

1. MyBatis 框架

1.1. MyBatis 是什么

MyBatis 是一个持久层框架,是 apache 下的顶级项目。

Mybatis 托管到 goolecode 下,后来又托管到 GitHub。

MyBatis 让程序员主要精力都集中到 sql 上,通过 mybatis 提供的映射方式,自由灵活生成(半自动化,大部分需要程序员编写 sql 语句)满足需要的 sql 语句。

MyBatis 可以将查询结果集灵活映射成 java 对象。

MyBatis 是支持普通 SQL 查询,存储过程和高级映射的优秀持久层框架。MyBatis 消除了几乎所有的 JDBC 代码和参数的手工设置以及对结果集的检索。**MyBatis** 可以使用简单的 XML 或注解用于配置和原始映射,将接口和 Java 的 POJO (Plain Old Java Objects, 普通的 Java 对象)映射成数据库中的记录。

1) MyBatis 目前提供了三种语言实现的版本,包括: Java、.NET 以及 Ruby。(我主要学习 java,就讲 java 的使用)

2) 它提供的持久层框架包括 SQL Maps 和 Data Access Objects (DAO)。

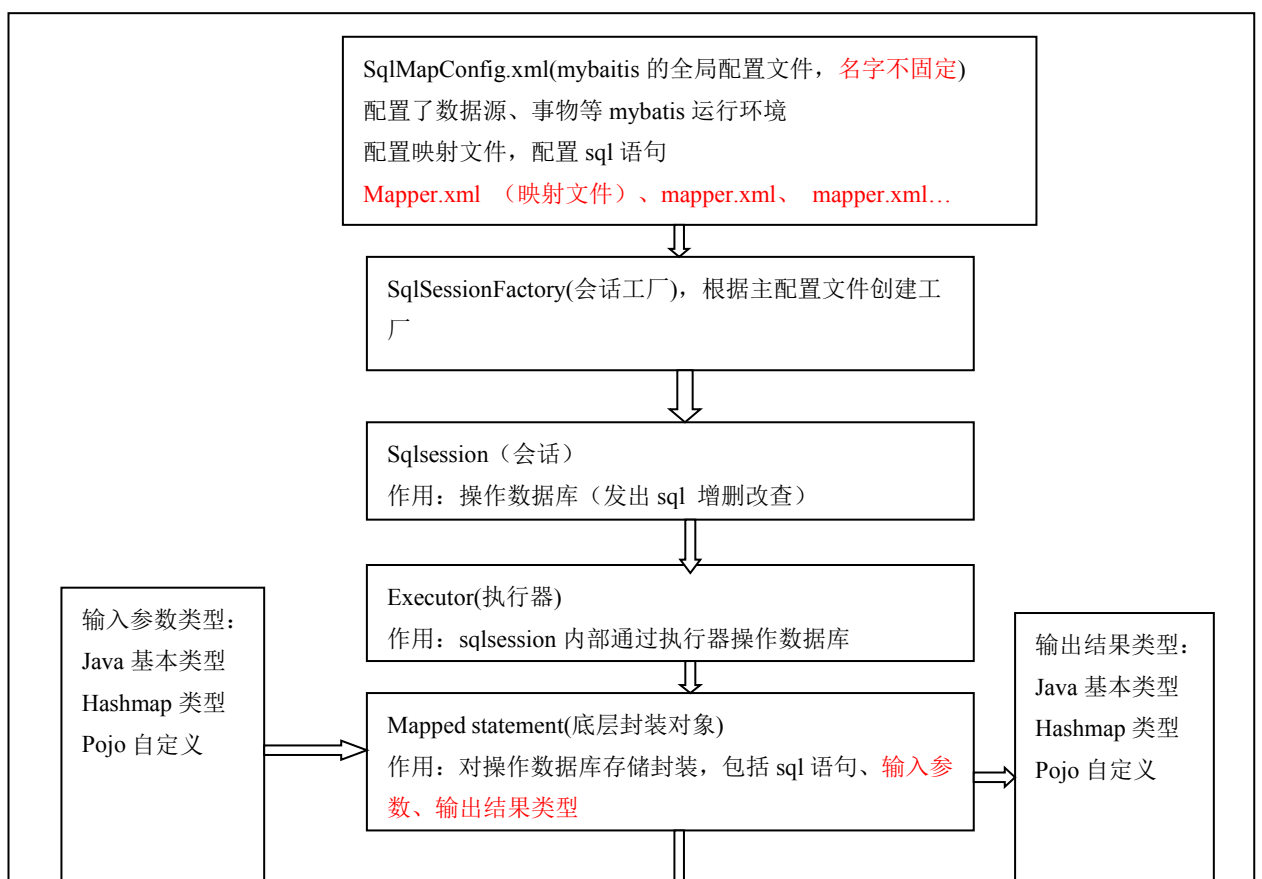
3) **mybatis** 与 **hibernate** 的对比?

mybatis 提供一种“半自动化”的 ORM 实现。

这里的“半自动化”,是相对 Hibernate 等提供了全面的数据库封装机制的“全自动化”ORM 实现而言,“全自动”ORM 实现了 POJO 和数据库表之间的映射,以及 SQL 的自动生成和执行。

而 mybatis 的着力点,则在于 POJO 与 SQL 之间的映射关系。

1.2. MyBatis 框架原理



1. 3. 框架搭建

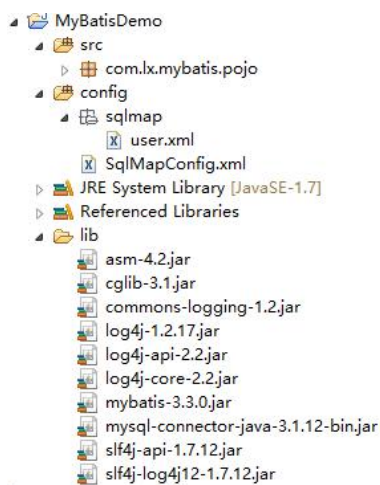
1. 3. 1. Mybatis 版本

mybatis-3.3.

1. 3. 2. 所需 jar



1. 3. 3. 工程结构



1. 3. 4. 核心配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
```

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org/DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 引入外部配置文件 -->
    <properties resource="db.properties"></properties>
    <!-- 配置mybatis运行环境 -->
    <environments default="cybatis">
        <environment id="cybatis">
            <!-- type="JDBC" 代表使用JDBC的提交和回滚来管理事务 -->
            <transactionManager type="JDBC" />
            <!-- mybatis提供3种数据源类型，分别是：POOLED, UNPOOLED, JNDI -->
            <!-- POOLED 表示支持JDBC数据源连接池 -->
            <!-- UNPOOLED 表示不支持数据源连接池 -->
            <!-- JNDI 表示支持外部数据源连接池 -->
            <dataSource type="POOLED">
                <property name="driver" value="${driverClass}" />
                <property name="url" value="${jdbcUrl}" />
                <property name="username" value="${user}" />
                <property name="password" value="${password}" />
            </dataSource>
        </environment>
    </environments>

    <!-- 加载sql映射文件 -->
    <mappers>
        <mapper resource="mapper/studentMapper.xml"></mapper>
    </mappers>
</configuration>

```

1.3.5. db.properties

在 src/main/resources 下新建：db.properties

```

jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/student
jdbc.username=root
jdbc.password=lanxin

```

在 sqlMapConfig.xml 主配置文件中引入 db.properties 文件

```

<!-- 加载属性文件 -->
<properties resource="db.properties">
    <!--properties 中还可以配置一些属性名和属性值 -->
    <!-- <property name="jdbc.driver" value="" /> -->
</properties>

```

在 datasource 中引入 db.properties 中的值

```

<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"></transactionManager>
        <!-- datasource -->
        <dataSource type="POOLED">
            <property name="driver" value="${jdbc.driver}" />
            <property name="url" value="${jdbc.url}" />
            <property name="username" value="${jdbc.username}" />
            <property name="password" value="${jdbc.password}" />
        </dataSource>
    </environment>
</environments>

```

</environments>

1.3.6. 映射文件

```
<?xml version="1.0" encoding="utf-8" ?>

<!DOCTYPE mapper

    PUBLIC "-//mybatis.org/DTD Mapper 3.0//EN"

    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org/DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<!--
    namespace命名空间:就是对sql进行分类管理。
-->
<mapper namespace="user">
    <!--
        在映射文件中配置很多sql语句
        id: 标识映射文件的sql
        最终是将sql语句封装到mappedstatement对象中, 所以将id称为statement的id,
        parameterType:指定输入参数的类型
        #{ }:表示一个占位符
        #{id}:其中的id表示接收输入的参数, 参数名称就是id,如果输入的是简单类型,
        #{ }中的参数名可以任意 可以value或其他名称
        resultType:指定sql查询的结果单条映射成java对象
    -->
    <select id="selectUserById" parameterType="int" resultType="java.Lang.String">
        select name from user where id = #{value}
    </select>
```

1.4. 基本CRUD

1.4.1. 根据id查询一个用户

1.4.1.1. 映射

```
<!--
    在映射文件中配置很多sql语句
    id: 标识映射文件的sql
    最终是将sql语句封装到mappedstatement对象中, 所以将id称为statement的id,
    parameterType:指定输入参数的类型
    #{ }:表示一个占位符
    #{id}:其中的id表示接收输入的参数, 参数名称就是id,如果输入的是简单类型,
    #{ }中的参数名可以任意 可以value或其他名称
    resultType:指定sql查询的结果单条映射成java对象
-->
<select id="selectUserById" parameterType="int" resultType="java.Lang.String">
    select name from user where id = #{value}
</select>
```

1.4.1.2. 测试代码

```
SqlSession session=null;
try {
    InputStream stream=Resources.getResourceAsStream("SqlMapConfig.xml");
    //创建会话工厂，传入mybatis核心配置文件的信息。（文件流）
    SqlSessionFactory sqlSessionFactory=new SqlSessionFactoryBuilder().build(stream);

    //通过工厂得到sqlsession
    session=sqlSessionFactory.openSession();

    //通过sqlsession操作数据库
    //第一个参数：映射文件中sql的id=namespace+"."+sql id
    User user=session.selectOne("user.selectUserById", 1);
    System.out.println(user);

} catch (Exception e) {
    e.printStackTrace();
} finally{
    //释放资源
    session.close();
}
```

