

# Spring Framework

## 15. AOP 어플리케이션 작성(2)

# CONTENTS

1

Aspect 클래스 선언 및 설정

2

Aspect 클래스 구현

3

Aspect 클래스 테스트

# 학습 목표

- Aspect 클래스 선언 및 설정에 대하여 이해할 수 있습니다.
- Aspect 클래스 구현에 대하여 이해할 수 있습니다.
- Aspect 클래스 테스트에 대하여 이해할 수 있습니다.



## 1. Aspect 클래스 선언 및 설정

## ■ Spring AOP의 구현 방식

### 01 XML 기반의 POJO 클래스를 이용한 AOP 구현

- ◉ 부가기능을 제공하는 Advice 클래스를 작성한다.
- ◉ XML 설정 파일에 <aop:config>를 이용해서 애스펙트를 설정한다.  
(즉, 어드바이스와 포인트컷을 설정함)

### 02 @Aspect 어노테이션을 이용한 AOP 구현

- ◉ @Aspect 어노테이션을 이용해서 부가기능을 제공하는 Aspect 클래스를 작성한다. 이때 Aspect 클래스는 어드바이스를 구현하는 메서드와 포인트컷을 포함한다.
- ◉ XML 설정 파일에 <aop:aspectj-autoproxy />를 설정한다.

## ■ @Aspect 어노테이션

- ◉ Aspect 클래스 선언할 때 @Aspect 어노테이션을 사용한다.
- ◉ AspectJ 5버전에 새롭게 추가된 어노테이션이다.
- ◉ @Aspect 어노테이션을 이용할 경우 XML 설정 파일에 어드바이스와 포인트컷을 설정하는 것이 아니라 클래스 내부에 정의할 수 있다.
- ◉ <aop:aspectj-autoproxy> 태그를 설정파일에 추가하면 @Aspect 어노테이션이 적용된 Bean을 Aspect로 사용 가능하다.

## ■ Aspect 클래스 정보

- **클래스명** : LoggingAspect.java
- **클래스 기능** : 이 Aspect 클래스는 4가지 유형의 어드바이스와 포인트컷을 설정하여 타겟 객체의 파라미터와 리턴값, 예외 발생 시 예외 메시지를 출력하는 기능을 제공
- **Advice 유형** : Before, AfterReturning, AfterThrowing, After
- **구현 메서드명** : before(JoinPoint joinPoint)  
afterReturning(JoinPoint joinPoint, Object ret)  
afterThrowing(JoinPoint joinPoint, Throwable ex)  
afterFinally(JoinPoint joinPoint)

## ■ Aspect 클래스 선언 및 설정

1. 클래스 선언부에 **@Aspect 어노테이션**을 정의한다.
2. 이 클래스를 애스펙트로 사용하려면 Bean으로 등록해야 하므로 **@Component 어노테이션**도 함께 정의한다.

```
package myspring.aop.annot;  
  
import org.aspectj.lang.JoinPoint;  
@Component  
@Aspect  
public class LoggingAspect {
```

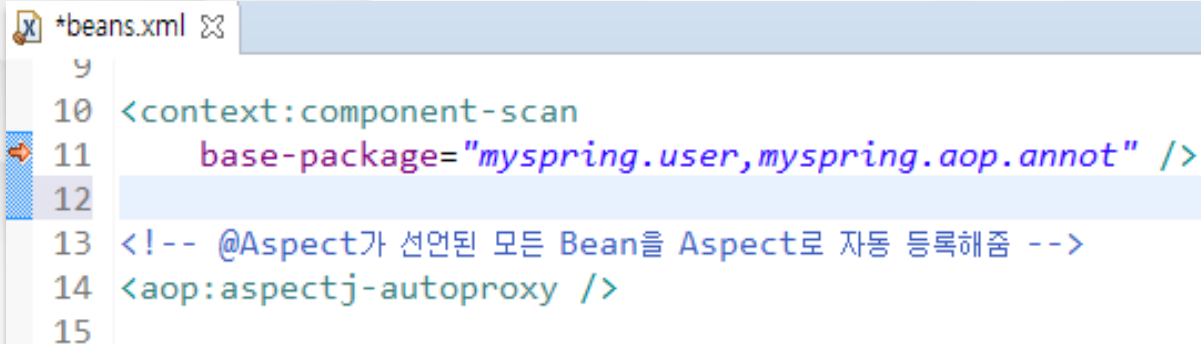
```
beans.xml  
10 <context:component-scan  
11     base-package="myspring.user,myspring.aop.annot" />  
12
```



