

6.5 스프링 AOP

여재환

6.5.1 자동 프록시 생성

6.5.1 자동 프록시 생성

- 6.4 까지 투명한 부가기능을 적용하는 과정에서 발견된 문제는 거의 제거
- 해결해야 할 과제
 - 프록시 팩토리 빈 방식의 접근방법의 한계
 - 부가기능이 타깃 오브젝트마다 새로 만들어지는 문제
→ 스프링 `ProxyFactoryBean`의 어드바이스를 통해 해결
 - 부가기능의 적용이 필요한 타깃 오브젝트마다 거의 비슷한 내용의 `ProxyFactoryBean` 설정정보를 추가해야하는 부분

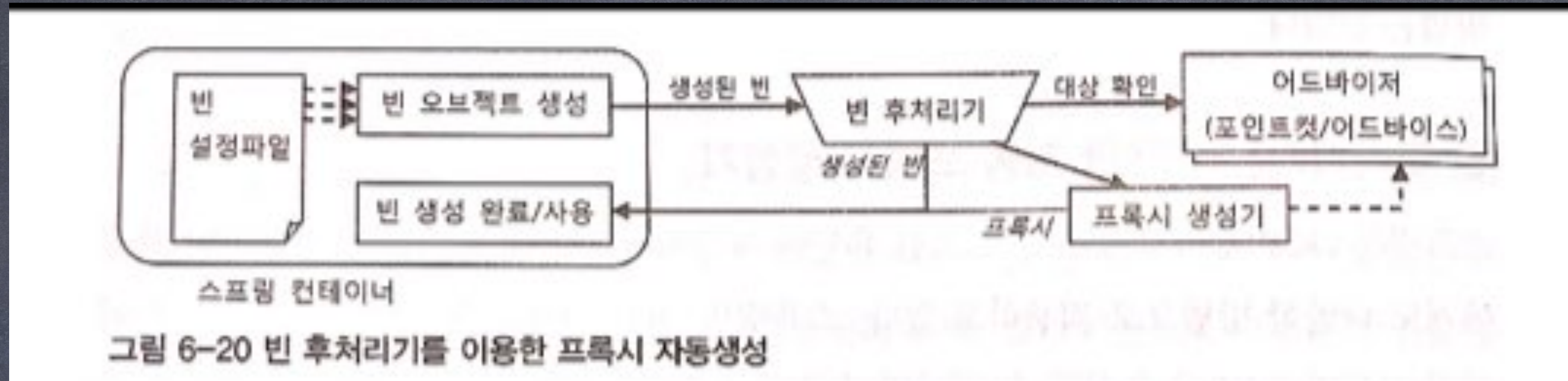
6.5.1 자동 프록시 생성

- 이때까지 다룬 반복적이고 기계적인 코드의 해결책
 - JDBC → template & callback, Client 분리
Strategy Pattern, DI 적용
 - 반복적 위임코드가 필요한 프록시 클래스 코드
 - 다이내믹 프록시라는 런타임 자동생성기법
 - 개발자가 일일이 인터페이스 메소드 구현하지 않아도 됨
- 반복적인 프록시 메소드 구현을 코드 자동생성 기법을 이용해 해결
- 그렇다면 ProxyFactoryBean 설정문제도 동일한 방식으로 해결할수 있을까?

6.5.1 자동 프록시 생성

- 빈 후처리를 이용한 자동 프록시 생성기
 - 스프링은 OCP의 유연한 확장 개념을 스프링 컨테이너 자신에게도 다양한 방법으로 적용
 - 컨테이너로서 제공하는 기능 중에서 변하지 않는 핵심적인 부분외에는 대부분 확장할 수 있도록 확장포인트 제공
- BeanPostProcessor를 구현해서 만드는 빈 후처리기
- DefaultAdvisorAutoProxyCreator
 - 어드바이저를 이용한 자동 프록시 생성기
 - 빈 오브젝트 프로퍼티 강제 수정, 초기화, 빈 오브젝트 바꿔치기 등 가능

