**Mybatis第一天**

上海千锋 文威

**课程总览：**

我们mybatis的课程一共分为4天。

第一天：

Mybatis入门

Mybatis的基本概述

Mybatis的基本环境搭建

Mybatis的入门案例

自定义Mybatis框架(主要是为了让大家了解Mybatis的执行细节)

第二天：

Mybatis基本使用

Mybatis单表的crud操作

Mybatis的参数和返回值

Mybatis的dao编写

Mybatis的配置细节--几个标签的使用

第三天：

Mybatis的深入和多表

Mybatis 的事务控制及设计方法

Mybatis的多表查询

一对多

多对多

第四天：

Mybatis的缓存和注解开发

Mybatis的加载时机(查询时机)

Mybatis的一级缓存和二级缓存

Mybatis 的注解开发

单表crud

多表查询

## 1.1 框架的基本概述

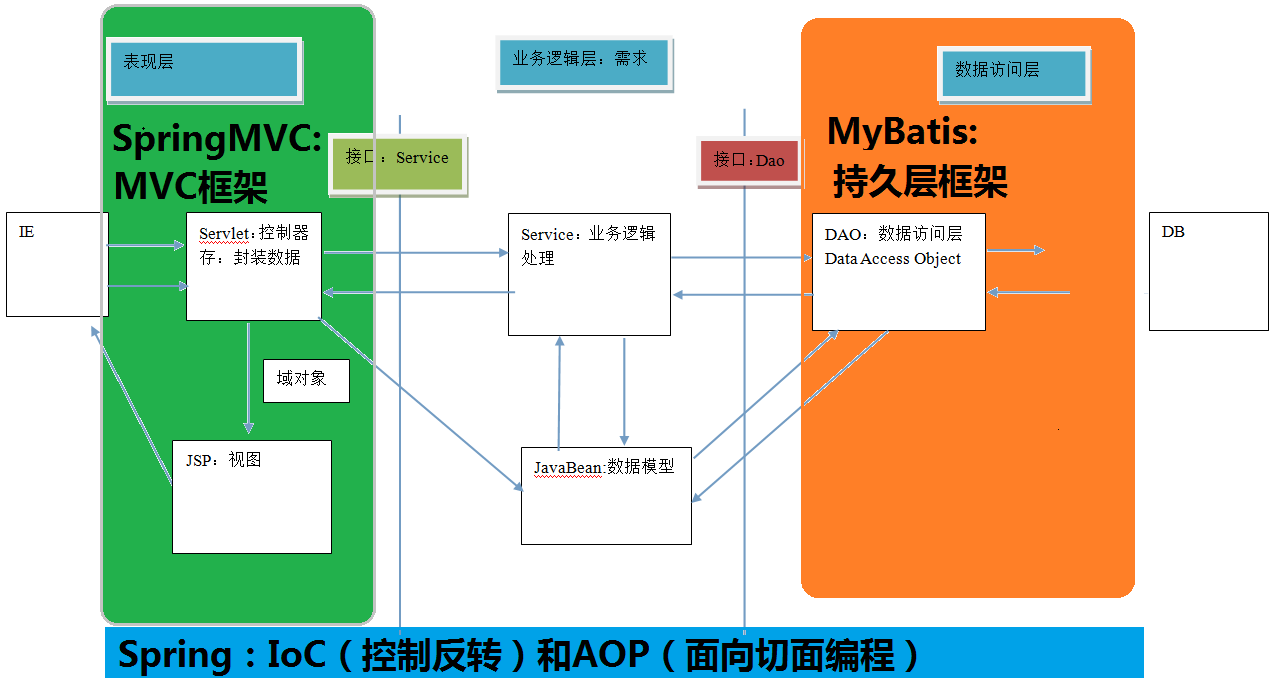
### 1.1.1 什么是框架

框架就是一套解决方案，**不同的框架解决不同的问题**。框架帮助我们封装了很多细节，使用开发者使用极简单的方式就可以实现功能，大大的提高了开发效率。

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是技术整合的问题，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。



我们今天要学习的就是基于数据访问层操作的Mybatis框架。

我们先回顾一下我们学习过的基于操作数据访问层的技术解决方案：

Jdbc:Connection StateMent PreparedStatement

JdbcTemplate ,Spring对jdbc简单封装 。

Apache的DbUtils，和Jdbctemplate很像，也是对jdbc的简单封装。

那我们为什么还需要学习Mybatis呢?