1.4.2. 根据姓名模糊查询

1.4.2.1. 映射

```
<select id="queryUserByName" parameterType="java.lang.String" resultType="com.Lx.my
    select * from user where name like #{value}
</select>
```

1.4.2.2. 代码

1.4.3. 查询用户根据多个条件

1.4.3.1. 映射

```
<select id="queryUserByParamers" parameterType="com.Lx.mybatis.pojo.User"
    resultType="com.Lx.mybatis.pojo.User">
    select * from user where name=#{name} and sex=#{sex}
</select>
```

1.4.3.2. 代码

1.4.4. 查询所有用户

1.4.4.1. 映射

```
<select id="selectAllUser" resultType="com.Lx.mybatis.pojo.User">
    select * from user
</select>
```

1.4.4.2. 代码

1.4.5. 插入用户

1.4.5.1. 映射

```
<insert id="saveUser" parameterType="com.Lx.mybatis.pojo.User">
    insert into user(name,sex) VALUES(#{name},#{sex})
```

1.4.5.2. 测试代码

```
@org.junit.Test
public void saveUser(){
    InputStream stream;
    SqlSession session=null;
    try {
        stream = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory=new SqlSessionFactoryBuilder().build(s

        session=sqlSessionFactory.openSession();

        User user=new User();
        user.setName("三胖1");
        user.setSex("男");
        session.insert("user.saveUser",user);
        System.out.println(user);
        session.commit();

    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        session.close();
    }
}
```

注意默认拿到的 **sqlsession** 是需要手动提交事务的，如果要实现自动提交事务需要

```
SqlSession session=sessionFactory.openSession(b: true);
```

1. 4. 6. 返回主键 ID

1. 4. 6. 1. 插入用户并返回主键值（自增）

1. 4. 6. 1. 1. 映射

```
<!--
    添加用户
    parameterType:指定输入参数类型是pojo（包括用户信息）
    #{}:指定pojo中的属性名，接收到pojo对象的属性值，mybatis使用OGNL获取属性的值
-->
<insert id="saveUser" parameterType="com.lx.mybatis.pojo.User">
    <!--
        =====返回自增主键的值=====
        将插入到数据库的主键值返回，返回到user对象中
        select LAST_INSERT_ID(): 得到insert进去的主键值，只适用于自增主键
        keyProperty: 将查询到的主键值设置到parameterType指定的对象的哪个属性
        order:select LAST_INSERT_ID() 的执行顺序，相对于insert语句来说它的执行顺序
        resultType:指定LAST_INSERT_ID() 的结果类型
    -->
    <selectKey keyProperty="id" resultType="java.lang.Integer" order="AFTER">
        select LAST_INSERT_ID()
    </selectKey>
    insert into user(name,sex) VALUES(#{name},#{sex})|
```

1. 4. 6. 1. 2. 代码

```
@org.junit.Test
public void saveUser(){
    InputStream stream;
    SqlSession session=null;
    try {
        stream = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory=new SqlSessionFactoryBuilder().build(
            stream);
        session=sqlSessionFactory.openSession();

        User user=new User();
        user.setName("三胖1");
        user.setSex("男");
        session.insert("user.saveUser",user);
        System.out.println(user);|
        session.commit();

    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        session.close();
    }
}
```

1. 4. 6. 2. 插入用户返回主键（UUID）

1. 4. 6. 2. 1. 映射

```

<!--
    使用mysql的uuid生成主键
    过程：首先通过uuid()得到主键，将主键设置到user对象的id属性中
    其次在执行insert时，从user对象中取出id属性值。
-->
<insert id="saveUser" parameterType="com.Lx.mybatis.pojo.User">
    <selectKey keyProperty="id" resultType="java.lang.String" order="BEFORE">
        select uuid()
    </selectKey>
    insert into user(id,name,sex) VALUES(#{id},#{name},#{sex})
</insert>

```

1.4.6.3. 插入用户返回主键（oracle 序列）

1.4.6.3.1. 映射

```

<insert id="saveUser" parameterType="com.Lx.mybatis.pojo.User">
    <selectKey keyProperty="id" resultType="java.lang.Integer" order="BEFORE">
        select seq_user.nextval
    </selectKey>
    insert into user(id,name,sex) VALUES(#{id},#{name},#{sex})
</insert>

```

1.4.7. 修改用户

1.4.7.1. 映射

```

<!--
    根据id修改用户信息
    parameterType指定user对象，包括id和更新信息，注意id必须存在
-->
<update id="updateUser" parameterType="com.Lx.mybatis.pojo.User">
    update user set name=#{name},sex=#{sex} where id=#{id}
</update>

```


1.4.7.2. 代码

```
public void updateUser() {
    InputStream stream;
    SqlSession session=null;
    try {
        stream = Resources.getResourceAsStream("SqlMap
        SqlSessionFactory sqlSessionFactory=new SqlSes

        session=sqlSessionFactory.openSession();

        User user=new User();
        user.setId(5);
        user.setName("金三胖");
        user.setSex("女");
        session.update("user.updateUser",user);
        session.commit();

    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        session.close();
    }
}
```

1.4.8. 删除用户

1.4.8.1. 映射

```
<!-- 删除用户 -->
<delete id="deleteUser" parameterType="java.lang.Integer">
    delete from user where id=#{id}
</delete>
```

1.5. Mybatis 添加日志信息

1.5.1. 使用标准日志输出

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/>
    </settings>
```

```
</configuration>
```

这里的 value 值可以是 SLF4J、Apache Commons Logging、Log4J2、Log4J、JDK logging，并会按顺序查找，如果都查不到则会关闭日志。

1.5.2. 使用 log4j

1.5.2.1. 配置开关

```
<settings>
```

```
<setting name="logImpl" value="LOG4J"/>
```

```
</settings>
```

1.5.2.2. 配置 jar 包

```
<dependency>
```

```
<groupId>log4j</groupId>
```

```
<artifactId>log4j</artifactId>
```

```
<version>1.2.17</version>
```

```
</dependency>
```

1.5.2.3. 配置 log4j.properties

在根目录下新建 log4j.properties

```
# 可设置级别: TRACE→DEBUG→INFO→WARNING→ERROR→FATAL→OFF
```

```
# 高级别 level 会屏蔽低级别 level。
```

```
# debug: 显示 debug、info、error
```

```
# info: 显示 info、error
```

```
log4j.rootLogger=DEBUG,console,file
```

```
#Log4j.rootLogger=INFO,console
```

```
#-----
```

```
#输出到控制台
```

```
log4j.appender.console=org.apache.log4j.ConsoleAppender
```

```
#设置输出样式
```

```
log4j.appender.console.layout=org.apache.log4j.PatternLayout
```

```
#日志输出信息格式为
```

```
log4j.appender.console.layout.ConversionPattern=[%-d{yyyy-MM-dd
```

```
HH:mm:ss}]-[%t-%5p]-[%C-%M(%L)]: %m%n
```

```

#-----
#根据日志文件大小自动产生新日志文件
log4j.appender.file=org.apache.log4j.RollingFileAppender
#日志文件输出目录
log4j.appender.file.File=c:/log/log1.log
#样式为PatternLayout
log4j.appender.file.layout=org.apache.log4j.PatternLayout
#定义文件最大大小
log4j.appender.file.MaxFileSize=3kb
#日志文件内容前面加时间，日志输出信息格式为
log4j.appender.file.layout.ConversionPattern=[%d{HH:mm:ss:SSS}][%C-%M] -%m%n
#保存几个备份文件
log4j.appender.file.MaxBackupIndex=5

```

1. 6. Mybatis-config.xml 详解

1. 6. 1. typeAliases（别名）

1. 6. 1. 1. mybatis 默认支持的别名

| 别名 | 映射的类型 |
|----------|---------|
| _byte | byte |
| _long | long |
| _short | short |
| _int | int |
| _integer | int |
| _double | double |
| _float | float |
| _boolean | boolean |
| string | String |

| | |
|------------|------------|
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |

1.6.1.2. 单个定义别名

```
<!-- 定义别名 -->
<typeAliases>
  <typeAlias type="com.lx.mybatis.pojo.User" alias="user"/>
</typeAliases>

<select id="selectUserById" parameterType="int" resultType="user">
  select * from user where id = #{value}
</select>
```

在映射文件中直接使用别名

1.6.1.3. 批量定义别名（常用）