6.5.1 자동 프록시 생성



- 프록시를 적용할 빈을 선정하는 로직이 추가된 포인트컷이 담긴 어드바이저가 있다면?
- `ProxyFactoryBean`을 등록 없이 타깃 오브젝트에 자동으로 프록시 적용 가능
- `ProxyFactoryBean`의 번거로운 설정 문제 해결

6.5.1 자동 프록시 생성

```
public interface Pointcut {  
    ClassFilter getClassFilter();  
    MethodMatcher getMethodMatcher();  
}
```

- 포인트컷
 - 클래스 필터, 메소드 매처를 반환하는 메소드 모두를 가지고 있음
 - 프록시를 적용할 클래스인지 판단 이후 어드바이스를 적용할 메소드인지 확인

6.5.2

DefaultAdvisorAutoProxyCreator
의 적용

6.5.2 DefaultAdvisorAutoProxyCreator의 적용

클래스 필터 기능이 추가된 포인트컷

```
public class NameMatchClassMethodPointcut extends NameMatchPointcut {
    public void setMappedClassName(String mappedClassName) {
        this.setClassFilter(new SimpleClassFilter(mappedClassName));
        // 모든 클래스를 다 허용하던 디폴트 클래스 필터를 프로퍼티로 받은 클래스 이름을 이용하여 필터를 만들도록 덮어씌운다.
    }

    static class SimpleClassFilter implements ClassFilter {
        String mappedName;

        private SimpleClassFilter(String mappedName) {
            this.mapppedName = mappedName;
        }

        public boolean matches(Class<?> clazz) {
            return PatterMatchUtils.simpleMatch(mappedName, clazz.getSimleName());
            // PatterMatchUtils.simpleMatch : 와일드카드(*) 가 들어간 문자열 비교를 지원하는 스프링 유틸리티 메소드
        }
    }
}
```


6.5.2 DefaultAdvisorAutoProxyCreator의 적용

- 어드바이저를 이용하는 자동 프록시 생성기 등록
- DefaultAdvisorAutoProxyCreator은 등록된 빈에서 Advisor 인터페이스를 implements한 것을 모두 찾음
- 이후 생성되는 모든 빈에 대해 advisor 의 포인트컷을 적용하며 프록시 대상 탐색,
- 대상이면 ? 프록시 오브젝트를 원래 오브젝트와 바꿔치기
- DefaultAdvisorAutoProxyCreator 등록 방법

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />
```


6.5.2 DefaultAdvisorAutoProxyCreator의 적용

- 포인트컷 등록

```
<bean id="transactionPointcut"
      class="springbook.service.NameMatchClassMethodPointcut" >
  <property name="mappedClassName" value="*ServiceImpl" />
  <property name="mappedName" value="upgrade*" />
</bean>
```

- 클래스 필터 지원 포인트컷

- ServiceImpl로 끝나는 클래스와 upgrade 로 시작하는 메소드 선정

6.5.2 DefaultAdvisorAutoProxyCreator의 적용

- 어드바이스와 어드바이저
 - 어드바이저를 이용하는 자동 프록시 생성기
DefaultAdvisorAutoProxyCreator 에 의해 자동수집
→ ProxyFactoryBean 으로 등록한 빈에서처럼 transactionAdvisor
를 명시적으로 DI 할 필요 없다.
- ProxyFactoryBean 제거와 서비스 빈의 원상복구
 - 프록시 도입으로 인해 아이디를 바꾸고 프록시에 DI 됐던
userServiceImpl의 빈 아이디 복구

