在线实验，请到PC端体验

类和对象（二）

一、实验介绍

1.1 实验内容

本文件就是講類的實際例子

有了前面的 Scala 的基本知识，本实验将介绍如何定义完整功能的 Scala 类定义。

本实验着重介绍如何定义 Functional objects（函数化对象或是方程化对象），函数化对象指的是所定义的类或对象不包含任何可以修改的状态。

本实验定义了一个有理数类定义的几个不同版本，以介绍 Scala 类定义的几个特性：类参数和构造函数，方法，操作符，私有成员，重载，过载，条件检查，引用自身。

1.2 实验知识点

- 类的定义规范
- 定义类
- 前提条件检查
- 添加成员变量
- 自身引用
- 辅助构造函数
- 私有成员变量和方法
- 定义运算符
- 标识符
- 方法重载
- 隐式类型转换

1.3 实验环境

- Scala 2.11.7
- Xfce 终端

1.4 适合人群

本课程难度为一般，属于初级级别课程，适合零基础或具有 Java 编程基础的用户。

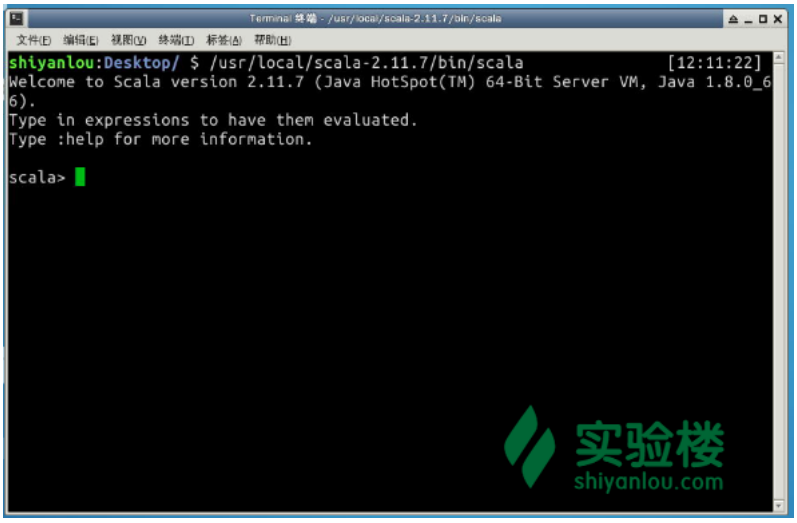
二、开发准备

为了使用交互式 Scala 解释器，你可以在打开的终端中输入命令：

```
cd /usr/local/scala-2.11.7/bin/

scala
```

当出现 scala> 开始的命令行提示符时，就说明你已经成功进入解释器了。如下图所示。



三、实验步骤（有理数类的表示）

3.1 Rational 类的定义规范

首先，我们回忆下有理数的定义：一个有理数(rational)可以表示成个分数形式： `n/d` ，其中 `n` 和 `d` 都是整数（ `d` 不可以为 `0` ），`n` 称为分子(numberator)，`d` 为分母(denominator)。和浮点数相比，有理数可以精确表达一个分数，而不会有误差。

因此我们定义的 `Rational` 类支持上面的有理数的定义。支持有理数的加减乘除，并支持有理数的规范表示，比如 `2/10` ，其规范表示为 `1/5` 。分子和分母的最小公倍数为 `1` 。

3.2 定义 Rational

有了有理数定义的实现规范，我们可以开始设计类 `Rational` 。一个好的起点是考虑用户如何使用这个类，我们已经决定使用 “Immutable” 方式来使用 `Rational` 对象，我们需要用户在定义 `Rational` 对象时提供分子和分母。因此我们可以开始定义 Rational 类如下：

```
class Rational( n:Int, d:Int)
```

可以看到，和 Java 不同的是，Scala 的类定义可以有参数，称为 类参数 ，如上面的 `n` 、 `d` 。 Scala 使用类参数，并把类定义和主构造函数合并在一起，在定义类的同时也定义了类的主构造函数。因此 Scala 的类定义相对要简洁些。

