# Mybatis

27（10（SSM（Spring+SpringMVC(Springboot)+mybatis）+Hibernate）+7(orcale+maven+linux+redis+ngix)+10(项目)）

## 知识点

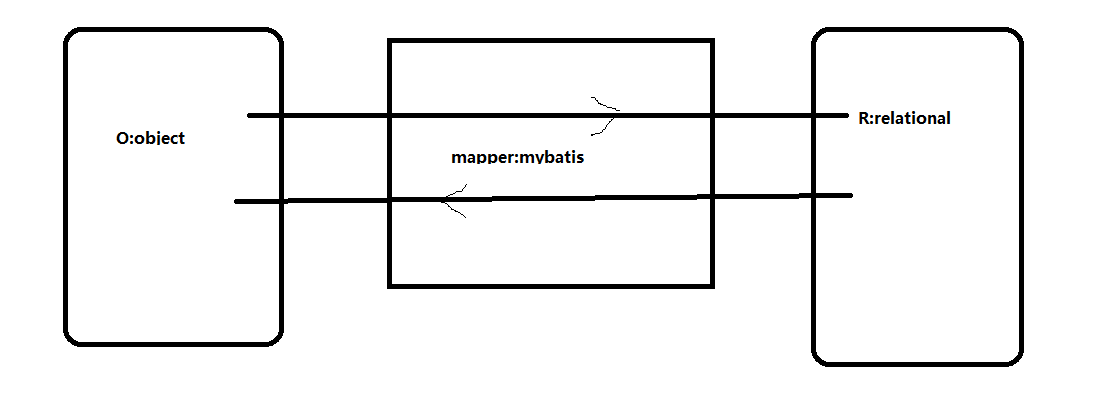
1. 框架的介绍
2. 框架入门程序的搭建
3. Mybatis中的Mapper代理模式
4. Mybatis完成CURD操作
5. 配置文件说明

## 框架的介绍

软件框架：框架就是制定一套规范或者规则（思想），大家（程序员）在该规范或者规则（思想）下工作。或者说使用别人搭好的舞台来做编剧和表演，框架基于jar包和xml文件。

Mybatis框架：

1. 是dao层的框架，对dao层的封装。
2. 是一个半自动化的ORM框架。



1. 特点：
2. 完成java对象和关系型数据库数据的转换，可以封装数据源。

2. 将sql语句和java代码分离，降低耦合度。

3. 支持缓存机制。

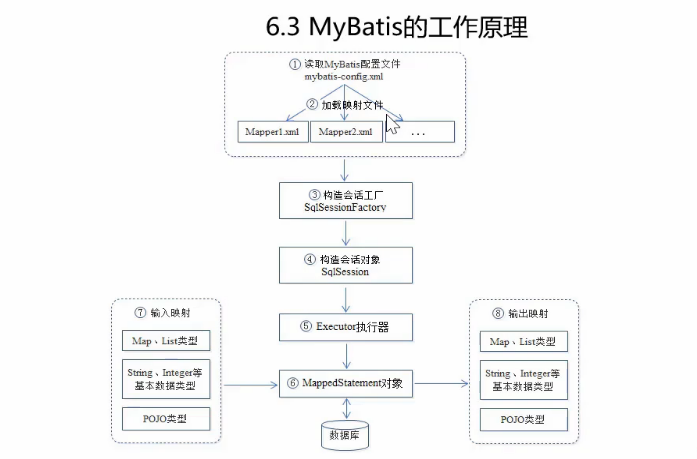
4. 将表与表之间的关系转换为实体与实体之间的关系。

4.工作流程

加载驱动------>Connection-------->Preparstatement--------->ResultSet

解析配置文件------>SqlSessionFactory-------SqlSession

-----Excutor(执行器)------MapperedStatement



## 框架入门程序的搭建

1. 创建项目，加入依赖包



1. 编写配置文件

**全局配置文件**：数据源，映射文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <!-- 数据源 -->  <environments default=*"development"*>  <environment id=*"development"*>  <transactionManager type=*"JDBC"* />  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"com.mysql.jdbc.Driver"* />  <property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatistest"* />  <property name=*"username"* value=*"root"* />  <property name=*"password"* value=*"root"* />  </dataSource>  </environment>  </environments>    <!-- 映射文件 -->  <mappers>  <mapper resource=*"com/offcn/bean/UserMapper.xml"* />  </mappers>  </configuration> |

