# MyBatis入门

## 简介

MyBatis 是一款优秀的持久层框架，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信息，将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java对象)映射成数据库中的记录。

优点：

1、简单易学

mybatis本身就很小且简单。没有任何第三方依赖，最简单安装只要两个jar文件+配置几个sql映射文件易于学习，易于使用，通过文档和源代码，可以比较完全的掌握它的设计思路和实现。

2、灵活

mybatis不会对应用程序或者数据库的现有设计强加任何影响。 sql写在xml里，便于统一管理和优化。通过sql基本上可以实现我们不使用数据访问框架可以实现的所有功能，或许更多。

3、解除sql与程序代码的耦合

通过提供DAL层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。sql和代码的分离，提高了可维护性。

4、提供映射标签，支持对象与数据库的orm字段关系映射

5、提供对象关系映射标签，支持对象关系组建维护

6、提供xml标签，支持编写动态sql。

缺点：

1、编写SQL语句时工作量很大，尤其是字段多、关联表多时，更是如此。

2、SQL语句依赖于数据库，导致数据库移植性差，不能更换数据库。

3、框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改。

4、二级缓存机制不佳

总结

mybatis的优点同样是mybatis的缺点，正因为mybatis使用简单，数据的可靠性、完整性的瓶颈便更多依赖于程序员对sql的使用水平上了。sql写在xml里，虽然方便了修改、优化和统一浏览，但可读性很低，调试也非常困难，也非常受限。

mybatis没有hibernate那么强大，但是mybatis最大的优点就是简单小巧易于上手，方便浏览修改sql语句。

## 入门

如果使用 Maven 来构建项目，则需将下面的 dependency 代码置于 pom.xml 文件中：

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>x.x.x</version>

</dependency>

Mybatis配置文件

<?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>

<environments default="development">

<environment id="development">

<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>

<mappers>

<mapper resource="org/mybatis/example/BlogMapper.xml"/>

</mappers>

</configuration>

配置详细介绍<http://www.mybatis.org/mybatis-3/zh/configuration.html>

# MyBatis xml方式的基本用法



SET FOREIGN\_KEY\_CHECKS=0;

-- ----------------------------

-- Table structure for country

-- ----------------------------

DROP TABLE IF EXISTS `country`;

CREATE TABLE `country` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`countryname` varchar(255) DEFAULT NULL,

`countrycode` varchar(255) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=6 DEFAULT CHARSET=utf8;

-- ----------------------------

-- Records of country

-- ----------------------------

INSERT INTO `country` VALUES ('1', '中国', 'CN');

INSERT INTO `country` VALUES ('2', '美国', 'US');

INSERT INTO `country` VALUES ('3', '俄罗斯', 'RU');

INSERT INTO `country` VALUES ('4', '英国', 'GB');

INSERT INTO `country` VALUES ('5', '法国', 'FR');

-- ----------------------------

-- Table structure for sys\_dict

-- ----------------------------

DROP TABLE IF EXISTS `sys\_dict`;

CREATE TABLE `sys\_dict` (

`id` bigint(32) NOT NULL AUTO\_INCREMENT COMMENT '主键',

`code` varchar(64) NOT NULL COMMENT '类别',

`name` varchar(64) NOT NULL COMMENT '字典名',

`value` varchar(64) NOT NULL COMMENT '字典值',

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=7 DEFAULT CHARSET=utf8;

-- ----------------------------

-- Records of sys\_dict

-- ----------------------------

INSERT INTO `sys\_dict` VALUES ('1', '性别', '男', '男');

INSERT INTO `sys\_dict` VALUES ('2', '性别', '女', '女');

INSERT INTO `sys\_dict` VALUES ('3', '季度', '第一季度', '1');

INSERT INTO `sys\_dict` VALUES ('4', '季度', '第二季度', '2');

INSERT INTO `sys\_dict` VALUES ('5', '季度', '第三季度', '3');

INSERT INTO `sys\_dict` VALUES ('6', '季度', '第四季度', '4');

-- ----------------------------

-- Table structure for sys\_privilege

-- ----------------------------

DROP TABLE IF EXISTS `sys\_privilege`;

CREATE TABLE `sys\_privilege` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT COMMENT '权限ID',

`privilege\_name` varchar(50) DEFAULT NULL COMMENT '权限名称',

`privilege\_url` varchar(200) DEFAULT NULL COMMENT '权限URL',

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=6 DEFAULT CHARSET=utf8 COMMENT='权限表';

-- ----------------------------

-- Records of sys\_privilege

-- ----------------------------

INSERT INTO `sys\_privilege` VALUES ('1', '用户管理', '/users');

INSERT INTO `sys\_privilege` VALUES ('2', '角色管理', '/roles');

INSERT INTO `sys\_privilege` VALUES ('3', '系统日志', '/logs');

INSERT INTO `sys\_privilege` VALUES ('4', '人员维护', '/persons');

INSERT INTO `sys\_privilege` VALUES ('5', '单位维护', '/companies');

-- ----------------------------

-- Table structure for sys\_role

-- ----------------------------

DROP TABLE IF EXISTS `sys\_role`;

CREATE TABLE `sys\_role` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT COMMENT '角色ID',

`role\_name` varchar(50) DEFAULT NULL COMMENT '角色名',

`enabled` int(11) DEFAULT NULL COMMENT '有效标志',

`create\_by` bigint(20) DEFAULT NULL COMMENT '创建人',

`create\_time` datetime DEFAULT NULL COMMENT '创建时间',

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8 COMMENT='角色表';

-- ----------------------------

-- Records of sys\_role

-- ----------------------------

INSERT INTO `sys\_role` VALUES ('1', '管理员', '1', '1', '2016-04-01 17:02:14');

INSERT INTO `sys\_role` VALUES ('2', '普通用户', '1', '1', '2016-04-01 17:02:34');

-- ----------------------------

-- Table structure for sys\_role\_privilege

-- ----------------------------

DROP TABLE IF EXISTS `sys\_role\_privilege`;

CREATE TABLE `sys\_role\_privilege` (

`role\_id` bigint(20) DEFAULT NULL COMMENT '角色ID',

`privilege\_id` bigint(20) DEFAULT NULL COMMENT '权限ID'

) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='角色权限关联表';

-- ----------------------------

-- Records of sys\_role\_privilege

-- ----------------------------

INSERT INTO `sys\_role\_privilege` VALUES ('1', '1');

INSERT INTO `sys\_role\_privilege` VALUES ('1', '3');

INSERT INTO `sys\_role\_privilege` VALUES ('1', '2');

INSERT INTO `sys\_role\_privilege` VALUES ('2', '4');

INSERT INTO `sys\_role\_privilege` VALUES ('2', '5');

-- ----------------------------

-- Table structure for sys\_user

-- ----------------------------

DROP TABLE IF EXISTS `sys\_user`;

CREATE TABLE `sys\_user` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT COMMENT '用户ID',

`user\_name` varchar(50) DEFAULT NULL COMMENT '用户名',

`user\_password` varchar(50) DEFAULT NULL COMMENT '密码',

`user\_email` varchar(50) DEFAULT 'test@mybatis.tk' COMMENT '邮箱',

`user\_info` text COMMENT '简介',

`head\_img` blob COMMENT '头像',

`create\_time` datetime DEFAULT NULL COMMENT '创建时间',

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=1035 DEFAULT CHARSET=utf8 COMMENT='用户表';

-- ----------------------------

-- Records of sys\_user

-- ----------------------------

INSERT INTO `sys\_user` VALUES ('1', 'admin', '123456', 'admin@mybatis.tk', '管理员用户', 0x1231231230, '2016-06-07 01:11:12');

INSERT INTO `sys\_user` VALUES ('1001', 'test', '123456', 'test@mybatis.tk', '测试用户', 0x1231231230, '2016-06-07 00:00:00');

-- ----------------------------

-- Table structure for sys\_user\_role

-- ----------------------------

DROP TABLE IF EXISTS `sys\_user\_role`;

CREATE TABLE `sys\_user\_role` (

`user\_id` bigint(20) DEFAULT NULL COMMENT '用户ID',

`role\_id` bigint(20) DEFAULT NULL COMMENT '角色ID'

) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='用户角色关联表';

-- ----------------------------

-- Records of sys\_user\_role

-- ----------------------------

INSERT INTO `sys\_user\_role` VALUES ('1', '1');

INSERT INTO `sys\_user\_role` VALUES ('1', '2');

INSERT INTO `sys\_user\_role` VALUES ('1001', '2');

-- ----------------------------

-- Table structure for user info

-- ----------------------------

DROP TABLE IF EXISTS `user info`;

CREATE TABLE `user info` (

`id` int(11) NOT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 一、select用法

### 1.主键查询

public interface UserMapper {

/\*\*

\* 通过 id 查询用户

\*

\* @param id

\* @return

\*/

SysUser selectById(Long id);

}

//property="实体类属性" column="数据库字段名" jdbcType="数据库字段类型"

<resultMap id="userMap" type="tk.mybatis.simple.model.SysUser">

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

<result property="userName" column="user\_name"/>

<result property="userPassword" column="user\_password"/>

<result property="userEmail" column="user\_email"/>

<result property="userInfo" column="user\_info"/>

<result property="headImg" column="head\_img" jdbcType="BLOB"/>

<result property="createTime" column="create\_time" jdbcType="TIMESTAMP"/>

</resultMap>

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

select \* from sys\_user where id = #{id}

</select>

@Test

public void testSelectById(){

//获取 sqlSession

SqlSession sqlSession = getSqlSession();

try {

//获取 UserMapper 接口

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

//调用 selectById 方法，查询 id = 1 的用户

SysUser user = userMapper.selectById(1l);

//user 不为空

Assert.assertNotNull(user);

//userName = admin

Assert.assertEquals("admin", user.getUserName());

} finally {

//不要忘记关闭 sqlSession

sqlSession.close();

}

}

### 2.查询全部

List<SysUser> selectAll();

<select id="selectAll" resultType="tk.mybatis.simple.model.SysUser">

select id,

user\_name userName,

user\_password userPassword,

user\_email userEmail,

user\_info userInfo,

head\_img headImg,

create\_time createTime

from sys\_user

</select>

@Test

public void testSelectAll(){

SqlSession sqlSession = getSqlSession();

try {

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

//调用 selectAll 方法查询所有用户

List<SysUser> userList = userMapper.selectAll();

//结果不为空

Assert.assertNotNull(userList);

//用户数量大于 0 个

Assert.assertTrue(userList.size() > 0);

} finally {

//不要忘记关闭 sqlSession

sqlSession.close();

}

}

## 二、insert用法

### 1.简单的insert用法

<insert id="insert">

insert into sys\_user(

user\_name, user\_password, user\_email,

user\_info, head\_img, create\_time)

values(

#{userName}, #{userPassword}, #{userEmail},

#{userInfo}, #{headImg, jdbcType=BLOB}, #{createTime, jdbcType=TIMESTAMP})

</insert>

### 2.使用JDBC方式返回主键自增的值

<insert id="insert2" useGeneratedKeys="true" keyProperty="id">

insert into sys\_user(

user\_name, user\_password,

<if test="userEmail != null">

<if test="userEmail != ''">

user\_email,

</if>

</if>

user\_info, head\_img, create\_time)

values(

#{userName}, #{userPassword},

<if test="userEmail != null">

<if test="userEmail != ''">

#{userEmail},

</if>

</if>

#{userInfo}, #{headImg, jdbcType=BLOB}, #{createTime, jdbcType=TIMESTAMP})

