基础知识

一、Scala 特点

1. 可以从`https://docs.scala-lang.org/api/all.html`查询`scala doc`文档。
2. `Scala`允许添加新的数据类型，这些新加的类型用起来和内建的类型一样。

   `Scala`也允许添加新的控制结构，其用法也和内建的控制结构一样。

1.1 面向对象&函数式

1. `Scala`同时支持面向对象编程和函数式编程。
2. `Scala`中，一切`value`都是对象，每个操作都是方法调用。如：`+`都是方法调用，`1+2`实际上是调用了`Int`类定义的、一个叫做`+`的方法。
   - 与`Scala`不同，`Java/C++`中的基本数据类型都不是对象，在这些语言中允许不是对象的`value`存在。
   - 与`Scala`不同，`Java/C++`中允许不以任何对象的成员形式存在的静态字段和静态方法。

   另外，在`Scala`中可以定义名字像操作符的方法，这样就可以用操作符表示法来调用。如：`obj ! msg`，`!`就是自定义的一个方法的名字。

   而且，`Scala`中函数就是对象，函数的类型是一个可以被继承的类，函数本身就是这个类的对象。
3. 函数式编程有两大核心理念：
   - 函数是一等公民，其地位与整数、字符串等相同。
     - 可以将函数作为参数传递给其它函数，也可以返回函数作为某个函数的返回值，也可以将函数保存在变量里。
     - 可以在函数中定义另一个函数，就像在函数中定义整数一样。
     - 可以在定义函数时不指定名字，这称作函数字面量，就像整数字面量`10`一样。

     与`Scala`不同，`Java/C/C++`中的函数是二等公民。
   - 程序中的操作应该将输入值映射成输出值，而不是原地修改数据。即：方法调用不应该产生副作用，方法只能通过接收入参、返回结果这两种方式与外部环境通信。
4. 不可变数据结构是函数式编程的基础之一，`Scala`类库在`Java API`的基础上定义了更多的不可变数据类型。如：不可变列表、不可变元组、不可变`set`、不可变`map`。

   函数式编程鼓励采用不可变数据结构和`referential transparent`的方法。`referential transparent`方法指的是：对任何给定的输入，该方法调用都可以直接被其结果替换，同时不会影响程序的语义。如：`c=f(a)`，假设`f(a)`的结果是`100`，则该语句用`c=100`替换不会对程序有任何影响。

1.2 Scala 优势

1. `Scala`与`Java`高度兼容。`Scala`运行在标准的`Java`平台上，可以和所有`Java`类库无缝协作：
   - `Scala`程序会被编译成`JVM`字节码，因此`Scala`程序运行期性能通常与`Java`程序相差无几。

- `Scala` 代码可以调用 `Java` 方法、访问 `Java` 字段、从 `Java` 类继承、实现 `Java` 接口。这并不需要任何特殊的语法或者额外的代码。
- `Scala` 重度复用了 `Java` 的数据类型。如：
  - `Scala` 的 `Int` 是用 `Java` 的基本类型 `int` 实现的。
  - `Scala` 的 `Float` 是用 `Java` 的基本类型 `float` 实现的。
  - `Scala` 的 `Boolean` 是用 `Java` 的基本类型 `boolean` 实现的。
  - `Scala` 的数组被映射成 `Java` 的数组。
  - `Scala` 的字符串字面量是一个 `java.lang.String` 。
  - `Scala` 抛出的异常必须是 `java.lang.Throwable` 的子类。

  基本上所有的 `Java` 基本数据类型在 `Scala` 包中都有对应的类，当 `Scala` 代码编译成 `Java` 字节码时， `Scala` 编译器会尽量使用 `Java` 基本数据类型从而达到更优的性能。
- 也可以在 `Java` 中调用 `Scala` 的代码。但是由于 `Scala` 比 `Java` 表达能力更为丰富，因此 `Scala` 某些高级特性需要经过加工之后才能映射到 `Java` 。

2. `Scala` 程序通常都很短。与 `Java` 相比， `Scala` 代码行数可以相差一个量级。更少的代码不仅意味着更少的打字，也意味着更少的 `bug` 和更容易阅读与理解。

   `Scala` 更少的代码归因于以下几点：
   - `Scala` 语法避免了 `Java` 程序中的一些样板代码 `boilerplate` 。如： `Scala` 中分号是可选的，通常大家也不写分号。
   - `Scala` 代码支持类型推断，因此冗余的类型信息可以去掉，这使得代码更为紧凑。
   - `Scala` 提供了大量工具来定义功能强大的类库，使得代码更加精炼。
3. `Scala` 是静态类型的，它拥有非常先进的静态类型系统。
   - `Scala` 通过类型推断避免了代码中到处是类型声明，从而导致代码过于罗嗦的问题。
   - `Scala` 通过模式匹配、创建新类型、类型组合的方式灵活的使用类型。

   静态类型系统的优点：
   - 可以证明某些运行期错误不可能发生。
     - 虽然这种保障比较简单，但这是真正的保障，不是单元测试所能提供的。
     - 静态类型系统不能代替单元测试，但是它可以减少单元测试的数量。
   - 可以安全的对代码进行重构。
   - 静态类型系统是程序化的文档，编译器会检查其正确性。

1.3 语法风格

1. `Scala` 的注释风格与 `Java` 相同：行内注释 `//` ，行间注释 `/*....*/` 。
2. `Scala` 推荐的代码缩进风格是：每个层级缩进 `2` 个空格。
3. `Scala` 的包和 `Java` 的包很类似，它们都将全局命名空间划分为多个区域，提供了信息隐藏的机制。
4. `Scala` 命令行参数可以通过 `args` 数组获取，数组下标从 `0` 开始。
   - 第 `0` 个参数是真正的命令行参数，而不是脚本文件名。
   - `Scala` 数组的索引是圆括号 `args(0)` ，这与 `Java` 不同（ `Java` 是 `args[0]` ）。
5. 在 `Scala` 中，每条语句最后的分号通常是可选的：如果当前行只有一条语句，则分号不是必须的；如果当前行有多条语句，则分号是必须的。

```
val s1="hello" ; println(s1) // 必须有分号
println("world") // 不必有分号
```

- 如果一条语句跨越多行，则大多数情况下直接换行即可，`Scala` 会自动断句：

```
if (x<2)
  println("1")
else
  println("2")
```

但是有时候会产生意外的效果：

```
x
+y
```

会被解析成两条语句`x` 和`+y`。如果希望编译器解析成单条语句`x+y` ，则有两种做法：

- 使用圆括号：

```
(x
+y)
```

- 将`+` 放到行尾。当使用中缀操作符（如`+`）来串接表达式时，`Scala` 风格是将操作符放到行尾：

```
x +
y
```

- 分号推断规则：除非以下任意一条为`true`，否则代码行的末尾就被当作分号处理：
  - 当前行以一个不能作为语句结尾的词结尾，如：英文句点`.`，中缀操作符。
  - 下一行以一个不能作为语句开头的词开头。
  - 当前行的行尾出现在圆括号`()` 或者方括号`[]` 内，因为圆括号、方括号不能包含多条语句。