```
<!-- 定义别名 -->
<typeAliases>
  <!-- <typeAlias type="com.lx.mybatis.pojo.User" alias="user"/> -->
  <!-- 批量定义别名
    指定包名，mybatis自动扫描包中的po类，自动定义别名，别名就是类名（首字母大写或小写都可以）
  -->
  <package name="com.lx.mybatis.pojo"/>
</typeAliases>
```

1. 6. 2. typeHandlers

在 mybatis 通过类型处理器来完成 jdbc 类型对 Java 类型的转换

| 类型处理器 | Java 类型 | JDBC 类型 |
|----------------------------|---|--------------------------------|
| BooleanTypeHandler | <code>java.lang.Boolean, boolean</code> | 数据库兼容的 BOOLEAN |
| ByteTypeHandler | <code>java.lang.Byte, byte</code> | 数据库兼容的 NUMERIC 或 BYTE |
| ShortTypeHandler | <code>java.lang.Short, short</code> | 数据库兼容的 NUMERIC 或 SHORT INTEGER |
| IntegerTypeHandler | <code>java.lang.Integer, int</code> | 数据库兼容的 NUMERIC 或 INTEGER |
| LongTypeHandler | <code>java.lang.Long, long</code> | 数据库兼容的 NUMERIC 或 LONG INTEGER |
| FloatTypeHandler | <code>java.lang.Float, float</code> | 数据库兼容的 NUMERIC 或 FLOAT |
| DoubleTypeHandler | <code>java.lang.Double, double</code> | 数据库兼容的 NUMERIC 或 DOUBLE |
| BigDecimalTypeHandler | <code>java.math.BigDecimal</code> | 数据库兼容的 NUMERIC 或 DECIMAL |
| StringTypeHandler | <code>java.lang.String</code> | CHAR, VARCHAR |
| ClobReaderTypeHandler | <code>java.io.Reader</code> | - |
| ClobTypeHandler | <code>java.lang.String</code> | CLOB, LONGVARCHAR |
| NStringTypeHandler | <code>java.lang.String</code> | NVARCHAR, NCHAR |
| NClobTypeHandler | <code>java.lang.String</code> | NCLOB |
| BlobInputStreamTypeHandler | <code>java.io.InputStream</code> | - |
| ByteArrayTypeHandler | <code>byte[]</code> | 数据库兼容的字节流类型 |
| BlobTypeHandler | <code>byte[]</code> | BLOB, LONGVARBINARY |
| DateTypeHandler | <code>java.util.Date</code> | TIMESTAMP |

| | | |
|-------------------------|--------------------|---|
| DateOnlyTypeHandler | java.util.Date | DATE |
| TimeOnlyTypeHandler | java.util.Date | TIME |
| SqlTimestampTypeHandler | java.sql.Timestamp | TIMESTAMP |
| SqlDateTypeHandler | java.sql.Date | DATE |
| SqlTimeTypeHandler | java.sql.Time | TIME |
| ObjectTypeHandler | Any | OTHER 或未指定类型 |
| EnumTypeHandler | Enumeration Type | VARCHAR-任何兼容的字符串类型，存储枚举的名称（而不是索引） |
| EnumOrdinalTypeHandler | Enumeration Type | 任何兼容的 NUMERIC 或 DOUBLE 类型，存储枚举的索引（而不是名称）。 |

1. 6. 3. Settings

常见全局 settings 配置

```
<settings>
  <!-- 使全局的映射器启用或禁用缓存。 -->
  <setting name="cacheEnabled" value="true"/>
  <!-- 全局启用或禁用延迟加载。当禁用时，所有关联对象都会即时加载。 -->
  <setting name="lazyLoadingEnabled" value="true"/>
  <!-- 当启用时，有延迟加载属性的对象在被调用时将会完全加载任意属性。否则，每种属性将会按需要加载。 -->
  <setting name="aggressiveLazyLoading" value="true"/>
  <!-- 是否允许单条 sql 返回多个数据集（取决于驱动的兼容性） default:true -->
  <setting name="multipleResultSetsEnabled" value="true"/>
  <!-- 是否可以使用列的别名（取决于驱动的兼容性） default:true -->
  <setting name="useColumnLabel" value="true"/>
  <!-- 允许 JDBC 生成主键。需要驱动器支持。如果设为了true，这个设置将强制使用被生成的主键，有一些驱动器不兼容不过仍然可以执行。 default:false -->
  <setting name="useGeneratedKeys" value="true"/>
  <!-- 指定 MyBatis 如何自动映射 数据基表的列 NONE：不隐射 PARTIAL：部分 FULL：全部 -->
  <setting name="autoMappingBehavior" value="PARTIAL"/>
  <!-- 这是默认的执行类型（SIMPLE：简单；REUSE：执行器可能重复使用prepared statements 语句；BATCH：执行器可以重复执行语句和批量更新） -->
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <!-- 使用驼峰命名法转换字段。 -->
  <setting name="mapUnderscoreToCamelCase" value="true"/>
  <!-- 设置本地缓存范围 session：就会有数据的共享 statement：语句范围（这样
```

就不会有数据的共享) default:session -->

```

<setting name="localCacheScope" value="SESSION"/>
<!-- 设置但 JDBC 类型为空时,某些驱动程序 要指定值,default:OTHER,插入空值时
不需要指定类型 -->
<setting name="jdbcTypeForNull" value="NULL"/>
</settings>

```

1.6.4. Mappers

1.6.4.1. 单个加载映射文件

```

<!-- 加载映射文件 -->
<mappers>
  <mapper resource="sqlmap/user.xml"/>
</mappers>

```

1.6.4.2. 批量加载映射文件（推荐）

```

<!-- 加载映射文件 -->
<mappers>
  <mapper resource="sqlmap/user.xml"/> -->
  <!--
    批量加载
    通过加载mapper接口来加载映射文件,前提是
    1、mapper接口的类名要和映射文件的文件名一样
    2、且在一个目录
    3、保证使用的是mapper代理
  -->
  <package name="com.lx.mybatis.dao"/>
</mappers>

```

1.7. Mybatis 动态代理实现 dao

Mapper 接口开发需要遵循以下规范:

- 1、 Mapper.xml 文件中的 namespace 与 mapper 接口的类路径相同。
- 2、 Mapper 接口方法名和 Mapper.xml 中定义每个 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 中定义每个 sql 的 resultType 的类型相同
- 5、 Mapper 接口的名字和 mapper.xml 的名字保持一致

总结:

selectOne 和 selectList

动态代理对象调用 sqlSession.selectOne()和 sqlSession.selectList()是根据 mapper 接口方法的返回值决定,如果返回 list 则调用 selectList 方法,如果返回单个对象则调用 selectOne 方法。

namespace

mybatis 官方推荐使用 mapper 代理方法开发 mapper 接口,程序员不用编写 mapper 接口实现类,使用 mapper 代理方法时,输入参数可以使用 pojo 包装对象或 map 对象,保证 dao 的通用性。

1.7.1. Mapper 映射

```
<!--
    1、namespace命名空间：就是对sql进行分类管理。
    2、namespace:值是mapper接口的全路径,实现mybatis动态代理
-->
<mapper namespace="com.lx.mybatis.mapper.UserMapper">
```

1.7.2. Mapper 接口

```
import com.lx.mybatis.pojo.User;
/**
 * mapper接口（就是我们以前写的dao）
 * 定义:
 * @author wangsq
 * @date 2017年9月28日上午9:30:45
 * version V1.0
 */
public interface UserMapper {
    public User selectUserById(int id);
    public List<User> selectUAllUser();
}
```

1.7.3. 调用

```
-----
@Test
public void queryUserByID(){
    InputStream stream;
    SqlSession session=null;
    try {
        stream = Resources.getResourceAsStream("SqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory=new SqlSessionFactoryBuilder().build(st

        session=sqlSessionFactory.openSession();

        UserMapper user=session.getMapper(UserMapper.class);
        List<User> list=user.selectUAllUser();
        System.out.println(list);

        session.commit();

    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        session.close();
    }
}
```


1.8. 输入输出类型

1.8.1. 输入映射

1.8.1.1. 基本类型

1.8.1.2. Pojo 包装类

1.8.1.3. Map 类型

1.8.2. 输出映射

1.8.2.1. resultType

使用 `resultType` 进行输出映射，只有查询出来的列名和 `pojo` 中的属性名一致，该列才可以映射成功。
如果查询出来的列名和 `pojo` 中的属性名全部不一致，没有创建 `pojo` 对象。
只要查询出来的列名和 `pojo` 中的属性有一个一致，就会创建 `pojo` 对象。

1.8.2.2. resultMap（高级查询）

1. `<constructor>` /*用来将查询结果作为参数注入到实例的构造方法中*/
2. `<idArg />` /*标记结果作为 ID*/
- 3.
4. `<arg />` /*标记结果作为普通参数*/
- 5.
6. `</constructor>`
7. `<id/>` /*一个 ID 结果，标记结果作为 ID*/
8. `<result/>` /*一个普通结果，JavaBean 的普通属性或字段*/
9. `<association>` /*关联其他的对象*/
10. `</association>`
11. `<collection>` /*关联其他的对象集合*/
12. `</collection>`
13. `<discriminator>` /*鉴别器，根据结果值进行判断，决定如何映射*/
14. `<case></case>` /*结果值的一种情况，将对应一种映射规则*/

15. `</discriminator>`

1.8.2.2.1. *id* 标签

```
<resultMap id="queryAllstudent_map" type="student">
  <id column="stuno" property="stuNo"></id> 一般映射主键字段
  <result column="stuname" property="stuName"></result>

  <association property="cla" column="classno" javaType="Classes"
select="query_classes"></association>
</resultMap>
```

1.8.2.2.2. *Result* 标签

普通的属性映射，跟 `resultType` 一样，但是不一样的地方是数据库的字段跟实体类里的字段可以不一样。

```
<resultMap id="query_student_resultmap_1" type="student">
  <result property="sex" column="sex"></result>
  <result property="stuname" column="stuName"></result>
</resultMap>
```

1.8.2.2.3. *Association*

对象的关联关系（比如查询员工级联查部门）（多对一）

① 嵌套查询

```
<!-- 嵌套查询结束-->

<!--===== 嵌套查询=====-->

<resultMap id="queryAllstudent_map" type="student">
  <id column="stuno" property="stuNo"></id>
  <result column="stuname" property="stuName"></result>

  <association property="cla" column="classno" javaType="Classes"
select="query_classes"></association>
</resultMap>

<resultMap id="query_classes_map" type="Classes">
  <id column="classno" property="cid"></id>
  <result column="classname" property="cname"></result>
</resultMap>

<select id="queryAllStuClasses" resultMap="queryAllstudent_map">
  SELECT * from student
</select>
```

```

<select id="query_classes" parameterType="int" resultMap="query_classes_map">
    SELECT * from classes where classno=#{VALUE}
</select>

```

② 嵌套结果

```

<!--嵌套结果开始 -->
<resultMap id="query_left_map" type="student">
    <id column="stuno" property="stuNo"></id>
    <result column="stuname" property="stuName"></result>
    <result column="classno" property="classno"></result>
    <result column="classname" property="cla.cname"></result>
</resultMap>

<select id="queryByClassno" resultMap="query_left_map">
    SELECT s.stuno,s.stuname,c.classname,s.classno from student s left JOIN classes c on
    s.classno=c.classno
</select>
<!--嵌套结果结束 -->

```

1.8.2.2.4 Collection

用于处理查询结果中关联其他对象集合的情况

嵌套查询

