

# 11. 스프링 AOP - 포인트컷

#2.인강/6.핵심 원리 - 고급편/강의#

- 11. 스프링 AOP - 포인트컷 - 포인트컷 지시자
- 11. 스프링 AOP - 포인트컷 - 예제 만들기
- 11. 스프링 AOP - 포인트컷 - execution1
- 11. 스프링 AOP - 포인트컷 - execution2
- 11. 스프링 AOP - 포인트컷 - within
- 11. 스프링 AOP - 포인트컷 - args
- 11. 스프링 AOP - 포인트컷 - @target, @within
- 11. 스프링 AOP - 포인트컷 - @annotation, @args
- 11. 스프링 AOP - 포인트컷 - bean
- 11. 스프링 AOP - 포인트컷 - 매개변수 전달
- 11. 스프링 AOP - 포인트컷 - this, target
- 11. 스프링 AOP - 포인트컷 - 정리

## 포인트컷 지시자

지금부터 포인트컷 표현식을 포함한 포인트컷에 대해서 자세히 알아보자.

애스펙트J는 포인트컷을 편리하게 표현하기 위한 특별한 표현식을 제공한다.

예) `@Pointcut("execution(* hello.aop.order..*(..))")`

포인트컷 표현식은 AspectJ pointcut expression 즉 애스펙트J가 제공하는 포인트컷 표현식을 줄여서 말하는 것이다.

## 포인트컷 지시자

포인트컷 표현식은 `execution` 같은 포인트컷 지시자(Pointcut Designator)로 시작한다. 줄여서 PCD라 한다.

- 포인트컷 지시자의 종류
  - `execution`: 메소드 실행 조인 포인트를 매칭한다. 스프링 AOP에서 가장 많이 사용하고, 기능도 복잡하다.
  - `within`: 특정 타입 내의 조인 포인트를 매칭한다.
  - `args`: 인자가 주어진 타입의 인스턴스인 조인 포인트
  - `this`: 스프링 빈 객체(스프링 AOP 프록시)를 대상으로 하는 조인 포인트
  - `target`: Target 객체(스프링 AOP 프록시가 가르키는 실제 대상)를 대상으로 하는 조인 포인트
  - `@target`: 실행 객체의 클래스에 주어진 타입의 애노테이션이 있는 조인 포인트

- `@within` : 주어진 애노테이션이 있는 타입 내 조인 포인트
- `@annotation` : 메서드가 주어진 애노테이션을 가지고 있는 조인 포인트를 매칭
- `@args` : 전달된 실제 인수의 런타임 타입이 주어진 타입의 애노테이션을 갖는 조인 포인트
- `bean` : 스프링 전용 포인트컷 지시자, 빈의 이름으로 포인트컷을 지정한다.

포인트컷 지시자가 무엇을 뜻하는지, 사실 글로만 읽어보면 이해하기 쉽지 않다. 예제를 통해서 하나씩 이해해보자.

`execution`은 가장 많이 사용하고, 나머지는 자주 사용하지 않는다. 따라서 `execution`을 중점적으로 이해하자.

## 예제 만들기

포인트컷 표현식을 이해하기 위해 예제 코드를 하나 추가하자.

### ClassAop

```
package hello.aop.member.annotation;

import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassAop {
}
```

### MethodAop

```
package hello.aop.member.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MethodAop {
    String value();
}
```

## MemberService

```
package hello.aop.member;

public interface MemberService {
    String hello(String param);
}
```

## MemberServiceImpl

```
package hello.aop.member;

import hello.aop.member.annotation.ClassAop;
import hello.aop.member.annotation.MethodAop;
import org.springframework.stereotype.Component;

@ClassAop
@Component
public class MemberServiceImpl implements MemberService {

    @Override
    @MethodAop("test value")
    public String hello(String param) {
        return "ok";
    }

    public String internal(String param) {
        return "ok";
    }
}
```

## ExecutionTest

```
package hello.aop.pointcut;

import hello.aop.member.MemberServiceImpl;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;

import java.lang.reflect.Method;

import static org.assertj.core.api.Assertions.assertThat;

@Slf4j
public class ExecutionTest {

    AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
    Method helloMethod;

    @BeforeEach
    public void init() throws NoSuchMethodException {
        helloMethod = MemberServiceImpl.class.getMethod("hello", String.class);
    }

    @Test
    void printMethod() {
        //public java.lang.String
        hello.aop.member.MemberServiceImpl.hello(java.lang.String)
        log.info("helloMethod={}", helloMethod);
    }
}
```

`AspectJExpressionPointcut` 이 바로 포인트컷 표현식을 처리해주는 클래스다. 여기에 포인트컷 표현식을 지정하면 된다. `AspectJExpressionPointcut` 는 상위에서 `Pointcut` 인터페이스를 가진다.

`printMethod()` 테스트는 `MemberServiceImpl.hello(String)` 메서드의 정보를 출력해준다.

## 실행 결과

```
helloMethod = public java.lang.String
hello.aop.member.MemberServiceImpl.hello(java.lang.String)
```

이번에 알아볼 `execution`으로 시작하는 포인트컷 표현식은 이 메서드 정보를 매칭해서 포인트컷 대상을 찾아낸다.

