# 课程安排

1. 一天半时间学习mybatis
2. 一天时间学习springmvc
3. 半天时间做综合练习，mybatis+springmvc实现

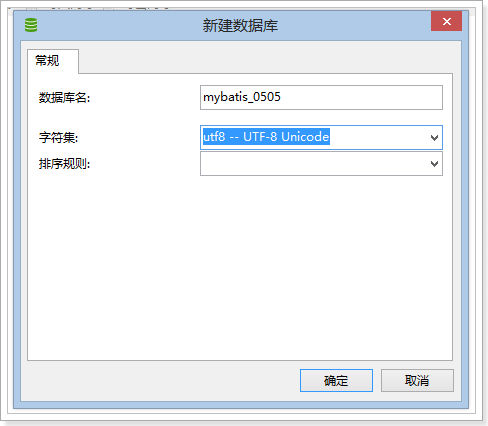
# 统一开发环境



# 从JDBC程序开始

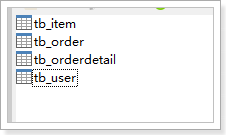
## 创建数据库

使用数据库是：mysql 5.6

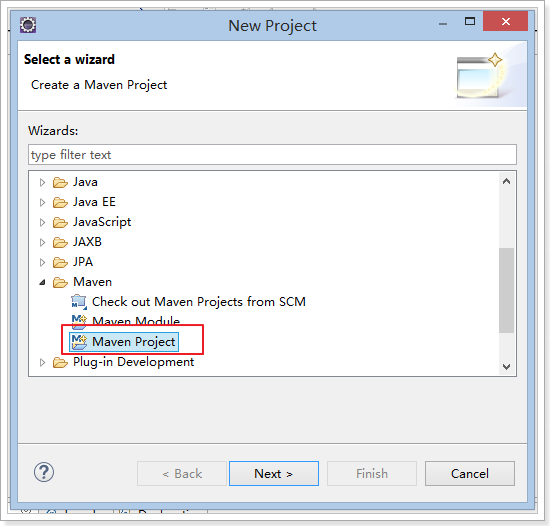


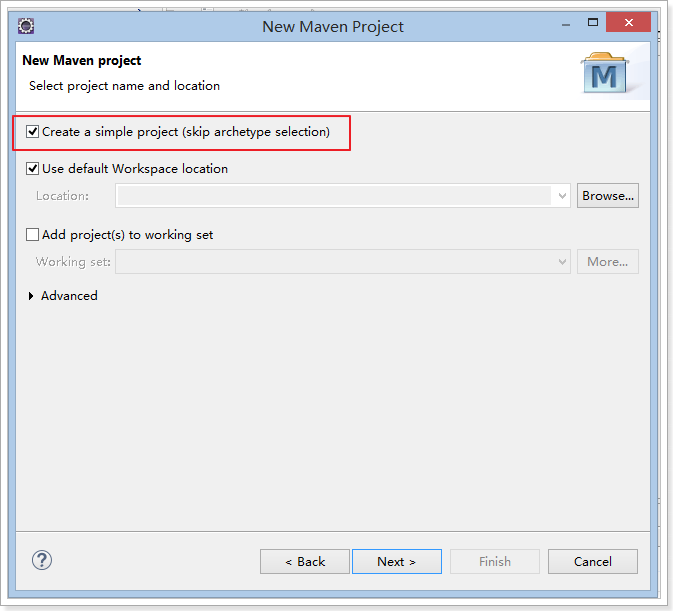
## 导入数据

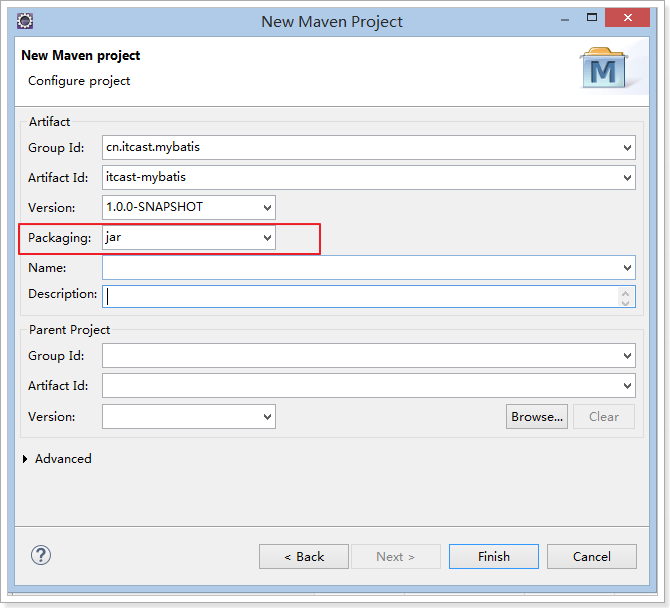
数据文件：day01（Mybatis）\开发资料\sql\mybatis.sql



