# Scala面向对象

Scala的类定义也是使用class关键字,后面跟类的名称，然后大括号

|  |
| --- |
| **class** Person { } |

## Java类与Scala类的区别

Java类体内：

属性

静态属性

非静态属性

方法

静态方法

非静态方法

静态代码块：类被加载时执行

Scala类体内

属性：非静态属性

方法：非静态方法

非静态代码块

scala中没有static关键词

Scala中用Object(单例类)来承担静态的成员定义

在class定义的属性和方法必须通过实例化的对象才能调用

在Object中定义的属性和方法，直接用Object名称就可以调用

Scala的成员也可以使用private和protected修饰，但是没有public，默认情况下是public

**权限修饰符**

没有：所有其他代码都可以访问

Private：只有自己可以访问，除了自己可以访问之外可以额外开放访问权限

Protected：只有之类和同包下可以访问，除了子类和同包外可以访问之外可以额外开放访问权限

Class里面的非定义(声明代码块)，在类实例化的时候被执行

## 构造方法

Java构造方法

1. 有默认无参构造方法
2. 自定义构造方法时，默认构造方法消失
3. 构造方法可以重载 [public 类名(参数列表){}]
4. 不同的构造方法地位是平等的

Scala构造方法

1. 有无参的默认构造方法
2. 构造方法也可以重载
3. 不同的构造方法之间地位不同，每个scala类都有一个唯一的主构造方法，除了主构造方法之外，所有的次构造方法体内都必须直接或间接的调用主构造方法来完成对象的构建
4. 主构造方法的声明在类声明后面来写，次构造方法是写在类体内的，他们的名字统一都叫this
5. 所有的次构造方法在方法体内必须直接或间接的调用主构造方法才能写自己的构造内容
6. 构造方法不需要返回值，它返回值是Unit类型

### 定义构造方法

|  |
| --- |
| **package** top.xiesen.oo **class** ConstructionTest(pattr1: String, pattr2: String) {  *// 属性1* **var** *attr1* = pattr1  *// 属性2* **var** *attr2* = pattr2   *// 副构造方法* **def this**() = {  **this**(**""**, **""**) *// 直接调用主构造方法  println*(**"------执行了副构造方法-------"**)  **this**.*attr2* = **"副构造方法内赋值"** }   **def this**(pattr1: String) = {  **this**() *//没有直接调用构造方法，但是间接调用了主构造方法  println*(**"----执行了副构造方法二"**)  **this**.*attr1* = pattr1  } }  **object** ConstructionTestObj {  **def** main(args: Array[String]): Unit = {  **val** c1 = **new** ConstructionTest(**"aaa"**, **"123"**)  *println*(**s"c1.attr1: $**{c1.*attr1*}**, c1.attr2: $**{c1.*attr2*}**"**)  **val** c2 = **new** ConstructionTest()  *println*(**s"c2.attr1: $**{c2.*attr1*}**, c2.attr2: $**{c2.*attr2*}**"**)  } } |

### 将主构造方法与属性进行合并

|  |
| --- |
| **package** top.xiesen.oo **class** AttributeDefine(**var** attr1: String, **var** attr2: Int,**val** valAttr: String, **private var** privateAttr: String) {  **var** *embobyAttr*: String = **""** }  **object** AttributeDefineObj {  **def** main(args: Array[String]): Unit = {  **val** ad = **new** AttributeDefine(**"aaa"**,123,**"常量"**,**"private"**)  *println*(**s"ad.attr1: $**{ad.attr1}**, ad.attr2: $**{ad.attr2}**,ad.valAttr: $**{ad.valAttr}**"**)  *// 私有变量下不允许访问 // println(s"ad.private: ${ad.privateAttr}")* ad.*embobyAttr* = **"内部声明的变量"** *println*(**s"ad.embobyAttr: $**{ad.*embobyAttr*}**"**)  } } |

