13. 스프링 AOP - 실무 주의사항

#인강/6.핵심 원리 - 고급편/강의#

프록시와 내부 호출 - 문제

스프링은 프록시 방식의 AOP를 사용한다.
따라서 AOP를 적용하려면 항상 프록시를 통해서 대상 객체(Target)을 호출해야 한다.
이렇게 해야 프록시에서 먼저 어드바이스를 호출하고, 이후에 대상 객체를 호출한다.
만약 프록시를 거치지 않고 대상 객체를 직접 호출하게 되면 AOP가 적용되지 않고, 어드바이스도 호출되지 않는다.

AOP를 적용하면 스프링은 대상 객체 대신에 프록시를 스프링 빈으로 등록한다. 따라서 스프링은 의존관계 주입시에 항상 프록시 객체를 주입한다. 프록시 객체가 주입되기 때문에 대상 객체를 직접 호출하는 문제는 일반적으로 발생하지 않는다. 하지만 대상 객체의 내부에서 메서드 호출이 발생하면 프록시를 거치지 않고 대상 객체를 직접 호출하는 문제가 발생한다. 실무에서 반드시 한번은 만나서 고생하는 문제이기 때문에 꼭 이해하고 넘어가자.

예제를 통해서 내부 호출이 발생할 때 어떤 문제가 발생하는지 알아보자. 먼저 내부 호출이 발생하는 예제를 만들어보자.

CallServiceV0

```
package hello.aop.internalcall;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;
```

```
@Slf4j
@Component
public class CallServiceV0 {

    public void external() {
        log.info("call external");
        internal(); //내부 메서드 호출(this.internal())
    }

    public void internal() {
        log.info("call internal");
    }

}
```

`CallServiceV0.external()` 을 호출하면 내부에서 `internal()` 이라는 자기 자신의 메서드를 호출한다. 자바 언어에서 메서드를 호출할 때 대상을 지정하지 않으면 앞에 자기 자신의 인스턴스를 뜻하는 `this` 가 붙게 된다. 그러니까 여기서는 `this.internal()` 이라고 이해하면 된다.

CallLogAspect

```
package hello.aop.internalcall.aop;

import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Slf4j
@Aspect
public class CallLogAspect {

    @Before("execution(* hello.aop.internalcall..*.*(..))")
    public void doLog(JoinPoint joinPoint) {
          log.info("aop={}", joinPoint.getSignature());
    }
}
```

`CallServiceV0` 에 AOP를 적용하기 위해서 간단한 `Aspect` 를 하나 만들자.

CallServiceV0Test

```
package hello.aop.internalcall;

import hello.aop.internalcall.aop.CallLogAspect;
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;

@Import(CallLogAspect.class)
@SpringBootTest
class CallServiceV0Test {

    @Autowired
    CallServiceV0 callServiceV0;

    @Test
    void external() {
        callServiceV0.external();
    }

    @Test
    void internal() {
        callServiceV0.internal();
    }
}
```

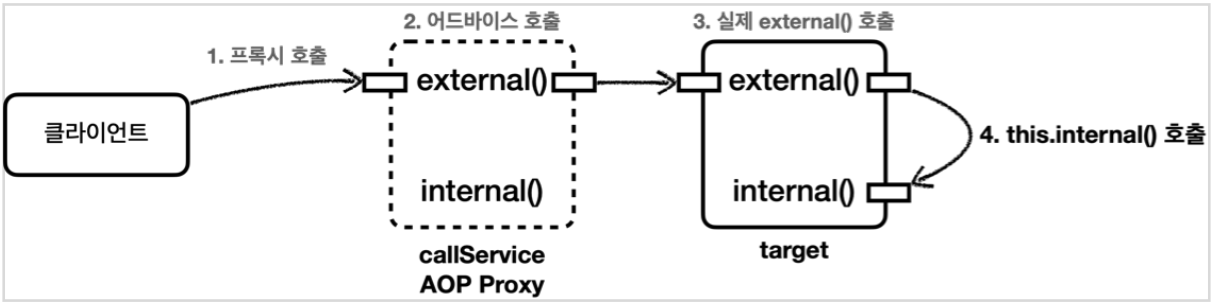
이제 앞서 만든 `CallServiceV0`을 실행할 수 있는 테스트 코드를 만들자.

- `@Import(CallLogAspect.class)` : 앞서 만든 간단한 `Aspect`를 스프링 빈으로 등록한다. 이렇게 해서 `CallServiceV0`에 AOP 프록시를 적용한다.
- `@SpringBootTest` : 내부에 컴포넌트 스캔을 포함하고 있다. `CallServiceV0`에 `@Component`가 붙어있으므로 스프링 빈 등록 대상이 된다.

먼저 `callServiceV0.external()`을 실행해보자. 이 부분이 중요하다.

실행 결과 - external()

```
1. //프록시 호출
2. CallLogAspect     : aop=void hello.aop.internalcall.CallServiceV0.external()
3. CallServiceV0     : call external
4. CallServiceV0     : call internal
```



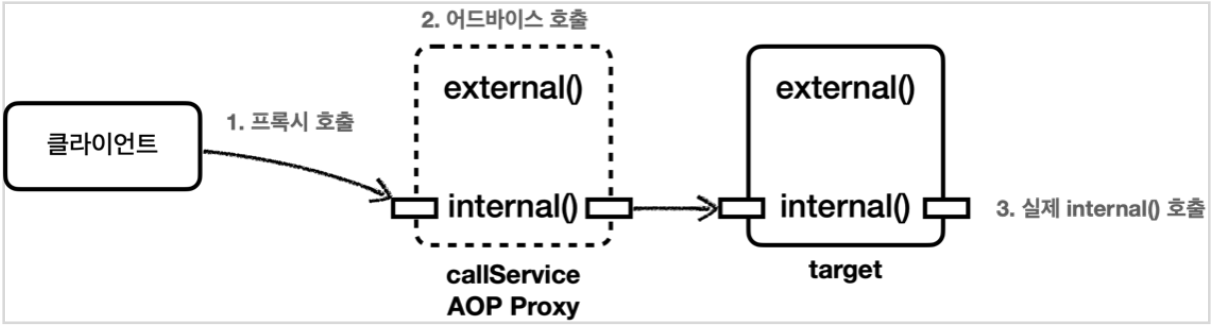
실행 결과를 보면 `callServiceV0.external()`을 실행할 때는 프록시를 호출한다. 따라서 `CallLogAspect` 어드바이스가 호출된 것을 확인할 수 있다.
그리고 AOP Proxy는 `target.external()`을 호출한다.
그런데 여기서 문제는 `callServiceV0.external()` 안에서 `internal()`을 호출할 때 발생한다. 이때는 `CallLogAspect` 어드바이스가 호출되지 않는다.

