**Mybatis框架课程第一天**

# 框架概述

## 什么是框架

### 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。

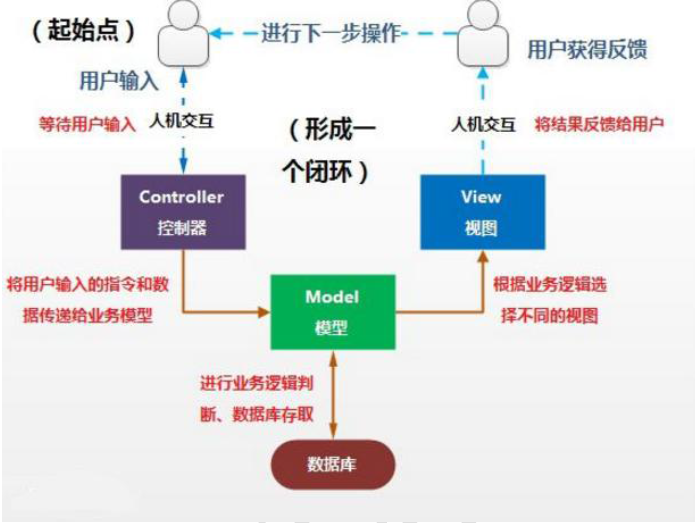
简而言之，框架其实就是某种应用的半成品，就是一组组件，供你选用完成你自己的系统，或者说是使用别人搭好的舞台，你来做表演。框架一般是成熟的，不断升级的软件。

### 框架要解决的问题

框架要解决的最重要的一个问题是技术整合的问题，在JavaEE的 框架中，有着各种各样的技术，不同的软件企业需要从JavaEE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。

### 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。

### 分层开发下的常见框架

#### 解决数据持久化问题的框架

**Mybatis**

MyBatis 本是[apache](https://baike.baidu.com/item/apache/6265" \t "_blank)的一个开源项目[iBatis](https://baike.baidu.com/item/iBatis" \t "_blank), 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。

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Mybatis框架是主流，未来使用Spring Data来实现数据持久化也是一种趋势。

#### 解决WEB层问题的框架

**SpringMVC**

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(一般老项目使用)等。

#### 解决技术整合问题的框架

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。

### 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。

## JDBC编程的分析

### 创建数据库并初始化测试数据

1、创建数据库，数据库名：mybatis

2、将如下Sql语句在数据库中执行，初始化测试数据

|  |
| --- |
| **-- ----------------------------**  **-- Table structure for `user`**  **-- ----------------------------**  **DROP TABLE IF EXISTS `user`;**  **CREATE TABLE `user` (**  **`id` int(11) NOT NULL AUTO\_INCREMENT,**  **`username` varchar(32) NOT NULL COMMENT '用户名称',**  **`birthday` date DEFAULT NULL COMMENT '生日',**  **`sex` char(1) DEFAULT NULL COMMENT '性别',**  **`address` varchar(256) DEFAULT NULL COMMENT '地址',**  **PRIMARY KEY (`id`)**  **) ENGINE=InnoDB AUTO\_INCREMENT=27 DEFAULT CHARSET=utf8;**  **-- ----------------------------**  **-- Records of user**  **-- ----------------------------**  **INSERT INTO `user` VALUES ('1', '张三', '2018-07-10', '1', '北京');**  **INSERT INTO `user` VALUES ('2', '李四', '2018-07-10', '1', '上海');**  **INSERT INTO `user` VALUES ('3', '王五', '2018-07-10', '1', '广州');**  **INSERT INTO `user` VALUES ('4', '王六', '2018-07-10', '1', '深圳');** |

