# 初识MyBatis

## 起源

IBatis = Internet aBatis

2010年 ：IBatis ---〉MyBatis

## MyBatis是什么

* MyBatis是一个支持Java的持久层框架。
* MyBatis是一个ORM(**Object Relational Mapping**)框架
* MyBatis提供了Sql Map和Data Access Objects(daos)

## 发展历程

JDBC ----〉DBUtil ---〉MyBatis ---〉Hibernate

* JDBC ：通过Sql语句执行相应操作。数据需要自行封装
* DBUtil ：程序员编写Sql语句，DBUtil将数据封装为对象
* MyBatis ：程序员编写配置文件或注解，通过配置文件可以对数 据进行处理或自动将数据封装为对象
* Hibernate ：程序员编写配置文件，通过配置文件可以对数据进行处理或自动将数据封装为对象

## 可以做什么

MyBatis其实就是一个高级的DBUtil。

## 示例

1. 创建项目
2. 导入相关jar包

|  |
| --- |
|  |

1. 添加配置文件

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <!-- 环境配置 -->  <environments default=*"development"*>  <!-- 配置数据源的环境，可以有一个或多个 -->  <environment id=*"development"*>  <!--  配置事物管理方式  type：  JDBC（推荐） --- 通过JDBC进行事物管理。他依赖于数据源  MANAGED --- 通过容器来管理事物的声明周期，而容器默认情况下会让他关闭  -->  <transactionManager type=*"JDBC"*></transactionManager>    <!--  配置数据源  type：  UNPOOLED --- 不适用连接池，适用于较小的项目。效率稍低  POOLED --- 使用连接池，推荐使用  -->  <dataSource type=*"POOLED"*>  <property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>  <property name=*"url"* value=*"jdbc:mysql://localhost:3306/studentdb"*/>  <property name=*"username"* value=*"root"*/>  <property name=*"password"* value=*"123"*/>  </dataSource>  </environment>  </environments>    <!-- 引入相关的映射文件 -->  <mappers>    </mappers>  </configuration> |

1. 创建VO类

|  |
| --- |
| **public** **class** ClassRoom {  **private** **int** roomId;  **private** String roomName;  **private** String description;  **private** **int** state;  ......  } |

1. 编写映射文件并引入到配置文件中

|  |
| --- |
| [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=*"cn.hl.vo.ClassRoom"*>  <!--  id ：映射的唯一标识符，一般通过Id对映射进行调用  resultType ：返回值的类型  -->  <select id=*"getAll"* resultType=*"cn.hl.vo.ClassRoom"*>  select \* from classroom  </select>  </mapper> |
| <!-- 引入相关的映射文件 -->  <mappers>  <!--  resource ：配置映射文件的位置（使用映射文件的相对物理路径）  -->  <mapper resource=*"cn/hl/vo/ClassRoomMapper.xml"*/>  </mappers> |

注：映射文件一般与vo类在同一目录下，也可以独立存放

1. 调用映射

* 加载配置文件
* 获取Factory对象
* 获取Session对象
* 执行操作
* 关闭连接(将连接交还给连接池)

|  |
| --- |
| /\*\*  \* 执行查询功能  \* **@throws** IOException  \*/  @Test  **public** **void** test1() **throws** IOException{  //1、加载配置文件  Reader reader = Resources.*getResourceAsReader*("conf.xml");  //2、获取SQLSessionFactory（通过SQLSessionFactory可以获取到SqlSession）  SqlSessionFactory fac = **new** SqlSessionFactoryBuilder().build(reader);  //3、获取Session对象  SqlSession session = fac.openSession();  //4、执行查询功能  List<ClassRoom> list = session.selectList("cn.hl.vo.ClassRoom.getAll");  //5、关闭Session  session.close();    //6、处理数据  System.*out*.println(list);  } |

# 列名于属性名不一致

## 概述

实际编写代码时,常会出现列名和对象的属性名不一致的情况.MyBatis通过ResultMap提供了解决方案.

## 表结构

|  |
| --- |
|  |

## 实体类

|  |
| --- |
| **public** **class** User {  **private** **int** id;  **private** String account;  **private** String name;  **private** String password;  **private** **int** age;  **private** **boolean** gender;  ......  } |

## 方式一:列别名方式

|  |
| --- |
| <select id=*"getAll1"* resultType=*"cn.hl.vo.User"*>  <!-- 1、通过查询语句进行解决：使用列别名 -->  select userId as id, userName as name,account, pwd as password, age,gender from users  </select> |
| @Test  **public** **void** test2() **throws** IOException{  //1、加载配置文件  Reader reader = Resources.*getResourceAsReader*("conf.xml");  //2、获取SqlSessionFactory对象  SqlSessionFactory fac =**new** SqlSessionFactoryBuilder().build(reader);  //3、获取SqlSession对象  SqlSession session = fac.openSession();    List<User> list = session.selectList("cn.hl.vo.User.getAll1");  session.close();    System.*out*.println(list);    } |

