Day28_Scala面向对象

大数据-张军锋 Day28 Scala 面向对象

- Day28_Scala面向对象
  - 类的定义
    - Java类 & Scala类
      - Java类
      - scala类
    - Scala类和Java类的相同点
    - Scala类和Java类的不同点
  - 构造方法
    - java构造方法
    - Scala构造方法
  - 单例对象Object
  - 相爱相惜—伴生
  - apply方法
  - 抽象类
    - java抽象类
    - scala抽象类
  - 继承
  - 匿名类
  - 组合和继承
    - 继承
    - 组合
    - 总结
  - Scala类层次结构
    - AnyVal
    - AnyRef
  - 接口
    - java的接口
    - scala的接口
    - 总结
  - 包和引入
    - 引入方式
    - 位置和作用域
    - 其他操作
  - 包对象
  - 权限访问控制和包的关系
  - scala创建maven项目

类的定义

scala的类定义也是使用class关键词后面跟类的名称然后大括号：`class Person {}`

Java类 & Scala类

Java类

- **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

- **属性**：非静态属性
- **方法**：非静态方法
- **非静态代码块** ：每一次被实例化的时候都会被执行一次
  （ class里面的过程代码（代码块），其实就是类的构造方法 ）
- 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要通过实例化的对象才能调用**
- **在Object里面定义的属性和方法，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**，不写默认public

Scala类和Java类的相同点

- **没有（public）**：所有其他代码都可以访问
- **private**：只有自己可以访问
- **protected**：只有子类和同包下面的可以访问

Scala类和Java类的不同点

- **private**
  Java：只有自己可以访问
  Scala：除了自己可以访问之外可以额外开放访问权限
- **protected**
  Java：只有子类和同包下面的可以访问
  Scala：除了子类和同包外也可以额外开放访问权限

构造方法

java构造方法

- **有默认无参构造方法**
- **自定义构造方法时默认构造方法消失**
- **构造方法可以重载** `[public 类名 （参数列表）{ }]`
- **不同的构造方法之间地位是平等的**

Scala构造方法

- **有无参的默认构造方法**
- **构造方法也可以重载**
- **不同的构造方法之间地位不平等，每个scala类都有唯一的主构造方法**
- 除了主构造方法之外，**所有副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创建**
- **主构造方法的声明是类声明后面来写**
- **副构造方法是写在类体内的，它的名字统一都叫this**
- 所有的副构造方法在方法体内必须先直接或间接的调用主构造方法后才能写自己的构造逻辑代码，**构造方法不需要返回值，它的返回值是Unit**
- **构造方法可以使用默认值参数**，这样能大大的提高构造方法调用的灵活性

```
class ConstructorWithDefault(var attr1:String,var attr2:String = "d
efaultATTR2",var attr3:Int = 3)
```

- **可以通过在主构造方法参数前面加private的方式来将主构造方法私有化**

```
class ConstructorMainPrivate private(var attr1:String,var attr2:Str
ing)
```

```scala
package com.bd14.zjf.oo

class ConstructorTest(pattr1: String, pattr2: Int) {

  //属性1
  var attr1 = pattr1
  //属性2
  var attr2 = pattr2

  //副构造方法
  def this() = {
    this("", 0) //直接调用主构造方法
    println("执行了副构造方法")
    this.attr1 = "副构造方法内赋值"
  }

  def this(pattr1: String) = {
    this()
    println("执行了副构造方法2")
    this.attr1 = pattr1
  }

}

object ConstructorTestOb {
  def main(args: Array[String]): Unit = {
    val c1 = new ConstructorTest("zhangsan", 16)
    println(s"姓名:${c1.attr1},年龄:${c1.attr2}")
    var c2 = new ConstructorTest()
    println(s"姓名:${c2.attr1},年龄:${c2.attr2}")
    var c3 = new ConstructorTest("qqqq")
    println(s"姓名:${c3.attr1},年龄:${c3.attr2}")
  }
}
```