### JDBC程序回顾

|  |
| --- |
| **public static void** main(String[] args) {  Connection connection = **null**;  PreparedStatement preparedStatement = **null**;  ResultSet resultSet = **null**;   **try** {  *// 1、加载数据库驱动* Class.*forName*(**"com.mysql.jdbc.Driver"**);  *// 2、获取数据库链接* connection = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8"**, **"root"**, **"root"**);  *// 3、定义sql语句 ?表示占位符* String sql = **"select** *\** **from user where username = ?"**;  *// 4、获取预处理statement* preparedStatement = connection.prepareStatement(sql);  *// 5、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, **"张三"**);  *// 6、向数据库发出sql执行查询，查询出结果集* resultSet = preparedStatement.executeQuery();  *// 7、解析处理结果集* **while** (resultSet.next()) {  System.***out***.println(resultSet.getString(**"id"**) + **" "** + resultSet.getString(**"username"**));  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  *// 8、释放资源* **if** (resultSet != **null**) {  **try** {  resultSet.close();  } **catch** (SQLException e) {  *//* ***TODO Auto-generated catch block*** e.printStackTrace();  }  }  **if** (preparedStatement != **null**) {  **try** {  preparedStatement.close();  } **catch** (SQLException e) {  *//* ***TODO Auto-generated catch block*** 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。

### 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。

下面我们将通过自己定义编写一个持久层框架的方式来解决Jdbc编程存在的部分问题。

# 自定义Mybatis框架

## 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）、构建者模式（Builder模式）、反射、Xml解析，数据库元数据等。

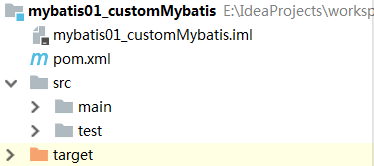
## 创建Maven工程并引入相关坐标

### 开发环境的准备及统一

1. Jdk环境：JDK 1.8 64bit
2. Maven环境：MAVEN 3.3.9

### 创建Maven工程并引入坐标

第一步：在Idea中创建一个Maven工程，名称为mybatis01\_customMybatis



第二步：在pom.xml中引入所需坐标

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>dom4j</**groupId**>  <**artifactId**>dom4j</**artifactId**>  <**version**>1.6.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>jaxen</**groupId**>  <**artifactId**>jaxen</**artifactId**>  <**version**>1.1.6</**version**>  </**dependency**> </**dependencies**> |

## 自定义Mybatis框架配置文件结构约定

### Sql文件约定

作为一个框架，通常都会定义自己的配置文件（可以有多个），用户只需要按照框架的要求编写相应的配置文件就可以了。因为我们自定义的Mybatis框架是一个持久层框架（操作数据库的），所以需要配置文件用于存放Sql语句，此类配置文件通常称为Sql映射文件。配置文件的名称可以任意，但是最好有一个统一的规范，方便见名知意，此处我们的约定是XXMapper.xml（XX为该文件所对应的表名，如User表为UserMapper.xml）。具体格式如下：

|  |
| --- |
| *<?*xml version="1.0" encoding="utf-8" *?>* <mapper namespace="com.itheima.pojo.User">  <select id="findAllUsers" resultType="com.itheima.pojo.User">  select \* from user  </select> </mapper> |

### 框架核心配置文件约定

除了映射文件（存放SQL语句的文件），还需要有一个框架的核心配置文件，主要是存放数据源相关信息的。因为不同的用户使用的数据库类型不同，数据库名称不同，使用的数据库用户和密码也都不相同，所以这些信息应该是由用户自己指定。用户通过这个核心配置文件就可以指定数据源相关信息了。此处我们约定配置文件名称为SqlMapConfig.xml。具体格式如下：

|  |
| --- |
| *<?*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/mybatis?characterEncoding=utf8"></property>  <property name="username" value="root"></property>  <property name="password" value="root"></property>  </dataSource>  </environment>  </environments>  <mappers>  <mapper resource="UserMapper.xml"/>  </mappers> </configuration> |

## 核心组件开发

### Mapper开发

通过前面Jdbc存在问题的分析，我们已经知道了要将SQL语句放入配置文件中，这样将来修改SQL语句会比较方便，但放在配置文件中的SQL语句还需要读取出来，这样我们就可以基于面向对象思维定义一个Mapper类，用于将配置文件中的SQL语句保存起来，使用时更方便。定义如下：

|  |
| --- |
| **package** frame.pojo; **public class** Mapper {  **private** String **sql**;  **private** String **resultType**;   **public** String getSql() {  **return sql**;  }   **public void** setSql(String sql) {  **this**.**sql** = sql;  }   **public** String getResultType() {  **return resultType**;  }   **public void** setResultType(String resultType) {  **this**.**resultType** = resultType;  } } |