### 使用默认值定义构造方法

构造方法可以使用默认值参数，这样能够大大提高构造方法的灵活性

|  |
| --- |
| **package** top.xiesen.oo */\*\*  \* 构造方法可以使用默认值参数，这样能够大大提高构造方法的灵活性  \*/* **class** ConstructionWithDefine (**var** attr1: String, **var** attr2:String = **"defaultAttr2"**,**var** attr3: Int = 10){   **override def** toString: String = {  **s"attr1: $**{attr1}**,attr2: $**{attr2}**,attr3: $**{attr3}**"** }  }  **object** ConstructionWithDefineObj {  **def** main(args: Array[String]): Unit = {  **val** c1 = **new** ConstructionWithDefine(**"c1参数"**)  *println*(c1)  **val** c2 = **new** ConstructionWithDefine(**"c1参数"**,**"c2参数"**)  *println*(c2)  **val** c3 = **new** ConstructionWithDefine(**"c1参数"**,**"c2参数"**,123)  *println*(c3)  **val** c4 = **new** ConstructionWithDefine(**"c1参数"**,attr3 = 1233)  *println*(c4)  } } |

### 主构造方法私有

可以通过在主构造方法参数前面添加private的方式来对主构造方法进行私有化

|  |
| --- |
| **package** top.xiesen.oo  */\*\*  \* 主构造方法私有  \*/* **class** ConstructionMainPrivate **private**(**var** attr1: String,**var** attr2: String) {   **override def** toString: String = {  **s"attr1:$**{attr1}**,attr2:$**{attr2}**"** }  **def this**() = {  **this**(**""**,**""**)  *println*(**"-----使用公有的辅助构造方法来构建对象-----"**)  } } **object** ConstructionMainPrivateObj{  **def** main(args: Array[String]): Unit = {  **val** c1 = **new** ConstructionMainPrivate()  c1.attr1 = **"属性1"** c1.attr2 = **"属性2"** *println*(c1)  } } |

### 单例对象

单例对象的属性和方法，可以直接通过单例对象的名称来调用，不需要实例化，它本身就是一个对象

|  |
| --- |
| **package** top.xiesen.oo  */\*\*  \* 单例对象  \*/* **object** ObjectTest {   **var** *varAttr1* = **""  private var** *varAttr2*: String = **""  val** *valAttr1* = **"常量1"  def** method1() = {  *println*(**"执行method1"**)  }   **private def** method2() = {  *println*(**"执行method2"**)  }   **def** main(args: Array[String]): Unit = {  ObjectTest.*varAttr1* = **"给单例对象属性赋值"** ObjectTest.*varAttr2* = **"给单例对象私有属性赋值"** *// 常量本省不能赋值 // ObjectTest.valAttr1 = "123"  println*(**s"ObjectTest单例对象---> valAttr1: $**{ObjectTest.*valAttr1*} **,varAttr1: $**{ObjectTest.*varAttr1*}**, varAttr2: $**{ObjectTest.*varAttr2*}**"**)  ObjectTest.*method1*()  ObjectTest.*method2*()  } } |

### 伴生类和伴生对象

如果一个Object和一个class他们的名称相同，那么在编译成class文件的时候，它们会公用一个Student.class文件，这样一个object和class它们互为伴生

在伴生类通过类名可以调用伴生对象的属性和方法，但是伴生对象不可以调用伴生类的属性和方法

伴生类和伴生对象可以互相访问私有成员

但是如果private添加泛型限定符，会额外的限定如private[this]

|  |
| --- |
| **package** top.xiesen.oo  *// student的伴生类* **class** Student(**var** studentNo:String,**var** studentName:String,**var** studentClass:String,**var** age:Int){  **private def** classPrivateMethod() = {  *println*(**"伴生类的私有方法"**)  Student.*objectPrivateMethod* }  **def** printlnStudent() = {  *println*(**s"schoolName: $**{Student.*schoolName*}**, studentNo: $**{studentNo}**,studentName: $**{studentName}**,studentClass: $**{studentClass}**,age: $**{age}**"**)  Student.*gotoSchool*()  }   **private**[**this**] **def** classPrivateThisMethod() = {  *println*(**"class的private[this]方法"**)  }   *// 这里要求泛型需要包对象进行封装，即student和studentTest要封装在同一个包对象中 /\* private[StudentTest] def classPrivateStudentTestMethod() = {  println("class的private[StudentTest]方法")  }\*/* } *// student的伴生对象* **object** Student {  **var** *schoolName*:String = **""  def** gotoSchool() = {  *println*(**"伴生对象的上学方法"**)  }  **private def** objectPrivateMethod() = {  *println*(**"伴生对象的私有方法"**)  }   **def** classStudentPrivateMethod() = {  **val** s = **new** Student(**"002"**,**"私有测试"**,**"二年级"**,15)  s.classPrivateMethod() *// s.classPrivateThisMethod()* }   **def** apply() = {  *println*(**"调用了apply方法"**)  }  **def** apply(studentNo:String,studentName:String,studentClass:String,age:Int) = {  **new** Student(studentNo,studentName,studentClass,age)  }  }  **object** StudentTest {  **def** main(args: Array[String]): Unit = {  **val** student = **new** Student(**"001"**,**"小王"**,**"一年级"**,18)  student.printlnStudent()  Student.*gotoSchool*()   */\*Student.classStudentPrivateMethod()  student.classPrivateStudentTestMethod\*/* Student.*apply*()  *Student*()   **val** student1 = *Student*(**"001"**,**"小王"**,**"一年级"**,18)  } } |

