5. 동적 프록시 기술

#1.인강/6.핵심 원리 - 고급편/강의#

리플렉션

지금까지 프록시를 사용해서 기존 코드를 변경하지 않고, 로그 추적기라는 부가 기능을 적용할 수 있었다. 그런데 문제는 대상 클래스 수 만큼 로그 추적을 위한 프록시 클래스를 만들어야 한다는 점이다.
로그 추적을 위한 프록시 클래스들의 소스코드는 거의 같은 모양을 하고 있다.

자바가 기본으로 제공하는 JDK 동적 프록시 기술이나 CGLIB 같은 프록시 생성 오픈소스 기술을 활용하면 프록시 객체를 동적으로 만들어낼 수 있다. 쉽게 이야기해서 프록시 클래스를 지금처럼 계속 만들지 않아도 된다는 것이다. 프록시를 적용할 코드를 하나만 만들어두고 동적 프록시 기술을 사용해서 프록시 객체를 찍어내면 된다. 자세한 내용은 조금 뒤에 코드로 확인해보자.

JDK 동적 프록시를 이해하기 위해서는 먼저 자바의 리플렉션 기술을 이해해야 한다.
리플렉션 기술을 사용하면 클래스나 메서드의 메타정보를 동적으로 획득하고, 코드도 동적으로 호출할 수 있다.
여기서는 JDK 동적 프록시를 이해하기 위한 최소한의 리플렉션 기술을 알아보자.

ReflectionTest

```
package hello.proxy.jdkdynamic;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import java.lang.reflect.Method;

@Slf4j
public class ReflectionTest {
```

```
    @Test
    void reflection0() {
        Hello target = new Hello();

        //공통 로직1 시작
        log.info("start");
        String result1 = target.callA(); //호출하는 메서드가 다름
        log.info("result={}", result1);
        //공통 로직1 종료

        //공통 로직2 시작
        log.info("start");
        String result2 = target.callB(); //호출하는 메서드가 다름
        log.info("result={}", result2);
        //공통 로직2 종료
    }

    @Slf4j
    static class Hello {
        public String callA() {
            log.info("callA");
            return "A";
        }
        public String callB() {
            log.info("callB");
            return "B";
        }
    }

}
```

- 공통 로직1과 공통 로직2는 호출하는 메서드만 다르고 전체 코드 흐름이 완전히 같다.
  - 먼저 start 로그를 출력한다.
  - 어떤 메서드를 호출한다.
  - 메서드의 호출 결과를 로그로 출력한다.
- 여기서 공통 로직1과 공통 로직 2를 하나의 메서드로 뽑아서 합칠 수 있을까?
- 쉬워 보이지만 메서드로 뽑아서 공통화하는 것이 생각보다 어렵다. 왜냐하면 중간에 호출하는 메서드가 다르기 때문이다.
- 호출하는 메서드인 `target.callA()`, `target.callB()` 이 부분만 동적으로 처리할 수 있다면 문제를 해결할 수 있을 듯 하다.

```
log.info("start");
String result = xxx(); //호출 대상이 다름, 동적 처리 필요
log.info("result={}", result);
```

이럴 때 사용하는 기술이 바로 리플렉션이다. 리플렉션은 클래스나 메서드의 메타정보를 사용해서 동적으로 호출하는 메서드를 변경할 수 있다. 바로 리플렉션 사용해보자.

> **참고**: 람다를 사용해서 공통화 하는 것도 가능하다. 여기서는 람다를 사용하기 어려운 상황이라 가정하자. 그리고 리플렉션 학습이 목적이니 리플렉션에 집중하자.

ReflectionTest - reflection1 추가

```
@Test
void reflection1() throws Exception {
    //클래스 정보
    Class classHello =
Class.forName("hello.proxy.jdkdynamic.ReflectionTest$Hello");

    Hello target = new Hello();
    //callA 메서드 정보
    Method methodCallA = classHello.getMethod("callA");
    Object result1 = methodCallA.invoke(target);
    log.info("result1={}", result1);

    //callB 메서드 정보
    Method methodCallB = classHello.getMethod("callB");
    Object result2 = methodCallB.invoke(target);
    log.info("result2={}", result2);
}
```

- `Class.forName("hello.proxy.jdkdynamic.ReflectionTest$Hello")` : 클래스 메타정보를 획득한다. 참고로 내부 클래스는 구분을 위해 `$`를 사용한다.
- `classHello.getMethod("call")` : 해당 클래스의 `call` 메서드 메타정보를 획득한다.
- `methodCallA.invoke(target)` : 획득한 메서드 메타정보로 실제 인스턴스의 메서드를 호출한다. 여기서 `methodCallA`는 `Hello` 클래스의 `callA()` 이라는 메서드 메타정보이다. `methodCallA.invoke(인스턴스)`를 호출하면서 인스턴스를 넘겨주면 해당 인스턴스의 `callA()` 메서드를 찾아서 실행한다. 여기서는 `target`의 `callA()` 메서드를 호출한다.

그런데 `target.callA()`나 `target.callB()` 메서드를 직접 호출하면 되지 이렇게 메서드 정보를 획득해서 메서드를 호출하면 어떤 효과가 있을까? 여기서 중요한 핵심은 클래스나 메서드 정보를 동적으로 변경할 수 있다는 점이다.

기존의 `callA()`, `callB()` 메서드를 직접 호출하는 부분이 `Method`로 대체되었다. 덕분에 이제 공통 로직을 만들 수 있게 되었다.

ReflectionTest - reflection2 추가

```
@Test
void reflection2() throws Exception {
    Class classHello =
Class.forName("hello.proxy.jdkdynamic.ReflectionTest$Hello");

    Hello target = new Hello();
    Method methodCallA = classHello.getMethod("callA");
    dynamicCall(methodCallA, target);

    Method methodCallB = classHello.getMethod("callB");
    dynamicCall(methodCallB, target);
}

private void dynamicCall(Method method, Object target) throws Exception {
    log.info("start");
    Object result = method.invoke(target);
    log.info("result={}", result);
}
```

- `dynamicCall(Method method, Object target)`
  - 공통 로직1, 공통 로직2를 한번에 처리할 수 있는 통합된 공통 처리 로직이다.
  - `Method method`: 첫 번째 파라미터는 호출할 메서드 정보가 넘어온다. 이것이 핵심이다. 기존에는 메서드 이름을 직접 호출했지만, 이제는 `Method`라는 메타정보를 통해서 호출할 메서드 정보가 동적으로 제공된다.
  - `Object target`: 실제 실행할 인스턴스 정보가 넘어온다. 타입이 `Object`라는 것은 어떠한 인스턴스도 받을 수 있다는 뜻이다. 물론 `method.invoke(target)`를 사용할 때 호출할 클래스와 메서드 정보가 서로 다르면 예외가 발생한다.

정리

정적인 `target.callA()`, `target.callB()` 코드를 리플렉션을 사용해서 `Method`라는 메타정보로 추상화했다. 덕분에 공통 로직을 만들 수 있게 되었다.

