===========================================================

## 重点

===========================================================

SqlMapperConfig.xml

mapper.xml和mapper.java

namespace【全类名】

statement的id【方法名】

输入映射【参数】：parameterType

输出映射【返回值】：resultType，resultMap

===========================================================

## 原生态JDBC问题总结

===========================================================

① 数据库连接：

使用是创建，不使用立即释放

频繁的开启和关闭，造成数据库资源浪费，数据库性能降低。

解决方案：

使用连接池管理

② sql语句：硬编码到Java代码中，不利于维护

解决方案：将SQL语句配置到xml配置文件中

③ PrepareStatement设置参数：

对占位符位置和设置参数值，硬编码在Java代码中

解决方案：

将SQL语句及占位符和参数全部配置在xml配置文件中

④ 结果集数据遍历：

将获取表的字段进行硬编码

解决方案：

将查询的结果集，自动映射成Java对象

 工作量大，操作数据库至少要5步；

 业务代码和技术代码耦合；

 连接资源手动关闭，带来了隐患

===========================================================

## Mybatis与Hibernate的本质区别和应用场景

===========================================================

Hibernate：标准的ORM框架

自动生成SQL语句，不利于优化

应用场景：需求变化不多的中小型项目

后台管理系统

erp

orm

oa

Mybatis：不完全的ORM框架

专注SQL本身【程序员编写SQL语句】，利于优化

也实现映射【输入输出映射】

应用场景：需求比较多的项目

互联网项目

===========================================================

## 框架原理

===========================================================

Mybatis：持久层框架，apache

主要精力在sql上，通过映射方式，自由灵活生成（半自动）满足需要的SQL语句【程序员自由编写】

PrepareStatement输入参数自动进行输入映射

将结果集映射成Java对象【输出映射】

下载：github

SqlMapConfig.xml【全局配置文件】

数据源

事务

mapper.xml【映射文件】

配置SQL语句

SqlSessionFactory【会话工厂】

创建SQLSession

SQLSession

操作数据库【增删改查】

Executor【执行器】接口

基本执行器

缓存执行器

SQLSession内部通过执行器操作数据库

mappedstatement【底层封装对象】

对操作数据库存储封装

SQL语句

输入参数

输出结果类型

输入参数 | 输出结果类型

java简单类型

hashmap

pojo自定义

===========================================================

## 目录结构和步骤

===========================================================

### 目录结构

* db.properties【连接信息】

jdbc.driver

jdbc.url

jdbc.username

jdbc.password

* log4j.properties【日志文件】

日志级别：

log4j.rootLogger=DEBUG，info，error，stdout

log4j.appender.stdout=org.apche.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apahce.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5 [%t] -%m%n

* SqlMapConfig.xml【核心配置文件】
* xxxMapper.java【映射接口】
* xxxMapper.xml【映射文件】
* POJO.java【DTO】

### 步骤

* 导包

mybatis：核心包

cglib：动态代理包

日志

log4j

log4j-api

log4j-core

slf4j-api

slf4j-log4j12

* SqlMapConfig.xml配置连接
* 编写xxxMapper.java接口
* 编写xxxMapper.xml映射
* 编写xxxMapperImpl.java接口实现类
* 编写POJO【resultType=POJO全类名】
* SqlMapConfig.xml加载映射
* 测试

===========================================================

需求：

根据用户Id（主键）查询用户信息

根据用户名称模糊查询用户信息

添加、删除、更新用户

===========================================================

## SqlSession

===========================================================

使用范围

①SQLSessionFactoryBuilder【构建SqlSessionFactory】【单例模式】

②SqlSessionFactory【构建SqlSession】【单例模式】

String resoure = “SqlMapConfig.xml”;

InputStream inputStream = Resoures.getResourceAsStream(resoure);

SqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

③SqlSession【面向用户的接口】

SqlSession = sqlSessionFactory.openSession();

crud操作：

selectOne、selectList

insert、delete、update

线程不安全

实现类除了接口中的操作数据库的方法外还有其他数据域属性

应用场合：方法体类【局部变量】

===========================================================

## 开发DAO的两种方法

===========================================================

### 原始DAO开发方法【DAO接口和DAO实现类】

思路：

Dao接口

Dao实现类

向Dao实现类中中注入SqlSessionFactory

构造函数注入

在方法体内通过SqlSessionFactory创建SqlSession

interface UserDao{

User findUserById(int id);

void insertUser(User user);

void deleteUser(int id);

