9. Scala 高级语法

9.1 隐式（implicit）详解

思考： 我们调用别人的框架，发现少了一些方法，需要添加，但是让别人为你一个人添加是不可能滴。

比如使用 java.io.File 读取文件非常的繁琐，能不能让 Oracle 公司给我们再添加一个 read 方法，直接返回文件中的所有内容，我想请问你脸大么？

伟大领袖毛主席教育我们说：“自己动手丰衣足食。”。

掌握 implicit 的用法是阅读 spark 源码的基础，也是学习 Scala 其它的开源框架的关键，implicit 可分为：

- 隐式参数
- 隐式转换类型
- 隐式类

9.1.1 隐式参数

定义方法时，可以把参数列表标记为 implicit，表示该参数是隐式参数。一个方法只会有一个隐式参数列表，置于方法的最后一个参数列表。如果方法有多个隐式参数，只需一个 implicit 修饰即可。

譬如：def fire(x: Int)(implicit a:String, b: Int = 9527)

当调用包含隐式参数的方法是，如果当前上下文中有合适的隐式值，则编译器会自动为该组参数填充合适的值，**且上下文中只能有一个符合预期的隐式值**。如果没有编译器会抛出异常。当然，标记为隐式参数的我们也可以手动为该参数添加默认值。

9.1.2 隐式的转换类型

使用隐式转换将变量转换成预期的类型是编译器最先使用 implicit 的地方。当编译器看到类型 X 而却需要类型 Y，它就在当前作用域查找是否定义了从类型 X 到类型 Y 的隐式定义。

```
val i: Int = 3.1415 // 此时程序会报错，因为声明的为 Int 类型
```

添加如下代码：

```
implicit def double2Int(d: Double) = d.toInt   // 在运行代码则没有错误，隐式方法
val i : Int = 3.5 // 成熟输出 3
```

现在我们就可以给给 File 类添加一个 read 方法，返回一个文件中的所有内容。

定义一个 RichFile 类及伴生对象，在伴生对象中定义一个隐式方法（file -> RichFile），在伴生

类中定义一个 read 方法并返回文件中的所有内容。

9.1.3 隐式类

在 Scala 中，我们可以在静态对象中使用隐式类。

```
// 自定义的隐式类
implicit class Calculator(x: Int){
    def add_10(a: Int): Int = (x + a) * 10
}
def main(args: Array[String]): Unit = {
    // 9.1.3.隐式类：
    // 为 Int 类型，添加一个自定义的 add_10 方法
    // 调用隐式类
    val z: Int = 6
    println(z.add_10(9))
}
```

9.2 泛型

通俗的讲，比如需要定义一个函数，函数的参数可以接受任意类型。我们不可能一一列举所有的参数类型重载函数。
那么程序引入了一个称之为泛型的东西，这个类型可以代表任意的数据类型。
例如 List，在创建 List 时，可以传入整形、字符串、浮点数等等任意类型。那是因为 List 在类定义时引用了泛型。

```
// Scala 枚举类型
object Em extends Enumeration {
    type Em = Value
    val 上衣, 内衣, 裤子, 袜子 = Value
}

// 在 Scala 定义泛型用[T]， s 为泛型的引用
abstract class Message[T](s: T) {
    def get: T = s
}
// 子类扩展的时候，约定了具体的类型
class StrMessage[String](msg: String) extends Message(msg)

class IntMessage[Int](msg: Int) extends Message(msg)

// 定义一个泛型类
class Clothes[A,B,C](val clothesType: A, var color: B, var size: C)

object Test {
```

```
    def main(args: Array[String]): Unit = {
        val s = new StrMessage("i hate you !")
        val i = new IntMessage(100)
        println(s.get)
        println(i.get)

        // 创建一个对象， 传入的参数为 Em, String, Int
        val c1 = new Clothes(Em.内衣, "Red", 36)
        c1.size = 38
        println(c1.clothesType, c1.size)

        // new 的时候，指定类型。那么传入的参数，必须是指定的类型
        val c2 = new Clothes[Em, String, String](Em.上衣, "黑色", "32B")
        println(c2.size)

        // 定义一个函数，可以获取各类 List 的中间位置的值
        val list1 = List("a","b","c")
        val list2 = List(1,2,3,4,5,6)

        // 定义一个方法接收任意类型的 List 集合
        def getData[T](l: List[T])={
            l(l.length/2)
        }
    }
}
```

9.3 类型约束

9.3.1 上界(Upper Bounds)/下界(lower bounds)

