

1. 高阶函数

1.1. 概念

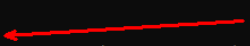
如果一个函数的传入参数为函数或者返回值是函数，则该函数即为高阶函数。

1.2. 传入参数为函数

Scala中，函数是头等公民，和数字一样。不仅可以调用，还可以在变量中存放函数，也可以作为参数传入函数，或者作为函数的返回值。

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

scala> val fun = (x: Int) => x*2
fun: Int => Int = <function1>

scala> arr.map(fun)  函数作为参数传入其他函数
res0: Array[Int] = Array(2, 4, 6)
```


1.3. 传入参数为匿名函数

在Scala中，你不需要给每一个函数命名，就像不必给每个数字命名一样，将函数赋给变量的函数叫做匿名函数

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

scala> val fun = (x: Int) => x*2
fun: Int => Int = <function1>


scala> arr.map(fun)
res0: Array[Int] = Array(2, 4, 6)

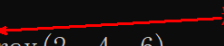
scala> arr.map((x: Int) => x*2)  直接将一个匿名函数作为参数传递给另一个函数
res1: Array[Int] = Array(2, 4, 6)
```

还可以

```
scala> arr.map(fun)
res0: Array[Int] = Array(2, 4, 6)

scala> arr.map((x: Int) => x*2)
res1: Array[Int] = Array(2, 4, 6)

scala> arr.map( x => x*2 )  精简方式1
res2: Array[Int] = Array(2, 4, 6)

scala> arr.map( _*2 )  精简方式2
res3: Array[Int] = Array(2, 4, 6)
```

1.4. 传入参数为方法（隐式转换方法到函数）

在Scala中，方法和函数是不一样的，最本质的区别是函数可以作为参数传递到方法中

```
case class WeeklyWeatherForecast(temperatures: Seq[Double]) {

  private def convertCtoF(temp: Double) = temp * 1.8 + 32
  //方法convertCtoF作为参数传入
  def forecastInFahrenheit: Seq[Double] = temperatures.map(convertCtoF)
}
```

1.5.返回值为函数

```
//返回值为函数类型: (String, String) => String
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {
  val schema = if (ssl) "https://" else "http://"
  (endpoint: String, query: String) => s"$schema$domainName/$endpoint?$query"
}

val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName)
val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query) // "https://www.example.com/users?id=1": String
```

2.方法的嵌套（备选）

方法里嵌套定义其他方法

```
def factorial(x: Int): Int = {
  def fact(x: Int, accumulator: Int): Int = {
    if (x <= 1) accumulator
    else fact(x - 1, x * accumulator)
  }
  fact(x, 1)
}

println("Factorial of 2: " + factorial(2))
println("Factorial of 3: " + factorial(3))
```

3.方法的多态（备选）

Scala里方法可以通过类型实现参数化，类似泛型。

```
def listOfDuplicates[A](x: A, length: Int): List[A] = {
  if (length < 1)
    Nil
  else
    x :: listOfDuplicates(x, length - 1)
}
println(listOfDuplicates[Int](3, 4)) // List(3, 3, 3, 3)
println(listOfDuplicates("La", 8)) // List(La, La, La, La, La, La, La, La)
```

4.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。

函数体内可以方法相应作用域内的任何变量。

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。

普通函数：

```
val multiplier = (i:Int) => i * 10
```

函数体内有一个变量 i，它作为函数的一个参数。

```
val multiplier = (i:Int) => i * factor
```

在 multiplier 中有两个变量：i 和 factor。其中的一个 i 是函数的形式参数，在 multiplier 函数被调用时，i 被赋予一个新的值。然而，factor 不是形式参数，而是自由变量，考虑下面代码：

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

这里我们引入一个自由变量 factor，这个变量定义在函数外面。

这样定义的函数变量 multiplier 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。

```
object Test {
  def main(args: Array[String]) {
    println( "multiplier(1) value = " + multiplier(1) )
    println( "multiplier(2) value = " + multiplier(2) )
  }
  var factor = 3
  val multiplier = (i:Int) => i * factor
}
```

