Mybatis-day03

第一章 多表分步查询

第一节 概念与需求

为了实现延迟加载，对Customer和Order的查询必须分开，分成两步来做，才能够实现。为此，我们需要单独查询Order，也就是需要在Mapper配置文件中，单独编写查询Order集合数据的SQL语句。

什么是延迟加载

延迟加载就是对于实体类关联的属性到**需要使用**时才查询，例如：查询到Customer的时候，不一定会使用Order的List集合数据。如果Order的集合数据始终没有使用，那么这部分数据占用的内存就浪费了。对此，我们希望不一定会被用到的数据，能够在需要使用的时候再去查询。 而我们前面学习的多表连接查询只执行了一个SQL语句，很明显不能实现延迟加载

第二节 一对一或者多对一的分步查询

1. 目标

根据订单ID查询出订单信息，并且查询出该订单所属的顾客信息，将查询到的结果集封装到Order对象中

2. 第一步: 根据订单ID查询订单信息

2.1 OrderMapper接口的抽象方法

```
/**
     * 根据orderId查询订单信息
     * @param orderId
     * @return
     */
Order selectOrderByOrderId(Integer orderId);
```

2.2 OrderMapper.xml映射配置文件

```
<select id="selectOrderByOrderId" resultMap="orderCustomerMap">
    SELECT * FROM t_order WHERE order_id=#{orderId}
</select>
```

3. 第二步: 根据顾客ID查询顾客信息

3.1 CustomerMapper接口的抽象方法

```
/**
     * 根据顾客id查询顾客信息
     * @param customerId
     * @return
     */
Customer selectCustomerByCustomerId(Integer customerId);
```

3.2 CustomerMapper.xml映射配置文件

```
<select id="selectCustomerByCustomerId" resultType="com.atguigu.pojo.Customer">
    SELECT * FROM t_customer WHERE customer_id=#{customerId}
</select>
```

4. 在OrderMapper.xml映射配置文件中配置resultMap

```
<!--
        自定义映射规则
    -->
<resultMap id="orderCustomerMap" type="com.atguigu.pojo.Order"
autoMapping="true">
    <!--
            要去使用select属性配置执行第二步，才能拿到包含顾客信息的结果集，才能赋值给
customer
            又因为调用第二步查询的时候需要将顾客id传过去，所以通过column属性传递数据
        -->
    <association property="customer" javaType="com.atguigu.pojo.Customer"

select="com.atguigu.mapper.CustomerMapper.selectCustomerByCustomerId"
                column="customer_id"
                fetchType="lazy">
    </association>
</resultMap>
```

5. 在全局配置文件中注册映射配置文件

```
<!--加载映射配置文件:指定映射配置文件的路径-->
<mappers>
    <mapper resource="mappers/OrderMapper.xml"/>
    <mapper resource="mappers/CustomerMapper.xml"/>
</mappers>
```

第三节 一对多的分步查询

1. 目标

根据客户的ID查询客户信息，并且查询出该客户的所有订单信息,将查询的结果集封装到Customer对象中

2. 第一步: 根据顾客ID查询顾客信息

2.1 CustomerMapper接口的抽象方法

```
/**
     * 根据顾客的id查询顾客信息
     * @param customerId
     * @return
     */
Customer selectCustomerByCustomerId(Integer customerId);
```

2.2 CustomerMapper.xml映射配置文件

```
<select id="selectCustomerByCustomerId" resultMap="customerOrderListMap">
    SELECT * FROM t_customer WHERE customer_id=#{customerId}
</select>
```

3. 第二步: 根据顾客ID查询订单集合

3.1 OrderMapper接口的抽象方法

```
/**
     * 根据顾客的id查询订单集合
     * @param customerId
     * @return
     */
List<Order> selectOrderListByCustomerId(Integer customerId);
```

3.2 OrderMapper.xml映射配置文件

```
<select id="selectOrderListByCustomerId" resultType="com.atguigu.pojo.Order">
    SELECT * FROM t_order WHERE customer_id=#{customerId}
</select>
```

4. 在CustomerMapper.xml映射配置文件中配置resultMap

```
<resultMap id="customerOrderListMap" type="com.atguigu.pojo.Customer"
autoMapping="true">
    <id column="customer_id" property="customerId"/>
    <!--
            调用外部查询:一对多映射
            select属性调用外部查询
            column属性往外部查询中传递值:传递的是什么本次查询中查到的字段
        -->
    <collection property="orderList" ofType="com.atguigu.pojo.Order"

 select="com.atguigu.mapper.OrderMapper.selectOrderListByCustomerId"
                column="customer_id"
                fetchType="lazy">
    </collection>
</resultMap>
```

