Scala

author：yuhang.sun

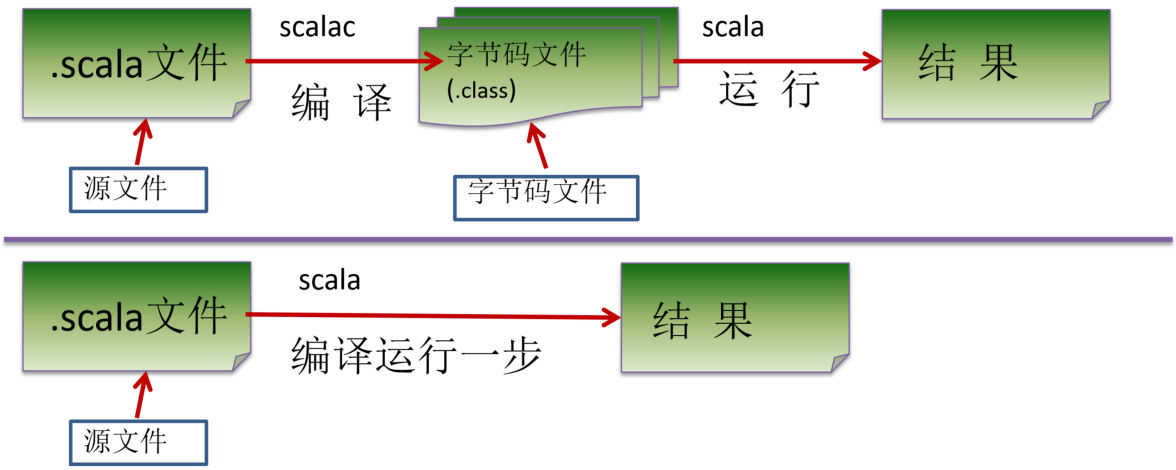
一、Scala语言的概述

1.1 Scala语言的特点

Scala是一门以java虚拟机（JVM）为运行环境并将面向对象和函数式编程的最佳特性结合在一起的静态类型编程语言。

1.Scala 是一门多范式 (multi-paradigm) 的编程语言，Scala支持面向对象和函数式编程
2.Scala源代码(.scala)会被编译成Java字节码(.class)，然后运行于JVM之上，并可以调用现有的Java类库，实现两种语言的无缝对接。
3.scala 单作为一门语言来看， 非常的简洁高效 （三元运算， ++ ， --）
4.Scala 在设计时，马丁·奥德斯基 是参考了Java的设计思想，可以说Scala是源于java，同时马丁·奥德斯基 也加入了自己的思想，将函数式编程语言的特点融合到JAVA中

1.2 Scala执行流程分析



1.3 Scala语言输出的三种形式

```
val name = "ApacheCN"
val age  = 1
val url  = "www.atguigu.com"
println("name=" + name + " age=" + age + " url=" + url) ---字符串拼接
printf("name=%s, age=%d, url=%s \n", name, age, url) --- C语言形式
println(s"name=$name, age=$age, url=$url") --- PHP形式
```

1.4 键盘输入语句

```
import scala.io.StdIn

StdIn.readline()..
StdIn.readInt()..
```

1.5 注释

1. 单行注释

```
格式：  //注释文字
```

2. 多行注释

```
格式：  /*  注释文字 */
```

3. 文档注释

注释内容可以被工具 scaladoc 所解析，生成一套以网页文件形式体现的该程序的说明文档

```
object Hello {
  /**
    * @deprecated xxx
    * @example testing coding
    * @param args
    */
  def main(args: Array[String]): Unit = {
     println("helllo")
  }
}
scaladoc -d d:/ Hello.scala
```

二、变量

2.1 变量使用的基本步骤

1.声明/定义变量 (scala要求变量声明时初始化)
2.使用

2.2 基本语法

```
var | val 变量名 [: 变量类型] = 变量值
```

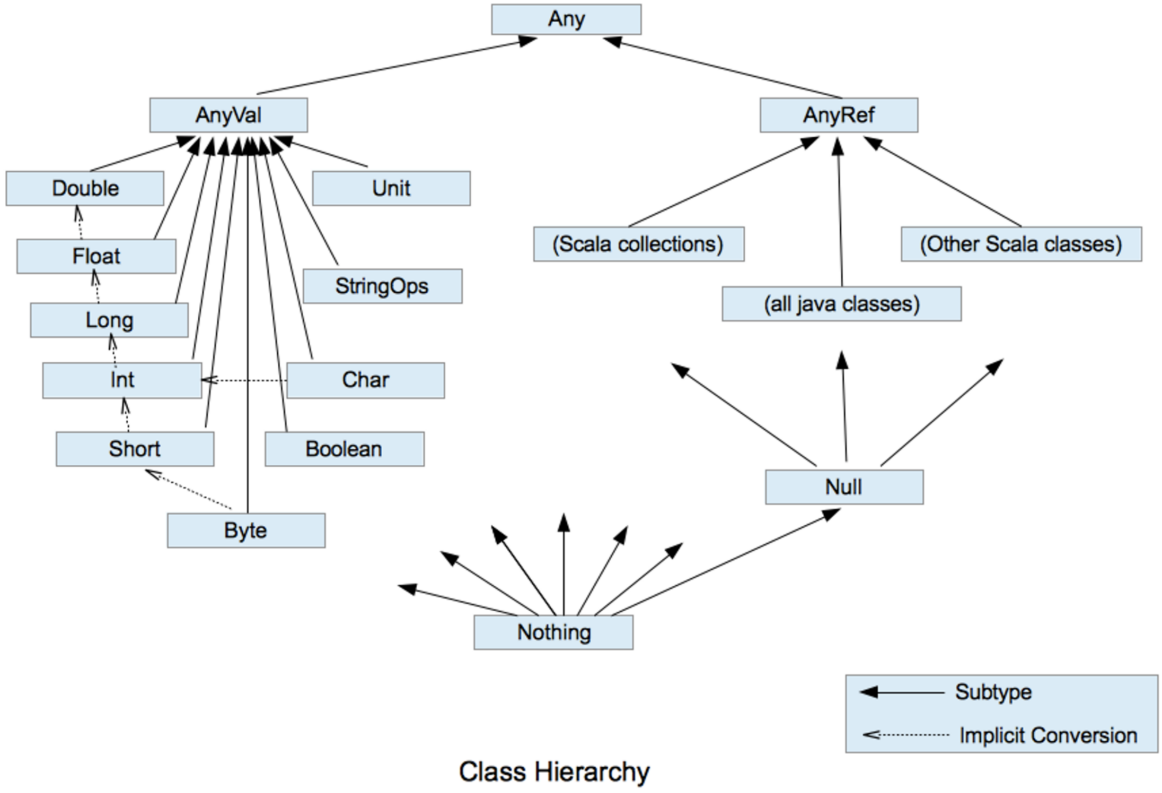
2.3 注意事项

1.声明变量时，类型可以省略（编译器自动推导,即类型推导）
2.类型确定后，就不能修改，说明Scala是强数据类型语言.
3.在声明/定义一个变量时，可以使用var或者val来修饰，var 修饰的变量可改变，val修饰的变量不可改 [案例].
4.val修饰的变量在编译后，等同于加上final，通过反编译看下底层代码。
5.var修饰的对象引用可以改变，val修饰的则不可改变，但对象的状态(值)却是可以改变的。(比如: 自定义对象、数组、集合等等)
6.变量声明时，需要初始值

2.4 数据类型

1. 介绍

1.Scala 与 Java有着相同的数据类型，在Scala中数据类型都是对象，也就是说scala没有java中的原生类型
2.Scala数据类型分为两大类 AnyVal(值类型) 和 AnyRef(引用类型)， 注意：不管是AnyVal还是AnyRef都是对象
3.相对于java的类型系统，scala要复杂些！也正是这复杂多变的类型系统才让面向对象编程和函数式编程完美的融合在了一起



Class Hierarchy

2. 数据类型一览表

数据类型	描述
Byte	8位有符号补码整数。数值区间为 -128 到 127
Short	16位有符号补码整数。数值区间为 -32768 到 32767
Int	32位有符号补码整数。数值区间为 -2147483648 到 2147483647
Long	64位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807
Float	32 位, IEEE 754标准的单精度浮点数
Double	64 位 IEEE 754标准的双精度浮点数
Char	16位无符号Unicode字符, 区间值为 U+0000 到 U+FFFF
String	字符序列
Boolean	true或false
Unit	表示无值，和其他语言中void等同。用作不返回任何结果的方法的结果类型。Unit只有一个实例值，写成()。
Null	null
Nothing	Nothing类型在Scala的类层级的最低端；它是任何其他类型的子类型。
Any	Any是所有其他类的超类
AnyRef	AnyRef类是Scala里所有引用类(reference class)的基类

3. 整型

1. 基本介绍

scala的整数类型就是用于存放整数值的，比如 12 ， 30， 3456等等

2. 整形的类型

数据类型	描述
Byte [1]	8位有符号补码整数。数值区间为 -128 到 127
Short [2]	16位有符号补码整数。数值区间为 -32768 到 32767
Int [4]	32位有符号补码整数。数值区间为 -2147483648 到 2147483647
Long [8]	64位有符号补码整数。数值区间为 -9223372036854775808 到 9223372036854775807 = 2的(64-1)次方-1

3. 使用细节

1.Scala各整数类型有固定的表数范围和字段长度，不受具体OS的影响，以保证Scala程序的可移植性。
2.Scala的整型 常量/字面量 默认为 Int 型，声明Long型 常量/字面量须后加‘l’’或‘L’ [反编译看]
3.Scala程序中变量常声明为Int型，除非不足以表示大数，才使用Long

4. 浮点型

1. 基本介绍

Scala的浮点类型可以表示一个小数，比如 123.4f，7.8 ，0.12等等

2. 浮点型的分类

数据类型	描述
Float [4]	32 位, IEEE 754标准的单精度浮点数
Double [8]	64 位 IEEE 754标准的双精度浮点数

3. 使用细节

1.与整数类型类似，Scala 浮点类型也有固定的表数范围和字段长度，不受具体OS的影响。
2.Scala的浮点型常量默认为Double型，声明Float型常量，须后加‘f’或‘F’。
3.浮点型常量有两种表示形式
　　十进制数形式：如：5.12　　512.0f　　.512　(必须有小数点）
　　科学计数法形式:如：5.12e2　= 5.12乘以10的2次方
5.12E-2　= 5.12除以10的2次方
4.通常情况下，应该使用Double型，因为它比Float型更精确(小数点后大致7位)
//测试数据 ：2.2345678912f　, 2.2345678912

5. 字符类型

1. 基本介绍

字符类型可以表示单个字符,字符类型是Char， 16位无符号Unicode字符(2个字节)，区间值为 U+0000 到 U+FFFF

2. 使用细节

1.字符常量是用单引号(‘ ’)括起来的单个字符。例如：var c1 = 'a‘ var c2 = '中‘ var c3 = '9'
2.Scala 也允许使用转义字符‘\’来将其后的字符转变为特殊字符型常量。例如：var c3 = ‘\n’ // '\n'表示换行符
3.可以直接给Char赋一个整数，然后输出时，会按照对应的unicode 字符输出
['\u0061' 97]
4.Char类型是可以进行运算的，相当于一个整数，因为它都对应有Unicode码.

3. 字符类型本质探讨

1.字符型存储到计算机中，需要将字符对应的码值（整数）找出来
　　存储：字符-->码值-->二进制-->存储
　　读取：二进制-->码值--> 字符-->读取
2.字符和码值的对应关系是通过字符编码表决定的(是规定好)， 这一点和Java一样。

6. 布尔类型

1. 基本介绍

```
1.布尔类型也叫Boolean类型，Booolean类型数据只允许取值true和false
2.boolean类型占1个字节。
3.boolean 类型适于逻辑运算，一般用于程序流程控制
```

7. Unit类型、Null类型和Nothing类型

1. 基本说明

数据类型	描述
Unit	表示无值，和其他语言中void等同。用作不返回任何结果的方法的结果类型。Unit只有一个实例值，写成()。32 位, IEEE 754标准的单精度浮点数
Null	null , Null 类型只有一个实例值 null64 位 IEEE 754标准的双精度浮点数
Nothing	Nothing类型在Scala的类层级的最低端；它是任何其他类型的子类型。当一个函数，我们确定没有正常的返回值，可以用Nothing 来指定返回类型，这样有一个好处，就是我们可以把返回的值（异常）赋给其它的函数或者变量（兼容性）

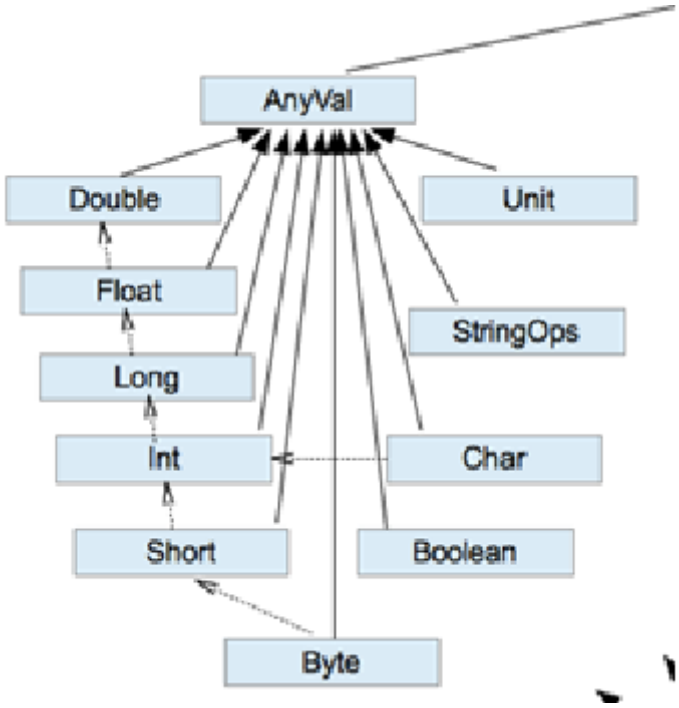
2. 注意事项和使用细节

```
1.Null类只有一个实例对象，null，类似于Java中的null引用。null可以赋值给任意引用类型(AnyRef)，但是不能赋值给值类型(AnyVal: 比如 Int, Float, Char, Boolean, Long, Double, Byte, Short)
2.Unit类型用来标识过程，也就是没有明确返回值的函数。由此可见，Unit类似于Java里的void。Unit只有一个实例，()，这个实例也没有实质的意义
3.Nothing，可以作为没有正常返回值的方法的返回类型，非常直观的告诉你这个方法不会正常返回，而且由于Nothing是其他任意类型的子类，他还能跟要求返回值的方法兼容。
```

2.5 值类型转换

1. 介绍

当Scala程序在进行赋值或者运算时，精度小的类型自动转换为精度大的数据类型，这个就是自动类型转换(隐式转换)。



2. 细节说明

1.有多种类型的数据混合运算时，系统首先自动将所有数据转换成容量最大的那种数据类型，然后再进行计算。 5.6 + 10 = 》double
2.当我们把精度(容量)大 的数据类型赋值给精度(容量)小 的数据类型时，就会报错，反之就会进行自动类型转换。
3.(byte, short) 和 char之间不会相互自动转换。
4.byte，short，char他们三者可以计算，在计算时首先转换为int类型。
5.自动提升原则： 表达式结果的类型自动提升为 操作数中最大的类

3. 强制类型转换

1. 介绍

自动类型转换的逆过程，将容量大的数据类型转换为容量小的数据类型。使用时要加上强制转函数，但可能造成精度降低或溢出,格外要注意。

2. 细节说明

```
1.当进行数据的 从 大-->小，就需要使用到强制转换
2.强转符号只针对于最近的操作数有效，往往会使用小括号提升优先级
3.Char类型可以保存Int的常量值，但不能保存Int的变量值，需要强转
4.Byte和Short类型在进行运算时，当做Int类型处理。
```

2.6 值类型和String类型的转换

1. 介绍

```
在程序开发中，我们经常需要将基本数据类型转成String 类型。或者将String类型转成基本数据类型。
```

2. 转换

```
基本类型转String类型：语法： 将基本类型的值+"" 即可
String类型转基本数据类型：语法：通过基本类型的String的toXxx方法即可
```

2.7 标识符

1. 概念

```
1.Scala 对各种变量、方法、函数等命名时使用的字符序列称为标识符
2.凡是自己可以起名字的地方都叫标识符
```

2. 规则

```
Scala中的标识符声明，基本和Java是一致的，但是细节上会有所变化。

1.首字符为字母，后续字符任意字母和数字，美元符号，可后接下划线_
2.数字不可以开头。
3.首字符为操作符(比如+ - * / )，后续字符也需跟操作符 ,至少一个(反编译)
4.操作符(比如+-*/)不能在标识符中间和最后.
5.用反引号`....`包括的任意字符串，即使是关键字(39个)也可以[true]
```

3. 39个关键字

```
package, import, class, object, trait, extends, with,
type, forSome
private, protected, abstract, sealed, final, implicit,
lazy, override
try, catch, finally, throw
if, else, match, case, do, while, for, return, yield
def, val, var
this, super
new
true, false, null
```

三、 运算符

3.1 运算符介绍

运算符是一种特殊的符号，用以表示数据的运算、赋值和比较等。
1.算术运算符
2.赋值运算符
3.比较运算符(关系运算符)
4.逻辑运算符
5.位运算符

运算符	运算	范例	结果
+	正号	+3	3
-	负号	b=4; -b	-4
+	加	5+5	10
-	减	6-4	2
*	乘	3*4	12
/	除	5/5	1
%	取模(取余)	7%5	2
+	字符串相加	“He”+”llo”	“Hello”