주의

리플렉션을 사용하면 클래스와 메서드의 메타정보를 사용해서 애플리케이션을 동적으로 유연하게 만들 수 있다. 하지만 리플렉션 기술은 런타임에 동작하기 때문에, 컴파일 시점에 오류를 잡을 수 없다.

예를 들어서 지금까지 살펴본 코드에서 `getMethod("callA")` 안에 들어가는 문자를 실수로 `getMethod("callZ")` 로 작성해도 컴파일 오류가 발생하지 않는다. 그러나 해당 코드를 직접 실행하는 시점에 발생하는 오류인 런타임 오류가 발생한다.
가장 좋은 오류는 개발자가 즉시 확인할 수 있는 컴파일 오류이고, 가장 무서운 오류는 사용자가 직접 실행할 때 발생하는 런타임 오류다.

따라서 리플렉션은 일반적으로 사용하면 안된다. 지금까지 프로그래밍 언어가 발달하면서 타입 정보를 기반으로 컴파일 시점에 오류를 잡아준 덕분에 개발자가 편하게 살았는데, 리플렉션은 그것에 역행하는 방식이다.

리플렉션은 프레임워크 개발이나 또는 매우 일반적인 공통 처리가 필요할 때 부분적으로 주의해서 사용해야 한다.

JDK 동적 프록시 - 소개

지금까지 프록시를 적용하기 위해 적용 대상의 숫자 만큼 많은 프록시 클래스를 만들었다. 적용 대상이 100개면 프록시 클래스도 100개 만들었다. 그런데 앞서 살펴본 것과 같이 프록시 클래스의 기본 코드와 흐름은 거의 같고, 프록시를 어떤 대상에 적용하는가 정도만 차이가 있었다. 쉽게 이야기해서 프록시의 로직은 같은데, 적용 대상만 차이가 있는 것이다.

이 문제를 해결하는 것이 바로 동적 프록시 기술이다.
동적 프록시 기술을 사용하면 개발자가 직접 프록시 클래스를 만들지 않아도 된다. 이름 그대로 프록시 객체를 동적으로 런타임에 개발자 대신 만들어준다. 그리고 동적 프록시에 원하는 실행 로직을 지정할 수 있다.

사실 동적 프록시는 말로는 이해하기 쉽지 않다. 바로 예제 코드를 보자.

> **주의**
> JDK 동적 프록시는 인터페이스를 기반으로 프록시를 동적으로 만들어준다. 따라서 인터페이스가 필수이다.

먼저 자바 언어가 기본으로 제공하는 JDK 동적 프록시를 알아보자.

기본 예제 코드

JDK 동적 프록시를 이해하기 위해 아주 단순한 예제 코드를 만들어보자.
간단히 `A`, `B` 클래스를 만드는데, JDK 동적 프록시는 인터페이스가 필수이다. 따라서 인터페이스와 구현체로 구분했

다.

AInterface

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.jdkdynamic.code;

public interface AInterface {
    String call();
}
```

AImpl

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.jdkdynamic.code;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class AImpl implements AInterface {
    @Override
    public String call() {
        log.info("A 호출");
        return "a";
    }
}
```

BInterface

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.jdkdynamic.code;

public interface BInterface {
    String call();
}
```

BImpl

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.jdkdynamic.code;

import lombok.extern.slf4j.Slf4j;

@Slf4j
```

```
public class BImpl implements BInterface {
    @Override
    public String call() {
        log.info("B 호출");
        return "b";
    }
}
```

JDK 동적 프록시 - 예제 코드

JDK 동적 프록시 InvocationHandler

JDK 동적 프록시에 적용할 로직은 `InvocationHandler` 인터페이스를 구현해서 작성하면 된다.

JDK 동적 프록시가 제공하는 InvocationHandler

```
package java.lang.reflect;

public interface InvocationHandler {

     public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

제공되는 파라미터는 다음과 같다.

- `Object proxy` : 프록시 자신
- `Method method` : 호출한 메서드
- `Object[] args` : 메서드를 호출할 때 전달한 인수

이제 구현 코드를 보자.

TimeInvocationHandler

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.jdkdynamic.code;

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

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

@Slf4j
public class TimeInvocationHandler implements InvocationHandler {

    private final Object target;

    public TimeInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        log.info("TimeProxy 실행");
        long startTime = System.currentTimeMillis();

        Object result = method.invoke(target, args);

        long endTime = System.currentTimeMillis();
        long resultTime = endTime - startTime;
        log.info("TimeProxy 종료 resultTime={}", resultTime);
        return result;
    }
}
```

- `TimeInvocationHandler`은 `InvocationHandler` 인터페이스를 구현한다. 이렇게해서 JDK 동적 프록시에 적용할 공통 로직을 개발할 수 있다.
- `Object target`: 동적 프록시가 호출할 대상
- `method.invoke(target, args)`: 리플렉션을 사용해서 `target` 인스턴스의 메서드를 실행한다. `args`는 메서드 호출시 넘겨줄 인수이다.

이제 테스트 코드로 JDK 동적 프록시를 사용해보자.

JdkDynamicProxyTest

```
package hello.proxy.jdkdynamic;

import hello.proxy.jdkdynamic.code.*;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
```

```
import java.lang.reflect.Proxy;

@Slf4j
public class JdkDynamicProxyTest {

    @Test
    void dynamicA() {
        AInterface target = new AImpl();
        TimeInvocationHandler handler = new TimeInvocationHandler(target);
        AInterface proxy = (AInterface)
Proxy.newProxyInstance(AInterface.class.getClassLoader(), new Class[]
{AInterface.class}, handler);
        proxy.call();
        log.info("targetClass={}", target.getClass());
        log.info("proxyClass={}", proxy.getClass());
    }

    @Test
    void dynamicB() {
        BInterface target = new BImpl();
        TimeInvocationHandler handler = new TimeInvocationHandler(target);
        BInterface proxy = (BInterface)
Proxy.newProxyInstance(BInterface.class.getClassLoader(), new Class[]
{BInterface.class}, handler);
        proxy.call();

        log.info("targetClass={}", target.getClass());
        log.info("proxyClass={}", proxy.getClass());
    }

}
```

- `new TimeInvocationHandler(target)` : 동적 프록시에 적용할 핸들러 로직이다.
- `Proxy.newProxyInstance(AInterface.class.getClassLoader(), new Class[] {AInterface.class}, handler)`
  - 동적 프록시는 `java.lang.reflect.Proxy` 를 통해서 생성할 수 있다.
  - 클래스 로더 정보, 인터페이스, 그리고 핸들러 로직을 넣어주면 된다. 그러면 해당 인터페이스를 기반으로 동적 프록시를 생성하고 그 결과를 반환한다.

dynamicA() 출력 결과

```
TimeInvocationHandler - TimeProxy 실행
```

```
AImpl - A 호출
TimeInvocationHandler - TimeProxy 종료 resultTime=0
JdkDynamicProxyTest - targetClass=class hello.proxy.jdkdynamic.code.AImpl
JdkDynamicProxyTest - proxyClass=class com.sun.proxy.$Proxy1
```

출력 결과를 보면 프록시가 정상 수행된 것을 확인할 수 있다.