</insert>

### 3.使用selectKey方式返回主键的值

<insert id="insert3">

insert into sys\_user(

user\_name, user\_password, user\_email,

user\_info, head\_img, create\_time)

values(

#{userName}, #{userPassword}, #{userEmail},

#{userInfo}, #{headImg, jdbcType=BLOB}, #{createTime, jdbcType=TIMESTAMP})

<selectKey keyColumn="id" resultType="long" keyProperty="id" order="AFTER">

SELECT LAST\_INSERT\_ID()

</selectKey>

</insert>

## 三、update用法

<update id="updateById">

update sys\_user

set user\_name = #{userName},

user\_password = #{userPassword},

user\_email = #{userEmail},

user\_info = #{userInfo},

head\_img = #{headImg, jdbcType=BLOB},

create\_time = #{createTime, jdbcType=TIMESTAMP}

where id = #{id}

</update>

## delete用法

<delete id="deleteById">

delete from sys\_user where id = #{id}

</delete>

## 多个接口参数的用法

List<SysRole> selectRolesByUserIdAndRoleEnabled (Long userId, Integer enabled);

List<SysRole>selectRolesByUserIdAndRoleEnabled(@Param("userId")LonguserId, @Param("enabled")Integer enabled);

<!--xml方式的基本用法 多个接口参数的用法 参数为基本类型-->

<select id="selectRolesByUserIdAndRoleEnabled"resultType="tk.mybatis.simple.model.SysRole">

select

r.id,

r.role\_name roleName,

r.enabled,

r.create\_by createBy,

r.create\_time createTime

from sys\_user u

inner join sys\_user\_role ur on u.id = ur.user\_id

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

where u.id = #{userId} and r.enabled = #{enabled}

</select>

<!--xml方式的基本用法 多个接口参数的用法 参数为java对象-->

<select id="selectRolesByUserAndRole" resultType="tk.mybatis.simple.model.SysRole">

select

r.id,

r.role\_name roleName,

r.enabled,

r.create\_by createBy,

r.create\_time createTime

from sys\_user u

inner join sys\_user\_role ur on u.id = ur.user\_id

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

where u.id = #{user.id} and r.enabled = #{role.enabled}

</select>

# MyBatis 动态SQL

## if用法

### 在WHERE条件中使用if

<select id="selectByUser" resultType="tk.mybatis.simple.model.SysUser">

<bind name="print" value="@tk.mybatis.util.StringUtil@print(\_parameter)"/>

select id,

user\_name userName,

user\_password userPassword,

user\_email userEmail,

user\_info userInfo,

head\_img headImg,

create\_time createTime

from sys\_user

where 1 = 1

<if test="@tk.mybatis.util.StringUtil@isNotEmpty(userName)">

and user\_name like concat('%', #{userName}, '%')

</if>

<if test="userEmail != '' and userEmail != null">

and user\_email = #{userEmail}

</if>

</select>

If标签有一个必填的属性**test**，test的属性值是一个符合OGNL要求的判断表达式，表达式的结果可以是true或false，除此之外所有的非0值都为true，只有0为false。为了方便理解，在表达式中，建议只用true或false作为结果。

判断条件property != null或property == null：适用于任何类型的字段，用于判断属性值是否为空。

判断条件property != ‘ ’ 或property == ‘ ’：仅适用于String类型的字段，用于判断是否为空字符串。

**and和or**：当有多个判断条件时，使用and或or进行连接，嵌套的判断可以使用小括号分组，and相当于Java中的与（&&），or相当于Java中的或（||）。

where 1 = 1 如果不加，当条件都不满足时，会报错。这种写法也不美观。可以使用where标签代替这种写法。

条件中的and（或or）需要手动添加，因为有where 1=1这个条件所以不需要判断第一个动态条件是否需要加上and（或or）。

### 在UPDATE更新列中使用if

<update id="updateByIdSelective">

<bind name="print" value="@tk.mybatis.util.StringUtil@print(\_parameter)"/>

update sys\_user

set

<if test="userName != null and userName != ''">

user\_name = #{userName},

</if>

<if test="userPassword != null and userPassword != ''">

user\_password = #{userPassword},

</if>

<if test="userEmail != null and userEmail != ''">

user\_email = #{userEmail},

</if>

<if test="userInfo != null and userInfo != ''">

user\_info = #{userInfo},

</if>

<if test="headImg != null">

head\_img = #{headImg, jdbcType=BLOB},

</if>

<if test="createTime != null">

create\_time = #{createTime, jdbcType=TIMESTAMP},

</if>

where id = #{id}

</update>

注意：第一点是If元素里面SQL后面的逗号；第二点就是where关键字前面的id=#{id}这个条件。

全部的查询条件都是null或者空。

如果有id = #{id}这个条件，最终的SQL如下。

update sys\_user set id = #{id} where id = #{id}

如果没有这个条件，最终的SQL如下。

update sys\_user set where id = #{id}

不符合语句规范

查询条件只有一个不是null也不是空（假设userName）。

如果有id = #{id}这个条件，最终的SQL如下。

update sys\_user set user\_name = #{userName} ,id = #{id} where id = #{id}

如果没有这个条件，最终的SQL如下。

update sys\_user set user\_name = #{userName} , where id = #{id}

不符合语句规范

### 在INSERT动态插入列中使用if

<insert id="insert2" useGeneratedKeys="true" keyProperty="id">

insert into sys\_user(

user\_name, user\_password,

<if test="userEmail != null">

<if test="userEmail != ''">

user\_email,

</if>

</if>

user\_info, head\_img, create\_time)

values(

#{userName}, #{userPassword},

<if test="userEmail != null">

<if test="userEmail != ''">

#{userEmail},

</if>

</if>

#{userInfo}, #{headImg, jdbcType=BLOB}, #{createTime, jdbcType=TIMESTAMP})

</insert>

在INSERT中使用时要注意，若在列的部分增加if条件，则value的部分也要增加相同的if条件，必须保证上下文可以互相对应，完全匹配。

## choose用法（if...else...）

<select id="selectByIdOrUserName" resultType="tk.mybatis.simple.model.SysUser">

select id,

user\_name userName,

user\_password userPassword,

user\_email userEmail,

user\_info userInfo,

head\_img headImg,

create\_time createTime

from sys\_user

where 1 = 1

<choose>

<when test="id != null">

and id = #{id}

</when>

<when test="userName != null and userName != ''">

and user\_name = #{userName}

</when>

<otherwise>

limit 0

</otherwise>

</choose>

</select>

choose元素中包含when和otherwise两个标签，一个choose中至少有一个when，有0个或者一个otherwise。

上述查询中如果没有otherwise这个条件，所有的用户都会被查询出来。

## 三、where、set、trim用法

### 1.where用法

<select id="selectByUser" resultType="tk.mybatis.simple.model.SysUser">

<bind name="print" value="@tk.mybatis.util.StringUtil@print(\_parameter)"/>

select id,

user\_name userName,

user\_password userPassword,

user\_email userEmail,

user\_info userInfo,

head\_img headImg,

create\_time createTime

from sys\_user

<where>

<if test="@tk.mybatis.util.StringUtil@isNotEmpty(userName)">

and user\_name like concat('%', #{userName}, '%')

</if>

<if test="userEmail != '' and userEmail != null">

and user\_email = #{userEmail}

</if>

</where>

</select>

where标签的作用：如果该标签包含的元素中有返回值，就插入一个where;如果where后面的字符串是以AND和OR开头的,就将它们剔除。

### set用法

<update id="updateByIdSelective">

<bind name="print" value="@tk.mybatis.util.StringUtil@print(\_parameter)"/>

update sys\_user

<set>

<if test="userName != null and userName != ''">

user\_name = #{userName},

</if>

<if test="userPassword != null and userPassword != ''">

user\_password = #{userPassword},

</if>

<if test="userEmail != null and userEmail != ''">

user\_email = #{userEmail},

</if>

<if test="userInfo != null and userInfo != ''">

user\_info = #{userInfo},

</if>

<if test="headImg != null">

head\_img = #{headImg, jdbcType=BLOB},

</if>

<if test="createTime != null">

create\_time = #{createTime, jdbcType=TIMESTAMP},

</if>

id = #{id},

</set>

where id = #{id}

</update>

set标签的作用：如果该标签包含的元素中有返回值，就插入一个set;如果set后面的字符串是以逗号结尾的,就将这个逗号剔除。

### trim用法

where和ser标签的功能都可以用trim标签来实现，并且在底层就是通过TrimSqlNode实现的。

where标签对应trim的实现如下。

<trim prefix=”WHERE” prefixOverrides=”AND |OR”>

. . .

</trim>

set标签对应trim的实现如下。

<trim prefix=”SET” prefixOverrides=”,”>

. . .

</trim>

trim标签有如下属性。

prefix：当trim元素内包含内容时，会给内容增加prefix指定的前缀。

prefixOverrides：当trim元素内包含内容时，会把内容中匹配的前缀字符串去掉。

suffix：当trim元素内包含内容时，会给内容增加suffix指定的前缀。

suffixOverrides：当trim元素内包含内容时，会把内容中匹配的后缀字符串去掉。

## 四、foreach用法

### 1.foreach实现in集合

<select id="selectByIdList" resultType="tk.mybatis.simple.model.SysUser">

select id,

user\_name userName,

user\_password userPassword,

user\_email userEmail,

user\_info userInfo,

head\_img headImg,

create\_time createTime

from sys\_user

where id in

<foreach collection="list" open="(" close=")" separator="," item="id" index="i">

#{id}

</foreach>

</select>

foreach包含以下属性。

* collection：必填,值为要迭代循环的属性名。这个属性值的情况有很多。
* item：变量名，值为从迭代对象中取出的每一个值。
* index：索引的属性名，在集合数组情况下值为当前索引值，当迭代对象时Map类型时，这个值为Map的key（键值）。
* open：整个循环内容开头的字符串。
* close：整个循环内容结尾的字符串。
* separator：每次循环的分隔符。

collection常见情况。

1. 只有一个数组参数或集合参数

当参数类型为集合的时候，默认会转换为Map类型，并添加一个key为collection的值（MyBatis3.3.0版本中增加），如果参数类型是List集合，那么就继续添加一个key为list的值（MyBatis3.2.8及低版本中只有这一个key），这样，当collection=”list”时，就能得到这个集合，并对它进行循环操作。

当参数类型为数组的时候，也会转换成Map类型，默认的key为array。

1. 有多个参数

要使用@Param注解给每个参数制定一个名字

1. 参数是Map类型

使用Map和使用@Param注解方式类似，将collection指定为对应Map中的key即可。

### foreach实现批量插入

<insert id="insertList" useGeneratedKeys="true" keyProperty="id">

insert into sys\_user(

user\_name, user\_password,user\_email,

user\_info, head\_img, create\_time)

values

<foreach collection="list" item="user" separator=",">

(

#{user.userName}, #{user.userPassword},#{user.userEmail},

#{user.userInfo}, #{user.headImg, jdbcType=BLOB},

#{user.createTime, jdbcType=TIMESTAMP})

