[第一章 Scala核心 2](#_Toc21493)

[1 Scala简介 2](#_Toc11849)

[1.1 Scala特点 2](#_Toc31701)

[1.2 运行Scala代码方式 2](#_Toc20296)

[2 数据处理 4](#_Toc27549)

[2.1 变量的定义与使用 4](#_Toc12739)

[2.2 基本数据类型和操作 8](#_Toc12369)

[2.3 Range操作 12](#_Toc6747)

[3 表达式和条件式 13](#_Toc24845)

[3.1 if条件表达式 13](#_Toc7346)

[3.2 循环表达式 16](#_Toc8197)

[3.3 匹配表达式 19](#_Toc763)

[4 集合 21](#_Toc9274)

[4.1 数组 21](#_Toc28488)

[4.2 列表 25](#_Toc6700)

[4.3 集 27](#_Toc27654)

[4.4 映射 30](#_Toc23253)

[4.5 Option 33](#_Toc3456)

[4.6 迭代器与元组 34](#_Toc13225)

[5 函数式编程 36](#_Toc5222)

[5.1 函数 36](#_Toc3955)

[5.2 占位符语法 38](#_Toc16815)

[5.3 递归函数 40](#_Toc3912)

[5.4 嵌套函数 41](#_Toc20035)

[5.5 高阶函数 42](#_Toc23530)

[5.6 高阶函数的使用 44](#_Toc19803)

[第二章 Scala面向对象编程 46](#_Toc31559)

[1 类与对象基础 47](#_Toc13943)

[1.1 类定义 47](#_Toc4862)

[1.2 创建对象 47](#_Toc31825)

[1.3 类成员的访问 48](#_Toc2069)

[1.4 构造函数 49](#_Toc67)

[1.5 常见对象类型 50](#_Toc11581)

[1.6 抽象类与匿名类 52](#_Toc28146)

[2 继承与多态 53](#_Toc31833)

[2.1 类的继承 54](#_Toc10560)

[2.2 构造函数执行顺序 56](#_Toc32039)

[2.3 方法重写 57](#_Toc2254)

[2.4 多态 58](#_Toc18998)

[3 特质 59](#_Toc10648)

[3.1 特质的使用 59](#_Toc28306)

[3.2 特质与类 61](#_Toc19572)

[3.3 多重继承问题 63](#_Toc27085)

# 第一章 Scala核心

## 1 Scala简介

在大数据领域，新一代分布式计算系统Spark能同时在一个平台上完成批处理、机器学习、流式计算、图计算、SQL查询等功能，因此得到广泛应用。而Spark核心代码就是使用Scala语言开发的，虽然在Spark中可以使用Scala、Java、Python语言进行分布式程序开发，但Spark提供的首选语言就是Scala。本节将介绍Scala语言的特点以及运行Scala代码的的几种常用方式。

### 1.1 Scala特点

Scala是一门现代的多范式编程语言，平滑地集成了面向对象和函数式语言的特性，旨在以简练、优雅的方式来表达常用编程模式。Scala的设计吸收借鉴了许多种编程语言的思想，只有很少量特点是Scala自己独有的。Scala语言的名称来自于“可伸展的语言”，从写个小脚本到建立个大系统的编程任务均可胜任。Scala运行于Java平台（JVM，Java 虚拟机）上，并兼容现有的Java程序，Scala代码可以调用Java方法，访问Java成员变量，继承Java类和实现Java接口。在面向对象方面，Scala是一门非常纯粹的面向对象编程语言，也就是说，在Scala中，每个值都是对象，每个操作都是方法调用。

Spark的设计目的之一就是使程序编写更快更容易，这也是Spark选择Scala的原因所在。总体而言，Scala具有以下突出的优点：

|  |  |
| --- | --- |
| 1 | Scala具备强大的并发性，支持函数式编程，可以更好地支持分布式系统 |
| 2 | Scala语法简洁，能提供优雅的API |
| 3 | Scala兼容Java，运行速度快，且能融合到Hadoop生态圈中 |

Scala的优势是提供了REPL（Read-Eval-Print Loop，交互式解释器），因此，在Spark Shell中可进行交互式编程（即表达式计算完成就会输出结果，而不必等到整个程序运行完毕，因此可即时查看中间结果，并对程序进行修改），这样可以在很大程度上提升开发效率。

### 1.2 运行Scala代码方式

Scala代码可以在Spark Shell中运行，其自带Scala解释器。当然，Scala代码也可以在Scala Shell中直接运行。在实际开发过程中，为提高开发效率，一般会借助一些IDE如IDEA、Eclipse等进行Scala程序开发。由于本书只是对Scala语言的常用语法知识进行讲解，不涉及太过复杂的Scala程序，故在Scala Shell里便可完成相关知识的学习。

下面以输出HelloWorld为例介绍三种Scala代码编译执行的方式。

1)Scala解释器中直接运行相应的代码

登录Linux系统,打开命令行终端,输入scala就可以进入Scala解释器。输入以下代码：

scala> println("Hello, world!")

Hello, world!

结果会直接显示在代码下面。但有时要编写的代码不像上面的中那么简短，需要输入多行代码。且在Scala解释器中不能直接复制粘贴多行代码。此时可以使用：paste命令，进入paste模式。将代码编辑完后，按住ctrl+D即可退出paste模式，运行结果即可在下方显示。

scala> :paste

// Entering paste mode (ctrl-D to finish)

val str=List("Hello","World")

for(i<-0 to 1)

print(str(i))

println()

// Exiting paste mode, now interpreting.

HelloWorld

str: List[String] = List(Hello, World)

另外,在Scala解释器中输入 ：quit可以退出Scala解释器。

2)以应用程序对象的方式写入scala文件中，再通过控制台进行编译及执行.

登录Linux系统,打开命令行终端。在当前路径下创建Demo.scala文件,在命令行终端输入Vim Demo.scala,即可在当前路径自动创建Demo.scala文件。在文件中输入以下代码：

object HelloWorld {

/\* 第一个Scala程序

\* 它将输出 'Hello World'

\*/

def main(args: Array[String]) {

println("Hello, World!") // 输出 Hello World

}

}

下面用scalac命令编译Demo.scala代码文件，并用scala命令执行，如下：

scalac Demo.scala//编译的时候使用的是Scala文件名称

scala -classpath . HelloWorld //执行的时候使用的是HelloWorld对象名称

注意，上面命令中一定要加入”-classpath .”，否则会出现“No such file or class on classpath: HelloWorld”。上述命令执行后，会在屏幕上打印出“Hello, World!”。关于上面代码，需要重点说明几点：

|  |  |
| --- | --- |
| 1 | 在上面代码中，定义了程序的入口main()方法。可以看出，关于main()方法的定义，Java和Scala是不同的，在Java中是用静态方法（public static void main(String[] args)），而Scala中则必须使用对象方法，本例中，也就是HelloWorld对象中的main()方法。 |
| 2 | 对象的命名HelloWorld可以不用和文件名称一致，这里对象名称是HelloWorld，而文件名称却是Demo.scala。这点和Java是不同的，按照Java的命名要求，这里的文件名称就必须起名为HelloWorld.scala，但是，在Scala中是没有这个一致性要求的。 |
| 3 | Scala是大小写敏感的，所以，不要输入错误，比如把小写开头的object输成大写开头的Object。文件名Demo.scala和demo.scala也是两个不同的文件。 |

## 2 数据处理

数据处理是任何编程语言的入门基础。本节将详细介绍Scala中变量的定义与使用、命名规范、基本数据类型及其使用以及Range操作等相关知识。

### 2.1 变量的定义与使用

Scala中有三种类型的变量：可变变量、不可变变量和惰性变量。

|  |  |
| --- | --- |
| 可变变量 | 指的是变量被赋值后，变量值可以随着程序的运行而改变，可变变量使用关键字var进行定义，相当于Java中的变量，声明的时候需要进行初始化，初始化以后还可以再次对其赋值； |
| 不可变量 | 指的是变量一旦被赋值，变量值在程序运行的过程中不会被改变，不可变变量使用关键字val进行定义，类似于Java中final关键字，在声明时就必须被初始化，而且初始化以后不能再赋值。 |
| 惰性变量 | 指使用lazy关键字来修饰的变量，经过lazy关键字修饰的变量只有在真正使用时才会被赋值。而且lazy关键字只能修饰val类型的变量，而不能修饰var类型的变量。 |

下面分别举例三种变量类型的具体使用。

1. **val变量**

下面示例为使用关键字val定义不可变变量的情况。

//声明一个val变量，Scala会自动类型推断

scala> val TestString="Hello World!"

TestString: String = Hello World!

上面第1行代码是输入的代码，敲入回车后，Scala解释器会解析输入的代码，然后返回执行结果，第2行就是Scala解释器执行后返回的结果，从中可以看到，TestString变量的类型是String类型，变量的值是Hello World! 。尽管在第1行代码的声明中，没有给出TestString是String类型，但是，Scala具有“类型推断”能力，可以自动推断出变量的类型。

//声明一个val变量，显式声明变量的类型

scala> val TestString:String="Hello World!"

TestString: String = Hello World!

需要说明的是，上面的String类型全称是java.lang.String，也就是说，Scala的字符串是由Java的String类来实现的，因此，也可以使用java.lang.String来声明。

//String全称是java.lang.String，Scala会默认导入java.lang包

scala> val TestString:java.lang.String="Hello World!"

TestString: String = Hello World!

但是，为什么可以不用java.lang.String，而只需要使用String就可以声明变量呢？这是因为，在每个应用程序中，Scala都会自动添加一些引用，这样，就相当于在每个程序源文件的顶端都增加了一行下面的代码：

//手动导入相应的包

scala> import java.lang.\_

import java.lang.\_

//打印变量

scala> println(TestString)

Hello World!

因为TestString是val变量，因此，一旦初始化以后，就不能再次赋值，所以，下面执行的再次赋值操作会报错。

//不能被重新赋值，因为它是val变量

scala>TestString="Hello Scala!"

<console>:15: error: reassignment to val

TestString="Hello Scala!"

^

**2) var变量**

如果一些变量，需要在初始化以后还要不断修改它的值，则需要声明为var变量。下面把TestString声明为var变量，并且在声明的时候需要进行初始化。

//var声明可变变量

scala> var TestString=”Hello Cruel World!”

TestString:String: = Hello Cruel World!

然后，可以再次对TestString进行赋值：

//对var变量重新赋值

scala> TestString=”Goodbye Cruel World!”

TestString:String: = Goodbye Cruel World!

通过上述代码可以看到，Scala中使用var关键字定义的变量在程序运行过程中值可以随着程序的运行而发生改变。

尽管var变量可以重新赋值，但是不能改变为它指定的类型，所以不能将一个变量重新赋值为类型不兼容的数据。例如，定义一个类型为Int的变量，然后为它赋一个String值，这就会导致一个编译错误。

//定义一个类型为Int的变量x

scala>var x=5

x:Int =5

//重新赋一个String值，会导致编译错误。

scala>x=”Hello World”

<console>:8:error:type mismatch;

found : String(“Hello World”)

required : Int

x=”Hello World”

^

不过，如果定义一个类型为Double的var变量，再赋值为一个Int值，这是可以的，因为Int数可以自动转换为Double数。

//定义一个类型为Double的变量y

scala>var y=1.5

y:Double =1.5

//重新赋值一个Int数

scala>y=42

y:Double =42.0

**3) lazy惰性变量**

Scala中的变量可以使用lazy关键字来修饰，经过lazy关键字修饰的变量只有在真正使用时才会被赋值，具体见下列代码。

//普通val变量定义

scala>val TestString=”Hello Scala”

TestString: String = Hello Scala

//经过lazy关键字修饰的变量在定义时不被赋值

scala> lazy val TestString=”Hello Scala”

TestString:String =<lazy>

//在使用时，变量才会赋值

scala>TestString

res0: String= Hello Scala

代码lazy val TestString=”Hello Scala”定义了一个val类型的变量，该变量在定义时没有赋值，其返回结果为TestString:String =<lazy>，在使用时才被赋值。可以看到，与普通变量不同的是，普通变量赋值后立即会得到赋值结果。例如：

scala>lazy var TestString2=”Goodbye Scala”

<console>:1:error:lazy not allowed here. Only vals can be lazy

Lazy var TestString2=”Goodbye Scala”

lazy关键字不能用于var类型变量主要是为了避免程序运行过程中变量未使用便被重新赋值。

1. **命名规范**

Scala中的名字可以使用字母、数字和一些特殊的操作字符。因此可以使用标准算术运算符(例如\* 和:+)和常量(例如π和ф)取代比较长的名字，从而使代码更有表述性。

下面是结合字母、数字和字符构成Scala合法标识符的规则：

|  |  |
| --- | --- |
| 1 | 一个字母后跟有0个或多个字母和数字，如a123、aa123等。 |
| 2 | 一个字母后跟有0个或者多个字母和数字，然后是一个下划线(\_),后面是一个或是多个字母和 数字，或者一个或多个操作符,如a\_1、a\_b、a1\_\*等 |
| 3 | 一个或多个操作符，如+\*、-、π等。 |
| 4 | 一个或者多个除反引号外的任意字符，所有这些字符要包围在一对反引号中，如a.b是错误 命名，而加上反引号的`a.b`是正确的。 |
| 5 | Scala中有些保留字，不能用作标识符，但是反引号括起来除外，如return是保留字，标志为Do.return是非法的，但可以标志为Do. ` retern `。具体保留字一般可以通过查询保留字表得出，往往是常用的有特定含义的单词，如case、else等 |

Scala中的保留字如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| abstract | case | catch | class | def | do | else |
| Extends | false | final | finally | for | forSome | If |
| implicit | import | lazy | match | new | null | object |
| override | package | private | protectd | return | sealed | super |
| this | throw | trait | try | true | type | val |
| var | while | with | yield | - | ： | **=** |
| **=>** | **<-** | **<:** | **<%** | **>:** | **#** | **@** |

