Scala核心编程

-面向对象编程（高级特性）

尚硅谷研究院

- 静态属性和静态方法

静态属性-提出问题

提出问题的主要目的就是让大家思考解决之道，从而引出我要讲的知识点.

说：有一群小孩在玩堆雪人,不时有新的小孩加入,请问如何知道现在共有多少人在玩?**请使用面向对象的思想**，编写程序解决。

- 静态属性和静态方法

基本介绍

- 回顾下Java的静态概念

  public static 返回值类型 方法名(参数列表) {方法体}
  静态属性...

说明: Java中静态方法并不是通过对象调用的，而是通过类对象调用的，所以静态操作并不是面向对象的。

scala伴生类和伴生对

- Scala中静态的概念-伴生对象
  Scala语言是完全面向对象(**万物皆对象**)的语言，所以并没有静态的操作(即在Scala中没有静态的概念)。但是为了能够和Java语言交互(因为Java中有静态概念)，就产生了一种特殊的对象来**模拟类对象**，我们称之为类的**伴生对象**。这个类的所有**静态内容**都可以**放置在它的伴生对象**中声明和调用

- 静态属性和静态方法

伴生对象的快速入门

```
class ScalaPerson {
  var name : String = _
}
```

说明：class ScalaPerson 是伴生类

```
object ScalaPerson {
  var sex : Boolean = true
}
```

说明：object ScalaPerson 是伴生对象

```
println(ScalaPerson.sex)
```

案例演示+反编译(图)+小结

- 静态属性和静态方法

伴生对象的小结

1) Scala中伴生对象采用object关键字声明，伴生对象中声明的全是 **"静态"** 内容，可以通过**伴生对象名称**直接调用。
2) 伴生对象对应的类称之为伴生类，伴生对象的名称应该和伴生类名一致。

3) 伴生对象中的属性和方法都可以通过**伴生对象名(类名)直接调用**访问
4) 从语法角度来讲，所谓的伴生对象其实就是类的静态方法和成员的集合
5) 从技术角度来讲，**scala还是没有生成静态的内容**，只不过是将伴生对象生成了一个新的类，**实现属性和方法的调用**。[反编译看源码]
6) 从底层原理看，伴生对象实现静态特性是依赖于 public static final ***MODULE$*** 实现的。

- **静态属性和静态方法**

伴生对象的小结

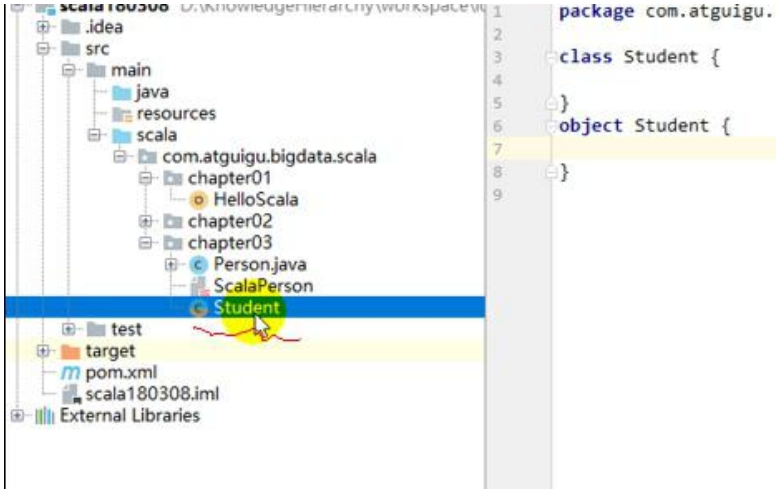
7) 伴生对象的声明应该和伴生类的声明在**同一个源码文件**中(如果不在同一个文件中会**运行错误**!)，但是如果没有伴生类，也就没有所谓的伴生对象了，所以放在哪里就无所谓了。

