1. **scala** 基本输出

package com.dahua

/\*\*

\* 项目中 scala 第一个类

\*/

object ScalaDemo {

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

// println 输出

println("hello scala")

// 数据类型 byte short int long float double char boolean unit

// 大家可以使用 cmd 命令行进行查看。

var a = 10 println(a)

// unit 空类型。

val result = println("hello") println(result)

}

}

1. **Scala** 数据类型

package com.dahua

import java.text.SimpleDateFormat

/\*\*

\*

\*/

object ScalaDemo01 {

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

// scala 语法。

/\*

定义变量

var a = 10 定义变量，变量可变

val a = 10 定义变量，变量不可变 final

\*/

var name = "张三" println(name)

name = "李四" println(name)

val age = 18

// age = 19 不可修改变量值。

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

/\*\*

* 输出类型
* println
* printf()
* println(s)

\*/

var stuName = "jack" var stuAge = 19

var stuHige = 170.5d println("hello"+stuName)

println(f"姓名：${stuName}",s"年龄：${stuAge}",s"身高：${stuHige}")

printf("姓名：%s,年龄：%s,身高，%s",stuName,stuAge,stuHige)

// 初始化对象

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

var student = new Student("tom",20) println(s"姓名：${student.name}")

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

// 问题：字符串和基本数据类型转换。可以使用 Java 的 API. var num = "10"

var num1 = 20

var sum = Integer.parseInt(num)+num1

println(sum)

}

}

/\*

定义类

\*/

// 有参构造方法

class Student(var name:String,var age:Integer)

1. **scala** 运算符

package com.dahua

/\*\*

* 运算符
* 和 java 相同，但也有不同
* 可以使用 java 中的算数运算符进行计算，换算成 scala:

\* .+() .-() .\*() ./() .%()

* 比较运算符
* 自增自减运算符 ++ -- 不存在。量自增。
* 逻辑运算符

\*

\*

\*

\*/

object ScalaDemo02 {

替代是：

+= -= 运算符结果无返回值，变

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

// 例如使用 java 中算数运算符

var a = 10 var b = 3

// 进行加法运算符

var c = a+b println(c)

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

// 在 scala 中，算数运算符都是函数的表现： + var d = a.+(b)

println(d)

和

.+() 表达同一个意思即加法。

val i: Int = a.\*(10).+(15) println(i)

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

// 问题：自增自减运算符 java 和 scala 各自的编写方式：

var j = 10 j+=1

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

// 比较运算符 逻辑运算符

var num1 = 10 var num2 = 20

println(num1>num2 || num1<num2)

}

}

scala 分支结构

package com.dahua

/\*\*

\* if else

\*/

object ScalaDemo03 {

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

/\*

if else 语法

\*/

var a = 10 println(if(a==10)"正确")

// if else 语句

var result= if(a%2==0)"偶数" else "奇数" println(result)

// if else if 语句。

val res1= if(a%3==0)"3 的倍数" else if(a%5==0)"5 的倍数" println(res1)

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

// if else if 语句。

var res2 = if(a%3==0){ println("3 的倍数")

}else if(a%5==0){ 100

20

}

// 任何函数都可以返回值。 不需要 return.也不要加 return

// if else 的返回值，在{最后一行即为返回值}

println("if 函数的返回值："+res2) println("============================")

没有{}

第一行就是返回值。

/\*\*\

* any 是代表 scala 任意类型
* anyVal 代表任意数值类型
* anyRef 代表任意对象类型。

\*/

var res4:Any= if(a%3==0){ "偶数"

}else{ 0

}

println(res4)

}

}

1. **for** 循环

package com.dahua

/\*\*

\* for 循环

\*/

object ScalaDemo04 {

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

// for 循环语法。

for(ele <- 1 to 10 ){// 和 shell 的语法相似

print(ele+" ")

} println("+=============================")

// Until 语法 for 循环

for(a <- 1 until(10)) print(a+" ")

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

// 定义数组

val ints = Array(3,4,67,90,22,75)// for(ele <- ints)

print(ele+" ")// 相当于 forEach 循环

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

for(ele <- 0 until(ints.length)) print(ints(ele)+" ")

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

// 需求：显示出所有的偶数。 给 for 循环添加守卫。for(ele <- 0 until(ints.length) if(ints(ele)%2==0)) print(ints(ele)+" ")

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

// 需求：执行 for 循环。将 偶数 循环后添加到新的数组中。

var arr2 = for(e <- ints if(e%2==0)) yield e

for(e <- arr2)print(e+" ") println("+=============================")

// 需求： 嵌套循环

for(i <- 1 to 10 ;j <- 1 to 5 ; k <- 1 to 3) println(i+" "+j+" "+k)

// 作业： 打印 99 乘法表

// 作业： 冒泡算法

// 预习： scala 中的方法和函数 先进行预习。

}

}

package com.dahua1

/\*\*

* + scala 中有方法和函数
  + 方法：method。
  + def 方法名（参数列表）：返回值 ={ 执行体}

\*

* + 函数：function
  + val f1:(Int,Int) => Int ={(x,y)=>x+y}
  + val 函数名：(参数数据类型)=>返回值类型={(参数命名)=>函数执行体}

\*

* + 一定要看懂这样的代表表达的意思：
  + res0:（Int,Int）=> Int = （funciton）
  + （Int,Int）=> Int

\*

* + 方法可以转换成函数，函数也可以执行和方法相同的功能。

\*

\*

\*/

object ScalaDemo01 {

// 无参，无返回值方法

def m1(): Unit = { println("method1")

}

// 有参，有返回值方法。

def m2(a: Int, b: Int): Int = {

println("方法具体要干点啥……，执行业务。") a + b

}

// 方法的简写。不设定返回值类型，由方法自动推算。

def m3(x: Int, y: Int) = { "string"

}

// 函数

// val 函数名：(参数数据类型)=>返回值类型={(参数命名)=>函数执行体} val f1: (Int, Int) => Int = { (x, y) => x + y }

// 函数简化模式

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

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

// 调用方法 1。m1

// 调用方法 2。

val res0: Int = m2(3, 5) println(res0)

// 调用方法 3.

val res1: String = m3(10, 4) println(res1)

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

// 调用函数 f1 val res2 = f1(4, 6)

println("函数的执行：" + res2)

