

AOP

목차

1. AOP란?

- AOP 개념 설명
- 예시

2. AOP 관련 주요 개념

- 개념 정리
- 구현 방법
- AspectJ vs Spring AOP
- AOP 작동 과정

3. 구현 예제

- Advice 별 호출 위치 확인
- JoinPoint로 파라미터 정보 받아오기
- 커스텀 어노테이션을 통한 AOP 구현

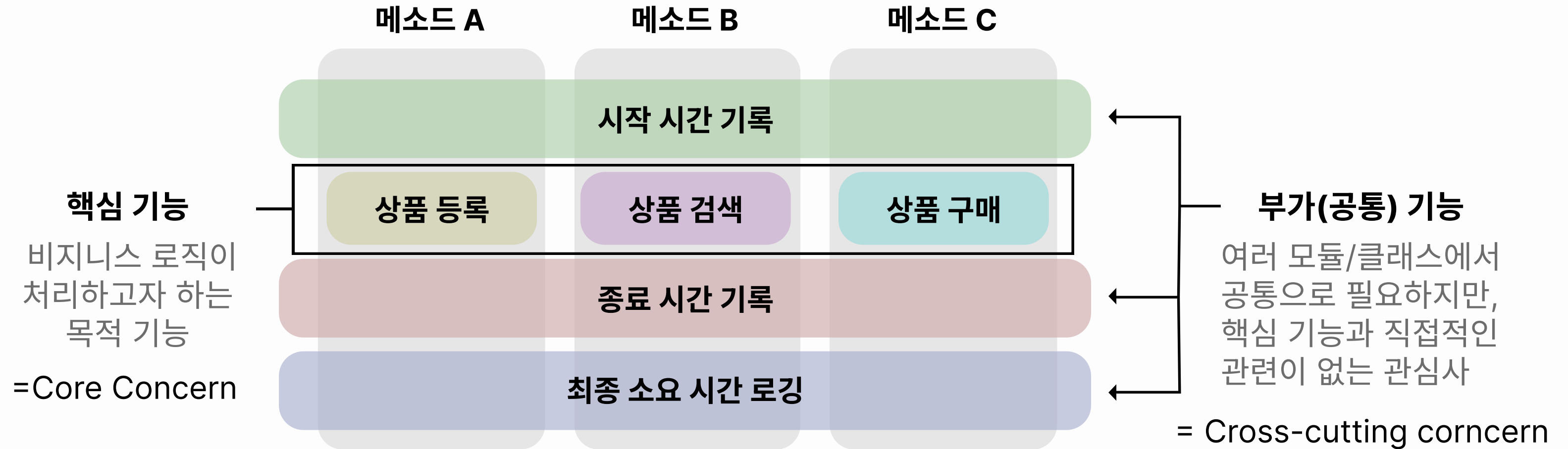
AOP (Aspect-Oriented Programming, 관점 지향 프로그래밍)

- 관점을 기준으로 다양한 기능들을 분리하여 보는 프로그래밍
- 횡단 관심사의 분리를 허용함으로써, 모듈성을 증가시키는 것이 목적인 프로그래밍 패러다임
- 모듈화된 객체를 편하게 적용할 수 있게 함으로써 개발자가 비즈니스 로직을 구현하는 데만 집중할 수 있게 도와줌

+) 활용 예시

- 로깅, 트랜잭션, 보안, 캐싱, 예외처리 등

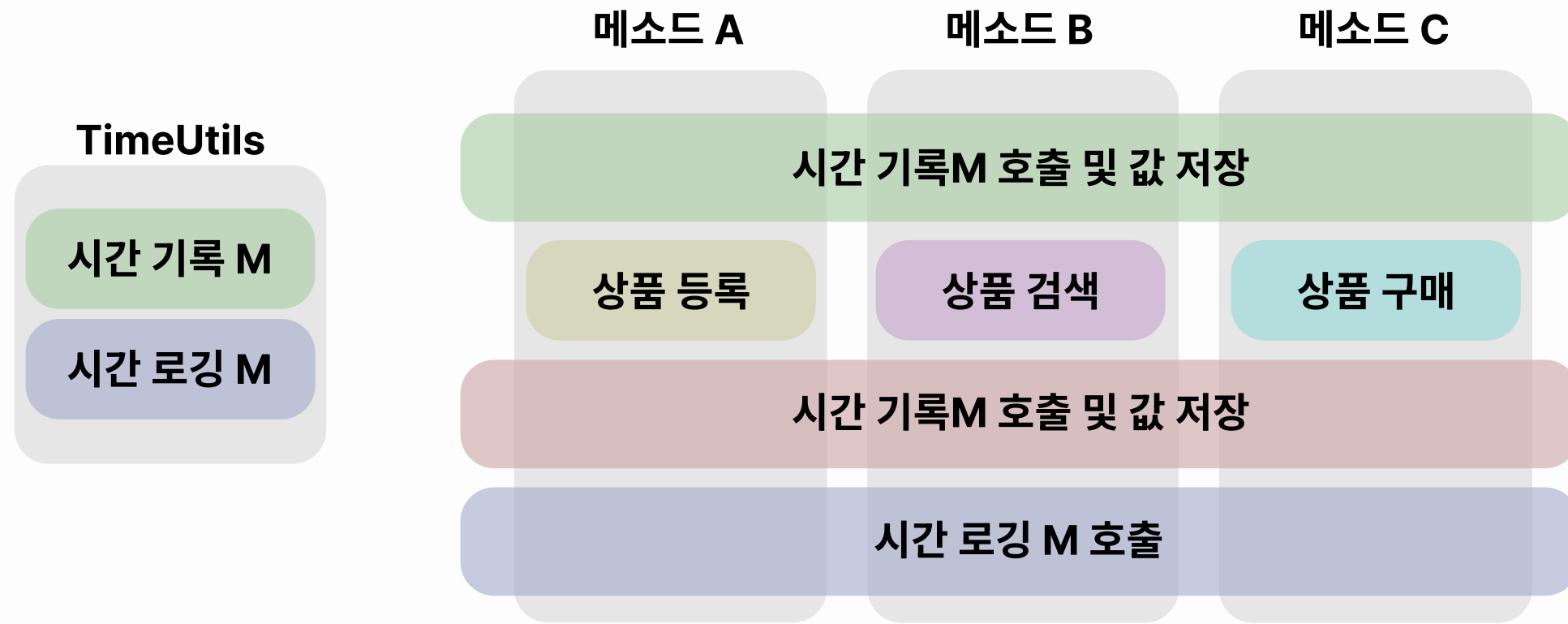
🕒 상품 등록 / 검색 / 구매 시 소요되는 시간을 알고 싶어요!



