第11章：Scala 基础

Objective(本课目标)

- [ ] Scala 开发环境设置
- [x] 变量与方法 (Variables & Functions)
- [x] 数据类型与集合(Data Types & Collections)
- [x] Scala函数编写

Scala和Java编程的区别

```
// Java实现数组求和求最大值
int[] nums = {123,141,12,123,523,724,271,795};
int sum=0;
for(int i=0;i<nums.length;i++) {
    sum+=nums[i]
}

//maxValue
int mv = nums[0];
for(int i=1;i<nums.length;i++) {
    mv = max(nums[i],mv)
}

// scala实现数组求和,与最大值
val nums = List[Int](1,2,3,4,5,6)
nums.reduce(_+_)
nums.max
```

Scala, Python vs. R

	Scala(开发)	Python(开发，分析*)	R(数据科学)
Ease of Learning (学习难度)	Moderate	Easy	Easiest
Productivity	Good (for complex)	Good (for simple)	Good (for research)
Resources/Libraries	Good	Best	Better
Documentation/Community	Good	Best	Best
Maturity(稳定)	Good	Best	Better
Performance(性能)	Fast	Slower	Slower
Spark Integration(集成)	Best	Good	Good
Feature Availability(功能可用性)	First	After	After

Set up Scala IDE

- Set-up Scala IDE in IntelliJ
  - Settings -> Settings -> Scala Plugin
- Set-up Scala IDE in Eclipse
- Set-up Interactive Scala IDE
  - http://scala-lang.org/download/

The "Hello, World!" Program

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    printf("Hello world!")
  }
}

// scalac HelloWorld.scala  编译
// scala HelloWorld         运行
// HelloWorld.main(null)    执行main函数
```

Scala Overview(概述)

- Scala is object-oriented – Every value is an object(scala是面向对象的，所有的value都是对象)
  - Class & Trait (interface)

    - Multiple Inheritance - mixin-based composition(多继承必须有一个基类)
- Scala is functional – Every function is a value (所有的function都是value)
    - Higher-order function (高阶函数)
    - Currying (柯里化)
    - Case Classes – Pattern Matching (样例类-模式匹配)
    - Extractor Objects (对象提取器)
- Scala is extensible (scala是可扩展的)

Variable & Function(变量和函数)

```
// 变量和值的定义
var i = 100
i += 1

// value不能改变
val x = 200
x = 500; //error

//显示的指定类型
var y:Double = 234.12

//方法的定义
def square(x: Int): Int = {
    println(x)
    x * x //返回值return关键字可以省略，最后一行代码代表返回值。
}

def printByValue(x:Any):Unit = {
    println(x);println(x)
  }

def printByName(x: => Any): Unit = {
    println(x); println(x)
  }

printByValue(square(10))
printByName(square(10))
```

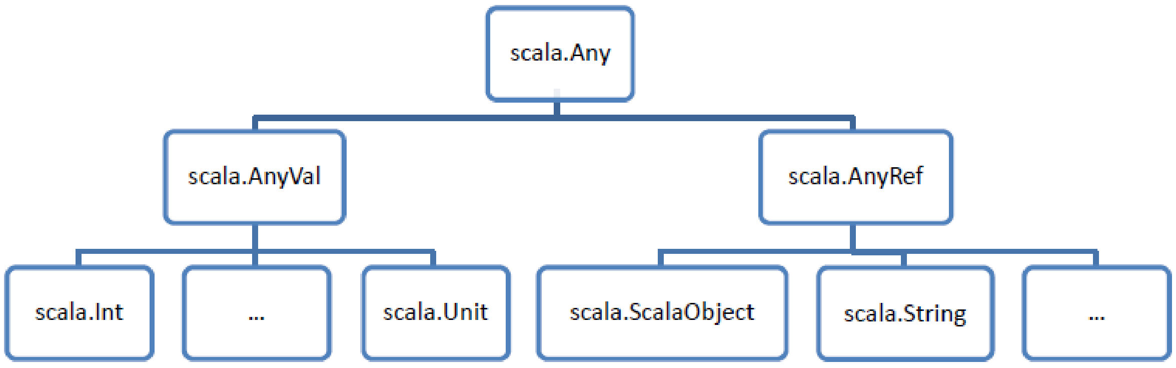
Loop in Scala(循环)

