尚硅谷大数据技术之 Scala
面向对象编程 – 高级

官网：www.atguigu.com

ShangGuigu Technologies Co., Ltd.

尚硅谷技术有限公司

Scala 是基于 Java 的语言，所以基本的语法非常的类似，但是，为了能好的体现，高效，简单，灵活等特性，Scala 语言做了极大的努力。对 Java 的很多语法进行了改进。

一 类

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。

```
// Java
class OuterClass {
    class InnerClass {
    }
    static class StaticInnerClass {
    }
}
// 创建对象
OuterClass outer = new OuterClass();
OuterClass.InnerClass inner = outer.new InnerClass(); // 很诡异的写法
OuterClass.StaticInnerClass staticInner = new OuterClass.StaticInnerClass();
```

```
// Scala
class OuterClass {
    class InnerClass {
    }
    //static class StatisInnerClass { // Scala 中没有静态的概念，所以静态内部类不是放在类中声明，而是放置在类的伴生对象中声明
    //}
}
```

```
object OuterClass {
    class StaticInnerClass {
    }
}
val outer = new OuterClass()
val inner = new outer.InnerClass()

val staticInner = new OuterClass.InnerClass()
```

内部类如果想要访问外部类的属性，可以通过外部类对象或外部类别名访问

```
    class OuterClass {
    var name = "zhangsan"
    class InnerClass {
        // 访问方式：外部类名.this.属性名
        def info = println("name = " + OuterClass.this.name)
    }
}
```

```
Class OuterClass {
    // 给外部类对象声明别名
    outer =>
    class InnerClass {
        def info = println("name = " + outer. name)
    }
    var name = "lisi"
}
```

注：Java 中的内部类从属于外部类。Scala 中内部类从属于实例。

二 对象

1 构造器

和 Java 一样，Scala 构造对象也需要调用构造方法，并且可以有任意多个构造方法，不过，Scala 类有一个构造方法比其他所有的构造方法都更为重要，我们称之为主构造器，其他的构造器我们称之为辅助构造器

1) 主构造器的声明直接放置于类名之后

```
class ConstructorClass() {
    // 类体
}
val obj = new ConstructorClass() // 当使用 new 构建对象时，等同于调用类的主构造器
// 这种构造方式类似于 JavaScript 中对象的构建
```

2) 主构造器会执行类定义中的所有语句

```
class ConstructorClass() {
    // 执行主构造器时，类体中所有语句都会执行
}
val obj = new ConstructorClass()
```

3) 主构造器传递参数

```
// 如果主构造器无参数，小括号可省略
// 构造器也是方法（函数），传递参数和使用方法和前面的函数部分内容没有区别
class ConstructorClass( p : String ) {
    println("参数为 = " + p)
}
val obj = new ConstructorClass("123")
```

4) 辅助构造器名称为 this（这个和 Java 是不一样的），多个辅助构造器通过不同参数列表进行区分

```
class Person() {
    private var sname = ""
    private var iage = 0
    // 辅助构造器无论是直接或间接，最终都一定要调用主构造器，执行主构造器的逻辑
    def this( name : String ) {
        this() // 放在第一行
        sname = name
    }
    def this( name : String, age : Int ) {
        this(name) //调用之前已经声明过该构造器
        iage = age
    }
}
```

5) 如果想让主构造器变成私有的，可以在()之前加上 **private**，这样用户只能通过辅助构造器来构造对象了

```
class ConstructorClass private () {
}
val obj = new ConstructorClass() // (X)
```

2 单例对象

如果构造方法私有化，那么想要在类的外部构建对象是不可能的，所以需要通过静态方法获取类的对象，如果获取对象时，这个对象只创建一次，那么这样的对象我们称之为单例对象

```
// Java
class Singleton {
    private Singleton() {}
```

```
    private static class SingletonInstance {
        private static final Singleton INSTANCE = new Singleton();
    }
    public static Singleton getInstance() {
        return SingletonInstance.INSTANCE;
    }
}
```

```
// Scala
// Scala 中没有静态的概念，所以为了实现 Java 中单例模式的功能，可以直接采用类对象方式构建单例对象
// object Singleton {}
```

