**SQL映射框架MyBatis源码分析**

主讲：Cat 老师

北京动力节点教育科技有限公司  
2020 • 北京

动力节点•版权所有•禁止传播

## 源码分析

第一步：

*//第一步：读取mybatis-config.xml配置文件*InputStream inputStream = Resources.*getResourceAsStream*(**"mybatis-config.xml"**);

通过Resources工具类，调用ClassLoader读取classpath下的**mybatis-config.xml**配置文件，得到一个输入流；

第二步：（有关配置文件的解析与处理都在该步骤中）

*//第二步：构建SqlSessionFactory*SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

1、创建一个SqlSessionFactoryBuilder对象：

**new** SqlSessionFactoryBuilder()；

2、调用SqlSessionFactoryBuilder对象的build方法：

build(InputStream inputStream)

3、创建XMLConfigBuilder 对象：

XMLConfigBuilder parser = **new** XMLConfigBuilder(inputStream, environment, properties);

4、创建XMLMapperEntityResolver对象：

**new** XMLMapperEntityResolver()

1. 创建XPathParser对象：

**new** XPathParser(inputStream, **true**, props, **new** XMLMapperEntityResolver())

1. 创建XPath对象：

XPathFactory factory = XPathFactory.*newInstance*();  
**this**.**xpath** = factory.newXPath();

1. 创建Document对象：（DOM解析）

*//JDK提供的文档解析工厂对象*DocumentBuilderFactory factory = DocumentBuilderFactory.*newInstance*();

*//创建一个DocumentBuilder对象*DocumentBuilder builder = factory.newDocumentBuilder();

*//解析输入源的xml数据为一个Document文件***return** builder.parse(inputSource);

8、创建Configuration对象

**new** Configuration()

注册类型别名

初始化Configuration对象的一些成员变量的默认值

9、通过XPath方法解析mybatis-config.xml文件：

parseConfiguration(**parser**.evalNode(**"/configuration"**));

将解析出来的信息封装到Configuration对象中；

10、通过持有Configuration对象创建一个DefaultSqlSessionFactory对象：

**new** DefaultSqlSessionFactory(configuration);

总结：

MyBatis解析配置文件的本质就是为了获得Configuration对象；

Configuration 对象可以理解是mybatis的XML配置文件在程序中的化身，是MyBatis非常重要的一个对象，里面封装了MyBatis的整个配置信息；

第三步：

*//第三步：打开SqlSession*Session session = sqlSessionFactory.openSession();

1、从configuration配置对象中获取环境environment信息；

2、根据环境environment信息获取事务工厂TransactionFactory；

3、根据环境信息中创建一个事务对象Transaction；

4、根据configuration和Transaction创建一个执行器Executor对象；

5、根据configuration和Executor创建一个默认的SqlSession对象；

第四步：

*//第四步：获取Mapper接口对象*UUserInfoMapper uUserInfoMapper = session.getMapper(UUserInfoMapper.**class**);

1、从configuration中获取Mapper接口对象；

2、从mapperRegistry中获取Mapper接口对象；

3、从knownMappers这个Map中获取MapperProxyFactory代理工厂；

4、通过MapperProxyFactory代理工厂创建MapperProxy实例；

5、JDK动态代理生成Mapper接口的代理对象；

第五步：

*//第五步：调用Mapper接口对象的方法操作数据库；*UUserInfo uUserInfo = uUserInfoMapper.selectByPrimaryKey(1);

