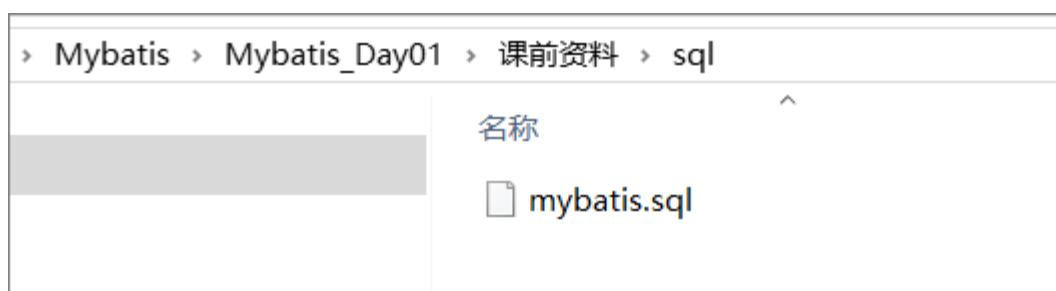


Mybatis

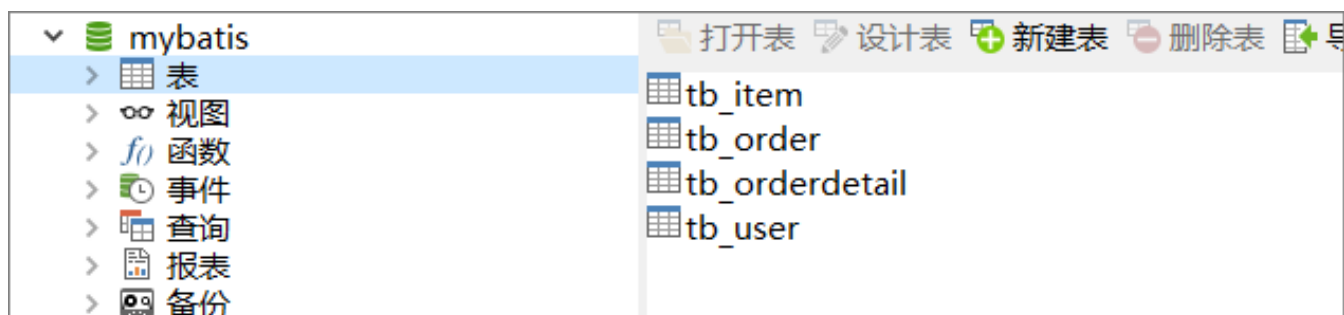
1.JDBC 回顾

1.1. 创建 mybatis 库，执行 sql

SQL 文件:



执行后:



tb_user 表:

```
CREATE TABLE `tb_user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(100) DEFAULT NULL COMMENT '用户名',
  `password` varchar(100) DEFAULT NULL COMMENT '密码',
  `name` varchar(100) DEFAULT NULL COMMENT '姓名',
  `age` int(10) DEFAULT NULL COMMENT '年龄',
  `sex` tinyint(1) DEFAULT NULL COMMENT '性别, 1男性, 2女性',
  `birthday` date DEFAULT NULL COMMENT '出生日期',
  `created` datetime DEFAULT NULL COMMENT '创建时间',
  `updated` datetime DEFAULT NULL COMMENT '更新时间',
  PRIMARY KEY (`id`),
  UNIQUE KEY `username` (`user_name`)
) ENGINE=InnoDB AUTO_INCREMENT=22 DEFAULT CHARSET=utf8;
```

id	user_name	password	name	age	sex	birthday	created	upc
1	zhangsan	123456	张三	30	1	1984-08-08	2014-09-19 16:56:04	201
2	lisi	123456	李四	21	2	1991-01-01	2014-09-19 16:56:04	201
3	wangwu	123456	王五	22	2	1989-01-01	2014-09-19 16:56:04	201
4	zhangwei	123456	张伟	20	1	1988-09-01	2014-09-19 16:56:04	201
5	lina	123456	李娜	28	1	1985-01-01	2014-09-19 16:56:04	201
6	lilei	123456	李磊	23	1	1988-08-08	2014-09-20 11:41:15	201

1.2. 导入 Maven 父工程

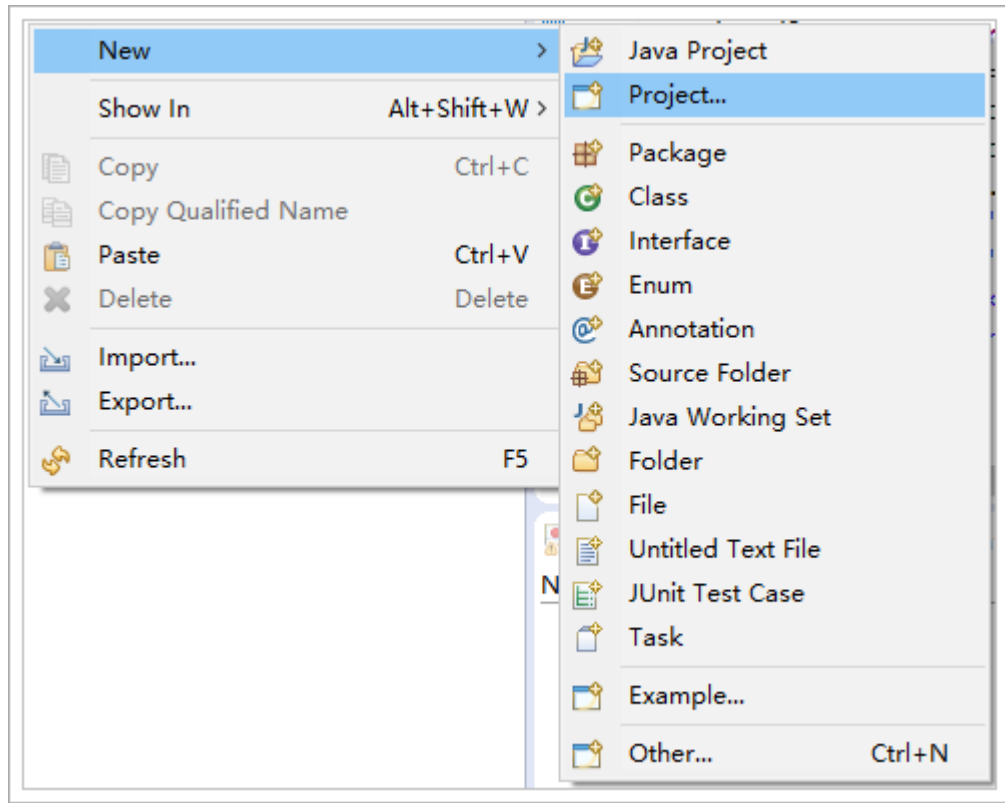
将课前资料中的 itcast-parent 工程拷贝到工作区并导入:

Desktop > Mybatis > Mybatis_Day01 > 课前资料 >		
名称	修改日期	类
dtd	2018/4/2 9:26	文
itcast-parent	2018/4/7 11:36	文
Mybatis相关资料	2018/4/2 9:26	文
pojo	2018/4/2 9:26	文
sql	2018/4/2 9:26	文
xsd	2018/4/2 9:26	文
pom.xml	2018/4/2 10:25	XI

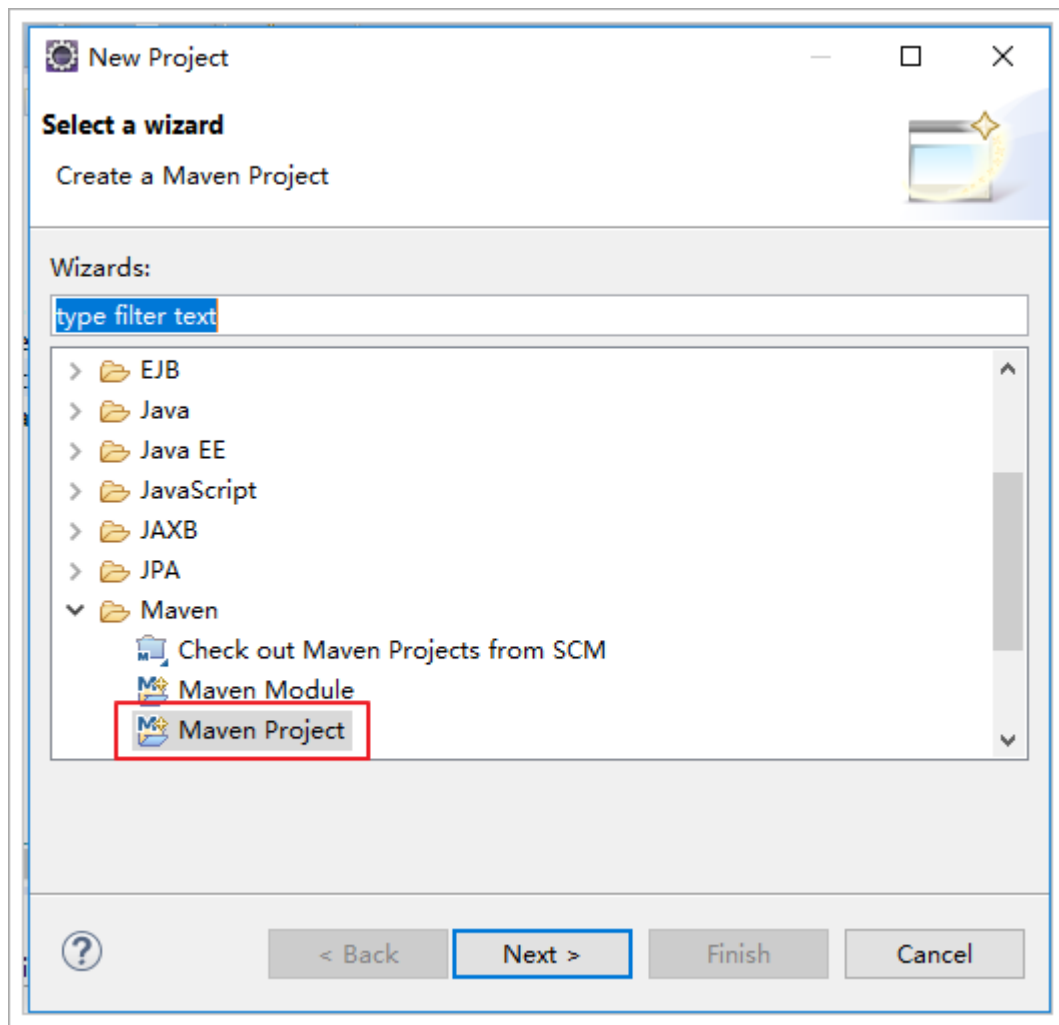
父工程的作用是为了项目的聚合和继承(mave 增强课中详解), 这里的作用是为了做工程的 jar 包版本的统一管理

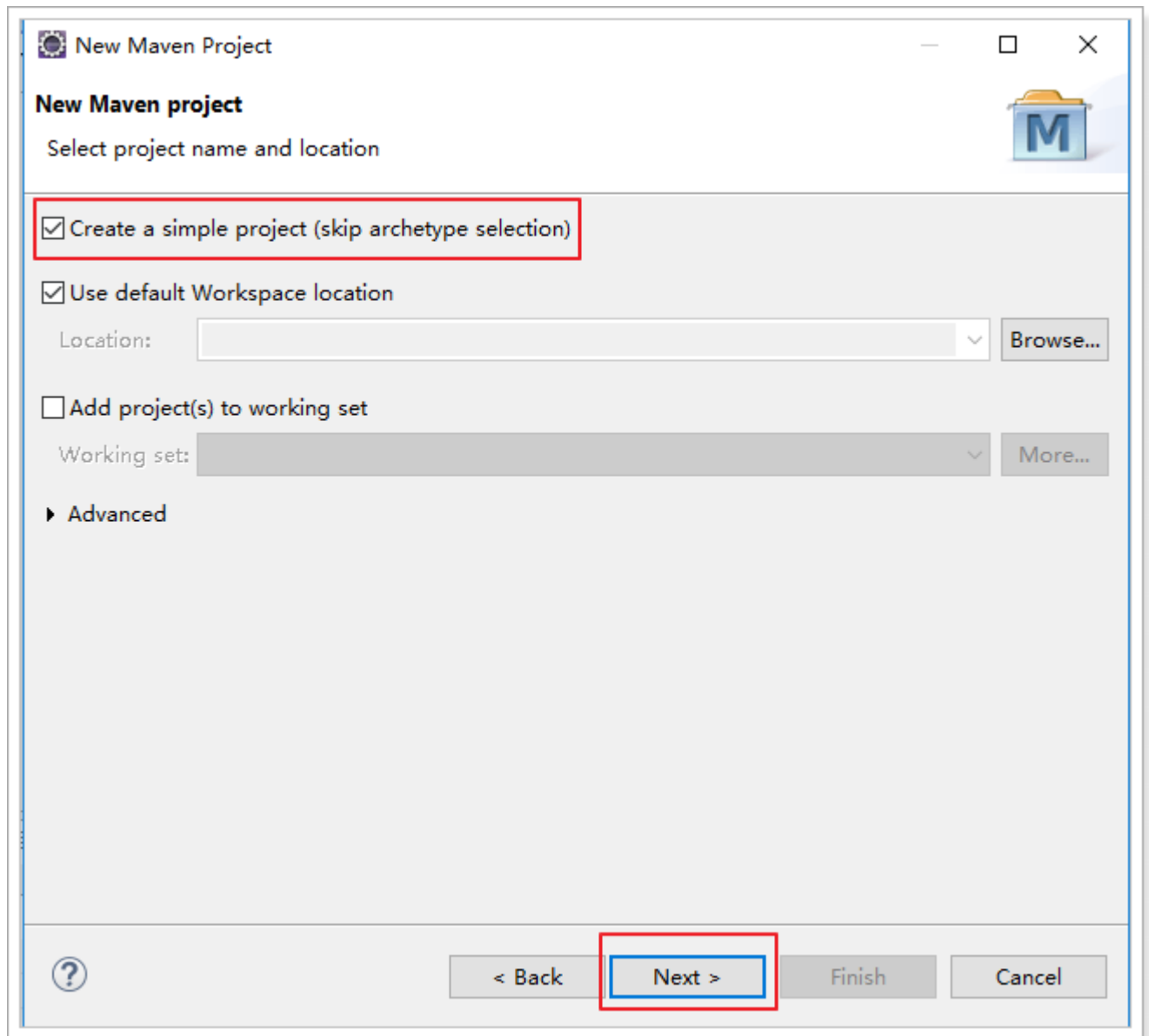
1.3. 创建 maven 工程

右键—>new-->Project



选择: Maven—>Maven Project





坐标：

New Maven Project


New Maven project


Configure project

Artifact

Group Id:

Artifact Id:

Version: 

Packaging: 


Name:

Description:

Parent Project

Group Id:

Artifact Id:

Version: 

► Advanced

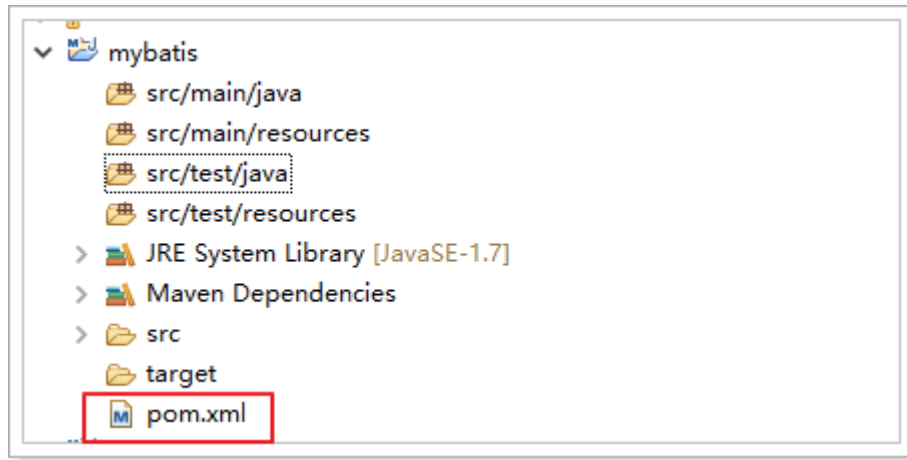


< Back

Next >

Finish

1.4.引入 mysql 依赖包



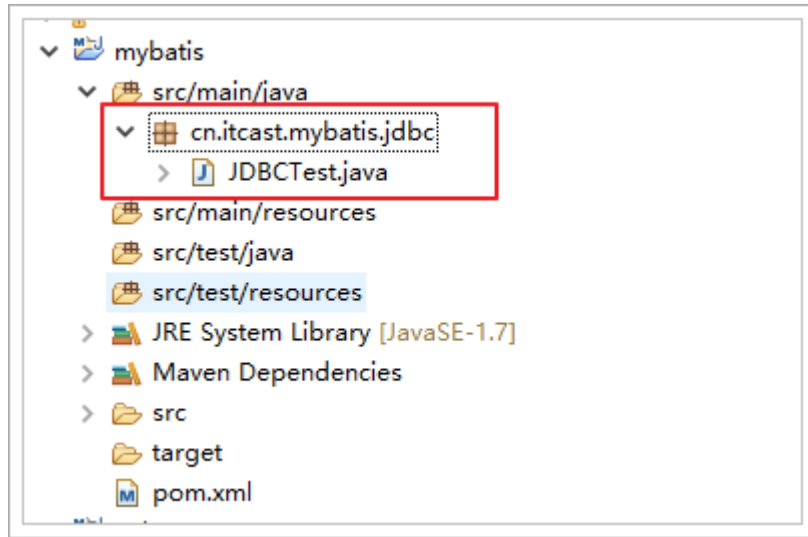
pom.xml 内容:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <!--配置父工程,获取父工程中管理的版本信息-->
  <parent>
    <groupId>cn.itcast.parent</groupId>
    <artifactId>itcast-parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <groupId>cn.itcast.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <dependencies>
    <!-- MySql -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
    </dependency>
  </dependencies>
</project>
```

1.5. 需求

根据 id 查询用户信息

1.6. JDBCTest



```
package cn.itcast.mybatis.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class JDBCTest {

    public static void main(String[] args) throws
Exception {
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;

        // 加载驱动
        Class.forName("com.mysql.jdbc.Driver");

        // 获取连接
        String url =
"jdbc:mysql://127.0.0.1:3306/mybatis";
```



```
String user = "root";
String password = "root";
conn = DriverManager.getConnection(url, user,
password);

// 获取statement, preparedStatement

String sql = "select * from tb_user where id=?";
ps = conn.prepareStatement(sql);

// 设置参数
ps.setLong(1, 11);

// 执行查询, 获取结果集
rs = ps.executeQuery();

// 处理结果集
while (rs.next()) {
    System.out.println(rs.getString("user_name"));
    System.out.println(rs.getString("name"));
    System.out.println(rs.getInt("age"));
}

// 关闭连接, 释放资源
if (rs != null) {
    rs.close();
}
if (ps != null) {
    ps.close();
}
if (conn != null) {
    conn.close();
}

}
}
```

1.7. 缺点及解决方案

```
ResultSet resultSet = null;
```

```
// 1.加载驱动
```

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

```
// 2.获取连接
```

```
String url = "jdbc:mysql://127.0.0.1:3306/mybatis-0323";
```

```
String user = "root";
```

```
String password = "root";
```

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