자바 언어에서 메서드 앞에 별도의 참조가 없으면 `this` 라는 뜻으로 자기 자신의 인스턴스를 가리킨다.
결과적으로 자기 자신의 내부 메서드를 호출하는 `this.internal()` 이 되는데, 여기서 `this` 는 실제 대상 객체(target)의 인스턴스를 뜻한다. 결과적으로 이러한 내부 호출은 프록시를 거치지 않는다. 따라서 어드바이스도 적용할 수 없다.

이번에는 외부에서 `internal()`을 호출하는 테스트를 실행해보자.

실행 결과 - internal()

```
CallLogAspect     : aop=void hello.aop.internalcall.CallServiceV0.internal()
CallServiceV0     : call internal
```



외부에서 호출하는 경우 프록시를 거치기 때문에 `internal()` 도 `CallLogAspect` 어드바이스가 적용된 것을 확인할 수 있다.

프록시 방식의 AOP 한계

스프링은 프록시 방식의 AOP를 사용한다. 프록시 방식의 AOP는 메서드 내부 호출에 프록시를 적용할 수 없다. 지금부터 이 문제를 해결하는 방법을 하나씩 알아보자.

참고

> 실제 코드에 AOP를 직접 적용하는 AspectJ를 사용하면 이런 문제가 발생하지 않는다. 프록시를 통하는 것이 아니라 해당 코드에 직접 AOP 적용 코드가 붙어 있기 때문에 내부 호출과 무관하게 AOP를 적용할 수 있다.
> 하지만 로드 타임 위빙 등을 사용해야 하는데, 설정이 복잡하고 JVM 옵션을 주어야 하는 부담이 있다.
> 그리고 지금부터 설명할 프록시 방식의 AOP에서 내부 호출에 대응할 수 있는 대안들도 있다.
>
> 이런 이유로 AspectJ를 직접 사용하는 방법은 실무에서는 거의 사용하지 않는다.
> 스프링 애플리케이션과 함께 직접 AspectJ 사용하는 방법은 스프링 공식 메뉴얼을 참고하자.
> https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aop-using-aspectj

프록시와 내부 호출 - 대안1 자기 자신 주입

내부 호출을 해결하는 가장 간단한 방법은 자기 자신을 의존관계 주입 받는 것이다.

CallServiceV1

```
package hello.aop.internalcall;

import lombok.extern.slf4j.Slf4j;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/**
 * 참고: 생성자 주입은 순환 사이클을 만들기 때문에 실패한다.
 */
@Slf4j
@Component
public class CallServiceV1 {

    private CallServiceV1 callServiceV1;

    @Autowired
    public void setCallServiceV1(CallServiceV1 callServiceV1) {
        this.callServiceV1 = callServiceV1;
    }

    public void external() {
        log.info("call external");
        callServiceV1.internal(); //외부 메서드 호출
    }

    public void internal() {
        log.info("call internal");
    }

}
```

`callServiceV1`를 수정자를 통해서 주입 받는 것을 확인할 수 있다. 스프링에서 AOP가 적용된 대상을 의존관계 주입 받으면 주입 받은 대상은 실제 자신이 아니라 프록시 객체이다.
`external()`을 호출하면 `callServiceV1.internal()`를 호출하게 된다. 주입받은 `callServiceV1`은 프록시이다. 따라서 프록시를 통해서 AOP를 적용할 수 있다.

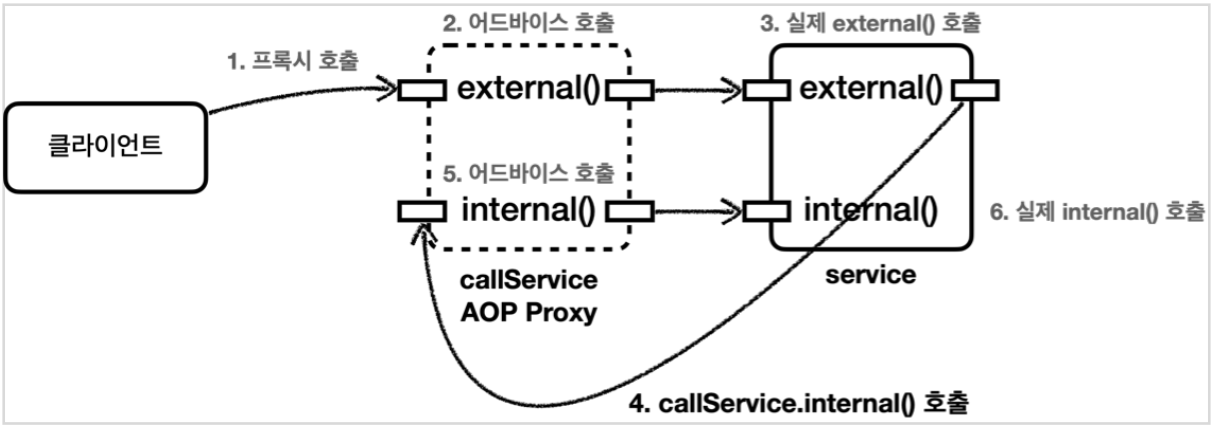
참고로 이 경우 생성자 주입시 오류가 발생한다. 본인을 생성하면서 주입해야 하기 때문에 순환 사이클이 만들어진다. 반면에 수정자 주입은 스프링이 생성된 이후에 주입할 수 있기 때문에 오류가 발생하지 않는다.

CallServiceV1Test

```
package hello.aop.internalcall;

import hello.aop.internalcall.aop.CallLogAspect;
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;

@Import(CallLogAspect.class)
@SpringBootTest
class CallServiceV1Test {

    @Autowired
    CallServiceV1 callServiceV1;

    @Test
    void external() {
        callServiceV1.external();
    }

}
```

실행 결과

```
CallLogAspect     : aop=void hello.aop.internalcall.CallServiceV1.external()
CallServiceV2     : call external
CallLogAspect     : aop=void hello.aop.internalcall.CallServiceV1.internal()
CallServiceV2     : call internal
```



실행 결과를 보면 이제는 `internal()` 을 호출할 때 자기 자신의 인스턴스를 호출하는 것이 아니라 프록시 인스턴스를 통해서 호출하는 것을 확인할 수 있다. 당연히 AOP도 잘 적용된다.

주의

스프링 부트 2.6부터는 순환 참조를 기본적으로 금지하도록 정책이 변경되었다. 따라서 이번 예제를 스프링 부트 2.6 이상의 버전에서 실행하면 다음과 같은 오류 메시지가 나오면서 정상 실행되지 않는다.

```
Error creating bean with name 'callServiceV1': Requested bean is currently in creation: Is there an unresolvable circular reference?
```

이 문제를 해결하려면 `application.properties` 에 다음을 추가해야 한다.

```
spring.main.allow-circular-references=true
```

앞으로 있을 다른 테스트에도 영향을 주기 때문에 **스프링 부트 2.6 이상이라면 이 설정을 꼭 추가**해야 한다.

