Mybatis

# Mybatis介绍



MyBatis是支持***普通SQL查询***，***存储过程***和***高级映射***的优秀持久层框架。MyBatis消除了几乎所有的JDBC代码和参数的手工设置以及对结果集的检索封装。MyBatis可以使用简单的XML或注解用于配置和原始映射，将接口和Java的POJO（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录.

**JDBC-🡪dbutils(自动封装)-🡪MyBatis-🡪Hibernate**

xml

# mybatis快速入门

**编写第一个基于mybaits的测试例子:**

## 2.1. 添加jar包

|  |
| --- |
| **【mybatis】**  mybatis-3.1.1.jar  **【MYSQL驱动包】** mysql-connector-java-5.1.7-bin.jar |

## 2.2. 建库+表

|  |
| --- |
| create database mybatis;  use mybatis;  CREATE TABLE users(id INT PRIMARY KEY AUTO\_INCREMENT, NAME VARCHAR(20), age INT);  INSERT INTO users(NAME, age) VALUES('Tom', 12);  INSERT INTO users(NAME, age) VALUES('Jack', 11); |

## 添加Mybatis的配置文件conf.xml

|  |
| --- |
| <?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>  <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/mybatis" />  <property name="username" value="root" />  <property name="password" value="root" />  </dataSource>  </environment>  </environments>  </configuration> |

## 定义表所对应的实体类

|  |
| --- |
| public class User {  private int id;  private String name;  private int age;  //get,set方法  } |

## 定义操作users表的sql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.atguigu.mybatis\_test.test1.userMapper">  <select id="getUser" parameterType="int"  resultType="com.atguigu.mybatis\_test.test1.User">  select \* from users where id=#{id}  </select>  </mapper> |

## 在conf.xml文件中注册userMapper.xml文件

|  |
| --- |
| <mappers>  <mapper resource="com/atguigu/mybatis\_test/test1/userMapper.xml"/>  </mappers> |

## 编写测试代码：执行定义的select语句

|  |
| --- |
| public class Test {  public static void main(String[] args) throws IOException {  String resource = "conf.xml";  //加载mybatis的配置文件（它也加载关联的映射文件）  Reader reader = Resources.getResourceAsReader(resource);  //构建sqlSession的工厂  SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(reader);  //创建能执行映射文件中sql的sqlSession  SqlSession session = sessionFactory.openSession();  //映射sql的标识字符串  String statement = "com.atguigu.mybatis.bean.userMapper"+".selectUser";  //执行查询返回一个唯一user对象的sql  User user = session.selectOne(statement, 1);  System.out.println(user);  }  } |

# 3. 操作users表的CRUD

## 3.1. XML的实现

### 1). 定义sql映射xml文件：

|  |
| --- |
| <insert id="insertUser" parameterType="com.atguigu.ibatis.bean.User">  insert into users(name, age) values(#{name}, #{age});  </insert>  <delete id="deleteUser" parameterType="int">  delete from users where id=#{id}  </delete>    <update id="updateUser" parameterType="com.atguigu.ibatis.bean.User">  update users set name=#{name},age=#{age} where id=#{id}  </update>    <select id="selectUser" parameterType="int" resultType="com.atguigu.ibatis.bean.User">  select \* from users where id=#{id}  </select>    <select id="selectAllUsers" resultType="com.atguigu.ibatis.bean.User">  select \* from users  </select> |

### 2). 在config.xml中注册这个映射文件

|  |
| --- |
| <mapper resource=" com/atguigu/ibatis/bean/userMapper.xml"/> |

### 3). 在dao中调用：

|  |
| --- |
| public User getUserById(int id) {  SqlSession session = sessionFactory.openSession();  User user = session.selectOne(URI+".selectUser", id);  return user;  } |

## 3.2. 注解的实现

### 1). 定义sql映射的接口

|  |
| --- |
| public interface UserMapper {  @Insert("insert into users(name, age) values(#{name}, #{age})")  public int insertUser(User user);  @Delete("delete from users where id=#{id}")  public int deleteUserById(int id);    @Update("update users set name=#{name},age=#{age} where id=#{id}")  public int updateUser(User user);  @Select("select \* from users where id=#{id}")  public User getUserById(int id);  @Select("select \* from users")  public List<User> getAllUser();  } |

