4.1 Maven

4.1.2 Maven相关概念

Maven是一个**项目管理工具**。两大核心：依赖管理和项目构建

4.1.3 Maven的依赖管理

指的是Maven对jar包的管理过程。

传统web项目中，我们需要手动下载jar包，再复制到web工程中；Maven不需要将jar包放到工程项目中，而是通过在pom.xml中添加所需jar包的坐标，项目运行时，会去一个专门存放jar的地方(Maven仓库)寻找。

创建Maven项目时，会在本地创建一个jar包仓库。首先会根据pom.xml的坐标在本地仓库中查找，若本地没有，maven仓库会自动的从互联中下载到本地仓库

坐标：公司名称（域名倒写） + 项目名 + 版本号

4.1.4 项目构建

我们开发的项目，要经过编译，测试，打包，安装，部署等一系列过程

4.1.6 Maven仓库

本地仓库

远程仓库

中央仓库：非常慢

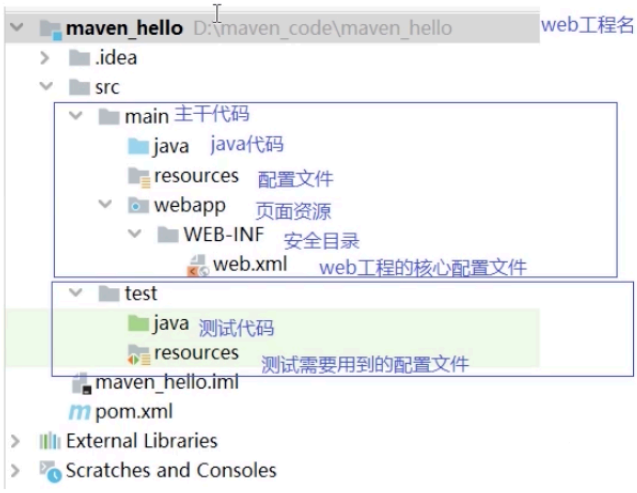
私服：架设在公司内部

第三方仓库：阿里云镜像

可以在setting.xml中配置本地仓库的位置，maven项目就可以根据setting.xml中的配置找到本地仓库。如果本地仓库没有对应jar包，就回去中央仓库中下载（联网），但非常慢。我们可以在setting.xml中进行配置，让maven项目在本地仓库没有的情况下，去访问第三方仓库。

4.1.7 Maven工程结构

1. 不同的开发工具创建的工程目录不统一
2. 项目上线前，删除测试代码和配置文件（误删）
3. 所有的java代码和配置文件都在src下没有进行分离



4.1.8 Maven常用命令

1. clean：maven工程的清理命令，执行clean会清楚target目录和内容（.class文件）
2. compile：将src/main/java下的文件编译成.class文件存到target下
3. test: 执行src/test/java下的单元测试类，并编译为.class文件
4. package：把java工程打成jar包，把web工程打包成war包；pom.xml中的packaging决定打包类型
5. install：maven工程的安装命令，执行install将mave工程打成jar包或war包，并发布到本地仓库。
6. deploy：maven工程部署命令，将jar或war包部署（上传）到私服中。

4.1.9 Maven生命周期

Maven对项目构造的过程分为"三套相互独立"的生命周期

1. Clean Lifecycle(清理生命周期)

   在进行真正的构建之前进行一些清理工作。命令:clean

2. Default Lifecycle(默认生命周期)

   构建的核心部分，编译，测试，打包，安装，部署等等。命令: compile test package install deploy

3. Site Lifecycle(站点生命周期)

   生成项目报告，站点，发布站点。命令: site

在同一个生命周期中的命令,执行后面的命令,前面的命令自动执行

4.1.14 依赖范围

依赖范围	对于编译有效	对于测试有效	对于运行时有效	例子
compile（默认值）	Y	Y	Y	mybatis
test		Y		junit
provided	Y	Y		servletapi
runtime		Y	Y	JDBC驱