프록시와 내부 호출 - 대안2 지연 조회

앞서 생성자 주입이 실패하는 이유는 자기 자신을 생성하면서 주입해야 하기 때문이다. 이 경우 수정자 주입을 사용하거나 지금부터 설명하는 지연 조회를 사용하면 된다.
스프링 빈을 지연해서 조회하면 되는데, `ObjectProvider(Provider)` , `ApplicationContext` 를 사용하면 된다.

CallServiceV2

```
package hello.aop.internalcall;
```

```java
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.ObjectProvider;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;

/**
 * ObjectProvider(Provider), ApplicationContext를 사용해서 지연(LAZY) 조회
 */
@Slf4j
@Component
@RequiredArgsConstructor
public class CallServiceV2 {

//  private final ApplicationContext applicationContext;
    private final ObjectProvider<CallServiceV2> callServiceProvider;

    public void external() {
        log.info("call external");
//      CallServiceV2 callServiceV2 =
applicationContext.getBean(CallServiceV2.class);
        CallServiceV2 callServiceV2 = callServiceProvider.getObject();
        callServiceV2.internal(); //외부 메서드 호출
    }

    public void internal() {
        log.info("call internal");
    }

}
```

`ObjectProvider`는 기본편에서 학습한 내용이다. `ApplicationContext`는 너무 많은 기능을 제공한다.
`ObjectProvider`는 객체를 스프링 컨테이너에서 조회하는 것을 스프링 빈 생성 시점이 아니라 실제 객체를 사용하는 시점으로 지연할 수 있다.
`callServiceProvider.getObject()`를 호출하는 시점에 스프링 컨테이너에서 빈을 조회한다.
여기서는 자기 자신을 주입 받는 것이 아니기 때문에 순환 사이클이 발생하지 않는다.

CallServiceV2Test

```java
package hello.aop.internalcall;

import hello.aop.internalcall.aop.CallLogAspect;
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;

@Import(CallLogAspect.class)
@SpringBootTest
class CallServiceV2Test {

    @Autowired
    CallServiceV2 callServiceV2;

    @Test
    void external() {
        callServiceV2.external();
    }

}
```

실행 결과

```
CallLogAspect     : aop=void hello.aop.internalcall.CallServiceV2.external()
CallServiceV2     : call external
CallLogAspect     : aop=void hello.aop.internalcall.CallServiceV2.internal()
CallServiceV2     : call internal
```

프록시와 내부 호출 - 대안3 구조 변경

앞선 방법들은 자기 자신을 주입하거나 또는 `Provider` 를 사용해야 하는 것 처럼 조금 어색한 모습을 만들었다.
가장 나은 대안은 내부 호출이 발생하지 않도록 구조를 변경하는 것이다. 실제 이 방법을 가장 권장한다.

CallServiceV3

```
package hello.aop.internalcall;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

/**
 * 구조를 변경(분리)
 */
@Slf4j
@Component
@RequiredArgsConstructor
public class CallServiceV3 {

    private final InternalService internalService;

    public void external() {
        log.info("call external");
        internalService.internal(); //외부 메서드 호출
    }
}
```

내부 호출을 `InternalService` 라는 별도의 클래스로 분리했다.

InternalService

```
package hello.aop.internalcall;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Component;

@Slf4j
@Component
public class InternalService {

    public void internal() {
```

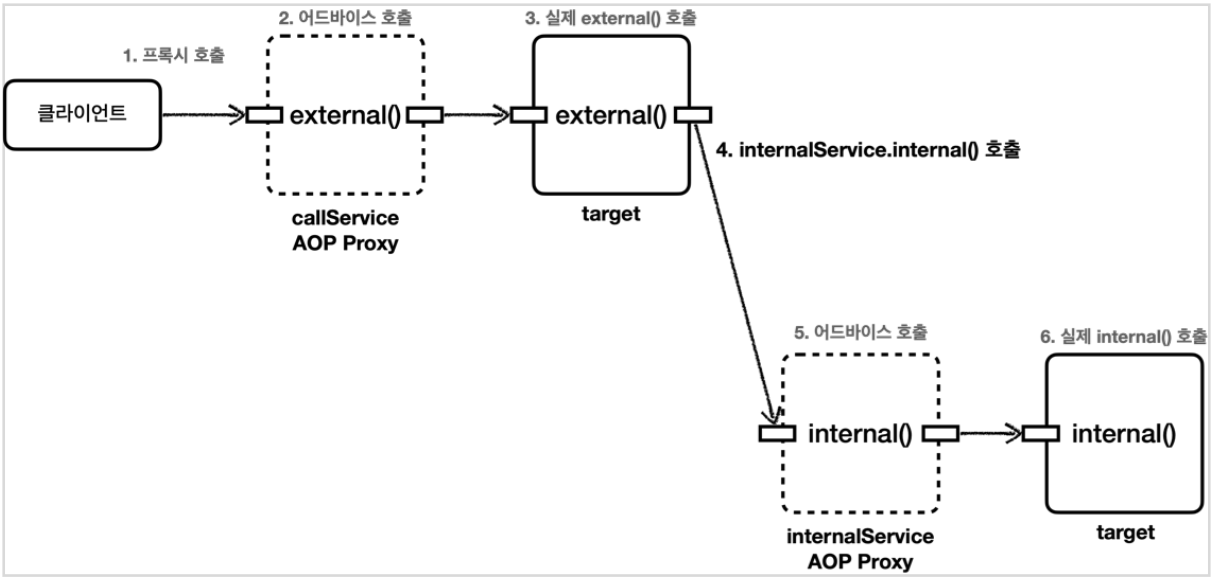
```
        log.info("call internal");
    }
}
```

CallServiceV3Test

```
package hello.aop.internalcall;

import hello.aop.internalcall.aop.CallLogAspect;
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;

@Import(CallLogAspect.class)
@SpringBootTest
class CallServiceV3Test {

    @Autowired
    CallServiceV3 callServiceV3;

    @Test
    void external() {
        callServiceV3.external();
    }

}
```

실행 결과

```
CallLogAspect     : aop=void hello.aop.internalcall.CallServiceV3.external()
CallServiceV3     : call external
CallLogAspect     : aop=void hello.aop.internalcall.InternalService.internal()
InternalService   : call internal
```



내부 호출 자체가 사라지고, `callService` ➡ `internalService`를 호출하는 구조로 변경되었다. 덕분에 자연스럽게 AOP가 적용된다.

여기서 구조를 변경한다는 것은 이렇게 단순하게 분리하는 것 뿐만 아니라 다양한 방법들이 있을 수 있다.

예를 들어서 다음과 같이 클라이언트에서 둘다 호출하는 것이다.

