尚马教育 JAVA 课程

Mybatis 高级

文档编号：C02

创建日期： 2017-07-07

最后修改日期：2021-01-18

版 本 号：V3.5

电子版文件名：尚马教育-第三阶段-2.mybatis 高级.docx

文档修改记录：

更新日期	更新作者	更新说明	版本号
2017-07-30	张元林	初始版本	V1.0
2018-08-01	王绍成	Mybatis 版本更新	V2.0
2019-08-09	冯勇涛	课件格式以及课程深度加深	V3.0
2021-01-18	冯勇涛	深化内容，突出重点	V3.5

目录

本章节学习内容：

深入学习 mybatis 映射文件中的各个标签用法；

理解静态代理与动态代理思想；

了解 mybatis 数据延迟加载思想；

了解 mybatis 数据查询缓存思想；

1. Sql 标签

Sql 标签作用：可被其他语句引用的可重用语句块。

实际使用中通常把查询字段包装为 sql 语句块。

新建映射文件 sysuserMapper.xml

```
<sql id="allFields">
    uid,uname,uphone,upwd,uemail,create_time,update_time
</sql>
<select id="selectUserByKey" parameterType="int"
resultType="Sysuser">
    select
     <include refid="allFields"></include>
     from sysuser where uid=#{uid}
</select>
```

2. resultMap 标签进行属性名映射

2.1. 使用场景

ResultMap 标签使用起来偏复杂，用来对实体对象的属性与数据库字段名做映射，同时提供子标签做对象之间的关系映射,进行级联查询操作。

ResultMap 的设计思想是，对于简单的语句不需要配置显式的结果映射(属性名与字段名一致)，而对于复杂一点的语句只需要描述它们的关系就行了。

2.2. resultMap 示例

```
<resultMap type="com.javasm.sys.entity.Sysuser" id="sysuserResultMap">
        <id column="uid" property="uid"/><!-- 对主键列做映射 -->
        <result column="uname" property="uname"/>
        <result column="uphone" property="uphone"/>
        <result column="user_address" property="uaddress"/>
        <result column="create_time"
property="createTime"></result>
        <result column="update_time"
property="updateTime"></result>
    </resultMap>
```

2.3. 应用 resultMap

```
<!--select 标签的 resultMap 属性的值对应 resultMap 标签的 id-->
<select id="selectUserById" parameterType="int"
resultMap="sysuserResultMap">
        select * from sysuser where uid=#{uid}
</select>
```

注意点：

注意点 1：resultMap 标签的 id 子标签用来进行主键列映射；result 子标签进行非主键列映

射；

注意点 2：数据库字段与实体类属性同名的字段可以不映射,不过一般都配置。

注意点 3：select 标签的 resultMap 属性与 resultType 属性不能同时出现，resultType 属性的值一般是实体类，map 或简单类型；resultMap 属性的值必须是 resultMap 标签的 id 值。

3. resultMap 标签进行对象关系映射

在实际开发中，经常出现多表联接查询的情况，使用 sql 交叉，左右联接，子查询等复杂的 sql 语句，mybatis 对类似的复杂 sql 语句查询结果进行封装。

3.1. 多对一持有关系

用户与角色对象之间维护成用户持有角色关系，查询用户时，把用户关联的角色信息同步查询。

3.1.1. 创建实体类与数据库表

```
public class Sysrole {
    private Integer rid;
    private String rname;
    private String createTime;
    private String updateTime;
    // TODO setter,getter
}

public class Sysuser {
  private Integer uid;
  private String uname;
  private String upwd;
  private String uphone;
  private String uemail;
  private String createTime;
  private String updateTime;
  private Integer rid;
  private Sysrole srole;//对象持有关系
```

```
    //TODO setter,gettter
}
```

3.1.2. 实现思路 1:手工进行多次查询

手工进行两次 sql 查询，第一次查询用户，第二次查询角色，把查询结果组装。

① 创建 SysuserMapper.java 与 sysuserMapper.xml

```
public interface SysuserMapper {
    public Sysuser selectUserByKey(Integer uid);
}
```

```
<select id="selectUserByKey" parameterType="int"
resultType="Sysuser">
    select * from sysuser where uid=#{uid}
</select>
```

② 创建 SysroleMapper.java 与 sysroleMapper.xml

```
public interface SysroleMapper {
    public Sysrole selectRoleByRid(Integer rid);
}
```

```
<select id="selectRoleByRid" parameterType="int"
resultType="sysrole">
    select * from sysrole where rid=#{rid}
</select>
```

