# Mybatis第一天

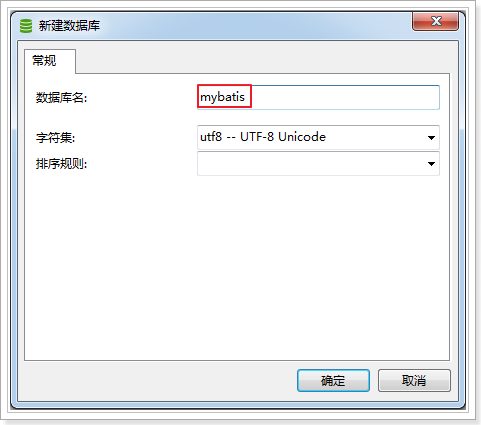
内容回顾：

1. svn
   1. 分享
   2. 检出：checkout
   3. 与资源库同步
   4. 提交:commit
   5. 更新：update
   6. 冲突的解决
2. mybatis开发流程
   1. 引入依赖
   2. 创建实体类
   3. 创建核心配置文件：mybatis-config.xml
   4. 创建映射文件，存放sql语句，在核心配置文件中引入映射文件
   5. 编写测试代码
      1. 加载核心配置文件
      2. 创建SqlSessionFactory
      3. 获取SqlSession
      4. 执行操作：通过namespace+id
      5. 释放资源

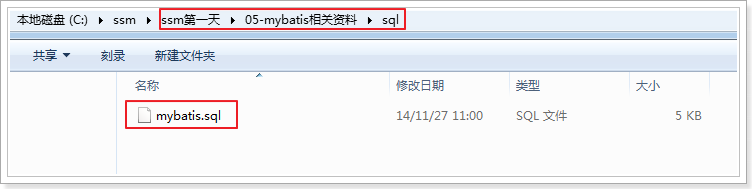
# Jdbc回顾

## 创建数据库并执行sql脚本

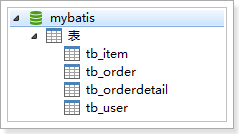
创建数据库，名字叫mybatis：



执行sql脚本：

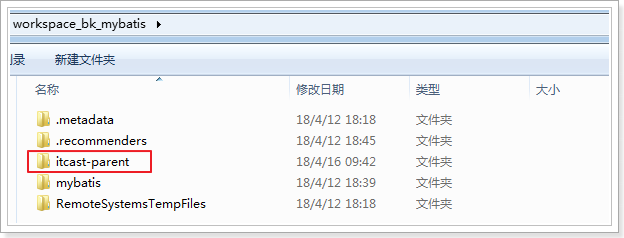


执行成功后，效果如下：

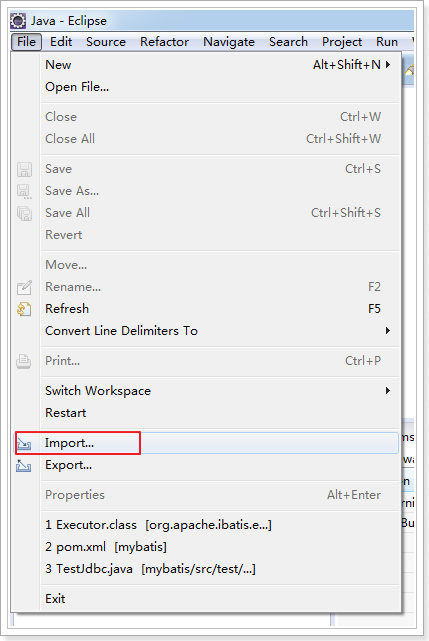


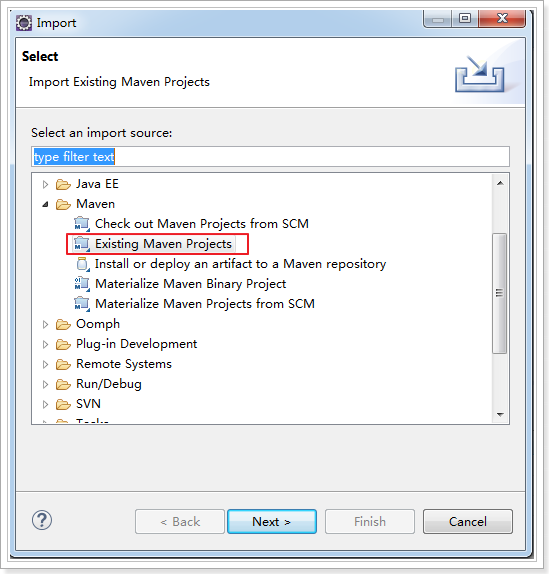
## 导入父工程

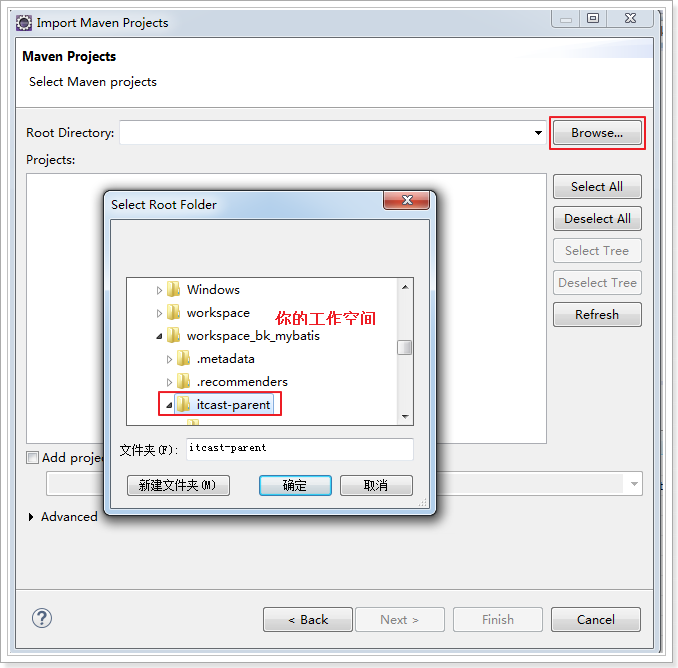
把“04-工具和环境”中的父工程itcast-parent复制到你的workspace中：



导入父工程：

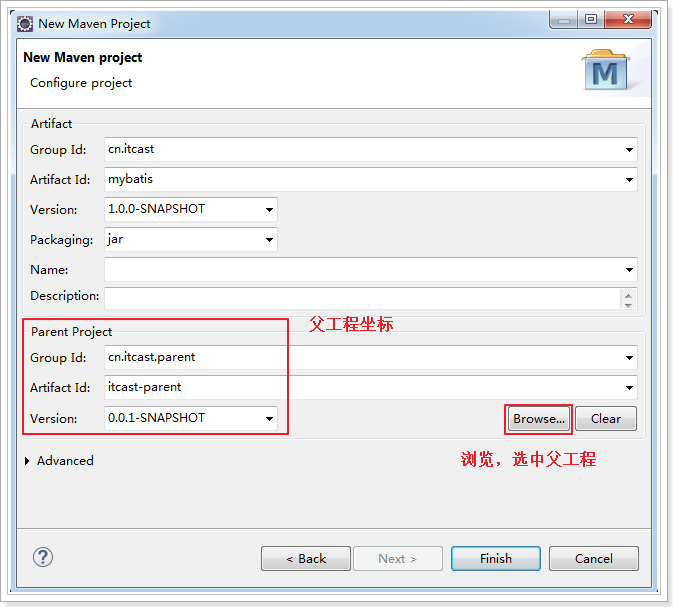








## 创建工程并引入依赖



引入mysql依赖，pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>mybatis</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<!-- MySql -->