其实上面的技术都不是框架，Jdbc只能说是操作数据访问层的一种规范。而JdbcTemplate技术和DbUtils技术都是参照了Jdbc技术进行封装的**组件**，这种封装是粗狂的，不够细致，在使用的过程中还有大量的细节需要我们去处理。

那我们为什么需要学习Mybatis?接下来我们分析一段代码。

### 1.1.3 传统Jdbc代码问题分析

|  |
| --- |
| **public class** TestMysql {  **public static void** main(String[] args) {  **try** {  *//加载数据库驱动* Class.forName(**"com.mysql.jdbc.Driver"**);  *//创建连接对象* String url = **"jdbc:mysql://192.168.10.137:3306/ssm"**;  Connection connection = DriverManager.getConnection(url, **"root"**, **"Admin123!"**);  *//准备sql语句* String sql = **"select \* from account"**;  *//准备PreparedStatement对象* PreparedStatement pst = connection.prepareStatement(sql);  *//执行sql语句* ResultSet rs = pst.executeQuery();  *//遍历结果集* **while**(rs.next()){  Integer id = rs.getInt(**"id"**);  String name = rs.getString(**"name"**);  Double money = rs.getDouble(**"money"**);  System.out.println(id + **" "** + name + **" "** + money);  }  *//关闭资源* rs.close();  pst.close();  connection.close();  } **catch** (Exception e) {  e.printStackTrace();  }  } } |

那么这段代码有什么问题呢?

1、数据库连接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库连接池可解决此问题。

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。

### 1.1.4 Mybatis框架基本概述

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。

## 1.2 Mybatis环境搭建

资料准备：在我们mybatis课程的学习中，我们会用到一些数据表，相关对应的sql语句，我们已经准备好了。

|  |
| --- |
| 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` datetime 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 DEFAULT CHARSET=utf8;  insert into `user`(`id`,`username`,`birthday`,`sex`,`address`) values (41,'老王','2018-02-27 17:47:08','男','上海'),(42,'小二王','2018-03-02 15:09:37','女','上海宝山'),(43,'小二王','2018-03-04 11:34:34','女','上海千锋'),(45,'千锋教育','2018-03-04 12:04:06','男','北京昌平'),(46,'王','2018-03-07 17:37:26','男','北京'),(48,'小峰','2018-03-08 11:44:00','女','上海');  DROP TABLE IF EXISTS `account`;  CREATE TABLE `account` (  `ID` int(11) NOT NULL COMMENT '编号',  `UID` int(11) default NULL COMMENT '用户编号',  `MONEY` double default NULL COMMENT '金额',  PRIMARY KEY (`ID`),  KEY `FK\_Reference\_8` (`UID`),  CONSTRAINT `FK\_Reference\_8` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  insert into `account`(`ID`,`UID`,`MONEY`) values (1,41,1000),(2,45,1000),(3,41,2000);  DROP TABLE IF EXISTS `role`;  CREATE TABLE `role` (  `ID` int(11) NOT NULL COMMENT '编号',  `ROLE\_NAME` varchar(30) default NULL COMMENT '角色名称',  `ROLE\_DESC` varchar(60) default NULL COMMENT '角色描述',  PRIMARY KEY (`ID`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  insert into `role`(`ID`,`ROLE\_NAME`,`ROLE\_DESC`) values (1,'院长','管理整个学院'),(2,'总裁','管理整个公司'),(3,'校长','管理整个学校');  DROP TABLE IF EXISTS `user\_role`;  CREATE TABLE `user\_role` (  `UID` int(11) NOT NULL COMMENT '用户编号',  `RID` int(11) NOT NULL COMMENT '角色编号',  PRIMARY KEY (`UID`,`RID`),  KEY `FK\_Reference\_10` (`RID`),  CONSTRAINT `FK\_Reference\_10` FOREIGN KEY (`RID`) REFERENCES `role` (`ID`),  CONSTRAINT `FK\_Reference\_9` FOREIGN KEY (`UID`) REFERENCES `user` (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  insert into `user\_role`(`UID`,`RID`) values (41,1),(45,1),(41,2); |

### 1.2.1 创建maven工程，添加对应坐标

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis</**artifactId**>  <**version**>3.4.5</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.10</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  <**scope**>runtime</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.12</**version**>  </**dependency**> </**dependencies**> |

