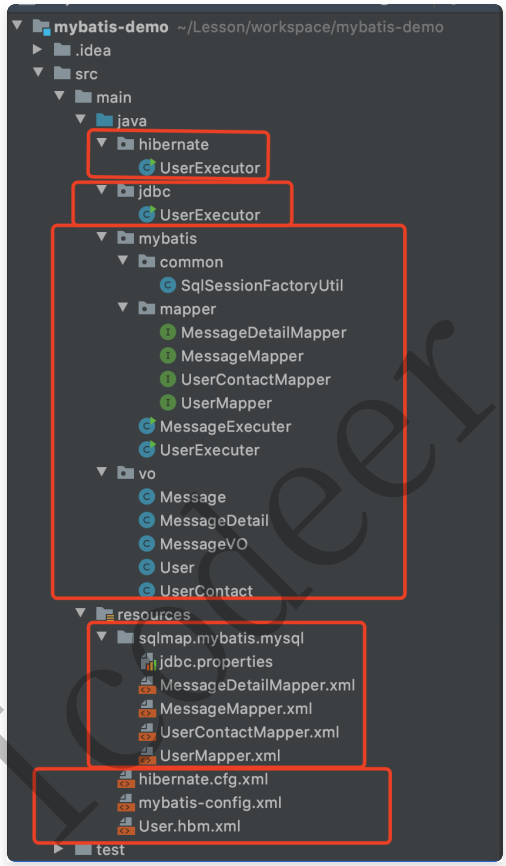
MyBatis应用篇

六、整体结构：

一、主流持久层技术框架

1.1> 前期准备

1.1.1> 项目结构图



1.1.2> 创建表

- 用户表 tb_user
- 用户联系方式表 tb_user_contact
- 消息表 tb_message
- 消息明细表 tb_message_detail

SQL | 复制代码

```sql
CREATE TABLE `tb_user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `name` varchar(255) NOT NULL DEFAULT '' COMMENT '姓名',
  `age` int(11) NOT NULL DEFAULT '-1' COMMENT '年龄',
  PRIMARY KEY (`id`),
  INDEX `index_name` (`name`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='用户信息表';

CREATE TABLE `tb_user_contact` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `user_id` bigint(20)  NOT NULL DEFAULT '-1' COMMENT '用户id',
  `contact_type` tinyint(4) NOT NULL DEFAULT '0' COMMENT '联系方式类型 0:未知 1:手机 2:邮箱 3:微信 4:QQ',
  `contact_value` varchar(255) NOT NULL DEFAULT '' COMMENT '联系方式信息',
  `create_time` DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '创建时间',
  `update_time` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`),
  INDEX `index_user_id` (`user_id`) USING BTREE,
  INDEX `index_contact_value` (`contact_value`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='联系方式信息表';

CREATE TABLE `tb_message` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `msg_id` varchar(255) NOT NULL DEFAULT '' COMMENT '消息ID',
  `status` int(11) NOT NULL DEFAULT '-1' COMMENT '消息状态，-1-待发送，0-发送中，1-发送失败 2-已发送',
  `content` varchar(255)  NOT NULL DEFAULT '' COMMENT '消息内容',
  `deleted` tinyint(4) NOT NULL DEFAULT '0' COMMENT '是否删除 0-未删除  1-已删除',
  `create_time` DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '创建时间',
  `update_time` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`),
  INDEX `index_msg_id` (`msg_id`) USING BTREE,
  INDEX `index_create_time` (`create_time`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='消息表';

CREATE TABLE `tb_message_detail` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增主键',
  `msg_id` varchar(255) NOT NULL DEFAULT '' COMMENT '消息ID',
  `detail_content` varchar(255)  NOT NULL DEFAULT '' COMMENT '详细消息内容',
```

```
38    `create_time` DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT
    '创建时间',
39    `update_time` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CUR
    RENT_TIMESTAMP COMMENT '更新时间',
40    PRIMARY KEY (`id`),
41    INDEX `index_msg_id` (`msg_id`) USING BTREE
42  ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='消息表详情表'
    ;
```

1.1.3> 创建实体类

【一对多关系】

- 用户实体类 User.java
- 用户联系方式类 UserContact.java

【一对一关系】

- 消息类Message.java
- 消息详情类MessageDetail.java

```java
public class User {
    private Long id;
    private String name;
    private Integer age;
    private List<UserContact> userContacts; // 联系方式列表
        ... 省略getXxx()方法和setXxx()方法...
}

public class UserContact {
    private Long id;
    private Long userId;
    private Integer contactType; // 联系方式类型 0:未知 1:手机 2:邮箱 3:微信 4:QQ
    private String contactValue; // 联系方式具体值
    private Date createTime;
    private Date updateTime;
        ... 省略getXxx()方法和setXxx()方法...
}

public class Message {
    private Long id;
    private String msgId; // 消息状态，-1-待发送，0-发送中，1-发送失败 2-已发送
    private Integer status;
    private String content; // 消息内容
    private Integer deleted;
    private Date createTime;
    private Date updateTime;
    private MessageDetail messageDetail;
     ... 省略getXxx()方法和setXxx()方法...
}

public class MessageDetail {
    private Long id;
    private String msgId;
    private String detailContent;
    private Date createTime;
    private Date updateTime;
    ... 省略getXxx()方法和setXxx()方法...
}
```

1.1.4> 开启日志 @Slf4j

- pom.xml

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.16.20</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.25</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.25</version>
</dependency>
```


1.2> JDBC

1.2.1> 概述

- 使用JDBC的五个步骤：

> 1> 注册驱动和数据库信息。
> 2> 获得Connection，并使用它打开Statement对象。
> 3> 通过Statement对象执行SQL语句，并获得结果对象ResultSet。
> 4> 通过代码将ResultSet对象转化为POJO对象。
> 5> 关闭数据库资源。

- 缺点：

> 1> 代码量很大，麻烦。
> 2> 需要我们对异常进行正确捕获并关闭链接。

1.2.2> 示例

- pom.xml

```xml
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.21</version>
</dependency>
```

- UserExecutor.java

Java | 复制代码

```java
public class UserExecutor {
    public static void main(String[] args) throws Throwable {
        /** 1> 加载驱动程序 mysql-connector-java 6及以上 */
        Class.forName("com.mysql.cj.jdbc.Driver");

        /** 2>获得Connection，并使用它打开Statement对象 */
        Connection connection = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/muse?useSSL=false", "root", "root");

            /** 3>通过Statement对象执行SQL语句，并获得结果对象ResultSet */
            ps = connection.prepareStatement("select name, age from tb_user");
            rs = ps.executeQuery();

            /** 4>通过代码将ResultSet对象转化为POJO对象 **/
            while (rs.next()) {
                System.out.println(String.format("姓名：%s, 年龄：%d", rs.getString("name"), rs.getInt("age")));
            }

