一、语言基础

1、数据类型

1.1、类型概述

a、Java

在java语言中，变量分为两种：基本类型和引用类型。

基本数据类型是CPU可以直接进行运算的类型。Java定义了以下几种基本数据类型：

- 整数类型：byte、short、int、long
- 浮点数类型：float、double
- 字符类型：char
- 布尔类型：bool

b、Go

在Go语言中，数据类型用于声明函数和变量。

数据类型的出现是为了把数据分成所需内存大小不同的类型，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

所有类型都能赋值给interface{}空接口；GO语言中没有类的概念，只能通过struct和method来模拟；

Go语言按类别有以下几种数据类型：

1. 布尔型：bool
2. 数字类型：
    - 无符号整数：uint8、uint16、uint32、uint64

- 有符号整数：int8、int16、int32、int64
- 浮点数：float32、float64
- 布尔：bool
- 其它类型：
  - byte、uint8 ： 两种类型可以互换，是一回事
  - rune、int32：两种类型可以互换，是一回事
  - int、uint：长度由CPU的位数决定(32或64位)、
  - uintptr：无符号整型，用于存放一个指针
- 复数：complex64、conplex128
- 字符：byte/uint8、rune
  - uint8/byte代表ASCII的一个字符
  - rune代表一个UTF-8字符，代表中文、韩文等字符。

3. 字符串类型：string
4. 派生类型：
   - 指针类型（pointer）
   - 数组类型：
   - 结构体：struct
   - channel类型
   - 函数类型
   - 切片类型
   - 接口类型（interface）
   - Map类型

c、Python

Python中一切事物皆对象，变量是对对象在内存中的存储和地址的抽象。

变量就是变量，它没有类型；Python使用对象模型来存储数据，构造任何类型的值都是对象。对象都拥有三个特性：身份、类型、值。

身份：每个对象都有一个唯一的身份标识自己，任何对象的身份都可以通过内建函数id()来得到。

类型：对象的类型决定了对象可以保存什么类型的值，可以进行什么样的操作，以及遵循什么样的规则。可以通过type()查看对象类型。返回的是对象而不是简单的字符串。

值：对象表示的数据

Python3中有六个标准数据类型：

1. Number（数字）：int、float、bool、complex(复数)
2. String（字符串）
3. Tuple（元组）
4. List（列表）
5. Set（集合）
6. Dictionary（字典）

- 不可变数据类型：Number、String、Tuple
- 可变数据类型：List、Dictionary、Set

d、JavaScript

JavaScript变量均为对象。当声明一个变量时，就创建了一个新的对象。

JavaScript不区分整数和浮点数，统一使用Number表示。

值类型（基本类型）：String、Number、Boolean、Null、Undefined、Symbol

引用类型（对象类型）：Object、Array、Function、RegExp、Date

```
console.log("------------------typeof-------------------------")
console.log(typeof 1)       // number
console.log(typeof 3.14)    // number
console.log(typeof 'abc')   // string
console.log(typeof "123")   // string
console.log(typeof true)    // boolean
console.log(typeof undefined)   // undefined
console.log(typeof function () {
})   // function
```

```
console.log(typeof [1, 2, 3])     // object
console.log(typeof new Array()) // object
console.log(typeof new Date())  // object
console.log(typeof null)        // object
console.log(typeof new Map())   // object
console.log(typeof new Set())   // object

// typeof 根本无法区分Object类型中的Array、Date、Map、Set等类型
// 使用Object.prototype.toString 可以解决此问题

console.log("------------------constructor.name-------------------------
")
int1 = 1
console.log(int1.constructor.name)       // Number
console.log(3.14.constructor.name)    // Number
console.log('abc'.constructor.name)   // String
console.log("123".constructor.name)   // String
console.log(true.constructor.name)    // Boolean
// console.log(undefined)   // 无法使用constructor.name
console.log(function () {
}.constructor.name)   // Function
console.log([1, 2, 3].constructor.name)     // Array
console.log(new Array().constructor.name) // Array
console.log(new Date().constructor.name)  // Date
// console.log(null)        // 无法使用constructor.name
console.log(new Map().constructor.name)   // Map
console.log(new Set().constructor.name)   // Set
```

1.2、数据类型对比

类型	Java	Go	Python	JavaScript	备注
byte	byte	byte			
bool	bool	bool	<class,bool>		
有符号整数	short/int/long	int8/int16/int32/int64/int/rune	<class,int>		Python bool继承int,可以当成int的0或1
无符号整数	--	uint8/uint16/uint32/uint64/uint/uintptr	--		
浮点数	float/double	float32/float64	<class,float>		
复数	--	complex	<class,complex>	--	
字符	char	byte/uint8/rune	--		Go中字符有2种，uint8/byte代表ASCII的一个字符，rune代表一个UTF-8字符
指针	--	pointer	--	--	
字符串	String	string	<class,str>	String	
空	null	nil	None	null、undefined	
元组	--	--	<class,tuple>	--	
数组	Array	array	--	Array	
List	ArrayList/LinkedList等	slice	<class,list>	List	
Map	HashMap/TreeMap等	map	<class,dict>	Map	
Set	HashSet/TreeSet等	--	<class,set>	Set	
结构体	--	struct	--	--	
函数	成员函数	func（函数和方法）	def	Function	
接口	Interface	interface		--	
类	Class	--	Class	Class	
Channel	--	channel	--	--	

2、变量

2.1、变量声明(定义)

a、Java

```
// 方式一：  数据类型 变量名 = 赋值;
// 方式二：  var 变量名 = 赋值;  // JDK10 局部变量类型推断，可以使用var代替实际类型

String name = "skwqy";
int age = 100;
var name2 = "skwqy";
```

b、GO

```
// 方式一：  var 变量名 数据类型 = 赋值      // 完整形式
// 方式二：  var 变量名 = 赋值              // 类型推断，省略数据类型
// 方式三：  变量名 := 赋值                 // 简短声明，省略var关键字

var name string = "skwqy"
var name = "skwqy"
name := "skwqy"
```

c、Python

```
# Python中变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建
name = 'skwqy'
name2 = "sk"
x = 10
```

d、JavaScript

```
// 方式一： var 变量名 = 赋值
// 方式二： let 变量名 = 赋值 （推荐）
// 方式三： const 变量名 = 赋值 （常量）
// let 声明的变量仅在块级作用域中有效。注意：let不允许在相同的作用域内，重复声明同一个变量。
// ----------------------------------------------
{
    let a = 10;
    var b = 1;
}
a // ReferenceError: a is not defined.
b //1
// ----------------------------------------------
var a = [];
for (var i = 0; i < 10; i++) {
  var c = i;
  a[i] = function () {
    console.log(c);
  };
}
a[6](); // 9
// ----------------------------------------------
var a = [];
for (let i = 0; i < 10; i++) {
  let c = i;
  a[i] = function () {
    console.log(c);
  };
}
a[6](); // 6
```

2.2、常量

常量一旦声明，常量的值就不能改变。

a、Java

```
// Java 常量使用final来修饰
public static final float PI = 3.14;
```

b、Go

```
// Go 常量使用const来修饰
const PI float32 = 3.14
const PI2 = 3.14
```

c、Python

```
# Python 中没有定义常量的关键字，也不存在受编译器保护的常量
# 常量只是程序员之间的默契，不收解析器的保护：使用大写字母和下划线来表示常量
PI = 3.14
```

d、JavaScript

```
const PI = 3.14
```

2.3、常用数据结构

2.3.1、字符串

1、字符串的声明和定义

a、Java

```
// 方式一：字面量定义，常量池
String name = "字符串定义";

// 方式二：new 对象定义，堆中
String name2 = new String("字符串定义2");

// 方式三：局部变量推断,仅限于函数内部定义局部变量 JDK提供该功能
var name3 = "字符串定义3";

// 方式4：Text块 JDK14提供该功能
String names = """
        hello
        skwqy
        周峰
        """;
```

b、Go

```
// 方式一：通过双引号创建字符串
var name string = "字符串"

// 方式二：通过反引号(``)来创建，也称为原始文本。不支持转义字符，可以跨越多行，可以包含除反引号之外的任何字符。
// 通常，它用于在正则表达式或HTML中编写多行消息。
var name2 string = `hello
skwqy`
```

c、Python

Python中没有字符类型，一个字符也是字符串。

```
# 方式一：  单引号，单行,可以包含双引号
name = '我叫"老大哥"'
```

```
# 方式二：  双引号，单行，可以包含单引号
name2 = "请叫我'老大哥'"
# 方式三：三个单引号，多行，可以包含单、双引号
name3 = '''请叫我
"老大哥'你好'"'''
# 方式四：  三个双引号，多行，可以包含单、双引号
name4 = """请叫我
'老大哥'
"你好"
"""

# 原始字符串,字符串不会转义，原样输出
# 方式一：不能包含单引号(')
rname = r'test\t123'
# 方式一：不能包含双引号(")
rname2 = R"test\t123"
# 方式三：可以包含单、双引号
rname3 = r'''姓名'''
# 方式四：可以包含单、双引号
rname4 = R"""姓名"""
```

d、JavaScript

```
// 方式一：单引号，字符串内容可以包含双引号
var name1 = '请叫我"老大哥"'

// 方式二：双引号，字符串内容可以包含单引号
let name2 = "请叫我'老大哥'"

// 方式三：反引号,多行字符串，可以包含单、双引号
const name3 = `请叫我
"老大哥"
'OK'`
```

