# 隐式转换和隐式参数

如果想使用别人提供的类库的同时保留修改和拓展这些类库的能力，scala提供了隐式转换和隐式参数的机制。

## 隐式转换

隐式转换的使用场景是在开发时完全不知道对方存在的软件或者类库时，它们各自都有自己的方式来描述某个概念，而这些概念本质上是同一件事。隐式转换可以**避免一个类型显示的转换成另外一个类型**。

比如java的Swing类库会处理操作系统的事件，将它们转换成平台独立的事件对象，并将这些事件传递给被称为事件监听器的应用代码

如果没有隐式转换，使用Swing的scala程序必须像java那样使用内部类。

val button = new JButton

button.addActionListener(

new ActionListener {

def actionPerformed(event: ActionEvent) = {

println(“pressed!”)

}

}

)

但使用隐式转换以上代码就可以简单的写出

val button = new JButton

button.addActionListener(

(\_:ActionEvent) => println(“pressed”)

)

函数到ActionListener的转换是通过如下的一个隐式转换完成的

Implicit def func2ActionListener(f: ActionEvent => Unit) = {

new ActionListener {

def actionPerformed(event: ActionEvent) = {

f(event)

}

}

}

如此就到达了避免写一个显示的类型转换的效果。

## 隐式规则

隐式定义指那些我们允许编译器插入程序以解决类型错误的定义。比如x + y不能通过编译，那么编译器可能会把它改成convert(x) + y，其中convert就是某种可用的隐式转换，如果convert能把x转成某种支持+方法的对象，那么这个改动将修复程序。这种方法就有助于澄清程序逻辑，因此也不需要显示的写出convert。

隐式转换收到如下规则的约束

1. 标记规则

必须使用关键字implicit，这个关键字可以标记任何变量、函数或者对象。（implicit标记变量、和对象的例子？？？）

1. 作用域规则

必须是当前作用域的单个标识符或者跟隐式转换的源类型或者目标类型有关联。

比如编译器不会插入someVariable.convert(x) + y这种形式。必须先引入someVariable。类库的常见做饭是提供一个包含一些有用的隐式转换的Preamble对象。

与目标有关联指如下这种情况

比如在Dollar的伴生对象中例子

object Dollar {

implicit def dollarToEuro(x: Dollar) : Euro = ...

}

Class Dollar {...}

1. 每次一个规则

编译器绝不会将x + y重写为convert1(convert2(x)) + y. 不过可以通过隐式定义包含隐式参数的方式绕过这个限制。

1. 显式优先规则

隐式定义会发生在如下三个场景

## 转换到一个预期的类型

每当编译器看见一个X而它需要一个Y的时候，它就会查找一个能将X转换成Y的隐式转换。例如，通常一个双精度浮点数不能被用作整数，因为这样会丢失精度：

所以val i:Int = 3.5会得到如下输出：

<console>:12: error: type mismatch;

found : Double(3.5)

required: Int

val i:Int = 3.5

通过下面的隐式函数能fix这个问题

implicit def doubl2Int(a:Double) = a.toInt

## 对某个选择接收端的转换

假定你写下了obj.doIt，而obj并没有一个名为doIt的成员，编译器会在放弃之前尝试插入转换。在本例中，这个转换需要应用于接收端，也就是obj。编译器会装作obj的预期“类型”为“拥有名为doIt的成员”。

比如对 1 + oneHalf这种用法，如果定义了这样一个隐式函数

Implicit def int2Rational(x:Int)= new Rational(x,1)

这种写法就是支持的。

这种做法可以用来模拟添加一个新的语法，比如Map的定义：

Map（1->”one”, 2->”two”, 3->”three”）其中当你写下1->”one”时，编译器会插入一个1到ArrowAssoc的转换，以便能使用->方法

Implicit def any2ArrayAssoc[A](x:A):ArrowAssoc[A] = new ArrowAssoc(x)

其中

Class ArrowAssoc[A](x:A) {  
 def -> [B](y:B):Tuple2[A,B] = Tuple2(x,y)

}

这样在scala中可以通过这样的富包装类模式可以方便的定义个内部DSL。对其他语言来说是不可能的。

同样的场景也适用于隐式类的定义

## 隐式参数

编译器会插入隐式定义的最后一个地方是参数列表，把someCall(a)替换成someCall(a)(b)或者new Some(a)替换成new Some(a)(b)，通过追加一个参数列表的方式来完成某个函数调用。隐式参数提供的是整个最后一组的柯里化的参数列表。

class PreferredPrompt(val preference:String)

class PreferredDrink(val name:String)

object Greeter {

def greet(name: String)(implicit prompt:PreferredPrompt, drink: PreferredDrink) = {

println("Welcome," + name)

println("before start, take some " + drink.name)

println(prompt.preference)

}

}

object JoesPrefs {

implicit val prompt = new PreferredPrompt("Yes > ")

implicit val drink = new PreferredDrink("caffe")

}

object Main extends App {

import JoesPrefs.\_

Greeter.greet("chen")

}

## 上下文界定

根据隐式参数的定义，可以写出如下的函数。

def maxListImp[T](elemets:List[T])(implicit ordering: Ordering[T]):T = {

elemets match {

case Nil => throw new IllegalArgumentException("empty List")

case x::Nil => x

case x::rest =>

val max =maxListImp(rest)(ordering) //这里可以省掉（ordering）

if (ordering.gt(x, max)) x

else max

}

}

还可以通过界上下文界定来缩短方法签名

def maxListImp[T:Ordering](elemets:List[T]):T = {

elemets match {

case Nil => throw new IllegalArgumentException("empty List")

case x::Nil => x

case x::rest =>

val max =maxListImp(rest)

if (implicitly[Ordering[T]].gt(x, max)) x

else max

}

}

[T: Ordering]这样的语法是一个上下文界定，它做了两件事：首先，它像平常那样引入了一个类型参数T；其次，它添加了一个类型为Ordering[T]的隐式参数。

