+++++++++++++++++

+Chapter 05 SPRING AOP +

+++++++++++++++++

: Application은 다양한 공통 기능을 필요로 한다. Logging과 같은 기본적인 기능에서부터 Transaction이나 Security 같은

기능에 이르기까지 Application 전반에 걸쳐 적용되는 공통 기능이 존재한다. 이들 공통 기능들은 어떤 특정 Module에서만

필요로 하는 것이 아니라, Application 전반에 걸쳐 필요한 기능이다. 또한, 이런 공통 기능들은 Application의

핵심 비지니스 로직과는 구분되는 기능이다. 핵심 비지니스 기능과 구분하기 위해 공통 기능을 공통 관심 사항(cross-cutting concern)

이라고 표현하며, 핵심 로직을 핵심 관심 사항(core concern)이라고 한다.

공통 관심 사항들은 객체 지향 기법(상속이나 위임 등)을 사용해서 여러 Module에 효과적으로 적용하는 데는 한계가 있으며,

중복된 Code를 양산하곤 한다. 이런 한계를 극복하기 위해 AOP라는 기법이 소개되었는데, 본 절에서는 AOP 프로그래밍이

무엇인지 살펴보고, SPRING에서 AOP 프로그래밍을 어떻게 구현하는 지 살펴보도록 하겠다.

====================================================================================

1. AOP 소개

: Aspect Oriented Programming, 줄여서 AOP는 문제를 바라보는 관점을 기준으로 프로그래밍하는 기법을 말한다.

AOP는 문제를 해결하기 위한 핵심 관심 사항과 전체에 적용되는 공통 관심 사항을 기준으로 프로그래밍 함으로써

공통 Module을 여러 Code에 쉽게 적용할 수 있도록 도와 준다.

AOP를 구현하는 다양한 방법이 존재하지만, 기본적인 개념은 공통 관심 사항을 구현한 Code를 핵심 로직을 구현한 Code 안에

삽입한다는 것이다.

공통 기능 Code public void business(){

public void business(){ [[공통 기능 Code]]

//핵심 로직 구현 //핵심 로직 구현

... =============> ...

AOP 적용 [[공통 기능 Code]]

} }

공통 기능 Code

\*\*\*AOP에서 공통 기능을 구현한 Code가 핵심 로직에 삽입되어 실행된다.

AOP 기법에서는 위와 같이 핵심 로직을 구현한 Code에서 공통 기능을 직접적으로 호출하지 않는다. 핵심 로직을 구현한 Code를

Compile 하거나, Compile 된 Class를 로딩하거나, 또는 로딩한 Class의 객체를 생성할 때 AOP가 적용되어 핵심 로직 구현 Code

안에 공통 기능이 삽입된다.

AOP 프로그래밍에서는 AOP Library가 공통 기능을 알맞게 삽입해 주기 때문에 개발자는 게시글을 쓰거나 목록 읽기와 같은

핵심 로직을 개발할 때 Transaction 적용이나 Security 검사와 같은 공통 기능을 처리하기 위한 Code를 핵심 로직 Code에 삽입할 필요가

없다. 핵심 로직을 구현한 Code에 공통 기능 관련 Code가 포함되어 있지 않기 때문에 적용해야 할 공통 기능이 변경되더라도

핵심 로직을 구현한 Code를 변경할 필요가 없다. 단지, 공통 기능 Code를 변경한 뒤 핵심 로직 구현 Code에 적용만 하면 된다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

1.1 AOP 용어

: AOP는 다양한 용어를 소개하고 있으며, 이들 용어를 알고 있지 않으면 본 장을 읽고 이해하는 데 어려움이 따른다. 그래서, 먼저

AOP와 관련된 용어부터 살펴보도록 하겠다. 아래는 AOP와 관련해서 반드시 알아야 할 주요 용어를 설명한 것이다.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-Advice : 언제 공통 관심 기능을 핵심 로직에 적용할 지를 정의하고 있다. 예를 들어, '메서드를 호출하기 전'(언제)에 'Transaction을

시작한다'(공통 기능) 기능을 적용한다는 것을 정의하고 있다.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-Joinpoint : Advice를 적용 가능한 지점을 의미한다. 메서드 호출, 필드 값 변경 등이 Joinpoint에 해당한다.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-Pointcut : Joinpoint의 부분 집합으로서 실제로 Advice가 적용되는 Joinpoint를 나타낸다. SPRING에서는 정규 표현식이나

AspectJ의 문법을 이용하여 Pointcut을 정의할 수 있다.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-Weaving : Advice를 핵심 로직 Code에 적용하는 것을 weaving이라고 한다. 즉, 위와 같이 공통 Code를 핵심 로직 Code에

삽입하는 것이 weaving이다.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-Aspect : 여러 객체에 공통으로 적용되는 공통 관심 사항을 Aspect라고 한다.

Transaction이나 Security 등이 Aspect의 좋은 예이다.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

1.2 세 가지 Weaving 방식

: Advice를 Weaving하는 방식에는 다음과 같이 세 가지 방식이 존재한다.

- Compile 시에 Weaving 하기

- Class 로딩 시에 Weaving 하기

- 런타임 시에 Weaving 하기

Compile 시에 Code를 삽입하는 방법은 AspectJ에서 사용하는 방식이다. Compile 방식에서는 핵심 로직을 구현한 Java Source Code를

Compile 할 때에 알맞은 위치에 공통 Code를 삽입한다. 따라서 AOP가 적용된 Class 파일이 생성된다. Compile 방식을 제공하는

AOP 도구는 공통 Code를 알맞은 위치에 삽입할 수 있도록 도와 주는 Compile러나 IDE를 함께 제공한다.

Class를 로딩할 때에 AOP를 적용할 수도 있다. AOP Library는 JVM이 Class를 로딩할 때 Class 정보를 변경할 수 있는

에이전트를 제공한다. 에이전트는 로딩한 Class의 바이너리 정보를 변경하여 알맞은 위치에 공통 Code를 삽입한 새로운

Class 바이너리 Code를 사용하도록 한다. 즉, 원본 Class 파일은 변경하지 않고 Class를 로딩할 때에 JVM이 변경된 바이트 Code를

사용하도록 함으로써 AOP를 적용한다. AspectJ 5/6 버전이 Compile 방식과 더불어 Class 로딩 방식을 함께 지원하고 있다.

런타임 시에 AOP를 적용할 때는 SourceCode나 Class 정보 자체를 변경하지 않는다. 대신, 프록시를 이용하여 AOP를 적용한다.

프록시 기반의 AOP는 핵심 로직을 구현한 객체에 직접 접근하는 것이 아니라 아래와 같이 중간에 프록시를 생성하여

프록시를 통해서 핵심 로직을 구현한 객체에 접근하게 된다.

이때, 프록시는 핵심 로직을 실행하기 전 또는 후에 공통 기능을 적용하는 방식으로 AOP를 적용하게 된다. 프록시 기반에서는

메서드가 호출될 때에만 Advice를 적용할 수 있기 때문에 필드 값 변경과 같은 Joinpoint에 대해서는 적용할 수 없는 한계가 있다.

====================================================================================

2. SPRING에서의 AOP

: SPRING은 자체적으로 프록시 기반의 AOP를 지원하고 있다. 따라서, SPRING AOP는 메서드 호출 Joinpoint만을 지원한다.

필드 값 변경과 같은 Joinpoint를 사용하고 싶다면 AspectJ와 같이 풍부한 기능을 지원하는 AOP 도구를 사용해야만 한다.

SPRING은 완전한 AOP 기능을 제공하는 것이 목적이 아니라 Enterprise Application을 구현하는 데 필요한 기능을 제공하는 것을

목적으로 하고 있다. 따라서, SPRING AOP는 JEE Application을 구현하는 데 필요한 수준의 기능만을 제공하고 있다.

강사의 경험에 의하면 SPRING이 제공하는 AOP만으로 Enterprise Application을 구현하는 데 부족함이 없었다. (단, 도메인 객체에

AOP를 적용하려면 SPRING AOP 자체로는 불가능하며, 이 경우 AspectJ와 같은 좀더 풍부한 AOP 기능을 제공하는 솔루션을 이용하여

해결 해야 한다.)

SPRING AOP의 또 다른 특징은 Java 기반이라는 점이다. AspectJ는 Aspect를 위한 별도의 문법을 제공하고 있는 반면에 SPRING은 별도의

문법을 익힐 필요 없이 Java 언어만을 이용하면 된다.

SPRING은 세 가지 방식으로 AOP를 구현할 수 있도록 하고 있다.

-XML 스키마 기반의 POJO Class를 이용한 AOP 구현.

-AspectJ 5/6에서 정의한 @Aspect Annotation 기반의 AOP 구현.

-SPRING API를 이용한 AOP 구현.

어떤 방식을 사용하더라도 내부적으로는 프록시를 이용하여 AOP가 구현되므로 메서드 호출에 대해서만 AOP를 적용할 수 있다는 것에

유의하자. (즉, AspectJ 5/6에서 정의한 @Aspect Annotation을 사용하더라도 메서드 호출과 관련된 Pointcut만 사용 가능하다.)

2.1 프록시를 이용한 AOP 구현

: SPRING은 프록시를 이용하여 AOP를 구현하고 있다. SPRING은 AspectJ의 적용 대상(target)이 되는 객체에 대한 프록시를 만들어 제공하며,

대상 객체를 사용하는 Code는 대상 객체에 직접 접근하기 보다는 프록시를 통해서 간접적으로 접근하게 된다. 이 과정에서 프록시는 공통 기능을

실행한 뒤 대상 객체의 실제 메서드를 호출하거나 또는 대상 객체의 실제 메서드가 호출된 뒤 공통 기능을 실행하게 된다.

1. operation()------------------>

client : Client =================== proxy : TargetProxy |

| <----- 2. operation()

|

target : TargetImpl

\*\*\*프록시를 통한 AOP 구현

SPRING에서 어떤 대상 객체에 대해 AOP를 적용할 지의 여부는 설정 파일을 통해서 지정할 수 있으며, SPRING은 설정 정보를 이용하여

런타임에 대상 객체에 대한 프록시 객체를 생성하게 된다. 프록시 객체를 생성하는 방식은 대상 객체가 Interface를 구현하고 있느냐

없느냐 여부에 따라 달라진다.

대상 객체가 Interface를 구현하고 있다면, SPRING은 Java 리플랙션 API가 제공하는 java.lang.reflect.Proxy를 이용하여 프록시 객체를

생성한다. 이때 생성된 프록시 객체는 아래와 같이 대상 객체와 동일한 Interface를 구현하게 되며, 클라이언트는 Interface를 통해서

필요한 메서드를 호출하게 된다. 하지만, Interface를 기반으로 프록시 객체를 생성하기때문에 Interface에 정의되어 있지 않은 메서드에

대해서는 AOP가 적용되지 않는 점에 유의해야 한다.

Client --------------------------> <<Interface>>

TargetIF

+operation()

||

||

==================

|| ||

TargetProxy TargetImpl

+operation() +operation()

\*\*\* Interface 기반 프록시 구조

대상 객체가 Interface를 구현하고 있지 않다면, SPRING은 CGLIB를 이용하여 Class에 대한 프록시 객체를 생성한다. CGLIB는 대상 Class를

상속받아 프록시를 구현한다. 따라서, 대상 Class가 fianl인 경우 프록시를 생성할 수 없으며, fianl인 메서드에 대해서는 AOP를 적용할 수 없게 된다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.2 구현 가능한 Advice 종류

: SPRING은 프록시를 이용해서 메서드를 호출할 때 Aspect를 적용하기 때문에 구현 가능한 Advice 종류는 아래와 같다.

