1. mybatis框架



>>课程计划：

1、MyBatis快速入门

2、MyBatis对数据库中数据的增删改查操作

3、#{}占位符的应用

4、动态SQL的应用

6、MyBatis的Mapper接口开发

# MyBatis简介(了解)

## 什么是MyBatis

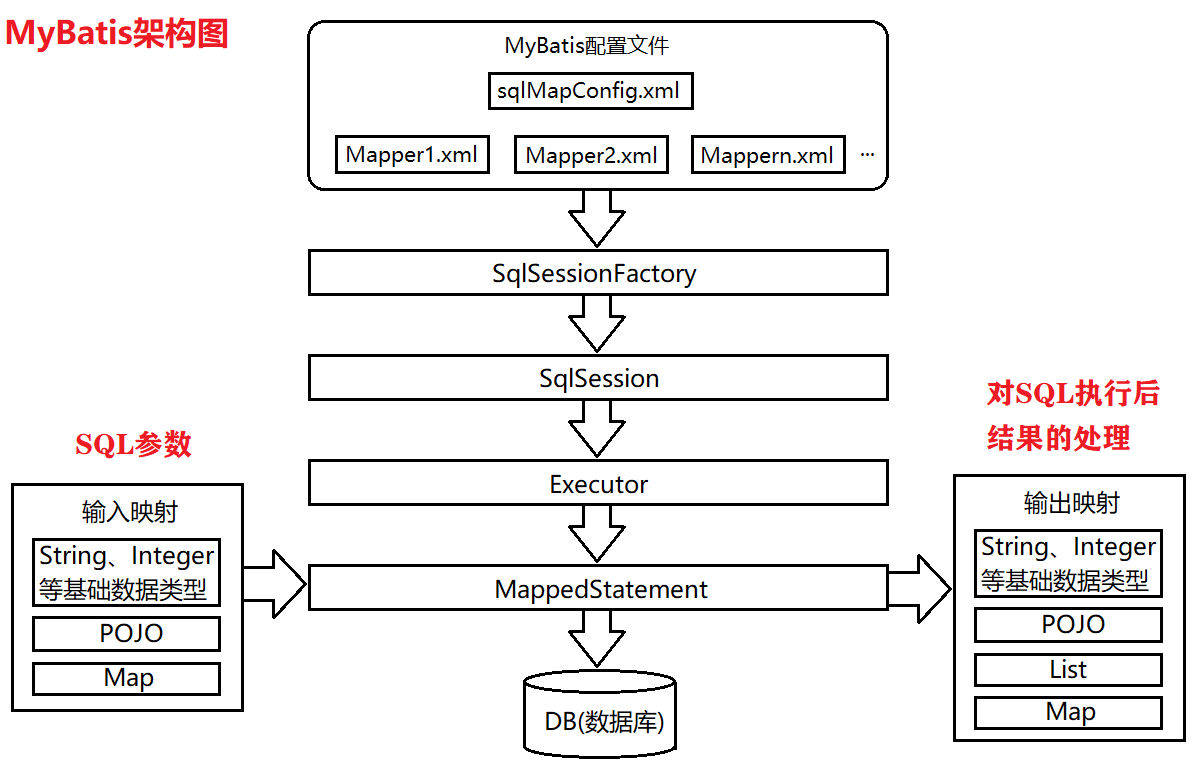
MyBatis 本是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github。

MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。

Mybatis通过xml或注解的方式将要执行的各种statement（statement、preparedStatemnt）配置起来，并通过java对象和statement中的sql进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射成java对象并返回。

总之，Mybatis对JDBC访问数据库的过程进行了封装，简化了JDBC代码，解决JDBC将结果集封装为Java对象的麻烦。

下图是MyBatis架构图：



(1)sqlMapConfig.xml是Mybatis的核心配置文件，通过其中的配置可以生成SqlSessionFactory,也就是SqlSession工厂

(2)基于SqlSessionFactory可以生成SqlSession对象

(3)SqlSession是一个既可以发送SQL去执行，并返回结果，类似于JDBC中的Connection对象，也是Mybatis中至关重要的一个对象。

(4)Executor是SqlSession底层的对象，用于执行SQL语句

(5)MapperStatement对象也是SqlSession底层的对象，用于接收输入映射（SQL语句中的参数），以及做输出映射（即将SQL查询的结果映射成相应的结果）

## 为什么要使用MyBatis

思考：在开始之前，思考下如何通过JDBC查询Emp表中的所有记录，并封装到一个List<Emp>集合中返回。（演示：准备数据、导包、导入JDBC程序）

1、使用传统方式JDBC访问数据库：

(1)使用JDBC访问数据库有大量重复代码（比如注册驱动、获取连接、获取传输器、释放资源等）；

(2)JDBC自身没有连接池，会频繁的创建连接和关闭连接，效率低；

(3)SQL是写死在程序中，一旦修改SQL，需要对类重新编译；

(4)对查询SQL执行后返回的ResultSet对象，需要手动处理，有时会特别麻烦；

...

2、使用mybatis框架访问数据库：

(1)Mybatis对JDBC对了封装，可以简化JDBC代码；

(2)Mybatis自身支持连接池（也可以配置其他的连接池），因此可以提高程序的效率；

(3)Mybatis是将SQL配置在mapper文件中，修改SQL只是修改配置文件，类不需要重新编译。

(4)对查询SQL执行后返回的ResultSet对象，Mybatis会帮我们处理，转换成Java对象。

...

总之，JDBC中所有的问题（代码繁琐、有太多重复代码、需要操作太多对象、释放资源、对结果的处理太麻烦等），在Mybatis框架中几乎都得到了解决！！

# MyBatis快速入门

## 准备数据，创建库和表

1、创建yonghedb库、emp表，并插入若干条记录

-- 1、删除 yonghedb 数据库（如果存在）

drop database if exists yonghedb;

-- 2、创建数据库 yonghedb 数据库

create database yonghedb charset utf8;

use yonghedb; -- 选择yonghedb数据库

-- 3、在 yonghedb 库中创建 emp 表

create table emp(

id int primary key auto\_increment,

name varchar(50),

job varchar(50),

salary double

);

-- 4、往 emp 表中, 插入若干条记录

insert into emp values(null, '王海涛', '程序员', 3300);

insert into emp values(null, '齐雷', '程序员', 2800);

insert into emp values(null, '刘沛霞', '程序员鼓励师', 3700);

insert into emp values(null, '陈子枢', '部门总监', 4200);

insert into emp values(null, '刘昱江', '程序员', 3000);

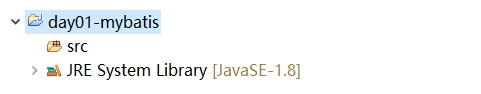
insert into emp values(null, '董长春', '程序员', 3500);

insert into emp values(null, '张慎政', '程序员', 2700);

