Mybatis

第一部分：自定义持久层框架

1.1 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

1.2 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

1.3 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

（2）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

3.创建SqlSessionFactory：

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

4.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

1.4 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();

        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
数class
            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;
    }
}
```

1.5 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.setId(1);
        user.setUsername("tom");
        //代理对象
        UserMapper userMapper = sqlSession.getMappper(UserMapper.class);
        User user1 = userMapper.selectOne(user);
        System.out.println(user1);
    }
```

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

2.1 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

2.2 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。

2.3 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)

2.4 Mybatis优势

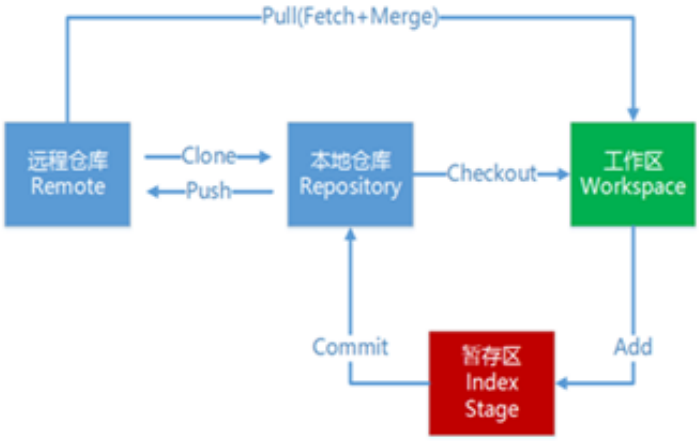
Mybatis是一个半自动化的持久层框架，对开发人员开说，核心sql还是需要自己进行优化，sql和java编码进行分离，功能边界清晰，一个专注业务，一个专注数据。

分析图示如下：

第三部分：Mybatis基本应用

3.1 快速入门

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>
```

2) 创建user数据表

名	类型	长度	小数点	允许空值 (	
id	int	11	0	☐	🔑1
username	varchar	50	0	☑	
password	varchar	50	0	☑	

3) 编写User实体

```
public class User {
  private int id;
  private String username;
  private String password;
    //省略get个set方法
}
```

4)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">
<mapper namespace="userMapper">
  <select id="findAll" resultType="com.lagou.domain.User">
    select * from User
  </select>
</mapper>
```

5) 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">
<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();
```

3.1.4 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>
```

2)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();
```

3)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()

3.1.5 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>
```

2)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();
```

3)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);

3.1.6 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>
```

2)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();
```

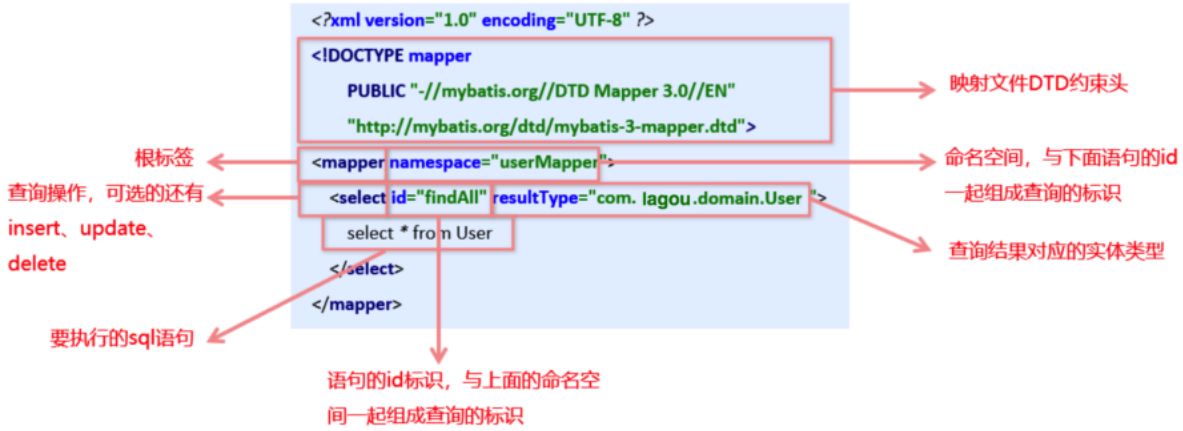
3)删除操作注意问题

• 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);

3.1.5 MyBatis的映射文件概述



3.1.6 入门核心配置文件分析：

MyBatis核心配置文件层级关系

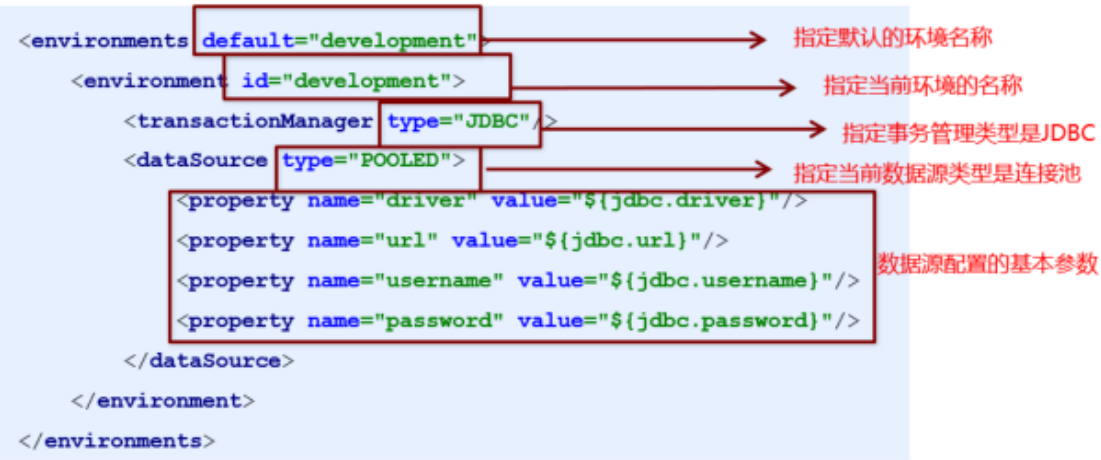
MyBatis核心配置文件层级关系

- configuration 配置
  - properties 属性
  - settings 设置
  - typeAliases 类型别名
  - typeHandlers 类型处理器
  - objectFactory 对象工厂
  - plugins 插件
  - environments 环境
    - environment 环境变量
      - transactionManager 事务管理器
      - dataSource 数据源
  - databaseIdProvider 数据库厂商标识
  - mappers 映射器

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




```
<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>
```

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。

2)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"/>
```

3.1.7 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：

方法	解释
openSession()	会默认开启一个事务，但事务不会自动提交，也就意味着需要手动提交该事务，更新操作数据才会持久化到数据库中
openSession(boolean autoCommit)	参数为是否自动提交，如果设置为true，那么不需要手动提交事务

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()
```

3.2 Mybatis的Dao层实现

3.2.1 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);
}
```

3.2.2 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：

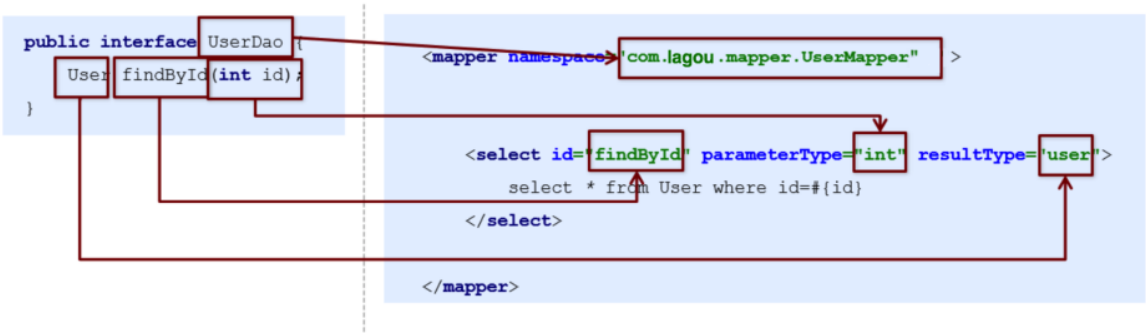
1) Mapper.xml文件中的namespace与mapper接口的全限定名相同

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

4.1 核心配置文件SqlMapConfig.xml

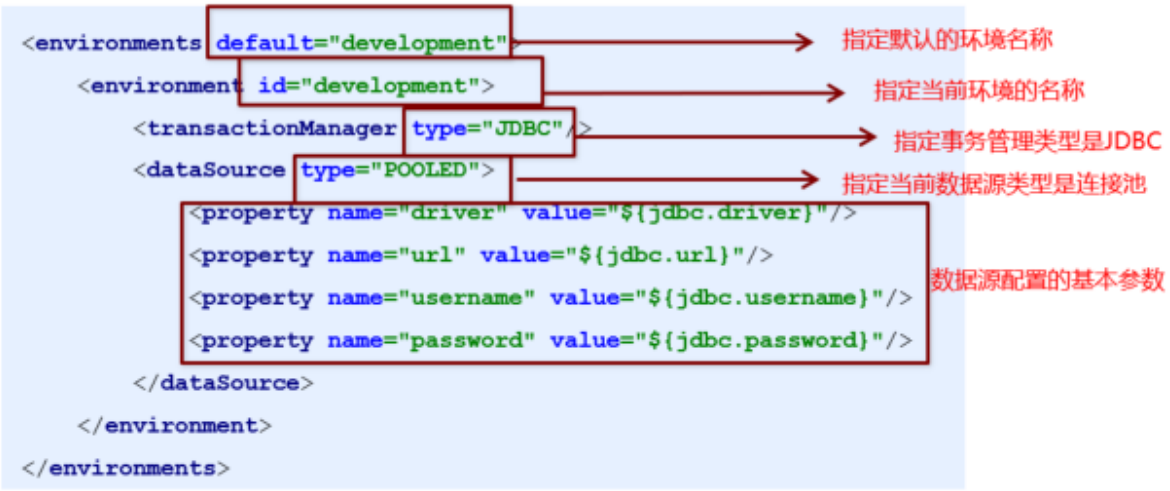
4.1.1 MyBatis核心配置文件层级关系

- configuration 配置
  - properties 属性
  - settings 设置
  - typeAliases 类型别名
  - typeHandlers 类型处理器
  - objectFactory 对象工厂
  - plugins 插件
  - environments 环境
    - environment 环境变量
      - transactionManager 事务管理器
      - dataSource 数据源
  - databaseIdProvider 数据库厂商标识
  - mappers 映射器

4.2 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




```
<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>
```

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。

2)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"/>
```

3)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

4)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

```
<typeAliases>
  <typeAlias type="com.lagou.domain.User"  alias="user"></typeAlias>
</typeAliases>
```

```
<select id="findAll" resultType="user">
  select * from User
</select>
```

user为别名

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

别名	数据类型
string	String
long	Long
int	Integer
double	Double
boolean	Boolean
... ...	... ...

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：

Dynamic SQL

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

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

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

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

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);
    … … …
```

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

当查询条件只有id存在时，控制台打印的sql语句如下：

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

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

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);
… … …
```

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

foreach标签的属性含义如下：

标签用于遍历集合，它的属性：

•collection：代表要遍历的集合元素，注意编写时不要写#{}

•open：代表语句的开始部分

•close：代表结束部分

•item：代表遍历集合的每个元素，生成的变量名

•sperator：代表分隔符

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>
```

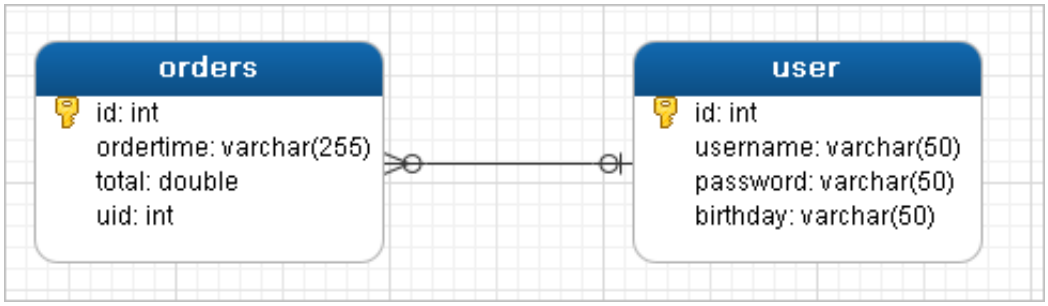
第五部分：Mybatis复杂映射开发

5.1 一对一查询

5.1.1 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



5.1.2一对一查询的语句

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

查询的结果如下：

信息 | 结果1 | 概况 | 状态

id	ordertime	total	uid	id1	username	password	birthday
1	2018-12-12	3000	1	1	lucy	123	1539751863457
2	2019-12-12	4000	1	1	lucy	123	1539751863457
3	2020-12-12	5000	2	2	tom	123	1539751863457

5.1.3 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;

}
```

5.1.4 创建OrderMapper接口

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

5.1.5 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>
```

5.1.6 测试结果

```
OrderMapper mapper = sqlSession.getMapper(OrderMapper.class);
List<Order> all = mapper.findAll();
for(Order order : all){
    System.out.println(order);
}
```

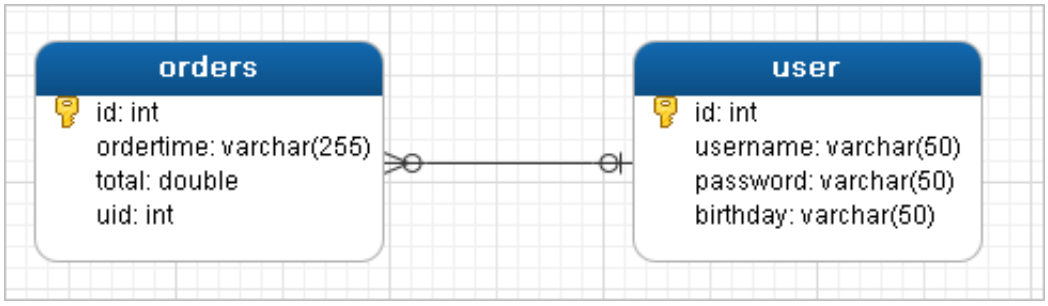
```
09:12:24,650 DEBUG findAll:54 - ==>  Preparing: select * from orders o,user u where o.uid=u.id
09:12:24,672 DEBUG findAll:54 - ==> Parameters:
09:12:24,699 DEBUG findAll:54 - <==      Total: 3
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=User{id=1, username='lucy',
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=User{id=1, username='lucy',
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=User{id=2, username='tom',
09:12:24,706 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.
09:12:24,706 DEBUG JdbcTransaction:54 - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@28ac3dc3
09:12:24,706 DEBUG PooledDataSource:54 - Returned connection 682376643 to pool.
```

5.2 一对多查询

5.2.1 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



5.2.2 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：

信息 | 结果1 | 概况 | 状态

id	username	password	birthday	id1	ordertime	total	uid	oid
1	lucy	123	2018-12-12	1	2018-12-12	3000	1	1
1	lucy	123	2018-12-12	2	2019-12-12	4000	1	2
2	tom	123	2018-12-12	3	2020-12-12	5000	2	3
5	haohao	123	2018-12-12	(Null)	(Null)	(Null)	(Null)	(Null)