3.2 运算符

1. 逻辑运算符

1. 介绍

> 用于连接多个条件（一般来讲就是关系表达式），最终的结果也是一个Boolean值。

运算符	描述	实例
&&	逻辑与	(A && B) 运算结果为 false
||	逻辑或	(A || B) 运算结果为 true
!	逻辑非	!(A && B) 运算结果为 true

2. 赋值运算符

1. 介绍

> 赋值运算符就是将某个运算后的值，赋给指定的变量。

2. 分类

运算符	描述	实例
=	简单的赋值运算符，将一个表达式的值赋给一个左值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C *= A 等于 C = C * A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A
<<=	左移后赋值	C <<= 2 等于 C = C << 2
>>=	右移后赋值	C >>= 2 等于 C = C >> 2
&=	按位与后赋值	C &= 2 等于 C = C & 2
^=	按位异或后赋值	C ^= 2 等于 C = C ^ 2
|=	按位或后赋值	C |= 2 等于 C = C | 2

3. 特点

1.运算顺序从右往左
2.赋值运算符的左边 只能是变量,右边 可以是变量、表达式、常量值/字面量
3.复合赋值运算符等价于下面的效果　　比如：a+=3 等价于a=a+3

3. 位运算符

运算符	描述	实例
&	按位与运算符	(a & b) 输出结果 12 ， 二进制解释： 0000 1100
|	按位或运算符	(a | b) 输出结果 61 ， 二进制解释： 0011 1101
^	按位异或运算符	(a ^ b) 输出结果 49 ， 二进制解释： 0011 0001
~	按位取反运算符	(~a) 输出结果 -61 ， 二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。
<<	左移动运算符	a << 2 输出结果 240 ， 二进制解释： 1111 0000
>>	右移动运算符	a >> 2 输出结果 15 ， 二进制解释： 0000 1111
>>>	无符号右移	A >>>2 输出结果 15, 二进制解释: 0000 1111

3.3 运算符的优先级

1） 运算符有不同的优先级，所谓优先级就是表达式运算中的运算顺序。如右表，上一行运算符总优先于下一行。
2） 只有单目运算符、赋值运算符是从右向左运算的。
3） 运算符的优先级和Java一样。

小结运算符的优先级
1.() []
2.单目运算
3.算术运算符
4.移位运算
5.比较运算符(关系运算符)
6.位运算

```
7.关系运算符
8.赋值运算
9.,
```

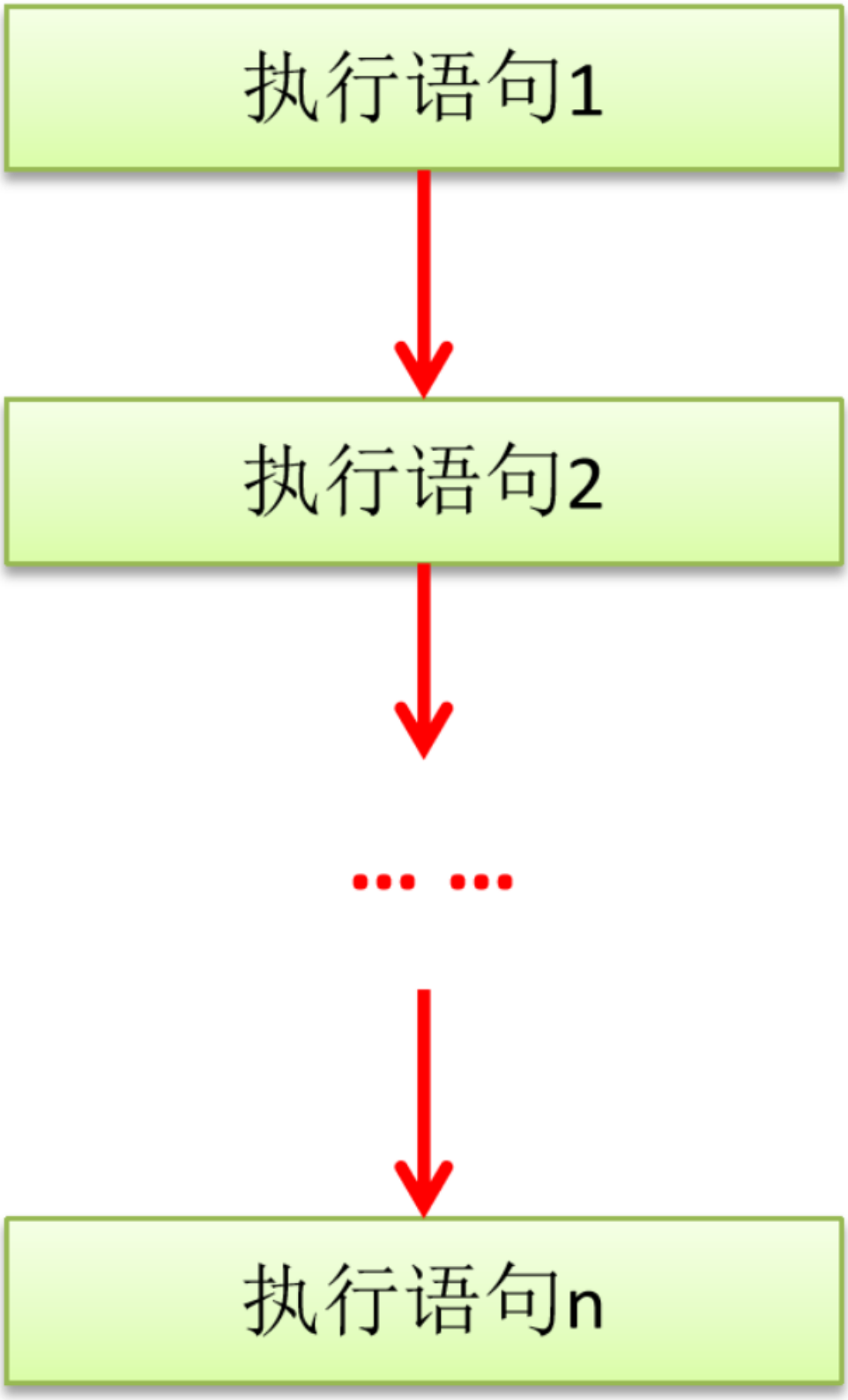
类别	运算符	关联性
1	() []	左到右
2	! ~	**右到左**
3	* / %	左到右
4	+ -	左到右
5	>> >>> <<	左到右
6	> >= < <=	左到右
7	== !=	左到右
8	&	左到右
9	^	左到右
10	|	左到右
11	&&	左到右
12	||	左到右
13	= += -= *= /= %= >>= <<= &= ^= |=	**右到左**
14	,	左到右

四、程序流程控制

4.1 介绍

```
1.顺序控制
2.分支控制
3.循环控制
```

4.2 顺序控制



1. 介绍

程序从上到下逐行地执行，中间没有任何判断和跳转。

2. 注意事项

Scala中定义变量时采用合法的前向引用。如：

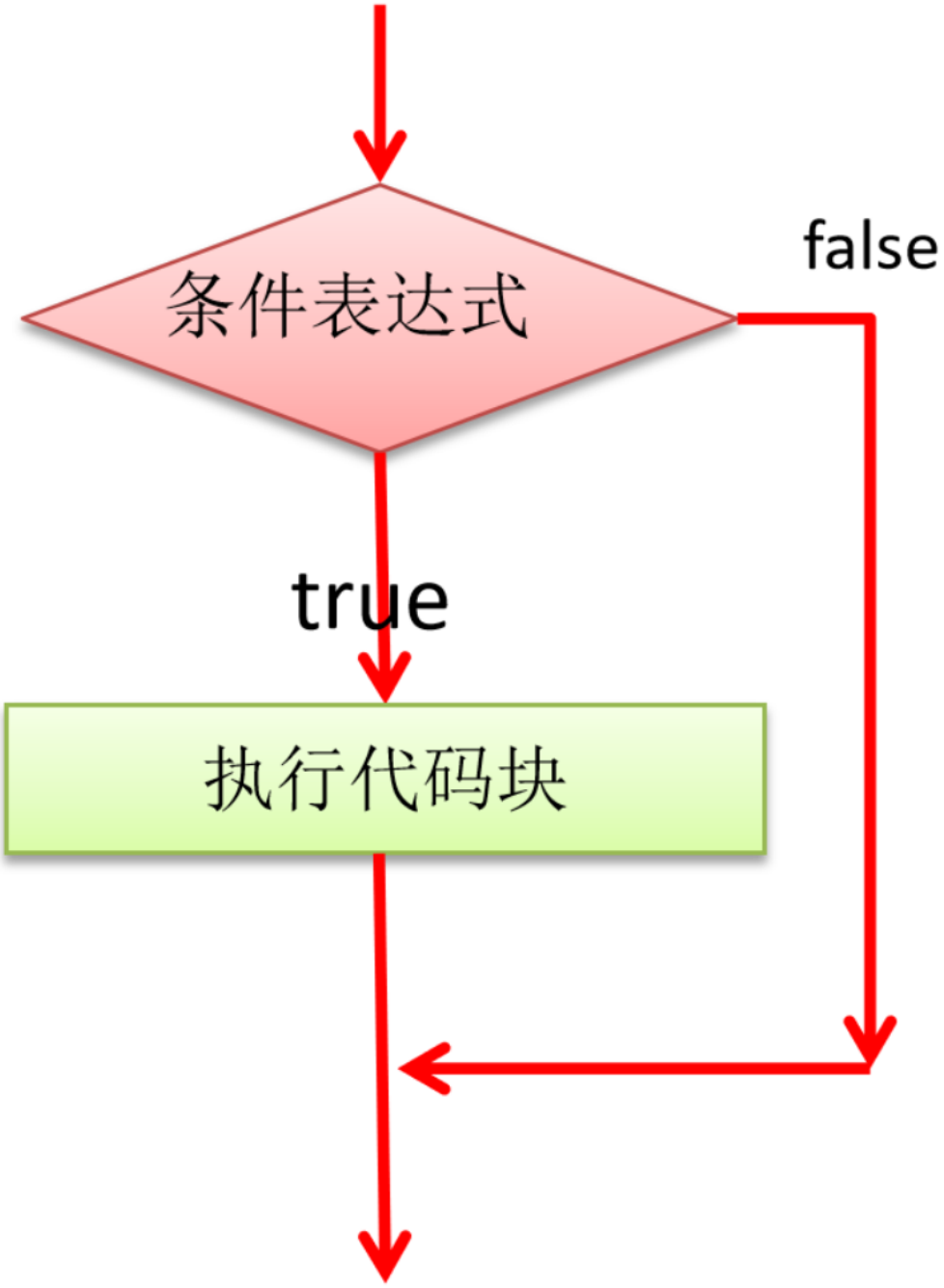
```
def main(args : Array[String]) : Unit = {
        var num1 = 12
        var num2 = num1 + 2
}
```

4.3 分支控制

1. 介绍

让程序有选择的的执行,分支控制有三种:
1.单分支
2.双分支
3.多分支

2. 单分支

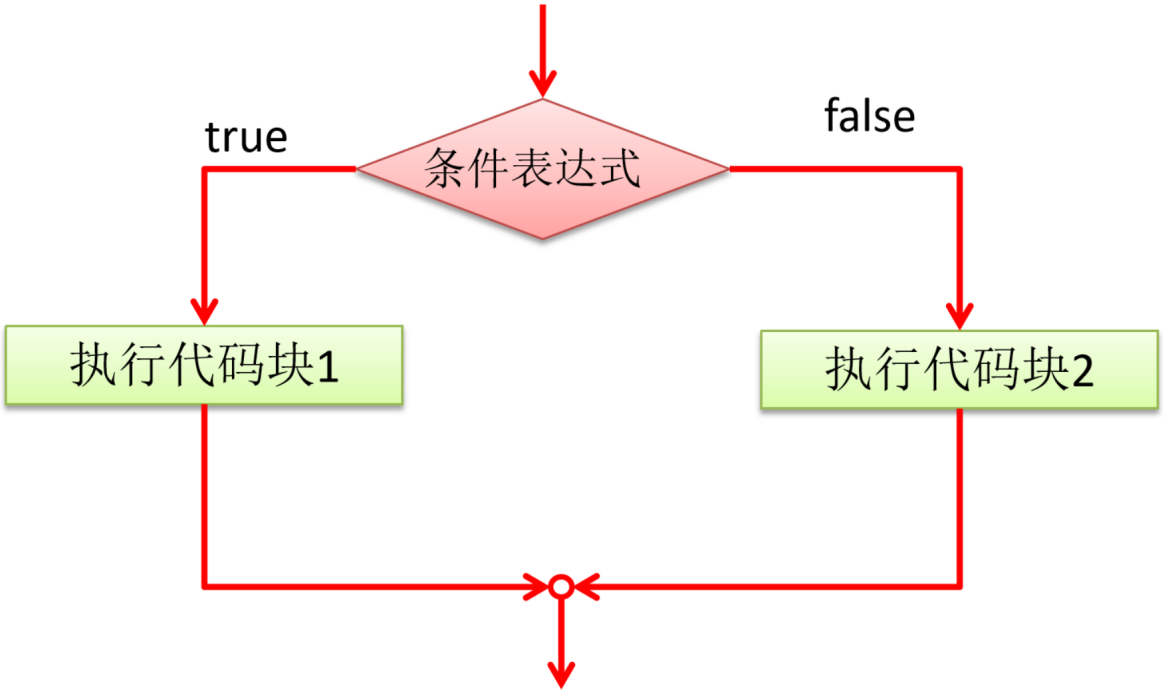


基本语法

```
if (条件表达式) {
    执行代码块
}
```

说明：当条件表达式为ture 时，就会执行 { } 的代码。

3. 双分支

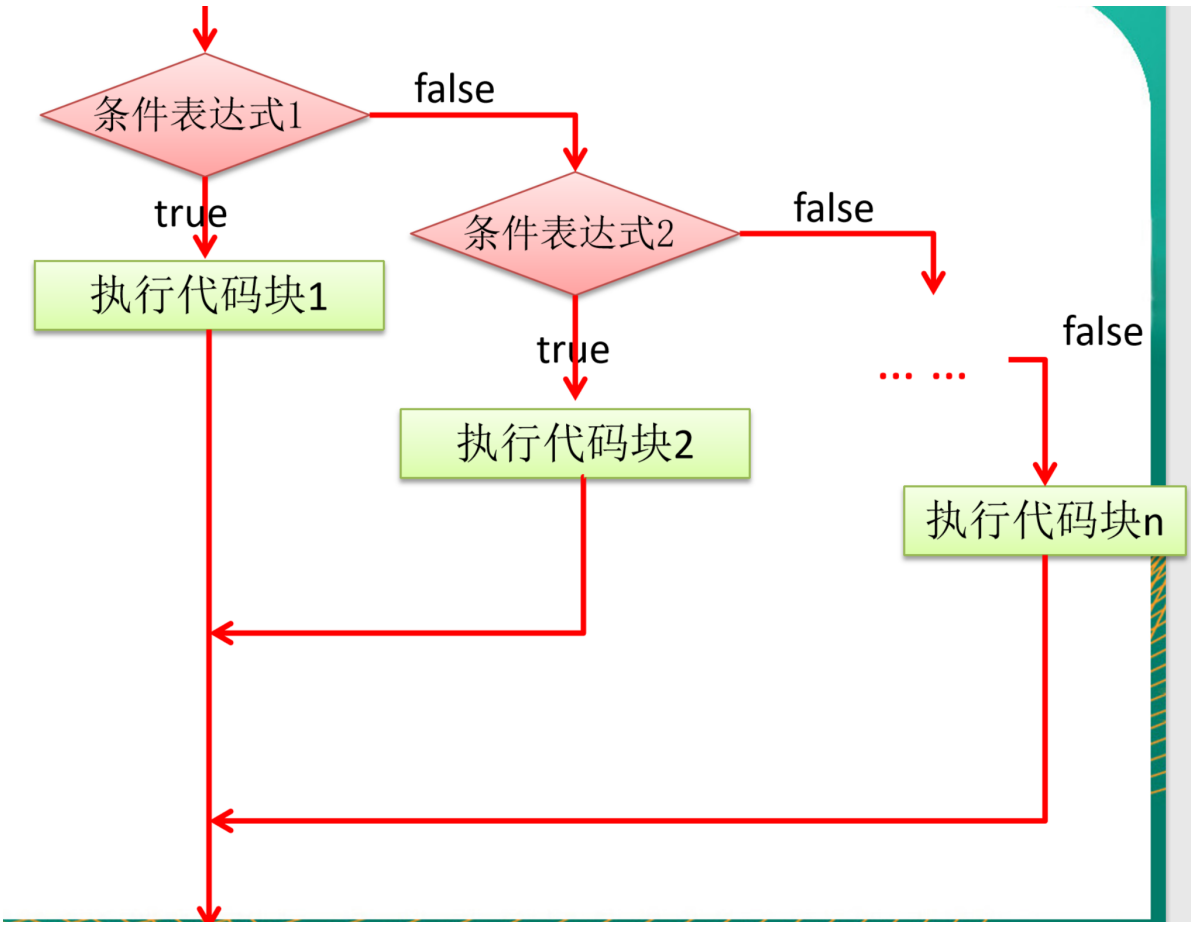


基本语法

```
if (条件表达式) {
执行代码块1
} else {
执行代码块2
      }
```

说明：当条件表达式成立，即执行代码块1，否则执行代码块2

4. 多分支



基本语法

```
if (条件表达式1) {
执行代码块1
}
else if (条件表达式2) {
执行代码块2
}
……
else {
执行代码块n
}
```

5. 注意事项

1.如果大括号{}内的逻辑代码只有一行，大括号可以省略，这点和java 的规定一样。
2.Scala中任意表达式都是有返回值的，也就意味着if else表达式其实是有返回结果的，具体返回结果的值取决于满足条件的代码体的最后一行内容.
3.Scala中是没有三元运算符，因为可以这样简写 `val result = if (flg) 1 else 0`