Scala 编译器会编译 Scala 类定义包含的任何不属于类成员和类方法的其它代码，这些代码将作为类的主构造函数。比如，我们定义一条打印消息作为类定义的代码：

```
scala> class Rational (n:Int, d:Int) {
     |    println("Created " + n + "/" +d)
     | }
defined class Rational

scala> new Rational(1,2)
Created 1/2
res0: Rational = Rational@22f34036
```

可以看到创建 `Ratiaonal` 对象时，自动执行类定义的代码（主构造函数）。

3.3 重新定义类的toString 方法

上面的代码创建 `Rational(1，2)` ，Scala 编译器打印出 `Rational@22f34036` ，这是因为使用了缺省的类的 `toString()` 定义（ `Object` 对象的），缺省实现是打印出对象的类名称 +

动手实践是学习 IT 技术最有效的方式！ 开始实验

+ 16进制数（对象的地址），显示结果不是很直观，因此我们可以重新定义类的 `toString()` 方法以显示更有意义的字符。

在 Scala 中，你也可以使用 `override` 来重载基类定义的方法，而且必须使用 `override` 关键字表示重新定义基类中的成员。比如：

```
scala> class Rational (n:Int, d:Int) {
     |    override def toString = n + "/" +d
     | }
defined class Rational

scala> val x= new Rational(1,3)
x: Rational = 1/3

scala> val y=new Rational(5,7)
y: Rational = 5/7
```

3.4 前提条件检查

前面说过有理数可以表示为 `n/d` (其中 `d`、 `n` 为整数，而 `d` 不能为 `0`)。对于前面的 `Rational` 定义 ，我们如果使用 `0` ，也是可以的。

```
scala> new Rational(5,0)
res0: Rational = 5/0
```

怎么解决分母不能为 `0` 的问题呢？面向对象编程的一个优点是实现了数据的封装，你可以确保在其生命周期过程中是有效的。对于有理数的一个前提条件是分母不可以为 `0` ， Scala 中定义为传入构造函数和方法的参数的限制范围，也就是调用这些函数或方法的调用者需要满足的条件。 Scala 中解决这个问题的一个方法是使用 `require` 方法（ `require` 方法为 `Predef` 对象的定义的一个方法，Scala 环境自动载入这个类的定义，因此无需使用 `import` 引入这个对象），因此修改 `Rational` 定义如下：

```
scala> class Rational (n:Int, d:Int) {
     |    require(d!=0)
     |    override def toString = n + "/" +d
     | }
defined class Rational

scala> new Rational(5,0)
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:211)
  ... 33 elided
```

可以看到，如果再使用 `0` 作为分母，系统将抛出 `IllegalArgumentException` 异常。

3.5 添加成员变量

前面我们定义了 Rational 的主构造函数，并检查了输入不允许分母为 0 。下面我们就可以开始实行两个 Rational 对象相加的操作。我们需要实现的函数化对象，因此 Rational 的加法操作应该是返回一个新的 Rational 对象，而不是返回被相加的对象本身。我们很可能写出如下的实现：

```
class Rational (n:Int, d:Int) {
   require(d!=0)
   override def toString = n + "/" +d
   def add(that:Rational) : Rational =
     new Rational(n*that.d + that.n*d,d*that.d)
}
```

实际上编译器会给出如下编译错误：

```
<console>:11: error: value d is not a member of Rational
           new Rational(n*that.d + that.n*d,d*that.d)
                                ^
<console>:11: error: value d is not a member of Rational
           new Rational(n*that.d + that.n*d,d*that.d)
```

这是为什么呢？尽管类参数在新定义的函数的访问范围之内，但仅限于定义类的方法本身(比如之前定义的 toString 方法，可以直接访问类参数），但对于 that 来说，无法使用 that.d 来访问 d 。因为 that 不在定义的类可以访问的范围之内。此时需要定类的成员变量。（注：后面定义的 case class 类型编译器自动把类参数定义为类的属性，这是可以使用 that.d 等来访问类参数）。