## execution1

### execution 문법

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-
pattern(param-pattern)
          throws-pattern?)

execution(접근제어자? 반환타입 선언타입?메서드이름 (파라미터) 예외?)
```

- 메소드 실행 조인 포인트를 매칭한다.
- ?는 생략할 수 있다.
- \* 같은 패턴을 지정할 수 있다.

실제 코드를 하나씩 보면서 `execution`을 이해해보자.

### 가장 정확한 포인트컷

먼저 `MemberServiceImpl.hello(String)` 메서드와 가장 정확하게 모든 내용이 매칭되는 표현식이다.

### ExecutionTest - 추가

```
@Test
void exactMatch() {
    //public java.lang.String
    hello.aop.member.MemberServiceImpl.hello(java.lang.String)
    pointcut.setExpression("execution(public String
```

```
hello.aop.member.MemberServiceImpl.hello(String))");

    assertThat(pointcut.matches(helloMethod,
        MemberServiceImpl.class)).isTrue();
}
```

- AspectJExpressionPointcut 에 pointcut.setExpression 을 통해서 포인트컷 표현식을 적용할 수 있다.
- pointcut.matches(메서드, 대상 클래스) 를 실행하면 지정한 포인트컷 표현식의 매칭 여부를 true, false 로 반환한다.

### 매칭 조건

- 접근제어자?: public
- 반환타입: String
- 선언타입?: hello.aop.member.MemberServiceImpl
- 메서드이름: hello
- 파라미터: (String)
- 예외?: 생략

MemberServiceImpl.hello(String) 메서드와 포인트컷 표현식의 모든 내용이 정확하게 일치한다. 따라서 true 를 반환한다.

### 가장 많이 생략한 포인트컷

```
@Test
void allMatch() {
    pointcut.setExpression("execution(* *(..))");
    assertThat(pointcut.matches(helloMethod,
        MemberServiceImpl.class)).isTrue();
}
```

가장 많이 생략한 포인트컷이다.

### 매칭 조건

- 접근제어자?: 생략
- 반환타입: \*
- 선언타입?: 생략
- 메서드이름: \*
- 파라미터: (..)

- 예외?: 없음
- \*은 아무 값이 들어와도 된다는 뜻이다.
- 파라미터에서 ..은 파라미터의 타입과 파라미터 수가 상관없다는 뜻이다. (0..\*) 파라미터는 뒤에 자세히 정리하겠다.

## 메서드 이름 매칭 관련 포인트컷

```
@Test
void nameMatch() {
    pointcut.setExpression("execution(* hello(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void nameMatchStar1() {
    pointcut.setExpression("execution(* hel*(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void nameMatchStar2() {
    pointcut.setExpression("execution(* *el*(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void nameMatchFalse() {
    pointcut.setExpression("execution(* nono(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isFalse();
}
```

메서드 이름 앞 뒤에 \*을 사용해서 매칭할 수 있다.

## 패키지 매칭 관련 포인트컷

```
@Test
void packageExactMatch1() {
    pointcut.setExpression("execution(*
hello.aop.member.MemberServiceImpl.hello(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void packageExactMatch2() {
    pointcut.setExpression("execution(* hello.aop.member.*.*(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void packageExactMatchFalse() {
    pointcut.setExpression("execution(* hello.aop.*.*(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isFalse();
}

@Test
void packageMatchSubPackage1() {
    pointcut.setExpression("execution(* hello.aop.member...*(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void packageMatchSubPackage2() {
    pointcut.setExpression("execution(* hello.aop.*.*(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}
```



```
hello.aop.member.*(1).*(2)
```

- (1): 타입
- (2): 메서드 이름

패키지에서 `.`, `..` 의 차이를 이해해야 한다.

- `.`: 정확하게 해당 위치의 패키지
- `..`: 해당 위치의 패키지와 그 하위 패키지도 포함

## execution2

### 타입 매칭 - 부모 타입 허용

```
@Test
void typeExactMatch() {
    pointcut.setExpression("execution(*
hello.aop.member.MemberServiceImpl.*(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void typeMatchSuperType() {
    pointcut.setExpression("execution(*
hello.aop.member.MemberService.*(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}
```

`typeExactMatch()` 는 타입 정보가 정확하게 일치하기 때문에 매칭된다.

`typeMatchSuperType()` 을 주의해서 보아야 한다.

`execution`에서는 `MemberService` 처럼 부모 타입을 선언해도 그 자식 타입은 매칭된다. 다형성에서 부모타입 = 자식타입 이 할당 가능하다는 점을 떠올려보면 된다.

## 타입 매칭 - 부모 타입에 있는 메서드만 허용

```
@Test
void typeMatchInternal() throws NoSuchMethodException {
    pointcut.setExpression("execution(*
hello.aop.member.MemberServiceImpl.*(..))");

    Method internalMethod = MemberServiceImpl.class.getMethod("internal",
String.class);

    assertThat(pointcut.matches(internalMethod,
MemberServiceImpl.class)).isTrue();
}

//포인트컷으로 지정한 MemberService 는 internal 이라는 이름의 메서드가 없다.
@Test
void typeMatchNoSuperTypeMethodFalse() throws NoSuchMethodException {
    pointcut.setExpression("execution(*
hello.aop.member.MemberService.*(..))");

    Method internalMethod = MemberServiceImpl.class.getMethod("internal",
String.class);

    assertThat(pointcut.matches(internalMethod,
MemberServiceImpl.class)).isFalse();
}
```

typeMatchInternal() 의 경우 MemberServiceImpl 를 표현식에 선언했기 때문에 그 안에 있는 internal(String) 메서드도 매칭 대상이 된다.

typeMatchNoSuperTypeMethodFalse() 를 주의해서 보아야 한다.

이 경우 표현식에 부모 타입인 MemberService 를 선언했다. 그런데 자식 타입인 MemberServiceImpl 의 internal(String) 메서드를 매칭하려 한다. 이 경우 매칭에 실패한다. MemberService 에는 internal(String) 메서드가 없다!

부모 타입을 표현식에 선언한 경우 부모 타입에서 선언한 메서드가 자식 타입에 있어야 매칭에 성공한다. 그래서 부모 타입에 있는 hello(String) 메서드는 매칭에 성공하지만, 부모 타입에 없는 internal(String) 는 매칭에 실패한다.

## 파라미터 매칭

```
//String 타입의 파라미터 허용
//(String)
@Test
```

```

void argsMatch() {
    pointcut.setExpression("execution(* *(String))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

//파라미터가 없어야 함
//(())
@Test
void argsMatchNoArgs() {
    pointcut.setExpression("execution(* *())");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isFalse();
}

