Mybatis任务三：加载策略及注解开发

一 MyBatis加载策略

1.1 什么是延迟加载?

问题

通过前面的学习，我们已经掌握了Mybatis中一对一，一对多，多对多关系的配置及实现，可以实现对象的关联查询。实际开发过程中很多时候我们并不需要总是在加载用户信息时就一定要加载他的订单信息。此时就是我们所说的延迟加载。

举个栗子

```
* 在一对多中，当我们有一个用户，它有个100个订单
    在查询用户的时候，要不要把关联的订单查出来？
    在查询订单的时候，要不要把关联的用户查出来？

* 回答
    在查询用户时，用户下的订单应该是，什么时候用，什么时候查询。
    在查询订单时，订单所属的用户信息应该是随着订单一起查询出来。
```

延迟加载

就是在需要用到数据时才进行加载，不需要用到数据时就不加载数据。延迟加载也称懒加载。

```
* 优点：
    先从单表查询，需要时再从关联表去关联查询，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。

* 缺点：
    因为只有当需要用到数据时，才会进行数据库查询，这样在大批量数据查询时，因为查询工作也要消耗时间，所以可能造成用户等待时间变长，造成用户体验下降。

* 在多表中：
    一对多，多对多：通常情况下采用延迟加载
    一对一（多对一）：通常情况下采用立即加载

* 注意：
    延迟加载是基于嵌套查询来实现的
```

1.2 实现

1.2.1 局部延迟加载

在association和collection标签中都有一个fetchType属性，通过修改它的值，可以修改局部的加载策略。

```
<!-- 开启一对多 延迟加载 -->
<resultMap id="userMap" type="user">
```

```
    <id column="id" property="id"></id>
    <result column="username" property="username"></result>
    <result column="password" property="password"></result>
    <result column="birthday" property="birthday"></result>
    <!--
        fetchType="lazy" 懒加载策略
        fetchType="eager" 立即加载策略
    -->
    <collection property="orderList" ofType="order" column="id"
                select="com.lagou.dao.OrderMapper.findByUid" fetchType="lazy">
    </collection>
</resultMap>

<select id="findAll" resultMap="userMap">
    SELECT * FROM `user`
</select>
```

1.2.2 设置触发延迟加载的方法

大家在配置了延迟加载策略后，发现即使没有调用关联对象的任何方法，但是在你调用当前对象的equals、clone、hashCode、toString方法时也会触发关联对象的查询。

我们可以在配置文件中使用lazyLoadTriggerMethods配置项覆盖掉上面四个方法。

```
<settings>
    <!--所有方法都会延迟加载-->
    <setting name="lazyLoadTriggerMethods" value="toString()"/>
</settings>
```

1.2.3 全局延迟加载

在Mybatis的核心配置文件中可以使用setting标签修改全局的加载策略。

```
<settings>
    <!--开启全局延迟加载功能-->
        <setting name="lazyLoadingEnabled" value="true"/>
</settings>
```

注意

局部的加载策略优先级高于全局的加载策略。

```
<!-- 关闭一对一 延迟加载 -->
<resultMap id="orderMap" type="order">
    <id column="id" property="id"></id>
    <result column="ordertime" property="ordertime"></result>
    <result column="total" property="total"></result>
    <!--
        fetchType="lazy" 懒加载策略
```

```
        fetchType="eager" 立即加载策略
    -->
    <association property="user" column="uid" javaType="user"
                 select="com.lagou.dao.UserMapper.findById" fetchType="eager">
</association>
</resultMap>

<select id="findAll" resultMap="orderMap">
    SELECT * from orders
</select>
```

二 MyBatis缓存

2.1 为什么使用缓存?

当用户频繁查询某些固定的数据时,第一次将这些数据从数据库中查询出来,保存在缓存中。当用户再次查询这些数据时,不用再通过数据库查询,而是去缓存里面查询。减少网络连接和数据库查询带来的损耗,从而提高我们的查询效率,减少高并发访问带来的系统性能问题。