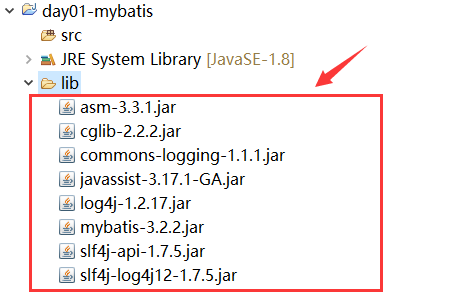
insert into emp values(null, '韩少云', 'CEO', 5000);

## 创建工程，导入所需jar包、创建测试类

1、创建java工程

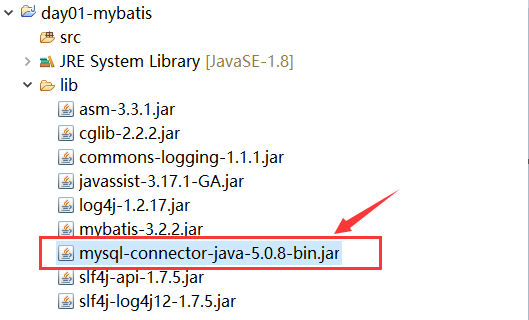


2、导入mybatis的jar包

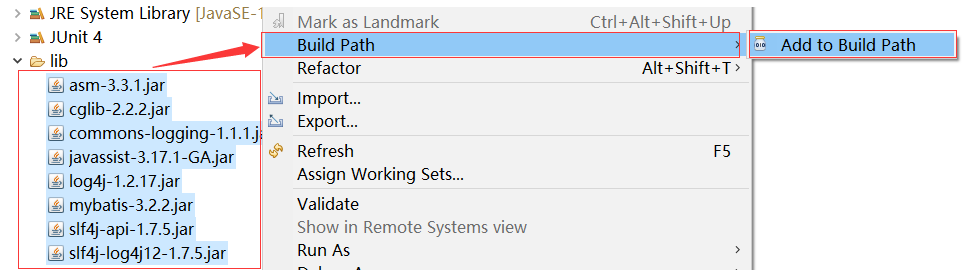


其中mybatis-3.2.2是mybatis框架的核心包，其他则为依赖包。

3、导入mysql驱动包



4、引用所有jar包



5、创建com.tedu.test.TestMybatis测试类，并提供findAll方法（查询emp表中所有的员工信息），开发步骤如下：



/\*\*

\* Mybatis的快速入门

\*/

**public** **class** TestMybatis {

/\*\*

\* 1.查询mybatisdb库中的emp表中所有员工，将所有员工记录以一个List

\* 集合（List<Emp>）返回

\* **@throws** IOException

\*/

@Test

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

//1.读取sqlMapConfig.xml文件，获取其中的基本信息

//2. 根据配置信息生成SqlSessionFactory工厂对象，

//3.通过工厂获取一个SqlSession对象(用于执行SQL及返回结果)

//4.执行SQL语句，查询emp表中的所有记录，封装到List集合中

//5.打印list集合

}

}

## 添加sqlMapConfig.xml文件

1、在src目录下，创建sqlMapConfig.xml文件 （MyBatis的核心配置文件）



**2、sqlMapConfig.xml文件配置如下:**

sqlMapConfig文件头信息如下：

<?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">

<!-- MyBatis的全局配置文件 -->

<configuration >

</configuration>

详细配置如下：

<?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">

<!-- MyBatis的全局配置文件 -->

<configuration >

<!-- 1.配置开发环境（需设置一个默认的环境） -->

<environments default=*"develop"*>

<!-- 这里可以配置多个环境，比如develop，test等 -->

<environment id=*"develop"*>

<!-- 1.1.配置事务管理方式：JDBC/MANAGED

JDBC：将事务交给JDBC管理（推荐）

MANAGED：自己管理事务

-->

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

<!-- 1.2.配置数据源，即连接池方式:JNDI/POOLED/UNPOOLED

JNDI：已过时

POOLED：使用连接池（推荐）

UNPOOLED：不使用连接池

-->

<dataSource type=*"POOLED"*>

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

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/yonghedb?characterEncoding=utf-8"*/>

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

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

</dataSource>

</environment>

</environments>

<!-- 2.加载Mapper配置文件(因mapper文件中配置了要执行的SQL语句) -->

<mappers>

<!-- 注意路径 -->

<mapper resource=*"com/tedu/pojo/EmpMapper.xml"*/>

</mappers>

</configuration>

## 编写Emp实体类

注意：在当前实例中，Emp类中的属性和数据库表的字段名称必须一致，否则将会无法将结果集封装到Java对象中。

实现Emp类： 提供私有属性以及对应的getter方法、setter方法，并重写toString方法

**package** com.tedu.pojo;

/\*\*

\* 实体类,用于封装Emp表中的一条用户信息

\*/

**public** **class** Emp {

//1.声明实体类中的属性

**private** Integer id;

**private** String name;

**private** String job;

**private** Double salary;

//2.提供对应的getter和setter方法

**public** Integer getId() {

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** String getJob() {

**return** job;

}

**public** **void** setJob(String job) {

**this**.job = job;

}

**public** Double getSalary() {

**return** salary;

}

**public** **void** setSalary(Double salary) {

**this**.salary = salary;

}

//3.重写toString方法

@Override

**public** String toString() {

**return** "Emp [id=" + id + ", name=" + name + ", job=" + job + ", salary=" + salary + "]";

}

}

## 添加EmpMapper.xml文件

1、在src/com/tedu/pojo目录下，创建EmpMapper.xml文件 （实体类的映射文件）



2、EmpMapper.xml文件配置如下:

EmpMapper文件头信息如下：

<?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">

<!--

namespace一般指定为当前文件的所在包路径+文件名（将来是接口名）

在程序中通过[ **namespace** + **id** ]定位到执行哪一条SQL语句

-->

<mapper namespace=*""*>

</mapper>

详细配置如下：

<?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">

<!--

namespace一般指定为当前文件的所在包路径+文件名

将来在程序中通过[ **namespace** + **id** ]定位到执行哪一条SQL语句

-->

<mapper namespace=*"****com.tedu.pojo.EmpMapper****"*>

<!-- 通过select、insert、update、delete标签声明要执行的SQL -->

<select id=*"****findAll****"* resultType=*"com.tedu.pojo.Emp"*>

select \* from emp

</select>

<!--

resultType：返回值类型，简单类型（例如：Integer，String，Emp等）

如果返回集合(List<Emp>)，只需配置集合中的元素类型即可！

resultMap：复杂对象结构（例如多表关联查询等），后面用到再讲解

-->

</mapper>

## 实现测试类,并测试

1、实现findAll方法，代码如下：

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

//1.读取sqlMapConfig.xml文件，获取其中的基本信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//2.根据配置信息生成SqlSessionFactory工厂对象，

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

//3.通过工厂获取一个SqlSession对象(用于执行SQL及返回结果)

SqlSession session = factory.openSession();

//4.执行SQL语句，查询emp表中的所有记录，封装到List集合中

String sqlId = "**com.tedu.pojo.EmpMapper**.**findAll**";

List<Emp> emplist = session.selectList( sqlId );

