프록시

목차

- 프록시 패턴
  - 인터페이스 기반 프록시
  - 상속 기반 프록시
- 동적 프록시
  - JDK동적 프록시
  - CGLIB 프록시
- 프록시 팩토리

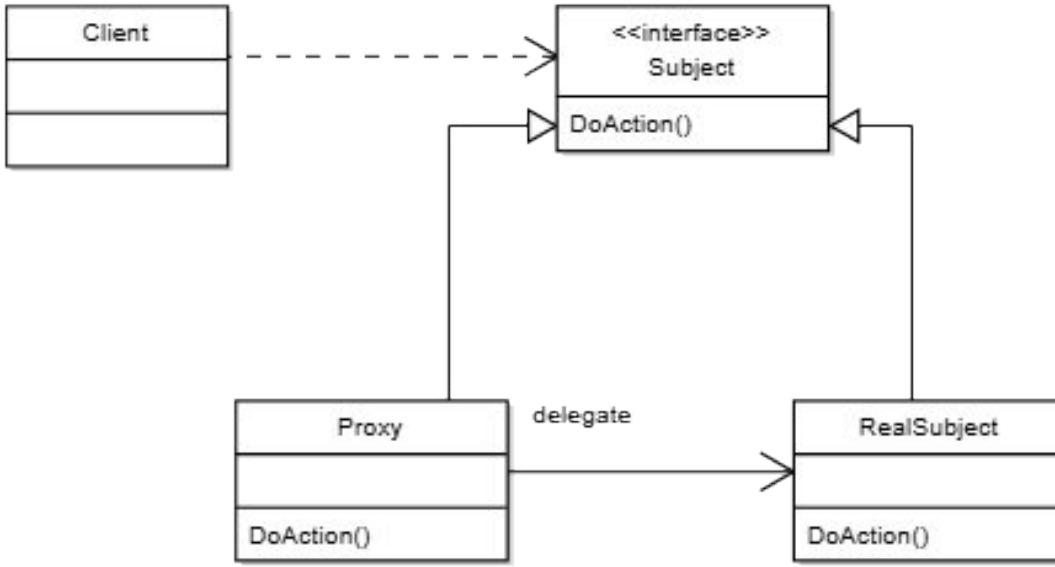
프록시 패턴

프록시

- 프록시의 사전적 정의는 대리자라는 뜻입니다.
- 소프트웨어에서 프록시는 클라이언트의 요청을 대신 받아 처리하고, 필요한 경우 실제 서버에 요청을 위임하는 대리 객체입니다.
- 클라이언트는 서버가 요청을 처리한 것인지 프록시가 요청을 처리한 것인지 알지 못합니다.
- 스프링에서는 어떻게 프록시를 사용하고 있을까요? 이를 알기전에 먼저 프록시패턴과 동적 프록시를 알아야합니다.

