尚硅谷大数据技术之Scala

（作者：尚硅谷研究院）

V4.0

# 第1章 Scala入门

## 1.1 概述

Scala将面向对象和函数式编程结合成一种简洁的高级语言。

语言特点如下：

（1）Scala和Java一样属于JVM语言，使用时都需要先编译为class字节码文件，并且Scala能够直接调用Java的类库。

（2）Scala支持两种编程范式面向对象和函数式编程。

（3）Scala语言更加简洁高效；语法能够化简，函数式编程的思想使代码结构简洁。

（4）作者马丁·奥德斯基设计Scala借鉴了Java的设计思想，同时优秀的设计也推动了Java语言的发展。

## 1.2 Scala环境搭建

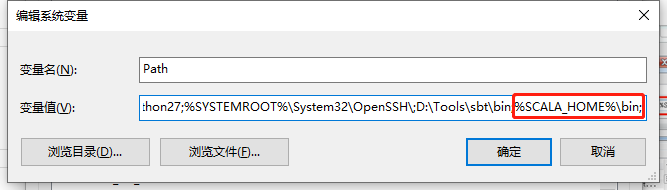
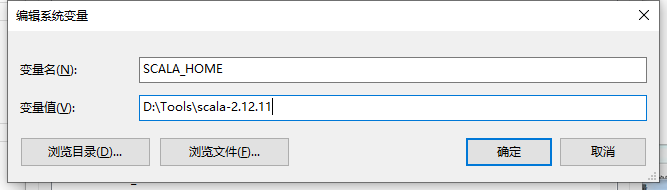
1）安装步骤

（1）首先确保JDK1.8安装成功

（2）下载对应的Scala安装文件scala-2.12.10.zip

（3）解压scala-2.12.10.zip，解压到任意没有中文的路径,例如D:\Tools

（4）配置Scala的环境变量



注意1：解压路径不能有任何中文路径，最好不要有空格。

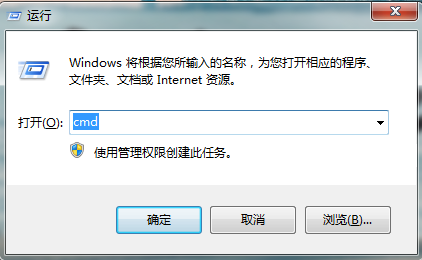
注意2：环境变量要大写SCALA\_HOME。

2）测试

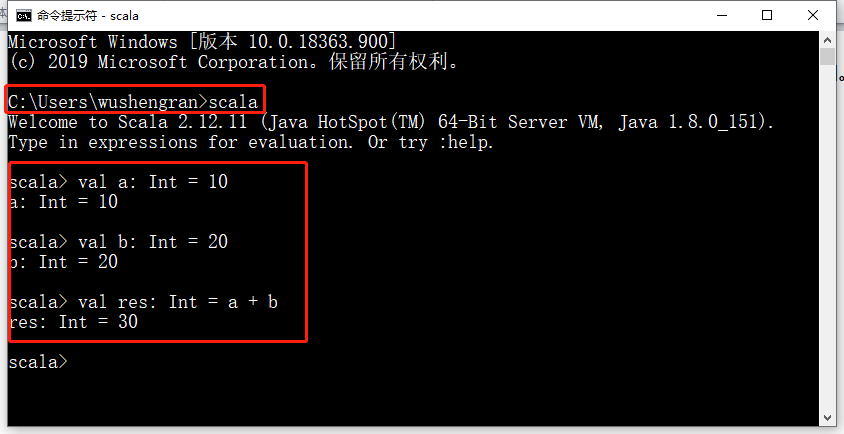
需求：计算两数a和b的和。

步骤

（1）在键盘上同时按win+r键，并在运行窗口输入cmd命令。



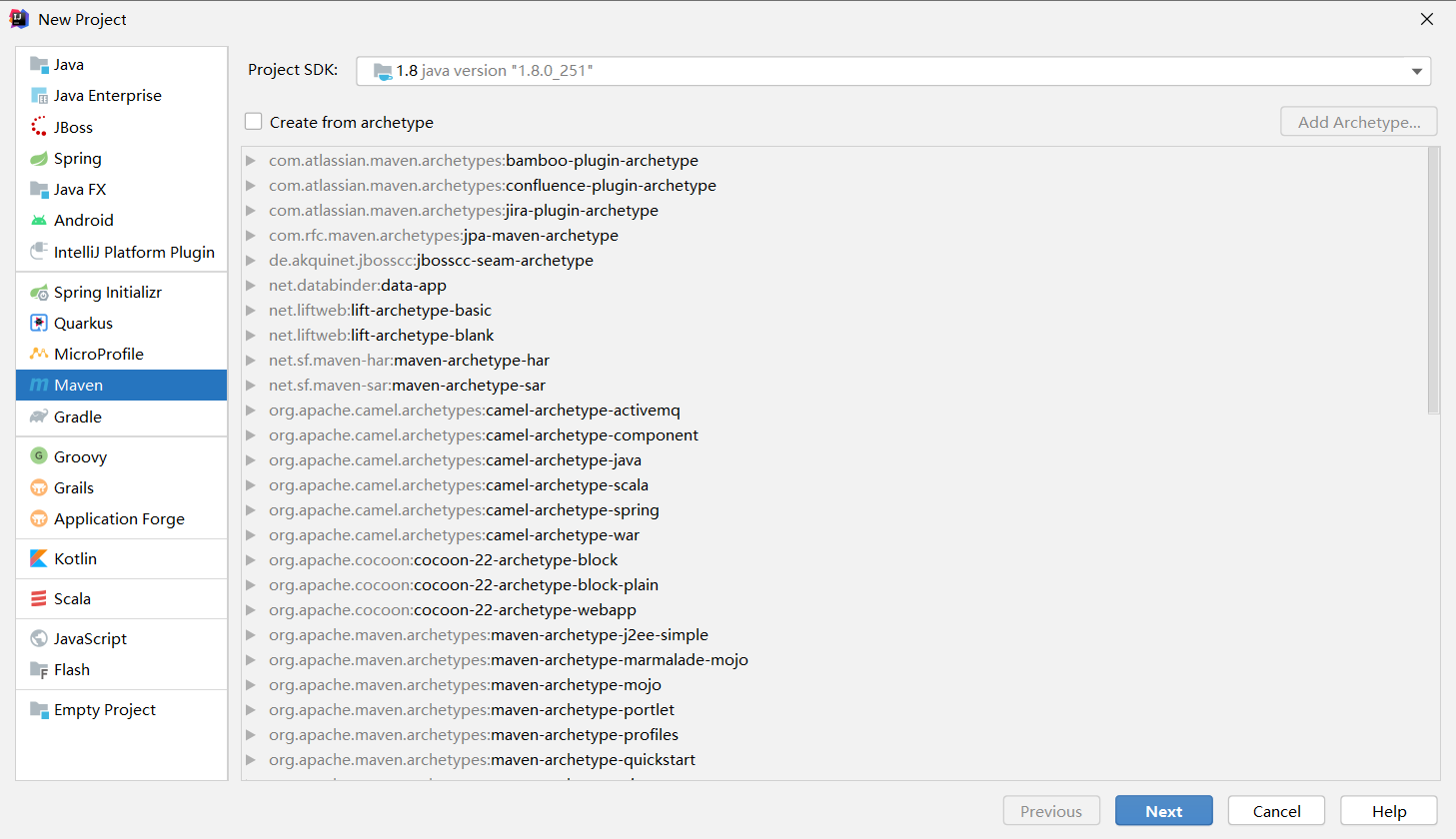
（2）输入Scala并按回车键，启动Scala环境。然后定义两个变量，并计算求和。



