

# Scala 函数式编程

## 目录

1、高阶函数和闭包.....	2
1.1、定义函数.....	2
1.1.1、匿名函数.....	2
1.1.2、函数的各种写法.....	2
1.2、高阶函数.....	2
1.2.1、函数返回值为函数.....	3
1.2.2、函数/方法的参数为函数.....	3
1.3、闭包.....	5
1.4、Scala 柯里化 Curry.....	6
2、Scala 隐式转换和隐式参数.....	7
2.1、Scala 隐式转换探讨.....	7
2.2、隐式转换的发生时机.....	12
2.2.1、时机一：当调用某个对象不存在的方法时.....	12
2.2.2、时机二：当方法参数类型不匹配时.....	13
2.2.3、时机三：在视图边界的时候.....	15
2.3、隐式转换忠告.....	15
3、Scala 泛型.....	15
3.1、Scala 泛型基础.....	15
3.2、Scala 类型变量界定.....	15
3.3、Scala 视图界定.....	17
3.4、Scala 上界下界.....	18
3.4.1、上界.....	18
3.4.2、下界.....	19
3.5、Scala 逆变和协变.....	20
3.5.1、协变.....	20
3.5.2、逆变.....	23
3.5.3、协变和逆变总结.....	26

# 1、高阶函数和闭包

## 1.1、定义函数

```
scala> val add = (x:Int, y:Int) => {x + y}
add: (Int, Int) => Int = <function2>

scala> array.reduce(add)
res5: Int = 10

scala> val max = (x:Int, y:Int) => {if (x>y) x else y}
max: (Int, Int) => Int = <function2>

scala> array.reduce(max)
res6: Int = 4

scala>
```

### 1.1.1、匿名函数

```
scala> val array = Array(1,2,3,4)
array: Array[Int] = Array(1, 2, 3, 4)

scala> array.map((x:Int) => x + 1)
res0: Array[Int] = Array(2, 3, 4, 5)

scala>
```

### 1.1.2、函数的各种写法

```
scala> val array = Array(1,2,3,4)
array: Array[Int] = Array(1, 2, 3, 4)

scala> array.map( (x:Int) => x + 2)
res1: Array[Int] = Array(3, 4, 5, 6)

scala> array.map( (x) => x + 2)
res2: Array[Int] = Array(3, 4, 5, 6)

scala> array.map( x => x + 2)
res3: Array[Int] = Array(3, 4, 5, 6)

scala> array.map( _ + 2)
res4: Array[Int] = Array(3, 4, 5, 6)

scala>
```

## 1.2、高阶函数

高阶函数主要有两种：

- 1、一种是将一个函数当做另外一个函数的参数（即函数的参数是函数）
- 2、另外一种返回值是函数的函数（即函数的返回值是函数）

### 1.2.1、函数返回值为函数

作为返回值：一个方法的返回结果值是一个函数

```
scala> def add(x:Int) = {(y:Int) => x + y}
add: (x: Int)Int => Int

scala> val add2 = add(2)
add2: Int => Int = <function1>

scala> add2(4)
res0: Int = 6

scala>
```

### 1.2.2、函数/方法的参数为函数

作为参数：一个方法的参数不是一个值，而是一个函数，就是一个计算逻辑

```
scala> def opt(f:(Int, Int) => Int) = f(3,4)
opt: (f: (Int, Int) => Int)Int

scala> val add = (x:Int, y:Int) => x + y
add: (Int, Int) => Int = <function2>

scala> val max = (x:Int, y:Int) => if (x > y) x else y
max: (Int, Int) => Int = <function2>

scala> opt(add)
res1: Int = 7

scala> opt(max)
res2: Int = 4

scala>
```

简单总结：

函数是 Scala 中的头等公民：

- 1、可以作为方法的返回值
- 2、可以作为方法的参数
- 3、方法也可以被转换为函数，特定场景下自动转换或者通过\_手动转换

综合代码：

```
/**
 * 作者： 马中华: http://blog.csdn.net/zhongqi2513
 */
object Demo009_MethodAndFunction {
    // 定义一个方法
    // 方法m2 参数要求是一个函数，函数的参数必须是两个 Int 类型
```

```
//返回值类型也是 Int 类型
def m1(f: (Int, Int) => Int) : Int = f(2, 6)
// 定义一个需要两个 Int 类型参数的方法
def m2(x:Int, y:Int):Int = x + y
// 定义一个计算数据不被写死的方法
def m3(f: (Int, Int) => Int, x:Int, y:Int) : Int = f(x, y)

// 定义一个函数 f1, 参数是两个 Int 类型, 返回值是一个 Int 类型
val f1 = (x: Int, y: Int) => x + y
// 再定义一个函数 f2
val f2 = (m: Int, n: Int) => m * n
// 定义一个传入函数的函数
val f3 = (f: (Int, Int) => Int, x:Int, y:Int) => f(x, y)

//main 方法
def main(args: Array[String]) {

    //调用 m1 方法, 并传入 f1 函数
    val r1 = m1(f1)
    println(r1)

    //调用 m1 方法, 并传入 f2 函数
    val r2 = m1(f2)
    println(r2)

    // 调用 m3 方法, 传入 f1 函数
    val result1 = m3(f1, 2, 4)
    println(result1)

    // 调用 m3 方法, 传入 f2 函数
    val result2 = m3(f2, 2, 4)
    println(result2)

    // 调用 m3 方法, 传入 m2 方法作为参数
    println(m3(m2, 2, 4))

    // 调用 f3 函数, 传入 f1 函数
    println(f3(f1, 3, 4))
}
}
```