### 2). 在config中注册这个映射接口

|  |
| --- |
| <mapper class="com.atguigu.ibatis.crud.ano.UserMapper"/> |

### 3). 在dao类中调用

|  |
| --- |
| public User getUserById(int id) {  SqlSession session = sessionFactory.openSession();  UserMapper mapper = session.getMapper(UserMapper.class);  User user = mapper.getUserById(id);  return user;  } |

# 4. 几个可以优化的地方

## 4.1. 连接数据库的配置单独放在一个properties文件中

|  |
| --- |
| **## db.properties**  <properties resource="db.properties"/>  <property name="driver" value="${driver}" />  <property name="url" value="${url}" />  <property name="username" value="${username}" />  <property name="password" value="${password}" /> |

## 4.2. 为实体类定义别名,简化sql映射xml文件中的引用

|  |
| --- |
| <typeAliases>  <typeAlias type="com.atguigu.ibatis.bean.User" alias="\_User"/>  </typeAliases> |

## 4.3. 可以在src下加入log4j的配置文件,打印日志信息

|  |
| --- |
| **1. 添加jar:**  log4j-1.2.16.jar  **2.1. log4j.properties(方式一)**  log4j.properties，  log4j.rootLogger=DEBUG, Console  #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  log4j.logger.java.sql.ResultSet=INFO  log4j.logger.org.apache=INFO  log4j.logger.java.sql.Connection=DEBUG  log4j.logger.java.sql.Statement=DEBUG  log4j.logger.java.sql.PreparedStatement=DEBUG  **2.2. log4j.xml(方式二)**  <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">  <log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">  <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">  <layout class="org.apache.log4j.PatternLayout">  <param name="ConversionPattern"  value="%-5p %d{MM-dd HH:mm:ss,SSS} %m (%F:%L) \n" />  </layout>  </appender>  <logger name="java.sql">  <level value="debug" />  </logger>  <logger name="org.apache.ibatis">  <level value="debug" />  </logger>  <root>  <level value="debug" />  <appender-ref ref="STDOUT" />  </root>  </log4j:configuration> |

# 5. 解决字段名与实体类属性名不相同的冲突

## 5.1. 准备表和数据：

|  |
| --- |
| CREATE TABLE orders(  order\_id INT PRIMARY KEY AUTO\_INCREMENT,  order\_no VARCHAR(20),  order\_price FLOAT  );  INSERT INTO orders(order\_no, order\_price) VALUES('aaaa', 23);  INSERT INTO orders(order\_no, order\_price) VALUES('bbbb', 33);  INSERT INTO orders(order\_no, order\_price) VALUES('cccc', 22); |

## 5.2. 定义实体类：

|  |
| --- |
| public class Order {  private int id;  private String orderNo;  private float price;  } |

## 5.3. 实现getOrderById(id)的查询：

|  |
| --- |
| **方式一: 通过在sql语句中定义别名**  <select id="selectOrder" parameterType="int" resultType="\_Order">  select order\_id id, order\_no orderNo,order\_price price from orders where order\_id=#{id}  </select>    **方式二: 通过<resultMap>**  <select id="selectOrderResultMap" parameterType="int" resultMap="orderResultMap">  select \* from orders where order\_id=#{id}  </select>  <**resultMap** type="\_Order" id="orderResultMap">  <id property="id" column="order\_id"/>  <result property="orderNo" column="order\_no"/>  <result property="price" column="order\_price"/>  </resultMap> |

# 6.实现关联表查询

## 6.1. 一对一关联

### 1). 提出需求

根据班级id查询班级信息(带老师的信息)

### 2). 创建表和数据

|  |
| --- |
| CREATE TABLE teacher(  t\_id INT PRIMARY KEY AUTO\_INCREMENT,  t\_name VARCHAR(20)  );  CREATE TABLE class(  c\_id INT PRIMARY KEY AUTO\_INCREMENT,  c\_name VARCHAR(20),  teacher\_id INT  );  ALTER TABLE class ADD CONSTRAINT fk\_teacher\_id FOREIGN KEY (teacher\_id) REFERENCES teacher(t\_id);  INSERT INTO teacher(t\_name) VALUES('LS1');  INSERT INTO teacher(t\_name) VALUES('LS2');  INSERT INTO class(c\_name, teacher\_id) VALUES('bj\_a', 1);  INSERT INTO class(c\_name, teacher\_id) VALUES('bj\_b', 2); |