-Before Advice : 대상 객체의 메서드 호출 전에 공통 기능을 실행한다.

-After Returning Advice : 대상 객체의 메서드가 예외 없이 실행한 이후에 공통 기능을 실행한다.

-After Throwing Advice : 대상 객체의 메서드를 실행하는 도중 예외가 발생한 경우에 공통 기능을 실행한다.

-After Advice : 대상 객체의 메서드를 실행하는 도중에 예외가 발생했는지의 여부와 상관없이 메서드 실행 후 공통 기능을 실행한다.

(try~catch~finally의 finally 블록과 비슷하다.)

-Around Advice : 대상 객체의 메서드 실행 전,후 또는 예외 발생 시점에 공통 기능을 실행하는데 사용된다.

이들 AOP 중에서 범용적으로 사용되는 것은 Around Advice인데, 그 이유는 대상 객체의 메서드를 실행하기 전/후에 원하는 기능을 삽입할 수

있기 때문이다. 이런 이유로 캐시 기능, 성능 모니터링 기능과 같은 Aspect를 구현할 때는 Around Advice를 주로 이용하게 된다.

====================================================================================

3. XML 스키마 기반의 POJO Class를 이용한 AOP 구현

: SPRING 2버전부터 SPRING API를 사용하지 않은 POJO Class를 이용하여 Advice를 개발하고 적용할 수 있는 방법이 추가되었다. SPRING 2에

새롭게 추가된 XML 스키마 확장 기법을 통해 설정 파일도 보다 쉽게 작성할 수 있게 되었다. 본 절에서는 XML 스키마 확장 방식을 이용해서

POJO Class로 Aspect를 구현하는 방법을 살펴보도록 하자.

3.1 XML 스키마 기반 AOP 퀵 스타트

: XML 스키마를 이용해서 AOP를 구현하는 과정은 다음과 같다.

-관련 jar파일을 Class 패스에 추가한다.

-공통 기능을 제공하는 Advice Class를 구현한다.

-XML 설정 파일에서 <aop:config>를 이용해서 Aspect를 설정한다. Advice를 어떤 Pointcut에 적용할 지를 지정하게 된다.

먼저 해야 할 작업은 공통 기능을 제공할 Advice Class를 작성하는 것이다. 아래는 Around Advice Class의 작성 예이다.

[ProfilingAdvice.java]

package madvirus.spring.chap05.aop.pojo;

import org.aspectj.lang.ProceedingJoinPoint;

public class ProfilingAdvice {

public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {

String signatureString = joinPoint.getSignature().toShortString();

System.out.println(signatureString + " 시작");

long start = System.currentTimeMillis();

try {

Object result = joinPoint.proceed();

return result;

} finally {

long finish = System.currentTimeMillis();

System.out.println(signatureString + " 종료");

System.out.println(signatureString + " 실행 시간 : " + (finish - start) + "ms");

}

}

}

위의 Code를 보면 이 Advice가 Around Advice인지 Before Advice인지의 여부가 명시되어 있지 않다.

단지, 특정 Joinpoint에서 실행될 메서드인 trace() 메서드만 구현하고 있다. trace() 메서드는 ProceedingJoinPoint 타입의

joinPoint 파라미터를 전달받는데, 이 파라미터를 이용해서 Around Advice를 구현할 수 있게 된다.

위 Code를 간단히 설명하면, ProfilingAdvice Class는 Around Advice를 구현한 Class이며, trace()메서드는 Advice가 적용될

대상 객체를 호출하기 전과 후에 시간을 구해서 대상 객체의 메서드 호출 실행 시간을 출력한다. 일단 지금은 이 정도만 이해하고

넘어가도록 하고, 뒤에서 Advice Class의 자세한 구현 방법은 살펴볼 것이다.

Advice를 작성했다면 그 다음으로 해야 할 작업은 XML 파일에 Advice를 빈으로 등록하고 Aspect를 설정하는 것이다.

아래 Code를 설정 예제 파일이다.

[acQuickStart.xml]

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

<!-- Advice Class를 빈으로 등록 -->

<bean id="performanceTraceAdvice" class="madvirus.spring.chap05.aop.pojo.ProfilingAdvice" />

<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->

<aop:config>

<aop:aspect id="traceAspect1" ref="performanceTraceAdvice">

<aop:pointcut id="publicMethod" expression="execution(public \* madvirus.spring.chap05.board..\*(..))" />

<aop:around pointcut-ref="publicMethod" method="trace" />

</aop:aspect>

<aop:aspect id="traceAspect2" ref="performanceTraceAdvice">

<aop:around pointcut="execution(public \* madvirus.spring.chap05.member..\*(..))" method="trace" />

</aop:aspect>

</aop:config>

<bean id="writeArticleService" class="madvirus.spring.chap05.board.service.WriteArticleServiceImpl">

<constructor-arg>

<ref bean="articleDao" />

</constructor-arg>

</bean>

<bean id="articleDao" class="madvirus.spring.chap05.board.dao.MySQLArticleDao" />

<bean id="memberService" class="madvirus.spring.chap05.member.service.MemberServiceImpl" />

</beans>

Advice Class를 적용하려면 일단 XML 설정 파일에서 Advice Class를 빈으로 등록해 주어야 한다. 또한, XML 스키마를 이용해서

AOP를 구현하려면 aop 네임스페이스를 추가해주어야 한다. aop 네임스페이스를 추가했다면, <aop:config>, <aop:aspect>,

<aop:pointcut>, <aop:around> 태그를 이용해서 AOP 설정을 할 수 있다.

위의 Code의 설정 내용을 간단히 설명하면 madvirus.spring.ch05 패키지 및 그 하위 패키지에 있는 모든 public 메서드를

Pointcut으로 설정하고, 이들 Pointcut에 Around Advice에 performanceTraceAdvice 빈 객체의 trace() 메서드를 적용한다.

따라서, writeArticleService빈의 public 메서드나 articleDao의 public 메서드가 호출되면 ProfilingAdvice Class의 trace() 메서드가

AroundAdvice로 적용된다.

참고로, Pointcut 표현식에서 execution 명시자(designator)는 Advice를 적용할 패키지, Class 그리고 메서드를 표현할 때 사용된다.

이 외에도 다양한 명시자가 존재하는 데 이에 대한 본 장의 'AspectJ의 Pointcut 표현식'에서 살펴보도록 하겠다.

실제로 Advice가 Pointcut으로 지정한 메서드에 적용되는 지의 여부를 확인하기 위해 아래와 같은 Class를 작성해 보자.

[MainQuickStart.java]

package madvirus.spring.chap05;

import madvirus.spring.chap05.board.Article;

import madvirus.spring.chap05.board.service.WriteArticleService;

import madvirus.spring.chap05.member.Member;

import madvirus.spring.chap05.member.service.MemberService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainQuickStart {

public static void main(String[] args) {

String[] configLocations = new String[] { "acQuickStart.xml" };

ApplicationContext context = new ClassPathXmlApplicationContext(configLocations);

WriteArticleService articleService = (WriteArticleService) context.getBean("writeArticleService");

articleService.write(new Article());

MemberService memberService = context.getBean("memberService", MemberService.class);

memberService.regist(new Member());

}

}

MainQuickStart Class는 위에서 작성한 asQuickStart.xml 파일을 이용해서 ApplicationContext를 생성한 뒤

writeArticleService 빈의 write() 메서드를 호출하고, memberService 빈의 regist() 메서드를 호출하고 있다.

앞서 asQuickStart.xml 설정 파일을 설명할 때 madvirus.spring.ch05 패키지 및 그 하위 패기지에 있는 빈 객체의

public 메서드를 호출하면 ProfilingAdvice Class의 trace() 메서드가 Around Advice로 사용된다고 했는데,

실제로 실행해보면 다음과 같은 결과가 출력된다.

\*WriteArticleService.write(...) 시작

WriteArticleServiceImpl.write() 메서드 실행

\*ArticleDao.insert(...) 시작

MySQLArticleDao(...) 실행

\*ArticleDao.insert(...) 종료

\*ArticleDao.insert(...) 실행 시간 : 0ms

\*WriteArticleService.write(...) 종료

\*WriteArticleService.write(...) 실행 시간 : 0ms

\*MemberService.regist(...) 시작

MemberServiceImpl.regist() 메서드 실행

\*MemberService.regist(...) 종료

\*MemberService.regist(...) 실행 시간 : 0ms

위 Code에서 '\*"로 표시한 부분이 ProfilingAdvice Class의 trace() 메서드에서 출력한 내용이다. 실행 결과를 보면

실제 빈 객체의 메서드가 호출되기 전/후로 trace() 메서드에서 실행한 내용이 출력되는 것을 확인 할 수 있다.

위 실행 결과를 통해서 눈 여겨 볼 부분은 WriteArticleServiceImpl, MySQLArticleDao, MemberServiceImpl Class를

변경하지 않고 (즉, 핵심 로직 Code의 변경 없이) 공통 기능을 (즉, 메서드 실행 시간을 출력해주는 기능을) 추가했다는

점이다. 이는 설정 파일만 변경하면 기존 Code의 변경 없이 공통 기능을 추가하거나 변경할 수 있는 AOP의 장점을

잘 보여주고 있다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.2 XML 스키마를 이용한 AOP 설정

: SPRING 2.5버전부터 AOP를 설정하기 위한 aop 네임스페이스 및 aop 네임스페이스와 관련된 XML 스키마가 추가되었다.

aop 네임스페이스와 관련된 XML 스키마는 다음과 같이 <beans> 태그에 명시할 수 있다.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

...

</beans>

aop 네임스페이스와 관련된 XML 스키마를 지정한 뒤에, 다음과 같이 <aop:config> 태그를 사용하여 AOP 관련 정보를

설정할 수 있다.

<!-- Advice Class를 빈으로 등록 -->

<bean id="performanceTraceAdvice" class="madvirus.spring.chap05.aop.pojo.ProfilingAdvice" />

<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->

<aop:config>

<aop:aspect id="traceAspect1" ref="performanceTraceAdvice">

<aop:pointcut id="publicMethod" expression="execution(public \* madvirus.spring.chap05.board..\*(..))" />

<aop:around pointcut-ref="publicMethod" method="trace" />

</aop:aspect>

</aop:config>

<bean id="memberService" class="madvirus.spring.chap05.member.service.MemberServiceImpl" />

위 Code에서 <aop:\*>에 해당하는 태그는 다음과 같은 정보를 설정한다.

- <aop:config> : AOP 설정 정보임을 나타낸다.

- <aop:aspect> : Aspect를 설정한다.

- <aop:pointcut> : Pointcut을 설정한다.

- <aop:around> : Around Advice를 설정한다. 이 외에도 다양한 Advice를 설정할 수 있다.

<aop:aspect> 태그의 ref 속성은 Aspect로서 기능을 제공할 빈을 설정할 때 사용된다. 위 Code의 경우 "performanceTraceAdvice"

빈이 Aspect 기능을 제공한다고 명시하고 있다.

위 Code를 보면 "traceAspect"가 MemberServiceImpl Class가 구현한 모든 Interface의 public 메서드에 Around Advice로 적용되며,

이때, Aspect의 구현 Class인 ProfilingAdvice의 trace() 메서드가 호출된다는 것을 쉽게 유추할 수 있다. XML 스키마 기반의 AOP설정은

이렇게 설정 파일만 보더라도 어렵지 않게 어떤 Code에 어떤 Aspect가 어떤 타입의 Advice로 적용되는지를 파악할 수 있다.

