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

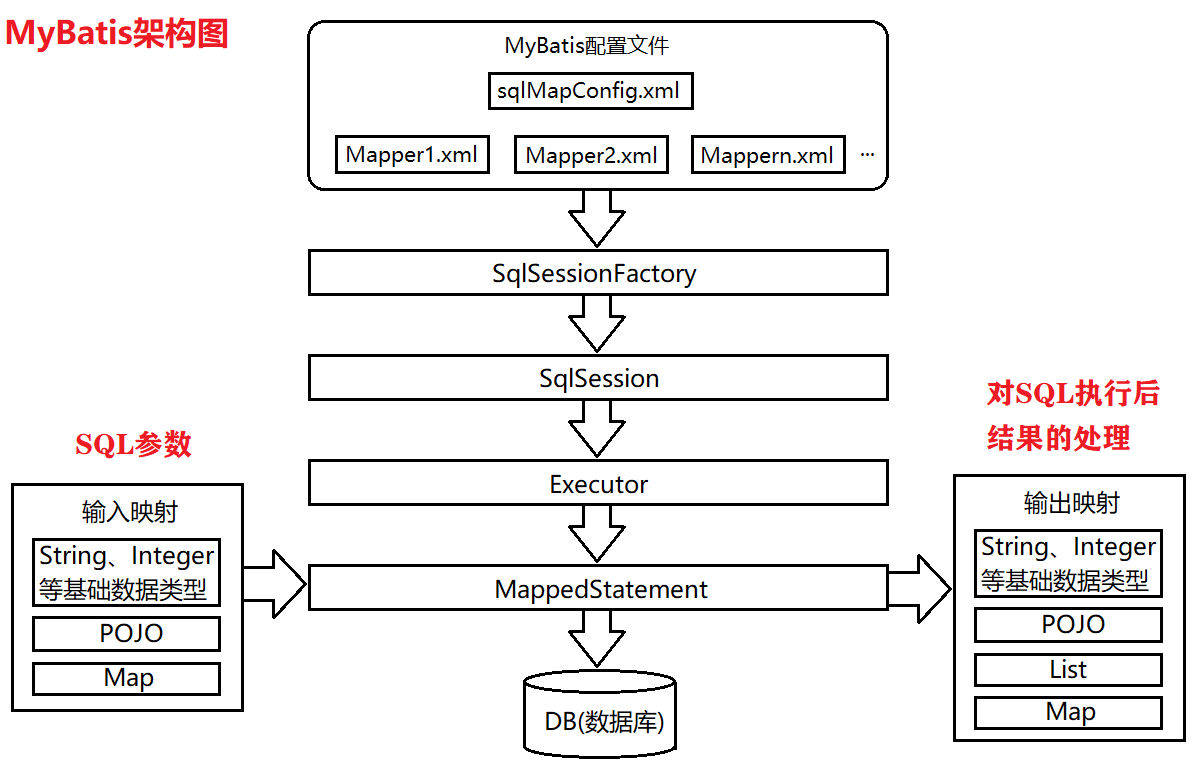
MyBatis 本是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github。

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)mybatis-config.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、创建数据库 yonghedb 数据库

create database if not exists yonghedb charset utf8;

use yonghedb; -- 选择yonghedb数据库

-- 2、删除emp表（如果存在）

drop table if exists emp;

-- 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', 2700);

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);

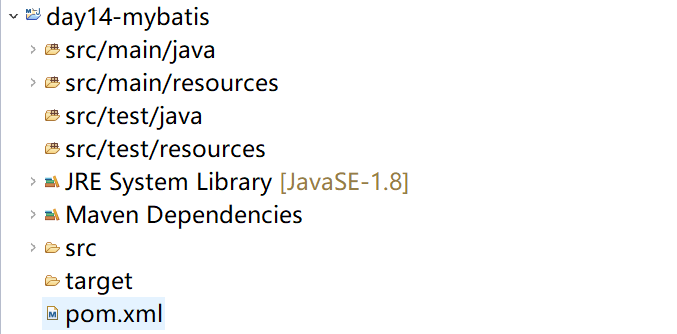
insert into emp values(null, '董长春', '程序员', 3500);

insert into emp values(null, '苍老师', '程序员', 3700);

