40 Spring AOP策略模式使用及示例实战

更新时间：2020-08-24 10:02:21

> 只有在那崎岖的小路上不畏艰险奋勇攀登的人,才有希望达到光辉的顶点。——马克思

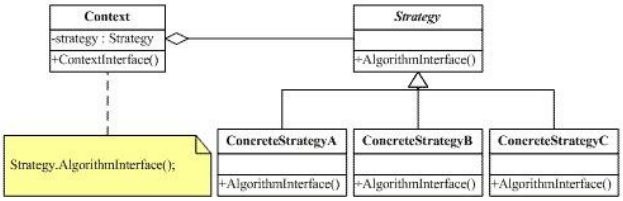
背景

在现实生活中常常遇到实现某种目标存在多种策略可供选择的情况，例如，出行旅游可以乘坐飞机、乘坐火车、骑自行车或自己开私家车等，支付可以采用支付宝、微信、银行卡等方法。

在软件开发中也常常遇到类似的情况，当实现某一个功能存在多种算法或者策略，我们可以根据环境或者条件的不同选择不同的算法或者策略来完成该功能。如果使用 if-else 或者 case-switch 语句实现，不但语句变得很复杂，而且增加、删除或更换算法要修改原代码，不易维护，违背开闭原则。如果采用策略模式就能很好解决该问题。

策略模式

> The Strategy Pattern defines a family of algorithms,encapsulates each one,and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it



策略模式UML图

策略模式组成：

- **抽象策略角色**： 策略类，通常由一个接口或者抽象类实现；
- **具体策略角色**：包装了相关的算法和行为；
- **环境角色**：持有一个策略类的引用，最终给客户端调用。

Spring AOP策略模式使用实例

目标类：

```
package com.davidwang456.test;
public class HelloService {
    public void sayHello(String world) {
        System.out.println("hello "+ world);
    }
}
```

切面类：

```
package com.davidwang456.test;

import org.aspectj.lang.annotation.Aspect;

@Aspect
public class LogAspect {
    @Before("execution(* sayHello(..))")
    public void beforeHello() {
        System.out.println("How are you !");
        throw new NullPointerException();
    }
}
```

测试类：

```java
package com.davidwang456.test;

import org.springframework.aop.aspectj.annotation.AspectJProxyFactory;

public class AspectTest {
    public static void main(String[] args) {
        HelloService target=new HelloService();
        AspectJProxyFactory factory=new AspectJProxyFactory();
        factory.setTarget(target);
        factory.addAspect(LogAspect.class);
        HelloService proxy=factory.getProxy();
        proxy.sayHello("world !");
    }

}
```

运行结果：

> How are you !
>
> hello world !

深入Spring AoP策略模式内部原理

为了对spring aop的实现有个全面的了解，决定在代码中打印出调用链。

```java
package com.davidwang456.test;
public class HelloService {
    public void sayHello(String world) {
        StackUtils.getStack();
        System.out.println("hello "+ world);
    }

}
```

此时，控制台打印出调用链如下：

```
how are you !
调用序号：1 调用类和方法 com.davidwang456.test.AspectTest$main
调用序号：2 调用类和方法 com.davidwang456.test.HelloService$$EnhancerBySpringCGLIB$$2cdb38ef$sayHello
调用序号：3 调用类和方法 org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor$intercept
调用序号：4 调用类和方法 org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation$proceed
调用序号：5 调用类和方法 org.springframework.aop.framework.ReflectiveMethodInvocation$proceed
调用序号：6 调用类和方法 org.springframework.aop.interceptor.ExposeInvocationInterceptor$invoke
调用序号：7 调用类和方法 org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation$proceed
调用序号：8 调用类和方法 org.springframework.aop.framework.ReflectiveMethodInvocation$proceed
调用序号：9 调用类和方法 org.springframework.aop.framework.adapter.MethodBeforeAdviceInterceptor$invoke
调用序号：10 调用类和方法 org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation$proceed
调用序号：11 调用类和方法 org.springframework.aop.framework.ReflectiveMethodInvocation$proceed
调用序号：12 调用类和方法 org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation$invokeJoinpoint
调用序号：13 调用类和方法 org.springframework.cglib.proxy.MethodProxy$invoke
调用序号：14 调用类和方法 com.davidwang456.test.HelloService$$FastClassBySpringCGLIB$$c597119c$invoke
调用序号：15 调用类和方法 com.davidwang456.test.HelloService$sayHello
调用序号：16 调用类和方法 com.davidwang456.test.StackUtils$getStack
调用序号：17 调用类和方法 java.lang.Thread$getAllStackTraces
hello world
```

