第三阶段 MyBatis-02-动态SQL语句及映射关系

回顾

1. MyBatis框架原理
2. MyBatis的基本数据交互方式
3. MyBatis基础配置
4. MyBatis环境搭建
5. MyBatis的基础CRUD操作

今天任务

```
1. MyBatis动态SQL
2. MyBatis关联映射
3. MyBatis延迟加载
4. MyBatis缓存
```

教学目标

```
1. 掌握MyBatis动态SQL
2. 掌握MyBatis关联映射
3. 掌握MyBatis延迟加载
4. 掌握MyBatis缓存
```

一、MyBatis动态SQL

1、动态SQL简介

MyBatis 的强大特性之一便是它的动态 SQL。如果你有使用 JDBC 或其他类似框架的经验，你就能体会到根据不同条件拼接 SQL 语句有多么痛苦。拼接的时候要确保不能忘了必要的空格，还要注意省掉列名列表最后的逗号。利用动态 SQL 这一特性可以彻底摆脱这种痛苦。

通常使用动态 SQL 不可能是独立的一部分,MyBatis 当然使用一种强大的动态 SQL 语言来改进这种情形,这种语言可以被用在任意的 SQL 映射语句中。

动态 SQL 元素和使用 JSTL 或其他类似基于 XML 的文本处理器相似。在 MyBatis 之前的版本中,有很多的元素需要来了解。MyBatis 3 大大提升了它们,现在用不到原先一半的元素就可以了。MyBatis 采用功能强大的基于 OGNL 的表达式来消除其他元素。

mybatis 的动态sql语句是基于OGNL表达式的。可以方便的在 sql 语句中实现某些逻辑. 总体说来mybatis 动态SQL 语句主要有以下几类:

- if 语句 (简单的条件判断)
- choose (when,otherwize) ,相当于java 语言中的 switch ,与 jstl 中的choose 很类似.
- trim (对包含的内容加上 prefix,或者 suffix 等，前缀，后缀)
- where (主要是用来简化sql语句中where条件判断的，能智能的处理 and or ,不必担心多余导致语法错误)
- set (主要用于更新时)
- foreach (在实现 mybatis in 语句查询时特别有用)

2、进行判断

2.1、if元素

动态 SQL 通常要做的事情是有条件地包含 where 子句的一部分。比如:

```
<select id="findActiveBlogWithTitleLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
</select>
```

这条语句提供了一个可选的文本查找类型的功能。如果没有传入“title”，那么所有处于“ACTIVE”状态的BLOG都会返回；反之若传入了“title”，那么就会把模糊查找“title”内容的BLOG结果返回(就这个例子而言，细心的读者会发现其中的参数值是可以包含一些掩码或通配符的)。

如果想可选地通过“title”和“author”两个条件搜索该怎么办呢？首先，改变语句的名称让它更具实际意义；然后只要加入另一个条件即可。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

Mybatis if 标签可用在许多类型的 SQL 语句中，我们以查询为例。首先看一个很普通的查询：

```
<!-- 查询用户列表，like用户名称 -->
<select id="getUserListLikeName" parameterType="User" resultMap="userResultMap">
    SELECT * from user u
WHERE u.username LIKE CONCAT(CONCAT('%', #{username}),'%')
</select>
```

但是当 username 或 sex 为 null 时，此语句很可能报错或查询结果为空。此时我们使用 if 动态 sql 语句先进行判断，如果值为 null 或等于空字符串，我们就不进行此条件的判断，增加灵活性。

参数为实体类：User。将实体类中所有的属性均进行判断，如果不为空则执行判断条件。

```
<!-- 添加 if(判断参数) - 将实体类 User 不为空的属性作为 where 条件 -->
<select id="getUserList" resultMap="resultMap_User"
parameterType="com.qf.pojo.User">
    SELECT u.username,
           u.password,
           u.sex,
           u.birthday,
```

```
            u.photo,
            u.score,
            u.sign
      FROM user u
     WHERE
    <if test="username !=null ">
        u.username LIKE CONCAT(CONCAT('%', #{username, jdbcType=VARCHAR}),'%')
    </if>
    <if test="sex!= null and sex != '' ">
        AND u.sex = #{Sex, jdbcType=INTEGER}
    </if>
    <if test="birthday != null ">
        AND u.birthday = #{birthday, jdbcType=DATE}
    </if>

    <if test="userId != null and userId != '' ">
        AND id.user_id = #{userId, jdbcType=VARCHAR}
    </if>
</select>
```

使用时比较灵活，创建新的一个这样的实体类，我们需要限制那个条件，只需要附上相应的值就会 where 这个条件，相反不去赋值就可以不在 where 中判断。

```
public void select_by_if() {
    User user = new User();
    user.setUsername("");
    user.setSex(1);
    user.setBirthday(DateUtil.parse("1990-08-18"));
    List<User> userList = this.dynamicSqlMapper.getUserList_if(user);
    for (user u : userList) {
        System.out.println(u.toString());
    }
}
```

我们再看看一下另一个示例，先来看看下面的代码：

```
<select id="dynamicIfTest" parameterType="Blog" resultType="Blog">
        select * from t_blog where 1 = 1
        <if test="title != null">
            and title = #{title}
        </if>
        <if test="content != null">
            and content = #{content}
        </if>
        <if test="owner != null">
            and owner = #{owner}
        </if>
    </select>
```

这条语句的意思非常简单，如果提供了 title 参数，那么就要满足 title=#{title}，同样如果提供了 Content 和 Owner 的时候，它们也需要满足相应的条件，之后就是返回满足这些条件的所有 Blog，这是非常有用的一个功能，以往我们使用其他类型框架或者直接使用 JDBC的时候， 如果我们要达到同样的选择效果的时候，我们就需要拼 SQL 语句，这是极其麻烦的，比起来，上述的动态SQL就比较简单了。

2.2、choose元素

有时候我们并不想应用所有的条件，而只是想从多个选项中选择一个。而使用if标签时，只要test中的表达式为 true，就会执行 if 标签中的条件。MyBatis 提供了 choose 元素。if标签是与(and)的关系，而 choose 是或(or)的关系。

choose标签是按顺序判断其内部when标签中的test条件出否成立，如果有一个成立，则 choose 结束。当 choose 中所有 when 的条件都不满则时，则执行 otherwise 中的sql。类似于Java 的 switch 语句，choose 为 switch，when 为 case，otherwise 则为 default。

例如下面例子，同样把所有可以限制的条件都写上，方面使用。choose会从上到下选择一个when标签的test为true的sql执行。安全考虑，我们使用where将choose包起来，放置关键字多于错误。

```
<!--  choose(判断参数) - 按顺序将实体类 User 第一个不为空的属性作为：where条件 -->
<select id="getUserList_choose" resultMap="resultMap_user"
parameterType="com.qf.pojo.User">
    SELECT *
      FROM User u
    <where>
        <choose>
            <when test="username !=null ">
                u.username LIKE CONCAT(CONCAT('%', #{username,
jdbcType=VARCHAR}),'%')
            </when >
            <when test="sex != null and sex != '' ">
                AND u.sex = #{sex, jdbcType=INTEGER}
            </when >
            <when test="birthday != null ">
                AND u.birthday = #{birthday, jdbcType=DATE}
            </when >
```

```
            <otherwise>
            </otherwise>
        </choose>
    </where>
</select>
```

choose (when,otherwize) ,相当于java 语言中的 switch ,与 jstl 中 的 choose 很类似。

```
<select id="dynamicChooseTest" parameterType="Blog" resultType="Blog">
        select * from t_blog where 1 = 1
        <choose>
            <when test="title != null">
                and title = #{title}
            </when>
            <when test="content != null">
                and content = #{content}
            </when>
            <otherwise>
                and owner = "owner1"
            </otherwise>
        </choose>
    </select>
```

when元素表示当 when 中的条件满足的时候就输出其中的内容，跟 JAVA 中的 switch 效果差不多的是按照条件的顺序，当 when 中有条件满足的时候，就会跳出 choose，即所有的 when 和 otherwise 条件中，只有一个会输出，当所有的我很条件都不满足的时候就输出 otherwise 中的内容。所以上述语句的意思非常简单， 当 title!=null 的时候就输出 and titlte = #{title}，不再往下判断条件，当title为空且 content!=null 的时候就输出 and content = #{content}，当所有条件都不满足的时候就输出 otherwise 中的内容。

3、拼关键字

3.1、where元素

当 where 中的条件使用的 if 标签较多时，这样的组合可能会导致错误。当 java 代码按如下方法调用时：

```
@Test
public void select_test_where() {
    User user = new User();
    user.setUsername(null);
    user.setSex(1);
    List<User> userList = this.dynamicSqlMapper.getUsertList_where(user);
    for (User u : userList ) {
        System.out.println(u.toString());
    }
}
```

如果上面例子，参数 username 为 null，将不会进行列 username 的判断，则会直接导“WHERE AND”关键字多余的错误 SQL。

这时可以使用 where 动态语句来解决。“where”标签会知道如果它包含的标签中有返回值的话，它就插入一个‘where’。此外，如果标签返回的内容是以 AND 或OR 开头的，则它会剔除掉。

