

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

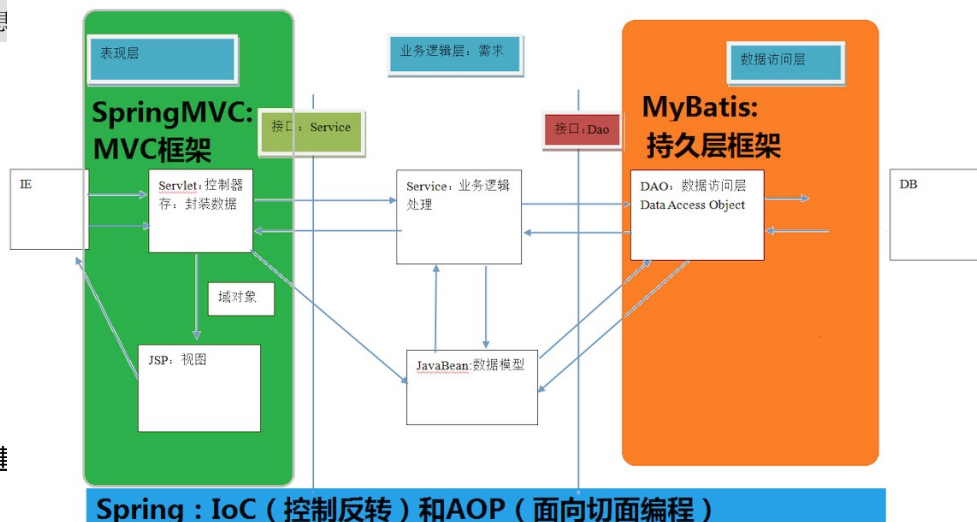
框架要解决的最重要的一个问题是技术整合的问题，在 J2EE 的框架中，有着各种各样的技术，不同的软件企业需要从 J2EE 中选择不同的技术，这就使得软件企业最终的应用依赖于这些技术，技术自身的复杂性和技术的风险性将会直接对应用造成冲击。而应用是软件企业的核心，是竞争力的关键所在，因此应该将应用自身的设计和具体的实现技术解耦。这样，软件企业的研发将集中在应用的设计上，而不是具体的技术实现，技术实现是应用的底层支撑，它不应该直接对应用产生影响。

