在线实验，请到PC端体验

组合和继承

一、实验介绍

1.1 实验内容

在前面我们介绍了 Scala 面向对象的一些基本概念，从本实验开始，我们将继续介绍 Scala 面向对象方法的知识。

1.2 实验知识点

- 抽象类
- 定义无参数方法
- 扩展类
- 重载成员函数和方法
- 定义参数化成员变量
- 调用基类构造函数
- 使用override修饰符
- 多态和动态绑定
- 定义final成员

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。

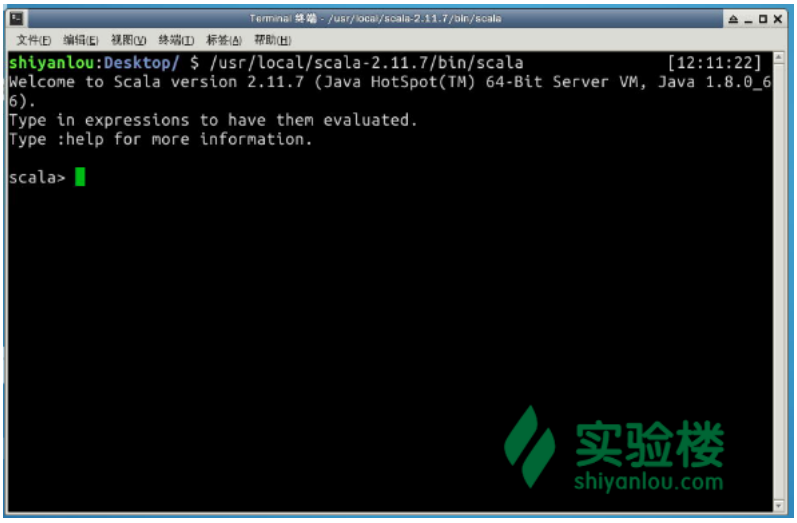
二、开发准备

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 概述

本3.1節是故弄玄虛，看不懂沒事，看了後面的內容就懂了

定义一个新类的方法主要有两种模式：一个通过组合的方式，新创建的类通过引用其它类组合而成；另一个则不是通过这些引用类组合来完成新功能，而是通过继承的方式来扩展基类。

为了更好的介绍 Scala 类的组合和继承，以及抽象类、无参数方法、扩展类、方法的重载等，我们打算使用一个现实的例子来说明。因此本节首先定义需要解答的问题。

我们需要定义一个函数库。这个库用来定义在平面上（二维空间）的布局元素，每个元素使用一个含有文字的矩形来表示。为方便起见，我们定义一个类构造工厂方法 `elem` ，根据传入的数据来创建一个布局元素。

这个方法的接口定义如下：

```
elem(s: String) : Element
```

你可以看到，布局元素使用类型 `Element` 来构造其模型。你可以调用 `above` 和 `beside` 方法来创建一个新的布局元素。这个新的布局元素由两个已经存在的布局元素组合而成。例如：下面的表达式使用多个布局元素构造一个更大区域的布局元素：

```
val column1 = elem(&amp;quot;Hello&amp;quot;) above elem(&amp;quot;***&amp;quot;)
val column2 = elem(&amp;quot;**&amp;quot;) above (&amp;quot;World&amp;quot;)
column beside column2
```

将打印出下面结果：

```
Hello ***
*** world
```

这个例子使用了布局元素。这是一个非常好的例子，可以用来说明一个对象可以由更简单的对象，通过组合的方式来构造。后文将以此为基础。我们将定义一些类，这些类支持使用数组，线段，矩形（简单部件）来构造，并定义组合算子（操作符）`above` 和 `beside` 。

使用组合算子的概念来设计函数库是一种非常好的方法，它是在应用域构建对象的基础方法。什么是简单对象？用什么方式能让更多有趣的对象通过简单对象构造出来？组合子是怎么挂在一起的？什么是最通用的组合？它们满足任何有趣的规则吗？如果你对这些问题都有好的答案，你的库设计就在正轨上了。