3 伴生对象

在 Scala 中，如果在同一个源码文件中，同时存在使用 object 声明的类对象（Person）以及使用 class 声明的类(Person)，那么这个类对象就是该类的**伴生对象**，而这个类就是这个类对象的**伴生类**。

```
// person.scala
object Person{ //  类对象
}
```

```
// person.scala
class Person { //  伴生类
}
object Person { //  伴生对象
}
```

注：类和它的伴生对象可以相互访问私有属性或方法，他们必须存在同一个源文件中。必须同名。

从技术的角度来讲，伴生对象和伴生类其实是两个不同的类，伴生对象所对应的类可以简答的理解为伴生类的辅助工具类。而伴生对象就是这个辅助工具类的单例对象，

专门用于处理伴生类中静态资源的相关功能.

伴生对象既然是一个单例对象，就意味着类已经声明过了，所以，伴生（类）对象是不能声明构造器的

4 apply 方法

因为伴生对象可以处理静态资源，所以完全可以通过伴生对象提供的方法对伴生类进行处理

增加 apply 方法，构建伴生类实例

```
object Person {
    def apply() = { // 在伴生对象中实现 apply 方法。
        return new Person() // 这里也可以采用单例模式实现
    }
}
…
val p = Person() // 此处不需要使用 new 关键字，而是直接增加小括号，在执行时，会自动调用伴生对象中的 apply 方法。返回伴生类实例对象
```

5 应用程序对象

每一个 Scala 应用程序都需要从一个对象的 main 方法开始执行，这个方法的类型为 Array[String]=>Unit：

```
object ScalaApp{
  def main(args: Array[String]) {
    println("Hello, Scala!")
  }
}
```

或者扩展一个 App 特质：

```
object ScalaApp extends App {
  println("Hello, Scala!")
}
```

6 枚举对象

Scala 中没有枚举类型，定义一个扩展 Enumeration 类的对象，并以 Value 调用初始化枚举中的所有可能值,模拟枚举类型。

```
object LightColorEnum extends Enumeration {
    val Red = Value(0, "Stop")
    val Yellow = Value(1, "Slow")
    val Green = Value(2, "Go")
}
```

三 属性

1 构造器参数

Scala 类的主构造器函数是可以添加参数的。如果参数未用任何修饰符修饰，那么这个参数是局部变量

```
class Person( name : String ) { // 此处的 name 未加任何修饰，是局部变量，无法通过对象进行访问
}
var p = new Person("lisi")
println(p.name) (X)
```

如果参数使用 val 关键字声明，那么 Scala 会将参数作为类的只读属性（不能修改）使用

```
class Person( val name : String ) {
}
var p = new Person("lisi")
println(p.name) (OK) // 这里调用对象的 name 属性，其实并不是属性，而是方法，因为方法无参数，所以省略了小括号，感觉和调用属性一样，这体现了 Scala 语言访问一致性
```

如果参数使用 var 关键字声明，那么那么 Scala 会将参数作为类的成员属性使用,并会提供属性对应的 setter/getter 方法

```
class Person( var name : String ) {
}
var p = new Person("lisi")
p.name = "wangwu" // setter
println(p.name)    //getter
```

2 Bean 属性

JavaBeans 规范定义了 Java 的属性是像 getXXX（）和 setXXX（）的方法。许多 Java

工具（框架）都依赖这个命名习惯。为了 Java 的互操作性。将 Scala 字段加@BeanProperty 时，这样会自动生成 setter/getter 方法。

```
import scala.beans.BeanProperty
class Person {
    @BeanProperty var name: String = null
}
var p = new Person()
p.setName("zhaoliu")
println(p.getName())
```

四 方法（请参考函数）

五 包

1 包/作用域

在 Java 和 Scala 中管理项目可以使用包结构，C 和 C#使用命名空间。

对于 package，有如下几种形式：

1) 形式体现：

```
package com.atguigu.scala
class Person{
    val name = "Nick"
    def play(message: String): Unit ={
    }
}
```