## ■ Aspect 클래스 선언 및 설정

3. XML 설정파일에 `<aop:aspectj-autoproxy />` 선언한다.

이 선언은 Bean으로 등록된 클래스 중에서 `@Aspect`가 선언된 클래스를 모두 애스펙트로 자동 등록 해주는 역할을 한다.



```
*beans.xml
9
10 <context:component-scan
11     base-package="myspring.user,myspring.aop.annot" />
12
13 <!-- @Aspect가 선언된 모든 Bean을 Aspect로 자동 등록해줌 -->
14 <aop:aspectj-autoproxy />
15
```

A person's hands are shown holding a smartphone, with the screen glowing. The background is a dark, out-of-focus city night scene with warm, bokeh-style light spots in yellow and orange. A semi-transparent dark banner is at the bottom, containing a yellow decorative element and the title text.

## 2. Aspect 클래스 구현

### ■ Advice를 정의하는 어노테이션

- ◉ Advice를 정의하기 위하여 아래와 같은 어노테이션을 제공한다.

@Before("pointcut")

- 타겟 객체의 메서드가 실행되기 전에 호출되는 어드바이스
- JoinPoint를 통해 파라미터 정보를 참조할 수 있다.

@After("pointcut")

- 타겟 객체의 메서드가 정상 종료됐을 때와 예외가 발생했을 때 모두 호출되는 어드바이스
- 리턴값이나 예외를 직접 전달받을 수는 없다.

@Around("pointcut")

- 타겟객체의 메서드가 호출되는 전 과정을 모두 담을 수 있는 가장 강력한 기능을 가진 어드바이스

### ■ Advice를 정의하는 어노테이션

- ◉ Advice를 정의하기 위하여 아래와 같은 어노테이션을 제공한다.

```
@AfterReturning(pointcut="",  
returning="")
```

- 타겟 객체의 메서드가 정상적으로 실행을 마친 후에 호출되는 어드바이스
- 리턴값을 참조할 때는 returning 속성에 리턴값을 저장할 변수 이름을 지정해야 한다.

```
@AfterThrowing(pointcut="",  
throwing="")
```

- 타겟 객체의 메서드가 예외가 발생하면 호출되는 어드바이스
- 발생한 예외를 참조할 때는 throwing 속성에 발생한 예외를 저장할 변수 이름을 지정해야 한다.

### ■ Before 어드바이스

- ◉ @Before 어드바이스를 이용해서 실행되는 타겟 객체의 메서드명과 파라미터를 출력하는 어드바이스이다.
- ◉ 아래의 before 메서드는 myspring 패키지 또는 그 하위 패키지에 있는 모든 public 메서드가 호출되기 이전에 호출된다.

```
15 @Before("execution(public * myspring..*(..))")
16 public void before(JoinPoint joinPoint) {
17     String signatureString = joinPoint.getSignature().getName();
18
19     System.out.println("@Before [ " + signatureString + " ] 메서드 실행 전처리 수행");
20
21     for (Object arg : joinPoint.getArgs()) {
22         System.out.println("@Before [ " + signatureString + " ] 아규먼트 " + arg);
23     }
24 }
```