## 方式二：ResultMap

|  |
| --- |
| <select id=*"getAll2"* resultMap=*"UserMap"*>  <!-- 2、通过resultMap进行配置 -->  select \* from users  </select>    <!--  配置返回数据的类型  type ：查询结果的返回值类型  id ：resultMap的Id  -->  <resultMap type=*"cn.hl.vo.User"* id=*"UserMap"*>  <!-- 一般用于主键列。column：用于配置表中列的名称；property：用于配置相匹配的对象的属性名 -->  <id column=*"userId"* property=*"id"*/>  <!-- 一般用于配置非主键列 -->  <result column=*"userName"* property=*"name"*/>  <result column=*"pwd"* property=*"password"*/>  </resultMap> |
| @Test  **public** **void** test3() **throws** IOException{  //1、加载配置文件  Reader reader = Resources.*getResourceAsReader*("conf.xml");  //2、获取SqlSessionFactory对象  SqlSessionFactory fac =**new** SqlSessionFactoryBuilder().build(reader);  //3、获取SqlSession对象  SqlSession session = fac.openSession();    List<User> list = session.selectList("cn.hl.vo.User.getAll2");  session.close();    System.*out*.println(list);    } |

# 常用对象

## SqlSessionFactoryBuilder

1. 作用

用于获取SqlSessionFactory。

1. 生命周期

创建SqlSessionFactory对象后即可销毁。所以最佳生命周期是方法内（一般定义为局部变量）。

## SqlSessionFactory

1. 作用

用于获取SqlSession对象。

1. 生命周期

应用执行期间都存在，一般建议注入到容器中。

## SqlSession

1. 作用

用于执行Sql映射。

1. 生命周期

非线程安全的，建议定义在方法内。

# CRUD

## 工具类

|  |
| --- |
| /\*\*  \* 工具类，提供用于获取SqlSessionFactory对象的方法  \* **@author** Terry  \*  \*/  **public** **class** FactoryBuilder {  //存储资源文件的位置  **private** **static** String *res* = "conf.xml";    //声明SQLSessionFactory对象  **private** **static** SqlSessionFactory *fac* = **null**;    /\*\*  \* 获取SqlsessionFactory对象  \* 单例方式获取SqlSessionFactory对象  \* **@return**  \* **@throws** IOException  \*/  **public** **static** SqlSessionFactory build() **throws** IOException{  //判断fac对象是否为null  **if**(*fac* == **null**){  //线程锁  **synchronized** (FactoryBuilder.**class**) {  //读取资源文件  Reader reader = Resources.*getResourceAsReader*(*res*);  //获取SqlSessionFactory对象  *fac* = **new** SqlSessionFactoryBuilder().build(reader);  }  }    **return** *fac*;  }  } |
| **public** **class** SessionFactory {  /\*\*  \* 创建一个SQLSession对象  \* **@return**  \* **@throws** IOException  \*/  **public** **static** SqlSession create() **throws** IOException{  //默认情况下是开启事物的，需要调用commit()方法对所执行操作进行提交  //false ：不进行自动提交；true ：执行自动提交  **return** FactoryBuilder.*build*().openSession(**true**);  }    /\*\*  \* 创建一个SqlSession对象  \* **@param** mode ：true--开启自动提交模式；false--关闭自动提交模式  \* **@return**  \* **@throws** IOException  \*/  **public** **static** SqlSession create(**boolean** mode) **throws** IOException{  //默认情况下是开启事物的，需要调用commit()方法对所执行操作进行提交  //false ：不进行自动提交；true ：执行自动提交  **return** FactoryBuilder.*build*().openSession(mode);  }  } |

## C

|  |
| --- |
| <!-- 添加数据 -->  <insert id=*"add"* parameterType=*"cn.hl.vo.ClassRoom"*>  insert into classroom  values(default,#{roomName},#{description},#{state})  </insert> |
| /\*\*  \* 执行添加操作  \*/  @Test  **public** **void** add(){  //声明Session对象  SqlSession session=**null**;  ClassRoom cr = **new** ClassRoom(0,"UUUUU","uuuu",0);    **try** {  //通过工具类SessionFactory构建session对象  session = SessionFactory.*create*();  //执行添加映射  **int** result = session.insert("cn.hl.vo.ClassRoom.add", cr);  //手动调用commit()方法对操作进行提交  //session.commit();  System.*out*.println(result);  } **catch** (IOException e) {  e.printStackTrace();  }  **finally**{  session.close();  }  } |

## U

|  |
| --- |
| <!-- 修改数据 -->  <update id=*"edit"* parameterType=*"cn.hl.vo.ClassRoom"*>  update classroom set roomName=#{roomName},description=#{description},  state=#{state} where roomId=#{roomId}  </update> |
| /\*\*  \* 执行修改操作  \*/  @Test  **public** **void** update(){  //声明Session对象  SqlSession session = **null**;  ClassRoom cr = **new** ClassRoom(21,"OOOO","oooo",1);    **try** {  //通过工具类sessionFactory构建session对象  session = SessionFactory.*create*();  //执行映射  **int** result = session.update("cn.hl.vo.ClassRoom.edit", cr);  System.*out*.println(result);  } **catch** (IOException e) {  e.printStackTrace();  }  **finally**{  session.close();  }  } |

## D

|  |
| --- |
| <!-- 删除数据 -->  <delete id=*"del"* parameterType=*"cn.hl.vo.ClassRoom"*>  delete from classroom where roomId=#{roomId}  </delete> |
| /\*\*  \* 执行删除操作  \*/  @Test  **public** **void** delete(){  //声明Session对象  SqlSession session = **null**;    **try** {  //通过工具类sessionFactory构建session对象  session = SessionFactory.*create*();  //执行映射  **int** result= session.delete("cn.hl.vo.ClassRoom.del", 20);  System.*out*.println(result);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |

## R

|  |
| --- |
| <!-- 按照Id查询数据 -->  <select id=*"getById"* parameterType=*"int"* resultType=*"cn.hl.vo.ClassRoom"*>  select \* from classroom where roomId=#{id}  </select>    <!--  id ：映射的唯一标识符，一般通过Id对映射进行调用  resultType ：返回值的类型  -->  <select id=*"getAll"* resultType=*"cn.hl.vo.ClassRoom"*>  select \* from classroom  </select> |
| /\*\*  \* 按Id执行查询语句  \*/  @Test  **public** **void** selectOne(){  //声明Session对象  SqlSession session = **null**;    **try** {  //通过工具类sessionFactory构建session对象  session = SessionFactory.*create*();  //执行映射  ClassRoom cr = session.selectOne("cn.hl.vo.ClassRoom.getById", 210);  System.*out*.println(cr);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  }    /\*\*  \* 按所有数据集  \*/  @Test  **public** **void** selectAll(){  //声明Session对象  SqlSession session = **null**;    **try** {  //通过工具类sessionFactory构建session对象  session = SessionFactory.*create*();  //执行映射  List<ClassRoom> list = session.selectList("cn.hl.vo.ClassRoom.getAll");  System.*out*.println(list);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |

# 注解

## 步骤

1. 创建vo类

|  |
| --- |
| **public** **class** City {  **private** **int** cityId;  **private** String cityName;  **private** **int** pId;  } |

1. 编写映射器

|  |
| --- |
| **public** **interface** CityMapper {  /\*\*  \* 添加数据到数据库  \* **@param** city  \* **@return**  \*/  @Insert("insert into city values(default,#{cityName},#{pId})")  **public** **int** add(City city);    /\*\*  \* 按Id修改数据  \* **@param** city  \* **@return**  \*/  @Update("update city set cityName=#{cityName},pid=#{pId} where cityId=#{cityId}")  **public** **int** update(City city);    /\*\*  \* 按Id删除数据  \* **@param** id  \* **@return**  \*/  @Delete("delete from city where cityId=#{id}")  **public** **int** delete(**int** id);    /\*\*  \* 按Id查询数据  \* **@param** id  \* **@return**  \*/  @Select("select \* from city where cityId=#{id}")  **public** City getById(**int** id);    /\*\*  \* 查询所有数据  \* **@return**  \*/  @Select("select \* from city")  **public** List<City> getAll();  } |

1. 注册接口

|  |
| --- |
| <mappers>  <!-- 注册接口 -->  <mapper class=*"cn.hl.mapper.CityMapper"*/>  </mappers> |

1. 测试

|  |
| --- |
| /\*\*  \* 添加城市信息到数据库  \*/  @Test  **public** **void** add(){  //声明SqlSession对象  SqlSession session = **null**;    **try** {  //获取SqlSession对象  session = SessionFactory.*create*();  //获取映射器（接口实例）  CityMapper mapper = session.getMapper(CityMapper.**class**);  //调用映射器（接口）的add()方法  **int** result = mapper.add(**new** City(0,"西宁",0));  System.*out*.println("result = " + result);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |

## 配置文件与注解

* 配置文件相对较复杂；注解简单方便
* 配置文件可以实现较复杂的业务；注解一般是用于简单的业务

# 应用优化

## 数据库连接配置

### 概述

数据库连接定义在配置文件中的缺点：

* 数据库连接配置定义在核心配置文件中，相对比较混乱
* 当配置项较多时，不便于修改

### properties配置文件

1. properties配置文件

|  |
| --- |
| driver = com.mysql.jdbc.Driver  url = jdbc:mysql://localhost:3306/studentdb  username = root  pwd =123 |

1. Mybatis核心配置文件

|  |
| --- |
| <configuration>  <!-- 配置properties配置文件的相关信息 -->  <properties resource=*"connection.properties"*></properties>  <environments default=*"dev"*>  <environment id=*"dev"*>  <transactionManager type=*"JDBC"*></transactionManager>    <dataSource type=*"POOLED"*>  <!-- 引入配置文件中的相关配置 -->  <property name=*"driver"* value=*"${driver}"*/>  <property name=*"url"* value=*"${url}"*/>  <property name=*"username"* value=*"${username}"*/>  <property name=*"password"* value=*"${pwd}"*/>  </dataSource>  </environment>  </environments>    <mappers>  <!-- 注册接口 -->  <mapper class=*"cn.hl.mapper.CityMapper"*/>  </mappers>  </configuration> |

