



# Mybatis 框架课程第一天

## 第1章 框架概述

### 1.1 什么是框架

#### 1.1.1 什么是框架

框架（Framework）是整个或部分系统的可重用设计，表现为一组抽象构件及构件实例间交互的方法；另一种定义认为，框架是可被应用开发者定制的应用骨架。前者是从应用方面而后者是从目的方面给出的定义。

简而言之，框架其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统。简单说就是使用别人搭好的舞台，你来做表演。而且，框架一般是成熟的，不断升级的软件。

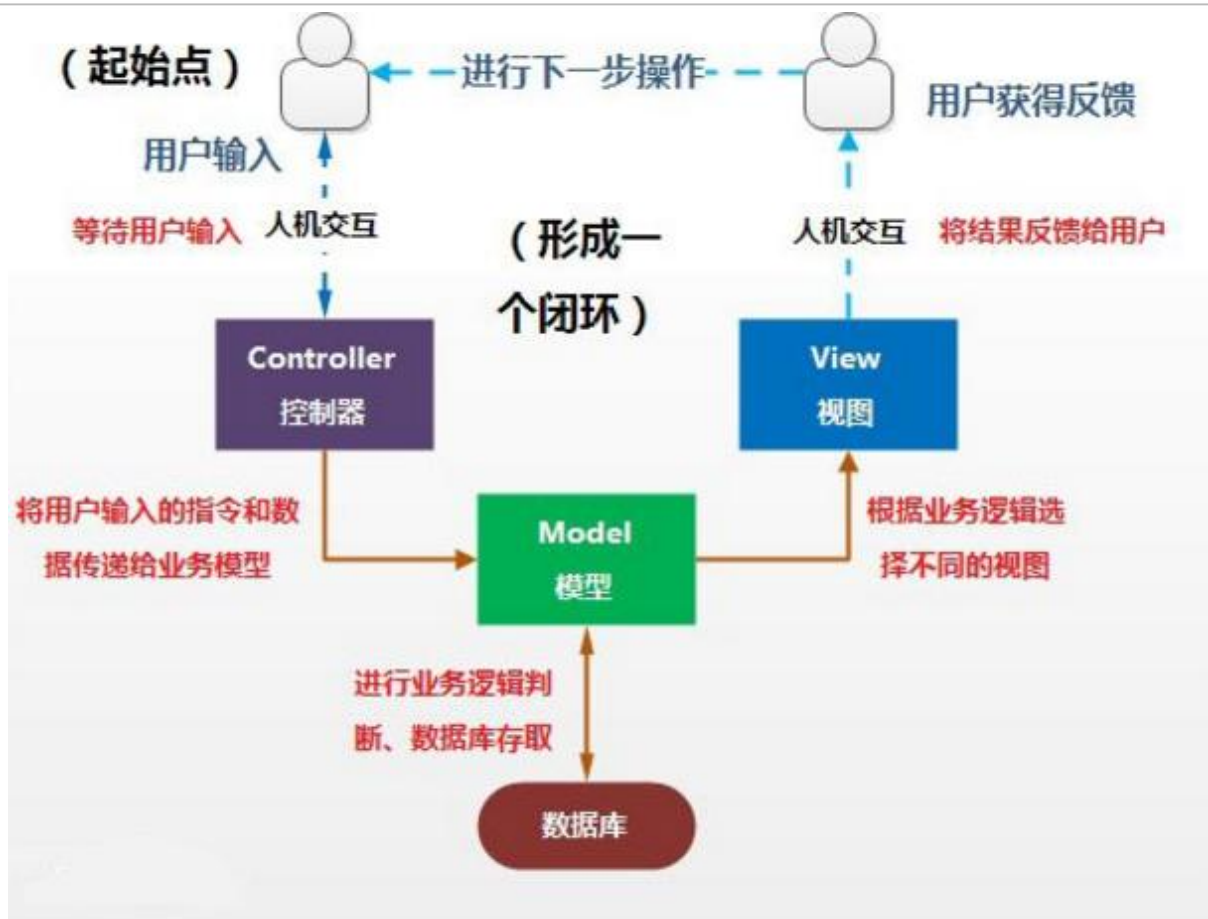
#### 1.1.2 框架要解决的问题

框架要解决的最重要的一个问题和技术整合的问题，在 J2EE 的 框架中，有着各种各样的技术，不同的软件企业需要从 J2EE 中选择不同的技术，这就使得软件企业最终的应用依赖于这些技术，技术自身的复杂性和技术的风险性将会直接对应用造成冲击。而应用是软件企业的核心，是竞争力的关键所在，因此应该将应用自身的设计和具体的实现技术解耦。这样，软件企业的研发将集中在应用的设计上，而不是具体的技术实现，技术实现是应用的底层支撑，它不应该直接对应用产生影响。

框架一般处在低层应用平台（如 J2EE）和高层业务逻辑之间的中间层。

#### 1.1.3 软件开发的分层重要性

框架的重要性在于它实现了部分功能，并且能够很好的将低层应用平台和高层业务逻辑进行了缓和。为了实现软件工程中的“高内聚、低耦合”。把问题划分开来各个解决，易于控制，易于延展，易于分配资源。我们常见的 MVC 软件设计思想就是很好的分层思想。



通过分层更好的实现了各个部分的职责，在每一层将再细化出不同的框架，分别解决各层关注的问题。

### 1.1.4 分层开发下的常见框架

常见的 JavaEE 开发框架：

#### 1、解决数据的持久化问题的框架

## MyBatis

编辑

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](#)。

iBatis 一词来源于 "internet" 和 "abatis" 的组合，是一个基于 Java 的持久层框架。iBatis 提供的持久层框架包括 SQL Maps 和 Data Access Objects (DAOs)。

作为持久层的框架，还有一个封装程度更高的框架就是 [Hibernate](#)，但这个框架因为各种原因目前在国内的流行程度下降太多，现在公司开发也越来越少使用。目前使用 [Spring Data](#) 来实现数据持久化也是一种趋势。

#### 2、解决 WEB 层问题的 MVC 框架



## spring MVC

Spring MVC属于SpringFrameWork的后续产品，已经融合在Spring Web Flow里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，从而在使用Spring进行WEB开发时，可以选择使用Spring的SpringMVC框架或集成其他MVC开发框架，如Struts1(现在一般不用)，Struts2等。

