scala

介绍

1.什么是Scala

Scala是一门现代的多范式语言，志在以简洁、优雅及类型安全的方式来表达常用的编程模型。它平滑地集成了面向对象和函数式语言的特性。

2.Scala是面向对象编程语言

鉴于一切值都是对象，可以说Scala是一门纯面向对象的语言。对象的类型和行为是由类和特质来描述的。类可以由子类化和一种灵活的、基于mixin的组合机制（它可作为多重继承的简单替代方案）来扩展。

3.Scala是函数式编程语言

鉴于一切函数都是值，又可以说Scala是一门函数式语言。Scala为定义匿名函数提供了轻量级的语法，支持高阶函数，允许函数嵌套及柯里化。Scala的样例类和内置支持的模式匹配代数模型在许多函数式编程语言中都被使用。对于那些并非类的成员函数，单例对象提供了便捷的方式去组织它们。

此外，通过对提取器的一般扩展，Scala的模式匹配概念使用了right-ignoring序列模式，自然地延伸到XML数据的处理。其中，for表达式对于构建查询很有用。这些特性使得Scala成为开发web服务等程序的理想选择。

4.Scala是静态类型的

Scala配备了一个拥有强大表达能力的类型系统，它可以静态地强制以安全、一致的方式使用抽象。典型来说，这个类型系统支持：

泛型类

型变注解

上、下 类型边界

作为对象成员的内部类和抽象类型

复合类型

显式类型的自我引用

隐式参数和隐式转化

多态方法

类型推断让用户不需要标明额外的类型信息。这些特性结合起来为安全可重用的编程抽象以及类型安全的扩展提供了强大的基础。

5.Scala是可扩展的

在实践中，特定领域应用的发展往往需要特定领域的语言扩展。Scala提供了一种语言机制的独特组合方式，使得可以方便地以库的形式添加新的语言结构。

很多场景下，这些扩展可以不通过类似宏（macros）的元编程工具完成。例如：

隐式类允许给已有的类型添加扩展方法。

字符串插值可以让用户使用自定义的插值器进行扩展。

6.Scala互操作

Scala设计的目标是与流行的Java运行环境（JRE）进行良好的互操作，特别是与主流的面向对象编程语言——Java的互操作尽可能的平滑。Java的最新特性如函数接口（SAMs）、lambda表达式、注解及泛型类 在Scala中都有类似的实现。

另外有些Java中并没有的特性，如缺省参数值和带名字的参数等，也是尽可能地向Java靠拢。Scala拥有类似Java的编译模型（独立编译、动态类加载），且允许使用已有的成千上万的高质量类库。

1 变量/不变量声明与定义

值与变量的声明

var/val x:T = e

variable 变量

value 值 不变量

val 不变量声明（常量，类似java中final修饰的变量）

var 变量声明

i,scala自动类型推断，即声明值时可以不声明类型，scala自动识别类型

ii,scala中变量/不变量 必须初始化（赋值）,不能先声明后赋值

iii, 为了符合ii的要求，在没有逻辑值使用的情况下，

可使用 \_ 对于变量/不变量进行赋值

代表默认值（参考java属性默认值）

iv, 声明可以连续声明，

val x1,x2,x3 = 10

等价于

val x1 = 10

val x2 = 10

val x3 = 10

var同上

v,在REPL中,所有裸值和运算结果都会用resX 接收以作为之后使用

resX是 val声明的

vi,在scala中 val是更推荐的一种变量类型

2 方法声明 与 函数声明

def methodName(param:T,...):R={

return R;

}

i,参数的类型必须声明,可以为参数提供默认值

def show(msg:String = "")=print(msg)

ii,如果没有返回值可以不写返回类型

iii,如果有返回值，可以省略return，代码块中最后一行必须产生返回值且类型要匹配

iv,Unit代表无返回值，scala中没有void

:Unit可以省略

v, 没有返回值的方法 可以省略 = ,此写法被称为 过程

vi,不建议声明过程,建议方法声明长期添加 =

vii, 方法有返回值类型,返回类型也可以省略,代码块最后一行

作为返回值，且scala会做类型推断，将最后一行结果类

型，当做方法返回值类型

viii,递归方法必须声明返回值类型

def fac(i:Int):Int={

if(i > 0) i\*fac(i-1) else 1

}

ix, 如果方法没有参数，可以省略小括号，此时

要注意调用方式，声明时带有小括号的方法，调用时可以有小括号

也可以没有小括号

对于无小括号的方法，调用时必须不带小括号

x,如果方法体中只有一行代码，则可以省略{}

函数声明

可以用变量接收的方法

函数是带参数的表达式。

1.可以定义一个匿名函数（即无名称），返回给定的整数加一的结果：

(x:Int)=>x+1

=>的左边是参数列表，右边是一个包含参数的表达式。

2.带名字的函数:

val addOne=(x:Int)=>x+1

println(addOne(1))//2

3.多个参数的函数

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

println(add(1,2))//3

4.无参数的函数

val getTheAnswer=()=>42

println(getTheAnswer())//42

3 类的声明

Scala中的类是用于创建对象的蓝图，其中包含了方法、常量、变量、类型、对象、特质、类，这些统称为成员。类型、对象和特质将在后面的文章中介绍。

class 类名 private[package/this] (主构造器参数列表) extends 类/特质 with 特质1 with 特质2{}

class name(param:T,....){

}

i,类名后面的参数列表代表构造器的参数列表

class Student{

def show=print("hello")

}

var stu = new Student

ii,类名后面声明的构造器称为主构造器,

class Student(name:String){

def show=print("hello"+name)

}

类中声明的构造器称为辅助构造器

1. 辅助构造器的名称为def this

2. 每个辅助构造器都必须以一个对先前已定义的其他辅助构造器或主构造器的调用开始

class Student {

private var name = " "

private var age = 0

def this(name: String){ //辅助构造器1

this() //调用主构造器

this.name = name

}

def this(name: String,age: Int){ //辅助构造器2

this(name) //调用前一个辅助构造器

this.age = age

}

}

iii,私有构造器

class Student private(name:String)

iv,主构造器的参数,如果在类内的方法中使用过，则升格成字段/属性

class Student(name:String){

def show=print("hello"+name)

}

v,主构造器参数，

如果用var修饰，则升格成字段/属性并默认提供get/set函数

如果用val修饰，则升格成字段/属性并默认提供get函数

get函数名为 字段名

set函数名为 字段名\_=

class Student(var name:String){

def show=print("hello"+name)

}

object Student extends App {

var s = new Student("zhangsan");

s.name\_=("lisi")

print(s.name)

}

@BeanProperty 可以让生成的字节码文件中提供 java中的set方法,并且调用 字段名\_= 时候转化成调用set字段名

class Student(@BeanProperty var name:String){

def show=print("hello"+name)

}

object Student extends App {

var s = new Student("zhangsan");

s.name="lisi" //等于调用了setName

print(s.name)

}

vi,主构造器 参数 带private var/val,则提供私有的get/set

vii,类中可以声明属性 和 函数,scala中没有static修饰符

类中的函数都是非静态函数

样例类

Scala有一种特殊的类叫做样例类（case class）。

默认情况下，样例类一般用于不可变对象，并且可作值比较。

你可以使用case class关键字来定义样例类。

case class Point(x: Int, y: Int)

1.实现了apply方法，意味着你不需要使用new关键字就能创建该类对象

val point = Point(1, 2)

val anotherPoint = Point(1, 2)

val yetAnotherPoint = Point(2, 2)

2.它们的值可以进行比较。

if (point == anotherPoint) {

println(point + " and " + anotherPoint + " are the same.")

} else {

println(point + " and " + anotherPoint + " are different.")

} // Point(1,2) and Point(1,2) are the same.

if (point == yetAnotherPoint) {

println(point + " and " + yetAnotherPoint + " are the same.")

} else {

println(point + " and " + yetAnotherPoint + " are different.")

} // Point(1,2) and Point(2,2) are different.

//具体原因之后讨论

3.实现了类构造参数的getter方法（构造参数默认被声明为val），但是当你构造参数是声明为var类型的，它将帮你实现setter和getter方法

4 对象的声明

object name{

}

i,scala中没有static,故想表示static的含义,利用对象表示

对象中的函数默认都是静态函数,用对象名直接调用

object Test{

def show=println(hello)

}

Test.show

ii,拥有同名class的object称为该class的伴生对象,在同一个源文件中的话，可以利用伴生对象访问class的私有属性

class Student(@BeanProperty private var name:String){

def show=print("hello"+name)

}

object Student extends App {

var s = new Student("zhangsan");

s.name="lisi" //等于调用了setName

print(s.name)

}

iii, apply方法的使用，通常情况下，我们会在伴生对象中定义一个apply方法，来产生该类的实例,scala允许语法简化成对象名() 参考scala api

class Student(name:String){

def show=print("hello"+name)

}

object Student{

def apply(name: String): Student = new Student(name)

def main(args:Array[String]){

var s = Student("zhangsan");

}

}

特质

特质 (Traits) 用于在类 (Class)之间共享程序接口 (Interface)和字段 (Fields)。 它们类似于Java 8的接口。 类和对象 (Objects)可以扩展特质，但是特质不能被实例化，因此特质没有参数。

1定义特质:

格式: trait 特质名 extends 类/特质 with 特质1 {}

总结:

1.可以包含具体方法/属性

2.可以包含抽象方法/属性

3.特质 可以 混入 类/对象/特质/实例(对象)

案例:

例1:最简化的特质就是关键字trait+标识符：

trait HairColor

例2:特征作为泛型类型和抽象方法非常有用。

trait Iterator[A] {

def hasNext: Boolean

def next(): A

}

扩展 trait Iterator [A] 需要一个类型 A 和实现方法hasNext和next。

2使用特质:

使用 extends 关键字来扩展特征。然后使用 override

关键字来实现trait里面的任何抽象成员：

trait Iterator[A] {

def hasNext: Boolean

def next(): A

}

class IntIterator(to: Int) extends Iterator[Int] {

private var current = 0

override def hasNext: Boolean = current < to

override def next(): Int = {

if (hasNext) {

val t = current

current += 1

t

} else 0

}

}

val iterator = new IntIterator(10)

iterator.next() // returns 0

iterator.next() // returns 1

这个类 IntIterator 将参数 to 作为上限。它扩展了 Iterator [Int]，这意味着方法 next 必须返回一个Int。

override 可以重写父类中的字段和方法

3子类型:

凡是需要特质的地方，都可以由该特质的子类型来替换。

import scala.collection.mutable.ArrayBuffer

trait Pet {

val name: String

}

class Cat(val name: String) extends Pet

class Dog(val name: String) extends Pet

val dog = new Dog("Harry")

val cat = new Cat("Sally")

val animals = ArrayBuffer.empty[Pet]

animals.append(dog)

animals.append(cat)

animals.foreach(pet => println(pet.name)) // Prints Harry Sally

在这里 trait Pet 有一个抽象字段 name ，name 由Cat和Dog的构造函数中实现。最后一行，我们能调用pet.name的前提是它必须在特质Pet的子类型中得到了实现。

4通过混入（MIXIN）来组合类

当某个特质被用于组合类时，被称为混入。

abstract class A {

val message: String

}

class B extends A {

val message = "I'm an instance of class B"

}

trait C extends A {

def loudMessage = message.toUpperCase()

}

class D extends B with C

val d = new D

println(d.message) // I'm an instance of class B

println(d.loudMessage) // I'M AN INSTANCE OF CLASS B

类D有一个父类B和一个混入C。一个类只能有一个父类但是可以有多个混入（分别使用关键字extend和with）。混入和某个父类可能有相同的父类。

5 类型声明

type A = Int

当类名过长，或者有重复时进行使用，减少代码长度

scala本身没有定义String类型，利用type 引用java.lang.String

查看Predef类

6 scala标示符与命名

变量 类名 对象名 函数 标示符

以字母或下划线开头，后面可以有更多的字母，数字或下划线。 $字符是Scala中的保留关键字，不应在标识符中使用。

以下是合法的字母数字标识符

age, salary, \_value, \_\_1\_value

以下是非法标识符

$salary, 123abc, -salary

反引号可以包围特殊标示符

var `return` = 10

var return = 10

7 scala中的关键字

implicit

Scala 2.10引入了一种叫做隐式类的新特性。隐式类指的是用implicit关键字修饰的类。在对应的作用域内，带有这个关键字的类的主构造函数可用于隐式转换。

例如，定义函数

def show(msg:String)=println(msg)

此时 调用show必须传参String,如果传参Int则报错

可以用implicit def帮助我们在一定代码范围内将

Int 转化成 String

object Student{

def show(msg:String)=print(msg)

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

implicit def intToString(i:Int)=i.toString

show("hello")

show(300)

}

}

i,隐式参数

当我们在定义方法时，可以把最后一个参数列表标记为implicit

表示该组参数是隐式参数。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。 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

改组参数填充合适的值。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。

scala > def calcTax(amount: Float)(implicit rate: Float): Float = amount \* rate

scala > implicit val currentTaxRate = 0.08F

scala > val tax = calcTax(50000F) // 4000.0

ii,隐式转换类型 （隐式函数,隐式类）

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

iii,隐式调用函数

隐式调用函数可以转换调用方法的对象，

比如编译器看到X .method，而类型 X

没有定义 method（包括基类)方法，那么编译器就查找

作用域内定义的从 X 到其它对象的类型转换，

比如 Y，而类型Y定义了 method 方法，

编译器就首先使用隐含类型转换把 X

转换成 Y，然后调用 Y 的 method。

查看Int api中的隐式函数

8 数据类型 纯粹面向对象的

val i:Int = 10;

Byte Short Int Long

Double Float

Char Boolean

以上类型为scala的基本数据类型，且为scala自定义类型

i,这些类型中的所有操作符都是方法，如 + - \* /

由此可以推导，scala中调用无参数的方法形式

10+20 即 10.+(20)

s.show("hello") 即 s show "hello"

ii,这些类型都隐式转化成了RichXXX,用来丰富功能

String 本身使用的就是java.lang.String,

会隐式转化成StringOps

统一类型

在Scala中，所有的值都有类型，包括数值和函数。下图阐述了类型层次结构的一个子集。

Any （顶级类型）

AnyVal AnyRef(java.lang.Object)

^ List

| Option

Double YourClass

Dloat ^

Long |

Int |

Short |

Byte |

Boolean |

Char Null

Unit ^

^ |

| |

Nothing（底部类型）

1Scala类型层级

Any是所有类型的超类型，也称为顶级类型。它定义了一些通用的方法如equals、hashCode和toString。Any有两个直接子类：AnyVal和AnyRef。

AnyVal代表值类型。有9个预定义的非空的值类型分别是：Double、Float、Long、Int、Short、Byte、Char、Unit和Boolean。Unit是不带任何意义的值类型，它仅有一个实例可以像这样声明：()。所有的函数必须有返回，所以说有时候Unit也是有用的返回类型。

AnyRef代表引用类型。所有非值类型都被定义为引用类型。在Scala中，每个用户自定义的类型都是AnyRef的子类型。如果Scala被应用在Java的运行环境中，AnyRef相当于java.lang.Object。

值类的空间是平坦的；所有的值类都是scala.AnyVal的子类型，但是它们不是其他类的子类。但是不同的值类类型之间可以隐式地互相转换。例如，需要的时候，类scala.Int的实例可以通过隐式转换放宽到类scala.Long的实例；Int支持min、max、until、to、abs等操作，其实是从类Int隐式转换到scala.runtime.RichInt的。

AnyRef是引用类型。所有非值类型都定义为引用类型.它其实是Java平台上java.lang.Object类的别名。因此Java里写的类和Scala里写的都继承自AnyRef。

Scala类与Java类的不同在于它们还继承自一个名为ScalaObject的特别记号特质。是想要通过ScalaObject包含的Scala编译器定义和实现的方法让Scala程序的执行更高效。

类型转化.

下面是一个示例，说明了字符串、整型、布尔值和函数都是对象，这一点和其他对象一样：

val list: List[Any] = List(

"a string",

732, // an integer

'c', // a character

true, // a boolean value

() => "an anonymous function returning a string"

)

list.foreach(element => println(element))

2类型转化

值类型可以按照下面的方向进行转换：

Byte->Short->Int->Long->Float->Double

^

|

Char

例1:

val x: Long = 987654321

val y: Float = x // 9.8765434E8 (note that some precision is lost in this case)

val face: Char = '☺'

val number: Int = face // 9786

例2:转换是单向，下面这样写将不会通过编译。

val x: Long = 987654321

val y: Float = x // 9.8765434E8

val z: Long = y // Does not conform

3Nothing和Null

scala.Null和scala.Nothing是用统一的方式处理Scala面向对象类型系统的某些"边界情况"的特殊类型。

Nothing是所有类型的子类型，也称为底层类型。没有一个值是Nothing类型的。它的用途之一是给出非正常终止的信号，如抛出异常、程序退出或者一个无限循环（可以理解为它是一个不对值进行定义的表达式的类型，或者是一个不能正常返回的方法）。

Null是所有引用类型的子类型（即AnyRef的任意子类型）。它有一个单例值由关键字null所定义。Null主要是使得Scala满足和其他JVM语言的互操作性，但是几乎不应该在Scala代码中使用。我们将在后面的章节中介绍null的替代方案。

例1:

def error(message: String): Nothing = throw new RuntimeException(message)

例2:

def divide(x: Int, y: Int): Int = if(y != 0) x / y else error("Can't divide by zero")

9 应用程序对象

object XXX{

def main(args:Array[String]){

}

}

object XXX extends App{

}

============================各种表达式=====================================

运算符

10 if表达式

if (boolean expression) value else value

i,scala是面向表达式的语言,所有东西都是表达式,并且有值

故 if表达式会返回值

ii,当if 值的类型 和 else 值的类型不一样时,注意接收结果

的类型

var s:Any = if(1 == 1) "hello" else 0

var s:Any = if(1 == 1) "hello" else new Student

11 复合表达式

i,复合表达式是用括号括起来的零个或多个表达式的序列。

ii,复合表达式的结果是代码块中的最后一行表达式

val average = {

val n1 = readLine("Enter a number: ").toDouble

val n2 = readLine

("Enter another number:").toDouble

(n1 + n2) / 2

}

iii,当一个方法只接受一个参数（除了对象）时，可以将该参数放在大括号内而不是小括号内

println {

var x = 2

while (x < 1000) x \*= 2

x

}

val in = new FileInputStream("src/test")

var len = 0

while({len = in.read();

len != -1}){

}

Array(100,200,300).map{x => println(x); println("hello")}

12 输入输出

Console.readLine

Console.println 过时

scala.io.StdIn.readLine

=============================for推导+模式匹配======================

13 for循环和for推导式

for(i <- e) E(i)

i,for(i <- e)称为发生器，E(i)称为函数体

ii,e 可以是各种集合，包括数组、列、表达式的等或有上下界的数据范围

1 to 10 (1 <= i <=10)

1 until 10 (1 < i <=10)

iii,守卫 即在for中添加if

for(i <- e)if() E(i)

for (i <- 1 to 10) if(i%2 != 0){

println(i)

}

for (i <- 1 to 10 by 2) if(i%2 != 0){

println(i)

}

for(i <- e;if()) E(i)

for (i <- 1 to 10 if(i%2 != 0)){

println(i)

}

可以添加多个守卫 分好隔开

for(i <- e;if();if()) E(i)

for (i <- 1 to 10;if(i%2 != 0);if(i%3!=0)){

println(i)

}

iv,嵌套循环

for(i <- e1)for(j <- e2)E(i,j)

for(i <- 1 to 5){

for(j <- 1 to i){

print("\* ")

}

println()

}

for(i <- e1;j <- e2)E(i,j)

for(i <- 1 to 5;j <- 5 to 10){

println(i+" "+j)

}

\* 可以使用任意多的表达式来获得循环中使用的变量

for(i <- 1 to 5;from = 10 - i;j <- 5 to i){

println(i+" "+j)

}

v,嵌套循环可以为每个发生器独立添加守卫

for(i <- e1;if A;j <- e2;if B)E(i,j)

for(i <- 1 to 5;if i%2 != 0;j <- 5 to 10;if j%2 != 0 ){

println(i+" "+j)

}

vi,搜集每次的循环的值，形成集合，带有yield的for循环称为for推导式

for(i <- e) yield E(i)

val n = for(i <- 1 to 10) yield i%3

vii,yield保留字返回的类型跟for循环体第一个 <- 语句的集合类型相同，如非集合时

返回的是序列

var arr:Array[Boolean] = for(i <- Array(100,200,300)) yield i==i

viii, zipWithIndex可以帮助我们获取元素的序号

var m = Array(20,30,40,50).zipWithIndex

for((x,y) <- m){

println(x+":"+y)

}

1.模式匹配

1.1 格式

express match{

case value1 => result1

case value2 => result2

case value3 => result3

........

