Scala implicit type 기초 이해하기

myjlms99@gmail.com

암묵적 처리 기준

암시적 처리 규칙

정의된 것을 내부적으로 규칙화 되므로 implicit 키워드 정의된 것을 처리한다.

표시규칙 : implicit 로 표시한 정의만 검토 대상이 된다

스코프 규칙 : 삽입할 implict 변환은 스코프 내에 단일 식별자로만 존재하거나, 변환의 결과나 원래 타입과 연관 필요

한번에 하나만 규칙 : 오직 하나의 암시적 선언만 사용

명시성 우선 규칙: 코드가 그 상태 그대로 타입 검사를 통과 한다면 암시를 통한 변환을 시도치 않음

함수에 암묵적 매개변수 지정하기

암묵적 매개변수 하나 지정.

함수 내에 매개변수를 implicit를 지정해서 정의하면 암묵적 매개변수가 지정된다. 실제 매개변수를 넣고도 실행이 되지만 함수만 호출하면 내부적으로 암묵적 정의한 값을 처리해 준다.

```
scala> def p(implicit i :Int) = print(i)
p: (implicit i: Int)Unit

scala> p(10)
10
scala> implicit val v = 2
v: Int = 2

scala> p
2
```

암묵적 매개변수 두개 지정

함수 내의 암묵적 매개변수가 하나인데 2개를 지정하면 실제 값을 제대로 찾을 수가 없어 오류를 발생한다.

```
scala> def pTwo(implicit i :Int) = print(i)
pTwo: (implicit i: Int)Unit

scala> implicit val v = 2
v: Int = 2

scala> implicit val v2 = 20
v2: Int = 20

scala> p
<console>:29: error: ambiguous implicit values:
 both value v of type => Int
 and value v2 of type => Int
 match expected type Int
       p
       ^
```

함수에는 하나만 암묵적 매개변수만 존재

함수를 정의할 때 제일 앞에 하나만 암묵적 매개변수를 지정해서 사용할 수 있다.

```
scala> def ppTwo(implicit i :Int, a:Int) = print(i)
ppTwo: (implicit i: Int, implicit a: Int)Unit

scala> def ppTwo(implicit i :Int, a:Int) = print(a,i)
ppTwo: (implicit i: Int, implicit a: Int)Unit

scala> def ppTwo(implicit i :Int, implicit a:Int) = print(a,i)
<console>:1: error: identifier expected but 'implicit' found.
       def ppTwo(implicit i :Int, implicit a:Int) = print(a,i)
                                  ^


scala> def ppTwo(i :Int, implicit a:Int) = print(a,i)
<console>:1: error: identifier expected but 'implicit' found.
       def ppTwo(i :Int, implicit a:Int) = print(a,i)
                         ^
```

부분함수를 암묵적으로 지정하기

함수 매개변수 목록을 이용해서 실제 들어오는 매개변수를 나눠서 처리할 수 있다. 이때도 실제 암묵적 매개변수 지정은 하나만 있어야 한다.

```
scala> def mul(x:Int)(implicit y:Int) = x * y
mul: (x: Int)(implicit y: Int)Int

scala> mul(3)
<console>:13: error: could not find implicit value for parameter y: Int
       mul(3)
           ^


scala> implicit val z:Int = 10
z: Int = 10

scala> mul(3)
res1: Int = 30
```

암묵적 함수 내의 타입 변환하기

컨버전 함수를 암묵적 생성

함수 내의 타입을 처리하기 위해서 암묵적인 함수가 작동하도록 정의한다.

```
scala> def convert(msg:String) : Unit = println(msg)
convert: (msg: String)Unit

scala> convert(1)
<console>:14: error: type mismatch;
 found   : Int(1)
 required: String
       convert(1)
               ^


scala> implicit def intToString(i:Int) : String = i.toString
<console>:12: warning: implicit conversion method intToString should be enabled
by making the implicit value scala.language.implicitConversions visible.
This can be achieved by adding the import clause 'import scala.language.implicit
Conversions'
or by setting the compiler option -language:implicitConversions.
See the Scaladoc for value scala.language.implicitConversions for a discussion
why the feature should be explicitly enabled.
       implicit def intToString(i:Int) : String = i.toString
                    ^

intToString: (i: Int)String

scala> convert(7)
7
```