4.2 Mybatis

4.2.3 JDBC问题分析

1. 数据库配置信息硬编码问题
2. 频繁创建/释放数据库连接
3. 存在sql语句硬编码问题
4. 手动封装结果集

4.2.4 Mybatis简介

1. 对JDBC的封装
2. 半自动框架
3. 轻量级
4. 基于ORM

4.2.5 ORM思想

object relational mapping 对象关系映射

O：对象模型，实体对象

R：关系型数据库的一张表

M：将实体对象与数据库表建立映射关系，借助XML或注解来完成映射关系

4.2.11 - 14 核心配置文件

1. enviroments：数据库环境的配置，支持多环境配置

   transactionManger配置事务的处理器：JDBC

   dataSoruce type：UNPOOLED/POOLED
2. properties：加载外部的properties文件。
3. typeAliases

   typeAliase给java自定义的类起个别名，方便使用

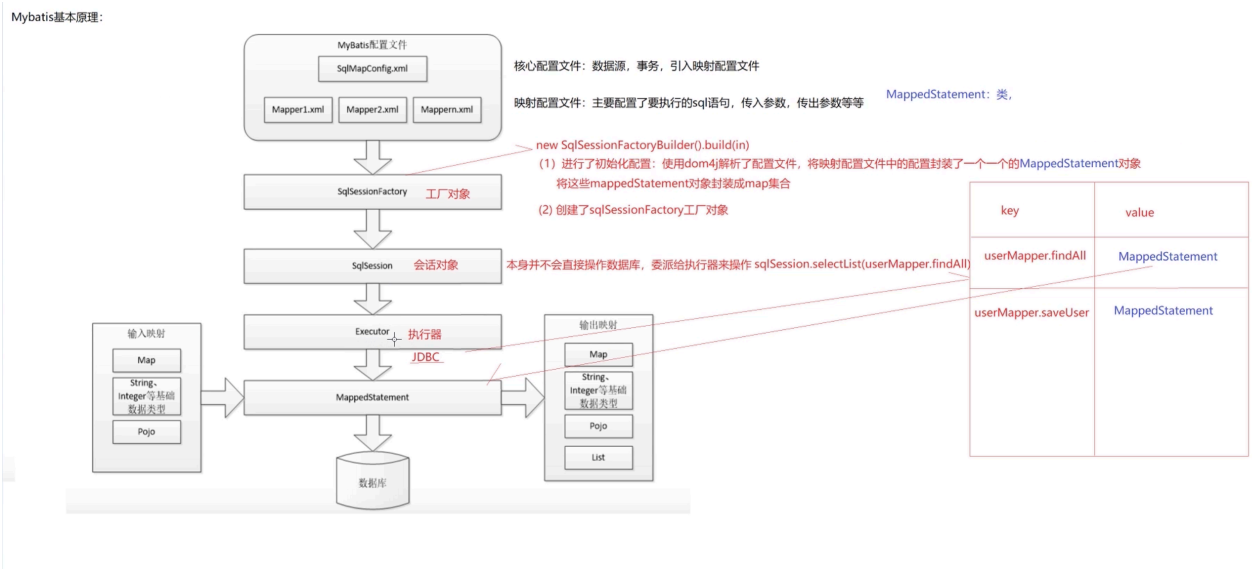
package给包内的所有类起别名，别名就是类名，不区分大小写

4. mappers：加载映射文件

4.2.15 Mybatis的API

1. 加载核心配置文件
2. 获取session工厂对象
3. 获取session对象

4.2.16 Mybatis的基本原理



4.2.17 Mybatis的dao传统使用

1. 编写interface
2. 编写实现类
3. 编写映射文件

问题：

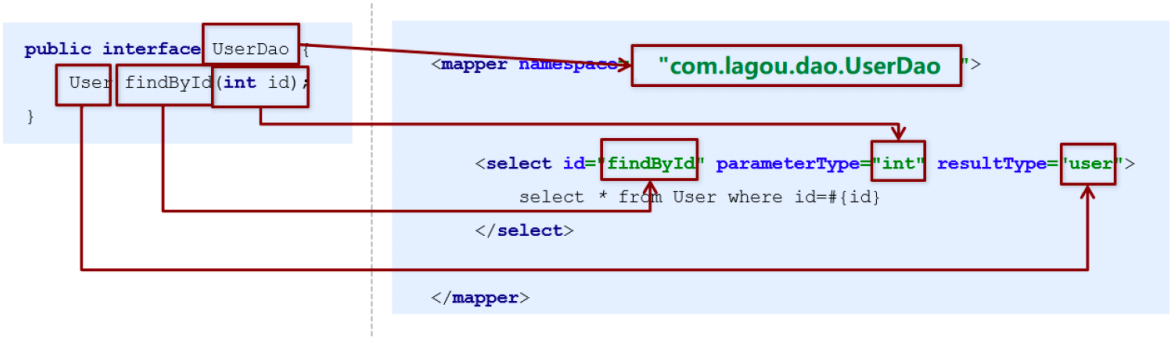
1. 代码重复
2. 硬编码问题

4.2.18 ～19 Mapper代理开发

映射文件要和接口相对应

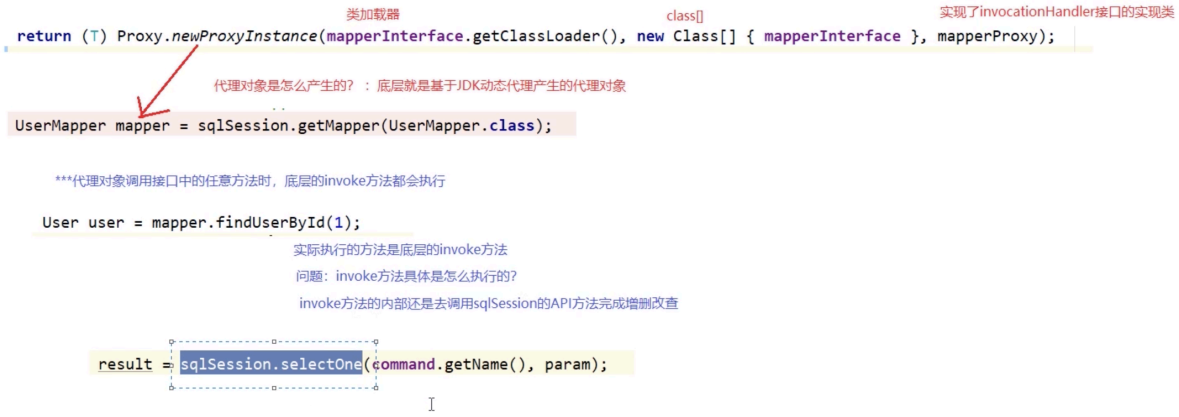
1. 接口名要和映射文件名一致
2. 接口要和映射文件处于同一层级，处于同包同名的状态
3. mapper的namespace要和接口的全限定名相同
4. id要和接口的方法名相同
5. 参数类型相同

6. 返回类型相同



4.2.20 Mapper底层原理

基于动态代理产生代理对象



4.3 Mybatis之复杂映射及深入配置

4.3.1 ResultMap

resultType：如果实体的属性名和表中字段名一致，将查询结果自动封装到实体类中

resultMap：如果实体的属性名和表中字段名不一致，可以使用resultMap手动封装到实体类中

4.3.2 ~ 3多条机查询

方式一

```
public List<User> findByIdAndUsername(int id, String username);
```

```
<select id="findByIdAndUsername" resultType="user">
  <!-- select * from user where id=#{arg0} and username=#{arg1} -->