```

<!-- 嵌套查询开始 -->
    <resultMap type="classes" id="query_classes_map">
        <id column="classno" property="cid"/>
        <result column="classname" property="cname"/>

        <collection column="classno" property="list" javaType="list"
select="queryStudentByClassNo"></collection>
    </resultMap>

    <select id="queryClassesById" parameterType="int" resultMap="query_classes_map">
        select * from classes where classno=#{value}
    </select>

    <select id="queryStudentByClassNo" parameterType="int" resultType="student">
        select * from student where classno=#{value}
    </select>
<!-- 嵌套查询结束-->

```

嵌套结果

```

<!-- conllection 嵌套结果开始 -->
    <resultMap type="classes" id="query_classes_result_map">
        <id column="classno" property="cid"/>

```

```

        <result column="classname" property="cname"/>

        <collection property="list" javaType="list" ofType="student" column="classno">
            <id column="stuno" property="stuno"/>
            <result column="stuname" property="stuname"/>
        </collection>
    </resultMap>

    <select id="queryClassesByIdResult" parameterType="int"
resultMap="query_classes_result_map">
        select c.classno,c.classname,s.stuno,s.stuname from classes c
        left join student s on c.classno=s.classno
        where c.classno=#{value}
    </select>

<!-- collection 嵌套结果结束 -->

```

1. 8. 2. 2. 5. *Discriminator*

<discriminator/>元素很像 Java 语言中的 switch 语句，允许用户根据查询结果中指定字段的取值来执行不同的映射规则。

```

<mapper namespace="com. hy. mybatis. mapper. StudentMapper">

    <resultMap type="user" id="query_stu_sex_map">

        <discriminator column="sex" javaType="string">

            <case value="1" resultMap="query_stu_sex_1"></case>

            <case value="2" resultMap="query_stu_sex_2"></case>

        </discriminator>

    </resultMap>

    <resultMap type="user" id="query_stu_sex_1">

        <id column="stuno" property="stuno"/>

        <result column="stuname" property="stuname"/>

        <result column="age" property="age"/>

    </resultMap>

    <resultMap type="user" id="query_stu_sex_2">

        <id column="stuno" property="stuno"/>

        <result column="stuname" property="stuname"/>

        <result column="classno" property="classes.cid"/>

        <result column="classname" property="classes.cname"/>

    </resultMap>

    <select id="queryStuBySex" resultMap="query_stu_sex_map">

        select * from student s left join classes c on
        s.classno=c.classno
    </select>

```

```
</select>

</mapper>
```

1.9. #和\$的区别

mybatis 中的#与\$的区别

在 mybatis 中我们使用 SqlMap 进行 Sql 查询时需要引用参数，在参数引用中遇到的符号#和\$之间的区分为，#可以进行预编译，进行类型匹配，而\$不进行数据类型匹配，例如：

select * from table where id = #{id} ，其中如果字段 id 为字符型，那么 #{id} 表示的就是'id'类型，如果 id 为整型，那么#id#就是 id 类型。

select * from table where id = \${id} ，如果字段 id 为整型，Sql 语句就不会出错，但是如果字段 id 为字符型，那么 Sql 语句应该写成 select * from table where id = '\${id}'

总结：

1、#和\$都可以取值

2、#可以对类型进行匹配

3、\$不能对类型进行匹配，如果传过来的是字符串，则需要自己拼接引号

4、\$会引起 sql 注入

```
studentMapper.queryById("'zhngsan' or 1=1");
```

```
<select id="queryById" parameterType="java.lang.String" resultType="map">
    select * from student where stuname=${value}
</select>
```

1.10. 动态 sql

1.10.1. 什么是动态 sql 语句

Mybatis 核心对 sql 语句进行灵活操作，通过表达式进行判断，对 sql 进行灵活拼接、组装。

1.10.2. 应用场景

多条件的组合查询，比如查询用户有姓名、性别、籍贯这个条件组合，对查询条件进行判断，如果查询条件不为空才进行查询条件拼接。

1. 10. 3. 动态指令

1. 10. 3. 1. if

```
<select id="queryUserByName" parameterType="user" resultType="user">
  select * from user
  <!-- where会自动去掉第一个条件的and -->
  <where>
    <if test="name!=null and name!=''">
      and name like '%${name}%'
    </if>
    <if test="sex!=null and sex!=''">
      and sex=#{sex}
    </if>
  </where>
</select>
```

1. 10. 3. 2. choose、when、otherwise

```
<select id="selectInCondition" parameterType="student" resultType="student">
  <![CDATA[
    select * from student where studentId > #{studentId}
  ]]>
  <choose>
    <when test="studentName != null">
      and studentName = #{studentName};
    </when>
    <when test="studentAge != 0">
      and studentAge = #{studentAge};
    </when>
    <otherwise>
      or 1 = 1;
    </otherwise>
  </choose>
</select>
```

1. 10. 4. 格式化输出

1. 10. 4. 1. Where 标签

当 if 标签较多时，这样的组合可能会导致错误。 如下：

```
<select id="getStudentListWhere" parameterType="Object" resultMap="BaseResultMap">
  SELECT * from STUDENT WHERE
  <if test="name!=null and name!=''">
    NAME LIKE CONCAT(CONCAT('%', #{name}),'%')
  </if>
  <if test="hobby!= null and hobby!= ''">
```

```

        AND hobby = #{hobby}
    </if>
</select>

```

当 name 值为 null 时，查询语句会出现 “WHERE AND” 的情况，解决该情况除了将“WHERE”改为 “WHERE 1=1” 之外，还可以利用 where

标签。这个 “where” 标签会知道如果它包含的标签中有返回值的话，它就插入一个 ‘where’ 。此外，如果标签返回的内容是以 AND 或 OR 开头的，则它会剔除掉。

```

<select id="getStudentListWhere" parameterType="Object" resultMap="BaseResultMap">
    SELECT * from STUDENT
    <where>
        <if test="name!=null and name!=' ">
            AND NAME LIKE CONCAT(CONCAT('%', #{name}), '%')
        </if>
        <if test="hobby!= null and hobby!=' ">
            AND hobby = #{hobby}
        </if>
    </where>
</select>

```

1. 10. 4. 2. Set 标签

没有使用 if 标签时，如果有一个参数为 null，都会导致错误。当在 update 语句中使用 if 标签时，如果最后的 if 没有执行，则会导致逗号多余错误。使用 set 标签可以将动态的配置 set 关键字，和剔除追加到条件末尾的任何不相关的逗号。

```

<update id="updateStudent" parameterType="Object">
    UPDATE STUDENT
    SET NAME = #{name},
        MAJOR = #{major},
        HOBBY = #{hobby}
    WHERE ID = #{id};
</update>

```

```

<update id="updateStudent" parameterType="Object">
    UPDATE STUDENT SET
    <if test="name!=null and name!=' ">
        NAME = #{name},
    </if>
    <if test="hobby!=null and hobby!=' ">
        MAJOR = #{major},
    </if>
    <if test="hobby!=null and hobby!=' ">
        HOBBY = #{hobby}
    </if>

```

```
WHERE ID = #{id};
</update>
```

使用 set+if 标签修改后，如果某项为 null 则不进行更新，而是保持数据库原值。

```
<update id="updateStudent" parameterType="Object">
    UPDATE STUDENT
    <set>
        <if test="name!=null and name!=' ">
            NAME = #{name},
        </if>
        <if test="hobby!=null and hobby!=' ">
            MAJOR = #{major},
        </if>
        <if test="hobby!=null and hobby!=' ">
            HOBBY = #{hobby}
        </if>
    </set>
    WHERE ID = #{id};
</update>
```

1. 10. 4. 3. Trim 标签

| 属性 | 描述 |
|-----------------|---|
| prefix | 给sql语句拼接的前缀 |
| suffix | 给sql语句拼接的后缀 |
| prefixOverrides | 去除sql语句前面的关键字或者字符，该关键字或者字符由prefixOverrides属性指定，假设该属性指定为"AND"，当sql语句的开头为"AND"，trim标签将会去除该"AND" |
| suffixOverrides | 去除sql语句后面的关键字或者字符，该关键字或者字符由suffixOverrides属性指定 |

1. 10. 4. 3. 1. trim 的作用

可以动态在 sql 语句加前缀关键字和后缀关键字，同时还可以去掉前缀多余的 and 或者 or 关键字，还可以去掉后缀多余的逗号。


```

<update id="update" parameterType="student">
    UPDATE student
    <trim prefix="set" suffixOverrides=",">
        <if test="stuName!=null and stuName!=''">
            stuname=#{stuName},
        </if>
        <if test="sex!=null and sex!=''">
            sex=#{sex},
        </if>
    </trim>
    where stuno=#{stuNo}
</update>

```

1.10.5. Sql 片段

1.10.5.1.1. 需求

将上边实现的动态 sql 判断代码块抽取出来，组成一个 sql 片段（**相当于一个模板**）。其他的 statement 就可以引用。

```

<sql id="query_user_where">
    <if test="stuname!=null and stuname!=''">
        and stuname like "%#{stuname}%"
    </if>
    <if test="sex!=null and sex!=''">
        and sex=#{sex}
    </if>
</sql>

```

1.10.5.1.2. 定义 sql 片段

```

<!-- 定义sql片段
    id: sql片段的唯一标识
    在sql片段中不要包含where
-->
<sql id="query_user_where">
    <if test="name!=null and name!=''">
        and name like '%#{name}%'
    </if>
    <if test="sex!=null and sex!=''">
        and sex=#{sex}
    </if>
</sql>

```

1.10.5.1.3. 引用 sql 片段