}

1.2 match表达式类似于其他语言的switch语句，提供多个备选项中进行选择

a match{

case e1 => E1

case e2 => E2

....

}

eg:

val ch='+';

ch match{

case '-' => -1

case '+' => 1

}

若a匹配e1则执行E1,若a匹配e2则执行E2,以此类推

a可以是数组、任意类型值等，en可以是对应的值，常量，变量，甚至是类型

match表达式能用以直接赋值，如val sign=a match{case e1=> 123;case e2=> "123"}

匹配是从上而下的

1.3匹配语句case后接 \_ 代表的是任意，一般在最后的case语句中这么写，即匹配不到上面的值时，执行

如 a match {

case e1 =>...

case e2 =>...

case \_ =>....

}

匹配语句返回的值有多种类型时，Scala不能执行类型推断，并且只会返回Any类型

如 val final=a match {

case e1 => "HI"

case e2 => 'H'

case e3 => 123

}

eg:

val sign ="+";

sign match{

case "-" => "---------"

case "+" => "1"

case \_ => "is error"

}

1.4 case 可以用来匹配 常量 变量 类型 IndexedSeq Option case class等

1.匹配常量

val num=10;

val one=1;

（num>5） match{

case true => num

case false => 5

}

num match{

case one => 1

case 2 => 2

case \_ => 0

}

scala默认首字母大写的字符串为常量，首字母小写的字符串为变量，如果希望在模式匹配中用首字母小写的常量需要用反引号 ` 进行标注.

scala> val ch = 3.14

scala> val mypi = 3.14

scala> val res = ch match {

| case `mypi` => true

| case \_ => false

| }

res: Boolean = true

2.匹配变量

object match{

case x if x==null =>

//......

case x =>

}

3.类型匹配

eg:

val num:Any=32

num match{

case s:String => s.toInt

case x:Int => x

case \_ => 0

}

eg:

def getType(a:Any){

a match{

case \_ :Array[Char] => println("Array[Char]")

case \_ :Int => println("Int")

case \_ :Char => println("Char")

case \_ => println(“Error")

}

}

注意：泛型的类型匹配要注意如List[String]、Map[Char,Int]等不会成功匹配，如 List[Int]等亦不可匹配，因而往往使用通配符List[ \_ ]进行匹配，但Array[Int]是可行的

匹配发生在运行期，Java虚拟机中泛型的类型信息是被擦除的。因此，不能用类型匹配特定的Map类型。

case m:Map[String,Int] => //别这样做！

可以匹配一个通用的映射：

case m:Map[\_,\_] => //OK

但是对于数组而言，元素的类型信息是完好的。你可以匹配到Array[Int]

4.匹配数组，列表和元组

scala> def matchArr(x:Array[Int]):String =

| x match {

| case Array(0) => "Only 0" //仅含一个元素0的数字

| case Array(x,y) => x+" "+y //仅含有两个元素的数组

| case Array(0,\_\*) => "Start from 0" //0开始的数组

| case \_=> "Something else"

| }

// matchArr: (x: Array[Int])String

scala> matchArr(Array(0))

// res21: String = Only 0

scala> matchArr(Array(0 to 20).flatten)

// res23: String = Strat from 0

5.匹配Option

opt match{

case Some(info) => info....

case None => ....

}

6.匹配case class 样例类

class Person

case class Teacher(name:String,subject:String) extends Person

case class Student(name:String,classroom:String) extends Person

def greetingCaseClass(p:Persion)={

p match{

case Teacher(name:String,subject:String) => println(name+" "+subject);

case Student(name:String,classroom:String) => println(name+" "+classroom)

case \_ => println(".......")

}

}

1.5 匹配中加if守卫

express:Any match {

case x:Int if x%2==0 => x...

}

1.6 使用模式匹配类进行类型转换

在scala，我们倾向使用这样的模式匹配，而不是isInstanceOf()操作符。

eg:

Animal a=new Dog();

if(a.isInstanceOf[Cat]){

val c:Cat=a.asInstanceOf[Cat];

c.eat();

}

被模式匹配取代

a match{

case a:Cat => c.eat();

}

3.封闭类sealed class

样本类的超类被封闭（sealed），封闭类除类定义文件外不能添加子类.

模式匹配完成后需要确保所有情况皆被考虑,因此Scala编译器会检测match表达式所遗漏的模式组合

sealed abstract class Expr

case class Number( n :Int) extends Expr

case class Sum(e1 : Expr , e2 : Expr) extends Expr

case class Mul(e1 : Expr , e2 : Expr) extends Expr

如何定义存在可能样本遗漏的模式匹配

def getType(a:Expr):String =

a match{

case Number(n) => “Number“

case Sum(m,n) => “Sum“

}

warning : match is not exhaustive

case \_ =>

添加注解

def getType(a:Expr):String = (a: @unchecked) match {...}

4.偏函数

1.定义

被包在花括号内的一组case语句是一个偏函数--一个并非对所有输入值都有定义的函数。它是PartialFuncation[A,B]类的一个实例。(A是参数类型，B是返回类型)

Scala中的PartialFunction是一个Trait，其的类型为PartialFunction[A,B]，其中接收一个类型为A的参数，返回一个类型为B的结果。

偏函数和其它函数一样，也定义了apply方法，apply方法会从匹配到的模式计算函数值。该特质有1个方法抽象方法：def isDefinedAt(a: A):Boolean，isDefinedAt方法决定了该方法的参数是否在给定的偏函数的定义域内，如果返回结果为true，表示在，否则不在。

例如：

scala> val pf:PartialFunction[Int,String] = {

| case 1=>"One"

| case 2=>"Two"

| case 3=>"Three"

| case \_=>"Other"

| }

pf: PartialFunction[Int,String] = <function1>

scala> pf(1)

res0: String = One

scala> pf(2)

res1: String = Two

scala> pf(3)

res2: String = Three

scala> pf(4)

res3: String = Other

偏函数内部有一些方法，比如isDefinedAt、OrElse、 andThen、applyOrElse等等。

1.isDefinedAt : 这个函数的作用是判断传入来的参数是否在这个偏函数所处理的范围内。

刚才定义的pf来尝试使用isDefinedAt()，只要是Int类型都是正确的，因为有case \_=> "Other"这一句。如果换成其他类型就会报错。

如果将case \_=> "Other"这一行去掉，执行pf(4)则会抛出MatchError异常

2.orElse : 将多个偏函数组合起来使用，效果类似case语句。

scala> val onePF:PartialFunction[Int,String] = {

| case 1=>"One"

| }

onePF: PartialFunction[Int,String] = <function1>

scala> val twoPF:PartialFunction[Int,String] = {

| case 2=>"Two"

| }

twoPF: PartialFunction[Int,String] = <function1>

scala> val threePF:PartialFunction[Int,String] = {

| case 3=>"Three"

| }

threePF: PartialFunction[Int,String] = <function1>

scala> val otherPF:PartialFunction[Int,String] = {

| case \_=>"Other"

| }

otherPF: PartialFunction[Int,String] = <function1>

scala> val newPF = onePF orElse twoPF orElse threePF orElse otherPF

newPF: PartialFunction[Int,String] = <function1>

scala> newPF(1)

res0: String = One

scala> newPF(2)

res1: String = Two

scala> newPF(3)

res2: String = Three

scala> newPF(4)

res3: String = Other

这样，newPF跟原先的pf效果是一样的。

3.andThen: 相当于方法的连续调用，比如g(f(x))。

scala> val pf1:PartialFunction[Int,String] = {

| case i if i == 1 => "One"

| }

pf1: PartialFunction[Int,String] = <function1>

scala> val pf2:PartialFunction[String,String] = {

| case str if str eq "One" => "The num is 1"

| }

pf2: PartialFunction[String,String] = <function1>

scala> val num = pf1 andThen pf2

num: PartialFunction[Int,String] = <function1>

scala> num(1)

res4: String = The num is 1

pf1的结果返回类型必须和pf2的参数传入类型必须一致，否则会报错。

4.applyOrElse：它接收2个参数，第一个是调用的参数，第二个是个回调函数。如果第一个调用的参数匹配，返回匹配的值，否则调用回调函数。

scala> onePF.applyOrElse(1,{num:Int=>"two"})

res5: String = One

scala> onePF.applyOrElse(2,{num:Int=>"two"})

res6: String = two

在这个例子中，第一次onePF匹配了1成功则返回的是"One"字符串。第二次onePF匹配2失败则触发回调函数，返回的是"Two"字符串。

===============================插值器=============================

2.10后推出 有 s f raw

s 字符串插值器

在任何字符串前加上s，就可以直接在串中使用变量了

val name="James"