5.2.3 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;
}
```

5.2.4 创建UserMapper接口

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

5.2.5 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>
```

5.2.6 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("-----------------------------------");
}
```

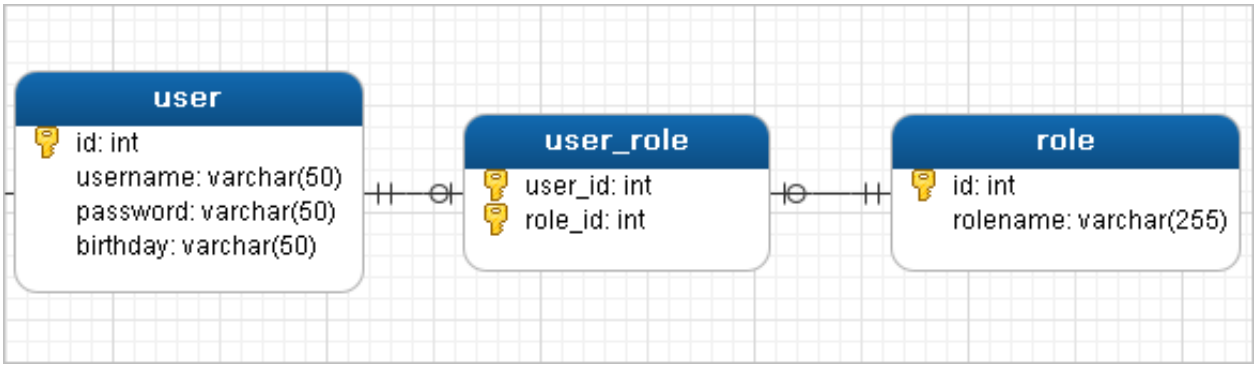
```
10:02:27,817 DEBUG findAll:54 - ==>  Preparing: select *,o.id oid from user u left join orders o on u.id=o.uid
10:02:27,843 DEBUG findAll:54 - ==> Parameters:
10:02:27,865 DEBUG findAll:54 - <==      Total: 4
lucy
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=null}
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=null}
----------------------------------
tom
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=null}
----------------------------------
haohao
----------------------------------

10:02:27,868 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Co
10:02:27,869 DEBUG JdbcTransaction:54 - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@289d1c02]
10:02:27,869 DEBUG PooledDataSource:54 - Returned connection 681384962 to pool.
```

5.3 多对多查询

5.3.1 多对多查询的模型

用户表和角色表的关系为，一个用户有多个角色，一个角色被多个用户使用

多对多查询的需求：查询用户同时查询出该用户的所有角色



5.3.2 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：

信息 | 结果1 | 概况 | 状态

id	username	password	birthday	id1	rolename
1	lucy	123	2018-12-12	1	CEO
1	lucy	123	2018-12-12	2	CFO
2	tom	123	2018-12-12	2	CFO
2	tom	123	2018-12-12	3	COO

5.3.3 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角色
    private List<Role> roleList;
}

public class Role {

    private int id;
    private String rolename;

}
```

5.3.4 添加UserMapper接口方法

```
List<User> findAllUserAndRole();
```

5.3.5 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>
```

5.3.6 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("----------------------------------");
}
```

```
10:34:36,884 DEBUG findAllUserAndRole:54 - ==>  Preparing: select u.*,r.*,r.id rid from user u left
10:34:36,903 DEBUG findAllUserAndRole:54 - ==> Parameters:
lucy
Role{id=1, rolename='CEO'}
Role{id=2, rolename='CFO'}
----------------------------------
tom
Role{id=2, rolename='CFO'}
Role{id=3, rolename='COO'}
----------------------------------
10:34:36,937 DEBUG findAllUserAndRole:54 - <==      Total: 4
10:34:36,939 DEBUG JdbcTransaction:54 - Resetting autocommit to true on JDBC Connection [com.mysql.
```

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

6.1 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

6.2 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>
```

6.3 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

注解	说明
@Results	代替的是标签<resultMap>该注解中可以使用单个@Result注解，也可以使用@Result集合。使用格式：@Results（{@Result（），@Result（）}）或@Results（@Result（））
@Resut	代替了<id>标签和<result>标签 @Result中属性介绍： column：数据库的列名 property：需要装配的属性名 one：需要使用的@One 注解（@Result（one=@One）（）） many：需要使用的@Many 注解（@Result（many=@many）（））

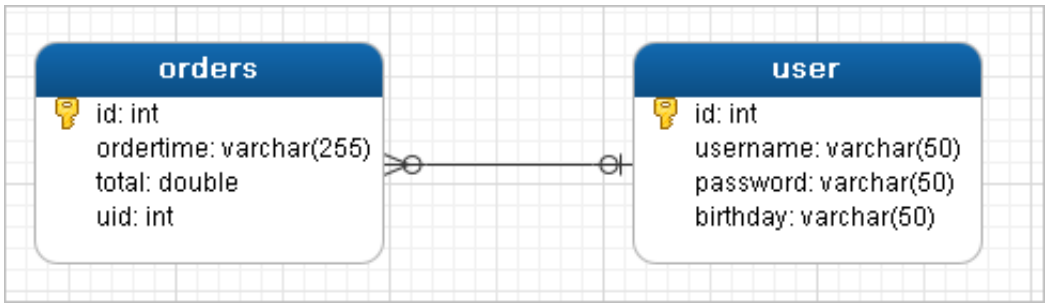
注解	说明
@One （一对一）	代替了<assocation> 标签，是多表查询的关键，在注解中用来指定子查询返回单一对象。 @One注解属性介绍： select: 指定用来多表查询的 sqlmapper 使用格式：@Result(column=" ",property="",one=@One(select=""))
@Many （多对一）	代替了<collection>标签, 是是多表查询的关键，在注解中用来指定子查询返回对象集合。 使用格式：@Result(property="",column="",many=@Many(select=""))

6.4 一对一查询

6.4.1 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



6.4.2 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：

信息 结果1 概况 状态

id	ordertime	total	uid	id1	username	password	birthday
1	2018-12-12	3000	1	1	lucy	123	1539751863457
2	2019-12-12	4000	1	1	lucy	123	1539751863457
3	2020-12-12	5000	2	2	tom	123	1539751863457

6.4.3 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;

}
```

6.4.4 创建OrderMapper接口

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

6.4.5 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);

}
```

6.4.6 测试结果

```
@Test
public void testSelectOrderAndUser() {
    List<Order> all = orderMapper.findAll();
    for(Order order : all){
        System.out.println(order);
    }
}
```

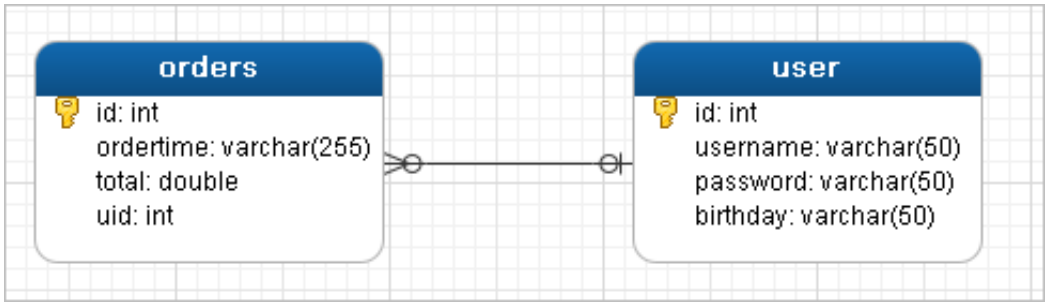
```
12:18:29,699 DEBUG findById:54 - ====>  Preparing: select * from user where id=?
12:18:29,699 DEBUG findById:54 - ====> Parameters: 2(Integer)
12:18:29,701 DEBUG findById:54 - <====      Total: 1
12:18:29,701 DEBUG findAll:54 - <==      Total: 3
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=User{id=1, username='lucy',
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=User{id=1, username='lucy',
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=User{id=2, username='tom',
```

6.5 一对多查询

6.5.1 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



6.5.2 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：

信息 | 结果1 | 概况 | 状态

id	username	password	birthday	id1	ordertime	total	uid	oid
1	lucy	123	2018-12-12	1	2018-12-12	3000	1	1
1	lucy	123	2018-12-12	2	2019-12-12	4000	1	2
2	tom	123	2018-12-12	3	2020-12-12	5000	2	3
5	haohao	123	2018-12-12	(Null)	(Null)	(Null)	(Null)	(Null)

6.5.3 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;
}
```

6.5.4 创建UserMapper接口

```
List<User> findAllUserAndOrder();
```

6.5.5 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);

}
```

6.5.6 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("-----------------------------");
}
```

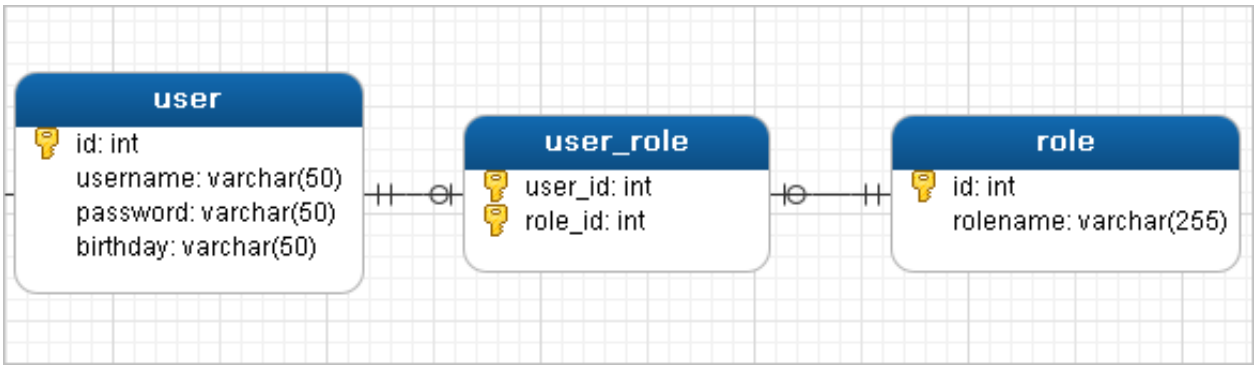
```
14:32:14,813 DEBUG findAllUserAndOrder:54 - ==>  Preparing: select * from user
14:32:14,844 DEBUG findAllUserAndOrder:54 - ==> Parameters:
14:32:14,860 DEBUG findByUid:54 - ====>  Preparing: select * from orders where uid=?
lucy
Order{id=1, ordertime=Wed Dec 12 00:00:00 GMT+08:00 2018, total=3000.0, user=null}
Order{id=2, ordertime=Thu Dec 12 00:00:00 GMT+08:00 2019, total=4000.0, user=null}
-----------------------------
tom
Order{id=3, ordertime=Sat Dec 12 00:00:00 GMT+08:00 2020, total=5000.0, user=null}
-----------------------------
haohao
-----------------------------
```

6.6 多对多查询

6.6.1 多对多查询的模型

用户表和角色表的关系为，一个用户有多个角色，一个角色被多个用户使用

多对多查询的需求：查询用户同时查询出该用户的所有角色



6.6.2 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：

信息 | 结果1 | 概况 | 状态

id	username	password	birthday	id1	rolename
1	lucy	123	2018-12-12	1	CEO
1	lucy	123	2018-12-12	2	CFO
2	tom	123	2018-12-12	2	CFO
2	tom	123	2018-12-12	3	COO

6.6.3 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角色
    private List<Role> roleList;
}

public class Role {

    private int id;
    private String rolename;
```

```
}
```

6.6.4 添加UserMapper接口方法

```
List<User> findAllUserAndRole();
```

6.6.5 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);
}
```

6.6.6 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("----------------------------------");
}
```