```

<select id="queryUserByName" parameterType="user" resultType="user">
    select * from user
    <!-- where会自动去掉第一个条件的and -->
    <where>
        <!-- 引用sql片段的id,如果refid不在当前的mapper文件中,需要前边加namespace -->
        <include refid="query_user_where"></include>
        <!-- ...可以引用多个sql片段 -->
    </where>
</select>

```

1. 10. 6. Foreach

foreach 标签主要用于构建 in 条件,可在 sql 中对集合进行迭代。也常用到批量删除、添加等操作中。

属性介绍:

collection: collection 属性的值有三个分别是 list、array、map 三种,分别对应的参数类型为: List、数组、map 集合。

item : 表示在迭代过程中每一个元素的别名

index : 表示在迭代过程中每次迭代到的位置(下标)

open : 前缀

close : 后缀

separator : 分隔符,表示迭代时每个元素之间以什么分隔

1. 10. 6. 1. 1. 需求

Select * from user where id in(1,3,5)

1. 10. 6. 1. 2. Foreach 定义

Sql 片段:

```

<sql id="query_user_nameandsex_where">
    <if test="name!=null and name!=''">
        and name like '%${name}%'
    </if>
    <if test="sex!=null and sex!=''">
        and sex=#{sex}
    </if>
</sql>

<sql id="query_user_address_where">
    <if test="address!=null and address!=''">
        and address=#{address}
    </if>
</sql>

<sql id="query_user_ids_where">
    and id in
    <foreach collection="ids" item="item_id" open="(" separator="," close=")">
        #{item_id}
    </foreach>
</sql>

```

可以引入多个 sql 片段:

```

<select id="queryAllUser" resultType="user" parameterType="user">
    select * from user
    <where>
        <include refid="query_user_nameandsex_where"></include>
        <include refid="query_user_address_where"></include>
        <include refid="query_user_ids_where"></include>
    </where>
</select>

<sql id="query_user_where">
    <if test="name!=null and name!=''">
        and name like '%${name}%'
    </if>
    <if test="sex!=null and sex!=''">
        and sex=#{sex}
    </if>
    <if test="ids!=null">
        <!--
            collection:指定输入对象中集合属性
            item:每次遍历的对象
            open:开始遍历时拼接的串
            close:结束遍历时拼接的串
            separator: 表示在每次进行迭代之间以什么符号作为分隔符.
        -->
        and id in
        <foreach collection="ids" item="item_id" open="(" close=")" separator=",">
            #{item_id}
        </foreach>
    </if>
</sql>

```

1、如果传入 map

```

<select id="queryById" parameterType="map" resultType="student">

    select * from student

    <where>

        and stuno IN

        <foreach collection="maps.keys" item="item_id" open="(" separator="," close=")">

            #{maps[${item_id}]}

        </foreach>

    </where>
</select>

```

```

*/
public interface StudentMapper {
    public List<Student> queryAll();
    public List<Student> queryById(@Param("maps") Map map);
    public void insert_1(Student student);
    public void update_1(Map map);
    public void delete_1(String id);
    public List<Student> queryAllByResult();
    public List<Student> queryBysex();
}

```

2、传数组

```

<select id="queryById" parameterType="string" resultType="student">
    select * from student
    <where>
        and stuno IN
        <foreach collection="array" item="item_id" open="(" separator="," close=")">
            #{item_id}
        </foreach>
    </where>
</select>

```

如果传的是数组，则这是固定值array

```

public interface StudentMapper {
    public List<Student> queryAll();
    public List<Student> queryById(String[] ids);
    public void insert_1(Student student);
    public void update_1(Map map);
    public void delete_1(String id);
    public List<Student> queryAllByResult();
    public List<Student> queryBysex();
}

```

3、传集合

则 collection 的值就是 list,固定值。

4、传对象

要求对象里面有集合、数组、map。

2. Mybatis 分页

2.1. 数组分页

查询出全部数据，然后再 list 中截取需要的部分：

mybatis 接口

```
List<Student> queryStudentsByArray();
```

xml 配置文件

```

<select id="queryStudentsByArray" resultMap="studentmapper">
    select * from student
</select>

```

service

接口

```
List<Student> queryStudentsByArray(int currPage, int pageSize);
```

实现接口

@Override

```

public List<Student> queryStudentsByArray(int currPage, int pageSize) {
    //查询全部数据
    List<Student> students = studentMapper.queryStudentsByArray();

    //从第几条数据开始
    int firstIndex = (currPage - 1) * pageSize;
}

```

```
//到第几条数据结束  
int lastIndex = currPage * pageSize;  
  
return students.subList(firstIndex, lastIndex); //直接在 list 中截取  
}
```

2.2. SQL 分页


mybatis接口

```
List<Student> queryStudentsBySql (Map<String, Object> data);
```

xml文件

```
<select id="queryStudentsBySql" parameterType="map" resultMap="studentmapper">  
    select * from student limit #{currIndex} , #{pageSize}  
</select>
```

service

 接口

```
List<Student> queryStudentsBySql(int currPage, int pageSize);
```

实现类

```
public List<Student> queryStudentsBySql(int currPage, int pageSize) {  
    Map<String, Object> data = new HashMap();  
    data.put("currIndex", (currPage-1)*pageSize);  
    data.put("pageSize", pageSize);  
    return studentMapper.queryStudentsBySql(data);  
}
```

2.3. RowBounds 分页

原理分析：



图中可以看出, mybatis 中首先要在配置文件中配置一些东西, 然后根据这些配置去创建一个会话工厂, 再根据会话工厂创建会话, 会话发出操作数据库的 sql 语句, 然后通过执行器操作数据, 再使用 mappedStatement 对数据进行封装, 这就是整个 mybatis 框架的执行情况。那么 mybatis 的插件作用在哪一环节呢? 它主要作用在 Executor 执行器与 mappedStatement 之间, 也就是说 mybatis 可以在插件中获得要执行的 sql 语句, 在 sql 语句中添加 limit 语句, 然后再去对 sql 进行封装, 从而可以实现分页处理。

RowBounds:在 mapper.java 中的方法中传入 RowBounds 对象。

```
RowBounds rowBounds = new RowBounds(offset, page.getPageSize()); // offset  
起始行 // limit 是当前页显示多少条数据  
public List<ProdProduct> findRecords(HashMap<String, Object>  
map, RowBounds rowBounds);
```

2. 4. 插件分页

2. 4. 1. Jar 依赖

```
<dependency>  
    <groupId>com.github.pagehelper</groupId>  
    <artifactId>pagehelper</artifactId>  
    <version>5.1.2</version>  
</dependency>
```

2. 4. 2. Pagehelper 相关配置

```
<!-- 配置拦截器 -->  
<plugins>  
    <plugin  
interceptor="com.github.pagehelper.PageInterceptor"></plugin>  
</plugins>
```

注意: PageHelper 插件 4.0.0 以后的版本支持自动识别使用的数据库, 可以不用配置
<property name="dialect" value="mysql"/>

2. 4. 3. 使用

StudentMapper

```
studentMapper=session.getMapper(StudentMapper.class);
```

```
Page page = PageHelper.startPage(2, 3, true);
```

```
studentMapper.queryAll();
```


2. 4. 4. PageHelper 常见 API

Page page = PageHelper.startPage(pageNum, pageSize, true); - true 表示需要统计总数，这样会多进行一次请求 select count(0); 省略掉 true 参数只返回分页数据。

1)统计总数，（将 SQL 语句变为 *select count(0) from xxx*,只对简单 SQL 语句其效果，复杂 SQL 语句需要自己写）

```
Page<?> page = PageHelper.startPage(1,-1);  
  
long count = page.getTotal();
```

2)分页， pageNum - 第 N 页， pageSize - 每页 M 条数

A、只分页不统计(每次只执行分页语句)

```
PageHelper.startPage([pageNum],[pageSize]);  
  
List<?> pagelist = queryForList( xxx.class, "queryAll" , param);  
  
//pagelist 就是分页之后的结果
```

B、分页并统计（每次执行 2 条语句，一条 select count 语句，一条分页语句）适用于查询分页时数据发生变动，需要将实时的变动信息反映到分页结果上

```
Page<?> page = PageHelper.startPage([pageNum],[pageSize],[iscount]);  
  
List<?> pagelist = queryForList( xxx.class , "queryAll" , param);  
  
long count = page.getTotal();  
  
//也可以 List<?> pagelist = page.getList(); 获取分页后的结果集
```

3)使用 PageHelper 查全部（不分页）

```
PageHelper.startPage(1,0);  
  
List<?> alllist = queryForList( xxx.class , "queryAll" , param);
```

3. Mybatis 缓存

一级缓存：默认开启，是基于 SqlSession 范围的缓存

二级缓存：默认不支持二级缓存，是基于 Mapper 范围的缓存

4. Mybatis 批量操作

```
<!-- 批量插入 开始-->  
    <insert id="insertClassesBath" parameterType="list">  
        INSERT into classes(classno,classname)
```



```

        values
        <foreach collection="list" item="cla" separator=",">
            (#{cla.cid},#{cla.cname})
        </foreach>
    </insert>
<!-- 批量插入 结束 -->

```

5. Mybatis 注解

从 mybatis3.4.0 开始加入了 @Mapper 注解, 目的就是为了不再写 mapper 映射文件 (那个 xml 写的是真的蛋疼。。。)。很恶心的一个事实是源码中并没有对于这个注解的详细解释

@Mapper

```
public interface UserDao {
```

```

    @Select("select * from user where name = #{name}")
    public User find(String name);

```

```

    @Select("select * from user where name = #{name} and pwd = #{pwd}")
    /**

```

```

        * 对于多个参数来说, 每个参数之前都要加上 @Param 注解,
        * 要不然会找不到对应的参数进而报错
        */

```

```

    public User login(@Param("name")String name, @Param("pwd")String pwd);
}

```

5.1. 基本单表 CRUD 注解

```

@Select("select * from student where stuname = #{name}")
public Student find(String name);

```

```

@Select("select * from student")
public List<Student> findAll();

```

```

@update("update student set stuname=#{name},age=#{age1} where stuno=#{stuNo}")
public void update(Student student);

```

```

@Select("select * from student where stuno=#{id} and sex=#{sex}")
public Student queryById(@Param("id") String id, @Param("sex") String sex);

```

```

@Insert("insert into student values(#{name},#{age})")

```

```
public void save(Student student);

@Delete("delete from student where stuno=#{value}")
public void delete(String id);
(如：该方法中表中数据的返回类型就是根据 mapper 中的方法返回值来决定的)
```

5.2. 插入返回数据返回主键值