//5.打印list集合

**for**(Emp emp : emplist){

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

}

}

2、执行findAll方法，输出结果为:

Emp [id=1, name=王海涛, job=程序员, salary=3300.0]

Emp [id=2, name=齐雷, job=程序员, salary=2800.0]

Emp [id=3, name=刘沛霞, job=程序员鼓励师, salary=3700.0]

Emp [id=4, name=陈子枢, job=部门总监, salary=4200.0]

Emp [id=5, name=刘昱江, job=程序员, salary=3000.0]

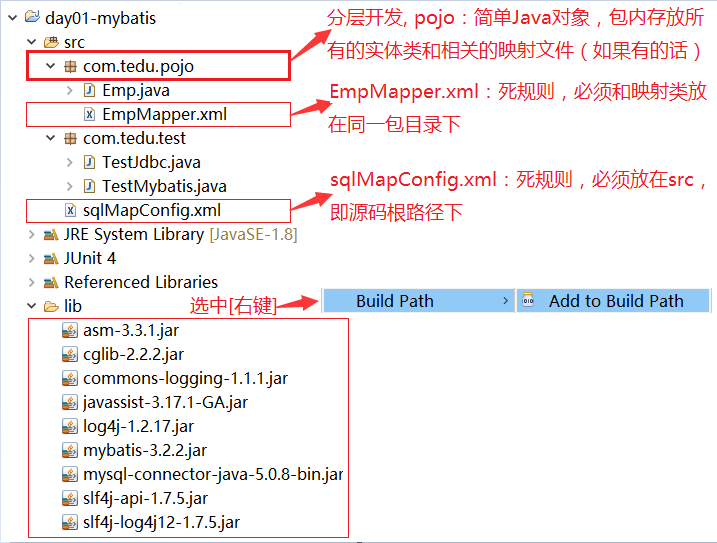
Emp [id=6, name=董长春, job=程序员, salary=3500.0]

Emp [id=7, name=张慎政, job=程序员, salary=2700.0]

Emp [id=8, name=韩少云, job=CEO, salary=5000.0]

# MyBatis入门细节

## 工程目录及文件介绍



**1、pojo（Plain Ordinary Java Object）：** 简单的Java对象，实际就是普通JavaBean，也叫做实体类，用于封装一类信息。例如：提供Emp类，用于封装和用户相关的属性信息，比如id员工编号、name员工姓名、job员工职位、salary薪资等。再例如：提供Product类，用于封装和商品相关的属性信息，比如id商品编号、name商品名称、price单价等。

而属性一般是私有的，因此我们还会提供属性相关的setter和getter方法。

我们一般会把实体类，放在pojo目录下，而和实体类相关的mapper文件，也放在相同的目录下（Maven工程除外）

**2、EmpMapper.xml：**Emp实体类的映射文件，和Emp类放在同一目录下（Maven工程除外）

**3、sqlMapConfig.xml：**mybatis的全局配置文件，很多配置信息都会在这里进行配置，比如事务、连接数据库基本信息，连接池（数据源）等配置

**4、lib及下的jar包：**将所需jar包拷贝过来后，还需要引用jar包到工程，也就是选中jar包，右键 --> Build Path --> Add to Build Path即可!

## sqlMapConfig.xml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">

<!-- MyBatis的全局配置文件 -->

<configuration >

<!-- 配置环境，可配置多个环境（比如：develop开发、test测试） -->

<environments default=*"develop"*>

<environment id=*"develop"*>

<!-- 配置事务管理方式：JDBC/MANAGED

JDBC：将事务交给JDBC管理（推荐）

MANAGED：自己管理事务

-->

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

<!-- 配置数据源，即连接池 JNDI/POOLED/UNPOOLED

JNDI：已过时

POOLED：使用连接池（推荐）

UNPOOLED：不使用连接池

-->

<dataSource type=*"POOLED"*>

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

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatisdb?useUnicode=true&amp;characterEncoding=utf-8"*/>

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

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

</dataSource>

</environment>

</environments>

<!-- 引入Mapper配置文件，可以配置多个 -->

<mappers>

<mapper resource=*"com/tedu/pojo/EmpMapper.xml"*/>

</mappers>

</configuration>

**configuration**是根标签，当前文件中所有的配置都在该标签内，注意其中配置的关键点:

默认的环境 ID（比如:default="*develop*"）。

每个 environment 元素定义的环境 ID（比如:id="*develop*"）。

事务管理器的配置（比如:type="*JDBC*"）。

数据源的配置（比如:type="*POOLED*"）。

**(1)environments标签：**该标签内部可以配置多个environment，即多种环境，每种环境可以做不同配置或连接不同数据库。例如，开发、测试、生产环境可能需要不同的配置，连接的数据库可能也不相同，因此我们可以配置三个environment，分别对应上面三种不同的环境。

但是要记住，environment可以配置多个，但是最终要使用的只能是其中一个！

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader);

SqlSessionFactory factory = sqlSessionFactoryBuilder.build(reader, environment);

**(2)environment标签：**内部可以配置多种配置信息，下面介绍事务管理配置和数据源配置。

**(3)transactionManage标签：**事务管理配置，mybatis中有两种事务管理方式，也就是 type="[JDBC|MANAGED]。

**JDBC**：这个配置就是直接使用了 JDBC 的提交和回滚设置，它依赖于从数据源得到的连接来管理事务范围。推荐使用。

**MANAGED**：这个配置几乎没做什么。它从来不提交或回滚一个连接。需要自己手动添加并管理。不推荐使用。

**(4)dataSource标签：**数据源，也就是连接池配置。这里type指定数据源类型，有三种内建的类型：JNDI、POOLED、UNPOOLED

**JNDI**：已过时，不推荐使用！

**POOLED**：使用连接池，mybatis会创建连接池，并从连接池中获取连接访问数据库，在操作完成后，将会把连接返回连接池。

**UNPOOLED**：不使用连接池，该方式适用于只有小规模数量并发用户的简单应用程序上。

**(5)mappers标签：**用于导入mapper文件的位置，其中可以配置多个mapper，即可以导入多个mapper文件。

## EmpMapper.xml配置文件

<?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">

<!--

namespace一般指定为当前文件的所在包路径+文件名

将来在程序中通过[ **namespace** + **id** ]定位到执行哪一条SQL语句

-->

<mapper namespace=*"****com.tedu.pojo.EmpMapper****"*>

<!-- 通过select、insert、update、delete标签声明要执行的SQL -->

<select id=*"****findAll****"* resultType=*"com.tedu.pojo.Emp"*>

select \* from emp

</select>

<!--

resultType：返回值类型，简单类型（例如：Integer，String，Emp等）

如果返回集合(List<Emp>)，只需配置集合中的元素类型即可！

resultMap：复杂对象结构（例如多表关联查询等），后面用到再讲解

-->

</mapper>

(1)第1行是xml的文档声明，用于声明xml的版本和编码

(2)第2、3、4行，引入了xml约束文档，当前xml文档将会按照mybatis-3-mapper.dtd文件所要求的规则进行书写。

**(3)Mapper标签：**根标签，其中namespace(名称空间，也叫命名空间)，要求不能重复。其实就是一个名称，一般我们指定为"包名+文件名"。

**(4)select标签：**用于指定将来要执行的各种SQL语句。标签上可以声明属性，下面介绍常用的属性：id、resultType、resultMap

id属性：要求值不能重复。将来在执行SQL时，可以通过namespace + id找到指定SQL并执行。

resultType属性：从这条SQL语句中返回所期望类型的类的完全限定名称（包名+类名）。注意如果是集合情形，那应该是集合可以包含的类型，而不能是集合本身。

简而言之，resultType控制查询SQL执行后返回值的类型或集合中的泛型，例如查询emp表中的单条记录，返回值是一个Emp对象，因此，resultType="com.tedu.pojo.Emp"；

