

第6章 MyBatis及其增强框架

广东财经大学信息学院

罗 东 俊 博士

ZSUJONE@126.COM

(内部资料，请勿外传)



目的和要求

- 掌握**MyBatis**的关联映射机制。
- 掌握**MyBatis**的动态**SQL**语法。
- 掌握**MyBatis-Plus**的常用注解和通用**CRUD**方法。
- 掌握**Spring Boot**整合**MyBatis-Plus**的使用。



主要内容

- 6.1 MyBatis框架运行机制
- 6.2 MyBatis-Plus框架的使用



6.1 MyBatis框架运行机制

- 6.1.1 MyBatis简介
- 6.1.2 MyBatis的关联映射
- 6.1.3 MyBatis的动态SQL
- 6.1.4 MyBatis的延迟加载机制
- 6.1.5 MyBatis的缓存机制

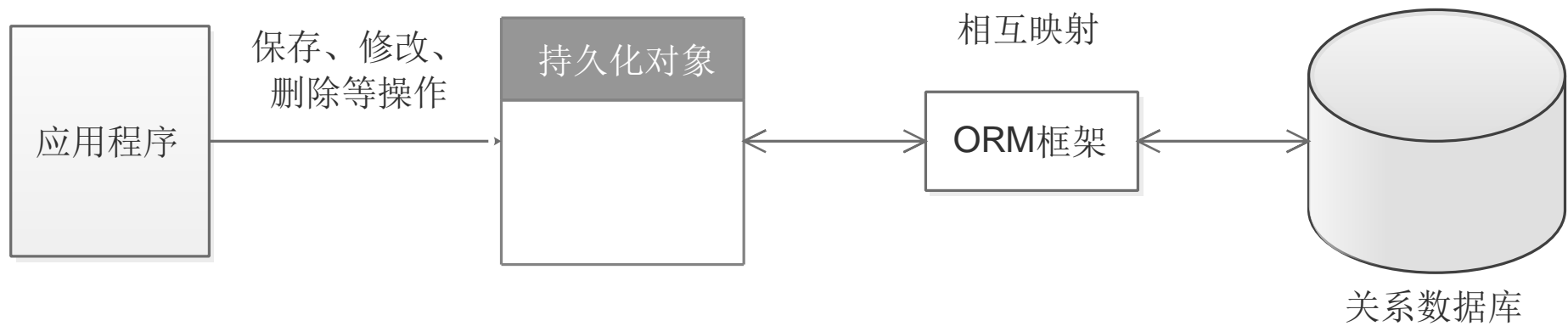


6.1.1 MyBatis简介

- **MyBatis**（前身是**iBatis**）是一个支持普通**SQL**查询、存储过程以及高级映射的持久层框架。
- **MyBatis**框架也被称之为**ORM**（**Object/Relation Mapping**，即对象关系映射）框架。所谓的**ORM**就是一种为了解决面向对象与关系型数据库中数据类型不匹配的技术，它通过描述**Java**对象与数据库表之间的映射关系，自动将**Java**应用程序中的对象持久化到关系型数据库的表中。



ORM框架的工作原理



Hibernate与MyBatis区别

Hibernate

- **Hibernate**是一个全表映射的框架。
- 通常开发者只需定义好持久化对象到数据库表的映射关系，就可以通过**Hibernate**提供的方法完成持久层操作。
- 开发者并不需要熟练的掌握SQL语句的编写，**Hibernate**会根据制定的存储逻辑，自动的生成对应的SQL，并调用JDBC接口来执行，所以其开发效率会高于**MyBatis**。
- **Hibernate**也存在一些缺点，例如它在多表关联时，对SQL查询的支持较差；更新数据时，需要发送所有字段；不支持存储过程；不能通过优化SQL来优化性能等。

MyBatis

- **MyBatis**是一个半自动映射的框架。
- “半自动”是相对于**Hibernate**全表映射而言的，**MyBatis**需要手动匹配提供POJO、SQL和映射关系，而**Hibernate**只需提供POJO和映射关系即可。
- 与**Hibernate**相比，虽然使用**MyBatis**手动编写SQL要比使用**Hibernate**的工作量大，但**MyBatis**可以配置动态SQL并优化SQL，可以通过配置决定SQL的映射规则，它还支持存储过程等。对于一些复杂的和需要优化性能的项目来说，显然使用**MyBatis**更加合适。

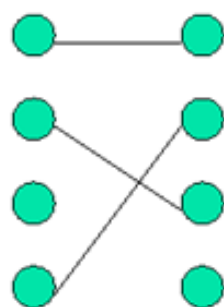
6.1.2 MyBatis的关联映射

- 1. 关联关系概述
- 2. MyBatis的关联映射机制
- 3. MyBatis的映射文件

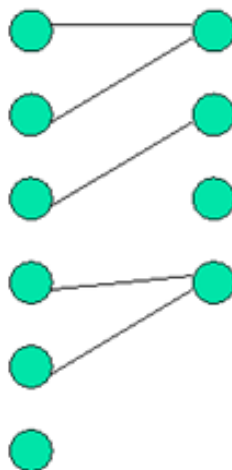


1. 关联关系概述

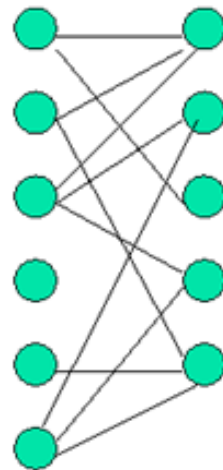
■ 现实生活中实体间存在着三种关联关系，分别为一对一、一对多和多对多，如图所示：



