集合 collection(一)

1. `Scala` 自带了一个强大的集合类库， 它们具有易用、精简、安全、快速、通用的优点。
   - 易用：一组`20`到`50`个方法足以支持大部分集合问题，不需要理解复杂的循环或者递归。

     持久化的集合加上无副作用的操作意味着不需要担心会意外地污染已有的集合。迭代器和集合更新之间的相互影响也没有了。
   - 精简：使用一个方法就可以做到一个或者多个循环才能完成的事情。
   - 安全：`Scala`集合的静态类型和函数式本质意味着我们可能会犯的错误大多数都可以在编译期间被发现。
   - 快速：类库对集合操作进行了调整和优化，通常来说集合都非常高效。

     另外，`Scala`集合对于多核并行执行也进行了适配，并行集合和串行集合一样支持同样的操作，无需学习任何新的语法。

     可以简单的通过调用`par`方法将一个串行集合转换成一个并行集合。
   - 通用：集合对于任何类型都可以提供相同的操作，只要这个操作对于该类型而言是讲得通的。

     例如，所有的集合都可以用`。emtpy`方法创建空集合。
2. 所有的集合类都可以在`scala.collection`包，或者它的子包`mutable、immutable、generic`中找到。

   `Scala`的集合类库区分了可变集合与不可变集合，其中：`Array`永远是可变的，`List`永远是不可变的，而`Set`和`Map`都拥有可变版本和不可变版本。
   - `scala.collection.mutable`： 包含可变集合类
   - `scala.collection.immutable`： 包含不可变集合类。
   - `scala.collection.generic`： 包含了实现集合的构建单元。集合框架的日常使用通常不需要使用这个包，极端情况除外。
   - `scala.collection`： 该包中的集合类型既可以是可变的，也可以是不可变的。一般而言，这个包的根`root`集合定义了与不可变集合相同的接口，而`immutable`包中的可变集合会在上述不可变接口基础上添加一些有副作用的修改操作。

   `scala`默认使用不可变集合。
3. 注意：这里的集合表示`collection`，而`Set`集是`collection`中具体的一种类型。
4. 最重要的集合如下：

```
Traversable
            --> Iterable
                        --> Seq
                               --> IndexdSeq
                                           --> Vector
                                           --> ResizableArray
                                           --> GenericArray
                                --> LinearSeq
                                            --> MutableList
                                            --> List
                                            --> Stream
                                --> Buffer
```

```
                                        --> ListBuffer
                                        --> ArrayBuffer
                        --> Set
                                --> SortedSet
                                        --> TreeSet
                                --> HashSet(可变的)
                                --> LinkedHashSet
                                --> HashSet(不可变的)
                                --> BitSet
                                --> EmptySet, Set1, Set2, Set3, Set4
                        --> Map
                                --> SortedMap
                                        --> TreeMap
                                --> HashMap(可变的)
                                --> LinkedHashMap(可变的)
                                --> HashMap(不可变的)
                                --> EmptyMap, Map1, Map2, Map3, Map4
```

这些类拥有相当多的共性：

- 每个集合都可以用一致的语法来创建：先写出类名，再写出元素：

```
Traversable(1,2,3)
Iterable("x", "y", "z")
```

- 所有集合的 `.toString` 方法也会产出对应格式的输出：类名名称加上用圆括号包围起来的元素。
- 所有集合都支持由`Transversable`提供的`API`，但是这些方法都返回自己的类而不是根类`Transversable`。
- 相等性对于所有集合类而言也是一致的。

5. 大部分类都有三个版本：根版本、可变版本、不可变版本。唯一例外是`Buffer`，它只有可变版本。

一、Traversable

1. `Traversable` 的所有操作为（`xs` 是一个 `Traversable` 对象）：
   - 抽象操作：
     - `xs foreach f`：对`xs`的每个元素执行函数`f`。
   - 添加操作：
     - `xs ++ ys`： 拼接`xs`和`ys`的所有元素， 返回一个新的`Traversable`。`ys`是一个`TraversableOnce`，它既可以是一个`Traversable`也可以是一个`Iterator`。
   - 映射操作：
     - `xs map f`：通过对`xs`的每个元素应用函数`f`，返回一个新的`Traversable`。
     - `xs flatmap f`：通过对`xs`的每个元素(是一个`Traversable`)的每个元素应用函数`f`，返回一个新的`Traversble`。
     - `xs collect f`： 通过对`xs`的每个元素应用偏函数`f`，并将定义的结果收集起来得到的集合。
   - 转化操作：
     - `xs.toArray`： 将`Traversable`转换成`Array`。

- `xs.toList`： 将 `Traversable` 转换为 `List` 。
- `xs.toIterable`： 将 `Traversable` 转换为 `Iterable`。
- `xs.toSeq`： 将 `Traversable` 转换为 `Seq` 。
- `xs.toIndexedSeq` ： 将 `Traversable` 转换成 `IndexedSeq` 。
- `xs.toStream` ： 将 `Traversable` 转换成 `Stream` 。
- `xs.toSet` ： 将 `Traversable` 转换成 `Set` 。
- `xs.toMap` ： 将 `Traversable` 转换成 `Map` 。

- 拷贝操作：
  - `xs.copyToBuffer(buf)`： 将 `xs` 的所有元素拷贝到缓冲 `buf` 中。
  - `xs.copyToArray(arr,s,len)`： 将 `xs` 的从下标 `s` 开始的、最多 `len` 个元素拷贝到 `arr` 中 。 `s` 和 `len` 是可选的， 默认从 `0` 开始， 拷贝长度为 `xs` 大小。
- 大小信息：
  - `xs.isEmpty`： 测试 `xs` 是否为空。
  - `xs.nonEmpty`： 测试 `xs` 是否非空。
  - `xs.size`： 返回 `xs` 的元素数量。
  - `xs.hasDefiniteSize`： 如果 `xs` 的大小有限， 则返回 `true`。
- 获取元素：
  - `xs.head`： 获取第一个元素(如果是有序的集合)， 或者某个元素(如果是无序的集合)。
  - `xs.headOption`： 获取第一个元素或者某个元素， 如果 `xs` 为空时返回 `None`。
  - `xs.last`： 获取最后一个元素(如果是有序的集合)， 或者某个元素(如果是无序的集合)。
  - `xs.lastOption`：获取最后一个元素或者某个元素， 如果 `xs` 为空时返回 `None`。
  - `xs find p`：返回`xs` 中满足 `p` 的首个元素，当 `xs` 为空时返回 `None` 。
- 子集：
  - `xs.tail`： 集合去掉 `xs.head` 的部分。
  - `xs.init`： 集合去掉 `xs.last` 的部分。
  - `xs.slice(from,to)`：集合位于`from` (包含)到 `to` (不包含)区间的元素组合的新集合。
  - `xs.take(n)`： 包含集合`xs` 的前 `n` 个元素(如果是有序的集合)， 或者任意 `n` 个元素(如果是无序的集合)。
  - `xs.drop(n)`： 集合去掉 `xs.take(n)` 的部分。
  - `xs.takeWhile(p)`： 寻求满足 `p` 条件为`true` 的第一个元素， 假设其索引为`n`， 则返回 `xs.take(n)`。
  - `xs.dropWhile(p)`： 寻求满足 `p` 条件为`true` 的第一个元素， 假设其索引为`n`， 则返回 `xs.drop(n)` 。
  - `xs.filter(p)`： 包含`xs` 中所有满足条件`p` 的元素。
  - `xs.withFilter(p)`： `xs.filter(p)` 的延迟版本。它不生成新的集合， 而是一个 `WithFilter` 对象， 它主要用于 `map/flatMap/foreach/withFilter` 等链式操作，在后续操作才生成新的集合， 从而提高性能。
  - `xs.filterNot(p)`： 包含`xs` 中所有不满足条件`p` 的元素。
- 拆分：
  - `xs.splitAt(n)`： 在指定位置拆分 `xs` ，返回一对集合 `(xs.take(n), xs.drop(n))` 。
  - `xs.span(p)`： 根据条件 `p` 切分`xs`， 返回一对集合 `(xs.takeWhile(p), xs.dropWhile(p))`。

    - `xs.partition(p)`： 根据条件`p`分离`xs`， 返回一对集合`(xs.filter(p), xs.filterNot(p))`。
    - `xs.groubBy(f)`： 根据分区函数`f`将`xs`转化为各分区到子集合的映射。其中`f`将元素映射为分区的`key`。
  - 元素条件：
    - `xs.forall(p)`： 是否`xs`的所有元素都满足条件`p`。
    - `xs.exists(p)`： 是否`xs`存在元素满足条件`p`。
    - `xs.count(p)`： `xs`中满足条件`p`的元素数量。
  - 折叠：
    - `(z /: xs)(op)`： 以`z`开始从左到右依次对`xs`中的连续元素应用二元操作`op`。相比`xs reduceLeft op`，这里提供了初始值`z`。
    - `(xs :\ z)(op)`： 以`z`开始从右到左依次对`xs`中的连续元素应用二元操作`op`。相比`xs reduceRight op`，这里提供了初始值`z`。
    - `xs.foldLeft(z)(op)`： 同`(z /: xs)(op)`。
    - `xs.foldRight(z)(op)`： 同`(xs :\ z)(op)`。
    - `xs reduceLeft op`： 从左到右依次对非空集合`xs`的连续元素应用二元操作`op`。
    - `xs reduceRight op`： 从右到左依次对非空集合`xs`的连续元素应用二元操作`op`。
  - 特殊折叠：
    - `xs.sum`： 数值集合`xs`中元素的和。
    - `xs.product`： 数值集合`xs`中元素的积。
    - `xs.min`： 有序集合`xs`中元素值的最小值。有序指的是元素之间可以排序，而不是序列已经排好序。
    - `xs.max`： 有序集合`xs`中元素值的最大值。
  - 字符串：
    - `xs addString (b,start,sep,end)`： 将一个显示`xs`从`start`到`end`的元素的字符串， 按照`sep`分隔， 添加到`StringBuilder b`中。`start, end, sep`均为可选。
    - `xs mkString (start,sep,end)`： 返回一个字符串，它包含`xs`从`start`到`end`的元素的字符串， 按照`sep`分隔。`start, end, sep`均为可选。
    - `xs.stringPrefix`： 返回集合的名称字符串（如`"List"`字符串）。
  - 视图：
    - `xs.view`： 产生一个`xs`的视图。
    - `xs.view(from, to)`： 产生一个代表`xs`中某个下标区间元素的视图。
2. `Traversable`特质唯一的抽象操作是`foreach`(当然还有其它非抽象操作)：

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

   该方法的本意是遍历集合中的所有元素，并对每个元素应用给定的操作`f`。

   - `f`的类型为`Elem => U`，其中`Elem`是集合的元素类型，而`U`是任意的结果类型。
   - 对`f`的调用仅仅是为了副作用，事实上`foreach`会丢弃函数调用`f`的所有结果。

   实现`Traversable`的集合类只需要定义`foreach`方法即可。

3. `Traversable`特质还定了很多具体方法， 其中包括：
   - `++`： 将两个`Traversable`加在一起， 或者将某个迭代器的所有元素添加到`Traversable`。
   - 映射操作： `map/flatMap/collect`通过对集合元素应用某个函数来产生一个新的集合。

- 转化： `toIndexedSeq/toIterable/toStream/toArray/toList/toSeq/toSet/toMap` 将一个 `Traversable` 转换到更具体的集合。 如果原始的集合已经是目标类型， 则直接返回原始集合。

  例如， 对`List` 对象调用 `.toList` 会直接返回它本身。

- 拷贝： `copyToBuffer/copyToArray` 分别将元素拷贝到 `Buffer` 或者 `Array`

- 大小操作： `isEmpty/nonEmpty/size/hasDefiniteSize` 可用于判断集合大小。

  能被遍历的集合可以是有限的， 也可以是无限的。 如表示自然数的流 `Stream.from(0)` 就是一个无限可遍历集合。此时 `hasDefiniteSize` 返回 `false`， 此时 `size` 方法会报错或者根本不返回。

