Chapter 27
Immutable and Mutable Collection Packages

27.1 Introduction

This chapter discusses the contents of the Scala collection framework packages for immutable and mutable collections.

27.2 Package Scala.Collection.Immutable

The key classes and traits in the `scala.collection.immutable` package are shown in Fig. 27.1 and described in the rest of this section.

27.2.1 Sequences

There are a number of different types of Sequences including:

- **Vector**. This is a general-purpose immutable indexed sequence. As a general rule you might choose to use a `Vector` over a `List` as it provides faster access. An example of using the `Vector` class is given below:

```
object VectorTest extends App {
  val v1 = Vector(3, 2, 1)
  println(v1)
  println(v1(0))
  println(v1.length)
  val v2 = 4 +: v1
  println(v2)
}
```

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_27

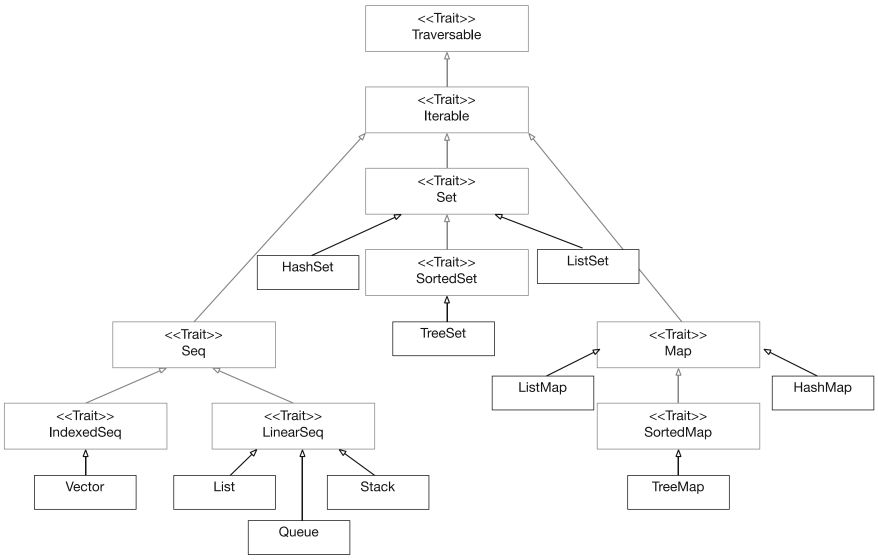

Fig. 27.1 Key classes and traits in the scala.collection.immutable package

The result of running this program is shown below:

```
Vector(3, 2, 1)
3
3
Vector(4, 3, 2, 1)
```

- **List**. The List class has been discussed extensively earlier in this chapter. However, it is worth noting that it is probably the most widely used type in the `scala.collection.immutable` package.
- **Queue**. A Queue is a first-in first-out (FIFO) type of collection. That is elements can be added to a queue and removed from the queue in the same order. The primary methods on a Queue are the `enqueue` method for adding an element to the `queue`, `dequeue` for removing an object from the queue. This type might at first seem a strange choice for the immutable a package. After all once you create an immutable Queue you cannot modify it. This is true, but there operations such as `enqueue` and `dequeue` that return a copy of the original queue with a new element added or an element removed, respectively. This is particularly useful in concurrent processing environments where this avoids the need to worry about multiple processes updating a common data structure such as a `Queue.` An example of using the `Queue` class is shown below:

```
import scala.collection.immutable.Queue

object QueueTest extends App {
  val q1 = Queue[Int]()
  val q2 = q1.enqueue(1)
  println(q2)
  val q3 = q2.enqueue(List(2, 3))
  println(q3)
  val (r, q4) = q3.dequeue
  println(r)
  println(q4)
}
```

The result of executing this program is:

```
Queue(1)
Queue(1, 2, 3)
1
Queue(2, 3)
```

Points to note about this example are that each of the operations to add or remove elements from the queue generated a new Queue instance. Also note that dequeue returned both the result of removing the first element from the queue and the newly generated queue instance containing all the elements of 'q3' minus the value removed. Finally, also note that we had to import the Queue type.

- **Stack**. This class was deprecated in 2.12. A Stack provides a last-in first-out behaviour (LIFO). As with the Queue any operations that add elements to the stack or remove elements from the stack actually result in a new Stack instance being created. A simple example is shown below:

```
import scala.collection.immutable.Stack

object StackTest extends App {
      val s1 = Stack[Int]()
      val s2 = s1.push(1)
      println(s2)
      val s3 = s2.push(2)
      println(s3)
      println(s3.top)
      val q4 = s3.pop
      println(q4)
}
```

The result of executing this program is given below:

```
Stack(1)
Stack(2, 1)
2
Stack(1)
```

Note that we have again imported the `scala.collection.immutable.Stack` class. Also note that the top method returns the value at the top of the stack but does not remove it. Pop on the other hand removes the top value and returns a copy of the original stack without the top element.
From Scala 2.12 onwards a list should be used with a var, along with the +: and head methods.

```
var stack = List.empty[Int]
stack = stack.+:(52)
stack = stack.+:(21)
val x = stack.head
```