上面例子修改为：

```
<select id="getUserList_whereIf" resultMap="resultMap_User"
parameterType="com.qf.pojo.User">
    SELECT u.user_id,
           u.username,
           u.sex,
           u.birthday
      FROM User u
    <where>
        <if test="username !=null ">
            u.username LIKE CONCAT(CONCAT('%', #{username, jdbcType=VARCHAR}),'%')

        </if>
        <if test="sex != null and sex != '' ">
            AND u.sex = #{sex, jdbcType=INTEGER}
        </if>
        <if test="birthday != null ">
            AND u.birthday = #{birthday, jdbcType=DATE}
        </if>
    </where>
</select>
```

where 主要是用来简化 sql 语句中 where 条件判断，自动地处理 AND/OR 条件。

```
<select id="dynamicWhereTest" parameterType="Blog" resultType="Blog">
        select * from t_blog
        <where>
            <if test="title != null">
                title = #{title}
            </if>
            <if test="content != null">
                and content = #{content}
            </if>
            <if test="owner != null">
                and owner = #{owner}
            </if>
        </where>
    </select>
```

where 元素的作用是会在写入 where 元素的地方输出一个 where，另外一个好处是你不需要考虑 where 元素里面的条件输出是什么样子的，MyBatis 会智能的帮处理，如果所有的条件都不满足那么 MyBatis 就会查出所有的记录，如果输出后是 and 开头的，MyBatis 会把第一个and忽略，当然如果是 or 开头的，MyBatis 也会把它忽略；此外，在 where 元素中你不需要考虑空格的问题，MyBatis 会智能的帮你加上。像上述例子中，如果 title=null， 而 content != null，那么输出的整个语句会是 select * from t_blog where content = #{content}，而不是 select * from t_blog where and content = #{content}，因为 MyBatis 会自动地把首个 and / or 给忽略。

3.2、set元素

当 update 语句中没有使用 if 标签时，如果有一个参数为 null，都会导致错误。

当在 update 语句中使用if标签时，如果前面的if没有执行，则或导致逗号多余错误。使用set标签可以将动态的配置 SET 关键字，并剔除追加到条件末尾的任何不相关的逗号。使用 if+set 标签修改后，如果某项为 null 则不进行更新，而是保持数据库原值。如下示例：

```
<!--  if/set(判断参数) - 将实体 User类不为空的属性更新 -->
<update id="updateUser_if_set" parameterType="com.pojo.User">
    UPDATE user
    <set>
        <if test="username!= null and username != '' ">
            username = #{username},
        </if>
        <if test="sex!= null and sex!= '' ">
           sex = #{sex},
        </if>
        <if test="birthday != null ">
            birthday = #{birthday},
        </if>
    </set>
    WHERE user_id = #{userid};
</update>
```

再看看下面的一个示例：

```
<update id="dynamicSetTest" parameterType="Blog">
        update t_blog
        <set>
            <if test="title != null">
                title = #{title},
            </if>
            <if test="content != null">
                content = #{content},
            </if>
            <if test="owner != null">
                owner = #{owner}
            </if>
        </set>
        where id = #{id}
```

```
    </update>
```

set 标签元素主要是用在更新操作的时候，它的主要功能和 where 标签元素其实是差不多的，主要是在包含的语句前输出一个 set，然后如果包含的语句是以逗号结束的话将会把该逗号忽略，如果 set 包含的内容为空的话则会出错。有了 set 元素就可以动态的更新那些修改了的字段。

3.3、trim元素

trim 是更灵活用来去处多余关键字的标签，它可以用来实现 where 和 set 的效果。

```
<!-- 使用 if/trim 代替 where(判断参数) - 将 User 类不为空的属性作为 where 条件 -->
<select id="getUsertList_if_trim" resultMap="resultMap_User">
    SELECT *
      FROM user u
    <trim prefix="WHERE" prefixOverrides="AND|OR">
        <if test="username !=null ">
            u.username LIKE CONCAT(CONCAT('%', #{username, jdbcType=VARCHAR}),'%')

        </if>
        <if test="sex != null and sex != '' ">
            AND u.sex = #{sex, jdbcType=INTEGER}
        </if>
        <if test="birthday != null ">
            AND u.birthday = #{birthday, jdbcType=DATE}
        </if>
    </trim>
</select>
```

trim 代替 set

```
<!-- if/trim代替set(判断参数) - 将 User 类不为空的属性更新 -->
<update id="updateUser_if_trim" parameterType="com.qf.pojo.User">
    UPDATE user
    <trim prefix="SET" suffixOverrides=",">
        <if test="username != null and username != '' ">
            username = #{username},
        </if>
        <if test="sex != null and sex != '' ">
            sex = #{sex},
        </if>
        <if test="birthday != null ">
            birthday = #{birthday},
        </if>

    </trim>
    WHERE user_id = #{user_id}
</update>
```

trim (对包含的内容加上 prefix,或者 suffix 等，前缀，后缀)

```
<select id="dynamicTrimTest" parameterType="Blog" resultType="Blog">
        select * from t_blog
        <trim prefix="where" prefixOverrides="and |or">
            <if test="title != null">
                title = #{title}
            </if>
            <if test="content != null">
                and content = #{content}
            </if>
            <if test="owner != null">
                or owner = #{owner}
            </if>
        </trim>
    </select>
```

trim 元素的主要功能是可以在自己包含的内容前加上某些前缀，也可以在其后加上某些后缀，与之对应的属性是 prefix 和 suffix；可以把包含内容的首部某些内容覆盖，即忽略，也可以把尾部的某些内容覆盖，对应的属性是 prefixOverrides 和 suffixOverrides；正因为 trim 有这样的功能，所以我们也可以非常简单的利用 trim 来代替 where 元素的功能。

4、进行循环

4.1、foreach元素

foreach的主要用在构建in条件中，它可以在SQL语句中进行迭代一个集合。 foreach元素的属性主要有 item，index，collection，open，separator，close。

- item表示集合中每一个元素进行迭代时的别名，
- index指 定一个名字，用于表示在迭代过程中，每次迭代到的位置，
- open表示该语句以什么开始，
- separator表示在每次进行迭代之间以什么符号作为分隔 符，
- close表示以什么结束。

在使用foreach的时候最关键的也是最容易出错的就是collection属性，该属性是必须指定的，但是在不同情况 下，该属性的值是不一样的，主要有一下3种情况： 1. 如果传入的是单参数且参数类型是一个List的时候，collection属性值为list 2. 如果传入的是单参数且参数类型是一个array数组的时候，collection的属性值为array

3.如果传入的参数是多个的时候，我们就需要把它们封装成一个Map了，当然单参数也可以封装成map，实际上如果你在传入参数的时候，在breast里面也是会把它封装成一个Map的，map的key就是参数名，所以这个时候collection属性值就是传入的List或array对象在自己封装的map里面的key

下面分别来看看上述三种情况的示例代码：

4.1.1、单参数List的类型

```
<select id="dynamicForeachTest" resultType="Blog">
           select * from t_blog where id in
       <foreach collection="list" index="index" item="item" open="(" separator="," close=")">
                #{item}
       </foreach>
    </select>
```

上述collection的值为list，对应的Mapper是这样的 public List dynamicForeachTest(List ids); 测试代码：

```
  @Test
      public void dynamicForeachTest() {
          SqlSession session = Util.getSqlSessionFactory().openSession();
          BlogMapper blogMapper = session.getMapper(BlogMapper.class);
           List ids = new ArrayList();
           ids.add(1);
           ids.add(3);
            ids.add(6);
          List blogs = blogMapper.dynamicForeachTest(ids);
          for (Blog blog : blogs)
              System.out.println(blog);
          session.close();
      }
```

4.1.2、单参数array数组的类型

```
 <select id="dynamicForeach2Test" resultType="Blog">
     select * from t_blog where id in
     <foreach collection="array" index="index" item="item" open="(" separator="," close=")">
          #{item}
     </foreach>
 </select>
```

上述collection为array，对应的Mapper代码：

public List dynamicForeach2Test(int[] ids);

对应的测试代码：

```
@Test
public void dynamicForeach2Test() {
        SqlSession session = Util.getSqlSessionFactory().openSession();
        BlogMapper blogMapper = session.getMapper(BlogMapper.class);
        int[] ids = new int[] {1,3,6,9};
        List blogs = blogMapper.dynamicForeach2Test(ids);
        for (Blog blog : blogs)
        System.out.println(blog);
        session.close();
}
```

4.1.3、自己把参数封装成Map的类型

```
<select id="dynamicForeach3Test" resultType="Blog">
        select * from t_blog where title like "%"#{title}"%" and id in
         <foreach collection="ids" index="index" item="item" open="("
separator="," close=")">
              #{item}
         </foreach>
</select>
```

上述collection的值为ids，是传入的参数Map的key，对应的Mapper代码： public List dynamicForeach3Test(Map params);对应测试代码：

```
@Test
    public void dynamicForeach3Test() {
        SqlSession session = Util.getSqlSessionFactory().openSession();
         BlogMapper blogMapper = session.getMapper(BlogMapper.class);
          final List ids = new ArrayList();
          ids.add(1);
          ids.add(2);
          ids.add(3);
          ids.add(6);
         ids.add(7);
         ids.add(9);
        Map params = new HashMap();
         params.put("ids", ids);
         params.put("title", "中国");
        List blogs = blogMapper.dynamicForeach3Test(params);
         for (Blog blog : blogs)
             System.out.println(blog);
         session.close();
     }
```

二、MyBatis关联映射

1、主键映射

1.1、主键映射作用

- 当数据插入操作不关心插入后数据的主键（唯一标识），那么建议使用 *不返回自增主键值* 的方式来配置插入语句，这样可以避免额外的SQL开销.
- 当执行插入操作后需要立即获取插入的自增主键值，比如一次操作中保存一对多这种关系的数据，那么就要使用 *插入后获取自增主键值* 的方式配置.

mybatis进行插入操作时，如果表的主键是自增的，针对不同的数据库相应的操作也不同。基本上经常会遇到的就是Oracle Sequece 和 Mysql 自增主键，解释如下。

1.2、自动递增

一对多的那种表结构，在插入多端数据时，需要获取刚刚保存了的一段的主键。那么这个时候，上述的配置就无法满足需要了。为此我们需要使用mybatis提供的`<selectKey />`来单独配置针对自增逐渐的处理。