```
// 3.获取statement, preparedStatement
```

```
String sql = "select * from tb_user where id=?";
```

```
preparedStatement = connection.prepareStatement(sql);
```

```
// 4.设置参数
```

```
preparedStatement.setLong(1, 11);
```

```
// 5.执行查询, 获取结果集
```

```
resultSet = preparedStatement.executeQuery();
```

```
// 6.处理结果集
```

```
while(resultSet.next()) {
```

```
    System.out.println(resultSet.getString("user_name"));
```

```
    System.out.println(resultSet.getString("password"));
```

```
    System.out.println(resultSet.getInt("age"));
```

```
}
```

```
// 7.关闭连接, 释放资源
```

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

```
    resultSet.close();
```

```
}
```

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

```
    preparedStatement.close();
```

```
}
```

1.驱动名称硬编码
2.每次都要加载驱动
连接池 + 配置文件

sql语言
mybat

1.参数类型需要手动判断
2.下标需要人为判断
3.参数值需要人为设置
框架

1.参
2.下
3.结
框架

1.每次都要关闭连接, 释放资源, 麻烦
连接池

2.框架简介

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。

可以说, 一个框架是一个可复用的设计构件, 它规定了应用的体系结构, 阐明了整个设

计、协作构件之间的依赖关系、责任分配和控制流程，表现为一组抽象类以及其实例之间协作的方法，它为构件复用提供了上下文(Context)关系。因此构件库的大规模重用也需要框架。

构件领域框架方法在很大程度上借鉴了硬件技术发展的成就，它是构件技术、软件体系结构研究和应用软件开发三者发展结合的产物。在很多情况下，框架通常以构件库的形式出现，但构件库只是框架的一个重要部分。框架的关键还在于框架内对象间的交互模式和控制流模式。

框架比构件可定制性强。在某种程度上，将构件和框架看成两个不同但彼此协作的技术或许更好。框架为构件提供重用的环境，为构件处理错误、交换数据及激活操作提供了标准的方法。

应用框架的概念也很简单。它并不是包含构件应用程序的小片程序，而是实现了某应用领域通用完备功能（除去特殊应用的部分）的底层服务。使用这种框架的编程人员可以在一个通用功能已经实现的基础上开始具体的系统开发。框架提供了所有应用期望的默认行为的类集合。具体的应用通过重写子类(该子类属于框架的默认行为)或组装对象来支持应用专用的行为。

应用框架强调的是软件的设计重用性和系统的可扩充性,以缩短大型应用软件系统的开发周期，提高开发质量。与传统的基于类库的面向对象重用技术比较，应用框架更侧重于面向专业领域的软件重用。应用框架具有领域相关性，构件根据框架进行复合而生成可运行的系统。框架的粒度越大，其中包含的领域知识就更加完整。

框架，即 framework。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。

3. EMyBatis 介绍

3.1. 简介

Mybatis的前身是iBatis, Apache的一个开源项目, 2010年这个项目从Apache迁移到Google Code改名为Mybatis 之后将版本升级到3.X, 其官网:
<http://blog.mybatis.org/>, 从3.2版本之后迁移到github, 目前最新稳定版本为: 3.2.8。

Mybatis是一个类似于Hibernate的ORM持久化框架, 支持普通SQL查询, 存储过程以及高级映射。Mybatis通过使用简单的XML或注解用于配置和原始映射, 将接口和POJO对象映射成数据库中的记录。

由于Mybatis是直接基于JDBC做了简单的映射包装, 所有从性能角度来看:

JDBC > Mybatis > Hibernate

3.2. 官方文档

MyBatis



最近更新: 10 十月 2014 | 版本: 3.2.8

参考文档

简介

入门

XML 配置

XML 映射文件

动态 SQL

简介

什么是 MyBatis ?

MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀持久层框架。它将几乎所有的 JDBC 代码和手工设置参数以及抽取结果集。MyBatis 使用简单的 XML 或注解来配置和映射数据库的复杂对象 (Plain Old Java Objects, 普通的 Java Objects)。

3.3. 特点

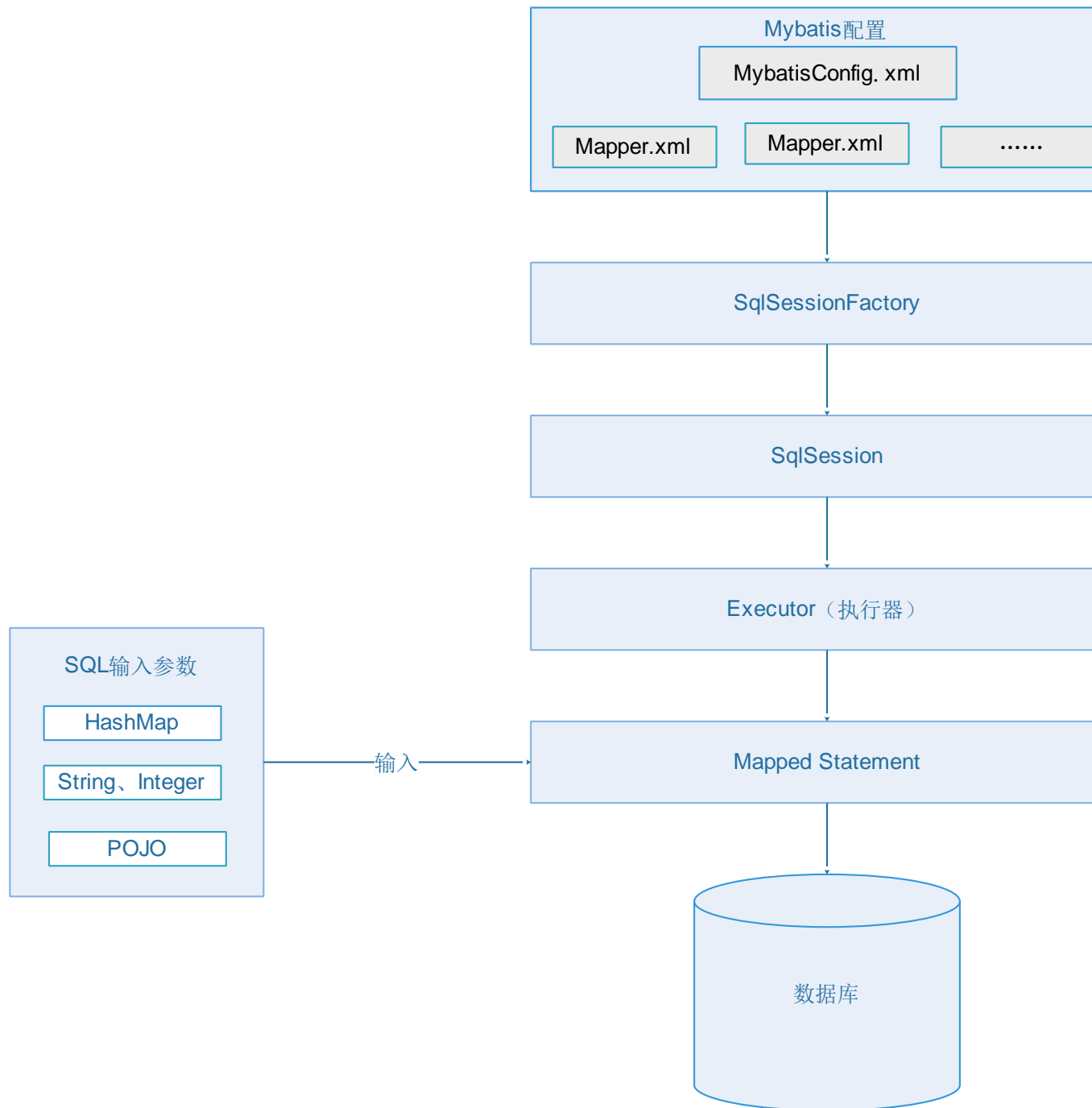
Mybatis:

- 1) 支持自定义 SQL、存储过程、及高级映射
- 2) 实现自动对 SQL 的参数设置
- 3) 实现自动对结果集进行解析和封装
- 4) 通过 XML 或者注解进行配置和映射，大大减少代码量
- 5) 数据源的连接信息通过配置文件进行配置

可以发现，MyBatis 是对 JDBC 进行了简单的封装，帮助用户进行 SQL 参数的自动设置，以及结果集与 Java 对象的自动映射。与 Hibernate 相比，配置更加简单、灵活、执行效率高。但是正因为此，所以没有实现完全自动化，需要手写 SQL，这是优点也是缺点。

因此，对性能要求较高的电商类项目，一般会使用 MyBatis，而对与业务逻辑复杂，不太在乎执行效率的传统行业，一般会使用 Hibernate

3.4. Mybaitis 整体架构



1、配置文件

全局配置文件：mybatis-config.xml 作用：配置数据源，引入映射文件

映射文件：XxMapper.xml 作用：配置 sql 语句、参数、结果集封装类型等

2、SqlSessionFactory 作用：获取 SqlSession

通过 `newSqlSessionFactoryBuilder().build(inputStream)` 来构建, `inputStream`: 读取配置文件的 IO 流

3、SqlSession 作用：执行 CRUD 操作

4、Executor

执行器, `SqlSession` 通过调用它来完成具体的 CRUD

它是一个接口, 提供了两种实现: 缓存的实现、数据库的实现

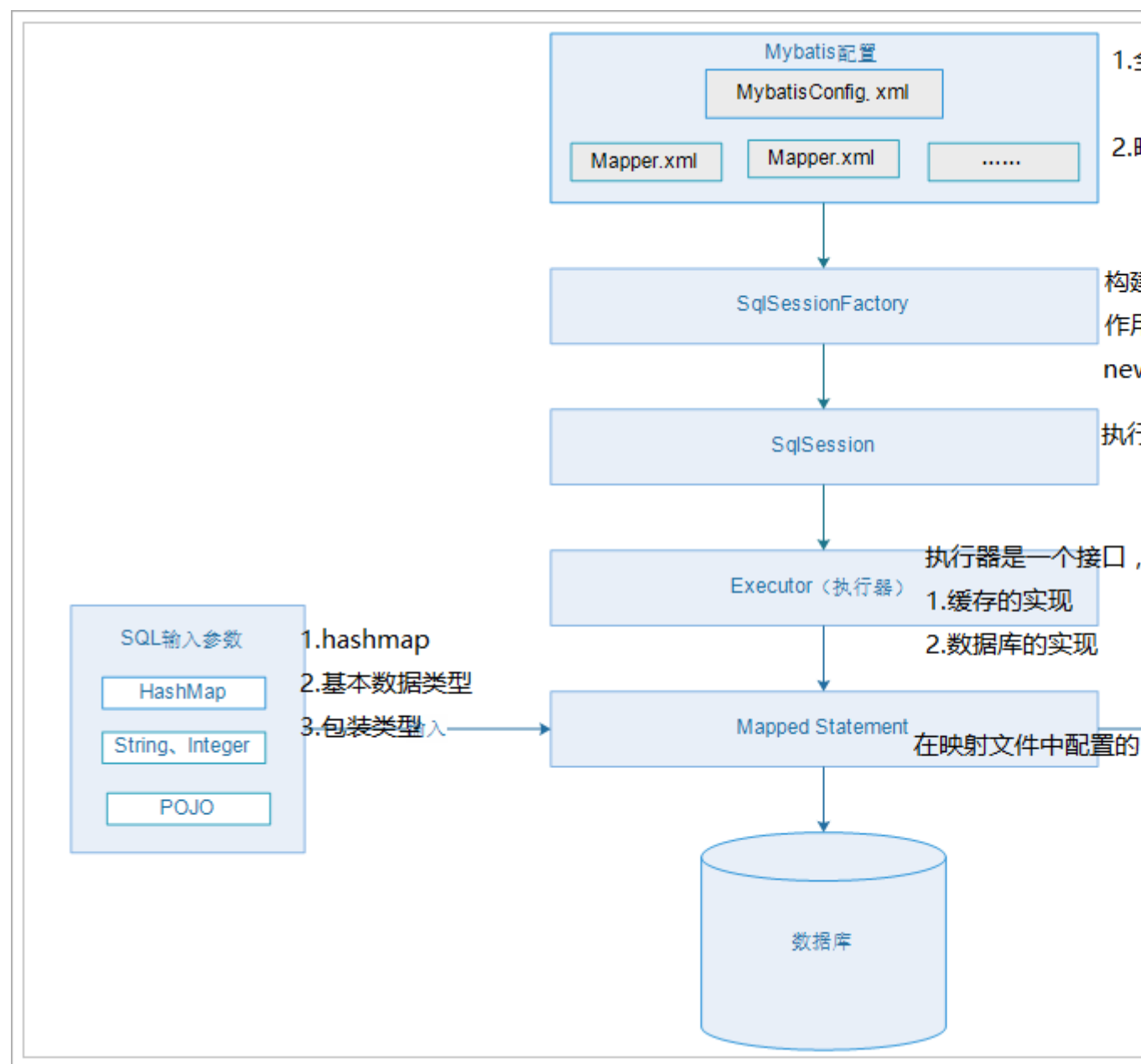
5、Mapped Statement

在映射文件里面配置, 包含 3 部分内容:

具体的 sql, sql 执行所需的参数类型, sql 执行结果的封装类型

参数类型和结果集封装类型包括 3 种:





`HashMap`, 基本数据类型, `pojo`

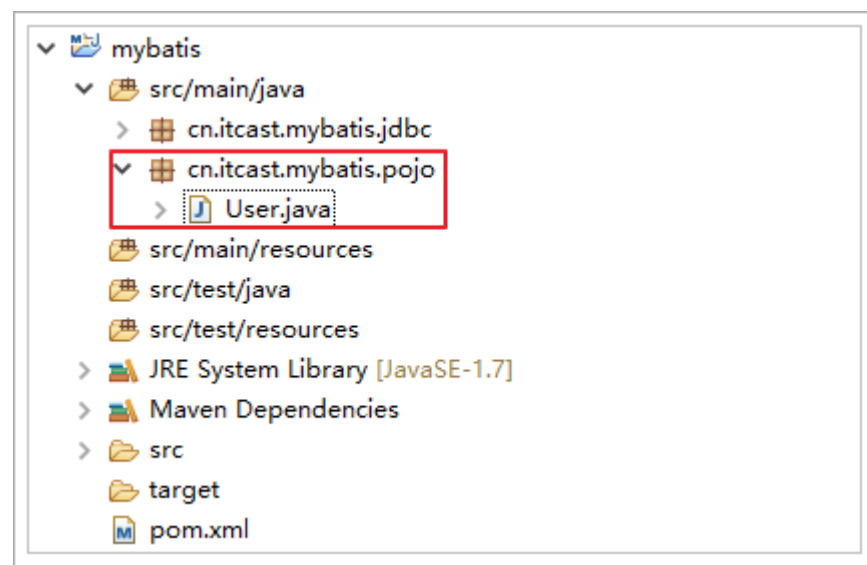


4.快速入门（Mybatis 第一个程序）

创建包 `cn.itcast.mybatis.pojo`

pojo 中的 `User.java` 参照课前资料：

esktop > Mybatis > Mybatis_Day01 > 课前资料 > pojo			
名称	修改日期	类型	大小
 Item.java	2014/11/26 21:52	JAVA 文件	
 Order.java	2014/11/26 21:53	JAVA 文件	
 Orderdetail.java	2014/11/26 21:53	JAVA 文件	
 User.java	2017/7/29 17:24	JAVA 文件	

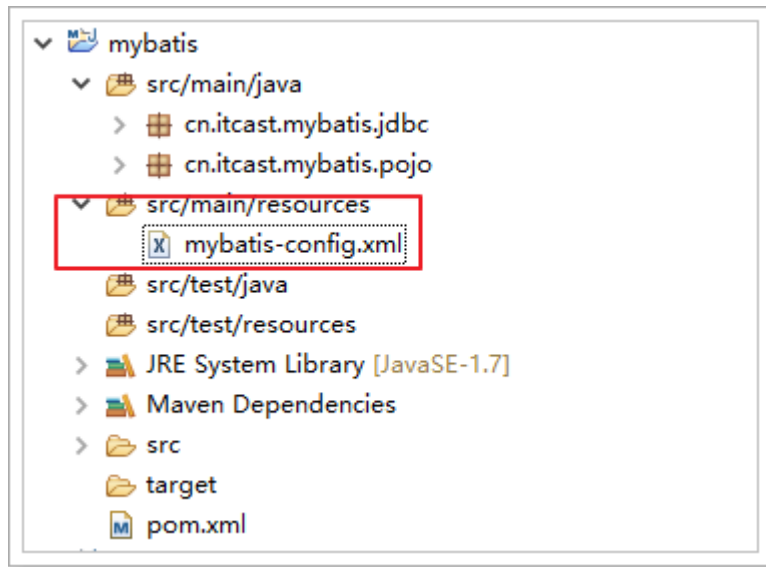


4.1. 引入依赖 (pom.xml)

参考 itcast-parent 工程

```
<!-- Mybatis -->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
</dependency>
```

4.2. 全局配置文件 (mybatis-config.xml)



Mybatis-config.xml 内容，参照官方文档：

```
<?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>

  <!-- 环境：说明可以配置多个，default:指定生效的环境 -->
  <environments default="development">

    <!-- id:环境的唯一标识 -->
    <environment id="development">

      <!-- 事务管理器，type: 类型 -->
      <transactionManager type="JDBC" />

      <!-- 数据源：type-池类型的数据源 -->
      <dataSource type="POOLED">
        <property name="driver"
value="com.mysql.jdbc.Driver" />
        <property name="url"
value="jdbc:mysql://127.0.0.1:3306/mybatis-49" />
        <property name="username" value="root" />
        <property name="password" value="root" />
      </dataSource>
    </environment>

  </environments>

</configuration>
```

```

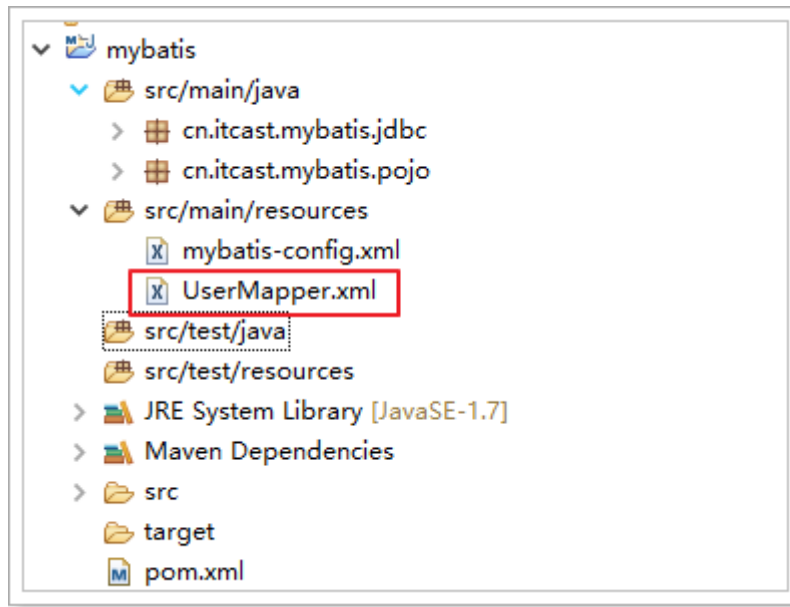
</environments>

<!-- 映射文件 -->

<mappers>
    <mapper resource="UserMapper.xml" />
</mappers>
</configuration>

```

4.3. 映射文件 (UserMapper.xml)



UserMapper.xml 映射文件内容，参照官方文档：