- **Upper Bounds**

在 Java 泛型里表示某个类型是 Test 类型的子类型，使用 extends 关键字：

```
<T extends Test>

//或用通配符的形式：
<? extends Test>
```

这种形式也叫 upper bounds(上限或上界)，同样的意思在 Scala 的写法为：

```
[T <: Test]

//或用通配符:
[_ <: Test]
```

```
def pr(list : List[_ <: Any]) {
    list.foreach(print)
}
```

- **Lower Bounds**

在 Java 泛型里表示某个类型是 Test 类型的父类型，使用 super 关键字：

```
<T super Test>

//或用通配符的形式：
<? super Test>
```

这种形式也叫 lower bounds(下限或下界)，同样的意思在 scala 的写法为：

```
[T >: Test]

//或用通配符:
[_ >: Test]
```

示例：

```
// 定义一个类， 用来比较 2 个数字的大小
class CmPareInt(first:Int, second: Int) {
    def bigger = if (first >  second) first else second
}
// 定一个类， 比较 2 个字符串的大小
class CmPareString(first: String, second: String) {
    def bigger = if (first >  second) first else second
}

// 如果还要定义 Double 类型的比较，也许还需要比较 2 个类的比较，咋办，还重复的劳动吗？
// 其实我们可以使用泛型，但是泛型类型必须实现了 Comparable，相当于约束了泛型的范围
//  T <: Comparable[T] 表示 T 类型是 Comparable 的实现类
//         <: 在 Scala 中叫上界 upper bounds,类似 Java CmPare<T extends
Comparable<T>>
//  T >: xxx[T] 表示 T 类型是 xxx 的超类
//         >: 在 Scala 中叫下界 lower bounds，类似 Java CmPare<T super xxx<T>>
class CmPare[T <: Comparable[T]](first: T, second: T) {
    // 此时使用 > 会报错， 原因是 T 类型没有 > 方法
    // def bigger = if (first >  second) first else second

    // 其实,我们可以在约定下传入的T必须是可比较的，也就意味着 T 实现了 Comparable 接口，即
T <: Comparable[T]
    def bigger = if (first.compareTo(second) > 0) first else second
}

val cpi = new CmPareInt(4, 6).bigger
```

```
val cps = new CmPareString("Hadoop", "Hive").bigger

// 这样就更通用了，这里必须使用装箱类型， 因为 Int 没有实现 Comparable 接口
val cpiv = new CmPare(Integer.valueOf(4), Integer.valueOf(6)).bigger
val cpsv = new CmPare("Hadoop", "Hive").bigger
println(cpsv)
```

9.3.2 视图界定/上下文界定

- **View bounds**

<% 的意思是“view bounds”(视界)，它比<:适用的范围更广，除了所有的子类型，还允许隐式转换过去的类型。

def method [A <% B](arglist): R = ...

等价于:

def method [A](arglist)(implicit viewAB: A => B): R = ...

或等价于:

implicit def conver(a:A): B = …

<% 除了方法使用之外，class 声明类型参数时也可使用：

class A[T <% Int]

示例：

```
// <% : 视图界定
class CmPare2[T <% Ordered[T]](first: T, second: T) {
    // 如果你觉得 compareTo 看着不爽， 就想使用 > 、< 这种， 那么你可以将 Comparable[T] 改成 Ordered[T]
    def bigger = if (first > second) first else second
}

// char 类型 没有实现 Comparable 接口， 需要隐式的进行装箱操作 char -> Character
// 此时就不能使用 <: 上界， 的使用视图界定 <%
val bigger2 = new CmPare2('a', 'f').bigger
println(bigger2)
```

- **Context bounds**

与 view bounds 一样 context bounds(上下文界定)也是隐式参数的语法糖。为语法上的方便，引入了”上下文界定”这个概念。

```
implicit val cp = new Comparator[Int] {
    override def compare(o1: Int, o2: Int): Int = o1 - o2
}
```

```
// 求 2 个参数的最大值， 还有一个隐式参数 cp，调用者在调用该方法时，编译器会自动从上下文中找这样的隐式参数
def max[T](a: T, b: T)(implicit cp: Comparator[T]): T = {
    if (cp.compare(a, b) > 0) a else b
}


def max2[T: Comparator](a: T, b: T): T = {
    // 这种做法只是把外部方法的隐式参数隐藏了，放到内部嵌套函数上
    def inner(implicit c:Comparator[T]) = c.compare(a,b);
    if(inner>0) a else b
}
```