목차

BCI - The Power of Bytecode Instrumentation in Java ... 3
The Power of Byte Code Instrumentation in Java - Part1 ... 3
Byte Code Instrumentation이란? ... 3
Java Bytecode ... 3
BCI를 지원하는 라이브러리들 ... 4
The Power of Byte Code Instrumenation in Java - Part2 ... 5
ASM ... 5
The Power of Byte Code Instrumentation in Java - Part 3 ... 6
One Simple but Powerful Example of BCI ... 6
참조 ... 10
The Power of Byte Code Instrumentation in Java - Part 4 ... 10
One Simple but Powerful Example of BCI ... 10
-Xbootclasspath 옵션 사용하기 ... 12
java.lang.Exception에 내가 필요한 바이트 코드 삽입하기 ... 13
The Power of Byte Code Instrumentation in Java - Part 5 ... 18
시나리오 ... 18
java.net..Socket을 어떻게 변경할 것인가? ... 19
ASM을 이용해 java.net.Socket의 바이트 코드 변경하기 ... 21
The Power of Byte Code Instrumentation in Java- Part 6 ... 24
JavaAgent BCI의 구조 ... 25
JavaAgent의 간단한 샘플 ... 25
Manifest 파일 ... 27
JavaAgent의 실행 ... 28
JavaAgent와 JVMTI와의 관계. 그리고 JMX ... 28
Epilogue ... 29

BCI - The Power of Bytecode Instrumentation in Java

BCI(Bytecode Instrumenation)와 관련하여 잘 정리된 글

- The Power of Bytecode Instrumentation in Java - Part 1
- The Power of Bytecode Instrumentation in Java - Part 2
- The Power of Bytecode Instrumentation in Java - Part 3
- The Power of Bytecode Instrumentation in Java - Part 4
- The Power of Bytecode Instrumentation in Java - Part 5
- The Power of Bytecode Instrumentation in Java - Part 6

출처 : http://ukja.tistory.com(조동욱,엑셈)

The Power of Byte Code Instrumentation in Java - Part1

Byte Code Instrumentation이란?

Java에서 가장 원초적이고 강력한 프로그래밍 기법은?

나의 대답은 단연 BCI, 즉 Byte Code Instrumentation이다. (또는 Byte Code Insertion) Byte Code Instrumenation이란 Java의 Byte Code에 대해 직접 수정을 가해서, 소스 파일의 수정없이 원하는 기능을 부여하는 기법을 말한다.

이러한 특징 때문에 Java 프로파일러나 모니터링 툴들이 대부분 BCI 기능을 이용하고 있으며, BCI를 통해 모니터링 대상이 되는 어플리케이션의 수정없이 성능 측정에 필요한 요소들을 삽입할 수 있다. Bytecode를 직접 수정할 수 있기 때문에 BCI를 통해서 구현할 수 있는 기능은 그야말로 무궁무진하다고 할 수 있다.

이 블로그에서 [AOP(Aspected Oriented Programming) In Java]라는 주제로 블로깅이 진행 중인데, AOP를 구현하는 핵심 기술이 바로 BCI이다. AOP 컴포넌트들이 컴파일 시간이나 로딩 시간, 또는 런타임 시간에 Aspect와 Business Logic을 Weaving할 수 있는 이유가 바로 BCI, 즉 Java 바이트 코드를 직접 수정할 수있는 기술을 사용하기 때문이다.

따라서 만일 AOP를 어떤 식으로든 사용한 적이 있다면 이미 암묵적으로 BCI를 사용하고 있다고 할 수 있다.

요즘 Jennifer, Performizer와 같은 WAS 모니터링 툴들이 많은 인기를 얻고 있는데, 이들 제품들이 성능 데이터를 수집하기 위해 가장 보편적으로 사용하는 기술이 바로 BCI이다.앞으로 몇 차례에 걸친 연재에서 이런 툴들이 어떻게 BCI를 이용해 성능 데이터를 수집하는지 몇가지 예를 보게 될 것이다.

Java Bytecode

Java가 Bytecode라는 일종의 기계어(머신코드)를 사용한다는 것은 익히 알려진 사실이다.

전통적인 기계어가 특정 OS/하드웨어에 의존적인데 반해 Java Bytecode는 JVM(Java Virtual Machine)에

만 의존적이라는 중요한 차이가 있다. 따라서 JVM만 동일하다면 어떤 OS/하드웨어에서든 동일한 Bytecode가 구동가능하다. Java가 오늘날 지배적인 언어가 된 것은 바로 OS 중립적인 기계어인 Bytecode 때문이다.

아래에 간단한 Java Bytecode의 예가 있다.

```
public getValue()I
L0 (0)
LINENUMBER 28 L0
SIPUSH 1000
ISTORE 1
L1 (3)
LINENUMBER 29 L1
ILOAD 1
IRETURN
L2 (6)
LOCALVARIABLE this Lflowlite/io/ASMTest; L0 L2 0
LOCALVARIABLE value I L1 L2 1
MAXSTACK = 1
MAXLOCALS = 2
```

위의 약간 암호같은 Bytecode는 아래의 Java Source코드가 컴파일된 것이다.

```
public int getValue() {
    int value = 1000;
    return value;
}
```

다행히 Sun에서 JVM의 스펙을 정할 때 Bytecode의 문법을 명확하게 정의하기 때문에, 약간의 노력을 기울이면 Bytecode를 직접 읽고 쓸 수 있다.(사실은 많은 노력이 필요하다)

http://java.sun.com/docs/books/jvms/에서 Java Virtual Machine의 상세한 스펙을 얻을 수 있다. 이 문서를 참조하면 Java Class File의 포맷과 Class File을 이루는 Bytecode에 대한 상세한 정보를 얻을 수 있다. 하지만 이 문서를 실제로 보는 사람은 거의 없을 것으로 믿는다. ^^)

비록 Bytecode를 직접 읽고 쓰는 것이 이론적으로는 가능하지만, 대단히 성가시고 복잡하다. 이런 이유로 Bytecode를 쉽게 조작할 수 있는 컴포넌트를 개발하는 프로젝트들이 진행되었으며 그 결과로 현재는 다양한 라이브러리 중 마음에 드는 것을 선택할 수 있게 되었다.

BCI를 지원하는 라이브러리들

많은 오픈 소스 커뮤니티들이 Bytecode 조작을 가능하게 하는 라이브러리들을 제공하고 있다.

- ASM : Object Web에서 제공 http://asm.objectweb.org/
- BCEL : Apache 프로젝트에서 제공 http://jakarta.apache.org/bcel/
- SERP : Sourceforge 프로젝트에서 제공 http://serp.sourceforge.net/

- Javassist : JBoss 프로젝트에서 제공 http://www.csg.is.titech.ac.jp/~chiba/javassist/(놀랍게도 일본 사람이 개발… ㅠㅠ)