            /** 附加：参数查询带 */
            ps = connection.prepareStatement("select id, msg_id, status, content, deleted, create_time, update_time "
                    + "from tb_message where id = ?");
            ps.setInt(1, 1);
            rs = ps.executeQuery();
            while (rs.next()) {
                System.out.println(String.format("消息id：%s, 状态：%d, 消息内容：%s",
                        rs.getString("msg_id"), rs.getInt("status"), rs.getString("content")));
            }
        } finally {
            /** 5>关闭数据库资源 **/
            close(connection, ps, rs);
        }
    }

    private static void close(Connection connection, PreparedStatement ps, ResultSet rs) throws Throwable{
```

1.3> Hibernate

1.3.1> 概述

- 优点：

> 1> 将映射规则分离到XML/注解中，减少了代码的耦合度。
> 2> 无需管理数据库连接，只需配置相应的XML。
> 3> 一个会话，只需要操作Session对象即可。
> 4> 关闭资源，只关闭Session即可。

- 缺点：

> 1> 全表映射不便利，更新时需要发送所有字段。
> 2> 无法根据不同的条件组装不同的SQL。
> 3> 对于多表关联和复杂SQL查询支持较差，需要自己写SQL；返回后，需要自己将数据组装为POJO。
> 4> HQL性能较差，无法优化SQL。
> 5> 不能有效支持存储过程。

1.3.2> 示例：

- pom.xml

XML | 复制代码

```xml
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.21</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-agroal</artifactId>
    <version>5.4.18.Final</version>
    <type>pom</type>
</dependency>
```

【注】一定要有<type>pom</type>，否则报错：

Plain Text | 复制代码

```
org.hibernate.HibernateException: java.lang.IllegalArgumentException: max size attribute is mandatory
```

- resources/hibernate.cfg.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- 配置连接MySQL数据库的基本参数 -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/muse?useSSL=false&amp;zeroDateTimeBehavior=CONVERT_TO_NULL</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>
        <!-- 数据库方言 MySQL -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- 输出打印SQL语句 -->
        <property name="hibernate.show_sql">true</property>
        <!-- 格式化SQL语句 -->
        <property name="hibernate.format_sql">true</property>
        <!-- 加载映射文件  -->
        <mapping resource="User.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

【注】"`hibernate.connection.url`"配置的url，多参数时，用&分割。

- resources/User.hbm.xml

XML | 复制代码

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <!-- 建立类与表的映射 -->
    <class name="vo.User" table="tb_user">
        <!-- 建立类中的属性与表中的主键相对应 -->
        <id name="id" column="id">
            <!-- 主键的生成策略-->
            <generator class="native" />
        </id>

        <!-- 建立类中的普通属性和表中的字段相对应 -->
        <property name="name" column="name" />
        <property name="age" column="age" />
    </class>
</hibernate-mapping>
```

- UserExecutor.java

```java
/**
 * Hibernate示例
 */
public class UserExecutor {

    //保存用户的案例
    public static void main(String[] args) {
        Configuration configuration = new Configuration().configure("hibernate.cfg.xml");
        SessionFactory sessionFactory = configuration.buildSessionFactory();
        Session session = null;
        try {
            session = sessionFactory.openSession();
            User user = session.get(User.class, 2L);
            System.out.println("姓名: " + user.getName() + ", 年龄: " + user.getAge());
        } finally {
            if (session != null) {
                //7. 释放资源
                session.close();
                sessionFactory.close();
            }
        }

    }
}
```

1.4> MyBatis

1.4.1> 概述

- 优点

> 1> 可以配置动态SQL。
> 2> 可以对SQL进行优化，并通过配置来决定SQL的映射规则。
> 3> 支持存储过程。
> 4> 具有自动映射功能，在注意命名规则的基础上，无需在写映射规则。
> 5> MyBatis提供接口编程的映射器，只需要一个接口和映射文件便可以运行。
> 6> 与代码耦合度低。

1.4.2> 示例

- pom.xml

```xml
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.21</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.5</version>
</dependency>
```

- jdbc.properties

```
#Establishing SSL connection without server's identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established by default if explicit option isn't set. For compliance with existing applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server certificate verification.
#增加useSSL=false，防止报上面的警告

driver=com.mysql.jdbc.Driver
url=jdbc:mysql://127.0.0.1:3306/muse?useUnicode=true&characterEncoding=utf-8&useSSL=false&allowPublicKeyRetrieval=true
username=root
password=root
```

- resources/mybatis-config.xml

XML | 复制代码

```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>
    <properties resource="sqlmap/mybatis/mysql/jdbc.properties" />

    <settings>
        <setting name="logImpl" value="STDOUT_LOGGING"/> <!-- 打印查询语句 -->
        <setting name="autoMappingBehavior" value="PARTIAL"/> <!-- NONE PARTIAL FULL -->
        <setting name="mapUnderscoreToCamelCase" value="true"/> <!-- 配置驼峰转下划线 数据库中的下划线, 转换Java Bean中的驼峰 -->
    </settings>

    <!-- 别名 -->
    <typeAliases>
        <package name="vo"/>
    </typeAliases>

    <!-- 配置数据库环境 -->
    <environments default="dev">
        <environment id="dev">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 数据库厂商标识 -->
    <databaseIdProvider type="DB_VENDOR"/>

    <!-- mappers 映射器 -->
    <mappers>
        <mapper resource="sqlmap/mybatis/mysql/UserMapper.xml"/>
        <mapper resource="sqlmap/mybatis/mysql/UserContactMapper.xml"/>
        <mapper resource="sqlmap/mybatis/mysql/MessageMapper.xml"/>
        <mapper resource="sqlmap/mybatis/mysql/MessageDetailMapper.xml"/>
    </mappers>
</configuration>
```

- resources/sqlmap/mybatis/mysql/UserMapper.xml

```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="mybatis.mappring.UserMapper">
    <!-- id: 这条SQL的唯一表示   parameterType: 定义参数类型   resultType: 定义返回值类型 -->
    <select id="getUserById" parameterType="long" resultType="user">
        select id, name, age from tb_user where id = #{id}
    </select>
</mapper>
```

- UserMapper.java

```java
public interface UserMapper {
    User getUserById(@Param("id") Long id);
}
```

- SqlSessionFactoryUtil.java

```java
public class SqlSessionFactoryUtil {
    private static SqlSessionFactory ssf;

    public static SqlSession openSqlSession() throws Throwable {
        if (ssf == null) {
            synchronized(SqlSessionFactoryUtil.class) {
                if (ssf == null) {
                    ssf = new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("mybatis-config.xml"));
                }
            }
        }
        return ssf.openSession(true); // true: 自动提交事务
    }
}
```

