23_JDK 动态代理和 CGLib 动态代理两种模式介绍

1、开篇

前面三节课主要是围绕 Spring IOC 进行讲解，包括：依赖查找、依赖注入以及依赖注入来源。从本节课开始的两节课会围绕 Spring AOP 进行讲解。本节课会介绍 Spring AOP 的基本概念和实现原理，包括如下内容：

- AOP 基本概念
- 为什么要使用 AOP
- JDK 动态代理
- GCLIB 动态代理
- JDK 代理和 GCLIB 代理的区别

2、AOP 基本概念

AOP （Aspect Orient Programming）,直译过来就是 面向切面编程。AOP 是一种编程思想，是面向对象编程（OOP）的一种补充。面向对象编程将程序抽象成各个层次的对象，而面向切面编程是将程序抽象成各个切面。

如图 1 所示，切面实现了横切关注点，也就是跨多个应用对象的逻辑。假设业务服务 1、2、3 各自实现的业务不仅相同，但是有些系统级别的应用是可以抽象的，例如蓝色箭头中标注的，日志、安全、事务。

所谓切面，相当于应用对象间的横切点，可以将其抽象为单独的模块。图 1 中的切面就是日志、安全、事务，换句话说就是可以将日志、安全、事务抽象成单独的模块。

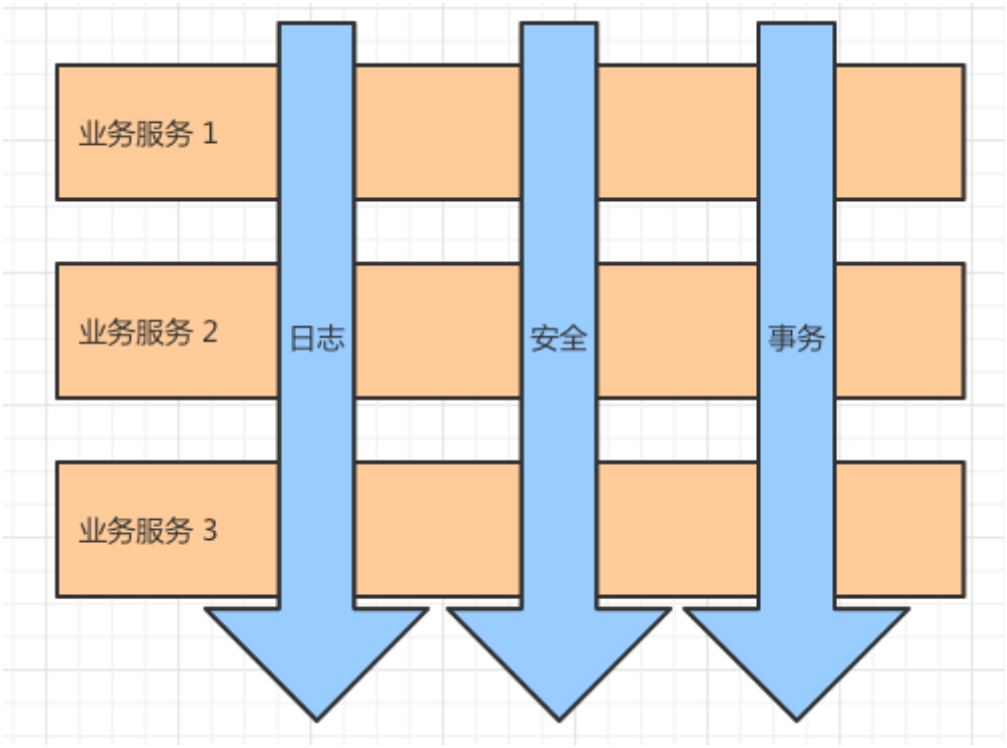

图 1 切面实现了横切关注点

3、为什么要使用 AOP

在实际开发过程中，业务代码中会不断重复的代码，例如：记录日志、判断权限等。遇到这种场景就可以将这些与业务无关的代码进行抽象成一个方法，在业务代码中去调用这些方法。这种问题就可以利用 AOP 来解决。