### ■ AfterReturning 어드바이스

- ◉ @AfterReturning 어드바이스를 이용해서 실행되는 타겟 객체의 메서드명과 리턴값을 출력하는 어드바이스이다.
- ◉ 아래의 afterReturning 메서드는 myspring.user.service 패키지 하위에 있는 모든 public 메서드가 정상 종료된 이후에 호출된다.
- ◉ 리턴값을 참조할 때는 returning 속성을 이용해서 리턴 값을 담은 변수 이름을 지정해야 한다.

```
26 @AfterReturning(pointcut="execution(public * myspring.user.service.*(..))", returning="ret")
27 public void afterReturning(JoinPoint joinPoint, Object ret) {
28     String signatureString = joinPoint.getSignature().getName();
29     System.out.println("@AfterReturning [ " + signatureString + " ] 메서드 실행 후처리 수행");
30     System.out.println("@AfterReturning [ " + signatureString + " ] 리턴값=" + ret);
31
32 }
```

### ■ AfterThrowing 어드바이스

- ◉ @AfterThrowing 어드바이스를 이용해서 실행되는 타겟 객체의 메서드명과 예외 메시지를 출력하는 어드바이스이다.
- ◉ 아래의 afterThrowing 메서드는 클래스명이 UserService로 시작되는 클래스에 속한 모든 메서드가 예외가 발생된 이후에 호출된다.
- ◉ 발생한 예외를 참조할 때는 throwing 속성을 이용해서 예외객체를 담은 변수 이름을 지정해야 한다.

```
28 @AfterThrowing(pointcut="execution(* *..UserService*.*(..))",  
29               throwing="ex")  
30 public void afterThrowing(JoinPoint joinPoint, Throwable ex) {  
31     String signatureString = joinPoint.getSignature().getName();  
32     System.out.println("@AfterThrowing [ " + signatureString + " ] 메서드 실행 중 예외 발생");  
33     System.out.println("@AfterThrowing [ " + signatureString + " ] 예외=" + ex.getMessage());  
34 }
```

### ■ After 어드바이스

- ◉ @After 어드바이스를 이용해서 실행되는 타겟 객체의 메서드명을 출력하는 어드바이스이다.
- ◉ 아래의 afterFinally 메서드는 메서드명이 User로 끝나는 메서드들이 정상 종료됐을 때와 예외가 발생했을 때 모두 호출된다.
- ◉ 반드시 반환해야 하는 리소스가 있거나 메서드 실행 결과를 항상 로그로 남겨야 하는 경우에 사용할 수 있다.  
하지만 리턴 값이나 예외를 직접 전달받을 수는 없다.

```
42 @After("execution(* *..*.*User(..))")
43 public void afterFinally(JoinPoint joinPoint) {
44     String signatureString = joinPoint.getSignature().getName();
45     System.out.println("@After [ " + signatureString + " ] 메서드 실행 완료");
46 }
```



A person's hands are shown holding a smartphone with a white screen. The background is dark with out-of-focus, colorful bokeh lights in shades of yellow, orange, and blue. A semi-transparent dark banner is at the bottom, containing a yellow decorative bar and the section title.

### 3. Aspect 클래스 테스트

#### ■ Aspect 클래스 테스트(1)

- ◉ UserService Bean의 getUser 메서드를 호출하면, Advice가 적용된 것을 확인해 볼 수 있다.

```
*UserClient.java
22 public class UserClient {
23     @Test
24     public void getUserTest() {
25         service = context.getBean(UserService.class);
26         UserVO user = service.getUser("gildong");
27         System.out.println("User 정보 : " + user);
28     }
```

#### ■ Aspect 클래스 테스트(1)

- UserService Bean의 getUser 메서드를 호출하면, Advice가 적용된 것을 확인해 볼 수 있다.

```
<terminated> UserClient (2) [JUnit] C:\Program Files (x86)\Java\jre1.8.0_91\bin\javaw.exe (2016. 7. 28. 오후 12:10:42)
```

```
UserService.getUser(..) 시작
```

```
@Before [ getUser ] 메서드 실행 전처리 수행
```

```
@Before [ getUser ] 마규먼트 gildong
```

```
@Before [ read ] 메서드 실행 전처리 수행
```

```
@Before [ read ] 마규먼트 gildong
```

```
@After [ getUser ] 메서드 실행 완료
```

```
@AfterReturing [ getUser ] 메서드 실행 후처리 수행
```

```
@AfterReturing [ getUser ] 리턴값=User [userId=gildong, name=홍길동, gender=남, city=서울]
```

```
UserService.getUser(..) 종료
```

```
UserService.getUser(..) 실행 시간 : 2189 ms
```

```
User 정보 : User [userId=gildong, name=홍길동, gender=남, city=서울]
```