// 调用函数 f2() val res3 = f2(5,1) println(res3)

// 一会儿再说

val arrs = Array("1","2","3")

val res4: Array[String] = arrs.map(x=>x+"\*\*") println(res4.toBuffer)

// 方法可以转换成函数 使用 方法名 \_ val function: (Int, Int) => String = m3 \_

val res5: String = function(4,6) println(res5)

}

}

package com.dahua1

/\*\*

* 函数和方法有什么区别：
* 在方法中，可以将函数作为参数进行传递和调用

\*/

object ScalaDemo02 {

// 创建函数 乘法

val f1: (Int, Int) => Int = { (x, y) => x \* y }

// 创建函数 加法

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

// 创建函数 减法

val f3 = (x: Int, y: Int) => x - y

// 定义方法，方法中，定义函数

def cheng(f: (Int, Int) => Int, x: Int, y: Int): Int = {f(x, y)}

/\*\*

* def cheng( x: Int, y: Int): Int = (执行代码)
* def 方法名（x,y 参数）：Int 是返回值 = 具体的执行

\*

* 希望在方法中，传入一个函数，我要执行这个函数。

\*

* def cheng(f:(Int,Int)=>Int,x:Int,y:Int)={我要执行这个函数 f(x,y)}

\*

\*/

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

// 调用方法 执行乘法

val res0: Int = cheng(f1, 5, 6) println(res0)

// 调用方法 执行加法

val res1: Int = cheng(f2, 5, 6) println(res1)

// 调用方法 执行减法

val res2: Int = cheng(f3, 5, 6) println(res2)

}

}

package com.dahua1

/\*\*

\* 函数的传递

\*/

object ScalaDemo03 {

/\*\*

\* 花钱和数钱

\*/

var money = 50

// 花钱

def huaqian() = { money = money - 5

}

// 数钱

def shuqian() = { huaqian money

}

// 关于值传递

def printByVal(x: Int) = { for (ele <- 0 until (3)) {

println(s"余额：${x}")

}

}

// 关于函数传递 f:() => Int def printByFun(x: => Int) = { for (ele <- 0 until (3)) {

println(s"余额：${x}")

}

}

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

// 值传递方法的调用 在参数中，传递的是值。printByVal(shuqian) println("=====================")

// 在方法中传递函数，传递的是函数的执行体。

printByFun(shuqian)

}

}

# 数组

package com.dahua1

import scala.collection.mutable.ArrayBuffer

/\*\*

\* 数组，在 Scala 中，为不可变数组。所以，数组不能 new。

\*

\*/

object ScalaDemo04 {

val f1 = (x: Int) => x \* 10

// 传入数组

def map1(x: Array[Int]) = { var arr = ArrayBuffer[Int]() for (ele <- x) {

println(ele)

arr += (ele \* 10) // java 中的 append

}

// 返回一个数组。数组中的值，都乘以 10 arr.toArray

}

// 编写函数，完成数组中的值乘以 10 输出。 考虑：如果我不用 ArrayBuffer。就用 x:

Array[Int]能不能得到结果。

val f2: (Array[Int]) => Array[Int] = { x => var arr = ArrayBuffer[Int]()

for (ele <- x) { arr += ele \* 10

}

arr.toArray

}

// 编写匿名函数

var f3 = (x: Array[Int]) => { var arr = ArrayBuffer[Int]() for (ele <- x) {

arr += ele \* 10

}

arr.toArray

}

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

// 数组

var arrs = Array(1, 2, 3, 4, 5)

// 将数组中的每个元素乘以 10 val res0: Array[Int] = arrs.map(f1) println(res0.toBuffer)

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

val res1: Array[Int] = arrs.map((x: Int) => x \* 10) println(res1.toBuffer) println("================================")

val res2: Array[Int] = arrs.map(\_ \* 10) println(res2.toBuffer) println("================================")

val res3: Array[Int] = arrs.map(f1(\_)) println(res3.toBuffer)

// 我们能否自己写一个和 map 类似的方法。 传入每个元素。 返回值是数组，并且， 每个元素的值都乘以 10

val res5: Array[Int] = map1(arrs) println(res5.toBuffer)

}

}

1. 多参函数

package com.dahua1

/\*\*

\* 多参数函数

\*/

object ScalaDemo05 {

def m1(x: Int\*) = { var sum = 0

for (ele <- x) { sum += ele

}

sum

}

def main(args: Array[String]): Unit = { val sum: Int = m1(1,2,3,4,5) println(sum)

}

}

# 函数默认值

package com.dahua1

/\*\*

\* 函数的默认值

\*/

object ScalaDemo06 {

def add(x:Int=3,y:Int=5)={ x+y

}

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

// 调用有默认值的方法。val sum: Int = add() println(sum)

val res0: Int = add(10) println(res0)

val res1: Int = add(y=10) println(res1)

val res2: Int = add(10,20) println(res2)

}

}

1. 高阶函数

package com.dahua1

/\*\*

* 高阶函数：函数的函数
* 把一个函数，当做另一个函数的参数。高阶函数

\*/

object ScalaDemo07 {

def apply(f: Int => String, v: Int) = f(v) def layout(x: Int) = "[" + x.toString + "]"

var f1: (Int) => String = {x => "{" + x.toString + "}"}

def main(args: Array[String]): Unit = { val res0 = apply(f1, 10) println(res0)

}

}

# 柯里化函数

package com.dahua2

/\*\*

\* 柯里化函数

\*/

object Currying {

def add(x: Int, y: Int) = x + y def add1(a: Int)(b: Int) = a + b

// 柯里化函数定义

def klh(a:Int)=(b:Int) => a+b

def main(args: Array[String]): Unit = { val res0: Int = add(4,5) println(res0)

val res1: Int => Int = add1(3) val res2: Int = res1(5) println(res2)

val res3: Int => Int = klh(2) val res4: Int = res3(4) println(res4)

}

}

偏函数

package com.dahua2

/\*\*

* 偏函数
* 函数中的参数类型可容纳多种类型。处理过程中，根据类型进行处理。
* def f3:PartialFunction[Any,String]
* any 是输入函数的数据类型
* String 进行处理的数据类型以及返回的数据类型。