### Apply

Apply方法在scala中是有特殊作用的方法，它可以直接通过Object后面的小括号的形式来调用

Student.apply() 等同于 Student()

Apply方法可以进行重载，我们可以利用这个特征，不使用new就可以创建对象

|  |
| --- |
| **def** apply(studentNo:String,studentName:String,studentClass:String,age:Int) = {  **new** Student(studentNo,studentName,studentClass,age)  }  **val** student1 = *Student*(**"001"**,**"小王"**,**"一年级"**,18) |

## 抽象类

Java抽象类:

1. 不能被实例化
2. 可以定义以实现的方法，也可以定义抽象方法
3. 子类必须实现抽象类中所有的抽象方法

Scala抽象类：

1. 不能被实例化
2. 可以定义属性，可以定义以实现的方法
3. 可以定义未被初始化的属性和未被实现的方法
4. 子类必须实现抽象类中所有的未初始化的属性，必须初始化抽象类中未实现的方法
5. 在定义未实现的方法上必须指定返回值类型

抽象类也可以定义构造方法，构造方法上也可以通过var、val等修饰符声明属性

抽象类主构造方法上属性定义，子类在继承时必须给抽象类的构造方法传值

在子类被实例化的时候先调用父类的构造方法，再调用子类的构造方法

|  |
| --- |
| **package** top.xiesen.oo.abstracttest  **abstract class** Person(**var** name:String) {  *println*(**"Person的构造方法被调用"**)  **var** pType: String   **def** printlnMethod() = {  *println*(**s"pType: $**{pType}**, name:$**name**"**)  }   **def** work(): Unit }  **class** Student(name: String) **extends** Person(name) {  *println*(**"Student的构造方法被调用"**)  **override var** *pType*: String = **"学生类型"   override def** work(): Unit = {  *println*(**"学生的工作是上课"**)  } } **class** Teacher(name:String) **extends** Person (name){  *println*(**"Teacher的构造方法被调用"**)  **override var** *pType*: String = **"教师类型"   override def** work(): Unit = {  *println*(**"教师的工作是授课"**)  } }  **object** PersonTest {  **def** main(args: Array[String]): Unit = {  **val** student = **new** Student(**"小张"**)  **val** teacher = **new** Teacher(**"小李"**)  student.printlnMethod()  student.work()  teacher.printlnMethod()  teacher.work()  } } |

### 方法重写

方法重写指的是当子类继承父类的时候，从父类继承过来的方法不能满足子类的需要，子类希望有自己的实现，这时需要对父类的方法进行重写，方法重写是实现多态和动态绑定的关键。   
scala中的方法重写同java一样，也是利用override关键字标识重写父类的算法。

子类重写父类方法的时候，如果父类是抽象类，重写的方法是抽象方法override关键词可以省略，否则必须写override

|  |
| --- |
| **package** top.xiesen.oo.abstracttest  */\*\*  \* Created by Allen on 2017/11/16.  \* DynamicActive用来操作存储系统数据  \*/* **class** DynamicActive {   **def** saveData() = {  *println*(**"保存数据到本地文件系统"**)  }   **def** deleteData() = {  *println*(**"删除本地文件系统数据"**)  }  }  **class** HBaseActive **extends** DynamicActive {  **override def** saveData(): Unit = {  *println*(**"保存数据到Hbase上"**)  } }  **class** MysqlActive **extends** DynamicActive {  **override def** saveData(): Unit = {  *println*(**"保存数据到Mysql上"**)  } }  **object** DynamicActiveFactory {  **def** mkInstance():DynamicActive = { *// new HBaseActive* **new** MysqlActive  } }  **object** DynamicActiveTest {  **def** main(args: Array[String]): Unit = {  **val** active = DynamicActiveFactory.*mkInstance*()  active.saveData()  } } |