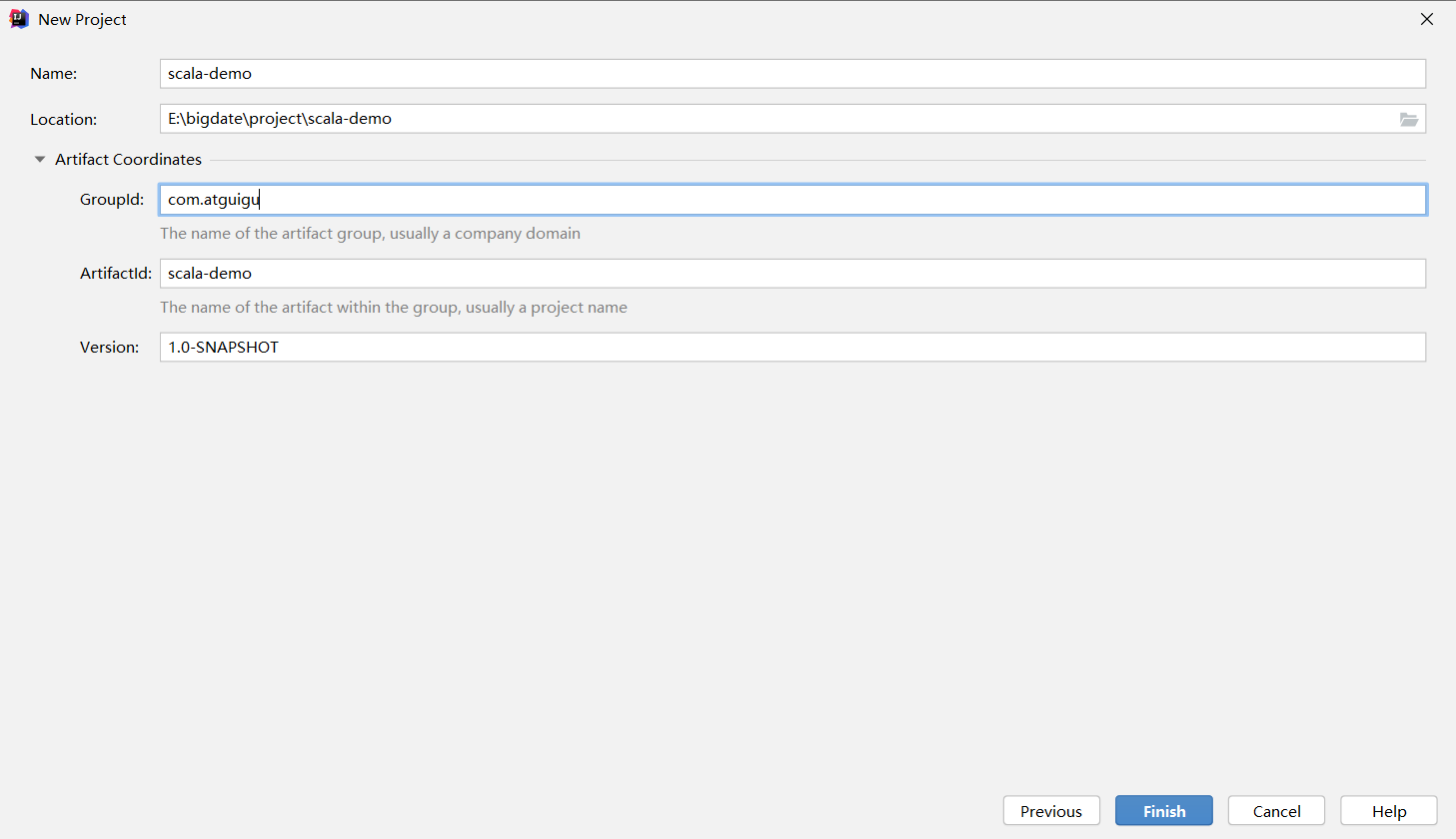
## 1.3 HelloWorld案例

### 1.3.1 idea中的hello world案例

1）创建新的maven工程

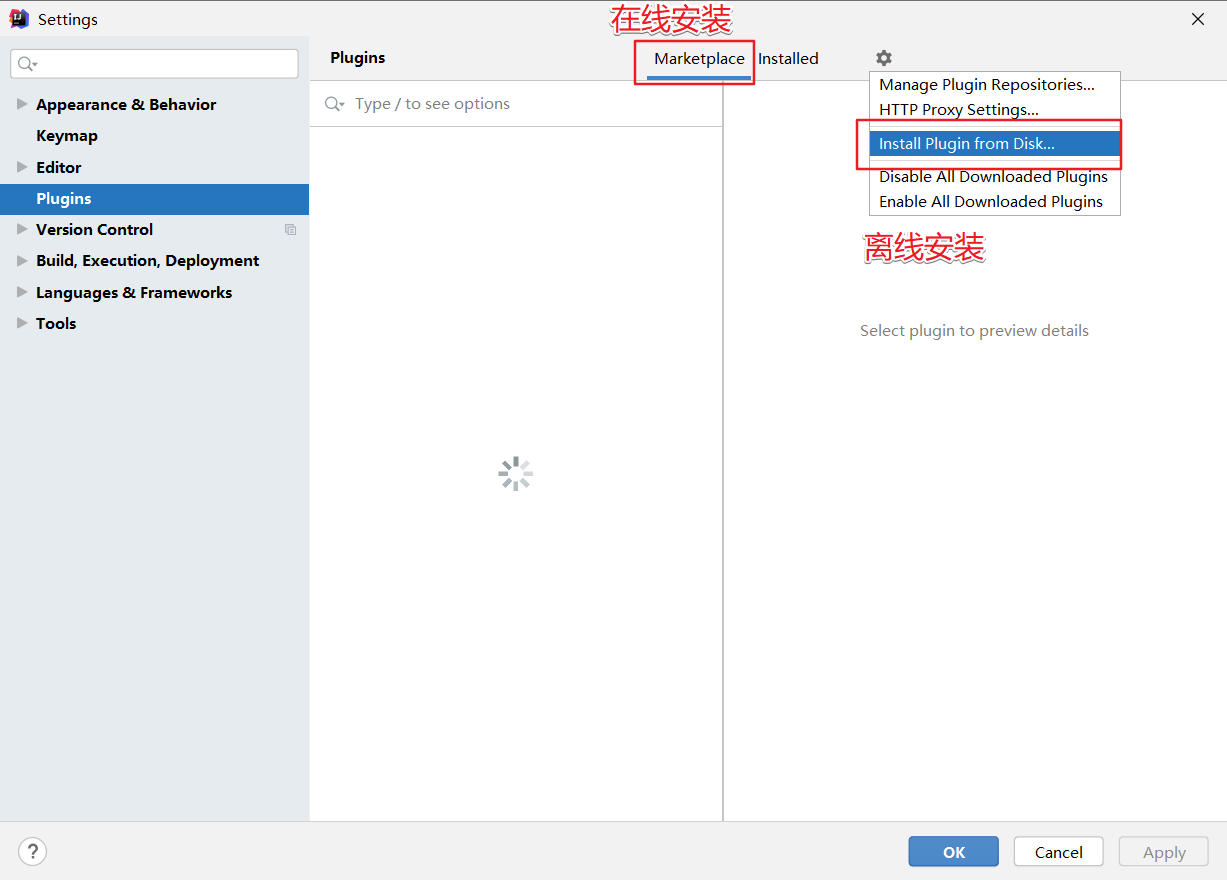


2）填写项目名称

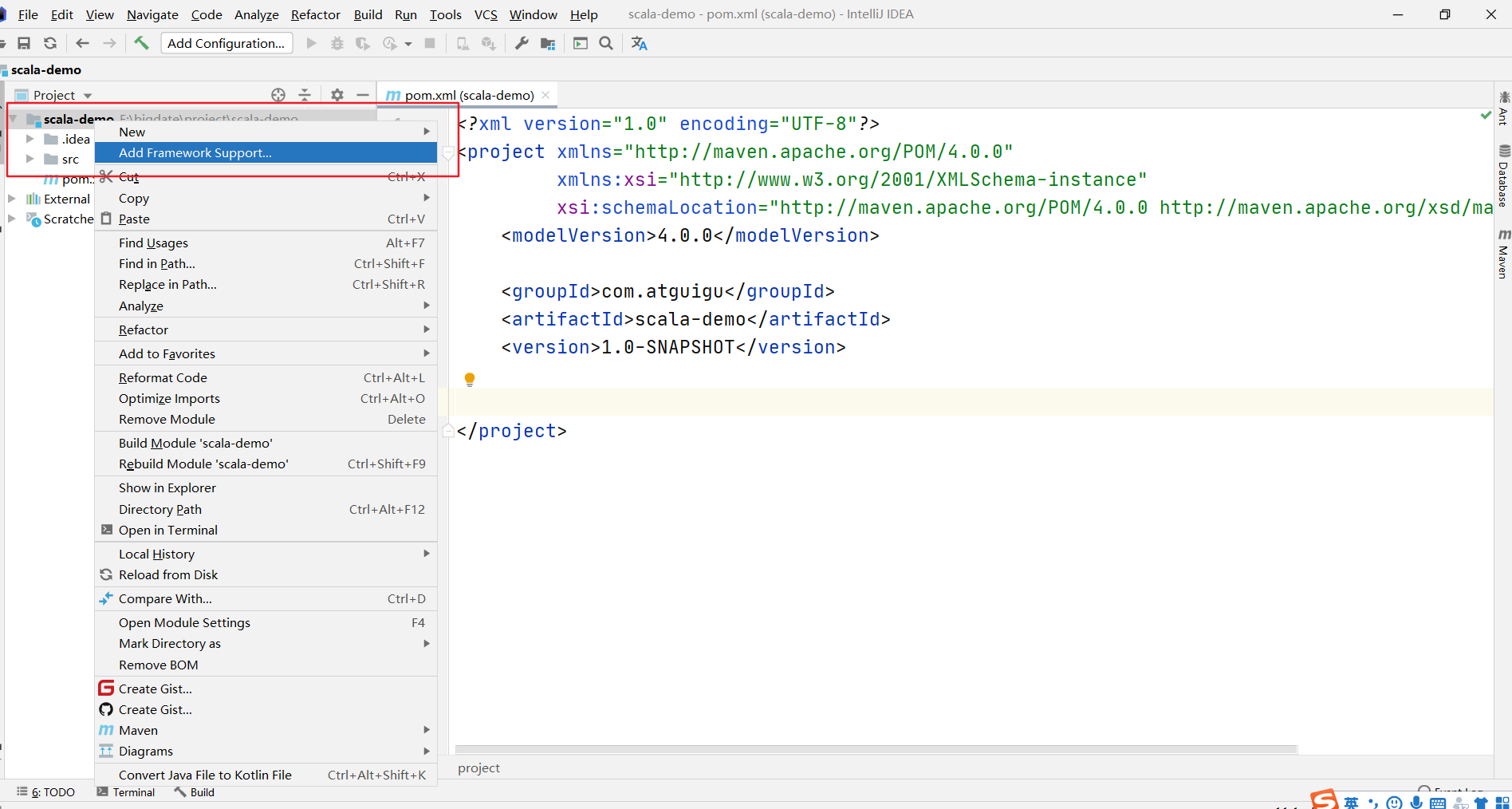


3）添加scala插件

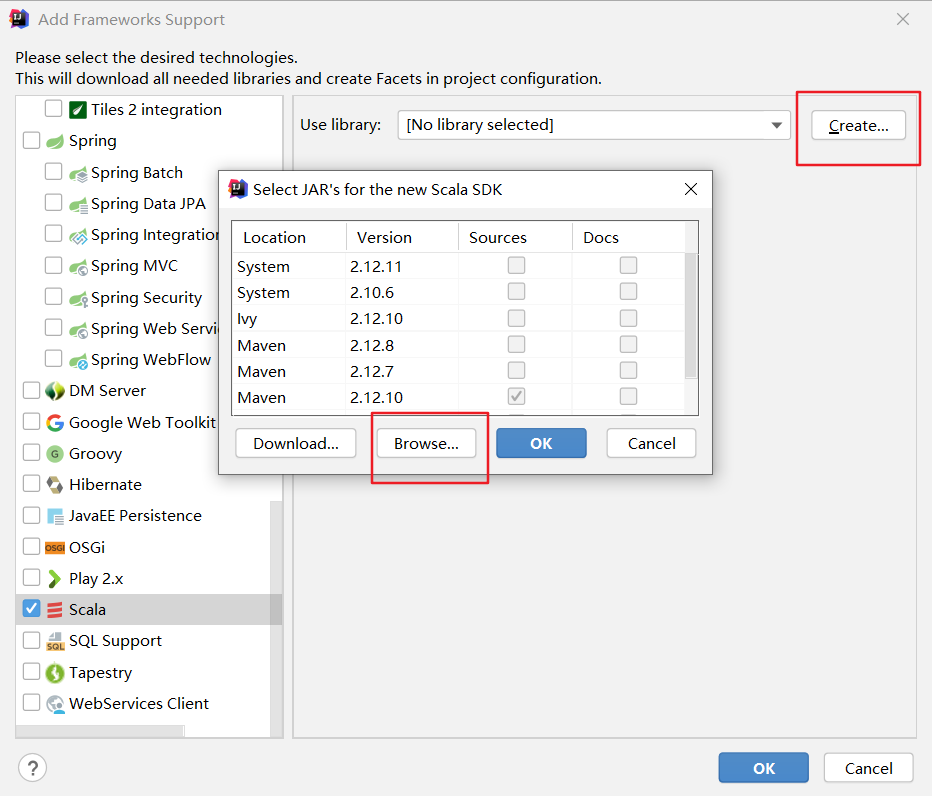
推荐使用离线安装，直接使用资料中提供的离线插件包scala-intellij-bin-2020.1.43.zip。（注意选择和自己IDEA相同的版本）。

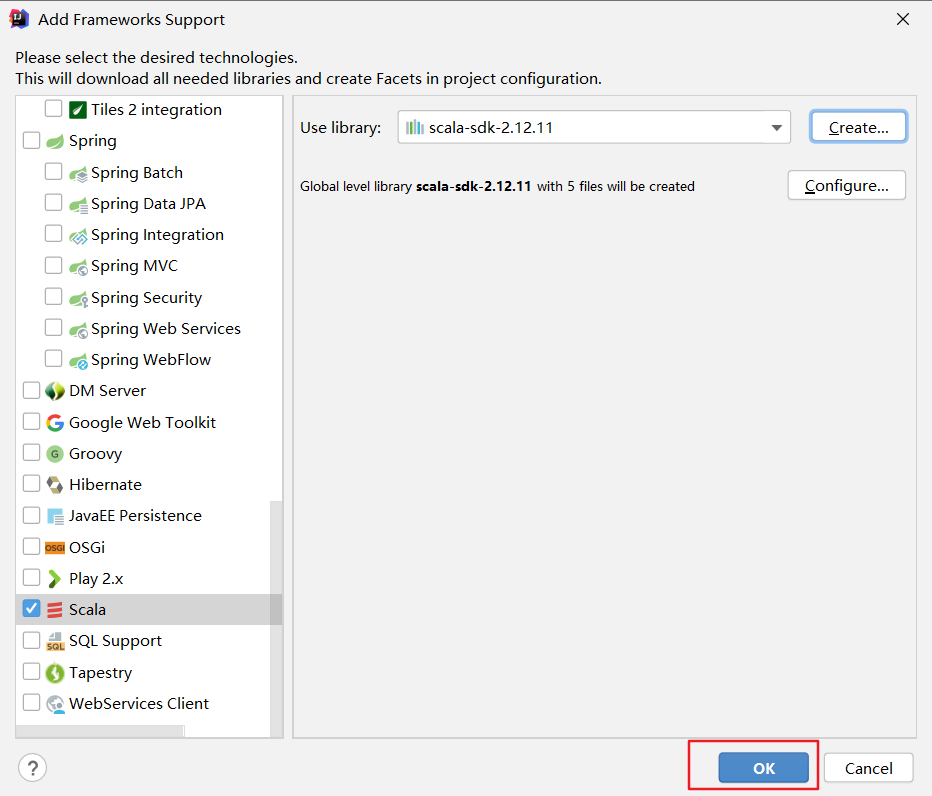


4）添加项目支持

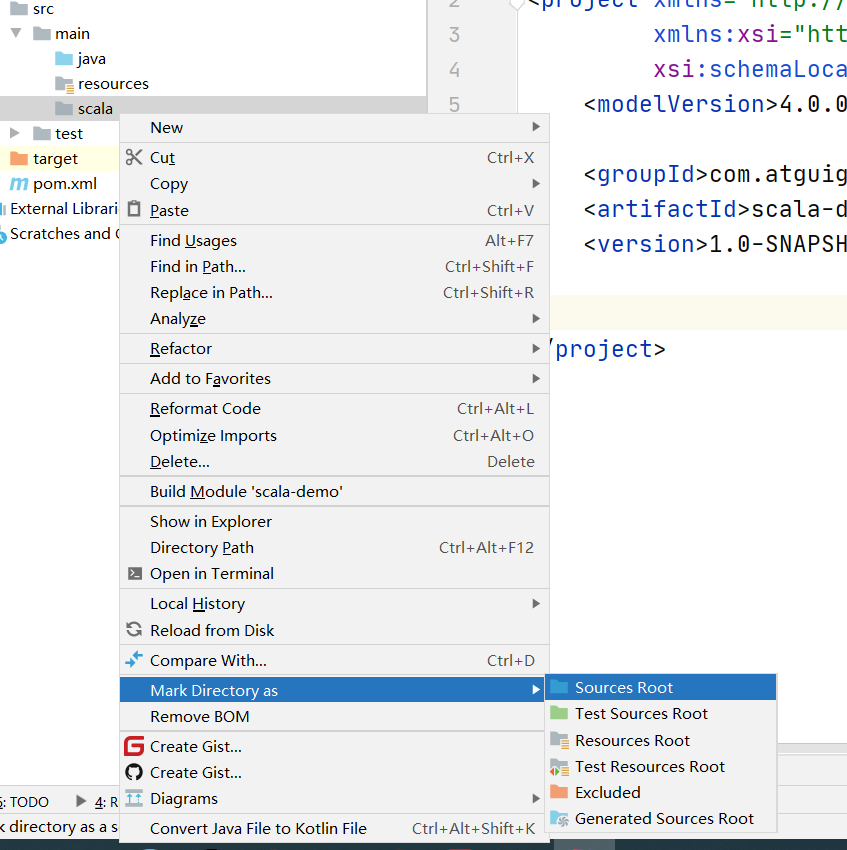


选择自己安装的SDK。

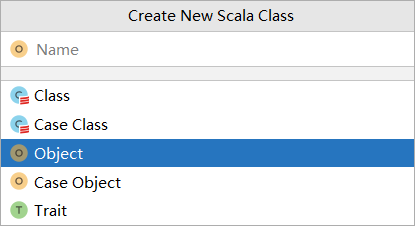




5）创建文件夹scala并标记为source-root



6）创建scala的object文件HelloScala



7）编写代码运行

object HelloScala {

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

// java的方法调用

System.out.println("hello scala")

// scala的方法调用

println("hello scala")

}

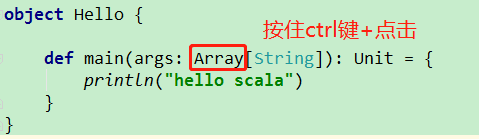
}

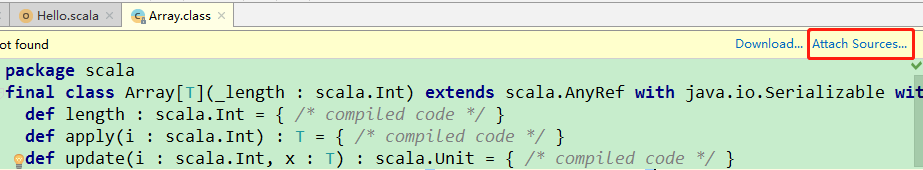
## 1.4 关联Scala源码

在使用Scala过程中，为了搞清楚Scala底层的机制，需要查看源码，下面看看如何关联和查看Scala的源码包。

1）查看源码

例如查看Array源码。按住ctrl键->点击Array->右上角出现Attach Soures…

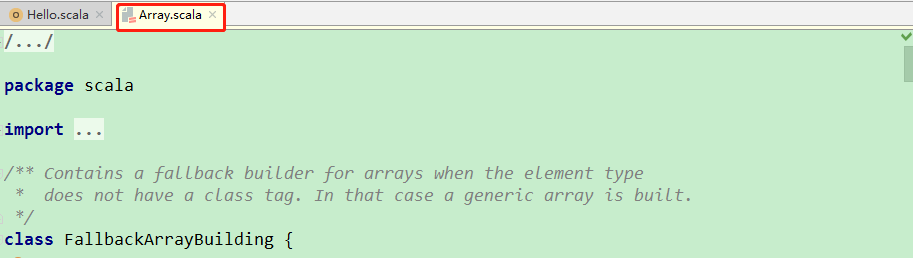




2）关联源码

（1）将我们的源码包scala-sources-2.12.11.tar.gz拷贝到D:\Tools\scala-2.12.11\lib文件夹下，并解压为scala-sources-2.12.11文件夹。

