6.面向对象

6.1 scala 单例对象

在 Scala 中，是没有 static 这个东西的，但是它也为我们提供了单例模式的实现方法，那就是使用**关键字 object, object 对象不能带参数**。

```
/**
  * 单例对象
  */
object ScalaSingleton {
    def saySomething(msg: String) = {
        println(msg)
    }
}
object test {
    def main(args: Array[String]): Unit = {
        ScalaSingleton.saySomething("滚犊子....")
        println(ScalaSingleton)
        println(ScalaSingleton)
        // 输出结果:
        // 滚犊子....
        // cn.demo.ScalaSingleton$@28f67ac7
        // cn.demo.ScalaSingleton$@28f67ac7
    }
}
```

6.2 scala 类

6.2.1 /类定义/主构造器/辅助构造器

```
/*
* 在 Scala 中，类并不用声明为 public。
* 如果你没有定义构造器，类会有一个默认的空参构造器
*
* var 修饰的变量，这个变量对外提供 getter setter 方法
* val 修饰的变量，对外提供了 getter 方法,没有 setter
* */
class Student {
```

```
    // _ 表示一个占位符，编译器会根据你变量的具体类型赋予相应初始值
    // 注意：使用_ 占位符是，变量类型必须指定
    var name: String = _

    // 错误代码, val 修饰的变量不能使用占位符
    // val age: Int = _

    val age: Int = 10
}
object Test{
    val name: String = "zhangsan"
    def main(args: Array[String]): Unit = {

        // 调用空参构造器,可以加() 也可以不加
        val student = new Student()
        student.name = "laoYang"

        // 类中使用 val 修饰的变量不能更改
        // student.age = 20

        println(s"student.name ====== ${student.name}
${student.age}")
        println("Test.name ======" + Test.name)
    }
}
```

定义在类后面的为类主构造器，一个类可以有多个辅助构造器

```
/*
* Student1(val name: String, var age: Int)
*   定义个 2 个参数的主构造器
* */
class Student1 (val name: String, var age: Int) {

    var gender: String = _

    // 辅助构造器, 使用 def this
    // 在辅助构造器中必须先调用类的主构造器
    def this(name: String, age:Int, gender: String){
        this(name, age)
        this.gender = gender
    }
}
```

```
object Test1{
    def main(args: Array[String]): Unit = {
        val s = new Student1("laoduan", 38)
        println(s"${s.name} ${s.age}")

        val s1 = new Student1("laoYang", 18, "male")
        println(s"${s1.gender}")

    }
}
```

6.2.2 访问权限

- 构造器的访问权限

```
/*
* Student1(val name: String, var age: Int)
*   定义个 2 个参数的主构造器
*
*   private 加在主构造器前面标识这个主构造器是私有的，外部不能访问这个构造器
* */
class Student2 private (val name: String, var age: Int) {
    var gender: String = _

    // 辅助构造器，使用 def this
    // 在辅助构造器中必须先调用类的主构造器
    def this(name: String, age:Int, gender: String){
        this(name, age)
        this.gender = gender
    }
}
object Test2{
    def main(args: Array[String]): Unit = {
        val s1 = new Student2("laoYang", 18, "male")
        println(s"${s1.gender}")
    }
}
```

- 成员变量的访问权限

```
/*
*   private 加在主构造器前面标识这个主构造器是私有的，外部不能访问这个构造器
*
*   private var age
*   age 在这个类中是有 getter setter 方法的
```

```
*   但是前面如果加上了private 修饰，也就意味着，age只能在这个类的内部以及其伴生类对象中可以访问修改
*   其他外部类不能访问
* */
class Student3 private (val name: String, private var age: Int) {

    var gender: String = _

    // 辅助构造器，使用def this
    // 在辅助构造器中必须先调用类的主构造器
    def this(name: String, age:Int, gender: String){
        this(name, age)
        this.gender = gender
    }

    // private[this]关键字标识该属性只能在类的内部访问，伴生类不能访问
    private[this] val province: String = "北京市"

    def getAge = 18
}

// 类的伴生对象
object Student3 {
    def main(args: Array[String]): Unit = {
        // 伴生对象可以访问类的私有方法和属性
        val s3 = new Student3("Angelababy", 30)
        s3.age = 29
        println(s"${s3.age}")
        // println(s"${s3.province}") 伴生类不能访问
    }
}
```

- 类包的访问权限

```
/*
* private[包名] class 放在类声明最前面，是修饰类的访问权限，也就是说类在某些包下不可见或不能访问
*
* private[sheep] class 代表student4在sheep包下及其子包下可以见，同级包中不能访问
*
* */
private[this] class Student4(val name: String, private var age: Int)
{
    var xx: Int = _
}
```

```
object Student4{
    def main(args: Array[String]): Unit = {
        val s = new Student4("张三", 20)

        println(s"${s.name}")
    }
}
```

6.2.3 伴生类/apply 方法

在 Scala 中，当单例对象与某个类共享同一个名称时，他被称作是这个类的伴生对象。必须在同一个源文件里定义类和它的伴生对象。类被称为是这个单例对象的伴生类。类和它的伴生对象可以互相访问其私有成员。

```
/**
  * 定义一个 apply 方法
  * object 对象中可以对 apply 方法各种进行重载
  */
def apply() = {
    println("--------------------")
}
// object 类() 默认调用的 apply()方法
// 不加()时不会调用
val p = Person() //语法糖(sugar)
val p = Person.apply()
```

