Mybatis可以把Mapper.xml文件直接映射到对应的接口，调用接口方法会自动去Mapper.xml文件中找到对应的标签，这个功能就是利用Java的动态代理在binding包中实现的。

一、注册Mapper

在初始化时会把获取到的Mapper接口注册到MapperRegistry，注册的时候创建一个Mapper代理工厂，这个工厂通过JDK的代理创建一个执行对象，创建代理需要的InvocationHandler为MapperProxy

```
//接口注册
public class MapperRegistry {
    public <T> void addMapper(Class<T> type) {
            //如果是接口
            if (type.isInterface()) {
                    if (hasMapper(type)) {
                            throw new BindingException("Type " + type + " is already known
                    }
                    boolean loadCompleted = false;
                    try {
                            //放到map中, value为创建代理的工厂
                            knownMappers.put(type, new MapperProxyFactory<T>(type));
                            // It's important that the type is added before the parser is
                            // otherwise the binding may automatically be attempted by the
                            //这里是解析Mapper接口里面的注解
                            MapperAnnotationBuilder parser = new MapperAnnotationBuilder(
                            parser.parse();
                            loadCompleted = true;
                    } finally {
                            if (!loadCompleted) {
                                    knownMappers.remove(type);
                            }
                    }
            }
    }
}
```

二、获取接口对象

从knownMappers中根据接口类型取出对应的代理创建工厂，用该工厂创建代理。

```
public class MapperRegistry {
    public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
                //取出MapperProxyFactory
```

858883986@qq.com

```
            final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>) known
            if (mapperProxyFactory == null)
                    throw new BindingException("Type " + type + " is not known to the Mapp
            try {
                        //创建代理
                    return mapperProxyFactory.newInstance(sqlSession);
            } catch (Exception e) {
                    throw new BindingException("Error getting mapper instance. Cause: " +
            }
    }
}

//创建代理的工厂
public class MapperProxyFactory<T> {
    /**
     * 需要创建代理的接口
     */
    private final Class<T> mapperInterface;
    /**
     * 执行方法的缓存,不需要每次都创建MapperMethod
     */
    private Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method, MapperM

    public MapperProxyFactory(Class<T> mapperInterface) {
            this.mapperInterface = mapperInterface;
    }

    public Class<T> getMapperInterface() {
            return mapperInterface;
    }

    public Map<Method, MapperMethod> getMethodCache() {
            return methodCache;
    }
    @SuppressWarnings("unchecked")
    protected T newInstance(MapperProxy<T> mapperProxy) {
            //创建代理, InvocationHanderl是MapperProxy
            return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class
                            mapperProxy);
    }
```

858883986@qq.com

```
    /**
     * 传入sqlSession创建代理
     * @param sqlSession
     * @return
     */
    public T newInstance(SqlSession sqlSession) {
            //把代理执行需要用到的对象传入
            final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInter
            return newInstance(mapperProxy);
    }
}
```

三、调用接口方法

调用代理方法会进入到MapperProxy的public Object invoke(Object proxy, Method method, Object[] args)方法

```
public class MapperProxy<T> implements InvocationHandler, Serializable {

    private static final long serialVersionUID = -6424540398559729838L;
    private final SqlSession sqlSession;
    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethod> methodCache;

    public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, Mappe
            this.sqlSession = sqlSession;
            this.mapperInterface = mapperInterface;
            this.methodCache = methodCache;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            //如果方法是Object里面的则直接调用方法
            if (Object.class.equals(method.getDeclaringClass())) {
                    try {
                            return method.invoke(this, args);
                    } catch (Throwable t) {
                            throw ExceptionUtil.unwrapThrowable(t);
                    }
            }

            //获取执行方法的封装对象
```

858883986@qq.com

```
            final MapperMethod mapperMethod = cachedMapperMethod(method);
            //里面就是找到对应的sql 执行sql语句
            return mapperMethod.execute(sqlSession, args);
    }
    //缓存，不需要每次都创建
    private MapperMethod cachedMapperMethod(Method method) {
            MapperMethod mapperMethod = methodCache.get(method);
            if (mapperMethod == null) {
                    //传入配置参数
                    mapperMethod = new MapperMethod(mapperInterface, method, sqlSession.g
                    methodCache.put(method, mapperMethod);
            }
            return mapperMethod;
    }
}
```

最终执行sql会进入到MapperMethod中execute方法：

```
//具体的根据接口找到配置文件标签的类
public class MapperMethod {

    private final SqlCommand command;
    private final MethodSignature method;

    public MapperMethod(Class<?> mapperInterface, Method method, Configuration config) {
            //SqlCommand封装该接口方法需要执行sql的相关属性，如：id(name)，类型
            this.command = new SqlCommand(config, mapperInterface, method);
            //执行方法特性进行封装，用于构造sql参数，判断执行sql逻辑走哪条分支
            this.method = new MethodSignature(config, method);
    }

    public Object execute(SqlSession sqlSession, Object[] args) {
            Object result;
            //先找到对应的执行sql类型，sqlSession会调用不同方法
            if (SqlCommandType.INSERT == command.getType()) {
                    Object param = method.convertArgsToSqlCommandParam(args);
                    result = rowCountResult(sqlSession.insert(command.getName(), param));
            } else if (SqlCommandType.UPDATE == command.getType()) {

                    Object param = method.convertArgsToSqlCommandParam(args);
```

858883986@qq.com

```
22                  result = rowCountResult(sqlSession.update(command.getName(), param));
23              } else if (SqlCommandType.DELETE == command.getType()) {
24                  Object param = method.convertArgsToSqlCommandParam(args);
25                  result = rowCountResult(sqlSession.delete(command.getName(), param));
26              } else if (SqlCommandType.SELECT == command.getType()) {//如果是查询，需要对返回
27                  //根据方法的特性判断进入哪个执行分支
28                  if (method.returnsVoid() && method.hasResultHandler()) {
29                          executeWithResultHandler(sqlSession, args);
30                          result = null;
31                  } else if (method.returnsMany()) {
32                          result = executeForMany(sqlSession, args);
33                  } else if (method.returnsMap()) {
34                          result = executeForMap(sqlSession, args);
35                  } else {
36                          //只查一条数据
37                          Object param = method.convertArgsToSqlCommandParam(args);
38                          result = sqlSession.selectOne(command.getName(), param);
39                  }
40              } else {
41                  throw new BindingException("Unknown execution method for: " + command
42              }
43              if (result == null && method.getReturnType().isPrimitive() && !method.returns
44                  throw new BindingException("Mapper method '" + command.getName()
45                                  + " attempted to return null from a method with a pri
46                                  + ").");
47              }
48              return result;
49      }
50  }
```

上面就是根据接口、方法、配置参数找到对应的执行sql，并构造参数，解析执行结果，具体sql执行在sqlSession流程里面，后面再看。

858883986@qq.com