➡ 코드의 중복 발생

➡ 변경 사항 발생 시 관리 어려움

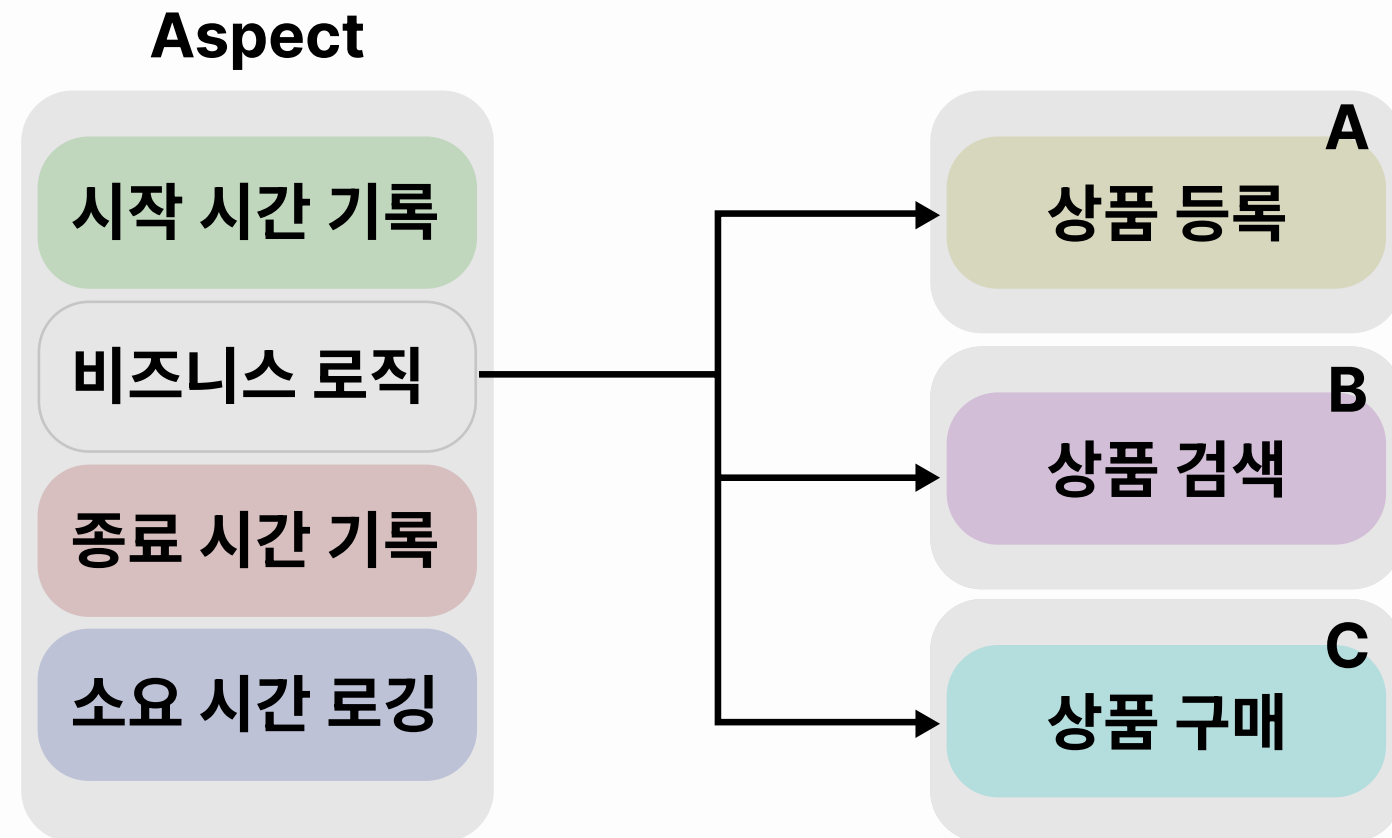
🔧 새로운 객체/메소드를 만들어 모듈화하면 유지보수가 쉬워지지 않나요?