单例对象Object

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

相爱相惜—伴生

> 如果定义**一个object和一个class**，他们的**名称一样**，那么在编译成class文件的时候，他们会**共用一个.class文件**，这样一个object和class他们**互为伴生**

- 在伴生类中可以通过类名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
- 伴生类和伴生对象之间可以互相访问private的成员
- 但是如果private添加泛型限制则会有额外的限制，如private[this]

```
package com.bd14.zjf.oo

//Student的伴生类
class Student(var studentNo: String, var studentName: String, var s
tudentClass: String, var age: Int) {
  private def classPrivateMethod() = {
    println("伴生类的私有方法classPrivateMethod")
  }

  def printlnStudent() = {
    println(s"studentNo:$studentNo,studentName:$studentName,student
Class:$studentClass,age:$age")
  }

  private[this] def classPrivateThisMethod() = {
    println("class的private[this]方法")
  }

  //对其开放访问权限
  //这里要求中括号内的类型需要使用对象封装,即StudentTest和Student要封装在一
个包内
  //private[StudentTest] def classPrivateStudentTestMethod() = {
  //  println("Student的private[StudentTest]方法")
  //}
}

//student的伴生对象
object Student {
  private def ObjectPrivateMethod() = {
    println("伴生对象的私有方法ObjectPrivateMethod")
  }

  var schoolName: String = ""

  def gotoSchool() = {
    println("伴生对象的上学方法")
  }

  def callStudentClassPrivateMethod() = {
    val s = new Student("002", "私有测试", "二年级", 19)
    s.classPrivateMethod()
    //s.classPrivateThisMethod()
  }
}

object StudentTest {
  def main(args: Array[String]): Unit = {
    val student = new Student("001", "小王", "一年级", 18)
```

```
    student.printlnStudent()
    Student.gotoSchool()
    //不能访问私有成员
    //student.classPrivateMethod()
    //Student.ObjectPrivateMethod()
    //student.classPrivateStudentTestMethod()
  }
}
```

apply方法

> apply方法在scala中是有特殊作用的方法，它可以直接通过object名称后面加小括号的形式来调用

在object Student中定义一个apply方法

```
def apply() = {
  println("调用了apply方法")
}
```

apply方法的调用

```
Student.apply() ====(等用于) Student()

//调用apply方法,可以不用new对象调用方法
val student1 = Student("003", "小李", "二年级", 19)
```

apply方法和普通方法一样可以被重载

```
def apply() = {
  println("调用了apply方法")
}
def apply(studentNo:String,studentName:String,studentClass:String,a
ge:Int) = new Student(studentNo,studentName,studentClass,age)
```

抽象类

java抽象类

- **不能够被实例化**
- **可以定义属性，可以定义已实现的方法，也可以定义抽象方法**
- **子类必须实现抽象类中所有的抽象方法**

scala抽象类

- **不能够被实例化**
- **可以定义属性，可以定义已实现的方法**
- **可以定义未被初始化的属性，和未被实现的方法**
- **子类必须初始化所有抽象类中未初始化的属性**
- **必须实现抽象类中未被实现的方法**
- **在定义未实现的方法上必须指定返回值类型**
- **抽象类可以定义构造方法**，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有属性定义，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会先调用父类的构造方法再调用子类的构造方法**

```
package com.bd14.zjf.oo.abstracttest

abstract class Person(var name: String) {

  println("Person的构造方法")
  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")

  override var ptype: String = "学生类型"

  override def work(): Unit = {
    println("学生工作是上课")
  }

}

class Teacher(name: String) extends Person(name) {

  println("Teacher的构造方法")

  override var ptype: String = "老师类型"

  override def work(): Unit = {
    println("学老师工作是讲课")
  }

}

object PersonTest {

  def main(args: Array[String]): Unit = {

    val student = new Student("塔利亚")
    val teacher = new Teacher("亚索")
    student.printlnMethod()
    teacher.printlnMethod()
    student.work()
```

```
    teacher.work()

  }

}
```

