Chapter 22

Implementing Lists

Lists have been ubiquitous in this book. Class List is probably the most commonly used structured data type in Scala. Chapter 16 showed you how to use lists. This chapter "opens up the covers" and explains a bit about how lists are implemented in Scala.

Knowing the internal workings of the List class is useful for several reasons. You gain a better idea of the relative efficiency of list operations, which will help you in writing fast and compact code using lists. You also gain a toolbox of techniques that you can apply in the design of your own libraries. Finally, the List class is a sophisticated application of Scala's type system in general and its genericity concepts in particular. So studying class List will deepen your knowledge in these areas.

22.1 THE LIST CLASS IN PRINCIPLE

Lists are not "built-in" as a language construct in Scala; they are defined by an abstract classList in the scala package, which comes with two subclasses for :: and Nil. In this chapter we will present a quick tour through class List. This section presents a somewhat simplified account of the class, compared to its real implementation in the Scala standard library, which is covered in Section 22.3.

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

List is an abstract class, so you cannot define elements by calling the empty List constructor. For instance the expression "new List" would be illegal. The class has a type parameter T. The +in front of this type parameter specifies that lists are covariant, as discussed in Chapter 19.

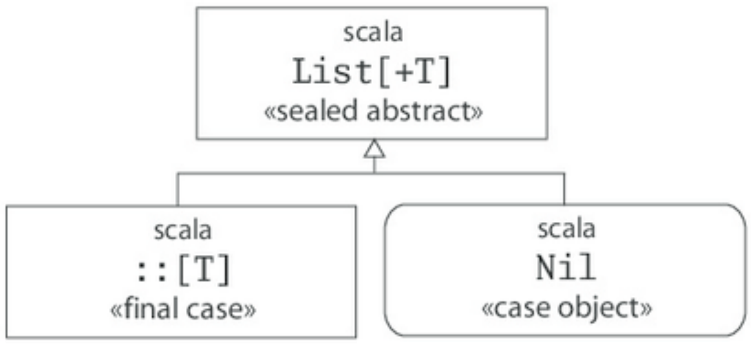

Figure 22.1 - Class hierarchy for Scala lists.

Because of this property, you can assign a value of type List[Int] to a variable of typeList[Any]:

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

scala> var ys: List[Any] = xs
ys: List[Any] = List(1, 2, 3)
```

All list operations can be defined in terms of three basic methods:

```
def isEmpty: Boolean
def head: T
def tail: List[T]
```

These three methods are all abstract in class List. They are defined in the subobject Nil and the subclass ::. The hierarchy for List is shown in Figure 22.1.

The Nil object

The Nil object defines an empty list. Its definition is shown in Listing 22.1. The Nil object inherits from type List[Nothing]. Because of covariance, this means that Nil is compatible with every instance of the List type.

```
  case object Nil extends List[Nothing] {
    override def isEmpty = true
    def head: Nothing =
      throw new NoSuchElementException("head of empty list")
    def tail: List[Nothing] =
      throw new NoSuchElementException("tail of empty list")
  }
```

Listing 22.1 - The definition of the Nil singleton object.

The three abstract methods of class List are implemented in the Nil object in a straightforward way: The isEmpty method returns true, and the head and tail methods both throw an exception. Note that throwing an exception is not only reasonable, but practically the only possible thing to do for head: Because Nil is a List of Nothing, the result type of headmust be Nothing. Since there is no value of this type, this means that head cannot return a normal value. It has to return abnormally by throwing an exception.[1]

The :: class

Class ::, pronounced "cons" for "construct," represents non-empty lists. It's named that way in order to support pattern matching with the infix ::. You have seen in Section 16.5 that every infix operation in a pattern is treated as a constructor application of the infix operator to its arguments. So the pattern x :: xs is treated as ::(x, xs) where :: is a case class.

Here is the definition of the :: class:

```
final case class ::[T](hd: T, tl: List[T]) extends List[T] {
  def head = hd
  def tail = tl
  override def isEmpty: Boolean = false
}
```

The implementation of the :: class is straightforward. It takes two parameters hd and tl, representing the head and the tail of the list to be constructed. The definitions of the head andtail method simply return the corresponding parameter. In fact, this pattern can be abbreviated by letting the parameters directly implement the head and tail methods of the superclass List, as in the following equivalent but shorter definition of the :: class:

```
final case class ::[T](head: T, tail: List[T])
    extends List[T] {

  override def isEmpty: Boolean = false
}
```

This works because every case class parameter is implicitly also a field of the class (it's like the parameter declaration was prefixed with val). Recall from Section 20.3 that Scala allows you to implement an abstract parameterless method such as head or tail with a field. So the code above directly uses the parameters head and tail as implementations of the abstract methods head and tail that were inherited from class List.

Some more methods

All other List methods can be written using the basic three. For instance:

```
def length: Int =
  if (isEmpty) 0 else 1 + tail.length
```

or:

```
def drop(n: Int): List[T] =
  if (isEmpty) Nil
  else if (n <= 0) this
  else tail.drop(n - 1)
```

or:

```
def map[U](f: T => U): List[U] =
  if (isEmpty) Nil
  else f(head) :: tail.map(f)
```

List construction

The list construction methods :: and ::: are special. Because they end in a colon, they are bound to their right operand. That is, an operation such as x :: xs is treated as the method callxs.::(x), not x.::(xs). In fact, x.::(xs) would not make sense, as x is of the list element type, which can be arbitrary, so we cannot assume that this type would have a :: method.

For this reason, the :: method should take an element value and yield a new list. What is the required type of the element value? You might be tempted to say it should be the same as the list's element type, but in fact this is more restrictive than necessary.

To see why, consider this class hierarchy:

```
abstract class Fruit
class Apple extends Fruit
class Orange extends Fruit
```

Listing 22.2 shows what happens when you construct lists of fruit:

```
scala> val apples = new Apple :: Nil
apples: List[Apple] = List(Apple@e885c6a)

scala> val fruits = new Orange :: apples
fruits: List[Fruit] = List(Orange@3f51b349, Apple@e885c6a)
```

Listing 22.2 - Prepending a supertype element to a subtype list.

The apples value is treated as a List of Apples, as expected. However, the definition of fruitsshows that it's still possible to add an element of a different type to that list. The element type of the resulting list is Fruit, which is the most precise common supertype of the original list element type (*i.e.*, Apple) and the type of the element to be added (*i.e.*, Orange). This flexibility is obtained by defining the :: method (cons) as shown in Listing 22.3:

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

Listing 22.3 - The definition of method :: (cons) in class List.

Note that the method is itself polymorphic—it takes a type parameter named U. Furthermore,U is constrained in [U >: T] to be a supertype of the list element type T. The element to be added is required to be of type U and the result is a List[U].

With the formulation of :: shown in Listing 22.3, you can check how the definition of fruitsshown in Listing 22.2 works out type-wise: In that definition, the type parameter U of :: is instantiated to Fruit. The lower-bound constraint of U is satisfied because the list apples has type List[Apple] and Fruit is a supertype of Apple. The argument to the :: is new Orange, which conforms to type Fruit. Therefore, the method application is type-correct with result typeList[Fruit]. Figure 22.2 illustrates the structure of the lists that result from executing the code shown in Listing 22.2.

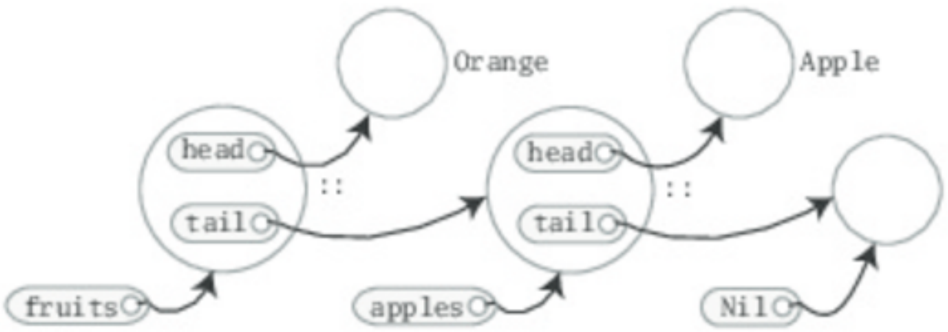

Figure 22.2 - The structure of the Scala lists shown in Listing 22.2.

In fact, the polymorphic definition of :: with the lower bound T is not only convenient, it is also necessary to render the definition of class List type-correct. This is because Lists are defined to be covariant.

Assume for a moment that we had defined :: like this:

```
// A thought experiment (which wouldn't work)
def ::(x: T): List[T] = new scala.::(x, this)
```

You saw in Chapter 19 that method parameters count as contravariant positions, so the list element type T is in contravariant position in the definition above. But then List cannot be declared covariant in T. The lower bound [U >: T] thus kills two birds with one stone: It removes a typing problem and leads to a :: method that's more flexible to use. The list concatenation method ::: is defined in a similar way to ::, as shown in Listing 22.4.