下面在Scala解释器中尝试这些命名规则：

//特殊字符”π”是一个合法的Scala标识符

scala>val π =3.14159

π：Double = 3.14159

//特殊字符”$”是一个合法的Scala标识符

Scala> val $ =”USD currency symbol”

$:String = USD currency symbol

//”o\_0 ”是一个合法的Scala标识符

Scala> val o\_0 =”Hmm”

o\_0:String =Hmm

//值名”50cent”是不合法的，因为标识符不能以数字开头。在这里，编译器最初会把这个名字解析为一个数字，解析到字母”c”时会出错

Scala>val 50cent=”$0.50”

<console>:1: error: Invalid literal number

val 50cent=”$0.50”

^

//值名”a.b”是不合法的，因为点号不是操作符字符

scala> val a.b=25

<console>:11: error: not found: value a

val a.b=25

^

//加上反引号重写就可以解决问题

scala> val `a.b`=4

a.b: Int = 4

按惯例，值和变量名应当用小写字母开头，然后其余单词的首字母大写。这通常称为camel case记法，尽管并不严格要求这样做，但建议所有scala开发人员都采用这种记法。这有助于与类型和类区分，类型与类也采用camel case记法,但会以一个大写字母开头。

### 2.2 基本数据类型和操作

**1)基本数据类型**

Scala的数据类型包括：Byte、Char、Short、Int、Long、Float、Double和Boolean。和Java不同的是，在Scala中，这些类型都是“类”，并且都是包scala的成员，比如，Int的全名是scala.Int。对于字符串，Scala用java.lang.String类来表示字符串。

Scala中数值数据类型如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型名 | 描述 | 大小 | 最小值 | 最大值 |
| Byte | 有符号整数 | 1字节 | -127 | 128 |
| Short | 有符号整数 | 2字节 | -32768 | 32767 |
| Int | 有符号整数 | 4字节 | -231 | 231-1 |
| Long | 有符号整数 | 8字节 | -263 | 263-1 |
| Float | 有符号浮点数 | 4字节 | n/a | n/a |
| Double | 有符号浮点数 | 8字节 | n/a | n/a |

下面分别介绍各数据类型的使用。

1. **Int类型**

Int类型对应的变量为整型变量，变量对应的是整型数据。Int类型变量的定义有多种方式，包括十六进制、十进制、八进制定义法。

下面的代码给出的是整型变量的十六进制定义法、十进制定义法和八进制定义法。

//16进制定义法

scala>val x=0x29

x：Int =41

//10进制定义法

scala>val x=41

x：Int =41

//8进制定义法

scala>val x=051

x：Int =41

在实际中使用最多的整型变量定义方式是十进制定义法。

1. **Float类型**

Float类型表示的是浮点数。如果直接输入浮点数，则Scala编译器会自动进行类型推导，并自动解释成Double类型，需要在浮点数后加F或f才能定义Float类型的变量。Float类型变量定义如下：

//要定义Float类型浮点数，需要在浮点数后面加F或f

scala>val floatNumber=3.14159F

floatNumber: Float=3.14159

//小写的f也可以

scala>val floatNumber=3.14159f

floatNumber: Float=3.14159

1. **Double类型**

Double数据类型表示的是双精度的浮点数。Double类型变量的定义如下：

//Double类型定义，直接输入浮点数，编译器会将其自动推断为Double类型

scala>val doubleNumber=3.14159

doubleNumber:Double=3.14159

双精度浮点类型的变量还可以采用指数表示法，浮点数后加E或e均可，例如：

//浮点数指数表示法，e也可以是E，0.314159e1与0.314159\*10等同

scala>val floatNumber=0.314159e1

floatNumber: Double=3.14159

1. **Char类型**

Char数据类型表示的是字符类型，用单引号将字符包裹起来。Char类型变量定义如下：

//字符定义，用’ ’将字符包裹

scala>var charLiteral=’A’

charLiteral:Char=A

部分特殊字符如双引号、换行符及反斜杠等的定义需要加转义符\或者使用对应的Unicode编码，下面的示例给出了双引号字符的定义：

//通过转义符\进行双引号的定义

scala>var x=’\” ’

x:Char = ”

//通过使用Unicode编码进行双引号的定义

scala>var y=’\u0022 ’

y:Char = ”

1. **String类型**

String数据类型表示的是字符串类型，用双引号””将字符串包裹起来。String类型变量的定义如下所示：

//字符串变量用双引号””包裹

scala>val helloWorld=”Hello World”

helloWorld:String =Hello World

如果字符串类型中有双引号，则需要使用转义符\。

//要定义”Hello World”,可以加入转义符\

scala>val helloWorld2=”\”Hello World\””

helloWorld2:String =”Hello World”

1. **Boolean类型**

Boolean数据类型表示的是布尔类型，Boolean类型变量的定义如下所示：

//直接定义Boolean类型变量

scala>var x=true

x:Boolean = true

//含逻辑表达式的Boolean类型变量

scala>var y=2>3

y:Boolean = false

1. **基本数据操作**

Scala中基本数据类型的操作主要包括算术运算操作、关系运算操作、逻辑运算操作。下面分别就这三种运算操作进行讲解。

**1.算术运算操作**

在Scala中，可以使用加(+)、减(-) 、乘(\*) 、除(/) 、余数（%）等操作符，而且，这些操作符就是方法。以加法为例，例如，5 + 3和(5).+(3)是等价的，也就是说：

a 方法 b

a.方法(b)

上面这二者是等价的。前者是后者的简写形式，这里的+是方法名，是Int类中的一个方法。具体代码如下:

//实际上调用了 (5).+(3)

sum1: Int = 8

//可以发现，写成方法调用的形式，和上面得到相同的结果

scala> val sum2 = (5).+(3)

sum2: Int = 8

需要注意的是，和Java不同，在Scala中并没有提供++和–操作符，当需要递增和递减时，可以采用如下方式表达：

//定义整型变量i

scala> var i = 5;

i: Int = 5

//将i递增

scala> i += 1

scala> println(i)

6

Scala语言还有一个非常值得注意的地方，那就是它提供了用+、-符号来表示正负数，并且这两个符号可以直接在操作中使用，例如：

//1 + -3 编译器将-3解释成一个负数

scala> var y=1 + -3

y: Int = -2

1. **关系运算操作**

Scala的关系运算操作包括大于(>)、小于(<)、大于等于(>=)和小于等于(<=)，会产生Boolean类型的结果。具体使用如下：

//>运算符

scala>var res=3>2

res: Boolean =true

//<=运算符，！为取反操作

scala>res=!(3<=-3)

res: Boolean = false

1. **逻辑运算操作**

逻辑运算操作包括逻辑与(&&)及逻辑或(||),逻辑与操作为真时必须保证两个变量同时为真，而逻辑或操作为真时只要求至少一个变量为真，具体使用如下：

//定义Boolean类型变量bool

scala>val bool=true

bool :Boolean = true

//逻辑与&&,同时为true时才会为true

scala>bool && bool

res0: Boolean = true

scala>bool && !bool

res1: Boolean = false

//逻辑或||，同时为false时才为false

scala>bool || bool

res0: Boolean = true

scala>bool || !bool

res1: Boolean = true

scala>false || false

res2: Boolean = false

### 

### 2.3 Range操作

有时需要一个数字序列，从某个起点到某个终点。比如在执行for循环时，i的值从1循环到5，这时就可以采用Range来实现。Range可以支持创建不同数据类型的数值序列，包括Int、Long、Float、Double、Char、BigInt和BigDecimal等。

在创建Range时，需要给出区间的起点和终点以及步长（默认步长为1）。创建的Range可以包含区间终点(to),也可以不包含区间终点(until)。下面通过几个示例来介绍：

1. 创建一个从1到5的数值序列，包含区间终点5，步长为1。

scala> 1 to 5

res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

上面的代码，也可以用下面的代码来实现：

scala> 1.to(5)

res1: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

1. 创建一个从1到5的数值序列，不包含区间终点5，步长为1 。

scala> 1 until 5

res2: scala.collection.immutable.Range = Range(1, 2, 3, 4)

1. 创建一个从1到10的数值序列，包含区间终点10，步长为2。

//Int类型

scala> 1 to 10 by 2

res3: scala.collection.immutable.Range = Range(1, 3, 5, 7, 9)

//Long类型

scala> 1L to 10L by 2

res4: scala.collection.immutable.NumericRange[Long] = NumericRange(1, 3, 5, 7, 9)

//BigInt类型

scala> BigInt(1) to BigInt(10) by 2

res5: scala.collection.immutable.NumericRange[BigInt] = NumericRange(1, 3, 5, 7, 9)

1. 创建一个从10到1的数值序列，包含区间终点1，步长为-2。

scala> 10 to 1 by -2

res6: scala.collection.immutable.Range = Range(10, 8, 6, 4, 2)

1. 创建一个浮点类型的数值序列，从0.5f到5.9f，步长为0.8f。

//Float类型

scala> 0.5f to 5.9f by 0.8f

res7: scala.collection.immutable.NumericRange[Float] = NumericRange(0.5, 1.3, 2.1, 2.8999999, 3.6999998, 4.5, 5.3)

//Double类型

scala> 0.5 to 5.9 by 0.8

res8: scala.collection.immutable.NumericRange[Double] = NumericRange(0.5, 1.3, 2.1, 2.9000000000000004, 3.7, 4.5, 5.3)

//BigDecimal类型

scala> BigDecimal(0.5) to BigDecimal(5.9) by 0.8

res9: scala.collection.immutable.NumericRange.Inclusive[scala.math.BigDecimal] = NumericRange(0.5, 1.3, 2.1, 2.9, 3.7, 4.5, 5.3)

1. 创建一个Char类型的数值序列，从’a’到’g’，步长为3。

scala> ‘a’ to ‘g’ by 3

res10: scala.collection.immutable.NumericRange[Char] = NumericRange(a,d,g)

## 3 表达式和条件式

表达式为函数式编程提供了基础，因为利用表达式可以返回数据而不会修改现有的数据，如变量。如果所有代码可以组织为一组有层次的表达式，包括一个或者多个返回值的表达式，使用不可变数据就很自然。表达式的返回值会传递到其他表达式，或者存储到值中。通过减少变量的使用，就可以减少函数和表达式的副作用。本节将详细介绍if条件表达式、while循环表达式、do...while循环表达式、for循环表达式以及匹配表达式等知识。

### 3.1 if条件表达式

if语句是许多编程语言中都会用到的控制结构。在Scala中，执行if语句时，会首先检查if条件是否为真，如果为真，就执行对应的语句块，如果为假，就执行下一个条件分支。

**1) if语句**

语法格式如下：

If（条件判断）{

//条件判断为真时执行

}

示例如下，建立一个Demo.scala文件，将以下代码放入

object Demo {

def main(args: Array[String]) {

var x = 10;

if( x < 20 ){

println("This is if statement");

}

}

}

保存文件后退出，再在命令行终端输入以下命令：

$ scalac Demo.scala

$ scala Demo

运行结果：

This is if statement

**2) If...else...语句**

语法格式如下：

if（条件判断）{

//条件判断为真时执行

}else{

//条件判断为假时执行

}

示例如下：

object Demo {

def main(args: Array[String]) {

var x = 30;

if( x < 20 ){

println("This is if statement");

} else {

println("This is else statement");

}

}

}

执行过程同上，运行结果：

This is else statement

**3) 多重if...else...的使用**

和Java一样，if语句可以采用各种嵌套的形式，语法格式如下：

if(条件判断 1){

//条件判断1为真时执行

} else if(条件判断 2){

//条件判断2为真时执行

} else if(条件判断3){

//条件判断3为真时执行

} else {

//条件判断1、2、3均为假时执行

}

示例如下：

object Demo {

def main(args: Array[String]) {

var x = 30;

if( x == 10 ){

println("Value of X is 10");

} else if( x == 20 ){

println("Value of X is 20");

} else if( x == 30 ){

println("Value of X is 30");

} else{

println("This is else statement");

}

}

}

运行结果：

Value of X is 30

1. **if语句的嵌套使用**

语法格式如下：

if(条件判断 1){

//条件判断1为真时执行

if(条件判断2){

条件判断2为真时执行

}

}

示例如下：

object Demo {

def main(args: Array[String]) {

var x = 30;

var y = 10;

if( x == 30 ){

if( y == 10 ){

println("X = 30 and Y = 10");

}

}

}

}

运行结果：

X = 30 and Y = 10

值得注意的是，Scala中的if表达式有一点与Java不同，Scala中的if表达式的值可以赋值给变量，比如：

val x = 6

val a = if (x>0) 1 else -1

上述代码执行结束后，a的值为1。

### 3.2 循环表达式

**1) while循环**

Scala中也有和Java类似的while循环语句，不过Scala中弱化了while循环语句的作用，在程序中不推荐使用while循环，尽量使用for循环或者递归来替代while循环。

while循环的语法格式如下：

while(条件判断){

//条件判断为真时执行

}

示例代码：

object Demo {

def main(args: Array[String]) {

var i = 9

while (i > 0) {

i -= 3

printf("here is %d\n",i)

}

}

}

运行结果：

here is 9

here is 6

here is 3

1. **do...while循环**

当然也会有do-while语句，语法格式如下：

do{

//执行循环体代码

}while(条件判断是否继续执行循环)

示例代码：

object Demo {

def main(args: Array[String]) {

var i = 9

do{ i -= 3; printf("here is %d\n",i)

}while (i > 0)

}

}

运行结果

here is 6

here is 3

here is 0

**3) for循环**

1、基础for循环

Scala中没有C、C++及Java等高级语言中的for(初始化变量；条件判断；更新变量)循环，而是有自己独特的for循环风格。Scala中的for循环语句格式如下：

for (变量<-表达式) {

//循环语句块

}

其中，“变量<-表达式”被称为“生成器（generator）”。

下面给出一个示例：

for (i <- 1 to 5) println(“i=”+i)

