

延迟加载和查询缓存

1、概要

动态SQL

动态SQL，主要用于解决查询条件不确定的情况，在程序运行期间，根据用户提交的查询条件进行查询，提交产讯条件不同，执行的sql不同，若将每种可能的情况主键一一列出，对所有的条件进行排列组合，将会出现大量的SQL语句，此时，可使用动态SQL来解决这样的问题。

用户自定义查询			
姓名:	<input type="text"/>		
学号:	<input type="text"/>		
年龄:	从	<input type="text"/>	到
成绩:	从	<input type="text"/>	到
<input type="button" value="查询"/>			

动态SQL，即通过MyBatis提供的各种标签对条件做出判断以实现动态拼接sql语句。常用的动态SQL标签有<if>,<where>,<choose>,<foreach>等。其语句形式与JSTL中的语句详细。

2、实例

- if标签

- StudentMapper类

```
//多条件查询:动态代理
public List<Student> findStuByAll(int id,String sname ,int sage ,String
ssex);
```

- StudentMapper.xml

```
<select id="findStuByAll" resultType="Student">
    select * from stu where 1 = 1
    <if test="arg0 != 0">
        AND id = #{arg0}
    </if>
    <if test="arg0 != null">
        AND sname like '%' #{arg1} '%'
    </if>
</select>
```

- studentTest类

```
@Test
//多条件查询: 动态代理
public void findStuByAllTest(){
    List<Student> studentList = studentMapper.findStuByAll(0,"沛",0,null);
    for (Student str:studentList){
        System.out.println(str);
    }
}
```

- where标签
 - StudentMapper类
 - StudentMapper.xml

```

<!--: 多条件查询:动态代理-->
<select id="findStuByAll" resultType="Student">
    select * from stu
    <where>
        <if test="arg0 != 0">
            AND id = #{arg0}
        </if>
        <if test="arg0 != null">
            AND sname like '%' #{arg1} '%'
        </if>
    </where>
</select>

```

- studentTest类
- choose标签
 - StudentMapper类
 - StudentMapper.xml

```

<!--: 多条件查询:动态代理-->
<select id="findStuByAll" resultType="Student">
    select * from stu
    <where>
        <choose>
            <when test="arg0 != 0">
                AND id = #{arg0}
            </when>
            <when test="arg0 != null">
                AND sname like '%' #{arg1} '%'
            </when>
        </choose>
    </where>
</select>

```

- studentTest类
- foreach标签

foreach标签

<foreach>标签用于实现对数组与集合的遍历，对其使用，需要注意：

- collection:表示要遍历的集合类型，例如数组（array，list）
- open、close、separator为对遍历内容的SQL拼接

- 遍历数组

- StudentMapper类

```
//多条件查询: 动态代理, 数组
public List<Student> findStudentByInCondition(int[] arr);
```

- StudentMapper.xml

```
<!--: 多条件查询: 动态代理, 数组-->
<select id="findStudentByInCondition" resultType="Student">
    select * from stu
    <where>
        id in
        <foreach collection="array" item="id" open="(" separator=","
close=")">
            #{id}
        </foreach>
    </where>
</select>
```

- studentTest类

```
@Test
//多条件查询: 动态代理, 数组
public void findStudentByInConditionTest(){
    List<Student> studentList =
studentMapper.findStudentByInCondition(new int[]{1,2,3,4,5});
    for (Student str:studentList){
        System.out.println(str);
    }
}
```

- 遍历基本类型的List

- StudentMapper类

```
//多条件查询: 动态代理, list集合
public List<Student> findStudentByInCondition(List<Integer> arr);
```

- StudentMapper.xml

```

<!--: 多条件查询:动态代理, list-->
<select id="findStudentByInCondition" resultType="Student">
    select * from stu
    <where>
        id in
        <foreach collection="list" item="id" open="(" separator=","
close=")">
            #{id}
        </foreach>
    </where>
</select>

```

■ studentTest类

```

@Test
//多条件查询: 动态代理, list
public void findStudentByInConditionTest(){
    List<Integer> list = new ArrayList<>();
    list.add(1);
    list.add(3);
    list.add(5);
    list.add(7);
    list.add(9);
    List<Student> studentList =
studentMapper.findStudentByInCondition(list);
    for (Student str:studentList){
        System.out.println(str);
    }
}