## 1.3、闭包

闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量。

闭包通常来讲可以简单的认为是可以访问一个函数里面局部变量的另外一个函数。

完整实例：

```
scala> var more = 9
more: Int = 9

scala> val add_more = (x:Int) => x + more
add_more: Int => Int = <function1>

scala> add_more(1)
res0: Int = 10

scala> more = 10
more: Int = 10

scala> add_more(1)
res1: Int = 11
```

在 `add_more` 中有两个变量：`x` 和 `more`。其中的一个 `x` 是函数的形式参数，在 `add_more` 函数被调用时，`x` 被赋予一个新的值。然而，`more` 不是形式参数，而是自由变量。这里我们引入一个自由变量 `more`，这个变量定义在函数外面。

这样定义的函数变量 `add_more` 成为一个“闭包”，因为它引用到函数外面定义的变量，定义这个函数的过程是将这个自由变量捕获而构成一个封闭的函数。

像这种运行时确定 `more` 类型及值的函数称为闭包，`more` 是个自由变量，在运行时其值和类型得以确定这是一个由开放(free)到封闭的过程，因此称为闭包

有趣的 Scala 闭包的完整测试：

```
1 package com.aura.mazh.day3.gao
2
3 /**
4  * 作者: 马中华 https://blog.csdn.net/zhongqi2513
5  * 时间: 2018/7/16 13:11
6  *
7  * 描述: Scala编程语言的闭包测试
8  */
9 object BiBaoTest {
10
11   def main(args: Array[String]): Unit = {
12
13     val result = bibao
14     result(2)
15     result(2)
16     result(2)
17     result(2)
18   }
19
20   val bibao = {
21     var sum = 0
22     val add_sum = (x:Int) => {
23       sum += x
24       println(sum)
25     }
26     add_sum
27   }
28 }
29
```

bibao是一个函数  
但是这个函数的返回值是add\_sum函数  
其实result==add\_sum  
其实在调用add\_sum的时候，sum变量的值被一直保存下来  
因为add\_sum和sum一样，都被保存在bibao这个函数的作用域中

2  
4  
6  
8

Process finished with exit code 0

在上面这种代码情况下，bibao 这个函数中的 sum 变量，只有 bibao 内部的 add\_sum 才能访问。但是 bibao 这个函数的返回值却是 add\_sum 这个函数，这个函数使用了 sum 这个变量，所以使得这个变量在 main 方法中调用 result 函数的时候，其实做到了变量到的访问和修改了 bibao 函数中定义的变量，所以本质上，闭包就是将函数内部和函数外部的连接起来的桥梁。但是千万注意，这个 sum 变量会一直驻留内存。

参考资料: [http://www.ruanyifeng.com/blog/2009/08/learning\\_javascript\\_closures.html](http://www.ruanyifeng.com/blog/2009/08/learning_javascript_closures.html)

维基百科:

[https://zh.wikipedia.org/wiki/%E9%97%AD%E5%8C%85\\_\(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6\)](https://zh.wikipedia.org/wiki/%E9%97%AD%E5%8C%85_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6))

## 1.4、Scala 柯里化 Curry

在计算机科学中，**柯里化（Currying）**是把接受多个参数的函数变换成接受一个单一参数(最初函数的第一个参数)的函数，并且返回接受余下的参数且返回结果是一个新函数的技术。有时需要允许他人一会在你的函数上应用一些参数，然后又应用另外的一些参数。例如一个乘法函数，在一个场景需要选择乘数，而另一个场景需要选择被乘数。所以柯里化函数就是将多个参数分开写，写在不同的小括号里，而不是在一个小括号中用逗号隔开

示例代码:

```
package com.mazh.scala.day3

/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
object CurryTest {

    def multiply(x:Int)(y:Int) = x * y
    //柯里化就是把参数可以分开来，把部分函数参数可以用下划线来代替
    def multiply2 = multiply(2) _

    // 一个普通的方法，接受两个 Int 类型参数做乘积
    def multiply3(x:Int, y:Int) = x * y
    def multiply4(x:Int, y:Int=10) = x * y
    def multiply5(x:Int)(y:Int=10) = x * y

    def main(args: Array[String]):Unit = {
        println(multiply(2)(4))
        println(multiply2(4))

        // 跟柯里化的函数在结果上没有区别，那到底有什么区别呢？
        println(multiply3(2, 4))

        println(multiply4(4))
    }
}
```

```
println(multiply5(4)())  
}  
}
```

那柯里化到底有什么作用？

降低通用性，提高适用性

## 2、Scala 隐式转换和隐式参数

隐式转换和隐式参数是 Scala 中两个非常强大的功能，利用隐式转换和隐式参数，你可以提供优雅类库，对类库的使用者隐匿掉那些枯燥乏味的细节。

隐式的对类的方法进行增强，丰富现有类库的功能

是指那种以 `implicit` 关键字声明的带有单个参数的函数。

可以通过：`:implicit -v` 这个命令显示所有做隐式转换的类。

### 2.1、Scala 隐式转换探讨

现在我们来考虑一个问题：

之前讲过：

**1 to 10 其实可以写成 1.to(10)**

那其实就是表示：1 是一个 Int 类型的变量，所以证明 Int 类中会有一个 to 的方法

但事实上，我们在 Int 类型中根本就没有寻找 to 方法

那也就是说对一个 Int 类型的变量 1 调用 Int 类型不存在的一个方法，这怎么还能正常运行呢？

原因就是 **隐式转换**

看一个最简单的隐式转换的例子：

```
scala> val aa = 3.5  
aa: Double = 3.5  
  
scala> val aa:Int = 3.5  
<console>:11: error: type mismatch;  
found   : Double(3.5)  
required: Int  
val aa:Int = 3.5  
           ^  
  
scala> implicit def double2Int(x:Double)=x.toInt  
warning: there was one feature warning; re-run with -feature for details  
double2Int: (x: Double)Int  
  
scala> val x:Int=3.5  
x: Int = 3  
  
scala>
```

