Mybatis



一.Mybatis 介绍

1.1 什么是Mybatis?

MyBatis 是一款优秀的持久层框架->对数库的操作它支持自定义 SQL、存储过程以及高级映射。

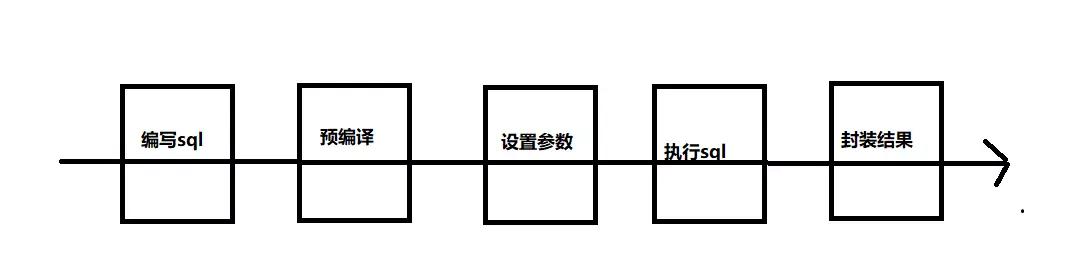
MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。 MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java

POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

1.2 为什么要使用Mybatis?

JDBC

– SQL夹在Java代码块里，耦合度高导致硬编码内伤 – 维护不易且实际开发需求中sql是有变化，频繁修改的情况多



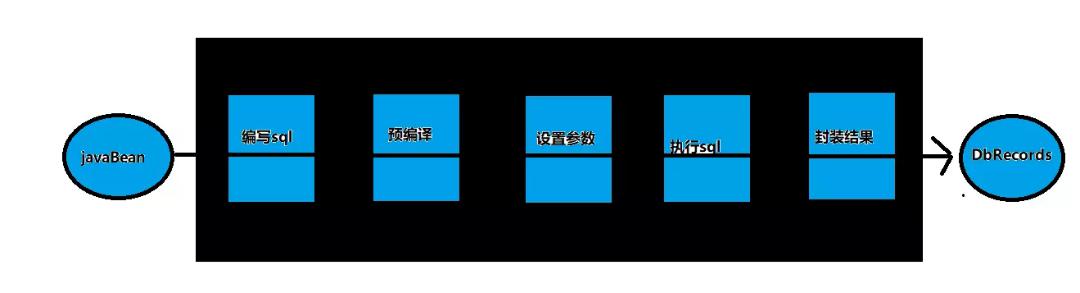
Hibernate

-半自动框架

– 内部自动生产的SQL，不容易做特殊优化。

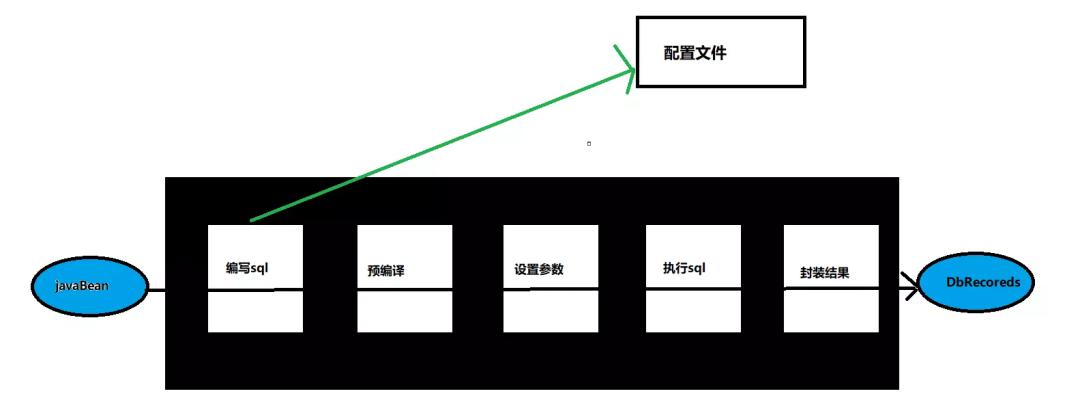
– 基于全映射的全自动框架，大量字段的POJO进行部分映射时比较困难。导致数据库性能下降。

-hql



MyBatis是一个半自动化的持久化层框架。

只需要掌握sql,后期有一些插件可以自动生成SQL



1.3 ORM 概念

对象关系映射\*\*

数据库表------>实体类

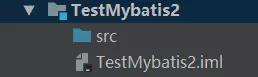
数​据库表中字段----->实体类的属性

数​据库表中字段的类型----->实体类中属性的类型

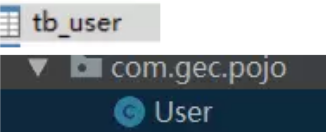
二.MyBatis 入门

1. 编写全局配置文件
2. 编写mapper映射文件
3. 加载全局配置文件，生成SqlSessionFactory
4. 创建SqlSession，调用mapper映射文件中的SQL语句来执行CRUD操作

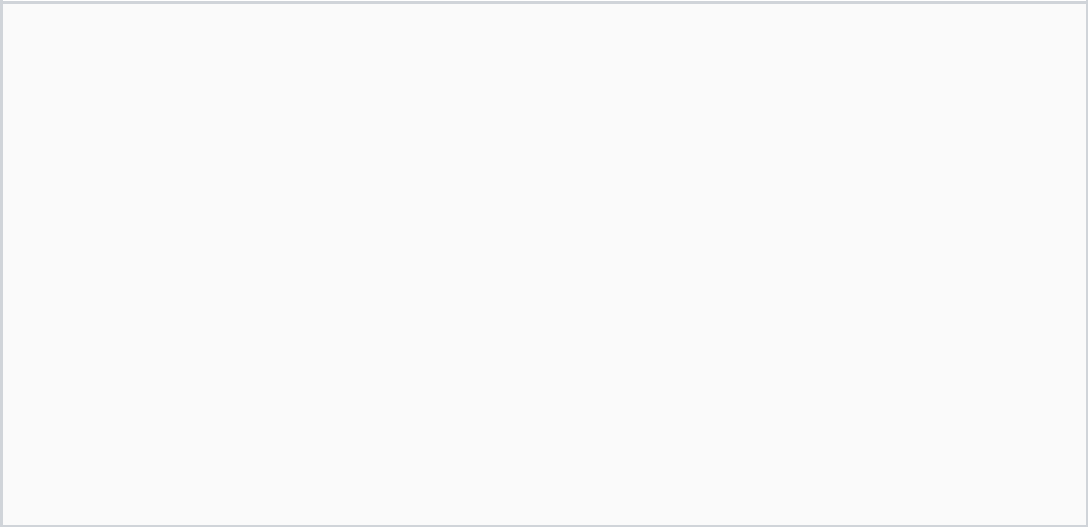
2.1 创建一个普通java 项目



2.2 根据数据表创建实体类



User



* public class User {
* private String id;
* private String account;

4private String nickName;

5private String password;

6private String roleld;

7private String no;

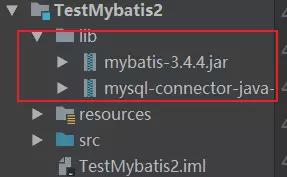
8private String sex;

9private Date createDate;

10

1. //get set toString ...
2. }

2.3 导入jar 包



2.4 创建mybatis-config.xml



* <?xml version="1.0" encoding="UTF-8" ?>

2 <!DOCTYPE configuration

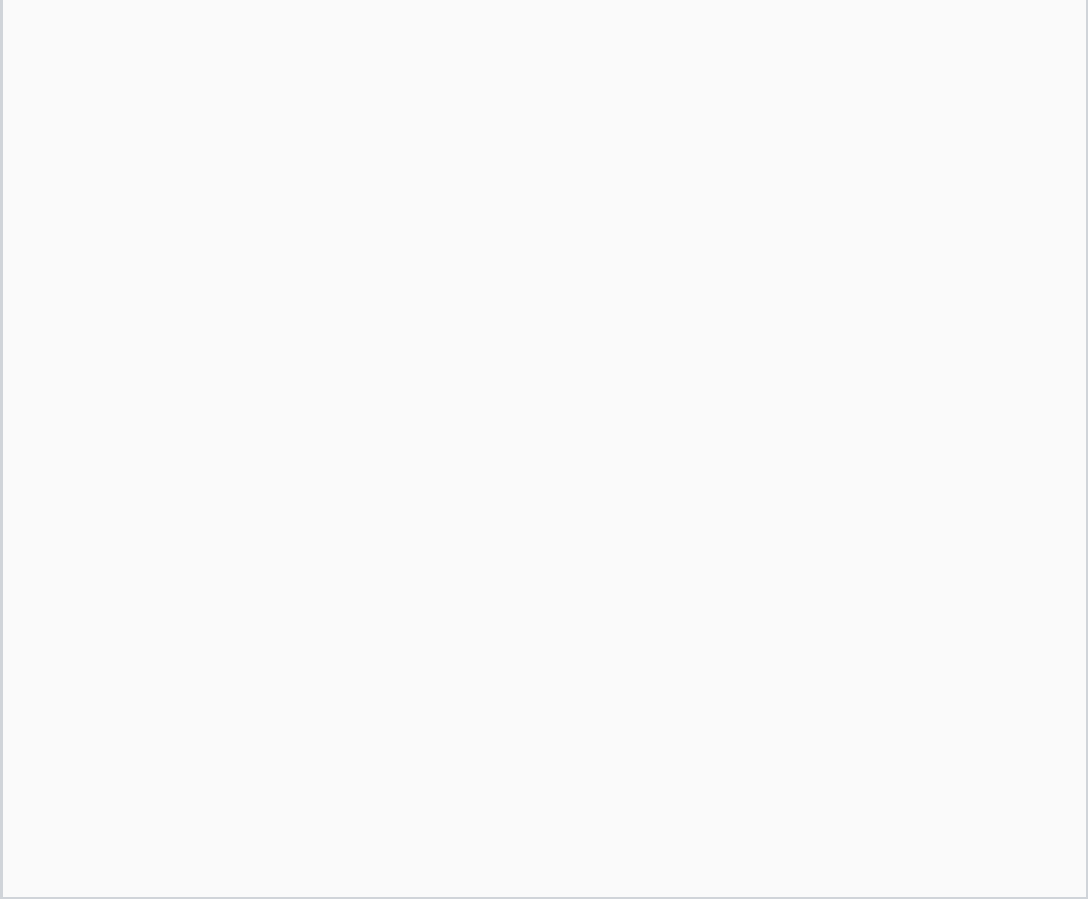
3PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

* "http://mybatis.org/dtd/mybatis-3-config.dtd">

5

* <configuration>
* <!-- 配置环境-->
* <environments default="development">
* <!-- id:数据源名称-->

1. <environment id="development">
2. <!--配置事务类:JDBC-->

1. <transactionManager type="JDBC"/>
2. <!--数据源-->
3. <dataSource type="POOLED">

15

1. <property name="driver" value="com.mysql.jdbc.Driver"/>

|  |  |
| --- | --- |
| 17 | <property name="url" |
| 18 | value="jdbc:mysql://localhost:3306/cjt?useUnicode=true&charac |
|  | terEncoding=utf-8"/> |
| 19 | <property name="username" value="root"/> |
| 20 | <property name="password" value=""/> |
| 21 |  |

1. </dataSource>
2. </environment>
3. </environments>

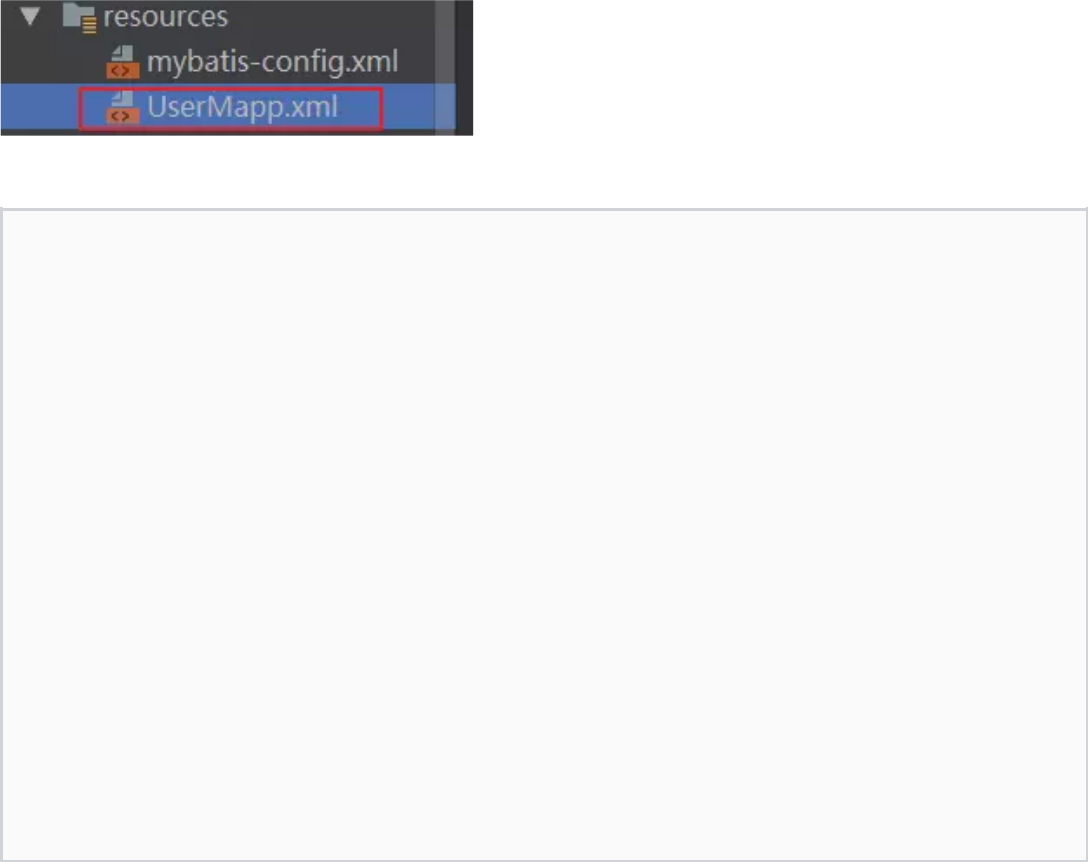
25

1. <!-- 将我们sql 文件 注册到全局配置文件中-->
2. <mappers>
3. <mapper resource="UserMapper.xml"/>
4. </mappers>

30

1. </configuration>

2.5 编写映射文件



* <?xml version="1.0" encoding="UTF-8" ?>

2 <!DOCTYPE mapper

3PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

* "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

5 <!--

6 namespace:名称空间

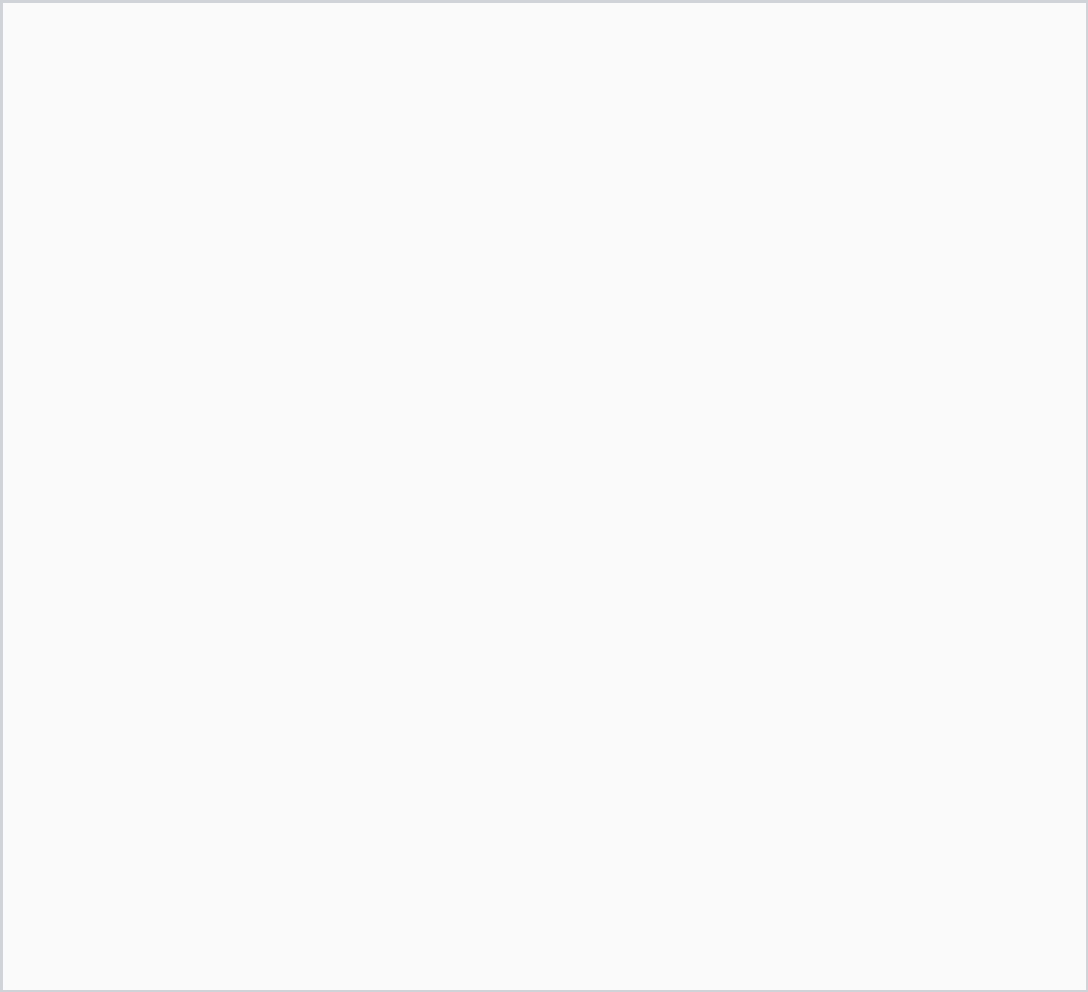
7 id:唯一标识

8 resultType:返回值类型

9 #{id}:接收传来的id 值

1. -->
2. <mapper namespace="com.gec.mapper.AA">
3. <select id="getUserById" resultType="com.gec.pojo.User">
4. select \* from tb\_user where id = #{id}
5. </select>
6. </mapper>

2.6 编写测试类

* @Test
* public void mTest1() throws IOException {
* /\*
* 根据xml配置文件（全局配置文件）创建一个SqlSessionFactory对象
* \*/
* String resources = "mybatis-config.xml";

7//使用 Resources 的工具类加载资源文件

8InputStream inputStream = Resources.getResourceAsStream(resources);

* //SqlSessionFactory对象,获取SqlSession

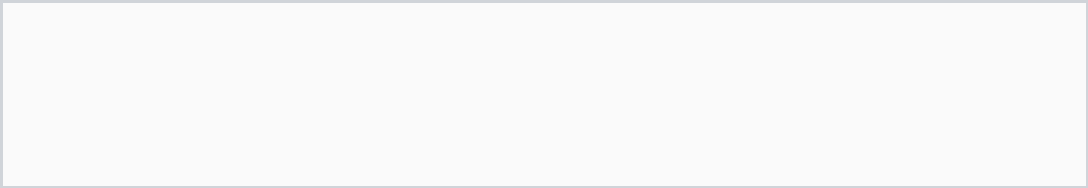
1. SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().buil d(inputStream);
2. //从 SqlSessionFactory 中获取 SqlSession,执行 SQL 命令
3. SqlSession sqlSession = sessionFactory.openSession();
4. /\*
5. 唯一标识符 namespace+id
6. sql 需要的参数
7. \*/
8. User user = sqlSession.selectOne("com.gec.mapper.AA.getUserById",12);
9. System.out.println(user);
10. System.out.println(sessionFactory);

20

1. sqlSession.close();
2. }

2.7 映入外部文件

1.配置db.properties



* jdbc.driver=com.mysql.jdbc.Driver
* jdbc.url =jdbc:mysql://localhost:3306/hotel

3 jdbc.username =root

4 jdbc.password=root

2.配置xml文件



* <?xml version="1.0" encoding="UTF-8" ?>

2 <!DOCTYPE configuration

3PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

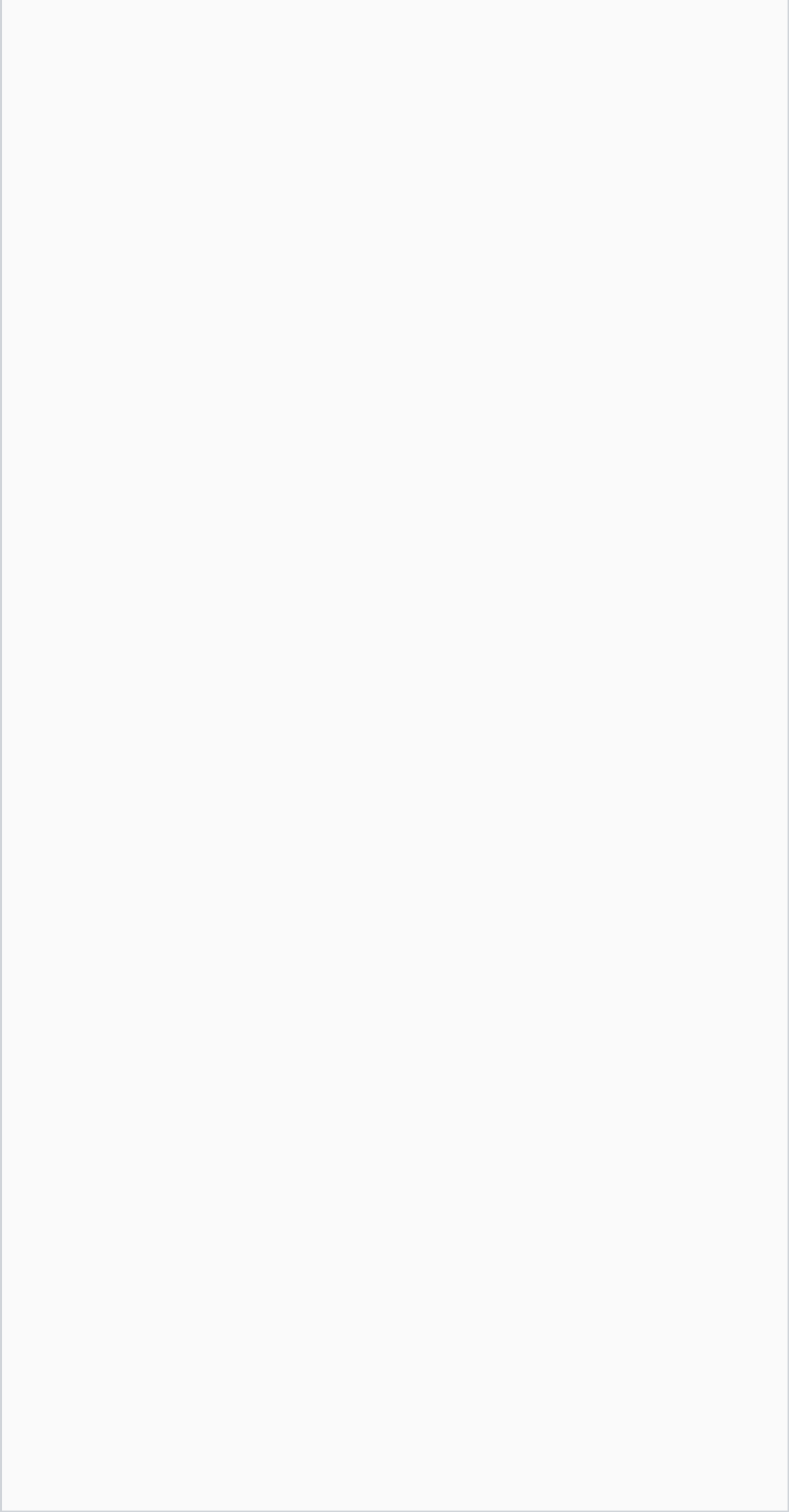
* "http://mybatis.org/dtd/mybatis-3-config.dtd">