\*

* 部分参数应用函数
* 柯里化

\*/

object Partial {

def f1(a:Any): Int ={ if(a.equals("a"))97 else 0

}

// 偏函数

def f2:PartialFunction[Any,Int]={

case "a" => 97 // 根据值进行匹配

case "b" => 98 case \_ => 0

}

// 偏函数在数组集合中，能够发挥作用。

def f3:PartialFunction[Any,String]={

case i:String => "["+i+"]"// 根据类型匹配

// 模式匹配： i:输入参数根据 String 类型进行判定。如果是其他类型，不处理。

}

def main(args: Array[String]): Unit = { val res0: Int = f1(9)

println(res0)

// 调用偏函数val res1 = f2(3.15) println(res1)

// 定义数组

val array: Array[Any] = Array[Any](1,2,3,4,5,"hello","scala") val res2: Array[String] = array.collect(f3) println(res2.toBuffer)

}

}

# 数组函数练习

package com.dahua3

/\*\*

\* 数组的函数练习

\*/

object ArrayDemo {

def main(args: Array[String]): Unit = { val arrs = Array(1, 4, 67, 89, 23, 90, 16)

// map 方法。 对数组中所有的元素的映射。

val res0: Array[Int] = arrs.map(x => x \* 10) println(res0.toBuffer)

// 数组中的聚合函数

println(arrs.sum) println(arrs.max) println(arrs.min)

println(arrs.count(x => x % 2 == 0))

/\*\*

* map 每个元素的映射
* flatten 扁平化。切分压平。
* flatMap 对内部元素切分压平。
* groupby 分组。
* mapValues() 针对 map 中的 value 字段进行操作。
* toList 转换成 List 集合。
* sortBy 函数。进行排序。

\*/ println("====================================================")

val strs = Array("hello world hello jack hello tom", "hello tom world world jerry")

// 计算 wordCount x = "hello world hello jack hello tom" val res1: Array[Array[String]] = strs.map(x => x.split(" "))

// 希望得到 Array[String] Array(hello world hello jack hello tom,hello tom world world jerry)

val flatten: Array[String] = res1.flatten

//基于以上用到了，两个函数：map flatten scala 考虑到写两个函数较为复杂，提出了一个函数 flatMap

val res2: Array[String] = strs.flatMap(\_.split(" "))

// (hello, world, hello, jack, hello, tom, hello, tom, world, world, jerry) val res3: Map[String, Array[String]] = res2.groupBy(x => x)

// Map(jack -> Array(jack), world -> Array(world, world, world), tom -> Array(tom, tom), hello -> Array(hello, hello, hello, hello), jerry ->

val res4: Map[String, Int] = res3.mapValues(x => x.length)

// 可否进行排序。 可以。转换成 List 集合就能排序。val list: List[(String, Int)] = res4.toList.sortBy(x => -x.\_2) for (ele <- list) {

println(ele.\_1 + " " + ele.\_2)

}

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

// 简化开发流程。

// 考试： 就这一行代码。

// 面试题中： 你会写几种 wordCount?

strs.flatMap(\_.split(" ")).groupBy(x => x).mapValues(x => x.length).toList.sortBy(x => x.\_2).foreach(x => println(x))

}

}

1. **list** 集合

package com.dahua3

/\*\*

* List 集合在 Scala 中 Seq

\* 集合：

* ：mutable:包中的集合是可变集合。
* 表示集合：长度可变：值可变。

\*scala.collection.mutable.ListBuffer

\* \*

* immutable: 中的是不可变集合。
* 表示集合有序：长度不可变。 值不可变。 和 Java 有很大的不同。

\*scala.collection.immutable List

\*

\*/

object ListDemo {

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

val list = List(1, 2, 3, 4) // 以后 大家就明白，因为这个 List 集合，会是《Kafka 中的消息队列》

// 获取集合中的元素。println(list(0)) println(list(1))

// 向其内部添加元素：

// list(0) = 5//错误，不能 修改，或添加集合中的值。

// 遍历集合中的元素。

for (ele <- 0 until (list.size)) print(list(ele) + " ") println("====================")

// 获取集合头部元素

println("头部元素：" + list.head)

// 获取尾部元素

println("尾部元素" + list.tail.toBuffer)

// 使用数据拼接创建 序列 :: :: 拼接序列val res0: List[Int] = 4 :: 5 :: 6 :: 7 :: Nil println(res0.toBuffer)

// ++ 合并序列到尾部。

val res1: List[Int] = list.++(res0) println(res1.toBuffer)

// 原集合会变吗？

println(list.toBuffer)

// ++: 添加集合到新集合的头部。val res2: List[Int] = list.++:(res0) println(res2.toBuffer)

// +： 添加元素到头部val res3: List[Int] = list.+:(9) println(res3.toBuffer)

// :+ 添加元素到尾部

val res4: List[Int] = list.:+(9) println(res4.toBuffer)

// ::: 拼接集合。

val ints: List[Int] = list ::: List(5, 5, 5, 5) println(ints.toBuffer)

// :: 拼接 元素。

val res6: List[Any] = list :: List(5, 6, 7, 8) println(res6.toBuffer)

// :: 元素拼接集合

val res7: List[Int] = 5 :: list println(res7.toBuffer)

val res8: List[Int] = list.::(5) println(res8.toBuffer)

// list 集合中的方法。

// site 集合长度

println("list 集合个数：" + list.size)

// count 统计个数

println("偶数个数："+list.count(x => x % 2 == 0))

// filter 过滤的方法。 元组

val list1:List[(String,Int)]= List(("hello",12),("world",6),("tom",4))

// 过滤 String 包含 l 的元素。

val tuples: List[(String, Int)] = list1.filter(x=>x.\_1.contains("l")) tuples.foreach(println)

}

}

# 集合中的方法。

package com.dahua3

import scala.collection.mutable.ListBuffer object ListDemo01 {

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

val list = List.apply(3, 67, 44, 2, 5, 90, 100)

// filter 过滤的方法。 元组

val list1: List[(String, Int)] = List(("hello", 12), ("world", 3), ("tom", 22), ("hello", 55))

// 过滤 String 包含 l 的元素。

// val tuples: List[(String, Int)] = list1.filter(x=>x.\_1.contains("o")) val tuples: List[(String, Int)] = list1.filter(x => x.\_2 < 10) tuples.foreach(println)