</foreach>

</insert>

通过item指定了循环变量名后，在引用值的时候使用的是“属性.属性”的方式，如user.userName。

从MyBatis3.3.1版本开始，MyBatis开始支持批量新增回写主键值的功能，要求数据库主键值为自增类型，目前为止只有MySql数据库支持该功能。

### foreach实现动态UPDATE

<update id="updateByMap">

update sys\_user

set

<foreach collection="\_parameter" item="val" index="key" separator=",">

${key} = #{val}

</foreach>

where id = #{id}

</update>

当参数是Map类型的时候，使用foreach实现动态UPDATE。

Map的key作为更新的列名，对应的值作为该列的更新值，通过foreach标签将更新的字段进行拼接。

# MyBatis 高级查询

## 一、高级结果映射

### 1.一对一映射

1.1 使用自动映射处理一对一关系

<select id="selectUserAndRoleById" resultType="tk.mybatis.simple.model.SysUser">

select

u.id,

u.user\_name userName,

u.user\_password userPassword,

u.user\_email userEmail,

u.user\_info userInfo,

u.head\_img headImg,

u.create\_time createTime,

r.id "role.id",

r.role\_name "role.roleName",

r.enabled "role.enabled",

r.create\_by "role.createBy",

r.create\_time "role.createTime"

from sys\_user u

inner join sys\_user\_role ur on u.id = ur.user\_id

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

where u.id = #{id}

</select>

使用自动映射就是通过别名让MyBatis自动将值匹配到对应的字段上，简单的别名映射如user\_name对应userName。除此之外MyBatis还支持复杂的属性映射，可以多层嵌套，例如将role.role\_name映射到role.roleName上。MyBatis会先查找role属性，如果存在role属性就创建role对象，然后在role对象中继续查找roleName，将role\_name的值绑定到role对象的roleName属性上。

优点：减少数据库查询次数，减轻数据库的压力

缺点：要写很复杂的SQL，当潜逃结果更复杂时，不容易一次写正确；增加应用服务器的压力

适用场景：当一定会使用到嵌套结果，并且整个复杂的SQL执行速度很快是，建议使用。

1.2 使用resultMap配置一对一映射

<resultMap id="userRoleMap" type="tk.mybatis.simple.model.SysUser">

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

<result property="userName" column="user\_name"/>

<result property="userPassword" column="user\_password"/>

<result property="userEmail" column="user\_name"/>

<result property="userInfo" column="user\_name"/>

<result property="headImg" column="user\_name"/>

<result property="createTime" column="create\_time"

jdbcType="TIMESTAMP"/>

<!--role相关属性-->

<result property="role.id" column="role\_id"/>

<result property="role.roleName" column="role\_role\_name"/>

<result property="role.enabled" column="role\_enabled"/>

<result property="role.createBy" column="role\_create\_by"/>

<result property="role.createTime" column="role\_create\_time"

jdbcType="TIMESTAMP"/>

</resultMap>

简化resultMap 使用继承

<resultMap id="userRoleMap" extends="userMap" type="tk.mybatis.simple.model.SysUser">

<result property="role.id" column="role\_id"/>

<result property="role.roleName" column="role\_role\_name"/>

<result property="role.enabled" column="role\_enabled"/>

<result property="role.createBy" column="role\_create\_by"/>

<result property="role.createTime" column="role\_create\_time"

jdbcType="TIMESTAMP"/>

</resultMap>

<select id="selectUserAndRoleById2" resultMap="userRoleMap">

select

u.id,

u.user\_name,

u.user\_password,

u.user\_email,

u.user\_info,

u.head\_img,

u.create\_time,

r.id role\_id,

r.role\_name role\_role\_name,

r.enabled role\_enabled,

r.create\_by role\_create\_by,

r.create\_time role\_create\_time

from sys\_user u

inner join sys\_user\_role ur on u.id = ur.user\_id

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

where u.id = #{id}

</select>

1.3 使用resultMap的association标签配置一对一映射

<resultMap id="userRoleMap" extends="userMap" type="tk.mybatis.simple.model.SysUser">

<association property="role" columnPrefix="role\_"

javaType="tk.mybatis.simple.model.SysRole">

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

<result property="roleName" column="role\_name"/>

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

<result property="createBy" column="create\_by"/>

<result property="createTime" column="create\_time"

jdbcType="TIMESTAMP"/>

</association>

</resultMap>

1.4 使用association标签的嵌套查询

<resultMap id="userRoleMapSelect" extends="userMap" type="tk.mybatis.simple.model.SysUser">

<association property="role"

fetchType="lazy"

select="tk.mybatis.simple.mapper.RoleMapper.selectRoleById"

column="{id=role\_id}"/>

</resultMap>

<select id="selectUserAndRoleByIdSelect" resultMap="userRoleMapSelect">

select

u.id,

u.user\_name,

u.user\_password,

u.user\_email,

u.user\_info,

u.head\_img,

u.create\_time,

ur.role\_id

from sys\_user u

inner join sys\_user\_role ur on u.id = ur.user\_id

where u.id = #{id}

</select>

### 一对多映射

2.1 collection集合的嵌套结果映射

<resultMap id="userRoleListMap" extends="userMap" type="tk.mybatis.simple.model.SysUser">

<collection property="roleList" columnPrefix="role\_"

resultMap="tk.mybatis.simple.mapper.RoleMapper.rolePrivilegeListMap"/>

</resultMap>

<select id="selectAllUserAndRoles" resultMap="userRoleListMap">

select

u.id,

u.user\_name,

u.user\_password,

u.user\_email,

u.user\_info,

u.head\_img,

u.create\_time,

r.id role\_id,

r.role\_name role\_role\_name,

r.enabled role\_enabled,

r.create\_by role\_create\_by,

r.create\_time role\_create\_time,

p.id role\_privilege\_id,

p.privilege\_name role\_privilege\_privilege\_name,

p.privilege\_url role\_privilege\_privilege\_url

from sys\_user u

inner join sys\_user\_role ur on u.id = ur.user\_id

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

inner join sys\_role\_privilege rp on rp.role\_id = r.id

inner join sys\_privilege p on p.id = rp.privilege\_id

</select>

2.2 collection集合的嵌套查询

<resultMap id="userRoleListMapSelect" extends="userMap" type="tk.mybatis.simple.model.SysUser">

<collection property="roleList"

fetchType="lazy"

select="tk.mybatis.simple.mapper.RoleMapper.selectRoleByUserId"

column="{userId=id}"/>

</resultMap>

<select id="selectAllUserAndRolesSelect" resultMap="userRoleListMapSelect">

select

u.id,

u.user\_name,

u.user\_password,

u.user\_email,

u.user\_info,

u.head\_img,

u.create\_time

from sys\_user u

where u.id = #{id}

</select>

实例

 <resultMap id="NavBarResultMap" type="com.ceying.system.dto.NavBarDto">  
        <id column="MENU\_CODE" jdbcType="VARCHAR" property="menuCode" />  
        <result column="parentName" jdbcType="VARCHAR" property="parentName" />  
        <result column="attribute1" jdbcType="VARCHAR" property="attribute1" />  
        <result column="attribute2" jdbcType="VARCHAR" property="attribute2" />  
        <result column="userId"  jdbcType="VARCHAR" property="userId" />  
        <result column="MENU\_CODE" jdbcType="VARCHAR" property="id" />  
        <result column="MENU\_NAME"  jdbcType="VARCHAR" property="name" />  
        <collection column ="{menuCode=MENU\_CODE,userId=userId}" ofType="com.ceying.chx.base.dto.NavBarDto" property="children" select="queryNavBar"></collection>  
    </resultMap>

-----------------------------------------------------------------

 <select id="queryNavBar" parameterType="java.util.Map"  resultMap="NavBarResultMap">  
        select distinct a.\*,b.user\_id as userId FROM(  
        select t.\*,t1.menu\_name as parent\_name,t.menu\_url as attribute1,t.menu\_icon as attribute2 FROM tsys\_menu t left join tsys\_menu t1 on t.parent\_code=t1.menu\_code  
        where t.parent\_code= #{menuCode,jdbcType=VARCHAR})a,tsys\_user\_rights b WHERE a.menu\_code=b.trans\_code and b.user\_id= #{userId,jdbcType=VARCHAR}  
    </select>

### 鉴别器映射

有时一个单独的数据库查询会返回很多不同数据类型（希望有些关联）的结果集。discriminator鉴别器标签就是用来处理这种情况的。鉴别器非常容易理解，因为它很像Java语言中的switch语句。

discriminator标签常用的两个属性如下。

* column：该属性用于设置要进行鉴别比较值的列。
* javaType：该属性用于指定列的类型，保证使用相同的Java类型来比较值。

discriminator标签可以有1个或多个case标签，case标签包含以下三个属性。

* value：该值为discriminator指定column用来匹配的值。
* resultMap：当column的值和value的值匹配时，可以配置使用resultMap指定的映射，resultMap优先级高于resultType。
* resultType：当column的值和value的值匹配时，用于配置使用resultType指定的映射。

case标签下面可以包含的标签和resultMap一样，用法也一样。

<resultMap id="rolePrivilegeListMapChoose" type="tk.mybatis.simple.model.SysRole">

<discriminator column="enabled" javaType="int">

<case value="1" resultMap="rolePrivilegeListMapSelect"/>

<case value="0" resultMap="roleMap"/>

</discriminator>

</resultMap>

<resultMap id="rolePrivilegeListMapSelect" extends="roleMap" type="tk.mybatis.simple.model.SysRole">

<collection property="privilegeList" fetchType="lazy"

select="tk.mybatis.simple.mapper.PrivilegeMapper.selectPrivilegeByRoleId"

column="{roleId=id}"/>

</resultMap>

<resultMap id="roleMap" type="tk.mybatis.simple.model.SysRole">

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

<result property="roleName" column="role\_name"/>

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

<association property="createInfo" javaType="tk.mybatis.simple.model.CreateInfo">

<result property="createBy" column="create\_by"/>

<result property="createTime" column="create\_time" jdbcType="TIMESTAMP"/>

</association>

</resultMap>

<select id="selectRoleByUserIdChoose" resultMap="rolePrivilegeListMapChoose">

select

r.id,

r.role\_name,

r.enabled,

r.create\_by,

r.create\_time

from sys\_role r

inner join sys\_user\_role ur on ur.role\_id = r.id

where ur.user\_id = #{userId}

</select>

## 存储过程

# 第一个存储过程

# 根据用户 id 查询用户其他信息

# 方法看着很奇葩，但是展示了多个输出参数

DROP PROCEDURE IF EXISTS `select\_user\_by\_id`;

DELIMITER ;;

CREATE PROCEDURE `select\_user\_by\_id`(

IN userId BIGINT,

OUT userName VARCHAR(50),

OUT userPassword VARCHAR(50),

OUT userEmail VARCHAR(50),

OUT userInfo TEXT,

OUT headImg BLOB,

OUT createTime DATETIME)

BEGIN

# 根据用户 id 查询其他数据

select user\_name,user\_password,user\_email,user\_info,head\_img,create\_time

INTO userName,userPassword,userEmail,userInfo,headImg,createTime

from sys\_user

WHERE id = userId;

END

;;

DELIMITER ;

# 第二个存储过程

# 简单的根据用户名和分页参数进行查询，返回总数和分页数据

DROP PROCEDURE IF EXISTS `select\_user\_page`;

DELIMITER ;;

CREATE PROCEDURE `select\_user\_page`(

IN userName VARCHAR(50),

IN \_offset BIGINT,

IN \_limit BIGINT,

OUT total BIGINT)

BEGIN

# 查询数据总数

select count(\*) INTO total

from sys\_user

where user\_name like concat('%', userName, '%');

# 分页查询数据

select \*

from sys\_user

where user\_name like concat('%', userName, '%')

limit \_offset, \_limit;

END

;;

DELIMITER ;

# 第三个存储过程

# 保存用户信息和角色关联信息

DROP PROCEDURE IF EXISTS `insert\_user\_and\_roles`;

DELIMITER ;;

CREATE PROCEDURE `insert\_user\_and\_roles`(

OUT userId BIGINT,

IN userName VARCHAR(50),

IN userPassword VARCHAR(50),

IN userEmail VARCHAR(50),

IN userInfo TEXT,

IN headImg BLOB,

OUT createTime DATETIME,

IN roleIds VARCHAR(200)

