Dart语言

一、Dart简介

1.Dart介绍

> Dart是由谷歌开发的计算机编程语言,它可以被用于web、服务器、移动应用 和物联网等领域的开发。
> Dart诞生于2011年，号称要取代JavaScript。但是过去的几年中一直不温不火。直到Flutter的出现现在被人们重新重视。

要学Flutter的话我们必须首先得会Dart。

官网： https://dart.dev/

2. Dart环境搭建

要在我们本地开发Dart程序的话首先需要安装Dart Sdk

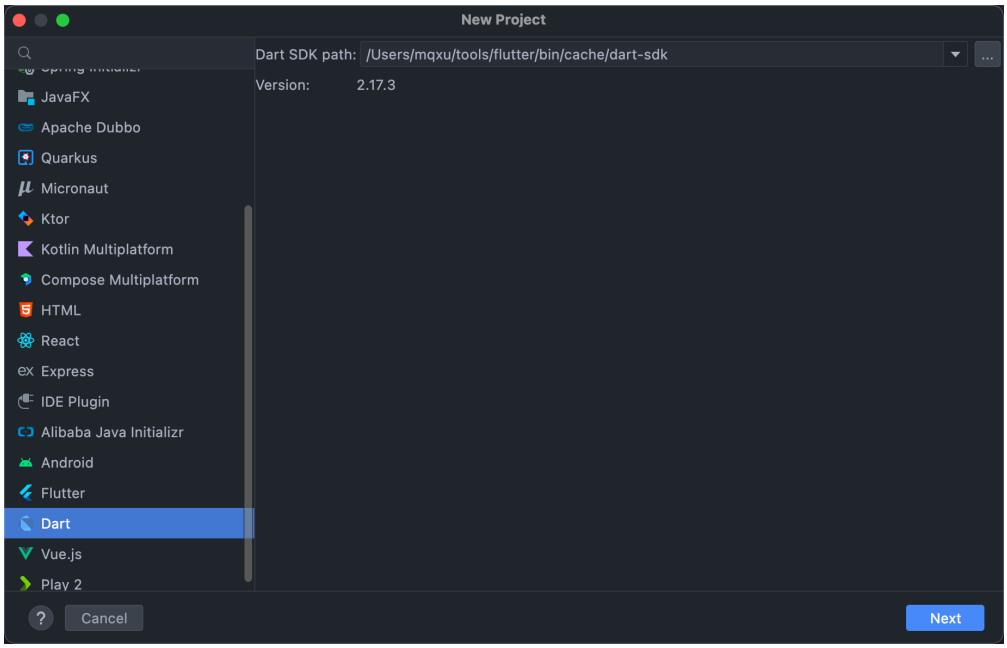
官方文档： https://dart.dev/get-dart

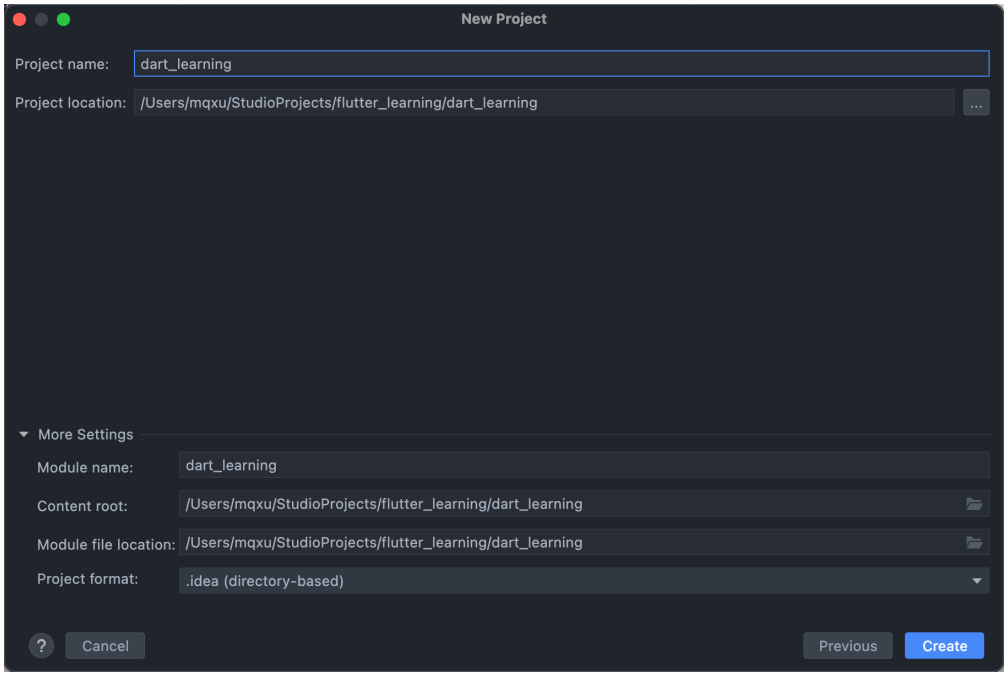
- windows(推荐):
  http://www.gekorm.com/dart-windows/
- mac
  安装homebrew： https://brew.sh/

```
brew tap dart-lang/dart
brew install dart
```

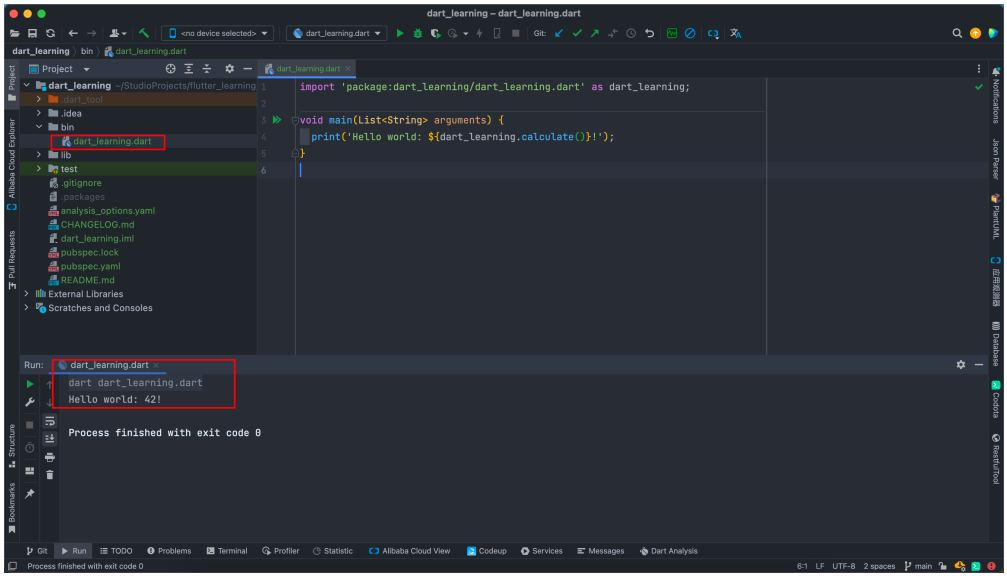
3. Dart 开发工具

Dart的开发工具有很多： IntelliJ IDEA 、 WebStorm、Vscode等
以IDEA为例，安装了Dart插件后，可以新建Dart项目，指定Dart SDK目录，如图所示