2、字符子串

a、Java

```java
String oriStr="012345678";
String sub1 = oriStr.substring(1);  // 12345678
String sub2=oriStr.substring(2,5);  // 234
```

b、Go

```go
var oriStr = "012345678"
println(oriStr[1:])  // 12345678
println(oriStr[2:5]) // 234
println(oriStr[:5])  // 01234
```

c、Python

```python
oriStr = "012345678"
print(oriStr[1:])   # 12345678
print(oriStr[2:5])  # 234
print(oriStr[:5])   # 01234
```

d、JavaScript

```javascript
let oriStr = "012345678"
console.log(oriStr.substring(1))        // 12345678
console.log(oriStr.substring(2,5))      // 234
```

3、字符串分割

a、Java

```java
String oriStr="123:456:789";
String[] members = oriStr.split(":");
```

b、Go

```go
findStr := "123:456:789"
arr := strings.Split(findStr, ":")
fmt.Println(arr) // [123 456 789]
```

c、Python

```python
findStr = "123:456:789"
arr = findStr.split(":")
print(arr)  # ['123', '456', '789']
```

d、JavaScript

```javascript
findStr = "123:456:789"
arr = findStr.split(":")
console.log(arr)    // [ '123', '456', '789' ]
```

4、字符串查找

a、Java

```java
String oriStr = "123:456:789:我爱中国";
int index = oriStr.indexOf(":");
System.out.println(oriStr.substring(index));        // :456:789:我爱中国
int lastIndex = oriStr.lastIndexOf(":");
System.out.println(oriStr.substring(lastIndex));    // :我爱中国
```

b、Go

```go
findSubStr := "123:456:789:我爱中国"
index := strings.IndexAny(findSubStr, ":")
fmt.Println(findSubStr[index:]) // :456:789:我爱中国
lastIndex := strings.LastIndexAny(findSubStr, ":")
fmt.Println(findSubStr[lastIndex:]) //:我爱中国
```

c、Python

```
findSubStr = "123:456:789:我爱中国"
index = findSubStr.find(":")
print(findSubStr[index:])  # :456:789:我爱中国
lastIndex = findSubStr.rfind(":")
print(findSubStr[lastIndex:])  # :我爱中国
```

d、JavaScript

```
findSubStr = "123:456:789:我爱中国"
index = findSubStr.indexOf(":")
console.log(findSubStr.substring(index)) // :456:789:我爱中国
lastIndex = findSubStr.lastIndexOf(":")
console.log(findSubStr.substring(lastIndex)) //:我爱中国
```

5、字符替换

a、Java

```

```

b、Go

```

```

c、Python

```

```

d、JavaScript

```

```

2.3.2、数组

Java和Go语言中的数组在创建时需要指定存放数据类型和数组大小，数组一旦创建，其大小是不能改变的。且数组中存放的数据类型也是确定的，虽然Go语言可以创建interface{}数组来存放任何数据类型（包含基本数据类型int8、uint8等）

Python和JavaScript这种动态语言中的数组在创建时，不需要指定大小和数据类型。无需指定大小这种特性更想Java中的List和Go中的切片Slice。

1、数组创建

a、Java

```java
// 方式一:
String[] names = new String[3];
names[0] = "skwqy";
names[1] = "sk";
names[2] = "skwqy2";

String[] names1 = new String[]{"skwqy", "sk", "skwqy2"};
System.out.println(names1.length); // 3

// 方式二:
String[] names2 = {"skwqy", "sk", "skwqy2"};
System.out.println(names2.length);// 3
```

b、Go

```go
// 方式一：指定容量，创建
var names [3]string
names[0] = "小明"
names[1] = "王二狗"
fmt.Println(names)                    // [小明 王二狗 ]
fmt.Println(len(names), cap(names)) // 3 3

// 方式二：指定容量，创建并初始化
var name2 [3]string = [3]string{"王二狗", "李小花", "张三丰"}
```

```
    fmt.Println(name2) // [王二狗 李小花 张三丰]

    // 方式三：指定容量，按照索引进行初始化
    name3 := [3]string{1: "郭靖", 2: "黄蓉"}
    fmt.Println(name3) // [ 郭靖 黄蓉]

    // 方式四：匹配容量，
    name4 := [...]string{"0", "1", "2", "3"}
    fmt.Println(name4)                  // [0 1 2 3]
    fmt.Println(len(name4), cap(name4)) // 4 4

    // 方式五：匹配容量，指定索引
    name5 := [...]string{0: "姓名0", 6: "姓名6"}
    fmt.Println(name5)                  // [姓名0      姓名6]
    fmt.Println(len(name5), cap(name5)) // 7 7
```

c、Python

```
    # Python 中没有数组的概念
```

d、JavaScript

```
    // JavaScript中的数组更向Java中的List，Go中的Slcie，放到List结构中进行对比
```

2、数组遍历

a、Java

```
    String[] names = {"欧阳修", "苏轼", "王安石"};
    // 方式一：
    for (int index = 0; index < names.length; index++) {
        var name = names[index];
        System.out.println(name);
    }
```

```
// 方式二:
for(var name:names){
    System.out.println(name);
}

// 方式三:
Arrays.stream(names).forEach(name-> System.out.println(name));
```

b、Go

```
namess := [...]string{"欧阳修", "苏轼", "王安石"}
// 方式一
for i := 0; i < len(namess); i++ {
  fmt.Println(namess[i])
}

// 方式二
for index, name := range namess {
  fmt.Println(index, name)
}
```

c、Python

```
# Python 中没有数组的概念
```

d、JavaScript

```
// JavaScript中的数组更向Java中的List，Go中的Slcie，放到List结构中进行对比
```

2.3.3、List 切片

Java中的List是定义为接口，jdk包括用户都可以基于该接口实现自己的List实现类。当前Java中实现List接口的数据结构有：

- ArrayList：非线程安全，底层数组数组实现
- LinkedList：非线程安全，底层使用链表实现
- CopyOnWriteArrayList：线程安全

- ……

Go、Python、JavaScript的类List都是编程语言内置的，Go语言的Slice、Python的<class 'list'>、JavaScript的Array都是语言内置的，提供容量动态增长特性，并提供丰富的操作接口。

1、创建

a、Java

```
// 方式一：
var list1 = new ArrayList<>(20);

// 方式二：
var list2 = Arrays.asList("王安石", "苏轼");

// 方式三：
var list3 = List.of("王安石", "苏轼");

// 方式四：
var list4 = Stream.of("王安石", "苏轼").toList();
```

b、Go

```
// 1. 数组
var arr [3]string

// 2.切片
// 方式一：
var s1 []string                                    // 注意没有长度，有长度的是数组，没有长度的是切片slice
fmt.Printf("容量:%d,  长度:%d \n", cap(s1), len(s1)) // 容量:0,  长度:0

// 方式二：和方式一没啥区别，就是创建并初始化
s2 := []string{"王安石", "苏轼"}
fmt.Printf("容量:%d,  长度:%d \n", cap(s2), len(s2)) // 容量:2,  长度:2
fmt.Printf("%T,%T\n", arr, s2)                     // [3]string,[]string
```

```go
// 方式三：指定长度和容量
s3 := make([]string, 3, 8)
fmt.Printf("容量:%d,  长度:%d \n", cap(s3), len(s3)) // 容量:8,  长度:3

// 特殊情况，如果切片类型为interface{}时，可以存放任意的数据类型
s4 := []interface{}{1, 2, 3.14, "王安石"}
fmt.Println(s4) // [1 2 3.14 王安石]
```

c、Python

```python
# Python的List中可以存放任意类型的数据
names = ["王安石", "苏轼", 10, True, [1, 2]]
print(names) # ['王安石', '苏轼', 10, True, [1, 2]]
```

d、JavaScript

```javascript
// JavaScript 的Array可以包含任意数据类型

// 方式一：推荐
let arr = [1, 2, 3.14, 'Hello', null, true, [2, false, 'sk']]
console.log(arr) // [ 1, 2, 3.14, 'Hello', null, true, [ 2, false, 'sk'
] ]

// 方式二：不推荐
let arr2 = new Array()  // 不推荐
console.log(arr2) // []
```