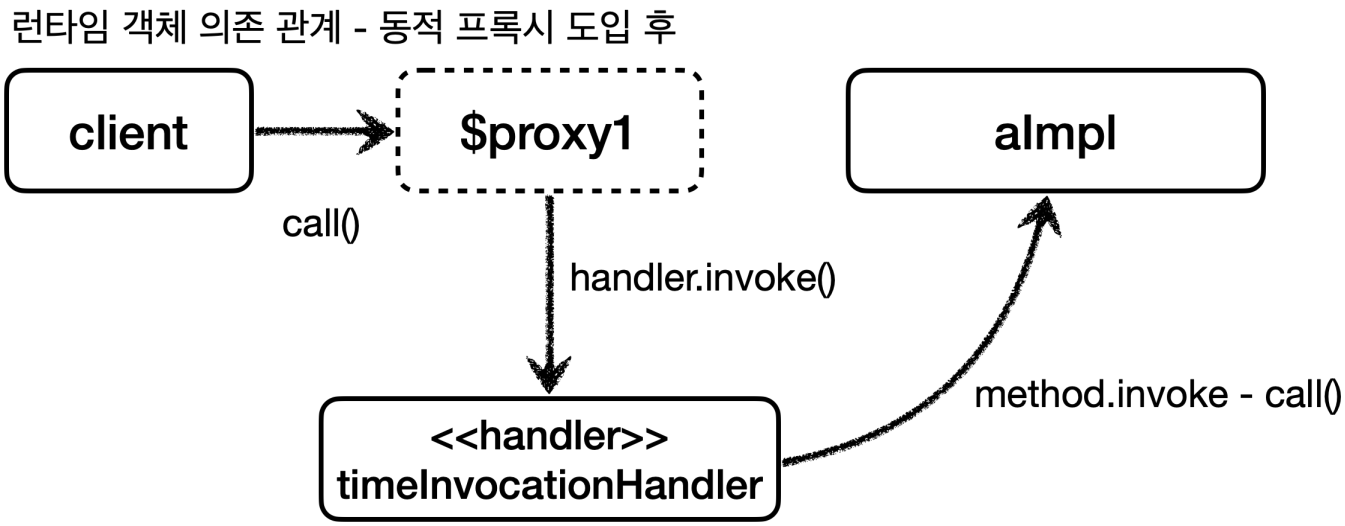
생성된 JDK 동적 프록시

`proxyClass=class com.sun.proxy.$Proxy1` 이 부분이 동적으로 생성된 프록시 클래스 정보이다. 이것은 우리가 만든 클래스가 아니라 JDK 동적 프록시가 이름 그대로 동적으로 만들어준 프록시이다. 이 프록시는 `TimeInvocationHandler` 로직을 실행한다.

실행 순서

1. 클라이언트는 JDK 동적 프록시의 `call()` 을 실행한다.
2. JDK 동적 프록시는 `InvocationHandler.invoke()` 를 호출한다. `TimeInvocationHandler` 가 구현체로 있으로 `TimeInvocationHandler.invoke()` 가 호출된다.
3. `TimeInvocationHandler` 가 내부 로직을 수행하고, `method.invoke(target, args)` 를 호출해서 `target` 인 실제 객체(`AImpl`)를 호출한다.
4. `AImpl` 인스턴스의 `call()` 이 실행된다.
5. `AImpl` 인스턴스의 `call()` 의 실행이 끝나면 `TimeInvocationHandler` 로 응답이 돌아온다. 시간 로그를 출력하고 결과를 반환한다.

실행 순서 그림



동적 프록시 클래스 정보

`dynamicA()` 와 `dynamicB()` 둘을 동시에 함께 실행하면 JDK 동적 프록시가 각각 다른 동적 프록시 클래스를 만들어주는 것을 확인할 수 있다.

```
proxyClass=class com.sun.proxy.$Proxy1 //dynamicA
proxyClass=class com.sun.proxy.$Proxy2 //dynamicB
```

정리

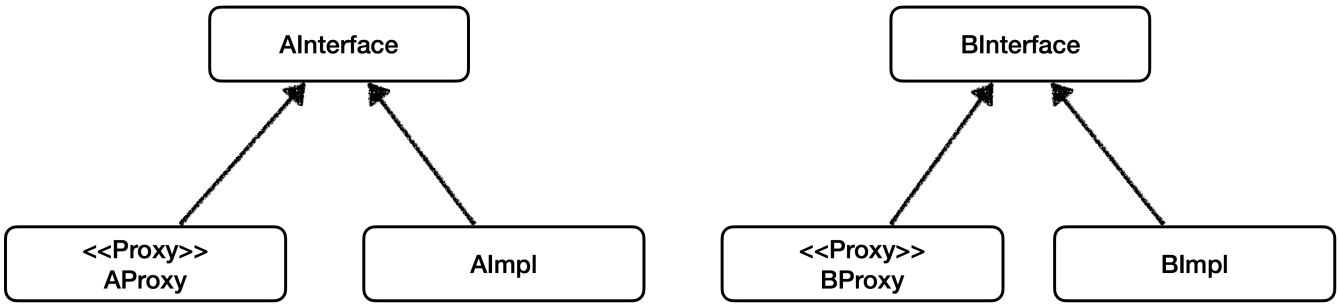
예제를 보면 `AImpl`, `BImpl` 각각 프록시를 만들지 않았다. 프록시는 JDK 동적 프록시를 사용해서 동적으로 만들고 `TimeInvocationHandler`는 공통으로 사용했다.

JDK 동적 프록시 기술 덕분에 적용 대상 만큼 프록시 객체를 만들지 않아도 된다. 그리고 같은 부가 기능 로직을 한번만 개발해서 공통으로 적용할 수 있다. 만약 적용 대상이 100개여도 동적 프록시를 통해서 생성하고, 각각 필요한 `InvocationHandler`만 만들어서 넣어주면 된다.
결과적으로 프록시 클래스를 수 없이 만들어야 하는 문제도 해결하고, 부가 기능 로직도 하나의 클래스에 모아서 단일 책임 원칙(SRP)도 지킬 수 있게 되었다.

JDK 동적 프록시 없이 직접 프록시를 만들어서 사용할 때와 JDK 동적 프록시를 사용할 때의 차이를 그림으로 비교해 보자.