➡ 시간 기록 / 로깅 로직은 분리에 성공하였지만, 코드 중복 문제는 해결X

➡ 만약 A, B, C가 서로 다른 클래스에 위치한다면, 매번 TimeUtils 객체 생성 필요

💡 AOP를 적용한다면?



- ➔ 핵심 관심사와 분리하여 변경 지점이 하나가 되도록 모듈화
- ➔ Service 단의 코드 변화 없이도 로깅을 적용시킬 수 있음 (유지보수성 ↑)

AOP 주요 개념

- **Aspect** 공통 기능을 모아둔 객체, 공통 기능을 구현할 객체 자체
- **Advice** Aspect 내부의 '공통 기능 각각의 로직', 즉 객체 내에서 실제로 구현되는 각각의 로직
 - @Around : 메서드 실행 전/후에 동작 추가
 - @Before : 메서드 실행 전에 동작 추가
 - @After : 메서드가 실행된 후 동작 추가 (성공/예외 발생 상관X)
 - @AfterReturning : 메서드가 성공적으로 실행된 후 동작 추가
 - @AfterThrowing : 메서드가 예외를 던진 후 동작 추가
- **JoinPoint** 공통 기능을 적용해야 하는 메소드의 '실행 시점'
- **Pointcut** 공통 기능을 적용할 '대상'
- **Weaving** 핵심 기능과 공통 기능을 연결하는 '행위'
- +) **Advisor** Advice와 Pointcut을 하나씩 가지고 있는 오브젝트 (Spring AOP에서 사용되는 개념)

AOP 구현 방법

1. 컴파일 타임(Compile-Time) 방식

- 컴파일 타임에서 AOP적용이 이루어지는 방식
- file.java → file.class로 컴파일하는 과정에서 해당하는 Aspect를 끼워넣음

2. 로드 타임(Load-Time) 방식

- 로드 타임에서 AOP가 이루어지는 방식
- 클래스 로더가 file.class라는 클래스를 메모리에 로드하는 시점에 Aspect를 끼워넣음

3. 런타임(Run-Time) 방식

- 런타임에서 AOP적용이 이루어지는 방식
- file이라는 클래스를, 부가 기능을 제공하는 프록시로 감싸서 실행
- 런타임 중에 프록시 객체를 생성하여 관점을 적용

1. **Compile-time weaving**: AspectJ 컴파일러는 우리의 aspect와 우리 애플리케이션의 소스 코드를 입력으로 취하고 출력으로 엮인 클래스 파일을 생성한다.

2. **Post-compile weaving** (컴파일 후 위빙): binary weaving.이라고도한다. 기존 클래스 파일과 JAR 파일을 위빙하는데 사용한다.

3. **Load-time weaving** : 클래스 로더가 클래스 파일을 JVM에로드 할 때까지 weaving이 연기된다는 점이 다르다.

- **Runtime weaving**: Aspect 가 대상 객체의 Proxy(JDK 동적 Proxy 나 CGLIB 의 Proxy)를 실행시 Weaving 된다.

Spring AOP는 사용자의 특정 호출 시점에 IoC 컨테이너에 의해 AOP를 할 수 있는 Proxy Bean을 생성해준다. 동적으로 생성된 Proxy Bean은 타깃의 메소드가 호출되는 시점에 부가기능을 추가할 메소드를 자체적으로 판단하고 가로채어 부가기능을 주입해준다. 이처럼 호출 시점에 동적으로 위빙을 한다 하여 런타임 위빙(Runtime Weaving)이라 한다. (인용)