8) 如果 class A 独立存在，那么A就是一个类， 如果 object A 独立存在，那么A就是一个"静态"性质的对象[即**类对象**], 在 object A中声明的属性和方法可以通过 A.属性 和 A.方法 来实现调用

- 静态属性和静态方法

伴生对象的小结

9) 当一个文件中，存在伴生类和伴生对象时，文件的图标会发生变化



- 静态属性和静态方法

伴生对象解决小孩游戏问题

如果,设计一个var total Int表示总人数,我们在创建一个小孩时，就把total加1,并且 total是所有对象共享的就ok了!，我们使用伴生对象来解决

画一个小图给大家理解。

- 静态属性和静态方法

伴生对象-apply方法

在伴生对象中定义apply方法，可以实现： 类名(参数) 方式来创建对象实例.

```
//介绍下通过apply方法，创建一个对象
class Cat(cName: String) {
  var name:String = cName

}
//提供一个伴生对象
object Cat {
  //apply方法是当我们创建对象时，可以通过Cat()调用
  def apply(): Cat = {
    println("apply被调用")
    new Cat("xx")
  }
  def apply(name: String): Cat = {
    println("apply被调用")
    new Cat(name)
  }
}
```

图片2.z

静态属性和静态方法

课堂练习

➢ 题1(学员思考)

下面的题，是一道java题，请使用Scala 完成该题的要求.

1) 在Frock类中声明私有的静态属性currentNum，初始值为100000，作为衣服出厂的序列号起始值。
2) 声明公有的静态方法getNextNum，作为生成上衣唯一序列号的方法。每调用一次，将currentNum增加100，并作为返回值。
3) 在TestFrock类的main方法中，分两次调用getNextNum方法，获取序列号并打印输出。
4) 在Frock类中声明serialNumber(序列号)属性，并提供对应的get方法；
5) 在Frock类的构造器中，通过调用getNextNum方法为Frock对象获取唯一序列号，赋给serialNumber属性。
6) 在TestFrock类的main方法中，分别创建三个Frock 对象，并打印三个对象的序列号，验证是否为按100递增。

- 单例对象

这个部分我们放在scala设计模式专题进行讲解

图片1.z

- 接口

回顾Java接口

➢ 声明接口

interface 接口名

➢ 实现接口

class 类名 implements 接口名1，接口2

1) 在Java中, 一个类可以实现多个接口。
2) 在Java中，接口之间支持多继承
3) 接口中属性都是常量
4) 接口中的方法都是抽象的

- 接口

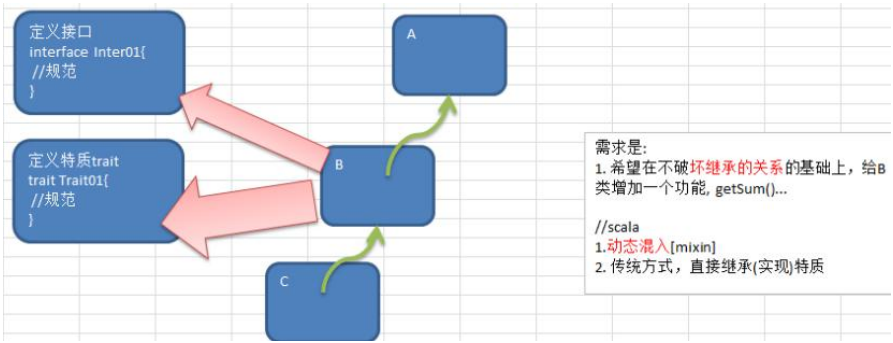
Scala接口的介绍

从面向对象来看，接口并不属于面向对象的范畴，Scala是**纯面向对象**的语言，在Scala中，没有接口。

Scala语言中，**采用特质trait（特征）来代替接口的概念**，也就是说，多个类具有相同的特征（特征）时，就可以将这个特质（特征）独立出来，采用关键字**trait**声明。 理解trait 等价于(interface + abstract class)

如何理解特质?图

trait原理示意



- 特质(trait)

trait 的声明

```
trait 特质名 {
        trait体
}
```

trait 命名 一般首字母大写.