## 创建工程

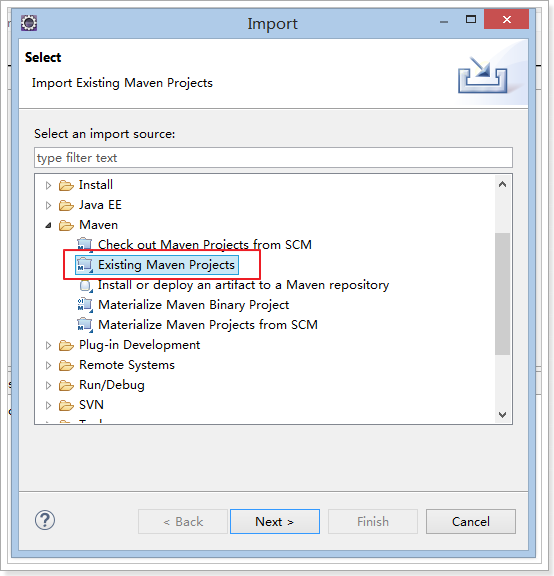


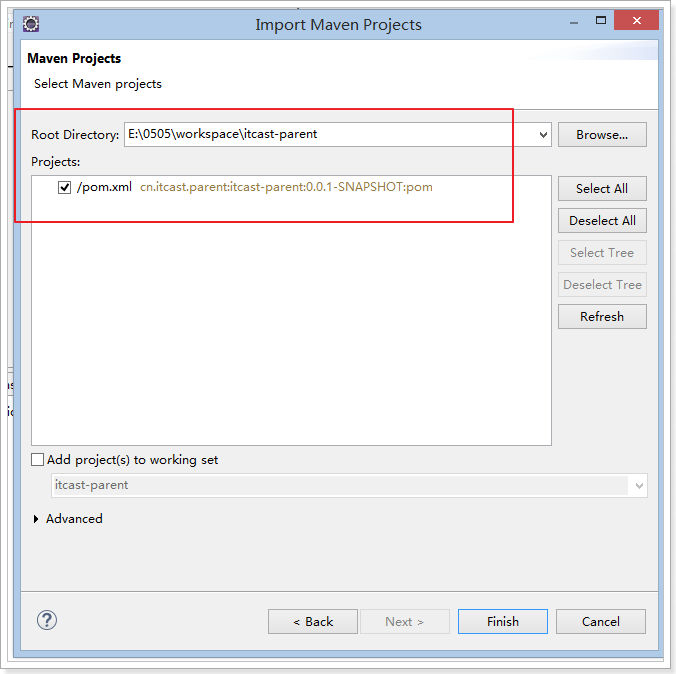




## 导入itcast-parent

day01（Mybatis）\开发工具环境\itcast-parent 拷贝到 E:\0505\workspace目录下

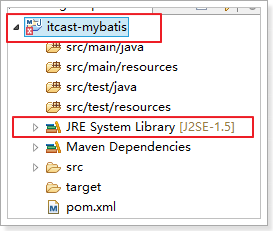


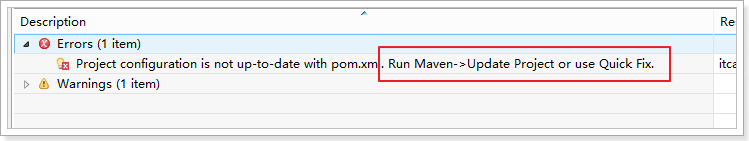


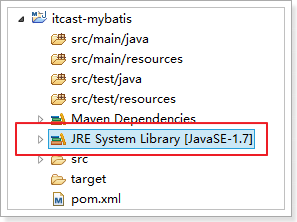
作用：统一维护依赖的版本号。



## 解决红叉







引入itcast-parent后导致红叉：



## 从原始JDBC编程中找出问题

### Jdbc程序

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

ResultSet resultSet = **null**;

PreparedStatement preparedStatement = **null**;

Connection connection = **null**;

**try** {

// 加载驱动

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

// 创建数据库连接

String url = "jdbc:mysql://127.0.0.1:3306/mybatis\_0505";

String user = "root";

String password = "123456";

connection = DriverManager.*getConnection*(url, user, password);

// 创建statmenet

String sql = "SELECT \* FROM tb\_user WHERE id = ?";

preparedStatement = connection.prepareStatement(sql);

// 设置参数、执行sql，2个参数，第一个是参数的下标，从1开始，第二个参数数据查询条件数据

preparedStatement.setLong(1, 1L);

resultSet = preparedStatement.executeQuery();

// 遍历结果集

**while** (resultSet.next()) {

System.***out***.println("ID: " + resultSet.getLong("id"));

System.***out***.println("userName: " + resultSet.getString("user\_name"));

System.***out***.println("password: " + resultSet.getString("password"));

System.***out***.println("name: " + resultSet.getString("name"));

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

// 释放连接（资源）

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

**try** {

resultSet.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

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

**try** {

preparedStatement.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

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

**try** {

connection.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

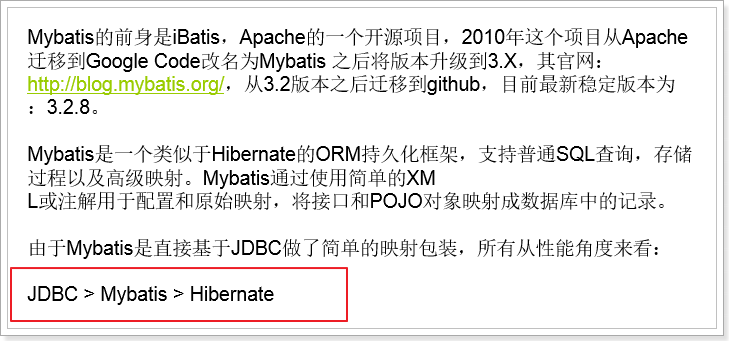
}

### 所存在的问题

1. 驱动包硬编码到java代码中，更换数据库必须修改java文件重新编写，解决：定义到配置文件中。
2. 数据库连接字符串、用户名、密码硬编码到java代码中，解决：定义到配置文件中。
3. Sql语句硬编码到java代码中
   1. 修改sql，必须修改java文件、重新编译
   2. 无法达到代码重用的作用
   3. 解决方案：能否将sql写到配置文件中？
4. 设置参数有问题
   1. 参数写死了，解决方案：传递进来即可
   2. 需要判断参数类型，解决方案：能否自动判断？
   3. 需要手动判断参数的位置，解决：能否自动判断？
5. 遍历结果集存在问题
   1. 需要判断类型
   2. 手动指定字段名
   3. 解决：能否直接将结果集映射为java对象？
6. 释放资源存在问题，频繁的创建连接和关闭连接，造成资源的浪费，影响系统的性能，解决：使用连接池

