

6. AOP와 트랜잭션



Objectives

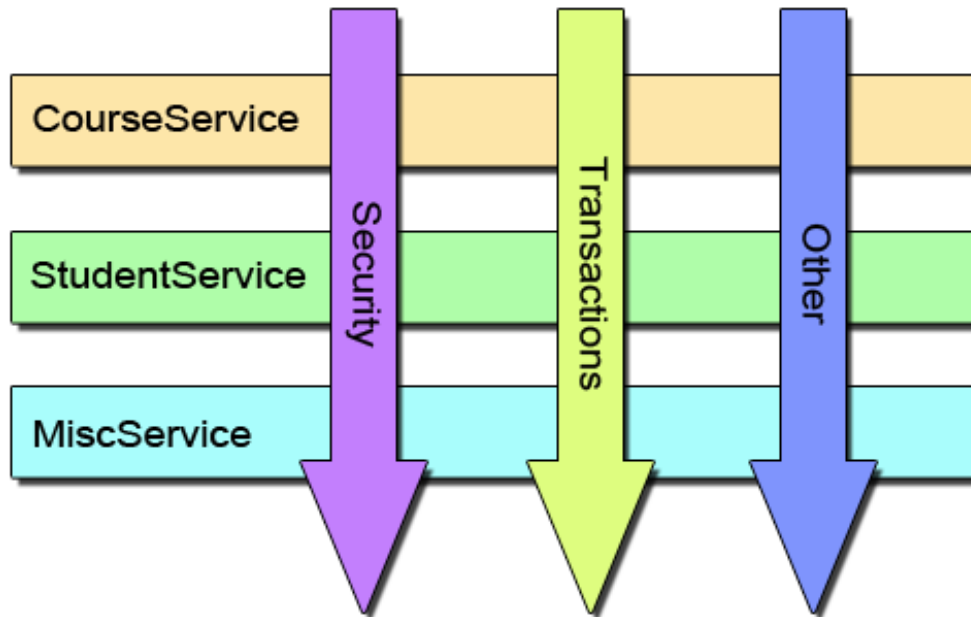
- AOP의 개념 이해와 적용
- AOP의 용어와 기법
- 트랜잭션의 처리 적용