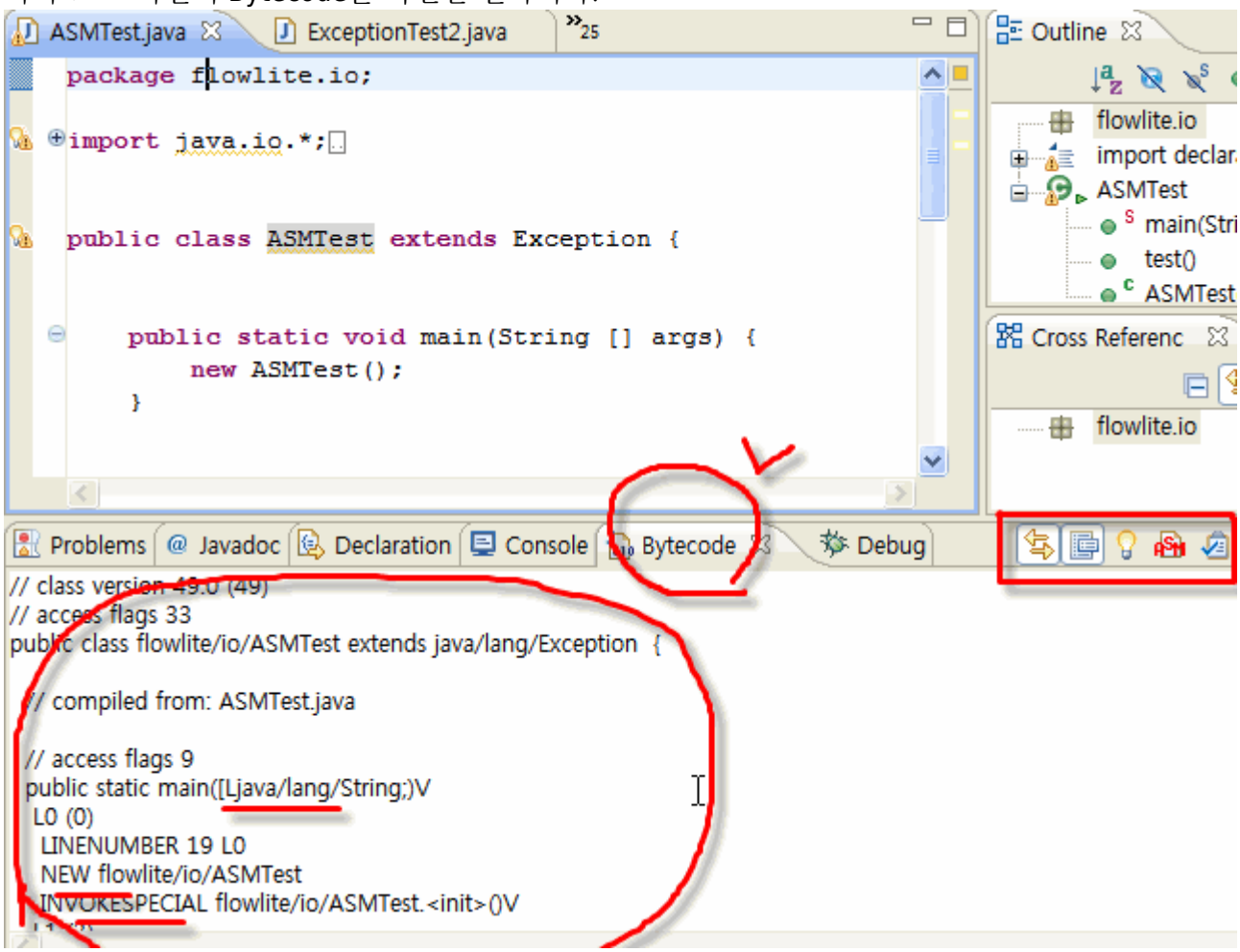
이 연재글에서는 ASM을 이용한 간단한 샘플들을 통해 BCI가 얼마나 강력한 프로그래밍 기법인지 공감하는 시간을 가질 것이다.

The Power of Byte Code Instrumenation in Java - Part2

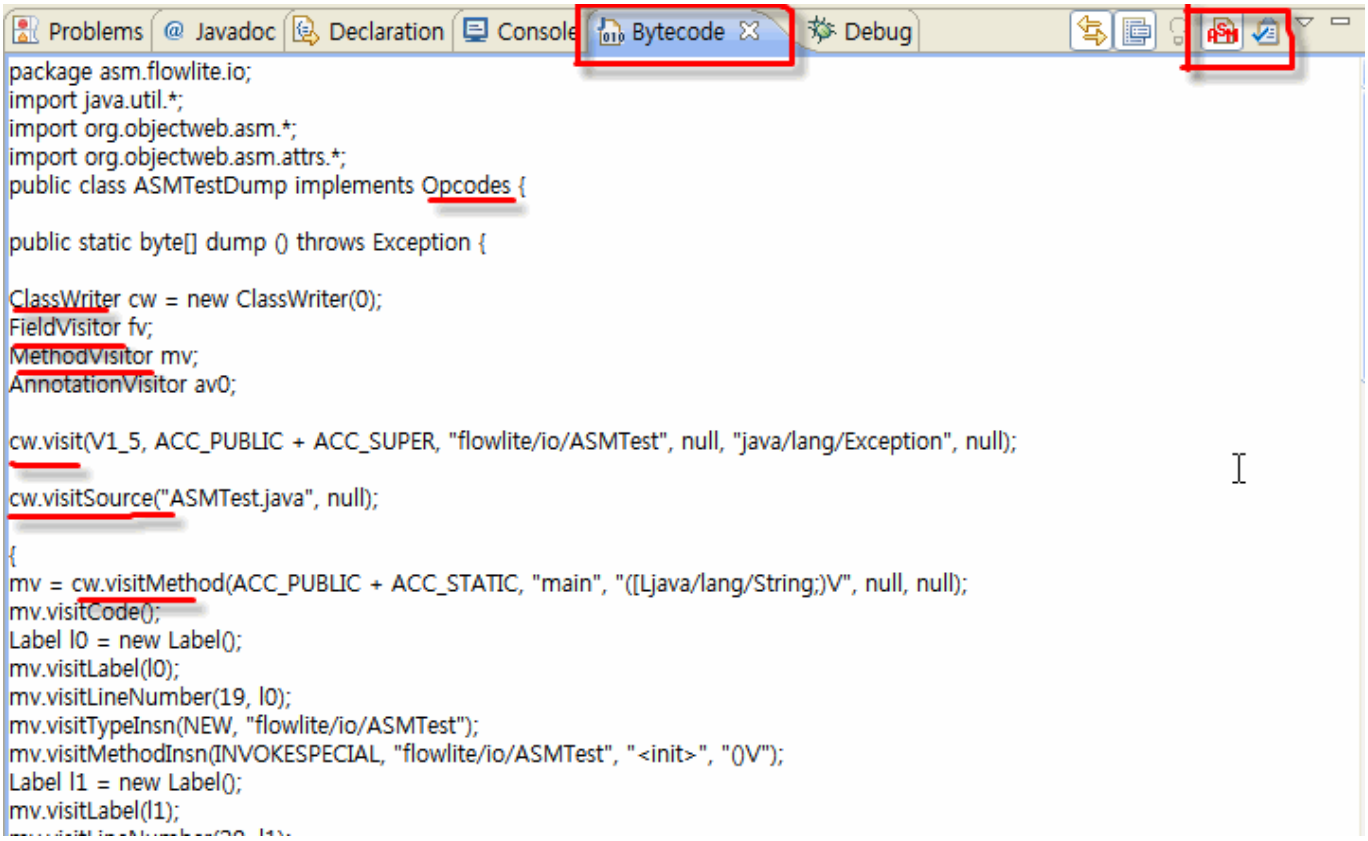
Byte Code Instrumentation(이하 BCI)의 예제를 논의하기 전에 우선 이 시리즈에서 사용할 BCI 라이브러리인 ASM의 사용법에 대해 간략하게 알아보자

ASM

ASM은 http://asm.objectweb.org/에서 설치에 필요한 라이브러리와 개발 도구(Eclipse Plugin - AMS Bytecode Outline Plugin)들을 다운받을 수 있다. Eclipse Plugin을 이용하면 ASM을 활용하는데필요한 학습 시간을 극적으로 줄일 수 있다. 아래 그림은 Eclipse ASM Bytecode Outline Plugin을 이용해서 특정 자바 소스 파일의 Bytecode를 확인한 결과이다.



Bytecode를 직접 본 적이 없다면 아마 상당히 생소할 것이다. 하지만 앞으로 몇 가지 예제를 거치다 보면 자연스럽게 Bytecode에 익숙해지게 된다. ASM Bytecode Outline Plugin의 가장 강력한 기능은 ASM화된코드를 손쉽게 확인할 수 있다는 것이다. 아래와 같이 [ASM]라는 이름의 아이콘을 선택하면 흉물스런 Bytecode가세련된 Java Code로 변환되는 것을 확인할 수 있다.



ASM화된 코드(ASMified Code)란 ASM 라이브러리가 제공하는 API를 이용해서 Bytecode를 생성하는 코드를 말한다. 이 ASM화된코드를 적절히Copy/Paste/Edit해서 필요한 거의 모든 Bytecode를 직접 생성하고 조작할 수 있다. 굉장하지 않은가? 나의 느낌은 전율 그 자체였다. 이 원리를 이용하면 어떠한 Java 어플리케이션이라도 내가원하는 코드를 어플리케이션의 코드 변경없이 삽입할 수있다. 응용처는 그야말로 무궁무진하다. 이런 강력함때문에 Java Monitor/Profiler/Analyzer 등 성능과 관련된 대부분의 툴들이 어떤 식으로든 BCI를 이용하고 있다. 이런 기능의 예를 앞으로 다양하게 살펴볼 것이다. ASM 라이브러리에 대한 자세한 설명은 http://asm.objectweb.org/에서 제공하는 매뉴얼과 Java Doc을 통해 얻을 수 있다. 반드시 매뉴얼을 읽어볼 것을 권장한다. ASM은 Visitor Pattern을 이용해 구현되어 있어서 직관적인 프로그래밍이 가능하다는 장점이 있다.

The Power of Byte Code Instrumentation in Java - Part 3

One Simple but Powerful Example of BCI

ASM 라이브러리를 이용한 Byte Code Instrumentation의 아주 간단한 예제를 하나 작성해보자.

비록 매우 심플한 예제이지만, 실제로는 매우 강력한 방법이다.

이 방법에 익숙해지면 아마 그 용도의 무궁무진함에 놀라게 될 것이다.

이런 질문에 한번 답해보자.

