一、什么是 MyBatis？  
 直接看官方文档：https://mybatis.org/mybatis-3/zh/index.html。  
  
 从上面我们了解到：  
  
 1、MyBatis 是一款优秀的持久层框架  
  
 2、MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。  
  
 3、MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。  
  
 二、原理解析  
 1、程序员和Mybatis 和数据的关系：人通过mybatis框架来操作数据库。  
  
 2、思考问题并解决  
  
 问题1：首先我们必须告诉MyBatis要怎么操作数据库？  
  
 我们把可以通过XML配置文件或者注解的方式，MyBatis提供了一个类Configuration， Mybatis 读取XML配置文件后会将内容放在一个Configuration类中，Configuration类会存在整个Mybatis生命周期，以便重复读取。  
  
 问题2：想要Mybatis与数据库打交道，就要有一个类似于JDBC的Connection对象，在MyBatis中叫SqlSesion，所以我们要有一个SqlSession。  
  
 Mybatis 读取XML配置文件后会将内容放在一个Configuration类中，SqlSessionFactoryBuilder会读取Configuration类中信息创建SqlSessionFactory。SqlSessionFactory创建SqlSession。  
  
 String resource = "mybatis-config.xml";  
 InputStream inputStream = Resources.getResourceAsStream(resource);  
 SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  
  
 SqlSession sqlSession=null;  
 try{  
 sqlSession=sqlSessionFactory.openSession();  
 //some code  
 sqlSession.commit();  
 } catch(Exception ex){  
 sqlSession.roolback();  
 } finally{  
 if(sqlSession!=null){  
 sqlSession.close();  
 }  
 }  
  
 关于SqlSessionFactory的创建，Mybatis采用构造模式来完成创建。  
  
 第一步：XMLConfigBuilder解析XML配置，读出配置参数，存入Configuration类中。  
  
 第二步：Configuration类创建SqlSessionFactory。（DefaultSqlSessionFactory的构造函数传入Configuration类）  
  
 深入了解：SqlSessionFactoryBuilder.builder（inputStream）  
  
 //该方法.builder中的主要内容：  
  
 XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);  
 SqlSessionFactory localSqlSessionFactory = build(parser.parse());  
  
//build(parser.parse())方法实则为：  
  
public SqlSessionFactory build(Configuration config) {  
 return new DefaultSqlSessionFactory(config);  
 }  
 问题三：SqlSession能干什么？  
  
 SqlSession用途主要有两种  
  
 ①. 获取对应的Mapper，让映射器通过命名空间和方法名称找到对应的SQL，发送给数据库执行后返回结果。  
  
 RoleMapper roleMapper = sqlSession.getMapper(RoleMapper.class);  
 Role role = roleMapper.getRole(1L);  
 ②. 直接使用SqlSession，通过命名信息去执行SQL返回结果，该方式是IBatis版本留下的，SqlSession通过Update、Select、Insert、Delete等方法操作。  
  
 Role role = (Role)sqlSession.select("com.mybatis.mapper.RoleMapper.getRole",1L);  
 Mybatis底层利用JDK动态代理技术实现该接口，底层最后还是使用的IBatis中SqlSession通过Update、Select、Insert、Delete等方法操作。  
  
 问题四：上面说到Mybatis底层利用JDK动态代理技术实现该接口，但是我们在使用MyBatis的时候，都是只写接口不用写实现类，为什么呢？  
  
 为什么要使用动态代理？可以在不修改别代理对象代码的基础上，通过扩展代理类，进行一些功能的附加与增强。  
  
 我们先看看传统的JDK动态代理：  
  
 1、首先有一个接口  
  
public interface Calculate {  
 void add(int i, int j);  
}  
2、然后是接口的实现类  
  
public class CalculateImp implements Calculate {  
 @Override  
 public void add(int i, int j) {  
 System.out.println("result = " + (i + j));  
 }  
}  
3、代理类实现InvocationHandler  
  
public class CalculateProxy implements InvocationHandler {  
 private Object target;  
 //总要让我知道要代理谁吧：构造方法中把传入一个代理类的实例  
 public CalculateProxy(Object target) {  
 this.target = target;  
 }  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 System.out.println("====== Before() ======");  
 method.invoke(target, args);  
 System.out.println("====== After () ======");  
 return null;  
 }  
}  
4、拿到代理对象，操作接口方法  
  
public class test {  
 public static void main(String[] args) {  
 InvocationHandler handler = new CalculateProxy(new CalculateImp());  
 Calculate calculateProxy =  
 (Calculate) Proxy.newProxyInstance(Calculate.class.getClassLoader(),  
 new Class[]{Calculate.class},  
 handler);  
 calculateProxy.add(10,20);  
 }  
}  
Proxy.newProxyInstance()方法有三个参数：  
  
 1. 类加载器(Class Loader)  
  
