# Mybatis

**第⼀部分：⾃定义持久层框架**

## 分析JDBC操作问题

public static void main(String[] args) { Connection connection = null;

PreparedStatement preparedStatement = null; ResultSet resultSet = null;

try {

// 加载数据库驱动

Class.forName("com.mysql.jdbc.Driver");

// 通过驱动管理类获取数据库链接

connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis? characterEncoding=utf-8", "root", "root");

// 定义sql语句？表示占位符

String sql = "select \* from user where username = ?";

// 获取预处理statement

preparedStatement = connection.prepareStatement(sql);

// 设置参数，第⼀个参数为sql语句中参数的序号(从1开始)，第⼆个参数为设置的参数值

preparedStatement.setString(1, "tom");

// 向数据库发出sql执⾏查询，查询出结果集

resultSet = preparedStatement.executeQuery();

// 遍历查询结果集

while (resultSet.next()) {

int id = resultSet.getInt("id");

String username = resultSet.getString("username");

// 封 装 User user.setId(id);

user.setUsername(username);

}

System.out.println(user);

}

} catch (Exception e) { e.printStackTrace();

} finally {

// 释放资源

if (resultSet != null) { try {

resultSet.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (preparedStatement != null) { try {

preparedStatement.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (connection != null) { try {

connection.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

#### JDBC问题总结：

原始jdbc开发存在的问题如下：

1、 数据库连接创建、释放频繁造成系统资源浪费，从⽽影响系统性能。

2、 Sql语句在代码中硬编码，造成代码不易维护，实际应⽤中sql变化的可能较⼤，sql变动需要改变

java代码。

3、 使⽤preparedStatement向占有位符号传参数存在硬编码，因为sql语句的where条件不⼀定，可能多也可能少，修改sql还要修改代码，系统不易维护。

4、 对结果集解析存在硬编码(查询列名)，sql变化导致解析代码变化，系统不易维护，如果能将数据 库记录封装成pojo对象解析⽐较⽅便

* 1. **问题解决思路**

①使⽤数据库连接池初始化连接资源

②将sql语句抽取到xml配置⽂件中

③使⽤反射、内省等底层技术，⾃动将实体与表进⾏属性与字段的⾃动映射

* 1. **⾃定义框架设计**

**使⽤端：**

提供核⼼配置⽂件：

sqlMapConfig.xml : 存放数据源信息，引⼊mapper.xml Mapper.xml : sql语句的配置⽂件信息

**框架端：**

1. 读取配置⽂件

读取完成以后以流的形式存在，我们不能将读取到的配置信息以流的形式存放在内存中，不好操作，可 以创建javaBean来存储

1. Configuration : 存放数据库基本信息、Map<唯⼀标识，Mapper> 唯⼀标识：namespace + "."

+ id

1. MappedStatement：sql语句、statement类型、输⼊参数java类型、输出参数java类型
2. 解析配置⽂件

创建sqlSessionFactoryBuilder类：

⽅法：sqlSessionFactory build()：

第⼀：使⽤dom4j解析配置⽂件，将解析出来的内容封装到Configuration和MappedStatement中 第⼆：创建SqlSessionFactory的实现类DefaultSqlSession

1. 创建SqlSessionFactory：

⽅法：openSession() : 获取sqlSession接⼝的实现类实例对象

1. 创建sqlSession接⼝及实现类：主要封装crud⽅法

⽅法：selectList(String statementId,Object param)：查询所有selectOne(String statementId,Object param)：查询单个

具体实现：封装JDBC完成对数据库表的查询操作

**涉及到的设计模式：**

Builder构建者设计模式、⼯⼚模式、代理模式

* 1. **⾃定义框架实现**

在使⽤端项⽬中创建配置配置⽂件创建 sqlMapConfig.xml

〈configuration〉

<!--数据库连接信息-->

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///zdy\_mybatis"></property>

<property name="user" value="root"></property>

<property name="password" value="root"></property>

<! --引⼊sql配置信息-->

<mapper resource="mapper.xml"></mapper>

</configuration>

mapper.xml

<mapper namespace="User">

<select id="selectOne" paramterType="com.lagou.pojo.User" resultType="com.lagou.pojo.User">

select \* from user where id = #{id} and username =#{username}

</select>

<select id="selectList" resultType="com.lagou.pojo.User"> select \* from user

</select>

</mapper>

User实体

public class User {

//主键标识

private Integer id;

//⽤户名

private String username;

public Integer getId() { return id;

}

public void setId(Integer id) { this.id = id;

}

public String getUsername() { return username;

}

public void setUsername(String username) { this.username = username;

}

@Override

public String toString() { return "User{" +

"id=" + id +

", username='" + username + '\'' + '}';

}

}

再创建⼀个Maven⼦⼯程并且导⼊需要⽤到的依赖坐标

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.encoding>UTF-8</maven.compiler.encoding>

<java.version>1.8</java.version>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.17</version>

</dependency>

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.1.2</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.12</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

</dependency>

<dependency>

<groupId>dom4j</groupId>

<artifactId>dom4j</artifactId>

<version>1.6.1</version>

</dependency>

<dependency>

<groupId>jaxen</groupId>

<artifactId>jaxen</artifactId>

<version>1.1.6</version>

</dependency>

</dependencies>

Configuration

public class Configuration {

//数据源

private DataSource dataSource;

//map集合： key:statementId value:MappedStatement

private Map<String,MappedStatement> mappedStatementMap = new HashMap<String, MappedStatement>();

public DataSource getDataSource() { return dataSource;

}

public void setDataSource(DataSource dataSource) { this.dataSource = dataSource;

}

public Map<String, MappedStatement> getMappedStatementMap() { return mappedStatementMap;

}

public void setMappedStatementMap(Map<String, MappedStatement> mappedStatementMap) {

this.mappedStatementMap = mappedStatementMap;

}

}

MappedStatement

public class MappedStatement {

//id

private Integer id;

//sql语句

private String sql;

//输⼊参数

private Class<?> paramterType;

//输出参数

private Class<?> resultType; public Integer getId() {

return id;

}

public void setId(Integer id) { this.id = id;

}

public String getSql() { return sql;

}

public void setSql(String sql) { this.sql = sql;

}

public Class<?> getParamterType() { return paramterType;

}

public void setParamterType(Class<?> paramterType) { this.paramterType = paramterType;

}

public Class<?> getResultType() { return resultType;

}

public void setResultType(Class<?> resultType) { this.resultType = resultType;

}

}

Resources

public class Resources {

public static InputStream getResourceAsSteam(String path){ InputStream resourceAsStream =

Resources.class.getClassLoader.getResourceAsStream(path); return resourceAsStream;

}

}

SqlSessionFactoryBuilder

public class SqlSessionFactoryBuilder { private Configuration configuration; public SqlSessionFactoryBuilder() {

this.configuration = new Configuration();

}

public SqlSessionFactory build(InputStream inputStream) throws DocumentException, PropertyVetoException, ClassNotFoundException {

//1.解析配置⽂件，封装Configuration XMLConfigerBuilder

xmlConfigerBuilder = new

XMLConfigerBuilder(configuration); Configuration configuration =

xmlConfigerBuilder.parseConfiguration(inputStream);

//2. 创 建 sqlSessionFactory SqlSessionFactory sqlSessionFactory = new

DefaultSqlSessionFactory(configuration);

return sqlSessionFactory;

}

XMLConfigerBuilder

public class XMLConfigerBuilder {

private Configuration configuration;

public XMLConfigerBuilder(Configuration configuration) { this.configuration = new Configuration();

}

public Configuration parseConfiguration(InputStream inputStream) throws DocumentException, PropertyVetoException, ClassNotFoundException {

Document document = new SAXReader().read(inputStream);

//<configuation>

Element rootElement = document.getRootElement(); List<Element> propertyElements =

rootElement.selectNodes("//property");

Properties properties = new Properties();

for (Element propertyElement : propertyElements) { String name = propertyElement.attributeValue("name");

String value = propertyElement.attributeValue("value"); properties.setProperty(name,value);

}

//连接池

ComboPooledDataSource comboPooledDataSource = new ComboPooledDataSource();

comboPooledDataSource.setDriverClass(properties.getProperty("driverClass")); comboPooledDataSource.setJdbcUrl(properties.getProperty("jdbcUrl")); comboPooledDataSource.setUser(properties.getProperty("username")); comboPooledDataSource.setPassword(properties.getProperty("password"));

// 填 充 configuration configuration.setDataSource(comboPooledDataSource);

//mapper 部 分

List<Element> mapperElements = rootElement.selectNodes("//mapper"); XMLMapperBuilder xmlMapperBuilder = new

XMLMapperBuilder(configuration);

for (Element mapperElement : mapperElements) {

String mapperPath = mapperElement.attributeValue("resource"); InputStream resourceAsSteam =

Resources.getResourceAsSteam(mapperPath); xmlMapperBuilder.parse(resourceAsSteam);

}

return configuration;

}

XMLMapperBuilder

public class XMLMapperBuilder {

private Configuration configuration;

public XMLMapperBuilder(Configuration configuration) {

this.configuration = configuration;

}

public void parse(InputStream inputStream) throws DocumentException, ClassNotFoundException {

Document document = new SAXReader().read(inputStream); Element rootElement = document.getRootElement();

数class

String namespace = rootElement.attributeValue("namespace"); List<Element> select = rootElement.selectNodes("select"); for (Element element : select) { //id的值

String id = element.attributeValue("id");

String paramterType = element.attributeValue("paramterType"); String resultType = element.attributeValue("resultType"); //输⼊参

Class<?> paramterTypeClass = getClassType(paramterType);

//返回结果class

Class<?> resultTypeClass = getClassType(resultType);

//statementId

String key = namespace + "." + id;

//sql语句

String textTrim = element.getTextTrim();

//封装 mappedStatement

MappedStatement mappedStatement = new MappedStatement(); mappedStatement.setId(id); mappedStatement.setParamterType(paramterTypeClass); mappedStatement.setResultType(resultTypeClass); mappedStatement.setSql(textTrim);

//填充 configuration

configuration.getMappedStatementMap().put(key, mappedStatement);

private Class<?> getClassType (String paramterType) throws ClassNotFoundException {

Class<?> aClass = Class.forName(paramterType); return aClass;

}

}

sqlSessionFactory 接⼝及D efaultSqlSessionFactory 实现类

public interface SqlSessionFactory { public SqlSession openSession();

}

public class DefaultSqlSessionFactory implements SqlSessionFactory { private Configuration configuration;

public DefaultSqlSessionFactory(Configuration configuration) { this.configuration = configuration;

}

public SqlSession openSession(){

return new DefaultSqlSession(configuration);

}

}

sqlSession 接⼝及 DefaultSqlSession 实现类

public interface SqlSession {

public <E> List<E> selectList(String statementId, Object... param) Exception;

public <T> T selectOne(String statementId,Object... params) throws Exception;

public void close() throws SQLException;

}

public class DefaultSqlSession implements SqlSession { private Configuration configuration;

public DefaultSqlSession(Configuration configuration) { this.configuration = configuration;

//处理器对象

private Executor simpleExcutor = new SimpleExecutor();

public <E > List < E > selectList(String statementId, Object...param) throws Exception {

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

List<E> query = simpleExcutor.query(configuration, mappedStatement, param);

return query;

}

//selectOne 中 调 ⽤ selectList

public <T > T selectOne(String statementId, Object...params) throws Exception {

List<Object> objects = selectList(statementId, params); if (objects.size() == 1) {

return (T) objects.get(0);

} else {

throw new RuntimeException("返回结果过多");

}

}

public void close () throws SQLException { simpleExcutor.close();

}

}

Executor

public interface Executor {

<E> List<E> query(Configuration configuration, MappedStatement mappedStatement,Object[] param) throws Exception;

void close() throws SQLException;

}

SimpleExecutor

public class SimpleExecutor implements Executor { private Connection connection = null;

public <E> List<E> query(Configuration configuration, MappedStatement mappedStatement, Object[] param) throws SQLException, NoSuchFieldException, IllegalAccessException, InstantiationException, IntrospectionException, InvocationTargetException {

//获取连接

connection = configuration.getDataSource().getConnection();

// select \* from user where id = #{id} and username = #{username} String sql = mappedStatement.getSql();

//对sql进⾏处理

BoundSql boundsql = getBoundSql(sql);

// select \* from where id = ? and username = ? String finalSql = boundsql.getSqlText();

//获取传⼊参数类型

Class<?> paramterType = mappedStatement.getParamterType();

//获取预编译preparedStatement对象PreparedStatement preparedStatement =

connection.prepareStatement(finalSql); List<ParameterMapping> parameterMappingList =

boundsql.getParameterMappingList();

for (int i = 0; i < parameterMappingList.size(); i++) { ParameterMapping parameterMapping = parameterMappingList.get(i); String name = parameterMapping.getName();

//反射

Field declaredField = paramterType.getDeclaredField(name); declaredField.setAccessible(true);

//参数的值

Object o = declaredField.get(param[0]);

//给占位符赋值preparedStatement.setObject(i + 1, o);

}

ResultSet resultSet = preparedStatement.executeQuery(); Class<?> resultType = mappedStatement.getResultType(); ArrayList<E> results = new ArrayList<E>();

while (resultSet.next()) {

ResultSetMetaData metaData = resultSet.getMetaData();

(E) resultType.newInstance();

int columnCount = metaData.getColumnCount(); for (int i = 1; i <= columnCount; i++) {

//属性名

String columnName = metaData.getColumnName(i);

//属性值

Object value = resultSet.getObject(columnName);

//创建属性描述器，为属性⽣成读写⽅法PropertyDescriptor propertyDescriptor = new

PropertyDescriptor(columnName, resultType);

//获取写⽅法

Method writeMethod = propertyDescriptor.getWriteMethod();

//向类中写⼊值writeMethod.invoke(o, value);

}

results.add(o);

}

return results;

}

@Override

public void close() throws SQLException { connection.close();

}

private BoundSql getBoundSql(String sql) {

//标记处理类：主要是配合通⽤标记解析器GenericTokenParser类完成对配置⽂件等的解析⼯作，其中TokenHandler主要完成处理

ParameterMappingTokenHandler parameterMappingTokenHandler = new ParameterMappingTokenHandler();

//GenericTokenParser :通⽤的标记解析器，完成了代码⽚段中的占位符的解析，然后再根

据给定的标记处理器(TokenHandler)来进⾏表达式的处理

//三个参数：分别为openToken (开始标记)、closeToken (结束标记)、handler (标记

处 理器)

GenericTokenParser genericTokenParser = new GenericTokenParser("# {",

"}", parameterMappingTokenHandler);

String parse = genericTokenParser.parse(sql); List<ParameterMapping> parameterMappings =

parameterMappingTokenHandler.getParameterMappings();

BoundSql boundSql = new BoundSql(parse, parameterMappings); return boundSql;

}

}

BoundSql

public class BoundSql {

// 解 析 过 后 的 sql 语 句 private String sqlText;

//解析出来的参数

private List<ParameterMapping> parameterMappingList = new

ArrayList<ParameterMapping>();

public BoundSql(String sqlText, List<ParameterMapping> parameterMappingList) {

this.sqlText = sqlText;

this.parameterMappingList = parameterMappingList;

}

public String getSqlText() { return sqlText;

}

public void setSqlText(String sqlText) { this.sqlText = sqlText;

}

public List<ParameterMapping> getParameterMappingList() { return parameterMappingList;

}

public void setParameterMappingList(List<ParameterMapping> parameterMappingList) {

this.parameterMappingList = parameterMappingList;

}

}

* 1. **⾃定义框架优化**

通过上述我们的⾃定义框架，我们解决了JDBC操作数据库带来的⼀些问题：例如频繁创建释放数据库连接，硬编码，⼿动封装返回结果集等问题，但是现在我们继续来分析刚刚完成的⾃定义框架代码，有没 有什么问题？

问题如下：

dao的实现类中存在重复的代码，整个操作的过程模板重复(创建sqlsession,调⽤sqlsession⽅ 法，关闭 sqlsession)

dao的实现类中存在硬编码，调⽤sqlsession的⽅法时，参数statement的id硬编码

解决：使⽤代理模式来创建接⼝的代理对象

@Test

public void test2() throws Exception {

InputStream resourceAsSteam = Resources.getResourceAsSteam(path： "sqlMapConfig.xml")

SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsSteam);

SqlSession sqlSession = build.openSession(); User user = new User();

user.setld(l); user.setUsername("tom");

//代理对象

UserMapper userMapper = sqlSession.getMappper(UserMapper.class); User userl = userMapper.selectOne(user);

System・out.println(userl);

}

在sqlSession中添加⽅法

public interface SqlSession {

public <T> T getMappper(Class<?> mapperClass);

实现类

@Override

public <T> T getMappper(Class<?> mapperClass) {

T o = (T) Proxy.newProxyInstance(mapperClass.getClassLoader(), new Class[] {mapperClass}, new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

// selectOne

String methodName = method.getName();

// className:namespace

String className = method.getDeclaringClass().getName();

//statementid

String key = className+"."+methodName; MappedStatement mappedStatement =

configuration.getMappedStatementMap().get(key);

Type genericReturnType = method.getGenericReturnType(); ArrayList arrayList = new ArrayList<> ();

//判断是否实现泛型类型参数化

if(genericReturnType instanceof ParameterizedType){ return selectList(key,args);

return selectOne(key,args);

}

});

return o;

}

# 第⼆部分：Mybatis相关概念

## 对象/关系数据库映射（ORM）

ORM全称Object/Relation Mapping：表示对象-关系映射的缩写

ORM完成⾯向对象的编程语⾔到关系数据库的映射。当ORM框架完成映射后，程序员既可以利⽤⾯向 对象程序设计语⾔的简单易⽤性，⼜可以利⽤关系数据库的技术优势。ORM把关系数据库包装成⾯向对 象的模型。ORM框架是⾯向对象设计语⾔与关系数据库发展不同步时的中间解决⽅案。采⽤ORM框架 后，应⽤程序不再直接访问底层数据库，⽽是以⾯向对象的⽅式来操作持久化对象，⽽ORM框架则将这 些⾯向对象的操作转换成底层SQL操作。ORM框架实现的效果：把对持久化对象的保存、修改、删除 等操作，转换为对数据库的操作

* 1. **Mybatis简介**

MyBatis是⼀款优秀的基于ORM的半⾃动轻量级持久层框架，它⽀持定制化SQL、存储过程以及⾼级映射。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。MyBatis可以使⽤简单的

XML或注解来配置和映射原⽣类型、接⼝和Java的POJO （Plain Old Java Objects,普通⽼式Java对 象）为数据库中的记录。

* 1. **Mybatis历史**

原是apache的⼀个开源项⽬iBatis, 2010年6⽉这个项⽬由apache software foundation 迁移到了

google code，随着开发团队转投Google Code旗下，ibatis3.x正式更名为Mybatis ，代码于2013年11

⽉迁移到Github。

iBATIS⼀词来源于“internet”和“abatis”的组合，是⼀个基于Java的持久层框架。iBATIS提供的持久层框架包括SQL Maps和Data Access Objects(DAO)

* 1. **Mybatis优势**

Mybatis是⼀个半⾃动化的持久层框架，对开发⼈员开说，核⼼sql还是需要⾃⼰进⾏优化，sql和java编码进⾏分离，功能边界清晰，⼀个专注业务，⼀个专注数据。

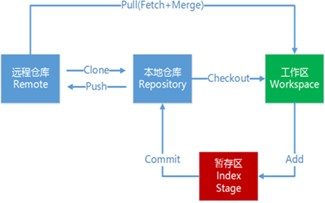
分析图示如下：



**第三部分：Mybatis基本应⽤**

**3.1 快速⼊⻔**

MyBatis官⽹地址：<http://www.mybatis.org/mybatis-3/>



**3.1.1 开发步骤：**

①添加MyBatis的坐标

②创建user数据表

③编写User实体类

④编写映射⽂件UserMapper.xml

⑤编写核⼼⽂件SqlMapConfig.xml

⑥编写测试类

**3.1.1 环境搭建：**

1. 导⼊MyBatis的坐标和其他相关坐标

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.encoding>UTF-8</maven.compiler.encoding>

<java.version>1.8</java.version>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<!--mybatis坐标-->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.4.5</version>

</dependency>

<!--mysql驱动坐标-->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.6</version>

<scope>runtime</scope>

</dependency>

<!--单元测试坐标-->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!--⽇志坐标-->

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.12</version>

</dependency>

1. 创建user数据表



1. 编写User实体

public class User { private int id;

private String username; private String password;

//省略get个set⽅法

}

1. 编写UserMapper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">](http://mybatis.org/dtd/mybatis-3-mapper.dtd)

<mapper namespace="userMapper">

<select id="findAll" resultType="com.lagou.domain.User"> select \* from User

</select>

</mapper>

1. 编写MyBatis核⼼⽂件

<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN“ ["http://mybatis.org/dtd/mybatis-3-config.dtd">](http://mybatis.org/dtd/mybatis-3-config.dtd)

<configuration>

<environments default="development">

<environment id="development">

<transactionManager type="JDBC"/>

<dataSource type="POOLED">

<property name="driver" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql:///test"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</dataSource>

</environment>

</environments>

<mappers>

<mapper resource="com/lagou/mapper/UserMapper.xml"/>

</mappers>

</configuration>

6） 编写测试代码

//加载核⼼配置⽂件

InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");

//获得sqlSession⼯⼚对象

SqlSessionFactory sqlSessionFactory = new

SqlSessionFactoryBuilder().build(resourceAsStream);

//获得sqlSession对象

SqlSession sqlSession = sqlSessionFactory.openSession();

//执⾏sql语句

List<User> userList = sqlSession.selectList("userMapper.findAll");

//打印结果System.out.println(userList);

//释放资源sqlSession.close();

* + 1. **MyBatis的增删改查操作**

**MyBatis的插⼊数据操作**

1. **编写UserMapper映射⽂件**

<mapper namespace="userMapper">

<insert id="add" parameterType="com.lagou.domain.User"> insert into user values(#{id},#{username},#{password})

</insert>

</mapper>

1. **编写插⼊实体User的代码**

InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml"); SqlSessionFactory sqlSessionFactory = new

SqlSessionFactoryBuilder().build(resourceAsStream); SqlSession sqlSession = sqlSessionFactory.openSession();

int insert = sqlSession.insert("userMapper.add", user); System.out.println(insert);

//提交事务

sqlSession.commit(); sqlSession.close();

1. **插⼊操作注意问题**

* 插⼊语句使⽤insert标签
* 在映射⽂件中使⽤parameterType属性指定要插⼊的数据类型

•Sql语句中使⽤#{实体属性名}⽅式引⽤实体中的属性值

* 插⼊操作使⽤的API是sqlSession.insert(“命名空间.id”,实体对象);
* 插⼊操作涉及数据库数据变化，所以要使⽤sqlSession对象显示的提交事务，即sqlSession.commit()
  + 1. **MyBatis的修改数据操作**

1. **编写UserMapper映射⽂件**

<mapper namespace="userMapper">

<update id="update" parameterType="com.lagou.domain.User">

update user set username=#{username},password=#{password} where id=#

{id}

</update>

</mapper>

1. **编写修改实体User的代码**

InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml"); SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream); SqlSession sqlSession = sqlSessionFactory.openSession(); int update = sqlSession.update("userMapper.update", user); System.out.println(update);

sqlSession.commit(); sqlSession.close();

1. **修改操作注意问题**

* 修改语句使⽤update标签
* 修改操作使⽤的API是sqlSession.update(“命名空间.id”,实体对象);
  + 1. **MyBatis的删除数据操作**

1. **编写UserMapper映射⽂件**

<mapper namespace="userMapper">

<delete id="delete" parameterType="java.lang.Integer"> delete from user where id=#{id}

</delete>

</mapper>

1. **编写删除数据的代码**

InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml"); SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream); SqlSession sqlSession = sqlSessionFactory.openSession(); int delete = sqlSession.delete("userMapper.delete",3); System.out.println(delete);

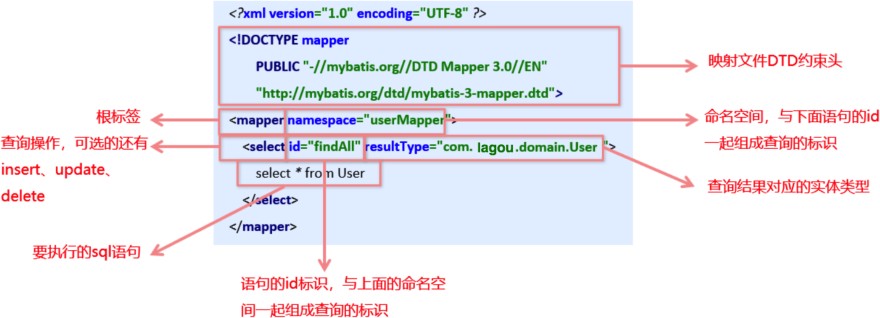
sqlSession.commit(); sqlSession.close();

1. **删除操作注意问题**

* 删除语句使⽤delete标签

•Sql语句中使⽤#{任意字符串}⽅式引⽤传递的单个参数

* 删除操作使⽤的API是sqlSession.delete(“命名空间.id”,Object);
  + 1. **MyBatis的映射⽂件概述**



* + 1. **⼊⻔核⼼配置⽂件分析：**

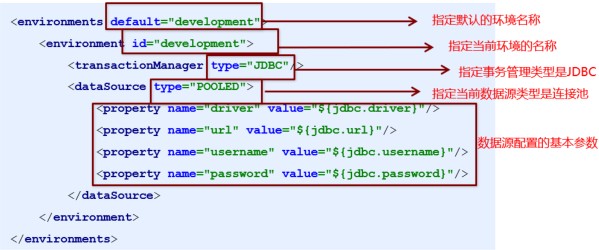
MyBatis核⼼配置⽂件层级关系



**MyBatis常⽤配置解析**

1. **environments标签**

数据库环境的配置，⽀持多环境配置



其中，事务管理器（transactionManager）类型有两种：

•JDBC：这个配置就是直接使⽤了JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务作

⽤域。

•MANAGED：这个配置⼏乎没做什么。它从来不提交或回滚⼀个连接，⽽是让容器来管理事务的整个⽣ 命周期（⽐如 JEE 应⽤服务器的上下⽂）。 默认情况下它会关闭连接，然⽽⼀些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻⽌它默认的关闭⾏为。

其中，数据源（dataSource）类型有三种：

•UNPOOLED：这个数据源的实现只是每次被请求时打开和关闭连接。

•POOLED：这种数据源的实现利⽤“池”的概念将 JDBC 连接对象组织起来。

•JNDI：这个数据源的实现是为了能在如 EJB 或应⽤服务器这类容器中使⽤，容器可以集中或在外部配置数据源，然后放置⼀个 JNDI 上下⽂的引⽤。

1. **mapper标签**

该标签的作⽤是加载映射的，加载⽅式有如下⼏种：

* 使⽤相对于类路径的资源引⽤，例如：

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

* 使⽤完全限定资源定位符（URL），例如：

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

* 使⽤映射器接⼝实现类的完全限定类名，例如：

<mapper class="org.mybatis.builder.AuthorMapper"/>

* 将包内的映射器接⼝实现全部注册为映射器，例如：

<package name="org.mybatis.builder"/>

* + 1. **Mybatis相应API介绍**

**SqlSession⼯⼚构建器SqlSessionFactoryBuilder**

常⽤API：SqlSessionFactory build(InputStream inputStream)

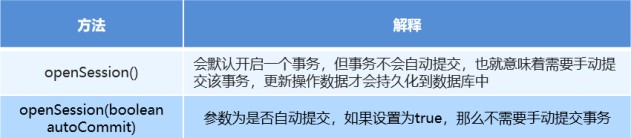
通过加载mybatis的核⼼⽂件的输⼊流的形式构建⼀个SqlSessionFactory对象

String resource = "org/mybatis/builder/mybatis-config.xml"; InputStream inputStream = Resources.getResourceAsStream(resource); SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder(); SqlSessionFactory factory = builder.build(inputStream);

其中， Resources ⼯具类，这个类在 org.apache.ibatis.io 包中。Resources 类帮助你从类路径下、⽂件系统或⼀个 web URL 中加载资源⽂件。

**SqlSession⼯⼚对象SqlSessionFactory**

SqlSessionFactory 有多个个⽅法创建SqlSession 实例。常⽤的有如下两个：



**SqlSession会话对象**

SqlSession 实例在 MyBatis 中是⾮常强⼤的⼀个类。在这⾥你会看到所有执⾏语句、提交或回滚事务和获取映射器实例的⽅法。

执⾏语句的⽅法主要有：

<T> T selectOne(String statement, Object parameter)

<E> List<E> selectList(String statement, Object parameter) int insert(String statement, Object parameter)

int update(String statement, Object parameter) int delete(String statement, Object parameter)

操作事务的⽅法主要有：

void commit() void rollback()

* 1. **Mybatis的Dao层实现**
     1. **传统开发⽅式**

编写UserDao接⼝

public interface UserDao {

List<User> findAll() throws IOException;

}

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;

}

}

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. **代理开发⽅式**

**代理开发⽅式介绍**

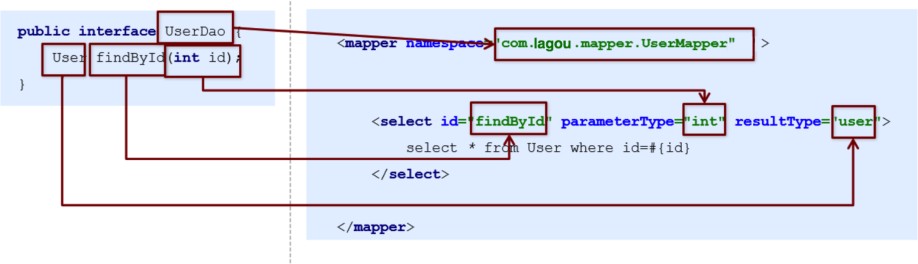
采⽤ Mybatis 的代理开发⽅式实现 DAO 层的开发，这种⽅式是我们后⾯进⼊企业的主流。

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**

编写UserMapper接⼝



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();

}

**第四部分：Mybatis配置⽂件深⼊**

* 1. **核⼼配置⽂件SqlMapConfig.xml**

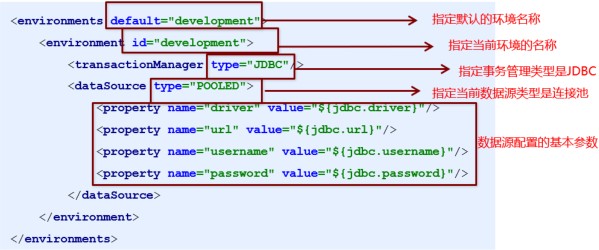
**4.1.1 MyBatis核⼼配置⽂件层级关系**



* 1. **MyBatis常⽤配置解析**

1. **environments标签**

数据库环境的配置，⽀持多环境配置



其中，事务管理器（transactionManager）类型有两种：

•JDBC：这个配置就是直接使⽤了JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务作

⽤域。

•MANAGED：这个配置⼏乎没做什么。它从来不提交或回滚⼀个连接，⽽是让容器来管理事务的整个⽣ 命周期（⽐如 JEE 应⽤服务器的上下⽂）。 默认情况下它会关闭连接，然⽽⼀些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻⽌它默认的关闭⾏为。