### 3、解决技术整合问题的框架

## spring框架

Spring框架是由于软件开发的复杂性而创建的。Spring使用的是基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅仅限于服务器端的开发。从简单性、可测试性和松耦合性角度而言，绝大部分Java应用都可以从Spring中受益。

- ◆目的：解决企业应用开发的复杂性
- ◆功能：使用基本的JavaBean代替EJB，并提供了更多的企业应用功能
- ◆范围：任何Java应用

Spring是一个轻量级控制反转(IoC)和面向切面(AOP)的容器框架。

### 1.1.5 MyBatis 框架概述

mybatis 是一个优秀的基于 java 的持久层框架，它内部封装了 jdbc，使开发者只需要关注 sql 语句本身，而不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。

mybatis 通过 xml 或注解的方式将要执行的各种 statement 配置起来，并通过 java 对象和 statement 中 sql 的动态参数进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并返回。

采用 ORM 思想解决了实体和数据库映射的问题，对 jdbc 进行了封装，屏蔽了 jdbc api 底层访问细节，使我们不用与 jdbc api 打交道，就可以完成对数据库的持久化操作。

为了我们能够更好掌握框架运行的内部过程，并且有更好的体验，下面我们将从自定义 Mybatis 框架开始来学习框架。此时我们将会体验框架从无到有的过程体验，也能够很好的综合前面阶段所学的基础。

