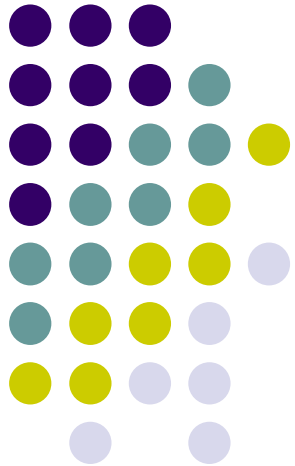
Mybatis技术简介

dezhaos@gmail.com



关联映射

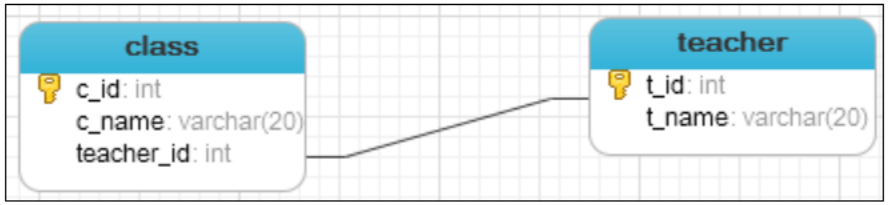
- 一对一
- 一对多
- 多对多

一对一关联

- **需求**
- 根据班级id查询班级信息(带老师的信息)
- 创建一张教师表和班级表，这里我们假设一个老师只负责教一个班，那么老师和班级之间的关系就是一种一对一的关系。

- CREATE TABLE teacher(
- t_id INT PRIMARY KEY AUTO_INCREMENT,
- t_name VARCHAR(20)
-);
- CREATE TABLE class(
- c_id INT PRIMARY KEY AUT(
- c_name VARCHAR(20),
- teacher_id INT
-);
- **ALTER TABLE class ADD CONSTRAINT fk_teacher_id FOREIGN KEY (teacher_id) REFERENCES teacher(t_id);**

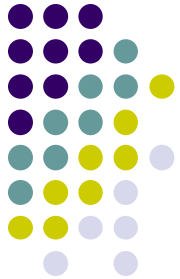
表之间的关系如下：



- INSERT INTO teacher(t_name) VALUES('teacher1');
- INSERT INTO teacher(t_name) VALUES('teacher2');

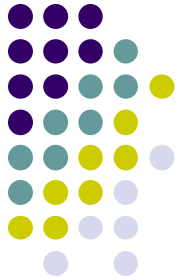
- INSERT INTO class(c_name, teacher_id) VALUES('class_a', 1);
- INSERT INTO class(c_name, teacher_id) VALUES('class_b', 2);

Teacher



```
package ssm.test;
public class Teacher {
    //定义实体类的属性，与teacher表中的字段对应
    private int id;           //id===>t_id
    private String name;    //name===>t_name
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Classes类，Classes类是class表对应的实体类



```
package ssm.test;
public class Classes {
    //定义实体类的属性，与class表中的字段对应
    private int id;          //id===>c_id
    private String name;    //name===>c_name
    /*** class表中有一个teacher_id字段，所以在Classes类中定义一个teacher属性，
     * 用于维护teacher和class之间的一对一关系，通过这个teacher属性就可以知道这个班级是由哪个老师负责的   */
    private Teacher teacher;
  public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
```

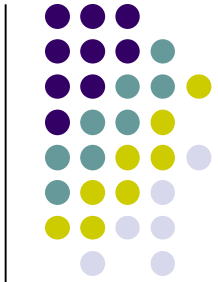
```
public Teacher getTeacher() {
        return teacher;
    }
    public void setTeacher(Teacher teacher)
{
        this.teacher = teacher;
    }
}
```

定义sql映射文件classMapper.xml

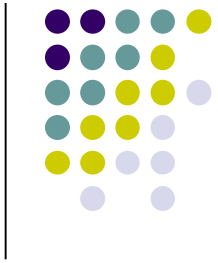
```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="ssm.test.classMapper">
<!--
    方式一：嵌套结果：使用嵌套结果映射来处理重复的联合结果的子集
          封装联表查询的数据(去除重复的数据)
      select * from class c, teacher t where c.teacher_id=t.t_id and c.c_id=1
    -->
    <select id="getClass" parameterType="int" resultMap="ClassResultMap">
      select * from class c, teacher t where c.teacher_id=t.t_id and c.c_id=#{id}
    </select>
    <!-- 使用resultMap映射实体类和字段之间的一一对应关系 -->
    <resultMap type=“ssm.test.Classes" id="ClassResultMap">
      <id property="id" column="c_id"/>
      <result property="name" column="c_name"/>
      <association property="teacher" javaType=“ssm.test.Teacher">
        <id property="id" column="t_id"/>
        <result property="name" column="t_name"/>
      </association>
    </resultMap>
```

```
<!--
  方式二：嵌套查询：通过执行另外一个SQL映射语句来返回预期的复杂类型
    SELECT * FROM class WHERE c_id=1;
    SELECT * FROM teacher WHERE t_id=1   //1 是上一个查询得到的teacher_id的值
  -->
  <select id="getClass2" parameterType="int" resultMap="ClassResultMap2">
    select * from class where c_id=#{id}
  </select>
  <!-- 使用resultMap映射实体类和字段之间的一一对应关系 -->
  <resultMap type="ssm.test.Classes" id="ClassResultMap2">
    <id property="id" column="c_id"/>
    <result property="name" column="c_name"/>
    <association property="teacher" column="teacher_id" select="getTeacher"/>
  </resultMap>