// 排序：sorted

val res0: List[Int] = list.sorted res0.foreach(x => print(x + " "))

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

// 排序：sortBy

val res1: List[(String, Int)] = list1.sortBy(x => -x.\_2) res1.foreach(println)

// fold： 折叠

val list2 = List(1, 2, 3, 4)

// （（（（5+1）+2）+3）+4） println(list2.fold(5)((x, y) => x + y))

// (1-1)-2)-3)-4

println(list2.foldLeft(1)((x, y) => x - y))

// (1+4)+3)+2)+1

println(list2.foldRight(1)((x, y) => x + y))

// 1-(2-(3-(4-5)

println(list2.foldRight(5)((x, y) => x - y))

// 分组统计： 方法中定义几个数值为一组。

val res3: List[List[Int]] = list.grouped(3).toList

res3.foreach(println)

//(("hello",12),("world",3),("tom",22),("hello",55))

val map: Map[String, List[(String, Int)]] = list1.groupBy(x => x.\_1) for (ele <- map) {

println("key：" + ele.\_1 + "\tvalue:" + ele.\_2)

}

// 集合反转 reverse

val res4: List[Int] = list.sorted.reverse println(res4)

// reduce 统计。

// "hello",12),("world",3),("tom",22),("hello",55 println(list.reduce((x, y) => x + y))

// val tuples1:List[(String,Int)]= List(("hello",12),("world",3),("tom",22),("hello",55))

// println(tuples1.reduce((x, y) => x.\_2 + y.\_2))

// 全集，交集，差集var list3 = List(1,2,3,4) var list4 = List(3,4,5,6)

val res5: List[Int] = list3.union(list4)// 全集

println(res5) println(list3.intersect(list4))// 交集println(list4.diff(list3))// 差集

// 拉链 zip

var list5 = List(44,3,67,90)

var list6 = List(12,6,89,71,88,71)

val res6: List[(Int, Int)] = list5.zip(list6) println(res6)

// 局部聚合，全局聚合 spark。 想想一下：combiner reduce

// 12,6,89,71,88,71)

println(list.aggregate(1)((x, y) => x + y, (x, y) => x + y))

// 集合中，长度。

println("集合长度："+list.length)

// sortWith 根据什么排序。

val res7: List[Int] = list.sortWith((x,y)=>x>y) println(res7)

// 可变集合。

val res8: ListBuffer[Int] = ListBuffer[Int]()// 和 java 中的 ArrayList 相似。

res8+=12 res8+=58 res8+=15 res8.append(30) res8.insert(2,1,2,3)

println(res8.toBuffer)

}

}

map 集合

package com.dahua3

/\*\*

* Map key value 映射集合。 不可变。

\*

* import scala.collection.mutable.Map 可变集合。

\*/

object MapDemo {

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

val map: Map[String, Int] = Map[String, Int]("a" -> 1, "blue" -> 4, "hello" -> 5, "sql" -> 3)

// 获取集合中的元素。getorElse 查询不到返回默认值

val res0: Int = map.getOrElse("aa", 0) println(res0)

// 获取集合中的元素

val value: Int = map.get("a").get println(value)

// 遍历 map

val keys: Iterable[String] = map.keys for (elem <- keys) {

print("\tkeys:" + elem + "\tvalue:" + map.getOrElse(elem, 0))

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

val values: Iterable[Int] = map.values values.foreach(x => print(" " + x))

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

// 使用元组形式遍历

for (e <- map) {

println("key:" + e.\_1 + "\t" + e.\_2)

}

// 向 map 中插入值。

val newMap: Map[String, Int] = map.++(Map("c" -> 3)) newMap.foreach(println) println("======================================")

// 向 map 拼接元素

val newMap1: Map[String, Int] = map.+("d" -> 5) newMap1.foreach(println)

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

// map 中的 size，count

println("map 集合长度：" + map.size)

// 进行任意统计

// Map[String,Int]("a" -> 1,"blue" -> 4,"hello" -> 5,"sql" -> 3) println("包含 L 的 key 有：" + map.count(\_.\_1.contains("l")) + "个") println("偶数 value 有：" + map.count(\_.\_2 % 2 == 0) + "个")

// filter

// exit

}

}

1. **Set** 集合

package com.dahua3

import scala.collection.mutable

/\*\*

\* Set 集合特征：无序的，不可重复的。不可变集合。

\*

\* import scala.collection.mutable.\_

\*/

object SetDemo extends App {

var set:Set[Int] = Set(4, 7, 9, 1, 12, 12, 56, 34, 78)

for (ele <- set) { print(ele + " ")

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

// 可以用可变集合。

val hashSet: mutable.HashSet[Int] = mutable.HashSet[Int]() hashSet += (12)

hashSet += (3)

hashSet += (5) println(hashSet)

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

// 所有的集合都可相互转换。

private val list: List[Int] = set.toList

Set("a","b","c")

.zipWithIndex //(a,0), (b,1), (c,2)

.map{case(v,i) => (v, i+1)} //increment each of those indexes

.toMap

private val set1 = Set("a","b","c") private val set2 = Set(1,2,3)

private val map: Map[String, Int] = set1.zip(set2).toMap println(map)

}

元组

package com.dahua3

/\*\*

\* 元组：

* Tuple。
* 在 scala 中，包含 8 种基本数据类型，引用数据类型，元组类型。
* 只要看见括号，就是元组。
* 访问元组中的数据 \_1 \_2 \_3

\*/

object TupleDemo {

def add(x: Int, y: Int): (String, Int, Int, Int, Int) = { ("结果：", x + y, x - y, x \* y, x / y)

}

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

val tup0: (String, Int, Boolean) = Tuple3("hello", 1, true) println(tup0.\_1)

println(tup0.\_2) println(tup0.\_3)

val tup1: (String, String, Int) = ("java", "scala", 100) println(tup1.\_1 + " " + tup1.\_2 + " " + tup1.\_3)

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

val tup2: (String, Int, Int, Int, Int) = add(10, 3) println(tup2.\_1 + tup2.\_2)

println(tup2.\_1 + tup2.\_3)

val arr:Array[Int] = Array(1, 3, 4, 5, 6) var tup3 : Array[(Int,Int)] = arr.map(x=>{

(x,x\*10)

})

// 返回值的个数

for(ele <- 0 until(tup3.length)){ println(tup3(ele).\_1+" "+tup3(ele).\_2)