3.2 组合和继承

3.2.1 抽象类

上一小节中，我们定义了我们需要解决的问题。我们首要的任务是定义 `Element` 类型，这个类型用来表示一个布局元素。由于每个元素是一个具有二维矩形形状的字符串，因此，我们理所当然的可以定义个成员变量 `contents` ，用它来表示这个二维布局元素的内容。我们使用一个字符串的数组来表示这个元素，这个数组的每个字符串元素代表布局的一行。也就是说， `contents` 的类型为 `Array[String]` 。

```
abstract class Element {
  def contents: Array[String]
}
```

在这个类中，成员 contents 使用了没有定义具体实现的方法来定义，这个方法称为——“抽象方法”。一个含有抽象方法的类必须定义成抽象类，也就是使用 abstract 关键字来定义类。

abstract 修饰符表示所定义的类可能含有一些没有定义具体实现的抽象成员，因此你不能构建抽象类的实例。如果你试图这么做，编译器将报错：

```
scala> new Element
<console>:9: error: class Element is abstract; cannot be instantiated
              new Element
              ^
```

后文将继续介绍如何创建这个抽象类的子类，你可以构造这些子类的具体实例。这是因为这些子类实现了抽象成员。

要注意的是，contents 方法本身没有使用 abstract 修饰符。一个没有定义实现的方法就是抽象方法，与 Java 不同的是，抽象方法不需要使用 abstract 修饰符来表示，只要这个方法没有具体实现，就是抽象方法。相反，如果该方法有具体实现，称为“具体(concrete)”方法。

另一个术语用法需要分辨：声明（ declaration ）和定义（ definition ）。类 Element 声明了抽象方法 contents ，但当前没有定义具体方法。在下一小节，我们要定义一些具体方法来加强 Element 。

3.2.2 定义无参数方法

作为接下来的一步，我们将向 Element 添加显示宽度和高度的方法。 height 方法返回 contents 里的行数。 width 方法返回第一行的长度，或如果元素没有行记录，返回零。

```
abstract class Element {
  def contents: Array[String]
  def height: Int = contents.length
  def width: Int = if (height == 0) 0 else contents(0).length
}
```

请注意 Element 的三个方法没一个有参数列表，甚至连个空列表都没有。这种无参数方法在 Scala 里是非常普通的。相对的，带有空括号的方法定义，如 def height(): Int ，被称为空括号方法(empty-paren method)。

Scala 的惯例是在方法不需要参数并且只是读取对象状态时，使用无参数方法。

此外，我们也可以使用成员变量来定义 width 和 height ，例如：

```
abstract class Element {
  def contents: Array[String]
  val height = contents.length
  val width = if (height == 0) 0 else contents(0).length
}
```

从使用这个类的客户代码来说，这两个实现是等价的。唯一的差别是：使用成员变量的方法调用速度要快些。因为字段值在类被初始化的时候被预计算，而方法调用在每次调用的时候都要计算。换句话说，字段在每个 Element 对象上需要更多的内存空间。

特别是如果类的字段变成了访问函数，且访问函数是纯函数的，也就是说它没有副作用，并且不依赖于可变状态。那么，类Element的客户不需要被重写。这称为统一访问原则： uniform access principle ，也就是说客户代码不应受到通过字段还是方法实现属性的决定的影响。

Scala 代码可以调用 Java 函数和类，而 Java 没有使用“统一访问原则”，因此 Java 里是 string.length() ，不是 string.length 。为了解决这个问题，Scala 对于无参数函数和空括号函数的使用上并不是区分得很严格。也就是说，你可以用空括号方法重载无参数方法，反之亦可。你还可以在调用任何不带参数的方法时省略空的括号。例如，下面两行在 Scala 里都是合法的：

```
Array(1, 2, 3).toString
"abc".length
```