等同于：

```
package atguigu
package scala
class Person{
    val name = "Nick"
    def play(message: String): Unit ={
    }
}
```

等同于：

```
package com.atguigu{
    package scala{
        class Person{
            val name = "Nick"
            def play(message: String): Unit ={
            }
```

```
    }
  }
}
```

注：位于文件顶部不带花括号的包声明在对当前整个文件内的包声明有效。

通过以上形式，总结如下：

1、 包也可以像嵌套类那样嵌套使用（包中有包）。

2、 作用域原则：可以直接向上访问。即，子包中直接访问父包中的内容。(即：作用域)

3、 源文件的目录和包之间并没有强制的关联关系

4、 可以在同一个.scala 文件中，声明多个并列的 package

5、 包名可以相对也可以绝对，比如，访问 BeanProperty 的绝对路径是：

root. scala.beans.BeanProperty

2 包对象

包可以包含类、对象和特质 trait，但不能包含函数或变量的定义。很不幸，这是 Java 虚拟机的局限。为了弥补这一点不足，scala 提供了**包对象**的概念来解决这个问题

```
package com.atguigu {
    package scala {
        class Test {
            def test() : Unit ={
                println(name)
            }
        }
    }
    package object scala { //每个包都可以有一个包对象。你需要在父包中定义它,且名称与子包一样。
```

```
        var name = "zhangsan"
    }
}
```

3 包可见性

在 Java 中，访问权限分为种：public, private,protected, 缺省的。在 Scala 中，你可以通过类似的修饰符达到同样的效果。但是使用上有区别。

1) 当访问权限缺省时，scala 默认为 public 访问权限

```
class Person {
    var name = "zhangsan" // 此时属性为 public 访问权限,任何地方都能访问
}
```

2) 私有权限

```
class Person {
    private var name = "zhangsan"
    // 此时属性为私有的，只在类的内部和伴生对象中可用
}
```

3) 受保护权限

```
class Person {
    protected var name = "zhangsan"
    // scala 中受保护权限比 Java 中更严格，只能子类访问，同包无法访问
}
```

4) 包访问权限

```
package com.atguigu.scala
class Person {
  private[scala] val pname="zhangsan" // 增加包访问权限后，private 同时起作用
}
```

当然，也可以将可见度延展到上层包：

```
private[atguigu] val description="zhangsan"
```

4 引入

因为 Scala 语言源自于 Java，所以 **java.lang** 包中的类会自动引入到当前环境中，而 Scala 中的 **scala** 包和 **Predef** 包的类也会自动引入到当前环境中。如果想要把其他包中的类引入到当前环境中，需要使用 import 语言

1) 在 Scala 中，import 语句可以出现在任何地方，并不仅限于文件顶部。

```
class User {
    import scala.beans.BeanProperty
    @BeanProperty var    name : String = ""
}
```

注：import 语句的效果一直延伸到包含该语句的块末尾

2) 在 Java 中如果想要导入包中所有的类，可以通过通配符星号，Scala 中采用下划线

```
class User {
    import scala.beans._ // 这里采用下划线作为通配符
    @BeanProperty var    name : String = ""
}
```

3) 如果不想要某个包中全部的类，而是其中的几个类，可以采用选取器（大括号）

```
def test(): Unit = {
    import scala.collection.mutable.{HashMap, HashSet}
    var map = new HashMap()
    var set = new HashSet()
}
```

5 重命名和隐藏

如果引入的多个包中含有相同的类，那么可以将不需要的类进行重命名进行区分

重命名：

```
import java.util.{ HashMap=>JavaHashMap, List}
import scala.collection.mutable._
```

```
var map = new HashMap() // 此时的 HashMap 指向的是 scala 中的 HashMap
var map1 = new JavaHashMap(); // 此时使用的 java 中 hashMap 的别名
```

这样一来，JavaHashMap 就是 java.utiI.HashMap，而 HashMap 则对应 scala.collection.mutable.HashMap。