### 1.2.2 编写实体类

|  |
| --- |
| **public class** User **implements** Serializable {  **private** Integer **id**;  **private** String **username**;  **private** Date **birthday**;  **private** String **sex**;  **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** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", birthday="** + **birthday** +  **", sex='"** + **sex** + **'\''** +  **", address='"** + **address** + **'\''** +  **'}'**;  } } |

### 1.2.4 编写mybatis的核心配置文件 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**>  *<!-- 配置 mybatis 的环境 -->* <**environments default="mysql"**>  *<!-- 配置 mysql 的环境 -->* <**environment id="mysql"**>  *<!-- 配置事务的类型 -->* <**transactionManager type="JDBC"**></**transactionManager**>  *<!-- 配置连接数据库的信息：用的是数据源(连接池) -->* <**dataSource type="POOLED"**>  <**property name="driver" value="com.mysql.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://192.168.10.137:3306/mybatis"**/>  <**property name="username" value="root"**/>  <**property name="password" value="Admin123!"**/>  </**dataSource**>  </**environment**>  </**environments**>  *<!-- 告知 mybatis 映射配置的位置 -->* <**mappers**>  <**mapper resource="com/qf/dao/UserDao.xml"**/>  </**mappers**> </**configuration**> |

### 1.2.5 编写持久层接口对应的映射文件 UserDao.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="com.qf.pojo.dao.UserDao"**>  *<!--  配置查询所有操作  id:方法名称，和接口名称需要对应   resultType:返回值类型  -->* <**select id="findAll" resultType="com.qf.pojo.User"**>  select *\** from user  </**select**> </**mapper**> |

**搭建环境总结:**

1、在创建UserDao.java和UserDao.xml的时候，接口的名称必须和映射文件的名称保持一样，否则会报错。有的时候别人也定义接口名称和映射文件也叫mapper。比如UserMapper.java或者UserMapper.xml。

2、在resources目录中创建的包的目录是创建directory。和java的package一样，也需要是三级目录，可以一级一级的创建。

3、接口文件和映射文件的目录必须要保持相同。

4、映射文件的mapper标签，namespace属性的取值必须要保持唯一，必须是接口名称的全限定名。

5、映射配置文件的操作配置(比如select标签，update标签、insert标签、delete标签)的id值必须和接口的名称保持一致。

**我们只有遵循了上面的3 4 5点之后，我们就无需写dao接口的实现类。接口的实现过程就由mybatis框架为我们实现。**

### 1.2.3 编写持久层接口

|  |
| --- |
| **public interface** UserDao {  **public** List<User> findAll(); } |

### 1.2.6 测试

|  |
| --- |
| */\*\*  \* 测试入门案例  \*/* **public class** MybatisTest {  **public static void** main(String[] args) **throws** Exception {  *//1.读取配置文件* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//2.创建 SqlSessionFactory 的构建者对象* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  *//3.使用构建者创建工厂对象 SqlSessionFactory* SqlSessionFactory factory = builder.build(in);  *//4.使用 SqlSessionFactory 生产 SqlSession 对象* SqlSession session = factory.openSession();  *//5.使用 SqlSession 创建 dao 接口的代理对象* UserDao userDao = session.getMapper(UserDao.**class**);  *//6.使用代理对象执行查询所有方法* List<User> users = userDao.findAll();  **for**(User user : users) {  System.***out***.println(user);  }  *//7.释放资源* session.close();  in.close();  } } |

