# Scala高级语法

## 柯里化(Currying)

科里化是一个过程。

柯里化(Currying)指的是将原来接受两个参数的方法变成新的接受一个单一参数的方法的过程。新的方法返回一个以原有第二个参数为参数的函数。

|  |
| --- |
| *// 我们看下这个方法的定义, 求2个数的和* **def** add(x: Int, y: Int) = x + y *// 那么我们应用的时候，应该是这样用：add(1,2) // 现在我们把这个函数变一下形：* **def** add(x:Int)(y:Int) = x + y *// 那么我们应用的时候，应该是这样用：add(1)(2),最后结果都一样是3，这种方式（过程）就叫柯里化。经过柯里化之后，方法的通用性有所降低，但是适用性有所提高。* |

## 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？？

如果要实现该功能，需要用到隐式转换

implicit 可分为：

* 隐式参数
* 隐式转换类型
* 隐式类（自己扩展）

### 隐式参数

隐式参数就用implicit 关键字修饰的参数，隐式参数初始化后，不需要显示的传入参数

如果说在当前环境（context上下文环境）找到了一个和隐式参数类型相同，且用implicit修饰的参数，那么默认的初始值就会被覆盖，而且可以不设置初始值。

上下文环境中，有且只能有一个满足条件的隐式值，没有或匹配上多个会报错。

有多个隐式参数，也只能有一个implicit关键字，且要置于最后一个参数列表中。

只要有隐式参数，就只能出现在最后一个参数列表的位置。

**def** m1(x:Int)( **implicit** y:Int=10)(z:Int)=x + y // 编译错误

|  |
| --- |
| **object** ImplicitDemo {  **implicit val** *abc* = 12000  **implicit val** *str*= **"hello"** *// implicit val str2= "hello"* **def** m1(x:Int)( **implicit** y:Int=10)=x + y  *// 隐式参数只能放在最后一个参数列表中，m2方法是错误的 // def m2(x:Int)( implicit y:Int=10)(z:Int)= x + y + z  // 有多个隐式参数的定义方式* **def** m3(x:Int)(**implicit** y:String,z:Double=30d) ={ x + z }  **def** main(args: Array[String]): Unit = {  *println*(*m1*(1)) *// println(m3(10))* } } |

如果在其他的object中定义的隐式值，可以通过import 对象导入：

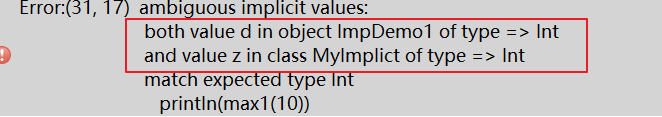
|  |
| --- |
| **import** ValuesDemo.\_ |

如果在其他的class中定义的隐式值，需要创建类的实例，然后 import实例对象.隐式值

|  |
| --- |
| **val** *vc* = **new** ValueClass() **import** *vc*.varparam |

隐式参数总结：

1. 使用implicit关键字修饰的参数，就是隐式参数
2. 一旦方法中有隐式参数，那么在调用该方法的时候，可以不用显示传递参数了
3. 隐式参数值的使用顺序，优先使用上下文环境中的值（和隐式参数类型一致，而且使用implicit关键字修饰，有且只能有一个），如果上下文环境中没有，就使用默认值，默认值也没有，就报错。
4. 编译器会去上下文环境中找对应的值，类型相同，变量必须使用implicit关键字修饰，最多只能找到一个，然后就会把这个找到的值赋予给隐式参数。如果找到多个，就报错



1. 如果想设置多个隐式参数，只能在**最后一个参数列表**中，而且只能使用一个implicit关键字



1. 我们可以把隐式的变量，写到外部的对象中，或者外部的类中，然后可以通过import关键字导入外部的隐式变量。类必须先new 类（）

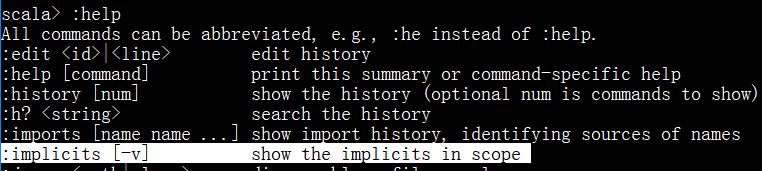
### 隐式转换

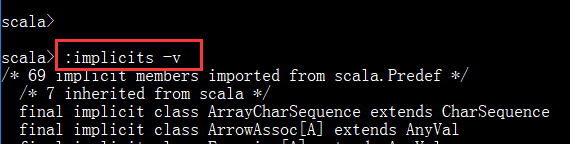
使用隐式转换将变量转换成预期的类型是编译器最先使用implicit的地方。当编译器看到类型X而却需要类型Y，它就在当前作用域查找是否定义了从类型X到类型Y的隐式定义。

