集合 collection（二）

一、视图

1. 集合有相当多的方法来构造新的集合，如 `map/filter/++` 等。我们称这些方法为变换器 `transformer`，因为它们至少接受一个集合作为参数，并产生另一个集合作为结果。

   变换器有两种主要的实现方式：

   - 严格的 `transformer`：构造出带有所有元素的新集合。
   - 非严格的 `transformer`（或者惰性的）：只是构造出结果集合的一个代理，结果集合的元素会根据需要来构造。

   考虑如下的一个惰性 `map`：

   ```
   def lazyMap[T, U](iter : Iterable[T],  f : T => U) = new Iterable[U]{
     def iterator = iter.iterator map f
   }
   ```

   该方法将给定的集合 `iter` 通过 `f` 函数映射到一个新集合。

   `lazyMap` 在构造新的 `Iterable` 时并没有遍历给定集合 `iter` 的所有元素，而是返回一个 `iterator`。只有在需要 `iterator` 元素的时候才去迭代。

2. 有一种系统化的方式可以将每个集合转换成惰性的版本，或者将惰性版本转换成非惰性版本。这个方式的基础是视图 `view`。

   视图是一种特殊的集合，它代表了某个基础集合，但是采用惰性的方式实现了所有的变换器。

3. 要想从集合得到它的视图，可以对集合采用 `view` 方法。如果 `xs` 是一个集合，则 `xs.view` 就是同一个集合、但是所有变换器都是按惰性的方式来实现的。

   ```
   val v = Vector(1 to 10 :  _*)
   v.map(_ +1).map( _ * 2)
   ```

   这里有两个连续的 `map` 操作，第一个 `map` 会构造出一个新的向量，然后第二个 `map` 会构造出第三个向量。因此，第一个 `map` 构造出来的中间向量有些浪费。

   可以通过将两个函数 `_ + 1` 和 `_ * 2` 整成一个函数，从而只需要一个 `map` 就可以。但是通常情况下，对一个集合的连续变化发生在不同的函数或者模块中，将这些变换整合在一起会破坏模块化设计。

   此时，使用视图就是最好的方法：

   ```
   v.view.map( _ + 1).map(_ * 2).force
   ```

   首先通过集合的 `.view` 方法将集合转换为视图，然后进行一系列的惰性变换，最后通过视图的 `.force` 方法将视图转换回集合。

   进一步拆解：

- `v.view`：该调用得到一个 `SeqView` 对象，即一个惰性求值的 `Seq`。该类型有两个类型参数：第一个类型参数 `Int` 给出了该视图的元素类型；第二个类型参数 `Vector[Int]` 给出了当你 `force` 该视图时将取回的类型构造器。

```
val v = Vector(1 to 10 : _*)
val vv = v.view // SeqView[Int, Vector[Int]]
```

- `vv.map( _ + 1)` 返回一个 `SeqViewM(...)` 的值。这本质上是一个记录了一个带有函数 `( _ + 1)` 的 `map` 操作需要被应用到向量 `v` 的包装器。但是，它并不会在 `force` 操作之前应用这个 `map` 操作。

  `SeqViewM` 后面的 `M` 表示该视图封装了一个 `map` 操作，还有其它字母表示其它惰性操作。如 `S` 表示惰性的 `slice` 操作，`R` 表示惰性的 `reverse` 操作。

- `vv.map( _ + 1).map( _ * 2)` 返回一个 `SeqViewMM(...)` 的值。这里 `MM` 表示封装了两个 `map` 操作。

- 最后的 `force` 操作，两个惰性的 `map` 操作会被执行，同时构造了新的 `Vector`。通过这种方式，我们避免了创建中间数据结构。

  注意一个细节：最终结果的静态类型是 `Seq`，而不是 `Vector`。这是因为第一次惰性 `map` 应用时，结果的静态类型为 `SeqViewM[Int, Seq[_]]`。也就是类型系统丢失了视图背后的 `Vector` 类型的信息。对任何特定类的视图的实现都需要大量的代码开发，因此 `Scala` 集合类库几乎只对一般化的集合类型而不是具体的实现提供了视图支持（`Array` 是个例外，对 `Array` 操作的惰性操作会得到 `Array` 本身）。

```
val v = Vector(1 to 10 : _*)
val vv = v.view          // 结果类型 SeqView[Int, Vector[Int]]
val m1 = vv.map( _ + 1) // 结果类型 SeqViewM[Int, Seq[_]]
val m2 = m1.map( _ * 2) // 结果类型 SeqViewMM[Int, Seq[_]]
val v2 = m2.force        // 结果类型 Seq[Int]
```

4. 采用视图有两个原因：

   - 首先是性能。通过将集合切换成视图，可以避免中间结果的产生。这些节约下来的开销可能非常重要。
   - 其次是针对可变序列。这类视图的很多变换器提供了对原始序列的一个窗口，可以用于有选择的对序列中的某些原始进行更新。

     如：

```
val arr = (0 to 9).toArray
```

     可以通过创建该数组的一个切片的视图来创建到该数组的子窗口，它返回一个叫做 `IndexedSeqViewS(...)` 的对象：

```
val subArr = arr.view.slice(3, 6) // 返回一个 IndexedSeqView[Int,
Array[Int]] 类型对象
```