**映射文件：sql语句**

准备数据库以及表

**创建对应的实体类User**

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">  <!-- namespace:命名空间，可以任意定义 -->  <mapper namespace=*"org.mybatis.example.BlogMapper"*>  <!-- id:在当前的命名空间下唯一, 可以任意定义，parameterType:输入参数类型,resultType：输出参数类型（包+类名）-->  <select id=*"selectUser"* parameterType=*"int"* resultType=*"com.offcn.bean.User"*>  <!--#{}等价于占位符？ 输入参数是数字类型,#{任意定义} -->  select \* from user where id = #{pid}  </select>  </mapper> |

1. 解析配置文件并调用sql语句

|  |
| --- |
| **public** **class** App {  **public** **static** **void** main(String[] args) **throws** IOException {  //全局配置文件的路径  String resource = "MybatisConfig.xml";  //将全局配置文件转化到输入流中  InputStream inputStream = Resources.*getResourceAsStream*(resource);  //通过全局配置文件的输入流得到SqlSessionFactory（会话工厂）  SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);  //通过会话工厂获得会话对象  SqlSession ss = sqlSessionFactory.openSession();  //通过会话对象调用sql语句，参数一：映射文件中对应的sql语句的id,参数二：占位符#{}的值  User u=ss.selectOne("org.mybatis.example.BlogMapper.selectUser", 1);  System.***out***.println(u);  }    } |

## 控制台sql日志打印

1. 加入日志依赖包



1. 编写日志的文件

在src下创建log4j.properties

|  |
| --- |
| # Global logging configuration  log4j.rootLogger=DEBUG, stdout  # MyBatis logging configuration...  log4j.logger.org.mybatis.example.BlogMapper=TRACE  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

## Mybatis中的Mapper代理模式

entity----dao----service----controller

dao,service层面向接口开发（UserDao,UserDaoImpl）

以mybatis完成传统的dao层开发

代理：A遵循一定协议或者规范的前提下，帮助B完成一部分额外的业务。

Mapper代理：程序员在开发的过程中遵循Mybatis的某些规范后，Mybatis会帮我们生成Dao层的实现类。

mapper代理的规范：映射文件和Dao层接口之间

1. 映射文件的命名空间是接口的全路径；
2. 映射文件的sql语句的id=接口里的方法名；
3. 映射文件的sql语句的输入输出参数和接口里的方法的输入参数以及返回结果保持一致。

### 实体类bean

User实体

|  |
| --- |
| **package** com.offcn.bean;  **import** java.util.Date;  **public** **class** User {  **private** **int** id;  **private** String username;  **private** Date birthday;  **private** String sex;  **private** String address;  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** Date getBirthday() {  **return** birthday;  }  **public** **void** setBirthday(Date birthday) {  **this**.birthday = birthday;  }  **public** String getSex() {  **return** sex;  }  **public** **void** setSex(String sex) {  **this**.sex = sex;  }  **public** String getAddress() {  **return** address;  }  **public** **void** setAddress(String address) {  **this**.address = address;  }  @Override  **public** String toString() {  **return** "User [id=" + id + ", username=" + username + ", birthday=" + birthday + ", sex=" + sex + ", address="  + address + "]";  }  } |

映射文件UserMapper.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=*"com.offcn.dao.UserDao"*>    <select id=*"getUserById"* parameterType=*"int"* resultType=*"com.offcn.bean.User"*>  select \* from user where id=#{uid}  </select>    </mapper> |

### 持久层dao

dao层接口UserDao

|  |
| --- |
| **public** **interface** UserDao {  **public** User getUserById(**int** id);  } |

### 业务service（这里通过主函数调用测试dao层）

业务层UserServiceImpl

|  |
| --- |
| **public** **class** UserServiceImpl {  **public** **static** **void** main(String[] args) **throws** IOException {  //全局配置文件的路径  String resource = "MybatisConfig.xml";  //将全局配置文件转化到输入流中  InputStream inputStream = Resources.*getResourceAsStream*(resource);  //通过全局配置文件的输入流得到SqlSessionFactory（会话工厂）  SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);  //通过会话工厂获得会话对象  SqlSession ss = sqlSessionFactory.openSession();    //获取接口对应的实现类（Mybatis框架在内存中生成）的类对象  UserDao udao = ss.getMapper(UserDao.**class**);  User user = udao.getUserById(5);  System.***out***.println(user);  }  } |