}

class UserDaoImpl{

private SqlSessionFactory factory;

UserDaoImpl(SqlSessionFactory factory){

this.factory = factory;

}

User findUserById(int id){

SqlSession session = factory.openSession();

T t = session.selectOne(“ns.findUserById”,id);

session.commit();

session.closse();

returen t;

}

User findUserById(String name){

SqlSession session = factory.openSession();

List<T> list=session.selectList(“ns.findUserByName”,name);

session.commit();

session.closse();

returen list;

}

void insertUser(User user){

SqlSession session = factory.openSession();

session.selectOne(“ns.insertUser”,user);

session.commit();

session.closse();

}

void deleteUser(int id){

SqlSession session = factory.openSession();

session.selectOne(“ns.deleteById”,id);

session.commit();

session.closse();

}

}

总结：【存在问题】

① Dao接口实现类中存在大量模板方法

② 调用SqlSession方法时将statement的id硬编码

③ 调用SqlSession方法时传参，由于SqlSession使用泛型，即使变量类型传入错误，在编译阶段不报错，不利于开发

### Mybatis的mapper接口【相当于DAO接口】代理模式

#### 开发规范

① mapper.java接口的类全名 == mapper.xml中namespace

即：namespace ==“全类名”

② mapper.java接口的方法名== mapper.xml中的id

③ mapper.java接口的方法输入参数类型 == mapper.xml中的parameterType

④ mapper.java接口的方法返回值类型 == mapper.xml中的resultType

#### 步骤

* 编写mapper.xml映射文件

<mapper namespace=”com.itcast.mybatis.mapper.UserMapper”>

<selsect

id = “findUserById”

paramterType=”int”

resultType=”com.itcast.mybatis.po.Use”

>

select \* from user where id=#{value}

</select>

</mapper>

* 加载映射文件

<mappers>

<mapper resource=”../UserMapper.xml”>

<mapper url=”../UserMapper”>

<mapper class=”UserMapper全部类”>

<package name=”UserMapper包名”>

</mappers>

* 编写mapper接口【开发规范】

interface UserMapper{

User findUserById(int id){}

}

* Mybatis自动生成mapper接口的代理实现对象

#### 实现

session = sqlSessionFactory.openSession();

UserMapper userMapper = session.getMapper(UserMapper.class)

User user = userMapper.findUserByI d();

session.commit();

session.close();

#### 疑惑解答

①代理对象内部调用selectOne或selectList如何实现

mapper方法返回单个pojo对象，代理对象内部通过SelectOne()查询

mapper方法返回集合对象，代理对象内部通过SelectList()查询

List<User> findUserByName(String name)【selectList】

User findUserByName(String name)【seleteOne】【报错】

②接口方法参数只能有一个，系统是否不利于拓展维护

系统框架中Dao层代码被业务层公用

即使用包装类型pojo满足不同业务方法的需求

注意：

持久层方法的参数可以使用包装类型，map

Service方法中不建议使用包装类型【不利于业务层拓展开发】

===========================================================

## mapper.xml【配置SQL语句 输入/输出映射】

===========================================================

命名：xxxMapper.xml

格式：SQL语句不需要

<?xml version=”1.0” encoding=”utf-8”>

<!docement mapper ...>

<mapper namespace=””>

【命名空间，使用Mapper代理方法开发，有其他重要作用】

<select id = “” parameterType=”类全名” resultType=””>

【id标识映射文件中的sql，statement的id】

【SQL语句封装到mappedStatement对象中】

【resultType结果集的单条数据类型】

select \* from user where id = 1

select \* FROM user where id = #{参数名}【输入参数】

【占位符：输入参数类型是简单类型，参数名可以任意】

</select>

</mapper>

SqlSession sqlSession = sqlSessionFactory.openSession();

XXX xxx = sqlSession.selectOne(statement, parament)

statement = namespace.id

parament = paramentType对应参数

返回对象与resultType一致

sqlSession.commit();

sqlSession.close();