  <select id="getTeacher" parameterType="int" resultType="ssm.test.Teacher">
    SELECT t_id id, t_name name FROM teacher WHERE t_id=#{id}
  </select>

</mapper>
```

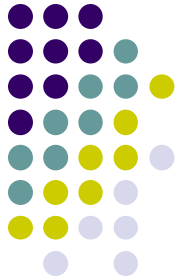
在conf.xml文件中注册classMapper.xml



- <mappers>
- <mapper resource=“ssm/test/classMapper.xml"/>
- </mappers>

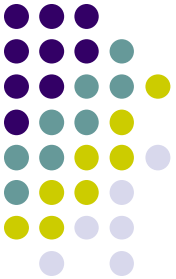


```
public class Test3 {
@Test
    public void testGetClass(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        /**
         * 映射sql的标识字符串，
         * me.gacl.mapping.classMapper是classMapper.xml文件中mapper标签的namespace属性的值， getClass是select标签的id属性值，通过select标签的id属性值就可以找到要执行的SQL
         */
        String statement = “ssm.test.classMapper.getClass";//映射sql的标识字符串
        //执行查询操作，将查询结果自动封装成Classes对象返回
        Classes clazz = sqlSession.selectOne(statement,1);//查询class表中id为1的记录
        //使用SqlSession执行完SQL之后需要关闭SqlSession
        sqlSession.close();
        System.out.println(clazz);
    }
}
```



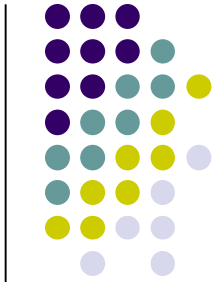
```
@Test
    public void testGetClass2(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        /**
         * 映射sql的标识字符串，
         * me.gacl.mapping.classMapper是classMapper.xml文件中mapper标签的namespace属性的值，
         * getClass2是select标签的id属性值，通过select标签的id属性值就可以找到要执行的SQL
         */
        String statement = "ssm.test.classMapper.getClass.getClass2";//映射sql的标识字符串
        //执行查询操作，将查询结果自动封装成Classes对象返回
        Classes clazz = sqlSession.selectOne(statement,1);//查询class表中id为1的记录
        //使用SqlSession执行完SQL之后需要关闭SqlSession
        sqlSession.close();
        System.out.println(clazz);
    }
```

MyBatis一对一关联查询总结



- MyBatis中使用association标签来解决一对一的关联查询，association标签可用的属性如下：
  - property:对象属性的名称
  - javaType:对象属性的类型
  - column:所对应的外键字段名称
  - select:使用另一个查询封装的结果

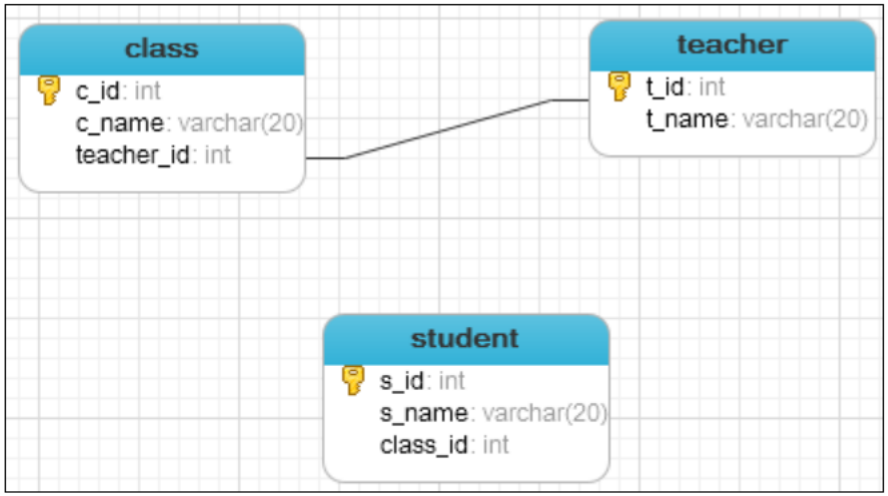
一对多关联



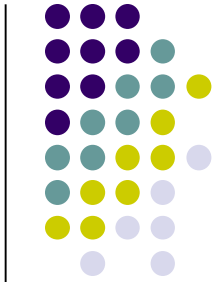
- **需求**
- 根据classId查询对应的班级信息,包括学生,老师。班级与学生之间存在一对多的关系

创建表和数据

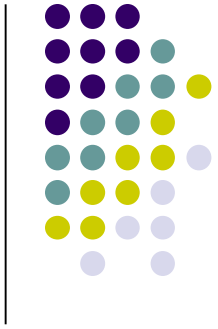
- CREATE TABLE student(
- s_id INT PRIMARY KEY AUTO_INCREMENT,
- s_name VARCHAR(20),
- class_id INT
-);
- INSERT INTO student(s_name, class_id) VALUES('student_A', 1);
- INSERT INTO student(s_name, class_id) VALUES('student_B', 1);
- INSERT INTO student(s_name, class_id) VALUES('student_C', 1);
- INSERT INTO student(s_name, class_id) VALUES('student_D', 2);
- INSERT INTO student(s_name, class_id) VALUES('student_E', 2);
- INSERT INTO student(s_name, class_id) VALUES('student_F', 2);



定义实体类

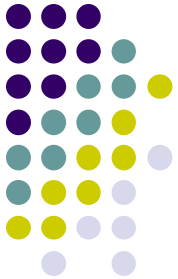