如果查询emp表中的多条记录，返回值是一个List<Emp>，此时resultType的值应该集合中的泛型，因此resultType="com.tedu.pojo.Emp"；

resultMap属性：复杂对象结构（例如多表关联查询等）。 使用 resultType 或 resultMap，但不能同时使用。

# MyBatis增删改查

## 新增员工

**1、编辑EmpMapper.xml文件, 添加新增员工对应的sql.**

<!-- 2.新增员工 -->

<insert id=*"****insert****"*>

insert into emp(name,job,salary) values('韩少云', 'CEO', 5000)

</insert>

2**、编写TestMybatis类，添加addEmp方法，实现新增员工操作。**

/\*\*

\* 2.新增：往emp表中添加一个新员工

\* **@throws** Exception

\*/

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

//1.读取sqlMapConfig.xml文件, 获取所有配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//2.创建工厂对象

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

//3.通过工厂获取一个SqlSession对象

SqlSession session = factory.openSession();

//4.执行新增操作, 往emp表中添加一个新员工

session.insert("com.tedu.pojo.EmpMapper.**insert**");

//>>提交事务

session.commit();

//5.打印结果

System.***out***.println("执行完成!");

}

## 修改员工

**1、编辑EmpMapper.xml文件, 添加新增员工对应的sql.**

<!-- 3.修改员工信息 -->

<update id=*"****update****"*>

update emp set salary=6500 where name='韩少云'

</update>

**2、编写TestMybatis类，添加**updateEmp**方法，实现修改员工信息。**

/\*\*

\* 3.修改：修改emp表中的员工信息

\* **@throws** Exception

\*/

@Test

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

//1.读取sqlMapConfig.xml文件, 获取所有配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//2.创建工厂对象, 用于创建SqlSession对象

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

//3.获取一个SqlSession对象

SqlSession session = factory.openSession();

//4.执行更新操作

session.update("com.tedu.pojo.EmpMapper.**update**");

//>>提交事务

session.commit();

//5.打印结果

System.***out***.println("执行完成!");

}

## 删除员工

**1、编辑EmpMapper.xml文件, 添加新增员工对应的sql**.

<!-- 4.删除员工信息 -->

<update id=*"****delete****"*>

delete from emp where name='韩少云'

</update>

**2、编写TestMybatis类，添加**deleteEmp**方法，实现删除员工。**

/\*\*

\* 4.：删除emp表中的一条员工信息

\* **@throws** Exception

\*/

@Test

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

//1.读取sqlMapConfig.xml文件, 获取所有配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//2.创建工厂对象, 用于创建SqlSession对象

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

//3.获取一个SqlSession对象

SqlSession session = factory.openSession();

//4.执行删除操作

session.update("com.tedu.pojo.EmpMapper.**delete**");

//>>提交事务

session.commit();

//5.打印结果

System.***out***.println("执行完成!");

}

## 查询员工

**1、编辑EmpMapper.xml文件, 添加查询员工对应的sql**.

<!-- 5.查询指定id员工信息 -->

<update id=*"****findById****"*>

select \* from emp where id=2

</update>

**2、编写TestMybatis类，添加**findById**方法，实现根据id查询员工信息。**

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+Id）

Emp emp = session.selectOne("com.tedu.pojo.EmpMapper.**findById**");

//4.输出结果

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

}

## #{}占位符

在上面的增删改查操作中，SQL语句中的值是写死在SQL中，而在实际开发中，此处的值往往是用户提交过来的值，因此这里我们需要将SQL中写死的值替换为占位符。

使用占位符完成上面的增删改查练习

**1、新增：往emp表中添加一个新员工**

mapper文件配置:

<!-- 6.新增员工信息 -->

<insert id=*"****insert2****"*>

insert into emp(name,job,salary) values(#{name}, #{job}, #{salary})

</insert>

java代码示例:

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

//>>准备参数

Emp emp = **new** Emp();

emp.setName("马云");

emp.setJob("阿里CEO");

emp.setSalary(10000.0);

//>>执行新增

session.insert("com.tedu.pojo.EmpMapper.**insert2**",emp);

//4.提交事务

session.commit();

System.***out***.println("执行完成！");

}

**2、查询：查询指定id的员工信息**

mapper文件配置:

<!-- 7.查询Emp表中指定id的员工信息 -->

<mapper namespace=*"com.tedu.pojo.EmpMapper"*>

<select id=*"****findById2****"* resultType=*"com.tedu.pojo.Emp"*>

select \* from emp where id=#{id}

</select>

...

java代码示例:

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+Id）

Emp emp = session.selectOne("com.tedu.pojo.EmpMapper.**findById2**",1);

//4.输出结果

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

}

**3、修改：修改emp表中指定id的员工信息**

mapper文件配置:

<mapper namespace="com.tedu.pojo.EmpMapper">

<!-- 8.修改：修改emp表中指定id的员工信息 -->

<update id="**update2**" parameterType="com.tedu.pojo.Emp">

update emp set name=#{name},job=#{job},salary=#{salary}

where id=#{id}

</update>

...

java代码示例:

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

//>>准备参数

Emp emp = **new** Emp();

emp.setId(1);

emp.setName("王五");

emp.setJob("架构师");

emp.setSalary(8000.0);

//>>执行修改

session.update("com.tedu.pojo.EmpMapper.**update2**",emp);

//>>提交事务

session.commit();

System.***out***.println("执行完成！");

}

**4、删除：删除指定name的员工信息**

mapper文件配置:

<!-- 9.删除：删除emp表指定name的员工信息 -->

<insert id=*"****delete2****"* parameterType=*"String"*>

delete from emp

where name=#{name}

</insert>

java代码示例:

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

//>>执行删除

session.delete("com.tedu.pojo.EmpMapper.**delete2**", "马云");

//>>提交事务

session.commit();

System.***out***.println("执行完成！");

}

# MyBatis扩展(了解)

## #{}和${}

在上面的增删改查练习中，当SQL语句中包含的参数值是传递过来的，在SQL语句中我们会通过 **#{}** 占位符进行占位，在SQL语句真正执行时，再将传递过来的值放在SQL语句指定的位置，和SQL语句一起执行！