③ 测试类

```
@Test
public void test1_selectUserAndRole2(){
    Integer uid=1;
    SysuserMapper mapper =
session.getMapper(SysuserMapper.class);
```

```
    SysroleMapper roleMapper =
session.getMapper(SysroleMapper.class);
//第一次查询用户
    Sysuser suser = mapper.selectUserByKey(uid);
//第二次查询角色
    Sysrole sysrole =
roleMapper.selectRoleByRid(suser.getRid());
//组装对象
    suser.setSrole(sysrole);
    System.out.println(suser);
}
```

3.1.3. 实现思路 2:association 标签进行多次查询

第二种思路本质仍然是第一种思路，但是二次查询由 mybaits 底层实现，并进行查询结果的组装。

① 在 sysuserMapper.xml 映射文件中添加 selectUserAndRoleByUserKey

```
<resultMap id="userAndRoleMap" type="sysuser">
    <id column="uid" property="uid"></id>
    <result column="uname" property="uname"></result>
    <result column="upwd" property="upwd"></result>
    <result column="create_time" property="createTime"></result>
    <result column="update_time" property="updateTime"></result>
    <association  property="srole" column="rid" javaType="Sysrole"
select="com.javasm.sys.mapper.SysroleMapper.selectRoleByRid"></as
sociation>
</resultMap>
<select id="selectUserAndRoleByUserKey" parameterType="int"
resultMap="userAndRoleMap">
```

```
  select * from sysuser where uid=#{uid}
</select>
```

映射文件注意点：

注意点 1：association 是 resultMap 的子标签，用来做持有关系映射。

注意点 2：association 标签的 column 属性是二次查询需要的外键列名，property 属性是属性名；javaType 是属性类型；select 是二次查询的 namespace.id

② 测试类

```
@Test
public void test1_selectUserAndRole(){
    Integer uid=1;
    SysuserMapper mapper = session.getMapper(SysuserMapper.class);
    Sysuser user1 = mapper.selectUserAndRoleByUserKey(uid);//使用mybatis 的二次关联查询
    System.out.println(user1);
    System.out.println(user1.getSrole());
}
```

3.1.4. 实现思路 3:association 标签对级联 sql 查询结果映射

① 在 sysuserMapper.xml 和 SysuserMapper.java 中添加代码

```
<resultMap id="userAndRoleMap2" type="sysuser">
    <id column="uid" property="uid"></id><!--对主键列做映射-->
    <result column="uname" property="uname"></result><!--对非主键列做映射-->
    <result column="upwd" property="upwd"></result><!--对非主键列做映射-->
    <result column="create_time"
property="createTime"></result><!--对非主键列做映射-->
    <result column="update_time"
property="updateTime"></result><!--对非主键列做映射-->
```

```
    <result column="rid" property="rid"></result>
    <association  property="srole" javaType="Sysrole">
        <id column="rid" property="rid"></id>
        <result column="rname" property="rname"></result>
        <result column="create_time"
property="createTime"></result>
        <result column="update_time"
property="updateTime"></result>
    </association>
</resultMap>

<select id="selectUserAndRoleByUserKey2" parameterType="int"
resultMap="userAndRoleMap2">
  select u.*,r.rname from sysuser u,sysrole r where u.rid=r.rid
and u.uid=#{uid}
</select>
```

注意点 1：selectUserAndRoleByUserKey2 的 sql 语句是表交叉查询。

注意点 2：association 标签不需要写 column 与 select 属性进行二次查询。

② 执行测试

```
@Test
public void test1_selectUserAndRole(){
    Integer uid=1;
    SysuserMapper mapper =
session.getMapper(SysuserMapper.class);
    Sysuser user2 =
mapper.selectUserAndRoleByUserKey2(uid);//mysql 数据库表的交叉连接查
询，把查询结果进行结果集映射。
    System.out.println(user2);
```

```
    System.out.println(user2.getSrole());
}
```

3.2. 一对多聚合关系

角色持有用户对象的集合，称为聚合关系，要求查询角色时，对该角色下的用户集合进行级联查询。

3.2.1. 修改实体类

```
public class Sysrole {
    private Integer rid;
    private String rname;
    private String createTime;
    private String updateTime;
    private List<Sysuser> userList;//对象聚合关系
}
```

3.2.2. 实现思路 1:手工进行多次查询

① 创建 SysuserMapper.java 与 sysuserMapper.xml

```
public interface SysuserMapper {
public List<Sysuser> selectUsersByRoleId(Integer rid);
}
```

```
<select id="selectUsersByRoleId" parameterType="int"
resultType="sysuser">
    select * from sysuser where rid=#{rid}
</select>
```