其中，数据源（dataSource）类型有三种：

•UNPOOLED：这个数据源的实现只是每次被请求时打开和关闭连接。

•POOLED：这种数据源的实现利⽤“池”的概念将 JDBC 连接对象组织起来。

•JNDI：这个数据源的实现是为了能在如 EJB 或应⽤服务器这类容器中使⽤，容器可以集中或在外部配置数据源，然后放置⼀个 JNDI 上下⽂的引⽤。

1. **mapper标签**

该标签的作⽤是加载映射的，加载⽅式有如下⼏种：

* 使⽤相对于类路径的资源引⽤，例如：

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

* 使⽤完全限定资源定位符（URL），例如：

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

* 使⽤映射器接⼝实现类的完全限定类名，例如：

<mapper class="org.mybatis.builder.AuthorMapper"/>

* 将包内的映射器接⼝实现全部注册为映射器，例如：

<package name="org.mybatis.builder"/>

1. **Properties标签**

实际开发中，习惯将数据源的配置信息单独抽取成⼀个properties⽂件，该标签可以加载额外配置的

properties⽂件



1. **typeAliases标签**

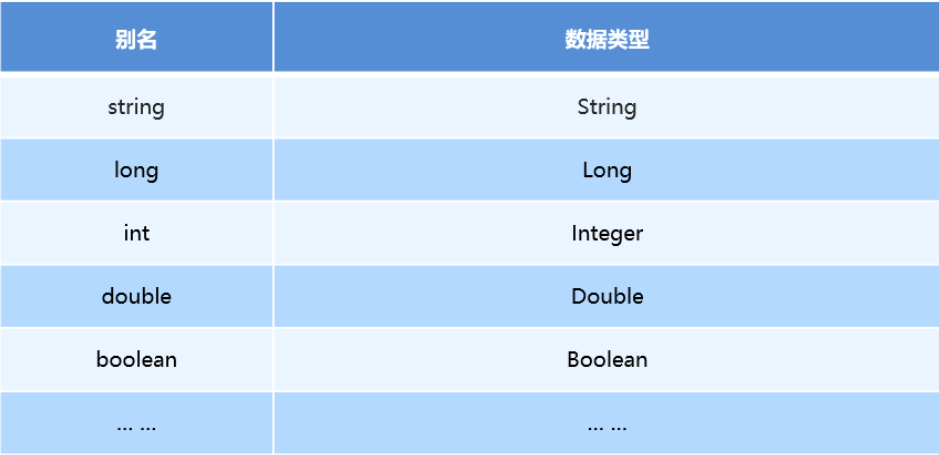
类型别名是为Java 类型设置⼀个短的名字。原来的类型名称配置如下



配置typeAliases，为com.lagou.domain.User定义别名为user



上⾯我们是⾃定义的别名，mybatis框架已经为我们设置好的⼀些常⽤的类型的别名



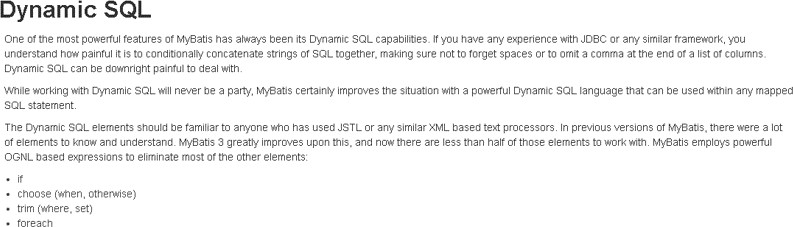
**4.2 映射配置⽂件mapper.xml**

**动态sql语句**

**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：



**动 态 SQL 之**

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>

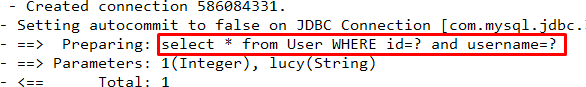
<if test="username!=null"> and username=#{username}

</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);

… … …

当查询条件只有id存在时，控制台打印的sql语句如下：

… … …

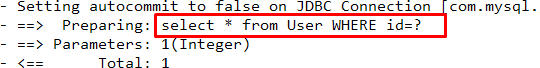
//获得MyBatis框架⽣成的UserMapper接⼝的实现类

UserMapper userMapper = sqlSession.getMapper(UserMapper.class); User condition = new User();

condition.setId(1);

User user = userMapper.findByCondition(condition);

… … …



**动 态 SQL 之**

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="list" open="id in(" close=")" item="id" separator=",">

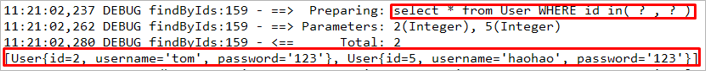
#{id}

</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);

… … …

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

**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>

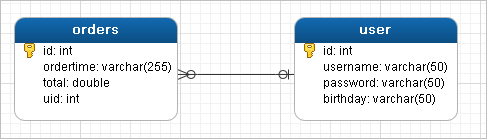
</select>

**第五部分：Mybatis复杂映射开发**

* 1. **⼀对⼀查询**
     1. **⼀对⼀查询的模型**

⽤户表和订单表的关系为，⼀个⽤户有多个订单，⼀个订单只从属于⼀个⽤户

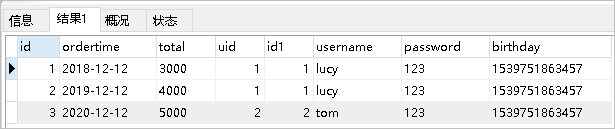
⼀对⼀查询的需求：查询⼀个订单，与此同时查询出该订单所属的⽤户



* + 1. **⼀对⼀查询的语句**

对应的sql语句：select \* from orders o,user u where o.uid=u.id;

查询的结果如下：



* + 1. **创建Order和User实体**

public class Order {

private int id;

private Date ordertime; private double total;

//代表当前订单从属于哪⼀个客户private User user;

}

public class User {

private int id;

private String username; private String password; private Date birthday;

}

* + 1. **创建OrderMapper接⼝**

public interface OrderMapper { List<Order> findAll();

}

* + 1. **配置OrderMapper.xml**

<mapper namespace="com.lagou.mapper.OrderMapper">

<resultMap id="orderMap" type="com.lagou.domain.Order">

<result column="uid" property="user.id"></result>

<result column="username" property="user.username"></result>

<result column="password" property="user.password"></result>

<result column="birthday" property="user.birthday"></result>

</resultMap>

<select id="findAll" resultMap="orderMap">

select \* from orders o,user u where o.uid=u.id

</select>

</mapper>

其中还可以配置如下：

<resultMap id="orderMap" type="com.lagou.domain.Order">

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

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

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

<association property="user" javaType="com.lagou.domain.User">

<result column="uid" property="id"></result>

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

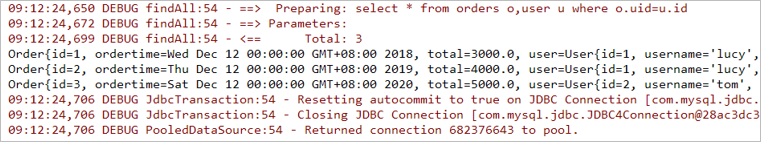
<result column="password" property="password"></result>

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

</association>

</resultMap>

* + 1. **测试结果**



OrderMapper mapper = sqlSession.getMapper(OrderMapper.class); List<Order> all = mapper.findAll();

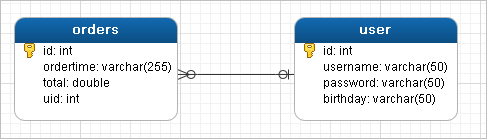
for(Order order : all){ System.out.println(order);

}

* 1. **⼀对多查询**
     1. **⼀对多查询的模型**

⽤户表和订单表的关系为，⼀个⽤户有多个订单，⼀个订单只从属于⼀个⽤户

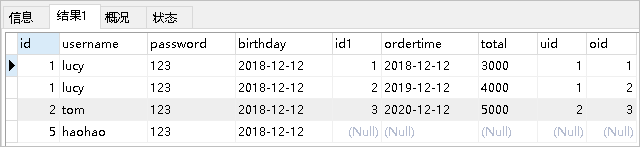
⼀对多查询的需求：查询⼀个⽤户，与此同时查询出该⽤户具有的订单



* + 1. **⼀对多查询的语句**

对应的sql语句：select \*,o.id oid from user u left join orders o on u.id=o.uid;

查询的结果如下：



* + 1. **修改User实体**

public class Order {

private int id;

private Date ordertime; private double total;

//代表当前订单从属于哪⼀个客户private User user;

}

public class User {

private int id;

private String username; private String password; private Date birthday;

//代表当前⽤户具备哪些订单

private List<Order> orderList;

}

* + 1. **创建UserMapper接⼝**

public interface UserMapper { List<User> findAll();

}

* + 1. **配置UserMapper.xml**

<mapper namespace="com.lagou.mapper.UserMapper">

<resultMap id="userMap" type="com.lagou.domain.User">

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

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

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

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

<collection property="orderList" ofType="com.lagou.domain.Order">

<result column="oid" property="id"></result>

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

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

</collection>

</resultMap>

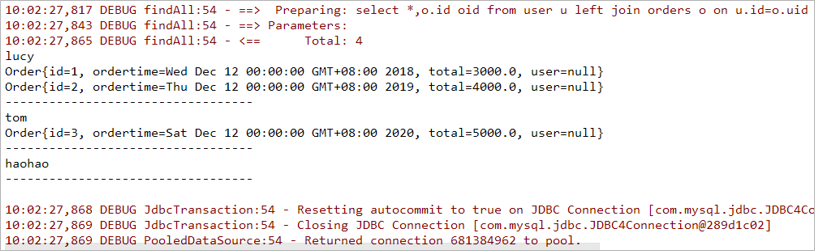
<select id="findAll" resultMap="userMap">

select \*,o.id oid from user u left join orders o on u.id=o.uid

</select>

</mapper>

* + 1. **测试结果**



UserMapper mapper = sqlSession.getMapper(UserMapper.class); List<User> all = mapper.findAll();

for(User user : all){ System.out.println(user.getUsername()); List<Order> orderList = user.getOrderList(); for(Order order : orderList){

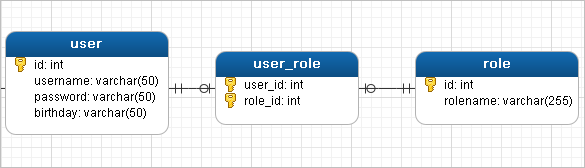
System.out.println(order);

}

System.out.println(" ");

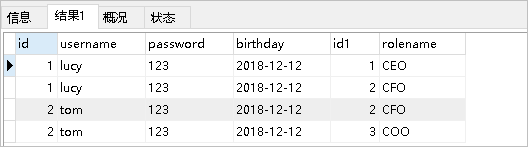
}

* 1. **多对多查询**
     1. **多对多查询的模型**

⽤户表和⻆⾊表的关系为，⼀个⽤户有多个⻆⾊，⼀个⻆⾊被多个⽤户使⽤ 多对多查询的需求：查询⽤户同时查询出该⽤户的所有⻆⾊

* + 1. **多对多查询的语句**

对应的sql语句：select u.*,r.*,r.id rid from user u left join user\_role ur on u.id=ur.user\_id inner join role r on ur.role\_id=r.id;

查询的结果如下：

* + 1. **创建Role实体，修改User实体**

public class User { private int id;

private String username; private String password; private Date birthday;

//代表当前⽤户具备哪些订单

private List<Order> orderList;

//代表当前⽤户具备哪些⻆⾊private List<Role> roleList;

}

public class Role {

private int id;

private String rolename;

}

* + 1. **添加UserMapper接⼝⽅法**

List<User> findAllUserAndRole();

* + 1. **配置UserMapper.xml**

<resultMap id="userRoleMap" type="com.lagou.domain.User">

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

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

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

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

<collection property="roleList" ofType="com.lagou.domain.Role">

<result column="rid" property="id"></result>

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

</collection>

</resultMap>

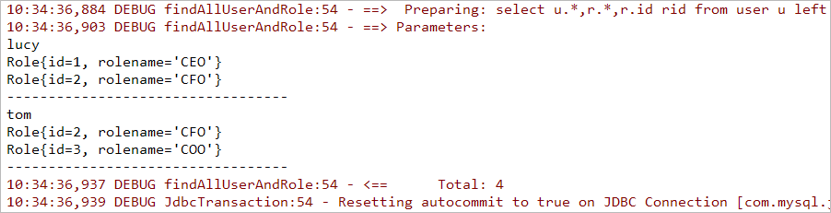
<select id="findAllUserAndRole" resultMap="userRoleMap">

select u.\*,r.\*,r.id rid from user u left join user\_role ur on u.id=ur.user\_id

inner join role r on ur.role\_id=r.id

</select>

* + 1. **测试结果**



UserMapper mapper = sqlSession.getMapper(UserMapper.class); List<User> all = mapper.findAllUserAndRole();

for(User user : all){ System.out.println(user.getUsername()); List<Role> roleList = user.getRoleList(); for(Role role : roleList){

System.out.println(role);

}

System.out.println(" ");

}

**5.4 知识⼩结**

MyBatis多表配置⽅式：

**⼀对⼀配置：使⽤做配置**

**⼀对多配置：使⽤+做配置多对多配置：使⽤+做配置**

**第六部分：Mybatis注解开发**

* 1. **MyBatis的常⽤注解**

这⼏年来注解开发越来越流⾏，Mybatis也可以使⽤注解开发⽅式，这样我们就可以减少编写Mapper 映射⽂件了。我们先围绕⼀些基本的CRUD来学习，再学习复杂映射多表操作。

@Insert：实现新增@Update：实现更新@Delete：实现删除@Select：实现查询@Result：实现结果集封装

@Results：可以与@Result ⼀起使⽤，封装多个结果集@One： 实 现 ⼀ 对 ⼀ 结 果 集 封 装 @Many：实现⼀对多结果集封装

* 1. **MyBatis的增删改查**

我们完成简单的user表的增删改查的操作

private UserMapper userMapper;

@Before

public void before() throws IOException { InputStream resourceAsStream =

Resources.getResourceAsStream("SqlMapConfig.xml"); SqlSessionFactory sqlSessionFactory = new

SqlSessionFactoryBuilder().build(resourceAsStream); SqlSession sqlSession = sqlSessionFactory.openSession(true); userMapper = sqlSession.getMapper(UserMapper.class);

}

@Test

public void testAdd() { User user = new User();

user.setUsername("测试数据");

user.setPassword("123"); user.setBirthday(new Date()); userMapper.add(user);

}

@Test

public void testUpdate() throws IOException { User user = new User();

user.setId(16);

user.setUsername("测试数据修改"); user.setPassword("abc"); user.setBirthday(new Date()); userMapper.update(user);

}

@Test

public void testDelete() throws IOException { userMapper.delete(16);

}

@Test

public void testFindById() throws IOException { User user = userMapper.findById(1); System.out.println(user);

}

@Test

public void testFindAll() throws IOException { List<User> all = userMapper.findAll(); for(User user : all){

System.out.println(user);

}

}

修改MyBatis的核⼼配置⽂件，我们使⽤了注解替代的映射⽂件，所以我们只需要加载使⽤了注解的

Mapper接⼝即可

<mappers>

<!--扫描使⽤注解的类-->

<mapper class="com.lagou.mapper.UserMapper"></mapper>

</mappers>

或者指定扫描包含映射关系的接⼝所在的包也可以

<mappers>

<!--扫描使⽤注解的类所在的包-->

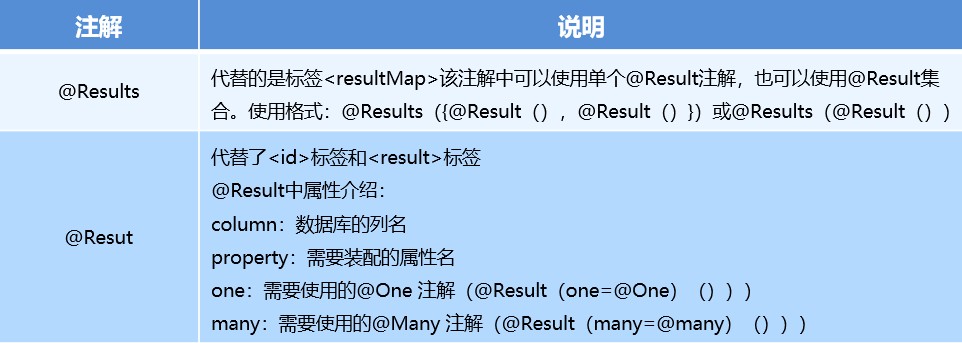
<package name="com.lagou.mapper"></package>

</mappers>

* 1. **MyBatis的注解实现复杂映射开发**

实现复杂关系映射之前我们可以在映射⽂件中通过配置来实现，使⽤注解开发后，我们可以使⽤

@Results注解，@Result注解，@One注解，@Many注解组合完成复杂关系的配置

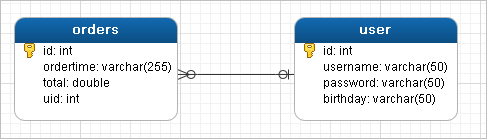




* 1. **⼀对⼀查询**
     1. **⼀对⼀查询的模型**

⽤户表和订单表的关系为，⼀个⽤户有多个订单，⼀个订单只从属于⼀个⽤户

⼀对⼀查询的需求：查询⼀个订单，与此同时查询出该订单所属的⽤户



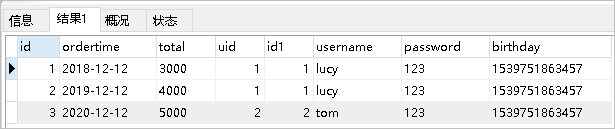
* + 1. **⼀对⼀查询的语句**

对应的sql语句：

select \* from orders;

select \* from user where id=查询出订单的uid;

查询的结果如下：



* + 1. **创建Order和User实体**

public class Order {

private int id;

private Date ordertime; private double total;

//代表当前订单从属于哪⼀个客户private User user;

}

public class User {

private int id;

private String username; private String password; private Date birthday;

}

* + 1. **创建OrderMapper接⼝**

public interface OrderMapper { List<Order> findAll();

}

* + 1. **使⽤注解配置Mapper**

public interface OrderMapper { @Select("select \* from orders") @Results({

@Result(id=true,property = "id",column = "id"), @Result(property = "ordertime",column = "ordertime"), @Result(property = "total",column = "total"), @Result(property = "user",column = "uid",

javaType = User.class, one = @One(select =

"com.lagou.mapper.UserMapper.findById"))

})

List<Order> findAll();

}

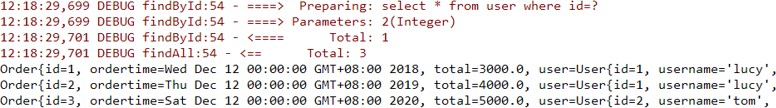
public interface UserMapper {

@Select("select \* from user where id=#{id}")

User findById(int id);

}

* + 1. **测试结果**



@Test

public void testSelectOrderAndUser() { List<Order> all = orderMapper.findAll(); for(Order order : all){

System.out.println(order);

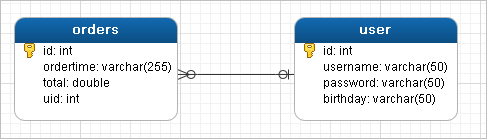
}

}

* 1. **⼀对多查询**
     1. **⼀对多查询的模型**

⽤户表和订单表的关系为，⼀个⽤户有多个订单，⼀个订单只从属于⼀个⽤户

⼀对多查询的需求：查询⼀个⽤户，与此同时查询出该⽤户具有的订单



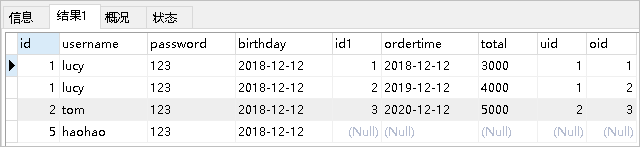
* + 1. **⼀对多查询的语句**

对应的sql语句：

select \* from user;

select \* from orders where uid=查询出⽤户的id;

查询的结果如下：



* + 1. **修改User实体**

public class Order {

private int id;

private Date ordertime; private double total;

//代表当前订单从属于哪⼀个客户private User user;

}

public class User {

private int id;

private String username; private String password; private Date birthday;

//代表当前⽤户具备哪些订单

private List<Order> orderList;

}

* + 1. **创建UserMapper接⼝**

List<User> findAllUserAndOrder();

* + 1. **使⽤注解配置Mapper**

public interface UserMapper { @Select("select \* from user") @Results({

@Result(id = true,property = "id",column = "id"), @Result(property = "username",column = "username"), @Result(property = "password",column = "password"), @Result(property = "birthday",column = "birthday"), @Result(property = "orderList",column = "id",

javaType = List.class, many = @Many(select =

"com.lagou.mapper.OrderMapper.findByUid"))

})

List<User> findAllUserAndOrder();

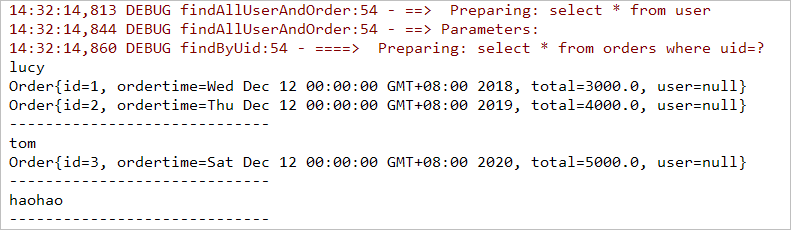
}

public interface OrderMapper {

@Select("select \* from orders where uid=#{uid}") List<Order> findByUid(int uid);

}

* + 1. **测试结果**



List<User> all = userMapper.findAllUserAndOrder(); for(User user : all){

System.out.println(user.getUsername()); List<Order> orderList = user.getOrderList(); for(Order order : orderList){

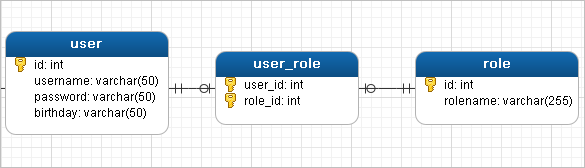
System.out.println(order);

}

System.out.println(" ");

}

* 1. **多对多查询**
     1. **多对多查询的模型**

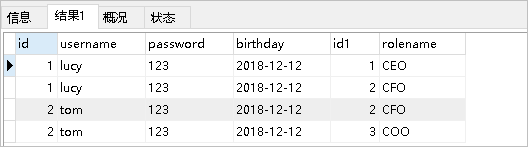
⽤户表和⻆⾊表的关系为，⼀个⽤户有多个⻆⾊，⼀个⻆⾊被多个⽤户使⽤ 多对多查询的需求：查询⽤户同时查询出该⽤户的所有⻆⾊

* + 1. **多对多查询的语句**

对应的sql语句：

select \* from user;

select \* from role r,user\_role ur where r.id=ur.role\_id and ur.user\_id=⽤户的id

查询的结果如下：

* + 1. **创建Role实体，修改User实体**

public class User { private int id;

private String username; private String password; private Date birthday;

//代表当前⽤户具备哪些订单

private List<Order> orderList;

//代表当前⽤户具备哪些⻆⾊private List<Role> roleList;

}

public class Role {

private int id;

private String rolename;

}

* + 1. **添加UserMapper接⼝⽅法**

List<User> findAllUserAndRole();

* + 1. **使⽤注解配置Mapper**

public interface UserMapper { @Select("select \* from user") @Results({

@Result(id = true,property = "id",column = "id"), @Result(property = "username",column = "username"), @Result(property = "password",column = "password"), @Result(property = "birthday",column = "birthday"), @Result(property = "roleList",column = "id",

javaType = List.class, many = @Many(select =

"com.lagou.mapper.RoleMapper.findByUid"))

})

List<User> findAllUserAndRole();}

public interface RoleMapper {

@Select("select \* from role r,user\_role ur where r.id=ur.role\_id and ur.user\_id=#{uid}")

List<Role> findByUid(int uid);

}

* + 1. **测试结果**

UserMapper mapper = sqlSession.getMapper(UserMapper.class); List<User> all = mapper.findAllUserAndRole();

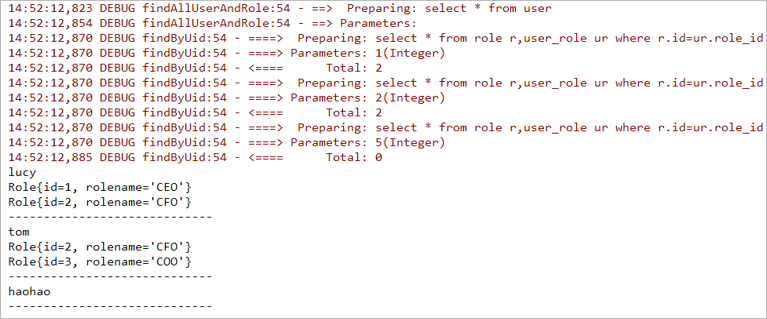
for(User user : all){ System.out.println(user.getUsername()); List<Role> roleList = user.getRoleList(); for(Role role : roleList){

System.out.println(role);

}

System.out.println(" ");

}



**第七部分：Mybatis缓存**

* 1. **⼀级缓存**

①、在⼀个sqlSession中，对User表根据id进⾏两次查询，查看他们发出sql语句的情况

@Test

public void test1(){

//根据 sqlSessionFactory 产⽣ session

SqlSession sqlSession = sessionFactory.openSession(); UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

//第⼀次查询，发出sql语句，并将查询出来的结果放进缓存中

User u1 = userMapper.selectUserByUserId(1); System.out.println(u1);

//第⼆次查询，由于是同⼀个sqlSession,会在缓存中查询结果

//如果有，则直接从缓存中取出来，不和数据库进⾏交互

User u2 = userMapper.selectUserByUserId(1); System.out.println(u2);

sqlSession.close();

}

查看控制台打印情况：



② 、同样是对user表进⾏两次查询，只不过两次查询之间进⾏了⼀次update操作。

@Test

public void test2(){

//根据 sqlSessionFactory 产⽣ session

SqlSession sqlSession = sessionFactory.openSession(); UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

//第⼀次查询，发出sql语句，并将查询的结果放⼊缓存中

User u1 = userMapper.selectUserByUserId( 1 ); System.out.println(u1);

//第⼆步进⾏了⼀次更新操作，sqlSession.commit()

u1.setSex("⼥");

userMapper.updateUserByUserId(u1); sqlSession.commit();

//第⼆次查询，由于是同⼀个sqlSession.commit(),会清空缓存信息

//则此次查询也会发出sql语句

User u2 = userMapper.selectUserByUserId(1); System.out.println(u2);

sqlSession.close();

}

查看控制台打印情况：

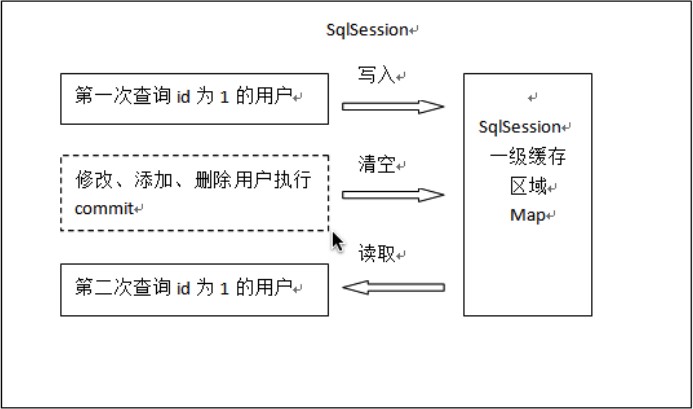


③、**总结**

1、第⼀次发起查询⽤户id为1的⽤户信息，先去找缓存中是否有id为1的⽤户信息，如果没有，从 数据库查询⽤户信息。得到⽤户信息，将⽤户信息存储到⼀级缓存中。

2、 如果中间sqlSession去执⾏commit操作（执⾏插⼊、更新、删除），则会清空SqlSession中的 ⼀级缓存，这样做的⽬的为了让缓存中存储的是最新的信息，避免脏读。

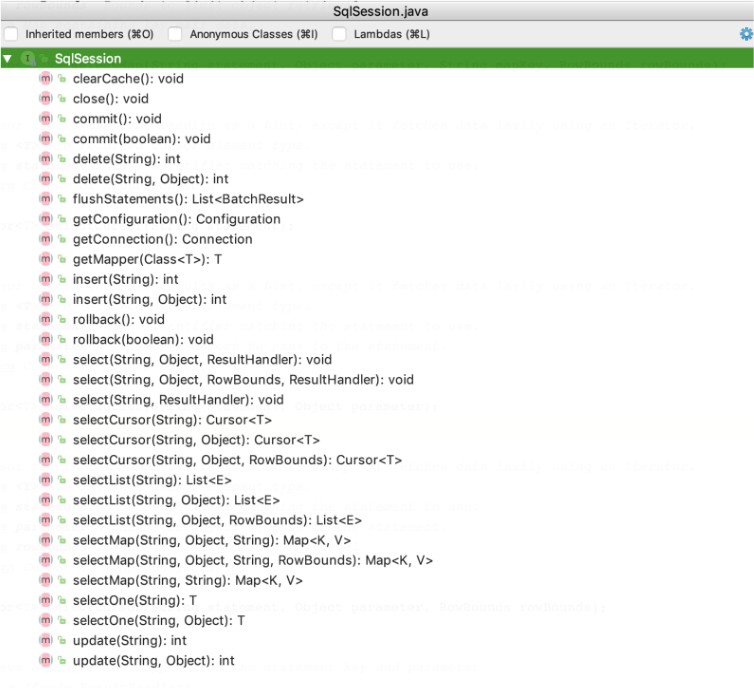
3、 第⼆次发起查询⽤户id为1的⽤户信息，先去找缓存中是否有id为1的⽤户信息，缓存中有，直 接从缓存中获取⽤户信息



**⼀级缓存原理探究与源码分析**

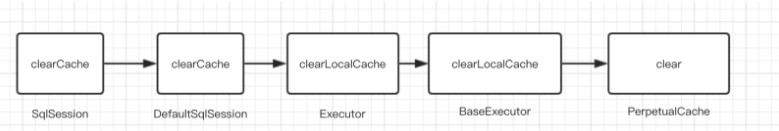
⼀级缓存到底是什么？⼀级缓存什么时候被创建、⼀级缓存的⼯作流程是怎样的？相信你现在应该会有 这⼏个疑问，那么我们本节就来研究⼀下⼀级缓存的本质

⼤家可以这样想，上⾯我们⼀直提到⼀级缓存，那么提到⼀级缓存就绕不开SqlSession,所以索性我们就直接从SqlSession，看看有没有创建缓存或者与缓存有关的属性或者⽅法



调研了⼀圈，发现上述所有⽅法中，好像只有clearCache()和缓存沾点关系，那么就直接从这个⽅ 法⼊

⼿吧，分析源码时**，我们要看它(此类)是谁，它的⽗类和⼦类分别⼜是谁**，对如上关系了解了，你才 会对这个类有更深的认识，分析了⼀圈，你可能会得到如下这个流程图



再深⼊分析，流程⾛到**Perpetualcache**中的clear()⽅法之后，会调⽤其**cache.clear()**⽅法，那 么这个

cache是什么东⻄呢？点进去发现，cache其实就是private Map cache = new

HashMap()；也就是⼀个Map，所以说cache.clear()其实就是map.clear()，也就是说，缓存其实就是 本地存放的⼀个map对象，每⼀个SqISession都会存放⼀个map对象的引⽤，那么这个cache是何 时创建的呢？

你觉得最有可能创建缓存的地⽅是哪⾥呢？我觉得是**Executor**，为什么这么认为？因为Executor是 执

⾏器，⽤来执⾏SQL请求，⽽且清除缓存的⽅法也在Executor中执⾏，所以很可能缓存的创建也很 有可能在Executor中，看了⼀圈发现Executor中有⼀个createCacheKey⽅法，这个⽅法很像是创 建缓存的

⽅法啊，跟进去看看，你发现createCacheKey⽅法是由BaseExecutor执⾏的，代码如下

CacheKey cacheKey = new CacheKey();

//MappedStatement 的 id

// id就是Sql语句的所在位置包名+类名+ SQL名称

cacheKey.update(ms.getId());

// offset 就 是 0 cacheKey.update(rowBounds.getOffset());

// limit 就 是 Integer.MAXVALUE cacheKey.update(rowBounds.getLimit());

//具体的SQL语句cacheKey.update(boundSql.getSql());

//后⾯是update 了 sql中带的参数cacheKey.update(value);

...

if (configuration.getEnvironment() != null) {

// issue #176 cacheKey.update(configuration.getEnvironment().getId());

}

创建缓存key会经过⼀系列的update⽅法，udate⽅法由⼀个CacheKey这个对象来执⾏的，这个

update⽅法最终由updateList的list来把五个值存进去，对照上⾯的代码和下⾯的图示，你应该能 理解这五个值都是什么了



这⾥需要注意⼀下最后⼀个值，configuration.getEnvironment().getId()这是什么，这其实就是 定义在

mybatis-config.xml中的标签，⻅如下。

<environments default="development">

<environment id="development">

<transactionManager type="JDBC"/>

<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>

那么我们回归正题，那么创建完缓存之后该⽤在何处呢？总不会凭空创建⼀个缓存不使⽤吧？绝对不会 的，经过我们对⼀级缓存的探究之后，我们发现⼀级缓存更多是⽤**于查询操作，毕竟⼀级缓存也叫做查 询缓存吧，为什么叫查询缓存我们⼀会⼉说**。我们先来看⼀下这个缓存到底⽤在哪了，我们跟踪到

query⽅法如下：

Override

public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException {

BoundSql boundSql = ms.getBoundSql(parameter);

//创建缓存

CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql); return query(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

@SuppressWarnings("unchecked")

Override

public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

...

list = resultHandler == null ? (List<E>) localCache.getObject(key) : null; if (list != null) {

//这个主要是处理存储过程⽤的。

handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);

} else {

list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

...

