*someNumbers*.foreach(*print* \_)

1. 像这种使用下划线的函数被称为部分应用函数（partially applied function），在scala 3以下使用部分应用函数对某个变量赋值时候必须写出下划线，scala3则可以酌情处理。

def sum(*a*:*Int*, *b*:*Int*) = *a*+*b*val *fu* = *sum*(1,\_)

1. 在参数后增加一个\*表示重复参数参数，重复参数的类型是一个所申明的参数类型的Seqs。比如：

def echo(*args*:String\*) =  
 for(arg <- *args*) *println*(arg)

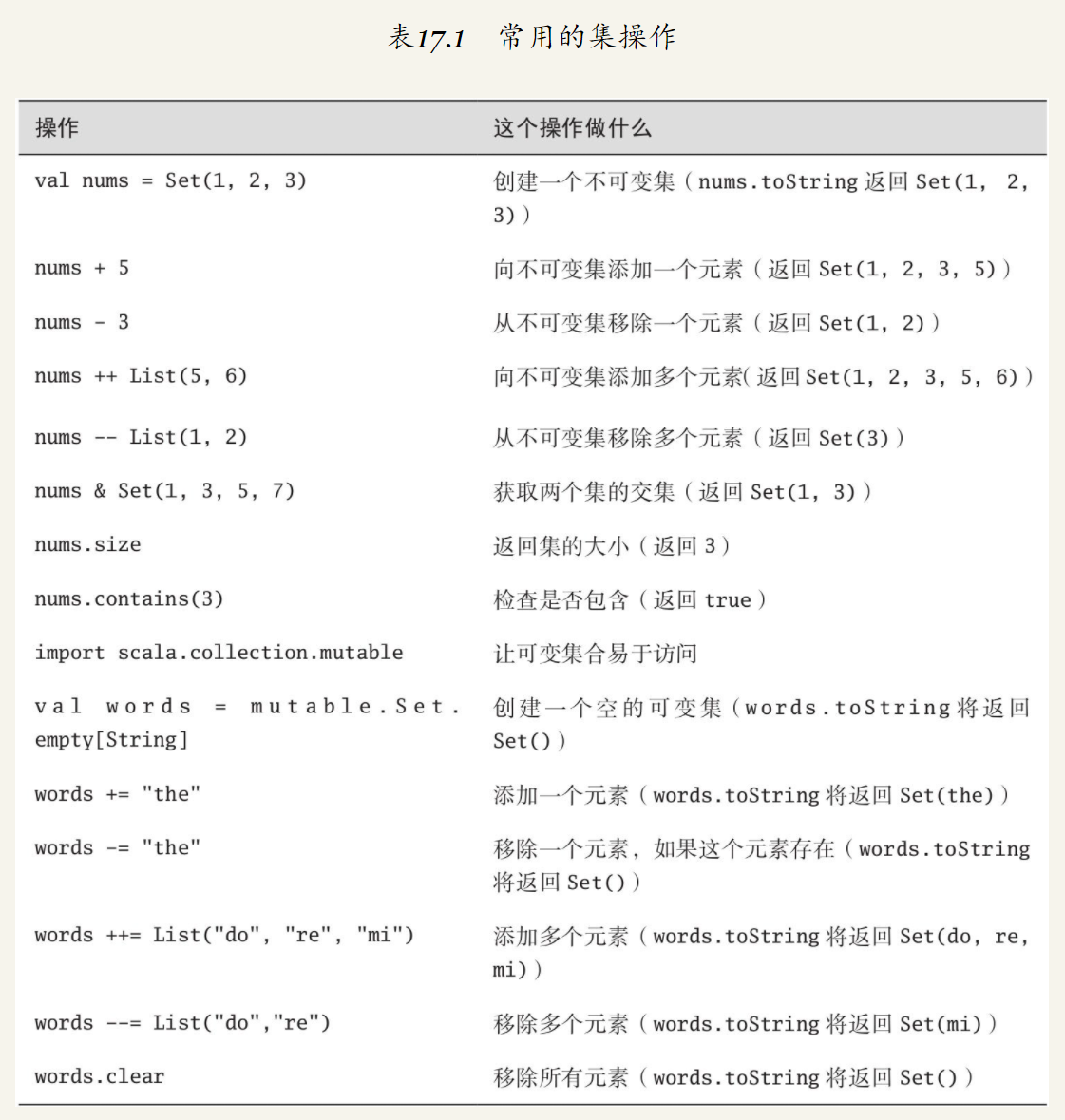
args本身是一个Seq[String]类型，当传入的参数是一个list或者是个array等类型会出错