- 元素获取： `head/last/headOption/lastOption/find` 这些操作用于获取元素。 注意：并非所有的集合都有定义定义好的“第一个” 或者 “最后一个”的语义。

  如果某个集合总是以相同的顺序交出元素，那么它就是有序的 `ordered` 。 对于有序的集合，它有“第一个” 和“最后一个” 语义。

  大多数集合都是有序的，但是有一些不是， 如`HashSet` 。 `Scala` 提供了对所有集合类型的有序版本，如 `HashSet` 的有序版本是 `LinkedHashSet` 。

- 子集获取：`tail/init/slice/take/takeWhile/drop/dropWhile/filter/filterNot/withFilter` 等操作都用于获取子集。

- 拆分： `splitAt/span/partition/groupBy` 等操作将集合切分成若干个子集。

- 元素测试： `exists/forall/count` 等操作用于使用给定的条件`p` 来对集合元素进行测试。

- 折叠： `foldLeft、foldRight、 /: 、 :\、 reduceLeft、reduceRight` 等操作用于对集合中连续的元素应用某个二元操作。

- 特定折叠： `sum/product/min/max` 等操作用于特定类型的集合(如数值类型或可比较类型)的特定折叠操作 (求和、乘积、最小、最大)。

- 字符串操作： `mkString/addString/stringPrefix` 操作用于将集合转换成字符串。

- 视图操作： `view` 操作用于构建视图， 其中视图是一个惰性求值的集合。

二、Iterable

1. `Iterable` 特质的所有方法都是用抽象方法 `iterator` 来定义的，该抽象方法的作用是逐个交出集合的元素。从 `Traversable` 继承下来的 `foreach` 方法在 `Iterable` 中的定义就用到了 `iterator` ：

   ```
   def foreach[U](f: Elem => U): Unit={
     val it = iterator
     while (it.hasNext) f(it.next())
   }
   ```

   很多 `Iterable` 的子类都重写了这个在 `Iterable` 中的 `foreach` 的标准实现， 因为它们可以提供更高效的实现。

   注意： `foreach` 是 `Traversable` 中所有操作实现的基础，因此它的性能表现非常重要。

2. `Iterable` 还有两个方法返回迭代器： `grouped` 和 `sliding` 。 但是这两个迭代器并不返回单个元素， 而是原始集合的某个子序列。

可以通过入参来指定这些子序列的最大长度。`grouped` 方法将元素进行分段， 而 `sliding` 交出的是对元素的一个滑动窗口。

```
val xs = List(1, 2, 3, 4, 5)
val g1 = xs grouped 3 // 迭代的结果为： List(1,2,3)， List(4,5)
val g2 = xs sliding 3 // 迭代的结果为： List(1,2,3)， List(2，3,4)，
List(3,4,5)
```

3. `Iterable` 特质还对 `Traversable` 特质添加了其它的一些方法， 这些方法只有在有迭代器存在的情况下才能得以高效的实现。

   `Iterable` 包含的操作：

   - 抽象方法：
     - `xs.iterator`： 按照与 `foreach` 遍历元素的顺序交出 `xs` 中每个元素的迭代器。
   - 其它迭代器：
     - `xs grouped size`： 交出该集合固定大小片段的迭代器。
     - `xs sliding size`： 交出该集合固定大小滑动窗口的迭代器。
   - 子集合：
     - `xs takeRight n`： 包含 `xs` 的后 `n` 个元素的集合。如果未定义顺序， 则为任意的 `n` 个元素。

       > `Traversable` 定义的 `xs take n` 获取 `xs` 的前 `n` 个元素的集合。

     - `xs dropRight n`： 集合移除 `xs takeRight n` 的部分。
   - 拉链：
     - `xs zip ys`： 由 `xs` 和 `ys` 对应位置元素的元组组成的 `Iterable` 。
     - `xs zipAll (ys， x，y)`： 由 `xs` 和 `ys` 对应位置元素的元组组成的 `Iterable`，其中较短的序列用 `x` (如果 `xs` 较短) 或者 `y` (如果 `ys` 较短) 的元素值扩展到相同的元素。
     - `xs.zipWithIndex`： 由 `xs` 中的元素及其下标的对偶组成的 `Iterable` 。
   - 比较：
     - `xs sameElements ys`： 测试 `xs` 是否和 `ys` 包含相同顺序的相同元素。
4. 为什么要在 `Iterable` 之上增加一个 `Traversable` 特质？因为有时候 `Traversable` 提供的 `foreach` 要比 `iterator` 的性能更好。

   例如我们有以下的二叉树：

   ```
   sealed abstract class Tree
   case class Branch(left: Tree， right: Tree) extends Tree
   case class Node(elem: Int) extends Tree
   ```

   如果我们让 `Tree` 继承自 `Traversable[Int]`， 则我们可以这样定义一个 `foreach` 方法：

```
sealed abstract class Tree extends Traversable[Int]{
  def foreach[U](f: Int => U) = this match{
    case Node(elem) =>
      f(elem)
    case Branch(left, right) =>
      left foreach f
      right foreach f
  }
}
```

这种遍历方式非常高效， 时间复杂度为 $O(N)$ ，其中 N 为叶子结点数量。

如果我们让 `Tree` 继承自 `Iterable[Int]`， 则我们可以这样定义一个 `iterator` 方法：

```
sealed abstract class Tree extends Iterable[Int]{
  def iterator: Iterator[Int] = this match{
    case Node(elem) => Iterator.single(elem)
    case Branch(left, right) => left.iterator ++ right.iterator
  }
}
```

表面上看起来这个实现并不复杂。 但是对于 `++` 的实现而言， 存在一个运行效率的问题： 像 `left.iterator ++ right.iterator` 这样拼接起来的迭代器， 每交出一个元素， 都需要多一层计算来判断是用哪个迭代器(`left.iterator` 还是 `right.iterator`)。 因此总体而言， 其计算复杂度为 $O(N \log N)$ 。

5. `Iterable` 下面的三个特质 `Seq、Set、Map` 的共同特点是： 它们都实现了 `PartialFunction` 特质， 定义了相应的 `apply` 方法和 `isDefinedAt` 方法。 不过， 每种特质实现 `PartialFunction` 的方式各不相同。

   例如： 对于 `Seq` 而言 `apply` 是位置下标， 元素下标从 `0` 开始；对于 `Set`， `apply` 是成员测试；对于 `Map`， `apply` 是选择指定 `key` 的值。

```
Seq(1,2,3)(0) // 返回 1
Set(1,2,3)(0) // 返回 false
Map(1-> "a", 0 -> "b")(0) // 返回 "a"
```

三、Seq

1. `Seq` 代表序列， 它有长度 `length` 、且元素从固定的下标 `0` 开始。
2. `Seq` 包含的操作：
   - 下标和长度：
     - `xs(i)`： 获取下标 `i` 对应的元素。 也可以展开为 `xs apply i` 。
     - `xs isDefinedAt i` ： 测试 `i` 是否是 `xs` 的一个有效索引。
     - `xs.length`： 返回 `xs` 的长度。
     - `xs.lengthCompare ys`： 对比 `xs` 和 `ys` 的长度。 如果 `xs` 长度较短， 则返回 `-1`； 如果 `xs` 长度较长，则返回 `1`； 如果长度相同， 则返回 `0` 。 如果其中一个序列的长度为无限长， 比较规则仍然有效。
   - 下标检索：
     - `xs indexof x` ： `xs` 中首次出现 `x` 元素的下标(允许多个存在)。

        - `xs lastIndexOf x`： `xs` 中最后一个等于 `x` 元素的下标(允许多个存在)。
        - `xs indexOfSlice ys`： `xs` 中首个出现序列 `ys` 的位置。
        - `xs lastIndexOf ys`： `xs` 中最后一个出现序列 `ys` 的位置。
        - `xs indexWhere p`： `xs` 中首个满足 `p` 的元素位置(允许多个存在)。
        - `xs segmentLength (p, i)`： `xs` 中从 `i` 开始的、连续满足 `p` 的片段的最大长度。
        - `xs prefixLength p`： `xs` 中连续满足 `p` 的最长前缀长度。
    - 添加:
        - `x +: xs`: 将 `x` 追加到 `xs` 头部得到新序列。
        - `xs +: x`： 将 `x` 追加到 `xs` 尾部得到新序列。
        - `xs padTo (len, x)`: 将`x` 追加到 `xs` 尾部、直到形成长度为 `len` 的新序列。
    - 更新:
        - `xs patch(i, ys, r)`: 将 `xs` 中从下标 `i` 开始的 `r` 个元素替换为 `ys` 序列得到新序列。
        - `xs updated(i, x)`: 将`xs` 中下标为 `i` 的元素替换为 `x` 得到新序列。
        - `xs(i) = x`: 将`xs` 中下标 `i` 的元素更新为`x` 。 也可以展开为 `xs.update(i, x)`， 但是仅对 `mutable.Seq` 有效。
    - 排序:
        - `xs sorted`: 用 `xs` 元素类型的标准顺序对 `xs` 排序， 得到新的排序好的序列。
        - `xs sortWith lessThan`: 以 `lessThan` 作为比较操作对 `xs` 排序， 得到新的排序好的序列。
        - `xs sortBy f`: 以 `f` 作为比较函数对 `xs` 排序， 得到新的排序好的序列。
    - 反转:
        - `xs.reverse`: 将 `xs` 顺序翻转，得到新的翻转后的序列。
        - `xs.reverseIterator`: 返回一个迭代器， 其迭代结果是 `xs` 的逆序对应的元素。
        - `xs.reverseMap(f)`： 逆序对 `xs` 的元素映射 `f` 后得到新的序列
    - 比较:
        - `xs startsWith ys`: `xs` 是否以 `ys` 开始。
        - `xs endsWith ys`: `xs` 是否以 `ys` 结束
        - `xs contains x`： `xs` 是否包含 `x`。
        - `xs containSlice ys`: `xs` 是否包含和 `ys` 相等的连续子序列。
        - `(xs corresponds ys)(p)`: 测试 `xs` 和 `ys` 对应元素是否满足二元条件函数 `p` 。
    - 集合操作:
        - `xs intersect ys`: 返回序列 `xs` 和 `ys` 的交集， 保持 `xs` 中的顺序
        - `xs diff ys`: 返回序列 `xs` 和 `ys` 的差集， 保持 `xs` 中的顺序。
        - `xs union ys`: 返回序列 `xs` 和 `ys` 的并集， 等同于 `xs ++ ys`。
        - `xs.distinct`： 返回 `xs` 不包含重复元素的子序列。
3. `Seq` 的下标和长度操作`apply/isDefinedAt/length/indices/lengthCompare`:
    - 对 `Seq` 而言， `.apply` 操作的含义是下标索引。
    - 另外， `Seq` 的 `length` 方法是通用集合的 `size` 方法的别名。
    - `lengthCompare` 允许你对两个序列的长度进行比较， 哪怕其中一个序列的长度是无限的。
4. 如果序列是可变的， 则它会提供额外的、带有副作用的 `update` 方法， 从而允许序列元素被更新。

   注意 `update` 方法和 `updated` 方法的区别:
    - `update` 方法用于原地修改序列的某个元素值， 且仅用于可变序列。

- `updated` 方法对所有序列都可用， 并且总是返回新的序列，而不是修改原序列。

四、LinearSeq/IndexedSeq

1. `Seq` 特质有两个子特质： `LinearSeq` 和 `IndexedSeq` 。 这两个特质并没有添加额外的操作，不过它们各自拥有自己的性能特点：
   - `LinearSeq` 拥有高效的 `head` 和 `tail` 操作。如 `List,Stream` 都是常用的 `LinearSeq` 。
   - `IndexedSeq` 拥有高效的 `apply，length` 操作。 如果是可变的， 则它还有高效的 `update` 操作 。如 `Array,ArrayBuffer` 都是常用的 `IndexedSeq` 。
2. `Vector` 提供了介于 `LinearSeq,IndexedSeq` 之间的折中： 它既拥有 $O(1)$ 时间的索引开销，又拥有 $O(n)$ 时间的访问开销。