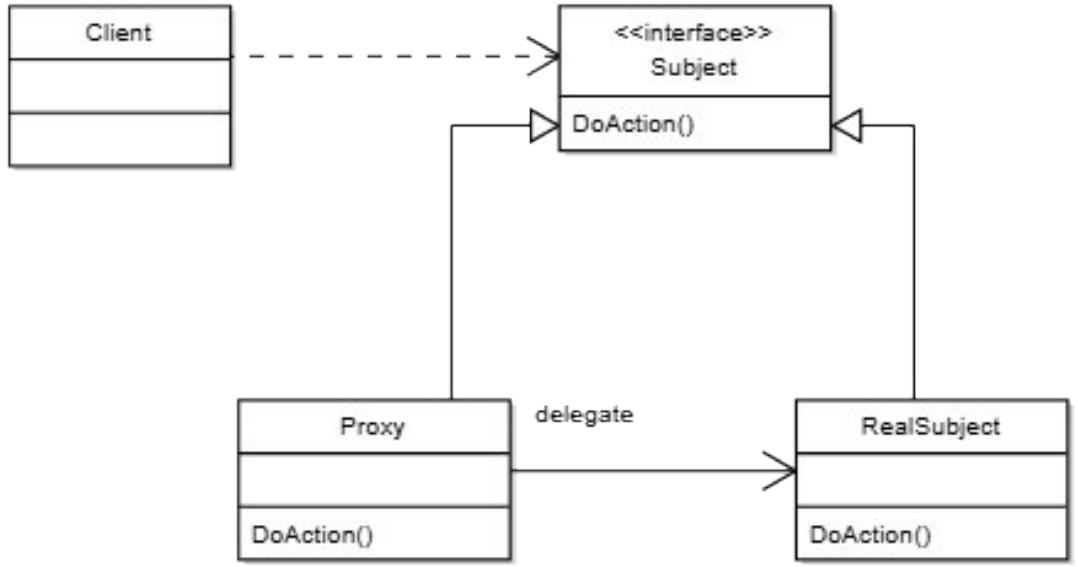
인터페이스 기반 프록시 패턴

1.Subject (인터페이스) 클라이언트가 사용하는 공통 인터페이스입니다. RealSubject와 Proxy 모두 이 인터페이스를 구현합니다.
2.RealSubject (실제 객체) 실제 비즈니스 로직을 수행하는 클래스입니다.
3.Proxy (대리자) RealSubject에 대한 참조를 가지고 있으며, 클라이언트의 요청을 가로채서 추가 작업을 수행한 후 RealSubject에 위임합니다.



상속 기반 프록시 패턴

1.인터페이스를 통해서 프록시 패턴을 설명했지만, **RealSubject를 Proxy가 상속받게 하여 프록시 패턴을 상속을 활용하여 만들 수도 있습니다.**



프록시 장점

프록시를 사용하면 좋은점이 직접 서버를 호출하는것과 다르게 프록시를 사용하여 서버를 호출하면 추가적인 여러가지 일을 할 수 있다는 점입니다.

- **접근제어 및 캐싱** 어떤 데이터 조회 작업이 반복적으로 발생하지만 결과가 자주 바뀌지 않는다면, 프록시가 이전 결과를 캐싱해두고 동일한 요청에 대해 빠르게 응답할 수 있습니다.

프록시 장점

- **프록시 체인** 하나의 실제 객체(RealSubject)를 감싸는 여러 개의 프록시들이 계층적으로 중첩되어 클라이언트 요청을 처리하는 구조를 말합니다.

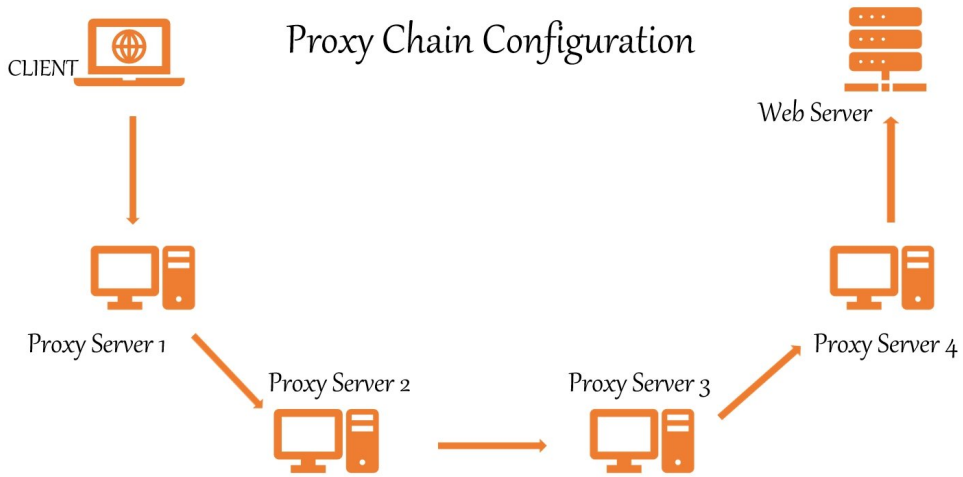
- 예를 들면 Client → LoggingProxy → CachingProxy → TransactionProxy → RealService 순으로 요청을 전달할 수 있습니다.

  LoggingProxy: 요청이 들어왔다는 로그 출력

  CachingProxy: 결과가 캐시에 있는지 확인

  TransactionProxy: 트랜잭션 시작

  RealService: 실제 비즈니스 로직 실행



- 부가기능을 프록시별로 나누어 개발할 수 있습니다.

동적 프록시

동적 프록시

- 정적 프록시는 직접 코드를 작성해서 프록시 클래스를 만들어야 하지만, 동적 프록시는 런타임에 자동으로 프록시 객체를 생성해줍니다.

- 정적 프록시를 사용하면 프록시 적용 대상만큼 프록시 클래스를 만들어야합니다. 100개가 적용 대상이라면 프록시도 100개를 만들어야한다는 것입니다. 그래서 이러한 문제를 해결하기 위해 동적 프록시를 사용합니다.