5

6

* <configuration>
* <!--
* 1、mybatis可以使用properties来引入外部properties配置文件的内容；

1. resource：引入类路径下的资源
2. url：引入网络路径或者磁盘路径下的资源
3. -->

1. <properties resource="db.properties"></properties>
2. <!--
3. nvironments：环境们，mybatis可以配置多种环境 ,default指定使用某种环境。可以达到快速切换环境。
4. environment：配置一个具体的环境信息；必须有两个标签；id代表当前环境的唯一标识
5. transactionManager：事务管理器；

2019 type：事务管理器的类型;

1. JDBC(JdbcTransactionFactory)
2. |MANAGED(ManagedTransactionFactory)
3. 自定义事务管理器：实现TransactionFactory接口.type指定为全类名

25

1. dataSource：数据源;
2. type:数据源类型;UNPOOLED(UnpooledDataSourceFactory)

|  |  |
| --- | --- |
| 28 | |POOLED(PooledDataSourceFactory) |
| 29 | |JNDI(JndiDataSourceFactory) |

1. 自定义数据源：实现DataSourceFactory接口，type是全类名
2. -->
3. <environments default="development">
4. <environment id="development">
5. <transactionManager type="JDBC"/>

35

1. <dataSource type="POOLED">
2. <property name="driver" value="${jdbc.driver}"/>
3. <property name="url" value="${jdbc.url}"/>
4. <property name="username" value="${jdbc.username}"/>
5. <property name="password" value="${jdbc.password}"/>
6. </dataSource>
7. </environment>
8. </environments>

44

1. <!-- 5、databaseIdProvider：支持多数据库厂商的；
2. type="DB\_VENDOR"：VendorDatabaseIdProvider
3. 作用就是得到数据库厂商的标识(驱动getDatabaseProductName())，mybati s就能根据数据库厂商标识来执行不同的sql;
4. MySQL，Oracle，SQL Server,xxxx
5. -->
6. <databaseIdProvider type="DB\_VENDOR">
7. <!-- 为不同的数据库厂商起别名 -->
8. <property name="MySQL" value="mysql"/>
9. <property name="Oracle" value="oracle"/>
10. <property name="SQL Server" value="sqlserver"/>
11. </databaseIdProvider>

56

57

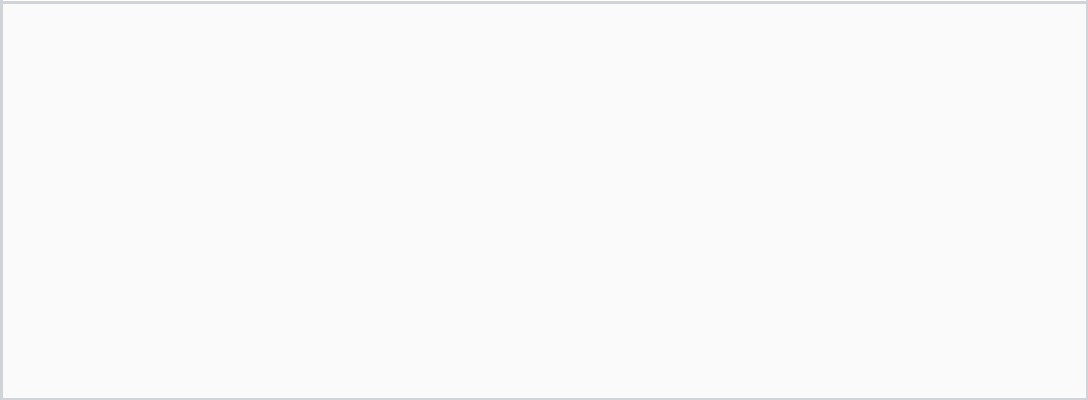
1. <mappers>
2. <mapper resource="UserMapper.xml"/>
3. </mappers>

61

1. </configuration>

2.8 xml 配置

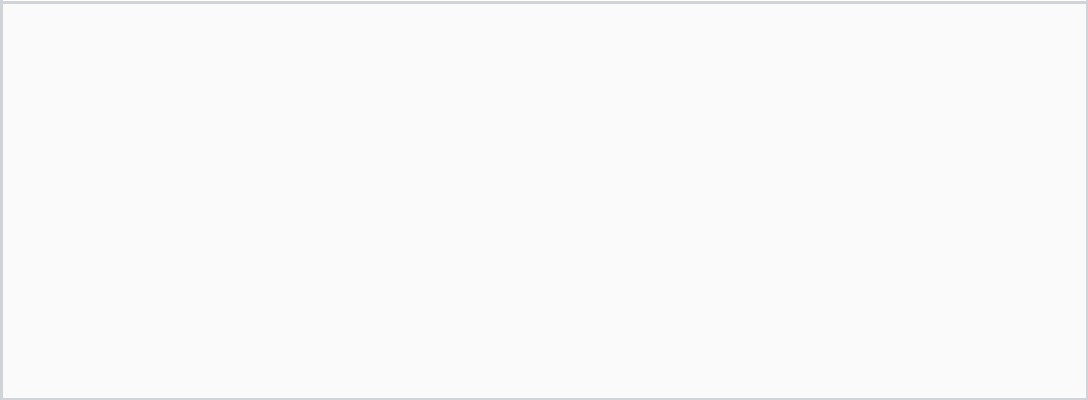
1.开启驼峰命名法



* <!--
* 2、settings包含很多重要的设置项
* setting:用来设置每一个设置项
* name：设置项名
* value：设置项取值
* -->
* <settings>
* <setting name="mapUnderscoreToCamelCase" value="true"/>

9</settings>

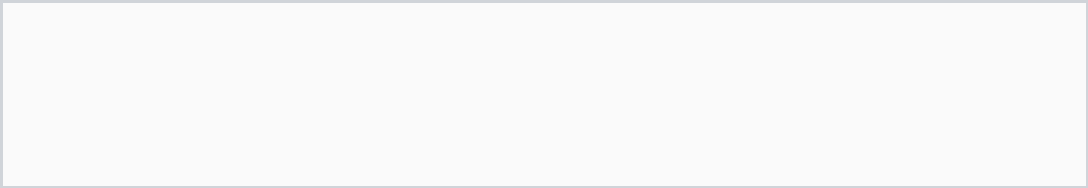
2.别名处理器



* <!-- 3、typeAliases：别名处理器：可以为我们的java类型起别名
* 别名不区分大小写
* -->
* <typeAliases>
* <!-- 1、typeAlias:为某个java类型起别名

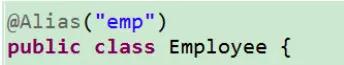
|  |  |
| --- | --- |
| 6 | type:指定要起别名的类型全类名;默认别名就是类名小写；employee |
| 7 | alias:指定新的别名 |

* -->
* <typeAlias type="com.cjt.mybatis.bean.Employee" alias="emp"/>

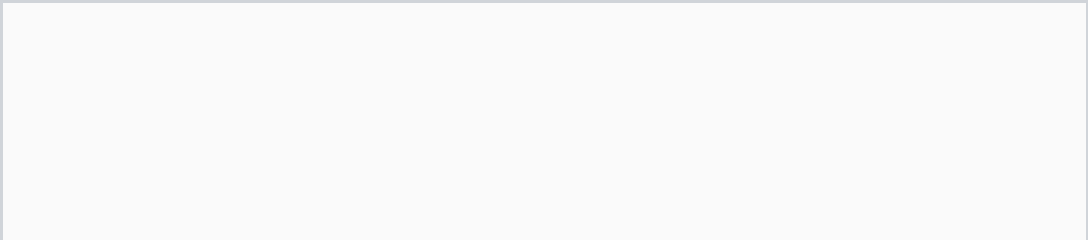


* <!-- 2、package:为某个包下的所有类批量起别名
* name：指定包名（为当前包以及下面所有的后代包的每一个类都起一个默认别名（类名写），）
* -->
* <package name="com.cjt.mybatis.bean"/>

3.@Alias注解为其指定一个别名



4.mappers

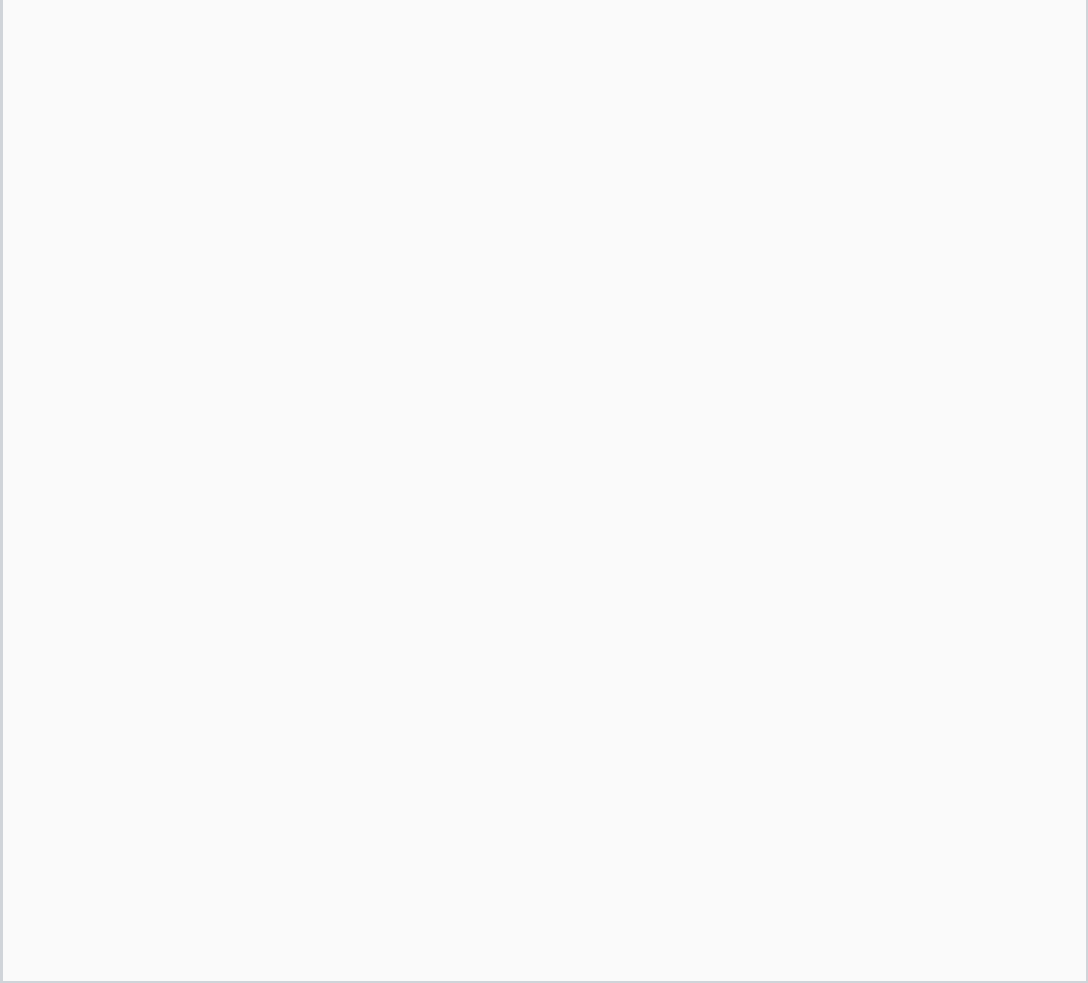


* <!-- 将我们写好的sql映射文件（EmployeeMapper.xml）一定要注册到全局配置文件（myba

tis-config.xml）中 -->

* <!-- 6、mappers：将sql映射注册到全局配置中 -->
* <mappers>

4<!--

* mapper:注册一个sql映射
* 注册配置文件

|  |  |
| --- | --- |
| 7 | resource：引用类路径下的sql映射文件 |
| 8 | mybatis/mapper/EmployeeMapper.xml |
| 9 | url：引用网路路径或者磁盘路径下的sql映射文件 |
| 10 | file:///var/mappers/AuthorMapper.xml |
| 11 |  |

1. 注册接口

|  |  |
| --- | --- |
| 13 | class：引用（注册）接口， |
| 14 | 1、有sql映射文件，映射文件名必须和接口同名，并且放在与接口同 |
|  | 一目录下； |
| 15 | 2、没有sql映射文件，所有的sql都是利用注解写在接口上; |

1. 推荐：

|  |  |
| --- | --- |
| 17 | 比较重要的，复杂的Dao接口我们来写sql映射文件 |
| 18 | 不重要，简单的Dao接口为了开发快速可以使用注解； |

1. -->
2. <!-- <mapper resource="mybatis/mapper/EmployeeMapper.xml"/> -->

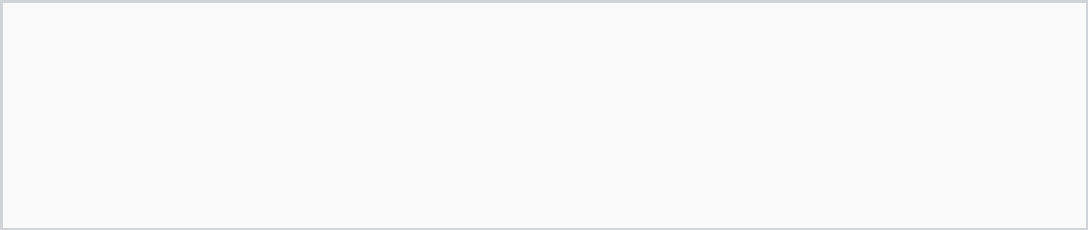
21

1. <!-- <mapper class="com.gec.mybatis.dao.EmployeeMapperAnnotation"/> -->

23

1. <!-- 批量注册： -->
2. <package name="com.cjt.mybatis.dao"/>
3. </mappers>

不推荐使用



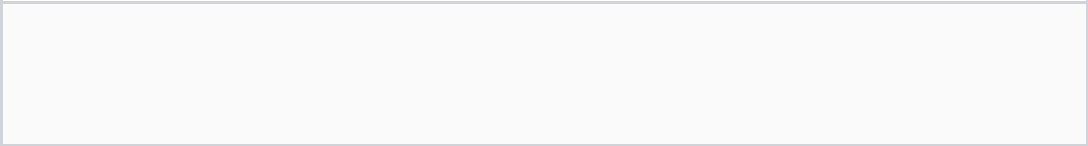
* public interface UserMapperAnnotation {

2

* @Select(" select \* from user where id = #{id}")

4public User getEmpById(Integer id);

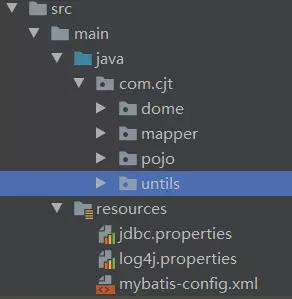
* }



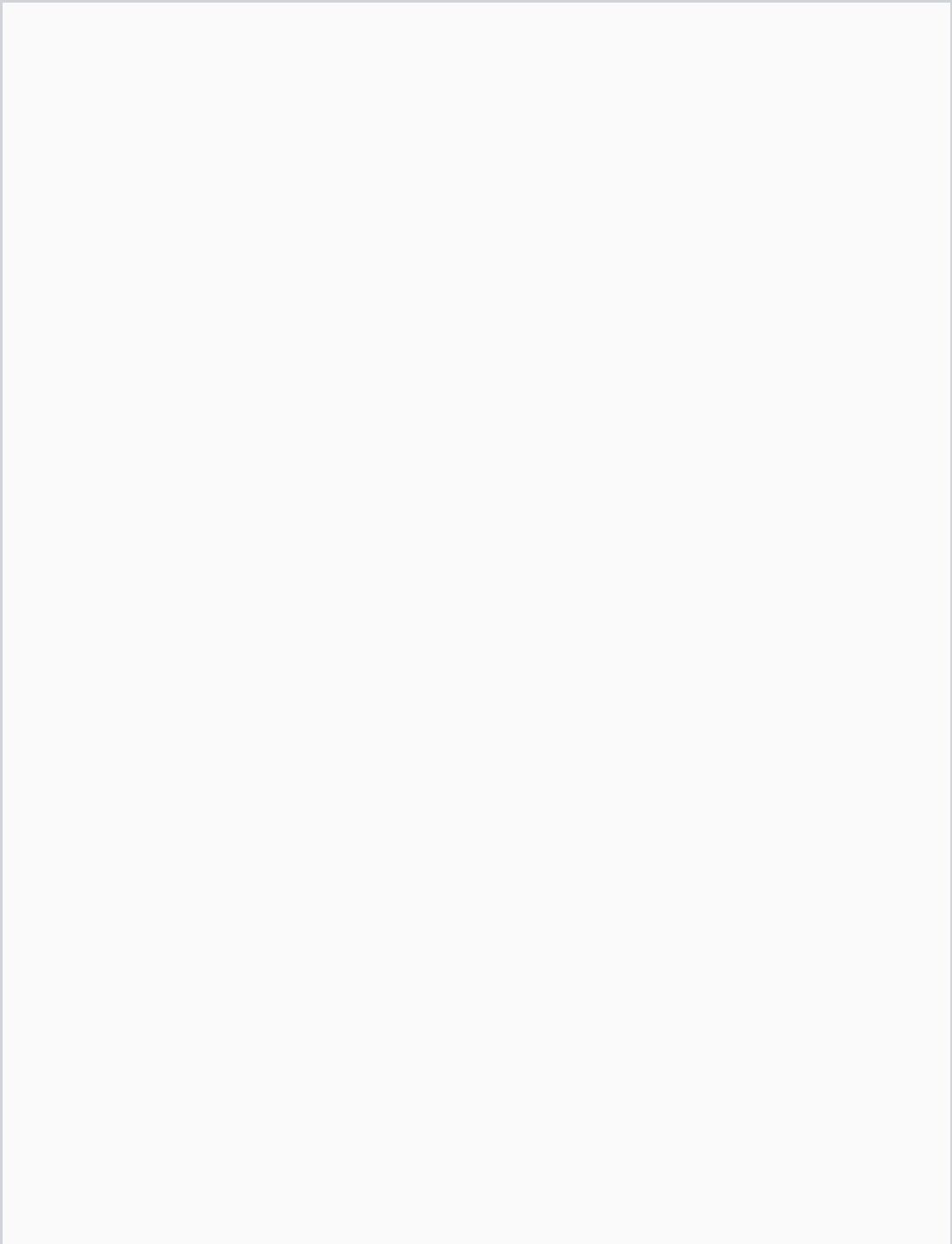
* <mappers>
* <mapper class="com.gec.mapper.UserMapperAnnotation"/> 3 </mappers>

三.MyBatis 入门(Mavean)

Mavean 工程



3.1 导入依赖



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

<!--导入MyBatis开发环境的依赖-->

<dependencies>

<!-- myBatis -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.4.5</version>

</dependency>

<!-- mysql驱动包 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.38</version>

</dependency>

<!-- Junit测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

<!--Log4J日志工具 打印运行日志用的！-->

<dependency>

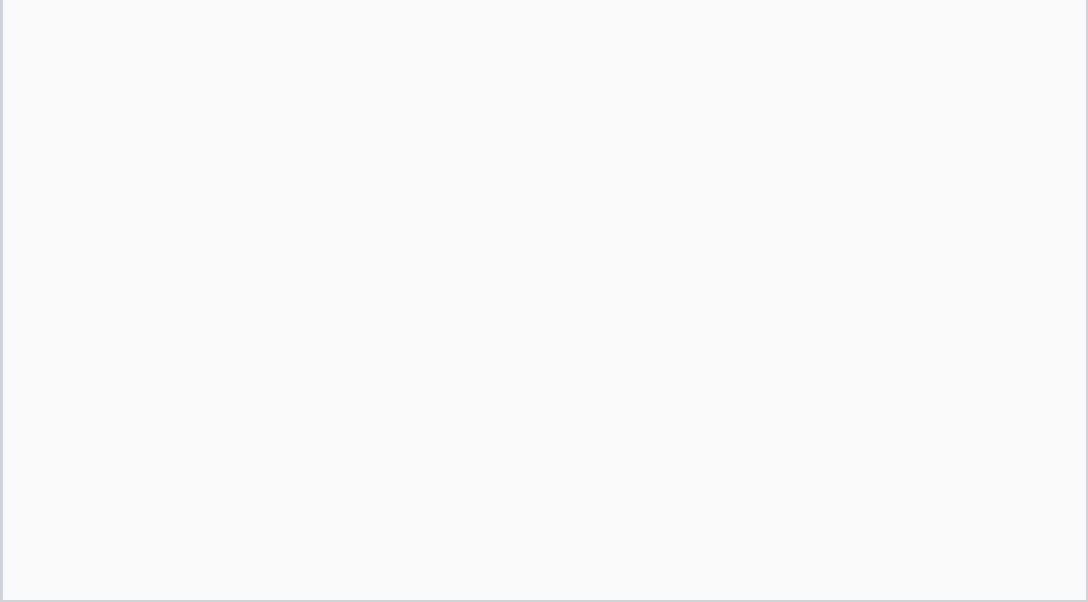
<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.14</version>

</dependency>

</dependencies>

1. <!--如果是WEB项目，那么不用创建bulid标签-->
2. <build>
3. <!--编译的时候同时也把包下面的xml同时编译进去-->
4. <resources>
5. <resource>

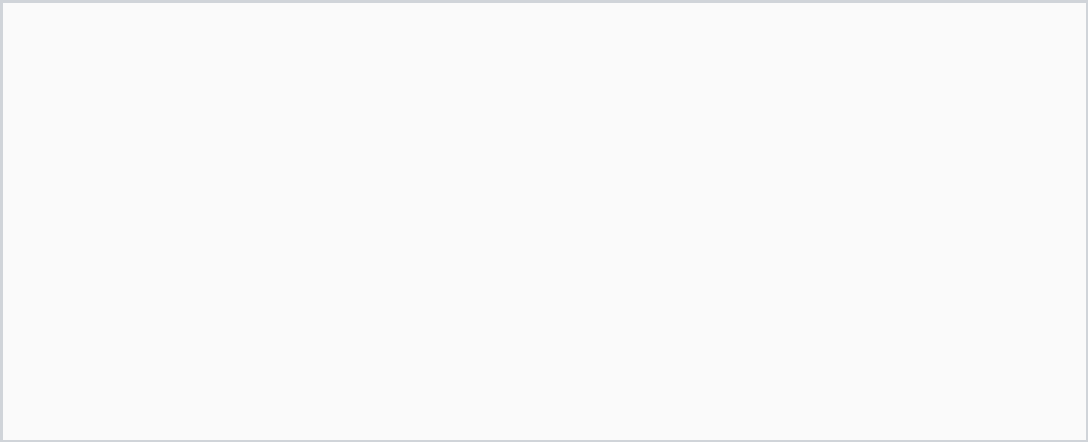
|  |  |
| --- | --- |
| 39 | <directory>src/main/java</directory> |
| 40 | <includes> |
| 41 | <include>\*\*/\*.xml</include> |
| 42 | </includes> |

