MyBatis 总结：

一、配置 MyBatis.xml

【configuration】标签

(1) properties?, ******属性配置

```
<properties>
    <property name="jdbc.driver" value="com.mysql.cj.jdbc.Driver"></property>
    <property name="jdbc.url"

    value="jdbc:mysql://localhost:3306/jdbc?useUnicode=true&amp;characterEncoding=utf8&amp;serverTim
    ezone=GMT%2B8&amp;useSSL=false"></property>
    <property name="jdbc.username" value="root"></property>
    <property name="jdbc.password" value="123456"></property>
</properties>
```

(2) settings?, *****全局配置（缓存 延迟加载）

```
<settings><!--设置全局延迟加载-->
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="aggressiveLazyLoading" value="false"/>
 </settings>
```

(3) typeAliases?, *******类型别名 给整个包的 pojo 起别名

```
<typeAliases>
    <package name="com.zyq.pojo"></package>
</typeAliases>
```

(4) typeHandlers?, *******类型转换

(5) objectFactory?,

(6) objectWrapperFactory?,

(7) plugins?, *********插件 eg：分页插件

(8)environments?, ********环境配置（数据源）

```
<environments default="mybatis"> 【数据库环境  可以配置多个环境 default：选择环境】
    <environment id="mybatis">【id：环境唯一标识】
        <transactionManager type="JDBC"/>  【事务管理 ：jdbc】
        <dataSource type="POOLED">      数据源（数据库连接池）
                                            dataSource：数据源类型
                                            POOLED：使用 MyBatis 自带数据源
                                            UNPOOLED：不使用 MyBatis 自带数据源
                                            JNDI（不常用）
            <property name="driver" value="${jdbc.driver}"/><!--ognl 表达式 ${jdbc.driver}-->
            <property name="url" value="${jdbc.url}"/>
            <property name="username" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
```

```
            </dataSource>
        </environment>
    </environments>
```

(9) databaseIdProvider?,

(10) mappers? *********引入映射配置文件

```
    <mappers>
        <mapper resource="mapper/UserMapper.xml"/>
        <mapper resource="mapper/AccountMapper.xml"/>
    </mappers>
```

二、Pojo 实体类

1、根据数据库表，配置 pojo 实体，属性名最好和列名相同；

2、提供 get/set 方法；

3、提供构造器（空构造器必须要）；

4、在 1-1 映射中 以查询表为中心 在表中加入需要查询的对象；

5、在 1-M 映射中 在中心表中需要加入一个 list 集合（存放需要查询的对象）

三、Mapper 接口

提供需要执行的方法

eg：

```
public List<Pojo> selectAll();

public int insertOne(Pojo pojo);

public List<Pojo> selectById(vo vo);//传递多个参数的时候 【第一种】需要写一个 vo 类 存放这几个参数再作为参数整体传过去

public List<Pojo> selectManyParm(String username,int start,int size);
```

四、Mapper.xml 映射器

1、配置映射器

```
<mapper namespace="com.zyq.mapper.MavenDao">  【namespace】接口的全限定名
    <select></select>
    <insert></insert>
    <update></update>
    <delete></delete>
    <resultMap id="result01" type="Pojo">
        <id property="" column=""></id>  id 配置主键的  property:属性名 eg：uid  column：列名 eg：id
        <result property="" column=""></result>
        *<association  property="pojo"  javaType="Pojo"  select="com.zyq.mapper.PojoMapper.findById"  column="u_id" fetchType="lazy"></association>  1-1 映射时候使用 【javaType】pojo 属性中包含另外一个的 pojo 属性的类型 【select】需要查询的方法的全限定名 【column】和另外一个表存在关系的列名（外键列） 【fetchType】延迟加载配置
```

*<collection property="accountList" ofType="account" column="uid" select="com.zyq.mapper.AccountMapper.findById" ></collection>【ofType】集合泛型类型