}

// queryFromDatabase ⽅ 法

private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

List<E> list;

localCache.putObject(key, EXECUTION\_PLACEHOLDER); try {

list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);

} finally { localCache.removeObject(key);

}

localCache.putObject(key, list);

if (ms.getStatementType() == StatementType.CALLABLE) { localOutputParameterCache.putObject(key, parameter);

}

return list;

}

如果查不到的话，就从数据库查，在queryFromDatabase中，会对localcache进⾏写⼊。 localcache

对象的put⽅法最终交给Map进⾏存放

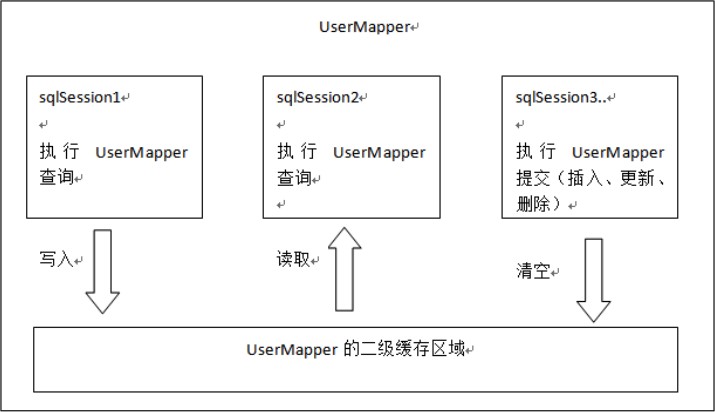
private Map<Object, Object> cache = new HashMap<Object, Object>(); @Override

public void putObject(Object key, Object value) { cache.put(key, value);

}

* 1. **⼆级缓存**

⼆级缓存的原理和⼀级缓存原理⼀样，第⼀次查询，会将数据放⼊缓存中，然后第⼆次查询则会直接去 缓存中取。但是⼀级缓存是基于sqlSession的，⽽⼆级缓存是基于mapper⽂件的namespace的，也 就是说多个sqlSession可以共享⼀个mapper中的⼆级缓存区域，并且如果两个mapper的namespace 相同，即使是两个mapper,那么这两个mapper中执⾏sql查询到的数据也将存在相同的⼆级缓存区域 中



如何使⽤⼆级缓存

**①** 、**开启⼆级缓存**

和⼀级缓存默认开启不⼀样，⼆级缓存需要我们⼿动开启

⾸先在全局配置⽂件sqlMapConfig.xml⽂件中加⼊如下代码:

<!--开启⼆级缓存-->

<settings>

<setting name="cacheEnabled" value="true"/>

</settings>

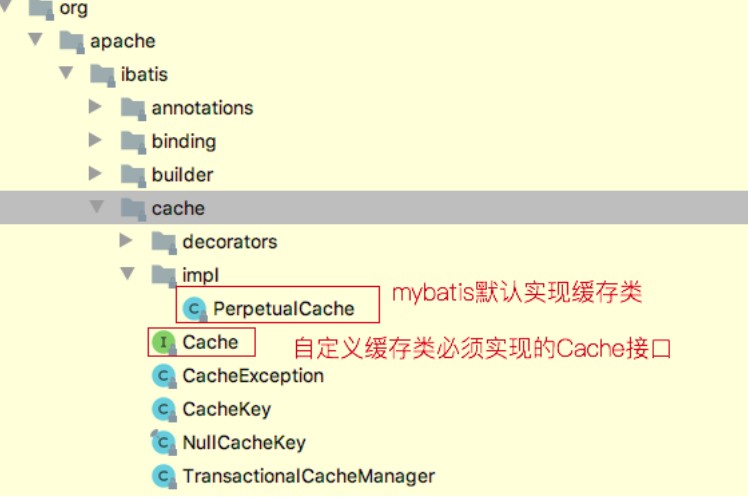
其次在UserMapper.xml⽂件中开启缓存

<!--开启⼆级缓存-->

<cache></cache>

我们可以看到mapper.xml⽂件中就这么⼀个空标签，其实这⾥可以配置,PerpetualCache这个类是

mybatis默认实现缓存功能的类。我们不写type就使⽤mybatis默认的缓存，也可以去实现Cache接⼝ 来⾃定义缓存。



public class PerpetualCache implements Cache { private final String id;

private MapcObject, Object> cache = new HashMapC);

public PerpetualCache(St ring id) { this.id = id;

}

我们可以看到⼆级缓存底层还是HashMap结构

public class User implements Serializable(

// ⽤ 户 ID private int id;

//⽤户姓名

private String username;

//⽤户性别

private String sex;

}

开启了⼆级缓存后，还需要将要缓存的pojo实现Serializable接⼝，为了将缓存数据取出执⾏反序列化操作，因为⼆级缓存数据存储介质多种多样，不⼀定只存在内存中，有可能存在硬盘中，如果我们要再取 这个缓存的话，就需要反序列化了。所以mybatis中的pojo都去实现Serializable接⼝

**③、测试**

⼀、测试⼆级缓存和sqlSession⽆关

@Test

public void testTwoCache(){

//根据 sqlSessionFactory 产⽣ session

SqlSession sqlSession1 = sessionFactory.openSession();

SqlSession sqlSession2 = sessionFactory.openSession();

UserMapper userMapper1 = sqlSession1.getMapper(UserMapper. class ); UserMapper userMapper2 = sqlSession2.getMapper(UserMapper. class );

//第⼀次查询，发出sql语句，并将查询的结果放⼊缓存中

User u1 = userMapper1.selectUserByUserId(1); System.out.println(u1);

sqlSession1.close(); //第⼀次查询完后关闭 sqlSession

//第⼆次查询，即使sqlSession1已经关闭了，这次查询依然不发出sql语句User u2 = userMapper2.selectUserByUserId(1); System.out.println(u2);

sqlSession2.close();

可以看出上⾯两个不同的sqlSession,第⼀个关闭了，第⼆次查询依然不发出sql查询语句

⼆、测试执⾏commit()操作，⼆级缓存数据清空

@Test

public void testTwoCache(){

//根据 sqlSessionFactory 产⽣ session

SqlSession sqlSession1 = sessionFactory.openSession(); SqlSession sqlSession2 = sessionFactory.openSession(); SqlSession sqlSession3 = sessionFactory.openSession();

String statement = "com.lagou.pojo.UserMapper.selectUserByUserld" ; UserMapper userMapper1 = sqlSession1.getMapper(UserMapper. class ); UserMapper userMapper2 = sqlSession2.getMapper(UserMapper. class ); UserMapper userMapper3 = sqlSession2.getMapper(UserMapper. class );

//第⼀次查询，发出sql语句，并将查询的结果放⼊缓存中

User u1 = userMapperl.selectUserByUserId( 1 ); System.out.println(u1);

sqlSessionl .close(); //第⼀次查询完后关闭sqlSession

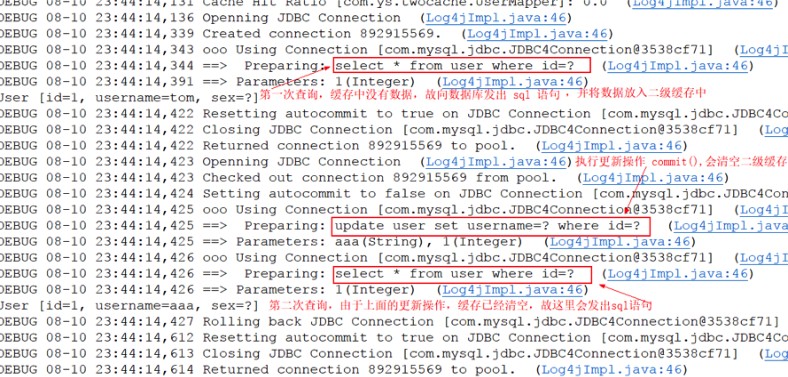
// 执 ⾏ 更 新 操 作 ，commit() u1.setUsername( "aaa" ); userMapper3.updateUserByUserId(u1); sqlSession3.commit();

//第⼆次查询，由于上次更新操作，缓存数据已经清空(防⽌数据脏读)，这⾥必须再次发出sql语User u2 = userMapper2.selectUserByUserId( 1 );

System.out.println(u2); sqlSession2.close();

}

查看控制台情况：



**④、useCache和flushCache**

mybatis中还可以配置userCache和flushCache等配置项，userCache是⽤来设置是否禁⽤⼆级缓 存的，在statement中设置useCache=false可以禁⽤当前select语句的⼆级缓存，即每次查询都会发出 sql 去查询，默认情况是true,即该sql使⽤⼆级缓存

<select id="selectUserByUserId" useCache="false" resultType="com.lagou.pojo.User" parameterType="int">

select \* from user where id=#{id}

</select>

这种情况是针对每次查询都需要最新的数据sql,要设置成useCache=false，禁⽤⼆级缓存，直接从数 据库中获取。

在mapper的同⼀个namespace中，如果有其它insert、update, delete操作数据后需要刷新缓 存，如果不执⾏刷新缓存会出现脏读。

设置statement配置中的flushCache="true”属性，默认情况下为true,即刷新缓存，如果改成false则 不会刷新。使⽤缓存时如果⼿动修改数据库表中的查询数据会出现脏读。

<select id="selectUserByUserId" flushCache="true" useCache="false" resultType="com.lagou.pojo.User" parameterType="int">

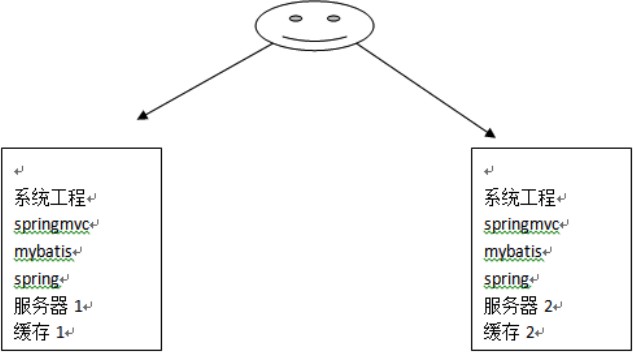
select \* from user where id=#{id}

</select>

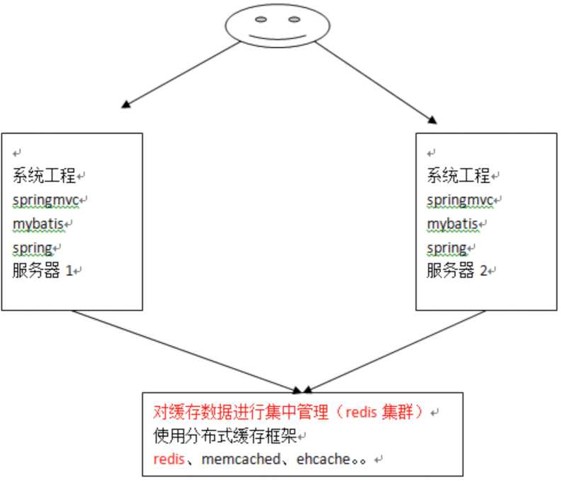
⼀般下执⾏完commit操作都需要刷新缓存，flushCache=true表示刷新缓存，这样可以避免数据库脏读。所以我们不⽤设置，默认即可

* 1. **⼆级缓存整合redis**

上⾯我们介绍了 mybatis⾃带的⼆级缓存，但是这个缓存是单服务器⼯作，⽆法实现分布式缓存。 那么什么是分布式缓存呢？假设现在有两个服务器1和2,⽤户访问的时候访问了 1服务器，查询后的缓 存就会放在1服务器上，假设现在有个⽤户访问的是2服务器，那么他在2服务器上就⽆法获取刚刚那个 缓存，如下图所示：



为了解决这个问题，就得找⼀个分布式的缓存，专⻔⽤来存储缓存数据的，这样不同的服务器要缓存数 据都往它那⾥存，取缓存数据也从它那⾥取，如下图所示：



如上图所示，在⼏个不同的服务器之间，我们使⽤第三⽅缓存框架，将缓存都放在这个第三⽅框架中, 然后⽆论有多少台服务器，我们都能从缓存中获取数据。

这⾥我们介绍**mybatis**与**redis**的整合。

刚刚提到过，mybatis提供了⼀个eache接⼝，如果要实现⾃⼰的缓存逻辑，实现cache接⼝开发即可。

mybati s本身默认实现了⼀个，但是这个缓存的实现⽆法实现分布式缓存，所以我们要⾃⼰来实现。

redis分布式缓存就可以，mybatis提供了⼀个针对cache接⼝的redis实现类，该类存在mybatis-redis包 中

实现：

* + 1. pom⽂件

<dependency>

<groupId>org.mybatis.caches</groupId>

<artifactId>mybatis-redis</artifactId>

<version>1.0.0-beta2</version>

</dependency>

* + 1. 配置⽂件Mapper.xml

<?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">](http://mybatis.org/dtd/mybatis-3-mapper.dtd)

<mapper namespace="com.lagou.mapper.IUserMapper">

<cache type="org.mybatis.caches.redis.RedisCache" />

<select id="findAll" resultType="com.lagou.pojo.User" useCache="true"> select \* from user

</select>

* + 1. redis.properties

redis.host=localhost redis.port=6379 redis.connectionTimeout=5000 redis.password= redis.database=0

* + 1. 测试

@Test

public void SecondLevelCache(){

SqlSession sqlSession1 = sqlSessionFactory.openSession();

SqlSession sqlSession2 = sqlSessionFactory.openSession(); SqlSession sqlSession3 = sqlSessionFactory.openSession(); IUserMapper mapper1 = sqlSession1.getMapper(IUserMapper.class); lUserMapper mapper2 = sqlSession2.getMapper(lUserMapper.class); lUserMapper mapper3 = sqlSession3.getMapper(IUserMapper.class); User user1 = mapper1.findUserById(1);

sqlSession1.close(); //清空⼀级缓存

User user = new User(); user.setId(1); user.setUsername("lisi"); mapper3.updateUser(user); sqlSession3.commit();

User user2 = mapper2.findUserById(1);

System.out.println(user1==user2);

}

**源码分析：**

RedisCache和⼤家普遍实现Mybatis的缓存⽅案⼤同⼩异，⽆⾮是实现Cache接⼝，并使⽤jedis操作缓 存；不过该项⽬在设计细节上有⼀些区别；

public final class RedisCache implements Cache { public RedisCache(final String id) {

if (id == null) {

throw new IllegalArgumentException("Cache instances require anID");

}

this.id = id;

RedisConfig redisConfig = RedisConfigurationBuilder.getInstance().parseConfiguration(); pool = new JedisPool(redisConfig, redisConfig.getHost(), redisConfig.getPort(),

redisConfig.getConnectionTimeout(), redisConfig.getSoTimeout(), redisConfig.getPassword(),

redisConfig.getDatabase(), redisConfig.getClientName());

}

RedisCache在mybatis启动的时候，由MyBatis的CacheBuilder创建，创建的⽅式很简单，就是调⽤

RedisCache的带有String参数的构造⽅法，即RedisCache(String id)；⽽在RedisCache的构造⽅法中， 调⽤了 RedisConfigu rationBuilder 来创建 RedisConfig 对象，并使⽤ RedisConfig 来创建JedisPool。

RedisConfig类继承了 JedisPoolConfig，并提供了 host,port等属性的包装，简单看⼀下RedisConfig的属性：

public class RedisConfig extends JedisPoolConfig { private String host = Protocol.DEFAULT\_HOST; private int port = Protocol.DEFAULT\_PORT;

private int connectionTimeout = Protocol.DEFAULT\_TIMEOUT; private int soTimeout = Protocol.DEFAULT\_TIMEOUT;

private String password;

private int database = Protocol.DEFAULT\_DATABASE; private String clientName;

RedisConfig对象是由RedisConfigurationBuilder创建的，简单看下这个类的主要⽅法：

public RedisConfig parseConfiguration(ClassLoader classLoader) { Properties config = new Properties();

InputStream input = classLoader.getResourceAsStream(redisPropertiesFilename);

if (input != null) { try {

config.load(input);

} catch (IOException e) { throw new RuntimeException(

"An error occurred while reading classpath property '"

+ redisPropertiesFilename

+ "', see nested exceptions", e);

} finally { try {

input.close();

} catch (IOException e) {

// close quietly

}

}

}

RedisConfig jedisConfig = new RedisConfig(); setConfigProperties(config, jedisConfig); return jedisConfig;

}

核⼼的⽅法就是parseConfiguration⽅法，该⽅法从classpath中读取⼀个redis.properties⽂件:

host=localhost port=6379 connectionTimeout=5000 soTimeout=5000

password= database=0 clientName=

并将该配置⽂件中的内容设置到RedisConfig对象中，并返回；接下来，就是RedisCache使⽤

RedisConfig类创建完成edisPool；在RedisCache中实现了⼀个简单的模板⽅法，⽤来操作Redis：

private Object execute(RedisCallback callback) { Jedis jedis = pool.getResource();

try {

return callback.doWithRedis(jedis);

} finally { jedis.close();

}

}

模板接⼝为RedisCallback，这个接⼝中就只需要实现了⼀个doWithRedis⽅法⽽已：

public interface RedisCallback { Object doWithRedis(Jedis jedis);

}

接下来看看Cache中最重要的两个⽅法：putObject和getObject，通过这两个⽅法来查看mybatis-redis 储存数据的格式：

@Override

public void putObject(final Object key, final Object value) { execute(new RedisCallback() {

@Override

public Object doWithRedis(Jedis jedis) { jedis.hset(id.toString().getBytes(), key.toString().getBytes(),

SerializeUtil.serialize(value)); return null;

}

});

}

@Override

public Object getObject(final Object key) { return execute(new RedisCallback() {

@Override

public Object doWithRedis(Jedis jedis) {

return SerializeUtil.unserialize(jedis.hget(id.toString().getBytes(), key.toString().getBytes()));

}

});

}

可以很清楚的看到，mybatis-redis在存储数据的时候，是使⽤的hash结构，把cache的id作为这个hash 的key (cache的id在mybatis中就是mapper的namespace)；这个mapper中的查询缓存数据作为 hash 的field,需要缓存的内容直接使⽤SerializeUtil存储，SerializeUtil和其他的序列化类差不多，负责 对象的序列化和反序列化；

**第⼋部分：Mybatis插件**

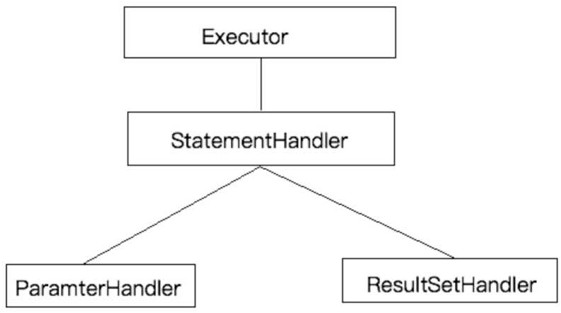
* 1. **插件简介**

⼀般情况下，开源框架都会提供插件或其他形式的拓展点，供开发者⾃⾏拓展。这样的好处是显⽽易⻅ 的，⼀是增加了框架的灵活性。⼆是开发者可以结合实际需求，对框架进⾏拓展，使其能够更好的⼯作。以MyBatis为例，我们可基于MyBati s插件机制实现分⻚、分表，监控等功能。由于插件和业务 ⽆关，业务也⽆法感知插件的存在。因此可以⽆感植⼊插件，在⽆形中增强功能

* 1. **Mybatis插件介绍**

Mybati s作为⼀个应⽤⼴泛的优秀的ORM开源框架，这个框架具有强⼤的灵活性，在四⼤组件

(Executor、StatementHandler、ParameterHandler、ResultSetHandler)处提供了简单易⽤的插 件扩展机制。Mybatis对持久层的操作就是借助于四⼤核⼼对象。MyBatis⽀持⽤插件对四⼤核⼼对象进 ⾏拦截，对mybatis来说插件就是拦截器，⽤来增强核⼼对象的功能，增强功能本质上是借助于底层的 动态代理实现的，换句话说，MyBatis中的四⼤对象都是代理对象



**MyBatis所允许拦截的⽅法如下：**

执⾏器Executor (update、query、commit、rollback等⽅法)；

SQL语法构建器StatementHandler (prepare、parameterize、batch、updates query等⽅ 法)； 参数处理器ParameterHandler (getParameterObject、setParameters⽅法)；

结果集处理器ResultSetHandler (handleResultSets、handleOutputParameters等⽅法)；

* 1. **Mybatis插件原理**

在四⼤对象创建的时候

1、每个创建出来的对象不是直接返回的，⽽是interceptorChain.pluginAll(parameterHandler);

2、获取到所有的Interceptor (拦截器)(插件需要实现的接⼝)；调⽤ interceptor.plugin(target);返回 target 包装后的对象

3、插件机制，我们可以使⽤插件为⽬标对象创建⼀个代理对象；AOP (⾯向切⾯)我们的插件可 以为四⼤对象创建出代理对象，代理对象就可以拦截到四⼤对象的每⼀个执⾏；

**拦截**

插件具体是如何拦截并附加额外的功能的呢？以ParameterHandler来说

public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object object, BoundSql sql, InterceptorChain interceptorChain){

ParameterHandler parameterHandler =

mappedStatement.getLang().createParameterHandler(mappedStatement,object,sql); parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHandler);

return parameterHandler;

}

public Object pluginAll(Object target) {

for (Interceptor interceptor : interceptors) { target = interceptor.plugin(target);

}

return target;

}

interceptorChain保存了所有的拦截器(interceptors)，是mybatis初始化的时候创建的。调⽤拦截器链 中的拦截器依次的对⽬标进⾏拦截或增强。interceptor.plugin(target)中的target就可以理解为mybatis 中的四⼤对象。返回的target是被重重代理后的对象

如果我们想要拦截Executor的query⽅法，那么可以这样定义插件：

@Intercepts({ @Signature(

type = Executor.class, method = "query", args=

{MappedStatement.class,Object.class,RowBounds.class,ResultHandler.class}

)

})

public class ExeunplePlugin implements Interceptor {

//省略逻辑

}

除此之外，我们还需将插件配置到sqlMapConfig.xm l中。

<plugins>

<plugin interceptor="com.lagou.plugin.ExamplePlugin">

</plugin>

</plugins>

这样MyBatis在启动时可以加载插件，并保存插件实例到相关对象(InterceptorChain，拦截器链) 中。待准备⼯作做完后，MyBatis处于就绪状态。我们在执⾏SQL时，需要先通过DefaultSqlSessionFactory 创建 SqlSession。Executor 实例会在创建 SqlSession 的过程中被创建， Executor实例创建完毕后，

MyBatis会通过JDK动态代理为实例⽣成代理类。这样，插件逻辑即可在 Executor相关⽅法被调⽤前执

⾏。

以上就是MyBatis插件机制的基本原理

* 1. **⾃定义插件**
     1. 插件接⼝

Mybatis 插件接⼝-Interceptor

* Intercept⽅法，插件的核⼼⽅法
* plugin⽅法，⽣成target的代理对象
* setProperties⽅法，传递插件所需参数
  + 1. ⾃定义插件

设计实现⼀个⾃定义插件

Intercepts ({//注意看这个⼤花括号，也就这说这⾥可以定义多个@Signature对多个地⽅拦截，都⽤

这个拦截器

@Signature (type = StatementHandler .class , //这是指拦截哪个接⼝method = "prepare"，//这个接⼝内的哪个⽅法名，不要拼错了

args = { Connection.class, Integer .class}),//// 这是拦截的⽅法的⼊参，按顺序写到这，不要多也不要少，如果⽅法重载，可是要通过⽅法名和⼊参来确定唯⼀的

})

public class MyPlugin implements Interceptor {

private final Logger logger = LoggerFactory.getLogger(this.getClass());

// //这⾥是每次执⾏操作的时候，都会进⾏这个拦截器的⽅法内

Override

public Object intercept(Invocation invocation) throws Throwable {

//增强逻辑

System.out.println("对⽅法进⾏了增强 ")；

return invocation.proceed(); //执⾏原⽅法

}

/\*\*

* //主要是为了把这个拦截器⽣成⼀个代理放到拦截器链中
* ^Description包装⽬标对象 为⽬标对象创建代理对象
* @Param target为要拦截的对象
* @Return代理对象

\*/ Override

public Object plugin(Object target) {

System.out.println("将要包装的⽬标对象："+target); return Plugin.wrap(target,this);

}

/\*\*获取配置⽂件的属性\*\*/

//插件初始化的时候调⽤，也只调⽤⼀次，插件配置的属性从这⾥设置进来

Override

public void setProperties(Properties properties) {

System.out.println("插件配置的初始化参数："+properties );

}

}

sqlMapConfig.xml

<plugins>

<plugin interceptor="com.lagou.plugin.MySqlPagingPlugin">

<!--配置参数-->

<property name="name" value="Bob"/>

</plugin>

</plugins>

mapper接⼝

public interface UserMapper { List<User> selectUser();

}

mapper.xml

<mapper namespace="com.lagou.mapper.UserMapper">

<select id="selectUser" resultType="com.lagou.pojo.User"> SELECT

id,username FROM

user

</select>

</mapper>

测试类

public class PluginTest { @Test

public void test() throws IOException { InputStream resourceAsStream =

Resources.getResourceAsStream("sqlMapConfig.xml"); SqlSessionFactory sqlSessionFactory = new

SqlSessionFactoryBuilder().build(resourceAsStream); SqlSession sqlSession = sqlSessionFactory.openSession(); UserMapper userMapper = sqlSession.getMapper(UserMapper.class); List<User> byPaging = userMapper.selectUser();

for (User user : byPaging) { System.out.println(user);

}

}

}

* 1. **源码分析**

执⾏插件逻辑

Plugin实现了 InvocationHandler接⼝，因此它的invoke⽅法会拦截所有的⽅法调⽤。invoke⽅法会 对所拦截的⽅法进⾏检测，以决定是否执⾏插件逻辑。该⽅法的逻辑如下：

// -Plugin

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

try {

/\*

\*获取被拦截⽅法列表，⽐如：

\* signatureMap.get(Executor.class), 可能返回 [query, update,

commit]

\*/

Set<Method> methods = signatureMap.get(method.getDeclaringClass());

//检测⽅法列表是否包含被拦截的⽅法

if (methods != null && methods.contains(method)) {

//执⾏插件逻辑

return interceptor.intercept(new Invocation(target, method,

args));

//执⾏被拦截的⽅法

return method.invoke(target, args);

} catch(Exception e){

}

}

invoke⽅法的代码⽐较少，逻辑不难理解。⾸先,invoke⽅法会检测被拦截⽅法是否配置在插件的

@Signature注解中，若是，则执⾏插件逻辑，否则执⾏被拦截⽅法。插件逻辑封装在intercept中，该

⽅法的参数类型为Invocationo Invocation主要⽤于存储⽬标类，⽅法以及⽅法参数列表。下⾯简单看

⼀下该类的定义

public class Invocation { private final Object target; private final Method method; private final Object[] args;

public Invocation(Object targetf Method method, Object[] args) { this.target = target;

this.method = method;

//省略部分代码

public Object proceed() throws InvocationTargetException, IllegalAccessException { //调⽤被拦截的⽅法

>> —

关于插件的执⾏逻辑就分析结束

* 1. **pageHelper分⻚插件**

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

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 测试分⻚数据获取

**①导⼊通⽤PageHelper坐标**

<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：分⻚助⼿的插件 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 测试分⻚代码实现**

@Test

public void testPageHelper() {

//设置分⻚参数PageHelper.startPage(1, 2);

List<User> select = userMapper2.select(null); for (User user : select) {

System.out.println(user);

}

}

}

**获得分⻚相关的其他参数**

//其他分⻚的数据

PageInfo<User> pageInfo = new PageInfo<User>(select);

System.out.println("总条数："+pageInfo.getTotal());

System.out.println("总⻚数："+pageInfo. getPages ());

System.out.println("当前⻚："+pageInfo. getPageNum());

System.out.println("每⻚显万⻓度："+pageInfo.getPageSize());

System.out.println("是否第⼀⻚："+pageInfo.isIsFirstPage());

System.out.println("是否最后⼀⻚："+pageInfo.isIsLastPage());

* 1. **通⽤ mapper**

**什么是通⽤Mapper**

通⽤Mapper就是为了解决单表增删改查，基于Mybatis的插件机制。开发⼈员不需要编写SQL,不需要在DAO中增加⽅法，只要写好实体类，就能⽀持相应的增删改查⽅法

**如何使⽤**

1. ⾸先在maven项⽬，在pom.xml中引⼊mapper的依赖

<dependency>

<groupId>tk.mybatis</groupId>

<artifactId>mapper</artifactId>

<version>3.1.2</version>

</dependency>

1. Mybatis配置⽂件中完成配置

<plugins>

<!--分⻚插件：如果有分⻚插件，要排在通⽤mapper之前-->

<plugin interceptor="com.github.pagehelper.PageHelper">

<property name="dialect" value="mysql"/>

</plugin>

<plugin interceptor="tk.mybatis.mapper.mapperhelper.MapperInterceptor">

<!-- 通⽤Mapper接⼝，多个通⽤接⼝⽤逗号隔开 -->

<property name="mappers" value="tk.mybatis.mapper.common.Mapper"/>

</plugin>

</plugins>

1. 实体类设置主键

@Table(name = "t\_user")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY) private Integer id;

private String username;

}

1. 定义通⽤mapper

import com.lagou.domain.User;

import tk.mybatis.mapper.common.Mapper;

public interface UserMapper extends Mapper<User> {

}

1. 测试

public class UserTest { @Test

public void test1() throws IOException {

Inputstream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");

SqlSessionFactory build = new SqlSessionFactoryBuilder().build(resourceAsStream);

SqlSession sqlSession = build.openSession();

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

User user = new User(); user.setId(4);

—个返回值

//(1)mapper基础接⼝

//select 接 ⼝

User user1 = userMapper.selectOne(user); //根据实体中的属性进⾏查询，只能有

List<User> users = userMapper.select(null); //查询全部结果userMapper.selectByPrimaryKey(1); //根据主键字段进⾏查询，⽅法参数必须包含完

整的主键属性，查询条件使⽤等号

userMapper.selectCount(user); //根据实体中的属性查询总数，查询条件使⽤等号

// insert 接 ⼝

int insert = userMapper.insert(user); //保存⼀个实体，null值也会保存，不会使

⽤数据库默认值

int i = userMapper.insertSelective(user); //保存实体，null的属性不会保存， 会使⽤数据库默认值

// update 接 ⼝

int i1 = userMapper.updateByPrimaryKey(user);//根据主键更新实体全部字段，

null值会被更新

// delete 接 ⼝

int delete = userMapper.delete(user); //根据实体属性作为条件进⾏删除，查询条件 使⽤等号

userMapper.deleteByPrimaryKey(1); //根据主键字段进⾏删除，⽅法参数必须包含完整的主键属性

//(2)example⽅法

Example example = new Example(User.class); example.createCriteria().andEqualTo("id", 1);

example.createCriteria().andLike("val", "1");

//⾃定义查询

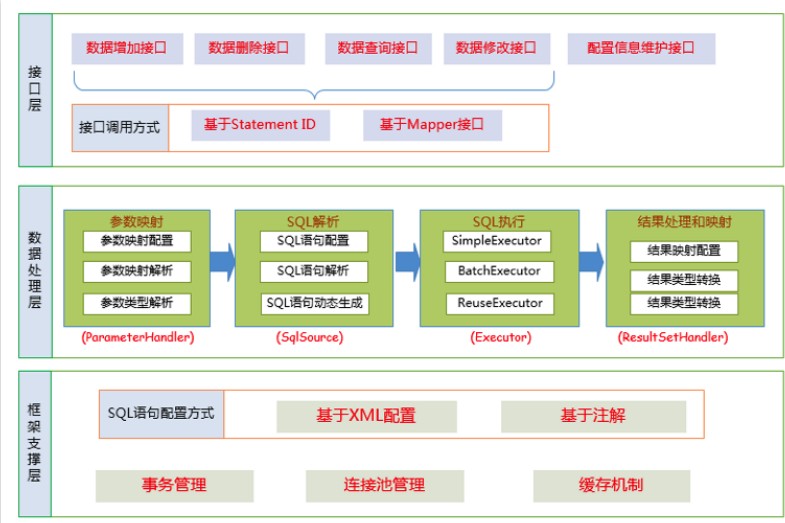
List<User> users1 = userMapper.selectByExample(example);