修改 Rational 定义，使用成员变量定义如下：

```
class Rational (n:Int, d:Int) {
   require(d!=0)
   val number =n
   val denom =d
   override def toString = number + "/" +denom
   def add(that:Rational)  =
     new Rational(
       number * that.denom + that.number* denom,
       denom * that.denom
     )
}
```

要注意的我们这里定义成员变量都使用了 val ，因为我们实现的是 “immutable” 类型的类定义。 number 和 denom 以及 add 都可以不定义类型，Scala 编译器能够根据上下文推算出它们的类型。

```
scala> val oneHalf=new Rational(1,2)
oneHalf: Rational = 1/2

scala> val twoThirds=new Rational(2,3)
twoThirds: Rational = 2/3

scala> oneHalf add twoThirds
res0: Rational = 7/6

scala> oneHalf.number
res1: Int = 1
```

可以看到，这是就可以使用 .number 等来访问类的成员变量。

3.6 自身引用

Scala 也使用 this 来引用当前对象本身，一般来说访问类成员时无需使用 this ，比如实现一个 lessThan 方法，下面两个实现是等效的。

第一种：

```
def lessThan(that:Rational) =
   this.number * that.denom < that.number * this.denom
```

第二种：

```
def lessThan(that:Rational) =
   number * that.denom < that.number * denom
```

但如果需要引用对象自身， this 就无法省略，比如下面实现一个返回两个 Rational 中比较大的一个值的一个实现：

```
def max(that:Rational) =
     if(lessThan(that)) that else this
```

其中的 this 就无法省略。

3.7 辅助构造函数

在定义类时，很多时候需要定义多个构造函数，在 Scala 中，除主构造函数之外的构造函数都称为辅助构造函数（或是从构造函数），比如对于 Rational 类来说，如果定义一个整数，就没有必要指明分母，此时只要整数本身就可以定义这个有理数。我们可以为 Rational 定义一个辅助构造函数， Scala 定义辅助构造函数使用 this(…) 的语法，所有辅助构造函数名称为 this 。

```
def this(n:Int) = this(n,1)
```

所有 Scala 的辅助构造函数的第一个语句都为调用其它构造函数，也就是 this(…) 。被调用的构造函数可以是主构造函数或是其它构造函数（最终会调用主构造函数）。这样使得每个构造函数最终都会调用主构造函数，从而使得主构造函数称为创建类单一入口点。在 Scala 中也只有主构造函数才能调用基类的构造函数，这种限制有它的优点，使得 Scala 构造函数更加简洁和提高一致性。

3.8 私有成员变量和方法

Scala 类定义私有成员的方法也是使用 private 修饰符，为了实现 Rational 的规范化显示，我们需要使用一个求分子和分母的最大公倍数的私有方法 gcd 。同时我们使用一个私有变量 g 来保存最大公倍数，修改 Rational 的定义：

```
scala> class Rational (n:Int, d:Int) {
     |    require(d!=0)
     |    private val g =gcd (n.abs,d.abs)
     |    val number =n/g
     |    val denom =d/g
     |    override def toString = number + "/" +denom
     |    def add(that:Rational)  =
     |      new Rational(
     |        number * that.denom + that.number* denom,
     |         denom * that.denom
     |      )
     |    def this(n:Int) = this(n,1)
     |    private def gcd(a:Int,b:Int):Int =
     |      if(b==0) a else gcd(b, a % b)
     | }
defined class Rational

scala> new Rational ( 66,42)
res0: Rational = 11/7
```

注意 gcd 的定义，因为它是个 回溯 函数，必须定义返回值类型。Scala 会根据成员变量出现的顺序依次初始化它们，因此 g 必须出现在 number 和 denom 之前。

3.9 定义运算符

本篇还将接着上篇 Rational 类，我们使用 add 定义两个 Rational 对象的加法。两个 Rational 加法可以写成 x.add(y) 或者 x add y 。