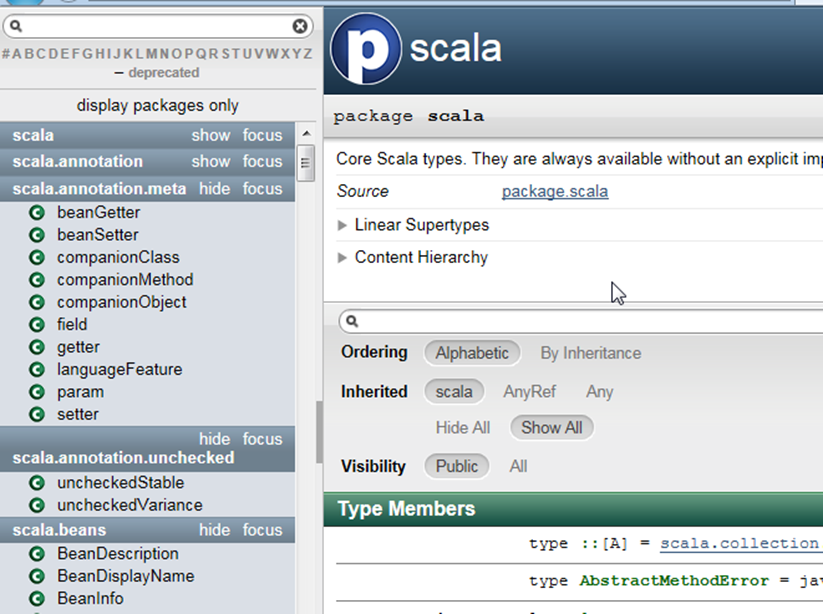
（2）点击Attach Sources…->选择D:\Tools\scala-2.12.11\lib\scala-sources-2.12.11，这个文件夹，就可以看到源码了。



## 1.5 官方编程指南

1）在线查看：<https://www.scala-lang.org/>

2）离线查看：解压scala-docs-2.12.11.zip，可以获得Scala的API操作。



# 第2章 变量和数据类型

## 2.1 注释

Scala注释使用和Java完全一样。

注释是一个程序员必须要具有的良好编程习惯。将自己的思想通过注释先整理出来，再用代码去体现。

1）基本语法

（1）单行注释：//

（2）多行注释：/\* \*/

（3）文档注释：/\*\*

\*

\*/

2）代码规范

（1）使用一次tab操作，实现缩进，默认整体向右边移动，用shift+tab整体向左移。

（2）或者使用ctrl + alt + L来进行格式化。

（3）运算符两边习惯性各加一个空格。比如：2 + 4 \* 5。

（4）一行最长不超过80个字符，超过的**请使用换行展示**，尽量保持格式优雅。

## 2.2 变量和常量（重点）

**常量：**在程序执行的过程中，其值不会被改变的变量。

0）回顾：Java变量和常量语法

变量类型 变量名称 = 初始值 int a = 10

final常量类型 常量名称 = 初始值 final int b = 20

注意：java中的final如果加static才会被存放在常量池中，否则作为不可修改的变量存在堆栈中。

1）基本语法

var 变量名 [: 变量类型] = 初始值 var i:Int = 10

val 常量名 [: 常量类型] = 初始值 val j:Int = 20

注意：能用常量的地方不用变量。

2）案例实操

（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）变量声明时，必须要有初始值。

（4）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。

（5）var修饰的对象引用可以改变，val修饰的对象则不可改变，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）。

package com.atguigu.chapter02

object Test01\_Var {

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

**// 声明变量和常量**

val a: Int = 10

var b: Int = 20

**// 常量值无法修改**

// a = 20

b = 30

**// （1）声明变量时，类型可以省略，编译器自动推导，即类型推导**

val c = 30

**// （2）类型确定后，就不能修改，说明Scala是强数据类型语言。**

// b = "30"

**// （3）变量声明时，必须要有初始值**

val d: Int = 0

// var d1:Int = \_

val test02\_Var = new Test02\_Var()

println(test02\_Var.i)

**// （4）var修饰的对象引用可以改变，val修饰的对象则不可改变，**

**// 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）**

val person0 = new Person02()

var person1 = new Person02()

**// 引用数据类型的常量和变量能否替换成别的对象**

**// var 可以修改引用数据类型的地址值 val不行**

person1 = new Person02()

**// 引用数据类型中的属性值能否发生变化 取决于内部的属性在定义的时候是var还是val**

// person0.name = "lisi"

person0.age = 11

}

}

class Test01\_Var {

**// scala中类的属性 如果是var变量也能使用默认值 但是必须要有等号**

var i: Int = \_

}

class Person01 {

val name: String = "zhangsan"

var age: Int = 10

}

## 2.3 标识符的命名规范

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。即：凡是自己可以起名字的地方都叫标识符。

1）命名规则

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，有以下三种规则：

（1）以字母或者下划线开头，后接字母、数字、下划线

（2）以操作符开头，且只包含操作符（+ - \* / # !等）

（3）用反引号`....`包括的任意字符串，即使是Scala关键字（39个）也可以

* package, import, class, object, trait, extends, with, type, for
* 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

注意：正常使用不能只遵守规则，必须按照规范来写，即使用大小驼峰命名法。

## 2.4 字符串输出(重点)

1）基本语法

（1）字符串，通过+号连接。

（2）重复字符串拼接。

（3）printf用法：字符串，通过%传值。

（4）字符串模板（插值字符串）：通过$获取变量值。

2）案例实操

package com.atguigu.chapter02

object Test02\_Str {

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

**// （1）字符串，通过+号连接**

System.out.println()

println("hello" + "world")

**// （2）重复字符串拼接**

println("linhailinhai" \* 200)

**// （3）printf用法：字符串，通过%传值。**

printf("name: %s age: %d\n", "linhai", 8)

**// （4）字符串模板（插值字符串）：通过$获取变量值**

val name = "linhai"

val age = 8

val s1 = s"name: $name,age:${age}"

println(s1)

val s2 = s"name: ${name + 1},age:${age + 2}"

println(s2)

**// （5）长字符串 原始字符串**

println("我" +

"是" +

"一首" +

"诗")

**//多行字符串，在Scala中，利用三个双引号包围多行字符串就可以实现。**

// 输入的内容，带有空格、\t之类，导致每一行的开始位置不能整洁对齐。

//应用scala的stripMargin方法，在scala中stripMargin默认是“|”作为连接符，

// 在多行换行的行头前面加一个“|”符号即可。

println(

"""我

|是

|一首

|诗

|""".stripMargin)

"""

|select id,

| age

|from user\_info

|""".stripMargin

s"""

|${name}

|${age}

|""".stripMargin

}

}

## 2.5 数据类型（重点）



object Test03\_Type {

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

**// 所有的代码都是代码块**

**// 表示运行一段代码 同时将最后一行的结果作为返回值**

**// 千万不要写return**

val i: Int = {

println("我是代码块")

10 + 10

}

**// 代码块为1行的时候 大括号可以省略**

val i1: Int = 10 + 10

**// 如果代码块没有计算结果 返回类型是unit**

val unit: Unit = {

println("hello")

println("我是代码块")

}

**// 当代码块没办法完成计算的时候 返回值类型为nothing**

// val value: Nothing = {

// println("hello")

// throw new RuntimeException

// }

}

}

## 2.7 整数类型（Byte、Short、Int、Long）

Scala的整数类型就是用于存放整数值的，比如12，30，3456等等。

1）整型分类

|  |  |
| --- | --- |
| 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 |

2）案例实操

（1）Scala各整数类型有固定的表示范围和字段长度，不受具体操作的影响，以保证Scala程序的可移植性。

object Test03\_Type {

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

**// 整数类型**

val i1 = 1

val l = 1L

**// （1）Scala各整数类型有固定的表示范围和字段长度，不受具体操作的影响，以保证Scala程序的可移植性。**

val b1: Byte = 2

// val b0: Byte = 128

val b2: Byte = 1 + 1

println(b2)

val i2 = 1

**//（2）编译器对于常量值的计算 能够直接使用结果进行编译**

**// 但是如果是变量值 编译器是不知道变量的值的 所以判断不能将大类型的值赋值给小的类型**

// val b3: Byte = i2 + 1

// println(b3)

**// （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**

val l1 = 2200000000L

}

}

## 2.8 浮点类型（Float、Double）

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

1）浮点型分类

|  |  |
| --- | --- |
| 数据类型 | 描述 |
| Float [4] | 32 位, IEEE 754标准的单精度浮点数 |
| Double [8] | 1. 位 IEEE 754标准的双精度浮点数 |

2）案例实操

Scala的浮点型常量默认为Double型，声明Float型常量，须后加‘f’或‘F’。

object Test03\_Type{

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

**// 浮点数介绍**

**// 默认使用double**

val d: Double = 3.14

**// 如果使用float 在末尾添加f**

val fl = 3.14f

**// 浮点数计算有误差**

println(0.1 / 3.3)

}

}

//运行的结果

0.030303030303030307

## 2.9 字符类型（Char）

1）基本说明

字符类型可以表示单个字符，字符类型是Char。

2）案例实操

（1）字符常量是用单引号 ' ' 括起来的单个字符。

（2）\t ：一个制表位，实现对齐的功能

（3）\n ：换行符

（4）\\ ：表示\

（5）\" ：表示"

object Test03\_Type{

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

**// （1）字符常量是用单引号 ' ' 括起来的单个字符。**

val c1: Char = 'a'

val c2: Char = 65535

**// （2）\t ：一个制表位，实现对齐的功能**

val c3: Char = '\t'

**// （3）\n ：换行符**

val c4: Char = '\n'

println(c3 + 0)

println(c4 + 0)

**// （4）\\ ：表示\**

val c5: Char = '\\'

println(c5 + 0)

**// （5）\" ：表示"**

val c6: Char = '\"'

println(c6 + 0)

}

}

## 2.10 布尔类型：Boolean

1）基本说明

（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。

2）案例实操

object Test03\_Type{

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

val bo1: Boolean = true

val bo2: Boolean = false

}

}

## 2.11 Unit类型、Null类型和Nothing类型（重点）

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()。 |
| Null | null , Null 类型只有一个实例值null |
| Nothing | Nothing类型在Scala的类层级最低端；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）案例实操

（1）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意义。

object Test03\_Type{

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

// unit

val unit1: Unit = {

10

println("1")

}

println(unit1)

**// 如果标记对象的类型是unit的话 后面有返回值也没法接收**

**// unit虽然是数值类型 但是可以接收引用数据类型 因为都是表示不接收返回值**

val i3: Unit = "aa"

println(i3)

}

}

（2）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）。**

object Test03\_Type {

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

**// scala当中使用的字符串就是java中的string**

val aa: String = "aa"

// null

var aa1: String = "aa"

aa1 = "bb"

aa1 = null

if (aa1 != null) {

val strings: Array[String] = aa1.split(",")

}

**// 值类型不能等于null,idea不会识别报错 编译器会报错**

var i4 = 10

// i4 = null

}

}

（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。

object Test03\_Type {

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

val value: Nothing = {

println("hello")

1 + 1

throw new RuntimeException()

}

}

}

## 2.12 类型转换

Java的隐式转换规则：

public class TestCast {

public static void main(String[] args) {

byte n = 23;

test(n);

}

public static void test(byte b) {

System.out.println("bbbb");

}

public static void test(short b) {

System.out.println("ssss");

}

public static void test(char b) {

System.out.println("cccc");

}

public static void test(int b) {

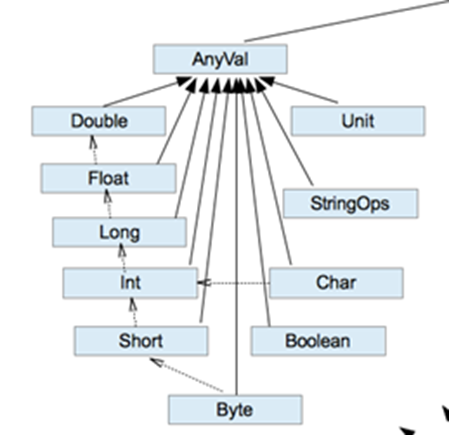
System.out.println("iiii");

}

}

### 2.12.1 数值类型自动转换

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值类型，这个就是自动类型转换（隐式转换）。数据类型按精度（容量）大小排序为：



1）基本说明

（1）自动提升原则：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精度大的那种数据类型，然后再进行计算。

（2）把精度大的数值类型赋值给精度小的数值类型时，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。

2）案例实操

object Test04\_TypeCast {

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

**// （1）自动提升原则：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精度大的那种数据类型，然后再进行计算。

val fl: Float = 1 + 1L + 3.14f

val d: Double = 1 + 1L + 3.14f + 3.14

**// （2）把精度大的数值类型赋值给精度小的数值类型时，就会报错，反之就会进行自动类型转换。**

val i = 10

val b: Double = i

**// （3）（byte，short）和char之间不会相互自动转换。**

// 因为byte和short是有符号的数值,而char是无符号的

val b1: Byte = 10

val c1: Char = 20

**// （4）byte，short，char他们三者可以计算，在计算时首先转换为int类型。**

val b2: Byte = 20

// val i1: Byte = b1 + b2

val i1: Int = 1100000000

val i2: Int = 1200000000

**// 超出范围的int值计算会造成结果错误**

val i3: Int = i1 + i2

println(i3)

}

}

注意：Scala还提供了非常强大的隐式转换机制（隐式函数，隐式类等），我们放在高级部分专门用一个章节来讲解。

### 2.12.2 强制类型转换

1）基本说明

自动类型转换的逆过程，将精度大的数值类型转换为精度小的数值类型。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。

Java : int num = (int)2.5

Scala : var num : Int = 2.7.toInt

2）案例实操

（1）将数据由高精度转换为低精度，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。

object Test04\_TypeCast {

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

**// 强制类型转换**

val d1 = 2.999

**// （1）将数据由高精度转换为低精度，就需要使用到强制转换**

println((d1 + 0.5).toInt)