```
<bean id="userService" class="springbook.service.UserServiceImpl" >  
    <property name="userDao" ref="userDao" />  
    <property name="mailSender" ref="mailSender" />  
</bean>
```


6.5.3

포인트컷 표현식을 이용한 포인트컷

6.5.3 포인트컷 표현식을 이용한 포인트컷

- 편리한 포인트컷 작성방법
- 단순한 클래스, 메서드 이름을 비교하는 일이 아닌 복잡하고 세밀한 기준으로 클래스, 메소드를 선정하기 위한 방법
- 필터, 매처에서 메타 정보 제공받음
 - 리플렉션 API를 통해 클래스·메소드의 이름, 패키지, 파라미터, 리턴값, 애노테이션, 인터페이스, 부모 클래스 까지 활용 가능
- But
 - 리플렉션 API는 작성하기 번거롭다.
 - 조건이 달라질 때마다 포인트컷 구현 코드가 수정돼야 함.

6.5.3 포인트컷 표현식을 이용한 포인트컷

- 해결책 : 스프링이 제공하는 포인트컷 표현식(Pointcut Expression)
- AspectJExpressionPointcut
 - AspectJ 의 일부 문법을 확장
 - 클래스와 메소드 선정 알고리즘을 포인트컷 표현식을 이용해 한번에 정의

6.5.3 포인트컷 표현식을 이용한 포인트컷

- ◉ `execution([접근 제한자 패턴] 타입 패턴 [타입 패턴.]이름패턴 (타입패턴 | "...", ...) [throws 예외패턴])`

[접근 제한자 패턴]	타입 패턴	[타입 패턴.]	이름패턴	(타입패턴 "...", ...)	[throws 예외패턴]
<code>public, private</code> 과 같은 접근 제한자, 생략 가능	리턴 값의 타입 패턴	패키지와 클래스 이름에 대한 패턴, 생략 가능, '.' 으로 연결	메소드 이름 패턴	파라미터의 타입 패턴, 와일드 카드 이용 가능	예외 이름 패턴

◉ Example

- ◉ `execution(int minus(int, int))`
- ◉ `int` 타입을 리턴, 메소드 이름은 `minus`, 파라미터는 두개의 `int`.
- ◉ 리턴 값의 타입에 대한 제한 무시 : `execution(* minus(int, int))`
- ◉ 파라미터의 갯수와 타입 무시 : `execution(* minus(..))`

6.5.3 포인트컷 표현식을 이용한 포인트컷

- ◉ AspectJ의 `execution()` 외의 표현식 스타일
 - ◉ `bean()`
 - ◉ 빈의 이름을 비교
 - ◉ `bean(*Service)` - 아이디가 `Service` 로 끝나는 모든 빈
 - ◉ `@annotation()`
 - ◉ 특정 애노테이션이 타입, 메소드, 파라미터에 적용되어 있는 것을 보고 메소드 선정
 - ◉ `@annotation(org.springframework.transaction.annotation.Transactional)`
 - ◉ `@Transactional` 애노테이션이 적용된 메소드 선정

6.5.3 포인트컷 표현식을 이용한 포인트컷

● 포인트컷 표현식 적용

```
<bean id="userService" class="springbook.service.UserServiceImpl" >  
    <property name="userDao" ref="userDao" />  
    <property name="mailSender" ref="mailSender" />  
</bean>
```