### Configuration开发

我们以面向对象思维操作自定义框架的核心配置文件时，需要有一个实体类与之对应。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。定义如下：

|  |
| --- |
| **package** frame.pojo; **import** java.util.HashMap; **import** java.util.Map;  **public class** Configuration {  **private** String **driver**;  **private** String **url**;  **private** String **username**;  **private** String **password**;  **private** Map<String,Mapper> **mappers** = **new** HashMap<>();   **public** Map<String, Mapper> getMappers() {  **return mappers**;  }   **public void** setMappers(Map<String, Mapper> mappers) {  **this**.**mappers** = mappers;  }   **public** String getDriver() {  **return driver**;  }   **public void** setDriver(String driver) {  **this**.**driver** = driver;  }   **public** String getUrl() {  **return url**;  }   **public void** setUrl(String url) {  **this**.**url** = url;  }   **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;  } } |

### SqlSession接口开发

我们开发的是框架，因此封装的对象是不固定的，因此这里使用泛型<T> List<T>。此处仅示例一个查询全部。

|  |
| --- |
| **package** frame.core;  **import** java.util.List; **public interface** SqlSession {  **public** <T> List<T> selectList(String mapperId) **throws** Exception; } |

### SqlSessionImpl实现类开发

|  |
| --- |
| **package** frame.core;  **import** frame.core.Executor; **import** frame.pojo.Configuration; **import** java.util.List;  **public class** SqlSessionImpl **implements** SqlSession {   **private** Configuration **configuration**;   **public void** setConfiguration(Configuration configuration) {  **this**.**configuration** = configuration;  }   @Override  **public** <T> List<T> selectList(String mapperId) **throws** Exception  {  Executor executor = **new** Executor(**configuration**);  **return** executor.executeQuery(mapperId) ;  } } |

### Executor执行器开发

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。

|  |
| --- |
| **package** frame.core;  **import** frame.pojo.Configuration; **import** frame.pojo.Mapper;  **import** java.lang.reflect.Method; **import** java.sql.\*; **import** java.util.ArrayList; **import** java.util.List;  **public class** Executor {   **private** Configuration **configuration**;   **public** Executor(Configuration configuration) {  **this**.**configuration** = configuration;  }   **public** <T> List<T> executeQuery(String mapperId) **throws** Exception {  List<T> list = **new** ArrayList<>();   Connection connection = **null**;  PreparedStatement preparedStatement = **null**;  ResultSet resultSet = **null**;  **try** {  Class.*forName*(**configuration**.getDriver());  connection = DriverManager.*getConnection*(**configuration**.getUrl(),**configuration**.getUsername(),**configuration**.getPassword());  Mapper mapper = **configuration**.getMappers().get(mapperId);  String sql = mapper.getSql();  preparedStatement = connection.prepareStatement(sql);  resultSet = preparedStatement.executeQuery();  String resultType = mapper.getResultType();  Class clazz = Class.*forName*(resultType);  Method[] mehtods = clazz.getMethods();  List<String> columnNames = **new** ArrayList<>();  ResultSetMetaData metaData = resultSet.getMetaData();  **int** count = metaData.getColumnCount();  **for**(**int** i = 1; i <= count; i++) {  columnNames.add(metaData.getColumnLabel(i));  }  Object object = **null**;  **while** (resultSet.next()) {  object = clazz.newInstance();**for**(String columnName : columnNames) {  **for**(Method method: mehtods) {  **if**(method.getName().equalsIgnoreCase(**"set"** + columnName)) {  method.invoke(object,resultSet.getObject(columnName));  }  }  }  list.add((T) object);  }  } **catch** (Exception e) {  **throw** e;  } **finally** {  **if**(resultSet != **null**) {  resultSet.close();  }  **if**(preparedStatement != **null**) {  preparedStatement.close();  }  **if**(connection != **null**){  connection.close();  }  }  **return** list;  } } |