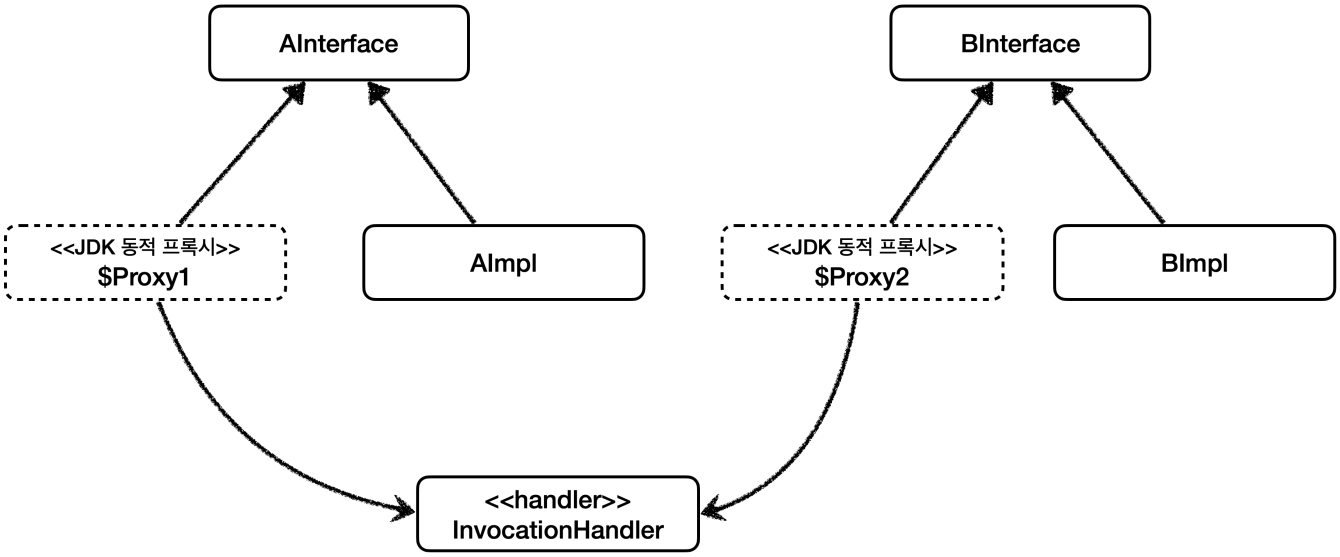
JDK 동적 프록시 도입 전 - 직접 프록시 생성

클래스 의존 관계 - JDK 동적 프록시 도입 전



JDK 동적 프록시 도입 후

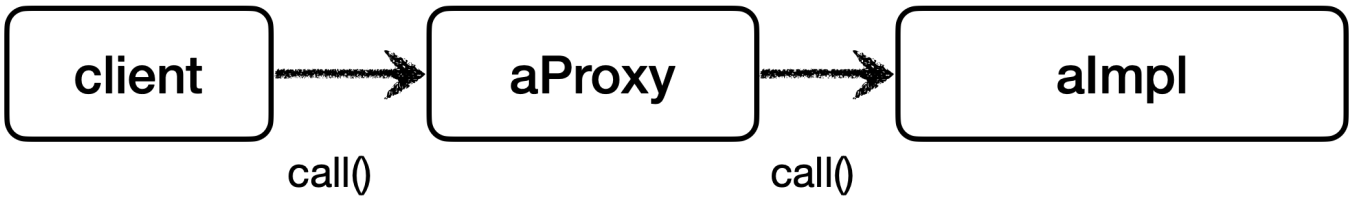
클래스 의존 관계 - JDK 동적 프록시 도입 후



- 점선은 개발자가 직접 만드는 클래스가 아니다.

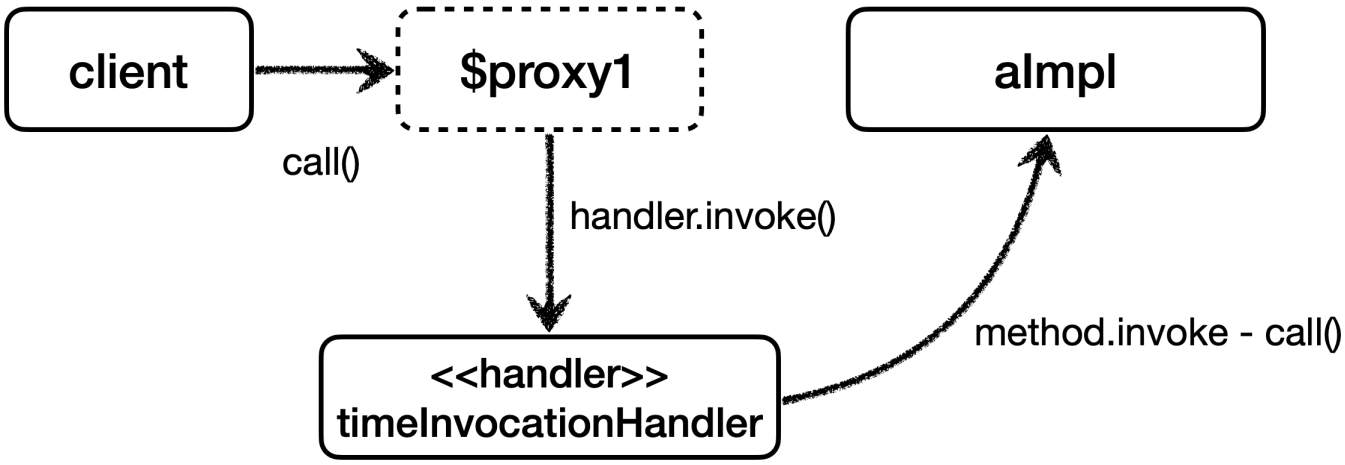
JDK 동적 프록시 도입 전

런타임 객체 의존 관계 - 동적 프록시 도입 전



JDK 동적 프록시 도입 후

런타임 객체 의존 관계 - 동적 프록시 도입 후



지금까지 학습한 JDK 동적 프록시를 애플리케이션에 적용해보자.

JDK 동적 프록시 - 적용1

JDK 동적 프록시는 인터페이스가 필수이기 때문에 V1 애플리케이션에만 적용할 수 있다.

먼저 `LogTrace`를 적용할 수 있는 `InvocationHandler`를 만들자.

LogTraceBasicHandler

```
package hello.proxy.config.v2_dynamicproxy.handler;

import hello.proxy.trace.TraceStatus;
import hello.proxy.trace.logtrace.LogTrace;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class LogTraceBasicHandler implements InvocationHandler {

    private final Object target;
    private final LogTrace logTrace;

    public LogTraceBasicHandler(Object target, LogTrace logTrace) {
        this.target = target;
        this.logTrace = logTrace;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

        TraceStatus status = null;
        try {
            String message = method.getDeclaringClass().getSimpleName() + "."
                    + method.getName() + "()";
            status = logTrace.begin(message);

            //로직 호출
            Object result = method.invoke(target, args);

            logTrace.end(status);
            return result;
```

```
            } catch (Exception e) {
                logTrace.exception(status, e);
                throw e;
            }

        }
    }
```

- `LogTraceBasicHandler`는 `InvocationHandler` 인터페이스를 구현해서 JDK 동적 프록시에서 사용된다.
- `private final Object target`: 프록시가 호출할 대상이다.
- `String message = method.getDeclaringClass().getSimpleName() + "." ...`
  - `LogTrace`에 사용할 메시지이다. 프록시를 직접 개발할 때는 `"OrderController.request()"`와 같이 프록시마다 호출되는 클래스와 메서드 이름을 직접 남겼다. 이제는 `Method`를 통해서 호출되는 메서드 정보와 클래스 정보를 동적으로 확인할 수 있기 때문에 이 정보를 사용하면 된다.