```
<bean id="transactionPointcut"  
    class="org.springframework.aop.aspectj.AspectJExpressionPointcut" >  
    <property name="expression" value="execution(* *..*ServiceImpl.upgrade*(..))" />  
</bean>
```


6.5.4 AOP 란 무엇인가?

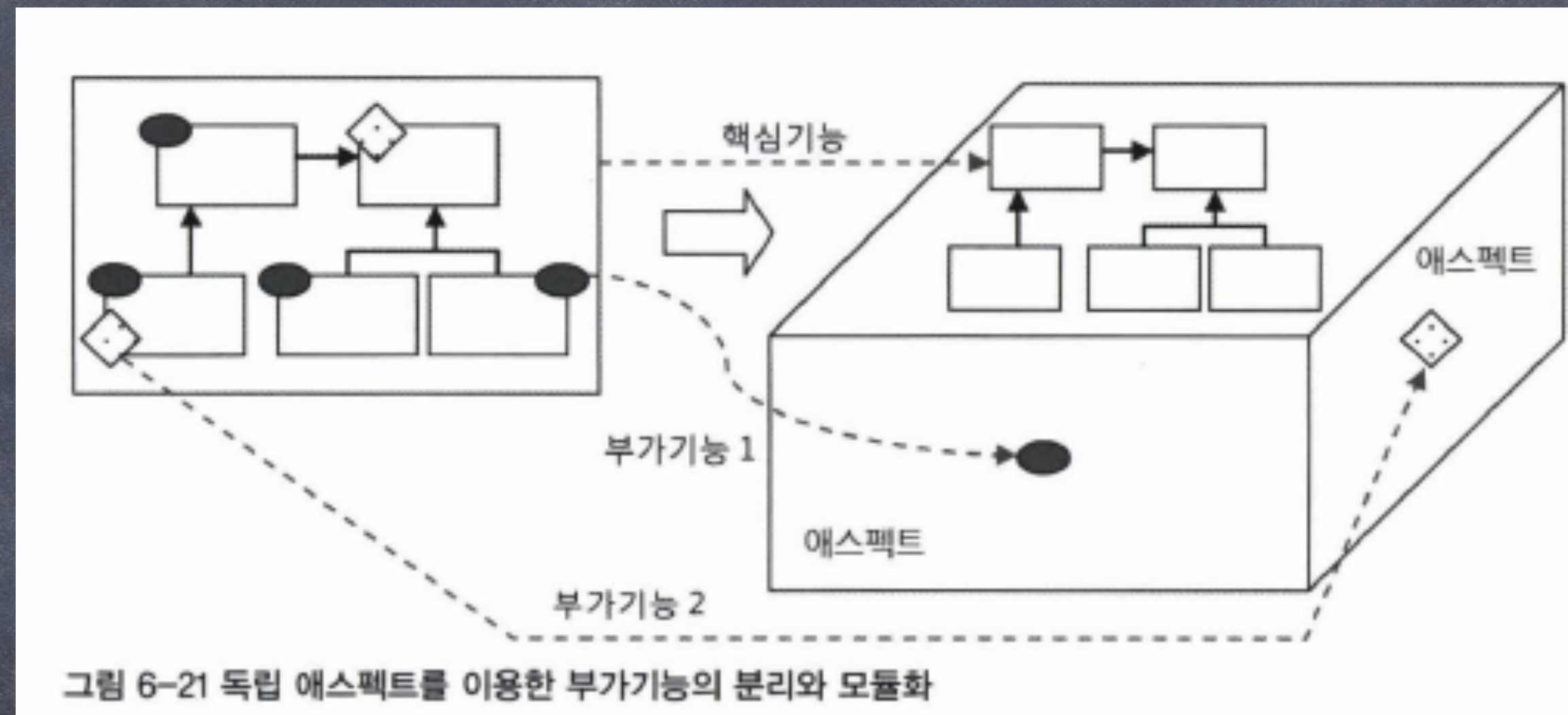
6.5.4 AOP 란 무엇인가?

aspect *noun*

★  B2 [countable] a particular part or feature of a situation, an idea, a problem, etc.; a way in which it may be considered

- AOP : Aspect Oriented Programming
 - Aspect : 전통적인 객체지향 기술의 설계방법으로는 독립적인 모듈화가 불가능한 트랜잭션 경계설정과 같은 부가기능을 담당하는 모듈
 - aspect 란 그 자체로 애플리케이션의 핵심기능을 담고 있지는 않지만, 애플리케이션을 구성하는 중요한 요소이고, 핵심 기능에 부가되어 의미를 갖는 특별한 모듈
 - aspect = advice + pointcut
(부가될 기능을 정의한 코드) (어드바이스를 어디 적용할 지 결정)

6.5.4 AOP란 무엇인가?



● 왼쪽 그림

- *aspect* 부가기능 분리 이전
- 핵심기능은 모듈화 되었지만 부가기능으로 인해 핵심기능에만 관심사를 두기 힘들어짐
- 테스트의 어려움

● 오른쪽 그림

- 핵심기능 코드 사이에 침투 가능한 *aspect* 분리
- 2차원구조의 한계를 3차원 구조로 구성하며 성격이 다른 부가기능은 다른 면에 존재하도록 구성

6.5.4 AOP 란 무엇인가?

- AOP는 OOP를 위한 보조적인 기술
 - 객체지향의 가치를 지키기 위한 기술이 AOP

6.5.5 AOP 적용 기술

6.5.5 AOP 적용 기술

- 프록시를 이용한 AOP

- 스프링은 DI로 연결된 빈 사이에서 프록시를 통해 타깃 메소드 호출 과정에서 부가기능 제공함으로써 AOP를 지원
- 따라서 스프링 컨테이너와 자바 기본 JDK 외에는 특별한 기술이나 환경을 요구하지 않음
- 독립적으로 개발한 부가기능 모듈을 타깃 오브젝트의 메소드에 다이내믹하게 적용해주기 위한 역할이 프록시

→ 스프링 AOP는 프록시 방식의 AOP

6.5.5 AOP 적용 기술

- 바이트코드 생성과 조작을 통한 AOP
 - AOP 기술의 원조이자 강력한 AOP 프레임워크로 꼽히는 AspectJ
 - AspectJ는 프록시처럼 간접적인 기술이 아닌 타깃 오브젝트를 직접 뜯어 고쳐 부가기능을 넣는 방식 사용