println(s"Hello,$name")

f 插值器

在任何字符串字面前加上 f，就可以生成简单的格式化串，功能相似于其他语言中的 printf 函数。

val height=1.9d

val name="James"

println(f"$name%s is $height%2.2f meters tall")

raw 插值器

除了对字面值中的字符不做编码外，raw 插值器与 s 插值器在功能上是相同的。

scala>s"a\nb"

res0:String=

a

b 这里，s 插值器用回车代替了\n。而raw插值器却不会如此处理。

scala>raw"a\nb"

res1:String=a\nb 当不想输入\n被转换为回车的时候，raw 插值器是非常实用的

===============================命名参数和可变参数=============================

14,函数的命名参数和可变参数

def number(hundreds: Int, tens: Int, ones: Int) = 100 \* hundreds + 10 \* tens + ones

i,正常调用number函数

number(2, 3, 5)

ii,带有命名参数的调用，参数顺序可以变化

number(tens = 3, hundreds = 2, ones = 5)

iii,部分带有命名参数的调用，要求没有使用命名参数的位置类型必须对应，使用

命名参数的可以变换顺序number

number(2, ones = 5, tens = 3)

iv,scala可以使用可变参数，同java一样可变参数放在参数列表最后一位且只能有一个

def show(str:String,strs:String\*)={

println("first word:"+str)

strs.foreach(println \_) //使用可变参数当做数组来使用

}

v,调用可变参数方法时 枚举传参 或者 集合:\_\* 传参

show("hello","world","briup")

var arr = Array("world","briup")

show("hello",arr:\_\*)

===============================异常处理=====================================

15,异常处理

i,scala里面没有检查型异常，即必须要处理的异常

ii,scala函数 可以使用throw 抛出异常

def show(msg:String)={

if(msg == ""){

throw new Exception("msg is empty")

}

println(msg)

}

iii,scala 可以用try catch finaly捕获异常，catch对于异常的

捕获使用的是模式匹配

try {

show("")

}catch {

case e1:NullPointerException => e1.printStackTrace()

case e2:Exception => e2.printStackTrace()

case \_ => println("特殊异常") //利用\_匹配不是e1 e2类型的异常

}finally {

println("默认执行")

}

iv,如果方法有返回值，按照处理异常的含义，即使发生异常，也要把值进行返回

def readFromFile(file:String):List[String]={

try{

Source.fromFile(file).getLines().toList

}catch {

case e:Exception => List()

}

}

iv,利用Option None Some 处理异常

Option[A] 是一个可能有值也可能没值的容器

def readFromFileOption(file:String):Option[List[String]]={

try{

Some(Source.fromFile(file).getLines().toList)

}catch {

case e:Exception => None

}

}

Option有两个子类别，Some和None。当程序回传Some的时候，代表这个函式成功地给了你一个List，而你可以透过get()函数拿到那个List，如果程序返回的是None，则代表没有List可以给你。

所以Option存在的意义, 就是为了在代码中注明, 让大家一看就知道: "这个东西可能是空的! 你们用的时候给我小心点" 这样的暗示.

有了这个暗示, 你可就不能随意取出option里面的东西了, 警醒你每次使用, 都要先判断. isEmpty 或是 nonEmpty

v,利用Try Success Failure处理异常

Try[A] 则表示一种计算： 这种计算在成功的情况下，返回类型为 A 的值，在出错的情况下，返回 Throwable

def readFromFileTry(file:String):Try[List[String]]={

Try(Source.fromFile(file).getLines().toList)

}

var result = readFromFileTry("src/test.txt")

result match {

case n:Success[List[\_]] => n.get.foreach(println(\_));

case m:Failure[List[\_]] => println("error");

}

vi,为了保证和java相互操作 建议在会throw 异常的scala函数上用注解标识，

那么java代码在调用此函数时，会提示要处理异常

@throws(classOf[Exception])

============集合/数组/元组(集合/数组会使用即可)=====================

数组

1.数组:存放一系列元素的容器

1.分类:

不可变数组 定长数组 Array

可变数组 缓冲数组 数组缓存 ArrayBuffer (集合,Seq)

ArrayBuffer位于scala.collection.mutable包下。

2.Array数组

2.1定义方式

1.通过类构建

eg:

val arr=new Array[T](size);

2.通过对象构建(统一对象构建原则)

eg:

val arr2=Array("hello","scala","java");

val arr3=Array.apply(1,2,3,4);

3.其他

Array.range

Array.empty

2.2 取值

arr(index)

本质上调用的arr.apply(index)这个方法

arr.take(num)

arr.takeRight(num)

arr.takeWhile(pf:T=>Boolean)

2.3 赋值

arr(index)=值

arr.update(index,值)

2.4 遍历数组

for(elem <- arr){...}

for(index <- 0 until arr.length){...}

arr.foreach(println)

2.5 获取数组的长度

arr.length

arr.size

3.ArrayBuffer数组缓冲

2.1定义方式

1.通过类构建

eg:

import scala.collection.mutable.ArrayBuffer

val arrBuffer=new ArrayBuffer[T]();

2.通过对象构建(统一对象构建原则)

eg:

val arrBuffer=ArrayBuffer(1,2,3)

2.2添加元素

+: ++ ++: +=: ++=: append appendAll insert insertAll /:

需要注意的是：

1. 当方法名中有一个+号时，指的是添加 一个元素，返回一个新的集合/数组

2. 当方法名中有两个+号时，指的是添加 一个集合/数组容器，返回一个新的集合/数组

3. 当方法名中出现=号时，指的是会修改原集合。（只有可变集合才有包含=的方法）

4. 当方法名中没有=号时，不会修改原集合/数组，一般只会返回一个新的集合/数组

5. 当:+方法用于在尾部追加元素，+:方法用于在头部追加元素

2.3移除元素

- -- -= --= remove(index) remove(index,count) drop(count) dropRight(count) dropWhile(pf:T=>Boolean)

2.4常见方法

take takeRight takeWhile count

算数集合：sum product max min

排序：

sorted 按照集合类型默认排序规则进行排序（默认升序）

sortBy 按照自定义指定规则进行排序

sortWith自定义升序还是降序排列

遍历输出：foreach

转换：map filter

val result=for(elem <- arr if elem %2==0)yield elem\*2

val newArr=arr.filter(\_%2==0).map(\_\*2)

val newArr=

arr.filter(

(x:Int) => {x%2==0}

).map(

(x:Int) => { x\*2 }

)

元组: Tuple1 - Tuple22

与数组一样，元组也是不可变的，但与数组不同的是元组可以包含不同类型的元素。

元组的实际类型取决于它的元素的类型，比如 (99, "runoob") 是 Tuple2[Int, String]。 ('u', 'r', "the", 1, 4, "me") 为 Tuple6[Char, Char, String, Int, Int, String]。

目前 Scala 支持的元组最大长度为 22。对于更大长度你可以使用集合，或者扩展元组。

1 若干个单个的值包含在圆括号便构成元组：

eg:

val t = new Tuple3(1 , 1.2,'A')

val g=(1 , 1.2,'A') 三元 元组 //(Int,Double,Char)类型的元组

var t1 = 10 -> 20;

2 利用方法\_1、\_2、\_3访问元组的组元

val h=g.\_1 或 val h=g \_1

或者利用隐式的模式匹配

val (first,second,three)=(1,3.14,"Free");

val (first,second,\_)=(1,3.14,"hhhhhhhhhhhh")

val (name,age,phone,address)=("tom",23,110,"南昌")

3 元组可以用于函数需要返回不止一个值得情况。

举例来说，StringOps的partition方法返回的是一对字符串，分别包含了满足某个条件和不满足条件的字符："New York".partition(\_.isUpper)

5 注意区分下边两个的不同

val x,y,z=(1,"hello",2)

val (x,y,z)=(1,"hello",2)

===================集合 整体类的结构===============

同一类型：同一个父类

1.集合

1.集合分为： 序列 集 映射

Traversable(Trait)

|

Iterable(Trait)

———————————————————————————-

| | |

Seq （序列，类似list） Set Map(Trait/object)

有序，可重复 ，单值 无序，不重复，单值 k和set一样， 双值

2.Seq 是一个有先后次序的值得序列,允许存放重复元素。

2.1整体上分为：索引序列IndexedSeq,线性序列(链表)LinearSeq.

Seq

———————————————————————————————————————————————-

| | | 只能从前往后找，无下标

IndexedSeq（索引序列，带下标） Buffer LinearSeq （线性序列）

| | |

Array Vector Range | List LinkedList

String StringBulid ArrayBuffer Queue Stack Stream View

ListBuffer

IndexSeq索引序列：允许我们通过整型的下标快速访问任意元素，如ArrayBuffer是带下标的。

LinearSeq线性序列：被分为了头尾部分，并且用head，tail和isEmpty方法等。

注意：Array其实不是真正的序列，是通过将Array包装成WrappedArray(mutable)，才可以像集合一样使用。

arr.

buffer.