# Mybatis

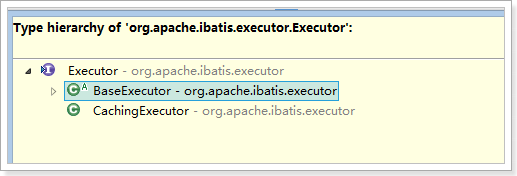
## Mybatis简介



## Mybatis的整体架构



总结：

1. mybatis的配置文件有2类
   1. mybatisconfig.xml，配置文件的名称不是固定的，配置了全局的参数的配置，全局只能有一个配置文件。
   2. Mapper.xml 配置多个statemement，也就是多个sql，整个mybatis框架中可以有多个Mappe.xml配置文件。
2. 通过mybatis配置文件得到SqlSessionFactory
3. 通过SqlSessionFactory得到SqlSession，得到SqlSession就可以操作数据了。
4. SqlSession通过底层的Executor（执行器），执行器有2类实现：
   1. 基本实现
   2. 带有缓存功能的实现  
      
5. MappedStatement是通过Mapper.xml中定义statement生成的对象。
6. 参数输入，无需手动判断参数类型和参数下标位置
   1. HashMap，KV格式的数据类型
   2. Java的基本数据类型
   3. POJO，java的对象
7. 结果集输出，自动将查询的结果集映射为java对象（POJO）
   1. HashMap，KV格式的数据类型
   2. Java的基本数据类型
   3. POJO，java的对象

# 第一个mybatis程序

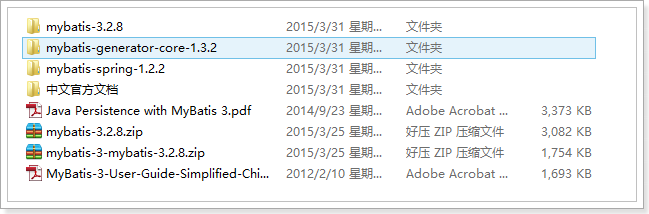
学习框架：看官网文档，一般都会有入门教程。

## 下载

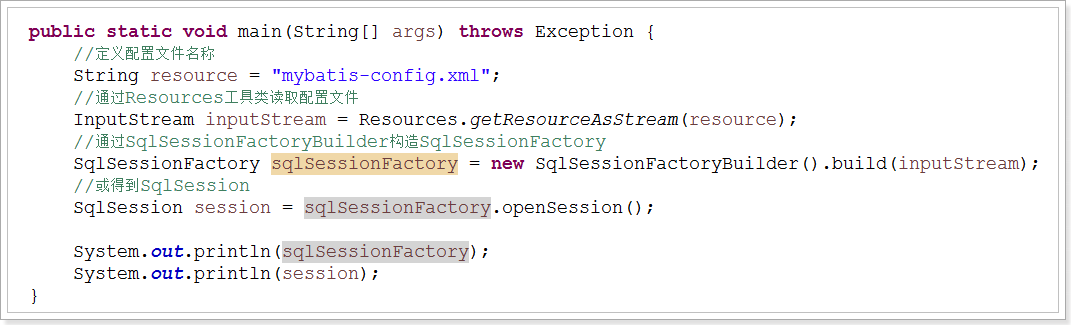
<https://github.com/mybatis/mybatis-3/releases>

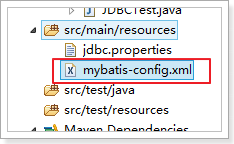


Mybatis学习相关的资料：



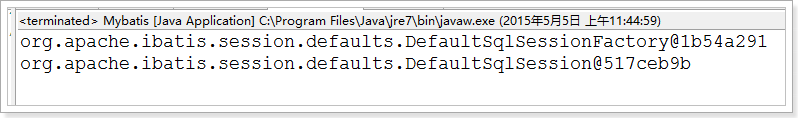
## 配置mybatis

1. 编写main方法实例化SqlSessionFactory  
   
2. 配置mybatis-config.xml

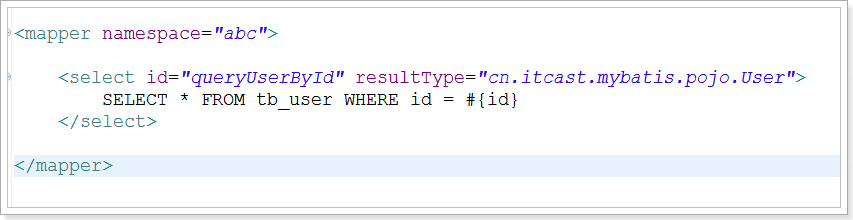
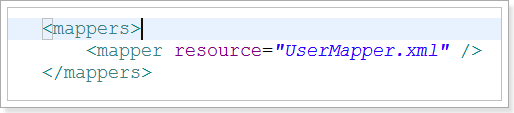
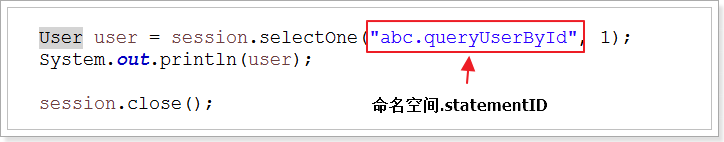
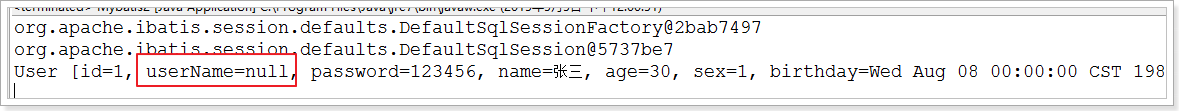