二、入门

2.1 基础数据类型

2.1.1 基础数据类型

1. `Java` 的基础类型和操作符在`Scala` 中具有相同的含义。
2. `Scala` 的基础数据类型包括：`String`、值类型（包括：`Int/Long/Short/Byte/Float/Double/Char`）、`Boolean`。
   - `Byte/Short/Int/Long/Char` 类型统称为整数类型，整数类型加上`Float` 和`Double` 称作数值类型。
   - `String` 位于`java.lang`，其它几种基础数据类型位于`scala` 包。如`Int` 的完整名称是`scala.Int` 。

     由于`scala` 源文件中默认自动引入了`java.lang` 包的所有成员和`scala` 包的所有成员，因此可以在任何地方使用这些类型的简单名字（而不是完整名称）。
   - 取值区间：
     - `Byte`：`8`位带符号整数。取值区间：$[-2^7, 2^7-1]$，闭区间。

- `Short`：16 位带符号整数。取值区间：$[-2^{15}, 2^{15}-1]$，闭区间。
- `Int`：32 位带符号整数。取值区间：$[-2^{31}, 2^{31}-1]$，闭区间。
- `Long`：64 位带符号整数。取值区间：$[-2^{63}, 2^{63}-1]$，闭区间。
- `Char`：16 位无符号 `Unicode` 字符。取值区间：$[0, 2^{16}-1]$，闭区间。
- `String`：`Char` 的序列。
- `Float`：32 位 `IEEE 754` 单精度浮点数。
- `Double`：64 位 `IEEE 754` 双精度浮点数。
- `Boolean`：`true` 或者 `false`。

3. `Scala` 基础数据类型和 `Java` 中对应的类型取值区间完全相同，这使得 `Scala` 编译器可以在生成的字节码中，将 `Scala` 的值类型，如：`Int/Double` 的实例转换成 `Java` 的基本类型。

4. `Scala` 中的每个基础数据类型都有一个`富包装类`，该富包装类提供额外的方法。当在基础数据类型调用更多的方法时，这些基础数据类型通过隐式转换得到对应的富包装类，并对富包装类调用这些方法。

   基础数据类型对应的富包装类为：`Byte -> scala.runtime.RichByte`、`Short -> scala.runtime.RichShort`、`Int -> scala.runtime.RichInt`、`Long -> scala.runtime.RichLong`、`Char -> scala.runtime.RichChar`、`Float -> scala.runtime.RichFloat`、`Double -> scala.runtime.RichDouble`、`Boolean -> scala.runtime.RichBoolean`、`String -> scala.collection.immutable.StringOps`。

```
0 max 5                 // 结果：5
-1.0 abs                // 结果：1.0
-2.7 round              // 结果：-3L
1.5 isInfinity          // 结果：  false
(1.0/0) isInfinity      // 结果：  true
4 to 6                  // 结果：  Range(4,5,6)
"hello" capitalize      // 结果：  "Hello"
```

2.1.2 字面量

1. 所有基础数据类型都可以用字面量 `literal` 来表示该类型的常量值。

2. 整数字面量：`Byte/Short/Int/Long` 的字面量有两种形式：十进制表示和十六进制表示（以 `0x` 或者 `0X` 开头，包含 `0~9` 以及大小的 `a~f` 或者小写的 `A~F`）。

   - 整数字面量不支持八进制的表示，也不支持以 `0` 开头的表示（如 `012`）。
   - `Scala` 的 `shell` 总是以十进制打印整数值，无论它是用什么形式初始化的。
   - 如果整数字面量是以 `l` 或者 `L` 结尾，则它是 `Long` 类型的；否则就是 `Int` 类型的。
   - 当一个 `Int` 类型的字面量赋值给一个 `Byte` 或者 `Short` 类型的变量时，该字面量会被当做 `Byte` 或者 `Short` 类型，只要这个字面量的值在变量类型的取值区间内即可。

     ```
     val byte : Byte = 12     // Int 字面量被当作 Byte 类型
     val short : Short = 123  // Int 字面量被当作 Short 类型
     ```

3. 浮点数字面量：由十进制数字、可选的小数点、可选的 `E` 或者 `e` 开头的指数组成（科学计数法）。

   如果浮点数字面量以 `f` 或者 `F` 结尾，则它是 `Float` 型的；否则就是 `Double` 型的。`Double` 型浮点数字面量也可以以 `d` 或者 `D` 结尾，但是这不是必须的。

```
val float1 : Float = 1.1f
val float2 : Float = 123E45F   // 带指数
val double1 : Double = 1.1
val double2 : Double = 123E45  // 也可以显式添加 d 或者 D 结尾
```

4. 字符字面量：由一对单引号、任意一个`Unicode`字符组成。这里除了显式的给出原始字符，也可以用字符的`Unicode`码来表示：`\u`加上`Unicode`码对应的四位十六进制数字。

```
val a1 = 'A'         // 字符字面量
val a2 = '\u0041'    // 使用 Unicode 码
```

事实上，这种`Unicode`字符的方式可以出现在`Scala`程序的任何位置，包括变量名中：

```
val \u0041B = 1 // 等价于 val AB=1
```

但这种方式并不友好，因为不容易阅读。

还有一些字符字面量是特殊转义字符：

- `\n`： 换行符`\u000A`。
- `\b`：退格符`\u000B`。
- `\t`：制表符`\u0009`。
- `\f`：换页符`\u000C`。
- `\r`：回车符`\u000D`。
- `\"`：双引号`\u0022`。
- `\'`：单引号`\u0027`。
- `\\`：反斜杠`\u005C`。

5. 字符串字面量：由双引号包起来的字符组成。其中每个字符也可以用`Unicode`码表示，也支持转义字符。

`Scala`支持一种特殊语法来表示原生字符串`raw string`：用三个双引号开始，并以三个双引号结束。其内部可以包含任何字符，包括换行、单双引号、以及其它特殊字符（三个双引号除外）。

原生字符串对于包含大量转义字符、或者跨越多行的字符串比较友好。

```
println("""This is a raw string: Contain "" and '' and \
and a new line
and another new line\n""")
```