“ 우리 회사에서 사용 중인WAS Application에서 발생하는 Exception을 체계적으로 수집하고 정리하고 싶다. 그 Exception이 Catch되어서 처리되고 있는지에 무관하게…. ”

(사실은 이 문제에 대해 블로그의 “Aspected Oriented Programming in Java”에서 다룬 바 있다)

대부분의 사람들이 본능적으로 다음과 같은 대답을 떠올린다.

```
public static void main(String[] args) {

    try {
        ExceptionTest2.doException(true);
    } catch(Exception ex) {

        // Do something about exceptions...
        // ExceptionManager.recordException(ex);
    }
```

즉, Application에서 사용 중인 모든 소스 코드에 try catch 구문을 사용해 Exception을 Catch해서 적절히 조작하는 것이다. 이 접근법은 다음과 같은 문제점들이 있다.

- 그 많은 소스들을 언제 다 바꿀 것인가?
- 내가 직접 작성하지 않은 Java Core Library(java.*, javax.*, ...)나 3rd Party Library는 어떻게 할 것인가?
- Exception 처리 정책이 바뀌었을 때는 또 어떻게 할 것인가?
- 무엇보다 엄청나게 지저분해질 코드는 또 어떻게 할 것인가?

Business Logic을 처리하는 코드에 Exception 수집을 위한 코드를 넣는 것은 가장 위험하고 비효율적인 방법임이 분명하다.

이런 문제를 해결하기 위해 AOP와 같은 방법론이 등장했을 정도로…

이 문제를 해결하는 가장 간단한 방법을 생각해보자….

나의 머리 속에 떠오른 방법은 이것이다.

- “java.lang.Excetpion” 클래스가 모든 Exception의 부모 클래스 아닌가? “
- “ 즉, 어떤 종류의Exception이라도 반드시 java.lang.Exception이 제공하는 생성자와 메소드를 공유하고 있다 ”
- “ 따라서java.lang.Exception의 생성자에 Exception 수집 코드를 삽입하면 모든 것이 해결된다 ”

빙고!!

문제는 java.lang.Exception 클래스의 생성자를 어떻게 조작하느냐는 것이다.

여기가 바로 BCI, 즉 Byte Code Instrumentation이 등장하는 곳이다.

우선 java.lang.Exception 클래스가 어떤 코드로 이루어져 있는지 Java Decompiler를 이용해 코드를 살펴보자.

Decompiler를 통해서 Exception 클래스의 소스 파일은 다음과 같다.

```
package java.lang;

// Referenced classes of package java.lang:
// Throwable, String

public class Exception extends Throwable {

    public Exception() {
```

```
    }

    public Exception(String s) {
        super(s);
    }

    public Exception(String s, Throwable throwable) {
        super(s, throwable);
    }

    public Exception(Throwable throwable) {
        super(throwable);
    }

    static final long serialVersionUID = 0xd0fd1f3e1a3b1cc4L;
}
```

ASM Bytecode Outline Plugin을 통해서 본 Byte Code는 아래와 같다.

```
/ class version 49.0 (49)
// access flags 33
public class Exception extends Throwable {


// access flags 24
final static long serialVersionUID = -3387516993124229948

// access flags 1
public () : void
ALOAD 0
INVOKESPECIAL Throwable.() : void
RETURN
MAXSTACK = 1
MAXLOCALS = 1

// access flags 1
public (String) : void
ALOAD 0
ALOAD 1
INVOKESPECIAL Throwable.(String) : void
RETURN
MAXSTACK = 2
MAXLOCALS = 2

// access flags 1
public (String,Throwable) : void
ALOAD 0
ALOAD 1
```

```
ALOAD 2
INVOKESPECIAL Throwable.(String,Throwable) : void
RETURN
MAXSTACK = 3
MAXLOCALS = 3

// access flags 1
public (Throwable) : void
ALOAD 0
ALOAD 1
INVOKESPECIAL Throwable.(Throwable) : void
RETURN
MAXSTACK = 2
MAXLOCALS = 2
}
```

만일 Exception 클래스의 생성자에서 다음과 같은 코드가 삽입되도록 한다면?

```
public class Exception extends Throwable {

    public Exception() {
        ExceptionCallBack.exceptionOccurred(this);
    }

    public Exception(String s) {
        super(s);
        ExceptionCallBack.exceptionOccurred(this);
    }

    public Exception(String s, Throwable throwable) {
        super(s, throwable);
        ExceptionCallBack.exceptionOccurred(this);
    }

    public Exception(Throwable throwable) {
        super(throwable);
        ExceptionCallBack.exceptionOccurred(this);
    }

    static final long serialVersionUID = 0xd0fd1f3e1a3b1cc4L;
}
```

이렇게 되면 어떤 Exception이 발생하든지 항상 ExceptionCallBack.exceptionOccurred(this) 코드에 의해 수집이 이루어진다.심지어 Java Core Library를 사용하는 과정에서 내부적으로 발생하는 Exception도 다 수집할 수 있다.

Part4에서 ASM을 이용해 java.lang.Exception 클래스의 바이트 코드를 직접 수정하는 예제를 보게 될 것이다.

참조

이런 반문을 할 지 모르겠다. “java.lang.Exception” 클래스를 디컴파일한 소스를 직접 수정해서 새로운 java.lang.Exception 클래스를 직접 만들면 안되나? ”

가능한 방법이다. 하지만 이런 문제점이 있다.

JVM의 Version이나 Vendor에 따라 소스는 모두 다를 수 있다. 따라서 타겟이 바뀔 때마다 디컴파일을 수행해서 소스를 생성한 후 작업을 해야 한다. 디컴파일러가 모든 클래스 파일을 다 완벽하게 디컴파일해 내는 것은 아니다. 이런 이유들 때문에 소스 코드를 직접 수정하는 방법은 정말 특별한 경우가 아니면 권장되지 않는다. 더구나 Java Core Library의 소스 코드를 변경해서 사용하는 것은 법적(?)인 문제를 일으킬 수도 있다.

The Power of Byte Code Instrumentation in Java - Part 4

One Simple but Powerful Example of BCI

Part3에서 언급한 봐와 같이 java.lang.Exception 클래스의 바이트 코드를 직접 수정함으로써 모든 Exception의 발생을 캡쳐할 수 있다. Exception 클래스의 생성자들에서 다음과 같은 코드를 수행하게끔 바이트 코드를 수정하면 된다.

```
public Exception(String s)
{
    super(s);
    ExceptionCallBack.exceptionOccurred(this);
}
```

ASM이 제공하는 Library를 이용하면 이 작업을 매우 손쉽게 수행할 수 있다.

ASM 라이브러리의 사용법을 상세히 설명하는 것은 이 블로그의 범위를 벗어나며 http://asm.objectweb.org 에서 제공하는 매뉴얼과 문서에 이미 상세하게 설명이 되어 있다.

아래 소스 코드는 java.lang.Exception 클래스를 ASM Library를 이용해 읽고(Read),그대로(변형없이) Write하는 예제이다.

```
package flowlite.exception2;

import java.io.FileOutputStream;

import org.objectweb.asm.*;
import org.objectweb.asm.commons.*;

public class ExceptionTransformer implements Opcodes {

    // Convert class
```

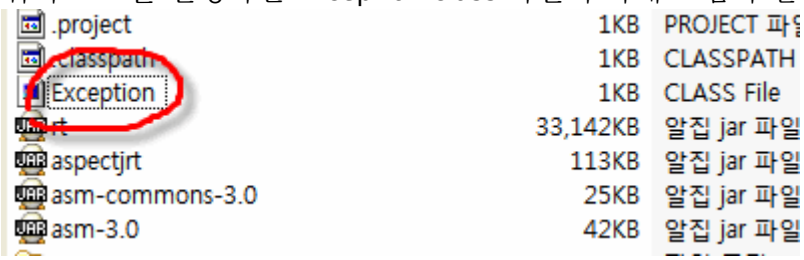
```
    public void transform(String newClassName) throws Exception {

        System.out.println("Starting transformation of
java.lang.Exception...");

        // Reader
        ClassReader reader = new ClassReader("java.lang.Exception");
        // Writer

        ClassWriter writer = new ClassWriter(ClassWriter.COMPUTE_MAXS);

        // Class Adapter
        ClassAdapter adapter = new ExceptionClassAdapter(writer);

        reader.accept(adapter, ClassReader.SKIP_FRAMES);

        byte[] b = writer.toByteArray();
        FileOutputStream fos = new FileOutputStream(newClassName +
".class");
        fos.write(b);
        fos.flush();

    }

    public static void main(String[] args) {
        try {
            String newClassName = "Exception";
            if (args.length >= 1)
                newClassName = args[0];
            ExceptionTransformer fit = new ExceptionTransformer();
            fit.transform(newClassName);
        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }
}

class ExceptionClassAdapter extends ClassAdapter implements Opcodes {

    public ExceptionClassAdapter(ClassVisitor visitor) {
        super(visitor);

    }

}
```

위의 소스 코드를 보면 바이트 코드 변환은 다음과 같은 과정을 통해 이루어진다는 것을 알 수 있다.

