第12章：Scala 进阶

Objective(本课目标)

- [x] 类和特征（Class & Trait）及对象(Objects)
- [x] Scala(Regular Expression)正则表达式 API及使用
- [ ] Scala中的Java集成使用
- [ ] Scala异常处理

Class & Trait

- The default base/super class is scala.ScalaObject(默认继承ScalaObject)
- val vs. var Members (成员)
- Traits define object types by specifying the signature of the supported methods. (Trait通过指定支持的方法的签名，来定义对象类型)
    - Scala allows traits to be partially implemented (traits可以部分实现)
    - Variables can be defined within traits (变量可以定义在trait内)
    - No constructor parameters (不能有带参数的构造函数)

```
//定义 Trait
trait Similarity {
  def isSimilar(x: Any): Boolean
  def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}

// 继承Trait, 实现抽象方法
class Point(xc: Int, yc: Int) extends Similarity {
  var x: Int = xc
  var y: Int = yc
  def this() {
```

```
    this(0, 0)
  }
  def move(dx: Int, dy: Int) {
    x = x + dx
    y = y + dy
  }
  override def toString(): String = "(" + x + ", " + y + ")"

  def isSimilar(obj: Any) = obj.isInstanceOf[Point] && obj.asInstanceOf[Point].x
== x
}

//只能有一个基类,with后面的只能是trait
class A extends BClass with CTrait with D
class A extends CTrait with D
```

• (Alternative & Powerful) Class with Trait

```
//案例
trait Shape {
  def draw(): Unit
}

trait Square extends Shape {
  override def draw(): Unit = println("draw a square")
}

trait Triangle extends Shape {
  override def draw(): Unit = println("draw a triangle")
}

class Drawing { self: Shape =>
  def start():Unit = draw()
}

// 演示
(new Drawing with Square).start()
(new Drawing with Triangle).start()
```

• Trait vs. Abstract Class(区别)

- An Abstract can only extend on super-class, while a Trait can extend multiple Traits; (一个抽象类只可以有一个基类，而一个trait可以继承多个trait)
- A Trait can only have parameter-less constructor, while an Abstract can have multiple constructors with parameters); (trait是没有构造函数的，如果说有的话就是不带参数的。抽象类可以有多个构造函数)
- An Abstract class is fully interoperable(完全可互操作) with Java, while a Trait is interoperable only when

it doesn't contain any implementation code; (抽象类可以和Java可互操作，trait是不可以的，如果你的
trait里面没有任何代码实现也是可以的)

Singleton Objects(单例对象)

- Singleton Object
  - Define "static" methods & values that are not associated with individual instances of a class. (定义与类的各别实例无关的“static”方法和值)
  - Singleton object can extend classes and traits. (单利对象可以继承 class和trait)
- Companions (伙伴)
  - Most singleton objects are associated with a class of the same name (大多数singleton object和关联
    的calss名称一样)
    - Companion Object <- -> Companion Class
  - A class and its companion object, if any, must be defined in the same source file (class及其companion object(如果有的话)必须在同一个源文件中定义)
- scala中没有静态变量和静态方法
  - 如果你要创建静态的变量和方法，那么就定义一个object，将变量和方法定义在object中

```
//案例1
object ListCount {
    def sum(l: List[Int]): Int = l.sum
}

ListCount.sum(List[Int](1, 2, 3, 4, 5))

// Note: 可以定义object不定义class,也可以定义calss不定义object
// 案例2   private:私有构造函数
class Person private (first_name: String,last_name:String)
  object Person {
    def newInstance(first_name: String,last_name: String): Person = new
               Person(first_name,last_name)
  }

//案例3
class Connection private (connection_string:String) {}
  object Connection {
    var conn:Option[Connection] = None
    def create(connection_string: String): Connection = {
      if (conn == None) {
        conn = Some(new Connection(connection_string))
      }
      conn.get
```

```
    }
  }
```