那我们首先来看一下 **隐式参数**：

```
package com.mazh.scala.day3

object ImplicitParamTest {
  // 正常的普通方法
  def add(x:Int, y:Int) = x + y

  // 柯里化的方法
  def add2(x:Int)(y:Int) = x + y
  def add3(x:Int)(y:Int = 10) = x + y

  // 如果变成下面这种形式:
  def add4(x:Int)(implicit y:Int = 10) = x + y

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

    println(add(2,3))
    // 不能只传一个参数取使用, 必须要传入两个参数
    println(add2(2)(3))
    println(add3(2)())
    // 调用带隐式参数的函数
    println(add4(2))
  }
}
```

在上面的代码中, 可以看出来, 如果对 `add2` 方法的第二个参数, 做了隐式声明, 发现之前需要传入两个参数才能执行的方法 `add2` 就可以只传入一个参数就能执行计算

那有什么应用场景呢?

比如汇率计算!!!!

```
package com.mazh.scala.day3

/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
object ImplicitParamTest2 {
  /**
   * 第一个参数是要换算成美元的人民币数目
   * 第二个参数是汇率
   */
  def rmb(dollar:Double)(implicit rate:Double = 6) = dollar * rate

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

    println(rmb(100))
    println(rmb(100)(7))
  }
}
```



```
// 引入隐式转换值，所以第二个参数被隐式的转换成了 6.66
import MyPredef._
println(rmb(100))
}

object MyPredef{
  // 声明一个 Double 类型的隐式转换值
  implicit var current_rate:Double = 6.66
}
```

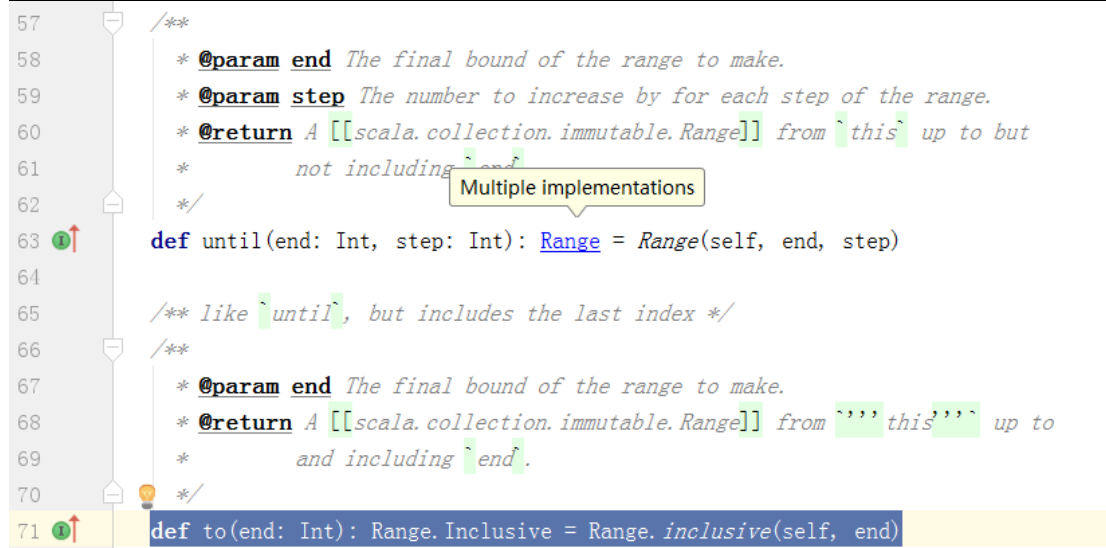
总结：

- 1、隐式转换会首先从全局中寻找，寻找不到，才使用隐式参数
- 2、隐式转换只能定义在 **object** 中
- 3、如果隐式转换存在二义性，那么程序会跑错

那现在再来考虑：

对一个 `Int` 类型的变量 `1` 调用 `Int` 类型不存在的一个方法，程序能正常运行得到期待的结果，没有抛错异常，到底是怎么回事？会不会就是 `Int` 类型的变量被隐式转换成了另一种包含 `to` 方法的类型了呢？

1、首先，我们在 `RichInt` 中发现了 `to` 方法：



```
57  /**
58   * @param end The final bound of the range to make.
59   * @param step The number to increase by for each step of the range.
60   * @return A [[scala.collection.immutable.Range]] from `this` up to but
61   *         not including `end`.
62   */
63  def until(end: Int, step: Int): Range = Range(self, end, step)
64
65  /** like `until`, but includes the last index */
66  /**
67   * @param end The final bound of the range to make.
68   * @return A [[scala.collection.immutable.Range]] from `this` up to
69   *         and including `end`.
70   */
71  def to(end: Int): Range.Inclusive = Range.inclusive(self, end)
```