2、遍历

a、Java

```java
var names = Arrays.asList("王安石", "苏轼");
// 方式一：
for (int i = 0, len = names.size(); i < len; i++) {
    System.out.println(names.get(i));
}
```

```
// 方式二:
var ite = names.iterator();
while (ite.hasNext()) {
    System.out.println(ite.next());
}

// 方式三
for(var name:names){
    System.out.println(name);
}
```

b、Go

```
names := []string{"王安石", "苏轼"}
// 方式一
for i := 0; i < len(names); i++ {
  fmt.Println(names[i])
}

// 方式二
for index, name := range names {
  fmt.Println(index, name)
}
```

c、Python

```
names = ["王安石", "苏轼"]

# 方式一
for name in names:
    print(name)

# 方式二: 带有index (0, '王安石') (1, '苏轼')
for name in enumerate(names):
    print(name)
```

```python
# 方式三:
for name in iter(names):
    print(name)

# 方式四
for i in range(len(names)):
    print(i, names[i])
```

d、JavaScript

```javascript
let names = ["王安石", "苏轼"]
// 方式一: for/while 根据数组长度遍历
for (let i = 0, len = names.length; i < len; i++) {
    console.log(names[i])
}

// 方式二:  for...in 可以遍历对象/数组。遍历数组时也会遍历非数字键名，所以不推荐for...in遍历数组
for (let key in names) {
    console.log(names[key])
}

// 方式三: for...of (ES6)
for(let name of names){
    console.log(name)
}

// 方式四: forEach()方法
names.forEach(name=>console.log(name))
```

3、增删改

a、Java

```java
var names = new ArrayList<String>();
// 1. 添加元素
names.add("王安石");  // 方式一：添加到末尾
names.add(0, "苏轼"); // 方式二：添加到指定位置
System.out.println(names); // [苏轼, 王安石]

// 2. 删除元素
names.remove("王安石");  // 方式一：根据元素
System.out.println(names); // [苏轼]
names.remove(0); // 方式二：根据索引
System.out.println(names); // []
names.add(0, "欧阳修");
var ite = names.iterator();
while (ite.hasNext()) {
    String name = ite.next();
    if ("欧阳修".equals(name)) {
        ite.remove();   // 方式三：使用迭代器删除
    }
}
System.out.println(names); // []

// 3. 修改元素
names.add("王安石");  // [王安石]
System.out.println(names);
names.set(0, "欧阳修");
System.out.println(names); // [欧阳修]
```

b、Go

```go
// 1. 增加元素
names = []string{}
names = append(names, "王安石", "曾巩")             // 方式一：在末尾插入
fmt.Printf("指向地址:%p,  切片内容：%s\n", names, names) //指向地
址:0xc00005a060,  切片内容：[王安石 曾巩]
```

```
    names = append(names[:1], "苏轼")                    // 方式二：在指定位置插入，
    插入时会覆盖后面位置元素
    fmt.Printf("指向地址:%p，  切片内容：%s\n", names, names) //指向地
    址:0xc00005a060，  切片内容：[王安石 苏轼]

    // 2. 删除元素
    names = append(names, "柳宗元")
    fmt.Printf("指向地址:%p，  切片内容：%s\n", names, names) // 指向地
    址:0xc0000200c0，  切片内容：[王安石 苏轼 柳宗元]
    names = append(names[:1], names[2:]...)         // 删除第二个元素 [0,1)+[2,
    结束+1)
    fmt.Printf("指向地址:%p，  切片内容：%s\n", names, names) // 指向地
    址:0xc0000200c0，  切片内容：[王安石 柳宗元]

    // 3. 修改元素
    names[1] = "李元龙"
    fmt.Println(names) // [王安石 李元龙]
```

c、Python

```
    names = ["王安石"]
    # 1. 增加元素
    names.append("苏轼")  # 方式一：在末尾添加
    names.insert(0, "柳宗元")  # 方式二：在指定位置添加
    print(names)  # ['柳宗元', '王安石', '苏轼']

    # 2. 删除元素
    names.pop()  # 方式一：在末尾删除
    print(names)  # ['柳宗元', '王安石']
    names.pop(0)  # 方式二：在指定位置删除
    print(names)  # ['王安石']

    # 3. 修改元素
    names[0] = "周润发"
    print(names)  # ['周润发']
```

d、JavaScript

```
names = ["王安石"]
// 1. 增加元素
names.push("苏轼")    // 方式一：在末尾添加元素
names.unshift("柳宗元") // 方式二：在头部增加元素
console.log(names)  // [ '柳宗元', '王安石', '苏轼' ]

// 2. 删除元素
names.pop()  // 方式一：从末尾删除元素
names.shift()   // 方式二：从头部删除数据
console.log(names) // [ '王安石' ]

// 3. 修改元素
names[0] = "周润发"
console.log(names)  // [ '周润发' ]
```

4、排序

a、Java

```
var numbers = Arrays.asList(4, 6, 8, 9, 1, 2);
numbers.sort((e1, e2) -> e1.compareTo(e2));
System.out.println(numbers); // [1, 2, 4, 6, 8, 9]
```

b、Go

```
numbers := []int{4, 6, 8, 9, 1, 2}
sort.Ints(numbers)
fmt.Println(numbers) // [1 2 4 6 8 9]
```

c、Python

```
numbers = [4, 6, 8, 9, 1, 2]
numbers.sort()
print(numbers)  # [1, 2, 4, 6, 8, 9]
```

d、JavaScript

```
numbers = [4, 6, 8, 9, 1, 2, 2]
numbers.sort()
console.log(numbers)    // [ 1, 2, 2, 4, 6, 8, 9 ]
numbers.sort((a, b) => {
    if (a > b) {
        return -1;
    } else if (a < b) {
        return 1
    } else {
        return 0;
    }
})
console.log(numbers)  // [9, 8, 6, 4, 2, 2, 1]
```

2.3.3、Map 字典

Java中的Map是定义为接口，jdk包括用户都可以基于该接口实现自己的Map实现类。当前Java中实现Map接口的数据结构有：

- Hashtable：线程安全；对象锁，所有方法sync
- HashMap：非线程安全；允许存在一条键为null的记录；底层实现为数组+链表(可演变为红黑树)
- TreeMap：非线程安全；不允许key为null；按key进行排序，可以自定义Comparator
- LinkedHashMap：非线程安全；保证插入的顺序
- ConcurrentHashMap：线程安全；采用分段锁
- ConcurrentSkipListMap：线程安全；默认按照key值升序；
- ……

参考： https://www.cnblogs.com/www-123456/p/10890855.html

Go、Python、JavaScript的类Map都是编程语言内置的，Go语言的map、Python的<class 'dict'>、JavaScript的Map都是语言内置的，提供容量动态增长特性，并提供丰富的操作接口。

JavaScript优先使用Map/Set：在试图将对象或数组用作查找的场合，建议优先使用ES6支持的Map/Set替代。Map/Set不需要规避特殊的键名（key），并且可以将字符串之外的其它数据类型也用作key，从而支持更丰富的查找操作。

1、创建

a、Java

```
// 方式一:
final Map<String, String> nameAlis = new HashMap<>();
nameAlis.put("苏轼", "东坡居士");
nameAlis.put("诸葛亮", "孔明");

// 方式二:
final Map<String, String> nameAlis2 = new HashMap<>() {{
    put("苏轼", "东坡居士");
    put("诸葛亮", "孔明");
}};

// 方式三: JDK9
final Map<String, String> nameAlis3 = Map.of("苏轼", "东坡居士", "诸葛亮",
"孔明");
```

b、Go

```
// 方式一: 先声明，在创建
var nameAlias map[string]string // 只有声明，没有初始化，值为nil
nameAlias = make(map[string]string)
nameAlias["苏轼"] = "东坡居士"
nameAlias["诸葛亮"] = "孔明"

// 方式二:
var nameAlias2 = map[string]string{"苏轼": "东坡居士", "诸葛亮": "孔明"}
fmt.Println(nameAlias2) // map[苏轼:东坡居士 诸葛亮:孔明]
```

c、Python

```python
nameAlias = {"苏轼": "东坡居士"}
nameAlias["诸葛亮"] = "孔明"
print(nameAlias) // {'苏轼': '东坡居士', '诸葛亮': '孔明'}
```

d、JavaScript

```javascript
// 方式一：推荐
nameAlias = new Map()
nameAlias.set("苏轼", "东坡居士")
nameAlias.set("诸葛亮", "孔明")
nameAlias["周润发"] = "发哥"

console.log(nameAlias.constructor) // [Function: Map]
console.log(nameAlias)
// Map(2) { '苏轼' => '东坡居士', '诸葛亮' => '孔明', '周润发': '发哥' }
// 2：表示Map对象有两个元素，key分别是苏轼、诸葛亮。 周润发这个属性是作为Object属性，并不是Map的key


// 方式二：推荐
nameAlias = new Map([["苏轼","东坡居士"],["诸葛亮","孔明"]])
nameAlias.set(100, "周润发")
console.log(nameAlias)
// Map(3) { '苏轼' => '东坡居士', '诸葛亮' => '孔明', 100 => '周润发' }

// 方式三： 类似Map，但其实是Object
nameAlias2 = {"苏轼": "东坡居士"}
nameAlias2["诸葛亮"] = "孔明"
console.log(nameAlias2.constructor)  // [Function: Object]
console.log(nameAlias2)  // { '苏轼': '东坡居士', '诸葛亮': '孔明' }
```