一句话概括：经常查询一些不经常发生变化的数据，使用缓存来提高查询效率。

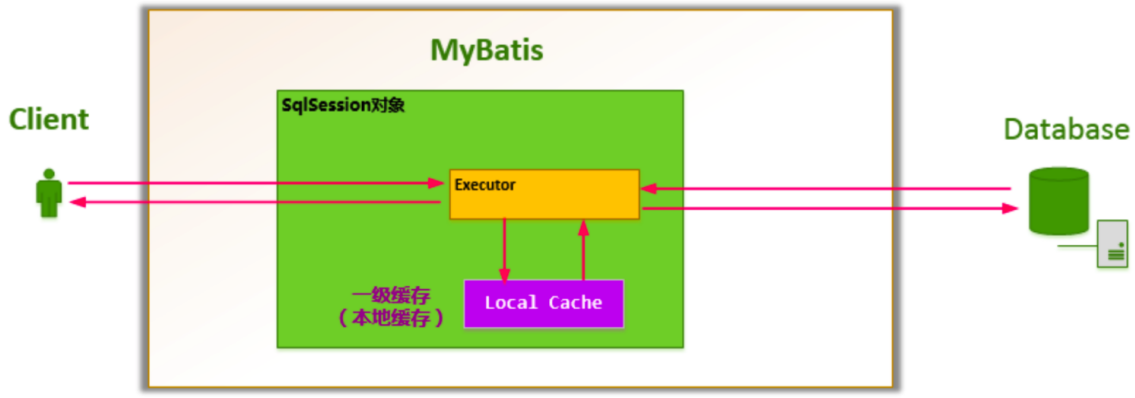
像大多数的持久化框架一样，Mybatis也提供了缓存策略，通过缓存策略来减少数据库的查询次数，从而提高性能。 Mybatis中缓存分为一级缓存，二级缓存。

2.2 一级缓存

2.2.1 介绍

一级缓存是SqlSession级别的缓存，是默认开启的

所以在参数和SQL完全一样的情况下，我们使用同一个SqlSession对象调用一个Mapper方法，往往只执行一次SQL，因为使用SelSession第一次查询后，MyBatis会将其放在缓存中，以后再查询的时候，如果没有声明需要刷新，并且缓存没有超时的情况下，SqlSession都会取出当前缓存的数据，而不会再次发送SQL到数据库。



MyBatis一级缓存简单示意图

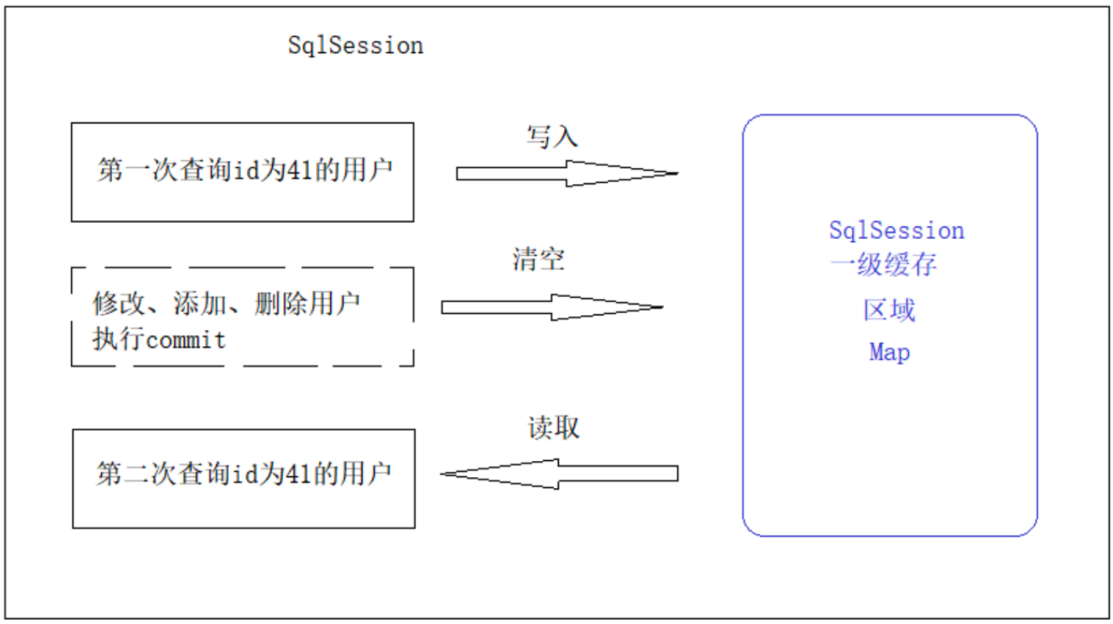
4.2.2 验证

```java
@Test
public void testOneCache() throws Exception {
    SqlSession sqlSession = MyBatisUtils.openSession();
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    User user1 = userMapper.findById(1);
    System.out.println("第一次查询的用户：" + user1);

    User user2 = userMapper.findById(1);
    System.out.println("第二次查询的用户：" + user2);

    sqlSession.close();
}
```