6. `Symbol`字面量：格式为`'ident`，其中`ident`可以是任何字母、数字组成的标识符。

一个`Symbol`是一种特殊的字符串，相比较于`String`类型，它更节省内存并且相等比较的速度很快。

- 事实上`String`类内部维护一个字符串池`strings pool`。当调用`String`的`intern()`方法时，如果字符串池中已经存在该字符串，则直接返回池中字符串对象的引用；如果不存在，则将该字符串添加到池中，并返回该字符串对象的引用。执行过`intern()`方法的字符串被称作内部化了的，默认情况下代码中的字符串字面量都是内部化了的（在`Java`中，字符串常量也是内部化了的）。同值字符串的`intern()`方法返回的引用都相同。

  而在`Scala`中，`Symbol`类型的对象是被内部化了的，任意同名`symbol`都指向同一个`Symbol`，因此节省了内存。

- 由于不同名`symbol`一定指向不同的`Symbol`对象，因此`symbol`对象之间可以使用操作符`==`快速的进行相等性比较，常数时间内即可完成。而字符串的`equals`方法需要逐个字符的比较两个字符串，取决于两个字符串的长度。
- `Symbol`类型一般用于快速比较，如`Map<Symbol,Data>`查询`Symbol`要比`Map<String,Data>`查询`String`快得多。
- `Symbol`字面量会被编译器展开成一个工厂方法的调用：`Symbol("ident")`。

```
val s = 'aSymbol // 等价于 Symbol("aSymbol")
```

- 对于`Symbol`对象，你唯一能做的是获取它的名字：

```
println(s.name) // 输出：aSymbol
```

7. 布尔值字面量：只有`true`和`false`。

2.1.3 字符串插值

1. `Scala`支持字符串插值：允许在字符串字面量中嵌入表达式。

```
val name="world"
println(s"hello,$name!")
```

其中表达式`s"hello,$name!"`称作`processed`字符串字面量。由于`s`出现在字符串首个双引号之前，因此`Scala`将使用`s插值器`来处理该字面量。`s`插值器将对内嵌的每个表达式求值，对求值结果调用`toString`来替代字面量中的那些表达式。

在被处理的字符串字面量中，可以随时用美元符号`$`开启一个表达式。`Scala`将从美元符号开始、直到首个非标识符的部分作为表达式。如果表达式包含非标识字符（如空格、操作符），则必须将其放入花括号`{}`中，左花括号需要紧跟`$`。

2. `scala`还提供另外两种字符串插值器：
   - `raw`字符串插值器：其行为跟`s`字符串插值器类似，但是`raw`字符串插值器并不识别转义字符。

```
println(raw"No new line\n") // \n 不经过转义
```

   - `f`字符串插值器：其行为跟`s`字符串插值器类似，但是允许为内嵌的表达式添加`printf`风格的格式化指令。格式化指令位于表达式之后，以百分号`%`开始，使用`java.util.Formatter`给出的语法。

```
println(f"${math.Pi}%.5f") // 输出：3.14159
```

     如果不给出格式化指令，则默认采用`%s`，其含义是用`toString`的值来替换，就像`s`字符串插值器一样。
3. 在`Scala`中，字符串插值器是通过编译期间重写代码来实现的。编译器会将任何由某个标记（如`r`或者`f`）紧跟着字符串字面量的左双引号这样的表达式当作字符串插值器表达式求值。

   你也可以定义自己的字符串插值器来满足不同的需求。

2.2 变量定义

1. Scala 的变量分为两种：val 和 var 。
   - val：和 Java 的 final 变量类似，一旦初始化就不能被重新赋值。

     注意：当采用 val 定义一个变量时，变量本身不能被重新赋值，但是它指向的对象可能发生改变。如：

     ```
     val string1 = new Array[String](3)
     string1(0)="hello"
     string1(1)=", "
     string1(2)="world!\n"
     ```

     不能将 string1 重新赋值为另一个数组，但是可以改变 string1 指向的数组的元素。
   - var：类似于 Java 的非 final 变量，在整个生命周期内可以被重新赋值。

   var 和 val 都有各自的用武之地，本质上并没有哪个更好或者更坏。

   如果代码中包含任何 var 变量，则它通常是非函数式的。如果代码中完全没有 var，则它可能是函数式的。因此函数式风格的编程尽量不使用 var 。 Scala 鼓励使用函数式编程，尽量采用 val 。因为这样的代码更容易阅读、更少出错。

   采用 val 的另一个好处是等式推理 equational reasoing 的支持。引入的 val 等于计算它的值的表达式（假设这个表达式没有副作用）。因此任何该 val 变量名出现的地方，都可以直接用对应的表达式替代。
2. Scala 的变量定义（以 val 为例）：

   ```
   val msg:String = "Hello word!"
   ```

   这中定义方式显式给出了类型标注，方式为：在变量名之后添加冒号和类型 。

   实际上 String 的完整形式为 java.lang.String 。因为 Scala 程序默认引入了 java.lang 包，因此可以直接写作 String 。

   由于 Scala 的类型推断可以推断出非显式指定的类型，因此上述定义可以修改为：

   ```
   val msg = "Hello world!"
   ```

   这样的代码更紧凑、易读。

2.3 标识符

1. 构成 Scala 标识符的两种最重要的形式：字母数字组合、操作符。
2. 字母数字组合标识符：以字母或者下划线开始，可以包含更多的字母、数字或下划线。
   - 字符 $ 也算字母，但是它预留给那些由 Scala 编译器生成的标识符。
   - Scala 遵循了 Java 的驼峰命名法 camel-case 的传统。如： toString、HashSet 。
     - 字段、方法参数、局部变量、函数的命名应该以小写字母开始。
     - 类、特质的命名应该以大写字母开始。
   - 虽然下划线是合法的标识符，但是它们在 Scala 中并不常用，原因有两个：一个原因是和 Java 保持一致。另一个原因是，下划线在 Scala 中还有很多其它非标识符的用法。

     在标识符结尾尽量不要使用下划线。如：

```
val name_: Int = 1 // 错误
val name_ : Int = 1 // 正确
```

第一行将被`Scala`识别为变量名`name_:`，这会引起编译错误。

- `Scala`中，常量命名只要求首字母大写，而`Java`中要求全大写而且下划线分隔不同的单词。

  `Scala`中的常量并不是`val`。如：方法的参数是`val`，但是每次被调用时，这些`val`都得到不同的值。

3. 操作符标识符：由一个或者多个操作符组成。操作符指的是那些可以被打印的`ASCII`字符，如`+,:,?,&`。

   `Scala`编译器会在内部将操作符标识符用内嵌的`$`方式转换为合法的`Java`标志符。如：`:->`被转换成`$colon$minus$greater`。如果你希望从`Java`代码中访问这些标识符，则需要使用这种内部形式。

4. 混合标识符：由一个字母数字组合操作符、一个下划线、一个符号操作符组成。如：`unary_+`用于表示类的`+`操作符的方法名。

5. 字面标识符：用反引号括起来的任意字符串。

   可以将任何能被`runtime`接收的字符串放在反引号当中，甚至当该字符串是`Scala`保留字时也生效。如：

```
val `val` = "hello" // val 是个保留字
```