② 创建 sysroleMapper.java 与 sysroleMapper.xml

```
public interface SysroleMapper {
    public Sysrole selectRoleByRid(Integer rid);
}
```

```
<select id="selectRoleByRid" parameterType="int"
resultType="sysrole">
    select * from sysrole where rid=#{rid}
</select>
```

③ 测试

```
@Test
public void test1_selectRoleAndUsers(){
    Integer rid=1;
    SysroleMapper mapper =
session.getMapper(SysroleMapper.class);
    SysuserMapper userMapper=
session.getMapper(SysuserMapper.class);
    //一次查询，查询角色对象
    Sysrole sysrole = mapper.selectRoleByRid(rid);
    //二次查询，查询用户集合
    List<Sysuser> userList =
userMapper.selectUsersByRoleId(rid);
    //组装对象
    sysrole.setUserList(userList);
    System.out.println(sysrole);
}
```

3.2.3. 实现思路 2:collection 标签进行多次查询

使用 collection 标签进行多次查询，本质仍然是两次单表查询，与思路 1 代码类似，代码交由 mybatis 底层完成。

① 在 sysroleMapper.xml 与 sysroleMapper.java 中添加代码

```
public Sysrole selectRoleAndUsersByRid(Integer rid);
```

```
<resultMap id="roleAndUserMap" type="sysrole">
    <id column="rid" property="rid"></id>
    <result column="rname" property="rname"></result>
    <result column="create_time" property="createTime"></result>
    <result column="update_time" property="updateTime"></result>
    <collection property="userList" column="rid"
ofType="sysuser"
select="com.javasm.sys.mapper.SysuserMapper.selectUsersByRoleId"
></collection>
</resultMap>

<select id="selectRoleAndUsersByRid" parameterType="int"
resultMap="roleAndUserMap">
    select * from sysrole where rid=#{rid}
</select>
```

注意点 1：collection 标签专用来做聚合关系查询，ofType 表示集合的泛型类型；select 表示二次查询的位置。

② 测试

```
@Test
public void test2_selectRoleAndUsers2(){
    Integer rid=1;
    SysroleMapper mapper =
session.getMapper(SysroleMapper.class);
    Sysrole sysrole = mapper.selectRoleAndUsersByRid(rid);
    System.out.println(sysrole);
}
```

3.2.4. 实现思路 3:collection 标签对级联 sql 查询结果映射

思路 3 是使用 sql 进行表交叉查询，一次查询出需要的所有数据，然后由 mybaits 进行结果映射。

① 映射文件与映射接口分别添加如下。

```
<resultMap id="roleAndUserMap2" type="sysrole">
    <id column="rid" property="rid"></id>
    <result column="rname" property="rname"></result>
    <result column="create_time" property="createTime"></result>
    <result column="update_time" property="updateTime"></result>
    <collection property="userList" ofType="sysuser" >
        <id column="uid" property="uid"></id>
        <result column="uname" property="uname"></result>
        <result column="upwd" property="upwd"></result>
        <result column="create_time"
property="createTime"></result>
        <result column="update_time"
property="updateTime"></result>
    </collection>
</resultMap>

<select id="selectRoleAndUsersByRid2" parameterType="int"
resultMap="roleAndUserMap2">
    select u.*,r.rname from sysrole r,sysuser u where
r.rid=u.rid and r.rid=#{rid}
</select>
```

```
public Sysrole selectRoleAndUsersByRid2(Integer rid);
```

注意：collection 子标签不需要 select 进行二次查询。

② 测试

```
@Test
public void test2_selectRoleAndUsers3(){
    Integer rid=1;
    SysroleMapper mapper =
session.getMapper(SysroleMapper.class);
    Sysrole sysrole = mapper.selectRoleAndUsersByRid2(rid);
    System.out.println(sysrole);
}
```

4. 动态 SQL 语句

所谓动态 sql 语句，即在 xml 映射文件中进行 sql 语句拼接。

4.1. If 与 where 标签

If 标签通常不单独使用，与 where 或 set 一起使用。

If 与 where 联合实现条件查询.

```
<select id="selectUsers" parameterType="Sysuser" resultType="Sysuser">
        Select  * from sysuser
       <where>
          <if test="uname != null and uname!=''">
             and uname like "%"#{uname}"%"
          </if>
          <if test="upwd != null and upwd!=''">
             and upwd=#{upwd}
          </if>
          <if test="uphone!= null and uphone!=''">
             and uphone=#{uphone}
```

```
        </if>
    </where>
</select>
```