### 模糊查询

select \* from user where username like ‘%小明%’

<select id parameterType resultType>

select \* from user where username like ‘%${value}%’

</select>

【输入参数类型是简单类型，${}只能是Value】

sqlSession.selectList(statement, “小明”)

总结：

parameterType：输入参数类型

resultType：输出结果类型

#{}：标识占位符

${}：表示拼接符号，会引起SQL注入【不建议使用】

selectOne()：查询一条记录进行映射

selectList()：查询多条记录进行映射，也可以表示一条记录

查询多条记录使用selectOne()会报错

===========================================================

## SqlMapConfig.xml

===========================================================

### properties【属性】

需求：数据库连接参数单独配置在db.properties

优点：便于管理与修改

加载外部文件db.properties

<properties resource=”db.properties”>

<propertie name value></propertie>

</properties>

获取属性值【${}】

${jdbc.driver}

### 注意：加载属性的顺序

① properties元素体内定义的属性

② properties元素resource或url加载的属性，且覆盖同名属性

③ parameterType【其他标签】传递的属性，且覆盖同名属性

不再properties元素体内添加任何属性

属性名有一定的特殊性：xxx.xxx 或xxx-xxx

### settings【配置参数】

百度查阅相关信息

### typeAliases【重点：类型别名】

定义输入输出映射类型的别名

statement：

parameterType

resultType

Mybatis支持的别名

|  |  |
| --- | --- |
| 别名 | 映射类型 |
| \_x | byte  long  short  int  double  float  boolean  Integer |
| 首字母小写 | String  Byte  Long  Short  Integer  Double  Float  Boolean  Date  BigDecimal |
| decimal | BigDecimal |

#### 自定义别名

单个

<typeAliasses>

<typeAlias type alias />

</typeAliasses>

批量【根据包名package】

<typeAliasses>

<package name>

<package name>

<package name>

</typeAliasses>

自动扫描表中的po类，自动定义别名【类名（首字母大写或小写】

多个包，多个定义

### typeHandlers【类型处理器】

完成JDBC类型和Java类型的转换

|  |  |  |
| --- | --- | --- |
| 类型处理器 | Java类型 | JDBC类型 |
| BooleanTypeHandler | Bealean,bealean | 布尔值 |
| Byte | Byte,byte | 数字或字节 |
| Short | Short,short | 数字或短型 |
| Integer | Integer,int | 数字或整型 |
| Long | Long,long | 数字或长整型 |
| Float | Float,float | 数字或单精度浮点型 |
| Double | Double,double | 数字或双精度浮点型 |
| BigDecimal | BigDecimal | 数字或十进制小数 |
| String | String | Char或Varchar |
| Clob | String | Clob或longVarchar |
| NString | String | NVarchar或NChar |
| NClob | String | NClob |
| ByteArray | byte[] | 字节流 |
| Blob | byte[] | Blob,longVarBinary |
| Date | Date(java.util) | Timestamp |
| DateOnly | Date(java.util) | Date |
| TimeOnly | Date(java.util) | Time |
| SqlTimestamp | Timestamp | Timestamp |
| SqlDate | Date(java.sql) | Date |
| SqlTime | Time(java.sql) | Time |
| Object | 任意 | 其他或未指定类型 |
| Enum | Enumeration | Varchar任意兼容的字符串类型，作为代码存储而不是索引 |