### 匿名类

当我们想要实例化一个类型的对象的时候，如果这个类型是一个抽象类，或者是一个接口，而我们又不想重新定义一个类型来继承抽象类或者接口，这时我们使用匿名类

可以使用

new abstractType(){

实现或重写父类的方法

}

|  |
| --- |
| **// 本例和抽象类是一个实例，下面是做修改的部分**  **object** DynamicActiveFactory {  **def** mkInstance():DynamicActive = { *// new HBaseActive // new MysqlActive* **new** DynamicActive(){  **override def** saveData(): Unit = {  *println*(**"保存数据到hive里面"**)  }  }  } } |

## 组合和继承

当我们想定义一个类型，并且希望这个类具有比较强大的功能的时候，我们可以考虑两种方式：

1. 继承：
   1. 优点：
      1. 操作简单
      2. 在功能使用上面，直接调用可访问的属性和方法(不需要实例化父类的对象)
   2. 缺点：
      1. 继承只能单继承
      2. 继承侵入性太强，没办法解耦
2. 聚合
   1. 优点：
      1. 可以多方引入，没有单继承出现的问题
      2. 方便解耦，使用接口来引入想要扩展的类型
   2. 缺点：
      1. 操作复杂，会额外写很多代码，比如说接口的定义
      2. 对功能方法的调用需要通过实例对象来进行

**继承实例代码：**

|  |
| --- |
| **class** DynamicActive {   **def** saveData() = {  *println*(**"保存数据到本地文件系统"**)  }   **def** deleteData() = {  *println*(**"删除本地文件系统数据"**)  } }  **class** HBaseActive **extends** DynamicActive {  **override def** saveData(): Unit = {  *println*(**"保存数据到Hbase上"**)  } }  **class** ExtendedClass **extends** HBaseActive {   *// 通过继承HBaseActive已经具有保存数据的功能* **def** displayData() = {  *println*(**"展示数据到控制台上"**)  } }  **object** ExtendedClassTest {  **def** main(args: Array[String]): Unit = {  **val** ec = **new** ExtendedClass  ec.saveData()  ec.displayData()  } } |

**组合实例代码：**

|  |
| --- |
| **class** DynamicActive {   **def** saveData() = {  *println*(**"保存数据到本地文件系统"**)  }   **def** deleteData() = {  *println*(**"删除本地文件系统数据"**)  } }  **class** HBaseActive **extends** DynamicActive {  **override def** saveData(): Unit = {  *println*(**"保存数据到Hbase上"**)  } }  **class** ExtendedClassCompoise {  *// 通过组合的方式将HBaseActive实例化成一个对象，通过该对象调用HBaseActive上的方法，来保存数据到hbase上* **val** *hBaseActive* = **new** HBaseActive   **def** displayData() = {  *println*(**"展示数据到控制台上"**)  }   **def** saveDataToHbase() = {  *hBaseActive*.saveData()  } }  **object** ExtendedClassTest {  **def** main(args: Array[String]): Unit = {  **val** ecc = **new** ExtendedClassCompoise  ecc.saveDataToHbase()  ecc.displayData()  } } |

## 接口

Scala的总父类类型是Any

AnyVal相当于java的基础数据类型

AnyRef相当于java中的Object

Java的接口:interface

1. 常量
2. 未实现的方法

Scala的接口:traits

1. 常量

2. 未实现的方法

3. 变量 初始化的变量和未初始化的变量

4. 以实现的方法

两种语言在定义接口实现类的时候都必须实现全部接口未实现的方法(scala中包含未初始化的变量)

Traint可以多实现 而抽象类只能继承一个

Traint不能直接进行实例化

如果一个类要实现多个接口，第一个接口使用extends关键词，后面的使用with关键词

如果一个类既要继承另一个类，又要实现接口，那么被继承的类写在extends后面，所有的接口使用with关键词

Class ExtendedAndImpClass extends SuperClass with Traint1 with Traint2...