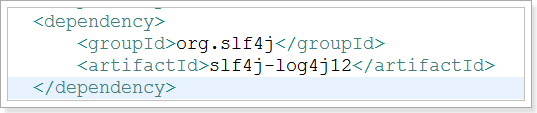
1. 测试

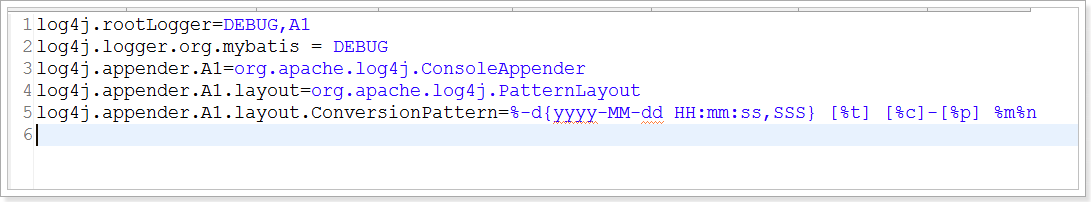


## 执行sql查询总结

1. 配置UserMapper.xml，配置sql  
   
2. 需要将UserMapper.xml添加到mybatis-config.xml中  
   
3. 通过SqlSession执行  
   
4. 测试  
   

## 添加日志支持





2015-05-05 14:33:57,561 [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Opening JDBC Connection

2015-05-05 14:33:57,807 [main] [org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] Created connection 1585118394.

2015-05-05 14:33:57,807 [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@5e7afcba]

2015-05-05 14:33:57,810 [main] [abc.queryUserById]-[DEBUG] ==> Preparing: SELECT \* FROM tb\_user WHERE id = ?

2015-05-05 14:33:57,892 [main] [abc.queryUserById]-[DEBUG] ==> Parameters: 1(Integer)

2015-05-05 14:33:57,914 [main] [abc.queryUserById]-[DEBUG] <== Total: 1

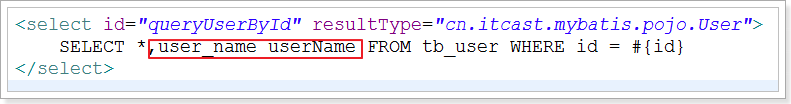
User [id=1, userName=null, password=123456, name=张三, age=30, sex=1, birthday=Wed Aug 08 00:00:00 CST 1984, created=Fri Sep 19 16:56:04 CST 2014, updated=Sun Sep 21 11:24:59 CST 2014]

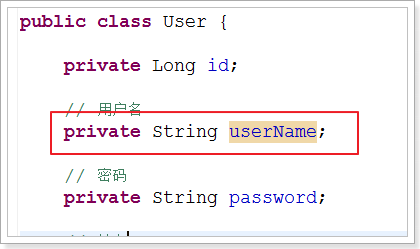
2015-05-05 14:33:57,918 [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@5e7afcba]

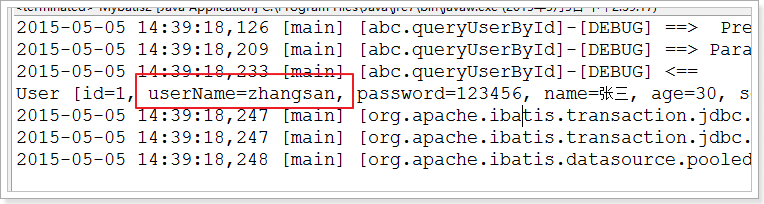
2015-05-05 14:33:57,919 [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@5e7afcba]

2015-05-05 14:33:57,926 [main] [org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] Returned connection 1585118394 to pool.

## 驼峰字段名的映射



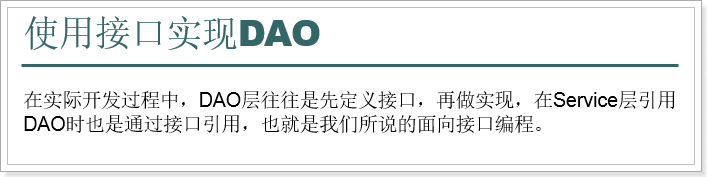




## Mybatis使用步骤总结



# 使用mybatis实现CRUD



## 定义UserDAO接口

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

/\*\*

\* 根据id查询用户数据

\*

\* **@param** id

\* **@return**

\*/

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

/\*\*

\* 新增user数据

\*

\* **@param** user

\*/

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

/\*\*

\* 更新user数据

\*

\* **@param** user

\*/

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

/\*\*

\* 根据id删除用户数据

\*

\* **@param** id

\*/

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

}

## 编写接口的实现类

**public** **class** UserDAOImpl **implements** UserDAO{

**private** SqlSessionFactory sqlSessionFactory;

**public** UserDAOImpl(SqlSessionFactory sqlSessionFactory){

**this**.sqlSessionFactory = sqlSessionFactory;

}

@Override

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

SqlSession session = **this**.sqlSessionFactory.openSession();

User user = session.selectOne("userDAO.queryUserById", id);

session.close();

**return** user;

}

@Override

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

SqlSession session = **this**.sqlSessionFactory.openSession();

session.insert("userDAO.saveUser", user);