Implicitly是一个标准类库中的定义

def implicitly[T](implicit t:T) = t

调用implicitly[Foo]的作用是编译器会查找一个类型为Foo的隐式定义。然后它会用这个对象来调用implicitly方法，这个方法再将这个对象返回。这样就可以在想要当前作用域找到类型为Foo的隐式对象时直接写implicitly[Foo]。所以implicitly[Ordering[T]]来通过其类型获取ordering参数的用法。

# 反射

## 定义

在运行时已知一个类型或者是某个对象的实例，反射是指如下能力：

1. 检索对象的类型包括Generic Types
2. **导入运行时反射的Universe，这样就能使用TypeTag**

scala> **import** scala.reflect.runtime.{universe => ru}

**import** scala.reflect.runtime.{universe=>ru}

1. **创建一个List的对象**

scala> **val** l = **List**(1,2,3)

l: **List**[**Int**] = **List**(1, 2, 3)

1. **定义个方法getTypeTag，这个方法会让编译器产生一个泛型类型的TypeTag**

scala> **def** **getTypeTag**[**T**: ru.**TypeTag**](obj: **T**) = ru.typeTag[**T**]

getTypeTag: [**T**](obj: **T**)(**implicit** evidence$1: ru.**TypeTag**[**T**])ru.**TypeTag**[**T**]

**问题：可否直接写成隐式参数的方式？？**

1. **使用刚刚创建的对象调用getTypeTag以获取一个List的typeTag**

scala> **val** theType = getTypeTag(l)

theType: TypeTag[List[Int]] = TypeTag[List[Int]]

1. **使用这个List的typeTag的方法tpe返回typeTag中的类型**

scala> **val** type = theType.tpe

typ: reflect.runtime.universe.Type = List[Int]

1. **得到的类型我们就能检查类型中的方法。**

scala> **val** decls = theType.decls.take(10)

decls: **Iterable**[ru.**Symbol**] = **List**(constructor **List**, method companion, method isEmpty, method head, method tail, method ::, method :::, method reverse\_:::, method mapConserve, method ++)

**注： 通过ru.typeTag[List[Int]]可直接获取到List[Int]的运行时typeTag。**

1. 初始化一个新的实例
2. **定义一个class**

**case class Person(name: String)**

1. **获取一个ClassLoader mirror，通过这个mirror可以获取到当前ClassLoader下的所有类和类型。**

**val m = ru.runtimeMirror(getClass.getClassLoader)**

1. **通过runtime universe获取目标class的ClassSymbol。**

**val classPerson:ru.ClassSymbol = ru.typeOf[Person].typeSymbol.asClass**

1. **ClassLoader mirror通过reflectClass方法传入目标的class的symbol，获取一个invoker mirror类型为ClassMirror。**

**val cm:ru.ClassMirror = m.reflectClass(classPerson)**

1. **通过runtime universe获取目标class的构造函数的MethodSymbol**

**ctor = ru.typeOf[Person].decl(ru.termNames.CONSTRUCTOR).asMethod**

1. **通过classMIrror的reflectConstructor方法传入对应的MethodSymbol获取到MethodMirror**

**val ctorm = cm.reflectConstructor(ctor)**

1. **得到的methodMirror就能被当成构造函数用了。**

**scala> val p = ctorm("Mike")**

**p: Any = Person(Mike)**

1. 访问或者调用这个对象的成员函数和变量

跟其他JVM语言一样，scala的类型会在编译时期被擦除，这表示如果你想检查某些实例的运行时类型，你可能无法访问像scala编译器在编译时能放回的所有类型信息。TypeTag是一个把在编译时可用的信息带到运行时的对象，这个对象的建立只能被两种方式触发，一种是implicit parameter，另一种是context bounds。

## 概念

1. 环境（Environment）

反射的环境根据反射任务发生的地方不同而有两种情况，一种是运行时反射环境，一种是编译时反射环境。两种不同的环境由universe封装。

另一种关于反射环境的重要方面是反射可以访问的一个实体（entity）的集合，这个集合就是所谓的mirror。比如通过ClassloaderMirror可以提供运行时访问的实体。这些实体被称为一个mirror，这个mirror就只提供被这个Classloader能访问的实体（packages、types、members等）。

同时mirror也提供了可被反射使用的操作，比如这个类的方法或者是构造函数。

1. Universes

* scala.reflect.runtime.universe for **runtime reflection**, or
* scala.reflect.macros.Universe for **compile-time reflection**.

一个Universe为反射中使用的主要概念提供了接口，比如Types，Trees，Annotations。

1. Mirrors

Mirrors提供了所有反射期间可以访问的信息。根据获取信息类型的不同或者使用的反射动作不同需要使用不同的mirror。

A．ClassLoader mirrors

这一类mirrors通过方法staticClass，staticModule，staticPackage会把names转换成symbols，从而获取类型或者方法的representions。

类型是scala.reflect.api.Mirrors#ReflectiveMirror，

通过方法scala.reflect.api.JavaMirrors#runtimeMirror获取得到。

通过ClassLoader mirror获取Invoker mirror包括（

scala.reflect.api.Mirrors#InstanceMirror,

scala.reflect.api.Mirrors#MethodMirror,

scala.reflect.api.Mirrors#FieldMirror,

scala.reflect.api.Mirrors#ClassMirror,

scala.reflect.api.Mirrors#ModuleMirror）。

B．Invoker mirrors

这一类mirror通过MethodMirror.apply, FieldMirros.get方法实现了反射的调用。