## 1.3 自定义Mybatis框架(XML)

我们已经通过案例体验到了mybatis的魅力。现在我们来看它的测试类，有几个对象我们需要搞清楚他们的作用，进而需要理解mybatis的整个工作流程和执行原理。

* Resources

加载配置文件，有一种是使用类加载进行加载，我们通过这个类的类加载器进行资源的加载。

* SqlSessionFactoryBuilder

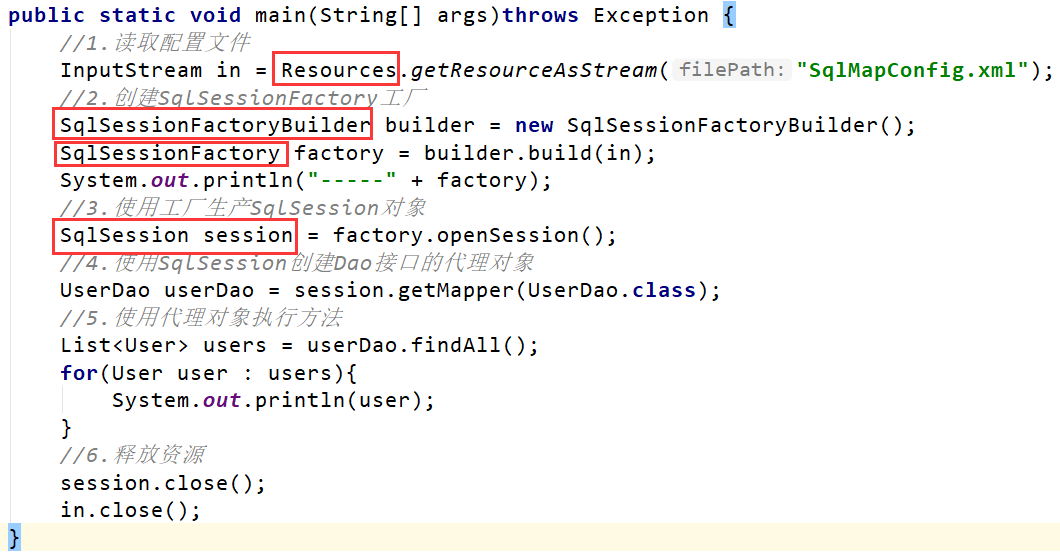
构建SqlSessionFactory工厂对象需要的对象。采用了**构建者模式**，屏蔽了对象构建的细节。

* SqlSessionFactory

创建SqlSession对象所用。使用**工厂模式**创建，目的就是解耦合。

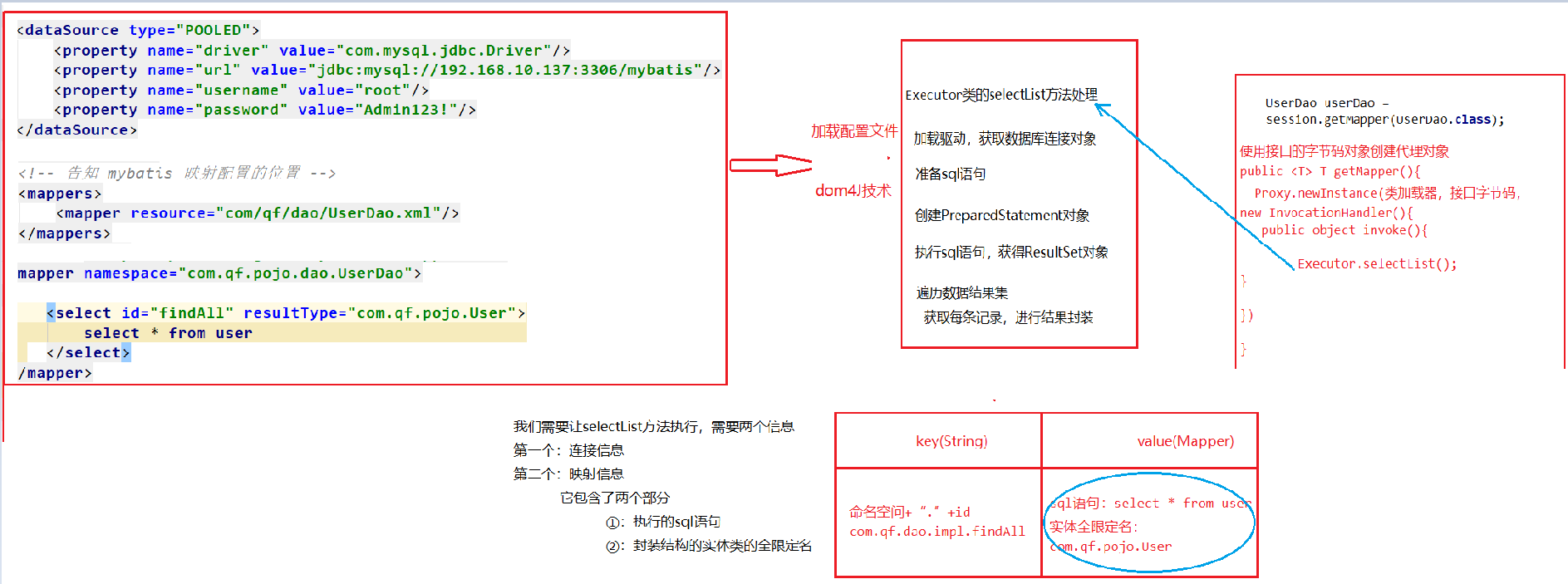
* SqlSession

创建代理对象，调用接口里面的方法。使用了**代理模式**。



下面我们就自己来手写mybatis框架，体验其工作原理。

### 1.3.1 流程分析



### 1.3.2 搭建环境

拷贝上一个工程代码，去掉mybatis的依赖。

依赖如下：

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.12</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.10</**version**>  </**dependency**>  **<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**> |