原则上，Scala 的函数调用中可以省略所有的空括号。但如果使用的函数不是纯函数，也就是说这个不带参数的函数可能修改对象的状态或是我们需要利用它的一些副作用（比如打印到屏幕，读写 I/O ），一般的建议还是使用空括号，比如：

```
"hello".length // 没有副作用，所以无须()
println() // 最好别省略()
```

总结起来，Scala 里定义不带参数也没有副作用的方法为无参数方法。也就是说，省略空的括号是鼓励的风格。另一方面，永远不要定义没有括号的带副作用的方法，因为那样的话，方法调用看上去会像选择一个字段。

3.3 扩展类

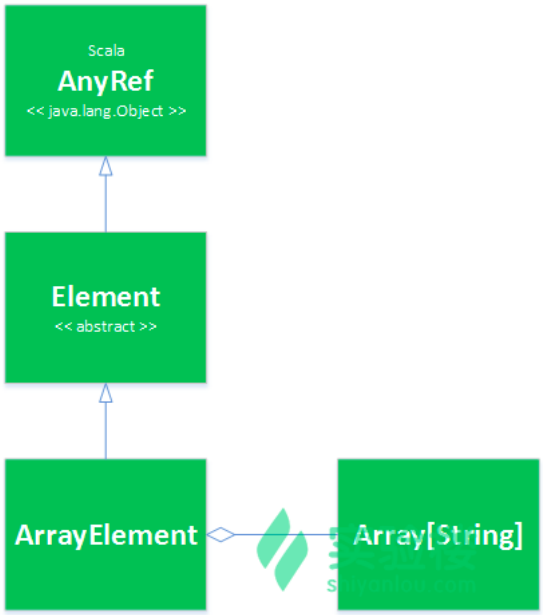
我们需要能够创建新的布局元素对象，前面定义的 `Element` 为抽象类，不能直接用来创建该类的对象。因此，我们需要创建 `Element` 的子类。这些子类需要实现 `Element` 类定义的抽象函数。

Scala 中派生子类的方法和 Java 一样，也是通过 `extends` 关键字。比如定义一个 `ArrayElement` ：

```
class ArrayElement(conts: Array[String]) extends Element {
  def contents: Array[String] = conts
}
```

其中 `extends` 具有两个功效：一是让 `ArrayElement` 继承所有 `Element` 类的非私有成员；二是使得 `ArrayElement` 成为 `Element` 的一个子类。而 `Element` 称为 `ArrayElement` 的父类。

如果你在定义类时没有使用 `extends` 关键字，在 Scala 中，这个定义类默认继承自 `scala.AnyRef` ，如同在 Java 中缺省继承自 `java.lang.Object` 。 这种继承关系如下图：



这幅图中也显示了 `ArrayElement 和 Array[String] 之间的“组合”关系”(composition)，类 ArrayElement 中定义了对 Array[String] 类型对象的一个引用。`

`ArrayElement 继承了 Element 的所有非私有成员，同时定义了一个 contents 函数。` 这个函数中，其父类（基类）中是抽象的，因此可以说 `ArrayElement` 中的 `contents` 函数实现了父类中的这个抽象函数，也可以说“重载”(override)了父类中的同名函数。

`ArrayElement` 继承了 `Element` 的 `width` 和 `height` 方法，因此你可以使用 `ArrayElement.width` 来查询宽度。比如：

```
scala> val ae=new ArrayElement(Array("hello","world"))
ae: ArrayElement = ArrayElement@729c1e43

scala> ae.width
res0: Int = 5
```

派生也意味着子类的值，可以用在任何可以使用同名父类值的地方。比如：

```
val e: Element = new ArrayElement(Array("hello"))
```

动手实践是学习 IT 技术最有效的方式！ 开始实验

3.4 重载成员函数和方法

和 Java 稍有不同的一点是，Scala 中成员函数和成员变量地位几乎相同，而且也处在同一个命名空间。也就是说，Scala 中不允许定义同名的成员函数和成员变量，但带来的一个好处是，可以使用成员变量来重载一个不带参数的成员函数。比如，接着前面的例子，你可以通过一个成员变量来实现基类中定义的抽象函数 `contents` 。

```
class ArrayElement(conts: Array[String]) extends Element {
    val contents: Array[String] = conts
}
```

