说明文档

> 由测试类出发，一步步解释说明

1. 见下图

```java
/**
 * 解析配置文件，创建工厂对象，生成对应实例，为IUserDao接口生成代理实现类
 */
InputStream resourceAsSteam = Resources.getResourceAsSteam( path: "sqlMapConfig.xml");
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsSteam);
SqlSession sqlSession = sqlSessionFactory.openSession();
IUserDao userDao = sqlSession.getMapper(IUserDao.class);
```

2. 要新建更新，删除操作，所以在 `IUserDao` 文件中创建俩个方法

```java
public Integer updateOne(User user) throws Exception;

public Integer deleteById(User user) throws Exception;
```

3. 调用方法，由于第一步为此类生成了代理类，所以此类调用任何发放都会触发代理类实现。这一步要求我们在 `getMapper` 中判断调用哪个方法，之前的课程中通过返回是否为泛型判断调用 `selectList` OR `selectOne` 这里新增 `updateOne` `deleteById` ，这四个方法明显的区别是 `select` `update` `delete` 三个不同的操作

```xml
<mapper namespace="com.lagou.dao.IUserDao">

    <!--sql的唯一标识: namespace.id来组成 ： statementId-->
    <select id="findAll" resultType="com.lagou.pojo.User">
        select * from user
    </select>


    <!--
        User user = new User()
        user.setId(1);
        user.setUsername("zhangsan")
    -->
    <select id="findByCondition" resultType="com.lagou.pojo.User" paramterType="com.lagou.pojo.User">
        select * from user where id = #{id} and username = #{username}
    </select>

    <update id="updateOne" paramterType="com.lagou.pojo.User" resultType="java.lang.Integer">
        update user set username = #{username} where id = #{id}
    </update>

    <delete id="deleteById" paramterType="com.lagou.pojo.User" resultType="java.lang.Integer">
        delete from user where id = #{id}
    </delete>
</mapper>
```

可以通过解析 `UserMapper` 的时候将标签名称放入 `configuration.getMappedStatementMap()`

```java
public class MappedStatement {

    //id标识
    private String id;
    //返回值类型
    private String resultType;
    //参数值类型
    private String paramaterType;
    //sql语句
    private String sql;

    //SQL操作类型 select/update/delete
    private String operationType;
```

```
    for (Element element : list) {
        String id = element.attributeValue( s: "id");
        String resultType = element.attributeValue( s: "resultType");
        String parameterType = element.attributeValue( s: "paramterType");
        String sqlText = element.getTextTrim();
        MappedStatement mappedStatement = new MappedStatement();
        mappedStatement.setId(id);
        mappedStatement.setResultType(resultType);
        mappedStatement.setParamterType(parameterType);
        mappedStatement.setSql(sqlText);
        mappedStatement.setOperationType(element.getName());
        String key = namespace+"."+id;
        configuration.getMappedStatementMap().put(key,mappedStatement);
    }
```

在代理方法中，利用这个标签来判断

```
@Override
public <T> T getMapper(Class<?> mapperClass) {
    // 使用JDK动态代理来为Dao接口生成代理对象，并返回

    Object proxyInstance = Proxy.newProxyInstance(DefaultSqlSession.class.getClassLoader(), new Class[]{mapperClass}, new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            // 底层都还是去执行JDBC代码 //根据不同情况，来调用selctList或者selectOne
            // 准备参数 1: statmentid :sql语句的唯一标识: namespace.id= 接口全限定名.方法名
            // 方法名: findAll
            String methodName = method.getName();
            String className = method.getDeclaringClass().getName();

            String statementId = className+"."+methodName;

            MappedStatement mappedStatement = configuration.getMappedStatementMap().get(statementId);

            switch (mappedStatement.getOperationType()){
                case "select":
                    Type genericReturnType = method.getGenericReturnType();
                    if(genericReturnType instanceof ParameterizedType){
                        return selectList(statementId, args);
                    }
                    return selectOne(statementId,args);
                case "update":
                case "delete":
                    return updateOne(statementId, args);
                default:
                    throw new RuntimeException("代理未找到查询配置");
            }
        }
    });

    return (T) proxyInstance;
}
```

`update` `delete` 和 `select` 在 `PreparedStatement` 类中的区别为调用的方法不同 `executeUpdate` 和 `executeQuery` 所以可以把公共方法提取出来

```
@Override
public Integer updateOne(Configuration configuration, MappedStatement mappedStatement, Object... params) throws Exception {
    PreparedStatement preparedStatement = basicOperate(configuration, mappedStatement, params);
    return preparedStatement.executeUpdate();
}
```

```java
 */
private PreparedStatement basicOperate(Configuration configuration, MappedStatement mappedStatement, Object... params) throws SQLException, ClassNotFoundExce
    // 1. 注册驱动，获取连接
    Connection connection = configuration.getDataSource().getConnection();

    // 2. 获取sql语句 : select * from user where id = #{id} and username = #{username}
    //转换sql语句:  select * from user where id = ? and username = ? , 转换的过程中，还需要对#{}里面的值进行解析存储
    String sql = mappedStatement.getSql();
    BoundSql boundSql = getBoundSql(sql);

    // 3.获取预处理对象: preparedStatement
    PreparedStatement preparedStatement = connection.prepareStatement(boundSql.getSqlText());

    // 4. 设置参数
    //获取到了参数的全路径
    String paramterType = mappedStatement.getParamterType();
    Class<?> paramtertypeClass = getClassType(paramterType);

    List<ParameterMapping> parameterMappingList = boundSql.getParameterMappingList();
    for (int i = 0; i < parameterMappingList.size(); i++) {
        ParameterMapping parameterMapping = parameterMappingList.get(i);
        String content = parameterMapping.getContent();

        //反射
        Field declaredField = paramtertypeClass.getDeclaredField(content);
        //暴力访问
        declaredField.setAccessible(true);
        Object o = declaredField.get(params[0]);

        preparedStatement.setObject( i: i + 1, o);
    }
    return preparedStatement;
}
```