其实，**#{}** 就是JDBC中的问号（**?**）占位符，因此为了安全考虑，在执行时会对传递过来的值进行转译处理。

例如：查询指定name的员工信息，SQL语句为：

select \* from emp where name=#{name}

其实就等价于JDBC中： select \* from emp where name=?，如果传过来的参数值为：王海涛，那么最终执行的SQL语句为：

select \* from emp where name=**'**王海涛**'**

那么如果我们在传递的时候不是一个参数值，而是SQL语句本身呢？

例如在查询时，我们想动态的传递查询的列：

select **查询的列??** from emp

此时传递过来的应该是一个SQL片段，不同于上面的参数值，如果此时还用 #{}，也会像上面一样被转译处理，这不是我们希望看到的。

如果不想让传过来的值被转译处理，那么这里可以使用 **${}**，例如：

select **${columns}** from emp

**示例:查询emp表中所有员工的名称(name)、职位(job):**

mapper文件配置:

<!-- 10.查询：查询Emp表中所有员工的名称,职位 -->

<select id=*"findAll2"* resultType=*"com.tedu.pojo.Emp"*>

select ${**cols**} from emp

</select>

java代码示例:

**public** **void** testFindAll2() **throws** IOException{

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

Map map = **new** HashMap();

map.put("**cols**", "name, job");

List<Emp> list = session.selectList(

"com.tedu.pojo.EmpMapper.findAll2",map);

//4.输出结果

**for**(Emp emp : list){

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

}

}

需要注意的是，在传递 ${}对应的值时，需要将值存入**map**集合中！！

# 动态SQL

## if元素

**示例：查询emp表中的所有员工信息，如果传递了name参数，则根据name进行模糊查询，否则将查询所有员工信息**

mapper文件配置:

<!-- 11.查询：查询emp表中的所有员工信息，如果传递了name参数，则根据name进行模糊查询，否则将查询所有员工信息 -->

<select id=*"****findAll3****"* resultType=*"com.tedu.pojo.Emp"*>

select \* from emp

where 1=1

**<if test=*"name != null"*>**

**and name like '%${name}%'**

**</if>**

</select>

java代码示例: 查询name中包含"刘"字的所有员工

**public** **void** testFindAll3() **throws** IOException{

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

Map map = **new** HashMap();

map.put("name", "刘");**//查询name中包含"刘"字的所有员工**

List<Emp> list = session.selectList(

"com.tedu.pojo.EmpMapper.**findAll3**",map);

//4.输出结果

**for**(Emp emp : list){

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

}

}

示例：或者不传递name参数，则查询所有员工薪资

**public** **void** testFindAll3() **throws** IOException{

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

**//不传递name参数，则查询所有员工薪资**

//Map map = new HashMap();

//map.put("name", "刘");

List<Emp> list = session.selectList(

"com.tedu.pojo.EmpMapper.**findAll3**");

//4.输出结果

**for**(Emp emp : list){

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

}

}

## where元素



示例：查询emp表中所有员工的信息，另：

如果传递了minSal(最低薪资)和maxSal(最高薪资)，则查询薪资大于minSal和小于maxSal的员工信息；

如果只传递了minSal，则查询薪资大于minSal的所有员工信息；

如果只传递了maxSal，则查询薪资小于maxSal的所有员工信息**；**

mapper文件配置:

<!-- 12.查询emp表中所有员工的信息，另：

如果传递了minSal(最低薪资)和maxSal(最高薪资)，则查询薪资大于minSal和小于maxSal的员工信息；

如果只传递了minSal，则查询薪资大于minSal的所有员工信息；

如果只传递了maxSal，则查询薪资小于maxSal的所有员工信息；

-->

<select id=*"****findAll4****"* resultType=*"com.tedu.pojo.Emp"*>

select \* from emp

<where>

**<if test=*"minSal != null"*>**

**salary > #{minSal}**

**</if>**

**<if test=*"maxSal != null"*>**

**and salary <![CDATA[ < ]]> #{maxSal}**

**</if>**

</where>

</select>

java代码示例：查询薪资大于3000和小于4000的员工

**public** **void** testFindAll4() **throws** IOException{

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

Map map = **new** HashMap();

**//查询薪资大于3000和小于4000的员工**

**map.put("minSal", 3000);**

**map.put("maxSal", 4000);**

List<Emp> list = session.selectList(

"com.tedu.pojo.EmpMapper.**findAll4**",map);

//4.输出结果

**for**(Emp emp : list){

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

}

}

java代码示例：查询薪资大于3000的员工

**public** **void** testFindAll4() **throws** IOException{

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

Map map = **new** HashMap();

**//查询薪资大于3000的员工**

**map.put("minSal", 3000);**

**//map.put("maxSal", 4000);**

List<Emp> list = session.selectList(

"com.tedu.pojo.EmpMapper.**findAll4**",map);

//4.输出结果

**for**(Emp emp : list){

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

}

}

java代码示例：查询薪资小于4000的员工

**public** **void** testFindAll4() **throws** IOException{

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

Map map = **new** HashMap();

**//查询薪资小于4000的员工**

**//map.put("minSal", 3000);**

**map.put("maxSal", 4000);**

List<Emp> list = session.selectList(

"com.tedu.pojo.EmpMapper.**findAll4**",**map**);

//4.输出结果

**for**(Emp emp : list){

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

}

}

## set元素

示例：修改emp表中指定id的员工信息，如果传递了name、job、salary列的值，则修改，否则不修改

mapper文件配置:

<!-- 13.修改emp表中指定id的员工信息，如果传递了name、job、salary列的值，则修改，否则不修改

-->

<update id="**updateEmpById**">

update emp

**<set>**

**<if test="name != null">name=#{name},</if>**

**<if test="job != null">job=#{job},</if>**

**<if test="salary != null">salary=#{salary}</if>**

**</set>**

where id=#{id}

</update>

java代码示例：

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

**Emp emp = new Emp();**

**emp.setId(1);**

**emp.setName("王海涛xx");**

**emp.setJob("金牌讲师");**

**emp.setSalary(4999.0);**

//4.执行更新

session.update("com.tedu.pojo.EmpMapper.**updateEmpById**",emp);

//>>提交事务

session.commit();

System.***out***.println("执行完成！");

}

可以尝试不传递name或job或salary中的某些值，运行程序，查询修改后的结果！！

## foreach元素

示例: 查询emp表中所有指定id的员工信息

mapper文件配置:

<!-- 14.查询emp表中所有指定id的员工信息 -->

<select id=*"****findByIds****"* resultType=*"com.tedu.pojo.Emp"*>

select \* from emp where id in

<foreach collection=*"array"* open=*"("* close=*")"* item=*"id"* separator=*","*>

#{**id**}

</foreach>

</select>

Java代码示例: 查询指定id的所有员工信息

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

**//查询指定id的所有员工信息**

**int[] ids = {1,3,5};**

//4.执行更新

List<Emp> list = session.selectList("com.tedu.pojo.EmpMapper.**findByIds**",**ids**);