### 3). 定义实体类：

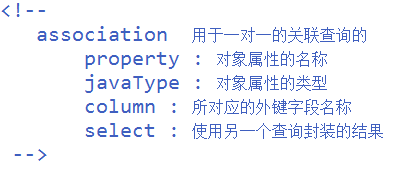
|  |
| --- |
| public class Teacher {  private int id;  private String name;  }  public class Classes {  private int id;  private String name;  private Teacher teacher;  } |

### 4). 定义sql映射文件ClassMapper.xml

|  |
| --- |
| <!--  方式一：**嵌套结果**：使用嵌套结果映射来处理重复的联合结果的子集  封装联表查询的数据(去除重复的数据)  select \* from class c, teacher t where c.teacher\_id=t.t\_id and c.c\_id=1  -->  <select id="getClass" parameterType="int" resultMap="ClassResultMap">  select \* from class c, teacher t where c.teacher\_id=t.t\_id and c.c\_id=#{id}  </select>  <resultMap type="\_Classes" id="ClassResultMap">  <id property="id" column="c\_id"/>  <result property="name" column="c\_name"/>  **<association** property="teacher" **javaType**="\_Teacher">  <id property="id" column="t\_id"/>  <result property="name" column="t\_name"/>  </association>  </resultMap>  <!--  方式二：**嵌套查询**：通过执行另外一个SQL映射语句来返回预期的复杂类型  SELECT \* FROM class WHERE c\_id=1;  SELECT \* FROM teacher WHERE t\_id=1 //1 是上一个查询得到的teacher\_id的值  -->  <select id="getClass2" parameterType="int" resultMap="ClassResultMap2">  select \* from class where c\_id=#{id}  </select>  <resultMap type="\_Classes" id="ClassResultMap2">  <id property="id" column="c\_id"/>  <result property="name" column="c\_name"/>  <association property="teacher" column="teacher\_id" **select**="getTeacher">  </association>  </resultMap>    <select id="getTeacher" parameterType="int" resultType="\_Teacher">  SELECT t\_id id, t\_name name FROM teacher WHERE t\_id=#{id}  </select> |

### 5). 测试

|  |
| --- |
| @Test  public void testOO() {  SqlSession sqlSession = factory.openSession();  Classes c = sqlSession.selectOne("com.atguigu.day03\_mybatis.test5.OOMapper.getClass", 1);  System.out.println(c);  }  @Test  public void testOO2() {  SqlSession sqlSession = factory.openSession();  Classes c = sqlSession.selectOne("com.atguigu.day03\_mybatis.test5.OOMapper.getClass2", 1);  System.out.println(c);  } |



## 6.2. 一对多关联

### 1). 提出需求

根据classId查询对应的班级信息,包括学生,老师

### 2). 创建表和数据：

|  |
| --- |
| CREATE TABLE student(  s\_id INT PRIMARY KEY AUTO\_INCREMENT,  s\_name VARCHAR(20),  class\_id INT  );  INSERT INTO student(s\_name, class\_id) VALUES('xs\_A', 1);  INSERT INTO student(s\_name, class\_id) VALUES('xs\_B', 1);  INSERT INTO student(s\_name, class\_id) VALUES('xs\_C', 1);  INSERT INTO student(s\_name, class\_id) VALUES('xs\_D', 2);  INSERT INTO student(s\_name, class\_id) VALUES('xs\_E', 2);  INSERT INTO student(s\_name, class\_id) VALUES('xs\_F', 2); |

### 3). 定义实体类：

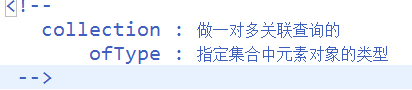
|  |
| --- |
| public class Student {  private int id;  private String name;  }  public class Classes {  private int id;  private String name;  private Teacher teacher;  private List<Student> students;  } |

