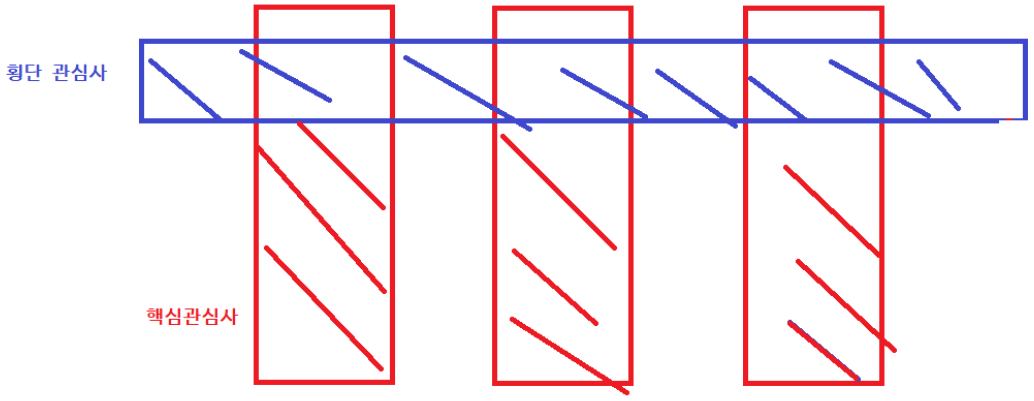
2024-10-07 4일차

Spirng AOP

AOP(Aspect Oriented Programming : 관점지향 프로그래밍)

- 객체지향 프로그래밍을 보안하기 위한 프로그래밍 개념이다.
    - 객체지향 프로그래밍을 적용해서 애플리케이션을 개발하더라도 다수의 객체에 분산되어
      중복적으로 존재하는 공통관심사(공통기능)는 여전히 존재한다.
- AOP는 다수의 객체에 분산되어 중복적으로 존재하는 공통관심사(공통기능)을 별도의 모듈로 분리하여 핵심기능과 엮어어서 처리할 수 있는 방법을 제공한다.



핵심관심사: 핵심기능 횡단관심사: 공통기능

AOP의 주요 개념

- **Target**
    - 핵심기능이 구현되어 있는 객체다.
- **Join Point**
    - 공통기능이 삽입되어 **적용될 지점**이다.

        - **핵심 기능이 구현되어있는 메소드라고 보면 된다.**
    - 메소드 실행, 클래스 초기화, 객체 생성 등 다양한 Join Point가 있다.
    - Spring AOP는 메소드 실행 Join Point를 지원한다.
- **Advice (when(어느 시점에) + what(무엇을))**
    - 공통기능을 분리해서 별도의 모듈로 구현체로 구현한 것이다.
    - Advice는 Joint Point에 결합되어 동작되는 싯점에 따라서

      Before Advice, After Advice, Around Advice로 구분할 수 있다.

      Advice의 실행싯점

        - Before
            - 대상 메소드 실행 전에 공통기능이 실행된다.

```
@Before(”within(kr.co.jhta.service.*)”)
pu1blic void logging(JoinPoint joinPoint){

{

@Before(”within(kr.co.jhta.service.*.*(..))”)
pu1blic void logging(JoinPoint joinPoint){

{
```

        - After
            - 대상 메소드가 실행이 완료되었을 때 실행된다.(예외가 발생했던,오류없이 실행 완료되었던 상관없다.)

```
@After("within(kr.co.jhta.service.*)")
public void doSomething(JoinPoint joinPoint){
    ...
}
```

        - After-Returing

- 대상 메소드가 오류없이 실행이 완료되었을 때 실행행된다.

```
@AfterRetuning(
    pointcut="within(kr.co.jhta.service.*)",
    returning="value" // 대상 메소드가 오류없이 실행되었기 때문에 대상 메소드의 반환값을 활용할 수 있다.
                      // returning은 대상 메소드의 반환값을 전달받을 매개변수 이름을 지정한다.
)
pulbic void doAccessCheck(JoinPoint joinPoint, Object value){
    System.out.println("대상메소드의 반환값:" + value)
}
```

- After-Throwing
  - 대상 메소드에서 예외가 발생했을 때 실행된다.

```
@AfterThrowing(
    pointcut="within(kr.co.jhta.service.*)",
    throwing="ex" // 대상메소드에서 발생한 예외를 전달받을 매개변수 이름을 지정한다.
)
pubic void doSomething(JoinPoint joinPoint, Exception ex){
    System.out.println("대상메소드에서 발생한 예외:" + ex);
}
```

- Around → 이거하나로 위에있는거 다 만들 수 있음
  - 대상 메소드 실행 전/후에 실행된다.

```
Around 기본적인형태

@Around("within(kr.co.jhta.service.*)")
public Object doSomething(ProceedingJoinPoint joinPoint){
    try{
```

```
        // 이 위치에 작성하는 코드는 Before Advice와 동
일한 싯점에 실행된다.

        Object value = joinPoint.proceed();  <----
----- 대상 메소드를 실행하는 코드다.

        // 이 위치에 작성하는 코드는 After-Returing Adv
ice와 동일한 싯점에 실행된다.

        return value;
        } catch (Exception ex){
                // 이 위치에 작성하는 코드는 After-Thro
wing Advice와 동일한 싯점에 실행된다.
        } finally{
                // 이 위치에 작성하는 코드는 After Advi
ce와 동일한 싯점에 실행된다.
        }
}
```