- 동적 프록시의 종류는 JDK동적 프록시와 CGLIB 프록시로 나누어집니다.

JDK동적 프록시

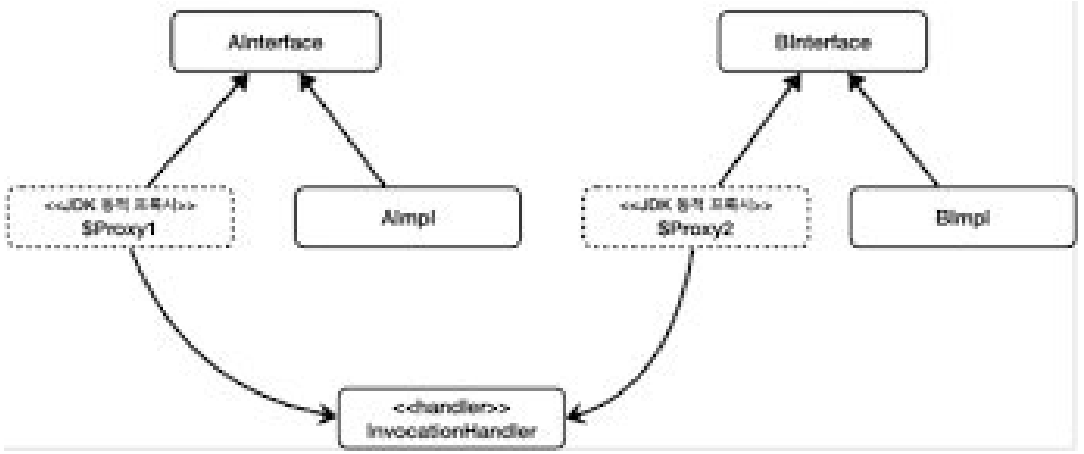
- JDK 동적 프록시는 인터페이스 기반의 프록시 객체를 런타임에 생성하는 방식입니다. 자바의 java.lang.reflect.Proxy 클래스를 활용하여 동작하며, 다음과 같은 특징이 있습니다

1. 인터페이스를 반드시 구현한 클래스만 프록시 생성 가능

2. 런타임 시 자동으로 프록시 클래스가 생성됨

3. InvocationHandler 인터페이스를 통해 공통 로직을 삽입

동적 프록시가 무엇인지 느낌이 잘 오지 않을테니 코드로 살펴보겠습니다.



JDK Dynamic proxy 코드

Subject1, Subject2: Client에서 사용할 인터페이스

```
public interface Subject1 {
    void call();
}
```

```
public interface Subject2 {
    void call();
}
```

RealSubject1, RealSubject2: 프록시의 적용대상

```
@Slf4j
public class RealSubject1 implements Subject1 {

    @Override
    public void call() {
        log.info("서버1 시작");
        log.info("서버1 끝");
    }
}
```

```
@Slf4j
public class RealSubject2 implements Subject2 {

    @Override
    public void call() {
        log.info("서버2 시작");
        log.info("서버2 끝");
    }
}
```

JDK Dynamic proxy 코드

DynamicProxyHandler: 동적 프록시를 만들어주는 핸들러

```java
@Slf4j
public class DynamicProxyHandler implements InvocationHandler {

    private Object target;
    public DynamicProxyHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        log.info("동적 프록시 실행");
        Object result = method.invoke(target, args);
        log.info("동적 프록시 종료");
        return result;
    }
}
```

- 여러 프록시 적용대상의 프록시를 만들어 주는 역할을 합니다.

invoke() :
Object proxy : 프록시 자신 Method method :호출한 메서드 Object[] args : 메서드를 호출할 때 전달한 인수

JDK Dynamic proxy 코드

테스트 코드

```
@Slf4j
public class JdkDynamicTest {

    @Test
    public void dynamicTest1() {
        Subject1 subject1 = new RealSubject1();
        DynamicProxyHandler handler1 = new DynamicProxyHandler(subject1);
        Subject1 proxy1 = (Subject1) Proxy.newProxyInstance(subject1.getClass().getClassL
oader(), new Class[]{Subject1.class}, handler1);
        proxy1.call();
        log.info("subject1 = {}", subject1.getClass());
        log.info("proxy1 = {}", proxy1.getClass());


        Subject2 subject2 = new RealSubject2();
        DynamicProxyHandler handler2 = new DynamicProxyHandler(subject2);
        Subject2 proxy2 = (Subject2) Proxy.newProxyInstance(subject2.getClass().getClassL
oader(), new Class[]{Subject2.class}, handler2);
        proxy2.call();
        log.info("subject2 = {}", subject2.getClass());
        log.info("proxy1 = {}", proxy2.getClass());
    }
}
```

- jdk동적 프록시는 Proxy.newProxyInstance()메서드를 통해서 생성할 수 있습니다.