<dependency>

<groupId>mysql</groupId>

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

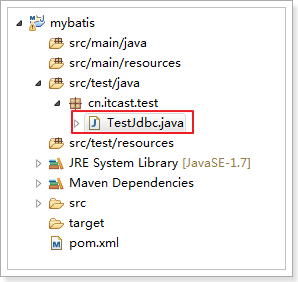
</dependency>

</dependencies>

</project>

## 编写Jdbc单元测试类

创建单元测试类TestJdbc:



TestJdbc代码：

**public** **class** TestJdbc {

@Test

**public** **void** test(){

Connection conn = **null**;

PreparedStatement ps = **null**;

ResultSet rs = **null**;

**try** {

//加载驱动

Class.*forName*("com.mysql.jdbc.Driver");

//获取数据库连接

conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mybatis", "root", "123456");

//创建预处理语句对象

ps = conn.prepareStatement("select \* from tb\_user where id = ?");

//给占位符设置值

ps.setLong(1, 1L);

//执行查询

rs = ps.executeQuery();

//遍历结果集

**while**(rs.next()){

System.***out***.println(rs.getLong("id"));

System.***out***.println(rs.getString("name"));

System.***out***.println(rs.getString("user\_name"));

}

} **catch** (Exception e) {

e.printStackTrace();

}**finally** {

//释放资源

**try**{

**if**(rs != **null**){

rs.close();

}

**if**(ps != **null**){

ps.close();

}

**if**(conn != **null**){

conn.close();

}

}**catch**(Exception e){

e.printStackTrace();

}

}

}

}

## 弊端及解决方案



# Mybatis介绍

## 简介

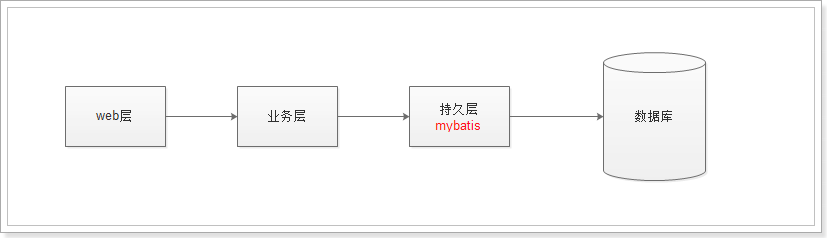


MyBatis是一个基于Java的持久层框架。那什么是框架呢？

框架是可以重复使用的一些或一整套代码，通常与具体业务无关，也可以认为是软件的半成品。

**使用框架的好处：简化项目的开发，提高开发效率。**

## MyBatis在项目的作用



# 快速入门

## 引入依赖

完整的pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>mybatis</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<!-- MySql -->

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<!-- Mybatis -->

**<dependency>**

**<groupId>org.mybatis</groupId>**

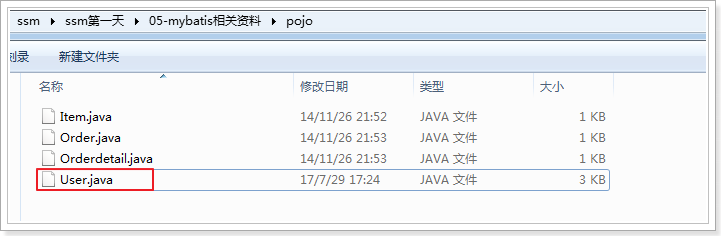
**<artifactId>mybatis</artifactId>**

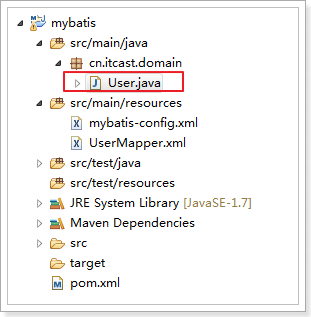
**</dependency>**

</dependencies>

</project>

## 创建User实体类





User实体类：

**public** **class** User **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Long id;

// 用户名

**private** String userName;

// 密码

**private** String password;

// 姓名

**private** String name;

// 年龄

**private** Integer age;

// 性别，1男性，2女性

**private** Integer sex;

// 出生日期

**private** Date birthday;

// 创建时间

**private** Date created;

// 更新时间

**private** Date updated;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getUserName() {

**return** userName;

}

**public** **void** setUserName(String userName) {

**this**.userName = userName;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

**public** Integer getSex() {

**return** sex;

}

**public** **void** setSex(Integer sex) {

**this**.sex = sex;

}

**public** Date getBirthday() {

**return** birthday;

}

**public** **void** setBirthday(Date birthday) {

**this**.birthday = birthday;

}

**public** Date getCreated() {

**return** created;

}

**public** **void** setCreated(Date created) {

**this**.created = created;

}

**public** Date getUpdated() {

**return** updated;

}

**public** **void** setUpdated(Date updated) {

**this**.updated = updated;

}

@Override

**public** String toString() {

**return** "User [id=" + id + ", userName=" + userName + ", password=" + password + ", name=" + name

+ ", age=" + age + ", sex=" + sex + ", birthday=" + birthday + ", created=" + created

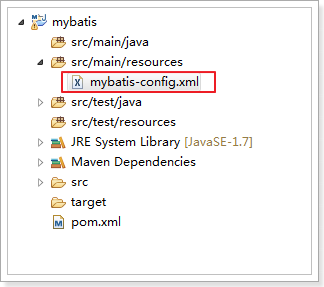
+ ", updated=" + updated + "]";

}

}

## 编写核心配置文件

在src/main/resources下创建mybatis核心配置文件mybatis-config.xml，核心配置文件的名字可以自定义，一般叫mybatis-config.xml



首先，需要引入约束信息，可以从官方文档中拷贝：



引入约束之后，mybatis-config.xml内容如下：

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

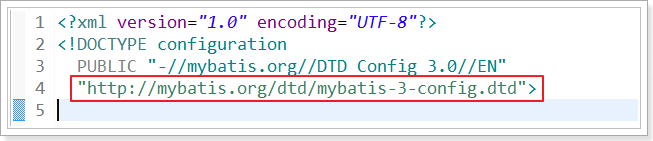
<!DOCTYPE configuration

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

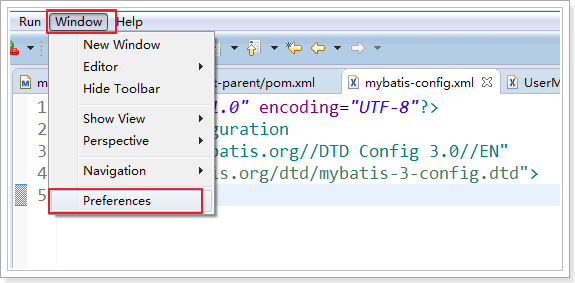
"http://mybatis.org/dtd/mybatis-3-config.dtd">