### 注意事项

* resource的引入需要定义在environments
* 使用properties的属性时注意大小写

### 优点

MyBatis的配置和数据库连接配置相分离，使得数据库连接的配置修改更加简单

## 类型别名

### 概述

默认情况下，映射文件中应用实体类名称时，需要使用类的全名（包名+类名）。这种方式导致代码的编写较为繁琐。

### 类型别名

|  |
| --- |
| [vo类]  **public** **class** City {  **private** **int** cityId;  **private** String cityName;  **private** **int** pId;  } |
| [核心配置文件]  <configuration>  <!-- 配置properties配置文件的相关信息 -->  <properties resource=*"connection.properties"*></properties>  <!-- 配置类型别名 -->  <typeAliases>  <!--  type ：配置类型的全名（包名 + 类名）  alias ：配置类型的别名。名称自定义（建议要求符合命名规范）  -->  <typeAlias type=*"cn.hl.vo.City"* alias=*"City"*/>  </typeAliases>  <environments default=*"dev"*>  <environment id=*"dev"*>  <transactionManager type=*"JDBC"*></transactionManager>    <dataSource type=*"POOLED"*>  <!-- 引入配置文件中的相关配置 -->  <property name=*"driver"* value=*"${driver}"*/>  <property name=*"url"* value=*"${url}"*/>  <property name=*"username"* value=*"${username}"*/>  <property name=*"password"* value=*"${pwd}"*/>  </dataSource>  </environment>  </environments>    <mappers>  <mapper resource=*"cn/hl/vo/CityMapper.xml"*/>  <!-- 注册接口 -->  <!--  <mapper class="cn.hl.mapper.CityMapper"/>  -->  </mappers>  </configuration> |
| [映射文件]  <mapper namespace=*"cn.hl.vo.City"*>  <insert id=*"add"* parameterType=*"City"*>  insert into city values(default, #{cityName},#{pId})  </insert>    <select id=*"getById"* resultType=*"City"*>  select \* from city where cityId=#{cityId}  </select>  </mapper> |

### 优点

使编码变得更加简洁

### 注意

* typeAliases的配置在environments的前面
* 别名的命名尽量符合标准的命名规范

# 多表关联查询

## 概述

在实际编码中，经常出现多表查询的情况。其中主要包含：一对一、一对多、多对一、多对多。

## 一对一关联查询

### 表结构

表名：公民信息表(Person)

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 类型 | 特征 | 描述 |
| PersonId | Int | 自增、主键 | 记录Id |
| Name | Varchar(20) | 非空 | 姓名 |
| CardId | Varchar(18) | 非空 | 身份证号码 |

表名：身份证信息表(IdCard)

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 类型 | 特征 | 描述 |
| CardId | Varchar（18） | 主键、唯一 | 身份证号码 |
| Address | Varchar（50） | 非空 | 家庭住址 |

### 关系

公民表(Person) 身份证表(IdCard)

|  |  |  |
| --- | --- | --- |
| PersonId | Name | CardId |
| 1 | Mike | 1001 |
| 2 | Jack | 1002 |
| 3 | Tom | 2001 |

|  |  |
| --- | --- |
| CardId | Address |
| 1001 | 北京 |
| 1002 | 北京 |
| 2001 | 上海 |

### 查询语句

1. 多表连接查询

|  |
| --- |
| select \*  from Person a  join IdCard b on a.cardId=b.cardId |

1. 多表独立查询

|  |
| --- |
| select \* from Person where personid=2;  select \* from IdCard where cardId=1002; |

### 实现

1. 方式一：多表连接查询（嵌套结果）

|  |
| --- |
| [vo--IdCard]  **public** **class** IdCard {  **private** String cardId;  **private** String address;  } |
| [vo--Person]  **public** **class** Person {  **private** **int** personId;  **private** String name;  **private** IdCard cardId;  } |
| [映射文件]  <mapper namespace=*"cn.hl.vo.Person"*>  <!--  按Id执行多表连接查询  select \* from Person a join IdCard b on a.cardId=b.cardId  where PersonId=2  -->  <select id=*"getById"* parameterType=*"int"* resultMap=*"personMap"*>  select \* from Person a join IdCard b on a.cardId=b.cardId  where PersonId=#{id}  </select>    <resultMap type=*"cn.hl.vo.Person"* id=*"personMap"*>  <id column=*"personId"* property=*"personId"*/>  <result column=*"name"* property=*"name"*/>  <!--  联合关系  property ：属性名，一般为表中外键的名称（同名）  javaType ：property属性所对应的Java类的名称  -->  <association property=*"cardId"* javaType=*"cn.hl.vo.IdCard"*>  <id column=*"cardId"* property=*"cardId"*/>  <result column=*"address"* property=*"address"*/>  </association>  </resultMap>  </mapper> |
| [测试代码]  /\*\*  \* 执行多表连接查询：一对一方式  \*  \*/  @Test  **public** **void** getById1(){  //声明Session对象  SqlSession session = **null**;      **try** {  //获取Session对象  session = SessionFactory.*create*();  Person p = session.selectOne("cn.hl.vo.Person.getById", 2);    System.*out*.println(p);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |

1. 方式二：独立查询语句（嵌套查询）

|  |
| --- |
| [vo--IdCard]  **public** **class** IdCard {  **private** String cardId;  **private** String address;  } |
| [vo--Person]  **public** **class** Person {  **private** **int** personId;  **private** String name;  **private** IdCard cardId;  } |
| [Mapper映射]  <!--  方式二：独立查询语句进行查询  select \* from Person where personid=2;  select \* from IdCard where cardId=1002;  -->  <select id=*"getById2"* parameterType=*"int"* resultMap=*"personMap2"*>  select \* from Person where personid=#{id}  </select>  <!-- 查询身份证信息 -->  <select id=*"getIdCard"* parameterType=*"int"* resultType=*"cn.hl.vo.IdCard"*>  select \* from IdCard where cardId=#{cardId}  </select>    <resultMap type=*"cn.hl.vo.Person"* id=*"personMap2"*>  <id column=*"personId"* property=*"personId"*/>  <result column=*"name"* property=*"name"*/>  <!--  property ：实体类中属性的名称  column ：表中列的名称  javaType ：相关联的的属性的类型（与Property相对应）  select ：需要调用的select映射的Id  -->  <association property=*"cardId"* column=*"cardId"* javaType=*"cn.hl.vo.IdCard"* select=*"getIdCard"*>  </association>  </resultMap> |
| /\*\*  \* 执行多表连接查询：一对一方式  \*  \*/  @Test  **public** **void** getById2(){  //声明Session对象  SqlSession session = **null**;      **try** {  //获取Session对象  session = SessionFactory.*create*();  Person p = session.selectOne("cn.hl.vo.Person.getById2", 2);    System.*out*.println(p);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |

## 一对多关联查询

### 表结构

表名：公民信息表(Person)

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 类型 | 特征 | 描述 |
| PersonId | Int | 自增、主键 | 记录Id |
| Name | Varchar(20) | 非空 | 姓名 |

表名：账户信息表(Card)

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 类型 | 特征 | 描述 |
| CardNo | Varchar(19) | 主键、唯一 | 银行卡号 |
| Blance | Float | 非空 | 余额 |
| PersonId | Int | 非空 | 用户Id |

### 关系

公民信息 账户信息

|  |  |
| --- | --- |
| Name | PersonId |
| Mike | 1 |
| Jack | 2 |

|  |  |  |
| --- | --- | --- |
| PersonId | Blance | CardNo |
| 1 | 199 | 6220001 |
| 2 | 1000000 | 6220090 |
| 1 | 90 | 6220002 |
| 1 | 18 | 6220003 |

### 查询语句

|  |
| --- |
| [连接查询]  select \* from person a  left join Card b  on a.personId=b.PersonId where a.personId = 1 |
| [独立语句查询]  select \* from person where personId=1;  select \* from card where personId=1 |

### 实现

1. 方式一

|  |
| --- |
| [vo---Card]  **public** **class** Card {  **private** String cardNo;  **private** **float** blance;  } |
| [vo---person2]  **public** **class** Person2 {  **private** **int** personId;  **private** String name;  **private** List<Card> cards;  } |
| [映射]  <!--  方式一：按Id执行多表连接查询  select \* from person a  left join Card b  on a.personId=b.PersonId where a.personId = 1  -->  <select id=*"getById1"* parameterType=*"int"* resultMap=*"PersonMap1"*>  select \* from person a  left join Card b  on a.personId=b.PersonId where a.personId = #{id}  </select>    <!-- 定义返回结果集 -->  <resultMap type=*"cn.hl.vo.Person2"* id=*"PersonMap1"*>  <id property=*"personId"* column=*"personId"*/>  <result property=*"name"* column=*"name"*/>  <!-- 集合 -->  <collection property=*"cards"* ofType=*"cn.hl.vo.Card"*>  <id property=*"cardNo"* column=*"cardNo"*/>  <result property=*"blance"* column=*"blance"*/>  </collection>  </resultMap> |
| [测试]  /\*\*  \* 测试方式一  \*/  @Test  **public** **void** test1(){  //声明SqlSession  SqlSession session = **null**;    **try**{  //获取Session对象  session = SessionFactory.*create*();  //调用映射  Person2 p = session.selectOne("cn.hl.vo.Person2.getById1", 1);  System.*out*.println(p);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  //关闭Session  session.close();  }  } |

1. 方式二

|  |
| --- |
| <!--  方式二：使用独立的查询语句进行实现  select \* from person where personId=1;  select \* from card where personId=1  -->  <select id=*"getById2"* parameterType=*"int"* resultMap=*"PersonMap2"*>  select \* from person where personId=#{id}  </select>    <!-- 查询银行卡信息 -->  <select id=*"getCard"* parameterType=*"int"* resultType=*"cn.hl.vo.Card"*>  select \* from card where personId=#{id}  </select>    <!-- 配置返回类型 -->  <resultMap type=*"cn.hl.vo.Person2"* id=*"PersonMap2"*>  <id property=*"personId"* column=*"personId"*/>  <result property=*"name"* column=*"name"*/>    <collection property=*"cards"* column=*"personId"* select=*"getCard"*></collection>  </resultMap> |
| [测试]  /\*\*  \* 测试方式二  \*/  @Test  **public** **void** test2(){  //声明SqlSession  SqlSession session = **null**;    **try**{  //获取Session对象  session = SessionFactory.*create*();  //调用映射  Person2 p = session.selectOne("cn.hl.vo.Person2.getById2", 1);  System.*out*.println(p);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  //关闭Session  session.close();  }  } |

# 模糊查询

## 区间查询

### 需求

查询用户表中所有Id在2~10范围内的所有信息。

### Sql语句

|  |
| --- |
| select \* from users where userId>=2 and UserId<=10; |
| select \* from users where userId between 2 and 10; |