2.4 操作符

1. `Scala`为基础数据类型提供了一组丰富的操作符，但这些操作符其实只是普通方法调用的`语法糖`。如：`1+2`实际上是`1.+(2)`，它调用的是`Int`类的一个名为`+`的方法，该方法接收一个`Int`参数并返回一个`Int`结果。

```
val int = 1+2 // 等价于 1.+(2)
```

   实际上`Int`包含多个重载的`+`方法，这些方法分别接收不同的参数类型。

2. 在`Scala`中，操作符表示法不仅仅局限于那些其它语言（如`Java/Python`）中看起来像是操作符的那些方法，也可以包括任何方法。即：任何方法都可以是操作符。如：

```
val s = "hello world"
val idx = s.indexOf('h') // 标准的方法调用
val idx2 = s indexOf 'h' // 操作符表示法
```

   如果方法的参数有多个，则在操作符表示法中需要将这些参数都放在圆括号里。

```
val idx3 = s indexOf ('o',3) // 从第3个位置开始查找
```

3. 操作符方法虽然方便使用，但是不能滥用。过度使用操作符方法会使得代码难于阅读和理解。

4. `Scala`将从数组到表达式的一切都视为带方法的对象来处理，从而实现了概念上的简化。这种统一描述并不会带来显著的性能开销，因为`Scala`在编译代码时，会尽可能使用`Java`数组、基本类型和原生的算术指令。

如，`Scala` 的数组`Array`的访问方式是：将下标放置在圆括号里，如：`string1(0)` 。这一点与`Java/C++/Python` 都不同。

- 在`Scala` 中，当用一个圆括号包围一组值应用到某个对象上时，将调用该对象的`.apply()` 方法。因此`string1(0)` 等价于`string1.apply(0)` 。

  因此在`Scala` 中访问数组的一个元素就是一个简单的方法调用，并没有任何特殊的地方。

- 在`Scala` 中，当用一个圆括号包围一组值应用到某个对象上并位于赋值`=` 的左侧时，将调用该对象的`.update()` 方法。因此`string1(0)="hello"` 等价于`string1.update(0,"hello")` 。

2.4.1 前/中/后缀操作符

1. 像`+` 这类操作符是中缀操作符，这意味着被调用的方法名位于调用对象和参数之间。如：`1+2` 。

   `Scala` 还提供了另外两类操作符：

   - 前缀操作符：方法名位于调用对象的前面。如：`-1` 。
   - 后缀操作符：方法名位于调用对象的后面。如：`1 toLong` 。
2. 跟中缀操作符表示法不同，前缀操作符和后缀操作符是一元的：它们只接受一个操作元。前缀操作符中，操作元位于操作符右侧；后缀操作符中，操作元位于操作符左侧。

   - 唯一能被用作前缀操作符的是`+,-,!,~` 。
   - 前缀操作符对应的完整方法名是`unary_` 加上操作符（注意：不包含圆括号）。如：

     ```
     -1 // 前缀操作符
     1.unary_- // 方法调用的形式
     ```

3. 后缀操作符是那些不接收参数，并且在调用时没有用`.()` 的方法。在`Scala` 中，可以在方法调用时省去空的圆括号。

   但是通常来讲，如果方法有副作用，则需要保留空的圆括号。如：`println()` 。如果方法没有副作用，则可以省略空的圆括号。`Scala` 支持进一步去掉句点`.`，从而演化为后缀操作符表示法。

   ```
   val s = "Hello world"
   val s1 = s.toLowerCase() // 标准调用
   val s2 = s.toLowerCase   // 省略空的圆括号
   val s3 = s toLowerCase   // 后缀操作符表示法
   ```

2.4.2 各类操作符

1. 算术操作符：`+,-,*,/,%`：加、减、乘、除、取余。它们都是中缀操作符，对任何数值类型调用对应的算术方法。

   - 当左右两个操作元都是整数类型时，`/` 操作符会计算出商的整数部分，不包含余数。`%` 操作符得到整数除法后的余数。
   - `%` 用于浮点数除法时，其余数与`IEEE 754` 标准不同。`IEEE 754` 的余数在计算时用四舍五入，而`%` 是截断。

     如果需要`IEEE 754` 的余数，则采用`scala.math.IEEEremainder()` 方法。

     ```
     11.0%4.0 // 结果：3.0
     math.IEEEremainder(11.0,4.0) // 结果：-1.0
     ```

2. `Scala` 对数值类型还提供了`+,-` 两个一元前缀操作符（`unary_+` 方法和`unary_-` 方法），用于表示数值型字面量是正数还是负数。
    - 如果不给出`+,-`，则数值字面量默认为正数。
    - 一元`+`仅仅是为了和一元`-` 对称，它没有任何作用。
    - 一元`-` 不仅可以作用于数值字面量，还可以作用于变量，用来给变量取负值。
3. `Java` 的 `++i,i++,--i,i--` 在`Scala` 中并不工作。在`Scala` 中你可以使用：`i = i+1` 或者`i+=1` 表示自增，`i = i-1` 或者 `i-=1` 表示自减。
4. 关系操作符：`>,<,>=,<=`：大于、小于、大于等于、小于等于。它们都是中缀操作符，用于比较数值类型的大小，返回`Boolean` 结果。

   一元前缀操作符`!`（`unary_!` 方法），用于对`Boolean` 值取反。
5. 逻辑操作符：`&&,&,||,|`：逻辑与，逻辑与，逻辑或，逻辑或。它们都是中缀操作符，用于对`Boolean` 操作元执行逻辑与/或，返回`Boolean` 结果。
    - `&&` 和`||` 是短路求值的：只会对结果有决定性作用的部分求值。当操作符左侧的操作元能够决定表达式的结果时，右侧的操作元不会被求值。
    - `&` 和`|` 是非短路求值的：它们会对所有的操作元进行求值。

    ```
    true || (1/0 >=0) // 返回 true，短路求值。右侧操作元不会被求值
    true | (1/0 >=0)  // runtime error。 右侧操作元会被求值
    ```
6. 在`Scala` 中，所有方法都有一个机制来延迟对入参的求值，或者干脆不对其求值。这个机制叫做传名参数`by-name parameter` 。
7. 位运算符：`&,|,^`：按位与、按位或、按位异或。它们都是中缀操作符，用于对整数类型执行位运算。

   一元前缀操作符`~`(`unary_~` 方法)，用于对操作元的每一位取反。
8. 位运算符：`<<,>>,>>>`：左移，右移，无符号右移。它们都是中缀操作符，用于将整数左移或者右移。