Case Class

- Pattern Matching (模式匹配)
  - Scala allows to match on any sort of data with a first-match policy (Scala允许使用first-match策略匹
    配任何类型的数据)
  - Case Class
    - Export the constructor parameters (导出构造函数参数)
    - Provide a recursive decomposition mechanism via pattern-matching (通过模式匹配提供递归
      分解机制)
    - 主要是用来描述数据结构的

```
// 案例1
case class Employee(name:String,age:Int,address:String)
e1 match {
      case Employee(name,age,address) => println(name,age,address)
      case _ => println("no match")
}

// 案例2 定义样例类
abstract class Term(code: String)
case class Var(name: String) extends Term(name)
case class Fun(arg: String, body: Term) extends Term(arg)
case class App(f: Term, v: Term) extends Term("App")

//模式匹配调用
  def printTerm(term: Term) {
    term match {
      case Var(n) => print(n)
      case Fun(x, b) => printTerm(b)
      case App(f, v) => printTerm(f)
    }
  }

//测试
val x = Var("x")
val fun = Fun("f",x)
val app = App(fun,x)
printTerm(app)
```

Note：Abstract case class can be defined, however case class cannot be inherited.(可以定义抽象的case类，但是不能继承case类)

Case Object vs Enumeration(对比枚举)

- Case Objects vs Enumeration:
  - Enumerations are easier and less code. (枚举容易理解，代码更少)
  - Case objects are bit powerful because they naturally supports more fields than a Value based Enumeration which supports a name and ID. (Case object更加强大一些，支持更多的fields，而不是
    基于一个值的枚举)
  - Case objects are extensible (可以扩展)

```
//定义枚举
object Month extends Enumeration {
    type Month = Value
    val January, Februry, March, April, May, June, July, August, September,
October,November,December = Value
}

//定义case object
//被sealed 声明的 trait仅能被同一文件的的类继承
sealed trait Role { def name: String }
object Roles {
case object Admin extends Role { val name = "Administrator"}
case object Analyst extends Role { val name = "Analyst" }
case object Developer extends Role { val name = "Developer"}
}

// 模式匹配调用
  def isValidRole(r: Role) : Boolean = {
    r match {
      case Roles.Admin => true
      case Roles.Analyst => true
      case Roles.Developer => true
      case _ => false
    }
  }
isValidRole(Roles.Analyst)
```

Case Class vs. Class

- Case classes can be seen as plain and immutable data-holding objects that should exclusively depend on their constructor arguments (Case classes 可以被看作是单纯的、不可变的数据持有对象，它们应该完
  全依赖于它们的构造函数参数)

    - Immutable by default (默认是不可变的)
    - Decomposable through pattern matching (可通过模式匹配分解)
    - Compared by structural equality instead of by reference (比较内容而不是比较引用)
    - Succinct to instantiate and operate on (实例化和操作的简洁)
- If an object performs stateful computations on the inside or exhibits other kinds of complex behavior, it should be an ordinary class. (如果一个对象在内部执行有状态的计算，或者表现出其他复杂的行为，那么
  它应该是一个普通的类)

Extractor Objects

- Patterns can be defined independently of case classes, by using an extractor which is created by defining an unapply method in an object. (通过使用提取器，可以独立于case classes定义模式，提取器是
  通过在对象中定义unapply方法创建的)

```
//案例1
class Company(name: String) { /**/ }

object Company {
  def apply(name: String): Company = new Company(name)
}

//调用 calss Company
val c = new Company("ss")
// object 的 def apply
val c = Company("cm")

//案例2
object Twice {
    def apply(x: Int): Int = x * 2
    def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else None
}
//调用，会自动匹配unapply
val x = Twice(22)
x match {
  case Twice(n) => println(n)
  case _ => println("not match")
}
```

- apply method is to mimic the constructor (apply方法是模仿构造函数)
- case Twice(n) causes Twice.unapply to be called (case Twice(n) 会调用 Twice.unapply)

Case Class with Extractor Objects

- case classes automatically create a companion object with the same name as the class, which contains
  apply and unapply methods. (case classes 默认创建同名的 companion object 在 class中，包含了apply和
  unapply 方法)
  - The apply method enables constructing instances without prepending with new. (apply方法能够构
    建一个实例，不需要在前面加上 new)
  - The unapply extractor method enables the pattern matching (unapply extractor 在模式匹配中启动)
  - 模式匹配里面匹配的对象或者类，必须要是case class

Mixin Class Composition

- One class can extend only one base class but with multiple traits (一个类只能扩展一个基类但具有多个
  特征)

```
//案例
class A {
    def print(s: String):Unit = {
      println(s + "from A")
    }
  }

trait B {
    def print(s: String) {
      println(s + " from B")
    }
    def work(s: String) {
      println(s + " from B")
    }
  }

trait C {
    def work(s: String) {
      println(s + " from C")
    }
  }

class D extends A with B with C {
    override def print(s: String) { super.print(s) }
    override def work(s: String) { super.work(s) }
  }

class E extends A with C with B {
```

```
    override def print(s: String) { super.print(s) }
    override def work(s: String) { super[C].work(s) }
  }

//called
val d = new D()
d.print("123")
d.work("abc")

val e = new E()
e.print("123")
e.work("abc")
```