     现在视图 `subArr` 指向 `arr` 数组位置中 `3` 到 `5` 的元素。该视图并不会复制这些元素，它只是提供了对它们的引用。注意，这是一个 `mutable.IndexedSeqView`。

     最后我们可以原地修改序列中的某些元素：

```
def negate(xs: scala.collection.mutable.Seq[Int]) = {
  for (i <- 0 until xs.length) xs(i) = - xs(i)
}
negate(subArr)
arr// 现在为  Array(0, 1, 2, -3, -4, -5, 6, 7, 8, 9)
```

5. 既然视图具有如此多的优点，为什么还需要严格求值的集合？有两个原因：
   - 惰性求值的视图并不是始终性能最优。对于小型集合而言，组织视图和应用闭包的额外开销通常会大于省略中间数据结构的收益。
   - 惰性求值可能带来副作用。比如：

     ```
     val arr = (0 to 9).toArray
     val v = arr.view
     .... //经过了很多步之后
     v.map( x => do_with(x))
     ```

     我们发现 `do_with(x)` 并没有真正的执行，这是因为 `map` 的结果是视图，元素并未真正创建。如果我们对 `map` 的结果调用 `.force` ， `do_with` 就能执行。也就是 `do_with` 执行的时刻由 `force` 来决定，这看起来比较奇怪。
6. `Scala` 类库从 `2.8` 开始规定：除了流之外的所有集合都是严格求值的；从严格求值的集合到惰性求值的集合的唯一方式是通过 `view` 方法；从惰性求值集合到严格求值集合的唯一方式是通过 `force` 方法。
7. 为了避免被延迟求值的各种细节困扰，应该将视图的使用局限在两种场景：
   - 要么在集合变换没有副作用的纯函数式的代码中应用视图。
   - 要么对所有修改都应用显式执行的可变集合应用视图。

   最好避免在既创建新集合、又有副作用的场景混合视图和各种集合操作。

二、迭代器

1. 迭代器并不是集合，而是逐个访问集合元素的一种方式。迭代器 `it` 的两个基本操作是 `next` 和 `hasNext`。
   - `it.next` 方法会返回迭代器的下一个元素，并将迭代器的状态往前推进一步。如果没有更多元素可以返回，则 `it.next` 会抛出 `NoSuchElementException` 异常。
   - `it.hasNext` 方法可以获知是否还有更多的元素可以返回。
2. "遍历"迭代器所有元素的最直接方式是通过 `while` 循环：

   ```
   while(it.hasNext)
     println(it.next())
   ```

3. `scala` 中的迭代器提供了 `Traversable, Iterable, Seq` 等特质中的大部分方法。如 `it.foreach` 方法可以用来对迭代器返回的每个元素执行给定的操作。采用 `foreach` 之后，`while` 循环等价于：

   ```
   it.foreach(println)
   ```

- 也可以用 `for` 表达式来替代涉及到 `foreach, map, filter, flatMap` 的表达式。因此上述方式等价于：

```
for (elem <- it) println(elem)
```

- 迭代器的 `foreach` 方法和 `Traversable` 的 `foreach` 方法的一个重要区别：
  - 对迭代器调用 `foreach` 之后，它执行完之后会将迭代器留在末尾。因此，对相同的迭代器再次调用 `next` 会抛出 `NoSuchElementException` 异常。
  - 对 `Traversable` 调用 `foreach` 之后，会保持集合中的元素数量不变。再次调用 `foreach` 可以得到上一轮 `foreach` 相同的结果。

  除了 `foreach` 操作之外，其它的操作比如 `map,filter` 等等也类似：在执行之后会将迭代器留在末尾。

  如：

```
val it = Iterator("a", "bc", "def")
val it2 = it.map(_.length) // 返回一个新的 Iterator[Int] 对象
it.next() // 抛出  NoSuchElementException 异常
```

- 另一个例子是 `dropWhile` 方法，但是注意：`it` 在 `dropWhile` 调用中被修改了。

```
val it = Iterator("a", "bc", "def")
val it2 = it.dropWhile(_.length <= 2) // 返回一个新的 Iterator[String] 对象
it.next() // 返回 "def"
```

  经过 `dropWhile` 调用之后， `it` 指向的是序列中的第三个单词 `"def"` 。注意，此时 `it` 和 `it2` 都返回相同的序列元素。

- 只有一个标准操作 `duplicate` 允许重用同一个迭代器：

```
val (it1, it2) = it.duplicate
```

  对 `duplicate` 的调用会返回两个迭代器，其中每个迭代器都返回和 `it` 完全相同的元素。而且这两个迭代器相互独立：推进其中一个并不会影响另外一个。

  而原始的 `it` 迭代器在 `duplicate` 调用之后就被推进到末端，不再可用。

4. 总体而言，如果你在调用迭代器的方法之后再也不访问它，则迭代器的行为跟集合很像。`Scala` 集合类库将这个性质显式表示为一个名为 `TraversableOnce` 的抽象，这是`Traversable` 和 `Iterator` 的公共超类特质。