//5.输出结果

**for** (Emp e : list) {

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

}

}

批量删除示例： 删除emp表中所有指定id的员工信息

mapper文件配置:

<!-- 15.批量删除emp表中所有指定id的员工信息 -->

<delete id=*"****deleteByIds****"*>

delete from emp where id in

<foreach collection=*"array"* open=*"("* close=*")"* item=*"id"* separator=*","*>

#{id}

</foreach>

</delete>

Java代码示例: 删除定id的所有员工信息

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.执行SQL语句（namespace+SQLId）

**//删除指定id的员工信息**

**int[] ids = {1,3,5};**

//4.执行更新

session.delete("com.tedu.pojo.EmpMapper.**deleteByIds**",**ids**);

//>>提交事务

session.commit();

System.***out***.println("执行完成!");

}

# Mapper接口开发

## Mapper接口开发介绍

在上面的Mybatis案例中, 通过SqlSession对象调用方法进行增删改查操作时, 方法中需要传入的第一个参数是一个字符串值, 该值对应的内容为: (Mapper文件中的) namespace + id, 通过这种方式, 找到Mapper文件中映射的SQL语句并执行!!

这种方式由于传入的是字符串值, 很容易发生字符串拼写错误且编译时期不会提示。

这里我们将会讲解比上面更加简单的方式，也是我们企业开发中最常用的方式，即使用mapper接口开发。使用mapper接口开发需要注意以下几点：

1、Mapper接口的全路径名+方法名，和mapper.xml文件中定义namespace+id值必须相同

2、Mapper接口中方法接收的参数类型，和mapper.xml中定义的sql的parameterType的类型相同

3、Mapper接口中方法的返回值类型，和mapper.xml中定义的sql的resultType的类型相同

## Mapper接口开发实现

下面将使用mapper接口开发的方式，实现根据id查询指定的员工信息

**1、创建com.tedu.dao.EmpMapper接口**

由于接口的全路径名(com.tedu.dao.EmpMapper)要和EmpMapper.xml的namespace值保持一致, 因此, 这里将namespace的值改为com.tedu.dao.EmpMapper：

<?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.tedu.dao.EmpMapper****"*>

</mapper>

**2、在接口中提供findById方法**

/\*\*

\* 根据id查询员工信息

\* **@param** id

\* **@return** Emp

\*/

**public** Emp **findById**(Integer id);

注意:方法的名字要和映射的sql标签的id值保持一致

方法的返回值类型和resultType的类型要一致

<!-- 1.查询：查询Emp表中指定id的员工信息 -->

<select id=*"****findById****"* resultType=*"com.tedu.pojo.Emp"*>

select \* from emp where id=#{id}

</select>

**3、提供实现类，测试Emp接口中的根据id查询员工的方法**

(1)创建com.tedu.test.TestMybatisInf类, 并提供testFindById方法

**public** **class** TestMybatisInf {

@Test

**public** **void** testFindById() **throws** Exception{}

}

**实现testFindById方法并测试**

@Test

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.获取Mapper接口对象

EmpMapper map = session.getMapper(EmpMapper.**class**);

//4.调用接口对象的方法进行查询

Emp e = map.findById(2);

//5.输出结果

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

}

**5、在接口中提供findAll方法**

/\*\*

\* 查询所有的员工信息

\* **@return** List<Emp>

\*/

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

注意:方法的名字要和映射的sql标签的id值保持一致

方法的返回值类型和resultType的类型要一致, 例如:

<!-- 2.查询Emp表中所有员工的信息 -->

<select id=*"****findAll****"* resultType=*"com.tedu.pojo.Emp"*>

select \* from emp

</select>

**6、提供实现类，测试Emp接口中的查询所有员工的方法**

(1)创建com.tedu.test.TestMybatisInf类, 并提供testFindAll方法

**public** **class** TestMybatisInf {

。。。

@Test

**public** **void** testFindAll () **throws** Exception{}

}

**实现**testFindAll**方法并测试**

@Test

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

//0.加载sqlMapConfig.xml文件，获取所有的配置信息

InputStream in = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//1.创建工厂对象

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

//2.通过工厂创建一个SqlSession对象

SqlSession session = factory.openSession();

//3.获取Mapper接口对象

EmpMapper map = session.getMapper(EmpMapper.**class**);

//4.调用接口对象的方法进行查询

List<Emp> list = map.findAll();

//5.输出结果

**for** (Emp e : list) {

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

}

}

# 几个可以优化的地方

## 加入log4j日志框架

在项目中加入log4j的配置文件，用于打印日志信息，便于开发调试。

在src（或相似的目录）下创建log4j.properties如下：

# Global logging configuration

log4j.rootLogger=DEBUG, stdout

# Console output...

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

mybatis默认使用log4j作为输出日志信息。

只要将该文件放在指定的位置，log4j工具会自动到指定位置加载上述文件，读取文件中的配置信息并使用！

## 可重复使用的SQL片段

在编写查询SQL时，我们书写的SQL往往为：

select **\*** from emp

select **\*** from emp where id=#{id}

...

其实这里书写星号（\*）不如直接写列名速度快，因为在查询时，还需要将星号（\*）转换为表列名，如果直接写列名则省去了转换操作。

因此下面在查询指定id的员工和查询所有员工的SQL中，将查询的列由星号（\*）替换为了列名。

<!-- 0.查询指定id的员工 -->

<select id="findById" parameterType="int" resultType="com.tedu.pojo.Emp">

select id,name,job,salary from emp

where id=#{id}

</select>

<!-- 1.查询所有员工 -->

<select id="findAll" resultType="com.tedu.pojo.Emp">

select id,name,job,salary from emp

</select>

但，如果直接写列名替换星号（\*），可能也会带来一些麻烦：

(1)假设这里还有很多查询，都要查询所有列，每次都书写所有列名，麻烦！

(2)如果表中的列有几十个，书写起来会更加麻烦，SQL语句的长度也会暴增！

(3)如果后期需求发生变化，要修改查询的列，可能要修改多处，可维护性差！

要解决上面的问题，可以使用sql片段，例如：

声明一个sql片段，指定要查询的列

<!-- 声明一个SQL片段，指定要查询的列 -->

**<sql id=*"empCols"*>id,name,job,salary</sql>**

将下面查询SQL中的列名替换为SQL片段

<!-- 0.查询指定id的员工 -->

<select id=*"findById"* parameterType=*"int"* resultType=*"com.tedu.pojo.Emp"*>

select **<include refid=*"empCols"*/>** from emp

where id=#{id}

</select>

<!-- 1.查询所有员工 -->

<select id=*"findAll"* resultType=*"com.tedu.pojo.Emp"*>

select **<include refid=*"empCols"*/>** from emp

</select>

## SQL语句中的特殊符号

**示例：添加一个查询功能：查询薪资小于3500的所有员工。**

1、编辑EmpMapper.xml文件, 添加查询对应的sql.