继承

- scala中的继承和java一样使用extends修饰符来定义父类和子类之间的关系
- 子类会从父类中继承访问控制权限内的属性和方法
- 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

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实现接口
- 这时候我们可以使用匿名类

```
def mkInstance():DynamicAction = {
  new DynamicAction(){
    override def saveData(): Unit = {
      //实现或重写父类的方法
      println("这里是重写后的方法")
    }
  }
}
```

组合和继承

> 当我们想定义一个类型，并且**希望这个类型具有比较强大的功能的时候**
> 我们可以考虑两种方式：继承和聚合

继承

- **操作简单**：继承的优点是子类可以重写父类的方法来方便地实现对父类的扩展。
- 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）
- 继承**只能单继承**
- **父类的内部细节对子类是可见的**
- **不灵活**：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为
- **侵入性太强，没办法解耦**：如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改。所以说子类与父类是一种高耦合，违背了面向对象思想。

组合

组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量

- **操作复杂**，会额外写很多代码，比方说接口的定义
- **容易产生过多的对象**：对功能方法的调用需要通过实例对象来进行
- **安全性**：当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的内部细节对当前对象时不可见的。
- **可以多方引入**，没有单继承出现的问题
- **方便解耦**，当前对象与包含的对象是一个低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码
- 当前对象可以在运行时动态的绑定所包含的对象。可以通过set方法给所包含对象赋值。

总结

组合比继承更具灵活性和稳定性，所以在设计的时候优先使用组合。只有当下列条件满足时才考虑使用继承：

1. 子类是一种特殊的类型，而不只是父类的一个角色
2. 子类的实例不需要变成另一个类的对象
3. 子类扩展，而不是覆盖或者使父类的功能失效

```scala
package com.bd14.zjf.oo.abstracttest

class ExtendedClass extends HBaseAction {

  //通过继承ExtendedClass已经具有保存数据到hbase的功能
  def displayData() = {
    println("展示数据Dashboard")
  }

}

trait OperPersistent {
  def saveData()
}

class ExtendedClassCompoise(val hBaseAction: OperPersistent) {
  //通过组合的方式把想要具有功能的类型实例化一个对象
  //通过该对象的调用具备保存数据到hbase的功能
  //val hBaseAction = new HBaseAction

  def displayData() = {
    println("展示数据到Dashboard")
  }

  def saveDataToHBase() = {
    hBaseAction.saveData()
  }
}

object ExtendedClassTest {
  def main(args: Array[String]): Unit = {
    val ec = new ExtendedClass()
    ec.saveData()
    ec.displayData()
    //val ecc = new ExtendedClassCompoise()
    //ecc.saveDataToHBase()
    //ecc.displayData()
    val ecc = new ExtendedClassCompoise(new HBaseAction())
    ecc.saveDataToHBase()
    ecc.displayData()
  }
}
```