Cloneable , Serializable

```
object T1 extends Serializable {

}
```

Serializable： 就是scala的一个特质。
在scala中， java中的接口可以当做特质使用

- 特质(trait)

Scala中trait 的使用

一个类具有某种特质（特征），就意味着这个类满足了这个特质（特征）的所有要素，所以在使用时，也采用了**extends**关键字，如果有多个特质或存在父类，那么需要采用**with**关键字连接

- 没有父类
  class 类名 extends 特质1 with 特质2 with 特质3 ..

- 有父类
  class 类名 extends 父类 with 特质1 with 特质2 with 特质3

- 特质(trait)

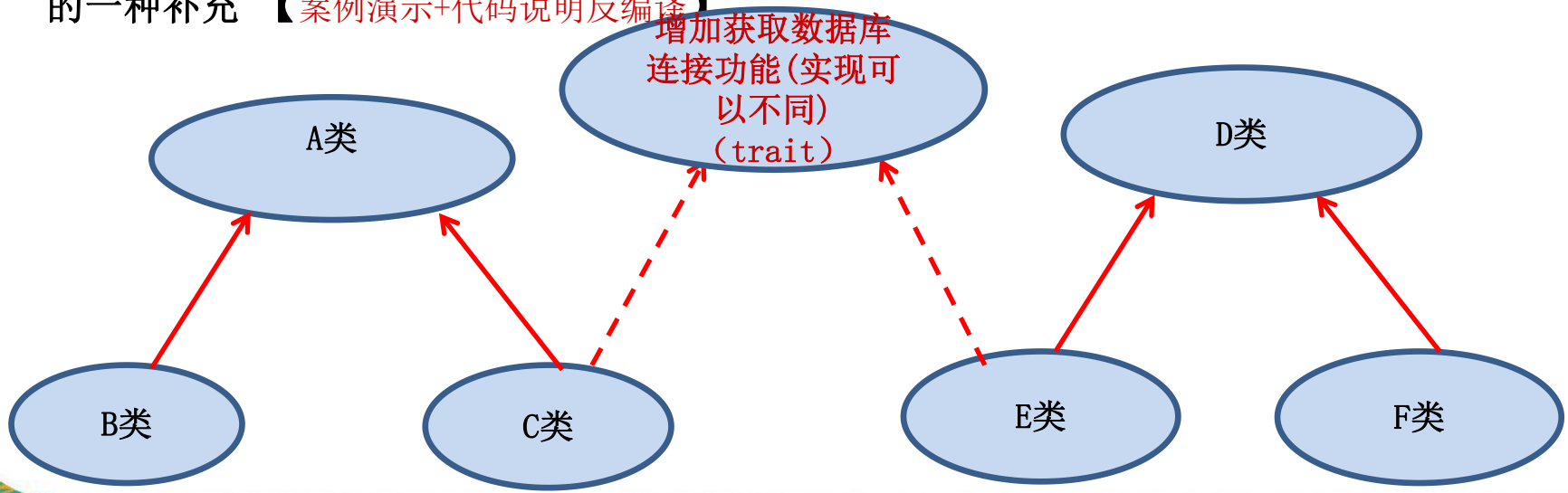
特质的快速入门案例

➢ 可以把特质可以看作是对继承的一种补充

Scala的继承是单继承,也就是一个类最多只能有一个父类,这种
的纯洁性,比c++中的多继承机制简洁。但对子类功能的扩展有一定影响.所以
我们认为: Scala引入trait特征 第一可以替代Java的接口, 第二个也是对单继承机制
的一种补充 【案例演示+代码说明反编译】

```
trait trait1 {
  //声明方法，抽象的.
  def getConnect(user: String, pwd: String): Unit
  //def test(n1:Int)
}
class A {}
class B extends A {}
class C extends A with trait1 {
  override def getConnect(user: String, pwd: String): Unit = {
    println("c连接mysql")
  }}

class D {}
class E extends D with trait1 {
  def getConnect(user: String, pwd: String): Unit = {
    println("e连接oracle")
  }}
class F extends D {}
```