|  |
| --- |
| *// 使用to这个生成器的时候，会发现在Int类中，根本就没有to方法,实际调用的是RichInt方法* **val** rg = 1.to(10) *println*(rg) *// 该类所在包的位置：scala.runtime.RichInt* **val** rg2 = **new** RichInt(1).to(10) *println*(**s"richint : $**{rg2}**"**)  */\*\*  \* 发生了什么？？就是因为implicit 隐式转换  \* 在调用Int的to方法时，发现没有，但是呢？在上下文context环境中，找到了一个implicit修饰的方法：  \* implicit def intWrapper(x: Int) = new runtime.RichInt(x)  \* 该方法的输入参数是一个Int类型，返回值类型是RichInt类型，该类型中有to 方法  \* 于是编译器，就调用了这个类型的to方法，并返回相应的结果  \*/* |

该隐式方法定义在Predef对象中，该对象会被编译器默认引入。

在scala交互式命令行中，可使用:implicits -v 来查看默认导入的隐式转化。





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

方案1：使用装饰模式实现File类的read方法增强：

|  |
| --- |
| **class** RichFile(**val** file: File) {  *// 使用scala.io.Source来读取文件， 并把结果拼接为字符串* **def** read() = Source.*fromFile*(file).mkString } **object** RichFile {  **def** main(args: Array[String]): Unit = {  **val** file = **new** File(**"hello.txt"**)  *// File类中并没有read方法来返回所有的内容 file.read()*  *// 使用装饰模式* **val** rf: RichFile = **new** RichFile(file)  **val** content = rf.read()  *println*(content)} } |

如果同时有同样功能的 隐式的方法和隐式的函数，优先使用函数。因为函数是头等公民。

方案2：

定义一个RichFile类及伴生对象，在伴生对象中定义一个隐式转换方法（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。

|  |
| --- |
| **class** RichFile(**val** file: File) {  *// 使用scala.io.Source来读取文件， 并把结果拼接为字符串* **def** read() = Source.*fromFile*(file).mkString } **object** RichFile {  **def** main(args: Array[String]): Unit = {  **val** file = **new** File(**"hello.txt"**)  *// scala 隐式转换* **implicit def** file2RichFile(file:File):RichFile=**new** RichFile(file)  *// 我们普通的一个File类，没有这样一个read方法，那么scala提供一种隐式转换的方式，  // 会在当前的上下文环境中，去找，有没有这样一个方法，把File转换为另一个类，  // 而且这个类中，有和read 方法签名一样的方法: 方法名，方法返回值类型，方法参数*  **implicit val** f = (file:File) => **new** RichFile(file)  **val** content:String= file.read()  *println*(content)  } } |

自定义示例：

|  |
| --- |
| **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 *// 成功输出3* |

隐式转换：

要调用的方法，当前对象上没有，

可以定义一个隐式方法（隐式函数），使用implicit关键字修饰的方法或者函数。

该方法的名称，是任意，但是参数类型，必须是当前对象，

返回值类型是有 该方法的对象。

richFile

File类型上，没有read方法，但是RichFIle有read方法(方法名称和返回值类必须一致)。

定义隐式方法，参数类型是file，返回值类型是richFIle

隐式函数的优先级高于隐式方法，因为函数是一等公民。

可以导入其他的object或者class中的隐式转换。

如果是object中的，直接使用对象名.\_ 导入所有的，或者使用对象名.方法名（函数名） 导入一个具体的隐式转换。

如果是在class中，就必须先创建实例对象。然后实例对象.\_ 实例对象.\_方法名称（函数名称）

隐式转换总结：

1. 使用implicit关键字修饰的方法和函数，就称为隐式方法和隐式函数，优先使用隐式函数。
2. scala默认导入的隐式转换都在Predef对象中,可以在scala交互式命令行中使用

：implicits -v 来查看所有默认导入的隐式转换

1. 方法的名称任意（要有意义），方法的 **参数和返回值类型**，要想从FIle到RichFIle的隐式转换，File是方法的参数，RichFile类型。
2. 除了在当前的上下文环境中定义隐式转换，还可以导入外部的隐式转换（外部的类和外部的object，类必须new 实例 ，然后才能导入实例上的隐式转换）。

## 泛型

java中：<T> scala: [T]