```
  def :::[U >: T](prefix: List[U]): List[U] =
    if (prefix.isEmpty) this
    else prefix.head :: prefix.tail ::: this
```

Listing 22.4 - The definition of method ::: in class List.

Like cons, concatenation is polymorphic. The result type is "widened" as necessary to include the types of all list elements. Note again that the order of the arguments is swapped between an infix operation and an explicit method call. Because both ::: and :: end in a colon, they both bind to the right and are both right associative. For instance, the else part of the definition of ::: shown in Listing 22.4 contains infix operations of both :: and :::.

These infix operations can be expanded to equivalent method calls as follows:

```
prefix.head :: prefix.tail ::: this
  equals (because :: and ::: are right-associative)

prefix.head :: (prefix.tail ::: this)
  equals (because :: binds to the right)

(prefix.tail ::: this).::(prefix.head)
  equals (because ::: binds to the right)

this.:::(prefix.tail).::(prefix.head)
```

22.2 THE LISTBUFFER CLASS

The typical access pattern for a list is recursive. For instance, to increment every element of a list without using map you could write:

```
def incAll(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => x + 1 :: incAll(xs1)
}
```

One shortcoming of this program pattern is that it is not tail recursive. Note that the recursive call to incAll above occurs inside a :: operation. Therefore each recursive call requires a new stack frame.

On today's virtual machines this means that you cannot apply incAll to lists of much more than about 30,000 to 50,000 elements. This is a pity. How do you write a version of incAllthat can work on lists of arbitrary size (as much as heap-capacity allows)?

One approach is to use a loop:

```
for (x <- xs) // ??
```

But what should go in the loop body? Note that where incAll constructs the list by prepending elements to the result of the recursive call, the loop needs to append new elements at the end of the result list. A very inefficient possibility is to use :::, the list append operator:

```
var result = List[Int]()    // a very inefficient approach
for (x <- xs) result = result ::: List(x + 1)
result
```

This is terribly inefficient. Because ::: takes time proportional to the length of its first operand, the whole operation takes time proportional to the square of the length of the list. This is clearly unacceptable.

A better alternative is to use a list buffer. List buffers let you accumulate the elements of a list. To do this, you use an operation such as "buf += elem", which appends the element elem at the end of the list buffer buf. Once you are done appending elements, you can turn the buffer into a list using the toList operation.

ListBuffer is a class in package scala.collection.mutable. To use the simple name only, you can import ListBuffer from its package:

```
import scala.collection.mutable.ListBuffer
```

Using a list buffer, the body of incAll can now be written as follows:

```
val buf = new ListBuffer[Int]
for (x <- xs) buf += x + 1
buf.toList
```

This is a very efficient way to build lists. In fact, the list buffer implementation is organized so that both the append operation (+=) and the toList operation take (very short) constant time.

22.3 THE LIST CLASS IN PRACTICE

The implementations of list methods given in Section 22.1 are concise and clear, but suffer from the same stack overflow problem as the non-tail recursive implementation of incAll. Therefore, most methods in the real implementation of class List avoid recursion and use loops with list buffers instead. For example, Listing 22.5 shows the real implementation of mapin class List:

```
  final override def map[U](f: T => U): List[U] = {
    val b = new ListBuffer[U]
    var these = this
    while (!these.isEmpty) {
      b += f(these.head)
      these = these.tail
    }
    b.toList
  }
```

Listing 22.5 - The definition of method map in class List.

This revised implementation traverses the list with a simple loop, which is highly efficient. A tail recursive implementation would be similarly efficient, but a general recursive implementation would be slower and less scalable. But what about the operation b.toList at the end? What is its complexity? In fact, the call to the toList method takes only a small number of cycles, which is independent of the length of the list.

To understand why, take a second look at class ::, which constructs non-empty lists. In practice, this class does not quite correspond to its idealized definition given previously inSection 22.1. The real definition is shown in Listing 22.6. As you can see, there's one peculiarity: the tl argument is a var! This means that it is possible to modify the tail of a list after the list is constructed. However, because the variable tl has the modifier private[scala], it can be accessed only from within package scala. Client code outside this package can neither read nor write tl.

