1.1 MyBatis简介

ORM对象关系映射（Object Relational Mapping）

ORM把数据库映射成对象

数据库的表（table） --> 类（class）

记录（record，行数据）--> 对象（object）

字段（field）--> 对象的属性（attribute）

**集成MyBatis**

Spring Boot 2.2.7

Packaging jar

Java 11

注意事项：

1.添加Spring Web依赖

2.添加Mybatis Framework依赖

3.添加MySQL Driver依赖

如果在现有的SpringBoot工程里集成，可以直接在pom.xml里集成

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>2.1.2</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

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

<scope>runtime</scope>

</dependency>

**配置MySQL连接**

配置好MyBatis环境后，如果启动应该会提示报错，因为SpringBoot会自动加载数据库连接，所以这个时候需要配置一下datasource

启动错误日志

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

APPLICATION FAILED TO START

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:

Consider the following:

If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.

If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).

配置数据源

在工程的application.properties文件里，数据源配置 **格式** 为：

**spring.datasource.url**=jdbc:mysql://mysql数据库地址:数据库端口/数据库名称?serverTimezone=GMT%2B8

**spring.datasource.username**=用户名

**spring.datasource.password**=密码

（serverTimezone=GMT%2B8这个参数的作用在于设置数据库的时区为中国区域，这样查询出来的时间和我们平常的时间是一致的，要不然根据默认时区显示时间会相差8小时）

**1.配置前准备**

启动MySQL服务后，创建名为comment的数据库：

CREATE DATABASE comment；

**2.配置写法**

工程的application.properties文件里添加数据源的配置为：

**spring.datasource.url**=jdbc:mysql://192.168.0.1:3306/comment?serverTimezone=GMT%2B8

**spring.datasource.username**=root

**spring.datasource.password**=

(每位同学在优课达平台上启动的MySQL服务都是专属的，所以不需要密码)

1.3 MyBatis映射对象

**DO对象规则**

所有的ORM框架都需要有一个Java对象来映射数据库的表，并且是一一对应的，一般我们把这类对象称为**DO** 对象，对象名称的规范是 **表名+DO**

User表对应的类对象名称就是UserDO；comment表对应的类对象名称就是CommentDO

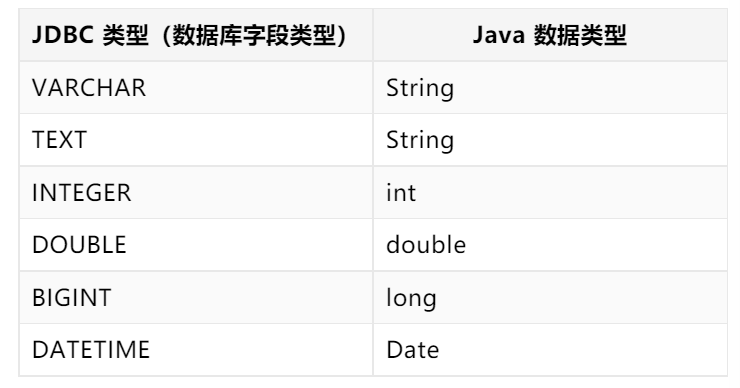
**DO对象包规则**

一般情况下企业中都会把这个DO对象存放在xxx.xxx.dataobject包下，所以DO就是dataobject的缩写。

现在评论组件的包约定是com.youkeda.comment，那么对应的DO包就是 com.youkeda.comment.dataobject

**DO对象数据类型**

DO对象和普通的POJO类并无不同，唯一要注意的点是属性类型要和数据库类型进行匹配，这个是JDBC指制定的规范，常用的数据类型映射如下



（注意默认情况下，数据库的日期类型对应的是java的java.util.Date（LocalDateTime）对象）

1.4MyBatis DAO

在java工程化中，我们一般会把数据层的服务称为DAO层，DAO层会包含对数据库操作的接口和实现类。

**创建DAO层**

以评论组件为例，我们先创建包

Com.youkeda.comment.dao

**定义DAO接口**

以user表为例，我们创建一个UserDAO这样的接口

**package** com.youkeda.comment.dao;

**import** org.apache.ibatis.annotations.Mapper;

**@Mapper**

**public** **interface** **UserDAO** {

}

这个接口的特殊性在于接口需要添加一个@Mapper的注解，当然别忘了import

**import** org.apache.ibatis.annotations.Mapper;

**引用DAO**

当我们完成MyBatis DAO的定义后，Spring启动的时候会自动加载这个接口并动态创建一个Spring Bean，所以我们只需要按照Spring Bean的方式完成资源注入

现在我们创建一个UserController用于处理用户的Web服务

**package** com.youkeda.comment.control;

**import** com.youkeda.comment.dao.UserDAO;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**@Controller**

**public** **class** **UserController** {

**@Autowired**

**private** UserDAO userDAO;

}

1.5 MyBatis查询

**1.添加接口方法**

**package** com.youkeda.comment.dao;

**import** org.apache.ibatis.annotations.Mapper;

**import** UserDO;

**import** java.util.List;

**@Mapper**

**public** **interface** **UserDAO** {

**public** List<UserDO> **findAll**();

}

FindAll方法：查询多条记录的时候，使用List作为返回类型；如果查询所有记录，方法名称一般定义为findAll

[**2.添加@Select**](mailto:2.添加@Select)**注解**

**package** com.youkeda.comment.dao;

**import** UserDO;

**import** org.apache.ibatis.annotations.Mapper;

**import** org.apache.ibatis.annotations.Select;

**import** java.util.List;

**@Mapper**

**public** **interface** **UserDAO** {

**@Select("SELECT id,user\_name as userName,pwd,nick\_name as nickName,avatar,gmt\_created as gmtCreated,gmt\_modified as gmtModified FROM user")**