2、遍历

a、Java

```java
final Map<String, String> nameAlis = new HashMap<>();
nameAlis.put("苏轼", "东坡居士");
nameAlis.put("诸葛亮", "孔明");

// 方式一：推荐
for (var item : nameAlis.entrySet()) {
    System.out.println(item.getKey() + ":" + item.getValue());
}

// 方式二：不推荐
for (var key : nameAlis.keySet()) {
    System.out.println(key + ":" + nameAlis.get(key));
}

// 方式三：
nameAlis.forEach((key, value) -> System.out.println(key + ":" + value));
```

b、Go

```go
var nameAlias3 = map[string]string{"苏轼": "东坡居士", "诸葛亮": "孔明"}
for k, v := range nameAlias3 {
  fmt.Println(k, ":", v)
}
```

c、Python

```python
nameAlias = {"苏轼": "东坡居士", "诸葛亮": "孔明"}
# 方式一:
for key in nameAlias:
    print(key + ":" + nameAlias[key])

# 方式二 推荐
for key, value in nameAlias.items():
    print(key + ":" + value)
```

d、JavaScript

```javascript
nameAlias3 = new Map([["test", "testValue"]])
nameAlias3.set("苏轼", "东坡居士")
nameAlias3.set("诸葛亮", "孔明")
nameAlias3["周润发"] = "发哥"

// 方式一：能遍历Map中的key和value，无法遍历Object的key、value属性
for (let item of nameAlias3) {
    console.log(item[0] + ":" + item[1])
}

// 方式二：能遍历Map中的key和value，无法遍历Object的key、value属性
nameAlias3.forEach(function (value, key, map) {
    console.log(key + ":" + value)
})

// 无法遍历Map中的key和value，能遍历Object的key、value属性
for (let key in nameAlias3) {
    console.log(key + ":" + nameAlias3[key])
}
```

3、增删查

a、Java

```java
final Map<String, String> nameAlis = new HashMap<>();

// 1. 增加key/value
nameAlis.put("苏轼", "东坡居士"); // 方式一：有就覆盖，没有就增加
nameAlis.putIfAbsent("苏轼", "苏轼"); // 方式二：如果key不存在就添加，存在就不增加
// 方式三：如果不存在，就给他计算出一个值，并put进去
nameAlis.computeIfAbsent("周润发", (key) -> key + "发哥");
// 方式四：如果以及存在，重新计算出一个值，并put进去，key不变
nameAlis.computeIfPresent("苏轼", (key, value) -> key + value);
System.out.println(nameAlis); // {周润发=周润发发哥, 苏轼=苏轼东坡居士}
// 方式五：
//nameAlis.compute("周慧敏",)
// 方式六：
//nameAlis.putAll();

// 2. 删除
nameAlis.remove("周慧敏");
nameAlis.remove("周慧敏", "周慧敏");

// 3. 查询
nameAlis.get("周慧敏");
nameAlis.getOrDefault("周慧敏", "默认值周慧敏");
```

b、Go

```go
var nameAlias4 = map[string]string{}
// 1. 增加
nameAlias4["苏轼"] = "东坡居士" // key不存在则增加，否则修改
nameAlias4["诸葛亮"] = "孔明"

// 2. 删除
delete(nameAlias4, "苏轼")

// 3. 查询
```

```
alias := nameAlias4["诸葛亮"]      // 方式一：直接获取
alias2, ok := nameAlias4["诸葛亮"] // 方式二：判断获取
if ok {
  fmt.Println(alias2)
}
fmt.Println(alias)
fmt.Println(nameAlias4)
```

c、Python

```
# 1. 增加
nameAlias2 = {"诸葛亮": "孔明"}  # 方式一：创建就增加
nameAlias2["苏轼"] = "东坡居士"  # 方式二：添加

# 2. 删除
nameAlias2.pop("诸葛亮")

# 3. 获取
nameAlias2["苏轼"]  # 方式一：不推荐，如果key不存在，会抛TypeError异常
nameAlias2.get("苏轼")  # 方式二：推荐，如果可以不存在，返回None
nameAlias2.get("诸葛亮", "孔明")  # 方式三：推荐，如果key不存在，返回指定的值
```

d、JavaScript

```
nameAlias = new Map()
// 1.增加
nameAlias.set("苏轼", "东坡居士")
nameAlias.set("诸葛亮", "孔明")

// 2.删除
nameAlias.delete("诸葛亮")

// 3.查询
nameAlias.has("苏格兰")  // 判断是是否存在
nameAlias.get("苏格兰")  // 如果不存在，返回 undefined
```

2.3.4、Set 集合

Java中的Set是定义为接口，jdk包括用户都可以基于该接口实现自己的Set实现类。当前Java中实现Set接口的数据结构有：

- HashSet：非线程安全；不允许重复元素；允许null值；元素无序；
- TreeSet：非线程安全；不允许重复元素；允许null值；元素有序（按key排序）；
- LinkedHashSet：非线程安全；保证插入的顺序
- ConcurrentSkipListSet：线程安全；默认按照key值升序；
- ……

Go、Python、JavaScript的类Set都是编程语言内置的，Go语言没有内置set结构、Python的<class 'set'>、JavaScript的Set都是语言内置的，提供容量动态增长特性，并提供丰富的操作接口。

JavaScript优先使用Map/Set：在试图将对象或数组用作查找的场合，建议优先使用ES6支持的Map/Set替代。Map/Set不需要规避特殊的键名（key），并且可以将字符串之外的其它数据类型也用作key，从而支持更丰富的查找操作。

1、创建

a、Java

```
// 方式一
var names=new HashSet<String>();
names.add("苏轼");
names.add("王安石");
names.add("周润发");
names.add("王安石");
System.out.println(names); // [王安石, 周润发, 苏轼]
```

b、Go

```
// Go语言没有内置set这种数据结构
```

c、Python

```python
# 方式一:
names = set()
names.add("苏轼")
names.add("王安石")

# 方式二:
names = set(["周润发", "周慧敏"])
names.add("周润发")
print(names)  # {'周慧敏', '周润发'}
```

d、JavaScript

```javascript
// 方式一:
names = new Set()
names.add("苏轼")
names.add("王安石")
names.add("王安石")
console.log(names)  // Set(2) { '苏轼', '王安石' }

// 方式二:
names = new Set(["苏轼", "周润发"])
console.log(names)  // Set(2) { '苏轼', '周润发' }
```

2、遍历

a、Java

```java
var names=new HashSet<String>();
names.add("苏轼");
names.add("王安石");
names.add("周润发");
names.add("王安石");
```

```java
// 方式一:
for(var name:names){
    System.out.println(name);
}

// 方式二:
var ite = names.iterator();
while(ite.hasNext()){
    var name = ite.next();
    System.out.println(name);
}

// 方式三:
names.forEach(name-> System.out.println(name));

// 方式四:
names.stream().forEach(name-> System.out.println(name));
```

b、Go

```go
// Go语言没有内置set这种数据结构
```

c、Python

```python
# 方式1:
for name in names:
    print(name)

# 方式2:
for name in iter(names):
    print(name)

# 方式3:
for index, name in enumerate(names):
    print(name)
```

d、JavaScript

```
// 方式1:
for(let name of names){
    console.log("方式1: ",name)
}

// 方式2:
names.forEach(name=>console.log("方式2: ",name))

// 方式3:
for(let name of names.keys()){
    console.log("方式3: ",name)
}

// 方式4:
for(let name of names.values()){
    console.log("方式4: ",name)
}

// 方式5:
for(let item of names.entries()){
    console.log("方式5: ",item[0],"-",item[1]) //方式5:  苏轼 - 苏轼
}
```

3、增删改查

a、Java

```
// 1.增加
names.add("苏轼");        // 方式1:
names.addAll(List.of("王安石","柳宗元")); // 方式2:
System.out.println(names); // [王安石, 苏轼, 柳宗元]

// 2.删除
names.remove("苏轼");
```

b、Go

```
// Go语言没有内置set这种数据结构
```

c、Python

```
names = set()
# 1. 添加
names.add("周润发")  # 方式1：单个添加
names.update({"周慧敏", "周星驰"})  # 方式2：批量添加，参数可迭代即可
names.update(["周总理", "周星驰"])
print(names)  # {'周总理', '周慧敏', '周润发', '周星驰'}

# 2. 删除
names.remove("周慧敏")  # 方式1：如果元素不存在时，会引发KeyError错误
names.discard("周慧敏")  # 方式2：元素不存在时，不会引发任何错误
names.pop()  # 方式3：随机删除并返回一个元素，set为空会引发KeyError
names.clear()  # 方式4：移除set中所有元素
```

d、JavaScript

```
names = new Set()
// 1.增加
names.add("周润发")
console.log(names)  // Set(1) { '周润发' }

// 2.删除
names.delete("周润发")
console.log(names)  // Set(0) {}
```

2.3.5、元组

3、字符串和编码

Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。现代操作系统和大多数编程语言都直接支持Unicode。

ASCII编码和Unicode编码的区别：ASCII编码是1个字节，而Unicode编码通常是2个字节。

如果统一成Unicode编码，乱码问题从此消失了。但是，如果你写的文本基本上全部是英文的话，用Unicode编码比ASCII编码需要多一倍的存储空间，在存储和传输上就十分不划算。

所以，本着节约的精神，又出现了把Unicode编码转化为“可变长编码”的 `UTF-8` 编码。UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。如果你要传输的文本包含大量英文字符，用UTF-8编码就能节省空间：

字符	ASCII	Unicode	UTF-8
A	01000001	00000000 01000001	01000001
中	不存在	01001110 00101101	11100100 10111000 10101101

