# Spring AOP

## Java Dynamic Proxy and CGLib

Jean Hu - 12 December 2016

# 前言

Spring 會依照不同設定實作 AOP，本文介紹其中的兩種實作機制 Java Dynamic Proxy 和 CGLib Proxy 及其限制。AOP (Aspect-Oriented Programming) 可將散落在各個商務流程中的共同邏輯集結成獨立可重用的服務，且不會與應用程式發生耦合，隨時可抽換。Spring 在 AOP 上的應用很多，會依據設定使用不同機制實現。目前支援三種機制：Java Dynamic Proxy、CGLib Proxy、AspectJ，本文利用簡單的應用介紹前面兩種Proxy機制。

# 目的

了解Spring在不同設定下對AOP的實作機制及其限制，利於專案初期規劃，避免日後重構。

# 開始前準備

本架構建立於以下版本的環境：

- JDK8
- STS 3.6.2.RELEASE
- Maven3.2.1 (STS 3.6.2中內建)

# Java Dynamic Proxy

一. 物件關聯

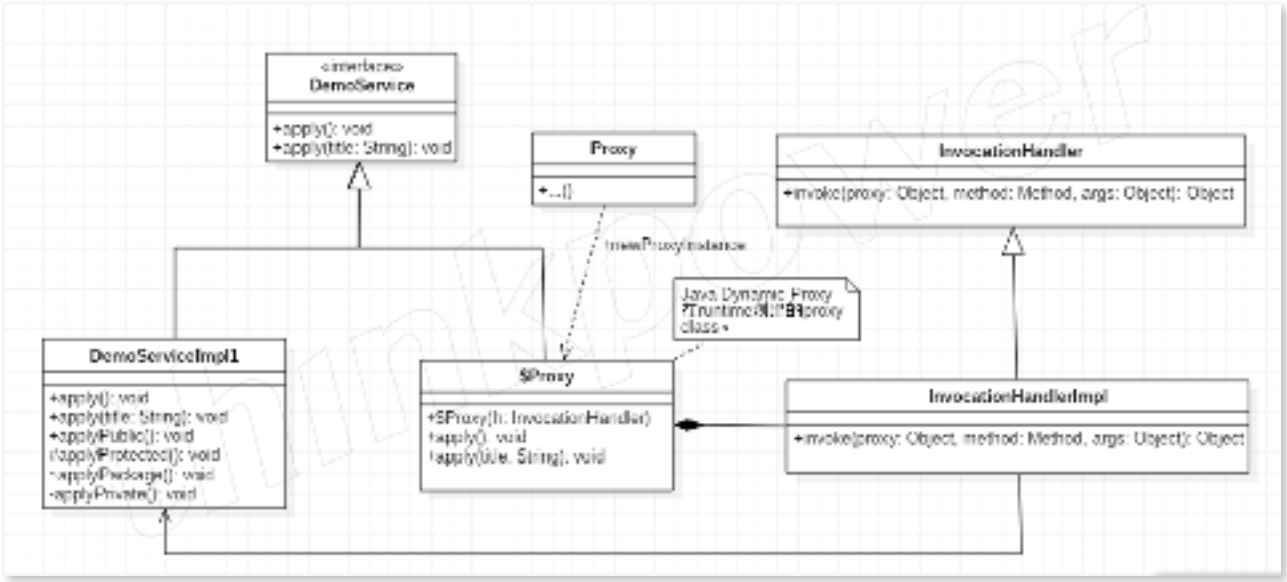


圖 1、Java Dynamic Proxy Class diagram

- Proxy (java.lang.reflect.Proxy) 在執行期創建實作 DemoService 介面的 $Proxy 物件。
- $Proxy 物件關聯至 InvocationHandlerImpl，進行攔截。
- InvocationHandlerImpl 需擁有被代理的 DemoServiceImpl1 物件，才能呼叫到需執行的功能。

二. 作業流程


圖 2、Java Dynamic Proxy Control flow

- 所有被 $Proxy 實作到的功能，呼叫時會先被 dispatch 到關聯的 InvocationHandlerImpl 中的 invoke。
- Invoke 執行加入的服務模組，再呼叫被代理的 DemoServiceImpl1 物件中原有功能。

三. 依照圖 1實現Demo程式

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public interface DemoService {
    void apply();
    void apply(String title);

    static void applyStatic() {
        Logger logger = LoggerFactory.getLogger(DemoService.class);
        logger.debug("I'm static method");
    }
}
```

Static Method
不能被繼承實作，
所以無法被Proxy攔截

圖 3、DemoService - 被代理的物件實作的介面


```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;

import com.codingcat.happy.service.DemoService;