### SqlSessionFactory开发

SqlSessionFactory的开发基于工厂模式，工厂模式是我们最常用的用来实例化对象的设计模式，是用工厂方法代替new操作的一种模式。创建对象的时候使用工厂模式会带来更大的可扩展性和尽量少的修改量。

|  |
| --- |
| **package** frame.factory;  **import** frame.core.SqlSession; **import** frame.core.SqlSessionImpl; **import** frame.pojo.Configuration; **import** frame.pojo.Mapper; **import** org.dom4j.Document; **import** org.dom4j.DocumentException; **import** org.dom4j.Element; **import** org.dom4j.io.SAXReader;  **import** java.io.InputStream; **import** java.util.List;  **public class** SqlSessionFactory {   **public void** setInputStream(InputStream inputStream) {  **this**.**inputStream** = inputStream;  }   **private** InputStream **inputStream**;   **public** SqlSession openSession() {  SqlSessionImpl sqlSession = **new** SqlSessionImpl();  Configuration configuration = loadXmlConfig();  sqlSession.setConfiguration(configuration);  **return** sqlSession;  }   **private** Configuration loadXmlConfig() {  Configuration configuration = **new** Configuration();  **try** {  SAXReader saxReader = **new** SAXReader();  Document document = saxReader.read(**inputStream**);  Element root = document.getRootElement();  List<Element> list = root.selectNodes(**"//property"**);  **for** (**int** i = 0; i < list.size(); i++) {  Element element = list.get(i);  String name = element.attributeValue(**"name"**);  String value = element.attributeValue(**"value"**);  **if**(**"driver"**.equalsIgnoreCase(name)) {  configuration.setDriver(value);  }  **if**(**"url"**.equalsIgnoreCase(name)){  configuration.setUrl(value);  }  **if**(**"username"**.equalsIgnoreCase(name)) {  configuration.setUsername(value);  }  **if**(**"password"**.equalsIgnoreCase(name)) {  configuration.setPassword(value);  }  }   List<Element> mappers = root.selectNodes(**"//mapper"**);   **for** (**int** i = 0; i < mappers.size(); i++) {  Element element = mappers.get(i);  String mapperPath = element.attributeValue(**"resource"**);  *// 读取mapper文件* loadXmlMapper(configuration,mapperPath);  }  } **catch** (Exception e) {  e.printStackTrace();  }  **return** configuration;  }   **private void** loadXmlMapper(Configuration configuration,String mapperPath) {  SAXReader saxReader = **new** SAXReader();  **try** {  Document document = saxReader.read(**this**.getClass().getClassLoader().getResourceAsStream(mapperPath));  Element root = document.getRootElement();  String namespace = root.attributeValue(**"namespace"**);  List<Element> list = root.selectNodes(**"//select"**);  **for** (**int** i = 0; i < list.size(); i++) {  Element element = list.get(i);  String id = element.attributeValue(**"id"**);  String resultType = element.attributeValue(**"resultType"**);  String sql = element.getText();  Mapper mapper = **new** Mapper();  mapper.setResultType(resultType);  mapper.setSql(sql);  configuration.getMappers().put(namespace + **"."** + id,mapper);  }  } **catch** (DocumentException e) {  e.printStackTrace();  }  } } |