### 实现

|  |
| --- |
| [vo类]  **public** **class** User {  **private** **int** userId;  **private** String account;  **private** String pwd;  **private** String userName;  **private** **int** age;  **private** **boolean** gender;  } |
| [映射配置]  <mapper namespace=*"cn.hl.vo.User"*>  <!-- 区间查询：查询所有Id在2~10之间的所有用户信息 -->  <select id=*"testBetween"* parameterType=*"int"* resultType=*"cn.hl.vo.User"*>  select \* from users where userId between #{minId} and #{maxId};  </select>  </mapper> |
| [vo~Exp]  **public** **class** Exp {  **private** **int** minId;  **private** **int** maxId;  } |
| [测试]  /\*\*  \* 执行区间模糊查询  \*/  @Test  **public** **void** testBetween(){  //声明SqlSession对象  SqlSession session = **null**;  Exp exp = **new** Exp(2,10);    **try** {  //获取SQLSession对象  session = SessionFactory.*create*();  //调用映射  List<User> list = session.selectList("cn.hl.vo.User.testBetween", exp);  //打印查询结果  System.*out*.println(list);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  } |

## like模糊查询

### 需求变更

查询用户表中所有Id在2~10范围内且帐号中包含字母“o”的所有信息

### Sql语句

|  |
| --- |
| select \* from users where userId>=2 and UserId<=10 and account like '%o%'; |
| select \* from users where userId between 2 and 10 and account like '%o%'; |

### 实现

|  |
| --- |
| [vo~User]  **public** **class** User {  **private** **int** userId;  **private** String account;  **private** String pwd;  **private** String userName;  **private** **int** age;  **private** **boolean** gender;  } |
| [映射]  <!-- Like模糊查询：查询用户表中所有Id在2~10范围内且帐号中包含字母“o”的所有信息 -->  <select id=*"testLike"* parameterType=*"cn.hl.vo.Exp"* resultType=*"cn.hl.vo.User"*>  select \* from users  where userId between #{minId} and #{maxId} and account like #{account};  </select> |
| [vo~Exp]  **public** **class** Exp {  **private** **int** minId;  **private** **int** maxId;  **private** String account;  } |
| [测试]  /\*\*  \* 执行Like模糊查询  \*/  @Test  **public** **void** testLike(){  //声明SqlSession对象  SqlSession session = **null**;  Exp exp = **new** Exp(2,10,"%o%");    **try** {  //获取SQLSession对象  session = SessionFactory.*create*();  //调用映射  List<User> list = session.selectList("cn.hl.vo.User.testLike", exp);  //打印查询结果  System.*out*.println(list);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  } |

## 动态Sql语句

### 需求变更

查询用户表中所有Id在2~10范围内且帐号中包含字母“o”的所有信息。如果未传查询的帐号内容，则省略该条件

### Sql语句

|  |
| --- |
| [未省略account]  select \* from users where userId between 2 and 10 and account like '%o%'; |
| [省略account]  select \* from users where userId between 2 and 10; |

### 实现

|  |
| --- |
| [映射]  <select id=*"testIf"* parameterType=*"cn.hl.vo.Exp"* resultType=*"cn.hl.vo.User"*>  select \* from users where userId between #{minId} and #{maxId}  <!-- if条件 -->  <if test=*"account != null"*>  and account like #{account}  </if>  </select> |
| [测试]  /\*\*  \* 执行Like模糊查询  \*/  @Test  **public** **void** testIf(){  //声明SqlSession对象  SqlSession session = **null**;  //Exp exp = new Exp(2,10,"%o%");  Exp exp = **new** Exp(2,10);    **try** {  //获取SQLSession对象  session = SessionFactory.*create*();  //调用映射  List<User> list = session.selectList("cn.hl.vo.User.testIf", exp);  //打印查询结果  System.*out*.println(list);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  } |

# Log4j

## 概述

Log4j = Log For Java

通过log4j可以将日志输出到任何目的地址，包含：控制台、文件、数据库等。log4j可以通过配置文件灵活的配置日志的输出方式。不需要编写任何代码。

## 步骤

1. 导入Jar包

|  |
| --- |
|  |

1. 配置log4j(src/log4j.properties)

|  |
| --- |
| ### defined root logger ###  ### log4j.rootLogger = [level:DEBUG/FATAL/ERROR/WARN/INFO/ALL],appenderName1,appenderName2.....appenderNameN  log4j.rootLogger = info,stdout  ### config the appender ###  ### log4j.appender.appenderName[.xxx] = value  ### config the out method for appender(ConsoleAppender/FileAppender/DailyRollingFileAppender/RollingFileAppender/WriterAppender)  ### print message to the console  log4j.appender.stdout = org.apache.log4j.ConsoleAppender  log4j.appender.stdout.Target = System.out  ### config the appender's layout(HTMLLayout/PatternLayout/SimpleLayout/TTCCLayout)  log4j.appender.stdout.layout = org.apache.log4j.PatternLayout  ###Pattern(%P:level/%R:times/%C:name of class/%t : thread name / %n:enter % )  log4j.appender.stdout.layout.ConversionPattern = [%-5p] method:%l%n%m%n |

3)在程序代码中使用日志

|  |
| --- |
| **public** **static** **void** main(String[] args) {  //获取Logger对象  Logger log = Logger.*getLogger*(TestLog.**class**);  //记录log信息  log.info("Log4j .... info");  log.debug("log4j .... debug");  log.warn("log4j .... warn");  log.error("log4j .... error");  } |