`클라이언트` ➡ `external()`
`클라이언트` ➡ `internal()`

물론 이 경우 `external()`에서 `internal()`을 내부 호출하지 않도록 코드를 변경해야 한다. 그리고 클라이언트 `external()`, `internal()`을 모두 호출하도록 구조를 변경하면 된다. (물론 가능한 경우에 한해서)

참고

AOP는 주로 트랜잭션 적용이나 주요 컴포넌트의 로그 출력 기능에 사용된다. 쉽게 이야기해서 인터페이스에 메서드가 나올 정도의 규모에 AOP를 적용하는 것이 적당하다. 더 풀어서 이야기하면 AOP는 `public` 메서드에만 적용한다. `private` 메서드처럼 작은 단위에는 AOP를 적용하지 않는다.

AOP 적용을 위해 `private` 메서드를 외부 클래스로 변경하고 `public`으로 변경하는 일은 거의 없다.

그러나 위 예제와 같이 `public` 메서드에서 `public` 메서드를 내부 호출하는 경우에는 문제가 발생한다.

실무에서 꼭 한번은 만나는 문제이기에 이번 강의에서 다루었다.

AOP가 잘 적용되지 않으면 내부 호출을 의심해보자.

프록시 기술과 한계 - 타입 캐스팅

JDK 동적 프록시와 CGLIB를 사용해서 AOP 프록시를 만드는 방법에는 각각 장단점이 있다.
JDK 동적 프록시는 인터페이스가 필수이고, 인터페이스를 기반으로 프록시를 생성한다.
CGLIB는 구체 클래스를 기반으로 프록시를 생성한다.

물론 인터페이스가 없고 구체 클래스만 있는 경우에는 CGLIB를 사용해야 한다. 그런데 인터페이스가 있는 경우에는 JDK 동적 프록시나 CGLIB 둘중에 하나를 선택할 수 있다.

스프링이 프록시를 만들때 제공하는 `ProxyFactory`에 `proxyTargetClass` 옵션에 따라 둘중 하나를 선택해서 프록시를 만들 수 있다.

- `proxyTargetClass=false` JDK 동적 프록시를 사용해서 인터페이스 기반 프록시 생성
- `proxyTargetClass=true` CGLIB를 사용해서 구체 클래스 기반 프록시 생성
- 참고로 옵션과 무관하게 인터페이스가 없으면 JDK 동적 프록시를 적용할 수 없으므로 CGLIB를 사용한다.

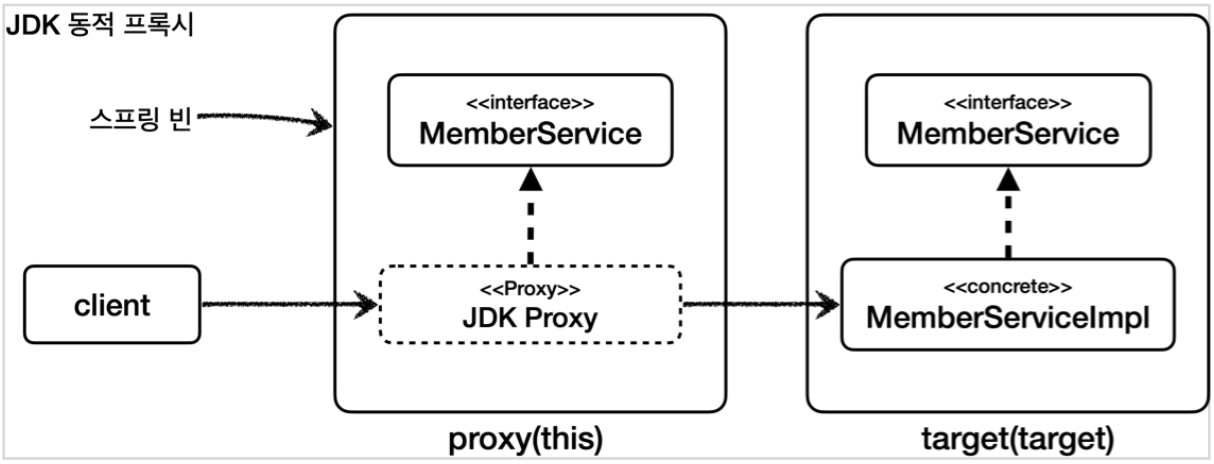
JDK 동적 프록시 한계
인터페이스 기반으로 프록시를 생성하는 JDK 동적 프록시는 구체 클래스로 타입 캐스팅이 불가능한 한계가 있다. 어떤 한계인지 코드를 통해서 알아보자.

ProxyCastingTest

```
package hello.aop.proxyvs;

import hello.aop.member.MemberService;
import hello.aop.member.MemberServiceImpl;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.aop.framework.ProxyFactory;

import static org.junit.jupiter.api.Assertions.assertThrows;

@Slf4j
public class ProxyCastingTest {

    @Test
    void jdkProxy() {
        MemberServiceImpl target = new MemberServiceImpl();
```

```
        ProxyFactory proxyFactory = new ProxyFactory(target);
        proxyFactory.setProxyTargetClass(false);//JDK 동적 프록시

        //프록시를 인터페이스로 캐스팅 성공
        MemberService memberServiceProxy = (MemberService)
proxyFactory.getProxy();

        log.info("proxy class={}", memberServiceProxy.getClass());

        //JDK 동적 프록시를 구현 클래스로 캐스팅 시도 실패, ClassCastException 예외 발생
        assertThrows(ClassCastException.class, () -> {
                    MemberServiceImpl castingMemberService =
(MemberServiceImpl) memberServiceProxy;
                }
        );
    }

}
```

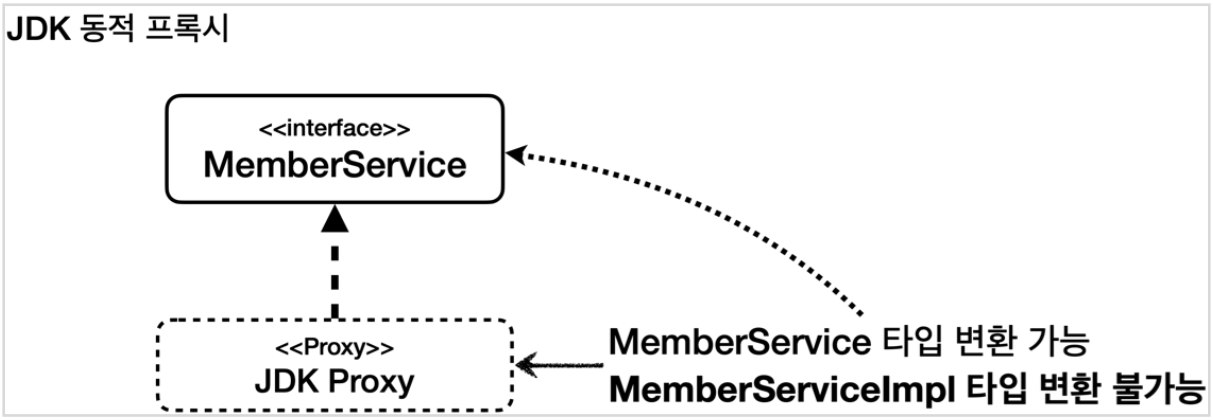
JDK 동적 프록시



jdkProxy() 테스트

여기서는 `MemberServiceImpl` 타입을 기반으로 JDK 동적 프록시를 생성했다. `MemberServiceImpl` 타입은 `MemberService` 인터페이스를 구현한다. 따라서 JDK 동적 프록시는 `MemberService` 인터페이스를 기반으로 프록시를 생성한다. 이 프록시를 `JDK Proxy` 라고 하자. 여기서 `memberServiceProxy` 가 바로 `JDK Proxy` 이다.

