



Scala语言核心编程

-函数式编程基础

尚硅谷研究院

Scala核心编程

函数式编程基础



● 函数式编程内容及授课顺序说明

函数式编程内容

➤ 函数式编程**基础**

- 1) 函数定义/声明
- 2) 函数运行机制
- 3) 递归//难点 [最短路径，邮差问题，迷宫问题, 回溯]
- 4) 过程
- 5) 惰性函数和异常

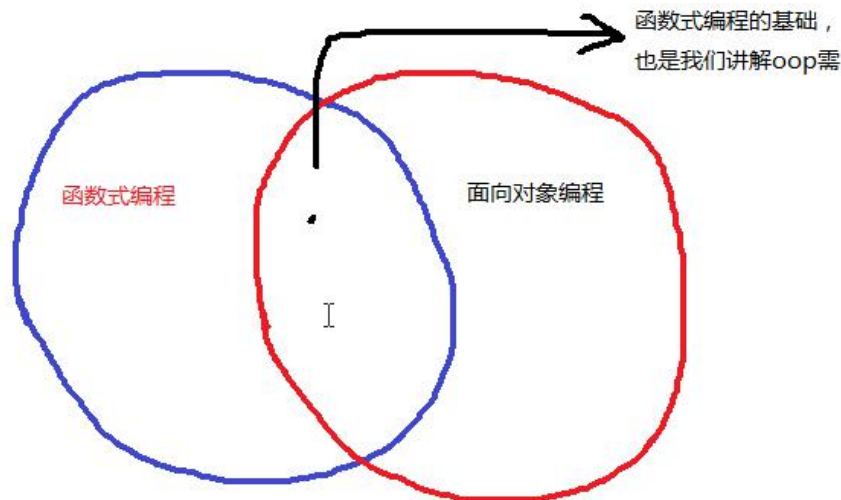
➤ 函数式编程**高级**

- 1) 值函数(函数字面量)
- 2) 高阶函数
- 3) 闭包
- 4) 应用函数
- 5) 柯里化函数，抽象控制...

- 函数式编程内容及授课顺序说明

函数式编程授课顺序

- 1) 在scala中，函数式编程和面向对象编程融合在一起，学习函数式编程需要oop的知识，同样学习oop需要函数式编程的基础。[矛盾]
- 2) 关系如下图:



- 3) 授课顺序： 函数式编程基础->面向对象编程->函数式编程高级



● 函数式编程介绍

几个概念的说明

在学习Scala中将方法、函数、函数式编程和面向对象编程明确一下：

- 1) 在scala中，**方法**和**函数**几乎可以等同(比如他们的定义、使用、运行机制都一样的)，只是函数的使用方式更加的灵活多样。
- 2) **函数式编程**是从编程方式(范式)的角度来谈的，可以这样理解：函数式编程把函数当做一等公民，**充分利用函数、支持的函数的多种使用方式**。

比如：

在Scala当中，函数是一等公民，像变量一样，既可以作为函数的参数使用，也可以将函数赋值给一个变量.，函数的创建不用依赖于类或者对象，而在Java当中，函数的创建则要依赖于类、抽象类或者接口。