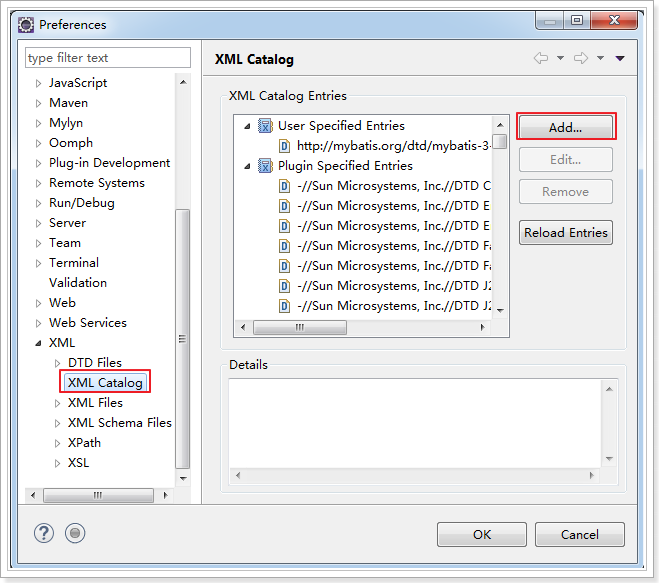
如果电脑可以联网，自动从网上下载 DTD，在编写xml文件时就有了提示；如果电脑没有联网，在本地配置DTD文件位置，提供本地提示；不管是否有网络，建议都配置本地提示。配置提示步骤如下：

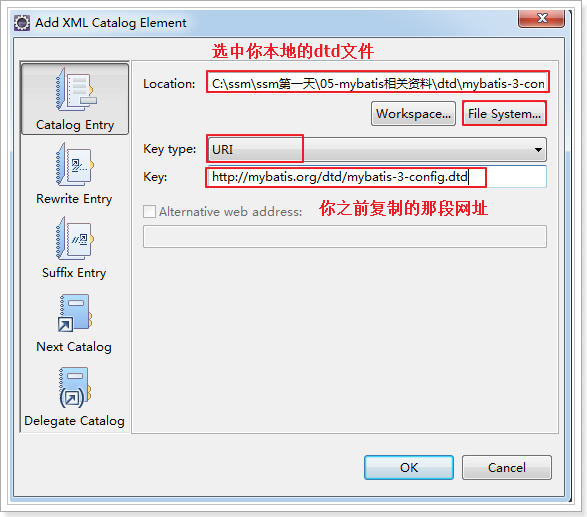
**第一步**：从mybatis-config.xml中复制网址：



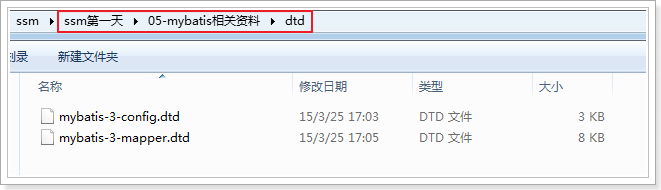
**第二步**：选择window-->Preferences







本地的dtd文件都在“05-mybatis相关资料中”：



到此，本地提示就已经配置好啦！

mybatis-config.xml内容如下（参考官方文档）：

<?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"*>

<!-- 指定驱动类名 -->

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

<!-- 指定连接地址 -->

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis"*/>

<!-- 指定用户名 -->

<property name=*"username"* value=*"root"*/>

<!-- 指定密码 -->

<property name=*"password"* value=*"123456"*/>

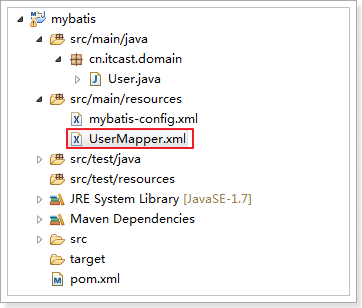
</dataSource>

</environment>

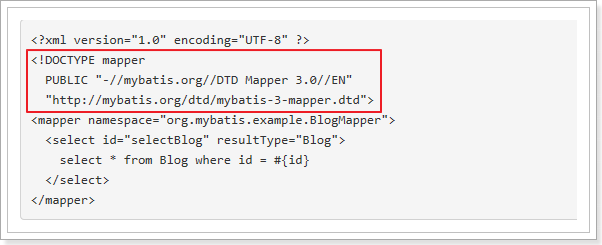
</environments>

</configuration>

## 编写映射文件



首先，需要引入约束，可以从官方文档拷贝：



引入约束之后的内容如下：

<?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">

然后，再配置提示，步骤同mybatis-config.xml配置提示。

在UserMapper增加一条sql语句，内容如下：

<?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文件的作用：主要用来存放sql语句

namespace属性：指定命名空间（名称空间），通过namespace+id的形式调用sql语句，

-->

<mapper namespace=*"UserMapper"*>

<!--

select标签：表示一条查询的语句

id属性：语句的唯一标识，在同一命名空间下唯一

resultType属性：语句的结果类型，此处写的是类的全路径名

在select标签中写查询sql语句：#{id}表示接收参数，如果只有一个参数，参数名随便写

-->

<select id=*"findUserById"* resultType=*"cn.itcast.domain.User"*>

select \* from tb\_user where id = #{id}

</select>

</mapper>

还需要在mybatis-config.xml中引入UserMapper.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=*"123456"*/>

</dataSource>

</environment>

</environments>

<mappers>

<!-- 引入映射文件 -->

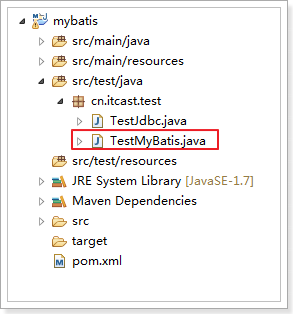
**<mapper resource=*"UserMapper.xml"*/>**

</mappers>

</configuration>

## 测试

在src/main/test下创建单元测试类TestMyBatis



TestMyBatis内容如下：

**public** **class** TestMyBatis {

@Test

**public** **void** test1() **throws** IOException{

SqlSession sqlSession = **null**;

**try** {

//读取mybatis-config.xml

InputStream in = Resources.*getResourceAsStream*("mybatis-config.xml");

//创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(in);

//创建SqlSession

sqlSession = sqlSessionFactory.openSession();

//执行查询，selectOne参数解释：

//参数1：要执行的语句，由namespace+id组成

//参数2：语句所需要的参数

User user = sqlSession.selectOne("UserMapper.findUserById", 1L);

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

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

//关闭sqlSession

**if**(sqlSession != **null**){

sqlSession.close();

}

}

}

}

## Mybatis执行流程



## 配置日志环境

执行上面的单元测试方法，我们发现控制台没有任何的日志信息输出，需要配置日志环境。

引入log4j的依赖，完整的pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>mybatis</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<!-- MySql -->

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<!-- Mybatis -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