如果某个冲突的类根本就不会用到，那么这个类可以直接隐藏掉。

```
import java.util.{ HashMap=>_, _}
var map = new HashMap() // 此时的 HashMap 指向的是 scala 中的 HashMap
```

六 继承

和 Java 一样使用 extends 关键字，在定义中给出子类需要而超类没有的字段和方法，或者重写超类的方法。

如果类声明为 final，他将不能被继承。如果单个方法声明为 final，将不能被重写

scala 明确规定，只要出现 override 情况，一定要显式声明关键字 override

1 重写方法

重写一个非抽象方法需要用 override 修饰符，调用超类的方法使用 super 关键字

```
class Person {
    var name : String = "zhangsan"
    def printName() {
        println(name);
    }
}
class Emp extends Person {
    override def printName() {
        println(name);
    }
}
```

2 类型检查和转换

要测试某个对象是否属于某个给定的类，可以用 **isInstanceOf** 方法。用 **asInstanceOf** 方法将引用转换为子类的引用。classOf 获取对象的类名。

1) classOf[String]就如同 Java 的 String.class
2) obj.isInstanceOf[T]就如同 Java 的 obj instanceof T
3) obj.asInstanceOf[T]就如同 Java 的(T)obj

```
println("Hello".isInstanceOf[String])
println("Hello".asInstanceOf[String])
```

```
println(classOf[String])
```

3 受保护的字段和方法

protected 在 scala 中比 Java 要更严格一点，即， 只有继承关系才可以访问，同一个包下，也是不可以的。具体请参考：包可见性

4 超类的构造

类有一个主构器和任意数量的辅助构造器，而每个辅助构造器都必须以对先前定义的辅助构造器或主构造器的调用开始。**子类的辅助构造器最终都会调用主构造器**。

```
class Person {
    var name = "zhangsan"
}
class Emp extends Person {
    def this(name : String) {
        this // 必须调用主构造器
        this.name = name
    }
}
```

只有主构造器可以调用超类的构造器。辅助构造器永远都不可能直接调用超类的构造器。在 Scala 的构造器中，你不能调用 super(params)。

```
class Person(name: String) {
}
class Emp (name: String) extends Person(name) {// 将子类参数传递给父类构造器
  // super(name)   (X) 没有这种语法
}
```

5 覆写字段

子类改写父类的字段

```
class Person(){
    println("父类构造器")
    val name="zhangsan"
}
class Emp extends Person{
    val name: String = "lisi"
}
```

注：

1、def 只能重写另一个 def

2、val 只能重写另一个 val 或不带参数的 def

3、var 只能重写另一个抽象的 var（**声明未初始化的变量就是抽象的变量**）

```
class Person {
        var name : String //  此处属性未初始化，为抽象的。
}
class Emp extends Person {
        var name : String = ""
}
```

6 抽象类

可以通过 abstract 关键字标记不能被实例化的类。方法不用标记 abstract，只要省掉方法体即可。抽象类可以拥有抽象字段，抽象字段就是没有初始值的字段。

```
abstract class Person() { //  抽象类
    var name: String //  抽象字段
    def printName //  抽象方法
}
class Emp extends Person {
    var name = "zhangsan";
    def printName = name
```

```
}
```

注：子类重写抽象方法不需要 override

7 匿名子类

和 Java 一样，你可以通过包含带有定义或重写的代码块的方式创建一个匿名的子类：

```
abstact class Person{
    var name : String
}
```

使用：

```
var p = new Person {
    var name: String = null
}
```

8 构造顺序和提前定义

当子类重写了父类的方法或者字段后，父类又依赖这些字段或者方法初始化，这个时候就会出现问题，比如：

```
class Person{
  val prefix = "person_"
  var name = prefix + "name"
}
class Emp extends Person {
  override val prefix = "emp_"
}
```

请问：name 字段的值是什么？

问题解决的 3 种方案：

1) 可以将 val 声明为 final，这样子类不可改写。

2) 可以将超类中将 val 声明为 lazy，这样安全但并不高效。

3) 还可以使用提前定义语法，可以在超类的构造器执行之前初始化子类的 val 字段：

```
class Emp extends {
    override val prefix = "emp_" // 这样的写法让人无法忍受，不推荐使用
} with Person
```