我们可以发现，虽然在上面的代码中我们查询了两次，但最后只执行了一次数据库操作，这就是Mybatis提供给我们的一级缓存在起作用了。因为一级缓存的存在，导致第二次查询id为1的记录时，并没有发出sql语句从数据库中查询数据，而是从一级缓存中查询。

4.2.3 分析

一级缓存是SqlSession范围的缓存，执行SqlSession的C（增加）U（更新）D（删除）操作，或者调用clearCache()、commit()、close()方法，都会清空缓存。



1. 第一次发起查询用户id为41的用户信息，先去找缓存中是否有id为41的用户信息，如果没有，从数据库查询用户信息。

2. 得到用户信息，将用户信息存储到一级缓存中。

3. 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。

4. 第二次发起查询用户id为41的用户信息，先去找缓存中是否有id为41的用户信息，缓存中有，直接从缓存中获取用户信息。

4.2.4 清除

```java
@Test
public void testClearOneCache() throws Exception {
    SqlSession sqlSession = MybatisUtils.openSession();
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    User user1 = userMapper.findById(41);
    System.out.println("第一次查询的用户：" + user1);

    //调用sqlSession清除缓存的方法
    sqlSession.clearCache();

    User user2 = userMapper.findById(41);
    System.out.println("第二次查询的用户：" + user2);
}
```

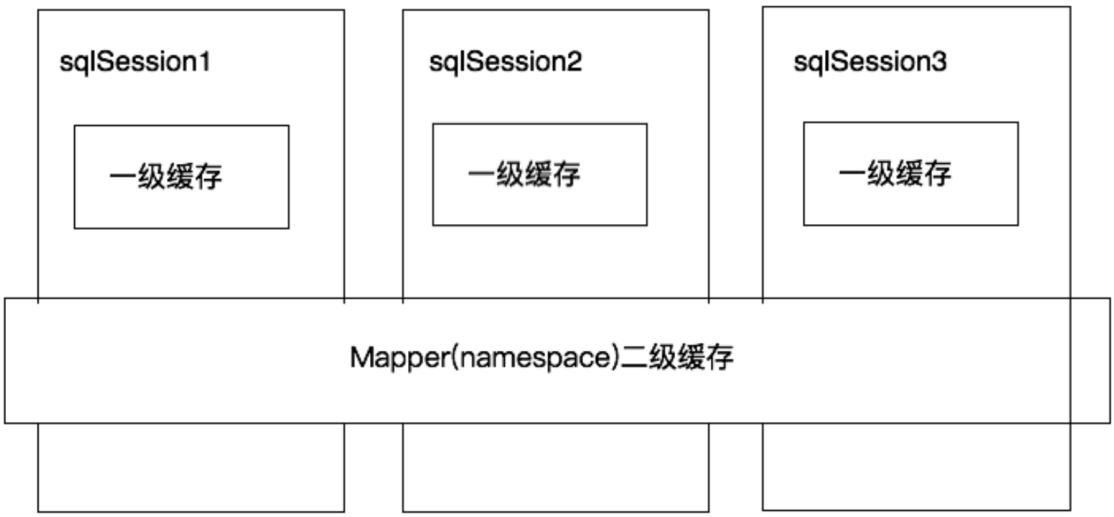
```xml
<!-- 每次查询时，都会清除缓存 -->
< select flushCache="true"></select>
```

2.3 二级缓存

2.3.1 介绍

二级缓存是namspace级别（跨sqlSession）的缓存，是默认不开启的

二级缓存的开启需要进行配置，实现二级缓存的时候，MyBatis要求返回的POJO必须是可序列化的。也就是要求实现Serializable接口，配置方法很简单，只需要在映射XML文件配置`<cache/>`就可以开启二级缓存了。