```

<?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">

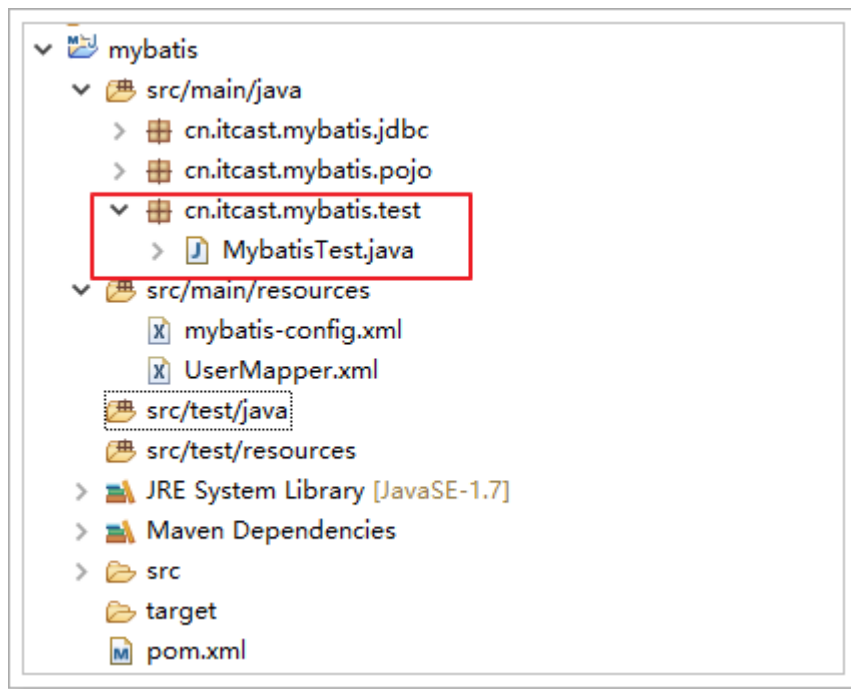
<!-- namespace(命名空间): 映射文件的唯一标识 -->
<mapper namespace="UserMapper">

    <!-- 查询的statement, id: 在同一个命名空间下的唯一标识,
resultType: sql语句的结果集封装类型 -->
    <select id="queryUserById" resultType="User">
        select * from tb_user where id=#{id}
    </select>

```

```
</mapper>
```

4.4. 编写 mybatis 程序 (MybatisTest.java)



MybatisTest.java 内容:

```
public static void main(String[] args) throws IOException {

    SqlSession sqlSession = null;
    try {
        // 指定mybatis的全局配置文件
        String resource = "mybatis-config.xml";
        // 读取mybatis-config.xml配置文件
        InputStream inputStream =
Resources.getResourceAsStream(resource);

        // 构建sqlSessionFactory
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

        // 获取sqlSession回话
        sqlSession = sqlSessionFactory.openSession();
```

```

        // 执行查询操作，获取结果集。参数：1-命名空间
        (namespace) + "." + statementId, 2 - sql 的占位符参数

        User user =
sqlSession.selectOne("UserMapper.queryUserById", 11);
        System.out.println(user);
    } finally {
        // 关闭连接
        if (sqlSession != null) {
            sqlSession.close();
        }
    }
}

```

4.5. 测试运行报错

```

... 4 more
Caused by: org.apache.ibatis.builder.BuilderException: Error resolving class
    at org.apache.ibatis.builder.BaseBuilder.resolveClass(BaseBuilder.java:100)
    at org.apache.ibatis.builder.xml.XMLStatementBuilder.parseStatementElement(XMLStatementBuilder.java:60)
    at org.apache.ibatis.builder.xml.XMLMapperBuilder.buildStatementFromElement(XMLMapperBuilder.java:130)
    at org.apache.ibatis.builder.xml.XMLMapperBuilder.buildStatementFromElement(XMLMapperBuilder.java:130)
    at org.apache.ibatis.builder.xml.XMLMapperBuilder.configurationElement(XMLMapperBuilder.java:130)
    ... 7 more
Caused by: org.apache.ibatis.type.TypeException: Could not resolve type alias: User
    at org.apache.ibatis.type.TypeAliasRegistry.resolveAlias(TypeAliasRegistry.java:65)
    at org.apache.ibatis.builder.BaseBuilder.resolveAlias(BaseBuilder.java:100)
    at org.apache.ibatis.builder.BaseBuilder.resolveClass(BaseBuilder.java:100)
    ... 11 more
Caused by: java.lang.ClassNotFoundException: Cannot find class: User
    at org.apache.ibatis.io.ClassLoaderWrapper.classForName(ClassLoaderWrapper.java:100)
    at org.apache.ibatis.io.ClassLoaderWrapper.classForName(ClassLoaderWrapper.java:100)
    at org.apache.ibatis.io.Resources.classForName(Resources.java:256)
    at org.apache.ibatis.type.TypeAliasRegistry.resolveAlias(TypeAliasRegistry.java:65)
    ... 13 more

```

原因：

```

4      http://mybatis.org/schema/mybatis-3-mapper.xsd
5  <!-- namespace(命名空间): 映射文件的唯一标识 -->
6  <mapper namespace="UserMapper">
7
8      <!-- 查询的statement, id: 在同一个命名空间下的唯一标识, resultType: sql语句的结果集封装
9  <select id="queryUserById" resultType="cn.itcast.mybatis.pojo.User"
10         select * from tb_user where id=#{id}
11 </select>

```

把User改成User的全路径

4.6. 其他典型出错

```

<terminated> MybatisTest [Java Application] C:\Resource\jdk7u80_x64\bin\javaw.exe (2017年7月21日 下午4:23:01)
Exception in thread "main" org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause: java.lang.IllegalArgumentException: Mapped Statements collection:
### Cause: java.lang.IllegalArgumentException: Mapped Statements collection:
    at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:25)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:145)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:145)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:130)
    at cn.itcast.mybatis.test.MybatisTest.main(MybatisTest.java:28)
Caused by: java.lang.IllegalArgumentException: Mapped Statements collection:
    at org.apache.ibatis.session.Configuration$StrictMap.get(Configuration.java:446)
    at org.apache.ibatis.session.Configuration.getMappedStatement(Configuration.java:385)
    at org.apache.ibatis.session.Configuration.getMappedStatement(Configuration.java:385)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:145)
    ... 3 more

```

原因:

```

5 <configuration>
6   <environments default="development">
7     <environment id="development">
8       <transactionManager type="JDBC" />
9       <dataSource type="POOLED">
10        <property name="driver" value="com.mysql.jdbc.Driver" />
11        <property name="url" value="jdbc:mysql://127.0.0.1:3306/" />
12        <property name="username" value="root" />
13        <property name="password" value="root" />
14      </dataSource>
15    </environment>
16  </environments>
17  <!-- <mappers>
18    <mapper resource="UserMapper.xml" />
19  </mappers> -->
20 </configuration>

```

如果映射文件未引入，

5.引入 log 日志

打印日志 2 个步骤：

- 1、在 pom.xml 中，引入 slf4j 的依赖
- 2、在 src/main/resources 目录下添加 log4j.properties 文件

5.1. 引入日志依赖包（pom.xml）

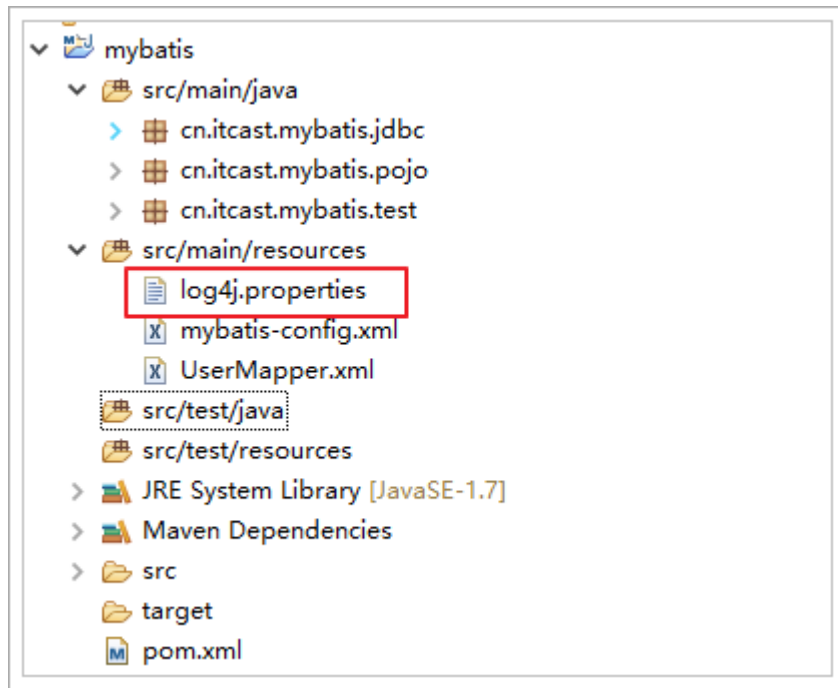
参照父工程

```

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
</dependency>

```

5.2. 添加 log4j.properties



内容如下：

```
log4j.rootLogger=DEBUG,A1
log4j.logger.org.apache=DEBUG
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd
HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n
```

5.3. 日志输出

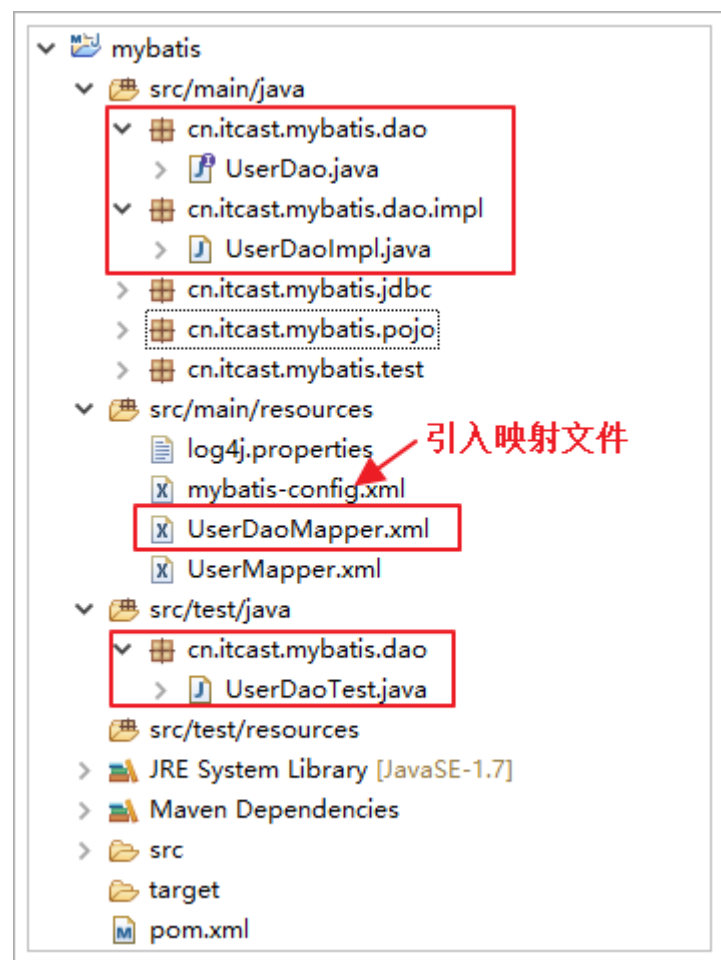
控制台截图：


```
[org.apache.ibatis.io.ResolverUtil]-[DEBUG] Reader entry: <?xml version="1.0" encoding="UTF-8"
[org.apache.ibatis.io.ResolverUtil]-[DEBUG] Checking to see if class cn.itcast.mybatis.mapper.U
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Opening JDBC Connection
[org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] Created connection 995235283.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Setting autocommit to false on JDB
[MyMapper.selectUser]-[DEBUG] ==> Preparing: select * from tb_user where id = ?
[MyMapper.selectUser]-[DEBUG] ==> Parameters: 1(Long)
[MyMapper.selectUser]-[DEBUG] <==      Total: 1
password=123456, name=张三, age=30, sex=1, birthday=Wed Aug 08 00:00:00 CST 1984, created=Fri Sep
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Resetting autocommit to true on JD
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Closing JDBC Connection [com.mysql
[org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] Returned connection 995235283 to
```

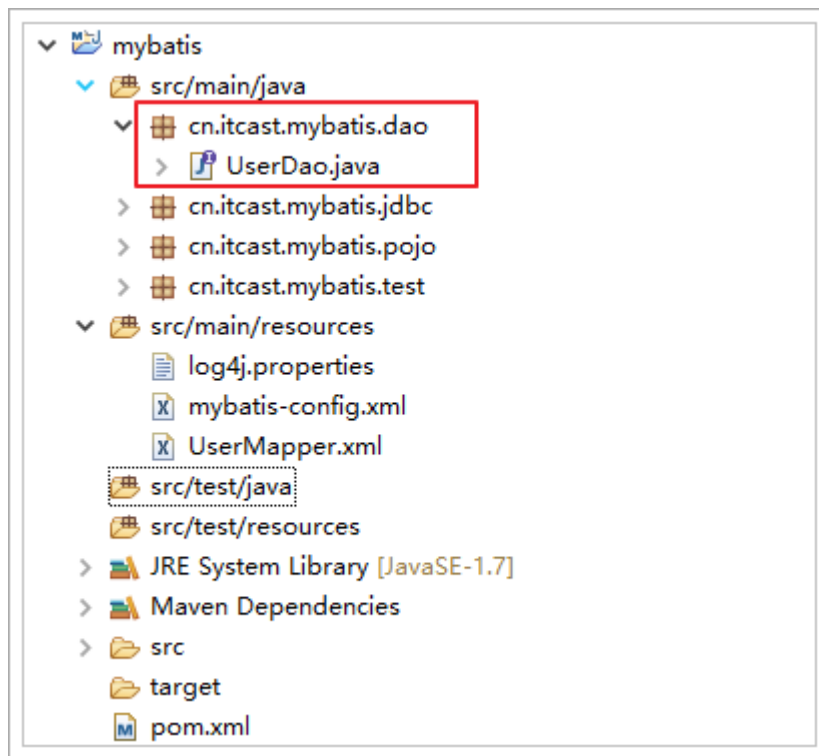
5.4. MyBatis 使用步骤总结

- 1) 配置 mybatis-config.xml 全局的配置文件 (1、数据源, 2、外部的 mapper)
- 2) 创建 SqlSessionFactory
- 3) 通过 SqlSessionFactory 创建 SqlSession 对象
- 4) 通过 SqlSession 操作数据库 CRUD
- 5) 调用 session.commit()提交事务
- 6) 调用 session.close()关闭会话

6.完整的 CRUD 操作



6.1. 创建 UserDao 接口



UserDao 接口的内容:

```
public interface UserDao {

    /**
     * 根据id获取用户信息
     * @param id
     * @return
     */
    public User queryUserById(Long id);

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

    /**
     * 新增用户
```

```

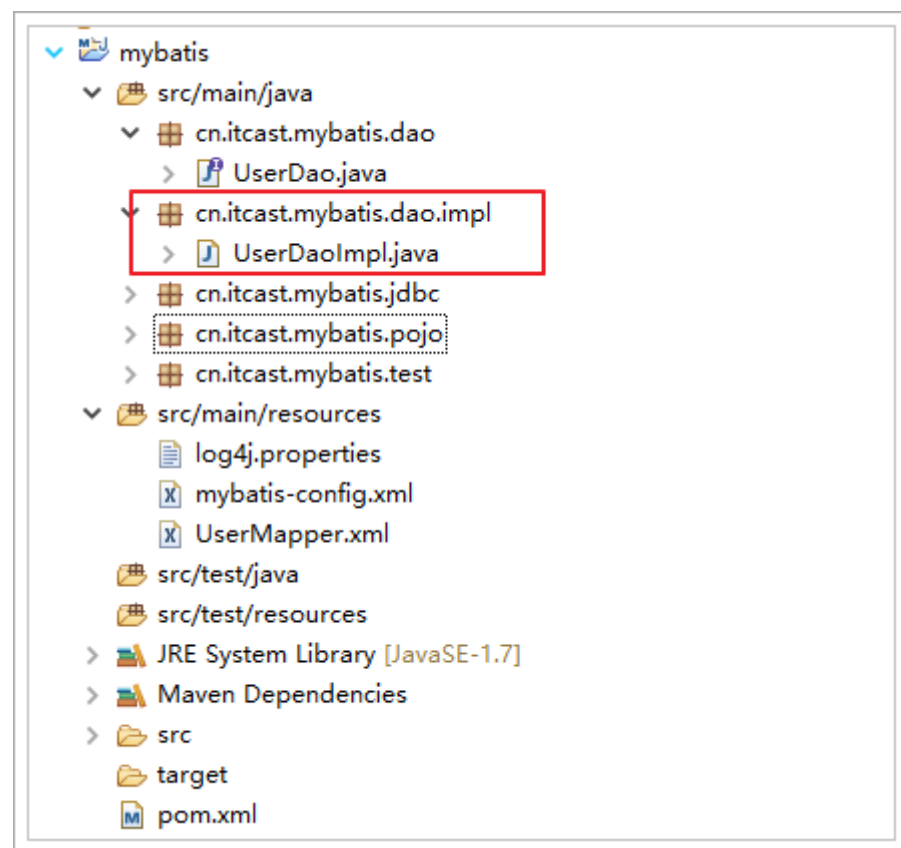
    * @param user
    */
    public void insertUser(User user);

    /**
     * 更新用户信息
     * @param user
     */
    public void updateUser(User user);

    /**
     * 根据id删除用户信息
     * @param id
     */
    public void deleteUserById(Long id);
}

```

6.2. 创建 UserDaoImpl



实现 UserDao 接口:

```
public class UserDaoImpl implements UserDao {

    private SqlSession sqlSession;

    public UserDaoImpl(SqlSession sqlSession){
        this.sqlSession = sqlSession;
    }

    @Override
    public User queryUserById(Long id) {
        return
this.sqlSession.selectOne("UserDaoMapper.queryUserById", id);
    }

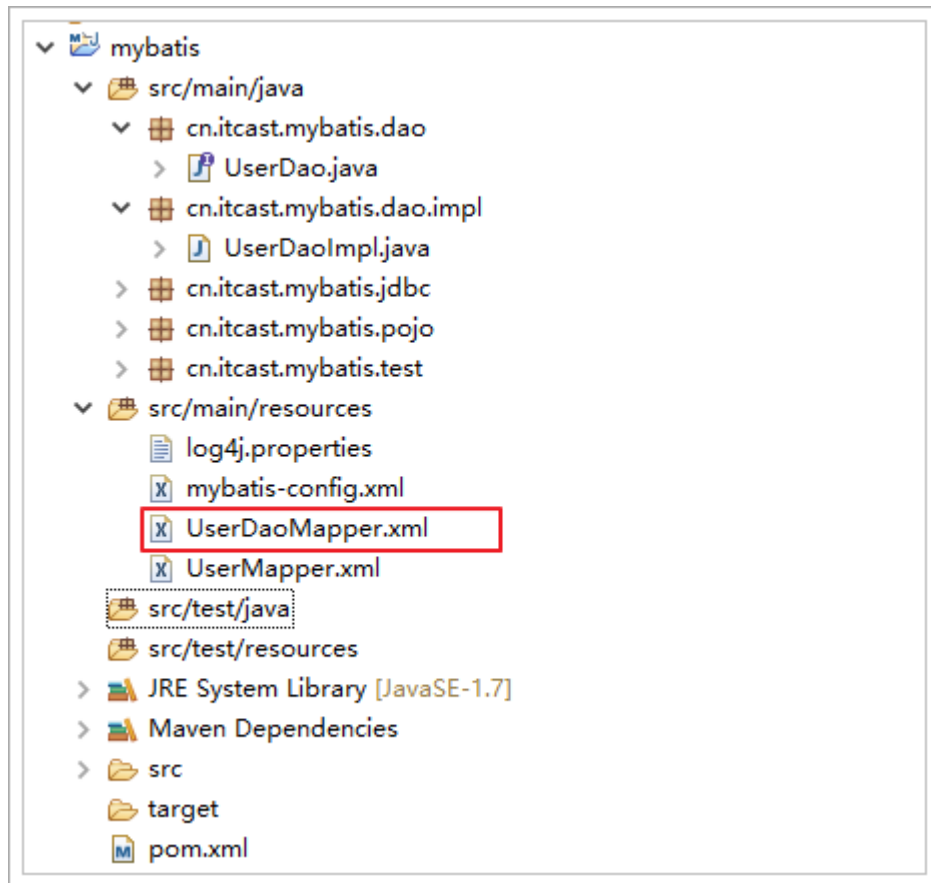
    @Override
    public List<User> queryUserAll() {
        return
this.sqlSession.selectList("UserDaoMapper.queryUserAll");
    }

    @Override
    public void insertUser(User user) {
        this.sqlSession.insert("UserDaoMapper.insertUser",
user);
        this.sqlSession.commit();
    }

    @Override
    public void updateUser(User user) {
        this.sqlSession.update("UserDaoMapper.updateUser",
user);
        this.sqlSession.commit();
    }

    @Override
    public void deleteUserById(Long id) {
        this.sqlSession.delete("UserDaoMapper.deleteUserById",
id);
        this.sqlSession.commit();
    }
}
```

6.3. 编写 UserDaoMapper.xml



编写对应的映射文件:

```
<?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="UserDaoMapper">

  <select id="queryUserById"
resultType="cn.itcast.mybatis.pojo.User">
    select * from tb_user where id = #{id}
  </select>

  <select id="queryUserAll"
resultType="cn.itcast.mybatis.pojo.User">
    select * from tb_user
  </select>
```

```

    <insert id="insertUser"
parameterType="cn.itcast.mybatis.pojo.User">
        INSERT INTO tb_user (
            user_name,
            password,
            name,
            age,
            sex,
            birthday,
            created,
            updated
        )
        VALUES
        (
            #{userName},
            #{password},
            #{name},
            #{age},
            #{sex},
            #{birthday},
            NOW(),
            NOW()
        );
    </insert>

    <update id="updateUser"
parameterType="cn.itcast.mybatis.pojo.User">
        UPDATE tb_user
        SET
            user_name = #{userName},
            password = #{password},
            name = #{name},
            age = #{age},
            sex = #{sex},
            birthday = #{birthday},
            updated = NOW()
        WHERE
            (id = #{id});
    </update>

    <delete id="deleteUserById" parameterType="java.lang.Long">
        delete from tb_user where id=#{id}
    </delete>

```

```
</mapper>
```

6.4. 引入 UserDaoMapper.xml

在 mybatis-config.xml 中引入 UserDaoMapper.xml 映射文件：

```
</environments>
<mappers>
  <mapper resource="UserMapper.xml"/>
  <mapper resource="UserDaoMapper.xml"/>
</mappers>
```

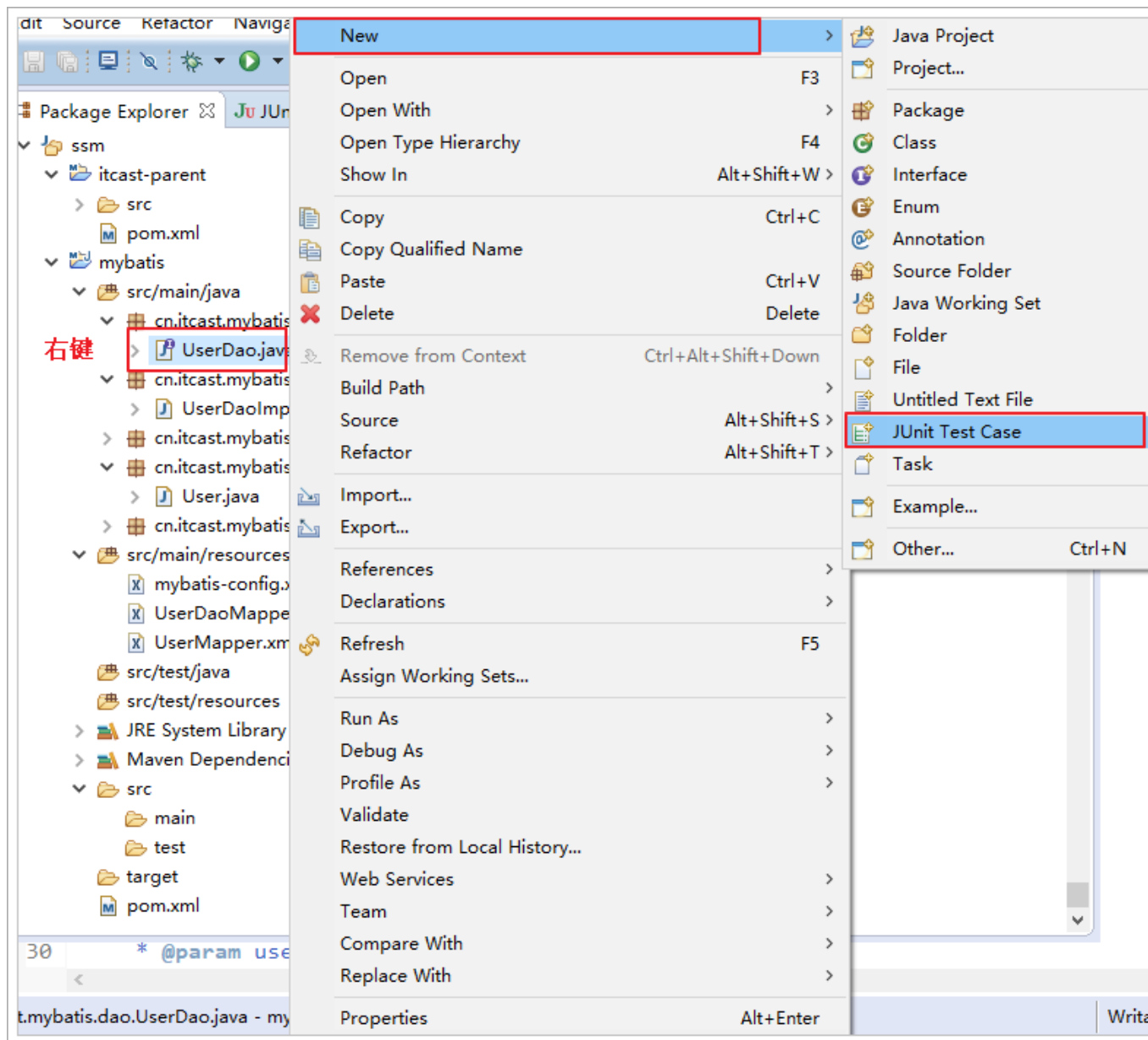
6.5. 测试 UserDao

针对 UserDao 的测试，咱们使用 Junit 进行测试。

1、引入 Junit 依赖：参照父工程，在 mybatis 工程中的 pom.xml 中引入 junit 的依赖