运行默认的入口文件，注意观察代码结构

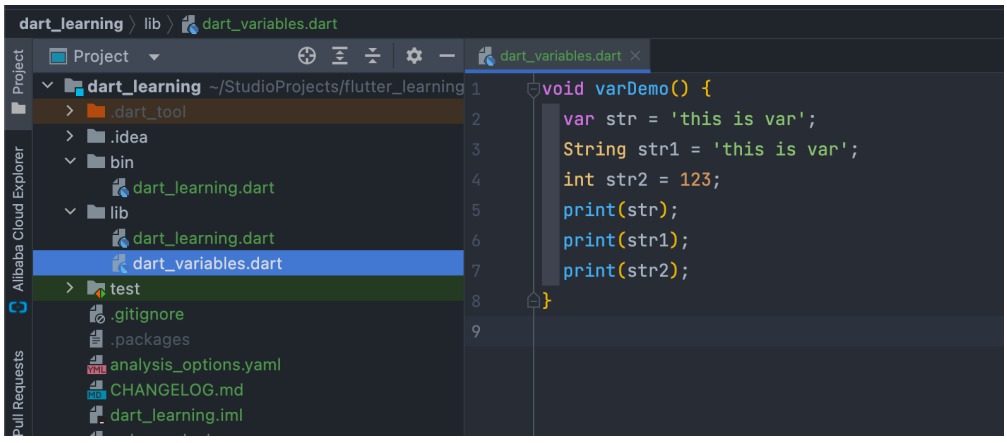


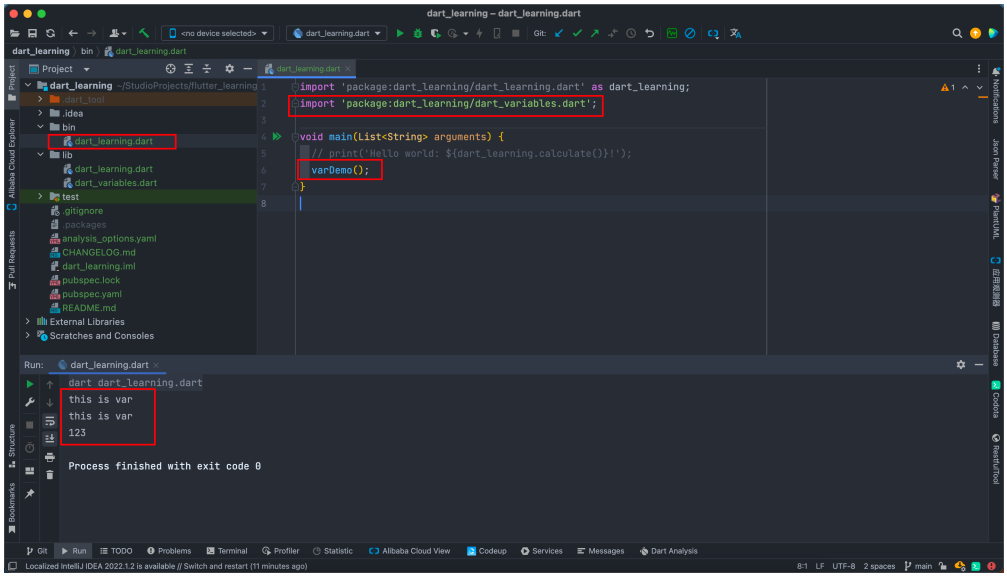
二、变量规则

1. Dart 变量

> dart是一个强大的脚本类语言，可以不预先定义变量类型 ，自动会类型推导

dart中定义变量可以通过**var**关键字可以通过类型来声明变量





注意： var 后就不要写类型 ， 写了类型就 不要var 。

如果两者都写 var a int = 5; 会 报错

2. Dart 常量

使用 **final** 和 **const**修饰符

```
//const值不变 一开始就得赋值
//final 可以开始不赋值 只能赋一次 ; 而final不仅有const的编译时常量的特性，最重要的它是运行时常量，并且final是惰性初始化，即在运行时第一次使用前才初始化

//永远不改量的量，请使用final或const修饰它，而不是使用var或其他变量类型。

  final name = 'Bob';
  final String nickname = 'Bobby';
  const bar = 1000000;
  const double atm = 1.01325 * bar;
```

3. Dart的命名规则

- 变量名称必须由数字、字母、下划线和美元符($)组成。
- 注意：标识符开头不能是数字、不能是保留字和关键字。
- 变量的名字是区分大小写的如: age和Age是不同的变量。在实际的运用中,也建议不要用一个单词大小写区分两个变量。
- 标识符(变量名称)一定要见名识意 ： 变量名称建议用名词，方法名称建议用动词

4. 代码练习

编写代码，理解掌握dart中的变量和常量

```
void varDemo() {
  //用数据类型定义变量
  //字符串类型
  String str = '你好dart';
  print(str);
  //数字类型
  int myNum = 123;
  print(myNum);

  //用var定义变量，dart有类型推导
  var str1 = '';
  // str1 = 123;    // 报错，只能接收字符串
  print(str1);
  var str2 = "123123";
  print(str2);
  var myNum1 = 123321;
  print(myNum1);

  // 命名规则，以下变量名非法
  //var 2str = '123';    //错误
  //var if = 123;     //错误

  // 变量名字区分大小写
  var age = 20;
  var Age = 30;
  print(age);
  print(Age);

  // const常量
  const PI = 3.14159; // 正确，const常量可以在声明的时候接受赋值
  // PI = 123.1243; //错误，const常量不能再被赋值

  // final 常量
  final age1 = age * 2; // 正确，final常量可以接受变量赋值，但是只能设置一次
  // age1 = age * 3;      //错误，final常量只能设置1次

  final date = DateTime.now();
  print(date); // 输出：2022-06-26 17:18:15.884583
```

```
    // const date1 = DateTime.now();    //报错,const常量只能接收常量值
}
```

三、数据类型

Dart中支持以下数据类型

1. 常用数据类型

- 数值类型：int、double
- 字符串类型：String
- 布尔类型：bool
- 集合类型
  - List：在Dart中，数组是列表对象，所以大多数时候称它们为列表
  - Set：可以去重
  - Map：Map 是一个键值对对象。 键和值可以是任何类型的对象。每个键只出现一次，而值则可以出现多次。
- dynamic（动态数据类型）：通过 dynamic 关键字定义的变量，在 编译时不会管定义的数据类型，运行时才处理变量的类型，根据变量赋值的类型推测当前的 dynamic 数据类型的运行时数据类型，dynamic 变量只有在运行时才知道数据类型

2. 不常用的数据类型

- Runes：Runes是UTF-32编码的字符串。它可以通过文字转换成符号表情或者代表特定的文字。

```
void datatypeDemo() {
  var clapping = '\u{1f44f}';
  print(clapping);
  print(clapping.codeUnits);
  print(clapping.runes.toList());

  Runes input = Runes('\u2665  \u{1f605}  \u{1f60e}  \u{1f47b}
\u{1f596}  \u{1f44d}');
  print(String.fromCharCodes(input));
}
```