Traint之间也可以相互继承，接口可以被多继承

|  |
| --- |
| **trait** OperatorPersistentSystem {  **def** saveData()   **def** implementMethod() = {  *println*(**"调用了traint的implementMethod方法"**)  }   **var** unInitAttr: String  **var** *initedAttr* = **"trait已初始化的属性"** }  **trait** DisplayData {  **def** showData() }  *// 实现多个接口，第一个接口使用extends，后面的使用with关键字* **class** OperatorAndDisplayData **extends** OperatorPersistentSystem **with** DisplayData {  **override def** saveData(): Unit = {  *println*(**"保存数据到本地文件系统中"**)  }   **override var** *unInitAttr*: String = **"OperatorAndDisplayData初始化OperatorPersistentSystem的unInitAttr 变量"   override def** showData(): Unit = {  *println*(**"展示数据到控制台"**)  } }  **object** OperatorPersistentSystemTest {  **def** main(args: Array[String]): Unit = {  **val** odd = **new** OperatorAndDisplayData  odd.saveData()  odd.showData()  } } |

## 包和引入

Scala中引入包的基本用法和java一致，除了导入包下所有对象的时候，java使用的是\*，scala使用的是\_

|  |
| --- |
| **import** java.util.Date.\*(java)  **import** java.util.Date.\_(scala) |

Scala中可以在引入同一个包的时候在一行代码中引入多个类型

|  |
| --- |
| **import** java.util.{Date, Random} |

Scala中可以在任何地方引入包

在代码块里面import引入包它的作用域就是代码块内

如果在scala文件的最上方引入，那么它在整个文件内都生效

|  |
| --- |
| **package** top.xiesen.oo.packagetest  **import** java.util **import** java.util.{Date, Random}  **class** PackageTest {  **val** *date* = **new** Date()  **val** *random* = **new** Random()   **def** test() = {  **import** java.util  **val** a = **new** util.ArrayList[Int]()  } } |

**Scala引入类型的时候可以给类型起别名**

|  |
| --- |
| **import** scala.collection.mutable.{Queue=>MutableQueue}  **object** PackageTest {  **def** main(args: Array[String]): Unit = {  *// 使用别名* **val** queue = **new** MutableQueue()  *println*(queue)  } } |

Scala还可以使用import的方式隐藏类型

**import** scala.collection.mutable.{HashMap=>\_}

相当于在当前引入的所有mutable下的类型隐藏了HashMap

|  |
| --- |
| **package** top.xiesen.oo.packagetest  **import** scala.collection.immutable.\_ **import** scala.collection.mutable.{HashMap => \_}  **object** PackageTest {  **def** main(args: Array[String]): Unit = {  **val** map = **new** HashMap[String, String]()  } } |

包的定义基本用法和java一致，同时也有更高级的用法

包的定义可以嵌套，可以和目录不一致

## 包对象

报对象经常用来定义一个包下使用的常量，函数，object等

在使用这些函数和常量时可以无引用的调用，类似println

一个包下面只能有一个包对象

定义方式：package object name...

包对象所定义的包下的类可以无引用的使用包对象中的属性和方法

|  |
| --- |
| **package** top.xiesen.oo  */\*\*  \* 包对象的定义  \*/* **package object** packagetest {  **def** sum(x1: Int, x2: Int) = {  x1 + x2  }   **val** *packageList* = *List*(1, 2, 3, 4) } |

**包对象测试案例**

|  |
| --- |
| **package** top.xiesen.oo.packagetest  */\*\*  \* 包对象测试  \*/* **object** PackageObjectTest {   **def** main(args: Array[String]): Unit = {  *println*(*sum*(1,2))  *println*(*packageList*)  } } |

包对象定义的包外的类也可以无引用的使用包对象中的属性和方法，但是需要使用**import** top.xiesen.oo.packagetest.\_导入

|  |
| --- |
| **package** top.xiesen  */\*\*  \* 外部使用包对象  \*/* **object** ooPackageTest {  **def** main(args: Array[String]): Unit = {  **import** top.xiesen.oo.packagetest.\_  *println*(*sum*(2, 3))  *println*(*packageList*)  } } |