5. 在全局配置文件中注册映射配置文件

```
<!--加载映射配置文件:指定映射配置文件的路径-->
<mappers>
    <mapper resource="mappers/CustomerMapper.xml"/>
    <mapper resource="mappers/OrderMapper.xml"/>
</mappers>
```

第四节 延迟加载

1. 概念

查询到Customer的时候，不一定会使用Order的List集合数据。如果Order的集合数据始终没有使用，那么这部分数据占用的内存就浪费了。对此，我们希望不一定会被用到的数据，能够在需要使用的时候再去查询。

例如：对Customer进行1000次查询中，其中只有15次会用到Order的集合数据，那么就在需要使用时才去查询能够大幅度节约内存空间。

延迟加载的概念：对于实体类关联的属性到**需要使用**时才查询。也叫**懒加载。**

2. 配置

2.1 配置全局懒加载

2.1.1 较低版本

在Mybatis核心配置文件中配置settings

```
<!-- 使用settings对Mybatis全局进行设置 -->
<settings>
    <!-- 开启延迟加载功能：需要配置两个配置项 -->
    <!-- 1、将lazyLoadingEnabled设置为true，开启懒加载功能 -->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!-- 2、将aggressiveLazyLoading设置为false，关闭“积极的懒加载” -->
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

2.1.2 较高版本

```
<!-- Mybatis全局配置 -->
<settings>
    <!-- 开启延迟加载功能 -->
    <setting name="lazyLoadingEnabled" value="true"/>
</settings>
```

2.2 配置局部懒加载

局部懒加载是在调用第二步查询的`association`或者`collection`标签上添加`fetch-type`属性，将属性值设置为`lazy`

局部懒加载只会对配置了局部懒加载的地方进行懒加载，未配置的地方无法进行懒加载

第二章 动态SQL

第一节 概念和作用

Mybatis框架的动态SQL技术是一种根据特定条件动态拼装SQL语句的功能，它存在的意义是为了解决拼接SQL语句字符串时的痛点问题。

> One of the most powerful features of MyBatis has always been its Dynamic SQL capabilities. If you have any experience with JDBC or any similar framework, you understand how painful it is to conditionally concatenate strings of SQL together, making sure not to forget spaces or to omit a comma at the end of a list of columns. Dynamic SQL can be downright painful to deal with.

> MyBatis的一个强大的特性之一通常是它的动态SQL能力。如果你有使用JDBC或其他相似框架的经验，你就明白条件地串联SQL字符串在一起是多么的痛苦，确保不能忘了空格或在列表的最后省略逗号。动态SQL可以彻底处理这种痛苦。

第二节 if和where标签

具体实现

if和where标签是根据不同的条件来实现拼接SQL语句,if标签可以判断在满足一定条件的情况下才拼接SQL语句；而where标签是配合if标签一起使用，可以实现在第一个条件前添加`where`关键字，并且去掉第一个条件前的`and`或者`or`关键字

1. Mapper接口

```
/**
     * 查询大于传入的empId的员工集合，如果传入的empId是null的话，就查询所有员工信息
     * @param empId
     * @return
     */
List<Employee> selectEmployeeListByEmpId(Integer empId);

/**
     * 查询大于传入的empId并且工资大于传入的empSalary的员工集合,如果传入的empId为null，则不考虑empId条件
     * 传入的empSalary为null则不考虑empSalary的条件
     * @param empId
     * @param empSalary
     * @return
     */
List<Employee> selectEmployeeListByEmpIdAndEmpSalary(@Param("id") Integer empId,@Param("salary")Double empSalary);
```

2. 映射配置文件

```
<select id="selectEmployeeListByEmpId" resultType="com.atguigu.pojo.Employee">
    select * from t_emp
    <!--
            判断:传入的customerId是否为null，如果不为null才添加:where customer_id=#{customerId}
            使用if标签进行判断,它有一个test属性表示判断条件
        -->
    <if test="empId != null">
        where emp_id=#{empId}
    </if>
</select>


<select id="selectEmployeeListByEmpIdAndEmpSalary" resultType="com.atguigu.pojo.Employee">
    select * from t_emp
    <!--
            where标签的作用:1.如果SQL语句有条件，就在第一个条件前加where关键字
                            2.如果SQL语句只有一个条件，那么就会去掉条件前的连接符(and、or)
            if标签的作用: 做判断，如果条件成立，则拼接if标签体内的SQL语句部分
        -->
```

```
    <where>
        <if test="id != null">
            emp_id>#{id}
        </if>

        <if test="salary != null">
            and emp_salary>#{salary}
        </if>
    </where>
</select>
```

第三节 set标签

1. 应用场景

实际开发时，对一个实体类对象进行更新。往往不是更新所有字段，而是更新一部分字段。此时页面上的表单往往不会给不修改的字段提供表单项。