## Mybatis完成CURD操作

案例：

映射文件

|  |
| --- |
| <select id=*"getUserById"* parameterType=*"int"* resultType=*"com.offcn.bean.User"*>  select \* from user where id=#{uid}  </select>    <!--输入参数是自定义类型，#{自定义类型的属性名} -->  <insert id=*"saveUser"* parameterType=*"com.offcn.bean.User"*>  insert into user values(null,#{username},#{birthday},#{sex},#{address})  </insert>    <update id=*"updUser"* parameterType=*"com.offcn.bean.User"*>  update user set username=#{username},sex=#{sex} where id=#{id}  </update>    <delete id=*"delUser"* parameterType=*"int"*>  delete from user where id=#{uid}  </delete>    <!-- 查询姓名包含某个字以及性别为男或者女的所有用户 -->  <!-- ${}和#{}：${}是拼接的方式，#{}是占位符的方式 -->  <select id=*"getByNameAndSex"* parameterType=*"com.offcn.bean.User"* resultType=*"com.offcn.bean.User"*>  select \* from user where username like '%${username}%' and sex=#{sex}  </select> |

接口

public User getUserById(int id);

public int saveUser(User user);

public int updUser(User user);

public int delUser(int uid);

public List<User> getByNameAndSex(User user);

测试省略。

## 多个输入参数的解决办法

1. 封装到一个对象里------#{对象数姓名}
2. 放到map中-------#{map的key}
3. 注解的方式@Param(“uname”) -------#{uname}
4. 参数的索引-----#{0}

案例：

映射文件

|  |
| --- |
| <select id=*"getByNameAndSex1"* parameterType=*"com.offcn.bean.User"* resultType=*"com.offcn.bean.User"*>  select \* from user where username like "%"#{username}"%" and sex=#{sex}  </select>    <select id=*"getByNameAndSex2"* parameterType=*"map"* resultType=*"com.offcn.bean.User"*>  select \* from user where username like "%"#{uname}"%" and sex=#{sex}  </select>    <select id=*"getByNameAndSex3"* resultType=*"com.offcn.bean.User"*>  select \* from user where username like "%"#{un}"%" and sex=#{usex}  </select>    <select id=*"getByNameAndSex4"* resultType=*"com.offcn.bean.User"*>  select \* from user where username like "%"#{0}"%" and sex=#{1}  </select> |

接口

public List<User> getByNameAndSex1(User user);

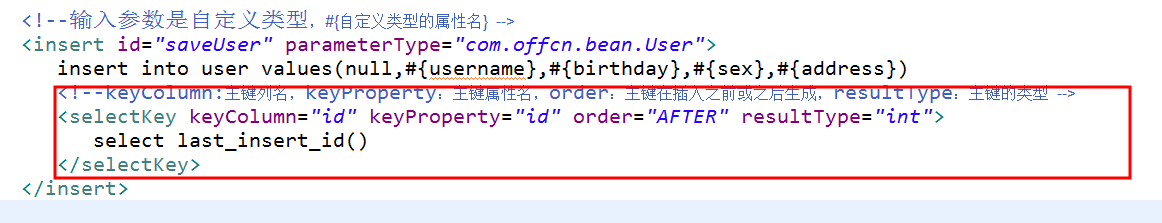
public List<User> getByNameAndSex2(Map map);

public List<User> getByNameAndSex3(@Param("un") String name,@Param("usex")String sex);

public List<User> getByNameAndSex4(String name,String sex);

单元测试

注意：如何获取最新插入数据的主键



## 配置文件说明

1. properties标签

|  |
| --- |
| <!-- 引入外部的properties文件 -->  <properties resource=*"db.properties"*>  <!-- <property name="db\_driver" value="com.mysql.jdbc.Driver"/>  <property name="db\_url" value="jdbc:mysql://localhost:3306/mybatistest"/>  <property name="db\_uname" value="root"/>  <property name="db\_pwd" value="root"/> -->  </properties> |

1. Setting标签：用来设置mybatis的一些运行行为，例如缓存，延迟加载，是否开启自动驼峰命名规则（camel case）映射等。

|  |
| --- |
| <!--是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射-->  <settings>  <setting name=*"mapUnderscoreToCamelCase"* value=*"true"*/>  </settings> |

1. typeAliases标签:类型别名,是为 Java 类型设置一个短的名字。 它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。

|  |
| --- |
| <typeAliases>  <!-- 单个起别名，type：自定义类型，alias：别名 -->  <!-- <typeAlias type="com.offcn.bean.User" alias="u"/> -->  <!-- 为包里的类批量起别名，默认别名是类名或者把类名的首字母小写 -->  <package name=*"com.offcn.bean"*/>  </typeAliases> |