- 첫 번째 인자는 클래스 로더,
- 두 번째는 인터페이스 배열,
- 세 번째는 호출 핸들러입니다.

- 이들을 기반으로 해당 인터페이스를 구현하는 프록시 객체가 생성됩니다.

JDK Dynamic proxy 코드

테스트 코드

```
@Slf4j
public class JdkDynamicTest {

    @Test
    public void dynamicTest1() {
        Subject1 subject1 = new RealSubject1();
        DynamicProxyHandler handler1 = new DynamicProxyHandler(subject1);
        Subject1 proxy1 = (Subject1) Proxy.newProxyInstance(subject1.getClass().getClassL
oader(), new Class[]{Subject1.class}, handler1);
        proxy1.call();
        log.info("subject1 = {}", subject1.getClass());
        log.info("proxy1 = {}", proxy1.getClass());


        Subject2 subject2 = new RealSubject2();
        DynamicProxyHandler handler2 = new DynamicProxyHandler(subject2);
        Subject2 proxy2 = (Subject2) Proxy.newProxyInstance(subject2.getClass().getClassL
oader(), new Class[]{Subject2.class}, handler2);
        proxy2.call();
        log.info("subject2 = {}", subject2.getClass());
        log.info("proxy1 = {}", proxy2.getClass());
    }
}
```

- jdk동적 프록시는 Proxy.newProxyInstance()메서드를 통해서 생성할 수 있습니다.

- 첫 번째 인자는 클래스 로더,
- 두 번째는 인터페이스 배열,
- 세 번째는 호출 핸들러입니다.

- 이들을 기반으로 해당 인터페이스를 구현하는 프록시 객체가 생성됩니다.

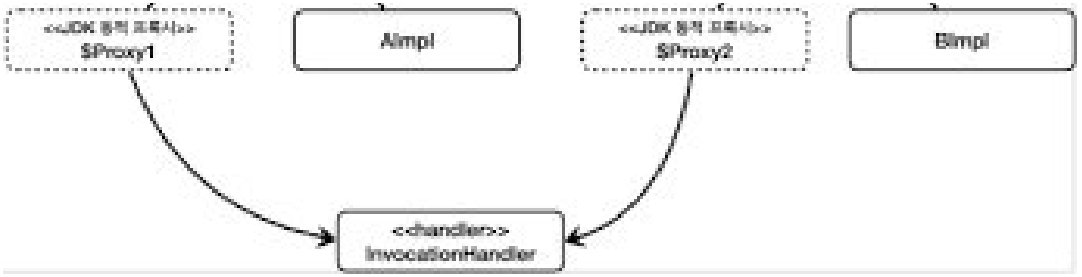
JDK Dynamic proxy 코드

✔ 테스트 통과: 1 /1개 테스트 – 12ms

```
nple.proxy_practice.jdk_dynamic_proxy.DynamicProxyHandler -- 동적 프록시 실행
nple.proxy_practice.jdk_dynamic_proxy.code1.RealSubject1 -- 서버1 시작
nple.proxy_practice.jdk_dynamic_proxy.code1.RealSubject1 -- 서버1 끝
nple.proxy_practice.jdk_dynamic_proxy.DynamicProxyHandler -- 동적 프록시 종료
nple.proxy_practice.jdk_dynamic_proxy.JdkDynamicTest -- subject1 = class com.example.proxy_practice.jdk_dynamic_proxy.code1.RealSubject1
nple.proxy_practice.jdk_dynamic_proxy.JdkDynamicTest -- proxy1 = class jdk.proxy3.$Proxy12
nple.proxy_practice.jdk_dynamic_proxy.DynamicProxyHandler -- 동적 프록시 실행
nple.proxy_practice.jdk_dynamic_proxy.code2.RealSubject2 -- 서버2 시작
nple.proxy_practice.jdk_dynamic_proxy.code2.RealSubject2 -- 서버2 끝
nple.proxy_practice.jdk_dynamic_proxy.DynamicProxyHandler -- 동적 프록시 종료
nple.proxy_practice.jdk_dynamic_proxy.JdkDynamicTest -- subject2 = class com.example.proxy_practice.jdk_dynamic_proxy.code2.RealSubject2
nple.proxy_practice.jdk_dynamic_proxy.JdkDynamicTest -- proxy1 = class jdk.proxy3.$Proxy13
```