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

- Symbols：Symbol对象表示在Dart程序中声明的运算符或标识符,在 Dart 中符号用 # 开头来表示。

  http://dart.goodev.org/guides/libraries/library-tour#dartmirrors---reflection

3. 代码练习

编写代码，理解掌握dart中的数据类型。

```
void datatypeDemo() {
  //Dart数据类型：字符串类型
  //1、字符串定义的几种方式
  var str1 = 'this is str1';
  var str2 = "this is str2";
  print(str1);
  print(str2);

  String str3 = 'this is str3';
  String str4 = "this is str4";
  print(str3);
  print(str4);

  String str5 = '''
    this is str5
```

```
    this is str5
    this is str5; ''';
print(str5);

String str6 = '''
    this is str1
    this is str1
    this is str1
    ''';
print(str6);

//2、字符串的拼接
String str7 = '你好';
String str8 = 'Dart';
print("$str1 $str2");
print(str1 + str2);
print(str1 + " " + str2);

//Dart数据类型：数值类型 int 、double
//1、int ： 必须是整型
int a = 123;
a = 45;
print(a);

//2、double ： 既可以是整型， 也可是浮点型
double b = 23.5;
b = 24;
print(b);

//Dart数据类型：布尔类型 bool 值true/false
bool flag1 = true;
print(flag1);
bool flag2 = false;
print(flag2);

//3、运算符
// + - * / %
var c = a + b;
print(c);

//4、条件判断语句
var flag = true;
if (flag) {
  print('真');
```

```
  } else {
    print('假');
  }

  var num = 123;
  var s = '123';
  if (num == s) {
    print(' num == s');
  } else {
    print(' num ≠ s');
  }

  var num1 = 123;
  var s1 = 123;
  if (num1 == s1) {
    print(' num1 == s1');
  } else {
    print(' num1 ≠ s1');
  }

  //Dart数据类型： List（数组/集合）
  //1、第一种定义List的方式：定义的同时初始化
  var list1 = ['aaa', 'bbb', 'ccc'];
  print(list1);
  print(list1.length);
  print(list1[1]);

  //2、第二种定义List的方式：先定义，再添加元素
  var list2 = [];
  list2.add('张三');
  list2.add(123);
  list2.add(true);
  print(list2);
  print(list2[2]);

  //3、定义List并指定元素类型
  var list3 = <String>[];
  list3.add('张三');
  list3.add('李四');
  // list3.add(123);  // 报错
  print(list3);

  //Dart数据类型： Map（字典）
  //第一种定义 Map 的方式：定义的同时初始化
```

```
  var person = {
    "name": "张三",
    "age": 20,
    "work": ["程序员", "送外卖"]
  };
  print(person);
  print(person["name"]);
  print(person["age"]);
  print(person["work"]);

  //第二种定义 Map 的方式：先定义，再以键值对的方式赋值
  var person1 = {};
  person1["name"] = "李四";
  person1["age"] = 22;
  person1["work"] = ["程序员", "送外卖"];
  print(person1);
  print(person1["age"]);

 // dynamic类型
  dynamic d = 'CSDN';
  // 打印运行时类型
  print(d.runtimeType);  //String
  // 类型判断
  print(d is String);   //true
}
```

四、运算符和表达式

1. Dart运算符

- 算术运算符：+、 - 、* 、 / 、 ~/ (取整) 、% （取余）
- 关系运算符：== 、 ！=、 > 、 <、 >= 、 <=
- 逻辑运算符：!、 &&、 ||
- 赋值运算符
  - 基础赋值运算符： = 、 ??=（如果变量没有赋值才进行赋值，否则不进行赋值）
  - 复合赋值运算符： += 、 -=、 *= 、 /= 、 %= 、 ~/=

- 条件表达式：if else   switch case
- 三目运算符
- ?? 运算符

2. 类型转换

- Number与String类型之间的转换
- 其他类型转换成Boolean类型

3. 代码练习

编写代码，理解掌握dart中的各种运算符和表达式。

```
void expressionDemo() {
//1、Dart运算符：
//    - 算术运算符：+、 - 、* 、  / 、  ~/（取整）  、%（取余）
//    - 关系运算符：==  、  !=、   >  、  <、    ≥  、  ≤
//    - 逻辑运算符：！ 、 &&、   ||
//    - 赋值运算符
//    - 基础赋值运算符：   = 、  ??=
//    - 复合赋值运算符：   += 、 -=、  *=  、 /=  、 %= 、 ~/=
//    - 条件表达式：if  else   switch case
//    - 三目运算符
//    - ??运算符

//2、类型转换
//    - Number与String类型之间的转换
//    - 其他类型转换成Boolean类型

  //算术运算符
  int a = 13;
  int b = 5;
  print(a + b);
  print(a - b);
  print(a * b);
  print(a / b);
  print(a % b);
  print(a ~/ b);
  var c = a * b;
  print(c);
```

```
print('----------------');


//关系运算符
a = 5;
b = 3;
print(a == b);
print(a ≠ b);
print(a > b);
print(a < b);
print(a ≥ b);
print(a ≤ b);
if (a > b) {
  print('a大于b');
} else {
  print('a小于b');
}
print('----------------');

//逻辑运算符
bool flag = false;
print(!flag);
bool flag1 = true;
bool flag2 = true;
print(flag1 && flag2);
bool flag3 = false;
bool flag4 = false;
print(flag3 || flag4);
// 指定条件输出
int age = 20;
String sex = "女";
if (age == 20 && sex == "女") {
  print("$age --- $sex");
} else {
  print("不打印");
}
print('----------------');

// 基础赋值运算符 : = , ??=
int num1 = 10;
int num2 = 3;
int num3 = 0;
num3 = num1 + num2;
print('num3 = $num3');
```

```dart
// ??=:  如果变量没有赋值才进行赋值，否则不进行赋值
var i = 2;
var j = i ?? 10;  // i已经被赋值，所以j的值不会执行 ?? 后面的10
print('j = $j');  // j = 2

var g;
var k = g ?? 10;  // g没有被赋值，所以k的值会执行 ?? 后面的10
print('k = $k');  // k = 10


// 复合的赋值运算符   +=  -=  *=   /=   %=  ~/=
var aa = 12;
aa = aa + 10;
print('aa = $aa');

var ab = 13;
ab += 10;
print('ab = $ab');

var ac = 4;
ac *= 3;
print('ac = $ac');
print('----------------');

//条件表达式
//if  else   switch case
bool f = true;
if (f) {
print('true');
} else {
print('false');
}

//多重 if else
var score = 41;
if (score > 90) {
print('优秀');
} else if (score > 70) {
print('良好');
} else if (score ≥ 60) {
print('及格');
} else {
print('不及格');
```

```dart
}
// switch
var gender = "女";
switch (gender) {
case "男":
print('性别是男');
break;
case "女":
print('性别是女');
print('性别是女');
break;
default:
print('传入参数错误');
break;
}

// 三目运算符
var res = true;
String res1 = res ? '结果是true' : '结果是false';
print('res1 = $res1');
print('----------------');

// ?? 运算符
var x;
var y = x ?? 10;
print('y= $y');
var xx = 22;
var yy = xx ?? 10;
print('yy= $yy');
print('----------------');

//类型转换
//1、Number与String类型之间的转换
// Number类型转换成String类型 toString()
// String类型转成Number类型  int.parse()
String str = '123';
var myNum = int.parse(str);
print('myNum = $myNum');
print(myNum is int);

String str1 = '123.1';
var myNum1 = double.parse(str1);
print('myNum1 = $myNum1');
print(myNum1 is double);
```

```
  String price = '12';
  var myNum2 = double.parse(price);
  print('myNum2 = $myNum2');
  print(myNum2 is double);

  //报错
  // String price1 = '';
  // var myNum3 = double.parse(price1);
  // print('myNum3 = $myNum3');
  // print(myNum3 is double);

  // 捕获异常，程序可以正常执行
  String price2 = '';
  try {
  var myNum4 = double.parse(price2);
  print('myNum4 = $myNum4');
  } catch (err) {
  print('程序出错');
  }

  var myNum5 = 12;
  var ss = myNum5.toString();
  print('ss = $ss');
  print(ss is String);
  print('----------------');

// 2、其他类型转换成Booleans类型
  // isEmpty:判断字符串是否为空
  var str2 = '';
  if (str2.isEmpty) {
  print('str2为空');
  } else {
  print('str2不为空');
  }

  var myNum6 = 123;
  if (myNum6 == 0) {
  print('0');
  } else {
  print('非0');
  }

  var myNum7;
```

```
    if (myNum7 == 0) {
    print('0');
    } else {
    print('非0');
    }

    var myNum8;
    if (myNum8 == null) {
    print('空');
    } else {
    print('非空');
    }

    var myNum9 = 0 / 0;
    print('myNum = $myNum');
    if (myNum9.isNaN) {
    print('NaN');
    }
  }
```

