一、嵌套查询

1.1 什么是嵌套查询

嵌套查询就是将原来多表查询中的联合查询语句拆成单个表的查询，再使用mybatis的语法嵌套在一起。

```
举个例子：
  需求：  查询账户的同时查询到用户信息
  联合查询：select * from account a,user u where a.uid = u.id
  改成嵌套查询：
    1）先查询到账户信息(这里面包含了一个跟用户相关的外键uid)
        select * from account    得到uid
    2）根据上一拿到的uid查询相关的用户信息
        select * from user where uid = #{第一步得到的uid}
    3）使用Mybatis提供的关键字，将上面两步嵌套起来
        。。。。
```

1.2 一对一的嵌套查询

1.2.1 编写接口

```
public interface AccountDao {
    public List<Account> findAccountWithUser();
}
```

1.2.2 编写AccountDao映射文件

```
<!--
        association表示要封装一个对象
        property="user"封装的是当前Account对象中的user属性对象
        javaType="user"  指定当前Account对象中的user属的类型
        association 中的column属性表示，当前要封装的对象可以通过哪一列的值间接查询到
        select="" 属性表示使用指定的条件来间接封装对象时执行的sql语句
                  这里面写的是sql语句所在的namespace+id，如果就在当前mapper映射内可以不写namespace
-->
<resultMap id="accountMap" type="Account">
    <association property="user" column="uid" javaType="user" fetchType="lazy"
                 select="com.itheima.dao.UserDao.findById">
    </association>
</resultMap>
<select id="findAccountWithUser" resultMap="accountMap">
    SELECT * FROM account
</select>
```

1.2.3 编写UserDao映射文件

```
<select id="findById" parameterType="int" resultType="user">
    select * from user where id = #{id}
</select>
```

1.3 一对多的嵌套查询【User下有多个Account】

1.3.1 编写接口

```
public List<User> findUsersWithAccounts();
```

1.3.2 编写UserDao映射文件

```
<!--
        collection 表示要封装一个集合
        property="accounts" 要封装当前User对象中的哪个属性
        ofType="Account"  指定List集合中的泛型类型
        javaType必须是当前对象是属性类型，ofType表示如果属性是集合要指定里面的泛型类型。
        注意：如果当前resultSet结果集中的某一列的值被作为条件查询了，当前的值要自己封装一次。
-->
<resultMap id="userMap" type="User">
    <id property="id" column="id"/>
    <collection property="accounts" column="id" ofType="Account"
                select="com.itheima.dao.AccountDao.findByUid">
    </collection>
</resultMap>
<select id="findUsersWithAccounts" resultMap="userMap">
    select * from user
</select>
```

1.3.3 编写AccountDao映射文件

```
<select id="findByUid" parameterType="int" resultType="account">
    select * from account where uid=#{uid}
</select>
```

1.4 一对多嵌套查询【User下有多个Role】

1.4.1 编写接口

```
public List<User> findUsersWithRoles();
```

1.4.2 编写UserDao映射文件【两条sql写在一个映射文件中】

```
<resultMap id="userMapWithRole" type="User">
    <id property="id" column="id"/>
    <!--注意这时select中可以直接写sql的id-->
    <collection property="roles" column="id" ofType="Role"
                select="findRolesByUid">
    </collection>
</resultMap>
<select id="findUsersWithRoles" resultMap="userMapWithRole">
    SELECT * FROM USER
</select>
<!--根据用户id查询角色集合-->
<select id="findRolesByUid" parameterType="int" resultType="role">
    SELECT r.id, r.role_name rolename, r.role_desc roledesc FROM role r, user_role ur
    WHERE r.id=ur.rid AND ur.uid=#{uid}
</select>
```

二、加载策略

2.1 什么是加载策略

当多个模型之间存在联系时，在加载一个模型的数据的时候，是否随之加载与其相关模型数据的策略，我们称之为加载策略。

在Mybatis中，常用的加载策略有**立即加载**和**延迟加载（懒加载）**两种。

举个例子:现在有用户和用户登录日志两个模型，当加载1个用户的时候，是否需要立即加载与其相关的登录日志的信息呢?