### objectFactory【对象工厂】

### plugins【插件】

### environments【环境集合属性对象】

#### environment【环境子属性对象】

#### transactionManager【事务管理】

#### dataSource【数据源】

<environments default=”development”>

<environment id = “”>

<transactionManager type=”JDBC”>【事务管理】

<dataSource type=”POOLED”>【数据源连接池】

【

UNPOOLED：不使用连接池

POOLED：Mybatis提供默认的连接池

JNDI：通过Web容器（Tomcat，jboos）中获取数据源

】

<property name = “driver/url/username/password” value=”${}”/>

</dataSource >

</environment>

</environments>

### mappers【映射器】

#### mapper

* + resource
  + url
  + class
  + package

<mappers>

#### 单个加载

<mapper resource />

<mapper url=”file:///D:\\\”>

<mapper class=”mapper接口全类名”/>

规范：

接口和映射文件在同一目录

类名==映射文件名【mapper.java mapper.xml】

#### 批量加载

<package name=””>

规范：

接口和映射文件在同一目录

类名==映射文件名【mapper.java mapper.xml】

原理：mapper的代理方法

</mappers>

===========================================================

## 输入映射

===========================================================

指定输入参数类型：parameterType【简单类型、hashmap、pojo的包装类型】

### 传递POJO的包装对象

需求：完成用户信息的综合查询

传入查询条件【用户信息，其他信息（商品，订单）】

定义包装类型pojo

使用自定义的包装类型的pojo【包装查询条件】

Vo：ViewObject

UserQueryVo{

private:

// 用户信息

UserCustomer userCustomer;

// 其他信息

...

set/get();

}

拓展类：扩充User【逆向工程】

UserCustomer extends User{

}

映射文件

定义用户综合信息查询【包装类对象的属性】

===========================================================

## 输出映射

===========================================================

### resultType

查询的列名和pojo的属性名

一致时，映射成功

全部不一致时，没有创建pojo对象

部分一致是，创建pojo对象

输出简单类型

需求：用户信息综合查询列表总数

查询结构集只有一行一列时，可以使用简单类型

输出pojo和pojo列表

指定的resultType一样

返回类型不一样

单个pojo对象

pojo集合对象

### resultMap

高级的输出结果映射

查询的列名和pojo的属性名不一致

通过定义resultMap对列名和pojo属性名之间作映射关系

<mapper namepace>

<resultMap 【自定义resultMap输出类型】

type【pojo类型】

id【resultMap 标识】

>

<id【查询结果集的唯一标识】

column【唯一标识的列】

property【pojo属性】

/>

<result

column【普通列】

property【映射pojo属性】

/>

</resultMap>

<select paramterType resultMap=”namespace:id”>

</mapper>

===========================================================

## 动态sql

===========================================================

Mybatis核心：

对sql语句灵活操作，通过表达式进行判断，对SQL进行灵活拼接，组装

需求：对查询条件进行判断，如果输入参数不为空才进行拼接

用户信息综合查询列表

用户查询列表总数

<select id parameterType>

Select \* from user

where

user.sex = #{userCustom.sex}

and

user.username like ‘%userCustom.username%’

</select>

【方法一】

where 1=1

<if test=” userCustom!=null ”>

<if test=” userCustomer.sex!=null and userCustom.sex!=’’ ”>

and user.sex = #{userCustom.sex}

</if>

<if test=” userCustomer.username!=null and userCustom.username!=’’ ”>

and user.username like ‘%userCustom.username%’

</if>

</if>

【方法二】

<where>【where可以自动去掉条件中的第一个and】

<if test=” userCustom!=null ”>

<if test=” userCustomer.sex!=null and userCustom.sex!=’’ ”>

and user.sex = #{userCustom.sex}

</if>

<if test=” userCustomer.username!=null and userCustom.username!=’’ ”>

and user.username like ‘%userCustom.username%’

</if>

</if>

<where>  
===========================================================

## sql片段

===========================================================

需求：将动态SQL判断代码块抽取出来，组成SQL片段，提高代码利用率

### 定义

<sql id>

...

</sql>

注：

一般基于单表定义SQL片段【可重用性才高】

不要<where>

### 引用

<where>

<include refid></include>

</where>

### foreach【遍历list或数组】

select \* from user where id=1 or id=4 or id=10

select \* from user where id in(1,4,10)

<foreach

collection【遍历对象】

item【每次遍历对象】

open【开始遍历时拼接字符串】

close【结束遍历时拼接字符串】

separator【遍历连个对象的拼接】

></foreach>

<foreach

collection=”ids”

item=”id”

open=”And (”

close=”)”

separator=”or”

>

id=#{id}

</foreach>

结果：select \* from user where and (id=1 or id=4 or id=10)

<foreach

collection=”ids”

item=”id”

open=”And id in(”

close=”)”

separator=”,”

>

#{id}

</foreach>

结果：select \* from user where and id in(1,4,10)

===========================================================

## 注解

===========================================================

@select

@update

@insert

@delete

@xxxProvider

type=xxx.class

method=”update”

xxx.class{

String update(){

return new SQL(){{

UPDATE();

WHERE();

...