}

}

# 第九部分：Mybatis架构原理

## 架构设计



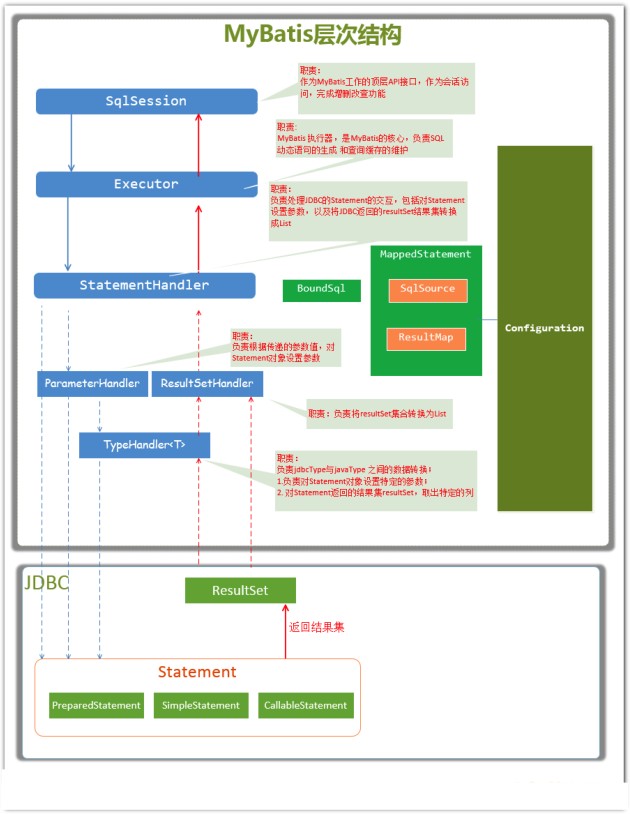
我们把Mybatis的功能架构分为三层：

1. API接⼝层：提供给外部使⽤的接⼝ API，开发⼈员通过这些本地API来操纵数据库。接⼝层⼀接收到 调⽤请求就会调⽤数据处理层来完成具体的数据处理。

MyBatis和数据库的交互有两种⽅式：

1. 使⽤传统的MyBati s提供的API ；
2. 使⽤Mapper代理的⽅式
3. 数据处理层：负责具体的SQL查找、SQL解析、SQL执⾏和执⾏结果映射处理等。它主要的⽬的是根据调⽤的请求完成⼀次数据库操作。
4. 基础⽀撑层：负责最基础的功能⽀撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是 共 ⽤的东⻄，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的⽀撑
   1. **主要构件及其相互关系**

|  |  |
| --- | --- |
| **构件** | **描述** |
| SqlSession | 作为MyBatis⼯作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能 |
| Executor | MyBatis执⾏器，是MyBatis调度的核⼼，负责SQL语句的⽣成和查询缓 存的维护 |
| StatementHandler | 封装了JDBC Statement操作，负责对JDBC statement的操作，如设置参数、将Statement结果集转换成List集合。 |
| ParameterHandler | 负责对⽤户传递的参数转换成JDBC Statement所需要的参数， |
| ResultSetHandler | 负责将JDBC返回的ResultSet结果集对象转换成List类型的集合； |
| TypeHandler | 负责java数据类型和jdbc数据类型之间的映射和转换 |
| MappedStatement | MappedStatement维护了⼀条＜select | update | delete | insert＞节点的封 装 |
| SqlSource | 负责根据⽤户传递的parameterObject，动态地⽣成SQL语句，将信息封 装到BoundSql对象中，并返回 |
| BoundSql | 表示动态⽣成的SQL语句以及相应的参数信息 |



* 1. **总体流程**

1. **加载配置并初始化 触发条件：**加载配置⽂件

配置来源于两个地⽅，⼀个是配置⽂件(主配置⽂件conf.xml,mapper⽂件\*.xml),—个是java代码中的 注解，将主配置⽂件内容解析封装到Configuration,将sql的配置信息加载成为⼀个mappedstatement 对象，存储在内存之中

1. **接收调⽤请求**

**触发条件**：调⽤Mybatis提供的API

**传⼊参数：为**SQL的ID和传⼊参数对象

**处理过程：**将请求传递给下层的请求处理层进⾏处理。

1. **处理操作请求**

**触发条件：**API接⼝层传递请求过来**传⼊参数：为**SQL的ID和传⼊参数对象**处理过程：**

* 1. 根据SQL的ID查找对应的MappedStatement对象。
  2. 根据传⼊参数对象解析MappedStatement对象，得到最终要执⾏的SQL和执⾏传⼊参数。
  3. 获取数据库连接，根据得到的最终SQL语句和执⾏传⼊参数到数据库执⾏，并得到执⾏结果。
  4. 根据MappedStatement对象中的结果映射配置对得到的执⾏结果进⾏转换处理，并得到最终的处理 结果。
  5. 释放连接资源。

1. **返回处理结果**

将最终的处理结果返回。

**第⼗部分：Mybatis源码剖析**

* 1. **传统⽅式源码剖析：**

**源码剖析-初始化**

Inputstream inputstream = Resources.getResourceAsStream("mybatis- config.xml");

//这⼀⾏代码正是初始化⼯作的开始。

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);

进⼊源码分析：

// 1.我们最初调⽤的build

public SqlSessionFactory build (InputStream inputStream){

//调⽤了重载⽅法

return build(inputStream, null, null);

}

// 2.调⽤的重载⽅法

public SqlSessionFactory build (InputStream inputStream, String environment,

Properties properties){ try {

// XMLConfigBuilder是专⻔解析mybatis的配置⽂件的类

XMLConfigBuilder parser = new XMLConfigBuilder(inputstream, environment, properties);

//这⾥⼜调⽤了⼀个重载⽅法。parser.parse()的返回值是Configuration对象

return build(parser.parse());

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error building SqlSession.", e)

}

MyBatis在初始化的时候，会将MyBatis的配置信息全部加载到内存中，使⽤org.apache.ibatis.session.Configuratio n 实例来维护

下⾯进⼊对配置⽂件解析部分：

⾸先对Configuration对象进⾏介绍：

Configuration对象的结构和xml配置⽂件的对象⼏乎相同。回顾⼀下xml中的配置标签有哪些：

properties (属性)，settings (设置)，typeAliases (类型别名)，typeHandlers (类型处理器)，objectFactory (对象⼯⼚)，mappers (映射器)等 Configuration也有对应的对象属性来封装它们

也就是说，初始化配置⽂件信息的本质就是创建Configuration对象，将解析的xml数据封装到

Configuration内部属性中

/\*\*

\* 解析 XML 成 Configuration 对象。

\*/

public Configuration parse () {

//若已解析，抛出BuilderException异常if (parsed) {

throw new BuilderException("Each XMLConfigBuilder can only be

used once.");

}

//标记已解析parsed = true;

// 解 析 XML configuration 节 点

parseConfiguration(parser.evalNode("/configuration"));

return configuration;

}

/\*\*

\*解析XML

\*/

private void parseConfiguration (XNode root){ try {

//issue #117 read properties first

// 解 析 <properties /> 标 签

propertiesElement(root.evalNode("properties"));

// 解析〈settings /> 标签

Properties settings =

settingsAsProperties(root.evalNode("settings"));

//加载⾃定义的VFS实现类loadCustomVfs(settings);

// 解 析 <typeAliases /> 标 签

typeAliasesElement(root.evalNode("typeAliases"));

//解析<plugins />标签pluginElement(root.evalNode("plugins"));

// 解 析 <objectFactory /> 标 签

objectFactoryElement(root.evalNode("objectFactory"));

// 解 析 <objectWrapperFactory /> 标 签

objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));

// 解 析 <reflectorFactory /> 标 签

reflectorFactoryElement(root.evalNode("reflectorFactory"));

// 赋 值 <settings /> ⾄ Configuration 属 性

settingsElement(settings);

// read it after objectFactory and objectWrapperFactory issue

#631

// 解析〈environments /> 标签

environmentsElement(root.evalNode("environments"));

// 解 析 <databaseIdProvider /> 标 签

databaseldProviderElement(root.evalNode("databaseldProvider"));

// 解 析 <typeHandlers /> 标 签

typeHandlerElement(root.evalNode("typeHandlers"));

//解析<mappers />标签mapperElement(root.evalNode("mappers"));

} catch (Exception e) {

throw new BuilderException("Error parsing SQL Mapper Configuration.Cause:" + e, e);

}

}

介绍⼀下 MappedStatement ：

作⽤：MappedStatement与Mapper配置⽂件中的⼀个select/update/insert/delete节点相对应。

mapper中配置的标签都被封装到了此对象中，主要⽤途是描述⼀条SQL语句。

**初始化过程：**回顾刚开 始介绍的加载配置⽂件的过程中，会对mybatis-config.xm l中的各个标签都进⾏解析，其中有mappers 标签⽤来引⼊mapper.xml⽂件或者配置mapper接⼝的⽬录。

<select id="getUser" resultType="user" > select \* from user where id=#{id}

</select>

样的⼀个select标签会在初始化配置⽂件时被解析封装成⼀个MappedStatement对象，然后存储在

Configuration对象的mappedStatements属性中，mappedStatements 是⼀个HashMap，存储时key

=全限定类名+⽅法名，value =对应的MappedStatement对象。

* 在configuration中对应的属性为

Map<String, MappedStatement> mappedStatements = new StrictMap<MappedStatement> ("Mapped Statements collection")

在 XMLConfigBuilder 中的处理：

private void parseConfiguration(XNode root) { try {

//省略其他标签的处理

mapperElement(root.evalNode("mappers"));

} catch (Exception e) {

throw new BuilderException("Error parsing SQL Mapper Configuration.

Cause:" + e, e);

}

}

到此对xml配置⽂件的解析就结束了，回到步骤2.中调⽤的重载build⽅法

// 5.调⽤的重载⽅法

public SqlSessionFactory build(Configuration config) {

//创建了 DefaultSqlSessionFactory 对象，传⼊ Configuration 对象。

return new DefaultSqlSessionFactory(config);

}

**源码剖析-执⾏SQL流程**

先简单介绍**SqlSession ：**

SqlSession是⼀个接⼝，它有两个实现类：DefaultSqlSession (默认)和SqlSessionManager (弃⽤，不做介绍)

SqlSession是MyBatis中⽤于和数据库交互的顶层类，通常将它与ThreadLocal绑定，⼀个会话使⽤⼀ 个SqlSession,并且在使⽤完毕后需要close

public class DefaultSqlSession implements SqlSession { private final Configuration configuration;

private final Executor executor; j

SqlSession中的两个最重要的参数，configuration与初始化时的相同，Executor为执⾏器

**Executor：**

Executor也是⼀个接⼝，他有三个常⽤的实现类： BatchExecutor ( 重 ⽤ 语 句 并 执 ⾏ 批 量 更 新 ) ReuseExecutor (重⽤预处理语句 prepared statements) SimpleExecutor (普通的执⾏器，默认)

继续分析，初始化完毕后，我们就要执⾏SQL 了

SqlSession sqlSession = factory.openSession(); List<User> list =

sqlSession.selectList("com.lagou.mapper.UserMapper.getUserByName");

获得 sqlSession

//6. 进⼊ o penSession ⽅法。

public SqlSession openSession() {

//getDefaultExecutorType()传递的是SimpleExecutor return

openSessionFromDataSource(configuration.getDefaultExecutorType(), null, false);

}

//7. 进⼊penSessionFromDataSource。

//ExecutorType 为Executor的类型，TransactionIsolationLevel为事务隔离级别，

autoCommit是否开启事务

//openSession的多个重载⽅法可以指定获得的SeqSession的Executor类型和事务的处理

private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {

Transaction tx = null; try{

final Environment environment = configuration.getEnvironment(); final TransactionFactory transactionFactory =

getTransactionFactoryFromEnvironment(environment);

tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);

//根据参数创建指定类型的Executor

final Executor executor = configuration.newExecutor(tx, execType);

//返回的是 DefaultSqlSession

return new DefaultSqlSession(configuration, executor, autoCommit);

} catch(Exception e){

closeTransaction(tx); // may have fetched a connection so lets call

close()

}

执⾏ sqlsession 中的 api

//8.进⼊selectList⽅法，多个重载⽅法。

public <E > List < E > selectList(String statement) { return this.selectList(statement, null);

public <E > List < E > selectList(String statement, Object parameter)

{

return this.selectList(statement, parameter, RowBounds.DEFAULT); public <E > List < E > selectList(String statement, Object

parameter, RowBounds rowBounds) {

try {

//根据传⼊的全限定名+⽅法名从映射的Map中取出MappedStatement对象MappedStatement ms =

configuration.getMappedStatement(statement);

//调⽤Executor中的⽅法处理

//RowBounds是⽤来逻辑分⻚

// wrapCollection(parameter)是⽤来装饰集合或者数组参数

return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO\_RESULT\_HANDLER);

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error querying database. Cause: + e, e);

} finally {

ErrorContext.instance().reset();

}

**源码剖析-executor**

继续源码中的步骤，进⼊executor.query()

//此⽅法在SimpleExecutor的⽗类BaseExecutor中实现

public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException {

//根据传⼊的参数动态获得SQL语句，最后返回⽤BoundSql对象表示

BoundSql boundSql = ms.getBoundSql(parameter);

//为本次查询创建缓存的Key

CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql); return query(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

//进⼊query的重载⽅法中

public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId());

if (closed) {

throw new ExecutorException("Executor was closed.");

}

if (queryStack == 0 && ms.isFlushCacheRequired()) { clearLocalCache();

: null;

}

List<E> list; try {

queryStack++;

list = resultHandler == null ? (List<E>) localCache.getObject(key)

if (list != null) { handleLocallyCachedOutputParameters(ms, key, parameter,

boundSql);

} else {

//如果缓存中没有本次查找的值，那么从数据库中查询

list = queryFromDatabase(ms, parameter, rowBounds,

resultHandler, key, boundSql);

}

} finally {

queryStack--;

}

if (queryStack == 0) {

for (DeferredLoad deferredLoad : deferredLoads) { deferredLoad.load();

}

// issue #601 deferredLoads.clear();

if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) { // issue #482 clearLocalCache();

}

}

return list;

}

//从数据库查询

private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

List<E> list;

localCache.putObject(key, EXECUTION\_PLACEHOLDER); try {

//查询的⽅法

list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);

} finally {

localCache.removeObject(key);

}

//将查询结果放⼊缓存localCache.putObject(key, list);

if (ms.getStatementType() == StatementType.CALLABLE) { localOutputParameterCache.putObject(key, parameter);

}

return list;

}

// SimpleExecutor中实现⽗类的doQuery抽象⽅法

public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException

{

Statement stmt = null; try {

Configuration configuration = ms.getConfiguration();

//传⼊参数创建StatementHanlder对象来执⾏查询StatementHandler handler =

configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql);

//创建jdbc中的statement对象

stmt = prepareStatement(handler, ms.getStatementLog());

// StatementHandler 进⾏处理

return handler.query(stmt, resultHandler);

} finally {

closeStatement(stmt);

}

}

//创建Statement的⽅法

private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {

Statement stmt;

//条代码中的getConnection⽅法经过重重调⽤最后会调⽤openConnection⽅法，从连接池中获 得连接。

Connection connection = getConnection(statementLog);

stmt = handler.prepare(connection, transaction.getTimeout()); handler.parameterize(stmt);

return stmt;

}

//从连接池获得连接的⽅法

protected void openConnection() throws SQLException { if (log.isDebugEnabled()) {

log.debug("Opening JDBC Connection");

}

//从连接池获得连接

connection = dataSource.getConnection(); if (level != null) {

connection.setTransactionIsolation(level.getLevel());

}

}

上述的Executor.query()⽅法⼏经转折，最后会创建⼀个StatementHandler对象，然后将必要的参数传 递给

StatementHandler，使⽤StatementHandler来完成对数据库的查询，最终返回List结果集。从上⾯的代码中我们可以看出，Executor的功能和作⽤是：

(1、根据传递的参数，完成SQL语句的动态解析，⽣成BoundSql对象，供StatementHandler使⽤；

(2、为查询创建缓存，以提⾼性能

(3、创建JDBC的Statement连接对象，传递给\*StatementHandler\*对象，返回List查询结果。

**源码剖析-StatementHandler**

StatementHandler对象主要完成两个⼯作：

对于JDBC的PreparedStatement类型的对象，创建的过程中，我们使⽤的是SQL语句字符串会包含若⼲个？占位符，我们其后再对占位符进⾏设值。StatementHandler通过

parameterize(statement)⽅法对 S tatement 进⾏设值；

StatementHandler 通过 List query(Statement statement, ResultHandler resultHandler)⽅法来完成执⾏Statement，和将Statement对象返回的resultSet封装成List；

进⼊到 StatementHandler 的 parameterize(statement)⽅法的实现：

public void parameterize(Statement statement) throws SQLException {

//使⽤ParameterHandler对象来完成对Statement的设值parameterHandler.setParameters((PreparedStatement) statement);

}

/\*\* ParameterHandler 类 的 setParameters(PreparedStatement ps) 实 现

\* 对某⼀个Statement进⾏设置参数

\* \*/

public void setParameters(PreparedStatement ps) throws SQLException {

ErrorContext.instance().activity("setting parameters").object(mappedStatement.getParameterMap().getId());

List<ParameterMapping> parameterMappings = boundSql.getParameterMappings(); if (parameterMappings != null) { for (int i = 0; i <

parameterMappings.size(); i++) { ParameterMapping parameterMapping = parameterMappings.get(i); if (parameterMapping.getMode() != ParameterMode.OUT)

{ Object value;

String propertyName = parameterMapping.getProperty();

if (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask first for additional params

value = boundSql.getAdditionalParameter(propertyName);

} else if (parameterObject == null) { value = null;

} else if (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) { value = parameterObject;

} else {

MetaObject metaObject = configuration.newMetaObject(parameterObject);

value = metaObject.getValue(propertyName); }

// 每⼀个 Mapping都有⼀个 TypeHandler，根据 TypeHandler 来对

preparedStatement 进 ⾏设置参数

TypeHandler typeHandler = parameterMapping.getTypeHandler(); JdbcType jdbcType = parameterMapping.getJdbcType();

if (value == null && jdbcType == null) jdbcType = configuration.getJdbcTypeForNull();

//设置参数

typeHandler.setParameter(ps, i + 1, value, jdbcType);

}

}

}

}

从上述的代码可以看到,StatementHandler的parameterize(Statement)⽅法调⽤了

ParameterHandler的setParameters(statement)⽅法，

ParameterHandler的setParameters(Statement )⽅法负责根据我们输⼊的参数，对statement对象的

?占位符处进⾏赋值。

进⼊到StatementHandler 的 List query(Statement statement, ResultHandler resultHandler)⽅法的实现：

public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {

// 1.调⽤preparedStatemnt。execute()⽅法，然后将resultSet交给ResultSetHandler处

理

PreparedStatement ps = (PreparedStatement) statement; ps.execute();

//2.使⽤ ResultHandler 来处理 ResultSet

return resultSetHandler.<E> handleResultSets(ps);

}

从上述代码我们可以看出，StatementHandler 的List query(Statement statement, ResultHandler

resultHandler)⽅法的实现，是调⽤了 ResultSetHandler 的 handleResultSets(Statement)⽅法。

ResultSetHandler 的 handleResultSets(Statement)⽅法会将 Statement 语句执⾏后⽣成的 resultSet

结 果集转换成List结果集

public List<Object> handleResultSets(Statement stmt) throws SQLException {

ErrorContext.instance().activity("handling results").object(mappedStatement.getId());

//多ResultSet的结果集合，每个ResultSet对应⼀个Object对象。⽽实际上，每 个 Object 是

List<Object> 对象。

//在不考虑存储过程的多ResultSet的情况，普通的查询，实际就⼀个ResultSet，也 就是说，

multipleResults最多就⼀个元素。

final List<Object> multipleResults = new ArrayList<>();

int resultSetCount = 0;

//获得⾸个ResultSet对象，并封装成ResultSetWrapper对象ResultSetWrapper rsw = getFirstResultSet(stmt);

//获得ResultMap数组

//在不考虑存储过程的多ResultSet的情况，普通的查询，实际就⼀个ResultSet，也 就是说，resultMaps就⼀个元素。

List<ResultMap> resultMaps = mappedStatement.getResultMaps(); int resultMapCount = resultMaps.size();

validateResultMapsCount(rsw, resultMapCount); // 校 验

while (rsw != null && resultMapCount > resultSetCount) {

//获得ResultMap对象

ResultMap resultMap = resultMaps.get(resultSetCount);

//处理ResultSet，将结果添加到multipleResults中handleResultSet(rsw, resultMap, multipleResults, null);

//获得下⼀个ResultSet对象，并封装成ResultSetWrapper对象rsw = getNextResultSet(stmt);

//清理cleanUpAfterHandlingResultSet();

// resultSetCount ++ resultSetCount++;

}

}

//因为'mappedStatement.resultSets'只在存储过程中使⽤，本系列暂时不考虑，忽略即可String[] resultSets = mappedStatement.getResultSets();

if(resultSets!=null)

{

while (rsw != null && resultSetCount < resultSets.length) { ResultMapping parentMapping =

nextResultMaps.get(resultSets[resultSetCount]); if (parentMapping != null) {

String nestedResultMapId =

parentMapping.getNestedResultMapId();

ResultMap resultMap = configuration.getResultMap(nestedResultMapId);

handleResultSet(rsw, resultMap, null, parentMapping);

}

rsw = getNextResultSet(stmt); cleanUpAfterHandlingResultSet(); resultSetCount++;

}

}

//如果是multipleResults单元素，则取⾸元素返回

return collapseSingleResultList(multipleResults);

}

## Mapper代理⽅式：

回顾下写法:

public static void main(String[] args) {

//前三步都相同

InputStream inputStream = Resources.getResourceAsStream("sqlMapConfig.xml");

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSession sqlSession = factory.openSession();

//这⾥不再调⽤SqlSession的api,⽽是获得了接⼝对象，调⽤接⼝中的⽅法。UserMapper mapper = sqlSession.getMapper(UserMapper.class);

List<User> list = mapper.getUserByName("tom");

}

思考⼀个问题，通常的Mapper接⼝我们都没有实现的⽅法却可以使⽤，是为什么呢？答案很简单动态 代理

开始之前介绍⼀下MyBatis初始化时对接⼝的处理：MapperRegistry是Configuration中的⼀个属性，

它内部维护⼀个HashMap⽤于存放mapper接⼝的⼯⼚类，每个接⼝对应⼀个⼯⼚类。mappers中可以配置接⼝的包路径，或者某个具体的接⼝类。

<mappers>

<mapper class="com.lagou.mapper.UserMapper"/>

<package name="com.lagou.mapper"/>

</mappers>

* 当解析mappers标签时，它会判断解析到的是mapper配置⽂件时，会再将对应配置⽂件中的增删 改查标签 封装成MappedStatement对象，存⼊mappedStatements中。(上⽂介绍了)当

判断解析到接⼝时，会

建此接⼝对应的MapperProxyFactory对象，存⼊HashMap中，key =接⼝的字节码对象，value =此接

⼝对应的MapperProxyFactory对象。

**源码剖析-getmapper()**

进⼊ sqlSession.getMapper(UserMapper.class )中

//DefaultSqlSession 中 的 getMapper public <T> T getMapper(Class<T> type) {

return configuration.<T>getMapper(type, this);

}

//configuration 中 的 给 g etMapper

public <T> T getMapper(Class<T> type, SqlSession sqlSession) { return mapperRegistry.getMapper(type, sqlSession);

}

//MapperRegistry 中 的 g etMapper

public <T> T getMapper(Class<T> type, SqlSession sqlSession) {

//从 MapperRegistry 中的 HashMap 中拿 MapperProxyFactory final MapperProxyFactory<T> mapperProxyFactory =

(MapperProxyFactory<T>) knownMappers.get(type); if (mapperProxyFactory == null) {

throw new BindingException("Type " + type + " is not known to the MapperRegistry.");

}

try {

//通过动态代理⼯⼚⽣成示例。

return mapperProxyFactory.newInstance(sqlSession);

} catch (Exception e) {

throw new BindingException("Error getting mapper instance. Cause:

" + e, e);

}

}

//MapperProxyFactory 类中的 newInstance ⽅法

public T newInstance(SqlSession sqlSession) {

//创建了 JDK动态代理的Handler类

final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession, mapperInterface, methodCache);

//调⽤了重载⽅法

return newInstance(mapperProxy);

}

//MapperProxy 类，实现了 InvocationHandler 接⼝

public class MapperProxy<T> implements InvocationHandler, Serializable {

//省略部分源码

private final SqlSession sqlSession; private final Class<T> mapperInterface;

private final Map<Method, MapperMethod> methodCache;

//构造，传⼊了 SqlSession，说明每个session中的代理对象的不同的！

public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {

this.sqlSession = sqlSession; this.mapperInterface = mapperInterface; this.methodCache = methodCache;

}

//省略部分源码

}

**源码剖析-invoke()**

在动态代理返回了示例后，我们就可以直接调⽤mapper类中的⽅法了，但代理对象调⽤⽅法，执⾏是 在MapperProxy中的invoke⽅法中

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

try {

//如果是Object定义的⽅法，直接调⽤

if (Object.class.equals(method.getDeclaringClass())) { return method.invoke(this, args);

} else if (isDefaultMethod(method)) {

return invokeDefaultMethod(proxy, method, args);

}

} catch (Throwable t) {

throw ExceptionUtil.unwrapThrowable(t);

}

// 获 得 MapperMethod 对 象

final MapperMethod mapperMethod = cachedMapperMethod(method);

//重点在这：MapperMethod最终调⽤了执⾏的⽅法 return mapperMethod.execute(sqlSession, args);

}

进⼊execute⽅法：

public Object execute(SqlSession sqlSession, Object[] args) { Object result;

//判断mapper中的⽅法类型，最终调⽤的还是SqlSession中的⽅法 switch

(command.getType()) {

case INSERT: {

//转换参数

Object param = method.convertArgsToSqlCommandParam(args);

//执⾏INSERT操作

// 转 换 rowCount

result = rowCountResult(sqlSession.insert(command.getName(),

param));

break;

}

case UPDATE: {

//转换参数

param));

}

Object param = method.convertArgsToSqlCommandParam(args);

// 转 换 rowCount

result = rowCountResult(sqlSession.update(command.getName(),

break;

case DELETE: {

//转换参数

Object param = method.convertArgsToSqlCommandParam(args);

// 转 换 rowCount

result = rowCountResult(sqlSession.delete(command.getName(), param));

break;

}

⾏处理

case SELECT:

//⽆返回，并且有ResultHandler⽅法参数，则将查询的结果，提交给 ResultHandler 进

if (method.returnsVoid() && method.hasResultHandler()) { executeWithResultHandler(sqlSession, args);

result = null;

//执⾏查询，返回列表

} else if (method.returnsMany()) {

result = executeForMany(sqlSession, args);

//执⾏查询，返回Map

} else if (method.returnsMap()) {

result = executeForMap(sqlSession, args);

//执⾏查询，返回Cursor

} else if (method.returnsCursor()) {

result = executeForCursor(sqlSession, args);

//执⾏查询，返回单个对象

} else {

//转换参数

Object param = method.convertArgsToSqlCommandParam(args);

//查询单条

result = sqlSession.selectOne(command.getName(), param); if (method.returnsOptional() &&

(result == null ||

!method.getReturnType().equals(result.getClass()))) { result = Optional.ofNullable(result);

}

}

break;

case FLUSH:

result = sqlSession.flushStatements(); break;

default:

throw new BindingException("Unknown execution method for: " + command.getName());

}

//返回结果为null，并且返回类型为基本类型，则抛出BindingException异常if(result ==null&&method.getReturnType().isPrimitive()

&&!method.returnsVoid()){

throw new BindingException("Mapper method '" + command.getName() + " attempted to return null from a method with a primitive

return type(" + method.getReturnType() + "). ");

}

//返回结果return result;

}

## ⼆级缓存源码剖析：

⼆级缓存构建在⼀级缓存之上，在收到查询请求时，MyBatis ⾸先会查询⼆级缓存，若⼆级缓存未命中，再去查询⼀级缓存，⼀级缓存没有，再查询数据库。

⼆级缓存------》 ⼀级缓存 》数据库

与⼀级缓存不同，⼆级缓存和具体的命名空间绑定，⼀个Mapper中有⼀个Cache，相同Mapper中的

MappedStatement共⽤⼀个Cache，⼀级缓存则是和 SqlSession 绑定。

**启⽤⼆级缓存**

分为三步⾛：

1）开启全局⼆级缓存配置：

<settings>

<setting name="cacheEnabled" value="true"/>

</settings>

2) 在需要使⽤⼆级缓存的Mapper配置⽂件中配置标签

<cache></cache>

3）在具体CURD标签上配置 **useCache=true**

<select id="findById" resultType="com.lagou.pojo.User" useCache="true"> select \* from user where id = #{id}

</select>

**标签 < cache/> 的解析**

根据之前的mybatis源码剖析，xml的解析⼯作主要交给XMLConfigBuilder.parse()⽅法来实现

// XMLConfigBuilder.parse() public Configuration parse() {

if (parsed) {

throw new BuilderException("Each XMLConfigBuilder can only be used

once.");

}

parsed = true; parseConfiguration(parser.evalNode("/configuration"));// 在 这 ⾥return configuration;

}

// parseConfiguration()

// 既然是在xml中添加的，那么我们就直接看关于mappers标签的解析

private void parseConfiguration(XNode root) { try {

Properties settings = settingsAsPropertiess(root.evalNode("settings"));

propertiesElement(root.evalNode("properties")); loadCustomVfs(settings); typeAliasesElement(root.evalNode("typeAliases")); pluginElement(root.evalNode("plugins")); objectFactoryElement(root.evalNode("objectFactory")); objectWrapperFactoryElement(root.evalNode("objectWrapperFactory")); reflectionFactoryElement(root.evalNode("reflectionFactory")); settingsElement(settings);

// read it after objectFactory and objectWrapperFactory issue #631 environmentsElement(root.evalNode("environments")); databaseIdProviderElement(root.evalNode("databaseIdProvider")); typeHandlerElement(root.evalNode("typeHandlers"));

// 就是这⾥

mapperElement(root.evalNode("mappers"));

} catch (Exception e) {

throw new BuilderException("Error parsing SQL Mapper Configuration.