//정확히 하나의 파라미터 허용, 모든 타입 허용
//(Xxx)
@Test
void argsMatchStar() {
    pointcut.setExpression("execution(* *(*))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

//숫자와 무관하게 모든 파라미터, 모든 타입 허용
//파라미터가 없어도 됨
//(), (Xxx), (Xxx, Xxx)
@Test
void argsMatchAll() {
    pointcut.setExpression("execution(* *(..))");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

//String 타입으로 시작, 숫자와 무관하게 모든 파라미터, 모든 타입 허용
//(String), (String, Xxx), (String, Xxx, Xxx) 허용
@Test
void argsMatchComplex() {
    pointcut.setExpression("execution(* *(String, ..))");
}

```

```

    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

```

### execution 파라미터 매칭 규칙은 다음과 같다.

- (String) : 정확하게 String 타입 파라미터
- () : 파라미터가 없어야 한다.
- (\*) : 정확히 하나의 파라미터, 단 모든 타입을 허용한다.
- (\*, \*) : 정확히 두 개의 파라미터, 단 모든 타입을 허용한다.
- (..) : 숫자와 무관하게 모든 파라미터, 모든 타입을 허용한다. 참고로 파라미터가 없어도 된다. 0..\*로 이해하면 된다.
- (String, ..) : String 타입으로 시작해야 한다. 숫자와 무관하게 모든 파라미터, 모든 타입을 허용한다.
  - 예) (String), (String, Xxx), (String, Xxx, Xxx) 허용

## within

within 지시자는 특정 타입 내의 조인 포인트들로 매칭을 제한한다. 쉽게 이야기해서 해당 타입이 매칭되면 그 안의 메서드(조인 포인트)들이 자동으로 매칭된다.

문법은 단순한데 execution에서 타입 부분만 사용한다고 보면 된다.

### WithinTest

```

package hello.aop.pointcut;

import hello.aop.member.MemberServiceImpl;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;

import java.lang.reflect.Method;

import static org.assertj.core.api.Assertions.assertThat;

public class WithinTest {

```

```

AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
Method helloMethod;

@BeforeEach
public void init() throws NoSuchMethodException {
    helloMethod = MemberServiceImpl.class.getMethod("hello", String.class);
}

@Test
void withinExact() {
    pointcut.setExpression("within(hello.aop.member.MemberServiceImpl)");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void withinStar() {
    pointcut.setExpression("within(hello.aop.member.*Service*)");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}

@Test
void withinSubPackage() {
    pointcut.setExpression("within(hello.aop.*)");
    assertThat(pointcut.matches(helloMethod,
MemberServiceImpl.class)).isTrue();
}
}

```

코드를 보면 이해하는데 어려움은 없을 것이다.

## 주의

그런데 `within` 사용시 주의해야 할 점이 있다. 표현식에 부모 타입을 지정하면 안된다는 점이다. 정확하게 타입이 맞아야 한다. 이 부분에서 `execution`과 차이가 난다.

## WithinTest - 추가

```

@Test
@DisplayName("타겟의 타입에만 직접 적용, 인터페이스를 선정하면 안된다.")
void withinSuperTypeFalse() {
    pointcut.setExpression("within(hello.aop.member.MemberService)");
    assertThat(pointcut.matches(helloMethod,
        MemberServiceImpl.class)).isFalse();
}

@Test
@DisplayName("execution은 타입 기반, 인터페이스를 선정 가능.")
void executionSuperTypeTrue() {
    pointcut.setExpression("execution(*
        hello.aop.member.MemberService.*(..))");
    assertThat(pointcut.matches(helloMethod,
        MemberServiceImpl.class)).isTrue();
}

```

부모 타입(여기서는 MemberService 인터페이스) 지정시 within 은 실패하고, execution 은 성공하는 것을 확인할 수 있다.

## args

- args : 인자가 주어진 타입의 인스턴스인 조인 포인트로 매칭
- 기본 문법은 execution 의 args 부분과 같다.

### execution과 args의 차이점

- execution 은 파라미터 타입이 정확하게 매칭되어야 한다. execution 은 클래스에 선언된 정보를 기반으로 판단한다.
- args 는 부모 타입을 허용한다. args 는 실제 넘어온 파라미터 객체 인스턴스를 보고 판단한다.

## ArgsTest

```
package hello.aop.pointcut;
```

```
import hello.aop.member.MemberServiceImpl;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;

import java.lang.reflect.Method;

import static org.assertj.core.api.Assertions.assertThat;

public class ArgsTest {

    Method helloMethod;

    @BeforeEach
    public void init() throws NoSuchMethodException {
        helloMethod = MemberServiceImpl.class.getMethod("hello", String.class);
    }

    private AspectJExpressionPointcut pointcut(String expression) {
        AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
        pointcut.setExpression(expression);
        return pointcut;
    }

    @Test
    void args() {
        //hello(String)과 매칭
        assertThat(pointcut("args(String)")
            .matches(helloMethod, MemberServiceImpl.class)).isTrue();
        assertThat(pointcut("args(Object)")
            .matches(helloMethod, MemberServiceImpl.class)).isTrue();
        assertThat(pointcut("args()")
            .matches(helloMethod, MemberServiceImpl.class)).isFalse();
        assertThat(pointcut("args(..)")
            .matches(helloMethod, MemberServiceImpl.class)).isTrue();
        assertThat(pointcut("args(*)")
            .matches(helloMethod, MemberServiceImpl.class)).isTrue();
        assertThat(pointcut("args(String,..)")
            .matches(helloMethod, MemberServiceImpl.class)).isTrue();
    }
}
```

```

}

/**
 * execution(* *(java.io.Serializable)): 메서드의 시그니처로 판단 (정적)
 * args(java.io.Serializable): 런타임에 전달된 인수로 판단 (동적)
 */
@Test
void argsVsExecution() {
    //Args
    assertThat(pointcut("args(String)")
        .matches(helloMethod, MemberServiceImpl.class)).isTrue();
    assertThat(pointcut("args(java.io.Serializable)")
        .matches(helloMethod, MemberServiceImpl.class)).isTrue();
    assertThat(pointcut("args(Object)")
        .matches(helloMethod, MemberServiceImpl.class)).isTrue();