即使使用 x add y 还是没有 x + y 来得简洁。

我们前面说过，在 Scala 中，运算符（操作符）和普通的方法没有什么区别，任何方法都可以写成操作符的语法。比如上面的 x add y 。

而在 Scala 中对方法的名称也没有什么特别的限制，你可以使用符号作为类方法的名称，比如使用 + 、 - 和 * 等符号。因此我们可以重新定义 Rational 如下：

```
class Rational (n:Int, d:Int) {
   require(d!=0)
   private val g =gcd (n.abs,d.abs)
   val numer =n/g
   val denom =d/g
   override def toString = numer + "/" +denom
   def +(that:Rational)  =
     new Rational(
       numer * that.denom + that.numer* denom,
       denom * that.denom
     )
   def * (that:Rational) =
     new Rational( numer * that.numer, denom * that.denom)
   def this(n:Int) = this(n,1)
   private def gcd(a:Int,b:Int):Int =
     if(b==0) a else gcd(b, a % b)
}
```

这样就可以使用 + 、 * 号来实现 Rational 的加法和乘法。 + 、 * 的优先级是 Scala 预设的，和整数的 + 、 - 、 * 和 / 的优先级一样。下面为使用 Rational 的例子：

```
scala> val x= new Rational(1,2)
x: Rational = 1/2

scala> val y=new Rational(2,3)
y: Rational = 2/3

scala> x+y
res0: Rational = 7/6

scala> x+ x*y
res1: Rational = 5/6
```

从这个例子也可以看出 Scala 语言的扩展性，你使用 Rational 对象就像 Scala 内置的数据类型一样。

3.10 Scala 中的标识符

从前面的例子我们可以看到 Scala 可以使用两种形式的标志符，字符数字和符号。字符数字使用字母或是下划线开头，后面可以接字母或是数字，符号 $ 在 Scala 中也看作为字母。然而以 $ 开头的标识符为保留的Scala编译器产生的标志符使用，应用程序应该避免使用 $ 开始的标识符，以免造成冲突。

Scala 的命名规则采用和 Java 类似的 camel 命名规则（驼峰命名法），首字符小写，比如 toString 。类名的首字符还是使用大写。此外也应该避免使用以下划线结尾的标志符以避免冲突。

符号标志符包含一个或多个符号，如 + 、 : 和 ? 。对于 + 、 ++ 、 ::: 、 <、 ?>、 :-> 之类的符号，Scala 内部实现时会使用转义的标志符。例如对 :-> 使用 $colon$minus$greater 来表示这个符号。因此，如果你需要在 Java 代码中访问 :-> 方法，你需要使用 Scala 的内部名称 $colon$minus$greater 。

混合标志符由字符数字标志符后面跟着一个或多个符号组成，比如 unary_+ 为 Scala 对 + 方法的内部实现时的名称。

字面量标志符为使用 " 定义的字符串，比如 "x" 、 "yield" 。 你可以在 " 之间使用任何有效的 Scala 标志符，Scala 将它们解释为一个 Scala 标志符，一个典型的使用是 Thread 的 yield 方法， 在 Scala 中你不能使用 Thread.yield() 是因为 yield 为 Scala 中的关键字， 你必须使用 Thread."yield"() 来使用这个方法。

3.11 方法重载

和 Java 一样，Scala 也支持方法重载，重载的方法参数类型不同而使用同样的方法名称，比如对于 Rational 对象， + 的对象可以为另外一个 Rational 对象，也可以为一个 Int 对象，此时你可以重载 + 方法以支持和 Int 相加。

```
def + (i:Int) =
     new Rational (numer + i * denom, denom)
```

3.12 隐式类型转换

上面我们定义 Rational 的加法，并重载 + 以支持整数， r + 2 。当如果我们需要 2 + r 如何呢？下面的例子：

```
scala> val x =new Rational(2,3)
x: Rational = 2/3

scala> val y = new Rational(3,7)
y: Rational = 3/7

scala> val z = 4
z: Int = 4

scala> x + z
res0: Rational = 14/3

scala> x + 3
res1: Rational = 11/3

scala> 3 + x
<console>:10: error: overloaded method value + with alternatives:
  (x: Double)Double <and>
  (x: Float)Float <and>
  (x: Long)Long <and>
  (x: Int)Int <and>
  (x: Char)Int <and>
  (x: Short)Int <and>
  (x: Byte)Int <and>
  (x: String)String
 cannot be applied to (Rational)
             3 + x
               ^
```