동적 프록시를 사용하도록 수동 빈 등록을 설정하자.

DynamicProxyBasicConfig

```
package hello.proxy.config.v2_dynamicproxy;

import hello.proxy.app.v1.*;
import hello.proxy.config.v2_dynamicproxy.handler.LogTraceBasicHandler;
import hello.proxy.trace.logtrace.LogTrace;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.lang.reflect.Proxy;

@Configuration
public class DynamicProxyBasicConfig {

    @Bean
    public OrderControllerV1 orderControllerV1(LogTrace logTrace) {
        OrderControllerV1 orderController = new
OrderControllerV1Impl(orderServiceV1(logTrace));

        OrderControllerV1 proxy = (OrderControllerV1)
Proxy.newProxyInstance(OrderControllerV1.class.getClassLoader(),
                new Class[]{OrderControllerV1.class},
                new LogTraceBasicHandler(orderController, logTrace)
        );
```

```
        return proxy;
    }

    @Bean
    public OrderServiceV1 orderServiceV1(LogTrace logTrace) {
        OrderServiceV1 orderService = new
OrderServiceV1Impl(orderRepositoryV1(logTrace));

        OrderServiceV1 proxy = (OrderServiceV1)
Proxy.newProxyInstance(OrderServiceV1.class.getClassLoader(),
                new Class[]{OrderServiceV1.class},
                new LogTraceBasicHandler(orderService, logTrace)
        );
        return proxy;
    }

    @Bean
    public OrderRepositoryV1 orderRepositoryV1(LogTrace logTrace) {
        OrderRepositoryV1 orderRepository = new OrderRepositoryV1Impl();

        OrderRepositoryV1 proxy = (OrderRepositoryV1)
Proxy.newProxyInstance(OrderRepositoryV1.class.getClassLoader(),
                new Class[]{OrderRepositoryV1.class},
                new LogTraceBasicHandler(orderRepository, logTrace)
        );
        return proxy;
    }
}
```

- 이전에는 프록시 클래스를 직접 개발했지만, 이제는 JDK 동적 프록시 기술을 사용해서 각각의 `Controller`, `Service`, `Repository`에 맞는 동적 프록시를 생성해주면 된다.
- `LogTraceBasicHandler`: 동적 프록시를 만들더라도 `LogTrace`를 출력하는 로직은 모두 같기 때문에 프록시는 모두 `LogTraceBasicHandler`를 사용한다.

ProxyApplication - 수정

```
import hello.proxy.config.v2_dynamicproxy.DynamicProxyBasicConfig;

//@Import({AppV1Config.class, AppV2Config.class})
//@Import(InterfaceProxyConfig.class)
//@Import(ConcreteProxyConfig.class)
@Import(DynamicProxyBasicConfig.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();
    }

}
```

`@Import(DynamicProxyBasicConfig.class)` : 이제 동적 프록시 설정을 `@Import` 하고 실행해보자.

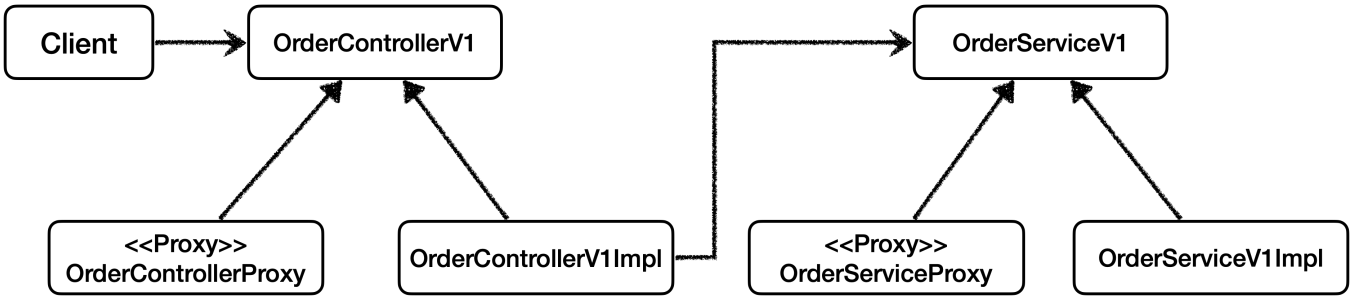
실행

- http://localhost:8080/v1/request?itemId=hello

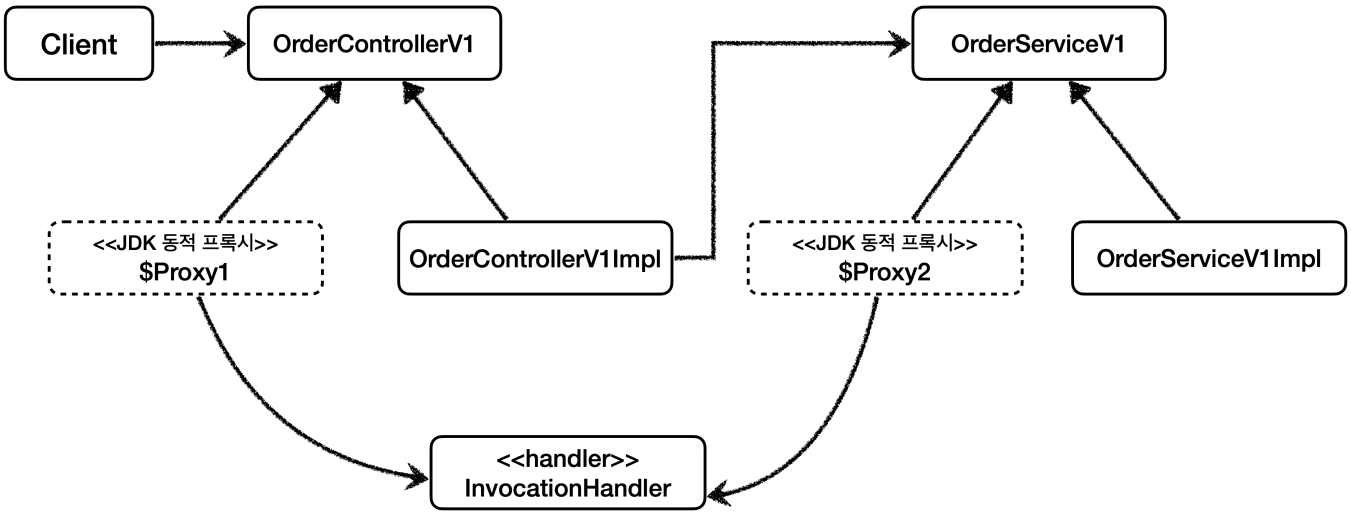
실행해보면 정상 수행되는 것을 확인할 수 있다.