- UserExecuter.java

```java
@Slf4j
public class UserExecuter {
    public static void main(String[] args) {
        SqlSession sqlSession = null;
        try {
            sqlSession = SqlSessionFactoryUtil.openSqlSession();
            UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
            User user = userMapper.getUserById(2L);
            System.out.println(String.format("姓名：%s，年龄：%d", user.getName(), user.getAge()));
        } catch (Throwable e) {
            System.out.println(e.getMessage());
            log.error("error", e);
        } finally {
            if (sqlSession != null) {
                sqlSession.close();
            }
        }
    }
}
```

二、MyBatis的使用

由于user表没有转驼峰字段（即：都是单词列），所以采用message来做实验。

2.1> select查询操作

2.1.1> 创建Message相关配置和类

- 第一步：创建Message实体类

  由于前期准备阶段，已经创建完毕，此处忽略。

- 第二步：创建MessageMapper

```java
public interface MessageMapper {
    Message getMessageById(@Param("id") Long id);
}
```

- 第三步：创建MessageMapper.xml 直接返回实体对象

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mybatis.mappring.MessageMapper">
    <sql id="allColumns">
        id, msg_id, status, content, deleted, create_time, update_time
    </sql>

    <select id="getMessageById" parameterType="long" resultType="message">
        select
            <include refid="allColumns"/>
        from
            tb_message
        where
            id = #{id}
    </select>
</mapper>
```

- 第四步：配置mybatis-config.xml

```xml
<mappers>
    ... ...
    <mapper resource="sqlmap/mybatis/mysql/MessageMapper.xml"/>
</mappers>
```

- 第五步：执行查询操作MessageExecuter.java

```java
@Slf4j
public class MessageExecuter {
    public static void main(String[] args) {
        SqlSession sqlSession = null;
        try {
            sqlSession = SqlSessionFactoryUtil.openSqlSession();
            MessageMapper messageMapper = sqlSession.getMapper(MessageMapper.class);
            Message message = messageMapper.getMessageById(2L);
            System.out.println(message);
        } catch (Throwable e) {
            log.error("error", e);
        } finally {
            if (sqlSession != null) {
                sqlSession.close();
            }
        }
    }
}
```

- 执行结果发现，**无法映射驼峰属性，可以映射单一的单词**。那么如何处理呢？ 我们来看**自动映射**。

```
Message{id=2, msgId='null', status=1, content='bbbb', deleted=0, createTime=null, updateTime=null, messageDetail=null}
```

2.1.2> 自动映射

2.1.2.1> 自动配置映射

- autoMappingBehavior包含三个值：

> - NONE：取消自动映射。
> - PARTIAL：只会自动映射，没有定义嵌套结果集映射的结果集。
> - FULL：会自动映射任意复杂的结果集（无论是否嵌套）。

- 例子：**配置mybatis-config.xml（必须配置mapUnderscoreToCamelCase=true，否则失效）**

```xml
<settings>
    <setting name="autoMappingBehavior" value="PARTIAL"/> <!-- PARTIAL是默认值 -->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

- 执行结果发现，已经可以正常将**下划线**的表列自动转换为**驼峰**的实体属性名。

```
Message{id=2, msgId='msg_2', status=1, content='bbbb', deleted=0, createTime=Sun Jun 27 02:15:32 CST 2021, updateTime=Sun Jun 27 02:15:32 CST 2021, messageDetail=null}
```

2.1.2.2> 手动配置映射

- 第一步：创建新的Message实体类MessageVO.java

```java
public class MessageVO {
    private Long idVo;
    private String msgIdVo; // 消息状态，-1-待发送，0-发送中，1-发送失败 2-已发送
    private Integer statusVo;
    private String contentVo; // 消息内容
    private Integer deletedVo;
    private Date createTimeVo;
    private Date updateTimeVo;
    private MessageDetail messageDetailVo;
     ... 省略getXxx()方法和setXxx()方法...
}
```

- 第二步：在MessageMapper中新增方法

```java
public interface MessageMapper {
    MessageVO getMessageVOById(@Param("id") Long id);
}
```

- 第三步：配置MessageMapper.xml的ResultMap

  直接返回ResultMap映射对象

- 配置步骤：

  - 使用 `<resultMap>` 标签配置映射关系

- 将<select>标签中的resultType修改为 `resultMap`

XML 复制代码

```xml
<resultMap id="messageVOResultMap" type="vo.MessageVO">
    <id column="id" property="idVo"/>
    <result column="msg_id" property="msgIdVo"/>
    <result column="status" property="statusVo"/>
    <result column="content" property="contentVo"/>
    <result column="deleted" property="deletedVo"/>
    <result column="create_time" property="createTimeVo"/>
    <result column="update_time" property="updateTimeVo"/>
</resultMap>

<sql id="allColumns">
    id, msg_id, status, content, deleted, create_time, update_time
</sql>

<select id="getMessageVOById" parameterType="long" resultMap="messageVOResultMap">
    select
        <include refid="allColumns"/>
    from
        tb_message
    where
        id = #{id}
</select>
```

【注意】<select ... resultMap="xxxx"> 一定不要用resultType！！ 否则查询失效！！！切记！！！

- 第四步：执行查询操作MessageExecuter.java

Java 复制代码

```java
... ...
MessageMapper messageMapper = sqlSession.getMapper(MessageMapper.class);
MessageVO messageVo = messageMapper.getMessageVOById(2L);
System.out.println("messageVo=" + messageVo);
... ...
```

输出如下：

Plain Text 复制代码

```
messageVo=MessageVO{idVo=2, msgIdVo='msg_2', statusVo=1, contentVo='bbbb', deletedVo=0, createTimeVo=Sun Jun 27 02:15:32 CST 2021, updateTimeVo=Sun Jun 27 02:15:32 CST 2021, messageDetailVo=null}
```

2.1.3> select的入参方式

2.1.3.1> 基础类型查询

- MessageMapper.java

```java
public interface MessageMapper {
    Message getMessageById(Long id);
}
```

- MessageMapper.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mybatis.mappring.MessageMapper">
    <sql id="allColumns">
        id, msg_id, status, content, deleted, create_time, update_time
    </sql>

    <select id="getMessageById" parameterType="long" resultType="message">
        select
            <include refid="allColumns"/>
        from
            tb_message
        where
            id = #{id}
    </select>
</mapper>
```

- MessageExecuter.java

```java
Message message = messageMapper.getMessageById(2L);
```

2.1.3.2> map类型查询

- MessageMapper.java

```java
public interface MessageMapper {
    Message getMessageByMap(Map<String, Object> params);
}
```

- MessageMapper.xml

```xml
<select id="getMessageByMap" parameterType="map" resultType="message">
    select
      <include refid="allColumns"/>
    from
      tb_message
    where
      id = #{id} and msg_id = #{msgId}
</select>
```

- MessageExecuter.java