## 1.2 JDBC 编程的分析

### 1.2.1 jdbc 程序的回顾

```
public static void main(String[] args) {  
    Connection connection = null;  
    PreparedStatement preparedStatement = null;  
    ResultSet resultSet = null;  
    try {  
        //加载数据库驱动  
        Class.forName("com.mysql.jdbc.Driver");  
    }  
}
```



```
//通过驱动管理类获取数据库链接
connection = DriverManager

.getConnection("jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8","root", "root");

//定义 sql 语句 ?表示占位符
String sql = "select * from user where username = ?";

//获取预处理 statement
preparedStatement = connection.prepareStatement(sql);

//设置参数，第一个参数为 sql 语句中参数的序号（从 1 开始），第二个参数为设置的
参数值

preparedStatement.setString(1, "王五");

//向数据库发出 sql 执行查询，查询出结果集
resultSet = preparedStatement.executeQuery();

//遍历查询结果集
while(resultSet.next()){
    System.out.println(resultSet.getString("id")+"
    "+resultSet.getString("username"));
}
} catch (Exception e) {
    e.printStackTrace();
}finally{
    //释放资源
    if(resultSet!=null){
        try {
            resultSet.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    if(preparedStatement!=null){
        try {
            preparedStatement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    if(connection!=null){
        try {
            connection.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

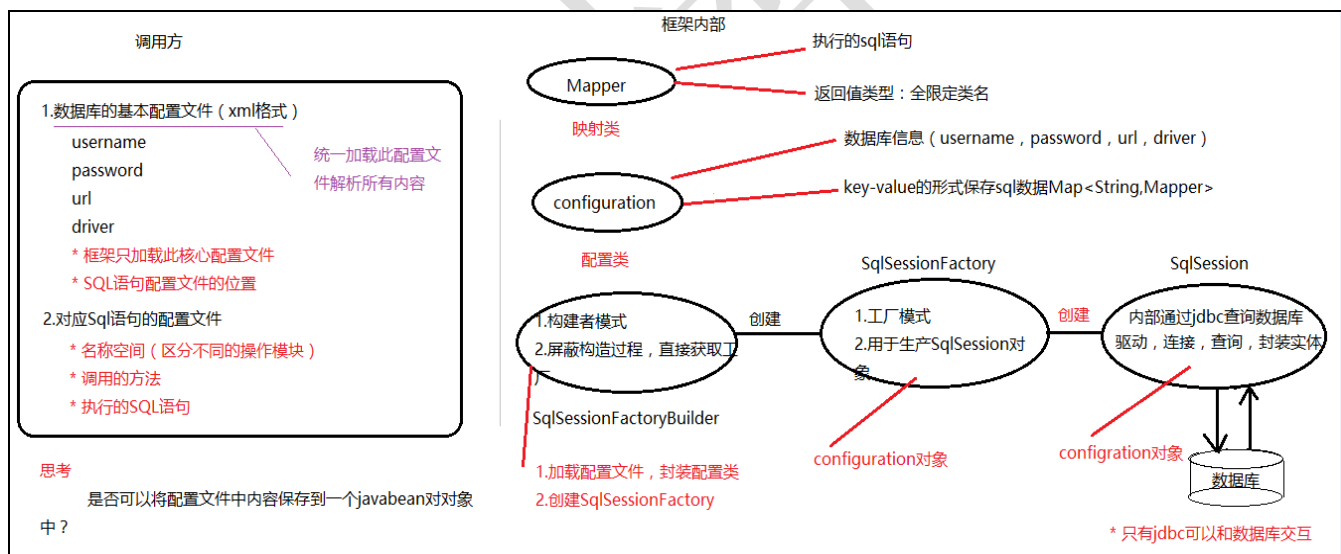
```
}  
}  
}
```

上边使用 jdbc 的原始方法（未经封装）实现了查询数据库表记录的操作。

## 1.2.2 jdbc 问题分析

- 1、数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。
- 2、Sql 语句在代码中硬编码，造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。
- 3、使用 preparedStatement 向占有位符号传参数存在硬编码，因为 sql 语句的 where 条件不一定，可能多也可能少，修改 sql 还要修改代码，系统不易维护。
- 4、对结果集解析存在硬编码（查询列名），sql 变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成 pojo 对象解析比较方便。

## 1.2.3 Jdbc 问题解决方案



通过上面的分析我们已找到了相对应的解决方案，下面我们将通过自己定义一个框架的方式来解决 jdbc 编程中所存在的问题。

# 第2章 自定义 Mybatis 框架

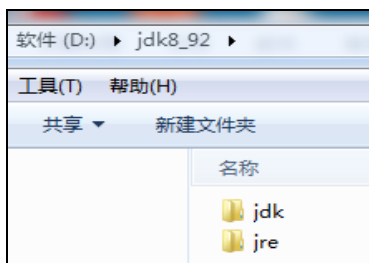
## 2.1 自定义 Mybatis 框架的前期准备

本章我们将使用前面所学的基础知识来构建一个属于自己的持久层框架，将会涉及到的一些知识点：工厂模式（Factory 工厂模式）、构造者模式（Builder 模式）、代理模式，反射，自定义注解，注解的反射，xml 解析，数据库元数据，元数据的反射等。

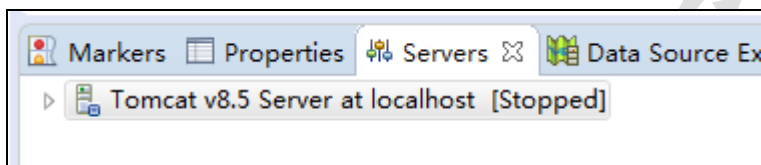
## 2.1.1 创建 Maven 工程并引入相关坐标

### 2.1.1.1 开发环境的准备及统一

#### 1. JDK 环境:

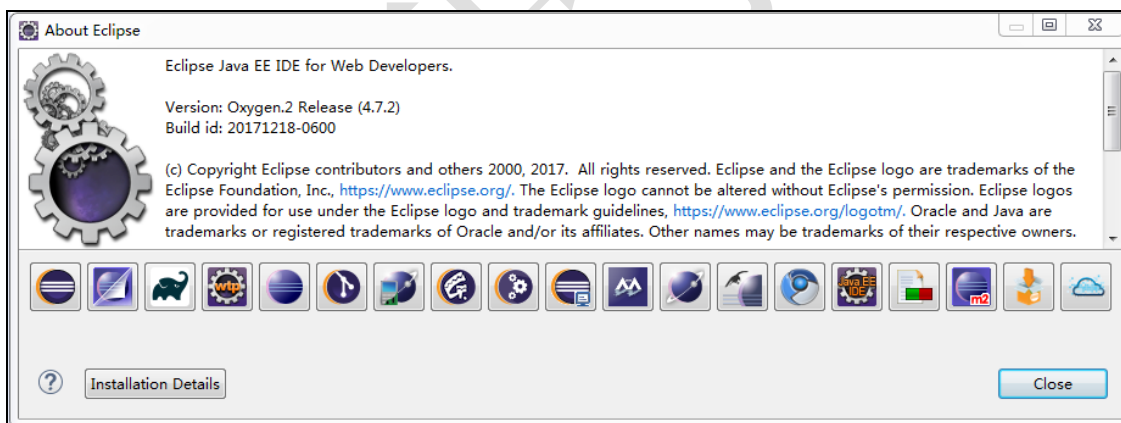


#### 2. Tomcat 环境:

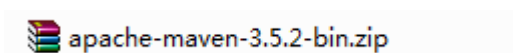


#### 3. Eclipse 环境准备:

目前使用的 Eclipse 版本 Oxygen (4.7.2 版本)

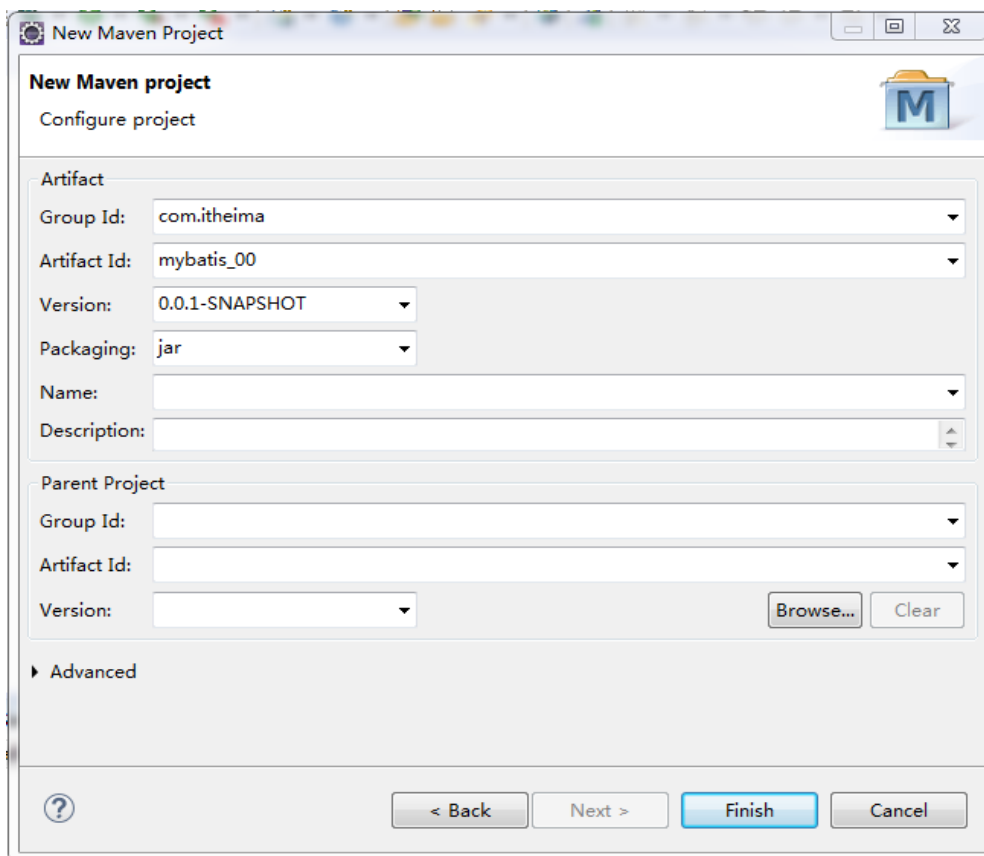


#### 4. maven 环境的准备

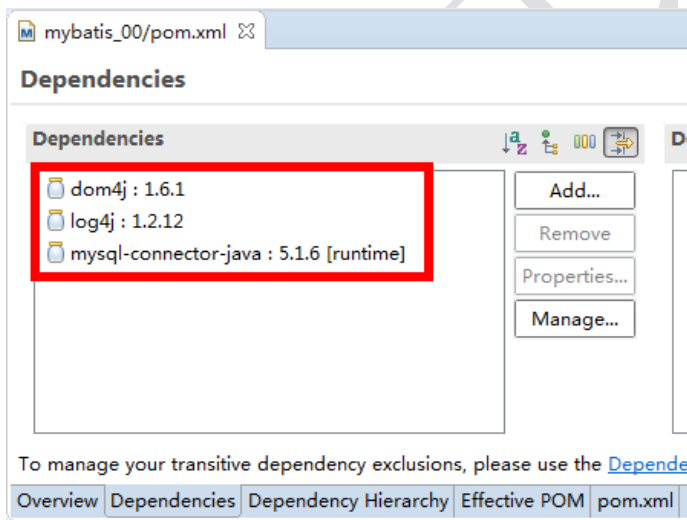


### 2.1.1.2 创建 Maven 工程

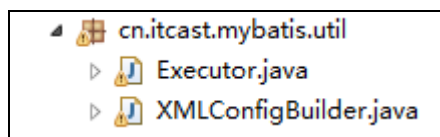
#### 1. 新建一个 maven project，并且使用传统的 java 方式。



第二步：引入相关的 jar 包坐标



第三步：在工程中将开发所需的工具类引入到项目中



所引入的类说明如下：

XMLConfigBuilder 类，用于实现 XML 文件解析的，解析工具目前采用的是 Dom4j 结合 xpath 实现。  
Executor 类，用于实现 SQL 语句的执行，主要是调用 JDBC 来实现 SQL 语句的执行。





## 2.1.2 创建 SqlMapConfig.xml 配置文件

为了更好地将数据库连接信息抽取出来，我们原来在 C3P0 连接池中也已经将数据库连接信息抽取出来，我们现在也一样将数据库的连接信息抽取出来，放到一个 xml (SqlMapConfig.xml) 文件中，后面再去对此配置文件进行 xml 解析，这样就可以将配置文件中的信息读取出来，以便在 jdbc 代码中直接使用这些数据库连接信息。

SqlMapConfig.xml 配置文件如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC" />
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver" ></property>
                <property name="url" value="jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf8" ></property>
                <property name="username" value="root"></property>
                <property name="password" value="root"></property>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

## 2.1.3 加入 Configuration 配置类

Configuration 配置类主要用于保存 SqlMapConfig.xml 文件中读取的 xml 结点的信息，以及映射的 SQL 语句的集合。该类的代码如下：

