在线实验，请到PC端体验

组合和继承（二）

一、实验介绍

1.1 实验内容

前面我们说过，构建新类的两个基本方法是组合和继承。如果你的主要目的是代码重用，那么最好使用组合的方法构造新类。使用继承的方法构造新类造成的可能问题是：无意地修改基类可能会破坏子类的实现。

在本实验中，我们就将探讨这个问题。

1.2 实验知识点

- 使用组合还是继承
- 实现类 Element 的 above，beside 和 toString 方法
- 定义 factory 对象
- 定义 heighten 和 widen 函数

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。

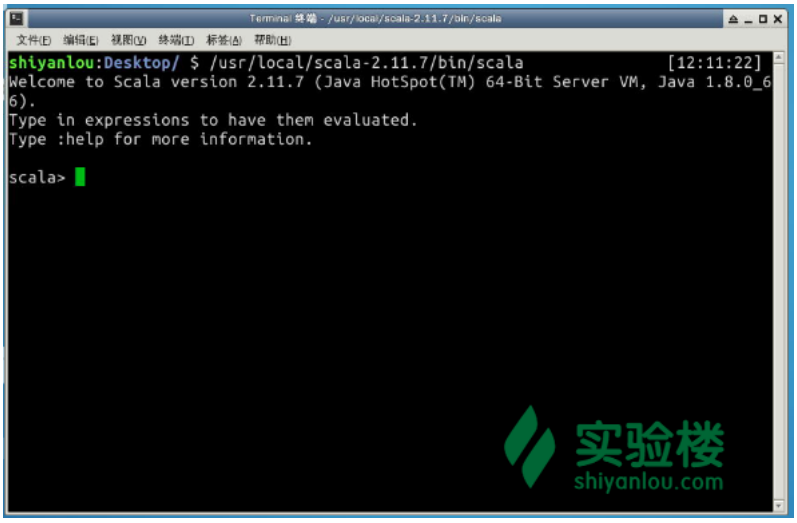
二、开发准备

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。



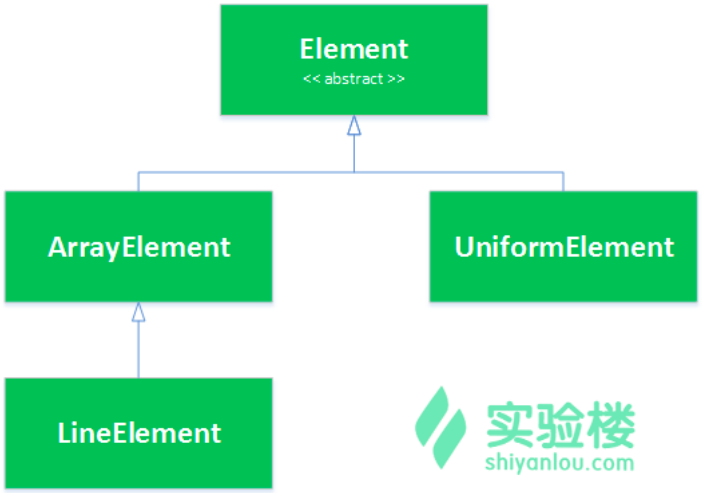
三、实验步骤

3.1 使用组合还是继承

关于继承关系，你可以问自己一个问题，它是否建模了一个 is-a 关系。例如，说 ArrayElement 是 Element 是合理的。你能问的另一个问题是，客户是否想要把子类类型当作基类类型来用。

前一个版本中， LineElement 与 ArrayElement 有一个继承关系，从那里继承了 contents 。现在它在 ArrayElement 的例子里，我们的确期待客户会想要把 ArrayElement 当作 Element 使用。

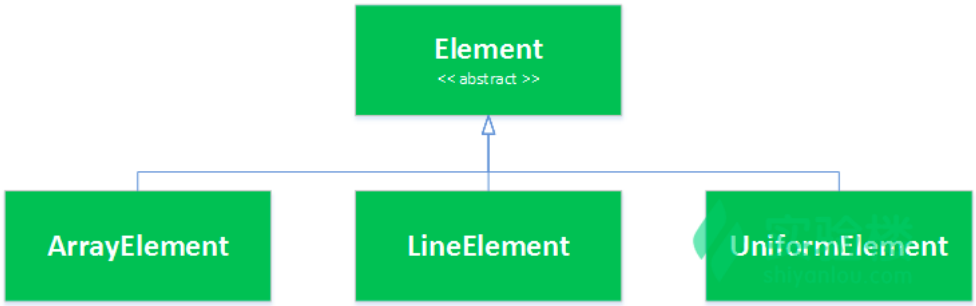
请看下面的类层次关系图：



看着这张图，问问上面的问题，感觉其中有的关系可疑吗？尤其对你而言，说 LineElement 是 ArrayElement 这个论断是否显而易见呢？你是否认为客户会需要把 LineElement 当作 ArrayElement 使用？实际上，我们把 LineElement 定义为 ArrayElement 主要是想重用 ArrayElement 的 contents 定义。因此或许把 LineElement 定义为 Element 的直接子类会更好一些，就像这样：

```
class LineElement(s: String) extends Element {
  val contents = Array(s)
  override def width = s.length
  override def height = 1
}
```