## 配置文件其他属性

### environments

数据源环境，可以包含多个数据源环境。

|  |
| --- |
| <!-- 数据源 -->  <environments default=*"test"*>  <environment id=*"development"*>  <transactionManager type=*"JDBC"* />  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"${db\_driver}"* />  <property name=*"url"* value=*"${db\_url}"* />  <property name=*"username"* value=*"${db\_uname}"* />  <property name=*"password"* value=*"${db\_pwd}"* />  </dataSource>  </environment>  <environment id=*"test"*>  <transactionManager type=*"JDBC"* />  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"${db\_driver}"* />  <property name=*"url"* value=*"${db\_url}"* />  <property name=*"username"* value=*"${db\_uname}"* />  <property name=*"password"* value=*"${db\_pwd}"* />  </dataSource>  </environment>  </environments> |

### databaseIdProvider

数据库厂商标志，可以根据不同的厂商标志执行不同的sql语句

|  |
| --- |
| <databaseIdProvider type=*"DB\_VENDOR"*>  <property name=*"SQL Server"* value=*"sqlserver"*/>  <property name=*"DB2"* value=*"db2"*/>  <property name=*"Oracle"* value=*"oracle"* />  </databaseIdProvider>  映射文件： |

### mappers

用来管理映射文件，共有四种方式可以使用

|  |
| --- |
| <!-- 映射文件 -->  <mappers>  <!-- 方式一：指定映射文件相对src的路径 -->  <!-- <mapper resource="com/offcn/bean/UserMapper.xml"/> -->  <!-- 方式二：指定映射文件在硬盘的路径 -->  <!-- <mapper url="file:///D:\myworksp\mybatis002\src\com\offcn\bean\UserMapper.xml"/> -->  <!-- 接口和映射文件必须同包同名 -->  <!-- 方式三：指定和映射文件同包同名的接口的路径 -->  <!-- <mapper class="com.offcn.dao.UserDao"/> -->  <!-- 方式四：批量管理映射文件 -->  <package name=*"com.offcn.dao"*/>  </mappers> |

## 动态sql

### resultMap

完成结果的手动映射，一般结果集的列名和实体的属性名不一致时使用。

映射文件UserDao.xml：

|  |
| --- |
| <!--type:返回结果的类型，id：任意定义在当前命名空间唯一 -->  <resultMap type=*"user"* id=*"rs"*>  <!--完成主键映射，column：列名 ，property：属性名 -->  <id column=*"uid"* property=*"id"*/>  <!--完成非主键映射 -->  <result column=*"uname"* property=*"username"*/>  </resultMap>  <select id=*"getUserById"* parameterType=*"int"* resultMap=*"rs"*>  select id uid,username uname from user where id=#{uid}  </select> |

接口：

public User getUserById(int id);

单元测试：

|  |
| --- |
| @Test  **public** **void** getUserById() {  User user = udao.getUserById(1);  System.***out***.println(user);  } |

### sql片段

提取sql语句的公共部分，统一维护。

语法：

定义:

<sql id="任意定义，要在当前命名空间唯一的">

select \* from user

</sql>

引用：

<include refid="sql片段的id"></include>

### foreach

遍历标签，遍历List集合和数组。

映射文件UserDao.xml

|  |
| --- |
| <!-- foreach -->  <delete id=*"delMulti"* parameterType=*"java.util.List"*>  delete from user where id in  <!--collection:输入参数为List集合时，必须写list,  item:为集合里的每一项起名，可以任意定义  separator：每一项中间的分割符  open:在执行循环体之前拼接的内容；  close：在执行循环体之后拼接的内容；-->  <foreach collection=*"list"* item=*"uid"* separator=*","* open=*"("* close=*")"*>  #{uid}  </foreach>  </delete>    <delete id=*"delMultiByArray"* parameterType=*"java.lang.reflect.Array"*>  delete from user where id in  <!--collection:输入参数为List集合时，必须写array,  item:为集合里的每一项起名，可以任意定义  separator：每一项中间的分割符  open:在执行循环体之前拼接的内容；  close：在执行循环体之后拼接的内容；-->  <foreach collection=*"array"* item=*"uid"* separator=*","* open=*"("* close=*")"*>  #{uid}  </foreach>  </delete>    <delete id=*"delMultiByObj"* parameterType=*"UserVo"*>  delete from user where id in  <!--collection:输入参数为自定义类型时，必须写实体类的属性名,  item:为集合里的每一项起名，可以任意定义  separator：每一项中间的分割符  open:在执行循环体之前拼接的内容；  close：在执行循环体之后拼接的内容；-->  <foreach collection=*"ids"* item=*"uid"* separator=*","* open=*"("* close=*")"*>  #{uid}  </foreach>  </delete>  <!-- 批量插入-->  <insert id=*"saveMulti"* parameterType=*"java.util.List"*>  insert into user(username,sex) values  <foreach collection=*"list"* item=*"u"* separator=*","*>  (#{u.username},#{u.sex})  </foreach>  </insert> |