1.2.1、Oracle Sequence 配置

```
 <sql id='TABLE_NAME'>TEST_USER</sql> <sql
id='TABLE_SEQUENCE'>SEQ_TEST_USER_ID.nextval</sql>
 <!-- 注意这里需要先查询自增主键值 --> <insert id="insert" parameterType="User">
<selectKey keyProperty="id" resultType="int" order="BEFORE">         select
<include refid="TABLE_SEQUENCE" /> from dual     </selectKey>     insert into
<include refid="TABLE_NAME" /> (ID,NAME,AGE)         values ( #{id}, #{name}, #
{age} ) </insert>
```

当使用了`<selectKey />`后，在实际的插入操作时，mybatis会执行以下两句SQL:

```
select SEQ_TEST_USER_ID.nextval from dual; // 语句1
insert into (ID,NAME,AGE) values ( ?, ?, ? ); // 语句2
```

在执行插入 *语句2* 之前，会先执行 *语句1* 以获取当前的ID值，然后mybatis使用反射调用`User`对象的`setId`方法，将 *语句1* 查询出的值保存在`User`对象中，然后才执行 *语句2* 这样就保证了执行完插入后

```
User user = new User(); user.setName("test"); user.setAge(24);
userMapper.insert(user); System.out.println(user.id); // user.id 不为空
```

user.id`是有值的。

1.2.2、Mysql自增主键配置

针对于Mysql这种自己维护主键的数据库，可以直接使用以下配置在插入后获取插入主键，

```
 <sql id='TABLE_NAME'>TEST_USER</sql>
 <insert id="insert" useGeneratedKeys="true" keyProperty="id"
parameterType="User">     insert into <include refid="TABLE_NAME" /> ( NAME, AGE )
        values ( #{name}, #{age} ) </insert>
```

当然，由于Mysql的自增主键可以通过SQL语句

```
 select LAST_INSERT_ID();
```

来获取的。因此针对Mysql，Mybatis也可配置如下：

```
 <sql id='TABLE_NAME'>TEST_USER</sql>
 <!-- 注意这里需要先查询自增主键值 --> <insert id="insert" parameterType="User">
<selectKey keyProperty="id" resultType="int" order="BEFORE">         SELECT
LAST_INSERT_ID()     </selectKey>     insert into <include refid="TABLE_NAME" />
(ID,NAME,AGE)         values ( #{id}, #{name}, #{age} ) </insert>
```

只不过该中配置需要额外的一条查询SQL!

1.3、非自动递增

如果考虑到插入数据的主键不作为其他表插入数据的外键使用，那么可以考虑使用这种方式。

1.3.1、Oracle Sequence 配置

```
 <sql id='TABLE_NAME'>TEST_USER</sql> <sql
id='TABLE_SEQUENCE'>SEQ_TEST_USER_ID.nextval</sql>
 <!-- 注意这里直接调用sequence的nextval函数 --> <insert id="insert"
parameterType="User">     insert into <include refid="TABLE_NAME" /> (ID,NAME,AGE)
        values ( <include refid="TABLE_SEQUENCE" /> ,#{name}, #{age} ) </insert>
```

当插入语句如上配置时，那么针对如下语句

```
 User user = new User(); user.setName("test"); user.setAge(24);
 userMapper.insert(user); System.out.println(user.id); // user.id 为空
```

`user.id`为空，也就是说如上的配置并不能在完成插入操作后将插入时的主键值存放到保存的对象中。

1.3.2、Mysql自增主键配置

由于mysql数据库中，可以设置表的主键为自增，所以对于Mysql数据库在mybatis配置插入语句时，不指定插入ID字段即可。主键的自增交由Mysql来管理。

```
 <sql id='TABLE_NAME'>TEST_USER</sql>
 <!-- 注意这里的插入SQL中是没有指明ID字段的！  --> <insert id="insert"
parameterType="User">     insert into <include refid="TABLE_NAME" /> (NAME,AGE)
      values (#{name}, #{age} ) </insert>
```

同样，针对Mysql如此配置mybaits，插入完成后`user.id`为空

2、关联映射

2.1、关联映射作用

在现实的项目中进行数据库建模时，我们要遵循数据库设计范式的要求，会对现实中的业务模型进行拆分，封装在不同的数据表中，表与表之间存在着**一对多**或是**多对多**的对应关系。进而，我们对数据库的增删改查操作的主体，也就从单表变成了多表。那么Mybatis中是如何实现这种多表关系的映射呢?

查询结果集ResultMap

resultMap 元素是 MyBatis 中最重要最强大的元素。它就是让你远离 90%的需要从结果 集中取出数据的 JDBC 代码的那个东西，而且在一些情形下允许你做一些 JDBC 不支持的事 情。 事实上, 编写相似于对复杂语句联合映射这些等同的代码，也许可以跨过上千行的代码。

有朋友会问，之前的示例中我们没有用到结果集，不是也可以正确地将数据表中的数据映射到Java对象的属性中吗？是的。这正是resultMap元素设计的初衷，就是简单语句不需要明确的结果映射，而很多复杂语句确实需要描述它们的关系。

- resultMap元素中，允许有以下直接子元素：
- constructor – 类在实例化时，用来注入结果到构造方法中（本文中暂不讲解）
- id – 作用与result相同，同时可以标识出用这个字段值可以区分其他对象实例。可以理解为数据表中的主键，可以定位数据表中唯一一笔记录
- result – 将数据表中的字段注入到Java对象属性中
- association – 关联，简单的讲，就是“有一个”关系，如“用户”有一个“帐号”
- collection – 集合，顾名思议，就是“有很多”关系，如“客户”有很多“订单”
- discriminator – 使用结果集决定使用哪个个结果映射（暂不涉及）

  每个元素的用法及属性我会在下面结合使用进行讲解。

  我们在数据库中额外创建三张数据表，分别表示销售人员、客户，以及销售和客户多对多的对应关系。每个销售、客户都有一个登录帐号。

```
CREATE TABLE `customer` (
  `customer_id` int(10) NOT NULL AUTO_INCREMENT,
  `customer_name` varchar(200) NOT NULL,
  `user_id` int(10) DEFAULT NULL,
  `is_valid` tinyint(4) NOT NULL DEFAULT '1',
  `created_time` datetime NOT NULL,
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`customer_id`),
```

```
  KEY `customer_name` (`customer_name`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `salesman` (
  `sales_id` int(10) NOT NULL AUTO_INCREMENT,
  `sales_name` varchar(64) NOT NULL,
  `sales_phone` varchar(32) DEFAULT NULL,
  `sales_fax` varchar(32) DEFAULT NULL,
  `sales_email` varchar(100) DEFAULT NULL,
  `user_id` int(10) DEFAULT NULL,
  `report_to` int(10) DEFAULT '0',
  `is_valid` tinyint(4) NOT NULL DEFAULT '1',
  `created_time` datetime DEFAULT NULL,
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`sales_id`),
  KEY `sales_name` (`sales_name`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `customer_sales` (
  `id` int(10) NOT NULL AUTO_INCREMENT,
  `customer_id` int(10) NOT NULL,
  `sales_id` int(10) NOT NULL,
  `created_time` datetime NOT NULL,
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `customer_id` (`customer_id`,`sales_id`) USING BTREE,
  KEY `sales_id` (`sales_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

实现销售与登录用户一对一关系

这里采用Mybatis的接口式编程。无论是对单表进行映射，还是对多表映射，步骤都是相同的，唯一的不同就在映射文件的编写上。

首先，我们需要销售创建一个Java类，其中的userInfo属性对应销售的登录用户信息的。

```
public class Sales {
    private int salesId;
    private String salesName;
    private String phone;
    private String fax;
    private String email;
    private int isValid;
    private Timestamp createdTime;
    private Timestamp updateTime;
    private User userInfo;
```

第二步，编写Mybatis映射文件，需要注意的是映射文件的名称空间，要与我们编写的接品的全限定名一致（包名＋接口名）

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

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.qf.dao.ISalesDao">
    <resultMap id="salesResultMap" type="com.qf.pojo.Sales">
        <id property="salesId" column="sales_id" />
        <result property="salesName" column="sales_name" />
        <result property="phone" column="sales_phone" />
        <result property="fax" column="sales_fax" />
        <result property="email" column="sales_email" />

        <!-- 定义多对一关联信息（每个销售人员对应一个登录帐号） -->
        <association property="userInfo" column="user_id" javaType="User"
select="selectUser">
            <id property="userId" column="userId" />
            <result property="userName" column="user_name" />
            <result property="userPassword" column="user_password" />
            <result property="nickName" column="nick_name" />
            <result property="email" column="email" />
            <result property="isValid" column="is_valid" />
            <result property="createdTime" column="created_time" />
            <result property="updateTime" column="update_time" />
        </association>
    </resultMap>

    <select id="selectUser" resultType="User">
        SELECT user_id, user_name, user_password, nick_name, email, is_valid,
created_time
        FROM sys_user WHERE user_id = #{id}
    </select>

    <select id="getById" parameterType="int" resultMap="salesResultMap" >
        SELECT sales_id, sales_name, sales_phone, sales_fax, sales_email, user_id,
is_valid, created_time, update_time
        FROM salesman WHERE sales_id=#{id}
    </select>
</mapper>
```

第三步，将映射文件注册到Mybatis中。

```
<mappers>
    <mapper resource="com/qf/mapping/User.xml" />
    <mapper resource="com/qf/mapping/Sales.xml" />
</mappers>
```

第四步，编写接口

```
public interface ISalesDao {
    public Sales getById(int id);
}
```