五、Buffer

1. `Buffer` 是一种重要的可变 `Seq` ， 它不仅可以对已有元素的更新， 还允许元素插入、移除、和在末尾高效的追加新元素。

   `Buffer` 主要支持的新的方法有：

   - 用于为尾部添加新元素的 `+=` 和 `++=` 方法。
   - 用于在头部添加新新元素的 `+=:` 和 `++=:` 方法。
   - 用于插入元素的 `insert` 和 `insertAll` 方法。
   - 用于移除元素的 `remove` 和 `-=` 方法。
2. 两个最常用的 `Buffer` 实现是 `ListBuffer` 和 `ArrayBuffer` 。
   - `ListBuffer` 背后是 `List` ， 支持到 `List` 的高效转换。
   - `ArrayBuffer` 背后是 `Array` ， 支持到 `Array` 的高效转换
3. `Buffer` 特质的操作：
   - 添加：
     - `buf += x` ： 将元素 `x` 追加到 `buf` 尾部， 返回 `buf` 本身。
     - `buf += (x,y,z)` ： 将给定的元素追加到 `buf` 尾部， 返回 `buf` 本身。
     - `buf ++= xs` ： 将 `xs` 中的所有元素追加到 `buf` 尾部， 返回 `buf` 本身。
     - `x +=: buf` ： 将元素 `x` 追加到 `buf` 头部 ， 返回 `buf` 本身 。
     - `xs ++: buf` ： 将 `xs` 中的所有元素追加到 `buf` 头部， 返回 `buf` 本身。
     - `buf insert (i, x)` ： 将元素 `x` 插入到 `buf` 下标 `i` 的位置， 返回 `buf` 本身。
     - `buf insertAll (i, xs)` ： 将 `xs` 中所有元素插入到 `buf` 下标 `i` 开始的位置， 返回 `buf` 本身。
   - 移除：
     - `buf -= x` ： 移除 `buf` 中的 `x` ， 返回 `buf` 本身。
     - `buf remove i` ： 移除 `buf` 中下标为 `i` 的元素， 返回 `buf` 本身。
     - `buf remove (i, n)` ： 移除 `buf` 中下标为 `i` 的连续 `n` 个元素， 返回 `buf` 本身。
     - `buf trimStart n` ： 移除`buf` 的头部 `n` 个元素， 返回 `buf` 本身。
     - `buf trimEnd n` ： 移除 `buf` 的尾部 `n` 个元素， 返回 `buf` 本身。
     - `buf.clear()` ： 移除 `buf` 中的所有元素， 返回 `buf` 本身.
   - 拷贝：
     - `buf.clone` ： 拷贝一份跟 `buf` 元素相同的`Buffer` ， 返回拷贝后的结果。

六、Array

1. `Array` 是 `Scala` 提供的数组，它是可变对象，即：可以修改 `Array` 的元素内容。

   数组在 `scala` 中是一种特殊的集合。

   - 一方面，`scala` 的数组跟 `java` 的数组一一对应。即，`scala` 中的 `Array[Int]` 是用 `java` 的 `int[]` 来表示，`scala` 中的 `Array[Double]` 是用 `java` 的 `double[]` 来表示。
   - 另一方面，`scala` 的数组比 `java` 数组提供更多的功能。
     - 首先，`scala` 数组支持泛型。即，可以有 `Array[T]`，`T` 为类型参数或者抽象类型。
     - 其次，`scala` 数组跟 `scala` 的序列兼容（可以在要求 `Seq[T]` 的任何地方传入 `Array[T]` ）。
     - 最后，`scala` 数组还支持所有的序列操作。
2. `scala` 通过使用隐式转换来支持 `java` 数组额外的那些功能。`Array` 并不是 `Seq` 类型，因为 `Array` 并不是 `Seq` 的子类型。

   - 每当数组被用于 `Seq` 时，它都被隐式转换为 `Seq` 的子类，这个子类叫做 `scala.collection.mutable.WrappedArray`。

     ```
     val a1 = Array(1,2,3)
     val seq: Seq[Int] = a1 // seq 为 WrappedArray[Int] 类型
     val a2 = seq.toArray   // a2 为 Array[Int] 类型
     ```

     从这些交互可以看到：数组和序列是兼容的，因为有一个从 `Array` 到 `WrappedArray` 的隐式转换。

     如果反过来，希望从 `WrappedArray` 转换为 `Array`，则可以通过 `Traversable` 中定义的 `toArray` 方法。此时可以得到与原始数组相同的结果（`a2` 的内容和 `a1` 完全相同）。
   - 另外，每当对数组执行序列的操作时，`Array` 都被转换为 `ArrayOps` 对象。`Array` 到 `ArrayOps` 对象的转换仅仅是将所有的序列方法 “添加” 到数组，但是并没有将数组转换成 `Seq` 的子类。`ArrayOps` 对象支持所有的序列方法。

     通常 `ArrayOps` 的生命周期很短：通常在调用完序列方法之后就不会再使用了，因此它的存储空间可以被立即收回。现代的 `VM` 会完全避免创建这个对象。

   这两种隐式转换的区别可以通过以下示例看到：

   ```
   val a1 = Array(1,2,3)
   val seq: Seq[Int] = a1   // seq 为 WrappedArray[Int] 类型
   val seq2 = seq.reverse   // seq2 为 WrappedArray[Int] 类型
   val ops: collection.mutable.ArrayOps[Int] = a1 //ops 为 ArrayOps[Int] 类型
   val a3 = ops.reverse    // a3 为 Array[Int] 类型
   ```

   对 `WrappedArray` 调用 `reverse` 返回 `WrappedArray` 类型，这合乎逻辑。因为对 `Seq` 调用 `reverse` 都会返回 `Seq` 。

   对 `ArrayOps` 调用 `reverse` 返回的是 `Array` 而不是 `Seq` ，也不是 `ArrayOps` 。
3. 通常你从来都不会自己去创建 `ArrayOps` 类型的对象，只需要对数据调用 `Seq` 方法即可：

   ```
   val a1 = Array(1,2,3)
   val a3 = a1 reverse // 等价于先人工创建  ArrayOps[Int] 对象，在对其调用 reverse
   ```

   `scala` 会自动插入 `ArrayOps` 对象，因此上述代码和下面的代码等价：

```
val ops: collection.mutable.ArrayOps[Int] = a1 //ops 为 ArrayOps[Int] 类型
val a3 = ops.reverse    // a3 为 Array[Int] 类型
```

4. 编译器是如何知道选择 `WrappedArray` 隐式转换，还是选择 `ArrayOps` 隐式转换呢？毕竟这两种隐式转换都可以将数组转换为支持 `reverse` 的类型。

   答案是：这两个隐式转换存在优先级。`ArrayOps` 的优先级高于 `WrappedArray` 转化。前者定义在 `Predef` 对象中，后者定义在 `scala.LowPriorityImplicits` 类中，这个类是 `Predef` 的超类。子类和子对象中的隐式转换比基类的隐式转换优先级更高。因此，如果两个隐式转换同时可用，编译器会选择 `Predef` 中的那个。

5. 在 `Java` 中，你没有办法定义数组的泛型，即没办法写 `T[]`，其中 `T` 为类型参数。`scala` 中的数组支持泛型，因此可以写出像 `Array[T]` 的泛型数组。

   `scala` 将 `Array[T]` 存储的对象统一映射到 `AnyRef` ，从而支持 `byte,short,char,int,long,float,double,boolean` 等 `java` 基础类型，以及 `java` 对象类型。在运行时，当类型为 `Array[T]` 的数组的元素被访问或更新时，有一系列的类型检查来决定实际的数组类型。然后才是对 `java` 数组的正确操作。这些类型检查会减慢数组操作。可以预期，对泛型数组的访问跟基本类型数组或对象数组的访问相比会慢三到四倍。

   这意味着：如果你希望最大限度的改善性能，应该考虑具体类型的确定数组，而不是泛型数组。

6. 仅能够表示泛型的数组类型是不够的，我们还需要某种方式来创建泛型数组。为说明问题，我们考虑以下代码：

```
def evenElems[T](xs: Vector[T]): Array[T] = {
  val arr = new Array[T]((xs.length + 1) / 2)
  for(i <- 0 until xs.length by 2){
    arr(i/2) = xs(i)
  }
  arr
}
```

   这段代码返回输入 `xs` 的偶数位置元素组成的 `Array`。考虑到类型参数 `T` 对应的实际类型信息在运行时被擦除，因此 `new Array[T]((xs.length + 1) / 2)` 甚至无法编译通过。

   这时编译器需要提供额外的信息来获取类型参数 `T` 的运行时线索。这个线索的表现形式是类型为 `scala.reflect.ClassTag` 的类标签 `class tag` 。

   类标签描述的是给定类型 “被擦除的类型”，这也是构造该类型的数组所需要的全部信息。在很多情况下，编译器多可以自行生成类标签。对于具体类型 `Int` 或者 `String` 就是如此。但是对于某些泛型类型，如 `List[T]` 也是如此，因为有足够多的已知信息来预测被擦除的类型。

   对于完全泛化的场景，通常的做法是用上下文界定传入类型标签，如：

```
import scala.reflect.ClassTag
def evenElems[T: ClassTag](xs: Vector[T]): Array[T] = {
    ....
}
```

   此时，当 `Array[T]` 被创建时，编译器会查找类型参数 `T` 的类标签。也就是说，它会查找一个类型为 `ClassTag[T]` 的隐式值。如果找到这样的值，类标签就被用于构造正确类型的数组。否则，编译器报错。

七、List

1. `List` 是`Scala` 提供的列表，它是不可变对象，这与`java.util.List` 不同。

   `List` 的行为有些类似`Java` 的字符串：当你调用`List` 的某个方法，而这个方法的名字看起来像是会修改`List` 时，它实际上是创建并返回一个新的`List` 。

2. `List` 常用方法：

   - `list1:::list2`：拼接两个列表，返回新列表。
   - `val2::list1`：在`list1` 的头部添加一个新元素。
   - `Nil`：一个空列表对象。因此可以采用下面的方式初始化一个列表：

     ```
     val list = 1 :: 2 :: 3 :: Nil
     ```

   另外 `List` 也继承自 `Traversable --> Iterable --> Seq --> LinearSeq`，因此`List` 具有这些超类的所有方法。

3. 虽然 `List` 也提供了追加`append` 操作，记做`:+`，但该操作很少使用。因为向`List` 末尾追加元素的时间开销是 $O(n)$ ，n 为列表大小。而使用`::` 在列表头部添加元素只需要常量时间 $O(1)$ 。

   如果希望通过追加的方式构建列表，则有两种方式：

   - 通过`::` 依次在头部添加，最后通过`.reverse()` 方法来获取列表。
   - 采用`ListBuffer` ，它是一个可变列表，支持追加操作。构建完成后调用`.toList()` 方法即可。

4. 列表支持在头部快速添加、移除元素(复杂度 $O(1)$)，但是不支持快速访问下标（复杂的 $O(n)$)。这种特性意味着模式匹配很快。

7.1. 基本用法

1. `List` 字面量：

   ```
   val list = List(1,2,3,4)
   val list2 = List("a","b","c")
   ```

2. `List`列表 和 `Array` 很像，但是有两个重要区别：

   - 列表是不可变的，即：列表的内容一旦给定就无法改变。
   - 列表的结构是以链表的形式组织的，而 `Array` 的结构是一个数组。

3. 和 `Array` 一样，列表也是同构的：同一个列表的所有元素必须都是相同的类型。元素类型为 `T` 的列表的类型写作 `List[T]` 。

4. 列表类型是协变的。即：如果类型 `S` 是类型 `T` 的子类型，则 `List[S]` 是类型 `List[T]` 的子类型。

5. 空列表的类型为 `List[Nothing]`。由于 `Nothing` 是底类型，它是所有类型的子类。又由于 `List` 是协变的，因此对于任何 `T` 而言，`List[Nothing]` 是 `List[T]` 的子类型。