- 特质trait

特质trait 的再说明

1) Scala提供了特质（trait），特质可以同时拥有抽象方法和具体方法，一个类可以实现/继承多个特质。【案例演示+反编译】


图片3.z

```
object TraitDemo02 {
  def main(args: Array[String]): Unit = {
    val a = new AA
    a.sayOk()
    a.sayHello()
    val trait02Obj = new Trait02 { //匿名子类
      override def sayOk(): Unit = {
        println("trait02Obj sayOk")
      }
    }
    trait02Obj.sayHello()
    trait02Obj.sayOk()
  }
}
trait Trait02 {
  def sayOk()
  //实现 [默认实现]
  def sayHello(): Unit = {
    println("Trait02 sayHello...")
  }
}
class AA extends Trait02{
  override def sayOk(): Unit = {
    println("AA sayOk")
```

Trait02.class [interface]
{
//将所有的方法都声明[抽象]
}

Trait02$class.class [抽象类]
{
//将特质中实现了方法放到这里
}

AA implements Trait02
{
//实现所有的方法。 如果有一个方法已经实现，则直接使用Trait02$class 方法。
}

- 特质trait

特质trait 的再说明

2) 特质中没有实现的方法就是抽象方法。类通过extends继承特质，通过with可以继承多个特质
3) 所有的java接口都可以当做Scala特质使用 【案例演示+小结】

```
trait Logger {
  def log(msg: String)
}
```

```
class Console extends Logger with Cloneable with Serializable {
  def log(msg: String) {
    println(msg)
  }
}
```

- 特质trait

带有具体实现的特质

说明：和Java中的接口不太一样的是特质中的方法并不一定是抽象的，也可以有非抽象方法(即：实现了的方法)。[案例演示+反编译]

```
trait Operate {
    def insert( id : Int ): Unit = {
        println("保存数据="+id)
    }
}
```

```
trait DB extends Operate {
    override  def insert( id : Int ): Unit = {
        print("向数据库中")
        super.insert(id)
    }
}
```

```
class MySQL extends DB {
}
```

- 特质trait

带有特质的对象，动态混入

1) 除了可以在类声明时继承特质以外，还可以在构建对象时**混入**特质，扩展目标类的功能【反编译看动态混入本质】
2) 此种方式也可以应用于对抽象类功能进行扩展
3) 动态混入**是Scala特有的方式（java没有动态混入）**，可在不修改类声明/定义的情况下，扩展类的功能，非常的灵活，**耦合性低**。
4) 动态混入可以在不影响原有的继承关系的基础上，给指定的类扩展功能。[如何理解]
5) 案例演示

```
trait Operate3 {
  def insert(id: Int): Unit = {
    println("插入数据 = " + id)
  }
}
```

```
class OracleDB {
}
abstract class MySQL3 {
}
```

```
var oracle = new OracleDB with  Operate3
oracle.insert(999)
val mysql = new MySQL3 with Operate3
mysql.insert(4)
```