JDK 동적 프록시 캐스팅



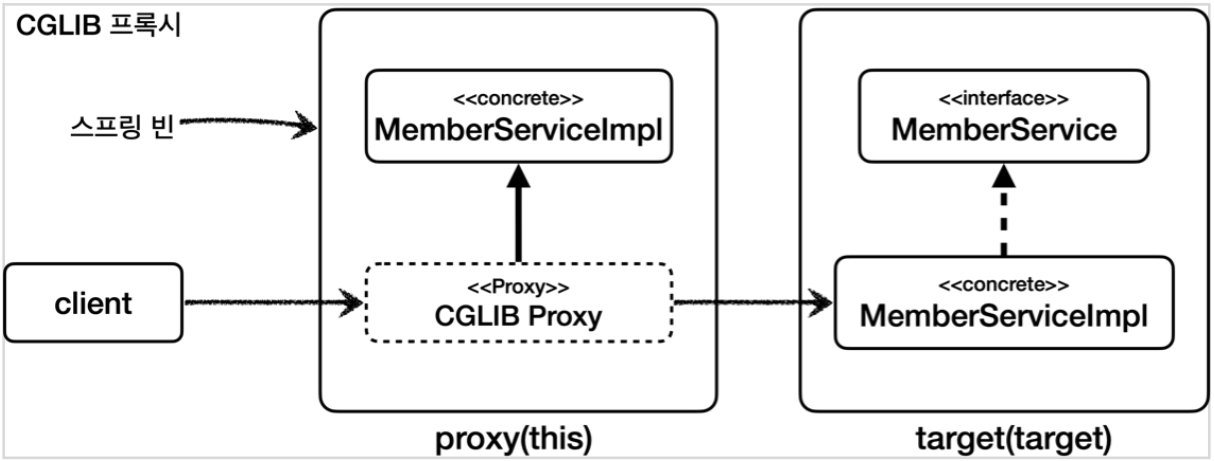
그런데 여기에서 JDK Proxy를 대상 클래스인 `MemberServiceImpl` 타입으로 캐스팅 하려고 하니 예외가 발생한다.
왜냐하면 JDK 동적 프록시는 인터페이스를 기반으로 프록시를 생성하기 때문이다. JDK Proxy는 `MemberService` 인터페이스를 기반으로 생성된 프록시이다. 따라서 JDK Proxy는 `MemberService`로 캐스팅은 가능하지만 `MemberServiceImpl`이 어떤 것인지 전혀 알지 못한다. 따라서 `MemberServiceImpl` 타입으로는 캐스팅이 불가능하다. 캐스팅을 시도하면 `ClassCastException.class` 예외가 발생한다.

이번에는 CGLIB을 사용해보자.

ProxyCastingTest - 추가

```
@Test
void cglibProxy() {
    MemberServiceImpl target = new MemberServiceImpl();
    ProxyFactory proxyFactory = new ProxyFactory(target);
    proxyFactory.setProxyTargetClass(true);//CGLIB 프록시

    //프록시를 인터페이스로 캐스팅 성공
    MemberService memberServiceProxy = (MemberService) proxyFactory.getProxy();

    log.info("proxy class={}", memberServiceProxy.getClass());

    //CGLIB 프록시를 구현 클래스로 캐스팅 시도 성공
    MemberServiceImpl castingMemberService = (MemberServiceImpl)
memberServiceProxy;
}
```

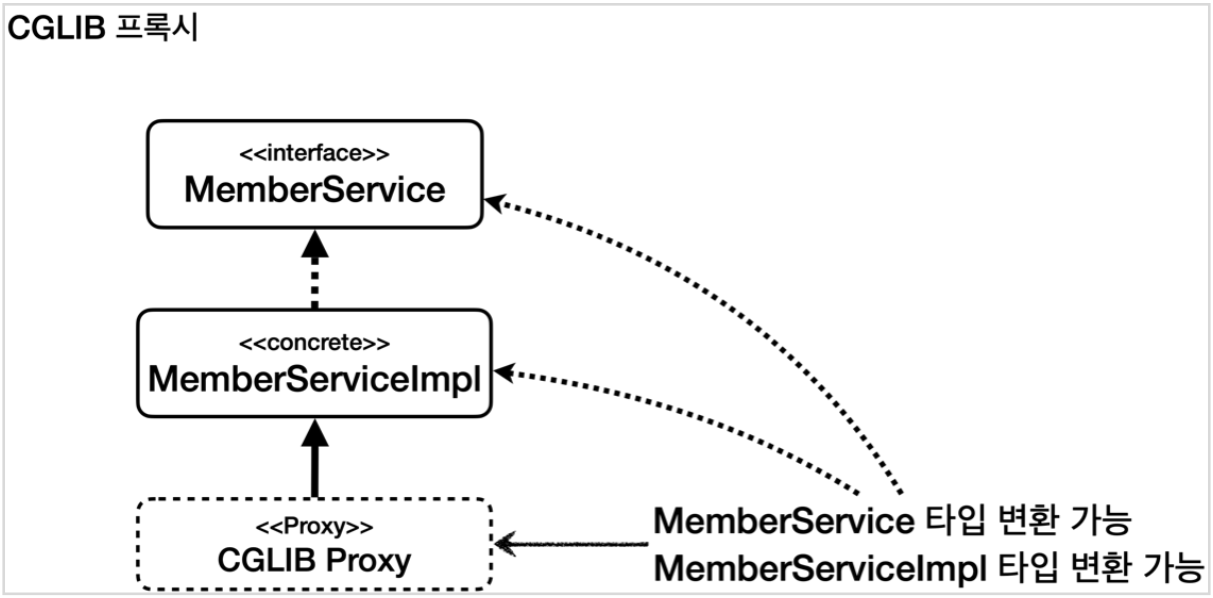
CGLIB 프록시



cglibProxy() 테스트

`MemberServiceImpl` 타입을 기반으로 CGLIB 프록시를 생성했다. `MemberServiceImpl` 타입은 `MemberService` 인터페이스를 구현했다. CGLIB는 구체 클래스를 기반으로 프록시를 생성한다. 따라서 CGLIB는 `MemberServiceImpl` 구체 클래스를 기반으로 프록시를 생성한다. 이 프록시를 CGLIB Proxy라고 하자. 여기서 `memberServiceProxy`가 바로 CGLIB Proxy이다.

