之前介绍过scala中的function，class，object和trait，想先告一段落。我一直有一个概念想讲但又怕自己理解的不够深入，但我觉得对于大家去学习spark和akka非常有用，所以我还是决定和大家说一下，那就是implicit。

Scala为什么需要implicit呢？

implicit是scala中的一个关键字，关于它有着丰富的用法，使得scala更灵活和容易扩展。Scala在面对编译出现类型错误时，提供了一个由编译器自我修复的机制，编译器试图去寻找一个隐式implicit的转换方法，转换出正确的类型，完成编译。这就是implicit 的意义。

我先从以下几个比较简单的方面阐述implicit：

1. Implicit Conversions
2. Using Implicits for Enriching Existing Classes
3. Importing Implicits
4. Rules for Implicit Conversions
5. Implicit Parameters
6. Implicit Conversions with Implicit Parameters

首先来看一下第一部分 ：

* Implicit Conversions

我们先来看一个例子：

**class** Fraction(n: Int, d: Int) {  
 **private val** *num* = n  
 **private val** *den* = d  
  
 **def** \*(other: Fraction) = **new** Fraction(*num* \* other.*num*, *den* \* other.*den*)  
  
 **override def** toString: String = **"num: "** + n + **","** + **"den: "** + d  
}

我们定义一个Fraction类。

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

一个隐式转换函数只能有一个参数，并且带有implicit关键字，这里只有一个n：Int。这个函数可以自动的将int转换成Fraction。

现在我们可以调用：

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

int2Fraction自动将integer 3转乘一个Fraction的对象，这样就可以做分数相乘。

下面是这个例子的完整代码：

**class** Fraction(n: Int, d: Int) {  
 **private val** *num* = n  
 **private val** *den* = d  
  
 **def** \*(other: Fraction) = **new** Fraction(*num* \* other.*num*, *den* \* other.*den*)  
  
 **override def** toString: String = **"num: "** + n + **","** + **"den: "** + d  
}  
  
**object** Fraction {  
  
 **implicit def** int2Fraction(n: Int) = *Fraction*(n, 1)  
  
 **def** main(args: Array[String]): Unit = {  
  
 **val** result = 3 \* *Fraction*(4, 5) // Calls int2Fraction(3)  
  
 *println*(result)  
 }  
  
 **def** apply(n: Int, d: Int): Fraction = **new** Fraction(n, d)

}

Output：

num: 12,den: 5

implicit告诉编译器，这个函数式一个隐式转换函数，能够把Int类型转换成Fraction类型。当我们调用3 \* Fraction(4, 5)时，Int类型并没有这样的操作，所以编译器会在当前作用域寻找合适的隐式转换，来尝试使这种操作合法。隐式转换发生在这两种情景：

e是一个S类型的表达式，而需要的却是T类型，编译器会寻找S=>T的隐式转换

e是一个S类型的表达式，使用点号访问e.m时，m不是类型S的成员，编译器会寻找合适的隐式转换使e.m合法

隐式转换最常用的用途就是扩展已有的类，在不修改原有类的基础上为其添加新的方法、成员。

* Using Implicits for Enriching Existing Classes

你是否曾经希望一个类有某个方法，但是它的创建者并没有提供？举个例子：如果java.io.File有一个read方法可以读取文件会不会更好一些呢？

Val contents = new File(“README”).read

作为一个java的程序员，你只能向Oracle公司求助添加这个方法。

在scala中，你可以定义一个class来提供你想要的：

class RichFile(val from: File) {

def read = Source.fromFile(from.getPath).mkString

}

然后提供一个隐式转换函数：

implicit def file2RichFile(from: File) = new RichFile(from)

现在你就可以调用在File的instance上调用read方法了。File的会隐式转换成RichFile。如果你觉得这样有点麻烦，可以结合起来定义一个隐式转换类RichFile：

