

## 8. @Aspect AOP

#1.인강/6.핵심 원리 - 고급편/강의#

- /@Aspect 프록시 - 적용
- /@Aspect 프록시 - 설명
- /정리

### @Aspect 프록시 - 적용

스프링 애플리케이션에 프록시를 적용하려면 포인트컷과 어드바이스로 구성되어 있는 어드바이저(Advisor)를 만들어서 스프링 빈으로 등록하면 된다. 그러면 나머지는 앞서 배운 자동 프록시 생성기가 모두 자동으로 처리해준다. 자동 프록시 생성기는 스프링 빈으로 등록된 어드바이저들을 찾고, 스프링 빈들에 자동으로 프록시를 적용해준다. (물론 포인트컷이 매칭되는 경우에 프록시를 생성한다.)

스프링은 @Aspect 애노테이션으로 매우 편리하게 포인트컷과 어드바이스로 구성되어 있는 어드바이저 생성 기능을 지원한다.

지금까지 어드바이저를 직접 만들었던 부분을 @Aspect 애노테이션을 사용해서 만들어보자.

**참고:** @Aspect 는 관점 지향 프로그래밍(AOP)을 가능하게 하는 AspectJ 프로젝트에서 제공하는 애노테이션이다. 스프링은 이것을 차용해서 프록시를 통한 AOP를 가능하게 한다. AOP와 AspectJ 관련된 자세한 내용은 다음에 설명한다. 지금은 프록시에 초점을 맞추자. 우선 이 애노테이션을 사용해서 스프링이 편리하게 프록시를 만들어준다고 생각하면 된다.

#### LogTraceAspect

```
package hello.proxy.config.v6_aop.aspect;

import hello.proxy.trace.TraceStatus;
import hello.proxy.trace.logtrace.LogTrace;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Slf4j
@Aspect
public class LogTraceAspect {
```

```

private final LogTrace logTrace;

public LogTraceAspect(LogTrace logTrace) {
    this.logTrace = logTrace;
}

@Around("execution(* hello.proxy.app..*(..))")
public Object execute(ProceedingJoinPoint joinPoint) throws Throwable {

    TraceStatus status = null;

    //      log.info("target={}", joinPoint.getTarget()); //실제 호출 대상
    //      log.info("getArgs={}", joinPoint.getArgs()); //전달인자
    //      log.info("getSignature={}", joinPoint.getSignature()); //join point 시그
    니처

    try {
        String message = joinPoint.getSignature().toShortString();
        status = logTrace.begin(message);

        //로직 호출
        Object result = joinPoint.proceed();

        logTrace.end(status);
        return result;
    } catch (Exception e) {
        logTrace.exception(status, e);
        throw e;
    }
}
}

```

- `@Aspect`: 애노테이션 기반 프록시를 적용할 때 필요하다.
- `@Around("execution(* hello.proxy.app..*(..))")`
  - `@Around`의 값에 포인트컷 표현식을 넣는다. 표현식은 AspectJ 표현식을 사용한다.
  - `@Around`의 메서드는 어드바이스(Advice)가 된다.
- `ProceedingJoinPoint joinPoint`: 어드바이스에서 살펴본 `MethodInvocation invocation`과 유사한 기능이다. 내부에 실제 호출 대상, 전달 인자, 그리고 어떤 객체와 어떤 메서드가 호출되었는지 정보가 포함되어 있다.
- `joinPoint.proceed()`: 실제 호출 대상(target)을 호출한다.

## AopConfig

```
package hello.proxy.config.v6_aop;

import hello.proxy.config.AppV1Config;
import hello.proxy.config.AppV2Config;
import hello.proxy.config.v6_aop.aspect.LogTraceAspect;
import hello.proxy.trace.logtrace.LogTrace;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import({AppV1Config.class, AppV2Config.class})
public class AopConfig {

    @Bean
    public LogTraceAspect logTraceAspect(LogTrace logTrace) {
        return new LogTraceAspect(logTrace);
    }
}
```

- `@Import({AppV1Config.class, AppV2Config.class})`: V1, V2 애플리케이션은 수동으로 스프링 빈으로 등록해야 동작한다.
- `@Bean logTraceAspect()`: `@Aspect`가 있어도 스프링 빈으로 등록을 해줘야 한다. 물론 `LogTraceAspect`에 `@Component` 애노테이션을 붙여서 컴포넌트 스캔을 사용해서 스프링 빈으로 등록해도 된다.