  select * from user where id=#{param1} and username=#{param2}
</select>
```

方式二

```
public List<User> findByIdAndUsername(@Param("id") int id, @Param("username")
String username);
```

```
<select id="findByIdAndUsername" resultType="user">
  select * from user where id=#{id} and username=#{username}
</select>
```

方式三

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

```
<select id="findByIdAndUsername" resultType="user" parameterType="user">
  select * from user where id=#{id} and username=#{username}
</select>
```

4.3.4 模糊查询

1. #{}表示占位符，引用参数时会自动添加单引号。可以防止sql注入
2. #{}可以接受简单类型值或pojo属性值
3. 如果parameterType是单个简单类型值，#{}的值可以随便写

```
<select id="findByUsername" resultType="user" parameterType="String">
  select * from user where username like #{username}
</select>
```

1. ${}表示sql原样拼接，在引用参数时，需要添加引号。
2. ${}可以接受简单类型值或pojo属性值
3. 如果parameterType是简单类型值，${}里面必须是value

```
<select id="findByUsername2" resultType="user" parameterType="String">
  select * from user where username like '${value}'
</select>
```

4.3.5 ～ 6 返回主键

方式一

1. useGenerateType：声明返回主键
2. keyProperty：把返回的主键值，封装到实体中的某个属性上

```
<insert id="addUser" parameterType="user" useGeneratedKeys="true"
keyProperty="id">
  insert into user(username, birthday, sex, address) values (#{username},#
{birthday},#{sex},#{address});
</insert>
```