在上面语句中，i不需要提前进行变量声明，可以在for语句括号中的表达式中直接使用。语句中，“<-”表示，之前的i要遍历后面1到5的所有值。语句执行结束后，会打印出下面结果：

i=1

i=2

i=3

i=4

i=5

当然，在前面的Range那一节，介绍了Range的使用方法，因此，这里可以改变步长，比如设置步长为2，如下所示：

for (i <- 1 to 5 by 2) println(“i=”+i)

这样会得到下面结果：

i=1

i=3

i=5

2、有过滤条件的for循环

有时候，可能不希望打印出所有的结果，希望过滤出一些满足制定条件的结果，这个时候，就需要使用到称为“守卫(guard)”的表达式。比如，只希望输出1到5之中的所有偶数，可以采用以下语句：

for (i <- 1 to 5 if i%2==0) println(“i=”+i)

这样，就只会得到下面的偶数结果：

i=2

i=4

Scala也支持“多个生成器”的情形，可以用分号把它们隔开，比如：

scala> for(i<-1 to 2;j<-1 to 3)printf("%d \* %d = %d\n",i,j,i\*j)

运行上述代码后得到如下执行结果：

1 \* 1 = 1

1 \* 2 = 2

1 \* 3 = 3

2 \* 1 = 2

2 \* 2 = 4

2 \* 3 = 6

也可以给每个生成器都添加一个“守卫”，如下：

for (i <- 1 to 5 if i%2==0; j <- 1 to 3 if j!=i) println(i\*j)

运行上述代码后得到如下执行结果：

2

6

4

8

12

3、多重for循环

Scala中可以使用多重for循环，其语法格式如下：

for(变量1<-表达式 条件判断1;if 条件判断2....){

for(变量2<-表达式 条件判断1;if 条件判断2....){

//所有条件判断都满足时才执行循环中的语句；

}

}

示例代码如下：

object Demo {

def main(args: Array[String]) {

for(i<-1 to 5 if (i>3)){

for(i<-5 to 7 if (j==6)){

println(“i=”+i+”,j=”+j) }

}

}

}

运行结果：

i=4,j=6

i=5,j=6

4、 for推导式

有时候需要对上述过滤后的结果进行进一步的处理，这时，就可以采用yield关键字，对过滤后的结果构建一个集合。比如，在Scala解释器中可以采用以下语句：

scala> for (i <- 1 to 5 if i%2==0) yield i

res0 : scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

上面这种带有yield关键字的for循环，被称为“for推导式”。这个概念源自函数式编程，也就是说，通过for循环遍历一个或多个集合，对集合中的元素进行“推导”，从而计算得到新的集合，用于后续的其他处理。

### 3.3 匹配表达式

Java中有switch-case语句，但是，只能按顺序匹配简单的数据类型和表达式。相对而言，Scala中的模式匹配的功能则要强大得多，可以应用到switch语句、类型检查、“解构”等多种场合。

1. **简单匹配**

Scala的模式匹配最常用于match语句中。语法格式如下：

<表达式> match{

case <模式匹配> => <表达式>

[case...]

}

下面是一个简单的整型值的匹配示例:

object Demo {

def main(args: Array[String]) {

println(matchTest(3))

}

def matchTest(x: Int): String = x match {

case 1 => "one"

case 2 => "two"

case \_ => "many"

}

}

运行结果：

many

另外，在模式匹配的case语句中，还可以使用变量。

object Demo {

def main(args: Array[String]) {

println(matchTest(5))

}

def matchTest(x: Int): String = x match {

case 1 => "one"

case 2 => "two"

case 3 => "three"

case unexpected => unexpected + " is Not Allowed"

}

}

运行结果：

5 is Not Allowed

也就是说，当x=5时，值5会被传递给unexpected变量。

同时，在模式匹配的case语句中可以使用不同的数据类型。比如：

object Demo {

def main(args: Array[String]) {

println(matchTest("two"))

println(matchTest("test"))

println(matchTest(1))

}

def matchTest(x: Any): Any = x match {

case 1 => "one"

case "two" => 2

case y: Int => "scala.Int"

case \_ => "many"

}

}

运行结果：

2

many

One

可以在模式匹配中添加一些必要的处理逻辑。

for (elem <- List(1,2,3,4)){

elem match {

case \_ if (elem %2 == 0) => println(elem + " is even.")

case \_ => println(elem + " is odd.")

}

}

上面代码中if后面条件表达式的圆括号可以不要。执行上述代码后可以得到以下输出结果：

1 is odd.

2 is even.

3 is odd.

4 is even.

**2)case类匹配**

case类是一种特殊的类，它们经过优化以被用于模式匹配。

object Demo {

def main(args: Array[String]) {

val alice = new Person("Alice", 25)

val bob = new Person("Bob", 32)

val charlie = new Person("Charlie", 32)

for (person <- List(alice, bob, charlie)) {

person match {

case Person("Alice", 25) => println("Hi Alice!")

case Person("Bob", 32) => println("Hi Bob!")

case Person(name, age) => println(

"Age: " + age + " year, name: " + name + "?")

}

}

}

case class Person(name: String, age: Int)

}

运行结果如下：

Hi Alice!

Hi Bob!

Age: 32 year, name: Charlie?

## 4 集合

Scala中集合分为两种，一种是可变的集合，另一种是不可变的集合。可变集合可以被更新或修改，添加、删除、修改元素将作用于原集合。而不可变集合一旦被创建，便不能被改变，添加、删除、更新操作返回的是新的集合，原有集合保持不变。本节将介绍Scala中常用的集合类型如数组、列表、集、映射、选项、迭代器及元组并给出对应集合的常用函数的使用方法。

### 4.1 数组

数组为相同数据类型的元素按照一定的顺序排列的集合，在Scala中数组是最常用、最重要的数据结构。Scala中的数组分为定长数组和变长数组，定长数组在定义时长度被确定，在运行时数组长度不会发生变化，而变长数组内存空间长度会随着程序运行的需要而动态扩容。下面对定长数组和变长数组的使用、遍历方式、多维数组等进行介绍。

1. 定长数组(Array)

定长数组指的是数组长度在定义时被确定，数组占有的内存空间在程序运行时不会被改变，定长数组的定义格式如下(以String类型为例)：

var z:Array[String] = new Array[String](3)

或者是

var z = new Array[String](3)

以上均是定义一个长度为3的字符串数组。值得注意的是，字符串等非数值对象类型在数组定义时若没有赋值，将会被初始化为null，而数值型数组在数组定义时未显性赋值，则被初始化为0。

可以给数组各字符串元素自行赋值，如：

z(0) = "Zara"; z(1) = "Nuha"; z(4/2) = "Ayan"

或者也可以在一开始定义数组时便进行赋值，比如：

var z = Array("Zara", "Nuha", "Ayan")

这种定义方式没有直接通过显式地new创建，而是调用了其apply方法进行数组创建操作。

1. 变长数组(ArrayBuffer)

变长数组在程序运行过程中，其数组长度可以随程序运行的需要而增加。最常用的变长数组为ArrayBuffer,它在包scala.collection.mutable中被使用时需要显式地引入，具体代码如下：

import scala.collection.mutable.ArrayBuffer

object Demo {

def main(args: Array[String]) {

//定义动态数组z

val z=ArrayBuffer[String]()

//向数组中添加元素

z+=”Zara”

println(z.length)

//一次添加多个元素

z+=(“Nuha”, "Ayan")

println(z.length)

//在数组索引为1地位置插入元素”Amy”

z.insert(1,”Amy”)

println(z(1))

//删除索引为2的“Nuha”,输出新的索引为2 的元素

z.remove(2,1)//从索引2开始删除1个元素

println(z(2))

}

}

运行结果：

1

3

Amy

Ayan

1. 数组遍历

上一小节中，已经介绍了for循环的使用方法，而集合遍历使用的都是for循环，数组遍历也不例外。数组的遍历有两种方式，通过索引遍历和直接数组遍历。具体示例如下：

object Demo {

def main(args: Array[String]) {

var myList = Array(1.9, 2.9, 3.4, 3.5)

// 直接数组遍历输出所有元素

for ( x <- myList ) {

println( x )

}

// 所有元素求和

var total = 0.0;

//索引遍历所有元素，进行累加操作

for ( i <- 0 to (myList.length - 1)) {

total += myList(i);

}

println("Total is " + total);

// /索引遍历所有元素，寻找数组最大值

var max = myList(0)

for ( i <- 1 to (myList.length - 1) ) {

if (myList(i) > max) max = myList(i);

}

println("Max is " + max);

}

}

运行结果如下：

1.9

2.9

3.4

3.5

Total is 11.7

Max is 3.5

无论是定长数组还是变长数组，在遍历时均可以生成新的数组。生成新数组时，原来数组内容保持不变。代码如下：

import Array.\_

object Demo {

def main(args: Array[String]) {

var myList1 = Array(1.9, 2.9, 3.4, 3.5)

//生成新的数组myList2

var myList2 = for(x<-myList1)yield x+1

// 输出所有数组元素

for ( x <- myList32) {

println( x )

}

}

}

运行结果如下：

2.9

3.9

4.4

4.5

另外，可以通过Range生成数组，并对其进行循环遍历,示例代码如下：

import Array.\_

object Demo {

def main(args: Array[String]) {

//使用Range生成数组

var myList1 = range(10, 20, 2)

var myList2 = range(10,20)

// 打印所有数组元素

for ( x <- myList1 ) {

print( " " + x )

}

println()

for ( x <- myList2 ) {

print( " " + x )

}

}

}

运行结果如下：

10 12 14 16 18

10 11 12 13 14 15 16 17 18 19

1. 多维数组

实现多维数组的定义如下：

var myMatrix = ofDim[Int](3,3)

上述定义了三行三列的整型二维数组。与一维数组相同，也可以对二维数组进行赋值及循环遍历，具体见以下代码：

import Array.\_

object Demo {

def main(args: Array[String]) {

//定义三行三列的整型二维数组

var myMatrix = ofDim[Int](3,3)

//给各元素赋值

for (i <- 0 to 2) {

for ( j <- 0 to 2) {

myMatrix(i)(j) = j;

}

}

// 打印二维数组

for (i <- 0 to 2) {

for ( j <- 0 to 2) {

print(" " + myMatrix(i)(j));

}

println();

}

}

}

运行结果如下：

0 1 2

0 1 2

0 1 2

### 4.2 列表

同数组一样，List也是Scala语言中应用十分广泛的集合类型数据结构。Scala列表与数组非常相似，列表中的所有元素都具有相同的类型，但是有一个个重要的区别。列表是不可变的，这意味着列表的元素不能被赋值更改。

1)创建列表

列表的创建如下所示：

//字符串类型List

scala>val fruit: List[String] = List("apples", "oranges", "pears")

fruit: List[String] = List(apples, oranges, pears)

//前一个语句与下面语句等同

scala>val fruit: List[String] = List.apply("apples", "oranges", "pears")

fruit: List[String] = List(apples, oranges, pears)

//数值类型List

scala>val nums: List[Int] = List(1, 2, 3, 4)

nums: List[Int] = List(1, 2, 3, 4)

//多重List，List的子元素为List

scala>val multiDList=List(List(1,2,3),List(4,5,6),List(7,8,9))

multiDList=:List[List[Int]]=List(List(1,2,3),List(4,5,6),List(7,8,9))

//遍历List

scala>for(i<-multiDList)println(i)

List(1,2,3)

List(4,5,6)

List(7,8,9)

所有的列表都可以用两个基本的构建块来定义，一个尾部为Nil，并且用：：进行右连接。Nil也代表空列表。上面的列表可以定义如下。

//字符串类型List

scala>val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

fruit: List[String] = List(apples, oranges, pears)

//数值类型List

scala>val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

nums: List[Int] = List(1, 2, 3, 4)

// 定义空列表

scala>val empty = Nil

empty: scala.collection.immutable.Nil.type = List()

//多重List，List的子元素为List

scala>val multiDList = (1 :: (2 :: (3 :: Nil))) ::

(4 :: (5 :: (6 :: Nil))) ::

(7 :: (8 :: (9 :: Nil))) :: Nil

multiDList=:List[List[Int]]=List(List(1,2,3),List(4,5,6),List(7,8,9))

可以使用list .fill()方法创建一个包含相同元素的零或多个副本的列表。具体代码如下：

object Demo {

def main(args: Array[String]) {

val fruit = List.fill(3)("apples") // 重复元素apples ,3次

println( "fruit : " + fruit )

val num = List.fill(10)(2) //重复元素2 ,10 次

println( "num : " + num )

}

}

运行结果如下：

fruit : List(apples, apples, apples)

num : List(2, 2, 2, 2, 2, 2, 2, 2, 2, 2)

1. 基本操作

List的基本操作包括:

|  |  |
| --- | --- |
| head | 该方法返回列表的第一个元素。 |
| tail | 该方法返回包含除第一个元素之外的所有元素的列表。 |
| isEmpty | 如果列表为空，则该方法返回true，否则为false。 |

下面的示例说明如何使用上述方法。

object Demo {

def main(args: Array[String]) {

val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

val nums = Nil

println( "Head of fruit : " + fruit.head )

println( "Tail of fruit : " + fruit.tail )

println( "Check if fruit is empty : " + fruit.isEmpty )

println( "Check if nums is empty : " + nums.isEmpty )

}

}

运行结果如下：

Head of fruit : apples

Tail of fruit : List(oranges, pears)

Check if fruit is empty : false

Check if nums is empty : true

3)连接列表

可以用::操作符或List.:::()方法或List.concat()方法，添加两个或更多的列表。具体代码如下：

object Demo {

def main(args: Array[String]) {

val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))

val fruit2 = "mangoes" :: ("banana" :: Nil)

