제8장 스프링 AOP 프레임워크(2)

- AOP 개념
- 중요용어정리
- **AOP 구현**

좋은 소프트웨어

- 비즈니스 개발에서 가장 중요한 원칙
  - 낮은 결합도 : 스프링의 DI/IoC와 관련
    - 비즈니스 컴포넌트를 구성하는 객체들의 결합도를 떨어뜨릴 수 있어 의존 관계를 쉽게 변경할 수 있음
  - 높은 응집도 : AOP와 관련
    - 부가적인 공통 코드들을 효율적으로 관리
    - 핵심관리와 공통횡단관리를 분리하여 사용하면, 구현하는 메소드에서 실제 비즈니스 로직만으로 구성할 수 있어, 더욱 간결하고 응집도 높은 코드를 유지할 수 있음
- 스프링
  - 객체지향 + 의존관계주입 + AOP 조합
  - 유연하고, 높은 보존성과 견고한 SW를 개발할 수 있음

스프링 AOP구현 방법

- AOP 프레임워크
  - AspectJ (eclipse.org/aspectj)
  - Jboss AOP (labs.jboss.com/portal/jbossaop/index.html)
  - Spring AOP (www.springframework.org)

- AOP 구현의 세가지 방법
  - POJO Class를 이용한 AOP구현
  - 스프링 API를 이용한 AOP구현
  - 애노테이션(Annotation) 을 이용한 AOP 구현

POJO기반 AOP구현 - 설정파일 작성(1/5)

- XML 스키마를 이용한 AOP 설정
  - aop 네임스페이스와 XML 스키마 추가

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       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-X.X.xsd
          http://www.springframework.org/schema/aop
          http://www.springframework.org/schema/aop/spring-aop-X.X.xsd">
</beans>
```

POJO기반 AOP구현 - 설정파일 작성(2/5)

- XML 스키마를 이용한 AOP 설정

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       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-X.X.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-X.X.xsd">
    <!--횡단관심에 등록하는 Advice등록 -->
    <bean id="writelog" class="org.kosta.spring.LogAspect"/>
    <!-- AOP 설정-->
    <aop:config>
        <aop:pointcut id="publicmethod"  expression="execution(public * org.kosta.spring..*.*(..))"/>
        <aop:aspect id="logingAspect" ref="writelog">
            <aop:around pointcut-ref="publicmethod" method="logging"/>
        </aop:aspect>
    </aop:config>
    <bean id="targetclass" class="org.kosta.spring.TargetClass"/>
</beans>
```

POJO기반 AOP구현 - 설정파일 작성(3/5)

- AOP 설정 태그의 선언
  - <aop:config> : aop설정의 root 태그 – weaving들의 묶음
  - <aop:aspect> : Aspect 설정 – 하나의 weaving에 대한 설정
  - <aop:pointcut> : Pointcut 설정- expression속성설정값에 따라 필터링 되는 메소드가 달라짐
    - 포인트컷 표현식

*	org.sample..	*XXX	.	get*(..)
리턴타입	패키지경로	클래스명		메소드명및매개변수

  - Advice 설정 태그들
    - <aop:before> - 메소드 실행 전 실행될 Advice
    - <aop:after-returning> - 메소드 정상 실행 후 실행될 Advice
    - <aop:after-throwing> - 메소드에서 예외 발생시 실행될 Advice
    - <aop:after>-메소드 정상 또는 예외 발생 상관없이 실행될 Advice – finally
    - <aop:around>-모든 시점에서 적용시킬 수 있는 Advice 구현

*위빙(Weaving):포인트컷을 지정한 핵심 관심 메소드가 호출될 때, advice에 해당하는 횡단 관심 메소드가 삽입되는 과정을 의미.
이 위빙을 통해 비즈니스 메소드를 수정하지 않고도 횡단 관심에 해당하는 기능을 추가하거나 변경할 수 있음

POJO기반 AOP구현 - <aop:aspect> (4/5)