```
14:52:12,823 DEBUG findAllUserAndRole:54 - ==>  Preparing: select * from user
14:52:12,854 DEBUG findAllUserAndRole:54 - ==> Parameters:
14:52:12,870 DEBUG findByUid:54 - ====>  Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ====> Parameters: 1(Integer)
14:52:12,870 DEBUG findByUid:54 - <====      Total: 2
14:52:12,870 DEBUG findByUid:54 - ====>  Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ====> Parameters: 2(Integer)
14:52:12,870 DEBUG findByUid:54 - <====      Total: 2
14:52:12,870 DEBUG findByUid:54 - ====>  Preparing: select * from role r,user_role ur where r.id=ur.role_id
14:52:12,870 DEBUG findByUid:54 - ====> Parameters: 5(Integer)
14:52:12,885 DEBUG findByUid:54 - <====      Total: 0
lucy
Role{id=1, rolename='CEO'}
Role{id=2, rolename='CFO'}
------------------------------
tom
Role{id=2, rolename='CFO'}
Role{id=3, rolename='COO'}
------------------------------
haohao
------------------------------
```

第七部分：Mybatis缓存

7.1 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：

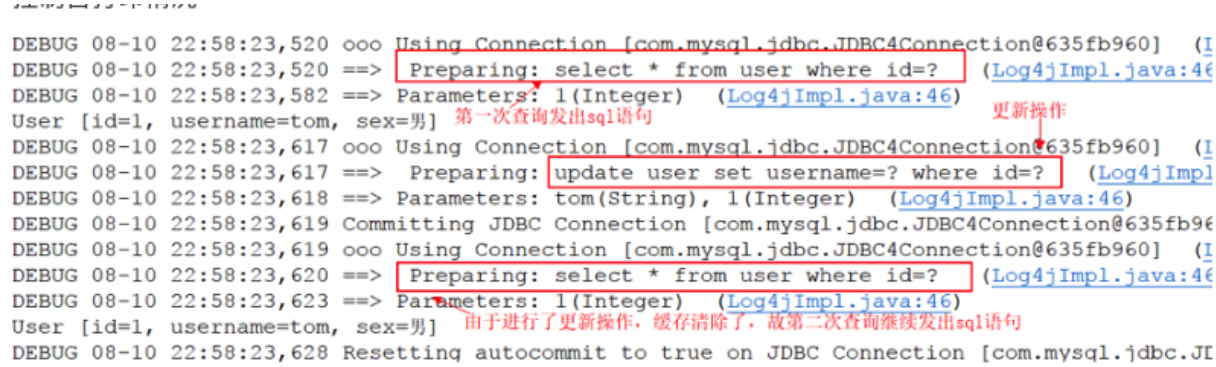
```
DEBUG 08-10 22:44:03,114 PooledDataSource forcefully closed/removed all connections.  (Lo
DEBUG 08-10 22:44:03,212 Openning JDBC Connection  (Log4jImpl.java:46)
DEBUG 08-10 22:44:03,464 Created connection 562266264.  (Log4jImpl.java:46)
DEBUG 08-10 22:44:03,467 ooo Using Connection [com.mysql.jdbc.JDBC4Connection@21838098]
DEBUG 08-10 22:44:03,467 ==>  Preparing: select * from user where id=?  (Log4jImpl.java:
DEBUG 08-10 22:44:03,514 ==> Parameters: 1(Integer)  (Log4jImpl.java:46)
User [id=1, username=tom, sex=男]
User [id=1, username=tom, sex=男]
DEBUG 08-10 22:44:03,538 Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.
DEBUG 08-10 22:44:03,539 Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@21838098
DEBUG 08-10 22:44:03,539 Returned connection 562266264 to pool.  (Log4jImpl.java:46)
```

就是在第一次查询时，发出了上面的sql语句，然后第二次查询直接打印用户对象
没有发出查询sql语句

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：

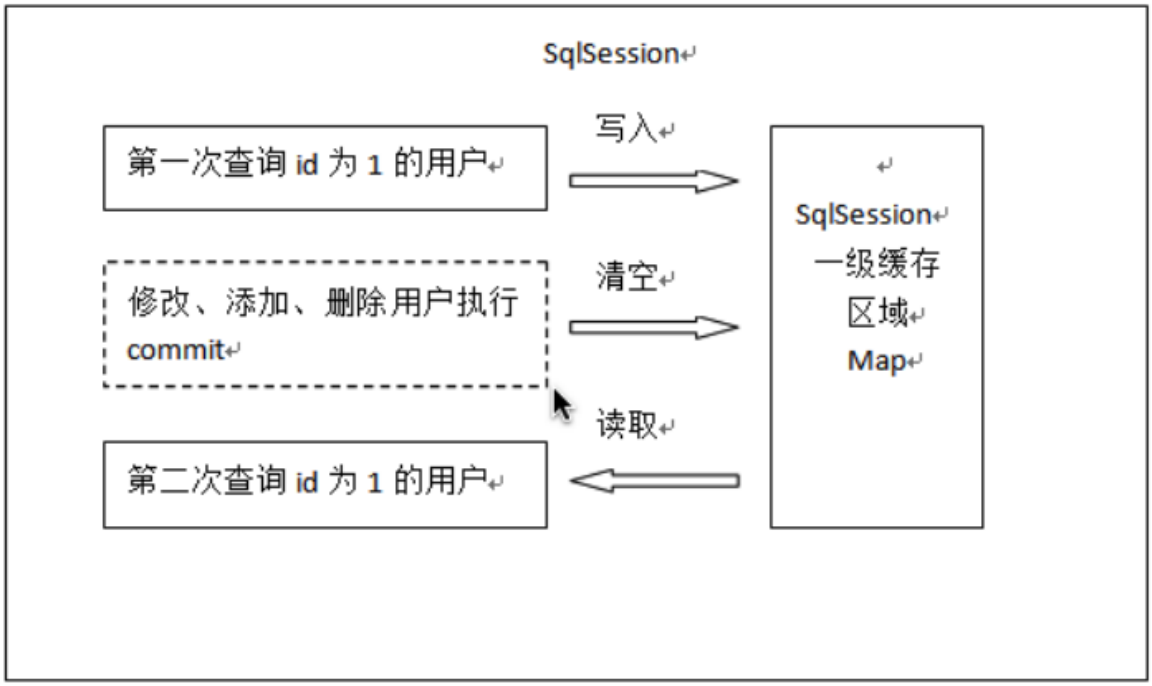


③、**总结**

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。

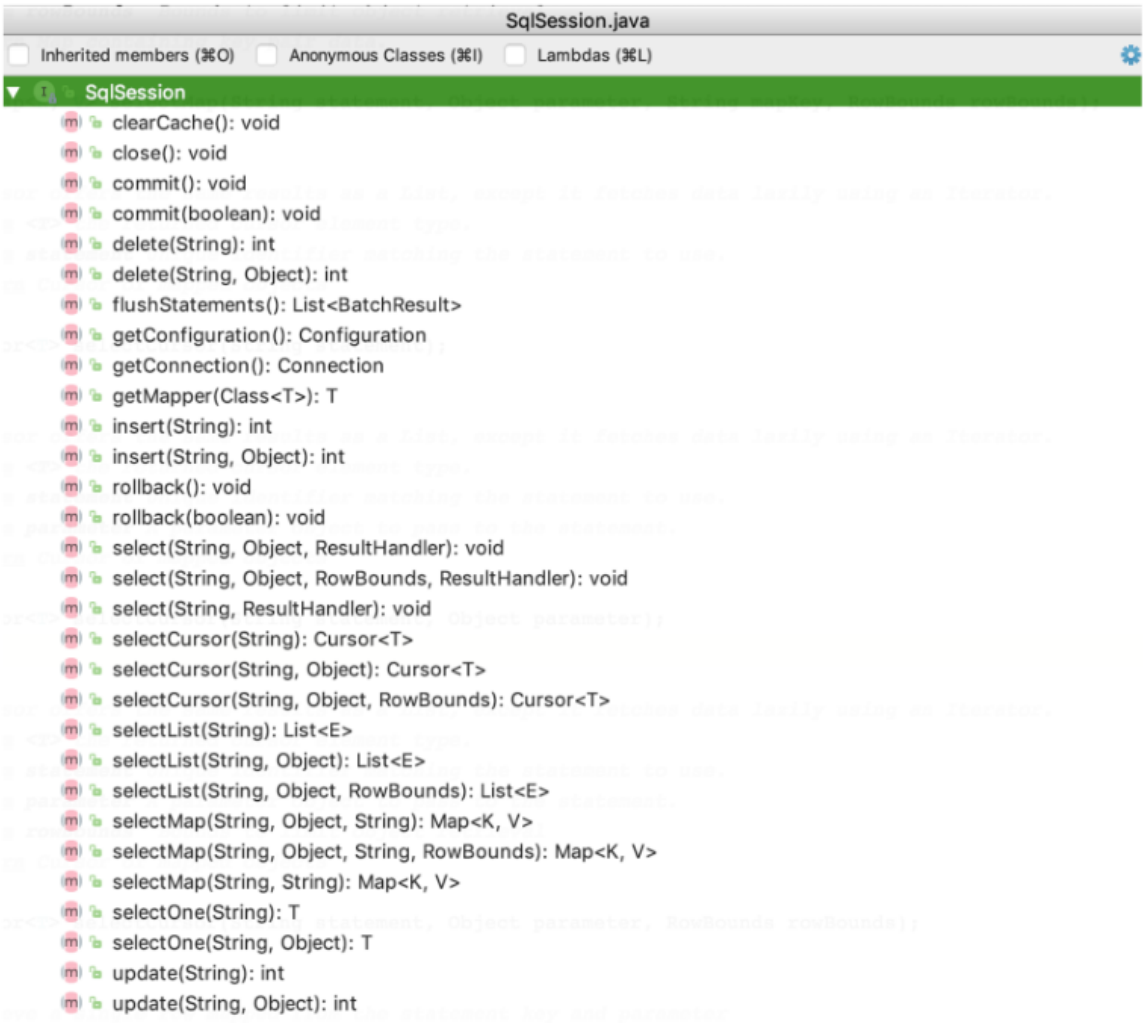
3、 第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直 接从缓存中获取用户信息



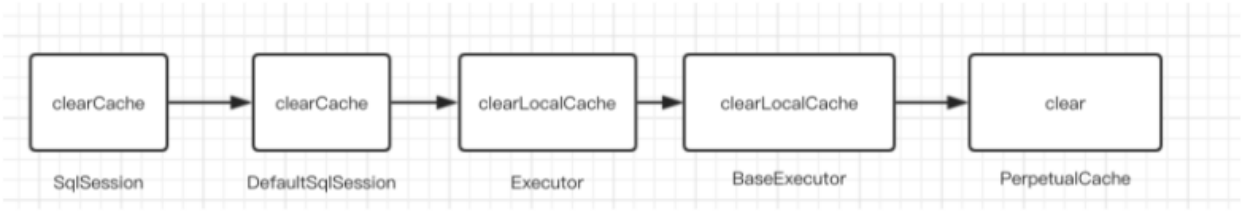
一级缓存原理探究与源码分析

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是什么东西呢？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SqlSession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，见如下。

```
<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;
```

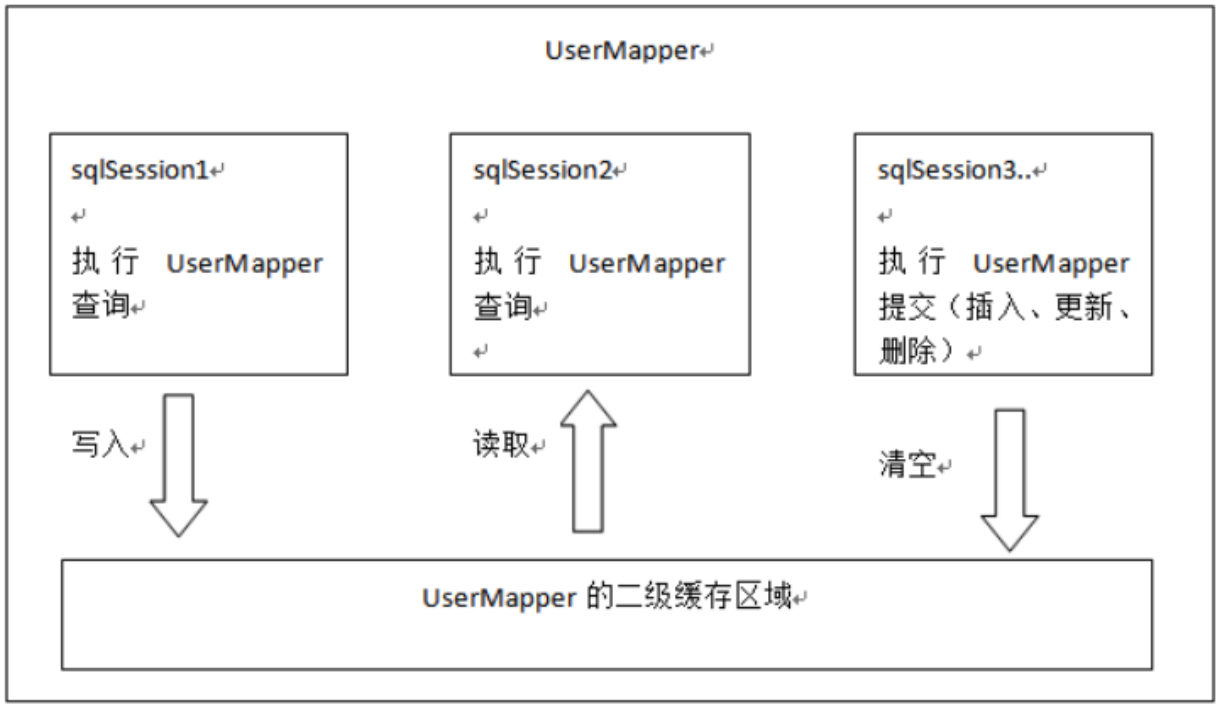
```
  localCache.putObject(key, EXECUTION_PLACEHOLDER);
  try {
    list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
  } finally {
    localCache.removeObject(key);
  }
  localCache.putObject(key, list);
  if (ms.getStatementType() == StatementType.CALLABLE) {
localOutputParameterCache.putObject(key, 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);
}
```

7.2 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

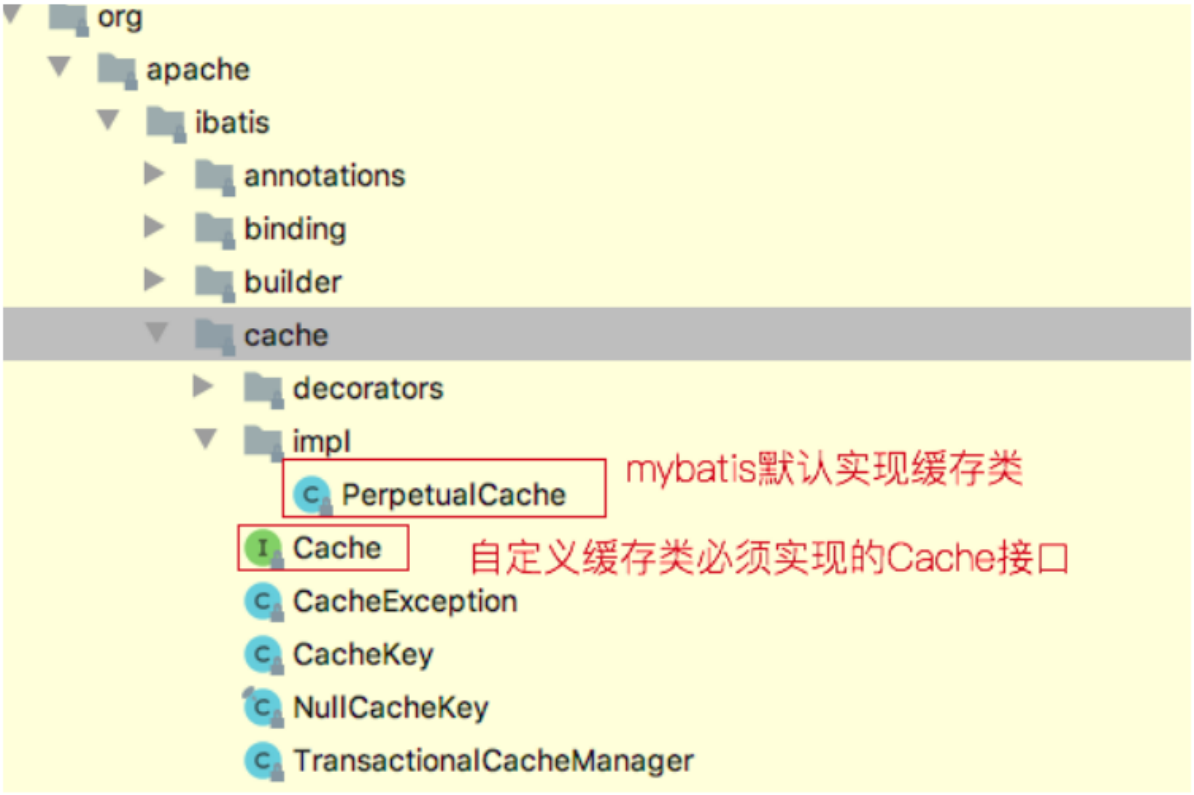
首先在全局配置文件sqlMapConfig.xml文件中加入如下代码:

```
<!--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
```

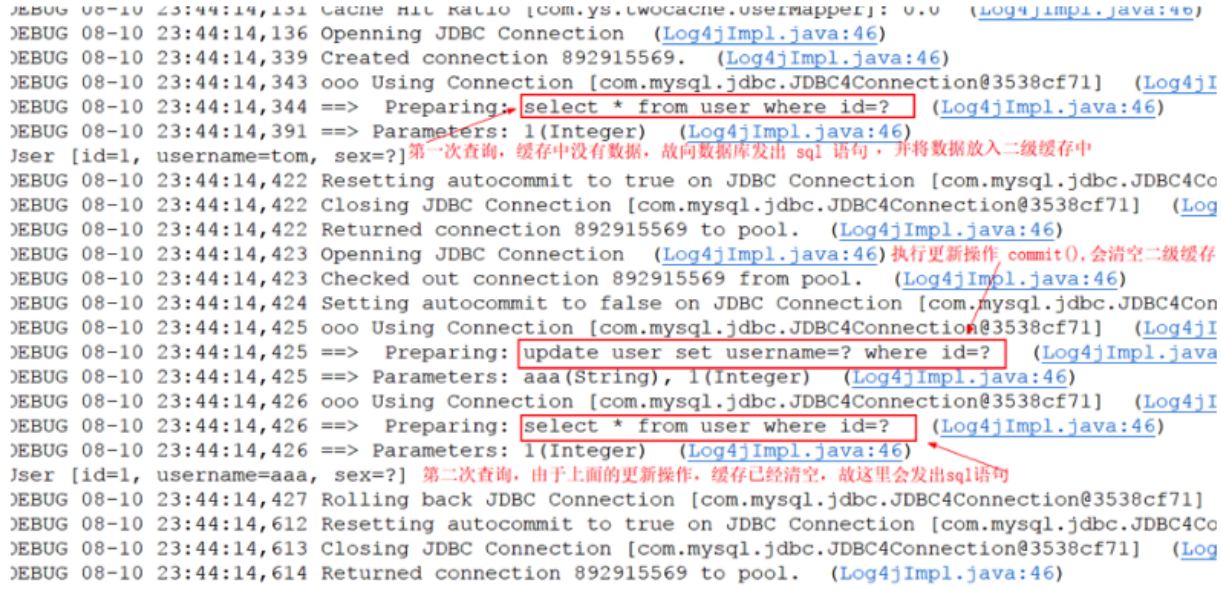
```
public void testTwoCache(){
  //根据 sqlSessionFactory 产生 session
  SqlSession sqlSession1 = sessionFactory.openSession();
  SqlSession sqlSession2 = sessionFactory.openSession();

  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.selectUserByUserId" ;
  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 = userMapper1.selectUserByUserId( 1 );
  System.out.println(u1);
  sqlSession1 .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。

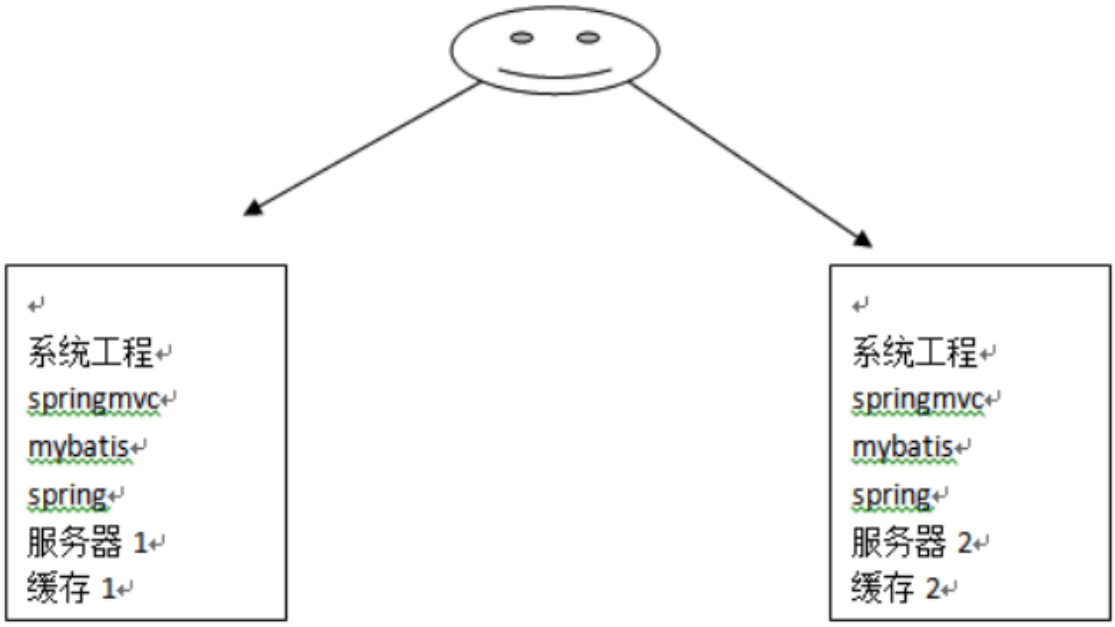
设置statement配置中的flushCache="true”属性，默认情况下为true,即刷新缓存，如果改成false则 不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。

```
<select id="selectUserByUserId" flushCache="true" useCache="false"
resultType="com.lagou.pojo.User" parameterType="int">
  select * from user where id=#{id}
</select>
```

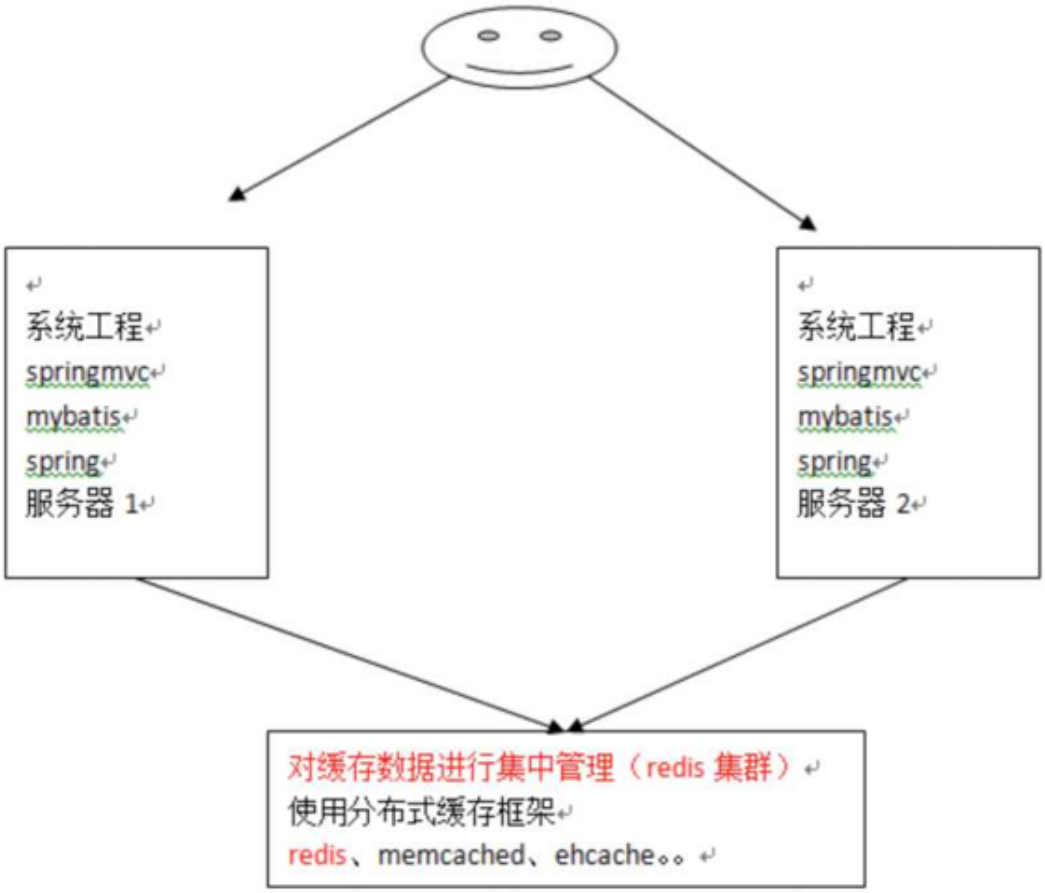
一般下执行完commit操作都需要刷新缓存，flushCache=true表示刷新缓存，这样可以避免数据库脏读。所以我们不用设置，默认即可

7.3 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，专门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>
```

2.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">
<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>
```

3.redis.properties

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

4.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

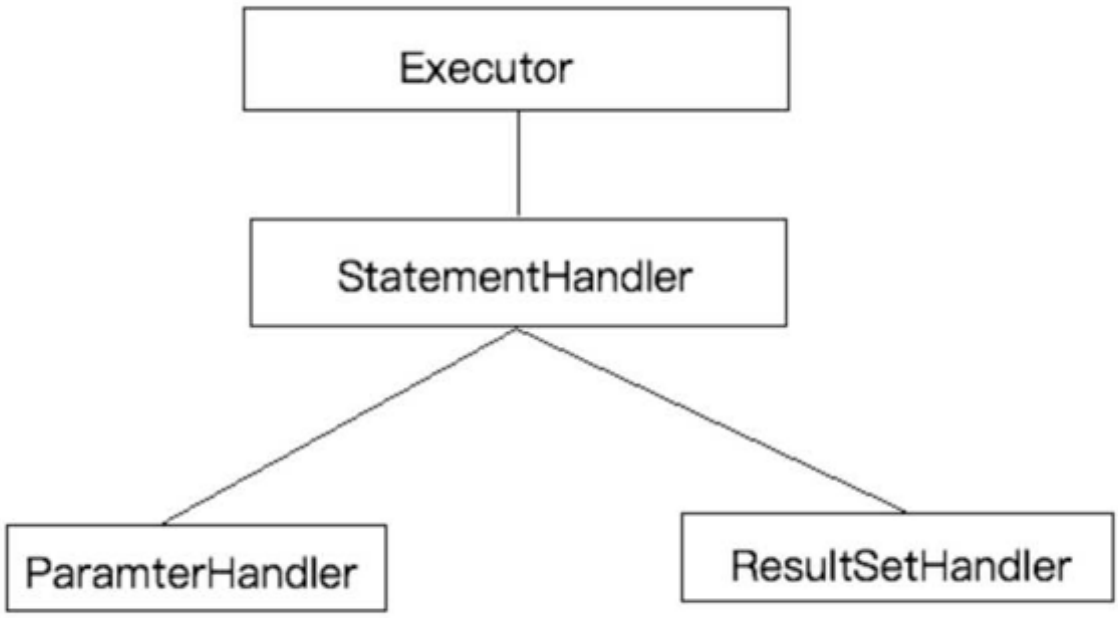
8.1 插件简介

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见的，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插件机制实现分页、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

8.2 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)；

8.3 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

8.4 自定义插件

8.4.1 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

8.4.2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);
    }
  }
}
```

8.5 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

8.6 pageHelper分页插件

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

开发步骤：

① 导入通用PageHelper的坐标

② 在mybatis核心配置文件中配置PageHelper插件

③ 测试分页数据获取

①导入通用PageHelper坐标

```
    <dependency>
        <groupId>com.github.pagehelper</groupId>
        <artifactId>pagehelper</artifactId>
        <version>3.7.5</version>
    </dependency>
    <dependency>
        <groupId>com.github.jsqlparser</groupId>
        <artifactId>jsqlparser</artifactId>
        <version>0.9.1</version>
     </dependency>
```

② 在mybatis核心配置文件中配置PageHelper插件

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

③ 测试分页代码实现

```
    @Test
    public void testPageHelper() {

        //设置分页参数
        PageHelper.startPage(1, 2);
        List<User> select = userMapper2.select(null);
        for (User user : select) {
            System.out.println(user);
        }
    }
}
```

获得分页相关的其他参数

```
//其他分页的数据
PageInfo<User> pageInfo = new PageInfo<User>(select);
System.out.println("总条数："+pageInfo.getTotal());
System.out.println("总页数："+pageInfo. getPages ());
System.out.println("当前页："+pageInfo. getPageNum());
System.out.println("每页显万长度："+pageInfo.getPageSize());
System.out.println("是否第一页："+pageInfo.isIsFirstPage());
System.out.println("是否最后一页："+pageInfo.isIsLastPage());
```

8.7 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>
```

2. Mybatis配置文件中完成配置

```
<plugins>
  <!--分页插件：如果有分页插件，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>
```

3. 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;

}
```

4. 定义通用mapper

```
import com.lagou.domain.User;

import tk.mybatis.mapper.common.Mapper;

public interface UserMapper extends Mapper<User> {

}
```

5. 测试

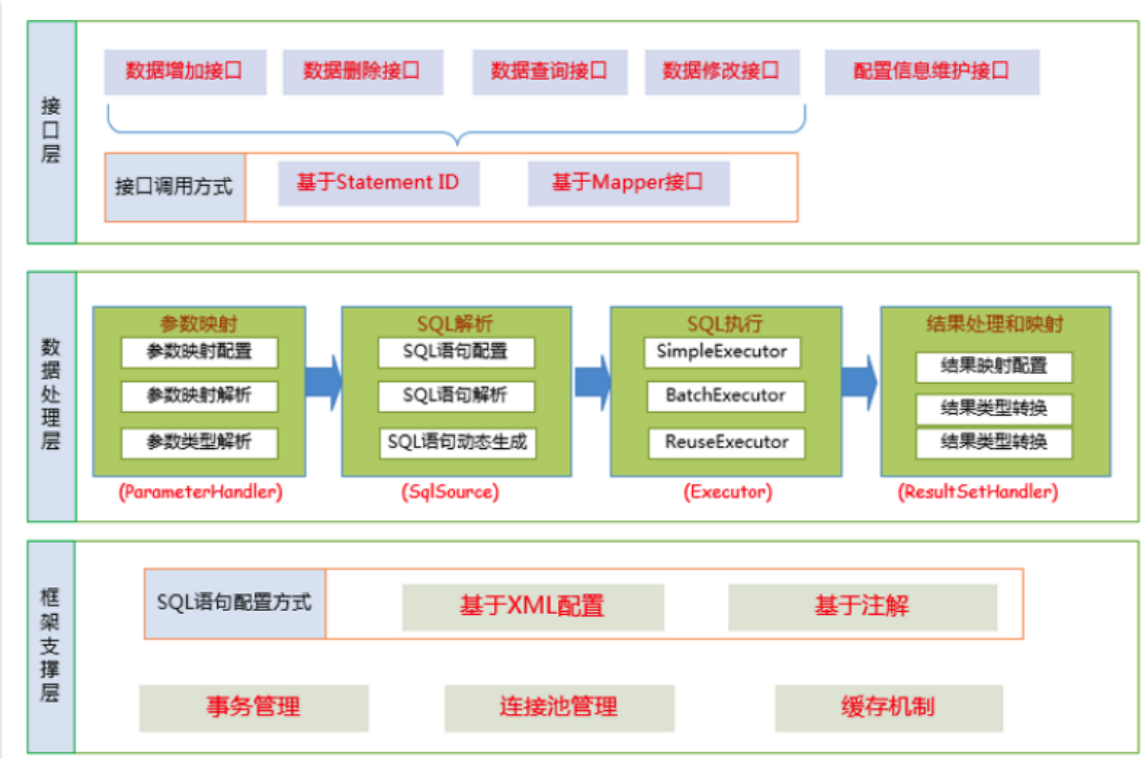
```
public class UserTest {
    @Test
    public void test1() throws IOException {

        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);

        //(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，只能有一个返回值
        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