```
public class Student {
    //定义属性，和student表中的字段对应
    private int id;           //id===>s_id
    private String name;    //name===>s_name
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "Student [id=" + id + ", name=" + name + "]";
    }
}
```



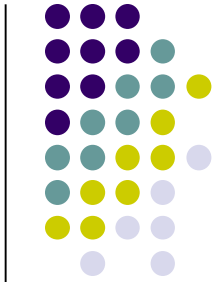
```
public class Classes {
    //定义实体类的属性，与class表中的字段对应
    private int id;          //id===>c_id
    private String name;    //name===>c_name
    /**class表中有一个teacher_id字段，所以在Classes类中定义一个teacher属性，
     * 用于维护teacher和class之间的一对一关系，通过这个teacher属性就可以知道这个班级是由哪个老师负责的
     */
    private Teacher teacher;
    //使用一个List<Student>集合属性表示班级拥有的学生
    private List<Student> students;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Teacher getTeacher() {
        return teacher;
    }
    public void setTeacher(Teacher teacher)
{
        this.teacher = teacher;
    }

    public List<Student> getStudents() {
        return students;
    }

    public void setStudents(List<Student>
students) {
        this.students = students;
    }
```

修改sql映射文件classMapper.xml

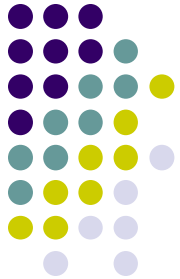


```
<!--
    根据classId查询对应的班级信息,包括学生,老师
 -->
<!--
方式一: 嵌套结果: 使用嵌套结果映射来处理重复的联合结果的子集
SELECT * FROM class c, teacher t,student s WHERE c.teacher_id=t.t_id AND c.C_id=s.class_id AND  c.c_id=1
 -->
<select id="getClass3" parameterType="int" resultMap="ClassResultMap3">
    select * from class c, teacher t,student s where c.teacher_id=t.t_id and c.C_id=s.class_id and  c.c_id=#{id}
</select>
<resultMap type="ssm.test.Classes" id="ClassResultMap3">
    <id property="id" column="c_id"/>
    <result property="name" column="c_name"/>
    <association property="teacher" column="teacher_id" javaType="ssm.test.Teacher">
        <id property="id" column="t_id"/>
        <result property="name" column="t_name"/>
    </association>
    <!-- ofType指定students集合中的对象类型 -->
    <collection property="students" ofType="ssm.test.Student">
        <id property="id" column="s_id"/>
        <result property="name" column="s_name"/>
    </collection>
</resultMap>
```

```
<!--
    方式二：嵌套查询：通过执行另外一个SQL映射语句来返回预期的复杂类型
      SELECT * FROM class WHERE c_id=1;
      SELECT * FROM teacher WHERE t_id=1   //1 是上一个查询得到的teacher_id的值
      SELECT * FROM student WHERE class_id=1  //1是第一个查询得到的c_id字段的值
  -->
  <select id="getClass4" parameterType="int" resultMap="ClassResultMap4">
    select * from class where c_id=#{id}
  </select>
  <resultMap type="ssm.test.Classes" id="ClassResultMap4">
    <id property="id" column="c_id"/>
    <result property="name" column="c_name"/>
    <association property="teacher" column="teacher_id" javaType="ssm.test.Teacher" select="getTeacher2"></association>
    <collection property="students" ofType="ssm.test.Student" column="c_id" select="getStudent"></collection>
  </resultMap>
  <select id="getTeacher2" parameterType="int" resultType="ssm.test.Teacher">
    SELECT t_id id, t_name name FROM teacher WHERE t_id=#{id}
  </select>

  <select id="getStudent" parameterType="int" resultType="ssm.test.Student">
    SELECT s_id id, s_name name FROM student WHERE class_id=#{id}
  </select>
```

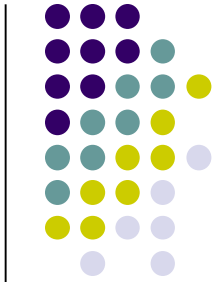


```
public class Test4 {

    @Test
    public void testGetClass3(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        /**
         * 映射sql的标识字符串，
         * me.gacl.mapping.classMapper是classMapper.xml文件中mapper标签的namespace属性的值，
         * getClass3是select标签的id属性值，通过select标签的id属性值就可以找到要执行的SQL
         */
        String statement = "ssm.test.getClass3";//映射sql的标识字符串
        //执行查询操作，将查询结果自动封装成Classes对象返回
        Classes clazz = sqlSession.selectOne(statement,1);//查询class表中id为1的记录
        //使用SqlSession执行完SQL之后需要关闭SqlSession
        sqlSession.close();
        System.out.println(clazz);
    }