}

// 对偶元组

val tup4: Array[(String, Int)] = Array(("hello",1),("scala",4)) val swap: (Int, String) = tup4(1).swap

println(swap.\_1+" "+swap.\_2)

}

}

**scala 简介**

1. **scala** 是什么？

java 是面向对象的编程语言。

scala 斯卡拉。是函数式编程语言。底层也是运行在虚拟机中。

hive 自定义函数。了解：

什么是函数？拥有返回值，函数名，参数列表，执行体的语句结构叫做函数，可以直接 调用或使用。

什么是方法？方法定义在对象中，只有创建对象，才能调用方法。

method function

之前我们学过语言：

java 语言。JDK8 的新特性。lambda 表达式。System.out.pritnln("%s,%n",a,b); javascript 脚本语言。弱类型语言。var a = 10; var b = "10".

sql 结构化查询语言。

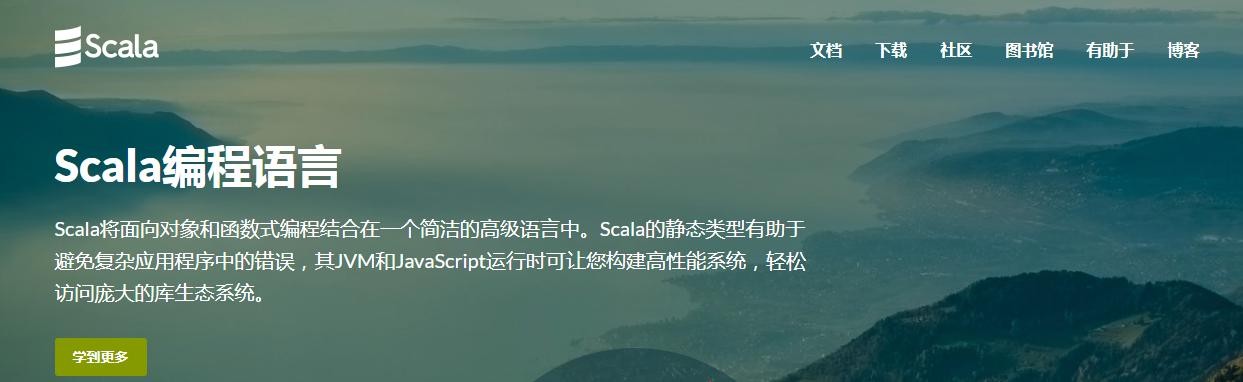
scala 。 源自于 JDK8 中的特性。要求在 JVM 中运行。 函数式编程。

scala 是一种多范式的编程语言。其设计的初衷是要集成面向对象式编程和函数式编程的各种特性，Scala 运行于 java 平台（java 虚拟机）。并<兼容于现有的 java 程序>。

Scala 原来的官网上有一句话： 对象遇见函数。

Scala，在设计之初就没有计划要替代 java。因为他也不能。

hadoop 生态圈中。Scala 与 spark 结合。对数据处理算法能够起到事半功倍的效果。比如：我们曾经写过 wordcount .java 至少写了 50 行。但是 scala 1 行。



柯里化(Currying)指的是将原来接受两个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数为参数的函数。

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。

个人理解：何为柯里化，加一句没有闭包就没有柯里化

1、必须是函数

2、接收多个参数

3、将接收的多个参数转化为接收一个参数

1. 为什么要学 **Scala**

1:优雅：这是框架设计师第一个要考虑的问题。框架的用户是应用开发程序员，API 是否优雅直接影响用户的体验。

(简洁，不择手段的简洁)

2：速度快（不表现在运行速度上。）：Scala 语言表达能力强。一行代码抵得上 java 多行代码。开发速度快。Scala 是静态编译的。所以和 JRuby,Groovy 速度比起来会快很多。