Cause: " + e, e);

}

}

// mapperElement()

private void mapperElement(XNode parent) throws Exception { if (parent != null) {

for (XNode child : parent.getChildren()) { if ("package".equals(child.getName())) {

String mapperPackage = child.getStringAttribute("name"); configuration.addMappers(mapperPackage);

} else {

String resource = child.getStringAttribute("resource"); String url = child.getStringAttribute("url");

String mapperClass = child.getStringAttribute("class");

// 按照我们本例的配置，则直接⾛该if判断

if (resource != null && url == null && mapperClass == null) { ErrorContext.instance().resource(resource);

InputStream inputStream = Resources.getResourceAsStream(resource);

XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream, configuration, resource, configuration.getSqlFragments());

// ⽣成XMLMapperBuilder，并执⾏其parse⽅法

mapperParser.parse();

} else if (resource == null && url != null && mapperClass ==

null) {

ErrorContext.instance().resource(url);

InputStream inputStream = Resources.getUrlAsStream(url); XMLMapperBuilder mapperParser = new

XMLMapperBuilder(inputStream, configuration, url, configuration.getSqlFragments());

mapperParser.parse();

} else if (resource == null && url == null && mapperClass !=

null) {

Class<?> mapperInterface =

Resources.classForName(mapperClass);

configuration.addMapper(mapperInterface);

} else {

throw new BuilderException("A mapper element may only specify a url, resource or class, but not more than one.");

}

}

}

}

}

我们来看看解析Mapper.xml

// XMLMapperBuilder.parse() public void parse() {

if (!configuration.isResourceLoaded(resource)) {

// 解析mapper属性configurationElement(parser.evalNode("/mapper")); configuration.addLoadedResource(resource); bindMapperForNamespace();

}

parsePendingResultMaps();

parsePendingChacheRefs(); parsePendingStatements();

}

// configurationElement()

private void configurationElement(XNode context) { try {

String namespace = context.getStringAttribute("namespace"); if (namespace == null || namespace.equals("")) {

throw new BuilderException("Mapper's namespace cannot be empty");

}

builderAssistant.setCurrentNamespace(namespace); cacheRefElement(context.evalNode("cache-ref"));

// 最终在这⾥看到了关于cache属性的处理

cacheElement(context.evalNode("cache")); parameterMapElement(context.evalNodes("/mapper/parameterMap")); resultMapElements(context.evalNodes("/mapper/resultMap")); sqlElement(context.evalNodes("/mapper/sql"));

// 这⾥会将⽣成的Cache包装到对应的MappedStatement

buildStatementFromContext(context.evalNodes("select|insert|update|delete"));

} catch (Exception e) {

throw new BuilderException("Error parsing Mapper XML. Cause: " + e,

e);

}

}

// cacheElement()

private void cacheElement(XNode context) throws Exception { if (context != null) {

//解析<cache/>标签的type属性，这⾥我们可以⾃定义cache的实现类，⽐如redisCache，

如果没有⾃定义，这⾥使⽤和⼀级缓存相同的PERPETUAL

String type = context.getStringAttribute("type", "PERPETUAL"); Class<? extends Cache> typeClass =

typeAliasRegistry.resolveAlias(type);

String eviction = context.getStringAttribute("eviction", "LRU"); Class<? extends Cache> evictionClass =

typeAliasRegistry.resolveAlias(eviction);

Long flushInterval = context.getLongAttribute("flushInterval"); Integer size = context.getIntAttribute("size");

boolean readWrite = !context.getBooleanAttribute("readOnly", false); boolean blocking = context.getBooleanAttribute("blocking", false); Properties props = context.getChildrenAsProperties();

// 构建Cache对象

builderAssistant.useNewCache(typeClass, evictionClass, flushInterval, size, readWrite, blocking, props);

}

}

先来看看是如何构建Cache对象的

**MapperBuilderAssistant.useNewCache()**

public Cache useNewCache(Class<? extends Cache> typeClass,

Class<? extends Cache> evictionClass, Long flushInterval,

Integer size, boolean readWrite, boolean blocking, Properties props) {

// 1.⽣成Cache对象

Cache cache = new CacheBuilder(currentNamespace)

//这⾥如果我们定义了<cache/>中的type，就使⽤⾃定义的Cache,否则使⽤和⼀级缓存相同的PerpetualCache

.implementation(valueOrDefault(typeClass, PerpetualCache.class))

.addDecorator(valueOrDefault(evictionClass, LruCache.class))

.clearInterval(flushInterval)

.size(size)

.readWrite(readWrite)

.blocking(blocking)

.properties(props)

.build();

// 2.添加到Configuration中configuration.addCache(cache);

// 3.并将cache赋值给MapperBuilderAssistant.currentCache currentCache = cache;

return cache;

}

我们看到⼀个Mapper.xml只会解析⼀次标签，也就是只创建⼀次Cache对象，放进configuration中， 并将cache赋值给MapperBuilderAssistant.currentCache

**buildStatementFromContext(context.evalNodes("select|insert|update|delete"));将Cache**

**包装到MappedStatement**

// buildStatementFromContext()

private void buildStatementFromContext(List<XNode> list) { if (configuration.getDatabaseId() != null) {

buildStatementFromContext(list, configuration.getDatabaseId());

}

buildStatementFromContext(list, null);

}

//buildStatementFromContext()

private void buildStatementFromContext(List<XNode> list, String requiredDatabaseId) {

for (XNode context : list) {

final XMLStatementBuilder statementParser = new XMLStatementBuilder(configuration, builderAssistant, context, requiredDatabaseId);

try {

// 每⼀条执⾏语句转换成⼀个MappedStatement statementParser.parseStatementNode();

} catch (IncompleteElementException e) { configuration.addIncompleteStatement(statementParser);

}

}

}

// XMLStatementBuilder.parseStatementNode(); public void parseStatementNode() {

String id = context.getStringAttribute("id");

String databaseId = context.getStringAttribute("databaseId");

...

Integer fetchSize = context.getIntAttribute("fetchSize"); Integer timeout = context.getIntAttribute("timeout");

String parameterMap = context.getStringAttribute("parameterMap"); String parameterType = context.getStringAttribute("parameterType"); Class<?> parameterTypeClass = resolveClass(parameterType);

String resultMap = context.getStringAttribute("resultMap"); String resultType = context.getStringAttribute("resultType"); String lang = context.getStringAttribute("lang"); LanguageDriver langDriver = getLanguageDriver(lang);

...

// 创建MappedStatement对象

builderAssistant.addMappedStatement(id, sqlSource, statementType, sqlCommandType,

fetchSize, timeout, parameterMap, parameterTypeClass, resultMap, resultTypeClass,

resultSetTypeEnum, flushCache,

useCache, resultOrdered,

databaseId, langDriver, resultSets);

}

keyGenerator, keyProperty, keyColumn,

// builderAssistant.addMappedStatement() public MappedStatement addMappedStatement(

String id,

...) {

if (unresolvedCacheRef) {

throw new IncompleteElementException("Cache-ref not yet resolved");

}

id = applyCurrentNamespace(id, false);

boolean isSelect = sqlCommandType == SqlCommandType.SELECT;

//创建MappedStatement对象MappedStatement.Builder statementBuilder = new

MappedStatement.Builder(configuration, id, sqlSource, sqlCommandType)

...

.flushCacheRequired(valueOrDefault(flushCache, !isSelect))

.useCache(valueOrDefault(useCache, isSelect))

.cache(currentCache);// 在这⾥将之前⽣成的Cache封装到MappedStatement

ParameterMap statementParameterMap = getStatementParameterMap(parameterMap, parameterType, id);

if (statementParameterMap != null) { statementBuilder.parameterMap(statementParameterMap);

}

MappedStatement statement = statementBuilder.build(); configuration.addMappedStatement(statement);

return statement;

}

我们看到将Mapper中创建的Cache对象，加⼊到了每个MappedStatement对象中，也就是同⼀个

Mapper中所有的2

有关于标签的解析就到这了。**查询源码分析CachingExecutor**

// CachingExecutor

public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException {

BoundSql boundSql = ms.getBoundSql(parameterObject);

// 创 建 CacheKey

CacheKey key = createCacheKey(ms, parameterObject, rowBounds, boundSql); return query(ms, parameterObject, rowBounds, resultHandler, key,

boundSql);

}

public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql)

throws SQLException {

// 从 MappedStatement 中获取 Cache，注意这⾥的 Cache 是从MappedStatement中获取的

// 也就是我们上⾯解析Mapper中<cache/>标签中创建的，它保存在Configration中

// 我们在上⾯解析blog.xml时分析过每⼀个MappedStatement都有⼀个Cache对象，就是这⾥

Cache cache = ms.getCache();

// 如果配置⽂件中没有配置 <cache>，则 cache 为空

if (cache != null) {

//如果需要刷新缓存的话就刷新：flushCache="true" flushCacheIfRequired(ms);

if (ms.isUseCache() && resultHandler == null) { ensureNoOutParams(ms, boundSql);

// 访问⼆级缓存

List<E> list = (List<E>) tcm.getObject(cache, key);

// 缓存未命中

if (list == null) {

// 如果没有值，则执⾏查询，这个查询实际也是先⾛⼀级缓存查询，⼀级缓存也没有的话，则进⾏DB查询

list = delegate.<E>query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);

// 缓存查询结果

tcm.putObject(cache, key, list);

}

return list;

}

}

return delegate.<E>query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);

}

如果设置了**flushCache="true"，则每次查询都会刷新缓存**

<!-- 执⾏此语句清空缓存 -->

<select id="findbyId" resultType="com.lagou.pojo.user" useCache="true" flushCache="true" >

select \* from t\_demo

</select>

如上，注意⼆级缓存是从 MappedStatement 中获取的。由于 MappedStatement 存在于全局配置中，可以多个 CachingExecutor 获取到，这样就会出现线程安全问题。除此之外，若不加以控制，多个事务共⽤⼀个缓存实例，会导致脏读问题。⾄于脏读问题，需要借助其他类来处理，也就是上⾯代码中 tcm 变量对应的类型。下⾯分析⼀下。

**TransactionalCacheManager**

/\*\* 事务缓存管理器 \*/

public class TransactionalCacheManager {

// Cache 与 TransactionalCache 的映射关系表

private final Map<Cache, TransactionalCache> transactionalCaches = new

HashMap<Cache, TransactionalCache>();

public void clear(Cache cache) {

// 获取 TransactionalCache 对象，并调⽤该对象的 clear ⽅法，下同

getTransactionalCache(cache).clear();

}

public Object getObject(Cache cache, CacheKey key) {

// 直接从TransactionalCache中获取缓存

return getTransactionalCache(cache).getObject(key);

}

public void putObject(Cache cache, CacheKey key, Object value) {

// 直接存⼊TransactionalCache的缓存中

getTransactionalCache(cache).putObject(key, value);

}

public void commit() {

for (TransactionalCache txCache : transactionalCaches.values()) { txCache.commit();

}

}

public void rollback() {

for (TransactionalCache txCache : transactionalCaches.values()) { txCache.rollback();

}

}

private TransactionalCache getTransactionalCache(Cache cache) {

// 从映射表中获取 TransactionalCache

TransactionalCache txCache = transactionalCaches.get(cache); if (txCache == null) {

// TransactionalCache 也是⼀种装饰类，为 Cache 增加事务功能

// 创建⼀个新的TransactionalCache，并将真正的Cache对象存进去

txCache = new TransactionalCache(cache); transactionalCaches.put(cache, txCache);

}

return txCache;

}

}

TransactionalCacheManager 内部维护了 Cache 实例与 TransactionalCache 实例间的映射关系，该类也仅负责维护两者的映射关系，真正做事的还是 TransactionalCache。TransactionalCache 是⼀种缓 存装饰器，可以为 Cache 实例增加事务功能。我在之前提到的脏读问题正是由该类进⾏处理的。下⾯分析⼀下该类的逻辑。

**TransactionalCache**

public class TransactionalCache implements Cache {

//真正的缓存对象，和上⾯的Map<Cache, TransactionalCache>中的Cache是同⼀个private final Cache delegate;

private boolean clearOnCommit;

// 在事务被提交前，所有从数据库中查询的结果将缓存在此集合中

private final Map<Object, Object> entriesToAddOnCommit;

// 在事务被提交前，当缓存未命中时，CacheKey 将会被存储在此集合中

private final Set<Object> entriesMissedInCache;

@Override

public Object getObject(Object key) {

// 查询的时候是直接从delegate中去查询的，也就是从真正的缓存对象中查询

Object object = delegate.getObject(key);

if (object == null) {

// 缓存未命中，则将 key 存⼊到 entriesMissedInCache 中

entriesMissedInCache.add(key);

}

if (clearOnCommit) { return null;

} else {

return object;

}

}

@Override

public void putObject(Object key, Object object) {

// 将键值对存⼊到 entriesToAddOnCommit 这个Map中中，⽽⾮真实的缓存对象

delegate 中

entriesToAddOnCommit.put(key, object);

}

@Override

public Object removeObject(Object key) { return null;

}

@Override

public void clear() { clearOnCommit = true;

// 清空 entriesToAddOnCommit，但不清空 delegate 缓存

entriesToAddOnCommit.clear();

}

public void commit() {

// 根据 clearOnCommit 的值决定是否清空 delegate if (clearOnCommit) {

delegate.clear();

}

// 刷新未缓存的结果到 delegate 缓存中

flushPendingEntries();

// 重 置 entriesToAddOnCommit 和 entriesMissedInCache reset();

}

public void rollback() { unlockMissedEntries(); reset();

}

private void reset() {

clearOnCommit = false;

// 清空集合entriesToAddOnCommit.clear(); entriesMissedInCache.clear();

}

private void flushPendingEntries() {

for (Map.Entry<Object, Object> entry : entriesToAddOnCommit.entrySet()) {

// 将 entriesToAddOnCommit 中的内容转存到 delegate 中

delegate.putObject(entry.getKey(), entry.getValue());

}

for (Object entry : entriesMissedInCache) {

if (!entriesToAddOnCommit.containsKey(entry)) {

// 存⼊空值

delegate.putObject(entry, null);

}

}

}

private void unlockMissedEntries() {

for (Object entry : entriesMissedInCache) { try {

// 调⽤ removeObject 进⾏解锁

delegate.removeObject(entry);

} catch (Exception e) { log.warn("...");

}

}

}

}

存储⼆级缓存对象的时候是放到了TransactionalCache.entriesToAddOnCommit这个map中，但是每 次查询的时候是直接从TransactionalCache.delegate中去查询的，所以这个⼆级缓存查询数据库后，设 置缓存值是没有⽴刻⽣效的，主要是因为直接存到 delegate 会导致脏数据问题

**为何只有SqlSession提交或关闭之后？** 那我们来看下SqlSession.commit()⽅法做了什么**SqlSession**

@Override

public void commit(boolean force) { try {

// 主要是这句

executor.commit(isCommitOrRollbackRequired(force));

dirty = false;

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error committing transaction.

Cause: " + e, e);

} finally {

ErrorContext.instance().reset();

}

}

// CachingExecutor.commit() @Override

public void commit(boolean required) throws SQLException {

delegate.commit(required); tcm.commit();// 在这⾥

}

// TransactionalCacheManager.commit() public void commit() {

for (TransactionalCache txCache : transactionalCaches.values()) { txCache.commit();// 在这⾥

}

}

// TransactionalCache.commit() public void commit() {

if (clearOnCommit) { delegate.clear();

}

flushPendingEntries();//这⼀句reset();

}

// TransactionalCache.flushPendingEntries() private void flushPendingEntries() {

for (Map.Entry<Object, Object> entry : entriesToAddOnCommit.entrySet()) {

// 在这⾥真正的将entriesToAddOnCommit的对象逐个添加到delegate中，只有这时，⼆级缓存才真正的⽣效

delegate.putObject(entry.getKey(), entry.getValue());

}

for (Object entry : entriesMissedInCache) {

if (!entriesToAddOnCommit.containsKey(entry)) { delegate.putObject(entry, null);

}

}

}

### ⼆级缓存的刷新

我们来看看SqlSession的更新操作

public int update(String statement, Object parameter) { int var4;

try {

this.dirty = true;

MappedStatement ms = this.configuration.getMappedStatement(statement); var4 = this.executor.update(ms, this.wrapCollection(parameter));

} catch (Exception var8) {

throw ExceptionFactory.wrapException("Error updating database. Cause: " + var8, var8);

} finally {

ErrorContext.instance().reset();

}

return var4;

}

public int update(MappedStatement ms, Object parameterObject) throws SQLException {

this.flushCacheIfRequired(ms);

return this.delegate.update(ms, parameterObject);

}

private void flushCacheIfRequired(MappedStatement ms) {

//获取MappedStatement对应的Cache，进⾏清空Cache cache = ms.getCache();

//SQL需设置flushCache="true" 才会执⾏清空

if (cache != null && ms.isFlushCacheRequired()) { this.tcm.clear(cache);

}

}

MyBatis⼆级缓存只适⽤于不常进⾏增、删、改的数据，⽐如国家⾏政区省市区街道数据。⼀但数据变 更，MyBatis会清空缓存。因此⼆级缓存不适⽤于经常进⾏更新的数据。

**总结：**

在⼆级缓存的设计上，MyBatis⼤量地运⽤了装饰者模式，如CachingExecutor, 以及各种Cache接⼝的装饰器。

⼆级缓存实现了Sqlsession之间的缓存数据共享，属于namespace级别

⼆级缓存具有丰富的缓存策略。

⼆级缓存可由多个装饰器，与基础缓存组合⽽成

⼆级缓存⼯作由 ⼀个缓存装饰执⾏器CachingExecutor和 ⼀个事务型预缓存TransactionalCache 完成。

* 1. **延迟加载源码剖析：**

**什么是延迟加载？**

**问题**

在开发过程中很多时候我们并不需要总是在加载⽤户信息时就⼀定要加载他的订单信息。此时就是我 们所说的延迟加载。

**举个栗⼦**

* 在⼀对多中，当我们有⼀个⽤户，它有个100个订单在查询⽤户的时候，要不要把关联的订单查出来？ 在查询订单的时候，要不要把关联的⽤户查出来？
* 回答

在查询⽤户时，⽤户下的订单应该是，什么时候⽤，什么时候查询。在查询订单时，订单所属的⽤户信息应该是随着订单⼀起查询出来。

**延迟加载**

就是在需要⽤到数据时才进⾏加载，不需要⽤到数据时就不加载数据。延迟加载也称懒加载。

* 优点：

先从单表查询，需要时再从关联表去关联查询，⼤⼤提⾼数据库性能，因为查询单表要⽐关联查询多张表速度要快。

* 缺点：

因为只有当需要⽤到数据时，才会进⾏数据库查询，这样在⼤批量数据查询时，因为查询⼯作也要消耗时间，所以可能造成⽤户等待时间变⻓，造成⽤户体验下降。

* 在多表中：

⼀对多，多对多：通常情况下采⽤延迟加载

⼀对⼀（多对⼀）：通常情况下采⽤⽴即加载

* 注意：

延迟加载是基于嵌套查询来实现的

**实现**

**局部延迟加载**

在association和collection标签中都有⼀个fetchType属性，通过修改它的值，可以修改局部的加载策 略。

<!-- 开启⼀对多 延迟加载 -->

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

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

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

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

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

<!--

fetchType="lazy" 懒加载策略

fetchType="eager" ⽴即加载策略

-->

<collection property="orderList" ofType="order" column="id" select="com.lagou.dao.OrderMapper.findByUid" fetchType="lazy">

</collection>

</resultMap>

<select id="findAll" resultMap="userMap"> SELECT \* FROM `user`

</select>

**全局延迟加载**

在Mybatis的核⼼配置⽂件中可以使⽤setting标签修改全局的加载策略。

<settings>

<!--开启全局延迟加载功能-->

<setting name="lazyLoadingEnabled" value="true"/>

</settings>

**注意**

7.。

<!-- 关闭⼀对⼀ 延迟加载 -->

<resultMap id="orderMap" type="order">

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

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

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

<!--

fetchType="lazy" 懒加载策略

fetchType="eager" ⽴即加载策略

-->

<association property="user" column="uid" javaType="user" select="com.lagou.dao.UserMapper.findById" fetchType="eager">

</association>

</resultMap>

<select id="findAll" resultMap="orderMap"> SELECT \* from orders

</select>

**延迟加载原理实现**

它的原理是，使⽤ CGLIB 或 Javassist( 默认 ) 创建⽬标对象的代理对象。当调⽤代理对象的延迟加载属

a.getB().getName()

性的 getting ⽅法时，**进⼊拦截器⽅法**。⽐如调⽤

a.getB()

⽅法，进⼊拦截器的

⽅法，发现

invoke(...)

需要延迟加载时，那么就会单独发送事先保存好的查询关联 B

对象的 SQL ，把 B 查询上来，然后调⽤ a.setB(b) ⽅法，于是成 a.getB().getName() ⽅法的调⽤。这就是延迟加载的基本原理



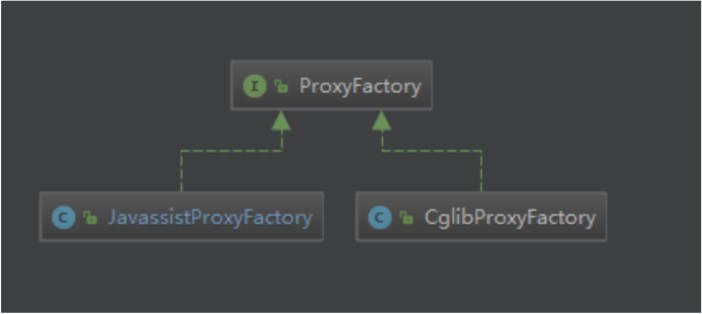
a

对象 b 属性就有值了，接着完

总结：延迟加载主要是通过动态代理的形式实现，通过代理拦截到指定⽅法，执⾏数据加载。

**延迟加载原理（源码剖析)**

MyBatis延迟加载主要使⽤：Javassist，Cglib实现，类图展示：



**Setting 配置加载：**

public class Configuration {

/\*\* aggressiveLazyLoading：

* 当开启时，任何⽅法的调⽤都会加载该对象的所有属性。否则，每个属性会按需加载（参考

lazyLoadTriggerMethods).

* 默认为true

\* \*/

protected boolean aggressiveLazyLoading;

/\*\*

* 延迟加载触发⽅法

\*/

protected Set<String> lazyLoadTriggerMethods = new HashSet<String> (Arrays.asList(new String[] { "equals", "clone", "hashCode", "toString" }));

/\*\* 是否开启延迟加载 \*/

protected boolean lazyLoadingEnabled = false;

/\*\*

* 默认使⽤Javassist代理⼯⼚
* @param proxyFactory

\*/

public void setProxyFactory(ProxyFactory proxyFactory) { if (proxyFactory == null) {

proxyFactory = new JavassistProxyFactory();

}

this.proxyFactory = proxyFactory;

}

//省略...

}

**延迟加载代理对象创建**

Mybatis的查询结果是由ResultSetHandler接⼝的handleResultSets()⽅法处理的。ResultSetHandler

接⼝只有⼀个实现，DefaultResultSetHandler，接下来看下延迟加载相关的⼀个核⼼的⽅法

<code class="language-Java">//#mark 创建结果对象

private Object createResultObject(ResultSetWrapper rsw, ResultMap resultMap, ResultLoaderMap lazyLoader, String columnPrefix) throws SQLException {

this.useConstructorMappings = false; // reset previous mapping result final List&lt;Class&lt;?&gt;&gt; constructorArgTypes = new

ArrayList&lt;Class&lt;?&gt;&gt;();

final List&lt;Object&gt; constructorArgs = new ArrayList&lt;Object&gt;();

//#mark 创建返回的结果映射的真实对象

Object resultObject = createResultObject(rsw, resultMap, constructorArgTypes, constructorArgs, columnPrefix);

if (resultObject != null &amp;&amp; !hasTypeHandlerForResultObject(rsw, resultMap.getType())) {

final List&lt;ResultMapping&gt; propertyMappings = resultMap.getPropertyResultMappings();

for (ResultMapping propertyMapping : propertyMappings) {

// 判断属性有没配置嵌套查询，如果有就创建代理对象

if (propertyMapping.getNestedQueryId() != null &amp;&amp; propertyMapping.isLazy()) {

//#mark 创建延迟加载代理对象

resultObject = configuration.getProxyFactory().createProxy(resultObject, lazyLoader, configuration, objectFactory, constructorArgTypes, constructorArgs);

break;

}

}

}

this.useConstructorMappings = resultObject != null &amp;&amp;

!constructorArgTypes.isEmpty(); // set current mapping result return resultObject;

}

默认采⽤javassistProxy进⾏代理对象的创建



JavasisstProxyFactory实现

public class JavassistProxyFactory implements org.apache.ibatis.executor.loader.ProxyFactory {

/\*\*

* 接⼝实现
* @param target ⽬标结果对象
* @param lazyLoader 延迟加载对象
* @param configuration 配 置
* @param objectFactory 对象⼯⼚
* @param constructorArgTypes 构造参数类型
* @param constructorArgs 构造参数值
* @return

\*/ @Override

public Object createProxy(Object target, ResultLoaderMap lazyLoader, Configuration configuration, ObjectFactory objectFactory, List&lt;Class&lt;? &gt;&gt; constructorArgTypes, List&lt;Object&gt; constructorArgs) {

return EnhancedResultObjectProxyImpl.createProxy(target, lazyLoader, configuration, objectFactory, constructorArgTypes, constructorArgs);

}

//省略...

/\*\*

* 代理对象实现，核⼼逻辑执⾏

\*/

private static class EnhancedResultObjectProxyImpl implements MethodHandler

{

/\*\*

* 创建代理对象
* @param type
* @param callback
* @param constructorArgTypes
* @param constructorArgs
* @return

\*/

static Object crateProxy(Class&lt;?&gt; type, MethodHandler callback, List&lt;Class&lt;?&gt;&gt; constructorArgTypes, List&lt;Object&gt; constructorArgs) {

ProxyFactory enhancer = new ProxyFactory(); enhancer.setSuperclass(type);

try {

//通过获取对象⽅法，判断是否存在该⽅法type.getDeclaredMethod(WRITE\_REPLACE\_METHOD);

// ObjectOutputStream will call writeReplace of objects returned by writeReplace

if (log.isDebugEnabled()) {

log.debug(WRITE\_REPLACE\_METHOD + &quot; method was found on bean &quot; + type + &quot;, make sure it returns this&quot;);

}

} catch (NoSuchMethodException e) {

//没找到该⽅法，实现接⼝

enhancer.setInterfaces(new Class[]{WriteReplaceInterface.class});

} catch (SecurityException e) {

// nothing to do here

}

Object enhanced;

Class&lt;?&gt;[] typesArray = constructorArgTypes.toArray(new Class[constructorArgTypes.size()]);

Object[] valuesArray = constructorArgs.toArray(new Object[constructorArgs.size()]);

try {

//创建新的代理对象

enhanced = enhancer.create(typesArray, valuesArray);

} catch (Exception e) {

throw new ExecutorException(&quot;Error creating lazy proxy. Cause: &quot; + e, e);

}

//设置代理执⾏器

((Proxy) enhanced).setHandler(callback); return enhanced;

}

/\*\*

* + 代理对象执⾏
  + @param enhanced 原对象
  + @param method 原对象⽅法
  + @param methodProxy 代理⽅法
  + @param args ⽅法参数
  + @return
  + @throws Throwable

\*/ @Override

public Object invoke(Object enhanced, Method method, Method methodProxy, Object[] args) throws Throwable {

final String methodName = method.getName(); try {

synchronized (lazyLoader) {

if (WRITE\_REPLACE\_METHOD.equals(methodName)) {

//忽略暂未找到具体作⽤Object original;

if (constructorArgTypes.isEmpty()) { original = objectFactory.create(type);

} else {

original = objectFactory.create(type, constructorArgTypes, constructorArgs);

}

PropertyCopier.copyBeanProperties(type, enhanced, original); if (lazyLoader.size() &gt; 0) {

return new JavassistSerialStateHolder(original, lazyLoader.getProperties(), objectFactory, constructorArgTypes, constructorArgs);

} else {

return original;

}

} else {

//延迟加载数量⼤于0

if (lazyLoader.size() &gt; 0 &amp;&amp;

!FINALIZE\_METHOD.equals(methodName)) {

//aggressive ⼀次加载性所有需要要延迟加载属性或者包含触发延迟加载⽅法

if (aggressive || lazyLoadTriggerMethods.contains(methodName)) { log.debug(&quot;==&gt; laze lod trigger method:&quot; +

methodName + &quot;,proxy method:&quot; + methodProxy.getName() + &quot; class:&quot; + enhanced.getClass());

//⼀次全部加载

lazyLoader.loadAll();

} else if (PropertyNamer.isSetter(methodName)) {

//判断是否为set⽅法，set⽅法不需要延迟加载final String property =

PropertyNamer.methodToProperty(methodName);

lazyLoader.remove(property);

} else if (PropertyNamer.isGetter(methodName)) { final String property =

PropertyNamer.methodToProperty(methodName);

if (lazyLoader.hasLoader(property)) {

//延迟加载单个属性lazyLoader.load(property);

log.debug(&quot;load one :&quot; + methodName);

}

}

}

}

}

return methodProxy.invoke(enhanced, args);

} catch (Throwable t) {

throw ExceptionUtil.unwrapThrowable(t);

}

}

}

**注意事项**

1. IDEA调试问题 当配置aggressiveLazyLoading=true，在使⽤IDEA进⾏调试的时候，如果断点打到代理执⾏逻辑当中，你会发现延迟加载的代码永远都不能进⼊，总是会被提前执⾏。 主要产⽣的原因在aggressiveLazyLoading，因为在调试的时候，IDEA的Debuger窗体中已经触发了延迟加载 对象的⽅法。

**第⼗⼀部分：设计模式**

虽然我们都知道有3类23种设计模式，但是⼤多停留在概念层⾯，Mybatis源码中使⽤了⼤量的设计模 式，观察设计模式在其中的应⽤，能够更深⼊的理解设计模式

Mybati s⾄少⽤到了以下的设计模式的使⽤：

|  |  |
| --- | --- |
| **模式** | **mybatis 体现** |
| Builder  模式 | 例如SqlSessionFactoryBuilder、Environment; |
| ⼯⼚⽅法模式 | 例如SqlSessionFactory、TransactionFactory、LogFactory |
| 单例模式 | 例如 ErrorContext 和 LogFactory； |
| 代理模式 | Mybatis实现的核⼼，⽐如MapperProxy、ConnectionLogger，⽤的jdk的动态代理还有executor.loader包使⽤了 cglib或者javassist达到延迟加载的效果 |
| 组合模式 | 例如SqlNode和各个⼦类ChooseSqlNode等； |
| 模板⽅法模式 | 例如 BaseExecutor 和 SimpleExecutor，还有 BaseTypeHandler 和所有的⼦类例如  IntegerTypeHandler； |
| 适配器模式 | 例如Log的Mybatis接⼝和它对jdbc、log4j等各种⽇志框架的适配实现； |
| 装饰者模式 | 例如Cache包中的cache.decorators⼦包中等各个装饰者的实现； |
| 迭代器模式 | 例如迭代器模式PropertyTokenizer； |

接下来对Builder构建者模式、⼯⼚模式、代理模式进⾏解读，先介绍模式⾃身的知识，然后解读在

Mybatis中怎样应⽤了该模式。

* 1. **Builder构建者模式**

Builder模式的定义是"将⼀个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表 示。”，它属于创建类模式，⼀般来说，如果⼀个对象的构建⽐较复杂，超出了构造函数所能包含的范 围，就可以使⽤⼯⼚模式和Builder模式，相对于⼯⼚模式会产出⼀个完整的产品，Builder应⽤于更加 复杂的对象的构建，甚⾄只会构建产品的⼀个部分，直⽩来说，就是使⽤多个简单的对象⼀步⼀步构建 成⼀个复杂的对象

**例⼦：使⽤构建者设计模式来⽣产computer 主要步骤：**

1、将需要构建的⽬标类分成多个部件（电脑可以分为主机、显示器、键盘、⾳箱等部件）；

2、 创建构建类；

3、 依次创建部件；

4、 将部件组装成⽬标对象

**1. 定 义 computer**

package com.lagou.dao;

import org.apache.ibatis.binding.BindingException; import org.apache.ibatis.session.SqlSession;

import java.util.Optional;

public class Computer { private String displayer; private String mainUnit; private String mouse; private String keyboard;

public String getDisplayer() { return displayer;

}

public void setDisplayer(String displayer) { this.displayer = displayer;

}

public String getMainUnit() { return mainUnit;

}

public void setMainUnit(String mainUnit) { this.mainUnit = mainUnit;

}

public String getMouse() { return mouse;

}

public void setMouse(String mouse) { this.mouse = mouse;

}

public String getKeyboard() { return keyboard;

}

public void setKeyboard(String keyboard) { this.keyboard = keyboard;

}

@Override

public String toString() {

return "Computer{" + "displayer='" + displayer + '\'' + ", mainUnit='"

+ mainUnit + '\'' + ", mouse='" + mouse + '\'' + ", keyboard='" + keyboard + '\'' + '}';

}

ComputerBuilder

public static class ComputerBuilder {

private ComputerBuilder target = new ComputerBuilder();

public Builder installDisplayer(String displayer) { target.setDisplayer(displayer);

return this;

}

public Builder installMainUnit(String mainUnit) { target.setMainUnit(mainUnit);

return this;

}

public Builder installMouse(String mouse) { target.setMouse(mouse);

return this;

}

public Builder installKeybord(String keyboard) { target.setKeyboard(keyboard);

return this;

}

public ComputerBuilder build() { return target;

}

}

**调⽤**

public static void main(String[]args){

ComputerBuilder computerBuilder=new ComputerBuilder();

computerBuilder.installDisplayer("显万器");

computerBuilder.installMainUnit("主机");

computerBuilder.installKeybord("键盘");

computerBuilder.installMouse("⿏标");

Computer computer=computerBuilder.Builder(); System.out.println(computer);