</dependency>

**<dependency>**

**<groupId>org.slf4j</groupId>**

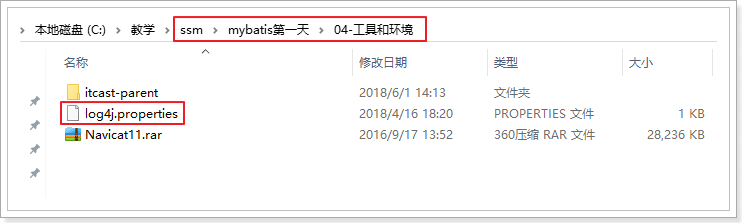
**<artifactId>slf4j-log4j12</artifactId>**

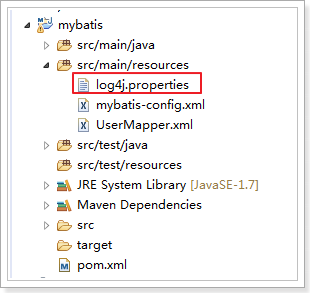
**</dependency>**

</dependencies>

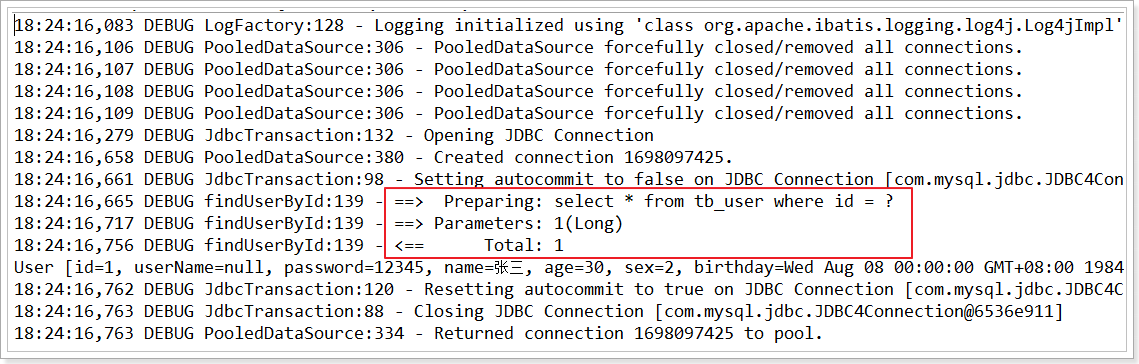
</project>

引入log4j.properties：





日志输出：



# MyBatis核心配置文件

## environments（环境）

<?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标签：配置多个环境（开发环境、测试环境）

default属性：指定默认环境的id

environment标签：配置一个具体的环境

id属性：环境的唯一标识

transactionManager标签：指定事物的类型

type属性：可以取的值有：JDBC|MANAGED

JDBC：表示从数据源获取连接来提交和回滚事物，如果单独使用mybatis，就使用该值

MANAGED：表示把事物交给容器来管理，一般不用

dataSource标签：配置数据源

type属性：指定数据源类型，可取的值有：UNPOOLED|POOLED|JNDI

UNPOOLED：无连接池的数据源，每次都重新获取数据库连接，使用完之后再关闭

POOLED：带连接池的数据源

JNDI：使用JNDI数据源

-->

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*></transactionManager>

<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=*"123456"*/>

</dataSource>

</environment>

</environments>

<mappers>

<mapper resource=*"UserMapper.xml"*/>

</mappers>

</configuration>

MyBatis 可以配置成适应多种环境，例如，开发、测试和生产环境需要有不同的配置；

尽管可以配置多个环境，每个 SqlSessionFactory 实例只能选择其一。

### 方式一：default

添加一个test（测试）环境，并在default参数中指向test环境：

<environments default=*"test"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<!-- 指定驱动类名 -->

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

<!-- 指定连接地址 -->

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis"*/>

<!-- 指定用户名 -->

<property name=*"username"* value=*"root"*/>

<!-- 指定密码 -->

<property name=*"password"* value=*"123456"*/>

</dataSource>

</environment>

<environment id=*"test"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<!-- 指定驱动类名 -->

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

<!-- 指定连接地址 -->

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis"*/>

<!-- 指定用户名 -->

<property name=*"username"* value=*"root"*/>

<!-- 指定密码 -->

<property name=*"password"* value=*"123456"*/>

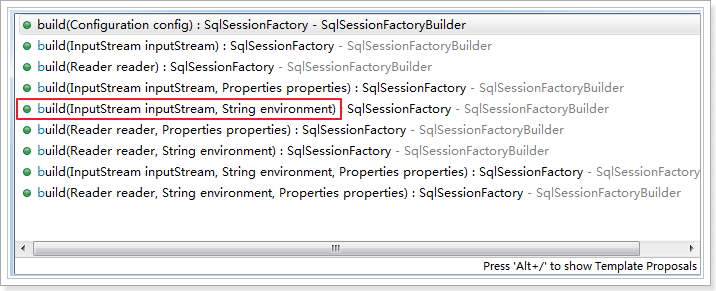
</dataSource>

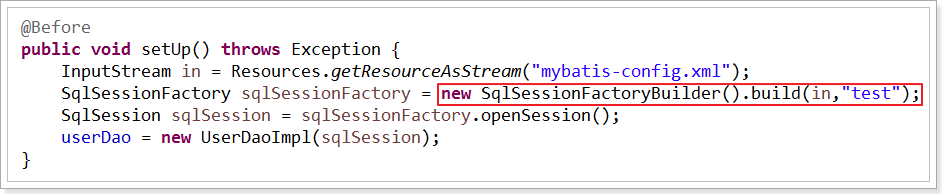
</environment>

</environments>

### 方式二：build方法

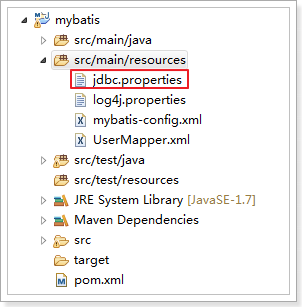
通过build的另一个重载方法来构建sqlSessionFactory





## 读取外部属性资源文件

在src/main/resources下创建jdbc.properties：



jdbc.properties内容：

jdbc.driverClass=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mybatis

jdbc.username=root

jdbc.password=123456

在mybatis-config.xml中引入jdbc.properties：

<?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>

<!-- 引入外部属性资源文件 -->

**<properties resource=*"jdbc.properties"* />**

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"****${jdbc.driverClass}****"*/>