1. </resource>
2. </resources>
3. </build>

46

1. </project>

3.2 添加日志配置-log4j.properties



* log4j.rootLogger=DEBUG, Console
* log4j.appender.Console=org.apache.log4j.ConsoleAppender
* log4j.appender.Console.layout=org.apache.log4j.PatternLayout
* log4j.appender.Console.layout.ConversionPattern=%d [%t] %-5p [%c] - %m%n

5

* log4j.logger.java.sql.ResultSet=INFO

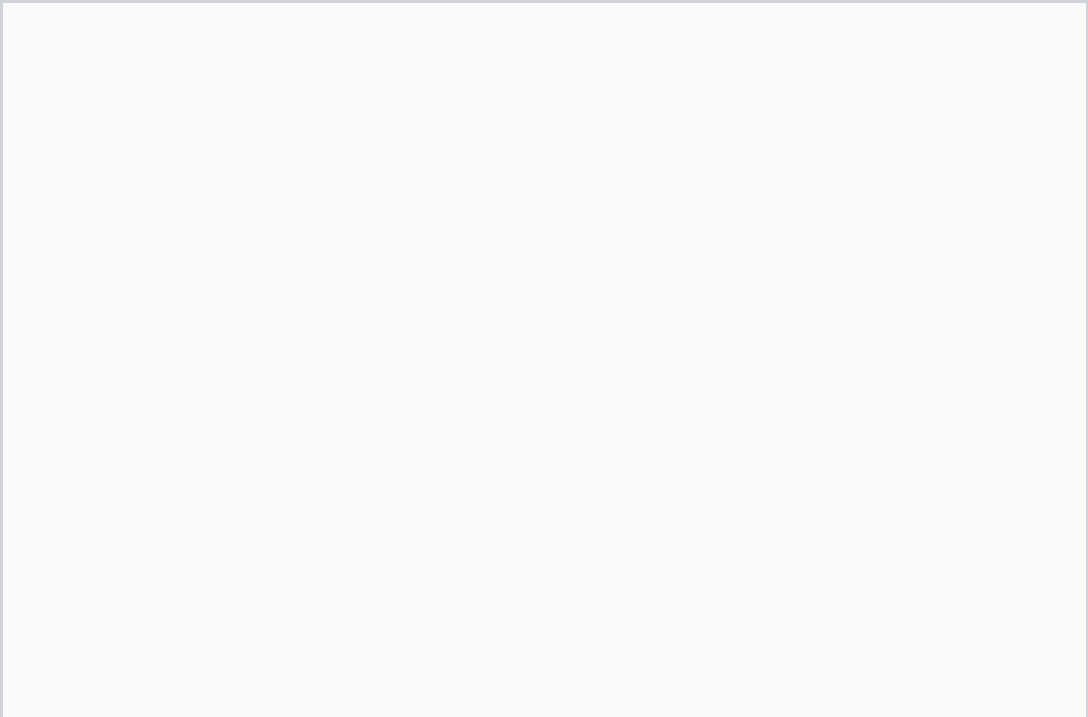
7 log4j.logger.org.apache=INFO

8 log4j.logger.java.sql.Connection=DEBUG

9 log4j.logger.java.sql.Statement=DEBUG

1. log4j.logger.java.sql.PreparedStatement=DEBUG

3.3 准备数据库



* SET FOREIGN\_KEY\_CHECKS=0;

2

* -- ----------------------------

4 -- Table structure for `user`

5 -- ----------------------------

6 DROP TABLE IF EXISTS `user`;

7 CREATE TABLE `user` (

8`id` int(11) NOT NULL AUTO\_INCREMENT,

9`username` varchar(32) NOT NULL COMMENT '用户名称',

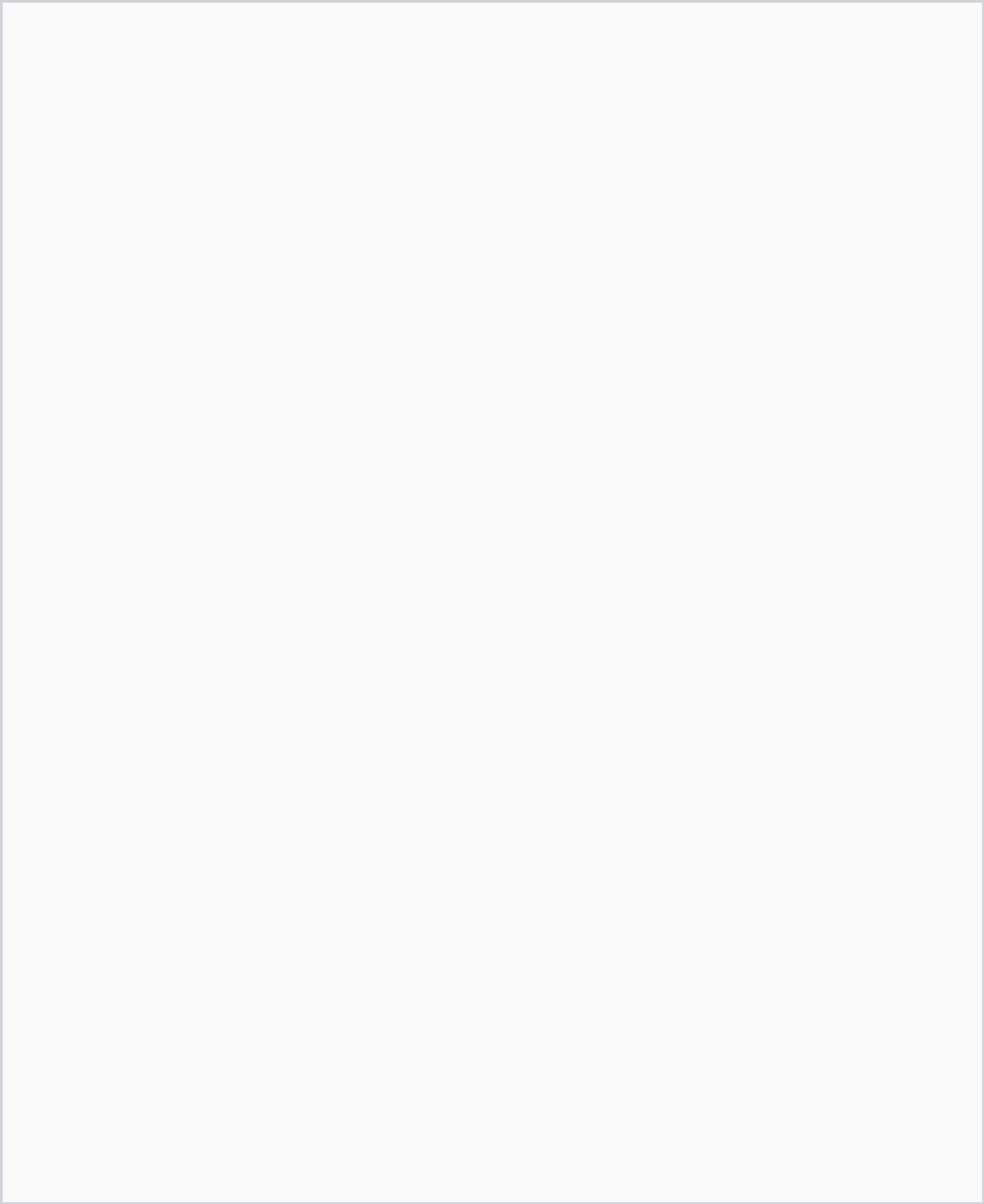
1. `birthday` date DEFAULT NULL COMMENT '生日',
2. `sex` char(1) DEFAULT NULL COMMENT '性别',
3. `address` varchar(256) DEFAULT NULL COMMENT '地址',
4. PRIMARY KEY (`id`)
5. ) ENGINE=InnoDB AUTO\_INCREMENT=27 DEFAULT CHARSET=utf8;

15

1. -- ----------------------------

1. -- Records of user
2. -- ----------------------------
3. INSERT INTO `user` VALUES ('1', '王五', '2018-09-06', '1', '四川成都');
4. INSERT INTO `user` VALUES ('10', '张三', '2014-07-10', '1', '北京市');
5. INSERT INTO `user` VALUES ('16', '张小明', '2018-09-06', '1', '河南郑州');
6. INSERT INTO `user` VALUES ('22', '陈小明', '2018-09-05', '1', '河南郑州');
7. INSERT INTO `user` VALUES ('24', '张三丰', '2018-09-13', '1', '河南郑州');
8. INSERT INTO `user` VALUES ('25', '陈小明', '2018-09-12', '1', '河南郑州');
9. INSERT INTO `user` VALUES ('26', '王五', '2018-09-05', '0', '河南郑州');

3.4 创建主配置文件：mybatis-config.xml



* <?xml version="1.0" encoding="UTF-8" ?>

2 <!DOCTYPE configuration

3PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

* "http://mybatis.org/dtd/mybatis-3-config.dtd"> 5 <configuration>

6

* <!-- 引入外部配置文件 -->
* <properties resource="jdbc.properties"/>

9

1. <!-- 环境 -->
2. <environments default="development">
3. <environment id="development">
4. <transactionManager type="JDBC" />
5. <dataSource type="POOLED">

|  |  |
| --- | --- |
| 15 | <property name="driver" value="${jdbc.driver}" /> |
| 16 | <property name="url" value="${jdbc.url}" /> |
| 17 | <property name="username" value="${jdbc.username}" /> |
| 18 | <property name="password" value="${jdbc.password}" /> |

1. </dataSource>
2. </environment>
3. </environments>

22

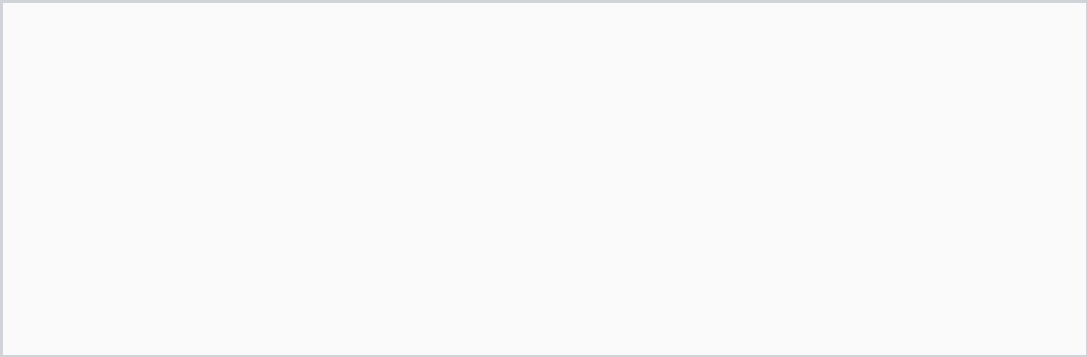
23

1. <!--映射Mapper文件-->
2. <!--引入映射文件-->
3. <mappers>
4. <mapper resource="com/bruceliu/mapper/UserMappper.xml">
5. </mapper>
6. </mappers>

30

1. </configuration>

3.5 创建实体类

* public class User {

2

* private Integer id;
* private String username;

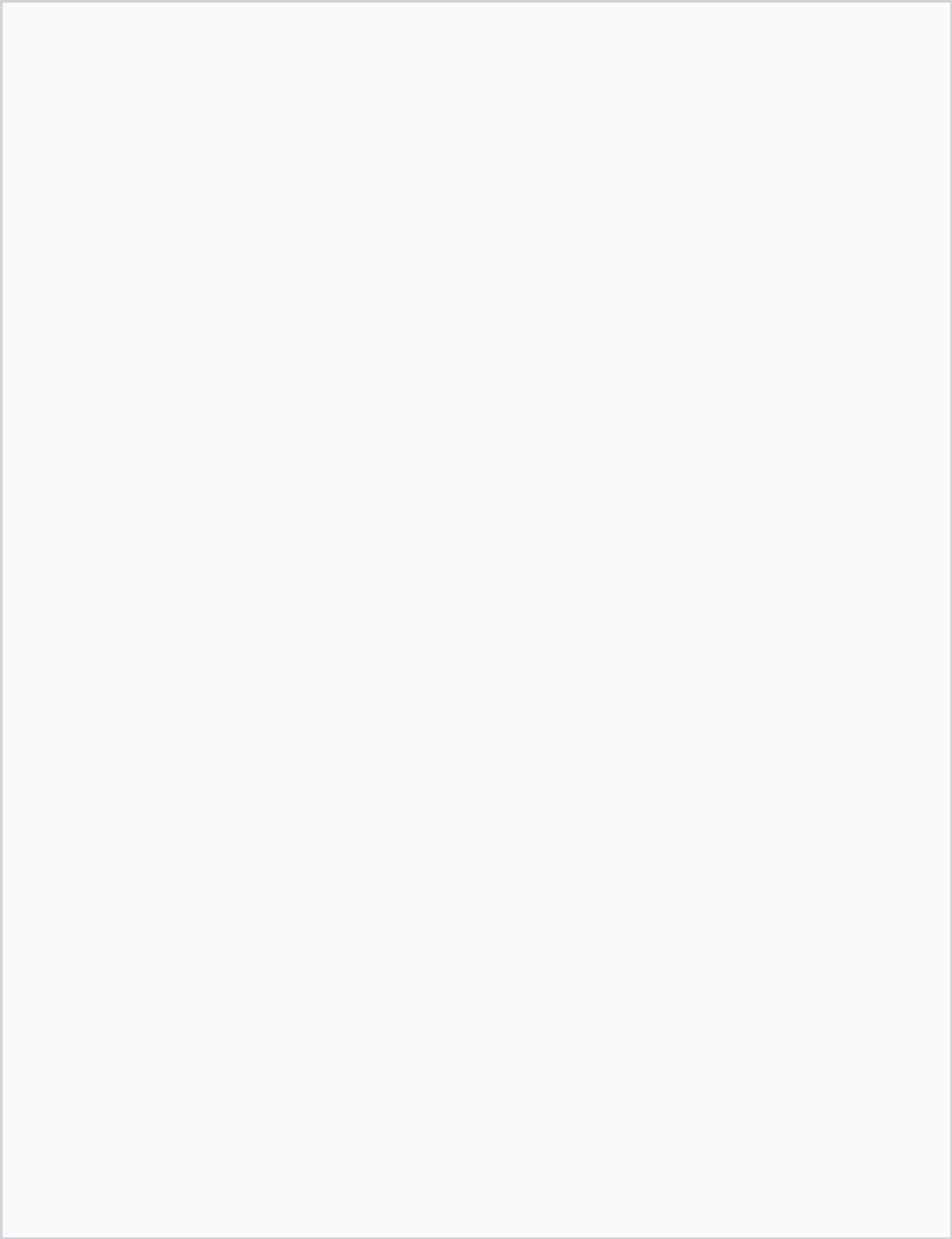
5private Date birthday;

6private String sex;

7private String address;

* }

3.6 创建接口层 UserMapper



* public interface UserMapper {
* public abstract List<User> findList();
* }

4

* public class UserMapperImpl implements UserMapper {

6

* public List<User> getList() {
* InputStream inputStream = null;
* SqlSessionFactory sqlSessionFactory = null;

10SqlSession session = null;

1. try {
2. String resource = "mybatis-config.xml"; // 配置文件
3. inputStream = Resources.getResourceAsStream(resource);
4. sqlSessionFactory = new SqlSessionFactoryBuilder().build(input

Stream);

1. session = sqlSessionFactory.openSession(); //获取session

16

1. List<User> list = session.selectList("com.bruceliu.dao.UserMap per.getList");
2. return list;
3. } catch (IOException e) {
4. e.printStackTrace();
5. } finally {
6. try {

|  |  |
| --- | --- |
| 23 | session.close(); |
| 24 | inputStream.close(); |

1. } catch (IOException e) {

26 e.printStackTrace();

1. }
2. }
3. return null;
4. }
5. }

3.7 创建接口Mapper(.xml)文件

Mapper映射文件是在实际开发过程中使用最多的，也是我们学习的重点。

Mapper文件中包含的元素有：

cache – 配置给定命名空间的缓存。

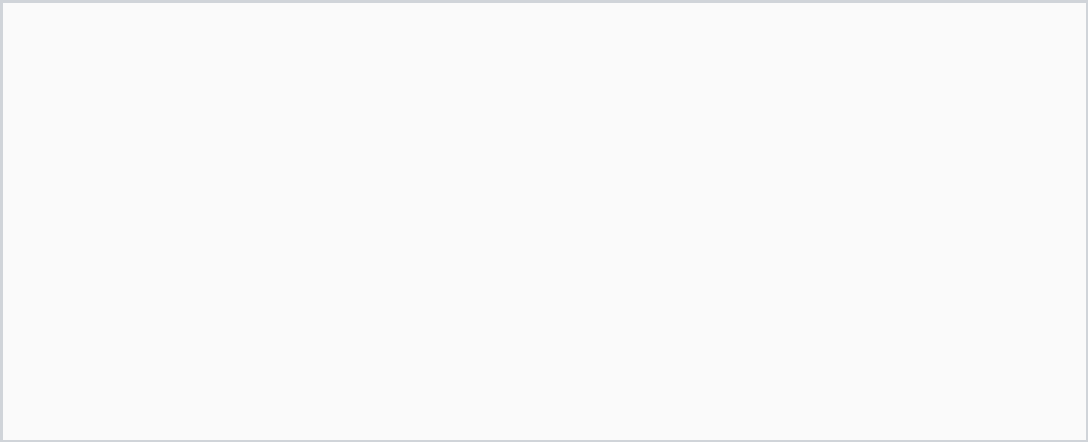
cache-ref – 从其他命名空间引用缓存配置。

resultMap – 映射复杂的结果对象。

sql – 可以重用的 SQL 块,也可以被其他语句引用。

insert – 映射插入语句 update – 映射更新语句 delete – 映射删除语句 select – 映射查询语句

select



* <?xml version="1.0" encoding="UTF-8"?>

2 <!DOCTYPE mapper

3PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

* "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> 5 <mapper namespace="com.bruceliu.dao.UserMapper">
* <!-- 返回值类型 -->
* <select id="getList" resultType="com.bruceliu.bean.User">

8select \* from user

* </select>

10 </mapper>

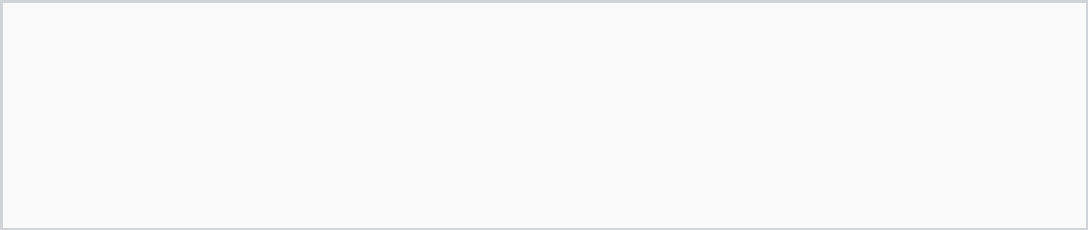
select标签叫Statement

id，必要属性，在当前的命名空间下不能重复。

指定输出类型，resultType

parameterType （不是必须）如果不指定，自动识别。

insert



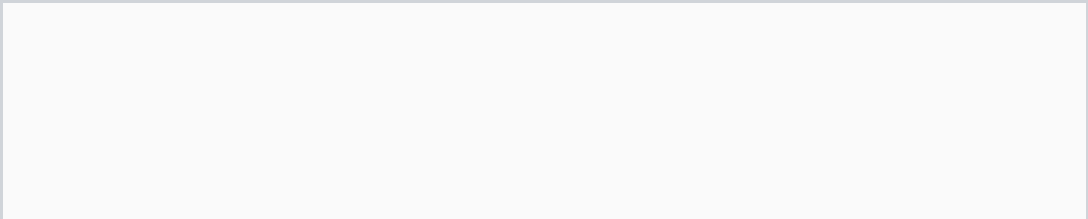
* <!-- 增删改返回的都是int值 不用写返回值 -->

2 <insert id="addUser">

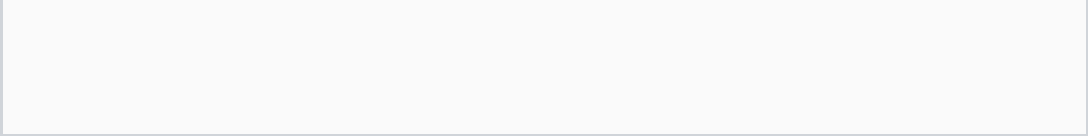
3INSERT INTO USER VALUES (null,#{username},#{birthday},

* #{sex},#{address})
* </insert>

获得到自增id(了解)



* <!-- 增删改返回的都是int值 不用写返回值 -->
* <insert id="addUser" parameterType="User" useGeneratedKeys="true" keyPrope rty="id">

* INSERT INTO USER VALUES (null,#{username},#{birthday},#{sex},#{addr

ess})

* </insert>

useGeneratedKeys：开启自增长映射

keyProperty：指定id所对应对象中的属性名

当执行完addUser()方法后，其返回值依然是执行sql影响的行数，并不是要获取的自增ID！Mybatis会自动将返回的主键值赋值给对象User的属性id，因此你可以通过属性的get方法获得插入的主键值： System.out.println(User.getId())；

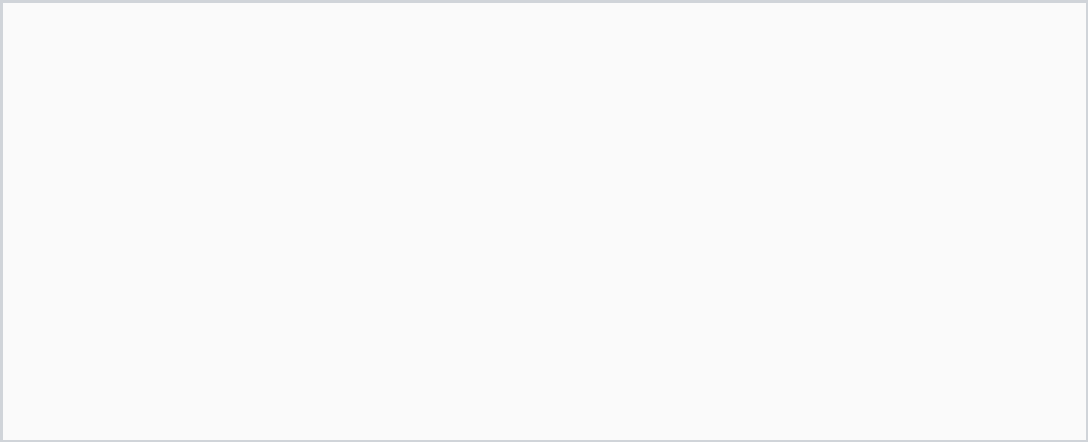
另外一种写法

在插入操作完成之前或之后，可以配置<selectKey>标签获得生成的主键的值，

获得插入之前还是之后的值，可以通过配置order属性来指定。

LAST\_INSERT\_ID：该函数是mysql的函数，获取自增主键的ID，它必须配合

insert语句一起使用



* <!-- 增删改返回的都是int值 不用写返回值 -->
* <insert id="addUser" parameterType="User">

3

* <selectKey keyProperty="id" resultType="int" order="AFTER">
* SELECT LAST\_INSERT\_ID()
* </selectKey>

