1.Mybatis的Dao层实现

1.1 传统开发方式

1.1.1编写UserDao接口

```
public interface UserDao {
    List<User> findAll() throws IOException;
}
```

1.1.2.编写UserDaoImpl实现

```
public class UserDaoImpl implements UserDao {
    public List<User> findAll() throws IOException {
        InputStream resourceAsStream =
                    Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory = new
                    SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = sqlSessionFactory.openSession();
        List<User> userList = sqlSession.selectList("userMapper.findAll");
        sqlSession.close();
        return userList;
    }
}
```

1.1.3 测试传统方式

```
@Test
public void testTraditionDao() throws IOException {
    UserDao userDao = new UserDaoImpl();
    List<User> all = userDao.findAll();
    System.out.println(all);
}
```

1.2 代理开发方式

1.2.1 代理开发方式介绍

采用 Mybatis 的代理开发方式实现 DAO 层的开发，这种方式是我们后面进入企业的主流。

Mapper 接口开发方法只需要程序员编写Mapper 接口（相当于Dao 接口），由Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边Dao接口实现类方法。

Mapper 接口开发需要遵循以下规范：

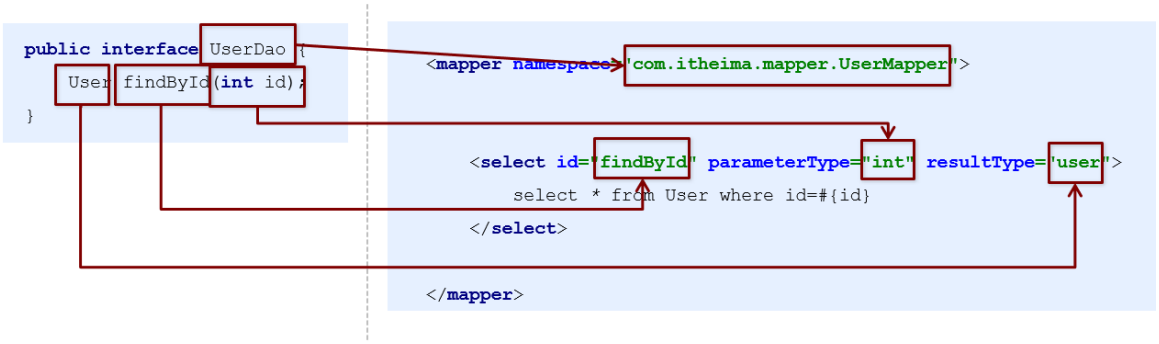
1) Mapper.xml文件中的namespace与mapper接口的全限定名相同

2) Mapper接口方法名和Mapper.xml中定义的每个statement的id相同

3) Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql的parameterType的类型相同

4) Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同

1.2.2 编写UserMapper接口



1.2.3测试代理方式

```
@Test
public void testProxyDao() throws IOException {
    InputStream resourceAsStream =
Resources.getResourceAsStream("SqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    //获得MyBatis框架生成的UserMapper接口的实现类
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    User user = userMapper.findById(1);
    System.out.println(user);
    sqlSession.close();
}
```

1.3 知识小结

MyBatis的Dao层实现的两种方式：

手动对Dao进行实现：传统开发方式

代理方式对Dao进行实现：

```
 **UserMapper userMapper = sqlSession.getMapper(UserMapper.class);**
```

2.MyBatis映射文件深入

2.1 动态sql语句

2.1.1动态sql语句概述

Mybatis 的映射文件中，前面我们的 SQL 都是比较简单的，有些时候业务逻辑复杂时，我们的 SQL是动态变化的，此时在前面的学习中我们的 SQL 就不能满足要求了。

参考的官方文档，描述如下：

Dynamic SQL

One of the most powerful features of MyBatis has always been its Dynamic SQL capabilities. If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

While working with Dynamic SQL will never be a party, MyBatis certainly improves the situation with a powerful Dynamic SQL language that can be used within any mapped SQL statement.