2.3.2 验证

a）配置核心配置文件

```xml
<settings>
    <!--
        因为cacheEnabled的取值默认就为true，所以这一步可以省略不配置。
        为true代表开启二级缓存；为false代表不开启二级缓存。
    -->
    <setting name="cacheEnabled" value="true"/>
</settings>
```

b）配置UserMapper.xml映射

```
<mapper namespace="com.lagou.dao.UserMapper">
    <!--当前映射文件开启二级缓存-->
    <cache></cache>
    <!--
        <select>标签中设置useCache=”true”代表当前这个statement要使用二级缓存。
        如果不使用二级缓存可以设置为false
        注意：
            针对每次查询都需要最新的数据sql，要设置成useCache="false"，禁用二级缓存。
    -->
    <select id="findById" parameterType="int" resultType="user" useCache="true"
>
        SELECT * FROM `user` where id = #{id}
    </select>
</mapper>
```

c）修改User实体

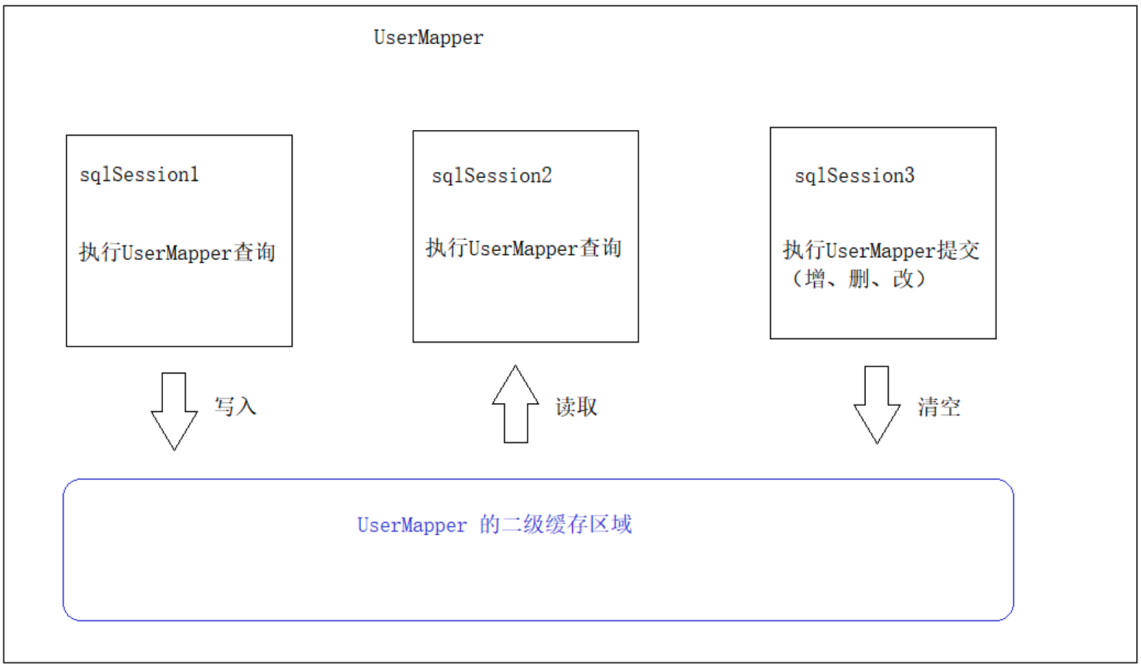
```
public class User implements Serializable {
    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    private List<Role> roleList;
    private List<Order> orderList;
}
```

d）测试结果

```
@Test
public void testTwoCache() throws Exception {
    SqlSession sqlSession = MyBatisUtils.openSession();
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    User user = userMapper.findById(41);
    System.out.println("第一次查询的用户：" + user);
    sqlSession.close();

    SqlSession sqlSession1 = MyBatisUtils.openSession();
    UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
    User user1 = userMapper1.findById(41);
    System.out.println("第二次查询的用户："+user1);
    sqlSession1.close();
}
```

2.3.3 分析

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



1. 映射语句文件中的所有select语句将会被缓存。

2. 映射语句文件中的所有insert、update和delete语句会刷新缓存。

2.3.4 注意问题（脏读）

mybatis的二级缓存因为是namespace级别，所以在进行多表查询时会产生脏读问题

2.4 小结

1. mybatis的缓存，都不需要我们手动存储和获取数据。mybatis自动维护的。

2. mybatis开启了二级缓存后，那么查询顺序：二级缓存-->一级缓存-->数据库

2. 注意：mybatis的二级缓存会存在脏读问题，需要使用第三方的缓存技术解决问题。

三 MyBatis注解

3.1 MyBatis常用注解

这几年来注解开发越来越流行，Mybatis也可以使用注解开发方式，这样我们就可以减少编写Mapper映射文件了。我们先围绕一些基本的CRUD来学习，再学习复杂映射多表操作。