接口UserDao

public int delMulti(List<Integer> list);

public int delMultiByArray(int[] ids);

public int delMultiByObj(UserVo uv);

public int saveMulti(List<User> ulist);

单元测试：

|  |
| --- |
| @Test  **public** **void** delByIds() {  List<Integer> ids=**new** ArrayList<Integer>();  ids.add(30);  ids.add(33);  ids.add(34);  udao.delMulti(ids);  ss.commit();  }    @Test  **public** **void** delByIdsByArray() {  udao.delMultiByArray(**new** **int**[] {39,40,41});  ss.commit();  }    @Test  **public** **void** delByIdsByObj() {  UserVo uv=**new** UserVo();  uv.setIds(**new** **int**[] {39,40,36});  udao.delMultiByObj(uv);  ss.commit();  }    @Test  **public** **void** saveMulti() {  List<User> ulist=**new** ArrayList<User>();  ulist.add(**new** User("李某某","男"));  ulist.add(**new** User("李某","女"));  ulist.add(**new** User("李","男"));  udao.saveMulti(ulist);  ss.commit();  } |

### where if

一般用作条件检索

映射文件

|  |
| --- |
| <!--where:解析成where关键字，会自动去掉第一个符合条件的限定条件中的and -->  <select id=*"getByNameSex"* parameterType=*"map"* resultType=*"user"*>  select \* from user  <where>  <if test=*"uname!=null and uname!=''"*>  and username like "%"#{uname}"%"  </if>  <if test=*"usex!=null and usex!=''"*>  and sex=#{usex}  </if>  </where>  </select> |

接口

public List<User> getByNameSex(Map map);

测试

|  |
| --- |
| @Test  **public** **void** getByWh() {  Map<String, Object> map=**new** HashMap<String, Object>();  map.put("uname", "");  map.put("usex", "");  List<User> ulist = udao.getByNameSex(map);  System.***out***.println(ulist);  ss.commit();  } |

### choose when orherwise

一般用作条件检索

映射文件

|  |
| --- |
| <!--choose when otherwise:某个判断满足条件后，其他条件就不会再执行 -->  <select id=*"getByNameSex1"* parameterType=*"map"* resultType=*"user"*>  select \* from user where  <choose>  <when test=*"uname!=null and uname!=''"*>  username like "%"#{uname}"%"  </when>  <when test=*"usex!=null and usex!=''"*>  sex=#{usex}  </when>  <otherwise>  1=1  </otherwise>  </choose>  </select> |

接口

public List<User> getByNameSex1(Map map);

测试

|  |
| --- |
| @Test  **public** **void** getByWh1() {  Map<String, Object> map=**new** HashMap<String, Object>();  map.put("uname", "");  map.put("usex", "男");  List<User> ulist = udao.getByNameSex1(map);  System.***out***.println(ulist);  ss.commit();  } |

### set if

用于动态更新某些字段

映射文件

|  |
| --- |
| <!-- set:解析为set关键字，可以自动去掉最后一个更新的字段后面的逗号 -->  <update id=*"updUser"* parameterType=*"user"*>  update user  <set>  <if test=*"username!=null and username!=''"*>  username=#{username},  </if>  <if test=*"sex!=null and sex!=''"*>  sex=#{sex}  </if>  </set>  where id=#{id}  </update> |

接口

public int updUser(User user);

测试

|  |
| --- |
| @Test  **public** **void** updUser() {  User u=**new** User();  u.setId(16);  u.setUsername("李晓雨");  //u.setSex("男");  udao.updUser(u);  ss.commit();  } |

### trim

格式化sql语句，同时设置sql的前后缀。

映射文件

|  |
| --- |
| <update id=*"updUser1"* parameterType=*"user"*>  <!--prefix:前缀，在trim中内容执行之前拼接  suffix：后缀：在trim中内容执行之后拼接  suffixOverrides：忽略后缀  prefixOverrides:忽略前缀-->  <trim prefix=*"update user set"* suffix=*"where id=#{id}"* suffixOverrides=*","*>    <if test=*"username!=null and username!=''"*>  username=#{username},  </if>  <if test=*"sex!=null and sex!=''"*>  sex=#{sex}  </if>    </trim>  </update> |

接口

public int updUser1(User user);