可以看到 `x + 3` 没有问题，`3 + x` 就报错了，这是因为整数类型不支持和 `Rational` 相加。我们不可能去修改 `Int` 的定义（除非你重写Scala的 `Int` 定义）以支持 `Int` 和 `Rational` 相加。如果你写过 .Net 代码，这可以通过静态扩展方法来实现，Scala 提供了类似的机制来解决这种问题。

如果 `Int` 类型能够根据需要自动转换为 `Rational` 类型，那么 `3 + x` 就可以相加。Scala 通过 `implicit def` 定义一个隐含类型转换，比如定义由整数到 `Rational` 类型的转换如下：

```
implicit def intToRational(x:Int) = new Rational(x)
```

再重新计算 `r + 2` 和 `2 + r` 的例子：

```
scala> val r = new Rational(2,3)
r: Rational = 2/3

scala> r + 2
res0: Rational = 8/3

scala> 2 + r
res1: Rational = 8/3
```

其实此时 `Rational` 的一个 `+` 重载方法是多余的，当Scala计算 `2 + r` ，发现 `2(Int)` 类型没有可以和 `Rational` 对象相加的方法，Scala 环境就检查 `Int` 的隐含类型转换方法是否有合适的类型转换方法，类型转换后的类型支持 `+ r` ，一检查发现定义了由 `Int` 到 `Rational` 的隐含转换方法，就自动调用该方法，把整数转换为 `Rational` 数据类型，然后调用 `Rational` 对象的 `+` 方法。从而实现了 `Rational` 类或是 `Int` 类的扩展。关于 `implicit def` 的详细介绍将由后面的文章来说明，隐含类型转换在设计 Scala 库时非常有用。

四、实验总结

在本节实验里，我们完善了类和对象的相关知识。结合前面已经学习过的第一部分，相信你可以在 Scala 中更好地使用类和对象了。

在任何时候，如果对于类和对象有任何疑问，可以选择在Scala 官网中查阅文档，或者回顾此系列课程，或者在实验楼的讨论区与我们交流。

上一节 (/courses/490/labs/1686/document)　　下一节 (/courses/490/labs/1688/document)

共发布过**6**门课程

CSDN 专家博主，擅长Java ME, Blackberry ,LWUIT , iPhone, Android, Windows Mobile, Mono , Windows Phone 7等平台开发，主页 http://www.imobilebbs.com/

查看老师的所有课程 > (/teacher/164063)

进阶课程

Scala 专题教程 - Case Class和模式匹配 (/courses/514)

Scala 专题教程 - 隐式变换和隐式参数 (/courses/515)

Scala 专题教程 - 抽象成员 (/courses/516)

Scala 专题教程 - Extractor (/courses/526)

动手做实验，轻松学IT

(http://weibo.com/shiyanlou2013)

公司

关于我们 (/aboutus)
联系我们 (/contact)
加入我们 (http://www.simplecloud.cn/jobs.html)
技术博客 (https://blog.shiyanlou.com)

合作

我要投稿 (/contribute)
教师合作 (/labs)
高校合作 (/edu/)
友情链接 (/friends)
开发者 (/developer)

服务

企业版 (/saas)
实战训练营 (/bootcamp/)
会员服务 (/vip)
实验报告 (/courses/reports)
常见问题 (/questions/?tag=%E5%B8%B8%E8%A7%81%E9%97%AE%E9%A2%98)
隐私条款 (/privacy)

学习路径

Python学习路径 (/paths/python)
Linux学习路径 (/paths/linuxdev)
大数据学习路径 (/paths/bigdata)
Java学习路径 (/paths/java)
PHP学习路径 (/paths/php)
全部 (/paths/)

动手实践是学习 IT 技术最有效的方式！ **开始实验**