第五步，编写测试用例

```
public class SalesDaoTest {

    private Reader reader;
    private SqlSessionFactory sqlSessionFactory;

    @Before
    public void setUp() throws Exception {
        try {
            reader = Resources.getResourceAsReader("mybatis.xml");
        } catch (IOException e) {
            e.printStackTrace();
        }
        sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void getById() {
        SqlSession session = sqlSessionFactory.openSession();
        try {
            ISalesDao sd = session.getMapper(ISalesDao.class);
            Sales sales = sd.getById(2);
            assertNotNull(sales);
            System.out.println(sales);
        } finally {
            session.close();
        }
    }

}
```

下面我们就针对第二步，映射文件中的resultMap编写进行详细讲解。

```
<resultMap id="salesResultMap" type="com.qf.pojo.Sales">
    <id property="salesId" column="sales_id" />
    <result property="salesName" column="sales_name" />
    <result property="phone" column="sales_phone" />
    <result property="fax" column="sales_fax" />
    <result property="email" column="sales_email" />
    <result property="isValid" column="is_valid" />
    <result property="createdTime" column="createdTime" />
    <result property="updateTime" column="update_time" />

    <!-- 定义多对一关联信息（每个销售人员对应一个登录帐号） -->
    <association property="userInfo" column="user_id" javaType="User"
select="selectUser">
        <id property="userId" column="userId" />
        <result property="userName" column="user_name" />
        <result property="userPassword" column="user_password" />
        <result property="nickName" column="nick_name" />
        <result property="email" column="email" />
        <result property="isValid" column="is_valid" />
        <result property="createdTime" column="created_time" />
        <result property="updateTime" column="update_time" />
    </association>
</resultMap>
```

和其他元素一样，我们都需要为其取一个唯一的id，并指定其在Java中对应的类型，由于我没有在Mybatis配置文件中为Sales类指定别名，所以这里使用的是全限定名。

```
<resultMap id="salesResultMap" type="com.qf.pojo.Sales">
```

使用id和result元素指定数据表中字段与Java类中属性的映射关系，除了我phone、fax和email三行映射代码，其余的全部可以省去不写。为什么？这个就像前面示例中使用到的User类一样，Mybatis会自动帮助我们完成映射工作，不需要我们额外编写代码。那么为什么phone、fax和email这三个字段的映射关系不能省略呢？这是因为我在编写Sales类的时候埋下了伏笔，我故意不按照按驼峰规则对这三个属性进行命名，同时也不与数据表中的字段名相同，为了确保可以正确的将字段映射到属性上，我们必须手工编写映射在代码，明确地告诉Mybatis我们的映射规则。

```
<resultMap id="salesResultMap" type="com.qf.pojo.Sales">
    <result property="phone" column="sales_phone" />
    <result property="fax" column="sales_fax" />
    <result property="email" column="sales_email" />
</resultMap>
```

下面重点来了，association元素来帮助我们完成销售与登录用户对应关系的映射。她实现了“有一个”的关系映射，我们需要做的只是告诉Mybatis，这个关系是通过哪一个字段来建立关联的，被关联的对象类型是什么，以及将关联对象映射到哪个属性上面。如果被关联对象的数据结构比较简单，就如本文中的登录用户表这样，那么可以有更简单的写法。

```
<association property="userInfo" column="user_id" javaType="User"
select="selectUser" />
```

我们还需要告诉Mybatis，加载关联的方式。MyBatis 在这方面会有两种不同的方式:

- 嵌套查询:通过执行另外一个 SQL 映射语句来返回预期的复杂类型。
- 嵌套结果:使用嵌套结果映射来处理重复的联合结果的子集。

2.2、嵌套查询映射

我们在这里先使用嵌套查询来实现。使用属性select指定了关联数据的查询语句。

```
<select id="selectUser" resultType="User">
    SELECT user_id, user_name, user_password, nick_name, email, is_valid,
created_time
    FROM sys_user WHERE user_id = #{id}
</select>
```

当对Sales进行映射的时候，Mybatis会使用这个名为selectUser的查询语句去获取相关联的数据信息。这种方法使用起来很简单。但是简单，不代表最好。对于大型数据集合和列表这种方式将会有性能上的问题，就是我们熟知的 “N+1 查询问题”。概括地讲,N+1 查询问题可以是这样引起的:

- 你执行了一个单独的 SQL 语句来获取结果列表(就是“+1”)。
- 对返回的每条记录,你执行了一个查询语句来为每个加载细节(就是“N”)。

这个问题会导致成百上千的 SQL 语句被执行。这通常不是期望的。

MyBatis 能延迟加载这样的查询就是一个好处,因此你可以分散这些语句同时运行的消耗。然而,如果你加载一个列表,之后迅速迭代来访问嵌套的数据,你会调用所有的延迟加载,这样的行为可能是很糟糕的。

2.3、嵌套结果映射

下面我们就来讲一下另一种实式方式：嵌套结果。使用这种方式，就可以有效地避免了N+1问题。

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

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.qf.dao.ISalesDao">
    <resultMap id="salesResultMap" type="com.qf.pojo.Sales">
        <id property="salesId" column="sales_id" />
        <result property="salesName" column="sales_name" />
```

```
        <result property="phone" column="sales_phone" />
        <result property="fax" column="sales_fax" />
        <result property="email" column="sales_email" />
        <result property="isValid" column="is_valid" />
        <result property="createdTime" column="created_time" />
        <result property="updateTime" column="update_time" jdbcType="TIMESTAMP" />

        <!-- 定义多对一关联信息（嵌套结果方式） -->
        <association property="userInfo" resultMap="userResult" />
    </resultMap>

    <resultMap id="userResult" type="User">
        <id property="userId" column="user_id" />
        <result property="userName" column="user_name" />
        <result property="userPassword" column="user_password" />
        <result property="nickName" column="nick_name" />
        <result property="email" column="user_email" />
        <result property="isValid" column="user_is_valid" />
        <result property="createdTime" column="user_created_time" />
        <result property="updateTime" column="user_update_time" />
    </resultMap>

    <select id="getById" parameterType="int" resultMap="salesResultMap">
        SELECT
        sales_id, sales_name, sales_phone, sales_fax, sales_email,
        salesman.is_valid, salesman.created_time, salesman.update_time,
        sys_user.user_id as user_id, user_name, user_password, nick_name,
        email as user_email,
        sys_user.is_valid as user_is_valid, sys_user.created_time as
        user_created_time,
        sys_user.update_time as user_update_time
        FROM
        salesman left outer join sys_user using(user_id)
        WHERE sales_id=#{id}
    </select>
</mapper>
```

和嵌套查询相比，使用嵌套结果方式，在映射文件上主要有以下三处修改：

一、修改association元素，无需指定column，另外将resultType改为使用resultMap。为什么？这是因为后面我们会把select语句改为多表关联查询，这样就会有些字段名是冲突的，我们不得不使用别名。这一点对于Mybatis而言，就相当于字段名发生了变化，那么就需要我们手工来维护映射关系。另外，我们也无需指定javaType属性了，因为在resultMap中，已经指定了对应的Java实体类，这里就可以省略了。

```
<association property="userInfo" resultMap="userResult" />
```

二、为关联结果集编写映射关系，大家可以看到，好多字段名称已经发生了变化，如is_valid这个字段由于salesman和sys_user表中都存在这个字段，所以我们不得不为其起了一个别名user_is_valid。

```
<resultMap id="userResult" type="User">
        <id property="userId" column="user_id" />
        <result property="userName" column="user_name" />
        <result property="userPassword" column="user_password" />
        <result property="nickName" column="nick_name" />
        <result property="email" column="user_email" />
        <result property="isValid" column="user_is_valid" />
        <result property="createdTime" column="user_created_time" />
        <result property="updateTime" column="user_update_time" />
    </resultMap>
```

三、修改查询语句，由单表查询改表多表关联查询

```
<select id="getById" parameterType="int" resultMap="salesResultMap">
    SELECT sales_id, sales_name, sales_phone, sales_fax, sales_email,
           salesman.is_valid, salesman.created_time, salesman.update_time,
           sys_user.user_id as user_id, user_name, user_password, nick_name,
           email as user_email,
           sys_user.is_valid as user_is_valid, sys_user.created_time as
           user_created_time,
           sys_user.update_time as user_update_time
    FROM salesman left outer join sys_user using(user_id)
    WHERE sales_id=#{id}
</select>
```

至此，关联映射已讲解完了。还有集合映射没有讲，哇咔咔，内空实在是太多了～～～～今晚通宵也未必能写得完了。暂时先写到这儿吧，下回再继续讲解如何实现多对多的集合映射。

3、集合映射

3.1、集合映射作用

集合映射，实现销售与客户的多对多关系

第一步，在动手编写映射文件之前，我们需要对Sales类增加一个List属性，用以保存销售员对应的客户列表。

```
private List<Customer> customers;

public Sales() {
    super();
    this.setCustomers(new ArrayList<Customer>());
}

public List<Customer> getCustomers() {
```

```
    return customers;
}

protected void setCustomers(List<Customer> customers) {
    this.customers = customers;
}
```

同时增加一个客户类。

```
public class Customer {
    private int customerId;
    private String customerName;
    private int isValid;
    private Timestamp createdTime;
    private Timestamp updateTime;
    private User userInfo;
```

第二步，修改映射文件。我们先使用嵌套查询方式来实现为销售加载客户列表。首先在resultMap中增加客户集合映射的定义。

3.2、嵌套查询映射

```
<!-- 定义一对多集合信息（每个销售人员对应多个客户） -->
<collection property="customers" javaType="ArrayList" column="sales_id"
ofType="Customer" select="getCustomerForSales" />
```

集合映射的定义与关联映射定义很相似，除了关键字不同外，还多了两个属性JavaType和ofType。

property用于指定在Java实体类是保存集合关系的属性名称

JavaType用于指定在Java实体类中使用什么类型来保存集合数据，多数情况下这个属性可以省略的。

column用于指定数据表中的外键字段名称。

ofType用于指定集合中包含的类型。

select用于指定查询语句。

然后再定义查询客户的查询语句。