- 한 개의 Aspect (공통 관심 기능)을 설정
- ref 속성을 통해 공통 기능을 가지고 있는 bean을 연결
- id는 이 태그의 식별자를 설정
- 자식 태그로 <aop:pointcut> advice관련 태그가 올 수 있다

```
<bean id="advice" class=k2.sample.service.SampleAdvice"></bean>
<!-- AOP 설정
<aop:config>
    <aop:pointcut id="allPointcut" expression="execution(* k2.sample..*Impl.*(..))" />
            // allPointcnt은 리턴타입과 매개변수를 무시하고 k2.sample 패키지로 시작하는 모든 클래스 중에서 이름이 Impl로 끝나는
            // 클래스의 모든 메소드를 포인트컷으로 설정
    <aop:pointcut id="selectPointcut" expression="execution(* k2.sample..*Impl.select*(..))" />
            // selectPointcnt은 리턴타입과 매개변수를 무시하고 k2.sample 패키지로 시작하는 모든 클래스 중에서 이름이 Impl로 끝나는
            // 클래스의 select으로 시작되는 메소드를 포인트컷으로 설정

    <aop:aspect ref="advice"> //포인트컷 참조
        <aop:before pointcut-ref="selectPointcut" method="beforeLogic" />  // select으로 시작되는 메소드만 호출하여 반응함
    </aop:aspect>
</aop:config>
-->
```

POJO기반 AOP구현 - <aop:pointcut>(5/5)

- Pointcut(공통기능이 적용될 곳)을 지정하는 태그
  - <aop:config>나 <aop:aspect>의 자식 태그
  - AspectJ 표현식을 통해 pointcut 지정
  - 속성 :
    - id : 식별자로 advice 태그에서 사용됨
    - expression : pointcut 지정

<aop:pointcut id="publicmethod" expression="execution(public * org.myspring..*.*(..))"/>

```
<aop:config>
        <aop:aspect id="logingAspect" ref="writelog">
            <aop:pointcut id="publicmethod"
                                expression="execution(public * org.myspring..*.*(..))"/>
            <aop:around pointcut-ref="publicmethod" method="logging"/>
        </aop:aspect>
</aop:config>
```

POJO기반 AOP구현 – AspectJ 표현식(1/4)

- AspectJ에서 지원하는 패턴 표현식(일명 포인트컷 표현식)
- 스프링은 메서드 호출 관련 명시자만 지원

명시자(제한자패턴? 리턴타입패턴 패키지패턴?이름패턴(파라미터패턴))
- ?는 생략가능

- 명시자
  - **execution** : 실행시킬 메소드 패턴을 직접 입력하는 경우
  - within : 메소드가 아닌 특정 타입에 속하는 메서드들을 설정할 경우
  - bean : 설정 파일에 지정된 빈의 이름(name속성)을 이용해 pointcut 설정

POJO기반 AOP구현 – AspectJ 표현식(2/4)

- 표현

명시자(수식어패턴? 리턴타입패턴 패키지패턴? 클래스이름패턴.메소드이름패턴(파라미터패턴))

예) execution(public * abc.def..*Service.set*(..))

리턴 타입 표현식	설명
*	모든 반환형 허용
void	반환형이 void인 메소드 선택
!void	반환형이 void가 아닌 메소드 선택

패키지 경로 지정 표현식	설명
K2.sample	정확하게 K2.sample패키지만 선택
K2..	K2패키지로 시작하는 모든 패키지 선택
K2..impl	K2패키지로 시작하면서 마지막 패키지 이름이 impl로 끝나는 패키지만 선택

클래스이름지정 표현식	설명
SampleServiceImpl	정확하게 SampleServiceImpl클래스만 선택
*Impl	클래스 이름이 Impl로 끝나는 모든 클래스들을 선택
SampleService+	클래스 이름 뒤에 +가 붙으면 해당 클래스로부터 파생된 모든 자식 클래스 선택 인터페이스 뒤에 +기 붙으면 해당 인터페이스를 구현한 모든 클래스 선택

메소드 지정 표현식	설명
*(..)	가장 기본설정으로 모든 메소드 선택
select*(..)	메소드 이름이 select으로 시작하는 모든 메소드 선택