(1) Aspect 설정

: Aspect 설정에서 <aop:aspect> 태그는 한 개의 Aspect를 설정한다.

<aop:aspect> 태그의 ref 속성에는 공통 기능을 구현하고 있는 빈을 전달한다.

<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->

<aop:config>

<aop:aspect id="traceAspect1" ref="performanceTraceAdvice">

<aop:pointcut id="publicMethod" expression="execution(public \* madvirus.spring.chap05.board..\*(..))" />

<aop:around pointcut-ref="publicMethod" method="trace" />

</aop:aspect>

</aop:config>

Advice를 적용할 Pointcut은 <aop:pointcut> 태그를 이용하여 설정한다. <aop:pointcut> 태그의 id 속성은

Pointcut을 구분하는데 사용되는 식별 값을 입력 받고, expression 속성은 Pointcut을 정의하는 AspectJ의 표현식을

입력 받는다.

Advice를 표현하는 태그에는 <aop:around>를 비록하여 각 타입의 Advice를 정의하기 위해 아래와 같은 태그를 제공한다.

- <aop:before> : 메서드 실행 전에 적용되는 Advice를 말한다.

- <aop:after-returning> : 메서드가 정상적으로 실행된 후에 적용되는 Advice를 정의한다.

- <aop:after-throwing> : 메서드가 예외를 발생시킬 때 적용되는 Advice를 정의한다. try~catch 블록에서 catch 블록과 비슷하다.

- <aop:after> : 메서드가 정상적으로 실행되는지 또는 예외를 발생시키는지 여부에 상관없이 적용되는 Advice를 정의한다.

try~cathc~finally 에서 finally 블록과 비슷하다.

- <aop:around> : 메서드 호출 이전, 예외 발생 등 모든 시점에 적용 가능한 Advice를 정의한다.

각 태그는 pointcut 속성 또는 pointcut-ref 속성을 사용하여 Advice가 적용될 Pointcut을 지정한다.

pointcut 속성은 직접 AspectJ 표현식을 이용하여 Pointcut을 지정할 때에 사용된다.

아래 Code는 pointcut 속성과 pointcut-ref 속성의 사용 예를 보여 주고 있다.

<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->

<aop:config>

<aop:aspect id="traceAspect1" ref="performanceTraceAdvice">

<aop:pointcut id="publicMethod" expression="execution(public \* madvirus.spring.chap05.board..\*(..))" />

<aop:around pointcut-ref="publicMethod" method="trace" />

</aop:aspect>

<aop:aspect id="traceAspect2" ref="performanceTraceAdvice">

<aop:around pointcut="execution(public \* madvirus.spring.chap05.member..\*(..))" method="trace" />

</aop:aspect>

</aop:config>

Advice의 각 태그는 Pointcut에 포함되는 대상 객체의 메서드가 호출될 때, <aop:aspect> 태그의 ref 속성으로 지정한

빈 객체에서 어떤 메서드를 실행할 지를 지정한다. 예를 들어, 위 Code의 경우 madvirus.spring.ch05.member 패키지 및

그 하위 패키지의 public 메서드가 호출될 때 performanceTraceAdvice 빈의 trace() 메서드가 호출되도록 설정하고 있다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.3 Advice 타입 별 Class 작성

: <aop:before>, <aop:around> 등의 태그는 <aop:aspect> 태그의 ref 속성에 전달된 빈 객체의 메서드를 실행함으로써

Advice를 대상 객체에 적용한다. 예를 들어, 아래 설정에서는 ProfilingAdvice Class의 trace() 메서드에 Around Advice가 제공해야 할

기능을 구현하게 된다.

<!-- Advice Class를 빈으로 등록 -->

<bean id="performanceTraceAdvice" class="madvirus.spring.chap05.aop.pojo.ProfilingAdvice" />

<!-- Aspect 설정: Advice를 어떤 Pointcut에 적용할 지 설정 -->

<aop:config>

<aop:aspect id="traceAspect1" ref="performanceTraceAdvice">

<aop:pointcut id="publicMethod" expression="execution(public \* madvirus.spring.chap05.board..\*(..))" />

<aop:around pointcut-ref="publicMethod" method="trace" />

</aop:aspect>

</aop:config>

각 타입의 Advice에 따라서 메서드의 구현 방법이 조금씩 차이가 나는데, 본 절에서는 Advice의 실제 기능을 제공할 Class의

메서드 구현 방법을 살펴보도록 하자.

(1) Before Advice

: Before Advice를 사용하려면 <aop:before> 태그를 이용하여 Before Advice를 설정 하면 된다.

아래 Code는 <aop:before> 태그의 설정 예이다.

<aop:config>

...

<aop:aspect id="loggingAspect" ref="logging">

<aop:before pointcut-ref="publicMethod" method="before"/>

</aop:aspect>

<aop:config>

Aspect로 사용될 빈 Class는 다음과 같이 <aop:before> 태그의 method 속성에 명시한 메서드를 구현해 주면 된다.

public void before(){

//대상 객체의 메서드 실행 이전에 적용할 기능 구현

...

}

대상 객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에 대한 정보가 필요한 경우 org.aspectj.lang.JoinPoint

타입의 파라미터를 메서드에 전달한다.

public void before(Joinpoint joinPoint){

//대상 객체의 메서드 실행 이전에 적용할 기능 구현

...

}

Before Advice를 구현한 메서드는 일반적으로 리턴 타입이 void인데, 그 이유는 리컨 값을 갖더라도 실제 Advice의 적용과정에

아무런 영향이 없기 때문이다.

Before Advice를 위한 메서드 구현시 주의해야 할 점은, 메서드에서 예외를 발생시킬 경우 대상 객체의 메서드가 호출되지

않게 된다는 점이다.

(2) After Returning Advice

: After Returning Advice는 대상 객체의 메서드가 정상적으로 실행된 후에 공통 기능을 적용하고 싶을 때 사용되는

Advice로서, 다음과 같이 <aop:after-returning> 태그를 이용하여 After Returning Advice를 설정한다.

<aop:config>

...

<aop:aspect id="loggingAspect" ref="logging">

<aop:after-returning pointcut-ref="publicMethod" method="afterReturning"/>

</aop:aspect>

<aop:config>

Aspect로 사용될 빈 Class는 아래 Code처럼 <aop:after-returning> 태그에 명시한 메서드를 구현한다.

public void afterReturning(){

//대상 객체의 메서드가 정상적으로 실행된 이후에 적용할 기능 구현

...

}

Advice를 구현한 메서드에서 리턴 값을 사용하고 싶다면 returnning 속성을 사용하여 리턴 값을 전달받을 파라미터의 이름을

명시해 주면 된다.

<aop:after-returning pointcut-ref="publicMethod" method="afterReturning" returning="ret"/>

Advice를 구현한 메서드는 returning 속성에 명시한 이름을 갖는 파라미터를 이용해서 리턴 값을 전달 받게 된다.

public void afterReturning(Object ret){

//대상 객체의 메서드가 정상적으로 실행된 이후에 적용할 기능 구현

...

}

만약 리턴된 객체가 특정 타입인 경우에 한해서 메서드를 실행하고 싶다면 다음과 같이 한정하고 싶은 타입의 파라미터를

사용하면 된다.

public void afterReturning(Article ret){

//대상 객체의 메서드가 정상적으로 실행된 이후에 적용할 기능 구현

...

}

대상 객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에 대한 정보가 필요한 경우 다음과 같이 org.aspectj.lang.JoinPoint를

파라미터로 추가한다.

public void afterReturning(JoinPoint joinPoint, Object ret){

//대상 객체의 메서드가 정상적으로 실행된 이후에 적용할 기능 구현

...

}

<aop:returning> 태그 뿐만 아니라 뒤에서 살펴볼 <aop:after-throwing> 태그 그리고 @Aspect Annotation 구현에서도 동일한

방식으로 파라미터 정보를 구한다.

(3) After Throwing Advice

: After Throwing Advice는 대상 객체의 메서드가 예외를 방생시킨 경우에 적용되는 Advice로 다음과 같이

<aop:after-throwing> 태그를 이용하여 설정한다.

<aop:config>

...

<aop:aspect id="loggingAspect" ref="logging">

<aop:after-throwing pointcut-ref="publicMethod" method="afterThrowing"/>

</aop:aspect>

<aop:config>

Advice 구현 Class는 다음과 같이 <aop:after-returning> 태그에 명시한 메서드를 구현한다.

public void afterThrowing(){

//대상 객체의 메서드가 예외를 발생시킨 경우에 적용할 기능 구현

...

}

대상 객체의 메서드가 발생시킨 예외 객체가 필요한 경우 throwing 속성에 예외 객체를 전달받을

파라미터의 이름을 명시하면 된다.

<aop:after-throwing pointcut-ref="publicMethod" method="afterThrowing" throwing="ex"/>

Advice 구현 메서드에서 발생된 예외를 사용하려면 <aop:after-throwing> 태그의 throwing 속성에

명시한 이름을 갖는 파라미터를 추가하면 된다.

public void afterThrowing(Throwable ex){

//대상 객체의 메서드가 예외를 발생시킨 경우에 적용할 기능 구현

...

}

만약 특정 타입의 예외에 대해서만 처리하고 싶다면, Throwable이나 Exception이 아니라 처리하고 싶은 예외 타입을 파라미터로

지정하면 된다. 예를 들어, 아래 Code는 발생된 예외가 ArticleNotFoundException인 경우에만 호출된다.

public void afterThrowing(ArticleNotFoundException ex){

//대상 객체의 메서드가 예외를 발생시킨 경우에 적용할 기능 구현

...

}

대상 객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에 대한 정보가 필요한 경우 다음과 같이 org.aspectj.lang.JoinPoint를

파라미터로 추가한다.

public void afterThrowing(JoinPoint joinPoint, Exception ex){

//대상 객체의 메서드가 예외를 발생시킨 경우에 적용할 기능 구현

...

}

(4) After Advice

: After Advice는 대상 객체의 메서드가 정상적으로 실행되었는지 아니면 예외를 발생시켰는지의 여부에 상관없이

메서드 실행이 종료된 이후에 적용되는 Advice로서 try~catch~finally 블록에서 fianlly 블록과 비슷한 기능을 수행한다. 다음과 같이

<aop:after> 태그를 이용하여 After Advice를 설정한다.

<aop:config>

...

<aop:aspect id="loggingAspect" ref="logging">

<aop:after pointcut-ref="publicMethod" method="afterFinally"/>

</aop:aspect>

<aop:config>

Aspect로 사용될 빈 Class는 다음과 같이 <aop:after> 태그에 명시한 메서드를 구현해 주면 된다. 이때 메서드는 파라미터를 갖지 않는다.

public void afterFinally(){

...

}

대상 객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에 대한 정보가 필요한 경우 다음과 같이 org.aspectj.lang.JoinPoint를

파라미터로 명시하면 된다.

public void afterFinally(JoinPoint joinPoint){

...

}

(5) Around Advice

: Around Advice는 앞서 살펴 본 Before, AfterReturning, After Throwing, After Advice를 모두 구현할 수 있는 Advice로서, 다음과 같이

<aop:around> 태그를 이용하여 Around Advice를 설정한다.