6) 思考：如果抽象类中有 抽象的方法，如何动态混入特质？

- 特质trait

带有特质的对象，动态混入

- **课堂练习：**在Scala中创建对象共有几种方式？

1. **new** 对象
2. **apply** 创建
3. 匿名子类方式
4. 动态混入

- 特质trait

叠加特质

➢ 基本介绍

构建对象的同时如果混入多个特质，称之为叠加特质，
那么特质声明顺序从左到右，方法执行顺序从右到左。

- 特质trait

叠加特质

➢ 叠加特质应用案例

目的：分析叠加特质时，对象的构建顺序，和执行方法的顺序

```
trait Operate4 {
  println("Operate4...")
  def insert(id : Int)
}
```

```
trait Data4 extends Operate4 {
  println("Data4")
  override  def insert(id : Int): Unit = {
    println("插入数据 = " + id)
  }
}
```

```
trait DB4 extends Data4 {
  println("DB4")
  override def insert(id : Int): Unit = {
    print("向数据库")
    super.insert(id)
  }
}
```

```
trait File4 extends  Data4 {
  println("File4")
  override def insert(id : Int): Unit = {
    print("向文件")
    super.insert(id)
  }}
class MySQL4 {}
```

```
// 1.Scala在叠加特质的时候，会首先从后面的特质开始执行
// 2.Scala中特质中如果调用super，并不是表示调用父特质的方法，而是向前面（左边）继续查找特质，如果找不到，才会去父特质查找
val mysql = new MySQL4 with DB4 with File4
//val mysql = new MySQL4 with File4 with DB4
mysql.insert(888)
```

- 特质trait

叠加特质

➢ 叠加特质注意事项和细节

1) 特质声明顺序从左到右。
2) Scala在执行叠加对象的方法时，会首先从后面的特质(从右向左)开始执行
3) Scala中特质中如果调用super，并不是表示调用父特质的方法，而是**向前面（左边）继续查找特质**，如果找不到，才会去父特质查找
4) 如果想要调用具体特质的方法，可以指定：super[特质].xxx(…).其中的泛型必须是该特质的直接超类类型

```
trait File4 extends  Data4 {
  println("File4")
  override def insert(id : Int): Unit = {
    print("向文件")
    super[Data4].insert(id)
  }}
```

➢ 叠加特质的课堂练习

要求：修改一下构建对象的混入多个特质的顺序，请学员说出输出结果.

- 特质trait

在特质中重写抽象方法特例

➢ 提出问题，看段代码

```
trait Operate5 {
  def insert(id : Int)
}
trait File5 extends Operate5 {
  def insert( id : Int ): Unit = {
    println("将数据保存到文件中..")
    super.insert(id)
  }
}
```

运行代码，并小结问题 （错误，原因就是没有完全的实现insert，同时你还没有声明 abstract overrid）

- 特质trait

在特质中重写抽象方法特例

➢ 解决问题

方式1 : 去掉 super()...

方式2: 调用父特质的抽象方法，那么在实际使用时，没有方法的具体实现，无法编译通过，为了避免这种情况的发生。可重写抽象方法，这样在使用时，就必须考虑动态混入的顺序问题。

```
trait Operate5 {
  def insert(id : Int)
}
```

```
trait File5 extends Operate5 {
  abstract override  def insert( id : Int ): Unit = {
    println("将数据保存到文件中..")
    super.insert(id)
  }
}
```

```
trait DB5 extends  Operate5 {
  def insert( id : Int ): Unit = {
    println("将数据保存到数据库中..")
  }
}
class MySQL5 {}
val mysql5 = new MySQL5 with DB5  with File5
```

- **特质trait**

在特质中重写抽象方法

- 理解 abstract override 的小技巧分享：
  可以这里理解，当我们给某个方法增加了**abstract override** 后，就是明确的告诉编译器，该方法确实是重写了父特质的抽象方法，但是重写后，该方法仍然是一个抽象方法（因为没有完全的实现，**需要其它特质继续实现[通过混入顺序]**）

- 特质trait

在特质中重写抽象方法

- 重写抽象方法时需要考虑混入特质的顺序问题和完整性问题
  看4个案例，并判断结果。

```
var mysql2 = new MySQL5  with DB5 //ok
mysql2.insert(100)
var mysql3 = new MySQL5  with File5 //error
mysql2.insert(100)

var mysql4 = new MySQL5 with File5 with DB5// error
mysql4.insert(100)
var mysql4 = new MySQL5 with DB5 with File5// ok
mysql4.insert(100)
```

- 特质trait

当作富接口使用的特质

富接口：即该特质中既有抽象方法，又有非抽象方法

```
trait Operate {
    def insert( id : Int ) //抽象
    def pageQuery(pageno:Int, pagesize:Int): Unit = { //实现
        println("分页查询")
    }
}
```