   左移和无符号右移会将空出的位自动填0，右移会将空出的位自动填上符号位（最高位）。

    ```
    -1 >> 31  // 结果：  -1
    -1 >>> 31 // 结果：  1
    1 << 2    // 结果：4
    ```
9. 相等运算符：`==,!=` ：相等比较、不等比较。它们都是中缀操作符，用于比较两个对象是否相等。
    - 这两个操作符实际上可以应用于所有对象，而不仅仅是数值类型。
    - 可以比较不同类型的两个对象，甚至可以和`null` 比较。背后的规则很简单：
        - 首先检查左侧是否为`null`，如果不是`null`，则调用左侧对象的`equals` 方法。
        - 如果左侧是`null`，则检查右侧是`null`。

      由于`Scala` 有自动的`null` 检查，你不必亲自做这个检查。
10. 在`Java` 中，可以用`==` 来比较基本类型和引用类型。对于基本类型，`==` 比较的是值的相等性；对于引用类型，`==` 比较的是引用的相等性。

    而`Scala` 中，`==` 对基本类型和引用类型都比较的是值的相等性。`Scala` 提供了`eq` 和 `ne` 来用于比较引用相等性。

2.4.3 优先级和结合性

1. 操作符优先级：决定了表达式中哪些部分优先求值。当然你也可以通过圆括号来指定求值顺序。
   - `Scala` 中的操作符仅是用操作符表示法来使用对象的方法而已，它根据方法名的首个字母来判定优先级。
   - `Scala` 的操作符优先级（依次递减）：`（所有其它特殊字符）`、`*,/,%`、`+,-`、`:`、`=,!`、`<,>`、`&`、`^`、`|`、`（所有字母）`、`（所有赋值操作符）`。
     - 位于同一级的操作符具有相同的优先级。
     - 操作符优先级查看的是该操作符打头的字符，如 `&&` 的优先级查看的是 `&` 字符，`<<` 的优先级查看的是 `<` 字符。
     - 一个例外是赋值操作符，它们以 `=` 结尾，且不是比较操作符（不是 `<=,>=,!=`），它们的优先级和简单的赋值操作符 `=` 相同。即：`*=` 优先级不是由 `*` 决定，而是由 `=` 决定。
2. 操作符结合性：当多个同等优先级的操作符并排时，操作符的结合性决定了操作符的分组。
   - `Scala` 中，操作符的结合性由操作符的最后一个字符决定。任何以 `:` 字符结尾的方法都是在它右侧的操作元上调用的，传入左侧的操作元；任何以其它字符结尾的方法都是在它左侧的操作元上调用的，传入右侧的操作元。

     如 `a*b` 等价于 `a.*(b)`，而 `a ::: b` 等价于 `b.:::(a)`。
   - 不论操作符的结合性是哪一种，其操作元的求值顺序都是从左到右。

     如：`a ::: b` 等价于：

     ```
     val x = a //  优先求值
     val y = b
     y.:::(x) // 操作元求值顺序：从左到右
     ```
3. 一个良好的编码风格是清晰的表达什么操作符被用在什么表达式上。你唯一可以放心的让其它程序员不查文档就能知道的优先级顺序是：乘除比加减优先级更高。因此通常添加圆括号 `()` 来显式的呈现表达式的优先级。

2.5 内建控制结构

1. `Scala` 只有很少的内建控制结构：`if,while,for,try,match` 和函数调用。
2. `Scala` 所有的控制结构都有返回值，如：`if、for` 等结构都有返回值。这是函数式语言采取的策略：程序被认为是用于计算出某个值，因此程序的各组成部分也应该计算出某个值。

2.5.1 if

1. `if` 控制结构：首先测试某个条件，然后根据条件是否满足来执行两个不同分支中的一个。`if` 表达式的返回值就是被选中分支的值。

   ```
   val s = "hello word"
   if(!args.isEmpty)
     s = args(0)
   val s = if(!args.isEmpty) args(0) else "hello word" // 等价形式
   ```

2.5.2 while

1. `while` 控制结构：包含了一个条件检查和一个循环体，只要条件检查为真则循环体继续执行。

- `Scala` 也有 `do while` 循环，它跟 `while` 循环类似，只是它会首先执行循环体然后再执行条件检查。
- `while` 和 `do-while` 并不会返回一个有意义的值，即返回类型为 `Unit` 。
- 在 `Scala` 中赋值表达式的结果是 `Unit`，而 `Java` 中赋值表达式的结果是被赋予的值。因此下面的做法在 `Scala` 中是不可行的：

```
var line = ""
while ((line=readLine())!="")
  println("read: "+line)
```

由于 `line=readLine()` 返回 `Unit`，因此 `Unit!=""` 永远成立，则循环体永远执行。

2. 通常 `while` 循环是和 `var` 成对出现的。由于 `while` 循环没有返回值（或者说返回 `Unit`），它要想对程序产生任何效果则要么更新一个 `var` 要么执行 `I/O` 。

   因此，对于代码中的 `while` 循环尽量采用其它方案来替代，除非确实难以替代它。

3. 对于 `while` 循环通常可以用递归函数的方式来替代。

```
// while 循环版本
var i = 0
var found = false
while (i<args.length && !found)
{
    if (!args(i).startsWith("-")){
        if (args(i).endsWith(".scala")) find = true
    }
    i += 1
}
// 递归函数版本
def search(i:Int):Int = {
    if (i>=args.length) -1
    else if (args(i).startsWith("-")) search(i+1)
    else if (args(i).endsWith(".scala")) i
    else search(i+1)
}
var  i = search(0)
```

这个递归函数比较特殊：所有的递归调用都发生在函数尾部，因此称作尾递归。编译器会将尾递归展开成和 `while` 循环类似的代码。

2.5.3 for

1. `for` 表达式：`Scala` 中的 `for` 表达式有很多功能。
2. 最简单的功能是遍历集合的所有元素。

```
val files = (new java.io.File(".")).listFiles
for (file <- files)  // file 是 val
  println(file)
```

通过 `file <- files` 这样的生成器语法，我们将遍历 `files` 的每个元素。每次迭代时，一个新的、名为 `file` 的 `val` 都被初始化成 `files` 中一个元素的值。

3. 你也可以遍历一个索引。

```
for (i <- 1 to 4)
  println(i)
```

`1 to 4` 将生成一个区间`Range`，范围是`[1,4]`（闭区间）。如果希望得到一个左闭右开区间，则使用 `1 until 4` 。

4. 有时需要遍历集合中的部分元素，而不是全部。此时可以在 `for` 中添加过滤器。形式为：`for` 表达式的圆括号中添加 `if` 子句。

   可以包含任意多的过滤器，直接添加 `if` 子句即可。

```
val files = (new java.io.File(".")).listFiles
for (
    file <- files
    if file.isFile
    if file.getName.endsWith(".scala")
)  // 添加过滤器
  println(file)
```