 2. 需要实现的接口数组  
  
 3. InvocationHandler接口。所有动态代理类的方法调用，都会交由InvocationHandler接口实现类里的invoke()方法去处理。这是动态代理的关键所在。  
  
 回到我们之前的问题，我们并没有接口实现类，那没有实现类还为什么还能调用方法操作。其实是这样的：  
  
 操作数据库主要是通过SQL语句，那么只要找到SQL语句然后执行不就可以！  
  
 借鉴：https://www.cnblogs.com/demingblog/p/9544774.html  
  
 通过例子分析：  
  
 BlogMapper mapper = session.getMapper(BlogMapper.class);  
 Blog blog = mapper.selectBlog(1);  
 这里 mapper 可以调用selectBlog(1) 这个方法，说明 mapper 是个对象，因为对象才具有方法行为实现啊。BlogMapper接口是不能实例化的，更没有具体方法实现。我们并没有定义一个类，让它实现BlogMapper接口，而在这里它只是通过调用session.getMapper() 所得到的。由此，我们可以推断：肯定是session.getMapper() 方法内部产生了BlogMapper的实现类。有什么技术可以根据BlogMapper 接口生成了一个实现类呢？想到这里，对于有动态代理 。  
  
 Mapper 接口的注册  
 我们既然能够从SqlSession中得到BlogMapper接口的，那么我们肯定需要先在哪里把它放进去了，然后 SqlSession 才能生成我们想要的代理类啊。我们可以从getMapper()联系，可能会有一个setMapper()或者addMapper()方法。确实是有！  
  
 configuration.addMapper(BlogMapper.class);  
 跟着这个 addMapper 方法的代码实现是这样的：  
  
public <T> void addMapper(Class<T> type) {  
 mapperRegistry.addMapper(type);  
 }  
 我们看到这里 mapper 实际上被添加到 mapperRegistry （mapper注册器）中。继续跟进代码：

public class MapperRegistry {  
 private final Map<Class<?>, MapperProxyFactory<?>> knownMappers  
 = new HashMap<Class<?>, MapperProxyFactory<?>>();  
  
 public <T> void addMapper(Class<T> type) {  
 if (type.isInterface()) { // 只添加接口  
 if (hasMapper(type)) { // 不允许重复添加  
 throw new BindingException("Type " + type + " is already known to the MapperRegistry.");  
 }  
 boolean loadCompleted = false;  
 try {  
 knownMappers.put(type, new MapperProxyFactory<T>(type)); // 注意这里  
  
 MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config, type);  
 parser.parse();  
 loadCompleted = true;  
  
 } finally {  
 if (!loadCompleted) {  
 knownMappers.remove(type);  
 }  
 }  
 }  
 }  
}

我们首先看到MapperRegistry类，有一个私有属性knowMappers，它是一个HashMap 。其Key 为当前Class对象，value 为一个MapperProxyFactory实例  
  
 在MapperRegistry类的addMapper()方法中，knownMappers.put(type, new MapperProxyFactory<T>(type));相当于把：诸如BlogMapper 之类的Mapper接口被添加到了MapperRegistry 中的一个HashMap中。并以 Mapper 接口的 Class 对象作为 Key , 以一个携带Mapper接口作为属性的MapperProxyFactory 实例作为value 。MapperProxyFactory从名字来看，好像是一个工厂，用来创建Mapper Proxy的工厂。  
  
 上面我们已经知道，Mapper 接口被到注册到了MapperRegistry中——放在其名为knowMappers 的HashMap属性中，我们在调用Mapper接口的方法的时候，是这样的：  
  
 BlogMapper mapper = session.getMapper(BlogMapper.class);  
 这里，我们跟踪一下session.getMapper() 方法的代码实现，这里 SqlSession 是一个接口，他有两个实现类，  
  
 一个是DefaultSqlSession,另外一个是SqlSessionManager,  
  
 这里我们用的是DefaultSqlSession. 为什么是DefaultSqlSession呢？因为我们在初始化SqlSessionFactory的时候所调用的SqlSessionFactoryBuilder的build()方法里边配置的就是DefaultSqlSession, 所以，我们进入到DefaultSession类中，看看它对session.getMapper(BlogMapper.class)是怎么实现的：  
  
public class DefaultSqlSession implements SqlSession {  
 private Configuration configuration;  
  
 @Override  
 public <T> T getMapper(Class<T> type) {  
 return configuration.<T>getMapper(type, this); //最后会去调用MapperRegistry.getMapper  
 }  
}  
如代码所示，这里的 getMapper 调用了 configuration.getMapper , 这一步操作其实最终是调用了MapperRegistry,而此前我们已经知道，MapperRegistry是存放了一个HashMap的，我们继续跟踪进去看看，那么这里的get，肯定是从这个hashMap中取数据。我们来看看代码：  
  
public class MapperRegistry {  
  
 private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap<Class<?>, MapperProxyFactory<?>>();// Mapper 映射  
  
 public <T> T getMapper(Class<T> type, SqlSession sqlSession) {  
 final MapperProxyFactory<T> mapperProxyFactory =  
 (MapperProxyFactory<T>) knownMappers.get(type);  
  
 try {  
 return mapperProxyFactory.newInstance(sqlSession); // 重点看这里  
 } catch (Exception e) {  
 }  
 }  
}  
我们调用的session.getMapper(BlogMapper.class);最终会到达上面这个方法，这个方法，根据BlogMapper的class对象,以它为key在knowMappers 中找到了对应的value —— MapperProxyFactory(BlogMapper) 对象，然后调用这个对象的newInstance()方法。根据这个名字，我们就能猜到这个方法是创建了一个对象，代码是这样的：  
  
public class MapperProxyFactory<T> { //映射器代理工厂  
  
 private final Class<T> mapperInterface;  
 private Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method, MapperMethod>();  
  
 public MapperProxyFactory(Class<T> mapperInterface) {  
 this.mapperInterface = mapperInterface;  
 }  
 // 删除部分代码，便于阅读  
  