可以看到，使用成员变量来实现基类中不带参数的抽象函数，是一个非常恰当的例子。Scala 中的这种实现是 Java 语言所不支持的，一般来说只有两个不同的命名空间来定义类，而 Java 可以有四个， Scala 支持的两个命名空间如下：

- 值（字段，方法，包还有单例对象）
- 类型（类和 `Trait` 名）

Scala 把字段和方法放进同一个命名空间的理由很清楚，因为这样做，你就可以使用 `val` 重载无参数的方法。

3.5 定义参数化成员变量

我们回到前面定义的类 `ArrayElement` ，它有一个参数 `conts` ，其唯一的目的，是用来复制到 `contents` 成员变量。而参数名称 `conts` 是为了让它看起来和成员变量 `contents` 类似，而又不至于和成员变量名冲突。

Scala 支持使用参数化成员变量，也就是把参数和成员变量定义合并到一起来避免上述代码：

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

要注意的是，现在参数 `contents` 前面加上了 `val` 关键字，这是前面使用同名参数和同名成员变量的一个缩写形式。使用 `val` 定义了一个无法重新赋值的成员变量。这个成员变量初始值为参数的值，可以在类的外面访问这个成员变量。它的一个等效的实现如下：

```
class ArrayElement(val x123: Array[String])
  extends Element {
   val contents: Array[String] = x123
}
```

Scala 也允许你使用 `var` 关键字来定义参数化成员变量，使用 `var` 定义的成员变量，可以重新赋值。

此外，Scala 也允许你使用 `private`、 `protected` 和 `override` 来修饰参数化成员变量。这与你定义普通的成员变量的用法一样。比如：

```
class Cat {
  val dangerous =false
}

class Tiger (
  override val dangerous: Boolean,
  private var age: Int
) extends Cat
```

这段代码中 `Tiger` 的定义其实为下面类定义的一个缩写：

```
class Tiger(param1: Boolean, param2: Int) extends Cat {
    override val dangerous = param1
    private var age = param2
}
```

两个成员都初始化自相应的参数。我们任意选择了这些参数名，即 `param1` 和 `param2` 。重要的是，它们不会与范围内的任何其它名称冲突。

3.6 调用基类构造函数

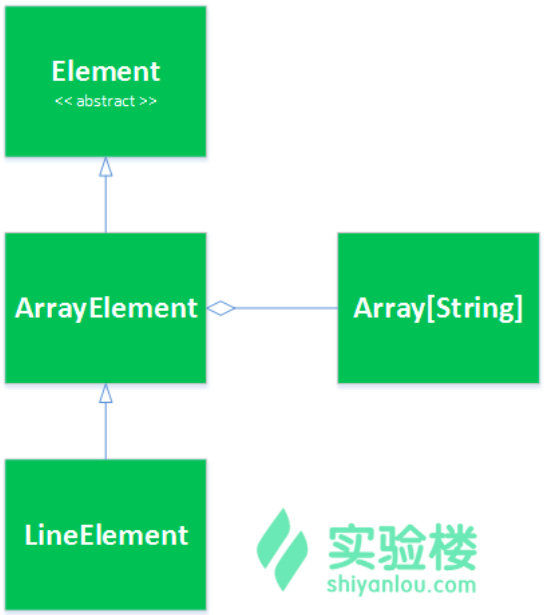
前面我们定义了两个类，一个为抽象类 `Element` ，另外一个为派生的实类 `ArrayElement` 。或许你打算再构造一个新类，这个类使用单个字符串来构造布局元素，使用面向对象的编程方法使得构造这种新类非常容易。比如下面的 LineElement 类：

