Spring AOP

다룰 내용

- ✓ **Spring AOP Introduction**
- ✓ **AspectJ Pointcut**
- ✓ **Schema Based AOP**
- ✓ **Annotation Based AOP**

기존 OOP 방식 한계

- ✓ 중복되는 코드 발생 : 복사 & 붙이기
- ✓ 지저분한 코드 : 핵심 기능 이외의 공통기능의 코드, 가독성이 안 좋아짐
- ✓ 생산성의 저하
- ✓ 재활용의 저하
- ✓ 변화의 어려움

기존 방식으로 코드 작성

AOP의 목적

- ✓ 관심의 분리 : 공통 관심사항과 핵심 관심사항
- ✓ 공통관심사항 : 전체 시스템에서 사용되는 기능

  예> 로그처리, 트랜잭션, 인증, 보안, 예외처리 등등...
- ✓ 핵심관심사항 : 일반 업무 프로세스

  예> 게시글 등록, 수정, 삭제, 조회 등등....
- ✓ 핵심관심 사항은 기존의 **OOP**기술로 쉽게 분리가 가능하나 공통관심은 기존의 **OOP** 기술로 분리가 쉽지 않음
- ✓ 프로그램 전반에 사용되는 공통 모듈 적용의 효율성을 높여줌
- ✓ 각 소스에 들어있던 공통 모듈을 설정을 통하여 적용함
- ✓ 공통 기능의 변경사항이 있을 경우 사용하는 쪽은 변경할 필요가 없음

AOP 기술

- ✓ 핵심 모듈의 코드를 직접 건들이지 않고 필요한 기능이 동작하도록 하는 것
- ✓ 핵심 모듈의 코드 사이에 공통 모듈이 동작하도록 처리
- ✓ **핵심 기능에 공통 기능을 삽입하기 위한 방법 세가지**
  1. **컴파일 시점에 코드에 공통 기능추가**
  2. **클래스 로딩 시점에 바이트 코드에 공통 기능 추가**
  3. **실행 시에 프록시 객체를 생성해서 공통 기능 추가**
- ✓ **스프링 AOP는 프록시 객체를 자동으로 생성, 개발자는 공통 기능만 구현한 클래스 작성**

Spring AOP의 특징

- ✓ 표준 자바 클래스로 작성
- ✓ **Runtime** 시점에 **Advice**가 적용
- ✓ 메서드 단위의 조인포인트만 가능

AOP 용어

이름	설명
조인포인트(Joinpoint)	공통 관심 모듈의 기능이 삽입되어 동작될 수 있는 위치 메서드 호출, 필드 값 변경시점등, 스프링은 메서드 호출만 가능
포인트컷	어떤 클래스의 조인포인트를 사용할 것인지 결정
어드바이스(Advice)	언제 공통 기능이 핵심 기능에 적용할 지를 정의 예> 게시판 목록 조회 시(핵심 기능)에 실행에 걸린 시간(공통 기능)을 출력한다.
타겟(Target)	핵심 기능을 수행하는 객체
위빙, 크로스컷팅	포인트 컷에 의해서 결정된 조인포인트에 지정된 어드바이스를 삽입하는 과정
어드바이저(Advisor)	포인트 컷과 어드바이스를 합쳐놓은 것

Advice 용어

이름	설명
Before Advice	대상 객체의 메서드 호출 전에 공통 기능 실행
After Returning Advice	대상 객체의 메서드가 익셉션 없이 실행된 이후에 공통 기능 실행
After Throwing Adivce	대상 객체의 메서드를 실행하는 도중 익셉션이 발생한 경우에 공통 기능 실행
After Advice	대상 객체의 메서드를 실행하는 도중에 익셉션이 발생했는지의 여부에 상관없이 메서드 실행 후 공통 기능을 실행
Around Advice	대상 객체의 메서드 실행 전, 후 또는 익셉션 발생 시점에 공통 기능을 실행

pom.xml 작성

✓ **dependency 추가**

- **spring-context , spring-aop, aspectjrt, aspectjweaver 추가**

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>X.x.x.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>X.x.x.RELEASE</version>
</dependency>
```

pom.xml 작성

```
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>X.x.x</version>
</dependency>

<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>X.x.x</version>
</dependency>
```

스키마 기반 AOP

✓ **<aop:*>** 에 해당하는 태그 정보

태그명	설명
<aop:config>	AOP 설정 태그 중 루트, AOP 관련 설정 정보임을 나타냄
<aop:aspect>	Pointcut 과 advice 설정을 위해 사용
<aop:pointcut>	포인트컷을 지정하기 위해 사용, <aop:config>, <aop:aspect> 하위태그
<aop:advisor>	포인트컷과 어드바이스 클래스를 직접 연결할 경우 사용

스키마 기반 AOP

✓ Advice 관련 태그 정보

태그명	설명
<aop:before>	메서드 실행 전에 호출
<aop:after-returning>	메서드가 정상 종료 후 호출
<aop:after-throwing>	메서드가 예외를 발생시킬 때 적용, catch 블록과 유사
<aop:after>	메서드 실행 후 무조건 호출, finally 와 유사
<aop:around>	메서드 호출 이전, 이후, 예외 발생 등 모든 시점에 적용 가능

XML 작성

✓ **around Advice**

실행시간 측정 공통 기능 클래스 등록