@Service("demoServiceImpl1")
public class DemoServiceImpl1 implements DemoService {

    private static Logger logger = LoggerFactory.getLogger(DemoServiceImpl1.class);

    @Override
    public void apply() {
        logger.debug("In apply public access");
        // self-invocation won't be caught
        apply("Title");
    }

    @Override
    public void apply(String title) {
        logger.debug("In apply public access, title: {}", title);
    }

    public void applyPublic() {
        logger.debug("In apply public access");
    }

    protected void applyProtected() {
        logger.debug("In apply protected access");
    }

    void applyPackage() {
        logger.debug("In apply package access");
    }

    @SuppressWarnings("unused")
    private void applyPrivate() {
        logger.debug("In apply private access");
    }
}
```

圖 4、DemoServiceImpl1 - 被代理的物件


圖 5、JavaDynamicProxySample1 - 流程Demo

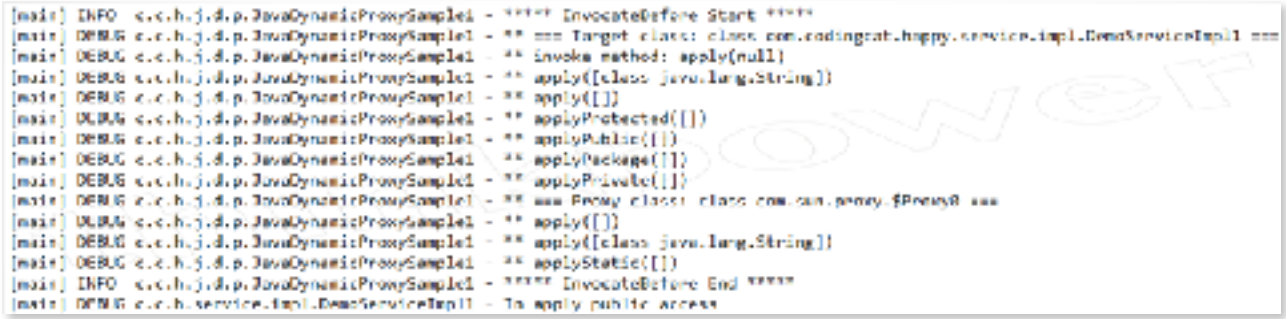

圖 6、Console log - JavaDynamicProxySample1 Run on Java Application

四. 歸納出Java Dynamic Proxy有以下幾點限制

- 無法攔截static method。
- 無法攔截未定義在介面上的method，換言之只能攔截public method。
- 無法攔截內部呼叫method。

## CGLib Proxy

一. 物件關聯

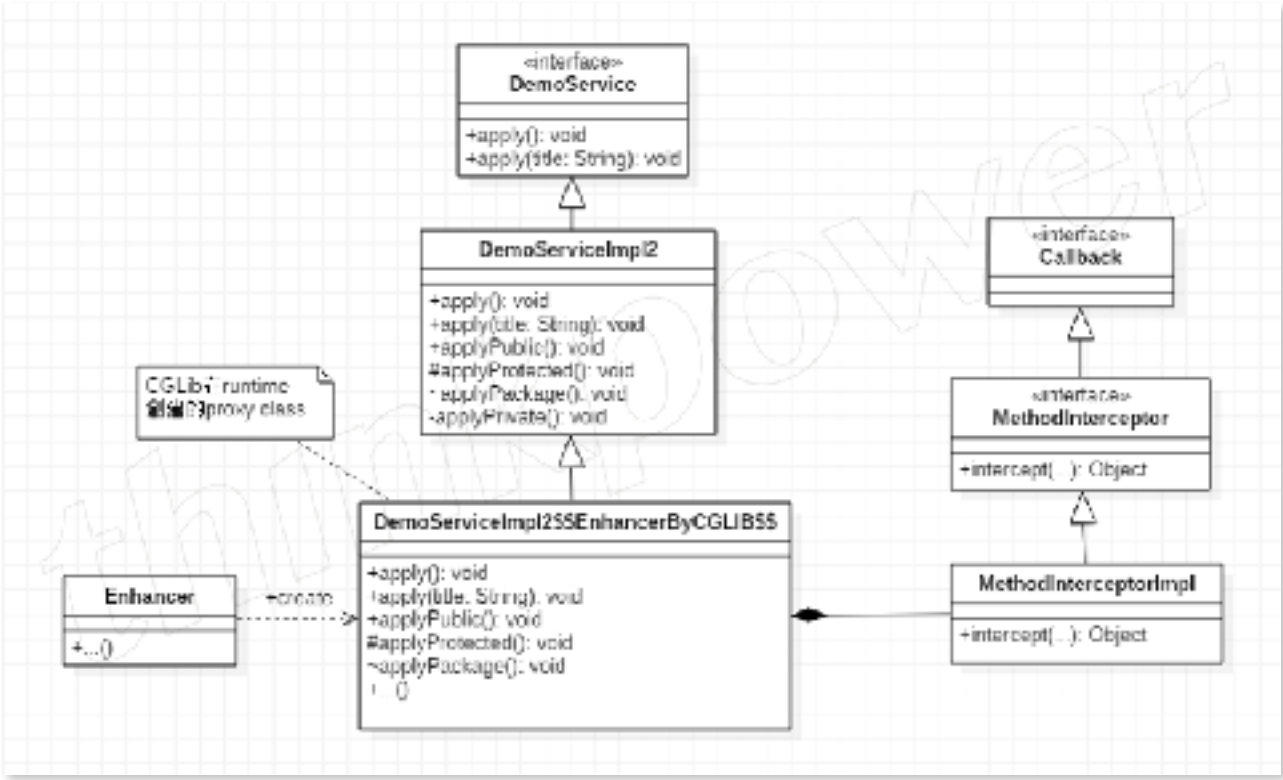


圖 7、CGLib Proxy Class diagram