```java
Map<String, Object> paramsMap = new HashMap();
paramsMap.put("id", 1L);
paramsMap.put("msgId", "msg_1");
Message message = messageMapper.getMessageByMap(paramsMap);
```

2.1.3.3> 注解方式传递参数

- MessageMapper.java

```java
public interface MessageMapper {
    Message getMessageByAnnotation(@Param("id") Long id, @Param("msgId") String msgId);
}
```

- MessageMapper.xml

```xml
<select id="getMessageByAnnotation" resultType="message">
    select
      <include refid="allColumns"/>
    from
      tb_message
    where
      id = #{id} and msg_id = #{msgId}
</select>
```

- MessageExecuter.java

```java
Message message = messageMapper.getMessageByAnnotation(1L, "msg_1");
```

2.1.3.4> JavaBean方式传递参数

- MessageMapper.xml

```xml
<select id="getMessageByMessage" parameterType="vo.Message" resultType="message">
    select
      <include refid="allColumns"/>
    from
      tb_message
    where
      id = #{id} and msg_id = #{msgId}
</select>
```

- MessageMapper.java

```java
public interface MessageMapper {
    Message getMessageByMessage(Message message);
}
```

- MessageExecuter.java

```java
Message param = new Message();
param.setId(1L);
param.setMsgId("1000");
Message message = messageMapper.getMessageByMessage(param);
```

【总结】

> - 使用Map传递参数
>
> 灵活度最高，但是会导致业务可读性的丧失，后续维护困难，实际工作中应该尽量避免使用这种方式
>
> - 使用@Param注解
>
> 如果参数 `<=5` 时，是最佳的传参方式，他比JavaBean更直观。但是如果参数多，那么会造成接口参数膨胀，可读性和维护性差。
>
> - 使用JavaBean

> 当参数个数 `>5` 时，建议采用这种方式。

2.2> insert插入操作

```xml
<sql id="insertAllColumns">
    msg_id, status, content, deleted, create_time
</sql>
```

2.2.1> 普通插入（无主键回填）

- MessageMapper.java

```java
public interface MessageMapper {
    int insert(Message message);
}
```

- MessageMapper.xml

```xml
<insert id="insert" parameterType="message" keyProperty="id">
    insert into tb_message(<include refid="insertAllColumns"/>) values (#{msgId}, #{status}, #{content},
      #{deleted}, #{createTime})
</insert>
```

- MessageExecuter.java

```java
Message message = new Message();
message.setMsgId("msg_4");
message.setStatus(1);
message.setContent("ccc");
message.setDeleted(0);
message.setCreateTime(new Date());
messageMapper.insert(message);
System.out.println("message = " + message);
```

输出结果中，id为null

```
message = Message{id=null, msgId='msg_4', status=1, content='ccc', deleted=0, createTime=Sat Jun 26 16:04:13 CST 2021, updateTime=null, messageDetail=null}
```

2.2.2> 主键回填插入——useGeneratedKeys="true"

- MessageMapper.xml

```xml
<insert id="insertAndGetIdBack" parameterType="message" keyProperty="id" useGeneratedKeys="true">
    insert into tb_message(<include refid="insertAllColumns"/>) values (#{msgId}, #{status}, #{content},
  #{deleted}, #{createTime})
</insert>
```

- MessageMapper.java

```java
public interface MessageMapper {
    int insertAndGetIdBack(Message message);
}
```

- MessageExecuter.java

```java
Message message = new Message();
message.setMsgId("msg_5");
message.setStatus(1);
message.setContent("ddd");
message.setDeleted(0);
message.setCreateTime(new Date());
messageMapper.insertAndGetIdBack(message);
System.out.println("message = " + message);
```

输出结果中，id不为空。

```
message = Message{id=8, msgId='msg_5', status=1, content='ddd', deleted=0, createTime=Sat Jun 26 16:13:33 CST 2021, updateTime=null, messageDetail=null}
```

2.3> update更新操作

- MessageMapper.xml

```xml
<update id="updateContentById">
    update tb_message set content=#{content} where id=#{id}
</update>
```

- MessageMapper.java

```java
int updateContentById(@Param("id") Long id, @Param("content") String content);
```

- MessageExecuter.java

```java
messageMapper.updateContentById(1L, "newContent");
```

2.4> delete删除操作

- MessageMapper.xml

```xml
<delete id="deleteById" parameterType="long">
    delete from tb_message where id = #{id}
</delete>
```

- MessageMapper.java

```java
int deleteById(@Param("id") Long id);
```

- MessageExecuter.java

```java
messageMapper.deleteById(28L);
// sqlSession.commit();
```

2.5> $与#的区别

2.5.1> 使用#方式

- MessageMapper.xml

```xml
<select id="getMessageByMsgId" resultMap="messageResult">
    select
      <include refid="allColumns"/>
    from
      tb_message
    where
      msg_id = #{msgId}
</select>
```

输出结果：

```
Preparing: select id, msg_id, status, content, deleted, create_time, update_time from tb_message where msg_id = ?
Parameters: 1001(String)
```

【结论】采用的是**预编译**的方式构建查询语句。

2.5.2> 使用$方式

- MessageMapper.xml

```xml
<select id="getMessageByMsgId" resultMap="messageResult">
    select
      <include refid="allColumns"/>
    from
      tb_message
    where
      msg_id = ${msgId}
</select>
```

输出结果：

```
Preparing: select id, msg_id, status, content, deleted, create_time, update_time from tb_message where msg_id = 1001
```

【结论】采用的是**值传递**的方式构建查询语句。

2.5.3> SQL注入

- MessageMapper.xml

```xml
<select id="getMessageByMsgId" resultType="message">
    select
        <include refid="allColumns"/>
    from
        tb_message
    where
        msg_id = #{msgId} <!-- 不会发生SQL注入 -->
   <!-- msg_id = ${msgId} 会发生SQL注入 -->
</select>
```

- MessageMapper.java

```java
List<Message> getMessageByMsgId(@Param("msgId") String msgId);
```

- MessageExecuter.java

```java
List<Message> messages = messageMapper.getMessageByMsgId("1001 or (select count(1) from tb_message) > 0 ");
System.out.println("messages = " + messages);
```

- 如果采用msg_id = #{msgId}，则输出如下：

```
==>  Preparing: select id, msg_id, status, content, deleted, create_time, update_time from tb_message where msg_id = ?
==> Parameters: 1001 or (select count(1) from tb_message) > 0 (String)
<==      Total: 0
messages = []
```

- 如果采用msg_id = ${msgId}，则输出如下：