```
<bean id="time" class="_04_schema.TimeAspect" />

<aop:config>
    <aop:pointcut id="pCut" expression="execution( * execute(..))" />
    <aop:aspect id="myAspect" ref="time">
        <aop:around method="executeTime" pointcut-ref="pCut" />
    </aop:aspect>
</aop:config>
```

Pointcut 지시자

- ✓ **execution**

```
<aop:pointcut id="pCut" expression="execution( * execute(..))" />
```

execution ([수식어]? [리턴타입] [클래스이름]? [메서드이름] ([파라미터])

수식어	리턴타입	클래스이름	메소드이름	파라미터
생략가능	필수	생략가능	필수	
public	*	패키지포함		
protected				
private				

execution(public * kr.co.mlec.board.service.BoardService.list (..))

AspectJ 에서의 사용되는 표현식

✓ 와일드 카드 연산자

와일드 카드	설명
*	모든값
..	0개 이상의 문자
+	주어진 타입의 임의의 서브 클래스나 서브 인터페이스

AspectJ 에서의 사용되는 표현식

✓ 패턴 매칭

형식	설명
set* (*)	set으로 시작하고 파라미터가 1개인 메서드
set*(*, *)	set으로 시작하고 파라미터가 2개인 메서드
set*(String, ..)	set으로 시작하고 첫번째 파라미터가 String 이고 파라미터의 개수가 2개 이상인 메서드
set* (..)	set으로 시작하는 모든 메서드
* main(..)	리턴타입 상관없이 이름이 main 인 모든 메서드

AspectJ 에서의 사용되는 표현식

✓ **Type**

형식	설명
Java.io.*	Java.io 밑에 있는 모든 클래스 지칭
ch4.aop..*	ch4.aop 패키지 및 그 하위 모든 패키지의 클래스 지칭
Number+	+ 는 서브 타입 지칭, Number 타입으로 표현되는 모든 하위 클래스
!(Number+)	Number 타입으로 표현되지 않는 모든 클래스

AspectJ 에서의 사용되는 표현식

✓ **Modifier 설정**

형식	설명
public static void main(..)	매개변수에 상관없이 접근제한자가 public인 main 함수
!private * * (..)	접근제한자가 private 가 아닌 모든 메서드
* main(..)	접근제한자 상관하지 않는 형식

JAVA 작성

✓ 공통기능 클래스

```
public Object executeTime(ProceedingJoinPoint pjp) throws Throwable {
    try {
        // 핵심 기능 메서드 호출
        Object retVal = pjp.proceed();
        return retVal;
    } finally {
        // Object pjp.getTarget( ) -> 핵심 기능 클래스
        String clzName = pjp.getTarget().getClass().getName();
        String metName = pjp.getSignature().getName();
        코드 생략..
    }
}
```

어노테이션 기반 AOP

- ✓ **XML** 설정부분을 간소화 하기 위해 어노테이션을 활용하는 방식
- ✓ **XML** 문서에 어노테이션을 활용하기 위한 선언 필요
  - **<aop:aspectj-autoproxy />**
- ✓ 클래스 위쪽에 **@Aspect** 선언

스키마 기반 AOP

✓ 스키마와 관련된 어노테이션

태그명	설명
<aop:before>	`@Before("execution(public * anno.*Controller.*(..))")`
<aop:after-returning>	`@AfterReturning(pointcut="execution(public * anno.*Controller.*(..))", returning="retVal")`
<aop:after-throwing>	`@AfterThrowing( pointcut="execution(public * anno.*Controller.*(..))", throwing="ex")`
<aop:after>	`@After("execution(public * anno.*Controller.*(..))")`
<aop:around>	`@Around("execution(public * anno.*Controller.*(..))")`

XML 작성

```
<context:component-scan base-package="anno" />

-- @Aspect 어노테이션이 적용된 클래스를 Advice 로 사용함

<aop:aspectj-autoproxy />
```

JAVA 작성

✓ **Aspect 클래스**

```
@Aspect
@Component
public class TimeAspect {
    -- 포인트컷과 어드바이스를 설정
    @Around("execution(public * _05_anno.*Controller.*(..))")
    public Object executeTime(ProceedingJoinPoint pjp)
            throws Throwable {
        코드 생략...
    }
}
```