}}

}

}

===========================================================

## 用户增删改查

===========================================================

<insert id parameterType=’..po.XXX” >【输入参数类型：POJO】

insert into user(username...) value(#{}...)

【#{}中指定pojo的属性名，接收到pojo对象的属性值，Mybatis通过OGNL获取对象属性值】

</insert>

XXX xxx = new XXX();

xxx.setXXX();

sqlSession.insert(statement, xxx);

主键返回

自增主键返回

mysql自增主键：执行insert提交之前，自动生成一个自增主键

通过Mysql函数【LAST\_INSERT\_ID()】获取刚插入记录的自增主键【在insert之前调用】

<insert id parameterType=’..po.XXX” >【输入参数类型：POJO】

<selectKey

keyProperty=”id” 【将主键值设置到parameterType指定对象的属性】

order=”AFTER”

resultType=”java.lang.Interger”

>

select LAST\_INSERT\_ID();

</selectKey>

insert into user(username...) value(#{}...)

</insert>

xxx.getId();

非自增主键返回

使用Mysql的uuid()函数生成主键

① 主键类型String，长度35

② 先通过uuid()查询主键，将主键输入到sql语句中

③ 执行uuid()语句顺序相对于insert语句之前

insert id parameterType=’..po.XXX” >【输入参数类型：POJO】

<selectKey

keyProperty=”id” 【将主键值设置到parameterType指定对象的属性】

order=”BEFORE”

resultType=”java.lang.Interger”

>

select uuid();

</selectKey>

insert into user(id,username...) value(#{},#{}...)

</insert>

Oracle 序列生成主键：

<selectKey

keyProperty=”id” 【将主键值设置到parameterType指定对象的属性】

order=”BEFORE”

resultType=”java.lang.Interger”

>

select 序列名.nextval();

</selectKey>

sqlSession.insert()

<delete id parameterType>

delete from user where id = #{id}

</delete>

sqlSession.delete()

<update id parameterType=”POJO”>

update user set username=#{username}... where id=#{id}

</update>

sqlSession.update()