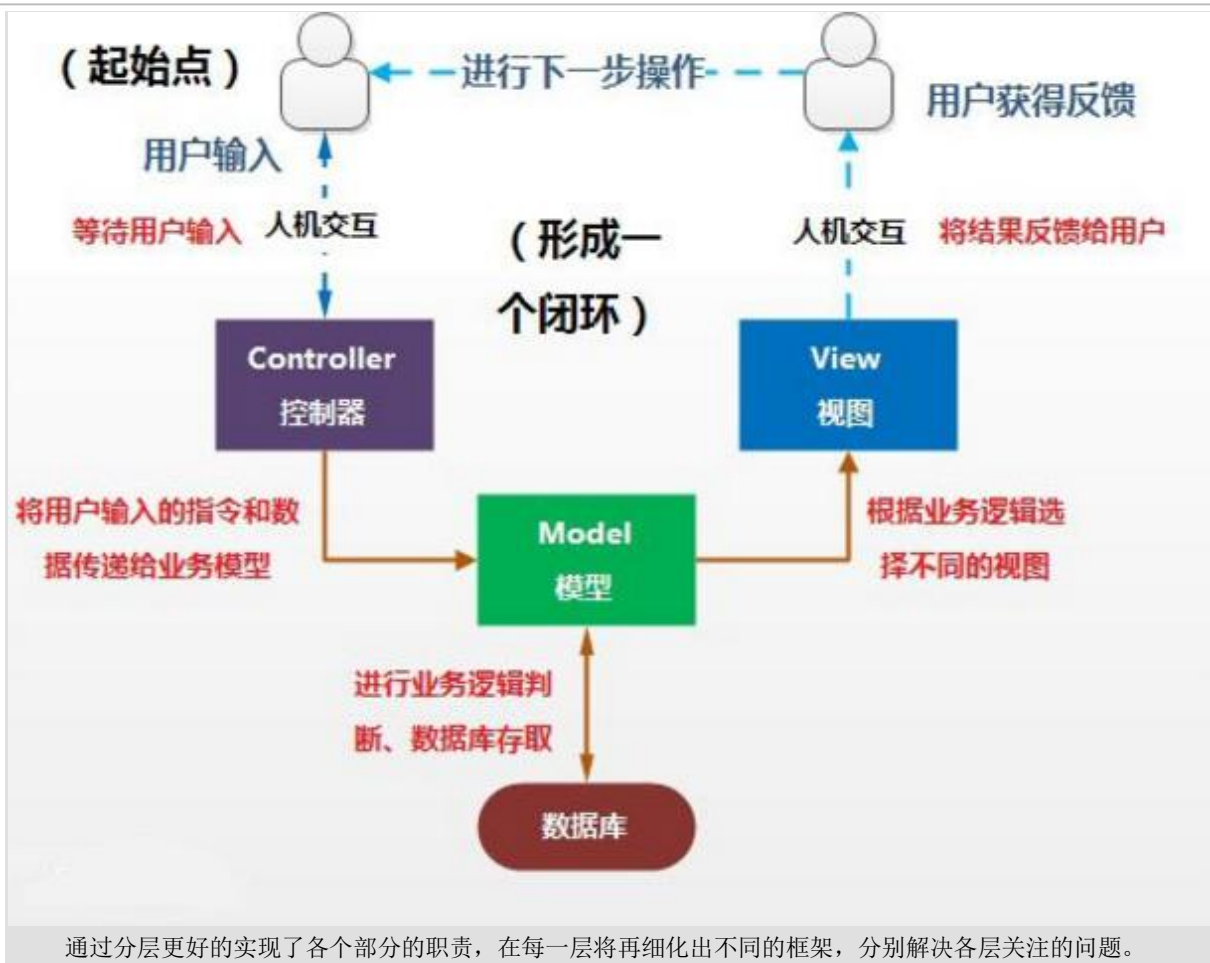
框架一般处在低层应用平台（如 J2EE）和高层业务逻辑之间的中间层。

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

表现层：用于展示数据
(SpringMVC)
业务层：处理业务需求
持久层：适合数据库交互
(Mybatis)





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

编辑

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

编辑

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 = ?";  
    }  
}
```



参数值

```
//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

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。

第2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。

2.1 Mybatis 框架开发的准备

2.1.1 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 - 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

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。

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。

2.2 搭建 Mybatis 开发环境

2.2.1 创建 maven 工程

创建 mybatis01 的工程，工程信息如下：

Groupid:com.itheima

ArtifactId:mybatis01

Packing:jar



2.2.2 添加 Mybatis3.4.5 的坐标

在 pom.xml 文件中添加 Mybatis3.4.5 的坐标，如下：

```
<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>
```

2.2.3 编写 User 实体类

```
/**
 *
 * <p>Title: User</p>
 * <p>Description: 用户的实体类</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
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 + "]";
}

}
```

2.2.4 编写持久层接口 IUserDao

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

```
/**
 *
```




```
* <p>Title: IUserDao</p>
* <p>Description: 用户的持久层操作</p>
* <p>Company: http://www.itheima.com/ </p>
*/

public interface IUserDao {

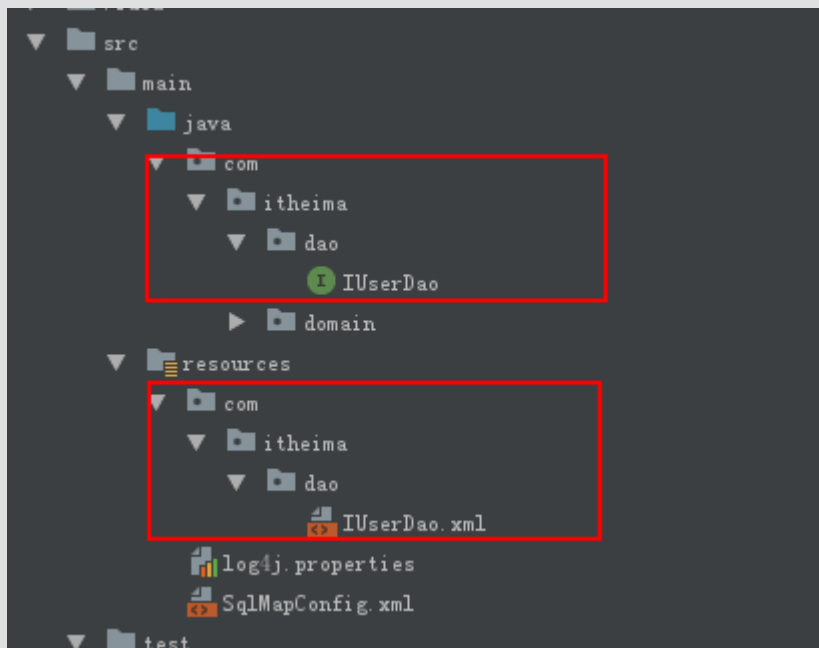
    /**
     * 查询所有用户
     * @return
     */
    List<User> findAll();
}
```

2.2.5 编写持久层接口的映射文件 IUserDao.xml

要求：

创建位置：必须和持久层接口在相同的包中。

名称：必须以持久层接口名称命名文件名，扩展名是.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.itheima.dao.IUserDao">
    <!-- 配置查询所有操作 -->
    <select id="findAll" resultType="com.itheima.domain.User">
        select * from user
    </select>
```