* @Insert：实现新增，代替了<insert></insert>

```
* @Delete：实现删除，代替了<delete></delete>

* @Update：实现更新，代替了<update></update>

* @Select：实现查询，代替了<select></select>


* @Result：实现结果集封装，代替了<result></result>

* @Results：可以与@Result 一起使用，封装多个结果集，代替了<resultMap></resultMap>

* @One：实现一对一结果集封装，代替了<association></association>

* @Many：实现一对多结果集封装，代替了<collection></collection>
```

3.2 MyBatis注解的增删改查【重点】

3.2.1 创建UserMapper接口

```
public interface UserMapper {


    @Select("SELECT * FROM `user`")
    public List<User> findAll();

    @Insert("INSERT INTO `user`(username,birthday,sex,address) VALUES(#
{username},#{birthday},#{sex},#{address})")
    public void save(User user);

    @Update("UPDATE `user` SET username = #{username},birthday = #{birthday},sex
= #{sex},address = #{address} WHERE id = #{id}")
    public void update(User user);

    @Delete("DELETE FROM `user` where id = #{id}")
    public void delete(Integer id);
}
```

3.2.2 编写核心配置文件

```
<!--我们使用了注解替代的映射文件，所以我们只需要加载使用了注解的Mapper接口即可-->
<mappers>
    <!--扫描使用注解的Mapper类-->
    <mapper class="com.lagou.mapper.UserMapper"></mapper>
</mappers>
```

```
<!--或者指定扫描包含映射关系的接口所在的包也可以-->
<mappers>
    <!--扫描使用注解的Mapper类所在的包-->
    <package name="com.lagou.mapper"></package>
</mappers>
```

3.2.3 测试代码

```
public class TestUser extends TestBaseMapper {

    // 查询
    @Test
    public void testFindAll() throws Exception {
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        List<User> list = userMapper.findAll();
        for (User user : list) {
            System.out.println(user);
        }
    }

    // 添加
    @Test
    public void testSave() throws Exception {
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        User user = new User();
        user.setUsername("于谦");
        user.setBirthday(new Date());
        user.setSex("男");
        user.setAddress("北京德云社");
        userMapper.save(user);

    }

    // 更新
    @Test
    public void testUpdate() throws Exception {
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        User user = new User();
        user.setId(49);
        user.setUsername("郭德纲");
        user.setBirthday(new Date());
        user.setSex("男");
        user.setAddress("北京德云社");
        userMapper.update(user);
    }

    // 删除
    @Test
    public void testDelete() throws Exception {
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        userMapper.delete(49);
    }
}
```

3.3 使用注解实现复杂映射开发

之前我们在映射文件中通过配置`<resultMap>`、`<association>`、`<collection>`来实现复杂关系映射。

使用注解开发后，我们可以使用`@Results、@Result，@One、@Many`注解组合完成复杂关系的配置。

注解	说明
@Results	代替的是标签<resultMap>该注解中可以使用单个@Result注解，也可以使用@Result集合。使用格式：@Results（{@Result（），@Result（）}）或@Results（@Result（））
@Resut	代替了<id>标签和<result>标签 @Result中属性介绍： column：数据库的列名 property：需要装配的属性名 one：需要使用的@One 注解（@Result（one=@One）（）） many：需要使用的@Many 注解（@Result（many=@many）（））

注解	说明
@One （一对一）	代替了<assocation> 标签，是多表查询的关键，在注解中用来指定子查询返回单一对象。 @One注解属性介绍： select: 指定用来多表查询的 sqlmapper 使用格式：@Result(column=" ",property="",one=@One(select=""))
@Many （一对多）	代替了<collection>标签, 是是多表查询的关键，在注解中用来指定子查询返回对象集合。 使用格式：@Result(property="",column="",many=@Many(select=""))

3.4 一对一查询

3.4.1 介绍

需求：查询一个订单，与此同时查询出该订单所属的用户

一对一查询语句

```
SELECT * FROM orders;
SELECT * FROM `user` WHERE id = #{订单的uid};
```