注意点 1：多条件之间使用 and 或 or 运算符。

注意点 2：加上 uphone!=''是因为前端传递的查询参数可能为空字符串。

注意点 3：Where 标签自动生成 where 关键字，在遇到第一个字段是 AND 或 OR 时,会做自动去除的处理。如果 where 标签中间无内容，则不生成 where

4.2. if 与 Set 标签

Set 标签用在 update 标签中，来完成修改 SQL 语句中的逗号去除。

```
<update id="updateUser" parameterType="Sysuser">
    update sysuser
    <set>
        <if test="uname != null">
            uname=#{uname},
        </if>
        <if test="upwd != null">
            upwd=#{upwd},
        </if>
        <if test="uphone!= null">
            uphone=#{uphone},
        </if>
        <if test="uemail!= null">
            uemail=#{uemail},
        </if>
    </set>
    where uid=#{uid}
</update>
```

注意点 1：set 标签自动生成 set 关键字，在遇到最后一个逗号时,会做自动去除的处理。

4.3. Foreach 标签

批量删除或批量添加或批量查询等操作可以使用 foreach 标签实现

collection - 传递参数类型 list/array/自定义 key

index - 循环索引

item - 集合中的元素

open -以什么开始; close - 以什么结束

4.3.1. 批量删除

① 写法 1：

```
<delete id="delUsersByKeys" >
        delete from sysuser where uid in
        <foreach collection="array" open="(" close=")" separator="," item="userid">
          #{userid}
        </foreach>
    </delete>
```

```
public int delUsersByKeys(Integer[] uids);
```

注意：由于 mapper 接口中的方法形参是数组，mybatis 底层把数组对象封装到 Map 中，key 为”array”,value 为数组对象，因此 foreach 标签的 collection 写 array

② 写法 2

```
public int delUsersByKeys2(List<Integer> uids);
```

```
  <delete id="delUsersByKeys2" >
        delete from sysuser where uid in
        <foreach collection="list" open="(" close=")" separator="," item="userid">
            #{userid}
```

```
        </foreach>
    </delete>
```

注意：由于 mapper 接口中的方法形参是 List，mybatis 底层把数组对象封装到 Map 中，key 为”list”,value 为 List 集合对象，因此 foreach 标签的 collection 写 list

③ 写法 3

```
public int delUsersByKeys3(@Param(“uids”)List<Integer> uids);
```

```
<delete id="delUsersByKeys3" >
        delete from sysuser where uid in
        <foreach collection="uids" open="(" close=")" separator="," item="userid">
            #{userid}
        </foreach>
    </delete>
```

注意：由于 mapper 接口中的方法形参通过@Param 注解指定名称 uids，mybatis 底层把参数封装到 Map 中，key 为”uids”,value 为 List 集合对象，因此 foreach 标签的 collection 写 uids。

4.3.2. 批量插入

```
 <insert id="addUser2">
        insert into sysuser(uname, upwd, uphone, uemail) values
        <foreach collection="array" separator="," item="suser">
            (#{suser.uname},#{suser.upwd},#{suser.uphone},#{suser.uemail})
        </foreach>
    </insert>
```

```
public int addUser2(Sysuser[] suser);
```

4.3.3. 批量查询

```
public List<Sysuser> selectusersByIds(Integer[] uids);
```

```
<select id="selectusersByIds" resultType="sysuser">
        select * from sysuser where rid in
```

```
        <foreach collection="array" open="(" close=")" separator="," item="userid">
            #{userid}
        </foreach>
    </select>
```

4.4. Choose-when-otherwise

Mybatis 中的 choose 标签相当于 Java 中的 switch。仅供了解。

5. 数据延迟加载

Mybatis 中的延迟加载是在使用 association 或 collection 标签进行多次单表查询时对二次查询进行延迟查询的一种实现。实际项目中不经常使用，仅供了解。

5.1. Mybatis 延迟加载使用

5.1.1. 核心配置

```
<setting name="logImpl" value="STDOUT_LOGGING"/>
    <!-- 启用延迟加载 -->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!-- 积极加载改为消极加载 -->
    <setting name="aggressiveLazyLoading" value="false"/>
    <!-- 调用 toString，equals 不触发延迟对象的加载 -->
    <setting name="lazyLoadTriggerMethods" value=""/>
```