그림으로 정리

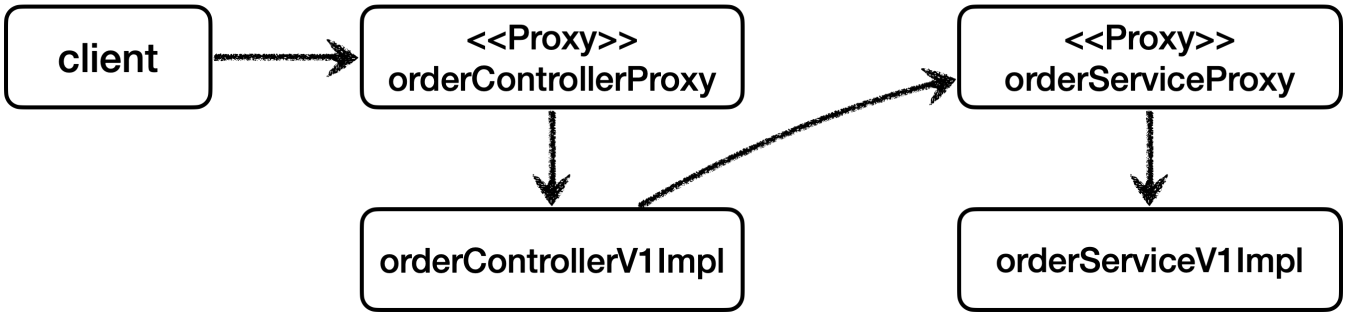
클래스 의존 관계 - 직접 프록시 사용



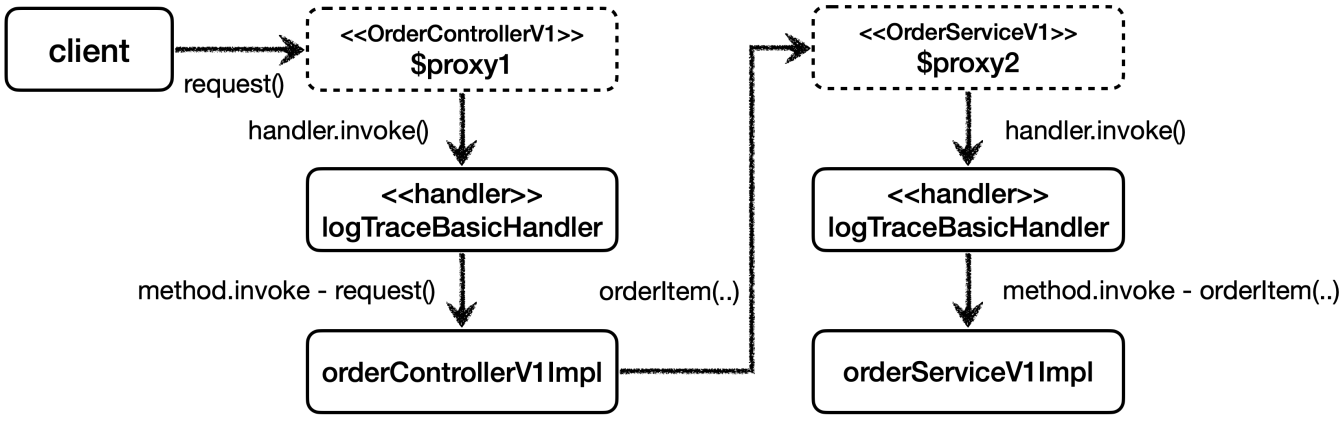
클래스 의존 관계 - JDK 동적 프록시 사용



런타임 객체 의존 관계 - 직접 프록시 사용



런타임 객체 의존 관계 - JDK 동적 프록시 사용



남은 문제

- http://localhost:8080/v1/no-log
- no-log를 실행해도 동적 프록시가 적용되고, `LogTraceBasicHandler`가 실행되기 때문에 로그가 남는다. 이 부분을 로그가 남지 않도록 처리해야 한다.

JDK 동적 프록시 - 적용2

메서드 이름 필터 기능 추가

- http://localhost:8080/v1/no-log

요구사항에 의해 이것을 호출 했을 때는 로그가 남으면 안된다.

이런 문제를 해결하기 위해 메서드 이름을 기준으로 특정 조건을 만족할 때만 로그를 남기는 기능을 개발해보자.

LogTraceFilterHandler

```
package hello.proxy.config.v2_dynamicproxy.handler;

import hello.proxy.trace.TraceStatus;
import hello.proxy.trace.logtrace.LogTrace;
import org.springframework.util.PatternMatchUtils;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class LogTraceFilterHandler implements InvocationHandler {

    private final Object target;
    private final LogTrace logTrace;
    private final String[] patterns;

    public LogTraceFilterHandler(Object target, LogTrace logTrace, String...
patterns) {
        this.target = target;
        this.logTrace = logTrace;
        this.patterns = patterns;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

        //메서드 이름 필터
        String methodName = method.getName();
        if (!PatternMatchUtils.simpleMatch(patterns, methodName)) {
```

```
                return method.invoke(target, args);
            }

            TraceStatus status = null;
            try {
                String message = method.getDeclaringClass().getSimpleName() + "."
                        + method.getName() + "()";
                status = logTrace.begin(message);

                //로직 호출
                Object result = method.invoke(target, args);

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

        }
}
```

- `LogTraceFilterHandler`는 기존 기능에 다음 기능이 추가되었다.
  - 특정 메서드 이름이 매칭 되는 경우에만 `LogTrace` 로직을 실행한다. 이름이 매칭되지 않으면 실제 로직을 바로 호출한다.
- 스프링이 제공하는 `PatternMatchUtils.simpleMatch(..)`를 사용하면 단순한 매칭 로직을 쉽게 적용할 수 있다.
  - `xxx` : xxx가 정확히 매칭되면 참
  - `xxx*` : xxx로 시작하면 참
  - `*xxx` : xxx로 끝나면 참
  - `*xxx*` : xxx가 있으면 참
- `String[] patterns` : 적용할 패턴은 생성자를 통해서 외부에서 받는다.