前一个版本中， LineElement 与 ArrayElement 有一个继承关系，从那里继承了 contents 。现在它与 Array 有一个组合关系：在它自己的 contents 字段中持有一个字串数组的引用。有了 LineElement 的这个实现， Element 的继承层级现在如下图所示：



因此在选用组合还是通过继承来构造新类时，需要根据需要选择合适的方法。

3.2 实现类 Element 的 above，beside 和 toString 方法

我们接着实现类 `Element` 的其它方法，如 `above`、 `beside` 和 `toString` 方法。

`above` 方法，意味着把一个布局元素放在另外一个布局元素的上方，也就是把这两个元素的 `contents` 的内容连接起来。我们首先实现 `above` 函数的第一个版本：

```
def above(that: Element) :Element =
    new ArrayElement(this.contents ++ that.contents)
```

Scala 中的 `Array` 通过Java Array 来实现，但添加了很多其它方法，尤其是 Scala 中 `Array` 可以转换为 `scala.Seq` 类的实例对象。 `scala.Seq` 为一个序列结构并提供了许多方法来访问和转换这个序列。

实际上，上面 `above` 的实现不是十分有效，因为它不允许你把不同长度的布局元素叠加到另外一个布局元素上面。但就目前来说，我们还只是暂时使用这个实现，只使用了同样长度的布局元素，后面再提供这个版本的增强版本。

下面我们再实现类 `Element` 的另外一个 `beside` 方法，把两个布局元素并排放置与和前面一样，为简单起见，我们暂时只考虑相同高度的两个布局元素：

```
def beside(that: Element) :Element = {
    val contents = new Array[String](this.contents.length)
    for(i <- 0 until this.contents.length)
      contents(i)=this.contents(i) + that.contents(i)
    new ArrayElement(contents)
  }
```

尽管上面的实现能够满足 `beside` 要求，但采用的还是指令式编程，我们使用函数式编程实现一下，如同下面的简化代码：

```
def beside(that: Element) :Element = {
    new ArrayElement(
        for(
            (line1,line2) <- this.contents zip that.contents
        ) yield line1+line2
    )
}
```

这里我们使用了 `Array` 的 `zip` 操作符，可以用来将两个数组转换成二元组的数组。 `zip` 分别取两个数组对应的元素组成一个新的二元组。比如：

```
scala> Array( 1,2,3) zip Array("a","b")
res0: Array[(Int, String)] = Array((1,a), (2,b))
```

如果一个数组长度大于另外一个数组，多余的元素被忽略。 `for` 的 `yield` 部分用来构成一个新元素。

最后，我们实现 `Element` 的 `toString` 方法，用来显示布局元素的内容：

```
override def toString = contents mkString "\n"
```

这里使用了 `mkString` 函数。这个函数可以应用到任何序列数据结构（包括数组），也就是把 `contents` 的每个元素调用 `toString` ，然后使用 `\n` 分隔。

3.3 定义 factory 对象

到目前为止，我们定义了关于布局元素类的一个层次结构。你可以把包含这个层次关系的类作为 API 接口提供给其它应用。但有时，你可能希望对函数库的用户隐藏这种层次关系。这通常可以使用 factory （构造工厂）对象来实现。

一个 factory 对象定义了用来构造其它对象的函数。库函数的用户可以通过工厂对象来构造新对象，而不需要通过类的构造函数来创建类的实例。

使用工厂对象的好处是，可以统一创建对象的接口并且隐藏被创建对象具体是如何来表示的。这种隐藏可以使得你创建的函数库使用变得更简单和易于理解，也正是隐藏部分实现细节，可以使你有机会修改库的实现而不至于影响库的接口。

实现 factory 对象的一个基本方法，是采用 singleton 模式。在 Scala 中，可以使用类的伴随对象(companion 对象) 来实现。比如：

```
object Element {
  def elem(contents: Array[String]):Element =
   new ArrayElement(contents)

  def elem(chr:Char, width:Int, height:Int) :Element =
    new UniformElement(chr,width,height)

  def elem(line:String) :Element =
    new LineElement(line)
}
```