5.1.2. 使用效果

以多对一查询或一对多查询思路 2 进行代码测试，查看执行结果

6. Mybatis 缓存

MyBatis 缓存是指对查询操作的结果进行缓存，再下一次进行查询时，先查询缓存中数据，如果缓存中查到则执行返回；如果缓存中查询不到则查询数据库，并把非空查询结果放在缓存中，供下次查询使用。

mybatis 缓存设计分为两个级别的缓存，一般认识是 session 会话级别缓存与 sessionFactory 全局缓存。

6.1. 一级缓存

一级缓存是默认开启的，仅对同一个会话中的查询生效。Session 关闭后，缓存清空，同时在当前 Session 中使用增删改时,会清空该 Session 中的缓存

执行测试：

```
 @Test
    public void test1_selectUserByKey(){
        SysuserMapper mapper =
session.getMapper(SysuserMapper.class);

        Sysuser sysuser = mapper.selectUserByKey(1);
        System.out.println(sysuser);
        System.out.println("------------分割-------------");
//        session.clearCache();
//        session.close();

        Sysuser sysuser1 = mapper.selectUserByKey(1);
        System.out.println(sysuser1);
    }
```

6.2. 二级缓存

二级缓存需要在 mybatis 配置文件中进行相应的配置才可以使用

当使用查询时,查询结果会存入对应的 namespace 中.

当所属 namespace 使用增删改时,会清空该 namespace 中的缓存

二级缓存可能会存入内存,也可能会存入硬盘

由于二级缓存可能会存入硬盘,所以需要将对应需要缓存的实体类进行序列化(implements Serializable)。

① Xml 核心配置中开启二级缓存

```
<!-- 启用二级缓存 -->
  <setting name="cacheEnabled" value="true"/>
```

② 映射文件配置

■ 在 Mapper XML 文件中设置缓存策略，默认情况下是没有开启缓存的

```
 <!-- 配置缓存策略 -->
  <cache eviction="FIFO"  flushInterval="60000"  size="512"
readOnly="true"></cache>
```

■ 在 Mapper XML 文件配置支持 cache 后，默认当前映射文件中所有查询启用二级缓存，如果需要对个别查询进行调整，可以单独设置 useCache=false 禁用二级缓存。

```
<select id="selectUserByKey" parameterType="int"
resultType="Sysuser">
    select * from sysuser where uid=#{uid}
</select>
```

注意：select 标签有 useCache=false 表示当前查询禁用二级缓存。

■ 测试运行

```
@Test
public void test2_factoryCache(){
    SysuserMapper mapper =
```

```
session.getMapper(SysuserMapper.class);
    Sysuser sysuser = mapper.selectUserByKey(1);
    System.out.println(sysuser);
    System.out.println("------------分割-------------");
    session.close();

    SqlSession session2 = factory.openSession();
    SysuserMapper mapper1 =
session2.getMapper(SysuserMapper.class);
    Sysuser sysuser1 = mapper1.selectUserByKey(1);
    System.out.println(sysuser1);
    session2.close();
}
```

7. 代理模式

代理模式的定义：代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

应用目的：一个对象中的某一个或多个方法不满足需求，需要扩展该对象的时候，在不修改源代码的情况下，去扩展对象中的方法。比如原生 Connection 对象的 close 方法进行连接关闭，如果引入了连接池对象，放入连接池对象中的 Connection 对象的 close 方法不能进行关闭，需要扩展 close 方法，把对象放回连接池。

代理模式分为静态代理与动态代理两种实现。包含两个对象：原对象，代理对象。

7.1. 静态代理

① 准备代码环境

```
public interface ISource {
    public String start(String a,Integer b);
    public String end();
}
```

```
public class SourceImpl implements  ISource {
    @Override
    public String start(String a,Integer b) {
        System.out.println("sourceImpl 对象中的 start 方法:"+a+"--
"+b);
        return a+b;
    }
    @Override
    public String end() {
        System.out.println("sourceImpl 对象中的 end 方法");
        return "ok";
    }
}
```

测试代码

```
ISource source = new SourceImpl();
source.start("aa",123);
```

此时我们认为 start 方法需要扩展，如何能够在不改动源代码的情况下扩展呢？引入代理模式，创建静态代理类：

```
public class SourceProxy implements ISource {
    private ISource source;//原对象

    public SourceProxy(ISource source) {
        this.source = source;
    }
```

```
    @Override
    public String start(String a, Integer b) {
        System.out.println("前置扩展");
        String result= this.source.start(a,b);
        System.out.println("后置扩展");
        return result;
    }

    @Override
    public String end() {
        return this.source.end();
    }
}
```