测试

|  |
| --- |
| @Test  **public** **void** updUser() {  User u=**new** User();  u.setId(16);  u.setUsername("李晓雨");  //u.setSex("男");  udao.updUser1(u);  ss.commit();  } |

### bind

在输入字符串的基础拼接一些其他的字符组成一个新的字符串。

映射文件

|  |
| --- |
| <select id=*"getByUname"* parameterType=*"string"* resultType=*"User"*>  <!--name:新的字符串的变量名 -->  <bind name=*"uname"* value=*"'%'+\_parameter+'%'"*/>  select \* from user where username like #{uname}  </select> |

接口

public List<User> getByUname(String uname);

测试

|  |
| --- |
| @Test  **public** **void** testBind() {  List<User> ulist = udao.getByUname("阿");  System.***out***.println(ulist);  ss.commit();  } |

## 关系映射

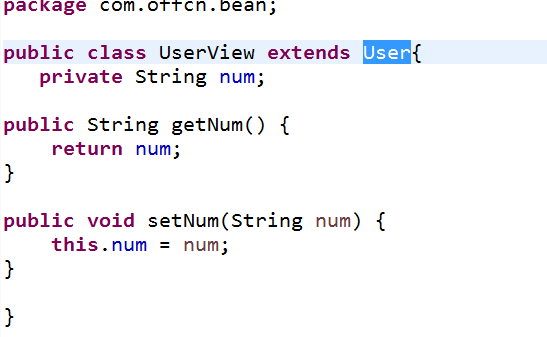
处理关系映射时可以采用自动映射和手动映射（关联查询，嵌套查询（支持延迟加载））。

### 一对一

自动映射

先编写级联查询的sql语句，创建一个类包含结果集里的所有列名作为属性名。

创建实体类UserView



映射文件：

|  |
| --- |
| <!-- 1-1:自动映射 -->  <select id=*"oneToOne"* resultType=*"UserView"*>  select u.\*,c.num from user u,card c where u.id=c.per\_fk  </select> |

接口：

|  |
| --- |
| public List<User> oneToOne(); |

测试：

|  |
| --- |
| @Test  **public** **void** testOneToOne() {  List<UserView> ulist = udao.oneToOne();  System.***out***.println(ulist);  ss.commit();  } |

手动映射

1. 把表和表之间的关系转换为实体和实体之间的关系

在其中一方加入另一方的实体类型属性（在User类中加Card类型的属性）

1. 在映射文件中完成手动映射

关联查询：通过一条sql语句查到结果，resultMap完成映射。

映射文件：

|  |
| --- |
| <!-- 1-1:手动映射之级联查询 -->  <resultMap type=*"user"* id=*"oTors"*>  <id column="id" property="id"/>  <result column="username" property="username"/>  <result column="sex" property="sex"/>  <!--association:完成自定义类型的映射  property:User类中自定义类型（Card）的属性名  javaType:指定对应属性的类型-->  <association property=*"card"* javaType=*"card"* resultMap=*"cardrs"*>  <result property="num" column="num"/>  </association>  </resultMap>  <select id=*"oneToOne1"* resultMap=*"oTors"*>  select u.\*,c.num from user u,card c where u.id=c.per\_fk  </select> |

接口：

|  |
| --- |
| public List<User> oneToOne1(); |

测试：

|  |
| --- |
| @Test  **public** **void** testOneToOne() {  List<User> ulist = udao.oneToOne1();  System.***out***.println(ulist);  ss.commit();  } |

嵌套查询：先查询主信息表的信息，然后再通过其他sql语句查询关联的信息。

映射文件：

|  |
| --- |
| <!-- 1-1:手动映射之嵌套查询（分步） -->  <resultMap type=*"User"* id=*"oTors2"* >  <id column=*"id"* property=*"id"*/>  <result column=*"username"* property=*"username"*/>  <result column=*"sex"* property=*"sex"*/>  <!-- select:调用另外一条sql语句（命名空间.sql语句id）  column:在第一个结果集中用来关联查询的列名 -->  <association property=*"card"* javaType=*"card"*  select=*"com.offcn.dao.UserDao.getCardByUid"* column=*"id"*></association>  </resultMap>  <select id=*"oneToOne2"* resultMap=*"oTors2"*>  select \* from user  </select>    <select id=*"getCardByUid"* parameterType=*"int"* resultType=*"card"*>  select \* from card where per\_fk=#{uid}  </select> |

接口：

|  |
| --- |
| **public** List<User> oneToOne2(); |

测试：

|  |
| --- |
| @Test  **public** **void** testOneToOne() {  List<User> ulist = udao.oneToOne2();  System.***out***.println(ulist);  ss.commit();  } |

