More collections

«Previous Next»

Scala provides a nice set of collection implementations. It also provides some abstractions for collection types. This allows you to write code that can work with a collection of `Foo`s without worrying whether that collection is a `List`, `Set`, or what-have-you.

This page offers a great way to follow the default implementations and links to all the scaladoc.

- Basics Collection types you'll use all the time
- Hierarchy Collection abstractions
- Methods
- Mutable
- Java collections just work

The basics

List

The standard linked list.

```
scala> List(1, 2, 3)
res0: List[Int] = List(1, 2, 3)
```

You can cons them up as you would expect in a functional language.

```
scala> 1 :: 2 :: 3 :: Nil
res1: List[Int] = List(1, 2, 3)
```

See also API doc

Set

Sets have no duplicates

```
scala> Set(1, 1, 2)
res2: scala.collection.immutable.Set[Int] = Set(1, 2)
```

See also API doc

Seq

Sequences have a defined order.

```
scala> Seq(1, 1, 2)
res3: Seq[Int] = List(1, 1, 2)
```

(Notice that returned a List. `Seq` is a trait; List is a lovely implementation of Seq. There's a factory object `Seq` which, as you see here, creates Lists.)

See also API doc

Map

Maps are key value containers.

```
scala> Map('a' -> 1, 'b' -> 2)
res4: scala.collection.immutable.Map[Char,Int] = Map((a,1), (b,2))
```

See also API doc

The Hierarchy

These are all traits, both the mutable and immutable packages have implementations of these as well as specialized implementations.

Traversable

All collections can be traversed. This trait defines standard function combinators. These combinators are written in terms of `foreach`, which collections must implement.

See Also API doc

Iterable

Has an `iterator()` method to give you an Iterator over the elements.

See Also API doc

Seq

Sequence of items with ordering.

See Also API doc

Set

A collection of items with no duplicates.

See Also API doc

Map

Key Value Pairs.

See Also API doc

The methods

Traversable

All of these methods below are available all the way down. The argument and return types types won't always look the same as subclasses are free to override them.

```
def head : A
def tail : Traversable[A]
```

Here is where the Functional Combinators are defined.

`def map [B] (f: (A) => B) : CC[B]`

returns a collection with every element transformed by `f`

`def foreach[U](f: Elem => U): Unit`

executes `f` over each element in a collection.

`def find (p: (A) => Boolean) : Option[A]`

returns the first element that matches the predicate function

`def filter (p: (A) => Boolean) : Traversable[A]`

returns a collection with all elements matching the predicate function

Partitioning:

`def partition (p: (A) ⇒ Boolean) : (Traversable[A], Traversable[A])`

Splits a collection into two halves based on a predicate function

`def groupBy [K] (f: (A) => K) : Map[K, Traversable[A]]`

Conversion:

Interestingly, you can convert one collection type to another.

```
def toArray : Array[A]
def toArray [B >: A] (implicit arg0: ClassManifest[B]) : Array[B]
def toBuffer [B >: A] : Buffer[B]
def toIndexedSeq [B >: A] : IndexedSeq[B]
def toIterable : Iterable[A]
def toIterator : Iterator[A]
def toList : List[A]
def toMap [T, U] (implicit ev: <:<[A, (T, U)]) : Map[T, U]
def toSeq : Seq[A]
def toSet [B >: A] : Set[B]
def toStream : Stream[A]
def toString () : String
def toTraversable : Traversable[A]
```

Let's convert a Map to an Array. You get an Array of the Key Value pairs.

```
scala> Map(1 -> 2).toArray
res41: Array[(Int, Int)] = Array((1,2))
```

Iterable

Adds access to an iterator.

```
  def iterator: Iterator[A]
```

What does an Iterator give you?

```
def hasNext(): Boolean
def next(): A
```

This is very Java-esque. You often won't see iterators used in Scala, you are much more likely to see the functional combinators or a for-comprehension used.

Set

```
  def contains(key: A): Boolean
  def +(elem: A): Set[A]
  def -(elem: A): Set[A]
```

Map

Sequence of key and value pairs with lookup by key.