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

由于 LineElement 扩展了 ArrayElement ，并且 ArrayElement 的构造器带一个参数（ Array[String] ）。 LineElement 需要传递一个参数到它的基类的主构造器。要调用基类构造器，只要把你要传递的参数或参数列表放在基类名之后的括号里即可。例如，类 LineElement 传递了 Array(s) 到 ArrayElement 的主构造器，把它放在基类 ArrayElement 的名称后面的括号里：

```
... extends ArrayElement(Array(s)) ...
```

有了新的子类，布局元素的继承级别现在看起来就如下图所示：



3.7 使用 override 修饰符

在前面的例子中， LineElement 使用了 override 来修饰 width 和 height 成员变量。在 Scala 中需要使用 override 来重载父类的一个非抽象成员，实现抽象成员无需使用 override 。如果子类没有重载父类中的成员，不可以使用 override 修饰符。

这个规则可以帮助编译器发现一些难以发现的错误，可以增强系统安全进化。比如，如果你把 height 拼写错误为 hight ，使用 override 后，编译器会报错：

```
root@shiyanlou:~/scala# scalac demo.scala
demo.scala:13: error: method hight overrides nothing
  override def hight = 1
               ^
one error found
```

这个规则对于系统的严谨尤为重要。假设你定义了一个 2D 图形库，并且你想把它公开和广泛使用。而在库的下一个版本里，你想在你的基类 Shape 里增加一个新方法：

```
def hidden(): Boolean
```

你的新方法将被用在许多画图方法中，去决定是否需要把形状画出来。这将可以大大提高系统绘图的性能，但你不可以冒着破坏客户代码的风险做这件事。毕竟客户说不定已经使用不同的 hidden 实现定义了 Shape 的子类。或许客户的方法实际上是让对象消失而不是检测是否对象是隐藏了。因为这两个版本的 hidden 互相重载，你的画图方法将停止对象的消失，这可真不是你想要的！

如果图形库和它的用户是用 Scala 写的，那么客户的 hidden 原始实现就不会有 override 修饰符，因为这时候还没有另外一个使用那个名字的方法。一旦你添加了 hidden 方法到你 Shape 类的第二个版本，客户的重编译将给出像下列这样的错误：

```
.../Shapes.scala:6: error: error overriding method
       hidden in class Shape of type ()Boolean;
method hidden needs 'override' modifier
def hidden(): Boolean =
```

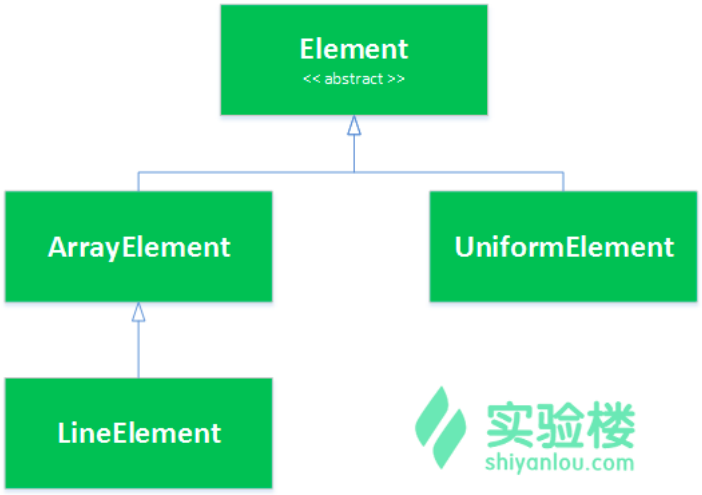
也就是说，源于错误的执行，你的客户将得到一个编译器错误，这常常是不可取的。

3.8 多态和动态绑定

在前面的例子中，我们看到类型为 `Element` 的变量可以保存 `ArrayElement` 类型的对象，这种现象称为“多态”。也就是说，基类类型的变量可以保存其子类类型的对象。到目前为止，我们定义了两个 `Element` 的子类， `ArrayElement` 和 `LineElement` 。你还可以定义其它子类，比如：

```
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)
}
```