}
```



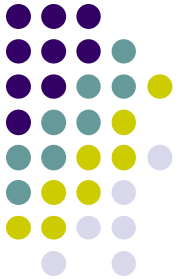
```
@Test
    public void testGetClass4(){
        SqlSession sqlSession = MyBatisUtil.getSqlSession();
        /**
         * 映射sql的标识字符串，
         * me.gacl.mapping.classMapper是classMapper.xml文件中mapper标签的namespace属性的值，
         * getClass4是select标签的id属性值，通过select标签的id属性值就可以找到要执行的SQL
         */
        String statement = "ssm.test.getClass4";//映射sql的标识字符串
        //执行查询操作，将查询结果自动封装成Classes对象返回
        Classes clazz = sqlSession.selectOne(statement,1);//查询class表中id为1的记录
        //使用SqlSession执行完SQL之后需要关闭SqlSession
        sqlSession.close();
        System.out.println(clazz);
    }
```

MyBatis一对多关联查询总结



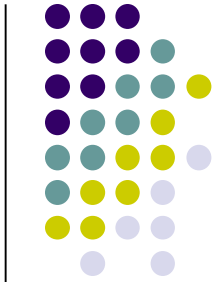
- MyBatis中使用collection标签来解决一对多的关联查询，
- ofType属性指定集合中元素的对象类型。

动态SQL



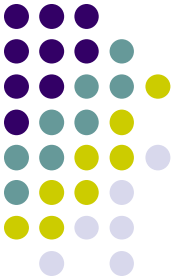
- 传统JDBC拼接SQL的困境：
- 繁琐易错： 需要手动拼接字符串，注意空格和逗号。
- 可读性差： 大量的 if 判断和字符串连接 (+ 或 StringBuilder)。
- 难以维护： SQL与Java代码混杂，修改困难。
- 引例： 一个不确定条件数量的用户查询系统。
- 动态SQL的解决方案： 通过特定的XML标签，根据参数动态地组装SQL语句，使SQL更灵活、更易维护

动态SQL



- MyBatis 的一个强大的特性之一通常是它的动态 SQL 能力
  - 条件判断： if, choose/when/otherwise
  - 语句修饰： where, set, trim
  - 循环遍历： foreach
  - 其他工具： bind, sql/include

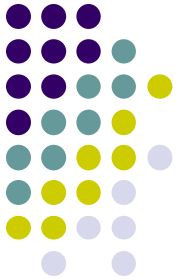
动态SQL：if语句—有条件的包含where子句部分



动态 SQL 通常会做的事情是有条件地包含 where 子句的一部分

- 作用： 根据条件判断，动态包含SQL片段
- <select id="findActiveBlogWithTitleLike" resultType="Blog">
- SELECT * FROM BLOG
- WHERE state = 'ACTIVE'
- <if test="title != null">
- AND title like #{title}
- </if>
- </select>
- 如果 title 参数不为null，则追加 AND title like ... 条件

if 的陷阱与 where 标签的拯救



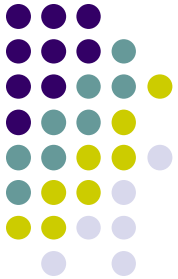
- 问题场景： 多个 if 条件组合时，可能产生 WHERE AND ... 或 WHERE 后无条件的错误SQL。

- 错误SQL示例：

- SELECT * FROM BLOG WHERE AND title like ?

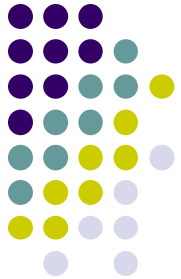
- SELECT * FROM BLOG WHERE

where 标签的解决方案



- 自动插入 WHERE 关键字：只有在至少一个子元素返回内容时才插入。

- 智能去除前缀：会自动去除子句开头多余的 AND 或 OR。

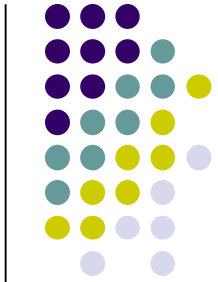


```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
       state = #{state}
    </if>
    <if test="title != null">
      AND title like #{title}
    </if>
  </where>
</select>
```

choose, when, otherwise - 多路选择

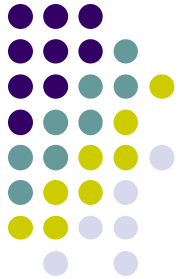
- 作用： 从多个条件中选择一个执行，类似Java中的 switch 语句。

- 逻辑： 按顺序判断 when 的 test 条件，一旦某个 when 成立，则输出其内容，并且 choose 结束。如果所有 when 都不成立，则输出 otherwise 的内容。

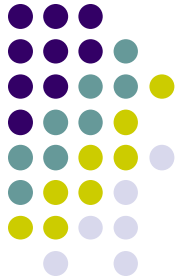


```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

set 标签 - 动态更新

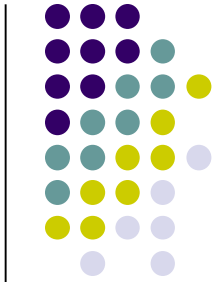


- 作用： 用于动态生成 **UPDATE** 语句中的 **SET** 部分。
- 功能：
- 自动添加 **SET** 关键字。
- 智能去除结尾多余的逗号，避免 **SET column1=?, column2=?,** 这样的语法错误