一对一
1: 1



一对多
1: N

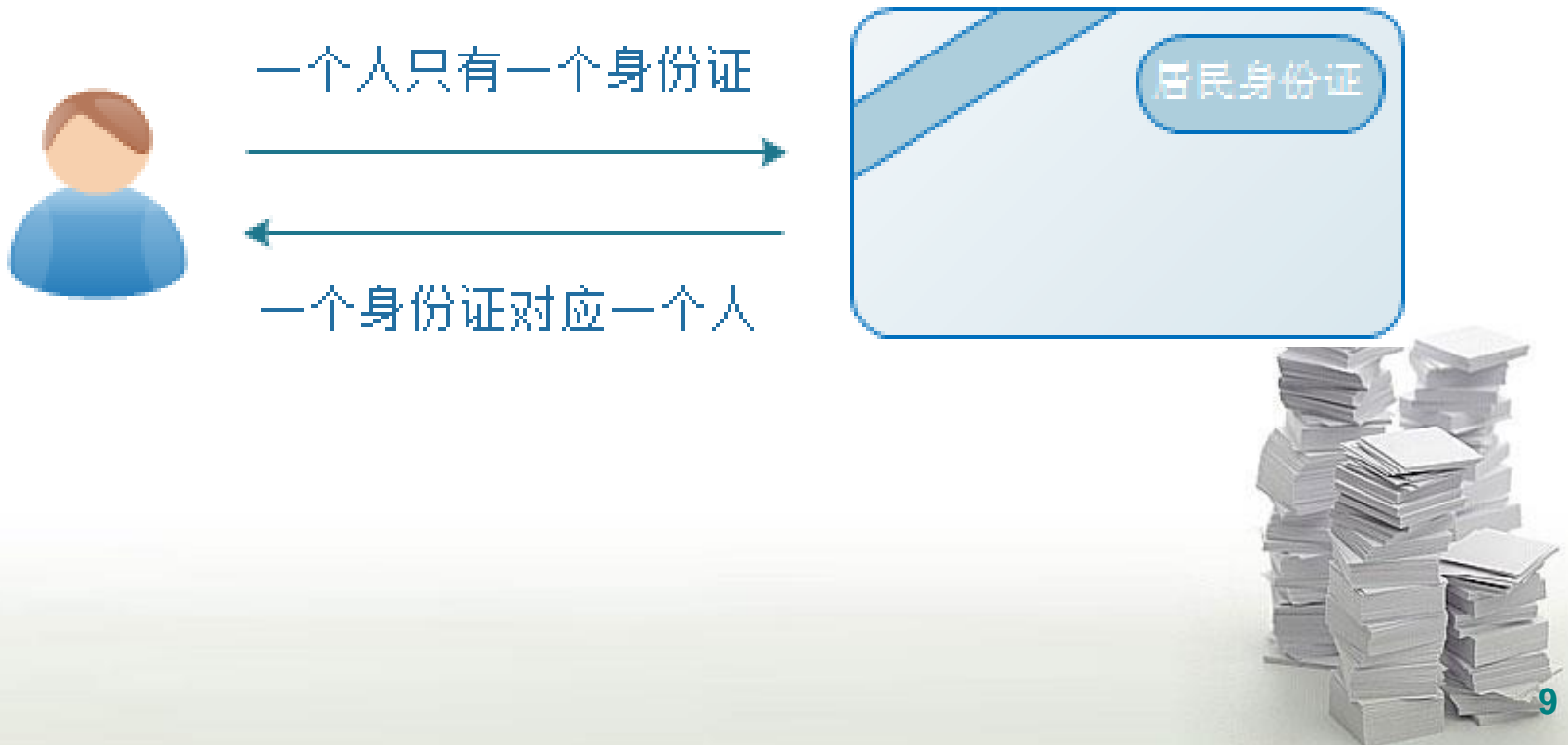


多对多
M: N



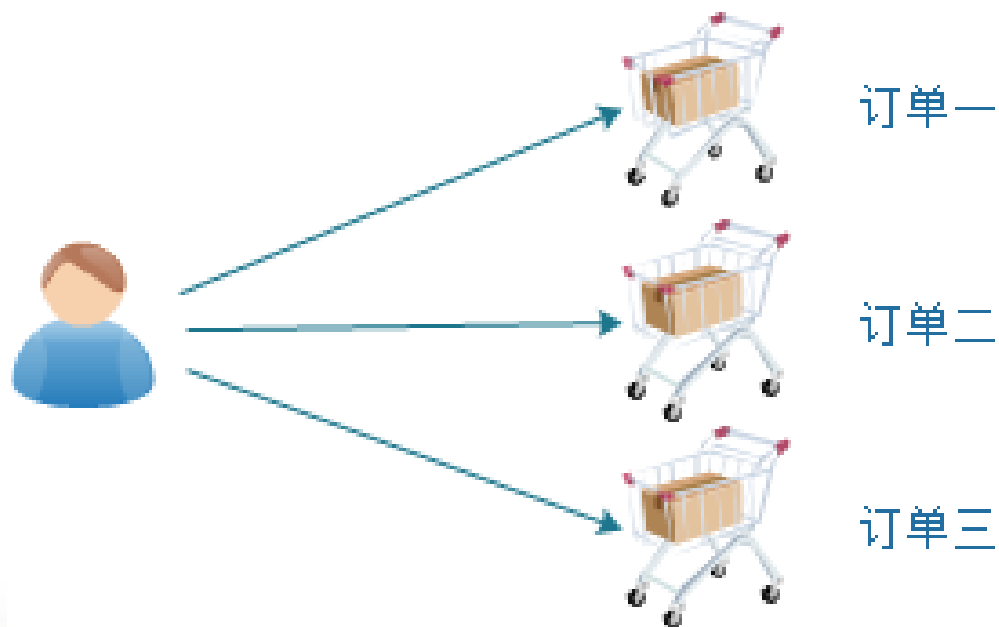
一对一示例

- 例如，一个人只能有一个身份证，同时一个身份证也只会对应一个人。



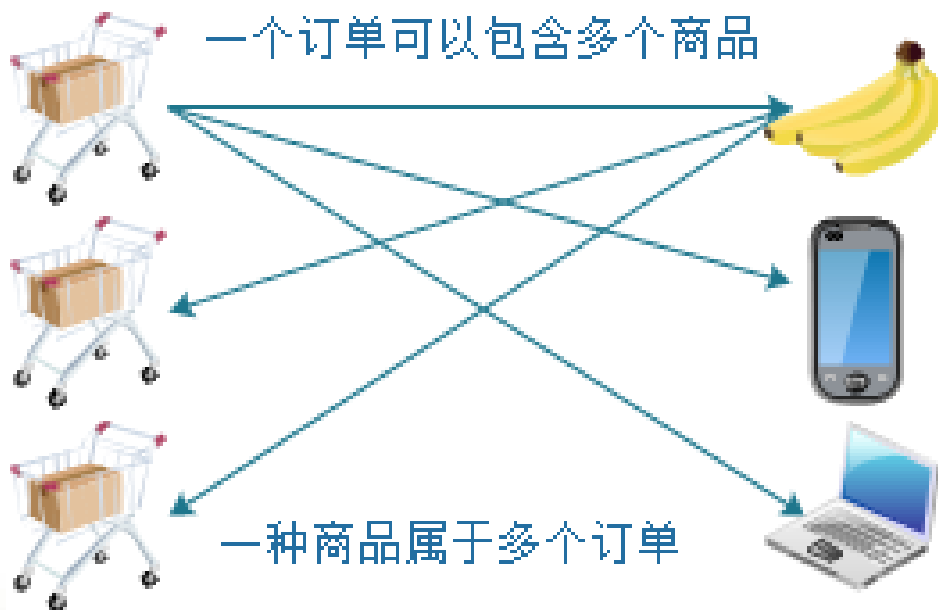
一对多示例

■ 例如，一个用户可以有多个订单，同时一个订单只归一个用户所有。



多对多示例

- 以订单和商品为例，一个订单可以包含多种商品，而一种商品又可以属于多个订单。



关联关系的描述

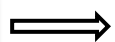
- ①关系数据库中的描述方法
- ②Java中的描述方法



①关系数据库中的描述方法

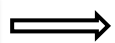
■在关系数据库中，描述多表间的关联关系方法如下：

一对一



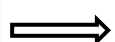
在任意一方引入对方主键作为外键；

一对多



在“多”的一方，添加“一”的一方的主键作为外键；

多对多



产生中间关系表，引入两张表的主键作为外键，两个主键成为联合主键或使用新的字段作为主键。

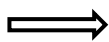


②Java中的描述方法

■在Java中，描述对象间的关联关系方法如下图所示：

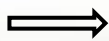
<pre>class A{ B b; } class B{ A a; }</pre>	<pre>class A{ List b; } class B{ A a; }</pre>	<pre>class A{ List b; } class B{ List<A> a; }</pre>
一对一	一对多	多对多