//提交事物

session.commit();

session.close();

}

@Override

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

SqlSession session = **this**.sqlSessionFactory.openSession();

session.update("userDAO.updateUser", user);

//提交事物

session.commit();

session.close();

}

@Override

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

SqlSession session = **this**.sqlSessionFactory.openSession();

session.delete("userDAO.deleteUserById", id);

//提交事物

session.commit();

session.close();

}

}

## 编写UserDAOMapp.xml

<mapper namespace=*"userDAO"*>

<select id=*"queryUserById"* resultType=*"cn.itcast.mybatis.pojo.User"*>

SELECT \*,user\_name userName FROM tb\_user WHERE id = #{id}

</select>

<insert id=*"saveUser"* parameterType=*"cn.itcast.mybatis.pojo.User"*>

INSERT INTO tb\_user (

id,

user\_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

NULL,

#{userName},

#{password},

#{name},

#{age},

#{sex},

#{birthday},

NOW(),

NOW()

);

</insert>

<update id=*"updateUser"* parameterType=*"cn.itcast.mybatis.pojo.User"*>

UPDATE tb\_user

SET

user\_name = #{userName},

password = #{password},

name = #{name},

age = #{age},

sex = #{sex},

birthday = #{birthday},

updated = NOW()

WHERE

id = #{id}

</update>

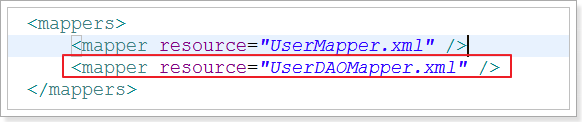
<delete id=*"deleteUserById"*>

DELETE FROM tb\_user WHERE id = #{id}

</delete>

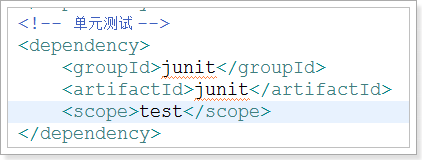
</mapper>

## 加入到mybatis-config.xml中

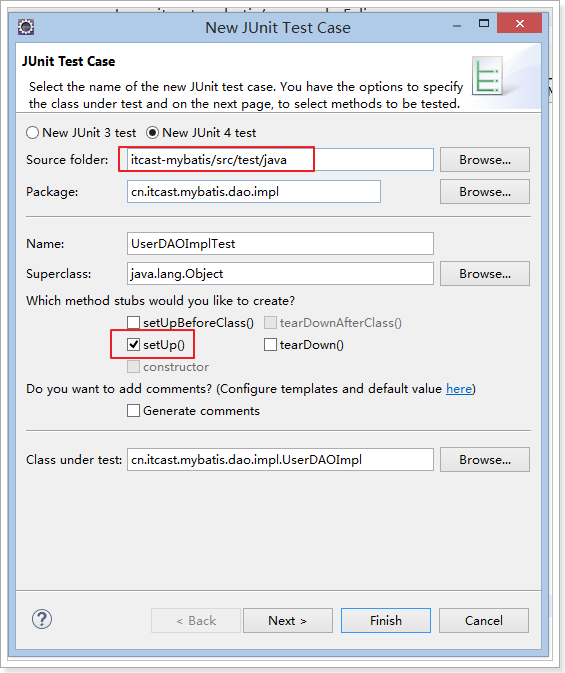


## 编写测试用例

导入junit支持：



创建测试用例：



实现：

**public** **class** UserDAOImplTest {

**private** UserDAO userDAO;

@Before

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

String resource = "mybatis-config.xml";

InputStream inputStream = Resources.*getResourceAsStream*(resource);

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

**this**.userDAO = **new** UserDAOImpl(sqlSessionFactory);

}

@Test

**public** **void** testQueryUserById() {

User user = **this**.userDAO.queryUserById(1L);

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

}

@Test

**public** **void** testSaveUser() {

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

user.setAge(20);

user.setBirthday(**new** Date());

user.setName("test\_1");

user.setPassword("123456");

user.setSex(1);

user.setUserName("test\_username\_1");

**this**.userDAO.saveUser(user);

}

@Test

**public** **void** testUpdateUser() {

User user = **this**.userDAO.queryUserById(2L);

user.setAge(30);

**this**.userDAO.updateUser(user);

}

@Test

**public** **void** testDeleteUserById() {

**this**.userDAO.deleteUserById(6L);

}

}

## 使用mybatis实现DAO存在的问题

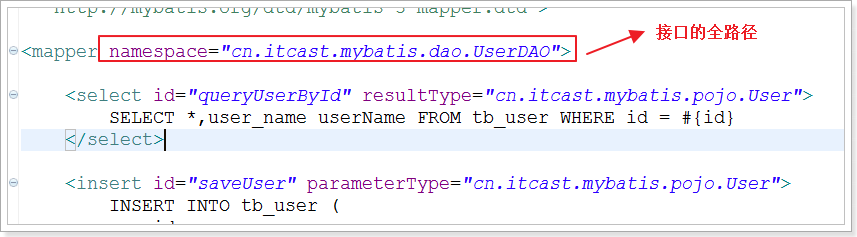
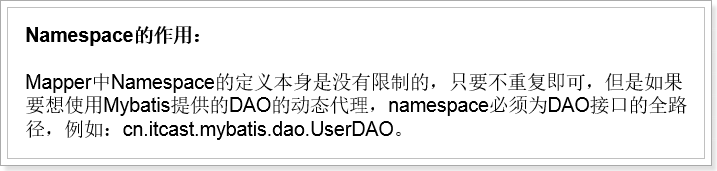
1. 所有的实现方法都非常的类似
2. Statementid硬编码到java代码中，不利维护