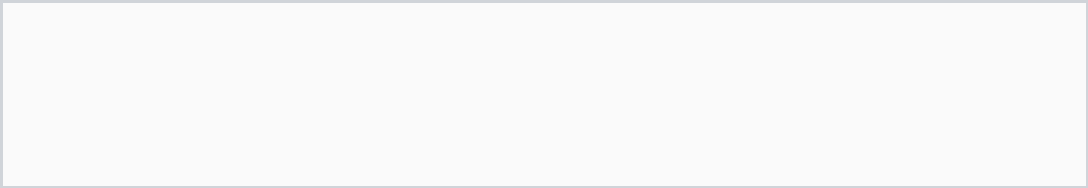
7

* INSERT INTO USER VALUES (null,#{username},#{birthday},#{sex},

|  |  |
| --- | --- |
| 9 | #{address}) |

1. </insert>

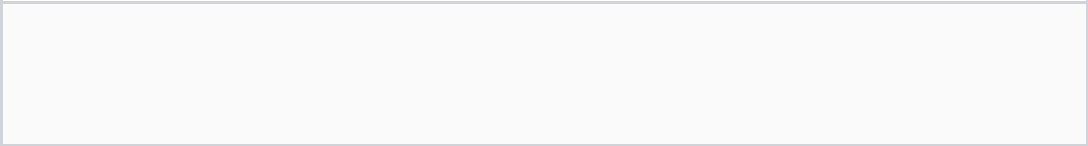
update



* <update id="updateUser" >
* update user set username=#{username},birthday=#{birthday}, 3 sex=#{sex},address=#{address} where id=#{id}

4</update>

delete



* <delete id="deleteUser" parameterType="int">
* delete from user where id=#{id} 3 </delete>

#和$区别(面试题)

#{}:相当于占位符

#{id}：其中的id可以表示输入参数的名称，如果是简单类型名称可以任意 # {} 实现的是sql语句的预处理参数，之后执行sql中用?号代替，使用时不需要关注数据类型，Mybatis自动实现数据类型的转换。并且可以防止SQL注

入。

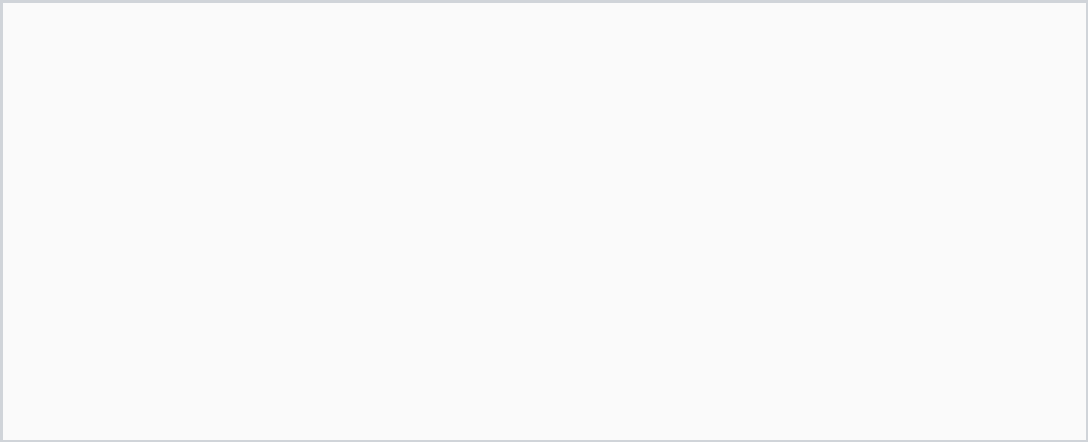
${}:表示拼接sql语句

${value}：表示输入参数的名称，如果参数是简单类型，参数名称必须是

value

${} 实现是sql语句的直接拼接，不做数据类型转换，需要自行判断数据类型。不能防止SQL注入。

3.8 测试类



* public class TestMybatis {
* UserMapperImpl um=new UserMapperImpl();
* @Test
* public void testQuery(){
* List<User> list = um.findList();

6for (User user : list) {

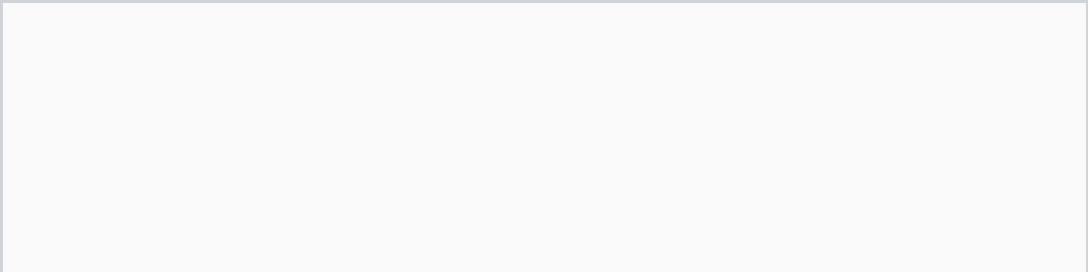
* System.out.println(user);
* }
* }

1. }

3.9 Mybatis 使用步骤

1. 创建SqlSessionFactory
2. 通过SqlSessionFactory创建SqlSession对象
3. 通过SqlSession操作数据库
4. 调用session.commit()提交事务
5. 调用session.close()关闭会话

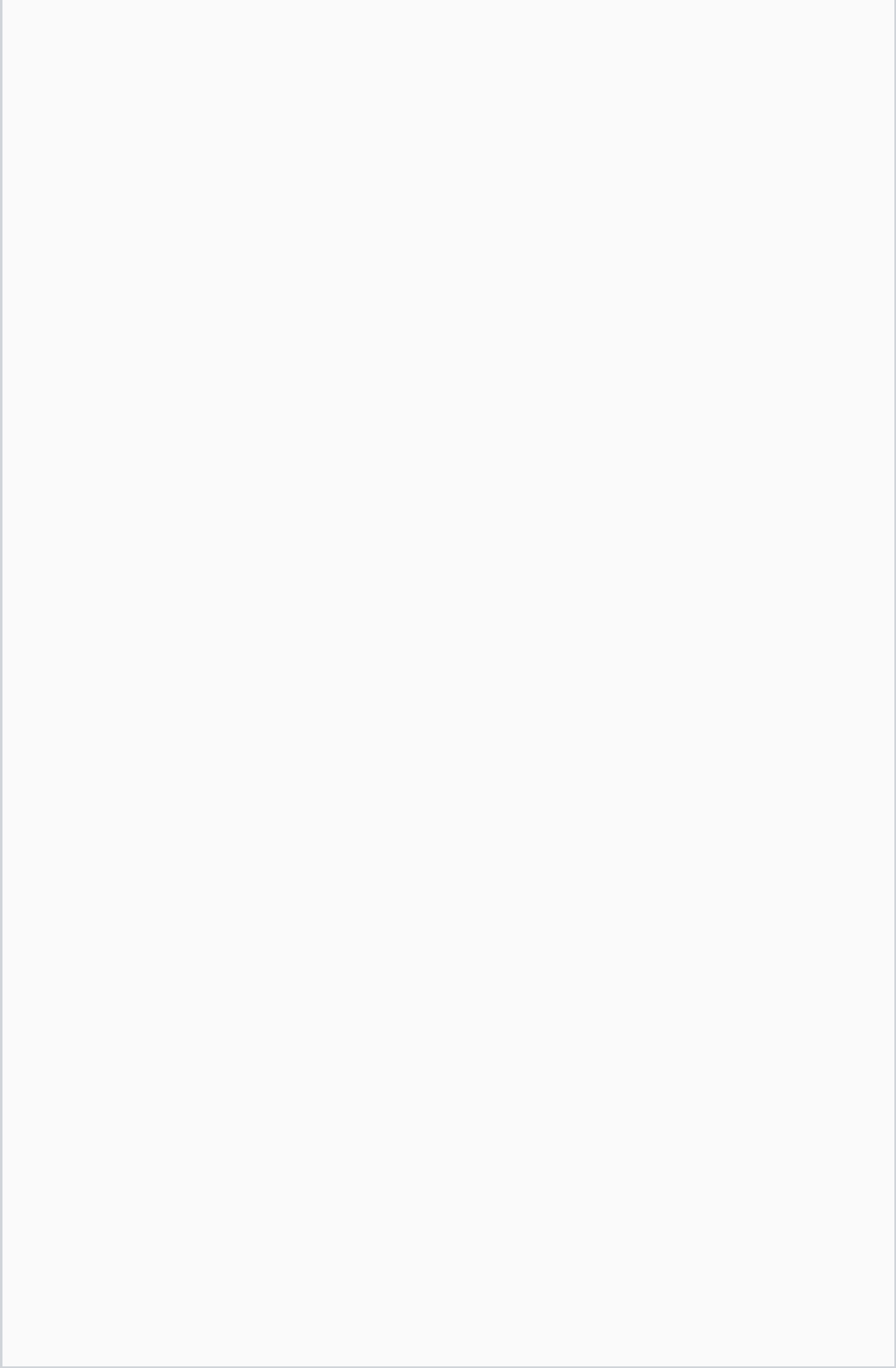
封装获取MyBatis中Session的工具类



* public class MyBatisUtils {
* /\*\*
* \* 01-获取SqlSession

4\* @return

* \*/
* public static SqlSession getSession(){

* SqlSession session=null;
* InputStream inputStream=null;
* try {

1. //配置文件的路径
2. String resource = "mybatis-config.xml";
3. //加载配置文件，得到一个输入流
4. inputStream = Resources.getResourceAsStream(resource);
5. //获取MyBatis的Session工厂
6. SqlSessionFactory sqlSessionFactory =

|  |  |
| --- | --- |
| 16 | new SqlSessionFactoryBuilder().build(inputStream); |

1. //通过session工厂获取到一个session (此session非Servlet中Session，

这个Session表示MyBatis框架和数据库的会话信息)

1. //获取到session就表示MyBatis连接上数据库啦！！类似于JDBC中 Connecti

on对象

1. session = sqlSessionFactory.openSession(true);//自动提交事务
2. //调用session的查询集合方法
3. return session;
4. } catch (IOException e) {
5. e.printStackTrace();
6. }finally {
7. try {

26 inputStream.close();

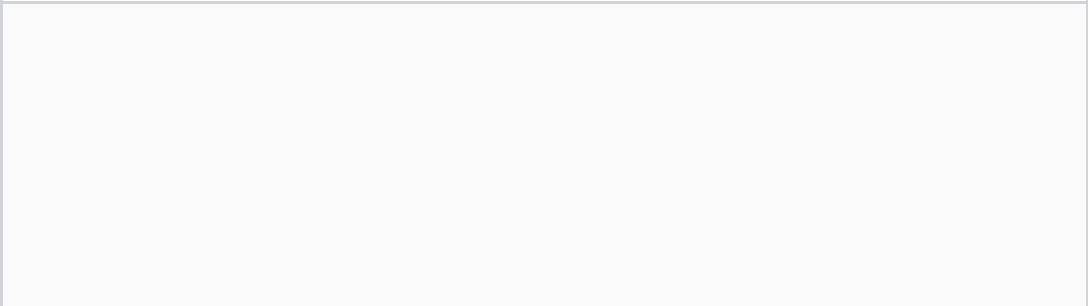
1. } catch (IOException e) {

28 e.printStackTrace();

1. }
2. }
3. return null;
4. }
5. /\*\*
6. \* 02-关闭SqlSession
7. \* @param session
8. \*/
9. public static void closeSession(SqlSession session){
10. if(session!=null){
11. session.close();
12. }
13. }
14. }

43

测试类



* public class TestMybatis {
* SqlSession session=null;

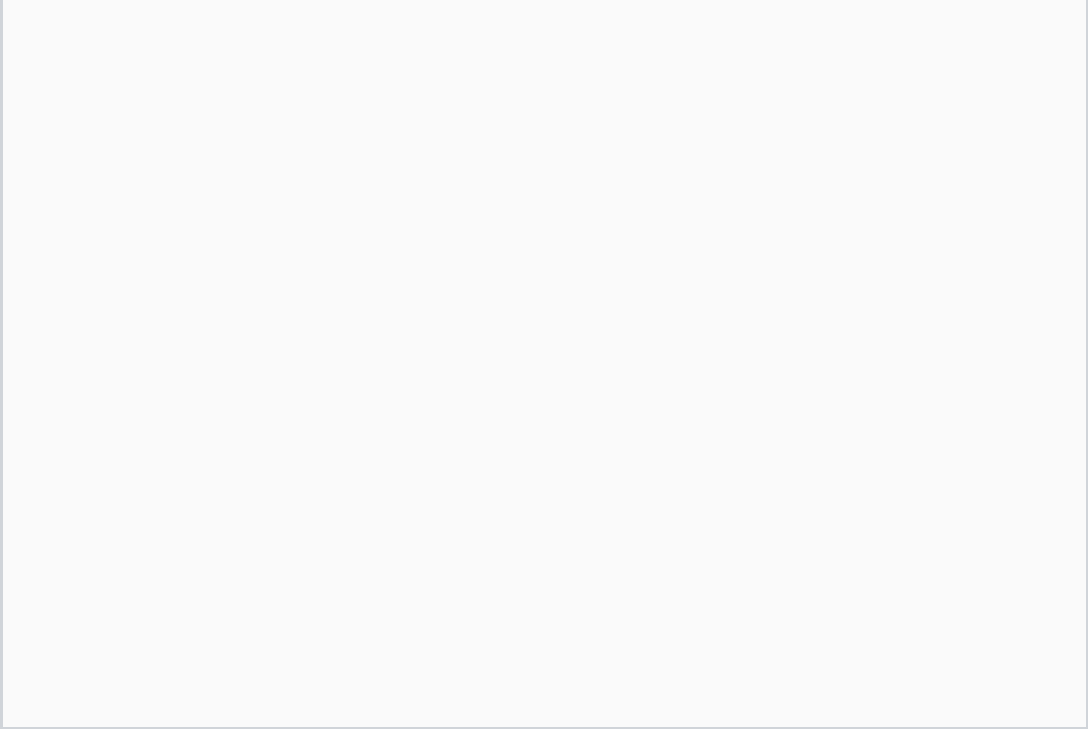
3UserMapper um = null;

4

5

* //在每次调用测试方法之前，自动调用init()方法

7@Before

* public void init(){
* session = MyBatisUtils.getSession();

1. //um就是Mapper的实现类
2. um = session.getMapper(UserMapper.class);
3. }

13

1. /\*
2. \* MyBatis增删改 需要手动提交事务
3. \*/
4. @Test
5. public void testAdd(){
6. User u=new User(0,"刘老师",new Date(),"女","日本东京");
7. int count = um.addUser(u);
8. System.out.println(count>0?"新增成功":"新增失败");
9. }

23

1. }

3.10 总结

使用mapper接口不用写接口实现类即可完成数据库操作，使用非常简单，也是官方所推荐的使用方法。

使用mapper接口的必须具备以几个条件：

1. Mapper的namespace必须和mapper接口的全路径一致。
2. Mapper接口的方法名必须和sql定义的id一致。
3. Mapper接口中方法的输入参数类型必须和sql定义的parameterType一

致。

4) Mapper接口中方法的输出参数类型必须和sql定义的resultType一致。

四.MyBatis-config配置

properties 属性 settings 设置

typeAliases 类型别名

typeHandlers 类型处理器 objectFactory 对象工厂 plugins 插件 environments 环境 environment 环境变量 transactionManager 事务管理器 dataSource 数据源 mappers 映射器

Mybatis的配置文件中配置项是有顺序的，即按照上面的顺序；

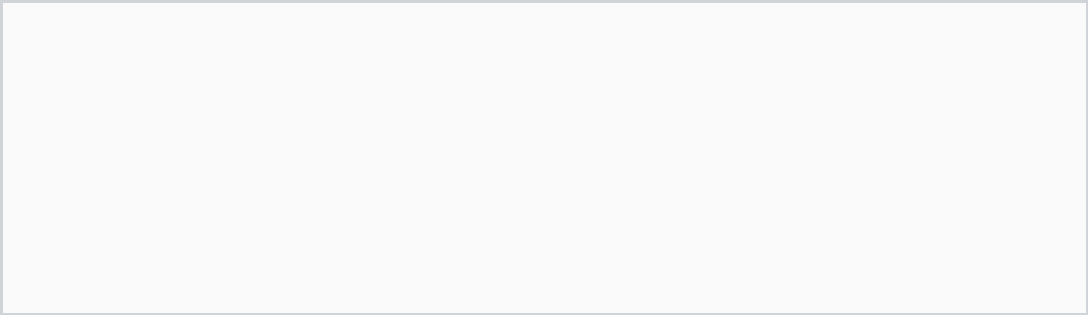
4.1 Properties 引入外部文件



4.2 typeAliases（别名）

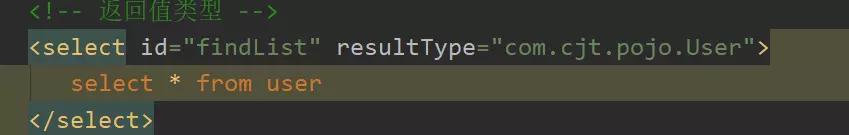
类型别名是为 Java 类型命名一个短的名字。 它只和 XML 配置有关, 只用来减少类完全限定名的多余部分。

自定义别名：

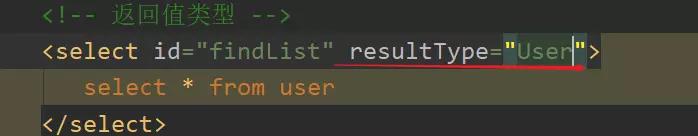


* <!--实体类取别名-->
* <typeAliases>
* <!--直接给所有的实体类取别名。默认的实体类的别名就是类名(不区分小大写)
* User实体类：User、user、USER
* -->
* <package name="com.bruceliu.bean"/>
* </typeAliases>

改变前



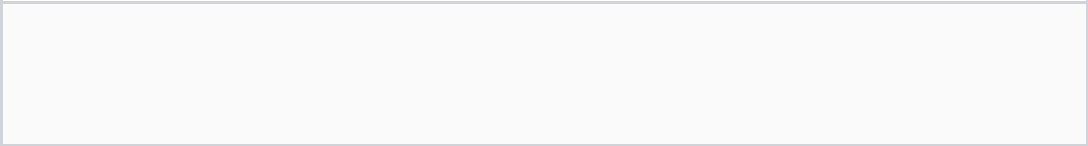
改变后



4.3 mappers

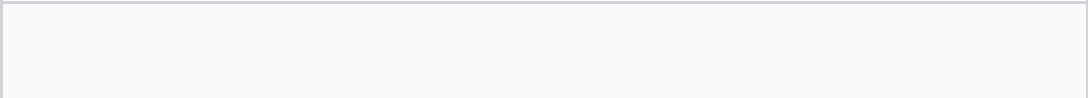
mapper映射文件的引入有3种方式：

1.路径相对于资源目录跟路径：



* <mappers>
* <mapper resource="com/bruceliu/dao/UserMapper.xml" /> 3 </mappers>

2.使用完整的文件路径

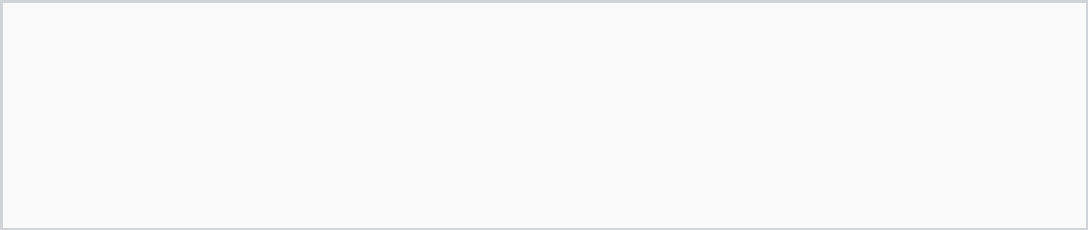


* <mappers>
* <mapper class="com.bruceliu.dao.UserMapper"/>

* </mappers>

注意：此种方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中

3.可直接配个扫描包：



* <!--引入映射文件-->

2 <mappers>

* <!--直接映射包的名字，那么这个包下面所有的Mapper接口全部映射！-->

4<package name="com.bruceliu.mapper"/>

5 </mappers>

4.4 parameterType的传递参数

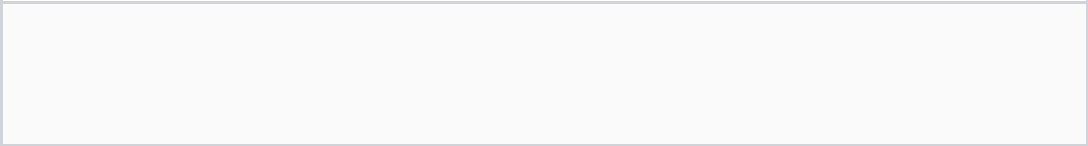
传入类型有三种：

1、简单类型，string、long、integer等

2、Pojo类型，User等

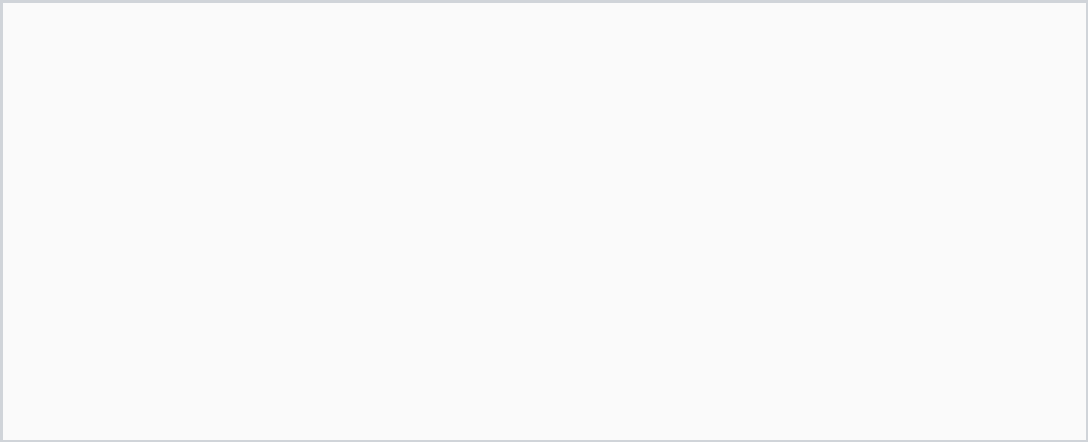
3、HashMap类型。

查询



* <select id="getUsers" parameterType="map" resultType="User">
* select \* from user where birthday between #{startdate} and #{enddate} 3 </select>

测试



* @Test
* public void test7(){
* HashMap<String, Object> map=new HashMap<String, Object>();

4map.put("startdate", "2018-09-07");