===========================================================

## 高级映射

===========================================================

### 订单商品数据模型分析

数据模型分析思路

① 每张表记录的数据内容

分模块对每张表的内容进行熟悉【系统需求（功能）分析】

② 每张表的字段设置

非空字段，外键字段

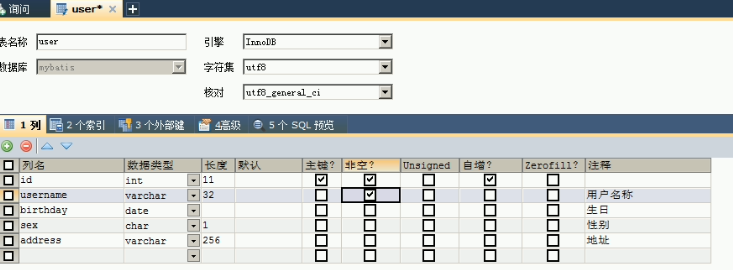
③ 数据库级别表与表之间的关系

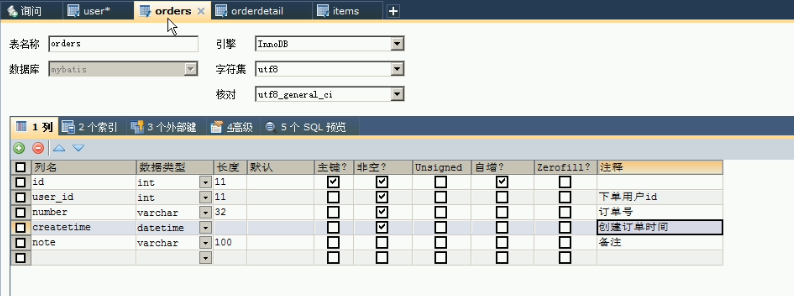
外键关系

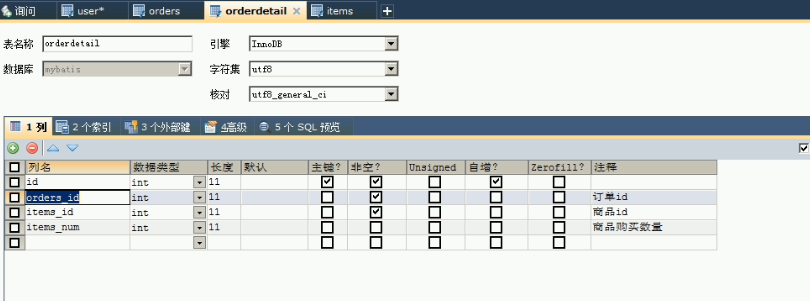
④ 表与表之间的业务关系

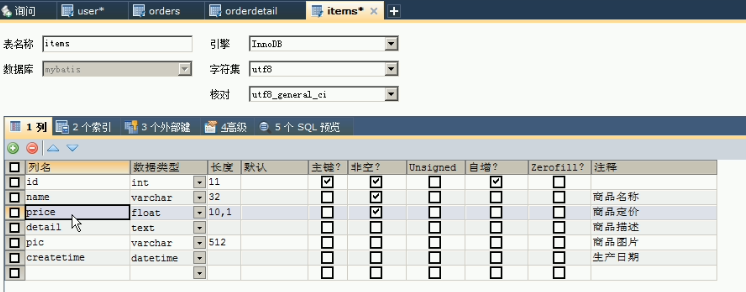
分析表与表之间的业务关系是一点要建立在某个业务意义基础啊上分析

数据模型的分析









用户表user

id

username：用户名

birthday：生日

sex：性别

address：地址

记录用户信息

订单明细表orderdetail

id

orders\_id订单id

items\_id商品id

items\_num商品购买数量

记录订单购买的商品信息

商品表items

id

name：名称

price：价格

detail：描述

pic：图片

createtime：生成日期

商品的信息

订单表orders

id

user\_id：外键

number：订单号

createtime：创建订单时间

note：备注

记录用户创建的订单信息

user\_id

items\_id

orders\_id

1:n

1:1

1:n

1:1

1:n

1:1

n:n

n:n

### 业务关系

user和orders

user-->orders：一个用户创建多个订单【一对多】

orders-->user：一个订单由一个用户创建【一对一】

orders和orderdetail

orders-->orderdetail：一个订单包括多个明细【一对多】

orderdetail-->order：一个明细只能包括在一个订单中【一对一】

orderdetail和items

orderdetail-->items：一个明细对应一个商品【一对一】

items-->orderdetail：一个商品对应过个明细【一对多】

orders和items

通过orderdetail建立关系

orders-->items：orders-1:n->orderdetail-1:1->items【一对多】

items-->orders：items-1:n->orderdetail-1:1->order【一对多】

user和items【多对多】

### 自动映射 autoMapping

### 一对一查询

#### 需求

查询订单信息，关联查询创建订单的用户信息

#### sql语句

确定查询的主表【orders】外键user\_id

确定查询的关联表【user】

关联查询，使用内连接？还是外连接？

内连接：

左外连接：

右外连接：

外键关联查询用户表只能查询一条记录，使用内连接？【最后的查询orders的记录数不变】

SELECT orders.\*,

user.username,

user.sex,

user.address

FROM orders,USER

WHERE orders.user\_id = user.id;创建pojo

sql查询的结果集映射到pojo

原始的pojo

Ordres

private：

Integer id

Integer userId

String number

Date createtime

String note

拓展新的pojo：创建新的pojo继承po类

【原则：包含越多，新建越少，越好】

OrderCustom extends Orders

private

String username;

String sex;

String address;

#### mapper.xml

<mapper namespce=”...OrdersMapperCustom”>

<select id=”findOrdersUser” resultType=”...OrderCustom”>

select ordres.\*,

user.username,

user.sex

user.address

from orders,user

where orders.user\_id = user.id;

</select>

</mapper>

#### OrdersMapperCustom.java【interface】

List<OrderCustom> findOrdersUser() throws Exception;