在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。

参考： https://www.liaoxuefeng.com/wiki/1016959663602400/1017075323632896

4、程序结构

4.1、条件判断

4.2、选择结构

4.2、循环结构

二、函数/方法

1、函数和方法

函数：是一段代码，通过名字来进行调用。它能将一些数据（参数）传递进去进行处理，然后返回一些数据（返回值），也可以没有返回值。所有传递给函数的数据都是显式传递的。

方法：也是一段代码，也通过名字来进行调用，但它跟一个对象相关联。方法和函数大致上是相同的，但有两个主要的**不同之处**：

1. 方法中的数据是隐式传递的；

   方法在C++中被称为成员函数。方法和函数的区别就是成员函数和函数的区别；

   Java语言只有方法；只有静态方法和方法的区别；

   C语言中只有函数；
2. 方法可以操作类内部的数据（请记住，对象是类的实例化–类定义了一个数据类型，而对象是该数据类型的一个实例化）

2、函数的定义和调用

2.1、函数/方法定义&调用

a、Java

```
/**
 * 定义方法的语法：
 * 返回值 方法名(参数){
 *      语句块;
 *      return 返回值;
 * }
 */
public class MethodDef {
    // 静态成员变量
    private static String localName = "sk";

    // 对象变量
    private String nickName = "skwqy";
```

```
    // 方式一：静态方法
    public static String hello() {
        return "Hello World";
    }

    // 方式二：成员方法
    public String hi(String name) {
        return "Hi," + name;
    }

    public static void main(String[] args) {
        System.out.println(MethodDef.hello());
        MethodDef methodDef = new MethodDef();
        System.out.println(methodDef.hi("sk"));
    }
}
```

b、Go

```
/**
定义函数的语法
  func funcName(param1 type1, param2 type2)(output1 out1, output2 out2){
    //......
    return value1, value2
  }
  A:func，定义函数的关键字
  B:funcName，函数名字
  C:(),函数的标志
  D:参数列表：形式参数用于接收外部传入函数中的数据
  E:返回值列表：函数执行后返回给调用处的结果。如果只有一个返回值，可以省略返回值的括
号

  方法：method
```

```
    一个方法就是一个包含了接受者的函数，接受者可以是命名类型或结构体类型的一个值或者是一个指针。
    所有给定类型的方法属于该类型的方法集

  语法:
    func (接受者) 方法名(参数列表)(返回值列表){

    }

  总结: method，同函数类似，区别需要有接受者。（也就是调用者）

  方法 vs 函数
    A:意义
      方法: 某个类别的行为功能，需要指定接受者调用
      函数: 一段独立功能的代码，可以直接调用

    B:语法
      方法: 方法名可以相同，只要接受者不同
      函数: 命名不能冲突
*/
// 函数定义1
func hello(name string) string {
  return "hello " + name
}

// 函数定义2
func hello2() (string, int) {
  return "hello sk", 0
}

// Worker 方法定义
// 1.定义一个工人结构体
type Worker struct {
  name string
  age  int
  sex  string
```

```
}

// 2.定义行为方法
func (w Worker) work() {
  fmt.Println(w.name, "在工作...")
}

// 函数/方法调用
func main() {
  fmt.Println(hello("skwqy")) // hello skwqy

  w1 := Worker{name: "王二狗", age: 30, sex: "男"}
  w1.work() // 王二狗 在工作...
}
```

c、Python

```
# 函数定义
def hello(name):
    return "hello " + name


class Person:
    # 构造函数
    def __init__(self, name):
        self.name = name

    # 成员方法
    def hi(self):
        return "hi " + self.name


if __name__ == '__main__':
    print(hello("skwqy"))  # hello skwqy
```

```
    person = Person("sk")
    print(person.hi())  # hi sk
```

d、JavaScript

```
function hello(name) {
    return "hello " + name;
}

class Person {
    constructor(name) {
        this.name = name;
    }

    hi() {
        return "hi " + this.name;
    }
}

// 调用
console.log(hello("skwqy")) // hello skwqy
person = new Person("sk")
console.log(person.hi()) // hi sk
```

3、函数的参数和返回值

3.1、形式参数 vs 实际参数

- 形式参数（形数）：**函数定义**的时候，用于接收外部传入的数据变量。函数中，某些变量的数值无法确定，需要由外部传入数据。
- 实际参数（实数）：**函数调用**的时候，给形参赋值的实际的数值

3.2、参数传递方式（实参-->形参）

- **值传递**：形参是实参的拷贝，改变形参的值并不会影响外部实参的值；从函数调用的角度来说，值传递是单向的（实参-->形参），参数的值只能传入，不能传出；当函数内部需要修改参数，并且不希望这个改变影响调用者时，采用值传递。
- **引用传递**：形参相当于实参的“别名”，对形参的操作其实就是对实参的操作，在引用传递过程中，被调函数的形参虽然也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参的任何操作都影响了主调函数中的实参变量。
- **指针传递**：形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作；

语言	值传递	引用传递	指针传递
Java	支持	支持	--
Go	支持	支持	支持
Python	--	支持	--
JavaScript	支持	支持	--

语言	值传递	引用传递	指针传递
Java	int、float、bool、char	对象传递	--
Go	array、int、float、string、bool、struct、complex	slice、map、chan	参数为指针
Python	--	可变对象(list、dict、set)不可变对象(number、string、tuple)	--
JavaScript	布尔、null、undefined、String、Number	Array、Function、Object	--

3.3、形参分类

形参分类：必选参数（位置参数）、默认参数、可变参数、命名关键字参数、关键字参数

- 必选参数：函数调用时，在指定的形参位置必须要传递对应的实参。
- 默认参数：在指定的形参位置如果未传递对应的实参，此时将形参赋值为一个默认值。
- 可变参数：传入的参数个数是可变的（可以传入0个或任意个参数）。这些参数在函数调用时自动组装成数组或tuple。
- 命名关键字参数：可以传入任意不受限制的关键字参数。
- 关键字参数：运行你传入0个或任意个含参数名的参数（key/value），这些关键字参数在函数内部自动组装称为一个dict。

语言	必选参数	默认参数	可变参数	命名关键字参数	关键字参数
Java	支持	--	支持	--	--
Go	支持	--	支持	--	--
Python	支持	支持	支持	支持	支持
JavaScript	支持	支持	支持	--	--

a、Java

```
// Java 不知默认参数、关键字参数、命名关键字参数
// JDK 中的可变参数
package java.util;
public interface List<E> extends Collection<E> {
  static <E> List<E> of(E... elements) {
        switch (elements.length) { // implicit null check of elements
            case 0:
                @SuppressWarnings("unchecked")
                var list = (List<E>) ImmutableCollections.EMPTY_LIST;
                return list;
            case 1:
                return new ImmutableCollections.List12<>(elements[0]);
            case 2:
```

```
                return new ImmutableCollections.List12<>(elements[0],
    elements[1]);
            default:
                return ImmutableCollections.listFromArray(elements);
        }
    }
}
```

b、Go

```
// Go不支持默认参数、关键字参数、命名关键字参数
// 可变参数
func getSum(num1 int, nums ...int) int {
  sum := num1
  for i := 0; i < len(nums); i++ {
    sum += nums[i]
  }
  return sum
}

func main() {
  fmt.Println(getSum(1, 2, 3, 4)) //10
}
```