// 使用 ::: 操作符连接两个或者多个列表

var fruit = fruit1 ::: fruit2

println( "fruit1 ::: fruit2 : " + fruit )

// 使用集合.:::()方法连接两个列表

fruit = fruit1.:::(fruit2)

println( "fruit1.:::(fruit2) : " + fruit )

// 通过两个或多个列表作为参数。

fruit = List.concat(fruit1, fruit2)

println( "List.concat(fruit1, fruit2) : " + fruit )

}

}

运行结果如下：

fruit1 ::: fruit2 : List(apples, oranges, pears, mangoes, banana)

fruit1.:::(fruit2) : List(mangoes, banana, apples, oranges, pears)

List.concat(fruit1, fruit2) : List(apples, oranges, pears, mangoes, banana)

4.3 集

集(Set)是不重复元素的集合。列表中的元素是按照插入的先后顺序来组织的，但是，”集”中的元素并不会记录元素的插入顺序，而是以“哈希”方法对元素的值进行组织，所以，它允许你快速地找到某个元素。

1)创建集Set

集包括可变集和不可变集，缺省情况下创建的是不可变集，通常使用不可变集。下面用默认方式创建一个不可变集，如下（在Scala解释器中执行）：

scala> var mySet = Set("Hadoop","Spark")

mySet: scala.collection.immutable.Set[String] = Set(Hadoop, Spark)

scala> mySet += "Scala" //向mySet中增加新的元素

scala> println(mySet.contains("Scala"))

true

上面声明时，如果使用val mySet += “Scala”执行时会报错，所以需要声明为var。如果要声明一个可变集，则需要引入scala.collection.mutable.Set包，具体如下（在Scala解释器中执行）：

//导入可变集包

scala> import scala.collection.mutable.Set

import scala.collection.mutable.Set

//定义一个集合，这里使用的是mutable

scala> val myMutableSet = Set("Database","BigData")

myMutableSet: scala.collection.mutable.Set[String] = Set(BigData, Database)

//向集中添加一个元素。scala> myMutableSet += "Cloud Computing"

res0: myMutableSet.type = Set(BigData, Cloud Computing, Database)

//输出结果

scala> println(myMutableSet)

Set(BigData, Cloud Computing, Database)

同前面的列表数组不同的是，Set在插入元素时并不保证元素的顺序，默认情况下，Set的实现方式是HashSet实现方式，集合的元素通过HashCode值进行组织。上面代码中，myMutableSet声明为val变量（不是var变量），由于是可变集，因此，可以正确执行myMutableSet += “Cloud Computing”，不会报错。值得注意的是，虽然可变集和不可变集都有添加或删除元素的操作，但是，二者有很大的区别。对不可变集进行操作，会产生一个新的集，原来的集并不会发生变化。 而对可变集进行操作，改变的是该集本身。

1. 集的基本操作

集的基本操作也包括head、tail和isEmpty。具体示例如下：

object Demo {

def main(args: Array[String]) {

val fruit = Set("apples", "oranges", "pears")

val nums: Set[Int] = Set()

println( "Head of fruit : " + fruit.head )

println( "Tail of fruit : " + fruit.tail )

println( "Check if fruit is empty : " + fruit.isEmpty )

println( "Check if nums is empty : " + nums.isEmpty )

}

}

运行结果如下：

Head of fruit : apples

Tail of fruit : Set(oranges, pears)

Check if fruit is empty : false

Check if nums is empty : true

同时可以使用 Set.min方法找到最小值和 Set.max方法，以找出集合中可用元素的最大值。下面是显示程序的示例。

object Demo {

def main(args: Array[String]) {

val num = Set(5,6,9,20,30,45)

//在集合中查找最大值与最小值

println( "Min element in Set(5,6,9,20,30,45) : " + num.min )

println( "Max element in Set(5,6,9,20,30,45) : " + num.max )

}

}

运行结果如下：

Min element in Set(5,6,9,20,30,45) : 5

Max element in Set(5,6,9,20,30,45) : 45

此外，可以使用Set.& 方法或Set.intersect方法来查找两个集合之间的公共值。具体代码如下：

object Demo {

def main(args: Array[String]) {

val num1 = Set(5,6,9,20,30,45)

val num2 = Set(50,60,9,20,35,55)

// 查找两个集合之间的公共值

println( "num1.&(num2) : " + num1.&(num2) )

println( "num1.intersect(num2) : " + num1.intersect(num2) )

}

}

运行结果如下：

num1.&(num2) : Set(20, 9)

num1.intersect(num2) : Set(20, 9)

1. 连接集Set

可以使用++运算符或Set.+()方法来连接两个或多个集合，但在添加集时，它将删除重复元素。

下面是连接两个集合的示例：

object Demo {

def main(args: Array[String]) {

val fruit1 = Set("apples", "oranges", "pears")

val fruit2 = Set("mangoes", "banana")

// 使用++ 操作符连接两个或多个集合

var fruit = fruit1 ++ fruit2

println( "fruit1 ++ fruit2 : " + fruit )

// 使用 ++ 作为方法连接两个集合

fruit = fruit1.++(fruit2)

println( "fruit1.++(fruit2) : " + fruit )

}

}

运行结果如下：

fruit1 ++ fruit2 : Set(banana, apples, mangoes, pears, oranges)

fruit1.++(fruit2) : Set(banana, apples, mangoes, pears, oranges)

4.4 映射

Scala映射是键/值对的集合。任何值都可以根据其键来检索。键在映射中是唯一的，但是值不一定是唯一的。映射也被称为哈希表。有两种映射，不变的和可变的。可变和不可变对象之间的区别是，当对象是不可变的时，对象本身就不能被改变。默认情况下，Scala使用不可变映射。如果想要使用可变映射，就必须导入scala.collection.mutable.Map包。

1)创建映射

下面创建一个不可变映射：

scala> val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")

colors: scala.collection.immutable.Map[String,String] = Map(red -> #FF0000, azure -> #F0FFFF, peru -> #CD853F)

如果要获取映射中的值，可以通过键来获取，如下：

scala> println(colors("red"))

#FF0000

通过"red"这个键，可以获得值#FF0000。上面定义的是不可变映射，是无法更新映射中的元素的，也无法增加新的元素。如果要更新映射的元素，就需要定义一个可变的映射，如下：

//导入Map包

scala> import scala.collection.mutable.Map

import scala.collection.mutable.Map

//定义可变映射

scala> val colors2 = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")

colors2: scala.collection.mutable.Map[String,String] = Map(azure -> #F0FFFF, red -> #FF0000, peru -> #CD853F)

//更改映射键值对

scala> colors2("red" ) = "#9C661F"

//添加映射键值对

scala> colors2("green") ="#00FF00"

//输出映射键值对

scala> for ((k,v) <- colors2) printf("( %s , %s)\n",k,v)

( green , #00FF00)

( azure , #F0FFFF)

( red , #9C661F)

( peru , #CD853F)

val colors2 = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")

colors2("red" ) = "#9C661F"//更新已有元素的值

colors2("green") = “#00FF00”//添加新元素

如果要将一个键值对添加到映射，可以使用运算符+如下：

//定义一个空映射

scala> var A:Map[Char,Int] = Map()

A: Map[Char,Int] = Map()

//通过运算符+向映射中添加元素

scala> A += ('I' -> 1)

scala> A += ('K' -> 10)

scala> A += ('L' -> 100)

//输出映射的键值对

scala> for ((k,v) <- A) printf("( %s , %d)\n",k,v)

( I , 1)

( J , 5)

( K , 10)

( L , 100)

1. 映射基本操作

映射的基本操作主要包括keys、values和isEmpty。

|  |  |
| --- | --- |
| keys | 该方法返回一个可迭代的包含映射中的每个键。 |
| values | 这个方法返回一个包含每个值的迭代。 |
| isEmpty | 如果列表为空，则该方法返回true，否则为false。 |

具体示例如下所示。

object Demo {

def main(args: Array[String]) {

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")

val nums: Map[Int, Int] = Map()

println( "Keys in colors : " + colors.keys )

println( "Values in colors : " + colors.values )

println( "Check if colors is empty : " + colors.isEmpty )

println( "Check if nums is empty : " + nums.isEmpty )

}

}

运行结果如下：

Keys in colors : Set(red, azure, peru)

Values in colors : MapLike(#FF0000, #F0FFFF, #CD853F)

Check if colors is empty : false

Check if nums is empty : true

同时，可以使用 Map.contains方法来检测一个映射中是否存在一个给定的关键字。具体代码如下：

object Demo {

def main(args: Array[String]) {

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF", "peru" -> "#CD853F")

if( colors.contains( "red" )) {

println("Red key exists with value :" + colors("red"))

} else {

println("Red key does not exist")

}

if( colors.contains( "maroon" )) {

println("Maroon key exists with value :" + colors("maroon"))

} else {

println("Maroon key does not exist")

}

}

}

运行结果如下：

Red key exists with value :#FF0000

Maroon key does not exist

3)循环遍历映射

循环遍历映射，是经常需要用到的操作，基本格式是：

for ((k,v) <- 映射) 语句块

下面给出一个示例如下：

object Demo {

def main(args: Array[String]) {

val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF","peru" -> "#CD853F")

//使用for循环输出键值对

for ((k,v) <- colors) printf("Color is : %s and the code is: %s\n",k,v)

//使用foreach输出键值对

colors.keys.foreach{ i =>

print( "Key = " + i )

println(" Value = " + colors(i) )}

}

}

运行结果如下：

Color is : red and the code is: #FF0000

Color is : azure and the code is: #F0FFFF

Color is : peru and the code is: #CD853F

Key = red Value = #FF0000

Key = azure Value = #F0FFFF

Key = peru Value = #CD853F

或者，也可以只遍历映射中的k或者v。比如说，如果只想把所有键打印出来：

for (k<-colors.keys) println(k)

运行结果如下：

red

azure

peru

再比如说，如果只想把所有值打印出来：

for (v<-colors.values) println(v)

运行结果如下：

#FF0000

#F0FFFF

#CD853F

### 4.5 Option

在Scala程序中经常使用Option类型，可以将其与Java中可用的空值进行比较，后者表示没有任何值。例如，java.util的get方法。HashMap返回存储在HashMap中的值，如果没有找到值，则返回null。

假设有一个方法，它根据主键从数据库中检索记录。

def findPerson(key: Int): Option[Person]

如果发现记录，则该方法将返回一些[Person]，但如果未找到记录，则不返回。示例如下：

object Demo {

def main(args: Array[String]) {

val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

println("capitals.get( \"France\" ) : " + capitals.get( "France" ))

println("capitals.get( \"India\" ) : " + capitals.get( "India" ))

}

}

运行结果如下：

capitals.get( "France" ) : Some(Paris)

capitals.get( "India" ) : None

将可选值分离的最常见方法是通过模式匹配。例如，尝试下面的程序。

object Demo {

def main(args: Array[String]) {

val capitals = Map("France" -> "Paris", "Japan" -> "Tokyo")

println("show(capitals.get( \"Japan\")) : " + show(capitals.get( "Japan")) )

println("show(capitals.get( \"India\")) : " + show(capitals.get( "India")) )

}

def show(x: Option[String]) = x match {

case Some(s) => s

case None => "?"

}

}

运行结果如下：

show(capitals.get( "Japan")) : Tokyo

show(capitals.get( "India")) : ?

如果在没有值的情况下给定值或者访问默认值时，可以使用getOrElse()方法。下面的示例程序演示了如何使用getOrElse()方法在没有值的情况下访问值或默认值。

object Demo {

def main(args: Array[String]) {

val a:Option[Int] = Some(5)

val b:Option[Int] = None

println("a.getOrElse(0): " + a.getOrElse(0) )

println("b.getOrElse(10): " + b.getOrElse(10) )

}

}

运行结果如下：

a.getOrElse(0): 5

b.getOrElse(10): 10

可以使用isEmpty()方法检查选项是否为空。下面的示例程序演示如何使用isEmpty()方法检查选项是否为空。

object Demo {

def main(args: Array[String]) {

val a:Option[Int] = Some(5)

val b:Option[Int] = None

println("a.isEmpty: " + a.isEmpty )

println("b.isEmpty: " + b.isEmpty )

}

}

运行结果如下：

a.isEmpty: false

b.isEmpty: true

### 4.6 迭代器与元组

1)迭代器（Iterator）

在Scala中，迭代器（Iterator）不是一个集合，但是，提供了访问集合的一种方法。当构建一个集合需要很大的开销时（比如把一个文件的所有行都读取内存），迭代器就可以发挥很好的作用。迭代器包含两个基本操作：next和hasNext。next可以返回迭代器的下一个元素，hasNext用于检测是否还有下一个元素。有了这两个基本操作，就可以顺利地遍历迭代器中的所有元素了。通常可以通过while循环或者for循环实现对迭代器的遍历。

1. while循环

object Demo {

def main(args: Array[String]) {

val iter = Iterator("Hadoop","Spark","Scala")

while (iter.hasNext) {

println(iter.next())

}

}

}

运行结果如下：

Hadoop

Spark

Scala

注意，上述操作执行结束后，迭代器会移动到末尾，就不能再使用了，如果继续执行一次println(iter.next)就会报错。另外，上面代码中，使用iter.next和iter.next()都是可以的，但是，hasNext后面不能加括号。

1. for循环

object Demo {

def main(args: Array[String]) {

val iter = Iterator("Hadoop","Spark","Scala")

for (elem <- iter) {

println(elem)

}

}

}

运行结果如下：

Hadoop

Spark

Scala

所以两种迭代方式可以得到相同的结果。

2)元组(Tuple)

