* 学习目标

 * 能够掌握MyBatis的动态Sql

 * 能够掌握MyBatis调用存储过程

 * 能够掌握MyBatis的缓存

 * 能够掌握MyBatis一些细节

 * 能够理解MyBatis的优点

---

 * 回顾

   * 阅读MyBatis的源码

     * SqlSessionFactoryBuilder--build--SqlSessionFactory（SqlMapConfig,DaoXML或者注解）

     * SqlSessionFactry.openSession--new DefaultSqlSession

     * SqlSession:selectList,getMapper

     * selectList:调用JDBC---ResultSet

     * getMapper:Proxy.newInstance(classLoader,interfaces[],InvocationHandler(MapperProxy))

       * invoke--selectList

   * 高级结果映射

     * 表的列名和类的属性名不一致

   * 一对多，多对多

     * collection

   * 懒加载

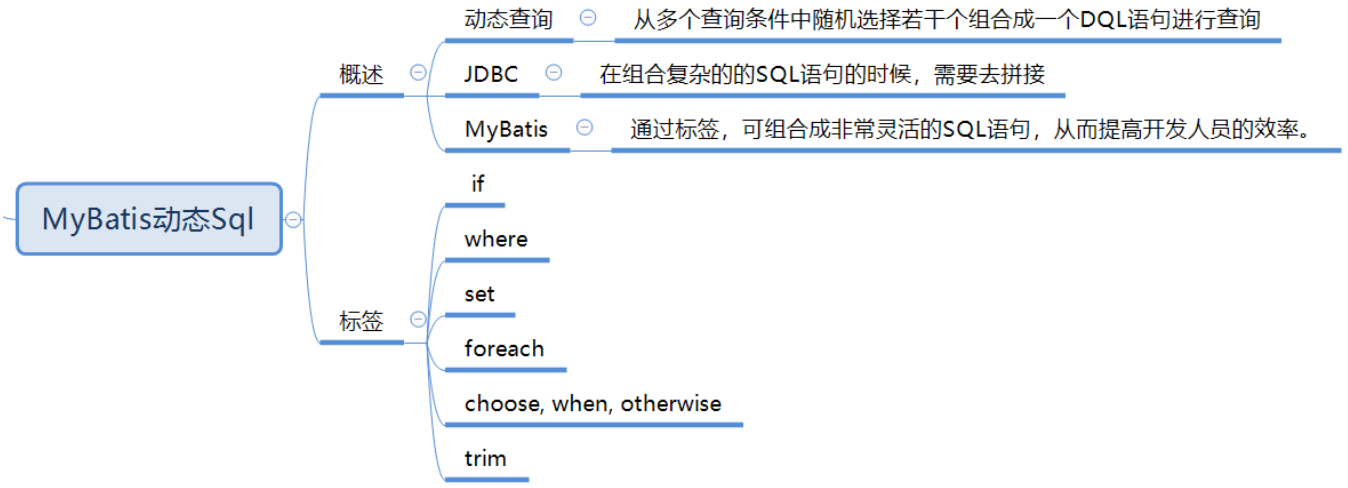
 * 能够掌握MyBatis延迟加载


```
* 全局配置:
<configuration>
<settings>
        <!--开启延迟加载-->
        <setting name="lazyLoadingEnabled" value="true"/>
        <!--关闭积极加载-->
        <setting name="aggressiveLazyLoading" value="false"/>
</settings>
</configuration>
* 局部配置:
 <collection ... fetchType="lazy"/>

* 测试案例（SQL语句分开写的，才可以做延时加载）
<?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.lg.dao.CompanyDao">
    <select id="getCompany" parameterType="int" resultMap="rst">
        SELECT * FROM company c WHERE c.code=#{code}
    </select>
    <resultMap id="rst" type="com.lg.bean.Company">
        <collection property="departments" ofType="com.lg.bean.Department" colu
    </resultMap>
    <select id="getDepartments" parameterType="int" resultType="com.lg.bean.Dep
        SELECT * FROM department d WHERE d.cid=#{cid};
    </select>
</mapper>
```

```java
public class CompanyDaoTest {
    @Test
    public void test1(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        CompanyDao companyDao = sqlSession.getMapper(CompanyDao.class);
        Company company = companyDao.getCompany(1001);
//        System.out.println(company); //打印compay已经去访问departments
        sqlSession.close();
    }
}
```