五、集合类型

1. List类型

常用属性：

- length　　长度
- reversed　　翻转
- isEmpty　　是否为空
- isNotEmpty　　是否不为空

常用方法：

- add　　增加
- addAll　拼接数组
- indexOf　查找 传入具体值
- remove　删除 传入具体值

- removeAt 删除 传入索引值
- fillRange 修改
- insert(index,value); 指定位置插入
- insertAll(index,list) 指定位置插入List
- toList() 其他类型转换成List
- join() List转换成字符串
- split() 字符串转化成List
- forEach
- map
- where
- any
- every

2. Set类型

- 最主要的功能就是去除数组重复内容
- Set是没有顺序且不能重复的集合，所以不能通过索引去获取值

3. Map类型

4. 代码练习

编写代码，理解掌握dart中的集合类型。

```
void listDemo() {
  //List的属性
  List myList = ['香蕉', '苹果', '西瓜'];
  print(myList.length);
  print(myList.isEmpty);
  print(myList.isNotEmpty);
  print(myList.reversed);
  //对列表倒序排序
```

```
  var newMyList = myList.reversed.toList();
  print('newMyList = $newMyList');

  //List的方法
  List myList1 = ['香蕉', '苹果', '西瓜'];
  myList1.add('桃子');
  myList1.addAll(['桃子', '葡萄']); //拼接数组
  print('myList1 = $myList1');
  print(myList1.indexOf('苹x果')); //indexOf查找数据 查找不到返回-1  查找到返回索引值
  myList1.remove('西瓜');
  myList1.removeAt(1);
  print('myList1 = $myList1');

  List myList2 = ['香蕉', '苹果', '西瓜'];
  myList2.fillRange(1, 2, 'aaa'); //修改
  myList2.fillRange(1, 3, 'aaa');
  myList2.insert(1, 'aaa'); //插入一个
  myList2.insertAll(1, ['aaa', 'bbb']); //插入多个
  print('myList2 = $myList2');

  List myList3 = ['香蕉', '苹果', '西瓜'];
  var str = myList3.join('-'); //list转换成字符串
  print('str = $str');
  print(str is String); //true
  var str1 = '香蕉-苹果-西瓜';
  var list = str1.split('-');
  print('list = $list');
  print(list is List);

  //Set
  //最主要的功能就是去除数组重复内容
  //Set是没有顺序且不能重复的集合，所以不能通过索引去获取值
  var s = <dynamic>{};
  s.add('香蕉');
  s.add('苹果');
  s.add('苹果');
  print('s = $s'); //{香蕉, 苹果}
  print(s.toList());

  List ll = ['香蕉', '苹果', '西瓜', '香蕉', '苹果', '香蕉', '苹果'];
  var ss = <dynamic>{};
  ss.addAll(ll);
  ss.forEach((value) => print(value));
```

```
    print(ss.toList());

    // Map
    Map person = {"name": "张三", "age": 20};
    var m = {};
    m["name"] = "李四";
    print('person = $person');
    print('m = $m');

    //常用属性
    Map person1 = {"name": "张三", "age": 20, "sex": "男"};
    person1.forEach((key, value) {
      print("$key---$value");
    });
    print(person1.keys.toList());
    print(person1.values.toList());
    print(person1.isEmpty);
    print(person1.isNotEmpty);

    //常用方法:
    Map person2 = {"name": "张三", "age": 20, "sex": "男"};
    person2.addAll({
      "work": ['敲代码', '送外卖'],
      "height": 160
    });
    print('person2 = $person2');
    person2.remove("sex");
    print(person2);
    print(person.containsValue('张三'));

  //forEach、map、where、any、every
    List myList4 = ['香蕉', '苹果', '西瓜'];
    for (var i = 0; i < myList4.length; i++) {
      print(myList4[i]);
    }
    for (var item in myList4) {
      print(item);
    }
    myList4.forEach((value) {
      print("$value");
    });

    List myList5 = [1, 3, 4];
    List newList = [];
```

```
    for (var i = 0; i < myList5.length; i++) {
      newList.add(myList5[i] * 2);
    }
    print('newList = $newList');

    List myList6 = [1, 3, 4];
    var newList1 = myList6.map((value) {
      return value * 2;
    });
    print('newList1 = $newList1.toList()');

    List myList7 = [1, 3, 4, 5, 7, 8, 9];
    var newList2 = myList7.where((value) {
      return value > 5;
    });
    print('newList2 = $newList2.toList()');

    List myList8 = [1, 3, 4, 5, 7, 8, 9];
    var f = myList8.any((value) {
      //只要集合里面有满足条件的就返回true
      return value > 5;
    });
    print('f = $f');

    List myList9 = [1, 3, 4, 5, 7, 8, 9];
    var ff = myList9.every((value) {
      //每一个都满足条件返回true  否则返回false
      return value > 5;
    });
    print('ff = $ff');
  }
```

六、方法和函数

1. 内置方法/函数

```
 print();
```

2. 自定义方法

自定义方法的基本格式：

```
返回类型  方法名称（参数1，参数2，...）{
  方法体
    return 返回值;
}

// 示例
void printInfo(){
  print('我是一个自定义方法');
}

int getNum(){
  var myNum=123;
  return myNum;
}

String printUserInfo(){
  return 'this is str';
}


List getList(){
  return ['111','2222','333'];
}

int getNum(int n){
  return n;
}

void main(){
   print('调用系统内置的方法');
   printInfo();
   var n=getNum();
   print(n);
```

```
    print(printUserInfo());
    print(getList());
}
```

3. 方法的作用域

```
void xxx(){
  aaa(){
    print(getList());
    print('aaa');
  }
  aaa();
}

// aaa();  错误写法
xxx();    //调用方法
```

4. 代码练习

1、定义一个方法，给定参数， 求1到这个数的所有数的和

```
int getSum(int n) {
  var sum = 0;
  for (var i = 1; i ≤ n; i++) {
    sum += i;
  }
  return sum;
}

void main() {
  var n1 = getSum(5);
  print(n1);
  var n2 = getSum(100);
  print(n2);
}
```