   正如其名称所示， `TraversableOnce` 对象可以用 `foreach` 来遍历，但是在遍历之后该对象的状态并无规定。

   - 如果 `TraversableOnce` 对象实际上是一个 `Iterator`，则遍历之后它将位于末端。
   - 如果 `TraversableOnce` 对象实际上是一个`Traversable`，则遍历之后它将保持原样。

   `TraversableOnce` 的一个常见用法是作为既可以接收迭代器、又可以接收可遍历集合的方法的入参类型声明。比如， `Traversable` 特质的 `++` 方法，它接收一个 `TraversableOnce` 参数，从而可以追加来自迭代器或可遍历集合的元素。

5. 有时候希望有一个可以“向前看”的迭代器，这样就可以检查下一个要返回的元素，但是并不向前推进。

   考虑这样的场景：从一个返回字符串的迭代器中迭代，并且跳过开头的空字符串。一种做法是：

```
def func(it: Iterator[String])={
  while (! it.netx().isEmpty){
     println(it.next())
  }
}
```

   问题在于：

   - 如果 `it.next()` 返回的是空字符串，则该循环会跳过空字符串并且 `it` 指向下一个字符串。
   - 如果 `it.next()` 返回的是非空字符串，那么循环结束并且 `it` 指向下一个字符串，在这个过程中这个非空字符串完全得不到处理。

   这个问题的解决方案是采用带缓冲的迭代器，即 `BufferedIterator` 特质的实例。`BufferedIterator` 是 `Iterator` 的子特质，它提供了一个额外的方法 `head` 。对一个带缓冲的迭代器调用 `head` 方法会返回它的当前元素，但是并不会推进到下一个元素。

```
def func(it: BufferedIterator[String])={
  while (! it.head.isEmpty){
    println(it.next())
  }
}
```

6. 每个迭代器都可以被转换为带缓冲的迭代器，方法是调用迭代器的 `buffered` 方法。

```
val it = Iterator(1,2,3,4)
val bit = it.buffered
println(bit.head)   // 打印 1 。只是查看，并不推进
println(bit.next()) // 打印 1 。向前推进
println(bit.next()) // 打印 2 。向前推进
```

7. `Iterator` 特质包含的操作：

   - 抽象方法：
     - `it.next()`：返回迭代器中的下一个元素，并将 `it` 推进到下一步。
     - `it.hasNext` ：如果 `it` 能返回下一个元素，则返回 `true`。
   - 创建另一种迭代器：
     - `it.buffered`：返回一个包含 `it` 所有元素的带缓冲的迭代器。
     - `it grouped size`：返回一个以固定大小的“分段”来交出 `it` 元素的迭代器。
     - `it sliding size`：以固定大小的滑动窗口交出 `it` 元素的迭代器。
   - 拷贝：
     - `it copyToBuffer buf`：将 `it` 返回的所有元素拷贝到缓冲 `buf` 。
     - `it copyToArray(arr,s,k)`：将 `it` 返回的最多 `k` 个元素拷贝到数组 `arr`，从数组的下标 `s` 开始。后两个入参皆为可选。
   - 复制：
     - `it.duplicate`：返回一对新的迭代器，它们都相互独立地返回 `it` 所有的元素。

- 添加：
  - `it1 ++ it2`：返回 `it1` 所有元素，以及 `it2` 所有元素的迭代器。
  - `it padTo (len, x)`：返回一个迭代器，新的迭代器返回 `it` 所有元素，以及 `x` 的拷贝直到返回元素的总长度达到 `len` 。
- 映射：
  - `it map f`：通过对 `it` 返回的每个元素应用函数 `f` 得到的迭代器。
  - `it flatMap f`：通过对 `it` 返回的每个元素应用函数 `f` ，其中 `f` 返回的结果是迭代器，将 `f` 返回迭代器结果通过 `++` 拼接得到的迭代器。
  - `it collect f`：通过对 `it` 返回的每个元素应用函数 `f`，并将有定义的结果收集起来得到的迭代器。注意这里的 `f` 必须是一个偏函数 `PartialFunction` 。
- 转换：
  - `it.toArray/toList/toIterable/toSeq/toIndexSeq/toStream/toSet/toMap`：将 `it` 返回的元素收集到数组/列表/`Iterable` / `Seq` / `IndexSeq` / `Stream` / `Set` / `Map` 。
- 大小信息：注意：大小信息指的是从 `it` 当前位置开始的大小，而不从头开始。
  - `it.isEmpty`：测试迭代器是否为空（跟 `hasNext` 相反）。
  - `it.noEmpty`：测试迭代器是否非空（跟 `hasNext` 相同）。
  - `it.size`：返回 `it` 元素数量。注意：该操作之后，`it` 将位于末端。
  - `it.length`：等价于 `it.size` 。注意：该操作之后，`it` 将位于末端。
  - `it.hasDefinitSize`：如果已知将返回有限多的元素，则返回 `true` 。（默认同 `isEmpty`）。
- 元素获取：
  - `it find p`：以`Option`返回 `it` 中首个满足条件 `p` 的元素。如果没有满足条件的，则返回 `None` 。