DynamicProxyFilterConfig

```
package hello.proxy.config.v2_dynamicproxy;

import hello.proxy.app.v1.*;
import hello.proxy.config.v2_dynamicproxy.handler.LogTraceFilterHandler;
import hello.proxy.trace.logtrace.LogTrace;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import java.lang.reflect.Proxy;

@Configuration
public class DynamicProxyFilterConfig {

    private static final String[] PATTERNS = {"request*", "order*", "save*"};

    @Bean
    public OrderControllerV1 orderControllerV1(LogTrace logTrace) {
        OrderControllerV1 orderController = new
OrderControllerV1Impl(orderServiceV1(logTrace));

        OrderControllerV1 proxy = (OrderControllerV1)
Proxy.newProxyInstance(OrderControllerV1.class.getClassLoader(),
                new Class[]{OrderControllerV1.class},
                new LogTraceFilterHandler(orderController, logTrace, PATTERNS)
        );
        return proxy;
    }

    @Bean
    public OrderServiceV1 orderServiceV1(LogTrace logTrace) {
        OrderServiceV1 orderService = new
OrderServiceV1Impl(orderRepositoryV1(logTrace));

        OrderServiceV1 proxy = (OrderServiceV1)
Proxy.newProxyInstance(OrderServiceV1.class.getClassLoader(),
                new Class[]{OrderServiceV1.class},
                new LogTraceFilterHandler(orderService, logTrace, PATTERNS)
        );
        return proxy;
    }

    @Bean
    public OrderRepositoryV1 orderRepositoryV1(LogTrace logTrace) {
        OrderRepositoryV1 orderRepository = new OrderRepositoryV1Impl();

        OrderRepositoryV1 proxy = (OrderRepositoryV1)
Proxy.newProxyInstance(OrderRepositoryV1.class.getClassLoader(),
                new Class[]{OrderRepositoryV1.class},
                new LogTraceFilterHandler(orderRepository, logTrace, PATTERNS)
        );
```

```
            return proxy;
        }
    }
```

- `public static final String[] PATTERNS = {"request*", "order*", "save*"};`
  - 적용할 패턴이다. `request`, `order`, `save`로 시작하는 메서드에 로그가 남는다.
- `LogTraceFilterHandler`: 앞서 만든 필터 기능이 있는 핸들러를 사용한다. 그리고 핸들러에 적용 패턴도 넣어준다.

ProxyApplication - 추가

```
import hello.proxy.config.v2_dynamicproxy.DynamicProxyFilterConfig;

@Import(DynamicProxyFilterConfig.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();
    }

}
```

`@Import(DynamicProxyFilterConfig.class)`으로 방금 만든 설정을 추가하자.

실행

- http://localhost:8080/v1/request?itemId=hello
- http://localhost:8080/v1/no-log

실행해보면 `no-log`가 사용하는 `noLog()` 메서드에는 로그가 남지 않는 것을 확인할 수 있다.

JDK 동적 프록시 - 한계

JDK 동적 프록시는 인터페이스가 필수이다.

그렇다면 V2 애플리케이션 처럼 인터페이스 없이 클래스만 있는 경우에는 어떻게 동적 프록시를 적용할 수 있을까?

이것은 일반적인 방법으로는 어렵고 `CGLIB`라는 바이트코드를 조작하는 특별한 라이브러리를 사용해야 한다.

CGLIB - 소개

CGLIB: Code Generator Library

- CGLIB는 바이트코드를 조작해서 동적으로 클래스를 생성하는 기술을 제공하는 라이브러리이다.
- CGLIB를 사용하면 인터페이스가 없어도 구체 클래스만 가지고 동적 프록시를 만들어낼 수 있다.
- CGLIB는 원래는 외부 라이브러리인데, 스프링 프레임워크가 스프링 내부 소스 코드에 포함했다. 따라서 스프링을 사용한다면 별도의 외부 라이브러리를 추가하지 않아도 사용할 수 있다.

참고로 우리가 CGLIB를 직접 사용하는 경우는 거의 없다. 이후에 설명할 스프링의 `ProxyFactory`라는 것이 이 기술을 편리하게 사용하게 도와주기 때문에, 너무 깊이있게 파기 보다는 CGLIB가 무엇인지 대략 개념만 잡으면 된다.
예제 코드로 CGLIB를 간단히 이해해보자.

공통 예제 코드

앞으로 다양한 상황을 설명하기 위해서 먼저 공통으로 사용할 예제 코드를 만들어보자.

- 인터페이스와 구현이 있는 서비스 클래스 - `ServiceInterface`, `ServiceImpl`
- 구체 클래스만 있는 서비스 클래스 - `ConcreteService`

ServiceInterface

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.common.service;

public interface ServiceInterface {
    void save();

    void find();
}
```

ServiceImpl

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.common.service;

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

```
@Slf4j
public class ServiceImpl implements ServiceInterface {

    @Override
    public void save() {
        log.info("save 호출");
    }

    @Override
    public void find() {
        log.info("find 호출");
    }

}
```

ConcreteService

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.common.service;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class ConcreteService {
    public void call() {
        log.info("ConcreteService 호출");
    }
}
```

CGLIB - 예제 코드

CGLIB 코드

JDK 동적 프록시에서 실행 로직을 위해 `InvocationHandler`를 제공했듯이, CGLIB는 `MethodInterceptor`를 제공한다.

MethodInterceptor -CGLIB 제공

```
package org.springframework.cglib.proxy;
```

```
public interface MethodInterceptor extends Callback {
    Object intercept(Object obj, Method method, Object[] args, MethodProxy
proxy) throws Throwable;
}
```

- `obj` : CGLIB가 적용된 객체
- `method` : 호출된 메서드
- `args` : 메서드를 호출하면서 전달된 인수
- `proxy` : 메서드 호출에 사용

TimeMethodInterceptor

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.proxy.cglib.code;

import lombok.extern.slf4j.Slf4j;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

@Slf4j
public class TimeMethodInterceptor implements MethodInterceptor {

    private final Object target;

    public TimeMethodInterceptor(Object target) {
        this.target = target;
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
        log.info("TimeProxy 실행");
        long startTime = System.currentTimeMillis();

        Object result = proxy.invoke(target, args);

        long endTime = System.currentTimeMillis();
        long resultTime = endTime - startTime;
        log.info("TimeProxy 종료 resultTime={}", resultTime);
        return result;
```

```
    }
}
```

- `TimeMethodInterceptor`는 `MethodInterceptor` 인터페이스를 구현해서 CGLIB 프록시의 실행 로직을 정의한다.
- JDK 동적 프록시를 설명할 때 예제와 거의 같은 코드이다.
- `Object target`: 프록시가 호출할 실제 대상
- `proxy.invoke(target, args)`: 실제 대상을 동적으로 호출한다.
  - 참고로 `method`를 사용해도 되지만, CGLIB는 성능상 `MethodProxy proxy`를 사용하는 것을 권장한다.

이제 테스트 코드로 CGLIB를 사용해보자.

CglibTest

```
package hello.proxy.cglib;

import hello.proxy.cglib.code.TimeMethodInterceptor;
import hello.proxy.common.service.ConcreteService;
import hello.proxy.common.service.ServiceImpl;
import hello.proxy.common.service.ServiceInterface;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.cglib.proxy.Enhancer;

@Slf4j
public class CglibTest {

    @Test
    void cglib() {
        ConcreteService target = new ConcreteService();