```
</mapper>
```

2.2.6 编写 SqlMapConfig.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">
<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://localhost:3306/ee50"/>
        <property name="username" value="root"/>
        <property name="password" value="1234"/>
      </dataSource>
    </environment>
  </environments>

  <!-- 告知 mybatis 映射配置的位置 -->
  <mappers>
    <mapper resource="com/itheima/dao/IUserDao.xml"/>
  </mappers>
</configuration>
```

2.2.7 编写测试类

```
/**
 *
 * <p>Title: MybatisTest</p>
 * <p>Description: 测试 mybatis 的环境</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
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
IUserDao userDao = session.getMapper(IUserDao.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();
}

}
```

2.3 小结

通过快速入门示例，我们发现使用 mybatis 是很容易的一件事情，因为只需要编写 Dao 接口并且按照 mybatis 要求编写两个配置文件，就可以实现功能。远比我们之前的 jdbc 方便多了。（我们使用注解之后，将变得更为简单，只需要编写一个 mybatis 配置文件就够了。）

但是，这里面包含了许多细节，比如为什么会有工厂对象（SqlSessionFactory），为什么有了工厂之后还要有构建者对象（SqlSessionFactoryBuilder），为什么 IUserDao.xml 在创建时有位置和文件名的要求等等。这些问题我们在自定义 mybatis 框架的章节，通过层层剥离的方式，给大家讲解。

请注意：我们讲解自定义 Mybatis 框架，不是让大家回去自己去写个 mybatis，而是让我们能更好了解 mybatis 内部是怎么执行的，在以后的开发中能更好的使用 mybatis 框架，同时对它的设计理念（设计模式）有一个认识。

2.4 补充（基于注解的 mybatis 使用）

2.4.1 在持久层接口中添加注解

```
/**
 *
 * <p>Title: IUserDao</p>
 * <p>Description: 用户的持久层操作</p>
```



```
* <p>Company: http://www.itheima.com/ </p>
*/
public interface IUserDao {

    /**
     * 查询所有用户
     * @return
     */
    @Select("select * from user")
    List<User> findAll();
}
```

2.4.2 修改 SqlMapConfig.xml

```
<!-- 告知 mybatis 映射配置的位置 -->
<mappers>
    <mapper class="com.itheima.dao.IUserDao"/>
</mappers>
```

2.4.3 注意事项：

在使用基于注解的 Mybatis 配置时，请移除 xml 的映射配置（IUserDao.xml）。
补充

第3章 自定义 Mybatis 框架

3.1 自定义 Mybatis 框架的分析

3.1.1 涉及知识点介绍

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。



3.1.2 分析流程



3.2 前期准备

3.2.1 创建 Maven 工程

创建 mybatis02 的工程，工程信息如下：

```

Groupid:com.itheima
ArtifactId:mybatis02
Packing:jar

```

3.2.2 引入相关坐标

```

<dependencies>
  <!-- 日志坐标 -->
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.12</version>
  </dependency>
  <!-- 解析 xml 的 dom4j -->
  <dependency>
    <groupId>dom4j</groupId>
    <artifactId>dom4j</artifactId>
    <version>1.6.1</version>
  </dependency>
</dependencies>

```



```
<!-- mysql 驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
</dependency>
<!-- dom4j 的依赖包 jaxen -->
<dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>1.1.6</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.10</version>
</dependency>
</dependencies>
```