```

// 操作循环数据集，掌握map函数的使用.
// 案例1
var num:Int = 10
for (i: Int <- 1 to num) {
    println(i)
```

```scala
}

// 案例2
val ss = List[String]("a","b","c","d");
for (s: String <- ss) {
    println(s)
}
// 案例3
for(i:Int <- 1 until 100; if i %2 ==0) {
    val y = 2 * i
    println(y)
}

//map函数的基本使用
// 案例4
(1 to 100 by 5).map { x =>
      val y = x * 2
      println(y)
      y
}

//案例5
var result = (1 to 100 by  5).map {x:Int => x*5}
def f2(x: Int) = 2 * x
(1 to 100) map f2

//案例6(***)
var s= Array((19,1),(20,1)).map {
    case(k:Int,v:Int) => (k,k+v)
}
s.foreach{ x=> println(x)}

// until生成的数据集不包含最后一个,掌握filter, 和 _ 的使用
var v1 = (1 to 100).filter {_%2 ==0 } map {_*2}
var v2 = (1 until 30 by 5) filter {_%2 == 0} map {_*2}

// while循环
var num = 1
while (num < 10) {
    println(num)
    num = num+1
}

// do while循环
var num = 51
    do {
        println(num)
        num = num + 1
    } while (num < 7)

//There is no break or continue
def test1() : Unit = {
    while (true) {
        val rand = Math.random()
        if (rand > 0) {
            println(rand)
        }
    }
    println("Done") // not reachable
}

// break使用
import scala.util.control.Breaks._
    def test2(): Unit = {
      println("start")
      val l = List[Int](1,2,3,4,5)
      breakable {
        for(x <- l) {
          println(x)
          if (x > 3) break
        }
      }
    println("Done")
  }
```

Unified Types(统一类型)

- scala.AnyRef -> java.lang.Object (相当于Java的Object)
- User defined classes implicitly extend the trait scala.ScalaObject (自定义的class默认继承ScalaObject类)



- final trait Null extends AnyRef (Null从AnyRef扩展而来)
    - Null is a subtype of all reference types. (Null cannot be assigned to any value types)
    - (Null是所有引用类型的子类型，Null不能被赋值给任意的值类型)
- final trait Nothing extends Any (Nothing从AnyRef扩展而来)
    - There are no instances of Nothing (Nothing没有实例)
- final class Unit extends AnyVal (Unit从AnyVal扩展而来)
    - Any methods with return of Unit has no return (methods with void return in java) (任何返回Unit的方法都没有返回(java中返回void的方法)

"Primitive" Types(基本类型)

Data Type	Description
Byte	8 bit signed value. Rang from -128 to 127
Short	16 bit signed value. Range -32768 to 32767
Int	32 bit signed value. Range -2147483648 to 2147483647
Long	64 bit signed value. -9223372036854775808 to 9223372036854775807
Float	32 bit IEEE 754 single-precision float
Double	64 bit IEEE 754 double-precision float
Char	16 bit unsigned Unicode character. Range from U+0000 to U+FFFF
Boolean	Either the literal true or the literal false

Tuples(数据结构-元组)

```
// the following 3 definitions are the same(下面三种定义的方式是一样的)
var t1 = ("zs","wuhan",60)
var t2 = new Tuple3("zs","wuhan",60)
var t3 = new Tuple3[String,String,Int]("zs","wuhan",60)

// tuple的定义长度不能超过22.
var x = new Tuple22( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21,22) //OK
x = new Tuple23( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22,23) //ERROR

//遍历Tuple
t1.productIterator.foreach {
      s => println(s)
}

t1.productIterator.map {
    x => x + "~"
}.foreach {
    s => println(s)
}

//Tuple取值
println(d._2)

//数据提取
val(name, address, age) = d
println(name)
```

```
//动态定义
def mike = "Mike" -> 5
mike.getClass
val(name, age) = mike
println(age)
```

Collections

- scala.collection (collection接口)
  - Defines the same interface as the immutable and mutable collections (定义可变与不可变的collection)
- scala.collection.immutable
  - Defines collections that are immutable (定义不可变的集合)