### 4). 定义sql映射文件ClassMapper.xml

|  |
| --- |
| <!--  方式一: 嵌套结果: 使用嵌套结果映射来处理重复的联合结果的子集  SELECT \* FROM class c, teacher t,student s WHERE c.teacher\_id=t.t\_id AND c.C\_id=s.class\_id AND c.c\_id=1  -->  <select id="getClass3" parameterType="int" resultMap="ClassResultMap3">  select \* from class c, teacher t,student s where c.teacher\_id=t.t\_id and c.C\_id=s.class\_id and c.c\_id=#{id}  </select>  <resultMap type="\_Classes" id="ClassResultMap3">  <id property="id" column="c\_id"/>  <result property="name" column="c\_name"/>  <association property="teacher" column="teacher\_id" javaType="\_Teacher">  <id property="id" column="t\_id"/>  <result property="name" column="t\_name"/>  </association>  <!-- ofType指定students集合中的对象类型 -->  <**collection** property="students" **ofType**="\_Student">  <id property="id" column="s\_id"/>  <result property="name" column="s\_name"/>  </collection>  </resultMap>  <!--  方式二：嵌套查询：通过执行另外一个SQL映射语句来返回预期的复杂类型  SELECT \* FROM class WHERE c\_id=1;  SELECT \* FROM teacher WHERE t\_id=1 //1 是上一个查询得到的teacher\_id的值  SELECT \* FROM student WHERE class\_id=1 //1是第一个查询得到的c\_id字段的值  -->  <select id="getClass4" parameterType="int" resultMap="ClassResultMap4">  select \* from class where c\_id=#{id}  </select>  <resultMap type="\_Classes" id="ClassResultMap4">  <id property="id" column="c\_id"/>  <result property="name" column="c\_name"/>  <association property="teacher" column="teacher\_id" javaType="\_Teacher" select="getTeacher2"></association>  <**collection** **property**="students" **ofType**="\_Student" **column**="c\_id" **select**="getStudent"></collection>  </resultMap>    <select id="getTeacher2" parameterType="int" resultType="\_Teacher">  SELECT t\_id id, t\_name name FROM teacher WHERE t\_id=#{id}  </select>    <select id="getStudent" parameterType="int" resultType="\_Student">  SELECT s\_id id, s\_name name FROM student WHERE class\_id=#{id}  </select> |

### 5). 测试：

|  |
| --- |
| @Test  public void testOM() {  SqlSession sqlSession = factory.openSession();  Classes c = sqlSession.selectOne("com.atguigu.day03\_mybatis.test5.OOMapper.getClass3", 1);  System.out.println(c);  }  @Test  public void testOM2() {  SqlSession sqlSession = factory.openSession();  Classes c = sqlSession.selectOne("com.atguigu.day03\_mybatis.test5.OOMapper.getClass4", 1);  System.out.println(c);  } |



# 7. 动态SQL与模糊查询

## 7.1. 提出需求:

实现多条件查询用户(姓名模糊匹配, 年龄在指定的最小值到最大值之间)

## 7.2. 准备数据表和数据:

|  |
| --- |
| create table d\_user(  id int primary key auto\_increment,  name varchar(10),  age int(3)  );  insert into d\_user(name,age) values('Tom',12);  insert into d\_user(name,age) values('Bob',13);  insert into d\_user(name,age) values('Jack',18); |

## 7.3. ConditionUser(查询条件实体类)

|  |
| --- |
| private String name;  private int minAge;  private int maxAge; |

## 7.4. User(表实体类)

|  |
| --- |
| private int id;  private String name;  private int age; |

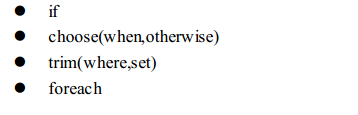
## 7.5. userMapper.xml(映射文件)

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace="com.atguigu.day03\_mybatis.test6.userMapper">  <select id="getUser" parameterType="com.atguigu.day03\_mybatis.test6.ConditionUser" resultType="com.atguigu.day03\_mybatis.test6.User">  select \* from d\_user where age>=#{minAge} and age&lt;=#{maxAge}  <if test='name!="%null%"'>and name like #{name}</if>  </select>  </mapper> |