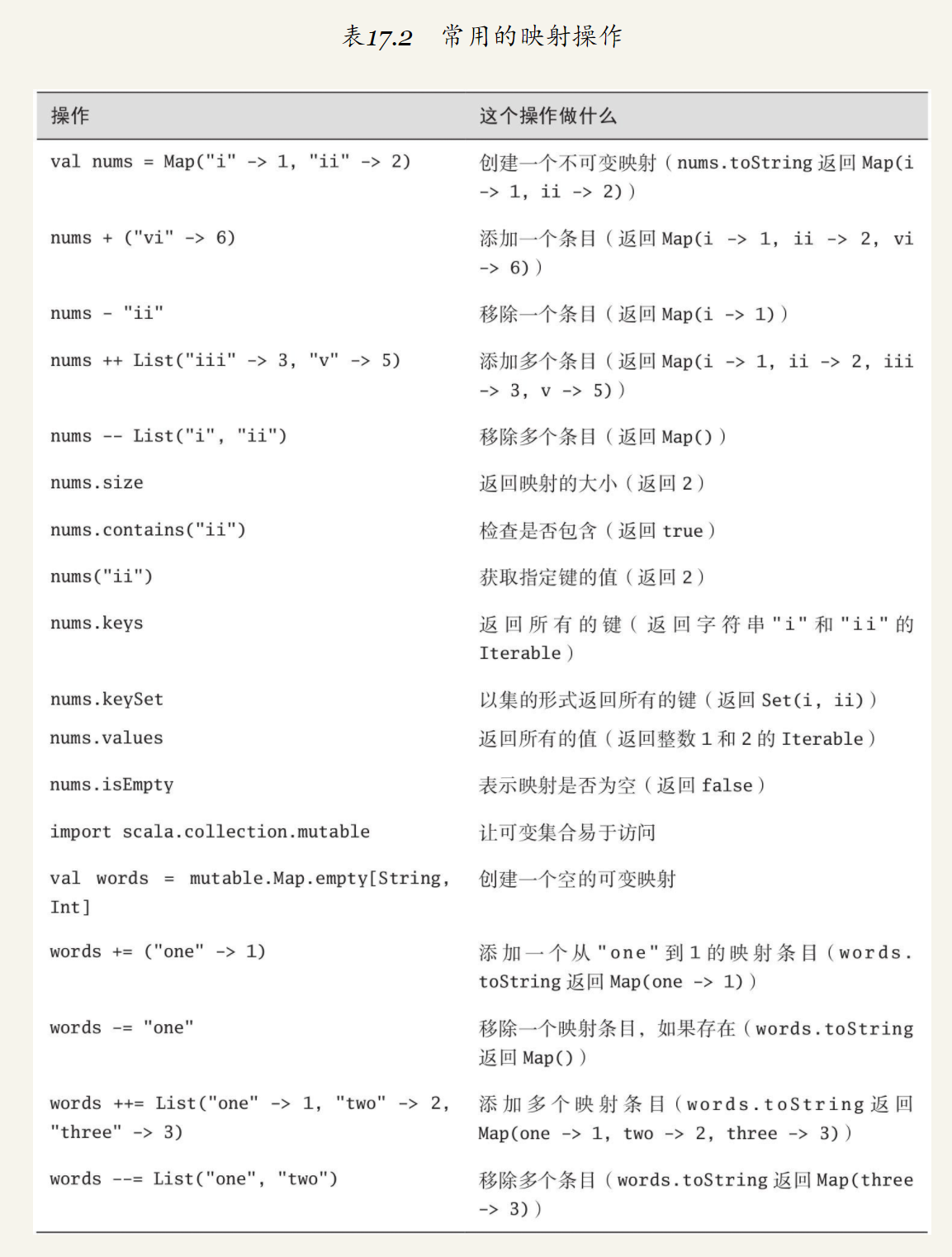
需要在传入的参数的后端增加类型声明比如 echo(List("1","2"):\_\*)表示参数不是将所有元素放在一起作为单个实参传入，而是分开一个一个传入。

