Scala语法基础

2018-12-17

Scala语法较复杂，参考软件的增量开发，学习一门编程语言也应先找到一种能够驾驭的表达方式，之后再逐步添枝加叶。Scala同时支持面向对象和函数式编程，是其语法复杂的原因之一。一些教程非常全面，但也因为全面，导致难以抽丝剥茧，抓住主干。

以下内容关注最简单的基础语法，希望根据这些内容，可以尝试编写面向对象风格的Scala代码。

语句

Scala允许语句结尾不加;，这一点类似JavaScript。

变量定义

val定义不可变变量（常量），var定义可变变量：

```
val msg1 = "Hello World"
var msg2 = "Hello Wrold"

val msg3: String = "Hello World"
```

定义变量时，类型声明在变量右侧，而且是可选的，可以不声明，编译器会自动推断。Scala中的基本类型包括：

Byte、Short、Int、Long、Char、String、Float、Double、Boolean

函数定义

函数即方法，下面是定义函数的例子：

```
def max(x: Int, y: Int): Int = {
  if (x > y) {
    return x
  } else {
    return y
  }
}
```

与Java中方法定义的显著区别有三处：一是使用def关键字定义函数；二是类型声明在变量右侧，上文已提及；三是函数声明和函数体中间使用=连接。

注意函数声明的参数必须明确定义类型，编译器无法自动推断入参类型。返回类型则是可选的，除非函数使用了递归。另外，return关键字也是可选的，如果没有显式的返回语句，程序会将最后一次运算结果作为返回。

当然if后是单个语句也可以不使用大括号，因此该函数还可以这样描述：

```
def max2(x: Int, y: Int) = if (x > y) x else y
```

选择结构

上面的示例已经用到了if语句，Scala的if语句并无特殊之处，不过与其他语言相比，Scala用模式匹配的概念代替传统的switch结构：

```
val a = 1
```

```
a match {
  case 1 => println(1)
  case 2 => println(2)
  case _ =>
}
```

_通配符匹配所有值，用于捕获默认情况。匹配表达式中，备选项永远不会掉到下一个case，因此不需要break或return。（如果将_放到首句，程序不会继续向下执行）。但是要小心，如果程序没有匹配到选项，会抛出MatchError。

循环结构

while循环并不是Scala推荐的代码风格：

```
var i = 0
while (i < 5) {
  println(i)
  i += 1
}
```

似乎并没有难以理解的地方，这就是典型的while循环。与指令式语言相比，Scala没有++运算符，只能使用`i += 1`这样的语句。

提起while，就一定会想到for。Scala中的for循环与指令式语言有一些差异，简单的示例如下，程序会从0打印直到5（不包括5）。

```
for (i <- 0 until 5) {
  println(i)
}
```

Scala不推荐while循环，而更倾向于函数式的编程风格，用于遍历的foreach方法就是其一：

```
"abc".foreach(c => println(c))
```

程序会依次换行打印出a b c三个字符。如果函数体只有一行语句并只有一个参数，这行代码还可以更简洁：

```
"abc".foreach(println)
```

数组

Scala的数组并不在语言层面实现，可以实例化Array类来使用。相应的，数组下标使用小括号（也就是方法参数）表示：

```
val greet  = new Array[String](3)

greet(0) = "a"
greet(1) = "b"
greet(2) = "c"

greet.foreach(println)
```

实例化对象时，也可以直接传入默认参数。Array确实只是一个普通的类，下面的书写方式并没有黑魔法，只是用到了样本类。关于样本类，后文有提及。

```
val greet2 = Array("a", "b", "c")
greet2.foreach(println)
```

类

类使用class关键字定义，类中也包含字段和方法，即典型的面向对象。与Python不同，Scala仍然支持权限控制：

```
class Accumulator {
  private var sum = 0
  def add(b: Byte): Unit = {
    sum += b
    println(sum)
  }
}
```

单例对象

单例对象（Singleton对象）相当于Java中的静态类，使用object替代class关键字定义。单例对象由程序共享，可直接调用。单例对象可以作为程序入口，即将main方法定义在单例对象中。下面的程序从上面定义的Accumulator类中实例化出对象c，并调用其add方法，最终程序打印1：

```
object Run {
  def main(args: Array[String]): Unit = {
    val a = new Accumulator
    a.add(1)
  }
}
```

在同一源文件中，当单例对象和类同名时，称单例对象为类的伴生对象，类为单例对象的伴生类。类可以访问其伴生对象的私有属性和方法。

构造方法

Scala中构造方法的规则比Java要严格。Scala通过类参数的概念来实现构造方法：

```
class Accumulator(a: Int, b: Int)
```

如果类没有主体，大括号是可以省略的。实例化这个类时，就需要传入参数。在Java中的构造方法重载，对应Scala中的辅助构造器，它看起来像这样：

```
class Accumulator(a: Int, b: Int) {
  def this(c: Int) = this(c, 1)
}
```

这时类拥有两个构造方法：

```
val a1 = new Accumulator(1)
val a2 = new Accumulator(1, 2)
```

Scala构造器的严格之处就在于，第二个构造器只能借助第一个或超类的构造器。

继承与重写

Scala的继承与Java没有明显差异，只是方法重写必须要使用override关键字：

```
class A(a: Int) {
  def test = println("a")
}

class B(b: Int) extends A(b) {
  override def test = println("b")
}
```

特质

特质（trait）和单例对象相像，除了定义时使用的关键字不同，其余和普通的类一样，可以包含字段和方法。特质的意义在于，支持混入（Mixins），并且允许混入多个特质。这一特性经常和多重继承进行对比。

```
trait A {
  def aMethod = println("A")
}

trait B {
  def bMethod = println("B")
}

class C extends A with B
```

这样C的实例就可以调用aMethod和bMethod：

```
val c = new C
c.aMethod
c.bMethod
```

样本类

样本类的定义要在class前加case关键字，即类在定义时用case修饰。这种修饰可以让Scala编译器自动为类添加一些便捷设定：1. 实例化可以省略new关键字；2. 自动将参数作为类字段；3. 自动为类添加toString、hashCode和equals：

```
case class A(a: Int) {
  def aMethod = println(a)
}

object Run {
  def main(args: Array[String]): Unit = {
    val a = A(1)
    a.aMethod     // 1
    println(a)    // A(1)
    println(a.a)  // 1
  }
}
```

其他

与Java相比，Scala支持抽象类，但不支持接口，抽象类使用abstract定义，接口则由特质代替。Scala同样支持泛型、注解等语法。

后续

以上内容并不全面，也许并不够用。使用一种编程语言，除了掌握它的基本语法外，还要熟悉它的惯用写法，尤其像Scala这种多范式的编程语言。之后会持续修改完善此篇内容，也将继续讨论Scala的其他语言特性。