只适用主键子增的数据库像mySql，Oracle不支持。

方式二

order：在sql执行之前还是之后

keyColumn：指定主键对应的列名

keyProperty：把返回的主键值，封装到实体中的某个属性上

resultType：指定主键类型

```
<insert id="addUser" parameterType="user">
  <selectKey order="AFTER" keyColumn="id" keyProperty="id" resultType="int">
    select LAST_INSERT_ID()
  </selectKey>
  insert into user(username, birthday, sex, address) values (#{username},#
{birthday},#{sex},#{address})
</insert>
```

4.3.7 动态SQL-IF

需求：动态查询where后的条件

where关键字相当于 where1 = 1，但如果没有条件的话，不会拼接上where关键字，并去掉第一个and

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

4.3.8 动态SQL-SET

需求：动态更新

set在更新的时候自动加上set关键字，还会去掉最后一条语句的逗号

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

4.3.9 动态配置-FOREACH

where：

collection：代表要遍历的集合元素，通常是collection或者list

open：语句开始的地方

close：语句结束的地方

item：代表遍历中的每一个元素

separator：分隔符

```
<select id="findByList" parameterType="list" resultType="user">
  select * from user
  <where>
    <foreach collection="collection" open="id in (" close=")" item="id"
separator=",">
      #{id}
    </foreach>
  </where>
</select>
```

4.3.10 sql语句抽取

把重复的sql提取出来

```
<sql id="selectUser">
  select * from user
</sql>

<select id="findByList" parameterType="list" resultType="user">
  <include refid="selectUser"></include>
  <where>
    <foreach collection="collection" open="id in (" close=")" item="id"
separator=",">
      #{id}
    </foreach>
  </where>
</select>
```

总结：

```
<select>：查询
<insert>：插入
<update>：修改
<delete>：删除
<selectKey>：返回主键
<where>：where条件
<if>：if判断
<foreach>：for循环
<set>：set设置
<sql>：sql片段抽取
```

4.3.11 pageHelper

设置分页

```
@Test
public void test8() throws IOException {
  InputStream resourceAsStream =
Resources.getResourceAsStream("sqlMapConfig.xml");
  SqlSessionFactory build = new
SqlSessionFactoryBuilder().build(resourceAsStream);
  SqlSession sqlSession = build.openSession();
  UserMapperInterface mapper = sqlSession.getMapper(UserMapperInterface.class);
  //参数1:当前页
  //参数2:每页显示的条数
  PageHelper.startPage(1,3);
  List<User> allUsers = mapper.findAllUsers();
  for ( User user: allUsers) {
      System.out.println(user);
  }
  sqlSession.close();
}
```

```
//获得分页的信息
PageInfo pageInfo = new PageInfo<User>(allUsers);
```