- **Pointcut (where) 적용규칙**
  - 공통기능을 어떤 클래스의 어떤 Join Point에 결합하여 사용할 것인지를 결정하는 규칙이다.
  - 가장 일반적인 적용규칙은 특정 클래스의 모든 메소드 실행이다.

```
    poin-cut 표현식

포인트컷 지정자**
    within: 공통기능이 적용될 대상 클래스를 지정한다.
    execution: 공통기능이 적용될 대상 메소드를 지정한다.

포인트컷 와일드카드
    *  : Any 아무거나 하나
    .. : 패키지경로 : 현재 패키지 및 모든 하위 패키지
              매개변수    : 매개변수 상관없이

pointcut="within(com.example.service.*)"
```

```
com.example.service 패키지의 아무 클래스나 적용대상이다.

pointcut="within(com.example.service..*)"
com.example.service 패키지의 아무클래스나 적용대상이다.

com.example.service의 모든 하위 패키지의 아무 클래스나  적용대상
이다.

pointcut="within(com.example.servicee.*ServiceL mpi)"
com.example.service 패키지의 클래스 중에서 클래스이름이**

ServiceLmple로 끝나는 아무 클래스나 적용대상이다.

pointcut="within(com.example.service.AccountService)"
com.example.service 패키지의 AccountService클래스만 적용대상이
다.

pointcut="execution"(*com.example.service.AccountServic
e.get*(..));
                    * :                   반환타입이 상관
없다.
                    com.example.service : 패키지명
                    AccountService      : 클래스명
                    get*(..)            : 메소드명

pointcut="execution"(*com.example.service.*.*(..))"
pointcut="execution"(*com.example.service..*.*(..))"
pointcut="execution"(*com.example.service.*ServiceLmpi*
(..))"
pointcut="execution"(*com.example.service.AccountSerivce
*(..))"
pointcut="execution"(*com.example.service.*.get*(..))"
pointcut="execution"(*com.example.service.*.add*(..)) ||
execution(*com.example.service.*.insert*(..))"
                    || execution(*com.example.service.
*.update*(..)) || execution(*com.example.service.*.delet
e*(..))"
```

```
execution(* service.*ServiceImpl.get*(*))
    get으로 시작하는 메소드중에서 매개변수가 한개있는 메소드
execution(* service.*ServiceImpl.get*(..))
    get으로 시작하는 아무 메소드(매개변수가 0 혹은 여러 개)
```

- **Aspect**
  - Advice와 Pointcut을 조합한 것이다.
  - 적용할 공통기능과 적용규칙이 결합된 것으로 AOP 적용을 위한 최종 패키지다.
- **Weaving**
  - Pointcut적용규칙에 의해 결정된 Joint Point에 Advice를 엮는 과정이다.
  - Weaving은 기존의 핵심기능 코드에 전혀 영향을 주지 않으면서 필요한 공통기능을 추가하는 핵심과정이다

*** 요즘은 XML 잘 안씀**

XML 스키마

스키마 = 구조

XMLNS

```
이름
xmlns="http://www.springframework.org/schema/beans"
= 식별자 중복X 이름 유일해야함
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
= "xxxx" 식별자
   xmlns:context="http://www.springframework.org/schema/
context" = "context" 식별자
   xmlns:aop="http://www.springframework.org/schema/aop"
= "aop" 식별자

정의파일
    xsi:schemaLocation="http://www.springframework.org/s
chema/beans
    https://www.springframework.org/schema/beans/spring-
beans.xsd
```

```
    http://www.springframework.org/schema/context
    https://www.springframework.org/schema/context/sprin
g-context.xsd
    http://www.springframework.org/schema/aop
    https://www.springframework.org/schema/aop/spring-ao
p.xsd">
```

코드리뷰

코드리뷰.zip

Spring의 데이터베이스 엑세스

- spring은 spring-jdbc.jar, spring-tx,jar, spring-orm.jar 등의 모듈을 제공한다.
- spring boot는 spring-data-jdbc, spring-data-jpa, spring-data-mongoodb 모듈을 제공하고 있으면
  위의 모듈들은 데이터베이스 엑세스 방법을 추상화 해서 일관된 방법으로 데이터베이스 엑세스가 가능하다.

- spring은 데이터베이스 엑세스에 커넥션 풀을 활용한다.
  * 데이터베이스 엑세스 작업을 첫번째 단계는 데이터베이스와 연결된 커넥션풀 객체를 생성해서 스프링 컨테이너에
    등록시키는 작업이다.
- spring은 데이터베이스 엑세스 작업 중 예외가 발생하면 DataAccessException을 발생시킨다.
  * DataAccessException은 RuntimeException의 하위 클래스이기 때문에 예외처리를 강제하지 않는다.
  * 애플리케이션 개발에 사용되는 많은 데이터베이스의 오류코드를 조사해서 오류에 맞는 적절한 예외를 발생시킨다.
  (발생되는 예외는 전부 DataAccessException의 하위 클래스다. 예외클래스의 이름이 매우 구체적이다.)

- spring은 데이터베이스 엑세스 작업은 선언적 트랜잭션 처리를 지원한다.
  * 선언적 트랜잭션 처리는 트랜잭션처리를 위한 코드를 작성할 필요없어 간단한 설정만으로
    트랜잭션처리가 적용되도록 한다.
- spring은 다양한 외부라이브러리(mybatis,JPA 등)와 쉽게 연동해서 데이터베이스 엑세스 작업을 구현할 수 있다.