```
/*
 * statement="select LAST_INSERT_ID()":表示定义的查询语句
 * before=true: 表示在之前执行, boolean 类型的, 所以为true
 * resultType=int.class: 表示返回值得类型
 * keyProperty="empNo" : 表示将该查询的属性设置到某个属性中, 此处设置到empNo 中
 */
@SelectKey(statement="select LAST_INSERT_ID()", before=true, keyColumn="myNo", resultType=int.class, keyProperty="empNo")
@Insert(value="insert into emp(empNo,ename,sal) values(#{empNo},#{ename},#{sal})")
```

5.3. 动态语言注解

@InsertProvider,@UpdateProvider,@DeleteProvider 和 @SelectProvider,

5.3.1. Mapper

```
@SelectProvider(type=DemoSqlProvider.class,method="select5")
public List<Demo> select5(Demo demo);
```

5.3.2. 动态 sql 类

```
package com.kfit.demo.mapper;

import com.kfit.demo.bean.Demo;

public class DemoSqlProvider {

    /**
     * 查询语句.
     * @param demo
     * @return
     */
    public String select5(Demo demo) {
```

```

        StringBuffer sql = new StringBuffer("select *from demo where 1=1 ");
        if(demo.getName() != null){
            sql.append(" and name=#{name}");
        }
        if(demo.getEmail() != null){
            sql.append(" and email=#{email}");
        }
        return sql.toString();
    }
}

```

5.4. 复杂注解

5.4.1. resultMap 对应注解

```

@Results({@Result(column="stuname",property="name"),@Result(column="age",property="age1")})
@Select("select * from student where stuno=#{value}")
public List<Student> findAll1(String id);

```

5.4.2. @One(association 联合)

```

@Results({
    @Result(column="stuname",property="stuName"),
    @Result(column="age",property="age1"),
    @Result(column = "stuno",property = "stuNo"),
    @Result(column = "classno",property = "classes",one = @One(select = "com.hy.mybatisdemo.mapper.StudentTestMapper.getClassesById",fetchType = FetchType.EAGER))
})
@Select("select * from student where stuno=#{value}")
public List<Student> findAllStuCla(String id);

@Results({
    @Result(column = "classname",property = "cname")
})
@Select("select * from classes where classno=#{value}")
public Classes getClassById(Integer cid);

```

5. 4. 3. @Many 注解（collection 聚集）

```
@Results({
    @Result(column = "classname",property = "cname"),
    @Result(column = "classno",property = "list",many
= @Many(select="findAll"))
})
@Select("select * from classes where classno=#{value}")
public Classes getClassesById1(Integer cid);

@Select("select * from student where classno=#{value}")
public List<Student> findAll(Integer cid);
```

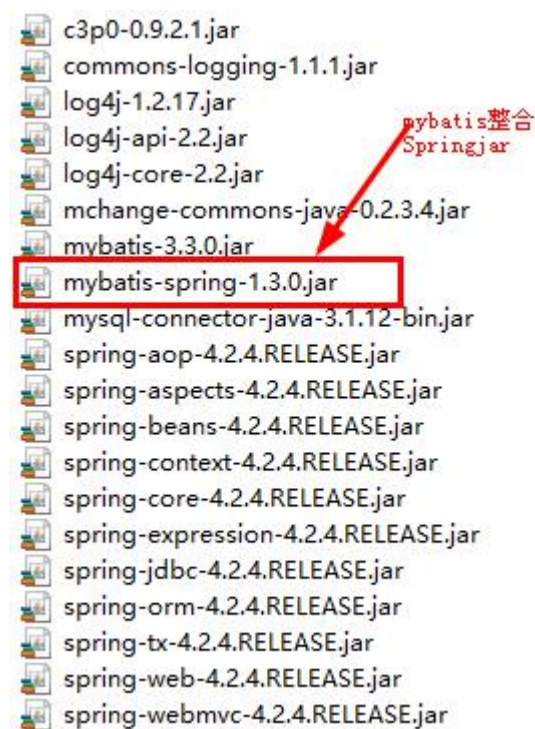
6. Spring 整合 mybatis

6. 1. 整合思路

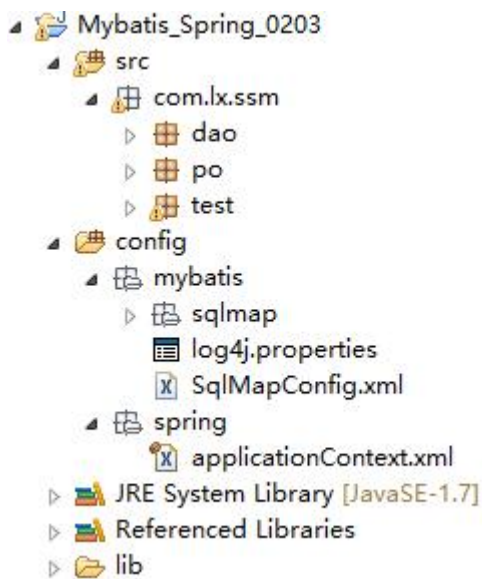
需要 spring 通过单例方式管理 SqlSessionFactory

Spring 和 mybatis 整合生成代理对象，使用 sqlSessionFactory 创建 sqlSession。（spring 和 mybatis 整合自动完成）。持久层的 mapper 都需要有 spring 管理。

6. 2. Jar 包



6.3. 工程结构



6.4. Spring application.xml 配置

```
<!-- 配置数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver"/>
    <property name="jdbcUrl" value="jdbc:mysql://127.0.0.1:3306/mybatis0203"/>
    <property name="user" value="root"/>
    <property name="password" value="123456"/>
</bean>

<!-- 创建sqlSessionFactory -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <!-- 加载mybatis核心配置文件 -->
    <property name="configLocation" value="classpath:mybatis/SqlMapConfig.xml"></pr
</bean>
```

6.5. 原始dao 方式整合

6.5.1. 配置

```
<!-- 原始dao -->
<bean id="userDao" class="com.lx.ssm.dao.ImpUserDao">
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
</bean>
```

6.5.2. 代码

接口:

```
public interface IUserDao {  
    public void queryAllUser();  
}
```

实现类:

```
//SqlSessionDaoSupport:mybatis提供整合Spring的类, 里面已经有sqlSession和sqlSessionFactory  
public class ImpUserDao extends SqlSessionDaoSupport implements IUserDao {  
    @Override  
    public List<User> queryAllUser() {  
        return this.getSqlSession().selectList("com.lx.ssm.mapper.UserMapper.selectAllUser");  
    }  
}
```

6.5.3. 测试

```
public static void main(String[] args) {  
    ApplicationContext app=new ClassPathXmlApplicationContext("spring/applicationContext.xml");  
    IUserDao user=(ImpUserDao)app.getBean("userDao");  
    List<User> list=user.queryAllUser();  
    for (int i = 0; i < list.size(); i++) {  
        System.out.println(list.get(i).getName());  
    }  
}
```

6.5.4. Mapper 代理开发

- 1、 mapper.java 接口名要和 mapper.xml 文件名保持一致
- 2、 mapper 接口的方法名要和 statement id 一致
- 3、 mapper 接口的方法参数要和 statement 的 parameterType 一致
- 4、 mapper 接口的方法返回值和 statement 的 resultType 一致
- 5、 mapper.xml 的 namespace 要和 mapper.java 接口的全路径一致

6.5.4.1. spring 单个 mybatis 的 mapper 配置

```
<!-- mapper配置  
    MapperFactoryBean:根据mapper接口生成代理对象  
-->  
<bean id="userMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">  
    <!-- 指定你的mapper接口, 因为spring要帮你创建代理对象 -->  
    <property name="mapperInterface" value="com.lx.ssm.mapper.UserMapper"></property>  
    <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>  
</bean>
```


6.5.4.2. spring 批量扫描 mybatis 的 mapper 接口

```
<!-- mapper批量扫描, 从mapper包中扫描出mapper接口, 自动创建代理对象并且在spring容器中注册
遵循规范: mapper.java和mapper.xml映射文件名字保持一致, 且在一个目录中,
自动扫描出来的mapper的bean的id为mapper类名(首字母小写)
-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

    <!-- 指定扫描的包名
    如果扫描多个包, 每个包中间使用半角逗号分隔 -->
    <property name="basePackage" value="com.lx.ssm.mapper"></property>
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"></property>
</bean>
```

注意: 用了这个批量扫描以后, sqlMapConfig.xml 里面不需要加载 mapper.xml

6.5.4.3. 代码测试

```
public static void main(String[] args) {
    ApplicationContext app=new ClassPathXmlApplicationContext("spring/applicationCo
    UserMapper user=(UserMapper)app.getBean("userMapper");
    List<User> list=user.selectAllUser();
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i).getName());
    }
}
```

6.5.5. 整合不要 mybatis 主配置文件

6.5.5.1. Settings 部分

```
<!--配置mybatissetting-->
<bean id="settings" class="org.apache.ibatis.session.Configuration" --> 相当于原来的settings
    <property name="logImpl" value="org.apache.ibatis.logging.log4j.Log4jImpl"></property>
</bean>
<!--sqlsessioninf-->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <property name="typeAliasesPackage" value="com.hy.ssm.bean"></property>
    <property name="configuration" ref="settings"></property>
</bean>
```

6.5.5.2. 别名部分

```
<!--sqlsessioninf-->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <property name="typeAliasesPackage" value="com.hy.ssm.bean"></property>
    <property name="configuration" ref="settings"></property>
</bean>
```

6.5.5.3. Plugins

```
<!-- 配置sessionFactory -->
<bean id="sessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <property name="configuration" ref="configuration"></property>
    <property name="typeAliasesPackage" value="com.hy.ssm.bean"></property>
    <property name="plugins">
        <list>
            <bean class="com.github.pagehelper.PageInterceptor"></bean>
        </list>
    </property>
</bean>
```

6.5.5.4. Mapper.xml 部分