**// （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**

println((10 \* 3.5 + 6 \* 1.5).toInt)

}

}

### 2.12.3 数值类型和String类型间转换

1）基本说明

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

2）案例实操

（1）基本类型转String类型（语法：将基本类型的值+"" 即可）。

（2）String类型转基本数值类型（语法：s1.toInt、s1.toFloat、s1.toDouble、s1.toByte、s1.toLong、s1.toShort）。

object Test04\_TypeCast {

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

**// （1）基本类型转String类型（语法：将基本类型的值+"" 即可）**

val string: String = 10.0.toString

println(string)

val str: String = 1 + ""

**// （2）String类型转基本数值类型（语法：s1.toInt、s1.toFloat、**s1.toDouble、s1.toByte、s1.toLong、s1.toShort）

val double: Double = "3.14".toDouble

println(double + 1)

println(double.toInt)

**// 不能直接将小数类型的字符串转换为整数 需要先转换为double再转换int**

// println("3.14".toInt)

**// 标记为f的float数能够识别**

// println("12.6f".toFloat)

}

}

**扩展面试题数值存储机制介绍：**

object Test04\_TypeCast {

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

**// 将int值130强转为byte 值为多少**

// 0000 0000 ..16.. 1000 0010 => 表示int的130

val i4 = 130

// 1000 0010

println(i4.toByte)

}

}

# 第3章 运算符

Scala运算符的使用和Java运算符的使用基本相同，只有个别细节上不同。

## 3.1 算术运算符

1）基本语法

|  |  |  |  |
| --- | --- | --- | --- |
| 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 关系运算符（比较运算符）

1）基本语法

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 运算 | 范例 | 结果 |
| == | 相等于 | 4==3 | false |
| != | 不等于 | 4！=3 | true |
| < | 小于 | 4<3 | false |
| > | 大于 | 4>3 | true |
| <= | 小于等于 | 4<=3 | false |
| >= | 大于等于 | 4>=3 | true |

2）案例实操

（1）需求1

object Test01\_Operation{

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

**// 测试：>、>=、<=、<、==、!=**

var a: Int = 2

var b: Int = 1

println(a > b) // true

println(a >= b) // true

println(a <= b) // false

println(a < b) // false

println("a==b" + (a == b)) // false

println(a != b) // true

}

}

（2）需求2：Java和Scala中关于==的区别

Java：

==比较两个变量本身的值，即两个对象在内存中的首地址；

equals比较字符串中所包含的内容是否相同。

public static void main(String[] args) {

String s1 = "abc";

String s2 = new String("abc");

System.out.println(s1 == s2);

System.out.println(s1.equals(s2));

}

**输出结果：**

false

true

Scala：==更加类似于Java中的equals，参照jd工具

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

val s1 = "abc"

val s2 = new String("abc")

println(s1 == s2)

println(s1.eq(s2))

}

**输出结果：**

true

false

## 3.3 逻辑运算符

1）基本语法

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。

假定：变量A为true，B为false

|  |  |  |
| --- | --- | --- |
| 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）案例实操

object Test01\_Operation {

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

**// 测试：&&、||、!**

var a = true

var b = false

println("a&&b=" + (a && b)) // a&&b=false

println("a||b=" + (a || b)) // a||b=true

println("!(a&&b)=" + (!(a && b))) // !(a&&b)=true

}

}

扩展避免逻辑与空指针异常

def isNotEmpty(String s){

**//如果按位与，s为空，会发生空指针**

return s!=null &**& !**"".equals(s.trim());

}

## 3.4 赋值运算符

1）基本语法

赋值运算符就是将某个运算后的值，赋给指定的变量。

|  |  |  |
| --- | --- | --- |
| 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 |

注意：Scala中没有++、--操作符，可以通过+=、-=来实现同样的效果；

2）案例实操

object Test01\_Operation {

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

var r1 = 10

r1 += 1 // 没有++

r1 -= 2 // 没有--

}

}

## 3.5 位运算符

1）基本语法

下表中变量 a 为 60，b 为 13。

|  |  |  |
| --- | --- | --- |
| 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： 0011 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 |

2）案例实操

object Test01\_Operation {

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

**// 测试：1000 << 1 =>10000**

var n1 :Int =8

n1 = n1 << 1

println(n1)

}

}

## 3.6 Scala运算符本质

在Scala中其实是没有运算符的，所有运算符都是方法。

（1）当调用对象的方法时，点.可以省略

（2）如果函数参数只有一个，或者没有参数，()可以省略

object Test01\_Operation {

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

**// 标准的加法运算**

val i:Int = 1.+(1)

**// （1）当调用对象的方法时，.可以省略**

val j:Int = 1 + (1)

**// （2）如果函数参数只有一个，或者没有参数，()可以省略**

val k:Int = 1 + 1

println(1.toString())

println(1 toString())

println(1 toString)

}

}

# 第4章 流程控制（没有Switch）

## 4.1 分支控制if-else

让程序有选择的的执行，分支控制有三种：单分支、双分支、多分支

1）案例实操

（1）需求1：需求：输入年龄，如果年龄小于18岁，则输出“童年”。如果年龄大于等于18且小于等于60，则输出“中年”，否则，输出“老年”。

object Test01\_If {

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

println("input age")

var age = StdIn.readInt()

if (age < 18){

println("童年")

}else if(age>=18 && age<60){

println("中年")

}else{

println("老年")

}

}

}

（2）需求2：Scala中if else表达式其实是有返回值的，具体返回值取决于满足条件的代码体的最后一行内容。

object Test01\_If {

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

println("input age")

var age = StdIn.readInt()

val res :String = if (age < 18){

"童年"

}else if(age>=18 && age<60){

"中年"

}else{

"老年"

}

println(res)

}

}

（3）需求3：Scala中返回值类型不一致，取它们共同的祖先类型。

object Test01\_If {

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

println("input age")

var age = StdIn.readInt()

val res:Any = if (age < 18){

"童年"

}else if(age>=18 && age<60){

"中年"

}else{

100

}

println(res)

}

}

（4）需求4：Java中的三元运算符可以用if else实现

如果大括号{}内的逻辑代码只有一行，大括号可以省略。如果省略大括号，if只对最近的一行逻辑代码起作用。

object Test01\_If {

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

// Java

// int result = flag?1:0

// Scala

println("input age")

var age = StdIn.readInt()

val res:Any = if (age < 18) "童年" else "成年"

"不起作用"

println(res)

}

}

## 4.2 Switch分支结构

在Scala中没有Switch，而是使用模式匹配来处理。

模式匹配涉及到的知识点较为综合，因此我们放在后面讲解。

## 4.3 For循环控制

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推导式或for表达式。

### 4.3.1 基本语法

1）基本语法

object Test02\_ForLoop {

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

**// scala中的for循环基础语法**

for (i <- 0 to 5) {

println(i)

}

for (i <- 0 until 5) {

println(i)

}

**// for循环的本质**

**// to是整数的方法 返回结果是一个集合**

**// 使用变量i 循环遍历一遍 后面集合的内容**

val inclusive: Range.Inclusive = 0.to(5)

}

### 4.3.1 补充语法

补充语法有：

（1）循环守卫

（2）循环返回值

object Test02\_ForLoop {

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

**// 循环守卫**

for (i <- 0 to 10) {

if (i > 5) {

println(i)

}

}

for (i <- 0 to 10 if i > 5) {

println(i)

}

**// 循环返回值**

val ints: immutable.IndexedSeq[Int] = for (i <- 0 to 3) yield {

10

}

}

}

## 4.5 While和do..While循环控制

While和do..While的使用和Java语言中用法相同。

### 4.5.1 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

}

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for语句不同，while语句没有返回值，即整个while语句的结果是Unit类型()

（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。

2）案例实操

object Test03\_While {

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

var i = 0

while (i < 5) {

println(i)

i += 1

}

while (i < 5) {

println(i)

i += 1

}

}

### 4.5.2 do..while循环控制

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)

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

2）案例实操

object Test03\_While {

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

var i = 0

**// do while 一定会执行一次 不管条件是否成立**

do {

println(i)

}while(i > 100)

}

}

## 4.6 循环中断

1）基本说明

Scala内置控制结构特地**去掉了break和continue**，是为了更好的适应**函数式编程**，推荐使用函数式的风格解决break和continue的功能，而不是一个关键字。Scala中使用breakable控制结构来实现break和continue功能。

2）案例实操

需求1：采用异常的方式退出循环

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

try {

for (elem <- 1 to 10) {

println(elem)

if (elem == 5) throw new RuntimeException

}

}catch {

case e =>

}

println("正常结束循环")

}

需求2：采用Scala自带的函数，退出循环

import scala.util.control.Breaks

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

Breaks.breakable(

for (elem <- 1 to 10) {

println(elem)

if (elem == 5) Breaks.break()

}

)

println("正常结束循环")

}

需求3：对break进行省略

import scala.util.control.Breaks.\_

object TestBreak {

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

breakable {

for (elem <- 1 to 10) {

println(elem)

if (elem == 5) break

}

}

println("正常结束循环")

}

}

# 第5章 函数式编程

1）面向对象编程

解决问题，分解对象，行为，属性，然后通过对象的关系以及行为的调用来解决问题。

对象：用户；

行为：登录、连接jdbc、读取数据库

属性：用户名、密码

Scala语言是一个完全面向对象编程语言。万物皆对象

2）函数式编程

解决问题时，将问题分解成一个一个的步骤，将每个步骤进行封装(函数)，通过调用这些封装好的步骤，解决问题。

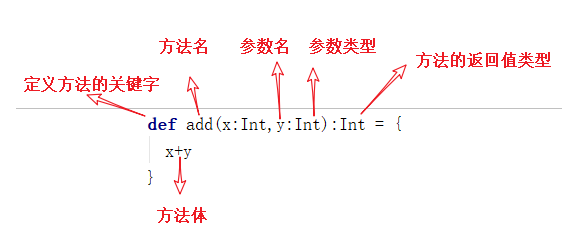
例如：请求->用户名、密码->连接jdbc->读取数据库

Scala语言是一个完全函数式编程语言。万物皆函数

3）在Scala中函数式编程和面向对象编程融合在一起了。

## 5.1 方法基本语法

1）基本语法



2）案例实操

需求：定义一个方法，实现将传入的名称打印出来。

object TestFunction {

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

// 1 方法声明

def f(arg: String): Unit = {

println(arg)

}

// 2 方法调用

// 方法名（参数）

f("hello world")

}

}

## 5.2 方法声明

1）方法声明

（1）方法1：无参，无返回值

（2）方法2：无参，有返回值

（3）方法3：有参，无返回值

（4）方法4：有参，有返回值

（5）方法5：多参，无返回值

2）案例实操

package com.atguigu.chapter06  
  
object TestFunctionDeclare {  
  
 def main(args: Array[String]): Unit = {  
  
 // 方法1：无参，无返回值  
 def test(): Unit ={  
 println("无参，无返回值")  
 }  
 test()  
  
 // 方法2：无参，有返回值  
 def test2():String={  
 return "无参，有返回值"  
 }  
 println(test2())  
  
 // 方法3：有参，无返回值  
 def test3(s:String):Unit={  
 println(s)  
 }  
 test3("jinlian")  
  
 // 方法4：有参，有返回值  
 def test4(s:String):String={  
 return s+"有参，有返回值"  
 }  
 println(test4("hello "))  
  
  
 // 方法5：多参，无返回值  
 def test5(name:String, age:Int):Unit={  
 println(s"$name, $age")  
 }  
 test5("dalang",40)  
 }  
}

## 5.3 方法参数

1）案例实操

（1）可变参数

（2）如果参数列表中存在多个参数，那么可变参数一般放置在最后，（不能和默认值一起用，和带名参数用时，不能改变带名参数的顺序）

（3）参数默认值，一般将有默认值的参数放置在参数列表的后面

（4）带名参数

object Test03\_FunArgs {

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

**// (1)可变参数**

def sayHi(names:String\*):Unit = {

println(s"hi $names")

**// 可变参数在函数值本质是一个数组**

for (elem <- names) {

}

}

sayHi()

sayHi("linhai")

sayHi("linhai","jinlian")

**// 可变参数使用:**

**// (2)可变参数必须在参数列表的最后**

def sayHi1(sex: String,names:String\*):Unit = {

println(s"hi $names")

}

**// (3)参数默认值**

def sayHi2(name:String = "linhai"):Unit = {

println(s"hi ${name}")

}

sayHi2("linhai")

sayHi2()

**// 默认值参数在使用的时候 可以不在最后**

def sayHi3( name:String = "linhai" , age:Int):Unit = {

println(s"hi ${name}")

}

**// (4)带名参数**

sayHi3(age = 10)

}

}

## 5.4 方法至简原则

方法至简原则：能省则省

1）至简原则细节

常用化简写法：

（1）return可以省略，Scala会使用方法体的最后一行代码作为返回值

（2）如果方法体只有一行代码，可以省略花括号

（3）返回值类型如果能够推断出来，那么可以省略（:和返回值类型一起省略）

特别注意事项：

（4）如果有return，则不能省略返回值类型，必须指定

（5）如果方法明确声明unit，那么即使方法体中使用return关键字也不起作用

（6）Scala如果期望是无返回值类型，可以省略等号（=号和方法体大括号不能同时省略）

（7）如果方法无参，但是声明了参数列表，那么调用时，小括号，可加可不加（声明无括号调用时也没有括号）

（8）如果方法没有参数列表，那么小括号可以省略，调用时小括号必须省略

2）案例实操

object Test04\_FuncSimply {

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

**// 定义一个原方法**

def func0(x: Int, y: Int): Int = {

println("func0的调用")

if (x < 20) {

return x + y

}

2 \* x + 3 \* y

}

println(func0(10, 20))

**// （1）return可以省略，Scala会使用方法体的最后一行代码作为返回值**

def func1(x: Int, y: Int): Int = {

println("func1的调用")

x + y

}

val i: Int = func1(10, 20)

println(i)