```
<form action="" method="">

    <input type="hidden" name="userId" value="5232" />

    年　龄：<input type="text" name="userAge" /><br/>
    性　别：<input type="text" name="userGender" /><br/>
    坐　标：<input type="text" name="userPosition" /><br/>
    <!-- 用户名：<input type="text" name="userName" /><br/>   -->
    <!-- 余　额：<input type="text" name="userBalance" /><br/>-->
    <!-- 等　级：<input type="text" name="userGrade" /><br/>  -->

    <button type="submit">修改</button>

</form>
```

例如上面的表单，如果服务器端接收表单时，使用的是User这个实体类，那么userName、userBalance、userGrade接收到的数据就是null。

如果不加判断，直接用User对象去更新数据库，在Mapper配置文件中又是每一个字段都更新，那就会把userName、userBalance、userGrade设置为null值，从而造成数据库表中对应数据被破坏。

此时需要我们在Mapper配置文件中，对update语句的set子句进行定制，此时就可以使用动态SQL的set标签。

2. 代码实现

修改员工的名字和薪水，但是如果传入的参数为null，就不修改对应的字段

2.1 Mapper接口

```
/**
     * 修改员工数据，只有当员工的某个数据不为null的时候才修改该字段
     * @param employee
     */
void updateEmployee(Employee employee);
```

2.2 映射配置文件

```
<update id="updateEmployee">
    update t_emp

    <!--
            set标签的作用:1. 在修改的第一个字段之前添加SET关键字
                          2. 去掉修改的第一个字段之前的连接符(,)
        -->
    <set>
        <!--
                判断EmpName是否为null
            -->
        <if test="empName != null">
            emp_name=#{empName}
        </if>
        <if test="empSalary != null">
            ,emp_salary=#{empSalary}
        </if>
    </set>
    where emp_id=#{empId}
</update>
```

第四节 trim标签

使用trim标签控制条件部分两端是否包含某些字符

- prefix属性：指定要动态添加的前缀
- suffix属性：指定要动态添加的后缀
- prefixOverrides属性：指定要动态去掉的前缀，使用“|”分隔有可能的多个值
- suffixOverrides属性：指定要动态去掉的后缀，使用“|”分隔有可能的多个值

用`trim`标签修改原来使用`where`标签的地方

```
<select id="selectEmployeeListByEmpIdAndEmpSalary"
resultType="com.atguigu.pojo.Employee">
    select * from t_emp
    <!--
            where标签的作用:1.如果SQL语句有条件，就在第一个条件前加where关键字
                            2.如果SQL语句只有一个条件，那么就会去掉条件前的连接符(and、or)
            if标签的作用: 做判断，如果条件成立，则拼接if标签体内的SQL语句部分
        -->
    <!--<where>
            <if test="id != null">
               emp_id>#{id}
            </if>

            <if test="salary != null">
               and emp_salary>#{salary}
            </if>
        </where>-->

    <!--
            trim标签:动态添加或者去掉前后缀
                1. prefix:动态添加某个前缀
                2. sufix:动态添加后缀
                3. prefixOverrides:动态去掉某个前缀
```

```
                4. suffixOverrides:动态去掉某个后缀
        -->
    <trim prefix="WHERE" prefixOverrides="and | or" suffixOverrides="and | or">
        <!--
                使用trim标签代替where标签
            -->
        <!--
                    判断empId是否为null
                -->
        <if test="empId != null">
            emp_id>#{empId} and
        </if>
        <!--
                判断empSalary是否为null
            -->
        <if test="empSalary != null">
            emp_salary>#{empSalary} and
        </if>
    </trim>
</select>
```

第五节 choose/when/otherwise标签

在多个分支条件中，仅执行一个。

使用`choose、when、otherwise`标签代替之前的`if`标签判断

EmployeeMapper接口中的方法

```
/**
     * 根据emp_id查询员工信息，如果0<emp_id<6,那么就查询所有大于该emp_id的员工，如果emp_id是大于6，那么就查询所有小于该emp_id的员工
     * 如果是其它情况，则查询所有员工信息
     * @param empId
     * @return
     */
List<Employee> selectEmployeeList(Integer empId);
```

映射配置文件中的标签

```
<select id="selectEmployeeList" resultType="Employee">
    select * from t_emp where
    <!--
            判断empId是否大于1并且小于6,多个when的条件必须是互斥的条件,就相当于if ... elseif ...elseif ...else
        -->
    <choose>
        <when test="empId>1 and empId&lt;6">
            emp_id>#{empId}
        </when>

        <when test="empId>6">
            emp_id &lt; #{empId}
        </when>

        <otherwise>
            1=1
```

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

第六节 foreach标签

1. 用批量插入举例:

1.1 Mapper接口