3.2.3 引入工具类到项目中

```
/**
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * 用于解析配置文件
 */
public class XMLConfigBuilder {

    /**
     * 解析主配置文件，把里面的内容填充到 DefaultSqlSession 所需要的地方
     * 使用的技术：
     *     dom4j+xpath
     * @param session
     */
    public static void loadConfiguration(DefaultSqlSession session,InputStream
config){
        try{
            //定义封装连接信息的配置对象（mybatis 的配置对象）
            Configuration cfg = new Configuration();

            //1.获取 SAXReader 对象
            SAXReader reader = new SAXReader();
            //2.根据字节输入流获取 Document 对象
            Document document = reader.read(config);
```



```
//3.获取根节点
Element root = document.getRootElement();
//4.使用 xpath 中选择指定节点的方式，获取所有 property 节点
List<Element> propertyElements = root.selectNodes("//property");
//5.遍历节点
for (Element propertyElement : propertyElements) {
    //判断节点是连接数据库的哪部分信息
    //取出 name 属性的值
    String name = propertyElement.attributeValue("name");
    if ("driver".equals(name)) {
        //表示驱动
        //获取 property 标签 value 属性的值
        String driver = propertyElement.attributeValue("value");
        cfg.setDriver(driver);
    }
    if ("url".equals(name)) {
        //表示连接字符串
        //获取 property 标签 value 属性的值
        String url = propertyElement.attributeValue("value");
        cfg.setUrl(url);
    }
    if ("username".equals(name)) {
        //表示用户名
        //获取 property 标签 value 属性的值
        String username = propertyElement.attributeValue("value");
        cfg.setUsername(username);
    }
    if ("password".equals(name)) {
        //表示密码
        //获取 property 标签 value 属性的值
        String password = propertyElement.attributeValue("value");
        cfg.setPassword(password);
    }
}
//取出 mappers 中的所有 mapper 标签，判断他们使用了 resource 还是 class 属性
List<Element> mapperElements = root.selectNodes("//mappers/mapper");
//遍历集合
for (Element mapperElement : mapperElements) {
    //判断 mapperElement 使用的是哪个属性
    Attribute attribute = mapperElement.attribute("resource");
    if (attribute != null) {
        System.out.println("使用的是 XML");
        //表示有 resource 属性，用的是 XML
        //取出属性的值
    }
}
```




```
String mapperPath = attribute.getValue();// 获取属性的值
"com/itheima/dao/IUserDao.xml"
//把映射配置文件的内容获取出来，封装成一个 map
Map<String,Mapper> mappers = loadMapperConfiguration(mapperPath);
//给 configuration 中的 mappers 赋值
cfg.setMappers(mappers);
} else {
    System.out.println("使用的是注解");
    //表示没有 resource 属性，用的是注解
    //获取 class 属性的值
    String daoClassPath = mapperElement.attributeValue("class");
    //根据 daoClassPath 获取封装的必要信息
    Map<String,Mapper> mappers = loadMapperAnnotation(daoClassPath);
    //给 configuration 中的 mappers 赋值
    cfg.setMappers(mappers);
}
}
//把配置对象传递给 DefaultSqlSession
session.setCfg(cfg);

} catch (Exception e) {
    throw new RuntimeException(e);
} finally {
    try {
        config.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

/**
 * 根据传入的参数，解析 XML，并且封装到 Map 中
 * @param mapperPath 映射配置文件的位置
 * @return map 中包含了获取的唯一标识（key 是由 dao 的全限定类名和方法名组成）
 * 以及执行所需的必要信息（value 是一个 Mapper 对象，里面存放的是执行的 SQL 语句和
要封装的实体类全限定类名）
 */
private static Map<String,Mapper> loadMapperConfiguration(String
mapperPath) throws IOException {
    InputStream in = null;
    try {
        //定义返回值对象
        Map<String,Mapper> mappers = new HashMap<String,Mapper>();
```