4.4 嵌套分支

1. 基本介绍

在一个分支结构中又完整的嵌套了另一个完整的分支结构，里面的分支的结构称为内层分支外面的分支结构称为外层分支。嵌套分支不要超过3层

2. 基本语法

```
if(){
  if(){
   }else{
   }
}
```

4.5 switch分支结构

在scala中没有switch,而是使用模式匹配来处理。
match-case

4.6 for循环控制

1. 介绍

Scala也为for循环这一常见的控制结构提供了非常多的特性，这些for循环的特性被称为for推导式(for comprehension)或for表达式(for expression)

2. 范围数据循环方式

1. 方式一

```
for(i <- 1 to 3){
  print(i + " ")
}
println()

说明
1.i表示循环的变量， <- 规定好 to 规定
2.i将会从 1-3 循环，前后闭合
```

2. 方式二

```
for(i <- 1 until 3) {
  print(i + " ")
}
println()
说明:
这种方式和前面的区别在于 i 是从1 到 3-1
前闭合后开的范围,和java的arr.length() 类似
for (int i = 0; i < arr.lenght; i++){}
```

3. 循环守卫

```
for(i <- 1 to 3 if i != 2) {
  print(i + " ")
}
println()

循环守卫，即循环保护式（也称条件判断式，守卫）。保护式为true则进入循环体内部，为false则跳过，类似于continue
上面的代码等价 for (i <- 1 to 3) {if (i != 2) {println(i+"")}}
```

4. 引入变量

```
for(i <- 1 to 3; j = 4 - i) {
  print(j + " ")
}
```

没有关键字，所以范围后一定要加；来隔断逻辑
上面的代码等价for (i <- 1 to 3) { val j = 4-i print(j+"")}

5. 嵌套循环

```
for(i <- 1 to 3; j <- 1 to 3) {
  println(" i =" + i + " j = " + j)
}
```

没有关键字，所以范围后一定要加；来隔断逻辑
上面的代码等价

```
for ( i <- 1 to 3) {
    for ( j <- 1 to 3)
      println(i + " " + j + " ")
    }
}
```

6. 循环返回值

```
val res = for(i <- 1 to 10) yield i
println(res)
```

将遍历过程中处理的结果返回到一个新Vector集合中，使用yield关键字

7. 使用花括号代替小括号

```
or(i <- 1 to 3; j =  i * 2) {
  println(" i= " + i + " j= " + j)
}
```

可以写成

```
for{
      i <- 1 to 3
      j = i * 2} {
      println(" i= " + i + " j= " + j)
 }
```

1.{}和()对于for表达式来说都可以
2.for 推导式有一个不成文的约定：当for 推导式仅包含单一表达式时使用圆括号，当其包含多个表达式时使用大括号
3.当使用{} 来换行写表达式时，分号就不用写了

4.7 while循环控制

1. 基本语法

```
循环变量初始化
while (循环条件) {
     循环体(语句)
     循环变量迭代
}
```

2. 注意事项和细节说明

1.循环条件是返回一个布尔值的表达式
2.while循环是先判断再执行语句
3.与If语句不同，while语句本身没有值，即整个while语句的结果是Unit类型的()
4.因为while中没有返回值,所以当要用该语句来计算并返回结果时,就不可避免的使用变量,而变量需要声明在while循环的外部，那么就等同于循环的内部对外部的变量造成了影响，所以不推荐使用，而是推荐使用for循环。

3. 中断

scala内置控制结构特地去掉了break和continue，是为了更好的适应函数化编程，推荐使用函数式的风格解决break和contine的功能，而不是一个关键字。

```
import util.control.Breaks._
breakable{
    while(n<=20){
        n += 1
        if(n==18){
            break()
        }
    }
}
```

4. 跳出循环

Scala内置控制结构特地也去掉了continue，是为了更好的适应函数化编程，可以使用if-else或是循环守卫实现continue的效果

案例

```
for (i <- 1 to 10 if (i != 2 && i != 3)) {
println("i=" + i)
}
```

4.8 do-while

1. 基本语法

```
循环变量初始化;
do{
循环体(语句)
循环变量迭代
} while(循环条件)
```

2. 注意事项和细节说明

1.循环条件是返回一个布尔值的表达式
2.do..while循环是先执行，再判断
3.和while 一样，因为do…while中没有返回值,所以当要用该语句来计算并返回结果时,就不可避免的使用变量 ，而变量需要声明在do...while循环的外部，那么就等同于循环的内部对外部的变量造成了影响，所以不推荐使用，而是推荐使用for循环

五、函数式编程基础

5.1 几个概念的说明

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 。

5.2 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。

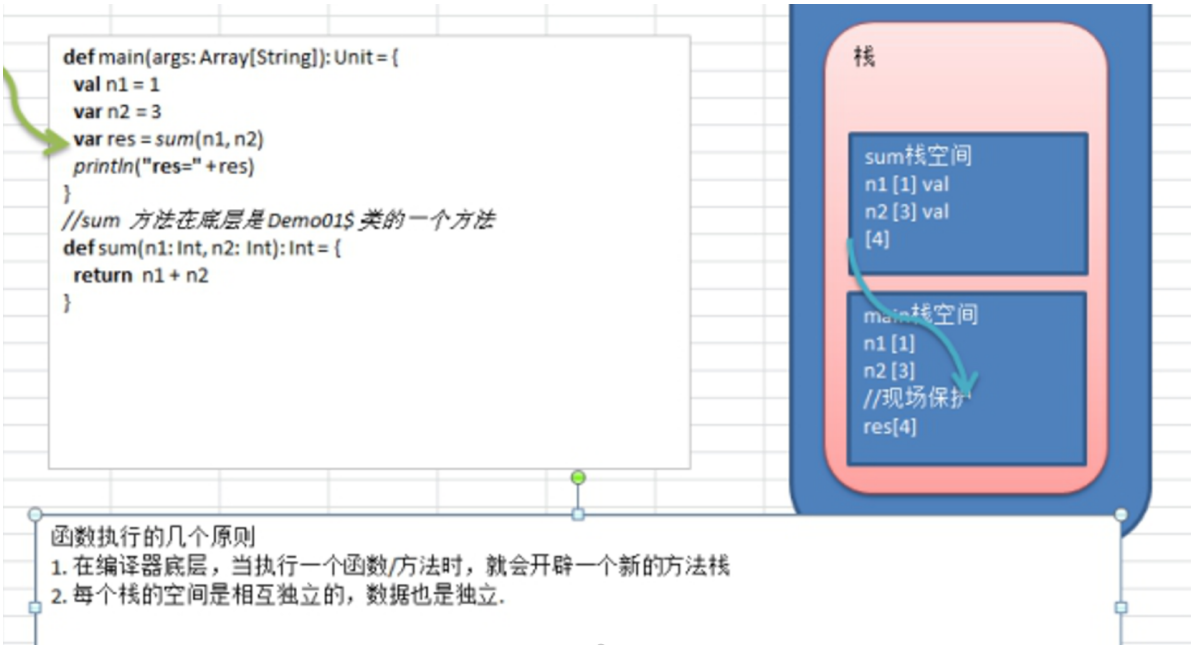
5.3 函数

1. 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

2. 调用机制



3. 递归调用

1. 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

2. 需要遵守的原则

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。

4. 函数注意事项和细节讨论

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。
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
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()

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)
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。
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
11.scala函数的形参默认是val的，因此不能在函数中进行修改
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
13.Scala函数支持可变参数

```
    def sum(n1: Int, args: Int*) = {
        println("args length =" + args.length) //求出长度
        var result = n1
        for (arg <- args) { //(遍历)
```

```
            result += arg
        }
    def main(args: Array[String]): Unit = {
        println(sum(1))
        println(sum(1,2,3,4,5))
      }
```

5. 过程

1. 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

2. 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 ={
 name + " hello "
 }

 //上面的写法也可以写成，去掉 返回类型和 =
def f10(name: String){
    name + " hello "
}
```

3. 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。
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.

6. 惰性函数

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。

1. 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)。

2. 注意事项和细节

```
1.lazy不能修饰var类型的变量
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
}
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
```

5.4 异常

1. 介绍

1.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。
2.语法处理上和Java类似，但是又不尽相同

2. 案例说明

```
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...")
    }
```

3. 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
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。
6.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。
7.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
}
```

六、面向对象编程

6.1 类与对象

1. 类的定义

```
[修饰符] class 类名 {
    类体
}
```

2. 定义类的注意事项

```
1.scala语法中，类并不声明为public，所有这些类都具有公有可见性(即默认就是public),[修饰符在后面再详解].
2.一个Scala源文件可以包含多个类.
```

3. 属性/成员变量

```
属性是类的一个组成部分，一般是值数据类型,也可是引用类型。
```

4. 属性/成员变量的注意事项和细节说明

1.属性的定义语法同变量，示例：[访问修饰符] var 属性名称 [：类型] = 属性值
2.属性的定义类型可以为任意类型，包含值类型或引用类型[案例演示]
3.Scala中声明一个属性,必须显示的初始化，然后根据初始化数据的类型自动推断，属性类型可以省略(这点和Java不同)。
4.如果赋值为null,则一定要加类型，因为不加类型，那么该属性的类型就是Null类型.
5.如果在定义属性时，暂时不赋值，也可以使用符号_(下划线)，让系统分配默认值.
6.不同对象的属性是独立，互不影响，一个对象对属性的更改，不影响另外一个。

类型	_ 对应的值
Byte Short Int Long	0
Float Double	0.0
String 和 引用类型	null
Boolean	false

6.2 对象的创建

1. 基本语法

```
val | var 对象名 [：类型]  = new 类型()
对象名.属性名
```

2. 说明

1.如果我们不希望改变对象的引用(即：内存地址)，应该声明为val 性质的，否则声明为var，scala设计者推荐使用val ,因为一般来说，在程序中，我们只是改变对象属性的值，而不是改变对象的引用。
2.scala在声明对象变量时，可以根据创建对象的类型自动推断，所以类型声明可以省略，但当类型和后面new 对象类型有继承关系即多态时，就必须写了

3. 创建流程

```
class Person {
    var age: Short = 90
    var name: String = _

    def this(n: String, a: Int) {
        this()
        this.name = n
        this.age = a
    }
}
var p : Person = new Person("小倩",20)

1.加载类的信息(属性信息，方法信息)
2.在内存中(堆)开辟空间
3.使用父类的构造器(主和辅助)进行初始
4.使用主构造器对属性进行初始化 【age:90, naem nul】
5.使用辅助构造器对属性进行初始化 【 age:20, naem 小倩 】
6.将开辟的对象的地址赋给p这个引用
```

4. 创建对象的方式

```
1.new对象
2.apply创建
3.匿名子类方式
4.动态混入
```

6.3 方法

1. 基本语法

```
def 方法名(参数列表) [: 返回值类型] = {
    方法体
}
```

2. 调用机制原理

1.当我们scala开始执行时，先在栈区开辟一个main栈。main栈是最后被销毁
2.当scala程序在执行到一个方法时，总会开一个新的栈。
3.每个栈是独立的空间，变量（基本数据类型）是独立的，相互不影响（引用类型除外）
4.当方法执行完毕后，该方法开辟的栈就会被jvm机回收

6.4 构造器

1. 基本介绍

构造器(constructor)又叫构造方法，是类的一种特殊的方法，它的主要作用是完成对新对象的初始化

2. 基本语法

```
class 类名(形参列表) {  // 主构造器
   // 类体
   def  this(形参列表) {  // 辅助构造器
   }
   def  this(形参列表) {  //辅助构造器可以有多个...
   }
}

1. 辅助构造器函数的名称this，可以有多个，编译器通过不同参数来区分.
```

3. 注意事项和细节

1.Scala构造器作用是完成对新对象的初始化，构造器没有返回值。
2.主构造器的声明直接放置于类名之后 [反编译]
3.主构造器会执行类定义中的所有语句，这里可以体会到Scala的函数式编程和面向对象编程融合在一起，即：构造器也是方法（函数），传递参数和使用方法和前面的函数部分内容没有区别
4.如果主构造器无参数，小括号可省略，构建对象时调用的构造方法的小括号也可以省略
5.辅助构造器名称为this（这个和Java是不一样的），多个辅助构造器通过不同参数列表进行区分， 在底层就是构造器重载。
6.如果想让主构造器变成私有的，可以在()之前加上private，这样用户只能通过辅助构造器来构造对象了
7.辅助构造器的声明不能和主构造器的声明一致,会发生错误(即构造器名重复)

```
class Person private() {
  var name: String = _
  var age: Int = _
  def this(name : String) {
    //辅助构造器无论是直接或间接，最终都一定要调用主构造器，执行主构造器的逻辑
    //而且需要放在辅助构造器的第一行[这点和java一样，java中一个构造器要调用同类的其它构造器，也需要放在第一行]
    this()  //直接调用主构造器
    this.name = name
  }
  def this(name : String, age : Int) {
    this() //直接调用主构造器
    this.name = name
    this.age = age
  }
  def this(age : Int) {
    this("匿名") //简介调用主构造器,因为 def this(name : String) 中调用了主构造器!
    this.age = age
  }
```

4. 构造器参数

1.Scala类的主构造器的形参未用任何修饰符修饰，那么这个参数是局部变量。
2.如果参数使用val关键字声明，那么Scala会将参数作为类的私有的只读属性使用
3.如果参数使用var关键字声明，那么那么Scala会将参数作为类的成员属性使用,并会提供属性对应的xxx()[类似getter]/xxx_$eq()[类似setter]方法，即这时的成员属性是私有的，但是可读写。

5. Bean属性

JavaBeans规范定义了Java的属性是像getXxx（）和setXxx（）的方法。许多Java工具（框架）都依赖这个命名习惯。为了Java的互操作性。将Scala字段加@BeanProperty时，这样会自动生成规范的 setXxx/getXxx 方法。这时可以使用 对象.setXxx()和对象.getXxx() 来调用属性。

6.5 包与包对象

1. 包

1. 基本介绍

和Java一样，Scala中管理项目可以使用包，但Scala中的包的功能更加强大，使用也相对复杂些。

2. 基本语法

```
package 包名
```

3. 包的三大作用

1.区分相同名字的类
2.当类很多时,可以很好的管理类
3.控制访问范围

4. 包的命名

1. 命名规则

只能包含数字、字母、下划线、小圆点.,但不能用数字开头， 也不要使用关键字。
demo.class.exec1 //错误 ， 因为class是关键字 demo.12a   //错误，因为不能以数字开头

2. 命名规范

一般是小写字母+小圆点一般是
com.公司名.项目名.业务模块名

5. 不需要显式引用的包

java.lang.*
scala包
Predef包

6. 注意事项和使用细节

1.包也可以像嵌套类那样嵌套使用（包中有包），这个在前面的第三种打包方式已经讲过了，在使用第三种方式时的好处是：程序员可以在同一个文件中，将类(class / object)、trait 创建在不同的包中，这样就非常灵活了。
2.作用域原则：可以直接向上访问。即：Scala中子包中直接访问父包中的内容，大括号体现作用域。(提示：Java中子包使用父包的类，需要import)。在子包和父包类重名时，默认采用就近原则，如果希望指定使用某个类，则带上包名即可。
3.父包要访问子包的内容时，需要import对应的类等
4.可以在同一个.scala文件中，声明多个并列的package(建议嵌套的pakage不要超过3层)
5.包名可以相对也可以绝对，比如，访问BeanProperty的绝对路径是：_root_. scala.beans.BeanProperty ，在一般情况下：我们使用相对路径来引入包，只有当包名冲突时，使用绝对路径来处理。

```
class Manager( var name : String ) {
  //第一种形式
  //@BeanProperty var age: Int = _
  //第二种形式, 和第一种一样，都是相对路径引入
  //@scala.beans.BeanProperty var age: Int = _
  //第三种形式, 是绝对路径引入，可以解决包名冲突
  @_root_. scala.beans.BeanProperty var age: Int = _
}
```

2. 包对象

1. 基本介绍

包可以包含类、对象和特质trait，但不能包含函数/方法或变量的定义。这是Java虚拟机的局限。为了弥补这一点不足，scala提供了包对象的概念来解决这个问题。

```
package object scala {
    var name = "jack"
    def sayOk(): Unit = {
      println("package object sayOk!")
    }
  }
package scala {
    class Test {
        def test() : Unit ={
            //这里的name就是包对象scala中声明的name
```

```
            println(name)
            sayOk()//这个sayOk 就是包对象scala中声明的sayOk
        }
    }
```

2. 包对象的底层实现机制分析

```
1.当创建包对象后，在该包下生成 public final class package 和 public final class package$
2.通过 package$ 的一个静态实例完成对包对象中的属性和方法的调用
```

3. 注意事项

```
1.每个包都可以有一个包对象。你需要在父包中定义它。
2.包对象名称需要和包名一致，一般用来对包的功能补充。
```

3. 包的可见性

1.当属性访问权限为默认时，从底层看属性是private的，但是因为提供了xxx_$eq()[类似setter]/xxx()[类似getter] 方法，因此从使用效果看是任何地方都可以访问)
2.当方法访问权限为默认时，默认为public访问权限
3.private为私有权限，只在类的内部和伴生对象中可用
protected为受保护权限，scala中受保护权限比Java中更严格，只能子类访问，同包无法访问 (编译器)
4.在scala中没有public关键字,即不能用public显式的修饰属性和方法
5.包访问权限（表示属性有了限制。同时包也有了限制），这点和Java不一样，体现出Scala包使用的灵活性。

```
package com.bytedance.scala
class Person {
  private[scala] val pname="hello" // 增加包访问权限后，1.private同时起作用。不仅同类可以使用
                                   //2. 同时com.bytedance.scala中包下其他类也可以使用
}
```