27.2.2 Sets

There are a number of different types of Sets including:

- **HashSet** is an immutable `Set` implementation based on a hashing function. A Set only allows a single instance of an element in the Set—the `equals` method is used to determine equality. For example,

```
// Create an immutable Set
var teams =
     HashSet("Liverpool", "West Ham",
          "Newcastle", "Everton", "West Ham")
println(teams)
```

  This results in the following output:

```
HashSet(Liverpool, West Ham, Newcastle, Everton)
```

  Note that there is only one occurrence of the String “West Ham” in this set.
- **TreeSet**. This class implements the `Set` concept based on a tree structure. Specifically on a Red-Black Tree structure. Red-Black Trees are a form of balanced binary tree where some nodes are designed *red* and some nodes are designated *black*. With any balanced tree structure the operations applied to the tree complete in time logarithmic to the size of the tree.

- **ListSet**. A `ListSet` is an immutable set using a list-based structure internally. It can be viewed as a List that restricts the occurrences of some element to one within the list. For example,

```
 var t2 = ListSet("Liverpool", "West Ham",
 "Newcastle", "Everton", "West Ham")
   println(t2)
```

  which results in the following output:

```
ListSet(Everton, Newcastle, West Ham, Liverpool)
```

27.2.3 Maps

There are a number of different types of Maps including

- **ListMap**. A `ListMap` is a map that uses a linked list-based structure to internally represent the key-value pairs in the Map. Operations on a list map take linear time relative to the size of the map. Due to this there is little benefit in using a ListMap and a `HashMap` is almost always a better choice.
- **HashMap**. It represents a collection of associated keys and values that are organized based on the hash code of the key.
- **TreeMap**. This class implements the Map as a tree structure.

27.3 Package Scala.Collection.Mutable

The key types in the `scala.collection.mutable` package are shown in Fig. 27.2. You will notice that many of the names are the same as those in the immutable collection described earlier. This can make it very confusing for someone working with the collection classes to understand the impact of the operations being performed. For example, an operation on an immutable collection may generate a new instance of that collection containing the new element, whereas the mutable version will merely add that element to the collection (and both collections are called `HashMap`!). Therefore ensuring that you know which type of collection you are working with is an important issue.

The actual set of mutable collection classes available is continually growing; for example, in Scala 2.11 additional mutable types, LongMap and AnyRefMap were added to provide improved performance when using Long or AnyRef keys. In turn, Scala 2.12 added a mutable TreeMap collection class.

In this section we will look at the `ArrayBuffer`, `ListBuffer`, `Stack` and `HashMap` classes.

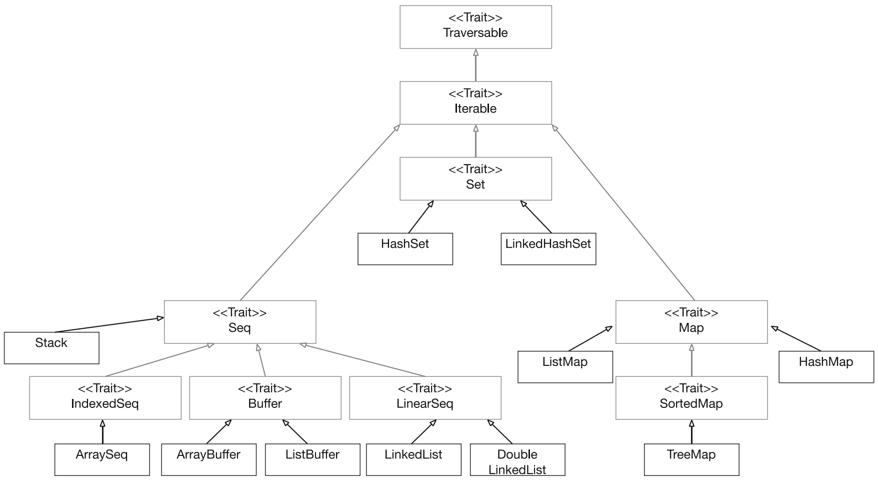

Fig. 27.2 Key types in the scala.collection.mutable package

27.3.1 ArrayBuffer

An `ArrayBuffer` is a growable collection that is backed by an array and has a current size (it is similar in nature to the ArrayList class in Java). The majority of operations on an array buffer have the same speed as for an array as the operations simply access and modify the underlying array, ArrayBuffers can therefore be used for efficiently building up a large collection whenever new items are added to the end of the array (e.g. as a result of reasoning data from a file or a database). The following example illustrates how they are used (remember elements in a collection such as `ArrayBuffer` are indexed from Zero not one):

```
import scala.collection.mutable.ArrayBuffer

object ArrayBufferTest extends App {
  val data = ArrayBuffer[Int]()
  data += 1
  println(data)
  data += 5
  data += 6
  println(data)
  println(data(2))
  val array = data.toArray
  println(array)
}
```

The result of executing this program is shown below:

```
ArrayBuffer(1)
ArrayBuffer(1, 5, 6)
6
[I@3d5b5376
```

As you can see from this example it is the `ArrayBuffer` data which is updated when we add an element using the '+=' operator. Thus we can see that data is a mutable collection as opposed to the immutable collections we looked at earlier. We have also included how an `ArrayBuffer` can be converted into an Array using the `toArray` (other options include `toList` and `toSet`).