部分

```
//1.根据路径获取字节输入流
in = Resources.getResourceAsStream(mapperPath);
//2.根据字节输入流获取 Document 对象
SAXReader reader = new SAXReader();
Document document = reader.read(in);
//3.获取根节点
Element root = document.getRootElement();
//4.获取根节点的 namespace 属性取值
String namespace = root.attributeValue("namespace");//是组成 map 中 key 的

//5.获取所有的 select 节点
List<Element> selectElements = root.selectNodes("//select");
//6.遍历 select 节点集合
for (Element selectElement : selectElements) {
    //取出 id 属性的值      组成 map 中 key 的部分
    String id = selectElement.attributeValue("id");
    //取出 resultType 属性的值 组成 map 中 value 的部分
    String resultType = selectElement.attributeValue("resultType");
    //取出文本内容      组成 map 中 value 的部分
    String queryString = selectElement.getText();
    //创建 key
    String key = namespace+"."+id;
    //创建 Value
    Mapper mapper = new Mapper();
    mapper.setQueryString(queryString);
    mapper.setResultType(resultType);
    //把 key 和 value 存入 mappers 中
    mappers.put(key, mapper);
}

return mappers;
} catch (Exception e) {
    throw new RuntimeException(e);
} finally {
    in.close();
}
}

/**
 * 根据传入的参数，得到 dao 中所有被 select 注解标注的方法。
 * 根据方法名称和类名，以及方法上注解 value 属性的值，组成 Mapper 的必要信息
 * @param daoClassPath
 * @return
 */
private static Map<String, Mapper> loadMapperAnnotation(String
```



```
daoClassPath) throws Exception {  
    //定义返回值对象  
    Map<String, Mapper> mappers = new HashMap<String, Mapper>();  
  
    //1.得到 dao 接口的字节码对象  
    Class daoClass = Class.forName(daoClassPath);  
    //2.得到 dao 接口中的方法数组  
    Method[] methods = daoClass.getMethods();  
    //3.遍历 Method 数组  
    for (Method method : methods) {  
        //取出每一个方法，判断是否有 select 注解  
        boolean isAnnotated = method.isAnnotationPresent(Select.class);  
        if (isAnnotated) {  
            //创建 Mapper 对象  
            Mapper mapper = new Mapper();  
            //取出注解的 value 属性值  
            Select selectAnno = method.getAnnotation(Select.class);  
            String queryString = selectAnno.value();  
            mapper.setQueryString(queryString);  
            //获取当前方法的返回值，还要求必须带有泛型信息  
            Type type = method.getGenericReturnType(); //List<User>  
            //判断 type 是不是参数化的类型  
            if (type instanceof ParameterizedType) {  
                //强转  
                ParameterizedType ptype = (ParameterizedType) type;  
                //得到参数化类型中的实际类型参数  
                Type[] types = ptype.getActualTypeArguments();  
                //取出第一个  
                Class domainClass = (Class) types[0];  
                //获取 domainClass 的类名  
                String resultType = domainClass.getName();  
                //给 Mapper 赋值  
                mapper.setResultType(resultType);  
            }  
            //组装 key 的信息  
            //获取方法的名称  
            String methodName = method.getName();  
            String className = method.getDeclaringClass().getName();  
            String key = className + "." + methodName;  
            //给 map 赋值  
            mappers.put(key, mapper);  
        }  
    }  
    return mappers;  
}
```



```
    }  
}  
  
/**  
 * @author 黑马程序员  
 * @Company http://www.ithiema.com  
 * 负责执行 SQL 语句，并且封装结果集  
 */  
public class Executor {  
  
    public <E> List<E> selectList(Mapper mapper, Connection conn) {  
        PreparedStatement pstmt = null;  
        ResultSet rs = null;  
        try {  
            //1.取出 mapper 中的数据  
            String queryString = mapper.getQueryString();//select * from user  
            String resultType = mapper.getResultType();//com.ithiema.domain.User  
            Class domainClass = Class.forName(resultType);//User.class  
            //2.获取 PreparedStatement 对象  
            pstmt = conn.prepareStatement(queryString);  
            //3.执行 SQL 语句，获取结果集  
            rs = pstmt.executeQuery();  
            //4.封装结果集  
            List<E> list = new ArrayList<E>();//定义返回值  
            while(rs.next()) {  
                //实例化要封装的实体类对象  
                E obj = (E)domainClass.newInstance();//User 对象  
  
                //取出结果集的元信息：ResultSetMetaData  
                ResultSetMetaData rsmd = rs.getMetaData();  
                //取出总列数  
                int columnCount = rsmd.getColumnCount();  
                //遍历总列数  
                for (int i = 1; i <= columnCount; i++) {  
                    //获取每列的名称，列名的序号是从 1 开始的  
                    String columnName = rsmd.getColumnName(i);  
                    //根据得到列名，获取每列的值  
                    Object columnValue = rs.getObject(columnName);  
                    //给 obj 赋值：使用 Java 内省机制（借助 PropertyDescriptor 实现属性的封装）  
                    PropertyDescriptor pd = new  
PropertyDescriptor(columnName, domainClass);//要求：实体类的属性和数据库表的列名保持一种  
                    //获取它的写入方法
```



```
        Method writeMethod = pd.getWriteMethod();//setUsername(String
username);

        //把获取的列的值，给对象赋值
        writeMethod.invoke(obj, columnValue);
    }
    //把赋好值的对象加入到集合中
    list.add(obj);
}
return list;
} catch (Exception e) {
    throw new RuntimeException(e);
} finally {
    release(pstm,rs);
}
}

private void release(PreparedStatement pstm,ResultSet rs){
    if(rs != null){
        try {
            rs.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    if(pstm != null){
        try {
            pstm.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

/**
 *
 * <p>Title: DataSourceUtil</p>
 * <p>Description: 数据源的工具类</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
```



```
public class DataSourceUtil {

    /**
     * 获取连接
     * @param cfg
     * @return
     */
    public static Connection getConnection(Configuration cfg) {
        try {
            Class.forName(cfg.getDriver());
            Connection conn =
            DriverManager.getConnection(cfg.getUrl(),cfg.getUsername() , cfg.getPassword());
            return conn;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

3.2.4 编写 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:///eesy" ></property>
                <property name="username" value="root"></property>
                <property name="password" value="1234"></property>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

注意：

此处我们直接使用的是 mybatis 的配置文件，但是由于我们没有使用 mybatis 的 jar 包，所以要把配置文件的约束删掉否则会报错（如果电脑能接入互联网，不删也行）



3.2.5 编写读取配置文件类

```
/**
 *
 * <p>Title: Resources</p>
 * <p>Description: 用于读取配置文件的类</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
public class Resources {