注：XML 文件解析采用的是Dom4j结合xpath实现。

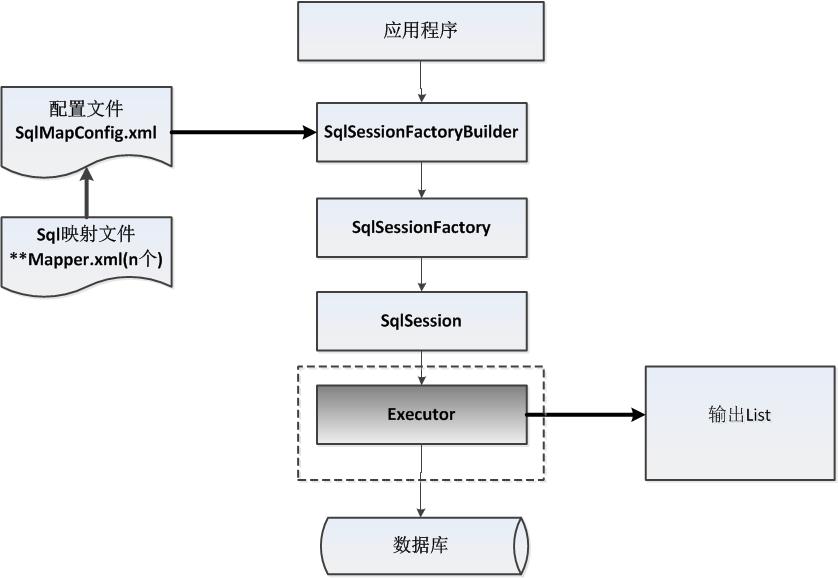
### SqlSessionFactoryBuilder开发

使用构建者模式，应对不同的场景变化。

|  |
| --- |
| **package** frame.factory;  **import** frame.factory.SqlSessionFactory; **import** java.io.InputStream; **public class** SqlSessionFactoryBuilder {   **public** SqlSessionFactory build() {  InputStream inputStream = **this**.getClass().getClassLoader().getResourceAsStream(**"SqlMapConfig.xml"**);  SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactory();  sqlSessionFactory.setInputStream(inputStream);  **return** sqlSessionFactory;  }   **public** SqlSessionFactory build(String filename) {  InputStream inputStream = **this**.getClass().getClassLoader().getResourceAsStream(filename);  SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactory();  sqlSessionFactory.setInputStream(inputStream);  **return** sqlSessionFactory;  }   **public** SqlSessionFactory build(InputStream inputStream) {  SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactory();  sqlSessionFactory.setInputStream(inputStream);  **return** sqlSessionFactory;  } } |

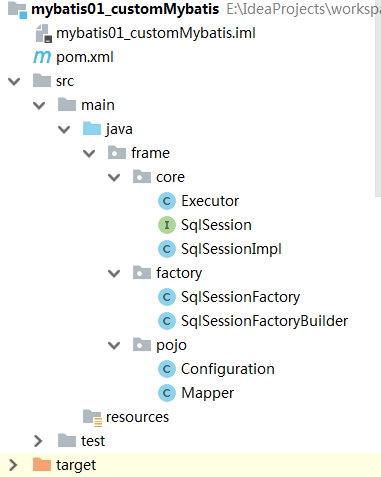
### 小结

自定义Mybatis框架架构图如下

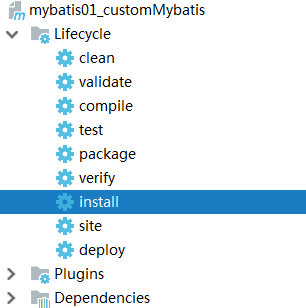


## 安装到Maven仓库

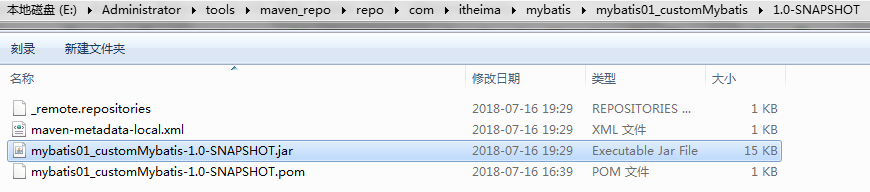
1、经过上面的开发过程，工程的结构如下：



2、使用Idea自带的打包和安装功能将当前工程打成Jar包安装到本地Maven仓库（双击install图标即可）：



3、查看本地Maven仓库



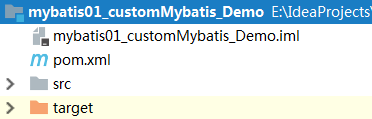
# 基于自定义Mybatis框架实现数据库操作

### 功能需求

基于自定义Mybatis框架和已有的Mysql数据库Mybatis，查询所有用户信息。

### 创建工程并引入自定义Mybatis框架的坐标

第一步：创建一个Maven工程mybatis01\_customMybatis\_Demo



第二步：修改mybatis01\_customMybatis\_Demo工程的pom.xml文件，添加自定义Mybatis框架的Maven坐标和MySQL数据库驱动的坐标

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>com.itheima.mybatis</**groupId**>  <**artifactId**>mybatis01\_customMybatis</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.46</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  </**dependency**> </**dependencies**> |

### 创建Pojo

|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **import** java.util.Date;  **public class** User {  **private** Integer **id**;  **private** String **username**;  **private** String **sex**;  **private** Date **birthday**;  **private** String **address**;   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", sex='"** + **sex** + **'\''** +  **", birthday="** + **birthday** +  **", address='"** + **address** + **'\''** +  **'}'**;  } } |

