泛型(或参数化)类型。这种类型具有形式化的参数。通常情况下，使用一个字母来代表类型参数，例如 E, T, S, K, 和 V 等。

- 使用泛型的原因：
  - 保证代码正常运行
  - 正确指定泛型类型可以提高代码质量
  - 使用泛型可以减少重复代码

```
void main() {
  var names = List<String>();
  names.addAll(["Simplation", "Simplation.WANG"]);
  // names.add(18);  // Error, names 只能添加 String 类型的变量
}

// 减少代码量
abstract class ObjectCache {
  Object getByKey(String key);
  void setByKey(String key, Object value);
}
```

- 使用集合字面量

  List, Set 和 Map 字面量也是可以参数化的。

```
// List 或 Set 在声明语句前加 <type> 前缀
var names = <String> ['Simplation', 'Simplation.WANG', 'Sara'];
var uniqueNames = <String> {'Simplation', 'Simplation.WANG', 'Sara'};

// Map 在声明语句前加 <keyType, valueType> 前缀
var pages = <String, String> {
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

- 使用泛型类型构造函数

  在调用构造函数的时，在类名字后面使用尖括号（<...>）来指定泛型类型。

```
var names = <String> ['Simplation', 'Simplation.WANG', 'Sara'];

var nameSet = Set<String>.from(names);
var views = Map<int, View>(); // 创建了一个 key 为 integer， value 为 View 的 map 对象
```

- 运行时中的泛型

  Dart 中泛型类型是固化的，也就是说它们在运行时是携带着类型信息的。

```
void main() {
 var names = List<String>();
 names.addAll(['Simplation', 'Simplation.WANG', 'Sara']);
 print(names is List<String>); // True
}
```

- 限制泛型类型

  使用 extends 实现参数类型的限制。

```
void main() {
 // 指定非 BaseClass 类别的就会报错
 // var food = Food<Object>(); // Error
 var baseClassFoo = Food<BaseClass>();
 var extenderFood = Food<Extender>();
}

// 定义父类
class BaseClass {
}

// 让子类继承父类
class Food<T extends BaseClass> {
 // Implementation goes here...
 String toString() => "Instance of 'Foo<$T>'";
}

class Extender extends BaseClass {
}
```

- 使用泛型函数

  Dart 的泛型最初只能用于类。新语法泛型方法，允许在方法和函数上使用类型参数。使用泛型函数

```
T first<T>(List<T> ts) {
 T tmp = ts[0];
 return tmp;
}
```

库和可见性

import 和 library 指令可以用来创建一个模块化的，可共享的代码库。库不仅提供了 API，而且对代码起到了封装的作用：以下划线 (_) 开头的标识符仅在库内可见。每个 *Dart* 应用程序都是一个库，虽然没有使用 library 指令。

- 使用库

  通过 import 指定一个库命名空间中的内如如何在另一个库中使用。格式：import "xxx";

  - 指定库前缀

    如果导入两个存在冲突标识符的库，则可以为这两个库，或者其中一个指定前缀。

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

// 使用 lib1 中的 Element。
Element element1 = Element();

// 使用 lib2 中的 Element。
lib2.Element element2 = lib2.Element();
```

- 导入库的一部分

  针对需要，导入库的一部分即可。

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

// 导入除 foo 以外的内容
import 'package:lib2/lib2.dart' hide foo;
```

- 延迟加载库

  Deferred loading (也称之为 *lazy loading*) 可以让应用在需要的时候再加载库。需要延迟加载库，使用：deferred as 来导入。需要使用延迟加载库的场景：减少 APP 的启动时间；执行 A/B 测试，例如尝试各种算法的不同实现；加载很少使用的功能，例如可选的屏幕和对话框。

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

// 当需要使用的时候，使用库标识符调用 loadLibrary() 函数来加载库
Future greet() async {
 await hello.loadLibrary();
 hello.printGreeting();
}
```

> 注意事项：
>
> - 延迟加载库的常量在导入的时候是不可用的。只有当库加载完毕的时候，库中常量才可以使用。
> - 在导入文件的时候无法使用延迟库中的类型。如果你需要使用类型，则考虑把接口类型移动到另外一个库中，让两个库都分别导入这个接口库。
> - Dart 隐含的把 loadLibrary() 函数导入到使用 deferred as 的命名空间中。loadLibrary() 方法返回一个 Future。

- 实现库

  实现库参考：Create Library Packages

异步支持

Dart 库中包含许多返回 Future 或 Stream 对象的函数. 这些函数在设置完耗时任务（例如 I/O 操作）后，就立即返回，不会等待耗任务完成。 使用 async 和 await 关键字实现异步编程。

- 处理 Future

  使用 async 和 await 关键字的代码是异步的；要使用 await，代码必须在异步函数（使用 async 标记的函数）中；使用 try，catch，和 finally 来处理代码中使用 await 导致的错误；在一个异步函数中可以多次使用 await 。

如果在使用 await 导致编译时错误，确认 await 是否在一个异步函数中。

  - 使用 async 和 await.
  - 使用 Future API，具体描述请参考库概览.