CGLIB 프록시

- 앞에서 프록시 패턴에서 프록시는 상속을 통해서도 생성가능하다고 설명드렸습니다.
- 하지만 JDK동적 프록시는 인터페이스를 통한 프록시만 생성할 수 있고, 상속을 통한 프록시는 생성하지 못합니다. 이는 CGLIB을 활용하여 해결가능합니다.
- CGLIB은 구체 클래스의 서브클래스를 런타임에 생성하므로, 인터페이스가 없어도 프록시 객체를 만들 수 있습니다.



CGLIB 프록시 코드

RealSubject1, RealSubject2: 프록시의 적용대상

```java
@Slf4j
public class RealSubject1{

    @Override
    public void call() {
        log.info("서버1 시작");
        log.info("서버1 끝");
    }
}
```

```java
@Slf4j
public class RealSubject2{

    @Override
    public void call() {
        log.info("서버2 시작");
        log.info("서버2 끝");
    }
}
```

CglibProxyHandler: 동적 프록시를 만들어주는 핸들러

```java
@Slf4j
public class CglibProxyHandler implements MethodInterceptor {
    private Object target;
    public CglibProxyHandler(Object target) {
        this.target = target;
    }
    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy)
throws Throwable {
        log.info("동적 프록시 실행");
        Object result = proxy.invoke(target, args);
        log.info("동적 프록시 종료");
        return result;
    }
}
```

intercept()메서드:
Object proxy : 프록시 자신
Method method : 호출한 메서드
Object[] args : 메서드를 호출할 때 전달한 인수
proxy : 메서드 호출에 사용

CGLIB 프록시 코드

테스트 코드

```java
@Slf4j
public class CglibTest {
    @Test
    public void cglibTest() {
        RealSubject1 subject1 = new RealSubject1();
        Enhancer enhancer1 = new Enhancer();
        enhancer1.setSuperclass(RealSubject1.class);
        enhancer1.setCallback(new CglibProxyHandler(subject1));
        RealSubject1 proxy1 = (RealSubject1) enhancer1.create();
        proxy1.call();

        log.info("subject1 = {}", subject1.getClass());
        log.info("proxy1 = {}", proxy1.getClass());


        RealSubject2 subject2 = new RealSubject2();
        Enhancer enhancer2 = new Enhancer();
        enhancer2.setSuperclass(RealSubject2.class);
        enhancer2.setCallback(new CglibProxyHandler(subject2));
        RealSubject2 proxy2 = (RealSubject2) enhancer2.create();
        proxy2.call();

        log.info("subject2 = {}", subject2.getClass());
        log.info("proxy2 = {}", proxy2.getClass());
    }
}
```

- CGLIB는 Enhancer를 사용해서 프록시를 생성합니다.
- enhancer.setSuperclass()를 사용해 CGLIB는 클래스를 상속 받아서 프록시를 생성할 수 있습니다.
- enhancer.setCallback()를 사용해 프록시에 적용할 실행 로직을 할당합니다.
- enhancer.create()를 사용해 프록시를 만듭니다.

CGLIB 프록시 코드

✓ 테스트 통과: 1 /1개 테스트 – 44ms

```
ractice.dynamic_proxy.CglibProxyHandler -- 동적 프록시 실행
ractice.dynamic_proxy.code1.RealSubject1 -- 서버1 시작
ractice.dynamic_proxy.code1.RealSubject1 -- 서버1 끝
ractice.dynamic_proxy.CglibProxyHandler -- 동적 프록시 종료
ractice.dynamic_proxy.CglibTest -- subject1 = class com.example.proxy_practice.dynamic_proxy.code1.RealSubject1
ractice.dynamic_proxy.CglibTest -- proxy1 = class com.example.proxy_practice.dynamic_proxy.code1.RealSubject1$$EnhancerByCGLIB$$608175a5
ractice.dynamic_proxy.CglibProxyHandler -- 동적 프록시 실행
ractice.dynamic_proxy.code2.RealSubject2 -- 서버2 시작
ractice.dynamic_proxy.code2.RealSubject2 -- 서버2 끝
ractice.dynamic_proxy.CglibProxyHandler -- 동적 프록시 종료
ractice.dynamic_proxy.CglibTest -- subject2 = class com.example.proxy_practice.dynamic_proxy.code2.RealSubject2
ractice.dynamic_proxy.CglibTest -- proxy2 = class com.example.proxy_practice.dynamic_proxy.code2.RealSubject2$$EnhancerByCGLIB$$7d098745
```

