一、获取新添加数据的主键值

使用场景说明

我们很多时候有这种需求，向数据库插入一条记录后，希望能立即拿到这条记录在数据库中的主键值。

1.1 方式一：useGeneratedKeys

```
<!--
    使用useGeneratedKeys="true" 声明返回主键
    使用keyProperty指定将主键映射到pojo的哪个属性
-->
<insert id="save" useGeneratedKeys="true" keyProperty="id" parameterType="user">
    insert into user (username, birthday, sex, address)
    values (#{username}, #{birthday}, #{sex}, #{address})
</insert>
```

1.2 方式二：selectKey

```
<!--
       keyColumn     指定主键列名称
       keyProperty    指定主键封装到实体的哪个属性
       resultType      指定主键类型
       order              指定在数据插入数据库前(后),执行此语句
-->
<insert id="save" parameterType="user">
    <selectKey keyColumn="id" keyProperty="id" order="AFTER" resultType="int">
        SELECT LAST_INSERT_ID()
    </selectKey>
    insert into user (username, birthday, sex, address)
    values (#{username}, #{birthday}, #{sex}, #{address})
</insert>
```

二、动态SQL

使用场景说明

根据程序运行时传入参数的不同而产生SQL语句结构不同，就是动态SQL。

findByIdAndUsername(User user):根据传入的user对象进行查询，将不为空的属性作为查询条件

```
用户输入的是：用户id
    select * from user where id= #{id}
用户输入的是：用户名
    select * from user where username= #{username}
用户输入的是：用户id和用户名
    select * from user where id= #{id} and username= #{username}
```

动态SQL是Mybatis的强大特性之一，Mybatis3之后，需要了解的动态SQL标签仅仅只有下面几个:

- if choose (when, otherwise)　　用于条件判断
- trim (where, set)　　用于去除分隔符
- foreach　　用于循环遍历

2.1 查询操作优化之if判断【最终拼接多个条件】

2.1.1 接口编写

```
public List<User> findByIdAndUsername(User user);
```

2.1.2 映射文件

```
<select id="findByIdAndUsername" parameterType="user" resultType="user">
    select * from user
    <where>
        <if test="id!=null and id!=0">
            and id=#{id}
        </if>
        <if test="username!=null and username!=''">
            and username=#{username}
        </if>
    </where>
</select>
```

2.1.3 测试

```
@Test
public void findByIdAndUsername(){
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = new User();
    //user.setId(96);
    //user.setUsername("传智播客");
    List<User> list = userDao.findByIdAndUsername(user);
    System.out.println(list);
}
```

2.2 查询操作优化之choose判断【最终拼接一个条件】

2.2.1 接口编写

```
public List<User> findByIdAndUsername(User user);
```

2.2.2 映射文件

```
<select id="findByIdAndUsername" parameterType="user" resultType="user">
    select * from user
    <where>
        <choose>
            <when test="id!=null and id!=0">
                and id=#{id}
            </when>
            <when test="username!=null and username!=''">
                and username=#{username}
            </when>
            <otherwise>
                and 1=1
            </otherwise>
        </choose>
    </where>
</select>
```

2.2.3 测试

```
@Test
public void findByIdAndUsername(){
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = new User();
    user.setId(96);
    //user.setUsername("传智播客");
    List<User> list = userDao.findByIdAndUsername(user);
    System.out.println(list);
}
```

2.3 修改操作优化之set判断【过滤掉不修改的字段】

2.3.1 接口编写

```
public void update(User user);
```

2.3.2 映射文件

```
<update id="update" parameterType="user">
    update user
    <set>
        <if test="username!=null and username!=''">
            username=#{username},
        </if>
        <if test="birthday!=null">
            birthday=#{birthday},
        </if>
        <if test="sex!=null and sex!=''">
            sex=#{sex},
        </if>
        <if test="address!=null and address!=''">
            address=#{address},
        </if>
    </set>
    where id=#{id}
</update>
```

2.3.3 测试

```
@Test
public void update(){
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = new User();
    user.setId(96);
    //user.setUsername("传智播客");
    //user.setBirthday(new Date());
    user.setSex("男");
    user.setAddress("北京");
    userDao.update(user);
}
```