5.柯里化

柯里化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而且返回结果的新函数的技术。

下面先给出一个普通的非柯里化的函数定义，实现一个加法函数：

```
scala> def plainOldSum(x:Int,y:Int) = x + y
plainOldSum: (x: Int, y: Int)Int
scala> plainOldSum(1,2)
res0: Int = 3
```

使用“柯里化”技术，把函数定义为多个参数列表：

```
scala> def curriedSum(x:Int)(y:Int) = x + y
curriedSum: (x: Int)(y: Int)Int
scala> curriedSum (1)(2)
res0: Int = 3
```

当你调用 `curriedSum (1)(2)` 时，实际上是依次调用两个普通函数（非柯里化函数），第一次调用使用一个参数 `x`，返回一个函数类型的值，第二次使用参数 `y` 调用这个函数类型的值，我们使用下面两个分开的定义在模拟 `curriedSum` 柯里化函数：

首先定义第一个函数：

```
scala> def first(x:Int) = (y:Int) => x + y
first: (x: Int)Int => Int
```

然后我们使用参数1调用这个函数来生成第二个函数。

```
scala> val second=first(1)
second: Int => Int = <function1>
scala> second(2)
res1: Int = 3
```

`first`，`second` 的定义演示了柯里化函数的调用过程，它们本身和 `curriedSum` 没有任何关系，但是我们可以使用 `curriedSum` 来定义 `second`，如下：

```
scala> val onePlus = curriedSum(1)_
onePlus: Int => Int = <function1>
```

下划线“`_`”作为第二参数列表的占位符，这个定义的返回值为一个函数，当调用时会给调用的参数加一。

```
scala> onePlus(2)
res2: Int = 3
```

通过柯里化，你还可以定义多个类似 `onePlus` 的函数，比如 `twoPlus`

```
scala> val twoPlus = curriedSum(2) _  
twoPlus: Int => Int = <function1>  
scala> twoPlus(2)  
res3: Int = 4
```

```
scala> val mul = (x: Int, y: Int) => x*y  
mul: (Int, Int) => Int = <function2>  
  
scala> val mulOneAtTime = (x: Int) => ((y: Int) => x*y) 柯里化  
mulOneAtTime: Int => (Int => Int) = <function1>  
  
scala> mulOneAtTime(6)(7)  
res4: Int = 42  
  
scala> def mulOneTime1(x: Int)(y: Int) = x*y 简写的柯里化  
mulOneTime1: (x: Int)(y: Int)Int  
  
scala> mulOneTime1(6)(7)  
res5: Int = 42
```

5. 隐式转换和隐式参数

5.1. 概念

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。

5.2. 作用

隐式的对类的方法进行增强，丰富现有类库的功能

```
object ImplicitDemo extends App{  
  //定义隐式类，可以把File转换成定义的隐式类RichFile  
  implicit class RichFile(from: File){  
    def read:String = Source.fromFile(from.getPath).mkString  
  }  
  //使用隐式类做已有类的功能的扩展  
  val contents = new File("src/test1.txt").read  
  println(contents)  
}
```

5.3. 隐式引用

Scala会自己主动为每一个程序加上几个隐式引用，就像Java程序会自己主动加上java.lang包一样。

Scala中。下面三个包的内容会隐式引用到每一个程序上。所不同的是。Scala还会隐式加进对Predef的引用。

```
import java.lang._ // in JVM projects, or system namespace in .NET  
import scala._     // everything in the scala package  
import Predef._    // everything in the Predef object
```

上面三个包，包括了经常使用的类型和方法。java.lang包包括了经常使用的java语言类型，假设在.NET环境中，则会引用system命名空间。相似的，scala还会隐式引用scala包，也就是引入经常使用的scala类型。

请注意 上述三个语句的顺序藏着一点玄机。

我们知道，通常，假设import进来两个包都有某个类型的定义的话，比方说，同一段程序。即引用了'scala.collection.mutable.Set'又引用了'import scala.collection.immutable.Set'则编译器会提示无法确定用哪一个Set。