4.3.13 多表查询-一对一（多对一）

封装时，将字段名和属性名进行匹配，然后调用set方法。

多表查询时，要使用resultMap进行手动配置，使用association标签对从表中的外键进行设置

4.3.14 多表查询-一对多

假如字段名的值为null，那么不能使用这个字段名作为column的值

使用collection标签对多的一方进行配置

4.3.15 多表查询-多对多

多对多有一张中间表

总结：

* 多对一（一对一）配置：使用+做配置

* 一对多配置：使用+做配置

* 多对多配置：使用+做配置

* 多对多的配置跟一对多很相似，难度在于SQL语句的编写。

4.3.17 嵌套查询_一对一

4.3.18 嵌套查询_一对多

4.3.19 嵌套查询_多对多

总结：

一对一配置：使用+做配置，通过column条件，执行select查询

一对多配置：使用+做配置，通过column条件，执行select查询

多对多配置：使用+做配置，通过column条件，执行select查询

优点：简化多表查询操作

缺点：执行多次sql语句，浪费数据库性能

4.4 加载策略及注解开发

4.4.1 延迟加载策略概念

立即加载：加载对象时，把关联的信息也都加载出来

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。**延迟加载是基于嵌套查询来实现的**

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。

注意：

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

4.4.2 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。

lazy：延迟加载/懒加载

eager：立即加载

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。

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

4.4.3 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>
```

局部的优先级高于全局

4.4.4 Mybatis缓存概念

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。

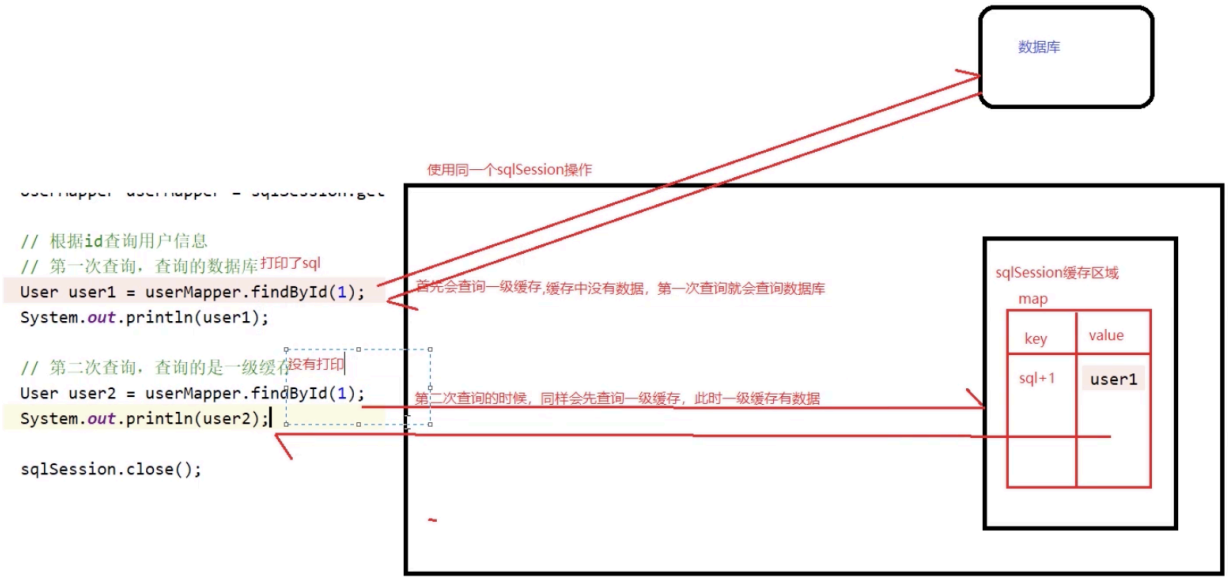
Mybatis中缓存分为一级缓存和二级缓存

4.4.5 Mybatis的一级缓存

一级缓存是SqlSession级别的，是默认开启的。

什么是SqlSession级别：

在**同一个SqlSession中**，执行**相同**的查询SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行SQL时两次查询中间发生了增删改操作，则SqlSession的缓存清空。



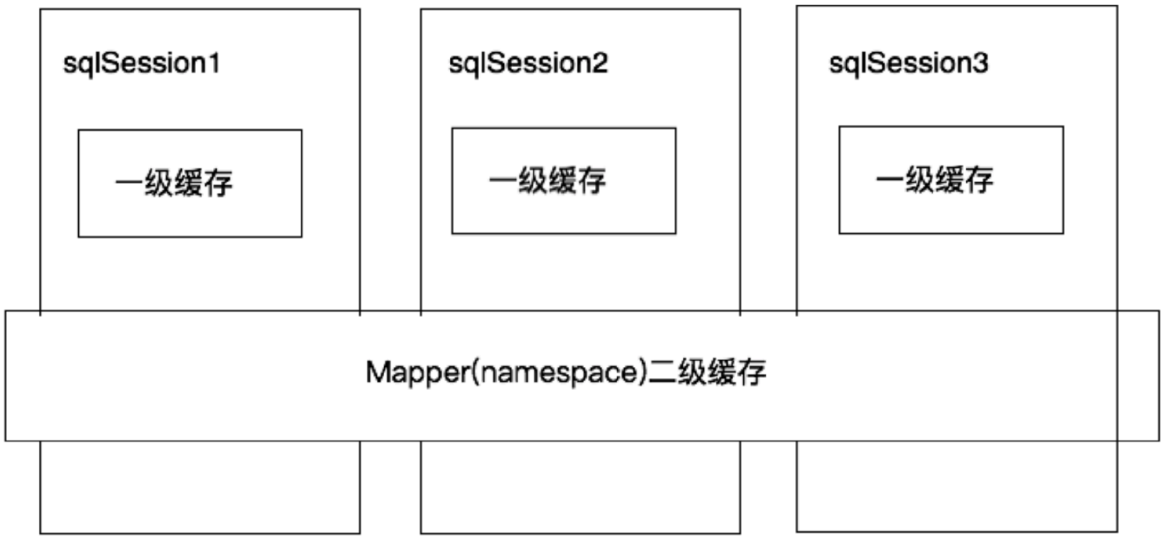
一级缓存是SqlSession范围的缓存，执行SqlSession的C（增加）U（更新）D（删除）操作，或者调用clearCache()、commit()、close()方法，都会清空缓存。

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

4.4.6 Mybatis的二级缓存

二级缓存是namespace/mapper(跨SqlSession)级别的，是默认不开启的

二级缓存的开启需要进行配置，配置方法很简单，只需要在映射XML文件配置 就可以开启二级缓存了。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。