6.3 特质

Scala Trait(特质) 相当于 Java 的接口，实际上它比接口还功能强大。
与接口不同的是，它还可以定义属性和方法的实现。
一般情况下 Scala 的类只能够继承单一父类，但是如果是 Trait(特质) 的话就可以继承多个，实现了多重继承。**使用的关键字是 trait .**

```
trait T2 {
    // 定义一个属性
    val className: String = "NB 大神班"

    // 定义一个没有实现的方法
    def teacherSay(name: String)

    // 定义个带有具体的实现的方法
    def doSomething() = {
        println("改吃中午饭了...")
```

```
    }
}
```

动态混入特质。

6.4 抽象类

在 Scala 中，使用 abstract 修饰的类称为抽象类. 在抽象类中可以定义属性、未实现的方法和具体实现的方法。

```
/*
* abstract 修饰的类是一个抽象类
* */
abstract class Animal {
    println("Animal's constructor ....")
    // 定义一个 name 属性
    val name: String = "animal"
    // 没有任何实现的方法
    def sleep()
    // 带有具体的实现的方法
    def eat(f: String): Unit = {
        println(s"$f")
    }
}
```

6.5 继承

继承是面向对象的概念，用于代码的可重用性。 被扩展的类称为超类或父类，扩展的类称为派生类或子类。Scala 可以通过使用 extends 关键字来实现继承其他类或者特质。

```
/*
* with 后面只能是特质
*
* 父类已经实现了的功能，子类必须使用 override 关键字重写
* 父类没有实现的方法，子类必须实现
* */
class Dog extends Animal {

    println("Dog's constructor ...")

    override val name: String = "Dog"

    def sleep(): Unit = {
        println("躺着睡...")
    }
```

```
    override def eat(f: String): Unit = {
        println("")
    }
}
```

6.5.1 final 关键字

被 final 修饰的类不能被继承；
被 final 修饰的属性不能重写；
被 final 修饰的方法不能被重写。

6.5.2 type 关键字

Scala 里的类型，除了在定义 class,trait,object 时会产生类型，还可以通过 type 关键字来声明类型。

type 相当于声明一个类型别名：

```
// 把 String 类型用 S 代替
type S = String

val name: S = "小星星"
println(name)
```

通常 type 用于声明某种复杂类型，或用于定义一个抽象类型。

```
114 trait A { type T ; def foo(i:T) = println(i) }
115 class B extends A { type T = Int }
116 val b = new B
117 b.foo(200)
118
119 class C extends A { type T = String }
120 val c = new C
121 c.foo("hello")
```

6.6 样例类/样例对象

```
/*
* 样例类,使用 case 关键字 修饰的类, 其重要的特征就是支持模式匹配
*
* 样例类默认是实现了序列化接口的
```

```
* */
case class Message(msgContent: String)

/**
  * 样例 object，不能封装数据，其重要特征就是支持模式匹配
  */
case object CheckHeartBeat

object TestCaseClass extends App{
    // 可以使用 new 关键字创建实例，也可以不使用
    val msg = Message("hello")
    println(msg.msgContent)
}
```

7. 模式匹配 match case

7.1 匹配字符串/类型/守卫

```
val arr = Array("YoshizawaAkiho", "YuiHatano", "AoiSola")
val i = Random.nextInt(arr.length)
println(i)
val name = arr(i)
println(name)
name match {
  case "YoshizawaAkiho" => println("吉泽老师...")
  case "YuiHatano" => {
    println("波多老师...")
  }
  case _ => println("真不知道你们在说什么...")
}

println("-----------骚气的分割线-----------")

//定义一个数组
val arr:Array[Any] = Array("hello123", 1, 2.0, CaseDemo02, 2L)

//取出一个元素
val elem = arr(3)

elem match {
  case x: Int => println("Int " + x)
  case y: Double if(y >= 0) => println("Double "+ y) // if 守卫
```

```
  case z: String => println("String " + z)
  case w: Long => println("long " + w)
  case CaseDemo02 => {
    println("case demo 2")
    //throw new Exception("not match exception")
  }
  case _ => { // 其他任意情况
    println("no")
    println("default")
  }
}
```

7.2 匹配数组

```
val arr = Array(1, 1, 7, 0, 2,3)
arr match {
  case Array(0, 2, x, y) => println(x + " " + y)
  case Array(2, 1, 7, y) => println("only 0 " + y)
  case Array(1, 1, 7, _*) => println("0 ...") // _* 任意多个
  case _ => println("something else")
}
```

7.3 匹配集合

```
val lst = List(0, 3, 4)

println(lst.head)
println(lst.tail)

lst match {
  case 0 :: Nil => println("only 0")
  case x :: y :: Nil => println(s"x $x y $y")
  case 0 :: a => println(s"value : $a")
  case _ => println("something else")
}
```

7.4 匹配元组

```
val tup = (1, 3, 7)
tup match {
  case (3, x, y) => println(s"hello 123 $x , $y")
```

```
  case (z, x, y) => println(s"$z, $x , $y")
  case (_, w, 5) => println(w)
  case  _ => println("else")
}
```

7.5 匹配样例类/样例对象

```
//样例类，模式匹配，封装数据（多例）,不用 new 即可创建实例
case class SubmitTask(id: String, name: String)

case class HeartBeat(time: Long)

//样例对象，模式匹配（单例）
case object CheckTimeOutTask


val arr = Array(CheckTimeOutTask, new HeartBeat(123),
HeartBeat(88888), new HeartBeat(666), SubmitTask("0001",
"task-0001"))

val i = Random.nextInt(arr.length)
val element = arr(i)
println(element)
element match {
  case SubmitTask(id, name) => {
    println(s"$id, $name")
  }
  case HeartBeat(time) => {
    println(time)
  }
  case CheckTimeOutTask => {
    println("check")
  }
}
```