3.Set是一组没有重复元素的集合。

Set

———————————————————————————

| | | |

（BitSet位集） HashSet ListSet SortedSet

快，运算大数集 |

TreeSet

在SortedSet中，元素以某种排过序的顺序被访问。

4.Map是一组（K,V）对偶，其中键必须是唯一的。

Map

—————————————————————————————————

| | | |

HashMap LinkedListMap ListMap SortedMap

|

TreeMap

SortedMap按照键的排序访问。

5.每个Scala集合特质或类，都有一个带有apply方法的伴生对象，这个apply方法可以用来构建该集合中的实例。

eg:

Iterable(0xFF, 0xFF00, 0xFF0000)

Seq(color.RED, color.GREEN, Color.BLUE)

Map(color.RED -> -0xFF0000, Color.GREEN -> 0xFF00, Color.BLUE -> 0xFF)

SortedSet("Hello" , "World")

2.Seq的一些具体实现类

2.1序列IndexSeq （集合是可存放不同类型的）

Vector是ArrayBuffer的不可变版本，一个带下标的序列，支持快捷的随机访问，以树形结构的形式实现。

Range表示一个整数序列，只存储 起始值，结束值和增值（步长）， 用 to 和 until 方法来构造Range对象。

2.2列表 LinearSeq

列表要么是Nil(空表)，要么是一个head元素和一个tail，tail又是一个列表。

val digits = List(4,2)

digits.head //4

digits.tail //List(2)

digits.tail.head // 2

digits.tail.tail //Nil

:: 操作符从给定的头和尾创建一个新的列表。

9 :: List(4,2) // List(9,4,2)

9 :: 4 :: 2 :: Nil // :: 是右结合，列表从末端开始构建

9 :: ( 4 :: (2 :: Nil ) )

求和，除了遍历外，可以用 递归 模式匹配

def sum(lst : List[Int]): Int =

if( lst == Nil) 0 else lst.head + sum(lst.tail)

def sum(lst:List[Int]): Int = lst match{

case Nil => 0

case h :: t => h+sum(t) // h 是 lst.head， 而t是lst.tail, ::将列表"析构"成头部和尾部

}

直接使用List的方法

List(9,4,2).sum

2.3可变列表LinkedList

LinkedList, elem指向当前值，next指向下一个元素

DoubleLinkedList多带一个prev

例1:将所有负值改为0

val lst = scala.collection.mutable.LinkedList(1,-2,7,-9)

var cur = lst

while(cur != Nil){

if(cur.elem<0) cur.elem = 0

cur = cur.next

} // (1,0,7,0)

例2: 去除每两个元素中的一个

var cur = lst

while(cur != Nil && cur.next != Nil){

cur.next = cur.next.next

cur = cur.next

}

注：当要把某个节点变为列表中的最后一个节点，不能讲next 设为Nil 或 null, 而将它设为LinkedList.empty。

3.Set的一些具体实现类

3.1 HashSet 不重复元素的集合，以哈希集实现，元素根据hashCode方法的值进行组织

Set(2,0,1) + 1 // (2,0,1)

HashSet(2,3,0)

3.2 LinkedHashSet，链式哈希集 记住元素被插入的顺序

val weekdays = scala.collection.mutable.LinkedHashSet(1,2,3,4)

3.3 排序的集

scala.collection.immutable.SortedSet(1,2,3,4)

// 用红黑树实现的

Treeset 只能排scala体系内的类型数据（基本数据类型），引用数据类型不能

3.4 位集(BitSet), 以一个字位序列的方式存放非负整数，如果集中有i，则第i个字位是1

高效的实现，只要最大元素不是特别大。

位集合是由单字或多字的紧凑位实现的非负整数的集合。其内部使用Long型数组来表示。第一个Long元素表示的范围为0到63，第二个范围为64到127，以此类推（值为0到127的非可变位集合通过直接将值存储到第一个或第两个Long字段的方式，优化掉了数组处理的消耗）。对于每个Long，如果有相应的值包含于集合中则它对应的位设置为1，否则该位为0。这里遵循的规律是，位集合的大小取决于存储在该集合的最大整数的值的大小。假如N是为集合所要表示的最大整数，则集合的大小就是N/64个长整形字，或者N/8个字节，再加上少量额外的状态信息字节。

因此当位集合包含的元素值都比较小时，它比其他的集合类型更紧凑。位集合的另一个优点是它的contains方法（成员测试）、+=运算（添加元素）、-=运算（删除元素）都非常的高效。

BitSet代表一个由小整数构成的容器，这些小整数的值表示了一个大整数被置1的各个位。比如说，一个包含3、2和0的bit集合可以用来表示二进制数1101和十进制数13. (可以通过bit.toBitMask来测试) 1110

BitSet内部的使用了一个64位long型的数组。数组中的第一个long表示整数0到63，第二个表示64到27，以此类推。所以只要集合中最大的整数在千以内BitSet的压缩率都是相当高的。

BitSet操作的运行时间是非常快的。查找测试仅仅需要固定时间。向集合内增加一个项所需时间同BitSet数组中long型的个数成正比，但这也通常是个非常小的值。这里有几个关于BitSet用法的例子：

scala> val bits = scala.collection.immutable.BitSet.empty

bits: scala.collection.immutable.BitSet = BitSet()

scala> val moreBits = bits + 3 + 4 + 4

moreBits: scala.collection.immutable.BitSet = BitSet(3, 4)

scala> moreBits(3)

res26: Boolean = true

scala> moreBits(0)

res27: Boolean = false

Scala提供 可变和不可变的两个 BitSet类

contains 检查是否包含， subsetOf 检查集的所有元素是否被另一个集包含

val digits = Set(1,7,2,9)

digits contains 0 // false

Set(1,2) subsetOf digits // true

3.5 集的操作

合集 union | ++

交集 intersect &

差集 diff &~ --

推荐使用: ++ & --

4.往集合中添加移除元素推荐操作：

4.1 一般而言，+用于将元素添加到无先后次序的集合，而+:和:+则是将元素添加到有先后次序的集合的开头或末尾。

Vector(1,2,3) :+ 5 //Vector(1,2,3,5)

1 +: Vector(1,2,3) //Vector(1,1,2,3)

4.2 以冒号结尾的操作符，+:是右结合的，这些操作符都返回新的集合

4.3 可变集合有 +=操作符 用于修改左侧操作元

val numbers = ArrayBuffer(1,2,3)

numbers += 5 // 将 5 添加到 numbers

4.4 不可变集合，可以在var上使用+=或:+=

var numbers = Set(1,2,3)

numbers += 5 // numbers 设为不可变的集numbers + 5

var numberVector = Vector(1,2,3)

numbersVector :+= 5 // 向量没有+操作符，只有:+

思考: mutable.LinkedList中为什么没有带=号的方法？？？？

5.如何选择一个集合

1.根据集合的特点选择其中一种集合。

2.想要可变的还是不可变的集合。

如何选择Seq集合中的具体类型:

推荐可以优先采用下边的集合。

通用的序列集合：

不可变 可变

索引： Vector ArrayBuffer

线性链表: List ListBuffer

不可变序列集合：

索引 线性 描述

List 对 单链表

Queue 对 先进先出的数据结构

Range 对 整数值范围

Stack 对 后进先出

Stream 对 与链表相似，但是延迟并且持久。适用于大型或无限序列

String 对 不可变的，索引字符序列

Vector 对 split和join非常有效率的实现

可变序列集合：

索引 线性 描述

Array 对 元素是可变的，但集合长度不可变

ArrayBuffer 对 元素可变，集合长度可变

ArrayStack 对 后进先出数据结构。

DoubleLinkedList 对 单链表，但是有一个prev前置指向

LinkedList 对 可变的单链表

ListBuffer 对 像ArrayBuffer,但依靠链表

Queue 对 先进先出

Stack 对 后进先出

StringBuilder 对

如何选择Map集合中的具体类型:

HashMap LinkedHashMap ListMap Map SortedMap TreeMap

如何选择Set集合中的具体类型:

HashSet LinkedHashSet ListSet TreeSet Set SortedSet BitSet

6.别的集合类（表现像集合一样的类型）

Enumeration

Iterator

Option 包含一个或零个元素的集合。

Tuple 元组类 Tuple1 到 Tuple22

7.集合类中的通用方法：

1.过滤方法

collect drop dropWhile filter filterNot find foldLeft foldRight head headOption init last lastOption reduceLeft reduceRight remove slice tail take takeWhile union diff intersect distinct等

2.转化方法

+ ++ - — diff distinct collect flatMap map reverse sortWith takeWhile zip zipWithIndex zipAll等

以及一系列的to\*\*\*\*方法，将当前集合转化成其他集合类型（Array,Buffer,Vector等）

3.分组方法

groupBy partition sliding span splitAt unzip unzip3 等

4.信息和数学方法

canEqual contains containsSlice count endsWith exists find forAll hasDefiniteSize indexOf indexOfSlice indexWhere max min nonEmpty product segmentLength size startsWith sum 等

5.其他

par view flatten foreach mkstring

6.化简 折叠 扫描

6.1

reduce

reduceLeft

reduceRight

6.2

fold

foldLeft /:

foldRight :\

6.3 scanLeft,scanRight， 得到包含所有中间结果的集合

scan

scanLeft

scalRight

作业：获取一个字符串中每个字符出现的频次。

1.for现实

2.折叠实现

val freq = scala.collection.mutable.Map[Char, Int]() // 可变映射

for( c <- "Mississippi"){

freq(c) =freq.getOrElse(c,0)+1 // Map('i' ->4, 'M' -> 1, 's' -> 4, 'p' ->2)

}

val map=(Map[Char, Int]() /: "Mississippi"){(m,c) => m + (c -> (m.getOrElse(c,0) +1))}

映射: Map(Map集合中每个元素是一个二元元组)

Map 也叫哈希表（Hash tables）。

1 二元元组的表示方法：

(key,value) 或 key -> value

2 分为可变映射和不可变映射

scala的集合系统的区分了可变（ mutable ）和不可变（immutable ）集合。一个mutable 集合能够更新甚至扩展空间，这意味着你能改变，增加，或者删除一个集合的元素。 一个immutable集合，刚好相反，不能改变。你仍然可以做一些类似的增加，删除，或者更新，但是实际上他返回了一个新的对象，这里面就是指返回了一个新的集合，而老的集合没有改变。

mutable.Map[K,V]

immutable.Map[K,V]

Map <==> immutable.Map <==> Predef.Map

注意：

scala.collection.Map 是immutable.Map和mutable.Map的超类

Scala优先采用不可变集合， scala.collection 包中的伴生对象产出不可变的集合

3 构建Map映射对象

Map 是一个trait 和 object

因此构建方式只有：统一对象构建原则

Map(elem1,elem2,elem2,...)