```
<!-- 批量扫描mapper -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.hy.ssm.mapper"></property>
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"></property>
</bean>
```

批量扫描mapper同时加载xml

6.5.5.5. 拦截器配置

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource"></property>
    <property name="typeAliasesPackage" value="com.hy.ssm.bean"></property>
    <property name="configuration" ref="settings"></property>
    <property name="plugins">
        <array>
            <bean class="com.hy.ssm.XxxInterceptor"></bean>
        </array>
    </property>
</bean>
```

6.5.6. 配置事物管理器

7. 逆向工程

7.1. 什么是逆向工程


Mybatis 需要程序员自己编写 sql 语句, mybatis 官方提供逆向工程 可以针对表单自动生成 mybatis 执行所需要的代码

Mapper.java

Mapper.xml

Po

7.2. 下载逆向工程

 generatorSqlmapCustom

7.3. 使用方法

修改工程根目录下面的 generatorConfig.xml 文件。

- 1、修改数据库连接
- 2、修改 po 的生成位置
- 3、修改 mapper.xml 的生成位置
- 4、修改 mapper.java 的生成位置

```
//自定义条件查询
@Test
public void testSelectByExample() {
    ItemsExample itemsExample = new ItemsExample();
    //通过criteria构造查询条件
    ItemsExample.Criteria criteria = itemsExample.createCriteria();
    criteria.andNameEqualTo("笔记本3");
    |
    itemsMapper.selectByExample(itemsExample);
}
```

```
InputStream in=Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory fact=new SqlSessionFactoryBuilder().build(in);

    SqlSession session=fact.openSession();

    StudentMapper mapper=session.getMapper(StudentMapper.class);

    Student s=mapper.selectByPrimaryKey(2);

    StudentExample se=new StudentExample();
    StudentExample.Criteria c=se.createCriteria();
    c.andStunameLike("%张%");
    c.andSexEqualTo("1");

    List<Student> list=mapper.selectByExample(se);
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i).getStuname());
    }
    System.out.println(list);
```

8. Mybatis plus

8.1. 简介

Mybatis-Plus（简称 MP）是一个 **Mybatis** 的增强工具，在 **Mybatis** 的基础上只做增强不做改变，为简化开发、提高效率而生。

8.2. 特性

- 无侵入：Mybatis-Plus 在 Mybatis 的基础上进行扩展，只做增强不做改变，引入 Mybatis-Plus 不会对您现有的 Mybatis 构架产生任何影响，而且 MP 支持所有 Mybatis 原生的特性
- 依赖少：仅仅依赖 Mybatis 以及 Mybatis-Spring
- 损耗小：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作
- 预防 Sql 注入：内置 Sql 注入剥离器，有效预防 Sql 注入攻击
- 通用 CRUD 操作：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分 CRUD 操作，更有强大的条件构造器，满足各类使用需求
- 多种主键策略：支持多达 4 种主键策略（内含分布式唯一 ID 生成器），可自由配置，完美解决主键问题
- 支持 ActiveRecord：支持 ActiveRecord 形式调用，实体类只需继承 Model 类即可实现基本 CRUD 操作
- 支持代码生成：采用代码或者 Maven 插件可快速生成 Mapper、Model、Service、Controller 层代码，支持模板引擎，更有超多自定义配置等您来使用（P.S. 比 Mybatis 官方的 Generator 更加强大！）
- 支持自定义全局通用操作：支持全局通用方法注入（Write once, use anywhere）
- 支持关键词自动转义：支持数据库关键词（order、key……）自动转义，还可自定义关键词
- 内置分页插件：基于 Mybatis 物理分页，开发者无需关心具体操作，配置好插件之后，写分页等同于普通 List 查询
- 内置性能分析插件：可输出 Sql 语句以及其执行时间，建议开发测试时启用该功能，能有效解决慢查询
- 内置全局拦截插件：提供全表 delete、update 操作智能分析阻断，预防误操作

8.3. 快速入门

1、单表 CURD（简单 + 批量）操作，自动完成（支持 like 比较等查询）。

2、分页插件，Count 查询 自动 或 自定义 SQL 查询。

3、Spring 根据不同环境加载不同配置支持（支持 typeAliasesPackage 通配符扫描）。

【自动生成 Entity Mapper Service 文件】

8.3.1. 需求

假设我们已存在一张 **User** 表，且已有对应的实体类 **User**，实现 **User** 表的 **CRUD** 操作我们需要做什么呢？

```
/** User 对应的 Mapper 接口 */
```

```
public interface UserMapper extends BaseMapper<User> { }
```

以上就是您所需的所有操作，甚至不需要您创建 XML 文件，我们如何使用它呢？

8.3.2. 引入 jar 包

```
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus</artifactId>
  <version>3.0-RELEASE</version>
</dependency>
```

8.3.3. 实体类

```
@TableName(value = "student")
public class Student {
    @TableId(value = "stuno", type = IdType.UUID)
    private String stuno;

    @TableField(value = "stuname")
    private String stuname1;

    private String pwd;

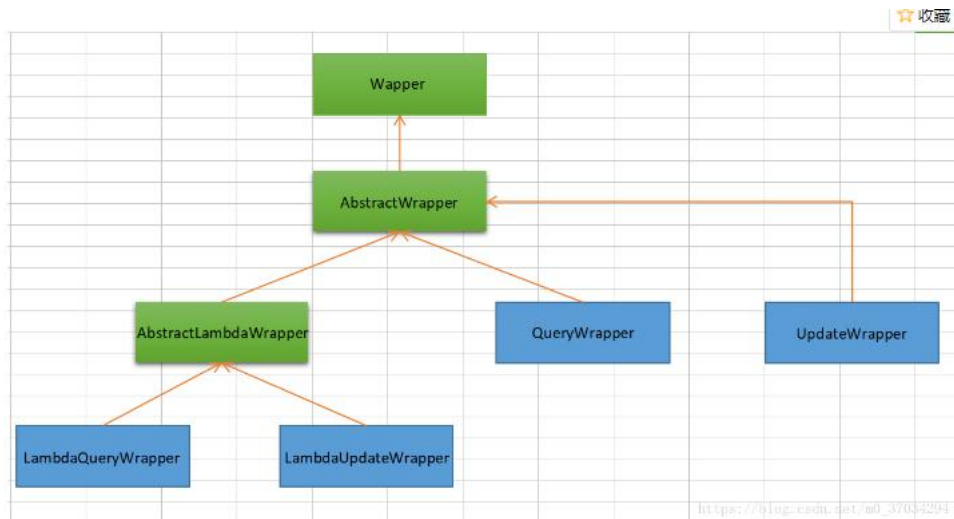
    private Integer classno;
```

8.3.4. 基本 CRUD

```
// 初始化 影响行数
int result = 0;
// 初始化 User 对象
User user = new User();
// 插入 User (插入成功会自动回写主键到实体类)
user.setName("Tom");
result = userMapper.insert(user);
// 更新 User
user.setAge(18);
result = userMapper.updateById(user);
// 查询 User
User exampleUser = userMapper.selectById(user.getId());
// 查询姓名为‘张三’的所有用户记录
```

```
List<User> userList = userMapper.selectList( new EntityWrapper<User>().eq("name", "张三") );
// 删除 User
result = userMapper.deleteById(user.getId());
```

8.3.5. 条件构造器



```
QueryWrapper queryWrapper=new QueryWrapper();
queryWrapper.eq( column: "stuname", val: "二黑");
List<Student> list=studentMapper.selectList(queryWrapper);
for(Student student1: list){
    System.out.println(student1.getStuname1());
}
```

条件构造器使用中的各个方法格式和说明，如有不懂可参考官方文档：

<https://mp.baomidou.com/guide/wrapper.html#abstractwrapper>

8.3.6. 自定义 sql

只需要在 mapper 里面直接添加方法即可（跟原生态 mybatis 一样）用 xml 或者注解都可以。

```
/**
 * @Author: wangsq
 * @Date: 2019/1/3 16:13
 * @Description:
 */
public interface StudentMapper extends BaseMapper<Student>{
    @Results({@Result(column = "stuname",property = "stuname1")})
    @Select("select * from student")
    public List<Student> queryAll();
}
```

8.3.7. 分页查询

8.3.7.1. 配置分页插件

```
<bean id="sqlSessionFactory"
class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource"></property>
  <property name="plugins">
    <array>
      <!-- 分页插件配置 -->
      <bean id="paginationInterceptor"
class="com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor"/>
      <!-- 性能拦截器，兼打印sql，不建议生产环境配置-->
      <bean id="performanceInterceptor"
class="com.baomidou.mybatisplus.extension.plugins.PerformanceInterceptor"/>
    </array>
  </property>
</bean>
```

8.3.7.2. 测试

```
QueryWrapper queryWrapper=new QueryWrapper();
queryWrapper.eq( column: "stuname", val: "二黑");
Page<Student> page=new Page<Student>( current: 5, size: 4);
IPage<Student> i1st= studentMapper.selectPage(page,queryWrapper);
for(Student student1: list.getRecords()){
    System.out.println(student1.getStuname1());
}
```

8.3.7.3. 自定义分页

```
public interface StudentMapper extends BaseMapper<Student>{

    /**
     * 3.x 的 page 可以进行取值, 多个入参记得加上注解
     * 自定义 page 类必须放在入参第一位
     * 返回值可以用 IPage<T> 接收 也可以使用入参的 MyPage<T> 接收
     * <li> 3.1.0 之前的版本使用注解会报错, 写在 xml 里就没事 </li>
     * <li> 3.1.0 开始支持注解, 但是返回值只支持 IPage , 不支持 IPage 的子类</li>
     *
     * @param page
     * @return 分页数据
     */
    public IPage<Student> queryAll(Page<Student> page);
}
```

```

@Service(value="studentService")
@Transactional
public class StudentService extends ServiceImpl<StudentMapper,Student> implements IStudentService{
    @Autowired
    private StudentMapper studentMapper;

    @Override
    public IPage<Student> queryAll(Integer pageNum) {
        if(pageNum==null){
            pageNum=1;
        }
        Page<Student> page=new Page<Student>(pageNum, size: 3);
        studentMapper.queryAll(page);
        return page;
    }
}