* 能够掌握MyBatis的动态Sql



* JDBC执行动态查询写法

```java
public List<User> getUsersByParams(Map<String,Object> params){
      Connection connection = null;
      PreparedStatement pst = null;
      ResultSet rs = null;
      // 用jdbc连接数据库获取数据
      try {
          // 1 加载数据库驱动
          Class.forName("com.mysql.jdbc.Driver");
          // 2 通过驱动类获取的数据库的链接
          connection = DriverManager.getConnection("jdbc:mysql://localhost:33
                  "root", "root");
          // 3 定义SQL语句
          //  // 关键是"where 1=1"，不需要再判断追加的查询条件前是否需要添加and，
          String sql = "SELECT * FROM user WHERE 1=1";
          StringBuilder sb=new StringBuilder();
          sb.append(sql);
          if(params==null || params.size()==0){
              // 没有传递参数查询所有
              pst = connection.prepareStatement(sql);
          }else{
              // 拼接参数
              for(Map.Entry<String,Object> entry:params.entrySet()){
                  sb.append(" and "+entry.getKey()+"=? ");
              }
              String sqlBuilder=sb.toString();
              System.out.println(sqlBuilder);
              pst = connection.prepareStatement(sqlBuilder);

              // 设置参数
              int index=1;
              for(Map.Entry<String,Object> entry:params.entrySet()){
                  pst.setObject(index,entry.getValue());
                  index++;
              }
          }
          rs = pst.executeQuery();
          List<User> users=new ArrayList<User>();
          while (rs.next()) {
              User user=new User();
              user.setId(rs.getInt(1));
```

```
                user.setUsername(rs.getString(2));
                user.setPsw(rs.getString(3));
                user.setSex(rs.getString(4));
                users.add(user);
            }
            return users;
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 8 释放资源
            if (rs != null) {
                try {
                    rs.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (pst != null) {
                try {
                    pst.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }

            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
        return null;
    }
* 单元测试
   @Test
  public void test8(){
        UserDaoImpl userDao=new UserDaoImpl();
        Map<String,Object> params=new HashMap<String,Object>();
```

```
        //根据传进来参数拼接不同SQL
        params.put("sex","男");
        params.put("username","xiaohei");
        List<User> users = userDao.getUsersByParams(params);
        System.out.println(users);
 }