2、查看系统是否为我们自动引入了默认的各种隐式转换：

在 Scala 交互命令行中执行命令：**`:implicits -v`**

```
scala> :implicit -v
/* 69 implicit members imported from scala.Predef */
/* 7 inherited from scala */
final implicit class ArrayCharSequence extends CharSequence
```

```
final implicit class ArrowAssoc[A] extends AnyVal
final implicit class Ensuring[A] extends AnyVal
final implicit class RichException extends AnyVal
final implicit class SeqCharSequence extends CharSequence
final implicit class StringFormat[A] extends AnyVal
final implicit class any2stringadd[A] extends AnyVal

/* 40 inherited from scala.Predef */
implicit def ArrowAssoc[A](self: A): ArrowAssoc[A]
implicit def Ensuring[A](self: A): Ensuring[A]
implicit def StringFormat[A](self: A): StringFormat[A]
implicit def any2stringadd[A](self: A): any2stringadd[A]

implicit def booleanArrayOps(xs: Array[Boolean]): mutable.ArrayOps[Boolean]
implicit def byteArrayOps(xs: Array[Byte]): mutable.ArrayOps[Byte]
implicit def charArrayOps(xs: Array[Char]): mutable.ArrayOps[Char]
implicit def doubleArrayOps(xs: Array[Double]): mutable.ArrayOps[Double]
implicit def floatArrayOps(xs: Array[Float]): mutable.ArrayOps[Float]
implicit def genericArrayOps[T](xs: Array[T]): mutable.ArrayOps[T]
implicit def intArrayOps(xs: Array[Int]): mutable.ArrayOps[Int]
implicit def longArrayOps(xs: Array[Long]): mutable.ArrayOps[Long]
implicit def refArrayOps[T <: AnyRef](xs: Array[T]): mutable.ArrayOps[T]
implicit def shortArrayOps(xs: Array[Short]): mutable.ArrayOps[Short]
implicit def unitArrayOps(xs: Array[Unit]): mutable.ArrayOps[Unit]

implicit def $conforms[A]: <:<[A,A]
implicit def ArrayCharSequence(__arrayOfChars: Array[Char]): ArrayCharSequence
implicit def Boolean2boolean(x: Boolean): Boolean
implicit def Byte2byte(x: Byte): Byte
implicit def Character2char(x: Character): Char
implicit def Double2double(x: Double): Double
implicit def Float2float(x: Float): Float
implicit def Integer2int(x: Integer): Int
implicit def Long2long(x: Long): Long
implicit def RichException(self: Throwable): RichException
implicit def SeqCharSequence(__sequenceOfChars: IndexedSeq[Char]): SeqCharSequence
implicit def Short2short(x: Short): Short
implicit val StringCanBuildFrom: generic.CanBuildFrom[String,Char,String]
implicit def augmentString(x: String): immutable.StringOps
implicit def boolean2Boolean(x: Boolean): Boolean
implicit def byte2Byte(x: Byte): Byte
implicit def char2Character(x: Char): Character
implicit def double2Double(x: Double): Double
implicit def float2Float(x: Float): Float
```

```
implicit def int2Integer(x: Int): Integer
implicit def long2Long(x: Long): Long
implicit def short2Short(x: Short): Short
implicit def tuple2ToZippedOps[T1, T2](x: (T1, T2)): runtime.Tuple2Zipped.Ops[T1,T2]
implicit def tuple3ToZippedOps[T1, T2, T3](x: (T1, T2, T3)):
runtime.Tuple3Zipped.Ops[T1,T2,T3]
implicit def unaugmentString(x: immutable.StringOps): String

/* 22 inherited from scala.LowPriorityImplicits */
implicit def genericWrapArray[T](xs: Array[T]): mutable.WrappedArray[T]
implicit def wrapBooleanArray(xs: Array[Boolean]): mutable.WrappedArray[Boolean]
implicit def wrapByteArray(xs: Array[Byte]): mutable.WrappedArray[Byte]
implicit def wrapCharArray(xs: Array[Char]): mutable.WrappedArray[Char]
implicit def wrapDoubleArray(xs: Array[Double]): mutable.WrappedArray[Double]
implicit def wrapFloatArray(xs: Array[Float]): mutable.WrappedArray[Float]
implicit def wrapIntArray(xs: Array[Int]): mutable.WrappedArray[Int]
implicit def wrapLongArray(xs: Array[Long]): mutable.WrappedArray[Long]
implicit def wrapRefArray[T <: AnyRef](xs: Array[T]): mutable.WrappedArray[T]
implicit def wrapShortArray(xs: Array[Short]): mutable.WrappedArray[Short]
implicit def wrapUnitArray(xs: Array[Unit]): mutable.WrappedArray[Unit]

implicit def booleanWrapper(x: Boolean): runtime.RichBoolean
implicit def byteWrapper(x: Byte): runtime.RichByte
implicit def charWrapper(c: Char): runtime.RichChar
implicit def doubleWrapper(x: Double): runtime.RichDouble
implicit def floatWrapper(x: Float): runtime.RichFloat
implicit def intWrapper(x: Int): runtime.RichInt
implicit def longWrapper(x: Long): runtime.RichLong
implicit def shortWrapper(x: Short): runtime.RichShort
implicit def unwrapString(ws: immutable.WrappedString): String
implicit def wrapString(s: String): immutable.WrappedString
```

通过观察发现，scala 会默认给我们引入 scala 中的 `Predef.scala` 中的所有隐式转换  
[https://www.scala-lang.org/api/2.11.8/#scala.Predef\\$](https://www.scala-lang.org/api/2.11.8/#scala.Predef$)

最后在倒数第五行发现，有一个隐式方法能够把 `Int` 类型的变量转换成 `runtime.RichInt` 变量符合我们的预期

最终解释：

当调用了：**1 to 10**

其实是调用了：**1.to(10)**

但是：`Int` 中没有 `to` 方法

所以：去寻找引入的隐式转换中有没有能把 `Int` 类型转换成能执行 `to` 方法的类型

果然：在系统引入的转换中发现：**`implicit def intWrapper(x: Int): runtime.RichInt`**

所以：最终 `int` 类型的 `1` 就被转换成了 `RichInt` 类型的变量

验证: RichInt 中确实存在 to 方法

最后: 顺理成章的调用 RichInt(1).to(10)生成返回结果

结论: 神奇但又合理

## 2.2、隐式转换的发生时机

到底在什么时候触发隐式转换呢?