注意：log4j只记录指定级别或更高级别的消息内容。

# 调用存储过程

## 概述

存储是为了完成特定功能的一段预编译Sql语句，存储过程通过名称进行调用。

## Sql语句与存储过程

### Sql语句

* 安全性较低
* 执行效率较低
* 复用性较差

### 存储过程

* 安全性较高
* 执行效率较高
* 具有较高的复用性

## 语法

|  |
| --- |
| CREATE PROCEDURE `名称`([[In|Out|InOut] 参数名 参数类型])  BEGIN  代码段;  END;; |

## 示例

|  |
| --- |
| [1、无参数无代码段的存储过程]  create procedure proc\_1()  begin  end; |
| [2、无参数有代码段的存储过程]  create procedure proc\_2()  begin  select \* from users;  end; |
| [3、带参数存储过程]  Create procedure proc\_3 (id int)  begin  select \* from users where userId=id;  end |
| [4、带输入、输出参数的存储过程]  /\*  in :输入参数  out ：输出参数  inout ：输入输出参数  \*/  create procedure proc\_4(in account varchar(20),out total int)  begin  select count(\*) into total from users where account = account;  end; |
| [5、带有变量的存储过程]  CREATE PROCEDURE proc\_5 (in acc varchar(20),out total int)  begin  declare i int ;  select count(\*) into i from users where account = acc;  if(i>0) then  set total = 1;  else  set total = -1;  end if;  end |

## 调用存储过程

|  |
| --- |
| Call 存储过程名([参数列表]) |

## MyBatis调用存储过程

### 存储过程

|  |
| --- |
| CREATE PROCEDURE `proc\_6`(in id int,out total int)  begin  select count(\*) into total from users where userId=id;  end |

### 步骤

* 配置映射

|  |
| --- |
| <mapper namespace=*"cn.hl.vo.User"*>  <!--  调用存储过程  -->  <select id=*"callProc"*>  set @total=0;  call proc\_5('admin',@total);  select @total;  </select>  </mapper> |

* 调用映射

|  |
| --- |
| **public** **static** **void** main(String[] args) {  //声明SqlSession  SqlSession session = **null**;    //创建Map对象进行参数传递  Map<String,Integer> map = **new** HashMap<String,Integer>();  //将参数添加到Map集合中  map.put("id", 100);  map.put("total", -1);      **try** {  //获取Session  session = SessionFactory.*create*();  session.selectOne("cn.hl.vo.User.callProc", map);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }    //输出返回结果  System.*out*.println(map.get("total"));  } |

* 完善映射

|  |
| --- |
| <mapper namespace=*"cn.hl.vo.User"*>  <!--  调用存储过程  -->  <select id=*"callProc"* parameterMap=*"paramMap"* statementType=*"CALLABLE"*>  <!-- 参数位置用占位符进行替代 -->  call proc\_6(?,?);  </select>    <parameterMap type=*"java.util.Map"* id=*"paramMap"*>  <!--  property ：map中key的名称  INTEGER ：必须全部大写  mode ：设置参数的方向（in、out、inout）  -->  <parameter property=*"id"* jdbcType=*"INTEGER"* mode=*"IN"*/>  <parameter property=*"total"* jdbcType=*"INTEGER"* mode=*"OUT"*/>  </parameterMap>  </mapper> |

### 注意事项

* Map中压入元素的顺序与存储过程的参数顺序相一致
* integer类型必须全部大写---INTEGER
* select配置的statementType属性值必须修改为callable

# 缓存

## 概述

MyBatis支持缓存机制，主要提供4中缓存：一级缓存、二级缓存、自定义缓存和参照缓存。其中一级缓存和二级缓存是以PurpetualCatch、HashMap进行实现的。

## 作用

当执行查询语句时，为了避免同样语句的重复提交，造成效率的下降。所以提供了缓存机制。

通过缓存机制可以避免同一条语句的重复执行下率下降。

## 一级缓存

### 概述

一级缓存也称为Session缓存。默认情况下一级缓存是开启的。

### 特点

1. 同一条语句只被执行一次（在此执行将通过缓存进行执行）

|  |
| --- |
| [Mapper]  <mapper namespace=*"cn.hl.vo.User"*>  <select id=*"getById"* parameterType=*"int"* resultType=*"cn.hl.vo.Users"*>  select \* from users where userId=#{id}  </select>  </mapper> |
| [测试]  /\*\*  \* 测试一级缓存  \* 1、同一条语句只执行一次  \* 以后执行从缓存直接读取数据（第二次就是直接读取）  \*/  @Test  **public** **void** testCache1(){  String mapper = "cn.hl.vo.User.getById";  //声明SqlSession对象  SqlSession session = **null**;      **try** {  //获取Session对象  session = SessionFactory.*create*();  Users user1 = session.selectOne(mapper, 1);  System.*out*.println(user1);    System.*out*.println("\n==============================================\n");    //Sql语句没有被发送给数据库  Users user2 = session.selectOne(mapper, 1);  System.*out*.println(user2);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |
| [效果] |