```
//默认情况一般都是使用 immutable collection
val is = collection.immutable.Stack(1,2,3) //immutable
is(1) = 100 //error
val ms = scala.collection.mutable.Stack(1,2,3)
ms(1) = 100 //OK
```

scala.collection features

Name	Features	
Buffer	mutable	Sequences of elements incrementally by appending, prepending or inserting new elements.(通过追加，添加或插入新元素来递增地排列元素序列)
Array	mutable	Fixed-size sequential collections of different types of values (不同类型值的固定大小的连续集合)
List	immutable	Ordered collections of elements of type A (指定类型元素的有序集合)
Map	mutable	Key-value pair collections (k-v对collection)
Set	mutable/immutable	No duplicate, no order collection.(不能重复，没有顺序的collection)
Vector	immutable	Tree structure with constant time fast random access and update, good for very large sequences(树型结构，快速随机存取和更新，适用于非常大的序列)
Stack	mutable/immutable	First in, last out (先进后出)
Queue	mutable/immutable	First in, first out (先进先出)
BitSet	mutable/immutable	BitSet(3, 2, 0) -> 1101 [64 bit Longs. The first Long is for 0 ~ 63]
ListMap	immutable	
HashSet	mutable	
HashMap	mutable	

Collection – Array & List

```
// 定义Array,与基本操作
var a1 = Array[Int](1, 2, 3, 4)
var a2 = Array[Int](100, 200, 300, 400)
var a = Array.concat(a1, a2)
a = a1 ++ a2
a(3) = 333

// 连接不同的数据结构
var b = Array(333, "333", '3',false)
var c = List.concat(a, b)

//数组过滤和反转
val x = a.filter(_ %2 != 0)
val y = a.reverse

var m = a.groupBy(t => t%2 == 0) //数据分组，返回Map
var n = a.slice(2, 4) //数组截取

//排序方式
a.sorted
a.sorted(Ordering.Int.reverse)
a.sortWith(_>_)
a.sortBy(x => x) //ascending
a.sortBy(x => x*(-1)) //descending

//排序案例，按照从单词长度排序 the same interface as the immutable and mutable
collections
var s  = "the same interface as the immutable and mutable collections"
s.split(" ").sortWith((x,y) => x.length < y.length)

// 定义List,与基本操作
var l = List[Int](2, 3, 4, 6, 8, 9, 11, 20)
var x = l grouped 3 //每组三个元素，返回迭代器
var y = l sliding 2 //数据滑动

// append
var c = List[Char]('a','b','c')
var x = 'x' +: c // (x, a, b, c)
var y = c :+ 'x' // (a, b, c, x)

import scala.collection.mutable._
var lb = ListBuffer[Int](1, 2, 3, 4)
lb += 100 //1, 2, 3, 4, 100
lb += (21, 33) //1, 2, 3, 4, 100, 21, 33
88 +=: lb //88, 1, 2, 3, 4, 100, 21, 33
List(77, 66) ++=: lb //77, 66, 88, 1, 2, 3, 4, 100, 21, 33
```

- **Collection – Set & Map**

```
// Set基本操作
var s = Set("aa","bb")
var t = Set("aa","cc","dd","ee","ff")

//intersection (交集)
t & s
t intersect s

//the union (并集)
t | s
t union s

//the difference(差集)
t &~ s
t diff s

//Set append
```

```scala
var s = Set("aa","bb")
s += "cc" //mn, ab, yz
s += "cc" //??
s -= "bb" //mn, ab
var t = Set("aa","cc","dd","ee","ff")
println(t -- s)
println(t ++ s)

//有序Set
var ss = SortedSet("85","25","93","55")
ss += "33"

//Map 基本操作
var m = Map[String, Int]("a"->1, "b"->2, "c" ->3, "d" ->4, "e"->5, "f" ->6)
m += ("j"->0)
m += ("j"->0) //??
m += ("j"->11) //??
var n = m ++ Map[String, Int]("a" -> 3,"h" -> 99)
n -= ("a","b")

// 变换
val m2 = m.map{
 case (k,v) => (k,v+1)
}
```

- **Enumeration(枚举)**

```scala
//定义枚举类
object WeekDay extends Enumeration {
    type WeekDay = Value
    val Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday = Value
}

//实例化
val monday:WeekDay = WeekDay.Monday
val sunday = WeekDay.withName("Sunday")
println(sunday.id) //输出index