Spring AOP vs AspectJ

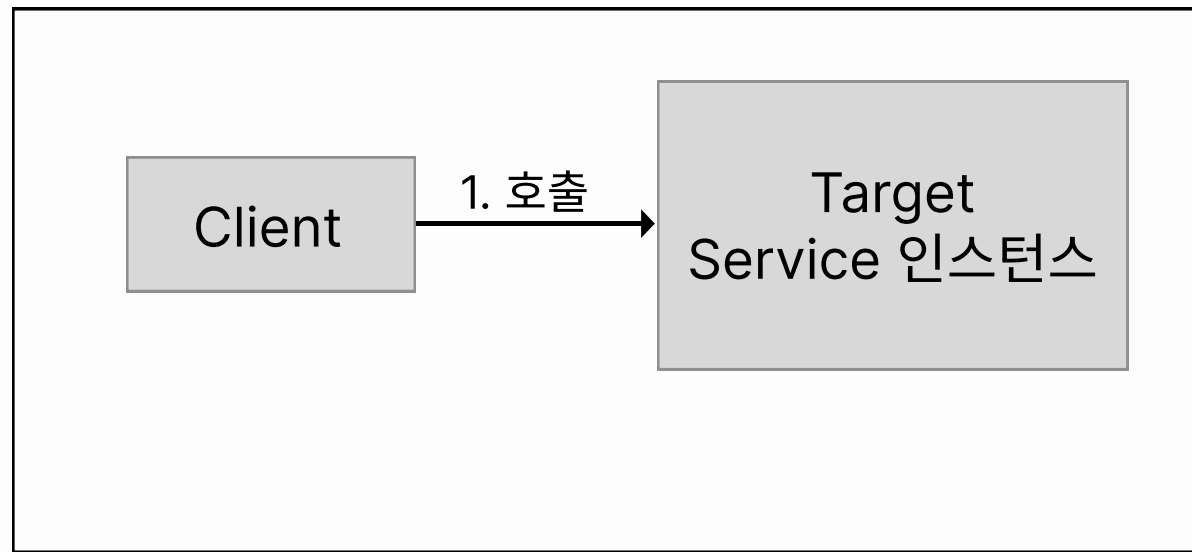
AspectJ

- Aspect 개념을 자바에 도입하여 AOP를 적용하도록 만든 프레임워크
- 자바 코드에서 동작하는 모든 객체에 대해, 완벽한 AOP 기능 제공을 목표로 함
- compile-time / post-compile / load-time 제공 (런타임 제공 X)
- JoinPoint : 생성자, 필드, 메소드 등 다양하게 지원
- 장점 : 성능 뛰어나고 기능이 강력
- 단점 : 비교적 사용법이나 내부 구조가 복잡

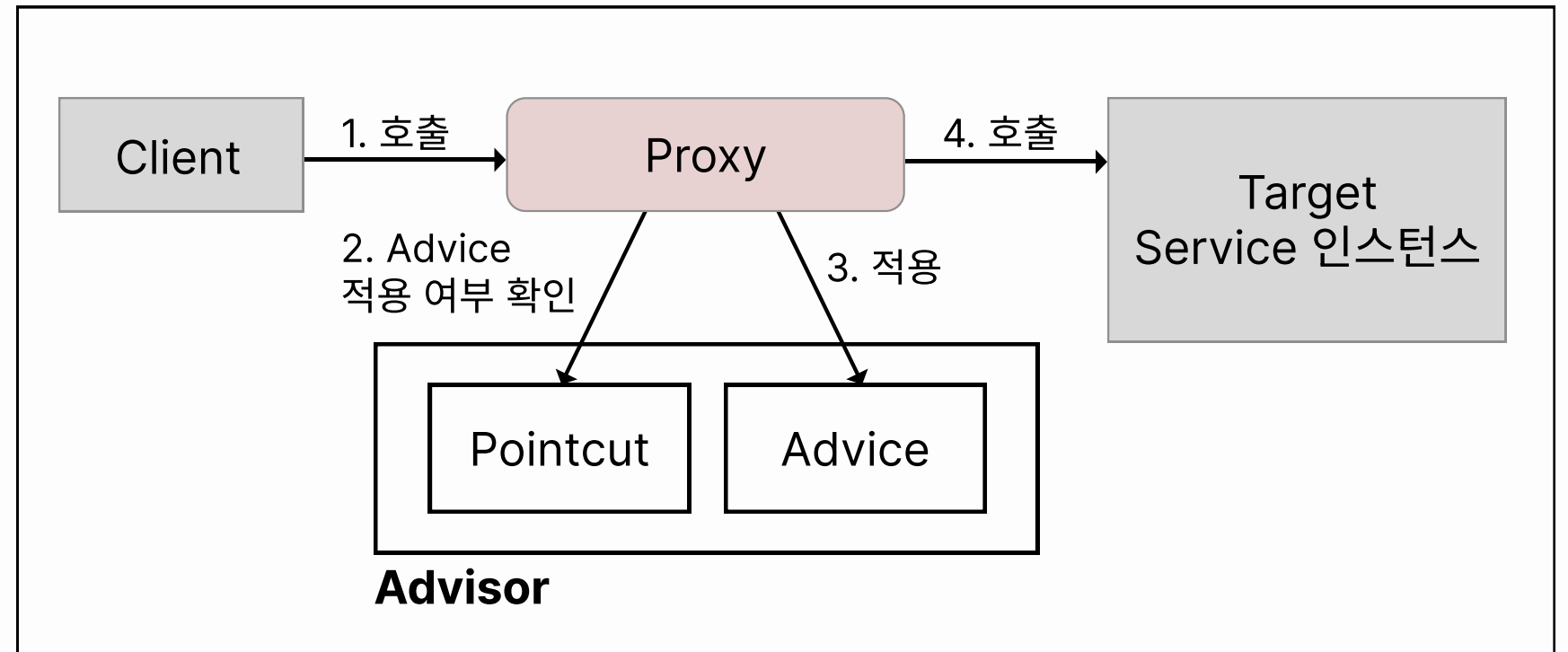
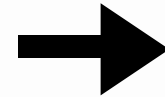
Spring AOP