<===>

Map.apply(elem1,elem2,elem2,...)

4 构建一个空集合,可以使用empty方法

import scala.collection.mutable;

val mutableMap=mutable.Map.empty[K,V]

var A:Map[Char,Int] = Map()

5 通过key获取value值

三种方式:

map.apply(key)

map.get(key)

map.getOrElse(key,defaultValue)

6 添加元素 + ++ ++: (+= ++=) insert insertAll append appendAll

7 移除元素 - -- (-= --=) remove drop dropRight

8 遍历集合

eg:

for(elem <- map){

val key=elem.\_1

val value=elem.\_2

}

或：

for( (key,value) <- map ){

println(key+":"+value)

}

只遍历key值

map.keys

map.keySet

map.keysIterator

只遍历value值

map.values

map.valuesIterator

9 拉链操作

zip 将两个集合进行"等值连接"

zipAll 将两个集合进行"全连接",三个参数,第一个参数为连接的集合；第二个参数为原集合元素不足时的补位元素；第三个参数为连接集合元素不足时的补位元素；

zipWithIndex 将集合中的每个元素变成一个二元元组，二元元组的\_2即位当前元素在集合中的索引。

unzip 将容器中的二元元组拆分,将每个二元元组中的\_1放到一个集合中,\_2的放到一个集合中。即拆分成两个集合。

unzip3 将容器中的三元元组拆分,将每个三元元组中的\_1放到一个集合中,\_2的放到一个集合中,\_3的放到一个集合中。即拆分成了三个集合。

eg:

val price=List(2,10,8)

val num=List(10,10,10)

val collection=list1.zip(list2)

val newColl=for( (price,num) <- collection )yield{

price\*num

}.sum

val count=collection.map(x=> x.\_1\*x.\_2).sum

8.Iterator

相对于集合而言是一个"懒"的替代品，只有在需要时才去取元素，如果不需要更多元素，不会付出计算剩余元素的代价

对于那些完整构造需要很大开销的集合，适合用迭代器

如Source.fromFile产出一个迭代器，因为整个文件加载进内存不高效。

迭代器的两种用法

while(iter.hasNext) iter.next()

for(elem <- iter) 对elem操作

上述两种循环都会讲迭代器移动到集合末端，不能再被使用，

调用 map filter take等转换方法，返回值为集合，因此指针不发生变化。

调用 count sum length find方法后，返回值为单个值 迭代器会位于集合的末端，不能使用

9.Stream

9.1 迭代器每次调用next都会改变指向,如果要缓存之前的值，可以使用流

9.2 流是一个尾部被懒计算的不可变列表，也就是说只有需要时才计算

def numsForm(n:BigInt) : Stream[BigInt] = n #:: numsForm(n+1) // #:: 操作符 构建出来的是一个流

var tenOrMore = numsForm(10) // Stream(10,?), 其尾部是未被求值得

tenOrMore.tail.tail.tail // Stream(13,?)

val squares = numsForm(1).map{ x=> x\*x) // Stream(1,?)

9.3 使用force去流强制求值

squares.take(5).force // Stream(1,4,9,16,25)

squares.force // 会尝试对一个无穷流的所有成员求值，最后 OutOfMemoryError

9.4 迭代器可以用来构造一个流 通过toStream方法

Source.fromFile("").getLines返回一个Iterator[String]，用这个迭代器，对于每一行只能访问一次，而流将缓存访问过的行，允许重新访问

val words = Sourcce.fromFile("/usr/share/dict/words").getLines.toStream

words // Stream(A, ?)

words(5) // Aachen

words // Stream(A, A'o, AOL, AOL's, Aachen, ?)

10.View

10.1 类似流的懒理念

10.2 与流的不同

1、连第一个元素都不会求值

2、不会缓存求过的值

10.3 懒试图的好处:可以避免在多种变换下产生的中间集合

(0 to 1000).map(pow(10,\_)).map(1/\_) //先第一个map,再第二个map, 构 建了一个中间集合

(0 to 1000).view.map(pow(10,\_)).map(1/\_).force // 记住两个map操作 每个元素被两个操作同时执行，不需要额外构中间集合

11.并行集合 par

11.1 为了更好利用计算机的多个处理器，支持并发通常是必需的如果coll是个大型集合，那么

coll.par.sum //并发求和，par方法产出当前集合的一个并行实现，该实 现会尽可能地并行执行集合方法

coll.par.count(\_ % 2 ==0) //计算偶数的数量

11.2 对数组、缓冲、哈希表、平衡树而言，并行实现会直接重用底层实际集合的实现，所以很高效

11.3 可以通过对要遍历的集合应用.par并行化for循环

for( i <- (0 until 100).par) print( i + " " ) //数字是按照作用于 该任务的线程产出的顺序输出

在for/yield循环中，结果是依次组装的

for( i <- (0 until 100).par) yield i +" "

11.4 par返回的并行集合扩展自ParSeq ParSet Parmap，都是ParIterable的子类 型，不是Iterable的子类型，所以不能将并行集合传递给预期Iterable Seq Set Map的方法。

11.5 可以用to方法将并行集合转换回串行的版本。

eg:

eg:查看打印的数字顺序

(0 until 10).par.foreach(println)

(0 until 10).foreach(println)

以下代码获取到参与并行计算的线程:

(0 to 10000).par.collect{case \_ => Thread.currentThread.getName}.distinct

(0 to 10000).collect{case \_ => Thread.currentThread.getName}.distinct

12.与Java集合互调用

12.1 借助于scala.collection.JavaConverters对象中的静态方法。

asScala

asJava

12.2 Java2Scala集合举例:Java中的集合只能转化成Scala中可变集合

import scala.collection.JavaConverters.\_;

//1.定义一个java的集合对象

val list=new util.ArrayList[String]();

list.add("java");

list.add("scala");

println(list);

//2.遍历集合对象

list.forEach(new Consumer[String] {

override def accept(t: String): Unit = {

println(t);

}

})

//3.转化成Scala集合

val list\_s:mutable.Buffer[String]=list.asScala;

//4.遍历Scala集合

list\_s.foreach(println \_)

val list\_s2:mutable.Iterable[String]=list.asScala;

val list\_s3:Seq[String]=list.asScala;

list\_s2.foreach(println \_)

list\_s3.foreach(println \_)

//5.其他方法

val a1=asScalaBuffer(list);

val a2=asScalaIterator(list.iterator());

12.3 Scala2Java集合举例:

import scala.collection.JavaConverters.\_

//1.定义一个Scala数组

val a1=new Array[Int](3);

val a2=(1 to 10 ).toArray;

//a2.asJava;

//2.定义一个Scala缓冲

val buffer=a2.toBuffer;ArrayBuffer

val buffer\_j=buffer.asJava;

buffer\_j.forEach(new Consumer[Int] {

override def accept(t: Int): Unit = {

println("java:"+t)

}

})

val source = new scala.collection.mutable.ListBuffer[Int]

val target: java.util.List[Int] = source.asJava

val other: scala.collection.mutable.Buffer[Int] = target.asScala

12.3 Java调用Scala的方法时，如果参数为可变参时，不能直接调用，需要我们在scala中定义方法时，添加注解@varargs

@varargs

def varargs( name:String\*)={

name.foreach(println)

}

====================函数==============================

Scala 有方法与函数，二者在语义上的区别很小。Scala 方法是类的一部分，而函数是一个对象可以赋值给一个变量。换句话来说在类中定义的函数即是方法。

Scala 中的方法跟 Java 的类似，方法是组成类的一部分。

Scala 中的函数则是一个完整的对象，Scala 中的函数其实就是继承了 Trait 的类的对象。

Scala 中使用 val var 语句可以定义函数，def 语句定义方法。

函数和方法的区别

1、函数可作为一个参数传入到方法中，而方法不行。

2、在Scala中无法直接操作方法，如果要操作方法，必须先将其转换成函数。有两种方法可以将方法转换成函数：

val f1 = m \_

在方法名称m后面紧跟一个空格和下划线告诉编译器将方法m转换成函数，而不是要调用这个方法。 也可以显示地告诉编译器需要将方法转换成函数：

val f1: (Int) => Int = m

通常情况下编译器会自动将方法转换成函数，例如在一个应该传入函数参数的地方传入了一个方法，编译器会自动将传入的方法转换成函数。

3、函数必须要有参数列表，而方法可以没有参数列表

匿名函数

scala> val sayHello = (name: String) => println("my name is:" + name)

sayHello: String => Unit = <function1>

scala> sayHello("cyony")

my name is:cyony

sayHello变量就是一个函数，它没有自己的函数名，只定义了函数签名，以及函数体，返回类型为Unit，可以看到编译器自动为这个匿名函数取名为function1。在实际调用时，直接调用这个函数变量，传入一个String类型的值，即可。

函数传参

1.Scala的解释器在解析函数参数(function arguments)时有两种方式：

传值调用（call-by-value）：先计算参数表达式的值，再应用到函数内部；

传名调用（call-by-name）：将未计算的参数表达式直接应用到函数内部

object Test {

def main(args: Array[String]) {

delayed(time());

}

def time() = {

println("获取时间，单位为纳秒")

System.nanoTime

}

def delayed( t: => Long ) = {

println("在 delayed 方法内")

println("参数： " + t)

t

}

}

以上实例中我们声明了 delayed 方法， 该方法在变量名和变量类型使用 => 符号来设置传名调用

===================高阶函数==================

高阶函数（Higher-Order Function）就是操作其他函数的函数。

高阶函数是指使用其他函数作为参数、或者返回一个函数作为结果的函数。在Scala中函数是"一等公民"，所以允许定义高阶函数。这里的术语可能有点让人困惑，我们约定，使用函数值作为参数，或者返回值为函数值的"函数"和"方法4"，均称之为"高阶函数"。