```
/**
 * 核心配置类
 * 1. 数据库信息
 * 2. sql 的 map 集合
 */
public class Configuration {
    private String username; //用户名
    private String password; //密码
    private String url; //地址
    private String driver; //驱动

    //map 集合 Map<唯一标识, Mapper> 用于保存映射文件中的 sql 标识及 sql 语句
    private Map<String, Mapper> mappers;

    public String getUsername() {
        return username;
    }
}
```





```

    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getDriver() {
        return driver;
    }

    public void setDriver(String driver) {
        this.driver = driver;
    }

    public Map<String, Mapper> getMappers() {
        return mappers;
    }

    public void setMappers(Map<String, Mapper> mappers) {
        this.mappers = mappers;
    }
}

```

## 2.1.4 创建数据表及 User 实体类

User 表:

Field Name	Datatype	Len	Default	Collation	PK?	Not Null?	Unsigned?	Auto Incr?	Zerofill?	Comment
* id	int	11			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
username	varchar	32		utf8_general_ci	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	用户名称
birthday	date				<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	生日
sex	char	1		utf8_general_ci	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	性别
address	varchar	256		utf8_general_ci	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	地址

User 实体类:

```

public class User implements Serializable {
    private int id;

    private String username; // 用户姓名

    private String sex; // 性别

    private Date birthday; // 生日
}

```



```
private String address;// 地址
//省略 getter 与 setter
@Override
public String toString() {
    return "User [id=" + id + ", username=" + username + ", sex=" + sex
        + ", birthday=" + birthday + ", address=" + address + "];"
}
}
```

## 2.1.5 创建自定义 @Select 注解

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Select {
    String value();
}
```

## 2.1.6 使用 @Select 注解开发 UserDao 接口

创建 UserDao 代理接口

```
public interface UserDao {
    @Select(value="select * from user")
    public List<User> getAllUserByMybatis() throws Exception;
    //ognl 表达式
    /**
     * sql
     *      --- #{username}
     *          username
     *          getUsername
     *      --- #{sex}
     *
     * insert into user (username,sex) values (user.get,#{sex})
     * @param user
     * @throws Exception
     */
    // @Insert(value="insert into user (username,sex) values (#{username},#{sex})")
    // public void insert(User user) throws Exception;
}
```

## 2.2 基于注解方式定义 Mybatis 框架

### 2.2.1 使用工厂模式开发

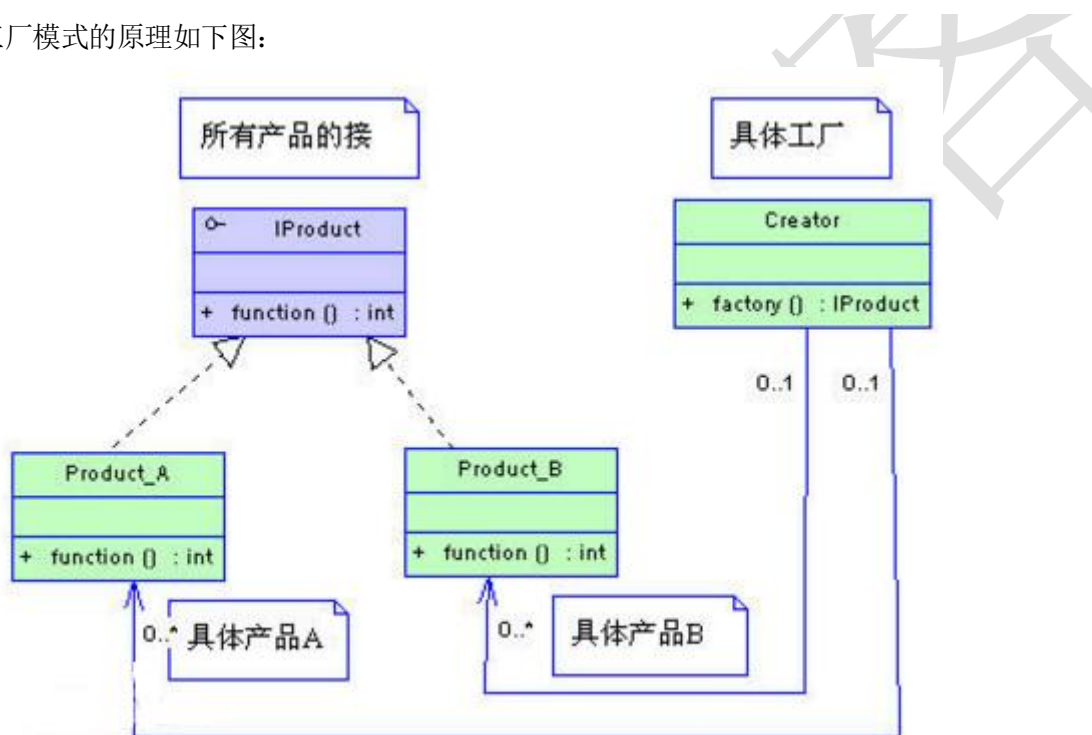
工厂模式的回顾

#### 工厂模式

编辑

工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。著名的Jive论坛，就大量使用了工厂模式，工厂模式在Java程序系统可以说是随处可见。因为工厂模式就相当于创建实例对象的new，我们经常要根据类Class生成实例对象，如A a=new A() 工厂模式也是用来创建实例对象的，所以以后new时就要多个心眼，是否可以考虑使用工厂模式，虽然这样做，可能多做一些工作，但会给你系统带来更大的可扩展性和尽量少的修改量。