c、Python

```
# coding:utf-8
# https://www.liaoxuefeng.com/wiki/1016959663602400/1017261630425888
# --------------------------默认参数-------------------------------
# 默认参数：用来计算x的n次方，由于我们经常计算平方，所以默认第二个参数n的默认值设定为
2
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
```

```
        s = s * x
    return s


print(power(2, 2))  # 4
print(power(2, 3))  # 8


# --------------------------可变参数--------------------------------
# 参数为list
def get_sum(numbers):
    number_sum = 0
    for n in numbers:
        number_sum += n
    return number_sum


print(get_sum([1, 2, 3, 4]))  # 10


# 可变参数版本,和定义一个list或tuple参数相比，仅仅在前面加了一个*号
# 在函数内部，参数numbers接收到的是一个tuple
def get_sum2(*numbers):
    number_sum = 0
    for n in numbers:
        number_sum += n
    return number_sum


print(get_sum2(1, 2, 3, 4))  # 10


# --------------------------关键字参数--------------------------------
# 关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数会在函数内部自动组装为一个dict
def person(name, age, **kw):
```

```
    print("name:", name, "age:", age, "other:", kw)


person("sk", 20)  # name: sk age: 20 other: {}
person("sk", 20, city="NanJing", job="Engineer")  # name: sk age: 20
other: {'city': 'NanJing', 'job': 'Engineer'}
# 和可变参数类似，也可以先组装出一个dict，然后把dict转换为关键字参数闯进去
extra = {'city': 'NanJing', 'jbo': 'Engineer'}
person('sk', 20, **extra)  # name: sk age: 20 other: {'city': 'NanJing',
'jbo': 'Engineer'}


# **extra表示把extra这个dict的所有key-value用关键字参数传入到函数的**kw参数，kw
将获得一个dict
# 注意kw获得的dict是extra的一份拷贝，对kw的改动不会影响到函数外的extra

# -------------------------命名关键字参数--------------------------
# 方式1：如果没有可变参数，必须增加一个*作为特殊分隔符
def person(name, age, *, city, job):
    print(name, age, city, job)


# person('sk', 20) # TypeError: person() missing 2 required keyword-only
arguments: 'city' and 'job'
person('sk', 20, city='NanJing', job="Engineer")  # sk 20 NanJing
Engineer


# 方式2：如果有可变参数，跟着的命名关键字参数就不再需要一个特殊的分隔符*了
def person(name, age, *args, city, job):
    print(name, age, city, job)


person('sk', 20, 20, 20, city='NanJing', job='Engineer')  # sk 20
NanJing Engineer
```

```python
# 方式3：命名关键字可以有缺省值
def person(name, age, *, city='NanJing', job):
    print(name, age, city, job)


person('sk', 20, job='Engineer')  # sk 20 NanJing Engineer


# --------------------------参数组合---------------------------
# 在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数、命名关键字参数，这5中参数都可以组合使用。
# 但是必须注意参数定义的顺序必须是：必选参数、默认参数、可变按树、命名关键字参数、关键字参数；

# 必选参数、默认参数、可变参数、关键字参数
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)


# 必须参数、默认参数、命名关键字参数、关键字参数
def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)


f1(1, 2)  # a = 1 b = 2 c = 0 args = () kw = {}
f1(1, 2, c=3)  # a = 1 b = 2 c = 3 args = () kw = {}
f1(1, 2, 3, 'a', 'b')  # a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
f1(1, 2, 3, 'a', 'b', x=99)  # a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
f2(1, 2, d=99, ext=None)  # a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}

# 最神奇的是通过一个tuple和dict，你也可以调用上述函数
print("------------------------------------")
args = (1, 2, 3, 4)
kw = {'d': 99, 'x': '#'}
```

```
f1(*args, **kw)  # a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x':
'#'}
args = (1, 2, 3)
kw = {'d': 88, 'x': '#'}
f2(*args, **kw)  # a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

d、JavaScript

```
// ----------------------------默认参数--------------------------------
function log(x, y = 'World') {
    console.log(x, y);
}

log('Hello') // Hello World
log('Hello', 'China') // Hello China
log('Hello', '') // Hello

// 注意1：参数变量是默认声明的，所以不能用let或const再次声明。
// 注意2：使用参数默认值时，函数不能有同名参数。
// 注意3：定义了默认值的参数，应该是函数的尾参数
// 注意4：参数默认值不是传值的，而是每次都重新计算默认值表达式的值。也就是说，参数默
认值是惰性求值的。
let x = 99;

function foo(p = x + 1) {
    console.log(p);
}

foo() // 100

x = 100;
foo() // 101

// ----------------------------可变(rest)参数-----------------------------
---
```

```
console.log("-----------------------------------")

function add(...values) {
    let sum = 0;
    for (let val of values) {
        sum += val;
    }
    return sum;
}

console.log(add(2, 5, 3)) // 10
// 注意：rest 参数之后不能再有其他参数（即只能是最后一个参数）

function add2(first = 0, ...values) {
    let sum = first;
    for (let val of values) {
        sum += val;
    }
    return sum;
}

console.log(add2(2, 5, 3)) // 10
```

3.2、返回值

语言	基本类型	对象/struct	函数	指针	多个返回值	默认返回值
Java	支持	支持	--	--	--	--
Go	支持	支持	支持	支持	支持	--
Python	--	支持	支持	--	--	None
JavaScript	--	支持	支持	--	--	undefined

说明：Java虽然不支持返回函数，但是可以很方便返回实现单个函数的对象（Lambda）来代替函数。

Python、JavaScript的返回值不需要声明类型。

a、Java

```
public class ReturnValue {
    // 返回基本数据类型、返回普通对象类型---省略

    // 模拟返回函数
    public Supplier<String> getSupplier() {
        return new Supplier<String>() {
            @Override
            public String get() {
                return "Hello World.";
            }
        };
    }

    public Supplier<String> getSupplier2() {
        return () -> "Hello skwqy";
    }

    public static void main(String[] args) {
        ReturnValue ins = new ReturnValue();
        System.out.println(ins.getSupplier().get());    // Hello World.
        System.out.println(ins.getSupplier2().get());   // Hello skwqy
    }

}
```

b、Go

```go
func main() {
  // 返回对象
  p := getPerson("sk", 20)
  fmt.Println(p) // {sk 20}

  // 返回指针
  ptr := getPerson2("skwqy", 20)
  fmt.Println(ptr) //&{skwqy 20}

  // 返回函数
  fun := increment()
  fmt.Printf("%T\n", fun) // func() int

  // 返回多个值
  zhouchang, area := rectangle(3, 4)
  fmt.Println(zhouchang, area) // 14 12
}

// 返回对象/struct
func getPerson(name string, age int) Person {
  return Person{name: name, age: age}
}

// 返回指针
func getPerson2(name string, age int) *Person {
  return &Person{name: name, age: age}
}

// 返回函数
func increment() func() int { // 外层函数
  // 1.定义了一个局部变量
  i := 0
  // 2.定义了一个匿名函数，给变量自增并返回
```

```
    fun := func() int {
        i++
        return i
    }
    return fun
}

// 返回多个结果
func rectangle(len, wid float64) (float64, float64) {
    peri := (len + wid) * 2
    area := len * wid
    return peri, area
}

type Person struct {
    name string
    age  int
}
```

c、Python

```
# 返回对象：好像返回的是基本类型，其实返回的还是对象，Python中一切皆对象
def get_age():
    return 10

age = get_age()
print(type(age))  # <class 'int'>


def get_names():
    return ["sk", "ssk"]
print(type(get_names()))  # <class 'list'>

print("-------------返回函数-----------------")
def inc(x=0):
```

```
    def fun():
        return x + 1

    return fun


fun1 = inc()
print(type(fun1))  # <class 'function'>
i = fun1()
print(type(i))  # <class 'int'>

print("-------------默认返回值-----------------")
def get_name():
    pass
print(get_name())  # None
```

d、JavaScript

```
// ----------------------返回对象------------------------
function getAge() {
    return 10
}

age = getAge()
console.log(age.constructor.name)     // Number
console.log(3.14.constructor.name)    // Number

function getNames() {
    return ["sk", "ssk"]
}

names = getNames()
console.log(names.constructor.name)   // Array
```

```
// ----------------------默认返回-------------------------
function get_default() {

}

default1 = get_default()
console.log(default1)   // undefined

// ----------------------返回函数-------------------------
function inc() {
    let i = 0
    return function fun() {
        return i++
    }
}

fun1 = inc()
console.log(fun1.constructor.name)  // Function
console.log(fun1()) // 0
console.log(fun1()) // 1
```

4、变量作用域与解构赋值

4.1、变量作用域

JavaScript全局作用域：不在任何函数内定义的变量就具有全局作用域。实际上，JavaScript默认有一个全局对象window，全局作用域的变量实际上被绑定到window的一个属性。

ES6引入新关键字let，用let替代var可以声明一个块级作用域的变量。

语言	全局作用域	类作用域	实例作用域	函数作用域	块作用域	
Java	--	支持	支持	支持	支持	
Go	支持	--	--	支持	支持	
Python	支持	支持	支持	支持	支持	
JavaScript	支持	--	--	支持	支持(ES6)	

4.2、变量解构赋值

d、JavaScript

JavaScript ES6标准引入了解构赋值，可以同时对一组变量进行赋值。

解构赋值其实就是将一个序列(List、Array)按照位置和层次，Map按照key值和层次分别赋值给几个变量。

```
// https://es6.ruanyifeng.com/#docs/destructuring
//--------------序列(List、Array)按位置解构---------------------
let arr = ['Hello', 'JavaScript', 'ES6']
let [x, y, z] = arr
console.log(x, y, z)  // Hello JavaScript ES6  // 全量获取
let [, yy] = arr
console.log(yy)  // JavaScript  // 按需获取

let [xxx, [, yyy]] = ['Hello', ['JavaScript', 'ES6']]
console.log(xxx, yyy)    // Hello ES6    // 按层次、按需获取

//--------------对象按key解构---------------------
console.log("-----------------------------------------------")
let person = {name: '小明', age: 20, gender: 'male'}
let {name, age, father} = person
console.log(name, age, father)   // 小明 20 undefined

let person2 = {name2: '小明', age: 20, gender: 'male', address: {city:
'NanJing', zipcode: '025'}}
```

```
let {name2, address: {city}} = person2
console.log(name2, city)    // 小明 NanJing

// --------------------字符串解构赋值--------------------------
const [a, b, c, d] = 'HELLO'
console.log(a, b, c, d)    // H E L L

// --------------------数值或布尔解构赋值-----------------------------
let {toString: s} = 123
console.log(s === Number.prototype.toString)  // true
let {toString: ss} = true
console.log(ss === Boolean.prototype.toString)    // true

// ---------------------函数参数解构--------------------------
function add([x, y]) {
    return x + y;
}

console.log(add([1, 2])); // 3


// -------------------------使用场景-----------------------------
console.log("------------------------------------------")
// 场景1：交换两个变量值
let num1 = 1, num2 = 2;
[num1, num2] = [num2, num1]
console.log(num1, num2) // 2 1