Scala类层次结构

scala的总父类型是Any
AnyVal相当于java的基础数据类型
AnyRef相当于java中的Object

Scala中的类层次结构图如下：

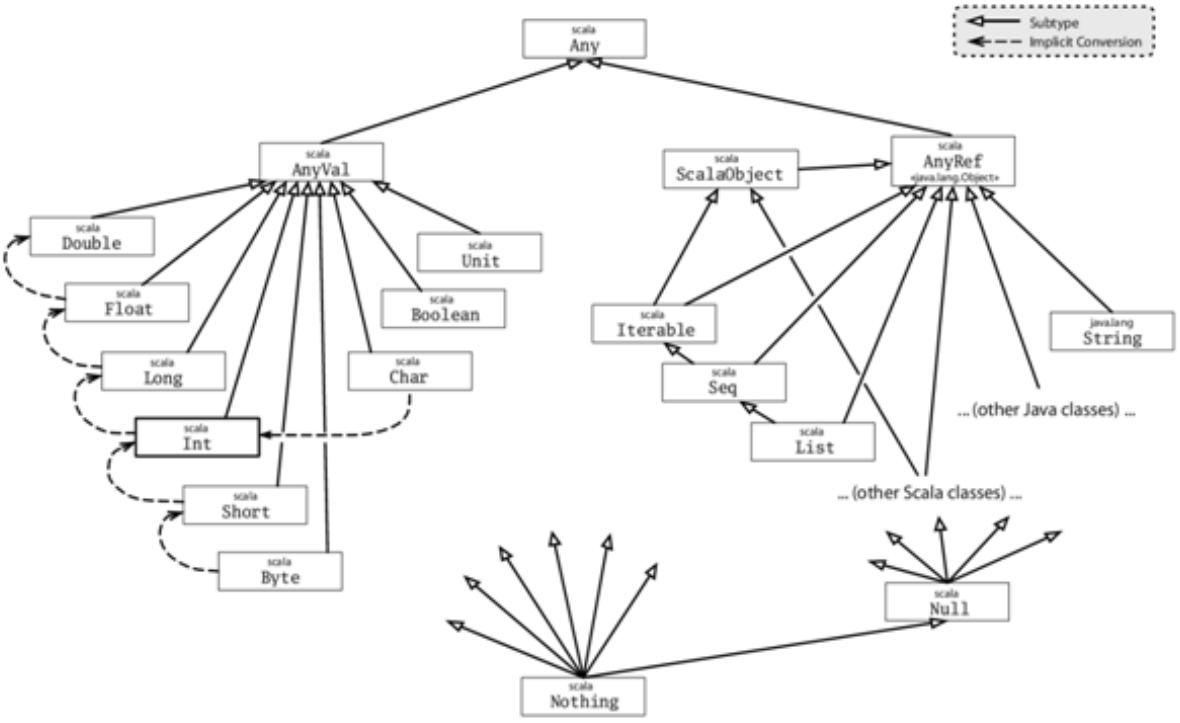

Figure 11.1 · Class hierarchy of Scala.

AnyVal

> AnyVal是所有scala内置的值类型（ Byte, Short, Char, Int, Long, Float, Double, Boolean, Unit.）的父类，其中 Byte, Short, Char, Int, Long, Float, Double, Boolean与java中的byte,short,char,int,long,float,double,boolean原生类型对应，而Unit对应java中的void类型，由于（ Byte, Short, Char, Int, Long, Float, Double, Boolean, Unit）继承AnyVal，而AnyVal又继承Any，因此它们也可以调用toString等方法。

值得一提的是，()可以作为Unit类型的实例，它同样可以调用toString等方法

AnyRef

> AnyRef是Any的另外一个子类，它是scala中所有非值类型的父类，对应Java.lang.Object类（可以看作是java.lang.Object类的别名），也即它是所有引用类型的父类（除值类型外）。那为什么不直接Java.lang.Object作为scala非值引用类型的父类呢？这是因为Scala还可以运行在其它平台上如.Net，所以它使用了AnyRef这个类，在JVM上它对应的是java.lang.Object，而对于其它平台有不同的实现。

接口

> scala和java语言一样，采用了很强的限制策略，避免了多种继承的问题。在java语言中，只允许继承一个超类，该类可以实现多个接口，但java接口有其自身的局限性：接口中只能包括抽象方法，不能包含字段、具体方法。Scala语言利用Trait解决了该问题，在scala的trait中，它不但可以包括抽象方法还可以包含字段和具体方法。

java的接口

- interface
- 常量
- 未实现的方法

scala的接口

- traits
- 常量
- 未实现的方法
- 变量 初始化的变量和未初始化的变量
- 已实现的方法

总结

- 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）
- traits可以多实现 而抽象类只能继承一个
- trait不可以被实例化
- 如果一个类要实现多个接口的话，第一个接口前用extends，后面每一个接口前都用with

```
class ImpClass extends Trait1 with Trait2 with Trait3...
```