정리

- 프록시의 기본 개념과 프록시 패턴, 그리고 자바에서 제공하는 두 가지 동적 프록시 방식인 **JDK 동적 프록시**와 **CGLIB 프록시**에 대해 알아보았습니다.
- **JDK 동적 프록시**는 인터페이스를 기반으로 하기 때문에, 인터페이스가 반드시 있어야 합니다. 반면 **CGLIB 프록시**는 클래스 자체를 상속하여 프록시를 만들 수 있으므로, 인터페이스가 없는 경우에도 사용 가능합니다.

구분	JDK Dynamic Proxy	CGLIB Proxy
기반	인터페이스 기반	클래스 상속 기반
사용 조건	인터페이스 필요	인터페이스 없어도 가능
생성 방식	Proxy.newProxyInstance()	Enhancer.create()
사용 예	스프링 AOP (인터페이스 기반)	스프링 AOP (클래스 기반)

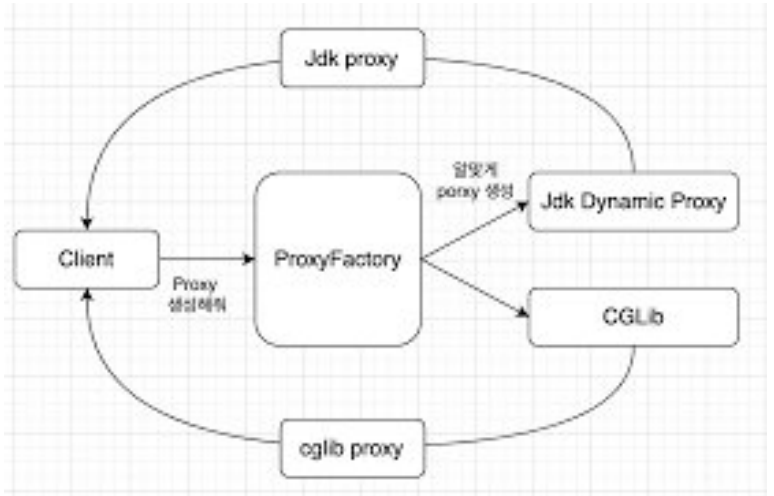
프록시 팩토리

프록시 팩토리

- 동적 프록시는 JDK 동적 프록시와 CGLIB 프록시 두 가지 방식으로 생성됩니다.

- **JDK 동적 프록시는 인터페이스 기반**으로 동작합니다. **CGLIB 프록시는 클래스 상속 기반**으로 동작합니다.
- 그런데, 인터페이스 기반과 상속 기반을 혼용해서 사용해야 할 때는 어떻게 해야 할까요? 이럴 때마다 JDK와 CGLIB 프록시를 각각 따로 구현해야 할까요? 그렇지 않습니다.

- 스프링은 이런 상황을 위해 **JDK와 CGLIB을 자동으로 선택해서 프록시를 만들어주는 ProxyFactory를 제공**합니다.



프록시 팩토리 코드

Subject: JDK 동적 프록시를 위한 인터페이스

```
public interface Subject {
    void call();
    void sayHello();
}
```

RealSubject: Subject 인터페이스를 구현한 클래스 (JDK 프록시 대상)

```
@Slf4j
public class RealSubject implements Subject {

    @Override
    public void call() {
        log.info("서버 시작");
        log.info("서버 끝");
    }

    @Override
    public void sayHello() {
        log.info("Hello!");
    }
}
```

ConcreteSubject: 인터페이스 없이 동작하는 클래스 (CGLIB 프록시 대상)

```
@Slf4j
public class ConcreteSubject{

    public void call() {
        log.info("서버 시작");
        log.info("서버 끝");
    }

    public void sayHello() {
        log.info("Hello!");
    }
}
```

프록시 팩토리 코드

AdviceImpl: ProxyFactory에 석용알 Advice

```
@Slf4j
public class AdviceImpl implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        log.info("proxy 실행");
        Object result = invocation.proceed();
        log.info("proxy 종료");
        return result;
    }
}
```