## 代理模式

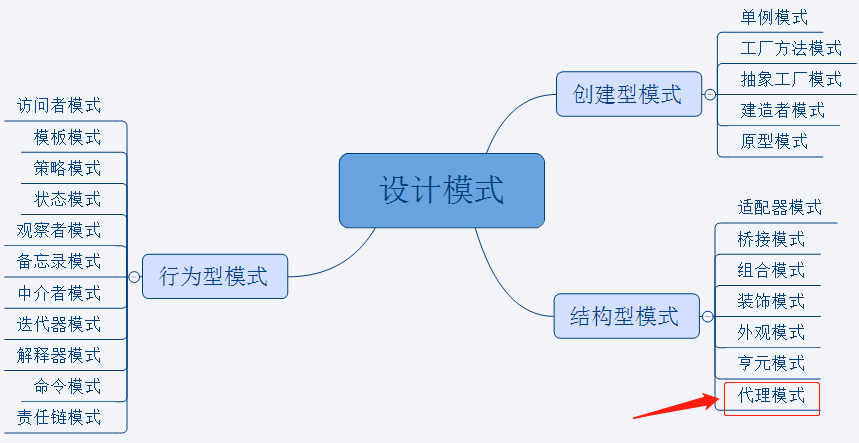
代理模式是23种设计模式中的一种；

代理模式就是给一个对象提供一个代理，就像我们生活中的中介（租房、留学、相亲），由代理对象控制对原目标对象的访问，所以在代理对象中就可以实现对目标对象功能的增强、扩展、甚至改写；

为什么需要代理？

因为一个良好的设计不应该轻易的修改，这正是开闭原则的体现：一个良好的设计应该对修改关闭，对扩展开放。而代理正是为了扩展类而存在的，它可以控制对现有类（也就是被代理的类）的访问，通俗来说就是可以拦截对现有类方法的调用并做相应处理；

代理模式应用场景：aop、权限控制、服务监控、缓存、日志、限流、事务、拦截过滤 等；



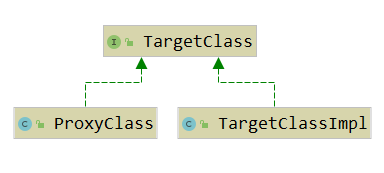
代理模式分为 静态代理 和 动态代理；

## 静态代理

静态代理就是我们自己手写代理类； aspectJ静态代理（编译期生成代理类）

静态代理可以实现在不修改目标对象代码的前提下，对目标对象的功能进行扩展；

静态代理：（目标接口、目标接口实现、代理类）



**优点：**可以实现不对目标对象进行修改的前提下，对目标对象进行功能的扩展和增强，也就是扩展原功能，不污染原代码。

**缺点：**因为代理对象，需要实现与目标对象一样的接口，如果目标接口类繁多，也会导致代理类繁多，另外一旦目标接口增加新方法，则代理类也需要维护；

## 动态代理

动态代理四种方案：JDK、CGLIB、Javassist、ASM（编码特别复杂,比较少用）；

## JDK动态代理

动态代理，是指在运行期动态的为指定的类生成其代理类；

JDK动态代理会在程序运行时生成一个$Proxy0.Class，该class类在代码中看不到，在磁盘中也没有保存，是程序运行时候在jvm内存中生成的，所以我们感觉动态代理很抽象，要想保存该class文件，我们可以在代码中设置：

System.*setProperty*(**"sun.misc.ProxyGenerator.saveGeneratedFiles"**, **"true"**);

这样就会在运行时候将该class文件保存到我们磁盘上；

动态代理在运行时将接口中声明的所有方法都转移到一个集中的InvocationHandler.invoke()方法进行处理，这样在接口方法数量比较多的时候，我们可以进行灵活处理，而静态代理需要在每一个方法进行中转；

动态代理是AOP思想的底层实现，当然JDK的动态代理，目标类必须实现某个接口，如果某个类没有实现接口则不能生成代理对象；

## Cglib动态代理

官方Github：<https://github.com/cglib>

CGLib (Code Generation Library) 是一个强大、高性能、高质量的代码生成库，它可以在运行时扩展JAVA类并实现接口；

字节码生成库是生成和转换Java字节码的高级API，它被AOP、测试、数据访问框架用来生成动态代理对象和拦截字段访问；



Rafael Winterhalter  
*来自德国，软件顾问，居住挪威的奥斯陆，也是hibernate的成员之一*

CGLib 比 Java 的 java.lang.reflect.Proxy 类更强大的地方在于它不仅可以接管接口类的方法，还可以接管普通类的方法，为JDK的动态代理提供了很好的补充，通常可以使用Java的动态代理创建代理，但当要代理的类没有实现接口时，那么CGLIB是一个更好的选择；