```
==>  Preparing: select id, msg_id, status, content, deleted, create_time, update_time from tb_message where msg_id = 1001 or (select count(1) from tb_message) > 0
==> Parameters:
<==    Columns: id, msg_id, status, content, deleted, create_time, update_time
<==        Row: 1, msg_1, 1, newContent, 0, 2021-06-26 13:15:32, 2021-06-26 16:26:21
<==        Row: 2, msg_2, 1, bbbb, 0, 2021-06-26 13:15:32, 2021-06-26 13:15:32
<==        Row: 5, msg_3, 1, ccc, 0, 2021-06-26 03:02:43, 2021-06-26 16:02:43
<==        Row: 6, msg_4, 1, ccc, 0, 2021-06-26 03:04:13, 2021-06-26 16:04:13
<==        Row: 7, msg_4, 1, ccc, 0, 2021-06-26 03:13:33, 2021-06-26 16:13:33
<==      Total: 5
messages = [Message{id=1, msgId='msg_1', status=1, content='newContent', deleted=0, createTime=Sun Jun 27 02:15:32 CST 2021, updateTime=Sun Jun 27 05:26:21 CST 2021, messageDetail=null}, Message{id=2, msgId='msg_2', status=1, content='bbbb', deleted=0, createTime=Sun Jun 27 02:15:32 CST 2021, updateTime=Sun Jun 27 02:15:32 CST 2021, messageDetail=null}, Message{id=5, msgId='msg_3', status=1, content='ccc', deleted=0, createTime=Sat Jun 26 16:02:43 CST 2021, updateTime=Sun Jun 27 05:02:43 CST 2021, messageDetail=null}, Message{id=6, msgId='msg_4', status=1, content='ccc', deleted=0, createTime=Sat Jun 26 16:04:13 CST 2021, updateTime=Sun Jun 27 05:04:13 CST 2021, messageDetail=null}, Message{id=7, msgId='msg_4', status=1, content='ccc', deleted=0, createTime=Sat Jun 26 16:13:33 CST 2021, updateTime=Sun Jun 27 05:13:33 CST 2021, messageDetail=null}]

```

【注】将表中的所有数据，都输出了出来，造成了数据泄漏。

2.6> 使用map存储结果集

- MessageMapper.xml——指定**resultType="map"**

```xml
<select id="getMessageMapById" resultType="map">
    select
      <include refid="allColumns"/>
    from
      tb_message
    where
      id = #{id}
</select>
```

- MessageMapper.java

```java
Map getMessageMapById(@Param("id") Long id);
```

- MessageExecuter.java——返回值只能是Map

```java
Map messageMap = messageMapper.getMessageMapById(2L);
System.out.println("messageMap = " + messageMap);
```

2.7> 使用POJO存储结果集

- MessageMapper.xml——指定resultMap="xxxx"

```xml
<resultMap id="messageResult" type="vo.Message">
    <id column="id" property="id"/>
    <result column="msg_id" property="msgId"/>
    <result column="status" property="status"/>
    <result column="content" property="content"/>
    <result column="deleted" property="deleted"/>
    <result column="create_time" property="createTime"/>
    <result column="update_time" property="updateTime"/>
</resultMap>

<select id="getMessageById" resultMap="messageResult">
    select
        <include refid="allColumns"/>
    from tb_message where id = #{id}
</select>
```

- MessageExecuter.java——返回值是Message对象

```java
Message message = messageMapper.getMessageById(2L);
```

2.8> 级联

2.8.1> association：一对一关系

- MessageDetailMapper.java

```java
MessageDetail getMessageByMsgId(@Param("msgId") String msgId);
```

- MessageDetailMapper.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mybatis.mapper.MessageDetailMapper">
    <sql id="allColumns">
        id, msg_id, detail_content, create_time, update_time
    </sql>

    <select id="getMessageByMsgId" parameterType="string" resultType="messageDetail">
        select
            <include refid="allColumns"/>
        from
            tb_message_detail
        where
            msg_id = #{msgId}
    </select>
</mapper>
```

- mybatis-config.xml

```xml
<mappers>
    ... ...
    <mapper resource="sqlmap/mybatis/mysql/MessageDetailMapper.xml"/>
</mappers>
```

- MessageMapper.java

```java
Message getMessageAndMessageDetailById(@Param("id") Long id);
```

- MessageMapper.xml

```xml
<resultMap id="messageAndDetailResult" type="vo.Message">
    <id column="id" property="id"/>
    <result column="msg_id" property="msgId"/>
    <result column="status" property="status"/>
    <result column="content" property="content"/>
    <result column="deleted" property="deleted"/>
    <result column="create_time" property="createTime"/>
    <result column="update_time" property="updateTime"/>
    <association column="msg_id" property="messageDetail" select="mybatis.mapper.MessageDetailMapper.getMessageByMsgId"/>
</resultMap>

<select id="getMessageAndMessageDetailById" parameterType="long" resultMap="messageAndDetailResult">
    select
        <include refid="allColumns"/>
    from tb_message where id = #{id}
</select>
```

【注】<association select="Mapper的全路径名.方法名">

- UnionQryExecuter.java

```java
Message message = messageMapper.getMessageAndMessageDetailById(1L);
System.out.println("message = " + message);
```

- MessageMapper.xml——association：多个参数关联

```xml
<resultMap id="messageAndDetailResult1" type="vo.Message">
    <id column="id" property="id"/>
    <result column="msg_id" property="msgId"/>
    <result column="status" property="status"/>
    <result column="content" property="content"/>
    <result column="deleted" property="deleted"/>
    <result column="create_time" property="createTime"/>
    <result column="update_time" property="updateTime"/>
    <association column="{msgId=msg_id, detailContent=content}" property="messageDetail"
                 select="mybatis.mapper.MessageDetailMapper.getMessageByMsgIdAndContent"/>
</resultMap>

<select id="getMessageAndMessageDetailById1" parameterType="long" resultMap="messageAndDetailResult1">
    select
      <include refid="allColumns"/>
    from
      tb_message
    where
      id = #{id}
</select>
```

- MessageMapper.java

```java
Message getMessageAndMessageDetailById1(@Param("id") Long id);
```

- MessageDetailMapper.xml

```xml
<select id="getMessageByMsgIdAndContent" resultType="messageDetail">
    select
      <include refid="allColumns"/>
    from
      tb_message_detail
    where
      msg_id = #{msgId} and detail_content= #{detailContent}
</select>
```

- MessageDetailMapper.java

```java
 MessageDetail getMessageByMsgIdAndContent(@Param("msgId") String msgId, @Param("detailContent") String detailContent);
```

- UnionQryExecuter.java

```java
Message message = messageMapper.getMessageAndMessageDetailById1(1L);
System.out.println("message = " + message);
```

2.8.2> collection：一对多关系

- UserContactMapper.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mybatis.mapper.UserContactMapper">
    <sql id="allColumn">
        id, user_id, contact_type, contact_value, create_time, update_time
    </sql>

    <select id="getUserContactByUserId" parameterType="long" resultType="vo.UserContact">
        select
            <include refid="allColumn"/>
        from
            tb_user_contact
        where
            user_id = #{userId}
    </select>
</mapper>
```

- mybatis-config.xml