我们先把之前 Element 的实现列在这里：

```
abstract class Element {
  def contents: Array[String]
  def height: Int = contents.length
  def width: Int = if (height == 0) 0 else contents(0).length
  def above(that: Element) :Element =
    new ArrayElement(this.contents ++ that.contents)
  def beside(that: Element) :Element = {
    new ArrayElement(
      for(
        (line1,line2) <- this.contents zip that.contents
      ) yield line1+line2
    )
  }
  override def toString = contents mkString "\n"
}
```

有了 object Element （类 Element 的伴随对象），我们可以利用 Element 对象提供的 factory 方法，重新实现类 Element 的一些方法：

```
abstract class Element {
  def contents: Array[String]
  def height: Int = contents.length
  def width: Int = if (height == 0) 0 else contents(0).length
  def above(that: Element) :Element =
    Element.elem(this.contents ++ that.contents)
  def beside(that: Element) :Element = {
    Element.elem(
      for(
        (line1,line2) <- this.contents zip that.contents
      ) yield line1+line2
    )
  }
  override def toString = contents mkString "\n"

}
```

这里我们重写了 above 和 beside 方法，使用伴随对象的 factory 方法 Element.elem 替代 new 构造函数。

这样修改之后，库函数的用户不要了解 Element 的继承关系，甚至不需要知道类 ArrayElement、 LineElement 定义的存在。为了避免用户直接使用 ArrayElement 或 LineElement 的构造函数来构造类的实例，我们可以把 ArrayElement、 UniformElement 和 LineElement 定义为私有。定义私有可以也可以把它们定义在类 Element内部 （嵌套类）。下面为这种方法的使用：

```
object Element {

  private class ArrayElement(val contents: Array[String])
    extends Element {
  }

  private class LineElement(s:String) extends ArrayElement(Array(s)) {
    override def width = s.length
    override def height = 1
  }

  private class UniformElement (ch :Char,
    override val width:Int,
    override val height:Int
  ) extends Element{
    private val line=ch.toString * width
    def contents = Array.fill(height)(line)
  }

  def elem(contents: Array[String]):Element =
   new ArrayElement(contents)

  def elem(chr:Char, width:Int, height:Int) :Element =
    new UniformElement(chr,width,height)

  def elem(line:String) :Element =
    new LineElement(line)
}
```

3.4 定义 heighten 和 widen 函数

我们还需要最后一个改进：之前的 `Element` 实现不够完善，只支持同样高度和同样宽度的 `Element` 使用 `above` 和 `beside` 函数。比如，下面的代码将无法正常工作，因为组合元素的第二行比第一行要长：

```
new ArrayElement(Array("hello")) above
new ArrayElement(Array("world!"))
```

与之相似，下面的表达式也不能正常工作。因为第一个 `ArrayElement` 高度为二，而第二个的高度只是一：

```
new ArrayElement(Array("one", "two")) beside
new ArrayElement(Array("one"))
```

下面的代码展示了一个私有帮助方法， `widen` 能够带一个宽度作为参数，并返回那个宽度的 `Element` 。结果包含了这个 `Element` 的内容，以及居中、左侧和右侧留需带的空格，从而获得需要的宽度。这段代码还展示了一个类似的方法， `heighten` 能在竖直方向执行同样的功能。 `widen` 方法被 `above` 调用以确保 `Element` 堆叠在一起有同样的宽度。

类似地， `heighten` 方法被 `beside` 调用以确保靠在一起的元素具有同样的高度。有了这些改变，布局库函数就可以使用了。

```
abstract class Element {
  def contents: Array[String]
  def height: Int = contents.length
  def width: Int = if (height == 0) 0 else contents(0).length
  def above(that: Element) :Element =
    Element.elem(this.contents ++ that.contents)
  def beside(that: Element) :Element = {
    Element.elem(
      for(
        (line1,line2) <- this.contents zip that.contents
      ) yield line1+line2
    )
  }
  def widen(w: Int): Element =
  if (w <= width) this
  else {
    val left = Element.elem(' ', (w - width) / 2, height)
        var right = Element.elem(' ', w - width - left.width, height)
        left beside this beside right
  }

def heighten(h: Int): Element =
  if (h <= height) this
  else {
    val top = Element.elem(' ', width, (h - height) / 2)
        var bot = Element.elem(' ', width, h - height - top.height)
        top above this above bot
  }
  override def toString = contents mkString "\n"

}
```