当然，也可以将可见度延展到上层包

```
private[bytedance] val description="zhangsan"
```

说明：private也可以变化，比如protected[bytedance]， 非常的灵活。

4. 包的引入

1.在Scala中，import语句可以出现在任何地方，并不仅限于文件顶部，import语句的作用一直延伸到包含该语句的块末尾。这种语法的好处是：在需要时在引入包，缩小import 包的作用范围，提高效率。
2.Java中如果想要导入包中所有的类，可以通过通配符*，Scala中采用下 _
3.如果不想要某个包中全部的类，而是其中的几个类，可以采用选取器(大括号)
4.如果引入的多个包中含有相同的类，那么可以将不需要的类进行重命名进行区分，这个就是重命名。

```
import java.util.{ HashMap=>JavaHashMap, List}
import scala.collection.mutable._
var map = new HashMap() // 此时的HashMap指向的是scala中的HashMap
var map1 = new JavaHashMap(); // 此时使用的java中hashMap的别名
```

5.如果某个冲突的类根本就不会用到，那么这个类可以直接隐藏掉

```
import java.util.{ HashMap=>_, _} // 含义为 引入java.util包的所有类，但是忽略 HahsMap类.
var map = new HashMap() // 此时的HashMap指向的是scala中的HashMap, 而且idea工具的提示也不会显示java.util的HashMaple
```

6.6 嵌套类

1. 基本介绍

在Scala中，你几乎可以在任何语法结构中内嵌任何语法结构。如在类中可以再定义一个类，这样的类是嵌套类，其他语法结构也是一样。
嵌套类类似于Java中的内部类。

2. 访问方式

方式一、内部类如果想要访问外部类的属性，可以通过外部类对象访问。 -->
即：访问方式：外部类名.this.属性名

```
class ScalaOuterClass {
  var name : String = "scott"
  private var sal : Double = 1.2
  class ScalaInnerClass { //成员内部类
    def info() = {
      // 访问方式：外部类名.this.属性名
```

```
      // 怎么理解 ScalaOuterClass.this 就相当于是
ScalaOuterClass 这个外部类的一个实例,
      // 然后通过 ScalaOuterClass.this 实例对象去访问 name 属性
      // 只是这种写法比较特别，学习java的同学可能更容易理解
ScalaOuterClass.class 的写法.
      println("name = " + ScalaOuterClass.this.name
        + " age =" + ScalaOuterClass.this.sal)
    }
  }
}

object ScalaOuterClass {  //伴生对象
  class ScalaStaticInnerClass { //静态内部类
  }
}
//调用成员内部类的方法
inner1.info()
```

方式二、内部类如果想要访问外部类的属性，也可以通过外部类别名访问(推荐)。--> 即：访问方式：外部类名别名.属性名

```
class ScalaOuterClass {
  myOuter =>  //这样写，你可以理解成这样写，myOuter就是代表外部类
的一个对象.
  class ScalaInnerClass { //成员内部类
    def info() = {
      println("name = " + ScalaOuterClass.this.name
        + " age =" + ScalaOuterClass.this.sal)
      println("----------------------------------")
      println("name = " + myOuter.name
        + " age =" + myOuter.sal)
    }
  }
  // 当给外部指定别名时，需要将外部类的属性放到别名后.
  var name : String = "scott"
  private var sal : Double = 1.2
}

object ScalaOuterClass {  //伴生对象
  class ScalaStaticInnerClass { //静态内部类
  }
```

```
}
inner1.info()
```

3. 类型投影

一个例子：

```
class ScalaOuterClass3 {
  myOuter =>
  class ScalaInnerClass3 { //成员内部类
    def test(ic: ScalaInnerClass3): Unit = {
      System.out.println(ic)
    }
  }
}


object Scala01_Class {
    def main(args: Array[String]): Unit = {
        val outer1 : ScalaOuterClass3 = new
ScalaOuterClass3()
        val outer2 : ScalaOuterClass3 = new
ScalaOuterClass3()
        val inner1 = new outer1.ScalaInnerClass3()
        val inner2 = new outer2.ScalaInnerClass3()
        inner1.test(inner1) // ok, 因为 需要
outer1.ScalanInner
        inner1.test(inner2) // error, 需要
outer1.ScalanInner•outer2.ScalanInner
      }
}

//说明下面调用test 的 正确和错误的原因：
//1.Java中的内部类从属于外部类,因此在java中 inner.test(inner2)
就可以，因为是按类型来匹配的。
//2 Scala中内部类从属于外部类的对象，所以外部类的对象不一样，创建出来
的内部类也不一样，无法互换使用
//3.比如你使用ideal 看一下在inner1.test()的形参上，它提示的类型是
outer1.ScalaOuterClass, 而不是ScalaOuterClass
inner1.test(inner1) // ok
inner1.test(inner2) // 错误
```

类型投影是指：在方法声明上，如果使用 外部类#内部类 的方式，表示忽略内部类的对象关系，等同于Java中内部类的语法操作，我们将这种方式称之为类型投影（即：忽略对象的创建方式，只考虑类型）

6.7 抽象类

1. 基本介绍

在Scala中，通过abstract关键字标记不能被实例化的类。方法不用标记abstract，只要省掉方法体即可。抽象类可以拥有抽象字段，抽象字段/属性就是没有初始值的字段

2. 基本语法

```
abstract class Person() { // 抽象类
  var name: String // 抽象字段, 没有初始化
  def printName // 抽象方法, 没有方法体
}
```

3. 注意事项和细节讨论

1.抽象类不能被实例化
2.抽象类不一定要包含abstract方法。也就是说,抽象类可以没有abstract方法
3.一旦类包含了抽象方法或者抽象属性,则这个类必须声明为abstract
4,抽象方法不能有主体，不允许使用abstract修饰。
5.如果一个类继承了抽象类，则它必须实现抽象类的所有抽象方法和抽象属性，除非它自己也声明为abstract类。
6.抽象方法和抽象属性不能使用private、final 来修饰，因为这些关键字都是和重写/实现相违背的。
7.抽象类中可以有实现的方法.
8.子类重写抽象方法不需要override，写上也不会错.

4. 匿名子类

```
abstract class Monster{
  var name : String
  def cry()
}


var monster = new Monster {
      override var name: String = "牛魔王"
      override def cry(): Unit = {
        println("牛魔王哼哼叫唤..")
      }
}
```

6.8 特质

1. 基本介绍

从面向对象来看，接口并不属于面向对象的范畴，Scala是纯面向对象的语言，在Scala中，没有接口。

Scala语言中，采用特质trait（特征）来代替接口的概念，也就是说，多个类具有相同的特征（特征）时，就可以将这个特质（特征）独立出来，采用关键字trait声明。 理解trait 等价于(interface + abstract class)

2. 基本语法

```
trait 特质名 {
    trait体
}
```

3. 使用

一个类具有某种特质（特征），就意味着这个类满足了这个特质（特征）的所有要素，所以在使用时，也采用了extends关键字，如果有多个特质或存在父类，那么需要采用with关键字连接

1.没有父类
```
class  类名  extends  特质1  with  特质2  with  特质3
..
```
2.有父类
```
class  类名  extends  父类  with  特质1  with  特质2
with  特质3
```

4. 说明

1.Scala提供了特质（trait），特质可以同时拥有抽象方法和具体方法，一个类可以实现/继承多个特质
2.特质中没有实现的方法就是抽象方法。类通过extends继承特质，通过with可以继承多个特质
3.所有的java接口都可以当做Scala特质使用

5. 动态混入特质

1.除了可以在类声明时继承特质以外，还可以在构建对象时混入特质，扩展目标类的功能
2.此种方式也可以应用于对抽象类功能进行扩展
3.动态混入是Scala特有的方式（java没有动态混入），可在不修改类声明/定义的情况下，扩展类的功能，非常的灵活，耦合性低 。
4.动态混入可以在不影响原有的继承关系的基础上，给指定的类扩展功能。

```
trait Operate3 {
  def insert(id: Int): Unit = {
    println("插入数据 = " + id)
  }
}

class OracleDB {
}

abstract class MySQL3 {
}
```

```
var oracle = new OracleDB with  Operate3
oracle.insert(999)
val mysql = new MySQL3 with Operate3
mysql.insert(4)
```

1. 叠加特质

1. 基本介绍

构建对象的同时如果混入多个特质，称之为叠加特质，那么特质声明顺序从左到右，方法执行顺序从右到左。

2. 应用案例

```
trait Operate4 {
  println("Operate4...")
  def insert(id : Int)
}

trait Data4 extends Operate4 {
  println("Data4")
  override  def insert(id : Int): Unit = {
    println("插入数据 = " + id)
  }
}

trait DB4 extends Data4 {
  println("DB4")
  override def insert(id : Int): Unit = {
    print("向数据库")
    super.insert(id)
  }
}

trait File4 extends  Data4 {
  println("File4")
  override def insert(id : Int): Unit = {
    print("向文件")
    super.insert(id)
  }
}
```

```
class MySQL4 {}

// 1.Scala在叠加特质的时候，会首先从后面的特质开始执行
// 2.Scala中特质中如果调用super，并不是表示调用父特质的方法，而是向前面（左边）继续查找特质，如果找不到，才会去父特质查找
val mysql = new MySQL4 with DB4 with File4
//val mysql = new MySQL4 with File4 with DB4
mysql.insert(888)
```

3. 注意事项和细节

1.特质声明顺序从左到右。
2.Scala在执行叠加对象的方法时，会首先从后面的特质(从右向左)开始执行
3.Scala中特质中如果调用super，并不是表示调用父特质的方法，而是向前面（左边）继续查找特质，如果找不到，才会去父特质查找
4.如果想要调用具体特质的方法，可以指定：super[特质].xxx(…).其中的泛型必须是该特质的直接超类类型

2. 在特质中重写抽象方法的特例

方式1 ： 去掉 super()...
方式2： 调用父特质的抽象方法，那么在实际使用时，没有方法的具体实现，无法编译通过，为了避免这种情况的发生。可重写抽象方法，这样在使用时，就必须考虑动态混入的顺序问题

```
trait Operate5 {
    def insert(id : Int)
}

trait File5 extends Operate5 {
    abstract override
    def insert( id : Int ): Unit = {
        println("将数据保存到文件中..")
        super.insert(id)
    }
}

trait DB5 extends  Operate5 {
    def insert( id : Int ): Unit = {
        println("将数据保存到数据库中..")
    }
```

```
}
class MySQL5 {}
val mysql5 = new MySQL5 with DB5  with File5
```

理解 abstract override 的小技巧分享：
可以这样理解，当我们给某个方法增加了abstract override 后，就是明确的告诉编译器，该方法确实是重写了父特质的抽象方法，但是重写后，该方法仍然是一个抽象方法（因为没有完全的实现，需要其它特质继续实现[通过混入顺序]）

6. 富特质

即该特质中既有抽象方法，又有非抽象方法

```
trait Operate {
    def insert( id : Int ) //抽象
    def pageQuery(pageno:Int, pagesize:Int): Unit = { //实现
        println("分页查询")
    }
}
```

7. 特质中的具体字段

特质中可以定义具体字段，如果初始化了就是具体字段，如果不初始化就是抽象字段。混入该特质的类就具有了该字段，字段不是继承，而是直接加入类，成为自己的字段。

```
trait Operate6 {
  var opertype : String
  def insert()
}

trait DB6 extends  Operate6 {
  var opertype : String = "insert"
  def insert(): Unit = {
  }
}
class MySQL6 {}
```

```
var mysql = new MySQL6 with DB6
//通过反编译，可以看到 opertype
println(mysql.opertype)
```

8. 特质构造顺序

1. 声明类的同时混入特质

```
1.调用当前类的超类构造器
2.第一个特质的父特质构造器
3.第一个特质构造器
4.第二个特质构造器的父特质构造器，如果已经执行过，就不再执行
5.第二个特质构造器
6........重复4,5的步骤(如果有第3个，第4个特质)
7.当前类构造器
```

2. 创建对象时，动态混入特质

```
1.调用当前类的超类构造器
2.当前类构造器
3.第一个特质构造器的父特质构造器
4.第一个特质构造器.
5.第二个特质构造器的父特质构造器，如果已经执行过，就不再执行
6.第二个特质构造器
7........重复5,6的步骤(如果有第3个，第4个特质)
8.当前类构造器    [案例演示]
```

3.分析

```
第1种方式实际是构建类对象，在混入特质时，该对象还没有创建。
第2种方式实际是构造匿名子类，可以理解成在混入特质时，对象已经创建了。
```

9. 扩展类的特质

```
1.特质可以继承类，以用来拓展该类的一些功能
2.所有混入该特质的类，会自动成为那个特质所继承的超类的子类
3.如果混入该特质的类，已经继承了另一个类(A类)，则要求A类是特质超类的子类，否则就会出现了多继承现象，发生错误。
```

10. 自身类型

1. 介绍

主要是为了解决特质的循环依赖问题，同时可以确保特质在不扩展某个类的情况下，依然可以做到限制混入该特质的类的类型。

2. 代码说明

```
//Logger就是自身类型特质
trait Logger {
  // 明确告诉编译器，我就是Exception,如果没有这句话，下面的
getMessage不能调用
  this: Exception =>
  def log(): Unit ={
    // 既然我就是Exception, 那么就可以调用其中的方法
    println(getMessage)
  }
}

class Console extends  Logger {} //对吗? 错 Console不是
Exception的子类
class Console extends Exception with Logger//对吗?
```

6.9 封装

1. 基本介绍

封装(encapsulation)就是把抽象出的数据和对数据的操作封装在一起,数据被保护在内部,程序的其它部分只有通过被授权的操作(成员方法),才能对数据进行操作。

2. 优点

1.隐藏实现细节
2.可以对数据进行验证，保证安全合理

3. 实现步骤

```
1.将属性进行私有化
2.提供一个公共的set方法，用于对属性判断并赋值
def  setXxx(参数名 : 类型) : Unit = {
    //加入数据验证的业务逻辑
    属性 = 参数名
}

3.提供一个公共的get方法，用于获取属性的值
def getXxx() [: 返回类型] = {
    return 属性
}
```

4. 注意事项和细节

```
1.Scala中为了简化代码的开发，当声明属性时，本身就自动提供了对应setter/getter方法，如果属性声明为private的，那么自动生成的setter/getter方法也是private的，如果属性省略访问权限修饰符，那么自动生成的setter/getter方法是public的
2.因此我们如果只是对一个属性进行简单的set和get ，只要声明一下该属性(属性使用默认访问修饰符) 不用写专门的get、set，默认会创建，访问时，直接对象.变量。这样也是为了保持访问一致性
3.从形式上看 dog.food 直接访问属性，其实底层仍然是访问的方法
4.有了上面的特性，目前很多新的框架，在进行反射时，也支持对属性的直接反射
```

6.10 继承

1. 基本介绍

```
继承可以解决代码复用,让我们的编程更加靠近人类思维.当多个类存在相同的属性(变量)和方法时,可以从这些类中抽象出父类(比如Student),在父类中定义这些相同的属性和方法，所有的子类不需要重新定义这些属性和方法，只需要通过extends语句来声明继承父类即可。
```

2. 基本语法

```
class 子类名 extends 父类名  { 类体 }
```

3. 优点

```
1.代码的复用性提高了
2.代码的扩展性和维护性提高
```

4. 属性的继承

子类继承了所有的属性，只是私有的属性不能直接访问，需要通过公共的方法去访问

5. 覆写字段

在Scala中，子类改写父类的字段，我们称为覆写/重写字段。覆写字段需使用override修饰

6. 重写

scala明确规定，重写一个非抽象方法需要用override修饰符，调用超类的方法使用super关键字

覆写字段的注意事项和细节
1.def只能重写另一个def(即：方法只能重写另一个方法)
2.val只能重写另一个val 属性 或 重写不带参数的def

```
class A {
   def sal(): Int = {
      return 10
  }}

class B extends A {
 override val sal : Int = 0
}
```

3.var只能重写另一个抽象的var属性

抽象属性：声明未初始化的变量就是抽象的属性,抽象属性在抽象类

var重写抽象的var属性小结

1.一个属性没有初始化，那么这个属性就是抽象属性
2.抽象属性在编译成字节码文件时，属性并不会声明，但是会自动生成抽象方法，所以类必须声明为抽象类
3.如果是覆写一个父类的抽象属性，那么override 关键字可省略 [原因：父类的抽象属性，生成的是抽象方法，因此就不涉及到方法重写的概念，因此override可省略]

7. 父类的构造器

1.类有一个主构器和任意数量的辅助构造器，而每个辅助构造器都必须先调用主构造器(也可以是间接调用.)
2.只有主构造器可以调用父类的构造器。辅助构造器不能直接调用父类的构造器。在Scala的构造器中，你不能调用super(params)

```
class Person(name: String) { //父类的构造器
}
class Emp (name: String) extends Person(name) {// 将子类参数传递给父类构造器,这种写法√

    // super(name)  (x) 没有这种语法
    def  this() {
        super("abc") // (x)不能在辅助构造器中调用父类的构造器
    }
}
```

8. 类型转换

要测试某个对象是否属于某个给定的类，可以用isInstanceOf方法。用asInstanceOf方法将引用转换为子类的引用。classOf获取对象的类名。

classOf[String]就如同Java的 String.class
obj.isInstanceOf[T]就如同Java的obj instanceof T 判断obj是不是T类型
obj.asInstanceOf[T]就如同Java的(T)obj 将obj强转成T类型

6.11 伴生类对伴生对象

1. 基本介绍

Scala中静态的概念-伴生对象 Scala语言是完全面向对象(万物皆对象)的语言，所以并没有静态的操作(即在Scala中没有静态的概念)。但是为了能够和Java语言交互(因为Java中有静态概念)，就产生了一种特殊的对象来模拟类对象，我们称之为类的伴生对象。这个类的所有静态内容都可以放置在它的伴生对象中声明和调用

2. 伴生对象-apply方法

在伴生对象中定义apply方法，可以实现： 类名(参数) 方式来创建对象实例.

```
class Cat(cName:String){
    var name:String = cName
}

object Cat{
    def apply():Cat = {
        new Cat("xx")
    }
    def apply(name:String):Cat = {
        new Cat(name)
    }
}
```