价值较高的语言：pathon Go scala java shell

3：能融合到 Hadoop 生态圈。Hadoop 现在是大数据实时标准。Spark 并不是要取代 Hadoop,

而是要完善 Hadoop 生态。JVM 语言大部分要应用到 java，但 java 的 API 太过繁重。

1. **Scala** 和 **java** 的对比：



1. 开发环境准备：

JDK1.8+

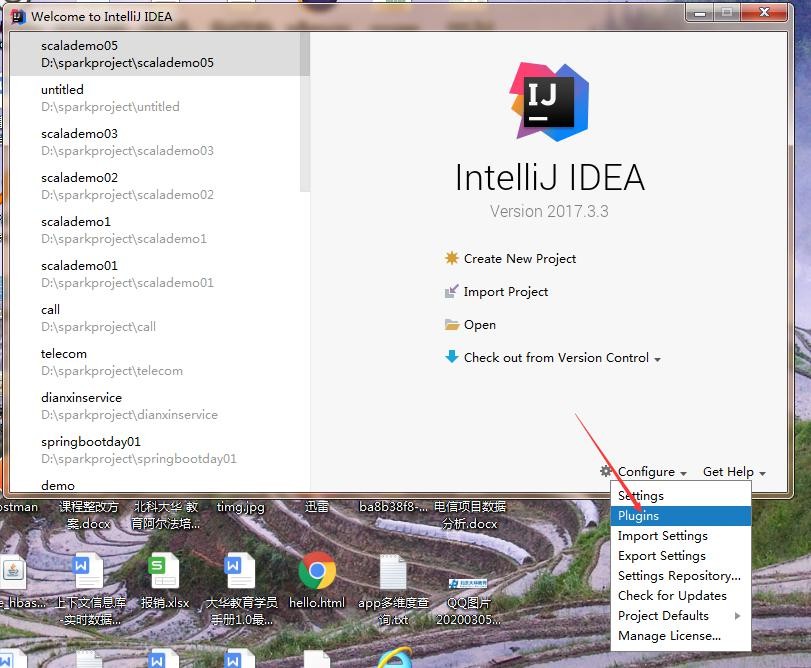
IDEA

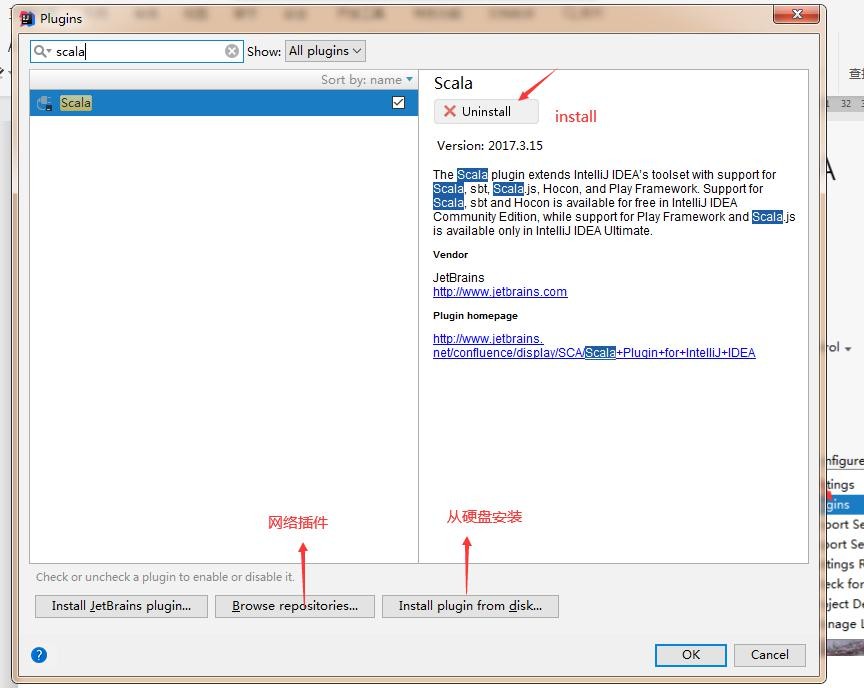
ScalaSDK 安装。

scala 官网地址：[https://www.scala-lang.org](https://www.scala-lang.org/)

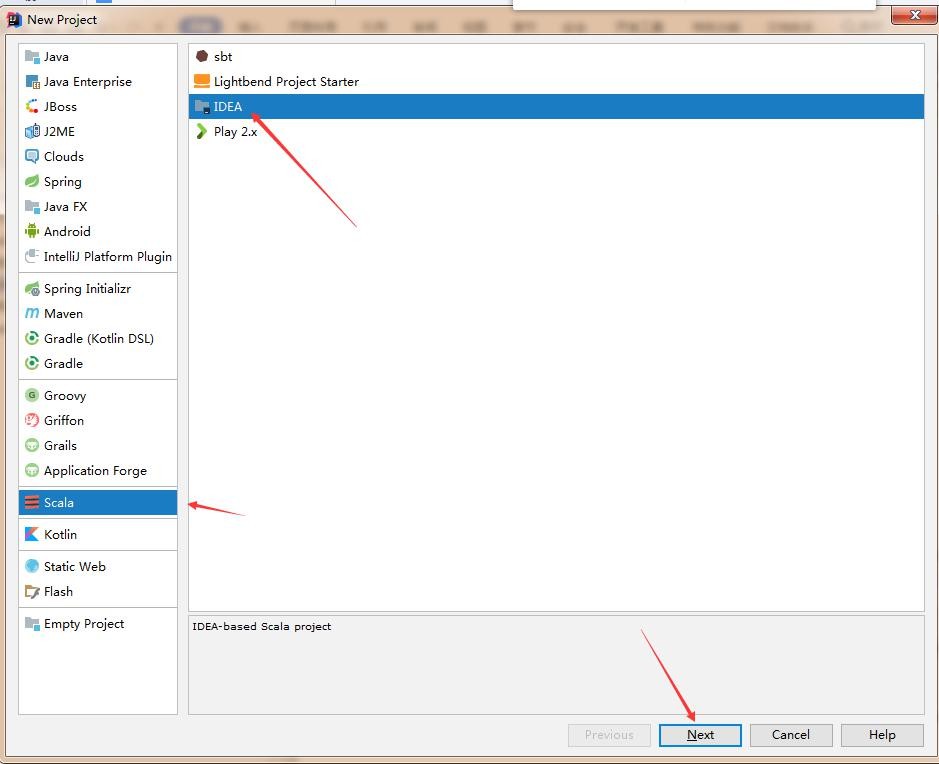
scala 插件地址：<https://plugins.jetbrains.com/plugin/1347-scala>

1. **IDEA** 配置 **scala** 插件





创建 scala 项目



1. 编写第一个 **HelloWorld**

object HelloWorld {

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

println("hello rrr ")

}

}

1. **Scala** 中的数据类型：

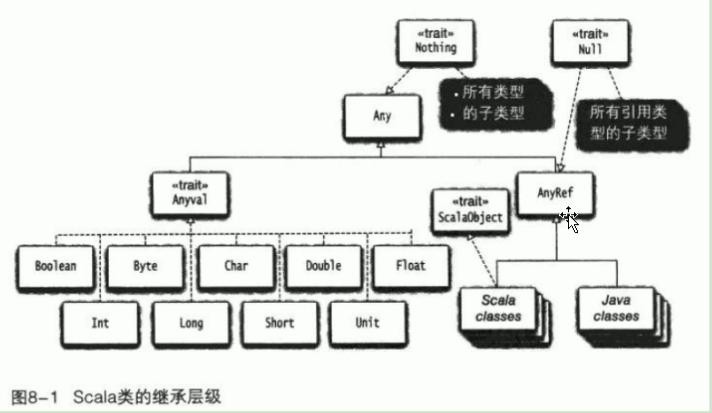
Scala 和 java 一样，有 7 中数值类型：

Byte Short Int Long Float Double Char

Boolean Unit 类型。字符串：String .