**// （2）如果方法体只有一行代码，可以省略花括号**

// 如果不写大括号 默认有效范围只有一行

def func2(x: Int, y: Int): Int = x + y

**// （3）返回值类型如果能够推断出来，那么可以省略（:和返回值类型一起省略）**

def func3(x: Int, y: Int) = x + y

**// （4）如果有return，则不能省略返回值类型，必须指定**

def func4(x: Int, y: Int): Int = {

if (x < 20) {

return x + y

}

2 \* x + 3 \* y

}

func4(10, 20)

**// （5）如果方法明确声明unit，那么即使函数体中使用return关键字也不起作用**

def func5(x: Int, y: Int): Unit = return x + y

val unit: Unit = func5(10, 20)

**// （6）Scala如果期望是无返回值类型，可以省略等号**

def func6(x: Int, y: Int) {

println(x + y)

}

**// （7）如果方法无参，但是声明了参数列表，那么调用时，小括号，可加可不加**

def func7(): Unit = {

println("hello")

}

func7()

func7

**// （8）如果方法没有参数列表，那么小括号可以省略，调用时小括号必须省略**

def func8 {

println("hello")

}

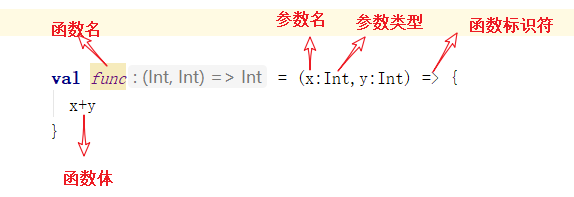
func8

}

}

## 5.5 函数的基本语法

1）基本语法：



函数的返回值就是函数体中最后一个表达式的结果值/return语句的返回值

2）案例实操

需求：定义一个函数，实现将传入的两个整数相加。

object TestFunction {

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

// 1 函数

val add = (x:Int,y:Int) => { x + y }

// 2 函数调用

// 函数名（参数）

val result = add(10,20)

println(result)

}

}

## 5.6 函数和方法的区别

1）方法定义在类中可以实现重载，函数不可以重载

2）方法是保存在方法区，函数是保存在堆中

3）定义在方法中的方法可以称之为函数，不可以重载

4）方法可以转成函数, 转换语法: 方法名 \_

object TestFunction {

//方法可以进行重载和重写，程序可以执行

def main(): Unit = {

}

//定义一个方法

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

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

//Scala语言的语法非常灵活，可以在任何的语法结构中声明任何的语法

import java.util.Date

new Date()

//函数没有重载和重写的概念，程序报错

val test =()=> {

println("无参，无返回值")

}

test()

val test = (name:String)=>{

println()

}

//（3）scala中函数可以嵌套定义

val test2 = ()=>{

val test3 = (name:String)=>{

println("函数可以嵌套定义")

}

}

//(4) 方法可以转成函数

val add2 = add \_

}

}

## 5.7 高阶函数

1）说明

定义：参数/返回值为函数的方法/函数称为高阶函数

2）案例实操

object TestFunction {

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

//制作一个计算器

//高阶函数————函数作为参数

def calculator(a: Int, b: Int, operater: (Int, Int) => Int): Int = {

operater(a, b)

}

//函数————求和

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

x + y

}

//方法————求积

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

x \* y

}

//函数作为参数

println(calculator(2, 3, plus))

println(calculator(2, 3, multiply))

}

}

## 5.8 匿名函数

1）说明

没有名字的函数/方法就是匿名函数。

(x:Int)=>{函数体}

x：表示输入参数类型；Int：表示输入参数类型；函数体：表示具体代码逻辑

2）案例实操

需求1：传递的函数有一个参数

传递匿名函数至简原则：

（1）参数的类型可以省略，会根据形参进行自动的推导

（2）类型省略之后，发现只有一个参数，则圆括号可以省略；其他情况：没有参数和参数超过1的永远不能省略圆括号。

（3）匿名函数如果只有一行，则大括号也可以省略

（4）如果参数只出现一次，且按照顺序出现则参数省略且后面参数可以用\_代替

不能化简为下划线的情况： 1.化简之后只有一个下划线 2.化简后的函数存在嵌套

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

val f0: (Int, Int) => Int = (x: Int, y: Int) => x + y

**// （1）参数的类型可以省略，会根据形参进行自动的推导**

val f1: (Int, Int) => Int = (x, y) => x + y

**// （2）类型省略之后，发现只有一个参数，则圆括号可以省略；**

// 其他情况：没有参数和参数超过1的永远不能省略圆括号。

val f2: (Int, Int) => Int = (x, y) => x + y

val f3: Int => Int = x => x + 22

val f4: () => Int = () => 10

**// （3）匿名函数如果只有一行，则大括号也可以省略**

val f5: (Int, Int) => Int = (x, y) => {

println("匿名函数")

x + y

}

**// （4）如果参数只出现一次，且按照顺序出现，则参数省略且后面参数可以用\_代替**

val f6: (Int, Int) => Int = \_ + \_

// 化简为\_的条件

**// 1. 传入的参数类型可以推断 所以可以省略**

val f7: (Int, Int) => Int = (x, y) => y - x

**// 2. 参数必须只使用一次 使用的顺序必要和定义的顺序一样**

val f8: (Int, Int) => Int = -\_ + \_

**// 如果化简为匿名函数 只剩下一个\_ 则不可以化简**

val function: String => String = \_ + ""

val str: String = function("linhai")

val function1: String => String = a => a

**// 如果化简的下划线在函数里面 也会报错**

// val function1: String => Unit = println(\_ + "hi")

val function2: String => Unit = println

function2("linhai")

}

## 5.9 函数柯里化&闭包

1）说明

函数柯里化：将一个接收多个参数的函数转化成一个接受一个参数的函数过程，可以简单的理解为一种特殊的参数列表声明方式。

闭包：就是**一个函数**和与**其相关的引用环境（变量）**组合的一个**整体**(实体)

2）案例实操

（1）闭包

//外部变量

var z: Int = 10

//闭包

def f(y: Int): Int = {

z + y

}

（2）柯里化

object TestFunction {

val sum = (x: Int, y: Int, z: Int) => x + y + z

val sum1 = (x: Int) => {

y: Int => {

z: Int => {

x + y + z

}

}

}

val sum2 = (x: Int) => (y: Int) => (z: Int) => x + y + z

def sum3(x: Int)(y: Int)(z: Int) = x + y + z

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

sum(1, 2, 3)

sum1(1)(2)(3)

sum2(1)(2)(3)

sum3(1)(2)(3)

}

}

## 5.10 递归

1）说明

一个函数/方法在函数/方法体内又调用了本身，我们称之为递归调用

2）案例实操

object TestFunction {

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

**// 阶乘**

**// 递归算法**

**// 1) 方法调用自身**

**// 2) 方法必须要有跳出的逻辑**

**// 3) 方法调用自身时，传递的参数应该有规律**

**// 4) scala中的递归必须声明函数返回值类型**

println(test(5))

}

def test( i : Int ) : Int = {

if ( i == 1 ) {

1

} else {

i \* test(i-1)

}

}

}

# 第6章 面向对象

Scala的面向对象思想和Java的面向对象思想和概念是一致的。

Scala中语法和Java不同，补充了更多的功能。

## 6.1 类和对象

类：可以看成一个模板

对象：表示具体的事物

### 6.1.1 定义类

0）回顾：Java中的类

如果类是public的，则必须和文件名一致。

一般，一个.java有一个public类

注意：Scala中没有public，一个.scala中可以写多个类。

1）基本语法

**[修饰符] class 类名 {**

**类体**

**}**

说明

（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）

（2）一个Scala源文件可以包含多个类

2）案例实操

package com.atguigu.chapter06

**//（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）**

class Person {

}

**//（2）一个Scala源文件可以包含多个类**

class Teacher{

}

### 6.1.2 属性和封装

属性是类的一个组成部分。

封装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。Java封装操作如下，

（1）将属性进行私有化

（2）提供一个公共的set方法，用于对属性赋值

（3）提供一个公共的get方法，用于获取属性的值

Scala中的public属性，底层实际为private，并通过get方法（obj.field()）和set方法（obj.field\_=(value)）对其进行操作。所以Scala并不推荐将属性设为private，再为其设置public的get和set方法的做法。但由于很多Java框架都利用反射调用getXXX和setXXX方法，有时候为了和这些框架兼容，也会为Scala的属性设置getXXX和setXXX方法（通过@BeanProperty注解实现）。

1）基本语法

**[修饰符] var|val 属性名称 [：类型] = 属性值**

**注：**Bean属性（@BeanPropetry），可以自动生成规范的setXxx/getXxx方法

2）案例实操

package com.atguigu.scala.test

import scala.beans.BeanProperty

object Test02\_Class {

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

val person0 = new Person02

val student0 = new Student02

person0.age = 10

person0.getName

person0.getAge

person0.setAge(20)

}

}

**// scala中一个文件可以由多个public的类**

// 默认的访问权限就是public

// 但是同一个包内部只能有一个相同名称的类

class Person02{

// 属性分两种

// 可变和不可变

// 不推荐大家自己写封装的东西

// 因为scala可以通过修改属性的val 和var 来控制读写权限

@BeanProperty

var age:Int = \_

**// val 只能生成get方法**

@BeanProperty

val name:String = "zhangsan"

}

class Student02{

}

### 6.1.3 访问权限

1）说明

在Java中，访问权限分为：public， protected，默认,和private 。在Scala中，你可以通过类似的修饰符达到同样的效果。但是使用上有区别。

（1）Scala 中属性和方法的默认访问权限为public，但Scala中无public关键字。

（2）private为私有权限，只在类的内部和伴生对象中可用。

（3）protected为受保护权限，Scala中受保护权限比Java中更严格，同类、子类可以访问，同包无法访问。

（4）private[包名]增加包访问权限，包名下的其他类也可以使用

2）案例实操

package chapter06

object Test04\_Access {

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

**// 同一个包都可以访问的到**

Person04.name1

**// 受保护的权限 同一个包也无法访问**

// Person04.name2

**// 访问公共的权限**

Person04.name3

}

}

class Person04{

val nameClass = Person04.name

val name1Class = Person04.name1

**// 受保护的权限**

protected val name2:String = "受保护的权限"

}

object Person04{

**// 私有的权限能够在当前类和当前伴生对象中调用**

private val name:String = "私有权限"

**// 包访问权限**

private[chapter06] val name1:String = "包访问权限"

**// public的权限**

val name3:String = "公共的权限"

}

不同包的调用:

package chapter06Test

import chapter06.Person04

/\*\*

\* @author yhm

\* @create 2021-09-15 16:11

\*/

object Test04\_Access {

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

**// 不同的包里面无法访问name1 包访问权限**

// Person04.name1

**// 不同的包也能访问到公共的权限**

Person04.name3

}

}

class Student04 extends Person04{

**// 即使不是一个包 继承的子类也能够访问到受保护的权限**

val name2Class = name2

}

### 6.1.4 方法

1）基本语法

**def 方法名(参数列表) [：返回值类型] = {**

**方法体**

**}**

2）案例实操

class Person {

def sum(n1:Int, n2:Int) : Int = {

n1 + n2

}

}

object Person {

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

val person = new Person()

println(person.sum(10, 20))

}

}

### 6.1.5 构造器

和Java一样，Scala构造对象也需要调用构造方法，并且可以有任意多个构造方法。

Scala类的构造器包括：**主构造器和辅助构造器**

1）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...

}

}

说明：

（1）辅助构造器，函数的名称this，可以有多个，编译器通过参数的个数及类型来区分。

（2）辅助构造方法不能直接构建对象，必须直接或者间接调用主构造方法。

（3）构造器调用其他另外的构造器，要求被调用构造器必须提前声明。

2）案例实操

（1）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。

package chapter06

object Test06\_Constructor {

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

val person0 = new Person06("zhangsan")

val person01 = new Person06()

println(person01.name1)

val person02 = new Person06("lisi", 18)

}

}

**// 主构造器 直接写在类的定义后面 可以添加参数 可以使用权限修饰符**

//class Person06 private(name:String){

// val name1 = name

//}

class Person06 (name:String){

println("调用主构造器")

val name1 = name

var age:Int = \_

**// 两个辅助构造器 再互相调用的时候 只能是下面的辅助构造器调用上面的辅助构造器**

def this(){

**// 辅助构造器的第一行 必须直接或简介的调用主构造器**

**// 直接调用主构造器**

this("zhangsan")

println("调用辅助构造器1")

}

def this(name:String,age1:Int){

**// 间接调用主构造器**

this()

this.age = age1

println("调用辅助构造器2")

}

}

### 6.1.6 构造器参数

1）说明

Scala类的主构造器函数的形参包括三种类型：未用任何修饰、var修饰、val修饰

（1）未用任何修饰符修饰，这个参数就是一个局部变量，底层有属性的特性

（2）var修饰参数，作为类的成员属性使用，可以修改

（3）val修饰参数，作为类只读属性使用，不能修改

2）案例实操

package chapter06

object Test07\_ConstructorArgs {

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

val person0 = new Person07("zhangsan",11,"男")

println(person0.name)

println(person0.age)

println(person0.sex)

}

}

**// 主构造器参数 分为3类:**

// 没有修饰符 : 作为构造方法中的传入参数使用

// val 修饰 : 会自动生产同名的属性 并且定义为val

// var 修饰 : 会自动生产同名的属性 并且定义为var

class Person07 (name1:String,val age:Int,var sex:String){

val name = name1

// val age = age

// var sex = sex

}

### 6.1.7 scala的object

java中存在静态属性、静态方法、非静态属性、非静态方法。

scala当中不存在静态与非静态。object中定义的所有属性与方法、函数,除开private修饰的，都可以通过对象名.属性、对象名.方法、对象名.函数 的方式调用，可以理解为java中的static修饰的。

object ObjectTest {

val name:String = "zhangsan"

var age:Int = 30

private val address:String = "深圳"

def getName():String = {

this.hello+" "+this.name

}

val func=(x:Int,y:Int)=>{

x \* y

}

}

object Test {

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

//**object中的属性直接通过 类名.属性名 方式调用**

println(ObjectTest.name)

println(ObjectTest.age)

//设置属性

ObjectTest.age = 66

println(ObjectTest.age)

//**object中的方法直接通过 类名.方法名 方式调用**

println(ObjectTest.getName())

//**object中的方法直接通过 类名.函数名 方式调用**

println(ObjectTest.func(2,3))