9.1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：

a. 使用传统的MyBati s提供的API ；

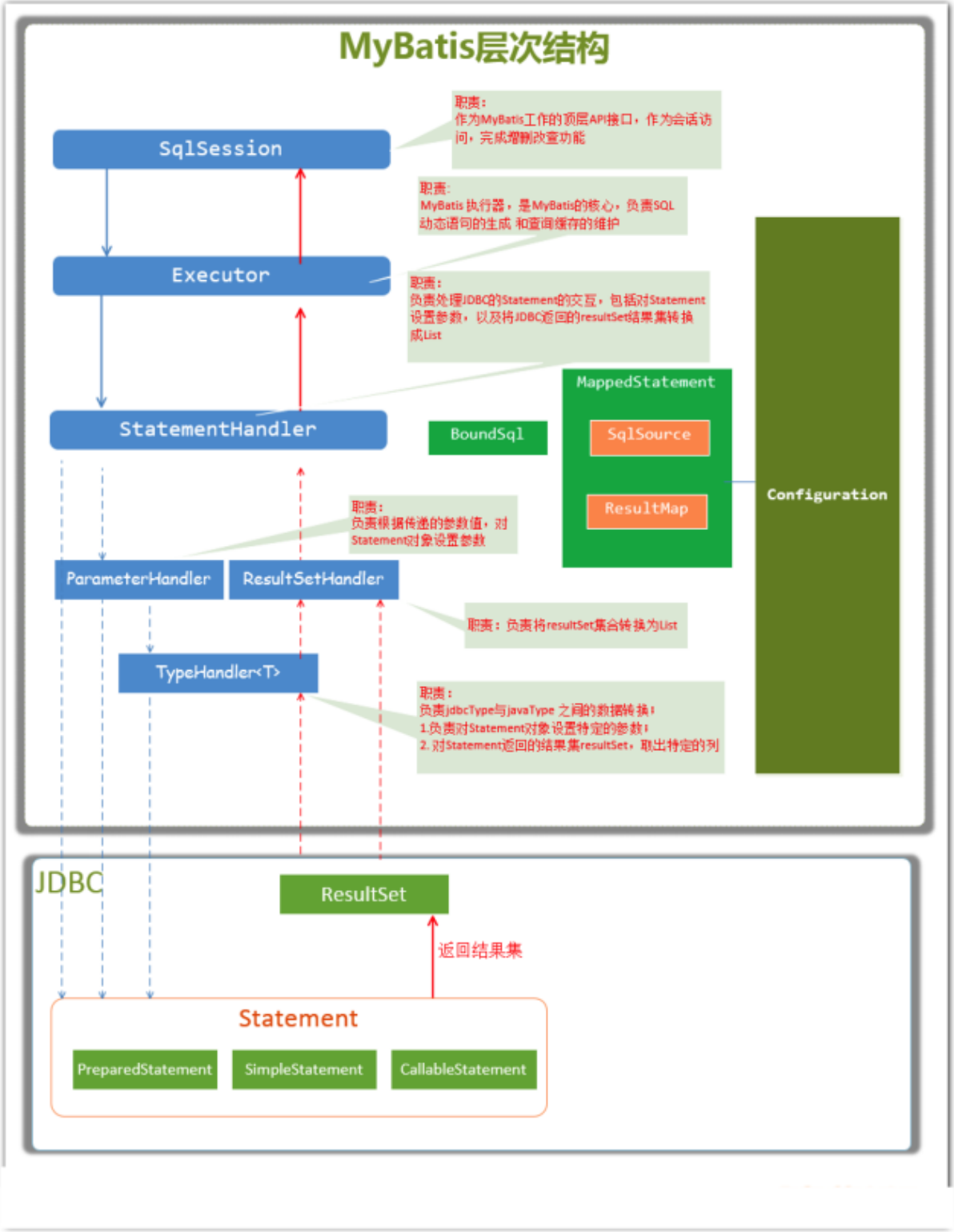
b. 使用Mapper代理的方式

(2) 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。

(3) 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 用的东西，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

9.2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



9.3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),一个是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

(2) 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。

(3) 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：

(A) 根据SQL的ID查找对应的MappedStatement对象。

(B) 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。

(C) 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。

(D) 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。

(E) 释放连接资源。

(4) 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

10.1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是专门解析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

 databaseIdProviderElement(root.evalNode("databaseIdProvider"));
                // 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是用来逻辑分页
                    // 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();
        }
        List<E> list;
        try {
            queryStack++;
            list = resultHandler == null ? (List<E>) localCache.getObject(key)
: null;
            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);
}
```

10.2 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
```

```
            Object param = method.convertArgsToSqlCommandParam(args);
            // 转换 rowCount
            result = rowCountResult(sqlSession.update(command.getName(),
param));
            break;
        }
        case DELETE: {
            //转换参数
            Object param = method.convertArgsToSqlCommandParam(args);
            // 转换 rowCount
            result = rowCountResult(sqlSession.delete(command.getName(),
                    param));
            break;
        }
        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
行处理
        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;
}
```

10.3 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,
                                        keyGenerator, keyProperty, keyColumn,
databaseId, langDriver, resultSets);
}

// 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。

10.4 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长，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>
```

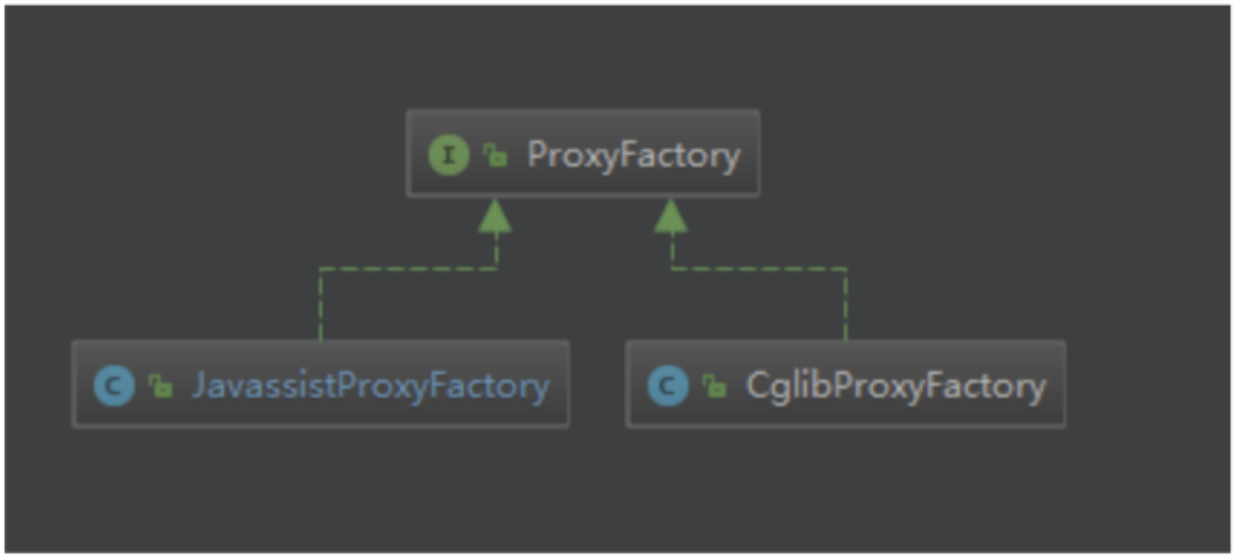
延迟加载原理实现

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性的 getting 方法时，**进入拦截器方法**。比如调用 `a.getB().getName()` 方法，进入拦截器的 `invoke(...)` 方法，发现 `a.getB()` 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 `a` 对象 `b` 属性就有值了，接着完成 `a.getB().getName()` 方法的调用。这就是延迟加载的基本原理

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

```
protected ProxyFactory proxyFactory = new JavassistProxyFactory();
```

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。

11.1 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;
    }
}
```

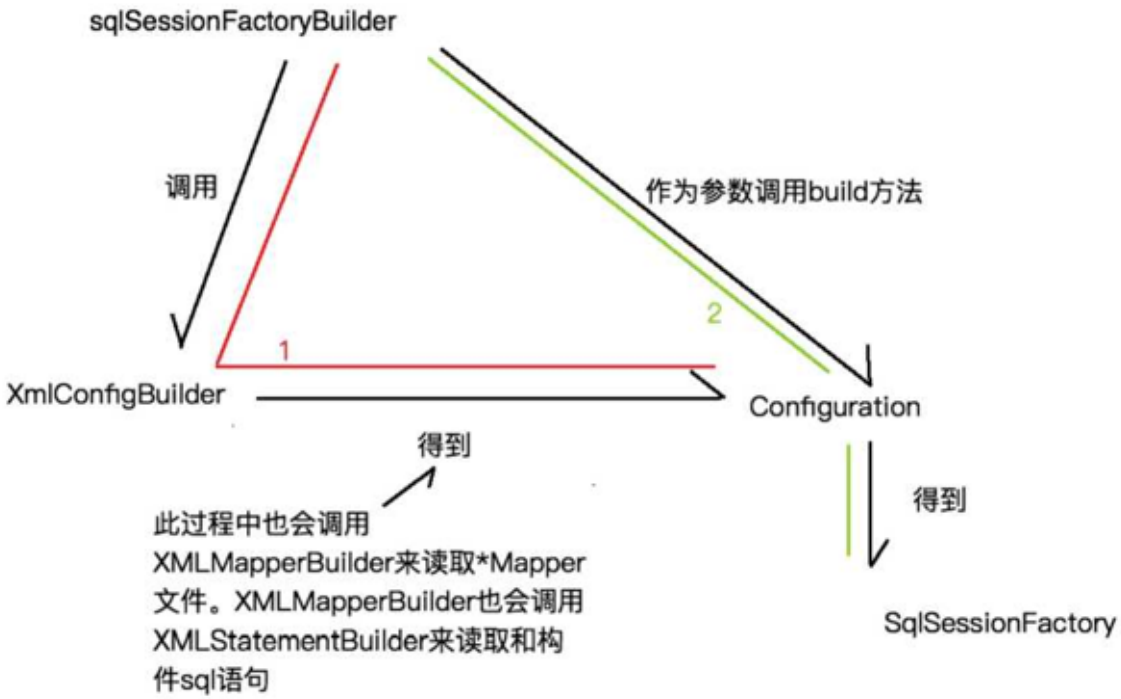
调用

```
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。

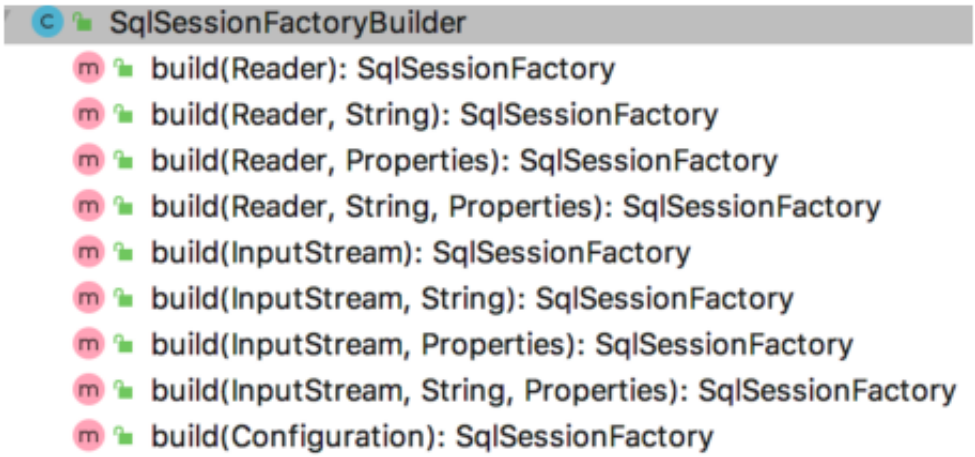
```
private void parseConfiguration(XNode root) {
try {
  //issue #117 read properties first
  //解析<properties />标签
  propertiesElement(root.evalNode("properties"));
  // 解析 <settings /> 标签
  Properties settings = settingsAsProperties(root.evalNode("settings"));
  //加载自定义的VFS实现类
  loadCustomVfs(settings);
  // 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
```

```
    databaseIdProviderElement(root.evalNode("databaseIdProvider"));
}
```

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

11.2 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。简单工厂模式专门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门生产电脑，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();
}
```

2. 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");
  }
}
```

3. 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;
    }
}
```