</resultMap>

<sql id="sel">select * from maven</sql> 如果需要多次使用一个 sql 语句 可以把它提取出来（类似封装一个 sql）然后在使用时候使用 include 标签 <include refid="sel"></include> 【调用】

</mapper>

2、标签内属性

id：接口方法名

parameterType：参数类型

-传递参数为 Pojo 时：#{写 pojo 属性名}

传递参数过多时候：写一个包含这些参数的 vo 数据类 传递参数 vo

-传递为基本数据类型时候：#{随便写}

-直接传递参数时候：#{param1}、#{param2}

-#{}和${}的区别：

（1）【当参数是简单数据类型的时候】

${}:里面必须写 value

#{}：随便写

（2）当里面是 pojo 类型（对象）

${} #{}：写属性名

（3） ${}:直接拼接 不会类型转化 不能防止 sql 注入

#{}:先类型转化 再拼接 相当于占位符? 可以防止 sql 注入

#{} -> 【好用】

-传递参数为 map 类型时候：#{写 map 的"key"}

resultType：当 pojo 属性名和列名完全相同的时候使用, 返回 pojo 类型

resultMap:当 pojo 属性名和列名有出入的时候使用

useGeneratedKeys="true" :设置开启主键返回（注：select 主键设置了自增）

keyProperty="" 返回的主键映射给哪个属性 eg： keyProperty="#{id}"（注：select 主键设置了自增）

3、动态 SQL

if where:

where 标签 只能自动处理第一个 and so 最好加上全都加上 and

eg: <select id="selectFindDynamicSQL" parameterType="Pojo" resultType="Pojo">

select * from maven

<where>

<if test="uname!=null">

and uname like "%"#{uname}"%"

</if>

<if test="uphone!=null">

and uphone like "%"#{uphone}"%"

</if>

```
            </where>

        </select>
```

foreach:

前提是参数需要循环

eg:

```
<delete id="deleteMore" parameterType="int[]">
    delete from maven
    <where>
```

【collrction：指定是 array（数组）还是 list（集合） open：循环开始 close：循环结束 separator：分割符 item:循环变量】

```
        <foreach collection="array" open=" uname in(" close=")" separator="," item="i">
            #{i}
        </foreach>
    </where>
</delete>
```

chaoose、when、otherwise：

【多种判断选择 （类似于 switch……case……default）】

【只能有一个语句被执行 如果需要全部条件都满足需要使用上面的 if 标签】

eg：

```
<select id="selByChoose" parameterType="Pojo" resultType="Pojo">
    <include refid="sel"></include>
    <where>
        <choose>
            <when test="uid!=null and uid!=''">
                and uid=#{uid}
            </when>
            <when test="uname!=null and uname!=''">
                and uname=#{uname}
            </when>
            <otherwise>
                and uphone  is not null
            </otherwise>
        </choose>
    </where>
</select>
```

Set：

和 update 搭配使用 好处：可以选择修改的内容

eg：

```
<update id="updBySet" parameterType="Pojo" >
    update maven
    <set>
        <if test="uname!=null and uname!=''">
            uname=#{uname},【注意逗号】
```

```
            </if>
            <if test="uphone!=null and uphone!=''">
                uphone=#{uphone}
            </if>
        </set>
        where uid=#{uid}
    </update>
```

五、动态代理开发

1、通过 SqlSessionFactoryBuilder 创建 SqlSessionFactory；

2、通过 SqlSessionFactory.openSession()创建 sqlSession；

3、sqlSession.getMapper 方法获取动态代理对象；

4、通过动态代理对象调用接口方法；

eg：

```
SqlSessionFactory factory;
    {
        try {
            factory=new SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("MyBatis.xml"));
        } catch (IOException e) {

        }
    }
    @Override
    public List<User> findAll() {
        SqlSession sqlSession = factory.openSession();
        UserMapper mapper = sqlSession.getMapper(接口.class);
        List<User> otm = mapper.接口方法();
        sqlSession.close();【必须关闭 sqlSession】
    }
```