2、定义一个方法然后打印用户信息

```
String printUserInfo(String username, int age) {
  return "姓名:$username---年龄:$age";
}

void main() {
  print(printUserInfo('张三', 20));
}
```

3、定义一个带可选参数的方法

```
String printUserInfo(String username, [age]) {
  if (age != null) {
    return "姓名:$username---年龄:$age";
  }
  return "姓名:$username---年龄:保密";
}

void main() {
  print(printUserInfo('张三',21));
  print(printUserInfo('张三'));
}
```

输出

```
姓名:张三---年龄:21
姓名:张三---年龄:保密
```

4、定义一个带默认参数的方法

```
String printUserInfo(String username, [String sex = '男', int age = 20])
{
  return "姓名:$username---性别:$sex---年龄:$age";
}

void main() {
  print(printUserInfo('张三'));
  print(printUserInfo('小李','女'));
  print(printUserInfo('小李','女',30));
}
```

输出

```
姓名:张三---性别:男---年龄:20
姓名:小李---性别:女---年龄:20
姓名:小李---性别:女---年龄:30
```

5、定义一个命名参数的方法

```
String printUserInfo(String username, {required int age, String sex =
'男'}) {
  return "姓名:$username---性别:$sex--年龄:$age";
}
void main(){
   print(printUserInfo3('张三', age: 20, sex: '未知'));
}
```

输出

```
姓名:张三---性别:未知--年龄:20
```

6、实现一个把方法当做参数的方法

```
var fn = () {
  print('我是一个匿名方法');
};

fn1() {
  print('fn1');
}

fn2(fn) {
  fn();
}

void main() {
  fn();
  //调用fn2方法 把fn1方法当做参数传入
  fn2(fn1);
}
```

7、使用forEach打印List里面的数据

```
fn3() {
  List list = ['苹果', '香蕉', '西瓜'];
  list.forEach((value) {
    print(value);
  });
  print('************');
  list.forEach((value) => print(value));
  print('************');
  list.forEach((value) => {print(value)});
}
```

8、修改List里面的数据，让数组中大于2的值乘以2

```
fn4() {
  List list = [4, 1, 2, 3, 4];
  var newList = list.map((value) {
    if (value > 2) {
      return value * 2;
    }
    return value;
  });
  print(newList.toList());
  var newList1 = list.map((value) => value > 2 ? value * 2 : value);
  print(newList1.toList());
}
```

9、实现以下需求

- 定义一个方法isEvenNumber来判断一个数是否是偶数
- 定义一个方法打印1-n以内的所有偶数

```
bool isEvenNumber(int n) {
  if (n % 2 == 0) {
    return true;
  }
  return false;
}

printNum(int n) {
  for (var i = 1; i < =  n; i++) {
    if (isEvenNumber(i)) {
      print(i);
```

```
    }
  }
}

void main(){
   printNum(10);
}
```

10、匿名方法、自执行方法

```
int getNum(int n) {
  return n;
}
void main(){
  print(getNum(12));
  //匿名方法
  var printNum = () {
    print(123);
  };
  printNum();

  var printNum1 = (int n) {
    print(n + 2);
  };
  printNum1(12);

//自执行方法
  ((int n) {
    print(n);
    print('我是自执行方法');
  })(12);
}
```

11、方法递归：通过方法的递归 求1-100的和

```
getSum(int n) {
  if (n == 0) {
    return 0;
  }
  return n + getSum(n - 1);
}

void main(){
  var res = getSum(100);
  print(res);
}
```

12、闭包

- 全局变量特点: 全局变量常驻内存、全局变量污染全局
- 局部变量的特点: 不常驻内存会被垃圾机制回收、不会污染全局

> 闭包: 函数嵌套函数, 内部函数会调用外部函数的变量或参数, 变量或参数不会被系统回收(不会释放内存)

闭包的写法: 函数嵌套函数，并return 里面的函数，这样就形成了闭包。

```
//全局变量
var a = 123;

void main() {
  print(a);  // 123

  fn() {
    a++;
    print(a);  //124
  }

  fn();

  //局部变量
  printInfo() {
    var myNum = 123;
    myNum++;
    print(myNum);  // 124
  }
```

```
  printInfo();  // 124

  //闭包
  // 不会污染全局 常驻内存
  fn1() {
    var a = 123;
    return () {
      a++;
      print(a);
    };
  }

  var b = fn1();
  b();  // 124
}
```

七、面向对象

> Dart是一门使用类和单继承的面向对象语言，所有的对象都是类的实例，并且所有的类都是Object的子类。
>
> 一个类通常由属性和方法组成。

1. 类的使用

```
void main(){
  Object a = 123;
  Object v = true;
  print(a);
  print(v);
}
```

创建类并使用

```
class Person {
  String name = "张三";
```

```
  int age = 23;

  void getInfo() {
    print("$name----$age");
  }

  void setInfo(int age) {
    this.age = age;
  }
}

void main(){
  Person p1 = Person();
  print(p1.name);
  p1.setInfo(28);
  p1.getInfo();
}
```

自定义默认构造函数

```
Person() {
  print('这是构造函数里面的内容,这个方法在实例化的时候触发');
}

void printInfo() {
  print("$name----$age");
}
}

void main(){
  Person p1 = Person();
  p1.printInfo();
}
```

自定义命名构造函数

dart里面构造函数可以写多个

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

  //默认构造函数的简写
```

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

    Person.now(this.name, this.age) {
      print('我是命名构造函数');
    }

    Person.setInfo(this.name, this.age);

    void printInfo() {
      print("$name----$age");
    }
  }

  void main(){
    Person p1 = Person('张三', 20); //默认实例化类的时候调用的是 默认构造函数
    Person p2 = Person.now('李四',22); //调用命名构造函数
    Person p3 = Person.setInfo('王五', 24);
    p1.printInfo();
    p2.printInfo();
    p3.printInfo();
  }
```

Dart和其他面向对象语言不一样，没有 public private protected这些访问修饰符，但是我们可以使用_把一个属性或者方法定义成私有。

Dart中，我们也可以在构造函数体运行之前初始化实例变量

```
  class Rect {
    int height;
    int width;

    Rect()
        : height = 2,
          width = 10 {
      print("$height---$width");
    }

    getArea() {
      return height * width;
    }
  }

  void main(){
```

```
  Rect r = Rect();
  print(r.getArea());
}
```

2. 继承

Dart中的类的继承：

1、子类使用**extends**关键词来继承父类
2、子类会继承父类里面可见的属性和方法 但是不会继承构造函数
3、子类能覆写父类的方法 getter和setter

extends

```
class Person {
  String name = '张三';
  num age = 20;

  void printInfo() {
    print("$name---$age");
  }
}

class Web extends Person {}

main() {
  Web w = Web();
  print(w.name);
  w.printInfo();
}
```

super

```
class Person {
  String name;
  num age;

  Person(this.name, this.age);

  void printInfo() {
    print("$name---$age");
```

```
  }
}

class Web extends Person {
  Web(String name, num age) : super(name, age);
}

main() {
  Person p = Person('李四', 20);
  p.printInfo();

  Person p1 = Person('张三', 20);
  p1.printInfo();

  Web w = Web('张三', 12);

  w.printInfo();
}
```

super实例化

```
class Person {
  String name;
  num age;

  Person(this.name, this.age);

  void printInfo() {
    print("$name---$age");
  }
}

class Web extends Person {
  String sex;

  Web(String name, num age, this.sex) : super(name, age);

  run() {
    print("$name---$age--$sex");
  }
}

void main(){
  Web w = Web('张三', 12, "男");
```

```
  w.printInfo();
  w.run();
}
```