```

* MyBatis动态查询

```
* 案例一（if）
 * Dao
 public interface UserDao5 {
    List<User> getUsersByParams(Map<String,Object> params);
 }
 * 配置文件
<?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.lg.dao.UserDao5">
    <select id="getUsersByParams" parameterType="hashMap" resultType="com.lg.be
        SELECT * FROM USER WHERE
             <if test="sex!=null">
                 sex=#{sex}
             </if>
             <if test="username!=null">
                 and username=#{username}
             </if>
    </select>
</mapper>
* 单元测试
 @Test
 public void test9(){
        SqlSession session = MyBatisUtils.getSqlSession();
        UserDao5 userDao = session.getMapper(UserDao5.class);
        Map<String,Object> params=new HashMap<String,Object>();
        params.put("sex","男");
        params.put("username","xiaohei");
```

```
        List<User> users = userDao.getUsersByParams(params);
        System.out.println(users);
 }

* 案例二：（where标签）
 * 测试之前if案例：注释掉第一个参数
  @Test
    public void test9(){
        SqlSession session = MyBatisUtils.getSqlSession();
        UserDao5 userDao = session.getMapper(UserDao5.class);
        Map<String,Object> params=new HashMap<String,Object>();
//        params.put("sex","男");
        params.put("username","xiaohei");
        List<User> users = userDao.getUsersByParams(params);
        System.out.println(users);
    }
   * 发现报错了,观察sql，发现多了一个and
    * SELECT * FROM USER WHERE and username=?
   * 解决方法：使用where标签（自动处理多余的and|or）
 * 配置文件
 <?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.lg.dao.UserDao5">
    <select id="getUsersByParams" parameterType="hashMap" resultType="com.lg.be
        SELECT * FROM USER
        <where>
            <if test="sex!=null">
                sex=#{sex}
            </if>
            <if test="username!=null">
                and username=#{username}
            </if>
        </where>
    </select>
</mapper>
* 案例三（set标签）
 * 在Dao上添加方法：
    void updateUser(Map<String,Object> params);
```

```
* 配置
<update id="updateUser" parameterType="hashMap">
        UPDATE USER SET
        <if test="username!=null">
            username=#{username},
        </if>
        <if test="psw!=null">
            psw=#{psw},
        </if>
        <if test="sex!=null">
            sex=#{sex}
        </if>
        <if test="id!=0">
            <where>
                id=#{id}
            </where>
        </if>
    </update>
* 单元测试
public void test10(){
        SqlSession session = MyBatisUtils.getSqlSession();
        UserDao5 userDao = session.getMapper(UserDao5.class);
        Map<String,Object> params=new HashMap<String,Object>();
        params.put("id",1);;
        params.put("sex","男");
        params.put("username","xiaohei123");
        params.put("psw","123");
        userDao.updateUser(params);
        session.commit();
    }
* 假如不错sex参数，进行测试，发现包报异常，观察SQL
 * UPDATE USER SET username=?, psw=?, WHERE id=? （多一个,）
* 解决方案使用set标签,会去掉多余的,
 <update id="updateUser" parameterType="hashMap">
        UPDATE USER
        <set>
            <if test="username!=null">
                username=#{username},
            </if>
            <if test="psw!=null">
```

```
                psw=#{psw},
            </if>
            <if test="sex!=null">
                sex=#{sex}
            </if>
        </set>
        <if test="id!=0">
            <where>
                id=#{id}
            </where>
        </if>
    </update>
  * 温馨提醒：where标签放在set标签外面

* 案例四（foreach标签）
 -- 通过动态部门编号数组查询员工信息
 * sql
  * SELECT * FROM employee WHERE departid IN(1,2,3)
 * 在EmployeeDao中添加方法
  List<Employee> getEmployeeByIds(int[] ids);
  List<Employee> getEmployeeByListIds(List<Integer> ids);
 * 配置文件
   <select id="getEmployeeByIds" parameterType="int" resultType="com.lg.bean.Em
        SELECT * FROM employee WHERE departid IN
        <foreach collection="array" item="id" separator="," open="(" close=")"
            ${id}
        </foreach>
    </select>
    <select id="getEmployeeByListIds" parameterType="list" resultType="com.lg.b
        SELECT * FROM employee WHERE departid IN
        <foreach collection="list" item="id" separator="," open="(" close=")" i
            ${id}
        </foreach>
    </select>
 * 单元测试
  @Test
    public void test2(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        EmployeeDao employeeDao = sqlSession.getMapper(EmployeeDao.class);
        int[] ids={1,2};
```