)

BEGIN

# 设置当前时间

SET createTime = NOW();

# 插入数据

INSERT INTO sys\_user(user\_name, user\_password, user\_email, user\_info, head\_img, create\_time)

VALUES (userName, userPassword, userEmail, userInfo, headImg, createTime);

# 获取自增主键

SELECT LAST\_INSERT\_ID() INTO userId;

# 保存用户和角色关系数据

SET roleIds = CONCAT(',',roleIds,',');

INSERT INTO sys\_user\_role(user\_id, role\_id)

select userId, id from sys\_role

where INSTR(roleIds, CONCAT(',',id,',')) > 0;

END

;;

DELIMITER ;

# 第四个存储过程

# 删除用户信息和角色关联信息

DROP PROCEDURE IF EXISTS `delete\_user\_by\_id`;

DELIMITER ;;

CREATE PROCEDURE `delete\_user\_by\_id`(IN userId BIGINT)

BEGIN

DELETE FROM sys\_user\_role where user\_id = userId;

DELETE FROM sys\_user where id = userId;

END

;;

DELIMITER ;

### 1.第一个存储过程

<select id="selectUserById" statementType="CALLABLE" useCache="false">

{call select\_user\_by\_id(

#{id, mode=IN},

#{userName, mode=OUT, jdbcType=VARCHAR},

#{userPassword, mode=OUT, jdbcType=VARCHAR},

#{userEmail, mode=OUT, jdbcType=VARCHAR},

#{userInfo, mode=OUT, jdbcType=VARCHAR},

#{headImg, mode=OUT, jdbcType=BLOB, javaType=\_byte[]},

#{createTime, mode=OUT, jdbcType=TIMESTAMP}

)}

</select>

### 第二个存储过程

<select id="selectUserPage" statementType="CALLABLE" useCache="false" resultMap="userMap">

{call select\_user\_page(

#{userName, mode=IN},

#{offset, mode=IN},

#{limit, mode=IN},

#{total, mode=OUT, jdbcType=BIGINT}

)}

</select>

### 第三个和第四个存储过程

<insert id="insertUserAndRoles" statementType="CALLABLE">

{call insert\_user\_and\_roles(

#{user.id, mode=OUT, jdbcType=BIGINT},

#{user.userName, mode=IN},

#{user.userPassword, mode=IN},

#{user.userEmail, mode=IN},

#{user.userInfo, mode=IN},

#{user.headImg, mode=IN, jdbcType=BLOB},

#{user.createTime, mode=OUT, jdbcType=TIMESTAMP},

#{roleIds, mode=IN}

)}

</insert>

<delete id="deleteUserById" statementType="CALLABLE">

{call delete\_user\_by\_id(#{id, mode=IN})}

</delete>

### 三、使用枚举或其他对象

### 1.使用MyBatis提供的枚举处理器

第一种枚举处理器org.apache.ibatis.type.EnumTypeHandler

public enum Enabled {

disabled,//禁用

enabled; //启用

}

该枚举处理器只对枚举的字面值进行处理。（“disabled”,”enabled”）

第二种枚举处理器org.apache.ibatis.type.EnumOrdinalTypeHandler

public enum Enabled {

disabled,//禁用

enabled; //启用

}

<typeHandlers>

<typeHandler

javaType="tk.mybatis.simple.type.Enabled"

handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"/>

</typeHandlers>

该枚举处理器使用枚举的索引进行处理。（“0”,”1”）

### 使用自定义的类型处理器

不使用枚举的字面值，也不使用枚举的索引。

public enum Enabled {

enabled(1), //启用

disabled(0);//禁用

}

<typeHandlers>

<typeHandler

javaType="tk.mybatis.simple.type.Enabled"

handler="tk.mybatis.simple.type.EnabledTypeHandler"/>

</typeHandlers>

public class EnabledTypeHandler implements TypeHandler<Enabled> {

private final Map<Integer, Enabled> enabledMap = new HashMap<Integer, Enabled>();

public EnabledTypeHandler() {

for(Enabled enabled : Enabled.values()){

enabledMap.put(enabled.getValue(), enabled);

}

}

public EnabledTypeHandler(Class<?> type) {

this();

}

@Override

public void setParameter(PreparedStatement ps, int i, Enabled parameter, JdbcType jdbcType) throws SQLException {

ps.setInt(i, parameter.getValue());

}

@Override

public Enabled getResult(ResultSet rs, String columnName) throws SQLException {

Integer value = rs.getInt(columnName);

return enabledMap.get(value);

}

@Override

public Enabled getResult(ResultSet rs, int columnIndex) throws SQLException {

Integer value = rs.getInt(columnIndex);

return enabledMap.get(value);

}

@Override

public Enabled getResult(CallableStatement cs, int columnIndex) throws SQLException {

Integer value = cs.getInt(columnIndex);

return enabledMap.get(value);

}

}

### 对Java8日期（JSR-310）的支持

MyBatis 3.4.0版本开始支持Java 8日期（JSR-310）

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-typehandlers-jsr310<artifactId>

<version>1.0.2</version>

</dependency>

如果使用 3.4.0以前的版本还要在mybatis配置文件中添加如下配置

<typeHandlers>

<typeHandler handler="org.apache.ibatis.type.InstantTypeHandler"/>

<typeHandler handler="org.apache.ibatis.type.LocalDateTimeTypeHandler"/>

<typeHandler handler="org.apache.ibatis.type.LocalDateTypeHandler"/>

<typeHandler handler="org.apache.ibatis.type.LocalTimeTypeHandler"/>

<typeHandler handler="org.apache.ibatis.type.OffsetDateTimeTypeHandler"/>

<typeHandler handler="org.apache.ibatis.type.OffsetTimeTypeHandler"/>

<typeHandler handler="org.apache.ibatis.type.ZonedDateTimeTypeHandler"/>

<typeHandler handler="org.apache.ibatis.type.YearTypeHandler"/>

<typeHandler handler="org.apache.ibatis.type.MonthTypeHandler"/>

</typeHandlers>

# MyBatis 缓存配置

## 一、MyBatis的一级缓存

session范围内缓存，session关闭，一级缓存同时删除。

在SqlSession未关闭之前，如果对于同样条件进行重复查询，此时采用的是（一级缓存）local session cache。

## 二、MyBatis的二级缓存

Sessionfactory范围内缓存

同一个sessionfactory生成的session共享二级缓存

1、全局开关：默认是true，如果它配成false，其余各个Mapper XML文件配成支持cache也没用。

<settings>

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

</settings>

2、各个Mapper XML文件，默认是不采用cache。在配置文件加一行就可以支持二级cache：

<cache />

3、Mapper XML文件配置支持cache后，文件中所有的Mapper statement就支持了。此时要个别对待某条，需要（如不使用缓存）：

<select id="inetAton" parameterType="string" resultType="integer" useCache=“false”>

select inet\_aton(#{name})

</select>

注意

1、如果readOnly为false，此时要结果集对象是可序列化的。

<cache readOnly="false"/>

要求所有缓存的对象必须实现Serializable(java.io.Serializable)接口，如下所示：

public class SysRole implements Serializable{

Private static final long serialVersionUID=6320941908222932112L;

//其他属性和getter,setter方法

}

2、在SqlSession未关闭之前，如果对于同样条件进行重复查询，此时采用的是（一级缓存）local session cache，而不是上面说的这些cache（二级缓存）。

3、MyBatis缓存查询到的结果集对象，而非结果集数据，是将映射的PO对象集合缓存起来。

说意义不大是在于：

a、面对一定规模的数据量，内置的cache方式就派不上用场了;

b、对查询结果集做缓存并不是MyBatis框架擅长的，它专心做的应该是sql mapper。

## 三、集成EhCache缓存

### 1.添加项目依赖

在pom.xml中添加如下依赖。

<dependency>

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

<artifactId>mybatis-ehcache</artifactId>

<version>1.0.3</version>

</dependency>

### 2.配置EhCache

在src/main/resources目录下新增ehcache.xml文件。

<?xml version="1.0" encoding="UTF-8"?>

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"

updateCheck="false" monitoring="autodetect"

dynamicConfig="true">

<diskStore path="D:/cache" />

<defaultCache

maxElementsInMemory="3000"

eternal="false"

copyOnRead="true"

copyOnWrite="true"

timeToIdleSeconds="3600"

timeToLiveSeconds="3600"

overflowToDisk="true"

diskPersistent="true"/>

</ehcache>

### 3.修改RoleMapper.xml中的缓存配置

修改RoleMapper.xml中的配置如下。

<mapper namespace="tk.mybatis.simple.mapper.RoleMapper">

<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

<!--其他配置-->

</mapper>

只通过设置type属性就可以使用EhCache缓存了，这时cache的其他属性都不会起到任何作用，针对缓存的配置都在ehcache.xml中进行。

## 集成Redis缓存

### 1.添加项目依赖

在pom.xml中添加如下依赖。

<dependency>

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

<artifactId>mybatis-redis</artifactId>

<version>1.0.0-beta2</version>

</dependency>

mybatis-redis目前只有beta版本

### 2.配置Redis

安装Redis服务

Redis服务启动后，在src/main/resources目录下新增redis.properties文件。

host=localhost

Port=6379

connectionTimeout=5000

soTimeout=5000

password=

database=0

clientName=

### 3.修改RoleMapper.xml中的缓存配置

修改RoleMapper.xml中的配置如下。

<mapper namespace="tk.mybatis.simple.mapper.RoleMapper">

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

<!--其他配置-->

</mapper>

RedisCache在保存数据和获取缓存数据时，使用了Java的序列化和反序列化，因此还需要保证被缓存的对象必须实现Serializable接口。

# MyBatis 事务使用

## MyBatis单独使用时

public void doxxxxx(){

TransactionFactory transactionFactory = new JdbcTransactionFactory();

userMapper userDao=getSession().getMapper(UserMapper.class);

Transaction newTransaction=transactionFactory.newTransaction(getSession().getConnection());

try {

userDao.insert(xxx);

userDao.update(xxx);

} catch (Exception e) {

newTransaction.rollback();

e.printStackTrace();

} finally {

newTransaction.close();

}

}

## 和Spring集成后，使用Spring的事务管理

### 声明式事务配置

<!-- 配置事务管理 -->

<bean name="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 事务相关控制配置：例如配置事务的传播机制 -->

<tx:advice id="iccardTxAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="delete\*" propagation="REQUIRED" read-only="false" rollback-for="java.lang.Exception" no-rollback-for="java.lang.RuntimeException"/>

<tx:method name="insert\*" propagation="REQUIRED" read-only="false" rollback-for="java.lang.RuntimeException" />

<tx:method name="add\*" propagation="REQUIRED" read-only="false" rollback-for="java.lang.RuntimeException" />

<tx:method name="create\*" propagation="REQUIRED" read-only="false" rollback-for="java.lang.RuntimeException" />

<tx:method name="update\*" propagation="REQUIRED" read-only="false" rollback-for="java.lang.Exception" />

<tx:method name="find\*" propagation="SUPPORTS" />

<tx:method name="get\*" propagation="SUPPORTS" />

<tx:method name="select\*" propagation="SUPPORTS" />

<tx:method name="query\*" propagation="SUPPORTS" />

</tx:attributes>

</tx:advice>

<!-- 把事务控制在service层 -->

<aop:config>

<aop:pointcut id="iccardTerm" expression="execution(public \* com.shfft.iccardterm.service.\*.\*(..))" />