   这也是为什么我们可以写如下的代码：

   ```
   val list: List[String] = List() // List() 是 List[Nothing] 类型的对象
   ```

6. 所有的列表都构建自两个基本的构建单元：`Nil` 和 `::` （读作 `cons` ）。

`Nil` 表示空列表，中缀操作符 `::` 表示在列表前面追加元素。即 `x :: list` 表示这样的列表：第一个元素为 `x`，接下来是列表 `xs` 的全部元素。

因此列表也可以如下定义：

```
val list = 1 :: ( 2 :: ( 3 :: ( 4 :: Nil ) ) )
val empty = Nil
```

事实上 `List(...)` 的定义最终展开成上述方式。

由于以 `::` 以冒号结尾，因此它是右结合的，因此前面的定义中可以去掉括号，得到：

```
val list = 1 :: 2 :: 3 :: 4 :: Nil
```

7. 列表的基本操作：
   - `list.head`：返回列表的第一个元素。它只对非空列表有定义，对空列表调用时将抛出异常。
   - `list.tail`：返回列表的、除第一个元素之外的所有元素。它只对非空列表有定义，对空列表调用时将抛出异常。
   - `list.isEmpty`：返回列表是否为空。

   对列表的所有操作都可以用上述三个基础方法来表达。
8. 列表可以用模式匹配解开。列表模式可以逐一对应到列表表达式。

   既可以用 `List(...)` 这样的模式匹配来匹配列表的所有元素，也可以用 `::` 和 `Nil` 常量一点点的将列表解开。

   ```
   val list = List(1,2,3,4)
   val List(a,b,c,d) = list
   ```

   `List(a,b,c,d)` 模式匹配长度为 `4` 的列表，并将四个元素分别绑定到变量 `a,b,c,d` 。

   如果事先并不知道列表中元素的个数，或者列表中元素个数成千上万非常庞大，则更好的做法是使用 `::` 来匹配。

   ```
   val a :: b :: rest = list
   ```

   这种模式匹配的是长度大于等于 `2` 的列表。其中 `a` 匹配第一个元素、`b` 匹配第二个元素、`rest` 匹配列表剩下的部分。
9. 事实上，`List(...)` 是一个由类库定义的提取器 `extractor` 模式的实例。

   而 `a :: b :: rest` 作为模式时，`p op q` 这样的中缀操作等同于 `op(p,q)`，也就是中缀操作符 `op` 被当作构造方法模式处理的。具体而言，`x :: xs` 表达式相当于 `:: (x,xs)` 。

   一个细节是：存在名为 `scala.::` 类，该类与 `:: (x, xs)` 模式构造方法相对应。因此 `::` 在 `scala` 中出现两次：
   - 一次是作为 `List` 类的方法名，其作用是产出 `scala.::` 类的实例。
   - 另一次是作为 `scala` 包中的一个类的名字。
10. 使用模式匹配是用基本方法 `head,tail,isEmpty` 来解开列表的变通方式。通常对列表做模式匹配要比用方法来解构更加清晰。

7.2 初阶用法

1. 如果一个方法不接收任何函数作为入参，则该方法被称作初阶方法。
2. `:::` 操作符：拼接两个列表。

```
list1 ::: list2
```

和 `::` 操作符类似，`:::` 操作符也是右结合的。即 `xs ::: ys ::: zs` 等价于 `xs ::: (ys ::: zs)` 。

3. `.length` 方法：获取列表长度。

不同于数组，在列表上的 `length` 操作相对更耗资源。因为找到一个列表的末尾需要遍历整个列表，这需要消耗 $O(n)$ 的时间。

这也是为什么推荐使用 `xs.isEmpty` 测试，而不推荐使用 `xs.length == 0` 的原因。因为二者在结果上没有区别，但是后者的执行速度更慢，尤其当列表 `xs` 很长的时候。

4. `.last` 方法：获取（非空）列表的最后一个元素。
5. `.init` 方法：获取（非空）列表除了最后一个元素之外的剩余部分。
6. 和 `.head` 方法和 `.tail` 方法一样，`.last` / `.init` 这两个方法在应用到空列表上会抛出异常。

不像 `.head/.tail` 那样在运行时消耗 $O(1)$ 的常量时间，`.last` / `.init` 需要遍历整个列表来获取计算结果。因此它们的消耗时间复杂度为 $O(n)$ 。

7. 最好将列表组织成：大量访问都集中在列表头部而不是尾部，从而获取更高的访问效率。

如果需要频繁的访问列表的末尾，一个比较好的做法是：先将列表反转，然后变成了对反转后的列表的头部的访问。`.reverse` 方法就是反转列表，该方法会创建一个新的列表。

8. `.drop` 和 `.take` 方法是对 `.tail` 和 `.init` 的一般化，它们返回的是列表任意长度的前缀或后缀。
   - `xs take n` 返回列表 `xs` 的前 `n` 个元素；如果 `n >= xs.length`，则返回整个 `xs` 列表。
   - `xs drop n` 返回列表 `xs` 除了前 `n` 个元素的所有元素；如果 `n >= xs.length`，则返回空列表。
9. `splitAt` 方法将 列表从指定的下标位置切开，返回这两个列表组成的元组。该方法只需要遍历数组一次，而不是两次。

```
List("a", "b", "c", "d", "e")。splitAt(2)// 返回结果 （List("a", "b"), List("c", "d", "e"))
```

10. `apply` 用于从任意位置取元素，相对于数组而言，列表使用该方法比较少。它等价于 `xs(n)` 。

```
xs.apply(2) // 选择 xs 的下标为 2 的元素
```

该方法用的少的原因是 `xs(n)` 的时间复杂度为 $O(n)$ 。

事实上该方法是通过 `drop` 和 `head` 定义的：`xs.apply(n)` 等价于 `(xs drop n).head` 。

11. 列表的下标从 `0` 开始到列表长度减 `1` ，和数组一样。`.indices` 方法返回了列表的所有有效下标的序列。
12. `xs.flatten` 方法将 `xs` 扁平化，返回单个列表。其中 `xs` 是列表的列表。如果 `xs` 不满足条件，则编译错误。

13. `xs.zip(ys)` 方法返回一个由元组组成的列表。如果两个列表长度不同，则以最短的那个为主，没有配对上的元素被丢弃。

14. `xs.zipwithIndex` 方法返回一个元组组成的列表，元组的元素分别为原列表元素、该元素的下标。

    对该返回结果调用 `.unzip` 方法，则能够转换回列表组成的元组，第一个元素为原始列表 ，第二个元素为下标列表。

15. `.toString` 方法返回列表的标准字符串表现形式。

```
List(1,2,3,4).toString // 返回 "List(1,2,3,4)"
```

    如果希望采用不同的表现形式，则可以使用 `.mkString(pre，sep，post)` 方法。该方法首先添加前缀字符串`pre`，然后添加每个元素并在元素之间添加分隔符 `sep`，最后添加后缀字符串 `post` 。

```
List("1", "2", "3", "4").mkString("pre","#","post")// 返回 pre1#2#3#4post
```

    `.mkString`方法有两个重载的变种：

    - `.mkString(sep)`：它等价于 `.mkString("",sep,"")` 。
    - `.mkString()`：它等价于 `.mkString("","","")` 。

    `.mkString` 还有其它的变种，如 `.addString`，该方法并不是返回一个字符串，而是把构建出来的字符串追加到一个 `StringBuilder` 对象中。

```
val buf = new StringBuilder
List("a","b","c")。addString(buf,"pre",";","post")
```

    `.mkString` 和 `.addString` 这两个方法继承自 `List` 的超特质 `Tranversable`，因此它们也可以应用在所有其它集合类型上。

16. `List` 和 `Array` 的转换：

    - `list.toArray` 和 `array.toList` 方法

```
val arr = "abcde".toArray // Array(a,b, c, d, e) :  Array[Char]
arr.toList                //  List(a,b, c, d, e) :  List[Char]
```

    - `list.copyToArray` 方法可以将列表中的元素依次拷贝到目标数组的指定位置。

```
val arr2 = new Array[Int](10)
List(1,2,3)。copyToArray(arr2， 3)
println(arr2.mkString(",")) // 0,0,0,1,2,3,0,0,0,0
```

17. 通过迭代器访问列表元素：`list.iterator` 方法。

```
val it = List(1,2,3).iterator
println(it.next) // 1
println(it.next) // 2
println(it.next) // 3
println(it.next) // 抛出异常 :  java.util。NoSuchElementException
```

7.3 高阶方法

1. 高阶方法是接受函数为参数的方法。
2. 迭代方法：`map` 、 `flatMap` 、`foreach` ：
   - `xs.map(f)`：将类型为 `List[T]` 的列表 `xs` 的每个元素，通过函数 `f: T => U` 映射；再将映射结果拼接成列表返回。返回结果是 `List[U]` 类型的列表。

     ```
     List("this", "is", "a", "word").map(_.length) // 返回 ： List(4,2,1,4)
     ```

   - `xs.flatMap(f)`：和 `map` 类似，但是要求 `f` 返回一个列表 `List[U]`，然后将返回的所有列表的元素拼接成一个列表返回。返回结果是 `List[U]` 类型的列表。

     ```
     List("this","is").map(_.toList)// 返回 List(List(t, h, i,
     s),List(i,s)), 类型为 List[List[Char]]
     List("this","is").flatMap(_.toList)// 返回 List(t,h,i,s,i,s), 类型为
     List[Char]
     ```

   - `foreach` 方法：`xs.foreach(f)`，其中 `f` 返回结果为 `Unit` 。它简单的将 `f` 应用到列表中的每个元素，整个 `foreach` 方法本身的结果类型也是 `Unit` 。

     ```
     var sum = 0
     List(1,2,3).foreach(sum += _)
     println(sum) // 6
     ```

3. 过滤方法：`filter, partition, find, takeWhile, dropWhile, span`：
   - `xs.filter(f)` 方法：将 `f` 应用到 `xs` 的每个元素，保留 `f(x)` 为 `true` 的元素。其中 函数 `f` 的类型为：`f: T=> Boolean` 。

     ```
     List(1,2,3,4).filter(_ %2 ==0)// 返回 List(2, 4)
     ```

   - `xs.partition(f)` 方法：和 `.filter` 方法类似，但是它返回一对列表：其中一个包含所有 `f(x)=true` 的元素，另一个包含所有 `f(x)=false` 的元素。

     ```
     List(1,2,3,4).partition(_ %2 ==0) // 返回 (List(2,4),List(1,3))
     ```

   - `.find` 方法和 `.filter` 方法很像，但是它返回满足给定条件的第一个元素，而不是所有元素。

     `xs.find(f)` 返回一个可选值。如果 `xs` 中存在一个元素 `x` 满足 `f(x)=true`，则返回 `Some(x)`；否则返回 `None` 。

     ```
     List(1,2,3,4).find(_ %2 ==0) // 返回 Some(2)
     List(1,2,3,4).find( _ <= 0)  // 返回 None
     ```

   - `xs.takeWhile(f)` 返回列表 `xs` 中连续满足 `f(x)=true` 的最长前缀。

     ```
     List(1,2,3,-1,4).takeWhile(_ >= 0) // 返回 List(1,2,3)
     ```

而 `xs.dropWhile(f)` 去除列表 `xs` 中连续满足 `f(x)=true` 的最长前缀。

```
List(1,2,3,-1,4).dropWhile(_ >= 0) // 返回 List(-1,4)
```