    //Execution
    assertThat(pointcut("execution(* *(String))")
        .matches(helloMethod, MemberServiceImpl.class)).isTrue();
    assertThat(pointcut("execution(* *(java.io.Serializable))") //매칭 실패
        .matches(helloMethod, MemberServiceImpl.class)).isFalse();
    assertThat(pointcut("execution(* *(Object))") //매칭 실패
        .matches(helloMethod, MemberServiceImpl.class)).isFalse();
}
}

```

- `pointcut()`: `AspectJExpressionPointcut`에 포인트컷은 한번만 지정할 수 있다. 이번 테스트에서는 테스트를 편리하게 진행하기 위해 포인트컷을 여러번 지정하기 위해 포인트컷 자체를 생성하는 메서드를 만들었다.
- 자바가 기본으로 제공하는 `String`은 `Object`, `java.io.Serializable`의 하위 타입이다.
- 정적으로 클래스에 선언된 정보만 보고 판단하는 `execution(* *(Object))`는 매칭에 실패한다.
- 동적으로 실제 파라미터로 넘어온 객체 인스턴스로 판단하는 `args(Object)`는 매칭에 성공한다. (부모 타입 허용)

## 참고

`args` 지시자는 단독으로 사용되기 보다는 뒤에서 설명할 파라미터 바인딩에서 주로 사용된다.



## @target, @within

### 정의

- @target : 실행 객체의 클래스에 주어진 타입의 애노테이션이 있는 조인 포인트
- @within : 주어진 애노테이션이 있는 타입 내 조인 포인트

### 설명

@target, @within 은 다음과 같이 타입에 있는 애노테이션으로 AOP 적용 여부를 판단한다.

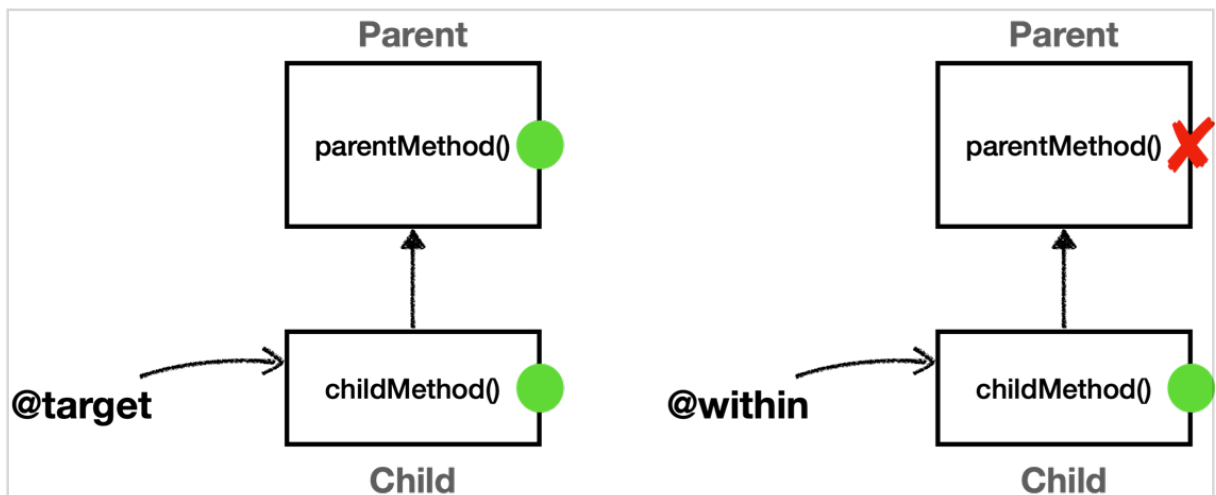
- @target(hello.aop.member.annotation.ClassAop)
- @within(hello.aop.member.annotation.ClassAop)

```
@ClassAop  
class Target{}
```

### @target vs @within

- @target 은 인스턴스의 모든 메서드를 조인 포인트로 적용한다.
- @within 은 해당 타입 내에 있는 메서드만 조인 포인트로 적용한다.

쉽게 이야기해서 @target 은 부모 클래스의 메서드까지 어드바이스를 다 적용하고, @within 은 자기 자신의 클래스에 정의된 메서드에만 어드바이스를 적용한다.



## AtTargetAtWithinTest

```
package hello.aop.pointcut;

import hello.aop.member.annotation.ClassAop;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;

@Slf4j
@Import({AtTargetAtWithinTest.Config.class})
@SpringBootTest
public class AtTargetAtWithinTest {

    @Autowired
    Child child;

    @Test
    void success() {
        log.info("child Proxy={}", child.getClass());
        child.childMethod(); //부모, 자식 모두 있는 메서드
        child.parentMethod(); //부모 클래스만 있는 메서드
    }

    static class Config {

        @Bean
        public Parent parent() {
            return new Parent();
        }
    }
}
```

```

@Bean
public Child child() {
    return new Child();
}

@Bean
public AtTargetAtWithinAspect atTargetAtWithinAspect() {
    return new AtTargetAtWithinAspect();
}
}

static class Parent {
    public void parentMethod(){} //부모에만 있는 메서드
}

@ClassAop
static class Child extends Parent {
    public void childMethod(){}
}

@Slf4j
@Aspect
static class AtTargetAtWithinAspect {

    //@target: 인스턴스 기준으로 모든 메서드의 조인 포인트를 선정, 부모 타입의 메서드도 적용
    @Around("execution(* hello.aop..*(..)) &&
@target(hello.aop.member.annotation.ClassAop)")
    public Object atTarget(ProceedingJoinPoint joinPoint) throws Throwable
    {
        log.info("[@target] {}", joinPoint.getSignature());
        return joinPoint.proceed();
    }