명시적으로 처리하기

암묵적으로 지정된 함수를 실제 명시적으로 사용해도 변환이 되는 것을 알 수 있다.

```
scala> implicit def intToString(i:Int) : String = i.toString
<console>:12: warning: implicit conversion method intToString should be enabled
by making the implicit value scala.language.implicitConversions visible.
This can be achieved by adding the import clause 'import scala.language.implicit
Conversions'
or by setting the compiler option -language:implicitConversions.
See the Scaladoc for value scala.language.implicitConversions for a discussion
why the feature should be explicitly enabled.
       implicit def intToString(i:Int) : String = i.toString
                    ^
intToString: (i: Int)String

scala> convert(7)
7

scala> convert(intToString(7))
7

scala>
```

암묵적 클래스 타입 변환하기

암묵적 클래스 타입 변환

암묵적으로 지정된 클래스의 인스턴스로 타입을 변경해서 내부의 메소드를 처리한다

```
scala> 3.chat
<console>:14: error: value chat is not a member of Int
       3.chat
         ^


scala> class LoInt(x:Int) {
     |     def chat:Unit = for (i <- 1 to x) println("Hi !")
     | }
defined class LoInt

scala> implicit def intToLoInt(x:Int) = new LoInt(x)
<console>:14: warning: implicit conversion method intToLoInt should be enabled
by making the implicit value scala.language.implicitConversions visible.
This can be achieved by adding the import clause 'import scala.language.implicit
Conversions'
or by setting the compiler option -language:implicitConversions.
See the Scaladoc for value scala.language.implicitConversions for a discussion
why the feature should be explicitly enabled.
       implicit def intToLoInt(x:Int) = new LoInt(x)
                    ^
intToLoInt: (x: Int)LoInt

scala> 3.chat
Hi !
Hi !
Hi !
```

암묵적 클래스 처리할 때 import 처리

먼저 scala.language.implicitConversions를 import한 후에 타입 컨버전을 처리하면 warnning이 사라진다.

```
scala> import scala.language.implicitConversions
import scala.language.implicitConversions

scala> implicit def intToLoInt(x:Int) = new LoInt(x)
intToLoInt: (x: Int)LoInt

scala> 3.chat
Hi !
Hi !
Hi !
```

암묵적 단순 클래스 정의하기

명시적 처리 클래스 정의

클래스를 명시적을 주고 인스턴스를 생성해서 메소드를 호출해서 처리

```
scala> class Awe1(val s:String) extends AnyVal {
     |    def awesome_ = s+ " Awesme "
     | }
defined class Awe1
```

```
scala> val a = new Awe1("SSSSSS")
a: Awe1 = Awe1@925ad720

scala> a.awesome_
res28: String = "SSSSSS Awesme "
```

암묵적 처리 클래스 정의

실제 암묵적 처리는 내부에서 자동으로 이 클래스 내의 메소드를 호출해서 처리하도록 자동으로 연결해 준다.

특정 클래스를 암묵적으로 정의하면 그 메소드를 호출할 때 처리한다.

```
scala> implicit class Awe(val s:String) extends AnyVal {
     |  def awesome_! = s + "! Awesone !"
     | }
defined class Awe
```

```
scala> "SSSS".awesome_!
res27: String = SSSS! Awesone !
```

암묵적 제너릭 클래스 정의하기

암묵적 처리 클래스 정의

인터페이스를 trait으로 정의하고 두개의 클래스를 암묵적으로 정의한다.

```
scala> trait Monoid[A] {
     |    def zero: A
     |    def plus(a:A, b:A) :A
     | }
defined trait Monoid

scala> implicit object IntMonoid extends Monoid[Int] {
     |    override def zero:Int = 0
     |    override def plus(a:Int, b:Int) : Int = a+b
     | }
defined object IntMonoid

scala> implicit object StringMonoid extends Monoid[String] {
     |    override def zero:String = ""
     |    override def plus(a:String, b:String) : String = a.concat(b)
     | }
defined object StringMonoid
```

함수 정의 및 정수처리

함수를 정의할 때 암묵적으로 trait에 연결하면 이 함수를 실행할 때 내부의 암묵적 메소드를 호출할 수 있다.

```
scala> def sum[A](values:Seq[A])(implicit ev:Monoid[A]) : A = values.foldLeft(ev
.zero)(ev.plus)
sum: [A](values: Seq[A])(implicit ev: Monoid[A])A

scala> sum(List(0,0))
res126: Int = 0

scala> sum(List(1,1))
res127: Int = 2
```

문자열 처리

암묵적으로 문자열을 호출하면 문자열 처리에 필요한 메소드가 호출되어 처리되는 것을 알 수 있다.

```
scala> sum(Seq("Hello","world"))
res129: String = Helloworld

scala> sum(List("Hello","world"))
res130: String = Helloworld

scala>
```

암묵적으로 타입 제한자 사용하기

generalized type constraints.