元组是不同类型的值的聚集。元组和列表不同，列表中各个元素必须是相同类型，而元组可以包含不同类型的元素。下面声明一个元组:

scala> (1,2) //不声明，直接创建元组

res0 :(Int, Int) = (1,2)

scala> 1->2 //同(1,2)

res1:(Int, Int) = (1,2)

scala> (1,"Alice","Math",95.5)// 不声明，直接创建不同类型元素的元组

res2: (Int, String, String, Double) = (1,Alice,Math,95.5)

由上述代码可知，元组中可以包含不同类型的元素。另外关于元组还有两点需要说明。其一、元组示例化以后，和Array数组不同，Array数组的索引从0开始，而元组的索引从1开始。其二、调用元组tuple元素的方法\_1、\_2、\_3来分别调用每一个元素。

下面是通过println函数输出元组tuple的每一个元素输出的具体示例。

object Demo {

def main(args: Array[String]) {

val tuple = ("BigData",2015,45.0)

println(tuple.\_1)

println(tuple.\_2)

println(tuple.\_3)

}

}

运行结果如下：

BigData

2015

45.0

从上述代码的执行效果可以看出，声明一个元组是很简单的，只需要用圆括号把多个元组的元素包围起来就可以了。

## 5 函数式编程

Scala是一门多范式编程语言，混合了面向对象编程和函数式编程的风格。在过去很多年，面向对象编程一直是主流，但是，随着大数据时代的到来，函数式编程开始迅速崛起，因为，函数式编程可以较好满足分布式并行编程的需求（函数式编程一个重要特性就是值不可变性，这对于编写可扩展的并发程序而言可以带来巨大好处，因为它避免了对公共的可变状态进行同步访问控制的复杂问题）。本章将介绍有关Scala函数的基础知识。

### 5.1 函数

1. **函数字面量**

函数字面量可以体现函数式编程的核心理念。字面量包括整数字面量、浮点数字面量、布尔型字面量、字符字面量、字符串字面量、符号字面量、函数字面量和元组字面量。举例如下：

val i = 123 //123就是整数字面量

val i = 3.14 //3.14就是浮点数字面量

val i = true //true就是布尔型字面量

val i = 'A' //'A'就是字符字面量

val i = "Hello" //"Hello"就是字符串字面量

除了函数字面量会比较陌生以外，其他几种字面量都很容易理解。现在来认识什么是“函数字面量”。在非函数式编程语言里，函数的定义包含了“函数类型”和“值”两种层面的内容。但是，在函数式编程中，函数是“头等公民”，可以像任何其他数据类型一样被传递和操作，也就是说，函数的使用方式和其他数据类型的使用方式完全一致了。这时，就可以像定义变量那样去定义一个函数，由此导致的结果是，函数也会和其他变量一样，开始有“值”。就像变量的“类型”和“值”是分开的两个概念一样，函数式编程中，函数的“类型”和“值”也成为两个分开的概念，函数的“值”，就是“函数字面量”。

**2) 函数类型和值**

在Scala中，一个标准的函数定义如下：

def gcd(x:Int,y:Int)={

if(x%y==0) y

else

gcd(x,x%y)

}

def关键字用于声明一个函数，gcd为函数名称，函数名称与变量名的定义类似，可以是任意的合法字符串，(x:Int,y:Int)为函数参数，函数参数后面的：Int用于指定函数的返回值类型，=号后面是函数体，函数体如果是多行语句，则需要将其放在大括号中，如果只有一条语句，则可以省略。

//函数体中只有一行语句时，可以省略{}

scala>def gcd(x:Int,y:Int):Int= if(x%y==0) y else gcd(x,x%y)

gcd:(x:Int,y:Int)Int

函数中最后一条执行语句为函数的返回值，函数返回值可以加return关键词，也可以将其省略。

//return关键字也可以不省略

scala>def gcd(x:Int,y:Int):Int= {

if(x%y==0)

return y

else

return gcd(x,x%y)

}

gcd:(x:Int,y:Int)Int

Scala具有类型推导功能，会根据最终的返回值推导函数的返回值类型，因此在实际应用中也常常会省略函数的返回值。

//省略函数返回值，Scala会通过类型推导来确定函数的返回值类型

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

sum:(x:Int,y:Int)

不过，类型推导有两个限制：

1. 如果需要return关键字指定返回值，则必须显示地指定函数返回值的类型。

scala>def sum(x:Int,y:Int)=return x+y

<console>:7: error: method sum has return statement;needs result type

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

1. 如果函数中存在递归调用，则必须显示地指定函数返回值的类型。

scala>def gcd(x:Int,y:Int)= {

if(x%y==0) return y

else return gcd(x,x%y)

}

<console>:12:error:recursive method gcd needs result type

gcd(y,x%y)

上面定义个这个函数的“类型”如下：

(Int,Int) => Int

圆括号可以省略，如下：

Int => Int

上面就得到了函数的“类型”，下面看看如何得到函数的“值”。实际上，只要把函数定义中的类型声明部分去除，剩下的就是函数的“值”，比如：

(value) => {value += 1} //只有一条语句时，大括号可以省略

上面就是函数的“值”，需要注意的是，采用“=>”而不是“=”，这是Scala的语法要求。

现在，再按照比较熟悉的定义变量的方式，采用Scala语法来定义一个函数。

//Int类型声明也可以省略，因为Scala具有自动推断类型的功能

val num: Int = 5

也可以按照上面类似的形式来定义Scala中的函数：

val counter: Int => Int = { (value) => value += 1 }

从上面可以看出，在Scala中，函数已经是“头等公民”，单独剥离出来了“值”的概念，一个函数“值”就是函数字面量。这样，只要在某个需要声明函数的地方声明一个函数类型，在调用的时候传一个对应的函数字面量即可，和使用普通变量一模一样。

### 5.2 占位符语法

占位符语法是函数字面量的一种缩写形式。可以使用下划线作为一个或多个参数的占位符，只要每个参数在函数字面量内仅出现一次。占位符语法在处理数据结构和集合时尤其有帮助。很多核心的排序、过滤和其他数据结构方法都会使用占位符语法来减少调用这些方法所需的额外代码。

下面是一个示例（在Scala解释器中运行，从而可以立即看到运行结果）：

//定义列表numList

scala> val numList = List(-3, -5, 1, 6, 9)

numList: List[Int] = List(-3, -5, 1, 6, 9)

//不使用占位符作列表过滤操作

scala> numList.filter(x => x > 0 )

res1: List[Int] = List(1, 6, 9)

//使用占位符做列表过滤操作

scala> numList.filter(\_ > 0)

res2: List[Int] = List(1, 6, 9)

从上面运行结果可以看出，x => x>0与\_ > 0两个函数字面量是等价的。当采用下划线的表示方法时，对于列表numList中的每个元素，都会依次传入用来替换下划线，比如，首先传入-3，然后判断-3>0是否成立，如果成立，就把该值放入结果集合，如果不成立，则舍弃，接着再传入-5，然后判断-5>0是否成立，依此类推。

另外，占位符的顺序对于执行结果也有影响。下面通过一个示例来说明占位符的顺序有什么影响，这个示例使用了两个占位符：

scala>def combination(x:Int,y:Int,f:(Int,Int)=>Int)=f(x,y)

combination:(x:Int,y:Int,f:(Int,Int)=>Int)Int

scala>combination(23,12,\_\*\_)

res3:Int = 276

上面函数是一个匿名函数，在之后章节会进行具体介绍。这里使用了两个占位符，它们会按位置替换输入的参数(分别是x和y)。如果在这里使用一个额外的占位符，将会导致一个错误，因为占位符数必须与输入参数个数一致。下面进行多个占位符的使用，以三个占位符为例，请见下面代码：

scala>def tripleOp(a:Int,b:Int,c:Int,f:(Int,Int,Int)=>Int) =f(a,b,c)

tripleOp:(a:Int,b:Int,c:Int,f:(Int,Int,Int)=>Int) Int

scala>tripleOp(23,92,14,\_\*\_+\_)

res4: Int = 2130

tripleOp函数有4个参数：3个Int值和1个函数，这个函数可以把这3个Int值规约为1个Int。具体的函数体比参数表还要短得多，将把这个函数应用到输入值。

这个示例函数tripleOp仅适用于整型值。如果它是通用的并支持类型参数，可能会更有用。下面使用两个类型参数重新定义tripleOp函数，一个表示通用的输入类型，另一个表示返回值类型。这可以提供灵活性，可以使用任何类型的输入或者选择的匿名函数(只要这个匿名函数有3个输入)来调用tripleOp函数：

scala>def tripleOp[A,B](a:A,b:A,c:A,f(A,A,A)=>B)=f(a,b,c)

tripleOp:[A,B](a:A,b:A,c:A,f(A,A,A)=>B)B

scala>tripleOp[Int,Int](23,92,14,\_\*\_+\_)

res5: Int =2130

scala>tripleOp[Int,Double](23,92,14,1.0\*\_ /\_ /\_)

res6: Double =0.017857142857142856

scala>tripleOp[Int,Boolean](23,92,14,\_>\_+\_)

res7: Boolean = false

这个语法有点难以理解。不过，读完这一章的内容，你会习惯占位符。

### 5.3 递归函数

递归在纯函数编程中起着重要的作用，Scala支持递归函数。递归意味着一个函数可以重复调用自己。

1)递归

在函数式编程中，递归比在命令式编程中更为重要。递归是实现“循环”的唯一方法，因为无法改变循环变量。阶乘的计算就是递归的一个很好的例子，它通过递归计算传递的数字的阶乘。具体代码如下：

object Demo {

def main(args: Array[String]) {

for (i <- 1 to 10)

println( "Factorial of " + i + ": = " + factorial(i) )

}

def factorial(n: BigInt): BigInt = {

if (n <= 1)

1

else

n \* factorial(n - 1)

}

}

运行结果如下：

Factorial of 1: = 1

Factorial of 2: = 2

Factorial of 3: = 6

Factorial of 4: = 24

Factorial of 5: = 120

Factorial of 6: = 720

Factorial of 7: = 5040

Factorial of 8: = 40320

Factorial of 9: = 362880

Factorial of 10: = 3628800

递归是表达函数的最常用方式，然而，递归也有两个缺点：一是反复调用函数带来的开销；二是存在栈溢出的风险。

1. 尾递归

有一种特殊的递归被称为尾递归。在尾递归中，函数可以调用自身，并且该调用是函数的最后一个(“尾部”)操作。尾递归非常重要，因为这是能把函数优化为循环的最重要的一种递归。循环可以消除潜在的栈溢出风险，同时也因为消除了函数调用开销而提升效率。尾递归函数中所有递归形式的调用都出现在函数的末尾，当编译器检测到一个函数调用是尾递归时，会覆盖当前的活动记录而不是在栈中去创建一个新的。下面是尾递归的一个示例。

object Demo {

def main(args: Array[String]) {

//求5的阶乘

factorial(5,1)

//尾递归求阶乘

@annotation.tailrec //告诉编译器要尾递归

def factorial(n:Int,m:Int):Int={

if(n<=0) m

else factorial(n-1,m\*n)

}

}

}

运行结果如下：

120

### 5.4 嵌套函数

函数是命名的参数化表达式块，而表达式块是可以嵌套的，所以函数本身也是可以嵌套的。有些情况下，需要在一个函数中重复某个逻辑，但是把它作为一个外部方法有没有太大意义。而Scala允许定义函数内的函数，函数内定义的函数称为内部函数，这个内部函数只能在该函数中使用。下面以一个阶乘计算器的实现为例。

object Demo {

def main(args: Array[String]) {

println( factorial(3) )

}

//计算阶乘

def factorial(i: Int): Int = {

def fact(i: Int, accumulator: Int): Int = {

if (i <= 1)

accumulator

else

fact(i - 1, i \* accumulator)

}

fact(i, 1)//调用内部函数

}

}

运行结果如下：

1

1

2

6

在这里factorial( Int)是外部函数，fact(Int, Int)是嵌套函数。嵌套函数中的逻辑只定义了一次，不过在外部函数中使用了三次，这就可以减少重复的逻辑，简化整个函数。这里嵌套函数与外部函数可以同名，由于它们的参数不同(外部函数只有一个整型参数)，所以不会发生冲突。Scala函数按函数名以及其参数类型列表来区分。不过，即使函数名和参数类型相同，它们也不会冲突，因为嵌套函数优先于外部函数。

### 5.5 高阶函数

1. **高阶函数定义**

Scala允许定义高阶函数。这些函数将其他函数作为参数，或者其结果是一个函数。尝试下面的示例程序，apply()函数接受另一个函数f和一个值v，并将函数f应用于v。

object Demo {

def main(args: Array[String]) {

println( apply( layout, 10) )

}

def apply(f: Int => String, v: Int) = f(v)

def layout[A](x: A) = "[" + x.toString() + "]"

}

运行结果如下：

[10]

上述代码中layout函数作为参数传到apply函数中，对应函数中的f。数字10则传给整型参数v。相当于执行了layout(10)。高阶函数除了能够将函数作为函数参数，还可以将函数作为返回值，例如：

scala> def layout(factor:Int):Double=>Double={

(x:Double)=>factor\*x

}

layout:(factor:Int)Double => Double

代码def layout(factor:Int):Double=>Double定义了一个函数，该函数输入参数类型为Int类型，返回值为函数类型Double=>Double，返回的函数类型输入参数为Double,返回值类型为Double,代码(x:Double)=>factor\*x为该函数的返回值。同一般的类型一样，Scala也可以进行函数类型推导。

1. **匿名函数与Lambda表达式**

在Scala语言中，不需要给函数命名的函数称为匿名函数。匿名函数的语法构成包含括号，命名参数列表。右箭头及函数体。例如，定义一个匿名函数(x:Int)=>x+1;此匿名函数将它的值加上1；在Scala中，匿名函数作为函数字面量可以赋值给变量。