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。

|  |
| --- |
| *// 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)  **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)  *// 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)  }  } } |

## 类型约束

### 比较器Ordered和Ordering

回顾java中的比较器Comparable和Compartor

定义一个类，实现Comparable[T]接口,并重写compareTo方法。

|  |
| --- |
| public class Girl implements Comparable<Girl>{  private String name ;  private int age;  public Girl(String name,int age){  this.name=name;  this.age=age;  }  … setter and getter …  @Override  public String toString() {  return "Girl{" +"name='" + name + '\'' +", age=" + age +'}';  }  @Override  public int compareTo(Girl o) {  return this.age - o.age;  } } |

测试：

|  |
| --- |
| **public class** TestGirl {  **public static void** main(String[] args) {  Girl g1 = **new** Girl(**"yangmi"**, 29);  Girl g2 = **new** Girl(**"dongjie"**, 18);  Girl g3 = **new** Girl(**"fengjie"**, 38);  ArrayList<Girl> girls = **new** ArrayList<>();  girls.add(g1);  girls.add(g2);  girls.add(g3);  Collections.*sort(girls);* **for**(Girl g :girls){  System.***out***.println(g);  }  } } |

不实现接口，直接使用Compartor接口方法

|  |
| --- |
| **public class** TestGirl {  **public static void** main(String[] args) {  Girl g1 = **new** Girl(**"yangmi"**, 29);  Girl g2 = **new** Girl(**"dongyue"**, 18);  Girl g3 = **new** Girl(**"fengjie"**, 38);  ArrayList<Girl> girls = **new** ArrayList<>();  girls.add(g1);  girls.add(g2);  girls.add(g3);  // 使用比较器  Collections.*sort*(girls, **new** Comparator<Girl>() {  @Override  **public int** compare(Girl o1, Girl o2) {  **return** -(o1.getAge() - o2.getAge());  }  });  **for**(Girl g :girls){  System.***out***.println(g);  }  } } |

### scala中的比较器特质

scala中也有类似的两个特质 Ordered 和 Ordering：

Ordered 相当于 java中的Comparable 功能更丰富 扩展了compare < > 等方法

Ordering 相当于 java中的Comparator 是一个比较器

### 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 >

或者通配符的形式：

<? 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 ]

或者：

[ \_ <: Test ]

* **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 >

或者通配符的形式：

<? super Test>

这种形式也叫lower bounds(下限或下界)， scala中的写法为：

[ T >: Test ]

或者：

[ \_ >: Test ]

示例：

|  |
| --- |
| *// 定义一个class 实现了Ordered特质* **class** BGirl(**val** name:String,**val** facVal:Int,**val** age :Int) **extends** Ordered[BGirl]{  *// 重写compare方法* **override def** compare(that: BGirl): Int = {  *// 比较 颜值相同，比较年龄 ; 如果颜值不同，就比较颜值* **if**(**this**.facVal == that.facVal){  **this**.age - that.age  }**else**{  **this**.facVal - that.facVal  }  } } *// 泛型限制了上界，比较的类型必须是Ordered的子类或者实现类* **class** TestOrderedCompare[T <: Ordered[T]] {  *// 定义一个比较方法 使用> 比较* **def** bigger(first: T, second: T): T = {  **if**(first > second) first **else** second  } }  **object** TestBGirl **extends** App{ *// 普通的int类型没有实现Ordered特质，无法进行比较* **val** *i*:Int = 10  **val** *j*:Int = 20  *// val com = new TestOrderedCompare[Int]* *// val result = compare.bigger(i,j) // println(res.name)   // class BGirl 实现了Ordered特质，所以可以传入比较器中进行比较* **private val** *g1* = **new** BGirl(**"ab"**,95,30)  **private val** *g2* = **new** BGirl(**"hatono"**,95,28)  **val** *compare* = **new** TestOrderedCompare[BGirl]  **val** *res* = *compare*.bigger(*g1*,*g2*)  *println*(*res*.name) } |

下界示例：

|  |
| --- |
| val arr = *Array*[Int](1,3,5,6) val f0:(AnyVal,AnyVal) => AnyVal = (x,y) => 100 val f:(AnyVal,AnyVal) => AnyVal = (x,y) => if(!x.isInstanceOf[Unit]) x.asInstanceOf[Int] + y.asInstanceOf[Int] val f2:(String,String) => String = (x,y) => "x" *println*(arr.reduce(f)) *// 正确：因为 泛型A的类型是Int，由arr的类型决定的，而且AnyVal是Int类型的父类，所以f可以传递 // println(arr.reduce(f2)) // 错误，因为 String类型不是Int类型的父类，所以这里不能传递* |