```

○ 遍历自定义类型的List

■ StudentMapper类

```

//多条件查询: 动态代理, 自定义类型地list
public List<Student> findStudentByInCondition(List<Student> arr);

```

■ StudentMapper.xml

```

<!--: 多条件查询:动态代理, 自定义类型-->
<select id="findStudentByInCondition" resultType="Student">
    select * from stu
    <where>
        id in
        <foreach collection="list" item="student" open="("
separator="," close=")">
            #{student.id}
        </foreach>
    </where>
</select>

```

■ studentTest类

```
//多条件查询：动态代理，自定义类型Student
public void findStudentByInConditionTest(){
    List<Student> list = new ArrayList<>();
    Student stu1 = new Student();
    stu1.setId(1);
    Student stu2 = new Student();
    stu2.setId(3);
    Student stu3 = new Student();
    stu3.setId(5);
    Student stu4 = new Student();
    stu4.setId(7);
    Student stu5 = new Student();
    stu5.setId(9);
    list.add(stu1);
    list.add(stu2);
    list.add(stu3);
    list.add(stu4);
    list.add(stu5);
    List<Student> studentList =
studentMapper.findStudentByInCondition(list);
    for (Student str:studentList){
        System.out.println(str);
    }
}
```

• 和

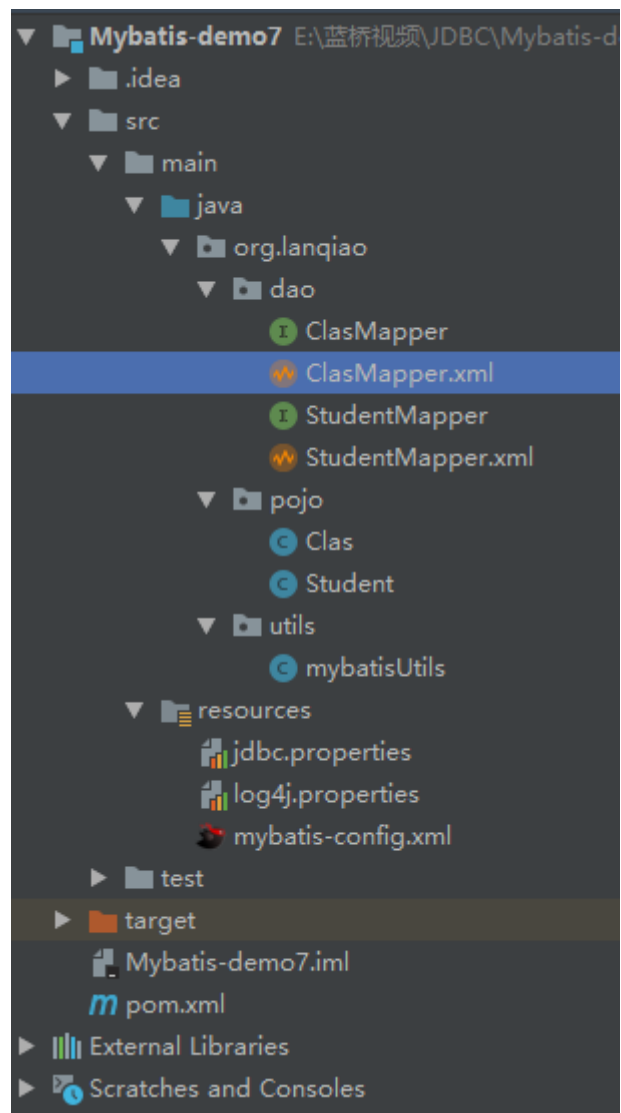
```
<!--sql片段-->
<sql id="basesql">
    select * from stu
</sql>
<!--：多条件查询:动态代理，自定义类型-->
<select id="findStudentByInCondition" resultType="Student">
    <!--包含sql片段-->
    <include refid="basesql"></include>
    <where>
        id in
        <foreach collection="list" item="student" open="(" separator=","
close=")">
            #{student.id}
        </foreach>
    </where>
</select>
```