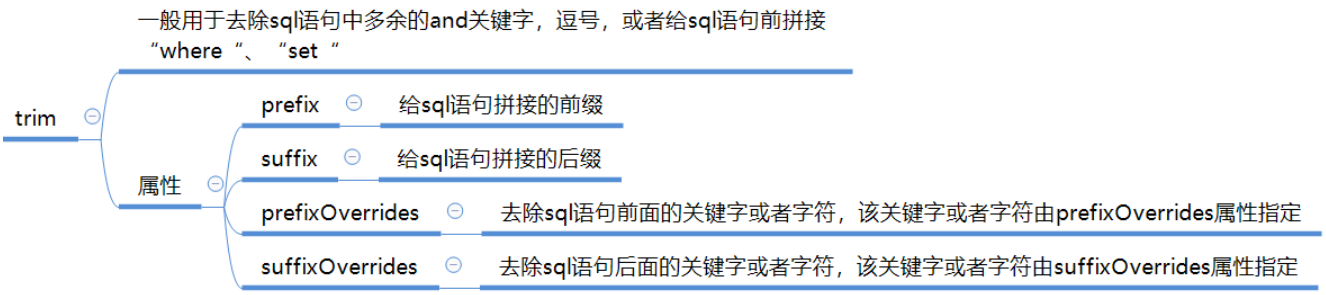
```
        List<Employee> employees = employeeDao.getEmployeeByIds(ids);
        System.out.println(employees);
        sqlSession.close();
    }

    @Test
    public void test3(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        EmployeeDao employeeDao = sqlSession.getMapper(EmployeeDao.class);
        List<Integer> ids=new ArrayList<>();
        ids.add(1);
        ids.add(2);
        List<Employee> employees = employeeDao.getEmployeeByListIds(ids);
        System.out.println(employees);
        sqlSession.close();
    }
* 案例五：(choose, when, otherwise标签)
 * 需求：
  * 当员工编号不为空的时候，通过员工编号查询
  * 当员工编号为空的时候并且员工名字不为空的时候，通过员工名字查询
  * 否则默认通过性别为男查询
 * 在EmployeeDao添加方法
   List<Employee> getEmployeeByEmpNoOrName(Employee employee);
 * 配置文件
 <select id="getEmployeeByEmpNoOrName" parameterType="com.lg.bean.Employee" res
        SELECT * FROM employee
        <where>
            <choose>
                <when test="empno!=null">
                    empno=#{empno}
                </when>
                <when test="name!=null">
                    name=#{name}
                </when>
                <otherwise>
                    sex='男'
                </otherwise>
            </choose>
        </where>
    </select>
```

```
 * 单元测试
 @Test
    public void test4(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        EmployeeDao employeeDao = sqlSession.getMapper(EmployeeDao.class);
        Employee employee=new Employee();
//        employee.setEmpno("LG001");
        employee.setName("老刘");
        List<Employee> employees = employeeDao.getEmployeeByEmpNoOrName(employe
        System.out.println(employees);
        sqlSession.close();
    }

* 案例六（trim 标签）
 * 替换where标签的
   <select id="getUsersByParams" parameterType="hashMap" resultType="com.lg.bea
        SELECT * FROM USER
        <trim prefix="where" prefixOverrides="and">
            <if test="sex!=null">
                sex=#{sex}
            </if>
            <if test="username!=null">
                and username=#{username}
            </if>
        </trim>
    </select>
 * 替换set的写法
   <update id="updateUser" parameterType="hashMap">
        UPDATE USER
        <trim prefix="set" suffixOverrides=",">
            <if test="username!=null">
                username=#{username},
            </if>
            <if test="psw!=null">
                psw=#{psw},
            </if>
            <if test="sex!=null">
                sex=#{sex}
            </if>
        </trim>
```

```
        <if test="id!=0">
            <where>
                id=#{id}
            </where>
        </if>
    </update>
```