注意 Unit 表示无值，和其他语言中的 void 等同。用作不返回任何结果的方法的返回值类型。

Unit 只有一个实例 （）



Scala 声明变量

class Student(var name:String,var age:Integer)/\*\* Scala 定义变量

var | val

var 表示定义变量，

val 表示定义变量，但其值不能发生改变。

\*/

var name = "张三"

println(name) val age = 18

// age = 19 错误，val 定义值不能发生改变。name = "李四"

println("姓名："+name,age);

}

/\*\*

\* Unit 类型。

\*

\*/

var s:Unit = println("hello") println(s)

/\*\*

* 输出类型
* println()
* printf()
* println(s)

\*/

var stuName = "Jack" var stuAge = 19;

var stuPrice = 333.88d;

// println()

println("姓名："+stuName,"年龄"+stuAge,"薪资"+stuPrice) println(f"姓名：$stuName","年龄"+stuAge,"薪资"+stuPrice) printf("姓名 %s,年龄%s,薪资 %s",stuName,stuAge,stuPrice) val student = new Student("李四",22)

println(s"姓名：${student.name}");

for 循环

def main(args: Array[String]): Unit = { for(ele <- 1 to 10)

print(ele+" ")

println("\n=======================================")

for(ele <- 1 until(10))

print(ele+" ")

/\*\*

\* if else

\*/

println(if(1 == 1) 10)

println(if(1==2)10 else 20)

var flag = println(if(1==2) 10 else if (1==2) 20)

println(flag)

var r1:Any = if(1==2) 10 else "sss"

println(r1)

println("\n=======================================")

var arr =Array(15,33,2,4,90) for(ele <- arr){

print(ele+" ")

}

println("\n=======================================")

for(ele <- 0 until(arr.length) ){

print(arr(ele)+" ")

}

println("\n=======================================")

// 添加守卫。

for(ele <- 0 until(arr.length) if arr(ele) %2 ==0 ){

print(arr(ele)+" ")

}

println("\n=======================================")

// 99 乘法表

for(i <- 1 to 9;j <- 1 to i){ print(j+"\*"+i+"="+(i\*j)+" ") if(i==j){

println("")

}

}

|  |  |  |
| --- | --- | --- |
|  | println(1.+(1)) |  |
| var arr2=for(e <- arr if e%2==0) yield e  for (i <- arr2){  print(i+" ")/\*output:2,4,6\*/  }  函数表达式 | | |

1. **Scala** 面向对象

package com.dahua4

/\*\*

\*

* Scala 中对象分为两种：
* 1： Object 是静态类。 类中定义的属性和方法都是静态的。

\*

* 2： class 类：具有属性，方法，构造方法。使用该类需要 new

\*

\*/

object Student {

// 属性

var name: String = "张三" var age: Int = 19

// 方法

def show(): Unit = {

println(s"姓名：${name}" + s" 年龄：${age}")

}

// 是否具有构造方法。

// apply()

def apply(str: String,num:Int): Unit = { name = str

age = num

}

// 允许多个 apply 方法存在。

def apply(str: String,num:Int,a:Int): Unit = { name = str

age = num

}

}

package com.dahua4

import com.dahua5.Teacher

/\*\*

\* Scala 的普通类。

\*/

class Teacher private(var n: String, var y: Int) {// n y 已经是类中的属性了。

// scala 的有参构造方法，参数值写在类名后边

// 属性

var name: String = \_ var age: Int = \_

// 方法

def show(param: String) = { println(param)

}

}

object Teacher{

def apply(n:String,y:Int):Teacher ={// 伴生方法。

new Teacher(n,y)

}

}

package com.dahua4

/\*\*

\* 测试类

\*/

object Test {

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

println(Student.name)// Object 定义的类是：静态类。属性是静态属性。

Student.show()// 调用静态方法。

// 改变对象中属性的值： Student.name = "李四" println(Student.name)

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

// 调用对象的 apply 方法。

Student.apply("糖果",10)// 语法糖

println(Student.name) println(Student.age)

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

Student("糖果 1",100)// 语法糖println(Student.name) println(Student.age)

// 可以参照以下类。

List(1,2,3,4)

List.apply(1,2,3,4)

}

}

# 抽象类

/\*\*

\* 抽象类

\*/

abstract class AbstractClass {

// 抽象方法。

def eat(str:String)

// 普通方法。

def play(str:String)={

println("喜欢玩的游戏："+str)

}

}

1. 继承抽象类

package com.dahua5

class AbsClassImpl extends AbstractClass { override

def eat(str: String): Unit = { println("喜欢吃的水果："+str)

}

override def play(str: String): Unit = { println("喜欢看的电影："+str);

}

}

package com.dahua5 object AbsClassTest {

def main(args: Array[String]): Unit = { val impl = new AbsClassImpl() impl.eat("菠萝")

// 由子类进行重写的方法。

impl.play("红鞭")

}

}

1. 特性 **Trant**

/\*\*

\* Trait 是 Scala 中的接口。相当于 java 中的 Interface

\*/

trait TraitOne {

// 在接口中定义属性

var PI = 3.14

// 接口中定义方法

def hello()

def say(str:String)

// 接口中可否有普通方法

def play(str:String)={

println(s"喜欢玩的游戏：${str}")

}

}

trait TraitTwo { def show()

}

package com.dahua5

/\*\*

\* 实现特性（接口）

\*/

class TraitOneImpl extends AbstractClass with TraitOne with TraitTwo{

// 强制实现的。特性中的方法。

override def hello(): Unit = {

PI = 45

println(PI)

}

// 强制实现的。特性中的方法。

override def say(str: String): Unit = println("say scala 语言")

override def play(str: String): Unit = super.play(str)

// 特性二中的方法。

override def show(): Unit = println("实现特性二")

// 抽象类中的方法.

override def eat(str: String): Unit = ???

}

# 混入特质

package com.dahua5

/\*\*

\* 混入特质

\*/

class Stu(){} object IntoTrait {

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

// 混入特质：像 java 中的匿名内部类。

var stu = new Stu() with TraitOne with TraitTwo { override def hello(): Unit = println("hello 混入特质")

override def say(str: String): Unit = ??? override def show(): Unit = ???