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Maven的java工程



2、导入junit、mysql、mybaits等开发包

在pom.xml文件中引入相关依赖包即可

<dependencies>

<!-- junit单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.9</version>

</dependency>

<!-- mysql驱动 -->

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.32</version>

</dependency>

<!-- mybatis -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.8</version>

</dependency>

<!-- 整合log4j -->

<dependency>

<groupId>org.slf4j</groupId>

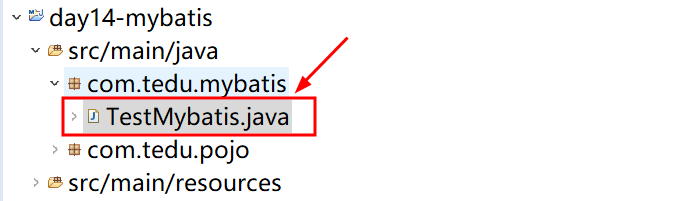
<artifactId>slf4j-log4j12</artifactId>

<version>1.6.4</version>

</dependency>

</dependencies>

3、创建com.tedu.mybatis.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：



/\*\* 练习1(快速入门): 查询emp表中的所有员工, 返回一个List<Emp>集合

\* **@throws** IOException \*/

@Test

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

//1.读取mybatis的核心配置文件(mybatis-config.xml)

//2.通过配置信息获取一个SqlSessionFactory工厂对象

//3.通过工厂获取一个SqlSession对象

//4.通过namespace+id找到要执行的sql语句并执行sql语句

//5.输出结果

}

## 添加mybatis-config.xml文件

1、在src/main/resources目录下，创建mybatis-config.xml文件 （MyBatis的核心配置文件）



**2、mybatis-config.xml文件配置如下:**

mybatis-config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=*"EmpMapper.xml"*/>

</mappers>

</configuration>

## 添加并编写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。

在src/main/java目录下创建com.tedu.pojo.Emp类并实现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() {

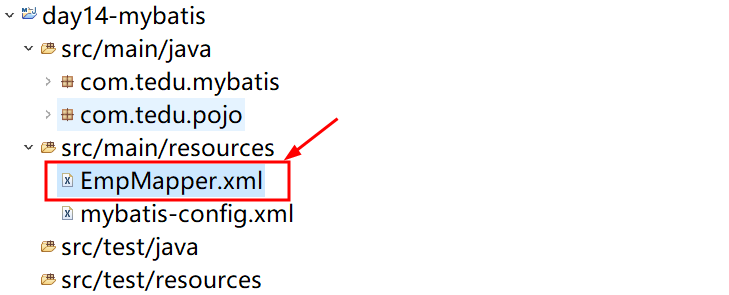
**return** "Emp [id=" + id + ", name=" + name + ", job=" + job + ", salary=" + salary + "]";

}

}

## 添加EmpMapper.xml文件

1、在src/main/resources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=*"****EmpMapper****"*>

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

<!-- 练习1: 查询emp表中的所有员工信息

resultType指定查询的结果将会封装到什么类型中 -->

<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！

-->

</mapper>

## 实现测试类,并测试

1、实现findAll方法，代码如下：

/\*\* 练习1(快速入门): 查询emp表中的所有员工, 返回一个List<Emp>集合

\* **@throws** IOException \*/

@Test

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

//1.读取mybatis的核心配置文件(mybatis-config.xml)

InputStream in = Resources

.*getResourceAsStream*("mybatis-config.xml");

//2.通过配置信息获取一个SqlSessionFactory工厂对象

SqlSessionFactory fac =

**new** SqlSessionFactoryBuilder().build( in );

//3.通过工厂获取一个SqlSession对象

SqlSession session = fac.openSession();