覆写父类方法

```
class Person {
  String name;
  num age;

  Person(this.name, this.age);

  void printInfo() {
    print("$name---$age");
  }

  work() {
    print("$name在工作 ... ");
  }
}

class Web extends Person {
  Web(String name, num age) : super(name, age);

  run() {
    print('run');
  }

  //覆写父类的方法,@override 可以省略，不过建议在覆写父类方法的时候加上
@override
  @override
  void printInfo() {
    print("姓名: $name---年龄: $age");
  }

  @override
  work() {
    print("$name的工作是写代码");
  }
}

void main(){
```

```
  Web w = Web('李四', 20);

  w.printInfo();

  w.work();
}
```

调用父类方法

```
class Person {
  String name;
  num age;

  Person(this.name, this.age);

  void printInfo() {
    print("$name---$age");
  }

  work() {
    print("$name在工作 ... ");
  }
}

class Web extends Person {
  Web(String name, num age) : super(name, age);

  run() {
    print('run');
    super.work(); //子类调用父类的方法
  }

  @override
  void printInfo() {
    print("$name---$age");
  }
}

void main(){
  Web w = Web('李四', 20);
  w.run();
}
```

八、类方法

1. Dart中的静态成员

- 使用static 关键字来实现类级别的变量和函数
- 静态方法不能访问非静态成员，非静态方法可以访问静态成员

```
class Person {
  static String name = '张三';

  static void show() {
    print(name);
  }
}

main() {
  print(Person.name);
  Person.show();
}
```

```
class Person {
  static String name = '张三';
  int age = 20;

  static void show() {
    print(name);
  }

  void printInfo() {
    /*非静态方法可以访问静态成员以及非静态成员*/
    print(name); //访问静态属性
    print(age); //访问非静态属性
    show(); //调用静态方法
  }
```

```
  static void printUserInfo() {
    //静态方法
    print(name); //静态属性
    show(); //静态方法
    // print(this.age); //静态方法没法访问非静态的属性
    // this.printInfo(); //静态方法没法访问非静态的方法
  }
}

void main(){
  print(Person.name);
  Person.show();

  Person p=Person();
  p.printInfo();

  Person.printUserInfo();
}
```

2. Dart中的对象操作符

- ? 条件运算符 （了解）
- as 类型转换
- is 类型判断
- .. 级联操作 （连缀） （记住）

```
class Person1 {
  String name;
  num age;

  Person1(this.name, this.age);

  void printInfo() {
    print("$name---$age");
  }
}

void main(){
  Person1 p1 = Person1('张三', 20);
```

```
  p1.printInfo();

  Person1 p2 = Person1('张三', 20);
  if (p2 is Person1) {
    p2.name = "李四";
  }
  p2.printInfo();
  print(p2 is Object);

  Person1 p3 = Person1('张三111', 20);
  p3.printInfo();
  p3.name = '张三222';
  p3.age = 40;
  p3.printInfo();

  Person1 p4 = Person1('张三', 20);
  p4.printInfo();
  p4
    ..name = "李四"
    ..age = 30
    ..printInfo();
}
```

九、接口和多态

1. 抽象类

> Dart中抽象类: Dart抽象类主要用于定义标准，子类可以继承抽象类，也可以实现抽象类接口。

1、抽象类通过abstract 关键字来定义

2、Dart中的抽象方法不能用abstract声明，Dart中没有方法体的方法称为抽象方法。

3、如果子类继承抽象类必须得实现里面的抽象方法

4、如果把抽象类当做接口实现的话必须得实现抽象类里面定义的所有属性和方法。

5、抽象类不能被实例化，只有继承它的子类可以

继承抽象类 和 实现抽象类的区别：

1、如果要复用抽象类里面的方法，并且要用抽象方法约束子类的话，就用extends继承抽象类

2、如果只是把抽象类当做标准的话，就用implements实现抽象类

> 案例：定义一个Animal 类，要求它的子类必须包含eat方法

```
abstract class Animal {
  eat(); //抽象方法
  run(); //抽象方法
  printInfo() {
    print('我是一个抽象类里面的普通方法');
  }
}

class Dog extends Animal {
  @override
  eat() {
    print('小狗在吃骨头');
  }

  @override
  run() {
    print('小狗在跑');
  }
}

class Cat extends Animal {
  @override
  eat() {
    print('小猫在吃老鼠');
  }

  @override
  run() {
    print('小猫在跑');
```

```
  }
}

main() {
  Dog d = Dog();
  d.eat();
  d.printInfo();

  Cat c = Cat();
  c.eat();

  c.printInfo();

  // Animal a = Animal();    //抽象类没法直接被实例化
}
```

2. 多态

> Datr中的多态：允许将子类类型的指针赋值给父类类型的指针, 同一个函数调用会有不同的执行效果 。

子类的实例赋值给父类的引用。

多态就是父类定义一个方法不去实现，让继承他的子类去实现，每个子类有不同的表现。

多态示例代码

```
abstract class Animal {
  eat(); //抽象方法
}

class Dog extends Animal {
  @override
  eat() {
    print('小狗在吃骨头');
  }

  run() {
```

```
    print('run');
  }
}

class Cat extends Animal {
  @override
  eat() {
    print('小猫在吃老鼠');
  }

  run() {
    print('run');
  }
}

main() {
  Animal d = Dog();
  d.eat();
  Animal c = Cat();
  c.eat();
}
```

3. 接口

> 和Java一样，dart也有接口，但是和Java还是有区别的。

首先，dart的接口没有interface关键字定义接口，而是普通类或抽象类都可以作为接口被实现，都使用implements关键字进行实现。

dart的接口如果实现的类是普通类，会将普通类和抽象中的属性的方法全部需要覆写一遍。

因为抽象类可以定义抽象方法，普通类不可以，所以如果要实现像Java接口那样的方式，一般会使用抽象类。

建议使用抽象类定义接口。

> 案例：定义一个DB库 支持 mysql mssql mongodb

> mysql mssql mongodb三个类里面都有同样的方法

```
abstract class Db {
  //当做接口    接口就是约定 、规范
  late String uri; //数据库的链接地址
  add(String data);

  save();

  delete();
}

class MySQL implements Db {
  @override
  String uri;

  MySQL(this.uri);

  @override
  add(data) {
    print('这是 MySQL 的add方法, $data');
  }

  @override
  delete() {
    return null;
  }

  @override
  save() {
    return null;
  }

  remove() {}
}

class MsSQL implements Db {
  @override
  late String uri;

  @override
  add(String data) {
    print('这是 MsSQL 的add方法, $data');
```

```
  }

  @override
  delete() {
    return null;
  }

  @override
  save() {
    return null;
  }
}

main() {
  MySQL mysql = MySQL('localhost:3306');
  mysql.add('插入的一条记录');
}
```

也可以将以上实现类分到不同的文件中。

4. 多接口实现Mixins

> mixins的中文意思是混入，就是在类中混入其他功能。
>
> 在Dart中可以使用mixins实现类似多继承的功能。

Dart2.x中使用mixins的条件：

1、作为mixins的类只能继承自Object，不能继承其他类
2、作为mixins的类不能有构造函数
3、一个类可以mixins多个mixins类
4、mixins绝不是继承，也不是接口，而是一种全新的特性