1. AOP라는 패러다임

관점 지향 프로그래밍

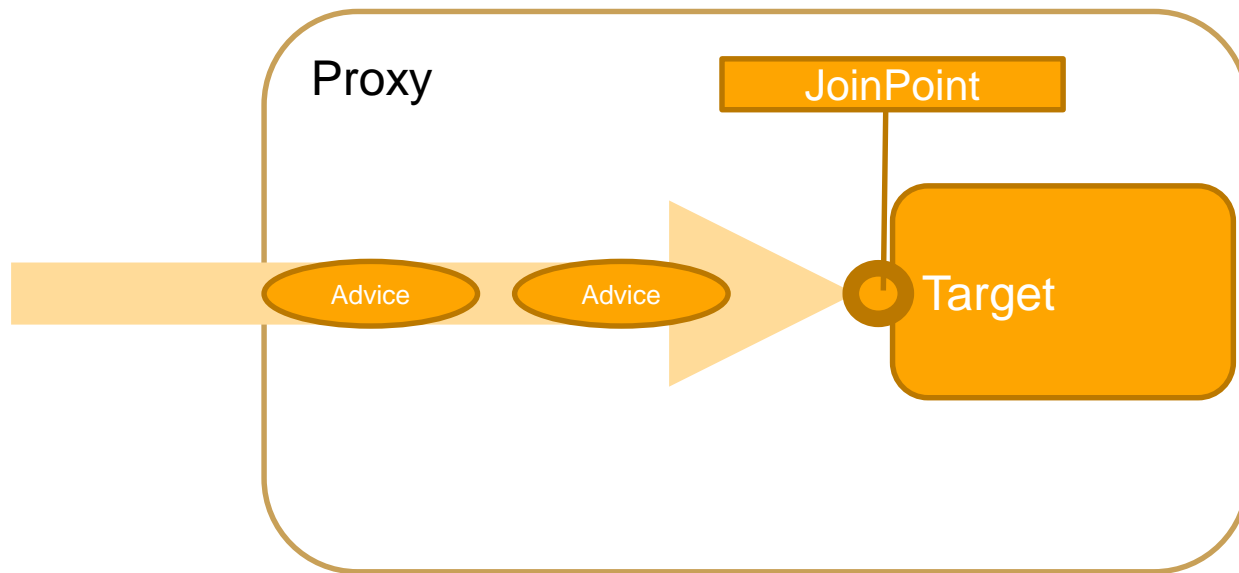
■ 관심사의 분리

- 핵심로직은 아니지만 코드에 전반적으로 반복적으로 사용하며, 필요한 로직들
- 횡단 관심사(cross-concerns)



주요 용어

- Aspect: 추상 명사로 횡단 관심사를 의미
 - ex> 로깅, 보안, 트랜잭션등
- Advice: 횡단 관심사를 구현한 객체
- Target: 핵심로직을 가지고 있는 객체
- Proxy 객체: Target객체 + Advice