// 场景2：从函数中返回多个值
function example() {
    return [1, 2, 3];
}

let [aa, bb, cc] = example();

// 返回一个对象
```

```
function example2() {
    return {
        foo: 1,
        bar: 2
    };
}

let {foo, bar} = example2();

// 场景3：函数参数的定义
function buildDate({year, month, day, hour = 0, minute = 0, second = 0})
{
    return new Date(year + '-' + month + '-' + day + ' ' + hour + ':' +
minute + ':' + second);
}

console.log(buildDate({year: 2011, month: 4, day: 23})) // 2011-04-
22T16:00:00.000Z
// 场景4：提取JSON数据

// 场景5：函数参数的默认值,同函数定义中，指定了minute、second等的默认值
// 指定参数的默认值，就避免了在函数体内部再写var foo = config.foo || 'default
foo';这样的语句。

// 场景6：遍历Map结构
const map = new Map();
map.set('first', 'hello');
map.set('second', 'world');

for (let [key, value] of map) {
    console.log(key + " is " + value);
}
// 场景7：输入模块的指定方法
// const {SourceMapConsumer, SourceNode} = require("source-map");
```

5、匿名函数

匿名函数在Java中叫Lambda表达式，在JavaScript中叫箭头函数

5.1、匿名函数定义

a、Java

```
public class LambdaReg {
    /*
    Lambda表达式语法：
    (参数1,参数2,...) -> {代码块}
        注意：
            1、Lambda表达式由形参列表、->、代码块组成，->用来将参数和代码块分隔开，不能省略；
            2、当只有一个参数时，小括号()可以省略；
            3、当只有一条语句时，大括号{}可以省略；代码块只有一条return语句时，可省略return关键字；
            4、Lambda表达式的目标类型必须是函数式接口(@FunctionalInterface)；函数式接口代表只能包含一个抽象方法的接口；
     */

    public static void main(String[] args) {
        // 省略return关键字，省略大括号，实现函数式接口：Supplier
        CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> "Hello");

        // 省略(),实现函数式接口：Function
        CompletableFuture<String> cf2 = cf.thenApply(str -> {
            System.out.println(str);
            return str + " World";
        });

        // 省略小括号、省略大括号，实现函数式接口：Consumer
```

```
            cf2.thenAccept(str -> System.out.println(str));
        }
    }
```

b、Go

```
func main() {
  // 将匿名函数赋值给变量
  func4 := func(a, b int) int {
    return a + b
  }
  fmt.Printf("%T\n", func4) //func(int, int) int
  res1 := func4(10, 100)
  fmt.Println(res1) // 110

  // 函数调用时，直接定义匿名函数
  res3 := oper(200, 300, func(a int, b int) int {
    return a + b
  })
  fmt.Println(res3) // 500
}

// 和Python、JavaScript一样，可以直接将函数作为普通参数，但是却能看到函数声明，代码阅读起来比Python、JavaScript清洗
func oper(a, b int, fun func(int, int) int) int {
  return fun(a, b)
}
```

c、Python

```
# coding:utf-8
"""
匿名函数语法:
    lambda [arg1 [,arg2, ... argN]] : expression
    注意:
```

```
        1、匿名函数由 关键字lambda、形参列表、冒号(:)、表达式 四部分组成；形参如果没有可以省略；
        2、冒号(:)是形参和表达式的分隔符
"""

def build(x, y):
    return lambda: x * x + y * y

lambda1 = build(2, 3)
print(type(lambda1))    # <class 'function'>
print(lambda1())    # 13
```

d、JavaScript

```
/*
 Lambda表达式语法：
    (参数1,参数2,...) => {代码块}
        注意：
            1、Lambda表达式由 形参列表、=>、代码块组成，=>用来将参数和代码块分隔开，不能省略；
            2、当只有一个参数时，小括号()可以省略；
            3、当只有一条语句时，大括号{}可以省略；代码块只有一条return语句时，可省略return关键字；
 */

let fn = x => x * x
console.log(fn.constructor.name)    // Function
console.log(fn(2))  // 4
```

6、闭包

7、高阶函数

高阶函数：如果一个函数的参数可以接受另一个函数作为参数，那么这种函数就称之为高阶函数。

7.1、高阶函数应用

7.1.1、Java

a、Stream

a.1、流创建

```
Stream<Integer> stream = List.of(1, 2, 3).stream();
Stream<String> stream1 = new HashSet<String>().stream();
IntStream stream2 = Arrays.stream(new int[]{1, 2, 3});
Map<String, String> map = new HashMap<>();
Stream<String> stream3 = map.keySet().stream();
Stream<String> stream4 = map.values().stream();
```

a.2、中间操作

操作	说明
Stream filter(Predicate)	对流中的元素进行过滤
Stream map(Function)	对流中的元素进行映射
Stream peek(Consumer)	用作debug，peek中操作并不会影响流中的元素，它的参数是Consumer，没有返回。
Stream flatMap(Function)	对流中的元素进行扁平化，扁平化之前，流中元素可能是数组
Stream limit(long)	截取指定个数的流
Stream skip(long)	跳过指定个数的的流
Stream distinct()	元素去重，根据元素的equals方法进行去重
static Stream concat(Stream a,Stream b)	合并a、b两个流为一个流
Stream sorted()	将流中的元素进行排序
Stream dropWhile(Predicate)	从流中删除满足条件的元素，直到遇到第一个不满足条件的终止。JDK9新增
Stream takeWhile(Predicate)	从流中获取满足条件的元素，直到遇到第一个不满足条件时终止。JDK9新增

a.3、终结操作

操作	说明
void forEach(Consumer)	对流中的每个元素执行指定操作，执行顺序不确定，有助于并行操作中提升性能
void forEachOrdered(Consumer)	对流中的每个元素执行指定操作，按照流中元素顺序执行。
T reduce(BinaryOperator)	对流中的元素进行计算，最终生成一个值。count、min、max都是reduce的一种计算方式。
long count()	返回流中元素个数
T max(Comparator)	返回流中最大的元素
T min(Comparator)	返回流中最小的元素
R collect(Collector)	把结果收集到集合中
List toList()	把结果收集到List中
Object[] toArray()	把结果收集到数组中
boolean allMatch(Predicate)	流中的元素是否全部匹配谓词Predicate
boolean anyMatch(Predicate)	流中的元素是否有匹配谓词Predicate
boolean noneMatch(Predicate)	流中元素是否都未匹配谓词
Optional findAny()	选择流中的任何元素，该方法行为是不确定的，该方法有助于在并行操作中获得最大性能
Optional findFirst()	选择第一个元素

b、CompletableFuture

```
// https://zhuanlan.zhihu.com/p/344431341
// 模板三件套
CompletableFuture<Void> thenAccept(Consumer); //同步操作
CompletableFuture<Void> thenAcceptAsync(Consumer);  // 异步操作，使用默认线程池
CompletableFuture<Void> thenAcceptAsync(Consumer,Executor); // 异步操作，使用自定义线程池
// 后续都以 thenAcceptAsync(Consumer) 这种形式为例
```

b.1、实例化CompletableFuture

方法	说明
static CompletableFuture supplyAsync(Supplier)	可以返回异步线程执行之后的结果。
static CompletableFuture supplyAsync(Supplier,Executor)	同上，并使用自定义的线程池进行异步执行。
static CompletableFuture runAsync(Runnable)	只是异步执行线程任务。
static CompletableFuture runAsync(Runnable,Executor)	同上，使用自定义的线程池执行异步任务。

b.2、同步获取结果

方法	说明
T get()	获取任务结果，阻塞直到任务完成
T get(long, TimeUnit)	同步获取任务结果，设置对应的超时时间。
T getNow(T valueIfAbsent)	获取任务结果，如果计算结果不存在或Now时刻没有完成任务，则给定一个默认值。
T join()	join()与get()的区别在于join返回计算的结果或抛出一个unchecked异常，而get返回一个具体的异常。

b.3、计算完成后操作（CompletableFuture 简写为CF）

方法	说明
CF whenComplete(BiConsumer<T, Throwable>)	能处理异常和返回结果（结果类型无法自定义）。三件套
CF handle(BiFunction<T, Throwable, U>)	能处理异常和返回结果（结果类型能自定义）。三件套
CF thenApplyAsync(Function<T, U>	无法处理异常，但能返回结果（结果类型可自定义）。三件套
CF thenAcceptAsync(Consumer)	只能作最终结果的消费，类似流的终端操作。三件套
CF exceptionally(Function<Throwable, T> fn)	专门处理异常，且必须return一个返回值。

b.4、组合

方法	说明
CF thenApplyAsync(Function<T, U>	将两个异步计算合并为一个，两个异步计算相互独立，第二依赖第一个结果。三件套
CF thenCompose(Function<T, CF> fn)	将两个异步计算合并为一个，两个异步计算相互独立，互不依赖。三件套
CF thenCombineAsync(CS other, BiFunction<T,U,V> fn)	将两个异步计算合并为一个，两个异步计算相互独立，互不依赖。三件套
CF allOf(CF<?>... cfs)	等待Feture集合中的所有任务都完成。
CF anyOf(CF<?>... cfs)	仅等待Feture集合中最快结束的任务完成，并发挥它的结果。

7.1.2、Go