List<UserDO> **findAll**();

}

**3.API测试**

现在DAO层完成开发了，我们可以在controller里进行测试一下服务，可以写一个API来完成测试

**@Controller**

**public** **class** **UserController** {

**@Autowired**

**private** UserDAO userDAO;

**@GetMapping("/users")**

**@ResponseBody**

**public** List<UserDO> **getAll**() {

**return** userDAO.findAll();

}

}

1.6MyBatis 插入

在MyBatis中支持插入，使用@Insert注解

**1.添加接口方法**

新增一个insert方法

**int** **insert**(UserDO userDO);

方法介绍：

执行SQL插入语句的时候，会返回插入函数，一般成功是返回1，所以设置返回类型为int

[**2.添加@Insert**](mailto:2.添加@Insert)**注解**

**@Mapper**

**public** **interface** **UserDAO** {

**@Insert("INSERT INTO user (user\_name, pwd, nick\_name,avatar,gmt\_created,gmt\_modified) VALUES(#{userName}, #{pwd}, #{nickName}, #{avatar},now(),now())")**

**int** **insert**(UserDO userDO);

**@Select("SELECT id,user\_name as userName,pwd,nick\_name as nickName,avatar,gmt\_created as gmtCreated,gmt\_modified as gmtModified FROM user")**

List<UserDO> **findAll**();

}

这里用到MyBatis获取动态值得一种方式，执行得时候会自动从上下文参数里获取这个变量名得值

VALUES(#{userName}, #{pwd}, #{nickName}, #{avatar},now(),now())

比如#{userName}实际上就是执行userDO.getUserName( )这个方式来获取userName变量值，MyBatis会自动更新生成正式的SQL语句去到数据库里执行，这就完成了动态数据存储

**3.API测试**

**@Controller**

**public** **class** **UserController** {

**@Autowired**

**private** UserDAO userDAO;

**@GetMapping("/users")**

**@ResponseBody**

**public** List<UserDO> **getAll**() {

**return** userDAO.findAll();

}

**@PostMapping("/user")**

**@ResponseBody**

**public** UserDO **save**(**@RequestBody** UserDO userDO) {

userDAO.insert(userDO);

**return** userDO;

}

}

**4.查看主键值**

上面输出的内容里并没有刚才插入的主键值，很多时候我们是需要知道这个值的。

我们可以通过在@Insert注解上继续添加一个@Options主键，

**@Insert("INSERT INTO user (user\_name, pwd, nick\_name,avatar,gmt\_created,gmt\_modified) VALUES(#{userName}, #{pwd}, #{nickName}, #{avatar},now(),now())")**

**@Options(useGeneratedKeys = true, keyColumn = "id", keyProperty = "id")**

**int** **insert**(UserDO userDO);

Options注解有三个参数

\*useGeneratedKeys 设置为true，代表运行数据库使用自增主键

\*keyColumn 设置表的主键字段名称，一般都是id

\*keyProperty 设置DO模型的主键字段，一般都是id

1.7 MyBatis 修改

**1.接口方法**

**int** **update**(UserDO userDO);

[**2.@Update**](mailto:2.@Update) **注解**

**@Update("update user set nick\_name=#{nickName},gmt\_modified=now() where id=#{id}")**

**3.API测试**

**@PostMapping("/user/update")**

**@ResponseBody**

**public** UserDO **update**(**@RequestBody** UserDO userDO) {

userDAO.update(userDO);

**return** userDO;

}

1.8 MyBatis 删除

**1.接口方法**

**int** **delete**(**long** id);

执行SQL删除语句时，会返回删除得行数，所以我们的方法返回类型设置为int

MyBatis除了支持DO对象传递参数之外，还可以接收普通参数，比如String、int、long之类得。为了能够在SQL语句中完成普通参数解析，需要对参数增加一个@Param，这样继续改进delete方法

**public** **interface** **UserDAO** {

**int** **delete**(**@Param("id")** **long** id);

}

[**2.@Delete**](mailto:2.@Delete) **注解**

**@Delete("delete from user where id=#{id}")**

（注意#{ id }中的id要和@Param（“id”）中的id一样的，如果我们定义的是@Param（“key”）那么SQL语句就是

**@Delete("delete from user where id=#{key}")**

）

完整代码如下：

**@Delete("delete from user where id=#{id}")**

**int** **delete**(**@Param("id")** **long** id);

**3.API测试**

**@GetMapping("/user/del")**

**@ResponseBody**

**public** **boolean** **delete**(**@RequestParam("id")** Long id) {

**return** userDAO.delete(id) > 0;

}

为了验证删除成功，使用

**return** userDAO.delete(id) > 0;

判断删除是否成功。

1.9 Mybatis 简单查询

**1.设计接口方法**

**public** **interface** **UserDAO** {

UserDO **findByUserName**(**@Param("userName")** String name);

}

[**2.@Select**](mailto:2.@Select)**注解**

**public** **interface** **UserDAO** {

**@Select("select id,user\_name as userName,pwd,nick\_name as nickName,avatar,gmt\_created as gmtCreated,gmt\_modified as gmtModified from user where user\_name=#{userName} limit 1")**

UserDO **findByUserName**(**@Param("userName")** String name);

}

为了防止数据意外，需要添加limit 1确保只取出一条数据。

**3.API测试**

**@GetMapping("/user/findByUserName")**

**@ResponseBody**

**public** UserDO **findByUserName**(**@RequestParam("userName")** String userName) {

**return** userDAO.findByUserName(userName);

}