CGLIB 프록시 캐스팅



여기에서 CGLIB Proxy를 대상 클래스인 `MemberServiceImpl` 타입으로 캐스팅하면 성공한다.
왜냐하면 CGLIB는 구체 클래스를 기반으로 프록시를 생성하기 때문이다. CGLIB Proxy는 `MemberServiceImpl` 구체 클래스를 기반으로 생성된 프록시이다. 따라서 CGLIB Proxy는 `MemberServiceImpl`은 물론이고, `MemberServiceImpl`이 구현한 인터페이스인 `MemberService`로도 캐스팅 할 수 있다.

정리

JDK 동적 프록시는 대상 객체인 `MemberServiceImpl`로 캐스팅 할 수 없다.

CGLIB 프록시는 대상 객체인 `MemberServiceImpl`로 캐스팅 할 수 있다.

그런데 프록시를 캐스팅 할 일이 많지 않을 것 같은데 왜 이 이야기를 하는 것일까? 진짜 문제는 의존관계 주입시에 발생한다.

프록시 기술과 한계 - 의존관계 주입

JDK 동적 프록시를 사용하면서 의존관계 주입을 할 때 어떤 문제가 발생하는지 코드로 알아보자.

ProxyDIAspect
주의: 테스트 코드(src/test)에 위치한다.

```
package hello.aop.proxyvs.code;

import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Slf4j
@Aspect
public class ProxyDIAspect {

    @Before("execution(* hello.aop..*.*(..))")
    public void doTrace(JoinPoint joinPoint) {
        log.info("[proxyDIAdvice] {}", joinPoint.getSignature());
    }
}
```

AOP 프록시 생성을 위해 간단한 `Aspect`를 만들자.

ProxyDITest

```
package hello.aop.proxyvs;
```

```java
import hello.aop.member.MemberService;
import hello.aop.member.MemberServiceImpl;
import hello.aop.proxyvs.code.ProxyDIAspect;
import lombok.extern.slf4j.Slf4j;
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
@SpringBootTest(properties = {"spring.aop.proxy-target-class=false"}) //JDK 동적 프록시, DI 예외 발생
//@SpringBootTest(properties = {"spring.aop.proxy-target-class=true"}) //CGLIB 프록시, 성공
@Import(ProxyDIAspect.class)
public class ProxyDITest {

    @Autowired MemberService memberService; //JDK 동적 프록시 OK, CGLIB OK
    @Autowired MemberServiceImpl memberServiceImpl; //JDK 동적 프록시 X, CGLIB OK

    @Test
    void go() {
        log.info("memberService class={}", memberService.getClass());
        log.info("memberServiceImpl class={}", memberServiceImpl.getClass());
        memberServiceImpl.hello("hello");
    }

}
```

코드 설명

- `@SpringBootTest` : 내부에 컴포넌트 스캔을 포함하고 있다. `MemberServiceImpl` 에 `@Component` 가 붙어있으므로 스프링 빈 등록 대상이 된다.
- `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.
- `@Import(ProxyDIAspect.class)` : 앞서 만든 Aspect를 스프링 빈으로 등록한다.

JDK 동적 프록시에 구체 클래스 타입 주입

JDK 동적 프록시에 구체 클래스 타입을 주입할 때 어떤 문제가 발생하는지 지금부터 확인해보자.

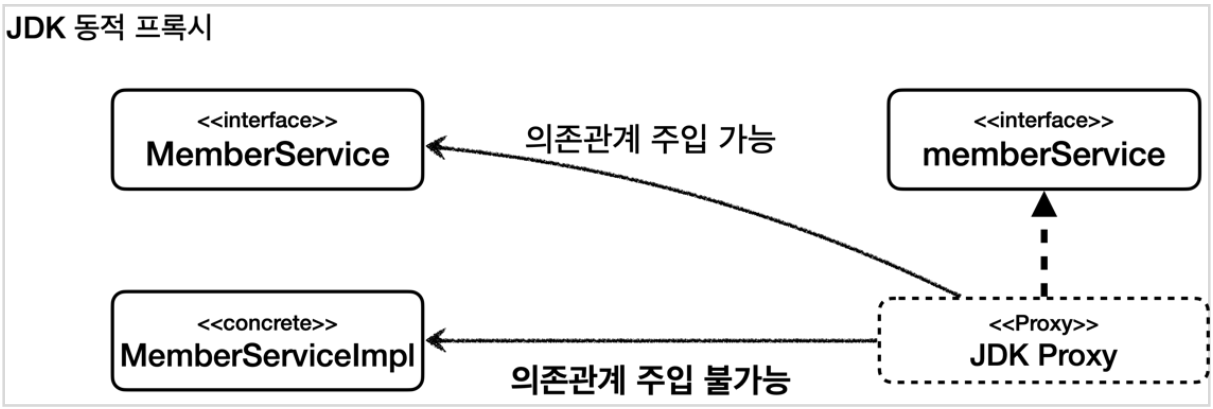
실행

먼저 `spring.aop.proxy-target-class=false` 설정을 사용해서 스프링 AOP가 JDK 동적 프록시를 사용하도록 했다. 이렇게 실행하면 다음과 같이 오류가 발생한다.

실행 결과

```
BeanNotOfRequiredTypeException: Bean named 'memberServiceImpl' is expected to
be of type 'hello.aop.member.MemberServiceImpl' but was actually of type
'com.sun.proxy.$Proxy54'
```

타입과 관련된 예외가 발생한다. 자세히 읽어보면 `memberServiceImpl`에 주입되길 기대하는 타입은 `hello.aop.member.MemberServiceImpl`이지만 실제 넘어온 타입은 `com.sun.proxy.$Proxy54`이다. 따라서 타입 예외가 발생한다고 한다.