<bean id="cache" class="madvirus.spring.ch05.aop.pojo.ArticleCacheAdvice" />

<aop:config>

<aop:aspect id-"cacheAspect" ref="cache">

<aop:around method="cache" pointcut="execution(public \* \*..ReadArticleService.\*(..))" />

</aop:aspect>

</aop:config>

Around Advice를 구현한 메서드는 org.aspectj.lang.ProceedingJoinPoint를 반드시 첫 번째 파라미터로 지정해야 한다. 그렇지 않을 경우

SPRING은 예외를 발생시킨다.

다음 Code는 Around Advice의 구현 예를 보여주고 있다.

[ArticleCacheAdvice.java]

package madvirus.spring.chap05.aop.pojo;

import java.util.HashMap;

import java.util.Map;

import madvirus.spring.chap05.board.Article;

import org.aspectj.lang.ProceedingJoinPoint;

public class ArticleCacheAdvice {

private Map<Integer, Article> cache = new HashMap<Integer, Article>();

public Article cache(ProceedingJoinPoint joinPoint) throws Throwable {

Integer id = (Integer) joinPoint.getArgs()[0];

Article article = cache.get(id);

if (article != null) {

System.out.println("[ACA] 캐시에서 Article[" + id + "] 구함");

return article;

}

Article ret = (Article) joinPoint.proceed();

if (ret != null) {

cache.put(id, ret);

System.out.println("[ACA] 캐시에 Article[" + id + "] 추가함");

}

return ret;

}

}

위의 Code를 보면 첫 번째 파라미터로 ProceedingJoinPoint를 전달받고 있다. ProceedingJoinPoint의 proceed() 메서드를

호출하면 프록시 대상 객체의 실제 메서드를 호출하게 된다. 따라서, ProceedingJoinPoint.proceed() 메서드를 호출하기 전과

후에 필요한 작업을 수행할 수 있다.

위의 Code의 경우 대상 객체의 메서드 호출 전후에 캐시 기능을 삽입하였다. proceed() 메서드를 호출하기 전에 Map에 ID에

해당하는 Article 객체가 존재하는지 검사한다. Article 객체가 존재 할 경우 Map에 보관된 Article 객체를 리턴한다. (존재하지

않을 경우 proceed() 메서드를 호출해서 실제 대상 객체의 메서드를 호출하고 , 그 결과로 전달받은 객체를 Map에 저장한다.

4. @Aspect Annotation을 이용한 AOP

: @Aspect Annotation은 AspectJ 5버전에 새롭게 추가된 Annotation으로서, @Aspect Annotation을 사용하면 XML 파일에

Advice 및 Pointcut 등의 설정을 하지 않고도 자동으로 Advice를 적용할 수 있게 되었다. SPRING 2버전부터

@Aspect Annotation을 지원하고 있으며,

본 절에서는 @Aspect Annotation 및 관련 Annotation을 이용해서 Aspect를 구현하는 방법을 살펴보도록 하겠다.

4.1 @Aspect Annotation 기반 AOP 퀵 스타트

: @Aspect Annotation을 이용해서 AOP를 구현하는 과정은 XML 스키마 기반의 AOP를 구현하는 과정과 거의 유사하며,

차이점은 다음과 같다.

- @Aspect Annotation을 이용해서 Aspect Class를 구현한다.

이때 Aspect Class는 Advice를 구현한 메서드와 Pointcut을 포함한다.

- XML 설정에서 <aop:aspectj-autoproxy/>를 설정한다.

@Aspect Annotation을 이용할 경우 XML 설정 파일에서 Pointcut을 설정하는 것이 아니라 Class에 Pointcut을 정의한다.

아래 Code는

@Aspect Annotation 및 Advice 관련 Annotation인 @Around Annotation을 이용해서 구현한 Aspect Class 예이다.

[ProfilingAspect.java]

package madvirus.spring.chap05.aop.annot;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class ProfilingAspect {

@Pointcut("execution(public \* madvirus.spring.chap05.board..\*(..))")

private void profileTarget() {}

@Around("profileTarget()")

public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {

String signatureString = joinPoint.getSignature().toShortString();

System.out.println(signatureString + " 시작");

long start = System.currentTimeMillis();

try {

Object result = joinPoint.proceed();

return result;

} finally {

long finish = System.currentTimeMillis();

System.out.println(signatureString + " 종료");

System.out.println(signatureString + " 실행 시간 : " + (finish - start) + "ms");

}

}

}

위의 Code에서 @Aspect Annotation을 적용했는데, @Aspect Annotation이 적용된 Class는 Advice 구현 메서드나

Pointcut 정의를 포함할 수 있게 된다. ProfilingAdvice Class의 경우는 Advice 구현 메서드와 Pointcut 정의를 모두

포함하고 있다.

라인11~12에서는 @Pointcut Annotation을 이용해서 Pointcut을 정의하고 있다.

@Pointcut("execution(public \* madvirus.spring.chap05.board..\*(..))")

private void profileTarget() {}

@Pointcut Annotation은 Pointcut을 정의하는 AspectJ의 표현식을 값으로 갖게 된다.

@Pointcut Annotation을 적용한 메서드는 리턴타입이 void여야 한다.

@Pointcut Annotation을 이용해서 Pointcut을 정의하면, Advice 관련 Annotation에서 해당 메서드 이름을 이용해서

Pointcut을 사용할 수 있게 된다. 라인14에서는 @Around Annotation을 이용해서 Around Advice를 구현하고 있는데,

이때 @Around Annotation 값으로 @Pointcut Annotation을 적용한 메서드의 이름을 지정한 것을 확인할 수 있다.

@Pointcut("execution(public \* madvirus.spring.chap05.board..\*(..))")

private void profileTarget() {}

@Around("profileTarget()")

public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {

...

}

위 Code의 경우 trace() 메서드는 Around Advice를 구현하게 되며, profileTarget() 메서드에 정의된 Pointcut에

Advice를 적용하게 된다.

@Aspect Annotation을 이용해서 Aspect Class를 작성했다면, 다음으로 할 작업은 XML 설정 파일에 Aspect Class를

빈으로 등록하는 것이다. 다음 Code는 설정 파일의 작성 예이다.

[asQuickStart2.xml]

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

<aop:aspectj-autoproxy />

<!-- Aspect Class를 빈으로 등록 -->

<bean id="performanceTraceAspect" class="madvirus.spring.chap05.aop.annot.ProfilingAspect" />

<bean id="writeArticleService" class="madvirus.spring.chap05.board.service.WriteArticleServiceImpl">

<constructor-arg>

<ref bean="articleDao" />

</constructor-arg>

</bean>

<bean id="articleDao" class="madvirus.spring.chap05.board.dao.MySQLArticleDao" />

<bean id="memberService" class="madvirus.spring.chap05.member.service.MemberServiceImpl" />

</beans>

위의 Code에서 <aop:aspectj-autoproxy /> 태그를 사용했는데, 이 태그를 사용하게 되면 @Aspect Annotation이 적용된

빈 객체를 Aspect로 사용하게 된다. ProfilingAspect의 경우 "execution(public \* madvirus.spring.ch05.board..\*(..))"

Pointcut에 적용되기 때문에 writeArticleService 빈과 articleDao 빈에 Aspect과 적용된다.

(즉, 두 빈에 대해 프록시 객체가 생성된다.)

실제로 Aspect가 적용되는 지 확인해 보기 위해 다음과 같은 Test Code를 작성해 보자.

[MainQuickStart2.java]

package madvirus.spring.chap05;

import madvirus.spring.chap05.board.Article;

import madvirus.spring.chap05.board.service.WriteArticleService;

import madvirus.spring.chap05.member.Member;

import madvirus.spring.chap05.member.service.MemberService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainQuickStart2 {

public static void main(String[] args) {

String[] configLocations = new String[] { "acQuickStart2.xml" };

ApplicationContext context = new ClassPathXmlApplicationContext(configLocations);

WriteArticleService articleService = (WriteArticleService) context.getBean("writeArticleService");

articleService.write(new Article());

MemberService memberService = context.getBean("memberService", MemberService.class);

memberService.regist(new Member());

}

}

MainQuickStart2 Class는 위 Code에서 작성한 asQuickStart2.xml 파일을 이용해서 ApplicationContext를 생성한 뒤

writeArticleService 빈의 write() 메서드를 호출하고, memberService 빈의 regist()를 호출하고 있다. 앞서 작성한

ProfilingAspect Class의 trace() 메서드는 madvirus.spring.ch05 패키지 및 그 하위 패키지에 있는 빈 객체의

public 메서드를 호출하면 적용되는 Around Advice 구현 메서드라고 했는데, 실제로 MainQuickStart2 Class를

실행해보면 다음과 같은 결과가 출력된다.

\*WriteArticleService.write(...) 시작

WriteArticleServiceImpl.write() 메서드 실행

\*ArticleDao.insert(...) 시작

MySQLArticleDao(...) 실행

\*ArticleDao.insert(...) 종료

\*ArticleDao.insert(...) 실행 시간 : 0ms

\*WriteArticleService.write(...) 종료

\*WriteArticleService.write(...) 실행 시간 : 0ms

\*MemberService.regist(...) 시작

MemberServiceImpl.regist() 메서드 실행

\*MemberService.regist(...) 종료

\*MemberService.regist(...) 실행 시간 : 0ms

위 Code에서 '\*'로 표시한 부분이 ProfilingAspect Class의 trace() 메서드에서 출력한 부분이다. 실행 결과를 보면 실제

빈 객체의 메서드가 호출되기 전/후로 trace() 메서드에서 실행한 내용이 출력되는 것을 확인할 수 있다.

4.2 Advice 타입 별 Class 작성

: XML 스키마 방식을 사용할 경우 <aop:around>, <aop:before>와 같은 태그를 이용해서 Advice 타입을 지정했는데,

@Aspect Annotation을 사용할 경우에는 @Before나 @Around와 같은 Annotation을 이용해서 Aspect 구현 메서드에

Advice 타입을 직접 지정한다. 본 절에서는 각 Advice 타입 별로 어떻게 구현하는지 차례대로 살펴보도록 하자.

(1) Before Advice

: Before Advice를 구현할 때는 @Before Annotation을 사용한다. 아래 Code는 @Before Annotation의 사용 예를 보여주고 있다.

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class LoggingAspect{

@Before("execution(public \* madvirus.spring.ch05..\*(..))")

public void before(){

System.out.println("[LA] 메서드 실행 전 전처리 수행");

}

}

위 Code에서 before() 메서드에 @Before Annotation을 적용했는데, 이 경우 before() 메서드는 Before Advice를 구현한

메서드를 사용한다.

@Before Annotation의 값으로는 AspectJ의 Pointcut 표현식이나 또는 @Pointcut Annotation이 적용된 메서드 이름이 올 수 있다.

위 Code에서는 직접 Pointcut 표현식을 사용한 예를 보여주고 있으며, 이 Before Advice 구현 메서드는 madvirus.spring.ch05 패키지

또는 그 하위 패키지에 있는 모든 public 메서드가 호출되기 이전에 호출된다.

@Pointcut Annotation을 적용하는 방법은 뒤에서 살펴보도록 하겠다.

앞서 XML 스키마 기반의 Before Advice 구현 메서드와 동일하게 JoinPoint 타입의 파라미터를 첫 번째 파라미터로 가질 수 있다.

@Before("execution(public \* madvirus.spring.ch05..\*(..))")

public void before(JoinPoint joinPoint){

// 대상 객체의 메서드 실행 이전에 적용할 기능 구현

...

}

(2) After Returning Advice

: After Returning Advice를 구현하려면 @AfterReturning Annotation을 Advice 구현 메서드에 적용하면 된다. 아래 Code에서 AfterReturning()

메서드에 @AfterReturning Annotation이 적용되어 있는데, 이 경우 afterReturning() 메서드가 After Returning Advice가 된다.

