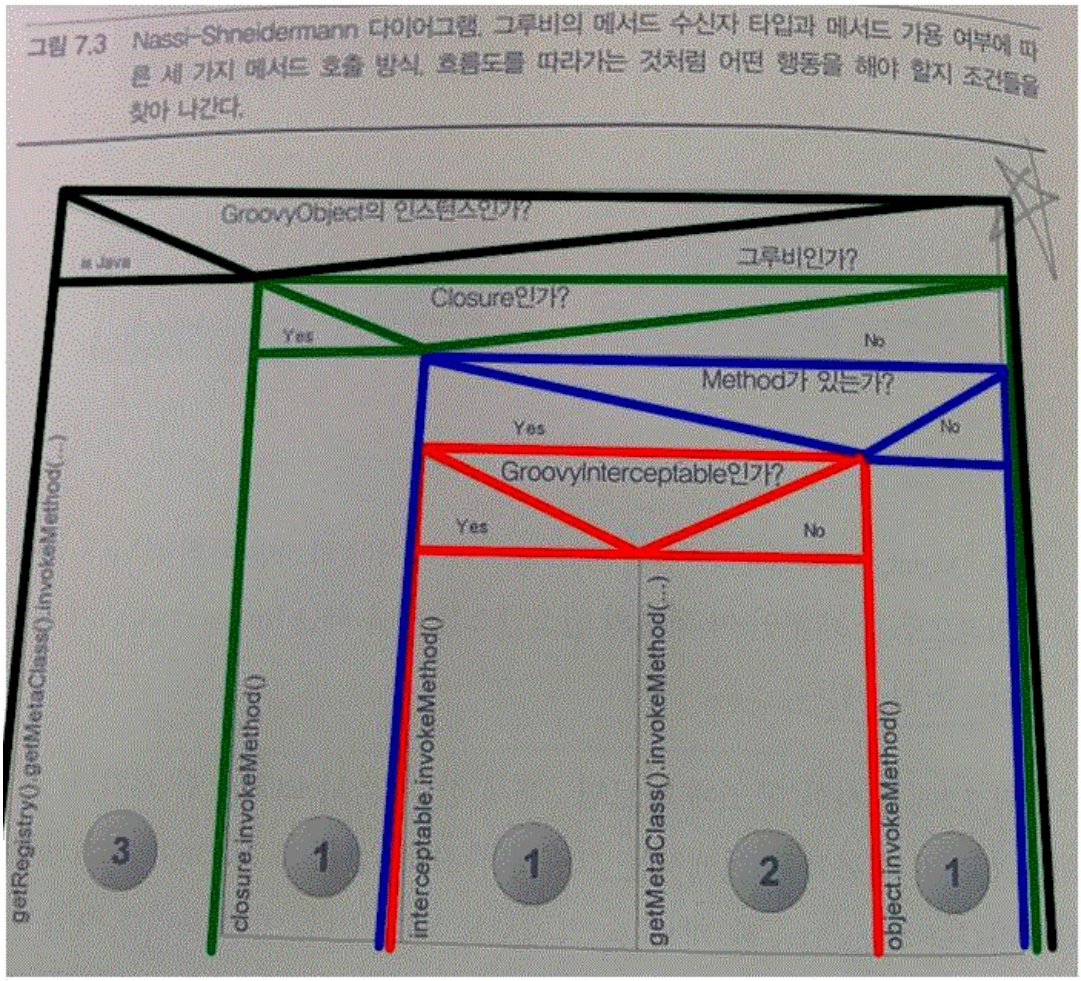
02.2주차.7장 그루비의 동적 객체지향-2

메서드 호출하기, 가로채기

Table of Contents

- 메서드 호출하기, 가로채기
- 26.GroovyInterceptable인터페이스를 구현해서 메서드의 호출
- 27.MetaClassRegistry에서 우리의 MetaClass를 바꿔치자

그루비가 메서드를 호출하는 자바 바이트 코드를 생성할 때는 (몇차례 전달 과정을 거친후에) 다음중 한가지 방식으로 작성한다
1. 클래스 자체적인 invokeMethod 구현 호출(내부적으로 다른 MetaClass로 연결될수있다)
2. 자신의 MetaClass이용 getMetaClass().invokeMethod()호출
3. MetaClassRegisty에 저장된 객체의 자료형에 대한 MetaClass를 이용



위그림은 Groovy in Action 책에서 발췌하였습니다

1. 모든 메서드 호출을 중간에 **가로챌(intercept) 수있다. 즉 모든 로깅(logging)이나 추적(tracing)을 가로채서 보안 관련 사항을 적용하고 트랜잭션 처리를 할수있다.**
2. 다른객체로 호출을 **전달할 (relay) 수있다 예를들어 감싸는 객체(wapper)가 자신이 처리하지 못하는 메서드를 감싸진(wrapped) 객체에게 전달하는 일이 가능하다**
* 이게 클로저가 동작하는 방식이다 클로저는 **메서드 호출을 자신의 대리인(delegate)에게 전달한다.**
3. 사실 다른 일을 하면서 겉으로 메서드인척 **가장할(pretend)수있다 예를들어 Html클래스는 body라는 메서드가 있는것처럼 보이면서 내부적으로는 print('body') 를수행할수도있다.**
* 이것이 바로 빌더(builder)의 동작방식이다 빌더들은 중첩된 구조를 만들기 위해 메서드가 있는 척한다.