- 声明异步函数

  函数体被 async 标示符标记的函数，即是一个异步函数。将 async 关键字添加到函数使其返回 Future。

```
String getVersion() => "1.0.0";

// 异步函数，返回值是 Future 。
Future<String> getVersions() async => "1.0.0";
```

- 处理 Stream

  从 Stream 中获取数据，有两种方式：使用 async 和一个异步循环（await for）；使用 Stream API。使用 break 或者 return 语句可以停止接收 stream 的数据，这样就跳出了 for 循环，并且从 stream 上取消注册。

生成器

如果需要延迟生成（lazily produce）一系列值时，可以考虑使用生成器函数，Dart 内置支持两种生成器函数：

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

```
// 都使用 yield 语句来传递值

// 通过在函数体标记 sync*， 可以实现一个同步生成器函数。
Iterable<int> naturalsTo(int n) sync* {
 int k = 0;
 while (k < n) yield k++;
}


// 通过在函数体标记 async*， 可以实现一个异步生成器函数。
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* 来提高其性能
Iterable<int> naturalsDownFrom(int n) sync* {
 if (n > 0) {
   yield n;
   yield* naturalsDownFrom(n -1);
}
}
```

可调用类

通过实现类的 call() 方法，能够让类像函数一样被调用。更多用法参考：Emulating Functions in Dart。

```
void main() {
 var add = AdditionFunction();
 assert(add(4, 5) == 9);
}

class AdditionFunction {
 int call(int a, int b) => a + b;
}
```

Isolates

Dart 代码都在隔离区(isolates)内运行，而不是线程。每个隔离区都有自己的内存堆，确保每个隔离区的状态都不会被其他隔离区访问。

参考链接：dart:isolate library documentation 。

Typedefs

在 Dart 中，函数也是对象，就想字符和数字对象一样。使用 *typedef*，或者 *function-type alias*为函数起一个别名，别名可以用来声明字段及返回值类型。当函数类型分配给变量时，typedef 会保留类型信息。

```
/*class SortedCollection {
  Function compare;

  SortedCollection(int f(Object a, Object b)) {
    // 当把 f 赋值给 compare 的时候，类型信息丢失了。 f 的类型是 (Object, Object) → int (这里 → 代表返回值类型)， 但是 compare 得到的类型是 Function 。
    compare = f;
  }
}

int sort(Object a, Object b) => 0;

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

  assert(sortedCollection.compare is Function);
}*/


/*// typedefs 只能使用在函数类型上
typedef Compare = int Function(Object a, Object b);

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

int sort(Object a, Object b) => 0;

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

```
}*/


// 判断任意函数的类型。
typedef Compare<T> = int Function(T a, T b);

int sort(int a, int b) => 0;

void main() {
  assert(sort is Compare<int>);
}
```

元数据

使用元数据可以提供有关代码的其他信息。元数据注释以字符 @ 开头， =后跟对编译时常量 (如 deprecated) 的引用或对常量构造函数的调用。对于所有 Dart 代码有两种可用注解：@deprecated 和 @override。关于 @override 的使用， 参考 扩展类（继承）。

元数据可以在 library、class、typedef、type parameter、constructor、factory、function、field、parameter 或者 variable 声明之前使用，也可以在 import 或者 export 指令之前使用。使用反射可以在运行时获取元数据信息。

```
library todo;

class Todo {
 final String who;
 final String what;

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

print("------------ 分割线 ------------")

// 导入
import "todo.dart"

// 程序入口
void main() {
 doSomething();
}

// 使用
@Todo("Simplation", "play basketball...")
void doSomething() {
 print("play tennis...");
}
```

注释

Dart 支持单行注释、多行注释和文档注释。

- 单行注释：单行注释以 // 开始。
- 多行注释：多行注释以 /* 开始，以 */ 结尾。多行注释可以嵌套。
- 文档注释：文档注释可以是多行注释，也可以是单行注释，文档注释以 /// 或者 /** 开始。在连续行上使用 /// 与多行文档注释具有相同的效果。使用 [xxx] 会生成一个链接指向 xxx 的 API 文档。