1. 执行CUD操作后缓存将被清空

|  |
| --- |
| [Mapper映射]  <mapper namespace=*"cn.hl.vo.User"*>  <select id=*"getById"* parameterType=*"int"* resultType=*"cn.hl.vo.Users"*>  select \* from users where userId=#{id}  </select>    <update id=*"update"*>  update Users set pwd='123' where UserId=5  </update>  </mapper> |
| [测试]  /\*\*  \* 测试一级缓存  \* 2、执行CUD操作后缓存将被清空  \*/  @Test  **public** **void** testCache2(){  String mapper = "cn.hl.vo.User.getById";  //声明SqlSession对象  SqlSession session = **null**;      **try** {  //获取Session对象  session = SessionFactory.*create*();  Users user1 = session.selectOne(mapper, 1);  System.*out*.println(user1);    System.*out*.println("\n==============================================\n");  session.update("cn.hl.vo.User.update");  System.*out*.println("\n==============================================\n");    //Sql语句没有被发送给数据库  Users user2 = session.selectOne(mapper, 1);  System.*out*.println(user2);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |
| [效果] |

1. 不同Session无法共享缓存

|  |
| --- |
| [Mapper]  <select id=*"getById"* parameterType=*"int"* resultType=*"cn.hl.vo.Users"*>  select \* from users where userId=#{id}  </select> |
| [测试]  /\*\*  \* 测试一级缓存  \* 3、不同的Session对象无法共享缓存  \*/  @Test  **public** **void** testCache3(){  String mapper = "cn.hl.vo.User.getById";  //声明SqlSession对象  SqlSession session1 = **null**;  SqlSession session2 = **null**;      **try** {  //获取Session对象  session1 = SessionFactory.*create*();  Users user1 = session1.selectOne(mapper, 1);  System.*out*.println(user1);    System.*out*.println("\n==============================================\n");    //Sql语句没有被发送给数据库  session2 = SessionFactory.*create*();  Users user2 = session2.selectOne(mapper, 1);  System.*out*.println(user2);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session1.close();  session2.close();  }  } |
| [效果] |

1. 语句发生改变时缓存将失效

|  |
| --- |
| [Mapper]  <select id=*"getById"* parameterType=*"int"* resultType=*"cn.hl.vo.Users"*>  select \* from users where userId=#{id}  </select> |
| [测试]  /\*\*  \* 测试一级缓存  \* 4、语句发生改变时缓存将失效  \*  \*/  @Test  **public** **void** testCache4(){  String mapper = "cn.hl.vo.User.getById";  //声明SqlSession对象  SqlSession session = **null**;      **try** {  //获取Session对象  session = SessionFactory.*create*();  Users user1 = session.selectOne(mapper, 1);  System.*out*.println(user1);    System.*out*.println("\n==============================================\n");    //Sql语句没有被发送给数据库  Users user2 = session.selectOne(mapper, 2);  System.*out*.println(user2);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |
| [效果] |

1. 调用clearCache()后缓存将实效

|  |
| --- |
| [Mapper]  <select id=*"getById"* parameterType=*"int"* resultType=*"cn.hl.vo.Users"*>  select \* from users where userId=#{id}  </select> |
| [测试]  /\*\*  \* 测试一级缓存  \* 5、调用clearCache()后缓存将失效  \*  \*/  @Test  **public** **void** testCache5(){  String mapper = "cn.hl.vo.User.getById";  //声明SqlSession对象  SqlSession session = **null**;      **try** {  //获取Session对象  session = SessionFactory.*create*();  Users user1 = session.selectOne(mapper, 1);  System.*out*.println(user1);    System.*out*.println("\n==============================================\n");  //清除缓存  session.clearCache();  System.*out*.println("\n==============================================\n");    //Sql语句没有被发送给数据库  Users user2 = session.selectOne(mapper, 1);  System.*out*.println(user2);  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **finally**{  session.close();  }  } |
| [效果] |

## 二级缓存

### 概述

二级缓存属于名命空间级缓存。

### 步骤

1. 编写Vo类

|  |
| --- |
| **public** **class** City **implements** Serializable {  **private** **int** cityId;  **private** String cityName;  **private** **int** pId;  } |

1. 在核心配置文件中启用缓存

|  |
| --- |
| <settings>  <!-- 设置：启用缓存 -->  <setting name=*"cacheEnabled"* value=*"true"*/>  </settings> |

1. Mapper中使用缓存

|  |
| --- |
| <mapper namespace=*"cn.hl.vo.City"*>  <!-- 映射文件中需要加入cache -->  <cache eviction=*"FIFO"* flushInterval=*"60000"* size=*"512"* readOnly=*"true"*/>  <select id=*"getById"* parameterType=*"int"* resultType=*"cn.hl.vo.City"*>  select \* from City where CityId=#{id}  </select>  </mapper> |

# MyBatis优缺点

## 优点

* 与JDBC相比，减少了50%左右的代码
* MyBatis是最简单的持久化框架。小巧且简单易学
* Mybatis相对比较灵活。不会对程序和数据库强加任何影响。Sql语句定义在Xml文件，从程序代码中彻底分离出来，降低了耦合度。便于统一管理和优化
* 提供Xml标签，支持动态Sql语句
* 提供Xml标签，支持对象与数据库的ORM关系映射

## 缺点

* Sql语句的编写工作两较大，尤其子多较多、关联表较多时更是如此。对于编写Sql语句的人员具有一定的要求
* Sql语句依赖数据库。导致数据库的移植性交差，不能随意更改数据库
* 二级缓存机制相对较弱，不够完善