    /**
     * 用于加载 xml 文件，并且得到一个流对象
     * @param xmlPath
     * @return
     * 在实际开发中读取配置文件：
     *     第一：使用类加载器。但是有要求：a 文件不能过大。 b 文件必须在类路径下 (classpath)
     *     第二：使用 ServletContext 的 getRealPath()
     */
    public static InputStream getResourceAsStream(String xmlPath) {
        return Resources.class.getClassLoader().getResourceAsStream(xmlPath);
    }
}
```

3.2.6 编写 Mapper 类

```
/**
 *
 * <p>Title: Mapper</p>
 * <p>Description: 用于封装查询时的必要信息：要执行的 SQL 语句和实体类的全限定类名</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
public class Mapper {

    private String queryString;//sql
    private String resultType;//结果类型的全限定类名
    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;
    }
}
```

3.2.7 编写 Configuration 配置类

```
/**
 * 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;
    }
}
```

3.2.8 编写 User 实体类

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 + "];"
    }
}
```

3.3 基于 XML 的自定义 mybatis 框架

3.3.1 编写持久层接口和 IUserDao.xml

```
/**
 *
 * <p>Title: IUserDao</p>
 * <p>Description: 用户的持久层操作</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
```



```
public interface IUserDao {

    /**
     * 查询所有用户
     * @return
     */
    List<User> findAll();
}

<?xml version="1.0" encoding="UTF-8"?>
<mapper namespace="com.itheima.dao.IUserDao">
    <!-- 配置查询所有操作 -->
    <select id="findAll" resultType="com.itheima.domain.User">
        select * from user
    </select>
</mapper>
```

注意：

此处我们使用的也是 mybatis 的配置文件，所以也要把约束删除了

3.3.2 编写构建者类

```
/**
 *
 * <p>Title: SqlSessionFactoryBuilder</p>
 * <p>Description: 用于构建 SqlSessionFactory 的</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
public class SqlSessionFactoryBuilder {

    /**
     * 根据传入的流，实现对 SqlSessionFactory 的创建
     * @param in 它就是 SqlMapConfig.xml 的配置以及里面包含的 IUserDao.xml 的配置
     * @return
     */
    public SqlSessionFactory build(InputStream in) {
        DefaultSqlSessionFactory factory = new DefaultSqlSessionFactory();
        //给 factory 中 config 赋值
        factory.setConfig(in);
        return factory;
    }
}
```



3.3.3 编写 SqlSessionFactory 接口和实现类

```
/**
 *
 * <p>Title: SqlSessionFactory</p>
 * <p>Description: SqlSessionFactory 的接口</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
public interface SqlSessionFactory {

    /**
     * 创建一个新的 SqlSession 对象
     * @return
     */
    SqlSession openSession();

}

/**
 *
 * <p>Title: DefaultSqlSessionFactory</p>
 * <p>Description: SqlSessionFactory 的默认实现 </p>
 * <p>Company: http://www.itheima.com/ </p>
 */
public class DefaultSqlSessionFactory implements SqlSessionFactory {

    private InputStream config = null;

    public void setConfig(InputStream config) {
        this.config = config;
    }

    @Override
    public SqlSession openSession() {
        DefaultSqlSession session = new DefaultSqlSession();
        //调用工具类解析 xml 文件
        XMLConfigBuilder.loadConfiguration(session, config);
        return session;
    }

}
```



3.3.4 编写 SqlSession 接口和实现类

```
/**
 *
 * <p>Title: SqlSession</p>
 * <p>Description: 操作数据库的核心对象</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
public interface SqlSession {

    /**
     * 创建 Dao 接口的代理对象
     * @param daoClass
     * @return
     */
    <T> T getMapper(Class<T> daoClass);

    /**
     * 释放资源
     */
    void close();

}

/**
 *
 * <p>Title: DefaultSqlSession</p>
 * <p>Description: SqlSession 的具体实现</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
public class DefaultSqlSession implements SqlSession {
    //核心配置对象
    private Configuration cfg;

    public void setCfg(Configuration cfg) {
        this.cfg = cfg;
    }

    //连接对象
    private Connection conn;

    //调用 DataSourceUtils 工具类获取连接
    public Connection getConn() {
```



```
        try {
            conn = DataSourceUtil.getDataSource(cfg).getConnection();
            return conn;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * 动态代理：
     * 涉及的类：Proxy
     * 使用的方法：newProxyInstance
     * 方法的参数：
     *      ClassLoader：和被代理对象使用相同的类加载器，通常都是固定的
     *      Class[]：代理对象和被代理对象要求有相同的行为。（具有相同的方法）
     *      InvocationHandler：如何代理。需要我们自己提供的增强部分的代码
     */
    @Override
    public <T> T getMapper(Class<T> daoClass) {
        conn = getConn();
        System.out.println(conn);
        T daoProxy = (T) Proxy.newProxyInstance(daoClass.getClassLoader(), new
Class[] {daoClass}, new MapperProxyFactory(cfg.getMappers(), conn));
        return daoProxy;
    }

    //释放资源
    @Override
    public void close() {
        try {
            System.out.println(conn);
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //查询所有方法
    public <E> List<E> selectList(String statement) {
        Mapper mapper = cfg.getMappers().get(statement);
        return new Executor().selectList(mapper, conn);
    }
}
```