- Aspect 개념을 Spring에 도입하여 AOP를 적용하도록 만든 프레임워크
- AspectJ를 내부적으로 구현하고 있음 (Spring AOP도 AspectJ를 기반으로 함)
- Spring Container가 관리하는 Bean에 대해, 간단한 AOP 기능만을 제공
- 런타임 시에만 weaving 가능
- JoinPoint : 메소드 레벨만 지원
- 런타임 시점에 동적으로 변할 수 있는 프록시 객체를 이용 → 앱 성능에 영향을 줄 수도 있음

AOP 적용 시 흐름 변화



AOP 적용 전

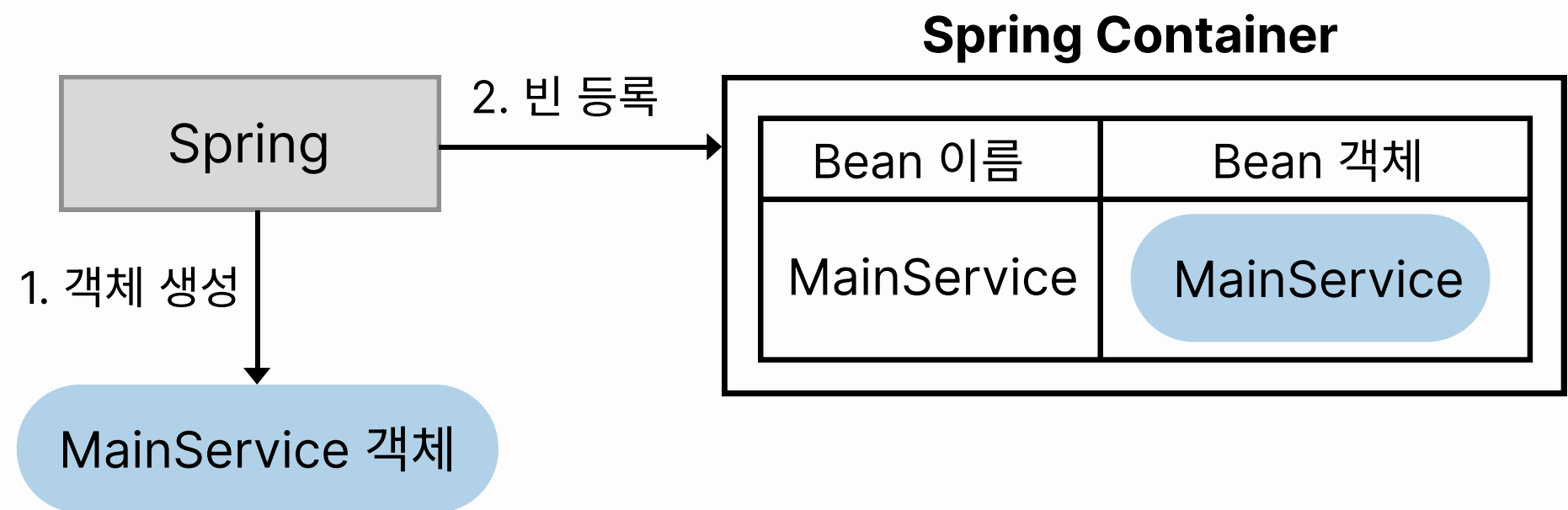


AOP 적용 후

1. Client가 바로 타겟 서비스 호출 (작업 수행)

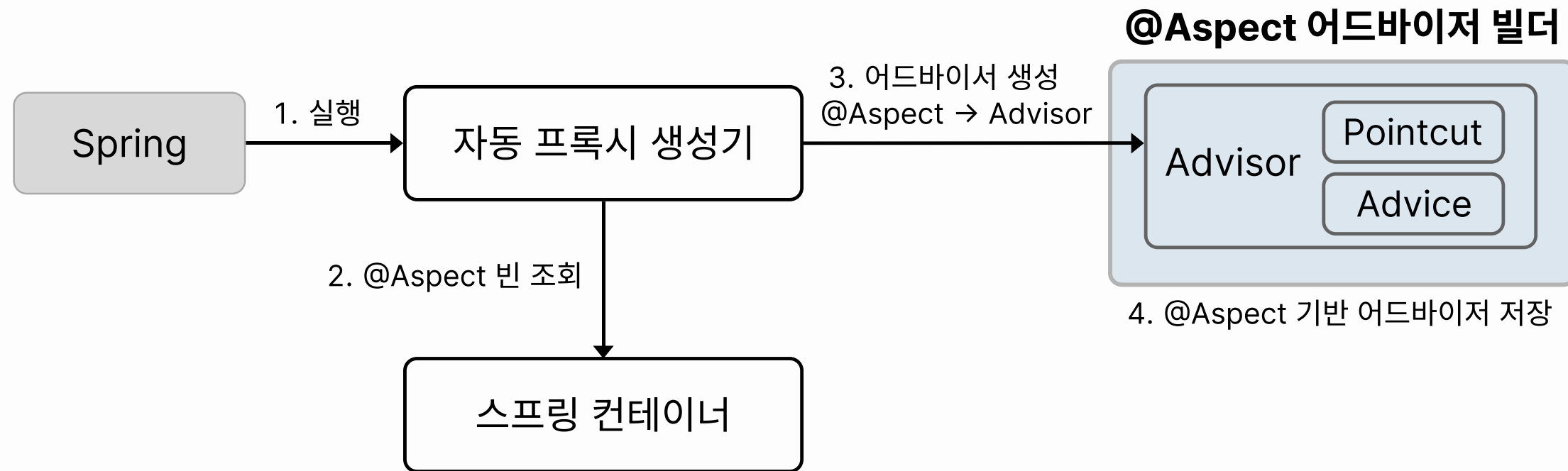
1. 서비스 호출
2. Proxy가 요청을 받아서 Advice 적용 여부 확인 (Pointcut)
3. 해당하는 Advice 적용
4. 타겟 서비스 호출 (작업 수행)