如果需要立即加载，我们把这种加载策略成为**立即加载**，

如果不需要立即加载，等到真正要使用日志信息的时候再加载，我们把这种加载策略成为**延迟加载**(懒加载)。

2.2 Mybatis加载策略

Mybatis的默认加载策略是**立即加载**，也就是在加载一个对象的时候会立即联合加载到其关联的对象。

那么Mybatis为什么要支持懒加载呢？这就要看看懒加载有什么优缺点了。

> 好处： 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。 。
> 坏处： 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。

2.3 Mybatis加载策略的配置

Mybatis的加载策略的配置分为全局和局部两种方式：

- 在Mybatis 的配置文件中可以使用setting修改全局的加载策略。

```
<setting name="lazyLoadingEnabled" value="true|false (默认值) "/>
```

- 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。
  - fetchType="lazy" 懒加载策略
  - fetchType="eager" 立即加载策略

```
<association fetchType="eager|lazy"></association>
<collection fetchType="eager|lazy"></collection>
```

注意:

- 局部的加载策略优先级高于全局的加载策略。
  - 在配置了延迟加载策略后，即使没有调用关联对象的任何方法，当你调用当前对象的equals、clone、hashCode、toString方法时也会触发关联对象的查询。

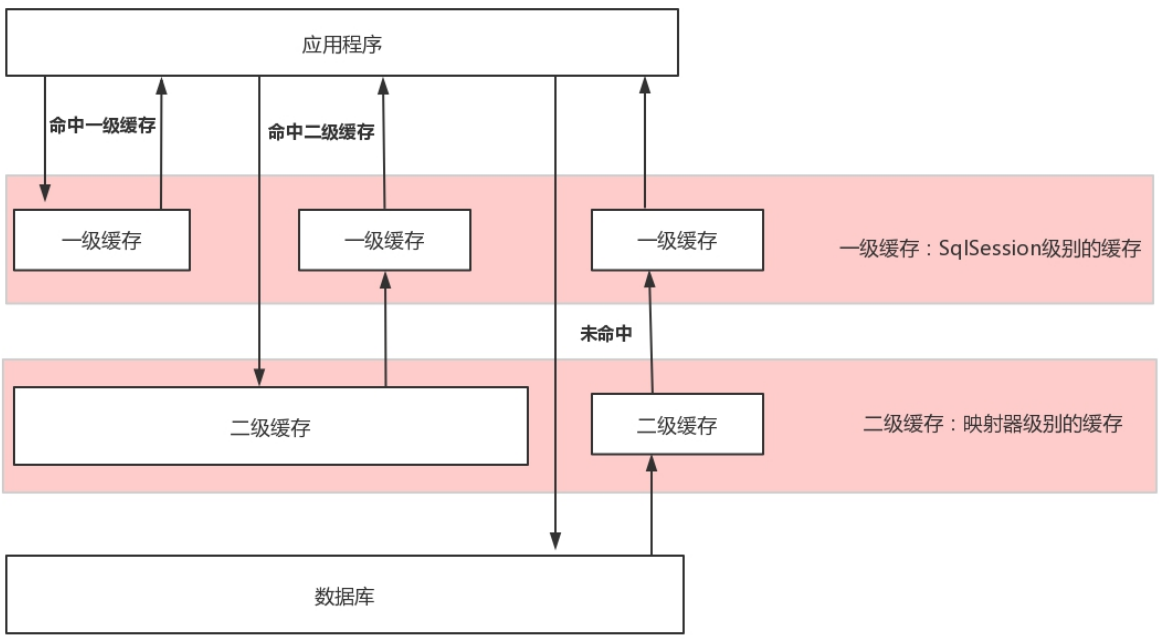
  我们可以在配置文件中使用lazyLoadTriggerMethods配置项覆盖掉上面四个方法。

```
<setting name="lazyLoadTriggerMethods" value="getUid,toString"/>
```

三、缓存机制

缓存是用来提高查询效率的，所有的持久层框架基本上都有缓存机制。

Mybatis有两级缓存，一级缓存是SqlSession级别的，二级缓存是映射器级别的。



3.1 一级缓存

一级缓存是SqlSession级别的缓存，是默认开启且无法关闭的。

```
@Test
public void findById() throws IOException {
    InputStream is = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = userDao.findById(41);//发送sql
    System.out.println(user);
    User userAgain = userDao.findById(41);//不发送sql
    System.out.println(userAgain);

    sqlSession.commit();
    sqlSession.close();

    SqlSession sqlSession1 = sqlSessionFactory.openSession();
    UserDao userDao1 = sqlSession1.getMapper(UserDao.class);
    User user1 = userDao1.findById(41);//发送sql
    System.out.println(user1);
}
```

- 同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。
- 当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。
- 不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

