今天的内容

- 一般类的实例
  - Tuples
  - Functions
- 列表集合

Stack 的完整定义

```scala
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
}
```

参考Lecture08.scala

一般类型实例
generic class: tuples and functions

Tuples

```scala
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

```scala
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；Function1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 = List(x_1, ..., x_n) , and
ys = List(y_1, ..., y_n) ,

xs zip ys constructs the list List((x_1, y_1), ..., (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

$$n + (n-1) + \ldots + 1 = n(n+1)/2$$

Merge sort

```scala
def msort[A](less: (A, A) => Boolean)(xs: List[A]): List[A] = {
  def merge(xs1: List[A], xs2: List[A]): List[A] =
    if (xs1.isEmpty) xs2
    else if (xs2.isEmpty) xs1
    else if (less(xs1.head, xs2.head)) xs1.head :: merge(xs1.tail, xs2) else xs2.head :: merge(xs1, xs2.tail)
  val n = xs.length / 2
  if (n == 0) xs
  else merge(msort(less)(xs take n), msort(less)(xs drop n))
}
```

```scala
val lst = msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
println(lst)

val intSort = msort((x: Int, y: Int) => x < y) _
val reverseSort = msort((x: Int, y: Int) => x > y) _

println(intSort(List(6,5,4,3,2,1)))
println(reverseSort(List(1,2,3,4,5,6)))
```

+

List 的高阶方法

List上的计算模式总结

- 计算模式
  - 对每个元素进行转换
  - 选出满足某个条件的所有元素
  - 对元素进行某种方式上的组合
- 通过高阶函数来实现以上模式
- List 的方法

Mapping (映射)

```
abstract class List[A] { ...
  def map[B](f: A => B): List[B] = this match {
    case Nil => this
    case x :: xs => f(x) :: xs.map(f)
  }

def scaleList(xs: List[Double], factor: Double) =
  xs map (x => x * factor)


def column[A](xs: List[List[A]], index: Int): List[A] =
  xs map (row => row(index))
```

- 变换每个元素

For each 方法

- 对每个元素应用一个函数，但不返回一个列表结果
- 为了副作用(side effect)而设
  - In computer science, a function or expression is said to have a **side effect** if, in addition to returning a value, it also modifies some state or has an *observable* interaction with calling functions or the outside world. For example, a function might modify a global variable or static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, or call other side-effecting functions.

```
def foreach(f: A => Unit) {
  this match {
    case Nil => ()
    case x :: xs => f(x); xs.foreach(f)
  }
}

 xs foreach (x => println(x))
```

Filtering（过滤列表）

- 根据一个原则来选择元素

```
def posElems(xs: List[Int]): List[Int] = xs match {
  case Nil => xs
  case x :: xs1 => if (x > 0) x :: posElems(xs1) else posElems(xs1)
}
```

```
def filter(p: A => Boolean): List[A] = this match {
  case Nil => this
  case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
}
```

```
def posElems(xs: List[Int]): List[Int] =
  xs filter (x => x > 0)
```

Forall , exists

- Forall : all elements satisfy a condition
- Exists: exists an element that satisfies a condition

```
def forall(p: A => Boolean): Boolean =
  isEmpty || (p(head) && (tail forall p))
def exists(p: A => Boolean): Boolean =
  !isEmpty && (p(head) || (tail exists p))
```

```
package scala
object List { ...
  def range(from: Int, end: Int): List[Int] =
    if (from >= end) Nil else from :: range(from + 1, end)
```

```
def isPrime(n: Int) =
  List.range(2, n) forall (x => n % x != 0)
```

质数定义

折叠和减少列表(folding and reducing)

List(x_1, ..., x_n).reduceLeft(op) = (...(x_1 op x_2) op ...) op x_n

```
def sum(xs: List[Int])      =  (0 :: xs) reduceLeft {(x, y) => x + y}
def product(xs: List[Int])  =  (1 :: xs) reduceLeft {(x, y) => x * y}
```

(List(x_1, ..., x_n) foldLeft z)(op)   =   (...(z op x_1) op ...) op x_n

```
def sum(xs: List[Int])      =  (xs foldLeft 0) {(x, y) => x + y}
def product(xs: List[Int])  =  (xs foldLeft 1) {(x, y) => x * y}
```

- Combine elements of a list with some operator.

FoldRight , ReduceRight

List(x_1, ..., x_n).reduceRight(op) = x_1 op (... (x_{n-1} op x_n)...)
(List(x_1, ..., x_n) foldRight acc)(op) = x_1 op (... (x_n op acc)...)

```
def reduceRight(op: (A, A) => A): A = this match {
  case Nil => error("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))

}
def foldRight[B](z: B)(op: (A, B) => B): B = this match {
  case Nil => z
  case x :: xs => op(x, (xs foldRight z)(op))
}
```

- Produce right-leaning trees.

Abbreviations for foldLeft and foldRight

```
def /:[B](z: B)(f: (B, A) => B): B = foldLeft(z)(f)
def :\[B](z: B)(f: (A, B) => B): B = foldRight(z)(f)
```

(z /: List(x_1, ..., x_n))(op) = (...(z op x_1) op ...) op x_n

(List(x_1, ..., x_n) :\ z)(op) = x_1 op (... (x_n op z)...)

Nested Mappings

- 高阶函数可以替代嵌套循环
- Find all pairs of positive integers I and j, where 1<=j<i<n such that i+j is prime.

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

```
List.range(1, n)
  .map(i => List.range(1, i).map(x => (i, x)))
  .foldRight(List[(Int, Int)]()) {(xs, ys) => xs ::: ys}
  .filter(pair => isPrime(pair._1 + pair._2))
```

+

Flattening Maps

- flatMap
  - Combination of mapping and then concatenating sublists

```
abstract class List[+A] { ...
  def flatMap[B](f: A => List[B]): List[B] = this match {
    case Nil => Nil
    case x :: xs => f(x) ::: (xs flatMap f)
  }
}

List.range(1, n)
  .flatMap(i => List.range(1, i).map(x => (i, x)))
  .filter(pair => isPrime(pair._1 + pair._2))
```