• 当当

3、高级查询(多表查询)

• 1、关联查询：查询内容涉及具有多个关系的多个表时

项目结构图:



- 一对多：班级关联学生
 - 多表链接查询
 - pojo层：实体类

```
public class Clas {  
    private int cid;  
    private String cname;  
    //一个班级对应多个学生, Set集合  
    private Set<Student> studentSet;  
    ...  
    ...  
}  
  
public class Student {  
    private int id;  
    private String sname;  
    private int sage;  
    private String ssex;
```

```

        private int cid;
        ...
        ...
    }

```

■ dao层

■ ClasMapper类

```

//查询：根据班级cid查询班级中学生信息
public Clas findCidByCid(int cid);

```

■ ClasMapper.xml

```

<?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">
<!--接口-->
<mapper namespace="org.lanqiao.dao.ClasMapper">
    <resultMap id="clasAndstu" type="Clas"><!--即:clas类中,属性的映射-->
        <id column="cid" property="cid"></id>
        <result column="cname" property="cname"></result><!--在其中包含一个集合-->
        <!--设置关联的集合属性,即:Student类,属性的映射-->
        <collection property="studentSet" ofType="Student"><!--属性,对应的类型-->
            <id column="id" property="id"></id>
            <result column="sname" property="sname"></result>
            <result column="ssex" property="ssex"></result>
        </collection>
    </resultMap>
    <select id="findCidByCid" resultMap="clasAndstu">/*结果如何去映射,映射一个实体*/
        select c.cid,c.cname,s.id,s.sname,s.ssex from class c,stu
        s where c.cid = s.cid and c.cid = #{cid};
    </select>
</mapper>

```

■ test层

```

@org.junit.Test
//多表链接查询
public void findClsByCidTest(){
    Clas clas =clasMapper.findCidByCid(1);
    System.out.println(clas);
}

```

■ 当当

■ 多表单独查询

- dao层

- 接口

- StudentMapper

```
public interface StudentMapper {  
    //查询:根据cid获取学生信息  
    public List<Student> findStuByCid();  
}
```

- ClasMapper

- 实现

- StudentMapper.xml

```
<?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">  
<!--接口-->  
<mapper namespace="org.lanqiao.dao.StudentMapper">  
    <select id="findStuByCid" resultType="Student">  
        select * from stu where cid = #{cid};  
    </select>  
</mapper>
```

- ClasMapper.xml

```
<mapper namespace="org.lanqiao.dao.ClasMapper">  
    <!--<resultMap id="clasAndstu" type="Clas">&lt;!&dash;  
    即:clas类中,属性的映射&dash;&gt;  
        <id column="cid" property="cid"></id>  
        <result column="cname" property="cname">  
</result>&lt;!&dash;在其中包含一个集合&dash;&gt;  
        &lt;!&dash;设置关联的集合属性,即:student类,属性的映射  
&dash;&gt;  
        <collection property="studentSet"  
ofType="Student">&lt;!&dash;属性,对应的类型&dash;&gt;  
            <id column="id" property="id"></id>  
            <result column="sname" property="sname">  
</result>  
                <result column="ssex" property="ssex">  
</result>  
            </collection>  
        </resultMap>  
        <select id="findCidByCid" resultMap="clasAndstu">/*结果如何  
去映射,映射一个实体*/  
            select c.cid,c.cname,s.id,s.sname,s.ssex from class  
c,stu s where c.cid = s.cid and c.cid = #{cid};  
        </select>-->  
        <resultMap id="clsAndStu" type="org.lanqiao.pojo.Clas">  
            <id column="cid" property="cid"></id>
```



```

        <result column="cname" property="cname"></result>
        <collection property="studentSet" ofType="Student"
select="org.lanqiao.dao.StudentMapper.findStuByCid"
column="cid">
            <id column="id" property="id"></id>
            <result column="sname" property="sname"></result>
            <result column="ssex" property="ssex"></result>
        </collection>
    </resultMap>
    <select id="findCidByCid" resultMap="clsAndStu">
        select * from class where cid = #{cid};
    </select>
</mapper>