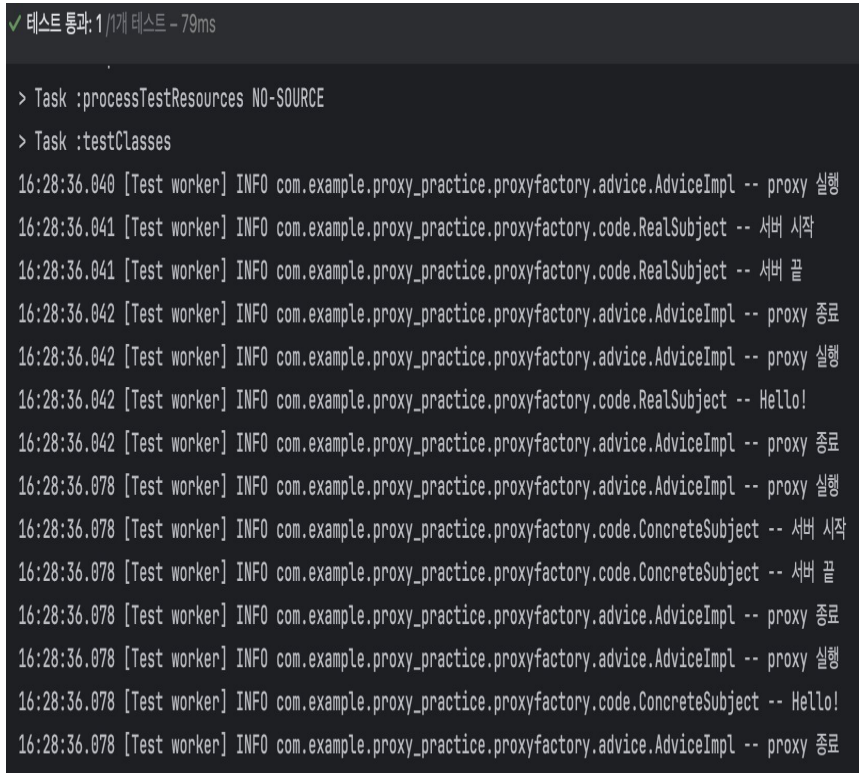
- 이제 Subject 인터페이스 + RealSubject, 그리고 ConcreteSubject를 이용해 ProxyFactory로 각각 JDK와 CGLIB 프록시를 자동 생성할 수 있습니다.
- 여기서 Advice란 무엇일까요? AdviceImpl 클래스는 MethodInterceptor를 구현하고 있죠. 이러한 클래스는 스프링 AOP에서 'Advice' 라고 불립니다.
- Advice란? Advice는 **공통 기능(예: 로깅, 트랜잭션, 보안 검사 등)을 핵심 로직과 분리해서 적용**할 수 있게 해주는 코드 조각입니다. 즉, 핵심 기능을 감싸서 실행 전/후에 부가 로직을 추가할 수 있는 것이죠.
- 또한 **ProxyFactory에서 프록시를 만들 때 Advice를 사용하여 공통 기능을 추가**할 수 있습니다.

프록시 팩토리 코드

테스트 코드

```java
public class ProxyFactoryTest {

    @Test
    public void proxyFactoryTest() {
        // --------------------------------
        // 1. JDK 동적 프록시 예제
        // --------------------------------

        // 프록시 대상: 인터페이스 기반 객체
        Subject subject = new RealSubject();

        // ProxyFactory 생성 (JDK 프록시 방식이 자동 선택됨)
        ProxyFactory proxyFactory = new ProxyFactory(subject);

        // 공통 기능(Advice) 추가
        proxyFactory.addAdvice(new AdviceImpl());

        // 프록시 생성
        Subject proxy = (Subject) proxyFactory.getProxy();

        // 프록시 메서드 호출 (call과 sayHello 모두 Advice 적용됨)
        proxy.call();       // → Advice + 실제 메서드 실행
        proxy.sayHello();   // → Advice + 실제 메서드 실행

        // --------------------------------
        // 2. CGLIB 프록시 예제
        // --------------------------------

        // 프록시 대상: 클래스 기반 객체 (인터페이스 없음)
        ConcreteSubject concreteSubject = new ConcreteSubject();

        // ProxyFactory 생성 (CGLIB 방식이 자동 선택됨)
        ProxyFactory proxyFactory2 = new ProxyFactory(concreteSubject);

        // 공통 기능(Advice) 추가
        proxyFactory2.addAdvice(new AdviceImpl());

        // 프록시 생성
        ConcreteSubject proxy2 = (ConcreteSubject) proxyFactory2.getProxy();

        // 프록시 메서드 호출 (call과 sayHello 모두 Advice 적용됨)
        proxy2.call();       // → Advice + 실제 메서드 실행
        proxy2.sayHello();   // → Advice + 실제 메서드 실행
    }
}
```