```xml
<mappers>
    ... ...
    <mapper resource="sqlmap/mybatis/mysql/UserContactMapper.xml"/>
</mappers>
```

- UserContactMapper.java

```java
public interface UserContactMapper {
    List<UserContact> getUserContactByUserId(@Param("userId") Long userId);
}
```

- UserMapper.xml

```xml
<resultMap id="userResult" type="vo.User">
    <id column="id" property="id"/>
    <result column="name" property="name"/>
    <result column="age" property="age"/>
    <collection property="userContacts" column="id" select="mybatis.mapper.UserContactMapper.getUserContactByUserId"/>
</resultMap>

<select id="getUserAndUserContactById" parameterType="long" resultMap="userResult">
    select
        <include refid="allColumn"/>
    from
        tb_user
    where
        id = #{id}
</select>
```

- UserMapper.java

```java
public interface UserMapper {
    User getUserAndUserContactById(@Param("id") Long id);
}
```

- UnionQryExecuter.java

```java
User user = userMapper.getUserAndUserContactById(1L);
System.out.println("user = " + user);
```

2.8.3> discriminator：鉴别器

略

三、缓存

3.1> 一级缓存

- MyBatis**默认开启一级缓存**，即：**同一个SqlSession对象**调用**同一个Mapper的方法**，如果没有声明需要刷新，并且**缓存没超时**的情况下，一般只执行一次SQL，其他的查询SqlSession都只会取出当前缓存的数据。如下所示：

Java | 复制代码

```java
public class CacheExecuter {
    public static void main(String[] args) {
        SqlSession sqlSession = SqlSessionFactoryUtil.openSqlSession();
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        User user1 = userMapper.getUserById(1L);
        System.out.println("----真实查询-----user1 = " + user1);
        User user2 = userMapper.getUserById(1L);
        System.out.println("----缓存查询-----user2 = " + user2);

        /**
         * 开启了新的sqlSession，则无法利用一级缓存。因为一级缓存是sqlSession之间隔离的。
         */
        sqlSession = SqlSessionFactoryUtil.openSqlSession();
        userMapper = sqlSession.getMapper(UserMapper.class);
        User user3 = userMapper.getUserById(1L);
        System.out.println("----真实查询-----user3 = " + user3);
    }
}
```

输入如下：

```
Created connection 1071097621.
Returned connection 1071097621 to pool.
Cache Hit Ratio [mapper.UserMapper]: 0.0
Opening JDBC Connection
Checked out connection 1071097621 from pool.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3fd7a715]
==>  Preparing: select id, name, age from tb_user where id = ?
==> Parameters: 1(Long)
<==    Columns: id, name, age
<==        Row: 1, muse1, 22
<==      Total: 1
----真实查询-----user1 = User{id=1, name='muse1', age=22, userContacts=null}
Cache Hit Ratio [mapper.UserMapper]: 0.0
----缓存查询-----user2 = User{id=1, name='muse1', age=22, userContacts=null}
Cache Hit Ratio [mapper.UserMapper]: 0.0
Opening JDBC Connection
Created connection 280265505.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@10b48321]
==>  Preparing: select id, name, age from tb_user where id = ?
==> Parameters: 1(Long)
<==    Columns: id, name, age
<==        Row: 1, muse1, 22
<==      Total: 1
----真实查询-----user3 = User{id=1, name='muse1', age=22, userContacts=null}

Process finished with exit code 0
```

3.2> 二级缓存

- 在**UserMapper.xml**中添加**<cache/>**标签。
- sqlSession.commit(); **当使用二级缓存的时候，只有调用了commit方法后才会生效。**
- POJO**必须实现Serializable接口**。

3.2.1> 具体操作

- CacheExecuter.java

```java
public class CacheExecuter {
    public static void main(String[] args) {
        SqlSession sqlSession = SqlSessionFactoryUtil.openSqlSession();
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

        User user1 = userMapper.getUserById(1L);
        System.out.println("----真实查询-----user1 = " + user1);

        sqlSession.commit(); //当使用二级缓存的时候，只有调用了commit方法后才会生效。

        User user2 = userMapper.getUserById(1L);
        System.out.println("----缓存查询-----user2 = " + user2);

        sqlSession = SqlSessionFactoryUtil.openSqlSession();
        userMapper = sqlSession.getMapper(UserMapper.class);
        User user3 = userMapper.getUserById(1L);
        System.out.println("----缓存查询-----user3 = " + user3);
    }
}
```

- UserMapper.xml

```xml
<cache/>
```

- User.java

```java
public class User implements Serializable
```

输入如下：

```
Created connection 1071097621.
Returned connection 1071097621 to pool.
Cache Hit Ratio [mapper.UserMapper]: 0.0
Opening JDBC Connection
Checked out connection 1071097621 from pool.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@3fd7a715]
==>  Preparing: select id, name, age from tb_user where id = ?
==> Parameters: 1(Long)
<==    Columns: id, name, age
<==        Row: 1, muse1, 22
<==      Total: 1
----真实查询-----user1 = User{id=1, name='muse1', age=22, userContacts=null}
Cache Hit Ratio [mapper.UserMapper]: 0.5
----缓存查询-----user2 = User{id=1, name='muse1', age=22, userContacts=null}
Cache Hit Ratio [mapper.UserMapper]: 0.6666666666666666
----缓存查询-----user3 = User{id=1, name='muse1', age=22, userContacts=null}
```

3.2.2> 配置缓存参数

- UserMapper.xml

```xml
<cache eviction="LRU" flushInterval="1000" size="1000" readOnly="true"/>
```

> eviction：缓存回收策略：
> LRU：最近最少使用，移除最长时间不用的对象。
> FIFO：先进先出，按对象进入缓存的顺序来移除它们。
> SOFT：软引用，移除基于垃圾回收器状态和软引用规则的对象。
> WEAK：弱引用，移除最长时间不用的对象。
> flushInterval：刷新间隔时间，单位为毫秒。如果不配置，那么当SQL被执行的时候才会去刷新缓存。
> size：引用数据，正整数，代表缓存最多可以存储多少个对象，不宜设置过大。否则会内存溢出。
> readOnly：只读。

3.2.3> 自定义缓存

- 我们可以通过实现org.apache.ibatis.cache.Cache接口，使用Redis，Memcache等缓存机制，来实现自定义缓存。使用方式：

UserMapper.xml

```xml
<cache type="com.muse.RedisCache"/>
```

四、动态SQL

4.1> if&test

- 最常用的判断语句。`if&test`属性用于**条件判断**的语句中。
- UserMapper.xml

```xml
<resultMap id="userResultMap" type="vo.User">
    <id column="id" property="id"/>
    <result column="name" property="name"/>
    <result column="age" property="age"/>
</resultMap>