我们需要两个工具类

一个是XMLConfigBuilder类，主要是用来解析XMl配置文件。

第二个是DataSourceUtil类，主要是用来获取数据库连接对象。

第三个是Executor类，主要是用来获取数据结果集，封装我们想要的数据。

### 1.3.3 编码实现

* 编写资源加载类。使用类加载器加载配置资源

|  |
| --- |
| **public class** Resources {  *//根据文件名称，加载类路径下面的配置文件* **public static** InputStream getResourceAsStream(String filePath){  **return** Resources.**class**.getClassLoader().getResourceAsStream(filePath);  } } |

* 编写SqlSessionFactoryBulider类

作用：加载配置资源，并将配置资源封装成Configuration对象，并且将该资源对象传到工厂对象中

|  |
| --- |
| **public class** SqlSessionFactoryBuilder {  */\*\*  \* 构建SqlSessionFactory对象  \** ***@param in*** *\** ***@return*** *\*/* **public** SqlSessionFactory build(InputStream in){  Configuration configuration = XMLConfigBuilder.*loadConfiguration*(in);  **return new** DefaultSqlSessionFactory(configuration);  } } |

此时没有配置类?那我们就创建。

* 创建Configuration配置类

|  |
| --- |
| **public class** Configuration {   **private** String **driver**;  **private** String **url**;  **private** String **username**;  **private** String **password**;   **private** Map<String,Mapper> **mappers** = **new** HashMap<String,Mapper>();   **public** Map<String, Mapper> getMappers() {  **return mappers**;  }   **public void** setMappers(Map<String, Mapper> mappers) {  **this**.**mappers**.putAll(mappers);*//此处需要使用追加的方式* }   **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;  } } |

配置类里面的Mapper是用来干嘛的?前面我们分析过，Mapper是用来封装sql语句和查询结果集的实体全限定名的。

|  |
| --- |
| **public class** Mapper {   **private** String **queryString**;*//SQL* **private** String **resultType**;*//实体类的全限定类名* **public** String getQueryString() {  **return queryString**;  }   **public void** setQueryString(String queryString) {  **this**.**queryString** = queryString;  }   **public** String getResultType() {  **return resultType**;  }   **public void** setResultType(String resultType) {  **this**.**resultType** = resultType;  } } |

也没有工厂?那我们就创建工厂。

* 创建SqlSessionFactory接口

|  |
| --- |
| **public interface** SqlSessionFactory {  *//获取SQLSession对象* **public** SqlSession openSession(); } |

由于工厂的类型也可以多样化定义，所以我们把工厂定义为接口，以后想设计什么样的工厂，我们只需要实现这个接口就可以了。

|  |
| --- |
| **public class** DefaultSqlSessionFactory **implements** SqlSessionFactory {   **private** Configuration **cfg**;   **public** DefaultSqlSessionFactory(Configuration cfg) {  **this**.**cfg** = cfg;  }   */\*\*  \* 获取一个SqlSession对象  \** ***@return*** *\*/* **public** SqlSession openSession() {  **return new** DefaultSqlSession(**cfg**);  } } |

**这里设计构造函数的作用是把SqlSessionFactoryBuilder类里面已经加载好的Configuration对象传入到DefaultSqlSessionFactory里面来。**

* 定义SqlSession对象

|  |
| --- |
| **public interface** SqlSession {  *//获取代理对象* **public** <T> T getMapper(Class<T> tClass);  *//释放资源* **void** close(); } |