3. 伴生对象的小结

1.Scala中伴生对象采用object关键字声明，伴生对象中声明的全是 "静态"内容，可以通过伴生对象名称直接调用。
2.伴生对象对应的类称之为伴生类，伴生对象的名称应该和伴生类名一致。
3.伴生对象中的属性和方法都可以通过伴生对象名(类名)直接调用访问
4.从语法角度来讲，所谓的伴生对象其实就是类的静态方法和成员的集合
5.从技术角度来讲，scala还是没有生成静态的内容，只不过是将伴生对象生成了一个新的类，实现属性和方法的调用。[反编译看源码]
6.从底层原理看，伴生对象实现静态特性是依赖于 public static final MODULE$ 实现的
7.伴生对象的声明应该和伴生类的声明在同一个源码文件中(如果不在同一个文件中会运行错误!)，但是如果没有伴生类，也就没有所谓的伴生对象了，所以放在哪里就无所谓了。
8.如果 class A独立存在，那么A就是一个类，如果 object A独立存在，那么A就是一个"静态"性质的对象[即类对象]， 在 object A中声明的属性和方法可以通过A.属性和A.方法来实现调用
9.当一个文件中，存在伴生类和伴生对象时，文件的图标会发生变化

4. 单例模式

1. 说明

单例模式是指：保证在整个的软件系统中，某个类只能存在一个对象实例

2. 应用场景

比如Hibernate的SessionFactory，它充当数据存储源的代理，并负责创建Session对象。SessionFactory并不是轻量级的，一般情况下，一个项目通常只需要一个SessionFactory就够，这是就会使用到单例模式。
Akka [ActorySystem 单例]

3. 懒汉式

```
object TestSingleTon extends App{
  val singleTon = SingleTon.getInstance
  val singleTon2 = SingleTon.getInstance
  println(singleTon.hashCode() + " " +
singleTon2.hashCode())
}
//将SingleTon的构造方法私有化
class SingleTon private() {}
```

```
object SingleTon  {
  private var s:SingleTon = null
  def getInstance = {
    if(s == null) {
      s = new SingleTon
    }
    s
   }
}
```

4. 饿汉式

```
object TestSingleTon extends App {
  val singleTon = SingleTon.getInstance
  val singleTon2 = SingleTon.getInstance
  println(singleTon.hashCode() + " ~ " +
singleTon2.hashCode())
  println(singleTon == singleTon2)
}
//将SingleTon的构造方法私有化
class SingleTon private() {
  println("~~~")
}
object SingleTon {
  private val s: SingleTon = new SingleTon
  def getInstance = {
    s
  }
}
```

七、隐式转换

7.1 隐式函数

1. 基本介绍

隐式转换函数是以implicit关键字声明的带有单个参数的函数。这种函数将会自动应用，将值从一种类型转换为另一种类型

2. 代码样例

```
implicit def f1(d: Double): Int = {
        d.toInt
}

implicit def f2(l: Long): Int = {
    l.toInt
}
```

3. 注意事项和细节

1.隐式转换函数的函数名可以是任意的，隐式转换与函数名称无关，只与函数签名（函数参数类型和返回值类型）有关
2.隐式函数可以有多个(即：隐式函数列表)，但是需要保证在当前环境下，只有一个隐式函数能被识别

7.2 隐式转换

1. 基本介绍

如果需要为一个类增加一个方法，可以通过隐式转换来实现。

2. 代码样例

```
class MySQL{
  def insert(): Unit = {
    println("insert")
  }
}
class DB {
  def delete(): Unit = {
    println("delete")
```

```
  }
}

implicit def addDelete(mysql:MySQL): DB = {
      new DB //
}
val mysql = new MySQL
mysql.insert()
mysql.delete()
```

7.3 隐式值

1. 基本介绍

隐式值也叫隐式变量，将某个形参变量标记为implicit，所以编译器会在方法省略隐式参数的情况下去搜索作用域内的隐式值作为缺省参数

2. 代码样例

```
implicit val str1: String = "jack"
def hello(implicit name: String): Unit = {
println(name + " hello")
}
hello //调用.不带()
```

7.4 隐式类

1. 基本介绍

在scala2.10后提供了隐式类，可以使用implicit声明类，隐式类的非常强大，同样可以扩展类的功能，比前面使用隐式转换丰富类库功能更加的方便，在集合中隐式类会发挥重要的作用

隐式类使用有如下几个特点：

1.其所带的构造参数有且只能有一个
2.隐式类必须被定义在“类”或“伴生对象”或“包对象”里，即隐式类不能是 顶级的(top-level objects)。
3.隐式类不能是case class（case class在后续介绍 样例类）
4.作用域内不能有与之相同名称的标识符

2. 代码样例

```
class MySQL1 {
  def sayOk(): Unit = {
    println("sayOk")
  }
}

def main(args: Array[String]): Unit = {
    //DB1会对应生成隐式类
    implicit class DB1(val m: MySQL1) {
      def addSuffix(): String = {
        m + " scala"
      }
    }
    val mysql1 = new MySQL1
    mysql1.sayOk()
}
```

7.5 隐式解析机制

即编译器是如何查找到缺失信息的，解析具有以下两种规则：
1.首先会在当前代码作用域下查找隐式实体（隐式方法、隐式类、隐式对象）。(一般是这种情况)
2.如果第一条规则查找隐式实体失败，会继续在隐式参数的类型的作用域里查找。类型的作用域是指与该类型相关联的全部伴生模块，一个隐式实体的类型T它的查找范围如下(第二种情况范围广且复杂在使用时，应当尽量避免出现)：
a) 如果T被定义为T with A with B with C,那么A,B,C都是T的部分，在T的隐式解析过程中，它们的伴生对象都会被搜索。
b) 如果T是参数化类型，那么类型参数和与类型参数相关联的部分都算作T的部分，比如List[String]的隐式搜索会搜索List的伴生对象和String的伴生对象。
c) 如果T是一个单例类型p.T，即T是属于某个p对象内，那么这个p对象也会被搜索。
d) 如果T是个类型注入S#T，那么S和T都会被搜索。

7.6 隐式转换的前提

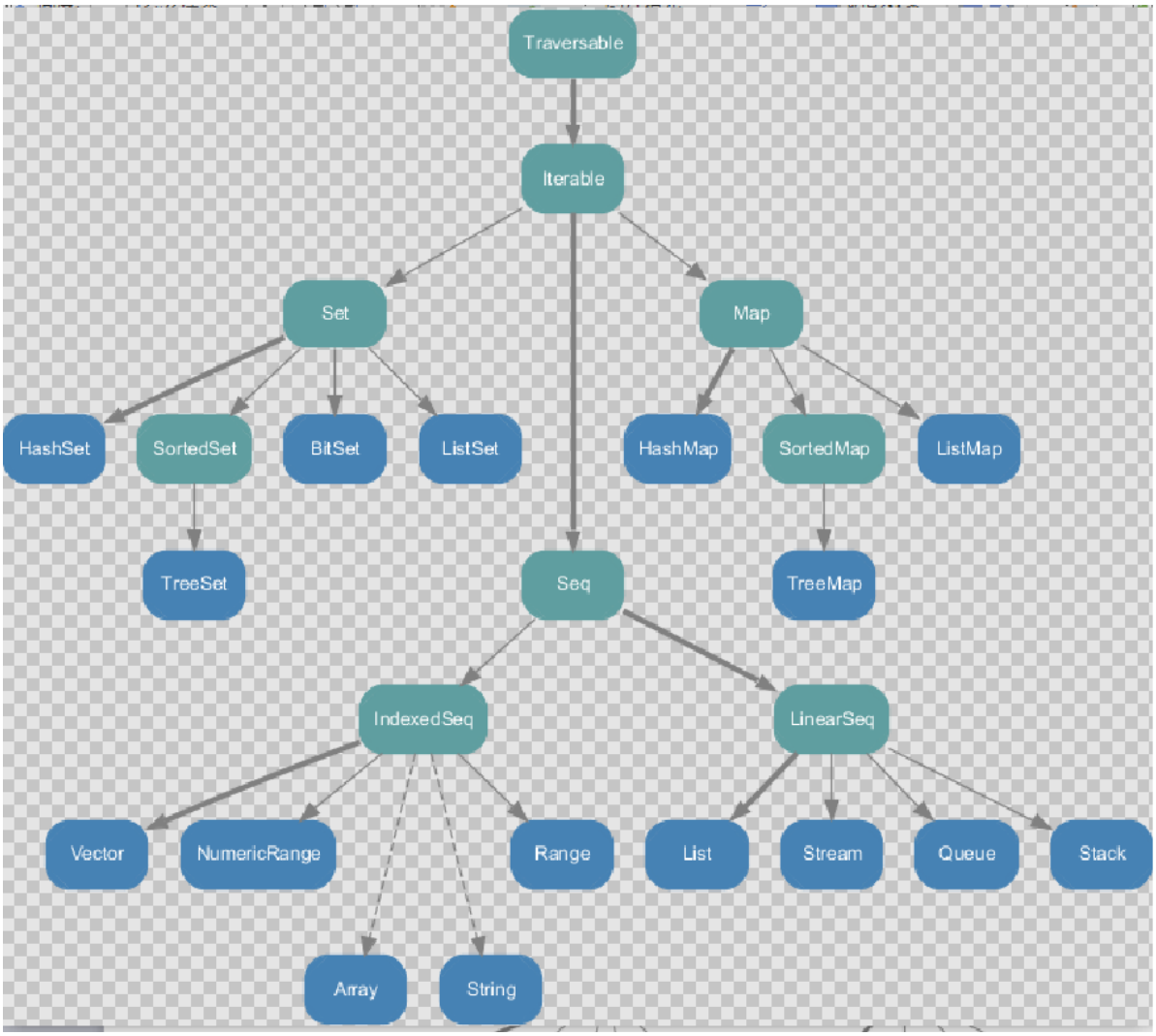
1.不能存在二义性
2.隐式操作不能嵌套使用

八、集合

8.1 基本介绍

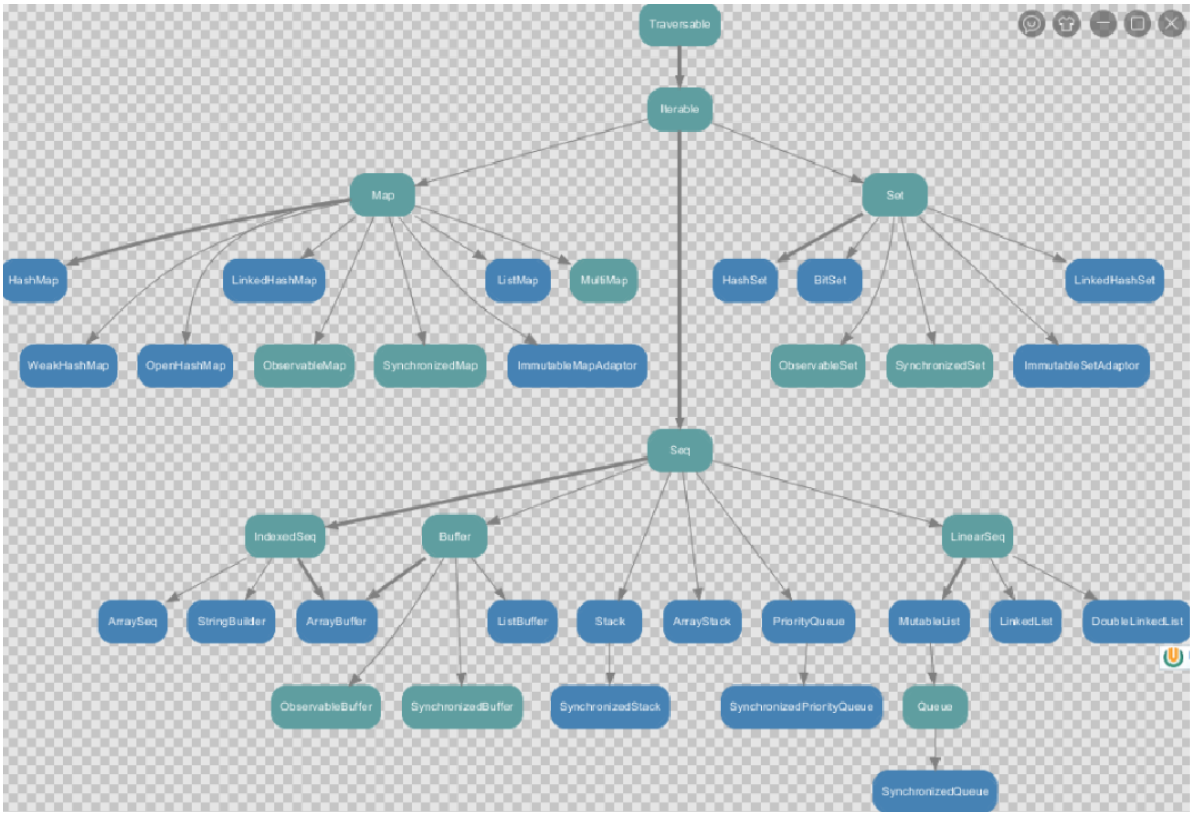
1.Scala同时支持不可变集合和可变集合，不可变集合可以安全的并发访问
2.两个主要的包：
不可变集合：scala.collection.immutable
可变集合： scala.collection.mutable
不可变集合：scala不可变集合，就是这个集合本身不能动态变化。(类似java的数组，是不可以动态增长的)
可变集合:可变集合，就是这个集合本身可以动态变化的。(比如:ArrayList ，是可以动态增长的)
3.Scala默认采用不可变集合，对于几乎所有的集合类，Scala都同时提供了可变(mutable)和不可变(immutable)的版本
4.Scala的集合有三大类：序列Seq、集Set、映射Map，所有的集合都扩展自Iterable特质，在Scala中集合有可变（mutable）和不可变（immutable）两种类型。

8.2 不可变集合继承层次一览图



1.Set、Map是Java中也有的集合
2.Seq是Java没有的，我们发现List归属到Seq了,因此这里的List就和java不是同一个概念了
3.我们前面的for循环有一个 1 to 3,就是IndexedSeq 下的Vector
4.String也是属于IndexeSeq
5.我们发现经典的数据结构比如 Queue 和 Stack被归属到LinearSeq
6.大家注意Scala中的Map体系有一个 SortedMap,说明Scala的Map可以支持排序
7.IndexSeq 和 LinearSeq 的区别
[IndexSeq是通过索引来查找和定位，因此速度快，比如String就是一个索引集合，通过索引即可定位]
[LineaSeq 是线型的，即有头尾的概念，这种数据结构一般是通过遍历来查找，它的价值在于应用到一些具体的应用场景（电商网站，大数据推荐系统 :最近浏览的10个商品)

8.3 可变集合继承层次一览图



8.4 数组

1.定长数组[Array]

1. 创建方式一

```
val arr01 = new Array[Int](4)
println(arr01.length)

println("arr01(0)=" + arr01(0))
for (i <- arr01) {
    println(i)
}
println("--------------------")
arr01(3) = 10
for (i <- arr01) {
    println(i)
}
```

2. 创建方式二

```
var arr02 = Array(1, 3, "xxx")
for (i <- arr02) {
    println(i)
}
```

2. 变长数组[ArrayBuffer]

1. 声明与一些操作

```
val arr01 = ArrayBuffer[Any](3, 2, 5)

println("arr01(1)=" + arr01(1))
for (i <- arr01) {
    println(i)
}
println(arr01.length) //?
println("arr01.hash=" + arr01.hashCode())
arr01.append(90.0,13)
println("arr01.hash=" + arr01.hashCode())

arr01(1) = 89 //修改
println("---------------------------")
for (i <- arr01) {
    println(i)
}

//删除
arr01.remove(0)
println("---------------------------")
for (i <- arr01) {
    println(i)
}
println("最新的长度=" + arr01.length)
```

2. ArrayBuffer小结

```
1.ArrayBuffer是变长数组，类似java的ArrayList
2.val arr2 = ArrayBuffer[Int]() 也是使用的apply方法构建对象
3.def append(elems: A*) { appendAll(elems) } 接收的是可变参数.
4.每append一次，arr在底层会重新分配空间，进行扩容，arr2的内存地址会发生变化，也就成为新的ArrayBuffer
```

3. 定长数组与变长数据的转换

```
val arr2 = ArrayBuffer[Int]()
// 追加值
arr2.append(1, 2, 3)
println(arr2)

val newArr = arr2.toArray //可变数组转定长数组
println(newArr)

val newArr2 = newArr.toBuffer //定长数组转可变数组
newArr2.append(123)
println(newArr2)

说明：
arr2.toArray 返回结果才是一个定长数组， arr2本身没有变化
arr1.toBuffer返回结果才是一个可变数组， arr1本身没有变化
```

4. 多维数组

1. 定义

```
val arr = Array.ofDim[Double](3,4)
//说明：二维数组中有三个一维数组，每个一维数组中有四个元素
```

2. 一些操作

```
val array1 = Array.ofDim[Int](3, 4)
array1(1)(1) = 9
for (item <- array1) {
    for (item2 <- item) {
```

```
        print(item2 + "\t")
    }
    println()
}
println("====================")
for (i <- 0 to array1.length - 1) {
for (j <- 0 to array1(i).length - 1) {
    printf("arr[%d][%d]=%d\t", i, j, array1(i)(j))
}
    println()
}
```