而 `xs.span(f)` 将 `takeWhile` 和 `dropWhile` 合二为一，就像 `splitAt` 将 `take` 和 `drop` 合二为一一样。它返回一组列表：`xs.span(f)` 等价于 `(xs.takeWhile(f), xs.dropWhile(f))` 。

4. 列表判断：
   - `xs.forall(f)` ：如果列表 `xs` 中全部元素都满足 `f` 就返回 `true`，否则返回 `false` 。
   - `xs.exists(f)` ：如果列表 `xs` 中存在元素满足 `f` 就返回 `true`，否则返回 `false` 。

```
List( 1,2,3,4).forall( _ > 0 ) // 返回 true
List(-1,2,3,4).forall( _ > 0 ) // 返回 false
List(-1,2,3,4).exists( _ > 0 ) // 返回 true
```

5. 列表折叠：用某种操作符合并元素。如：

```
sum(List(1,2,3,4)) // 等于 1+2+3+4 = 10
product(List(1,2,3,4)) // 等于 1x2x3x4 = 24
```

   - 左折叠 `(z /: xs)(op)` ：以 `z` 为前缀，对列表元素以此连续应用 `op` 操作。

```
(z /: List(a,b,c))(op) // 等于 op(op(op(z,a),b),c)
//              op
//             /   \
//            op    c
//           /   \
//          op    b
//         / \
//        z   a
```

     左折叠会产生一颗向左的操作树。它从左向右折叠。

   - 右折叠 `(xs :\ z)(op)` ：以 `z` 为后缀，对列表元素以此连续应用 `op` 操作。

```
 (List(a,b,c) :\ z)(op) // 等于 op(a,op(b,op(c,z)))
//              op
//             /   \
//            a     op
//                 /   \
//                b     op
//                     / \
//                    c   z
```

     右折叠会产生一颗向右的操作树。它从右向左折叠。

左折叠和右折叠的效率是相同的，不存在执行效率上的差异。

你也可以使用 `foldLeft` 和 `foldRight` 这两个方法名，它是定义在 `List` 类中的方法。

6. 列表拼接 `xs ::: ys` 的执行时间和第一个参数 `xs` 的长度成正比。因为需要找到 `xs` 的尾指针指向 `ys` 的头指针。获取尾指针的算法复杂度为 $O(n)$、获取头指针的算法复杂度为 $O(1)$ 。
7. 因为类型擦除， 所以 `Scala` 类型推断不能自动推断出正确的列表类型。
8. `xs.sortWith(before)` 用于对列表 `xs` 中的元素进行排序，其中 `before` 是一个用来比较两个元素的函数。

   `before(x,y)`：如果 `x` 应该排在 `y` 的前面，则返回 `true`。

7.4 List 伴生对象方法

1. 前面介绍的所有操作都是 `List` 类的方法，因此我们在每个具体的 `List` 对象上调用它们。还有一些方法是定义在全局可访问的 `scala.List` 上的，这是 `List` 的伴生对象。

   有一些操作是用于创建列表的工厂方法，另一些是对特定形状的列表进行操作。
2. `List.apply`：从给定元素创建列表。

   `List(1,2,3)` 仅仅是 `List.apply(1,2,3)` 的便利方式，二者等效。

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

3. `List.range`：从数值区间创建列表。
   - `List.range(from,until)`：创建一个包含了从 `from` 开始递增到 `until-1` 的数的列表。结果不包含 `until` 。
   - `List.range(from,until,step)`：创建一个包含从 `from` 开始、步长为 `step`、不超过 `until` 的数的列表。步长可以为正，也可以为负。

```
List.range(1,5)
List.range(1,-5,-1)
```

4. `List.fill`：创建包含零个或者多个相同元素拷贝的列表。它接收两个参数：要创建的列表长度、需要重复的元素。两个参数各以不同的参数列表给出。

```
List.fill(3)("a")
```

   如果给 `fill` 的参数多于 `1`个，那么它会创建多维列表，如：列表的列表、列表的列表的列表。多出来的参数放在第一个参数列表中。

```
List.fill(2,3)("a") // List(List("a","a","a"),List("a","a","a"))
```

5. `List.tabulate`：根据给定的函数计算列表的元素。它也有两个参数列表，第一个参数列表给出列表的维度，第二个参数描述列表的元素。它和 `List.fill` 区别在于：`List.tabulate` 的元素值不再固定，而是根据函数计算而来，函数参数为被计算的元素的所处的位置（索引）。

```
List.tabulate(5)(n => n*n) // List(0,1,4,9,16)
List.tabulate(5,5)( - * _)
/*
List(
  List(0,0,0,0,0)
  List(0,1,2,3,4)
  List(0,2,4,6,8)
  List(0,3,5,9,12)
  List(0,4,6,12,16)
)
*/
```

6. `List.concat`：拼接多个列表。

```
List.concat(List("a","b","c"),List("d")) // List("a","b","c","d")
```

7.5 同时处理多个列表

1. 可以通过对多个列表组成的元组调用 `zipped` 方法，然后执行 `map/exists/forall` 等操作，就可以同时处理多个列表。

```
(List(1,2,3,4), List(11,12,13),List("a","b","c","d","e")).zipped.foreach{
  case (item1:Int,item2:Int,item3:String)
    => println("%d,%d,%s".format(item1,item2,item3))
}
```

注意：`zipped` 方法把所有列表中同一个位置的元素 `zip` 在一起，以最短的列表为基准。

八、ListBuffer/ArrayBuffer

8.1 ListBuffer

1. `ListBuffer` 是一个可变对象（包含在 `scala.collection.mutable` 包中），它帮助我们在需要追加元素来构建列表时更加高效。`ListBuffer` 提供了常量时间的向后追加和向前追加的操作：
   - `+=` 操作符提供向后追加元素功能。
   - `+=:` 操作符提供向前追加元素功能。

   构建完以后，可以调用 `ListBuffer` 的 `toList` 方法来获取最终的 `List`。
2. 使用 `ListBuffer` 而不是 `List` 有两个原因：
   - 希望通过在列表尾部追加元素的方式构建列表。因为向列表头部追加元素效率较高，而尾部追加元素效率较低。
   - 防止可能出现的栈溢出。当递归的在列表头部追加元素，但是递归并非尾递归时，可能出现栈溢出。
3. 所有 `List` 操作在 `ListBuffer` 上都可用。

8.2 ArrayBuffer

1. `ArrayBuffer` 和数组很像，除了可以额外从序列头部和尾部添加或移除元素之外。

2. 所有`Array`操作在`ArrayBuffer`上都可用，但是由于实现的封装，`ArrayBuffer`会慢一些：添加新元素和移除旧元素操作平均而言是常量时间，不过偶尔需要线性时间。这是因为其实现需要不时的分配新的数组来保存缓存的内容。
3. 在创建`ArrayBuffer`时，必须给出类型参数，不过并不需要指定长度。`ArrayBuffer`在需要时自动调整分配的空间。
4. 可以用`+=`方法向`ArrayBuffer`追加元素。

九、Set

1. `Set`是没有重复元素的`Iterable`。`Scala`提供了`scala.collection.immutable.Set`和`scala.collection.mutable.Set`这两个`trait`来分别表示不可变集和可变集。

   这里我们称`set`为"集"。

2. 可以对`Set`调用`+`方法来添加元素。

   - 对于不可变集，该方法返回一个新的、包含了新元素的集。
   - 对于可变集，该方法先将元素添加到集中，然后返回集本身。

3. `Scala collection`类库同时提供了可变和不可变两个版本的`Set`和`Map`。当写下`Set`或`Map`时，默认为不可变的版本。`Scala`让我们更容易访问不可变版本，这是为了鼓励大家尽量使用不可变的`Set/Map`。

   这种访问便利是通过`Predef`对象完成的，这个对象的内容在每个`Scala`源文件都会隐式导入：

   ```
   object Predef{
     type Map[A, +B] = collection.immutable.Map[A, B]
     type Set[A] = collection.immutable.Set[A]
     val Map = collection.immutable.Map
     val Set = collection.immutable.Set
     // ...
   }
   ```

   `Predef`利用`Type`关键字定义了`Set`和`Map`这两个别名，分别对应不可变`Set`和不可变`Map`的完整名称。

   如果希望使用可变版本，则需要显式导入：

   ```
   import scala.collection.mutable
   val set1 = mutable.Set("1", "2")  // 可变版本
   set1 += "3" // 可以是 val，也可以是 var

   var set2 = Set("1", "2") // 不可变版本
   set2 += "3" // 必须是 var
   ```

   - `set1`是可变的`Set[String]`，经过`+=`方法调用之后，`set1`仍然指向旧的`Set`。
   - `set2`是不可变的`Set[String]`，经过`+=`方法调用之后，`set2`指向新创建的`Set`。因此`set2`必须是`var`。

4. `Set`的关键特性是：以`==`为标准，同一时刻，`Set`内的每个对象最多出现一次。

5. `Set`特质的操作：

   - 测试：

     - `xs contains x`：测试`x`是否为`xs`的元素。

    - `xs(x)`： 等价于 `xs contains x` 。
    - `xs subsetOf ys`： 测试 `xs` 是否为 `ys` 的子集。
  - 添加：
    - `xs + x`： 返回包含 `xs` 所有元素以及 `x` 的新 `Set`。
    - `xs + (x,y,z)`： 返回包含 `xs` 所有元素以及 `x,y,z` 的新 `Set`。
    - `xs ++ ys`： 返回包含 `xs` 和 `ys` 所有元素的新 `Set` 。
  - 移除：
    - `xs - x`： 返回包含除 `x` 外 `xs` 所有元素的新 `Set` 。
    - `xs - (x,y,z)`： 返回包含除 `x,y,z` 外 `xs` 所有元素的新 `Set` 。
    - `xs -- ys`： 返回包含除 `ys` 元素之外 `xs` 所有元素的新 `Set`。
    - `xs.empty`： 返回跟 `xs` 类型相同的空 `Set` 。
  - 二元操作：
    - `xs & ys`： 返回 `xs` 和 `ys` 交集。
    - `xs | ys`： 返回 `xs` 和 `ys` 并集。
    - `xs union ys`： 等价于 `xs | ys` 。
    - `xs &~ ys`： 返回 `xs` 和 `ys` 差集。
    - `xs diff ys`： 等价于 `xs &~ ys` 。
6. 可变的 `mutable.Set` 特质的操作：
  - 添加：
    - `xs += x`： 将 `x` 添加到 `xs` 并返回 `xs` 本身。
    - `xs += (x,y,z)`： 将 `x,y,z` 添加到 `xs` 并返回 `xs` 本身。
    - `xs ++= ys`： 将 `ys` 所有元素添加到 `xs` 并返回 `xs` 本身。
    - `xs add x`： 将元素 `x` 添加到 `xs` 中。 如果 `x` 此前并未包含在 `xs` 中，则返回 `true`；否则返回 `false` 。
  - 移除：
    - `xs -= x`： 将 `x` 从 `xs` 中移除并返回 `xs` 本身。
    - `xs -= (x,y,z)`： 将 `x,y,z` 从 `xs` 中移除并返回 `xs` 本身。
    - `xs -- = ys`： 将 `ys` 所有元素从 `xs` 中移除并返回 `xs` 本身。
    - `xs remove x`： 将元素 `x` 从 `xs` 中移除。 如果 `x` 此前包含在 `xs` 中， 则返回 `true`；否则返回 `false` 。
    - `xs retrain p`： 仅保留 `xs` 中那些满足条件 `p` 的元素。
    - `xs.clear()`： 从 `xs` 中移除所有元素。
  - 更新：
    - `xs(x) = b`： 如果布尔参数 `b` 为 `true` ，则将 `x` 添加到 `xs`；否则将 `x` 从 `xs` 移除。
    - `xs.update(x, b)`： 等价于 `xs(x) = b` 。
  - 拷贝：
    - `xs.clone`： 返回与 `xs` 有相同元素的新的可变 `Set` 。
7. `Set` 的交、并、差集有两种形式： 字母的和符号的。 字母的版本有 `intersect、union、diff` ， 符号的版本有 `& 、| 、&~` 。