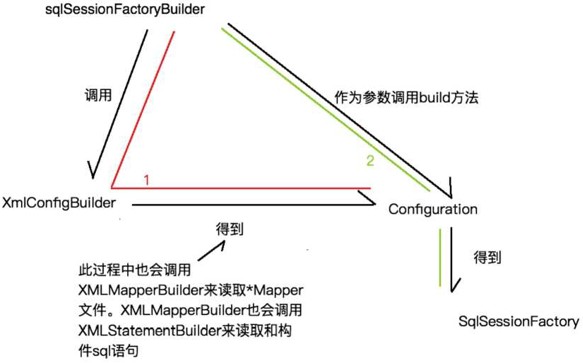
}

**Mybatis中的体现**

SqlSessionFactory 的构建过程：

Mybatis的初始化⼯作⾮常复杂，不是只⽤⼀个构造函数就能搞定的。所以使⽤了建造者模式，使⽤了

⼤ 量的Builder，进⾏分层构造，核⼼对象Configuration使⽤了 XmlConfigBuilder来进⾏构造



在Mybatis环境的初始化过程中，SqlSessionFactoryBuilder会调⽤XMLConfigBuilder读取所有的

MybatisMapConfig.xml 和所有的 \*Mapper.xml ⽂件，构建 Mybatis 运⾏的核⼼对象 Configuration

对 象，然后将该Configuration对象作为参数构建⼀个SqlSessionFactory对象。

private void parseConfiguration(XNode root) { try {

//issue #117 read properties first

//解析<properties />标签propertiesElement(root.evalNode("properties"));

// 解 析 <settings /> 标 签

Properties settings = settingsAsProperties(root.evalNode("settings"));

//加载⾃定义的VFS实现类loadCustomVfs(settings);

// 解 析 <typeAliases /> 标 签

typeAliasesElement(root.evalNode("typeAliases"));

//解析<plugins />标签pluginElement(root.evalNode("plugins"));

// 解 析 <objectFactory /> 标 签

objectFactoryElement(root.evaINode("obj ectFactory"));

// 解 析 <objectWrapper Factory /> 标 签

obj ectWrappe rFacto ryElement(root.evalNode("objectWrapperFactory"));

// 解 析 <reflectorFactory /> 标 签

reflectorFactoryElement(root.evalNode("reflectorFactory"));

// 赋 值 <settings /> 到 Configuration 属 性

settingsElement(settings);

// read it after objectFactory and objectWrapperFactory issue #631

// 解 析 <environments /> 标 签

environmentsElement(root.evalNode("environments"));

// 解 析 <databaseIdProvider /> 标 签

databaseldProviderElement(root.evalNode("databaseldProvider"));

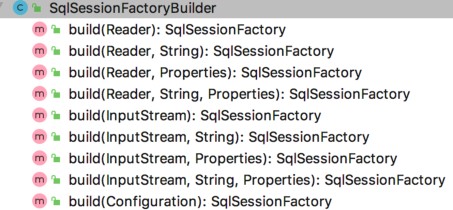
}

其中 XMLConfigBuilder 在构建 Configuration 对象时，也会调⽤ XMLMapperBuilder ⽤于读取

\*Mapper ⽂件，⽽XMLMapperBuilder会使⽤XMLStatementBuilder来读取和build所有的SQL语句。

//解析<mappers />标签mapperElement(root.evalNode("mappers"));

在这个过程中，有⼀个相似的特点，就是这些Builder会读取⽂件或者配置，然后做⼤量的XpathParser 解析、配置或语法的解析、反射⽣成对象、存⼊结果缓存等步骤，这么多的⼯作都不是⼀个构造函数所 能包括的，因此⼤量采⽤了 Builder模式来解决



SqlSessionFactoryBuilder类根据不同的输⼊参数来构建SqlSessionFactory这个⼯⼚对象

* 1. **⼯⼚模式**

在Mybatis中⽐如SqlSessionFactory使⽤的是⼯⼚模式，该⼯⼚没有那么复杂的逻辑，是⼀个简单⼯⼚ 模式。

简单⼯⼚模式(Simple Factory Pattern)：⼜称为静态⼯⼚⽅法(Static Factory Method)模式，它属于创建型模式。

在简单⼯⼚模式中，可以根据参数的不同返回不同类的实例。简单⼯⼚模式专⻔定义⼀个类来负责创建 其他类的实例，被创建的实例通常都具有共同的⽗类

例⼦：⽣产电脑

假设有⼀个电脑的代⼯⽣产商，它⽬前已经可以代⼯⽣产联想电脑了，随着业务的拓展，这个代⼯⽣产 商还要⽣产惠普的电脑，我们就需要⽤⼀个单独的类来专⻔⽣产电脑，这就⽤到了简单⼯⼚模式。

下⾯我们来实现简单⼯⼚模式：

* + 1. 创建抽象产品类

我们创建⼀个电脑的抽象产品类，他有⼀个抽象⽅法⽤于启动电脑：

public abstract class Computer {

/\*\*

\*产品的抽象⽅法，由具体的产品类去实现

\*/

public abstract void start();

}

* + 1. 创建具体产品类

接着我们创建各个品牌的电脑，他们都继承了他们的⽗类Computer，并实现了⽗类的start⽅法：

public class LenovoComputer extends Computer{ @Override

public void start() {

System.out.println("联想电脑启动");

}

public class HpComputer extends Computer{ @Override

public void start() {

System.out.println("惠普电脑启动");

}

}

* + 1. 创建⼯⼚类

接下来创建⼀个⼯⼚类，它提供了⼀个静态⽅法createComputer⽤来⽣产电脑。你只需要传⼊你想⽣ 产的电脑的品牌，它就会实例化相应品牌的电脑对象

import org.junit.runner.Computer;

public class ComputerFactory {

public static Computer createComputer(String type){ Computer mComputer=null;

switch (type) { case "lenovo":

mComputer=new LenovoComputer(); break;

case "hp":

mComputer=new HpComputer(); break;

}

return mComputer;

}

}

客户端调⽤⼯⼚类

客户端调⽤⼯⼚类，传⼊“hp”⽣产出惠普电脑并调⽤该电脑对象的start⽅法：

public class CreatComputer {

public static void main(String[]args){

ComputerFactory.createComputer("hp").start();

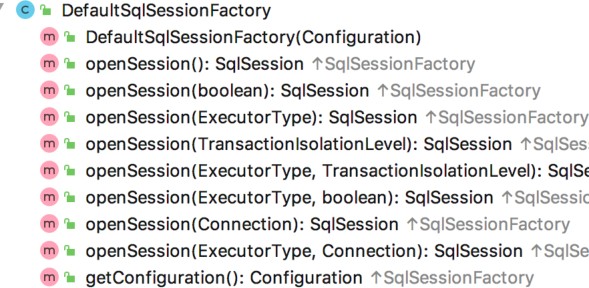
}

}

**Mybatis 体现：**

Mybatis中执⾏Sql语句、获取Mappers、管理事务的核⼼接⼝SqlSession的创建过程使⽤到了⼯⼚模 式。

有⼀个 SqlSessionFactory 来负责 SqlSession 的创建



SqlSessionFactory

可以看到，该Factory的openSession ()⽅法重载了很多个，分别⽀

持**autoCommit**、**Executor**、**Transaction**等参数的输⼊，来构建核⼼的SqlSession对象。在**DefaultSqlSessionFactory**的默认⼯⼚实现⾥，有⼀个⽅法可以看出⼯⼚怎么产出⼀个产品:

private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level,boolean autoCommit){ Transaction tx=null;

try{

final Environment environment=configuration.getEnvironment(); final TransactionFactory transactionFactory=

getTransactionFactoryFromEnvironment(environment);

tx=transactionFactory.newTransaction(environment.getDataSource(),level,autoCo

mmit);

//根据参数创建制定类型的Executor

final Executor executor=configuration.newExecutor(tx,execType);

//返回的是 DefaultSqlSession

return new DefaultSqlSession(configuration,executor,autoCommit);

}catch(Exception e){

closeTransaction(tx); // may have fetched a connection so lets call close()

throw ExceptionFactory.wrapException("Error opening session. Cause: "+ e,e);

}finally{ ErrorContext.instance().reset();

}

}

这是⼀个openSession调⽤的底层⽅法，该⽅法先从configuration读取对应的环境配置，然后初始化TransactionFactory 获得⼀个 Transaction 对象，然后通过 Transaction 获取⼀个 Executor 对象，最后通过configuration、Executor、是否autoCommit三个参数构建了 SqlSession

* 1. **代理模式**

代理模式(Proxy Pattern):给某⼀个对象提供⼀个代理，并由代理对象控制对原对象的引⽤。代理模式的英⽂叫做Proxy，它是⼀种对象结构型模式，代理模式分为静态代理和动态代理，我们来介绍动态代 理

**举例：**

创建⼀个抽象类，Person接⼝，使其拥有⼀个没有返回值的doSomething⽅法。

/\*\*

\* 抽象类⼈

\*/

public interface Person { void doSomething();

}

创建⼀个名为Bob的Person接⼝的实现类，使其实现doSomething⽅法

/\*\*

\* 创建⼀个名为Bob的⼈的实现类

\*/

public class Bob implements Person { public void doSomething() {

System.out.println("Bob doing something!");

}

}

(3) 创建JDK动态代理类，使其实现InvocationHandler接⼝。拥有⼀个名为target的变量，并创建

getTa rget获取代理对象⽅法

/\*\*

* JDK动态代理
* 需实现 InvocationHandler 接⼝ \*/

public class JDKDynamicProxy implements InvocationHandler {

//被代理的对象Person target;

// JDKDynamicProxy 构造函数

public JDKDynamicProxy(Person person) { this.target = person;

}

//获取代理对象

public Person getTarget() { return (Person) Proxy.newProxylnstance(target.getClass().getClassLoader(),

target.getClass().getInterfaces(), this);

}

//动态代理invoke⽅法

public Person invoke(Object proxy, Method method, Object[] args) throws Throwable {

//被代理⽅法前执⾏

System.out.println("JDKDynamicProxy do something before!");

//执⾏被代理的⽅法

Person result = (Person) method.invoke(target, args);

//被代理⽅法后执⾏

System.out.println("JDKDynamicProxy do something after!"); return

result;

}

创建JDK动态代理测试类J DKDynamicTest

/\*\*

\* JDK动态代理测试

\*/

public class JDKDynamicTest {

public static void main(String[] args) {

System.out.println("不使⽤代理类，调⽤doSomething⽅法。");

//不使⽤代理类

Person person = new Bob();

// 调 ⽤ doSomething ⽅ 法

person.doSomething();

System.out.println("分割线 ");

System.out.println("使⽤代理类，调⽤doSomething⽅法。");

//获取代理类

Person proxyPerson = new JDKDynamicProxy(new Bob()).getTarget();

// 调 ⽤ doSomething ⽅ 法 proxyPerson.doSomething();

}

}

**Mybatis中实现：**

代理模式可以认为是Mybatis的核⼼使⽤的模式，正是由于这个模式，我们只需要编写Mapper.java接

⼝，不需要实现，由Mybati s后台帮我们完成具体SQL的执⾏。

当我们使⽤Configuration的getMapper⽅法时，会调⽤mapperRegistry.getMapper⽅法，⽽该⽅法⼜会调⽤ mapperProxyFactory.newInstance(sqlSession)来⽣成⼀个具体的代理：

public class MapperProxyFactory<T> { private final Class<T> mapperInterface;

private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method, MapperMethod>();

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) { 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);

}

}

在这⾥，先通过T newInstance(SqlSession sqlSession)⽅法会得到⼀个MapperProxy对象，然后调⽤T newInstance(MapperProxy mapperProxy)⽣成代理对象然后返回。⽽查看MapperProxy的代码，可以看到如下内容：

public class MapperProxy<T> implements InvocationHandler, Serializable { @Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

try {

if (Object.class.equals(method.getDeclaringClass())) { return method.invoke(this, args);

} else if (isDefaultMethod(method)) {

return invokeDefaultMethod(proxy, method, args);

}

} catch (Throwable t) {

throw ExceptionUtil.unwrapThrowable(t);

}

final MapperMethod mapperMethod = cachedMapperMethod(method); return mapperMethod.execute(sqlSession, args);

}

⾮常典型的，该MapperProxy类实现了InvocationHandler接⼝，并且实现了该接⼝的invoke⽅法。通 过这种⽅式，我们只需要编写Mapper.java接⼝类，当真正执⾏⼀个Mapper接⼝的时候，就会转发给MapperProxy.invoke⽅法，⽽该⽅法则会调⽤后续的sqlSession.cud>executor.execute>prepareStatement 等⼀系列⽅法，完成 SQL 的执⾏和返回

**加餐：Mybatis-Plus**

1. **Mybatis-Plus概念**
   1. **Mybatis-Plus介绍**

官⽹： https://mybatis.plus/ 或 https://mp.baomidou.com/ Mybatis-Plus介绍

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。



**愿景**

我们的愿景是成为 MyBatis 最好的搭档，就像 魂⽃罗 中的 1P、2P，基友搭配，效率翻倍。



* 1. **特性**

**⽆侵⼊**：只做增强不做改变，引⼊它不会对现有⼯程产⽣影响，如丝般顺滑**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

**强⼤的 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

**⽀持 Lambda 形式调⽤**：通过 Lambda 表达式，⽅便的编写各类查询条件，⽆需再担⼼字段写错

**⽀持主键⾃动⽣成**：⽀持多达 4 种主键策略（内含分布式唯⼀ ID ⽣成器 - Sequence），可⾃由配

置，完美解决主键问题

**⽀持 ActiveRecord 模式**：⽀持 ActiveRecord 形式调⽤，实体类只需继承 Model 类即可进⾏强

⼤的 CRUD 操作

**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 ）

**内置代码⽣成器**：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

**内置分⻚插件**：基于 MyBatis 物理分⻚，开发者⽆需关⼼具体操作，配置好插件之后，写分⻚等同于普通 List 查询

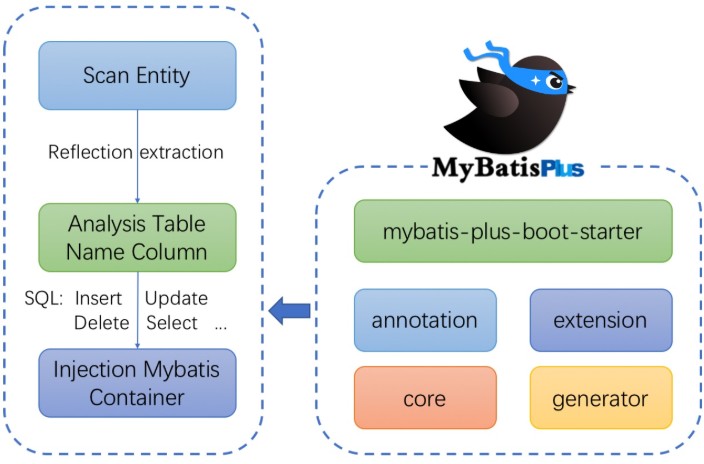
**分⻚插件⽀持多种数据库**：⽀持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、

Postgre、SQLServer 等多种数据库

**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，能快速揪出慢查询

**内置全局拦截插件**：提供全表 delete 、 update 操作智能分析阻断，也可⾃定义拦截规则，预防误操作

* 1. **架构**



* 1. **作者**

Mybatis-Plus是由baomidou（苞⽶⾖）组织开发并且开源的，⽬前该组织⼤概有30⼈左右。码云地址：https://gitee.com/organizations/baomidou



1. **Mybatis-Plus快速⼊⻔**
   1. **安装**

lambda

全新的如下：

MyBatis-Plus

JDK 8+

Maven or Gradle

**Release**

**Spring Boot**

Maven：

3.0 版本基于 JDK8，提供了

形式的调⽤，所以安装集成 MP3.0 要求

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-boot-starter</artifactId>

<version>3.4.0</version>

</dependency>

**Spring MVC**

Maven:

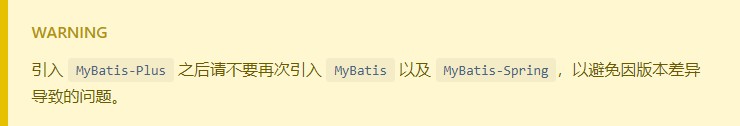
<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus</artifactId>

<version>3.4.0</version>

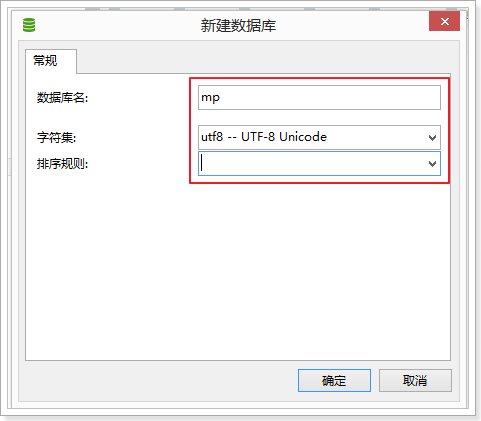
</dependency>



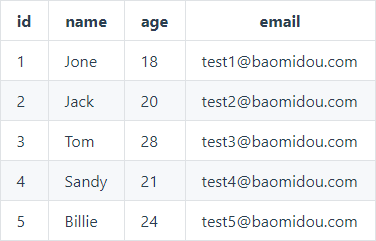
对于Mybatis整合MP有常常有三种⽤法，分别是Mybatis+MP、Spring+Mybatis+MP、Spring

Boot+Mybatis+MP。

* 1. **创建数据库以及表**



创建User表，其表结构如下：



-- 创建测试表

DROP TABLE IF EXISTS tb\_user;

CREATE TABLE user (

id BIGINT(20) NOT NULL COMMENT '主键ID',

name VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名', age INT(11) NULL DEFAULT NULL COMMENT ' 年 龄 ', email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',

PRIMARY KEY (id)

);

-- 插⼊测试数据

INSERT INTO user (id, name, age, email) VALUES (1, 'Jone', 18, 'test1@baomidou.com'),

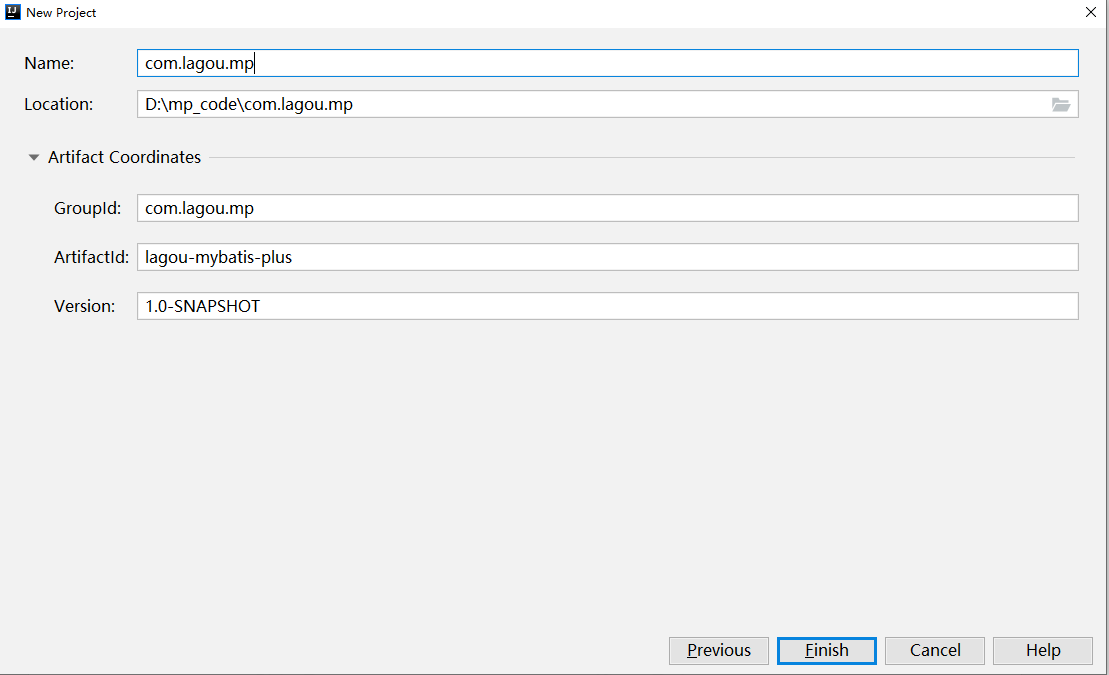
(2, 'Jack', 20, 'test2@baomidou.com'),

(3, 'Tom', 28, 'test3@baomidou.com'),

(4, 'Sandy', 21, 'test4@baomidou.com'),

(5, 'Billie', 24, 'test5@baomidou.com');

* 1. **创建⼯程**



导⼊依赖：

<dependencies>

<!-- mybatis-plus插件依赖 -->

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus</artifactId>

<version>3.1.1</version>

</dependency>

<!--Mysql-->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.47</version>

</dependency>

<!--连接池-->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.0.11</version>

</dependency>

<!--简化bean代码的⼯具包-->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.4</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.4</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

* 1. **Mybatis + MP**

下⾯演示，通过纯Mybatis与Mybatis-Plus整合。

**创建⼦Module**

<?xml version="1.0" encoding="UTF-8" ?>

<project xmlns=["http://maven.apache.org/POM/4.0.0"](http://maven.apache.org/POM/4.0.0) xmlns:xsi=["http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation=["http://maven.apache.org/POM/4.0.0](http://maven.apache.org/POM/4.0.0)

[http://maven.apache.org/xsd/maven-4.0.0.xsd"](http://maven.apache.org/xsd/maven-4.0.0.xsd)>

<parent>

<artifactId>lagou-mybatis-plus</artifactId>

<groupId>com.lagou.mp</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>lagou-mybatis-plus-simple</artifactId>

</project>

log4j.properties：

log4j.rootLogger=DEBUG,A1

log4j.appender.A1=org.apache.log4j.ConsoleAppender log4j.appender.A1.layout=org.apache.log4j.PatternLayout

log4j.appender.A1.layout.ConversionPattern=[%t] [%c]-[%p] %m%n

**Mybatis实现查询User**

**第⼀步，编写mybatis-config.xml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">](http://mybatis.org/dtd/mybatis-3-config.dtd)

<configuration>

<properties resource="jdbc.properties"></properties>

<!--environments: 运⾏环境-->

<environments default="development">

<environment id="development">

<!--当前的事务事务管理器是JDBC-->

<transactionManager type="JDBC"></transactionManager>

<!--数据源信息 POOLED：使⽤mybatis的连接池-->

<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>

<!--引⼊映射配置⽂件-->

<mappers>

<mapper resource="mapper/UserMapper.xml"></mapper>

</mappers>

</configuration>

**第⼆步，编写User实体对象：（这⾥使⽤lombok进⾏了进化bean操作）**

@Data // getter setter @toString @NoArgsConstructor @AllArgsConstructor

public class User {

private Long id; private String name; private Integer age;

private String email;

}

**第三步，编写UserMapper接⼝：**

public interface UserMapper {

List<User> findAll();

}

**第四步，编写UserMapper.xml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">](http://mybatis.org/dtd/mybatis-3-mapper.dtd)

<mapper namespace="com.lagou.mapper.UserMapper">

<!-- 查询所有 -->

<select id="findAll" resultType="com.lagou.pojo.User"> select \* from user

</select>

</mapper>

**第五步，编写TestMybatis测试⽤例：**

public class MPTest {

@Test

public void test1() throws IOException {

InputStream resourceAsStream =

Resources.getResourceAsStream("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);

SqlSession sqlSession = sqlSessionFactory.openSession(); UserMapper mapper = sqlSession.getMapper(UserMapper.class); List<User> all = mapper.findAll();

for (User user : all) { System.out.println(user);

}

}

}

测试结果：

User(id=1, name=Jone, age=18, email=test1@baomidou.com) User(id=2, name=Jack, age=20, email=test2@baomidou.com) User(id=3, name=Tom, age=28, email=test3@baomidou.com) User(id=4, name=Sandy, age=21, email=test4@baomidou.com) User(id=5, name=Billie, age=24, email=test5@baomidou.com)

注：如果实体类名称和表名称不⼀致，可以在实体类上添加注解@TableName("指定数据库表名")

**Mybatis+MP实现查询User**

**第⼀步，将UserMapper继承BaseMapper，将拥有了BaseMapper中的所有⽅法：**

import com.baomidou.mybatisplus.core.mapper.BaseMapper;

import com.lagou.pojo.User;

public interface UserMapper extends BaseMapper<User> {

}

第⼆步，使⽤MP中的MybatisSqlSessionFactoryBuilder进程构建：

@Test

public void test2() throws IOException {

InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");

//这⾥使⽤的是MP中的MybatisSqlSessionFactoryBuilder

SqlSessionFactory sqlSessionFactory = new MybatisSqlSessionFactoryBuilder().build(resourceAsStream);

SqlSession sqlSession = sqlSessionFactory.openSession(); UserMapper mapper = sqlSession.getMapper(UserMapper.class);

// 可以调⽤BaseMapper中定义的⽅法List<User> all = mapper.selectList(null); for (User user : all) {

System.out.println(user);

}

}

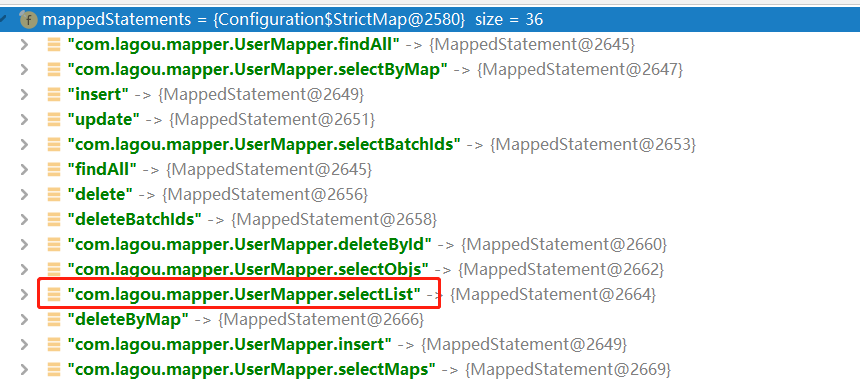
测试：

User(id=1, name=Jone, age=18, email=test1@baomidou.com) User(id=2, name=Jack, age=20, email=test2@baomidou.com) User(id=3, name=Tom, age=28, email=test3@baomidou.com) User(id=4, name=Sandy, age=21, email=test4@baomidou.com) User(id=5, name=Billie, age=24, email=test5@baomidou.com)

注：如果实体类名称和表名称不⼀致，可以在实体类上添加注解@TableName("指定数据库表名") 简单说明：

由于使⽤了 MybatisSqlSessionFactoryBuilder进⾏了构建，继承的BaseMapper中的⽅法就载⼊到了

SqlSession中，所以就可以直接使⽤相关的⽅法； 如图



* 1. **Spring + Mybatis + MP**

引⼊了Spring框架，数据源、构建等⼯作就交给了Spring管理。

**创建⼦Module**

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns=["http://maven.apache.org/POM/4.0.0"](http://maven.apache.org/POM/4.0.0) xmlns:xsi=["http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation=["http://maven.apache.org/POM/4.0.0](http://maven.apache.org/POM/4.0.0)

[http://maven.apache.org/xsd/maven-4.0.0.xsd"](http://maven.apache.org/xsd/maven-4.0.0.xsd)>

<parent>

<artifactId>lagou-mybatis-plus</artifactId>

<groupId>com.lagou.mp</groupId>

<version>1.0-SNAPSHOT</version>

</parent>

<modelVersion>4.0.0</modelVersion>

<artifactId>lagou-mybatis-plus-spring</artifactId>

<properties>

<spring.version>5.1.6.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>${spring.version}</version>

</dependency>

</dependencies>

**实现查询User**

第⼀步，编写jdbc.properties

jdbc.driver=com.mysql.jdbc.Driver jdbc.url=jdbc:mysql://127.0.0.1:3306/mp?serverTimezone=GMT%2B8&useSSL=false jdbc.username=root

jdbc.password=root

第⼆步，编写applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns=["http://www.springframework.org/schema/beans"](http://www.springframework.org/schema/beans) xmlns:xsi=["http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xmlns:context=["http://www.springframework.org/schema/context"](http://www.springframework.org/schema/context) xsi:schemaLocation="

<http://www.springframework.org/schema/beans> <http://www.springframework.org/schema/beans/spring-beans.xsd>

<http://www.springframework.org/schema/context> [http://www.springframework.org/schema/context/spring-context.xsd"](http://www.springframework.org/schema/context/spring-context.xsd)>

<!--引⼊properties-->

<context:property-placeholder location="classpath:jdbc.properties"/>

<!--dataSource-->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">

<property name="driverClassName" value="${jdbc.driver}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<!--这⾥使⽤MP提供的sqlSessionFactory,完成spring与mp的整合-->

<bean id="sqlSessionFactory" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"

>

<property name="dataSource" ref="dataSource"/>

</bean>

<!--扫描mapper接⼝，使⽤的依然是mybatis原⽣的扫描器-->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="basePackage" value="com.lagou.mapper"/>

</bean>

</beans>

第三步，编写User对象以及UserMapper接⼝：

@Data @NoArgsConstructor @AllArgsConstructor

public class User {

private Long id; private String name; private Integer age;

private String email;

}

public interface UserMapper extends BaseMapper<User> {

List<User> findAll();

}

第四步，编写测试⽤例：

@RunWith(SpringJUnit4ClassRunner.class) @ContextConfiguration(locations = "classpath:applicationContext.xml")

public class TestSpringMP {

@Autowired

private UserMapper userMapper;

@Test

public void test2() throws IOException {

List<User> users = this.userMapper.selectList(null); for (User user : users) {

System.out.println(user);

}

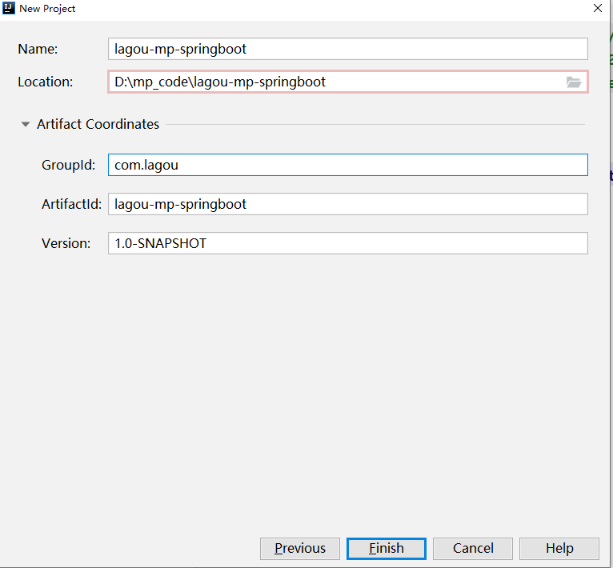
}

* 1. **SpringBoot + Mybatis + MP**

使⽤SpringBoot将进⼀步的简化MP的整合，需要注意的是，由于使⽤SpringBoot需要继承parent，所 以需要重新创

建⼯程，并不是创建⼦Module。

**创建⼯程**



**导⼊依赖**

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<!--简化代码的⼯具包-->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<!--mybatis-plus的springboot⽀持-->

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-boot-starter</artifactId>

<version>3.1.1</version>

</dependency>

<!--mysql驱动-->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.47</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

log4j.properties：

log4j.rootLogger=DEBUG,A1

log4j.appender.A1=org.apache.log4j.ConsoleAppender log4j.appender.A1.layout=org.apache.log4j.PatternLayout

log4j.appender.A1.layout.ConversionPattern=[%t] [%c]-[%p] %m%n

**编写application.properties**

spring.datasource.driver-class-name=com.mysql.jdbc.Driver spring.datasource.url=jdbc:mysql://127.0.0.1:3306/mp? useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=tr ue&useSSL=false

spring.datasource.username=root spring.datasource.password=root

**编写pojo**

@Data @NoArgsConstructor @AllArgsConstructor

public class User {

private Long id; private String name; private Integer age;

private String email;

}

}

**编写mapper**

public interface UserMapper extends BaseMapper<User> {

}

**编写启动类**

package com.lagou.mp;

import org.mybatis.spring.annotation.MapperScan; import org.springframework.boot.SpringApplication; import org.springframework.boot.WebApplicationType;

import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.boot.builder.SpringApplicationBuilder;

@MapperScan("com.lagou.mp.mapper") //设置mapper接⼝的扫描包@SpringBootApplication

public class MyApplication {

public static void main(String[] args) { SpringApplication.run(MyApplication.class, args);

}

}

**编写测试⽤例**

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testSelect() {

List<User> userList = userMapper.selectList(null); for (User user : userList) {

System.out.println(user);

}

}

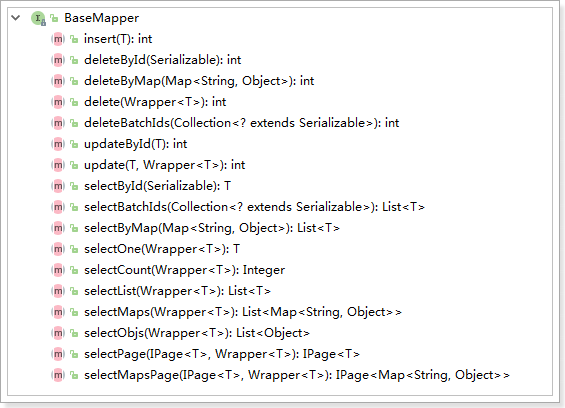
}

测试：

User(id=1, name=Jone, age=18, email=test1@baomidou.com) User(id=2, name=Jack, age=20, email=test2@baomidou.com) User(id=3, name=Tom, age=28, email=test3@baomidou.com) User(id=4, name=Sandy, age=21, email=test4@baomidou.com) User(id=5, name=Billie, age=24, email=test5@baomidou.com)

1. **通⽤CRUD**

通过前⾯的学习，我们了解到通过继承BaseMapper就可以获取到各种各样的单表操作，接下来我们将 详细讲解这些操作。



* 1. **插⼊操作**

**⽅法定义**

/\*\*

* 插⼊⼀条记录

\*

* @param entity 实体对象.