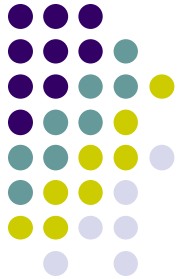
```
<update id="updateAuthorIfNecessary">
  update Author
  <set>
    <if test="username != null">username=#{username},</if>
    <if test="password != null">password=#{password},</if>
    <if test="email != null">email=#{email},</if>
  </set>
  where id=#{id}
</update>
```

foreach 标签 - 遍历集合



- 作用： 遍历一个集合（如List, Set, Map, 数组），生成重复的SQL片段，常用于 IN 条件。

- 核心属性：
- collection：指定要迭代的集合参数名。
- item：本次迭代获取的元素别名。
- index：当前迭代的序号或Map的key。
- open：整个循环内容开始时的字符串。
- close：整个循环内容结束时的字符串。
- separator：每次迭代之间的分隔符。

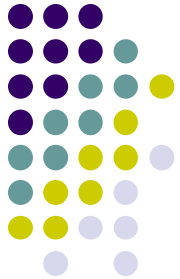


- <select id="selectPostIn" resultType="domain.blog.Post">
- SELECT * FROM POST P
- <where>
- <foreach item="item" index="index" collection="list"
- open="ID in (" separator="," close=")" nullable="true">
- #{item}
- </foreach>
- </where>
- </select>
- SELECT * FROM POST P WHERE ID in (1, 2, 3)

trim 标签详解 - 高度自定义的SQL修剪

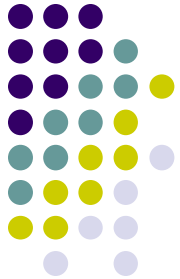
- trim 标签的作用： 对标签体内的SQL内容进行"修剪"，通过配置来自定义添加前缀、后缀，以及去除不必要的首部或尾部字符串。

- 设计思想： where 和 set 标签实际上是 trim 标签的两种特定实现。理解了 trim，你就掌握了动态SQL字符串处理的底层原理



- 属性 描述　　示例值
- prefix　　在整个修剪后的内容前添加指定的前缀　WHERE, SET
- suffix　　在整个修剪后的内容后添加指定的后缀　)
- prefixOverrides　　忽略（去除）内容开头指定的字符（可多个，用` `分隔）`AND　OR`
- suffixOverrides　　忽略（去除）内容结尾指定的字符（可多个，用` `分隔），

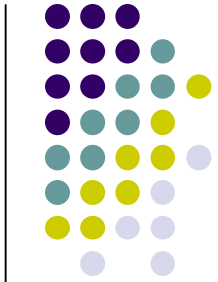
用 trim 实现 where 标签的功能



- 多条件用户查询，需要智能处理 WHERE 关键字和开头的 AND/OR
- 传统方法

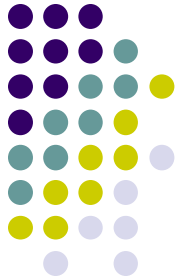
```
<select id="findUser" resultType="User">
    SELECT * FROM users
    <where>
        <if test="name != null">AND name = #{name}</if>
        <if test="age != null">AND age = #{age}</if>
        <if test="email != null">AND email = #{email}</if>
    </where>
</select>
```

使用 trim 标签的等价实现



- <select id="findUser" resultType="User">
- SELECT * FROM users
- <trim prefix="WHERE" prefixOverrides="AND |OR ">
- <if test="name != null">AND name = #{name}</if>
- <if test="age != null">AND age = #{age}</if>
- <if test="email != null">AND email = #{email}</if>
- </trim>
- </select>
- 执行过程解析：
- 首先，trim 标签内的所有 if 条件被评估，生成类似 AND name = ? AND age = ? 的SQL片段
- prefixOverrides="AND |OR "：从开头去除 AND 或 OR（注意后面的空格）
- prefix="WHERE"：在修剪后的内容前添加 WHERE 关键字
- 最终生成：SELECT * FROM users WHERE name = ? AND age = ?

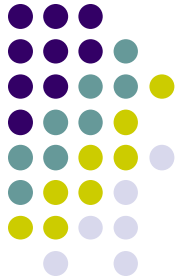
用 trim 实现 set 标签的功能



- 场景： 动态更新用户信息，只更新非空字段，需要处理结尾的逗号

- 使用 set 标签的写法（回顾）：
- <update id="updateUser">
- UPDATE users
- <set>
- <if test="name != null">name = #{name},</if>
- <if test="age != null">age = #{age},</if>
- <if test="email != null">email = #{email},</if>
- </set>
- WHERE id = #{id}
- </update>

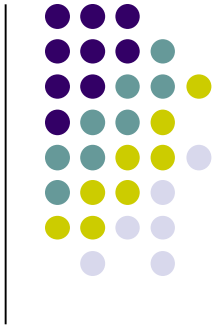
使用 trim 标签的等价实现



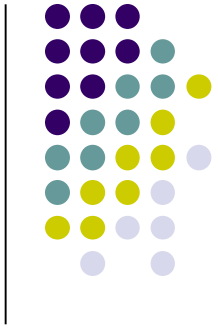
```
<update id="updateUser">
    UPDATE users
    <trim prefix="SET" suffixOverrides=",">
        <if test="name != null">name = #{name},</if>
        <if test="age != null">age = #{age},</if>
        <if test="email != null">email = #{email},</if>
    </trim>
    WHERE id = #{id}
</update>
```