<select id="getUserByUser" parameterType="vo.User" resultMap="userResultMap">
  select
    id, name, age
  from
    tb_user
  where 1=1
  <if test="id != null">
    and id = #{id}
  </if>
  <if test="name != null and name != ''">
    and name = #{name}
  </if>
  <if test="age != null">
    and age = #{age}
  </if>
</select>
```

- UserMapper.java

```java
public interface UserMapper {
    List<User> getUserByUser(User user);
}
```

- UserExecutor.java

```java
User userParam = new User();
userParam.setName("muse");
userParam.setId(1L);
List<User> user = userMapper.getUserByUser(userParam);
System.out.println("user = " + user);
```

4.2> choose、when、otherwise

- 相当于 `if-else if-else`
- UserMapper.xml

```xml
<select id="getUserByUser2" parameterType="vo.User" resultMap="userResultMap">
  select id, name, age
  from tb_user
  where 1=1
  <choose>
    <when test="id != null">
      and id = #{id}
    </when>
    <when test="name != null and name != ''">
      and name = #{name}
    </when>
    <otherwise>
      and age is not null
    </otherwise>
  </choose>
</select>
```

- UserMapper.java

```java
public interface UserMapper {
    List<User> getUserByUser2(User user);
}
```

- UserExecuter.java

```java
User userParam = new User();
userParam.setName("muse");
// userParam.setId(1L);
userParam.setAge(22);
List<User> user = userMapper.getUserByUser2(userParam);
System.out.println("user = " + user);
```

4.3> trim、where、set

4.3.1> where

- 我们可以通过标签，**避免去写where 1=1**。如下所示：
- UserMapper.xml

```xml
<select id="getUserByUser3" parameterType="vo.User" resultMap="userResultMap">
  select
    id, name, age from tb_user
  <where>
    <if test="id != null">
      and id = #{id}
    </if>
  </where>
</select>
```

- UserMapper.java

```java
public interface UserMapper {
    List<User> getUserByUser3(User user);
}
```

- UserExecuter.java——指定id

```java
User userParam = new User();
userParam.setId(1L);
List<User> user = userMapper.getUserByUser3(userParam);
System.out.println("user = " + user);
```

输入如下：

```
==>  Preparing: select id, name, age from tb_user WHERE id = ?
==> Parameters: 1(Long)
<==    Columns: id, name, age
<==        Row: 1, muse1, 22
<==      Total: 1
user = [User{id=1, name='muse1', age=22, userContacts=null}]
```

- UserExecuter.java——不指定id

```java
User userParam = new User();
// userParam.setId(1L);
List<User> user = userMapper.getUserByUser3(userParam);
System.out.println("user = " + user);
```

输入如下：

```
==>  Preparing: select id, name, age from tb_user
==> Parameters:
<==    Columns: id, name, age
<==        Row: 1, muse1, 22
<==        Row: 2, muse2, 24
<==      Total: 2
user = [User{id=1, name='muse1', age=22, userContacts=null}, User{id=2, name='muse2', age=24, userContacts=null}]
```

4.3.2> trim

- 有时候我们要**去掉一些特殊的SQL语法**，比如 `and` 、 `or` 。则可以使用trim元素。
- UserMapper.xml

```xml
<select id="getUserByUser4" parameterType="vo.User" resultMap="userResultMap">
  select id, name, age from tb_user
  <trim prefix="where" prefixOverrides="and">
    and id = #{id}
  </trim>
/select>
```

【解释】

> **prefix**：表示输出前缀语句“where”
> **prefixOverrides**：表示where后面的语句的前缀（第一个）“and”要清除掉

- UserMapper.java

```java
public interface UserMapper {
    List<User> getUserByUser4(User user);
}
```

- UserExecuter.java

```java
User userParam = new User();
userParam.setId(1L);
List<User> user = userMapper.getUserByUser4(userParam);
System.out.println("user = " + user);
```

4.3.3> set

- set元素会默认把**最后一个逗号去掉**。
- UserMapper.xml

```xml
<update id="updateUserByUser" parameterType="vo.User">
    update tb_user
    <set>
        <if test="name != null and name != ''">
            name = #{name},
        </if>
        <if test="age != null">
            age = #{age},
        </if>
    </set>
    where id = #{id}
</update>
```

- 也可以采用trim的方式：

```xml
<update id="updateUserByUser" parameterType="vo.User">
    update tb_user
    <trim prefix="set" suffixOverrides=",">
        <if test="name != null and name != ''">
            name = #{name},
        </if>
        <if test="age != null">
            age = #{age},
        </if>
    </trim>
    where id = #{id}
</update>
```

- UserMapper.java

```java
public interface UserMapper {
    int updateUserByUser(User user);
}
```

- UserExecuter.java

```java
User userParam = new User();
userParam.setId(1L);
userParam.setName("muse");
userParam.setAge(22);
userMapper.updateUserByUser(userParam);
```

4.4> foreach

- foreach语句用于**循环遍历**传入的集合数据。
- UserMapper.java

```java
public interface UserMapper {
    List<User> getUserByIds(@Param("idList") List<Long> idList);
}
```

- UserMapper.xml

```xml
<select id="getUserByIds" resultMap="userResultMap">
    select
      id, name, age
    from
      tb_user
    where
      id in
    <foreach collection="idList" index="index" item="id" open="(" separator="," close=")">
        #{id}
    </foreach>
</select>
```

- UserExecuter.java

```java
List<Long> ids = new ArrayList<>();
ids.add(1L);
ids.add(2L);
ids.add(3L);
ids.add(4L);
List<User> users = userMapper.getUserByIds(ids);
System.out.println("users = " + users);
```

【解释】

> - collection：传递进来的参数名称，可以是数组、List、Set等集合。
> - index：当前元素在集合的下标位置。
> - item：循环中当前的元素。
> - open和close：使用什么符号包装集合元素。
> - separator：每个元素的间隔符号。

4.5> concat & bind

- UserMapper.xml

```xml
<select id="getUserByName" parameterType="string" resultMap="userResultMap">
    <bind name="namePattern" value="'%' + name + '%'"/>
    select
        id, name, age
    from
        tb_user
    where
        name like #{namePattern}
    <!-- name like concat('%', #{name}, '%') -->
</select>
```

- UserMapper.java

```java
List<User> getUserByName(@Param("name") String name);
```

- UserExecuter.java

```java
List<User> users = userMapper.getUserByName("muse");
```

输出如下：

```
==>  Preparing: select id, name, age from tb_user where name like ?
==> Parameters: %muse%(String)
<==    Columns: id, name, age
<==        Row: 1, muse, 22
<==        Row: 2, muse2, 24
<==      Total: 2
```

五、动态代理

5.1> 反射