示例1：

```
class A {
  String info = "this is A";

  void printA() {
    print("A");
  }
```

```
}

class B {
  void printB() {
    print("B");
  }
}

class C with A, B {}

void main() {
  var c = C();
  c.printA();
  c.printB();
  print(c.info);
}
```

示例2：

```
class Person {
  String name;
  num age;

  Person(this.name, this.age);

  printInfo() {
    print('$name----$age');
  }

  void run() {
    print("Person Run");
  }
}

class A {
  String info = "this is A";

  void printA() {
    print("A");
  }

  void run() {
    print("A Run");
  }
```

```
}

class B {
  void printB() {
    print("B");
  }

  void run() {
    print("B Run");
  }
}

class C extends Person with B, A {
  C(String name, num age) : super(name, age);
}

void main() {
  var c = C('张三', 20);
  c.printInfo();
  c.printB();
  print(c.info);
  c.run();
}
```

Mixins 实例类型

mixins的实例类型是什么？很简单，就是其超类的子类型。

```
class A {
  String info = "this is A";

  void printA() {
    print("A");
  }
}

class B {
  void printB() {
    print("B");
  }
}

class C with A, B {}
```

```
void main() {
  var c = C();
  print(c is C); //true
  print(c is A); //true
  print(c is B); //true
}
```

十. 泛型

> 通俗理解：泛型就是解决 类 接口 方法的复用性、以及对不特定数据类型的支持(类型校验)

1. 泛型、泛型方法

只能返回String类型的数据

```
String getData(String value){
  return value;
}
```

同时支持返回 String类型 和int类型 （代码冗余）

```
String getData1(String value){
  return value;
}

int getData2(int value){
  return value;
}
```

同时返回 String类型 和number类型 —— 不指定类型可以解决这个问题

```
getData(value){
  return value;
}
```

> 不指定类型放弃了类型检查。
>
> 我们现在想实现的是：传入什么类型，就返回什么类型。
>
> 比如:传入number 类型必须返回number类型 ，传入 String类型必须返回String类型

```
T getData<T>(T value) {
  return value;
}

void main() {
  print(getData(21));
  print(getData('hello'));
  print(getData<String>('你好'));
  print(getData<int>(12));
}
```

2. 泛型类

> 案例：把下面的类转换成泛型类，要求List里面可以增加int类型的数据，也可以增加String类型的数据。但是每次调用增加的类型要统一。

原代码

```
class PrintClass {
  List list = <int>[];

  void add(int value) {
    list.add(value);
  }

  void printInfo() {
    for (var i = 0; i < list.length; i++) {
      print(list[i]);
```

```
    }
  }
}

void main(){
  // 类型可以是各种
    PrintClass p =  PrintClass();
    p.add(123);
    p.add('abc');
    p.printInfo();
}
```

修改如下：

```
class PrintClass<T> {
  List list = <T>[];

  void add(T value) {
    list.add(value);
  }

  void printInfo() {
    for (var i = 0; i < list.length; i++) {
      print(list[i]);
    }
  }
}

void main() {
  PrintClass p1 = PrintClass<String>();
  p1.add('abc');
  // p1.add(123); // 报错，因为不符合泛型类中指定的String泛型
  p1.add('xyz');
  p1.printInfo();

  PrintClass p2 = PrintClass<int>();
  p2.add(12);
  p2.add(34);
  p2.printInfo();
}
```

3. 泛型接口

> 案例：实现数据缓存的功能：有文件缓存、和内存缓存。
>
> 内存缓存和文件缓存按照接口约束实现。

1、定义一个泛型接口 约束实现它的子类必须有getByKey(key) 和 setByKey(key,value)

2、要求setByKey的时候的value的类型和实例化子类的时候指定的类型一致

```
abstract class ObjectCache {
  getByKey(String key);

  void setByKey(String key, Object value);
}

abstract class StringCache {
  getByKey(String key);

  void setByKey(String key, String value);
}

abstract class Cache<T> {
  getByKey(String key);

  void setByKey(String key, T value);
}

class FileCache<T> implements Cache<T> {
  @override
  getByKey(String key) {
    return null;
  }

  @override
  void setByKey(String key, T value) {
    print("内存缓存把key=$key, value=$value 写到了内存中");
  }
}
```

```
class MemoryCache<T> implements Cache<T> {
  @override
  getByKey(String key) {
    return null;
  }

  @override
  void setByKey(String key, T value) {
    print("内存缓存把key=$key, value=$value 写到了内存中");
  }
}

void main() {
  MemoryCache m = MemoryCache<String>();
  m.setByKey('index', '首页数据');
  MemoryCache m1 = MemoryCache<Map>();
  m1.setByKey('index', {"name": "张三", "age": 20});
}
```

十一、空安全

1. 概述

- Dart支持健全的空安全机制
- 在变量的类型声明上加 `?` 表示变量是可空类型，否则是非空类型

使用

非空类型：

```
String getAddress() {
  return "北京市";
}

void demo1() {
  String name = "小明";
  int age = 18;
  String address = getAddress();
  print('demo1:$name,$age,$address');
}
```

可空类型：

```
void demo2() {
  String? name;
  int? age;
  String? address;
  print('demo2:$name,$age,$address');
}
```

2. 空安全操作符

?.

左边值为空返回null，否则返回右边的值。

```
void demo3() {
  String? address = getAddress();
  print('demo3:${address.length}');
}
```

??

如果当左边表达式的值不为null，则直接返回自己的值，否则返回右边表达式的值。

```
void demo4() {
  String? address = getAddress();
  String newAddress = address ?? "未知";
  print('demo4:$newAddress');
}
```

??=

左边值不为null时，直接返回自己的值，否则返回右边表达式的值。

```
void demo5() {
  String? address;
  address ??= "未知";
  print('demo5:$address');
}
```

3. 判断技巧

```
void main() {
  var a;
  var b = "";
  var c = 0;

  if ([null, "", 0].contains(a)) {
    print(a);
  }
  if ([null, "", 0].contains(b)) {
    print(b);
  }
  if ([null, "", 0].contains(c)) {
    print(c);
  }
}
```

十二、异步编程

> Dart 是一门单线程编程语言。
>
> **异步 IO + 事件循环**

1. 函数当参数传递

JS写法

```
new Promise((resolve, reject) =>{
    // 发起请求
    const xhr = new XMLHttpRequest();
    xhr.open("GET", 'https://www.nowait.xin/');
    xhr.onload = () => resolve(xhr.responseText);
    xhr.onerror = () => reject(xhr.statusText);
    xhr.send();
}).then((response) => { //成功
   console.log(response);
}).catch((error) => { // 失败
   console.log(error);
});
```

Dart写法

```
Future<Response> respFuture = http.get('https://example.com'); //发起请
求
respFuture.then((response) { //成功，匿名函数
  if (response.statusCode == 200) {
    var data = reponse.data;
  }
}).catchError((error) { //失败
   handle(error);
});
```

http://t.weather.sojson.com/api/weather/city/101030100

2. Future

> Future 对象封装了Dart 的异步操作，它有未完成（uncompleted）和已完成（completed）两种状态。

completed状态也有两种：一种是代表操作成功，返回结果；另一种代表操作失败，返回错误。

```
Future<String> fetchUserOrder() {
  //想象这是个耗时的数据库操作
  return Future(() => 'Large Latte');
}

void main() {
  fetchUserOrder().then((result){print(result)})
  print('Fetching user order ... ');
}
```