## ProxyApplication

```
//@Import({AppV1Config.class, AppV2Config.class})
//@Import(InterfaceProxyConfig.class)
//@Import(ConcreteProxyConfig.class)
//@Import(DynamicProxyBasicConfig.class)
//@Import(DynamicProxyFilterConfig.class)
//@Import(ProxyFactoryConfigV1.class)
//@Import(ProxyFactoryConfigV2.class)
//@Import(BeanPostProcessorConfig.class)
//@Import(AutoProxyConfig.class)
//@Import(AutoProxyMultiAdvisorConfig.class)
@Import(AopConfig.class)
```

```

@SpringBootApplication(scanBasePackages = "hello.proxy.app")
public class ProxyApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProxyApplication.class, args);
    }

    @Bean
    public LogTrace logTrace() {
        return new ThreadLocalLogTrace();
    }
}

```

- AopConfig.class 를 등록하자.

## 실행

- <http://localhost:8080/v1/request?itemId=hello>
- <http://localhost:8080/v2/request?itemId=hello>
- <http://localhost:8080/v3/request?itemId=hello>

실행해보면 모두 프록시가 잘 적용된 것을 확인할 수 있다.

## @Aspect 프록시 - 설명

앞서 자동 프록시 생성기를 학습할 때, 자동 프록시 생성기(AnnotationAwareAspectJAutoProxyCreator)는 Advisor 를 자동으로 찾아와서 필요한 곳에 프록시를 생성하고 적용해준다고 했다. 자동 프록시 생성기는 여기에 추가로 하나의 역할을 더 하는데, 바로 @Aspect 를 찾아서 이것을 Advisor 로 만들어준다. 쉽게 이야기해서 지금까지 학습한 기능에 더해 @Aspect 를 Advisor 로 변환해서 저장하는 기능도 한다. 그래서 이름 앞에 AnnotationAware (애노테이션을 인식하는)가 붙어 있는 것이다.

## 어드바이저 = 포인트컷 + 어드바이스

```
@Aspect
public class LogTraceAspect {

    @Around("execution(* hello.proxy.app..*(..))")
    public Object execute(ProceedingJoinPoint joinPoint) {
        //Advice 로직
    }
}
```

포인트컷 - Pointcut

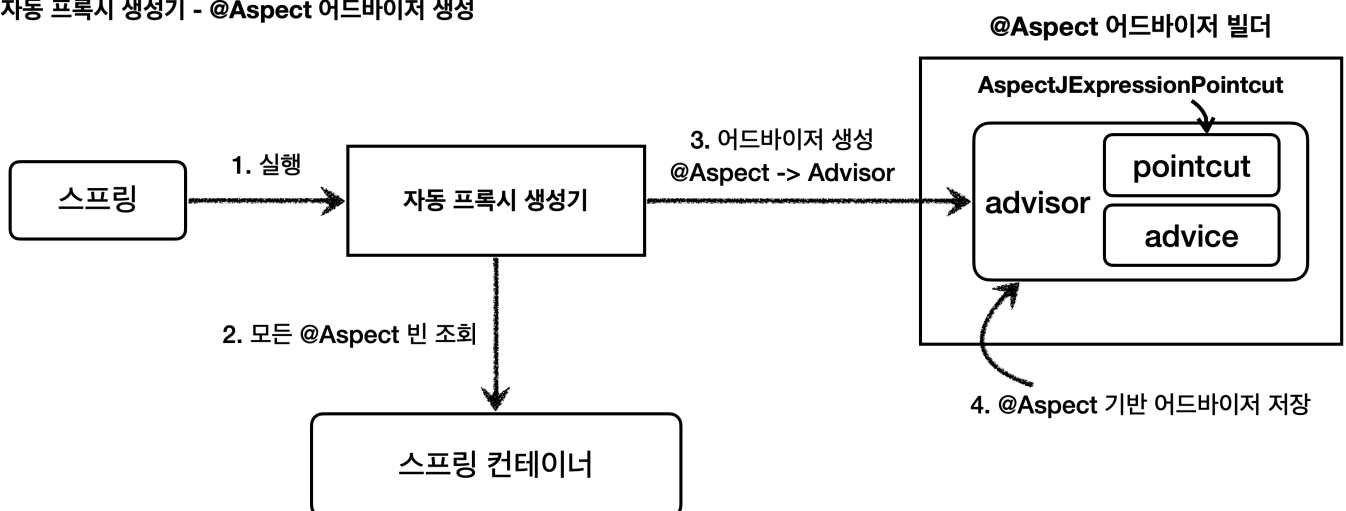
어드바이스 - Advice

자동 프록시 생성기는 2가지 일을 한다.