2.4 trim（了解）

trim标签是一个格式化标签，可以完成set或者是where的功能。

```
//使用trim替代where
//prefix 代表必要的时候在trim标签的最前面加上一个where
//prefixOverrides  代表必要的时候干掉trim生成语句中的第一个and或者or
<select id="findByIdAndUsername" parameterType="user" resultType="user">
    select * from user
    <trim prefix="where" prefixOverrides="and">
        <choose>
            <when test="id!=null and id!=0">
                and id=#{id}
            </when>
            <when test="username!=null and username!=''">
                and username=#{username}
            </when>
```

```
            <otherwise>
                and 1=1
            </otherwise>
        </choose>
    </trim>
</select>
```

```
//使用trim替代set
//prefix 代表必要的时候在trim标签的最前面加上一个set
//suffixOverrides  代表必要的时候干掉trim生成语句中的最后一个，
<update id="update" parameterType="user">
    update user
    <trim prefix="set" suffixOverrides=",">
        <if test="username!=null and username!=''">
            username=#{username},
        </if>
        <if test="birthday!=null">
            birthday=#{birthday},
        </if>
        <if test="sex!=null and sex!=''">
            sex=#{sex},
        </if>
        <if test="address!=null and address!=''">
            address=#{address},
        </if>
    </trim>
    where id=#{id}
</update>
```

2.5 循环遍历

foreach主要是用来做数据的循环遍历。

典型的应用场景是SQL中的in语法中，select * from user where uid in (***1,2,3***) 在这样的语句中，传入的参数部分必须依靠 foreach遍历才能实现。我们传入的参数，一般有下面几个形式:

- 集合(List Set)
- 数组
- pojo

```
foreach的选项
    collection：数据源【重点关注这一项，它的值会根据出入的参数类型不同而不同】
    open:开始遍历之前的拼接字符串
    close:结束遍历之后的拼接字符串
    separator:每次遍历之间的分隔符
    item:每次遍历出的数据
    index:遍历的次数，从0开始
```

2.5.1 集合

```
<!--findByIds参数是集合，此时要注意collection中必须放collection或者list-->
<select id="findByIds" parameterType="list" resultType="User">
    <include refid="queryUser"/>
    <where>
        <foreach collection="list" item="id" open="id IN (" separator="," close=")">
            #{id}
        </foreach>
    </where>
</select>
```

```
public List<User> findByIds(List<Integer> ids);//collection="collection"
//或者
public List<User> findByIds(@Param("ids") List<Integer> ids);//collection="ids"
```

2.5.2 数组

```
<!--findByArray参数是数组，此时要注意collection中必须放array-->
<select id="findByArray" parameterType="integer[]" resultType="User">
    <include refid="queryUser"/>
    <where>
        <foreach collection="array" item="id" open="id IN (" separator="," close=")">
            #{id}
        </foreach>
    </where>
</select>
```

```
public List<User> findByArray(Integer[] ids);//collection="array"
//或者
public List<User> findByArray(@Param("ids") Integer[] ids);//collection="ids"
```

2.5.3 pojo

```
<select id="findByVo" parameterType="QueryVo" resultType="User">
    select * from user
    <where>
        <foreach collection="ids" item="id" open="id IN (" separator="," close=")">
            #{id}
        </foreach>
    </where>
</select>
```

```
public List<User> findByVo(QueryVo vo);//在QueryVo中封装了一个List<Integer> ids属性
```

总结:foreach的使用主要是用来遍历数据源。关键点在于数据源的指定:

- 如果是集合，使用collection

- 如果是数组，使用array
- 如果是pojo，使用pojo中的属性名称

2.6 sql片段

2.6.1 编写sql片段

```
<!--sql片段-->
<sql id="queryUser">
    select * from user
</sql>
```

2.6.2 使用sql片段

```
<select id="findByArray" parameterType="integer[]" resultType="User">
    <!--引用sql片段-->
    <include refid="queryUser"/>
    <where>
        <foreach collection="array" item="id" open="id IN (" separator="," close=")">
            #{id}
        </foreach>
    </where>
</select>
```

三、多表关联查询

3.1准备工作

3.1.1 多表关系和javaBean对象关系分析

数据库设计的三种表间关系分别为： 一对一 、一对多（多对一）、 多对多关系。

```
常见示例：
    一对一：人和身份证、QQ号码和QQ详情
    一对多：用户和账户、每个人可以有多张银行卡、但每张银行卡只能属于一个人
    多对多：用户和角色、每个人可以有多个身份、每种身份也可以有多个人
```

而我们做java开发要将表关系兑换成javaBean之间的关系

```
java对象之间的关系：
    一对一：Account对象中有一个User对象的私有属性  User user;
    一对多：User对象中有个Account对象属性  List<Account> accounts;
注：
    数据库中的一对一表关系、在java中变成了两个一对一关系
    数据库中的一对多表关系、在java中变成了一个一对一关系和一个一对多关系

    数据库中的多对多表关系、在java中变成了两个一对对关系
```