```
<select id="getCustomerForSales" resultType="com.qf.pojo.Customer">
    SELECT c.customer_id, c.customer_name, c.user_id, c.is_valid,
    c.created_time, c.update_time
    FROM customer c INNER JOIN customer_sales s USING(customer_id)
    WHERE s.sales_id = #{id}
</select>
```

需要注意的是，无论是关联还是集合，在嵌套查询的时候，查询语句的定义都不需要使用parameterType属性定义传入的参数类型，因为通常作为外键的，都是简单数据类型，查询语句会自动使用定义在association或是collection元素上column属性作为传入参数的。

运行测试用例，看到如下结果就说明我们的映射文件是正确的了。

3.3、嵌套结果映射

```
<resultMap id="salesResultMap" type="com.qf.pojo.Sales">
    <id property="salesId" column="sales_id" />
    <result property="salesName" column="sales_name" />
    <result property="phone" column="sales_phone" />
    <result property="fax" column="sales_fax" />
    <result property="email" column="sales_email" />
    <result property="isValid" column="is_valid" />
    <result property="createdTime" column="created_time" />
    <result property="updateTime" column="update_time" />

    <!-- 定义多对一关联信息（嵌套结果方式） -->
    <association property="userInfo" resultMap="userResult" />

    <!-- 定义一对多集合信息（每个销售人员对应多个客户） -->
    <!-- <collection property="customers" column="sales_id"
select="getCustomerForSales" /> -->

    <collection property="customers" ofType="com.qf.pojo.Customer">
        <id property="customerId" column="customer_id" />
        <result property="customerName" column="customer_name" />
        <result property="isValid" column="is_valid" />
        <result property="createdTime" column="created_time" />
        <result property="updateTime" column="update_time" />
        <!-- 映射客户与登录用户的关联关系，请注意columnPrefix属性 -->
        <association property="userInfo" resultMap="userResult" columnPrefix="cu_"
/>
    </collection>
</resultMap>
```

这里将客户的映射关系直接写在了销售的resultMap中。上述代码与关联映射十分相似，只是有一点需要朋友们留心，那就是在对客户数据进行映射的时候，**我们使用了association元素的一个新的属性columnPrefix**。这个属性是做什么用的呢？从名字上理解，就是给每个栏位之前加上前缀。Bingo！答对了，那么什么情况下会使用到这个属性呢？后面我们会结合着修改后的查询语句来说明这个属性的使用场景。请耐心的往下看。：）

映射结果修改好了，紧接着我们就要修改查询语句了。

```
<select id="getById" parameterType="int" resultMap="salesResultMap">
    SELECT
        s.sales_id, s.sales_name, s.sales_phone, s.sales_fax, s.sales_email,
        s.is_valid, s.created_time, s.update_time,
```

```
        su.user_id as user_id, su.user_name, su.user_password, su.nick_name,
        su.email as user_email,
        su.is_valid as user_is_valid,
        su.created_time as user_created_time,
        su.update_time as user_update_time,
        c.customer_id, c.customer_name, c.is_valid as customer_is_valid,
        c.created_time as customer_created_time,
        c.update_time as customer_update_time,
        cu.user_id as cu_user_id, cu.user_name as cu_user_name, cu.user_password
as cu_user_password,
        cu.nick_name as cu_nick_name, cu.email as cu_user_email, cu.is_valid as
cu_user_is_valid,
        cu.created_time as cu_user_created_time, cu.update_time as
cu_user_update_time
    FROM
        salesman s LEFT OUTER JOIN sys_user su ON s.user_id = su.user_id
        INNER JOIN customer_sales cs USING(sales_id)
        LEFT OUTER JOIN customer c USING(customer_id)
        LEFT OUTER JOIN sys_user cu ON c.user_id = cu.user_id
    WHERE sales_id=#{id}
</select>
```

这个语句乍看起来有些复杂，其实很容易理解。这里用到了四张数据表，销售、客户、客房销售关系表和登录用户表。具体的字段我就不说了，主要说一下这个登录用户表。这张数据表在查询语句中出现了两次，为什么呢？因为销售与登录用户有关联关系，同样地，客户也与登录用户表有关联关系，所以我们需要对用户表进行两次Join操作。

那么问题来了，销售要用户有关联，客户也要与用户有关联，这种映射语句应该如何写呢？难道要对用户表写两次映射？聪明的朋友一定会说，我们可以复用之前写过的用户映射结果集呀！答案是肯定的。我们不妨在这里再次贴出这段代码，一起回忆一下。

```
<resultMap id="userResult" type="User">
    <id property="userId" column="user_id" />
    <result property="userName" column="user_name" />
    <result property="userPassword" column="user_password" />
    <result property="nickName" column="nick_name" />
    <result property="email" column="user_email" />
    <result property="isValid" column="user_is_valid" />
    <result property="createdTime" column="user_created_time" />
    <result property="updateTime" column="user_update_time" />
</resultMap>
```

数据表中的字段与Java实体类中的属性的映射关系是一一对应的，Mybatis会根据我们定义的映射关系，将数据表中字段的映射到Java实体类属性上。

可是我们的查询语句中对用户表进行了两次Join操作，第一次是销售与用户的Join，第二次是客户与用户的Join。而SQL语句是不允许在同一条查询语句中出现相同字段名的（虽然我们有时候会这样写，但是数据库会自动帮我们为重名的字段名起个别名的，比如在字段名后添加数字）。如果我们为第二次Join进来的用户表中的字段使用别名方式，那么就会导致映射的到客户类中的用户信息缺失，因为字段名与我们在映射文件中的定义不一致。如何解决这个问题呢？这时候该columnPrefix属性出场了。

Mybatis也考虑到这种情况的出现，她允许我们在重复出现的字段名前加上一个**统一的字符前缀**，这样就可以有效的避免字段重名，又可以复用之前定义的映射结果集。

在上述的查询语句中，我们为第二次Join进来的用户表中的字段都加上了“cu”*做为区分重名字段的前缀，同时使用columnPrefix属性告诉Mybatis在第二次对用户表映射的时候，将字段名是以“cu”*打头的字段值映射到Java实体类属性当中。这样就可以正确的把客户与用户的关联信息映射到Customer对象当中了。

```
<association property="userInfo" resultMap="userResult" columnPrefix="cu_" />
```

我们之前在User.xml文件中定义过用户表的映射结果集，现在在Sales.xml中也需要使用到同样的结果集，是否可以直接跨文件引用呢？答案是肯定的了，不然对于同一个映射结果集，我们要多处编写，多处维护，这样不仅工作量大，对日后的维护也带来了一定的麻烦。我们只需要在引用处使用结果集的全限定名就可以了。

```
    <resultMap id="salesResultMap" type="com.qf.pojo.Sales">
        <id property="salesId" column="sales_id" />
        <result property="salesName" column="sales_name" />
        <result property="phone" column="sales_phone" />
        <result property="fax" column="sales_fax" />
        <result property="email" column="sales_email" />
        <result property="isValid" column="is_valid" />
        <result property="createdTime" column="created_time" />
        <result property="updateTime" column="update_time" />

        <!-- 定义多对一关联信息（嵌套查询方式） -->
        <!-- <association property="userInfo" column="user_id" javaType="User"
            select="selectUser" fetchType="lazy"> </association> -->

        <!-- 定义多对一关联信息（嵌套结果方式） -->
        <association property="userInfo" resultMap="com.qf.xml.user.userResult" />

        <!-- 定义一对多集合信息（每个销售人员对应多个客户） -->
        <!-- <collection property="customers" column="sales_id"
select="getCustomerForSales"
            /> -->

        <collection property="customers" ofType="com.qf.pojo.Customer">
            <id property="customerId" column="customer_id" />
            <result property="customerName" column="customer_name" />
            <result property="isValid" column="is_valid" />
            <result property="createdTime" column="created_time" />
```

```
            <result property="updateTime" column="update_time" />
            <association property="userInfo"
resultMap="com.qf.xml.user.userResult" columnPrefix="cu_" />
        </collection>
    </resultMap>
```

4、鉴别器

4.1、鉴别器的作用

鉴别器在于确定使用那个ResultMap来映射SQL查询语句，在实现中我们往往有一个基类，然后可以派生一些类。比如我们要选择一群人可以用List，然而Person里面有个性别sex，根据它还可以分为Male或者Female。鉴别器就要根据sex决定用Male还是用Female相关的Mapper进行映射。

这些话还是很抽象，不过说起鉴别器，语言真的不好用描述，不过不要紧，我们来看一个实例就豁然开朗了，我们知道在上篇中我们已经有了一个员工的POJO，然后继承这个POJO分成一个男性，一个女性的POJO。

4.2、鉴别器的使用

当我们查询一批员工的时候，我们希望的是返回一个List,而里面的元素根据性别(sex)自动匹配是MaleEmployee或者是FemaleEmployee，于是我们需要根据sex的值去决定使用MaleEmployee或者是FemaleEmployee的resultMap去映射，这便是鉴别器。

让我们来定义employ的mapper,xml代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.qf.mapper.EmployeeMapper">
    <resultMap id="employeeMap" type="com.qf.pojo.Employee">
        <id property="id" column="id" />
        <result property="empName" column="emp_name" />
        <result property="sex" column="sex" />
        <association property="employeeCard" column="id"
            select="com.qf.mapper.EmployeeCardMapper.getEmployeeCardByEmpId" />
        <collection property="projectList" column="id"
            select="com.qf.mapper.ProjectMapper.findProjectByEmpId" />
        <discriminator javaType="int" column="sex">
            <case value="1" resultMap="maleEmployeeMap" />
            <case value="2" resultMap="femaleEmployeeMap" />
        </discriminator>
    </resultMap>

    <select id="getEmployee" parameterType="int" resultMap="employeeMap">
        select id, emp_name as empName, sex from t_employee where id =#{id}
    </select>
```

```
    <resultMap id="maleEmployeeMap" type="com.qf.pojo.MaleEmployee"
extends="employeeMap">
        <collection property="prostateList"
select="com.qf.mapper.MaleEmployeeMapper.findProstateList" column="id" />
    </resultMap>

    <resultMap id="femaleEmployeeMap" type="com.qf.pojo.FemaleEmployee"
extends="employeeMap">
        <collection property="uterusList"
select="com.qf.mapper.FemaleEmployeeMapper.findUterusList" column="id" />
    </resultMap>
</mapper>
```