这里的隐式引用则不同，假设有同样的类型。后面的包的类型会将前一个隐藏掉。

比方。java.lang和scala两个包里都有StringBuilder。这样的情况下，会使用scala包里定义的那个。java.lang里的定义就被隐藏掉了，除非显示的使用java.lang.StringBuilder。

5.4. 隐式转换例子

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

//隐式的增强File类的方法
class RichFile(val from: File) {
  def read = Source.fromFile(from.getPath).mkString
}

object RichFile {
  //隐式转换方法
  implicit def file2RichFile(from: File) = new RichFile(from)
}

object ImplicitTransferDemo{
  def main(args: Array[String]): Unit = {
    //导入隐式转换
    import RichFile._
    //import RichFile.file2RichFile
    println(new File("c://words.txt").read)
  }
}
```

5.5. 隐式类

创建隐式类时，只需要在对应的类前加上implicit关键字。比如：

```
object Helpers {
  implicit class IntWithTimes(x: Int) {
    def times[A](f: => A): Unit = {
      def loop(current: Int): Unit =
        if(current > 0) {
          f
          loop(current - 1)
        }
      loop(x)
    }
  }
}
```

这个例子创建了一个名为IntWithTimes的隐式类。这个类包含一个int值和一个名为times的方法。要使用这个类，只需将其导入作用域内并调用times方法。比如：

```
scala> import Helpers._
import Helpers._
scala> 5 times println("HI")
HI
HI
HI
HI
HI
```

使用隐式类时，类名必须在当前作用域内可见且无歧义，这一要求与隐式值等其他隐式类型转换方式类似。

1. 只能在别的trait/类/对象内部定义。

```
object Helpers {
  implicit class RichInt(x: Int) // 正确！
}
implicit class RichDouble(x: Double) // 错误！
```

2. 构造函数只能携带一个非隐式参数。

```
implicit class RichDate(date: java.util.Date) // 正确！
implicit class Indexer[T](collecton: Seq[T], index: Int) // 错误！
implicit class Indexer[T](collecton: Seq[T])(implicit index: Index) // 正确！
```

虽然我们可以创建带有多个非隐式参数的隐式类，但这些类无法用于隐式转换。

3. 在同一作用域内，不能有任何方法、成员或对象与隐式类同名。

```
object Bar
implicit class Bar(x: Int) // 错误!

val x = 5
implicit class x(y: Int) // 错误!

implicit case class Baz(x: Int) // 错误!
```

5.6. 隐式转换函数

是指那种以implicit关键字声明的带有单个参数的函数，这种函数将被自动引用，将值从一种类型转换成另一种类型。

使用隐含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。这个规则非常简单，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。

比如，通常情况下，双精度实数不能直接当整数使用，因为会损失精度：

```
scala> val i:Int = 3.5
<console>:7: error: type mismatch;
 found   : Double(3.5)
 required: Int
    val i:Int = 3.5
           ^
```

当然你可以直接调用 3.5.toInt。

这里我们定义一个从 Double 到 Int 的隐含类型转换的定义，然后再把 3.5 赋值给整数，就不会报错。

```
scala> implicit def doubleToInt(x:Double) = x.toInt
doubleToInt: (x: Double)Int
scala> val i:Int = 3.5
i: Int = 3
```

此时编译器看到一个浮点数 3.5，而当前赋值语句需要一个整数，此时按照一般情况，编译器会报错，但在报错之前，编译器会搜寻是否定义了从 Double 到 Int 的隐含类型转换，本例，它找到了一个 doubleToInt。因此编译器将把

```
val i:Int = 3.5
```

转换成

```
val i:Int = doubleToInt(3.5)
```

这就是一个隐含转换的例子，但是从浮点数自动转换成整数并不是一个好的例子，因为会损失精度。Scala 在需要时会自动把整数转换成双精度实数，这是因为在 Scala.Predef 对象中定义了一个