3.1.2 创建表并添加测试数据

```
/* 创建账户表 */
CREATE TABLE `account` (
  `ID` int(11) NOT NULL COMMENT '编号',
  `UID` int(11) default NULL COMMENT '用户编号',
  `MONEY` double default NULL COMMENT '金额',
  PRIMARY KEY  (`ID`),
  KEY `FK_Reference_8` (`UID`),
  CONSTRAINT `FK_Reference_8` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

/* 添加账户表测试数据 */
insert  into `account`(`ID`,`UID`,`MONEY`) values (1,41,1000),(2,45,1000),(3,41,2000);

/* 创建角色表 */
CREATE TABLE `role` (
  `ID` int(11) NOT NULL COMMENT '编号',
  `ROLE_NAME` varchar(30) default NULL COMMENT '角色名称',
  `ROLE_DESC` varchar(60) default NULL COMMENT '角色描述',
  PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

/* 添加角色表测试数据 */
insert  into `role`(`ID`,`ROLE_NAME`,`ROLE_DESC`) values (1,'院长','管理整个学院'),(2,'总裁','管理整个公司'),(3,'校长','管理整个学校');

/* 创建用户和角色的中间表 */
CREATE TABLE `user_role` (
  `UID` int(11) NOT NULL COMMENT '用户编号',
  `RID` int(11) NOT NULL COMMENT '角色编号',
  PRIMARY KEY  (`UID`,`RID`),
  KEY `FK_Reference_10` (`RID`),
  CONSTRAINT `FK_Reference_10` FOREIGN KEY (`RID`) REFERENCES `role` (`ID`),
  CONSTRAINT `FK_Reference_9` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

/* 添加用户和角色的中间表测试数据 */
insert  into `user_role`(`UID`,`RID`) values (41,1),(45,1),(41,2);
```

3.2 一对一

多对一的映射方式一般有下面几种：

- 采用外键+别名的形式进行映射
- 采用resultMap形式进行映射
- 采用resultMap + association 形式进行映射 【推荐】

下面分别来看。

3.2.1 采用外键+别名的形式进行映射【了解】

```
<select id="findAll" resultType="com.itheima.domain.Account">
    select
        a.*,
        u.id "user.id",
        u.username "user.username",
        u.birthday "user.birthday",
        u.sex "user.sex",
        u.address "user.address"
    from user u, account a where u.id = a.uid
</select>
```

3.2.2 采用resultMap形式进行映射

```
<resultMap id="accountMap" type="Account">
    <id property="id" column="id"/>
    <result property="money" column="money"/>
    <result property="user.id" column="uid"/>
    <result property="user.username" column="username"/>
    <result property="user.birthday" column="birthday"/>
    <result property="user.sex" column="sex"/>
    <result property="user.address" column="address"/>
</resultMap>
<select id="findAccountWithUser" resultMap="accountMap">
    SELECT * FROM account a LEFT JOIN USER u ON a.uid=u.id
</select>
```

3.2.3 采用resultMap + association 形式进行映射

```
<!--
        association表示要封装一个对象
        property="user"封装的是当前Account对象中的user属性对象
        javaType="user"  指定当前Account对象中的user属的类型
-->
<resultMap id="accountMap" type="Account">
    <id property="id" column="id"/>
    <result property="money" column="money"/>
    <association property="user" javaType="user">
        <result property="id" column="uid"/>
        <result property="username" column="username"/>
        <result property="birthday" column="birthday"/>
        <result property="sex" column="sex"/>
        <result property="address" column="address"/>
    </association>
</resultMap>
<select id="findAccountWithUser" resultMap="accountMap">
    SELECT * FROM account a LEFT JOIN USER u ON a.uid=u.id

</select>
```

3.3 一对多之User下有多个Account

一对多的映射方式一般采用resultMap + collection形式进行实现。

```
<resultMap id="userMap" type="User">
    <id property="id" column="id"/>
    <result property="username" column="username"/>
    <result property="birthday" column="birthday"/>
    <result property="sex" column="sex"/>
    <result property="address" column="address"/>
    <collection property="accounts" ofType="Account">
        <id property="id" column="aid"/>
        <result property="money" column="money"/>
    </collection>
</resultMap>
<select id="findUsersWithAccounts" resultMap="userMap">
    SELECT u.*, a.id aid, a.money FROM USER u LEFT JOIN account a ON a.uid=u.id
</select>
```

3.4 一对多之User下有多个Role

```
<resultMap id="userMapWithRole" type="User">
    <id property="id" column="id"/>
    <result property="username" column="username"/>
    <result property="birthday" column="birthday"/>
    <result property="sex" column="sex"/>
    <result property="address" column="address"/>
    <collection property="roles" ofType="Role">
        <id property="id" column="rid"/>
        <result property="roleName" column="role_name"/>
        <result property="roleDesc" column="role_desc"/>
    </collection>
</resultMap>
<select id="findUsersWithRoles" resultMap="userMapWithRole">
    SELECT * FROM USER u LEFT JOIN user_role ur
    ON u.id=ur.uid LEFT JOIN role r ON ur.rid=r.id
</select>
```