- ClassReader 객체를 이용해 원래 클래스의 바이트 코드를 읽어들인다.
- ClassAdapter 객체를 이용해 바이트 코드를 변경한다.

- ClassWriter 객체를 이용해 변경된 바이트 코드를 얻는다.
- 변경된 바이트 코드는 파일로 저장하거나 ClassLoader에게 넘겨 준다.

위의 코드를 실행하면 Exception.class 파일이 아래 그림과 같이 파일시스템에 저장된다.



-Xbootclasspath 옵션 사용하기

문제는 어떻게 하면 원래 JVM의 rt.jar 에서 제공하는 Exception 클래스 파일이 아닌 내가 생성한 Exception 클래스 파일을 쓰게 하느냐이다. rt.jar 파일을 직접 변경시키는 것은 매우 위험하고, 법적으로도 (이건 농담이 아님) 문제가 될 수 있기 때문에 권장되지 않는다.

답은 `-Xbootclasspath` 옵션을 이용하는 것이다. “java -X” 명령을 수행하면 다음과 같은 결과를 확인할 수 있다.

```
C:₩Documents and Settings₩mulder>java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                      append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by ;>
```

즉, `-Xbootclasspath/p:<`내가 작성한 `Exception Class`의 패스`>`를 지정하면 JVM은 rt.jar보다 먼저(prepend) 내가 작성한 Exception Class 파일을 읽어 들인다. 이렇게 함으로써 rt.jar 파일에 대한 수정을 가하지 않아도 된다.

우선 아래와 같은 Test Class를 작성한 후

```
public class ExceptionTest2 {

    public static void doException(boolean bException) throws
RuntimeException {
        if (bException)
            throw new RuntimeException("test");

    }

    public static void main(String[] args) {

        try {
            ExceptionTest2.doException(true); // <-- 여기서 Exception발생
```

```
        } catch (Exception ex) {
        }

        try {
            ExceptionTest2.doException(false);
        } catch (Exception ex) {
        }
    }
}
```

아래와 같이 -Xbootclasspath 옵션을 이용해서 수행한다. (./converted_classes/java/lang 디렉토리에 내가 만든 Exception Class가 있다…)

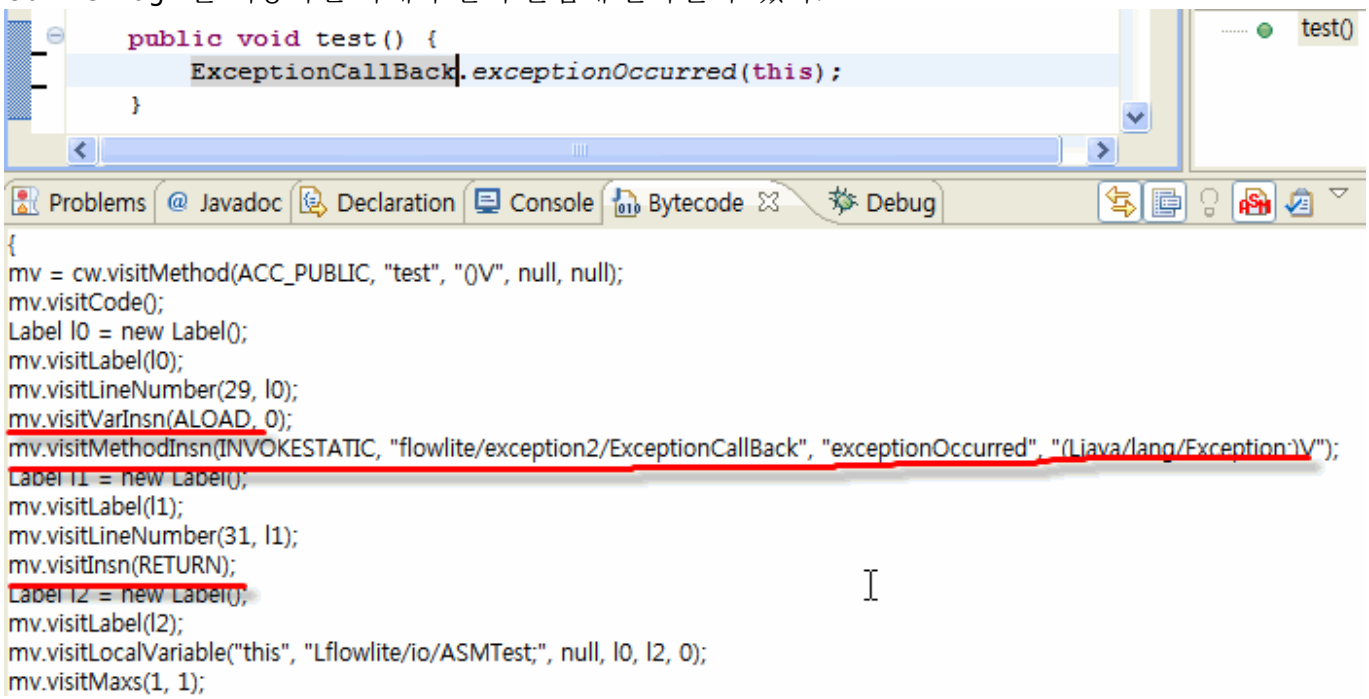
```
java -Xbootclasspath/p:./converted_classes ExcetpionTest2
```

java.lang.Exception에 내가 필요한 바이트 코드 삽입하기

우리가 java.lang.Exception 클래스의 생성자에 삽입하고자 하는 코드는 다음과 같다.

```
ExceptionCallBack.exceptionOccurred(this);
```

따라서 가장 먼저 알아야 할 것은 이 코드에 해당하는 바이트 코드가 무엇이냐이다. ASM Bytecode Outline Plugin을 이용하면 아래와 같이 손쉽게 알아낼 수 있다.



위의 그림에서 빨간 색으로 밑줄이 그어진 부분이 우리가 필요로 하는 부분이다. 간략하게 설명하면…

- mv.visitVarInsn(ALOAD, 0) : 0번째 변수는 항상 나 자신(this)를 가리킨다.
- mv.visitMethodInsns(INVOKESTATIC, [className], [methodName], [methodDescription]) : 메소드를 호출한다.
    - INVOKESTATIC은 Static Method를 실행하겠다는 의미이다. methodDescription은 메소드의

파라미터와 리턴값을 의미한다.
  - (Ljava/lang/Exception;) 은 java.lang.Exception 클래스가 호출된 메소드의 파라미터 타입이라는 것을 의미한다.
  - V는 리턴타입이 void라는 것을 의미한다.

이제 처리해야할 마지막 작업은 위와 같은 형태의 코드를 java.lang.Exception 클래스의 생성자마다 삽입해주는 것이다. 아래에 샘플 코드가 있다.

```
package flowlite.exception2;

import java.io.FileOutputStream;

import org.objectweb.asm.*;
import org.objectweb.asm.commons.*;

public class ExceptionTransformer implements Opcodes {

    // Convert class
    public void transform(String newClassName) throws Exception {

        System.out.println("Starting transformation of
java.lang.Exception...");

        ClassReader reader = new ClassReader("java.lang.Exception");
        ClassWriter writer = new ClassWriter(ClassWriter.COMPUTE_MAXS);
        ClassAdapter adapter = new ExceptionClassAdapter(writer);

        reader.accept(adapter, ClassReader.SKIP_FRAMES);

        byte[] b = writer.toByteArray();
        FileOutputStream fos = new FileOutputStream(newClassName +
".class");
        fos.write(b);
        fos.flush();

    }

    public static void main(String[] args) {
        try {
            String newClassName = "Exception";
            if (args.length >= 1)
                newClassName = args[0];
            ExceptionTransformer fit = new ExceptionTransformer();
            fit.transform(newClassName);
        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }
```

```
}

class ExceptionClassAdapter extends ClassAdapter implements Opcodes {

    public ExceptionClassAdapter(ClassVisitor visitor) {
        super(visitor);

    }

    public MethodVisitor visitMethod(int access, String name, String desc,
String sig, String[] exes) {
        MethodVisitor mv = super.visitMethod(access, name, desc, sig, exes);

        if (name.equals("")) { // Constructor
            System.out.println("Redefine Constructor...");
            ExceptionConstructorAdviceAdapter ecaa = new
ExceptionConstructorAdviceAdapter(access, name, desc, mv);
            return ecaa;
        }

        return mv;

    }
}

class ExceptionConstructorAdviceAdapter extends AdviceAdapter {

    public ExceptionConstructorAdviceAdapter(int access, String name, String
desc, MethodVisitor mv) {
        super(mv, access, name, desc);
    }

    protected void onMethodEnter() {

    }

    protected void onMethodExit(int opcode) {
        if (opcode == RETURN) {
            mv.visitVarInsn(ALOAD, 0);
            mv.visitMethodInsn(INVOKESTATIC,
"flowlite/exception2/ExceptionCallBack", "exceptionOccurred",
"(Ljava/lang/Exception;)V");
            mv.visitEnd();
        }
    }

}
```

위의 소스 코드에 대한 설명은 다음과 같다.

