MVC 例子

- 见lecture04.scala。

今天的内容

- 实例类和模式匹配
- 一般的类型和一般的方法
- 列表集合

实例类和模式匹配

+ 实例：算术表达式，（方法1）

```
new Sum(new Number(1), new Sum(new Number(3), new Number(7)))
```

```
abstract class Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
class Number(n: Int) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = error("Number.leftOp")
  def rightOp: Expr = error("Number.rightOp")
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = error("Sum.numValue")
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
```

```
def eval(e: Expr): Int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else error("unrecognized expression kind")
}
```

每个类内容繁冗

如果新加一个乘法类（Prod）？
扩展性差

实例：算术表达式，方法2

```
abstract class Expr {
  def eval: Int
}
class Number(n: Int) extends Expr {
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}


class Prod(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval * e2.eval
}
```

- 方法2：面向对象编程
  - 使eval变成每个类的方法
  - 简化
  - 加Prod（新***数据类型***）不改变其他类

实例：算术表达式

- 如果，新加表达式操作？（打印一个表达式）

方法1：只加一个独立函数

```
def print(e: Expr) {
  if (e.isNumber) Console.print(e.numValue)
  else if (e.isSum) {
    Console.print("(")
    print(e.leftOp)
    Console.print("+")
    print(e.rightOp)
    Console.print(")")
  } else error("unrecognized expression kind"
}
```

似乎回到了原点

方法2：每个类都要改

```
abstract class Expr {
  def eval: Int
  def print
}
class Number(n: Int) extends Expr {
  def eval: Int = n
  def print { Console.print(n) }
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
  def print {
    Console.print("(")
    print(e1)
    Console.print("+")
    print(e2)
    Console.print(")")
  }
}
```

实例类和实例对象

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

  def Number(n: Int) = new Number(n)
  def Sum(e1: Expr, e2: Expr) = new Sum(e1, e2)

  Sum(Sum(Number(1), Number(2)), Number(3))


 def n: Int          def e1: Expr, e2: Expr
```

- 使用case前缀
  - 隐式构建函数
  - 隐式实现方法：toString, equals, and hashCode
  - 隐式实现输入参数访问方法
  - 模式识别，根据构建函数的形式

模式匹配

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(l, r) => eval(l) + eval(r)
}
```

- Switch 语句的通用化
- Match方法对所有对象可见
- 模式变量，绑定值
- MatchError

模式匹配，替换过程

```
eval(Sum(Number(1), Number(2)))

->                    (by rewriting the application)

Sum(Number(1), Number(2)) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
}

->                    (by rewriting the pattern match)

eval(Number(1)) + eval(Number(2))

->                    (by rewriting the first application)

Number(1) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
} + eval(Number(2))

->                    (by rewriting the pattern match)

1 + eval(Number(2))

->* 1 + 2 -> 3
```

模式匹配，匿名函数

$$\texttt{\{ case } P_1 \texttt{ => } E_1 \ \ldots \ \texttt{case } P_n \texttt{ => } E_n \texttt{ \}}$$

$$\downarrow$$

$$\texttt{(x => x match \{ case } P_1 \texttt{ => } E_1 \ \ldots \ \texttt{case } P_n \texttt{ => } E_n \texttt{ \})}$$

+

一般的类型和一般的方法（Generic Types and Methods）

齐琦

为什么需要一般类型？

```
abstract class IntStack {
  def push(x: Int): IntStack = new IntNonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: Int
  def pop: IntStack
}
class IntEmptyStack extends IntStack {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class IntNonEmptyStack(elem: Int, rest: IntStack) extends IntStack {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

如何再定义一个字符串堆栈？

一般类型(generic types)
含有类型参数

```scala
abstract class Stack[A] {
  def push(x: A): Stack[A] = new NonEmptyStack[A](x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}
class EmptyStack[A] extends Stack[A] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class NonEmptyStack[A](elem: A, rest: Stack[A]) extends Stack[A] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

- 类型参数名字任意，在[ ]中

如何使用

```scala
val x = new EmptyStack[Int]
val y = x.push(1).push(2)
println(y.pop.top)
```