工厂模式的原理如下图：



#### 2.2.1.1 第一步：自定义 SqlSession 接口

```
public interface SqlSession {  
    public <E> List<E> selectList(String mapperId) throws Exception;  
    /**  
     * 创建动态代理对象  
     * clazz : 接口的字节码  
     */  
    public <E> E getMapper(Class clazz) throws Exception;  
}
```

这就相当于按工厂模式的图中，创建了一个用于生产产品的接口。其中 getMapper()方法可以用于模



拟按 mybatis 的代理方式来生成，而 selectList()方法可以模拟按 mybatis 的传统方式来生成。我们先只来关注传统方式来实现。

### 2.2.1.2 第二步：定义 DefaultSqlSession 实现类

DefaultSqlSession 类主要用于生成 mapper 代理对象，以及执行 SQL 语句的方法。

编写 DefaultSqlSession.java 实现类：

```
public class DefaultSqlSession implements SqlSession {

    private Configuration cfg; //封装 mybatis 配置文件及映射文件
    private Executor executor; //SQL 语句执行器

    public DefaultSqlSession(Configuration cfg) {
        this.cfg = cfg;
        this.executor = new Executor(this.cfg);
    }

    /**
     * 生成 mapper 的代理对象
     */
    public <E> E getMapper(Class clazz) throws Exception {
        //创建 invocationHandler
        MapperProxyFactory proxy = new MapperProxyFactory(this);
        //创建动态代理对象
        return (E) Proxy.newProxyInstance(clazz.getClassLoader(), new Class[]
        {clazz}, proxy);
    }

    /**
     * 查询列表
     * 定位到 sql 语句
     */
    public <E> List<E> selectList(String mapperId) throws Exception {
        //通过 id 获取 Mapper 对象
        Mapper mapper = cfg.getMappers().get(mapperId);
        //通过 mapper 获取 sql 语句和返回值类型
        String sql = mapper.getQuerySql();
        String resultType = mapper.getResultType(); //返回值的全限定类名
        //调用 SQL 语句的 Executor 执行器来执行 sql 语句
        return executor.executeQuery(sql, resultType);
    }

    public Executor getExecutor() {
        return executor;
    }
}
```

上面所编写的类就相当于有了一个具体产品，就是我们的 `SqlSessionImpl` 实现类。但其中 `MapperProxyFactory` 类就会存在问题，因为我们目前还没有这个类。`MapperProxyFactory` 类采用的是代理方式来实现的。

## 2.2.2 使用代理模式编写 `MapperProxyFactory` 类

### 2.2.2.1 代理模式的回顾

#### 代理模式

编辑

组成：

抽象角色：通过接口或抽象类声明真实角色实现的业务方法。

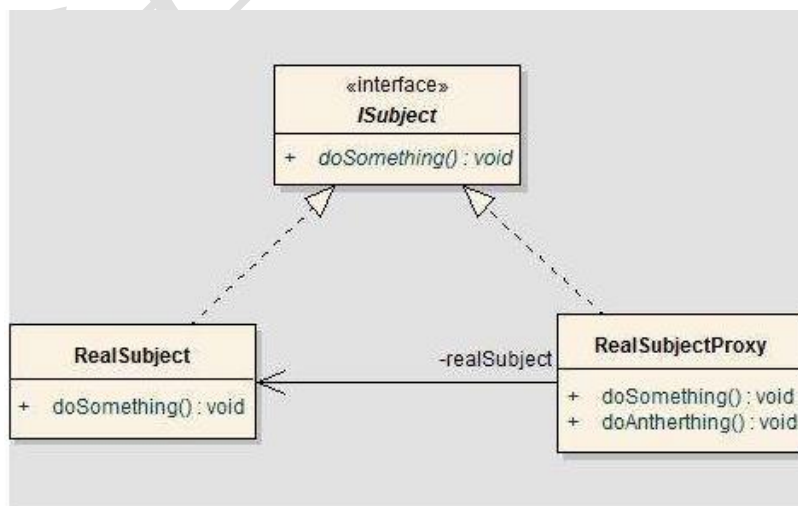
代理角色：实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。

真实角色：实现抽象角色，定义真实角色所要实现的业务逻辑，供代理角色调用。

代理模式分为静态和动态代理。静态代理，我们通常都很熟悉。有一个写好的代理类，实现与要代理的类的一个共同的接口，目的是为了约束也为了安全。具体不再多说。

这里主要想说的是关于动态代理。我们知道静态代理若想代理多个类，实现扩展功能，那么它必须具有多个代理类分别取代理不同的实现类。这样做的后果是造成太多的代码冗余。那么我们会思考如果做，才能既满足需求，又没有太多的冗余代码呢？————动态代理。通过前面的课程我们已经学过了基于 `JDK` 的动态代理实现方式，今天我们就使用 `JDK` 动态代理方式来编写 `MapperProxyFactory` 类。