- ClassReader를 이용해 rt.jar의 원본 java.lang.Exception 클래스를 읽는다.

- ExceptionClassAdapter(extends ClassAdapter)객체를 이용해 원본 바이트 코드에 대한 변경을 시도한다.
    - ExceptionClassAdpater는 visitMethod를 이용해 Exception 클래스의 생성자에 대한 변경을 시도한다.
    - ClassAdapter.visitMethod는 바이트 코드에서 특정 메소드에 대한 정의가 시작될 때 호출된다. 따라서 visitMethod를 재정의함으로써 원하는 메소드를 우리 입맞에 맞게 수정할 수 있다.
    - 생성자(Constructor)의 메소드 이름은 항상 이다.
- ExceptionClassAdapter.visitMethod 메소드는 만일 메소드가 생성자()이면 ExceptionConstructorAdviceAdapter(extends AdviceAdapter) 객체를 이용해 생성자에 대한 재정의를 시도한다.
    - AdviceAdpater 객체는 onMethodEnter, onMethodExit(opcode)라는 두 메소드를 제공한다.
    - onMethodEnter는 메소드 시작 시점에, onMethodExit는 메소드 리턴 직전 시점에 수행될 코드를 정의할 공간을 마려해준다.
    - 즉, ExceptionConstructorAdviceAdapter 객체는 이 두 메소드를 재정의함으로써 메소드의 시작/끝 시점에 자신이 원하는 코드를 삽일할수 있다.

위의 설명과 같이 하나의 ClassAdapter와 AdviceAdpater를 이용해서 우리가 원하는 작업을 수행할 수 있다.

이 예제에서는 onMethodExit를 이용해서 생성자가 종료되기 직전에 우리가 원하는 바이트 코드를 삽입한다. `ExceptionConstructorAdviceAdapter.onMethodExit` 메소드를 좀 더 자세히 살펴보자.

```
protected void onMethodExit(int opcode) {
    if(opcode == RETURN) {
            mv.visitVarInsn(ALOAD, 0);
            mv.visitMethodInsn(INVOKESTATIC,
"flowlite/exception2/ExceptionCallBack", "exceptionOccurred",
"(Ljava/lang/Exception;)V");
            mv.visitEnd();
    }
}
```

위의 코드는 생성자가 정상적으로 리턴될 때`(opcode == RETURN)` `“ExceptionCallBack.exceptionOccurred(this)”` 코드를 수정하게끔 바이트 코드를 변경한다.

ExceptionTransformer를 수행한 후 생성된 Exception Class의 바이트 코드를 보면 아래와 같이 원하는 결과를 얻었음을 알 수 있다.

```
// class version 49.0 (49)
// access flags 33
public class Exception extends Throwable {

// access flags 24
final static long serialVersionUID = -3387516993124229948

// access flags 1
public () : void
ALOAD 0
```

```
INVOKESPECIAL Throwable.() : void
ALOAD 0
INVOKESTATIC ExceptionCallBack.exceptionOccurred(Exception) : void
RETURN
MAXSTACK = 1
MAXLOCALS = 1

// access flags 1
public (String) : void
ALOAD 0
ALOAD 1
INVOKESPECIAL Throwable.(String) : void
ALOAD 0
INVOKESTATIC ExceptionCallBack.exceptionOccurred(Exception) : void
RETURN
MAXSTACK = 2
MAXLOCALS = 2

// access flags 1
public (String,Throwable) : void
ALOAD 0
ALOAD 1
ALOAD 2
INVOKESPECIAL Throwable.(String,Throwable) : void
ALOAD 0
INVOKESTATIC ExceptionCallBack.exceptionOccurred(Exception) : void
RETURN
MAXSTACK = 3
MAXLOCALS = 3

// access flags 1
public (Throwable) : void
ALOAD 0
ALOAD 1
INVOKESPECIAL Throwable.(Throwable) : void
ALOAD 0
INVOKESTATIC ExceptionCallBack.exceptionOccurred(Exception) : void
RETURN
MAXSTACK = 2
MAXLOCALS = 2
}
```

이제 마지막 단계로 ExceptionCallBack.exceptionOccurred의 클래스와 메소드를 다음과 같이 정의한다.

```
package flowlite.exception2;

public class ExceptionCallBack {

    public static void exceptionOccurred(Exception ex) {
```

```
        System.out.println("Oops~ " + ex + " has occurred...");
        ex.printStackTrace();
    }
}
```

이 작업을 마무리하고, 다음과 같이 -Xbootclasspath 옵션을 이용해서 Exception을 발생시키는 ExceptionTest2를 수행해보자. (새로 생성된 Exception.class 파일을 converted_classes/java/lang 디렉토리로 카피하는 것은 기본!!!)

```
java -Xbootclasspath/p:./converted_classes ExcetpionTest2
```

이제 Exception이 발생할 때마다 ExceptionCallBack.exceptionOccurred가 수행되므로 다음과 같은 수행 결과가 나타난다.

```
Oops~ java.lang.RuntimeException: test has occurred...
java.lang.RuntimeException: test
        at flowlite.exception2.ExceptionTest2.doException(ExceptionTest2.java:15)
        at flowlite.exception2.ExceptionTest2.main(ExceptionTest2.java:22)
```

놀랍지 않은가?

바이트 코드에 대한 아주 간단한 조작만으로도 모든 Exception 발생에 대한 처리 로직을 추가할 수 있다.

이번 예제를 통해 ASM을 이용한 BCI의 개념이 이해되었기를 바라며, 좀 더 복잡한 예제를 통해 더 의미있는 논의를 해보기로 한다.

The Power of Byte Code Instrumentation in Java - Part 5

Part4에서 Byte Code Instrumentation(이하 BCI)을 사용하는 간단한 예제를 살펴봤는데, 이제 보다 복잡한 예제를 통해 BCI가 어떤 정도의 위력을 가지고 있는지 알아보자.

시나리오

우리가 풀어야 할 문제는 다음과 같다.

> “ 특정Java 어플리케이션이 Socket을 통해 읽고 쓰는 데이터를 추적하고 싶다. 즉, 어떤 Socket을 통해 어떤 쓰레드가 얼마나 I/O를 일으키는가를 추적하고 싶다. ”

BCI를사용하지 않고 이것을 수행하는 방법은 OS에서 제공하는 Network Monitoring Tool이나 API를 사용해서 Java Application의 특정 Port를 모니터링하는 것이다. 하지만 이러 Tool이나 API를 이용해 우리 입맛에 맞는 분석 데이터를 만드는것은 대단히 어렵거나 혹은 불가능하다.

하지만 BCI를 사용하면 이 작업을 매우 손쉽게 처리할 수 있다. 어떻게 이것이 가능한가?

java.net..Socket을 어떻게 변경할 것인가?

우선 Socket Class에서 우리가 필요로 하는 API를 보자.

```
public InputStream getInputStream()throws IOException
public OutputStream getOutputStream()throws IOException
```

위의 두 메소드는 Socket을 통해서 데이터를 읽고 쓰는 InputStream과 OutputStream을 얻어온다. 논의를 간단하게 하기 위해 InputStream만을 살펴 보자. InputStream Class에서 우리가 관심을 가지는 API는 다음과 같다.

```
public int read(byte[]b,intoff,intlen)throws IOException
```

위의 메소드는 InputStream으로부터 특정 데이터(b[])를 읽어 들이고 실제로 읽어들인 바이트 수를 return하는 역할을 한다. 이 메소드에 우리가 원하는 모든 것이 있는 셈이다.

BCI를 이용해서 InputStream.read 메소드를 캡쳐 혹은 변경하는 방법은 무궁무진한데, 여기서 필자는 다음과 같은 방법을 사용하고자 한다.