- 特质trait

特质中的具体字段

特质中可以定义具体字段，如果初始化了就是具体字段，如果不初始化就是抽象字段。**混入该特质的类就具有了该字段**，字段不是继承，而是直接加入类，成为自己的字段。

【案例演示+反编译】

```
trait Operate6 {
  var opertype : String
  def insert()
}
```

```
trait DB6 extends  Operate6 {
  var opertype : String = "insert"
  def insert(): Unit = {
  }
}
class MySQL6 {}
```

```
var mysql = new MySQL6 with DB6
//通过反编译，可以看到 opertype
println(mysql.opertype)
```

- 特质trait

特质中的抽象字段

特质中未被初始化的字段在具体的子类中必须被重写。

- **特质trait**

特质构造顺序

➢ 介绍

特质也是有构造器的，构造器中的内容由“字段的初始化”和一些其他语句构成。具体实现请参考“特质叠加”

➢ 第一种特质构造顺序(**声明类的同时混入特质**)

1) 调用当前类的超类构造器
2) 第一个特质的父特质构造器
3) 第一个特质构造器
4) 第二个特质构造器的父特质构造器, 如果已经执行过，就不再执行
5) 第二个特质构造器
6) .....重复4,5的步骤(如果有第3个，第4个特质)
7) 当前类构造器 [案例演示]

```
trait AA {
  println("A...")
}
trait BB extends  AA {
  println("B....")
}
trait CC extends  BB {
  println("C....")
}
trait DD extends  BB {
  println("D....")
}
class EE {
  println("E...")
}
class FF extends EE with CC with DD {
  println("F....")
}
class KK extends EE {
  println("K....")
}
```

```
val ff1 = new FF()
println(ff1)
val ff2 = new KK()
println(ff2)
```

- **特质trait**

特质构造顺序

➢ 第2种特质构造顺序(**在构建对象时，动态混入特质**)

1) 调用当前类的超类构造器
2) 当前类构造器
3) 第一个特质构造器的父特质构造器
4) 第一个特质构造器.
5) 第二个特质构造器的父特质构造器, 如果已经执行过，就不再执行
6) 第二个特质构造器
7) .....重复5,6的步骤(如果有第3个，第4个特质)
8) 当前类构造器 [案例演示]

➢ 分析两种方式对构造顺序的影响

第1种方式实际是构建类对象, 在混入特质时，该对象还没有创建。
第2种方式实际是构造匿名子类，可以理解成在混入特质时，对象已经创建了。

- 特质trait

扩展类的特质

➢ 特质可以继承类，以用来拓展该类的一些功能

```
trait LoggedException extends Exception{
  def log(): Unit ={
    println(getMessage()) // 方法来自于Exception类
  }
}
```

- 特质trait

扩展类的特质

➢ 所有混入该特质的类，会自动成为那个特质所继承的超类的子类

```
trait LoggedException extends Exception{
  def log(): Unit ={
    println(getMessage()) // 方法来自于Exception类
  }
}
//UnhappyException 就是Exception的子类.
class UnhappyException extends LoggedException{
   // 已经是Exception的子类了，所以可以重写方法
   override def getMessage = "错误消息！"
}
```

- 特质trait

扩展类的特质

- 如果混入该特质的类，已经继承了另一个类(A类)，则要求A类是特质超类的子类，否则就会出现了多继承现象，发生错误。

```
//正确：因为 IndexOutOfBoundsException 是 LoggedException特质的超类Excetion的子类
class UnhappyException2 extends IndexOutOfBoundsException with LoggedException{
  override def getMessage = "错误信息！"
}
class CCC {

}
//错误：因为 CCC 不是 LoggedException特质的超类Excetion的子类
class UnhappyException3 extends CCC with LoggedException{
  override def getMessage = "错误信息！"
}
```