- CGLib Proxy Enhancer (org.springframework.cglib.proxy.Enhancer) 在執行期創建繼承 DemoServiceImpl2 的 DemoServiceImpl2$$EnhancerByCGLIB$$ 物件。
- DemoServiceImpl2$$EnhancerByCGLIB$$ 物件關聯至 MethodInterceptorImpl，進行攔截。

二. 作業流程

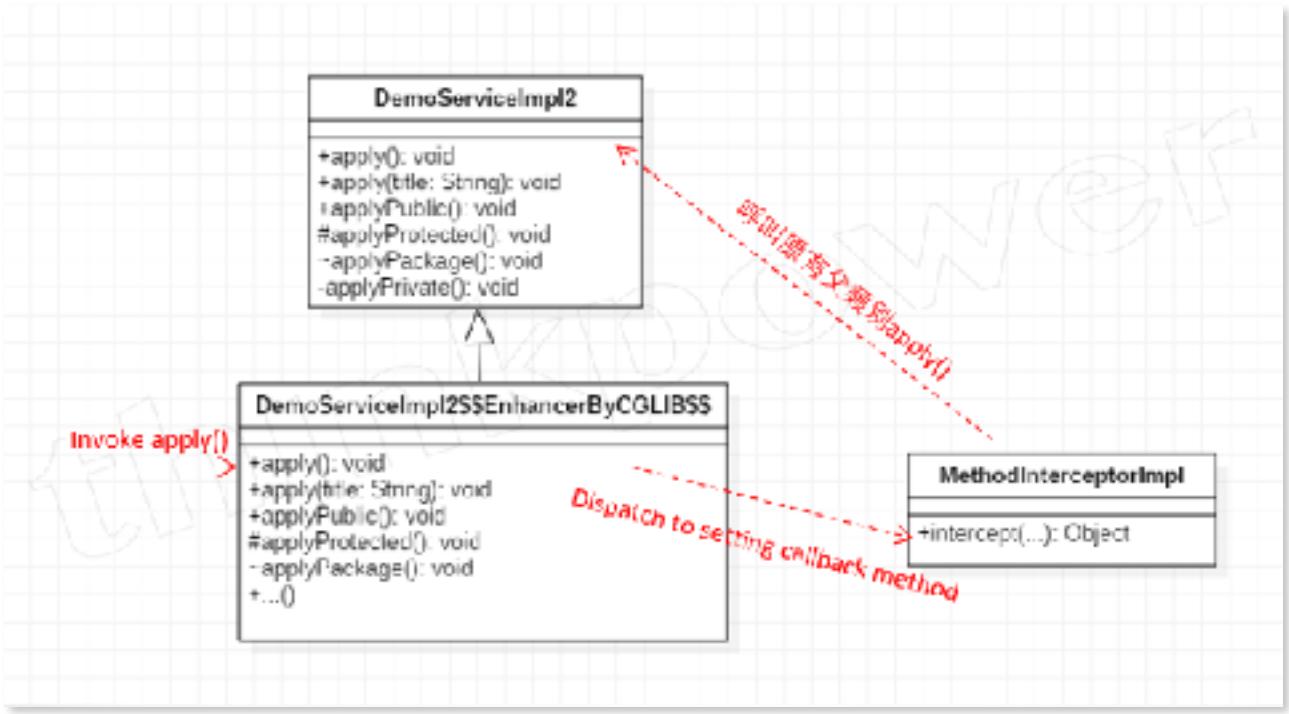


圖 8、CGLib Proxy Control flow

三. 依照圖 7 實現 Demo 程式


圖 9、DemoServiceImpl2 - 被代理的物件


圖 10、CGLibProxySample1 - 流程Demo

圖 11、Console log - CGLibProxySample2 Run on Java Application

四. 歸納出CGLib Proxy有以下幾點限制

- 無法攔截static method。
- 無法攔截無法繼承的method。
- 無法攔截內部呼叫method。

# Spring AOP 設定

一. Spring 要用哪一種AOP實作機制決定要素

- 若被攔截的 bean 有繼承介面，使用 Java Dynamic Proxy，反之被攔截 bean 未繼承任何介面，則會使用 CGLib Proxy。
- Java config 可以在 bean 上加 @Scope，強制使用何種 Proxy，如圖 9，DemoServiceImpl2 雖然有繼承 DemoService 介面，但是使用 CGLib Proxy 實作。
- Xml config 可以加上 scoped-proxy 或 proxy-target-class 屬性來決定使用何種方式實作。

二. Demo程式

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.codingcat.happy</groupId>
    <artifactId>demo</artifactId>
    <version>1.0.0</version>
    <packaging>war</packaging>

    <name>springAOP_sample1</name>
    <description>Demo project for Spring AOP</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.2.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

圖 12、pom.xml - maven設定