WeekDay(1) //根据index输出对应的星期
WeekDay.maxId //最大的index

//方式二 定义枚举
object WeekDay extends Enumeration {
    type WeekDay = Value
    val Monday = Value("Monday")
    val Tuesday = Value("Tuesday")
    val Wednesday = Value("Wednesday")
    val Thursday = Value("Thursday")
    val Friday = Value("Friday")
    val Saturday = Value("Saturday")
    val Sunday = Value("Sunday")
}

// Pattern Matching(模式匹配)
def isWeekend(wd: WeekDay.Value): Boolean = {
    wd match {
      case WeekDay.Saturday => true
      case WeekDay.Sunday => true
      case _ => false
    }
}
//调用
isWeekend(WeekDay.Monday)
```

Null & null

```
// 定义参数为Null的function
def tryit(thing: Null): Unit = {
    println("That worked!");
}

// 调用function方式一
val someRef: String = null //error: type mismatch;

// 调用function方式二
tryit(null) //is ok
val nullRef: Null = null
tryit(nullRef) //is ok
```

- null only one instance of Null (Null只有一个实例就是null)

Nothing

- final trait Nothing extends Any(Nothing继承 Any)
  - Nothing is a subtype of everything (Nothing是所有类型的子类型)
  - There is no instance of Nothing (Nothing没有实例)
  - Any collection of Nothing must necessarily be empty (任何Nothing的collection都必须为空)

```
// 无实例
val emptyStringList: List[String] = List[Nothing]()
emptyStringList: List[String] = List()

// 无实例
scala> val emptyIntList: List[Int] = List[Nothing]()
emptyIntList: List[Int] = List()

// 错误例子
val emptyStringList: List[String] = List[Nothing]("aaa")
error: type mismatch;
found : java.lang.String("aaa")
required: Nothing
```

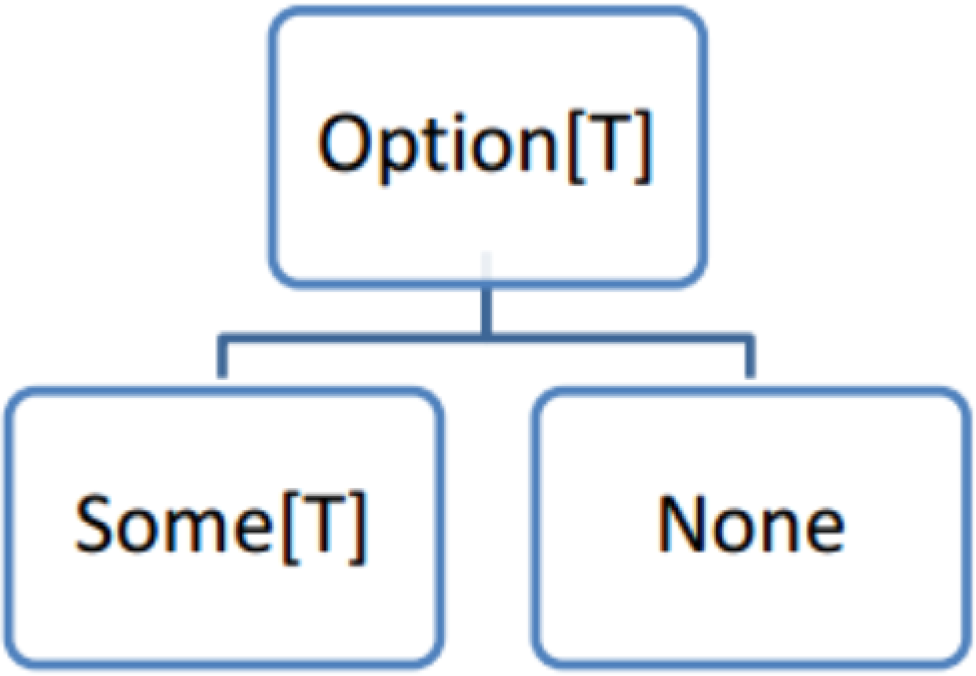
- Any methods with return of Nothing never return normally, such as one method always throws exception（任何返回Nothing的方法都不会正常返回，例如method抛出异常）

Nil & None

- case object Nil extends List[Nothing]
- object None extends Option[Nothing]
- Nil和None都是属于值