<property name=*"url"* value=*"****${jdbc.url}****"*/>

<property name=*"username"* value=*"****${jdbc.username}****"*/>

<property name=*"password"* value=*"****${jdbc.password}****"*/>

</dataSource>

</environment>

</environments>

<mappers>

<!-- 引入映射文件 -->

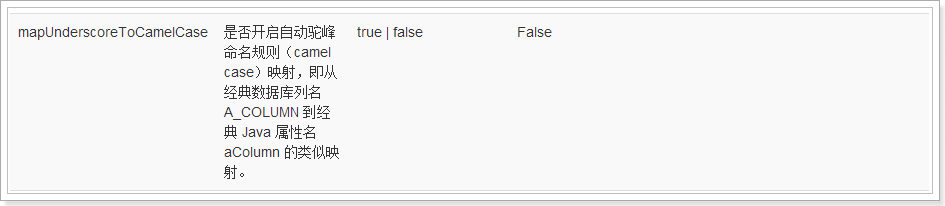
<mapper resource=*"UserMapper.xml"*/>

</mappers>

</configuration>

## settings设置

settings中可以设置参数很多，今天咱们先学习驼峰匹配。



经典数据库命名规则：如果多个单词之间，通常使用下划线进行连接。

Java中属性命名规则：驼峰式命名。

在3.5小节中，我们发现user\_name列的值没有查询出来，就是因为默认情况下，不能完成user\_name列到userName属性的映射。

需要在mybatis-config.xml中开启驼峰匹配：

<?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>

<!-- 引入外部属性资源文件 -->

<properties resource=*"jdbc.properties"* />

<settings>

<!-- 开启驼峰匹配 -->

**<setting name=*"mapUnderscoreToCamelCase"* value=*"true"*/>**

</settings>

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<!-- 指定驱动类名 -->

<property name=*"driver"* value=*"${jdbc.driverClass}"*/>

<!-- 指定连接地址 -->

<property name=*"url"* value=*"${jdbc.url}"*/>

<!-- 指定用户名 -->

<property name=*"username"* value=*"${jdbc.username}"*/>

<!-- 指定密码 -->

<property name=*"password"* value=*"${jdbc.password}"*/>

</dataSource>

</environment>

</environments>

<mappers>

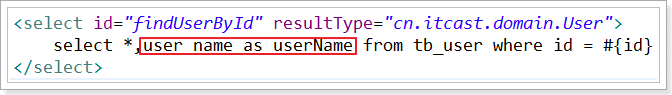
<!-- 引入映射文件 -->

<mapper resource=*"UserMapper.xml"*/>

</mappers>

</configuration>

关于解决数据库中列名有下划线而不能映射到属性的问题，还有一个解决方案，就是给列加别名：



## typeAliases

typeAliases表示类型别名，是为 Java 类型取的一个短的名字，存在的意义仅在于用来减少类完全限定名的冗余。

### 方式一：typeAliase

<?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>

<!-- 引入外部属性资源文件 -->

<properties resource=*"jdbc.properties"* />

<settings>

<!-- 开启驼峰匹配 -->

<setting name=*"mapUnderscoreToCamelCase"* value=*"true"*/>

</settings>

<typeAliases>

**<typeAlias type=*"cn.itcast.domain.User"* alias=*"User"*/>**

</typeAliases>

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<!-- 指定驱动类名 -->

<property name=*"driver"* value=*"${jdbc.driverClass}"*/>

<!-- 指定连接地址 -->

<property name=*"url"* value=*"${jdbc.url}"*/>

<!-- 指定用户名 -->

<property name=*"username"* value=*"${jdbc.username}"*/>

<!-- 指定密码 -->

<property name=*"password"* value=*"${jdbc.password}"*/>

</dataSource>

</environment>

</environments>

<mappers>

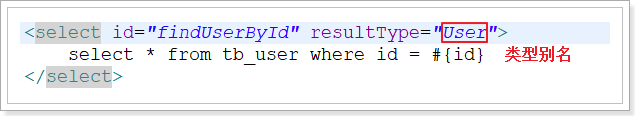
<!-- 引入映射文件 -->

<mapper resource=*"UserMapper.xml"*/>

</mappers>

</configuration>

UserMapper.xml中使用别名：



弊端：每个实体类都需要配置。

### 方式二：package

扫描该包下所有类，别名就是类名，大小写不敏感，建议使用的时候和类名一致。

<?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>

<!-- 引入外部属性资源文件 -->

<properties resource=*"jdbc.properties"* />

<settings>

<!-- 开启驼峰匹配 -->

<setting name=*"mapUnderscoreToCamelCase"* value=*"true"*/>

</settings>

<typeAliases>

<!-- <typeAlias type="cn.itcast.domain.User" alias="User"/> -->

**<package name=*"cn.itcast.domain"*/>**

</typeAliases>

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<!-- 指定驱动类名 -->

<property name=*"driver"* value=*"${jdbc.driverClass}"*/>

<!-- 指定连接地址 -->

<property name=*"url"* value=*"${jdbc.url}"*/>

<!-- 指定用户名 -->

<property name=*"username"* value=*"${jdbc.username}"*/>

<!-- 指定密码 -->

<property name=*"password"* value=*"${jdbc.password}"*/>

</dataSource>

</environment>

</environments>

<mappers>

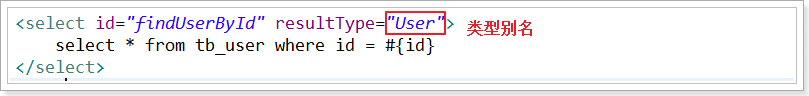
<!-- 引入映射文件 -->

<mapper resource=*"UserMapper.xml"*/>

</mappers>

</configuration>

UserMapper.xml中使用别名：



### 内置的类型别名

已经为普通的 Java 类型内建了许多相应的类型别名。它们都是大小写不敏感的，需要注意的是由于重载原始类型的名称所做的特殊处理。

|  |  |
| --- | --- |
| 别名 | 映射的类型 |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| object | Object |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

在UserMapper.xml中增加一条根据id查询用户名的sql语句：

<?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=*"UserMapper"*>

<select id=*"findUserById"* resultType=*"User"*>

select \* from tb\_user where id = #{id}

</select>

<select id=*"findUserNameById"* resultType=*"string"*>

select user\_name from tb\_user where id = #{id}

</select>

</mapper>

在TestMyBatis中编写单元测试方法：

@Test

**public** **void** test2(){

SqlSession sqlSession = **null**;

**try** {

//读取mybatis核心配置文件

InputStream in = Resources.*getResourceAsStream*("mybatis-config.xml");

//创建SqlSessionFactory

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(in);

//创建SqlSession

sqlSession = sqlSessionFactory.openSession();

//执行sql语句，selectOne参数解释

//参数1：要执行的sql语句，由namespace+id组成

//参数2：语句所需要的参数

String userName = sqlSession.selectOne("**UserMapper.findUserNameById**", 1L);

System.***out***.println(userName);

} **catch** (IOException e) {

e.printStackTrace();

}**finally**{

**if**(sqlSession != **null**){

sqlSession.close();

}

}

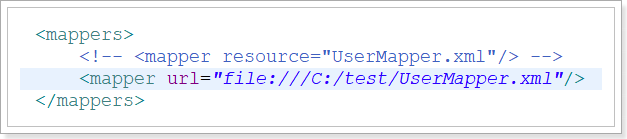
}

## mappers

### 方式一：resource

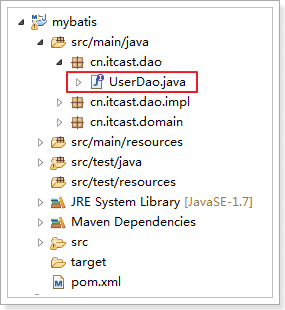


### 方式二：file（不采用）



# Mybatis中的CRUD

## 编写UserDao接口



UserDao接口：

**public** **interface** UserDao {

/\*\*

\* 根据id查询用户

\* **@param** id

\* **@return**

\*/

**public** User findUserById(Long id);

/\*\*

\* 查询所有用户

\* **@return**

\*/

**public** List<User> findAllUsers();

/\*\*

\* 保存用户

\* **@param** user

\*/

**public** **void** save(User user);

/\*\*

\* 根据id删除用户

\* **@param** id

\*/

**public** **void** delete(Long id);

/\*\*

\* 更新用户

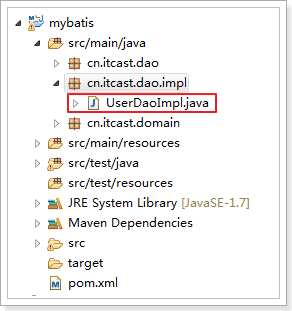
\* **@param** user

\*/

**public** **void** update(User user);

}

## 编写UserDao接口的实现类



UserDaoImpl：

**public** **class** UserDaoImpl **implements** UserDao {

**private** SqlSession sqlSession;//dao层增删改查需要sqlSession

//采用构造方法给sqlSession赋值

**public** UserDaoImpl(SqlSession sqlSession) {

**this**.sqlSession = sqlSession;

}

@Override

**public** User findUserById(Long id) {

User user = **this**.sqlSession.selectOne("UserDaoMapper.findUserById",id);