一般的方法（generic methods）

类型参数

值参数

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}
```

```
val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix[String](s1, s2))
```

- 有类型参数的方法
- 多形的(polymorphic = “having many forms”)
- 局部类型推理，在使用时可省略类型参数
  - isPrefix(s1,s2)

类型参数界限：为什么需要？

```
class EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, new EmptySet, new EmptySet)
}

class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
```

```
abstract class Set[A] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

- 加入类型参数[A]
- A 有定义 > 或 < 方法吗？

类型参数界限

```
trait Set[A <: Ordered[A]] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

```
class EmptySet[A <: Ordered[A]] extends Set[A] {
  def contains(x: A): Boolean = false
  def incl(x: A): Set[A] = new NonEmptySet(x, new EmptySet[A], new EmptySet[A])
}
```

```
class NonEmptySet[A <: Ordered[A]]
        (elem: A, left: Set[A], right: Set[A]) extends Set[A] {
  def contains(x: A): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: A): Set[A] =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

- 输入类型需要限制
- Trait Ordered[A]{…}
  - 本身类的值可相互比较
- 规定输入类型的值必须可比较（它是Ordered的子类型）

类型参数界限，继续

```
case class Num(value: Double) extends Ordered[Num] {
  def compare(that: Num): Int =
    if (this.value < that.value) -1
    else if (this.value > that.value) 1
    else 0
}
```

```
val s = new EmptySet[Num].incl(Num(1.0)).incl(Num(2.0))
s.contains(Num(1.5))
```

```
val s = new EmptySet[java.io.File]
                    ^ java.io.File does not conform to type
                      parameter bound Ordered[java.io.File].
```

- 使用之前的类定义值

类型参数界限，view bounds

```
trait Set[A <% Ordered[A]] ...
class EmptySet[A <% Ordered[A]] ...
class NonEmptySet[A <% Ordered[A]] ...
```

- 如果还不是Ordered的子类？
- Int, Double, String，从Java继承来的
- View bounds，<%
- 隐式转换存在就行

一般类型的变化标识(generic types' variance)

- 问题：如何避免程序中类型不匹配？

潜在的问题？

```
val x = new Array[String](1)
val y: Array[Any] = x
y(0) = new Rational(1, 2)  // this is syntactic sugar for
                           // y.update(0, new Rational(1, 2))
```

Java：在#3行，运行时检查(run-time check)类型

Scala：在#2行，静态时编译检查；Array 在Scala里不允许变体子类（non-variant subtyping）

同变（子类化）（Co-variant subtyping）

- If T 是 S的子类, then Stack[T] 是 Stack[S]的子类
- Scala里的一般类型(generic types)缺省是无变体子类化(non-variant subtyping)
- 强制同变的符号： +
  - Class Stack[+A]
- 纯函数世界，所有类型都可能成为同变的，但是当引入可变数据时，情况就不一样了。

反变子类化 (contra-variant subtyping)

- 表示：Class Stack[-A]
- 如果T 是 S 的子类，那么Stack[S] 是 Stack[T] 的子类

问题：如何验证变化标识的合理性？

- 如果array被定义成同变的(co-variant)，怎么及时检测出这个潜在的问题？
- Scala采用了一种保守的估计方法
- 同变类型参数只应出现在同变的位置上
  - 同变的位置有：
    - 类中值的类型
    - 类中方法的返回类型
    - 其他同变类型的类型输入参数
  - 非同变位置有：
    - 类中正式方法的参数类型

问题：如何验证变化标识的合理性？

```
class Array[+A] {
  def apply(index: Int): A
  def update(index: Int, elem: A)
                              ^ covariant type parameter A
                                appears in contravariant position.
}
```

Update 改变状态，影响了同变子类化的合理性

```
class Stack[+A] {
  def push(x: A): Stack[A] =
              ^ covariant type parameter A
                appears in contravariant position.
```

Stacks 是纯函数数据类型
Push 没有改变状态，但仍会被挑错，怎么办？

一般类型参数的：底界

```
class Stack[+A] {
  def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)
```

- T >: S , 类型参数T 只能是 类型S 的父类(supertypes)
- T >: S <: U
- Push in Stack，A不出现在push的参数类型位置；一个方法的类型参数的底界（这个位置是同变（co-variant）位置）。
- 不仅解决了技术问题，也通用化了push的定义。
- Push是一个多态方法(polymorphic method)
- What if push[B >: A] changed to [B<:A] ? What would happen?
- Can I push an Int value onto a String Stack?