}

}

Scala语言是完全面向对象的语言，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，该对象为**单例对象**。若单例对象名与类名一致，则称该单例对象这个类的**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。

### 6.1.8伴生类与伴生对象

1、如果有一个class,另外还有一个object,并且二者同名

2、class与object在同一个文件中

如果满足上两个条件，那么就称这个object为class的伴生对象，称class为object的伴生类

伴生类与伴生对象可以互相访问对方的私有成员

3）案例实操

class ClassObjectTest {

val name:String = "lisi"

//**用private修饰的只能在类或者伴生对象中使用**

private val age = 20

//**此时可以调用伴生对象中用private修饰的 address属性**

def getAddress() = ClassObjectTest.address

}

object ClassObjectTest{

private val address = "shenzhen"

def getName() = {

//创建伴生类的对象

val obj = new ClassObjectTest()

//**此时可以调用伴生类中用private修饰的name属性**

obj.name

}

}

### 6.1.9 apply方法

1）说明

（1）通过伴生对象的apply方法，实现不使用new方法创建对象。

（2）如果想让主构造器变成私有的，可以在()之前加上private。

（3）apply方法可以重载。

（4）Scala中**obj（arg）**的语句实际是在调用该对象的**apply**方法，即obj.apply（arg）。用以统一面向对象编程和函数式编程的风格。

（5）当使用new关键字构建对象时，调用的其实是类的构造方法，当直接使用类名构建对象时，调用的其实时伴生对象的apply方法。

2）案例实操

package chapter06

object Test11\_Apply {

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

// val person1 = new Person11

val person1: Person11 = Person11.getPerson11

**// 如果调用的方法是apply的话 方法名apply可以不写**

val person11: Person11 = Person11()

val zhangsan: Person11 = Person11("zhangsan")

**// 类的apply方法调用**

person11()

}

}

class Person11 private() {

var name:String = \_

def this(name:String){

this()

this.name = name

}

def apply(): Unit = println("类的apply方法调用")

}

object Person11 {

**// 使用伴生对象的方法来获取对象实例**

def getPerson11: Person11 = new Person11

**// 伴生对象的apply方法**

def apply(): Person11 = new Person11()

**// apply方法的重载**

def apply(name: String): Person11 = new Person11(name)

}

}

注意：也可以创建其它类型对象，并不一定是伴生类对象

### 6.1.10 类型检查和转换

1）说明

（1）obj.isInstanceOf[T]：判断obj是不是T类型。

（2）obj.asInstanceOf[T]：将obj强转成T类型。

（3）classOf获取类模板。

2）案例实操

object Test15\_Extends {

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

**// 判断对象的类型 以及转换对象的类型**

**// 只有多态会使用到**

val person1: Person15 = new Student15

// person1.sayHi()

**// 想要调用子类独有的属性和方法 需要对其进行转换 转换为对应的子类才行**

val student1: Student15 = person1.asInstanceOf[Student15]

student1.sayHi()

**// 即使在多态中 也会存在多个子类 不能直接转换 需要先判断**

if (person1.isInstanceOf[Teacher15]) {

val teacher1: Teacher15 = person1.asInstanceOf[Teacher15]

teacher1.sayHi1()

}

**// 调用固定的方法 返回类模板**

val value: Class[Student15] = classOf[Student15]

}

}

# 第7章 集合

## 7.1 集合简介

（1）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。

（2）对于几乎所有的集合类，Scala都同时提供了**可变**和**不可变**的版本，分别位于以下两个包。

不可变集合：scala.collection.**immutable**

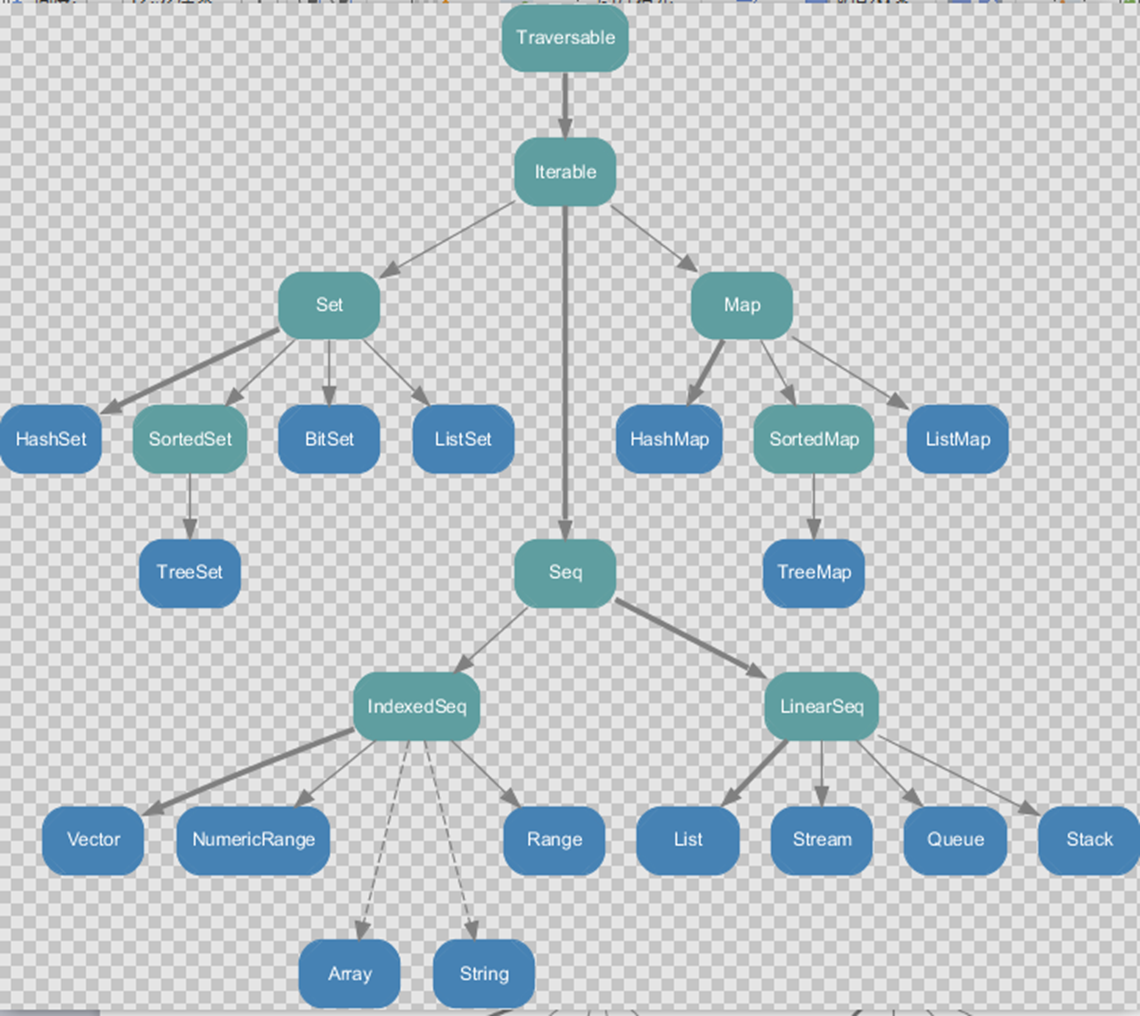
可变集合： scala.collection.**mutable**

（3）Scala不可变集合，就是指该集合对象不可修改，每次修改就会返回一个新对象，而不会对原对象进行修改。类似于java中的String对象。

（4）可变集合，就是这个集合可以直接对原对象进行修改，而不会返回新的对象。类似于java中StringBuilder对象。

**建议：在操作集合的时候，不可变用符号，可变用方法。**

### 7.1.1 不可变集合继承图



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Range。

4）String也是属于IndexedSeq。

5）我们发现经典的数据结构比如Queue和Stack被归属到LinearSeq（线性序列）。

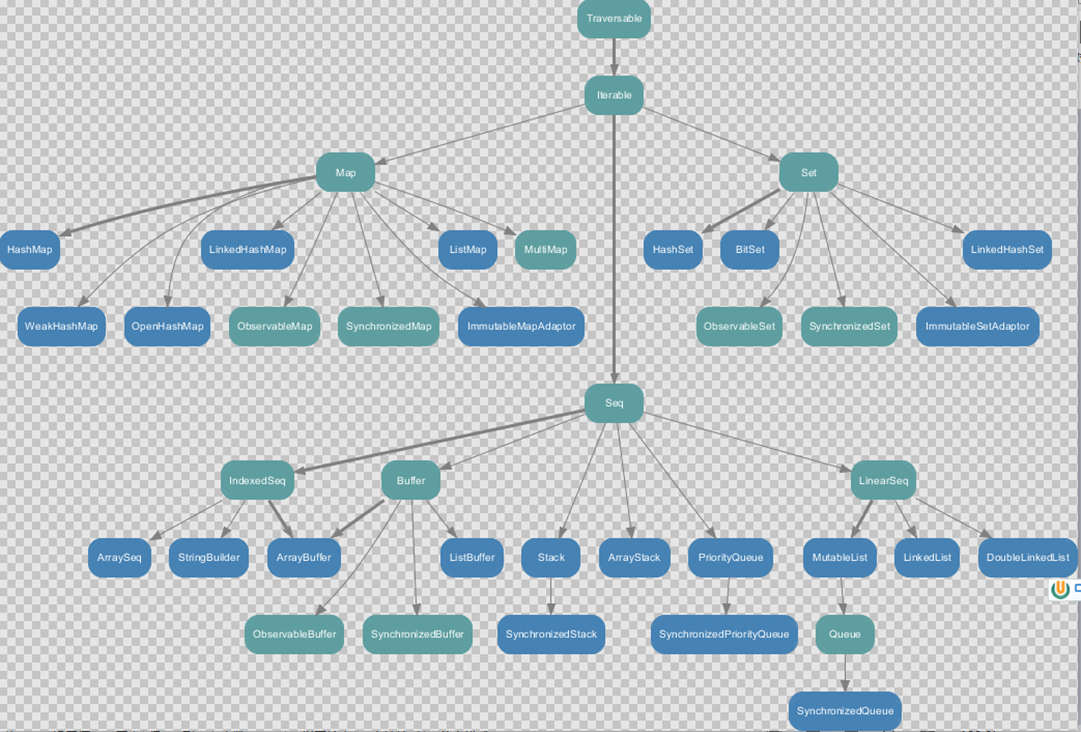
6）大家注意Scala中的Map体系有一个SortedMap，说明Scala的Map可以支持排序。

7）IndexedSeq和LinearSeq的区别。

（1）**IndexedSeq**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。

（2）**LinearSeq**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。

### 7.1.2 可变集合继承图



## 7.2 数组

### 7.2.1 不可变数组

1）第一种方式定义数组

定义：val arr1 = new Array[Int](10)

（1）new是关键字。

（2）[Int]是指定可以存放的数据类型，如果希望存放任意数据类型，则指定Any。

（3）(10)，表示数组的大小，确定后就不可以变化。

2）案例实操

package chapter07

object Test01\_Array {

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

**// 创建不可变数组**

val array = new Array[Int](10)

**// 也可以使用伴生对象的apply方法**

val array1: Array[Int] = Array(1, 2, 3, 4)

**// 遍历读取array**

// println(array)

for (elem <- array1) {

println(elem)

}

**// 使用迭代器遍历数组**

val iterator: Iterator[Int] = array1.iterator

while(iterator.hasNext){

val i: Int = iterator.next()

println(i)

}

println("===========================")

**// scala函数式编程的写法**

def myPrint(i:Int):Unit = {

println(i)

}

**// 放入自定义出来的函数**

array1.foreach(myPrint)

**// 直接使用匿名函数**

array1.foreach( i => println(i \* 2) )

**// 最简单的打印形式 直接使用系统的函数**

array1.foreach(println)

**// 修改数组的元素**

println(array1(0))

array1(0) = 10

println(array1(0))

**// 添加元素**

**// array1保持不变**

val array2: Array[Int] = array1 :+ 1

}

}

### 7.2.2 可变数组

1）定义变长数组

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

（1）[Any]存放任意数据类型

（2）(3, 2, 5)初始化好的三个元素

（3）ArrayBuffer需要引入scala.collection.mutable.ArrayBuffer

2）案例实操

（1）ArrayBuffer是有序的集合。

（2）增加元素使用的是append方法()，支持可变参数。

package chapter07

import scala.collection.mutable

import scala.collection.mutable.ArrayBuffer

object Test02\_ArrayBuffer {

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

**// 可变数组**

**// 默认使用的集合都是不可变的**

**// 使用可变集合 需要自己提前导包**

val arrayBuffer: ArrayBuffer[Int] = new ArrayBuffer[Int]()

val arrayBuffer1: ArrayBuffer[Int] = ArrayBuffer(1, 2, 3, 4)

**// 向可变数组中添加元素**

arrayBuffer.append(10)

arrayBuffer1.appendAll(Array(1,2,3,4))

**// 遍历打印**

arrayBuffer.foreach(println)

arrayBuffer1.foreach(println)

println(arrayBuffer1)

**// 修改元素**

arrayBuffer1.update(0,100)

arrayBuffer1(1) = 200

println(arrayBuffer1)

**// 查看元素**

println(arrayBuffer1(0))

**// 删除元素**

arrayBuffer1.remove(0)

println(arrayBuffer1)

arrayBuffer1.remove(1,3)

println(arrayBuffer1)

}

}

### 7.2.3 不可变数组与可变数组的转换

1）说明

arr1.toBuffer //**不可变数组转可变数组**

arr2.toArray //**可变数组转不可变数组**

（1）arr2.toArray返回结果才是一个不可变数组，arr2本身没有变化。

（2）arr1.toBuffer返回结果才是一个可变数组，arr1本身没有变化。

2）案例实操

**// 可变数组和不可变数组的转换和关系**

**// 不可变**

val ints: Array[Int] = Array(1, 2, 3, 4)

**// 可变**

val ints1: ArrayBuffer[Int] = ArrayBuffer(5, 6, 7, 8)

**// 不可变的用符号**

val b: Array[Int] = ints :+ 1

ints.foreach(println)

b.foreach(println)

**// 可变的用方法**

ints1.append(1)

println(ints1)

val ints2: ArrayBuffer[Int] = ints1 :+ 2

println(ints1)