我们这里定义了employee的resultMap，它除了级联其他的和平时我们定义的没什么不一样。这里先不看别的级联，先看看鉴别器：元素，我们定义了用javaType说明它用的是整数作为参数，而column指的是SQL对应的列为sex。

那么定义的是你的条件分支：

当sex=1时候，采用maleEmployeeMap；

当sex=2时，采用femaleEmployeeMap。

maleEmployeeMap和femaleEmployeeMap都继承了employeeMap,并且扩展了一个属性，它们用select属性，来定义如何取对应的属性数据。要记住下面这句话，后面我们还将讨论它：*这里使用了全限定路径，其次用column="id"作为参数传递，如果是多个参数的，需要用逗号分隔。

3、关联Mapper：

上面我们看到了我们使用了select关联其他的sql语句，而select里面给的就是一个全限定的路径。分别是：

com.qf.mapper.MaleEmployeeMapper.findProstateList

和

com.qf.mapper.FemaleEmployeeMapper.findUterusList

现在让我们看看这两个Mapper是怎么样的：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="comqf.mapper.MaleEmployeeMapper">
    <select id="findProstateList" parameterType="int" resultType="string">
        select prostate from t_healthy_male where emp_id = #{emp_id}
    </select>
</mapper>


<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
```

```
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="comqf.mapper.FemaleEmployeeMapper">
    <select id="findUterusList" parameterType="int" resultType="string">
        select uterus from t_healthy_female where emp_id = #{emp_id}
    </select>
</mapper>
```

显然他们都比较简单，和我们定义的普通Mapper没什么区别。

三、性能优化

1、延迟加载

1.1、 什么是延迟加载

resultMap中的association和collection标签具有延迟加载的功能。

延迟加载的意思是说，在关联查询时，利用延迟加载，先加载主信息。需要关联信息时再去按需加载关联信息。这样会大大提高数据库性能，因为查询单表要比关联查询多张表速度要快。

1.2 、设置延迟加载

Mybatis默认是不开启延迟加载功能的，我们需要手动开启。

需要在SqlMapConfig.xml文件中，在标签中开启延迟加载功能。

lazyLoadingEnabled、aggressiveLazyLoading

设置项	描述	允许值	默认值
lazyLoadingEnabled	全局性设置懒加载。如果设为‘false’，则所有相关联的都会被初始化加载。	true | false	true
aggressiveLazyLoading	当设置为‘true’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。	true | false	true

1.3、使用association进行延迟加载

1.3.1 需求

查询订单并且关联查询用户信息（对用户信息的加载要求是按需加载）

1.3.2 编写映射文件

需要定义两个mapper的方法对应的statement。

1、只查询订单信息

SELECT * FROM orders

在查询订单的statement中使用association去延迟加载（执行）下边的satatement(关联查询用户信息)

```
<!-- 定义OrdersUserLazyLoadingRstMap -->
<resultMap type="com.qf.mybatis.po.Orders" id="OrdersUserLazyLoadingRstMap">
    <id column="id" property="id" />
    <result column="user_id" property="userId" />
    <result column="number" property="number" />
    <result column="createtime" property="createtime" />
    <result column="note" property="note" />
    <!-- 延迟加载用户信息 -->
    <!-- select：指定延迟加载需要执行的statement的id（是根据user_id查询用户信息的
statement）
       我们使用UserMapper.xml中的findUserById完成根据用户ID（user_id）查询用户信
       如果findUserById不在本mapper中，前边需要加namespace
    -->
    <!-- column：主信息表中需要关联查询的列，此处是user_id -->
    <association property="user"
select="com.qf.mybatis.mapper.UserMapper.findUserById" column="user_id">
</association>
</resultMap>
<!-- 查询订单信息，延迟加载关联查询的用户信息 -->
<select id="findOrdersUserLazyLoading" resultMap="OrdersUserLazyLoadingRstMap">
    SELECT * FROM orders
</select>
```

2、关联查询用户信息

通过上边查询到的订单信息中user_id去关联查询用户信息

使用UserMapper.xml中的findUserById

```
<select id="findUserById" parameterType="int"
       resultType="com.qf.mybatis.po.User">
    SELECT * FROM user WHERE id = #{id}
</select>
```

上边先去执行findOrdersUserLazyLoading，当需要去查询用户的时候再去执行findUserById，通过resultMap的定义将延迟加载执行配置起来。

1.3.3 加载映射文件

<package name="com.qf.mybatis.mapper”/>

1.3.4 编写mapper接口

// 查询订单信息，延迟加载关联查询的用户信息

public List findOrdersUserLazyLoading();

1.3.5 编写测试代码

思路：

1、执行上边mapper方法（findOrdersUserLazyLoading），内部去调用com.qf.mybatis.mapper.OrdersMapper中的findOrdersUserLazyLoading只查询orders信息（单表）。

2、在程序中去遍历上一步骤查询出的List，当我们调用Orders中的getUser方法时，开始进行延迟加载。

3、执行延迟加载，去调用UserMapper.xml中findUserbyId这个方法获取用户信息。

```
@Test
public void testFindOrdersUserLazyLoading() {
    // 创建sqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 通过SqlSession构造usermapper的代理对象
    OrdersMapper ordersMapper = sqlSession.getMapper(OrdersMapper.class);
    // 调用usermapper的方法
    List<Orders> list = ordersMapper.findOrdersUserLazyLoading();
    for(Orders orders : list){
       System.out.println(orders.getUser());
    }
    // 释放SqlSession
    sqlSession.close();
}
```

1.4、延迟加载思考

不使用mybatis提供的association及collection中的延迟加载功能，如何实现延迟加载？？

实现方法如下：

定义两个mapper方法：

1、查询订单列表

2、根据用户id查询用户信息

实现思路：

先去查询第一个mapper方法，获取订单信息列表

在程序中（service），按需去调用第二个mapper方法去查询用户信息。

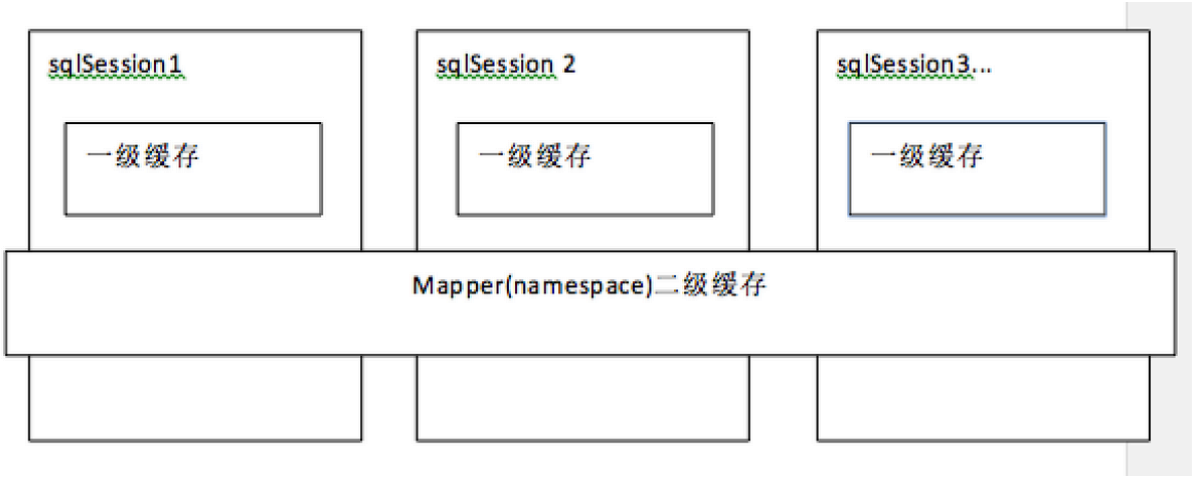
总之：

使用延迟加载方法，先去查询简单的sql（最好单表，也可以关联查询），再去按需要加载关联查询的其它信息。

2、一级缓存

2.1 mybatis缓存分析

mybatis提供查询缓存，如果缓存中有数据就不用从数据库中获取，用于减轻数据压力，提高系统性能。



一级缓存是SqlSession级别的缓存。在操作数据库时需要构造 sqlSession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据。不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

Mybatis的缓存，包括一级缓存和二级缓存

一级缓存指的就是sqlsession，在sqlsession中有一个数据区域，是map结构，这个区域就是一级缓存区域。一级缓存中的key是由sql语句、条件、statement等信息组成一个唯一值。一级缓存中的value，就是查询出的结果对象。

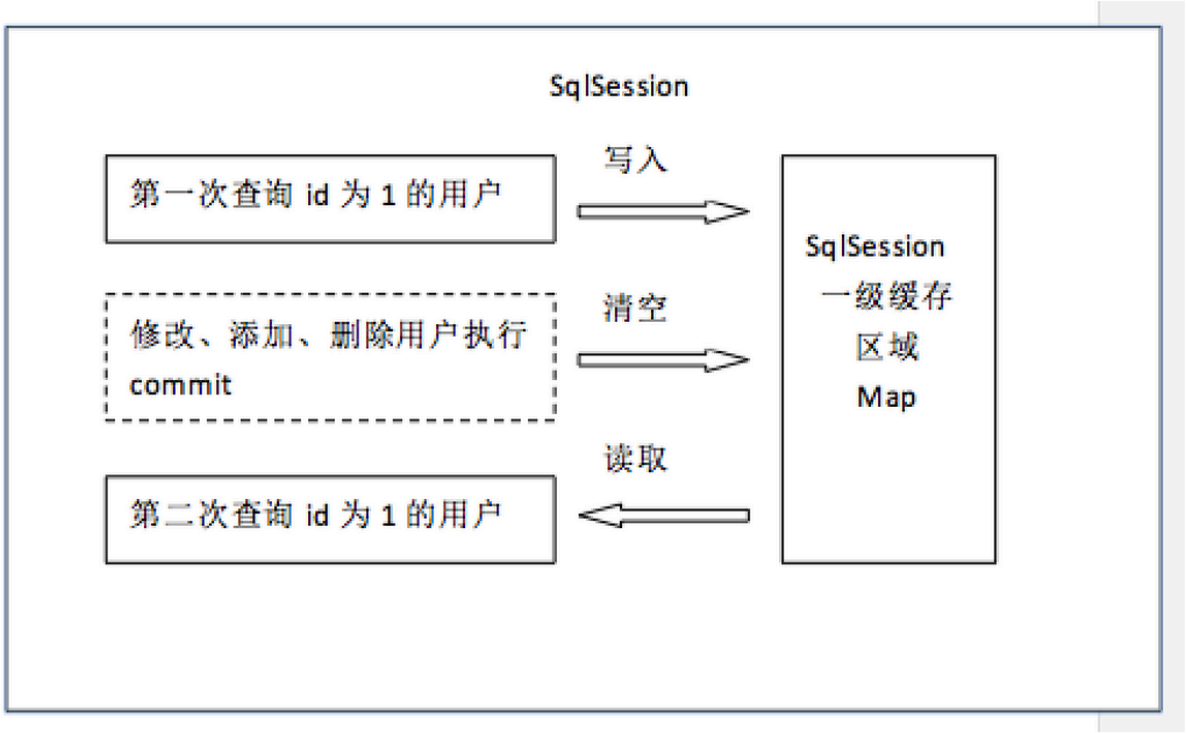
二级缓存指的就是同一个namespace下的mapper，二级缓存中，也有一个map结构，这个区域就是二级缓存区域。二级缓存中的key是由sql语句、条件、statement等信息组成一个唯一值。二级缓存中的value，就是查询出的结果对象。

一级缓存是默认使用的。

二级缓存需要手动开启。

2.2、 一级缓存

2.3、原理



第一次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，如果没有，从数据库查询用户信息。得到用户信息，将用户信息存储到一级缓存中。

如果sqlSession去执行commit操作（执行插入、更新、删除），清空SqlSession中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，避免脏读。

第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息。

Mybatis默认支持一级缓存。

2.4、测试1