一对一



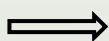
在本类中定义对方类型的对象，如A类中定义B类类型的属性b，B类中定义A类类型的属性a；

一对多



一个A类类型对应多个B类类型的情况，需要在A类中以集合的方式引入B类类型的对象，在B类中定义A类类型的属性a；

多对多



在A类中定义B类类型的集合，在B类中定义A类类型的集合。

2. MyBatis的关联映射机制

■ **MyBatis**的关联映射机制就是解决关联关系在关系数据库和**Java**中两种不同描述方法间的映射，包括：

- ✧①一对一关联的映射机制
- ✧②一对多关联的映射机制
- ✧③多对多关联的映射机制



①一对一关联的映射机制

Person类

```
private Integer id;  
private String name;  
private Integer age;  
private String sex;  
private IdCard card;
```

IdCard类

```
private Integer id;  
private String code;
```

tb_person表

```
id int(32) primary key auto_increment,  
name varchar(32),  
age int,  
sex varchar(8)  
card_id int unique,  
foreign key(card_id) references tb_idcard(id)
```

tb_idcard表

```
id int(32) primary key auto_increment,  
code varchar(18)
```



一对一关联的映射机制

PersonMapper:

关联属性类型

```
<association property="card" javaType="IdCard">  
  <id property="id" column="card_id" />  
  <result property="code" column="code" />  
</association>
```

类属性

表字段(或别名)



②一对多关联的映射机制

User类

```
private Integer id;           // 用户编号
private String username;      // 用户姓名
private String address;       // 用户地址
private List<Order> orderList; // 用户关联的订单
```

Order类

```
private Integer id; // 订单id
private String number; // 订单编号
// 关联商品集合信息
private List<Product> productList;
```

tb_user表

```
id int(32) primary key auto_increment,
username varchar(32),
address varchar(256)
```

tb_order表

```
id int(32) primary key auto_increment,
number varchar(32) not null,
user_id int(32) not null,
foreign key(user_id) reference tb_user(id)
```

一对多关联的映射机制

UserMapper:

关联的集合类属性类型

```
<collection property="orderList" ofType="Order">  
  <id property="id" column="order_id" />  
  <result property="number" column="number" />  
</collection>
```

类属性

表字段(或别名)



③多对多关联的映射机制

Product类

```
private Integer id;           // 商品编号
private String name;         // 商品名称
private Double price;        // 商品单价
private List<Order> orderList; //商品关联的订单
```

tb_product表

```
id int(32) primary key auto_increment,
name varchar(32),
price double
```

tb_order表

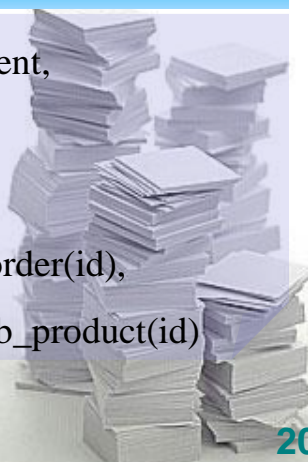
```
id int(32) primary key auto_increment,
number varchar(32) not null,
user_id int(32) not null,
foreign key(user_id) reference tb_user(id)
```

Order类

```
private Integer id; //订单id
private String number; //订单编号
//关联商品集合信息
private List<Product> productList;
```

tb_orderitem中间表

```
id int(32) primary key auto_increment,
product_id int(32) not null,
order_id int(32) not null,
foreign key(oder_id) reference tb_order(id),
foreign key(product_id) reference tb_product(id)
```



多对多关联的映射机制

ProductMapper:

关联的集合类属性类型

```
<collection property="orderList" ofType="Order">  
  <id property="id" column="order_id" />  
  <result property="number" column="number" />  
</collection>
```

类属性

表字段(或别名)

关联的集合类属性类型

OrderMapper:

```
<collection property="productList" ofType="Product">  
  <id property="id" column="product_id" />  
  <result property="name" column="name" />  
  <result property="price" column="price" />  
</collection>
```

类属性

表字段(或别名)

3. MyBatis的映射文件

■ MyBatis的ORM功能是通过映射文件（Mapper）来实现的。

✧如：在映射文件中定义了关联关系映射、选择语句映射、插入语句映射、更新语句映射、删除语句映射等。

■ 在映射文件中，**<mapper>**元素是映射文件的根元素，其他元素都是它的子元素。



MyBatis映射文件的主要元素



IdCardMapper.xml映射文件

```
<?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">
<!--namespace为Mapper接口的类路径-->
<mapper namespace="com.itheima.mapper.IdCardMapper">
    <!-- 对应 “public IdCard findCodeById(Integer id)”： 根据id查询证件信息-->
    <select id="findCodeById" parameterType="Integer" resultType="IdCard">
        SELECT * from tb_idcard where id=#{id}
    </select>
</mapper>
```



PersonMapper.xml映射文件

```
<?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.itheima.mapper.PersonMapper">
    <!-- 对应 “public Person findPersonById(Integer id)”: 根据id查询个人信息-->
    <select id="findPersonById" parameterType="Integer" resultMap="PersonWithIdCard">
        SELECT p.*,c.id card_id,c.code
        from tb_person p,tb_idcard c
        where p.card_id=c.id and p.id= #{id}
    </select>
    <!--结果映射集-->
    <resultMap id="PersonWithIdCard" type="Person" >
        <id property="id" column="id" />
        <result property="name" column="name" />
        <result property="age" column="age" />
        <result property="sex" column="sex" />
        <association property="card" javaType="IdCard">
            <id property="id" column="card_id" />
            <result property="code" column="code" />
        </association>
    </resultMap>
</mapper>
```


UserMapper.xml映射文件

```
<?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.itheima.mapper.UserMapper">
    <!-- 对应 “public User findUserById(Integer id)”：根据id查询用户信息-->
    <select id="findUserById" parameterType="Integer" resultMap="UserWithOrders">
        SELECT u.*,o.id order_id,o.number
        from tb_user u,tb_order o
        WHERE u.id=o.user_id and u.id=#{id}
    </select>
    <!--结果映射集-->
    <resultMap id="UserWithOrders" type="User">
        <id property="id" column="id"/>
        <result property="username" column="username"/>
        <result property="address" column="address"/>
        <collection property="orderList" ofType="Order">
            <id property="id" column="order_id"/>
            <result property="number" column="number"/>
        </collection>
    </resultMap>
</mapper>
```