- 우선 Socket.getInputStream() 메소드의 이름을 orig$getInputStream으로 변경한다.
- Socket.getInputStream() 메소드를 다음과 같이 재정의한다.

```
public InputStream getInputStream() {
  // 원래의 getInputStream(이름이 __orig$getInputStream__으로 바뀐)을 호출해서
InputStream을 얻어온다.
    InputStream is = __orig$getInputStream__();
  // 이 InputStream을 이용해서 나만의 InputStream을 만들어준다.
    FlowLiteSocketInputStream fsis = new FlowLiteSocketInputStream(this,
is);

    return fsis;
}
```

이렇게 함으로써 Socket과 관련된 InputStream에서 발생하는 모든 액션을 캡쳐할 수 있다. JDK가 제공하는 Socket용 InputStream을 대신할 나만의 InputStream, 즉 FlowLiteSocketInputStream은 다음과 같이 InputStream을 상속받으며, 각 메소드는 모든 액션을 캡쳐할 수 있도록 구현된다.

```
package java.net;

public class FlowLiteSocketInputStream extends InputStream {

    Socket s = null;
    InputStream is = null;
```

```
    public FlowLiteSocketInputStream(Socket s, InputStream is) {
        this.s = s;
        this.is = is;
        SocketIOCallBack.createCalled(this);
    }

    public int read() throws IOException {
        int len = is.read();
        SocketIOCallBack.readCalled(this, 4);

        return len;

    }

    public int read(byte[] b) throws IOException {
        int len = is.read(b);
        SocketIOCallBack.readCalled(this, len);
        return len;
    }

    public int read(byte[] b, int off, int len) throws IOException {
        int len2 = is.read(b, off, len);
        SocketIOCallBack.readCalled(this, len2);
        return len2;
    }

    public int available() throws IOException {
        return is.available();
    }

    public void close() throws IOException {
        is.close();
    }

    public void mark(int readlimit) {
        is.mark(readlimit);
    }

    public boolean markSupported() {
        return is.markSupported();
    }

    public Socket getSocket() {
        return this.s;
    }
}
```

SocketIOCallBack 클래스는 FlowLiteInputStream으로부터 콜백받은 결과를 저장하고 그 결과를 보여주게끔 구현하면 된다.

ASM을 이용해 java.net.Socket의 바이트 코드 변경하기

ASM 라이브러리가 제공하는 라이브러리를 이용하면 우리가 원하는 대로 java.net.Socket 클래스의 바이트 코드를 변경할 수 있다. 아래에 그 샘플 코드가 있다.

```
package flowlite.net;

import java.io.*;
import org.objectweb.asm.*;
import org.objectweb.asm.commons.*;

/**
 * Net io class transformer. Execute byte code transformation to convert
 * NetInputStream class You must convert native java.io.Socket class using
this
 * class
 *
 * The technique is the most powerful. So keep in mind~~
 *
 * Must use ASM 3.0 library
 *
 * @history 2007/07/17Dongwook ChoInitial Coding
 *
 */
public class SocketTransformer implements Opcodes {

    public SocketTransformer() {

    }

    // Convert class
    public void transform(String newClassName) throws Exception {
        System.out.println("Starting transformation of java.net.Socket...");
        // Prepared reader, writer, adapter

        ClassReader reader = new ClassReader("java.net.Socket");
        ClassWriter writer = new ClassWriter(ClassWriter.COMPUTE_MAXS +
ClassWriter.COMPUTE_FRAMES);
        ClassAdapter adapter = new SocketClassAdapter(writer);

        reader.accept(adapter, ClassReader.SKIP_DEBUG);

        byte[] b = writer.toByteArray();
        FileOutputStream fos = new FileOutputStream(newClassName +
".class");
        fos.write(b);
        fos.flush();
    }

    public static void main(String[] args) {
```

```
        try {
            String newClassName = "Socket";
            if (args.length >= 1)
                newClassName = args[0];
            SocketTransformer fit = new SocketTransformer();
            fit.transform(newClassName);
        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }
}

// Socket Class 변환기
class SocketClassAdapter extends ClassAdapter implements Opcodes {

    public SocketClassAdapter(ClassVisitor cv) {
        super(cv);
    }

  // 각메소드를 방문하면서 필요하면 변경한다.
    public MethodVisitor visitMethod(int access, String name, String
descriptor, String signature,
            String[] exceptions) {

        if (name.equals("getInputStream")) {
            // getInputStream을 __orig$getInputStreram__ 으로 변경한다.

            System.out.println("Rename getInputStream to
__orig$getInputStream__");
            MethodVisitor mv = cv.visitMethod(ACC_PUBLIC,

                    "__orig$getInputStream__", "()Ljava/io/InputStream;",
null, new String[] { "java/io/IOException" });
            mv.visitCode();
            mv.visitEnd();

            return mv;
        }

        return super.visitMethod(access, name, descriptor, signature,
exceptions);
    }

  // 새로운getInputStream 메소드를 추가한다.

    public void visitEnd() {
        MethodVisitor mv = cv.visitMethod(ACC_PUBLIC, "getInputStream",
"()Ljava/io/InputStream;",
```

```
                null, new String[] { "java/io/IOException" });

        mv.visitCode();
        mv.visitTypeInsn(NEW, "java/net/FlowLiteSocketInputStream");
        mv.visitInsn(DUP);
        mv.visitVarInsn(ALOAD, 0);
        mv.visitVarInsn(ALOAD, 0);

    // 원래메소드를 호출하고,
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/net/Socket",
"__orig$getInputStream__", "()Ljava/io/InputStream;");
        mv.visitMethodInsn(INVOKESPECIAL,
"jav`a/net/FlowLiteSocketInputStream",

                "", "(Ljava/net/Socket;Ljava/io/InputStream;)V");
        mv.visitInsn(ARETURN);
        mv.visitMaxs(4, 1);
        mv.visitEnd();

    }
}
```

아래 결과는 위에서 구현한 FlowLiteInputStream, Socket, SocketIOCallBack을 이용해 Socket 통신을 통해 주고받는 데이터를 보여주고 있다.

```
[Socket Info] Thread id = 20, Host name =localhost:10583, Status = 1, Access
time = Fri Aug 10 18:11:06 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Fri Aug 10
18:11:06 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Fri Aug 10
18:11:06 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Fri Aug 10
18:11:06 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Fri Aug 10
18:11:06 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Fri Aug 10
18:11:06 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Fri Aug 10
18:11:06 KST 2007
....

[Net IO], Thread id = 20, Bytes read = 736, Access time = Fri Aug 10
18:11:06 KST 2007
```

놀랍지 않은가? 간단한 Byte code 수정만으로 손쉽게 Socket에서 발생하는 모든 I/O를 추적할 수 있다.

이러한 단순한 기법을 잘 발전시켜면 그 용도는 실로 무궁무진하다. File I/O, Network I/O, JDBC Request, Servlet Request, EJB Request, Struts Request, Spring Framework Request 등 Enterprise Java의 성능 모니터링과 분석에 필요한 데이터의 대부분을 수집하고 분석할 수 있다.

실제로 상업적으로 판매되는 대부분의 WAS 모니터링 툴들이 이런 비슷한 기법을 사용해서 성능 데이터를 수집하고 있다.

관심이 있는 사람이라면 이런 비싼 상용툴들을 도입하기 전에, 자신만의 Byte Code Instrumentation 기법을 이용해 WAS나 Java Application의 성능을 분석해보기를 권장한다.

이 BCI가 제공하는 기법이 너무나 강력하기 때문에, 한번 여기에 익숙해지면 다른 기법들은 모두 한 수 아래로 보일 것이다.

다음 Part에서 Java 5 (JDK 1.5)에서 BCI를 JDK 스펙 차원에서 지원하기 위해 새롭게 등장한 java.lang.instrument 패키지에 대해서 논의하고 Byte Code Instrumetation에 대한 논의를 마무리지을 것이다.

PS)

필자는 BCI를 이용해 Exception Tracking, Object Creation Tracking, File I/O Tracking, Network I/O Tracking, JDBC Tracking 등을 구현한 사례가 있는데, 사용할 수록 그 유용성에 놀라곤 한다. 이글을 읽는 여러분들도 이러한 놀라움을 나누지 않겠는가!!!

The Power of Byte Code Instrumentation in Java- Part 6

JDK는 전통적으로 JVM이 구동될 때 이 JVM 안에서 특정 작업을 수행할 에이전트(Agent), 즉 요원을 지정할 수 있는 방식을 제공해왔다. Java 5, 즉 JDK 1.5 이전 버전에서는 JVMPI(JVM Profiler Interface)라고 불렀다. Java 5부터는 JVMTI(JVM Tool Interface)라는 새로운 인터페이스가 제공된다. JVMPI/JVMTI의 특징은 C/C++로 구현 가능한 에이전트의 인터페이스를 제공한다는 것이다.