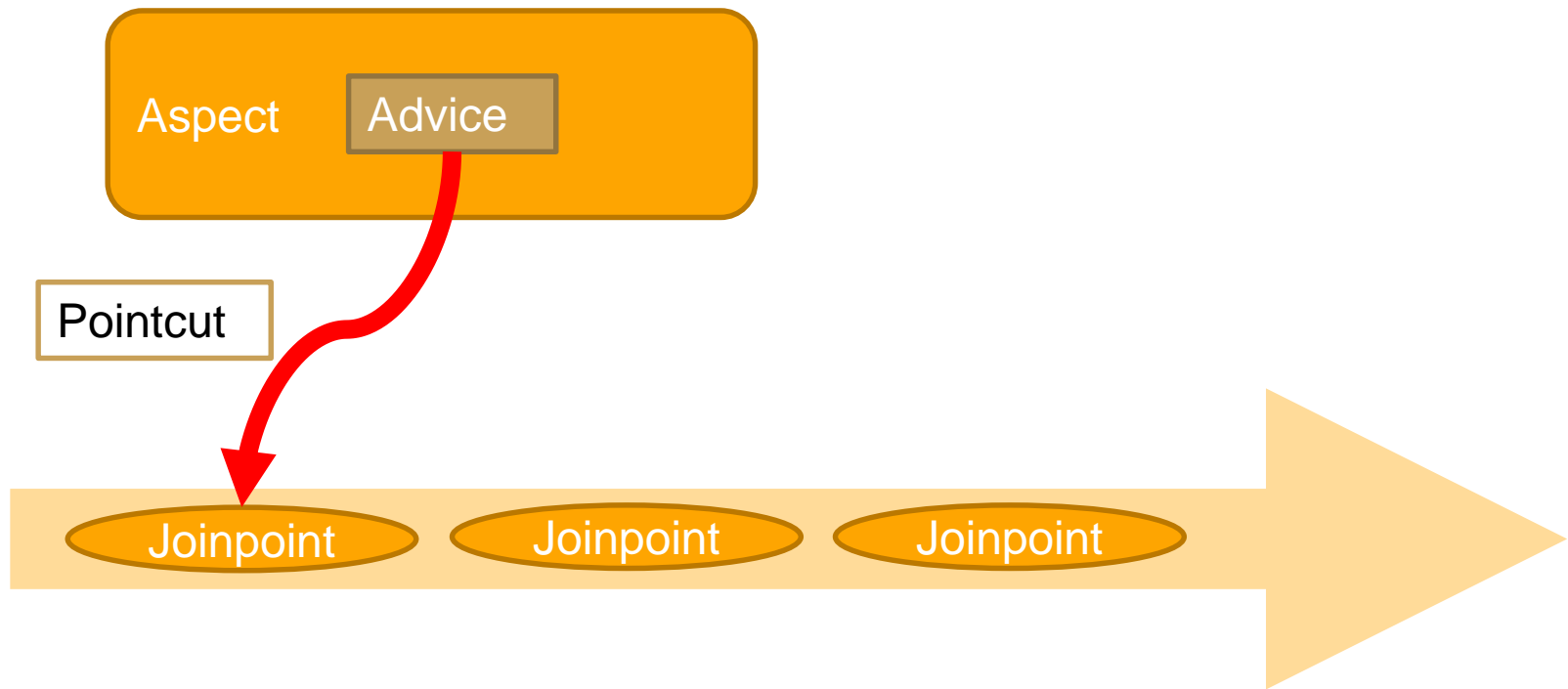


- JoinPoint

- Advice의 적용대상 – 스프링에서는 target의 메서드

- Pointcut

- 여러 JoinPoint들 중에서 Advice가 적용되는 select 기준



Advice의 종류

■ 실제로 개발하는 관심사 코드

| 구분 | 설명 |
|-------------------------------|------------------------------------------------------------------------|
| Before Advice | Target의 JoinPoint를 호출하기 전에 실행되는 코드입니다. 코드의 실행 자체에는 관여할 수 없습니다. |
| After Returning Advice | 모든 실행이 정상적으로 이루어진 후에 동작하는 코드입니다. |
| After Throwing Advice | 예외가 발생한 뒤에 동작하는 코드입니다. |
| After Advice | 정상적으로 실행되거나 예외가 발생했을 때 구분 없이 실행되는 코드입니다. |
| Around Advice | 메서드의 실행 자체를 제어할 수 있는 가장 강력한 코드입니다. 직접 대상 메서드를 호출하고 결과나 예외를 처리할 수 있습니다. |

Pointcut

- Advice를 어떤 JoinPoint에 결합할 것인지를 결정하는 설정

| 구분 | 설명 |
|------------------------------|------------------------------------------|
| execution(@execution) | 메서드를 기준으로 Pointcut을 설정합니다. |
| within(@within) | 특정한 타입(클래스)을 기준으로 Pointcut을 설정합니다. |
| this | 주어진 인터페이스를 구현한 객체를 대상으로 Pointcut을 설정합니다. |
| args(@args) | 특정한 파라미터를 가지는 대상들만을 Pointcut으로 설정합니다. |
| @annotation | 특정한 어노테이션이 적용된 대상들만을 Pointcut으로 설정합니다. |

AOP의 실습

■ pom.xml 수정

```
<properties>
    <java-version>1.8</java-version>
    <org.springframework-version>5.0.7.RELEASE</org.springframework-version>
    <org.aspectj-version>1.9.0</org.aspectj-version>
    <org.slf4j-version>1.7.25</org.slf4j-version>
</properties>

<!-- AspectJ -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${org.aspectj-version}</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.aspectj/aspectjweaver -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>${org.aspectj-version}</version>
</dependency>
```

서비스 계층 설계/Advice 작성/Pointcut

- Advice에는 어노테이션을 이용해서 Pointcut 설정

```
@Aspect
@Log4j
@Component
public class LogAdvice {
    @Before( "execution(* org.zerock.service.SampleService*.*(..))" )
    public void logBefore() {
        Log.info("=====");
    }
}
```

AOP처리를 위한 설정

root-context.xml의 일부

```
<context:annotation-config></context:annotation-config>
```

```
<context:component-scan
```

```
  base-package="org.zerock.service"></context:component-scan>
```

```
<context:component-scan
```

```
  base-package="org.zerock.aop"></context:component-scan>
```

```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

```
8- @Override
9   public Integer doAdd(String str1, String str2) throws Exception {
10
11       return Integer.parseInt(str1) + Integer.parseInt(str2);
12   }
13
```

AOP 테스트

- AOP가 적용되면 일반 객체가 아니라 Proxy처리가 된 객체가 생성되는 것을 확인할 수 있음