    //@within: 선택된 클래스 내부에 있는 메서드만 조인 포인트로 선정, 부모 타입의 메서드는
    적용되지 않음
    @Around("execution(* hello.aop..*(..)) &&
@within(hello.aop.member.annotation.ClassAop)")
    public Object atWithin(ProceedingJoinPoint joinPoint) throws Throwable
    {
        log.info("[@within] {}", joinPoint.getSignature());
    }
}

```

```

        return joinPoint.proceed();
    }
}

```

## 실행 결과

```

[@target] void hello.aop.pointcut.AtTargetAtWithinTest$Child.childMethod()
[@within] void hello.aop.pointcut.AtTargetAtWithinTest$Child.childMethod()
[@target] void hello.aop.pointcut.AtTargetAtWithinTest$Parent.parentMethod()

```

`parentMethod()` 는 `Parent` 클래스에만 정의되어 있고, `Child` 클래스에 정의되어 있지 않기 때문에 `@within` 에서 AOP 적용 대상이 되지 않는다.

실행결과를 보면 `child.parentMethod()` 를 호출 했을 때 `@within` 이 호출되지 않은 것을 확인할 수 있다.

## 참고

`@target`, `@within` 지시자는 뒤에서 설명할 파라미터 바인딩에서 함께 사용된다.

## 주의

다음 포인트컷 지시자는 단독으로 사용하면 안된다. `args`, `@args`, `@target`

이번 예제를 보면 `execution(* hello.aop..*(..))` 를 통해 적용 대상을 줄여준 것을 확인할 수 있다. `args`, `@args`, `@target` 은 실제 객체 인스턴스가 생성되고 실행될 때 어드바이스 적용 여부를 확인할 수 있다.

실행 시점에 일어나는 포인트컷 적용 여부도 결국 프록시가 있어야 실행 시점에 판단할 수 있다. 프록시가 없다면 판단 자체가 불가능하다. 그런데 스프링 컨테이너가 프록시를 생성하는 시점은 스프링 컨테이너가 만들어지는 애플리케이션 로딩 시점에 적용할 수 있다. 따라서 `args`, `@args`, `@target` 같은 포인트컷 지시자가 있으면 스프링은 모든 스프링 빈에 AOP를 적용하려고 시도한다. 앞서 설명한 것 처럼 프록시가 없으면 실행 시점에 판단 자체가 불가능하다.

문제는 이렇게 모든 스프링 빈에 AOP 프록시를 적용하려고 하면 스프링이 내부에서 사용하는 빈 중에는 `final` 로 지정된 빈들도 있기 때문에 오류가 발생할 수 있다.

따라서 이러한 표현식은 최대한 프록시 적용 대상을 축소하는 표현식과 함께 사용해야 한다.

## @annotation, @args

### @annotation

#### 정의

@annotation : 메서드가 주어진 애노테이션을 가지고 있는 조인 포인트를 매칭

#### 설명

@annotation(hello.aop.member.annotation.MethodAop)

다음과 같이 메서드(조인 포인트)에 애노테이션이 있으면 매칭한다.

```
public class MemberServiceImpl {  
    @MethodAop("test value")  
    public String hello(String param) {  
        return "ok";  
    }  
}
```

### AtAnnotationTest

```
package hello.aop.pointcut;  
  
import hello.aop.member.MemberService;  
import lombok.extern.slf4j.Slf4j;  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.annotation.Around;  
import org.aspectj.lang.annotation.Aspect;  
import org.junit.jupiter.api.Test;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.context.annotation.Import;  
  
@Slf4j  
@Import(AtAnnotationTest.AtAnnotationAspect.class)  
@SpringBootTest  
public class AtAnnotationTest {  
  
    @Autowired
```

```

MemberService memberService;

@Test
void success() {
    log.info("memberService Proxy={}", memberService.getClass());
    memberService.hello("helloA");
}

@Slf4j
@Aspect
static class AtAnnotationAspect {

    @Around("@annotation(hello.aop.member.annotation.MethodAop)")
    public Object doAtAnnotation(ProceedingJoinPoint joinPoint) throws
    Throwable {
        log.info("[@annotation] {}", joinPoint.getSignature());
        return joinPoint.proceed();
    }

}
}

```

## 실행 결과

```
[@annotation] String hello.aop.member.MemberService.hello(String)
```

## @args

### 정의

- `@args`: 전달된 실제 인수의 런타임 타입이 주어진 타입의 애노테이션을 갖는 조인 포인트

### 설명

전달된 인수의 런타임 타입에 `@Check` 애노테이션이 있는 경우에 매칭한다.

```
@args(test.Check)
```

# bean

## 정의

- `bean`: 스프링 전용 포인트컷 지시자, 빈의 이름으로 지정한다.

## 설명

- 스프링 빈의 이름으로 AOP 적용 여부를 지정한다. 이것은 스프링에서만 사용할 수 있는 특별한 지시자이다.
- `bean(orderService) || bean(*Repository)`
- `*` 과 같은 패턴을 사용할 수 있다.

```
package hello.aop.pointcut;

import hello.aop.order.OrderService;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Import;

@Slf4j
@Import({BeanTest.BeanAspect.class})
@SpringBootTest
public class BeanTest {

    @Autowired
    OrderService orderService;

    @Test
    void success() {
        orderService.orderItem("itemA");
    }

    @Aspect
```

```
static class BeanAspect {
    @Around("bean(orderService) || bean(*Repository)")
    public Object doLog(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("[bean] {}", joinPoint.getSignature());
        return joinPoint.proceed();
    }
}
```

OrderService, \*Repository(OrderRepository) 의 메서드에 AOP가 적용된다.

### 실행 결과

```
[bean] void hello.aop.order.OrderService.orderItem(String)
[orderService] 실행
[bean] String hello.aop.order.OrderRepository.save(String)
[orderRepository] 실행
```

### 매개변수 전달

다음은 포인트컷 표현식을 사용해서 어드바이스에 매개변수를 전달할 수 있다.

**this, target, args, @target, @within, @annotation, @args**

다음과 같이 사용한다.