EmptyStack的定义

```
object EmptyStack extends Stack[Nothing] { ... }

val s = EmptyStack.push("abc").push(new AnyRef())
```

- 对象不能有类型参数
- Nothing 是所有其他类型的子类；For co-variant stacks, Stack[Nothing] is a subtype of Stack[T]

Stack 的完整定义

```
abstract class Stack[+A] {
  def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}
object EmptyStack extends Stack[Nothing] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class NonEmptyStack[+A](elem: A, rest: Stack[A]) extends Stack[A] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

+ 一般类型实例
generic class: tuples and functions

Tuples

```
case class TwoInts(first: Int, second: Int)
def divmod(x: Int, y: Int): TwoInts = new TwoInts(x / y, x % y)

package scala
case class Tuple2[A, B](_1: A, _2: B)

def divmod(x: Int, y: Int) = new Tuple2[Int, Int](x / y, x % y)
```

类型参数可忽略

```
val xy = divmod(x, y)
println("quotient: " + xy._1 + ", rest: " + xy._2)

divmod(x, y) match {
  case Tuple2(n, d) =>
    println("quotient: " + n + ", rest: " + d)
}
```

- Tuple2: 含有两个值； Tuplen(): n个值
- 可直接用（x/y, x % y）

Tuples

```
def divmod(x: Int, y: Int): (Int, Int) = (x / y, x % y)
```

```
divmod(x, y) match {
  case (n, d) => println("quotient: " + n + ", rest: " + d)
}
```

函数(Functions)

- Scala, a functional language; functions are first-class **values**
- Also a object-oriented language; every value is an **object**.
- Functions are objects.

```
package scala
trait Function1[-A, +B] {
  def apply(x: A): B
}
```

- (T1, …, Tn) => S   缩写   Functionn[T1, …, Tn, S]
- f(x) shorthand for f.apply(x)

函数(Functions)

```
    val f: (AnyRef => Int) = x => x.hashCode()
    val g: (String => Int) = f

//      val g: (String => Int) = x => x.length()
//      val f: (AnyRef => Int) = g
```

- 反变类型参数的应用(contra-variant type parameter)
- Function subtyping is contra-variant in its argument type whereas co-variant in its result type.
- S' => T'
- S => T

函数的对象本质

```
val plus1: (Int => Int)  =  (x: Int) => x + 1
plus1(2)
```

通常的函数使用

```
val plus1: Function1[Int, Int] = new Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
plus1.apply(2)
```

实体化一个抽象类？行吗？

面向对象代码；**New Function1**构建了匿名类，实现了**apply**方法；F**unction1**是抽象类

```
val plus1: Function1[Int, Int] = {
  class Local extends Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
  }
  new Local: Function1[Int, Int]
}
plus1.apply(2)
```

使用命名的类扩展**Function1**

类型参数的子类变化控制的本质用意

- 子类值可以被赋给父类(变量)；反之不行。

列表集合（Lists）

海南大学

齐琦

List 简介

- 和数组(C或Java里的Array)的区别
  - 不可变的（immutable）；元素不能被赋值改变
  - 递归结构；数组是平的
  - 支持更丰富的操作

使用列表(Lists)

```
val fruit: List[String]    = List("apples", "oranges", "pears")
val nums : List[Int]       = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Int]       = List()
```

- 元素都是同类型的；**List[T]**
- 构造组件
  - Nil　　空表
  - ::　（『**cons**』）中间操作符，扩展表达
    - x :: **xs**　　第一个元素是**x**
    - 向右结合

