## 递归函数

Val functionName: 函数类型 = 函数字面量

Val functionName = 函数字面量

**定义一个函数，接收两个参数，一个是x,一个是y 返回x的n次方**

|  |
| --- |
| *// 定义一个函数，接收两个参数，一个是x,一个是y 返回x的n次方* **def** nMultiply(x: Int, n: Int) = {  **var** y = x  **for** (i <- 1 until n) {  y = y \* x  }  y  }  *// 递归写法* **def** nMul(x: Int, n: Int): Int = {  x \* (**if** (n == 1) 1 **else** *nMul*(x, n - 1)) } |

**计算n的阶乘**

|  |
| --- |
| *// 计算n的阶乘* **def** nFactorial(n: Int): Int = {  n \* ((**if** (n == 1) 1 **else** *nFactorial*(n - 1))) } |

**求阶乘的和**

|  |
| --- |
| *// 求阶乘的和* **def** nFactorialSum(n: Int): Int = {  **if**(n == 1) 1 **else** n + n \* ((**if** (n == 1) 1 **else** *nFactorial*(n - 1))) } |

**遍历list集合，求和**

|  |
| --- |
| *// 遍历list,从左向右计算list的和* **def** lList(list:List[Int]): Int = {  **if**(list == *Nil*) 0 **else** list.head + *lList*(list.tail) }  *// 遍历list,从右向左计算list的和* **def** rList(list: List[Int]): Int = {  **if**(list == *Nil*) 0 **else** list.last + *rList*(list.init) } |

## 函数书写省略规则

1. 如果对对象调用函数，函数只接收一个参数，点号和小括号可省略，如果接收多个参数，点号可省略，小括号不可省略

|  |
| --- |
| **val** list = *List*(2, 3, 67, 21, 2) *// 函数正常方法调用 println*(list.mkString(**","**)) *// 省略点和小括号 println*(list mkString **","**) *// 省略点 println*(list slice(2, 4)) |

1. 在传递函数作为参数的时候，因为函数位置处可以判断函数的类型，因此被传递的函数字面量可以不用指定参数的类型

|  |
| --- |
| **val** list = *List*(2, 3, 67, 21, 2)  *// 正常传递函数字面量 println*(list.reduce((x1: Int, x2: Int) => x1 + x2)) *// 省略参数类型指定，因为自动推断 println*(list.reduce((x1, x2) => x1 + x2)) |

1. 当字面量函数只要一个参数的时候，小括号也可以省略

|  |
| --- |
| **val** list = *List*(2, 3, 67, 21, 2)  *// 省略小括号 println*(list.filter(x => x > 20)) |

1. 当参数在函数体内只使用一次的时候，参数的定义可以省略，在函数体使用\_代替参数

|  |
| --- |
| **val** list = *List*(2, 3, 67, 21, 2)  *// 省略参数的定义 println*(list.filter(\_ > 20)) *println*(list.reduce(\_ + \_)) |

1. 如果传递一个函数，不写字面量而是用val定义的函数名称的话，则不需要考虑参数的传递

|  |
| --- |
| **val** list = *List*(2, 3, 67, 21, 2)  *// 可以传递val定义的函数，不需要传参数* **val** sum = (x: Int, y: Int) => x + y *println*(list.reduce(sum)) |

1. 调用函数的时候，如果函数的参数相对复杂，可以使用大括号代替小括号

|  |
| --- |
| **val** list = *List*(2, 3, 67, 21, 2)  *// 小括号使用大括号代替 println*(list.map(x => x + 100)) **val** result = list.map{  **val** t = 100  x => x + t } *println*(result) |

## 函数参数的设置

Scala中函数的参数可以在定义时赋一个默认值，如果一个参数有默认值，那么在这个函数调用时有默认值的参数不用传参数，不传参数代表使用默认值

如果传参数的话就使用传递的参数

Scala中在调用函数的时候可以不按照函数定义的参数顺序，我们自己可以通过代名传参来设置定参数的顺序

|  |
| --- |
| **package** top.xiesen.function  */\*\*  \* 函数参数的设置  \*/* **object** FunctionParam {   **def** multySum(x: Int, y: Int = 1, z: Int = 1) = {  *println*(**s"x = $**x**, y = $**y**, z = $**z**"**)  x \* y \* z  }   **def** main(args: Array[String]): Unit = {  *// 不传 默认值参数调用函数  println*(multySum(2))  *// 传递默认值参数,则默认参数失效  println*(multySum(10,2))  *// 当我们传递的默认值顺序和默认值顺序不一致时，可使用代名传参来指定，参数是为谁传递的  println*(multySum(y = 2,z = 5, x = 10))  } } |