## 7.6. UserTest(测试)

|  |
| --- |
| public class UserTest {  public static void main(String[] args) throws IOException {    Reader reader = Resources.getResourceAsReader("conf.xml");    SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(reader);    SqlSession sqlSession = sessionFactory.openSession();    String statement = "com.atguigu.day03\_mybatis.test6.userMapper.getUser";    List<User> list = sqlSession.selectList(statement, new ConditionUser("%a%", 1, 12));    System.out.println(list);  }  } |

## MyBatis中可用的动态SQL标签



# 8.调用存储过程

## 8.1. 提出需求:

**查询得到男性或女性的数量, 如果传入的是0就女性否则是男性**

## 8.2. 准备数据库表和存储过程:

|  |
| --- |
| create table p\_user(  id int primary key auto\_increment,  name varchar(10),  sex char(2)  );  insert into p\_user(name,sex) values('A',"男");  insert into p\_user(name,sex) values('B',"女");  insert into p\_user(name,sex) values('C',"男");  #创建存储过程(**查询得到男性或女性的数量, 如果传入的是0就女性否则是男性**)  DELIMITER $  CREATE PROCEDURE mybatis.ges\_user\_count(IN sex\_id INT, OUT user\_count INT)  BEGIN  IF sex\_id=0 THEN  SELECT COUNT(\*) FROM mybatis.p\_user WHERE p\_user.sex='女' INTO user\_count;  ELSE  SELECT COUNT(\*) FROM mybatis.p\_user WHERE p\_user.sex='男' INTO user\_count;  END IF;  END  $  #调用存储过程  DELIMITER ;  SET @user\_count = 0;  CALL mybatis.ges\_user\_count(1, @user\_count);  SELECT @user\_count; |

## 8.3. 创建表的实体类

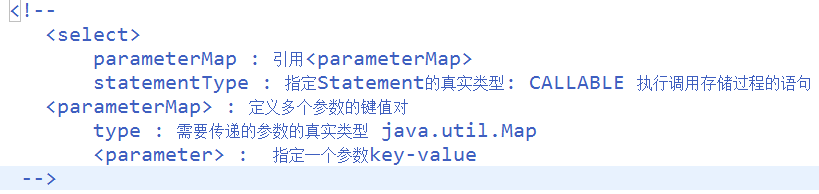
|  |
| --- |
| public class User {  private String id;  private String name;  private String sex;  } |

## 8.4. userMapper.xml

|  |
| --- |
| <mapper namespace=*"com.atguigu.mybatis.test7.userMapper"*>  <!--  查询得到男性或女性的数量, 如果传入的是0就女性否则是男性  CALL mybatis.get\_user\_count(1, @user\_count);  -->  <select id=*"getCount"* statementType=*"CALLABLE"* parameterMap=*"getCountMap"*>  call mybatis.get\_user\_count(?,?)  </select>  <parameterMap type=*"java.util.Map"* id=*"getCountMap"*>  <parameter property=*"sex\_id"* mode=*"IN"* jdbcType=*"INTEGER"*/>  <parameter property=*"user\_count"* mode=*"OUT"* jdbcType=*"INTEGER"*/>  </parameterMap>  </mapper> |

## 8.5. 测试调用:

|  |
| --- |
| Map<String, Integer> paramMap = **new** HashMap<>();  paramMap.put("sex\_id", 0);  session.selectOne(statement, paramMap);    Integer userCount = paramMap.get("user\_count");  System.*out*.println(userCount); |



# 9. Mybatis缓存

## 9.1. 理解MyBatis缓存

正如大多数持久层框架一样，MyBatis 同样提供了**一级缓存**和**二级缓存**的支持

|  |
| --- |
| 1. 一级缓存: 基于PerpetualCache 的 HashMap本地缓存，其存储作用域为 **Session**，当 **Session flush 或 close** 之后，该Session中的所有 Cache 就将清空。   2. 二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap存储，不同在于其存储作用域为 **Mapper(Namespace)**，并且**可自定义存储源**，如 Ehcache。  3. 对于缓存数据更新机制，当某一个作用域(一级缓存Session/二级缓存Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被clear。 |