The Dynamic SQL elements should be familiar to anyone who has used JSTL or any similar XML based text processors. In previous versions of MyBatis, there were a lot of elements to know and understand. MyBatis 3 greatly improves upon this, and now there are less than half of those elements to work with. MyBatis employs powerful OGNL based expressions to eliminate most of the other elements:

- if
- choose (when, otherwise)
- trim (where, set)
- foreach

2.1.2动态 SQL 之<if>

我们根据实体类的不同取值，使用不同的 SQL语句来进行查询。比如在 id如果不为空时可以根据id查询，如果username 不同空时还要加入用户名作为条件。这种情况在我们的多条件组合查询中经常会碰到。

```
<select id="findByCondition" parameterType="user" resultType="user">
    select * from User
    <where>
        <if test="id!=0">
            and id=#{id}
        </if>
        <if test="username!=null">
            and username=#{username}
        </if>
    </where>
</select>
```

当查询条件id和username都存在时，控制台打印的sql语句如下：

```
    … … …
    //获得MyBatis框架生成的UserMapper接口的实现类
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    User condition = new User();
    condition.setId(1);
    condition.setUsername("lucy");
    User user = userMapper.findByCondition(condition);
    … … …
```

```
 - Created connection 586084331.
- Setting autocommit to false on JDBC Connection [com.mysql.jdbc.
- ==>  Preparing: select * from User WHERE id=? and username=?
- ==> Parameters: 1(Integer), lucy(String)
- <==      Total: 1
```

当查询条件只有id存在时，控制台打印的sql语句如下：

```
… … …
//获得MyBatis框架生成的UserMapper接口的实现类
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
User condition = new User();
condition.setId(1);
User user = userMapper.findByCondition(condition);
… … …
```

```
- Setting autocommit to false on JDBC Connection [com.mysql.
- ==>  Preparing: select * from User WHERE id=?
- ==> Parameters: 1(Integer)
- <==      Total: 1
```

2.1.3 动态 SQL 之<foreach>

循环执行sql的拼接操作，例如：SELECT * FROM USER WHERE id IN (1,2,5)。

```
<select id="findByIds" parameterType="list" resultType="user">
    select * from User
    <where>
        <foreach collection="array" open="id in(" close=")" item="id"
separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

测试代码片段如下：

```
… … …
//获得MyBatis框架生成的UserMapper接口的实现类
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
int[] ids = new int[]{2,5};
List<User> userList = userMapper.findByIds(ids);
System.out.println(userList);
… … …
```

```
11:21:02,237 DEBUG findByIds:159 - ==>  Preparing: select * from User WHERE id in( ? , ? )
11:21:02,262 DEBUG findByIds:159 - ==> Parameters: 2(Integer), 5(Integer)
11:21:02,280 DEBUG findByIds:159 - <==      Total: 2
[User{id=2, username='tom', password='123'}, User{id=5, username='haohao', password='123'}]
```

foreach标签的属性含义如下：

标签用于遍历集合，它的属性：

•collection：代表要遍历的集合元素，注意编写时不要写#{}

•open：代表语句的开始部分

•close：代表结束部分

•item：代表遍历集合的每个元素，生成的变量名

•sperator：代表分隔符

2.2 SQL片段抽取

Sql 中可将重复的 sql 提取出来，使用时用 include 引用即可，最终达到 sql 重用的目的

```
<!--抽取sql片段简化编写-->
<sql id="selectUser" select * from User</sql>
<select id="findById" parameterType="int" resultType="user">
    <include refid="selectUser"></include> where id=#{id}
</select>
<select id="findByIds" parameterType="list" resultType="user">
    <include refid="selectUser"></include>
    <where>
        <foreach collection="array" open="id in(" close=")" item="id"
separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

2.3 知识小结

MyBatis映射文件配置：

：插入

：修改

：删除

：where条件

：if判断

：循环

：sql片段抽取

3. MyBatis核心配置文件深入

3.1typeHandlers标签

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时， 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器（截取部分）。

类型处理器	Java 类型	JDBC 类型
`BooleanTypeHandler`	`java.lang.Boolean`, `boolean`	数据库兼容的 `BOOLEAN`
`ByteTypeHandler`	`java.lang.Byte`, `byte`	数据库兼容的 `NUMERIC` 或 `BYTE`
`ShortTypeHandler`	`java.lang.Short`, `short`	数据库兼容的 `NUMERIC` 或 `SHORT INTEGER`
`IntegerTypeHandler`	`java.lang.Integer`, `int`	数据库兼容的 `NUMERIC` 或 `INTEGER`
`LongTypeHandler`	`java.lang.Long`, `long`	数据库兼容的 `NUMERIC` 或 `LONG INTEGER`