9 继承层级

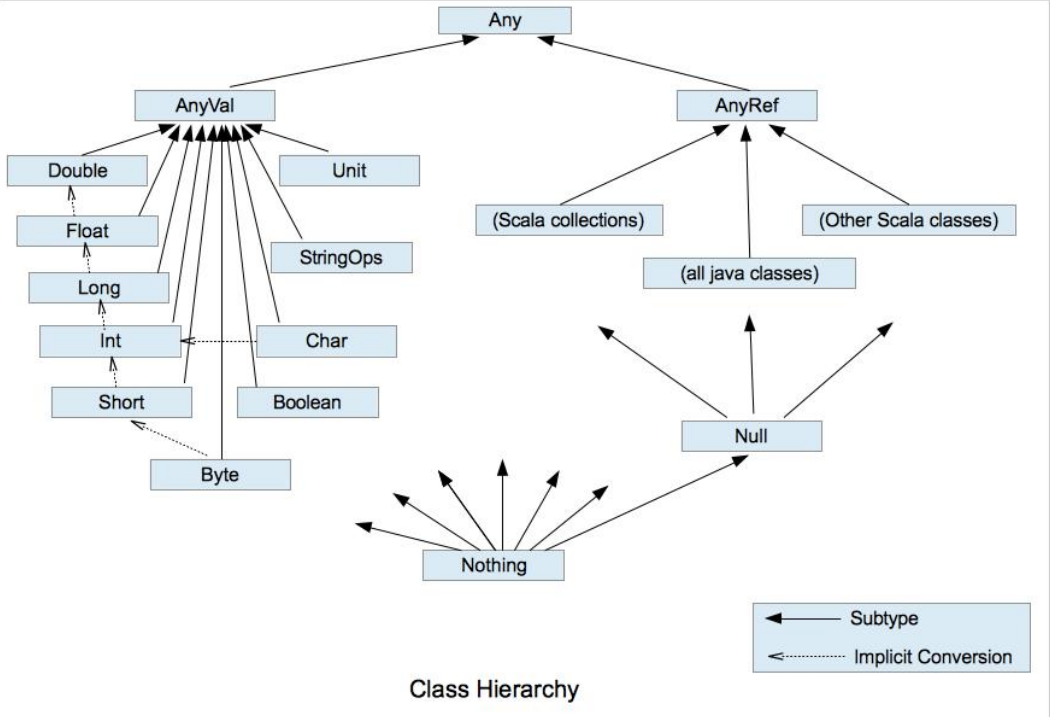
在 scala 中，所有其他类都是 AnyRef 的子类，类似 Java 的 Object。

AnyVal 和 AnyRef 都扩展自 Any 类。Any 类是根节点

Any 中定义了 isInstanceOf、asInstanceOf 方法，以及哈希方法等。

Null 类型的唯一实例就是 null 对象。可以将 null 赋值给任何引用，但不能赋值给值类型的变量。

Nothing 类型没有实例。它对于泛型结构是有用处的，举例：空列表 Nil 的类型是 List[Nothing]，它是 List[T]的子类型，T 可以是任何类。



七 特质

所有的面向对象的语言都不允许直接的多重继承，因为会出现“deadly diamond of death”问题。Scala 提供了特质（trait），特质可以同时拥有抽象方法和具体方法，一个类可以实现多个特质。

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。

```
trait Logger {
   def log(msg: String)
}
// 所有的 java 接口都可以当做 Scala 特质使用
class Console extends Logger with Cloneable with Serializable {
   def log(msg: String) {
      println(msg)
   }
}
```

1 带有具体实现的特质

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，也可以有默认实现

```
trait DBOperate {
      def insert( id : Int ) {
            println("插入主键【"+id+ "】数据" )
      }
}
class MySQL extends DBOperate {
}
var mysql = new MySQL
mysql.insert(1)
```