   `Set` 从 `Traversable` 继承的 `++` 可以视为 `union` 或者 `|` 的一个别名， 只不过 `++` 接受 `Traversable` 类型的参数， 而 `union` 和 `|` 接受 `Set` 类型的参数。
8. `Set` 的 `.apply` 方法等价于 `.contains` 方法，因此 `Set` 可以被用做测试函数：对于那些它包含的元素返回 `true` 。

```
val s = Set[Int](1,3,5,7,9)
s(1) // 返回 true
```

9. 通常不可变 `Set` 的 `+, ++, -, --` 等操作用的不多， 因为它们会拷贝整个 `Set` 来构建一个新的 `Set`。相比之下， 可变 `Set` 的 `+=, ++=, -=, --=` 使用得更多。

   对于不可变 `Set` ， `+=` 等操作可以配合 `var` 使用：

   ```
   var s  = Set(1,2,3)
   s += 4 // 新的 Set (1,2,3,4)
   s -= 2 // 新的 Set(1,3,4)

   val s2 = mutable.Set(1,2,3)
   s2 += 4 // 旧的 Set(1,2,3,4)
   s2 -= 2 // 旧的 Set(1,3,4)
   ```

   因此可以看到有一个重要原则： 可以用一个 `var` 的不可变 `Set` 替换一个 `val` 的可变 `Set` ，或者相反。

10. 可变 `Set` 还提供了 `add/remove` 作为 `+=` 和 `-=` 的变种。区别在于： 前者返回的是表示该 `Set` 是否发生改变的布尔值结果。

11. 目前可变 `Set` 默认实现采用了使用 `Hash` 表来保存 `Set` 的元素。不可变 `Set` 默认实现使用了一个跟 `Set` 元素数量相匹配的底层表示：

    - 空 `Set` 为单例对象。
    - 四个元素以内的 `Set` 也是由单个以字段保存所有元素的对象。
    - 超出四个元素的不可变 `Set` 实现为哈希字典树 `hash trie` 。

    因此， 对于四个元素以内的小型 `Set` 而言， 不可变 `Set` 比可变 `Set` 更加紧凑和高效。 因此，如果你预期用到的 `Set` 比较小， 则尽量使用不可变 `Set` 。

12. 创建空 `Set` 的方法：

    ```
    Set.empty         // 不可变集
    mutable.Set.empty // 可变集
    ```

13. 初始化 `collection` 最常见方法是将初始元素传入对应 `collection` 的伴生对象的工厂方法。编译器将会将其转换成对伴生对象的 `apply` 方法的调用。

    ```
    List(1,2,3) // 等价于 List.apply(1,2,3)
    ```

    你也可以显示的指定类型：

    ```
    List[Long](1,2,3)
    ```

    如果希望用另一种类型的 `collection` 来初始化，则可以使用 `++` 操作符：

    ```
    val treeSet1 = TreeSet[Long]() ++ List[Long](1,2,3) // 类型为 TreeSet[Long]
    val treeSet2 = TreeSet[Long]() ++ List(1,2,3)       // 类型为 TreeSet[Any]
    ```

14. 如果希望将其它 `collection` 转换为 `Array`，则直接调用其 `.toArray` 方法；如果希望将其它 `collection` 转换为 `List`，则直接调用其 `.toList` 方法。

- 调用 `toList/toArray` 方法时会进行所有元素的拷贝，因此对于大型 `collection` 来说可能比较费时。
- 对于 `Set/Map` 等 `collection` 来说，将它们转化为`Array/List` 时，转换后的元素顺序等于原始`collection` 进行迭代的顺序。

  如：`TreeSet[Long]` 的迭代器会按照数字大小的顺序产生元素，因此对 `TreeSet[Long]` 调用 `toList` 是按照大小有序排列的列表。

十、Map

1. `Map` 是由键值对组成的 `Iterable` 。 `scala` 的 `Predef` 类提供了一个隐式转化， 让我们可以用 `key -> value` 这样的写法来表示 `(key, value)` 这样的 `pair` 。 因此 `Map ( "x" -> 1, "y" -> 2)` 和 `Map(("x",1), ("y",2))` 的含义完全 相同， 但是前者更易读。
2. 与`Set` 类似， `Scala` 也提供了可变、不可变版本的`Map` 。
   `Set(),mutable.Set(),Map(),mutable.Map()` 等工厂方法返回的`Set` 和 `Map` 其实都是不同的类型。
   - 对于可变的 `mutable.Set()` 工厂方法，它返回的是`mutable.HashSet` 对象。
   - 对于可变的 `mutable.Map()` 工厂方法，它返回的是 `mutable.HashMap` 对象。
   - 不可变的 `Set()` 工厂方法返回的类型取决于我们传入多少个元素。对于少于五个元素的 `Set`，有专门特定大小的类与之对应从而达到最佳性能；大于等于五个元素的 `Set` 返回的是 `HashSet` 类型。
     - 零个元素：`immutable.EmptySet` 。
     - 一个元素：`immutable.Set1` 。
     - 二个元素：`immutable.Set2` 。
     - 三个元素：`immutable.Set3` 。
     - 四个元素：`immutable.Set4` 。
     - 五个或更多元素：`immutable.HashSet`。
   - 同理，不可变的 `Map()` 工厂方法返回的类型取决于我们传入多少个元素。对于少于五个元素的`Map`，有专门特定大小的类与之对应从而达到最佳性能；大于等于五个元素的 `Map` 返回的是 `HashMap` 类型。
     - 零个：`immutable.EmptyMap` 。
     - 一个：`immutable.Map1.` 。
     - 二个：`immutable.Map2` 。
     - 三个：`immutable.Map3` 。
     - 四个：`immutable.Map4` 。
     - 五个或更多：`immutable.HashMap`。

   默认的不可变实现可以提供最佳性能。另外，如果添加一个元素到 `EmptySet`，则返回一个 `Set1` 对象。如果从 `Set1` 对象移除一个元素，则得到一个 `EmptySet` 对象。
3. 可以通过 `+=` 和 `->` 来向`Map` 中添加键值对。

```
import scala.collection.mutable
val map1 = mutable.Map[Int,String]()
map1 += (1 ->"1")
map1 += (2 ->"2")
```

   - `1 -> "1"` 转换成标准的方法调用，即`1.->("1")`。

你可以在任何对象上调用 `->` 方法，它返回的是一个二元元组。

- `+=` 将调用 `map1` 对象的 `.+=()` 方法，向 `map1` 中添加键值对。

4. `Map` 特质和操作和 `Set` 类似：
   - 查找：
     - `ms get k`： 以 `Option` 表示的、 `ms` 中跟键 `k` 关联的值， 如果没有对应的 `k` 则返回 `None` 。
     - `ms(k)`： 返回 `ms` 中跟键 `k` 关联的值， 如果没有对应的 `k` 则抛出异常。
     - `ms apply k`： 等价于 `ms(k)`。
     - `ms getOrElse(k,d)`： 返回 `ms` 中跟键 `k` 关联的值， 如果没有对应的 `k` 则返回 `d` 。
     - `ms contains k`： 测试 `ms` 中是否包含键 `k`。
     - `ms isDefinedAt k`： 等价于 `ms contains k` 。
   - 添加和更新：
     - `ms + (k -> v)`： 创建一个包含 `ms` 和给定 `k -> v` 键值对的新的 `Map`。
     - `ms + (k -> v, w-> u)`： 创建一个包含 `ms` 和给定 `k -> v， w -> u` 键值对的新的 `Map` 。
     - `ms + kvs`： 创建一个包含 `ms` 和给定 `kvs` 包含的所有键值对的新的 `Map` 。
     - `ms updated(k,v)`： 等价于 `ms + (k -> v)`
   - 移除：
     - `ms - k`： 创建一个包含 `ms` 所有映射关系、但不包含键 `k` 的新的 `Map` 。
     - `ms - (k,l,m)`： 创建一个包含 `ms` 所有映射关系、但不包含键 `k,l,m` 的新的 `Map` 。
     - `ms -- ks`： 创建一个包含 `ms` 所有映射关系、但不包含 `ks` 中所有键的新的 `Map` 。
   - 子集合：
     - `ms.keys`： 返回 `ms` 中所有键的 `Iterable` 。
     - `ms.keySet`： 返回 `ms` 中所有键的 `Set`。
     - `ms.keysIterator`： 返回一个迭代器， 该迭代器迭代时交出 `ms` 中每个键。
     - `ms.values`： 返回 `ms` 中所有值的 `Iterable`。
     - `ms.valuesIterator`： 返回一个迭代器， 该迭代器迭代时交出 `ms` 中每个值。
   - 变换 `transform`：
     - `ms filterKeys p`： 只包含 `ms` 中那些键满足条件 `p` 的映射关系的 `Map` 视图。
     - `ms mapValues f`： 通过对 `ms` 中每个 `value` 应用函数 `f` 得到的 `Map` 视图。
5. `mutable.Map` 特质的操作：
   - 添加和更新：
     - `ms(k) = v`： 以副作用将 `k -> v` 映射关系添加到 `ms`， 或者更新 `ms` 的键 `k` 对应的值。 也可以写作 `ms.update(k, v)`。
     - `ms += (k -> v)`： 以副作用的方式将 `k -> v` 映射关系添加到 `ms`， 返回 `ms` 本身。
     - `ms += (k1 -> v1, k2 -> v2)`： 以副作用的方式将 `k1 -> v1, k2 -> v2` 映射关系添加到 `ms`， 返回 `ms` 本身。
     - `ms ++= kvs`： 以副作用的方式将 `kvs` 中的映射关系添加到 `ms`， 并返回 `ms` 本身。
     - `ms put (k, v)`： 将 `k -> v` 映射关系添加到 `ms` 中， 并以 `Option` 的方式返回之前与 `k` 关联的值。

        - `ms getOrElseUpdate(k, d)`： 如果键 `k` 在 `ms` 中有定义， 则返回关联的值。否则用 `k -> d` 更新 `ms` 并返回 `d` 。
    - 移除:
        - `ms -= k`： 以副作用的方式移除键 `k` 的映射关系， 并返回 `ms` 本身。
        - `ms -= (l, k, m)`： 以副作用的方式从 `ms` 移除给定键的映射关系， 并返回 `ms` 本身。
        - `ms --= ks`： 以副作用的方式从 `ms` 移除 `ks` 中所有键的映射关系， 并返回 `ms` 本身。
        - `ms remove k`： 从 `ms` 中移除键 `k` 的映射关系并以 `Option` 的形式返回键 `k` 之前关联的值。
        - `ms retrain p`： 仅保留 `ms` 中那些键满足条件 `p` 的映射关系。
        - `ms.clear()`： 从 `ms` 中移除所有的映射关系。
    - 变换和克隆
        - `ms transform f`： 用函数 `f` 变化 `ms` 中所有关联的 `key-value` 的 `pair` 对。 `f` 的输入包括 `key` 和 `value` 。
        - `ms.clone`： 返回跟 `ms` 包含相同映射关系的新的可变映射。
6. 和 `Set` 一样， `Map` 的添加和移除操作涉及到对象的拷贝， 因此不常用。相比之下， `mutable.Map` 的添加和移除使用得更多。
7. `mutable.Map.getOrElseUpdate` 适合用作对 `cache` 的映射的访问:

```
def f(x :String) :String = {
  .... // 这里需要经历一个开销巨大的计算， 如连接数据库或者发送网络请求
}
val cache = collection.mutable.Map[String, String]()
def cachedF(s : String) = cache.getOrElseUpdate(s, f(s))
```