\*/

int insert(T entity);

**测试⽤例**

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testInsert(){ User user = new User(); user.setAge(18);

user.setEmail(["test@lagou.cn"](mailto:test@lagou.cn));

user.setName("⼦慕");

//返回的result是受影响的⾏数，并不是⾃增后的id

int result = userMapper.insert(user);

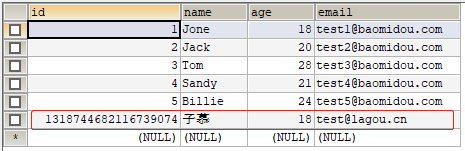
System.out.println(result);

System.out.println(user.getId());

}

}

**测试**



1

1318744682116739074

可以看到，数据已经写⼊到了数据库，但是，id的值不正确，我们期望的是数据库⾃增⻓，实际是MP⽣ 成了id的值写⼊到了数据库。

如何设置id的⽣成策略呢？

MP⽀持的id策略：

package com.baomidou.mybatisplus.annotation;

import lombok.Getter;

/\*\*

* ⽣成ID类型枚举类

\*

* @author hubin

\* @since 2015-11-10

\*/ @Getter

public enum IdType {

/\*\*

* + 数据库ID⾃增

\*/ AUTO(0),

/\*\*

* + 该类型为未设置主键类型

\*/ NONE(1),

/\*\*

* ⽤户输⼊ID
* <p>该类型可以通过⾃⼰注册⾃动填充插件进⾏填充</p>

\*/ INPUT(2),

/\* 以下3种类型、只有当插⼊对象ID 为空，才⾃动填充。 \*/

/\*\*

* 全局唯⼀ID (idWorker)

\*/ ID\_WORKER(3),

/\*\*

* 全局唯⼀ID (UUID)

\*/ UUID(4),

/\*\*

* 字符串全局唯⼀ID (idWorker 的字符串表示)

\*/

ID\_WORKER\_STR(5);

private final int key;

IdType(int key) { this.key = key;

}

}

修改User对象：

package com.lagou.mp.pojo;

import com.baomidou.mybatisplus.annotation.IdType; import com.baomidou.mybatisplus.annotation.TableId; import com.baomidou.mybatisplus.annotation.TableName; import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data @NoArgsConstructor @AllArgsConstructor @TableName("tb\_user") public class User {

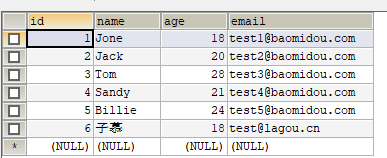
@TableId(type = IdType.AUTO) //指定id类型为⾃增⻓private Long id;

private String userName; private String password; private String name; private Integer age;

private String email;

}

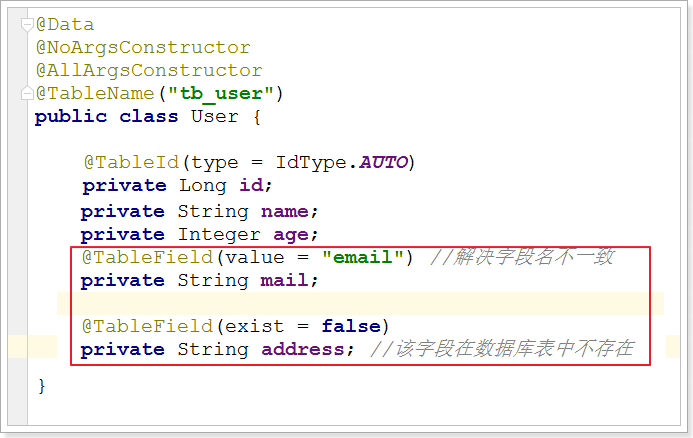
数据插⼊成功：



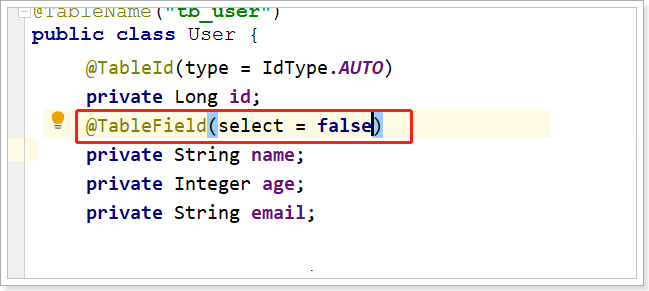
**@TableField**

在MP中通过@TableField注解可以指定字段的⼀些属性，常常解决的问题有2个：

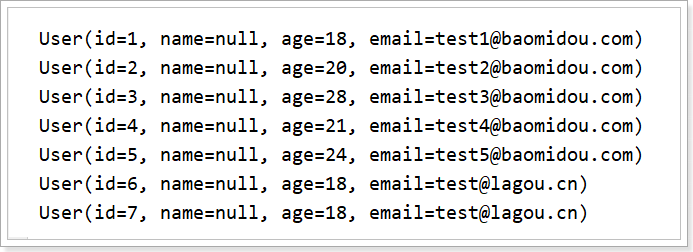
1、对象中的属性名和字段名不⼀致的问题（⾮驼峰）

2、对象中的属性字段在表中不存在的问题使⽤：

其他⽤法，如⼤字段不加⼊查询字段：



效果：



* 1. **更新操作**

在MP中，更新操作有2种，⼀种是根据id更新，另⼀种是根据条件更新。

**根据id更新**

⽅法定义：

/\*\*

* 根 据 ID 修 改

\*

* @param entity 实体对象

\*/

int updateById(@Param(Constants.ENTITY) T entity);

测试：

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

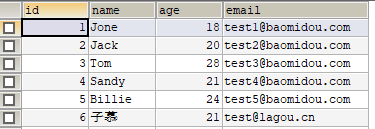
public void testUpdateById() { User user = new User(); user.setId(6L); //主键user.setAge(21); //更新的字段

//根据id更新，更新不为null的字段this.userMapper.updateById(user);

}

}

结果：



**根据条件更新**

⽅法定义：

/\*\*

* 根据 whereEntity 条件，更新记录

\*

* @param entity 实体对象 (set 条件值,可以为 null)
* @param updateWrapper 实体对象封装操作类（可以为 null,⾥⾯的 entity ⽤于⽣成

where 语句）

\*/

int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER) Wrapper<T> updateWrapper);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.Wrapper;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;

import com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper; import net.minidev.json.writer.UpdaterMapper;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testUpdate() { User user = new User();

user.setAge(22); //更新的字段

//更新的条件

QueryWrapper<User> wrapper = new QueryWrapper<>(); wrapper.eq("id", 6);

//执⾏更新操作

int result = this.userMapper.update(user, wrapper); System.out.println("result = " + result);

}

}

或者，通过UpdateWrapper进⾏更新：

@Test

public void testUpdate() {

//更新的条件以及字段

UpdateWrapper<User> wrapper = new UpdateWrapper<>(); wrapper.eq("id", 6).set("age", 23);

//执⾏更新操作

int result = this.userMapper.update(null, wrapper); System.out.println("result = " + result);

}

测试结果：

[main] [com.lagou.mp.mapper.UserMapper.update]-[DEBUG] ==> Preparing: UPDATE tb\_user SET age=? WHERE id = ?

[main] [com.lagou.mp.mapper.UserMapper.update]-[DEBUG] ==> Parameters: 23(Integer), 6(Integer)

[main] [com.lagou.mp.mapper.UserMapper.update]-[DEBUG] <== Updates: 1

均可达到更新的效果。

关于wrapper更多的⽤法后⾯会详细讲解。

* 1. **删除操作**

**deleteById**

⽅法定义：

/\*\*

* 根 据 ID 删 除

\*

* @param id 主键ID

\*/

int deleteById(Serializable id);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testDeleteById() {

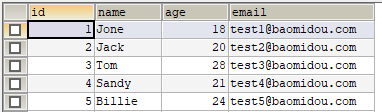
//执⾏删除操作

int result = this.userMapper.deleteById(6L); System.out.println("result = " + result);

}

}

结果：



[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] ==> Preparing: DELETE FROM tb\_user WHERE id=?

[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] ==> Parameters: 6(Long)

[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] <== Updates: 1

数据被删除。

**deleteByMap**

⽅法定义：

/\*\*

* 根据 columnMap 条件，删除记录

\*

* @param columnMap 表字段 map 对象

\*/

int deleteByMap(@Param(Constants.COLUMN\_MAP) Map<String, Object> columnMap);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.HashMap;

import java.util.Map;

@RunWith(SpringRunner.class)

@SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testDeleteByMap() {

Map<String, Object> columnMap = new HashMap<>(); columnMap.put("age",21); columnMap.put("name","⼦慕");

//将columnMap中的元素设置为删除的条件，多个之间为and关系 int result = this.userMapper.deleteByMap(columnMap); System.out.println("result = " + result);

}

}

结果：

[main] [com.lagou.mp.mapper.UserMapper.deleteByMap]-[DEBUG] ==> Preparing: DELETE FROM tb\_user WHERE name = ? AND age = ?

[main] [com.lagou.mp.mapper.UserMapper.deleteByMap]-[DEBUG] ==> Parameters: ⼦

慕 (String), 21(Integer)

[main] [com.lagou.mp.mapper.UserMapper.deleteByMap]-[DEBUG] <== Updates: 0

**delete**

⽅法定义：

/\*\*

* 根据 entity 条件，删除记录

\*

* @param wrapper 实体对象封装操作类（可以为 null）

\*/

int delete(@Param(Constants.WRAPPER) Wrapper<T> wrapper);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.HashMap;

import java.util.Map;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testDeleteByMap() { User user = new User(); user.setAge(20);

user.setName("⼦慕");

//将实体对象进⾏包装，包装为操作条件

QueryWrapper<User> wrapper = new QueryWrapper<>(user);

int result = this.userMapper.delete(wrapper);

System.out.println("result = " + result);

}

}

结果：

[main] [com.lagou.mp.mapper.UserMapper.delete]-[DEBUG] ==> Preparing: DELETE FROM tb\_user WHERE name=? AND age=?

[main] [com.lagou.mp.mapper.UserMapper.delete]-[DEBUG] ==> Parameters: ⼦ 慕

(String), 20(Integer)

[main] [com.lagou.mp.mapper.UserMapper.delete]-[DEBUG] <== Updates: 0

**3.3.4、deleteBatchIds**

⽅法定义：

/\*\*

* 删除（根据ID 批量删除）

\*

* @param idList 主键ID列表(不能为 null 以及 empty)

\*/

int deleteBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.Arrays;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testDeleteByMap() {

//根据id集合批量删除int result =

this.userMapper.deleteBatchIds(Arrays.asList(1L,10L,20L)); System.out.println("result = " + result);

}

}

结果：

[main] [com.lagou.mp.mapper.UserMapper.deleteBatchIds]-[DEBUG] ==> Preparing: DELETE FROM tb\_user WHERE id IN ( ? , ? , ? )

[main] [com.lagou.mp.mapper.UserMapper.deleteBatchIds]-[DEBUG] ==> Parameters: 1(Long), 10(Long), 20(Long)

[main] [com.lagou.mp.mapper.UserMapper.deleteBatchIds]-[DEBUG] <== Updates: 1

* 1. **查询操作**

MP提供了多种查询操作，包括根据id查询、批量查询、查询单条数据、查询列表、分⻚查询等操作。

* + 1. **、selectById**

⽅法定义：

/\*\*

* 根 据 ID 查 询

\*

* @param id 主键ID

\*/

T selectById(Serializable id);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testSelectById() {

//根据id查询数据

User user = this.userMapper.selectById(2L); System.out.println("result = " + user);

}

}

结果：

result = User(id=2, name=Jack, age=20, email=test2@baomidou.com)

* + 1. **、selectBatchIds**

⽅法定义：

/\*\*

* 查询（根据ID 批量查询）

\*

* @param idList 主键ID列表(不能为 null 以及 empty)

\*/

List<T> selectBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.Arrays;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testSelectBatchIds() {

//根据id集合批量查询

List<User> users = this.userMapper.selectBatchIds(Arrays.asList(2L, 3L, 10L));

for (User user : users) { System.out.println(user);

}

}

}

结果：

User(id=2, name=Jack, age=20, email=test2@baomidou.com) User(id=3, name=Tom, age=28, email=test3@baomidou.com)

* + 1. **、selectOne**

⽅法定义：

/\*\*

* 根据 entity 条件，查询⼀条记录

\*

* @param queryWrapper 实体对象封装操作类（可以为 null）

\*/

T selectOne(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testSelectOne() {

QueryWrapper<User> wrapper = new QueryWrapper<User>(); wrapper.eq("name", "jack");

//根据条件查询⼀条数据，如果结果超过⼀条会报错

User user = this.userMapper.selectOne(wrapper); System.out.println(user);

}

}

结果：

User(id=2, name=Jack, age=20, email=test2@baomidou.com)

* + 1. **、selectCount**

⽅法定义：

/\*\*

* 根据 Wrapper 条件，查询总记录数

\*

* @param queryWrapper 实体对象封装操作类（可以为 null）

\*/

Integer selectCount(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testSelectCount() {

QueryWrapper<User> wrapper = new QueryWrapper<User>(); wrapper.gt("age", 23); //年龄⼤于23岁

//根据条件查询数据条数

Integer count = this.userMapper.selectCount(wrapper); System.out.println("count = " + count);

}

}

结果：

count = 2

* + 1. **、selectList**

⽅法定义：

/\*\*

* 根据 entity 条件，查询全部记录

\*

* @param queryWrapper 实体对象封装操作类（可以为 null）

\*/

List<T> selectList(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testSelectList() {

QueryWrapper<User> wrapper = new QueryWrapper<User>(); wrapper.gt("age", 23); //年龄⼤于23岁

//根据条件查询数据

List<User> users = this.userMapper.selectList(wrapper); for (User user : users) {

System.out.println("user = " + user);

}

}

}

结果：

user = User(id=3, name=Tom, age=28, email=test3@baomidou.com) user = User(id=5, name=Billie, age=24, email=test5@baomidou.com)

* + 1. **、selectPage**

⽅法定义：

/\*\*

* 根据 entity 条件，查询全部记录（并翻⻚）

\*

* @param page 分⻚查询条件（可以为 RowBounds.DEFAULT）
* @param queryWrapper 实体对象封装操作类（可以为 null）

\*/

IPage<T> selectPage(IPage<T> page, @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

配置分⻚插件：

package com.lagou.mp;

import com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor; import org.mybatis.spring.annotation.MapperScan;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

@MapperScan("com.lagou.mp.mapper") //设置mapper接⼝的扫描包public class MybatisPlusConfig {

/\*\*

\* 分⻚插件

\*/ @Bean

public PaginationInterceptor paginationInterceptor() { return new PaginationInterceptor();

}

}

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import com.baomidou.mybatisplus.core.metadata.IPage;

import com.baomidou.mybatisplus.extension.plugins.pagination.Page; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testSelectPage() {

QueryWrapper<User> wrapper = new QueryWrapper<User>(); wrapper.gt("age", 20); //年龄⼤于20岁

Page<User> page = new Page<>(1,1);

//根据条件查询数据

IPage<User> iPage = this.userMapper.selectPage(page, wrapper);

System.out.println("数据总条数：" + iPage.getTotal());

System.out.println("总⻚数：" + iPage.getPages());

List<User> users = iPage.getRecords(); for (User user : users) {

System.out.println("user = " + user);

}

}

}

结果：

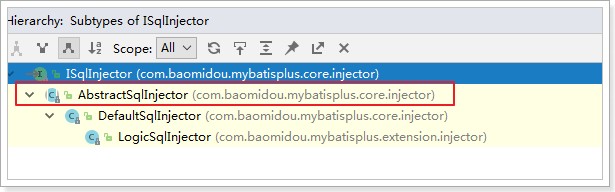
数据总条数：4 总⻚数：4

user = User(id=3, name=Tom, age=28, email=test3@baomidou.com)

* 1. **SQL注⼊的原理**

前⾯我们已经知道，MP在启动后会将BaseMapper中的⼀系列的⽅法注册到meppedStatements中， 那么究竟是如何注⼊的呢？流程⼜是怎么样的？下⾯我们将⼀起来分析下。

在MP中，ISqlInjector负责SQL的注⼊⼯作，它是⼀个接⼝，AbstractSqlInjector是它的实现类，实现关系如下：



在AbstractSqlInjector中，主要是由inspectInject()⽅法进⾏注⼊的，如下：

@Override

public void inspectInject(MapperBuilderAssistant builderAssistant, Class<?> mapperClass) {

Class<?> modelClass = extractModelClass(mapperClass); if (modelClass != null) {

String className = mapperClass.toString(); Set<String> mapperRegistryCache =

GlobalConfigUtils.getMapperRegistryCache(builderAssistant.getConfiguration()); if (!mapperRegistryCache.contains(className)) {

List<AbstractMethod> methodList = this.getMethodList(); if (CollectionUtils.isNotEmpty(methodList)) {

TableInfo tableInfo = TableInfoHelper.initTableInfo(builderAssistant, modelClass);

// 循环注⼊⾃定义⽅法

methodList.forEach(m -> m.inject(builderAssistant, mapperClass, modelClass, tableInfo));

} else {

logger.debug(mapperClass.toString() + ", No effective injection method was found.");

}

mapperRegistryCache.add(className);

}

}

}

在实现⽅法中， methodList.forEach(m -> m.inject(builderAssistant, mapperClass, modelClass, tableInfo)); 是关键，循环遍历⽅法，进⾏注⼊。

最终调⽤抽象⽅法injectMappedStatement进⾏真正的注⼊：

/\*\*

* 注⼊⾃定义 MappedStatement

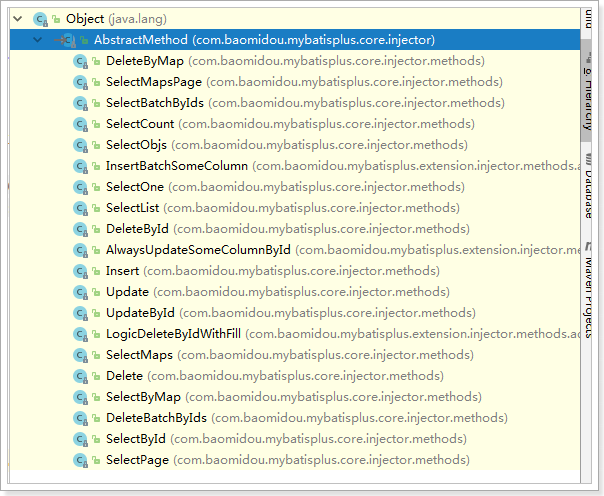
\*

* @param mapperClass mapper 接 ⼝
* @param modelClass mapper 泛 型
* @param tableInfo 数据库表反射信息
* @return MappedStatement

\*/

public abstract MappedStatement injectMappedStatement(Class<?> mapperClass, Class<?> modelClass, TableInfo tableInfo);

查看该⽅法的实现：



以SelectById为例查看：

public class SelectById extends AbstractMethod {

@Override

public MappedStatement injectMappedStatement(Class<?> mapperClass, Class<?

> modelClass, TableInfo tableInfo) {

SqlMethod sqlMethod = SqlMethod.LOGIC\_SELECT\_BY\_ID; SqlSource sqlSource = new RawSqlSource(configuration,

String.format(sqlMethod.getSql(),

sqlSelectColumns(tableInfo, false), tableInfo.getTableName(), tableInfo.getKeyColumn(),

tableInfo.getKeyProperty(),

tableInfo.getLogicDeleteSql(true, false)), Object.class); return this.addSelectMappedStatement(mapperClass,

sqlMethod.getMethod(), sqlSource, modelClass, tableInfo);

}

}

可以看到，⽣成了SqlSource对象，再将SQL通过addSelectMappedStatement⽅法添加到

meppedStatements中。



1. **配置**

在MP中有⼤量的配置，其中有⼀部分是Mybatis原⽣的配置，另⼀部分是MP的配置，详情：https://m ybatis.plus/config/

下⾯我们对常⽤的配置做讲解。

* 1. **、基本配置**
     1. **、configLocation**

MyBatis 配置⽂件位置，如果有单独的 MyBatis 配置，请将其路径配置到 configLocation 中。

MyBatis Configuration 的具体内容请参考MyBatis 官⽅⽂档

Spring Boot：

mybatis-plus.config-location = classpath:mybatis-config.xml

Spring MVC：

<bean id="sqlSessionFactory" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"

>

<property name="configLocation" value="classpath:mybatis-config.xml"/>

</bean>

* + 1. **、mapperLocations**

MyBatis Mapper 所对应的 XML ⽂件位置，如果您在 Mapper 中有⾃定义⽅法（XML 中有⾃定义实现），需要进⾏该配置，告诉 Mapper 所对应的 XML ⽂件位置。

Spring Boot：

mybatis-plus.mapper-locations = classpath\*:mybatis/\*.xml

Spring MVC：

<bean id="sqlSessionFactory" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"

>

<property name="mapperLocations" value="classpath\*:mybatis/\*.xml"/>

</bean>

**Maven 多模块项⽬的扫描路径需以 开头 （即加载多个 jar 包下的 XML ⽂件）**

**classpath\*:**

测试：

UserMapper.xml：

<?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">](http://mybatis.org/dtd/mybatis-3-mapper.dtd)

<mapper namespace="com.lagou.mp.mapper.UserMapper">

<select id="findById" resultType="com.lagou.mp.pojo.User"> select \* from tb\_user where id = #{id}

</select>

</mapper>

package com.lagou.mp.mapper;

import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;

public interface UserMapper extends BaseMapper<User> {

User findById(Long id);

}

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

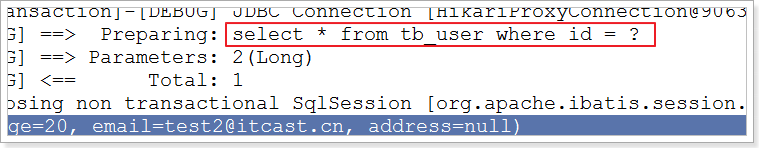
public void testSelectPage() {

User user = this.userMapper.findById(2L); System.out.println(user);

}

}

运⾏结果：



* + 1. **、typeAliasesPackage**

MyBaits 别名包扫描路径，通过该属性可以给包中的类注册别名，注册后在 Mapper 对应的 XML ⽂件中可以直接使⽤类名，⽽不⽤使⽤全限定的类名（即 XML 中调⽤的时候不⽤包含包名）。

Spring Boot：

mybatis-plus.type-aliases-package = com.lagou.mp.pojo

Spring MVC：

<bean id="sqlSessionFactory" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"

>

<property name="typeAliasesPackage" value="com.baomidou.mybatisplus.samples.quickstart.entity"/>

</bean>

* 1. **、进阶配置**

本部分（Configuration）的配置⼤都为 MyBatis 原⽣⽀持的配置，这意味着您可以通过 MyBatis XML

配置⽂件的形式进⾏配置。

* + 1. **、mapUnderscoreToCamelCase**

类型： boolean

默认值： true

是否开启⾃动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN（下划线命名） 到经典 Java 属性名 aColumn（驼峰命名） 的类似映射。

注意：

此属性在 MyBatis 中原默认值为 false，在 MyBatis-Plus 中，此属性也将⽤于⽣成最终的 SQL 的

select body

如果您的数据库命名符合规则⽆需使⽤示例（SpringBoot）：

@TableField

注解指定数据库字段名

#关闭⾃动驼峰映射，该参数不能和mybatis-plus.config-location同时存在mybatis-plus.configuration.map-underscore-to-camel-case=false

* + 1. **、cacheEnabled**

类型： boolean

默认值： true

全局地开启或关闭配置⽂件中的所有映射器已经配置的任何缓存，默认为 true。示例：

mybatis-plus.configuration.cache-enabled=false

* 1. **、DB 策略配置**
     1. **、idType**

类型： com.baomidou.mybatisplus.annotation.IdType

默认值： ID\_WORKER

全局默认主键类型，设置后，即可省略实体对象中的@TableId(type = IdType.AUTO)配置。示例：

SpringBoot：

mybatis-plus.global-config.db-config.id-type=auto

SpringMVC：

<!--这⾥使⽤MP提供的sqlSessionFactory，完成了Spring与MP的整合-->

<bean id="sqlSessionFactory" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"

>

<property name="dataSource" ref="dataSource"/>

<property name="globalConfig">

<bean class="com.baomidou.mybatisplus.core.config.GlobalConfig">

<property name="dbConfig">

<bean class="com.baomidou.mybatisplus.core.config.GlobalConfig$DbConfig">

<property name="idType" value="AUTO"/>

</bean>

</property>

</bean>

</property>

</bean>

* + 1. **、tablePrefix**

类型： String

默认值： null

表名前缀，全局配置后可省略@TableName()配置。

SpringBoot：

mybatis-plus.global-config.db-config.table-prefix=tb\_

SpringMVC：

<bean id="sqlSessionFactory" class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean"

>

<property name="dataSource" ref="dataSource"/>

<property name="globalConfig">

<bean class="com.baomidou.mybatisplus.core.config.GlobalConfig">

<property name="dbConfig">

<bean class="com.baomidou.mybatisplus.core.config.GlobalConfig$DbConfig">

<property name="idType" value="AUTO"/>

<property name="tablePrefix" value="tb\_"/>

</bean>

</property>

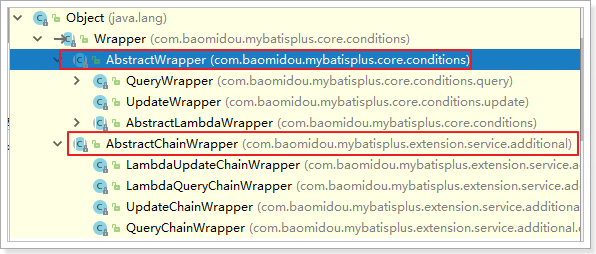
</bean>

</property>

</bean>

1. **条件构造器**

在MP中，Wrapper接⼝的实现类关系如下：



可以看到，AbstractWrapper和AbstractChainWrapper是重点实现，接下来我们重点学习

AbstractWrapper以及其⼦类。

说 明 : QueryWrapper(LambdaQueryWrapper) 和 UpdateWrapper(LambdaUpdateWrapper) 的 ⽗ 类

⽤于⽣成 sql 的 where 条件, entity 属性也⽤于⽣成 sql 的 where 条件

注意: entity ⽣成的 where 条件与 使⽤各个 api ⽣成的 where 条件**没有任何关联⾏为**

官⽹⽂档地址：https://mybatis.plus/guide/wrapper.html

* 1. **、allEq**
     1. **、说明**

allEq(Map<R, V> params)

allEq(Map<R, V> params, boolean null2IsNull)

allEq(boolean condition, Map<R, V> params, boolean null2IsNull)

全部eq(或个别isNull)

个别参数说明:

params : key 为数据库字段名, value 为字段值

null2IsNull : 为 true 则在 map 的 value 为 null 时调⽤ isNull ⽅法,为 false 时则忽略 value 为 null 的

例1: allEq({id:1,name:"⽼王",age:null}) ---> id = 1 and name = '⽼王' and age is null

例2: allEq({id:1,name:"⽼王",age:null}, false) ---> id = 1 and name = '⽼王'

allEq(BiPredicate<R, V> filter, Map<R, V> params)

allEq(BiPredicate<R, V> filter, Map<R, V> params, boolean null2IsNull) allEq(boolean condition, BiPredicate<R, V> filter, Map<R, V> params, boolean null2IsNull)

个别参数说明:

filter : 过滤函数,是否允许字段传⼊⽐对条件中

null2IsNull

params 与

: 同上

例1: allEq((k,v) -> k.indexOf("a") > 0, {id:1,name:"⽼王",age:null}) ---

> name = '⽼王' and age is null

例2: allEq((k,v) -> k.indexOf("a") > 0, {id:1,name:"⽼王",age:null}, false) ---> name = '⽼王'

### 、测试⽤例

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.HashMap; import java.util.List;

import java.util.Map;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testWrapper() {

QueryWrapper<User> wrapper = new QueryWrapper<>();

//设置条件

Map<String,Object> params = new HashMap<>(); params.put("name", "jack");

params.put("age", "20");

//

wrapper.allEq(params);//SELECT \* FROM tb\_user WHERE password IS NULL

AND name = ? AND age = ?

//

wrapper.allEq(params,false); //SELECT \* FROM tb\_user WHERE name = ?

AND age = ?

//

wrapper.allEq((k, v) -> (k.equals("name") || k.equals("age"))

,params);//SELECT \* FROM tb\_user WHERE name = ? AND age = ?

List<User> users = this.userMapper.selectList(wrapper); for (User user : users) {

System.out.println(user);

}

}

}

* 1. **、基本⽐较操作**

eq

等于 =

ne

不等于 <>

gt

⼤于 >

ge

⼤于等于 >=

lt

⼩于 <

le

⼩于等于 <= between

BETWEEN 值1 AND 值2

notBetween

NOT BETWEEN 值1 AND 值2

in

字段 IN (value.get(0), value.get(1), ...)

notIn

字段 NOT IN (v0, v1, ...)

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

QueryWrapper<User> wrapper = new QueryWrapper<>();

//SELECT id,name,age,email FROM tb\_user WHERE password = ? AND age >=

? AND name IN (?,?,?)

wrapper.eq("email", ["test2@baomidou.com"](mailto:test2@baomidou.com))

.ge("age", 20)

.in("name", "jack", "jone", "tom");

List<User> users = this.userMapper.selectList(wrapper); for (User user : users) {

System.out.println(user);

}

}

}

## 、模糊查询

like

LIKE '%值%'

例: like("name", "王") ---> name like '%王%'

notLike

NOT LIKE '%值%'

例: notLike("name", "王") ---> name not like '%王%'

likeLeft

LIKE '%值'

例: likeLeft("name", "王") ---> name like '%王'

likeRight

LIKE '值%'

例: likeRight("name", "王") ---> name like '王%'

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testWrapper() {

QueryWrapper<User> wrapper = new QueryWrapper<>();

//SELECT id,user\_name,password,name,age,email FROM tb\_user WHERE name

LIKE ?

//Parameters: %⼦%(String)

wrapper.like("name", "⼦");

List<User> users = this.userMapper.selectList(wrapper);

for (User user : users) { System.out.println(user);

}

}

}

* 1. **、排序**

orderBy

排序：ORDER BY 字段, ...