```
@Before("allMember() && args(arg,...)")
public void logArgs3(String arg) {
    log.info("[logArgs3] arg={}", arg);
}
```

- 포인트컷의 이름과 매개변수의 이름을 맞추어야 한다. 여기서는 arg 로 맞추었다.



- 추가로 타입이 메서드에 지정한 타입으로 제한된다. 여기서는 메서드의 타입이 `String`으로 되어 있기 때문에 다음과 같이 정의되는 것으로 이해하면 된다.
  - `args(arg,...)` → `args(String,...)`

다양한 매개변수 전달 예시를 확인해보자.

## ParameterTest

```
package hello.aop.pointcut;

import hello.aop.member.MemberService;
import hello.aop.member.annotation.ClassAop;
import hello.aop.member.annotation.MethodAop;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Import;

@Slf4j
@Import(ParameterTest.ParameterAspect.class)
@SpringBootTest
public class ParameterTest {

    @Autowired
    MemberService memberService;

    @Test
    void success() {
        log.info("memberService Proxy={}", memberService.getClass());
        memberService.hello("helloA");
    }

    @Slf4j
```

```

@Aspect
static class ParameterAspect {

    @Pointcut("execution(* hello.aop.member..*.*(..))")
    private void allMember() {}

    @Around("allMember()")
    public Object logArgs1(ProceedingJoinPoint joinPoint) throws Throwable
    {
        Object arg1 = joinPoint.getArgs()[0];
        log.info("[logArgs1]{}", arg={}, joinPoint.getSignature(), arg1);
        return joinPoint.proceed();
    }

    @Around("allMember() && args(arg,..)")
    public Object logArgs2(ProceedingJoinPoint joinPoint, Object arg)
    throws Throwable {
        log.info("[logArgs2]{}", arg={}, joinPoint.getSignature(), arg);
        return joinPoint.proceed();
    }

    @Before("allMember() && args(arg,..)")
    public void logArgs3(String arg) {
        log.info("[logArgs3] arg={}", arg);
    }

    @Before("allMember() && this(obj)")
    public void thisArgs(JoinPoint joinPoint, MemberService obj) {
        log.info("[this]{}", obj={}, joinPoint.getSignature(),
obj.getClass());
    }

    @Before("allMember() && target(obj)")
    public void targetArgs(JoinPoint joinPoint, MemberService obj) {
        log.info("[target]{}", obj={}, joinPoint.getSignature(),
obj.getClass());
    }

    @Before("allMember() && @target(annotation)")

```

```

    public void atTarget(JoinPoint joinPoint, ClassAop annotation) {
        log.info("[@target]{}", obj={}, joinPoint.getSignature(),
annotation);
    }

    @Before("allMember() && @within(annotation)")
    public void atWithin(JoinPoint joinPoint, ClassAop annotation) {
        log.info("[@within]{}", obj={}, joinPoint.getSignature(),
annotation);
    }

    @Before("allMember() && @annotation(annotation)")
    public void atAnnotation(JoinPoint joinPoint, MethodAop annotation) {
        log.info("[@annotation]{}", annotationValue={},
joinPoint.getSignature(), annotation.value());
    }
}
}

```

- logArgs1 : joinPoint.getArgs()[0] 와 같이 매개변수를 전달 받는다.
- logArgs2 : args(arg,..) 와 같이 매개변수를 전달 받는다.
- logArgs3 : @Before 를 사용한 축약 버전이다. 추가로 타입을 String 으로 제한했다.
- this : 프록시 객체를 전달 받는다.
- target : 실제 대상 객체를 전달 받는다.
- @target , @within : 타입의 애노테이션을 전달 받는다.
- @annotation : 메서드의 애노테이션을 전달 받는다. 여기서는 annotation.value() 로 해당 애노테이션의 값을 출력하는 모습을 확인할 수 있다.

## 실행 결과

순서는 조정했음

```

memberService Proxy=class hello.aop.member.MemberServiceImpl$
$EnhancerBySpringCGLIB$$82
[logArgs1]String hello.aop.member.MemberServiceImpl.hello(String), arg=helloA
[logArgs2]String hello.aop.member.MemberServiceImpl.hello(String), arg=helloA
[logArgs3] arg=helloA

```

```
[this]String hello.aop.member.MemberServiceImpl.hello(String), obj=class
hello.aop.member.MemberServiceImpl$$EnhancerBySpringCGLIB$$8
[target]String hello.aop.member.MemberServiceImpl.hello(String), obj=class
hello.aop.member.MemberServiceImpl

[@target]String hello.aop.member.MemberServiceImpl.hello(String),
obj=@hello.aop.member.annotation.ClassAop()
[@within]String hello.aop.member.MemberServiceImpl.hello(String),
obj=@hello.aop.member.annotation.ClassAop()
[@annotation]String hello.aop.member.MemberServiceImpl.hello(String),
annotationValue=test value
```

## this, target

### 정의

- `this`: 스프링 빈 객체(스프링 AOP 프록시)를 대상으로 하는 조인 포인트
- `target`: Target 객체(스프링 AOP 프록시가 가르키는 실제 대상)를 대상으로 하는 조인 포인트

### 설명

- `this`, `target` 은 다음과 같이 적용 타입 하나를 정확하게 지정해야 한다.