```
@Setter(onMethod = @__( { @Autowired } ))  
private SampleService service;  
  
@Test  
public void testClass() {  
    log.info(service);  
    log.info(service.getClass().getName());  
}
```

```
INFO : org.zerock.service.SampleServiceTests - org.zerock.service.SampleServiceImpl@31ea9581  
INFO : org.zerock.service.SampleServiceTests - com.sun.proxy.$Proxy20
```

args를 이용하는 파라미터 추적

- args라는 특별한 변수를 이용해서 파라미터를 설정하고 기록 가능

```
@Before("execution(* org.zerock.service.SampleService*.doAdd(String, String)) &&
args(str1, str2)")

public void logBeforeWithParam(String str1, String str2) {
    log.info("str1: " + str1);
    log.info("str2: " + str2);
}
```

```
INFO : org.zerock.aop.LogAdvice - =====
INFO : org.zerock.aop.LogAdvice - str1: 123
INFO : org.zerock.aop.LogAdvice - str2: 456
INFO : org.zerock.service.SampleServiceTests - 579
```

@AfterThrowing

- 예외 발생을 감지해서 AOP로 처리

```
@AfterThrowing(pointcut = "execution(*  
org.zerock.service.SampleService*.*(..))", throwing="exception")  
public void logException(Exception exception) {  
    Log.info("Exception....!!!!");  
    Log.info("exception: "+ exception);  
}
```

@Around와 ProceedingJoinPoint

- @Around의 경우는 직접 해당 메소드를 실행할 것을 결정할 수있음
- 파라미터로 ProceedingJoinPoint로 지정하고 사용해야 함
- ProceedingJoinPoint의 메서드
 - getTarget(): 실제로 실행해야 하는 객체
 - proceed(): 실제 메서드의 실행

