Scala Collections API

Expressivity and Brevity upgrade from Java

Dhananjay Nene

IndicThreads Conference On Java

December 2, 2011

Scala Collections

Some of the basic scala collection types are :

- HashSet
- Sequence
- Vector
- List
- Stream
- Queue
- Stack

Largely focus on Lists

We shall predominantly focus on lists with a slightly lesser focus on other types. Most of the frequently used operations are available on lists, so these are a good starting point.

Simple List Construction

```
val cities = List("Pune", "Mumbai", "Bangalore", "Kolkata", "New Delhi")
val primes = List(2,3,5,7,11,13)
```

Note :

- No "new" .. but thats syntax sugar
- No type declaration. The types are inferred to be List[String] & List[Int]
- No semi colons at the end

A simple list iteration

Code closest to java

```
val numbers = List(1,3,5,9)
for(n <- numbers) {
  println(n)
}
```

But thats not the only way

A simple recursion traversal

```
def recursionTraversal(numbers: List[Int]): Unit = {
  if ((numbers length) > 0) {
    println(numbers head); recursionTraversal(numbers tail)
  }
}

recursionTraversal(List(1,3,5,9))
```

- Did you notice the function arguments ?
- And what is "numbers length"? Ans: It is the same as numbers.length().
- Can you figure out what "numbers head" and "numbers tail" do?

A different recursion traversal

```
patternMatchingTraversal(1 :: (3 :: (5 :: (9 :: Nil))))
def patternMatchingTraversal(numbers: List[Int]): Unit = {
  numbers match {
    case h :: t =>
      println(h)
      patternMatchingTraversal(t)
    case _ => ; // DO Nothing
  }
}
```

- Note how the list is created. And the pattern matching too

Other List Methods

- *length* : Length of list
- *reverse* : Reverse a list
- *toArray* : Convert to array
- *iterator* & *next* : Iterator protocol
- *mkString* : Concatenate elements into a string
- *apply* : Random element access
- *init* : All but last element in a list
- *last* : Last element in a list

Patterns of collection computations

- There are some frequently occurring general purpose patterns used over collections.
- These are all a part of the scala collections API.
- This is one area where you will find a significant expressivity and brevity upgrade over java.

map : Applying to each element

```
def double(x: Int) = x * 2

List(1,3,5) map double
// OR
List(1,3,5) map {n => n * 2}
// OR
List(1,3,5) map {_ * 2}
// All result into List(2, 6, 10)
```

- Note the inline and anonymous function declarations the the parameter "_"

Equivalent Java Code

```
public static List<Integer>
    doubleAll(List<Integer> numbers) {
    List<Integer> doubled = new LinkedList<Integer>();
    for(Integer n : numbers) {
      doubled.add(n * 2);
    }
    return doubled;
}
public static void main(String[] args) {
    System.out.println(doubleAll(Arrays.asList(1,3,5)));
}
```

Regarding map

- In the earlier example, how would you write tripleAll instead of doubleAll? Ans: Copy the entire function and modify slightly.
- Map is a very frequently occurring operation on collections
- The process of mapping is abstracted away orthogonally from the operation performed on each element
- This results in a substantial reduction in size
- This also removes a lot of ceremony and boilerplate.

Regarding map .. 2

- You focus on the very essence of what is being done eg. *myList map double*
- Stripping away of ceremony & boiler plate, and ability to focus on the essence, is a consistent characteristic of many methods on List & other collections
- If you think about it, you will find lots and lots of uses of map in your regular tasks eg. computation of tax on each item in a list
- When we pass a function as an argument to a function (or expect a function as a return type), thats termed Higher Order Functions (HOFs). eg. *double*

filter : Selecting elements

```
// select only odd numbers from a list
List(1,2,3,4,5) filter { _%2 != 0}
// results into List(1, 3, 5)
```

- Again this is a frequently used pattern. Again scala strips away the ceremony and allows you to focus on the essence. (I'll stop repeating this)
- Also notice the predicate selecting odd numbers being used as a HOF (I'll stop repeating this too).

fold : Summarising / Aggregation

```
(0 /: List(1,2,3,4)) {(sum,num) => sum + num}
List(1,2,3,4).foldLeft(0){(sum,num) => sum + num}
List(1,2,3,4).foldLeft(0){_ + _}
// All result in the value 10
```