5. Scala数组与Java数组转换

```
// Scala集合和Java集合互相转换
val arr = ArrayBuffer("1", "2", "3")
import scala.collection.JavaConversions.bufferAsJavaList
val javaArr = new ProcessBuilder(arr)
val arrList = javaArr.command()
println(arrList) //输出 [1, 2, 3]

//对下面代码说明:
//1. scala.collection.JavaConversions.bufferAsJavaList的
bufferAsJavaList是一个隐式转换函数 将 Buffer => java.util.List
    //implicit def bufferAsJavaList[A](b :
scala.collection.mutable.Buffer[A]) : java.util.List[A] =
{ /* compiled code */ }

//2. ProcessBuilder(arr) 是一个java的构造函数
/*
      public ProcessBuilder(List<String> command) {
        if (command == null)
            throw new NullPointerException();
        this.command = command;
    }
*/
//3.  val javaArr = new ProcessBuilder(arr) 返回的是
ProcessBuilder 对象 即 javaArr是ProcessBuilder
//4.  javaArr.command() 返回的是 java.util.List
```

6. Java的List与Scala数组转换

```
import scala.collection.JavaConversions.asScalaBuffer
import scala.collection.mutable
// java.util.List ==> Buffer
val scalaArr: mutable.Buffer[String] = arrList
scalaArr.append("jack")
println(scalaArr)
```

8.5 元组

1. 基本介绍

元组也是可以理解为一个容器，可以存放各种相同或不同类型的数据。
说的简单点，就是将多个无关的数据封装为一个整体，称为元组， 最多的特点灵活,对数据没有过多的约束。

注意：元组中最大只能有22个元素

2. 创建

```
val tuple1 = (1, 2, 3, "hello", 4)
println(tuple1)

1.t1 的类型是 Tuple5类 是scala特有的类型
2.t1 的类型取决于 t1 后面有多少个元素， 有对应关系，比如 4个元素=》Tuple4
3.看一个Tuple5 类的定义,大家就了然了
    /*
    final case class Tuple5[+T1, +T2, +T3, +T4, +T5](_1: T1, _2: T2, _3: T3, _4: T4, _5: T5)
  extends Product5[T1, T2, T3, T4, T5]
  {
  override def toString() = "(" + _1 + "," + _2 + "," + _3 + "," + _4 + "," + _5 + ")"
   }
     */
4.元组中最大只能有22个元素 即 Tuple1...Tuple22
```

3. 遍历

访问元组中的数据,可以采用顺序号（_顺序号），也可以通过索引（productElement）访问。

```
val t1 = (1, "a", "b", true, 2)
println(t1._1) //访问元组的第一个元素 ，从1开始
println(t1.productElement(0)) // 访问元组的第一个元素，从0开始

for(item <- t1.productIterator){
    println(item)
}
```

8.6 列表

1.不可变列表[List]

1. 基本介绍

Scala中的List 和Java List 不一样，在Java中List是一个接口，真正存放数据是ArrayList，而Scala的List可以直接存放数据，就是一个object，默认情况下Scala的List是不可变的，List属于序列Seq。

```
val List = scala.collection.immutable.List
object List extends SeqFactory[List]
```

2. 创建

```
val list01 = List(1, 2, 3) //创建时，直接分配元素
println(list01)
val list02 = Nil  //空集合
println(list02)

1.List默认为不可变的集合
2.List 在 scala包对象声明的,因此不需要引入其它包也可以使用
3.val List = scala.collection.immutable.List
4.List 中可以放任何数据类型，比如 arr1的类型为 List[Any]
5.如果希望得到一个空列表，可以使用Nil对象， 在 scala包对象声明的,因此不需要引入其它包也可以使用
val Nil = scala.collection.immutable.Nil
```

3. 查找

```
val value1 = list1(1) // 1是索引，表示取出第2个元素.
println(value1)
```

4. 元素的添加

向列表中增加元素，会返回新的列表/集合对象。注意：Scala中List元素的追加形式非常独特，和Java不一样。

1. 在列表的最后增加数据

```
var list1 = List(1, 2, 3, "abc")
// :+运算符表示在列表的最后增加数据
val list2 = list1 :+ 4
println(list1) //list1没有变化
println(list2) //新的列表结果是 [1, 2, 3, "abc", 4]
```

2. 在列表的最前面增加数据

```
var list1 = List(1, 2, 3, "abc")
// :+运算符表示在列表的最后增加数据
val list2 = 4 +: list1
println(list1) //list1没有变化
println(list2)
```

3. 在列表的最后增加数据

```
1.符号::表示向集合中   新建集合添加元素。
2.运算时，集合对象一定要放置在最右边，
3.运算规则，从右向左。
4.::: 运算符是将集合中的每一个元素加入到空集合中去

val list1 = List(1, 2, 3, "abc")
val list5 = 4 :: 5 :: 6 :: list1 :: Nil
println(list5)
//下面等价 4 :: 5 :: 6 :: list1
val list7 = 4 :: 5 :: 6 :: list1 ::: Nil
println(list7)
```

2. 可变列表[ListBuffer]

1. 基本介绍

ListBuffer是可变的list集合，可以添加，删除元素,ListBuffer属于序列

2. 基本操作

```
val lst0 = ListBuffer[Int](1, 2, 3)

//如何访问
println("lst0(2)=" + lst0(2))
for (item <- lst0) {
    println("item=" + item)
}

//动态的增加元素，lst1就会变化，增加一个一个的元素
val lst1 = new ListBuffer[Int] //空的ListBuffer
lst1 += 4 // list1 (4)
lst1.append(5) //lst1 (4,5)


lst0 ++= lst1
println("lst0=" + lst0) // lst0=ListBuffer(1, 2, 3, 4, 5)

val lst2 = lst0 ++ lst1
println("lst2=" + lst2) // lst2=ListBuffer(1, 2, 3, 4, 5,
4, 5)

val lst3 = lst0 :+ 5
println("lst3=" + lst3) // lst3=ListBuffer(1, 2, 3, 4, 5,
5)

println("=====删除=======")
println("lst1=" + lst1)
lst1.remove(1)
for (item <- lst1) {
    println("item=" + item)
}
```

8.7 队列

1. 说明

1.队列是一个有序列表，在底层可以用数组或是链表来实现。
2.其输入和输出要遵循先入先出的原则。即：先存入队列的数据，要先取出。后存入的要后取出
3.在Scala中，由设计者直接给我们提供队列类型使用。
4.在scala中，有 scala.collection.mutable.Queue 和 scala.collection.immutable.Queue ，一般来说，我们在开发中通常使用可变集合中的队列。

2. 创建

```
import scala.collection.mutable
//说明: 这里的Int是泛型，表示q1队列只能存放Int类型
//如果希望q1可以存放其它类型，则使用 Any 即可。
val q1 = new mutable.Queue[Int]
println(q1)
```

3. 队列中元素的添加

```
val q1 = new Queue[Int]
q1 += 20 // 底层?
println(q1)

q1 ++= List(2,4,6) //
println(q1)

//q1 += List(1,2,3) //泛型为Any才ok
println(q1)
```

4. 入队和出队

```
val q1 = new mutable.Queue[Int]//
q1 += 12
q1 += 34
q1 ++= List(2,9)

q1.dequeue() //出队，队列头
println(q1)

q1.enqueue(20,60) //入队，队列尾
println(q1)
```

5. 其他操作

返回队列的第一个元素

```
println(q1.head)
```

返回队列最后一个元素

```
println(q1.last)
```

返回队列的尾部

即：返回除了第一个以外剩余的元素， 可以级联使用，这个在递归时使用较多。

```
println(q1.tail)
println(q1.tail.tail)
```

8.8 映射

1. 基本介绍

1.Scala中的Map 和Java类似，也是一个散列表，它存储的内容也是键值对(key-value)映射，Scala中不可变的Map是有序的，可变的Map是无序的。
2.Scala中，有可变Map (scala.collection.mutable.Map) 和 不可变Map(scala.collection.immutable.Map)

2. 创建映射

1. 构造不可变映射

Scala中的不可变Map是有序，构建Map中的元素底层是Tuple2类型。

```
val map1 = Map("Alice" -> 10, "Bob" -> 20, "Kotlin" -> "北京")
```

1.从输出的结果看到，输出顺序和声明顺序一致
2.构建Map集合中，集合中的元素其实是Tuple2类型
3.默认情况下（即没有引入其它包的情况下）,Map是不可变map
4.为什么说Map中的元素是Tuple2 类型

2. 构造可变映射

```
//需要指定可变Map的包
val map2 = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 20, "Kotlin" -> 30)
```

说明
1.从输出的结果看到，输出顺序和声明顺序不一致

3. 创建空的映射

```
val map3 = new scala.collection.mutable.HashMap[String, Int]
println(map3)
```

4. 对偶元组

即创建包含键值对的二元组， 和第一种方式等价，只是形式上不同而已。

对偶元组 就是只含有两个数据的元组。

```
val map4 = mutable.Map( ("A", 1), ("B", 2), ("C", 3),("D", 30) )
println("map4=" + map4)
println(map4("A"))
```

3. 取值

1. 使用map(key)

```
val value1 = map2("Alice")
println(value1)

1.如果key存在，则返回对应的值
2.如果key不存在，则抛出异常[java.util.NoSuchElementException]
3.在Java中,如果key不存在则返回null
```

2. 使用contains

```
// 返回Boolean
// 1.如果key存在，则返回true
// 2.如果key不存在，则返回false
map4.contains("B")

说明:
使用containts先判断在取值，可以防止异常，并加入相应的处理逻辑

val map4 = mutable.Map( ("A", 1), ("B", 2), ("C", 3),("D",
30.9) )
if( map4.contains("B") ) {
    println("key存在 值= " + map4("B"))
} else {
    println("key不存在")
}
```

3. 使用map.get(key).get()

通过映射.get(键) 这样的调用返回一个Option对象，要么是Some，要么是None

```
var map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )
println(map4.get("A")) //Some
println(map4.get("A").get) //得到Some在取出
```

小结：
1.map.get方法会将数据进行包装
2.如果 map.get(key) key存在返回some,如果key不存在，则返回None
3.如果 map.get(key).get key存在，返回key对应的值,否则，抛出异常
java.util.NoSuchElementException: None.get

4. 使用map.getOrElse()

getOrElse 方法 ： def getOrElse[V1 >: V](key: K, default: => V1)

说明：
1.如果key存在，返回key对应的值。
2.如果key不存在，返回默认值。在java中底层有很多类似的操作。

```
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )
println(map4.getOrElse("A","默认"))
```

5. 选择取值方式的建议

1.如果我们确定map有这个key ,则应当使用map(key), 速度快
2.如果我们不能确定map是否有key ,而且有不同的业务逻辑，使用map.contains() 先判断在加入逻辑
3.如果只是简单的希望得到一个值，使用map4.getOrElse("ip","127.0.0.1")

4. 修改

```
更新map的元素
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )
map4("AA") = 20
println(map4)

说明:
1.map是可变的，才能修改，否则报错
2.如果key存在：则修改对应的值,key不存在,等价于添加一个key-val
```

5. 添加

```
方式1-增加单个元素
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )
map4 += ( "D" -> 4 )
map4 += ( "B" -> 50 )
println(map4)

方式2-增加多个元素
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )
val map5 = map4 + ("E"->1, "F"->3)
map4 += ("EE"->1, "FF"->3)
```

6. 删除

```
val map4 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )
map4 -= ("A", "B")
println("map4=" + map4)
map4.remove("EE")
println("map4=" + map4)
```

7. 遍历

```
val map1 = mutable.Map( ("A", 1), ("B", "北京"), ("C", 3) )
for ((k, v) <- map1) println(k + " is mapped to " + v)
for (v <- map1.keys) println(v)
for (v <- map1.values) println(v)
for(v <- map1) println(v) //v是Tuple?
```

说明
1.每遍历一次，返回的元素是Tuple2
2.取出的时候，可以按照元组的方式来取

8.9 映射

1. 基本介绍

集是不重复元素的结合。集不保留顺序，默认是以哈希集实现
默认情况下，Scala 使用的是不可变集合，如果想使用可变集合，需要引用scala.collection.mutable.Set包

2. 创建

```
val set = Set(1, 2, 3) //不可变
println(set)

import scala.collection.mutable.Set
val mutableSet = Set(1, 2, 3) //可变
```

3. 添加

可变集合的元素添加

```
mutableSet.add(4) //方式1
mutableSet += 6  //方式2
mutableSet.+=(5) //方式3

说明：如果添加的对象已经存在，则不会重复添加，也不会报错
val set02 = mutable.Set(1,2,4,"abc")
set02.add(90)
set02 += 78
set02 += 90
println(set02)
```

4. 删除

可变集合的元素删除

```
val set02 = mutable.Set(1,2,4,"abc")
set02 -= 2 // 操作符形式
set02.remove("abc") // 方法的形式，scala的Set可以直接删除值
println(set02)
说明：说明：如果删除的对象不存在，则不生效，也不会报错
```

5. 遍历

```
val set02 = mutable.Set(1, 2, 4, "abc")
for(x <- set02) {
    println(x)
}
```

6. 更多操作

序号	方法	描述
1	def +(elem: A): Set[A]	为集合添加新元素，并创建一个新的集合，除非元素已存在
2	def -(elem: A): Set[A]	移除集合中的元素，并创建一个新的集合
3	def contains(elem: A): Boolean	如果元素在集合中存在，返回 true，否则返回 false。
4	def &(that: Set[A]): Set[A]	返回两个集合的交集
5	def &~(that: Set[A]): Set[A]	返回两个集合的差集
6	def ++(elems: A): Set[A]	合并两个集合
7	def drop(n: Int): Set[A]]	返回丢弃前n个元素新集合
8	def dropRight(n: Int): Set[A]	返回丢弃最后n个元素新集合
9	def dropWhile(p: (A) => Boolean): Set[A]	从左向右丢弃元素，直到条件p不成立
10	def max: A //演示下	查找最大元素
11	def min: A //演示下	查找最小元素
12	def take(n: Int): Set[A]	返回前 n 个元素

九、集合操作

9.1 map

```
val list1 = List(3, 5, 7)
def f1(n1: Int): Int = {
    2 * n1
}
val list2 = list1.map(f1)
```

```
println(list2)

深刻理解map映射函数的机制-模拟实现
def main(args: Array[String]): Unit = {
    val list1 = List(3, 5, 7)
    def f1(n1: Int): Int = {
        println("xxx")
        2 * n1
    }
    val list2 = list1.map(f1)
    println(list2)

    val myList = MyList()
    val myList2 = myList.map(f1)
    println("myList2=" + myList2)
    println("myList=" + myList.list1)
}

class MyList {
  var list1 = List(3, 5, 7)
  var list2 = List[Int]()
  def map(f:Int=>Int): List[Int] = {
    for (item<-list1) {
      list2 = list2 :+ f(item)
    }
    list2
  }
}
object MyList {
  def apply(): MyList = new MyList()
}
```

9.2 flatmap

flatmap：flat即压扁，压平，扁平化，效果就是将集合中的每个元素的子元素映射到某个函数并返回新的集合

```
val names = List("Alice", "Bob", "Nick")
def upper( s : String ) : String = {
    s. toUpperCase
}
//注意：每个字符串也是char集合
println(names.flatMap(upper))
```

9.3 filter

filter：将符合要求的数据(筛选)放置到新的集合中

```
//将val names = List("Alice", "Bob", "Nick") 集合中首字母为'A'的筛选到新的集合。
val names = List("Alice", "Bob", "Nick")
def startA(s:String): Boolean = {
s.startsWith("A")
}
val names2 = names.filter(startA)
println("names=" + names2)
```

9.4 reduce

```
val list = List(1, 20, 30, 4, 5)
def sum(n1: Int, n2: Int): Int = {
n1 + n2
}
val res = list.reduceLeft(sum)
println("res=" + res)

//说明
1.def reduceLeft[B >: A](@deprecatedName('f) op: (B, A) =>
B): B
2.reduceLeft(f) 接收的函数需要的形式为 op: (B, A) => B): B
3.reduceleft(f) 的运行规则是 从左边开始执行将得到的结果返回给第一个
参数
4.然后继续和下一个元素运行，将得到的结果继续返回给第一个参数，继续..
5.即: //((((1 + 2)  + 3) + 4) + 5) = 15
```

9.5 fold

```
fold函数将上一步返回的值作为函数的第一个参数继续传递参与运算，直到
list中的所有元素被遍历

1.可以把reduceLeft看做简化版的foldLeft。
如何理解:
def reduceLeft[B >: A](@deprecatedName('f) op: (B, A) =>
B): B = if (isEmpty) throw new
UnsupportedOperationException("empty.reduceLeft") else
tail.foldLeft[B](head)(op)
大家可以看到. reduceLeft就是调用的foldLeft[B](head)，并且是默认
从集合的head元素开始操作的
2.相关函数：fold，foldLeft，foldRight，可以参考reduce的相关方法
理解

// 折叠
val list = List(1, 2, 3, 4)
def minus( num1 : Int, num2 : Int ): Int = {
    num1 - num2
}
println(list.foldLeft(5)(minus)) // 函数的柯里化
println(list.foldRight(5)(minus)) //
```

```
foldLeft和foldRight 缩写方法分别是：/:和:\
val list4 = List(1, 9, 2, 8)
def minus(num1: Int, num2: Int): Int = {
num1 - num2
}
var i6 = (1 /: list4) (minus) // =等价=> list4.foldLeft(1)
(minus)
println(i6)
i6 = (100 /: list4) (minus)
println(i6)
i6 = (list4 :\ 10) (minus) // list4.foldRight(10)(minus)
println(i6)
```

9.6 scan

扫描，即对某个集合的所有元素做fold操作，但是会把产生的所有中间结果放置于一个集合中保存

```
def minus( num1 : Int, num2 : Int ) : Int = {
    num1 - num2
}
//5 (1,2,3,4,5) =>(5,4,2,-1,-5,-10)
val i8 = (1 to 5).scanLeft(5)(minus) //IndexedSeq[Int]
println(i8)
def add( num1 : Int, num2 : Int ) : Int = {
    num1 + num2
}
//5 (1,2,3,4,5) =>(5,6,8, 11,15,20)
val i9 = (1 to 5).scanLeft(5)(add) //IndexedSeq[Int]
println(i9)
```

9.7 zip

在开发中，当我们需要将两个集合进行 对偶元组合并，可以使用拉链。

```
// 拉链
val list1 = List(1, 2 ,3)
val list2 = List(4, 5, 6)
val list3 = list1.zip(list2) // (1,4),(2,5),(3,6)
println("list3=" + list3)
```

1.拉链的本质就是两个集合的合并操作，合并后每个元素是一个 对偶元组。
2.如果两个集合个数不对应，会造成数据丢失。
3.集合不限于List, 也可以是其它集合比如 Array
4.如果要取出合并后的各个对偶元组的数据，可以遍历

9.8 iterator

通过iterator方法从集合获得一个迭代器，通过while循环和for表达式对集合进行遍历

```
val iterator = List(1, 2, 3, 4, 5).iterator // 得到迭代器
println("--------遍历方式1 -----------------")
while (iterator.hasNext) {
    println(iterator.next())
}
println("--------遍历方式2 for -----------------")
for(enum <- iterator) {
    println(enum) //
}
```

1.iterator 的构建实际是 AbstractIterator 的一个匿名子类，该子类提供了

```
/*
     def iterator: Iterator[A] = new AbstractIterator[A] {
    var these = self
    def hasNext: Boolean = !these.isEmpty
    def next(): A =
*/
```

2.该AbstractIterator 子类提供了 hasNext next 等方法.
3.因此，我们可以使用 while的方式，使用hasNext next 方法变量

9.9 Stream

stream是一个集合。这个集合，可以用于存放无穷多个元素，但是这无穷个元素并不会一次性生产出来，而是需要用到多大的区间，就会动态的生产，末尾元素遵循lazy规则(即：要使用结果才进行计算的) 。

```
def numsForm(n: BigInt) : Stream[BigInt] = n #::
numsForm(n + 1)
val stream1 = numsForm(1)
```

说明
1.Stream 集合存放的数据类型是BigInt
2.numsForm 是自定义的一个函数，函数名是程序员指定的。
3.创建的集合的第一个元素是 n ，后续元素生成的规则是 n + 1
4.后续元素生成的规则是可以程序员指定的 ，比如 numsForm(n * 4)...