- 3) **面向对象编程**是以对象为基础的编程方式。
- 4) 在scala中函数式编程和面向对象编程融合在一起了。

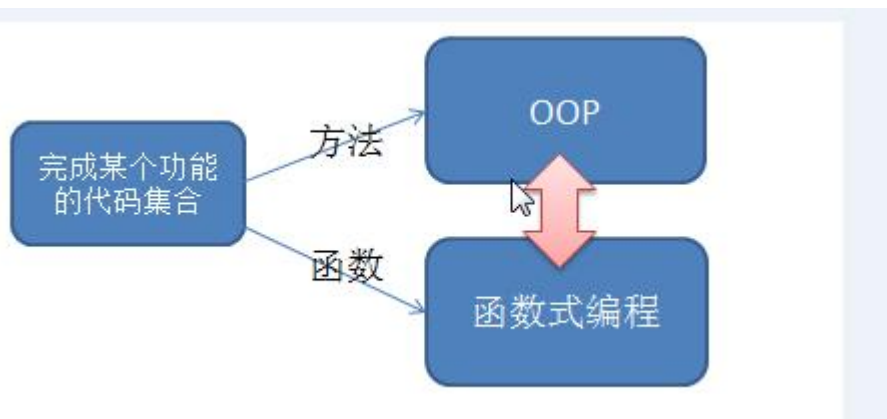
- 函数式编程介绍

几个概念的说明

在学习Scala中将方法、函数、函数式编程和面向对象编程关系分析图：



方法函数函数式编程



- 函数式编程介绍

- "函数式编程"是一种"编程范式"（programming paradigm）。
- 它属于"结构化编程"的一种，主要思想是把运算过程尽量写成一系列嵌套的函数调用。
- 函数式编程中，将函数也当做数据类型，因此可以接受函数当作输入（参数）和输出（返回值）。
- 函数式编程中，最重要的就是函数。

• 为什么需要函数

请大家完成这样一个需求: (学习技术套路)
输入两个数,再输入一个运算符(+,-), 得到结果.。


先使用传统的方式来解决, 看看有什么问题没有?

- 1) 代码冗余
- 2) 不利于代码的维护



函数

抽取功能, 形成代码



```
val n1 = 10
val n2 = 20
var oper = "-"
if (oper == "+") {
    println("res=" + (n1 + n2))
} else if (oper == "-") {
    println("res=" + (n1 - n2))
}
```

```
println("-----做了其他的工作...")
val n3 = 10
val n4 = 20
oper = "-"
if (oper == "+") {
    println("res=" + (n1 + n2))
} else if (oper == "-") {
    println("res=" + (n1 - n2))
}
```




- 函数介绍

为完成某一功能的程序指令(语句)的集合,称为函数。



● 函数的定义

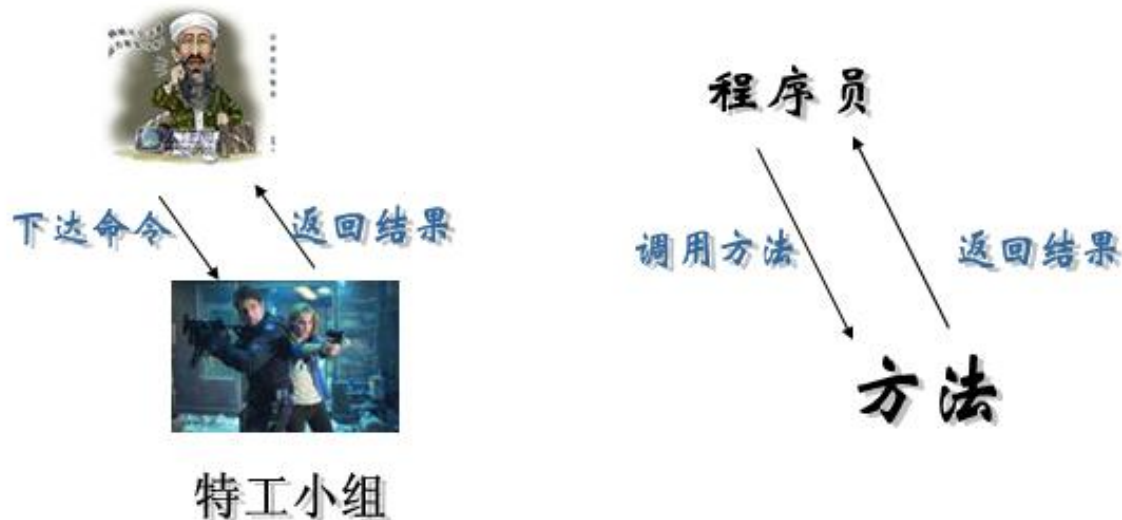
基本语法

```
def 函数名 ([参数名: 参数类型], ...) [: 返回值类型] = {  
    语句...  
    return 返回值  
}
```