```
this(hello.aop.member.MemberService)
target(hello.aop.member.MemberService)
```

- \* 같은 패턴을 사용할 수 없다.
- 부모 타입을 허용한다.

### this vs target

단순히 타입 하나를 정하면 되는데, `this` 와 `target` 은 어떤 차이가 있을까?

스프링에서 AOP를 적용하면 실제 `target` 객체 대신에 프록시 객체가 스프링 빈으로 등록된다.

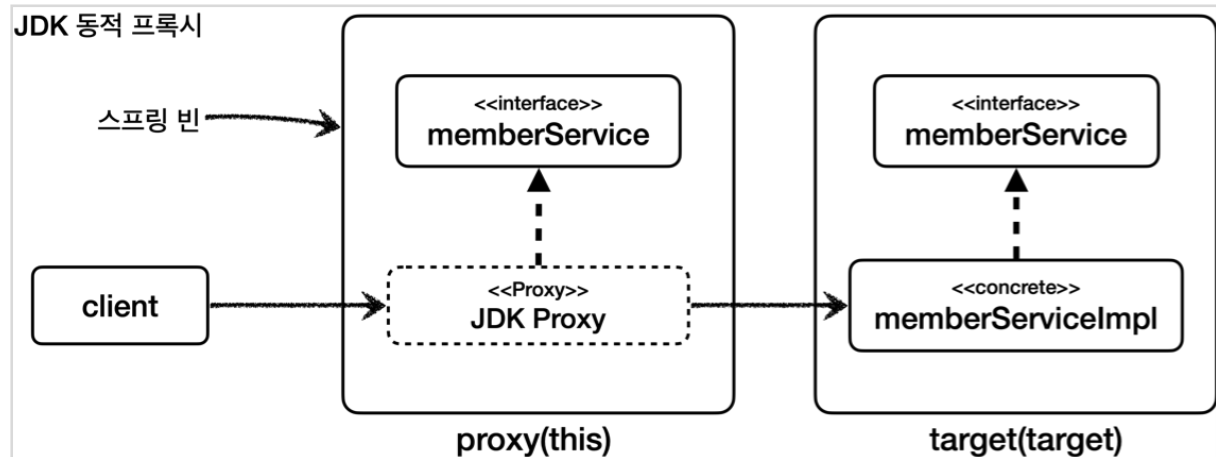
- `this` 는 스프링 빈으로 등록되어 있는 **프록시 객체**를 대상으로 포인트컷을 매칭한다.
- `target` 은 실제 **target 객체**를 대상으로 포인트컷을 매칭한다.

### 프록시 생성 방식에 따른 차이

스프링은 프록시를 생성할 때 JDK 동적 프록시와 CGLIB를 선택할 수 있다. 둘의 프록시를 생성하는 방식이 다르기 때문에 차이가 발생한다.

- JDK 동적 프록시: 인터페이스가 필수이고, 인터페이스를 구현한 프록시 객체를 생성한다.
- CGLIB: 인터페이스가 있어도 구체 클래스를 상속 받아서 프록시 객체를 생성한다.

## JDK 동적 프록시



먼저 JDK 동적 프록시를 적용했을 때 `this`, `target` 을 알아보자.

## MemberService 인터페이스 지정

- `this(hello.aop.member.MemberService)`
  - proxy 객체를 보고 판단한다. `this` 는 부모 타입을 허용하기 때문에 AOP가 적용된다.
- `target(hello.aop.member.MemberService)`
  - target 객체를 보고 판단한다. `target` 은 부모 타입을 허용하기 때문에 AOP가 적용된다.

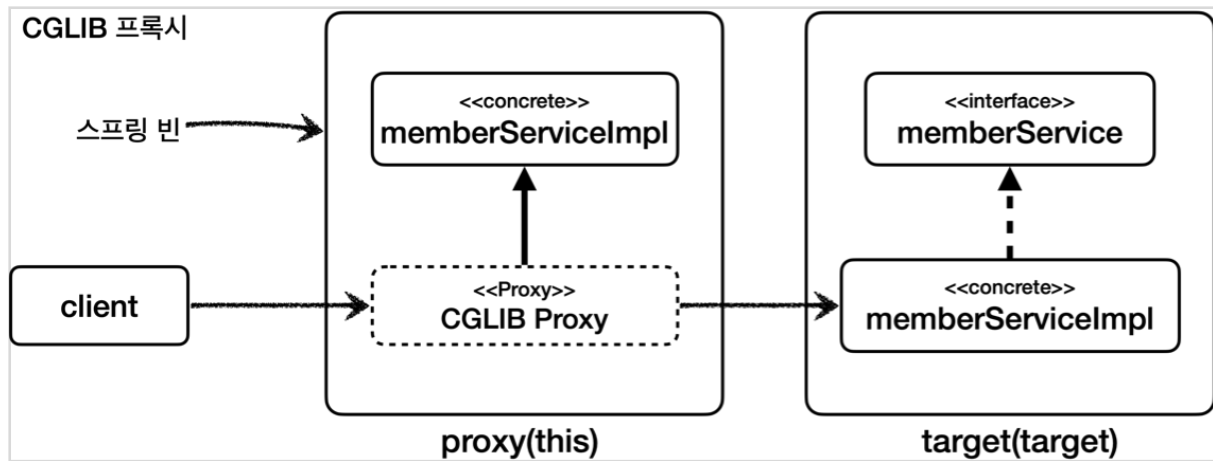
## MemberServiceImpl 구체 클래스 지정

- `this(hello.aop.member.MemberServiceImpl)` : proxy 객체를 보고 판단한다. JDK 동적 프록시로 만들어진 proxy 객체는 `MemberService` 인터페이스를 기반으로 구현된 새로운 클래스다. 따라서 `MemberServiceImpl` 를 전혀 알지 못하므로 **AOP 적용 대상이 아니다**.
- `target(hello.aop.member.MemberServiceImpl)` : target 객체를 보고 판단한다. target 객체가 `MemberServiceImpl` 타입이므로 AOP 적용 대상이다.

## 영상 오류 정정

영상에서 JDK Proxy는 `MemberService` 를 알 수 없다고 설명했는데, `MemberServiceImpl` 을 알 수 없다고 정정합니다.

## CGLIB 프록시



### MemberService 인터페이스 지정

- `this(hello.aop.member.MemberService)` : proxy 객체를 보고 판단한다. `this` 는 부모 타입을 허용하기 때문에 AOP가 적용된다.
- `target(hello.aop.member.MemberService)` : target 객체를 보고 판단한다. `target` 은 부모 타입을 허용하기 때문에 AOP가 적용된다.

### MemberServiceImpl 구체 클래스 지정

- `this(hello.aop.member.MemberServiceImpl)` : proxy 객체를 보고 판단한다. CGLIB로 만들어진 proxy 객체는 `MemberServiceImpl` 를 상속 받아서 만들었기 때문에 AOP가 적용된다. `this` 가 부모 타입을 허용하기 때문에 포인트컷의 대상이 된다.
- `target(hello.aop.member.MemberServiceImpl)` : target 객체를 보고 판단한다. target 객체가 `MemberServiceImpl` 타입이므로 AOP 적용 대상이다.

### 정리

프록시를 대상으로 하는 `this` 의 경우 구체 클래스를 지정하면 프록시 생성 전략에 따라서 다른 결과가 나올 수 있다는 점을 알아두자.

### ThisTargetTest