```
val fruit  = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums   = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3  = (1 :: (0 :: (0 :: Nil))) ::
             (0 :: (1 :: (0 :: Nil))) ::
             (0 :: (0 :: (1 :: Nil))) :: Nil
val empty  = Nil
```

Lists 基本操作

- head
- tail
- isEmpty
- Head 和 tail 方法只为非空链表定义；空的调用会出错(exception)

```
empty.isEmpty    = true
fruit.isEmpty    = false
fruit.head       = "apples"
fruit.tail.head = "oranges"
diag3.head       = List(1, 0, 0)
```

使用举例：插入排序

```
def isort(xs: List[Int]): List[Int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))
```

- 如何实现insert函数？

列表模式(List pattern)

- :: 被定义为一个实例类(case class)，可以用模式匹配来分解链表

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => insert(x, isort(xs1))
}
```

where

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List() => List(x)
  case y :: ys => if (x <= y) x :: xs else y :: insert(x, ys)
}
```

List 类定义介绍

- 不是嵌入式类型；抽象类，和子类::, Nil.
- Co-variant 类型参数A
  - List[S] <: List[T] for all types S and T such that S <: T

```
package scala
abstract class List[+A] {
```

分解列表(Decomposing Lists)

```
def isEmpty: Boolean = this match {
  case Nil => true
  case x :: xs => false
}
def head: A = this match {
  case Nil => error("Nil.head")
  case x :: xs => x
}
def tail: List[A] = this match {

  case Nil => error("Nil.tail")
  case x :: xs => xs
}
```

List 方法

```
def length: Int = this match {
  case Nil => 0
  case x :: xs => 1 + xs.length
}
```

```
def last: A
def init: List[A]
```

```
def last: A = this match {
  case Nil      => error("Nil.last")
  case x :: Nil => x
  case x :: xs  => xs.last
}
```

- 长度方法
  - 如何实现尾递归形式
- Last element; all elements except the last
  - Have to traverse entire list

List 方法，继续

- Return a prefix , or a suffix, or both

```
def take(n: Int): List[A] =
  if (n == 0 || isEmpty) Nil else head :: tail.take(n-1)

def drop(n: Int): List[A] =
  if (n == 0 || isEmpty) this else tail.drop(n-1)

def split(n: Int): (List[A], List[A]) = (take(n), drop(n))


def apply(n: Int): A = drop(n).head
```

返回第n个元素；indices start at 0.
Xs.apply(3) or xs(3); 好像函数
子表 xs_m, …, xs_n-1 ， xs.drop(m).take(n-m)

拉链(Zipping lists)

$xs = \mathtt{List}(x_1, \ldots, x_n)$, and
$ys = \mathtt{List}(y_1, \ldots, y_n)$,

xs zip ys constructs the list $\mathtt{List}((x_1, y_1), \ldots, (x_n, y_n))$.

- Longer list will be truncated.
- Zip， a polymorphic method

```
def zip[B](that: List[B]): List[(a,b)] =
  if (this.isEmpty || that.isEmpty) Nil
  else (this.head, that.head) :: (this.tail zip that.tail)
```

在列表头添加元素

```
x :: y = y.::(x)        whereas        x + y = x.+(y)

x :: y :: z = x :: (y :: z)    whereas    x + y + z = (x + y) + z
```

- ::， implemented as a method in class List
- 向右结合

```
def ::[B >: A](x: B): List[B] = new scala.::(x, this)
```

串联列表（Concatenating lists）

- :::，右结合，右手操作元素的一个方法。

```
xs ::: ys ::: zs  =   xs ::: (ys ::: zs)
                  =   zs.:::(ys).:::(xs)
```

```
def :::[B >: A](prefix: List[B]): List[B] = prefix match {
  case Nil => this
  case p :: ps => this.:::(ps).::(p)
}
```

反转列表

- Reverse 方法

```
def reverse[A](xs: List[A]): List[A] = xs match {
  case Nil => Nil
  case x :: xs => reverse(xs) ::: List(x)
}
```

- 这个实现效率低，为什么？
- 时间复杂度

$$n + (n-1) + ... + 1 = n(n+1)/2$$