使用tail，会动态的向stream集合按规则生成新的元素

```
//创建Stream
def numsForm(n: BigInt) : Stream[BigInt] = n #::
numsForm(n + 1)
val stream1 = numsForm(1)
println(stream1) //
//取出第一个元素
println("head=" + stream1.head) //
println(stream1.tail) //
println(stream1)
```

9.10 View

Stream的懒加载特性，也可以对其他集合应用view方法来得到类似的效果，具有如下特点：
1.view方法产出一个总是被懒执行的集合。
2.view不会缓存数据，每次都要重新计算，比如遍历View时。

```
def multiple(num: Int): Int = {
    num
}
def eq(i: Int): Boolean = {
    i.toString.equals(i.toString.reverse)
}

//说明: 没有使用view
```

```
val viewSquares1 = (1 to 100).map(multiple).filter(eq)
println(viewSquares1)
//for (x <- viewSquares1) {}
//使用view
val viewSquares2 = (1 to
100).view.map(multiple).filter(eq)
println(viewSquares2)
```

9.11 parallel

```
1.打印1~5
(1 to 5).foreach(println(_))
println()
(1 to 5).par.foreach(println(_))

2.查看并行集合中元素访问的线程
val result1 = (0 to 100).map{case _ =>
Thread.currentThread.getName}
val result2 = (0 to 100).par.map{case _ =>
Thread.currentThread.getName}
println(result1)
println(result2)
```

十、模式匹配

10.1 基本介绍

模式匹配语法中，采用match关键字声明，每个分支采用case关键字进行声明，当需要匹配时，会从第一个case分支开始，如果匹配成功，那么执行对应的逻辑代码，如果匹配不成功，继续执行下一个分支进行判断。如果所有case都不匹配，那么会执行case _ 分支，类似于Java中default语句。

10.2 使用细节和注意事项

1.如果所有case都不匹配，那么会执行case _ 分支，类似于Java中default语句
2.如果所有case都不匹配，又没有写case _ 分支，那么会抛出MatchError
3.每个case中，不用break语句，自动中断case
4.可以在match中使用其它类型，而不仅仅是字符
5.等价于 java swtich 的 :
6.后面的代码块到下一个 case， 是作为一个整体执行，可以使用{} 扩起来，也可以不扩。

10.3 守卫

如果想要表达匹配某个范围的数据，就需要在模式匹配中增加条件守卫

```
for (ch <- "+-3!") {
var sign = 0
var digit = 0
ch match {
    case '+' => sign = 1
    case '-' => sign = -1
    // 说明..
    case _ if ch.toString.equals("3") => digit = 3
    case _ => sign = 2
}
println(ch + " " + sign + " " + digit)
}
```

10.4 模式中的变量

如果在case关键字后跟变量名，那么match前表达式的值会赋给那个变量

```
val ch = 'V'
ch match {
    case '+' => println("ok~")
    case mychar => println("ok~" + mychar)
    case _ => println ("ok~~")
}
```

10.5 类型匹配

可以匹配对象的任意类型，这样做避免了使用isInstanceOf和asInstanceOf方法

```
val result = obj match {
    case a : Int => a
    case b : Map[String, Int] => "对象是一个字符串-数字的Map集合"
    case c : Map[Int, String] => "对象是一个数字-字符串的Map集合"
    case d : Array[String] => "对象是一个字符串数组"
    case e : Array[Int] => "对象是一个数字数组"
    case f : BigInt => Int.MaxValue
    case _ => "啥也不是"
}
println(result)
```

注意事项:

1.Map[String, Int] 和Map[Int, String]是两种不同的类型，其它类推

2.在进行类型匹配时，编译器会预先检测是否有可能的匹配，如果没有则报错

3.一个说明:

```
val result = obj match {
    case i : Int => i
}
```

case i : Int => i 表示 将 i = obj (其它类推)，然后再判断类型

4.如果 case _ 出现在match 中间，则表示隐藏变量名，即不使用,而不是表示默认匹配

```
val result = obj match {
    case a : Int => a
    case _ : BigInt => Int.MaxValue //看这里!
    case b : Map[String, Int] => "对象是一个字符串-数字的Map集合"
    case c : Map[Int, String] => "对象是一个数字-字符串的Map集合"
    case d : Array[String] => "对象是一个字符串数组"
    case e : Array[Int] => "对象是一个数字数组"
    case _ => "啥也不是"
}
println(result)
```

10.6 匹配数组

基本介绍:
1.Array(0) 匹配只有一个元素且为0的数组。
2.Array(x,y) 匹配数组有两个元素，并将两个元素赋值为x和y。当然可以依次类推Array(x,y,z) 匹配数组有3个元素的等等....
3.Array(0,_*) 匹配数组以0开始

```
for (arr <- Array(Array(0), Array(1, 0), Array(0, 1, 0),
Array(1, 1, 0), Array(1, 1, 0, 1))) {
val result = arr match {
    case Array(0) => "0"
    case Array(x, y) => x + "=" + y
    case Array(0, _*) => "以0开头和数组"
    case _ => "什么集合都不是"
}
    println("result = " + result)
}
```

10.7 匹配列表

```
for (list <- Array(List(0), List(1, 0), List(0, 0, 0),
List(1, 0, 0))) {
val result = list match {
    case 0 :: Nil => "0"
    case x :: y :: Nil => x + " " + y
    case 0 :: tail => "0 ..."
    case _ => "something else"
}
    println(result)
}
```

10.8 匹配元组

```
for (pair <- Array((0, 1), (1, 0), (1, 1),(1,0,2))) {
val result = pair match {
    case (0, _) => "0 ..."
    case (y, 0) => y
    case _ => "other"
}
    println(result)
}
```

10.9 对象匹配

基本介绍:
1.case中对象的unapply方法(对象提取器)返回Some集合则为匹配成功
2.返回none集合则为匹配失败

```
object Square {
    def unapply(z: Double): Option[Double] =
Some(math.sqrt(z))
    def apply(z: Double): Double = z * z
}
// 模式匹配使用:
val number: Double = 36.0
number match {
    case Square(n) => println(n)
    case _ => println("nothing matched")
}
```

小结:
1.构建对象时apply会被调用 ，比如 val n1 = Square(5)
2.当将 Square(n) 写在 case 后时[case Square(n) => xxx]，会默认调用unapply 方法(对象提取器)
3.number会被传递给def unapply(z: Double)的z形参
4.如果返回的是Some集合，则unapply提取器返回的结果会返回给 n 这个形参
5.case中对象的unapply方法(提取器)返回some集合则为匹配成功
6.返回none集合则为匹配失败
7.当case 后面的对象提取器方法的参数为多个，则会默认调用def unapplySeq() 方法
8.如果unapplySeq返回是Some，获取其中的值,判断得到的sequence中的元素的个数是否是三个,如果是三个，则把三个元素分别取出，赋值给first，second和third
9.其它的规则不变.

```
object Names {
def unapplySeq(str: String): Option[Seq[String]] = {
    if (str.contains(",")) Some(str.split(","))
    else None
    }
}
```

```
val namesString = "Alice,Bob,Thomas"
//说明
namesString match {
    case Names(first, second, third) => {
    println("the string contains three people's names")
    // 打印字符串
    println(s"$first $second $third")
    }
    case _ => println("nothing matched")
}
```

10.10 变量声明中的模式

基本介绍:match中每一个case都可以单独提取出来，意思是一样的

```
val (x, y) = (1, 2)
val (q, r) = BigInt(10) /% 3  //说明  q = BigInt(10) / 3 r
= BigInt(10) % 3
val arr = Array(1, 7, 2, 9)
val Array(first, second, _*) = arr // 提出arr的前两个元素
println(first, second)
```

10.11 for表达式中的模式

基本介绍:for循环也可以进行模式匹配.

```
val map = Map("A"->1, "B"->0, "C"->3)
for ( (k, v) <- map ) {
    println(k + " -> " + v)
}
//说明
for ((k, 0) <- map) {
    println(k + " --> " + 0)
}
//说明
for ((k, v) <- map if v == 0) {
    println(k + " ---> " + v)
}
```

十一、样例类

11.1 基本介绍

1.样例类仍然是类
2.样例类用case关键字进行声明。
3.样例类是为模式匹配而优化的类
4.构造器中的每一个参数都成为val——除非它被显式地声明为var（不建议这样做）
5.在样例类对应的伴生对象中提供apply方法让你不用new关键字就能构造出相应的对象
6.提供unapply方法让模式匹配可以工作
7.将自动生成toString、equals、hashCode和copy方法(有点类似模板类，直接给生成，供程序员使用)
8.除上述外，样例类和其他类完全一样。你可以添加方法和字段，扩展它们

11.2 代码样例

```
abstract class Amount
case class Dollar(value: Double) extends Amount
case class Currency(value: Double, unit: String) extends
Amount
case object NoAmount extends Amount
```

11.3 实践

1.当我们有一个类型为Amount的对象时，可以用模式匹配来匹配他的类型，并将属性值绑定到变量(即：把样例类对象的属性值提取到某个变量,该功能有用)

```
for (amt <- Array(Dollar(1000.0), Currency(1000.0, "RMB"),
NoAmount)) {
val result = amt match {
    case Dollar(v) => "$" + v
    case Currency(v, u) => v + " " + u
    case NoAmount => ""
}
    println(amt + ": " + result)
}
```

2.样例类的copy方法和带名参数,copy创建一个与现有对象值相同的新对象,并可以通过带名参数来修改某些属性

```
val amt = Currency(29.95, "RMB")
val amt1 = amt.copy() //创建了一个新的对象，但是属性值一样
val amt2 = amt.copy(value = 19.95) //创建了一个新对象，但是修改了货币单位
val amt3 = amt.copy(unit = "英镑")//..
println(amt)
println(amt2)
println(amt3)
```

11.4 case语句中的中缀表达式

```
List(1, 3, 5, 9) match { //修改并测试
    //1.两个元素间::叫中置表达式,至少first，second两个匹配才行.
    //2.first 匹配第一个 second 匹配第二个, rest 匹配剩余部分(5,9)
    case first :: second :: rest => println(first + second + rest.length)
    case _ => println("匹配不到...")
}
```

11.5 匹配嵌套结构

基本介绍:操作原理类似于正则表达式

```
abstract class Item // 项

case class Book(description: String, price: Double) extends Item
//Bundle 捆 ， discount: Double 折扣 ， item: Item* ，
case class Bundle(description: String, discount: Double, item: Item*) extends Item

//给出案例表示有一捆数，单本漫画（40-10） +文学作品(两本书)（80+30-20） = 30 + 90 = 120.0
val sale = Bundle("书籍", 10,  Book("漫画", 40), Bundle("文学作品", 20, Book("《阳关》", 80), Book("《围城》", 30)))
```

知识点1-将descr绑定到第一个Book的描述

```
知识点2-通过@表示法将嵌套的值绑定到变量。_*绑定剩余Item到rest
知识点3-不使用_*绑定剩余Item到rest
def price(it: Item): Double = {
  it match {
    case Book(_, p) => p
    //生成一个新的集合,_是将its中每个循环的元素传递到price中it中。递归操作,分析一个简单的流程
    case Bundle(_, disc, its @ _*) => its.map(price _).sum - disc
  }
}
```

11.6 密封类

基本介绍

1.如果想让case类的所有子类都必须在申明该类的相同的源文件中定义，可以将样例类的通用超类声明为sealed，这个超类称之为密封类。

2.密封就是不能在其他文件中定义子类。

十二、函数式编程高级

12.1 偏函数

1. 基本介绍

1.在对符合某个条件，而不是所有情况进行逻辑操作时，使用偏函数是一个不错的选择

2.将包在大括号内的一组case语句封装为函数，我们称之为偏函数，它只对会作用于指定类型的参数或指定范围值的参数实施计算，超出范围的值会忽略（未必会忽略，这取决于你打算怎样处理）

3.偏函数在Scala中是一个特质PartialFunction

2. 基本代码

```
val list = List(1, 2, 3, 4, "abc")
//说明
val addOne3= new PartialFunction[Any, Int] {
    def isDefinedAt(any: Any) = if (any.isInstanceOf[Int])
true else false
    def apply(any: Any) = any.asInstanceOf[Int] + 1
}
val list3 = list.collect(addOne3)
println("list3=" + list3)
```

3. 小结

1.使用构建特质的实现类(使用的方式是PartialFunction的匿名子类)
2.PartialFunction 是个特质(看源码)
3.构建偏函数时，参数形式 [Any, Int]是泛型，第一个表示参数类型，第二个表示返回参数
4.当使用偏函数时，会遍历集合的所有元素，编译器执行流程时先执行isDefinedAt()如果为true ,就会执行 apply, 构建一个新的Int对象返回
5.执行isDefinedAt() 为false 就过滤掉这个元素，即不构建新的Int对象.
6.map函数不支持偏函数，因为map底层的机制就是所有循环遍历，无法过滤处理原来集合的元素
7.collect函数支持偏函数

4. 简写形式

声明偏函数，需要重写特质中的方法，有的时候会略显麻烦，而Scala其实提供了简单的方法

简化形式1:

```
def f2: PartialFunction[Any, Int] = {
  case i: Int => i + 1 // case语句可以自动转换为偏函数
}
val list2 = List(1, 2, 3, 4,"ABC").collect(f2)
```

简化形式2:

```
val list3 = List(1, 2, 3, 4,"ABC").collect{ case i: Int =>
i + 1 }
println(list3)
```

12.2 作为函数的参数

1. 基本介绍

函数作为一个变量传入到了另一个函数中，那么该作为参数的函数的类型是：function1，即：(参数类型) => 返回类型

2. 代码案例

```
//说明
def plus(x: Int) = 3 + x
//说明
val result1 = Array(1, 2, 3, 4).map(plus(_))
println(result1.mkString(","))
```

3. 小结

1.map(plus(_)) 中的plus(_) 就是将plus这个函数当做一个参数传给了map，_这里代表从集合中遍历出来的一个元素。
2.plus(_) 这里也可以写成 plus 表示对 Array(1,2,3,4) 遍历，将每次遍历的元素传给plus的 x
3.进行 3 + x 运算后，返回新的Int ，并加入到新的集合 result1中
4.def map[B, That](f: A => B) 的声明中的 f: A => B 一个函数

12.3 匿名函数

1. 基本介绍

没有名字的函数就是匿名函数，可以通过函数表达式来设置匿名函数

2. 代码案例

```
val triple = (x: Double) => 3 * x
println(triple(3))
```

说明
1.(x: Double) => 3 * x 就是匿名函数
2.(x: Double) 是形参列表， => 是规定语法表示后面是函数体， 3 * x 就是函数体，如果有多行，可以 {} 换行写.
3.triple 是指向匿名函数的变量。

12.4 高阶函数

1. 基本介绍

能够接受函数作为参数的函数，叫做高阶函数 (higher-order function)。可使应用程序更加健壮。

2. 代码案例

```
//test 就是一个高阶函数，它可以接收f: Double => Double
def test(f: Double => Double, n1: Double) = {
    f(n1)
}
//sum 是接收一个Double,返回一个Double
def sum(d: Double): Double = {
    d + d
}
val res = test(sum, 6.0)
println("res=" + res)
```

3. 高阶函数可以返回函数类型

```
def minusxy(x: Int) = {
 (y: Int) => x - y //匿名函数
}
val result3 = minusxy(3)(5)
println(result3)
```

12.5 参数(类型)推断

1. 基本介绍

参数推断省去类型信息（在某些情况下[需要有应用场景]，参数类型是可以推断出来的，如list=(1,2,3) list.map()　　map中函数参数类型是可以推断的)，同时也可以进行相应的简写。