특정 타입을 implicit 내에서 한정을 하기 위해 타입 제한자를 사용해야 한다.

A =:= B	A는 B와 같아야 함
A <:< B	A는 B의 하위 타입이어야 함
A <%< B	A는 B로 볼 수 있어야 함

클래스에 타입매개변수만 사용

클래스에 타입 매개변수를 받고 이를 메소드에 한정한다. 메소드 내의 타입과 클래스의 타압 매개변수가 명확하게 처리되지 않는다.

바인딩 된 A <: Int 형식이 작동하지 않습니다. A는 클래스 선언에서 클래스 본문에 정의되었습니다. 스칼라 컴파일러는 모든 유형 바인딩이 A의 정의와 일치해야합니다. 여기서 A는 바운드가 없으므로 Int가 아닌 Any로 묶입니다.

```
scala> class C[A](value:A) {
     |   def diff[A <: Int](b:Int) = value -b
     | }
<console>:25: error: value - is not a member of type parameter A
         def diff[A <: Int](b:Int) = value -b
                                           ^
```

메소드 내에서 타입 제한자 사용

클래스에 정의된 타입 매개변수가 메소드 내에서도 명확히 사용되려면 implict로 명확한 타입에 대한 정보를 제공해야 합니다.

유형 바운드를 설정하는 대신 메소드는 유형에 대한 특정 임시 "증거"를 요구할 수 있습니다.

```
scala> class C[A](value:A) {
     |    def diff(b:Int)(implicit ev:A =:= Int) = value -b
     | }
defined class C

scala> val c = new C[Int](100)
c: C[Int] = C@66f15f62

scala> c.diff(40)
res107: Int = 60
```

타입 제한자 사용 처리

클래스의 타입 매개변수를 메소드에서 타입제한자로 정의하면 실제 처리할 때도 그 범위에서만 처리된다.

```
scala> class D[A](value : A) {
     |    def add(implicit evidence: A =:= Int) = 100 + value
     | }
defined class D

scala> (new D(123)).add
res114: Int = 223

scala> (new D("123")).add
<console>:25: error: Cannot prove that String =:= Int.
       (new D("123")).add
                      ^
```

타입매개변수와 타입제한자 비교

함수에 타입 매개변수

함수를 정의할 때 특정 상한 경계로 매개변수를 지정해서 Int를 넣었지만 실제 Any 타입으로 인식되므로 처리가 된다.

```
scala> def foo[A, B<:A](a:A, b:B) = (a,b)
foo: [A, B <: A](a: A, b: B)(A, B)

scala> foo(1,List(1,2,3))
res116: (Any, List[Int]) = (1,List(1, 2, 3))
```

함수에 타입 제한자

함수 타입매개변수를 타입제한자로 한정하면 실제 타입제한한 타입으로만 처리되므로 한정되어 처리된다.

```
scala> def bar[A,B](a:A, b:B)(implicit ev: B<:<A) =(a,b)
bar: [A, B](a: A, b: B)(implicit ev: B <:< A)(A, B)

scala> bar(1,List(1,2,3))
<console>:25: error: Cannot prove that List[Int] <:< Int.
       bar(1,List(1,2,3))
           ^


scala> bar(1,2)
res118: (Int, Int) = (1,2)
```

타입 정보 알아보기

typeOf 로 타입관계 알아보기

추상타입은 다양한 클래스로 해석할 수 있는 명세이다. 추상 타입은 보편적으로 전달받을 수 있는 즉 허용가능한 타입의 범위를 지정하기 위해 타입 매개변수로 사용된다.

A =:= B A는 B와 같아야 함

A <:< B A는 B의 하위 타입이어야 함

A <%< B A는 B로 볼 수 있어야 함

```
scala> import scala.reflect.runtime.universe._
import scala.reflect.runtime.universe._

scala> class A
defined class A

scala> class B extends A
defined class B

scala> typeOf[A]
res103: reflect.runtime.universe.Type = A

scala> typeOf[B]
res104: reflect.runtime.universe.Type = B

scala> typeOf[A] <:< typeOf[B]
res105: Boolean = false

scala> typeOf[B] <:< typeOf[A]
res106: Boolean = true
```

typeOf 를 이용해서 멤버 알아보기

위에서 지정한 C 클래스의 멤버 즉 메소드가 존재하는 지를 member 메소드로 확인한다.

```
scala> typeOf[C[A]]
res112: reflect.runtime.universe.Type = C[A]

scala> typeOf[C[A]].member(TermName("diff"))
res113: reflect.runtime.universe.Symbol = method diff
```