### 2.2.1、时机一：当调用某个对象不存在的方法时

当一个对象去调用某个方法，但是这个对象并不具备这个方法。这个时候会触发隐式转换，会把这个对象（偷偷的）隐式转换为具有这个方法的那个对象。这就和刚才解释的为什么 Int 类型没有 to 方法还是能够调用 to 方法，因为 Int 类型的变量 1 在调用 to 方法的时候，被隐式转换成了 RichInt 的对象

再次演示一个案例：

```
package com.mazh.scala.day3

import java.io.File
import scala.io.Source

/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
object ImplicitTest2 {
  def main(args: Array[String]): Unit = {
    import FileImplicit._
    val file = new File("c:\\words.txt")
    // file 对象是没有 readAll 方法的, 那么在调用一个不存在的方法的时候
    // scala 会查看是否有隐式转换能把 file 对象转换成具有 readAll 方法对象
    val allText = file.readAll()
    println(allText)
  }
}

class RichFile(f:File) {
  def readAll():String = {
    Source.fromFile(f).mkString
  }
}

object FileImplicit{
```

```
implicit def file2RichFile(f:File):RichFile = new RichFile(f)
}
```

## 2.2.2、时机二：当方法参数类型不匹配时

正常情况下，我们在编写代码，如果去调用某个方法，确实这个方法也存在，要是传入的参数类型不匹配，程序会抛错，这是再正常不过的事情了。

可是 Scala 却在为我们做努力，努力帮助我们把这个方法调用执行起来。因为隐式转换的存在

第一个案例：

```
package com.aura.mazh.day3.gao

/**
 * 作者： 马中华   https://blog.csdn.net/zhongqi2513
 * 时间： 2017/7/10 13:12
 * 描述： 隐式转换小李子
 */
object Demo009_Implicit_Type {

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

        // 定义一个隐式转换，能够、把一个 double 类型的数编程 int 类型。
        implicit def double2Int(a:Double) = a.toInt

        // 定义三个方法
        def sum1(x:Int, y:Int) = x + y
        def sum2(x:Int, y:Double) = x + y
        def sum3(x:Double, y:Double) = x + y

        // 使用
        println(sum1(1, 2.0))    // 触发隐式转换
        println(sum2(1, 2))     // 触发隐式转换
        println(sum3(1,2))      // 触发隐式转换
    }
}
```

第二个案例：

```
package com.mazh.scala.day3

//特殊人群
class SpecialPerson(var name:String)
```

```
//特殊人群之一
class Young(name:String)
//特殊人群之二
class Older(name:String)
//正常人群之一
class Worker(var name:String)
//正常人群之二
class Adult(var name:String)

class TicketHouse{
    def buyTicket(p:SpecialPerson): Unit ={
        println(p.name+"票给你!! 爽去吧!!");
    }
}

object ObjectImplicit{
    implicit def object2special(obj:AnyRef):SpecialPerson={
        // 这么写的原因是给大家演示这三个方法的使用。其实有更简单的实现
        if(obj.getClass == classOf[Young]){
            val young = obj.asInstanceOf[Young]
            new SpecialPerson(young.name)
        }else if(obj.getClass == classOf[Older]){
            val older = obj.asInstanceOf[Older]
            new SpecialPerson(older.name)
        }else{
            new SpecialPerson("NULL")
        }
    }
}

object ImplicitTest3 {
    def main(args: Array[String]): Unit = {
        val ticketHouse = new TicketHouse()
        val young = new Young("Young")
        val older = new Older("Older")
        val worker = new Worker("Worker")
        val adult = new Worker("Adult")

        import ObjectImplicit._
        // ticketHouse.buyTicket(worker)    // 报错
        // ticketHouse.buyTicket(adult)      // 报错
        ticketHouse.buyTicket(young)
        ticketHouse.buyTicket(older)
    }
}
```

## 2.2.3、时机三：在视图边界的时候

内容见 **3.3、视图界定**

## 2.3、隐式转换忠告

下面给出我自己开发实践中的部分总结，供大家参考：

- 1、即使你能轻松驾驭 Scala 语言中的隐式转换，能不用隐式转换就尽量不用
- 2、如果一定要用，在涉及多次隐式转换时，必须要说服自己这样做的合理性
- 3、如果只是炫耀自己的 Scala 编程能力，请大胆使用

# 3、Scala 泛型

## 3.1、Scala 泛型基础

泛型用于指定方法或类可以接受任意类型参数，参数在实际使用时才被确定，泛型可以有效地增强程序的适用性，使用泛型可以使得类或方法具有更强的通用性。泛型的典型应用场景是集合及集合中的方法参数，可以说同 Java 一样，Scala 中泛型无处不在，具体可查看 Scala 的 API

**泛型类：指定类可以接受任意类型参数。**

**泛型方法：指定方法可以接受任意类型参数。**

```
package com.mazh.scala.day3

class Person[T](var name:T)
class Student[T,S](name:T,var age:S) extends Person(name)

/**
 * 作者： 马中华: http://blog.csdn.net/zhongqi2513
 */
object GenericTypeTest {
  def main(args: Array[String]): Unit = {
    println(new Student[String, Int]("黄渤", 33).name)
  }
}
```

## 3.2、Scala 类型变量界定

类型变量界定是指在泛型的基础上，对泛型的范围进行进一步的界定，从而缩小泛型的具体

范围

比如下面的代码编译不通过:

```
package com.mazh.scala.day3

/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
class GenericTypeTest2 {
  def compare[T](first:T, second:T) = {
    if (first.compareTo(second)>0)
      first
    else
      second
  }
}

object GenericTypeTest2{
  def main(args: Array[String]): Unit = {
    val tvb = new GenericTypeTest2
    println(tvb.compare("A", "B"))
  }
}
```

代码为什么编译不通过, 是因为: **泛型 T 并不一定具备 compareTo 方法**

如果想编译通过, 请做如下更改:

```
package com.mazh.scala.day3

/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
class GenericTypeTest2 {
  def compare[T <: Comparable[T]](first:T, second:T)={
    if (first.compareTo(second)>0)
      first
    else
      second
  }
}

object GenericTypeTest2{
  def main(args: Array[String]): Unit = {
    val tvb=new GenericTypeTest2
    println(tvb.compare("A", "B"))
  }
}
```



```
}  
}
```