```
// Nil演示
import scala.collection.immutable.Nil
println(Nil.length) // 0

//option演示
val s1: Optoin[Int] = scala.util.Try("1234a".toInt).toOption
val s2: Optoin[Int] = scala.util.Try("1234".toInt).toOption
s2.get //获取值
s2.getOrElse("failure") //获取值

val s3: String = null //不建议
val s3: Option[String] = None //建议
s3 = Some("jack")
s match {
    case Some("jack") => println(ss)
    case _ => println("No value")
}

// 返回值为Option[String]
def getAStringMaybe(num: Int):Option[String] = {
    if(num >=0) Some("A positive number!")
    else None
}
```

```
//调用
println(getAStringMaybe(123))
println(getAStringMaybe(-1))

// 模式匹配
def printResult(num: Int) = {
    getAStringMaybe(num) match {
    case Some(str) => println(str)
    case None => println("No string!")
    }
}

//调用
printResult(123)
printResult(-123)
```



Left & Right

- Either is just like Option(和Option比较相似)
- Right -> Some (Right对应Some)
- Left -> None, except content can be included to describe the problem (Left对应None，除了内容可以包括描述问题)
- Either与Option的区别是返回两种不同的值，而Option是返回不同的类型。

```
//返回值为Either.
def divideBy(x:Int,y:Int):Either[String,Int] = {
  if (y == 0) Left("can't divide by 0")
  else Right(x / y)
}

//模式匹配
divideBy(100, 20) match {
  case Left(s) => println(s)
  case Right(v) => print("right = %d".format(v))
}
```

- **Success & Failure**

```
// 案例1 Try: 返回Success,Failure
import scala.util.Try
scala> Try("123")
res1: scala.util.Try[String] = Success(123)

import scala.util.Success
import scala.util.Failure
Try("123".toInt) match {
    case Success(v) => println("thi value is %d".format(v))
    case Failure(e) => println("Exception occurred - %s".format(e.getMessage()))
}

Try("123".toInt) match {
    case Success(v) => println("thi value is %d".format(v))
    case Failure(e) => throw e
}

//案例2
val t: Try[Int] = Success(100)
val e: Try[Int] = Failure(new Exception("error"))

def isSuccess(z:Try[Int]):Boolean = {
    z match {
      case Success(v) => true
      case _ => false
    }
  }

//案例3
val s:String = "100"

def isInt(s: String):Boolean = Try(s.toInt) match {
    case Success(_) => true
    case _ => false
  }
```

```
//案例4 scala代码和java代码区别
  def isDate(s: String):Boolean = Try {
    val fmt = new java.text.SimpleDateFormat("yyyy-MM-dd")
    fmt.parse(s)
  }match {
    case Success(dt) => true
    case Failure(e) => false
  }

  public Boolean isDate(String s ) {
    try {
      SimpleDateFormat fmt = new java.text.SimpleDateFormat("yyyy-MM-dd")
      fmt.parse(s)
      return true;
    }catch (Exception e) {
      return false;
    }
  }

//案例5
  def getDate(s: String): Either[Date,Throwable] = Try {
    val fmt = new java.text.SimpleDateFormat("yyyy-MM-dd")
    fmt.parse(s)
  } match {
    case Success(dt) => Left(dt)
    case Failure(e) => Right(e)
  }

  getDate("2020-12-11") match {
      case Left(dt) => println("the date is %s".format(dt))
      case Right(e) => throw e
  }
```

• Anonymous Function(匿名函数)

```
//匿名函数使用
def f = (i:Int) => i + 1
var l1 = List(100,200,300).map(_*3)
var l2 = List(100,200,300).map(f)
```

• Higher Order Functions(高阶函数)

- Higher Order Functions are functions that take other functions as parameters or whose result is a function. (高阶函数是将其他函数作为参数，或结果是函数的函数)

```
//定义类
class Decorator(left: String, right: String) {
  def layout(x: Any) = left + x.toString() + right
}

// 定义方法
def apply(f: Int => String, v: Int) = f(v)

//调用
val decorator = new Decorator("[", "]")
println(apply(decorator.layout, 7))
```

- **Nested Functions(嵌套函数)**