- 特质trait

自身类型

➢ 说明

自身类型：主要是为了解决特质的循环依赖问题，同时可以确保特质在不扩展某个类的情况下，依然可以做到限制混入该特质的类的类型。

➢ 应用案例

举例说明自身类型特质，以及如何使用自身类型特质

```
//Logger就是自身类型特质
trait Logger {
 // 明确告诉编译器，我就是Exception,如果没有这句话，下面的getMessage不能调用
 this: Exception =>
 def log(): Unit ={
  // 既然我就是Exception, 那么就可以调用其中的方法
  println(getMessage)
 }
}
```

```
class Console extends  Logger {} //对吗?
class Console extends Exception with Logger//对吗?
```

- 嵌套类

基本介绍

在Scala中，你几乎可以在任何语法结构中内嵌任何语法结构。如在类中可以再定义一个类，这样的类是嵌套类，其他语法结构也是一样。
嵌套类**类似于Java中的内部类**。

面试题：Java中，类共有五大成员，请说明是哪五大成员
1. 属性
2. 方法
3. 内部类
4. 构造器
5. 代码块

- 嵌套类

Java内部类的简单回顾

在Java中，一个类的内部又完整的嵌套了另一个完整的类结构。被嵌套的类称为内部类(inner class)，嵌套其他类的类称为**外部类**。内部类最大的特点就是**可以直接访问私有属性**，并且可以体现**类与类之间的包含关系** 【完整详细的回顾看我以前授课视频】

Java内部类基本语法

```
class Outer{              //外部类
      class Inner{        //内部类

      }
}
class Other{              //外部其他类
}
```

- 嵌套类

Java内部类的分类

从定义在外部类的**成员位置**上来看，
1) 成员内部类（没用static修饰）
2) 和静态内部类（使用static修饰），

定义在外部类**局部位置上（比如方法内）来看：**
1) 分为局部内部类（有类名）
2) 匿名内部类（没有类名）

这里我们就回顾一下成员内部类和静态内部类。

- 嵌套类

Java内部类回顾案例

```
package com.atguigu.chapter02;
public class TestJavaClass {
    public static void main(String[] args) {
        //创建一个外部类对象
        OuterClass outer1 = new OuterClass();
        //创建一个外部类对象
        OuterClass outer2 = new OuterClass();
        // 创建Java成员内部类
        // 说明在Java中，将成员内部类当做一个属性，因此使用下面的方式来创建 outer1.new InnerClass().
        OuterClass.InnerClass inner1 = outer1.new InnerClass();
        OuterClass.InnerClass inner2 = outer2.new InnerClass();

        //下面的方法调用说明在java中，内部类只和类型相关，也就是说,只要是
        //OuterClass.InnerClass 类型的对象就可以传给 形参 InnerClass ic
        inner1.test(inner2);
        inner2.test(inner1);

        // 创建Java静态内部类
        // 因为在java中静态内部类是和类相关的，使用 new OuterClass.StaticInnerClass()
        OuterClass.StaticInnerClass staticInner = new OuterClass.StaticInnerClass();
    }}
class OuterClass { //外部类
    class InnerClass { //成员内部类
```

- 嵌套类

Scala嵌套类的使用1

请编写程序，定义Scala 的成员内部类和静态内部类，并创建相应的对象实例。

```
class ScalaOuterClass {
  class ScalaInnerClass { //成员内部类
  }
}
object ScalaOuterClass {  //伴生对象
  class ScalaStaticInnerClass { //静态内部类
  }
}
```

```
 val outer1 : ScalaOuterClass = new ScalaOuterClass();
 val outer2 : ScalaOuterClass = new ScalaOuterClass();

 // Scala创建内部类的方式和Java不一样，将new关键字放置在前，使用 对象.内部类 的方式创建
 val inner1 = new outer1.ScalaInnerClass()
 val inner2 = new outer2.ScalaInnerClass()
 //创建静态内部类对象
 val staticInner = new ScalaOuterClass.ScalaStaticInnerClass()
 println(staticInner)
```