- Yet again, a frequently found pattern. eg. creating total for an invoice

Other Patterns

- *take* and *takeWhile* : Select first few elements of a list
- *drop* and *dropWhile* : Select last few elements of a list
- *zip* and *unzip* : Combine two lists into list of pairs (& vice-versa)
- *partition* : Split a list into two based on a predicate
- *forall* and *exists* : Test all or any one element of a list for a predicate
- *sortWith* : Sort given a comparation predicate
- *flatten* : Flatten a List of List of *X* into a List of *X*
- *flatMap* : Flatten a List of List obtained by mapping a function that returns a list

Back to iteration over a list

Earlier we saw a for loop and a recursive method of traversing a list. One might be tempted to use map to print all elements of a list

```
// Whats the return value ?
List(1,2,3,4) map { println _ }

// When you are not interested in the return value
// interested only in the side effects of the function
List(1,2,3,4) foreach { println _ }
```

For Comprehensions

Combines for loops, filters, and assignments

```
case class Cell(val row: Int,  val col: Int)
def isOccupied(c: (Int, Int)) = (c._1 + c._2) % 2 == 0

def findFreeCells(): List[String] = {
  val p = for(i <- 1 to 3;
              j <- 1 to 3;
              cell = (i,j);
              if (!isOccupied(cell))
              ) yield (List(i,j).mkString("C","",""))
  p toList
}
```

Equivalent Java Code

```
class Cell {
    private Integer row;
    private Integer col;

    public Cell(Integer row, Integer col) {
        this.row = row;
        this.col = col;
    }

    public Integer getRow() {
        return this.row;
    }
```

Java code continued...

```
    public Integer getCol() {
        return this.col;
    }
}

public static boolean isOccupied(Cell cell) {
    return ((cell.getRow() + cell.getCol()) % 2) == 0;
}
```

Java code continued...

```
public static List<String> findFreeCells() {
    List<String> retList = new LinkedList<String>();
    for(int i = 1 ; i <= 3 ; i++) {
        for(int j = 1 ; j <= 3 ; j++) {
            Cell c = new Cell(i,j);
            if (isOccupied(c) == false) {
                retList.add("C" + c.getRow() + c.getCol());
            }
        }
    }
    return retList;
}
```

Aside: Case classes

- Case classes are simple java beans like classes.
- scala throws in getters / setters / hashCode / toString / equals etc. automatically.
- Also quite useful when pattern matching (beyond scope of this talk)
- Can also add additional methods

```
case class Cell(val row: Int,  val col: Int) {
  def isOccupied = (row + col) % 2 == 0
}
```

Maps

```
val states = Map("GJ" -> "Gujarat", "MH" -> "Maharashtra", "KL" -> "Kerala")
states get "GJ"
states("GJ")
states("ZZ") // will raise an exception
states.getOrElse("ZZ","Unknown")
states + ("SK" -> "Sikkim")
states - "KL"
states filterKeys  {_(0) == 'G'}
states filter {_._2.length() <=  7}
states transform  {(key,value) => value.toLowerCase }
```

Other collections

- Set (HashSet / TreeSet) : Fairly similar semantics. Few extra eg. intersect, union, diff
- Vector : Indexed access, Constant time non head access
- Queue
- Stack

Streams

Lazily evaluated.

```
def fibonacci(first: Int, second: Int): Stream[Int] =
  first #:: fibonacci(second, first + second)
val f = fibonacci(1,1)
// result = List(1, 1, 2, 3, 5, 8, 13)
f.takeWhile {_ < 20} toList
```

Mutable or Immutable

- Most collections come in both mutable and immutable variants
- (Surprisingly?) the preferred usage is immutable
- A detailed discussion on the matter is beyond the scope of the talk
- A good java equivalent to immutable classes is String. Think the same design principles applied across the collections.
- Just like string, there exist mutable Buffer/Builder classes to slowly collect elements of a collection and finally convert them into an immutable collection.