举例：Book案例

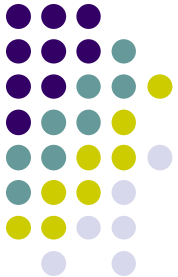
```
CREATE TABLE book
( id BIGINT PRIMARY KEY AUTO_INCREMENT, title VARCHAR(200) NOT NULL,
author VARCHAR(100) NOT NULL, isbn VARCHAR(20) UNIQUE, price DECIMAL(10,2),
 category VARCHAR(50), publisher VARCHAR(100), publish_date DATE, stock_quantity INT DEFAULT 0,
 status TINYINT DEFAULT 1 COMMENT '1:可用, 0:不可用', create_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
update_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP );
 INSERT INTO book (title, author, isbn, price, category, publisher, publish_date, stock_quantity) VALUES
('Java编程思想', 'Bruce Eckel', '9787111213826', 108.00, '编程', '机械工业出版社', '2007-06-01', 50),
('Spring实战', 'Craig Walls', '9787115528322', 89.00, '编程', '人民邮电出版社', '2020-01-01', 30),
 ('MyBatis从入门到精通', '刘增辉', '9787121346235', 79.00, '编程', '电子工业出版社', '2018-07-01', 25),
 ('数据库系统概念', 'Abraham Silberschatz', '9787111615421', 99.00, '数据库', '机械工业出版社', '2019-03-01', 20),
 ('算法导论', 'Thomas H. Cormen', '9787111407010', 128.00, '算法', '机械工业出版社', '2012-12-01', 15);
```



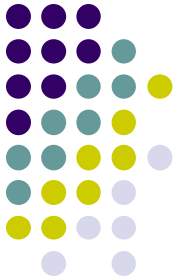
```
    // 构造方法
    public Book() {}

    public Book(String title, String author, String isbn, BigDecimal price,
                String category, String publisher, LocalDate publishDate,
                Integer stockQuantity) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
        this.price = price;
        this.category = category;
        this.publisher = publisher;
        this.publishDate = publishDate;
        this.stockQuantity = stockQuantity;
    }

    // Getter和Setter方法
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }

    public String getIsbn() { return isbn; }
    public void setIsbn(String isbn) { this.isbn = isbn; }

    public BigDecimal getPrice() { return price; }
    public void setPrice(BigDecimal price) { this.price = price; }

    public String getCategory() { return category; }
    public void setCategory(String category) { this.category = category; }

    public String getPublisher() { return publisher; }
    public void setPublisher(String publisher) { this.publisher = publisher; }

    public LocalDate getPublishDate() { return publishDate; }
    public void setPublishDate(LocalDate publishDate) { this.publishDate = publishDate; }

    public Integer getStockQuantity() { return stockQuantity; }
    public void setStockQuantity(Integer stockQuantity) { this.stockQuantity = stockQuantity; }

    public Integer getStatus() { return status; }
```



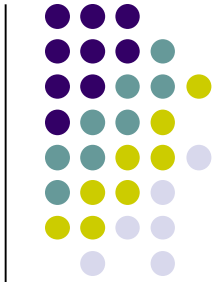
```java
public interface BookMapper {
    // 1. 插入书籍
    int insert(Book book);
    // 2. if 标签 - 条件查询
    List<Book> selectByCondition(Book book);
    // 3. choose/when/otherwise 标签 - 多条件选择
    List<Book> selectByChoose(Book book);
    // 4. where 标签 - 自动处理WHERE条件
    List<Book> selectByWhere(Book book);
    // 5. set 标签 - 动态更新
    int updateBySet(Book book);
    // 6. trim 标签 - 自定义前缀后缀
    List<Book> selectByTrim(Book book);
    int updateByTrim(Book book);
    // 7. foreach 标签 - 批量操作
    List<Book> selectByIds(@Param("ids") List<Long> ids);
    int batchInsert(@Param("books") List<Book> books);
    int batchDelete(@Param("ids") List<Long> ids);
    int batchUpdatePrice(@Param("ids") List<Long> ids, @Param("price") BigDecimal price);
    // 8. bind 标签 - 创建变量
    List<Book> selectByTitleLike(@Param("title") String title);
    // 9. 复杂条件查询
    List<Book> selectByComplexCondition(Map<String, Object> params);
    // 10. 价格范围查询
    List<Book> selectByPriceRange(@Param("minPrice") BigDecimal minPrice,
                                  @Param("maxPrice") BigDecimal maxPrice);
    // 11. 分类统计
    List<Map<String, Object>> countByCategory();
    // 12. 根据ID查询
    Book selectById(Long id);
    // 13. 删除书籍
    int deleteById(Long id);
    // 14. 更新库存
    int updateStock(@Param("id") Long id, @Param("quantity") Integer quantity);
}
```