5map.put("enddate", "2018-09-25");

6List<User> users = um.getUsers(map);

7for (User user : users) {

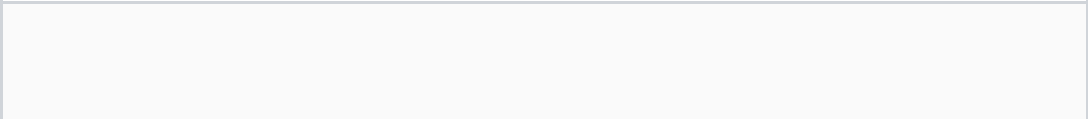
* System.out.println(user);
* }

1. }

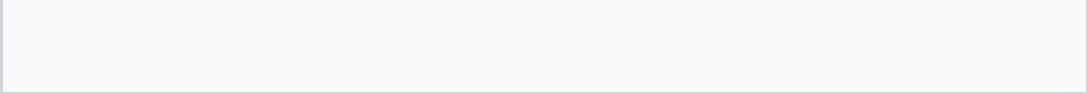
注意：map的key要和sql中的占位符保持名字一致

4.5 分页查询

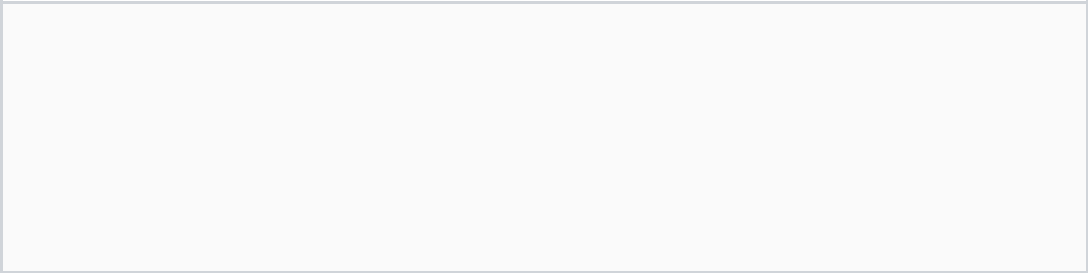
查询



* <!-- 分页：map传参 -->
* <select id="selectAuthorByPage" resultType="User">

* SELECT \* FROM USER LIMIT #{offset}, #{pagesize}
* </select>

接口

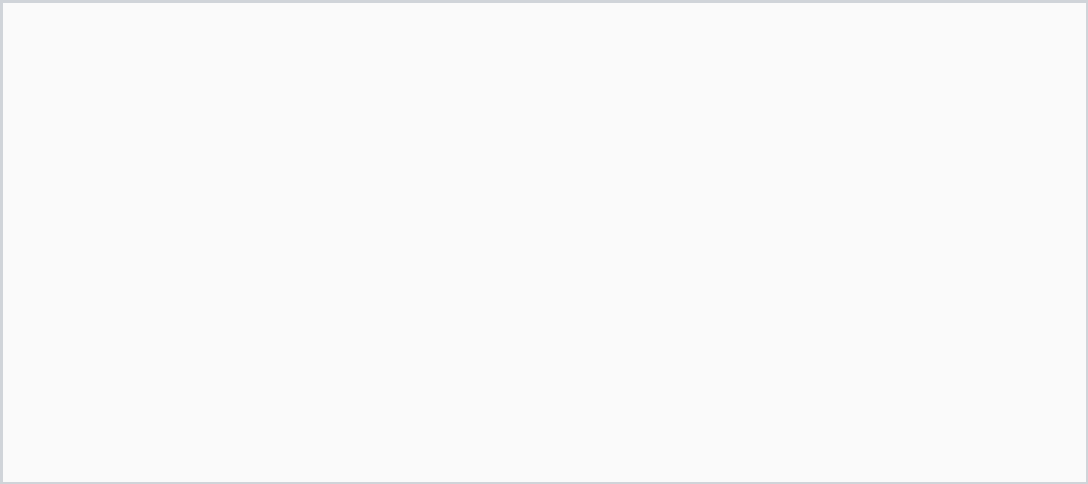


* /\*\*
* \* 根据分页参数查询
* \* @param paramList 分页参数

4\* @return 分页后的用户列表

* \*/
* List<User> selectAuthorByPage(Map<String, Object> paramList);

测试



* @Test
* public void testSelectAuthorByPage() {
* Map<String, Object> map = new HashMap<String, Object>();
* map.put("offset", 0);
* map.put("pagesize", 2);

6

* List<User> authorList = mapper.selectAuthorByPage(map);

8for (int i = 0; i < authorList.size(); i++) {

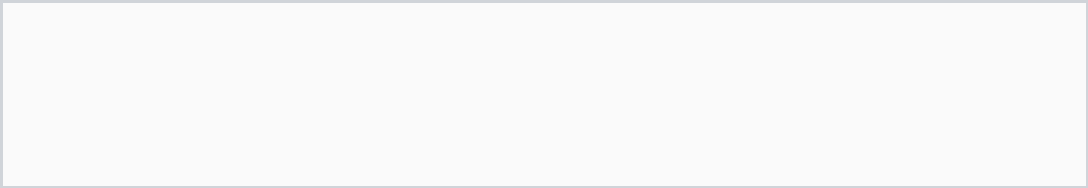
* System.out.println(authorList.get(i));

1. }
2. }

4.6 使用注解方式

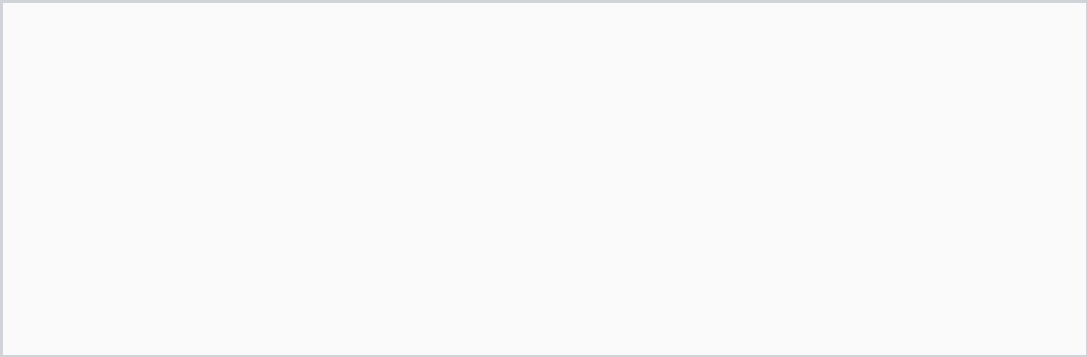
注意：mapper文件中的参数占位符的名字一定要和接口中参数的注解保持一致

mapper



* <!-- 分页：map传参 -->
* <select id="selectUserByPage2" resultType="User">
* SELECT \* FROM USER LIMIT #{offset}, #{pagesize}
* </select>

接口

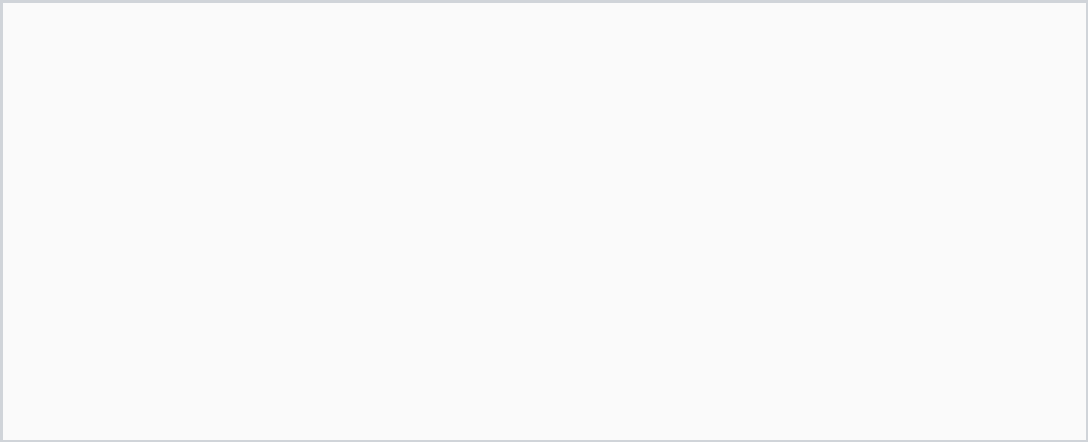


* /\*\*
* \* 根据分页参数查询
* \* @param offset 偏移量
* \* @param pagesize 每页条数

5\* @return 分页后的用户列表

* \*/
* List<User> selectUserByPage2(@Param(value = "offset") int offset, @Par am(value = "pagesize") int pagesize);

测试



* @Test
* public void testSelectAuthorByPage2() {

3

* List<User> authorList = mapper.selectUserByPage2(0, 2);

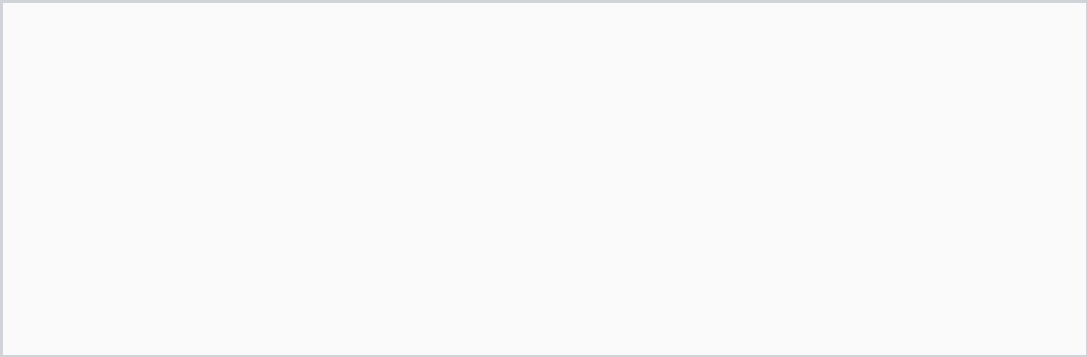
5

* for (int i = 0; i < authorList.size(); i++) {
* System.out.println(authorList.get(i));
* System.out.println("----------------------");
* }

1. }

4.7 返回map 类型

mapper



* <resultMap id="resultMap1" type="HashMap">

2<result property="key" column="r1" />

3<result property="value" column="r2" />

* </resultMap>

5

* <select id="getResult" resultMap="resultMap1">
* select count(\*) r1, max(birthday) r2 from user
* </select>

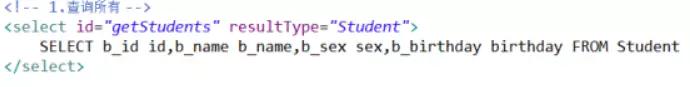
接口



* public HashMap<String, Object> getResult();

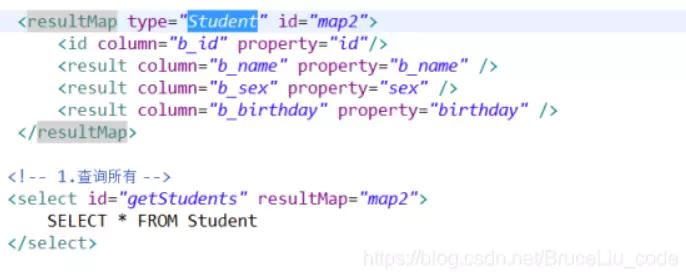
解决数据库字段和实体类属性不同

解决办法一: 通过在查询的sql语句中定义字段名的别名，让字段名的别名和实体类的属性名一致，这样就可以表的字段名和实体类的属性名一一对应上了，这种方式是通过在sql语句中定义别名来解决字段名和属性名的映射关系的。



解决办法二: 通过<resultMap>来映射字段名和实体类属性名的一一对应关系。

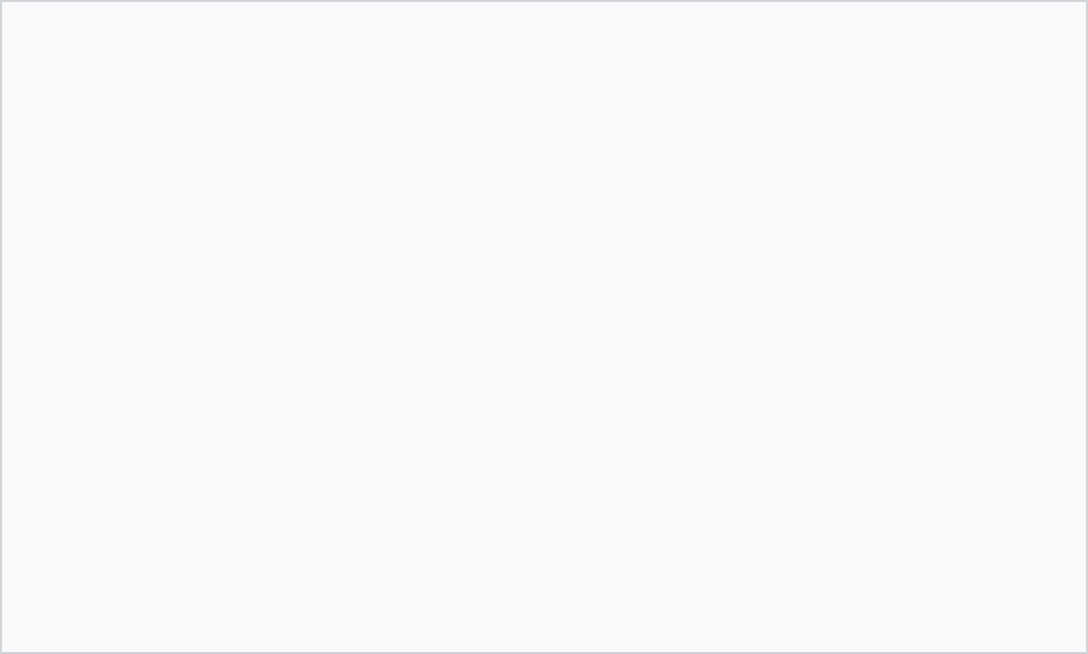
这种方式是使用MyBatis提供的解决方式来解决字段名和属性名的映射关系的。



五.MyBatis 动态SQL

5.1 if +where 带条件查询

mapper



* <select id="selectUserByUsernameAndSex" resultType="User" parameterType="Use

2select \* from user

* <where>
* <if test="username != null">
* username=#{username}
* </if>
* <if test="sex != null">
* and sex=#{sex}
* </if>

1. </where>
2. </select>

12

1. ----------------------------------------------------------------------
2. //1.根据姓名和性别查询用户
3. public List<User> selectUserByUsernameAndSex(User user);

5.2 if+set 修改

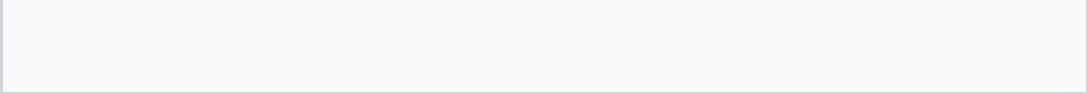


* <update id="updateUser" parameterType="User">

2update user u

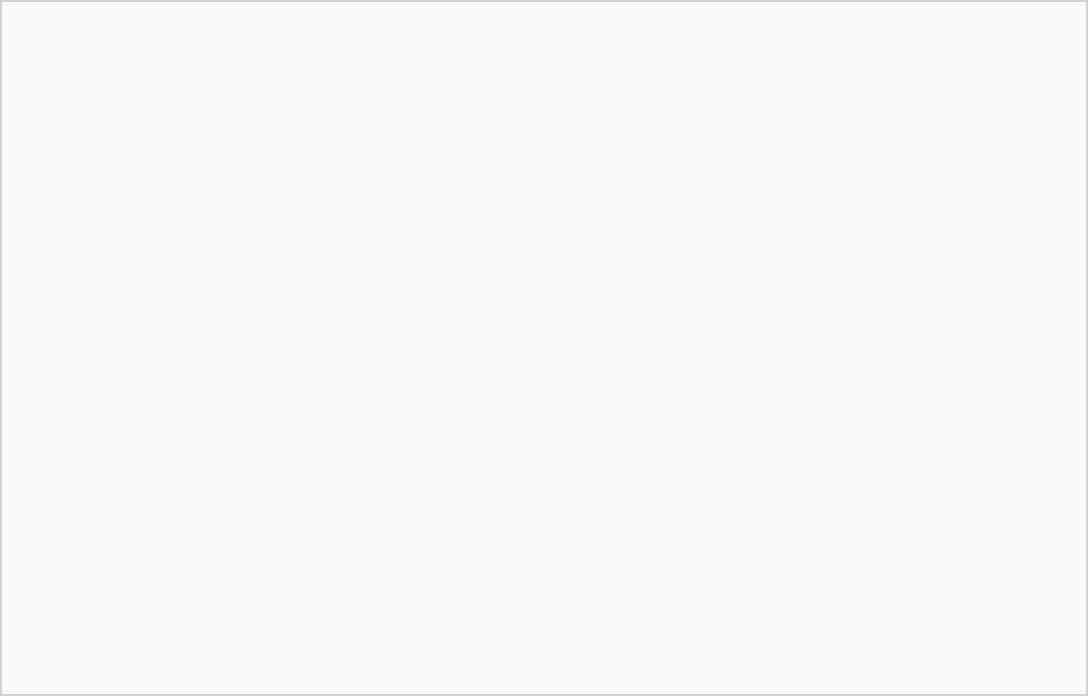
* <set>
* <if test="username!=null and username!=''">
* u.username = #{username},
* </if>
* <if test="sex != null and sex != ''">
* u.sex = #{sex},
* </if>

10</set>

1. where id = #{id}
2. </update>

5.3 choose(when,otherwise) 语句 了解

我们不想用到所有的查询条件，只想选择其中的一个，查询条件有一个满足即可，使用 choose 标签可以解决此类问题，类似于 Java 的 switch 语句



* <select id="selectUserByUsernameAndSex1" resultType="User" parameterType="Us

2select \* from user

* <where>
* <choose>
* <when test="id !='' and id != null">

6 id=#{id}

* </when>
* <when test="username !='' and username != null">

|  |  |
| --- | --- |
| 9 | and username=#{username} |

1. </when>
2. <otherwise>

|  |  |
| --- | --- |
| 12 | and sex=#{sex} |

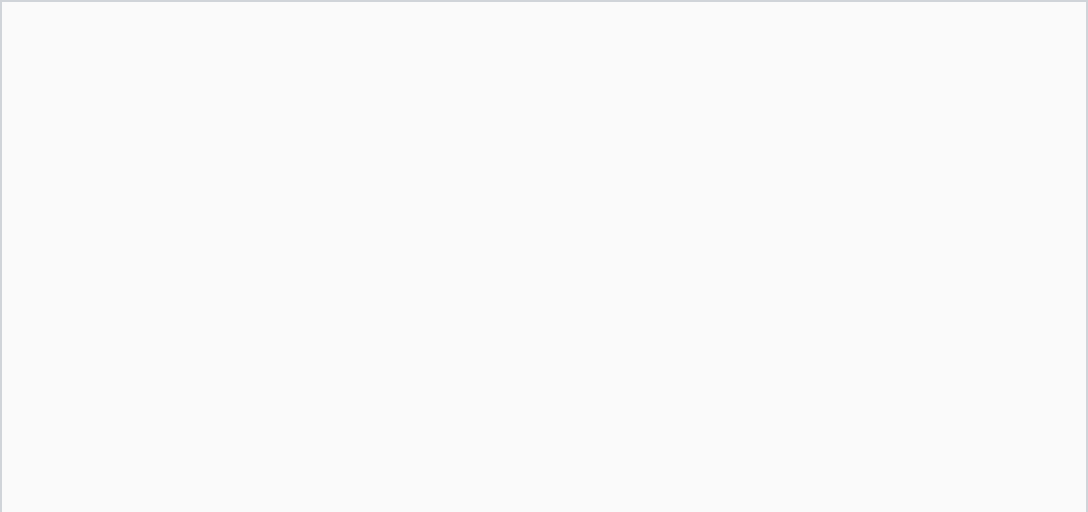
1. </otherwise>
2. </choose>
3. </where>
4. </select>

这里我们有三个条件，id,username,sex，只能选择一个作为查询条件 如 果 id 不为空，那么查询语句为：select \* from user where id=? 如果 id 为空，那么

看 username 是否为空，如果不为空，那么语句为 select \* from user where username=?; 如果 username 为空，那么查询语句为 select \* from user where sex=?