5. 也可以添加多个 `<-` 子句，此时得到嵌套的“循环”。

   如果愿意，你也可以使用花括号 `{}` 而不是圆括号 `()` 来包括生成器和过滤器，好处是可以在需要时省略某些分号。因为`Scala` 编译器在圆括号中并不会自动推断分号。

```
def getLines(file:java.io.File) =
scala.io.Source.fromFile(file).getLines().toList

val files = (new java.io.File(".")).listFiles
for (
    file <- files
    if file.getName.endsWith(".scala") ;  //必须添加分号 ;
    line <- getLines(file)
    if line.trim.matches("test*.scala")
)  // 添加过滤器
  println(file+":"+line.trim)
```

6. 在`for` 的生成器和过滤器中，支持中途变量绑定：将表达式的结果绑定到新的变量上。被绑定的变量引入和使用就跟`val` 一样。

   上述例子中，`line.trim` 被重复调用两次。通过中途变量绑定可以只需要调用一次。

```
def getLines(file:java.io.File) =
scala.io.Source.fromFile(file).getLines().toList

val files = (new java.io.File(".")).listFiles
for {
    file <- files
    if file.getName.endsWith(".scala")   //采用大括号，所以不用添加分号 ;
    line <- getLines(file)
    trimmed = line.trim          // 中途变量绑定，trimmed 初始化为 line.trim
的结果
    if trimmed.matches("test*.scala")
}  // 添加过滤器
  println(file+":"+trimmed)
```

7. `for` 表达式可以返回有效的值。这是通过 `yield` 关键字实现的。

```
val files = (new java.io.File(".")).listFiles
def scalaFiles = for {
    file <- files
    if file.getName.endsWith(".scala")
} yield file
```

   - `for` 表达式的代码体每次都被执行，都会产出一个值。当 `for` 表达式执行完毕后，其结果将包含所有产出的值，包含在一个集合中。结果集合的类型基于迭代子句中处理的集合种类。这个例子中，每次产出的值就是 `file`，返回的集合类型为 `Array[File]` 。
   - `for` 表达式的代码体如果有多行表达式，则最后一个表达式的返回值就是该表达式代码体的结果。
   - 注意：`yield` 关键字的位置是：`for 子句 yield 代码体` 。如果代码体由花括号 `{}` 包围的，则 `yield` 必须在花括号之前：

     ```
     for (file <- files  if file.getName.endsWith(".scala"))
     {
        yield file  // 语法错误
     }
     // 应该是：yield {file}
     ```

2.5.4 match

1. `match` 表达式：让你从若干个可选项中选择，就像其它语言中的 `switch` 语句一样。但是 `match` 表达式允许你使用任意的 `pattern` 来选择。

```
val firstArg = if (args.length >0) args(0) else ""
firstArg match {
    case "apple" =>  "apple"
    case "orange" =>  "orange"
    case _ =>  "unknown"
}
```

   缺省的 `case` 以下划线 `_` 来表示，这个通配符在 `Scala` 中经常用于表示某个完全不知道的值。

2. `Scala` 的 `match` 与 `Java` 的 `switch` 相比有一些重要区别：

- 任何常量、字符串等等都可以用作`case`，而不仅局限于`Java`的`case`支持的整数、枚举和字符串常量。
- 在每个`case`结尾并没有`break`。在`Scala`中，`break`是隐含的，并不会出现某个`case`执行结束之后继续执行下一个`case`的情况。
- `Scala`的`match`表达式会返回值。匹配到的`case`的子句的结果就是`match`表达式的返回值。

2.5.5 break

1. `Scala`中并没有`break`和`continue`关键字，如果想实现对应的功能，最简单的方式是采用`if-else`结构。

   如果仍然需要`break`功能，则`scala.util.control.Breaks`类给出了`break`方法，它可以被用于退出包含它的、用`breakable`标记的代码块。

   ```
   import scala.util.control.Breaks._
   import java.io._
   val in = new BufferedReader(new InputStreamReader(System.in))
   breakable{                    // 标记
       while(true){
           println("hello")
           if (in.readLine()=="") break // break
       }
   }
   ```

   其实现方式为：由`Breaks.break`抛出一个异常，然后由外围的`breakable`方法的应用所捕获。因此，对`break`的调用并不需要一定和`breakable`的调用放在同一个方法内。

2.6 异常

1. `Scala`的异常处理也和其它语言类似，方法除了正常返回某个值意外，还可以通过抛出异常来终止执行。方法的调用方要么捕获并处理这个异常，要么自我终止并让该异常传播到更上层调用方。

   异常通过这种方式传播，逐个展开调用栈，直到某个方法处理该异常或者再没有更多方法了为止。

2. `Scala`中抛出异常与`Java`看起来一样，你首先创建一个异常对象，然后通过`throw`关键字将其抛出。

   ```
   throw new IllegalArgumentException
   ```

   与`Java`不同，`Scala`中的`throw`是一个有返回类型的表达式。技术上来讲，`throw`表达式的类型是`Nothing`。因此可以将`throw`表达式的值当作任何类型的值来看待，因为任何想使用这个返回值的地方都没有机会真正使用它。

   ```
   val half = if (n%2 ==0) n/2 else throw new RuntimeException("n must be
   even")
   ```

   如果`throw`表达式没有返回值，则上述的`if/else`结构无法通过编译。

3. 可以通过`catch`子句来捕获异常，其语法与`Scala`的模式匹配相一致。

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
try {
    val f = new FileReader("input.txt")
    // 使用并关闭文件
} catch {
    case ex : FileNotFoundException => // 处理找不到文件的情况
    case ex : IOException => // 处理其它IO的情况
}
```

`try-catch` 表达式和其它语言一样：首先代码体被执行。如果抛出异常，则依次尝试每个 `catch` 子句，执行第一个匹配的 `catch` 子句。如果所有的子句都不匹配，则异常继续向上传播。

4. 在 `scala` 中，并不会要求你捕获 `checked exception` 或者在 `throws` 子句里声明，这和 `Java` 不同。

   当然你也可以通过 `@ throws` 注解来声明一个 `throws` 子句，但这不是必须的。

5. 可以将那些无论是否抛出异常都想执行的代码以表达式的形式包括在 `finally` 子句里。

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
try {
    val f = new FileReader("input.txt")
    // 使用并关闭文件
} catch {
    case ex : FileNotFoundException => // 处理找不到文件的情况
    case ex : IOException => // 处理其它IO的情况
}
finally {
    f.colse()   // 确保关闭文件
}
```

这是确保那些非内存资源（如：文件、套接字、数据库连接）被正确关闭的惯用做法：首先获取资源，然后在 `try` 块中使用资源，最后在 `finally` 块中释放资源。这和 `Java` 是一致的。

6. 和 `Scala` 中的大多数控制结构一样，`try-catch-finally` 最终返回一个值：
   - 如果没有异常抛出，则 `try` 子句的结果就是整个表达式的结果。

