

Functional Scala

刘光聪

2016.10

Contents

- 编程范式
- 面向对象设计
- 函数式设计
- 设计本质
- 致谢

编程范式

主流编程范式

- ① 指令式: Imperative Programming
- ② 函数式: Functional Programming
- ③ 逻辑式: Logic Programming
- ④ 面向对象: Object Oriented Programming

大师互毆

“Object-oriented programming is an exceptionally bad idea which could only have originated in California.”

E.W. Dijkstra

“You probably know that arrogance, in computer science, is measured in nanodijkstras.”

- Alan Kay

不公开的较量

OO pattern/principle

- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern

FP pattern/principle

- Functions
- Functions
- Functions, also
- Functions
- Yes, functions
- Oh my, functions again!
- Functions
- Functions ☐

Scala

Scala = OO + FP

- 自由的
- 开放的
- 简洁的
- 高阶的

面向对象设计

需求

❶ 需求 1：判断某个单词是否包含数字

迭代 1

快速实现

numberbychapter

```
def hasDigit(word: String): Boolean = {  
  var i = 0  
  while (i < word.length) {  
    if (word.charAt(i).isDigit)  
      return true  
    i += 1  
  }  
  return false  
}
```

numberbychapter

迭代 1

for 推导式

numberbychapter

```
def hasDigit(word: String): Boolean = {  
  for (c <- word if c.isDigit)  
    return true  
  false  
}
```

numberbychapter

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母

迭代 2

复制-粘贴

numberbychapter

```
def hasUpper(word: String): Boolean = {  
  for (c <- word if c.isUpper)  
    return true  
  false  
}
```

numberbychapter

迭代 2

抽象

numberbychapter

```
trait CharSpec {  
  def satisfy(c: Char): Boolean  
}  
  
def exists(word: String, spec: CharSpec): Boolean = {  
  for (c <- word if spec.satisfy(c))  
    return true  
  false  
}
```

numberbychapter

匿名内部类

numberbychapter

```
exists(word, new CharSpec {  
  def satisfy(c: Char): Boolean = c.isDigit  
})  
  
exists(word, new CharSpec {  
  def satisfy(c: Char): Boolean = c.isUpper  
})
```

numberbychapter

复用对象

numberbychapter

```
object IsDigit extends CharSpec {  
  def satisfy(c: Char): Boolean = c.isDigit  
}  
  
object IsUpper extends CharSpec {  
  def satisfy(c: Char): Boolean = c.isUpper  
}  
  
exists(word, IsDigit)  
exists(word, IsUpper)
```

numberbychapter

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线

具名函数对象

numberbychapter

```
case class Equals(ch: Char) extends CharSpec {  
  def satisfy(ch: Char): Boolean = this.ch == ch  
}
```

```
exists(word, Equals('_'))
```

numberbychapter

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线
- ❹ 需求 4: 判断某个单词是否不包含下划线

修饰语义

```
numberbychapter
def exists(word: String, spec: CharSpec): Boolean = {
  n   for (c <- word if spec.satisfy(c))
  n   return true
  n   false
  n }
n
n def forall(word: String, spec: CharSpec): Boolean = {
  n   for (c <- word if !spec.satisfy(c))
  n   return false
  n   true
  n }
n
n case class Not(spec: CharSpec) extends CharSpec {
  n   def satisfy(ch: Char): Boolean = !spec.satisfy(ch)
  n }
n
n forall(word, Not(Equals('_')))
```

numberbychapter

消除重复

numberbychapter

```
private def comb(word: String, spec: CharSpec, shortcut: Boolean): \
  Boolean = {
  for (c <- word if spec.satisfy(c) == shortcut)
    return shortcut
  !shortcut
}

def exists(word: String, spec: CharSpec): Boolean =
  comb(word, spec, true)

def forall(word: String, spec: CharSpec): Boolean =
  comb(word, spec, false)
```

numberbychapter

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线
- ❹ 需求 4: 判断某个单词是否不包含 _
- ❺ 需求 5: 判断某个单词是否包含 _，或者 *

组合或

numberbychapter

```
case class Or(left: CharSpec, right: CharSpec) extends CharSpec {  
  def satisfy(c: Char): Boolean =  
    left.satisfy(c) || right.satisfy(c)  
}  
  
exists(word, Or(Equals('_'), Equals('*')))
```

numberbychapter

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线
- ❹ 需求 4: 判断某个单词是否不包含 _
- ❺ 需求 5: 判断某个单词是否包含 _，或者
- ❻ 需求 6: 判断某个单词是否包含空白符，但除去空格