OrderMapper.xml映射文件

```
<?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.itheima.mapper.OrderMapper">
    <!-- 对应 “public Order findOrderById(Integer id)”：根据id查询订单信息-->
    <select id="findOrderById" parameterType="Integer" resultMap="OrderWithProducts">
        select o.*,p.id product_id,p.name,p.price
        from tb_order o,tb_product p,tb_orderitem oi
        WHERE oi.order_id=o.id and oi.product_id=p.id and o.id=#{id}
    </select>
    <!--结果映射集-->
    <resultMap id="OrderWithProducts" type="Order" >
        <id property="id" column="id" />
        <result property="number" column="number" />
        <collection property="productList" ofType="Product">
            <id property="id" column="product_id" />
            <result property="name" column="name" />
            <result property="price" column="price" />
        </collection>
    </resultMap>
</mapper>
```

6.1.3 MyBatis的动态SQL

- MyBatis提供对SQL语句动态组装的功能。
- 动态SQL主要元素如下表所示：

元素 [↵]	说明 [↵]
<if> [↵]	判断语句，用于单条件分支判断 [↵]
<choose> (<when>、<otherwise>) [↵]	相当于 Java 中的 switch...case...default 语句，用于多条件分支判断 [↵]
<where>、<trim>、<set> [↵]	辅助元素，用于处理一些 SQL 拼装、特殊字符问题 [↵]
<foreach> [↵]	循环语句，常用于 in 语句等列举条件中 [↵]
<bind> [↵]	从 OGNL 表达式中创建一个变量，并将其绑定到上下文，常用于模糊查询的 sql 中 [↵]

1.<if>元素

```
select * from t_customer where 1=1
```

```
<if test="username !=null and username !="">  
    and username like concat('%',{username}, '%')
```

```
</if>
```

```
<if test="jobs !=null and jobs !="">
```

```
    and jobs= #{jobs}
```

```
</if>
```

使用<if>元素
对username和
jobs进行非空
判断, 并动态
组装SQL



2.<choose>元素

```
select * from t_customer where 1=1
```

```
<choose>
```

```
  <when test="username !=null and username !="">  
    and username like concat('%',{username}, '%')
```

```
  </when>
```

```
  <when test="jobs !=null and jobs !="">  
    and jobs= #{jobs}
```

```
  </when>
```

```
  <otherwise>  
    and phone is not null
```

```
  </otherwise>
```

```
</choose>
```

使用<choose>
及其子元素依次
对条件进行非空
判断, 并动态组
装SQL

3. <where>、<trim>元素

■ <where>、<trim>这两个元素是“where 1=1”条件的另外两种等效表达方式。

✧ <where>会自动判断SQL语句，只有<where>内的条件成立时，才会在拼接SQL中加入where关键字，且去除多余的“AND”或“OR”。

✧ <trim>的作用是去除特殊的字符串，它的prefix属性代表语句的前缀，prefixOverrides属性代表需要去除的哪些特殊字符串。



使用<where>元素

```
select * from t_customer
```

```
<where>
```

```
  <if test="username !=null and username !="">  
    and username like concat('%',{username}, '%')
```

```
  </if>
```

```
  <if test="jobs !=null and jobs !="">  
    and jobs= #{jobs}
```

```
  </if>
```

```
</where>
```



使用<trim>元素

```
select * from t_customer
<trim prefix="where" prefixOverrides="and">
  <if test="username !=null and username !="">
    and username like concat('%',{username}, '%')
  </if>
  <if test="jobs !=null and jobs !="">
    and jobs= #{jobs}
  </if>
</trim>
```



4. <set>元素

```
<update id="updateCustomer" parameterType="Customer">
  update t_customer
  <set>
    <if test="username !=null and username !="">
      username=#{username},
    </if>
    <if test="jobs !=null and jobs !="">
      jobs=#{jobs},
    </if>
  </set>
  where id=#{id}
</update>
```

使用<set>和<if>元素对username和jobs进行更新判断，并动态组装SQL。这样想更新一条记录就只需要传入想要更新的字段即可。



5. <foreach>元素

<!-- 对应 “public List<Customer> findCustomerByIds(List ids)”：根据id列表查询列表中每客户信息-->

```
<select id="findCustomerByIds" parameterType="List" resultType="Customer">
    select * from t_customer where id in
        <foreach item="id" index="index" collection="list" open="(" separator="," close=")">
            #{id}
        </foreach>
</select>
```



<foreach>元素中使用的几种属性

- **item**: 配置的是循环中当前的元素。
- **index**: 配置的是当前元素在集合的位置下标。
- **collection**: 配置的“**list**”是传递过来的参数类型（首字母小写），它可以是一个**array**、**list**（或**collection**）、**Map**集合的键、**POJO**包装类中数组或集合类型的属性名等。
- **open**和**close**: 配置的是以什么符号将这些集合元素包装起来。
- **separator**: 配置的是各个元素的间隔符。



6.<bind>元素

<!-- 对应 “public Customer findCustomerByLikeName(String likename)”：根据客户名模糊查询客户信息-->

```
<select id="findCustomerByLikeName" parameterType="String" resultType="Customer">
    <bind name="pattern_name" value="'%'+_parameter+'%' />
    select * from t_customer
    where username like #{pattern_name}
</select>
```

绑定后，需要的地方直接引用<bind>元素的name属性值即可，SQL语句即可适用于各类数据库

parameter表示传递进来的字符串类型参数

6.1.4 MyBatis的延迟加载机制

■ **MyBatis**加载关联关系对象主要通过两种方式：**嵌套查询加载**和**嵌套结果加载**。

第一种

嵌套查询是通过执行另外一条SQL映射语句来返回预期的复杂类型。

嵌套查询是在查询SQL中嵌入一个子查询SQL；

嵌套查询会执行多条SQL语句；

嵌套查询SQL语句编写较为简单；

第二种

嵌套结果是使用嵌套结果映射来处理重复的联合结果的子集。

嵌套结果是一个嵌套的多表查询SQL；

嵌套结果只会执行一条复杂的SQL语句；

嵌套结果SQL语句编写比较复杂；

例子

类属性

表字段

方法1：嵌套查询

```
<association property="card" column="card_id"
              javaType="com.itheima.po.IdCard"
              select="com.itheima.mapper.IdCardMapper.findCodeById" />
```

嵌套的子查询

关联属性类型

方法2：嵌套结果

```
<association property="card" javaType="com.itheima.po.IdCard">
  <id property="id" column="card_id" />
  <result property="code" column="code" />
</association>
```

类属性

表字段



嵌套查询加载的缺陷

- 虽然使用嵌套查询的方式比较简单，但是嵌套查询的方式要执行多条**SQL**语句，这对于大型数据集合和列表展示不是很好，因为这样可能会导致成百上千条关联的**SQL**语句被执行，从而**极大的消耗数据库性能并且会降低查询效率**。
- 解决办法：延迟加载



MyBatis延迟加载的配置

- 使用**MyBatis**的延迟加载在一定程度上可以降低运行消耗并提高查询效率。
- **MyBatis**默认没有开启延迟加载，需要在属性配置文件中配置，具体配置方式如下：