```java
public class Reflaction {
    public static void main(String[] args) throws Throwable {
        Class clazz = User.class;
        User user = (User) clazz.newInstance();
        Method method = clazz.getMethod("setName", String.class);
        method.invoke(user, "张三");
        System.out.println(user.getName()); // 输出：张三
    }
}
```

5.2> JDK动态代理

- JDK动态代理是需要提供接口，而MyBatis的Mapper就是一个接口，它采用的就是JDK动态代理。如下所示：
- MessageService.java

```java
public interface MessageService {
    void sendMessage();
}
```

- MessageServiceImpl.java

```java
public class MessageServiceImpl implements MessageService {
    public void sendMessage() {
        System.out.println("MessageServiceImpl.sendMessage");
    }
}
```

- JdkProxy.java

```java
public class JdkProxy<T> implements InvocationHandler {
    T target;

    public T getProxy(T target) {
        this.target = target;
        return (T) Proxy.newProxyInstance(target.getClass().getClassLoader(),
                                          target.getClass().getInterfaces(), this);
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("JDK动态代理拦截开始！");
        Object result =  method.invoke(target, args);
        System.out.println("JDK动态代理拦截结束！");
        return result;
    }
}
```

- Executer.java

```java
public class Executer {
    public static void main(String[] args) {
        JdkProxy<MessageService> jdkProxy = new JdkProxy();
        MessageService messageService = jdkProxy.getProxy(new MessageServiceImpl());
        messageService.sendMessage();
    }
}
```

5.3> CGLIB

- CGLIB不需要提供接口即可实现动态代理，当然，它也可以代理有接口的服务类。如下所示：
- PlayService.java

```
public class PlayService {
    public void play() {
        System.out.println("PlayService.play");
    }
}
```

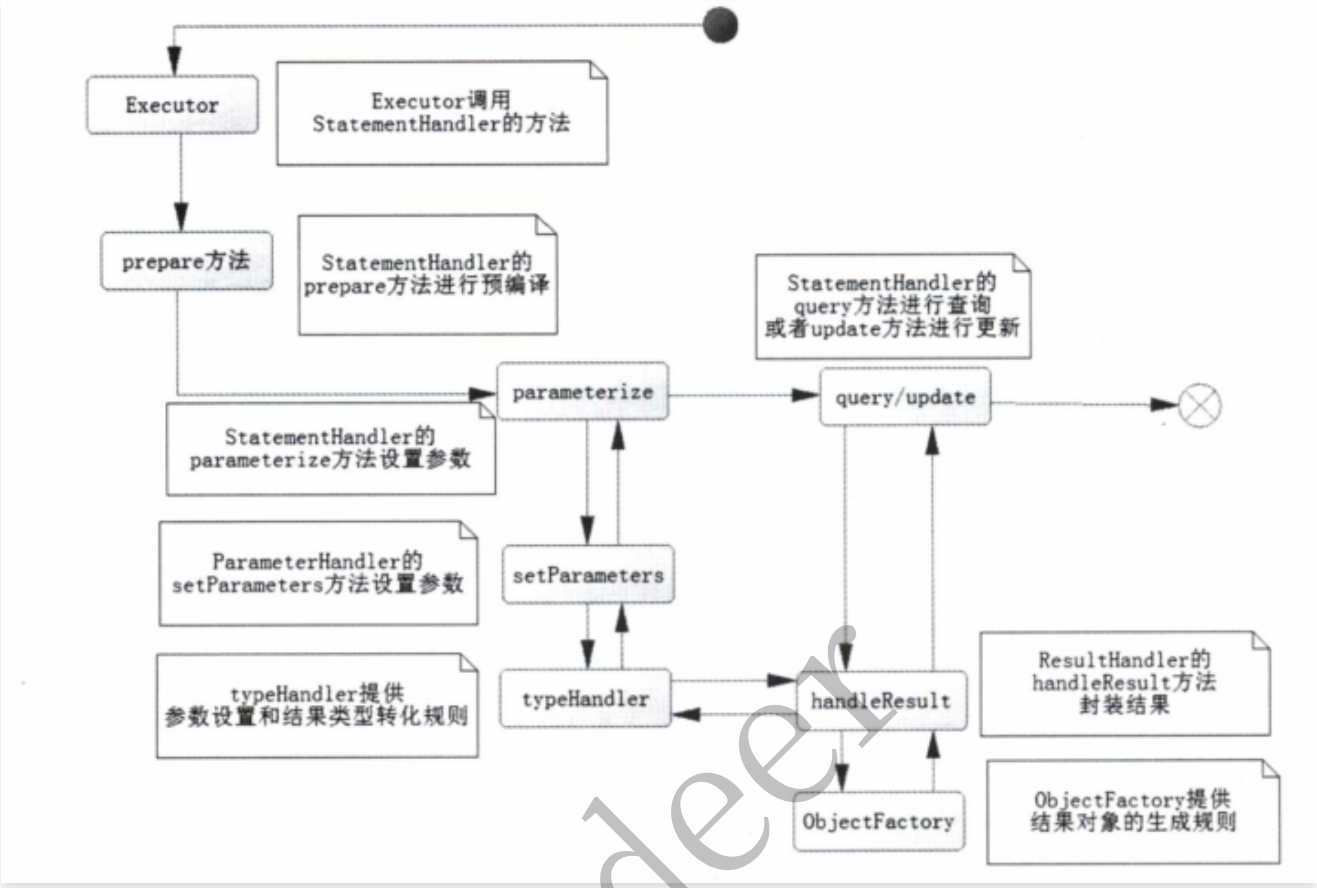
- CglibProxy.java

```java
public class CglibProxy<T> implements MethodInterceptor {
    T target;

    public T getProxy(T target) {
        this.target = target;
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(target.getClass());
        enhancer.setCallback(this);
        return (T) enhancer.create();
    }

    public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy)
      throws Throwable {
        System.out.println("CGLIB动态代理拦截开始!");
        Object result = methodProxy.invokeSuper(o, objects);
        System.out.println("CGLIB动态代理拦截结束!");
        return result;
    }
}
```

- Executer.java

```java
public class Executer {
    public static void main(String[] args) {
        CglibProxy<PlayService> cglibProxy = new CglibProxy();
        PlayService playService = cglibProxy.getProxy(new PlayService());
        playService.play();
    }
}
```

六、整体结构：



吾尝终日而思矣，不如须臾之所学也；
吾尝跂而望矣，不如登高之博见也。
登高而招，臂非加长也，而见者远；
顺风而呼，声非加疾也，而闻者彰。
假舆马者，非利足也，而致千里；
假舟楫者，非能水也，而绝江河。
君子生非异也，善假于物也。
------ 摘自《劝学》

愿本文可以成为大家的“山”、“风”、“马”、“舟”，助大家在技术之路上乘风破浪，大展宏图~~

同时，也欢迎大家关注我的公众号“爪哇缪斯” ~\(^o^)/~ 「干货分享，每天更新」