일반적인 Bean 등록 과정



- 1. @Bean 이나 @Component를 통해 빈등록
- 2. 스프링이 대상 객체를 생성
- 3. 스프링 컨테이너 내부의 빈 저장소에 등록

@Aspect → Advisor 생성 과정 (feat.자동 프록시 생성기)



- **빈 후처리기 (BeanPostProcessor)**

: 빈등록을 하기 전에 빈을 원하는대로 조작할 수 있는 기능을 제공하는 것

- **자동 프록시 생성기(AnnotationAwareAspectJAutoProxyCreator)**

: 빈 후처리기(BeanPostProcessor)를 구현한 클래스

- **@Aspect 어드바이저 빌더(BeansFactoryAspectJAdvisorsBuilder)**

: 어드바이저 생성하는 역할, 생성된 어드바이저는 이 안에 저장됨

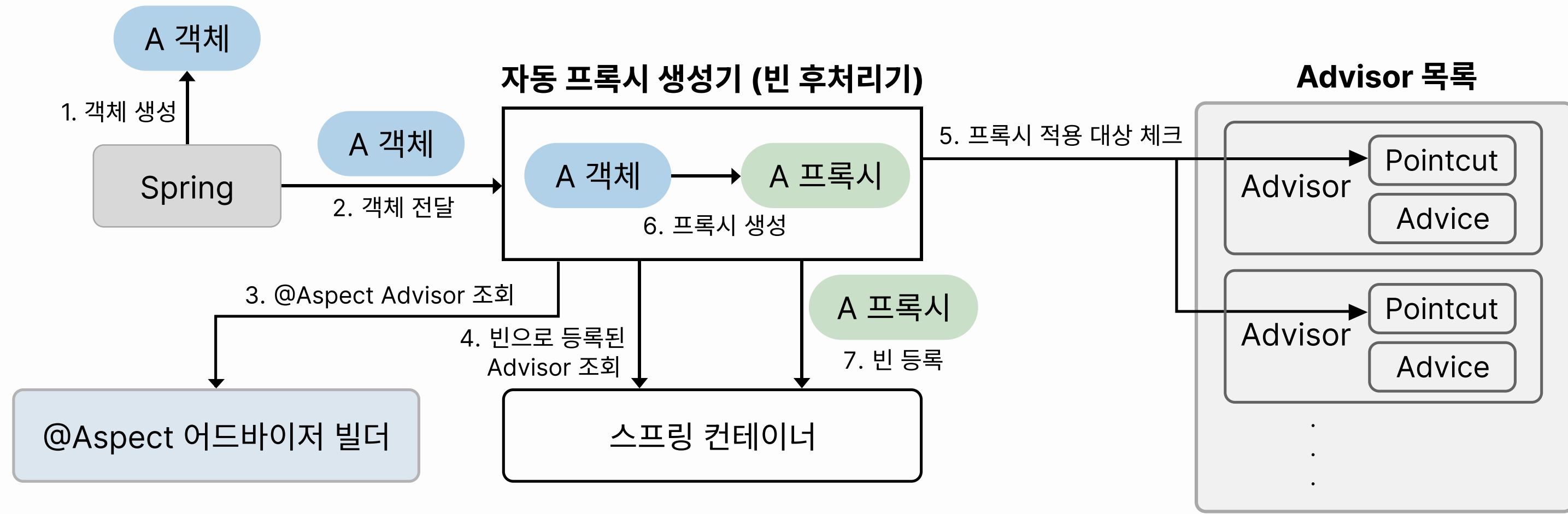
- **자동 프록시 생성기 (빈후처리기) 의 역할**

- @Aspect → Advisor로 변환 및 저장(캐싱)
- Advisor를 기반으로 프록시를 생성해주는 역할

+) 빈 후처리기의 처리 과정



AOP 적용 시 빈 등록 과정



1. 객체 생성 후 빈 후처리기에 객체 전달
2. @Aspect 어드바이저 빌더 / 스프링 컨테이너에 등록된 Advisor들을 조회
3. 조회된 Advisor들의 Pointcut을 사용해 프록시 적용 대상 여부 확인
4. 적용 대상이면 프록시 생성 및 반환, 아니면 기존 객체 반환
5. 스프링 컨테이너에 빈으로 등록