## 高阶函数

1. 把函数当做参数的函数
2. 把函数当做返回值的函数(闭包)

函数的柯里化

把一个函数的多个参数用多个小括号隔开，这种函数的定义形式就叫函数的柯里化

def sum (x: Int, y: Int) = x + y

def sum (x: Int)(y: Int) = x + y

1. 可以通过传部分参数让其返回一个函数
2. 和隐式转换结合使用，因为隐式参数的定义范围是小括号

|  |
| --- |
| **package** top.xiesen.function  */\*\*  \* 高阶函数  \*/* **object** HighFunction {   *// 定义一个函数，它能够把金额转换成金额，并加上币种的符号  // 把函数当做参数的高阶参数* **def** converToAmount(f: (Double) => String, x: Double) = f(x)   **val** *rmbConver* = (amount: Double) => **s"$**amount **￥"  val** *dollarConver* = (amount: Double) => **s"$**amount **$**{**"$"**}**"** *// 把函数当做返回值  // 定义一个乘法器，他能帮我们生成x\*N这样一个函数* **def** multiplyN(n: Int) = {  (x: Int) => n \* x  }   *// 定义一个函数，这个函数每次调用返回值都是原值(自由变量)+1，初始值0* **var** *addOne* = 0   **def** addOneFunction() = {  *addOne* += 1  *addOne* }   **def** addOneFunctionEnClosure() = {  **var** closeOne = 0  **val** addOneEnclose = () => {  closeOne += 1  closeOne  }  addOneEnclose  }   **def** main(args: Array[String]): Unit = {  println(*converToAmount*(*rmbConver*,100))  println(*converToAmount*(*dollarConver*,100))   **val** multiply5 = *multiplyN*(5)  println(multiply5(3))  println(multiply5(6))  println(multiply5(9))   **val** multiply6 = *multiplyN*(6)  println(multiply6(3))  println(multiply6(6))  println(multiply6(9))   **var** free = 10  **def** addFree = (x: Int) => x + free  println(addFree(5))  println(addFree(6))  free = 20  println(addFree(5))  println(addFree(6))   println(**"----------------------"**)  println(*addOneFunction*())  println(*addOneFunction*())  *addOne* = 0  println(*addOneFunction*())  println(*addOneFunction*())  println(**"----------------------"**)   **val** addOneEnclose = *addOneFunctionEnClosure*()  println(addOneEnclose())  println(addOneEnclose())  println(addOneEnclose())  } } |

## 隐私转换

Scala中定义了隐式转换的机制，让我们可以在写代码的时候很多转换或者参数的部分由编译器自动给我们添加，大大减少了代码量

1. 可以在方法或者变量前面添加隐式转换标记implicit
2. 可以将方法的参数前面添加隐式转换标记implicit
3. 可以在类前面添加隐式转换标记implicit

|  |
| --- |
| **package** top.xiesen.implicittest  */\*\*  \* Created by Allen on 2017/11/17.  \* 隐式转换implicit  \*/* **object** ImplicitTest {  *// 把金额转换成带币种符号的字符串,接收两个参数，1.币种符号，2.金额* **def** convertToAmount(amountType: String, amount: Double) = {  **s"$**amount**[$**amountType**]"** }   *// 隐式参数* **def** convertToAmount1(amount: Double)(**implicit** amountType: String) = {  **s"$**amount**[$**amountType**]"** }   *// 如果将amountType做成隐式的参数，就不用再传递amountType参数值* **def** main(args: Array[String]): Unit = {  *println*(*convertToAmount*(**"￥"**, 180.4))   *// 隐式值* **implicit val** amountType: String = **"$"** *println*(*convertToAmount1*(180.4))   *// 不能够重复定义  /\*implicit val amountType1: String = "￥"  println(convertToAmount1(180.4))\*/* }  } |

## 隐式值和隐式参数函数

Implicit在修饰函数参数的时候它必须加载到小括号的最前面，同时小括号内的所有参数都将被声明成隐式参数

Implicit修饰参数后的函数如果是被keli并且拥有非隐式的参数的话，带有implicit的小括号必须写在不带implicit的后面

带有隐式参数的函数被调用时，必须先定义隐式值

隐私值在一个应用中只能被定义一个，重复定义会报错

同一个类型的隐式值只能定义一个否则会编译报错：隐式值冲突

隐式值一般定义在包对象中

隐式函数一般用于类型的自动转换(扩充某些类型的功能)

|  |
| --- |
| **package** top.xiesen.implicittest  */\*\*  \* Created by Allen on 2017/11/17.  \* 隐式转换函数  \*/* **object** implicitConvertFunction {   *// 定义一个隐式转换把Double类型的数据转换成Int类型* **implicit def** doubleToInt(x: Double): Int = x.toInt  **implicit def** convertStringToMyString(x: String) = **new** MyString(x)   **def** main(args: Array[String]): Unit = {  **val** intType:Int = 33.5  *println*(intType)  **val** str = **"abcd"** str.printForEachChar()  } }  **class** MyString(**val** str: String) {  **def** printForEachChar() = {  str.foreach(x => *print*(**s"$**x**, "**))  } } |