注意：静态代理类具有两个特征：

特征 1：代理类与原对象同类型；

特征 2：代理类持有原对象的引用。

此时测试代码如下：

```
//原对象
ISource source = new SourceImpl();
//代理对象
ISource proxy= new SourceProxy(source);
proxy.start("aa",123);
```

静态代理缺点：在原对象中的方法非常多时，需要在静态代理类中一个个的调用原对象的方法。代码比较繁琐。

7.2. 动态代理

使用 jdk 的 Proxy 类在程序运行期间在虚拟机中创建代理类,类名规则$Proxy0.$Proxy1...。并编译该代理类，返回实例化对象。

```
public static ISource getProxy(ISource source){
    ClassLoader loader = source.getClass().getClassLoader();
    Class[] interfaces=new Class[]{ISource.class};

    InvocationHandler handler = new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
            String name = method.getName();
            if("start".equals(name)){
                System.out.println("前置扩展");
                Object result = method.invoke(source,args);
                System.out.println("后置扩展");
                return result;
            }else{
                Object result = method.invoke(source, args);//执行某个对象中的方
法
                return result;
            }
        }
    };

    ISource proxy =  (ISource)
Proxy.newProxyInstance(loader,interfaces,handler);
    return proxy;
}
```

此时测试代码如下：

```
//原对象
ISource source = new SourceImpl();
//代理对象
ISource proxy= getProxy(source);
proxy.start("aa",123);
```

注意：可以修改虚拟机参数，把内存中的代理类保存本地，查看代理类源码

```
System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles",
"true");
```

注意点 1：

Proxy.newProxyInstance()方法接受三个参数：

ClassLoader loader:指定当前目标对象使用的类加载器,获取加载器的方法是固定的

Class<?>[] interfaces:指定目标对象实现的接口的类型,使用泛型方式确认类型

InvocationHandler:指定动态处理器，执行目标对象的方法时,会触发事件处理器的 invoke 方法

注意点 2：

Proxy 类创建代理类要求原对象必须有接口才适用。因为$Proxy0 代理类结构大致如下：

```
Public class $Proxy0 extends Proxy implements ISource{
    Private InvocationHandler h;
    Private static Method method0=null;
    Private staitc Method method1=null;
Public $Proxy0(InvocationHandler h){
    This.h=h;
}
     Public Stirng start(String arg1,Integer arg2){
        This.h.invoke(method0,this,new Object[]{arg1,arg2} )
```

```
    }
    Static{
        Method0=Class.forname("ISource").getMethod("start");
       Method1=Class.forname("ISource").getMethod(end)
    }
}
```

该类继承自 Proxy 类，只能适用 implements 关键字派生,因此原对象必须有接口。

8. 总结

Sql 映射 XML 文件进行增删改查操作。

resultMap

动态 sql 语句

if

choose、when、otherwise

trim、where、set

foreach

9. mybatis 总体任务

完成系统管理下用户，角色，权限，字典项四个模块的 dao 层接口及映射文件，并进行测试。

sysuser		InnoDB	16 KB	1	系统用户表
sysrole	2	InnoDB	16 KB	0	系统角色表
syspermission		InnoDB	16 KB	0	系统权限表
sysdicts		InnoDB	16 KB	0	系统字典项
sys_user_role		InnoDB	16 KB	0	用户角色关系表
sys_role_permission		InnoDB	16 KB	0	角色权限关系表

实现内容:(create_time,update_time,create_by,update_by)

1. 每张表基本的添加方法,根据 id 批量删除,插入单个用户,批量插入用户,根据 id 查询,高级条件查询(一定要指定排序方式).添加用户时,维护用户所属部门.

2. 为用户设置角色;

3. 查询用户时,级联查询角色信息;

4. 为角色设置菜单权限;

5. 查询角色下的权限列表;

6. 查询某用户的权限列表;

7. 字典项:项目中固定的列表数据,一般下拉框,复选框,单选框使用.

添加

id	字典名称	字典类型	备注	操作
1	用户性别	user_type	性别列表	列表管理 修改 删除
2	订单状态	order_status	状态列表	列表管理 修改 删除
3	商品颜色			

添加

id	字典名称	字典值	排序	隶属父级	备注	操作
1	男	1	1	user_type	无	修改 删除
2	女	2	2	user_type	无	
3	未知	3	3	user_type	无	
4	未付款	1	1	order_stats		
5						

8.组织机构管理