Scala 中允许使用高阶函数, 高阶函数可以使用其他函数作为参数，或者使用函数作为输出结果。

以下实例中，apply() 函数使用了另外一个函数 f 和 值 v 作为参数，而函数 f 又调用了参数 v：

object Test {

def main(args: Array[String]) {

println( apply( layout, 10) )

}

// 函数 f 和 值 v 作为参数，而函数 f 又调用了参数 v

def apply(f: Int => String, v: Int) = f(v)

def layout[A](x: A) = "[" + x.toString() + "]"

}

最常见的一个例子是Scala集合类（collections）的高阶函数map

val salaries = Seq(20000, 70000, 40000)

val doubleSalary = (x: Int) => x \* 2

val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)

函数doubleSalary有一个整型参数x，返回x \* 2。一般来说，在=>左边的元组是函数的参数列表，而右边表达式的值则为函数的返回值。在第3行，函数doubleSalary被应用在列表salaries中的每一个元素。

为了简化压缩代码，我们可以使用匿名函数，直接作为参数传递给map:

val salaries = Seq(20000, 70000, 40000)

val newSalaries = salaries.map(x => x \* 2) // List(40000, 140000, 80000)

注意在上述示例中x没有被显式声明为Int类型，这是因为编译器能够根据map函数期望的类型推断出x的类型。对于上述代码，一种更惯用的写法为：

val salaries = Seq(20000, 70000, 40000)

val newSalaries = salaries.map(\_ \* 2)

既然Scala编译器已经知道了参数的类型（一个单独的Int），可以只给出函数的右半部分，不过需要使用\_代替参数名（在上一个例子中是x）

1.强制转换方法为函数

可以传入一个对象方法作为高阶函数的参数，这是因为Scala编译器会将方法强制转换为一个函数。

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

private def convertCtoF(temp: Double) = temp \* 1.8 + 32

def forecastInFahrenheit: Seq[Double] = temperatures.map(convertCtoF) // <-- passing the method convertCtoF

}

在这个例子中，方法convertCtoF被传入forecastInFahrenheit。这是可以的，因为编译器强制将方法convertCtoF转成了函数x => convertCtoF(x) （注: x是编译器生成的变量名，保证在其作用域是唯一的）。

2.接收函数作为参数的函数

使用高阶函数的一个原因是减少冗余的代码。比方说需要写几个方法以通过不同方式来提升员工工资，若不使用高阶函数，代码可能像这样：

object SalaryRaiser {

def smallPromotion(salaries: List[Double]): List[Double] =

salaries.map(salary => salary \* 1.1)

def greatPromotion(salaries: List[Double]): List[Double] =

salaries.map(salary => salary \* math.log(salary))

def hugePromotion(salaries: List[Double]): List[Double] =

salaries.map(salary => salary \* salary)

}

注意这三个方法的差异仅仅是提升的比例不同，为了简化代码，其实可以把重复的代码提到一个高阶函数中：

object SalaryRaiser {

private def promotion(salaries: List[Double], promotionFunction: Double => Double): List[Double] =

salaries.map(promotionFunction)

def smallPromotion(salaries: List[Double]): List[Double] =

promotion(salaries, salary => salary \* 1.1)

def bigPromotion(salaries: List[Double]): List[Double] =

promotion(salaries, salary => salary \* math.log(salary))

def hugePromotion(salaries: List[Double]): List[Double] =

promotion(salaries, salary => salary \* salary)

}

新的方法promotion有两个参数，薪资列表和一个类型为Double => Double的函数（参数和返回值类型均为Double），返回薪资提升的结果。

3.返回函数的函数

有一些情况你希望生成一个函数， 比如：

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

注意urlBuilder的返回类型是(String, String) => String，这意味着返回的匿名函数有两个String参数，返回一个String。在这个例子中，返回的匿名函数是(endpoint: String, query: String) => s"https://www.example.com/$endpoint?$query"。

9.嵌套方法

在Scala中可以嵌套定义方法。例如以下对象提供了一个factorial方法来计算给定数值的阶乘：

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))

程序的输出为:

Factorial of 2: 2

Factorial of 3: 6

10.多参数列表（柯里化）

方法可以定义多个参数列表，当使用较少的参数列表调用多参数列表的方法时，会产生一个新的函数，该函数接收剩余的参数列表作为其参数。这被称为柯里化。

下面是一个例子，在Scala集合中定义的特质Traversable：

def foldLeft[B](z: B)(op: (B, A) => B): B

foldLeft从左到右，以此将一个二元运算op应用到初始值z和该迭代器（traversable)的所有元素上。以下是该函数的一个用例：

从初值0开始, 这里 foldLeft 将函数 (m, n) => m + n 依次应用到列表中的每一个元素和之前累积的值上。

val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val res = numbers.foldLeft(0)((m, n) => m + n)

print(res) // 55

多参数列表有更复杂的调用语法，因此应该谨慎使用，建议的使用场景包括：

单一的函数参数

在某些情况下存在单一的函数参数时，例如上述例子foldLeft中的op，多参数列表可以使得传递匿名函数作为参数的语法更为简洁。如果不使用多参数列表，代码可能像这样：

numbers.foldLeft(0, {(m: Int, n: Int) => m + n})

注意使用多参数列表时，我们还可以利用Scala的类型推断来让代码更加简洁（如下所示），而如果没有多参数列表，这是不可能的。

numbers.foldLeft(0)(\_ + \_)

像上述语句这样，我们可以给定多参数列表的一部分参数列表（如上述的z）来形成一个新的函数（partially applied function），达到复用的目的，如下所示：

val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val numberFunc = numbers.foldLeft(List[Int]())\_

val squares = numberFunc((xs, x) => xs:+ x\*x)

print(squares.toString()) // List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

val cubes = numberFunc((xs, x) => xs:+ x\*x\*x)

print(cubes.toString()) // List(1, 8, 27, 64, 125, 216, 343, 512, 729, 1000)

最后，foldLeft 和 foldRight 可以按以下任意一种形式使用，

val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

numbers.foldLeft(0)((sum, item) => sum + item) // Generic Form

numbers.foldRight(0)((sum, item) => sum + item) // Generic Form

numbers.foldLeft(0)(\_+\_) // Curried Form

numbers.foldRight(0)(\_+\_) // Curried Form

(0 /: numbers)(\_+\_) // Used in place of foldLeft

(numbers :\ 0)(\_+\_) // Used in place of foldRight

隐式（IMPLICIT）参数

如果要指定参数列表中的某些参数为隐式（implicit），应该使用多参数列表。例如：

def execute(arg: Int)(implicit ec: ExecutionContext) = ???

===========类型系统与隐式转化=======================================

1.泛型

1.类、特质、函数、方法可带有类型参数

//泛型类

class Pair[T,S](val first:T,val second:S)

//泛型特质

trait Pair[T]

//泛型函数

def getType(a:Any) //a是Any类型

def getType[T](a:T) //a是泛型，getType是泛型函数

当类型被指定的时候构成具体的类、函数等

val getInt = getType[Int] \_ 支持类型推断

val p1=new Pair(25,25.0） //生成Pair[Int,Double]类

val p2=new Pair(25.0,25)

2.类型系统

1.类型界定：

上界： A <: T

下界： A >: T 很少使用

上下界定： A >:T1 <: T2

视图界定： A <% T 隐式转化T

上下文界定： T:M 要求必须存在一个类型为M[T]的"隐式值"

1.1.上下界：

有时，需要对类型变量进行限制。考虑这样一个Pair类型，它要求它的两个组件类型相同，就像这样：

class Pair[T](val first:T,val second:T)

现在你想要添加一个方法，产出较小的那个值：

class Pair[T](val first:T,val second:T){

def smaller=if (first.compareTo(second) <0) first else second //错误

}

这是错的----我们并不知道first是否有compareTo方法。要解决这个问题，我们可以添加一个上界T<:Comparable[T]。

class Pair[T<:Comparable[T]](val first:T,val second:T){

def smaller=if (first.compareTo(second) <0) first else second

}

这意味着T必须是Comparable[T]的子类型。

这样一来，我们可以实例化Pair[java.lang.String],但不能实例化Pair[java.io.File],因为String是Comparable[String]的子类型，而File并没有实现Comparable[File]。

例如：

val p=new Pair("Fred","Books")

println(p.smaller)

1.2.视图界定：

val p=new Pair(4,2)

编译器会抱怨说Int不是Comparable[Int]的子类型。和java.lang.Integer包装类型不同，Scala的Int类型并没有实现Comparable。不过，RichInt实现了Comparable[Int],同时还有一个从Int到RichInt的隐式转换。

解决办法就是使用"视图界定",就像下边一样：

class Pair[T <% Comparable[T]]

<%关系意味着T可以被隐式转化成Comparable[T]

eg:

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

1.3.多重界定

T >: lower <:upper //同时有上下界，但不能同时有多个上界或下界

T <: Ordered[T] with Cloneable //能要求同时混入多个特质

T <% Ordered[T] <% String //能同时多个视图限定

T : Ordering : Manifest //能同时多个上下文界定

2.类型约束：

A =:= T A是否等于T

A <:< T A是否为T的子类

A <%< T A是否可以隐式转化成T

注意：使用类型约束需要使用implicit关键字

比如： class Pair[T](val name:T)(implicit ev:T<:<Comparable[T])

3.型变：

在声明Scala的泛型类型时，“+”表示协变，而“-”表示逆变。

class List[-T]

List[String] List[Object]

协变：C[+T]：如果A是B的子类，那么C[A]是C[B]的子类。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。

逆变：C[-T]：如果A是B的子类，那么C[B]是C[A]的子类。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。

不变：C[T]：无论A和B是什么关系，C[A]和C[B]没有从属关系。

例如：

class ScalaObject;

class Father extends ScalaObject;

class Son extends Father;

class Test1[T]

class Test2[+T]

class Test3[-T]

object Test extends App{

def test1(a:Test1[Father])={}

def test2(a:Test2[Father])={}

def test3(a:Test3[Father])={}

test1(new Test1[Father]);