프록시 팩토리



- 실행 결과를 보시면 ProxyFacotry에서 JDK프록시, CGLIB프록시 둘다 상황에 맞게 프록시를 만들어 준것을 알수 있습니다. Call()메서드를 부르면 "서버 시작","서버 종료" 출력하고, SayHello()메서드를 부르면 "Hello!"를 호출합니다.
- 여기서 SayHello()메서드를 호출할때는 Advice의 로직을 추가하고 싶지 않다면 어떻게 해야할까요?
- **Pointcut을 사용**하면 됩니다! Pointcut은 어떤 메서드에 Advice를 적용할지 선택적으로 지정할 수 있는 기능입니다. 즉, call()에는 Advice를 적용하고, sayHello()는 건너뛰게 만들 수 있죠.

프록시 팩토리 코드(Pointcut)

```java
public class ProxyFactoryWithAdvisorTest {

    @Test
    public void advisorTest() {
        // 대상 객체
        Subject target = new RealSubject();

        // 프록시 팩토리 생성
        ProxyFactory proxyFactory = new ProxyFactory(target);

        // Pointcut: call() 메서드에만 적용
        NameMatchMethodPointcut pointcut = new NameMatchMethodPointcut();
        pointcut.setMappedName("call");  // call() 메서드만 매칭

        // Advice 설정
        Advice advice = new AdviceImpl();

        // Advisor 생성 및 추가
        Advisor advisor = new DefaultPointcutAdvisor(pointcut, advice);
        proxyFactory.addAdvisor(advisor);

        // 프록시 생성
        Subject proxy = (Subject) proxyFactory.getProxy();

        // 호출 테스트
        proxy.call();       // Advice 적용됨
        proxy.sayHello();   // Advice 미적용
    }
}
```

```
✓ 테스트 통과: 1 /1개 테스트 – 39ms
> Task :testClasses UP-TO-DATE
17:05:00.706 [Test worker] INFO com.example.proxy_practice.proxyfactory.advice.AdviceImpl -- proxy 실행
17:05:00.708 [Test worker] INFO com.example.proxy_practice.proxyfactory.code.RealSubject -- 서버 시작
17:05:00.708 [Test worker] INFO com.example.proxy_practice.proxyfactory.code.RealSubject -- 서버 끝
17:05:00.708 [Test worker] INFO com.example.proxy_practice.proxyfactory.advice.AdviceImpl -- proxy 종료
17:05:00.708 [Test worker] INFO com.example.proxy_practice.proxyfactory.code.RealSubject -- Hello!
```

- 실행 결과를 보시면 call()메서드에만 포인트 컷을 걸어주어, sayHello()메서드가 부가 로직이 들어가지 않은 것을 확인할 수 있습니다.

정리

- 스프링에서 어떻게 프록시를 사용하는지, 그리고 JDK 동적 프록시와 CGLIB 프록시의 차이, 그리고 이 둘을 자동으로 선택해주는 ProxyFactory의 사용법까지 알아보았습니다.
- 또한, **부가 기능을 프록시에 적용하기 위한 Advice**, **적용 대상을 지정하기 위한 Pointcut**, **이 둘을 결합한 Advisor**를 통해 특정 메서드에만 프록시 로직을 적용하는 방법도 함께 실습해보았습니다.
- 프록시는 단순한 디자인 패턴이지만, 스프링에서는 이를 기반으로 트랜잭션, 보안, 캐싱, 비동기 처리 등 다양한 기능을 모듈화하여 적용하고 있습니다.
- 예를 들어 @Transactional로 트랜잭션 자동 처리, @Async로 비동기 로직 실행, @PreAuthorize로 보안 인가 처리, @Cacheable로 캐싱, @Validated로 검증 로직 자동화 등이 **프록시 기반 AOP로 구현**되어 있습니다.
- 스프링은 프록시를 기반으로 **수많은 부가기능을 코드 변경 없이 간단하게 적용할 수 있게 해주며, 이는 관심사의 분리 원칙을 실현**하는 기술입니다.