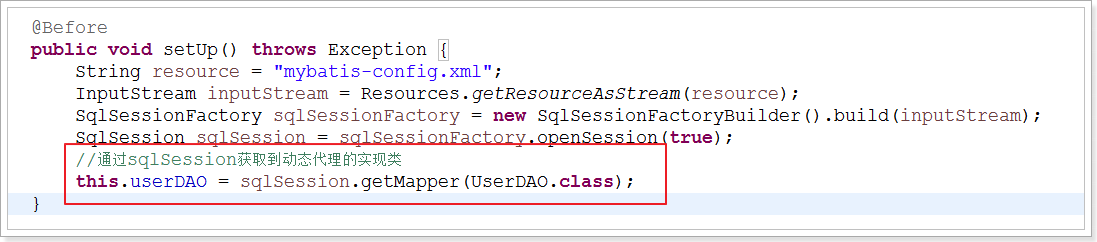
解决方案：

1. 抽取，用泛型实现
2. 能否不编写实现类，只是编写接口即可？
   1. 动态代理

## 使用动态代理实现UserDAO

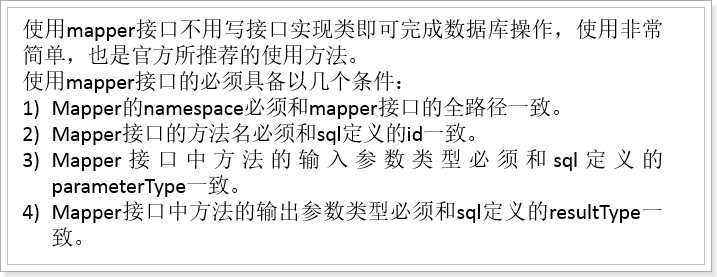
1. 编写UserDAO
   1. 同上代码
2. 编写Mapper.xml

1. 获取动态代理的实现类  
   
2. 测试

同上测试代码。

## 使用动态代理总结



约定优于配置。

# Mybatis-Config配置

在config.xml配置文件中的内容和**顺序**如下：

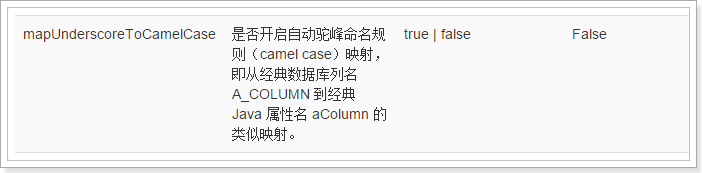


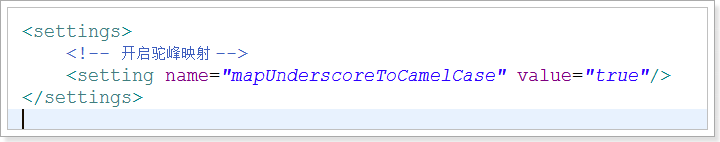
## Properties





## Setting

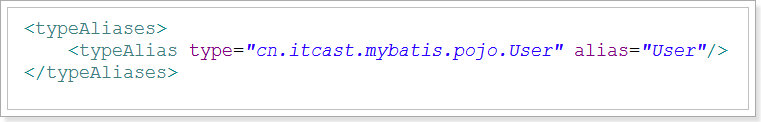




| **设置参数** | **描述** | **有效值** | **默认值** |
| --- | --- | --- | --- |
| cacheEnabled | 该配置影响的所有映射器中配置的缓存的全局开关。 | true | false | true |
| lazyLoadingEnabled | 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。特定关联关系中可通过设置fetchType属性来覆盖该项的开关状态。 | true | false | false |
| aggressiveLazyLoading | 当启用时，带有延迟加载属性的对象的加载与否完全取决于对任意延迟属性的调用；反之，每种属性将会按需加载。 | true | false | true |
| multipleResultSetsEnabled | 是否允许单一语句返回多结果集（需要兼容驱动）。 | true | false | true |
| useColumnLabel | 使用列标签代替列名。不同的驱动在这方面会有不同的表现，具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。 | true | false | true |
| useGeneratedKeys | 允许 JDBC 支持自动生成主键，需要驱动兼容。如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。 | true | false | False |
| autoMappingBehavior | 指定 MyBatis 是否以及如何自动映射指定的列到字段或属性。NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。FULL 会自动映射任意复杂的结果集（包括嵌套和其他情况）。 | NONE, PARTIAL, FULL | PARTIAL |
| defaultExecutorType | 配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）；BATCH 执行器将重用语句并执行批量更新。 | SIMPLE REUSE BATCH | SIMPLE |
| defaultStatementTimeout | 设置超时时间，它决定驱动等待数据库响应的秒数。 | Any positive integer | Not Set (null) |
| safeRowBoundsEnabled | 允许在嵌套语句中使用行分界（RowBounds）。 | true | false | False |
| mapUnderscoreToCamelCase | 是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。 | true | false | False |
| localCacheScope | MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。 | SESSION | STATEMENT | SESSION |
| jdbcTypeForNull | 当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。 | JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER | OTHER |
| lazyLoadTriggerMethods | 指定哪个对象的方法触发一次延迟加载。 | A method name list separated by commas | equals,clone,hashCode,toString |
| defaultScriptingLanguage | 指定动态 SQL 生成的默认语言。 | A type alias or fully qualified class name. | org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver |
| callSettersOnNulls | 指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这对于有 Map.keySet() 依赖或 null 值初始化的时候是有用的。注意原始类型（int、boolean等）是不能设置成 null 的。 | true | false | false |
| logPrefix | 指定 MyBatis 增加到日志名称的前缀。 | Any String | Not set |
| logImpl | 指定 MyBatis 所用日志的具体实现，未指定时将自动查找。 | SLF4J | LOG4J | LOG4J2 | JDK\_LOGGING | COMMONS\_LOGGING | STDOUT\_LOGGING | NO\_LOGGING | Not set |
| proxyFactory | 为 Mybatis 用来创建具有延迟加载能力的对象设置代理工具。 | CGLIB | JAVASSIST |  |