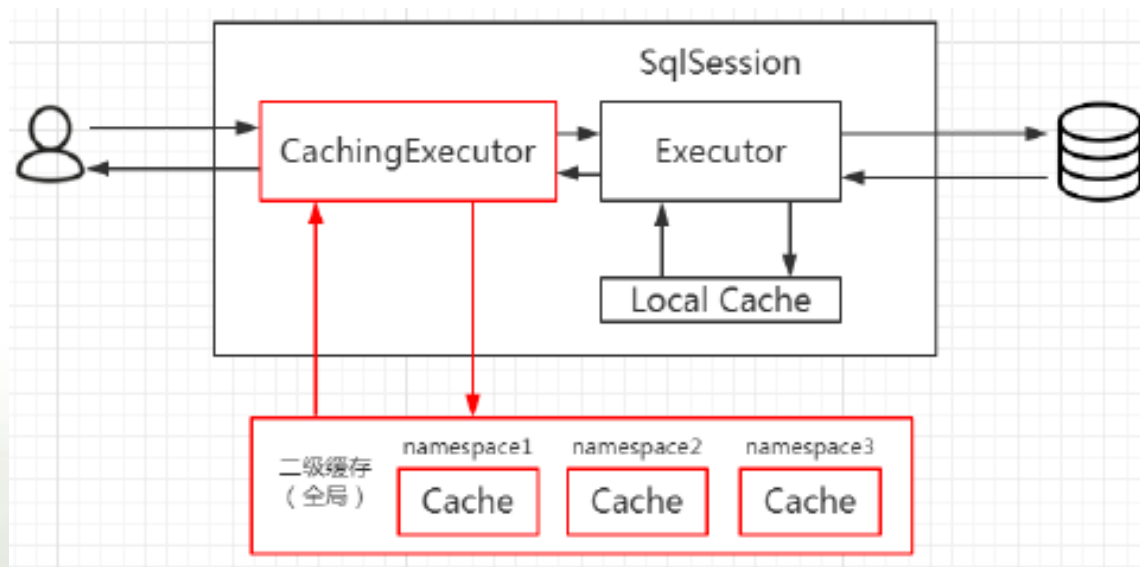
```
mybatis.configuration.lazyLoadingEnabled=true
```



6.1.5 MyBatis的缓存机制

■ MyBatis的缓存分为：

- ✧一级缓存，默认开启。
- ✧二级缓存，需要手动配置开启。
 - **MyBatis**官方不建议配置二级缓存，而建议在业务层开启缓存管理。



一级缓存

■ 一级缓存是**SqlSession**级别的缓存，作用域是一个**SqlSession**。

✧ 在同一个**SqlSession**中，执行相同的查询**SQL**，第一次会先去查询数据库，并写入缓存。第二次再执行时，则直接从缓存中取数据。

✧ 如果两次执行查询**SQL**的中间执行了增删改操作，则会清空该**SqlSession**的缓存。



二级缓存

- 二级缓存是mapper级别的缓存，作用域是mapper的同一个namespace下的sql语句。
 - ✧ 在同一个namespace中，第一次执行查询SQL时，会将查询结果存到二级缓存区域内。第二次执行相同的查询SQL，则直接从缓存中取出数据。
 - ✧ 如果两次执行查询sql的中间执行了增删改操作，则会清空该namespace下的二级缓存。



6.2 MyBatis-Plus框架的使用

- 6.2.1 MyBatis-Plus简介
- 6.2.2 MyBatis-Plus常用注解
- 6.2.3 MyBatis-Plus通用CRUD方法
- 6.2.4 MyBatis-Plus的整合支持
- 6.2.5 MyBatis-Plus代码生成器

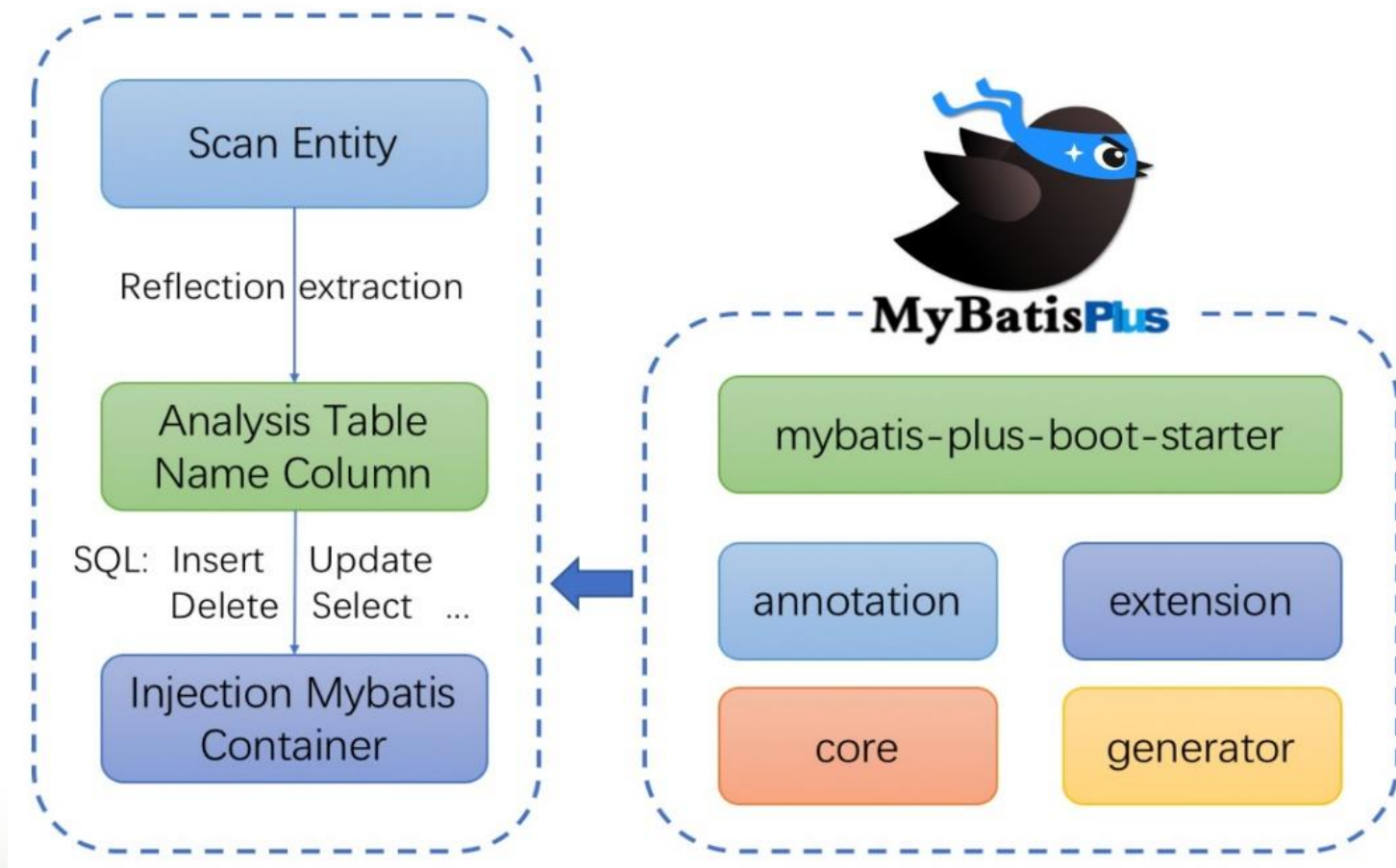


6.2.1 MyBatis-Plus简介

- **Mybatis-Plus**（简称**MP**）是一个国产的开源框架，是**Mybatis**的增强工具，在**Mybatis**的基础上只做增强不做改变，为简化开发、提高效率而生。
- **Mybatis-Plus**核心功能包括代码生成器、**CRUD**接口、条件构造器、分页插件、**Sequence**主键、自定义**ID**生成器等。