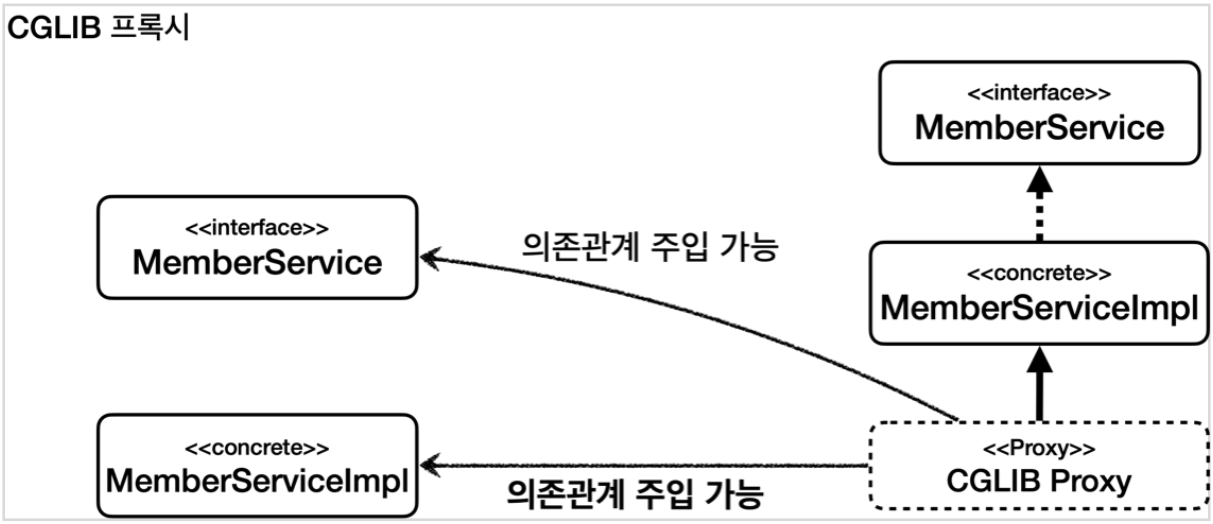
- `@Autowired MemberService memberService` : 이 부분은 문제가 없다. JDK Proxy는 `MemberService` 인터페이스를 기반으로 만들어진다. 따라서 해당 타입으로 캐스팅 할 수 있다.
  - `MemberService = JDK Proxy`가 성립한다.
- `@Autowired MemberServiceImpl memberServiceImpl` : 문제는 여기다. JDK Proxy는 `MemberService` 인터페이스를 기반으로 만들어진다. 따라서 `MemberServiceImpl` 타입이 뭔지 전혀 모른다. 그래서 해당 타입에 주입할 수 없다.
  - `MemberServiceImpl = JDK Proxy`가 성립하지 않는다.

CGLIB 프록시에 구체 클래스 타입 주입

이번에는 JDK 동적 프록시 대신에 CGLIB를 사용해서 프록시를 적용해보자. 다음과 같이 주석을 반대로 걸어보자.

```
//@SpringBootTest(properties = {"spring.aop.proxy-target-class=false"}) //JDK 동적 프록시, DI 예외 발생
@SpringBootTest(properties = {"spring.aop.proxy-target-class=true"}) //CGLIB 프록시, 성공
```

실행해보면 정상 동작하는 것을 확인할 수 있다.



- `@Autowired MemberService memberService` : CGLIB Proxy는 `MemberServiceImpl` 구체 클래스를 기반으로 만들어진다. `MemberServiceImpl`은 `MemberService` 인터페이스를 구현했기 때문에 해당 타입으로 캐스팅 할 수 있다.
  - `MemberService = CGLIB Proxy`가 성립한다.
- `@Autowired MemberServiceImpl memberServiceImpl` : CGLIB Proxy는 `MemberServiceImpl` 구체 클래스를 기반으로 만들어진다. 따라서 해당 타입으로 캐스팅 할 수 있다.
  - `MemberServiceImpl = CGLIB Proxy`가 성립한다.

정리
JDK 동적 프록시는 대상 객체인 `MemberServiceImpl` 타입에 의존관계를 주입할 수 없다.
CGLIB 프록시는 대상 객체인 `MemberServiceImpl` 타입에 의존관계 주입을 할 수 있다.

지금까지 JDK 동적 프록시가 가지는 한계점을 알아보았다. 실제로 개발할 때는 인터페이스가 있으면 인터페이스를 기반으로 의존관계 주입을 받는 것이 맞다.
DI의 장점이 무엇인가? DI 받는 클라이언트 코드의 변경 없이 구현 클래스를 변경할 수 있는 것이다. 이렇게

하려면 인터페이스를 기반으로 의존관계를 주입 받아야 한다. `MemberServiceImpl` 타입으로 의존관계 주입을 받는 것 처럼 구현 클래스에 의존관계를 주입하면 향후 구현 클래스를 변경할 때 의존관계 주입을 받는 클라이언트의 코드도 함께 변경해야 한다.
따라서 올바르게 잘 설계된 애플리케이션이라면 이런 문제가 자주 발생하지는 않는다.
그럼에도 불구하고 테스트, 또는 여러가지 이유로 AOP 프록시가 적용된 구체 클래스를 직접 의존관계 주입 받아야 하는 경우가 있을 수 있다. 이때는 CGLIB를 통해 구체 클래스 기반으로 AOP 프록시를 적용하면 된다.

여기까지 듣고보면 CGLIB를 사용하는 것이 좋아보인다. CGLIB를 사용하면 사실 이런 고민 자체를 하지 않아도 된다. 다음 시간에는 CGLIB의 단점을 알아보자.

프록시 기술과 한계 - CGLIB

스프링에서 CGLIB는 구체 클래스를 상속 받아서 AOP 프록시를 생성할 때 사용한다.
CGLIB는 구체 클래스를 상속 받기 때문에 다음과 같은 문제가 있다.

CGLIB 구체 클래스 기반 프록시 문제점

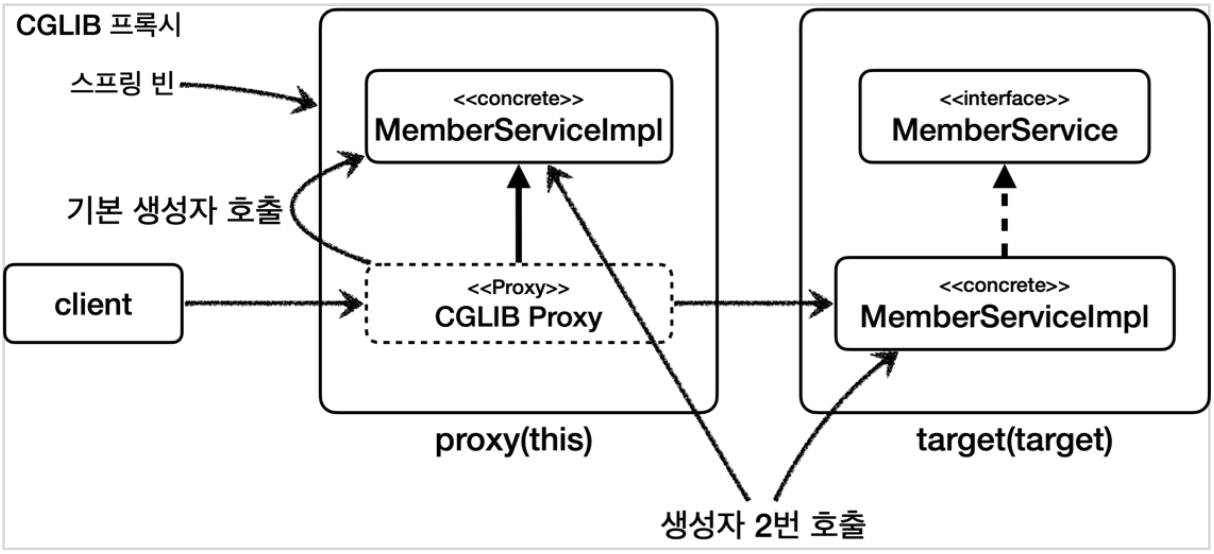
- 대상 클래스에 기본 생성자 필수
- 생성자 2번 호출 문제
- final 키워드 클래스, 메서드 사용 불가

하나씩 자세히 알아보자.