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterReturning;

@Aspect

public class LoggingAspect{

@AfterReturning("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()")

public void afterReturning(){

System.out.println("[LA] 메서드 실행 후 후처리 수행");

}

}

만약 Advice 구현 메서드 내에서 Advice 대상 객체가 리턴한 값을 사용하고 싶다면, 다음과 같이 returning 속성을 이용해서 리턴 값을 전달받을

파라미터의 이름을 지정해주면 된다.

@AfterReturning("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()", returning="ret")

public void afterReturning(Object ret){

System.out.println("[LA] 메서드 실행 후 후처리 수행" + ret);

}

만약 리턴된 객체가 특정 타입인 경우에 한해서 메서드를 실행하고 싶다면 다음과 같이 한정하고 싶은 타입의 파라미터를 사용하면 된다.

@AfterReturning("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()", returning="ret")

public void afterReturning(Article ret){

//대상 객체의 메서드가 정상적으로 실행된 이후에 적용할 기능 구현

}

대상 객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에 대한 정보가 필요한 경우 다음과 같이 org.aspectj.lang.JoinPoint를

파라미터로 추가한다.

@AfterReturning("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()", returning="ret")

public void afterReturning(JoinPoint joinPoint, Object ret){

//대상 객체의 메서드가 정상적으로 실행된 이후에 적용할 기능 구현

}

(3) After Throwing Advice

: @AfterThrowing Annotation을 사용하면 After Throwing Advice 메서드를 구현할 수 있다. 아래 Code는 구현 예를 보여주고 있다.

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterThrowing;

@Aspect

public class LoggingAspect{

@AfterThrowing("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()")

public void afterThrowing(){

System.out.println("[LA] 메서드 실행 중 예외 발생");

}

}

Advice 구현 메서드내에서 Advice 대상 메서드가 발생시킨 예외 객체에 접근하려면 throwing 속성을 이용해서 예외객체를 전달받을

파라미터의 이름을 지정해주면 된다.

@AfterThrowing("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()", throwing="ex")

public void afterThrowing(Throwable ex){

System.out.println("[LA] 메서드 실행 중 예외 발생" + ex.getMessage());

}

만약 특정 타입의 예외에 대해서만 처리하고 싶다면, Throwable이나 Exception이 아니라 처리하고 싶은 타입을 파라미터로

지정하면 된다. 예를 들어, 아래 Code의 경우는 발생된 예외가 ArticleNotFoundException인 경우에만 호출된다.

@AfterThrowing("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()", throwing="ex")

public void afterThrowing(ArticleNotFoundException ex){

// 대상 객체의 메서드가 예외를 발생시킨 경우에 적용할 기능 구현

}

대상 객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에 대한 정보가 필요한 경우 다음과 같이 org.aspectj.lang.JoinPoint를

파라미터로 추가한다.

@AfterThrowing("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()", throwing="ex")

public void afterThrowing(JoinPoint joinPoint, Exception ex){

// 대상 객체의 메서드가 예외를 발생시킨 경우에 적용할 기능 구현

}

(4) After Advice

: After Advice를 구현하고 싶은 경우 @After Annotation을 사용한다.

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.After;

@Aspect

public class LoggingAspect{

@After("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()")

public void afterFinally(){

System.out.println("[LA] 메서드 완료");

}

}

대상 객체 및 호출되는 메서드에 대한 정보나 전달되는 파라미터에 대한 정보가 필요한 경우 다음과 같이 org.aspectj.lang.JoinPoint를

파라미터로 명시하면 된다.

@After("madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()")

public void afterFinally(JoinPoint joinPoint){

...

}

(5) Around Advice

: @Around Annotation을 이용하면 Around Advice를 구현할 수 있다. Around Advice를 구현한 메서드는 org.aspectj.ProceedingJoinPoint를

반드시 첫 번째 파라미터로 지정해야 한다. 그렇지 않은 경우 SPRING은 예외를 발생시킨다.

아래 Code는 @Around Annotation을 이용해서 Around Advice를 구현한 Class의 예제 Code이다.

[ArticleCacheAspect.java]

package madvirus.spring.chap05.aop.annot;

import java.util.HashMap;

import java.util.Map;

import madvirus.spring.chap05.board.Article;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.core.Ordered;

@Aspect

public class ArticleCacheAspect implements Ordered {

private Map<Integer, Article> cache = new HashMap<Integer, Article>();

@Around("execution(public \* \*..ReadArticleService.\*(..))")

public Article cache(ProceedingJoinPoint joinPoint) throws Throwable {

Integer id = (Integer) joinPoint.getArgs()[0];

Article article = cache.get(id);

if (article != null) {

System.out.println("[ACA] 캐시에서 Article[" + id + "] 구함");

return article;

}

Article ret = (Article) joinPoint.proceed();

if (ret != null) {

cache.put(id, ret);

System.out.println("[ACA] 캐시에 Article[" + id + "] 추가함");

}

return ret;

}

@Override

public int getOrder() {

return 2;

}

}

4.3 @Pointcut Annotation을 이용한 Pointcut 설정

: 앞서 XML 스키마를 사용할 때에는 다음과 같이 <aop:pointcut> 태그와 Advice 설정 태그의 pointcut-ref 속성을

이용해서 Pointcut을 설정하고 여러 Advice에서 동일한 Pointcut을 참조할 수 있다.

<aop:aspect id="loggingAspect" ref="loggingAdvice" order="1">

<aop:pointcut id="publicMethod" expression="execution(public \* madvirus.spring.ch05..\*(..))"/>

<aop:before method="before" pointcut-ref="publicMethod"/>

<aop:after-rrturning method="afterReturning" pointcut-ref="publicMethod" returning="ret"/>

<aop:after-throwing method="afterThrowing" pointcut-ref="publicMethod" throwing="ex"/>

<aop:after method="afterFianlly" pointcut-ref="publicMethod"/>

</aop:aspect>

XML 스키마를 사용하는 경우와 동일하게 @Aspect Annotation을 사용하는 경우에도 @Pointcut Annotation을 이용해서

Pointcut 설정을 재사용할 수 있다. 아래 Code는 @Pointcut Annotation의 사용 예를 보여주고 있다.

@Aspect

public class ProfilingAspect {

@Pointcut("execution(public \* madvirus.spring.chap05.board..\*(..))")

private void profileTarget() {}

@Around("profileTarget()")

public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {

...

}

}

위 Code에서 @Pointcut Annotation은 Pointcut 표현식을 값으로 가지며 @Pointcut Annotation이 적용된 메서드는

리턴 타입이 void여야만 한다. 일반적으로 @Pointcut Annotation이 적용된 메서드는 위 Code에서와 같이 메서드 몸체에

Code를 갖지 않는다. (정확하게 Code를 가져도 의미가 없다.)

@Pointcut Annotation을 이용해서 Pointcut을 정의하면, 위 Code와 같이 Advice 관련 Annotation에서 @Pointcut Annotation이

적용된 메서드(이하, @Pointcut 메서드)의 이름을 이용해서 Pointcut을 참조할 수 있다. 이 때 메서드 이름은 다음과 같이

범위에 따라서 알맞게 입력해야 한다.

- 같은 Class에 위치한 @Pointcut 메서드는 '메서드이름'만 입력.

- 같은 패키지에 위치한 @Pointcut 메서드는 'Class단순이름.메서드이름'을 입력.

- 다른 패키지에 위치한 @Pointcut 메서드는 '완전한Class이름.메서드이름'을 입력.

아래 Code는 Class의 단순 이름 Class 이름을 사용하는 경우의 예이다.

@Aspect

public class LoggingAspect{

//Class의 단순 이름을 사용.

@Before("PublicPointcut.publicMethod()")

public void before(){

System.out.println("[LA] 메서드 실행 전 전처리 수행");

}

@AfterReturning(pointcut="madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()")

public void afterReturning(Object ret){

System.out.println("[LA] 메서드 실행 후 후처리 수행. 리턴값=" + ret);

}

}

@Pointcut 메서드를 사용할 때 주의할 점은 메서드 접근 제어가 그대로 적용된다는 점이다. 예를 들어, 다음과 같이

private으로 생성된 @Pointcut 메서드가 있다고 해보자.

public class CommonPointcut{

@Pointcut("execution(public \* \*(..))")

private void publicMethod(){}

}

publicMethod() 메서드는 private이기 때문에 다음과 같이 CommonPointcut Class가 아닌 다른 Class에서 publicMethod()

메서드에 정의된 Pointcut을 사용할 경우 예외가 발생한다.

@Aspect

public class LoggingAspect{

//publicMethod()는 private이므로 접근할 수 없다는 예외 발생.

@Before("CommonPointcut.publicMethod()")

public void before(){

...

}

}

@Pointcut 메서드는 @Aspect 기반의 Aspect 구현뿐만 아니라 XML 스키마의 pointcut 속성 값으로도 사용할 수 있다.

<aop:after method="afterFianlly" pointcut-ref="madvirus.spring.ch05.aop.annot.PublicPointcut.publicMethod()"/>

5. JoinPoint 사용

: Around Advice를 제외한 나머지 Advice 타입을 구현한 메서드는 org.aspectj.lang.JoinPoint 객체를 선택적으로 전달받을 수 있다.

단, JoinPoint를 파라미터로 전달받을 때에는 반드시 첫 번째 파라미터로 지정해 주어야 한다. 만약, JoinPoint를 다음과 같이

첫 번째가 아닌 파라미터로 지정한다면 SPRING은 예외를 발생시킨다.

//JoinPoint이 첫 번째 파라미터가 아닌 경우 예외 발생.

public void afterLogging(Object retVal, JoinPoint joinPoint){

...

}

JoinPoint Interface는 호출되는 대상 객체, 메서드 그리고 전달되는 파라미터 목록에 접근할 수 있는 메서드를 제공하고 있으며,

이는 다음과 같다.

- Signature getSignature() : 호출되는 메서드에 대한 정보를 구한다.

- Object getTarget() : 대상 객체를 구한다.

- Object[] getArgs() : 파라미터 목록을 구한다.

org.aspectj.lang.Signature Interface는 호출되는 메서드와 관련된 정보를 제공하기 위해 다음과 같은 메서드를 정의하고 있다.

- String getName() : 메서드의 이름을 구한다.

- String toLongString() : 메서드를 완전하게 표현한 문장을 구한다.(메서드의 리턴타입, 파라미터 타입이 모두 표시된다.)

- String toShortString() : 메서드를 축약해서 표현한 문장을 구한다.(기본 구현은 메서드의 이름만을 구한다.)

Around Advice의 경우 org.aspectj.lang.ProceedingJoinPoint를 첫 번째 파라미터로 전달받는데, ProceedingJoinPoint Interface는

프록시 대상 객체를 호출할 수 있는 proceed() 메서드를 제공하고 있다. ProceedingJoinPoint는 JoinPoint Interface를 상속 받았다.

따라서, Around Advice 역시 앞서 설명한 메서드와 Signature를 이용하여 대상 객체, 메서드 및 전달되는 파라미터에 대한 정보를

구할 수 있다.

6. 타입을 이용한 파라미터 접근

: JoinPoint의 getArgs() 메서드를 이용하면 대상 객체의 메서드를 호출할 때 사용한 인자에 접근할 수 있다고 했는데, JoinPoint를

사용하지 않고 Advice 메서드에서 직접 파라미터를 이용해서 호출 시 사용된 인자에 접근할 수 있다. Advice 구현 메서드에서

파라미터를 이용해서 대상 객체의 메서드를 호출할 때 사용한 인자에 접근하려면 다음과 같이 두 가지 작업을 진행해주면 된다.