## typeAliases 别名

别名的作用：简化使用java对象类型。

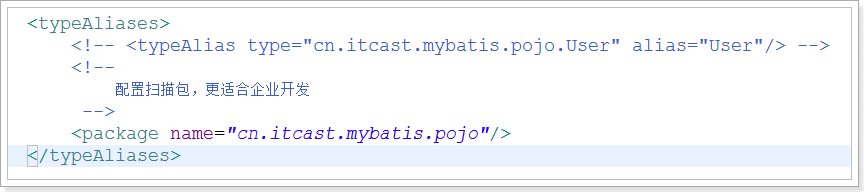


别名的首字母不区分大小写。

问题：

如果有很多pojo需要手动配置很多的别名，如果添加一个新的pojo对象就需要添加一个别名。

使用扫描包的方式解决这个问题：



Mybatis内置别名：

| **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 |

## typeHandlers

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时，都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。



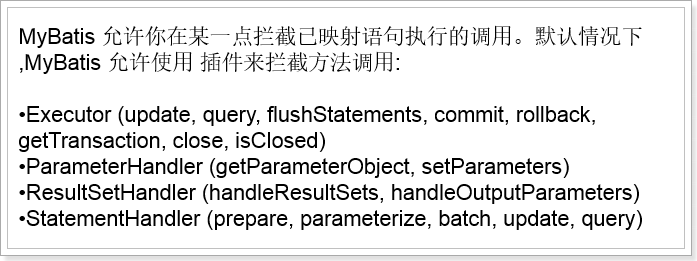
自定义类型处理器：



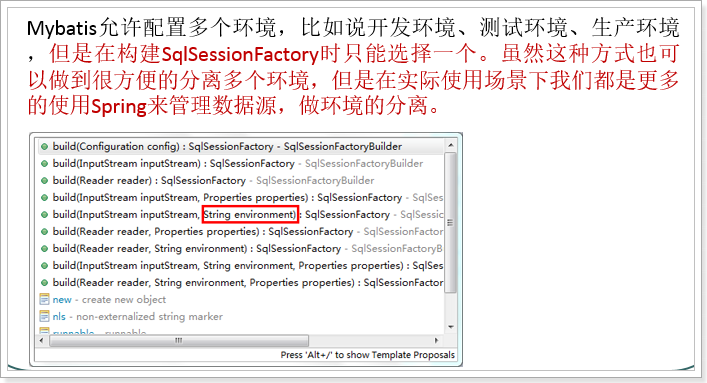
## objectFactory

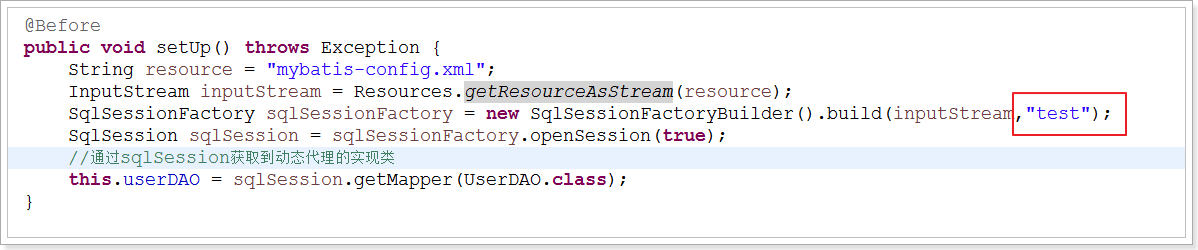


## plugins（插件又叫拦截器）



## environments（环境）







## mappers

mapper映射文件的引入有4种方式：

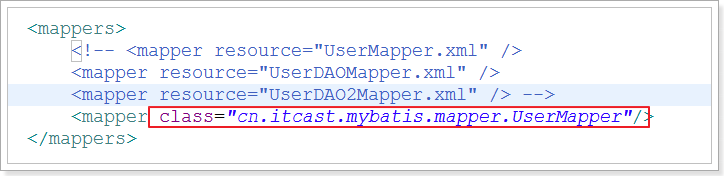


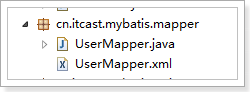
重点看后2种方式：

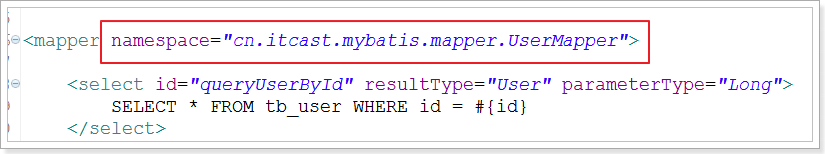
### class配置方式

要求：

1. Mapper.xml必须放置到接口所在的包中
2. Mapper.xm文件的文件名必须和接口名一致
3. Mapper.xml中的命名空间必须是该接口的全路径

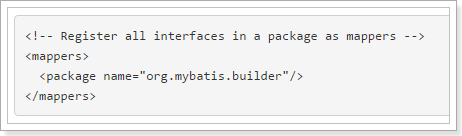


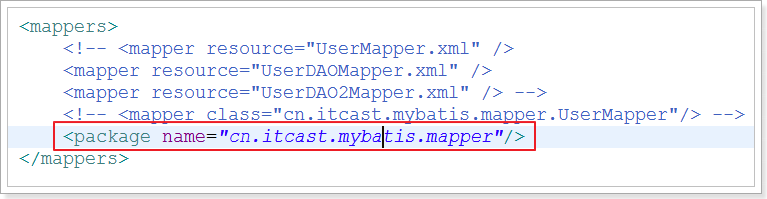