   注意： `getOrElseUpdate` 方法的第二个参数是传名 `by-name` 的， 因此只有当 `getOrElseUpdate` 需要第二个参数的值时， `f(s)` 的计算才会真正执行。
8. 有时候我们需要迭代器按照特定顺序迭代的 `Set` 或 `Map`，对此 `Scala` 提供了 `SortedSet` 和 `SortedMap` 这两个 `Trait` 。这些 `Trait` 被 `TreeSet` 和 `TreeMap` 类实现，这些实现采用红黑树来保持元素（对于 `TreeSet` ）或者键（对于 `Map` ）的顺序，具体顺序由 `Ordered Trait` 决定， `Set` 元素的类型或者 `Map` 键的类型必须都混入了 `Ordered Trait` 或者能够被隐式转换为 `Ordered` 。

   这两个类只有不可变的版本。

```
import scala.collection.immutable.{TreeSet, TreeMap}
val ts = TreeSet(1,9,2,4,8)
val tm = TreeMap(1 -> "a",9 -> "b", 2 -> "c")
```

9. 可变集合类型和不可变集合类型的选择:
    - 通常首选采用不可变集，因为不可变集合容易理解和使用。
    - 在元素不多的情况下，不可变集合通常比可变集合更紧凑。
        - 一个空的可变 `Map`，如果按照默认的 `HashMap` 实现，会占用 `80` 字节，每增加一项需要额外的 `1.6` 字节。
        - 而一个空的不可变 `Map` 是个单例对象，它被所有引用共享，因此使用它本质上只需要一个指针的大小。

因此对于小型的 `Map` 和 `Set`，不可变版本比可变版本紧凑的多，可以节省大量空间，并带来重要的性能优势。

- `Scala` 提供了一些语法糖来让不可变 `Set` 和可变 `Set` 的切换更加容易。

  如：虽然不可变 `Set` 不支持 `+=` 操作，但是 `Scala` 提供了一个变通途径：只要看到 `a += b` 而 `a` 并不支持 `+=` 方法，则 `Scala` 尝试将其解读为 `a = a + b`。此时 `a` 必须被声明为 `var` 而不是 `val` 。

```
val a = Set(1,2,3)
var c = Set(1,2,3)
a + = 4 // 不支持
c + = 4 // 等价于 c = c + 4， c 指向一个新的 Set
```

同样的思想适合所有以 `=` 结尾的方法。 这时候只需要将 `Set` 替换为 `mutable.Set` 就能够用很小的代价来替换到可变 `Set` 的版本。

这样的特殊语法不仅适合于 `Set`，它也适合于 `Map` 、以及所有的值类型。

十一、String&StringOps

1. 和数组一样，字符串也不是序列但是可以隐式转换为序列，因此可以支持所有序列操作。
   - 第一个优先级较低的隐式转换为：将 `String` 转换为 `WrappedString` ，它是 `immutable.IndexedSeq` 的子类。
   - 第二个优先级较高的隐式转换为：将 `String` 转换为 `StringOps`，它包含所有 `Seq` 的方法。

     `StringOps` 是一个序列，它实现了很多序列的方法。由于 `Predef` 有一个从 `String` 到 `StringOps` 的隐式转换，因此可以将任何字符串当作序列来处理。

```
val str = "hello"
str.reverse             // 发生隐式转换 String -> StringOps
val s:Seq[Char] = str   // 发生隐式转换 String -> WrappedString， s 类型为 WrappedString
"abc".exists(_.isUpper) // 发生隐式转换 String -> WrappedString
```

十二、元组

1. 元组也是 `Scala` 提供的不可变对象。与列表不同的是：元组可以容纳不同类型的元素。由于元组可以将不同类型的对象组合起来，因此它并不继承自 `Traversable` 。

   当你需要从方法中返回不同类型的多个对象时，元组非常有用。元组的一个常见应用场景是从方法返回多个值，而且这些值是不同类型的。

2. 实例化一个新的元组非常简单，只需要将对象放在圆括号当中，并用逗号隔开即可。

```
val tuple1 = (99, "Hello")
```

一旦实例化好一个元组，则可以用英文句点 `.`、下划线、和从 `1` 开始的序号来访问每个元素：

```
println(tuple1._1) // 99
println(tuple1._2) // "Hello"
```

- 元组的访问序号从`1`开始，而不是从`0`开始。
- `Scala`可以自动推断元组的类型，如`tuple1`的类型为`Tuple2[Int,String]`。

  元组的实际类型不仅取决于元素的类型，还取决于元素的数量。

3. 元组的访问与`List`不同，其背后原因是：列表的`.apply()`方法永远只会返回一种类型，而元组里的元素可以是不同类型的。
4. 尽管理论上可以创建任意长度的元组，但是目前`Scala`标准类库仅仅支持定义`22`个长度的元组`Tuple22`。
5. 可以将元组的元素分别赋值给不同的变量。

```
val t1 = ("It is a string", 13)
val (word, idx) = t1 // word 为字符串 "It is a string" ;  idx 为整数 13
```

如果去掉括号将得到不同的结果，每个变量都通过等号右侧的表达式求值来初始化：

```
val word,idx = t1
// word 为元组 ("It is a string", 13)
// idx 为元组 ("It is a string", 13)
```

十三、其它不可变集合

1. `Scala`提供了很多具体的不可变集合类，它们实现的特质各不相同（`Map,Set,Sequence`），可以是无限的也可以是有限的。
2. `List`是有限的不可变序列。它提供 $O(1)$ 时间的对首个元素和剩余元素的访问，以及 $O(1)$ 时间的在列表头部新增元素的操作。其它很多操作都是 $O(n)$ 时间的。
3. `Stream`和列表很像，不过其元素都是惰性的。因此，`Stream`是无限长的，只有被请求到的元素才会被计算。除此之外，流的特性和`List`是一样的。
   - 列表通过`::`操作符构造，而流是通过`#::`来构造：

     ```
     val str = 1 #:: 2 #:: 3 #:: Stream.emtpy
     ```

     流的基本要求是惰性计算，因此它的`toString`方法并不会强制任何额外的求值计算，只会给出头部的结果。如上面的`str`的`.toString`为`Stream(1. ?)`。
   - 下面是一个更复杂的例子：

     ```
     def fibFrom(a: Int,  b: Int): Stream[Int] =  a #:: fibFrom(b, a + b)
     ```

     需要说明的是：计算`Fibonacci`序列的同时不会引发无限递归。如果函数使用了`::`而不是`#::`，那么每次对`fibFrom`的调用都会引发另一个调用，这样就会造成无限递归。不过由于它用的是`#::`，因此表达式右边只会在被请求时才会被求值。如：

     ```
     val fibs = fibFrom(1, 1).take(7) // 此时 fibs 还没有全部求值
     fibs.toList // 此时 fibs 每个元素求值
     ```

4. `Vector`是对头部之外的其它元素也提供 $O(1)$ 复杂度访问的集合类型。但是，这个常量时间是“从效果上讲的常量时间”。

   向量的创建和修改和其它序列没有什么不同：

```
val vec = Vector.empty
val vec2 = vec :+ 1 :+ 2
val vec3 = 100 +: vec2
```

- 向量的内部结构是宽而浅的树，树的每个顶点包含多达 `32` 个元素、或 `32` 个子节点。
  - 小于等于 `32` 个元素的向量可以用单个结点来表示。
  - 小于等于 `32 x 32` = `1024` 个元素的向量可以通过单次额外的间接跳转来访问。
- 如果我们允许从根部到叶子最多`2`跳，则向量可以包含多达 2^{15} 个元素；如果允许最多 `3` 跳，则向量可以包含多达 2^{25} 个元素。如果允许最多 `5` 跳，则向量可以包含多达 2^{30} 个元素。

  因此对于所有正常大小的向量，选择一个元素只需要最多 `5` 次基本的数组操作。这就是我们说的"从效果上讲的常量时间"。
- 向量是不可变的，因此无法原地修改向量元素的值。不过，通过 `updated` 方法可以创建一个与给定向量在单个元素上有差异的新向量：

  ```
  val vec = Vector(1,2,3)
  val vec2 = vec updated(2,4) // Vector(1,2,4)
  ```

  和选择操作一样，函数式向量的更新也是消耗"从效果上讲的常量时间"：更新向量中的某个元素可以通过拷贝包含该元素的结点、以及从根部开始到该结点路径上的所有结点，其它结点无需拷贝。

  这意味着一次函数式的向量更新只会创建出 `1~5` 个新的结点，其中每个结点包含 `32` 个元素或者子节点。这比可变数组的原地更新要更加昂贵，但是比起拷贝整个向量而言要便宜的多。

由于向量在快速的任意位置的索引，以及任意位置的函数式更新之间取得了较好的平衡，因此它们目前是不可变的带下标索引的序列 `IndexedSeq` 的默认实现。

5. `immutable.Stack` 是不可变的栈。可以使用 `push` 来压一个元素入栈；可以用 `pop` 来弹一个元素出栈；可以用 `top` 来查看栈顶的元素而不是将它弹出。所有这些操作都是常量时间。

   ```
   val stack = scala.collection.immutable.Stack.empty
   val stack1 = stack.push(1)
   ```

   不可变的栈在 `Scala` 中很少用到，因为它们的功能完全被 `List` 包含。对不可变的栈的 `push` 操作对应于 `List` 的 `::` 操作，对不可变的栈的 `pop` 操作对应于 `List` 的 `tail` 操作。

6. `immutable.Queue` 是不可变的队列。可以用 `enqueue` 来追加一个元素或者一组元素，可以用 `dequeue` 来移除头部的元素（出队列）：

   ```
   val queue = scala.collection.immutable.Queue
   val queue1 = queue.enqueue(1)
   val queue2 = queue.enqueue(List(1,2,3))
   val (ele,queue3) = queue.dequeue
   ```

   注意：`dequeue` 返回的是一个包含被移除元素以及队列剩余部分的 `tuple` 。

7. `immutable.Range` 是一个有序的整数序列，整数之间有相同的间隔。在 `Scala` 中创建 `Range` 的方法是 `to` 和 `by`：

```
val range1  = 1 to 3 // Range(1,2,3)
val range2 = 5 to 14 by 3 // Rnage(5,8,11,14)
```

其中 `to` 指定 `Range` 的起止位置， `by` 指定 `Range` 间隔。

如果需要创建的 `Range` 不包含上限，则可以用 `until` 而不是 `to`：

```
val range3 = 1 until 3 // Range(1,2)
```

`Range` 的内部表示占据了 $O(1)$ 的空间，因为它只需要用三个数来表示：开始位置、结束位置、步长。因此大多数 `Range` 操作非常快。

8. 哈希字典树 `Hash Trie`：哈希字典树是实现高效的不可变 `Set` 和不可变 `Map` 的标准方式。它们的内部表现形式和向量类似，也是每个结点有 `32` 个元素或 `32` 棵子树的树，不过元素选择是基于哈希码的（而不是指针）。

   举例来说：要想找到 `Map` 中给定的键，首先用键的哈希码的最低 `5` 位找到第一棵子树、然后用哈希码的下一个 `5` 位找到第二棵子树，依次类推。当某个结点所有元素的哈希码（已用到部分）各不相同时，该选择过程就停止了。

   哈希字典树在快速的查找、高效的函数式插入 `+` 、高效的函数式删除 `-` 之间取得了不错的平衡。这也是为什么它是不可变 `Map` 和不可变 `Set` 默认的实现的原因。

   实际上， `Scala` 对于元素少于 `5` 个的不可变 `Set` 和不可变 `Map` 还有进一步的优化：

   - 带有 `k` （`1<=k <=4`）个元素的 `Set` 和 `Map` 采用的是包含对应 `k` 个字段的单个对象，每个元素对应一个字段。
   - 空的 `Set` 和 `Map` 使用单例对象来表示。

9. 红黑树：红黑树是一种平衡二叉树，某些结点标记为"红色"、某些结点标记为"黑色"。和其它平衡二叉树一样，对它们的操作可以可靠地在 $O(\log n)$ 时间内完成。