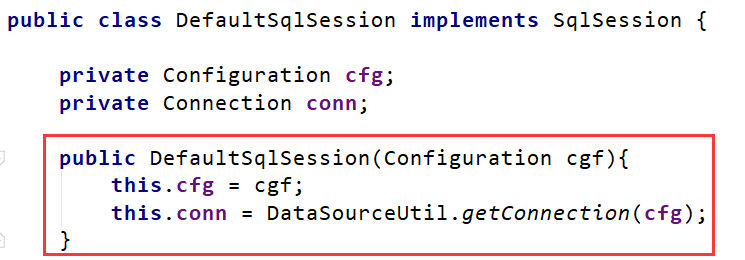
为了提高这个方法的可重用性，这个方法定义为泛型方法。

由于这里面获取代理对象的方式有多种方式可以实现，所以SqlSession我们也可以定义为接口。以后你想用什么方式获取代理对象，只需要实现这个接口即可。

我们创建这个接口的实现类 DefaultSqlSession

|  |
| --- |
| **public class** DefaultSqlSession **implements** SqlSession {   **private** Configuration **cfg**;  **private** Connection **conn**;   **public** DefaultSqlSession(Configuration cgf){  **this**.**cfg** = cgf;  **this**.**conn** = DataSourceUtil.*getConnection*(**cfg**);  }    */\*  \* 创建代理对象  \*/* **public** <T> T getMapper(Class<T> tClass) {  **return (T)Pr*oxy.newProxyInst*ance(tClass.getClassLoader(),  new Class[]{tClass},  new ProxyFactory(cfg.getMappers(),conn));**  }   **public void** close() {  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  } } |

红色部分是在产生代理对象。其中的第三个参数，这个ProxyFactory我们实现了InvocationHandler接口。目的就是为了对接口的方法进行增强！！！而我们之前分析的selectList就是增强的方法！！！然而Executor类里面的selectList方法执行需要两个参数。分别是Map<String,Mapper> mappers,和Connection conn。所以需要将这个两个参数准备好。而Map<String,Mapper> mappers是定义在Configuration类里面的。可以通过Configuration直接获取。而Configuration对象从SqlSessionFactoryBuilder就一直传递过来啦。所以我们只需要在DefaultSqlSession构造函数里面初始化连接对象即可。



* 定义ProxyFactory类

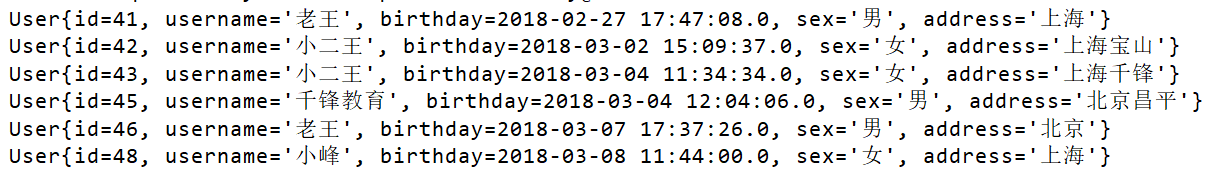
|  |
| --- |
| **public class** ProxyFactory **implements** InvocationHandler {   **private** Map<String, Mapper> **mappers**;  **private** Connection **conn**;  **public** ProxyFactory(Map<String, Mapper> mappers, Connection conn){  **this**.**mappers** = mappers;  **this**.**conn** = conn;  }   **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *//1.获取方法名* String methodName = method.getName();  *//2.获取方法所在类的名称* String className = method.getDeclaringClass().getName();  *//3.组合key* String key = className+**"."**+methodName;  *//4.获取mappers中的Mapper对象* Mapper mapper = **mappers**.get(key);  *//5.判断是否有mapper* **if**(mapper == **null**){  **throw new** IllegalArgumentException(**"传入的参数有误"**);  }  *//6.调用工具类执行查询所有* **return new** Executor().selectList(mapper,**conn**);  } } |

最后目录如下:



* 测试

|  |
| --- |
| **public static void** main(String[] args)**throws** Exception {  *//1.读取配置文件* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//2.创建SqlSessionFactory工厂* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory factory = builder.build(in);  System.***out***.println(**"-----"** + factory);  *//3.使用工厂生产SqlSession对象* SqlSession session = factory.openSession();  *//4.使用SqlSession创建Dao接口的代理对象* UserDao userDao = session.getMapper(UserDao.**class**);  *//5.使用代理对象执行方法* List<User> users = userDao.findAll();  **for**(User user : users){  System.***out***.println(user);  }  *//6.释放资源* session.close();  in.close(); } |



到这里我们基于XMl手写Mybatis框架完成。

## 1.4 总结