代码中改动的地方：

**T <: Comparable[T]**

这是什么意思呢？

**compareTo** 方法中如果输入的类型处于 **Comparable** 类对应继承层次结构中，则是合法的，  
否则的话编译会报错

### 3.3、Scala 视图界定

上面讲的类型变量界定建立在类继承层次结构的基础上，但有时候这种限定不能满足实际要求，如果**希望跨越类继承层次结构时，可以使用视图界定来实现的**，其后面的**原理是通过隐式转换**来实现。

隐含参数和方法也可以定义隐式转换，称作视图。视图的绑定从另一个角度看就是 **implicit** 的转换。主要用在两个场合：

- 1、当一个 **T** 类型的变量 **t** 要转换成 **A** 类型时
- 2、当一个类型 **T** 的变量 **t** 无法拥有 **A** 类型的 **a** 方法或变量时

其实视图的绑定是为了更方便的使用隐式转换

视图界定利用 **<%** 符号来实现

如下：代码的执行会抛出异常

```
package com.mazh.scala.day3  
  
case class Student11[T,S <: Comparable[S]](var name:T, var height:S)  
  
/**  
 * 作者： 马中华: http://blog.csdn.net/zhongqi2513  
 */  
object GenericTypeTest3{  
  
    def main(args: Array[String]): Unit = {  
        // 这是合法的语句  
        val s= Student11("john","170")  
        //下面这条语句不合法，这是因为, Int 类型没有实现 Comparable 接口  
        val s2= Student11("john", 170)  
    }  
}
```

如果想通过，那么要使用视图界定的技术

更改之后的代码：

```
package com.mazh.scala.day3  
  
case class Student11[T,S <% Comparable[S]](var name:T, var height:S)
```

```
/**
 * 作者： 马中华：http://blog.csdn.net/zhongqi2513
 */
object GenericTypeTest3{

    def main(args: Array[String]): Unit = {
        // 这是合法的语句
        val s= Student11("john","170")
        // Int 类型的变量经过隐式转换成了 RichInt 类型，RichInt 类型是实现了 Comparable 接口的
        val s2= Student11("john",170)
    }
}
```

改动的代码：

```
case class Student11[T, S <= Comparable[S]](var name:T, var height:S)
```

能运行的原因：

利用<=符号对泛型 S 进行限定，它的意思是 S 可以是 Comparable 类继承层次结构中实现了 Comparable 接口的类，也可以是能够经过隐式转换得到的实现了 Comparable 接口的类。

上面改动的语句在视图界定中是合法的，因为 Int 类型此时会隐式转换为 RichInt 类，而 RichInt 类属于 Comparable 继承层次结构。Int 类会隐式转换成 RichInt 类，RichInt 并不是直接实现 Comparable 口，而是通过 ScalaNumberProxy 类将 Comparable 中的方法继承过来。

## 3.4、Scala 上界下界

上界、下界介绍

在指定泛型类型时，有时需要界定泛型类型的范围，而不是接收任意类型。比如，要求某个泛型类型，必须是某个类的子类，这样在程序中就可以放心的调用父类的方法，程序才能正常的使用与运行。此时，就可以使用上下边界 Bounds 的特性；

Scala 的上下边界特性允许泛型类型是某个类的子类，或者是某个类的父类；

### 1、U >: T

这是类型下界的定义，也就是 U 必须是类型 T 的父类(或本身，自己也可以认为是自己的父类)。

### 2、S <: T

这是类型上界的定义，也就是 S 必须是类型 T 的子类（或本身，自己也可以认为是自己的子类）。

### 3.4.1、上界

在讲解类型变量绑定的内容中，咱们写过这么一段代码：

```
class GenericTypeTest2 {  
    def compare[T <: Comparable[T]](first:T,second:T)={  
        if (first.compareTo(second)>0)  
            first  
        else  
            second  
    }  
}
```

标红的地方,其实就是**上界**,因为它限定了继承层次结构中最顶层的类,例如 `T <: Comparable[T]` 表示泛型 `T` 的类型的最顶层类是 `Comparable`,所有输入是 `Comparable` 的子类都是合法的,其它的都是非法的,因为被称为**上界**

### 3.4.2、下界

当然,除了上界之外,还有个非常重要的内容就是**下界**,下界通过 `>` 符号来标识,比如:

```
class A  
class B extends A  
class C extends B  
class D extends C
```

如果代码中这么写: `opt[T >: C]`

那么表示 `T` 的类型只能是 `A,B,C` 了。不能是 `D`,其实就是限制了最底层的类型是什么。在类的继承结构体系中,从上到下,只能到类型 `C` 为止

**下界的作用主要是保证类型安全**

实例代码:

```
package com.mazh.scala.day3.gao  
  
/**  
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513  
 */  
object GenericTypeTest4 {  
  
    def getIDCard[R >: Son](person:R): Unit = {  
        println("好吧,他的身份证就交给你保管了");  
    }  
  
    def main(args: Array[String]): Unit = {  
  
        // getIDCard[T](t:T) 前面这个 T 表示方法中的参数类型被固定下来。  
        // 在定义的时候还不知道这个 T 类型到底应该是什么,但是调用的时候,被确定下来是某种类型  
        // 但是,参数中的类型,无论如何都可以是 T 的子类型,这属于多态范畴  
        getIDCard[GranderFather](new GranderFather)
```

```
getIDCard[GranderFather](new Father)
getIDCard[Father](new Father)
getIDCard[Son](new Son)

// 这句代码会报错
getIDCard[Tongzhuo](new Tongzhuo)
}
}

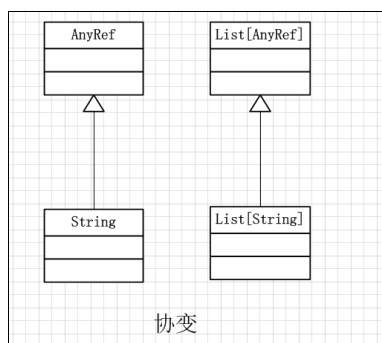
class GranderFather
class Father extends GranderFather
class Son extends Father
class Tongzhuo
```