Mybatis-Plus架构图



6.2.2 MyBatis-Plus常用注解

- 1. 实体类中的常用注解
- 2. Mapper接口类中的常用注解



1. 实体类中的常用注解

- ① @TableName注解
- ② @TableId注解
- ③ @TableField注解
- ④ @Version注解
- ⑤ @TableLogic注解
- ⑥ @EnumValue注解



① @TableName注解

- **@TableName("t_user")**: 将实体类名（如 **User**）与数据库表名（**t_user**）映射起来。
- 也可在全局配置文件**application.properties**中添加以下全局映射配置来实现：

#配置全局的表名前缀

```
mybatis-plus.global-config.db-config.table-prefix=t_
```



② @TableId注解

- **@TableId(type = IdType.AUTO):** 指定主键生成策略为数据库自增长，开发者无需赋id值

```
@TableId(type = IdType.AUTO)
```

```
private Long id;
```

- 也可在全局配置文件**application.properties**中添加以下全局映射配置来实现：

```
#配置全局的id生成策略
```

```
mybatis-plus.global-config.db-config.id-type=auto
```

@TableId注解

■此外， **IdType**类型还有：

- ✧ **NONE**: MP自动赋值，根据雪花算法实现
- ✧ **INPUT**: 需要开发者手动赋值，如果开发者没有手动赋值，则数据库通过自增的方式给主键赋值
- ✧ **ASSIGN_ID**: MP自动赋值，根据雪花算法实现，要求主键为**Long**、**Integer**或**String**类型
- ✧ **ASSIGN_UUID**: MP自动生成**UUID**进行赋值，要求主键必须为**String**类型



③ @TableField注解

■ 该注解常常通过以下属性解决非主键字段的以下4个问题：

✧ **value**: 对象中的属性名和字段名不一致

✧ **exist**: 对象中的属性字段在表中不存在

✧ **select**: 查询时是否返回该字段的值

✧ **fill**: 非主键字段自动赋值（填充）

- 需要创建自动填充处理器，即**MetaObjectHandler**接口的实现类



示例代码

//映射属性名和字段名，查询时不返回该字段的值

```
@TableField(value="pwd",select = false)
```

```
private String password;
```

```
@TableField(exist = false)
```

```
private List<Comment> commentList;
```

//插入数据时进行填充

```
@TableField(fill = FieldFill.INSERT)
```

```
private Date createTime;
```

//插入和更新数据时进行填充

```
@TableField(fill = FieldFill.INSERT_UPDATE)
```

```
private Date updateTime;
```



自动填充处理器示例代码

```
import com.baomidou.mybatisplus.core.handlers.MetaObjectHandler;
import org.apache.ibatis.reflection.MetaObject;
import org.springframework.stereotype.Component;
import java.util.Date;

@Component
public class MyMetaObjectHandler implements MetaObjectHandler {

    @Override
    public void insertFill(MetaObject metaObject) {
        this.setFieldValByName("createTime",new Date(),metaObject);
        this.setFieldValByName("updateTime",new Date(),metaObject);
    }

    @Override
    public void updateFill(MetaObject metaObject) {
        this.setFieldValByName("updateTime",new Date(),metaObject);
    }
}
```


④ @Version注解

- 标记乐观锁，通过**version**字段和属性来保证数据的安全性，当修改数据的时候，会以**version**作为条件，当条件成立的时候才会修改成功。

- ✧ **version**属性支持的数据类型有：int, Integer, long, Long, Date, Timestamp, LocalDateTime

- ✧ 整数类型下 $\text{newVersion} = \text{oldVersion} + 1$

- ✧ **newVersion**会回写到 **entity**中

- 该注解需要与乐观锁插件 **OptimisticLockerInterceptor**配合使用，在乐观锁插件作用下保证数据的安全性。



示例代码

```
ALTER TABLE t_user  
ADD COLUMN version int(10) NULL DEFAULT 1 COMMENT '乐观锁版本字段' AFTER  
email;  
UPDATE t_user SET version='1';
```

//乐观锁的版本字段

@Version

private Integer version;

@Bean //配置乐观锁插件

```
public OptimisticLockerInterceptor optimisticLockerInterceptor() {  
    return new OptimisticLockerInterceptor();  
}
```

示例代码

```
//基于乐观锁修改指定id的用户记录年龄
//@Transactional 须关闭事务处理
public void updateUserWithOptimisticLocker(Integer id) {
    User user = new User();
    User u1 = user.selectById(id);
    User u2 = user.selectById(id);
    u1.setAge(35);
    u2.setAge(25);
    boolean result2 = u2.updateById(); // result2=true
    boolean result1 = u1.updateById(); // result1=false
}
```

线程 2:update ... set version = 2 where version = 1

线程 1:update ... set version = 2 where version = 1



⑤ @TableLogic注解

- 该注解用于逻辑删除。
- 所谓逻辑删除就是将数据标记为删除，而并非真正的物理删除，查询时需要携带状态条件，确保被标记的数据不被查询到。
 - ◇ 这样做的目的就是避免数据被真正的删除。
- 如果记录被逻辑删除后要真正被物理删除，可用 **@Delete** 注解执行删除SQL语句。



示例代码

```
ALTER TABLE t_user  
ADD COLUMN deleted int(1) NULL DEFAULT 0 COMMENT '1-被删除, 0-未被删除'  
AFTER version;  
UPDATE t_user SET deleted='0';
```

```
//逻辑删除字段 , 1-删除, 0-未删除  
@TableLogic  
private Integer deleted;
```



示例代码

//逻辑删除指定id的用户记录详细信息

@Transactional

```
public void deleteUser(Integer id) {  
    User user = new User();  
    user.setId(id);  
    boolean result = user.deleteById();  
    if(result){  
        System.out.println("您成功删除了数据！");  
    }else {  
        System.out.println("执行删除操作失败！");  
    }  
}
```



⑥ @EnumValue注解

- 该注解为通用枚举类注解，用于将数据库字段映射成实体类的枚举类型成员变量。
- @EnumValue注解添加（可选）在枚举类的需要存储到数据库的属性上。
- 创建枚举类后，需要在全局配置文件 **application.properties** 里添加配置，定义扫描枚举类的包路径。

#配置枚举包扫描

mybatis-plus.type-enums-package=com.itheima.enums

示例代码

```
ALTER TABLE t_user  
ADD COLUMN sex int(1) NULL DEFAULT 1 COMMENT '1-男, 2-女' AFTER deleted;  
UPDATE t_user SET sex='1';
```

```
//性别, 枚举类型  
private SexEnum sex;
```