```
@Test
    public void testOneLevelCache() {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        // 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库
        User user1 = mapper.findUserById(1);
        System.out.println(user1);
        // 第二次查询ID为1的用户
        User user2 = mapper.findUserById(1);
        System.out.println(user2);
        sqlSession.close();
    }
```

2.5、测试2

```
@Test
    public void testOneLevelCache() {
```

```
        SqlSession sqlSession = sqlSessionFactory.openSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        // 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库
        User user1 = mapper.findUserById(1);
        System.out.println(user1);
       User user = new User();
        user.setUsername("tom");
        user.setAddress("北京");
        //执行增删改操作，清空缓存
        mapper.insertUser(user);
        // 第二次查询ID为1的用户
        User user2 = mapper.findUserById(1);
        System.out.println(user2);
        sqlSession.close();
    }
```

2.6、应用

正式开发，是将mybatis和spring进行整合开发，事务控制在service中。

一个service方法中包括 很多mapper方法调用。

service{

//开始执行时，开启事务，创建SqlSession对象

//第一次调用mapper的方法findUserById(1)

//第二次调用mapper的方法findUserById(1)，从一级缓存中取数据

//方法结束，sqlSession关闭

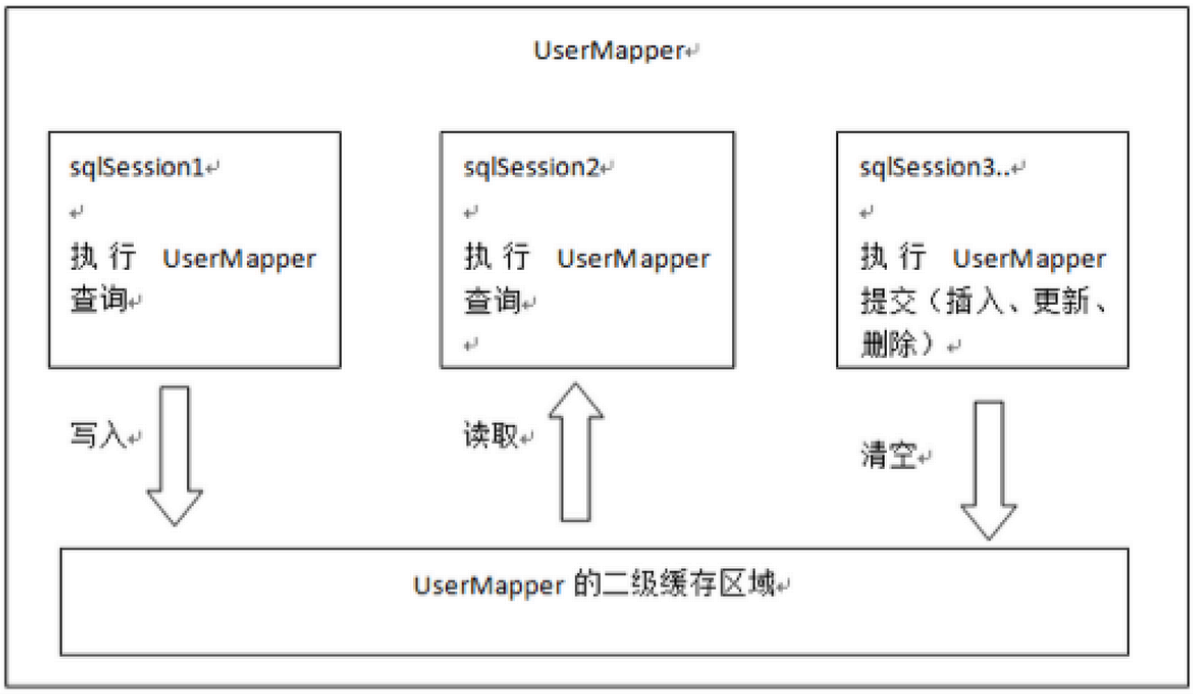
}

如果是执行两次service调用查询相同 的用户信息，不走一级缓存，因为session方法结束，sqlSession就关闭，一级缓存就清空。

3、二级缓存

3.1、原理

下图是多个sqlSession请求UserMapper的二级缓存图解。



二级缓存是mapper级别的。

第一次调用mapper下的SQL去查询用户信息。查询到的信息会存到该mapper对应的二级缓存区域内。

第二次调用相同namespace下的mapper映射文件中相同的SQL去查询用户信息。会去对应的二级缓存内取结果。

如果调用相同namespace下的mapper映射文件中的增删改SQL，并执行了commit操作。此时会清空该namespace下的二级缓存。

3.2、开启二级缓存

Mybatis默认是没有开启二级缓存

1、 在核心配置文件SqlMapConfig.xml中加入以下内容（开启二级缓存总开关）：

在settings标签中添加以下内容：

2、 在UserMapper映射文件中，加入以下内容，开启二级缓存：

3.3、实现序列化

由于二级缓存的数据不一定都是存储到内存中，它的存储介质多种多样，所以需要给缓存的对象执行序列化。

如果该类存在父类，那么父类也要实现序列化。

3.4、测试1

```
@Test

    public void testTwoLevelCache() {

        SqlSession sqlSession1 = sqlSessionFactory.openSession();
```

```
        SqlSession sqlSession2 = sqlSessionFactory.openSession();
        SqlSession sqlSession3 = sqlSessionFactory.openSession();
        UserMapper mapper1 = sqlSession1.getMapper(UserMapper.class);
        UserMapper mapper2 = sqlSession2.getMapper(UserMapper.class);
        UserMapper mapper3 = sqlSession3.getMapper(UserMapper.class);
        // 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库
        User user1 = mapper1.findUserById(1);
        System.out.println(user1);
        // 关闭SqlSession1
        sqlSession1.close();
        // 第二次查询ID为1的用户
        User user2 = mapper2.findUserById(1);
        System.out.println(user2);
        // 关闭SqlSession2
        sqlSession2.close();
    }
```

Cache Hit Radio： 缓存命中率

第一次缓存中没有记录，则命中率0.0；

第二次缓存中有记录，则命中率0.5（访问两次，有一次命中）

3.5、测试2

```
@Test

    public void testTwoLevelCache() {

        SqlSession sqlSession1 = sqlSessionFactory.openSession();

        SqlSession sqlSession2 = sqlSessionFactory.openSession();

        SqlSession sqlSession3 = sqlSessionFactory.openSession();
        UserMapper mapper1 = sqlSession1.getMapper(UserMapper.class);

        UserMapper mapper2 = sqlSession2.getMapper(UserMapper.class);

        UserMapper mapper3 = sqlSession3.getMapper(UserMapper.class);

        // 第一次查询ID为1的用户，去缓存找，找不到就去查找数据库

        User user1 = mapper1.findUserById(1);

        System.out.println(user1);

        // 关闭SqlSession1

        sqlSession1.close();
```

```
        //修改查询出来的user1对象，作为插入语句的参数
        user1.setUsername("tom");
        user1.setAddress("北京");

        mapper3.insertUser(user1);