* 能够掌握MyBatis调用存储过程

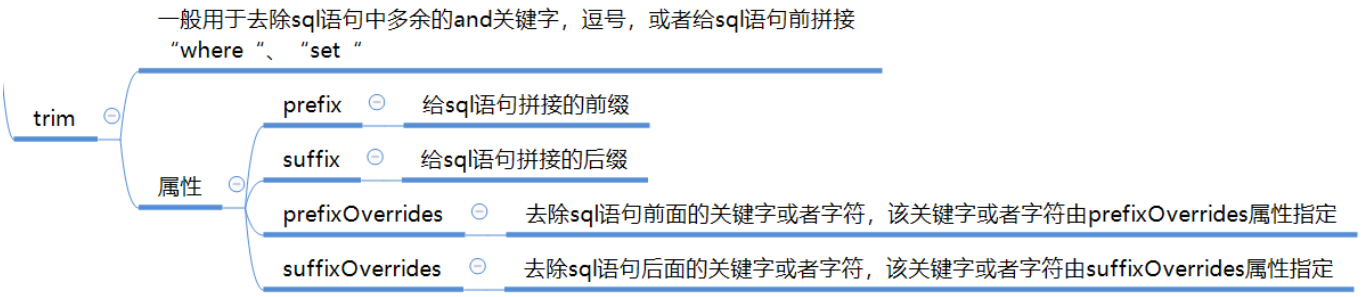
```
案例：
 * 查询得到男性或女性的数量, 如果传入的是0就女性否则是男性
DELIMITER $
CREATE PROCEDURE lg01.user_count(IN sex INT, OUT COUNT INT)
BEGIN
IF sex=0 THEN
SELECT COUNT(*) FROM lg01.user WHERE user.sex='女' INTO COUNT;
ELSE
SELECT COUNT(*) FROM lg01.user WHERE user.sex='男' INTO COUNT;
END IF;
END
$

-- 调用存储过程
DELIMITER ;
SET @COUNT = 0;
CALL lg01.user_count(0, @COUNT);
SELECT @COUNT;

* Dao
 void getUserCountBySex(Map<String,Object> params);
```

```
* 配置文件
 <select id="getUserCountBySex" parameterMap="pmap" statementType="CALLABLE">
        CALL lg01.user_count(?,?);
    </select>
 <parameterMap id="pmap" type="java.util.Map">
        <parameter property="sex" jdbcType="INTEGER" mode="IN"></parameter>
        <parameter property="count" jdbcType="INTEGER" mode="OUT"></parameter>
 </parameterMap>

* 单元测试
 @Test
    public void test11(){
        SqlSession session = MyBatisUtils.getSqlSession();
        UserDao5 userDao = session.getMapper(UserDao5.class);
        Map<String,Object> params=new HashMap<String,Object>();
        params.put("sex",1);
        params.put("count",-1);
        userDao.getUserCountBySex(params);
        System.out.println(params.get("count"));
    }
```



* 能够掌握MyBatis的缓存

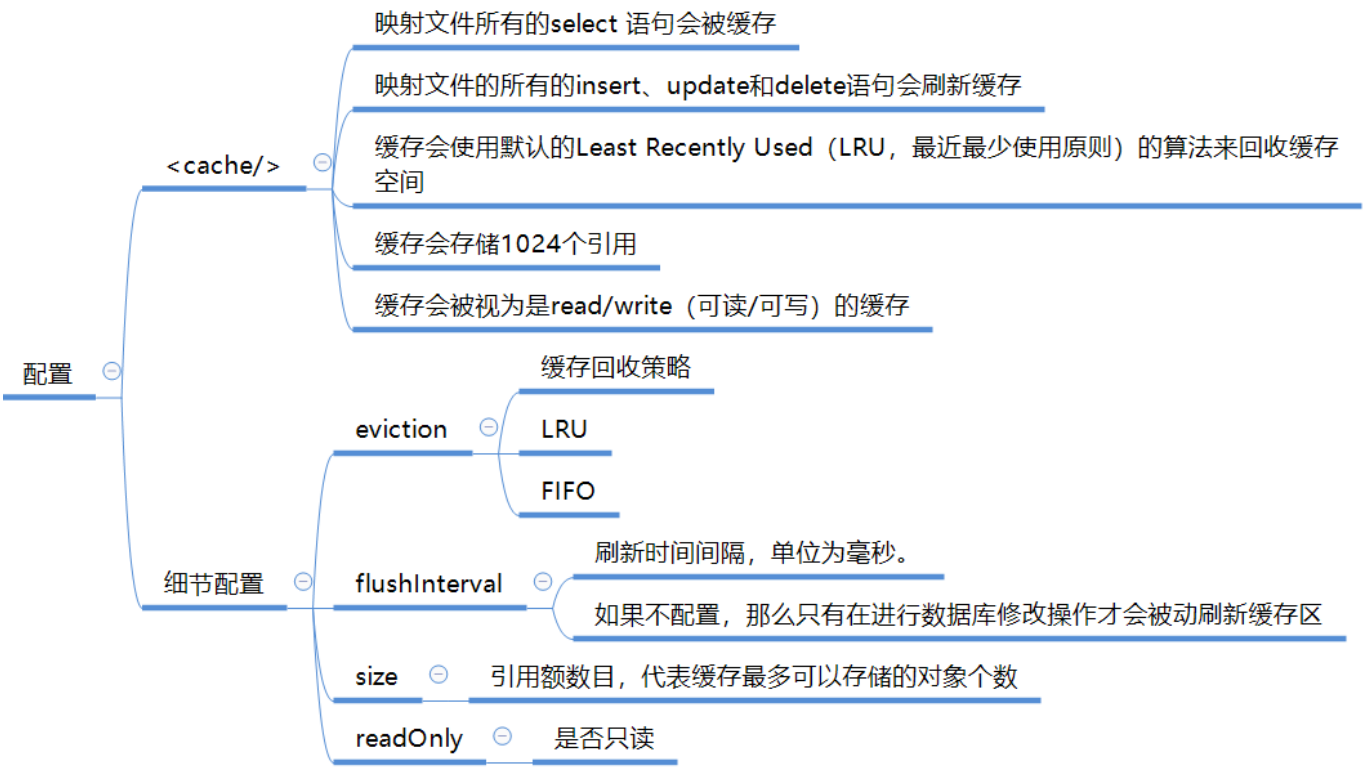
  * 上知识点前提问，为什么要用缓存?