## 3.5、Scala 逆变和协变

### 3.5.1、协变

协变定义形式如：trait List[+T]{}

当类型 **B** 是类型 **A** 的子类型时，则 **List[B]**也可以认为是 **List[A]**的子类型，即 **List[B]**可以泛化为 **List[A]**。也就是被参数化类型的泛化方向与参数类型的方向是一致的，所以称为**协变**（covariance）



首先，Java 中不存在协变：

```
package com.ghgj.sp.util;

import java.util.LinkedList;

/**
 * 作者： 马中华： http://blog.csdn.net/zhongqi2513
 * 日期： 2018年4月22日 上午8:30:47
 *
 * 描述： 测试Java是否有协变
 */
```

```
*/  
public class TypeTest {  
  
    public static void main(String[] args) {  
  
        java.util.List<String> s1 = new LinkedList<String>();  
        java.util.List<Object> s2 = new LinkedList<Object>();  
        /**  
         * 下面这条语句会报错  
         * Type mismatch: cannot convert from List<String> to List<Object>  
         */  
        s2 = s1;  
    }  
}
```

然在类层次结构上看，String 是 Object 类的子类，但 List<String>并不是的 List<Object>子类，也就是说它不是协变的。Java 的灵活性就这么差吗？其实 Java 不提供协变和逆变这种特性是有其道理的，这是**因为协变和逆变会破坏类型安全**。假设上面的代码是合法的，我们此时完全可以 s2.add(new Person("xuzheng"))往集合中添加 Person 对象，但此时我们知道，s2 已经指向了 s1，而 s1 里面的元素类型是 String 类型，这时其类型安全就被破坏了，从这个角度来看，Java 不提供协变和逆变是有其合理性的。

### 那为什么 Java 不支持，而 Scala 支持呢？

先来看一段 Scala 代码：Scala 比 Java 更灵活，当不指定逆变和协变时，和 Java 是一样的。代码如下：

```
package com.mazh.scala.day3.gao  
  
/**  
 * 作者： 马中华: http://blog.csdn.net/zhongqi2513  
 */  
object XieBianTest {  
    def main(args: Array[String]): Unit = {  
        val list1:MyList[String]= new MyList[String]("黄",null)  
        val list2:MyList[String]= new MyList[String]("黄",new MyList[String]("黄",null))  
    }  
}  
  
class MyList[T](val head: T, val tail: MyList[T]){}  

```

看效果：

```
/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
object XieBianTest {

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

    val list1: MyList[String] = new MyList[String]("黄", null)
    val list2: MyList[String] = new MyList[String]("黄", new MyList[String]("黄", null))
  }

}

class MyList[T](val head: T, val tail: MyList[T]) {}
```

但是,如果把代码稍微改变一下,发现编译就会报错。因为 `MyList[String]` 不是 `MyList[Any]` 的子类。

```
/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
object XieBianTest {

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

    val list1: MyList[Any] = new MyList[String]("黄", null)
    val list2: MyList[Any] = new MyList[String]("黄", new MyList[String]("黄", null))
  }

}

class MyList[T](val head: T, val tail: MyList[T]) {}
```

可以看到,当不指定类为协变的时候,而是一个普通的 Scala 类,此时它跟 Java 一样是具有类型安全的,称这种类是**非变/不变的(Nonvariance)**。Scala 的灵活性在于它提供了协变与逆变语言特点供你选择。上述的代码要使其合法,可以定义 List 类是协变的,泛型参数前面用 `+` 符号表示,此时 List 就是协变的,即如果 T 是 S 的子类型,那 `List[T]` 也是 `List[S]` 的子类型

看如下指定协变的代码: 当把 T 指定为协变时,代码编译正常

```
/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
object XieBianTest {

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

    val list1: MyList[Any] = new MyList[String]("黄", null)
    val list2: MyList[Any] = new MyList[String]("黄", new MyList[String]("黄", null))
  }

}

class MyList[+T](val head: T, val tail: MyList[T]) {}
```

指定协变

但是,如果想要往 `MyList` 添加方法时,又会报错:

```
/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
object XieBianTest {

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

    val list1: MyList[Any] = new MyList[String]("黄", null)
    val list2: MyList[Any] = new MyList[String]("黄", new MyList[String]("黄", null))
  }

}

class MyList[+T](val head: T, val tail: MyList[T]){

  def prepend(newHead: T): MyList[T] = new MyList(newHead, this)
}
```

记住：如果定义其他成员方法的时候，必须也要将成员方法声明为泛型

```
package com.aura.mazh.day3.gao

/**
 * 作者: 马中华: http://blog.csdn.net/zhongqi2513
 */
object XieBianTest {

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

    val list1: MyList[Any] = new MyList[String]("黄", null)
    val list2: MyList[Any] = new MyList[String]("黄", new MyList[String]("黄", null))
  }

}

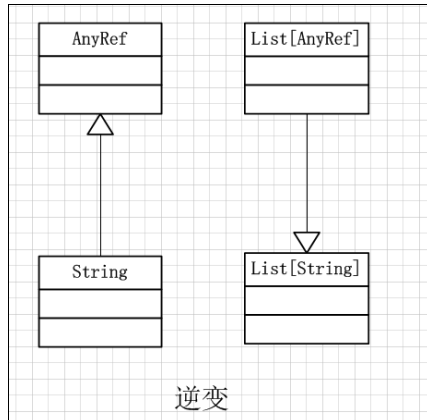
class MyList[+T](val head: T, val tail: MyList[T]){

  /**
   * 将函数也用泛型表示。因为是协变的，输入的类型必须是T的超类
   * 这样返回值类型 MyList[U] 就是 MyList[T] 的超类。符合协变的性质
   */
  def prepend[U >: T](newHead: U): MyList[U] = new MyList(newHead, this)
}
```