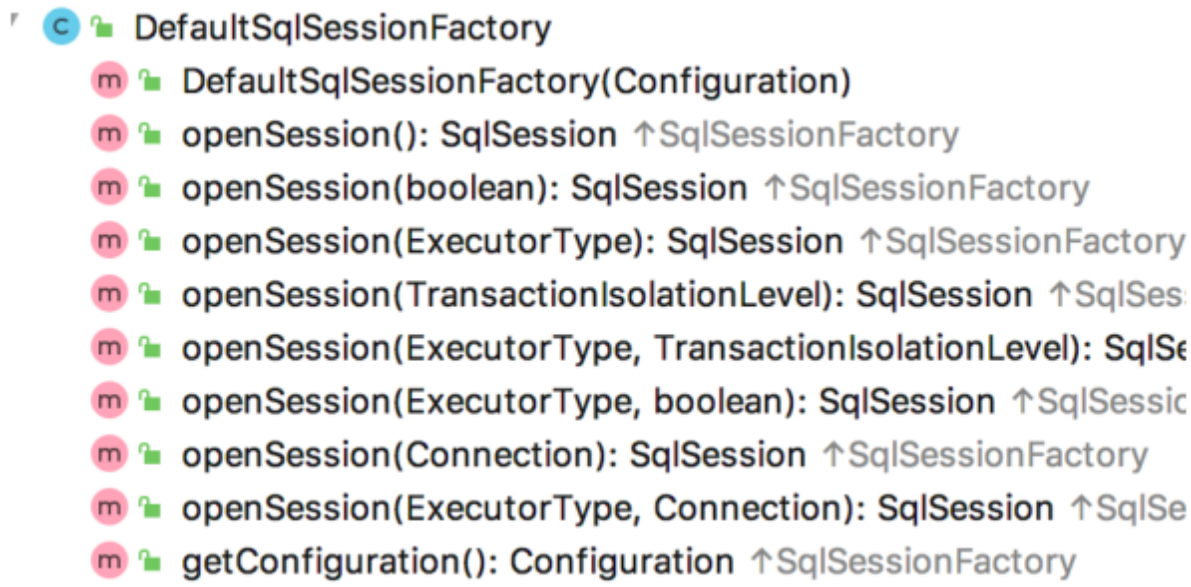
客户端调用工厂类
客户端调用工厂类，传入“hp”生产出惠普电脑并调用该电脑对象的start方法：

```
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

11.3 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

1.1 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。
>
> 
>

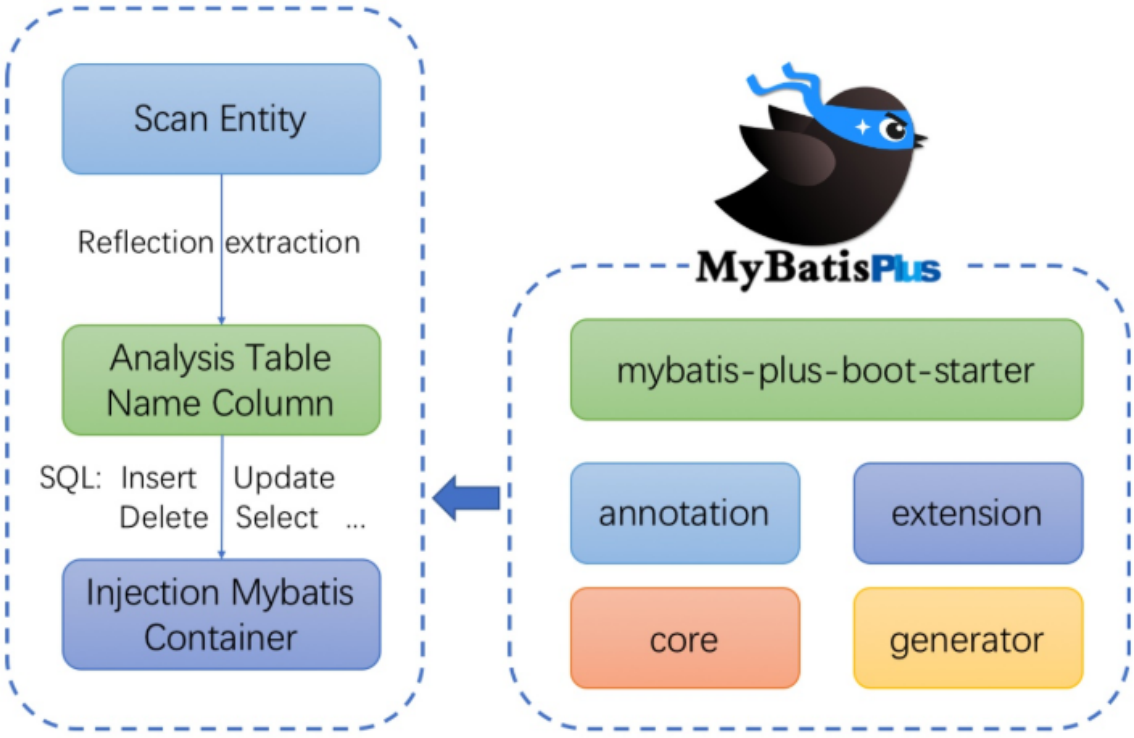
1.2 特性

- **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
- **内置分页插件**：基于 MyBatis 物理分页，开发者无需关心具体操作，配置好插件之后，写分页等同于普通 List 查询
- **分页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

1.3 架构



1.4 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

2. Mybatis-Plus快速入门

2.1 安装

全新的 `MyBatis-Plus` 3.0 版本基于 JDK8，提供了 `lambda` 形式的调用，所以安装集成 MP3.0 要求如下：

- JDK 8+
- Maven or Gradle

Release

Spring Boot

Maven：

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.0</version>
</dependency>
```

Spring MVC

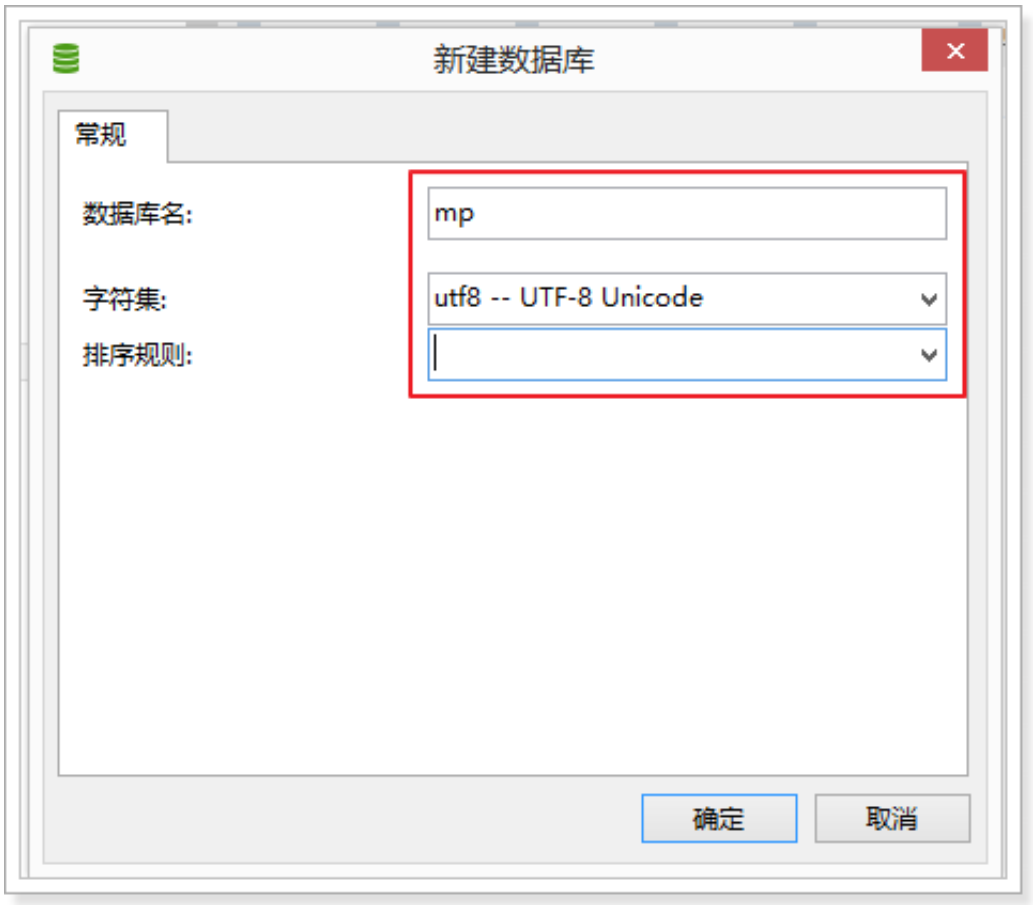
Maven:

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus</artifactId>
    <version>3.4.0</version>
</dependency>
```

> WARNING
>
> 引入 `MyBatis-Plus` 之后请不要再次引入 `MyBatis` 以及 `MyBatis-Spring`，以避免因版本差异导致的问题。

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。

2.2 创建数据库以及表

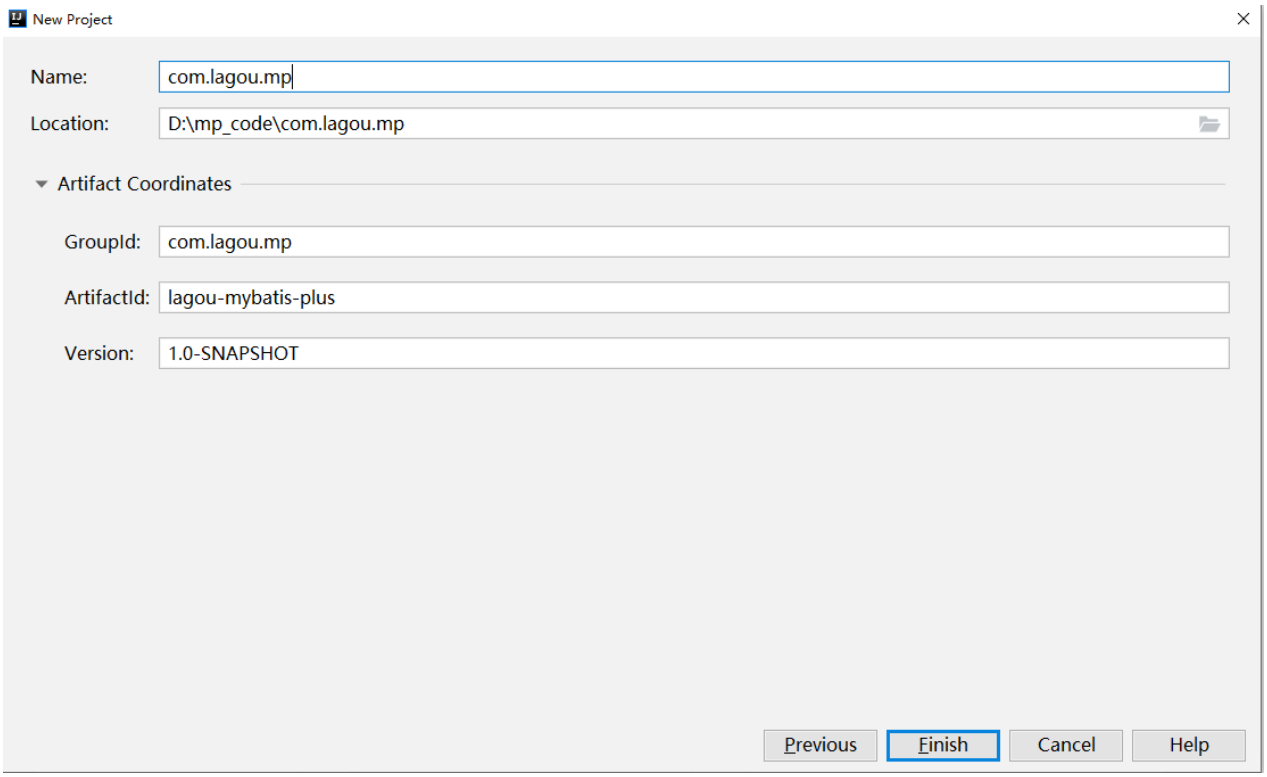


创建User表，其表结构如下：

id	name	age	email
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

```
--  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');
```

2.3 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>
```

2.4 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"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
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">

<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">

<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);
    }
}
```

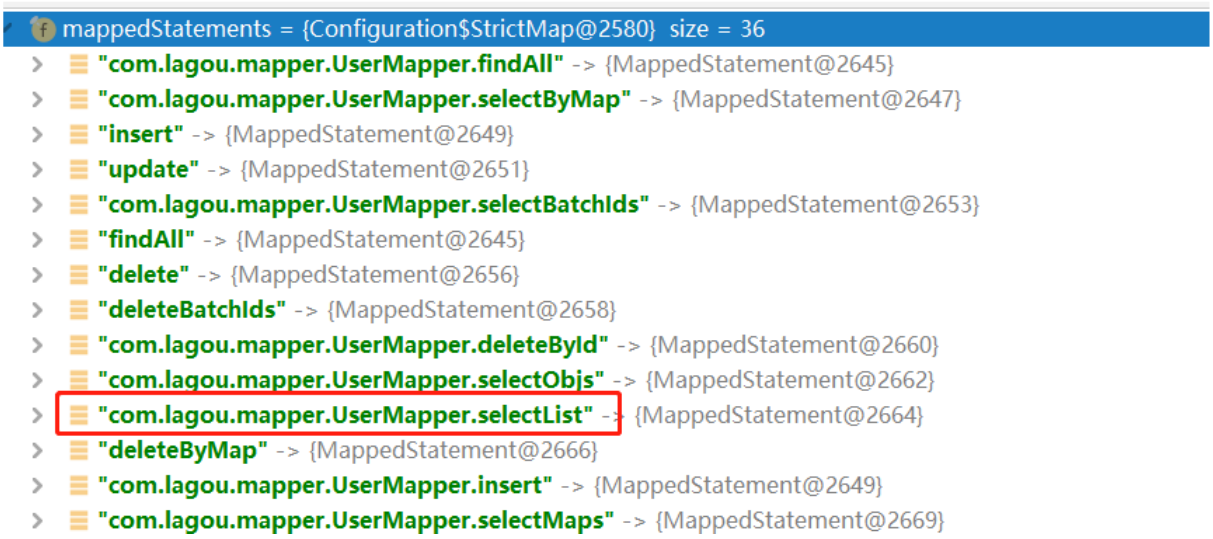
测试：

```
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



2.5 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"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
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"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns: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">
```

```
    <!--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：

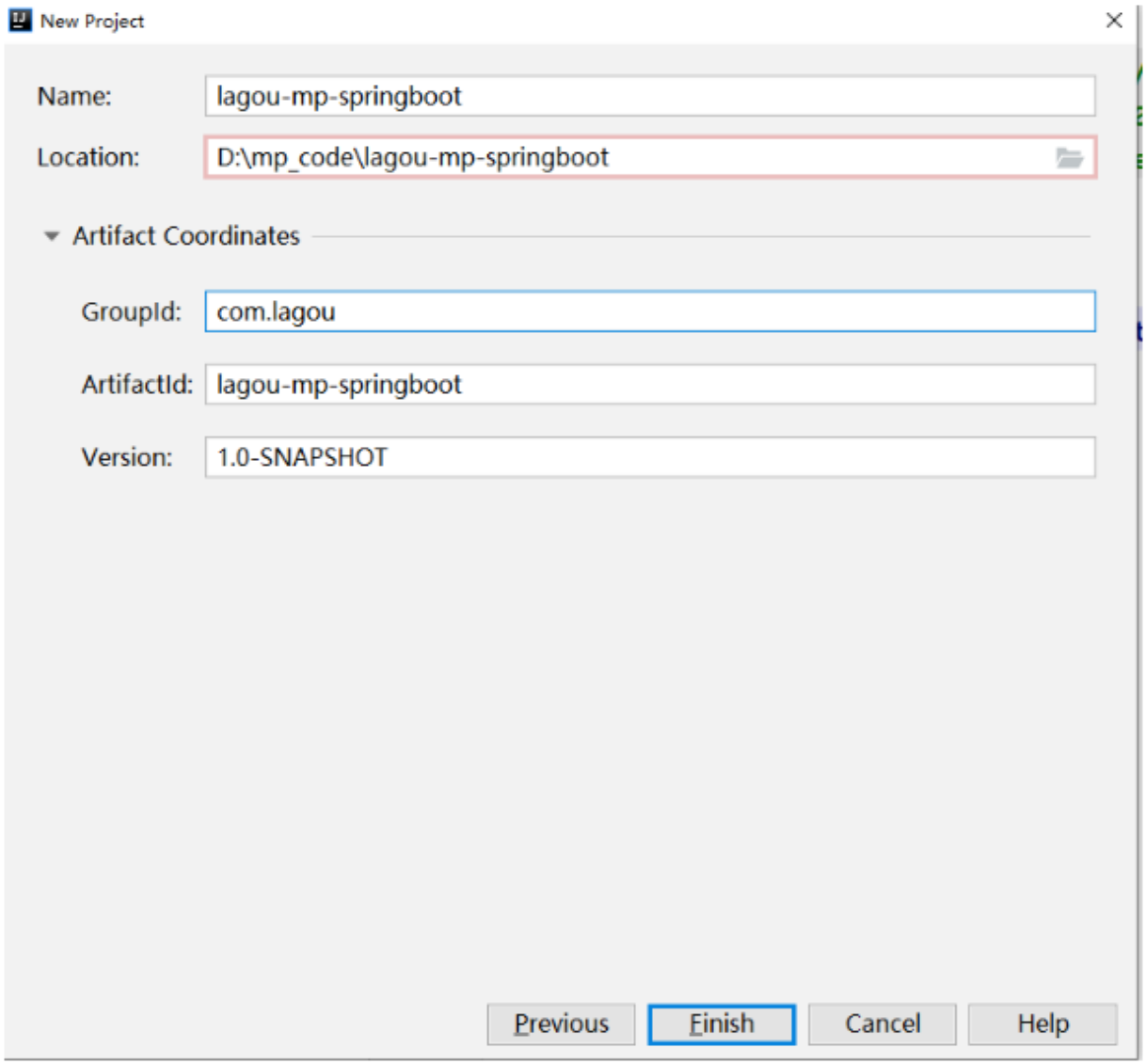
```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
```

```
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);
        }

    }
```