```

- 当当
- 多对一
 - 多表链接查询
 - pojo层

```

public class Student {
    private int id;
    private String sname;
    private int sage;
    private String ssex;
    private int cid;
    //学生关联班级：添加一个班级属性
    private Clas clas;

    public Student(){

    }

    ...
    ...
}

public class Clas {
    private int cid;
    private String cname;

    public Clas(){

    }

    ...
    ...
}

```

- dao层
 - StudentMapper类

```
//查询: 根据sid获取学生信息, 包括他的班级信息
public Student findStuBySid(int id);
```

- StudentMapper.xml

```
<!--查询: 一对多: 根据sid查询学生信息包括学生的班级信息-->
<resultMap id="stuAndCls" type="org.lanqiao.pojo.Student">
    <id column="id" property="id"></id>
    <result column="sname" property="sname"></result>
    <result column="sage" property="sage"></result>
    <result column="ssex" property="ssex"></result>
    <!--设置关联的集合属性, 即: Clas类, 属性的映射-->
    <association property="clas" javaType="Clas">
        <id column="cid" property="cid"></id>
        <result column="cname" property="cname"></result>
    </association>
</resultMap>
<select id="findStuBySid" resultMap="stuAndCls">
    select s.id,s.sname,s.sage,s.ssex,c.cid,c.cname from stu
s,class c where s.cid = c.cid and s.id = #{id};
</select>
```

- test层

- test类

```
@org.junit.Test
//多表链接查询: 多对一
public void findStuBySidTest(){
    Student student =studentMapper.findStuBySid(2);
    System.out.println(student);
}
```

- 多表单独查询

- dao层

- StudentMapper类

```
//单表查询: 根据sid获取学生信息, 包括他的班级信息
public Student findStuBySids(int id);
```

- StudentMapper.xml

```
<!--单表查询-->
<resultMap id="stuToCls" type="Student">
    <id column="id" property="id"></id>
    <result column="sname" property="sname"></result>
    <result column="sage" property="sage"></result>
    <result column="ssex" property="ssex"></result>
    <!--设置关联的集合属性, 即: clas类, 属性的映射-->
```

```

        <association property="clas" javaType="Clas"
select="org.lanqiao.dao.ClasMapper.findCidByCid" column="cid">
        <id column="cid" property="cid"></id>
        <result column="cname" property="cname"></result>
    </association>
</resultMap>
<select id="findStuBySids" resultMap="stuToCls">
    select id,sname,sage,ssex from stu where id = #{id};
</select>

```

■ test层

```

@org.junit.Test
//单表查询: 多对一
public void findStuBySidsTest(){
    Student student = studentMapper.findStuBySids(2);
    System.out.println(student);
}

```

○ 多对多

多对多

多对多关联！ 其实就是两个一对多的关联！ 比如说 一个学生可以有多个老师！ 一个老师可以有多个学生！

那么 学生和老师之间的关系 可以理解为 多对多的关联关系！

关键是怎么建立数据库中两个表之间的关系？？？

这时候需要一个中间表来组织两张表的关系！

创建对应的数据库表！

■ pojo层

■ Teacher类

```

public class Teacher {
    private int tid;
    private String tname;
    private Set<Student> studentSet = new HashSet<>();

    public Teacher(){

    }

    ...

    ...
}

```

■ Student类

```
public class Student {
    private int sid;
    private String sname;
    private int sage;
    private String ssex;
    private Set<Teacher> teachersSet = new HashSet<>();

    public Student(){

    }
}
```

■ dao层

■ StudentMapper类

```
public interface StudentMapper {
    //根据学生信息查询所有对应老师的信息
    public Student findStuAndTeaBySid(int id);
}
```

■ StudentMapper.xml

```
<resultMap id="stuAndTea" type="Student">
    <id column="sid" property="sid"></id>
    <result column="sname" property="sname"></result>
    <result column="sage" property="sage"></result>
    <result column="ssex" property="ssex"></result>
    <collection property="teachersSet" ofType="Teacher">
        <id column="tid" property="tid"></id>
        <result column="tname" property="tname"></result>
    </collection>
</resultMap>
<!--多表查询：关联查询：根据学生信息查询所有对应老师的信息-->
<select id="findStuAndTeaBySid" resultMap="stuAndTea">
    select s.sid,sname,sage,ssex,t.tid,tname from stu s,teacher
t,stu_tea st where s.sid = st.id and t.tid = st.id and s.sid = #{sid};
</select>
```