3.3.5 编写用于创建 Dao 接口代理对象的类

```
/**
 *
 * <p>Title: MapperProxyFactory</p>
 * <p>Description: 用于创建代理对象是增强方法</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
public class MapperProxyFactory implements InvocationHandler {

    private Map<String, Mapper> mappers;
    private Connection conn;

    public MapperProxyFactory(Map<String, Mapper> mappers, Connection conn) {
        this.mappers = mappers;
        this.conn = conn;
    }

    /**
     * 对当前正在执行的方法进行增强
     * 取出当前执行的方法名称
     * 取出当前执行的方法所在类
     * 拼接成 key
     * 去 Map 中获取 Value (Mapper)
     * 使用工具类 Executor 的 selectList 方法
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
    {
        //1.取出方法名
        String methodName = method.getName();
        //2.取出方法所在类名
        String className = method.getDeclaringClass().getName();
        //3.拼接成 Key
        String key = className+"."+methodName;
        //4.使用 key 取出 mapper
        Mapper mapper = mappers.get(key);
        if(mapper == null) {
            throw new IllegalArgumentException("传入的参数有误，无法获取执行的必要条件");
        }
        //5.创建 Executor 对象
        Executor executor = new Executor();
    }
}
```




```
        return executor.selectList(mapper, conn);  
    }  
}
```

3.3.6 运行测试类

```
/**  
 *  
 * <p>Title: MybatisTest</p>  
 * <p>Description: 测试 mybatis 的环境</p>  
 * <p>Company: http://www.itheima.com/ </p>  
 */  
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  
        IUserDao userDao = session.getMapper(IUserDao.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();  
    }  
}
```

3.4 基于注解方式定义 Mybatis 框架

3.4.1 自定义@Select 注解

```
/**
```



```
*  
* <p>Title: Select</p>  
* <p>Description: 自定义查询注解</p>  
* <p>Company: http://www.itheima.com/ </p>  
*/  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Select {  
  
    String value();  
}
```

3.4.2 修改持久层接口

```
/**  
*  
* <p>Title: IUserDao</p>  
* <p>Description: 用户的持久层操作</p>  
* <p>Company: http://www.itheima.com/ </p>  
*/  
  
public interface IUserDao {  
  
    /**  
    * 查询所有用户  
    * @return  
    */  
    @Select("select * from user")  
    List<User> findAll();  
}
```

3.4.3 修改 SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<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://localhost:3306/ee50"/>
<property name="username" value="root"/>
<property name="password" value="1234"/>
</dataSource>
</environment>
</environments>