**// 可变数组转换为不可变数组**

val array: Array[Int] = ints1.toArray

// array.append

**// 不可变数组转可变数组**

**// 结果用多态表示**

val buffer: mutable.Buffer[Int] = ints.toBuffer

val buffer1: ArrayBuffer[Int] = buffer.asInstanceOf[ArrayBuffer[Int]]

buffer.append(1)

### 7.2.4 多维数组

1）多维数组定义

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）案例实操

package chapter07

object Test03\_ArrayDim {

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

**// 多维数组**

val arrayDim = new Array[Array[Int]](3)

arrayDim(0) = Array(1,2,3,4)

arrayDim(1) = Array(1,2,3,4)

arrayDim(2) = Array(1,2,3,4)

for (array <- arrayDim) {

for (elem <- array) {

print(elem + "\t")

}

println()

}

**// scala中的方法**

val arrayDim1: Array[Array[Int]] = Array.ofDim[Int](3, 4)

arrayDim1(0)(1) = 100

for (array <- arrayDim1) {

for (elem <- array) {

print(elem + "\t")

}

println()

}

}

}

## 7.3 Seq集合（List）

### 7.3.1 不可变List

1）说明

（1）List默认为不可变集合

（2）创建一个List（数据有顺序，可重复）  
（3）遍历List

（4）List增加数据

（5）集合间合并：将一个整体拆成一个一个的个体，称为扁平化  
（6）取指定数据  
（7）空集合Nil

2）案例实操

object Test04\_List {

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

**// （1）List默认为不可变集合**

**// （2）创建一个List（数据有顺序，可重复）**

val list: List[Any] = List(1,1,1, 1.0, "hello", 'c')

val list3 = List(1, 2, 3, 4)

**// （3）遍历List**

list.foreach(println)

**// （4）List增加数据**

val list1: List[Any] = list :+ 1

println(list1)

val list2: List[Int] = 2 :: list3

println(list2)

val list5: List[Any] = list2 :: list3

println(list5)

**// （5）集合间合并：将一个整体拆成一个一个的个体，称为扁平化**

val list4: List[Int] = list2 ::: list3

println(list4)

**// （6）取指定数据**

val i: Int = list4(0)

**// （7）空集合Nil**

val list6: List[Int] = 1 :: 2 :: 3 :: 4 :: Nil

}

}

### 7.3.2 可变ListBuffer

1）说明

（1）创建一个可变集合ListBuffer

（2）向集合中添加数据

（3）删除元素

（4）查看修改元素

2）案例实操

import scala.collection.mutable.ListBuffer

object Test05\_ListBuffer {

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

**// （ 1）可变list创建**

val listBuffer = new ListBuffer[Int]()

val listBuffer1: ListBuffer[Int] = ListBuffer(1, 2, 3, 4)

**// （ 2）增加元素**

listBuffer1.append(5)

listBuffer1.prepend(0)

println(listBuffer1)

**// （ 3）删除元素**

listBuffer1.remove(0)

println(listBuffer1)

**// （ 4）查看修改**

listBuffer1(0) = 1

}

}

## 7.4 Set集合

默认情况下，Scala使用的是不可变集合，如果你想使用可变集合，需要引用 scala.collection.mutable.Set包。

### 7.4.1 不可变Set

1）说明

（1）Set默认是不可变集合

（2）数据无序不可重复

（3）默认使用hash set

2）案例实操

object Test06\_Set {

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

**// (1) 创建set 使用伴生对象的apply方法**

val set: Set[Int] = Set(4, 3, 2, 1)

val set1 = Set(1, 2, 3, 4, 2, 8, 4, 3, 7)

**// (2) set的特点 无序不可重复**

println(set)

**// (3) 默认使用hash set**

**// 如果元素少于等于4个 会创建特定类型的set**

println(set.isInstanceOf[HashSet[Int]])

val hashSet: HashSet[Int] = HashSet(1, 2, 3, 4, 5)

**// 不可变使用符号**

val set2: Set[Int] = set + 1

println(set2)

**// 作用 判断集合是否包含某个元素**

val bool: Boolean = set.contains(2)

}

}

### 7.4.2 可变mutable.Set

1）说明

（1）创建可变集合mutable.Set

（2）集合添加元素

（3）删除数据

2）案例实操

object Test06\_Set {

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

// (1)可变的set

val set3: mutable.Set[Int] = mutable.Set(1, 2, 3, 4, 4, 3, 2, 1)

**// 同样数据不可重复且无序**

println(set3)

**// (2) 添加元素**

**// 会使用返回值来告诉你有没有加入进去**

val bool1: Boolean = set3.add(5)

println(set3)

**// 遍历查询set**

set3.foreach(println)

**// (3)删除元素 填写的不是下标是删除的元素**

val bool2: Boolean = set3.remove(3)

println(set3)

}

}

## 7.5 Map集合

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。

### 7.5.1 不可变Map

1）说明

（1）创建不可变集合Map

（2）循环打印

（3）读取数据

2）案例实操

object Test07\_Map {

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

**// (1) 创建不可变map**

val map: Map[String, Int] = Map("hello" -> 1, "world" -> 2)

val map1 = Map(("hello", 1), ("world", 2))

**// (2) 遍历打印map**

for (elem <- map) {

println(elem)

}

map.foreach(println)

val keys: Iterable[String] = map.keys

keys.foreach(println)

val values: Iterable[Int] = map.values

**// 直接打印map**

println(map)

**// key是无序不可重复的**

val map2 = Map( ("z", 3),("a", 1), ("a", 2), ("c", 3),("f",4),("d",5))

println(map2)

**// (3) 获取value的值**

val option: Option[Int] = map2.get("a")

println(option)

if (!map2.get("m").isEmpty) {

val value: Int = map2.get("m").get

}

**// option有区分是否有数据的方法 使用getOrElse 如果为None 去默认值**

option.getOrElse(1)

**// 如果不确认存在**

val i: Int = map2.getOrElse("m", 10)

**// 如果确认存在的话**

val i1: Int = map2("a")

}

}

### 7.5.2 可变Map

1）说明

（1）创建可变集合

（2）向集合增加数据

（3）修改数据

（4）删除数据

2）案例实操

object Test07\_Map {

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

**// (1) 创建可变map**

val map3: mutable.Map[String, Int] = mutable.Map(("z", 3), ("a", 1), ("a", 2), ("c", 3), ("f", 4), ("d", 5))

**// (2) 可变map可以使用put方法放入元素**

map3.put("z",10)

println(map3)

**// (3) 修改元素的方法**

map3.update("z",20)

map3("z") = 30

**// (4) 删除元素**

map3.remove("z")

}

}

## 7.6 元组

1）说明

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。

注意：元组中最大只能有22个元素。

2）案例实操

（1）声明元组的方式：（元素1，元素2，元素3）。

（2）访问元组。

（3）Map中的键值对其实就是元组,只不过元组的元素个数为2，称之为对偶。

object TestTuple {  
  
 def main(args: Array[String]): Unit = {  
  
 **//（1）声明元组的方式：(元素1，元素2，元素3)** val tuple: (Int, String, Boolean) = (40,"bobo",true)  
  
 **//（2）访问元组  
 //（2.1）通过元素的顺序进行访问，调用方式：\_顺序号** println(tuple.\_1)  
 println(tuple.\_2)  
 println(tuple.\_3)  
  
 **//（2.2）通过索引访问数据** println(tuple.productElement(0))  
  
 **//（2.3）通过迭代器访问数据** for (elem <- tuple.productIterator) {  
 println(elem)  
 }  
  
 **//（3）Map中的键值对其实就是元组,只不过元组的元素个数为2，称之为对偶** val map = Map("a"->1, "b"->2, "c"->3)

val map1 = Map(("a",1), ("b",2), ("c",3))  
  
 map.foreach(tuple=>{println(tuple.\_1 + "=" + tuple.\_2)})  
 }  
}

## 7.7 集合常用函数

### 7.7.1 基本属性和常用操作

1）说明

（1）获取集合长度

（2）获取集合大小

（3）循环遍历

（4）迭代器

（5）生成字符串

（6）是否包含

2）案例实操

object TestList {

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

val list: List[Int] = List(1, 2, 3, 4, 5, 6, 7)

**//（1）获取集合长度**

println(list.length)

**//（2）获取集合大小,等同于length**

println(list.size)

**//（3）循环遍历**

list.foreach(println)

**//（4）迭代器**

for (elem <- list.itera tor) {

println(elem)

}

**//（5）生成字符串**

println(list.mkString(","))

**//（6）是否包含**

println(list.contains(3))

}

}

### 7.7.2 衍生集合

1）说明

（1）获取集合的头

（2）获取集合的尾（不是头的就是尾）

（3）集合最后一个数据

（4）集合初始数据（不包含最后一个）

（5）反转

（6）取前（后）n个元素

（7）去掉前（后）n个元素

（8）并集

（9）交集

（10）差集

（11）拉链

（12）滑窗

2）案例实操

object TestList {

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

val list1: List[Int] = List(1, 2, 3, 4, 5, 6, 7)

val list2: List[Int] = List(4, 5, 6, 7, 8, 9, 10)

**//（1）获取集合的头**

println(list1.head)

**//（2）获取集合的尾（不是头的就是尾）**

println(list1.tail)

**//（3）集合最后一个数据**

println(list1.last)

**//（4）集合初始数据（不包含最后一个）**

println(list1.init)

**//（5）反转**

println(list1.reverse)

**//（6）取前（后）n个元素**

println(list1.take(3))

println(list1.takeRight(3))

**//（7）去掉前（后）n个元素**

println(list1.drop(3))

println(list1.dropRight(3))

**//（8）并集**

println(list1.union(list2))

**//（9）交集**

println(list1.intersect(list2))

**//（10）差集**

println(list1.diff(list2))

**//（11）拉链 注:如果两个集合的元素个数不相等，那么会将同等数量的数据进行拉链，多余的数据省略不用**

println(list1.zip(list2))

**//（12）滑窗**

list1.sliding(2, 5).foreach(println)

}

}

### 7.7.3 集合计算初级函数

1）说明

（1）求和

（2）求乘积

（3）最大值

（4）最小值

（5）排序

2）实操

object Test11\_LowFunc {

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

val list: List[Int] = List(1, 5, -3, 4, 2, -7, 6)

val list1: ListBuffer[Int] = ListBuffer(1, 5, -3, 4, 2, -7, 6)

**// （1）求和**

val sum: Int = list.sum

println(sum)

**// （2）求乘积**

val product: Int = list.product

println(product)

**// （3）最大值**

val max: Int = list.max

**// （4）最小值**

val min: Int = list.min

**// （5）排序**

val sorted: List[Int] = list.sorted

println(list)

println(sorted)

**// 修改排序规则 从大到小**

val ints: List[Int] = list.sorted(Ordering[Int].reverse)

println(ints)

**// 对元组进行排序**

val tuples = List(("hello", 10), ("world", 2), ("scala", 9), ("haha", 4),("hello", 1))

**// 按照元组的默认字典序排列**

val sorted1: List[(String, Int)] = tuples.sorted

println(sorted1)

**// 按照后面数字从小到大排序**

val tuples1: List[(String, Int)] = tuples.sortBy((tuple: (String, Int)) => tuple.\_2)

println(tuples1)

**// 按照后面数字从大到小排序**

val tuples2: List[(String, Int)] = tuples.sortBy((tuple: (String, Int)) => tuple.\_2)(Ordering[Int].reverse)

println(tuples2)

tuples.sortBy( \_.\_2 )

**// 自定义排序规则**

val tuples3: List[(String, Int)] = tuples.sortWith((left: (String, Int), right: (String, Int)) => left.\_2 > right.\_2)

println(tuples3)

val tuples4: List[(String, Int)] = tuples.sortWith(\_.\_2 > \_.\_2)

println(tuples4)

}

}

1）sorted

对一个集合进行自然排序，通过传递隐式的Ordering。

2）sortBy

对一个属性或多个属性进行排序，通过它的类型。

3）sortWith

基于函数的排序，通过一个comparator函数，实现自定义排序的逻辑。

### 7.7.4 集合计算高级函数

1）说明

（1）过滤

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。

（2）**转化/映射（map）**

将集合中的每一个元素映射到某一个函数。

（3）扁平化

（4）扁平化+映射 注：flatMap相当于先进行map操作，在进行flatten操作。

集合中的每个元素的子元素映射到某个函数并返回新集合。

（5）**分组（groupBy）**

**按照指定的规则对集合的元素进行分组。**

（6）简化（归约）

（7）折叠

2）实操

object TestList {

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

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

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

val wordList: List[String] = List("hello world", "hello atguigu", "hello scala")

**//（1）过滤**

println(list.filter(x => x % 2 == 0))

**//（2）转化/映射**

println(list.map(x => x + 1))

**//（3）扁平化**

println(nestedList.flatten)

**//（4）扁平化+映射 注：flatMap相当于先进行map操作，在进行flatten操作**

println(wordList.flatMap(x => x.split(" ")))

**//（5）分组**

println(list.groupBy(x => x % 2))

}

}

**3）Reduce方法**

Reduce简化（归约） ：通过指定的逻辑将集合中的数据进行聚合，从而减少数据，最终获取结果。

案例实操。

object TestReduce {  
  
 def main(args: Array[String]): Unit = {  
  
 val list = List(1,2,3,4)  
  
 **// 将数据两两结合，实现运算规则** val i: Int = list.reduce( (x,y) => x-y )  
 println("i = " + i)  
  
 **// 从源码的角度，reduce底层调用的其实就是reduceLeft** //val i1 = list.reduceLeft((x,y) => x-y)  
  
 **// ((4-3)-2-1) = -2** val i2 = list.reduceRight((x,y) => x-y)  
 println(i2)  
 }  
}

**4）Fold方法**

Fold折叠：化简的一种特殊情况，可以添加初始值

（1）案例实操：fold基本使用

object TestFold {

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

val list = List(1,2,3,4)

**// fold方法使用了函数柯里化，存在两个参数列表**

**// 第一个参数列表为 ： 零值（初始值）**

**// 第二个参数列表为： 简化规则**

**// fold底层其实为foldLeft**

val i = list.foldLeft(1)((x,y)=>x-y)

val i1 = list.foldRight(10)((x,y)=>x-y)

println(i)

println(i1)