POJO기반 AOP구현 – AspectJ 표현식

매개변수 지정 표현식	설명
(..)	가장 기본 설정으로 맴개변수의 개수와 타입에 제약이 없음을 의미
(*)	반드시 1개의 매개변수를 가지는 메소드만 선택
(Sample)	매개변수로 Sample을 가지는 메소드만 선택, 이때 클래스의 패키지 경로가 반드시 포함되어야 함
(!Sample)	매개변수로 Sample을 가지지 않는 메소드만 선택
(Integer, ..)	한 개 이상의 매개변수를 가지되, 첫번째 매개변수의 타입이 Integer인 메소드만 선택
(Integer, *)	반드시 두 개의 매개변수를 가지되, 첫 번째 매개변수의 타입이 Integer인 메소드만 선택

POJO기반 AOP구현 – AspectJ 표현식(3/4)

- 수식어 패턴에는 public, protected 또는 생략한다.
  - * : 1개의 모든 값을 표현
    - argument에서 쓰인 경우 : 1개의 argument
    - package에 쓰인 경우 : 1개의 하위 package
  - .. : 0개 이상
    - argument에서 쓰인 경우 : 0개 이상의 argument
    - package에 쓰인 경우 : 0개의 이상의 하위 package
  - argument에 type을 명시할 경우 객체 타입은 fullyName으로 넣어야 한다.
  - 예 설명

```
예) execution(public * abc.def..*Service.set*(..))
```

    - 적용 하려는 메소드들의 패턴은 public 제한자를 가지며, 리턴 타입에는 모든 타입(*)이 다 올 수 있다. 이름은 abc.def 패키지와 그 하위 패키지에 있는 모든 클래스(..) 중 Service로 끝나는 클래스들에서 set으로 시작하는 메소드이며, argument는 0개 이상(..) 오며 타입은 상관 없다.

POJO기반 AOP구현 – AspectJ 표현식(4/4)

- execution(* test.spring.*.*())
- execution(public * test.spring..*.*())
- execution(public * test.*.*.get*(*))
- execution(String test.spring.MemberService.registMember(..))
- execution(* test.spring..*Service.regist*(..))
- execution(public * test.spring..*Service.regist*(String, ..))

//within(패키지명.클래스명) 형식,스프링빈으로 등록된 클래스의 모든 메소드를 타겟 메소드로 지정

- within(test.spring.service.MemberService)
- within(test.spring..MemberService)
- within(test.spring.aop..*)

- bean(memberService)
- bean(*Service)

POJO기반 AOP구현 – Advice 작성

- POJO 기반 Aspect클래스 작성
  - 설정 파일의 advice 관련 태그에 맞게 작성한다.
  - <bean>으로 등록 하며 <aop:aspect> 의 ref 속성으로 참조한다.

    <aop:aspect ref="advice">
  - 공통 기능 메소드 : advice 관련 태그들의 method 속성의 값이 메소드의 이름이 된다.

POJO기반 AOP구현 – Advice 정의관련태그

- 속성
  - pointcut-ref : <aop:pointcut>태그의 id명을 넣어 pointcut지정
  - pointcut : 직접 pointcut을 설정 한다.
  - method : Aspect bean에서 호출할 메소드명 지정

```
<bean id="writelog" class="org.kosta.spring.LogAspect"/>

<aop:config>
    <aop:aspect id="logingAspect" ref="writelog">
        <aop:pointcut id="publicmethod"
                      expression="execution(public * org.my.spring..*.*(..))"/>

        // Joinpoint 설정 : 핵심관심모듈의 어느 지점에 횡단관심모듈을 연결시킬지 설정
        <aop:before pointcut-ref="publicmethod" method="logging"/>
                                                  // 타겟 메소드 지정
    </aop:aspect>
</aop:config>
```