```
```

7.1.3、Python

```
```

7.1.4、JavaScript

a、map/reduce

```
//--------------------map--------------------------
function pow(x) {
    return x * x
}

let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
let res = arr.map(pow)
console.log(res)    //[1,  4,  9, 16, 25, 36, 49, 64, 81]

// ---------------------reduce-----------------------------
function add(x, y) {
    return x + y
}

res2 = arr.reduce(add)
// [x1, x2, x3, x4].reduce(f) = f(f(f(x1,x2),x3),x4)
console.log(res2)  // 45

// ------------------------使用场景简介-------------------------
// 将[1,3,5,7,9]变成13579
let arr2 = [1, 3, 5, 7, 9]
res3 = arr2.reduce(function (x, y) {
    return x * 10 + y
```

```
})
console.log(res3)   // 13579
res31 = arr2.reduce((x, y) => 10 * x + y)
console.log(res31)   // 13579

// ['1','2','3'] 字符串变整数
let arr4 = ['1', '2', '3']
res4 = arr4.map(parseInt)
console.log(res4)   // [ 1, NaN, NaN ]

res41 = arr4.map((x) => parseInt(x))
console.log(res41)  // [ 1, 2, 3 ]
```

b、forEach、filter

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// 遍历：无返回值
arr.forEach(x => console.log(x))

// 过滤：返回true保留、返回false丢弃
res = arr.filter(x => x % 2 === 0)
console.log(res)  // [ 2, 4, 6, 8, 10 ]
```

c、sort

```
let arr = [2, 1, 9, 6, 4, 8, 5, 3, 7]

let res1 = arr.sort((a, b) => {
    if (a > b) {
        return 1
    } else if (a < b) {
        return -1
    } else {
        return 0
```

```
    }
})
console.log(res1)   // [1,2,3,4,5,6,7,8,9]

let res2 = arr.sort((a, b) => {
    if (a > b) {
        return -1
    } else if (a < b) {
        return 1
    } else {
        return 0
    }
})
console.log(res2)   // [9,8,7,6,5,4,3,2,1]
```

d、Array

```
// Array还有一些其它的高阶函数
// --------------------every-----------------------
let arr = ['Apple', 'pear', 'orange']
// 判断是否每个元素的length都大于0
let isNotEmpty = arr.every(x => x.length > 0)
console.log(isNotEmpty) // true

// --------------------find-----------------------
// 找到符合条件的第一个元素
let firstLowerWorld = arr.find(x => x.toLowerCase() === x)
console.log(firstLowerWorld)    // pear

// --------------------findIndex-----------------------
let firstLowerWorldIdx = arr.findIndex(x => x.toLowerCase() === x)
console.log(firstLowerWorldIdx) // 1
```

7.2、

8、递归函数

9、生成器(generator)

三、面向对象

面向对象的特点：封装、继承、多态

1、封装

封装的目的：隐藏程序的具体实现细节；让使用者看到刚刚好的数据和行为（接口）

封装的含义：

- 属性和行为捆绑：把对象的属性和行为看成一个密不可分的整体，将这两者“封装”在一个不可分割的独立单元（类、对象）中；
- 信息隐藏：把不需要外界知道的信息隐藏起来，有些对象的属性及行为允许外界用户知道或使用，但不允许更改；另外一些属性或行为，则不允许外界知晓，只允许使用对象的功能，而尽可能隐藏对象的功能实现细节。

封装的优点：

- 良好的封装能够减少耦合，符合程序设计追求的“高内聚，低耦合”。
- 能够自由的控制被隐藏的属性和行为，这些操作对使用者都是透明的。

封装手段：

- 接口：提供给使用者；没有甚至很要包含具体的实现细节；
- 类：具体的实现；包含所有的实现细节；包含并控制数据和行为；
- 对象：捆绑了自己实例的数据和类模板的行为，将其融为一体，对外提供真正的服务

2、继承

继承的目的：代码复用

继承的特点：

- 继承可以获得父类的某些属性和行为；
- 可以选择覆盖（重写）父类的行为；
- 可以添加自己特有的属性和行为；
-

3、多态

多态的目的：提升了编程的灵活性，简化使用者使用，使用者不比关注谁来提供服务（具体实现），只用关注能使用声明服务（接口）

多态发送在运行运行期间，即类型多态，不同类型的对象实体具有统一接口，相同的消息给予不同的对象会引发不同的动作。

语言	面向对象手段	优点	缺点
Java	Interface、Class、implement、extend	都是优点	侵入式接口
Go	interface、struct、method	非侵入式接口	封装不足，需要通过Interface、struct组合模拟；继承不足，需要通过struct成员模拟；
Python	Class、extend()		
JavaScript	class、extends、prototype(实际)		

4、面向对象实践

a、Java

```
public class Oop {
    public static interface IPerson {
        void eat();

        void study();
    }

    public static class Person implements IPerson {
        protected final String name;
        private final int age;

        public Person(String name, int age) {
            this.name = name;
            this.age = age;
        }

        @Override
        public void eat() {
            System.out.println("Person....eat....");
        }

        @Override
        public void study() {
            System.out.println("Person....study....");
        }
    }

    public static class Student extends Person {
        private final String school;

        public Student(String name, int age, String school) {
            super(name, age);
```

```
            this.school = school;
        }

        @Override
        public void study() {
            System.out.println("Strudent(" + name + ").....study....in "
+ school);
        }
    }

    public static void main(String[] args) {
        IPerson person = new Person("王二狗", 30);
        person.eat();   // Person....eat....
        person.study(); // Person....study....
        person = new Student("张小花", 20, "银城小学");
        person.eat();   // Person....eat....   // 继承父类的
        person.study(); // Strudent(张小花)....study....in 银城小学  // 覆
盖父类的
    }
}
```

b、Go

```
func main() {
  /*
    OOP中的继承性:
      如果两个类(class)存在继承关系，其中一个是子类，另一个作为父类，那么:
      1.子类可以直接访问父类的属性和方法
      2.子类可以新增自己的属性和方法
      3.子类可以重写父类的方法(orverride，就是将父类已有的方法，重新实现)
    Go语言的结构体嵌套:
      1.模拟继承性: is - a
        type A struct{
          field
        }
        type B struct{
```

```
            A //匿名字段
          }
        2.模拟聚合关系：has - a
          type C struct{
            field
          }
          type D struct{
            c C //聚合关系
          }
    */
    //1.创建Person类型
    var p1 IPerson = Person{name: "王二狗", age: 30}
    p1.eat()   // 父类的方法，吃窝窝头。。
    p1.study() // 父类的方法，学生学习啦。。。

    //2.创建Student类型
    var s1 IPerson = Student{Person{"Ruby", 18}, "千锋教育"}

    s1.eat()   //子类重写的方法：吃炸鸡喝啤酒。。 // 此时子类没有该方法时，就访问匿名字段的
    s1.study() //子类重写的方法，学生学习啦。。。
}

//1.定义一个"父类"
type Person struct {
    name string
    age  int
}

//2.定义一个"子类"
type Student struct {
    Person //结构体嵌套，模拟继承性
    school string
}

//3.方法
```

```
func (p Person) eat() {
  fmt.Println("父类的方法，吃窝窝头。。")
}
func (p Person) study() {
  fmt.Println("父类的方法，学生学习啦。。。")
}

func (s Student) study() {
  fmt.Println("子类重写的方法，学生学习啦。。。")
}

func (s Student) eat() {
  fmt.Println("子类重写的方法：吃炸鸡喝啤酒。。")
}

type IPerson interface {
  eat()
  study()
}
```

c、Python

```
# coding:utf-8

class Person:
    # 构造方法
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def eat(self):
        print("Person(", self.name, ")....eat.....")

    def study(self):
        print("Person.....study.......")
```

```python
class Student(Person):

    def eat(self):
        print("Student(", self.name, ")....eat.....")

    def study(self):
        print("Student.....study.......")


person = Person("sk", 30)
person.eat()  # Person( sk )....eat.....
person.study()  # Person.....study.......

student = Student("skwqy", 30)
student.eat()  # Student( skwqy )....eat.....
student.study()  # Student.....study.......
```

d、JavaScript

```javascript
class Person {
    // 构造函数
    constructor(name, age) {
        this.name = name
        this.age = age
    }

    // 成员方法，定义时不需要关键字function
    eat() {
        console.log("Person(", this.name, ") .....eat......")
    }

    study() {
        console.log("Person(", this.name, ") .....study......")
    }
}
```

```
class Student extends Person {
    eat() {
        console.log("Student(", this.name, ") .....eat......")
    }

    study() {
        console.log("Student(", this.name, ") .....study......")
    }
}

let person = new Person("sk", 30)
person.eat()    // Person( sk ) .....eat......
person.study()  // Person( sk ) .....study......

let student = new Student("skwqy", 29)
student.eat()   // Student( skwqy ) .....eat......
student.study() // Student( skwqy ) .....study......
```

反射

四、并发编程

单元测试