结合前面定义的类定义，我们就有了如下图所示的类层次关系：



Scala将接受下列所有的赋值，因为赋值表达式的类型符合定义的变量类型：

```
val e1: Element = new ArrayElement(Array("hello", "world"))
val ae: ArrayElement = new LineElement("hello")
val e2: Element = ae val
e3: Element = new UniformElement('x', 2, 3)
```

若你检查继承层次关系，你会发现：这四个 `val` 定义的每一个表达式，等号右侧表达式的类型都在将被初始化的等号左侧的 `val` 类型的层次之下。

另一方面，如果调用变量（对象）的方法或成员变量，这个过程是一个动态绑定的过程。也就是说，调用哪个类型的方法，取决于运行时变量当前的类型，而不是定义变量的类型。

为了显示这种行为，我们在 `Element` 中添加一个 `demo` 方法，定义如下：

```
abstract class Element {
  def demo() {
    println("Element's implementation invoked")
  }
}

class ArrayElement extends Element {
  override def demo() {
    println("ArrayElement's implementation invoked")
  }
}

class LineElement extends ArrayElement {
  override def demo() {
    println("LineElement's implementation invoked")
  }

}

// UniformElement inherits Element’s demo
class UniformElement extends Element
```

在交互式 Scala 解释器中测试时（例如实验楼提供的环境中），你可以定义如下的方法：

```
def invokeDemo(e: Element) {
  e.demo()
}
```

下面我们分别使用 `ArrayElement`、 `LineElement` 和 `UniformElement` 来调用这个方法：

```
scala> invokeDemo(new ArrayElement)
ArrayElement's implementation invoked

scala> invokeDemo(new LineElement)
LineElement's implementation invoked

scala> invokeDemo(new UniformElement)
Element's implementation invoked
```

可以看到，由于 `ArrayElement` 和 `LineElement` 重载了 `Element` 的 `demo` 方法，因此在调用 `invokeDemo` 时，由于“动态绑定”，这些子类的 `demo` 方法会被调用，而由于 `UniformElement` 没有重载 `Element` 的 `demo` 方法，动态绑定时也会调用 `UniformElement` 的 `demo` 方法（但此时实际为基类的demo方法）。

3.9 定义 final 成员

在定义类的继承关系时，有时你可能不希望基类的某些成员被子类重载。和 Java 类似，在 Scala 中也是使用 `final` 来修饰类的成员。比如，在前面的 `ArrayElement` 例子中，在 demo 方法前加上 `final` 修饰符：

```
class ArrayElement extends Element {
  final override def demo() {
    println("ArrayElement's implementation invoked")
  }
}
```

如果 `LineElement` 试图重载 `demo` ，则会报错：

```
scala> class LineElement extends ArrayElement {
     |   override def demo() {
     |     println("LineElement's implementation invoked")
     |   }
     |
     | }
<console>:10: error: overriding method demo in class ArrayElement of type ()Unit;
 method demo cannot override final member
         override def demo() {
```

如果你希望某个类不可以派生子类，则可以在类定义前加上 `final` 修饰符：

```
final class ArrayElement extends Element {
   override def demo() {
    println("ArrayElement's implementation invoked")
  }
}
```

此时如果还是重载 `LineElement` 的 `demo` 函数，则会报错：

```
scala> class LineElement extends ArrayElement {
     |   override def demo() {
     |     println("LineElement's implementation invoked")
     |   }
     |
     | }
<console>:9: error: illegal inheritance from final class ArrayElement
       class LineElement extends ArrayElement {
```

四、实验总结

在本实验中，我们学习了抽象类的定义、无参数方法的定义、扩展类、重载成员函数和方法、定义参数化成员变量等知识，这些还只是面向对象理论的一部分，我们将在下一节继续为你讲解剩余部分。

上一节 (/courses/490/labs/1690/document)

下一节 (/courses/490/labs/1692/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)