### 3.5.2、逆变

逆变定义形式如：trait List[-T]{}

当类型 B 是类型 A 的子类型，则 Queue[A] 反过来可以认为是 Queue[B] 的子类型。也就是被参数化类型的泛化方向与参数类型的方向是相反的，所以称为**逆变 (contravariance)**



```
// 声明逆变
class Person2[-A]{ def test(x:A){} }

// 声明协变，但会报错
class Person3[+A]{ def test(x:A){} }
```

要理解清楚后面的原理，先要理解清楚什么是  
**协变点(covariant position)** 和 **逆变点(contravariant position)**

**协变点：方法返回值的位置点**

```
class Person5[-A]{
  def test:A=null.asInstanceOf[A]
}
```

**逆变点：方法参数的位置点**

```
class Person3[+A]{ def test(x:A){} }
```

我们先假设 class Person3[+A]{ def test(x:A){} }能够编译通过，则对于 Person3[Any] 和 Person3[String] 这两个父子类型来说，它们的 test 方法分别具有下列形式：

```
//Person3[Any]
def test(x:Any){}

//Person3[String]
def test(x:String){}
```

由于 AnyRef 是 String 类型的父类，由于 Person3 中的类型参数 A 是协变的，也即 Person3[Any] 是 Person3[String]的父类，因此如果定义了

```
val pAny = new Person3[AnyRef]
```

```
val pString = new Person3[String]
```

调用 pAny.test(123)是合法的，但如果将 pAny = pString 进行重新赋值（这是合法的，因为父类可以指向子类，也称里氏替换原则），此时再调用 pAny.test(123)时候，这是非法的，因为子类类型不接受非 String 类型的参数。也就是父类能做的事情，子类不一定能做，子类只是部分满足。



为满足里氏替换原则，**子类中函数参数都必须是父类中函数参数的超类**，这样的话父类能做的子类也能做。因此需要将类中的泛型参数声明为逆变或不变的。

```
class Person2[-A]{ def test(x:A){} }
```

我们可以对 Person2 进行分析，同样声明两个变量：

```
val pAnyRef = new Person2[AnyRef]
```

```
val pString = new Person2[String]
```

**由于是逆变的，所以 Person2[String] 是 Person2[AnyRef] 的超类**，pAnyRef 可以赋值给 pString，从而 pString 可以调用范围更广泛的函数参数（比如未赋值之前，pString.test("123") 函数参数只能为 String 类型，则 pAnyRef 赋值给 pString 之后，它可以调用 test(x:AnyRef) 函数，使函数接受更广泛的参数类型。**方法参数的位置称为做逆变点**(contravariant position)，这是 class Person3[+A]{ def test(x:A){} } 会报错的原因。

为使 class Person3[+A]{ def test(x:A){} } 合法

可以利用下界进行泛型限定，如：

```
class Person3[+A]{ def test[R>:A](x:R){} }
```

将参数范围扩大，从而能够接受更广泛的参数类型。

通过前述的描述，我们弄明白了什么是逆变点，现在我们来看一下什么是协变点，先看下面的代码：

```
//下面这行代码能够正确运行
class Person4[+A]{
  def test:A=null.asInstanceOf[A]
}
//下面这行代码会编译出错
//contravariant type A occurs
//in covariant position in type => A of method test
class Person5[-A]{
  def test:A=null.asInstanceOf[A]
}
```

这里我们同样可以通过里氏替换原则来进行说明：

```
scala> class Person[+A]{def f():A=null.asInstanceOf[A]}
defined class Person

scala> val p1=new Person[AnyRef]()
p1: Person[AnyRef] = Person@8dbd21

scala> val p2=new Person[String]()
p2: Person[String] = Person@1bb8cae

scala> p1.f
res0: AnyRef = null
```

```
scala> p2.f  
res1: String = null
```

### 3.5.3、协变和逆变总结

来自维基百科的解释：

在一门程序设计语言的类型系统中，一个类型规则或者类型构造器是：

**协变 (covariant)**，如果它保持了子类型序关系 $\leq$ 。该序关系是：子类型 $\leq$ 基类型。

**逆变 (contravariant)**，如果它逆转了子类型序关系。

**不变 (invariant)**，如果上述两种均不适用。

1、`IEnumerable<Cat>`是 `IEnumerable<Animal>`的子类型，因为类型构造器 `IEnumerable<T>`是协变的 (covariant)。注意到复杂类型 `IEnumerable` 的子类型关系和其接口中的参数类型是一致的，亦即，参数类型之间的子类型关系被保持住了。

2、`Action<Cat>` 是 `Action<Animal>` 的超类型，因为类型构造器 `Action<T>` 是逆变的 (contravariant)。(在此，`Action<T>`被用来表示一个参数类型为 `T` 或 `sub-T` 的一级函数)。注意到 `T` 的子类型关系在复杂类型 `Action` 的封装下是反转的，但是当它被视为函数的参数时其子类型关系是被保持的。

3、`IList<Cat>`或 `IList<Animal>`彼此之间没有子类型关系。因为 `IList<T>`类型构造器是不变的 (invariant)，所以参数类型之间的子类型关系被忽略了。