组合与

numberbychapter

```
case class And(left: CharSpec, right: CharSpec) extends CharSpec {  
  def satisfy(c: Char): Boolean =  
    left.satisfy(c) && right.satisfy(c)  
}  
  
object IsWhitespace extends CharSpec {  
  def satisfy(c: Char): Boolean = c.isWhitespace  
}  
  
exists(word, And(IsWhitespace, Not(Equals(' '))))
```

numberbychapter

需求

- 1 需求 1: 判断某个单词是否包含数字
- 2 需求 2: 判断某个单词是否包含大写字母
- 3 需求 3: 判断某个单词是否包含下划线
- 4 需求 4: 判断某个单词是否不包含 _
- 5 需求 5: 判断某个单词是否包含 _， 或者
- 6 需求 6: 判断某个单词是否包含空白符， 但除去空格
- 7 需求 7: 判断某个单词是否包含字母 x， 且不区分大小写

修饰

numberbychapter

```
case class IgnoringCase(spec: CharSpec) extends CharSpec {  
  def satisfy(c: Char): Boolean = spec.satisfy(c.toLowerCase)  
}  
  
object IgnoringCase {  
  def equals(ch: Char) = IgnoringCase(Equals(ch.toLowerCase))  
}  
  
exists(word, IgnoringCase.equals('x'))
```

numberbychapter

需求

- ❶ 需求 1: 判断某个单词是否包含数字
- ❷ 需求 2: 判断某个单词是否包含大写字母
- ❸ 需求 3: 判断某个单词是否包含下划线
- ❹ 需求 4: 判断某个单词是否不包含 _
- ❺ 需求 5: 判断某个单词是否包含 _，或者
- ❻ 需求 6: 判断某个单词是否包含空白符，但除去空格
- ❼ 需求 7: 判断某个单词是否包含字母 x，且不区分大小写
- ❽ 需求 8: 判断某个单词满足某种特征，总时成功

```
sealed class Placeholder(bool: Boolean) extends CharSpec {
  def satisfy(c: Char): Boolean = bool
}

object Always extends Placeholder(true)
object Never extends Placeholder(false)

exists(word, Always)
exists(word, Never)
```

numberbychapter

增强 String

numberbychapter

```
implicit class RichString(s: String) {  
  def exists(spec: CharSpec): Boolean = comb(spec, true)  
  def forall(spec: CharSpec): Boolean = comb(spec, false)  
  
  private def comb(spec: CharSpec, shortcut: Boolean): Boolean = {  
    for (c <- s if spec.satisfy(c) == shortcut)  
      return shortcut  
    !shortcut  
  }  
}  
  
word.exists(IgnoringCase.equals('x'))
```

numberbychapter

函数式设计

函数式接口

函数类型

numberbychapter

```
trait CharSpec {  
  def satisfy(c: Char): Boolean  
}  
  
trait Function1[-T, +R] {  
  def apply(t: T): R  
  
  def compose[A](g: A => T): A => R = x => apply(g(x))  
  def andThen[A](g: R => A): T => A = x => g(apply(x))  
}  
  
val hasUpper: Char => Boolean = _.isUpper  
val hasDigit: Char => Boolean = _.isDigit
```

numberbychapter

递归

尾递归

numberbychapter

```
def exists(s: Seq[Char])(p: Char => Boolean): Boolean =  
  s match {  
    case h +: t if (p(h)) => true  
    case h +: t => exists(t)(p)  
    case _ => false  
  }  
  
def forall(s: Seq[Char])(p: Char => Boolean): Boolean =  
  s match {  
    case h +: t if (!p(h)) => false  
    case h +: t => forall(t)(p)  
    case _ => true  
  }
```

numberbychapter

表准库实现: String <-> StringOps

隐式转换: StringOps 实现了 exists, forall 等所有集合方法

```
object Predef {  
  implicit def wrap(x: String): StringOps = new StringOps(x)  
  implicit def unwrap(x: StringOps): String = x.repr  
}
```