- 从调用序号：3 可以看到，本次调用使用的是 CglibAopProxy 生成的 java 代理类；
- LogAspect 是切面，@Before 注解触发 MethodBeforeAdviceInterceptor 的 invoke() 方法
- Spring 使用 MethodProxy 代理生成 HelloService 的代理类

可能会有人问：Spring AOP的策略模式实现有两种，不是默认使用 JdkDynamicAopProxy，这个程序为什么使用了 CglibAopProx？是不是搞错了？

程序是不会骗人的，我们来看看为什么使用了 CglibAopProx 而非 JdkDynamicAopProxy。

在策略模式中，CglibAopProx 和 JdkDynamicAopProxy 两种实现被封装到 DefaultAopProxyFactory 中，创建代理的方式如下：


```
/**
 * Default {@link AopProxyFactory} implementation, creating either a CGLIB proxy
 * or a JDK dynamic proxy.
 *
 * <p>Creates a CGLIB proxy if one the following is true for a given
 * {@link AdvisedSupport} instance:
 * <ul>
 * <li>the {@code optimize} flag is set
 * <li>the {@code proxyTargetClass} flag is set
 * <li>no proxy interfaces have been specified
 * </ul>
 *
 * <p>In general, specify {@code proxyTargetClass} to enforce a CGLIB proxy,
 * or specify one or more interfaces to use a JDK dynamic proxy.
 *
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @since 12.03.2004
 * @see AdvisedSupport#setOptimize
 * @see AdvisedSupport#setProxyTargetClass
 * @see AdvisedSupport#setInterfaces
 */
@SuppressWarnings("serial")
public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {

    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine target class: " +
                        "Either an interface or a target is required for proxy creation.");
            }
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }

    /**
     * Determine whether the supplied {@link AdvisedSupport} has only the
     * {@link org.springframework.aop.SpringProxy} interface specified
     * (or no proxy interfaces specified at all).
     */
    private boolean hasNoUserSuppliedProxyInterfaces(AdvisedSupport config) {
        Class<?>[] ifcs = config.getProxiedInterfaces();
        return (ifcs.length == 0 || (ifcs.length == 1 && SpringProxy.class.isAssignableFrom(ifcs[0])));
    }

}
```

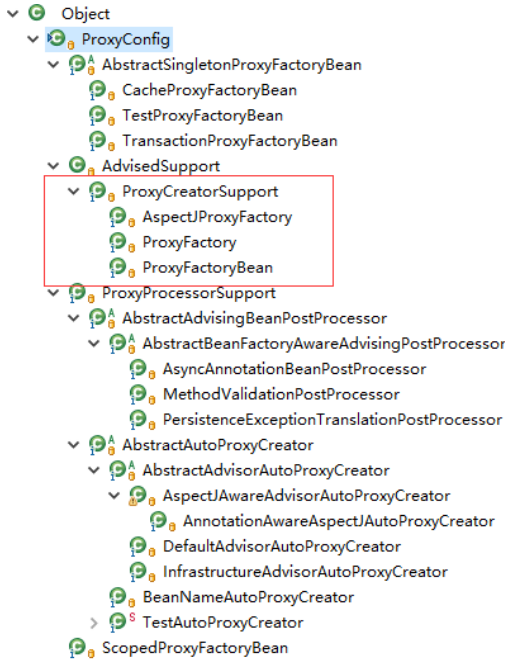
使用 CGLIG 的 3 种情况：

- ProxyConfig 中的 optimize 标识被置为 true；
- ProxyConfig 中的 proxyTargetClass 标识被置为 true；
- 目标类没有可用的代理接口即目标类没有实现接口。