Parallel Collections

```
(1 to 10) foreach  {println}
(1 to 10).par foreach  {println}
```

Sample Code

(To be shown separately)

```
package ordersystem
import scala.collection.mutable

case class CustomerOrder(     val orderId: String,
                              val customerCode: String,
                              val productCode: String,
                              val quantity: Int)

case class ProductMaster(     val productCode: String,
                              val manufacturingTime: Int,
                              val components: List[(String,Int)])

case class ShippingNote(      val customerCode: String,
                              val items: List[ShipItem])

case class ShipItem(          val customerCode: String,
                              val orderId: String,
                              val productCode: String,
                              val quantity: Int)

case class MakeProduct(       val customerCode: String,
                              val orderId: String,
```

```
                            val productCode: String,
                            val quantity: Int)

object OrderSystem {
  var inventory = mutable.Map("foo" -> 14,
                      "bar" -> 14,
                      "baz" -> 14)

  val productMaster =  List( ProductMaster("foo",5,List(("zoo", 7),("zar",9))),
                             ProductMaster("bar",7,List(("boo", 9),("moo",6))),
                             ProductMaster ("baz",6,List(("zar",4),("moo",5)))) map {p => p.productCode -> p} toMap

  val itemInventory = mutable.Map("zoo" -> 80, "zar" -> 100, "boo" -> 25, "moo" -> 30)

  def shipOrMake(order: CustomerOrder) = {
    // find existing available products in stock
    val available = inventory.getOrElse(order.productCode,0)

    // compute products that can be shipped, updated stocks and those that need to be made
    val (ship, stock, make)  =
      if (order.quantity < available) (order.quantity, available - order.quantity, 0)
      else (available, 0, order.quantity - available)
    inventory.put(order.productCode,stock)

    // create shipment instruction
    val shipItem = if (ship > 0) Some(ShipItem(order.customerCode, order.orderId, order.productCode, ship)) else None
    println("Ship:" + shipItem)

    // and/or make product instruction
    val makeProduct = if (make > 0) Some(MakeProduct(order.customerCode, order.orderId, order.productCode, make)) else None
    (shipItem, makeProduct)
  }

  def main(args: Array[String]) = {
    val orders = List(
      CustomerOrder("c11","c1", "foo", 10),
      CustomerOrder("c12","c1", "bar", 12),
      CustomerOrder("c12","c1", "baz", 14),
      CustomerOrder("c21","c2", "foo", 12),
      CustomerOrder("c22","c2", "bar", 10),
      CustomerOrder("c22","c2", "baz", 12))

    // find which products to ship directly and which to make
    val shipMakeProducts = orders map shipOrMake // will give shipitem and makeitem

    // create ship item notices
    val shipItems = (shipMakeProducts map {_._1} flatten) groupBy {_.customerCode} map { x => ShippingNote(x._1,x._2) };
    println(shipItems)

    // create make product notices
    val makeProducts = shipMakeProducts map {_._2} flatten  ;
    println(makeProducts)

    val todo = makeProducts map { order =>
      // for each product that has to be made
      // get the corresponding product master
```

```
      println("Processing Make Product Order:" + order)
      productMaster.get(order.productCode) match {
        case Some(master) =>
          // for each of the components required
          master.components map  {
              // compute the quantities of items required
              pmc => (pmc._1, pmc._2 * order.quantity)
            } map { requiredPair =>

              val item = requiredPair._1
              val itemQuantity = requiredPair._2

              // compare with available quantities
              val available = itemInventory.getOrElse(item,0)
              // requisition if not available
              val (used, requisitioned) = if (available >= itemQuantity) {
                (itemQuantity,0)
              } else {
                (available, itemQuantity - available)
```

```
              }
              itemInventory.put(item,available - used)
              println("Using " + used + " units of " + item + " and requisitioning " + requisitioned)
              (item,used,requisitioned)
            }
        case None =>
          throw new Exception("product not in product master")
      }
    }
    // flatten the resultant list and group by item code
    val itemMap = (todo flatten) groupBy {_._1}
    println(itemMap)
    // for each item, fold the corresponding list to arrive at cumulative totals
    val itemMap2 = itemMap map {keyVal =>
      (keyVal._1,((0,0) /: keyVal._2){(total,each) => (total._1 + each._2, total._2 + each._3)})
    }
    println(itemMap2)
  }
}
```

Thank You

Contact Information:

Dhananjay Nene

Vayana Services

http://blog.dhananjaynene.com

http://twitter.com/dnene

firstname dot lastname at gmail dot com