**细节注意**：

1. resultMap可以继承

### 一对多

用户（1）和订单（n）

手动映射

1. 在用户的实体里添加订单的集合属性，set，get方法。
2. 映射文件

关联映射

|  |
| --- |
| <!-- 1-n:关联查询 -->  <resultMap type=*"Orders"* id=*"orderrs"*>  <id column=*"oid"* property=*"id"*></id>  <result column=*"number"* property=*"number"*/>  </resultMap>  <resultMap type=*"user"* id=*"oTmrs"* extends=*"commrs"*>  <collection property=*"olist"* ofType=*"Orders"* resultMap=*"orderrs"*>  <id column=*"oid"* property=*"id"*></id>  <result column=*"number"* property=*"number"*/>  </collection>  </resultMap>  <select id=*"oneToMany"* resultMap=*"oTmrs"*>  select u.\*,o.id oid,o.number from user u,orders o where u.id=o.userId  </select> |

接口：public List<User> oneToMany();

测试：

|  |
| --- |
| @Test  **public** **void** testOneToMany() {  List<User> ulist = udao.oneToMany();  System.***out***.println(ulist);  ss.commit();  } |

|  |
| --- |
| <!--1-n:嵌套查询-->  <resultMap type=*"user"* id=*"oTmrs2"*>  <collection property=*"olist"* ofType=*"Orders"*  select=*"com.offcn.dao.UserDao.getOrdersByUid"* column=*"id"*></collection>  </resultMap>  <select id=*"oneToMany2"* resultMap=*"oTmrs2"*>  select \* from user  </select>    <select id=*"getOrdersByUid"* parameterType=*"int"* resultType=*"Orders"*>  select \* from orders where userId=#{uid}  </select> |

嵌套查询的映射文件

接口：public List<User> oneToMany2();

测试：

|  |
| --- |
| @Test  **public** **void** testOneToMany() {  List<User> ulist = udao.oneToMany2();  System.***out***.println(ulist);  ss.commit();  } |

多对多

用户和商品

查询某个用户购买所有的商品（商品购买的订单号，下单人的信息，以及商品购买的数目）

|  |
| --- |
| <!-- 查询某个用户购买所有的商品（商品购买的订单号，下单人的信息，以及商品购买的数目） -->  <resultMap type=*"user"* id=*"mTmrs"*>  <result column=*"username"* property=*"username"*/>  <collection property=*"olist"* ofType=*"Orders"*>  <result property=*"number"* column=*"number"*/>  <collection property=*"dlist"* ofType=*"Orderdetail"*>  <result column=*"itemsNum"* property=*"itemsNum"*/>  <association property=*"items"* javaType=*"Items"*>  <result column=*"name"* property=*"name"*/>  <result column=*"price"* property=*"price"*/>  </association>  </collection>  </collection>  </resultMap>  <select id=*"manyToMany"* resultMap=*"mTmrs"*>  select u.username,o.number,i.name,i.price,d.itemsNum  from user u,orders o,orderdetail d,items i  where u.id=o.userId and o.id=d.ordersId and d.itemsId=i.id  </select> |

接口：

测试：

## 延迟加载

先加载主信息表的查询，在需要关联的信息时，再查询对应的关联信息。

配置：

在全局配置文件启用延时加载

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

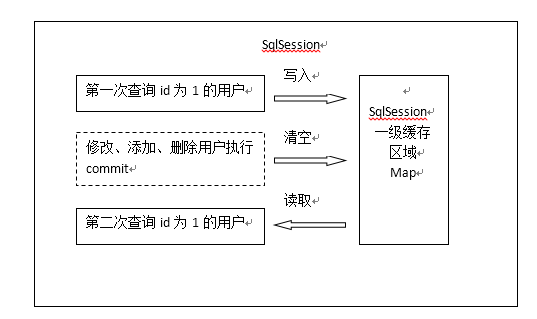
注意：延时加载仅针对与嵌套查询。

## 缓存机制

1. 为避免频繁的从数据库读取数据，所以会常用作查询的数据放到缓存中。
2. Mybatis内部提供的缓存策略有两种

一级缓存：mybatis框架内置提供，不需要开发人员做任何配置。SqlSession级别的缓存，每一个SqlSession只能使用自己的一级缓存区域数据，不可以跨sqlSession共享一级缓存。

在执行增删改之后一级缓存域的数据会被自动清除掉，也可以手动清除，调用clearCache();