你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。具体做法为：实现 org.apache.ibatis.type.TypeHandler 接口， 或继承一个很便利的类 org.apache.ibatis.type.BaseTypeHandler， 然后可以选择性地将它映射到一个JDBC类型。例如需求：一个Java中的Date数据类型，我想将之存到数据库的时候存成一个1970年至今的毫秒数，取出来时转换成java的Date，即java的Date与数据库的varchar毫秒值之间转换。

开发步骤：

①定义转换类继承类BaseTypeHandler

②覆盖4个未实现的方法，其中setNonNullParameter为java程序设置数据到数据库的回调方法，getNullableResult为查询时 mysql的字符串类型转换成 java的Type类型的方法

③在MyBatis核心配置文件中进行注册

测试转换是否正确

```
public class MyDateTypeHandler extends BaseTypeHandler<Date> {
    public void setNonNullParameter(PreparedStatement preparedStatement, int i,
Date date, JdbcType type) {
        preparedStatement.setString(i,date.getTime()+"");
    }
    public Date getNullableResult(ResultSet resultSet, String s) throws
SQLException {
        return new Date(resultSet.getLong(s));
    }
    public Date getNullableResult(ResultSet resultSet, int i) throws
SQLException {
        return new Date(resultSet.getLong(i));
    }
    public Date getNullableResult(CallableStatement callableStatement, int i)
throws SQLException {
        return callableStatement.getDate(i);
    }
}
```

```
<!--注册类型自定义转换器-->
<typeHandlers>
    <typeHandler handler="com.itheima.typeHandlers.MyDateTypeHandler">
</typeHandler>
</typeHandlers>
```

测试添加操作：

```
user.setBirthday(new Date());
userMapper.add2(user);
```

数据库数据：

id	username	password	birthday
1	lucy	123	1539751863457
2	tom	123	1539751863457
5	haohao	123	1539751863457

测试查询操作：

```
13:53:10,222 DEBUG findAll:159 - <==      Total: 9
[User{id=1, username='lucy', password='123', birthday=Wed Oct 17 12:51:03 GMT+08:00 2018}
13:53:10,222 DEBUG JdbcTransaction:123 - Resetting autocommit to true on JDBC Connection
13:53:10,222 DEBUG JdbcTransaction:91 - Closing JDBC Connection [com.mysql.jdbc.JDBC4Conn
13:53:10,222 DEBUG PooledDataSource:363 - Returned connection 249155636 to pool.
```

3.2 plugins标签

MyBatis可以使用第三方的插件来对功能进行扩展，分页助手PageHelper是将分页的复杂操作进行封装，使用简单的方式即可获得分页的相关数据

开发步骤：

①导入通用PageHelper的坐标

②在mybatis核心配置文件中配置PageHelper插件

③测试分页数据获取

①导入通用PageHelper坐标

```
<!-- 分页助手 -->
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>3.7.5</version>
</dependency>
<dependency>
    <groupId>com.github.jsqlparser</groupId>
    <artifactId>jsqlparser</artifactId>
    <version>0.9.1</version>
</dependency>
```

②在mybatis核心配置文件中配置PageHelper插件

```
<!-- 注意：分页助手的插件  配置在通用馆mapper之前 -->
<plugin interceptor="com.github.pagehelper.PageHelper">
    <!-- 指定方言 -->
    <property name="dialect" value="mysql"/>
</plugin>
```

③测试分页代码实现

```
@Test
public void testPageHelper(){
    //设置分页参数
    PageHelper.startPage(1,2);

    List<User> select = userMapper2.select(null);
    for(User user : select){
        System.out.println(user);
    }
}
```

获得分页相关的其他参数

```
//其他分页的数据
PageInfo<User> pageInfo = new PageInfo<User>(select);
System.out.println("总条数："+pageInfo.getTotal());
System.out.println("总页数："+pageInfo.getPages());
System.out.println("当前页："+pageInfo.getPageNum());
System.out.println("每页显示长度："+pageInfo.getPageSize());
System.out.println("是否第一页："+pageInfo.isIsFirstPage());
System.out.println("是否最后一页："+pageInfo.isIsLastPage());
```

3.3 知识小结

MyBatis核心配置文件常用标签：

1、properties标签：该标签可以加载外部的properties文件

2、typeAliases标签：设置类型别名

3、environments标签：数据源环境配置标签

4、typeHandlers标签：配置自定义类型处理器

5、plugins标签：配置MyBatis的插件

```
"
```

#####