### 创建Sql文件UserMapper.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* <**mapper namespace="test"**> *<!-- select 查询 -->* <**select id="queryUserList" resultType="com.itheima.mybatis.pojo.User"**>  select \* from user   </**select**>  </**mapper**> |

### 创建核心配置文件SqlMapConfig.xml

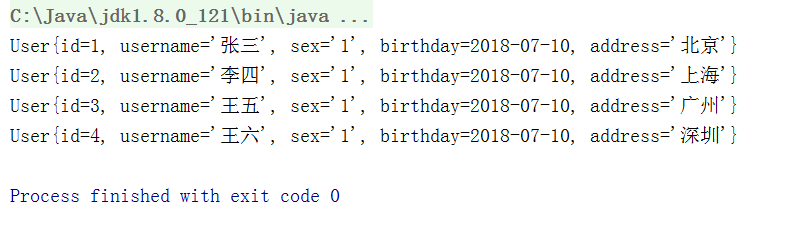
|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* <**configuration**>   <**environments default="development"**>   <**environment id="development"**>   <**transactionManager type="JDBC"** />   <**dataSource type="POOLED"**>   <**property name="driver" value="com.mysql.jdbc.Driver"** ></**property**>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

### 测试

#### 测试程序

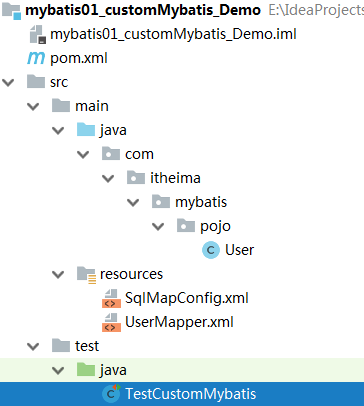
|  |
| --- |
| @Test **public void** testQueryUserList() **throws** Exception {  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(**"SqlMapConfig.xml"**);  SqlSession sqlSession = sqlSessionFactory.openSession();  List<User> list = sqlSession.selectList(**"test.queryUserList"**);  **for** (**int** i = 0; i < list.size(); i++) {  User user = list.get(i);  System.***out***.println(user);  } } |

#### 测试效果



### 工程结构

测试工程结构如下：



### 小结

以上通过自定义Mybatis框架的学习，我们将前面的基础知识很好的结合在一起，并且强化了我们对设计模式的使用。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。