Java 5가 제공하는 희소식 중 하나는 Java로 구현 가능한 에이전트의 인터페이스를 제공한다는 것이다. 만세~~ 다음 명령어로 확인 가능하다.

```
Prompt> java

....

-agentlib:[=]
load native agent library , e.g. -agentlib:hprof
see also, -agentlib:jdwp=help and -agentlib:hprof=help
-agentpath:[=]
load native agent library by full pathname
-javaagent:[=]
load Java programming language agent, see java.lang.instrument
```

위의 옵션 들 중 agentlib/agentpath 옵션은 JVMPI/JVMTI 에이전트를 활성화하는 것이다. 반면 javaagent 옵션은 Java로 구현된 에이전트를 활성화하는 역할을 한다.

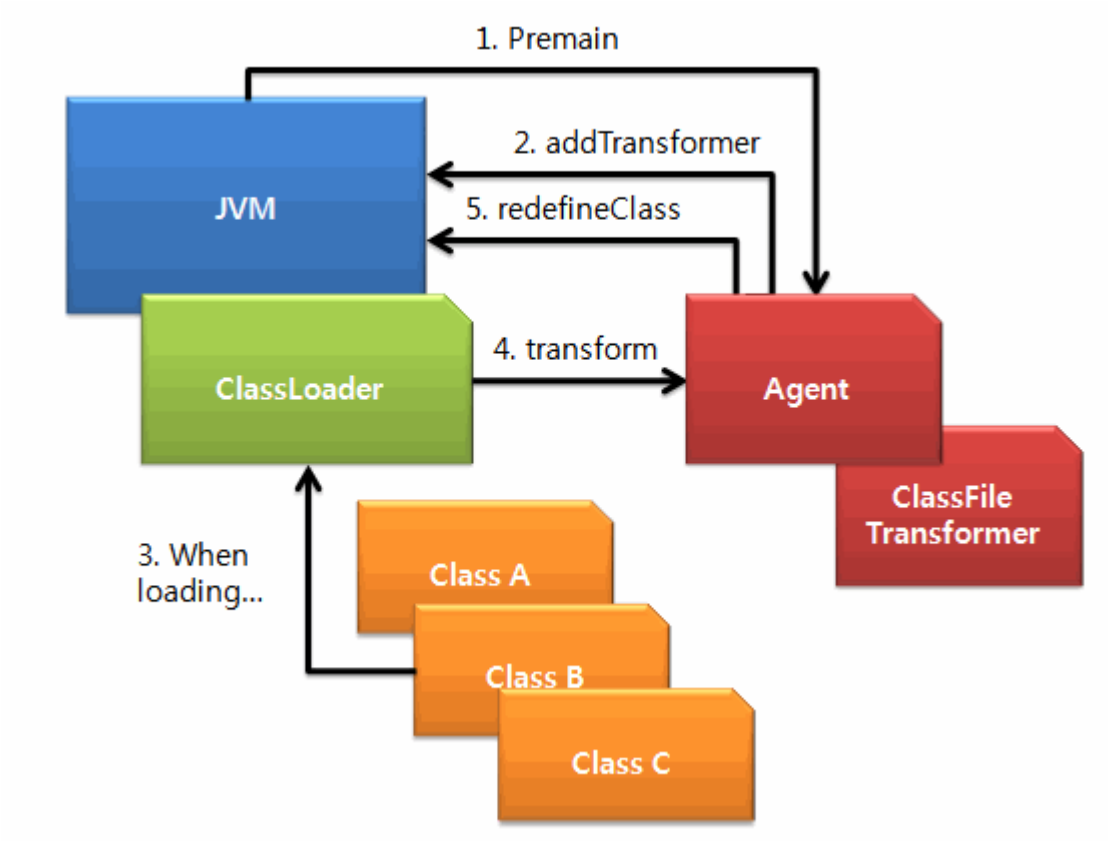
위의 설명이 (불)친절하게 안내하고 있는바대로 java.lang.instrument 패키지가 Java 에이전트의 인터페이스 역할을 한다.

java.lang.instrument 패키지에 대한 자세한 설명은

http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html를 참조한다.

JavaAgent BCI의 구조

아래 그림은 java.lang.instrument 패키지가 제공하는 기능을 그림으로 간략하게 표현한 것이다.



위의 과정을 간략하게 설명하면 다음과 같다.

1. Java Agent는 premain이라는 메소드를 구현한다. JVM은 Java Agent의 premain 메소드를 호출해서 Agent를 구동한 후에 Application의 main 메소드를 호출한다.
2. Java Agent는 ClassFileTransformer(클래스파일 변환기) 인터페이스를 구현하고, Instrumentation.addTransformer를 이용해서 클래스파일 변환기를 JVM에 등록한다.
3. JVM은 클래스 파일을 로드할 때 등록된 ClassFileTransformer의 transform 메소드를 호출해서 클래스파일의 바이트 코드를 변환하고, 변환된 바이트 코드를 원래 클래스 대신 사용한다. 바이트 코드를 변환하기 위해서 ASM 라이브러리를 사용한다.
4. Agent는 필요한 시점에 Instrumentation.redefineClasses 메소드를 이용해 특정 클래스의 바이트 코드를 런타임에 변경한다.

JavaAgent의 간단한 샘플

아래에 JavaAgent의 아주 간단한 샘플 소스가 있다.(실제 바이트 코드 변환에는 ASM 라이브러리를 사용한다)

```
public class SimpleProfiler implements ClassFileTransformer {

    public SimpleProfiler() {
```

```
        super();
    }

    public static void premain(String args, Instrumentation inst) {
        try {
            // Redefine preloadedclasses. Especially rt.jar boot class
            ArrayList defs = new ArrayList();
            for( Class c : inst.getAllLoadedClasses()) {
                try {
                    if(c.getName().equals("java.io.File")) { // 여기서는 샘플
로 java.io.File 만 변환한다.
                        System.out.println("Redefining class " +
c.getName());
                        ClassReader reader = new ClassReader(c.getName());
                        ClassWriter writer = new
ClassWriter(ClassWriter.COMPUTE_MAXS+ClassWriter.COMPUTE_FRAMES);

              // 여기가 구현부. ASM 라이브러리를 이용한다.
              // --> 여기서는 소스를제공하지 않으며, 내부적으로 모든 클래스의 메소드의 시작
점과 끝점에
                        // --> Method Start와 Method End를 알리는 코드를 삽입한다.
                        ClassAdapter adapter = new
SimpleClassAdapter(writer, c.getName));
                        reader.accept(adapter, ClassReader.SKIP_DEBUG);
                        byte[] result = writer.toByteArray();
                        if(result != null) defs.add(new ClassDefinition(c,
result));
                    }
                } catch(Exception ex) {
                    ex.printStackTrace();
                }
            }
            ClassDefinition[] cdef = defs.toArray(new
ClassDefinition[defs.size()]);
            inst.redefineClasses(cdef);
            inst.addTransformer(new SimpleProfiler());
            (new Thread(new MonitoringThread())).start();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    public byte[] transform(ClassLoader l, String className, Class c,
ProtectionDomain pd, byte[] b)
            throws IllegalClassFormatException {

        if (l != ClassLoader.getSystemClassLoader()) {
            return b;
        }
```

```
        if (className.startsWith("instrument/profiler")) {
            return b;
        }

        System.out.println("Classloader = " + l);
        System.out.println("Class = " + className);

        byte[] result = b;
        ClassReader reader = new ClassReader(b);
        ClassWriter writer = new ClassWriter(ClassWriter.COMPUTE_MAXS +
ClassWriter.COMPUTE_FRAMES);
        ClassAdapter adapter = new SimpleClassAdapter(writer, className); //
여기가 구현부...
        reader.accept(adapter, ClassReader.SKIP_DEBUG);

        result = writer.toByteArray();

        return result;

    }
}
```

SimpleProfiler 클래스는 JavaAgent로 다음과 같은 일을 수행한다.

1. premain메소드에서 현재까지Load된클래스들(특히rt.jar에 속하는Class들)을 Redefine한다.(같은 방법으로 어떠한 클래스이든지 원하는 대로 바이트 코드를 변환할 수 있다)
2. transform 메소드에서필요한 클래스들이 로딩되는 시점에 변환한다.

Manifest 파일

JavaAgent 파일은 필요한 클래스 파일들을 Jar 파일로 묶어서 JVM에 제공하게 된다. (여기서는 profiler.jar 라고 부르자)

특히 JVM에게 JavaAgent에 대한 정보를 넘겨주기 위해서 Manifest 파일을 작성해서 Jar 파일에 같이 넣어주어야 한다. Mainfest 파일에 기록해야 할 정보는 다음과 같다.