```
implicit def int2double(x:Int) :Double = x.toDouble
```

而 `Scala.Predef` 是自动引入到当前作用域的，因此编译器在需要时会自动把整数转换成 `Double` 类型。

5.7. 隐式参数

看最开始的例子：

```
def compare[T](x:T,y:T)(implicit ordered: Ordering[T]):Int = {  
    ordered.compare(x,y)  
}
```

在函数定义的时候，支持在最后一组参数使用 `implicit`，表明这是一组隐式参数。在调用该函数的时候，可以不用传递隐式参数，而编译器会自动寻找一个 `implicit` 标记过的合适的值作为该参数。

例如上面的函数，调用 `compare` 时不需要显式提供 `ordered`，而只需要直接 `compare(1,2)` 这样使用即可。

再举一个例子：

```
object Test{  
    trait Adder[T] {  
        def add(x:T,y:T):T  
    }  
  
    implicit val a = new Adder[Int] {  
        override def add(x: Int, y: Int): Int = x+y  
    }  
  
    def addTest(x:Int,y:Int)(implicit adder: Adder[Int]) = {  
        adder.add(x,y)  
    }  
  
    addTest(1,2)          // 正确, = 3  
    addTest(1,2)(a)       // 正确, = 3  
    addTest(1,2)(new Adder[Int] {  
        override def add(x: Int, y: Int): Int = x-y  
    }) // 同样正确, = -1  
}
```

`Adder` 是一个 `trait`，它定义了 `add` 抽象方法要求子类必须实现。

`addTest` 函数拥有一个 `Adder[Int]` 类型的隐式参数。

在当前作用域里存在一个 `Adder[Int]` 类型的隐式值 `implicit val a`。

在调用 `addTest` 时，编译器可以找到 `implicit` 标记过的 `a`，所以我们不必传递隐式参数而是直接调用 `addTest(1,2)`。而如果你想要传递隐式参数的话，你也可以自定义一个传给它，像后两个调用所做的一样。

5.8 隐式转换机制

隐式转换的前提：

在进行隐式转换时，需要遵守两个基本的前提：

- 不能存在二义性
- 隐式操作不能嵌套使用

隐式转换的时机：

- 当方法中的参数的类型与目标类型不一致时
- 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（根据类型）

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：

- 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）。
- (一般是这种情况)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(第二种情况范围广且复杂在使用时，应当尽量避免出现)：
 - 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。

6.泛型类(备选)

为什么要用泛型？

提高某些类或者方法的灵活性。

什么是泛型？

带有一个或多个类型参数的类是泛型的。

[B <: A] upper bounds 上限或上界：B类型的上界是A类型，也就是B类型的父类是A类型
[B >: A] lower bounds 下限或下界：B类型的下界是A类型，也就是B类型的子类是A类型
[B <% A] view bounds 视界：表示B类型要转换为A类型，需要一个隐式转换函数
[B : A] context bounds 上下文界定（用到了隐式参数的语法糖）：需要一个隐式转换的值
[-A]：逆变，作为参数类型。是指实现的参数类型是接口定义参数类型的父类
[+B]：协变，作为返回类型。是指返回类型是接口定义返回类型的子类

泛型类的定义：

```
//带有类型参数A的类定义
class Stack[A] {
  private var elements: List[A] = Nil
  //泛型方法
  def push(x: A) { elements = x :: elements }
  def peek: A = elements.head
  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}
```

使用上述的泛型类，使用具体的类型代替类型参数A。

```
val stack = new Stack[Int]
stack.push(1)
stack.push(2)
println(stack.pop) // prints 2
println(stack.pop) // prints 1
```

6.1.协变

定义一个类型List[+A]，如果A是协变的，意思是：对类型A和B，A是B的子类型，那么List[A]是List[B]的子类型。

```
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal
```

Scala标准库有一个泛型类sealed abstract class List[+A]，因为其中的类型参数是协变的，那么下面的程序调用时成功的。