- 1) 函数声明关键字为def (definition)
- 2) **[参数名: 参数类型], ...**: 表示函数的输入(就是参数列表), 可以没有。如果有, 多个参数使用逗号间隔
- 3) 函数中的语句: 表示为了实现某一功能代码块
- 4) 函数可以有返回值, 也可以没有
- 5) 返回值形式1: **: 返回值类型 =**
- 6) 返回值形式2: **= 表示返回值类型不确定, 使用类型推导完成**
- 7) 返回值形式3: **表示没有返回值, return 不生效**
- 8) **如果没有return, 默认以执行到最后一行的结果作为返回值**

- 函数-调用机制

➤ 如何理解方法这个概念,给大家举个通俗的示例:



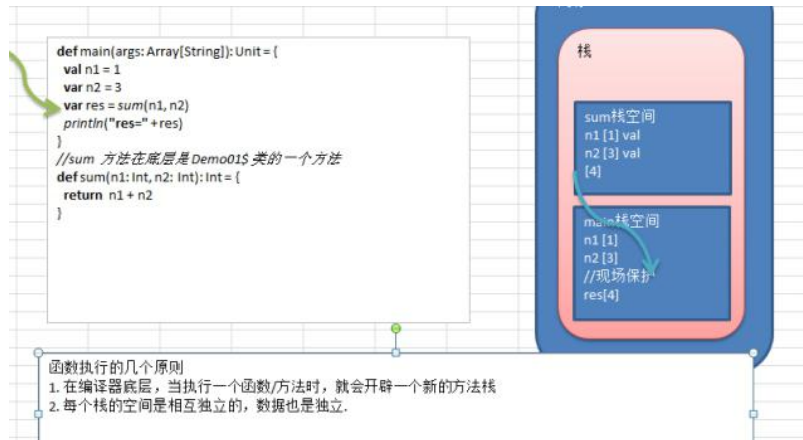
拉登同志给特工小组下达命令:去炸美国白宫,特工小组返回结果
程序员调用方法:给方法必要的输入,方法返回结果.

● 函数-调用机制

函数-调用过程

为了让大家更好的理解函数调用机制, 看1个案例, 并**画出示意图**, 这个很重要, 比如getSum 计算两个数的和, 并返回结果。

```
object Test01 {  
  def main(args: Array[String]): Unit = {  
    val n1 = 1  
    val n2 = 3  
    val res = sum(n1, n2)  
    println("res=" + res)  
  }  
  def sum(n1: Int, n2: Int): Int = {  
    return n1 + n2  
  }  
}
```





● 函数-递归调用

基本介绍

一个函数在函数体内又调用了本身，我们称为递归调用

递归调用快速入门

```
def test (n: Int) {  
    if (n > 2) {  
        test (n - 1)  
    }  
    println("n=" + n) //  
}
```

```
def test2 (n: Int) {  
    if (n > 2) {  
        test2 (n - 1)  
    }else {  
        println("n=" + n)  
    }  
}
```

当调用**test(4)** 和 **test2(4)** 上面两段代码分别输出什么？
递归调用并分析原因(画出示意图)



● 函数-递归调用

函数递归需要遵守的重要原则（总结）：

- 1) 程序执行一个函数时，就创建一个新的受保护的独立空间(新函数栈)
- 2) 函数的局部变量是独立的，不会相互影响
- 3) 递归必须向退出递归的条件逼近，否则就是无限递归，死龟了:)
- 4) 当一个函数**执行完毕**，**或者遇到return**，就会返回，遵守谁调用，就将结果返回给谁。

- 函数-递归调用

递归课堂练习题

➤ 题1：斐波那契数 [学员练习10min]

请使用递归的方式，求出斐波那契数1,1,2,3,5,8,13...
给你一个整数n，求出它的斐波那契数是多少？

➤ 题2：求函数值 [演示]