- Premain-Class: Premain 메소드를 담고 있는 Agent Class의 이름. 필수 예) instrument.profiler.SimpleProfiler
- Boot-Class-Path: Boot classloader에 의해 로딩될 Jar 파일의 경로. 옵션 예) profiler.jar;asm-3.0.jar
- Can-Redefine-Classes: Instrmentation.redefineClasses를 통해 Runtime으로 바이트 코드를 변환하는 것을 허용할지의 여부. 옵션. Default는 False 예) True

아래에 샘플로 사용한 Manifest 파일이 있다.

```
Premain-Class: instrument.profiler.SimpleProfiler
Can-Redefine-Classes: True
Boot-Class-Path: profiler.jar;asm-3.0.jar
```

위의 정보를 해석하면

1. instrument.profiler.SimpleProfiler가 premain 메소드를 구현하고 있는 JavaAgent이며
2. 바이트 코드의 런타임 변환(Instrumentation.redefineClasses)을 허용하며
3. profiler.jar(JavaAgent 자제) 파일과 asm-3.0.jar(ASM 라이브러리) 파일을 Boot class path에 등록한다.

두번째 옵션은 클래스 파일의 런타임 변환에 사용된다. 만일 위의 예제와 같이 rt.jar에 속한 클래스들, 즉 Boot class path에 속한 클래스들의 바이트 코드를 변환하려면 Profiler 자체가 Boot class path에 속해야 한다. 이것은 ClassLoader에 관한 JVM 고유의 속성에 기인한다.

JavaAgent의 실행

위와 같이 JavaAgent Jar 파일을 생성한 후, 다음과 같은 Java 명령어로 실행하면 된다.

```
java -javaagent:profiler.jar [Your Target Applicatin]
```

아래에 간단한 실행 결과가 있다. 모든 클래스의 모든 메소드의 실행 결과를 캡쳐할 수 있다.

```
---------- Call Table --------------------------------
instrument/target/RunThread.do31179[4694]
instrument/target/RunThread.do2791[5730]
instrument/target/RunThread.do1396[3189]
java.io.File.14[0]
java.io.File.getPath8[0]
java.io.File.exists6[0]
instrument/target/RunThread.5[0]
java.io.File.lastModified4[0]
java.io.File.length2[0]
java.io.File.getPrefixLength2[0]
java.io.File.getCanonicalPath2[0]
instrument/target/ProfilerTarget.main1[0]
```

JavaAgent와 JVMTI와의 관계. 그리고 JMX

JavaAgent와 JVMTI가 모두 Java 5 (JDK1.5)에서 지원되는 것은 결코 우연이 아니다. JavaAgent와 JVMTI는 몇가지 특징을 공유하고 있다. 실시간 클래스 코드 변환이 대표적인 케이스이고, Java 오브젝트의 크기를 계산한는 함수를 제공하는 것도 같은 맥락이다. (Instrumentation.getObjectSize와 JVMTI의 GetObjectSize)

특히 실시간 클래스 코드 변환은 다른 BCI 라이브러리가 제공하지 못하는 뛰어난 기능으로 이제 BCI는 크게 세가지로 분류가 가능해졌다.

- Static BCI : 특정 클래스의 바이트 코드를 정적으로 변환하는 방식
- Load-Time BCI : 특정 클래스가 로딩되는 시점에 변환하는 방식
- Runt-Time BCI : JVM이 실행되고 있는 시점에 특정 클래스의 바이트 코드를 변환하는 방식

Sun이 Java 5에서 JVMTI와 JavaAgent를 제공하면서 의도한 것은 이 두가지 방법을 상호 보완적으로 사용하라는 것이다. 즉, JVM 자체를 프로파일링하는 것은 여전히 JVMTI와 같은 C 언어 레벨의 API를 사용하되, BCI를 통해 특정 클래스아 액션을 프로파일링하는 것은 JavaAgent와 같은 Java 언어 레벨의 API를 사용하기를 권고한다.

여기에 JMX의 Platform MBean(JVM 정보를 제공하는 MBean) 기능까지 추가되었으니, 이제 Java는 성능 관리에 있어 그 어느때보다 폭넓은 기능을 제공하는 셈이다.

앞으로 시간날 때마다 JVMTI나 JMX같은 추가적인 기능들에 대해서도 논의할 기회를 가질려고 한다.

Epilogue

BCI는 성능 관리를 위해 사용가능한 현존하는 방법들 중 가장 보편적이고 편리한 기법이다.많은 공개 라이브러리가 제공되고 있고, 특히Java 5에서는 JVM 레벨에서 BCI를 지원한다.

만일 WAS와 같은 Java Application시스템 모니터링을 계획하고 있다면, BCI에 대한 검토가 반드시 필요할 것이다.

또는실제로 이 기법을 사용할 기회가 없다고 하더라도, 3rd party의 모니터링을 위해 툴을 도입할 때 이런 기법들이 어떻게 사용되고 있는지 이해해야만 툴의 특징을 파악할 수 있을 것이다.

PS)

아래 예제는 BCI를 이용해 Exception, Call Tree, File I/O, Net I/O, JDBC Request 등을 프로파일링한 결과이다. 상용 모니터링 툴들이 제공하는 것과 거의 동일한 수준의 정보를 BCI를 이용해 손쉽게 수집할 수 있다.

```
--------------------[Exception Tracking]--------------------
Exception count = 9
[Exception] Exception = java.io.IOException, Message = Something Bad2~~,
Thread id = 11, Caller =
[Sigature]flowlite.server.ExceptionGenerator.doSomething2, Called from =
[Sigature]flowlite.server.ExceptionGenerator.doSomething2, When = Mon Aug 13
11:06:10 KST 2007
flowlite.server.ExceptionGenerator.doSomething2(FlowLiteMBeanServer.java:128
)
flowlite.server.ExceptionGenerator.run(FlowLiteMBeanServer.java:113)
java.lang.Thread.run(Unknown Source)

[Exception] Exception = java.io.IOException, Message = Something Bad2~~,
Thread id = 11, Caller =
[Sigature]flowlite.server.ExceptionGenerator.doSomething2, Called from =
[Sigature]flowlite.server.ExceptionGenerator.run, When = Mon Aug 13 11:06:10
KST 2007
flowlite.server.ExceptionGenerator.doSomething2(FlowLiteMBeanServer.java:128
)
flowlite.server.ExceptionGenerator.run(FlowLiteMBeanServer.java:113)
java.lang.Thread.run(Unknown Source)

...
```

```
--------------------[Active Thread & Call Tree Tracking]--------------------
Thread count = 26
[ActiveThread] Thread id = 62, Thread name = Thread-49, Group name = main
[Call Tree]
[Call][Sigature]flowlite.server.CallTreeGenerator$InnerGenerator.depth1_1,
duration = 31[ms]
[Call][Sigature]java.util.Random.nextLong, duration = 0[ms]
[Call][Sigature]java.lang.Math.abs, duration = 0[ms]
[Call][Sigature]java.lang.Thread.sleep, duration = 0[ms]
[Call][Sigature]flowlite.server.CallTreeGenerator$InnerGenerator.depth2_1,
duration = 15[ms]
[Call][Sigature]java.util.Random.nextLong, duration = 0[ms]
[Call][Sigature]java.lang.Math.abs, duration = 0[ms]
[Call][Sigature]java.lang.Thread.sleep, duration = 15[ms]

...



--------------------[File I/O Tracking]--------------------
File I/O count = 22
[File Info] Thread id = 22, File name = c:test.txt, Status = 2, File mode =
[r+w], read time = Mon Aug 13 11:06:15 KST 2007
[File IO], Thread id = 22, File name =c:test.txt, Bytes read = 1024, Access
time = Mon Aug 13 11:06:15 KST 2007
[File IO], Thread id = 22, File name =c:test.txt, Bytes read = 1024, Access
time = Mon Aug 13 11:06:15 KST 2007
...


Socket Info] Thread id = 20, Host name =localhost:17904, Status = 1, Access
time = Mon Aug 13 11:06:13 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Mon Aug 13
11:06:13 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Mon Aug 13
11:06:13 KST 2007
[Net IO], Thread id = 20, Bytes read = 1024, Access time = Mon Aug 13
11:06:13 KST 2007

...


--------------------[JDBC I/O Tracking]--------------------
Connection count = 1
Statement count = 1
FetchCount count = 100
[Connection] Statement count = 10, DB = Oracle, Oracle Database 10g
Enterprise Edition Release 10.2.0.3.0 - 64bit Production
With the Partitioning, OLAP and Data Mining options, Created = 2007-08-13
```

```
[Statement] Query = SELECT name FROM t_pstmt_test WHERE id = ?, Execution
count = 10, Fetch size = 10, Statement Type = Prepared
[Execution] Column Count = 1, Fetch count = 1, {Parameters} = (1, 4),
[Fetch] Fetched value count = 1
(1, name4),

...
```

From:
https://atl.kr/dokuwiki/ - **AllThatLinux!**

Permanent link:
https://atl.kr/dokuwiki/doku.php/bci_-_the_power_of_bytecode_instrumentation_in_java

Last update: **2017/06/16 03:00**