在一些开源框架中都采用了cglib：

Hibernate

Spring

Guice

CGLib 的底层是Java字节码操作框架 —— ASM <https://asm.ow2.io>

## CGLIB类库结构

net.sf.cglib.core: 底层字节码处理类，他们大部分与ASM有关；

net.sf.cglib.transform: 编译期或运行期类和类文件的转换

net.sf.cglib.proxy: 实现创建代理和方法拦截器的类

net.sf.cglib.reflect: 实现快速反射类

net.sf.cglib.util: 集合排序等工具类

net.sf.cglib.beans: javabean相关的工具类

## CGLIB实现动态代理

使用CGLib实现动态代理，完全不受代理类必须实现接口的限制，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JDK8之前CGLib动态比JDK的动态代理（使用Java反射）效率要高。

唯一需要注意的是，如果被代理的类被final修饰，那么它不可被继承，即不可被代理，同样，如果被代理的类中存在final修饰的方法，那么该方法也不可被代理；

因为CGLib原理是动态生成被代理类的子类；

final类不能被继承，final方法不能被复写；

使用CGLIB，需要两个jar包：

cglib-3.3.0.jar

asm-7.1.jar

cglib-nodep-3.3.0.jar：使用nodep包不需要关联asm的jar包,jar包内部已经包含了asm的类.

cglib-3.3.0.jar：使用此jar包需要关联asm的jar包,否则运行时报错；

## CGLib动态代理原理是什么？

CGLIB原理： 动态生成一个要代理类的子类，子类重写要代理的类的所有不是final的方法。在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑；

CGLIB底层：使用字节码处理框架ASM，来转换字节码并生成新的类。但不推荐直接使用ASM编码开发，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉，编码比较复杂繁琐；

CGLIB缺点：对于final类和final方法，无法进行代理；

## JDK动态代理与CGLib代理的区别？

原理区别：

JDK动态代理是利用反射机制生成一个实现代理接口的类（这个类看不见摸不着，在jvm内存中有这个类），在调用具体方法前调用InvokeHandler来处理。核心是实现InvocationHandler接口，使用invoke()方法进行面向切面的处理，调用相应的拦截和处理；

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理，核心是实现MethodInterceptor接口，使用intercept()方法进行面向切面的处理，调用相应的拦截和处理；

各自局限：

1、JDK的动态代理机制只能代理实现了接口的类，而没有实现接口的类就不能实现JDK的动态代理。

2、CGLib的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类进行代理。

JDK Proxy 的优势：

最小化依赖关系，减少依赖意味着简化开发和维护，JDK 本身的支持，可能比 cglib 更加可靠；

可以平滑进行 JDK 版本升级，而字节码类库通常需要进行更新以保证在新版 Java 上能够使用。

CGLib 的优势：

从某种角度看，必须要求调用者实现接口是有些侵入性的实践，CGLib 动态代理就没有这种限制，只操作我们关心的类，而不必为其他相关类增加工作量，一个普通的类也可以实现代理；

## Javassist动态代理

自1999年以来的Java字节码工程工具包，Javassis使Java字节码操作变得简单，它是用于在Java中编辑字节码的类库，由东京工业大学的数学和计算机科学系的 Shigeru Chiba （千叶 滋）所创建的，它加入了开放源代码JBoss 应用服务器项目,通过使用Javassist对字节码操作为JBoss实现动态"AOP"框架；

官网：<http://www.javassist.org/>

Github：<https://github.com/jboss-javassist/javassist>