动态代理模型图：





## 2.2.2.2 编写 MapperProxyFactory 类

因为 DefaultSqlSession 类会自动去针对 mapper 接口生成它的代理实现类，而这个代理实现的过程就会用到我们的 MapperProxyFactory 类。

```
public class MapperProxyFactory implements InvocationHandler {  
    private DefaultSqlSession sqlSession;  
    public MapperProxyFactory(DefaultSqlSession sqlSession) {  
        this.sqlSession = sqlSession;  
    }  
    /**  
     * proxy:代理对象的引用  
     *      获取代理对象这个实现类的接口类型  
     *      获取当前接口的全限定类名  
     * method: 当前执行的方法对象  
     *      当前执行的方法名称  
     *  
     * mapperId = 获取当前接口的全限定类名 + "." + 当前执行的方法名称  
     * args: 当前参数  
     */  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable  
{  
  
        //判断当前方法上是否存在 select 注解  
        boolean present = method.isAnnotationPresent(Select.class);  
        if(present) { //存在 select 注解  
            //1.获取 sql 语句  
            Select select = method.getAnnotation(Select.class); //获取当前方法上的 select 注解  
            String sql = select.value();  
            String returnType = method.getGenericReturnType().toString();  
            if(returnType.startsWith(List.class.getName())) {  
                //处理 List 集合中的<>部分  
                returnType = returnType.replace(List.class.getName(),  
"".replace("<", "").replace(">", ""));  
            }  
            //2.得到返回值类型  
            return sqlSession.getExecutor().executeQuery(sql, returnType);  
        } else { //没有注解  
            //invoke 方法: 就是在查询数据库  
            //获取当前接口的全限定类名  
            //getGenericInterfaces: 获取实体类的所有接口  
            String className =  
                proxy.getClass().getGenericInterfaces()[0].getTypeName();  
            //获取当前执行的方法名称
```



```
String methodName = method.getName();  
String mapperId = className + "." + methodName;  
return sqlSession.selectList(mapperId);  
}  
}  
}
```

在上面的代码中 `Select selectAnn = method.getAnnotation(Select.class);` 使用注解的反射来得到注解对象，再通过 `String returnType = method.getGenericReturnType().toString();` 来得到注解的元数据，从而得到返回值类型。

## 2.2.3 使用构建者模式来创建 SqlSessionFactoryBuilder 类

### 2.2.3.1 构建者模式的学习

首先我们一起来学习构建者模式，通过百度百科如下：

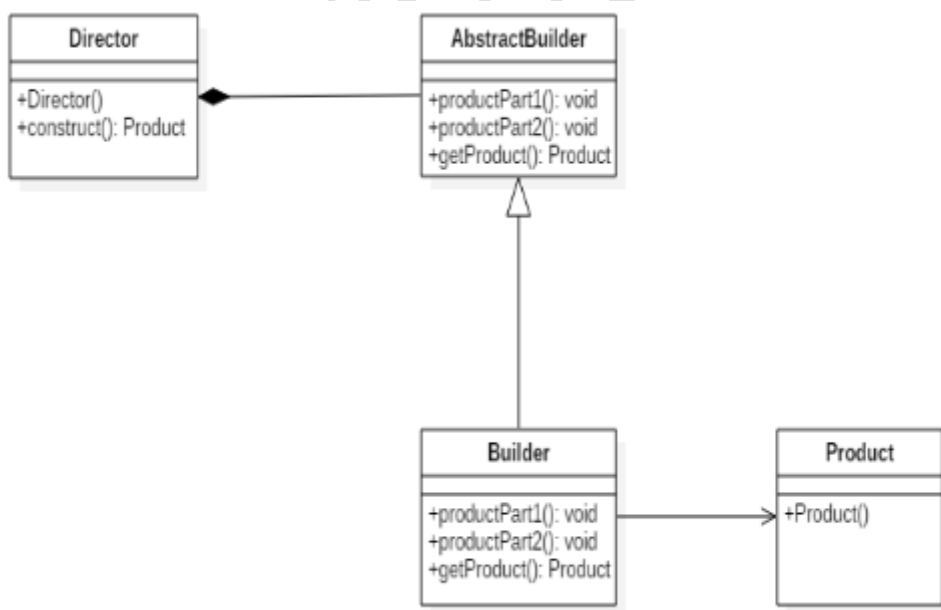
#### 创建者模式

[编辑](#)

本词条缺少名片图，补充相关内容使词条更完整，还能快速升级，赶紧来[编辑](#)吧！

java23种设计模式之一，英文叫Builder Pattern。其核心思想是将一个“复杂对象的构建算法”与它的“部件及组装方式”分离，使得构件算法和组装方式可以独立应对变化；复用同样的构建算法可以创建不同的表示，不同的构建过程可以复用相同的部件组装方式。

具体设计模式的模型图如下：



从图中我们可以看出，创建者模式由四部分组成。

**抽象创建者角色：** 给出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此接口独立于应用程序的商业逻辑。模式中直接创建产品对象的是具体创建者角色。具体创建者必须实现





这个接口的两种方法：一是建造方法，比如图中的 `buildPart1` 和 `buildPart2` 方法；另一种是结果返回方法，即图中的 `getProduct` 方法。一般来说，产品所包含的零件数目与建造方法的数目相符。换言之，有多少零件，就有多少相应的建造方法。

**具体创建者角色：**他们在应用程序中负责创建产品的实例。这个角色要完成的任务包括：

- 1、实现抽象创建者所声明的抽象方法，给出一步一步的完成产品创建实例的操作。
- 2、在创建完成后，提供产品的实例。

**导演者角色：**这个类调用具体创建者角色以创建产品对象。但是导演者并没有产品类的具体知识，真正拥有产品类的具体知识的是具体创建者角色。

**产品角色：**产品便是建造中的复杂对象。一般说来，一个系统中会有多于一个的产品类，而且这些产品类并不一定有共同的接口，而完全可以使不相关联的。

### 2.2.3.2 编写 `SqlSessionFactoryBuilder` 类

```
/**
 * SqlSessionFactoryBuilder:
 *     1.构建者模式
 *     2.用来创建 SqlSessionFactory
 *     3.隐藏工厂的构建细节
 *         * 解析 xml 配置文件
 *         * 构建一个 Configuration 对象
 */
public class SqlSessionFactoryBuilder {
    /**
     * 解析 xml
     *     dom4j
     *     saxReader
     *     reader 方法（字节流）
     * @param is
     * @return
     */
    public SqlSessionFactory build(InputStream is) throws Exception {
        //1.构造 Configuration 调用 XMLConfigBuilder 类实现 xml 解析
        Configuration cfg = XMLConfigBuilder.buildConfiguration(is);
        return new SqlSessionFactory(cfg);
    }
}
```

上面的类中调用了 `XMLConfigBuilder` 类中读取 xml 文件并构建出 `Configuration` 对象的方法。最后通过 `Configuration` 对象来构造 `SqlSessionFactory` 对象。此时我们已经将构建者模式和工厂模式结合，而工厂模式和动态代理模式结合，有点小激动。下面我们一起学习 `SqlSessionFactory` 工厂，它主要是用于创建 `SqlSession` 对象的。