(x:Int)=>x+1

上面这种匿名函数的定义形式，经常称为“Lambda表达式”。“Lambda表达式”的形式如下：

(参数) => 表达式 //如果参数只有一个，参数的圆括号可以省略

下面以具体示例来了解匿名函数的用法。

//把匿名函数定义为一个值，赋值给myNumFunc变量

scala> val myNumFunc: Int=>Int = (num: Int) => num \* 2

myNumFunc: Int => Int = <function1>

//调用myNumFunc函数，给出参数值3，得到乘法结果是6

scala> println(myNumFunc(3))

6

实际上，Scala具有类型推断机制，可以自动推断变量类型。所以可以省略函数的类型声明，也就是去掉“Int=>Int”。但是不能同时省略参数的类型声明，比如：

val myNumFunc= (num) => num \* 2

否则会报错。因为全部省略以后，实际上，解释器也无法推断出类型，所以需要提供类型声明。下面尝试一下，省略num的类型声明，但是，给出myNumFunc的类型声明，在Scala解释器中的执行过程如下：

scala> val myNumFunc: Int=>Int = (num) => num \* 2

myNumFunc: Int => Int = <function1>

可以看出，顺利运行通过，不会报错，因为，给出了myNumFunc的类型为“Int=>Int”以后，解释器可以推断出num类型为Int类型。

**3)闭包**

在Scala中，任何带有自由变量的函数字面量，需要先明确自由变量的值，只有在关闭这个自由变量开放项的前提下，函数才会运行，计算出结果，称函数为闭包。闭包是一个函数，一种比较特殊的函数。闭包是由函数和运行时的数据决定的。事实上，闭包可以理解为函数和上下文，例如：

scala>var i=15

i：Int=15

//定义一个函数字面量f,函数中使用了前面定义的变量i

scala>var f=(x:Int)=>x+i

f：Int=>Int=<function1>

//执行函数

scala>f(10)

res0：Int=25

//变量重新赋值

scala>i=20

i：Int=20

//执行函数

scala>f(10)

res1：Int=30

代码var f=(x:Int)=>x+i定义了一个函数字面量，函数中使用了自由变量i，变量i在程序的运行过程中会发生变化，在函数执行时如调用f(10)时会根据运行时变量i的值的不同，得到不同的运行结果。自由变量i在运行过程中会不断的发生变化，它处于一种开放状态，而当函数执行时。自由变量i的值已经被确定下来，此时可以认为在运行时它暂时处于封闭状态，这种存在从开放到封闭过程的函数被称为闭包。函数字面量var f=(x:Int)=>x+i中便是函数（f）+上下文（自由变量i）的结合。

### 5.6 高阶函数的使用

通过上一小节，已经清楚了高阶函数的定义及使用方法，本节将介绍集合中常见高阶函数的使用。

1) map操作

map操作是针对集合的典型变换操作，它将某个函数应用到集合中的每个元素，并产生一个结果集合。下面分别介绍Array类型、List类型及Map类型的map函数使用示例，如下所示。

//Array类型的map函数使用

scala> Array("Hadoop", "Hive", "HDFS").map(\_\*2)//字符串加倍

res0: Array[String] = Array(HadoopHadoop, HiveHive, HDFSHDFS)

//List类型的map函数使用

scala> val list = List("Hadoop"->1, "Hive"->2, "HDFS"->2)

list: List[(String,Int)] = List((Hadoop,1), (Hive,2), (HDFS,2))

//省略值函数的输入参数类型

scala> list.map(x=>x.\_1)

res1: List[String] = List(Hadoop, Hive, HDFS)

//参数x在=>中只出现一次，进一步简化

scala> list.map(\_.\_1)

res2: List[String] = List(Hadoop, Hive, HDFS)

//Map类型的map函数使用

scala>Map("Hadoop"->1, "Hive"->2, "HDFS"->3).map(\_.\_1)

res3: scala.collection.immutable.Iterable[String] = List(Hadoop, Hive, HDFS)

scala>Map("Hadoop"->1, "Hive"->2, "HDFS"->2).map(\_.\_2)

res4: scala.collection.immutable.Iterable[Int] = List(1,2,3)

1. flatMap操作

flatMap是map的一种扩展。在flatMap中，会传入一个函数，该函数对每个输入都会返回一个集合（而不是一个元素），然后，flatMap把生成的多个集合“拍扁”成为一个集合。

下面在Scala解释器中执行下面代码（下面同时包含了解释器运行代码的反馈结果）：

scala> val books = List("Hadoop","Hive","HDFS")

books: List[String] = List(Hadoop, Hive, HDFS)

scala> books.flatMap (s => s.toList)

res0: List[Char] = List(H, a, d,o, o, p, H, i, v, e, H, D, F, S)

上面的flatMap执行时，会把books中的每个元素都调用toList，生成List[Char]，最终，多个Char的集合被“拍扁”成一个集合。

3) filter操作

在实际编程中，经常会用到一种操作，遍历一个集合并从中获取满足指定条件的元素组成一个新的集合。Scala中可以通过filter操作来实现。

scala> val listInt=List(1,4,6,8,6,9,12)

listInt:List[Int]=List(1,4,6,8,6,9,12)

//返回所有偶数元素构成的集合

scala>listInt.filter(x=>x%2==0)

res0:List[Int]=List(4,6,8,12)

//简化的写法

scala>listInt.filter(\_%2==0)

res1:List[Int]=List(4,6,8,12)

Array等类型的集合中filter方法使用也类似，这里不再一一赘述。

4) reduce操作

在Scala中，可以使用reduce这种二元操作对集合中的元素进行归约。reduce包含reduceLeft和reduceRight两种操作，前者从集合的头部开始操作，后者从集合的尾部开始操作。在Scala解释器中运行代码。

scala> val list = List(1,2,3,4,5)

list: List[Int] = List(1, 2, 3, 4, 5)

scala> list.reduceLeft((x:Int,y:Int)=>{println(x,y);x+y})

(1,2)

(3,3)

(6,4)

(10,5)

res0: Int = 15

scala> list.reduceRight((x:Int,y:Int)=>{println(x,y);x+y})

(4,5)

(3,9)

(2,12)

(1,14)

res1: Int = 15

可以看出，reduceLeft和reduceRight都是针对两两元素进行操作。上面是加法操作，看不出结果的区别，下面可以用减法操作，reduceLeft和reduceRight就会得到不同的结果：

scala> val list = List(1,2,3,4,5)

list: List[Int] = List(1, 2, 3, 4, 5)

scala> list.reduceLeft((x:Int,y:Int)=>{println(x,y);x-y})

(1,2)

(-1,3)

(-4,4)

(-8,5)

res2: Int = -13

scala> list.reduceRight((x:Int,y:Int)=>{println(x,y);x-y})

(4,5)

(3,-1)

(2,4)

(1,-2)

res3: Int = 3

实际上，可以直接使用reduce，而不用reduceLeft和reduceRight，这时，默认采用的是reduceLeft。

5) fold操作

折叠(fold)操作和reduce（归约）操作比较类似。fold操作需要从一个初始值开始，并以该值作为上下文，处理集合中的每个元素。下面在Scala解释器中运行代码。

scala> val list = List(1,2,4,3,5)

list: List[Int] = List(1, 2, 4, 3, 5)

scala> list.fold(10)(\_\*\_)

res0: Int = 1200

可以看出，fold函数实现了对list中所有元素的累乘操作。fold函数需要两个参数，一个参数是初始种子值，这里是10，另一个参数是用于计算结果的累计函数，这里是累乘。执行list.fold(10)(\*)时，首先把初始值拿去和list中的第一个值1做乘法操作，得到累乘值10，然后再拿这个累乘值10去和list中的第2个值2做乘法操作，得到累乘值20，依此类推，一直得到最终的累乘结果1200。

fold有两个变体：foldLeft()和foldRight()，其中，foldLeft()，第一个参数为累计值，集合遍历的方向是从左到右。foldRight()，第二个参数为累计值，集合遍历的方向是从右到左。对于fold()自身而言，遍历的顺序是未定义的，不过，一般都是从左到右遍历。

scala> list.reduceLeft(0)((x:Int,y:Int)=>{println(x,y);x+y})

(0,1)

(1,2)

(3,4)

(7,3)

(10,5)

res1: Int = 15

scala> list.reduceRight(0)((x:Int,y:Int)=>{println(x,y);x+y})

(5,0)

(3,5)

(4,8)

(2,12)

(1,14)

res2: Int = 3

# Scala面向对象编程

Scala是一门纯面向对象编程语言，Scala中的一切变量都是对象，一切操作都是方法调用。在本章将对Scala中类的定义、对象的创建、构造函数的定义与使用、继承、类成员访问控制、抽象类等内容进行介绍。

## 1 类与对象基础

类和对象是面向对象编程的核心内容。本节将详细介绍如何定义类，包括伴生类、抽象类、匿名类，如何创建对象，包括单例对象、应用程序对；类成员的访问等相关内容。

### 1.1 类定义

Scala定义类最简单的形式和Java很相似，如定义class Student类：

class Student{

//声明成员变量，这里成员变量必须初始化

private var age = 0 //私有成员变量、初始化为0

val name=”Scala” //定义属性name

def increment(): Unit = { age+= 1} //方法默认是公有的

def current(): Int = {age}

}

在上面定义中，把age成员变量设置为private，这样它就成为私有成员变量，外界无法访问，只有在类内部可以访问该成员变量。如果成员变量前面什么修饰符都没有，就默认是public，外部可以访问该成员变量。对于类而言，并不需要声明为public，Scala文件中包含的多个类之间，都是彼此可见的。

对于方法的定义，是通过def实现的。上面的代码“def increment(): Unit = { age+= 1}”中，increment()是方法，没有参数，冒号后面的Unit是表示返回值的类型，在Scala中不返回任何值，那么就用Unit表示，相当于Java中的void类型。方法的返回值，不需要靠return语句，方法里面的最后一个表达式的值就是方法的返回值，比如，上面current()方法里面只有一条语句“age”，那么，age的值就是该方法的返回值。

因为increment()方法只是对age的值进行了增加1的操作，并没有返回任何值，所以，返回值类型是Unit。Unit后面的等号和大括号后面，包含了该方法要执行的具体操作语句。如果大括号里面只有一行语句，那么也可以直接去掉大括号，例如：

def increment(): Unit = value += 1

或者，还可以去掉返回值类型和等号，只保留大括号，如下：

def increment() {value += 1} //去掉了返回值类型和等号，只保留大括号

### 1.2 创建对象

下面新建对象，并调用其中的方法：

val student= new Student

student.increment() //或者也可以不用圆括号，写成student.increment

println(student.current)

从上面代码可以看出，Scala在调用无参方法时，是可以省略方法名后面的圆括号的。下面在Demo.scala文件中执行相关代码：

class Student{

private var age = 18

val name=”Scala”

def increment(): Unit = { age+= 1}

def current(): Int = {age}

}

object Demo {

def main(args: Array[String]) {

val student= new Student

student.increment() //或者也可以不用圆括号，写成student.increment

println(student.current)

}

}

### 

运行结果如下：

19

### 1.3 类成员的访问

创建Student类对象后考虑如何给类中的成员变量设置值以及读取值。在Java中，这是通过getter和setter方法实现的。在Scala中，也提供了getter和setter方法的实现，但是并没有定义成getXxx和setXxx,而是类似于Xxx和Xxx\_的定义，理解为Xxx就是getXxx方法，Xxx\_就是setXxx方法，和Java中的用意一样。

Java中getter和setter方法的定义如下：

public class Student{

private int age;

public int getAge(){return age;}//get方法

public void setAge(int age){this.age=age;}//set方法

}

Scala中属性的定义，属性中带有getter和setter方法：

class Counter {

private var privateAge= 0 //私有成员变量

def age = privateAge //定义一个方法，方法的名称就是原来想要的成员变量的名称

def age\_=(newAge: Int){

if (newAge>18) privateAge = newAge//只有提供的新值大于18，才允许修改

}

}

Scala中对每个成员变量都提供了getter和setter方法，在代码中不用显性定义，但在Scala编译字节码中实际生成了getter和setter方法，即age和age\_的定义。

### 1.4 构造函数

构造函数，是一种特殊的方法。主要用来在创建对象时初始化对象， 即为对象成员变量赋初始值，总与new运算符一起使用在创建对象的语句中。

1) 辅助构造函数

首先认识一下辅助构造函数。辅助构造函数的名称为this，每个辅助构造函数都必须调用一个此前已经定义的辅助构造函数或主构造函数。

下面定义一个带有辅助构造函数的类，对上面的Student类定义进行修改：

class Student {

private var age= 18//age用来存储学生的年纪

private var name = "" //表示学生的名字

private var classNum = 1 //ClassNum用来表示学生的班级

def this(name: String){ //第一个辅助构造函数

this() //调用主构造函数

this.name = name

}

def this (name: String, classNum: Int){ //第二个辅助构造函数

this(name) //调用前一个辅助构造函数

this.classNum = classNum

}

def increment(step: Int): Unit = { age += step} //增加年龄

def current(): Int = {age}

def info(): Unit = {printf("Name:%s and classNum is %d\n",name,classNum)}

}

object MyStudent{

def main(args:Array[String]){

val myStudent1 = new Student //主构造函数

val myStudent2 = new Student("ZhangSan") //第一个辅助构造函数，学生名字设 置为 ZhangSan

val myStudent3 = new Student("LiSi",75) //第二个辅助构造函数，学生名字设置为 LiSi，班级为75班

myStudent1.info //显示学生信息

myStudent1.increment(1) //设置步长

printf("Current age is: %d\n",myStudent1.current) //显示学生年纪

myStudent2.info //显示学生信息

myStudent2.increment(2) //设置步长

printf("Current age is: %d\n",myStudent2.current) //显示学生年纪

myStudent3.info //显示学生信息

myStudent3.increment(3) //设置步长

printf("Current age is: %d\n",myStudent3.current) //显示学生年纪

}

}

编译执行上述代码后，得到如下结果：

Name: and classNum is 1

Current Value is: 19

Name:ZhangSanand classNum is 1

Current Value is: 20

Name:LiSi and classNum is 75

Current Value is: 21

2)主构造函数

Scala的每个类都有主构造函数。但是，Scala的主构造函数和Java有着明显的不同，Scala的主构造函数是整个类体，需要在类名称后面罗列出构造函数所需的所有参数，这些参数被编译成成员变量，成员变量的值就是创建对象时传入的参数的值。