- Advice 구현 메서드에 인자를 전달받을 파라미터를 명시한다.

- Pointcut 표현식에서 args() 명시자를 사용해서 인자 목록을 지정한다.

먼저 Advice 구현 메서드에 다음 Code와 같이 사용할 파라미터를 명시한다. 아래 Code는 두 개의 대상 객체의 메서드 호출 시 사용되는

인자가 두 개이고 각각 String 타입과 UpdateInfo 타입인 경우에 적용될 Advice 메서드가 된다.

public class updateMemberInfoTraceAdvice{

public void traceReturn(String memberId, UpdateInfo info){

System.out.println("[TA] 정보 수정 : 대상 회원=" + memberId + ", 수정 정보=" + info);

}

}

그 다음 할 작업은 XML 설정 파일에서 args() 명시자를 이용해서 인자 목록을 지정해주는 것이다. 다음은 설정 예를 보여주고 있다.

<bean id="traceAdvice" class="madvirus.spring.ch05.aop.pojo.UpdateMemberInfoTraceAdvice"/>

<aop:aspect id="traceAspect" ref="traceAdvice">

<aop:after-returning pointcut="args(memberId, info)" method="traceReturn"/>

</aop:aspect>

위 설정에서 args() 명시자가 의미하는 것은 다음과 같다.

- 대상 객체의 메서드 호출 시 인자가 두 개 전달되고,

- 이 중 첫 번째 파라미터는 traceReturn 메서드의 memberId 파라미터의 타입이고 두 번째 인자는 info 파라미터의 타입이다.

Advice 구현 메서드인 traceReturn()의 memberId 파라미터와 info 파라미터는 각각 타입이 String과 UpdateInfo이므로,

위 Pointcut 설정은 다음 Code의 update() 메서드에 Advice를 적용하게 된다.

public interface MemberService{

boolean update(String id, UpdateInfo updateInfo);

}

args() 명시자의 경우 메서드 정의에 있는 타입이 아닌 실제 메서드 호출시 전달되는 인자의 타입에 따라서 적용 여부가 결정된다.

예를 들어, update() 메서드가 다음과 같이 정의되어 있다고 해 보자.

public interface MemberService{

boolean update(String id, UpdateInfo updateInfo);

}

update() 메서드의 두 번째 파라미터 타입은 Object인데, 이렇게 메서드 선언에서 사용된 타입이 args() 명시자와 매칭되는

타입과 다르다 하더라도, 실제로 메서드에 전달되는 인자의 타입이 args() 명시자를 통해서 지정한 것과 동일하다면

Advice가 적용된다.

MemberService service = context.getBean("memberService", MemberService.class);

UpdateInfo updateInfo = new UpdateInfo();

...

//실제로 전달되는 객체의 타입이 UpdateInfo이므로 적용됨.

service.update("madvirus", updateInfo);

@Aspect Annotation을 사용하는 경우에도 XML 스키마를 사용하는 경우와 마찬가지로 Pointcut 표현식에 args() 명시자를 사용하면 된다.

@Aspect

public class UpdateMemberInfoTraceAspect{

@AfterReturning(pointcut="args(memberId, info)", returning="result")

public void traceReturn(JoinPoint joinPoint, boolean result, String memberId, UpdateInfo info){

...

}

}

6.1 인자의 이름 Mapping 처리

: 앞서 args() 명시자를 이용해서 메서드 호출시 사용된 인자를 파라미터로 전달받을 수 있다고 하였다. args() 명시자에서 지정한 이름과

Advice 구현 메서드의 파라미터 이름이 일치하는 지의 여부를 확인하는 순서는 다음과 같다.

(1) Advice Annotation 태그의 argNames 속성이나 Advice 설정 XML 스키마의 arg-names 속성에서 명시한 파라미터 이름을 사용한다.

: argNames 속성은 Advice 구현 메서드의 파라미터의 이름을 입력할 때 사용된다. 아래 Code는 사용 예를 보여주고 있다.

argNames 속성은 모든 파라미터의 이름을 순서대로 표시해서 Pointcut 표현식에서 사용된 이름이 몇 번째 파라미터인지 검색할 수 있도록 한다.

@AfterReturning(pointcut="args(memberId,info)", argNames="memberId, info")

public void traceReturn(String memberId, UpdateInfo updateInfo){

System.out.println("[TA]정보 수정 : 대상 회원=" + memberId + ", 수정 정보=" + info);

}

만약 첫 번째 파라미터의 타입이 JoinPoint나 ProceedingJoinPoint라면 첫 번째 파라미터를 제외한 나머지 파라미터의 이름을 argNames

속성에 입력해주면 된다.

@AfterReturning(pointcut="args(memberId,info)", argNames="joinPoint, memberId, info")

public void traceReturn(JoinPoint joinPoint, String memberId, UpdateInfo updateInfo){

System.out.println("[TA]정보 수정 : 결과=" + "" + ", 대상 회원=" + memberId + ", 수정 정보=" + info);

}

XML 스키마를 사용하는 경우 다음과 같이 arg-names 속성을 이용해서 파라미터 이름을 지정한다.

<aop:afterreturning pointcut="args(memberId, info)" method="traceReturn()" returning="result" arg-names="joinPoint, memberid, info"/>

(2) 디버그 옵션을 이용해서 Compile할 때 생성되는 디버그 정보를 이용해서 파라미터 이름이 일치하는지의 여부를 확인한다.

: argNames 속성 또는 arg-names 속성을 지정하지 않은 경우에는 디버그 정보를 이용한다.

(3) 디버그 옵션이 없을 경우 파라미터 개수를 이용해서 일치 여부를 유추한다.

마지막으로 디버그 정보도 없는 경우에는 파라미터 개수를 이용해서 유추한다. 예를 들어, Pointcut 표현식에서 사용한 파라미터 개수가

1개이고 실제 Advice 구현 메서드의 파라미터 개수가 1개라면 그 두 개가 일치한다고 판단한다. 만약 Pointcut 표현식에서 사용된

파라미터 개수와 실제 구현 메서드의 파라미터 개수가 다르다면 예외가 발생한다.

(1)부터 (3)까지 모두 해당되지 않는다면 IllegalArgumentException 예외를 발생시킨다.

7. AspectJ의 Pointcut 표현식

: 본 장에서는 POJO Class를 이용하여 AOP를 적용하는 두 가지 방법을 살펴보았다.

한 가지 방법은 XML 스키마를 이용하여 Aspect를 설정하는 방법이었고, 다른 한 가지는 @Aspect Annotation을 이용하여 Aspect를

설정하는 방법이었다.

이 두 가지 방법의 공통점이 있는데, 그것은 AsepctJ의 문법을 이용하여 Pointcut을 설정한다는 점이다. 예를 들어, <aop:태그>를

이용하여 AspectJ를 설정하는 경우 다음과 같이 execution 명시자를 이용하여 Advice가 적용될 Pointcut을 설정하였다.

<aop:aspect id="cacheAspect" ref="cacheAdvice">

<aop:around pointcut="execution(public \* \*..ReadArticleService.\*(..))" method="cache"/>

</aop:aspect>

AspectJ는 Pointcut을 명시할 수 있는 다양한 명시자를 제공하는데, SPRING은 메서드 호출과 관련된 명시자만을 지원하고 있다.

본 절에서는 execution 명시자와 within 명시자, 그리고 SPRING에서 추가적으로 제공하는 bean 명시자에 대해서 살펴보도록 하겠다.

execution 명시자는 Advice를 적용할 메서드를 명시할 때 사용되며, 기본 형식은 다음과 같다.

execution(수식어패턴?리턴타입패턴 Class이름패턴?이름패턴(파라미터패턴))

'수식어패턴' 부분은 생략 가능한 부분으로서 public, protected 등이 온다. '리턴타입패턴' 부분은 리턴 타입을 명시한다. 'Class이름패턴'과

'이름패턴' 부분은 Class 이름 및 메서드 이름을 패턴으로 명시한다. '파라미터패턴' 부분은 매칭될 파라미터에 대해서 명시한다.

각 패턴은 '\*'을 이용하여 모든 값을 표현할 수 있다. 또한 '..'을 이용하여 0개 이상의 의미를 표현할 수 있다.

다음은 몇 가지 예이다.

- execution(public void set\*(..)) : 리턴 타입이 void이고 메서드 이름이 set으로 시작하고, 파라미터가 0개 이상인 메서드 호출

- execution(\* madvirus.spring.ch05.\*.\*()) : madvirus.spring.ch05 패키지의 파라미터가 없는 모든 메서드 호출

- execution(\* madvirus.spring.ch05..\*.\*(..)) : madvirus.spring.ch05 패키지 및 하위 패키지에 있는, 파라미터가 0개 이상인 메서드 호출

- execution(Integer madvirus.spring.ch05..WriteArticleService.write(..)) : 리턴 타입이 Integer인 WriteArticleService 민터페이스의 wrtie() 메서드 호출

- execution(\* get\*(\*,\*)) : 이름이 get으로 시작하고 2개의 파라미터를 갖는 메서드 호출

- execution(\* read\*(Integer, ..)) : 메서드 이름이 read로 시작하고, 첫 번째 파라미터 타입이 Integer이며, 1개 이상의 파라미터를 갖는 메서드 호출

within 명시자는 메서드가 아닌 특정 타입에 속하는 메서드를 Pointcut으로 설정할 때 사용된다. 다음은 설정 예이다.

- within(madvirus.spring.chap05.board.service.WriteArticleService) : WriteArticleService Interface의 모든 메서드 호출

- within(madvirus.spring.chap05.board.service.\*) : madvirus.spring.chap05.board.service 패키지에 있는 모든 메서드 호출

- within(madvirus.spring.chap05.board..\*) : madvirus.spring.chap05.board.service 패키지 및 그 하위 패키지에 있는 모든 메서드 호출

bean 명시자는 SPRING 2.5버전부터 SPRING에서 추가적으로 제공하는 명시자로서, SPRING 빈 이름을 이용하여 Pointcut을 정의한다.

bean 명시자는 빈 이름의 패턴을 갖는다. 다음은 설정 예이다.

- bean(WriteArticleService) : 이름이 WriteArticleService인 빈의 메서드 호출

- bean(\*ArticleService) : 이름이 ArticleService로 끝나는 빈의 메서드 호출

7.1 Pointcut의 조합

: 각각의 표현식은 '&&'나 '||' 연산자를 이용하여 연결할 수 있다. 예를 들어, @Aspect Annotation을 이용하는 경우, 다음과 같이

'&&' 연산자를 사용하여 두 표현식을 모두 만족하는 Joinpoint에만 Advice가 적용되도록 할 수 있다.

@AfterThrowing(pointcut="execution(public \* get\*()) && execution(public void set\*(..))")

public void throwingLogging(){

...

}

XML 스키마를 이용하여 Advice를 설정하는 경우에도 다음과 같이 '&&'나 '||' 연산자를 사용할 수 있다.

<aop:pointcut id="propertyMethod" expression="execution(public \* get\*()) &amp;&amp; execution(public void set\*(..))"/>

설정 파일은 XML 문서이기 때문에 값에 들어가는 '&&'를 '&amp;&amp;'로 표현하였다. 이렇게 입력하는 것은 불편하므로,

SPRING은 설정 파일에서 '&&'나 '||' 대신 'and'와 'or'를 사용할 수 있도록 하고 있다. 따라서, 위 XML 설정을 다음과 같이

입력할 수도 있다.

