为什么scala

scala不是一门纯粹的面向对象的语言，他强调函数式编程，函数式编程的两大核心理念是函数是一等的，以及程序中应该将值直接映射到输出，尽量的减少在函数中对程序的副作用。

- 一等函数

  函数可以做为别的函数的参数、返回值，赋值给变量

- 直接映射到输出

  操作直接将输入值映射到输出值，而不是当场修改数据。

变量定义：变量分为两种，一周是val类型，一种是var类型，val类型一旦被初始化就不能修改其值，var类型在整个生命周期之内都可以被重新赋值。

函数定义：def开始，接下来是函数名，之后吃参数列表，紧跟一个冒号，之后是返回类型，接下来是等号，花括号之内是函数体

```
 def max(x: int, y:Int): Int = {
  if(x > y)
    x
  else
    y
}
```

函数字面量：用圆括号括起来一组带名字的参数，一个又箭头和函数体

```
(x: Int, y: Int) => x + y
```

foreach,跟for遍历

```
args.foreach(arg => println(arg))

for(arg <- args)
  println(arg)

for(i <- 0 to 2)
  println(i)
//for括号中，左边是变量，右边是数组
```

方法调用：在scala中并没有操作符重载，1 + 2被解释成(1).+(2)调用Int的+方法

从文件读取文本行：

```
import scala.io.Source
if(args.length > 0){
  for(line <- Source.fromFile(args(0)).getLines())
    println(line.length + " " + line)
}
else
  Console.err.println("please enter filename")
```

类和对象

类的定义

```
class ClassName(){
} //类的A定义

val a = new ClassName() //实例化

a.method //调用字段方法
```

若是定义的返回值位uint，其作用就是强调函数的副作用，要是没有return语句，将会将函数中最后的结果做为返回值

单例对象

当有同名类时，单例对象位伴生对象，可以访问伴生类的所有成员变量，其可以存放静态方法，类中不能有静态方法

```
object ObjectName{
}
```

若无伴生类则位孤立对象，其中可以用maim方法

```
object Summer{
  def main(args: Array[String]) = {
    for(arg <- args)
      println("hello world")
  }
}

```

直接使用scala命令进行编译，时间会比较长，因为他会遍历jar所有的包，使用fsc进行编译，第一次时间比较长，之后会创建一个守护进程若是程序没有改动，则时间上会很快

基础类型跟操作

- **字面量**：是表达源码中一个固定值的方法
- **字符串字面量**：这里需要注意的是若是有 """括起来的字符串将不会进行转译
- **符号字面量**：'ident这样的形势，用来再动态语言中当作标识符的场合
- **字符串插值**：s" $name"将回去寻找变量name，对字符串中的name进行插值，还可以使用f插值如f"{math.Pi}%.5f"进行printf风格的插值，row插值将不会对\进行转译
- **操作符即方法**

调用1 + 3 这样的方法，会调用1.+(3)，同时，任何方法都是操作符，s.indexOf('o')可以写成s indexOf 'o'

前缀标识符：+ - ! ~会表示成p.unary_!这样的类似形式g

后缀标识符是那些不接受参数且再调用时没有英文句点圆括号的方法，可以再调用的时候省区圆括号，如s.length返回字符串s的长度

- **无符号右移**： >>>左移跟无符号右移都会自动填充0
- **对象相等**：直接使用==的方法，可以直接比较对象的内容上相等。
- **结合性**：任何:的方法都由他右边的对象调用，传入左侧的操作元。a:::b将会是b.:::(a)

函数式对象

如果一个类没有定义体，并不需要给出花括号，因为没有显示的构造方法，若需要检查一些前置条件的话，需要用到require这个方法

```
class Rational(n: Int, d: Int){
  require(d != 0)  //检查分母不能为0
  override def toString = n + "/" + d //没有参数的话可以直接写 =
}
```

n，d做为类参数，scala编译器会自动采集到这两个参数，并创建一个主构造方法，同时接受这两个参数。

重写toString方法：因为编写一个类之后，新建他会打印初该类的一些默认信息，要是没有从写toString方法的话会导致打印的只是地址

```
class Rational(n: Int, d: Int){
  override def toString = n + "/" + d
}
```