- 1. `@Aspect` 를 보고 어드바이저(Advisor)로 변환해서 저장한다.
- 2. 어드바이저를 기반으로 프록시를 생성한다.

### 1. `@Aspect` 를 어드바이저로 변환해서 저장하는 과정

자동 프록시 생성기 - `@Aspect` 어드바이저 생성



### `@Aspect`를 어드바이저로 변환해서 저장하는 과정을 알아보자

- 1. **실행**: 스프링 애플리케이션 로딩 시점에 자동 프록시 생성기를 호출한다.
- 2. **모든 `@Aspect` 빈 조회**: 자동 프록시 생성기는 스프링 컨테이너에서 `@Aspect` 애노테이션이 붙은 스프링 빈을 모두 조회한다.
- 3. **어드바이저 생성**: `@Aspect` 어드바이저 빌더를 통해 `@Aspect` 애노테이션 정보를 기반으로 어드바이저를 생성한다.

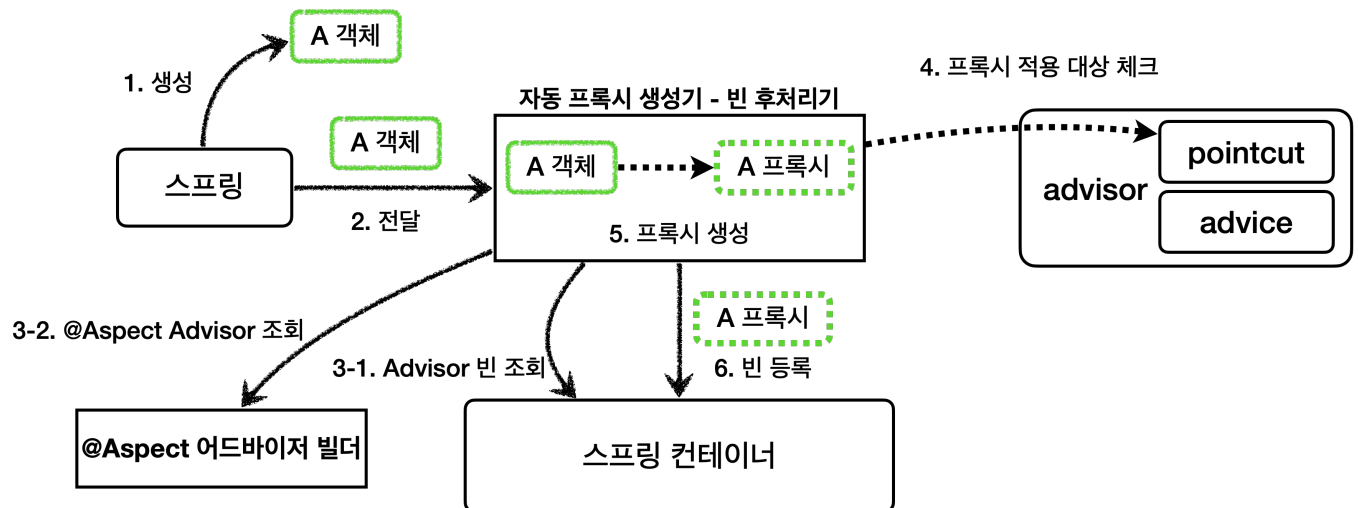
- 4. **@Aspect 기반 어드바이저 저장**: 생성한 어드바이저를 **@Aspect** 어드바이저 빌더 내부에 저장한다.

## @Aspect 어드바이저 빌더

**BeanFactoryAspectJAdvisorsBuilder** 클래스이다. **@Aspect** 의 정보를 기반으로 포인트컷, 어드바이스, 어드바이저를 생성하고 보관하는 것을 담당한다. **@Aspect** 의 정보를 기반으로 어드바이저를 만들고, **@Aspect** 어드바이저 빌더 내부 저장소에 캐시한다. 캐시에 어드바이저가 이미 만들어져 있는 경우 캐시에 저장된 어드바이저를 반환한다.