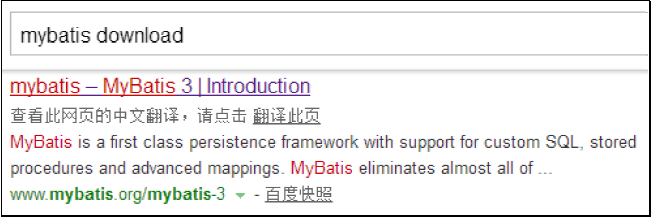
# Mybatis框架快速入门

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。

## Mybatis框架开发的准备

### 官网下载Mybatis框架

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



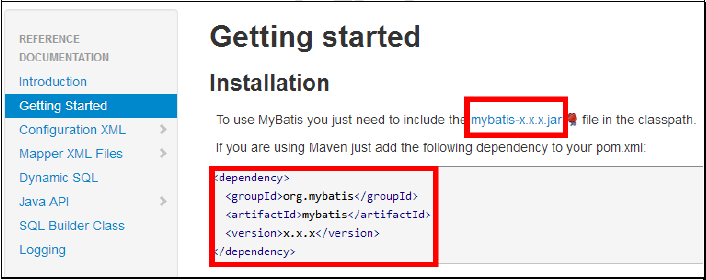
进入选择语言的界面，进入中文版本的开发文档



我们可以看到熟悉的中文开发文档了



下载相关的Jar包或Maven开发的坐标



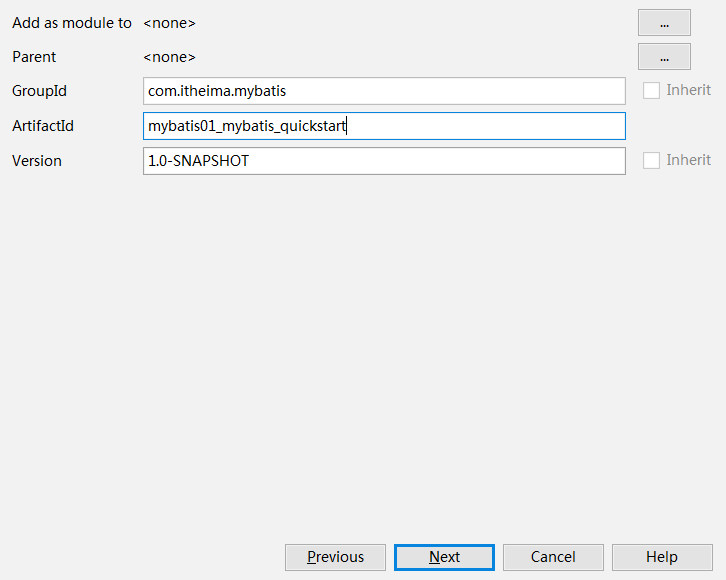
下载的zip文件如下（ 我们的资料文件夹）：



我们所使用的Mybatis版本是3.4.5版本。

### 搭建Mybatis开发环境

#### 创建Maven工程



#### 引入Maven坐标

在pom.xml文件中添加坐标，如下

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis</**artifactId**>  <**version**>3.4.5</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.46</**version**>  </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.17</**version**>  </**dependency**> </**dependencies**> |

#### 编写log4j.properties

在resources下创建log4j.properties，内容如下：

|  |
| --- |
| # 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作为输出日志组件。

#### 编写User实体类

|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **import** java.util.Date;  **public class** User {  **private** Integer **id**;  **private** String **username**;  **private** String **sex**;  **private** Date **birthday**;  **private** String **address**;   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", sex='"** + **sex** + **'\''** +  **", birthday="** + **birthday** +  **", address='"** + **address** + **'\''** +  **'}'**;  } } |

#### 编写UserMapper.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>*** <**mapper namespace="test"**>  *<!-- select 查询 -->* <**select id="queryUserList" resultType="com.itheima.mybatis.pojo.User"**>  select *\** from user  </**select**> </**mapper**> |

#### 编写SqlMapConfig.xml

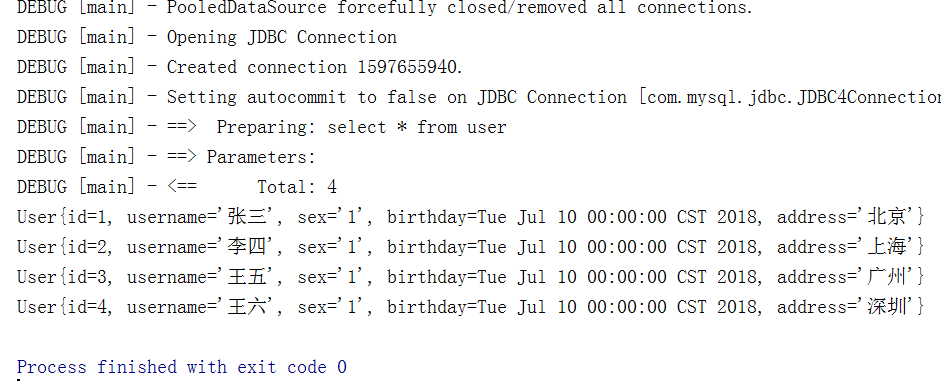
|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"*>*** <**configuration**>  <**environments default="development"**>  <**environment id="development"**>  <**transactionManager type="JDBC"** />  <**dataSource type="POOLED"**>  <**property name="driver" value="com.mysql.jdbc.Driver"** />  <**property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis?characterEncoding=utf8"** />  <**property name="username" value="root"** />  <**property name="password" value="root"** />  </**dataSource**>  </**environment**>  </**environments**>  <**mappers**>  <**mapper resource="UserMapper.xml"**></**mapper**>  </**mappers**> </**configuration**> |

### 测试

#### 测试程序

|  |
| --- |
| @Test **public void** testQueryUserList() **throws** Exception {  InputStream inputStream = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  List<User> list = sqlSession.selectList(**"test.queryUserList"**);  **for** (**int** i = 0; i < list.size(); i++) {  User user = list.get(i);  System.***out***.println(user);  } } |

#### 测试效果



## 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。