2 带有特质的对象，动态混入

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

```
trait DBOperate {
    def insert( id : Int ) {
        println("插入主键【"+id+ "】数据" )
    }
}
class MySQL {
}
var db = new MySQL with DBOperate
db.insert(1)
```

此种方式也可以应用于对抽象类功能进行扩展

3 叠加在一起的特质

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

```
trait Operate {
    println("Oper...")
    def insert( id : Int )
}
trait Data extends Operate {
    println("Data...")
    def insert( id : Int ) {
        println("插入数据【"+id+ "】" )
    }
}
trait File extends Data {
    println("File...")
```

```
        override def insert( id : Int ) {
            print("向文件" )
            super.insert(id)
        }
}
trait DB extends Data {
        println("DB...")
        override def insert( id : Int ) {
            print("向数据库" )
            super.insert(id)
        }
}
abstract class MySQL extends Operate {
}
var db = new MySQL with DB with File
db.insert(2)
```

此时会发现 super 关键字并不是我们理解的调用父特质的方法，而是调用下一个特质的同名方法（如果存在的话）

如果想要调用具体特质的方法，可以指定：super[Data].xxx(…).其中的泛型必须是该特质的直接超类类型

```
trait File extends Data {
        println("File...")
        override def insert( id : Int ) {
            print("向文件" )
            super[Data].insert(id)
        }
}
```

```
trait DB extends Data {
    println("DB...")
    override def insert( id : Int ) {
        print("向数据库" )
        super[Data].insert(id)
    }
}
```

4 在特质中重写抽象方法

如果调用父特质的抽象方法，为了保证一定要有方法实现，可以让当前特质重写父特质中的抽象方法。防止混入特质时出现错误

```
trait Operate {
    def insert( id : Int )
}
trait Data extends Operate {
    // abstract override def insert( id : Int )
    def insert( id : Int ) {
        super.insert(id) // 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，就必须实现该方法，就不会出现错误了，参考特质 DB
    }
}
trait DB extends Operate {
    def insert( id : Int ) {
        println("id =" + id)
    }
}
var db = new MySQL with DB with Data // 顺序不能写错
```

```
db.insert(3)
```

5 当作富接口使用的特质

即该特质中既有抽象方法，又有非抽象方法

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

6 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，而是简单的加入类。是自己的字段。

```
trait Operate {
    val opertype : String = "insert"
    def insert( id : Int )
    def pageQuery(pageno:Int, pagesize:Int): Unit = {
        println("分页查询")
    }
}
```

7 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。

```
trait Operate {
    val opertype : String // 抽象字段
}
class MySQL extends Operate {
    val opertype : String = "" // 给抽象字段初始化
```

```
}
```

8 特质构造顺序

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“叠加在一起的特质”

步骤总结：

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)、当前类构造器

9 初始化特质中的字段

在某些情况下，Scala 特质和 Java 抽象类是非常类似的，所以也有可能会在声明属性时，出现属性值发生改变的情况，问题请参考“构造顺序和提起定义”

10 扩展类的特质

1) 特质可以继承自类，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
  }
}
```

2) 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

```
class UnhappyException extends LoggedException{
  override def getMessage = "错误消息！"// 已经是 Exception 的子类了，所以可以重写方法
}
```

3) 如果混入该特质的类，已经继承了另一个类，不就矛盾了？注意，只要继承的那个

类是特质超类的子类即可。

正确：

```
class UnhappyException2 extends IndexOutOfBoundsException with LoggedException{
    override def getMessage = "错误信息！"
}
```

错误：

```
class UnhappyException3 extends JFrame with LoggedException{
    override def getMessage = "错误信息！"
}
```

11 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。

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

这样一来，在该特质中，可以随意调用“自身类型”中的各种方法。

```
class MySQL extends Exception { // 此处必须继承 Exception 类，否则无法混入 logger 特质
}
var db = new MySQL with Logger
db.log
```

八 隐式转换和隐式参数

1 隐式转换

隐式转换函数是以 implicit 关键字声明的带有单个参数的函数。这种函数将会自动应用，将值从一种类型转换为另一种类型。