    注意：迭代器会推进到刚好跳过首个满足`p` 的元素，如果没找到则推进到末端。
  - `it indexOf x`：返回 `it` 中首个等于 `x` 元素的下标。注意：迭代器会推进到刚好跳过首个 `x` 的位置。如果找不到，则返回 `-1`，并且`it` 推进到末端。
  - `it indexWhere p`：返回 `it` 中首个满足条件`p` 的元素的下标。注意：迭代器会推进到刚好跳过首个满足`p` 的元素。如果找不到，则返回 `-1`，并且 `it` 推进到末端。
- 子迭代器：
  - `it take n`：返回 `it` 的头 `n` 个元素的迭代器。注意：`it` 将推进到第 `n` 个元素最后的位置，如果少于 `n` 个元素则 `it` 将推进到末端。
  - `it drop n`：返回从 `it` 的第 `n+1` 个元素开始的迭代器。注意：`it` 将推进到与新迭代器相同的位置。
  - `it slice (m, n)`：返回从 `it` 的第 `m` 个元素开始到第 `n` 个元素之前为止（不包含）的元素的迭代器。
  - `it takeWhile p` ： 返回`it` 中连续位置、直到满足条件 `p` 的元素的迭代器。
  - `it dropWhile p` ：返回跳过 `it` 中连续位置、直到满足条件 `p` 的元素的迭代器。
  - `it filter p`：返回 `it` 中所有满足条件 `p` 的元素的迭代器。
  - `it withFilter p`：同 `it filter p`。这是为了支持 `for` 表达式语法。
  - `it filterNot p`：返回 `it` 中所有不满足条件 `p` 的元素的迭代器。
- 划分：
  - `it partition p`：将 `it` 划分为两个迭代器，其中一个返回 `it` 中所有满足条件 `p` 的元素；另一个返回 `it` 中所有不满足条件 `p` 的元素。

- 元素条件判断：
  - `it forall p`：返回是否 `it` 中所有元素满足条件 `p` 的布尔值。
  - `it exists p`：返回是否 `it` 中存在元素满足条件 `p` 的布尔值。
  - `it count p`：返回 `it` 中满足条件 `p` 的元素数量。
- 折叠：
  - `(z /: it)(op)`：以 `z` 开始从左向右依次对 `it` 中的连续元素应用二元操作 `op` 。
  - `(it :\ z)(op)`：以 `z` 开始从右向左依次对 `it` 中的连续元素应用二元操作 `op`。
  - `it.foldLeft(z)(op)`：同 `(z /: it)(op)`。
  - `it.foldRight(z)(op)`：同 `(it :\z)(op)`。
  - `it reduceLeft op`：从左向右依次对非空迭代器 `it` 的连续元素应用二元操作 `op`。
  - `it reduceRight op`：从右向左依次对非空迭代器 `it` 的连续元素应用二元操作 `op`。
- 特殊折叠：
  - `it.sum`：迭代器 `it` 中所有元素值的和。
  - `it.product`：迭代器 `it` 中所有元素值的乘积。
  - `it.min`：迭代器 `it` 中所有元素值的最小值。
  - `it.max`：迭代器 `it` 中所有元素值的最大值。
- `zip`：
  - `it1 zip it2`：返回由 `it1` 和 `it2` 对应元素的对偶组成的新迭代器。
  - `it1 zipAll (it2, x, y)`：返回由 `it1` 和 `it2` 对应元素的对偶组成的新迭代器，其中较短的序列用 `x` 或 `y` 的元素延展成相同的长度。
  - `it zipWithIndex`：返回由`it` 中的元素及其下标的对偶组成的新迭代器。
- 更新：
  - `it1 patch (i, it2, r)`：将 `it1` 中从位置 `i` 开始的 `r` 个元素替换成 `it2` 的元素得到的新迭代器。
- 比较：
  - `it1 sameElement it2`：测试是否 `it1` 和 `it2` 包含相同顺序的相同元素。
- 字符串：
  - `it addString (b, start, sep, end)`：将一个显示了 `it` 所有元素的字符串添加到 `StringBuilder b` 中，元素以 `sep` 分隔并仅考虑 `start` 和 `end` 之间的元素。
  - `it mkString ( start, seq, end)`：将迭代器转换为一个显示了 `it` 所有元素的字符串， 元素以 `sep` 分隔，并仅考虑 `start` 和 `end` 之间的元素。

三、集合相等性

1. 集合类库对于相等性和哈希的处理方式是一致的。
   - 首先将集合分为 `Set`、`Map`、`Seq` 等不同类别，不同类别下的集合永远不相等。如 `Set(1,2,3)` 和 `List(1,2,3)` 尽管元素完全相同，但是他们不等。
   - 在相同类别下，当且仅当集合拥有相同的元素时，它们才相等。进一步地，对于 `Seq`，不仅要求包含的元素相同，这些元素的顺序也要相同。

     如：

```
Vector(1,2,3) == List(1,2,3)  // true
Vector(1,2,3) == List(3,2,1)  // false
Vector(1,2,3) == Set(1,2,3)   // false
HashSet(1,2,3) == TreeSet(1,2,3)   // true
HashSet(1,2,3) == TreeSet(3,2,1)   // true
```

2. 集合是可变的还是不可变的，并不影响相等性检查。
   - 对于不可变集合而言，两个集合要么一直相等，要么一直不相等。
   - 对于可变集合而言，相等性的判断仅取决于执行相等性检查时刻的元素。这意味着某个时刻两个集合相等，但是下一个时刻这两个集合不相等。