```
    final case class ::[U](hd: U,
        private[scala] var tl: List[U]) extends List[U] {

      def head = hd
      def tail = tl
      override def isEmpty: Boolean = false
    }
```

Listing 22.6 - The definition of the :: subclass of List.

Since the ListBuffer class is contained in a subpackage of package scala,scala.collection.mutable, ListBuffer can access the tl field of a cons cell. In fact the elements of a list buffer are represented as a list and appending new elements involves a modification of the tl field of the last :: cell in that list. Here's the start of class ListBuffer:

```
  package scala.collection.immutable
  final class ListBuffer[T] extends Buffer[T] {
    private var start: List[T] = Nil
    private var last0: ::[T] = _
    private var exported: Boolean = false
    ...
```

You see three private fields that characterize a ListBuffer:

start	points to the list of all elements stored in the buffer
last0	points to the last :: cell in that list
exported	indicates whether the buffer has been turned into a list using a toList operation

The toList operation is very simple:

```
  override def toList: List[T] = {
    exported = !start.isEmpty
    start
  }
```

It returns the list of elements referred to by start and also sets exported to true if that list is nonempty. So toList is very efficient because it does not copy the list which is stored in aListBuffer. But what happens if the list is further extended after the toList operation? Of course, once a list is returned from toList, it must be immutable. However, appending to thelast0 element will modify the list which is referred to by start.

To maintain the correctness of the list buffer operations, you need to work on a fresh list instead. This is achieved by the first line in the implementation of the += operation:

```
override def += (x: T) = {
  if (exported) copy()
  if (start.isEmpty) {
    last0 = new scala.::(x, Nil)
    start = last0
  } else {
    val last1 = last0
    last0 = new scala.::(x, Nil)
    last1.tl = last0
  }
}
```

You see that += copies the list pointed to by start if exported is true. So, in the end, there is no free lunch. If you want to go from lists which can be extended at the end to immutable lists, there needs to be some copying. However, the implementation of ListBuffer is such that copying is necessary only for list buffers that are further extended after they have been turned into lists. This case is quite rare in practice. Most use cases of list buffers add elements incrementally and then do one toList operation at the end. In such cases, no copying is necessary.

22.4 FUNCTIONAL ON THE OUTSIDE

In the previous section, we showed key elements of the implementation of Scala's List andListBuffer classes. You saw that lists are purely functional on the "outside" but have an imperative implementation using list buffers on the "inside." This is a typical strategy in Scala programming—trying to combine purity with efficiency by carefully delimiting the effects of impure operations.

But you might ask, Why insist on purity? Why not just open up the definition of lists, making the tail field, and maybe also the head field, mutable? The disadvantage of such an approach is that it would make programs much more fragile. Note that constructing lists with :: re-uses the tail of the constructed list.

So when you write:

```
val ys = 1 :: xs
val zs = 2 :: xs
```

the tails of lists ys and zs are shared; they point to the same data structure. This is essential for efficiency; if the list xs was copied every time you added a new element onto it, this would be much slower. Because sharing is pervasive, changing list elements, if it were possible, would be quite dangerous. For instance, taking the code above, if you wanted to truncate listys to its first two elements by writing:

```
  ys.drop(2).tail = Nil  // can't do this in Scala!
```

you would also truncate lists zs and xs as a side effect.

Clearly, it would be quite difficult to keep track of what gets changed. That's why Scala opts for pervasive sharing and no mutation for lists. The ListBuffer class still allows you to build up lists imperatively and incrementally, if you wish. But since list buffers are not lists, the types keep mutable buffers and immutable lists separate.

The design of Scala's List and ListBuffer is quite similar to what's done in Java's pair of classesString and StringBuffer. This is no coincidence. In both situations the designers wanted to maintain a pure immutable data structure but also provide an efficient way to construct this structure incrementally. For Java and Scala strings, StringBuffers (or, in Java 5, StringBuilders) provide a way to construct a string incrementally. For Scala's lists, you have a choice: You can either construct lists incrementally by adding elements to the beginning of a list using ::, or you use a list buffer for adding elements to the end. Which one is preferable depends on the situation. Usually, :: lends itself well to recursive algorithms in the divide-and-conquer style. List buffers are often used in a more traditional loop-based style.

22.5 CONCLUSION

In this chapter, you saw how lists are implemented in Scala. List is one of the most heavily used data structures in Scala, and it has a refined implementation. List's two subclasses, Niland ::, are both case classes. Instead of recursing through this structure, however, many core list methods are implemented using a ListBuffer. ListBuffer, in turn, is carefully implemented so that it can efficiently build lists without allocating extraneous memory. It is functional on the outside, but uses mutation internally to speed up the common case where a buffer is discarded after toList has been called. After studying all of this, you now know the list classes inside and out, and you might have learned an implementation trick or two.

Footnotes for Chapter 22:

[1] To be precise, the types would also permit for head to always go into an infinite loop instead of throwing an exception, but this is clearly not what's wanted.