1.参数类型是可以推断时，可以省略参数类型
2.当传入的函数，只有单个参数时，可以省去括号
3.如果变量只在=>右边只出现一次，可以用_来代替

2. 代码案例

```
val list = List(1, 2, 3, 4)
println(list.map((x:Int)=>x + 1)) //(2,3,4,5)
println(list.map((x)=>x + 1))
println(list.map(x=>x + 1))
println(list.map(_ + 1))
val res = list.reduce(_+_)
```

3. 小结

1.map是一个高阶函数，因此也可以直接传入一个匿名函数，完成map
2.当遍历list时，参数类型是可以推断出来的，可以省略数据类型Int
println(list.map((x)=>x + 1))
3.当传入的函数，只有单个参数时，可以省去括
println(list.map(x=>x + 1))
4.如果变量只在=>右边只出现一次，可以用_来代替
println(list.map(_ + 1))

12.6 闭包

1. 基本介绍

闭包就是一个函数和与其相关的引用环境组合的一个整体(实体)

2. 代码案例

```
//1.用等价理解方式改写 2.对象属性理解
def minusxy(x: Int) = (y: Int) => x - y
//f函数就是闭包.
val f = minusxy(20)
println("f(1)=" + f(1)) // 19
println("f(2)=" + f(2)) // 18
```

3. 小结

1.(y: Int) => x - y
返回的是一个匿名函数 ，因为该函数引用到到函数外的x,那么该函数和x整体形成一个闭包
如：这里 val f = minusxy(20) 的f函数就是闭包
2.可以这样理解，返回函数是一个对象，而x就是该对象的一个字段，他们共同形成一个闭包
3.当多次调用f时（可以理解多次调用闭包），发现使用的是同一个x，所以x不变。
4.在使用闭包时，主要搞清楚返回函数引用了函数外的哪些变量，因为他们会组合成一个整体(实体),形成一个闭包

12.7 函数柯里化

1. 基本介绍

1.函数编程中，接受多个参数的函数都可以转化为接受单个参数的函数，这个转化过程就叫柯里化
2.柯里化就是证明了函数只需要一个参数而已。
3.不用设立柯里化存在的意义这样的命题。柯里化就是以函数为主体这种思想发展的必然产生的结果。(即：柯里化是面向函数思想的必然产生结果)

2. 代码案例

编写一个函数，接收两个整数，可以返回两个数的乘积，要求：

使用常规的方式完成：

```
def mul(x: Int, y: Int) = x * y
println(mul(10, 10))
```

使用闭包的方式完成：

```
def mulCurry(x: Int) = (y: Int) => x * y
println(mulCurry(10)(9))
```

使用函数柯里化完成：

```
def mulCurry2(x: Int)(y:Int) = x * y
println(mulCurry2(10)(8))
```

12.8 控制抽象

1. 基本介绍

1.参数是函数
2.函数参数没有输入值也没有返回值

2. 代码案例

```
def myRunInThread(f1: () => Unit) = {
    new Thread {
        override def run(): Unit = {
            f1()
        }
    }.start()
}
myRunInThread {
    () => println("干活咯！5秒完成...")
    Thread.sleep(5000)
    println("干完咯！")
}

简化处理，省略()
def myRunInThread(f1:  => Unit): Unit = {
     new Thread {
         override def run(): Unit = {
```

```
            f1
        }
    }.start()
}
myRunInThread {
    println("干活咯！5秒完成...")
    Thread.sleep(5000)
    println("干完咯！")
}
```

3. 进阶用法

```
var x = 10
def until(condition: => Boolean)(block: => Unit): Unit = {
//类似while循环，递归
if (!condition) {
    block
    until(condition)(block)
}
//      println("x=" + x)
//      println(condition)
//      block
//      println("x=" + x)
}

until(x == 0) {
    x -= 1
    println("x=" + x)
}
```

十三、泛型

13.1 基本介绍

1.如果我们要求函数的参数可以接受任意类型。可以使用泛型，这个类型可以代表任意的数据类型。
2.例如 List，在创建 List 时，可以传入整型、字符串、浮点数等等任意类型。那是因为 List 在 类定义时引用了泛型。比如在Java中：public interface List<E> extends Collection<E>

13.2 代码案例

1. 案例一

```
要求:
1.编写一个Message类
2.可以构建Int类型的Message,String类型的Message.
3.要求使用泛型来完成设计,(说明：不能使用Any)

object GenericUse {
  def main(args: Array[String]): Unit = {
    val mes1 = new StrMessage[String]("10")
    println(mes1.get)
    val mes2 = new IntMessage[Int](20)
    println(mes2.get)
  }
}

// 在 Scala 定义泛型用[T]， s 为泛型的引用
abstract class Message[T](s: T) {
  def get: T = s
}

// 子类扩展的时候，约定了具体的类型
class StrMessage[String](msg: String) extends Message(msg)
class IntMessage[Int](msg: Int) extends Message(msg)
```

2. 案例二

```
要求:
1.请设计一个EnglishClass (英语班级类)，在创建EnglishClass的一个实例时，需要指定[ 班级开班季节(spring,autumn,summer,winter)、班级名称、班级类型]
2.开班季节只能是指定的，班级名称为String， 班级类型是(字符串类型 "高级班", "初级班"..) 或者是 Int 类型(1, 2, 3 等)

// Scala 枚举类型
object SeasonEm extends Enumeration {
  type SeasonEm = Value //自定义SeasonEm，是Value类型,这样才能使用
```

```
  val spring, summer, winter, autumn = Value
}

object GenericDemo02 {
  def main(args: Array[String]): Unit = {
    //使用
    val class01 = new EnglishClass[SeasonEnum, String,
String](SeasonEnum.spring, "0705班", "高级班")
    println("class01 " + class01.classSeason + " " +
class01.className + class01.classType)

    val class02 = new EnglishClass[SeasonEnum, String,
Int](SeasonEnum.spring, "0707班", 1)
    println("class02 " + class02.classSeason + " " +
class02.className + class02.classType)
  }
}

// 定义一个泛型类
class EnglishClass[A, B, C](val classSeason: A, val
className: B, val classType: C){}
```

3. 案例三

要求:
1.定义一个函数，可以获取各种类型的 List 的中间index的值
2使用泛型完成

```
object GenericUse3 {
  def main(args: Array[String]): Unit = {
    // 定义一个函数，可以获取各种类型的 List 的中间index的值
    val list1 = List("jack",100,"tom")
    val list2 = List(1.1,30,30,41)

    println(getMidEle(list1))

  }
  // 定义一个方法接收任意类型的 List 集合
  def getMidEle[A](l: List[A])={
    l(l.length/2)
  }
```

```
}
```

十四、泛型约束

14.1 上界

1. 基本介绍

在 scala 里表示某个类型是 A 类型的子类型，也称上界或上限，使用 <: 关键字，语法如下：

```
[T <: A]
//或用通配符:
[_ <: A]
```

2. 代码案例

1. 案例一

```
object UpperBoundsDemo {
  def main(args: Array[String]): Unit = {
    //常规方式
    /*
    val compareInt = new CompareInt(-10, 2)
    println("res1=" + compareInt.greater)
    val compareFloat = new CompareFloat(-10.0f, -20.0f)
    println("res2=" + compareFloat.greater)*/
    /*val compareComm1 = new CompareComm(20, 30)
    println(compareComm1.greater)*/
    val compareComm2 = new
CompareComm(Integer.valueOf(20), Integer.valueOf(30))
    println(compareComm2.greater)
    val compareComm3 =
      new CompareComm(java.lang.Float.valueOf(20.1f),
java.lang.Float.valueOf(30.1f))
    println(compareComm3.greater)
    val compareComm4 = new CompareComm[java.lang.Float]
(201.9f, 30.1f)
    println(compareComm4.greater)
```

```
  }
}
/*class CompareInt(n1: Int, n2: Int) {
  def greater = if(n1 > n2) n1 else n2
}
class CompareFloat(n1: Float, n2: Float) {
  def greater = if(n1 > n2) n1 else n2
}*/
//使用上界的方式，可以有更好的通用性
class CompareComm[T <: Comparable[T]](obj1: T, obj2: T) {
    def greater = if(obj1.compareTo(obj2) > 0) obj1 else
obj2
}
```

2. 案例二

```
object LowerBoundsDemo {
  def main(args: Array[String]): Unit = {
    biophony(Seq(new Bird, new Bird)) //?
    biophony(Seq(new Animal, new Animal)) //?
    biophony(Seq(new Animal, new Bird)) //?
    biophony(Seq(new Earth, new Earth)) //?
  }
  def biophony[T <: Animal](things: Seq[T]) = things map
(_.sound)
}
class Earth { //Earth 类
  def sound(){ //方法
    println("hello !")
  }
}
class Animal extends Earth{
  override def sound() ={ //重写了Earth的方法sound()
    println("animal sound")
  }
}
class Bird extends Animal{
  override def sound()={ //将Animal的方法重写
    print("bird sounds")
  }
}
```

14.2 下界

1. 基本介绍

```
scala中下界
在 scala 的下界或下限，使用 >: 关键字，语法如下：

[T >: A]
//或用通配符:
[_ >: A]
```

2. 代码案例

```
object LowerBoundsDemo {
  def main(args: Array[String]): Unit = {

    biophony(Seq(new Earth, new Earth)).map(_.sound())
    biophony(Seq(new Animal, new Animal)).map(_.sound())
    biophony(Seq(new Bird, new Bird)).map(_.sound())

    val res = biophony(Seq(new Bird))
    val res2 = biophony(Seq(new Object))
    val res3 = biophony(Seq(new Moon))
    println("\nres2=" + res2)
    println("\nres3=" + res2)

  }
  def biophony[T >: Animal](things: Seq[T]) = things
}

class Earth { //Earth 类
  def sound(){ //方法
    println("hello !")
  }
}

class Animal extends Earth{
  override def sound() ={ //重写了Earth的方法sound()
    println("animal sound")
  }
```

```
}

class Bird extends Animal{
  override def sound(){ //将Animal的方法重写
    print("bird sounds")
  }
}

class Moon {}
```

3. 小结

1.对于下界，可以传入任意类型
2.传入和Animal直系的，是Animal父类的还是父类处理，是Animal子类的按照Animal处理
3.和Animal无关的，一律按照Object处理
4.也就是下界，可以随便传，只是处理是方式不一样
5.不能使用上界的思路来类推下界的含义

14.3 视图

1. 基本介绍

<% 的意思是“view bounds”(视界)，它比<:适用的范围更广，除了所有的子类型，还允许隐式转换类型。

```
def method [A <% B](arglist): R = ... 等价于:
def method [A](arglist)(implicit viewAB: A => B): R = ...
或等价于:
implicit def conver(a:A): B = …

<% 除了方法使用之外，class 声明类型参数时也可使用：
class A[T <% Int]
```

2. 代码案例

1. 案例一

```
object ViewBoundsDemo {
  def main(args: Array[String]): Unit = {
    val compareComm1 = new CompareComm(20, 30) //
    println(compareComm1.greater)

    val compareComm2 = new
CompareComm(Integer.valueOf(20), Integer.valueOf(30))
    println(compareComm2.greater)

    val compareComm4 = new CompareComm[java.lang.Float]
(201.9f, 30.1f)
    println(compareComm4.greater)
    //上面的小数比较，在视图界定的情况下，就可以这样写了
    val compareComm5 = new CompareComm(201.9f, 310.1f)
    println(compareComm5.greater)
  }
}

/**
  * <% 视图界定 view bounds
  *    会发生隐式转换
  */
class CompareComm[T <% Comparable[T]](obj1: T, obj2: T) {
  def greater = if(obj1.compareTo(obj2) > 0) obj1 else
obj2
}
```

2. 案例二

```
val p1 = new Person("tom", 10)
val p2 = new Person("jack", 20)
val compareComm2 = new CompareComm2(p1, p2)
println(compareComm2.getter)

class Person(val name: String, val age: Int) extends
Ordered[Person] {
  override def compare(that: Person): Int = this.age -
that.age
  override def toString: String = this.name + "\t" +
this.age}

class CompareComm2[T <% Ordered[T]](obj1: T, obj2: T) {
  def getter = if (obj1 > obj2) obj1 else obj2
  def geatter2 = if (obj1.compareTo(obj2) > 0) obj1 else
obj2
}
```

3. 案例三

```
// 隐式将Student -> Ordered[Person2]//放在object MyImplicit
中
implicit def person22OrderedPerson2(person: Person2) = new
Ordered[Person2]{
    override def compare(that: Person2): Int = person.age
- that.age
}

 val p1 = new Person2("tom", 110)
 val p2 = new Person2("jack", 20)
 import MyImplicit._
 val compareComm3 = new CompareComm2(p1, p2)
 println(compareComm3.geatter)

class Person2(val name: String, val age: Int)  {
  override def toString = this.name + "\t" + this.age
}
class CompareComm3[T <% Ordered[T]](obj1: T, obj2: T) {
  def geater = if (obj1 > obj2) obj1 else obj2
}
```

14.4 上下文界定

1. 基本介绍

与view bounds一样context bounds(上下文界定)也是隐式参数的语法糖。为语法上的方便，引入了”上下文界定”这个概念

2. 代码案例

```
object ContextBoundsDemo {
  //这里我定义一个隐式值  Ordering[Person]类型
  implicit val personComparator: Ordering[Person4] = new
Ordering[Person4] {
    override def compare(p1: Person4, p2: Person4): Int =
      p1.age - p2.age
  }

  def main(args: Array[String]): Unit = {
    //
    val p1 = new Person4("mary", 30)
    val p2 = new Person4("smith", 35)
    val compareComm4 = new CompareComm4(p1, p2)
    println(compareComm4.getter) // "smith", 35

    val compareComm5 = new CompareComm5(p1, p2)
    println(compareComm5.getter) // "smith", 35

    println("personComparator hashcode=" +
personComparator.hashCode())
    val compareComm6 = new CompareComm6(p1, p2)
    println(compareComm6.getter) // "smith", 35

  }
}


//一个普通的Person类
class Person4(val name: String, val age: Int) {

  //重写toString
```

```
  override def toString: String = this.name + "\t" +
this.age
}

//方式1
//说明:
//1. [T: Ordering] 泛型
//2. obj1: T, obj2: T 接受T类型的对象
//3. implicit Comparator: Ordering[T] 是一个隐式参数
class CompareComm4[T: Ordering](obj1: T, obj2: T)(implicit
comparator: Ordering[T]) {
  def getter: T = if (comparator.compare(obj1, obj2) > 0)
obj1 else obj2
}

//方式2
//方式2,将隐式参数放到方法内
class CompareComm5[T: Ordering](o1: T, o2: T) {
  def getter: T = {
    def f1(implicit comparator: Ordering[T]) =
comparator.compare(o1, o2) //返回一个数字
    //如果f1返回的值>0,就返回o1,否则返回o2
    if (f1 > 0) o1 else o2
  }
  def lower: T = {
    def f1(implicit comparator: Ordering[T]) =
comparator.compare(o1, o2) //返回一个数字
    //如果f1返回的值>0,就返回o1,否则返回o2
    if (f1 > 0) o2 else o1
  }
}

//方式3
//方式3,使用implicitly语法糖，最简单(推荐使用)
class CompareComm6[T: Ordering](o1: T, o2: T) {
  def getter: T = {
    //这句话就是会发生隐式转换，获取到隐式值 personComparator
    //底层仍然使用编译器来完成绑定(赋值的)工作
    val comparator = implicitly[Ordering[T]]
    println("comparator hashcode=" +
comparator.hashCode())
```

```
    if (comparator.compare(o1, o2) > 0) o1 else o2
  }
}
```

14.5 协变、逆变和不变

1. 基本介绍

1.Scala的协变(+)，逆变(-)，协变covariant、逆变contravariant、不可变invariant
2.对于一个带类型参数的类型，比如 List[T]，如果对A及其子类型B，满足List[B]也符合List[A]的子类型，那么就称为covariance(协变) ，如果List[A]是 List[B]的子类型，即与原来的父子关系正相反，则称为contravariance(逆变)。如果一个类型支持协变或逆变，则称这个类型为variance(翻译为可变的或变型)，否则称为invariance(不可变的)
3.在Java里，泛型类型都是invariant，比如 List<String> 并不是List<Object> 的子类型。而scala支持，可以在定义类型时声明(用加号表示为协变，减号表示逆变)，如: trait List[+T] // 在类型定义时声明为协变这样会把List[String]作为List[Any]的子类型。

2. 代码案例

C[+T]：如果A是B的子类，那么C[A]是C[B]的子类，称为协变
C[-T]：如果A是B的子类，那么C[B]是C[A]的子类，称为逆变
C[T]：无论A和B是什么关系，C[A]和C[B]没有从属关系。称为不变.

```
object CovariantContravariantDemo {
  def main(args: Array[String]): Unit = {
    val t1: Temp3[Sub] = new Temp3[Sub]("hello") //ok
    //val t2: Temp3[Sub] = new Temp3[Super]
("hello")//error
    //val t3: Temp3[Super] = new Temp3[Sub]
("hello")//error
    val t4: Temp3[Sub] = new Temp3[Sub]("hello") //ok
    val t5: Temp4[Super] = new Temp4[Sub]("hello") //ok
    //val t6: Temp4[Sub] = new Temp4[Super]("hello")
//error
    val t7: Temp5[Sub] = new Temp5[Sub]("hello") //ok
    val t8: Temp5[Sub] = new Temp5[Super]("hello") //ok
```

```
    //val t9: Temp5[Super] = new Temp5[Sub]("hello")
//error
  }
}


//协变
class Temp4[+A](title: String) { //Temp3[+A] //Temp[-A]
  override def toString: String = {
    title
  }
}

//逆变
class Temp5[-A](title: String) { //Temp3[+A] //Temp[-A]
  override def toString: String = {
    title
  }
}

//不变
class Temp3[A](title: String) { //Temp3[+A] //Temp[-A]
  override def toString: String = {
    title
  }
}

//支持协变
class Super //父类

//Sub是Super的子类
class Sub extends Super
```