<!-- 查询薪资小于3500的所有员工 -->

<select id=*"findBySal"*>

select <include refid=*"empCols"*/> from emp

where salary < 3500

</select>

2、但在书写完后，xml文件提示有错误：



原来，小于号（<）在xml文件中是特殊字符，被xml文件当成了标签的开始符号。

3、解决方法：将特殊符号包含在CDATA区（）中，这是因为放在CDATA区中的内容，只会被xml解析器当作普通文本来处理。而不是被当成标签的一部分处理。

<!-- 查询薪资小于3500的所有员工 -->

<select id=*"findBySal"*>

select <include refid=*"empCols"*/> from emp

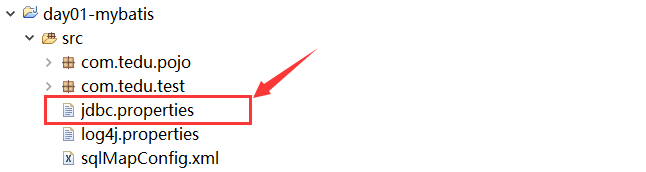
where salary <![CDATA[ **<** ]]> 3500

</select>

## jdbc.properties文件

在开发中，通常我们会将连接数据库的配置信息单独放在一个properties文件中（方便管理和维护）， 然后在MyBatis的mapper文件中引入properties文件的配置信息即可！

**1、在src目录下创建一个名称为jdbc.properties的文件**



**2、jdbc.properties文件内容如下:**

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8

jdbc.username=root

jdbc.password=root

**3、在sqlMapConfig.xml文件中引入jdbc.properties文件**



1、其中 <properties resource="jdbc.properties"/> 标签用于引入jdbc.properties文件，默认到classpath即类目录下寻找指定的文件；

2、properties标签上value属性中配置的 ${jdbc.xxx}：

${jdbc.driver}：其实就是jdbc.properties文件中的 jdbc.driver的值，即：

com.mysql.jdbc.Driver

${jdbc.url}：其实就是jdbc.properties文件中的 jdbc.url的值，即：

jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8

${jdbc.username}：其实就是jdbc.properties文件中的 jdbc.username的值，即：

root

${jdbc.password}：其实就是jdbc.properties文件中的 jdbc.password的值，即：

root

# 扩展内容

## Jdbc回顾

通过JDBC查询Emp表中的所有记录，并封装到一个List<Emp>集合中返回

1、创建TestJdbc类，完成查询所有员工：

**package** com.tedu;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** java.util.ArrayList;

**import** java.util.List;

**import** com.tedu.pojo.Emp;

/\*\* Jdbc回顾 \*/

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

**public** **static** **void** main(String[] args) {

/\* 查询emp表中的所有员工信息，将每个员工信息的封装到一个

\* Emp对象中，再将封装了员工信息所有Emp对象存入List集合

\* 中，并遍历输出所有的员工信息

\*/

List<Emp> empList = *findAll*();

**for**(Emp emp : empList){

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

}

}

/\*\*

\* 查询emp表中的所有员工信息,封装到List集合并返回

\*/

**private** **static** List<Emp> findAll() {

Connection conn = **null**;

Statement stat = **null**;

ResultSet rs = **null**;

**try** {

//1.注册数据库驱动

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

//2.获取数据库连接(Connection)

conn = DriverManager.*getConnection*(

"jdbc:mysql:///yonghedb",

"root", "root");

//3.获取传输器

stat = conn.createStatement();

//4.利用传输器发送sql到数据库执行,并返回执行结果

String sql = "select \* from emp";

rs = stat.executeQuery(sql);

//5.处理结果

//5.1.声明List集合,用于封装所有的员工信息

List<Emp> empList = **new** ArrayList();

//5.2.遍历ResultSet结果集

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

//5.3.获取结果集中的每一条员工信息

**int** id = rs.getInt("id");

String name = rs.getString("name");

String job = rs.getString("job");

**double** salary = rs.getDouble("salary");

//5.4.将每一条员工信息封装到一个Emp对象中

Emp emp = **new** Emp();

emp.setId(id);

emp.setName(name);

emp.setJob(job);

emp.setSalary(salary);

//5.5.将Emp对象存入List集合中

empList.add(emp);

}

**return** empList;

} **catch** (Exception e) {

e.printStackTrace();

System.***out***.println("查询失败!");

} **finally**{

//6.释放资源

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

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}**finally**{

rs = **null**;

}

}

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

**try** {

stat.close();

} **catch** (SQLException e) {

e.printStackTrace();

}**finally**{

stat = **null**;

}

}

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

**try** {

conn.close();

} **catch** (SQLException e) {

e.printStackTrace();

}**finally**{

conn = **null**;

}

}

}

**return** **null**;

}

}

2、声明Emp实体类，用于封装员工信息：

**package** com.tedu.pojo;

**public** **class** Emp {

//声明私有属性

**private** Integer id; //员工编号

**private** String name; //员工姓名

**private** String job; //员工职位

**private** **double** salary; //员工薪资

//提供getter和setter方法

**public** Integer getId() {

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** String getJob() {

**return** job;

}

**public** **void** setJob(String job) {

**this**.job = job;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

//提供toString方法

@Override

**public** String toString() {

**return** "Emp [id=" + id + ", name=" + name + ", job=" + job + ", salary=" + salary + "]";

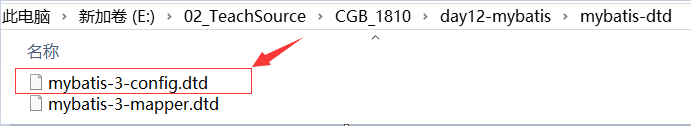
}

}

## sqlMapConfig文件没有提示的解决办法

如果在没有网络(外网)的情况下，编写sqlMapConfig.xml文件没有提示，可以按照下面的步骤进行配置：

(1)找到mybatis-3-config.dtd的文件的位置，例如：

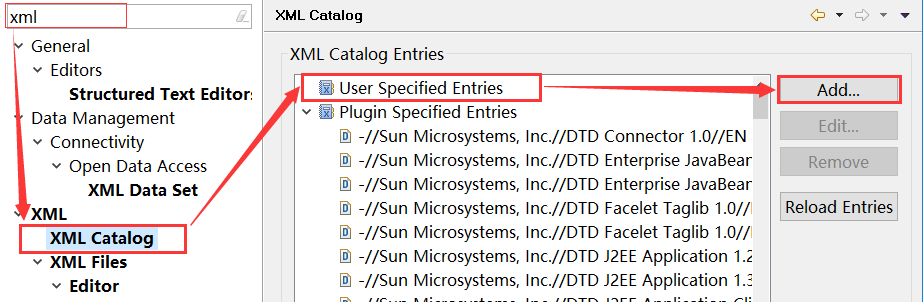


(2)复制下面的url地址：

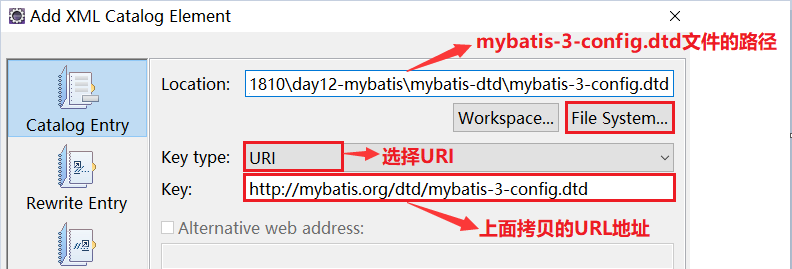
http://mybatis.org/dtd/mybatis-3-config.dtd