AOP 可以保证开发者不修改源代码的前提下，为业务服务的组件添加通用功能。其本质是通过 AOP 框架将通用代码织入到业务组件的方法中。

按照 AOP 框架修改源代码的时机，可以将其分为两类：

静态 AOP 实现， AOP 框架在编译阶段对程序源代码进行修改，生成了静态的 AOP 代理类（生成的 *.class 文件已经被改掉了，需要使用特定的编译器），比如 AspectJ。

动态 AOP 实现， AOP 框架在运行阶段对动态生成代理对象（在内存中以 JDK 动态代理，或 CGlib 动态地生成 AOP 代理类），如 Spring AOP。

如图 2 所示，这里列出了 AOP 的几大类别，其中以动态 AOP 的两个类别使用最为广泛，它们就是 JDK 动态代理和 CGLIB 代理方式。接下来就让我们一起来看看它们是如何实现 AOP 的吧。

类别	机制	原理	优点	缺点
静态 AOP	静态织入	在编译起，切面直接以字节码的形式编译到目标字节码文件中	对系统无性能影响	灵活性不够
动态 AOP	JDK 动态代理	在运行期，目标类加载后，为接口动态生成代理类，将切面织入到代理类中	相对于静态 AOP 更加灵活	切入的关注点需要实现接口。对系统有一点性能影响
动态字节码生成	CGLIB	在运行期，目标类加载后，动态生成目标类的子类，将切面逻辑加入到子类中	没有接口也可以织入	扩展类的实例方法用 final 修饰时，则无法进行织入
自定义类加载器		在运行期，目标类加载前，将切面逻辑加到目标字节码里	可以对绝大部分类进行织入	代码中如果使用了其他类加载器，则这些类将不会织入
字节码转换		在运行期，所有类加载器加载字节码前进行拦截	可以对所有类进行织入	

图 2 AOP 类别

4、JDK 动态代理

AOP 框架有很多种， Spring 中的 AOP 是通过动态代理实现的。这里就来介绍 JDK 动态代理的实现原理。JDK 动态代理是在运行期，目标类加载后，为接口动态生成代理类，将切面织入代理类中。

这里通过一个例子来讲解 JDK 动态代理的原理。如图 1 所示，定义 Service 接口，其中有一个 help 方法。用 ServiceImpl 去实现 Service 接口，在 help 方法中实现“打印买书”的输出。这个 ServiceImpl 就是我们要代理的目标类。

```java
public interface Service {
    public void help();
}
public class ServiceImpl implements Service {
    @Override
    public void help() {
        System.out.println("买书");
    }
}
```

图 3 定义目标类和接口

接下来实现代理类，如图 4 所示，DynamicProxy 作为动态代理类实现了 InvocationHanlder 接口。其中定义了 Service 的接口作为代理类 DynamicProxy 构造方法的输入参数。在 Override 的 invoke 方法中通过参数 Method 来代理要执行的代理类中的方法，method.invoke 的输入参数为目标类 Service。method.invoke 是用来执行代理类中的 help 方法，如果这个假设成立的话，在这个方法前面执行的 System.out.println(“买书之前”)，会在“买书”方法之前执行，同理 System.out.println(“买书之后”)会在“买书”方法之后执行。

```java
public class DynamicProxy implements InvocationHandler {
    //代理的真实对象
    private Object service;
    //构造方法，给代理的真实对象赋值
    public DynamicProxy(Object service) {
        this.service = service;
    }
    @Override
    public Object invoke(Object object, Method method, Object[] args)
            throws Throwable {
        //真实方法之前执行
        System.out.println("买书之前");
        //调用真实方法
        method.invoke(service, args);
        //真实方法之后执行
        System.out.println("买书之后");

        return null;
    }
}
```

图 4 代理类编写

回过头我们来测试一下 JDK 的代理类是否工作，如图 5 所示，在 main 函数中通过 Proxy 的 newProxyInstance 方法传入 ClassLoader 的类加载器，同时传入 Service 的接口，以及代理类 DynamicProxy（以 InvocationHanlder 的形式）。然后执行代理类的 help 方法。

```java
public class Client {
    public static void main(String[] args) {
        //要代理的真实对象
        Service service = new ServiceImpl();
        //要代理哪个真实对象，就将该对象传进去，最后是通过该真实对象来调用其方法
        InvocationHandler handler = new DynamicProxy(service);
        //添加以下的几段代码，就可以将代理生成的字节码保存起来
        try {
            Service serviceProxy = (Service) Proxy.newProxyInstance(service.getClass().getClassLoader(),
                    service.getClass().getInterfaces(), handler);
            serviceProxy.help();
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }

    }
}
```