主构造函数与类定义在一起，主构造函数的参数直接放在类名之后。主构造函数的参数加val或var时自动升级为成员变量，其值被初始化成构造函数时传入的参数。主构造函数在类中除了方法以外，会执行类定义中的所有语句

对于上面给学生设置name和classNum的例子，是使用辅助构造函数来对name和classNum的值进行设置，现在采用主构造函数来设置name和classNum的值。

class Student(val name: String, val classNum: Int) {

private var age = 18 //value用来存储学生年纪的起始值

def increment(step: Int): Unit = {age += step}

def current(): Int = {age}

def info(): Unit = {printf("Name:%s and classNum is %d\n",name,classNum)}

}

object MyStudent{

def main(args:Array[String]){

val myStudent = new Student("ZhangSan",67)

myStudent.info //显示学生信息

myStudent.increment(1) //设置步长

printf("Current age is: %d\n",myStudent.current) //显示学生年纪

}

}

编译执行上述代码后，得到如下结果：

Name:ZhangSan and classNum is 67

Current age is: 19

### 1.5 常见对象类型

Scala中常见的对象类型包括单例对象、应用程序对象以及伴生对象。下面分别就对三种对象类型进行详细介绍。

1)单例对象

Scala比Java更面向对象，因为Scala中没有静态方法和静态成员变量，Scala中使用object关键字来实现同样的效果。单例是一个类，它只能有一个实例，即对象。由于不能实例化单例对象，所以不能将参数传递给主构造函数。object对象定义了某个类的单个示例，如：

//通过object关键字定义单例对象University

scala>object University{

private var studentNum=0;

def newStuNum={//定义newStuNum方法，将学号加1，返回新的学号studentNum

studentNum+=1

studentNum

}

}

defined object University

//直接通过单例对象名称访问其成员方法

scala> University.newStuNum

res0: Int = 1

//单例对象中的成员变量状态随程序执行而改变

scala> University.newStuNum

res1: Int = 2

实际上Scala单例对象是通过Java语言的单例模式和静态类成员来实现的。

1. 应用程序对象

在Java语言中，只有定义了public static void main(String[] args)方法，该类便可以成为程序执行的人口。在Scala语言中同样使用main函数作为程序的执行入口，只不过main函数必须定义在单例对象中。在单例对象中定义了main方法，该单例对象便称为应用程序对象。在Demo.Scala中写入示例如下：

object University{

private var studentNum=0;

def newStuNum={//定义newStuNum方法，将学号加1，返回新的学号studentNum

studentNum+=1

studentNum

}

//通过main方法作为程序的入口

def main(args:Array[String]){

println(“New num is ”+University.newStuNum)

}

}

在命令行输入：

scalac Demo.scala//编译的时候使用的是Scala文件名称

scala -classpath . University //执行的时候使用的是University对象名称

运行结果如下：

New num is 1

1. 伴生对象

在Java中，经常需要用到同时包含示例方法和静态方法的类，在Scala中可以通过伴生对象来实现。当单例对象与某个类具有相同的名称时，它被称为这个类的“伴生对象”。类和它的伴生对象必须存在于同一个文件中，而且可以相互访问私有成员（成员变量和方法）。下面通过一个示例演示一下伴生对象的使用方法。

定义类class University,类class University拥有自己的属性id，number(id直接调用伴生对象的newStuId方法赋值)，类class University拥有自己的方法aClass;同时定义类class University的伴生对象object University，伴生对象object University拥有自己的属性学号StuId,及新增一个学号的方法newStuId,返回新学号。伴生对象相当于Java中的静态类，无须示例化，可以直接调用伴生对象的方法。

class University{//定义类University

val id = University.newStuId() //调用了伴生对象中的方法

private var number =0

def aClass(number: Int) { this.number= number}

}

object University {//定义类University的伴生对象object University

private var StuId = 0 //学号

def newStuId() = {

StuId +=1

StuId

}

}

Object ObjectOps{

def main(args: Array[String]){

println(University.newStuId)//直接调用伴生对象University的方法newStuId

println(University.newStuId)//再次调用newStuId方法，学号StuId在加1

}

}

运行结果：

1

2

### 1.6 抽象类与匿名类

1. 抽象类

抽象类是一种不能被实例化的类，抽象类中存在抽象成员变量或成员方法，这些成员方法或成员变量在子类中被具体化。在Scala中，通过abstract关键字定义抽象类。Scala语言中的抽象类不仅可以有抽象方法，还可以有抽象成员变量。Scala中一般的成员变量在定义时必须初始化。示例如下：

//普通类成员变量在定义时必须显式初始化

scala> class Phone{

var phoneBrand: String=\_

}

defined class Phone

//不显式初始化成员变量会报错，提示Phone类应该被声明为abstract

scala> class Phone{

var phoneBrand: String

}

<console>:11: error: class Phone needs to be abstract, since variable phoneBrand is not defined

(Note that variables need to be initialized to be defined)

class Phone{

^

可以看到，当普通类的成员变量不显式地对其进行初始化时会报错并提示类应该被声明为abstract,这说明Scala语言中抽象类可以的抽象成员变量(也叫抽象成员变量)。

//成员变量如果不显式初始化，则将类声明为抽象类，通过abstract关键字来定义

scala> abstract class Phone{

| var phoneBrand: String

| }

defined class Phone

Scala中的抽象类中除了抽象方法还可以定义相应的具体方法及具体的成员变量，例如：

scala> abstract class Phone{

| var phoneBrand: String//抽象类中的抽象成员变量

| var price:Int=0//抽象类中的具体成员变量

| def info()//抽象类中的抽象方法

| def greeting() =println("Welcome to use phone!")//抽象类中的具体方法

| }

defined class Phone

1. 匿名类

匿名类，顾名思义就是没有名字的类。当某个类在代码中仅使用一次时，可以考虑使用匿名类，示例如下：

//定义抽象类

scala> abstract class Phone(var phoneBrand:String,var price:Int){

| def info:Unit

| }

defined class Phone

//使用匿名类，并创建匿名类对象

scala> val p=new Phone("HuaWei",4100){

//重写抽象内中的抽象方法

| override def info:Unit=println(s"Phone($phoneBrand,$price)")

| }

p: Phone = $anon$1@225e09f0

scala> p.info

Phone(HuaWei,4100)

上述匿名类创建对象的方式等同于下面的代码：

class NamedClass( phoneBrand:String,price:Int) extends Phone( phoneBrand,price){

override def info:Unit=println(s"Phone($phoneBrand,$price)")

}

val p=new NamedClass(“HuaWei”,4100)

只不过，命名类NamedClass一旦被定义就可以反复使用，而匿名类只使用一次，代码更简洁。

## 2 继承与多态

继承是面向对象编程语言实现代码复用的关键特性，它是从一般到特殊的过程，指的是在原有类的基础上定义一个新的类，原有类称为父类，新的类称为子类。实现继承后，子类可以拥有父类的属性和方法，也可以在子类中添加新的类成员，同时又可以根据子类具体的行为重写父类的方法。多态是在继承的基础上实现的一种语言特性，它指的是允许不同类的对象对同一消息做出响应，即同一消息可以根据发送对象的不同而采用多种不同的行为方式。本章将对Scala语言中的继承和多态进行基础介绍。

### 2.1 类的继承

Scala语言同Java语言一样，也是通过extends关键字来实现类间的继承。但是Scala中的继承与Java有着显著的不同：

（1）重写一个非抽象方法必须使用override修饰符。

（2）只有主构造函数可以调用父类的主构造函数。

（3）在子类中重写父类的抽象方法时，不需要使用override关键字。

（4）可以重写父类中的成员变量。

Scala和Java一样，不允许类从多个父类继承，因此，只讨论继承自一个类的情形。

1. 抽象类的继承

以手机为例子，首先创建一个抽象类，让这个抽象类被其他类继承。

abstract class Phone{ //是抽象类，不能直接被示例化

val phoneBrand: String //成员变量没有初始化值，就是一个抽象成员变量

def info() //抽象方法，不需要使用abstract关键字

def greeting() {println("Welcome to use phone!")}

}

关于上面的定义，说明几点：

（1）定义一个抽象类，需要使用关键字abstract。

（2）定义一个抽象类的抽象方法，也不需要关键字abstract，只要把方法体空着，不写方法体就可以。

（3）抽象类中定义的成员变量，只要没有给出初始化值，就表示是一个抽象成员变量，但是，抽象成员变量必须要声明类型，比如：val phoneBrand: String，就把phoneBrand声明为字符串类型，这个时候，不能省略类型，否则编译会报错。

当子类继承抽象类时，需要在子类中对父类中的抽象成员变量进行初始化，否则子类也必须声明为抽象类。

/\*父类中包含抽象成员变量时，子类如果为普通类则必须将该成员变量初始化，否则子类也应声明为抽象类\*/

scala> class XiaoMi extends Phone

<console>:12: error: class XiaoMi needs to be abstract, since:

it has 2 unimplemented members.

/\*\* As seen from class XiaoMi, the missing signatures are as follows.

\* For convenience, these are usable as stub implementations.

\*/

def info(): Unit = ???

val phoneBrand: String = ???

class XiaoMi extends Phone

^

//在子类中对父类中的抽象成员变量及抽象方法进行初始化，使用override关键字

scala> class Apple extends Phone{

| override val phoneBrand:String="Apple"

| override def info:Unit=println("Welcome to use iphone!")

| }

defined class Apple

//也可以省略override关键字

scala> class Apple extends Phone{

| val phoneBrand:String="Apple"

| def info:Unit=println("Welcome to use iphone!")

| }

defined class Apple

通过上述代码可以看到，如果Phone类中存在抽象成员变量phoneBrand，子类Apple如果没有对该成员变量进行初始化的话，系统会报错并提示应该将该类也定义为抽象，如果不需要子类为抽象类，则需要对该成员变量进行初始化，值得注意的是子类对父类抽象成员变量进行重写可以加override关键字也可以省略。

1.2 扩展类的继承

抽象类不能直接被示例化，所以，下面定义几个扩展类，它们都是扩展了Phone类，或者说继承自Phone类。

class Apple extends Phone{

//重写父类成员变量，可以使用override关键字。

override val phoneBrand = "Apple"

//重写父类的抽象方法时，不需要使用override关键字，不过，如果加上override编译也不错报错

def info() {printf("This is a/an %s phone. It is on sale", phoneBrand)}

//重写父类的非抽象方法，必须使用override关键字

override def greeting() {println("Welcome to use Apple Phone!")}

}

class HuaWei extends Phone {

//重写父类成员变量，需要使用override关键字，否则编译会报错

override val phoneBrand = "HuaWei"

//重写父类的抽象方法时，不需要使用override关键字，不过，如果加上override编译也不错报错

def info() {printf("This is a/an %s phone. It is cheap.", phoneBrand)}

//重写父类的非抽象方法，必须使用override关键字

override def greeting() {println("Welcome to use HuaWei Phone!")}

}

由上述代码可知，子类重写父类的非抽象成员变量、非抽象方法必须使用override关键字，否则会报错。而子类重写父类的抽象成员变量及抽象方法可以不使用override关键字。下面，把上述代码放入一个完整的代码文件Demo.scala，编译运行。

abstract class Phone{

val phoneBrand: String

def info()

def greeting() {println("Welcome to use phone!")}

}

class Apple extends Phone{

override val phoneBrand = "Apple"

def info() {printf("This is a/an %s phone. It is on sale", phoneBrand)}

override def greeting() {println("Welcome to use Apple Phone!")}

}

class HuaWei extends Phone {

override val phoneBrand = "HuaWei"

def info() {printf("This is a/an %s phone. It is cheap.", phoneBrand)}

override def greeting() {println("Welcome to use HuaWei Phone!")}

}

object MyPhone {

def main(args: Array[String]){

val myPhone1 = new Apple()

val myPhone2 = new HuaWei ()

myPhone1.greeting()

myPhone1.info()

myPhone2.greeting()

myPhone2.info()

}

}

保存文件。执行后，屏幕上会显示以下结果：

Welcome to use Apple Phone!

This is a/an Apple phone. It is on sale

Welcome to use HuaWei Phone!

This is a/an HuaWei phone. It is cheap.