## 权限访问控制和包的关系

Private和protected是scala中仅有的两个权限控制修饰符

Private后面可以通过添加中括号的方式来更灵活的进行自己的权限控制

中括号中可以写：

1. 伴生对象伴生类可以各自访问私有成员变量
2. 内部类

Private[this] ---- 限制伴生对象访问自己的私有变量

Private[类名]

Private[包名] ---- 扩充包下的类，也可以访问自己的私有成员

**Private[this]示例代码：**

|  |
| --- |
| *// 伴生类* **class** ActionTest {  **private var** *pAttr1* = **"private私有成员变量"  private**[**this**] **var** *pAttr2* = **"private[this]私有成员变量"  class** InnerClass {  **def** readActionTestPrivate() = {  *// 可直接访问ActionTest私有成员  println*(*pAttr1*)  *println*(*pAttr2*)}  } } *// 伴生对象* **object** ActionTest {  *// 伴生对象访问私有成员变量* **def** readActionTestPrivate() = {  **val** obj = **new** ActionTest  *println*(obj.*pAttr1*)  *// 伴生对象不能访问private[this]修饰的私有成员变量  // println(obj.pAttr2)* } } |

**Private[包名]示例代码：**

|  |
| --- |
| **package** top {  **package** xiesen {   **import** top.xiesen.actiontest.ActionTest   **object** ActionXiesenTestMain {  **def** readActionTestPrivate() = {  **val** actionTest = **new** ActionTest  *println*(actionTest.*pAttr4*)  }  }  **package** actiontest {   *// 伴生类* **class** ActionTest {  **private var** *pAttr1* = **"private私有成员变量"  private**[**this**] **var** *pAttr2* = **"private[this]私有成员变量"** *// private[InnerClass] var pAttr3 = "private[InnerClass]私有成员变量"* **private**[xiesen] **var** *pAttr4* = **"private[xiesen]私有成员变量"   class** InnerClass {  **def** readActionTestPrivate() = {  *// 可直接访问ActionTest私有成员  println*(*pAttr1*)  *println*(*pAttr2*)  *// println(pAttr3)* }  }   }   *// 伴生对象* **object** ActionTest {  *// 伴生对象访问私有成员变量* **def** readActionTestPrivate() = {  **val** obj = **new** ActionTest  *println*(obj.*pAttr1*)  *// 伴生对象不能访问private[this]修饰的私有成员变量  // println(obj.pAttr2)   // println(obj.pAttr3)* }   }   **object** ActionTestMain {  **def** main(args: Array[String]): Unit = {  *println*(**"伴生对象的访问"**)  ActionTest.*readActionTestPrivate*()  *println*(**"内部类的访问"**)  **val** actionTest = **new** ActionTest  **val** innerClass = **new** actionTest.InnerClass  innerClass.readActionTestPrivate()  *// val innerClass1 = new actionTest.InnerClass   println*(**"包下访问"**)  ActionXiesenTestMain.*readActionTestPrivate*()  }  }   }   } } |

## 创建maven的scala项目

1. 找模板(百度 scala maven archetype关键词 <http://docs.scala-lang.org/tutorials/scala-with-maven.html>)

G: net.alchim31.maven

A: scala-archetype-simple

V: 1.5

1. intellj中new--》project--》maven

勾选create from archetype

点击 add archetype按钮

模板下载完之后选择我们添加的模板：net.alchim31.maven:scala-archetype-simple

next

3.填写自己项目的GAV

4.next---next---finish

5.import changens

6.pom中修改

|  |
| --- |
| <scala.tools.version>2.11</scala.tools.version>  <scala.version>2.11.8</scala.version> |

改成自己的版本

删除：

|  |
| --- |
| <dependency>  <groupId>org.specs2</groupId>  <artifactId>specs2\_${scala.tools.version}</artifactId>  <version>1.13</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.scalatest</groupId>  <artifactId>scalatest\_${scala.tools.version}</artifactId>  <version>2.0.M6-SNAP8</version>  <scope>test</scope>  </dependency> |

删除：

|  |
| --- |
| <arg>-make:transitive</arg> |