     如果子句是多行表达式，则最后一个表达式的结果就是整个表达式的结果。
   - 如果有异常抛出，且被 `catch` 子句捕获，则该 `catch` 子句的结果就是整个表达式的结果。
   - 如果有异常抛出，且没有被 `catch` 子句捕获，则整个表达式就没有结果。
   - 如果有 `finally` 子句，则该子句计算出来的值会被丢弃。因此该子句一般执行清理工作，且不应该改变主体代码或者 `catch` 子句中计算出来的值。

     当 `finally` 子句包含一个**显式**的 `return` 语句或者抛出某个异常，则该返回值或者异常会“改写”任何在之前的 `try` 代码块或者 `catch` 子句中产生的值。

     ```
     def f1():Int = try return 1 finally return 2 // 调用 f1() 返回 2
     def f2():Int = try return 1 finally 2        // 调用 f2() 返回 1
     ```

与 `Scala` 不同，`Java` 的 `try-finally` 并不返回值。

2.7 表达式&语句

1. 与 `Java` 一样，在 `Scala` 中的 `while/if` 语句中的 `boolean` 表达式必须放在圆括号里，不能像 `python` 一样写 `if x<0` 。
2. 和 `Java` 一样，在 `Scala` 中如果 `if` 代码块只有一条语句，则可以选择不写花括号 `{}` 。
3. `Scala` 支持使用分号 `;` 来分隔语句，但是 `Scala` 通常都不写分号。
4. `Scala` 推荐使用 `foreach` 来代替 `while` 循环，因为 `foreach` 是函数式风格，而 `while` 是指令式风格。如：

```
var i =0
while(i<args.length)
{
    println(args(i))
    i+=1
}
```

采用 `foreach` 替代为：

```
args.foreach((arg:String) => println(arg))
```

其中 `(arg:String) => println(arg)` 是一个函数字面量，可以进一步简化为：`args.foreach(println)` 。

5. `Scala` 不支持 `for` 循环语句，但是支持 `for` 表达式。其用法为：

```
for (arg <- args )
  println(arg)
```

其中 `arg` 是一个 `val` 变量，这确保它无法在循环体被被重新赋值。

2.8 输入输出

1. `Console.in,Console.out,Console.err` 分别为标准输入流、标准输出流、标准异常流对象。
   - 输出：`Console.out/err` 的 `print/printf/println` 等方法。
   - 输入：`Console.in` 的 `read/readLine` 等方法。
2. `scala.io.Source` 类提供了文件 `IO` 的方法。
   - `Source.fromeFile(filename)`：打开指定的文件并返回一个 `Source` 对象。
   - `source.getLines()`：读取 `source` 对象指向的文件，并返回一个迭代器 `Iterator[String]` 。

2.9 作用域

1. `Scala` 的变量作用域和 `Java` 几乎完全一样，一个区别是：`Scala` 允许你在嵌套的作用域内定义同名变量。
   - 花括号 `{}` 一般都会引入一个新的作用域，因此任何在花括号中定义的元素都会在右花括号 `}` 之后离开作用域。

- 函数中定义的所有变量都是局部变量。这些变量在定义它们的函数内部有效。函数每次被调用时，都会使用全新的局部变量。
- 变量一旦定义好，就不能在相同作用域内定义相同名字的变量。但是可以在嵌套的作用域内定义一个跟外部作用域中同名的变量。内嵌作用域中的变量会屏蔽外部作用域中的同名变量。

  一个良好的编程习惯是：在内嵌作用域内选择一个新的、有意义的变量名，而不是和外部作用域中的变量同名。
- 在解释器中，可以随心所欲的使用变量名，理论上解释器会对你录入的每一条语句创建一个新的作用域。

三、For 表达式

1. 所有给出 `yield` 结果的 `for` 表达式都会被编译器翻译成对高阶函数 `map,flatMap` 和 `withFilter` 的调用；所有不带 `yield` 结果的 `for` 表达式都被翻译成更小范围（只有 `withFilter` 和 `foreach` ）的高阶函数。

```
case class Person(name:String, isMale:Boolean, children:Person*)
// 找出所有母亲和其孩子的 pair 对
persons.filter(p => !p.isMale).flatMap{
  case p => p.children.map( c => (p.name,c.name))
}
// 或者
persons.withFilter(p => !p.isMale).flatMap.flatMap{
  case p=> p.children.map(c => (p.name,c.name))
}
// 或者
for (p <- persons
     if !p.isMale ;
     c<- p.children)
  yield (p.name,c.name)
```

2. 一般而言一个 `for` 表达式的格式为：

```
for (seq) yield expr
```

`seq` 是一个序列的生成器`generator`、定义 `definition` 和过滤器 `filter` ，它们用分号隔开。

如：

```
for (p <- persons; n = p.name; if(n.startsWith("To")))
  yield n
```

其中 `p <- persons` 表示生成器， `n = p.name` 表示定义， `if(n.startsWith("To"))` 表示过滤器。

也可以将 `seq` 放到花括号而不是圆括号中，此时分号就变成可选的：

```
for {
  p <- persons // 一个生成器
  n = p.name;   // 一个定义
  if(n.startsWith("To")) // 一个过滤器
} yield n
```

- 生成器 `generator` 的格式为：

  ```
  pat <- expr
  ```

  这里的表达式 `expr` 通常返回一个列表，然后模式 `pat` 会跟这个列表里的每个元素依次匹配。如果匹配成功，则模式中的变量就会被绑定上该元素对应的部分；如果匹配失败，则列表的当前元素就被丢弃，并不会抛出 `MatchError` 异常。

  最常见的情况下，模式 `pat` 只是一个变量 `x`，如 `x <- expr` 。此时变量 `x` 仅仅是简单的遍历 `expr` 返回的所有元素。

- 定义 `definition` 的格式为：

  ```
  pat = expr
  ```

  这个定义将模式 `pat` 绑定到 `expr` 的值，因此跟如下的 `val` 定义的作用是一样的：

  ```
  val x = expr
  ```

  最简单的情况是简单的变量 `x` 。

- 过滤器 `filter` 的格式为：

  ```
  if expr
  ```

  这里 `expr` 是个类型为 `Boolean` 的表达式。过滤器会将迭代中所有让 `expr` 返回 `false` 的元素丢弃。

3. 每个 `for` 表达式都以生成器开始。如果一个 `for` 表达式中存在多个生成器，则出现在后面的生成器比出现在前面的生成器调用得更频繁。
4. 组合数学问题是 `for` 表达式特别适合的应用领域。