의존성 추가

implementation 'org.springframework.boot:spring-boot-starter-aop'

MainService 작성

- cmd 1이면 성공, 아니면 예외 던짐

```
1 package ureca.com.study.aop.service;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.springframework.stereotype.Service;
5
6 @Slf4j 2 usages new *
7 @Service
8 public class MainService {
9
10     // cmd가 1이면 성공, 아니면 실패하는 메소드
11     public String method(int cmd) throws Exception { 2 usages new *
12         if(cmd == 1) {
13             log.info("메소드 실행~");
14         } else {
15             log.info("!!!예외 발생!!!");
16             throw new Exception("");
17         }
18         return "반환값";
19     }
20 }
```

Test 코드 작성

- method 호출, 반환값 출력

```
⚡ @Test new *
void loggingTest() {
    String result = "초기값";
    try {
        result = mainService.method(cmd: 1);
    } catch (Exception e) {
        log.info("예외가 발생했습니다.");
    }
    log.info(result);
}
```

LoggingAspect 작성

```
8  @Slf4j new *
9  @Aspect
10 @Component
11 public class LoggingAspect {
12
13     @Pointcut("execution(* ureca.com.study.aop.service.*(..))") new *
14     private void allService();
15
16     @Around("allService()") new *
17     public Object aroundLogging(ProceedingJoinPoint joinPoint) throws Throwable {
18         log.info("@Around: 실행(전)");
19         Object result = joinPoint.proceed();
20         log.info("@Around: 실행(후)");
21         return result;
22     }
23
24     @Before("allService()") new *
25     public void beforeLogging() {
26         log.info("@Before: 실행");
27     }
28
29     @After("allService()") new *
30     public void afterLogging() {
31         log.info("@After: 실행");
32     }
33
34     @AfterReturning("allService()") new *
35     public void afterReturningLogging() {
36         log.info("@AfterReturning: 실행");
37     }
38
39     @AfterThrowing("allService()") new *
40     public void afterThrowingLogging() {
41         log.info("@AfterThrowing: 실행");
42     }
43 }
```

Annotation 추가

- @Component : 빈 등록 필요
- @Aspect : Advisor 편하게 생성하도록 도와줌

Pointcut 작성

- @Around() 내부에 바로 적어줘도 되지만, 중복 사용/유지보수를 위해 필드 선언을 해주기도
- 패키지 경로 / Bean의 이름 / 어노테이션 등으로도 설정 가능

Advice 작성

- 적용 대상(Pointcut)을 설정
- Advice는 기본적으로 순서를 보장하지 X
- 순서를 설정하고 싶다면 @Order 사용

@Around 테스트

```
@Around("allService(){}") new *
public Object aroundLogging(ProceedingJoinPoint joinPoint) throws Throwable {
    log.info("@Around: 실행(전)");
    Object result = joinPoint.proceed();
    log.info("@Around: 실행(후)");
    return result;
}
```

테스트 결과

```
ureca.com.study.aop.aop.LoggingAspect      : @Around: 실행(전)
u.com.study.aop.service.MainService        : 메소드 실행~
ureca.com.study.aop.aop.LoggingAspect      : @Around: 실행(후)
u.com.study.aop.AopApplicationTests        : 반환값
```

+) ProceedingJoinPoint

- 로직 내에서 비즈니스 로직이 실행될 지점을 설정 가능 (JoinPoint)
- 반환값을 Object로 받아와서 확인 및 후처리 가능
- 기타 등등...

@AfterReturning / @AfterThrowing 테스트

```
@AfterReturning("allService(){}") new *
public void afterReturningLogging() {
    log.info("@AfterReturning: 실행");
}

@AfterThrowing ("allService(){}") new *
💡 public void afterThrowingLogging() {
    log.info("@AfterThrowing: 실행");
}
```

성공 테스트

```
u.com.study.aop.service.MainService      : 메소드 실행~
ureca.com.study.aop.aop.LoggingAspect    : @AfterReturning: 실행
u.com.study.aop.AopApplicationTests      : 반환값
```

실패 테스트

```
u.com.study.aop.service.MainService      : !!!예외 발생!!!
ureca.com.study.aop.aop.LoggingAspect    : @AfterThrowing: 실행
u.com.study.aop.AopApplicationTests      : 예외가 발생했습니다.
u.com.study.aop.AopApplicationTests      : 초기값
```

JoinPoint로 파라미터 정보 받아오기

```
@Around("allService(){}") new *
public Object aroundLogging(ProceedingJoinPoint joinPoint) throws Throwable {
    MethodSignature signature = (MethodSignature) joinPoint.getSignature();
    String[] pNames = signature.getParameterNames();
    Object[] pVals = joinPoint.getArgs();

    for (int i = 0; i < pVals.length; i++) {
        String pInfo = pNames[i] + " = " + pVals[i];
        log.info(pInfo);
    }

    log.info("@Around: 실행(전)");
    Object result = joinPoint.proceed();
    log.info("@Around: 실행(후)");
    return result;
}
```

JoinPoint

- 클라이언트가 호출한 메소드의 시그니처 정보가 저장된 Signature 객체 리턴
- 리턴타입, 이름, 매개변수 등
- 기타 등등...