示例代码

```
import com.baomidou.mybatisplus.annotation.EnumValue;
import com.baomidou.mybatisplus.core.enums.IEnum;
public enum SexEnum implements IEnum<Integer> {
    MAN(1,"男"),
    WOMAN(2,"女");
    //@EnumValue
    private Integer value;
    private String desc;
    SexEnum(Integer value, String desc) {
        this.value = value;
        this.desc = desc;
    }
    @Override
    public Integer getValue() { return this.value;}
    @Override
    public String toString() { return this.desc; }
}
```

2.Mapper接口类中的常用注解

- **@Select**
- **@Insert**
- **@Update**
- **@Delete**



示例代码

```
import com.itheima.domain.Comment;
import org.apache.ibatis.annotations.*;

@Mapper
public interface CommentMapper {

    @Select("SELECT * FROM t_comment WHERE id =#{id}")
    public Comment findCommentById(Integer id);

    @Insert("INSERT INTO t_comment(content,author,a_id) " + "values (#{content},#{author},#{aId})")
    public int insertComment(Comment comment);

    @Update("UPDATE t_comment SET content=#{content} WHERE id=#{id}")
    public int updateComment(Comment comment);

    @Delete("DELETE FROM t_comment WHERE id=#{id}") //物理删除
    public int deleteComment(Integer id);
}
```

注：为了让驼峰命名方式的`aId`属性与表中的`a_id`字段成功映射，须在全局配置文件`application.properties`中添加开启驼峰命名匹配映射配置：

`mybatis-plus.configuration.map-underscore-to-camel-case=true`

6.2.3 MyBatis-Plus通用CRUD方法

- 1.基于BaseMapper的CRUD操作
- 2.基于AR化实体类的CRUD操作
- 3.关于条件构造器



1. 基于BaseMapper的CRUD操作

```
▼ ⓘ BaseMapper
  (m) insert(T): int
  (m) deleteById(Serializable): int
  (m) deleteByMap(Map<String, Object>): int
  (m) delete(Wrapper<T>): int
  (m) deleteBatchIds(Collection<? extends Serializable>): int
  (m) updateById(T): int
  (m) update(T, Wrapper<T>): int
  (m) selectById(Serializable): T
  (m) selectBatchIds(Collection<? extends Serializable>): List<T>
  (m) selectByMap(Map<String, Object>): List<T>
  (m) selectOne(Wrapper<T>): T
  (m) selectCount(Wrapper<T>): Integer
  (m) selectList(Wrapper<T>): List<T>
  (m) selectMaps(Wrapper<T>): List<Map<String, Object>>
  (m) selectObjs(Wrapper<T>): List<Object>
  (m) selectPage(IPage<T>, Wrapper<T>): IPage<T>
  (m) selectMapsPage(IPage<T>, Wrapper<T>): IPage<Map<String, Object>>
```



基于BaseMapper的CRUD操作

- **insert(T entity)**: 插入一条记录, **entity**为实体对象, 返回受影响的行数。
- **updateById(T entity)**: 根据实体对象**entity**的**Id**更新实体对象, 返回受影响的行数。
- **update(T entity, Wrapper<T> wrapper)**: 根据**wrapper**中设置的实体条件更新实体对象**entity**, 返回受影响的行数。
 - ✧ **wrapper**为null时表示全表更新
 - ✧ **MP**提供了**SQL**执行分析插件**SqlExplainInterceptor**, 可用作阻断全表更新、删除的误操作。



基于BaseMapper的CRUD操作

- **deleteById(Serializable id):** 根据主键id删除一条记录，返回受影响的行数。
- **deleteByMap(Map<String, Object> columnMap):** 根据Map中设置的实体条件删除记录，多个Map元素之间为and关系，返回受影响的行数。
- **delete(Wrapper<T> wrapper):** 根据wrapper中设置的实体条件删除记录，返回受影响的行数。
- **deleteBatchIds(Collection idList):** 根据id列表批量删除，返回受影响的行数。



基于BaseMapper的CRUD操作

- **selectById(Serializable id):** 根据主键id查询数据，返回对应的实体对象。
- **selectBatchIds(Collection idList):** 根据主键id列表查询数据，返回对应的实体对象列表。
- **selectOne(Wrapper<T> wrapper):** 根据wrapper中设置的实体条件查询数据，返回一个实体对象，如果结果超过一条会报错。
- **selectCount(Wrapper<T> wrapper):** 根据wrapper中设置的实体条件查询总记录数。



基于BaseMapper的CRUD操作

- **selectList(Wrapper<T> wrapper):** 根据 **wrapper** 中设置的实体条件查询全部记录，返回对应的实体对象列表。
- **selectPage(IPage<T> page, Wrapper<T> wrapper):** 根据 **wrapper** 中设置的实体条件查询全部记录，并将结果按分页设置进行组织并返回。
 - ✧ 分页查询需要与分页插件 **PaginationInterceptor** 配合使用，从而获得分页设置中的当前页的记录。



示例代码1

```
User user = new User();  
user.setAge(22); //更新的字段  
//更新的条件  
QueryWrapper<User> wrapper = new QueryWrapper<>();  
wrapper.eq("id", 6);  
//执行更新操作  
int result = this.userMapper.update(user, wrapper);  
System.out.println("result = " + result);  
//执行全表更新操作:UPDATE t_user SET age=?  
int result = this.userMapper.update(user, null);
```

```
//更新的条件以及字段  
UpdateWrapper<User> wrapper = new UpdateWrapper<>();  
wrapper.eq("id", 6).set("age", 23);  
//执行更新操作  
int result = this.userMapper.update(null, wrapper);  
System.out.println("result = " + result);
```


示例代码2

//根据id列表批量删除

```
int result = this.userMapper.deleteBatchIds(Arrays.asList(1L,10L,20L));
```

```
Map<String, Object> columnMap = new HashMap<>();
```

```
columnMap.put("age",20);
```

```
columnMap.put("name","张三");
```

//将columnMap中的元素设置为删除的条件，多个之间为and关系

```
int result = this.userMapper.deleteByMap(columnMap);
```

```
System.out.println("result = " + result);
```

```
User user = new User();
```

```
user.setAge(20);
```

```
user.setName("张三");
```

//将实体对象进行包装，包装为操作条件

```
QueryWrapper<User> wrapper = new QueryWrapper<>(user);
```

//根据实体的条件，删除记录

```
int result = this.userMapper.delete(wrapper);
```

```
System.out.println("result = " + result);
```

示例代码3

@Bean //配置分页插件（拦截器）

```
public PaginationInterceptor paginationInterceptor() {  
    return new PaginationInterceptor();  
}
```

```
QueryWrapper<User> wrapper = new QueryWrapper<User>();  
wrapper.gt("age", 20); //年龄大于20岁  
Page<User> page = new Page<>(1,2); //设置当前页号和分页大小：当前页号为1，每页2个记录  
//根据分页设置和实体条件查询数据  
IPage<User> iPage = this.userMapper.selectPage(page, wrapper);  
System.out.println("数据总条数： " + iPage.getTotal());  
System.out.println("总页数： " + iPage.getPages());  
List<User> users = iPage.getRecords(); //获取当前页中的记录  
for (User user : users) {  
    System.out.println("user = " + user);  
}
```