3.1 翻译

1. 每个 `for` 表达式都可以用三个高阶函数 `map、flatMap、withFilter` 来表示。
   - 单个生成器的 `for` 表达式：

     ```
     for (x <- expr1) yield expr2
     ```

     等价于：

     ```
     expr1.map(x => expr2)
     ```

   - 带一个生成器和一个过滤器的 `for` 表达式：

```
for(x <- expr1 if expr2) yield expr3
```

等价于：

```
for( x <- expr1 withFilter (x => expr2)) yield expr3
```

这进一步等价于：

```
expr1.withFilter(x => expr2).map(x => expr3)
```

- 相同的翻译机制对于过滤器后面更多元素也同样适用。如果 `seq` 是一组任意的生成器、定义和过滤器序列，则有：

```
for (x <- expr1 if expr2; seq) yield expr3
```

等价于：

```
for(x <- expr1 withFilter expr2; seq) yield expr3
```

然后翻译过程继续处理第二个表达式，这个表达式已经比原始版本少了一个元素。

- 如果有两个生成器开始的 `for` 表达式，则等价于对 `flatMap` 的应用。如：

```
for(x <- expr1; y<- expr2; seq) yield expr3
```

等价于：

```
expr1.flatMap(x => for (y <- expr2; seq) yield expr3 )
```

这里传递给 `flatMap` 的函数值当中，会有另一个 `for` 表达式。这个 `for` 表达式比原始版本少一个元素，因此更简单。它也会按照相同的规则翻译。

2. 如果生成器左边的部分是模式 `pat` 而不是简单变量时，翻译机制变得复杂。

- 如果 `for` 表达式绑定一个元组的情况相对比较容易处理，此时跟单变量的规则几乎相同。

  如：

```
for ((x1,..., xn) <- expr1 ) yield expr2
```

被翻译为：

```
expr1.map{ case (x1,...,xn) => expr2}
```

- 如果生成器左边的部分是一个任意的模式 `pat` 而不是单个变量或者元组时，情况更复杂。

  如：

```
for (pat <- expr1) yield expr2
```

被翻译为：

```
expr1 withFilter {
  case pat => true
  case _ => false
} map{
  case pat => expr2
}
```

也就是说，生成的项首先会被过滤，只有那些跟 `pat` 匹配的项才能进入下一步处理。因此，采用模式匹配的生成器不会抛出 `MatchError` 。

- 这里的机制仅处理包含单个模式匹配的生成器的 `for` 表达式的 `case` 。如果 `for` 表达式包含了其它生成器、过滤器或定义，编译器也有类似的规则来处理。

3. 当 `for` 表达式中包含内嵌定义时，如：

```
for (x <- expr1; y = expr2; seq) yield expr3
```

我们假设 `seq` 是一个（或者为空的）生成器、定义和过滤器的序列。则上述表达式被翻译为：

```
for ((x,y) <- for (x <- expr1) yield (x,expr2); seq) yield epxr3
```

可以看到：每当新的 `x` 值生成出来时， `expr2` 就会被重新求值。这个重新求值操作是必要的，因为 `expr2` 可能会用到 `x` ，因此需要针对 `x` 值的变化重新求值。

因此可以看到：如果 `y` 的值在 `for` 循环内是不变的，则没必要在 `for` 表达式内部内嵌该定义。如：

```
for (x <- expr1; y = func();) yield x + y
```

其中 `func` 和 `x` 无关，且非常耗时。由于在 `for` 循环过程中 `func` 结果不变，且又非常耗时，因此可以修改为：

```
y = func()
for (x <- expr1) yield x + y
```

4. 如果 `for` 循环只是简单的执行副作用，但是并不返回任何值（没有 `yield` 表达式），则其翻译也是类似的。

从原理上讲，之前的翻译机制中用到 `map` 和 `flatMap` 的地方，这里都用 `foreach` 。

如：

```
for ( x <- expr1) body
```

等价于：

```
expr1 foreach (x => body)
```

如：

```
for (x <- expr1; if expr2; y <- expr3) body
```

等价于：

```
expr1 withFilter (x => expr2) foreach ( x=> expr3 foreach (y => body ) )
```

5. 事实上，也可以把高阶函数翻译成 `for` 表达式：每个 `map`、`flatMap`、`filter` 的应用也可以由 `for` 表达式来表示。

```
object Demo{
  def map[A, B](xs: List[A], f: A=> B) : List[B] = for (x <- xs) yield
f(x)
  def flatMap[A, B](xs: List[A], f: A => List[B]) : List[B] =
    for (x <- xs; y <- f(x)) yield y
  def filter[A](xs: List[A], p: A => Boolean): List[A] = for ( x <- xs if
p(x)) yield x
}
```

因此不难看出：`for` 表达式就是对 `map,flatMap,withFilter` 这三个函数的等效表达。

3.2 **泛化** for

1. 由于编译器对 `for` 表达式的翻译仅依赖于相应的 `map,flatMap,withFilter` 方法，因此我们可以对很多数据类型应用 `for` 表达式。
   - 除了列表、数组之外，`Scala` 标准类库中还有很多类型支持 `map,flatMap,withFilter,foreach` 方法，因此允许对它们使用 `for` 表达式。
   - 对于自定义数据类型，你也可以通过支持这四个方法来支持 `for` 表达式。你也可以仅支持其中的一部分方法，从而部分的支持 `for` 表达式。
     - 如果你的类型仅定义了 `map` 方法，则你可以对该类型的对象应用包含单个生成器的 `for` 表达式。
     - 如果你的类型同时定义了 `map, flatMap` 方法，则你可以对该类型的对象应用包含单个或者多个生成器的 `for` 表达式。
     - 如果你的类型仅定义了 `foreach` 方法，则你可以对该类型的对象应用 `for` 循环（而不是表达式），此时可以支持单个生成器或者多个生成器。
     - 如果你的类型定义了 `withFilter` 方法，则你可以对该类型的对象应用的 `for` 循环或者 `for` 表达式中存在 `if` 过滤器。
2. `for` 表达式的翻译发生在类型检查之前。`Scala` 对于 `for` 表达式本身没有规定任何类型规则，也不要求 `map,flatMap,withFilter,foreach` 有任何特定的类型签名。

   如：

```
abstract class C[A] {
  def map[B](f: A => B): C[B]
  def flatMap[B](f: A => C[B]): C[B]
  def withFilter(p: A => Boolean): C[A]
  def foreach(b: A=> Unit): Unit
}
```

在这里 `withFilter` 产出相同类的新集合，这意味着每次 `withFilter` 调用都会创建新的 `C` 对象，就跟 `filter` 做的事情一样。

如果 `withFilter` 创建的对象会被接下来某个方法再次解开，且 `C` 对象很大（比如很长的一个字符串），则你可能希望避免创建这个中间对象。一个标准的做法是：让 `withFilter` 不要返回 `C` 的对象，而是返回一个“记住”这个元素需要被过滤的包装器，然后继续处理。