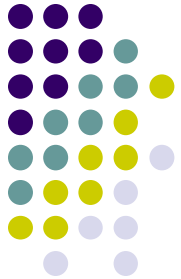
- <!-- 2. if 标签示例 - 条件查询 -->
- <select id="selectByCondition" parameterType="ssm.test.Book" resultMap="BookResultMap">
- SELECT * FROM book WHERE status = 1
- <if test="title != null and title != ""> AND title LIKE CONCAT('%', #{title}, '%') </if>
- <if test="author != null and author != ""> AND author LIKE CONCAT('%', #{author}, '%') </if>
- <if test="category != null and category != ""> AND category = #{category} </if>
- <if test="publisher != null and publisher != ""> AND publisher = #{publisher} </if>
- <if test="price != null"> AND price = #{price} </if>
- ORDER BY create_time DESC
- </select>



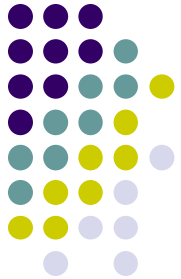
- <!-- 4. where 标签 - 自动处理WHERE条件 -->
- <select id="selectByWhere" parameterType="ssm.test.Book" resultMap="BookResultMap">
- SELECT * FROM book
- <where>
- <if test="status != null"> AND status = #{status} </if>
- <if test="title != null and title != ""> AND title LIKE CONCAT('%', #{title}, '%') </if>
- <if test="author != null and author != ""> AND author LIKE CONCAT('%', #{author}, '%') </if>
- <if test="category != null and category != ""> AND category = #{category} </if>
- </where>
- ORDER BY create_time DESC
- </select>



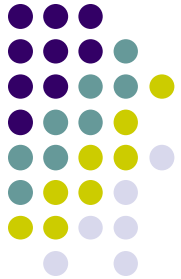
- <!-- 5. set 标签 - 动态更新 -->
- <update id="updateBySet" parameterType="ssm.test.Book">
- UPDATE book
- <set>
- <if test="title != null and title != "">title = #{title},</if>
- <if test="author != null and author != "">author = #{author},</if>
- <if test="price != null">price = #{price},</if>
- <if test="category != null and category != "">category = #{category},</if>
- <if test="publisher != null and publisher != "">publisher = #{publisher},</if>
- <if test="publishDate != null">publish_date = #{publishDate},</if>
- <if test="stockQuantity != null">stock_quantity = #{stockQuantity},</if>
- <if test="status != null">status = #{status},</if>
- </set>
- WHERE id = #{id}
- </update>



- <!-- 6. trim 标签示例 -->
- <select id="selectByTrim" parameterType="ssm.test.Book" resultMap="BookResultMap">
- SELECT * FROM book
- <trim prefix="WHERE" prefixOverrides="AND |OR ">
- <if test="status != null"> AND status = #{status} </if>
- <if test="title != null and title != ""> AND title LIKE CONCAT('%', #{title}, '%') </if>
- <if test="category != null and category != ""> AND category = #{category} </if>
- </trim>
- ORDER BY create_time DESC
- </select>

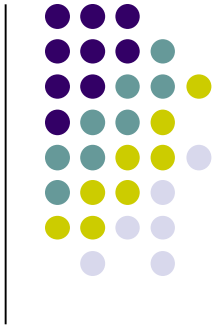


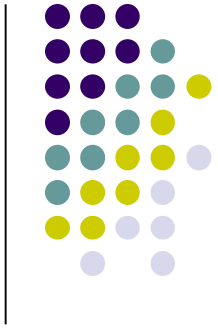
- <!-- 7. foreach 标签 - 批量查询 -->
- <select id="selectByIds" resultMap="BookResultMap">
- SELECT * FROM book WHERE id IN
- <foreach collection="ids" item="id" open="(" separator="," close=")"> #{id} </foreach>
- AND status = 1 ORDER BY create_time DESC
- </select>



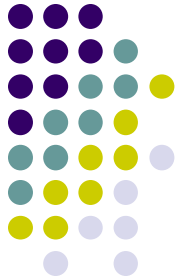
- <!-- 3. choose/when/otherwise 标签 -->
- <select id="selectByChoose" parameterType="ssm.test.Book" resultMap="BookResultMap">
- SELECT * FROM book WHERE status = 1
- <choose>
- <when test="title != null and title != ""> AND title LIKE CONCAT('%', #{title}, '%') </when>
- <when test="author != null and author != ""> AND author LIKE CONCAT('%', #{author}, '%') </when>
- <otherwise> AND stock_quantity > 0 </otherwise>
- </choose>
- ORDER BY create_time DESC
- </select>

```xml
<select id="selectByIds" resultMap="BookResultMap">
    SELECT * FROM book WHERE id IN
    <foreach collection="ids" item="id" open="(" separator="," close=")">
        #{id}
    </foreach>
    AND status = 1
    ORDER BY create_time DESC
</select>
<!-- 批量插入 -->
<insert id="batchInsert" parameterType="java.util.List">
    INSERT INTO book (title, author, isbn, price, category, publisher, publish_date, stock_quantity)
    VALUES
    <foreach collection="books" item="book" separator=",">
        (#{book.title}, #{book.author}, #{book.isbn}, #{book.price},
        #{book.category}, #{book.publisher}, #{book.publishDate}, #{book.stockQuantity})
    </foreach>
</insert>
<!-- 批量删除 -->
<delete id="batchDelete">
    DELETE FROM book WHERE id IN
    <foreach collection="ids" item="id" open="(" separator="," close=")">
        #{id}
    </foreach>
</delete>
<!-- 批量更新价格 -->
<update id="batchUpdatePrice">
    UPDATE book SET price = #{price} WHERE id IN
    <foreach collection="ids" item="id" open="(" separator="," close=")">
        #{id}
    </foreach>
</update>
<!-- 8. bind 标签 -->
<select id="selectByTitleLike" resultMap="BookResultMap">
    <bind name="pattern" value="'%' + title + '%'"/>
    SELECT * FROM book
    WHERE title LIKE #{pattern} AND status = 1
    ORDER BY create_time DESC
</select>
<!-- 9. 复杂条件查询 -->
<select id="selectByComplexCondition" parameterType="map" resultMap="BookResultMap">
    SELECT * FROM book
    <where>
        <if test="status != null">AND status = #{status}</if>
        <if test="title != null and title != ''">
            AND title LIKE CONCAT('%', #{title}, '%')
```