(3)在eclipse菜单栏中: window --> Preferences --> 在搜索框中搜索 [ xml ]

XML --> XML Catalog --> User Specified Entries --> Add…



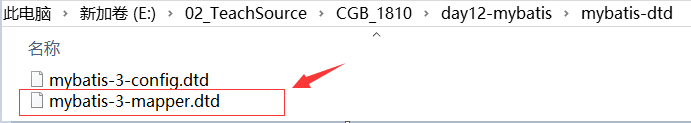
(4)在弹出的窗口中:



## Mapper文件没有提示的解决办法

如果在没有网络(外网)的情况下，编写XxxMapper.xml文件没有提示，可以按照下面的步骤进行配置：

(1)找到mybatis-3-mapper.dtd的文件的位置，例如：

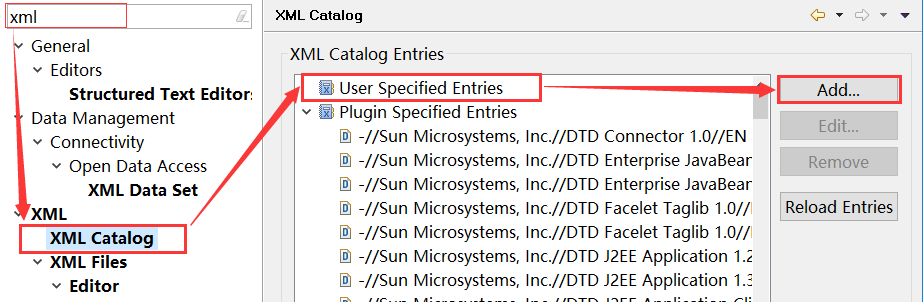


(2)复制上面的url地址，即：

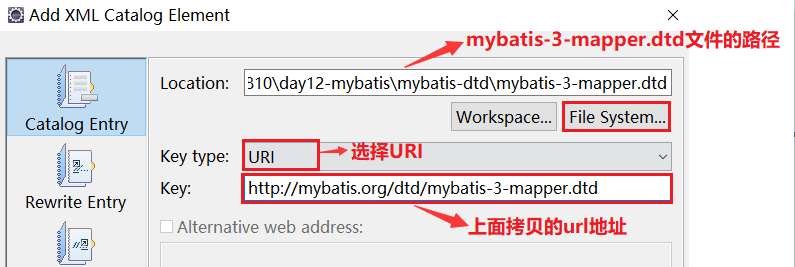
http://mybatis.org/dtd/mybatis-3-mapper.dtd

(3)在eclipse菜单栏中: window --> Preferences --> 在搜索框中搜索 [ xml ]

XML --> XML Catalog --> User Specified Entries --> Add…

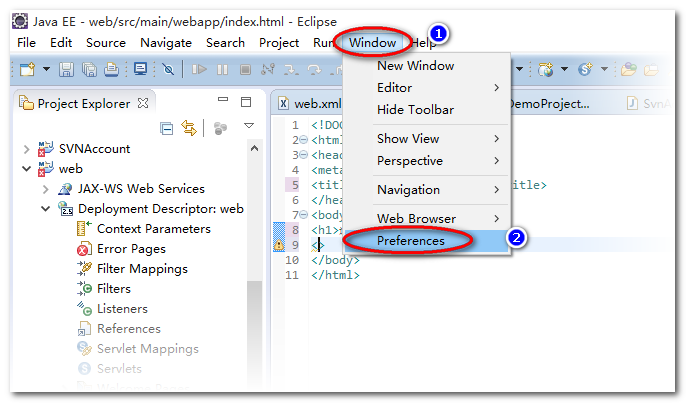


(4)在弹出的窗口中:

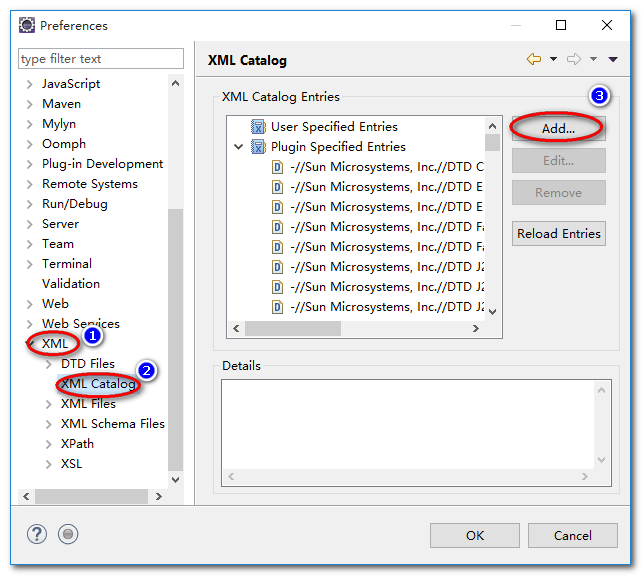


## 配置达内XML Schema代理服务器

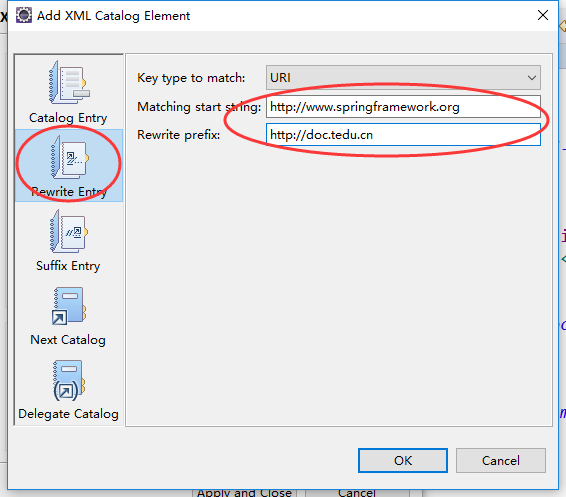
1、打开Eclipse的配置首选项



2、找到 XML -> XML Catalog, 添加配置项目：



3、添加URI代理配置, 比如：配置Spring Schema



**4、可以配置的代理有**

|  |  |  |
| --- | --- | --- |
|  | http://ibatis.apache.org | http://doc.tedu.cn |
|  | http://www.springframework.org | http://doc.tedu.cn |
|  | http://mybatis.org | http://doc.tedu.cn |
|  | http://hibernate.sourceforge.net, http://www.hibernate.org | http://doc.tedu.cn |
|  | http://struts.apache.org | http://doc.tedu.cn |