3.5 小结

在前面的内容中，我们基本完成了布局元素的函数库。现在，我们就可以写个程序来使用这个函数库，下面显示螺旋线的程序如下：

```
object Spiral {
  val space = elem (" ")
  val corner = elem ("+")
  def spiral(nEdges:Int, direction:Int): Element = {
    if(nEdges==1)
      elem("+")
    else{
      val sp=spiral(nEdges -1, (direction +3) % 4)
      def verticalBar = elem ('|',1, sp.height)
      def horizontalBar = elem('-',sp.width,1)
      if(direction==0)
        (corner beside horizontalBar) above (sp beside space)
      else if (direction ==1)
        (sp above space) beside ( corner above verticalBar)
      else if(direction ==2 )
        (space beside sp) above (horizontalBar beside corner)
      else
        (verticalBar above corner) beside (space above sp)
  }
}

 def main(args:Array[String]) {
   val nSides=args(0).toInt
   println(spiral(nSides,0))
 }
}
```

因为 Sprial 是一个单例对象，并包含 `main` 方法，因此它是一个 Scala 应用程序。我们可以在命令行使用 `scala Sprial xx` 来运行这个应用。

```
root@shiyanlou:~/scala# scala Spiral 5
+----
|
| ++
|  |
+--+

root@shiyanlou:~/scala# scala Spiral 23
+----------------------
|
| +------------------+
| |                  |
| | +--------------+ |
| | |              | |
| | | +----------+ | |
| | | |          | | |
| | | | +------+ | | |
| | | | |      | | | |
| | | | | +--+ | | | |
| | | | | |  | | | | |
| | | | | ++ | | | | |
| | | | |    | | | | |
| | | | +----+ | | | |
| | | |        | | | |
| | | +--------+ | | |
| | |            | | |
| | +------------+ | |
| |                | |
| +----------------+ |
|                    |
+--------------------+
```

这个例子的完整代码如下：

动手实践是学习 IT 技术最有效的方式！ **开始实验**

```
object Element {
  private class ArrayElement(val contents: Array[String])
    extends Element

  private class LineElement(s:String) extends Element {
    val contents=Array(s)
    override def width = s.length
    override def height = 1
  }

  private class UniformElement (ch :Char,
    override val width:Int,
    override val height:Int
  ) extends Element{
    private val line=ch.toString * width
    def contents = Array.fill(height)(line)
  }

  def elem(contents: Array[String]):Element =
   new ArrayElement(contents)

  def elem(chr:Char, width:Int, height:Int) :Element =
    new UniformElement(chr,width,height)

  def elem(line:String) :Element =
    new LineElement(line)
  }

import Element.elem

abstract class Element {
  def contents: Array[String]
  def height: Int = contents.length
  def width: Int =  contents(0).length
  def above(that: Element) :Element = {
    val this1=this widen that.width
    val that1=that widen this.width
    elem (this1.contents ++ that1.contents)
  }


  def beside(that: Element) :Element = {
    val this1=this heighten that.height
    val that1=that heighten this.height
    Element.elem(
      for(
        (line1,line2) <- this1.contents zip that1.contents
      ) yield line1+line2
    )
  }

  def widen(w: Int): Element =
  if (w <= width) this
  else {
    val left = Element.elem(' ', (w - width) / 2, height)
        var right = Element.elem(' ', w - width - left.width, height)
        left beside this beside right
  }

def heighten(h: Int): Element =
  if (h <= height) this
  else {
    val top = Element.elem(' ', width, (h - height) / 2)
        var bot = Element.elem(' ', width, h - height - top.height)
        top above this above bot
  }
  override def toString = contents mkString "\n"

}

object Spiral {
  val space = elem (" ")
  val corner = elem ("+")
  def spiral(nEdges:Int, direction:Int): Element = {
```

```
    if(nEdges==1)
      elem("+")
    else{
      val sp=spiral(nEdges -1, (direction +3) % 4)
      def verticalBar = elem ('|',1, sp.height)
      def horizontalBar = elem('-',sp.width,1)
      if(direction==0)
        (corner beside horizontalBar) above (sp beside space)
      else if (direction ==1)
        (sp above space) beside ( corner above verticalBar)
      else if(direction ==2 )
        (space beside sp) above (horizontalBar beside corner)
      else
        (verticalBar above corner) beside (space above sp)
  }
}

 def main(args:Array[String]) {
   val nSides=args(0).toInt
   println(spiral(nSides,0))
 }
}
```

四、实验总结

结合上一个实验，我们通过两个实验的内容完成了组合和继承这一章节的学习。在最后的例子中，我们终于完善了一个 Scala 应用程序。组合和继承是面向对象理论中非常重要的组成部分，然而学习它不仅要通过理论知识，更多的还需要在今后的实践中不断总结其设计思想。

上一节 (/courses/490/labs/1691/document)

下一节 (/courses/490/labs/1693/document)

课程教师

引路蜂

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)