```
<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>
```

```
<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>
```

```
//返回的POJO必须是可序列化的，也就是要求实现Serializable接口
```

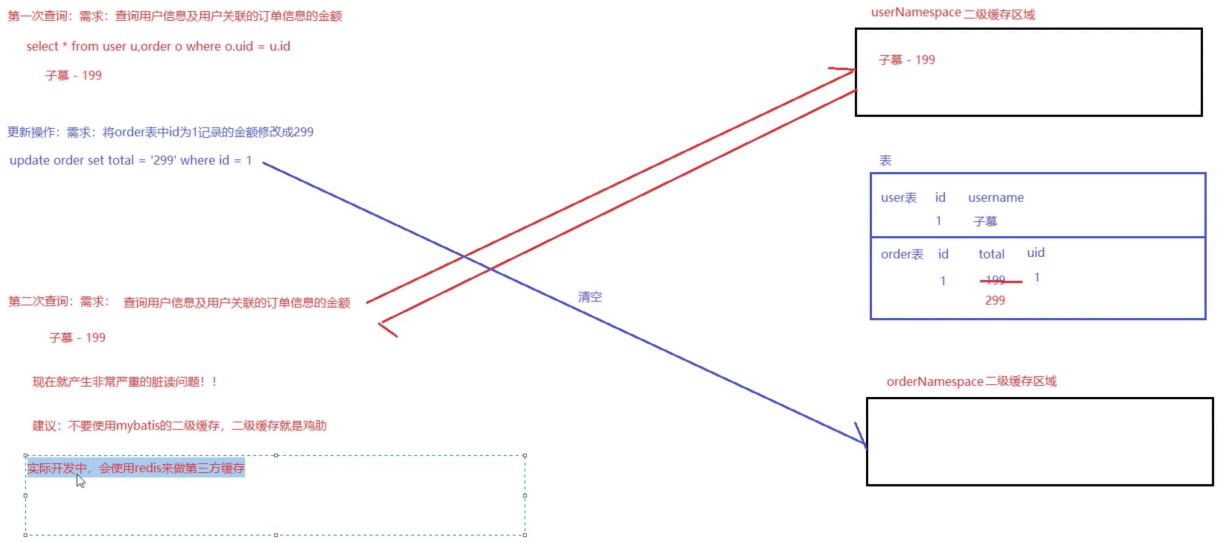
```
@Test
public void testTwoCache() throws Exception {
  SqlSession sqlSession = MyBatisUtils.openSession();
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
  User user = userMapper.findById(1);
  System.out.println("第一次查询的用户：" + user);
  sqlSession.close();

  SqlSession sqlSession1 = MyBatisUtils.openSession();
  UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);
  User user1 = userMapper1.findById(1);
  System.out.println("第二次查询的用户："+user1);
  sqlSession1.close();
}
```

注意⚠️：只有执行sqlSession.close()或者sqlSession.commit()那么一级缓存的内容才会刷新到二级缓存中

4.4.7 Mybatis二级缓存分析及脏读

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



注意⚠️：

1. 不要使用mybatis的二级缓存
2. 实际开放中，使用第三方redis缓存

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

4.4.8 ～ 11 Mybatis注解开发

* @Insert：实现新增，代替了

* @Delete：实现删除，代替了

* @Update：实现更新，代替了

* @Select：实现查询，代替了

* @Result：实现结果集封装，代替了

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

* @One：实现一对一结果集封装，代替了

* @Many：实现一对多结果集封装，代替了

@Before - @Test - @After

注解开发不需要xml文件，把逻辑写在对应的interface里

4.4.12 基于注解二级缓存

@CacheNamespace

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。单表开发

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。多表，复杂映射开发