六、缓存

一级缓存（默认开启）【一级缓存必须存在于同一个 sqlSession 下】：第二次查询相同 sql 的时候没有执行 sql 语句 说明啦一级缓存的存在

* 在执行增删改之后 缓存会清空---或者使用 sqlSession.clearCache();也可以清空缓存

*

过程：第一次查询 sql 语句 将查询语句和查到的数据存放在一级缓存中；---sql 语句为“键” ---数据为“值” 存入一个 map

* 第二次查询 sql 先查找一级缓存中是否存在 sql 语句 如果存在（直接返回） 如果没有再查询数据库

二级缓存：是 sqlSessionFactory 级别的，同一个 sqlSessionFactory 共享

*

*　　配置流程 1、配置 MyBatis.xml 【<setting name="cacheEnabled" value="true"/>】

*　　2、配置 mapper 映射器： 【<cache></cache>】

*　　3、Pojo 对象实现序列化接口 【public class User implements Serializable 】

*

*

*　　流程：在任意的 sqlSession 对象中进行 sql 查询 在【sqlSession 关闭】时候在二级缓存中保存 --以 namespace.sql 为“键”

*　　以对象为“值” 其他 sqlSession 对象查询时候 查询 namespace.sql 是否存在缓存

*　　大坑：在表于表关联查询时候 数据不准确

七、注解开发

sql 类型主要分成 : select@Select(${sql}), update@Update(${sql}), insert@Insert($sql), delete(${sql}).

@Select:

@Results 用来设置 table 信息与 bean 相关字段的映射关系， 每一个字段的关系使用 @Result 控制。

默认情况下对于每一 table 字段，例如 name, 会调用 bean 中的 setName(..). 如果找不到，对于新版本的 mybatis 会报错。

例如上面的 cluster_name 会调用 setCluster_name(). 但是 java 中使用的 clusterName,可以通过 Result 注解控制.

@ResultMap 可以通过 Id,应用其他的 Results

eg:

```
public interface ClusterMessageMapper {

    // Insert
    @Insert("insert into cluster_manager(cluster_name, cluster_time, cluster_address, cluster_access_user, cluster_access_passwd) " +
            "values(#{clusterName}, #{clusterTime}, #{clusterAddress}, #{clusterAccessUser}, #{clusterAccessPasswd})")
    @Options(useGeneratedKeys = true, keyColumn = "cluster_id", keyProperty = "clusterId")
    public void insertClusterMessage(ClusterMessage clusterMessage);

    // select
    @Select("select * from cluster_manager")
    @Results(
            id = "clusterMessage",
            value = {
                    @Result(column = "cluster_name", property = "clusterName", javaType = String.class, jdbcType = JdbcType.VARCHAR),
                    @Result(column = "cluster_time", property = "clusterTime", javaType = Long.class, jdbcType = JdbcType.BIGINT),
                    @Result(column = "cluster_address", property = "clusterAddress", javaType = String.class, jdbcType = JdbcType.VARCHAR),
                    @Result(column = "cluster_access_user", property = "clusterAccessUser", javaType = String.class, jdbcType = JdbcType.VARCHAR),
                    @Result(column = "cluster_access_passwd", property = "clusterAccessPasswd", javaType = String.class, jdbcType = JdbcType.VARCHAR)
```

```
            }
    )
    public List<ClusterMessage> getClusterMessage();

    @Select("select * from cluster_manager")
    @MapKey("clusterId")
    public Map<Integer, ClusterMessage> getClusterMessageMapper();


    @Select("select * from cluster_manager where cluster_id=#{clusterId}")
    @ResultMap("clusterMessage")
    public ClusterMessage getClusterMessageById(@Param("clusterId") int clusterId);


    // update
    @Update("update cluster_manager set cluster_name=#{clusterName} where cluster_id=#{clusterId}")
    public void updateClusterMessage(ClusterMessage clusterMessage);

    // delete
    @Delete("delete from cluster_manager where cluster_id=#{clusterId}")
    public void deleteClusterMessage(@Param("clusterId")int clusterId);
}
```