#### ■ Aspect 클래스 테스트(2)

- ◉ UserService Bean의 getUser 메서드가 예외 발생 시, Advice가 적용된 것을 확인해 볼 수 있다.

```
*UserClient.java ✕  
22 public class UserClient {  
23     @Test  
24     public void getUserTest() {  
25         service = context.getBean(UserService.class);  
26         UserVO user = service.getUser("gildong");  
27         System.out.println("User 정보 : " + user);  
28     }
```

#### ■ Aspect 클래스 테스트(2)

- UserService Bean의 getUser 메서드가 예외 발생 시, Advice가 적용된 것을 확인해 볼 수 있다.

UserService.getUser(..) 시작

@Before [ getUser ] 메서드 실행 전처리 수행

@Before [ getUser ] 아규먼트 gildong

@Before [ read ] 메서드 실행 전처리 수행

@Before [ read ] 아규먼트 gildong

@After [ getUser ] 메서드 실행 완료

@AfterThrowing [ getUser ] 메서드 실행 중 예외 발생

@AfterThrowing [ getUser ] 예외=PreparedStatementCallback; bad SQL grammar

[select \* from users where userid = ?]; nested exception

UserService.getUser(..) 종료

UserService.getUser(..) 실행 시간 : 857 ms

### Aspect 클래스 테스트(3)

- UserService Bean의 updateUser 메서드를 호출하면, Advice가 적용된 것을 확인해 볼 수 있다.

```
@Test
public void updateUserTest() {
    service.updateUser(new UserVO("gildong", "홍길동2", "남2", "경기2"));
    System.out.println(service.getUser("gildong"));
}
```

## ■ Aspect 클래스 테스트(3)

UserService.updateUser(..) 시작

@Before [ updateUser ] 메서드 실행 전처리 수행

@Before [ updateUser ] 아규먼트 User [userId=gildong, name=홍길동2, gender=남2, city=경기2]

@Before [ update ] 메서드 실행 전처리 수행

@Before [ update ] 아규먼트 User [userId=gildong, name=홍길동2, gender=남2, city=경기2]

갱신된 Record with ID = gildong

@After [ updateUser ] 메서드 실행 완료

@AfterReturing [ updateUser ] 메서드 실행 후처리 수행

@AfterReturing [ updateUser ] 리턴값=null

UserService.updateUser(..) 종료

UserService.updateUser(..) 실행 시간 : 1034 ms

UserService.getUser(..) 시작

@Before [ getUser ] 메서드 실행 전처리 수행

@Before [ getUser ] 아규먼트 gildong

@Before [ read ] 메서드 실행 전처리 수행

@Before [ read ] 아규먼트 gildong

@After [ getUser ] 메서드 실행 완료

@AfterReturing [ getUser ] 메서드 실행 후처리 수행

@AfterReturing [ getUser ] 리턴값=User [userId=gildong, name=홍길동2, gender=남2, city=경기2]

UserService.getUser(..) 종료

UserService.getUser(..) 실행 시간 : 139 ms

User [userId=gildong, name=홍길동2, gender=남2, city=경기2]



학습정리



지금까지 **[AOP 어플리케이션 작성(2)]**에 대해서 살펴보았습니다.

## Aspect 클래스 선언 및 설정

`@Aspect, @Component , <aop:aspectj-autoproxy />`

## Aspect 클래스 구현

`@Before, @AfterReturning, @AfterThrowing, @After`

## Aspect 클래스 테스트

- ◉ `execution()` 지시자에 설정된 메서드를 호출
- ◉ `LoggingAspect`가 적용되어 출력된 로그를 확인