已知 $f(1)=3$; $f(n) = 2*f(n-1)+1$;
请使用递归的思想编程，求出 $f(n)$ 的值？

- 函数-递归调用

递归课堂练习题

➤ 题3：猴子吃桃子问题

有一堆桃子，猴子第一天吃了其中的一半，并再多吃了一个！以后每天猴子都吃其中的一半，然后再多吃一个。当到第十天时，想再吃时（还没吃），发现只有1个桃子了。**问题：最初共多少个桃子？**


● 函数注意事项和细节讨论

```
class Cat {  
    var name : String = "terry"  
}
```

- 1) 函数的形参列表可以是多个, 如果函数没有形参, 调用时 **可以不带()**
- 2) 形参列表和返回值列表的数据类型可以是值类型和引用类型。【案例演示】
- 3) **Scala**中的函数可以根据函数体最后一行代码自行推断函数返回值类型。那么在这种情况下, **return**关键字可以省略

```
def getSum(n1: Int, n2: Int): Int = {  
    n1 + n2  
}
```

```
def getSum(n1: Int, n2: Int) = {  
    n1 + n2  
}
```



- 4) 因为**Scala**可以自行推断, 所以在省略**return**关键字的场合, 返回值类型也可以省略
- 5) 如果函数明确使用**return**关键字, 那么函数返回就不能使用自行推断了, 这时要明确写成: **返回类型 =**, 当然如果你什么都不写, 即使有**return** 返回值为()

```
def getSum(n1: Int, n2: Int): Int = {  
    //因为这里有明确的return, 这时 getSum 就不能省略 : Int = 的 Int了
```

● 函数注意事项和细节讨论

- 6) 如果函数明确声明无返回值（声明Unit），那么函数体中即使使用return关键字也不会有返回值
- 7) 如果明确函数无返回值或不确定返回值类型，那么返回值类型可以省略(或声明为Any)

```
def f3(s: String) = {  
    if(s.length >= 3)  
        s + "123"  
    else  
        3  
}
```

```
def f4(s: String): Any = {  
    if(s.length >= 3)  
        s + "123"  
    else  
        3  
}
```

- 8) Scala语法中任何的语法结构都可以嵌套其他语法结构(灵活)，即：**函数中可以再声明/定义函数**，类中可以再声明类，方法中可以再声明/定义方法
- 9) Scala函数的形参，在声明参数时，直接赋初始值(默认值)，这时调用函数时，如果没有指定实参，则会使用默认值。如果指定了实参，则**实参会覆盖默认值**。

```
def sayOk(name : String = "jack"): String = {  
    return name + " ok!"  
}
```

● 函数注意事项和细节讨论

- 10) 如果函数存在多个参数，每一个参数都可以设定默认值，那么这个时候，传递的参数到底是覆盖默认值，还是赋值给没有默认值的参数，就不确定了(默认按照声明顺序[**从左到右**])。在这种情况下，可以采用**带名参数** [案例演示+练习]

```
def mysqlCon(add:String = "localhost",port : Int = 3306,
             user: String = "root", pwd : String = "root"): Unit = {
    println("add=" + add)
    println("port=" + port)
    println("user=" + user)
    println("pwd=" + pwd)
}
```

```
def f6 ( p1 : String = "v1", p2 : String ) {
    println(p1 + p2);
}
f6("v2" ) // (?)
f6(p2="v2") // (?)
```

- 11) scala 函数的**形参默认是val**的，因此不能在函数中进行修改.

- 函数注意事项和细节讨论

12) 递归函数未执行之前是无法推断出来结果类型，在使用时**必须有明确的返回值类型**

```
def f8(n: Int) = { //? 错误，递归不能使用类型推断，必须指定返回的数据类型
  if(n <= 0)
    1
  else
    n * f8(n - 1)
}
```




- 函数注意事项和细节讨论

13) Scala函数支持可变参数

```
//支持0到多个参数
def sum(args :Int*) : Int = {
}
//支持1到多个参数
def sum(n1: Int, args: Int*) : Int = {
}
```

说明:

- (1) **args 是集合, 通过 for循环 可以访问到各个值。**
- (2) 案例演示: 编写一个函数sum, 可以求出 1到多个int的和
- (3) 可变参数需要写在形参列表的最后。