5.4 SQL 片段

有时候可能某个 sql 语句我们用的特别多，为了增加代码的重用性，简化代码，我们需要将这些代码抽取出来，然后使用时直接调用。



* <!--公有的代码抽象出去 -->

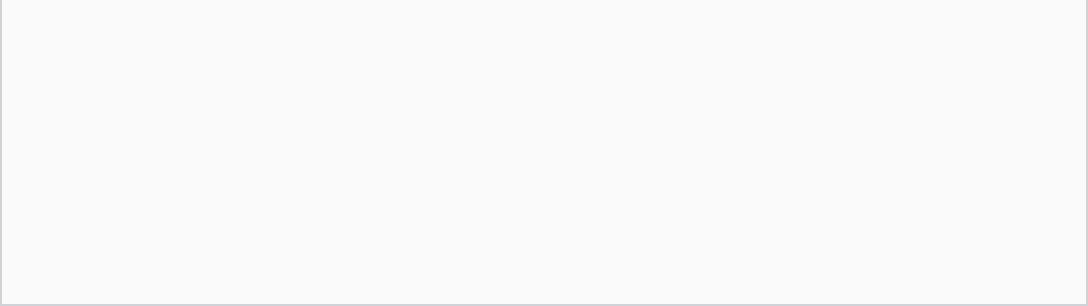
2<sql id="querySQL">

3<if test="username!=null and username!=''">

* and username like #{username}
* </if>
* <if test="sex!=null and sex!=''">
* and sex=#{sex}
* </if>
* </sql>

10

1. 引用 sql 片段

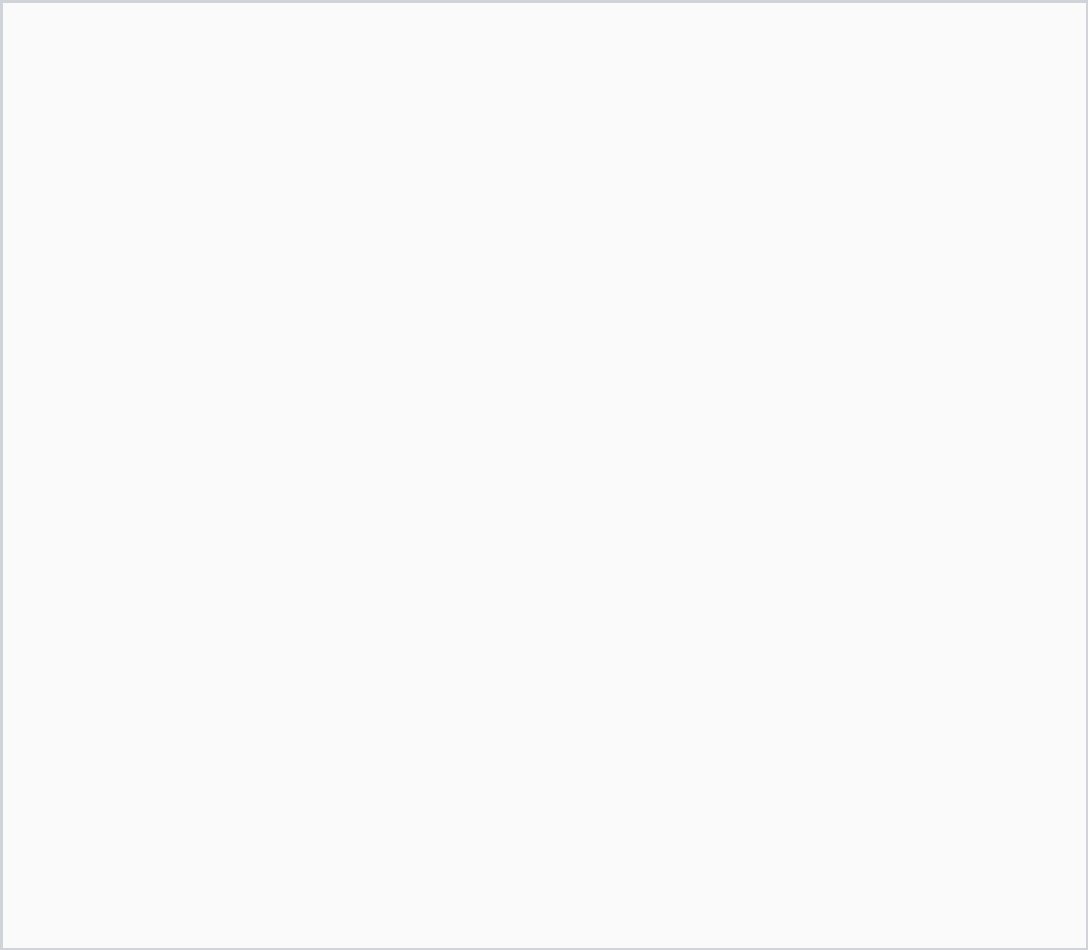
1. <select id="selectUserByUsernameAndSex" resultType="User" parameterType="
2. select \* from user
3. <where>

15

1. <include refid="querySQL"></include>
2. </where>

18

5.5 foreach 语句



* <!--批量删除-->
* <delete id="deleteSome">
* delete from user where id in
* <foreach collection="list" item="xx" open="(" separator="," close=

")">

* #{xx}
* </foreach>
* <!--
* delete from user where id in (10,4,29,30)
* -->

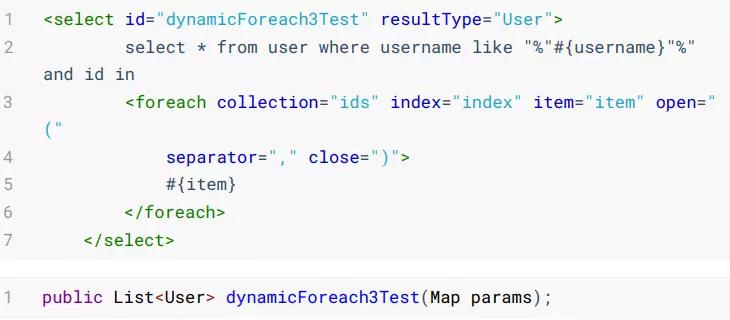
1. </delete>

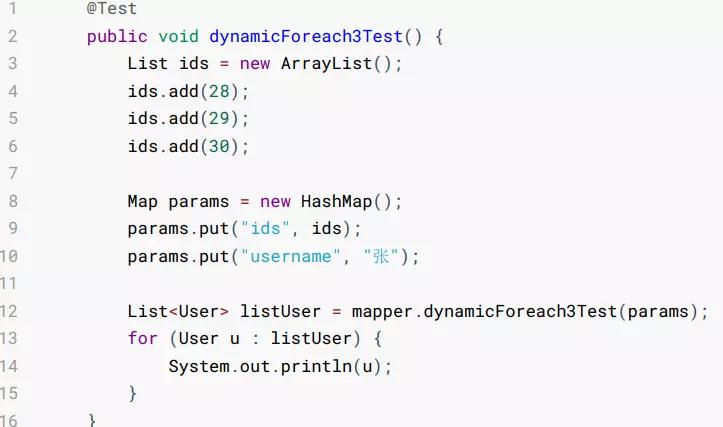
11

12

1. <delete id="deleteSome1">
2. delete from user where
3. <foreach collection="array" item="xx" separator="or" >
4. id=#{xx}
5. </foreach>
6. <!--
7. delete from user where id=? or id=? or id=? or id=?
8. -->
9. </delete>

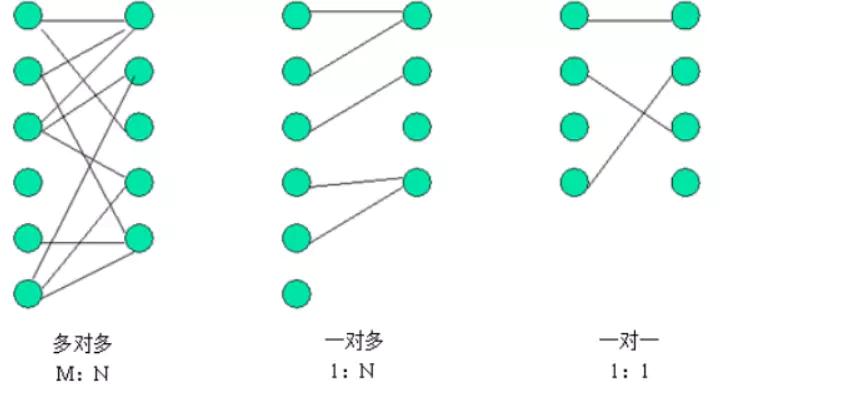
将参数封装成Map的类型





六.MyBatis 关联映射

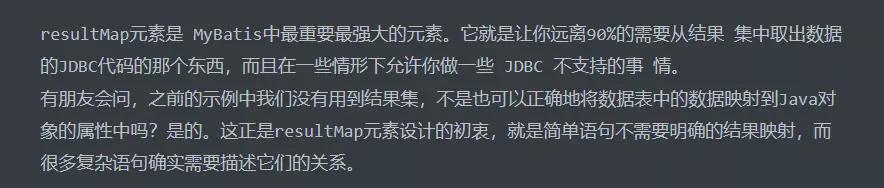
数据库中多表之间存在着三种关系，如图所示。



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 多对多： 程序员<------ | | >项目 | 用户--------- | >角色 |  |  |
| 2 | 一对多: | 班级------ | >学生 | 学校------ | >班级 | 帅哥 | ----->多个女朋友 |
| 3 |  | 土豪--- | 多辆豪车 |  |  |  |  |
| 4 | 一对一: | 学生----- | >学位证 | 人------ | >DNA | 公民---- | 身份证 |
|  |  |  |  |  |  |  |  |

6.1 ResultMap

ResultMap查询结果集



resultMap元素中，允许有以下直接子元素：

constructor － 类在实例化时，用来注入结果到构造方法中（本文中暂不讲

解）

id － 作用与result相同，同时可以标识出用这个字段值可以区分其他对象实

例。可以理解为数据表中的主键，可以定位数据表中唯一一笔记录 result － 将数据表中的字段注入到Java对象属性中

association － 关联，简单的讲，就是“有一个”关系，如“用户”有一个“帐号”

collection － 集合，顾名思议，就是“有很多”关系，如“客户”有很多“订单”

6.2 一对一关联(了解)

MyBatis中使用association（有一个）标签来解决一对一的关联查询，association标签可用的属性如下：

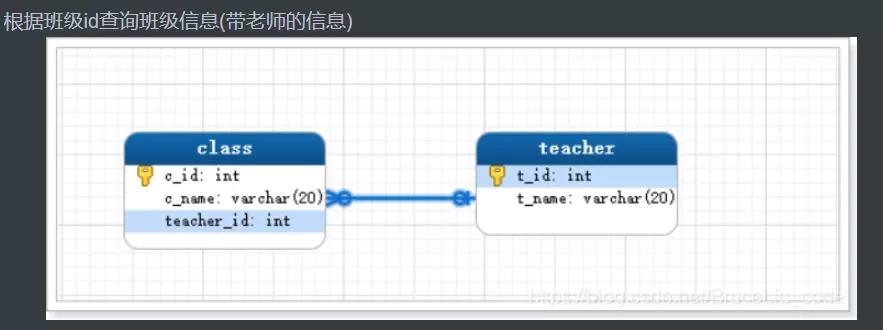
property:对象属性的名称

javaType:对象属性的类型

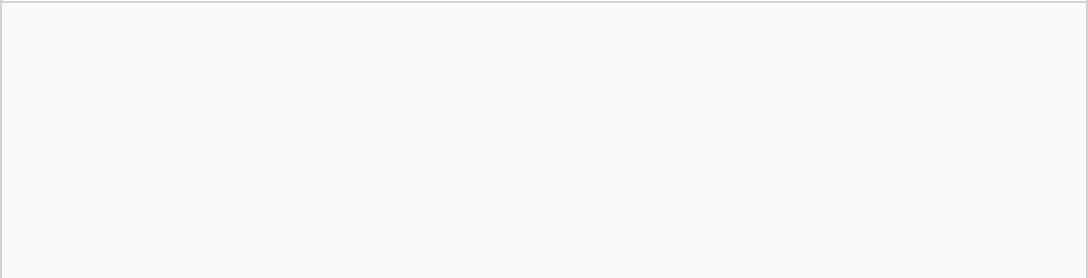
column:所对应的外键字段名称

select:使用另一个查询封装的结果

1.需求



2.数据表



* CREATE TABLE teacher(
* t\_id INT PRIMARY KEY AUTO\_INCREMENT,

3t\_name VARCHAR(20)

4 );

5 CREATE TABLE class(

6c\_id INT PRIMARY KEY AUTO\_INCREMENT,

* c\_name VARCHAR(20),

8teacher\_id INT

9 );

1. ALTER TABLE class ADD CONSTRAINT fk\_teacher\_id FOREIGN KEY (teacher\_id) REFE

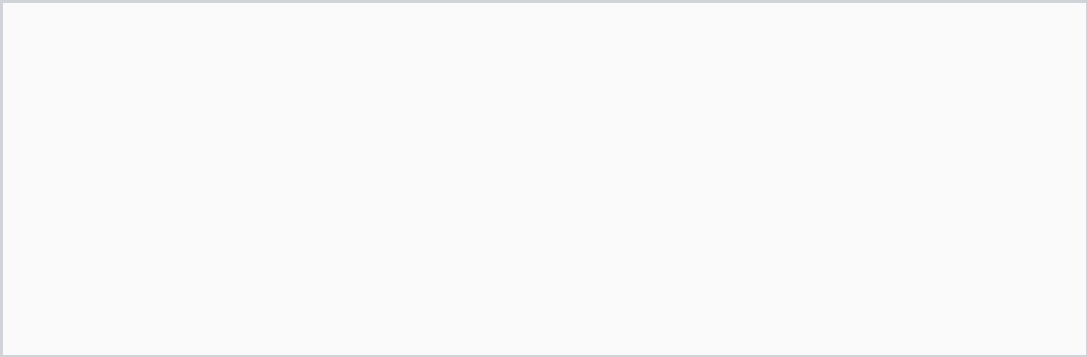
11

1. INSERT INTO teacher(t\_name) VALUES('teacher1');
2. INSERT INTO teacher(t\_name) VALUES('teacher2');

14

1. INSERT INTO class(c\_name, teacher\_id) VALUES('class\_a', 1);
2. INSERT INTO class(c\_name, teacher\_id) VALUES('class\_b', 2);

3.定义实体类



* public class Teacher {

2

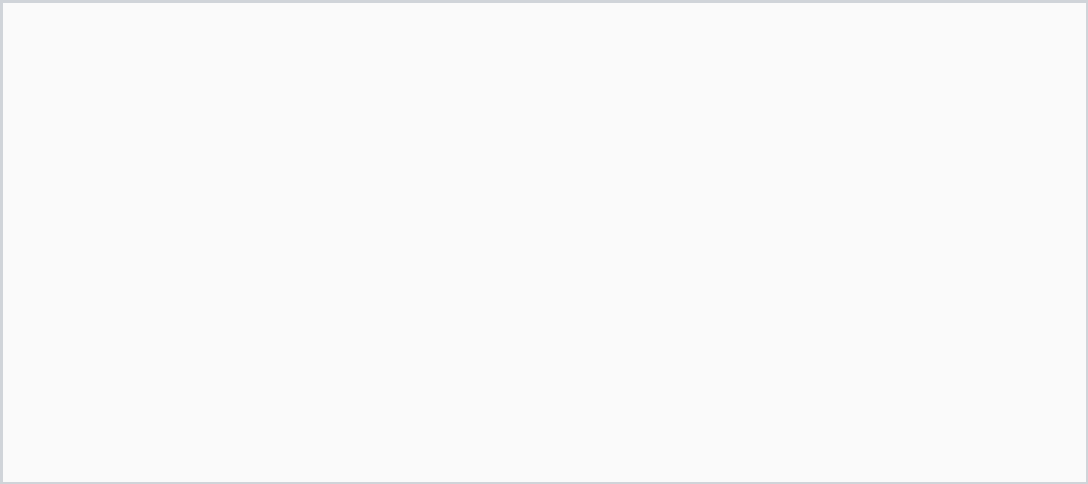
* // 定义实体类的属性，与teacher表中的字段对应

4private int id; // id===>t\_id

5private String name; // name===>t\_name

6

* //get set 方法
* }



* public class Classes {
* // 定义实体类的属性，与class表中的字段对应

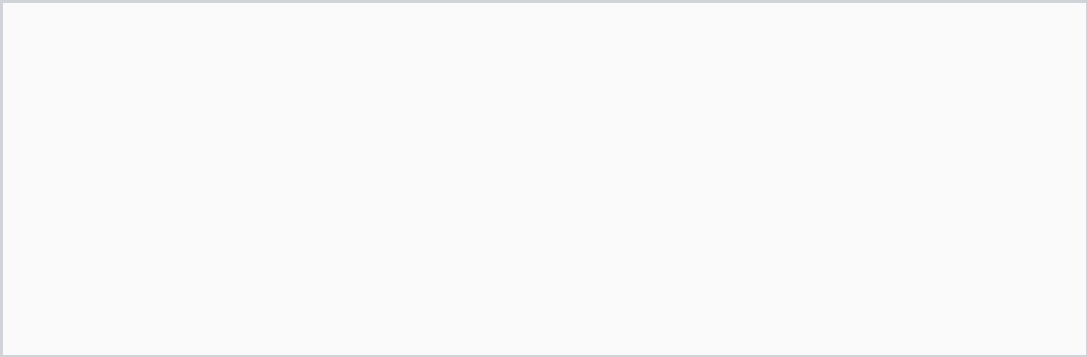
3private int id; // id===>c\_id

4private 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;

1. }

4.接口

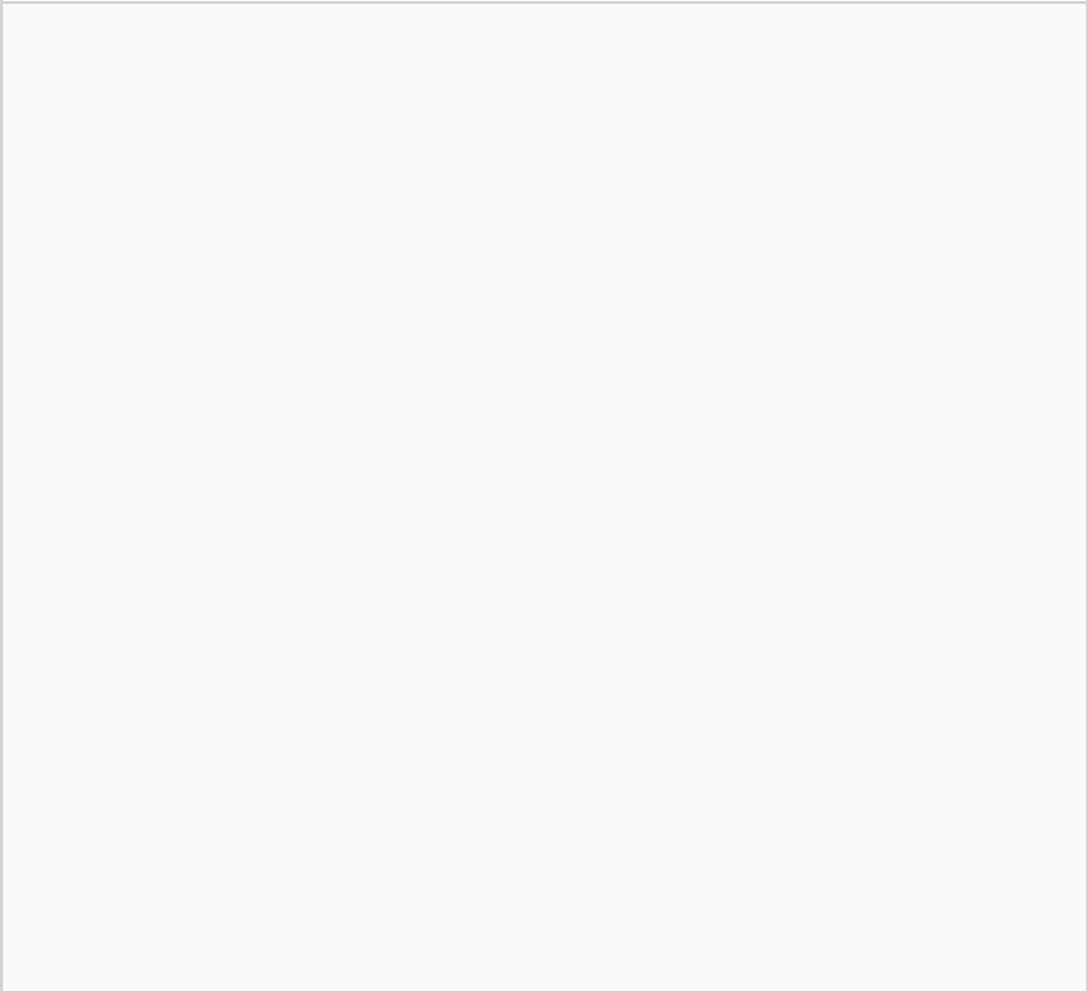


* public interface ClassesMapper {
* public List<Classes> getClasses();
* /\*\*
* \* 根据班级的ID查询班级信息(包含该班级下老师的信息)

5\* @return

* \*/
* public Classes getClassesById(int cId);
* }

5.mapper.xml



* <?xml version="1.0" encoding="UTF-8"?>

2 <!DOCTYPE mapper

3PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

* "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> 5 <mapper namespace="com.cjt.mapper.ClassesMapper">
* <!--
* select \* from class c, teacher t where c.teacher\_id=t.t\_id and c.c

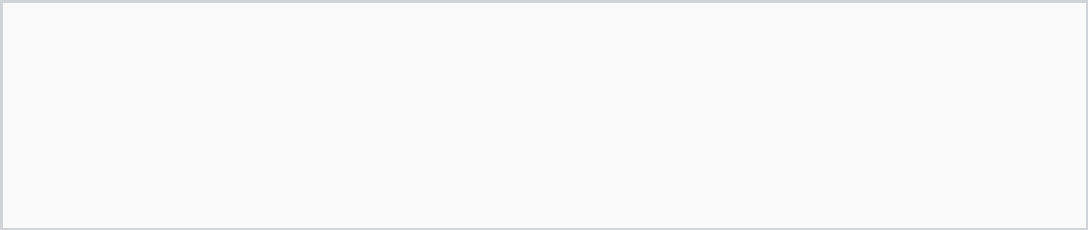
\_id=1

* -->
* <select id="getClassesById" parameterType="int" resultMap="classaa">

10select \* from class c, teacher t where c.teacher\_id=t.t\_id and c.c \_id=#{id}

1. </select>
2. <!-- 使用resultMap映射实体类和字段之间的一一对应关系 -->
3. <resultMap type="Classes" id="classaa">
4. <id property="id" column="c\_id"/>
5. <result property="name" column="c\_name"/>
6. <association property="teacher" javaType="com.cjt.pojo.Teacher">
7. <id property="id" column="t\_id"/>
8. <result property="name" column="t\_name"/>
9. </association>
10. </resultMap>
11. </mapper>

6.测试类

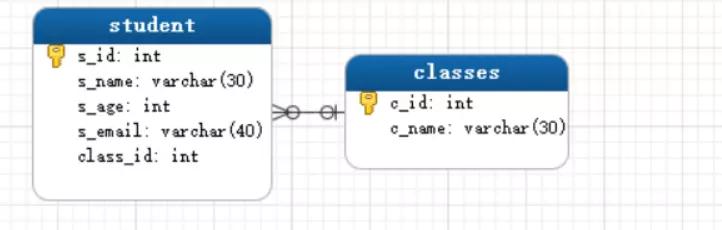


* @Test
* public void test2(){
* Classes c1 = mapper.getClassesById(2);
* System.out.println(c1);
* }

6.3 一对多关联映射

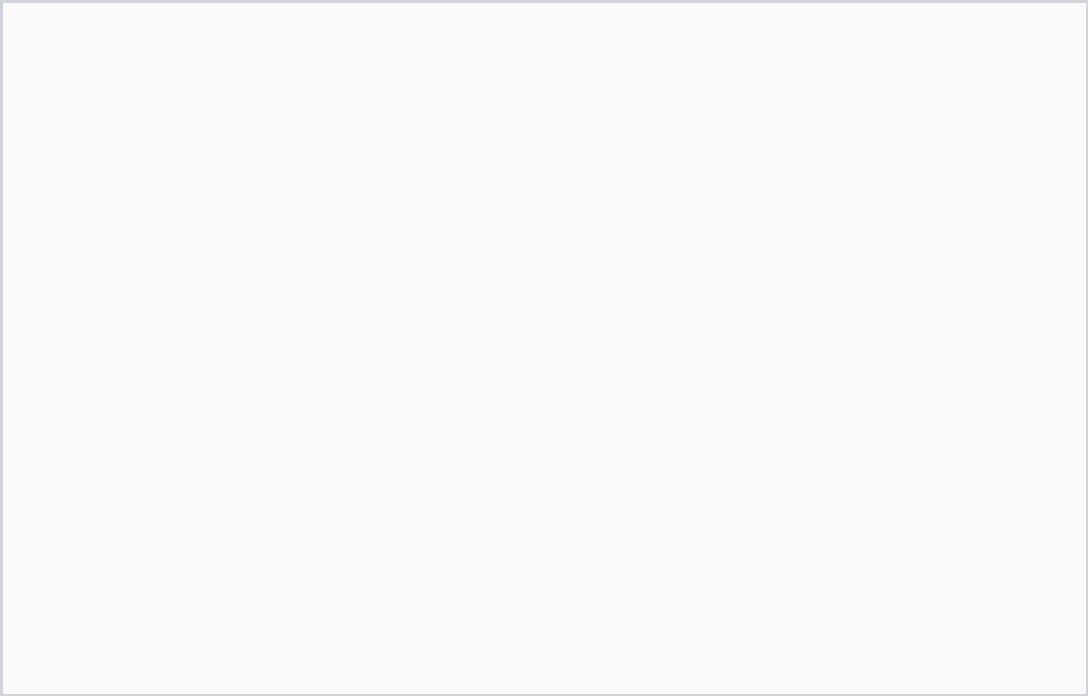
MyBatis中使用collection标签来解决一对多的关联查询，ofType属性指定集合中元素的对象类型。