■ TeacherMapper类

```
//根据老师信息查询所有对应学生的信息
public Teacher findTeaAndStuByTid(int id);
```

■ TeacherMapper.xml

```
<resultMap id="teaAndStu" type="Teacher">
    <id column="tid" property="tid"></id>
    <result column="tname" property="tname"></result>
    <collection property="studentSet" ofType="Student">
        <id column="sid" property="sid"></id>
        <result column="sname" property="sname"></result>
```

```

        <result column="sage" property="sage"></result>
        <result column="ssex" property="ssex"></result>
    </collection>
</resultMap>
<select id="findTeaAndStuByTid" resultMap="teaAndStu">
    select t.tid,tname,s.sid,sname,sage,ssex from teacher t,stu
    s,stu_tea st where t.tid = st.id
    and s.sid =st.id and t.tid = #{tid};
</select>

```

■ test层

```

@org.junit.Test
public void findStuAndTeaBySidTest(){
    Student student =studentMapper.findStuAndTeaBySid(1);
    System.out.println(student);
}

@org.junit.Test
public void findTeaAndStuByTidTest(){
    Teacher teacher = teacherMapper.findTeaAndStuByTid(2);
    System.out.println(teacher);
}

```

○ 当当

• 2、延迟加载

延迟加载

MyBatis中的延迟加载，也称为懒加载（lazy load），是指在进行关联查询时，按照设置延迟加载规则推迟对关联对象的select查询。延迟加载可以有效的减少数据库压力。延迟加载机制是为了避免一些无谓的性能开销而提出来的。

所谓延迟加载就是当在真正需要数据的时候，才真正执行数据加载操作。

延迟加载，可以简单理解为，只有在使用的时候，才会发出sql语句进行查询。

延迟加载的有效期是在session打开的情况下，当session关闭后，会报异常。当调用load方法加载对象时，返回代理对象，等到真正用到对象的内容时才发出sql语句。

需要注意的是，MyBatis的延迟加载只是对关联对象的查询有延迟设置，对于主加载对象是直接执行查询语句的。

延迟加载

MyBatis根据对关联对象查询的select语句的执行时机，分为3种类型：

- 1.直接加载
- 2.侵入式延迟加载
- 3.深度延迟加载

注意：延迟加载的应用要求：

关联对象的查询与主加载对象的查询必须是分别进行的select语句，不能是使用多表连接所进行的select查询。

因为，多表连接查询，实质是对一张表的查询，对由多个表连接后形成的一张表的查询。会一次性将多张表的所有信息查询出来。

○ 应用场景

应用场景

`resultMap`可以实现高级映射（使用`association`、`collection`实现一对一及一对多映射），`association`、`collection`具备延迟加载功能。

需求：如果查询订单并且关联查询用户信息。如果先查询订单信息即可满足要求，当我们需要查询用户信息时再查询用户信息。把对用户信息的按需去查询就是延迟加载。

延迟加载：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。

如何配置加载

settings

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。下表描述了设置中各项的意图、默认值等。

设置参数	描述	有效值	默认
cacheEnabled	全局地开启或关闭配置文件中的所有映射器已经配置的任何缓存。	true false	true
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。特定关联关系中可通过设置 <code>fetchType</code> 属性来覆盖该项的开关状态。	true false	false
aggressiveLazyLoading	当开启时，任何方法的调用都会加载该对象的所有属性。否则，每个属性会按需加载（参考 <code>lazyLoadTriggerMethods</code> ）。	true false	false (true in ≤3.4.1)

```
<!--mybatis的延迟加载的总开关 -->
<setting name="lazyLoadingEnabled" value="false"></setting>
```

直接加载

```
<setting name="lazyLoadingEnabled" value="false"></setting><setting
name="aggressiveLazyLoading" value="false"></setting>
```