        // 提交事务
        sqlSession3.commit();
        // 关闭SqlSession3
        sqlSession3.close();

        // 第二次查询ID为1的用户
        User user2 = mapper2.findUserById(1);
        System.out.println(user2);
        // 关闭SqlSession2
        sqlSession2.close();
    }
```

根据SQL分析，确实是清空了二级缓存了。

3.6、禁用二级缓存

该statement中设置useCache=false，可以禁用当前select语句的二级缓存，即每次查询都是去数据库中查询，默认情况下是true，即该statement使用二级缓存。

<select id="findUserById" parameterType="int"

resultType="com.qf.mybatis.po.User" useCache="true">

SELECT * FROM user WHERE id = #{id}

3.7、刷新二级缓存

该statement中设置flushCache=true可以刷新当前的二级缓存，默认情况下如果是select语句，那么flushCache是false。如果是insert、update、delete语句，那么flushCache是true。

如果查询语句设置成true，那么每次查询都是去数据库查询，即意味着该查询的二级缓存失效。

如果查询语句设置成false，即使用二级缓存，那么如果在数据库中修改了数据，而缓存数据还是原来的，这个时候就会出现脏读。

flushCache设置如下：

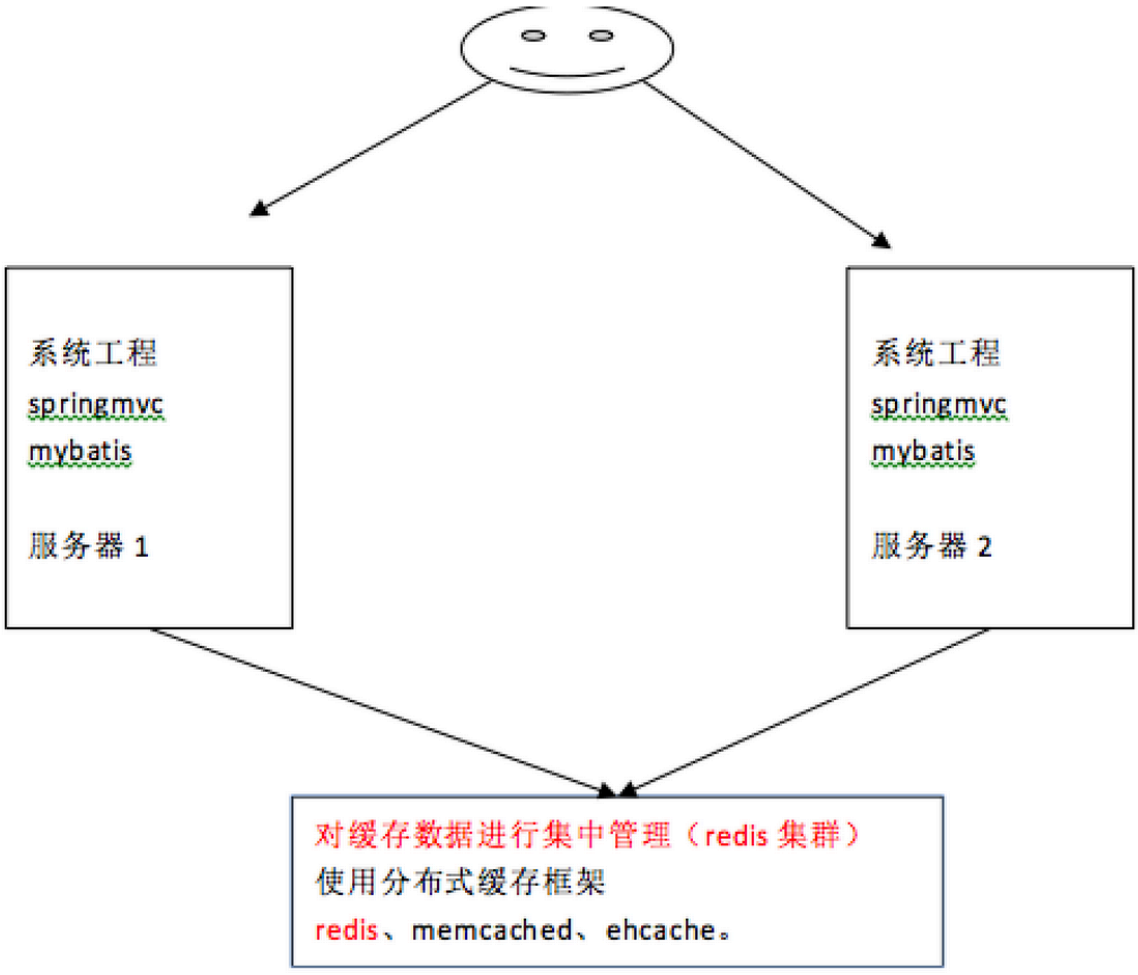
```
<select id="findUserById" parameterType="int"
 resultType="com.qf.mybatis.po.User" useCache="true" flushCache="true">
 SELECT * FROM user WHERE id = #{id}
```

3.8、整合ehcache（了解）

Ehcache是一个分布式缓存。

3.8.1、分布式缓存

系统为了提高性能，通常会对系统采用分布式部署（集群部署方式）



不使用分布式缓存，缓存的数据在各个服务单独存储，不方便开发。所以要使用分布式缓存对缓存数据进行集中式管理。

Mybatis自身无法实现分布式缓存，需要和其它分布式缓存框架进行整合。

3.8.2、整合思路（重点）

Mybatis提供了一个cache接口，同时它自己有一个默认的实现类 PerpetualCache。

通过实现cache接口可以实现mybatis缓存数据通过其他缓存数据库整合，mybatis的特长是sql，缓存数据管理不是mybatis的特长，为了提高mybatis的性能，所以需要mybatis和第三方缓存数据库整合，比如ehcache、memcache、redis等

Mybatis提供接口如下：

Mybatis的默认实现类：

3.8.3、 整合ehcache的步骤

- 引入ehcache的jar包；
- 在mapper映射文件中，配置cache标签的type为ehcache对cache接口的实现类类型。
- 加入ehcache的配置文件

3.8.4、第一步：引入ehcache的jar包

3.8.5、第二步：配置cache的type属性

```
<!-- 使用默认二级缓存 -->
<cache type="org.mybatis.caches.ehcache.EhcacheCache" />
```

3.8.6、第三步：添加ehcache的配置文件

在classpath下添加ehcache.xml

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
    <!-- 缓存数据要存放的磁盘地址 -->
    <diskStore path="F:\develop\ehcache" />
    <!-- diskStore：指定数据在磁盘中的存储位置。 defaultCache：当借助
CacheManager.add("demoCache")创建Cache时，EhCache便会采用<defalutCache/>指定的的管理
策略
        以下属性是必须的： maxElementsInMemory - 在内存中缓存的element的最大数目
maxElementsOnDisk
        - 在磁盘上缓存的element的最大数目，若是0表示无穷大 eternal - 设定缓存的
elements是否永远不过期。如果为true，则缓存的数据始终有效，如果为false那么还要根据
timeToIdleSeconds，timeToLiveSeconds判断
        overflowToDisk - 设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上 以下
属性是可选的： timeToIdleSeconds
        - 当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时，这
些数据便会删除，默认值是0,也就是可闲置时间无穷大
        timeToLiveSeconds - 缓存element的有效生命期，默认是0.,也就是element存活时间无
穷大 diskSpoolBufferSizeMB
        这个参数设置DiskStore(磁盘缓存)的缓存区大小.默认是30MB.每个Cache都应该有自己的一
个缓冲区. diskPersistent
        - 在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false。
diskExpiryThreadIntervalSeconds
```

```
        - 磁盘缓存的清理线程运行间隔，默认是120秒。每个120s，相应的线程会进行一次EhCache
中数据的清理工作 memoryStoreEvictionPolicy
        - 当内存缓存达到最大，有新的element加入的时候， 移除缓存中element的策略。默认是
LRU（最近最少使用），可选的有LFU（最不常使用）和FIFO（先进先出） -->

    <defaultCache maxElementsInMemory="1000"
       maxElementsOnDisk="10000000" eternal="false" overflowToDisk="false"
       timeToIdleSeconds="120" timeToLiveSeconds="120"
       diskExpiryThreadIntervalSeconds="120" memoryStoreEvictionPolicy="LRU">
    </defaultCache>
</ehcache>
```

3.9、应用场景

使用场景：对于访问响应速度要求高，但是实时性不高的查询，可以采用二级缓存技术。

注意：在使用二级缓存的时候，要设置一下刷新间隔（cache标签中有一个flashInterval属性）来定时刷新二级缓存，这个刷新间隔根据具体需求来设置，比如设置30分钟、60分钟等，单位为毫秒。

3.10、局限性

Mybatis二级缓存对细粒度的数据级别的缓存实现不好。

场景：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次查询都是最新的商品信息，此时如果使用二级缓存，就无法实现当一个商品发生变化只刷新该商品的缓存信息而不刷新其他商品缓存信息，因为二级缓存是mapper级别的，当一个商品的信息发送更新，所有的商品信息缓存数据都会清空。

解决此类问题，需要在业务层根据需要对数据有针对性的缓存。比如可以对经常变化的 数据操作单独放到另一个namespace的mapper中。

课前默写

1. 掌握MyBatis框架原理
2. 掌握MyBatis的基本数据交互方式
3. 掌握MyBatis基础配置
4. 掌握MyBatis环境搭建
5. 掌握MyBatis的基础CRUD操作

作业

1. 使用MySchool数据库
2. 建立各实体间的关系
3. 实现各表的CRUD操作

面试题

1. 简述MyBatis的动态SQL及其作用
2. 简述MyBatis的关联映射与Hibernate中关联映射的异同
3. 简述MyBatis的延迟加载
4. 简述MyBatis缓存的作用及其优缺点