        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(ConcreteService.class);
        enhancer.setCallback(new TimeMethodInterceptor(target));
        ConcreteService proxy = (ConcreteService)enhancer.create();
        log.info("targetClass={}", target.getClass());
        log.info("proxyClass={}", proxy.getClass());

        proxy.call();
    }

}
```

`ConcreteService`는 인터페이스가 없는 구체 클래스이다. 여기에 CGLIB를 사용해서 프록시를 생성해보자.

- `Enhancer` : CGLIB는 `Enhancer`를 사용해서 프록시를 생성한다.
- `enhancer.setSuperclass(ConcreteService.class)` : CGLIB는 구체 클래스를 상속 받아서 프록시를 생성할 수 있다. 어떤 구체 클래스를 상속 받을지 지정한다.
- `enhancer.setCallback(new TimeMethodInterceptor(target))`
  - 프록시에 적용할 실행 로직을 할당한다.
- `enhancer.create()` : 프록시를 생성한다. 앞서 설정한 `enhancer.setSuperclass(ConcreteService.class)`에서 지정한 클래스를 상속 받아서 프록시가 만들어진다.

JDK 동적 프록시는 인터페이스를 구현(implement)해서 프록시를 만든다. CGLIB는 구체 클래스를 상속(extends)해서 프록시를 만든다.

실행 결과

```
CglibTest - targetClass=class hello.proxy.common.service.ConcreteService
CglibTest - proxyClass=class hello.proxy.common.service.ConcreteService$
$EnhancerByCGLIB$$25d6b0e3
TimeMethodInterceptor - TimeProxy 실행
ConcreteService - ConcreteService 호출
TimeMethodInterceptor - TimeProxy 종료 resultTime=9
```

실행 결과를 보면 프록시가 정상 적용된 것을 확인할 수 있다.

CGLIB가 생성한 프록시 클래스 이름

CGLIB를 통해서 생성된 클래스의 이름을 확인해보자.

`ConcreteService$$EnhancerByCGLIB$$25d6b0e3`

CGLIB가 동적으로 생성하는 클래스 이름은 다음과 같은 규칙으로 생성된다.

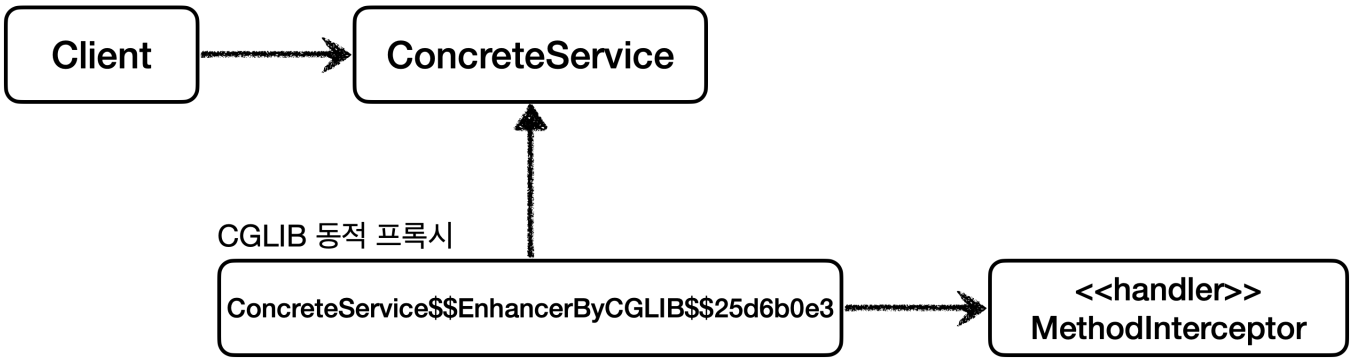
`대상클래스$$EnhancerByCGLIB$$임의코드`

참고로 다음은 JDK Proxy가 생성한 클래스 이름이다.

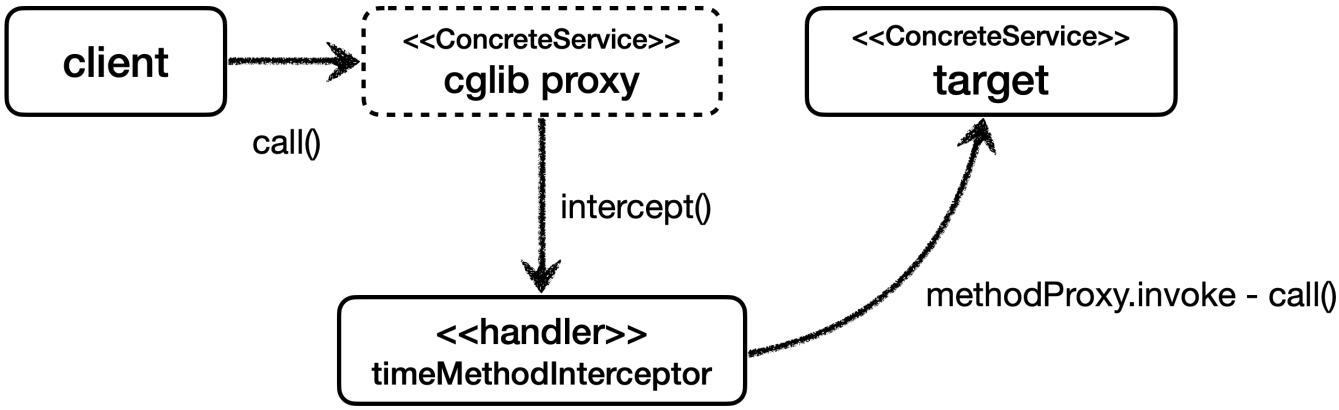
`proxyClass=class com.sun.proxy.$Proxy1`

그림으로 정리

클래스 의존 관계 - CGLIB



런타임 객체 의존 관계 - CGLIB



CGLIB 제약

- 클래스 기반 프록시는 상속을 사용하기 때문에 몇가지 제약이 있다.
  - 부모 클래스의 생성자를 체크해야 한다. ➜ CGLIB는 자식 클래스를 동적으로 생성하기 때문에 기본 생성자가 필요하다.
  - 클래스에 `final` 키워드가 붙으면 상속이 불가능하다. ➜ CGLIB에서는 예외가 발생한다.
  - 메서드에 `final` 키워드가 붙으면 해당 메서드를 오버라이딩 할 수 없다. ➜ CGLIB에서는 프록시 로직이 동작하지 않는다.

> **참고**
> CGLIB를 사용하면 인터페이스가 없는 V2 애플리케이션에 동적 프록시를 적용할 수 있다. 그런데 지금 당장 적용하기에는 몇가지 제약이 있다. V2 애플리케이션에 기본 생성자를 추가하고, 의존관계를 `setter`를 사용해서 주입하면 CGLIB를 적용할 수 있다. 하지만 다음에 학습하는 `ProxyFactory`를 통해서 CGLIB를 적용하면 이런 단점을 해결하고 또 더 편리하기 때문에, 애플리케이션에 CGLIB로 프록시를 적용하는 것은 조금 뒤에 알아보겠다.

정리

남은 문제

- 인터페이스가 있는 경우에는 JDK 동적 프록시를 적용하고, 그렇지 않은 경우에는 CGLIB를 적용하려면 어떻게 해야할까?
- 두 기술을 함께 사용할 때 부가 기능을 제공하기 위해서 JDK 동적 프록시가 제공하는 `InvocationHandler`와 CGLIB가 제공하는 `MethodInterceptor`를 각각 중복으로 만들어서 관리해야 할까?
- 특정 조건에 맞을 때 프록시 로직을 적용하는 기능도 공통으로 제공되었으면?