2.6 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

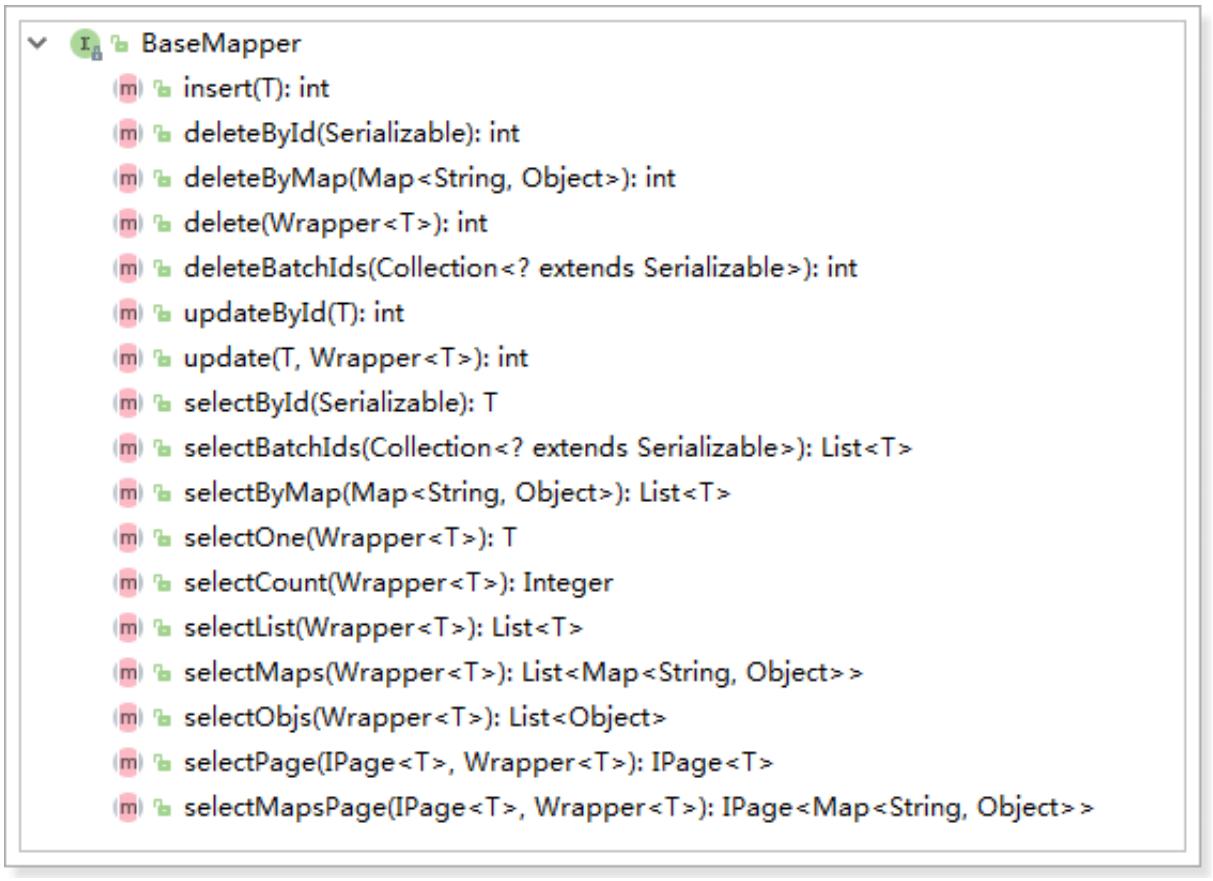
```
package com.lagou.mp;

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)
```

3. 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。



3.1 插入操作

方法定义

```
    /**
     * 插入一条记录
     *
     * @param entity 实体对象.
     */
    int insert(T entity);
```

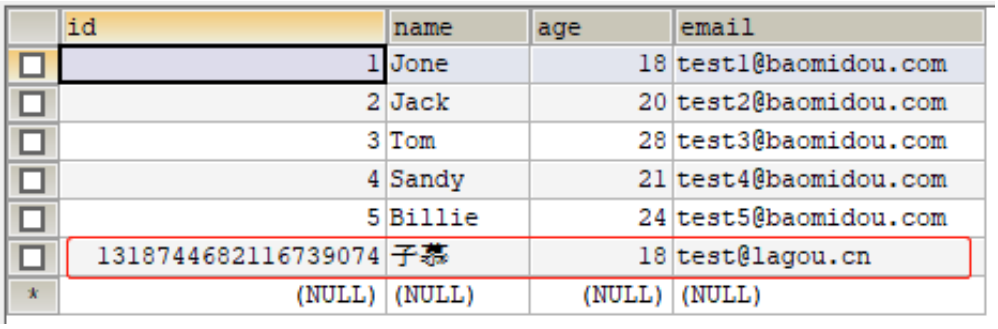
测试用例

```
@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");
        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
```



	id	name	age	email
	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
	1318744682116739074	子慕	18	test@lagou.cn
*	(NULL)	(NULL)	(NULL)	(NULL)

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长，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类型为自增长
    private Long id;
```

```
    private String userName;
    private String password;
    private String name;
    private Integer age;
    private String email;
}
```

数据插入成功：

	id	name	age	email
☐	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
☐	6	子慕	18	test@lagou.cn
*	(NULL)	(NULL)	(NULL)	(NULL)

@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：

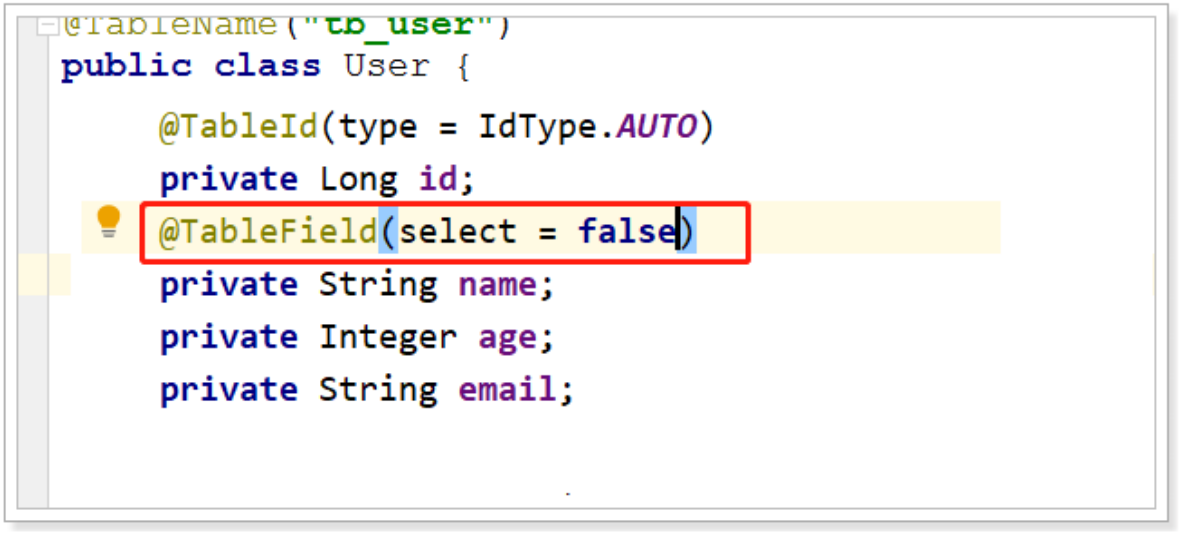
1、对象中的属性名和字段名不一致的问题（非驼峰）

2、对象中的属性字段在表中不存在的问题

使用：

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@TableName("tb_user")
public class User {

    @TableId(type = IdType.AUTO)
    private Long id;
    private String name;
    private Integer age;
    @TableField(value = "email") //解决字段名不一致
    private String mail;

    @TableField(exist = false)
    private String address; //该字段在数据库表中不存在

}
```

其他用法，如大字段不加入查询字段：



效果：

```
User(id=1, name=null, age=18, email=test1@baomidou.com)
User(id=2, name=null, age=20, email=test2@baomidou.com)
User(id=3, name=null, age=28, email=test3@baomidou.com)
User(id=4, name=null, age=21, email=test4@baomidou.com)
User(id=5, name=null, age=24, email=test5@baomidou.com)
User(id=6, name=null, age=18, email=test@lagou.cn)
User(id=7, name=null, age=18, email=test@lagou.cn)
```

3.2 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：

	id	name	age	email
☐	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
☐	6	子慕	21	test@lagou.cn

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。

3.3 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
```

id	name	age	email
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

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
```

3.4 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、分页查询等操作。

3.4.1、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)
```

3.4.2、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)
```

3.4.3、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)
```

3.4.4、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
```

3.4.5、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)
```

3.4.6、selectPage

方法定义：

```
/**
 * 根据 entity 条件，查询全部记录（并翻页）
 *
 * @param page         分页查询条件（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);
```

配置分页插件：

```
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 {

    /**
     * 分页插件
     */
    @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("总页数: " + iPage.getPages());


        List<User> users = iPage.getRecords();
        for (User user : users) {
            System.out.println("user = " + user);
        }
    }

}
```

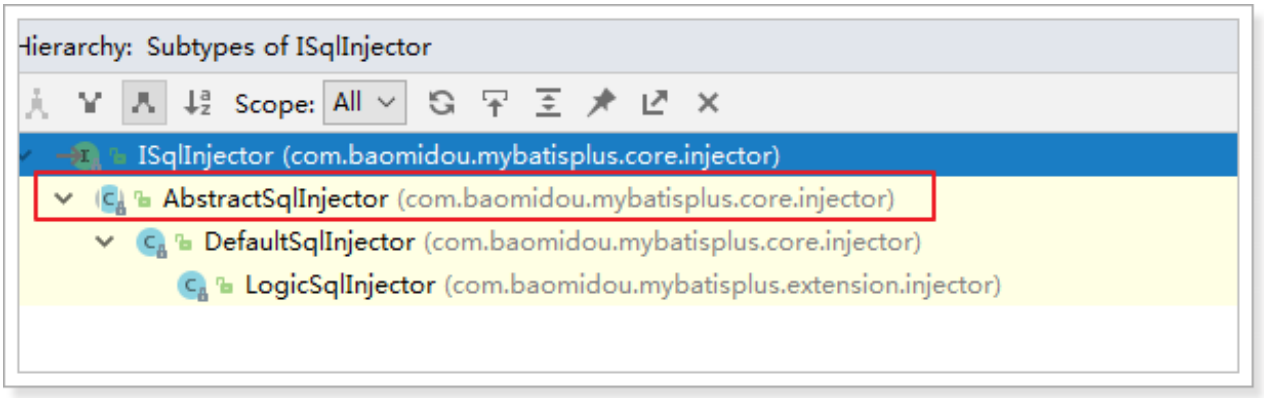
结果:

```
数据总条数: 4
总页数: 4
user = User(id=3, name=Tom, age=28, email=test3@baomidou.com)
```

3.5 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：

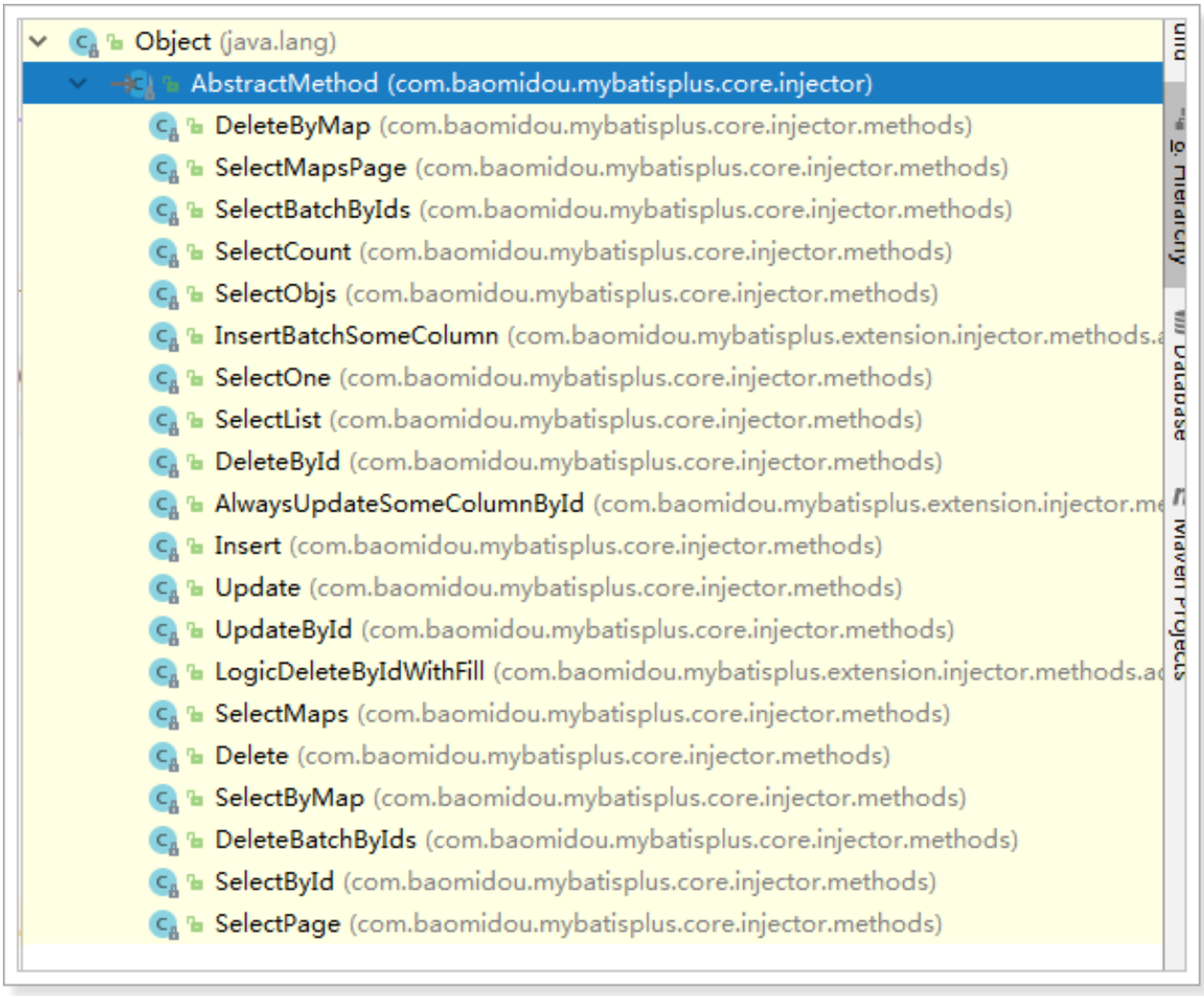
```
@Override
public void inspectInject(MapperBuilderAssistant builderAssistant, Class<?>
mapperClass) {
    Class<?> modelClass = extractModelClass(mapperClass);
    if (modelClass != null) {
        String className = mapperClass.toString();
        Set<String> mapperRegistryCache =
GlobalConfigUtils.getMapperRegistryCache(builderAssistant.getConfiguration());
        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);
```

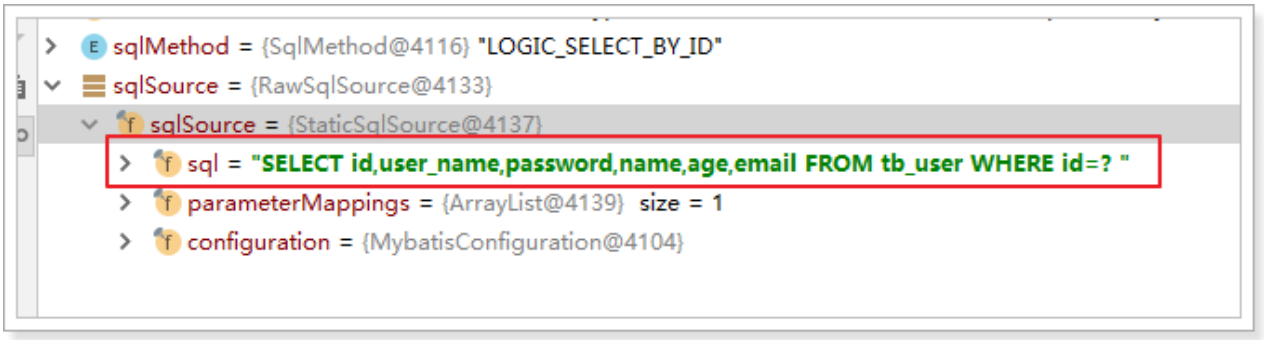
查看该方法的实现：



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。



4. 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://mybatis.plus/config/

下面我们对常用的配置做讲解。

4.1、基本配置

4.1.1、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>
```