通过then来回调成功结果，main会先于Future里面的操作，输出结果：

```
Fetching user order ...
Large Latte
```

Future同名构造器是factory Future(FutureOr computation())，它的函数参数返回值为FutureOr类型，我们发现还有很多Future中的方法比如Future.then、Future.microtask的参数类型也是FutureOr，看来有必要了解一下这个对象。

FutureOr 是个特殊的类型，它没有类成员，不能实例化，也不可以继承，看起来是一个语法糖。

```
abstract class FutureOr<T> {
  // Private generative constructor, so that it is not subclassable,
mixable, or
  // instantiable.
  FutureOr._() {
    throw new UnsupportedError("FutureOr can't be instantiated");
  }
}
```

3. async 和 await

想象一个这样的场景：

1. 先调用登录接口；
2. 根据登录接口返回的token获取用户信息；
3. 最后把用户信息缓存到本机；

```
Future<String> login(String name,String password){
  //登录
}
Future<User> fetchUserInfo(String token){
  //获取用户信息
}
Future saveUserInfo(User user){
  // 缓存用户信息
}
```

用Future可以这样写：

```
login('name','password').then((token) => fetchUserInfo(token))
  .then((user) => saveUserInfo(user));
```

换成async 和await 则可以这样写：

```
void doLogin() async {
  String token = await login('name','password'); //await 必须在 async 函数体内
  User user = await fetchUserInfo(token);
  await saveUserInfo(user);
}
```

声明了async 的函数，返回值是必须是Future对象。

在async函数里直接返回T类型数据，编译器会自动帮你包装成Future类型的对象，如果是void函数，则返回Future对象。在遇到await的时候，又会把Futrue类型拆包，又会原来的数据类型暴露出来，注意：**await 所在的函数必须添加async关键词**。

await的代码发生异常，捕获方式跟同步调用函数一样：

```
void doLogin() async {
  try {
    var token = await login('name','password');
    var user = await fetchUserInfo(token);
    await saveUserInfo(user);
  } catch (err) {
    print('Caught error: $err');
  }
}
```

语法糖：

```
Future<String> getUserInfo() async {
  return 'aaa';
}
//等价于：
Future<String> getUserInfo() async {
  return Future.value('aaa');
}
```

十三、系统自定义第三方库

1. 库简介

> 前面介绍Dart基础知识的时候基本上都是在一个文件里面编写Dart代码的，但实际开发中不可能这么写，模块化很重要，所以这就需要使用到库的概念。

在Dart中，库的使用时通过**import**关键字引入的。

library指令可以创建一个库，每个Dart文件都是一个库，即使没有使用library指令来指定。

Dart中的库主要有三种：

1、我们自定义的库

```
  import 'lib/xxx.dart';
```

示例代码

person.dart

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

  //默认构造函数的简写
  Person(this.name, this.age);

  Person.setInfo(this.name, this.age);

  void printInfo() {
    print("Person1:$name----$age");
  }
}
```

my_data.dart

```
void getName() {
  print('张三');
}

void getAge() {
  print(20);
}

void getSex() {
  print('男');
}
```

import_custom_lib.dart

```dart
import 'my_data.dart';
import 'person.dart';

void main() {
  Person p1 = Person('张三', 20);
  p1.printInfo();

  getName();
  getAge();
  getSex();
}
```

2、系统内置库

```dart
import 'dart:math';
import 'dart:io';
import 'dart:convert';
```

示例代码

```dart
import 'dart:math';
import 'dart:io';
import 'dart:convert';

void main() async {
  print(min(12, 23));
  print(max(12, 25));
  var result = await getDataFromZhihuAPI();
  print(result);
}

getDataFromZhihuAPI() async {
  //1、创建HttpClient对象
  var httpClient = HttpClient();
  //2、创建Uri对象
  var uri = Uri.http('news-at.zhihu.com', '/api/3/stories/latest');
  //3、发起请求，等待请求
  var request = await httpClient.getUrl(uri);
  //4、关闭请求，等待响应
  var response = await request.close();
  //5、解码响应的内容
  return await response.transform(utf8.decoder).join();
}
```

3、Pub包管理系统中的库

https://pub.dev/packages
https://pub.flutter-io.cn/packages
https://pub.dartlang.org/flutter/

- 需要在项目根目录的pubspec.yaml中配置库的名称 、描述、依赖等信息

```
    name: xxx
    description: A new flutter module project.
    dependencies:
        http: ^0.13.4
        date_format: ^2.0.6
```

- 然后运行 `pub get` 获取远程库
- 看文档引入库使用

示例代码

```
// 引入第三方库
import 'dart:convert' as convert;
import 'package:http/http.dart' as http;
import 'package:date_format/date_format.dart';

void main() async {
  var httpsUri = Uri(
    scheme: 'https',
    host: 'news-at.zhihu.com',
    path: '/api/3/stories/latest',
  );
  var response = await http.get(httpsUri);
  if (response.statusCode == 200) {
    var jsonResponse = convert.jsonDecode(response.body);
    print(jsonResponse);
  } else {
    print("Request failed with status: ${response.statusCode}.");
  }
  print(formatDate(DateTime(2022, 06, 22), [yyyy, '-', mm, '-', dd]));
}
```

```
## 2. 命名冲突
```

1、冲突解决

当引入两个库中有相同名称标识符的时候，如果是Java，通常我们可以通过写上完整的包名路径来指定使用的具体标识符，甚至不用import都可以；

但是Dart里面是必须import的，当冲突的时候，可以使用as关键字来指定库的前缀。

示例代码

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


Element element1 = new Element();           // Uses Element from lib1.
lib2.Element element2 = new lib2.Element(); // Uses Element from lib2.
```

```
import 'lib/Person1.dart';
import 'lib/Person2.dart' as lib;

main(List<String> args) {
  Person p1 = Person('张三', 20);
  p1.printInfo();
  lib.Person p2=new lib.Person('李四', 20);
  p2.printInfo();
}
```

3.部分导入

> 如果只需要导入库的一部分，有两种模式

模式一：只导入需要的部分，使用show关键字，如下所示：

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

模式二：隐藏不需要的部分，使用hide关键字，如下所示：

```
import 'package:lib2/lib2.dart' hide foo;
```

```
import 'my_data.dart' show getAge;
import 'my_data.dart' hide getName;

void main() {
  // getName();
  getAge();
}
```

4. 延迟加载

> 也称为懒加载，可以在需要的时候再进行加载。 懒加载的最大好处是可以减少APP的启动时间。

懒加载使用**deferred as**关键字来指定，如下所示：

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

 // 当需要使用的时候，需要使用loadLibrary()方法来加载：
  greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

4. 异步方法调用

- async和await
  - 只有async方法才能使用await关键字调用方法
  - 如果调用别的async方法必须使用await关键字

- async是让方法变成异步
- await是等待异步方法执行完成

示例代码

```
//异步方法
testAsync() async {
  return 'Hello async';
}

void main() async {
  var result = await testAsync();
  print(result);
}
```