테스트 결과

```
ureca.com.study.aop.aop.LoggingAspect : cmd = 1
ureca.com.study.aop.aop.LoggingAspect : @Around: 실행(전)
u.com.study.aop.service.MainService : 메소드 실행~
ureca.com.study.aop.aop.LoggingAspect : @Around: 실행(후)
u.com.study.aop.AopApplicationTests : 반환값
```

Custom Annotation을 활용한 AOP 구현

```
@Retention(RetentionPolicy.RUNTIME) no usages new
@Target(ElementType.METHOD)
public @interface LogAnnotation {
}
```

```
// 커스텀 어노테이션 테스트
public String methodA() { 1 usage new *
    log.info("메소드 A 실행~");
    return "반환값 A";
}
```

 Change signature

```
// 커스텀 어노테이션 테스트
@LogAnnotation no usages new *
public String methodB() {
    log.info("메소드 B 실행~");
    return "반환값 B";
}
```

Service 작성

- 동일 로직 2개, methodB에만 어노테이션을 달아줌

어노테이션 생성

- @Retention : 어노테이션의 유효 기간
- @Target : 어노테이션을 달 대상 (default = 전체)

Test 코드 작성

```
@Test new *
void annoLoggingTest() {
    String resultA = mainService.methodA();
    log.info(resultA);

    log.info("=====");

    String resultB = mainService.methodB();
    log.info(resultB);
}
```

LoggingAspect 작성

```
// 어노테이션이 붙은 메소드들
@Pointcut("@annotation(ureca.com.study.aop.annotation.LogAnnotation)") new *
private void annoMethod(){};

@After("annoMethod(){}") new *
public void annoLogging() {
    log.info("Annotation 메소드가 실행되었습니다.");
}
```

- Pointcut 수정
- 어노테이션 포함 메소드 실행 시 추가 로깅을 하도록 로직 수정

테스트 결과

```
u.com.study.aop.service.MainService      : 메소드 A 실행~
u.com.study.aop.AopApplicationTests       : 반환값 A
u.com.study.aop.AopApplicationTests       : =====
u.com.study.aop.service.MainService       : 메소드 B 실행~
ureca.com.study.aop.aop.LoggingAspect     : Annotation 메소드가 실행되었습니다.
u.com.study.aop.AopApplicationTests       : 반환값 B
```

Annotation으로 값 받아오기 & 메소드의 Parameter 값 받아오기

```
@Around("@annotation(updateScore)")  Yeoeun Yang
public Object checkUpdatable(ProceedingJoinPoint joinPoint, UpdateScore updateScore) throws Throwable {
    boolean needUpdate = true;
    int addScore = updateScore.addScore();
    User user = null;

    // User 정보 찾기 (매개변수 검사로 찾기)
    Object[] args = joinPoint.getArgs();
    String[] paramNames = ((MethodSignature) joinPoint.getSignature()).getParameterNames();

    for (int i = 0; i < args.length; i++) {
        if (paramNames[i].equals("user") && args[i] instanceof User) {
            user = (User) args[i];
            break;
        } else if (paramNames[i].equals("userId") && args[i] instanceof Long userId) {
            user = userRepository.findById(userId).orElseThrow(() -> new GlobalException(NOT_FOUND_USER));
            break;
        }
    }
}
```

Joinpoint로 매개변수 전달

- 변수값, 변수이름 각각 받아와서 비교
- getArgs()
 - 매개변수값을 받아옴
- signature.getParameterNames()
 - 매개변수 이름을 받아옴

+) Signature

AOP에서 제공하는 인터페이스로, 실행 중인 메소드의 기본 정보를 담고 있는 객체

```
@AfterReturning("@annotation(updateScoreAnno) && args(userId, ..)") new *
public void increaseScore(Long userId, UpdateScore updateScoreAnno) {
    User user = userRepository.findById(userId).orElseThrow(() -> new GlobalException(NOT_FOUND_USER));

    int addScore = updateScoreAnno.addScore(); // 어노테이션에서 추가할 점수 가져오기
    increaseScore(user, addScore);
}
```

Pointcut에서 매개변수 전달

→ 어노테이션 가진 메소드 중 변수명이 userId인 매개변수를 포함한 메소드를 대상으로 하라는 의미가 됨

참고 자료

- <https://velog.io/@dkwktm45/Spring-AOP를-알고-사용-방법을-알자>
- https://youtu.be/Hm0w_9ngDpM?si=suWIANUppT6iylli
- <https://junior-datalist.tistory.com/282>
- https://youtu.be/hdO_V7EMU4s?si=XdTbCE4khi2RyjrD
- <https://bepoz-study-diary.tistory.com/407>
- https://youtu.be/Hm0w_9ngDpM?si=yM7kfw0D7JbXAYnw
- <https://velog.io/@ddongh1122/SpringAOP-빈-후처리기-Aspect-프록시>
- <https://jiwondev.tistory.com/152>

추가 공부 사항

- JoinPoint / ProceedingJoinPoint
- Pointcut 정규 표현식