}

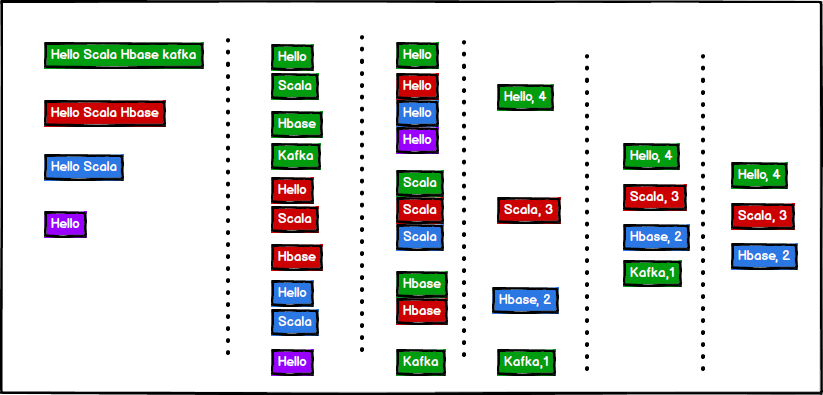
}

### 7.7.5 WordCount案例

1）需求

单词计数：将集合中出现的相同的单词，进行计数，取计数排名前三的结果。

2）需求分析



3）案例实操

object TestWordCount {  
  
 def main(args: Array[String]): Unit = {  
  
 **// 单词计数：将集合中出现的相同的单词，进行计数，取计数排名前三的结果** val stringList = List("Hello Scala Hbase kafka", "Hello Scala Hbase", "Hello Scala", "Hello")  
  
 **// （1）将每一个字符串转换成一个一个单词** val wordList: List[String] = stringList.flatMap(str=>str.split(" "))  
 //println(wordList)  
  
 **// （2）将相同的单词放置在一起** val wordToWordsMap: Map[String, List[String]] = wordList.groupBy(word=>word)  
 //println(wordToWordsMap)  
  
 **// （3）对相同的单词进行计数** // (word, list) => (word, count)  
 val wordToCountMap: Map[String, Int] = wordToWordsMap.map(tuple=>(tuple.\_1, tuple.\_2.size))  
  
 **//（4）对计数完成后的结果进行排序（降序）** val sortList: List[(String, Int)] = wordToCountMap.toList.sortWith {  
 (left, right) => {  
 left.\_2 > right.\_2  
 }  
 }  
  
 **// （5）对排序后的结果取前3名** val resultList: List[(String, Int)] = sortList.take(3)  
  
 println(resultList)  
 }  
}

## 7.8 队列

1）说明

Scala也提供了队列（Queue）的数据结构，队列的特点就是先进先出。进队和出队的方法分别为enqueue和dequeue。

2）案例实操

object TestQueue {  
  
 def main(args: Array[String]): Unit = {  
  
 val que = new mutable.Queue[String]()  
  
 que.enqueue("a", "b", "c")  
  
 println(que.dequeue())  
 println(que.dequeue())  
 println(que.dequeue())  
 }  
}

# 第8章 模式匹配

Scala中的模式匹配类似于Java中的switch语法

int i = 10

switch (i) {

case 10 :

System.out.println("10");

break;

case 20 :

System.out.println("20");

break;

default :

System.out.println("other number");

break;

}

但是scala从语法中补充了更多的功能，所以更加强大。

## 8.1 基本语法

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。

object TestMatchCase {

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

var a: Int = 10

var b: Int = 20

var operator: Char = 'd'

var result = operator **match** {

case '+' => a + b

case '-' => a - b

case '\*' => a \* b

case '/' => a / b

case \_ => "illegal"

}

println(result)

}

}

1）说明

（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，若此时没有case \_ 分支，那么会抛出MatchError。

（2）每个case中，不需要使用break语句，自动中断case。

（3）match case语句可以匹配任何类型，而不只是字面量。

（4）=> 后面的代码块，直到下一个case语句之前的代码是**作为一个整体执行**，可以使用{}括起来，也可以不括。

## 8.2 模式匹配常见用法

### 8.2.1 匹配类型

1）说明

需要进行类型判断时，可以使用前文所学的isInstanceOf[T]和asInstanceOf[T]，也可使用模式匹配实现同样的功能。

2）案例实操

object Test02\_MatchValue {

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

**// 匹配类型**

def func2(x:Any):String ={

x match {

case i:Int => "整数"

case c:Char => "字符"

case s:String => "字符串"

case \_ => "其他"

}

}

println(func2(1515))

println(func2('\t'))

println(func2("1515"))

}

}

### 8.2.2 匹配元组

object TestMatchTuple {

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

//对一个元组集合进行遍历

for (tuple <- Array((0, 1), (1, 0), (1, 1), (1, 0, 2))) {

val result = tuple match {

case (0, \_) => "0 ..." //是第一个元素是0的元组

case (y, 0) => "" + y + "0" // 匹配后一个元素是0的对偶元组

case (a, b) => "" + a + " " + b

case \_ => "something else" //默认

}

println(result)

}

}

}

### 8.2.3 匹配对象及样例类

1）基本语法

object Test05\_MatchObject {

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

val zhangsan = new Person05("zhangsan", 18)

zhangsan match {

case Person05("zhangsan",18) => println("找到张三啦")

case \_ => println("你不是zhangsan")

}

}

}

class Person05 (val name:String,var age:Int){

}

object Person05{

**// 创建对象的方法**

def apply(name: String, age: Int): Person05 = new Person05(name, age)

**// 解析对象的方法**

def unapply(arg: Person05): Option[(String, Int)] = {

**// 如果解析的参数为null**

if (arg == null ) None else Some((arg.name,arg.age))

}

}

小结

* val user = Person05("zhangsan",11)，该语句在执行时，实际调用的是Person05伴生对象中的**apply**方法，因此不用**new关键字**就能构造出相应的对象。
* 当将Person05 **("zhangsan", 11)**写在case后时[case User("zhangsan", 11) => "yes"]，会默认调用**unapply**方法(对象提取器)，**user作为unapply方法的参数**，unapply方法将user对象的name和age属性提取出来，与User("zhangsan", 11)中的属性值进行匹配
* case中对象的unapply方法(提取器)返回Some，且所有属性均一致，才算匹配成功,属性不一致，或返回None，则匹配失败。
* 若只提取对象的一个属性，则提取器为**unapply**(obj:Obj):**Option[T]**

若提取对象的多个属性，则提取器为**unapply**(obj:Obj):**Option[(T1,T2,T3…)]**

若提取对象的可变个属性，则提取器为**unapplySeq**(obj:Obj):**Option[Seq[T]]**

**2）样例类**

（1）语法：

**case class** Person05 (name: String, age: Int)

（2）说明

样例类仍然是类，和普通类相比，只是其自动生成了伴生对象，并且伴生对象中自动提供了一些常用的方法，如**apply**、**unapply**、**toString**、equals、hashCode和copy。

样例类是为模式匹配而优化的类，因为其默认提供了unapply方法，因此，样例类可以直接使用模式匹配，而无需自己实现unapply方法。

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）

（3）实操

上述匹配对象的案例使用样例类会节省大量代码

case class Person05(var name: String, age: Int)

## 8.3 偏函数中的模式匹配

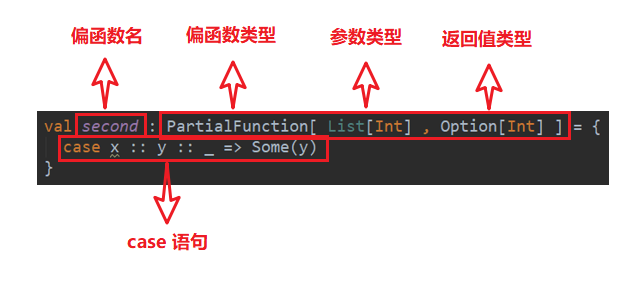
偏函数也是函数的一种，通过偏函数我们可以方便的对输入参数做更精确的检查。例如该偏函数的输入类型为List[Int]，而我们需要的是第一个元素是0的集合，这就是通过模式匹配实现的。

1. **偏函数定义**

val second: **PartialFunction**[List[Int], Option[Int]] = {

case x :: y :: \_ => Some(y)

}



**注**：该偏函数的功能是返回输入的List集合的第二个元素。

**2）偏函数原理**

上述代码会被scala编译器翻译成以下代码，与普通函数相比，只是多了一个用于参数检查的函数——isDefinedAt，其返回值类型为Boolean。

val second = new PartialFunction[List[Int], Option[Int]] {

**//检查输入参数是否合格**

override def isDefinedAt(list: List[Int]): Boolean = list match {

case x :: y :: \_ => true

case \_ => false

}

**//执行函数逻辑**

override def apply(list: List[Int]): Option[Int] = list match {

case x :: y :: \_ => Some(y)

}

}

**3）偏函数使用**

偏函数不能像second(List(1,2,3))这样直接使用，因为这样会直接调用apply方法，而应该调用applyOrElse方法，如下

second.applyOrElse(List(1,2,3), (\_: List[Int]) => None)

applyOrElse方法的逻辑为 **if** (**ifDefinedAt(**list**)**) **apply(**list**)** else default。如果输入参数满足条件，即isDefinedAt返回true，则执行apply方法，否则执行defalut方法，default方法为参数不满足要求的处理逻辑。

1. **案例实操**

**（1）需求**

将该List(1,2,3,4,5,6,"test")中的Int类型的元素加一，并去掉字符串。

**（2）实操**

object Test06\_PartitionFunc {

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

**// 将该List(1,2,3,4,5,6,"test")中的Int类型的元素加一，并去掉字符串。**

val list = List(1, 2, 3, 4, 5, 6, "test")

**// 步骤一: 过滤掉字符串**

val list1: List[Any] = list.filter((a: Any) => a match {

case s: String => false

case i: Int => true

})

**// 步骤二: 对int值加一**

val list2: List[Int] = list1.map((a: Any) => {

a match {

case i: Int => i + 1

}

})

println(list2)

val list3: List[Int] = list.collect({

case i: Int => i + 1

})

println(list3)

val value:PartialFunction[Any, Int] = {

case i: Int => i + 1

}

**// 函数的定义 需要多写一个math关键字**

**// 偏函数将match关键字省略**

val function: Any => Int = (a: Any) => a match {

case i: Int => i + 1

}

}

}

方法一：

List(1,2,3,4,5,6,"test").**filter**(\_.isInstanceOf[Int]).**map**(\_.asInstanceOf[Int] + 1).foreach(println)

方法二：

List(1, 2, 3, 4, 5, 6, "test").**collect** **{ case x: Int => x + 1 }**.foreach(println)

## 8.4 下划线的使用总结

附加内容：下划线的使用总结。

（1）用于类中的var属性，使用默认值。

（2）用于高阶函数的第一种用法，表示函数自身。

（3）匿名函数化简，用下划线代替变量。

（4）用于导包下的所有内容。

（5）用于起别名时表示匿名。

（6）用于模式匹配表示任意数据。

object Test01\_Extends {

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

**// 下滑线的使用总结**

**// （2）用于高阶函数的第一种用法，表示函数自身**

def sayHi(name: String): Unit = {

println(s"hi $name")

}

val function: String => Unit = sayHi \_

**// （3）匿名函数化简，用下划线代替变量**

val function1: (Int, Int) => Int = (a: Int, b: Int) => a + b

val function2: (Int, Int) => Int = \_ + \_

**// （4）用于导包下的所有内容**

import scala.util.control.Breaks.\_

**// （5）用于起别名时表示匿名**

import scala.util.control.{Breaks => \_}

// Breaks

**// （6）用于模式匹配表示任意数据**

10 match {

case 10 => println(10)

case \_ => println("其他数据")

}

}

}

class Person01 {

**// （1）用于类中的var属性，使用默认值**

var name: String = \_

}

# 第9章 异常

语法处理上和**Java类似**，但是又不尽相同。

## 9.1 Java异常处理

public class ExceptionDemo {

public static void main(String[] args) {

try {

int a = 10;

int b = 0;

int c = a / b;

}catch (ArithmeticException e){

// catch时，需要将范围小的写到前面

e.printStackTrace();

}catch (Exception e){

e.printStackTrace();

}finally {

System.out.println("finally");

}

}

}

注意事项

（1）Java语言按照try—catch—finally的方式来处理异常

（2）不管有没有异常捕获，都会执行finally，因此通常可以在finally代码块中释放资源。

（3）可以有多个catch，分别捕获对应的异常，这时需要把范围小的异常类写在前面，把范围大的异常类写在后面，否则编译错误。

## 9.2 Scala异常处理

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

try {

var n= 10 / 0

}catch {

case ex: ArithmeticException=>{

**// 发生算术异常**

println("发生算术异常")

}

case ex: Exception=>{

**// 对异常处理**

println("发生了异常1")

println("发生了异常2")

}

}finally {

println("finally")

}

}

（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）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。

（4）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。

（5）用throw关键字，抛出一个异常对象。所有异常都是Throwable的子类型。throw表达式是有类型的，就是Nothing，因为Nothing是所有类型的子类型，所以throw表达式可以用在需要类型的地方

def test():Nothing = {  
 throw new Exception("不对")  
}

（6）Java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注解来声明异常

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

# 第10章 隐式转换

**当编译器第一次编译失败的时候，会在当前的环境中查找能让代码编译通过的方法，用于将类型进行转换，实现二次编译，用于拓展类的方法。**

## 10.1 隐式函数

1）说明

隐式转换可以在不需改任何代码的情况下，扩展某个类的功能。

2）案例实操

需求：通过隐式转化为Int类型增加方法。

object Test02\_Imp {

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

**// 隐式函数**

**// 将当前作用域下所有传入参数的类型 隐式转换为 返回值类型**

implicit def changeInt(self: Int) = {

new MyRichInt(self)

}

val i: Int = 10

**// 比较自身和传入参数的大小 返回较大的值**

val value: Int = i.myMax(20)

println(value)

val i1: Int = i << 2

println(i1)

}

**// 隐式转换的目标**

class MyRichInt(val self: Int) {

def myMax(i: Int): Int = {

if (i > self) i else self

}

**// 如果隐式转换和自身的方法冲突 会使用它自身的 因为不会编译失败**

def <<(x: Int): Int = {

0

}

}

}