### 2.2.3.3 编写 SqlSessionFactory 类

SqlSessionFactory 类如下：

```
/**
 * SqlSession 工厂
 * 1.工厂模式
 * 2.获取 SqlSession
 */
public class SqlSessionFactory {
    private Configuration cfg ;
    public SqlSessionFactory(Configuration cfg) {
        this.cfg = cfg;
    }
    //获取 SqlSession
    public SqlSession openSession() {
        return new DefaultSqlSession(cfg);
    }
}
```

可以发现 SqlSessionFactory 类进一步通过 openSession()方法，包装了 SqlSession 的实现创建过程。

### 2.2.4 测试自定义 Mybatis 框架效果

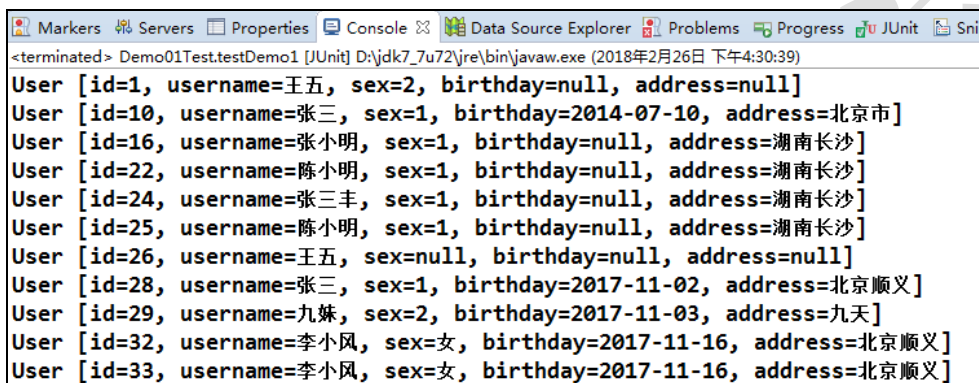
#### 2.2.4.1 编写测试类

```
public class UserDaoTest {
    /**
     * 动态代理对象中查询：
     * mapperId: 接口全限定类名+"."+执行的方法名
     * mapperId: cn.itcast.user.dao.UserDao.getAllUserByMybatis
     */
    @Test
    public void testGetAllUserByMybatis() throws Exception {
        InputStream is =
        this.getClass().getClassLoader().getResourceAsStream("SqlMapConfig.xml");
        //c 创建 SqlSessionFactory
        SqlSessionFactory sessionFactory
            = new SqlSessionFactoryBuilder().build(is);
        //创建 sqlSession
        SqlSession sqlSession = sessionFactory.openSession();
    }
}
```



```
//获取接口的动态代理对象
UserDao userDao = sqlSession.getMapper(UserDao.class);
//调用方法查询
List<User> list = userDao.getAllUserByMybatis();
for (User user : list) {
    System.out.println(user);
}
}
```

运行效果如下:



```
<terminated> Demo01Test.testDemo1 [JUnit] D:\jdk7_7u72\jre\bin\javaw.exe (2018年2月26日 下午4:30:39)
User [id=1, username=王五, sex=2, birthday=null, address=null]
User [id=10, username=张三, sex=1, birthday=2014-07-10, address=北京市]
User [id=16, username=张小明, sex=1, birthday=null, address=湖南长沙]
User [id=22, username=陈小明, sex=1, birthday=null, address=湖南长沙]
User [id=24, username=张三丰, sex=1, birthday=null, address=湖南长沙]
User [id=25, username=陈小明, sex=1, birthday=null, address=湖南长沙]
User [id=26, username=王五, sex=null, birthday=null, address=null]
User [id=28, username=张三, sex=1, birthday=2017-11-02, address=北京顺义]
User [id=29, username=九妹, sex=2, birthday=2017-11-03, address=九天]
User [id=32, username=李小明, sex=女, birthday=2017-11-16, address=北京顺义]
User [id=33, username=李小明, sex=女, birthday=2017-11-16, address=北京顺义]
```

#### 2.2.4.2 小结

通过自定义 Mybatis 框架的学习，我们将前面的基础知识很好的结合在一起，并且强化了我们的设计模式及使用。希望大家能够抽时间多练习，这也是系统架构师的必由之路。

## 第3章 Mybatis 框架快速入门

通过前面的学习，我们已经能够使用所学的基础知识构建自定义的 Mybatis 框架了。这个过程是基本功的考验，我们已经强大了不少，但现实是残酷的，我们所定义的 Mybatis 框架和真正的 Mybatis 框架相比，还是显得渺小。行业内所流行的 Mybatis 框架现在我们将开启学习。

### 3.1 Mybatis 框架开发的准备

#### 3.1.1 官网下载 Mybatis 框架

从百度中“mybatis download”可以下载最新的 Mybatis 开发包。



mybatis download

### mybatis – MyBatis 3 | Introduction

查看此网页的中文翻译，请点击 [翻译此页](#)

MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of ...