가로채기(intecept) 전달하기(relay) 가장하기(pretend)등 몇가지 구현방법

1. GroovyObject의 invokeMethod를 구현 혹은 재정의해서 메서드를 가장하거나 전달할수있다 (정의해놓은 다른 메서드들은 평사시처럼 동작함)
2. GroovyObject의 invokeMethod를 구현 혹은 재정의하기 GroovyInterceptable인터페이스를 구현해서 메서드의 호출을 가로챈다.
3. MetaClass를 구현하고 GroovyObject의 setMetaClss를 호출한다.(넣는다)
4. MetaClass를 구현하고 MetaClassRegistry에 대상 클래스들을 등록한다(그루비와 자바 클래스가 모두 가능하다) 이방식은 ProxyMetaClass를 통해 지원된다.

일반적으로 invokeMethod를 재정의 또는 구현하는것은 도트메서드 "."메서드 이름 연산자를 재정의하는것과같다.

26.GroovyInterceptable인터페이스를 구현해서 메서드의 호출

```
package _26

import org.codehaus.groovy.runtime.StringBufferWriter
import org.codehaus.groovy.runtime.InvokerHelper

class Traceable implements GroovyInterceptable {                    //#1

    private int indent = 0

    Object invokeMethod(String name, Object args){                   //#3
        System.out.println ("\n" + '  '*indent + "before methodName : '$name'")
2       indent++
        def metaClass = InvokerHelper.getMetaClass(this)             //|#4
        def result = metaClass.invokeMethod(this, name, args)        //|#4
        indent--
        System.out.println ("\n" + '  '*indent + "after  methodName : '$name',  reuslt : $result ")
        return result
    }
}
class Whatever extends Traceable {                                   //#5
    int outer(){
1       return inner()
    }
    int inner(){
        return 1
    }
}

def traceMe = new Whatever() //#6

println 'traceMe.outer() :  '+traceMe.outer();

/*결과

before methodName : 'outer'

  before methodName : 'inner'

  after  methodName : 'inner',  reuslt : 1

after  methodName : 'outer',  reuslt : 1
traceMe.outer() :  1
*/
```

```
1  package _26
2
3
4  import org.codehaus.groovy.runtime.StringBufferWriter
5  import org.codehaus.groovy.runtime.InvokerHelper
6
7  class Traceable implements GroovyInterceptable{                //#1
8
9      private int indent = 0
10
11     Object invokeMethod(String name, Object args){              //#3
12         System.out.println ("n" + '  '*indent + "before methodName : '$name'")
13         indent++
14         def metaClass = InvokerHelper.getMetaClass(this)        //|#4
15         def result = metaClass.invokeMethod(this, name, args)   //|#4
16         indent--
17         System.out.println ("n" + '  '*indent + "after  methodName : '$name',  reuslt : $result ")
18         return result
19     }
```

```
}
class Whatever extends Traceable {                          //#5
    int outer(){
        return inner()
    }
    int inner(){
        return 1
    }
}

def traceMe = new Whatever() //#6

println 'traceMe.outer() :  '+traceMe.outer();

/*결과

before methodName : 'outer'

  before methodName : 'inner'

  after  methodName : 'inner',  reuslt : 1

after  methodName : 'outer',  reuslt : 1
traceMe.outer() :  1
*/
```

아쉬운점이있다면
1. GroovyObject 대해서 만 동작한다는점 따라서 다른 일반적인 자바클래스에서 쓸수없음
2. 대상클래스가 이미 상속을 받고있다면 적용할수없음

27.MetaClassRegistry에서 우리의 MetaClass를 바꿔치자

MetaClassRegistry에서 우리의 MetaClass를 바꿔치는 방법도 있을것이다.
이를위해 ProxyMetaClass라는 클래스가 있다.
이클래스는 기존 MetaClass의 기능과 더불어 Interceptor 를 이용한 가로채기 기능을 제공한다

interface Interceptor

```
package groovy.lang;
public interface Interceptor {
Object beforeInvoke(Object object, String methodName, Object[] arguments);
Object afterInvoke(Object object, String methodName, Object[] arguments, Object result);
boolean doInvoke();
}
```

```
package _27

class CustomInterceptor implements Interceptor{
    def beforeInvoke(Object object, String methodName, Object[] arguments) {        //메서드호출전
        println "beforeInvoke   object : ${object}, methodName : ${methodName}, arguments : ${arguments}"
        return null;
    }
    def afterInvoke( Object object, String methodName, Object[] arguments, Object result) {//메서드호출후
        println "afterInvoke   object : ${object}, methodName : ${methodName},  arguments : ${arguments}, result : ${result}"
        return result;
    }
    public boolean doInvoke() {
        return true;
    }
}

class Whatever1 {
    int outer(){
        return inner()
    }
    int inner(){
        return 1
    }
}

def log = new StringBuffer("n")
def tracer = new CustomInterceptor()                    //# implements Interceptor 구현한다.

def proxy = ProxyMetaClass.getInstance(Whatever1.class) //적용할 클래스를 선택한다.
proxy.interceptor = tracer                              //proxy에 인텁셉터를 등록한다. (내가만든거)
proxy.use {
    assert 1 == new Whatever1().outer()
}
/*결과
beforeInvoke   object : class _27.Whatever1, methodName : ctor, arguments : []
afterInvoke   object : class _27.Whatever1, methodName : ctor,  arguments : [], result : _27.Whatever1@e79839
beforeInvoke   object : _27.Whatever1@e79839, methodName : outer, arguments : []
beforeInvoke   object : _27.Whatever1@e79839, methodName : inner, arguments : []
afterInvoke   object : _27.Whatever1@e79839, methodName : inner,  arguments : [], result : 1
afterInvoke   object : _27.Whatever1@e79839, methodName : outer,  arguments : [], result : 1
*/
```