<aop:advisor pointcut-ref="iccardTerm" advice-ref="iccardTxAdvice" />

</aop:config>

### 注解式事务配置

<!-- 定义事务管理器 -->

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

<!--使用注释事务 -->

<tx:annotation-driven transaction-manager="transactionManager" />

事物配置中有哪些属性可以配置

（1）、事务的传播性：@Transactional(propagation=Propagation.REQUIRED)

如果有事务, 那么加入事务, 没有的话新建一个(默认情况下)

（2）、事务的超时性：@Transactional(timeout=30) //默认是30秒

注意这里说的是事务的超时性而不是Connection的超时性，这两个是有区别的

（3）、事务的隔离级别：@Transactional(isolation = Isolation.READ\_UNCOMMITTED)

读取未提交数据(会出现脏读, 不可重复读) 基本不使用

（4）、回滚：

指定单一异常类：@Transactional(rollbackFor=RuntimeException.class)

指定多个异常类：@Transactional(rollbackFor={RuntimeException.class, Exception.class})

该属性用于设置需要进行回滚的异常类数组，当方法中抛出指定异常数组中的异常时，则进行事务回滚。

（5）、只读：@Transactional(readOnly=true)

该属性用于设置当前事务是否为只读事务，设置为true表示只读，false则表示可读写，默认值为false。

解释说明

事务的传播级别定义的是事务的控制范围，主要是父子事务之间的相互影响关系；事务的隔离级别定义的是事务读写的控制范围，主要是两个事务之间的相互影响关系。

传播级别：

1）、REQUIRED

如果当前方法已经在事务中，那么就以父事务执行，不需要新建事务；如果当前方法不在事务中，那么就为当前方法新建事务。回滚情况：父子方法中任何地方出现问题，都会全部回滚。

2）、SURPPORTED

如果当前方法已经在事务中，那么就以当前事务执行；如果当前方法不再事务中，那么就以非事务方式运行。如果运行在事务中，那么只要出现异常都会回滚。

3）、NOT\_SURPPORTED

如果当前方法已经在事务中，那么就挂起当前事务，以非事务方式运行，方法执行完毕后，恢复事务；如果当前方法不再事务中，那么就以非事务方式执行。

4）、MANDATORY

强制以事务方式执行，如果当前方法不在事务中，那么会抛出异常。

5）、NEVER

与MANDATORY相反，强制以非事务方式执行，如果当前方法在事务中，那么会抛出异常。

6）、REQUIRED\_NEW

与REQUIRED不同的是，无论该方法当前是不是在事务中，都会为自己新建一个事务。如果当前已经在事务中，那么会挂起父事务，为自己新建一个事务。父事务不影响它子事务的提交和回滚。

7）、NESTED

嵌套事务。理解Nested的关键是savepoint。他与PROPAGATION\_REQUIRES\_NEW的区别是，PROPAGATION\_REQUIRES\_NEW另起一个事务，将会与他的父事务相互独立，而Nested的事务和他的父事务是相依的，他的提交是要等和他的父事务一块提交的。也就是说，如果父事务最后回滚，他也要回滚的。而Nested事务的好处是他有一个savepoint。

# MyBatis 代码生成器（MBG）

## XML配置详解

xml配置文件的文件头

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE generatorConfiguration

PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"

"http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">

<generatorConfiguration>

<!--具体配置内容-->

</generatorConfiguration>

上边两部分是MBG必备的基本信息

generatorConfiguration标签下有3个子级标签，分别是properties、classPathEntry和context。

在配置这3个标签的时候，必须严格按照列举这些标签的顺序进行配置。

**properties标签**用于指定一个需要在配置中解析使用的外部属性文件，引入属性文件后，可以在配置中使用${property}这种形式的引用。properties标签包含resource和url两个属性，只能使用其中一个属性来指定，同时出现则会报错。

resource：指定classpath下的属性文件，类似com/myproject/generatorConfig.properties这样的属性值。

url：指定文件系统上的特定位置，例如file:///C:/myfolder/generatorConfig.properties。

**ClassPathEntry**这个标签可以配置多个，也可以不配置。

classPathEntry标签最常见的用法是通过属性location指定驱动的路径，代码如下。

<classPathEntry location=”E:\mysql\mysql-connector-java-5.1.29.jar”/>

**context**该标签至少配置一个，可以配置多个，该标签用于指定生成一组对象的环境。例如指定要连接的数据库，要生成对象的类型和要处理的数据库中的表。运行MBG的时候可以指定要运行的context。

context标签只有一个必选属性**id**，用来唯一确定该标签，该id属性可以在运行MBG时使用。此外还有几个可选属性。

**defaultModelType**：该属性定义了MBG如何生成实体类，有以下几个可选值。

* conditional：默认值，和下面的hierarchical类似，如果一个表的主键只有一个字段，那么不会为该字段生成单独的实体类，而是会将该字段合并到基本实体类中。
* flat：该模型只为每张表生成一个实体类。这个实体类包含表中的是所有字段。这种模型最简单，推荐使用。
* hierarchical：如果表有主键，那么该模型会产生一个单独的主键实体类，如果表还有BLOB字段，则会为表生成一个包含所有BLOB字段的实体类，然后为所有其他的字段另外生成一个单独的实体类。MBG会在所有生成的实体类之间维护一个继承关系。

**targetRuntime**：此属性用于指定生成的代码的运行时环境，有以下可选值。

* MyBatis3：默认值。
* MyBatis3Simple：这种情况下不会生成与Example相关的方法。

context的子标签如下（有严格的配置顺序）。

* property（0个或多个）
* plugin（0个或多个）
* commentGenerator（0个或1个）
* jdbcConnection（1个）
* javaTypeResolver（0个或1个）
* javaModelGenerator（1个）
* sqlMapGenerator（0个或1个）
* javaClientGenerator（0个或1个）
* table（1个或多个）

### property标签

在MySQL中可以使用返单引号“`”作为分隔符；MGB会自动为关键字添加分隔符，关键字列表在org.mybatis.generator.internal.db.SqlReservedWords类中。

* autoDelimitKeywords
* beginningDelimiter
* endingDelimiter
* javaFileEncoding
* javaFormatter
* xmlFormatter

<!--MySql分隔符-->

<property name="beginningDelimiter" value="`"/>

<property name="endingDelimiter" value="`"/>

<property name="javaFileEncoding" value="UTF-8"/>

### plugin标签

plugin标签可以配置0个或者多个，个数不受限制。plugin标签用来定义一个插件，用于扩展或修改通过MBG生成的代码。该插件将按在配置中配置的顺序执行。

### commentGenerator标签

该标签用来配置如何生成注释信息，最多可以配置一个。

该标签有一个可选属性type，可以指定用户的实现类，该类需要实现org.mybatis.generator.api.CommentGenerator接口，而且必有一个默认空的构造方法。type属性接收默认的特殊值DEFAULT，使用默认的实现类org.mybatis.generator.internal.DdfaultCommentGenerator。

默认的实现类中提供了三个可选属性，需要通过property属性进行配置。

* suppressAllComments：组织生成注释，默认为false。
* suppressDate：阻止生成的注释包含时间戳，默认为false。
* addRemarkComments：注释是否添加数据库表的备注信息，默认为false。

一般情况下，由于MBG生成的注释信息没有任何价值，而且有时间戳的情况下每次生成的注释都不一样，使用版本控制的时候每次都会提交，因而一般情况下都会屏蔽注释信息，可以如下配置。

<commentGenerator>

<property name="suppressDate" value="true"/>

<property name="addRemarkComments" value="true"/>

</commentGenerator>

### jdbcConnection标签

jdbcConnection标签用于指定MBG要连接的数据库信息，该标签必选，并且只能有一个。

该标签有两个必选属性。

* driverClass：访问数据库的JDBC驱动程序的完全限定类名。
* connectionURL：访问数据库的JDBC连接URL。

该标签还有两个可选属性。

* userId：访问数据库的用户ID。
* password：访问数据库的密码。

### javaTypeResolver标签

该标签的配置用来指定JDBC类型和Java类型如何转换，最多可以配置一个。

该标签提供了一个可选的属性type。另外，和commentGenerator类似，该标签提供了默认的实现DEFAULT，一般情况下使用默认即可，需要特殊处理的情况可以通过其他标签配置来解决，不建议修改该属性。

### javaModelGenerator标签

该标签用来控制生成的实体类，根据context标签中配置的defaultModelType属性值的不同，一个表可能会生成多个不同的实体类。一个表对应多个类时使用并不方便，所以前面推荐使用flat，保证一个表对应一个实体类。该标签必须配置一个，并且最多配置一个。

该标签只有两个必选属性。

* targetPackage：生成实体类存放的包名。一般就是放在该包下，实际还会受到其他配置的影响。
* targetProject：指定目标项目路径，可以使用相对路径或绝对路径。

该标签还支持以下几个property子标签属性。

constructorBased：该属性只对MyBatis3有效，如果为true就会使用构造方法入参，如果为false就会使用setter方式。默认为false。

enableSubPackages：如果为true，MBG会根据catalog和schema来生成子包。如果为false就会直接使用targetPackage属性。默认为false。

immutable：用来配置实体类属性是否可变。如果设置为true，那么constructorBased不管设置成什么，都会使用构造方法入参，并且不会生成setter方法。如果为false，实体类属性就可以改变。默认为false。

rootClass：设置说有实体类的基类。如果设置，则需要使用类的全限定名称。并且，如果MBG能够加载rootClass，那么MBG不会覆盖和父类中完全匹配的属性。

trimStrings：判断是否对数据库查询结果进行trim操作，默认值为false，如果为true就会生成如下代码。

public void setUsername(String username){

this.username = username==null ? null : username.trim();

}

javaModelGenerator标签的配置如下。

<javaModelGenerator targetPackage="test.model" targetProject="src\main\java">

<property name="trimStrings" value="true" />

</javaModelGenerator>

### sqlMapGenerator标签

该标签用于配置SQL映射生成器（Mapper.xml文件）的属性，该标签可选，最多配置一个。如果targetRuntime设置为MyBatis3，则只有当javaClientGenerator配置需要XML时，该标签才必须配置一个。如果没有配置javaClientGenerator，则使用以下规则。

如果指定了一个sqlMapGenerator，那么MBG将只生成XML的SQL映射文件和实体类。

如果没有指定sqlMapGenerator，那么MBG将只生成实体类。

该标签只有两个必选属性。

* targetPackage：生成SQL映射文件（xml文件）存放的包名。一般就是放在该报下，实际还会受到其他配置的影响。
* targetProject：指定目标项目路径，可以使用相对路径或绝对路径。

### javaClientGenerator标签

该标签用于配置Java客户端生成器（Mapper接口）的属性，该标签可选，最多配置一个。如果不配置该标签，就不会生成Mapper接口。

该标签有以下3个必选属性。

* type：用于选择客户端代码（Mapper接口）生成器，用户可以自定义实现，需要继承org.mybatis.generator.codegen.AbstractJavaClientGenerator类，必须有一个默认空的构造方法。type属性的一般选项。
* XMLMAPPER：推荐使用，将接口和XML完全分离，容易维护，接口中不出现SQL语句，只在XML中配置SLQ，修改SQL时不需要重新编译。
* ANNOTATEDMAPPER：不推荐使用，纯注解方式。
* MIXEDMAPPER：不推荐使用，这种情况下注解和XML混合使用会很乱，不利于维护。
* targetPackage：生成Mapper接口存放的包名。
* targetProject：指定目标项目路径，可以使用相对路径或绝对路径。