[www.mybatis.org/mybatis-3](http://www.mybatis.org/mybatis-3) - 百度快照

进入选择语言的界面，进入中文版本的开发文档。



我们可以看到熟悉的中文开发文档了。

## 简介

### 什么是 MyBatis ?

MyBatis 是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java对象)映射成数据库中的记录。



# MyBatis

下载相关的 jar 包或 maven 开发的坐标。



下载的 zip 文件如下（我们的资料文件夹）：

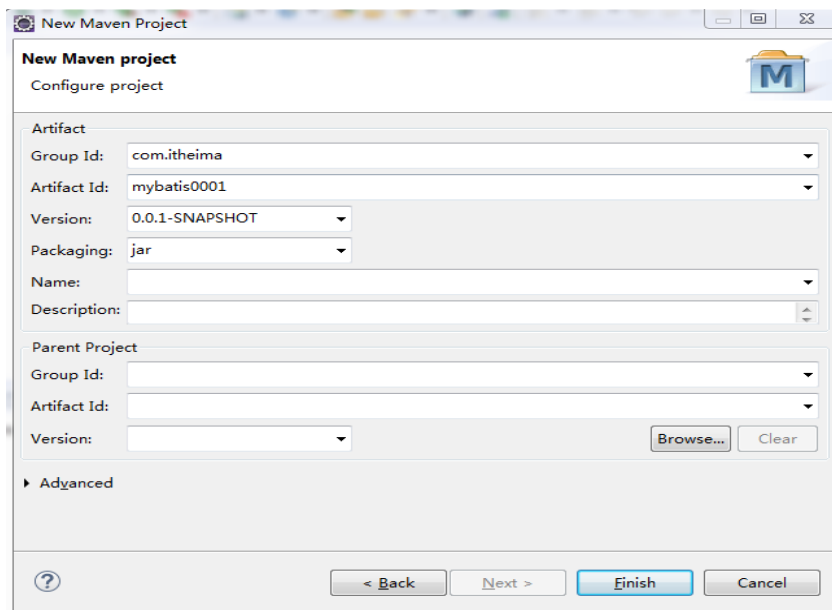


mybatis-3.4.5.zip

我们所使用的 Mybatis 版本是 3.4.5 版本。

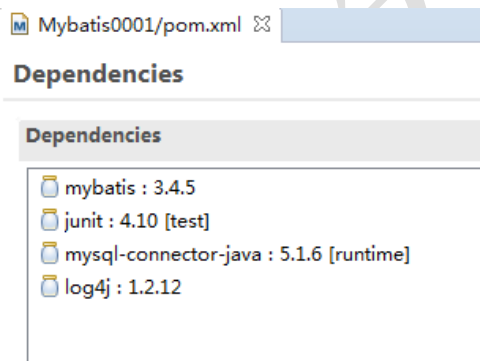
## 3.1.2 搭建 Mybatis 开发环境

### 3.1.2.1 1.创建 maven 工程



### 3.1.2.2 添加 Mybatis3.4.5 的坐标

在 pom.xml 文件中添加 Mybatis3.4.5 的坐标，如下：





### 3.1.2.3 编写 User 实体类

```
User.java
1 package com.itheima.domain;
2
3 import java.io.Serializable;
4
5
6 public class User implements Serializable {
7     private int id;
8     private String username; // 用户姓名
9     private String sex; // 性别
10    private Date birthday; // 生日
11    private String address; // 地址
12 }
```

### 3.1.2.4 编写 UserMapper 接口

UserMapper 接口就是我们的持久层接口（也可以写成 UserDao），具体代码如下：

```
UserMapper.java
1 package com.itheima.mapper;
2
3 import java.util.List;
4
5
6
7 public interface UserMapper {
8
9     public List<User> findAll();
10 }
```

### 3.1.2.5 编写 UserMapper.xml 映射文件

```
UserMapper.xml
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.itheima.mapper.UserMapper">
6     <select id="findAll" resultType="com.itheima.domain.User">
7         select * from user
8     </select>
9 </mapper>
```



### 3.1.2.6 编写 SqlMapConfig.xml 配置文件

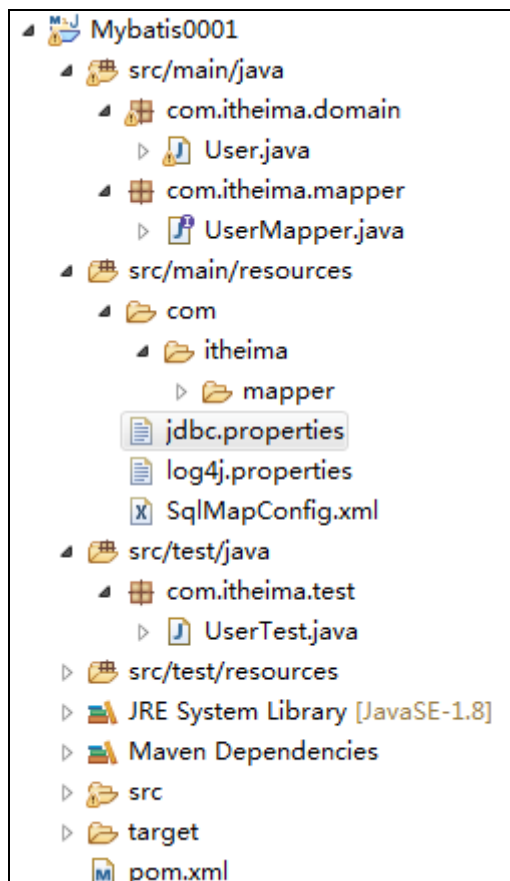
```
SqlMapConfig.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6     <properties resource="jdbc.properties"></properties>
7     <environments default="mysql">
8         <environment id="mysql">
9             <transactionManager type="JDBC" />
10            <dataSource type="pooled">
11                <property name="driver" value="${jdbc.driver}"/>
12                <property name="url" value="${jdbc.url}"/>
13                <property name="username" value="${jdbc.username}"/>
14                <property name="password" value="${jdbc.password}"/>
15            </dataSource>
16        </environment>
17    </environments>
18    <!-- 加载映射文件 -->
19    <mappers>
20        <mapper resource="com/itheima/mapper/UserMapper.xml"/>
21    </mappers>
22 </configuration>
23
```

在这个文件中，我们加载了 jdbc.properties 文件，下面是文件内容：

```
jdbc.properties
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/mybatisdb?characterEncoding=utf-8
3 jdbc.username=root
4 jdbc.password=root
```

项目总体结构如下：





### 3.1.2.7 编写测试类

```
UserTest.java
1 package com.itheima.test;
2
3 import java.io.InputStream;
4
14
15 public class UserTest {
16     @Test
17     public void testMapper() throws Exception {
18         InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
19         SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
20
21         SqlSession session = factory.openSession();
22         UserMapper userMapper = session.getMapper(UserMapper.class);
23         List<User> list = userMapper.findAll();
24         System.out.println(list.size());
25
26
27         session.close();
28         is.close();
29     }
30 }
```

测试效果如下：



```
ts Console Progress Maven Repositories JUnit
午9:57:31)
- Logging initialized using 'class org.apache.ibatis.logging.log4j.Log4jImpl' adapter
- PooledDataSource forcefully closed/removed all connections.
- PooledDataSource forcefully closed/removed all connections.
- PooledDataSource forcefully closed/removed all connections.
- PooledDataSource forcefully closed/removed all connections.
- Opening JDBC Connection
- Created connection 133250414.
- Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f13d6e]
- ==> Preparing: select * from user
- ==> Parameters:
- <==          Total: 11

- Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f13d6e]
- Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f13d6e]
- Returned connection 133250414 to pool.
```

### 3.1.3 小结

通过快速入门示例，我们发现前面的自定义 Mybatis 框架和我们官方的 Mybatis 框架很多是相同的，通过这个推导过程告诉我们，只要基础扎实，我们完全可以运用基础知识，将一个框架的基本原理掌握好。希望大家一定要掌握住 Mybatis 入门示例。