```
val i1: Int = 3.5 //（X）类型不匹配，无法转换
println(i1)
```

```
implicit def a(d: Double) = d.toInt // 声明隐式函数
val i1: Int = 3.5 //（OK）当发现程序有误时，Scala 编译器会尝试在隐式函数列表中查询可以进行转换的函数
println(i1)
```

Scala 进行编译时，会自动识别后将程序进行转换，开发人员无需进行操作。

需要保证在当前环境下，只有一个隐式函数能被识别

```
implicit def a(d: Double) = d.toInt
implicit def b(d: Double) = d.toInt
val i1: Int = 3.5 //（X）在转换时，识别出有两个方法可以被使用，就不确定调用哪一个，所以出错
println(i1)
```

2 利用隐式转换丰富类库功能

如果需要为一个类增加一个方法，可以通过隐式转换来实现。比如想为 MySQL 增加一个 delete 方法

```
class MySQL {
    def insert( id : Int ): Unit = {
        println("向数据库中插入数据：" + id)
    }
}
```

```
var mysql = new MySQL()
mysql.insert(1)
// 在当前程序中，如果想要给 MySQL 类增加功能是非常简单的，但是在实际项目中，如果想要增加新的功能就会需要改变源代码，这是很难接受的。而且违背了软件开发的 OCP 开发原则
// 在这种情况下，可以通过隐式转换函数给类动态添加功能。
class DB {
    def delete( id : Int ): Unit = {
        println("从数据库中删除数据：" + id)
    }
}
val mysql = new MySQL
implicit def addFun( db : MySQL ) : DB ={
    new DB()
}
mysql.delete(1)
```

3 隐式值

将 name 变量标记为 implicit，所以编译器会在方法省略隐式参数的情况下去搜索作用域内的隐式值作为缺少参数。

```
implicit val name = "Nick"
def person(implicit name: String) = name
println(person)
```

但是如果此时你又相同作用域中定义一个隐式变量，再次调用方法时就会报错：出现二义性

```
implicit val name = "Nick"
implicit val name2 = "Nick"
def person(implicit name: String) = name
```

```
println(person)
```

4　隐式类

在 scala2.10 后提供了隐式类，可以使用 implicit 声明类，但是需要注意以下几点：

1) 其所带的构造参数有且只能有一个

2) 隐式类必须被定义在“类”或“伴生对象”或“包对象”里

3) 隐式类不能是 case class（case class 在后续介绍）

4) 作用域内不能有与之相同名称的标示符

```
object StringUtils {
  implicit class StringImprovement(val s : String){ //隐式类
    def addSuffix = s + " Scala"
  }
}
println("Hello". addSuffix)
```

5　隐式的转换时机

1) 当方法中的参数的类型与目标类型不一致时

2) 当对象调用所在类中不存在的方法或成员时，编译器会自动将对象进行隐式转换

6　隐式解析机制

即编译器是如何查找到缺失信息的，解析具有以下两种规则：

1) 首先会在当前代码作用域下查找隐式实体（隐式方法、隐式类、隐式对象）。

2) 如果第一条规则查找隐式实体失败，会继续在隐式参数的类型的作用域里查找。类型的作用域是指与该类型相关联的全部伴生模块，一个隐式实体的类型 T 它的查找范围如下：

a) 如果 T 被定义为 T with A with B with C,那么 A,B,C 都是 T 的部分，在 T 的隐式解析过程中，它们的伴生对象都会被搜索。

b) 如果 T 是参数化类型，那么类型参数和与类型参数相关联的部分都算作 T 的部分，比如 List[String]的隐式搜索会搜索 List 的伴生对象和 String 的伴生对象。

c) 如果 T 是一个单例类型 p.T，即 T 是属于某个 p 对象内，那么这个 p 对象也会

被搜索。

d) 如果 T 是个类型注入 S#T，那么 S 和 T 都会被搜索。

7 隐式转换的前提

1) 不能存在二义性

2) 隐式操作不能嵌套