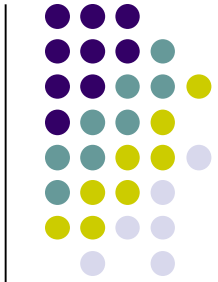


```
        Book condition = new Book();
        // condition.setTitle("Spring"); // 注释掉测试otherwise
        List<Book> books = mapper.selectByChoose(condition);
        System.out.println("=== Choose查询结果 ===");
        books.forEach(System.out::println);
    }
}

private static void testSelectByWhere() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        BookMapper mapper = session.getMapper(BookMapper.class);

        Book condition = new Book();
        condition.setAuthor("机械工业");
        condition.setStatus(1);

        List<Book> books = mapper.selectByWhere(condition);
        System.out.println("=== Where查询结果 ===");
        books.forEach(System.out::println);
    }
}

private static void testUpdateBySet() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        BookMapper mapper = session.getMapper(BookMapper.class);

        Book book = new Book();
        book.setId(1L);
        book.setPrice(new BigDecimal("115.00"));
        book.setStockQuantity(45);

        int result = mapper.updateBySet(book);
        session.commit();
        System.out.println("动态更新影响行数: " + result);
    }
}

private static void testSelectByTrim() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        BookMapper mapper = session.getMapper(BookMapper.class);

        Book condition = new Book();
        condition.setPublisher("人民邮电出版社");
```

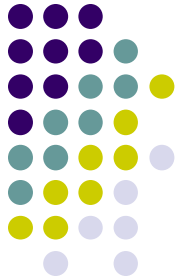


- MyBatis有两种实现方法，分别为基于注解和基于映射文件。当需要操作的实体类较多时，逐个编写基于注解或基于映射文件的CURD耗时长且容易出错，使用MyBatis Generator可以保证CRUD的正确性，以及节省大量的时间。

MyBatis Generator



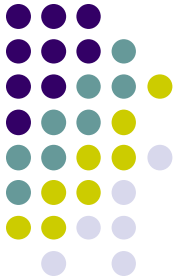
- 根据创建好的数据库表生成MyBatis的表对应的实体类，SQL映射文件和dao



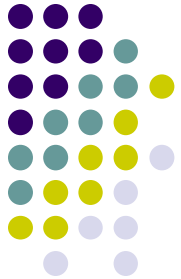
- Create DATABASE spring4_mybatis3;
- USE spring4_mybatis3;

- DROP TABLE IF EXISTS t_user;
- CREATE TABLE t_user (
- user_id char(32) NOT NULL,
- user_name varchar(30) DEFAULT NULL,
- user_birthday date DEFAULT NULL,
- user_salary double DEFAULT NULL,
- PRIMARY KEY (user_id)
-) ENGINE=InnoDB DEFAULT CHARSET=utf8;

下载mybatis-generator-core-1.3.2



- <?xml version="1.0" encoding="UTF-8"?>
- <!DOCTYPE generatorConfiguration
- PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
- "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
- <generatorConfiguration>
- <classPathEntry location="C:\Users\wyxhhwin\Downloads\mybatis-generator-core-1.3.2\mybatis-generator-core-1.3.2\lib\mysql-connector-5.1.8.jar" />
- <context id="sysGenerator" targetRuntime="MyBatis3">
- <jdbcConnection driverClass="com.mysql.jdbc.Driver"
- connectionURL="jdbc:mysql://localhost:3306/mybatis"
- userId="root" password="333333">
- </jdbcConnection>
- <javaModelGenerator targetPackage="me.gacl.domain"
- targetProject="C:\Users\wyxhhwin\Downloads\mybatis-generator-core-1.3.2\mybatis-generator-core-1.3.2\lib">
- <property name="enableSubPackages" value="true" />
- <property name="trimStrings" value="true" />
- </javaModelGenerator>
- <sqlMapGenerator targetPackage="me.gacl.mapping"
- targetProject="C:\Users\wyxhhwin\Downloads\mybatis-generator-core-1.3.2\mybatis-generator-core-1.3.2\lib">
- <property name="enableSubPackages" value="true" />



- java -jar mybatis.jar -configfile generator.xml –overwrite
- 自动生成