### table标签

table标签是最重要的一个标签，该标签用于配置需要通过内省数据库的表，只有在table配置过的表，才能经过上述其他配置生成最终的代码，该标签最少配置一个，可以配置多个。

table标签有一个必填属性tableName，该属性指定要生成的表名，可以使用SQL通配符匹配多个表。例如要生成全部的表，可以如下配置。

<table tableName="%">

table的子标签如下。

* generatedKey标签，该标签用来指定自动生成主键的属性（identity字段或者sequences序列）。如果指定这个标签，MBG将在生成insert的SQL映射文件中插入一个selectKey标签。这个标签非常重要，而且只能配置一个。
* columnRenamingRule标签，该标签最多可以配置一个，使用该标签可以在生成列之前对列进行重命名。可用来删除前缀等。
* columnOverride标签,该标签用于将某些默认计算的属性值更改为指定的值，标签可选，可以配置多个。（可用于修改默认的转换类型等）
* ignoreColumn标签，该标签可以用来屏蔽不需要生成的列，该标签可选，可以配置多个。该标签有一个必选属性column，表示要忽略的列名。

## 一个配置参考示例

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE generatorConfiguration

PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"

"http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">

<generatorConfiguration>

<!-- <classPathEntry location="F:\.m2\repository\mysql\mysql-connector-java\5.1.38\mysql-connector-java-5.1.38.jar"/> -->

<context id="MySqlContext" targetRuntime="MyBatis3" defaultModelType="flat">

<!--MySql分隔符-->

<property name="beginningDelimiter" value="`"/>

<property name="endingDelimiter" value="`"/>

<property name="javaFileEncoding" value="UTF-8"/>

<commentGenerator>

<property name="suppressDate" value="true"/>

<property name="addRemarkComments" value="true"/>

</commentGenerator>

<jdbcConnection driverClass="com.mysql.jdbc.Driver"

connectionURL="jdbc:mysql://localhost:3306/mybatis"

userId="root"

password="">

</jdbcConnection>

<javaModelGenerator targetPackage="test.model" targetProject="src\main\java">

<property name="trimStrings" value="true" />

<property name="rootClass" value="tk.mybatis.simple.model.BaseEntity" />

</javaModelGenerator>

<sqlMapGenerator targetPackage="test.xml" targetProject="src\main\resources"/>

<javaClientGenerator type="XMLMAPPER" targetPackage="test.dao" targetProject="src\main\java"/>

<table tableName="%">

<generatedKey column="id" sqlStatement="MySql"/>

</table>

</context>

</generatorConfiguration>

## 运行MyBatis Generator

### 使用Java编写代码运行

在写代码前，需要先把MBG的jar包添加到项目中。

第一种方式是，从地址https://github.com/mybatis/generator/releases下载jar包。

第二种方式是，使用Maven方式直接引入依赖，在pom.xml中添加如下依赖。

<!-- MyBatis 生成器 -->

<dependency>

<groupId>org.mybatis.generator</groupId>

<artifactId>mybatis-generator-core</artifactId>

<version>1.3.3</version>

</dependency>

创建Generator.java类。

/\*\*

\* 读取 MBG 配置生成代码

\*/

public class Generator {

public static void main(String[] args) throws Exception {

//MBG 执行过程中的警告信息

List<String> warnings = new ArrayList<String>();

//当生成的代码重复时，覆盖原代码

boolean overwrite = true;

//读取我们的 MBG 配置文件

InputStream is = Generator.class.getResourceAsStream("/generator/generatorConfig.xml");

ConfigurationParser cp = new ConfigurationParser(warnings);

Configuration config = cp.parseConfiguration(is);

is.close();

DefaultShellCallback callback = new DefaultShellCallback(overwrite);

//创建 MBG

MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config, callback, warnings);

//执行生成代码

myBatisGenerator.generate(null);

//输出警告信息

for(String warning : warnings){

System.out.println(warning);

}

}

}

### 从命令提示符运行

配置不方便，不再详细介绍

### 使用Maven Plugin运行

使用Maven插件方式和第一种Java编码方式类似，都是和项目绑定在一起的，当需要引用其他类时，需要给Maven插件添加依赖，否则会找不到类。pom.xml添加如下插件配置。

<plugin>

<groupId>org.mybatis.generator</groupId>

<artifactId>mybatis-generator-maven-plugin</artifactId>

<version>1.3.3</version>

<configuration>

<configurationFile>${basedir}/src/main/resources/generator/generatorConfig.xml</configurationFile>

<overwrite>true</overwrite>

<verbose>true</verbose>

</configuration>

<dependencies>

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.38</version>

</dependency>

<dependency>

<groupId>tk.mybatis</groupId>

<artifactId>simple</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

</plugin>

## Example介绍

CountryMapper测试类下

@Test

public void testExample() {

// 获取 sqlSession

SqlSession sqlSession = getSqlSession();

try {

// 获取 CountryMapper 接口

CountryMapper countryMapper = sqlSession.getMapper(CountryMapper.class);

//创建 Example 对象

CountryExample example = new CountryExample();

//设置排序规则

example.setOrderByClause("id desc, countryname asc");

//设置是否 distinct 去重

example.setDistinct(true);

//创建条件，只能有一个 createCriteria

CountryExample.Criteria criteria = example.createCriteria();

//id >= 1

criteria.andIdGreaterThanOrEqualTo(1);

//id < 4

criteria.andIdLessThan(4);

//countrycode like '%U%'

//最容易出错的地方，注意 like 必须自己写上通配符的位置，不可能默认两边加 %，like 可以是任何情况

criteria.andCountrycodeLike("%U%");

//or 的情况，可以有多个 or

CountryExample.Criteria or = example.or();

//countryname = 中国

or.andCountrynameEqualTo("中国");

//执行查询

List<Country> countryList = countryMapper.selectByExample(example);

printCountryList(countryList);

} finally {

// 不要忘记关闭 sqlSession

sqlSession.close();

}

}

# MyBatis 插件开发

MyBatis允许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接口和方法包括以下几个。

* Executor（update、query、flushStatements、commit、rollback、getTransaction、close、isClosed）
* ParameterHandler（getParameterObject、setParameters）
* ResultSetHandler（handleResultSets、handleCursorResultSets、handleOutputParameters）
* StatementHandler（prepare、parameterize、batch、update、query）

这4个接口

## 拦截器接口介绍

MyBaits插件可以用来实现拦截器接口Interceptor（org.apache.ibatis.plugin.Interceptor），在实现类中对拦截对象和方法进行处理。Interceptor接口代码如下。

public interface Interceptor{

Object intercept(Invocation invocation) throws Throwable;

Object plugin(Object target);

void setProperties(Properties properties);

}

setProperties方法，这个方法用来传递插件的参数，可以通过参数来改变插件的行为。

<plugins>

<plugin interceptor=”tk.mybatis.simple.plugin.XXXInterceptor”>

<property name=”prop1” value=”value1”/>

<property name=”prop2” value=”value2”/>

</plugin>

</plugins>

在配置拦截器时，plugin的interceptor属性为拦截器实现类的全限定名称，如果需要参数，可以在plugin标签内通过property标签进行配置，配置后的参数在拦截器初始化时会通过setProperties方法传递给拦截器。

plugin方法的参数target就是拦截器要拦截的对象，该方法会在创建被拦截接口实现类时被调用。

intercept方法是MyBatis运行时要执行的拦截方法。通过该方法的参数invocation可以得到很多有用的信息，该参数的常用方法如下。

@Override

Public Object intercept（Invocation invocation） throws Throwable{

Object target = invocation.getTarget();

Method method = invocation.getMethod();

Object[] args = invocation.getArgs();

Object result = invocation.proceed();

return result;

}

使用getTarget()方法可以获取当前被拦截的对象，使用getMethod()可以获取当前被拦截的方法，使用getArgs方法可以返回被拦截方法中的参数。通过调用invocation.proceed();

可以执行被拦截对象的真正的方法，proceed()方法实际上执行了method.invoke(target,args)方法，上面的代码中没有做任何特殊处理，直接返回了执行的结果。

当配置多个拦截器时，MyBatis会遍历所有拦截器，按顺序执行拦截器的plugin方法，被拦截的对象就会被层层代理。在执行拦截对象的方法时，会一层层地调用拦截器，拦截器通过invocation.proceed()调用下一层的方法，直到真正的方法被执行。方法执行的结果会从最里面开始向外一层层返回，所以如果存在按顺序配置的A、B、C三个签名相同的拦截器，MyBatis会按照C>B>A>target.proceed()>A>B>C的顺序执行。如果A、B、C签名不同，就会按照MyBatis拦截对象的逻辑执行。

## 二、拦截器签名介绍

除了需要实现拦截器接口外，还需要给实现类配置以下的拦截器注解。

@Intercepts（org.apache.ibatis.plugin.Intercepts）和签名注解@Signature（org.apache.ibatis.plugin.Signature），这两个注解用来配置拦截器要拦截的接口的方法。

@Intercepts注解中的属性是一个@Signature（签名）数组，可以在同一个拦截器中同时拦截不同的接口和方法。

以拦截ResultSetHandler接口的handeResultSets方法为例，配置签名如下。

@Intercepts({

@Signature(

type = ResultSetHandler.class,

method = “handleResultSets”,

args = {Statement.class})

})

public class ResultSetInterceptor implements Interceptor

@Signature注解包含以下三个属性。

* type：设置拦截的接口，可选值时前面提到的4个接口。
* method：设置拦截接口中的方法名，可选值是前面4个接口对应的方法，需要和接口匹配。
* args：设置拦截方法的参数类型数组，通过方法名和参数类型可以确定唯一一个方法。

### Executor接口

* int update(MappedStatement var1, Object var2) throws SQLException;

该方法会在所有的INSERT、UPDATE、DELETE执行时被调用，因此如果星耀拦截这3类操作，可以拦截该方法。接口对应的签名如下。

@Signature(

type = Executor.class,

method = “update”,

args = {MappedStatement.class,Object.class})

* <E> List<E> query(MappedStatement var1, Object var2, RowBounds var3, ResultHandler var4) throws SQLException;

该方法会在所有SELECT查询方法执行时被调用。通过这个接口可以获取很多有用的信息，因此这是最常被拦截的一个方法。使用该方法需要注意的是，虽然接口中还有一个参数更多的同名接口，但由于MyBatis的设计原因，这个参数多的接口不能被拦截。接口方法对应的签名如下。

@Signature(

type = Executor.class,

method = “query”,

args = {MappedStatement.class,Object.class,RowBounds.class,ResultHandler.class})

* <E> Cursor<E> queryCursor(MappedStatement var1, Object var2, RowBounds var3) throws SQLException;

该方法只有在查询的返回值类型为Cursor时被调用。接口方法对应的签名如下。

@Signature(

type = Executor.class,

method = “queryCursor”,

args = {MappedStatement.class,Object.class,RowBounds.class})

* List<BatchResult> flushStatements() throws SQLException;

该方法只在通过SqlSession方法调用flushStatements方法或执行的接口方法中带有@Flush注解时才被调用，接口方法对应的签名如下。

@Signature(

type = Executor.class,

method = “flushStatements”,

args = {})

* void commit(boolean var1) throws SQLException;

该方法只在通过SqlSession方法调用commit方法时才被调用，接口方法对应的签名如下。

@Signature(

type = Executor.class,

method = “commit”,

args = {boolean.class})

* void rollback(boolean var1) throws SQLException;

该方法只在通过SqlSession方法调用rollback方法时才被调用，接口方法对应的签名如下。

@Signature(

type = Executor.class,

method = “rollback”,

args = {boolean.class})

* Transaction getTransaction();