关于MP的Page类

■ **com.baomidou.mybatisplus.extension.plugins.pagination.Page**类提供了以下分页属性，**page**对象以**JSON**格式被传输到前端后这些属性可被直接使用。

✧ **records**: 当前页记录

✧ **total**: 数据总条数

✧ **size**: 分页大小

✧ **current**: 当前页号



2.基于AR化实体类的CRUD操作

```
▼ (c) Model
  m Model()
  m insert(): boolean
  m insertOrUpdate(): boolean
  m deleteById(Serializable): boolean
  m deleteById(): boolean
  m delete(Wrapper<T>): boolean
  m updateById(): boolean
  m update(Wrapper<T>): boolean
  m selectAll(): List<T>
  m selectById(Serializable): T
  m selectById(): T
  m selectList(Wrapper<T>): List<T>
  m selectOne(Wrapper<T>): T
  m selectPage(IPage<T>, Wrapper<T>): IPage<T>
  m selectCount(Wrapper<T>): Integer
```



关于AR

- **ActiveRecord**（简称**AR**）由**Rails**最早提出，遵循标准的**ORM**模型：表映射到类，记录映射到对象，字段映射到对象属性。
- **ActiveRecord**负责把自己持久化，在其中封装了对数据库的访问，即**CRUD**。
- 在**MP**中，要开启**AR**只需要将实体类继承**Model**即可。
- 基于**AR**化实体类的**CRUD**操作是对基于**BaseMapper**的**CRUD**操作的再包装。



示例代码

```
public class User extends Model<User> {  
    private Long id;  
  
    .....  
}
```

```
User user = new User();  
QueryWrapper<User> wrapper = new QueryWrapper<User>();  
wrapper.gt("age", 20); //年龄大于20岁  
Page<User> page = new Page<>(1,2); //设置当前页号和分页大小  
//根据分页设置和实体条件查询数据  
IPage<User> iPage = user.selectPage(page, wrapper);  
System.out.println("数据总条数: " + iPage.getTotal());  
System.out.println("总页数: " + iPage.getPages());  
List<User> users = iPage.getRecords(); //获取当前页中的记录  
for (User u : users) {  
    System.out.println("user = " + u);  
}
```


3.关于条件构造器

■ Wrapper的两种实现:

✧ QueryWrapper

✧ LambdaQueryWrapper（推荐）：所有查询操作都封装成方法调用，并支持动态拼写查询条件



两种实现比较

```
LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<>();  
lqw.like(Book::getName, "Spring"); //通过lambda方法引用获取字段属性名
```

```
QueryWrapper<Book> qw = new QueryWrapper<>();  
qw.like("name", "Spring");
```

```
String name = "Spring";  
LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<>();  
lqw.like(Strings.isNotEmpty(name), Book::getName, "Spring"); //动态拼写查询条件
```



Wrapper可用操作

- ①allEq
- ②基本比较操作
- ③模糊查询
- ④排序查询
- ⑤逻辑查询
- ⑥返回指定字段



①allEq

- `allEq(Map<R, V> params)`
- `allEq(Map<R, V> params, boolean null2IsNull)`
- `allEq(boolean condition, Map<R, V> params, boolean null2IsNull)`
- `allEq(BiPredicate<R, V> filter, Map<R, V> params)`
- `allEq(BiPredicate<R, V> filter, Map<R, V> params, boolean null2IsNull)`
- `allEq(boolean condition, BiPredicate<R, V> filter, Map<R, V> params, boolean null2IsNull)`



参数说明

- **params** : **key**为数据库字段名, **value**为字段值
- **nullIsNull** : 为**true**则在**map**的**value** 为**null**时调用 **isNull**方法; 为**false**时则忽略**value**为**null**的条件
- **filter**: 过滤函数, 表示是否允许**params**中的字段传入比对条件中



示例代码

```
QueryWrapper<User> wrapper = new QueryWrapper<>();  
//设置条件  
Map<String,Object> params = new HashMap<>();  
params.put("name", "曹操");  
params.put("age", "20");  
params.put("password", null);  
// SELECT * FROM t_user WHERE password IS NULL AND name = ? AND age = ?  
wrapper.allEq(params);  
// SELECT * FROM t_user WHERE name = ? AND age = ?  
wrapper.allEq(params,false);  
// SELECT * FROM t_user WHERE name = ?  
wrapper.allEq((k, v) -> (k.equals("name") || k.equals("id")),params);
```



②基本比较操作

- **eq:** 等于 =
- **ne:** 不等于 <>
- **gt:** 大于 >
- **ge:** 大于等于 >=
- **lt:** 小于 <
- **le:** 小于等于 <=
- **between:** BETWEEN 值1 AND 值2
- **notBetween:** NOT BETWEEN 值1 AND 值2
- **in:** 字段 IN (value.get(0), value.get(1), ...)
- **notIn:** 字段 NOT IN (v0, v1, ...)



示例代码

```
QueryWrapper<User> wrapper = new QueryWrapper<>();  
//SELECT * FROM t_user WHERE password = ? AND age >= ? AND name IN (?, ?, ?)  
wrapper.eq("password", "123456").ge("age", 20).in("name", "李四", "王五", "赵六");
```



③模糊查询

■ **like: LIKE '%值%'**

✧例: `like("name", "王")` ---> `name like '%王%'`

■ **notLike: NOT LIKE '%值%'**

✧例: `notLike("name", "王")` ---> `name not like '%王%'`

■ **likeLeft: LIKE '%值'**

✧例: `likeLeft("name", "王")` ---> `name like '%王'`

■ **likeRight: LIKE '值%'**

✧例: `likeRight("name", "王")` ---> `name like '王%'`



④排序查询

■ **orderBy: ORDER BY 字段, ...**

✧例: `orderBy(true, true, "id", "name")` ---> `order by id ASC,name ASC`

■ **orderByAsc: ORDER BY 字段, ... ASC**

✧例: `orderByAsc("id", "name")` ---> `order by id ASC,name ASC`

■ **orderByDesc: ORDER BY 字段, ... DESC**

✧例: `orderByDesc("id", "name")` ---> `order by id DESC,name DESC`



⑤逻辑查询

- 多条件查询时默认是**and**逻辑关系
- 主动调用**or()**时，表示紧接的条件是**or**的逻辑关系

```
QueryWrapper<User> wrapper = new QueryWrapper<>();  
//SELECT * FROM tb_user WHERE name = ? OR age = ?  
wrapper.eq("name","李四").or().eq("age", 24);  
//SELECT * FROM tb_user WHERE name = ? AND age = ?  
wrapper.eq("name","李四").eq("age", 24);
```



⑥返回指定字段

- 在MP查询中，默认返回记录所有字段的值，如果有需要也可以通过**select**方法进行指定返回的字段。

```
QueryWrapper<User> wrapper = new QueryWrapper<>();  
//SELECT id,name,age FROM tb_user WHERE name = ? OR age = ?  
wrapper.eq("name", "李四").or().eq("age", 24).select("id", "name", "age");
```