隐式类型不能直接定义在类的最外层，它需要放在object中来定义

隐私转换

Def定义的函数，函数名称后面加不加小括号都是对函数的调用

Val 定义的函数 函数名称后面不加小括号是对函数的引用，加小括号是对函数的调用

## Scala模式匹配

Scala中的模式匹配 match case

Java switch case

只能对基础数据类型进行匹配

每一个case都需要加break跳出匹配

只能对值进行匹配

只是一种流程控制语法，无返回值

Scala match case

可以匹配任意类型

不需要break跳出，所有case表达式中，只有一个被匹配，立马跳出

除了对值进行匹配外，还有7,8中匹配方式

它有返回值

### 普通matchcase

|  |
| --- |
| *// 普通matchcase* **def** simpleMatchCase(x: Any) = {  x **match** {  *// => 描述函数类型，函数字面量，重命名，隐藏,math case匹配  // ; 作为单行之间的分隔符* **case** 1 => {  *println*(**"值是1"**)  }  **case** 2 => *println*(**"值是2"**)  **case true** => *println*(**"只是true"**)  **case** \_ => *println*(**"其他值"**)  } } |

### 带返回值的matchcase

|  |
| --- |
| *// 带返回值的matchcase* **def** retValueSimpleMatchCase(x: Any) = {  **val** result = x **match** {  **case** 1 => **"值是1"  case** 2 => **"值是2"  case true** => **"只是ture"  case** \_ => **"其他值"** }  result } |

### 常量模式匹配

|  |
| --- |
| *// 常量模式匹配* **def** constantMatch(x: Any) = {  x **match** {  **case "abc"** => **"常量值abc"  case true** => **"常量值true"  case** 123 => **"常量值123"  case** 22.00 => **"常量值22.00"  case** \_ => **"其他值"** } } |

### 常量模式匹配

|  |
| --- |
| *// 常量模式匹配* **def** constantMatch1(x: Any) = {  *// 如果x 匹配上apeciaValue，返回特殊数值* **val** apeciaValue = **"123abc"** x **match** {  **case "abc"** => **"常量值abc"  case true** => **"常量值true"  case** 123 => **"常量值123"  case** 22.00 => **"常量值22.00"** *// 如果匹配变量中的字符，需要将变量添加上反引号* **case** `apeciaValue` => **"特殊字符"  case** \_ => **"其他值"** } } |

### 变量模式匹配

|  |
| --- |
| *// 变量模式匹配* **def** variableMatch(x: Any) = {  x **match** {  **case** z => *println*(**s"$**z**"**)  **case** \_ => *println*(**"其他"**)  } } |

### 变量匹配可以在变量后面添加过滤条件进行匹配

|  |
| --- |
| *// 变量匹配可以在变量后面添加过滤条件进行匹配* **def** scoreAnayLize(score: Int) = {  score **match** {  **case** x **if** x > 90 => **s"优秀,分数: $**x**"  case** x **if** x > 80 => **s"良好,分数: $**x**"  case** x **if** x > 70 => **s"中等,分数: $**x**"  case** \_ => **s"差,分数: $**{score}**"** } } |

### 类型匹配

|  |
| --- |
| *// 类型匹配* **def** typeMatchCase(x: Any) = {  x **match** {  **case** z: Boolean => **s"$**z **是一个boolean类型"  case** z: Int => **s"$**z **是Int类型"  case** z: String => **s"$**z **是String类型"  case** \_ => **s"$**x **是其他数据类型"** } } |

### option some none

|  |
| --- |
| *// option some none* **def** osnMatchCase(x: Any) = {  x **match** {  **case** *Some*(1) => **"结果是1"  case** *Some*(2) => **"结果是2"  case** *Some*(y) => **s"结果是$**y**"  case** None => **"没有结果"** } } |

### 序列匹配

|  |
| --- |
| *// 序列匹配，集合匹配* **def** listMatchCase(x: List[Int]) = {  x **match** {  **case** List(9, \_\*) => **"list中第一个元素是9"  case** List(a, b) => **s"list中有两个元素,第一个值是$**a**,第二个值是$**b**"  case** List(a, b, \_) => **s"list中有三个元素,第一个值是$**a**,第二个值是$**b**"  case** List(\_, \_, \_, \_) => **s"list中有四个元素"  case** List(a, b, \_\*) => **s"list中至少有三个元素,第一个值是$**a**,第二个值是$**b**"  case** *Nil* => **"空list"  case** \_ => **"其他list"** } } |