```

package hello.aop.pointcut;

import hello.aop.member.MemberService;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
  
```

```

import org.springframework.context.annotation.Import;

/**
 * application.properties
 * spring.aop.proxy-target-class=true CGLIB
 * spring.aop.proxy-target-class=false JDK 동적 프록시
 */
@Slf4j
@Import(ThisTargetTest.ThisTargetAspect.class)
@SpringBootTest(properties = "spring.aop.proxy-target-class=false") //JDK 동적
프록시
//@SpringBootTest(properties = "spring.aop.proxy-target-class=true") //CGLIB
public class ThisTargetTest {

    @Autowired
    MemberService memberService;

    @Test
    void success() {
        log.info("memberService Proxy={}", memberService.getClass());
        memberService.hello("helloA");
    }

    @Slf4j
    @Aspect
    static class ThisTargetAspect {

        //부모 타입 허용
        @Around("this(hello.aop.member.MemberService)")
        public Object doThisInterface(ProceedingJoinPoint joinPoint) throws
Throwable {
            log.info("[this-interface] {}", joinPoint.getSignature());
            return joinPoint.proceed();
        }

        //부모 타입 허용
        @Around("target(hello.aop.member.MemberService)")
        public Object doTargetInterface(ProceedingJoinPoint joinPoint) throws
Throwable {

```

```

        log.info("[target-interface] {}", joinPoint.getSignature());
        return joinPoint.proceed();
    }

    //this: 스프링 AOP 프록시 객체 대상
    //JDK 동적 프록시는 인터페이스를 기반으로 생성되므로 구현 클래스를 알 수 없음
    //CGLIB 프록시는 구현 클래스를 기반으로 생성되므로 구현 클래스를 알 수 있음
    @Around("this(hello.aop.member.MemberServiceImpl)")
    public Object doThis(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("[this-impl] {}", joinPoint.getSignature());
        return joinPoint.proceed();
    }

    //target: 실제 target 객체 대상
    @Around("target(hello.aop.member.MemberServiceImpl)")
    public Object doTarget(ProceedingJoinPoint joinPoint) throws Throwable
    {
        log.info("[target-impl] {}", joinPoint.getSignature());
        return joinPoint.proceed();
    }
}

```

this, target 은 실제 객체를 만들어야 테스트 할 수 있다. 테스트에서 스프링 컨테이너를 사용해서 target, proxy 객체를 모두 만들어서 테스트해보자.

- properties = {"spring.aop.proxy-target-class=false"} : application.properties 에 설정하는 대신에 해당 테스트에서만 설정을 임시로 적용한다. 이렇게 하면 각 테스트마다 다른 설정을 손쉽게 적용할 수 있다.
- spring.aop.proxy-target-class=false : 스프링이 AOP 프록시를 생성할 때 JDK 동적 프록시를 우선 생성한다. 물론 인터페이스가 없다면 CGLIB를 사용한다.
- spring.aop.proxy-target-class=true : 스프링이 AOP 프록시를 생성할 때 CGLIB 프록시를 생성한다. 참고로 이 설정을 생략하면 스프링 부트에서 기본으로 CGLIB를 사용한다. 이 부분은 뒤에서 자세히 설명한다.

```

@SpringBootTest(properties = "spring.aop.proxy-target-class=false") //JDK 동적
프록시

```



```
//@SpringBootTest(properties = "spring.aop.proxy-target-class=true") //CGLIB
```

## spring.aop.proxy-target-class=false

### JDK 동적 프록시 사용

```
memberService Proxy=class com.sun.proxy.$Proxy53
[target-impl] String hello.aop.member.MemberService.hello(String)
[target-interface] String hello.aop.member.MemberService.hello(String)
[this-interface] String hello.aop.member.MemberService.hello(String)
```

JDK 동적 프록시를 사용하면 `this(hello.aop.member.MemberServiceImpl)` 로 지정한 `[this-impl]` 부분이 출력되지 않는 것을 확인할 수 있다.

```
//@SpringBootTest(properties = "spring.aop.proxy-target-class=false") //JDK 동적
프록시
@SpringBootTest(properties = "spring.aop.proxy-target-class=true") //CGLIB
```

## spring.aop.proxy-target-class=true, 또는 생략(스프링 부트 기본 옵션)

### CGLIB 사용

```
memberService Proxy=class hello.aop.member.MemberServiceImpl$
$EnhancerBySpringCGLIB$$7df96bd3
[target-impl] String hello.aop.member.MemberServiceImpl.hello(String)
[target-interface] String hello.aop.member.MemberServiceImpl.hello(String)
[this-impl] String hello.aop.member.MemberServiceImpl.hello(String)
[this-interface] String hello.aop.member.MemberServiceImpl.hello(String)
```

### 참고

`this`, `target` 지시자는 단독으로 사용되기 보다는 파라미터 바인딩에서 주로 사용된다.

### 참고

혹시 해당 내용이 잘 이해가 되지 않으면 스프링 AOP 실무 주의 사항에서 프록시 기술과 한계를 듣고 다시 들어보면 더 이해가 쉬울 것이다.

정리