**this**.sqlSession.close();

**return** user;

}

@Override

**public** List<User> findAllUsers() {

List<User> users = **this**.sqlSession.selectList("UserDaoMapper.findAllUsers");

**this**.sqlSession.close();

**return** users;

}

@Override

**public** **void** save(User user) {

**this**.sqlSession.insert("UserDaoMapper.insertUser",user);

//mybatis把connection的自动提交功能关了，所以，在做增加、修改、删除时，我们需要手动提交事务

**this**.sqlSession.commit();

**this**.sqlSession.close();

}

@Override

**public** **void** delete(Long id) {

**this**.sqlSession.delete("UserDaoMapper.deleteUser", id);

//mybatis把connection的自动提交功能关了，所以，在做增加、修改、删除时，我们需要手动提交事务

**this**.sqlSession.commit();

**this**.sqlSession.close();

}

@Override

**public** **void** update(User user) {

**this**.sqlSession.update("UserMapper.updateUser",user);

//mybatis把connection的自动提交功能关了，所以，在做增加、修改、删除时，我们需要手动提交事务

**this**.sqlSession.commit();

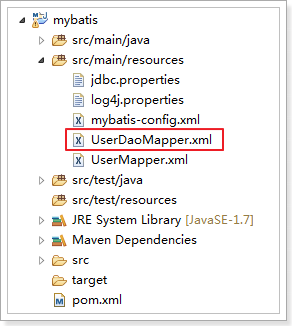
**this**.sqlSession.close();

}

}

## CRUD标签

创建UserDaoMapper.xml：



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

<!DOCTYPE mapper

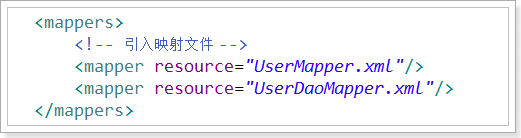
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

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

<mapper namespace=*"****UserDaoMapper****"*>

</mapper>

在mybatis-config.xml引入UserDaoMapper.xml：



下面在UserDaoMapper.xml中配置CRUD语句：

### select

根据id查询用户：

<!--

查询statement

id：statement的唯一标识，在同一命名空间下唯一

resultType：查询语句的结果类型，写的是全路径名或别名

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

内容：查询的sql

-->

<select id=*"findUserById"* resultType=*"User"*>

select \* from tb\_user where id = #{id}

</select>

查询所有用户：

<select id=*"findAllUsers"* resultType=*"User"*>

select \* from tb\_user

</select>

### insert

<!--

插入statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

useGeneratedKeys：开启主键回写

keyColumn：主键列名

keyProperty：主键对应的属性名

内容：插入的sql

插入数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<insert id=*"insertUser"* useGeneratedKeys=*"true"* keyColumn=*"id"* keyProperty=*"id"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

NOW(),

NOW()

)

</insert>

### update

<!--

更新的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：更新的sql

更新数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<update id=*"updateUser"*>

UPDATE tb\_user set

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

id = #{id}

</update>

### delete

<!--

删除的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：删除的sql

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

-->

<delete id=*"deleteUser"*>

delete from tb\_user where

id = #{id}

</delete>

### 完整的UserDaoMapper.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=*"UserDaoMapper"*>

<!-- 要给user\_name列加别名，否则不能把user\_name列的值封装到user对象的userName属性中 -->

<!--

查询statement

id：statement的唯一标识，在同一命名空间下唯一

resultType：查询语句的结果类型，写的是全路径名或别名

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

内容：查询的sql

-->

<select id=*"findUserById"* resultType=*"User"*>

select \* from tb\_user where id = #{id}

</select>

<select id=*"findAllUsers"* resultType=*"User"*>

select \* from tb\_user

</select>

<!--

插入statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

useGeneratedKeys：开启主键回写

keyColumn：主键列名

keyProperty：主键对应的属性名

内容：插入的sql

插入数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<insert id=*"insertUser"* useGeneratedKeys=*"true"* keyColumn=*"id"* keyProperty=*"id"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

NOW(),

NOW()

)

</insert>

<!--

更新的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：更新的sql

更新数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<update id=*"updateUser"*>

UPDATE tb\_user set

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

id = #{id}

</update>

<!--

删除的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：删除的sql

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

-->

<delete id=*"deleteUser"*>

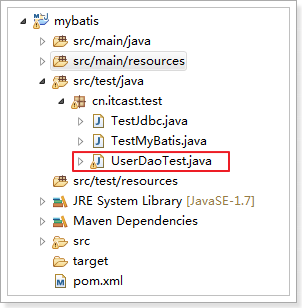
delete from tb\_user where

id = #{id}

</delete>

</mapper>

## 创建单元测试类



UserDaoTest:

**public** **class** UserDaoTest {

**private** UserDao userDao;

@Before

**public** **void** setUp() **throws** Exception {

InputStream in = Resources.*getResourceAsStream*("mybatis-config.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(in);

SqlSession sqlSession = sqlSessionFactory.openSession();

userDao = **new** UserDaoImpl(sqlSession);

}

@Test

**public** **void** testFindUserById() {

User user = userDao.findUserById(15L);

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

}

@Test

**public** **void** testFindAllUsers() {

List<User> users = userDao.findAllUsers();

**for** (User user : users) {

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

}

}

@Test

**public** **void** testSave() {

User user = **new** User();

user.setUserName("cury222");

user.setPassword("123456");

user.setName("库里");

user.setSex(1);

user.setAge(22);

Date date = **new** Date();

user.setBirthday(date);

userDao.save(user);

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

}

@Test

**public** **void** testUpdate() {

User user = **new** User();

user.setId(7L);

user.setUserName("cury333");

user.setPassword("666666");

user.setName("库里");

user.setSex(1);

user.setAge(22);

Date date = **new** Date();

user.setBirthday(date);

userDao.update(user);

}

@Test

**public** **void** testDelete() {

userDao.delete(14L);