//test1(new Test1[ScalaObject]);

//test1(new Test1[Son]);

test2(new Test2[Father]);

//test2(new Test2[ScalaObject]);

test2(new Test2[Son]);

test3(new Test3[Father]);

test3(new Test3[ScalaObject]);

//test3(new Test3[Son]);

}

3.隐式转换

1.隐式转换

2.利用隐式转化丰富现有类库的功能

3.引入隐式转换

4.隐式转换规则

5.隐式参数

6.利用隐式参数进行隐式转换

7.上下文界定

8.类型证明

3.1.隐式转换

所谓隐式转换函数指的是以implicit关键字声明的带有单个参数的函数。

这样的函数将被自动应用，将值从一种类型转化为另一种类型。

例如：将整数n转化为分数n/1

implicit def int2Fraction(n:Int)=new Fraction(n,1);

调用：

val result=3\*Fraction(4,5)//将调用int2Fraction(3)

隐式转化函数将整数3转化成一个Fraction对象。这个对象接着乘以Fraction(4,5)

3.2.利用隐式转化丰富现有类库的功能

例如：往String类中添加自定义方法

解决办法：定义一个隐式转化的类

1.直接定义隐式转化类

implicit class MyString(val str:String){

def increment=str.map(x=>(x+1).toChar)

}

val str="hello";

str.increment//ifmmp

2.对象中的隐式转换类

package com.briup.base;

object StringUtils{

implicit class StringImprovements(val s:String){

def increment=s.map(x=>(x+1).toChar)

def decrement=s.map(x=>(x-1).toChar)

def hideAll=s.replaceAll(".","\*")

}

}

import com.briup.base.StringUtils.\_

object Main extends App{

println("HAL".increment);

}

3.隐式转化类放在包对象中

package com.briup.base;

package object utils{

implicit class StringImprovements(val s:String){

def increment=s.map(x=>(x+1).toChar)

}

}

import com.briup.base.utils.\_

object Main extends App{

println("HAL".increment);

}

4.使用scala 2.10之前的版本

//定义一个经过丰富的类型，提供你想要的功能

class MyString(val str:String){

def increment=str.map(x=>(x+1).toChar)

}

//在提供一个隐式转换函数将原类的类型转化到这个新的类型

implicit def string2String(s:String)=new MyString(s);

val str="hello";

str.increment

3.3.引入隐式转换

Scala会考虑如下的隐式转换函数：

1.位于源或目标类型的伴生对象中的隐式函数。

2.位于当前作用域可以以单个标识符指代的隐式函数。

例如：上边的int2Fraction函数，

1.我们可以将它放在目标类型Fraction的伴生对象中，这样他就能够用来将整数转化成分数了。

2.或者我们可以将它放到一个FractionConversions对象中，而这个对象位于com.briup.test包。那么如果想要使用这个转换，就需要引入FractionConversions对象，

如：

//引入所有成员

import com.briup.test.FractionConversions.\_

//只引入int2Fraction方法

import com.briup.test.FractionConversions.int2Fraction

//引入除了int2Fraction之外的所有成员

import com.briup.test.FractionConversions.{int2Fraction => \_,\_}

单凭如下的引入语句是不够用的：

import com.briup.test.FractionConversions

上面的这个语句引入的是FractionConversions对象本身，而int2Fraction这个方法只能通过FractionConversions.int2Fraction的形式被任何想要显示调用它的人使用。

但如果该函数不能直接以int2Fraction访问到，不加限定词的话，编译器是不会使用它的。

注：在REPL中，可以通过“:implicits”查看所有除了Predef外被引入的隐式成员，或着输入“:implicits -v”查看全部。

3.4.隐式转换规则

隐式转换在如下三种各不相同的情况下被考虑：

1.当表达式的类型与预期类型不同时：如 sqrt(Fraction(1,4))//将调用fraction2Double,因为sqrt预期的是一个Double

2.当对象访问一个不存在的成员时：如 "hello".increment //String中没有increment方法，则采用因素转换中的合适方法

3.当对象调用某个方法，而该方法的参数声明与传入参数不匹配时：如 3\*Fraction(4,5) //将调用int2Fraction，因为Int的\*方法不接受Fraction作为参数。

另一方面，有三种情况下编译器不会尝试使用隐式转换：

1.如果代码能够在不使用隐式转换的前提下通过编译，则不会使用隐式转换。如：a\*b能编译通过，那么编译器不会尝试a\*convert(b)或者convert(a)\*b

2.编译器不会尝试同时执行多个转化。如：convert1(convert2(a))\*b

3.存在二义性的转换是个错误。如：convert(a)\*b 与 convert2(a)\*b 都是合法的，编译器报错

注意：如果想要弄清楚编译器使用了哪一些隐式转换，可以使用如下命令行参数来编译自己的程序。

scalac -Xprint:typer MyTest.scala

将会看到加入隐式转换之后的源码。

3.5.隐式参数

函数或者方法可以带有一个标记为implicit的参数列表。

这种情况下编译器会去查找缺省值，提供给该函数或者方法。

例如：

case class Delimiters(left:String,right:String)

def quote(what:String)(implicit delims:Delimiters)={

delims.left+what+delims.right

}

我们可以通过显示传参数调用quote方法:

quote("^\_^")(Delimiters("[","]"))

同时我们也可以略去隐式参数列表：

quote("^\_^")

在我们略去隐式参数列表的情况下，编译器将会查找一个类型为Delimiters的隐式值。这必须是一个声明为implicit的值。编译器会在如下两个地方查找这个一个对象：

1.在当前作用域所有的可以用单个标识符指代的满足类型要求的val和def.

2.与所要求类型相关联的类型的伴生对象。相关联的类型包括所要求类型本身以及他的类型参数。

我们可以做一个对象，比如：

object FrenchTest{

implicit val quoteDelimiters = Delimi./ters("[","]")

}

我们就可以从这个对象引入所有的值：

import FrenchTest.\_

或特定的值：

import FrenchTest.quoteDelimiters

如此一来，函数或者方法中的隐式参数就会被隐式的赋值。

3.6.利用隐式参数进行隐式转换

隐式的函数参数也可以被用做隐式转换。

例如：这个泛型函数

def smaller[T](a:T,b:T)=if (a>b) b else a

这实际行不通。编译器不会接受这个函数，因为它并不知道a和b是否属于一个带有>操作符的类型。

我们可以提供一个转换函数来达到目的：

def smaller[T](a:T,b:T)(implicit order: T => Ordered[T])={if (order(a)>b) b else a}

由于Ordered[T]特质有一个接受T作为参数>操作符，因此这个版本的是正确的。

Predef对象中大量类型都定义了T=>Ordered[T]，包括所有已经实现了Ordered[T]或Comparable[T]的类型。正因为如此，我们才可以调用：

smaller(40,2)

smaller("Hello","World")

如果你想要调用：

smaller(Fraction(1,7),Fraction(2,9))

那么你就需要定一个一个Fraction=>Ordered[Fraction]的函数，要么在调用的时候显示写出，要么把它做成一个implicit val.

注意order是一个带有单个参数的函数，被打上了implicit标签，并且有一个以单个标识符出现的名词。因此它不仅是一个隐式参数，它还是一个隐式转换。

因此可以简写为：

def smaller[T](a:T,b:T)(implicit order: T => Ordered[T])={if (a>b) a else b//如果a没有>操作符，那么将调用order(a)>b}

3.7.上下文界定

// Int Ordered[Int] T <:Ordering[T]

// Int : Ordered

类型参数可以🈶️一个形式为T:M的上下文界定，其中M是另一个泛型类型。它要求作用域中存在一个类型为M[T]的隐式值。

例如：class Pair[T:Ordering]

要求存在一个类型为Ordering[T]的隐式值。该隐式值可以被用在该类的方法当中，考虑如下示例：

class Pair[T:Ordering](val first:T,val second:T) {

def smaller(implicit order:Ordering[T])={

if (order.compare(first,second) > 0) first else second

}

}

你也可以用Predef类中的implicity方法获取到该值：

class Pair[T:Ordering](val first:T,val second:T) {

def smaller={

if (implicitly[Ordering[T]].compare(first,second) > 0) first else second

}

}

或者，也可以利用Ordered特质中定义的从Ordering到Ordered的隐式转换。一旦引入这个转换，你就可以使用关系操作符：

class Pair[T:Ordering](val first:T,val second:T) {

def smaller={

import scala.math.Ordered.\_;

//if (first.compare(second) < 0) first else second

if (first < second) first else second

}

}

这些都是些细微的变化，重要的好似你可以随时实例化Pair[T]，只要满足存在类型为Ordering[T]的隐式值的条件即可。举例来说，如果你想要一个Pair[Point]，则可以组织一个隐式的Ordering[Point]值：

implicit object PointOrdering extends Ordering[Point]{

override def compare(x: Point, y: Point): Int = {

if(x.start+x.end < y.start+y.end) 1 else -1

}

}

3.8.类型证明

类型约束：

T=:=U

T<:<U

T<%<U

要使用这样的类型约束，做法是提供一个隐式参数。

类型界定常常用于以下两种场景：

1.在泛型类中，定义在特定条件下的才能使用的方法

2.协助类型推演

def firstLast[A,C](it:C)(implicit ev: C <:<Iterable[A])=(it.head,it.last)

=:= <:< <%< 是带有隐式值的类，定义在Predef对象当中。例如：<:<从本质上讲就是

sealed abstract class <:<[-From, +To] extends (From => To) with Serializable

//注解告诉编译器错误信息

@implicitNotFound(msg="Cannot prove that ${C} <:< ${Iterable[A]}")

def firstLast[A,C](it:C)(implicit ev:C<:<Iterable[A])=

(it.head,it.last)

val info=firstLast[Int,List[Int]](List(1,2))

println(info)