```
object CovarianceTest extends App {
  //定义参数类型List[Animal]
  def printAnimalNames(animals: List[Animal]): Unit = {
    animals.foreach { animal =>
      println(animal.name)
    }
  }

  val cats: List[Cat] = List(Cat("Whiskers"), Cat("Tom"))
  val dogs: List[Dog] = List(Dog("Fido"), Dog("Rex"))
  //传入参数类型为List[Cat]
  printAnimalNames(cats)
  // Whiskers

  // Tom
```

```
//传入参数类型为List[Dog]
printAnimalNames(dogs)
// Fido
// Rex
}
```

6.2.逆变

定义一个类型Writer[-A]，如果A是逆变的，意思是：对类型A和B，A是B的子类型，那么Writer[B]是Writer[A]的子类型。

```
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal
```

定义对应上述类进行操作的打印信息类

```
abstract class Printer[-A] {
  def print(value: A): Unit
}
class AnimalPrinter extends Printer[Animal] {
  def print(animal: Animal): Unit =
    println("The animal's name is: " + animal.name)
}

class CatPrinter extends Printer[Cat] {
  def print(cat: Cat): Unit =
    println("The cat's name is: " + cat.name)
}
```

逆变的测试

```
object ContravarianceTest extends App {
  val myCat: Cat = Cat("Boots")

  //定义参数类型为Printer[Cat]
  def printMyCat(printer: Printer[Cat]): Unit = {
    printer.print(myCat)
  }

  val catPrinter: Printer[Cat] = new CatPrinter
  val animalPrinter: Printer[Animal] = new AnimalPrinter

  printMyCat(catPrinter)
  //可以传入参数类型为Printer[Animal]
  printMyCat(animalPrinter)
}
```

6.3.上界

上界定义：`T <: A`，表示类型变量 `T` 必须是 类型 `A` 子类

```
abstract class Animal {
  def name: String
}

abstract class Pet extends Animal {}

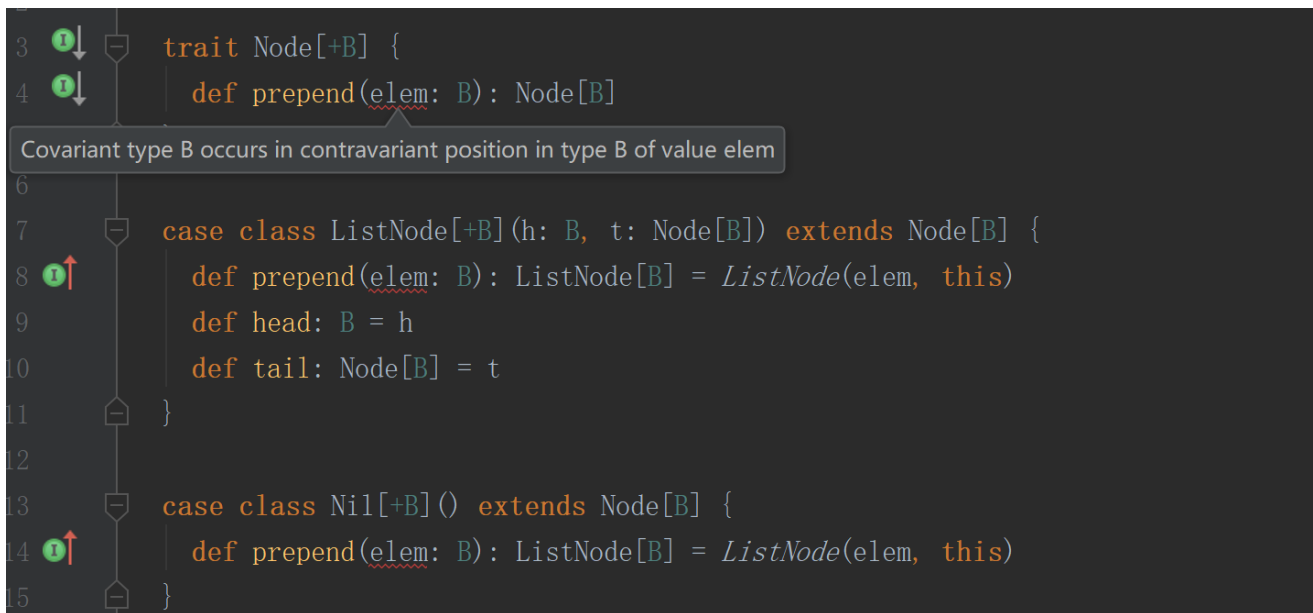
class Cat extends Pet {
  override def name: String = "Cat"
}