* 一级缓存

```
* 案例一：在同一次SqlSession调用两次findUsers()
 @Test
    public void test12(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findUsers();
        users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);
    }
 * 结果：发现sql语句只执行一次
* 案例二：在不同的SqlSession调用findUsers()
  @Test
    public void test13(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);
        sqlSession = MyBatisUtils.getSqlSession();
        userDao = sqlSession.getMapper(UserDao.class);
        users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);
    }
 * 结果：发现sql语句只执行两次
```

```
* 案例三：在同一次SqlSession，在CUD的时候会清除缓存
   @Test
    public void test14(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findUsers();
        User user=new User();
        user.setId(11);
        user.setUsername("xiao123");
        userDao.update(user);
        users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);
    }
 * 结果：发现做更新的时候，会清除缓存，再次调用findUsers，会发送Sql语句
```

* 二级缓存




```
案例一：在不同的SqlSession调用findUsers()，开启了二级缓存，会缓存起来
* 缓存的实体Bean需要序列化，否则会报异常
 * public class User implements Serializable
* 配置文件
 <mapper namespace="com.lg.dao.UserDao">
    <cache/>
```

```
 </mapper>
* 单元测试
 @Test
    public void test13(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);
        sqlSession = MyBatisUtils.getSqlSession();
        userDao = sqlSession.getMapper(UserDao.class);
        users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);
    }
 * 结果：发现在不同的SqlSession调用findUsers()，只执行一次sql

* 案例二：useCache 的使用:select 有权利选择要不要被二级缓存
 <select id="findUsers" resultType="com.lg.bean.User" useCache="false">
        SELECT * FROM user;
 </select>
* 案例三：
  * 在二级缓存中，在CUD的时候会清除缓存
  @Test
    public void test15(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findUsers();
        User user=new User();
        user.setId(11);
        user.setUsername("xiao123");
        userDao.update(user);
        sqlSession.commit();
        MyBatisUtils.close(sqlSession);

        sqlSession = MyBatisUtils.getSqlSession();
        userDao = sqlSession.getMapper(UserDao.class);
        users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);
    }
   * flushCache：可以配置要不要刷新缓存
   <update id="update" parameterType="com.lg.bean.User" flushCache="false">
```

```
        UPDATE USER SET username=#{username} WHERE id=#{id};
    </update>

 * 案例四：测试缓存的size
  * <cache eviction="LRU" flushInterval="100" readOnly="true" size="1"></cache>
  * 测试
    @Test
    public void test16(){
        SqlSession sqlSession = MyBatisUtils.getSqlSession();
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);

        sqlSession = MyBatisUtils.getSqlSession();
        userDao = sqlSession.getMapper(UserDao.class);
        users = userDao.findListUserByLikeName("xiao");
        MyBatisUtils.close(sqlSession);

        sqlSession = MyBatisUtils.getSqlSession();
        userDao = sqlSession.getMapper(UserDao.class);
        users = userDao.findUsers();
        MyBatisUtils.close(sqlSession);
    }
```