     当我们使用可变集合作为 `HashMap` 的键时，这是一个潜在的坑。

     ```
     import collection.mutable.{HashMap,ArrayBuffer}
     val buf = ArrayBuffer(1,2,3)
     val map = HashMap(buf -> 3)
     map(buf) // 返回 3
     buf(0) += 1
     map(buf) // 抛出  NoSuchElementException 异常
     ```

     这里倒数第二行改变了 `buf` 的内容，这导致 `buf` 的哈希码被改变了。因此，基于哈希码的查找操作会指向不同于 `buf` 的存储位置。

四、创建集合对象

1. 可以通过集合名字 + 圆括号给出的元素列表的方式创建集合。如：

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

   在背后，这些代码都是调用了某个对象的 `apply` 方法。如上述代码背后展开的是：

   ```
   List.apply()
   Vector.apply(1,2,3)
   ```

   因此这是对 `List` 类的伴生对象的 `apply` 方法的调用，以及 `Vector` 类的伴生对象的 `apply` 方法的调用。它们的 `apply` 方法接收任意数量的入参，并基于这些入参构造相应的集合。

   `scala` 类库中的每个集合类都有一个带 `apply` 方法的伴生对象。至于集合类代表具体的实现是类（如 `List,Stream,Vector` ）还是特质（如 `Seq,Set,Traversable` ）并不重要。对后者而言，调用 `apply` 将会产出该特质的某种默认实现。

   如：

   ```
   List(1,2,3)                // 创建 List 集合
   Traversable(1,2,3)         // 默认的实现为 List
   mutable.Traversable(1,2,3) // 默认的实现为 ArrayBuffer
   ```

2. 除了 `apply` 外，每个集合伴生对象还定义了另一个成员方法 `empty` ，返回一个空的集合。因此，除了写 `List()` 之外，还可以通过 `List.empty` 来创建空列表。

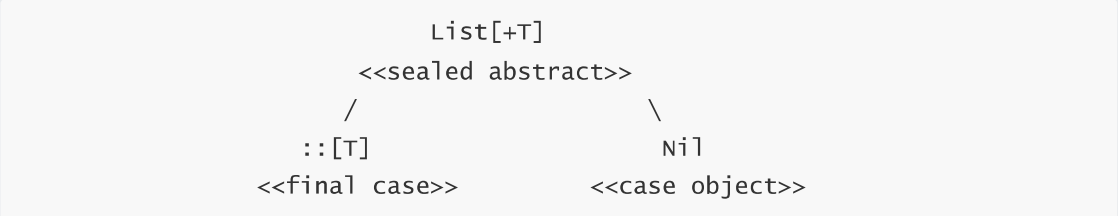
3. `Seq` 特质的后代还通过伴生对象提供了其它工厂方法，概括而言有如下这些：
   - `concat`：将任意数量的可遍历集合拼接起来。
   - `fill/tabulate`：生成指定大小的单维或多维的序列，并用某种表达式或函数来初始化。
   - `range`：用某个常量步生成整数的序列。
   - `iterate`：通过对某个起始元素反复应用某个函数来生成序列。
4. `Seq` 特质的工厂方法：
   - `S.empty`： 创建空序列。
   - `S(x,y,z)`：创建由元素 `x,y,z` 组成的序列。
   - `S.concat(xs, ys, zs)`：通过拼接 `xs, ys, zs` 的元素得到的序列。
   - `S.fill(n)(e)`：创建长度为 `n` 的序列，其中每个元素由表达式 `e` 计算。
   - `S.fill(m, n)(e)`：创建维度为 `m x n` 的序列，其中每个元素由表达式 `e` 计算。还有更高维的版本。
   - `S.tabulate(n)(f)`：创建长度为 `n` 的序列，其中下标 `i` 对应的元素由 `f(i)` 计算得出。
   - `S.tabulate(m, n)(f)`：创建 `m x n` 的序列，其中每个下标为 `(i,j)` 的元素由 `f(i,j)` 计算得出。还有更高维的版本。
   - `S.range(start, end)`：创建整数序列 `start, ... , end-1.`。
   - `S.range(start, end, step)`：创建从 `start` 开始、以 `step` 步长、直到（不包括） `end` 值为止的整数序列。
   - `S.iterate(x, n)(f)`：创建长度为 `n` 的序列，元素值为 `x, f(x), f(f(x)), ...` 。

五、List 原理

1. `List` 并不是 `Scala` 内建的语法结构，它是由`scala` 包里的抽象类 `List` 定义的，这个抽象类有两个子类`::` 和 `Nil` 。

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

   继承关系：

   - 由于 `List` 是一个抽象类，因此无法通过调用空的`List` 构造方法来定义列表，即：`new List` 是非法的。
   - `List` 有一个类型参数 `T`，参数前面的 `+` 表明列表是协变的。正因为如此，我们可以将类型为 `List[Int]` 的对象赋值给类型为 `List[Any]` 的变量：

     ```
     val xs  = List(1,2,3)
     val ys: List[Any] = xs
     ```

   - 所有的列表操作都可以通过三个基本的方法来定义：

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

这些方法在 `List` 类中都是抽象的，其具体定义出现在子类 `::` 以及子对象 `Nil` 中。

2. `Nil` 对象：它定义了一个空列表，其定义如下。

```
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")
}
```