- 如果一个类既要继承另一个类又要实现多个接口，那么被继承的类写在extends后面，所有的接口前面加with

```
class ExtendedAndImpClass extends SuperClass with Trait1 with Trait
2 ...
```

- trait之间也可以继承，可以被多继承

```
package com.bd14.zjf.oo.abstracttest

trait OperPersistentSystem {
  def saveData()

  def implementsMethod() = {
    println("调用了trait实现的implementsMethod方法")
  }

  var unInitAttr:String
  var initedAttr = "trait的已初始化的属性"
}

trait DisplayData{
  def showData()
}

class OperPersistentSystemImp extends OperPersistentSystem{
  override def saveData(): Unit = {
    println("保存数据到本地文件系统中")
  }
  override var unInitAttr: String = "在OperPersistentSystemImp中初始
化unInitAttr属性"
}

//实现多个接口，第一个接口用extends 后面的全用with
class OperAndDisplayData extends OperPersistentSystem with DisplayD
ata{
  override def saveData(): Unit = {
    println("保存数据到本地文件系统")
  }
  override var unInitAttr: String = "OperAndDisplayData初始化OperPer
sistentSystem的变量"

  override def showData(): Unit = {
    println("展示数据到DashBoard")
  }
}

object OperPersistentSystemTest{
  def main(args: Array[String]): Unit = {
    val operPersistentSystem = new OperPersistentSystemImp()
    operPersistentSystem.saveData()
    println(operPersistentSystem.initedAttr)
    println(operPersistentSystem.unInitAttr)
    operPersistentSystem.implementsMethod()
    println("------------------------------")
    val operAndDisplayData = new OperAndDisplayData()
```

```
    operAndDisplayData.saveData()
    operAndDisplayData.showData()
  }
}
```

包和引入

引入方式

- scala中引入包（import）的基本用法和java一致，除了导入包下所有类型的时候Java用的是*，scala中用的是_
- scala中可以在引入统一个包的时候用一行代码引入多个类型：`import java.util.{Date,Random}`

位置和作用域

- scala中可以在任意的地方引入包
- 在代码块里用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

其他操作

scala中引入的类型的时候可以给类型起别名

```
import scala.collection.mutable.{Queue=>MutableQueue}
```

scala还可以使用import的方式类隐藏类型

```
import scala.collection.immutable.{HashMap=>_,_}
```

相当于在当前引入的所有immutable下的类型隐藏HashMap
在使用HashMap的时候就不会发生类名冲突的问题

> 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

- 包对象上经常用来定义一个包下可以使用的常量，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

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来导入

```
import com.zhiyou.bd17.oo.packagetest._
```

权限访问控制和包的关系

private或protected是scala中仅有的两个权限控制修饰

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**
2. **内部类**
   private[this] – 限制上述伴生对象访问自己的私有成员
   private[类名] –
   private[包名] – 扩充包下的类也可以访问自己的私有成员

scala创建maven项目

1. 找模板（scala maven archetype） http://docs.scala-lang.org/tutorials/scala-with-maven.html
   G：net.alchim31.maven
   A：scala-archetype-simple
   V：1.5
2. **intellj中new–>project–>maven**
   勾选create from archetype
   点击 add archetype按钮
   G：net.alchim31.maven
   A：scala-archetype-simple
   V：1.5

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
   改成自己的版本

```
 <scala.tools.version>2.11</scala.tools.version>
 <scala.version>2.11.11</scala.version>
```

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>
```

删除 test.scala.samples下的所有代码

7. **打开App程序 右键运行**
   打印结果
   scala maven就创建成功

在pom中导入mysql驱动
用scala写一个mysql的增删改查