Pass a List of Pairs into apply() like so

```
scala> Map("a" -> 1, "b" -> 2)
res0: scala.collection.immutable.Map[java.lang.String,Int] = Map((a,1), (b,2))
```

Or also like:

```
scala> Map(("a", 2), ("b", 2))
res0: scala.collection.immutable.Map[java.lang.String,Int] = Map((a,2), (b,2))
```

DIGRESSION

What is `->` ? That isn't special syntax, it's a method that returns a Tuple.

```
scala> "a" -> 2

res0: (java.lang.String, Int) = (a,2)
```

Remember, that is just sugar for

```
scala> "a".->(2)

res1: (java.lang.String, Int) = (a,2)
```

You can also build one up via `++`

```
scala> Map.empty ++ List(("a", 1), ("b", 2), ("c", 3))
res0: scala.collection.immutable.Map[java.lang.String,Int] = Map((a,1), (b,2), (c,3))
```

Commonly-used subclasses

HashSet and HashMap Quick lookup, the most commonly used forms of these collections. HashSet API, HashMap API

TreeMap A subclass of SortedMap, it gives you ordered access. TreeMap API

Vector Fast random selection and fast updates. Vector API

```
scala> IndexedSeq(1, 2, 3)
res0: IndexedSeq[Int] = Vector(1, 2, 3)
```

Range Ordered sequence of Ints that are spaced apart. You will often see this used where a counting for-loop was used before. Range API

```
scala> for (i <- 1 to 3) { println(i) }
1
2
3
```

Ranges have the standard functional combinators available to them.

```
scala> (1 to 3).map { i => i }
res0: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3)
```

Defaults

Using apply methods on the traits will give you an instance of the default implementation, For instance, Iterable(1, 2) returns a List as its default implementation.

```
scala> Iterable(1, 2)

res0: Iterable[Int] = List(1, 2)
```

Same with Seq, as we saw earlier

```
scala> Seq(1, 2)
res3: Seq[Int] = List(1, 2)

scala> Iterable(1, 2)
res1: Iterable[Int] = List(1, 2)

scala> Sequence(1, 2)
warning: there were deprecation warnings; re-run with -deprecation for details
res2: Seq[Int] = List(1, 2)
```

Set

```
scala> Set(1, 2)
res31: scala.collection.immutable.Set[Int] = Set(1, 2)
```

Some descriptive traits

IndexedSeq fast random-access of elements and a fast length operation. API doc

LinearSeq fast access only to the first element via head, but also has a fast tail operation. API doc

Mutable vs. Immutable

immutable

Pros

- Can't change in multiple threads

Con

- Can't change at all

Scala allows us to be pragmatic, it encourages immutability but does not penalize us for needing mutability. This is very similar to var vs. val. We always start with val and move back to var when required.

We favor starting with the immutable versions of collections but switching to the mutable ones if performance dictates. Using immutable collections means you won't accidentally change things in multiple threads.

Mutable

All of the above classes we've discussed were immutable. Let's discuss the commonly used mutable collections.

HashMap defines `getOrElseUpdate`, `+=` HashMap API

```
scala> val numbers = collection.mutable.Map(1 -> 2)
numbers: scala.collection.mutable.Map[Int,Int] = Map((1,2))

scala> numbers.get(1)
res0: Option[Int] = Some(2)

scala> numbers.getOrElseUpdate(2, 3)
res54: Int = 3

scala> numbers
res55: scala.collection.mutable.Map[Int,Int] = Map((2,3), (1,2))

scala> numbers += (4 -> 1)
res56: numbers.type = Map((2,3), (4,1), (1,2))
```

ListBuffer and ArrayBuffer Defines `+=` ListBuffer API, ArrayBuffer API

LinkedList and DoubleLinkedList LinkedList API, DoubleLinkedList API

PriorityQueue API doc

Stack and ArrayStack Stack API, ArrayStack API

StringBuilder Interestingly, StringBuilder is a collection. API doc

Life with Java

You can easily move between Java and Scala collection types using conversions that are available in the JavaConverters package. It decorates commonly-used Java collections with `asScala` methods and Scala collections with `asJava` methods.

```
import scala.collection.JavaConverters._
val sl = new scala.collection.mutable.ListBuffer[Int]
val jl : java.util.List[Int] = sl.asJava
val sl2 : scala.collection.mutable.Buffer[Int] = jl.asScala
assert(sl eq sl2)
```

Two-way conversions:

```
scala.collection.Iterable <=> java.lang.Iterable
scala.collection.Iterable <=> java.util.Collection
scala.collection.Iterator <=> java.util.{ Iterator, Enumeration }
scala.collection.mutable.Buffer <=> java.util.List
scala.collection.mutable.Set <=> java.util.Set
scala.collection.mutable.Map <=> java.util.{ Map, Dictionary }
scala.collection.mutable.ConcurrentMap <=> java.util.concurrent.ConcurrentMap
```

In addition, the following one way conversions are provided:

```
scala.collection.Seq => java.util.List
scala.collection.mutable.Seq => java.util.List
scala.collection.Set => java.util.Set
scala.collection.Map => java.util.Map
```

Built at @twitter by @stevej, @marius, and @lahosken with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman, and @mccv; Russian translation by appigram; Chinese simple translation by jasonqu; Korean translation by enshahar;