implicit class RichFile(val from: File) { ... }

隐式转换类的主构造函数只能有一个参数。这个构造函数就成为了隐式转换函数。

代码如下：

Implicit convert：

**import** java.io.File  
**import** scala.io.Source  
  
**object** TestImplicitClass **extends** App{  
   
 **class** RichFile(**val** from: File) {  
 **def** read = Source.*fromFile*(from.getPath).mkString  
 }  
   
 **implicit def** file2RichFile(from: File) = **new** RichFile(from)  
  
 **val** *file* = **new** File(**"C:/test.txt"**)  
 *println*(*file*.read)  
}

output：

test implicit

implicit class：

**import** java.io.File  
**import** scala.io.Source  
  
**object** TestImplicitClass **extends** App{  
  
 **implicit class** RichFile(**val** from: File) {  
 **def** read = Source.*fromFile*(from.getPath).mkString  
 }  
  
 **val** *file* = **new** File(**"C:/test.txt"**)  
 *println*(*file*.read)  
}

output：

test implicit

CAUTION：

隐式类不能是顶层类。您可以将它放在使用类型转换的类中，也可以放在您导入的另一个对象或类中。

* Import implicits

Scala 会尝试一下范围的隐式转换函数：

1. 源或目标类型的伴随对象中的隐式函数或类
2. 在作用范围内的隐式函数或者类

还是以之前Fraction的为例子，我把隐式转换放到伴生对象中，可以起作用。但是如果我把int2Fraction放到其他包中呢？比如我把int2Fraction放到FractionConversions object中，FractionConversions在implicitLearning.chapter21 包中，如果你想用这个隐式转换，可以这样：

import implicitLearning.chapter21.FractionConversions.\_

将隐式转换导入当前作用域，不能有前缀。怎么理解呢？

继续拿刚才的例子，我们不能这么导入import implicitLearning.chapter21.FractionConversions这样导入我们使用int2Fraction就是有前缀的，FractionConversions.int2Faction，这样我们可以显示调用，但是编译器不会把int2Fraction当做implicit来处理。

你也可以指定你需要的隐式转换：

import implicitLearning.chapter21.FractionConversions.int2Fraction

也可以排除可能导致发送错误的转换：

import implicitLearning.chapter21.FractionConversions.{int2Fraction => \_, \_}

这样就可以导入除int2Fraction的其他部分。

TIP：如果你想找为什么编译器没有用到你想用的隐式转换，你可以显示的去调用。还是刚才的例子，你可以直接调用int2Fraction(3) \* Fraction(4,5).这样你就能得到相应的错误信息。

* Rules for Implicit Conversions

隐式转换在什么时候发生呢？为了说明这个问题，我们还是引用之前的例子，假设int2Fraction和fraction2Double都存在。

隐式转换在下面三种不同的情况下被考虑到：

1. 表达式的类型与期望的不一致：

3 \* Fraction(4, 5) // Calls fraction2Double

Int class没有\*(Fraction)方法，但是有\*(Double)

1. Object访问一个不存在的成员

3.den // Calls int2Fraction

Int class没有den成员，但是Fraction有

1. 如果object调用方法的参数与给定的参数不匹配

Fraction(4, 5) \* 3 // Calls int2Fraction

\* 方法不能接收Int，但是可以接收Fraction。

另一方面，有三种情况隐式转换不会发生：

1. 如果代码编译没有使用隐式转换。
2. 编译器不会尝试多次转换，比如：convert1(convert2(a)) \* b
3. 有歧义的转换。比如，convert1(a) \* b和convert2(a) \* b都可行的时候，编译器会报错。

CAUTION：

3 \* Fraction(4, 5)可以转换成3 \* fraction2Double(Fraction(4, 5))或者int2Fraction(3) \* Fraction(4, 5)，这并不是一个有歧义的问题。第一个转换优先于第二个转换，因为因为它不需要修改调用\*方法的对象。

* Implicit Parameters

函数可以有一个参数列表被标记为implicit。在这种情况下，编译器会寻找默认值来支持函数调用。这里有一个例子：

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

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

delims.left + what + delims.right

你可以通过显示Delimiters object调用quote方法，像这样：

quote("Bonjour le monde")(Delimiters("«", "»")) // Returns «Bonjour le monde»

不知道看到这个例子是否想起我之前说的柯里化，你可以省略隐式参数列表：

quote("Bonjour le monde")

在这种情况下，编译器会自动寻找类型为Delimiters的隐式值。这个值必须声明为implicit。编译器会去一下两个地方去寻找：

1. 所有声明为该类型的val和def的值，并且在当前作用域中，没有前缀（我刚才说前面讲到的）
2. 在与所需类型关联的类型的伴随对象中。关联的类型包括所需类型本身，如果他是一个参数类型，包括类型参数。

说的有点难以理解，举个例子：

object FrenchPunctuation {

implicit val quoteDelimiters = Delimiters("«", "»")

...

}

然后导入这个object中的所有值：

import FrenchPunctuation.\_

或者指定某个值：

import FrenchPunctuation.quoteDelimiters

现在French delimiters可以支持quote函数的隐式调用了。

CAUTION：

给定数据类型只能有一个隐式值。 因此，使用通用类型的隐式参数并不是一个好主意。

def quote(what: String)(implicit left: String, right: String) // No!

再来举个例子：

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。