 - 컴파일된 타깃의 클래스 파일을 수정하거나 클래스가 JVM에 로딩되는 시점을 가로채서 바이트 코드를 조작
 - 바이트 코드 조작의 장점
 - DI, 자동프록시 생성방식이 없어도 AOP 적용 가능
 - 오브젝트 생성, 필드값 조회 및 조작, 스테틱 초기화 등의 강력하고 유연한 AOP

6.5.7 AOP 네임스페이스

6.5.7 AOP 네임스페이스

- AOP 적용을 위해 추가했던 어드바이저, 포인트컷, 자동프록시 생성기 같은 빈들은 `UserDao`, `UserService` 같은 빈과는 성격이 다름
- 비즈니스 로직, 애플리케이션의 기능의 일부도 아니고 `dataSource`처럼 애플리케이션 빈에서 사용되는 것도 아니다.
- 스프링 컨테이너에 의해 특별한 작업을 위해 사용
- 프록시 방식의 AOP 를 적용하기 위한 4가지 빈
 - 자동프록시 생성기, 어드바이스, 포인트컷, 어드바이저

6.5.7 AOP 네임스페이스

- AOP 네임스페이스

- 스프링은 AOP를 위해 기계적으로 사용하는 빈들을 간편한 방식으로 등록할 수 있도록 지원
- aop 스키마를 이용하면 디폴트 네임 스페이스 `<bean>` 태그와 구분해서 사용
- aop 스키마에 정의된 태그를 사용하는 방법

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframewokr.org/schema/aop
                           http://www.springframewokr.org/schema/aop/spring-aop-3.0.xsd">

    ...

</beans>
```


6.5.7 AOP 네임스페이스

- aop 네임스페이스를 이용한 AOP 관련 빈 설정

```
<aop:config>
  <aop:pointcut id="transactionPointcut"
    expression="execution(* *..*ServiceImpl.upgrade*(..))" />
  <aop:advisor advice-ref="transactionAdvice" pointcut-ref="transactionPointcut" />
</aop:config>
```


6.5.7 AOP 네임스페이스

- 어드바이저 내장 포인트컷
 - AspectJ 포인트컷 표현식을 활용하는 포인트컷은 스트링으로 된 표현식을 담은 *expression* 프로퍼티 하나만 설정해주면 된다.
 - 포인트컷은 어드바이저에 참조되야므로 포인트컷을 어드바이저 태그와 결합하는 방법도 가능

```
<aop:config>
  <aop:advisor advice-ref="transactionAdvice"
               pointcut="execution(* *..*ServiceImpl.upgrade*(..))" />
</aop:config>
```