- 函数练习题

判断下面的代码是否正确：

```
object Hello01 {  
  def main(args: Array[String]): Unit = {  
    def f1 = "venassa" //  
    println(f1)  
  }  
}
```

题1 //输出 venassa

def f1 = "venassa" 等价于

```
def f1() = {  
  "venassa"  
}
```



- 过程

基本介绍

将函数的返回类型为Unit的函数称之为过程(**procedure**)，如果明确函数没有返回值，那么等号可以省略

案例说明：

```
//f10 没有返回值，可以使用Unit来说明  
//这时，这个函数我们也叫过程 (procedure)  
def f10(name: String): Unit = {  
    println(name + " hello ")  
}
```

- 过程

注意事项和细节说明

- 1) 注意区分: 如果函数声明时没有返回值类型, 但是有 = 号, 可以进行**类型推断**最后一行代码。这时这个函数实际是有返回值的, 该函数并不是过程。(这点在讲解函数细节的时候讲过的.)
- 2) 开发工具的自动代码补全功能, 虽然会自动加上Unit, 但是考**虑到Scala语言的简单, 灵活**, 最好不加.

- 惰性函数

看一个应用场景

惰性计算（尽可能延迟表达式求值）是许多函数式编程语言的特性。惰性集合在需要时提供其元素，无需预先计算它们，这带来了一些好处。首先，您可以将耗时的计算推迟到绝对需要的时候。其次，您可以创造无限个集合，只要它们继续收到请求，就会继续提供元素。函数的惰性使用让您能够得到更高效的代码。Java 并没有为惰性提供原生支持，Scala提供了。

画图说明[大数据推荐系统]



推荐系统



- 惰性函数

Java实现懒加载的代码

```
public class LazyDemo {  
    private String property; //属性也可能是一个数据库连接，文件等资源  
    public String getProperty() {  
        if (property == null) { //如果没有初始化过，那么进行初始化  
            property = initProperty();  
        }  
        return property;  
    }  
    private String initProperty() {  
        return "property";  
    }  
}  
//比如常用的单例模式懒汉式实现时就使用了上面类似的思路实现
```

● 惰性函数

介绍

当函数返回值被声明为**lazy**时，函数的执行将被推迟，直到我们首次对此取值，该函数才会执行。这种函数我们称之为**惰性函数**，在Java的某些框架代码中称之为懒加载(延迟加载)。

案例演示

看老师演示

```
def main(args: Array[String]): Unit = {  
    lazy val res = sum(10, 20)  
    println("-----")  
    println("res=" + res) //在要使用res 前，才执行 }  
def sum(n1 : Int, n2 : Int): Int = {  
    println("sum() 执行了..")  
    return n1 + n2}  
}
```

注意事项和细节

- 1) **lazy** 不能修饰 **var** 类型的变量
- 2) 不但是 在调用函数时，加了 **lazy** ,会导致函数的执行被推迟，我们在声明一个变量时，如果给声明了 **lazy** ,那么变量值得分配也会推迟。 比如 **lazy val i = 10**



● 异常

介绍

- Scala提供try和catch块来处理异常。try块用于包含可能出错的代码。catch块用于处理try块中发生的异常。可以根据需要在程序中有任意数量的try...catch块。
- 语法处理上和Java类似，但是又不尽相同

Java异常处理回顾

```
// 可疑代码  
int i = 0;  
int b = 10;  
int c = b / i;
```

```
try {  
    // 可疑代码  
    int i = 0;  
    int b = 10;  
    int c = b / i; // 执行代码时，会抛出ArithmeticException异常  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    // 最终要执行的代码  
    System.out.println("java finally");  
}
```

- 异常

Java异常处理的注意点.