}

}

# parameterType参数类型

parameterType有三种类型的输入参数：

1. 基本数据类型
2. hashmap
3. pojo包装类

前面已经使用了基本数据类型和pojo类型的参数，那么hashMap这种类型的参数怎么传递参数呢？

其实，它的使用方式和pojo有点类似，简单类型通过#{key}或者${key}，复杂类型通过${key.属性名}或者#{key.属性名}

在UserDaoMapper.xml中添加login语句：

<?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=*"UserDaoMapper"*>

<!--

查询statement

id：statement的唯一标识，在同一命名空间下唯一

resultType：查询语句的结果类型，写的是全路径名或别名

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

内容：查询的sql

-->

<select id=*"findUserById"* resultType=*"User"*>

select \* from tb\_user where id = #{id}

</select>

<select id=*"findAllUsers"* resultType=*"User"*>

select \* from tb\_user

</select>

<!--

插入statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

useGeneratedKeys：开启主键回写

keyColumn：主键列名

keyProperty：主键对应的属性名

内容：插入的sql

插入数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<insert id=*"insertUser"* useGeneratedKeys=*"true"* keyColumn=*"id"* keyProperty=*"id"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

NOW(),

NOW()

)

</insert>

<!--

更新的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：更新的sql

更新数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<update id=*"updateUser"*>

UPDATE tb\_user set

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

id = #{id}

</update>

<!--

删除的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：删除的sql

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

-->

<delete id=*"deleteUser"*>

delete from tb\_user where

id = #{id}

</delete>

<!-- 当参数类型是map时，根据#{key}来取参数值 ，parameterType="hashmap"可以省略不写-->

**<select id=*"login"* resultType=*"User"*>**

**select \* from tb\_user where user\_name = #{userName} and password = #{password}**

**</select>**

</mapper>

在UserDao接口中添加login方法：

**public** **interface** UserDao {

/\*\*

\* 根据id查询用户

\* **@param** id

\* **@return**

\*/

**public** User findUserById(Long id);

/\*\*

\* 查询所有用户

\* **@return**

\*/

**public** List<User> findAllUsers();

/\*\*

\* 保存用户

\* **@param** user

\*/

**public** **void** save(User user);

/\*\*

\* 根据id删除用户

\* **@param** id

\*/

**public** **void** delete(Long id);

/\*\*

\* 更新用户

\* **@param** user

\*/

**public** **void** update(User user);

/\*\*

\* 登录

\* **@param** userName

\* **@param** password

\* **@return**

\*/

**public** User login(String userName,String password);

}

在UserDaoImpl中实现login方法：

**public** **class** UserDaoImpl **implements** UserDao {

**private** SqlSession sqlSession;//dao层增删改查需要sqlSession

//采用构造方法给sqlSession赋值

**public** UserDaoImpl(SqlSession sqlSession) {

**this**.sqlSession = sqlSession;

}

@Override

**public** **void** save(User user) {

**this**.sqlSession.insert("UserDaoMapper.insertUser",user);

**this**.sqlSession.commit();

}

@Override

**public** **void** delete(Long id) {

**this**.sqlSession.delete("UserDaoMapper.deleteUser", id);

**this**.sqlSession.commit();

}

@Override

**public** **void** update(User user) {

**this**.sqlSession.update("UserDaoMapper.updateUser",user);

**this**.sqlSession.commit();

}

@Override

**public** User findUserById(Long id) {

User user = **this**.sqlSession.selectOne("UserDaoMapper.findUserById",id);

**return** user;

}

@Override

**public** List<User> findAllUsers() {

List<User> users = **this**.sqlSession.selectList("UserDaoMapper.findAllUsers");

**return** users;

}

@Override

**public** User login(String userName, String password) {

Map<String,Object> params = **new** HashMap<String,Object>();

params.put("userName", userName);

params.put("password", password);

User user = **this**.sqlSession.selectOne("UserDaoMapper.login",params);

**return** user;

}

}

在UserDaoTest中增加testLogin方法：

@Test

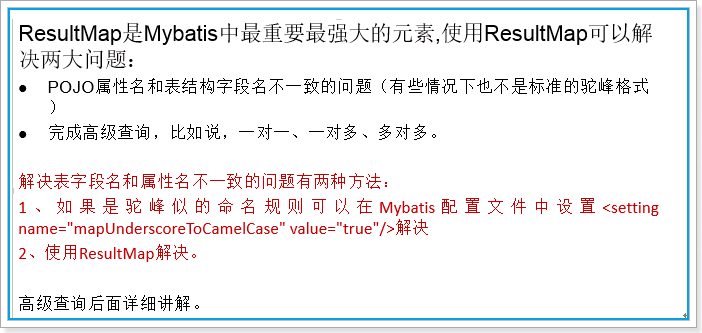
**public** **void** testLogin() {

User user = userDao.login("zhangsan", "12345");

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

}

# resultMap



查询数据时，查询不到userName的信息。

原因：表中的列名叫user\_name，而实体类中的属性名却叫userName，两者不一致，造成mybatis无法把user\_name列的值填充到userName属性中。

有三种解决方案：

1. 开启驼峰匹配
2. 加别名
3. resultMap自定义结果映射

UserDaoMapper.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=*"UserDaoMapper"*>

<!-- resultMap自定义结果集映射关系

id属性：唯一标识

type属性：映射结果的类型

autoMapping：开启自动映射，列名对应相同名字属性名，如果开启了驼峰匹配，会按照驼峰来匹配，默认true

id标签：映射主键，不要省略，可以提高性能，column指定列名，property指定属性名

result标签：映射普通属性

-->

**<resultMap type=*"User"* id=*"userMap"* autoMapping=*"true"*>**

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

**<!-- <result column="user\_name" property="userName"/> -->**

**</resultMap>**

<!-- 要给user\_name列加别名，否则不能把user\_name列的值封装到user对象的userName属性中 -->

<!--

查询statement

id：statement的唯一标识，在同一命名空间下唯一

resultType：查询语句的结果类型，写的是全路径名或别名

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

内容：查询的sql

-->

<select id=*"findUserById"* **resultMap=*"userMap"***>

select \* from tb\_user where id = #{id}

</select>

<select id=*"findAllUsers"* resultType=*"User"*>

select \* from tb\_user

</select>

<!--

插入statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

useGeneratedKeys：开启主键回写

keyColumn：主键列名

keyProperty：主键对应的属性名

内容：插入的sql

插入数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<insert id=*"insertUser"* useGeneratedKeys=*"true"* keyColumn=*"id"* keyProperty=*"id"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

NOW(),

NOW()

)

</insert>

<!--

更新的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：更新的sql

更新数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<update id=*"updateUser"*>

UPDATE tb\_user set

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

id = #{id}

</update>

<!--

删除的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：删除的sql

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

-->

<delete id=*"deleteUser"*>

delete from tb\_user where

id = #{id}

</delete>

<!-- 当参数类型是map时，根据#{key}来取参数值，parameterType="hashmap"可以省略不写 -->

<select id=*"login"* resultType=*"User"*>

select \* from tb\_user where user\_name = #{userName} and password = #{password}