```
/**
     * 进行批量插入员工信息
     * @param employeeList
     */
void insertEmployeeBatch(@Param("employeeList") List<Employee> employeeList);
```

1.2 映射配置文件

```
<!--
        批量插入员工信息
    -->
<insert id="insertEmployeeBatch">
    insert into t_emp (emp_id,emp_name,emp_salary)
    values
    <!--
            对List参数进行遍历:使用foreach标签,它有如下一些属性
                1. collection: 表示要遍历的元素是什么,要么就是你传入数据的时候给参数取一个名字;要么就是你传入的是List集合就写list
                2. item: 表示遍历出来的每一个元素
                3. separator: 表示分隔符
            每遍历出来一个元素，就要往SQL语句中拼接啥内容，就将这个内容写到foreach标签里面
        -->
    <foreach collection="employeeList" item="employee" separator=",">
        (#{employee.empId},#{employee.empName},#{employee.empSalary})
    </foreach>
</insert>
```

1.3 测试代码

```
@Test
public void testInsertEmployeeBatch(){
    List<Employee> employeeList = new ArrayList<>();
    long start = System.currentTimeMillis();
    for (int i=1;i<=5000;i++){
        employeeList.add(new Employee(null,"奥拉夫"+i,1000d));
    }

    employeeMapper.insertEmployeeBatch(employeeList);

    long end = System.currentTimeMillis();
    System.out.println(end - start);
}
```

2. 用批量查询举例:

2.1 Mapper接口

```
/**
     * 根据empId的集合查询多个员工信息,如果empIdList中没有任何元素或者empIdList为空的话，就查询所有员工信息
     * @param empIdList
     * @return
     */
List<Employee> selectEmployeeListByEmpIdList(List<Integer> empIdList);
```

2.2 映射配置文件

```
<!--
        批量查询:foreach标签
            属性列表:
                1. collection属性: 表示要遍历的对象，如果要遍历的参数使用@Param注解取名了就使用该名字，
                    如果没有取名但是要遍历的参数是List集合，就写list
                2. item属性: 表示遍历出来的元素，我们到时候要拼接SQL语句就得使用这个元素:如果遍历出来的元素是POJO对象，
                    那么我们就通过#{遍历出来的元素.POJO的属性}获取数据;如果遍历出来的元素是简单类型的数据,那么我们就使用#{遍历出来的元素}获取这个简单类型数据
                3. separator属性: 遍历出来的元素之间的分隔符
                4. open属性: 在遍历出来的第一个元素之前添加前缀
                5. close属性: 在遍历出来的最后一个元素之后添加后缀

    -->
<select id="selectEmployeeListByEmpIdList"
resultType="com.atguigu.pojo.Employee">
    select * from t_emp
    <!--
            使用foreach标签进行遍历
        -->
    <foreach collection="list" item="empId" separator="," open="where emp_id in (" close=")">
        #{empId}
    </foreach>
</select>
```

2.3 测试代码

```
@Test
public void testSelectEmployeeListByEmpIdList(){
    List<Integer> empIdList = new ArrayList<>();
    /*empIdList.add(1);
        empIdList.add(3);
        empIdList.add(4);
        empIdList.add(5);
        empIdList.add(6);
        empIdList.add(7);
        empIdList.add(16);
        empIdList.add(20);
        empIdList.add(21);*/
```

```
    List<Employee> employeeList =
employeeMapper.selectEmployeeListByEmpIdList(empIdList);

    System.out.println(employeeList);
}
```

关于foreach标签的collection属性

如果没有给接口中List类型的参数使用@Param注解指定一个具体的名字，那么在collection属性中默认可以使用collection或list来引用这个list集合。这一点可以通过异常信息看出来：

```
Parameter 'empList' not found. Available parameters are [collection, list]
```

在实际开发中，为了避免隐晦的表达造成一定的误会，建议使用@Param注解明确声明变量的名称，然后在foreach标签的collection属性中按照@Param注解指定的名称来引用传入的参数。

第七节 sql标签和include标签

1. sql标签抽取重复的SQL片段

```
<!-- 使用sql标签抽取重复出现的SQL片段 -->
<sql id="mySelectSql">
    select emp_id,emp_name,emp_age,emp_salary,emp_gender from t_emp
</sql>
```

2. include标签引用已抽取的SQL片段

```
<!--
            include标签引用SQL片段，它的refid属性就是需要引用的SQL片段的唯一标识
                1. 如果SQL片段和include在同一文件中，那么唯一标识就是SQL片段的id
                2. 如果SQL片段和include不再同一文件中，那么唯一标识就是SQL片段所在的那个映
射配置文件的namespace的值+"."+SQL片段的id
        -->
<include refid="mySelectSql"/>
```