```
<!-- 单元测试 -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <scope>test</scope>
</dependency>
```

2、创建 junit 测试用例：右键 UserDao—>New—>JUnit Test Case



New JUnit Test Case

JUnit Test Case

Warning: Class under test 'cn.itcast.mybatis.dao.UserDao' is an interface.

☐ New JUnit 3 test

☒ New JUnit 4 test

Source folder:

mybatis/src/test/java

Browse...

Package:

cn.itcast.mybatis.dao

Browse...

Name:

UserDaoTest

Superclass:

java.lang.Object

Browse...

Which method stubs would you like to create?

☐ setUpBeforeClass()

☐ tearDownAfterClass()

☒ setUp()

☐ tearDown()

☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

cn.itcast.mybatis.dao.UserDao

Browse...

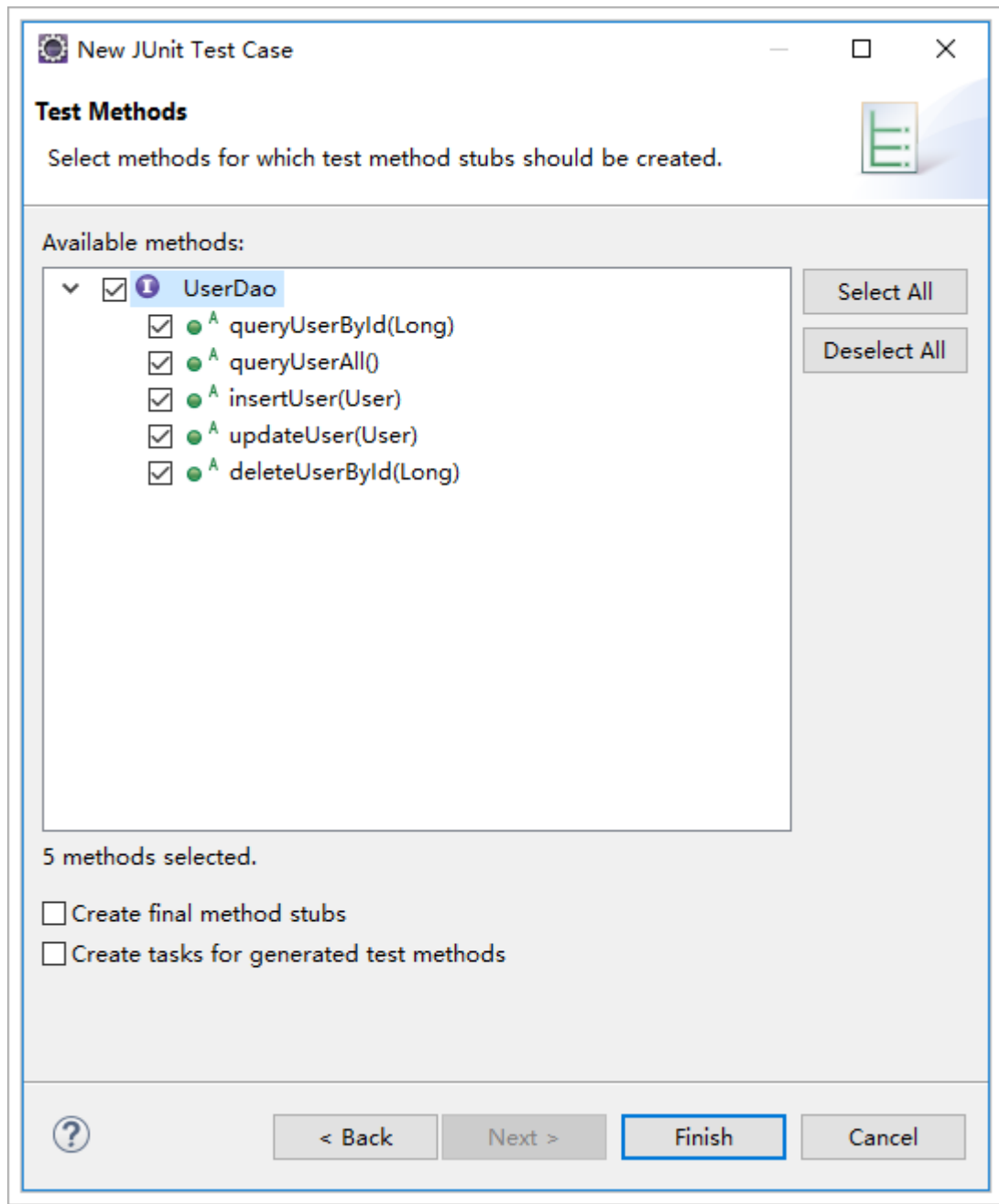
?

< Back

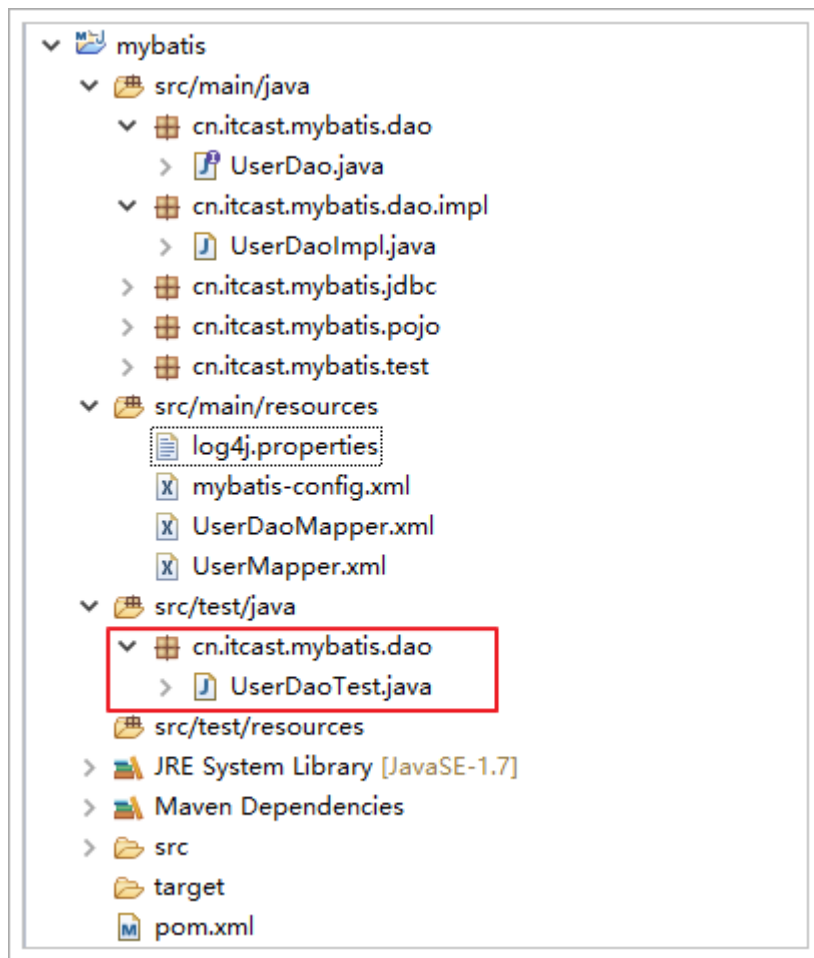
Next >

Finish

Cancel



创建完成之后的目录结构：



3、编写测试用例：

```
public class UserDaoTest {  
  
    private UserDao userDao;  
  
    @Before  
    public void setUp() throws Exception {  
  
        String resource = "mybatis-config.xml";  
  
        // 读取配置文件  
  
        InputStream inputStream =  
Resources.getResourceAsStream(resource);  
  
        // 构建sqlSessionFactory  
  
        SqlSessionFactory sqlSessionFactory = new  
SqlSessionFactoryBuilder().build(inputStream);  
    }  
}
```

```
// 获取sqlSession
SqlSession sqlSession = sqlSessionFactory.openSession();

// 初始化userDao
this.userDao = new UserDaoImpl(sqlSession);
}

@Test
public void testQueryUserById () {
    User user = this.userDao.queryUserById(11);
    System.out.println(user);
}

@Test
public void testQueryUserAll() {
    List<User> users = this.userDao.queryUserAll();
    for (User user : users) {
        System.out.println(user);
    }
}

@Test
public void testInsertUser() {
    User user = new User();
    user.setAge(18);

    user.setName("柳岩");

    user.setPassword("123456");
    user.setUsername("yanyan");
    user.setSex(3);
    user.setBirthday(new Date());
    this.userDao.insertUser(user);
}

@Test
public void testUpdateUser() {
    // 查询
    User user = this.userDao.queryUserById(71);

    // 更新
    user.setAge(28);
}
```

```
        user.setPassword("111111");
        this.userDao.updateUser(user);
    }

    @Test
    public void testDeleteUserById() {
        this.userDao.deleteUserById(71);
    }
}
```

6.6. 总结

- 1、 编写 UserDao 接口
- 2、 编写 UserDao 的实现类 UserDaoImpl 及映射文件 UserDaoMapper.xml
- 3、 修改全局配置文件，引入 UserDaoMapper.xml
- 4、 编写 UserDao 的 Junit Test Case 测试用例

6.7. 解决 UserName 为 null

查询数据的时候，查不到 userName 的信息，原因：数据库的字段名是 user_name

POJO 中的属性名是 userName

两端不一致，造成 mybatis 无法填充对应的字段信息。修改方法：在 sql 语句中使用别名

解决方案 1：在 sql 语句中使用别名

修改映射文件（UserDaoMapper.xml）中的 sql 语句

```
<select id="queryUserAll" resultType="cn.itcast.mybatis.pojo.User">
    select *,user_name as username from tb_user
</select>

<select id="queryUserById" resultType="cn.itcast.mybatis.pojo.User">
    select *,user_name as username from tb_user where id=#{id}
</select>
```

控制台 Log 日志:

```
2017-04-11 16:56:36,784 [main] [UserDaoMapper.queryUserAll]-[DEBUG] ==> Preparing: select *,user
2017-04-11 16:56:36,826 [main] [UserDaoMapper.queryUserAll]-[DEBUG] ==> Parameters:
2017-04-11 16:56:36,859 [main] [UserDaoMapper.queryUserAll]-[DEBUG] <==      Total: 7
User [id=1, username=zhangsan, password=123456, name=张三, age=30, sex=1, birthday=Wed Aug 08 00:00:00
User [id=2, username=lisi, password=123456, name=李四, age=21, sex=2, birthday=Tue Jan 01 00:00:00
User [id=3, username>wangwu, password=123456, name=王五, age=22, sex=2, birthday=Sun Jan 01 00:00:00
```

7. 动态代理 Mapper 实现类

7.1. 思考 CRUD 中的问题

- 1、接口->实现类->mapper.xml。
- 2、实现类中，使用 mybatis 的方式非常类似。
- 3、sql statement 硬编码到 java 代码中。

思考：能否只写接口，不书写实现类，只编写 Mapper.xml 即可

因为在 dao (mapper) 的实现类中对 sqlsession 的使用方式很类似。mybatis 提供了

接口的动态代理

Mapper 接口的动态代理实现，需要满足以下条件：

1. 映射文件中的命名空间与 Mapper 接口的全路径一致
2. 映射文件中的 statementId 与 Mapper 接口的方法名保持一致

3. 映射文件中的 statement 的 ResultType 必须和 mapper 接口方法的返回类型一致
(即使不采用动态代理, 也要一致)
4. 映射文件中的 statement 的 parameterType 必须和 mapper 接口方法的参数类型一致 (不一定, 该参数可省略)

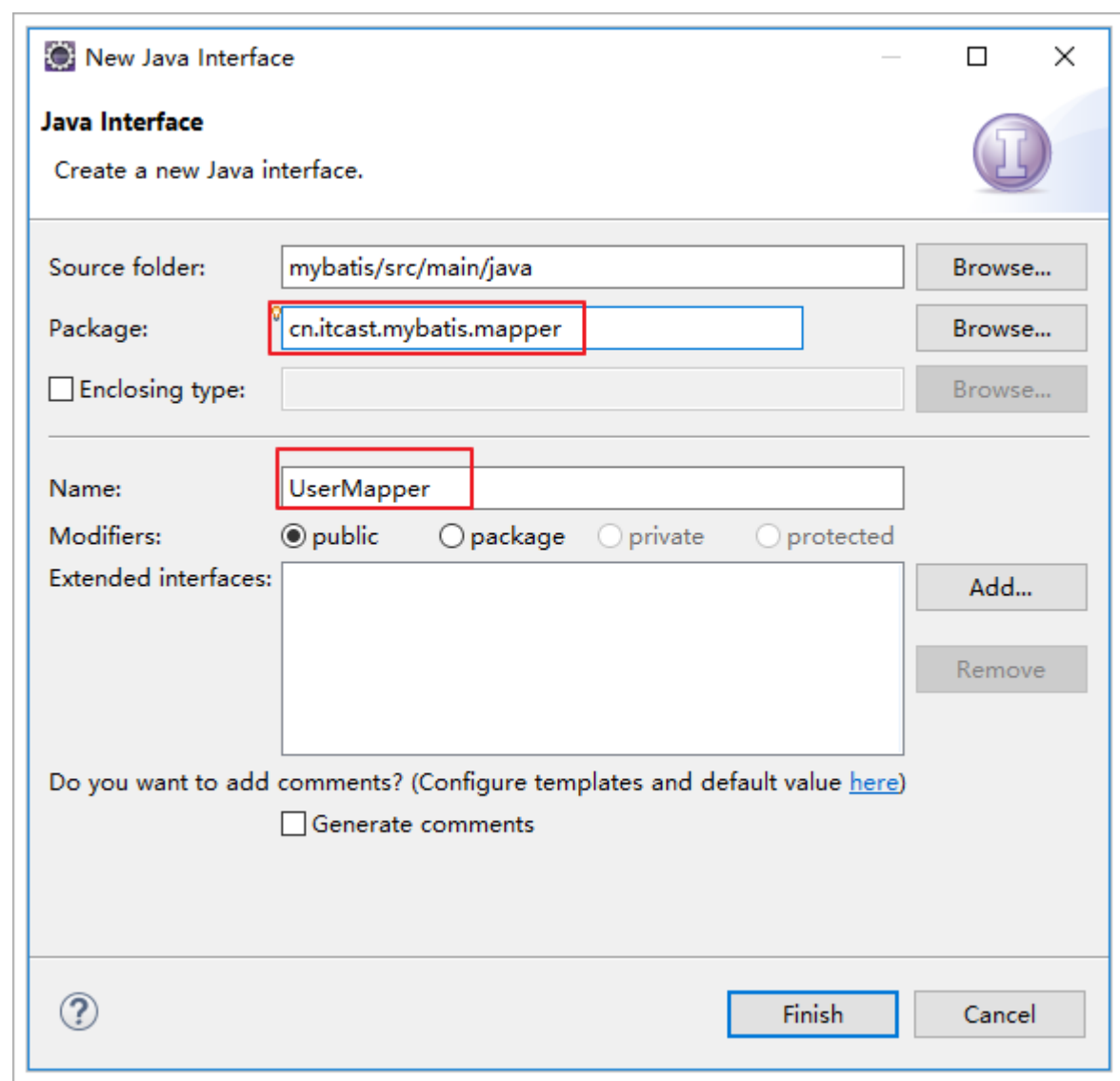
7.2. 使用动态代理改造 CRUD

在 mybatis 中, 持久层的 XxxDao 通常习惯上命名为 XxxMapper (例如: 以前命名为 UserDao 的接口, 现在命名为 UserMapper)。

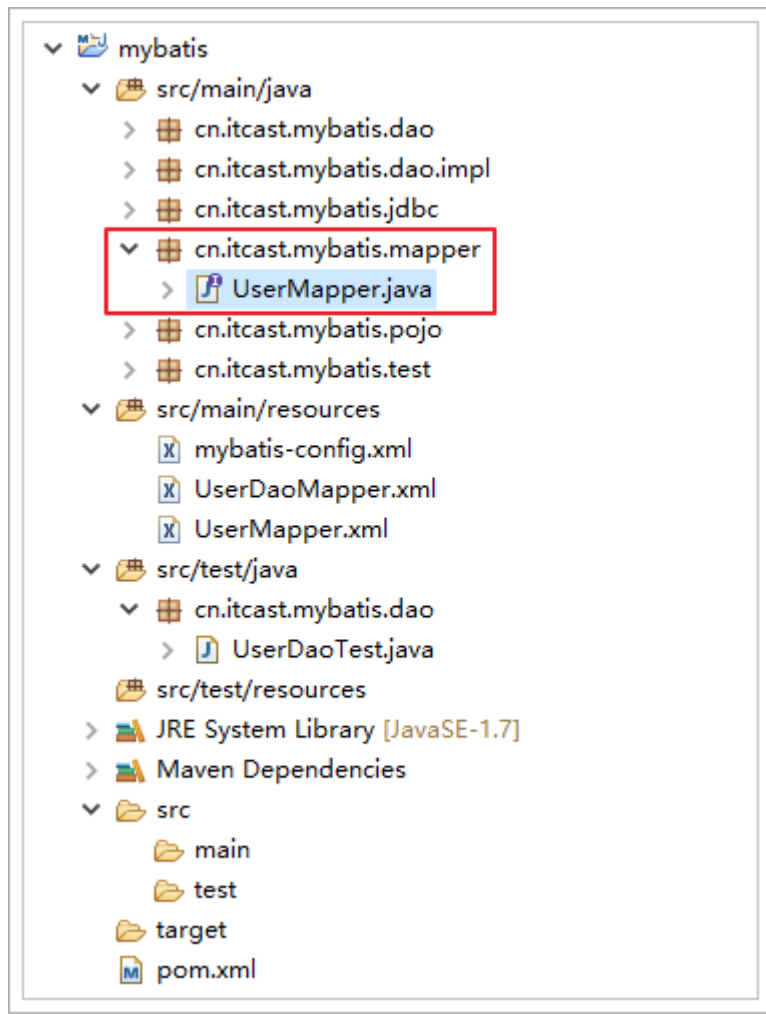
当然, 仍有少部分项目会保留 Dao 的命名, 以后见到 Mapper 或者 Dao 的命名都不要奇怪。

采用动态代理之后, 只剩下 **UserMapper 接口**、**UserMapper.xml 映射文件** 以及 UserMapper 接口的测试文件, 即可以少写一个接口的实现类

6.2.1. 创建 UserMapper 接口



添加后:



UserMapper.java 的内容参考 UserDao.java 的内容。(可以从 UserDao 直接 Copy 过来)

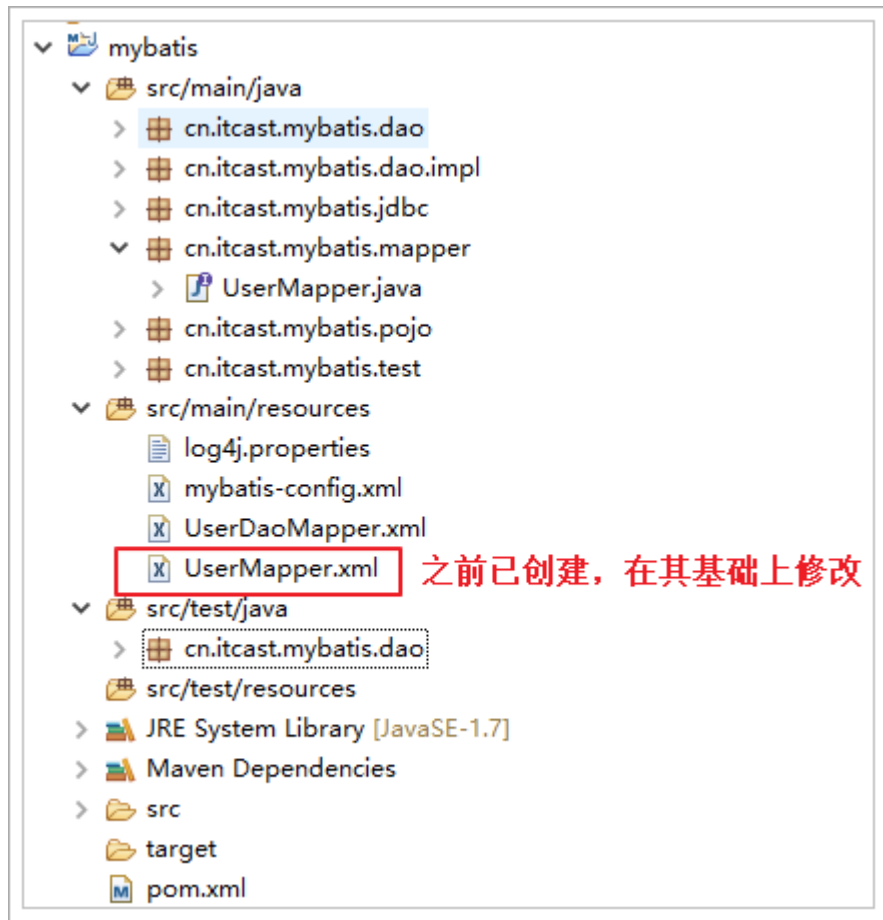
```
public interface UserMapper {  
    /**  
     * 根据id获取用户信息  
     * @param id  
     * @return  
     */  
    public User queryUserById(Long id);  
  
    /**  
     * 查询所有用户  
     * @return  
     */  
    public List<User> queryUserAll();  
  
    /**
```

```
    * 新增用户
    * @param user
    */
    public void insertUser(User user);

    /**
     * 更新用户信息
     * @param user
     */
    public void updateUser(User user);

    /**
     * 根据id删除用户信息
     * @param id
     */
    public void deleteUserById(Long id);
}
```

6.2.2. UserMapper 映射文件



对应的 UserMapper 映射文件，之前咱们已创建，但是内容参考 UserDaoMapper。

名称空间必须改成 UserMapper 接口的全路径，StatementId 必须和接口方法名一致，结

果集的封装类型已经和方法的返回类型一致

```
<?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="cn.itcast.mybatis.mapper.UserMapper">

  <select id="queryUserById"
resultType="cn.itcast.mybatis.pojo.User">
    select * from tb_user where id = #{id}
  </select>

  <select id="queryUserAll"
```

```
resultType="cn.itcast.mybatis.pojo.User">
    select * from tb_user
</select>

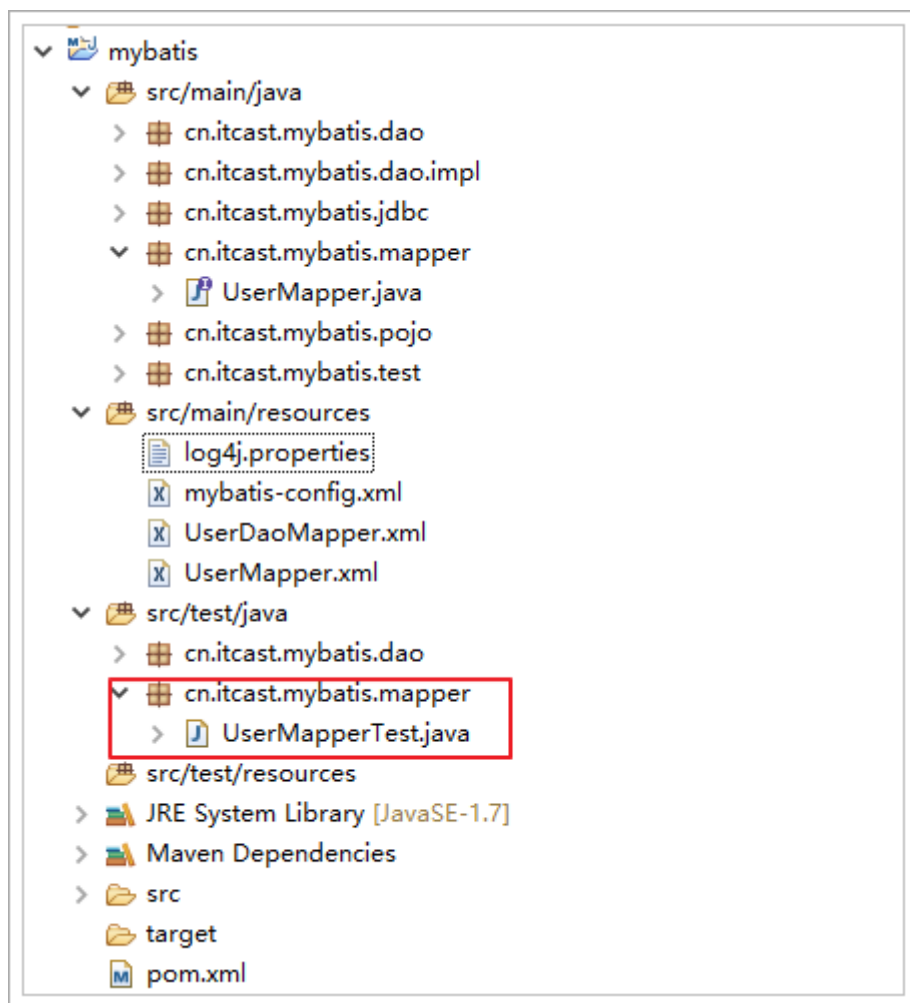
<insert id="insertUser"
parameterType="cn.itcast.mybatis.pojo.User">
    INSERT INTO tb_user (
        user_name,
        password,
        name,
        age,
        sex,
        birthday,
        created,
        updated
    )
    VALUES
    (
        #{userName},
        #{password},
        #{name},
        #{age},
        #{sex},
        #{birthday},
        NOW(),
        NOW()
    );
</insert>