<aop:pointcut id="propertyMethod" expression="execution(public \* get\*()) and execution(public void set\*(..))"/>

@Pointcut Annotation을 사용하는 경우 메서드 이름을 이용해서 손쉽게 두 개 이상의 Pointcut을 조합할 수 있다.

아래 Code는 @Pointcut 메서드를 이용한 Pointcut 조합의 작성 예이다.

@Pointcut("execution(public \* \*(..))")

private void publicMethod(){}

@Pointcut("within(madvirus.spring.chap05.board..\*")

private void inBoard(){}

@Pointcut("publicMethod() && inBoard()")

private void boardPublicOperation(){}

@Pointcut Annotation을 이용하면 XML 스키마를 이용하는 경우에 비해 Pointcut의 조합 및 재사용이 보다 쉽기 때문에

공통으로 사용되는 Pointcut은 @Pointcut Annotation을 이용해서 표현하는 것이 좋다.

8. Advice 적용 순서

: 하나의 JoinPoint에 한 개 이상의 Advice가 적용될 경우, 순서를 명시적으로 지정할 수 있다. Advice의 적용 순서를 명시적으로

지정하는 첫 번째 방법은 Advice 구현 Class에 다음과 같이 @Order Annotation을 적용하거나 Ordered Interface를 구현하는 것이다.

- org.springframework.core.annotation.Order Annotation을 적용한다.

- org.springframework.core.Ordered Interface를 구현한다.

아래 Code는 Ordered Interface의 구현 예제 Code를 보여주고 있다. Ordered Interface의 getOrder() 메서드를 구현해서

Advice 적용 순서 값을 입력하면 된다.

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.core.Ordered;

@Aspect

public class ArticleCacheAspect implements Ordered {

@Around("execution(public \* \*..ReadArticleService.\*(..))")

public Article cache(ProceedingJoinPoint joinPoint) throws Throwable {

...

}

@Override

public int getOrder() {

return 2;

}

}

@Order Annotation을 사용할 경우 다음과 같이 @Order Annotation의 값으로 적용 순서 값을 지정해주면 된다.

import org.aspectj.lang.annotation.Aspect;

import org.springframework.core.annotation.Order;

@Aspect

@Order(3)

public class ProfilingAspect{

...

}

적용 순서 값이 낮은 Advice의 우선 순위가 더 높다. 즉, 순서 값이 1인 Advice가 순서 값이 2인 Advice보다 우선 순위가 높다.

메서드 호출 전, 메서드 호출에 대해서는 우선 순위가 높은 Advice가 먼저 실행되고, 메서드 호출 후에 대해서는 우선 순위가

낮은 Advice가 먼저 실행된다.

XML 스키마를 사용할 경우 <aop:aspect> 태그의 order 속성을 사용해서 Advice 순서를 지정할 수 있다.

<aop:aspect id="cacheAspect" ref="cacheAdvice" order="2">

<aop:around pointcut="execution(public \* \*..ReadArticleService.\*(..))" method="cache" />

</aop:aspect>

9. 프록시 생성 방식 설정

: SPRING AOP는 대상 객체가 한 개 이상의 Interface를 구현하고 있는 경우 Java의 다이나믹 프록시 기능을 이용해서 객체를 생성하고,

Interface를 구현하고 있지 않은 경우에는 CGLIB를 이용해서 프록시 객체를 생성한다고 하였다.

만약 강제로 CGLIB를 이용해서 프록시 객체를 생성하고 싶다면 proxy-target-class 속성의 값을 지정해주면 된다.

예를 들어 <aop:config> 태그를 사용하는 경우에는 다음과 같이 설정한다.

<aop:config proxy-target-class="true">

<aop:aspect id="traceAspect" ref="traceAdvice" order="1">

<aop:after-returning pointcut="args(memberId, info)" method="traceReturn" returning="result" arg-name="joinPoint,result,memberId,info"/>

</aop:aspect>

</aop:config>

@Aspect Annotation을 이용하는 경우 <aop:aspectj-autoproxy> 태그에 proxy-target-class 속성의 값을 true로 지정함으써 CGLIB를 이용해서

프록시를 생성할 수 있다.

<aop:aspectj-proxy proxy-target-class="true"/>

10. AspectJ 사용하기

: SPRING AOP는 SPRING 빈 객체에 대해 AOP를 적용한다. 즉, SPRING Container에 의해 생성되는 Java 객체에 대해서만 AOP를 적용할 수 있다.

하지만, 경우에 따라서 SPRING Container가 생성하지 않은 다른 객체에 AOP를 적용해야 할 때도 있다. 이 경우, SPRING XML 파일에 약간의

설정만 추가해주면 AspectJ를 SPRING Application에 적용할 수 있게 된다. 본 절에서는 SPRING Application에서 AspectJ의 Aspect를

사용하는 방법과 이를 이용해서 도메인 객체에 DI를 적용하는 방법을 살펴보도록 하겠다.

10.1 AspectJ LTW 적용

: 본 장의 앞 부분에서 살펴봤듯이 AOP는 크게 Compile 시에, Class 로딩 시에, 또는 런타임 시에 적용할 수 있다. AspectJ는 이 세 가지 방식을

모두 지원하고 있으며 이 세 가지 방식 중에서 Class 로딩 시에 Aspect를 적용하는 방법은 별도의 AspectJ 개발 도구가 없어도 적용할 수 있는

방식이다. Class 로딩 시에 Aspect를 적용하기 때문에 이를 Load-time Weaving, 줄여서 LTW라고 표현한다. 본 절에서는 SPRING이 제공하는

AspectJ LTW 연동기능을 이용해서 SPRING Container가 생성하지 않는 Java 객체에 대해 AspectJ를 적용하는 방법을 살펴보도록 하겠다.

SPRING Framework에서 AspectJ LTW를 사용하는 순서는 다음과 같다.

1) @Aspect Annotation을 이용해서 Aspect를 구현한다.

2) Class 패스의 META-INF/aop.xml 파일에 Aspect 설정 및 적용할 범위를 설정한다.

3) SPRING 설정 파일에 <context:load-time-weaver>를 추가한다.

4) java 실행시 -javaagent 옵션 값으로 org.springframework.instrument-3.0.1.RELEASE-A.jar를 지정한다.

차례대로 각 단계를 살펴보도록 하자. 첫 번째로 할 작업은 @Aspect Annotation을 이용해서 Aspect를 구현하는 것이다. 예제로 사용 할 작업은

다음 Code와 같다. (이 Code는 앞서 살펴본 Code와 동일한 Code이다.)

[ProfilingAspect.java]

package madvirus.spring.chap05.aop.annot;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.core.annotation.Order;

@Aspect

public class ProfilingAspect {

@Pointcut("execution(public \* madvirus.spring.chap05.board..\*(..))")

private void profileTarget() {}

@Around("profileTarget()")

public Object trace(ProceedingJoinPoint joinPoint) throws Throwable {

String signatureString = joinPoint.getSignature().toShortString();

System.out.println(signatureString + " 시작");

long start = System.currentTimeMillis();

try {

Object result = joinPoint.proceed();

return result;

} finally {

long finish = System.currentTimeMillis();

System.out.println(signatureString + " 종료");

System.out.println(signatureString + " 실행 시간 : " + (finish - start) + "ms");

}

}

}

Aspect를 개발했다면 다음으로 할 작업은 Class패스의 META-INF/aop.xml 파일에 어떤 패키지에 어떤 Aspect를 적용할 지

설정하는 것이다. 다음 Code는 aop.xml 파일의 작성 예이다.

[aop.xml]

<?xml version="1.0" encoding="utf-8" ?>

<!DOCTYPE aspectj PUBLIC

"-//AspectJ//DTD//EN"

"http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>

<weaver>

<include within="madvirus.spring.chap05.board.service.\*" />

<include within="madvirus.spring.chap05.board.Article" />

</weaver>

<aspects>

<aspect name="madvirus.spring.chap05.aop.annot.ProfilingAspect"/>

</aspects>

</aspectj>

세 번째로 할 작업은 설정 파일에 <context:load-time-weaver> 태그를 추가하는 것이다.

[acLTW.xml]

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:load-time-weaver />

<context:spring-configured />

<bean id="readArticleService" class="madvirus.spring.chap05.board.service.ReadArticleServiceImpl" />

<bean id="writeArticleService" class="madvirus.spring.chap05.board.service.WriteArticleServiceImpl">

<constructor-arg>

<ref bean="articleDao" />

</constructor-arg>

</bean>

<bean id="articleDao" class="madvirus.spring.chap05.board.dao.MySQLArticleDao" />

<bean class="madvirus.spring.chap05.board.Article">

<property name="articleDao" ref="articleDao" />

</bean>

</beans>

<context:load-time-weaver>는 실행 환경에 따라서 알맞은 LoadTimeWeaver 구현체를 등록한다. LoadTimeWeaver는

런타임에 ClassLoader에 ClassFileTransformer를 추가해서 LTW를 가능하게 만들어 준다.

AspectJ를 만들고 관련 설정 파일을 작성했는데, 실제 AspectJ LTW가 적용되는 지 확인하기 위해 아래와 같은 Code를 작성하였다.

[MainLTW.java]

package madvirus.spring.chap05;

import madvirus.spring.chap05.board.Article;

import madvirus.spring.chap05.board.service.ArticleNotFoundException;

import madvirus.spring.chap05.board.service.ReadArticleService;

import madvirus.spring.chap05.board.service.ReadArticleServiceImpl;

import madvirus.spring.chap05.board.service.WriteArticleService;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainLTW {

public static void main(String[] args) throws ArticleNotFoundException {

String[] configLocations = new String[] { "acLTW.xml" };

ApplicationContext context = new ClassPathXmlApplicationContext(configLocations);

ReadArticleService readArticleService = new ReadArticleServiceImpl();

readArticleService.getArticleAndIncreaseReadCount(10);

WriteArticleService writeArticleService = context.getBean( "writeArticleService", WriteArticleService.class);

writeArticleService.write(new Article());

Article article = new Article();

article.increaseReadCount();

System.out.println("-----");

article = new Article();

article.increaseReadCount();

}

}

MainLTW.java Code를 보면 라이19~20은 Java 객체를 직접 생성해서 메서드를 호출하고 있고 라인22~24는 SPRING Container로부터

빈 객체를 가져와서 메서드를 호출하고 있다.

이제 마지막 남은 것은 MainLTW.java를 실행해 보는 것이다. MainLTW Class를 실핼할 때는 다음과 같이 -javaagent 옵션의 값으로

org.springframework.instrument-3.0.1.RELEASE-A.jar 파일을 지정해 주어야 한다.(아래 Code는 실제로 한줄이다.)

java -javaagent:경로/org.springframework.instrument-3.0.1.RELEASE-A.jar madvirus.spring.chap05.MainLTW

다음은 출력 결과 중 일부를 정리한 내용이다.

ReadArticleServiceImpl.getArticleAndIncreaseReadCount(..) 시작

ReadArticleServiceImpl.getArticleAndIncreaseReadCount(..) 종료

ReadArticleServiceImpl.getArticleAndIncreaseReadCount(..) 실행 시간 : 0ms

WriteArticleServiceImpl.write(..) 시작

WriteArticleServiceImpl.write() 메서드 실행

MySQLArticleDao.insert() 실행

WriteArticleServiceImpl.write(..) 종료

WriteArticleServiceImpl.write(..) 실행 시간 : 0ms

출력 결과를 보면 라인 19~20에서 생성한 ReadArticleServiceImpl 객체와 라인 22~24의 SPRING Container에서 가져온 WriteArticleServiceImpl

객체에 모두 ProfilingAspect가 적용된 것을 확인할 수 있다.