class Dog extends Pet {
  override def name: String = "Dog"
}

class Lion extends Animal {
  override def name: String = "Lion"
}
//参数类型须是Pet类型的子类
class PetContainer[P <: Pet](p: P) {
  def pet: P = p
}
//Dog是Pet类型的子类
val dogContainer = new PetContainer[Dog](new Dog)
//Cat是Pet类型的子类
val catContainer = new PetContainer[Cat](new Cat)
//Lion不是Pet类型的子类，编译通不过
// val lionContainer = new PetContainer[Lion](new Lion)
```

6.4.下界

语法 `B >: A` 表示参数类型或抽象类型 `B` 须是类型 `A` 的父类。通常，`A` 是类的类型参数，`B` 是方法的类型参数。



上面这段代码，因为作为协变类型的B，出现在需要逆变类型的函数参数中，导致编译不通过。解决这个问题，就需要用到下界的概念。

```
trait Node[+B] {
  def prepend[U >: B](elem: U): Node[U]
}

case class ListNode[+B](h: B, t: Node[B]) extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
  def head: B = h
  def tail: Node[B] = t
}

case class Nil[+B]() extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
}
```

测试

```
trait Bird
case class AfricanSwallow() extends Bird
case class EuropeanSwallow() extends Bird

val africanSwallowList= ListNode[AfricanSwallow](AfricanSwallow(), Nil())
val birdList: Node[Bird] = africanSwallowList
birdList.prepend(new EuropeanSwallow)
```

8.5 视界(view bounds)

注意:已过时,了解即可

视界定义：`A <% B`，表示类型变量A 必须是 类型 B`的子类,或者A能够隐式转换到B

```

class Pair_Int[T <% Comparable[T]] (val first: T, val second: T){
  def bigger = if(first.compareTo(second) > 0) first else second
}

class Pair_Better[T <% Ordered[T]](val first: T, val second: T){
  def smaller = if(first < second) first else second
}
object View_Bound {

  def main(args: Array[String]) {
    // 因为Pair[String] 是Comparable[T]的子类型, 所以String有compareTo方法
    val pair = new Pair_Int("Spark", "Hadoop");
    println(pair.bigger)

    /**
     * Scala语言里 Int类型没有实现Comparable;
     * 那么该如何解决这个问题那;
     * 在scala里 RichInt实现了Comparable, 如果我们把int转换为RichInt类型就可以这样实例化了.
     * 在scala里 <% 就起这个作用, 需要修改Pair里的 <: 为<% 把T类型隐身转换为Comparable[Int]
     * String可以被转换为RichString. 而RichString是Ordered[String] 的子类.
     */
    val pair_int = new Pair_Int(3 ,45)
    println(pair_int.bigger)

    val pair_better = new Pair_Better(39 ,5)
    println(pair_better.smaller)

  }

}

```

8.6 上下文界定(context bounds)

上下文界定的形式为 $T : M$, 其中M 必须为泛型类, 必须存在一个M[T]的隐式值.

```

class Pair_Context[T : Ordering](val first: T, val second: T){
  def smaller(implicit ord: Ordering[T]) =
    if(ord.compare(first, second) < 0) first else second
}

object Context_Bound {

  def main(args: Array[String]) {

    val pair = new Pair_Context("Spark", "Hadoop")
    println(pair.smaller)

    val int = new Pair_Context(3, 5)
    println(int.smaller)

  }

}

```