总结：

**T <:**  Test upper bounds 上界 相当于给定一个最高标准 传入的类型必须是最高标准的子类或实现类 <? **extends** Comparable<T>>  
**>:** lower bounds下界 相当于给定了一个最低标准 传入的类型必须是最低标准的父类或超类 <? **super** Comparable<T>

|  |
| --- |
| 类型判断：  isInstanceOf  “abc”.isInstanceOf[String]  类型转换：  x.asInstanceOf[Int] |

### 视图界定/上下文界定(扩展)

* **View bounds**

[ T <% Test ]

**<%** 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。

包括：隐式方法或者隐式函数或者隐式参数（科里化）

示例：

|  |
| --- |
| **class** Boy(**val** name:String,**val** facVal :Int) {} **class** Pari[T <% Ordered[T]] {  *// 定义一个比较方法* **def** bigger(first: T, second: T): T = {  **if** (first > second) first **else** second  } } **object** Pari **extends** App {  **val** *b1* = **new** Boy(**"段子王"**, 99)  **val** *b2* = **new** Boy(**"laozhao"**, 999)  *// Boy类没有实现Ordered，就不能进行比较*  *// 但是在类的定义中，使用视图界定 <% 就可以使用隐式方法 进行隐式转换*  *// 然后Boy类就可以正常比较了* **implicit def** man2OrderedMan2(man:Boy):Ordered[Boy]=**new** Ordered[Boy]{  *// 排序的规则 按照颜值 升序* **override def** compare(that: Boy): Int = {  man.facVal - that.facVal  }  }  *// 除了隐式方法，还可以使用隐式函数 来达到隐式转化的目的 二者选其一即可，优先使用函数* **implicit val** *man2OrderedMan* = (boy:Boy) => **new** Ordered[Boy]{  **override def** compare(that: Boy): Int = {  boy.facVal - that.facVal  } }  **val** *p1* = **new** Pari[Boy]  **val** *bigger* = *p1*.bigger(*b1*, *b2*)  *print*(*bigger*.name) } |

使用科里化方法：

|  |
| --- |
| **class** TestBGirl[T]{  *// 定义了一个比较的方法 添加了一个科里化的参数* **def** bigger(x:T,y:T)(**implicit** order:T => Ordered[T]):T={  **if**(x > y) x **else** y  } } |

总结：

视图界定，如果要比较的类实现了Ordered特质，可以直接进行比较。

视图界定支持隐式转换，如果要比较的对象没有实现Ordered特质，可使用隐式转换

隐式转换支持：隐式函数，隐式方法，

Ordered特质的类型限定，可以在类上，也可以在方法上添加隐式参数实现。

* **Context bounds**

[ T : Test ]

与view bounds一样context bounds(上下文界定)也是隐式参数的语法。上下文 界定 可以实现隐式转换，需要传入隐式值，实际上就是 一个隐式的object

|  |
| --- |
| **class** Pari2[T: Ordering](**implicit** ord:Ordering[T]) {  **def** bigger(first: T, second: T)*/\*(implicit ord:Ordering[T])\*/*: T = {  **if** (ord.gt(first, second)) first **else** second  } } **object** Pari2 **extends** App{  **val** *b1* = **new** Boy(**"段子王"**,99)  **val** *b2* = **new** Boy(**"laozhao"**,999)  *// 没有implicit Ordering 被定义 不能比较   // 方法一：定义一个变量,类型为要求的比较器类型Ordering[Boy]* **implicit val** *b2Ordering*:Ordering[Boy] = **new** Ordering[Boy] {  **override def** compare(x: Boy, y: Boy): Int = x.facVal - y.facVal  }   *// 方法二：定义一个隐式值（隐式object） 继承自比较器类型 // implicit object man2Ordering extends Ordering[Boy] { // override def compare(x: Boy, y: Boy): Int = { // x.facVal - y.facVal // } // }  // 在创建比较对象时，进行隐式转换* **val** *p* = **new** Pari2[Boy]  **val** *re* = *p*.bigger(*b1*,*b2*)  *println*(*re*.name) }  **class** Boy(**val** name: String, **val** facVal:Int) |

总结：

1. 上下文界定，需要有比较器。
2. 可以在类定义时利用科里化方式传递隐式的参数，也可以在比较方法定义时，传入一个科里化参数，参数类型为一个比较器。

3，具体的隐式值，可以定义一个比较器类型的变量。也可以定义一个隐式的object对象，然后该对象继承Ordering特质。