Aspect설정의 동작 순서

SampleAdvice.java

```
public class SampleAdvice{
        public void beforeLogic(){     ....
        }
```

beans.xml

```
<bean id="advice" class=k2.sample.service.SampleAdvice"></bean>

<aop:config>
     <aop:pointcut id="allPointcut" expression="execution(* k2.sample..*Impl.*(..))" />
          <aop:aspect ref="advice">
                                ②
               <aop:before pointcut-ref="allPointcut" method="beforeLogic" />
                     ④                    ①                    ③
          </aop:aspect>
</aop:config>
```

*동작 순서
①allPointcnt으로 설정한 비즈니스 메소드가 호출될 때
②advice라는 어드바이스 객체의
③beforLogic()메소드가 실행되고,
④이때 beforeLogic()메서드 동작 시점이 <aop:before>라는 내용임

POJO기반 AOP구현 – Aspect클래스 작성(1/7)

- POJO 기반의 클래스로 작성한다.
  - 클래스 명이나 메서드 명에 대한 제한은 없다.
  - 메소드 구문은 호출되는 시점에 따라 달라 질 수 있다.
  - 메소드의 이름은 advice 태그(<aop:before/>)에서 method 속성의 값이 메소드 명이 된다.
- before 메소드
  - 대상 객체의 메소드가 실행되기 전에 실행됨
  - return type : 상관없으나 void로 한다.
  - argument : 없거나 JoinPoint 객체를 받는다.
  - ex) public void beforeLogging(JoinPoint jp){ }

POJO기반 AOP구현 – Aspect클래스 작성(2/7)

- After Returning Advice
  - 대상 객체의 메소드 실행이 정상적으로 끝난 뒤 실행됨
  - return type : 상관없으나 void로 한다.
  - argument :
    - 없거나 JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용된다.
    - 대상 메소드에서 리턴 되는 값을 argument로 받을 수 있다.
      type : Object 또는 대상 메소드에서 return하는 value의 type

POJO기반 AOP구현 – Aspect클래스 작성(3/7)

```
<aop:after-returning pointcut-ref="publicmethod"
        method="returnLogging" returning="retValue"/>


public void returnLogging(Object retValue){

}
```

POJO기반 AOP구현 – Aspect클래스 작성(4/7)

- After Throwing Advice
  - 대상 객체의 메소드 실행 중 예외가 발생한 경우 실행됨
  - return type : 상관없으나 대개 void로 한다.
  - argument :
    - 없거나 JoinPoint 객체를 받는다. JoinPoint는 항상 첫 argument로 사용된다.
    - 대상메소드에서 전달되는 예외객체를 argument로 받을 수 있다.

POJO기반 AOP구현 – Aspect클래스 작성(5/7)

```
<aop:after-throwing pointcut-ref="publicmethod"
        method="returnLogging"  throwing="ex"/>
```

```
public void returnLogging(MyException ex){
    // 대상객체에서 리턴 되는 값을 받을 수는 있지만
    // 수정할 수는 없다.
}
```

POJO기반 AOP구현 – Aspect클래스 작성(6/7)

- Around Advice
  - 위의 네 가지 Advice를 다 구현 할 수 있는 Advice
  - return type : Object
  - argument
    - org.aspectj.lang.ProceedingJoinPoint 를 반드시 첫 argument로 지정한다.

POJO기반 AOP구현 – Aspect클래스 작성(7/7)

```
<aop:around pointcut-ref="publicmethod" method="returnLogging" />

public Object returnLogging(ProceedingJoinPoint joinPoint) throws Throwable{
   //대상 객체의 메소드 호출 전 해야 할 전 처리 코드
   try{
      Object retValue = joinPoint.proceed(); //대상객체의 메소드 호출
   //대상 객체 처리 이후 해야 할 후처리 코드
      return retValue; //호출 한 곳으로 리턴 값 넘긴다. – 넘기기 전 수정 가능
  }catch(Throwable e){
     throw e; //예외 처리
  }
}
```

POJO기반 AOP구현 – JoinPoint

- 대상객체에 대한 정보를 가지고 있는 객체로 Spring container로 부터 받는다.
- org.aspectj.lang 패키지에 있음
- 반드시 Aspect 메소드의 첫 argument로 와야 한다.
- 메소드들