## 9.2. Mybatis一级缓存

### 1) 提出需求:

根据id查询对应的用户记录对象

### 2). 准备数据库表和数据

|  |
| --- |
| CREATE TABLE c\_user(  id INT PRIMARY KEY AUTO\_INCREMENT,  NAME VARCHAR(20),  age INT  );  INSERT INTO c\_user(NAME, age) VALUES('Tom', 12);  INSERT INTO c\_user(NAME, age) VALUES('Jack', 11); |

### 3). 创建表的实体类

|  |
| --- |
| public class User implements Serializable{  private int id;  private String name;  private int age;  } |

### 4). 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.atguigu.mybatis.test8.userMapper">  <select id="getUser" parameterType="int" resultType="\_CUser">  select \* from c\_user where id=#{id}  </select>  <update id="updateUser" parameterType="\_CUser">  update c\_user set  name=#{name}, age=#{age} where id=#{id}  </update>  </mapper> |

### 5). 测试:

|  |
| --- |
| /\*  \* 一级缓存: 也就Session级的缓存(默认开启)  \*/  @Test  public void testCache1() {  SqlSession session = MybatisUtils.getSession();  String statement = "com.atguigu.mybatis.test8.userMapper.getUser";  User user = session.selectOne(statement, 1);  System.out.println(user);    /\*  \* 一级缓存默认就会被使用  \*/  /\*  user = session.selectOne(statement, 1);  System.out.println(user);  \*/    /\*  1. 必须是同一个Session,如果session对象已经close()过了就不可能用了  \*/  /\*  session = MybatisUtils.getSession();  user = session.selectOne(statement, 1);  System.out.println(user);  \*/    /\*  2. 查询条件是一样的  \*/  /\*  user = session.selectOne(statement, 2);  System.out.println(user);  \*/    /\*  3. 没有执行过session.clearCache()清理缓存  \*/  /\*  session.clearCache();  user = session.selectOne(statement, 2);  System.out.println(user);  \*/    /\*  4. 没有执行过增删改的操作(这些操作都会清理缓存)  \*/  /\*  session.update("com.atguigu.mybatis.test8.userMapper.updateUser",  new User(2, "user", 23));  user = session.selectOne(statement, 2);  System.out.println(user);  \*/  } |

## 9.3. Mybatis二级缓存

### 1). 添加一个<cache>在userMapper.xml中

|  |
| --- |
| <mapper namespace="com.atguigu.mybatis.test8.userMapper">  **<cache/>** |

### 2). 测试

|  |
| --- |
| /\*  \* 测试二级缓存  \*/  @Test  public void testCache2() {  String statement = "com.atguigu.mybatis.test8.userMapper.getUser";  SqlSession session = MybatisUtils.getSession();  User user = session.selectOne(statement, 1);  session.commit();  System.out.println("user="+user);    SqlSession session2 = MybatisUtils.getSession();  user = session2.selectOne(statement, 1);  session.commit();  System.out.println("user2="+user);  } |

### 3). 补充说明

|  |
| --- |
| 1. 映射语句文件中的所有select语句将会被缓存。  2. 映射语句文件中的所有insert，update和delete语句会刷新缓存。  3. 缓存会使用Least Recently Used（LRU，最近最少使用的）算法来收回。  4. 缓存会根据指定的时间间隔来刷新。  5. 缓存会存储1024个对象 |

|  |
| --- |
| <cache  eviction="FIFO" //回收策略为先进先出  flushInterval="60000" //自动刷新时间60s  size="512" //最多缓存512个引用对象  readOnly="true"/> //只读 |

# 10. spring集成mybatis

## 10.1. 添加Jar包

|  |
| --- |
| **【mybatis】**  mybatis-3.2.0.jar  **mybatis-spring-1.1.1.jar**  log4j-1.2.17.jar  **【spring】**  spring-aop-3.2.0.RELEASE.jar  spring-beans-3.2.0.RELEASE.jar  spring-context-3.2.0.RELEASE.jar  spring-core-3.2.0.RELEASE.jar  spring-expression-3.2.0.RELEASE.jar  spring-jdbc-3.2.0.RELEASE.jar  spring-test-3.2.4.RELEASE.jar  spring-tx-3.2.0.RELEASE.jar  aopalliance-1.0.jar  cglib-nodep-2.2.3.jar  commons-logging-1.1.1.jar  **【MYSQL驱动包】**  mysql-connector-java-5.0.4-bin.jar |