`Nil` 对象继承自类型 `List[Nothing]`，因为协变的原因，这意味着 `Nil` 和 `List` 类型的每个实例都兼容。

这里的三个抽象方法的实现非常简单：

- `isEmpty` 直接返回 `true`。
- 其它两个方法都直接抛出异常。因为 `head` 没办法返回一个正常的值，所以它只能够通过抛出异常的方式非正常的返回；`tail` 方法也是如此。

3. `::` 类：该类表示非空列表，读作 `cons`（即英文的 `construct`）。它之所以如此命名，是为了支持用中缀 `::` 实现模式匹配。

由于模式匹配中的每个中缀操作符都被当作是用入参调用该中缀操作符对应的构造方法处理，所以 `x :: xs` 被处理为 `::(x, xs)`，其中 `::` 是一个样例类。

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

可以通过构造方法的参数直接实现超类的 `head` 和 `tail` 方法：

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

之所以可行，是因为样例类的每个参数都隐式的作为该类的字段，就跟在类内部定义了 `val head` 和 `val tail` 字段一样。而 `Scala` 允许我们用字段来实现抽象的无参方法。

4. `List` 的所有其它方法都可以用这三个基本方法来编写，如：

```
def length: Int = if(isEmpty) 0 else 1 + tail.length
def drop(n: Int): List[T] = if(isEmpty) Nil else if(n <= 0 ) this else
tail.drop(n-1)
def map[U](f: T => U): List[U] = if(isEmpty) Nil else f(head) ::
tail.map(f)
```

5. 列表的构造方法 `::` 和 `:::` 是以冒号结尾，所以它们会绑定到右操作元上。即：`x :: xs` 会被当作 `xs.::(x)`，而不是 `x.::(xs)`，因为 `x` 的类型是列表元素的类型。

   事实上 `x` 的类型和 `xs` 的元素类型可以不同：

   ```
   abstract class Fruit
   class Apple extends Fruit
   class Orange extends Fruit
   val apples = new Apple :: Nil
   val fruits = new Orange :: apples
   ```

   `::` 返回原始列表元素类型 `Apple` 和待添加元素的类型 `Orange` 最具体的公共超类。这种灵活性归功于 `::` 方法的定义：

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

   这个方法本身是多态的，其类型参数 `U` 受到 `[ U >: T]` 的约束，它必须是列表元素类型 `T` 的超类。要添加的元素必须是类型 `U` 的值并且结果是 `List[U]` 。

6. 和 `::` 一样，拼接方法也是多态的，结果类型会按需“自动放宽” 从而包含所有的列表元素。

   由于 `:::` 和 `::` 都是以冒号结尾，因此它们都是和右操作元绑定，也就是右结合的。

7. `List` 类大多数方法的真实实现并没有使用递归，因为递归会出现栈溢出的问题（很多递归都不是尾递归）。这些方法都是通过循环和`ListBuffer`来实现。

   `ListBuffer` 的 `toList` 方法调用一般是常量时间复杂度，和列表长度无关。

   ```
   package scala.collection.immutable
   final class ListBuffer[T] extends Buffer[T]{
     private var start: List[T] = Nil        // 指向 ListBuffer 中保存的所有元素的列表
     private var last0: ::[T] = _            // 指向该列表最近一个添加的 ::
     private var exported: Boolean = false   // 表示该 ListBuffer 是否已经使用过 toList 转换

     override def += (x: T) = { // 追加元素
       if (exported) copy() // 已经导出过，但是需要扩展，则拷贝底层数据
       if(start.isEmpty){
         last0 = new scala.::(x, Nil)
         start = last0
       }else{
         val last1 = last0
         last0 = new scala.::(x,Nil)
         last1.tl = last0
       }
     }
   }
   ```

   其中：

```
final case class ::[U](hd:U,  private[scala] var t1: List[U]) extends
List[U]{
  def head = hd
  def tail = tl
  override def isEmpty: Boolean = false
}
```

这里 t1 是一个 var，这意味着它是可变的。而 private[scala] 的修饰符表明 t1 只能在 scala 这个包内部被访问。

ListBuffer 的实现方式确保只有当 ListBuffer 被转成 List 之后，还需要进一步扩展时，才需要拷贝。实际上这种操作实际很少出现，ListBuffer 的大部分应用是逐个添加元素，然后最后做一次 toList 操作，此时不需要任何拷贝。

8. 列表从"外面"看是纯函数式的，但是它的实现从"里面"看是过程式的。这是 Scala 编程的一个典型策略：通过仔细界定非函数式操作，将纯函数式和效率集合起来。
9. 注意：当我们用 :: 构造列表时，会复用列表的尾部：