대상 클래스에 기본 생성자 필수
CGLIB는 구체 클래스를 상속 받는다. 자바 언어에서 상속을 받으면 자식 클래스의 생성자를 호출할 때 자식 클래스의 생성자에서 부모 클래스의 생성자도 호출해야 한다. (이 부분이 생략되어 있다면 자식 클래스의 생성자 첫줄에 부모 클래스의 기본 생성자를 호출하는 `super()` 가 자동으로 들어간다.) 이 부분은 자바 문법 규약이다.
CGLIB를 사용할 때 CGLIB가 만드는 프록시의 생성자는 우리가 호출하는 것이 아니다. CGLIB 프록시는 대상 클래스를 상속 받고, 생성자에서 대상 클래스의 기본 생성자를 호출한다. 따라서 대상 클래스에 기본 생성자를 만들어야 한다. (기본 생성자는 파라미터가 하나도 없는 생성자를 뜻한다. 생성자가 하나도 없으면 자동으로 만들어진다.)

생성자 2번 호출 문제
CGLIB는 구체 클래스를 상속 받는다. 자바 언어에서 상속을 받으면 자식 클래스의 생성자를 호출할 때 부모 클래스의 생성자도 호출해야 한다. 그런데 왜 2번일까?

- 1. 실제 target의 객체를 생성할 때
- 2. 프록시 객체를 생성할 때 부모 클래스의 생성자 호출



final 키워드 클래스, 메서드 사용 불가
final 키워드가 클래스에 있으면 상속이 불가능하고, 메서드에 있으면 오버라이딩이 불가능하다. CGLIB는 상속을 기반으로 하기 때문에 두 경우 프록시가 생성되지 않거나 정상 동작하지 않는다.

프레임워크 같은 개발이 아니라 일반적인 웹 애플리케이션을 개발할 때는 `final` 키워드를 잘 사용하지 않는다. 따라서 이 부분이 특별히 문제가 되지는 않는다.

정리
JDK 동적 프록시는 대상 클래스 타입으로 주입할 때 문제가 있고, CGLIB는 대상 클래스에 기본 생성자 필수, 생성자 2번 호출 문제가 있다.

그렇다면 스프링은 어떤 방법을 권장할까?

프록시 기술과 한계 - 스프링의 해결책

스프링은 AOP 프록시 생성을 편리하게 제공하기 위해 오랜 시간 고민하고 문제들을 해결해왔다.

스프링의 기술 선택 변화

스프링 3.2, CGLIB를 스프링 내부에 함께 패키징

CGLIB를 사용하려면 CGLIB 라이브러리가 별도로 필요했다. 스프링은 CGLIB 라이브러리를 스프링 내부에 함께 패키징해서 별도의 라이브러리 추가 없이 CGLIB를 사용할 수 있게 되었다. `CGLIB spring-core org.springframework`

CGLIB 기본 생성자 필수 문제 해결
스프링 4.0부터 CGLIB의 기본 생성자가 필수인 문제가 해결되었다.
`objenesis`라는 특별한 라이브러리를 사용해서 기본 생성자 없이 객체 생성이 가능하다.
참고로 이 라이브러리는 생성자 호출 없이 객체를 생성할 수 있게 해준다.

생성자 2번 호출 문제
스프링 4.0부터 CGLIB의 생성자 2번 호출 문제가 해결되었다.
이것도 역시 `objenesis`라는 특별한 라이브러리 덕분에 가능해졌다.
이제 생성자가 1번만 호출된다.

스프링 부트 2.0 - CGLIB 기본 사용
스프링 부트 2.0 버전부터 CGLIB를 기본으로 사용하도록 했다.
이렇게 해서 구체 클래스 타입으로 의존관계를 주입하는 문제를 해결했다.
스프링 부트는 별도의 설정이 없다면 AOP를 적용할 때 기본적으로 `proxyTargetClass=true`로 설정해서 사용한다.
따라서 인터페이스가 있어도 JDK 동적 프록시를 사용하는 것이 아니라 항상 CGLIB를 사용해서 구체클래스를 기반으로 프록시를 생성한다.
물론 스프링은 우리에게 선택권을 열어주기 때문에 다음과 같이 설정하면 JDK 동적 프록시도 사용할 수 있다.

`application.properties`

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

정리
스프링은 최종적으로 스프링 부트 2.0에서 CGLIB를 기본으로 사용하도록 결정했다. CGLIB를 사용하면 JDK 동적 프록시에서 동작하지 않는 구체 클래스 주입이 가능하다. 여기에 추가로 CGLIB의 단점들이 이제는 많이 해결되었다. CGLIB의 남은 문제라면 `final` 클래스나 `final` 메서드가 있는데, AOP를 적용할 대상에는 `final` 클래스나 `final` 메서드를 잘 사용하지는 않으므로 이 부분은 크게 문제가 되지는 않는다.

개발자 입장에서 보면 사실 어떤 프록시 기술을 사용하든 상관이 없다. JDK 동적 프록시든 CGLIB든 또는 어떤 새로운 프록시 기술을 사용해도 된다. 심지어 클라이언트 입장에서 어떤 프록시 기술을 사용하는지 모르고 잘 동작하는 것이 가장 좋다. 단지 문제 없고, 개발하기에 편리하면 되는 것이다.

마지막으로 `ProxyDITest`를 다음과 같이 변경해서 아무런 설정 없이 실행해보면 CGLIB가 기본으로 사용되는 것을 확인할 수 있다.

```
@Slf4j
//@SpringBootTest(properties = {"spring.aop.proxy-target-class=false"}) //JDK 동적 프록시, DI 예외 발생
//@SpringBootTest(properties = {"spring.aop.proxy-target-class=true"}) //CGLIB 프록시, 성공
@SpringBootTest //추가
@Import(ProxyDIAspect.class)
public class ProxyDITest {...}
```

`@SpringBootTest` 이 부분을 추가해야 한다. `application.properties`에 `spring.aop.proxy-target-class` 관련 설정이 없어야 한다.

실행 결과

```
memberService class=class hello.aop.member.MemberServiceImpl$$EnhancerBySpringCGLIB$$83e257b3
memberServiceImpl class=class hello.aop.member.MemberServiceImpl$$EnhancerBySpringCGLIB$$83e257b3
```

정리