27.3.2 ListBuffer

A `ListBuffer` is like an `ArrayBuffer` except that it uses a linked list as its underlying structure (rather than an array). This means that insertion may be faster than for an `ArrayBuffer`. Also if you intend to convert your structure to a List once you have added all the elements to the buffer, then a `ListBuffer` is a more efficient option. Here is an example of working with a `ListBuffer`:

```
import scala.collection.mutable.ListBuffer

object ListBufferTest extends App {
val data = ListBuffer[Int]()
  data += 1
  println(data)
  data += 5
  data += 6
  println(data)
  println(data(2))
  val array = data.toList
  println(array)
}
```

The output from this program is given below:

```
ListBuffer(1)
ListBuffer(1, 5, 6)
6
List(1, 5, 6)
```

27.3.3 LinkedList

Linked lists are mutable sequences that consist of nodes that are linked together via pointers. That is, the first node has a reference to the second node, the second node has a reference to the third, etc., with next pointers.

27.3.4 Stack

The mutable `Stack` provides similar functionality to the immutable Stack with the primary difference being that when an element is added to the stack or removed from the stack it affects that stack (rather than creating a copy of the stack modified as appropriate). The following listing illustrates typical mutable stack behaviour:

```
import scala.collection.mutable.Stack

object StackTest extends App {
   val stack = Stack[Int]()
       stack.push(1)
       stack.push(2)
       stack.push(3)
       println(stack)
       println("top: " + stack.top)
       println("pop: " + stack.pop)
       println("pop: " + stack.pop)
       println(stack)
    }
```

The result of running this program is:

```
Stack(3, 2, 1)
top: 3
pop: 3
pop: 2
Stack(1)
```

Note that the push operation updates the stack directly as the pop option. Note that for the mutable stack pop returns the item popped whereas for the immutable version it returned the state of the stack after the pop.

27.3.5 *HashMap*

The mutable `HashMap` class is very similar to the immutable `HashMap` except that modifications are made to the receiver of the operation rather than to a copy. The following listing illustrates `HashMap` usage:

```
import scala.collection.mutable.HashMap

object HashMapTest extends App {
   val map = HashMap[String,String]()
   map += ("UK" -> "London")
   map += ("FRANCE" -> "Paris")
   map += ("Spain" -> "Madrid")
   map += ("USA" -> "Wasington. DC")
   println(map)
   println(map.size)
   println(map.keys)
   println(map.values)
   println(map.isEmpty)
   println(map.get("UK"))
   println(map("UK"))
   println(map.contains("UK"))
   println(map.getOrElse("Ireland", "Not known"))
}
```

The result of executing this program is given below:

```
Map(FRANCE - > Paris, UK - > London, Spain - > Madrid, USA - > Wasington.
DC)
4
Set(FRANCE, UK, Spain, USA)
HashMap(Paris, London, Madrid, Wasington. DC)
false
Some(London)
London
true
Not known
```

27.4 Generic Collections

One interesting area is to look at what happens if you create one of the `scala.collection` core collections such as `Set` or `Map`. The following program uses the Map type from the `scala.collection`.

```
object Test3 extends App {
  var flights = Map[Int, String]()
  println(flights.getClass())
  flights += (121 -> "Miami")
  println(flights.getClass())
  flights += (231 -> "Dublin")
  println(flights.getClass())
  flights += (456 -> "Paris")
  println(flights.getClass())
  println(flights)
  println(flights(231))
}
```

Now consider the output from this program:

```
class scala.collection.immutable.Map$EmptyMap$
class scala.collection.immutable.Map$Map1
class scala.collection.immutable.Map$Map2
class scala.collection.immutable.Map$Map3
Map(121 - > Miami, 231 - > Dublin, 456 - > Paris)
Dublin
```

There is something strange happening here. The class name retrieved from the `getClass` method appears to be different!

This is because by default the core `Map` type uses the immutable `Map` from the `scala.collection.immutable` package. However, because our program then adds a series of key-value pairs to that map it must create new instances of the `Map` class to represent the new map and bind those to the variable flights. It does this as flights actually reference a wrapper around an immutable Map and it is this inner map that is replaced.

If we add an import to this program to import the `scala.collection.-mutable.Map`, for example:

```
import scala.collection.mutable.Map
```

we can now use the same immutable HashMap throughout the application:

```
class scala.collection.mutable.HashMap
class scala.collection.mutable.HashMap
class scala.collection.mutable.HashMap
class scala.collection.mutable.HashMap
Map(456 - > Paris, 121 - > Miami, 231 - > Dublin)
Dublin
```

27.5 Summary

It is likely that, just as in C# and Java, these classes will become the most used classes in Scala. The various collection API interfaces and classes will form the basis of the data structures you build and will be the cornerstone of most of your implementations. Stick with them, try them out, implement some simple programs using them, and you will soon find that they are easy to use and extremely useful. You will very quickly come to wonder why every language doesn't have the same facilities!