Object getTarget() : 대상객체를 리턴(클라이언트가 호출한 비즈니스 메소드를 포함하는 객체 반환)
Object[] getArgs() : 파라미터로 넘겨진 값들을 배열로 리턴, 넘어온 값이 없으면 빈 배열개체가 return 됨
Signature getSignature () : 클라이언트가 호출한 메소드의 시그니처(반환형, 이름,매개변수)정보가 저장된 객체 반환

- Signature : 호출 되는 메소드에 대한 정보를 가진 객체

String getName() : 대상 메소드명 리턴
String toShortString() : 대상 메소드명 리턴
String toLongString() : Package 를 포함한 full name(반환형, 이름, 매개변수) 리턴
String getDeclaringTypeName() : 대상 메소드가 포함된 type을 return (package명.type명)

AOP 구현 방법1

```
public class Singer implements Performer {
        private String name = “Someone”;
        private Song song;

        public Singer() {}
        public Singer(Song song) {
                this.song = song;
        }
        public Singer(Stirng name, Song song) {
                this.name = name;
                this.song = song;
        }

        public void perform() {
                System.out.prtinln(name + “IS SINGING” + song.getTitle());
        }
}
```

```
public class Song {
        private String title;
        public Song(String title) {
                this.title = title;
        }
        public String getTitle() {
                return title;
        }
}
```

AOP구현 방법(2)

```
public class Monitor {
        public void greetPerformer() {
                System.out.prtinln(“Greeting Performer”);
        }
        public void saySomethingNice() {
                System.out.prtinln(“That was Great”);
        }
        public void saySomethingNiceAnyway() {
                System.out.prtinln(“It WASN’t GREAT”);
        }
}
```

```
public class AopApp {
        public static void main(String[] args) throws Exception {
                ApplicationContext context = new ClassPathXmlApplcationContext("kr/example/aop/aop.xml“);
                Performer singer = (Performer ) factory.getBean(“bo");
                singer.perform());
        }
}
```

AOP구현 방법(3)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans….중간생략….

        <bean id=“song" class="kr.example.aop.song">
                <constructor-arg value=“Sweet Home Alabama"/>
        </bean>
        <bean id=“bo" class="kr.example.aop.Singer">
                <constructor-arg value=“Bo Bice"/>
                <constructor-arg ref=“song”/>
        </bean>
        <bean id=“monitor" class="kr.example.aop.Monitor">

        <aop:config>
                <aop:aspect ref =“monitor”>
                        <aop:before pointcut =“execution(* *.perform(..))” method =“greetingPerformer”/>
                        <aop:after-returning pointcut =“execution(* *.perform(..))” method =“saySomethingNice”/>
                        <aop:after-throwing pointcut =“execution(* *.perform(..))” method =“saySomethingNiceAnyway”/>
                </aop:aspect>
        </aop:config>
</beans>
```

AOP구현 방법(4)

```
@Aspect
public class Monitor {
        @Before(value = "execution(* *.perform(..))")
        public void greetPerformer() {
                System.out.println("GREETING PERFORMER");
        }
        @AfterReturning(value = "execution(* *.perform(..))")
        public void saySomethingNice() {
                System.out.println("THAT WAS GREAT");
        }
        @AfterThrowing(value = "execution(* *.perform(..))")
        public void saySomethingNiceAnyway() {
                System.out.println("IT WASN'T GREAT, BUT I LOVE YOU ANYWAY");
        }
        @Around(value = "execution(* *.perform(..))")
        public void around(ProceedingJoinPoint  jp) throws Throwable {
                System.out.println("around...before");
                jp.proceed();
                System.out.println("around...after");
        }
}
```

```
<bean id="monitor" class="kr.example.aop.Monitor">

<aop:config>
        <aop:aspect ref ="monitor">
                <aop:before pointcut ="execution(* *.perform(..))" method ="greetingPerformer"/>
                <aop:after-returning pointcut ="execution(* *.perform(..))" method ="saySomethingNice"/>
                <aop:after-throwing pointcut ="execution(* *.perform(..))" method ="saySomethingNiceAnyway"/>
        </aop:aspect>
</aop:config>
```