4.1.2、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 `classpath*:` 开头 （即加载多个 jar 包下的 XML 文件）**

> 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">
<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>
```

```java
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：

```java
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：

```
ansaction]-[DEBUG] JDBC Connection [HikariProxyConnection@9063
G] ==>  Preparing: select * from tb_user where id = ?
G] ==> Parameters: 2(Long)
G] <==      Total: 1
osing non transactional SqlSession [org.apache.ibatis.session.
ge=20, email=test2@itcast.cn, address=null)
```

4.1.3、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>
```

4.2、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。

4.2.1、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 `@TableField` 注解指定数据库字段名

示例（SpringBoot）：

```
#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
```

4.2.2、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
```

4.3、DB 策略配置

4.3.1、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>
```

4.3.2、tablePrefix

- 类型：`String`
- 默认值：`null`

表名前缀，全局配置后可省略@TableName()配置。

SpringBoot：

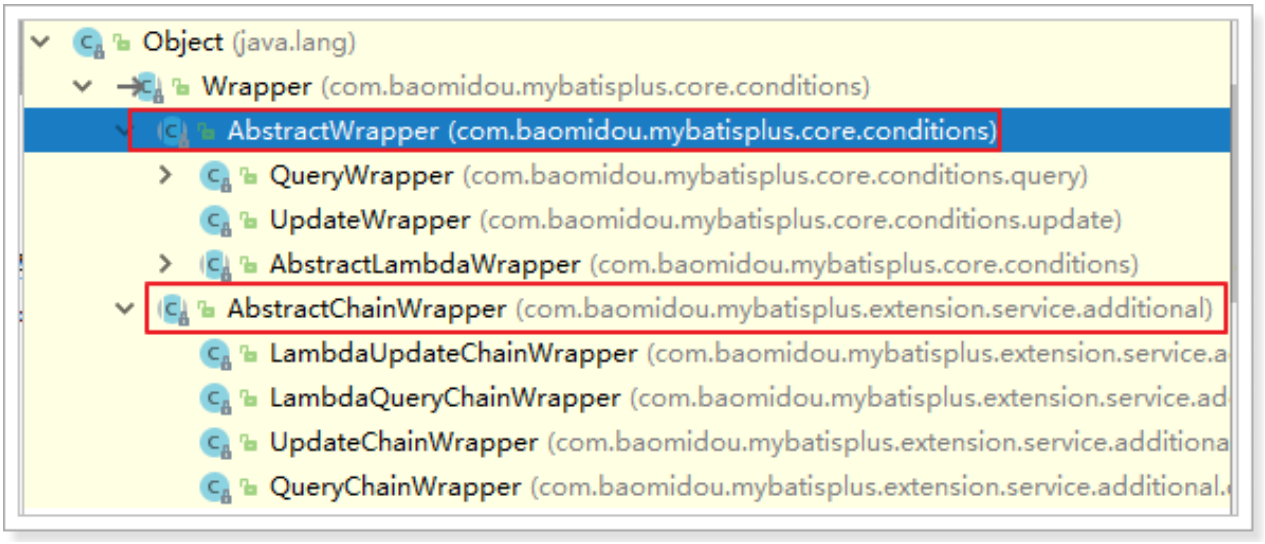
```
mybatis-plus.global-config.db-config.table-prefix=tb_
```

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>
```

5. 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

5.1、allEq

5.1.1、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
> `params` 与 `null2IsNull` : 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'`

5.1.2、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);
        }

    }

}
```

5.2、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")
                .ge("age", 20)
                .in("name", "jack", "jone", "tom");

        List<User> users = this.userMapper.selectList(wrapper);
        for (User user : users) {
            System.out.println(user);
        }
    }

}
```

5.3、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);
        }

    }

}
```

5.4、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);
        }

    }

}
```

5.5、逻辑查询

- or
  - 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
  - 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);
        }

    }

}
```

5.6、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);
        }

    }

}
```

6. 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；

6.1、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;

}
```

6.2、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);
    }

}
```

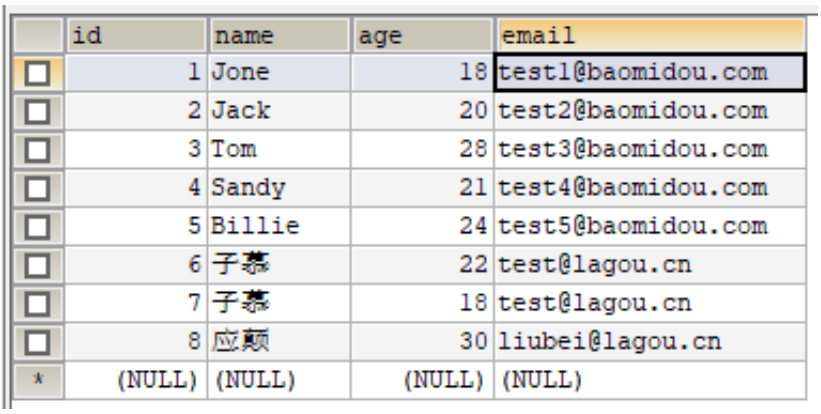
6.3、新增数据

```
@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");

        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
```



	id	name	age	email
☐	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
☐	6	子慕	22	test@lagou.cn
☐	7	子慕	18	test@lagou.cn
☐	8	应颠	30	liubei@lagou.cn
*	(NULL)	(NULL)	(NULL)	(NULL)

6.4、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
```

id	name	age	email
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
6	子慕	22	test@lagou.cn
7	子慕	18	test@lagou.cn
8	应颜	35	liubei@lagou.cn
(NULL)	(NULL)	(NULL)	(NULL)

6.5、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
```

6.6、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)
```

7. 插件

7.1、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">
<configuration>
    <plugins>
        <plugin interceptor="com.lagou.mp.plugins.MyInterceptor"></plugin>
    </plugins>
</configuration>
```

7.2、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。

7.3、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">
<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
长 -->
        <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)
  ................
```

7.4、乐观锁插件

7.4.1、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

7.4.2、插件配置

spring xml:

```
<bean
class="com.baomidou.mybatisplus.extension.plugins.OptimisticLockerInterceptor"
/>
```

spring boot:

```
@Bean
public OptimisticLockerInterceptor optimisticLockerInterceptor() {
    return new OptimisticLockerInterceptor();
}
```

7.4.3、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;
```

7.4.4、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。

7.4.5、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`
- `newVersion` 会回写到 `entity` 中
- 仅支持 `updateById(id)` 与 `update(entity, wrapper)` 方法
- **在 update(entity, wrapper) 方法下, wrapper 不能复用!!!**

8. 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。

8.1、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);
}
```

8.2、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;
    }
}
```

8.3、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);
    }
```

```
}
```

8.4、注册到Spring容器

```
/**
 * 自定义SQL注入器
 */
@Bean
public MySqlInjector mySqlInjector(){
    return new MySqlInjector();
}
```

8.5、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。

9. 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。

9.1、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
}
```

9.2、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) {

    }
}
```

9.3、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：

	id	name	age	email	version
☐	1	Jone	18	test1@baomidou.com	1
☐	2	Jack	30	test2@baomidou.com	2
☐	3	Tom	28	test3@baomidou.com	1
☐	4	Sandy	21	test4@baomidou.com	1
☐	5	Billie	24	test5@baomidou.com	1
☐	6	子慕	22	test@lagou.cn	1
☐	7	子慕	18	test@lagou.cn	1
☐	8	应颜	35	liubei@lagou.cn	1
☐	9	冰冰	30	test@lagou.cn	1
*	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)

10. 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。

10.1、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;
```

10.2、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
```

10.3、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
```

id	name	age	email	version	deleted
1	Jone	18	test1@baomidou.com	1	0
2	Jack	30	test2@baomidou.com	2	1
3	Tom	28	test3@baomidou.com	1	0
4	Sandy	21	test4@baomidou.com	1	0
5	Billie	24	test5@baomidou.com	1	0
6	子慕	22	test@lagou.cn	1	0
7	子慕	18	test@lagou.cn	1	0
8	应颜	35	liubei@lagou.cn	1	0
9	冰冰	30	test@lagou.cn	1	0
(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	0

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
```

可见，已经实现了逻辑删除。

###

11. 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。

11.1、创建工程

pom.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="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>
```

11.2、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;
```

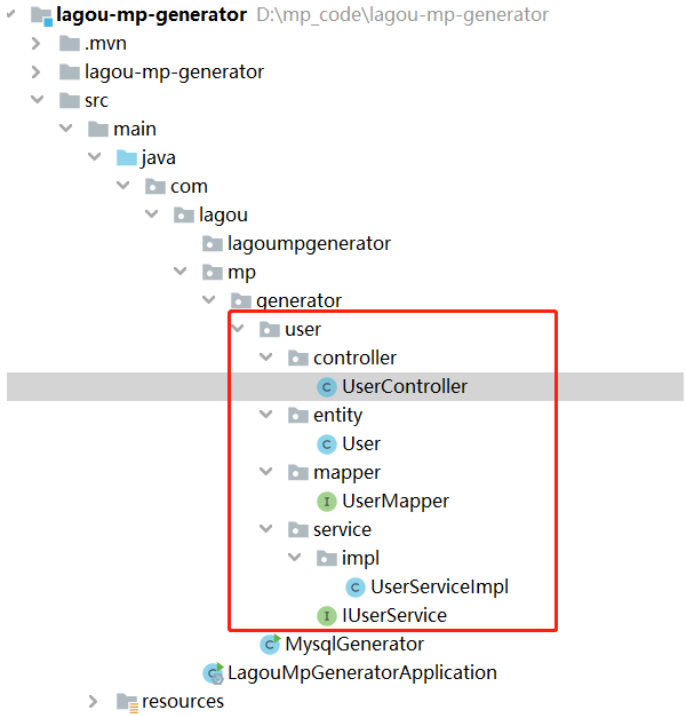
```
            }
        }
        throw new MybatisPlusException("请输入正确的" + tip + "! ");
    }

    /**
     * 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();
    }

}
```

11.3、测试

```
MysqlGenerator
请输入模块名：
user
请输入表名：
user
17:51:12.077 [main] DEBUG com.baomidou.mybatisplus.generator.AutoGenerator - ==========准备生成文件...=======
17:51:12.508 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 创建目录： [D:\mp_code\lagou-mp
17:51:12.509 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 创建目录： [D:\mp_code\lagou-mp
17:51:12.510 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 创建目录： [D:\mp_code\lagou-mp
17:51:12.510 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 创建目录： [D:\mp_code\lagou-mp
17:51:12.511 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 创建目录： [D:\mp_code\lagou-mp
log4j:WARN No appenders could be found for logger (freemarker.cache).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
17:51:12.624 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 模板:/templates/mapper.xml.ftl;
17:51:12.731 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 模板:/templates/entity.java.ftl
17:51:12.735 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 模板:/templates/mapper.java.ftl
17:51:12.739 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 模板:/templates/service.java.ft
17:51:12.743 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 模板:/templates/serviceImpl.jav
17:51:12.748 [main] DEBUG com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine - 模板:/templates/controller.java
17:51:12.748 [main] DEBUG com.baomidou.mybatisplus.generator.AutoGenerator - ==========文件生成完成！！！====
```

代码已生成：


lagou-mp-generator D:\mp_code\lagou-mp-generator
.mvn
lagou-mp-generator
src
main
java
com
lagou
lagoumpgenerator
mp
generator
user
controller
UserController
entity
User
mapper
UserMapper
service
impl
UserServiceImpl
IUserService
MysqlGenerator
LagouMpGeneratorApplication
resources

实体对象：

```java
@Accessors(chain = true)
public class TbUser implements Serializable {

    private static final long serialVersionUID = 1L;

    /**
     * 用户名
     */
    private String userName;

    /**
     * 密码
     */
    private String password;

    /**
     * 姓名
     */
    private String name;

    /**
     * 年龄
     */
    private Integer age;

    /**
     * 邮箱
     */
    private String email;
```

12. 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

```
*/
public interface UserMapper extends BaseMapper<User> {

    public void selectListBySql();
}
```

点击跳转到对应xml文件

```
<mapper namespace="com.lagou.mp.generator.user.mapper.UserMapper">

    <select id="selectListBySql">
    </select>
</mapper>
```

点击跳转到对应接口中

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

    public void selectListBySql(); ctrl + enter

}
```

- Generate statement
- Safe delete 'selectListBySql()'
- Make 'selectListBySql' private
- Make 'selectListBySql()' default
- [Mybatis] Generate @Param
- [Mybatis] Generate mapper of xml
- [Mybatis] Generate new statement
- Add Javadoc

xml中快速生成标签