### 2.2 构造函数执行顺序

在Java语言中,如果两个类之间存在继承关系，创建子类对象时会先调用父类的构造函数，然后再调用子类的构造函数来完成对象的创建。在Scala语言中 ，创建子类对象时的构造函数执行顺序也是如此。

//定义Phone类，带主构造函数

class Phone(var phoneBrand:String,var price:Int){

//类中执行语句会在调用主构造函数时执行

println(“执行Phone类的主构造函数”)

}

//定义Apple类，继承自Phone类，同样也带主构造函数

class Apple(phoneBrand:String,price:Int)extends Phone(phoneBrand,price){

//类中执行语句会在调用主构造函数时执行

Println(“执行Apple类的主构造函数”)

}

Object Demo{

def main(args:Array[String]){

//创建子类对象时，先调用父类的主构造函数，然后调用子类的主构造函数

new Apple(“iphone”,5400)

}

}

代码执行结果如下：

执行Phone类的主构造函数

执行Apple类的主构造函数

通过执行结果不难看出，在使用代码new Apple(“iphone”,5400)创建对象时，首先会调用Phone类的主构造函数，然后再调用Apple类的主构造函数。

### 2.3 方法重写

方法重写指的是当子类继承父类时，从父类继承过来的方法不能满足子类的需要，子类希望有自己的实现，这时需要对父类的方法进行重写（override）,方法重写是实现多态和动态绑定的关键。Scala语言中的方法重写与Java语言中的方法重写一样，也是通过override关键字对父类中的方法进行重写，从而实现子类自身处理逻辑。例如：

class Phone(var phoneBrand:String,var price:Int){

//对父类Any中的toString方法进行重写

override def toString=s”Phone($phonebrand,$price)”

}

class Apple(phoneBrand:String,price:Int,var place:String)extends Phone(phoneBrand,price){

//对父类Phone中的toString方法进行重写

override def toString=s”Apple($phonebrand,$price,$place)”

}

Object Demo{

def main(args:Array[String]){

//调用Apple类自身的toString方法返回结果

Println(new Apple(“iphone”,5400,”Shenzhen”))

}

}

代码运行结果如下：

Apple(iphone,5400,shenzhen)

如果不重写父类的toString方法，则返回的结果是类名加hashcode值，例如：

scala> class Phone(var phoneBrand:String,var price:Int)

defined class Phone

scala> new Phone("HuaWei",4500)

res1: Phone = Phone@55881f40

通过父类方法的重写可以改变子类中的代码行为，例如：

scala> class Phone(var phoneBrand:String,var price:Int){

| override def toString=s"Phone($phoneBrand,$price)"

| }

defined class Phone

scala> new Phone("HuaWei",4500)

res2: Phone = Phone(HuaWei,4500)

### 2.4 多态

多态也称动态绑定或延迟绑定，指在执行期间而非编译期间确定所引用对象的实际类型，根据其实际类型调用其相应的方法，也就是说子类的引用可以赋给父类，程序在运行时根据实际类型调用相应的方法。示例如下：

//定义父类Phone

class Phone(var phoneBrand:String,var price:Int){

//buy方法，无参数

def buy():Unit=println(“buy() method in Phone”)

//compare方法，参数为Person类型

def compare(p:Phone):Unit=println(“compare() method in Phone”)

}

//定义子类Apple

class Apple( phoneBrand:String,price:Int) extends Phone((phoneBrand,price)){

private var AphoneNo:Int=0

//重写父类compare方法

override def compare(p:Phone):Unit={

println(“compare() method in Apple”)

println(this.phoneBrand+” is compared with ”+p.phoneBrand)

}

}

//定义子类HuaWei

class HuaWei(phoneBrand:String,price:Int) extends Phone((phoneBrand,price)){

private var HphoneNo:Int=0

//重写父类buy方法

override def buy():Unit=println(“buy() method in HuaWei”)

//重写父类compare方法

override def compare(p:Phone):Unit={

println(“compare() method in HuaWei”)

Println(this.phoneBrand+” is compared with ”+p.phoneBrand)

}

}

//运行入口

Object Demo{

def main(args: Array[String]){

val p1:Phone=new HuaWei(“huawei”,4500)

val p2:Phone=new Apple(“iphone”,6400)

//p1实际上引用的是HuaWei类型的对象，HuaWei类对父类中buy方法进行了重写，因此它调用的是重写后的方法

p1.buy()

//compare方法参数类型为Phone，调用的是HuaWei类重写后的compare方法

p1.compare(p2)

Println(“////////////////////”)

//p2引用的是Apple类型的对象，Apple类未对父类中的buy方法进行重写

//因此它调用的是继承自父类的buy方法

p2.buy()

//p2.talkto(p1)传入的实际类型是HuaWei，调用的是Apple类重写后的compare方法

p2.compare(p1)

}

}

示例运行结果如下：

buy() method in HuaWei

compare() method in HuaWei

huawei is compared with iphone

/////////////////////////

buy() method in Phone

compare() method in Apple

Iphone is compared with huawei

## 3 特质

Java中提供了接口，允许一个类实现任意数量的接口。在Scala中没有接口的概念，而是提供了“特质(trait)”，它不仅实现了接口的功能，还具备了很多其他的特性。Scala的特质，是代码重用的基本单元，可以同时拥有抽象方法和具体方法。Scala中，一个类只能继承自一个父类，却可以实现多个特质，从而重用特质中的方法和成员变量，实现了多重继承。

### [3.1 特质的使用](#_Toc22392)

1. 特质的定义

特质的定义和类的定义非常相似，相当于Java中的接口。有区别的是，特质定义使用关键字trait。

trait PhoneId{

var id: Int

def currentId(): Int //定义了一个抽象方法

}

上面定义了一个特质，里面包含一个抽象字段id和抽象方法currentId。注意，抽象方法不需要使用abstract关键字，特质中没有方法体的方法，默认就是抽象方法。

1. 把特质混入类中

特质定义好以后，就可以使用extends或with关键字把特质混入类中。

class ApplePhoneId extends PhoneId{ //使用extends关键字

override var id = 10000 //Apple手机编号从10000开始

def currentId(): Int = {id += 1; id} //返回手机编号

}

class HuaWeiPhoneId extends PhoneId{ //使用extends关键字

override var id = 20000 //HuaWei手机编号从20000开始

def currentId(): Int = {id += 1; id} //返回手机编号

}

下面，把上述代码放入一个完整的代码文件Demo.scala，编译运行。 在Demo.scala文件中输入以下内容：

trait PhoneId{

var id: Int

def currentId(): Int //定义了一个抽象方法

}

class ApplePhoneId extends PhoneId{ //使用extends关键字

override var id = 10000 //Apple手机编号从10000开始

def currentId(): Int = {id += 1; id} //返回手机编号

}

class HuaWeiPhoneId extends PhoneId{ //使用extends关键字

override var id = 20000 //HuaWei手机编号从20000开始

def currentId(): Int = {id += 1; id} //返回手机编号

}

object MyPhone {

def main(args: Array[String]){

val myPhone1 = new ApplePhoneId()

val myPhone2 = new HuaWeiPhoneId ()

printf("My first PhoneId is %d.\n",myPhone1.currentId)

printf("My second PhoneId is %d.\n",myPhone2.currentId)

}

}

然后，使用scalac命令编译这个代码文件，并用scala命令执行，如下：

scalac Demo.scala

scala -classpath . MyPhone //MyPhone是代码中object后面的MyPhone

上面命令执行后，会在屏幕输出以下结果：

My first PhoneId is 10001.

My second PhoneId is 20001.

同时特质可以包含具体实现。上面的示例中，特质只包含了抽象字段和抽象方法，相当于实现了类似Java接口的功能。实际上，特质也可以包含具体实现，也就是说，特质中的字段和方法不一定要是抽象的。如：

trait PhoneGreeting{

def greeting(msg: String) {println(msg)}

}

CarGreeting会把欢迎信息打印出来。

1. 把多个特质混入类中

上面已经定义了两个特质，即PhoneId和PhoneGreeting。现在，可以把两个特质都混入到类中。在Demo.scala文件中输入以下内容：

trait PhoneId{

var id: Int

def currentId(): Int //定义了一个抽象方法

}

trait PhoneGreeting{

def greeting(msg: String) {println(msg)}

}

//使用extends关键字混入第1个特质，后面可以反复使用with关键字混入更多特质

class ApplePhoneId extends PhoneId with PhoneGreeting{

override var id = 10000 //Apple手机编号从10000开始

def currentId(): Int = {id += 1; id} //返回手机编号

}

//使用extends关键字混入第1个特质，后面可以反复使用with关键字混入更多特质

class HuaweiPhoneId extends PhoneId with PhoneGreeting{

override var id = 20000 //HuaWei手机编号从10000开始

def currentId(): Int = {id += 1; id} //返回手机编号

}

object MyPhone{

def main(args: Array[String]){

val myPhone1 = new ApplePhoneId()

val myPhone2 = new HuaWeiPhoneId ()

myPhone1.greeting("Welcome my first phone.")

printf("My first PhoneId is %d.\n",myPhone1.currentId)

myPhone2.greeting("Welcome my second phone.")

printf("My second PhoneId is %d.\n",myPhone2.currentId)

}

}

运行结果如下：

Welcome my first phone.

My first PhoneId is 10001.

Welcome my second phone.

My second PhoneId is 20001.

### [3.2 特质与类](#_Toc17777)

1. 特质与类的相似点

特质trait可以像普通类一样，定义成员变量和成员方法，而无论其成员变量与成员方法是具体的还是抽象的，特质trait在抽象程度上更接近于抽象类，例如：

scala> trait Phone{

| println("Phone")

| def phone1(msg:String):Unit

| def phone2(msg:String):Unit=println(msg)

| }

defined trait Phone

scala> abstract class Phone{

| println("Phone")

| def phone1(msg:String):Unit

| def phone2(msg:String):Unit=println(msg)

| }

defined class Phone

代码中定义一个trait Phone和一个抽象类Phone,除定义形式不同外，它们都存在执行语句、抽象方法和具体方法。在使用语法上也有相似之处，普通的类都可以使用extends关键字扩展类或混入trait,例如：

//Phone为trait

scala> trait Phone{

| println("Phone")

| def phone1(msg:String):Unit

| def phone2(msg:String):Unit=println(msg)

| }

defined trait Phone

//使用extends关键字混入Phone

scala> class Apple extends Phone{

| def phone1(msg:String):Unit=println("phone1:"+msg)

| }

defined class Apple

scala> val p=new Apple

Phone

p: Apple = Apple@718ad3a6

scala> p.phone1("Apple extends Phone trait")

phone1:Apple extends Phone trait

上面代码给出的是Phone是trait,Apple使用extends关键字混入Phone,然后实现混入trait中的抽象方法。对于抽象类，普通类同样使用extends关键字实现对该抽象类的继承，例如：

//将Phone定义为抽象类

scala> abstract class Phone{

| println("Phone")

| def phone1(msg:String):Unit

| def phone2(msg:String):Unit=println(msg)

| }

defined class Phone

//通过extends关键字扩展类Phone,实现继承

scala> class Apple extends Phone{

| def phone1(msg:String):Unit=println("phone1:"+msg)

| }

defined class Apple

scala> val p=new Apple

Phone

p: Apple = Apple@7d6019d5

scala> p.phone1("Apple extends abstract class Phone")

phone1:Apple extends abstract class Phone

通过代码不难发现，除了将Phone声明为抽象类之外，和将Phone定义为trai时的代码是相同的。还有一个值得注意的地方就是在定义trait时可以使用extends关键字继承类，例如：

//定义一个普通类

scala> class A{

| val msg:String="msg"

| }

defined class A

//trait B继承自类A

scala> trait B extends A{

| def print()=println(msg)

| }

defined trait B

scala> new B{}.print()

msg

1. 特质与类的不同点

trait与类之间还存在许多不同之处，首先无论是普通类还是抽象类都可以在类定义时使用主构造函数定义类的成员变量，但trait不能，例如：

scala> abstract class Phone(val msg:String)

defined class Phone

scala> trait Phone(val msg:String)

<console>:1: error: traits or objects may not have parameters

trait Phone(val msg:String)

^

通过上述代码可以看到，trait不能有使用主构造函数定义的成员变量，这是trait与类间的一个重要区别。其次，Scala语言中的类不能继承多个类，但可以混入多个trait,例如：

scala> trait A

defined trait A

scala> trait B

defined trait B

//类可以混入多个trait

scala> class C extends A with B

defined class C

### [3.3 多重](#_Toc10458)继承问题

Scala语言中可以通过使用trait实现多重继承，不过在实际使用时常常会遇到菱形继承的问题，如以下代码：

scala> trait A{

| def print:Unit

| }

defined trait A

scala> trait B1 extends A{

| var B1="trait B1"

| override def print=println(B1)

| }

defined trait B1

scala> trait B2 extends A{

| var B2="trait B2"

| override def print=println(B2)

| }

defined trait B2

scala> class C extends B1 with B2

defined class C

scala> val c=new C

c: C = C@6c2f8ecb

//使用的时trait B2中的print方法

scala> c.print

trait B2

上述代码存在菱形继承问题，即trait B1,trait B2中分别混入了trait A,然后类C又混入了B1、B2，这导致类C中会存在两个print方法。为解决方法调用时的冲突问题，Scala会对类进行线性化，在存在多重继承时会使用最右深度优先遍历查找的方法，例如class C extends B1 with B2混入了trait B1和B2，在调用print方法时会采用最右深度优先遍历算法查找，在本例中为B2中的print方法，而B1中的print方法没有被执行。

sscala> trait A{

| val a="Trait A"

| def print(msg:String)=println(msg+":"+a)

| }

defined trait A

scala> trait B1 extends A{

| val b1="Trait B1"

| override def print(msg:String)=super.print(msg+":"+b1)

| }

defined trait B1

scala> trait B2 extends A{

| val b2="Trait B2"

| override def print(msg:String)=super.print(msg+":"+b2)

| }

defined trait B2

scala> class C extends B1 with B2

defined class C

scala> new C().print("print method in")

print method in:Trait B2:Trait B1:Trait A

在上述代码中，在trait A中定义了方法def print(msg:String)=println(msg),然后再trait B1、trait B2中对方法进行重写，只不过在方法实现中使用super关键字进行父类的方法调用，如super.print(msg+”:”+b1)、super.print(msg+”:”+b2)。这种super关键字的使用方式也是一种惰性求值，super关键字调用的方法不会马上执行而是在真正被调用时执行，它的执行原理同样按照最右深度优先遍历算法进行，先将B2、B1、A中的成员变量按序组装得到:Trait B2:Trait B1:Trait A,然后再调用print("print method in":Trait B2:Trait B1:Trait A)得到最终结果，这种方式是解决多重继承菱形问题的最常用方法。