图 5 测试 JDK 代理类

执行上图代码之后，会发现如下输出，表明在调用 ServiceImpl 类的时候，虽然只执行了其中的 help 方法，该方法输出为“买书”。但同时通过代理类 DynamicProxy 的织入方式，把“买书之前”和“买书之后”两端代码放在了 help 方法的前后，完成面线切面的织入。

买书之前

买书

买书之后

5、GCLIB 动态代理

CGLIB(Code Generation Library)是一个开源项目，是一个强大的，高性能，高质量的 Code 生成类库，它可以在运行期扩展 Java 类与实现 Java 接口。其原理是在运行期间，通过字节码的方式在目标类生成的子类中织入对应的代码完成代理。还是通过一个例子来看，如图 6 所示，依旧定义一个 Biz 的目标类，里面有一个 help 方法打印“买书”。我们要对这个方法进行织入，在方法的前后加入“买书之前”和“买书之后”的打印语句。定义一个 BizTntercepter 作为代理类，实现了 MethodIntercepter 的接口，并且 Override intercept 方法。

介绍一下方法的参数， Object 是生成的子类对象，Method 是要代理目标类的方法，Object[]是参数，MethodProxy 子类生成的代理方法。通过 methodProxy.invokeSuper 方法执行生成子类的代理方法，第一个输入是目标类生成的子类，第二个输入是参数，该方法就是调用目标类 biz 中的 help 方法。

最后就是执行测试类的方法了，在 main 函数中 new 一个 enhancer，通过 setSuperclass 指定目标类，通过 setCallback 方法指定代理类，最后使用通过 create 方法生成 Biz 类的实例，并且执行对应的 help 方法。

```java
public class Biz {                                   目标类
    public void help() {
        System.out.println("买书");
    }
}

public class BizInterceptor implements MethodInterceptor {        代理类
    //Object 是生成的子类对象，Method是要代理目标类的方法，Object[]是参数，MethodProxy子类生成的代理方法
    @Override
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {
        System.out.println("买书之前");
        methodProxy.invokeSuper(o, objects);
        System.out.println("买书之后");
        return null;
    }
}

public class BizCglibClient {
    public static void main(String[] args) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(Biz.class);
        enhancer.setCallback(new BizInterceptor());
        Biz biz = (Biz)enhancer.create();
        biz.help();

    }
}
```

图 6 CGLIB 代理

执行完毕以后与 JDK 的结果一致，显示在“买书”的前后分别加入了“买书之前”和“买书之后”的打印语句。

买书之前

买书

买书之后

6、JDK 代理和 CGLIB 代理的区别

1、介绍完两种动态代理模式，这里稍微总结一下两者的区别。

- JDK 动态代理只能对实现了接口的类生成代理，而不能针对类
- CGLib 是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法（继承）

2、Spring AOP 选择用 JDK 还是 CGLib 的依据

- 当 Bean 实现接口时，Spring 就会用 JDK 的动态代理。

- 当 Bean 没有实现接口时，Spring 使用 CGLib 来实现。
- 默认是选择 JDK 代理，可以强制使用 CGLib（在 Spring 配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）。

3、JDK 和 CGLib 的性能对比

使用 CGLib 实现动态代理，CGLib 底层采用 ASM 字节码生成框架，使用字节码技术生成代理类，在 JDK1.6 之前比使用 Java 反射效率要高。唯一需要注意的是，CGLib 不能对声明为 final 的方法进行代理，因为 CGLib 原理是动态生成被代理类的子类。

7、总结

今天开始介绍 Spring AOP 的内容，首先从 AOP 的基本概念入手，阐明了为什么要使用 AOP，并且提出了 AOP 静态和动态代理的概念。然后，深入讲解了 Spring AOP 使用了动态代理的两种方式：JDK 和 CGLIB，并且举例说明，最后总结了二者的区别。下节课会告诉大家 Spring AOP 中具体使用的方法，包括：Aspect、Join Points、Pointcuts 和 Advice 语法和特性。下期见，拜拜。