### 集合类型匹配

|  |
| --- |
| *// 集合类型匹配* **class** Parent {}  **class** Child **extends** Parent {}  *// array可以匹配集合类型，List类型不能匹配泛型* **def** listTypeMatchCase(x: Any) = {  x **match** {  **case** v: Array[Child] => **"x是一个Child的集合类型"  case** v: Array[Parent] => **"x是一个Parent的集合类型"  case** v: Array[Int] => **"x是一个Int的集合类型"  case** \_ => **"其他类型的集合类"** } } |

### 元组匹配

|  |
| --- |
| *// 元组匹配 // match case 一次只能匹配一个值，如果想要匹配多个值，那么需要使用元组将多个值封装起来* **def** tupleMatchCase(x: Any, y: Any) = {  (x, y) **match** {  **case** (x: Int, y: String) => **"x是Int类型, y是String类型"  case** (3, 4) => **"x=3, y=4"  case** (5, \_) => **"x=5,y随意"  case** \_ => **"其他类型"** } } |

### 正则表达式匹配

|  |
| --- |
| *// 正则表达式匹配* **def** regularRegexMatch(x: String) = {  **val** regExp = **"(\\d+)\\.(\\d+)\\.(\\d+)\\.(\\d+)"**.r  x **match** {  **case** regExp(one, two, three, four) => **s"第一段: $**one**,第二段: $**two**,第三段: $**three**,第四段: $**four**"  case** \_ => **"不是ip地址"** } } |

### 构造方法匹配 case class来使用

|  |
| --- |
| *//构造方法匹配case class 来使用,相当于javabean 更牛逼* **def** caseClassMatch(x:Any)={  x **match** {  **case** *Student*(id,name,18)=>*println*(**s"年龄18,id为$**id**,姓名:$**name**"**)  **case** *Student*(id ,name,\_)=>*println*(**s"x是student类型,id为$**id**,姓名:$**name**"**)  **case** \_=>**"其他类型"** } } |

### Case class嵌套

|  |
| --- |
| *//case class 中有一个属性是其它的 case class 那么可以用两个嵌套匹配* **def** caseClassMutiplyMatch(x:Any)={  x **match** {  **case** *SchoolClass*(id ,name,*Student*(sid,sname,18))=>**s"class里有一个学生18岁,id为$**sid**,姓名:$**sname**"  case** *SchoolClass*(id ,**"san"**,\_)=>**s"san年纪的学生"  case** \_=>**"其他"** } } |

### case class 类型匹配

|  |
| --- |
| *//case class 类型匹配  //sealed 表示父类是封闭类* **sealed abstract class** MStudent  **case class** LittleStudent(id :String,name :String)**extends** MStudent  **case class** HighStudent(id :String,name :String)**extends** MStudent  **case class** CollageStudent(id :String,name :String)**extends** MStudent  **case class** OtherStudent(id :String,name :String)**extends** MStudent  **def** caseClassMatchMstudent(x:MStudent)={  x **match** {  **case** *LittleStudent*(id,name)=>**"小学生"  case** *HighStudent*(id,name)=>**"中学生"  case** *CollageStudent*(id,name)=>**"大学生"** *// case OtherStudent(id,name)=>"其他学生" // 如果没有在match case 中罗列所有的类型,那么编译会给出警告可能会出问题* }  }  **def** caseClassTypeMatch(x:Any)={  x **match** {  *// case o:Student=>"student类型"* **case** *Student*(id ,name, age)=>**"student类型"** } } |

### Case class特性

|  |
| --- |
| *//没有var val 自动默认是val,要想var 那么手动添加* **case class** Student(id :String,name :String,age :Int) **case class** SchoolClass(id :String,name :String,student :Student) **object** MatchC {  **def** main(args: Array[String]): Unit = {  *//定义case class 会自动生成伴生对象,并且自动apply方法  //case class 自动帮我们重写了tostring hashcode equals 方法* **val** student = *Student*(**"1"**,**"校长"**,55)  *println*(student)  *println*(student.age)  *//判断三个属性一样不一样* **val** student1 = *Student*(**"2"**,**"xx"**,10)  **val** student2 = *Student*(**"2"**,**"xx"**,10)  *println*(student1==student2)  *//内容一样的两个对象 hashcode值一样  println*(student1.hashCode())  } } |