例 : orderBy(true, true, "id", "name") ---> order by id ASC,name ASC

orderByAsc

排序：ORDER BY 字段, ... ASC

例: orderByAsc("id", "name") ---> order by id ASC,name ASC

orderByDesc

排序：ORDER BY 字段, ... DESC

例: orderByDesc("id", "name") ---> order by id DESC,name DESC

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testWrapper() {

QueryWrapper<User> wrapper = new QueryWrapper<>();

//SELECT id,user\_name,password,name,age,email FROM tb\_user ORDER BY age DESC

wrapper.orderByDesc("age");

List<User> users = this.userMapper.selectList(wrapper); for (User user : users) {

System.out.println(user);

}

}

}

* 1. **、逻辑查询**

or

and

拼接 OR

主动调⽤ or 表示紧接着下⼀个**⽅法**不是⽤ and 连接!(不调⽤ or 则默认为使⽤ and 连接)

AND 嵌套

例: and(i -> i.eq("name", "李⽩").ne("status", "活着")) ---> and (name = '李

⽩' and status <> '活着')

测试⽤例：

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testWrapper() {

QueryWrapper<User> wrapper = new QueryWrapper<>();

//SELECT id,user\_name,password,name,age,email FROM tb\_user WHERE name

= ? OR age = ?

wrapper.eq("name","jack").or().eq("age", 24);

List<User> users = this.userMapper.selectList(wrapper); for (User user : users) {

System.out.println(user);

}

}

}

* 1. **、select**

在MP查询中，默认查询所有的字段，如果有需要也可以通过select⽅法进⾏指定字段。

package com.lagou.mp;

import com.lagou.mp.mapper.UserMapper; import com.lagou.mp.pojo.User;

import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testWrapper() {

QueryWrapper<User> wrapper = new QueryWrapper<>();

//SELECT id,name,age FROM tb\_user WHERE name = ? OR age = ? wrapper.eq("name", "jack")

.or()

.eq("age", 24)

.select("id", "name", "age");

List<User> users = this.userMapper.selectList(wrapper);

for (User user : users) { System.out.println(user);

}

}

}

1. **ActiveRecord**

ActiveRecord（简称AR）⼀直⼴受动态语⾔（ PHP 、 Ruby 等）的喜爱，⽽ Java 作为准静态语⾔，对于 ActiveRecord 往往只能感叹其优雅，所以我们也在 AR 道路上进⾏了⼀定的探索，希望⼤家能够喜欢。

**什么是ActiveRecord？**

ActiveRecord也属于ORM（对象关系映射）层，由Rails最早提出，遵循标准的ORM模型：表映 射到记录，记录映射到对象，字段映射到对象属性。配合遵循的命名和配置惯例，能够很⼤程度 的快速实现模型的操作，⽽且简洁易懂。

ActiveRecord的主要思想是：

每⼀个数据库表对应创建⼀个类，类的每⼀个对象实例对应于数据库中表的⼀⾏记录；通常 表 的 每 个 字 段 在 类 中 都 有 相 应 的 Field； ActiveRecord同时负责把⾃⼰持久化，在ActiveRecord中封装了对数据库的访问，即CURD;；

ActiveRecord是⼀种领域模型(Domain Model)，封装了部分业务逻辑；

* 1. **、开启AR之旅**

在MP中，开启AR⾮常简单，只需要将实体对象继承Model即可。

package com.lagou.mp.pojo;

import com.baomidou.mybatisplus.annotation.IdType; import com.baomidou.mybatisplus.annotation.TableField; import com.baomidou.mybatisplus.annotation.TableId; import com.baomidou.mybatisplus.annotation.TableName;

import com.baomidou.mybatisplus.extension.activerecord.Model; import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data @NoArgsConstructor @AllArgsConstructor

public class User extends Model<User> {

private Long id;

private String userName; private String password; private String name; private Integer age; private String email;

}

* 1. **、根据主键查询**

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testAR() { User user = new User(); user.setId(2L);

User user2 = user.selectById();

System.out.println(user2);

}

}

* 1. **、新增数据**

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testARInsert() { User user = new User();

user.setName("应颠");

user.setAge(30);

user.setEmail(["yingdian@lagou.cn"](mailto:yingdian@lagou.cn));

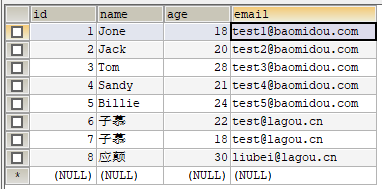
boolean insert = user.insert();

System.out.println(insert);

}

}

结果：



[main] [com.lagou.mp.mapper.UserMapper.insert]-[DEBUG] ==> Preparing: INSERT INTO tb\_user ( name, age, email ) VALUES ( ?, ?, ?, ?, ? )

[main] [com.lagou.mp.mapper.UserMapper.insert]-[DEBUG] ==> Parameters: 应癫

(String), 30(Integer), liubei@lagou.cn(String)

[main] [com.lagou.mp.mapper.UserMapper.insert]-[DEBUG] <== Updates: 1

* 1. **、更新操作**

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testAR() { User user = new User(); user.setId(8L); user.setAge(35);

boolean update = user.updateById();

System.out.println(update);

}

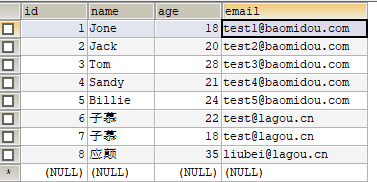
}

结果：

[main] [com.lagou.mp.mapper.UserMapper.updateById]-[DEBUG] ==> Preparing: UPDATE tb\_user SET age=? WHERE id=?

[main] [com.lagou.mp.mapper.UserMapper.updateById]-[DEBUG] ==> Parameters: 35(Integer), 8(Long)

[main] [com.lagou.mp.mapper.UserMapper.updateById]-[DEBUG] <== Updates: 1



* 1. **、删除操作**

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testAR() { User user = new User(); user.setId(7L);

boolean delete = user.deleteById();

System.out.println(delete);

}

}

结果：

[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] ==> Preparing: DELETE FROM tb\_user WHERE id=?

[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] ==> Parameters: 7(Long)

[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] <== Updates: 1

* 1. **、根据条件查询**

@RunWith(SpringRunner.class) @SpringBootTest

public class UserMapperTest {

@Autowired

private UserMapper userMapper;

@Test

public void testARFindById() { User user = new User();

QueryWrapper<User> userQueryWrapper = new QueryWrapper<>();

userQueryWrapper.le("age","20");

List<User> users = user.selectList(userQueryWrapper); for (User user1 : users) {

System.out.println(user1);

}

}

}

结果：

User(id=1, name=Jone, age=18, email=test1@baomidou.com) User(id=2, name=Jack, age=20, email=test2@baomidou.com) User(id=7, name=⼦慕, age=18, email=test@lagou.cn)

1. **插件**
   1. **、mybatis的插件机制**

MyBatis 允许你在已映射语句执⾏过程中的某⼀点进⾏拦截调⽤。默认情况下，MyBatis 允许使⽤插件来拦截的⽅法调⽤包括：

1. Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
2. ParameterHandler (getParameterObject, setParameters)
3. ResultSetHandler (handleResultSets, handleOutputParameters)
4. StatementHandler (prepare, parameterize, batch, update, query)

我们看到了可以拦截Executor接⼝的部分⽅法，⽐如update，query，commit，rollback等⽅法，还有 其他接⼝的⼀些⽅法等。

总体概括为：

1. 拦截执⾏器的⽅法
2. 拦截参数的处理
3. 拦截结果集的处理
4. 拦截Sql语法构建的处理

拦截器示例：

package com.lagou.mp.plugins;

import org.apache.ibatis.executor.Executor; import org.apache.ibatis.mapping.MappedStatement;

import org.apache.ibatis.plugin.\*;

import java.util.Properties;

@Intercepts({@Signature(

type= Executor.class, method = "update",

args = {MappedStatement.class,Object.class})})

public class MyInterceptor implements Interceptor {

@Override

public Object intercept(Invocation invocation) throws Throwable {

//拦截⽅法，具体业务逻辑编写的位置return invocation.proceed();

}

@Override

public Object plugin(Object target) {

//创建target对象的代理对象,⽬的是将当前拦截器加⼊到该对象中return Plugin.wrap(target, this);

}

@Override

public void setProperties(Properties properties) {

//属性设置

}

}

注⼊到Spring容器：

/\*\*

\* ⾃定义拦截器

\*/ @Bean

public MyInterceptor myInterceptor(){ return new MyInterceptor();

}

或者通过xml配置，mybatis-config.xml：

<?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">](http://mybatis.org/dtd/mybatis-3-config.dtd)

<configuration>

<plugins>

<plugin interceptor="com.lagou.mp.plugins.MyInterceptor"></plugin>

</plugins>

</configuration>

* 1. **、执⾏分析插件**

在MP中提供了对SQL执⾏的分析的插件，可⽤作阻断全表更新、删除的操作，注意：该插件仅适⽤于开 发环境，不适⽤于⽣产环境。

SpringBoot配置：

@Bean

public SqlExplainInterceptor sqlExplainInterceptor(){

SqlExplainInterceptor sqlExplainInterceptor = new SqlExplainInterceptor();

List<ISqlParser> sqlParserList = new ArrayList<>();

// 攻击 SQL 阻断解析器、加⼊解析链

sqlParserList.add(new BlockAttackSqlParser());

sqlExplainInterceptor.setSqlParserList(sqlParserList);

return sqlExplainInterceptor;

}

测试：

@Test

public void testUpdate(){ User user = new User(); user.setAge(20);

int result = this.userMapper.update(user, null);

System.out.println("result = " + result);

}

结果：

Caused by: com.baomidou.mybatisplus.core.exceptions.MybatisPlusException:

Prohibition of table update operation at

com.baomidou.mybatisplus.core.toolkit.ExceptionUtils.mpe(ExceptionUtils.java:4 9)

at com.baomidou.mybatisplus.core.toolkit.Assert.isTrue(Assert.java:38) at com.baomidou.mybatisplus.core.toolkit.Assert.notNull(Assert.java:72) at

com.baomidou.mybatisplus.extension.parsers.BlockAttackSqlParser.processUpdate( BlockAttackSqlParser.java:45)

at com.baomidou.mybatisplus.core.parser.AbstractJsqlParser.processParser(Abstract JsqlParser.java:92)

at com.baomidou.mybatisplus.core.parser.AbstractJsqlParser.parser(AbstractJsqlPar ser.java:67)

at com.baomidou.mybatisplus.extension.handlers.AbstractSqlParserHandler.sqlParser (AbstractSqlParserHandler.java:76)

at com.baomidou.mybatisplus.extension.plugins.SqlExplainInterceptor.intercept(Sql ExplainInterceptor.java:63)

at org.apache.ibatis.plugin.Plugin.invoke(Plugin.java:61) at com.sun.proxy.$Proxy70.update(Unknown Source)

at org.apache.ibatis.session.defaults.DefaultSqlSession.update(DefaultSqlSession. java:197)

... 41 more

可以看到，当执⾏全表更新时，会抛出异常，这样有效防⽌了⼀些误操作。

* 1. **、性能分析插件**

性能分析拦截器，⽤于输出每条 SQL 语句及其执⾏时间，可以设置最⼤执⾏时间，超过时间会抛出异常。

**该插件只⽤于开发环境，不建议⽣产环境使⽤。**

配置：

javaconfig⽅式

@Bean

public PerformanceInterceptor performanceInterceptor(){ PerformanceInterceptor performanceInterceptor = new

PerformanceInterceptor(); performanceInterceptor.setMaxTime(100);

performanceInterceptor.setFormat(true);

return performanceInterceptor;

}

xml⽅式

<?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">](http://mybatis.org/dtd/mybatis-3-config.dtd)

<configuration>

<plugins>

<!-- SQL 执⾏性能分析，开发环境使⽤，线上不推荐。 maxTime 指的是 sql 最⼤执⾏时

⻓ -->

<plugin interceptor="com.baomidou.mybatisplus.extension.plugins.PerformanceInterceptor ">

<property name="maxTime" value="100" />

<!--SQL是否格式化 默认false-->

<property name="format" value="true" />

</plugin>

</plugins>

</configuration>

执⾏结果：

Time：11 ms - ID：com.lagou.mp.mapper.UserMapper.selectById Execute SQL：

SELECT

id, user\_name, password, name,

age, email

FROM

tb\_user WHERE

id=7

可以看到，执⾏时间为11ms。如果将maxTime设置为1，那么，该操作会抛出异常。

Caused by: com.baomidou.mybatisplus.core.exceptions.MybatisPlusException: The SQL execution time is too large, please optimize !

at com.baomidou.mybatisplus.core.toolkit.ExceptionUtils.mpe(ExceptionUtils.java:4 9)

at com.baomidou.mybatisplus.core.toolkit.Assert.isTrue(Assert.java:38)

................

* 1. **、乐观锁插件**
     1. **、主要适⽤场景**

意图：

当要更新⼀条记录的时候，希望这条记录没有被别⼈更新 乐观锁实现⽅式：

取出记录时，获取当前version 更新时，带上这个version

执⾏更新时， set version = newVersion where version = oldVersion

如果version不对，就更新失败

* + 1. **、插件配置**

spring xml:

<bean class="com.baomidou.mybatisplus.extension.plugins.OptimisticLockerInterceptor"

/>

spring boot:

@Bean

public OptimisticLockerInterceptor optimisticLockerInterceptor() { return new OptimisticLockerInterceptor();

}

* + 1. **、注解实体字段**

需要为实体字段添加@Version注解。

第⼀步，为表添加version字段，并且设置初始值为1：

ALTER TABLE `tb\_user`

ADD COLUMN `version` int(10) NULL AFTER `email`;

UPDATE `tb\_user` SET `version`='1';

第⼆步，为User实体对象添加version字段，并且添加@Version注解：

@Version

private Integer version;

* + 1. **、测试**

测试⽤例：

@Test

public void testUpdate(){ User user = new User(); user.setAge(30); user.setId(2L);

user.setVersion(1); //获取到version为1

int result = this.userMapper.updateById(user);

System.out.println("result = " + result);

}

执⾏⽇志：

main] [com.baomidou.mybatisplus.extension.parsers.BlockAttackSqlParser]-

[DEBUG] Original SQL: UPDATE tb\_user SET age=?,

version=? WHERE id=? AND version=?

[main] [com.baomidou.mybatisplus.extension.parsers.BlockAttackSqlParser]- [DEBUG] parser sql: UPDATE tb\_user SET age = ?, version = ? WHERE id = ? AND version = ?

[main] [org.springframework.jdbc.datasource.DataSourceUtils]-[DEBUG] Fetching JDBC Connection from DataSource

[main] [org.mybatis.spring.transaction.SpringManagedTransaction]-[DEBUG] JDBC Connection [HikariProxyConnection@540206885 wrapping com.mysql.jdbc.JDBC4Connection@27e0f2f5] will not be managed by Spring

[main] [com.lagou.mp.mapper.UserMapper.updateById]-[DEBUG] ==> Preparing: UPDATE tb\_user SET age=?, version=? WHERE id=? AND version=?

[main] [com.lagou.mp.mapper.UserMapper.updateById]-[DEBUG] ==> Parameters: 30(Integer), 2(Integer), 2(Long), 1(Integer)

[main] [com.lagou.mp.mapper.UserMapper.updateById]-[DEBUG] <== Updates: 1 [main] [org.mybatis.spring.SqlSessionUtils]-[DEBUG] Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@30135202]

result = 1

可以看到，更新的条件中有version条件，并且更新的version为2。

如果再次执⾏，更新则不成功。这样就避免了多⼈同时更新时导致数据的不⼀致。

* + 1. **、特别说明**

**⽀持的数据类型只有:int,Integer,long,Long,Date,Timestamp,LocalDateTime**

整数类型下

newVersion = oldVersion + 1

仅⽀持

会回写到 中

与 ⽅法

newVersion

entity

updateById(id)

update(entity, wrapper)

**在 update(entity, wrapper) ⽅法下, wrapper 不能复⽤!!!**

1. **Sql 注⼊器**

我们已经知道，在MP中，通过AbstractSqlInjector将BaseMapper中的⽅法注⼊到了Mybatis容器，这样这些⽅法才可以正常执⾏。

那么，如果我们需要扩充BaseMapper中的⽅法，⼜该如何实现呢？ 下⾯我们以扩展findAll⽅法为例进⾏学习。

* 1. **、编写MyBaseMapper**

package com.lagou.mp.mapper;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;

import java.util.List;

public interface MyBaseMapper<T> extends BaseMapper<T> {

List<T> findAll();

}

其他的Mapper都可以继承该Mapper，这样实现了统⼀的扩展。如：

package com.lagou.mp.mapper;

import com.lagou.mp.pojo.User;

public interface UserMapper extends MyBaseMapper<User> {

User findById(Long id);

}

* 1. **、编写MySqlInjector**

如果直接继承AbstractSqlInjector的话，原有的BaseMapper中的⽅法将失效，所以我们选择继承

DefaultSqlInjector进⾏扩展。

package com.lagou.mp.sqlInjector;

import com.baomidou.mybatisplus.core.injector.AbstractMethod;

import com.baomidou.mybatisplus.core.injector.DefaultSqlInjector;

import java.util.List;

public class MySqlInjector extends DefaultSqlInjector {

@Override

public List<AbstractMethod> getMethodList() { List<AbstractMethod> methodList = super.getMethodList();

methodList.add(new FindAll());

// 再扩充⾃定义的⽅法

list.add(new FindAll());

return methodList;

}

}

* 1. **、编写FindAll**

package com.lagou.mp.sqlInjector;

import com.baomidou.mybatisplus.core.enums.SqlMethod;

import com.baomidou.mybatisplus.core.injector.AbstractMethod; import com.baomidou.mybatisplus.core.metadata.TableInfo; import org.apache.ibatis.mapping.MappedStatement;

import org.apache.ibatis.mapping.SqlSource;

public class FindAll extends AbstractMethod {

@Override

public MappedStatement injectMappedStatement(Class<?> mapperClass, Class<?

> modelClass, TableInfo tableInfo) { String sqlMethod = "findAll";

String sql = "select \* from " + tableInfo.getTableName();

SqlSource sqlSource = languageDriver.createSqlSource(configuration, sql, modelClass);

return this.addSelectMappedStatement(mapperClass, sqlMethod, sqlSource, modelClass, tableInfo);

}

}

## 、注册到Spring容器

/\*\*

\* ⾃定义SQL注⼊器

\*/ @Bean

public MySqlInjector mySqlInjector(){ return new MySqlInjector();

}

* 1. **、测试**

@Test

public void testFindAll(){

List<User> users = this.userMapper.findAll(); for (User user : users) {

System.out.println(user);

}

}

输出的SQL：

[main] [com.lagou.mp.mapper.UserMapper.findAll]-[DEBUG] ==> Preparing: select

\* from tb\_user

[main] [com.lagou.mp.mapper.UserMapper.findAll]-[DEBUG] ==> Parameters: [main] [com.lagou.mp.mapper.UserMapper.findAll]-[DEBUG] <== Total: 10

⾄此，我们实现了全局扩展SQL注⼊器。

1. **⾃动填充功能**

有些时候我们可能会有这样的需求，插⼊或者更新数据时，希望有些字段可以⾃动填充数据，⽐如密 码、version等。在MP中提供了这样的功能，可以实现⾃动填充。

* 1. **、添加@TableField注解**

@TableField(fill = FieldFill.INSERT) //插⼊数据时进⾏填充private String version;

为email添加⾃动填充功能，在新增数据时有效。

FieldFill提供了多种模式选择：

public enum FieldFill {

/\*\*

* 默认不处理

\*/ DEFAULT,

/\*\*

* 插⼊时填充字段

\*/ INSERT,

/\*\*

* 更新时填充字段

\*/ UPDATE,

/\*\*

* 插⼊和更新时填充字段

\*/

INSERT\_UPDATE

}

* 1. **、编写MyMetaObjectHandler**

package com.lagou.mp.handler;

import com.baomidou.mybatisplus.core.handlers.MetaObjectHandler; import org.apache.ibatis.reflection.MetaObject;

import org.springframework.stereotype.Component;

@Component

public class MyMetaObjectHandler implements MetaObjectHandler {

@Override

public void insertFill(MetaObject metaObject) {

Object password = getFieldValByName("version", metaObject); if(null == password){

//字段为空，可以进⾏填充

setFieldValByName("version", "123456", metaObject);

}

}

@Override

public void updateFill(MetaObject metaObject) {

}

}

* 1. **、测试**

@Test

public void testInsert(){ User user = new User();

user.setName("冰冰");

user.setAge(30);

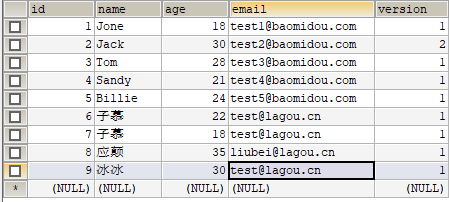
user.setVersion(1);

int result = this.userMapper.insert(user);

System.out.println("result = " + result);

}

结果：



1. **逻辑删除**

开发系统时，有时候在实现功能时，删除操作需要实现逻辑删除，所谓逻辑删除就是将数据标记为删 除，⽽并⾮真正的物理删除（⾮DELETE操作），查询时需要携带状态条件，确保被标记的数据不被查 询到。这样做的⽬的就是避免数据被真正的删除。

MP就提供了这样的功能，⽅便我们使⽤，接下来我们⼀起学习下。

* 1. **、修改表结构**

为tb\_user表增加deleted字段，⽤于表示数据是否被删除，1代表删除，0代表未删除。

ALTER TABLE `tb\_user`

ADD COLUMN `deleted` int(1) NULL DEFAULT 0 COMMENT '1代表删除，0代表未删除' AFTER `version`;

同时，也修改User实体，增加deleted属性并且添加@TableLogic注解：

@TableLogic

private Integer deleted;

* 1. **、配置**

application.properties：

# 逻辑已删除值(默认为 1)

mybatis-plus.global-config.db-config.logic-delete-value=1 # 逻辑未删除值(默认为 0)

mybatis-plus.global-config.db-config.logic-not-delete-value=0

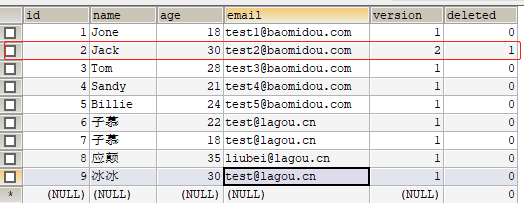
* 1. **、测试**

@Test

public void testDeleteById(){ this.userMapper.deleteById(2L);

}

执⾏的SQL：



[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] ==> Preparing:

UPDATE tb\_user SET deleted=1 WHERE id=? AND deleted=0

[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] ==> Parameters: 2(Long)

[main] [com.lagou.mp.mapper.UserMapper.deleteById]-[DEBUG] <== Updates: 1

测试查询：

@Test

public void testSelectById(){

User user = this.userMapper.selectById(2L); System.out.println(user);

}

执⾏的SQL：

[main] [com.lagou.mp.mapper.UserMapper.selectById]-[DEBUG] ==> Preparing: SELECT id,user\_name,password,name,age,email,version,deleted FROM tb\_user WHERE id=? AND deleted=0

[main] [com.lagou.mp.mapper.UserMapper.selectById]-[DEBUG] ==> Parameters: 2(Long)

[main] [com.lagou.mp.mapper.UserMapper.selectById]-[DEBUG] <== Total: 0

可⻅，已经实现了逻辑删除。

###

1. **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。

* 1. **、创建⼯程**

pom.xml：

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns=["http://maven.apache.org/POM/4.0.0"](http://maven.apache.org/POM/4.0.0) xmlns:xsi=["http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance)

xsi:schemaLocation=["http://maven.apache.org/POM/4.0.0](http://maven.apache.org/POM/4.0.0) https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.3.4.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.lagou</groupId>

<artifactId>lagou-mp-generator</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>lagou-mp-generator</name>

<description>Demo project for Spring Boot</description>

<properties>

<java.version>11</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<!--mybatis-plus的springboot⽀持-->

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-boot-starter</artifactId>

<version>3.1.1</version>

</dependency>

<dependency>

<groupId>com.baomidou</groupId>

<artifactId>mybatis-plus-generator</artifactId>

<version>3.1.1</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-freemarker</artifactId>

</dependency>

<!--mysql驱动-->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.47</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!--简化代码的⼯具包-->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

## 、代码

package com.lagou.mp.generator;

import java.util.ArrayList; import java.util.List; import java.util.Scanner;

import com.baomidou.mybatisplus.core.exceptions.MybatisPlusException; import com.baomidou.mybatisplus.core.toolkit.StringPool;

import com.baomidou.mybatisplus.core.toolkit.StringUtils; import com.baomidou.mybatisplus.generator.AutoGenerator; import com.baomidou.mybatisplus.generator.InjectionConfig;

import com.baomidou.mybatisplus.generator.config.DataSourceConfig; import com.baomidou.mybatisplus.generator.config.FileOutConfig; import com.baomidou.mybatisplus.generator.config.GlobalConfig; import com.baomidou.mybatisplus.generator.config.PackageConfig; import com.baomidou.mybatisplus.generator.config.StrategyConfig; import com.baomidou.mybatisplus.generator.config.TemplateConfig; import com.baomidou.mybatisplus.generator.config.po.TableInfo;

import com.baomidou.mybatisplus.generator.config.rules.NamingStrategy; import com.baomidou.mybatisplus.generator.engine.FreemarkerTemplateEngine;

/\*\*

\* <p>

\* mysql 代码⽣成器演示例⼦

\* </p>

\*/

public class MysqlGenerator {

/\*\*

\* <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 + "！");

}

/\*\*

\* RUN THIS

\*/

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("lagou");

gc.setOpen(false); mpg.setGlobalConfig(gc);

// 数据源配置

DataSourceConfig dsc = new DataSourceConfig(); dsc.setUrl("jdbc:mysql://127.0.0.1:3306/mp?

useUnicode=true&useSSL=false&characterEncoding=utf8");

// dsc.setSchemaName("public"); dsc.setDriverName("com.mysql.jdbc.Driver"); dsc.setUsername("root"); dsc.setPassword("root"); mpg.setDataSource(dsc);

// 包配置

PackageConfig pc = new PackageConfig(); pc.setModuleName(scanner(" 模 块 名 ")); pc.setParent("com.lagou.mp.generator");

mpg.setPackageInfo(pc);

// ⾃定义配置

InjectionConfig cfg = new InjectionConfig() { @Override

public void initMap() {

// to do nothing

}

};

List<FileOutConfig> focList = new ArrayList<>(); focList.add(new FileOutConfig("/templates/mapper.xml.ftl") {

@Override

public String outputFile(TableInfo tableInfo) {

// ⾃定义输⼊⽂件名称

return projectPath + "/lagou-mp- generator/src/main/resources/mapper/" + pc.getModuleName()

+ "/" + tableInfo.getEntityName() + "Mapper" +

StringPool.DOT\_XML;

}

});

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

mpg.setTemplate(new TemplateConfig().setXml(null));

// 策略配置

StrategyConfig strategy = new StrategyConfig(); strategy.setNaming(NamingStrategy.underline\_to\_camel); strategy.setColumnNaming(NamingStrategy.underline\_to\_camel);

//

strategy.setSuperEntityClass("com.baomidou.mybatisplus.samples.generator.commo n.BaseEntity");

strategy.setEntityLombokModel(true);

// strategy.setSuperControllerClass("com.baomidou.mybatisplus.samples.generator.c ommon.BaseController");

strategy.setInclude(scanner("表名"));

strategy.setSuperEntityColumns("id"); strategy.setControllerMappingHyphenStyle(true); strategy.setTablePrefix(pc.getModuleName() + "\_"); mpg.setStrategy(strategy);

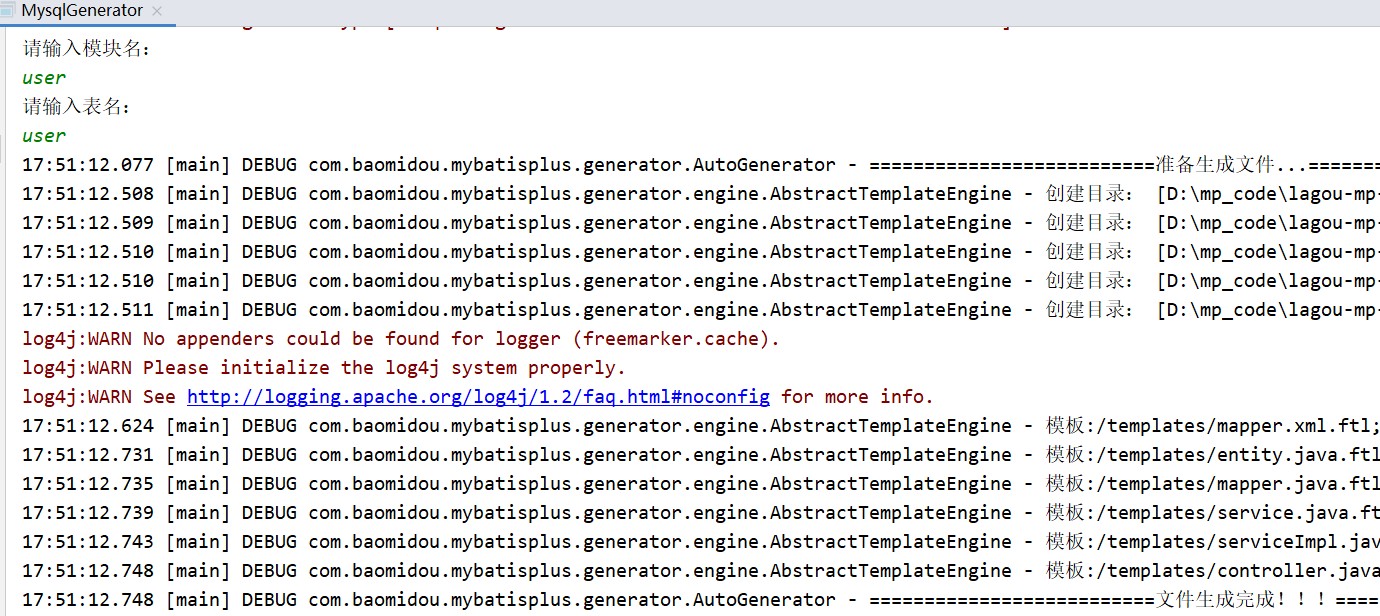
// 选择 freemarker 引擎需要指定如下加，注意 pom 依赖必须有！

mpg.setTemplateEngine(new FreemarkerTemplateEngine()); mpg.execute();

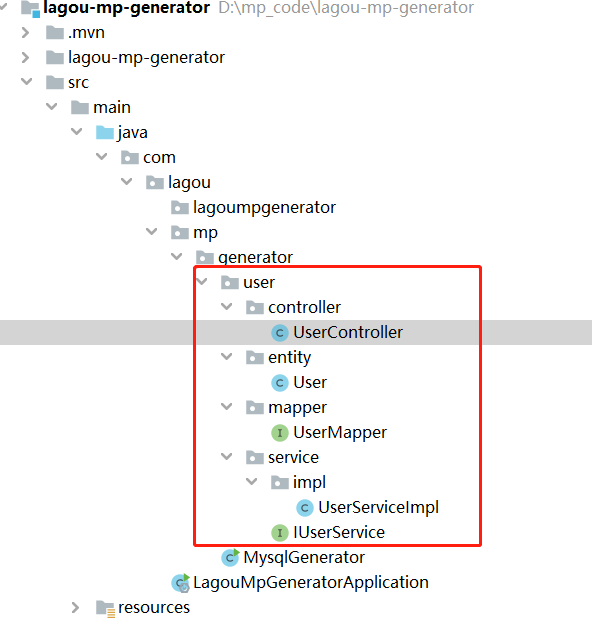
}

}

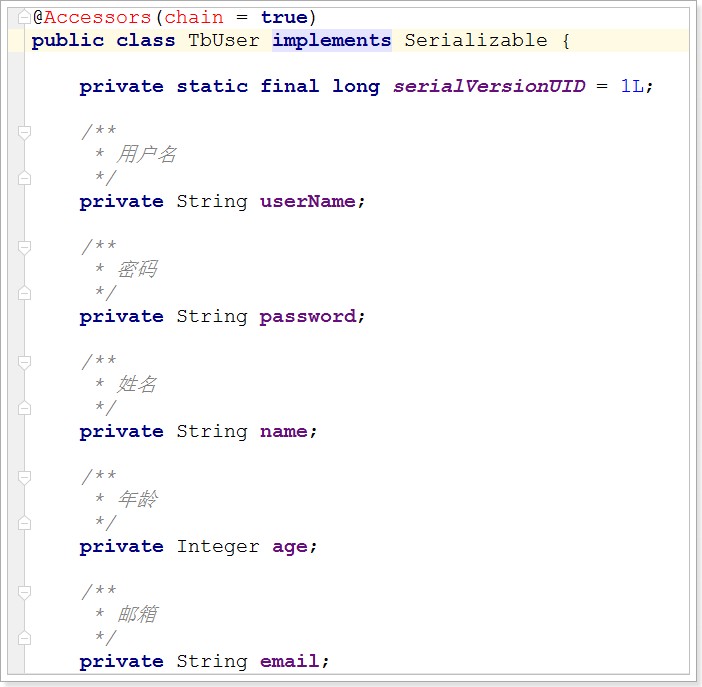
## 、测试



代码已⽣成：



实体对象：



1. **MybatisX 快速开发插件**

MybatisX 是⼀款基于 IDEA 的快速开发插件，为效率⽽⽣。

安装⽅法：打开 IDEA，进⼊ File -> Settings -> Plugins -> Browse Repositories，输⼊ mybatisx 搜索并安装。

功能：

Java 与 XML 调回跳转

Mapper ⽅法⾃动⽣成 XML