HelloService 是一个具体类，并没有实现任何接口，故满足使用 CGLIB 的条件。

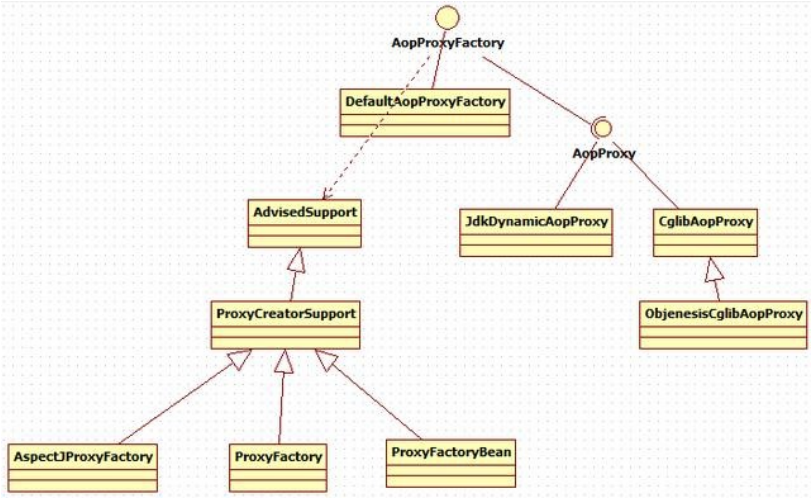
如果实现了接口的类也想使用 CGLIB 来生成代理类，可以通过 ProxyConfig 配置来改变，它的使用也很方便，因为

AoPProxy 工厂类实现类 AspectJProxyFactory，proxyFactory，ProxyFactoryBean 都间接实现了 ProxyConfig 类。



策略模式的拓展：策略模式+工厂模式=策略工厂模式

在实际开发中，单独使用一种模式可能无法更完美解决问题，需要多种模式结合使用。



分析角色

- 抽象策略角色：AopProxy；
- 具体策略角色：CglibAopProx 和 JdkDynamicAopProxy；
- 工厂类：ProxyCreatorSupport 类实现 AspectJProxyFactory，proxyFactory，ProxyFactoryBean。

总结

很多开发者习惯了使用 XML 配置方式：

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">
    <aop:aspectj-autoproxy />
    <context:component-scan base-package="com.davidwang456.test" />
    <bean id="loggingAspect" class="com.davidwang456.test.EmployeeCRUDAspect" />
</beans>
```

或者注解方式：

```java
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EmployeeCRUDAspect {

    @Before("execution(* EmployeeManager.getEmployeeById(..))")
    public void logBeforeV1(JoinPoint joinPoint)
    {
        System.out.println("EmployeeCRUDAspect.logBeforeV1() : " + joinPoint.getSignature().getName());
        throw new NullPointerException();
    }

    @Before("execution(* EmployeeManager.*(..))")
    public void logBeforeV2(JoinPoint joinPoint)
    {
        System.out.println("EmployeeCRUDAspect.logBeforeV2() : " + joinPoint.getSignature().getName());
    }

    @After("execution(* EmployeeManager.getEmployeeById(..))")
    public void logAfterV1(JoinPoint joinPoint)
    {
        System.out.println("EmployeeCRUDAspect.logAfterV1() : " + joinPoint.getSignature().getName());
    }

    @After("execution(* EmployeeManager.*(..))")
    public void logAfterV2(JoinPoint joinPoint)
    {
        System.out.println("EmployeeCRUDAspect.logAfterV2() : " + joinPoint.getSignature().getName());
    }
}
```

会忘记程序怎么写或者内部的原理是什么,这样在面试的适合非常吃亏，因为别人也知道 XML 或者 Annotation 的方式，你没有深入进去，自然就没有亮点。

本文从策略模式入手，详细阐述了策略模式在Spring AOP 的应用，并详解了Spring AOP 的主要类及使用方法，值得一看。

}

39 图说AOP---妈妈再也不担心我概念混淆了

41 注解 @Aspect 是如何工作的？