辅助构造方法：相当于多态，以def this打头，但是辅助构造放法必须调用主构造方法或者调用同一个类中的另外一个构造方法，但这个构造方法最终还是会调用主构造方法，主构造方法是类的单一入口，因为只有主构造方法才能调用超类

```
class Rational(n: Int, d: Int){
  def this(n: Int) = this(n, 1)
}
```

变量定义：先申明为val或var之后定义标识符，最后定义类型，若是val需要给出初始化参数（正常情况下），但是要是标识符以下划线结尾，需要再标识符与冒号之间加上空格

```
val n: Int = 1
val n_ : Int = 1
```

隐式转换：重载了操作符之后就可以进行运算了，比如 r * 2这样的方法没问题，操作符的左结核性会调用r对应的 * 方法，但是使用2 * r 这样的方法并不行，因为Int类型并没有定义 * r的方法，若想调用，需要加入隐式转换

```
implicit def intToRational(x: Int) = new Rational(x)
```

该方法将int转换为Rational类型，之后在调用Rational的 * 方法

Rational程序例子：

```
class Rational(d: Int, n: Int){
  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)
  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
      )

  def + (i: Int):Rational =
    new Rational(numer + i * denom, denom)

  def - (that: Rational): Rational =
    new Rational(numer * that.denom - that.numer * denom,
      denom * that.denom)

  def - (i: Int): Rational =
    new Rational(numer - i * denom, denom)

  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  def * (i: Int): Rational =
    new Rational(numer * i, denom)

  def / (that: Rational): Rational =
    new Rational(numer * that.denom, denom * that.numer)

  def / (i: Int): Rational =
    new Rational(numer, denom * i)

  override def toString = numer + "/" + denom
  private def gcd(a: Int, b: Int): Int =
    if(b == 0) a else gcd(b, a % b)

}
```

内建的控制结构

- **if表达式**

  首先测试条件，根据条件是否满足来执行两个不同代码分支中的一个

```
println(if (!args.isEnpty) args(0) else "default.txt")
```

- **for循环**

```
for (file <- filesName)
  println(file)
```

像`file <- filesName`这样的语法是生成器，遍历filsname中的所有元素

- **过滤**

```
for (file <- filesName if file.getName.endWith(".scala"))
  println(file)
```

使用if子句进行过滤

- **嵌套循环**

使用多个<-子句，将会先遍历外层迭代器，在根据条件遍历内层迭代器，还可以在中途绑定变量。

```
for (
  file <- filesName
  if file.getName.endWith(".scala")
  line <- fileLines(file)
    trimed = line.trim
  if trimed.matchs(pattern)
)println(file + ":" line.trim)
```

- **产出一个新的集合**

在for循环的后面加上`yield`关键字将会产生一个集合，如`for 子句 yield 代码体`

```
def scalaFiles =
  for{
        file <- filesName
        if file.getName.endsWith(".scala")
    }yield file
```

- **捕获处理异常**

try：代码体

cache：捕获处理异常，且里面用cace进行匹配

finally：不管咋样都要执行的代码

- **没有break跟continue**

通过使用变量var来进行控制，要是实在是需要break支持，可以在`scala.util.control.Braks._`中找到

函数跟闭包

- **函数字面量**

函数字面量存在于源码中，在运行时编译成类，并在运行时实例化为函数值，所以函数值对象形式存在于运行时

```
(x: Int) => x + 1
```

上述例子时函数字面量，左侧时参数列表，右侧是操作，多条语句的话，使用{}进行组合，=>的意思表示将左侧的内容转换为右侧的内容。

```
someNumbers.filter(x => x > 0)
```

接受一个函数做为入参，并对列表中的每个元素调用这个函数。同时由于知道了someNumbers是整数，所以x必定是整数，不用再申明类型

- **占位符语法**

  用占位符来表示一个或多个参数，只要满足每个参数只在函数字面量中出现一次就行，第一个下划线代表第一个参数，第二个下划线代表第二个参数，第三个下划线代表第三个参数，以此类推。

```
someNumber.filter(_ > 0)  //变量是啥都不管
```