//4.通过namespace+id找到要执行的sql语句并执行sql语句

List<Emp> list = session

.selectList("EmpMapper.findAll");

//5.输出结果

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

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

}

}

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=2700.0]

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=苍老师, job=程序员, salary=3700.0]

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

# MyBatis入门细节

## mybatis-config.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/yonghedb?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=*"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！

**(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=*"****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.**

<!-- 练习2: 新增员工信息: 刘德华 歌手 888

增删改的标签上不用指定resultType, 因为返回值都是int类型

-->

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

insert into emp value(null, '刘德华', '歌手', 888)

</update>

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

/\*\* 练习2: 新增员工信息: 刘德华 歌手 888 \*/

@Test

**public** **void** testInsert() {

//执行sql语句, 返回执行结果

**int** rows = *session*.update("EmpMapper.***insert***");

//提交事务

*session*.commit();

System.***out***.println("影响的行数: "+rows);

}

## 修改员工

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

<!-- 练习3：修改员工信息 -->

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

update emp set job='歌手&演员', salary=8888 where name='刘德华'

</update>

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

/\*\* 练习3: 修改员工信息, 将刘德华的job改为'歌手&演员'

\* salary改为88888 \*/

@Test

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

//执行sql语句, 返回执行结果

int rows = session.update("EmpMapper.***update***");

//提交事务

session.commit();

System.***out***.println("影响行数:"+rows);

}

## 删除员工

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

<!-- 练习4: 删除name为'刘德华'的记录 -->

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

delete from emp where name='刘德华'

</update>

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

/\*\* 练习4: 删除name为'刘德华'的记录 \*/

@Test

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

//执行sql语句, 返回执行结果

int rows = session.update("EmpMapper.**delete**");

//提交事务

session.commit();

System.***out***.println("影响行数:"+rows);

}

## #{}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

**练习5、查询：查询指定id的员工信息**

mapper文件配置:

<!-- 练习5: 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>

java代码示例:

/\*\* 练习5: 查询emp表中指定id的员工信息 \*/

@Test

**public** **void** testFindById() {

//执行sql语句, 返回执行结果

Emp emp = *session*.selectOne(

"EmpMapper.findById", 1 );

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

}

**练习6：往emp表中添加一个新员工**

mapper文件配置:

<!-- 练习6: 新增员工信息: 张飞 Java开发工程师 15000

如果通过map集合传输参数, 需要保证占位符中的变量名

和map集合中的key保持一致

如果通过pojo对象传输参数, 需要保证占位符中的变量名

和对象中的属性名保持一致, 或者在pojo中有对应的

getXxx方法

-->

<update id=*"insert2"*>

insert into emp values

(null, #{name}, #{job}, #{salary})

</update>

java代码示例:

/\*\* 练习6: 新增员工信息: 张飞 Java开发工程师 15000 \*/

@Test

**public** **void** testInsert2() {

//将要传输的参数封装到map集合中

//Map map = new HashMap();

//map.put("name", "张飞");

//map.put("job", "Java开发工程师");

//map.put("salary", 15000);

//也可以将要传输的参数封装到Emp对象中

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

emp.setName("关羽123");

emp.setJob("保安");

emp.setSalary(8000.0);

//执行sql语句

**int** rows = *session*.update("EmpMapper.insert2", emp);

//提交事务

*session*.commit();

System.***out***.println( "影响的行数: "+rows );

}

**练习7：修改员工信息: 张飞 架构师 25000**

mapper文件配置:

<!-- 练习7: 修改员工信息: 张飞 架构师 25000 -->

<update id=*"update2"*>

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

where name=#{name}

</update>

java代码示例:

/\*\* 练习7: 修改员工信息: 张飞 架构师 25000 \*/

@Test

**public** **void** testUpdate2() {

//将参数封装到Emp对象中

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

emp.setName("张飞");

emp.setJob("架构师");

emp.setSalary(25000.0);

//执行sql语句

**int** rows = *session*.update("EmpMapper.update2", emp);

//提交事务

*session*.commit();

System.***out***.println("影响的行数: "+rows);

}

**练习8：删除指定id的员工信息**

mapper文件配置:

<!-- 练习8：删除emp表中指定id的员工信息 -->

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

delete from emp where id=#{id}

</insert>

java代码示例:

/\*

\* 练习8：删除emp表中指定id的员工信息

\*/

**public** **void** testDelete2() **throws** IOException{

......