<!-- 告知 mybatis 映射配置的位置 -->
<mapppers>
    <mapper class="com.itheima.dao.IUserDao"/>
</mapppers>
</configuration>
```

3.5 自定义 Mybatis 的设计模式说明

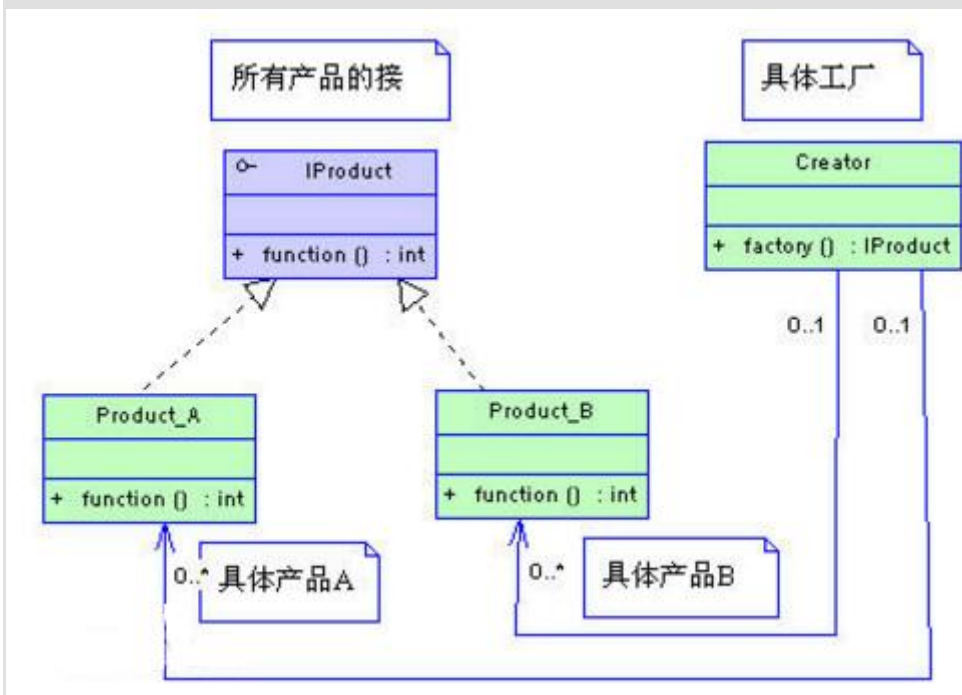
3.5.1 工厂模式 (SqlSessionFactory)

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：



3.5.2 代理模式(MapperProxyFactory)

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。

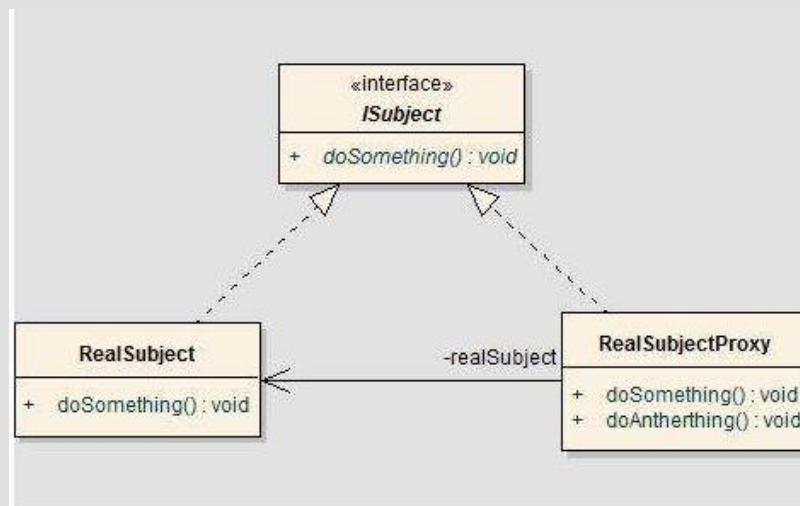
代理角色：实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。

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，今天我们会使用 JDK 动态代理方式来编写 MapperProxyFactory 类。

动态代理模型图：



3.5.3 构建者模式(SqlSessionFactoryBuilder)

首先我们一起来学习构建者模式，通过百度百科如下：

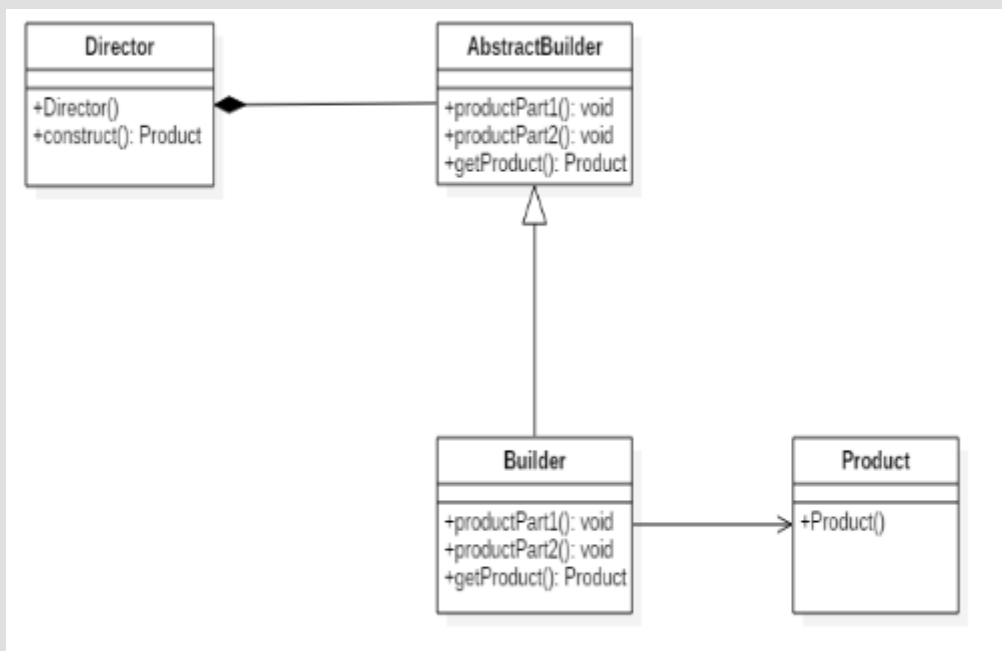
创建者模式

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。

3.5.4 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。