该方法只在通过SqlSession方法获取数据库连接时才被调用，接口方法对应的签名如下。

@Signature(

type = Executor.class,

method = “getTransaction”,

args = {})

* void close(boolean var1);

该方法只在延迟加载获取新的Exexutor后才会被执行，接口方法对应的签名入下。

@Signature(

type = Executor.class,

method = “close”,

args = {boolean.class})

* boolean isClosed();

该方法只在延迟加载执行查询方法前被执行，接口方法对应的签名入下。

@Signature(

type = Executor.class,

method = “isClosed”,

args = {})

### ParamterHandler接口

* Object getParameterObject();

该方法只在执行存储过程处理出参的时候被调用。解耦方法对应的签名如下。

@Signature(

type = ParamterHandler.class,

method = “getParameterObject”,

args = {})

* void setParameters(PreparedStatement var1) throws SQLException;

该方法在所有数据库方法设置SQL参数时被调用。接口方法对应的签名如下。

@Signature(

type = ParamterHandler.class,

method = “setParameter”,

args = {PreparedStatement .class})

### ResultSetHandler接口

* <E> List<E> handleResultSets(Statement var1) throws SQLException;

该方法会在除存储过程及返回值类型为Cursor<T> （org.apache.ibatis.cursor.Cursor<T>）以外的查询方法中被调用。接口方法对应的签名如下。

@Signature(

type = ResultSetHandler.class,

method = “handleResultSets”,

args = {Statement .class})

* <E> Cursor<E> handleCursorResultSets(Statement var1) throws SQLException;

该方法是3.4.0版本中新增加的，只会在返回值类型为Cursor<E>的查询犯法中被调用，接口方法对应的签名如下。

@Signature(

type = ResultSetHandler.class,

method = “handleCursorResultSets”,

args = {Statement .class})

* void handleOutputParameters(CallableStatement var1) throws SQLException;

该方法只在使用存储过程处理出参时被调用，接口方法对应的签名如下。

@Signature(

type = ResultSetHandler.class,

method = “handleOutputParameters”,

args = {CallableStatement .class})

### StatementHandler接口

* Statement prepare(Connection var1, Integer var2) throws SQLException;

该方法会在数据库执行前被调用，优先于当前接口中的其他方法而被执行。接口方法对应的签名如下。

@Signature(

type = StatementHandler.class,

method = “prepare”,

args = {Connection .class,Integer.class})

* void parameterize(Statement var1) throws SQLException;

该方法在prepare方法之后执行，用于处理参数信息，接口方法对应的签名如下。

@Signature(

type = StatementHandler.class,

method = “parameterize”,

args = {Statement .class})

* void batch(Statement var1) throws SQLException;

在全局设置配置defaultExecutorType=”BATCH”时，执行数据操作才会调用该方法，接口方法对应的签名方法如下。

@Signature(

type = StatementHandler.class,

method = “batch”,

args = {Statement .class})

* <E> List<E> query(Statement var1, ResultHandler var2) throws SQLException;

执行SLLECT方法时调用，接口方法对应的签名如下。

@Signature(

type = StatementHandler.class,

method = “query”,

args = {Statement .class,ResultHandler.class})

* <E> Cursor<E> queryCursor(Statement var1) throws SQLException;

该方法时3.4.0版本中新增加的，只会在返回值类型为Cursor<T>的查询中被调用，接口方法对应的签名如下。

@Signature(

type = StatementHandler.class,

method = “queryCursor”,

args = {Statement .class})

# MyBatis类型转换器

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时， 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。

|  |  |  |
| --- | --- | --- |
| 类型处理器 | Java 类型 | JDBC 类型 |
| BooleanTypeHandler | java.lang.Boolean, boolean | 数据库兼容的 BOOLEAN |
| ByteTypeHandler | java.lang.Byte, byte | 数据库兼容的 NUMERIC 或 BYTE |
| ShortTypeHandler | java.lang.Short, short | 数据库兼容的 NUMERIC 或 SHORT INTEGER |
| IntegerTypeHandler | java.lang.Integer, int | 数据库兼容的 NUMERIC 或 INTEGER |
| LongTypeHandler | java.lang.Long, long | 数据库兼容的 NUMERIC 或 LONG INTEGER |
| FloatTypeHandler | java.lang.Float, float | 数据库兼容的 NUMERIC 或 FLOAT |
| DoubleTypeHandler | java.lang.Double, double | 数据库兼容的 NUMERIC 或 DOUBLE |
| BigDecimalTypeHandler | java.math.BigDecimal | 数据库兼容的 NUMERIC 或 DECIMAL |
| StringTypeHandler | java.lang.String | CHAR, VARCHAR |
| ClobReaderTypeHandler | java.io.Reader | - |
| ClobTypeHandler | java.lang.String | CLOB, LONGVARCHAR |
| NStringTypeHandler | java.lang.String | NVARCHAR, NCHAR |
| NClobTypeHandler | java.lang.String | NCLOB |
| BlobInputStreamTypeHandler | java.io.InputStream | - |
| ByteArrayTypeHandler | byte[] | 数据库兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB, LONGVARBINARY |
| DateTypeHandler | java.util.Date | TIMESTAMP |
| DateOnlyTypeHandler | java.util.Date | DATE |
| TimeOnlyTypeHandler | java.util.Date | TIME |
| SqlTimestampTypeHandler | java.sql.Timestamp | TIMESTAMP |
| SqlDateTypeHandler | java.sql.Date | DATE |
| SqlTimeTypeHandler | java.sql.Time | TIME |
| ObjectTypeHandler | Any | OTHER 或未指定类型 |
| EnumTypeHandler | Enumeration Type | VARCHAR-任何兼容的字符串类型，存储枚举的名称（而不是索引） |
| EnumOrdinalTypeHandler | Enumeration Type | 任何兼容的 NUMERIC 或 DOUBLE 类型，存储枚举的索引（而不是名称）。 |
| InstantTypeHandler | java.time.Instant | TIMESTAMP |
| LocalDateTimeTypeHandler | java.time.LocalDateTime | TIMESTAMP |
| LocalDateTypeHandler | java.time.LocalDate | DATE |
| LocalTimeTypeHandler | java.time.LocalTime | TIME |
| OffsetDateTimeTypeHandler | java.time.OffsetDateTime | TIMESTAMP |
| OffsetTimeTypeHandler | java.time.OffsetTime | TIME |
| ZonedDateTimeTypeHandler | java.time.ZonedDateTime | TIMESTAMP |
| YearTypeHandler | java.time.Year | INTEGER |
| MonthTypeHandler | java.time.Month | INTEGER |
| YearMonthTypeHandler | java.time.YearMonth | VARCHAR or LONGVARCHAR |
| JapaneseDateTypeHandler | java.time.chrono.JapaneseDate | DATE |

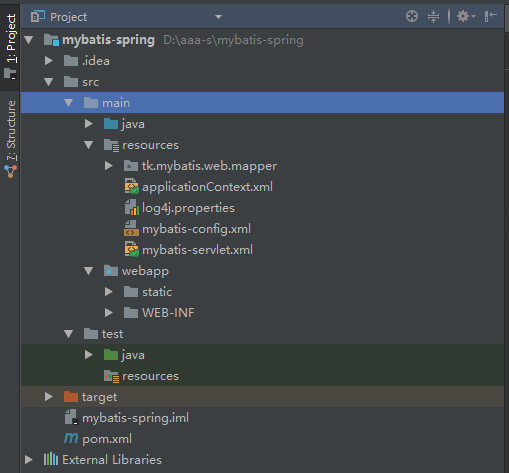
# Spring集成MyBatis

## 创建基本的Maven Web项目

### 在pom.xml中添加packaging配置

<packaging>war</packaging>

### 增加Web基本目录和配置



### 在pom.xml中添加Web相关依赖

<!--web-->

<!--支持 Servlet-->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<!--支持 JSP-->

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

<!--支持 JSTL-->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

### 添加一个简单页面 index.jsp

<%@ page import="java.util.Date" %>

<%@ page language="java" contentType="text/html; charset=UTF8" pageEncoding="UTF8"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF8">

<title>Index</title>

</head>

<body>

<p>

Hello Spring MVC!

</p>

<p>

服务器时间：<fmt:formatDate value="${now}" pattern="yyyy-MM-dd HH:mm:ss"/>

</p>

</body>

</html>

## 集成Spring和Spring MVC

### 添加Spring项目清单用于管理Spring依赖

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-framework-bom</artifactId>

<version>4.3.4.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

### 添加Spring依赖

<!--Spring 上下文，核心依赖-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<!--Spring JDBC-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

</dependency>

<!--Spring 事务-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

</dependency>

<!--Spring 面向切面编程-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

</dependency>

<!--spring-aop 依赖-->

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.8.2</version>

</dependency>

### 添加Spring MVC依赖

<!--Spring Web 核心-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

</dependency>

<!--Spring MVC-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

</dependency>

<!--spring mvc-json依赖-->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.8.4</version>

</dependency>

### 添加Spring XML配置文件

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<context:component-scan base-package="tk.mybatis.\*.service.impl"/>

<bean id="dataSource" class="org.apache.ibatis.datasource.pooled.PooledDataSource">

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

<property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>

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

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

</bean>

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="configLocation" value="classpath:mybatis-config.xml"/>

<property name="dataSource" ref="dataSource"/>

<property name="mapperLocations">

<array>

<value>classpath:tk/mybatis/\*\*/mapper/\*.xml</value>

</array>

</property>

<property name="typeAliasesPackage" value="tk.mybatis.web.model"/>

</bean>

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="addToConfig" value="true"/>

<property name="basePackage" value="tk.mybatis.\*\*.mapper"/>

</bean>

<aop:aspectj-autoproxy/>

<aop:config>

<aop:pointcut id="appService" expression="execution(\* tk.mybatis.\*.service..\*Service\*.\*(..))"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="appService"/>

</aop:config>

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<tx:method name="select\*" read-only="true"/>

<tx:method name="find\*" read-only="true"/>

<tx:method name="get\*" read-only="true"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

</beans>

### 添加Spring MVC的配置文件

<?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"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<mvc:annotation-driven/>

<mvc:resources mapping="/static/\*\*" location="static/"/>

<context:component-scan base-package="tk.mybatis.\*.controller"/>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>

</bean>

</beans>

### 配置web.xml

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

version="3.0">

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<servlet>

<servlet-name>mybatis</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:mybatis-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>mybatis</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<filter>

<filter-name>SpringEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

<init-param>

<param-name>forceEncoding</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>SpringEncodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

</web-app>

### 增加一个简单的Controller示例

@Controller

public class IndexController {

@RequestMapping(value = {"", "/index"})

public ModelAndView dicts() {

ModelAndView mv = new ModelAndView("index");

mv.addObject("now", new Date());

return mv;

}

}

## 集成MyBatis

### 在pom.xml中添加mybatis-spring依赖

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>1.3.0</version>

</dependency>

### 配置SqlSessionFactoryBean

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="configLocation" value="classpath:mybatis-config.xml"/>

<property name="dataSource" ref="dataSource"/>

<property name="mapperLocations">

<array>

<value>classpath:tk/mybatis/\*\*/mapper/\*.xml</value>

</array>

</property>

<property name="typeAliasesPackage" value="tk.mybatis.web.model"/>

</bean>

### 配置MapperScannerConfigurer

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="addToConfig" value="true"/>

<property name="basePackage" value="tk.mybatis.\*\*.mapper"/>

</bean>

# Spring Boot集成MyBatis

## 基本的Spring Boot项目

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://maven.apache.org/POM/4.0.0"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>tk.mybatis</groupId>