1.需求



2.定义实体类

班级对象



* package com.bean;
* import java.util.List;

3 public class Classes {

4// 定义实体类的属性，与class表中的字段对应

5private int id; // id===>c\_id

6private 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，

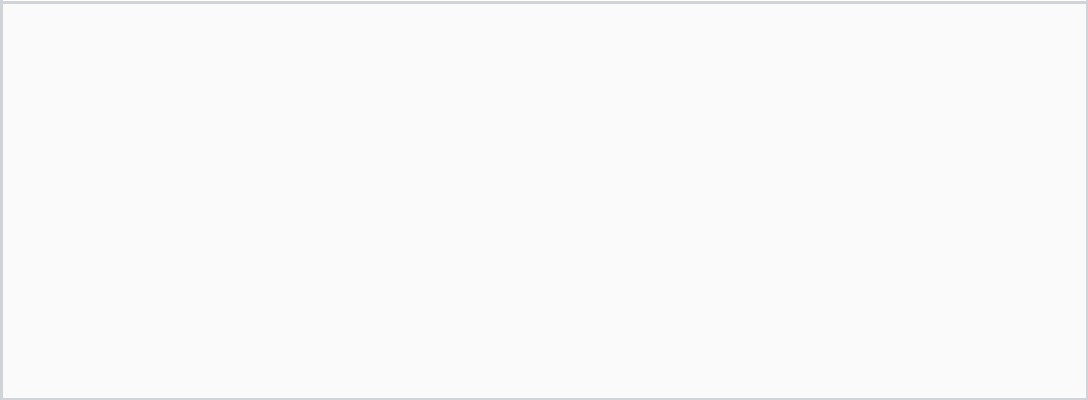
9\* 用于维护teacher和class之间的一对一关系，

1. 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
2. \*/
3. //private Teacher teacher;

13

1. //表示含义 1个班级下有多个学生
2. private List<Student> students; //学生集合 多的一方
3. }

学生对象



* public class Student { 2 private long sId;

3private String sName;

4private long sAge;

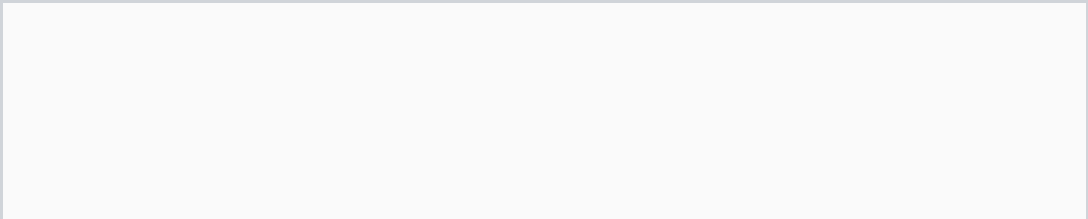
5private long classId;

* //额外准备一个班级对象
* private Classes classes; //体现一个学生在一个班级中

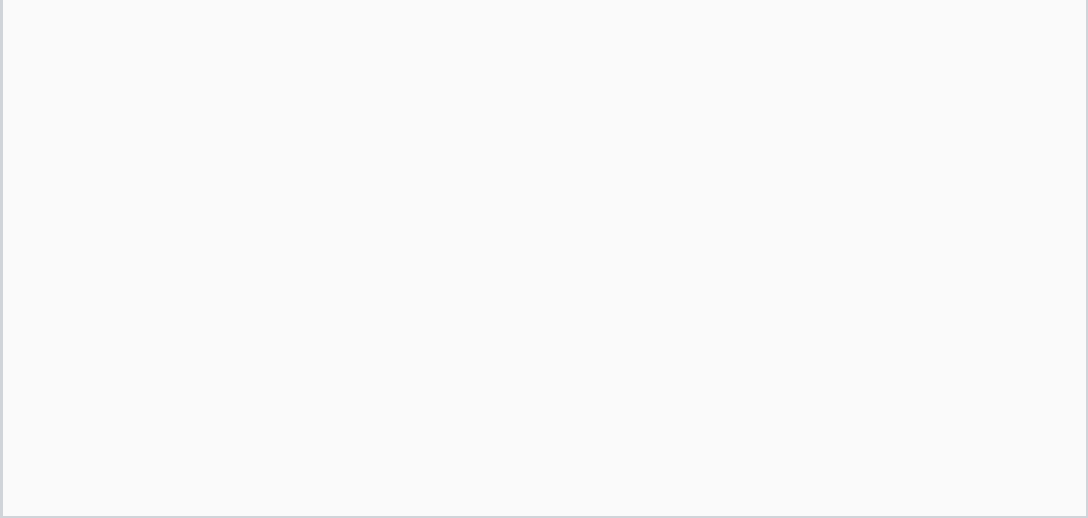
8

* }

3 mapper.xml 映射文件



* <resultMap id="StuMap" type="Classes">
* <!--主键-->
* <id property="id" column="c\_id"/>
* <result property="name" column="c\_name"/>

* <!--配置一个包含关系 “有很多”关系 -->
* <collection property="students" ofType="Student">

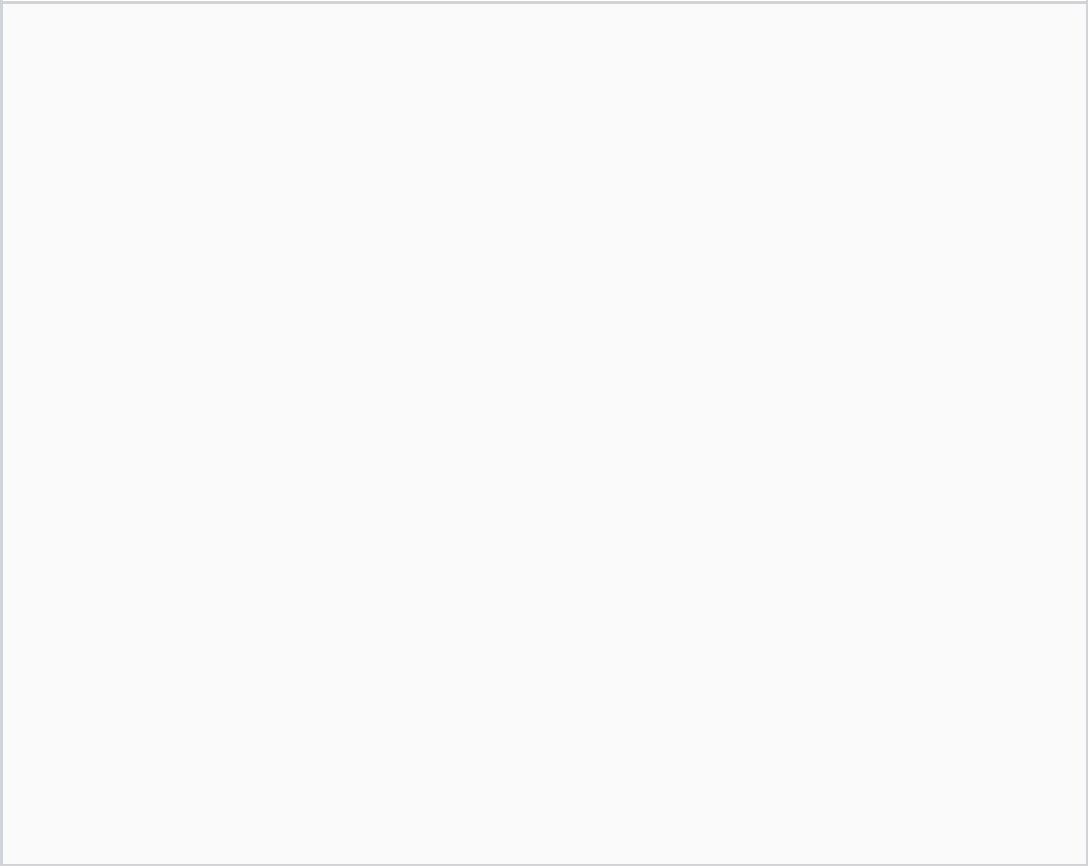
7<id property="sId" column="s\_id"/>

8<result property="sName" column="s\_name"/>

9<result property="sAge" column="age"/>

1. <result property="classId" column="class\_id"/>
2. </collection>
3. </resultMap>
4. <select id="getStuList" parameterType="int" resultMap="StuMap">
5. SELECT C.\*,S.\* FROM classes C
6. INNER JOIN student S on C.c\_id=S.class\_id where C.c\_id=#{classId}
7. </select>

5 测试



* public class MybatisTest2 {

2

* SqlSession session = null;
* ClassMapper classMapper = null;

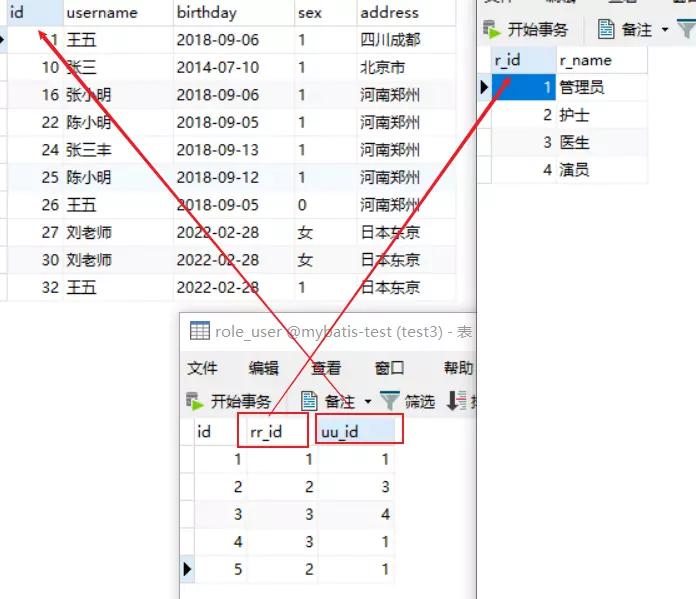
5

* @Before
* public void init(){
* session = MyBatisUtils.getSession();
* //um就是Mapper的实现类

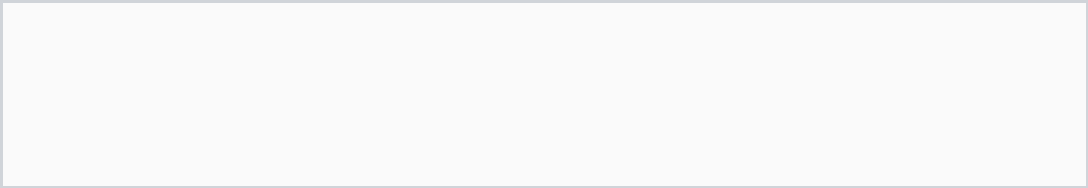
1. classMapper = session.getMapper(ClassMapper.class);
2. }
3. @After
4. public void destory(){
5. session.close();
6. }
7. @Test
8. public void test1(){
9. Classes stuList = classMapper.getStuList(1);
10. System.out.println(stuList);
11. }

6.4 多对多关联映射

1.需求



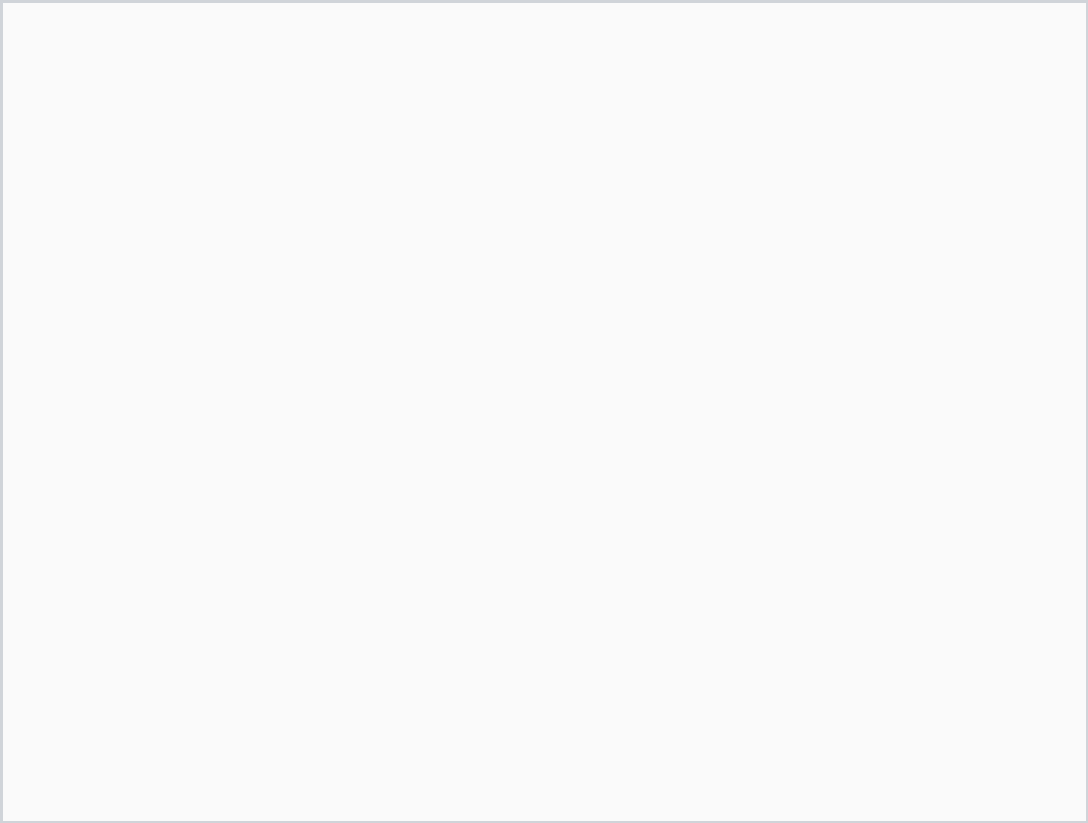
2.准备SQL 语句



* SELECT \* from `user` u
* INNER JOIN role\_user ru ON u.id = ru.uu\_id

3 INNER JOIN role r on r.r\_id = ru.rr\_id 4 WHERE u.id = 1

3.实体类



* public class User {
* private Integer id;
* private String username;

4private Date birthday;

5private String sex;

6private String address;

* }

8

* public class Role { 10 private long rId;

11private String rName;

1. }

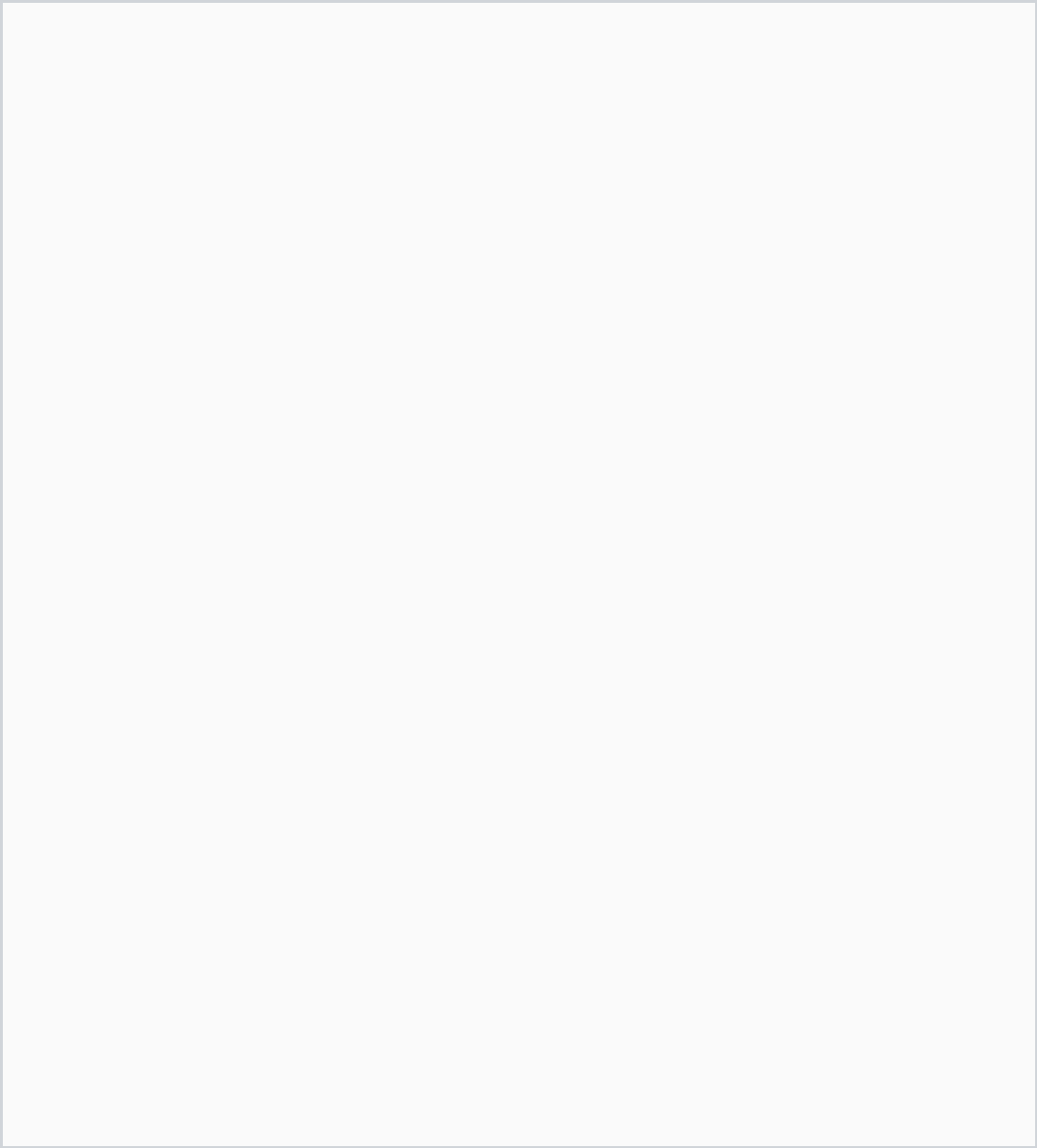
13

1. public class RoleUser {
2. private Integer id;
3. private Integer rr\_id;
4. private Integer uu\_id;

18

1. }

4.UserMapper.xml



* <?xml version="1.0" encoding="UTF-8"?>

2 <!DOCTYPE mapper

3PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

* "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> 5 <mapper namespace="com.dao.UserMapper">

6<resultMap id="getUserMap" type="User">

7<id property="id" column="id"/>

8<result property="username" column="username"/>

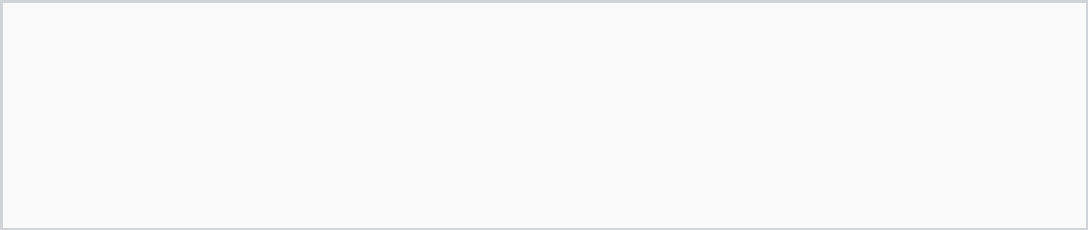
9<result property="birthday" column="birthday"/>

1. <result property="sex" column="sex"/>
2. <result property="address" column="address"/>
3. <collection property="roles" ofType="Role">
4. <id property="rId" column="r\_id"/>
5. <result property="rName" column="r\_name"/>
6. </collection>
7. </resultMap>

|  |  |  |
| --- | --- | --- |
| 17 | <!-- | SELECT \* FROM `user` U INNER JOIN role\_user RU ON U.u\_id=R |
|  | U.uu\_\_id |  |
| 18 |  | INNER JOIN role R ON RU.rr\_id=R.r\_id |

1. where U.u\_id=#{uid}
2. -->
3. <select id="getUserById" resultMap="getUserMap" parameterType="int">
4. SELECT \* from `user` u
5. INNER JOIN role\_user ru ON u.id = ru.uu\_id
6. INNER JOIN role r on r.r\_id = ru.rr\_id
7. WHERE u.id = #{id}
8. </select>
9. </mapper>

测试



* @Test
* public void test1(){
* User userById = userMapper.getUserById(24);
* System.out.println(userById);
* }

七.MyBatis 性能优化

7.1 什么是延迟加载(按需加载)

resultMap中的association（has a）和collection（has some）标签具有延迟加载的功能。



延迟加载：在真正使用数据的时候才发起查询，不用的时候不查询关联的数据，

延迟加载又叫按需查询（懒加载）

立即加载：不管用不用，只要一调用方法，马上发起查询。

使用场景：在对应的四种表关系中，一对多、多对多通常情况下采用延迟加载，多对一、一对一通常情况下采用立即加载。

7.2 怎么设置延迟加载?

Mybatis**默认是没开启延迟加载功能的，我们需要手动开启。**

**需要在**SqlMapConfig.xml**、文件中，在**<settings>**标签中开启延迟加载功能。** lazyLoadingEnabled aggressiveLazyLoading

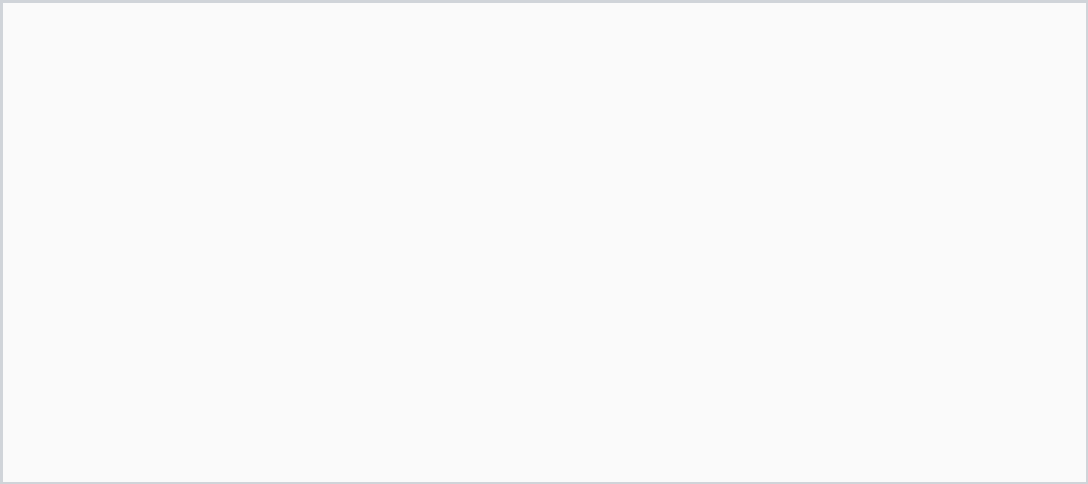
|  |  |  |  |
| --- | --- | --- | --- |
| 设置项 | 描述 | 允许值 | 默认值 |
|  |  |  |  |
| lazyLoadingEnabled | 延迟加载的全局开关。当 | true | false | false |
|  | 开启时，所有关联对象都 |  |  |
|  | 会延迟加载。特定关联关 |  |  |
|  | 系中可通过设 |  |  |
|  | 置 fetchType 属性来覆盖 |  |  |
|  | 该项的开关状态。 |  |  |
|  |  |  |  |
| aggressiveLazyLoading | 当开启时，任何方法的调 | true | false | false |
|  | 用都会加载该对象的所有 |  | （在 |
|  | 属性。否则，每个属性会 |  | 3.4.1 及 |
|  | 按需加载（参 |  | 之前的版 |
|  | 考 lazyLoadTriggerMethod |  | 本默认值 |
|  | s)。 |  | 为 true） |
|  |  |  |  |

lazyLoadingEnabled 默认值为false,那么在有级联关系的resultMap里，查询后会加载出所有的级联关系，当然有时候我们并不需要这些所有的时候，我们就可以应用到延迟加载给我们带来的好处了。

aggressiveLazyLoading默认值是true，这里我称之为分层加载，大概意思是如果它为true，那么当我使用了延迟加载，要么所有级联都不加载，要么如果我加载一个，其他都得加载.