- 1) java语言按照try—catch-catch...—finally的方式来处理异常
- 2) 不管有没有异常捕获，都会执行finally, 因此通常可以在finally代码块中释放资源
- 3) 可以有多个catch，分别捕获对应的异常，这时需要把范围小的异常类写在前面，把范围大的异常类写在后面，否则编译错误。会提示 "*Exception 'java.lang.xxxxxx' has already been caught*" 【案例演示】



- 异常

Scala异常处理举例

```
try {  
    val r = 10 / 0  
} catch {  
    case ex: ArithmeticException => println("捕获了除数为零的算数异常")  
    case ex: Exception => println("捕获了异常")  
} finally {  
    // 最终要执行的代码  
    println("scala finally...")  
}
```

- 异常

Scala异常处理小结

- 1) 我们将可疑代码封装在try块中。在try块之后使用了一个catch处理程序来捕获异常。如果发生任何异常，catch处理程序将处理它，程序将不会异常终止。
- 2) Scala的异常的工作机制和Java一样，但是Scala没有“checked(编译期)”异常，即Scala没有编译异常这个概念，异常*都是在运行的时候捕获处理*。
- 3) 用throw关键字，抛出一个异常对象。所有异常都是Throwable的子类型。throw表达式是有类型的，就是Nothing，因为Nothing是所有类型的子类型，所以throw表达式可以用在需要类型的地方

```
def main(args: Array[String]): Unit = {  
    val res = test()  
    println(res.toString)  
}  
def test(): Nothing = {  
    throw new Exception("不对")  
}
```

- 异常

Scala异常处理小结

- 4) 在Scala里，借用了模式匹配的思想来做异常的匹配，因此，在`catch`的代码里，是一系列`case`子句来匹配异常。【前面案例可以看出这个特点, 模式匹配我们后面详解】，当匹配上后 => 有多条语句可以换行写，类似java 的 `switch case x:` 代码块..
- 5) 异常捕捉的机制与其他语言中一样，如果有异常发生，`catch`子句是按次序捕捉的。因此，在`catch`子句中，越具体的异常越要靠前，越普遍的异常越靠后，如果把越普遍的异常写在前，把具体的异常写在后，在scala中也不会报错，但这样是非常不好的编程风格。

- 异常

Scala异常处理小结

- 7) `finally`子句用于执行不管是正常处理还是有异常发生时都需要执行的步骤，一般用于对象的清理工作，这点和Java一样。
- 8) Scala提供了`throws`关键字来声明异常。可以使用方法定义声明异常。它向调用者函数提供了此方法可能引发此异常的信息。它有助于调用函数处理并将该代码包含在try-catch块中，以避免程序异常终止。在scala中，可以使用`throws`注释来声明异常

```
def main(args: Array[String]): Unit = {  
    f11()  
}  
@throws(classOf[NumberFormatException])//等同于NumberFormatException.class  
def f11() = {  
    "abc".toInt  
}
```


● 函数的课堂练习题

- 1) 函数可以没有返回值案例，编写一个函数,从终端输入一个整数打印出对应的金子塔。 【课后练习】

```
  *
 ***
*****
*****
*****
*****
*****
```

- 2) 编写一个函数,从终端输入一个整数(1—9),打印出对应的乘法表: 【上机练习】

```
1×1=1
1×2=2  2×2=4
1×3=3  2×3=6  3×3=9
1×4=4  2×4=8  3×4=12  4×4=16
1×5=5  2×5=10  3×5=15  4×5=20  5×5=25
1×6=6  2×6=12  3×6=18  4×6=24  5×6=30  6×6=36
1×7=7  2×7=14  3×7=21  4×7=28  5×7=35  6×7=42  7×7=49
1×8=8  2×8=16  3×8=24  4×8=32  5×8=40  6×8=48  7×8=56  8×8=64
1×9=9  2×9=18  3×9=27  4×9=36  5×9=45  6×9=54  7×9=63  8×9=72  9×9=81
```


- 函数的课堂练习题

3) 编写函数,对给定的一个二维数组(3×3)转置, 这个题讲数组的时候再完成。

1 2 3		1 4 7
4 5 6		2 5 8
7 8 9		3 6 9



谢谢！ 欢迎收看！