Generic Classes

- Generic Classes: classes parameterized with types (自定义类作为形参)

```
//定义 Generic Class
class Stack[T] {
    var elements: List[T] = Nil
    def push(x: T) { elements = x :: elements }
    def top: T = elements.head
    def pop(): T {
      var t = elements.head
      elements = elements.tail
      t
    }
  }

// 演示
val ms = new Stack[Int]()
ms.push(10)
ms.push(20)
val t = ms.pop()
```

Sequence Comprehensions(列表生成式)

- Scala offers a lightweight notation for expressing sequence comprehensions. Comprehensions have the form for (enumerators) yield e, where enumerators refers to a semicolon-separated list of enumerators (Scala提供了一个轻量级的符号来表示sequence comprehensions。理解的形式为(枚举数)产
  生e，其中枚举数引用以分号分隔的枚举数列表)

```
//案例1
def get(): Seq[Int] = {
    val lb = ListBuffer[Int]()
```

```
    for (i: Int <- Array(111, 222, 333)) {
      if (i % 2 == 0) {
        lb.append(i)
      }
    }
    lb
  }

// 案例2
def get_yield(): Seq[Int] = for(i: Int <- Array(111,222,333,444) if (i%2 == 0))
yield i

// 案例3
def even(from: Int, to: Int): List[Int] = {
    for (i <- List.range(from, to) if i % 2 == 0) yield i
  }

// 案例4
def foo(n: Int, v: Int) = for (i <- 0 until n; j <- i until n if i + j == v)
yield (i, j);

foo(20,30) foreach {
      case (i, j) => println("(" + i + ", " + j + ")")
}

// 案例5
for (i <- Iterator.range(0, 20); j <- Iterator.range(i, 20) if i + j == 32)
  println("(" + i + ", " + j + ")")
```

Access Modifiers

- Java Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

- Scala Access Levels

Modifier	Class	Companion	Subclass	Package	World
default	Y	Y	Y	Y	Y
protected	Y	Y	Y	N	N
private	Y	Y	N	N	N

Access Modifiers - private

- private
  - The private members can be accessed only from within the directly enclosing template and its companion module or companion class. (私有成员只能从直接封闭的模板，及其companion module or companion class中访问)
  - Top-level protected, private members can be defined (可以定义顶级受保护的私有成员)

```
// error
package com.bigdata.spark {
private class Zi {
  def parse( file: String ) : Unit = { }
}

package com.bigdata {
class RunDriver {
  val tp: Zi = new Zi()
}


// ok
package com.bigdata.spark {

class Zi {
// private[bigdata] 指定使用范围，在[bigdata]包里面的都可以读取
private[bigdata] def parse( file: String ) : Unit = {}
}

package com.bigdata {
class RunDriver {
  val tp: Zi = new Zi()
  tp.parse("test.txt")
}
```

Matching with Regular Expression(正则表达式)

```
//案例1
val pattern = "([a-zA-Z][0-9][a-zA-Z] [0-9][a-zA-Z][0-9])".r
```

```
    "A5S 1U1" match {
      case Pattern(zc) => println( "Valid zip-code: " + zc )
      case _ => println("Invalid zip-code")
}

//案例2
val pattern = "([A-Za-z0-9._%+-]+@[A-Za-z0-9.-]*\\.[A-Za-z]{2,6})".r
    "san.zhang@qq.com" match {
      case Pattern(email) => println( "Valid email address: " + email )
      case _ => println("Invalid email address")
    }
}

//案例3
val pattern = "(@TAG_[A-Za-z0-9]+)".r
    "@TAG_2ff4faca" match {
      case pattern(id) => println( "Valid user id: " + id)
      case _ => println("Invalid user id")
}

//案例4
"#222".matches("[a-zA-Z0-9]{4}") //false
"22Ba".matches("[a-zA-Z0-9]{4}") //true
```