OrdersMapperCustom ordersMapperCustom =

sqlSession.getMapper(OrdersMapperCustom);

List<OrderCustom> list = ordersMapperCustom.findOrdersUser();

#### resultMap

多个列组成唯一标识<id>配置多个id

Orders

private：User

<id column property>

<result column property>

<association 【配置关联】

property【关联属性名】

javaType【关联】

>

<id />

<result />

</association>

#### 总结

resultMap和resultType区别

resultMap

自定义查询映射pojo的属性

关联查询

实现延迟加载

### 一对多查询

#### 需求

查询订单及其订单明细的信息

#### sql语句

主查询表：订单表

关联查询表：订单明细表

SELECT

orders.\*,

user.username,

user.sex,

user.address,

orderdetail.`orders\_id`,

orderdetail.`items\_id`,

orderdetail.`items\_num`

FROM

orders,

USER,

orderdetail

WHERE

orders.user\_id = user.id

AND

orderdetail.orders\_id = orders.id;



要求：

对Orders映射不能出现重复记录

Orders.java中添加List<orderDetail> orderDetails属性

mapper.xml

<resultMap>

<collect property oftype extends>

<id/>

<result />

</collect>

</resultMap>

mapper.java

### 多对多

需求

查询用户及用户购买信息

sql语句

主表：用户表

关联表【间接关联】

orders

orderdetail

items

SELECT

orders.\*,

user.username,

user.sex,

user.address,

orderdetail.id orderdetail\_id,

orderdetail.items\_id,

orderdetail.items\_num,

orderdetail.orders\_id,

items.`name` items\_name,

items.`detail` items\_detail,

items.`price` items\_price

FROM

orders,

USER,

orderdetail,

items

WHERE

orders.user\_id = user.id

AND

orderdetail.orders\_id = orders.id

AND

orderdetail.items\_id = items.id;

映射思路

将用户信息映射到user中

User中添加订单列表属性List<Orders> ordersList

Orders中添加订单明细属性List<OrderDetail> orderdetails

OrderDetial中添加Items属性

mapper.xml

<resultMap type=”User” id=””>

【用户信息】

<id>

<result>

【订单信息：一个用户多个订单】

<collect property=”ordersList” ofType=”Orders”>

<id>

<result>

【订单明细：一个订单多个订单明细】

<collect propetry=”orderdetails” ofType=”Orderdetail”>

<id>

<result>

【商品信息：一个订单明细对应一个商品】

<asscociation property=”items” javaType=”Items”>

<id>

<result>

</asscociation>

</collect>

</collect>

</resultMap>

总结

用户购买商品信息明细清单

用户名

用户地址

商品名称

商品时间

商品数量

使用resultType将查询到的记录映射到一个拓展的pojo，功能清晰

需求2：

用户账号

用户名称

商品数量

商品明细（鼠标查看时显示明细）

使用resultMap将用户购买的商品明细列表映射到user对象中

===========================================================

## 延迟加载

===========================================================

resultMap实现高级映射【association collection实现一对一、一对多映射】

association、collection具备延迟加载的功能

核心配置开启延迟加载

lazyLoadingEnabled

默认false【所有相关联的都会一次全部加载】

aggressiveLazyLoading

默认true【一次全部加载】

false【按需加载】

<settings>

<settiong name=”lazyLoadingEnabled” value=”true” />

<settiong name=”aggressiveLazyLoading” value=”false” />

</settings>

延迟加载：

先查询单表

后查询关联表

使用association实现

订单及关联的用户信息

sql语句

select

orders.\*,

(select \* from user where orders.user\_id = user.id)

from orders

mapper.xml

①查询订单信息

select \* from orders

查询到user\_id,查询关联表

②查询关联的用户信息

通过①查询的user\_id查询用户信息

<select id resultMap>

select \* from orders

</select>

<resultMap type=”Orders” id>

【订单信息】

<id />

<result />

【用户信息延迟加载】

<association property=”user” javaType=”User”

select=”findUserById”【延迟加载的sql语句】

column=”user\_id ”【订单信息关联用户信息的列】

>

</association>

<resultMap>

<select id=”findUserById” parameterType=”int” resultType=”user”>.