   `Scala` 的 `TreeSet, TreeMap` 的内部使用红黑树来实现，红黑树也是 `SortedSet` 的标准实现。

   ```
   val set = collection.immutable.TreeSet.empty[Int]
   set +1 +3 +3
   ```

10. 不可变位组 `bit set` ： 位组 `bit set` 用于表示某个非负整数的 `bit` 的集合。从内部讲，位组使用一个 `Array[Long]` 来表示。第一个 `Long` （`64位`）表示整数 `0~63` ，第二个 `Long` 表示整数 `64~127` ，以此类推。

    如果集合中最大整数在数百这个量级，则位组非常紧凑；如果集合中最大整数非常大，比如 `100万` ，则它需要 `1.00万/64` 大约 `1.6` 万个 `Long` 来表示。

    对位组的操作非常快：测试某个整数是否在位组中只需要 $O(1)$ 的时间；向位组中添加整数的时间复杂度和底层 `Array[Long]` 的长度成正比，而这个长度通常很小。

    ```
    val bitSet = scala.collection.immutable.BitSet.empty
    val bitSet2 = bitSet ++ List(3,4,4,0)
    // 内部表示为 Array[Long](25)。 3,4,4,0 用 bit 序列表示为 10011，我们从左到右某个 bit 为 1 代表该位置下标对应的整数出现。 10011 转换为十进制就是 25。
    ```

11. 列表映射 `ListMap`： 将 `Map` 表示为一个由键值对构成的链表。一般而言，对 `ListMap` 的操作需要遍历整个链表，因此对于 `ListMap` 的操作耗时为 $O(n)$ 。

事实上 Scala 对于 ListMap 用得很少，因为标准的不可变 Map 几乎总是比 ListMap 更快。唯一可能有区别的场景是：因为某种原因，需要经常访问 List 的首个元素的频率远高于访问其它元素时。

```
val map = collection.immutable.ListMap(1 -> "a", 2 -> "b")
map(1) // 返回 "a"
```

十四、其它可变集合

1. ArrayBuffer 数组缓冲：数组缓冲包含一个数组和一个大小。对数组缓冲的大部分操作和数组的速度一样，因为这些操作只是简单地访问和修改底层的数组。

   数组缓冲可以在尾部高效地添加数据，对数组缓冲追加数据需要的时间为平摊的 $O(1)$ 时间。因此，数组缓冲对于那些通过向尾部追加新元素来构建大型集合的场景非常有用。

   ```
   val buf = collection.mutable.ArrayBuffer.empty[Int]
   buf += 1
   buf += 10
   buf.toArray // Array[Int](1,10)
   ```

2. ListBuffer 列表缓冲：列表缓冲和数组缓冲很像，但是它内部使用的是链表而不是数组。如果你打算在构建完成之后，再将缓冲转换为链表，则可以考虑使用列表缓冲。

   ```
   val buf = collection.mutable.ListBuffer.empty[Int]
   buf += 1
   buf += 10
   buf.toList // List[Int](1, 10)
   ```

3. StringBuilder ：用于构建字符串。由于它非常常用，因此已经被引入到默认的命名空间中。只需要简单地使用 new StringBuilder 就可以创建：

   ```
   val buf = new StringBuilder
   buf += 'a'
   buf ++= "bcdefg"
   buf.toString // 转换为字符串
   ```

4. 链表 LinkedList：链表是由用 next 指针链接起来的节点组成的可变序列。

   在大多数语言中，null 表示空链表，但是在 scala 中行不通。因为即使是空的链表也需要支持所有的链表操作（其它集合也是类似的）。尤其是 LinkedList.empty.isEmpty 应该返回 true ，而不是抛出 NullPointerException 异常。因此，空链表是特殊处理的：它们的 next 字段指向节点本身。

   跟它的不可变版本一样，链表支持的最佳操作是顺序操作。不仅如此，在链表中插入元素或者其他链表非常容易。

5. 双向链表 DoubleLinkedList ：和 LinkedList 很相似，只不过双向链表除了 next 指针之外，还有一个 prev 指针指向当前节点的前一个节点。这个额外的链接使得移除元素的操作非常快。

6. 可变列表 MutableList：可变列表由一个单向链表和一个指向该链表末端的空节点组成，这使得向链表尾部追加元素是 $O(1)$ 复杂度的，因为它避免了遍历链表来找到末端的需要。

   目前 Scala 的 mutable.LinearSeq 是采用 MutableList 来实现的。

7. 队列 `Queue`： `scala` 除了不可变队列之外还提供了可变队列。可以像使用不可变队列一样使用可变队列，不过不是用 `enqueue` 而是用 `+=` 或 `++=` 操作符来追加元素。

   另外，对于可变队列而言， `dequeue` 方法只会简单地移除头部的元素并返回。

   ```
   val queue = new scala.collection.mutable.Queue[String]
   queue += "a"
   queue += List("b","c")
   queue.dequeue // 返回 "a"，此时 queue 仅剩下元素 "b"，"c"
   ```

8. 数组序列 `ArraySeq`：数组序列是固定大小的，内部使用 `Array[AnyRef]` 来存放元素的可变序列。

   如果你想要数组的性能，但是又不想创建泛型的序列实例（你不知道元素类型，也没有一个可以在运行时提供类型信息的 `ClassTag`），可以选用 `ArraySeq` 。

9. 栈 `Stack`：前面介绍了不可变的栈，可变的栈的工作机制和不可变版本一样，只是它的修改是原地的。

   ```
   val stack = new scala.collection.mutable.Stack[Int]
   stack.push(1)
   stack.push(2)
   stack.top // 返回2， stack 元素还是 1,2
   stack.pop // 返回2， stack 元素现在只有 1
   ```

10. 数组栈 `ArrayStack`：它是可变栈的另一种实现，内部是一个 `Array`，在需要时重新改变大小。它提供了快速的下标索引，一般而言对于大多数操作而言都比普通的可变栈更快。
11. 哈希表 `HashMap`（也叫哈希映射）：哈希表底层用数组存放其元素，元素的存放位置取决于该元素的哈希码。

    向哈希表中添加元素只需要 $O(1)$ 时间，只要数组中没有其他元素拥有相同的哈希码。因此，只要哈希表中的对象能够按照哈希码分布得足够均匀，哈希表的操作就非常快。正因为如此， `scala` 中默认的可变 `Map` 和可变 `Set` 的实现都是基于哈希表的。

    ```
    val map = collection.mutable.HashMap.empty[Int, String]
    map += ( 1 -> "a")
    map(1) // 返回 "a"
    ```

    对哈希表的遍历并不能保证按照某个特定的顺序，遍历只不过是简单地遍历底层的数组，底层数组的顺序是什么样就是什么样。如果你需要某种有保障的迭代顺序，则可以使用链式的 `HashMap` 或哈希 `HashSet` ，而不是常规的 `HashMap` 或 `HashSet` 。

    链式的 `HashMap` 或 `HashSet` 的区别在于：链式的版本额外包含了一个按照元素的添加顺序保存的元素链表。对这样的集合进行遍历，则总是按照元素添加的顺序来进行的。

12. 弱哈希映射 `WeakHashMap`： `WeakHashMap` 是一种特殊的 `HashMap` ，对于这种 `HashMap`，垃圾收集器并不会跟踪映射到其中的键的链接。这意味着：如果没有其他引用指向某个键，则该键和它关联值就会从映射中消失。

    `WeakHashMap` 对于类似缓存这类的任务而言十分有用。如果缓存的 `key -> value` 是保存在常规的哈希映射当中，这个哈希映射就会无限增长，所有的键都不会被当做垃圾来处理。

    使用弱哈希映射可以避免这个问题：一旦某个键对象不再使用，则该项就会从弱哈希表中移除。 `scala` 中弱哈希表的实现是对底层 `Java` 实现 `java.util.WeakHashMap` 的包装。

13. 并发映射 `ConcurrentHashMap`：并发映射可以被多个线程同时访问。除了常见的 `Map` 操作外，它还提供了如下原子操作：

    - `m putIfAbsent (k, v)`：除非 `k` 已经在 `m` 中， 否则添加 `k -> v` 到 `m` 中。
    - `m remove(k, v)`：如果 `k` 已经在 `m` 中，则移除该项。
    - `m replace( k, old, new)`：如果 `k` 原先在 `m` 中且它对应的值为 `old`，则将 `k` 对应的值修改为 `new`。
    - `m replace(k, v)`：如果 `k` 原先在 `m` 中，则将 `k` 对应的值修改为 `v` 。

    并发映射是 `Scala` 标准类库 中的一个特质。目前它唯一的实现是 `Java` 的 `java.util.concurrent.ConcurrentHashMap`，通过标准的 `Java/Scala` 集合转换，可以自动转换为 `Scala` 映射。

14. 可变位组 `BitSet`：可变位组和不可变位组一样，只是它可以原地修改。可变位组在更新方面比不可变位组效率稍高，因为它们不需要将那些没有改变的 `Long` 复制来复制去。

```
val bits = scala.collection.mutable.BitSet.empty
bits += 1
bits += 3
```

十五、性能

1. 不可变序列的性能特征：

	head	tail	apply	update	prepend	append	insert
List	C	C	L	L	C	L	-
Stream	C	C	L	L	C	L	-
Vector	eC	eC	eC	eC	eC	eC	-
Stack	C	C	L	L	C	L	-
Queue	aC	aC	L	L	L	C	-
Range	C	C	C	-	-	-	-
String	C	L	C	L	L	L	-

其中：

- `C` 表示 $O(1)$ 的常量时间复杂度。
- `eC` 表示从效果上来看是常量时间复杂度，但是这取决于某些前提假设，如：假设向量的最大长度或者哈希键的分布情况。
- `aC` 表示平摊的常量时间复杂度。该操作的某些调用可能会耗时一些，但是大量操作的平均时间为常量时间。
- `L` 表示$O(n)$ 的线性时间复杂度。
- `Log` 表示 $O(\log n)$ 的对数时间复杂度。
- `-` 表示对应的集合不支持该操作。

每一列表示：

- `head`：选择序列的第一个元素。
- `tail`：选择序列除了第一个元素之外的外部元素集合。
- `apply`：下标索引。

- `update`：对不可变序列执行函数式更新（`updated` 方法），对可变序列执行原地更新（`update` 方法）。
- `prepend`：将元素添加到序列头部。对不可变序列而言，该操作创建一个新的序列；对可变序列而言，该操作执行原地修改。
- `append`：将元素添加到序列尾部。对不可变序列而言，该操作创建一个新的序列；对可变序列而言，该操作执行原地修改。
- `insert`：将元素插入到序列中的指定位置。该操作只对可变序列有效。

2. 可变序列的性能特征：

	head	tail	apply	update	prepend	append	insert
ArrayBuffer	C	L	C	C	L	aC	L
ListBuffer	C	L	L	L	C	C	L
StringBuilder	C	L	C	C	L	aC	L
MutableList	C	L	L	L	C	C	L
Queue	C	L	L	L	C	C	L
ArraySeq	C	L	C	C	-	-	-
Stack	C	L	L	L	C	L	L
ArrayStack	C	L	C	C	aC	L	L
Array	C	L	C	C	-	-	-

3. 不可变 `Set/Map` 的性能特征：

	lookup	add	remove	min
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC
ListMap	L	L	L	L

4. 可变 `Set/Map` 的性能特征：

	lookup	add	remove	min
HashSet/HashMap	eC	eC	eC	L
WeakHashMap	eC	eC	eC	L
BitSet	C	aC	C	eC

其中每一列表示：

- `lookup`：测试某个元素是否包含在 `set` 中，或者选择某个 `key` 关联的值。
- `add`：添加新元素到集合中。
- `remove`：从集合中移除元素。

- `min`：`set` 的最小元素，或者 `map` 的最小 `key` 。