}

stu.hello()

}

}

1. 隐士转换函数

package com.dahua6

/\*\*

\* 隐士转换函数

\*/

object Scala\_01 {

// 隐士转换函数

// Double==>Int

implicit def dou2Int(d:Double):Int={ d.toInt

}

// 隐士函数不能有相同参数和相同的返回值类型。会让 scala 无法精确进行转换。

// implicit def Idou2Int(d:Double):Int={

// d.toInt

// }// 错误

implicit def str2Int(s:String):Int={ Integer.parseInt(s)

}

def main(args: Array[String]): Unit = { var num:Int = 3.15

println(num)

var num1:Int = "1234" println(num1)

}

}

# 隐士类

package com.dahua6

/\*\*

* 隐士类
* A a = new A()

\* a.a()

\*

* B b = new B()

\* b.b()

\*

* 我要通过怎样的方式：

\* a.a()

\* a.b()

\*

* 隐士类增强。

\*

\*/

object Scala\_02 {

// 隐士转换

implicit def a2B(a:A):B={ new B

}

def main(args: Array[String]): Unit = { val a = new A()

val b = new B()

// 我想要：a 中也能调用 B 的方法。

a.b("B")

}

}

class A{

def a(name:String)={

println(s"我是 A 方法：${name}")

}

}

class B{

def b(name:String)={

println(s"我是 B 方法：${name}")

}

}

1. 隐士转换增强

package com.dahua6

/\*\*

\* 隐士转换，增强类型。

\*/

object Scala\_03 {

implicit def mysql2DB(ms:MySql):DB=new DB implicit def mysql2Nosql(ms:MySql):NoSql=new NoSql

def main(args: Array[String]): Unit = { val mysql = new MySql mysql.insert()

mysql.select()

// mysql 想要创建数据。mysql 没有方法，而 DB 有。

mysql.create() mysql.createTable mysql.keyVal

}

}

class MySql{

def select(): Unit ={

println("我能够进行查询……")

}

def insert(): Unit ={

println("我能够进行插入……")

}

}

class DB{

def create(): Unit ={

println("我能创建数据库")

}

def createTable: Unit ={ println("我能建表")

}

}

class NoSql{

def keyVal: Unit ={

println("转换成 key Value 形式")

}

package com.dahua6

/\*\*

\* 隐士类。

\*/

object Scala\_05 {

implicit class DB1(val m:Mysql1){// 通过构造函数，传入对象。其对象，就拥有了可调用方法的能力。

def display = println(m.n+"hello")

}

def main(args: Array[String]): Unit = { val mysql = new Mysql1("mysql") mysql.display

}

}

class Mysql1(var n:String){

}

隐士参数

package com.dahua6

/\*\*

* 隐士函数

\*

* 隐士类

\*

* 隐士参数

\*/

object Scala\_06 {

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

// 隐士参数

implicit val name:String = "学生"

def hello(str:String)(implicit content:String)={ println(s"你好：${str}"+" 职业： "+content)

}

}

package com.dahua6

/\*\*

* 购票系统：
* 约束购票的人群。什么人可以买票，什么人，不可以买票。

\*/

class Worker(val name:String){}

class Student2(val name:String){}// 学生

class Older(val name:String){}// 老人

class SpecialPerson(val name:String){}// 特殊人群

class TichkerSystem{

def buy(p:SpecialPerson): Unit ={ if(p.name.length>1){

println("您好："+p.name+"\t 恭喜您购票成功")

}else{

println("购票失败")

}

}

}

object Scala\_07 {

implicit def object2SpecialPerson(obj:Object):SpecialPerson ={ if(obj.getClass == classOf[Student2]){

// 判断是学生对象。

val student: Student2 = obj.asInstanceOf[Student2] new SpecialPerson(student.name)

}else if(obj.getClass == classOf[Older]){

val older: Older = obj.asInstanceOf[Older] new SpecialPerson(older.name)

}else{

new SpecialPerson("")

}

}

def main(args: Array[String]): Unit = { val zs = new Student2("张三")

val old = new Older("老人")

val work = new Worker("工作人员")

val system = new TichkerSystem system.buy(zs)

system.buy(old) system.buy(work)

}

}

# 模式匹配

package com.dahua6

/\*\*

* Scala 的模式匹配：
* 就是像 java 中 Switch case

\*/

object Scala\_08 {

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

// 匹配字符串内容

def stringMatch(str: String) = str match { case "hello" => println("hello scala") case "dog" => println("hello dog") case "1" => println("hello 2")

case \_ => println("匹配无效")

}

stringMatch("hello") stringMatch("dog") stringMatch("2")

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

// 匹配：类型

def typeMatch(ty: Any) = ty match { case x: Int => println(s"Int${x}")

case y: Double => println(s"double${y}") case \_ => println("无效匹配")

}

typeMatch(123) typeMatch(123.15)

// 匹配数组

println("----------匹配数组 ")

def arrayMatch(arr: Any) = arr match { case Array(0) => println("空数组")

case Array(0, \_) => println("以 0 开头,拥有两个元素")

case Array(0, \_, 3) => println("以 0 开头,以 3 结尾,中间有无限其他元素") case Array(0, \_\*) => println("以 0 开头,最后有无限其他元素")

}

arrayMatch(Array(0)) arrayMatch(Array(0, 2))

arrayMatch(Array(0, 9, 3))

arrayMatch(Array(0, 9, 4, 5, 6, 7, 89, 100)) println("----------匹配 List 集合 ")

def listMatch(list: Any) = list match {

case 0 :: Nil => println("只有一个 0 的集合") case 0 :: 7 :: Nil => println("只有 0 和 7 的集合")

case x :: y :: z :: Nil => println("任意三个元素的集合") case m :: n if n.length > 0 => println(" ")

case \_ => println("任意集合")

}

listMatch(List(0)) listMatch(List(0, 7)) listMatch(List(0, "true", true)) listMatch(List("hello", "true")) listMatch(new Stu)

println("-------------匹配元组 ")

def tupleMatch(tuple: Any) = tuple match {

case (0, \_) => println("以 0 开头的一个元组,第二个值任意") case (x, y, z) => println("拥有任意 3 个元素的元组")

case (\_, "tom") => println("第二元素必须是 tom") case \_ => println("任意值")

}

tupleMatch((0, "jack"))

tupleMatch((1, "tom")) tupleMatch((1, "lucy",true))

println("------------任意匹配 ")

def nothingMatch(nothing:Any)=nothing match{ case "hello" => println("你是字符串")

case x:Int => println("你是数字:"+x)

case Array(\_\*) =>

val ints: Array[Int] = nothing.asInstanceOf[Array[Int]] ints.foreach(println)

}

nothingMatch("hello") nothingMatch(Array(0,1,2,32))

println(" 匹配对象 ") def objMatch(obj:Any)=obj match{

case Stu()=> println("学生类型")

case Teacher() => println("老师类型") case Major() => println("专业类型")

}

objMatch(new Stu()) objMatch(new Teacher()) objMatch(new Major())

}

}

case class Stu() {} case class Teacher(){}

case class Major(){}