<artifactId>mybatis-spring-boot</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.6.RELEASE</version>

</parent>

<properties>

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

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

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

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

## 集成MyBatis

<dependency>

<groupId>tk.mybatis</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.2.0</version>

</dependency>

接口添加@Mapper注解

添加mapper . xml映射文件

# mybatis 配置（application.properties）

mybatis.type-aliases-package=tk.mybatis.springboot.model

mybatis.mapper-locations=classpath:mapper/\*.xml

## MyBatis Starter配置介绍

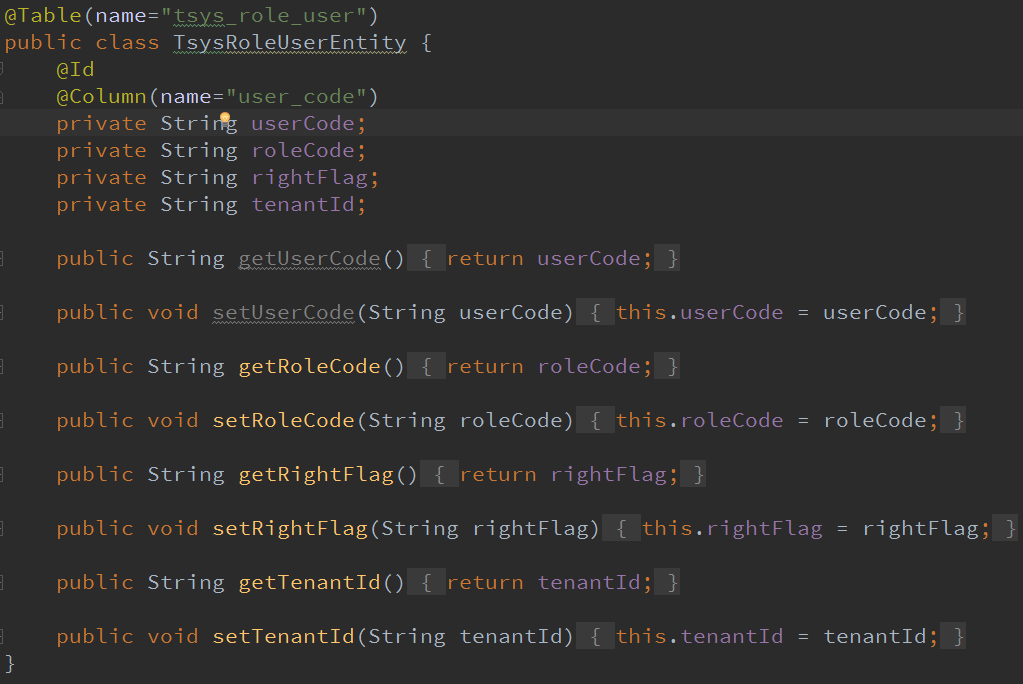
# mybatis 配置

mybatis.type-aliases-package=tk.mybatis.springboot.model

mybatis.mapper-locations=classpath:mapper/\*.xml

# mapper-3.5.0 通用接口使用教程

## 创建和数据库表对应的实体类



支持驼峰命名（userCode 对应数据库中的 user\_code）

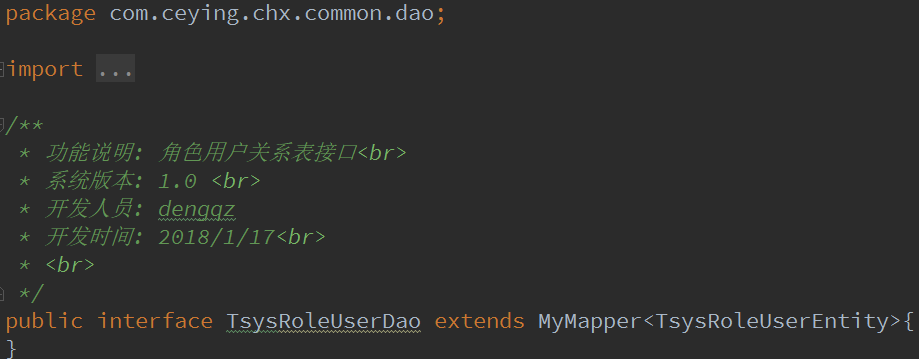
@Table 指定该实体类对应数据库的表名

@Id 指定该属性对应数据库的主键字段

@Colum 声明该属性对应数据库的字段名（例如： userCode对应数据库的主键user\_code）

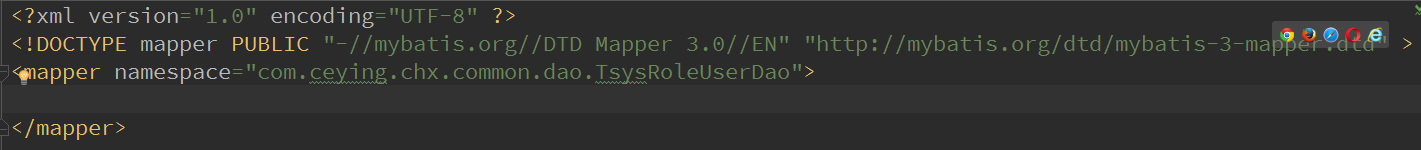
@Transient 声明该属性为数据库表中不存在字段

## 创建对应的Dao接口



继承通用接口MyMapper<T>, T为上一步创建的实体类

## 创建对应的mapper.xml文件



指定namespace为上一步创建的Dao

## 使用继承的接口方法

@Resource

private TsysRoleUserDao tsysRoleUserDao;

List<TsysRoleUserEntity> list = tsysRoleUserDao.selectAll();

# mapper-3.5.0 通用接口说明

注：红色字体为目前未继承使用的接口，开发中不可使用，目前共22个可用接口

## Select(tk.mybatis.mapper.common.base.select)

接口：ExistsWithPrimaryKeyMapper<T>  
方法：boolean existsWithPrimaryKey(Object key)  
说明：根据主键字段查询总数，方法参数必须包含完整的主键属性，查询条件使用等号

接口：SelectAllMapper<T>  
方法：List<T> selectAll()  
说明：查询全部结果

接口：SelectByPrimaryKeyMapper<T>  
方法：T selectByPrimaryKey(Object key)  
说明：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

接口：SelectCountMapper<T>  
方法：int selectCount(T record)  
说明：根据实体中的属性查询总数，查询条件使用等号

接口：SelectMapper<T>  
方法：List<T> select(T record)  
说明：根据实体中的属性值进行查询，查询条件使用等号

接口：SelectOneMapper<T>  
方法：T selectOne(T record)  
说明：根据实体中的属性进行查询，只能有一个返回值，有多个结果是抛出异常，查询条件使用等号

## Insert(tk.mybatis.mapper.common.base.insert)

接口：InsertMapper<T>  
方法：int insert(T record)  
说明：保存一个实体，null的属性也会保存，不会使用数据库默认值

接口：InsertSelectiveMapper<T>  
方法：int insertSelective(T record)  
说明：保存一个实体，null的属性不会保存，会使用数据库默认值

## Update(tk.mybatis.mapper.common.base.update)

接口：UpdateByPrimaryKeyMapper<T>  
方法：int updateByPrimaryKey(T record)  
说明：根据主键更新实体全部字段，null值会被更新

接口：UpdateByPrimaryKeySelectiveMapper<T>  
方法：int updateByPrimaryKeySelective(T record)  
说明：根据主键更新属性不为null的值

## Delete(tk.mybatis.mapper.common.base.delete)

接口：DeleteByPrimaryKeyMapper<T>  
方法：int deleteByPrimaryKey(Object key)  
说明：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

接口：DeleteMapper<T>  
方法：int delete(T record)  
说明：根据实体属性作为条件进行删除，查询条件使用等号

## base组合接口(tk.mybatis.mapper.common.base)

接口：BaseSelectMapper<T>

方法：包含上面Select的6个方法

接口：BaseInsertMapper<T>

方法：包含上面Insert的2个方法

接口：BaseUpdateMapper<T>

方法：包含上面Update的2个方法

接口：BaseDeleteMapper<T>

方法：包含上面Delete的2个方法

## CRUD组合接口(tk.mybatis.mapper.common)

接口：BaseMapper<T>

方法：继承了base组合接口中的4个组合接口，包含完整的CRUD方法

## Example方法(tk.mybatis.mapper.common.example)

接口：DeleteByExampleMapper<T>  
方法：int deleteByExample(Object example)  
说明：根据Example条件删除数据

接口：SelectByExampleMapper<T>  
方法：List<T> selectByExample(Object example)  
说明：根据Example条件进行查询

接口：SelectCountByExampleMapper<T>  
方法：int selectCountByExample(Object example)  
说明：根据Example条件进行查询总数

接口：SelectOneByExampleMapper<T>  
方法：T selectOneByExample(Object example)  
说明：根据Example条件进行查询

接口：UpdateByExampleMapper<T>  
方法：int updateByExample(T record, Object example)  
说明：根据Example条件更新实体`record`包含的全部属性，null值会被更新

接口：UpdateByExampleSelectiveMapper<T>  
方法：int updateByExampleSelective(T record, Object example)  
说明：根据Example条件更新实体`record`包含的不是null的属性值

## Example组合接口(tk.mybatis.mapper.common)

接口：ExampleMapper<T>

方法：包含上面Example中的6个方法

## RowBounds方法(tk.mybatis.mapper.common.rowbounds)

接口：SelectByExampleRowBoundsMapper<T>  
方法：List<T> selectByExampleAndRowBounds(Object example, org.apache.ibatis.session.RowBounds rowBounds)  
说明：根据example条件和RowBounds进行分页查询

接口：SelectRowBoundsMapper<T>  
方法：List<T> selectByRowBounds(T record, org.apache.ibatis.session.RowBounds rowBounds)  
说明：根据实体属性和RowBounds进行分页查询

接口：SelectByConditionRowBoundsMapper<T>  
方法：List<T> selectByConditionAndRowBounds(Object condition, org.apache.ibatis.session.RowBounds rowBounds)  
说明：根据example条件和RowBounds进行分页查询，该方法和selectByExampleAndRowBounds完全一样，只是名字改成了Condition

## RowBounds组合接口

接口：RowBoundsMapper<T>

方法：包含上面RowBounds中的前两个方法，不包含selectByConditionAndRowBounds

## special特殊接口(tk.mybatis.mapper.common.special)

接口：InsertListMapper<T>  
方法：int insertList(List<T> recordList)  
说明：批量插入，支持批量插入的数据库可以使用，例如MySQL,H2等，另外该接口限制实体包含`id`属性并且必须为自增列

接口：InsertUseGeneratedKeysMapper<T>  
方法：int insertUseGeneratedKeys(T record)  
说明：插入数据，限制为实体包含`id`属性并且必须为自增列，实体配置的主键策略无效

## MySQL专用(tk.mybatis.mapper.common)

接口：MySqlMapper<T>

继承方法：int insertList(List<T> recordList);

继承方法：int insertUseGeneratedKeys(T record);

说明：该接口不包含方法，继承了special中的InsertListMapper<T>和InsertUseGeneratedKeysMapper<T>

## 乐观锁方法（此部分方法为在原有通用方法上封装，使其乐观更新失败时抛出异常）

接口：MyMapper<T>  
方法：int updateByPrimaryKeyAndVersion(T record)  
说明：根据主键更新实体全部字段，null值会被更新,返回值为0时，抛出一个运行时异常

接口：MyMapper<T>  
方法：int updateByPrimaryKeySelectiveAndVersion(T record)  
说明：根据主键更新属性不为null的值,返回值为0时，抛出一个运行时异常