深度延迟加载

深度延迟加载

执行对主加载对象的查询时，不会执行对关联对象的查询。访问主加载对象的详情时也不会执行关联对象的`select`查询。只有当真正访问关联对象的详情时，才会执行对关联对象的`select`查询。

修改主配置文件：

```
<!-- 全局参数设置 -->
<settings>
  <!-- 延迟加载总开关 -->
  <setting name="lazyLoadingEnabled" value="true"/>
  <!-- 侵入式延迟加载开关 -->
  <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

其他代码均不作改变，此时运行会发现，只有当代码执行到`Student`对象详情时，底层才执行了`select`语句对`stu`表进行了查询，这已经将查询推迟到了不能在推的时间，故称为深度延迟加载。

```
<setting name="lazyLoadingEnabled" value="true"></setting>
<setting name="aggressiveLazyLoading" value="false"></setting>
```

侵入式延迟加载

执行对主加载对象的查询时，不会执行对关联对象的查询。但是当要访问主加载对象的详情时（此处的详情可以只是主加载对象本身的一个属性），**就会**马上执行关联对象的select查询。

即对关联对象的查询执行，侵入到了主加载对象的详情访问中。也可以这样理解：将关联对象的详情侵入到了主加载对象的详情中，即将关联对象的详情作为主加载对象详情的一部分出现了

修改配置文件：

```
<!-- 全局参数设置 -->
<settings>
  <!-- 延迟加载开关 -->
  <setting name="lazyLoadingEnabled" value="true"/>
  <!-- 侵入式延迟加载开关 -->
  <setting name="aggressiveLazyLoading" value="true"/>
</settings>
```

```
<setting name="aggressiveLazyLoading" value="true"></setting>
<setting name="lazyLoadingEnabled" value="true"></setting>
```

◦ 当当

• 3、查询缓存

为什么要使用查询缓存？查询缓存的使用，主要是为了提高查询访问速度，将用户对同一数据的重复查询过程简化，不再每次均从数据库中查询获取结果数据，从而提高访问速度

正如大多数持久层框架一样，MyBatis 同样提供了一级缓存和二级缓存的支持 一级缓存: 基于 PerpetualCache 的 HashMap本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session中的所有 Cache 就将清空。

1. 二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。
2. 对于缓存数据更新机制，当某一个作用域(一级缓存Session/二级缓存Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被clear。

◦ 一级缓存

- MyBatis一级缓存是基于org.apache.ibatis.cache.impl.PerpetualCache类的HashMap本地缓存，其作用域是SqlSession。在同一个SqlSession中两次执行相同的sql查询语句，第一次执行完毕后，会将查询结果写入缓存中，第二次会从缓存中直接获取数据，而不在查询数据库。当一个sqlsession结束后，该SqlSession中的依据缓存也就不存在了。MyBatis默认一级缓存是开启的状态，且不能关闭

- 证明一级缓存存在

Test类

```
@org.junit.Test
//单表查询: 多对一
public void findStuBySidsTest() {
    Student student = studentMapper.findStuBySids(2);
    System.out.println(student);
    Student student1 = studentMapper.findStuBySids(2);
    System.out.println(student1);
}
```



```

==> Preparing: select id,sname,sage,ssex from stu where id = ?;
[DEBUG] 2018-10-26 09:54:50,768 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(Base
==> Parameters: 2(Integer)
[DEBUG] 2018-10-26 09:54:50,798 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(Base
<==      Total: 1
Student{id=2, sname='李斯', sage=23, ssex='男', cid=0, clas=null}
Student{id=2, sname='李斯', sage=23, ssex='男', cid=0, clas=null}

```

当一个sqlsession结束后，该SqlSession中的依据缓存也就不存在了。

```

Student student = studentMapper.findStuBySids(2);
System.out.println(student);
sqlSession.close();
Student student1 = studentMapper.findStuBySids(2);
System.out.println(student1);

```

```

<==      Total: 1
Student{id=2, sname='李斯', sage=23, ssex='男', cid=0, clas=null}
[DEBUG] 2018-10-26 10:00:52,376 method:org.apache.ibatis.transaction.jdbc.JdbcTransaction.resetAutoCommit(JdbcTransac
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@2a40cd94]
[DEBUG] 2018-10-26 10:00:52,377 method:org.apache.ibatis.transaction.jdbc.JdbcTransaction.close(JdbcTransaction.java:
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@2a40cd94]
[DEBUG] 2018-10-26 10:00:52,379 method:org.apache.ibatis.datasource.pooled.PooledDataSource.pushConnection(PooledData
Returned connection 708890004 to pool.