//执行SQL语句

int rows = session.delete("EmpMapper.**delete2**", 1);

//提交事务

session.commit();

System.***out***.println("影响行数:"+rows);

}

## #{}和${}占位符

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，再将传递过来的值替换SQL语句中的占位符。

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，而是一个SQL片段呢？

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SQL片段被转译处理，而是拼接在SQL语句中，那么这里可以使用 **${}**，例如：

select **${columns}** from emp

**示例1:查询emp表中所有员工的名称(name)、职位(job):**

mapper文件配置:

<!-- 练习9: 动态显示要查询的列 -->

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

select ${cols} from emp

</select>

java代码示例:

/\*\* 练习9: 动态显示要查询的列

\* select \* from emp

\* select id, name from emp

\* select id, salary from emp

\* select name, job from emp \*/

@Test

**public** **void** testFindAll2() {

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

//map.put("cols", "id, name");

//map.put("cols", "id, name, salary");

map.put("cols", "id,name,job,salary");

//执行sql语句, 返回结果

List<Emp> list = *session*.selectList(

"EmpMapper.**findAll2**", map);

//输出结果

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

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

}

}

**示例2:** **根据name模糊查询emp表**

mapper文件配置:

<!-- 练习10: 根据name模糊查询emp表 -->

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

select \* from emp

where name like '%${name}%'

</select>

<!-- 练习11: 根据name模糊查询emp表 -->

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

select \* from emp

where name like #{name}

</select>

java代码示例:

/\*\*

\* 练习10: 根据name模糊查询emp表

\* '%王%' '%刘%'

\*/

@Test

**public** **void** testFindAll3() {

//将参数封装到map集合中

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

map.put("name", "涛");

//执行sql, 返回结果

List<Emp> list = *session*.selectList(

"EmpMapper.**findAll3**", map);

//输出结果

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

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

}

}

/\*\*

\* 练习11: 根据name模糊查询emp表

\* '%王%' '%刘%'

\*/

@Test

**public** **void** testFindAll4() {

//将参数封装到map集合中

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

map.put("name", "%刘%");

//执行sql, 返回结果

List<Emp> list = *session*.selectList(

"EmpMapper.**findAll4**", map);