</select>

</mapper>

# sql片段

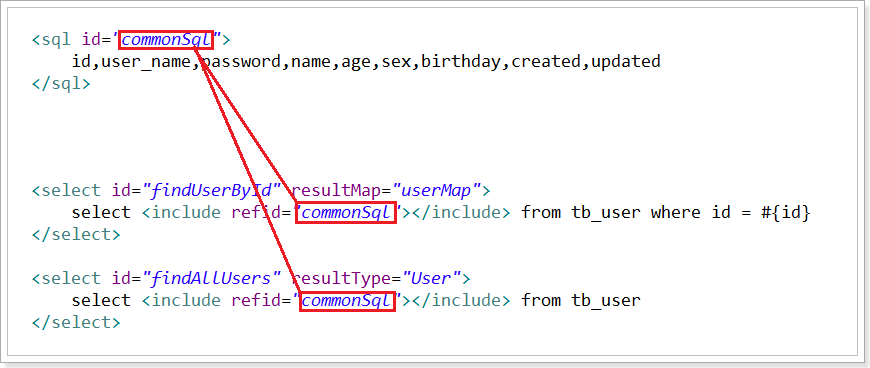
在java代码中，针对一些出现频率较高的代码，我们会抽出来形成一个公共方法或公共类。

那么，在mybatis中如果某些sql语句经常出现，那我们又该怎么做呢？

可以采用sql片段。

## 用法一

sql标签可以定义一个sql片段，再需要用到该sql片段的地方，用include标签引入：



UserDaoMapper.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=*"UserDaoMapper"*>

<!-- resultMap自定义结果集映射关系

id属性：唯一标识

type属性：映射结果的类型

autoMapping：开启自动映射，列名对应相同名字属性名，如果开启了驼峰匹配，会按照驼峰来匹配，默认true

id标签：映射主键，不要省略，可以提高性能，column指定列名，property指定属性名

result标签：映射普通属性

-->

<resultMap type=*"User"* id=*"userMap"* autoMapping=*"true"*>

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

<!-- <result column="user\_name" property="userName"/> -->

</resultMap>

**<sql id=*"commonSql"*>**

**id,user\_name,password,name,age,sex,birthday,created,updated**

**</sql>**

<!-- 要给user\_name列加别名，否则不能把user\_name列的值封装到user对象的userName属性中 -->

<!--

查询statement

id：statement的唯一标识，在同一命名空间下唯一

resultType：查询语句的结果类型，写的是全路径名或别名

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

内容：查询的sql

-->

<select id=*"findUserById"* resultMap=*"userMap"*>

select **<include refid=*"commonSql"*></include>** from tb\_user where id = #{id}

</select>

<select id=*"findAllUsers"* resultType=*"User"*>

select **<include refid=*"commonSql"*></include>** from tb\_user

</select>

<!--

插入statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

useGeneratedKeys：开启主键回写

keyColumn：主键列名

keyProperty：主键对应的属性名

内容：插入的sql

插入数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<insert id=*"insertUser"* useGeneratedKeys=*"true"* keyColumn=*"id"* keyProperty=*"id"*>

INSERT INTO tb\_user (

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

NOW(),

NOW()

)

</insert>

<!--

更新的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：更新的sql

更新数据时，传过来的参数是一个对象，#{\*\*\*}中的名字必须与对象中属性名一致

-->

<update id=*"updateUser"*>

UPDATE tb\_user set

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

id = #{id}

</update>

<!--

删除的statement

id：statement的唯一标识，在同一命名空间下唯一

parameterType：语句中占位符的值的类型，写的是全路径名或别名，可以省略

内容：删除的sql

#{id}表示接收参数的值，如果只有一个参数，#{\*\*\*}中的名字可以随便写

-->

<delete id=*"deleteUser"*>

delete from tb\_user where

id = #{id}

</delete>

<!-- 当参数类型是map时，根据#{key}来取参数值，parameterType="hashmap"可以省略不写 -->

<select id=*"login"* resultType=*"User"*>

select \* from tb\_user where user\_name = #{userName} and password = #{password}

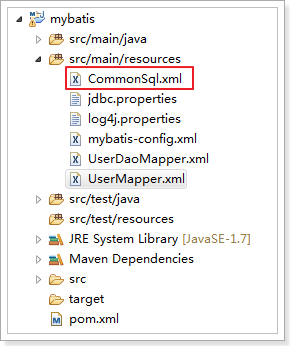
</select>

</mapper>

## 用法二

很多时候，同一个sql片段可能在很多个mapper文件里都有使用，这个时候就要添加一个映射文件，来统一管理sql片段。

创建CommonSql.xml文件：



CommonSql.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=*"CommonSql"*>

<sql id=*"commonSql"*>

id,user\_name,password,name,age,sex,birthday,created,updated

</sql>

</mapper>

在mybatis-conig.xml中引入CommonSql.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>

<!-- 引入外部属性资源文件 -->

<properties resource=*"jdbc.properties"* />

<settings>

<!-- 开启驼峰匹配 -->

<setting name=*"mapUnderscoreToCamelCase"* value=*"true"*/>

</settings>

<typeAliases>

<!-- <typeAlias type="cn.itcast.domain.User" alias="user"/> -->

<package name=*"cn.itcast.domain"*/>

</typeAliases>

<!--

environments标签：配置mybatis的环境，可以有多个环境

default属性：指定默认环境的id

environment标签：表示每一个mybatis环境

id属性：该环境的id

transactionManager标签：指定事务类型

type属性：指定事务类型。可以取的值有：JDBC|MANAGED

JDBC:表示从数据源得到连接来提交和回滚事务,如果单独使用mybatis就使用该值

MANAGED:这个配置几乎没做什么，让容器(spring)来管理事务

dataSource标签：配置数据源

type属性：指定数据源的类型，可取的值有:UNPOOLED|POOLED|JNDI

UNPOOLED:无连接池的数据源，每次都新开一个连接，使用完之后关闭连接

POOLED:有连接池的数据源

JNDI：配置JNDI数据源

-->

<environments default=*"test"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<!-- 指定驱动类名 -->

<property name=*"driver"* value=*"${jdbc.driverClass}"*/>

<!-- 指定连接地址 -->

<property name=*"url"* value=*"${jdbc.url}"*/>

<!-- 指定用户名 -->

<property name=*"username"* value=*"${jdbc.username}"*/>

<!-- 指定密码 -->

<property name=*"password"* value=*"${jdbc.password}"*/>

</dataSource>

</environment>

<environment id=*"test"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<!-- 指定驱动类名 -->

<property name=*"driver"* value=*"${jdbc.driverClass}"*/>

<!-- 指定连接地址 -->

<property name=*"url"* value=*"${jdbc.url}"*/>

<!-- 指定用户名 -->

<property name=*"username"* value=*"${jdbc.username}"*/>

<!-- 指定密码 -->

<property name=*"password"* value=*"${jdbc.password}"*/>

</dataSource>

</environment>

</environments>

<mappers>

<!-- 引入映射文件 -->

<mapper resource=*"UserMapper.xml"*/>

<mapper resource=*"UserDaoMapper.xml"*/>

**<mapper resource=*"CommonSql.xml"*/>**

</mappers>

</configuration>

在UserDaoMapper.xml中使用sql片段，使用的时候，要带上命名空间：

<select id=*"findUserById"* resultMap=*"userMap"*>

select **<include refid=*"CommonSql.commonSql"*></include>** from tb\_user where id = #{id}

</select>

<select id=*"findAllUsers"* resultType=*"User"*>

select **<include refid=*"CommonSql.commonSql"*></include>** from tb\_user

</select>