* 自定义缓存

```
* 自定义缓存
public class LruCache implements Cache {
    private String id;
    private ReadWriteLock lock = new ReentrantReadWriteLock();
    private LinkedHashMap cache = new LinkedHashMap(16, 0.75f, true);

    public LruCache() {
        System.out.println("LruCache 初始化");
    }

    public LruCache(String id) {
        System.out.println("LruCache 初始化:" + id);
        this.id = id;
```

```
14      }
15
16      @Override
17      public String getId() {
18          return this.id;
19      }
20
21      @Override
22      public void putObject(Object key, Object value) {
23          System.out.println("放进缓存了....");
24          try {
25              lock.writeLock().lock();
26              cache.put(key, value);
27          } finally {
28              lock.writeLock().unlock();
29          }
30      }
31
32      @Override
33      public Object getObject(Object key) {
34          lock.readLock().lock();
35          try {
36              System.out.println("获得缓存："+cache.get(key)+"缓存的大小："+cache.s
37              return cache.get(key);
38          } finally {
39              lock.readLock().unlock();
40          }
41
42      }
43
44      @Override
45      public Object removeObject(Object key) {
46          System.out.println("移除缓存对象：" + key);
47          try {
48              lock.writeLock().lock();
49              return cache.remove(key);
50          } finally {
51              lock.writeLock().unlock();
52          }
53
```

```
    }

    @Override
    public void clear() {
        System.out.println("清除缓存！");
        cache.clear();
    }

    @Override
    public int getSize() {
        System.out.println("获取缓存大小！" + cache.size());
        return cache.size();
    }


    @Override
    public ReadWriteLock getReadWriteLock() {
        System.out.println("获取锁对象！！！");
        return lock;
    }
}
* 配置
 <cache type="com.lg.utils.LruCache"></cache>
* 观察效果
```

* 能够掌握MyBatis一些细节

  * 实体类取别名

```
* SqlMapConfig文件中
<configuration>
  <typeAliases>
        <typeAlias type="com.lg.bean.User" alias="User"></typeAlias>
    </typeAliases>
</configuration>
* 在mapper文件中使用
 *  <select id="getUsersByParams" parameterType="hashMap" resultType="User">
```

* 实体bean配置方式(包的方式)

```
<typeAliases>
<!--        <typeAlias type="com.lg.bean.User" alias="User"></typeAlias>-->
        <package name="com.lg.bean"></package>
</typeAliases>
 * 温馨提醒：配置完报名，就可以直接使用简短的类名
```

* mapper配置方式(包的方式)

```
 <mappers>
        <package name="com.lg.dao"></package>
 </mappers>
 * 温馨提醒：配置包后，就不用再配置<mapper resource|clazz>
            但是要注意配置xml的时候，需要带xml后缀，例如UserDao.xml
```

* 简化Sql编写的片段

* Sql 中可将重复的 sql 提取出来，使用时用 include 引用即可，最终达到 sql 重用的目的。

```
* 案例
<sql id="muser">
        SELECT * FROM USER
</sql>
<select id="getUsersByParams" parameterType="hashMap" resultType="User">
        <include refid="muser"/>
        <trim prefix="where" prefixOverrides="and">
            <if test="sex!=null">
                sex=#{sex}
            </if>
            <if test="username!=null">
                and username=#{username}
            </if>
        </trim>
</select>
```

* 能够理解MyBatis的优点