测试案例：

|  |
| --- |
| @Test  **public** **void** oneCache() {  User user = udao.getUser();  System.***out***.println("第一次查询："+user);    /\*udao.delMultiByArray(new int[] {41,45});  ss.commit();\*/  ss.clearCache();//清除一级缓存区域的数据    User user2=udao.getUser();  System.***out***.println("第二次查询："+user2);  } |

二级缓存：Mapper接口级别的缓存。不同的sqlSession操作同一个Mapper接口可以共享二级缓存区域的数据。



手动设置：

1. 指定二级缓存的类型（mybatis框架提供的二级缓存实现类，也可以使用第三方插件实现mybatis的二级缓存（ehcahce,redis）），在增删改后二级缓存区域的数据也会被清除掉。

<!-- 指定二级缓存的类型,type省略不写默认是自带的-->

<cache type="org.apache.ibatis.cache.impl.PerpetualCache"></cache>

1. 序列化缓存的对象。



|  |
| --- |
| @Test  **public** **void** secondCache() {  User user = udao.getUser();//第一个SqlSession对象ss来操作UserDao接口  System.***out***.println("第一次查询："+user);      udao.delMultiByArray(**new** **int**[] {41,45});  ss.commit();  ss.close();      User user2=udao1.getUser();//第二个SqlSession对象ss1来操作UserDao接口  System.***out***.println("第二次查询："+user2);  ss1.close();  } |

### 逆向工程

根据数据库的表自动生成实体类，dao层接口，映射文件。

1. 依赖包
2. 配置文件

|  |
| --- |
| <?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>  <context id=*"testTables"* targetRuntime=*"MyBatis3"*>  <commentGenerator>  <!-- 是否去除自动生成的注释 true：是 ： false:否 -->  <property name=*"suppressAllComments"* value=*"true"* />  </commentGenerator>  <!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->  <jdbcConnection driverClass=*"com.mysql.jdbc.Driver"*  connectionURL=*"jdbc:mysql://localhost:3306/mybatistest"* userId=*"root"*  password=*"root"*>  </jdbcConnection>  <!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver"  connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"  userId="yycg"  password="yycg">  </jdbcConnection> -->  <!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL 和  NUMERIC 类型解析为java.math.BigDecimal -->  <javaTypeResolver>  <property name=*"forceBigDecimals"* value=*"false"* />  </javaTypeResolver>  <!-- targetProject:生成实体类的位置 -->  <javaModelGenerator targetPackage=*"com.offcn.bean"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"false"* />  <!-- 从数据库返回的值被清理前后的空格 -->  <property name=*"trimStrings"* value=*"true"* />  </javaModelGenerator>  <!-- targetProject:mapper映射文件生成的位置 -->  <sqlMapGenerator targetPackage=*"com.offcn.dao"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"false"* />  </sqlMapGenerator>  <!-- targetPackage：mapper接口生成的位置 -->  <javaClientGenerator type=*"XMLMAPPER"*  targetPackage=*"com.offcn.dao"*  targetProject=*".\src"*>  <!-- enableSubPackages:是否让schema作为包的后缀 -->  <property name=*"enableSubPackages"* value=*"false"* />  </javaClientGenerator>  <!-- 指定数据库表 -->    <table tableName=*"user"*></table>  <table tableName=*"card"*></table>  <table tableName=*"orders"*></table>    </context>  </generatorConfiguration> |

1. 解析配置

|  |
| --- |
| package com.offcn.utils;  import java.io.File;  import java.io.IOException;  import java.util.ArrayList;  import java.util.List;  import org.mybatis.generator.api.MyBatisGenerator;  import org.mybatis.generator.config.Configuration;  import org.mybatis.generator.config.xml.ConfigurationParser;  import org.mybatis.generator.exception.XMLParserException;  import org.mybatis.generator.internal.DefaultShellCallback;  public class GeneratorSqlmap {  public void generator() throws Exception{  List<String> warnings = new ArrayList<String>();  boolean overwrite = true;  File configFile = new File("D:\\myworksp\\mybatis004\\src\\com\\offcn\\utils\\generatorConfig.xml");  ConfigurationParser cp = new ConfigurationParser(warnings);  Configuration config = cp.parseConfiguration(configFile);  DefaultShellCallback callback = new DefaultShellCallback(overwrite);  MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,  callback, warnings);  myBatisGenerator.generate(null);  }  public static void main(String[] args) throws Exception {  try {  GeneratorSqlmap generatorSqlmap = new GeneratorSqlmap();  generatorSqlmap.generator();  } catch (Exception e) {  e.printStackTrace();  }    }  } |

注意：在重复执行主函数时，java文件会覆盖，xml文件追加。