```
  def filter(xs: List[Int],threshold:Int):List[Int] = {
    def process(ys:List[Int]):List[Int] =
      if (ys.isEmpty) ys
      else if (ys.head < threshold) {
        ys.head :: process(ys.tail)
      }
      else
        process(ys.tail)
    process(xs)
  }
println(filter(List(1,2,3,4,5),3))
```

- **Currying(柯里化)**

  - Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.(方法可以定义多个参数列表。当用较少的参数列表调用一个方法时，这将产生一个以缺失的参数列表作为参数的函数)

```
// 定义 currying函数，二个参数列表
def modN(n: Int)(x: Int):Boolean = ((x % n) == 0)

// 4种变换形式
def f1(n: Int, x: Int) = modN(n)(x)
def f2(x: Int) = modN(10)(x)
def f3(n: Int) = modN(n)(10)
def f4 = modN(10)(_)

// 定义函数
  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

//高级调用
val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
println(filter(nums, modN(2)))
println(filter(nums, modN(3)))
```

- **Implicit Parameters(隐式参数)**

```
abstract class SemiGroup[A] {
def add(x: A, y: A): A
}

abstract class Monoid[A] extends SemiGroup[A] {
    def unit: A
}

implicit object StringMonoid extends Monoid[String] {
def add(x: String, y: String): String = x concat y
def unit: String = ""
}

implicit object IntMonoid extends Monoid[Int] {
def add(x: Int, y: Int): Int = x + y
def unit: Int = 0
}

def sum[A](xs: List[A])(implicit m: Monoid[A]): A = {
if (xs.isEmpty) m.unit
else m.add(xs.head, sum(xs.tail))
}
println(sum(List(1, 2, 3)))
println(sum(List("a", "b", "c")))

//cannot compile
def sum[A] (implicit m: Monoid[A])(xs: List[A]): A = {
//…
}

//implicit values can not be
top-level, they have to be
members of a template.
```

- **Named Parameters & Default Values(命名参数和默认值)**

  - Named Parameters

```
// 指定参数名称
  def printName(first:String, last:String) = {
    println(first + " " + last)
  }

// 指定参数名调用
printName("John","Smith")
printName(first = "John",last = "Smith")
printName(last = "Smith",first = "John")

//Default Parameter Values (默认参数值)
def initialCapacityByDefault(initialCapacity:Int = 16, loadFactor:Float = 0.75f)
:Unit = {}
```

- **Underscore _ in scala(下划线_在scala中)**

```
// 案例1. Pattern Match(模式匹配)
  def matchValue(x: Int): String = x match {
    case 1 => "one"
    case 2 => "two"
    case _ => "anything else"
  }

Some(5) match {case Some(_) => println("yes")}

// 案例2. Anonymous Function(匿名函数)
List(1, 2, 3, 4, 5).foreach(print(_))
List(1, 2, 3, 4, 5) foreach{ _ => print("hi") }
List(1,2,3,4,5).foreach( a => print(a))

// 案例3. Import
import scala.util.matching._

// 案例4. Property Setter(设置属性)
class Test {
    private var a = 0
    def age = a
    def age_=(n:Int) = {
      require(n>0)
      a = n
    }
  }

val t = new Test
t.age = 5
println(t.age)

// 案例5. PlaceHolder Syntax(占位符的语法)
List(1, 2, 3, 4, 5) map(_ + 2)
List(1, 2, 3, 4, 5) filter(_%2 == 0)
List(1, 2, 3, 4, 5) reduce(_ + _)
List(1, 2, 3, 4, 5) exists(_ > 3)
List(1, 2, 3, 4, 5) takeWhile(_ < 4)

// 案例6. Function Assignment(函数赋值)
def prt():String = {println("call");"ok"}
var s = prt()
println(s)
val f = prt _
f()
```

总结 (Summary)

- Scala Data Types & Collections
  - Null vs null
  - Nothing
  - Nil vs None
  - Option
- Scala Functions
  - Anonymous Functions
  - High-Order Functions
  - Nested Functions
- Mutable vs Immutable

作业

- 写一个函数，求一正整数的阶乘
- 求两个数的最大公约数
- 统计最长的单词
- 每个单词出现多少次
- scala的if else和java if else有什么区别?
- 如何用scala的方式将一个字符串转换为int
  - Try(s.toInt).toOption.getOrElse(-999)
- 从素组中找到第一个大于999的值，并返回该值和index，找不到返回None