1. 函数定义时候可以有默认参数，函数调用时候可以使用形参=实参的形式调用。
2. 函数柯里化本质上会产生两个或者两个以上的函数调用，比如：

def sum(*x*:*Int*)(*y*:*Int*):*Int* = *x*+*y*val *s* = *sum*(2)(\_) *//scala3中可以省略掉占位符  
println*(*s*(9)) *// 11*

1. 无参方法的定义和调用在只通过读取所在对象字段的方式访问状态的情况下推荐省略掉()，这样的做法支持所谓的统一访问原则，这种情况下def的定义跟val的定义和调用没有太多区别，唯一的不同点是直接访问字段会比方法快。
2. override关键字使用在需要对父类的实现进行重写的情况，除了加入类型检查的功能之外也是防止写出脆弱基类的必要手段。
3. ListBuffer提供了常量时间的队列后追加功能，它是属于mutable的容器，所以对List的追加功能除了使用reverse之外最好的替代方案就是使用ListBuffer。 + =和+ = : 操作符分别是append和prepend的别名，分别表示在后边追加和在前追加。
4. ArrayBuffer和Array的关系类似于STL中deque和vector的关系，ArrayBuffer实现了前后常数时间的添加和删除功能，相比与Array而言稍微慢一些。
5. StringOps实现了字符串的很多方法，字符串在处理不能直接定义的函数时都需要先隐式转换成StringOps类的对象。
6. Set和Map的实现都有两个版本，一个可变一个不可变。默认使用的Set和Map都是不可变，可变需要显示引用。scala同时提供了排序后的Set和Map版本，分别是TreeSet和TreeMap，使用的红黑树排序。
7. 对于imutable的容器都被称为完全持久化的数据结构，这类型的数据结构在经过拓展或者修改过后老版本将继续保持可用。
8. 当可变集合的代码开始变得复杂难懂的时候尤其是当我们发现经常需要担心在正确的地方对可变集合做拷贝，或者花大量时间思考谁拥有或者包含某个可变集合时候考虑换成不可变集合，不可变集合的好处除了代码比较好推敲之外还有一个好处是实现更紧凑，花费的空间更少。





1. var变量加不可变集合的组合一样能支持+=操作，本质上说是新生成了一个不可变集合并重新赋值给var变量。
2. Java中使用==对值类型来说是判断自然相等性，对引用类型而言是判断对象一致性。equals方法是用户定义的引用类型的规约相等性。Scala中==方法对值类型来说跟java一样，对引用类型的对象而言跟equals方法一样。用户可以重写equals来自定义引用类型的规约相等性。==方法在Any类中没定义成了final方法所以不能被重载。

trait *Queue*[T] {  
 def head:T  
 def tail:*Queue*[T]  
 def enqueue(*x*:T):*Queue*[T]  
}

1. Queue[T]被称为“类型构造方法”，通过指定类型参数来构造一个类型，比如Queue[Int],Queue[String]等。也可以说它是一个泛型的特质（接受类型参数的类和特质是“泛型”的，生成的类型是“参数化”的）
2. 如果S是类型T的子类型，那么Queue[S]是Queue[T]的子类型。这种情况说Queue特质在类型参数T上是协变的。默认情况下泛型类型都不是协变的，通过在T前加入一个+可以把Queue变成协变的。