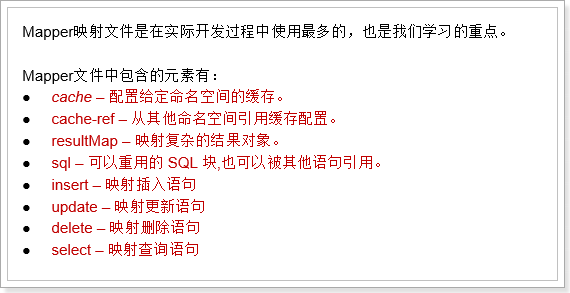
### 扫描包

建立在class配置的基础之上，无需一个个的配置接口。



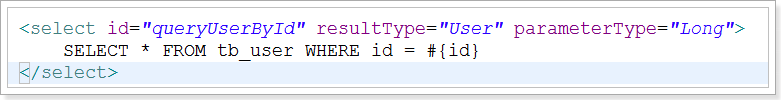


# Mapper XML



## CRUD

### Select

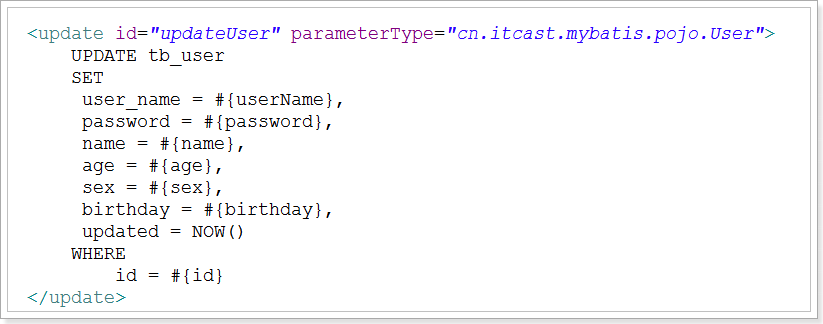


Id：必须属性，保证在同一个命名空间中唯一。

RestulltType：必须属性，值可以是对象全路径或者是别名。

ParameterType：非必须属性，如果不指定mybatis自动识别

### Update



Id：必须属性，保证在同一个命名空间中唯一。

ParameterType：非必须属性，如果不指定mybatis自动识别

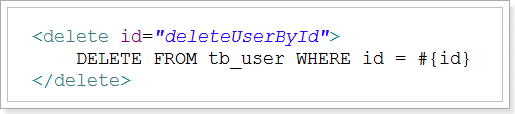
### Insert



Id：必须属性，保证在同一个命名空间中唯一。

ParameterType：非必须属性，如果不指定mybatis自动识别

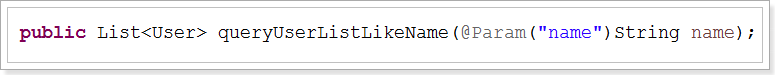
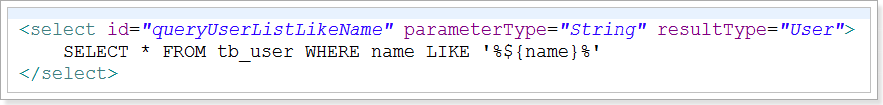
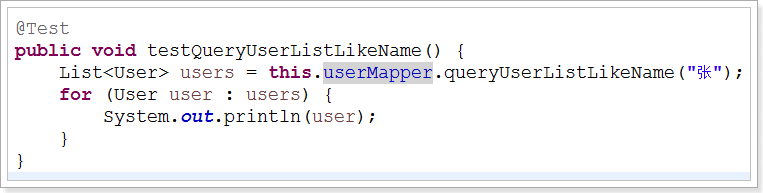
### Delete



Id：必须属性，保证在同一个命名空间中唯一。

ParameterType：非必须属性，如果不指定mybatis自动识别

## 实现根据用户姓名模糊查询数据

1. 接口中定义方法  
   
2. 在Mapper.xml中添加方法  
   
3. 测试  
   

在mybatis中获取传入参数有2种方式：

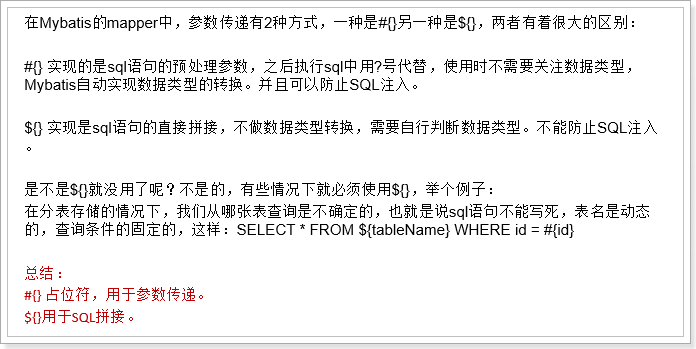
1、#{} – 占位符，和参数名无关

2、${} -- 获取参数值，拼接到sql中，和参数名有关

1、可以使用默认参数名获取参数值，${value}获取

2、在@Param注解指定参数的名称

## 面试题：#{}与${}的区别



# 小结