//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！！

总结：在大多数情况下还是使用#{}占位符，而${}多用于为不带引号的字符串进行占位！

# 动态SQL

## if、where元素

(1)mybatis中的if元素用于对某一字段进行判断，比如根据判断传过来的参数是否为空，从而决定是否执行包含在其中的SQL片段。

(2)where元素则用于对包含在其中的SQL语句进行检索，需要时会剔除多余的连接词（比如and或者or），并且在需要时可以添加where关键词



**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:

<!-- 练习12.1: 根据薪资查询员工信息

\* 练习12: 根据薪资查询员工信息

\* 如果参数有 minSal、maxSal, 则

\* ... where salary > minSal and salary < maxSal

\* 如果参数中只有minSal, 则

\* ... where salary > minSal

\* 如果参数中只有maxSal, 则

\* ... where salary < maxSal

\* 如果没有参数, 则不执行where子句, 默认查询所有员工 -->

<select id=*"findAllBySal"*

resultType=*"com.tedu.pojo.Emp"*>

select \* from emp

where 1=1

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

and salary>#{minSal}

</if>

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

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

</if>

</select>

<!-- 练习12.2: 根据薪资查询员工信息 -->

<select id=*"findAllBySal2"*

resultType=*"com.tedu.pojo.Emp"*>

select \* from emp

<where>

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

and 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5000的员工

/\*----------- 动态SQL -----------

\* if标签: 判断若为true, 则执行if中的sql片段

\* where标签: 生成where关键字, 对其中多余的连接词

\* 进行处理

\* set标签: 生产set关键字, 对其中多余的连接符进行处理

\* set 列1=值1, 列2=值2

\* foreach标签: 进行循环遍历, 比如实现批量删除

\*/

/\*\*

\* 练习12: 根据薪资查询员工信息

\* 如果参数有 minSal、maxSal, 则

\* ... where salary > minSal and salary < maxSal

\* 如果参数中只有minSal, 则

\* ... where salary > minSal

\* 如果参数中只有maxSal, 则

\* ... where salary < maxSal

\* 如果没有参数, 则不执行where子句, 默认查询所有员工 \*/

@Test

**public** **void** testFindAllBySal() {

//封装参数到map集合中

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

map.put("minSal", 3000);

map.put("maxSal", 5000.0);

List<Emp> list = *session*.selectList(

"EmpMapper.findAllBySal2", map);

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

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

}

}

## set元素

mybatis中的set元素用于对包含在其中的SQL语句进行检索，在需要时可以剔除其中多余的连接符（比如逗号），并且在需要时可以添加set关键词。

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:

<!-- 练习13: 根据参数传输与否,修改表中的列 -->

<update id=*"update3"*>

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：

/\*\* 练习13: 根据参数传输与否,修改表中的列

\* 如果传输了 name, job, salary

\* 则修改对应的列, 否则就不修改!

\*/

@Test

**public** **void** testUpdate3() {

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

emp.setId( 8 );

emp.setName("马云");

emp.setJob("教师");

emp.setSalary(2200d);

**int** rows = *session*.update(

"EmpMapper.update3", emp);

//提交事务

*session*.commit();

System.***out***.println("影响的行数: "+rows);

}

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

mapper文件配置:

<!-- 练习14: 根据员工的id批量删除员工信息

delete from emp

where id in (1,3,5,7)

-->

<update id=*"deleteByIds"*>

delete from emp where id in

<foreach collection=*"array"* open=*"("*

item=*"id"* separator=*","* close=*")"*>

#{id}

</foreach>

</update>

Java代码示例:

/\*\* 练习14: 根据员工的id批量删除员工信息 \*/

@Test

**public** **void** testDeleteByIds() {

Integer[] ids = { 1, 3, 5, 7 };

**int** rows = *session*.update(

"EmpMapper.deleteByIds", ids);

*session*.commit();

System.***out***.println("影响的行数: "+rows);

}

/\*\* 练习15(自己完成): 根据员工的id批量查询员工信息

\* Integer[] ids = [2,4,6,8]; \*/

@Test

**public** **void** testFindByIds() {

}

# 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、创建一个接口，**接口的全路径名**和mapper文件的**namespace值**要相同

2、mapper文件中**每条要执行的SQL语句**，在接口中要**添加一个对应的方法**，并且接口中的**方法名**和SQL标签上的**id值**相同

3、Mapper接口中方法接收的参数类型，和mapper.xml中定义的sql的接收的参数类型要相同

4、**接口中方法的返回值类型**和**SQL标签上的resultType即返回值类型**相同(如果方法返回值是集合,resultType只需要指定集合中的泛型)

## 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{

......

//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{

......