```
@Around("execution(* org.zerock.service.SampleService.*(..))")
```

```
public Object logTime( ProceedingJoinPoint pjp) {  
    long start = System.currentTimeMillis();  
    Log.info("Target: " + pjp.getTarget());  
    Log.info("Param: " + Arrays.toString(pjp.getArgs()));  
    //invoke method  
    Object result = null;  
    try {  
        result = pjp.proceed();  
    } catch (Throwable e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
    long end = System.currentTimeMillis();  
    Log.info("TIME: " + (end - start));  
    return result;  
}
```

2. 스프링에서 트랜잭션 처리

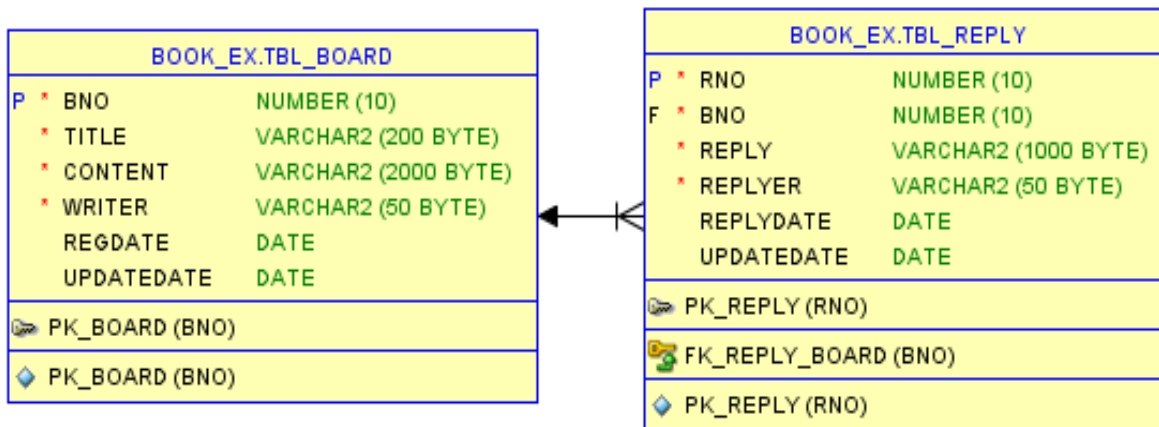
트랜잭션

- 비즈니스 용어에서 ‘거래’의 의미
- 하나의 ‘거래’는 여러 번의 데이터베이스 관련 작업이 이루어지므로 이런 작업들을 ‘하나의 트랜잭션으로 처리’한다고 표현
- 트랜잭션의 원칙 ACID

| | |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 원자성(Atomicity) | 하나의 트랜잭션은 모두 하나의 단위로 처리되어야 합니다. 좀 더 쉽게 말하자면 어떤 트랜잭션이 A 와 B 로 구성된다면 항상 A , B 의 처리결과는 동일한 결과이어야 합니다. 즉 A 는 성공했지만, B 는 실패할 경우 A , B 는 원래의 상태로 되돌려져야만 합니다. 어떤 작업이 잘못되는 경우 모든 것은 다시 원점으로 되돌아가야만 합니다. |
| 일관성(Consistency) | 트랜잭션이 성공했다면 데이터베이스의 모든 데이터는 일관성을 유지해야만 합니다. 트랜잭션으로 처리된 데이터와 일반 데이터 사이에는 전혀 차이가 없어야만 합니다. |
| 격리(Isolation) | 트랜잭션으로 처리되는 중간에 외부에서의 간섭은 없어야만 합니다. |
| 영속성(Durability) | 트랜잭션이 성공적으로 처리되면, 그 결과는 영속적으로 보관되어야 합니다. |

댓글과 게시물의 반정규화

- 정규화를 하면 여러번의 조인이 필요하고, 성능의 저하가 오는 경우 '반 정규화'를 통해서 해결
- 반정규화는 자주 사용하는 값을 컬럼으로 작성해서 유지하는 방식
- 게시물과 댓글의 숫자의 경우



트랜잭션 설정

- root-context.xml의 트랜잭션 관련 설정 추가

```
<bean id="transactionManager"  
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource"></property>  
</bean>  
<tx:annotation-driven />
```

트랜잭션 설정의 테스트

- 2 개 이상의 테이블에 insert 작업을 하나의 '트랜잭션' 이라고 가정하고 트랜잭션 설정이 없는 경우와 있는 경우를 비교
- 스프링의 경우 @Transactional을 이용해서 설정 가능
 - 메서드의 @Transactional 설정이 가장 우선시 됩니다.
 - 클래스의 @Transactional 설정은 메서드보다 우선순위가 낮습니다.
 - 인터페이스의 @Transactional 설정이 가장 낮은 우선순위입니다.

댓글과 트랜잭션 설정

- tbl_board 테이블에 댓글 수를 의미하는 replycnt 컬럼 추가
- 댓글의 추가와 삭제시에 replycnt는 트랜잭션하에 관리되어야 함

```
alter table tbl_board add (replycnt number default 0);
```

```
update tbl_board set replycnt = (select count(rno) from  
tbl_reply where tbl_reply.bno = tbl_board.bno);
```

```
<update id="updateReplyCnt">
```

```
    update tbl_board set replycnt = replycnt + #{amount} where bno = #{bno}
```

```
</update>
```

ReplyServiceImpl의 수정

...생략...

@Transactional

@Override

```
public int register(ReplyVO vo) {  
    Log.info("register....." + vo);  
    boardMapper.updateReplyCnt(vo.getBno(), 1);  
    return mapper.insert(vo);  
}
```

...생략...

@Transactional

@Override

```
public int remove(Long rno) {  
    Log.info("remove...." + rno);  
    ReplyVO vo = mapper.read(rno);  
    boardMapper.updateReplyCnt(vo.getBno(), -1);  
    return mapper.delete(rno);  
}
```