## 10.2. 数据库表

|  |
| --- |
| CREATE TABLE s\_user(  user\_id INT AUTO\_INCREMENT PRIMARY KEY,  user\_name VARCHAR(30),  user\_birthday DATE,  user\_salary DOUBLE  ) |

## 10.3. 实体类: User

|  |
| --- |
| **public** **class** User {  **private** **int** id;  **private** String name;  **private** Date birthday;  **private** **double** salary;    //set,get方法  } |

## 10.4. DAO接口: UserMapper (XXXMapper)

|  |
| --- |
| **public** **interface** UserMapper {  **void** save(User user);  **void** update(User user);  **void** delete(**int** id);  User findById(**int** id);  List<User> findAll();  } |

## 10.5. SQL映射文件: userMapper.xml(与接口忽略大小写同名)

|  |
| --- |
| <?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.atguigu.mybatis.test9.UserMapper">  <resultMap type="User" id="userResult">  <result column="user\_id" property="id"/>  <result column="user\_name" property="name"/>  <result column="user\_birthday" property="birthday"/>  <result column="user\_salary" property="salary"/>  </resultMap>  <!-- 取得插入数据后的id -->  <insert id="save" keyColumn="user\_id" keyProperty="id" useGeneratedKeys="true">  insert into s\_user(user\_name,user\_birthday,user\_salary)  values(#{name},#{birthday},#{salary})  </insert>  <update id="update">  update s\_user  set user\_name = #{name},  user\_birthday = #{birthday},  user\_salary = #{salary}  where user\_id = #{id}  </update>    <delete id="delete">  delete from s\_user  where user\_id = #{id}  </delete>  <select id="findById" resultMap="userResult">  select \*  from s\_user  where user\_id = #{id}  </select>    <select id="findAll" resultMap="userResult">  select \*  from s\_user  </select>  </mapper> |

## 10.6. spring的配置文件: beans.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-3.2.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx-3.2.xsd">  <!-- 1. 数据源 : DriverManagerDataSource -->  <bean id="dataSource"  class="org.springframework.jdbc.datasource.DriverManagerDataSource">  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>  <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>  <property name="username" value="root"/>  <property name="password" value="root"/>  </bean>    <!--  2. mybatis的SqlSession的工厂: SqlSessionFactoryBean  dataSource / typeAliasesPackage  -->  <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  <property name="dataSource" ref="dataSource"/>  <property name="typeAliasesPackage" value="com.atuigu.spring\_mybatis2.domain"/>  </bean>  <!--  3. mybatis自动扫描加载Sql映射文件 : MapperScannerConfigurer  sqlSessionFactory / basePackage  -->  <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  <property name="basePackage" value="com.atuigu.spring\_mybatis2.mapper"/>  <property name="sqlSessionFactory" ref="sqlSessionFactory"/>  </bean>    <!-- 4. 事务管理 : DataSourceTransactionManager -->  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"/>  </bean>  <!-- 5. 使用声明式事务 -->  <tx:annotation-driven transaction-manager="txManager" />    </beans> |

## 10.7. mybatis的配置文件: mybatis-config.xml

|  |
| --- |
| <?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>  <!-- Spring整合myBatis后，这个配置文件基本可以不要了-->  <!-- 设置外部配置文件 -->  <!-- 设置类别名 -->  <!-- 设置数据库连接环境 -->  <!-- 映射文件 -->  </configuration> |

## 10.8. 测试

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) //使用Springtest测试框架  @ContextConfiguration("/beans.xml") //加载配置  **public** **class** SMTest {  @Autowired //注入  **private** UserMapper userMapper;  @Test  **public** **void** save() {  User user = **new** User();  user.setBirthday(**new** Date());  user.setName("marry");  user.setSalary(300);  userMapper.save(user);  System.*out*.println(user.getId());  }  @Test  **public** **void** update() {  User user = userMapper.findById(2);  user.setSalary(2000);  userMapper.update(user);  }  @Test  **public** **void** delete() {  userMapper.delete(3);  }  @Test  **public** **void** findById() {  User user = userMapper.findById(1);  System.*out*.println(user);  }  @Test  **public** **void** findAll() {  List<User> users = userMapper.findAll();  System.*out*.println(users);  }  } |