//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！

## 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

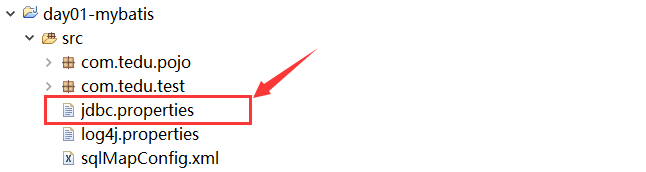
where salary <![CDATA[ **<** ]]> 3500

</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/yonghedb?characterEncoding=utf-8

jdbc.username=root

jdbc.password=root

**3、在mybatis-config.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.jdbc;

**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;

/\*\*

\* 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 + "]";

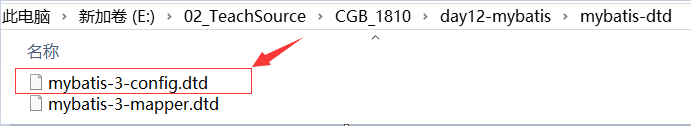
}

}

## mybatis-config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mybatis-config.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：

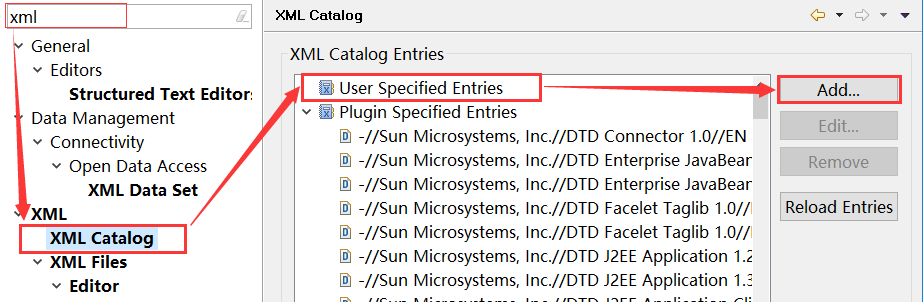


(2)复制下面的url地址：

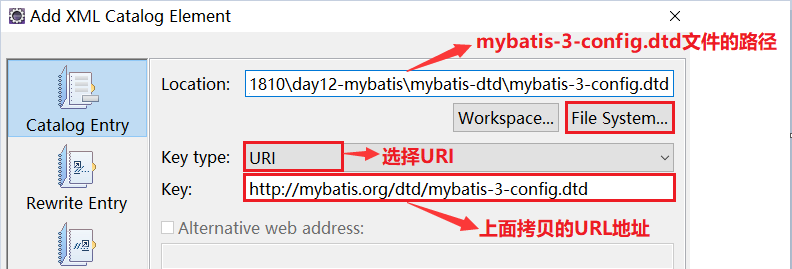
http://mybatis.org/dtd/mybatis-3-config.dtd

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

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



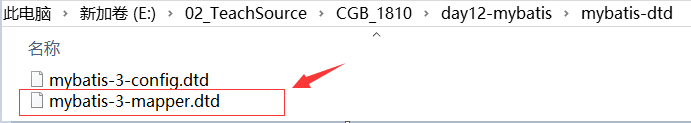
(4)在弹出的窗口中:



## 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：

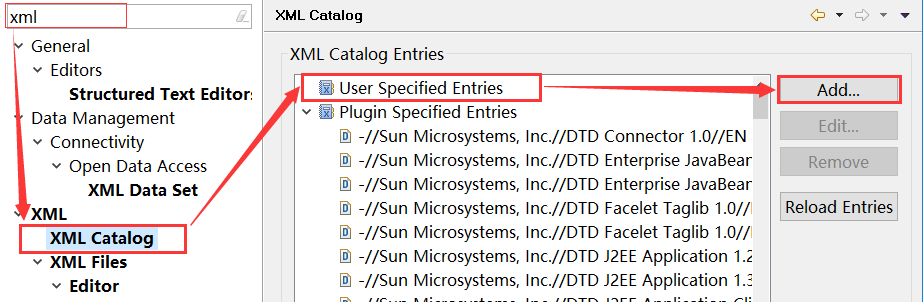


(2)复制上面的url地址，即：

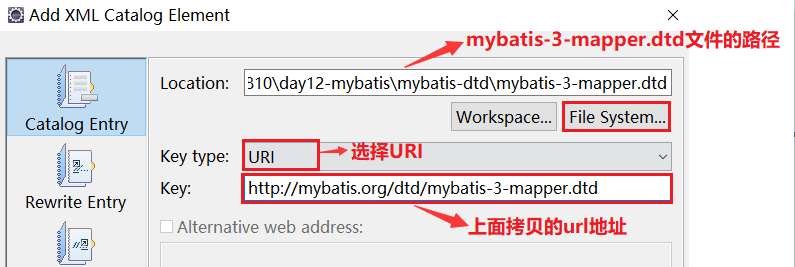
http://mybatis.org/dtd/mybatis-3-mapper.dtd

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

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

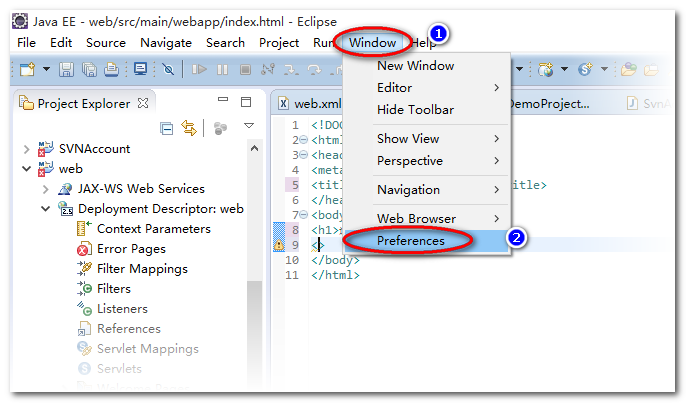


(4)在弹出的窗口中:

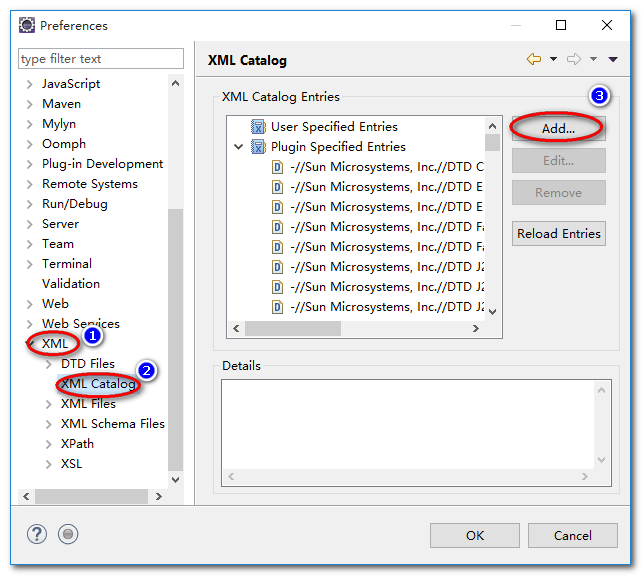


## 配置达内XML Schema代理服务器

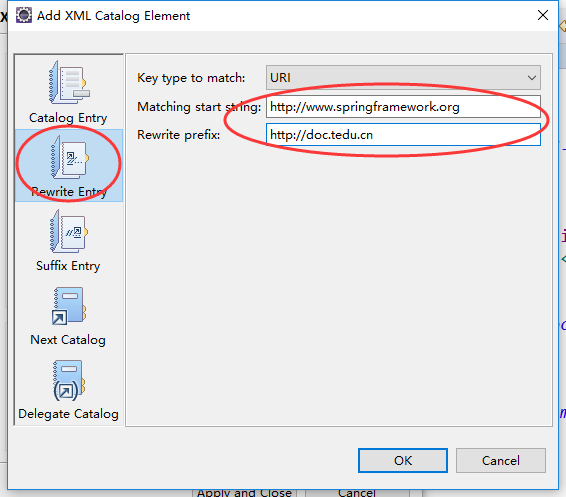
1、打开Eclipse的配置首选项



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 |