10.2 <context:load-time-weaver> 설정

: SPRING LTW 지원을 사용하려면 LoadTimeWeaver를 설정해 주어야 한다. 다음과 같이 <context:load-time-weaver /> 태그를 설정 파일에

추가만 하면 런타임에 따라 알맞은 LoadTimeWeaver가 설정 된다.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:load-time-weaver />

...

<context:load-time-weaver />는 기본적으로 DefaultContextLoadTimeWeaver Class를 사용하는데, 이 Class는

런타임 환경에 알맞은 LoadTimeWeaver를 자동으로 찾아서 설정해 준다. 런타임 환경에 별로 사용되는 LoadTimeWeaver는

다음과 같다.

[Runtime] [구현 Class]

===================================================================

BEA Weblogic10 WeblogicLoadTimeWeaver

--------------------------------------------------------------------------------------------------------------------------------------

org.springframework.instrument-3.0.1.RELEASE-A.jar를 instrumentationLoadTimeWeaver

Java 에이전트로 사용한 경우(java -javaagent로 실행)

--------------------------------------------------------------------------------------------------------------------------------------

톰캣에서 TomcatInstrumentableClassLoader 설정시 ReflectiveLoadTimeWeaver

--------------------------------------------------------------------------------------------------------------------------------------

GlassFish GlassFishLoadTimeWeaver

--------------------------------------------------------------------------------------------------------------------------------------

오라클 OC4J OC4JLoadTimeWeaver

--------------------------------------------------------------------------------------------------------------------------------------

기타(Resin등에서 사용) ReflectiveLoadTimeWeaver

일반적인 Java Application에서 SPRING이 지원하는 LTW를 사용하려면 SPRING이 제공하는 Java 에이전트를 -javaagent

옵션으로 설정해주면 된다.

java -javaagent:경로/org.springframework.instrument-3.0.1.RELEASE-A.jar ...

Weblogic, OC4J, 그리고 Resin의 경우는 실행 스크립트를 수정할 필요 없이 SPRING 설정만으로(즉, <context:load-time-weaver>

태그 추가만으로) SPRING이 지원하는 LTW 기능을 사용할 수 있다.

10.3 AspectJ를 이용한 객체의 DI 처리

: SPRING은 Container를 통해서 생성되는 객체들에 대한 위존 관계를 처리해주는데, 이로 인해 Code에서는 의존하는 Class에 대한

직접적인 커플링(coupling)을 제거할 수 있게 된다. 하지만, SPRING Container가 생성하지 않는 객체에 대해서는 SPRING이 제공하는

DI 기능을 사용할 수 없다.

SPRING이 제공하는 AspectJ 지원 기능을 사용하면 SPRING Container가 생성하는 객체뿐만 아니라 SPRING Container 밖에서 생성되는

객체에 대해서도 DI를 적용할 수 있게 된다. 즉, SPRING이 제공하는 DI를 이용해서 모든 객체의 프로퍼티를 설정할 수 있다.

AspectJ를 이용해서 SPRING Container 밖에서 생성되는 객체에 DI를 적용하려면 다음과 같은 작업을 수행하면 된다.

1) Class패스에 org.springframework.aspects-3.0.1.RELEASE-A.jar 파일을 추가한다.(@Configuration Annotation 제공)

2) SPRING DI를 적용할 Class에 @Configuration Annotation을 적용한다.

3) SPRING 설정 파일에 해당 Class를 위한 설정을 추가한다.

4) @Configuration Annotation이 적용된 Class를 SPRING LTW 지원을 이용해서 로딩한다.

5) SPRING 설정 파일에 <context:spring-configured> 태그를 추가한다.

특정 Class의 객체를 초기화 할 때 SPRING에 설정된 정보를 이용해서 DI를 적용하려면 다음과 같이 @Configurable Annotation을

Class에 적용해야 한다.

import org.springframework.beans.factory.annotation.Configurable;

@Configurable

public class Article {

private int id;

private ArticleDao articleDao;

public Article() {

}

public Article(int id) {

this.id = id;

}

public int getId() {

return id;

}

public void increaseReadCount() {

articleDao.updateReadCount(id, 1);

}

//SPRING 설정된 정보를 이용해서 DI 처리

public void setArticleDao(ArticleDao articleDao) {

this.articleDao = articleDao;

}

}

@Configurable Annotation이 적용된 Class는 SPRING의 다른 빈과 마찬가지로 set 스타일의 프로퍼티 설정 메서드를

통해서 SPRING Container가 생성한 빈 객체를 전달 받을 수 있다.

@Configurable Annotation이 적용된 Class는 객체를 생성할 때 SPRING Container는 Class의 완전한 이름과 동일한

이름을 갖는 빈 설정을 이용한다. SPRING 설정에서 빈 객체의 이름을 설정하지 않을 경우 빈 객체의 완전한 Class 이름을

빈의 이름으로 사용하므로, 위 Article Class의 객체를 생성할 때 다음과 같은 빈 설정을 사용하게 된다.

<bean class="madvirus.spring.chap05.board.Article">

<property name="articleDao" ref="articleDao"/>

</bean>

위 설정에서 articleDao 프로퍼티의 값으로 articleDao 빈 객체를 설정하고 있으므로, 새로운 Article 객체를 생성할 때 마다

해당 객체의 setArticleDao() 메서드를 이용해서 articleDao 빈 객체를 전달하게 된다.

다른 이름의 빈 객체를 이용하고 싶다면 @Configurable Annotation의 값으로 빈 객체의 이름을 지정하면 된다.

@Configurable("articleConfig")

public class Article {

private int id;

private ArticleDao articleDao;

public Article() {

}

//SPRING 설정된 정보를 이용해서 DI 처리

public void setArticleDao(ArticleDao articleDao) {

this.articleDao = articleDao;

}

...

}

@Configurable Annotation이 적용된 Class 객체를 생성할 때 SPRING 설정을 적용하려면 @Configurable Annotation이 적용된

Class에 AspectJ를 이용해서 Weaving을 수행해야 한다. Compile 방식이나 LTW 방식 등을 이용할 수 있는데, 본 절에서는 앞에서

살펴본 SPRING의 LTW 지원을 이용하는 방법을 설명할 것이다.

@Configuration Annotation이 적용된 Class에 LTW를 적용하려면 먼저 META-INF/aop.xml파일에 해당 Class를 Weaving 대상으로

추가해 주어야 한다.

<?xml version="1.0" encoding="utf-8" ?>

<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>

<weaver>

...

<include within="madvirus.spring.chap05.board.Article" />

</weaver>

<aspects>

...

</aspects>

</aspectj>

META-INF/aop.xml 파일에 @Configuration Annotation을 적용한 Class를 Weaving 대상으로 추가했다면, 이제 SPRING 설정 파일에

<context:spring-configured> 태그를 추가하면 설정이 완료된다.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:load-time-weaver />

<context:spring-configured />

...

<bean class="madvirus.spring.chap05.board.Article">

<property name="articleDao" ref="articleDao"/>

</bean>

</beans>

이제 남은 작업은 @Configuration Annotation이 적용된 Class의 객체를 생성하고 사용하는 것이다. SPRING LTW 지원을

사용할 것이므로 앞에서 살펴봤듯이 런타임에 따라 알맞은 LoadTimeWeaver를 설정하면 된다.(예를 들어, 일반 Application

에서는 -javaagent옵션을 사용한다.) 아래 Code는 사용 예를 보여주고 있다.

String[] configLocations = new String[] { "acLTW.xml" };

ApplicationContext context = new ClassPathXmlApplicationContext( configLocations);

//Article 객체가 생성될 때 SPRING Container를 통해 DI가 설정됨

//즉, article.setArticleDao() 를 통해서 articleDao 빈 객체가 설정됨

Article article = new Article();

article.increaseReadCount();

article = new Article();

article.increaseReadCount();

SPRING의 LTW 지원 기능은 Article 객체가 생성될 때 Article 객체의 setArticleDao() 메서드를 호출해서 articleDao 빈 객체를 전달한다.

따라서, Code에서 직접 setArticleDao() 메서드를 호출하지 않음에도 불구하고 article.increaseReadCount() 메서드는 articleDao 객체를

사용할 수 있게 된다.

(1) @Configurable Annotation 적용 Class의 설정 순서

: SPRING LTW 지원 기능은 @Configuration Annotation이 적용된 객체의 생성자가 호출된 다음에 DI를 적용한다. 다라서, DI를 통해서

전달받을 객체를 생성자에서 사용할 경우 NullPointerException 속성 값을 true로 지정해 주면 된다.

@Configurable(preConstruction=true)

public class Article {

private int id;

private ArticleDao articleDao;

public Article() {

//articleDao 객체에 접근 가능

}

...

}

(2) @Configurable Annotation 적용 Class의 자동 설정

: @Configuration Annotation이 적용된 Class에 DI를 적용하기 위해 SPRING 설정 파일에 관련된 빈 설정을 추가했었다.

자동 설정 기능을 사용하면 SPRING 설정 파일에 Mapping되는 빈 설정을 추가하지 않아도 프로퍼티를 설정할 수 있게 된다.

@Configuration Annotation의 autowire 속성값을 Autowire.BY\_TYPE 이나 Autowire.BY\_NAME으로 설정하면 자동 설정

기능이 적용된다.

import org.springframework.beans.factory.annotation.Autowire;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable(autowire=Autowire.BY\_TYPE)

public class Article {

private int id;

private ArticleDao articleDao;

public setArticleDao(ArticleDao articleDao) {

this.articleDao = articleDao;

}

...

}

위 Code와 같이 autowire 속성의 값을 Autowire.BY\_TYPE으로 설정하면 ArticleDao 타입의 빈 객체가 setArticleDao() 메서드에

전달된다. 비슷하게 Autowire.BY\_NAME을 사용하면 이름이 articleDao이 빈 객체가 setArticleDao() 메서드에 전달된다.

autowire 속성을 사용할 때, Annotation의 dependencyCheck 속성을 사용하면 도메인 객체를 생성한 후 기본 데이터 타입 및

String을 제외한 Reference 타입의 프로퍼티가 모두 설정되었는지의 여부를 검사할 수 있다. 아래 Code는 설정 예이다.

@Configurable(autowire=Autowire.BY\_NAME, dependencyCheck=true)

public class Article {

private int id;

private ArticleDao articleDao;

public setArticleDao(ArticleDao articleDao) {

this.articleDao = articleDao;

}

...

}

autowire 속성 값을 Autowire.BY\_NAME으로 지정했으므로, Article 객체를 생성할 때 이름이 articleDao인 빈 객체를 setArticleDao()

메서드에 전달하게 된다. 그런데 만약 이름이 articleDao인 빈 객체가 존재하지 않는다면, Article 객체가 생성될 때 의존이 설정되지

않으므로 SPRING은 예외를 발생시킨다.

4장에서 살펴봤던 @Autowire Annotation이나 @Resource Annotation을 이용해서 프로퍼티를 설정할 수도 있다. 아래 Code는

@Resource Annotation을 사용한 경우의 예를 보여주고 있다.

import javax.annotation.Resource;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable

public class Article {

private int id;

private ArticleDao articleDao;

@Resource(name="articleDao")

public setArticleDao(ArticleDao articleDao) {

this.articleDao = articleDao;

}

...

}