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

现在 ys 的尾部和 zs 的尾部是共享的，这是为了效率。因为如果每次添加新元素都拷贝 xs，则会慢很多。

由于到处都是共享的，因此如果允许改变列表的成员，则很容易出问题：

```
ys.drop(2).tail = Nil // Scala 中不支持
```

这个操作会同时截断 xs 和 zs 。所以 Scala 中不允许这么做。

六、Java 和 Scala 集合

1. Java 的集合与 Scala 集合的一个重要区别是：Scala 类库的集合更加强调不可变集合，并提供了更多将集合变成新集合的操作。
2. Scala 在 JavaConversions 对象中提供了所有主要的集合类型和 Java 集合之间的隐式转换。具体而言，你可以找到如下类型之间的双向转换：

```
Iterator <==> java.util.Iterator
Iterator <==> java.util.Enumeration
Iterable <==> java.lang.Iterable
Iterable <==> java.util.Collection
mutable.Buffer  <==> java.util.List
mutable.Set     <==> java.util.Set
mutable.Map     <==> java.util.Map
```

要允许这些转换，你只需要做一次导入：

```
import collection.JavaConversions._
```

现在你就可以拥有在 Scala 集合和对应的 Java 集合之间自动相互转换的能力了。

在 Scala 内部，这些转换是通过一个 Wrapper 对象，该 Wrapper 对象将所有操作都转发到底层集合对象来实现的。因此集合在 Java 和 Scala 之间转换时，并不会做拷贝。

一个有趣的性质是：如果你完成一次往返的转化，比如将 Java 类型转换成对应的 Scala 类型，然后再转换回 Java 类型，你得到的还是最开始的那个集合对象。

3. 还有其它一些常用的 Scala 集合可以被转换成 Java 类型，但是并没有反方向的转化与之对应。

   这些转换有：

```
Seq          ==> java.util.List
mutable.Seq  ==> java.uitl.List
Set          ==> java.util.Set
Map          ==> java.util.Map
```

   由于 Java 并不在类型上区分可变和不可变的集合，从 collection.immutable.List 转换成 java.util.List 后，如果再尝试对其进行原地修改操作，则会抛出 UnsupportedOperationException 异常。

七、Scala 集合框架

1. 很多不同的集合实现都支持相同的操作。如果对于每种集合类型我们都重新实现这些方法，将产生大量重复的代码，其中大部分代码都是从其他地方拷贝过来的。

   随着时间推进，某些集合类库中某个部分添加了新的操作、或者修改了原有操作，这将导致这些代码和其他地方拷贝的代码出现不一致。

   为解决该问题，scala 设计了集合框架，其目标是尽可能避免重复，在尽量少的地方定义每个操作。设计思路是：在集合 “模板” 中实现大多数操作，并由各个基类和实现 implementation 灵活地继承。

7.1. 集合构建器

1. 几乎所有的集合操作都是用遍历器 traversal 和构建器 builder 来实现的。
   - Traversal 的 foreach 方法解决了遍历的部分。
   - Builder 类的实例解决了构建集合的部分。

   下面给出了 Builer 类的一个简化版的示例：

```
package scala.collection.generic
class Builder[-Elem, +To] {
  def += (elem:  Elem): this.type
  def result(): To
  def clear()
  def mapResult[NewTo](f: To => NewTo): Builder[Elem, NewTo]=...
}
```

   其中 Elem 为元素类型， To 是返回的集合类型。

   可以用 b += x 向构建器 b 添加元素 x ；也可以一次性添加多个元素，如 b += (x, y) 以及 b ++ xs 。

   result() 方法从构建器中返回一个集合。在获取 result 之后，构建器的状态是未定义的，可以调用 clear() 方法将它重设为新的空状态。

2. 通常，一个构建器可以引用另一个构建器来组装某个集合的元素，不过需要对另外这个构建器的结果进行变换（比如给它一个不同的类型），因此可以通过 `Builer` 的 `mapResult` 方法来简化。

   例如，`ArrayBuffer` 本身就是一个构建器，因此对它调用 `result` 方法会返回本身。如果希望用 `ArrayBuffer` 来创建一个数组的构建器，则可以用 `mapResult`：

```
val buf = new ArrayBuffer[Int]
val bldr = buf mapResult(_.toArray) // builder 为一个 Builder[Int,
Array[Int]]
```

7.2 抽取公共操作

1. `scala` 集合遵循“相同结果类型”的原则：只要有可能，对集合的变换操作将交出相同类型的集合。如：`filter` 操作应用于 `List` 将返回 `List` 类型，应用于 `Map` 将返回 `Map` 类型。
2. `scala` 集合类库是通过使用所谓的实现特质 `implementation traits` 中的泛型构建器和遍历器来避免代码重复、并达成“相同结果类型” 原则的。这些特质的命名中都带有 `Like` 后缀。例如 `IndexedSeqLike` 是 `IndexedSeq` 的实现特质， `TraversableLike` 是 `Traversable` 的实现特质。诸如 `Traversable` 或 `IndexedSeq` 这样的集合类的具体方法的实现，都是从这些特质中继承下来的。
3. 实现特质不同于一般的集合，它们有两个类型参数：它们不仅在集合元素的类型上是参数化的，它们在集合的表示类型(`representation type` ，也就是底层的集合，比如 `Traversable` 类型默认的底层集合为 `List`) 上也是参数化的。

   例如，`TraversableLike` 特质为：

```
trait TraversableLike[+Elem, +Repr] {...}
```

   类型参数 `Elem` 表示集合的元素类型，类型参数 `Repr` 为集合的表示类型。

   对于 `Repr` 并没有什么限制，它可以被实例化为不是 `TraversableLike` 的子类，例如 `String` 或者 `Array` 也可以作为 `Repr` 的类型实例化。