- **Replacement in Strings(字符串替换)**

```
//案例1
import scala.util.matching.Regex
val nums = "[0-9]+".r.findAllIn("111 hello world scala 222")
println(nums.next())
println(nums.next())

//案例2
val str = "[0-9]+".r.replaceFirstIn("111 hello world scala 222", "777")
val str1 = "[0-9]+".r.replaceAllIn("111 hello world scala 222", "777")
```

- **Finding Patterns in Strings(在字符串中查找模式)**

```
//案例1
val date = """(\d\d\d\d)-(\d\d)-(\d\d)""".r // 等价于 \\d
    "2020-10-20" match {
      case date(year,month,_*) => println("The year & month are: " + year + "," +
month)
      case _ => println("Unmatched date")
    }
    "2020-10-20" match {
      case date(year,_,day) => println("The year & day are: " + year + "," + day)
      case _ => println("Unmatched date")
```

```
    }
}

//案例2
val embeddedDate = date.unanchored
val res = "Date: 2020-10-20 17:25:18 GMT (10 years, 28 weeks, 5 days, 17 hours
and 51 minutes ago)" match {
      case embeddedDate("2020", "10", "20") => "A Scala is born."
}

//案例3
val dates = "Important dates in history: 2020-10-20, 1958-09-05, 2010-10-06,
2011-07-15"
val firstDate = date findFirstIn dates getOrElse "No date found."
```

• Grouping in Pattern with Scala(使用Scala按匹配分组)

- Having phone # as follows:
  - 1 855 9215588
  - 86-10-9000239
- How to organize the phone #s by country code, area code and phone #? (如何按国家，区域编码，手机号组织)

```
//案例
    val pattern = """([0-9]{1,3})[ -]([0-9]{1,3})[ -]([0-9]{4,10})""".r
    val phones = List("136 698 16677", "86-10-80001364")
    phones.foreach {
      p => {
        val allMatches = pattern.findAllMatchIn(p)
        allMatches.foreach {
          m => println("CC=" + m.group(1) + "AC=" + m.group(2) + "Number=" +
m.group(3))
        }
      }
    }
```

• Java API in Scala(scala中使用JavaAPI)

```
//案例
import java.text.SimpleDateFormat
import java.util.{Calendar, Date}

val dateFmt = "yyyy-MM-dd"

def today(): String = {
    val date = new Date
```

```
    val sdf = new SimpleDateFormat(dateFmt)
    sdf.format(date)
}

def yesterday(): String = {
    val calender = Calendar.getInstance()
    calender.roll(Calendar.DAY_OF_YEAR, -1)
    val sdf = new SimpleDateFormat(dateFmt)
    sdf.format(calender.getTime())
}

def daysAgo(days: Int): String = {
    val calender = Calendar.getInstance()
    calender.roll(Calendar.DAY_OF_YEAR, -days)
    val sdf = new SimpleDateFormat(dateFmt)
    sdf.format(calender.getTime())
}
```

- **Exception Handling in Scala(Scala中的异常处理)**

```
//案例
def exe(args: Array[String]) {
    try {
      val f = new FileReader("input.txt")
    } catch {
      case ex: FileNotFoundException => {
        println("Missing file exception")
      }
      case ex: IOException => {
        println("IO Exception")
      }
    } finally {
      println("Exiting finally...")
    }
  }

throw new IllegalArgumentException
scala.util.control.Exception.allCatch.opt(“42”.toInt) // Some(42)
scala.util.control.Exception.allCatch.opt(“42a”.toInt) // None
scala.util.control.Exception.allCatch.toTry(“42”.toInt) // 42
scala.util.control.Exception.allCatch.toTry(“42a”.toInt) // Failure (e)
scala.util.control.Exception.allCatch.withTry(“42”.toInt) // Success(42)
scala.util.control.Exception.allCatch.withTry(“42a”.toInt) // Failure (e)
scala.util.control.Exception.allCatch.either(“42”.toInt) // Right(42)
scala.util.control.Exception.allCatch.either(“42a”.toInt) // Left(e)
```

- **作业**

- 给定一篇文章，完成词频统计，使用fold和foldLeft
- 假设类book有属性title和author,books是book的列表
    - 请将下列代码翻译成high order function
- 通过三种方式计算贷款利率

总结 (Summary)

- 类和特征(Class & Trait)及对象(Objects)
    - case class
    - Singleton Object
- Scala(Regular Expression)正则表达式 API及使用
    - Pattern Match
- Scala异常处理
    - try – catch – finally
    - allCatch -> Try, Option, Either
    - FailAsValue