3.2 二级缓存

二级缓存是映射器（Mapper）级别的缓存，是默认开启，但是可以关闭的。

二级缓存的设置：

```
<!--主配置文件中配置cacheEnable为true,这也是默认配置，可以不加-->
<setting name="cacheEnabled" value="true|false"/>
```

```
<!--在Mapper.xml文件中加入cache标签 -->
<cache />
```

验证:

```
@Test
public void findById() throws IOException {
    InputStream is = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(is);

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

```
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = userDao.findById(41);//发送sql
    System.out.println(user);
    User userAgain = userDao.findById(41);//不发送sql
    System.out.println(userAgain);

    sqlSession.commit();
    sqlSession.close();

    SqlSession sqlSession1 = sqlSessionFactory.openSession();
    UserDao userDao1 = sqlSession1.getMapper(UserDao.class);
    User user1 = userDao1.findById(41);//不发送sql
    System.out.println(user1);
}
```

二级缓存是多个SqlSession共享的，不同的sqlSession两次执行相同namespace下的sql语句，第一次执行完毕会将数据库中查询的数据写到二级缓存（内存），第二次会从二级缓存中获取数据，而不再从数据库查询，从而提高查询效率。

注意:

- 执行C（增加）U（更新）D（删除）操作，都会清空缓存。
- 查询语句中这样的配置< select flushCache="true"/>也会清除缓存。

四、配置文件

Mybatis的配置文件中的各个配置项要严格按照dtd中规定的顺序书写，可以省略某些项，但是不能颠倒顺序。

4.1 properties:引入外部配置文件

用来引入外部的配置。

我们经常把一些敏感信息单独放在一个文件中，然后通过properties 引入到配置文件。

举个例子:我们跟数据库相关的信息单独配置

1）创建一个db.properties存放相关信息。

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/mybatisdb
jdbc.username=root
jdbc.password=root
```

2）在mybatis的配置文件中引入配置文件

```
<!--还支持使用<properties url="">的方式引入网络上的配置文件-->
<properties resource="db.properties" />
```

3) 在需要的地方使用el表达式引入需要的值

```
<dataSource type="POOLED">
    <property name="driver" value="${jdbc.driver}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</dataSource>
```

4.2 mappers:注册映射配置

对于将我们的Mapper文件注册到主配置文件，mybatis支持三种方式：

```
<!--1、直接注册Mapper映射文件-->
<mappers>
    <mapper resource="mapper/UserDao.xml"/>
</mappers>

<!--2、直接注册Mapper接口全限定名-->
<mappers>
    <!--此种方式有个前提条件，dao接口必须和dao的xml映射文件在同一个包路径下，而且名称必须一致。-->
    <mapper class="com.itheima.dao.UserDao"/>
</mappers>

<!--3、注册mapper包，包下的配置文件会全部被注册-->
<mappers>
    <!--此种方式有个前提条件，dao接口必须和dao的xml映射文件在同一个包路径下，而且名称必须一致。-->
    <package name="com.itheima.dao"/>
</mappers>
```

4.3 transactionManager:事务控制

Mybatis底层使用的是JDBC的事务管理，只不过在上面封装了一下。

```
<transactionManager type="JDBC"></transactionManager>
```

- 在JDBC中我们可以通过调用Connection的setAutoCommit()方法来开关对事务的支持。

  true代表自动提交事务(默认)，false代表需要手动调用commit()\rollback()等事务控制方法。
- mybatis框架是对JDBC的封装，底层也是调用Connection的setAutoCommit()方法来控制事务的。

  只不过，Mybatis在JDBC的基础上做了修改，**默认手动提交事务。**

```
//Mybatis的事务控制很简单主要是下面三个API
SqlSession sqlSession = SqlSessionFactory.openSession(true);//开启事务(此事务自动提交)
SqlSession sqlSession = SqlSessionFactory.openSession();//提交事务(此事务需要手动提交)

SqlSession.commit();//提交事务
SqlSession.rollback();//回滚事务
```

4.4 dataSource: 数据源

```
<dataSource type="POOLED">
      <!--这里的name都是数据源声明好的，必须按照这个写,每一种数据源都应该提供这些基本的参数接受项-->
      <property name="driver" value="${jdbc.driver}"/>
      <property name="url" value="${jdbc.url}"/>
      <property name="username" value="${jdbc.username}"/>
      <property name="password" value="${jdbc.password}"/>
</dataSource>
```

mybatis中使用了标准的 JDBC 数据源（javax.sql.DataSource）接口来配置 JDBC 连接对象，可以通过mybatis的配置文件dataSource的type属性来指定。可配置项有三个:

- UNPOOLED：不使用连接池的数据源。这个数据源的实现只是每次被请求时打开和关闭连接。
- POOLED：使用连接池的数据源。这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来，避免了创建新的连接实例时所必需的初始化和认证时间。**它继承并扩展了UNPOOLED，增加了连接池的功能。**
- JNDI：不做了解，知道就行。

思考：如何来更换连接池呢？比如我们想druid连接池，怎么办？（扩展）

步骤:

1) 引入druid坐标

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.15</version>
</dependency>
```

2）自定义一个工厂类继承UnpooledDataSourceFactory，在构造方法中返回DruidDataSource数据源

```
public class DruidDataSourceFactory extends UnpooledDataSourceFactory {
    public DruidDataSourceFactory() {
        this.dataSource = new DruidDataSource();
    }
}
```

3）修改配置文件

```
<!--这里的type其实就是自定义产生数据源的工厂-->
<dataSource type="com.itheima.datasource.DruidDataSourceFactory">
    <!--这里的name的值要根据DruidDataSource规定的方式写-->
    <property name="driverClass" value="${jdbc.driver}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</dataSource>
```

五、mybatis注解开发

Mybatis除了支持在xml中书写SQL,也支持使用注解的形式编写SQL。

Mybatis的常见注解有：

- @Insert:实现新增
- @Update:实现更新
- @Delete:实现删除
- @Select:实现查询
- @Result:实现结果集封装
- @Results:可以与@Result 一起使用，封装多个结果集
- @ResultMap：可以引用Results
- @One:实现一对一结果集封装
- @Many:实现一对多结果集封装

使用注解，要在配置文件中引入接口文件

```
<mappers>
        <mapper class="com.itheima.mapper.UserMapper2" />
</mappers>
```

5.1 单表CRUD注解开发

```
public interface UserMapper {
      @Insert("insert into user(name) values(#{name})")
      public void insertT(User user);

      @Delete("delete from user where id=#{id}")
      public void deleteById(int id);

      @Update("update user set name=#{name} where id=#{id}")
      public void updateT(User user);

      @Select("select * from user where id=#{id}")
      public User getUserById(int id);

      @Select("select * from user")
      public List<User> getAll();
}
```

5.2 手动指定对象关系映射【类似于xml中resultMap功能】

```
//@Results   相当于<resultMap id=""  type="">
@Results(
      id = "userMap", //id是标识，相当于resultMap的Id，是一个唯一标识
      value = {
            //@相当于<id column="" property="" />
            @Result(column = "uid",property = "uid",id = true),
            //@Result相当于<property column="" property="" />
            @Result(column = "phone_number", property = "phoneNumber"),
            @Result(column = "uid",property = "uid")
      }
)
@Select("select * from user")
List<User> findAll();

@ResultMap("userMap")  //相当于select标签中的resultMap,   <select  resultMap="userMap">
@Select("select * from user where uid = #{uid}")
User findByUid(Integer uid);
```

5.3 多表关联查询注解开发

5.3.1 一对一关联查询

```
//AccountDao接口
@Select("select * from account")
@Results(id = "accountMap", value = {
    @Result(property = "user", column = "uid", javaType = User.class,
            one = @One(select = "com.itheima.dao.UserDao.findById", fetchType = FetchType.LAZY))
})
public List<Account> findAccountWithUser();

//UserDao接口
@Select("select * from user where id = #{id}")
public User findById(Integer id);
```

5.3.2 一对多关联查询

5.3.2.1 用户到账户

```
//UserDao接口
@Select("select * from user")
@Results({
    @Result(id = true, property = "id", column = "id"),
    @Result(property = "accounts", column = "id", javaType = List.class,
            many = @Many(select = "com.itheima.dao.AccountDao.findByUid"))
})
public List<User> findUsersWithAccounts();

//AccountDao接口
@Select("select * from account where uid = #{uid}")
public List<Account> findByUid(Integer uid);
```

5.3.2.2 用户到角色

```
//UserDao接口
@Select("select * from user")
@Results({
    @Result(id = true, property = "id", column = "id"),
    @Result(property = "roles", column = "id", javaType = List.class,
            many = @Many(select = "com.itheima.dao.RoleDao.findByUid"))
})
public List<User> findUsersWithRoles();

//RoleDao接口
@Select("SELECT r.id, r.role_name rolename, r.role_desc roledesc FROM role r, user_role ur " +
        "WHERE r.id=ur.rid AND ur.uid=#{uid}")
public List<Role> findByUid(Integer uid);
```