4. 我们以 `filter` 为例，这个操作定义在 `TraversableLike` 特质中，只定义了一次，但是对所有集合类都可用。

```
package scala.collection

trait TraversableLike[+Elem, +Repr]{
  def newBuiler: Builer[Elem, Repr]  // 延迟实现
  def foreach[U](f: Elem => U)       // 延迟实现
  ...
  def filter(p: Elem => Boolean): Repr = {
    val b = newBuiler
    foreach{ elem => if(p(elem)) b += elem}
    b.result
  }
}
```

   该特质声明了两个抽象方法：`newBuiler` 和 `foreach` ，这些方法在具体的集合类中实现。

   `filter` 操作对于所有使用这些方法的集合的实现方式都是一致的。

   - 它首先使用 `newBuiler` 构造出一个新的、表示类型为 `Repr` 的构建器。

- 然后使用 `foreach` 遍历当前集合中的所有元素，如果元素满足给定的前提条件，则将该元素添加到构建器中。
- 最后，构建器收集到的元素通过构建器的 `result` 方法，以 `Repr` 集合类型的实例返回。

5. 集合的 `map` 操作要更复杂一些，因为 `map` 操作的返回类型不一定是原始的集合类型。

   举个例子：

```
import collection.immutable.BitSet
val bits = BitSet(1,2,3)
bits map (_ * 2)     // 结果是一个 BitSet
bits map (_.toFloat) // 结果是一个 Set[Float]
```

   可以看到，`map` 操作的返回类型取决于传入的函数。

   `Scala` 解决该问题的方法是重载：不是 `Java` 采用的那种简单的重载（那样不够灵活），而是隐式参数提供的更系统化的重载。下面给出 `TraversableLike` 的 `map` 实现：

```
def map[B, That](f: Elem => B)(implicit bf: CanBuildFrom[Repr, B,
That]):That = {
  val b = bf(this)
  for (x <- this) b += f(x)
  b.result
}
```

   这里主要区别在于：`filter` 用的是 `TraversableLike` 的抽象方法 `newBuilder`，而 `map` 用的是一个以额外的隐式参数的形式传入的、类型为 `CanBuildFrom` 的构建器工厂 `bf` 。

   `CanBuildFrom` 特质的定义为：

```
package scala.collection.generic
trait CanBuildFrom[-From, -Elem, +To]{
  // 创建新的构建器
  def apply(from: From): Builder[Eleem, To]
}
```

   该特质代表了构建器工厂，有三个类型参数：`Elem` 表示要构建的集合的元素类型，`To` 表示要构建的集合类型（即：目标集合类型），`From` 表示从哪个集合来构建。

   通过定义正确的构建器工厂的隐式定义，可以按需定制正确的类型行为。

   例如，`BitSet` 类的伴生对象包含一个类型为 `CanBuildFrom[BitSet, Int, BitSet]` 的构建器工厂。这意味着当操作一个 `BitSet` 时，可以构造出另一个 `BitSet`，只要构建的集合的元素类型为 `Int` 。

   如果集合类型不是 `Int`，则退而求其次，采用另一个隐式构建器工厂，一个在 `mutable.Set` 伴生对象中实现的隐式构建器工厂。这是一个更通用的构建器工厂，定义为（其中 `A` 为泛型的类型参数）：`CanBuildFrom[Set[_]],A, Set[A]]` 。这意味着当操作一个以 `Set[_]` 通配类型表示的任意类型的 `Set` 时，仍然可以构建出一个 `Set`，而无论元素类型 `A` 是什么。

   有了这两个 `CanBuildFrom` 的隐式实例，就可以依赖 `Scala` 的隐式解析规则来选取合适的、并且是最具体的那个构建器工厂了。

6. 上述机制对于静态类型的 `map` 处理较好，但是动态类型的 `map` 还需要一层额外的处理机制。

```
val xs: Iterable[Int] = List(1,2,3) // xs 静态类型为 Iterable，动态类型为 List
val ys = xs.map(x => x*x)  // ys 的静态类型为 Iterable，  动态类型也为 List
```

`CanBuildFrom` 的 `apply` 方法接受原始集合作为入参传入，大多数构建器工厂都将这个调用转发到原始集合的 `genericBuilder` 方法，这个 `genericBuilder` 方法进行调用属于该集合的构建器。

即：`scala` 使用静态的隐式解析规则来决定 `map` 的类型约束，用虚拟分发来选择与这些类型约束相对应的最佳动态类型。

7. `Map/Array` 都是一种函数，因为它们都继承自 `Function1` 特质，并且有 `apply` 方法。如：

```
val m = Map("a" -> 1, "b" -> 2)
m("a") // 就像函数调用一样
val arr = Array("a","b","c")
arr(0) // 就像函数调用一样
```

8. 如果想要自定义一个新的集合类库到框架中，需要注意如下几点：
    - 决定该集合是可变的还是不可变的。
    - 选择合适的特质作为集合的基础。
    - 从合适的实现特质继承来实现大多数集合操作。
    - 如果你想让 `map` 和类似操作返回你的集合类型，你需要在你的类的伴生对象中提供一个隐式的 `CanBuildFrom` 。