org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause: org.apache.ibatis.executor.ExecutorException: Executor was closed.
### Cause: org.apache.ibatis.executor.ExecutorException: Executor was closed.

```

■ 当当

○ 二级缓存

- 1、MyBatis查询缓存的作用域是根据映射文件的mapper的namespace划分的，相同的namespace的mapper查询数据存放在同一个缓存区域，不同的namespace下的数据互不干扰。
- 2、无论是一级缓存还是二级缓存，都是按照namespace进行分别存放的。
- 3、但是一、二及缓存的不同之处在于，sqlSession一旦关闭，则SqlSession中的数据将不存在，即一级缓存就不复存在。而二级缓存的生命周期会与整个应用同步，与SqlSession是否关闭无关。
- 4、二级缓存的用法: (1) 实体序列化--实现Serializable序列化接口

二级缓存的常用设置 <cache

eviction="FIFO" //回收策略为先进先出

flushInterval="60000" //自动刷新时间60s size="512" //最多缓存512个引用对象

readOnly="true"/> //只读 eviction:回收策略。当二级缓存中的对象达到最大值时，就需要通过回收策略将缓存中的对象移除缓存，默认为LRU，常用的策略有： FIFO: first in first out 先进先出 LRU: Least recently Used 未被使用时间最长的 补充说明

映射语句文件中的所有select语句将会被缓存。

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

缓存会使用Least Recently Used (LRU, 最近最少使用的) 算法来收回。

缓存会根据指定的时间间隔来刷新。

缓存会存储1024个对象

■ 验证增删改对二级缓存的影响

- 二级缓存关闭
- 级缓存的使用原则
- ehcache二级查询缓存



- 引入依赖

```
<!--引入ehcache缓存-->
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.5.2</version>
</dependency>
<!--
https://mvnrepository.com/artifact/org.mybatis.caches/mybatis-ehcache
-->

<!--mybatis整合ehcache-->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.1.0</version>
</dependency>
```

- 添加ehcache的配置: ehcache.xml

```
<ehcache>
<!--
    磁盘存储:将缓存中暂时不使用的对象,转移到硬盘,类似于windows系统的虚拟内存
    path:指定在硬盘上存储对象的路径
-->
<diskStore path="java.io.tmpdir" />
<!--
    defaultCache:默认的缓存配置信息,如果不加特殊说明,则所有对象按照此配置项
    处理
    maxElementsInMemory:设置了缓存的上限,最多存储多少个记录对象
    eternal:代表对象是否永不过期
    timeToIdleSeconds:最大的空闲时间
    timeToLiveSeconds:最大的存活时间
    overflowToDisk:是否允许对象被写入到磁盘
-->
<defaultCache maxElementsInMemory="10000" eternal="false"
```

```

        timeToIdleSeconds="120" timeToLiveSeconds="120"
        overflowToDisk="true" />
        <!--
            cache:为指定名称的对象进行缓存的特殊配置
            name:指定对象的完整名
        -->
        <cache name="org.lanqiao.pojo.Student" maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="300" timeToLiveSeconds="600"
        overflowToDisk="true" />
    </ehcache>

```

- 切换默认的二级缓存的实现:StudentMapper.xml

```

<mapper namespace="org.lanqiao.dao.StudentMapper">
    <!--开启二级缓存-->
    <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
    ...
</mapper>

```

- 二级缓存的使用原则

01. 很少被修改的数据
02. 不是很重要的数据，允许出现偶尔并发的数据
03. 不会被并发访问的数据
04. 多个namespace不能操作同一张表

由于二级缓存中的数据是基于namespace的，即不同的namespace中若均存在对同一个表的操作，那么这多个namespace中的数据可能会出现不一致的现象。
05. 不能在关联关系表上执行增删改操作

- 当当

4、获取新增数据的自动增长的主键列