## 2. 어드바이저를 기반으로 프록시 생성

자동 프록시 생성기 - **@Aspect** 어드바이저 적용



### 자동 프록시 생성기의 작동 과정을 알아보자

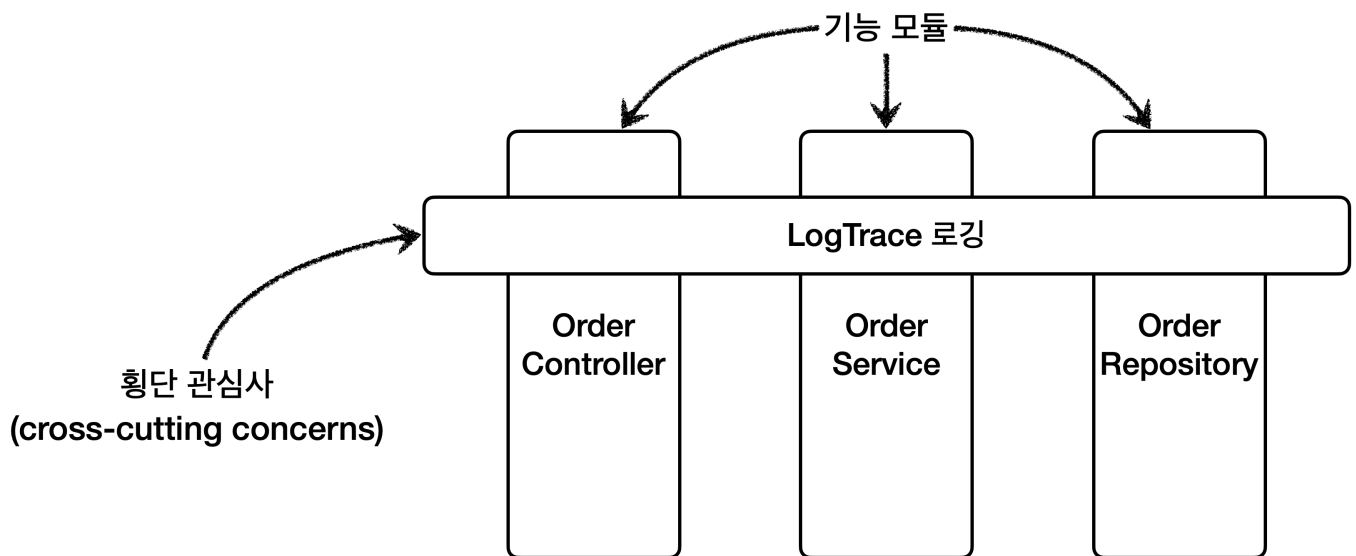
- 1. **생성**: 스프링 빈 대상이 되는 객체를 생성한다. (**@Bean**, 컴포넌트 스캔 모두 포함)
- 2. **전달**: 생성된 객체를 빈 저장소에 등록하기 직전에 빈 후처리기에 전달한다.
- 3-1. **Advisor 빈 조회**: 스프링 컨테이너에서 **Advisor** 빈을 모두 조회한다.
- 3-2. **@Aspect Advisor 조회**: **@Aspect** 어드바이저 빌더 내부에 저장된 **Advisor** 를 모두 조회한다.
- 4. **프록시 적용 대상 체크**: 앞서 3-1, 3-2에서 조회한 **Advisor** 에 포함되어 있는 포인트컷을 사용해서 해당 객체가 프록시를 적용할 대상인지 아닌지 판단한다. 이때 객체의 클래스 정보는 물론이고, 해당 객체의 모든 메서드를 포인트컷에 하나하나 모두 매칭해본다. 그래서 조건이 하나라도 만족하면 프록시 적용 대상이 된다. 예를 들어서 메서드 하나만 포인트컷 조건에 만족해도 프록시 적용 대상이 된다.
- 5. **프록시 생성**: 프록시 적용 대상이면 프록시를 생성하고 프록시를 반환한다. 그래서 프록시를 스프링 빈으로 등록한다. 만약 프록시 적용 대상이 아니라면 원본 객체를 반환해서 원본 객체를 스프링 빈으로 등록한다.
- 6. **빈 등록**: 반환된 객체는 스프링 빈으로 등록된다.

## 정리

@Aspect 를 사용해서 애노테이션 기반 프록시를 매우 편리하게 적용해보았다. 실무에서 프록시를 적용할 때는 대부분 이 방식을 사용한다.

지금까지 우리가 진행한 애플리케이션 전반에 로그를 남기는 기능은 특정 기능 하나에 관심이 있는 기능이 아니다. 애플리케이션의 여러 기능들 사이에 걸쳐서 들어가는 관심사이다.

이것을 바로 **횡단 관심사(cross-cutting concerns)**라고 한다. 우리가 지금까지 진행한 방법이 이렇게 여러곳에 걸쳐 있는 횡단 관심사의 문제를 해결하는 방법이었다.



지금까지 프록시를 사용해서 이러한 횡단 관심사를 어떻게 해결하는지 점진적으로 매우 깊이있게 학습하고 기반을 다져 두었다.

이제 이 기반을 바탕으로 이러한 횡단 관심사를 전문으로 해결하는 스프링 AOP에 대해 본격적으로 알아보자.