```

8.3.7.4. 自定义动态 sql 加 mybatisplus 分页

```

@Mapper
public interface StudentMapper1 extends BaseMapper<Student>{
    @SelectProvider(type = StudentSql.class,method = "queryAll")
    public List<StudentBo> queryAll(IPage<StudentBo> page, @Param("stu") Student student);
}

```

```

public class StudentSql {
    public String queryAll(@Param("stu") Student student){
        StringBuffer sql=new StringBuffer("select * from student s left join classes c on s.classno=c.cno w
        if(!StringUtils.isEmpty(student) && !StringUtils.isEmpty(student.getStuName())){
            sql.append(" and stuname like '"+student.getStuName()+"'");
        }
        if(!StringUtils.isEmpty(student) && !StringUtils.isEmpty(student.getSex()) && !"1".equals(student.g
            sql.append(" and sex='"+student.getSex());
        }
        if(!StringUtils.isEmpty(student) && !StringUtils.isEmpty(student.getClassno()) && student.getClassno
            sql.append(" and classno='"+student.getClassno());
        }
        return sql.toString();
    }
}

```

```

@Override
public LayUiData queryAll(Integer page,Integer limit,Student student) {
    IPage<StudentBo> p=new Page<StudentBo>(page,limit);
    List<StudentBo> list=studentMapper1.queryAll(p,student);
    p.setRecords(list);
    LayUiData layUiData=new LayUiData();
    layUiData.setCode(0);
    layUiData.setMsg("");
    layUiData.setCount(Integer.parseInt(String.valueOf(p.getTotal())));
    layUiData.setData(list);
    return layUiData;
}

```

8.3.8. 与spring 集成

8.3.8.1. 需要把 sqlSessionSessionFactoryBean 换成 plus 的。

```
<bean id="sessionFactory"
class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource"></property>
  <property name="typeAliasesPackage" value="com.hy.ssm.bean"></property>
  <property name="configLocation" value="classpath:mybatis.xml"></property>
  <property name="plugins">
    <array>
      <!-- 分页插件-->
      <bean
class="com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor"></bean>
    </array>
  </property>
</bean>

<!-- 批量扫描 mapper-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
  <property name="basePackage" value="com.hy.ssm.mapper"></property>
  <property name="sqlSessionFactoryBeanName" value="sessionFactory"></property>
</bean>

<!-- 事物管理器-->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 开启事物的注解扫描-->
<tx:annotation-driven
transaction-manager="transactionManager"></tx:annotation-driven>
```

在上面的配置中,除了 mybatis 的常规配置,多了一个分页插件的配置和全局配置,mybatis-plus 提供了很方便的使用分页的插件,还有一个全局配置如下:

```
<bean id="globalConfig"
class="com.baomidou.mybatisplus.entity.GlobalConfiguration">
  <!--
    AUTO->`0` ("数据库 ID 自增")
  -->
```



```

        INPUT->`1` (用户输入 ID")
        ID_WORKER->`2` ("全局唯一 ID")
        UUID->`3` ("全局唯一 ID")
    -->
    <property name="idType" value="2" />
    <!--
        MYSQL->`mysql`
        ORACLE->`oracle`
        DB2->`db2`
        H2->`h2`
        HSQL->`hsql`
        SQLITE->`sqlite`
        POSTGRE->`postgresql`
        SQLSERVER2005->`sqlserver2005`
        SQLSERVER->`sqlserver`
    -->
    <!-- Oracle 需要添加该项 -->
    <!-- <property name="dbType" value="oracle" /> -->
    <!-- 全局表为下划线命名设置 true -->
    <property name="dbColumnUnderline" value="true" />
</bean>

```

8.3.8.2. 创建一个 user 表

```

@Table("user")
public class User implements Serializable {

    /** 用户 ID */
    private Long id;

    /** 用户名 */
    private String name;

    /** 用户年龄 */
    private Integer age;

    @TableField(exist = false)
    private String state;
}

```

这里有两个注解需要注意,第一是`@tableName("user")`,它是指定与数据库表的关联,这里的注解意味着你的数据库里应该有一个名为 **user** 的表与之对应,并且数据表的列名应该就是 **User** 类的属性,对于 **User** 类中有而 **user** 表中没有的属性需要加第二个注解`@TableField(exist = false)`,表示排除 **User** 类中的属性。

8.3.8.3. 新建 dao 层接口 UserMapper

```
/**
 * User 表数据库控制层接口
 */
public interface UserMapper extends BaseMapper<User> {
    @Select("selectUserList")
    List<User> selectUserList(Pagination page,String state);
}
```

dao 接口需要实现 Basemapper,这样就能够使用封装好的很多通用方法,另外也可以自己编写方法,@select 注解引用自第三步的 UserMapper 文件

8.3.8.4. 新建 userMapper 配置文件

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE mapper PUBLIC
"-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd"><mapper
namespace="com.baomidou.springmvc.mapper.system.UserMapper">

    <!-- 通用查询结果列-->
    <sql id="Base_Column_List">
        id, name, age
    </sql>

    <select id="selectUserList" resultType="User">
        SELECT * FROM sys_user WHERE state=#{state}
    </select></mapper>
```

8.3.8.5. 新建 service 层类 UserService

```
/**
 *
 * User 表数据服务层接口实现类
 */
@Service
public class UserService extends ServiceImpl<UserMapper, User>{
    public Page<User> selectUserPage(Page<User> page, String state) {
        page.setRecords(baseMapper.selectUserList(page,state));
        return page;
    }
}
```

UserService 继承了 ServiceImpl 类,mybatis-plus 通过这种方式为我们注入了 UserMapper,这样可以使用 service 层默认为我们提供的很多方法,也可以调用我们自己在 dao 层编写的操作数据库的方法.Page 类是 mybatis-plus 提供分页功能的一个 model,继承了 Pagination,这样我们也不需要自己再编写一个 Page 类,直接使用即可。

8.3.8.6. 新建 controller 层 UserController

```
@Controller
public class UserController extends BaseController {

    @Autowired
    private IUserService userService;

    @ResponseBody
    @RequestMapping("/page")
    public Object selectPage(Model model){

        Page page=new Page(1,10);
        page = userService.selectUserPage(page, "NORMAL");
        return page;
    }
}
```

8.3.9. 代码生成器

AutoGenerator 是 MyBatis-Plus 的代码生成器,通过 AutoGenerator 可以快速生成 Entity、Mapper、Mapper XML、Service、Controller 等各个模块的代码,极大的提升了开发效率。

// 演示例子,执行 main 方法控制台输入模块表名回车自动生成对应项目目录中

```
public class CodeGenerator {

    /**
     * <p>
     * 读取控制台内容
     * </p>
     */
    public static String scanner(String tip) {
        Scanner scanner = new Scanner(System.in);
        StringBuilder help = new StringBuilder();
        help.append("请输入" + tip + "：");
        System.out.println(help.toString());
        if (scanner.hasNext()) {
            String ipt = scanner.next();
            if (StringUtils.isNotEmpty(ipt)) {
                return ipt;
            }
        }
    }
}
```

```

    }
}
throw new MybatisPlusException("请输入正确的" + tip + "! ");
}

public static void main(String[] args) {
    // 代码生成器
    AutoGenerator mpg = new AutoGenerator();

    // 全局配置
    GlobalConfig gc = new GlobalConfig();
    String projectPath = System.getProperty("user.dir");
    gc.setOutputDir(projectPath + "/src/main/java");
    gc.setAuthor("jobob");
    gc.setOpen(false);
    mpg.setGlobalConfig(gc);

    // 数据源配置
    DataSourceConfig dsc = new DataSourceConfig();
    dsc.setUrl("jdbc:mysql://localhost:3306/ant?useUnicode=true&useSSL=false&characterEncoding=utf8");
    // dsc.setSchemaName("public");
    dsc.setDriverName("com.mysql.jdbc.Driver");
    dsc.setUsername("root");
    dsc.setPassword("密码");
    mpg.setDataSource(dsc);

    // 包配置
    PackageConfig pc = new PackageConfig();
    pc.setModuleName(scanner("模块名"));
    pc.setParent("com.baomidou.ant");
    mpg.setPackageInfo(pc);

    // 自定义配置
    InjectionConfig cfg = new InjectionConfig() {
        @Override
        public void initMap() {
            // to do nothing
        }
    };

    // 如果模板引擎是 freemarker
    String templatePath = "/templates/mapper.xml.ftl";

```

```
// 如果模板引擎是 velocity
// String templatePath = "/templates/mapper.xml.vm";

// 自定义输出配置
List<FileOutConfig> focList = new ArrayList<>();
// 自定义配置会被优先输出
focList.add(new FileOutConfig(templatePath) {
    @Override
    public String outputFile(TableInfo tableInfo) {
        // 自定义输出文件名
        return projectPath + "/src/main/resources/mapper/" + pc.getModuleName()
            + "/" + tableInfo.getEntityName() + "Mapper" + StringPool.DOT_XML;
    }
});

cfg.setFileOutConfigList(focList);
mpg.setCfg(cfg);

// 配置模板
TemplateConfig templateConfig = new TemplateConfig();

// 配置自定义输出模板
// templateConfig.setEntity();
// templateConfig.setService();
// templateConfig.setController();

templateConfig.setXml(null);
mpg.setTemplate(templateConfig);

// 策略配置
StrategyConfig strategy = new StrategyConfig();
strategy.setNaming(NamingStrategy.underline_to_camel);
strategy.setColumnNaming(NamingStrategy.underline_to_camel);
strategy.setSuperEntityClass("com.baomidou.ant.common.BaseEntity");
strategy.setEntityLombokModel(true);
strategy.setRestControllerStyle(true);
strategy.setSuperControllerClass("com.baomidou.ant.common.BaseController");
strategy.setInclude(scanner("表名"));
strategy.setSuperEntityColumns("id");
strategy.setControllerMappingHyphenStyle(true);
strategy.setTablePrefix(pc.getModuleName() + "_");
mpg.setStrategy(strategy);
mpg.setTemplateEngine(new FreemarkerTemplateEngine());
mpg.execute();
```