StringOps 的继承树

隐式转换: StringOps 实现了 exists, forall 等所有集合方法

```
numberbychapter
trait IndexedSeqOptimized[+A, +Repr] with IndexedSeqLike[A, Repr] {
  private def prefixLength(p: A => Boolean, expectTrue: Boolean): \
  Int = {
    var i = 0
    while (i < length && p(apply(i)) == expectTrue) i += 1
    i
  }

  override def forall(p: A => Boolean): Boolean =
    prefixLength(p, true) == length

  override def exists(p: A => Boolean): Boolean =
    prefixLength(p, false) != length
}
```

占位符

numberbychapter

```
def always: Any => Boolean = _ => true
def never: Any => Boolean = _ => false

word.exists(always)
```

numberbychapter

逻辑相等性

numberbychapter

```
def equalTo[T](expected: T): T => Boolean = _ == expected  
word.exists(equalTo('_'))
```

numberbychapter

对象一致性

```
def same[T <: AnyRef](t: T): T => Boolean = t eq _
```

类型校验

numberbychapter

```
def instanceOf[T : ClassTag]: Any => Boolean = x =>
  x match {
    case _: T => true
    case _    => false
  }
```

numberbychapter

语法糖

numberbychapter

```
val nil    = equalTo[AnyRef](null)
val empty = equalTo("")
```

numberbychapter

组合器

numberbychapter

```
def allop[T](matchers: (T => Boolean)*): T => Boolean =  
  actual => matchers.forall(_(actual))  
  
def anyof[T](matchers: (T => Boolean)*): T => Boolean =  
  actual => matchers.exists(_(actual))
```

numberbychapter

语法糖

numberbychapter

```
def blank: String => Boolean =  
  ""\s*"".r.pattern.matcher(_).matches  
  
val emptyOrNull = anyof(nil, equalTo(""))  
val blankOrNull = anyof(nil, blank)
```

numberbychapter

修饰器

```
numberbychapter
def not[T](matcher: T => Boolean): T => Boolean = !matcher(_)
def is[T](matcher: T => Boolean): T => Boolean = matcher
word.exist(not(equalTo('_')))
```

语法糖

numberbychapter

```
def not[T](expected: T): T => Boolean = not(equalTo(expected))
def is[T](expected: T): T => Boolean = is(equalTo(expected))
word.exist(not('_'))
```

numberbychapter

字符修饰器

numberbychapter

```
type CharMatcher = Char => Char => Boolean
def ignoringCase(matcher: CharMatcher): CharMatcher = sc =>
  c => matcher(sc.toLowerCase)(c.toLowerCase)
word.exists(ignoringCase(equalTo('x')))
```

numberbychapter

字符串修饰器

numberbychapter

```
type StringMatcher = String => String => Boolean
def starts: StringMatcher = prefix =>
  _ startsWith prefix
def ends: StringMatcher = suffix =>
  _ endsWith suffix
def contains: StringMatcher = substr =>
  _ contains substr
def ignoringCase(matcher: StringMatcher): StringMatcher = substr =>
  str => matcher(substr.toLowerCase)(str.toLowerCase)
```

numberbychapter

流式接口

numberbychapter

```
word.exists { anyof(is('a'), is('z')) }  
word.exists { is('a') || is('z') }
```

numberbychapter

增强谓词: Matcher

numberbychapter

```
implicit class Matcher[-A](pred: A => Boolean) extends (A => \
Boolean) {
  self =>

  def &&[A1 <: A](that: A1 => Boolean): A1 => Boolean =
    x => self(x) && that(x)

  def ||[A1 <: A](that: A1 => Boolean): A1 => Boolean =
    x => self(x) || that(x)

  def unary_![A1 <: A]: A1 => Boolean =
    !self(_)

  def apply(x: A): Boolean = pred(x)
}
```

numberbychapter

设计本质

人类认知

心智的活动，除了尽力产生各种简单的认识之外，主要表现在如下三个方面：1）将若干简单认识组合为一个复合认识，由此产生出各种复杂的认识。2）将两个认识放在一起对照，不管它们如何简单或者复杂，在这样做时并不将它们合而为一。由此得到有关它们的相互关系的认识。3）将有关认识与那些在实际中和它们同在的所有其他认识隔离开，这就是抽象，所有具有普遍性的认识都是这样得到的。

John Locke, *An Essay Concerning Human Understanding*

（有关人类理解的随笔，1690）

设计本质

- 抽象
- 原子
- 组合

参考文献

推荐书籍

- Programming in Scala, 3th, Martin Odersky.
- Functional Programming in Scala, Paul Chiusano.
- Structure and Interpretation of Computer Programs, 2th, Harold Abelson, Gerald J. Sussman.

致谢