```java
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class PointcutDefinition {

    @Pointcut("within(@org.springframework.stereotype.Service *)")
    public void serviceLayer() {

    }
}
```

圖 13、PointcutDefinition - Pointcut定義

```java
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.reflect.MethodSignature;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggerAspect {

    private static Logger logger = LoggerFactory.getLogger(LoggerAspect.class);

    @Before(value = "PointcutDefinition.serviceLayer()")
    public void logBefore(JoinPoint joinPoint) {
        logger.info("logBefore Start");

        MethodSignature signature = (MethodSignature)joinPoint.getSignature();
        logger.debug("=== Signature class: {} ===", signature.getDeclaringType().getName());
        logger.debug("{}({})", signature.getMethod().getName(), joinPoint.getArgs());

        Object targetBean = joinPoint.getTarget();
        logger.debug("=== Target class: {} ===", targetBean.getClass());
        Arrays
            .stream(targetBean.getClass().getDeclaredMethods())
            .forEach(x -> logger.debug("{}({})", x.getName(), x.getParameterTypes()));

        Object thisBean = joinPoint.getThis();
        logger.debug("=== This class: {} ===", thisBean.getClass());

        Arrays
            .stream(thisBean.getClass().getDeclaredMethods())
            .filter(x -> x.getName().contains("apply"))
            .forEach(x -> logger.debug("{}({})", x.getName(), x.getParameterTypes()));

        logger.info("logBefore End");
    }

}
```



圖 14、LoggerAspect - 定義需要動態加入的服務

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

import com.codingcat.happy.service.DemoService;
import com.codingcat.happy.service.impl.DemoServiceImpl2;

@SpringBootApplication(scanBasePackages = {
                        "com.codingcat.happy.service.impl",
                        "com.codingcat.happy.aop.aspect"
                       })
public class SpringAopSample1Application {

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
                SpringApplication.run(SpringAopSample1Application.class, args);

        // Demo1: Use Java proxy
        DemoService demoService1 = context.getBean("demoServiceImpl1", DemoService.class);
        demoService1.apply();

        // Demo2: Use CGLib proxy
        DemoServiceImpl2 demoService2 = context.getBean(DemoServiceImpl2.class);
        demoService2.applyPublic();
    }
}
```



圖 15、SpringAopSample1Application - Demo Spring AOP use Spring Boot

- DemoService 參考圖 3。
- DemoServiceImpl1 參考圖 4。
- DemoServiceImpl2 參考圖 9。


圖 16、Console log - SpringAopSample1Application Run on Java Application

# Proxy 機制限制及解決辦法

一. Proxy機制限制

Proxy機制是在執行期間創建一個Proxy類別，所以有以下限制：

- 無法攔截不能繼承的 method。
- Java Dynamic Proxy 為介面繼承，只能繼承 public method。
- CGLib 為類別繼承，不能繼承 private or static method。
- 無法攔截內部呼叫 method (沒有經過 Proxy 類別)。

二. 解決辦法

使用AspectJ進行程式碼的織入，支援compile time、load time weaving。

# 參考來源


- AOP概念 - http://openhome.cc/Gossip/SpringGossip/AOPConcept.html
- Java Dynamic Proxy - https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html
- CGLib Proxy - https://github.com/cglib/cglib
- Spring AOP - http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html
- Spring proxy pitfalls - http://www.nurkiewicz.com/2011/10/spring-pitfalls-proxying.html