7.3 延迟加载操作

1.实体类



* @Data
* @Getter

3 @Setter

4 @ToString

5 public class Account implements Serializable {

6private Integer id;

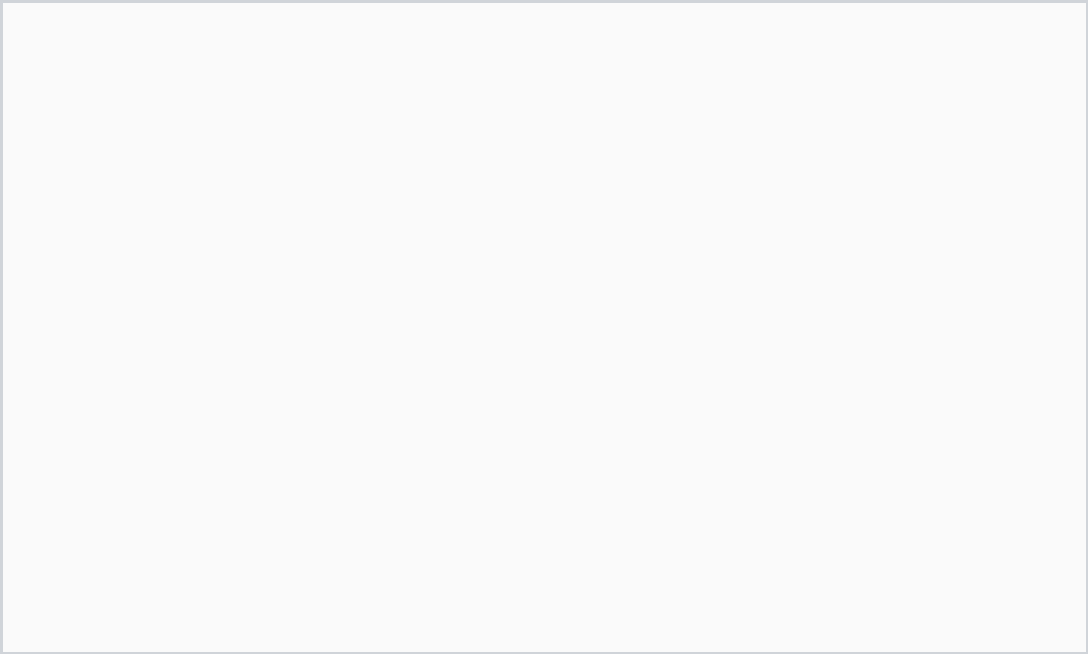
7private Integer uid;

8private Double money;

* //从表实体应该包含一个主表实体的对象引用

10private User user;

1. }



* @Data
* @Getter

3 @Setter

4 @ToString

5 public class User implements Serializable {

6private Integer id;

7private String username;

8private String telephone;

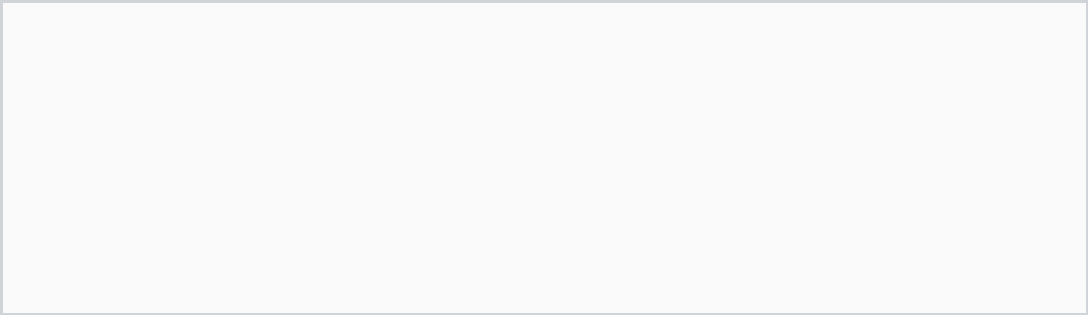
9private Date birthday;

1. private String gender;
2. private String address;
3. //一对多关系映射，主表实体应该包含从表实体的集合引用
4. private List<Account> accounts;

14

1. }

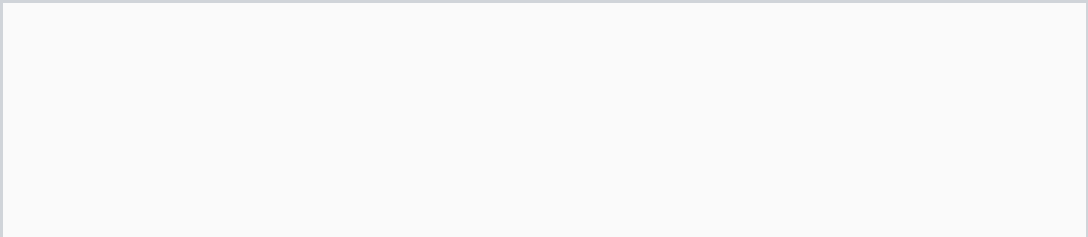
2.接口



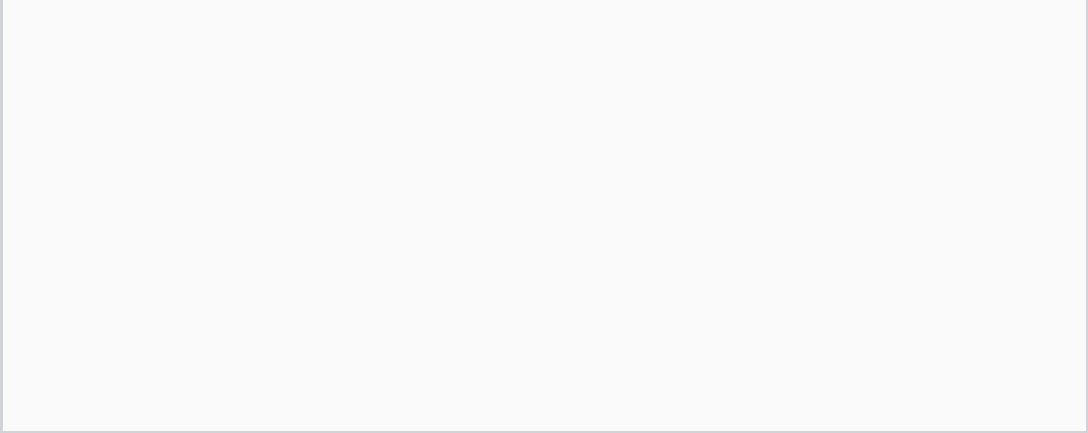
* public interface AccountMapper {
* /\*\*
* \* 查询所有账户

4\* @return

* \*/
* List<Account> findAll();
* }



* public interface UserMapper {
* /\*\*
* \* 查询所有用户信息,同时显示出该用户下的所有账户
* \*
* \* @return

* \*/
* List<User> findAll();

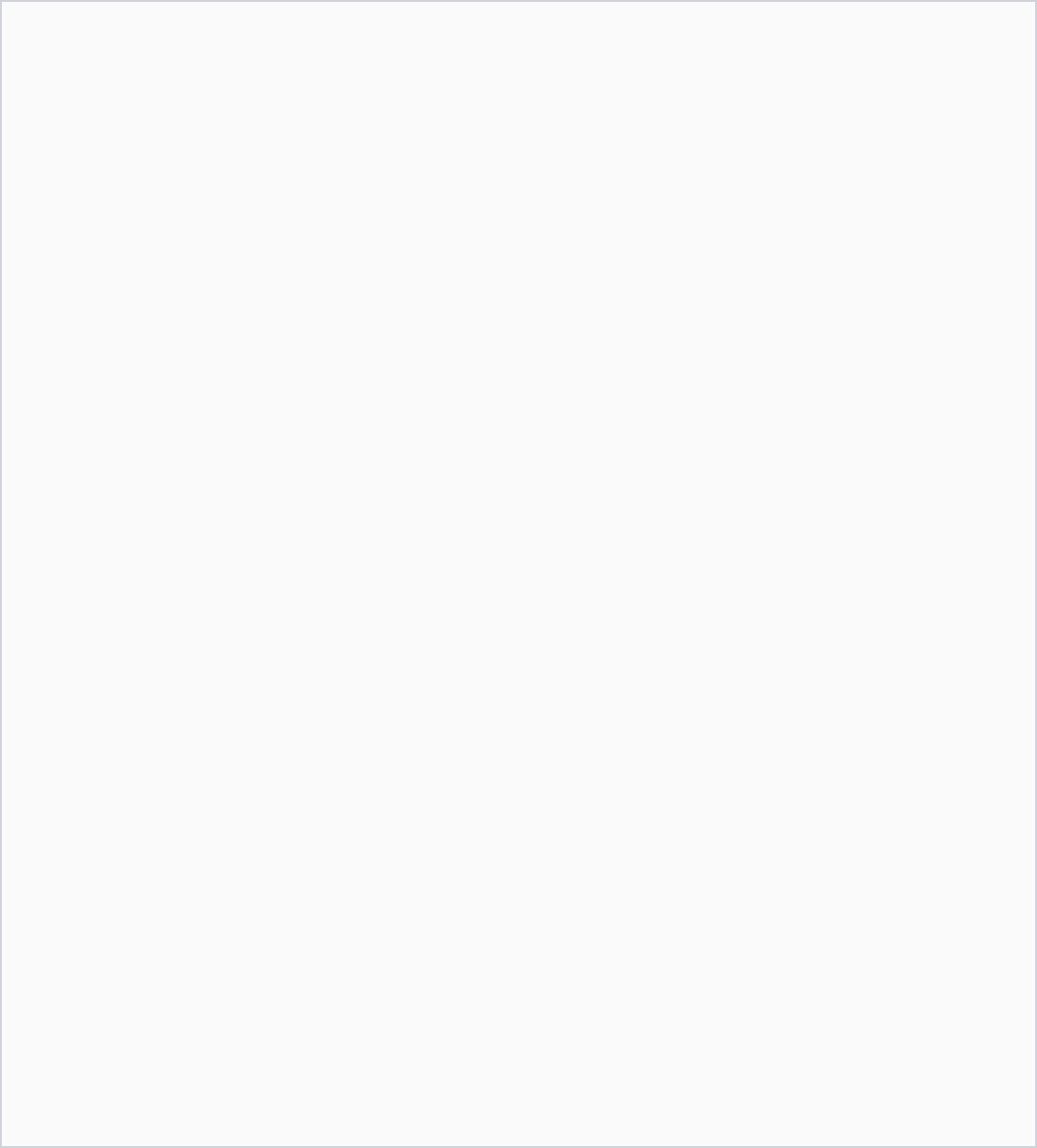
8

* /\*\*

1. \* 根据id查询用户信息
2. \* @param userId
3. \* @return
4. \*/
5. User findById(Integer userId);
6. }

3.mapper.xml 映射文件

UserMapper.xml



* <?xml version="1.0" encoding="UTF-8"?>

2 <!DOCTYPE mapper

3PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

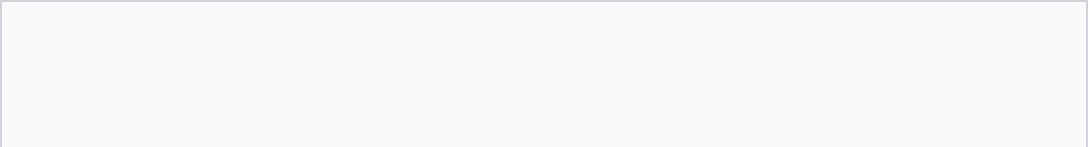
* "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> 5 <mapper namespace="com.dao.UserMapper">
* <!-- 定义User的resultMap-->
* <resultMap id="userAccountMap" type="User">

8<id property="id" column="id"></id>

9<result property="username" column="username"></result>

1. <result property="telephone" column="telephone"></result>
2. <result property="birthday" column="birthday"></result>
3. <result property="gender" column="gender"></result>
4. <result property="address" column="address"></result>
5. <collection property="accounts" ofType="Account">
6. <id property="id" column="aid"></id>
7. <result property="uid" column="uid"></result>
8. <result property="money" column="money"></result>
9. </collection>
10. </resultMap>
11. <!-- 查询所有用户 并且显示对应账户信息 -->
12. <select id="findAll" resultMap="userAccountMap">
13. SELECT u.\*,a.id as aid,a.uid,a.money FROM user u LEFT OUTER JOIN acc
14. </select>
15. <!-- 根据id查询用户 -->
16. <select id="findById" parameterType="INT" resultType="User">
17. select \* from user where id = #{uid}
18. </select>
19. </mapper>

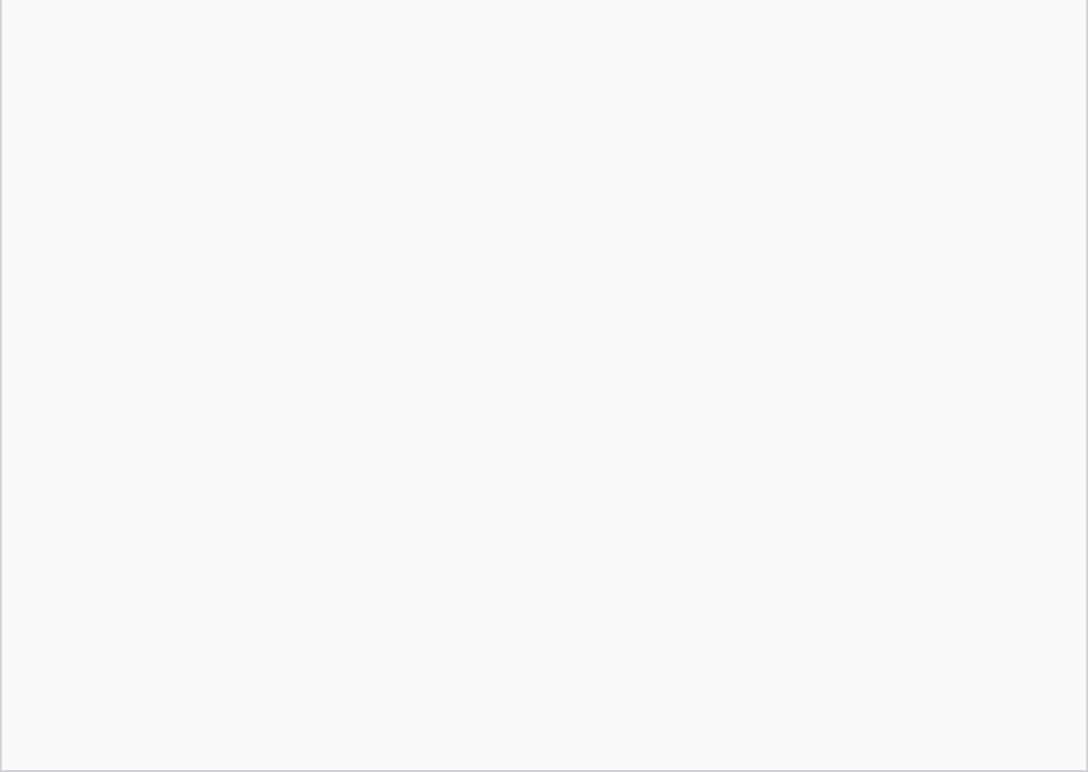
AccountMap.xml



* <?xml version="1.0" encoding="UTF-8"?>

2 <!DOCTYPE mapper

3PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

* "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> 5 <mapper namespace="com.dao.AccountMapper">

6<!-- 定义封装 Account和User 的resultMap -->

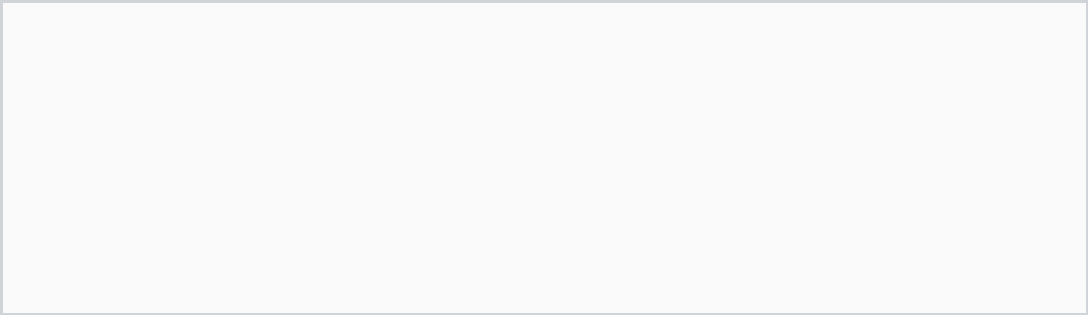
7<resultMap id="userAccountMap" type="Account">

8<id property="id" column="id"></id>

9<result property="uid" column="uid"></result>

1. <result property="money" column="money"></result>
2. <!-- 配置封装 User 的内容
3. select：查询用户的唯一标识
4. column：用户根据id查询的时候，需要的参数值
5. -->
6. <association property="user" column="uid" javaType="User" select="co
7. </resultMap>
8. <!-- 根据查询所有账户 -->
9. <select id="findAll" resultMap="userAccountMap">
10. SELECT \* FROM account
11. </select>
12. </mapper>

4.测试



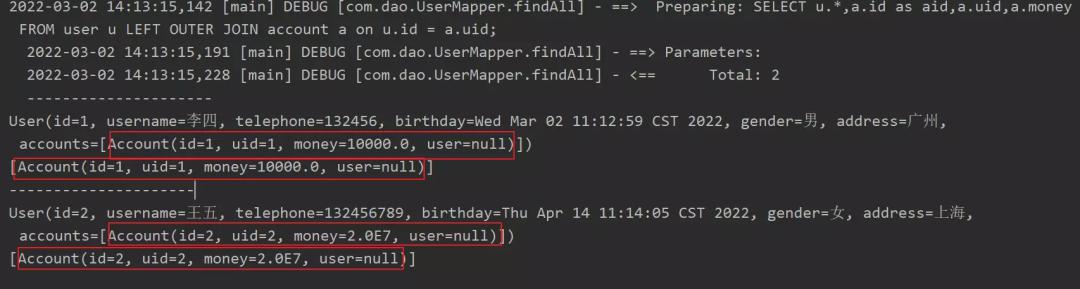
* @Test
* public void testFindAll(){
* List<Account> accounts = accountMapper.findAll();

4//for (Account a :accounts){

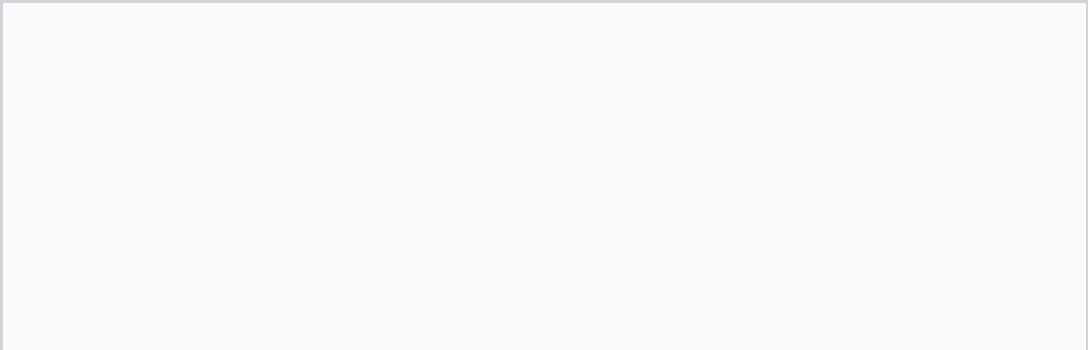
|  |  |  |
| --- | --- | --- |
| 5 | // | System.out.println(a); |

* //}
* }

效果



5.修改MapperConfi.xml

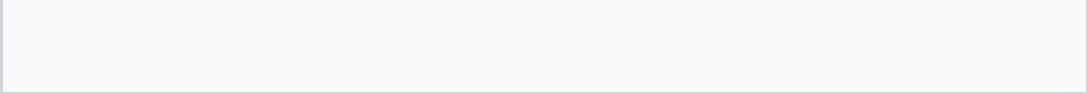


* <!--全局设置-->

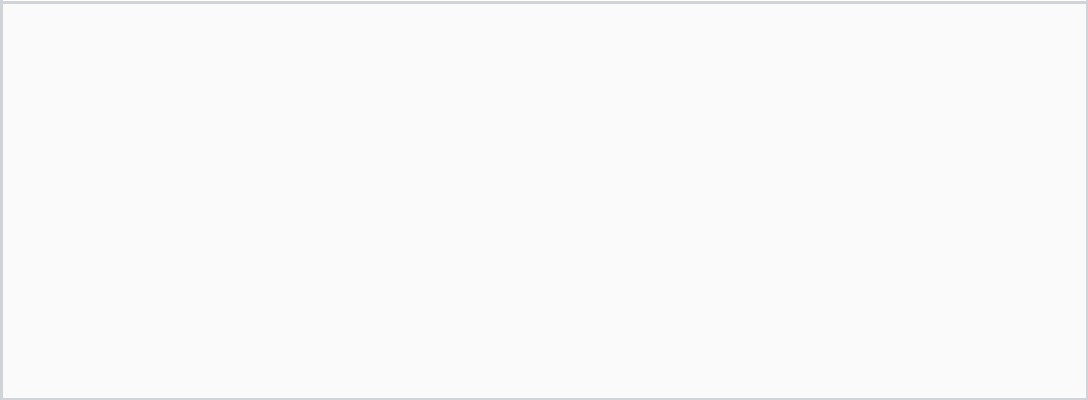
2 <settings>

* <!--开启延迟加载-->
* <setting name="lazyLoadingEnabled" value="true"/>
* <!--false 默认，表示就按需加载 true 如果加载一个关联属性，那么属性就全部立即加载！-->
* <setting name="aggressiveLazyLoading" value="false"/>

7<!--打开二级缓存 开启所有的二级缓存-->

* <setting name="cacheEnabled" value="true"/> 9 </settings>

6.测试



* @Test
* public void testFindAll(){
* List<Account> accounts = accountMapper.findAll();

4for (Account a :accounts){

* System.out.println("---------------------");
* System.out.println(a);
* System.out.println(a.getUser());
* }
* }