 @SuppressWarnings("unchecked")  
 protected T newInstance(MapperProxy<T> mapperProxy) {  
 //使用了JDK自带的动态代理生成映射器代理类的对象  
 return (T) Proxy.newProxyInstance(  
 mapperInterface.getClassLoader(),  
 new Class[] { mapperInterface },  
 mapperProxy);  
 }  
  
 public T newInstance(SqlSession sqlSession) {  
 final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);  
 return newInstance(mapperProxy);  
 }  
  
}  
  
看到这里，就清楚了，最终是通过Proxy.newProxyInstance产生了一个BlogMapper的代理对象。Mybatis 为了完成 Mapper 接口的实现，运用了代理模式。具体是使用了JDK动态代理，这个Proxy.newProxyInstance方法生成代理类的三个要素是：  
  
 ClassLoader —— 指定当前接口的加载器即可  
 当前被代理的接口是什么 —— 这里就是 BlogMapper  
 代理类是什么 —— 这里就是 MapperProxy  
 代理模式中，代理类(MapperProxy)中才真正的完成了方法调用的逻辑。我们贴出MapperProxy的代码，如下：  
  
public class MapperProxy<T> implements InvocationHandler, Serializable {// 实现了InvocationHandler  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 //代理以后，所有Mapper的方法调用时，都会调用这个invoke方法  
  
 if (Object.class.equals(method.getDeclaringClass())) {  
 try {  
 return method.invoke(this, args); // 注意1  
 } catch (Throwable t) {  
 throw ExceptionUtil.unwrapThrowable(t);  
 }  
 }  
  
 final MapperMethod mapperMethod = cachedMapperMethod(method); // 使用了缓存  
 //执行CURD  
 return mapperMethod.execute(sqlSession, args); // 注意2  
 }  
  
}  
  
 我们调用的 Blog blog = mapper.selectBlog(1); 实际上最后是会调用这个MapperProxy的invoke方法。这段代码中，if 语句先判断，我们想要调用的方法是否来自Object类，这里的意思就是，如果我们调用toString()方法，那么是不需要做代理增强的，直接还调用原来的method.invoke()就行了。只有调用selectBlog()之类的方法的时候，才执行增强的调用——即mapperMethod.execute(sqlSession, args);这一句代码逻辑。  
  
 而mapperMethod.execute(sqlSession, args);这句最终就会执行增删改查了，代码如下：  
  
public Object execute(SqlSession sqlSession, Object[] args) {  
 Object result;  
 if (SqlCommandType.INSERT == command.getType()) { //insert 处理，调用SqlSession的insert  
 Object param = method.convertArgsToSqlCommandParam(args);  
 result = rowCountResult(sqlSession.insert(command.getName(), param));  
 } else if (SqlCommandType.UPDATE == command.getType()) { // update  
 Object param = method.convertArgsToSqlCommandParam(args);  
 result = rowCountResult(sqlSession.update(command.getName(), param));  
 } else if (SqlCommandType.DELETE == command.getType()) { // delete  
 Object param = method.convertArgsToSqlCommandParam(args);  
 result = rowCountResult(sqlSession.delete(command.getName(), param));  
 } else if (SqlCommandType.SELECT == command.getType()) {  
 // 删除部分代码  
 } else {  
 throw new BindingException("Unknown execution method for: " + command.getName());  
 }  
 // 删除部分代码  
 return result;  
 }  
  
 再往下一层，就是执行JDBC那一套了，获取链接，执行，得到ResultSet，解析ResultSet映射成JavaBean。  
  
 总结一下各个过程：  
 1、Mybatis 读取XML配置文件后会将内容放在一个Configuration类中，SqlSessionFactoryBuilder会读取Configuration类中信息创建SqlSessionFactory。  
  
 2、在初始化SqlSessionFactory时，Mapper 接口进行注册，注册在了名为 MapperRegistry 类的 HashMap中，  
  
 key = Mapper class， value = 创建当前Mapper的工厂。  
  
 3、SqlSessionFactory创建SqlSession。  
  
 4、SqlSession中可以通过getMapper()拿到代理对象，SqlSession.getMapper 运用了 JDK动态代理，产生了目标Mapper接口的代理对象。  
  
 5. 动态代理的 代理类是 MapperProxy ，这里边mapperMethod.execute(sqlSession, args)最终完成了增删改查方法的调用。  
 ————————————————  
 版权声明：本文为CSDN博主「Lin\_Dong\_Tian」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
 原文链接：https://blog.csdn.net/qq\_40645822/article/details/101844675