select \* from user where id=#{value}

</select>

延迟加载思考

不使用Mybatis提供的association和collection中的延迟加载的功能，如何实现延迟加载？

实现方法：

定义两个mapper

1. 查询订单列表
2. 根据用户id查询用户信息

实现思路：

先查询第一个mapper方法，获取订单信息列表

在查询中（Service），按需调用第二个mapper方法查询用户信息

总结：首先简单的sql【单表或多表关联】，然后延迟加载

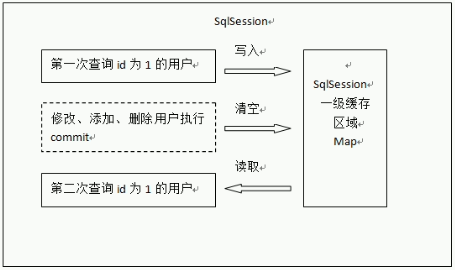
===========================================================

## 缓存

===========================================================



一级缓存



SqlSession对象

包含一个HashMap对象【用于存储缓存数据】

不同的SqlSession，互不影响

Mybatis 默认支持以及缓存，不需要配置

一级缓存应用

事务控制在Service中

在Service中包含很多mapper方法调用

Service{

开启事务，创建SqlSession

。。。【从一级缓存中取数据】

方法结束SqlSession关闭

}

二级缓存【跨SqlSession】

多个SqlSession操作同一Mapper的sql语句

多个SqlSession共用二级缓存

（1）开启二级缓存

① 核心配置SqlMapConfig.xml

<setting name=”cacheEnabled” value=”true” />

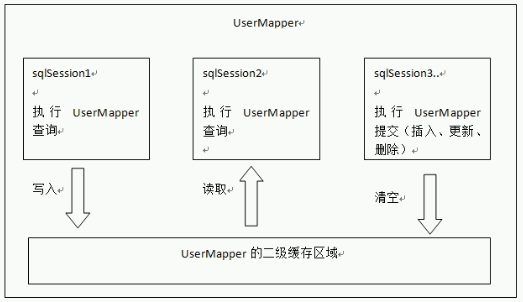
② 开启mapper.xml中的二级缓存

<mapper namespace>

<cache />

</mapper>

（2）读取数据



POJO类实现序列化接口

将缓存数据取出执行反序列化操作

二级缓存数据存储介质多种多样

一级缓存和二级缓存区别：

一级缓存

一个SqlSession一个一级缓存区域

二级缓存跨SqlSession

多个SqlSession共享一个mapper二级缓存区域【按照namespace区分】

每一个namespace的mapper都有一个二级缓存区域

如果namespace相同，mapper使用同样一个二级缓存

===========================================================

## 与Spring整合

===========================================================

===========================================================

## 逆向工程

===========================================================

===========================================================

## 日志管理

===========================================================

导包：

log4j

log4j.properties

消息级别：

info

debug

warn

error

log.ingo()

log.debug()

log.warn()

log.error()

log4j.rootLogger=info, stdout, log, errorlog

log4j.Logger=search,Test

###Console ###

log4j.appender.stdout = org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target = System.out

log4j.appender.stdout.layout = org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern = %d{ABSOLUTE} [ %t ] [ %p ]:%L - %m%n

### Log ###

log4j.appender.log = org.apache.log4j.DailyRollingFileAppender

log4j.appender.log.File = log/log.log

log4j.appender.log.Append = true

log4j.appender.log.Threshold = INFO

log4j.appender.log.DatePattern='.'yyyy-MM-dd

log4j.appender.log.layout = org.apache.log4j.PatternLayout

log4j.appender.log.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t ] %m%n

### Error ###

log4j.appender.errorlog = org.apache.log4j.DailyRollingFileAppender

log4j.appender.errorlog.File = log/errorlog.log

log4j.appender.errorlog.Append = true

log4j.appender.errorlog.Threshold = ERROR

log4j.appender.errorlog.DatePattern='.'yyyy-MM-dd

log4j.appender.errorlog.layout = org.apache.log4j.PatternLayout

log4j.appender.errorlog.layout.ConversionPattern =%-d{yyyy-MM-dd HH:mm:ss} [ %t ] %m%