3.4.2 代码实现

a）OrderMapper接口

```
public interface OrderMapper {

    @Select("SELECT * FROM orders")
    @Results({
            @Result(id = true, column = "id", property = "id"),
            @Result(column = "ordertime", property = "ordertime"),
            @Result(column = "money", property = "money"),
            @Result(property = "user", javaType = User.class,
                    column = "uid", one = @One(select =
"com.lagou.mapper.UserMapper.findById", fetchType = FetchType.EAGER))
    })
    public List<Order> findAllWithUser();

}
```

b）UserMapper接口

```
public interface UserMapper {

    @Select("SELECT * FROM `user` WHERE id = #{id}")
    public User findById(Integer id);
}
```

c）测试代码

```
@Test
public void testOrderWithUser() throws Exception {
    OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.class);

    List<Order> list = orderMapper.findAllWithUser();

    for (Order order : list) {
        System.out.println(order);
    }
}
```

3.5 一对多查询

3.5.1 介绍

需求：查询一个用户，与此同时查询出该用户具有的订单

一对多查询语句

```
SELECT * FROM `user`;
SELECT * FROM orders where uid = #{用户id};
```

3.5.2 代码实现

a）UserMapper接口

```
public interface UserMapper {

    @Select("SELECT * FROM `user`")
    @Results({
        @Result(id = true, column = "id", property = "id"),
        @Result(column = "brithday", property = "brithday"),
        @Result(column = "sex", property = "sex"),
        @Result(column = "address", property = "address"),
        @Result(property = "orderList", javaType = List.class,
                column = "id" ,
                many = @Many(select = "com.lagou.mapper.OrderMapper.findByUid"))
    })
    public List<User> findAllWithOrder();
}
```

b）OrderMapper接口

```
public interface OrderMapper {

    @Select("SELECT * FROM orders WHERE uid = #{uid}")
    public List<Order> findByUid(Integer uid);
}
```

c）测试代码

```
@Test
public void testUserWithOrder() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    List<User> list = userMapper.findAllWithOrder();

    for (User user : list) {
        System.out.println(user);
    }
}
```

3.6 多对多查询

3.6.1 介绍

需求：查询所有用户，同时查询出该用户的所有角色

多对多查询语句

```
SELECT * FROM `user`;
SELECT * FROM role r INNER JOIN user_role ur ON r.`id` = ur.`rid`
    WHERE ur.`uid` = #{用户id};
```

3.6.2 代码实现

a）UserMapper接口

```
public interface UserMapper {

    @Select("SELECT * FROM `user`")
    @Results({
        @Result(id = true, column = "id", property = "id"),
        @Result(column = "brithday", property = "brithday"),
        @Result(column = "sex", property = "sex"),
        @Result(column = "address", property = "address"),
        @Result(property = "roleList", javaType = List.class,
                column = "id" ,
                many = @Many(select = "com.lagou.mapper.RoleMapper.findByUid"))
    })
    public List<User> findAllWithRole();
}
```

b）RoleMapper接口

```
public interface RoleMapper {

    @Select("SELECT * FROM role r INNER JOIN user_role ur ON r.`id` = ur.`rid`
WHERE ur.`uid` = #{uid}")
    public List<Role> findByUid(Integer uid);
}
```

c）测试代码

```
@Test
public void testUserWithRole() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    List<User> list = userMapper.findAllWithRole();

    for (User user : list) {
        System.out.println(user);
    }
}
```

3.7 基于注解的二级缓存

5.7.1 配置SqlMapConfig.xml文件开启二级缓存的支持

```
<settings>
    <!--
        因为cacheEnabled的取值默认就为true，所以这一步可以省略不配置。
        为true代表开启二级缓存；为false代表不开启二级缓存。
    -->
    <setting name="cacheEnabled" value="true"/>
</settings>
```

3.7.2 在Mapper接口中使用注解配置二级缓存

```
@CacheNamespace
public interface UserMapper {...}
```

3.8 注解延迟加载

不管是一对一还是一对多， 在注解配置中都有fetchType的属性

```
* fetchType = FetchType.LAZY     表示懒加载

* fetchType = FetchType.EAGER 表示立即加载

* fetchType = FetchType.DEFAULT 表示使用全局配置
```

3.9 小结

```
* 注解开发和xml配置优劣分析
    1.注解开发和xml配置相比，从开发效率来说，注解编写更简单，效率更高。
    2.从可维护性来说，注解如果要修改，必须修改源码，会导致维护成本增加。xml维护性更强。
```