- 嵌套类

Scala嵌套类的使用2

请编写程序，在内部类中访问外部类的属性。

➢ **方式1**

内部类如果想要访问外部类的属性，可以通过外部类对象访问。

即：访问方式：**外部类名.this.属性名**

```
class ScalaOuterClass {
  var name : String = "scott"
  private var sal : Double = 1.2
  class ScalaInnerClass { //成员内部类
    def info() = {
      // 访问方式：外部类名.this.属性名
      // 怎么理解 ScalaOuterClass.this 就相当于是 ScalaOuterClass 这个外部类的一个实例,
      // 然后通过 ScalaOuterClass.this 实例对象去访问 name 属性
      // 只是这种写法比较特别，学习java的同学可能更容易理解 ScalaOuterClass.class 的写法.
      println("name = " + ScalaOuterClass.this.name
        + " age =" + ScalaOuterClass.this.sal)
    }}}
```

```
object ScalaOuterClass { //伴生对象
  class ScalaStaticInnerClass { //静态内部类
  }
}
//调用成员内部类的方法
inner1.info()
```

- 嵌套类

Scala嵌套类的使用2

请编写程序，在内部类中访问外部内的属性。

➢ **方式2**

内部类如果想要访问外部类的属性，也可以通过外部类别名访问(推荐)。

即：访问方式：外部类名别名.属性名 【外部类名.this 等价 外部类名别名】

```
class ScalaOuterClass {
 myOuter => //这样写，你可以理解成这样写，myOuter就是代表外部类的一个对象.
 class ScalaInnerClass { //成员内部类
  def info() = {
   println("name = " + ScalaOuterClass.this.name
    + " age =" + ScalaOuterClass.this.sal)
   println("---------------------------------")
   println("name = " + myOuter.name
    + " age =" + myOuter.sal)
  }}
 // 当给外部指定别名时，需要将外部类的属性放到别名后.
 var name : String = "scott"
 private var sal : Double = 1.2
```

```
object ScalaOuterClass {  //伴生对象
 class ScalaStaticInnerClass { //静态内部类
 }}
inner1.info()
```

- 嵌套类

类型投影

➢ 先看一段代码，引出类型投影

```
class ScalaOuterClass3 {
  myOuter =>
  class ScalaInnerClass3 { //成员内部类
    def test(ic: ScalaInnerClass3): Unit = {
      System.out.println(ic)
    }
  }
}
```

➢ 对上面代码正确和错误的分析.[重点

```
object Scala01_Class {

  def main(args: Array[String]): Unit = {
    val outer1 : ScalaOuterClass3 = new ScalaOuterClass3();
    val outer2 : ScalaOuterClass3 = new ScalaOuterClass3();
    val inner1 = new outer1.ScalaInnerClass3()
    val inner2 = new outer2.ScalaInnerClass3()

    inner1.test(inner1) // ok, 因为 需要outer1.ScalaInner
    inner1.test(inner2) // error, 需要outer1.ScalaInner
outer2.ScalaInner

  }
}
```

```
//说明下面调用test 的 正确和错误的原因：
//1.Java中的内部类从属于外部类,因此在java中 inner.test(inner2) 就可以，因为是按类型来匹配的。
//2 Scala中内部类从属于外部类的对象，所以外部类的对象不一样，创建出来的内部类也不一样，无法互换使
//3. 比如你使用ideal 看一下在inner1.test()的形参上，它提示的类型是 outer1.ScalaOuterClass, 而不是ScalaOuter
```

- 嵌套类

类型投影

- 解决方式-使用类型投影

  类型投影是指：在方法声明上，如果使用 **外部类#内部类** 的方式，表示忽略内部类的对象关系，等同于Java中内部类的语法操作，我们将这种方式称之为 类型投影（即：忽略对象的创建方式，只考虑类型）【案例演示】

谢谢！ 欢迎收看！