- **部分应用函数**

  可以用_代替独立的参数列表，但是要与主函数之间加上空格。

```
def sum(a: Int, b: Int, c: Int): Int = a + b + c
val a = sum _  //_ 表示的是三个参数的参数列表 可以通过a(1, 2, 3)调用
val b = sum(1, _: Int, 3)  //可以调用b(2)
someNumbers.foreach(println _)
someNumbers.foreach(println)
```

最后的这两种形式，可以再明确需要函数的地方给出。

- **闭包**

  函数字面量创建出来的函数值被称为闭包，该名称源于“捕获”其自由变量从而“闭合”该函数字面量的操作。没有自由变量的字面量成为闭合语，比如(x: Int => x + 1)。任何带有自由变量的函数字面量称为开放语，比如(x: Int => x + more)

- **特殊函数的调用形式**

  - 重复参数

```
def echo(args: String*) =
  for (arg <- args) println(arg)
echo(arr: _*)  //传入一个String类型的数组
```

  - 缺省值参数，带名参数

    缺省值参数就是将参数提前赋值，与python中一致。

    带名参数的作用就是再调用过程中，参数的顺序可以被打乱。

- **尾递归**

  在函数的尾部自己调用自己，这样的话就是尾递归，尾递归会自动的优化成while循环的形式，减少栈的开销。但是只能对那些直接尾递归的函数进行优化，要是间接的调用尾递归并不会进行优化

```
def isEven(x: Int): Boolean =
  if (x == 0) true else isOdd(x - 1)

def isodd(x: Int): Boolean =
  if (x == 0) false else isEven(x - 1)  //这样的相互递归不会优化，最后一步调用的是另外的一个函数值，并不会进行优化
```

抽象控制

- **减少代码重复**

  利用一等函数与占位符消除代码的重复

```
object FileMatcher{
  private def filesHere = (new java.io.File(".")).listFiles
  private def filesMatching(matcher: String => Boolean) =
    for (file <- filesHere; if matcher(file.getName))
      yield file

  def filesEnding(query: String) =
    filesMatching(_.endsWith(query))

  def filesContaining(query: String) =
    filesMatching(_.contains(query))

  def filesRegex(query: String) =
    filesMatching(_.matches(query))
}

```

- **柯里化**

  支持多个参数列表编写的一种形式，可以通过传入多组参数来应用他，也可以部分应用，然后绑定到变量上，通过变量进行调用

```
def curriedSum(x: Int)(y: Int) = x + y
val second = curriedSum(1)_
val twoPlis = second(2)  //1 + 2 的值，通过部分绑定调用
```

- **编写新的控制结构**

  通过一等函数，与柯里化，将某个控制模式简化。

```
def withPrintWriter(flie: File)(op: PrintWriter => Unit){
  val writer = new printWriter(file)
  try{
    op(writer)
  } finally{
  writer.close()
  }
}

val file = new File("data.txt")
withPrintWriter(file){wirter => writer.println(new java.util.Data)}  //单个参数可以使用花括号
```

- **传名参数 - 传值参数**

  传值参数中表达式的值将先于函数被调用，而传名参数将会在函数中调用该名字的地方进行调用。其实就是一个参数求值先后的问题。

```
def byNameAssert(predicate: => Boolean) = {
  if (assertionsEnabled && !predicate)  //传名参数，运行到这里才会进行求值
    throw new AssertionError
}

def boolAssert(predicate: Boolean) = //传值参数，其实在这里predicate就进行求值了
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

组合跟继承

- **抽象成员**

  申明为一个没有实现方法的成员，包含其的类为抽象类，具体需要在class之前加上abstract

- **无参方法**

  当函数没有参数列表的时候，参数列表的()都可以省略不写，但是在当其不仅仅表示一个属性，而是将会对某些var产生副作用的时候，将建议写上圆括号

```
abstract class Element{
  def contents: Array[String]  //抽象方法，因此需要抽象类
  def height: Int = contents.length  //参数列表省略，调用的时候也可以不使用参数列表
  def width: Int = if (height == 0) 0 else contents(0).height
}
```

-