<update id="updateUser"
parameterType="cn.itcast.mybatis.pojo.User">
    UPDATE tb_user
    SET
        user_name = #{userName},
        password = #{password},
        name = #{name},
        age = #{age},
        sex = #{sex},
        birthday = #{birthday},
        updated = NOW()
    WHERE
        (id = #{id});
</update>
```

```
<delete id="deleteUserById" parameterType="java.lang.Long">
    delete from tb_user where id=#{id}
</delete>

</mapper>
```

6.2.3. UserMapperTest 测试



添加 UserMapper.java 的测试用例。只需要修改一句代码即可

```
public class UserMapperTest {

    private UserMapper userMapper;

    @Before
    public void setUp() throws Exception {
```

```

        // 读取mybatis的全局配置文件
        InputStream inputStream =
Resources.getResourceAsStream("mybatis-config.xml");

        // 构建sqlSessionFactory
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

        // 获取sqlSession会话
        SqlSession sqlSession = sqlSessionFactory.openSession();

        // 初始化userDao
        this.userMapper =
sqlSession.getMapper(UserMapper.class);
    }

    @Test
    public void testQueryUserAll() {
        List<User> userList = this.userMapper.queryUserAll();
        for (User user : userList) {
            System.out.println(user);
        }
    }
}

```

1、如果名称空间不和 mapper 接口的全路径保持一致

```

5 <mapper namespace="cn.itcast.mybatis.mapper.UserMapper1">
6
7 <select id="queryUserById" resultType="cn.itcast.mybatis.pojo.User">
8     select * from tb_user where id = #{id}
9 </select>
10
11 <select id="queryUserAll" resultType="cn.itcast.mybatis.pojo.User">
12     select * from tb_user
13 </select>
14
15 <insert id="insertUser" parameterType="cn.itcast.mybatis.pojo.User">

```

```
org.apache.ibatis.binding.BindingException: Type interface cn.itcast.mybatis.mapper
    at org.apache.ibatis.binding.MapperRegistry.getMapper(MapperRegistry.java:47)
    at org.apache.ibatis.session.Configuration.getMapper(Configuration.java:655)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.getMapper(DefaultSqlSession.java:114)
    at cn.itcast.mybatis.mapper.UserMapperTest.setUp(UserMapperTest.java:35)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:59)
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:71)
    at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:27)
    at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:263)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:101)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:71)
    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:231)
    at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:60)
    at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:229)
    at org.junit.runners.ParentRunner.access$000(ParentRunner.java:50)
    at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:222)
    at org.junit.runners.ParentRunner.run(ParentRunner.java:300)
    at org.eclipse.jdt.internal.junit4.runner.JUnit4TestReference.run(JUnit4TestReference.java:16)
    at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:153)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:105)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.java:73)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:19)
```

7.mybatis-config.xml 配置

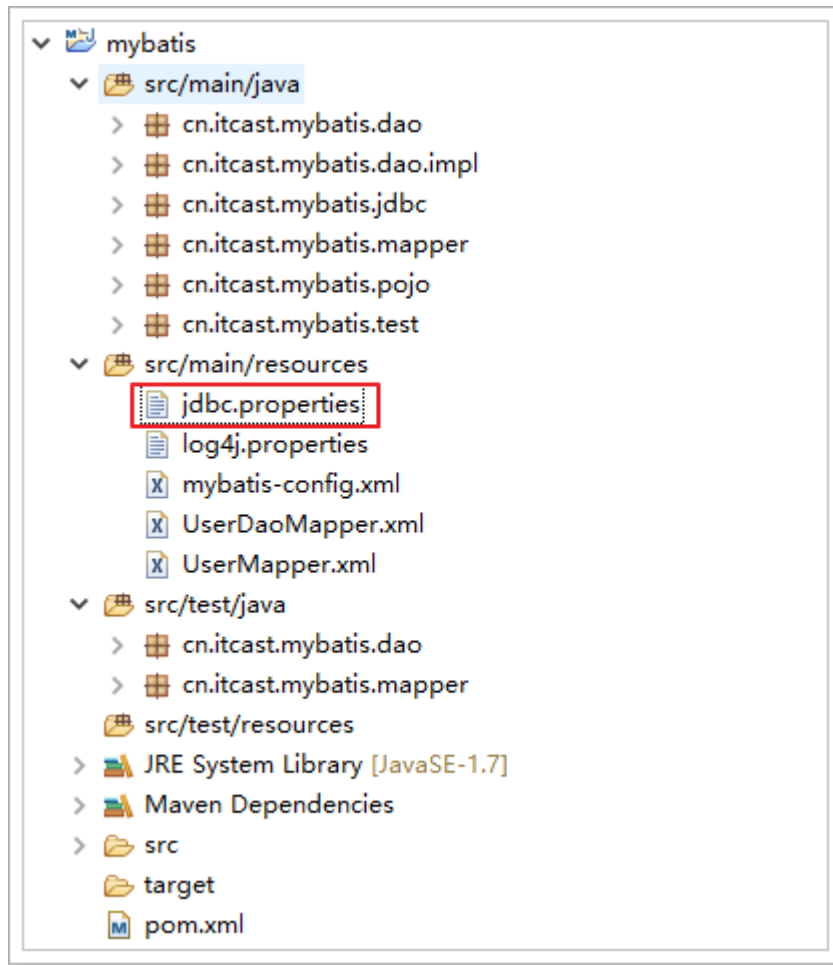
mybatis-config.xml 讲究严格的顺序，具体顺序遵循文档的顺序

MyBatis 的配置文件包含了影响 MyBatis 行为甚深的设置（ settings ）和属性（ properties ）信息。文档的顶

- configuration 配置
 - properties 属性
 - settings 设置
 - typeAliases 类型命名
 - typeHandlers 类型处理器
 - objectFactory 对象工厂
 - plugins 插件
 - environments 环境
 - environment 环境变量
 - transactionManager 事务管理器
 - dataSource 数据源
 - databaseIdProvider 数据库厂商标识
 - mappers 映射器

7.1. properties 属性读取外部资源

添加 jdbc.properties 资源文件：



jdbc.properties 资源文件内容:

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://127.0.0.1:3306/mybatis-44
username=root
password=root
```

在 Mybatis-config.xml 中引入 jdbc.properties 资源文件:

```
<?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>

  <!-- 引入外部资源文件, resource: 相对路径, url: 绝对路径 -->
  <properties resource="jdbc.properties" />
```

```

<!-- 环境：说明可以配置多个，default:指定生效的环境 -->
<environments default="development">
    <!-- id:环境的唯一标识 -->
    <environment id="development">
        <!-- 事务管理器，type: 类型 -->
        <transactionManager type="JDBC" />
        <!-- 数据源：type-池类型的数据源 -->
        <dataSource type="POOLED">
            <property name="driver" value="${driver}" />
            <property name="url" value="${url}" />
            <property name="username" value="${username}" />
            <property name="password" value="${password}" />
        </dataSource>
    </environment>
</environments>

<!-- 映射文件 -->
<mappers>
    <mapper resource="UserMapper.xml" />
    <mapper resource="UserDaoMapper.xml" />
</mappers>
</configuration>

```

通过 properties 引入外部资源文件之后，就可以通过\${xxx}的方式使用资源文件里的参数了。

7.2. settings 设置

settings 参数有很多，咱们只需要学习以下 4 个参数就行了，今天咱们先学习驼峰匹配。

设置参数	描述
cacheEnabled	该配置影响的所有映射器中配置的缓存的全局开关。

lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 可通过设置 <code>fetchType</code> 属性来覆盖该项的开关状态。
aggressiveLazyLoading	当启用时，带有延迟加载属性的对象的加载与否完全取决于对 用；反之，每种属性将会按需加载。
mapUnderscoreToCamelCase	是否开启自动驼峰命名规则（camel case）映射，即从经典数 A_COLUMN 到经典 Java 属性名 aColumn 的类似映射。

开启驼峰匹配：完成经典的数据库命名到 java 属性的映射

经典数据库命名：如果多个单词之间，通常使用下划线进行连接。

java 中命名：第二个单词首字母大写。

驼峰匹配：相当于去掉数据中的名字的下划线，和 java 进行匹配

查询数据的时候，查不到 `userName` 的信息，原因：数据库的字段名是 `user_name`，POJO

中的属性名字是 `userName`，两端不一致，造成 mybatis 无法填充对应的字段信息。修改

方法：在 sql 语句中使用别名

解决方案 1：在 sql 语句中使用别名

解决方案 2：参考驼峰匹配 --- mybatis-config.xml

mybatis-config.xml 中开启驼峰匹配：

```
<?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>
```

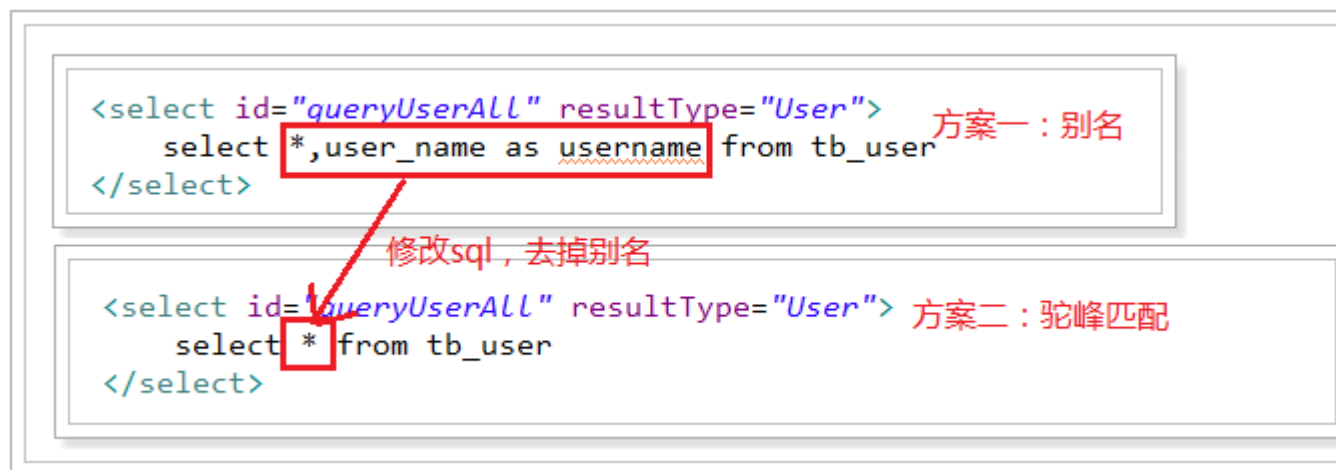
```

<!-- 引入外部资源文件, resource: classpath路径, url: 绝对路径
(不建议使用) -->
<properties resource="jdbc.properties"></properties>
<settings>

    <!-- 开启驼峰匹配: 经典的数据库列名 (多个单词下划线连接) 映射
    到经典的java属性名 (多个单词驼峰连接) -->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC" />
        <dataSource type="POOLED">
            <property name="driver" value="${driver}" />
            <property name="url" value="${url}" />
            <property name="username" value="${username}" />
            <property name="password" value="${password}" />
        </dataSource>
    </environment>
</environments>
<mappers>
    <mapper resource="UserMapper.xml" />
    <mapper resource="UserDaoMapper.xml" />
</mappers>
</configuration>

```

在 UserMapper.xml 中修改 sql 语句删除别名



测试日志截图，如下：

```
2017-04-12 12:44:30,506 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 12:44:30,547 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 12:44:30,586 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, b
User [id=2, userName=lisi, password=123456, name=李四, age=21, sex=2, birth
User [id=3, userName=wangwu, password=123456, name=王五, age=22, sex=2, bir
User [id=6, userName=lilei, password=123456, name=李磊, age=23, sex=1, birt
User [id=8, userName=xionгда, password=111111, name=柳岩, age=20, sex=2, bi
User [id=11, userName=liuyan3333, password=111111, name=柳岩, age=20, sex=2
User [id=15, userName=liuyan22222, password=2222, name=柳岩, age=20, sex=4,
```

删除 sql 中的别名，开启驼峰匹配后，仍然能够获取用户名，说明驼峰匹配起到了作用

7.3. typeAliases

之前咱们在映射文件中用到 java 类型时，都是使用类的全路径，书写起来非常麻烦

解决方案：

类型别名是为 Java 类型命名的一个短的名字。它只和 XML 配置有关，存在的意义仅在

于用来减少类完全限定名的冗余。（官方文档）

7.3.1. 方式一：typeAlias

```
<?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>

  <!-- 引入外部资源文件，resource: classpath路径, url: 绝对路径
  (不建议使用) -->

  <properties resource="jdbc.properties"></properties>
  <settings>
```

```

    <!-- 开启驼峰匹配：经典的数据库列名（多个单词下划线连接）映射
到经典的java属性名（多个单词驼峰连接） -->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
<typeAliases>
    <!-- 类型别名：type-pojo类的全路径，alias-别名名称（可随便
写，推荐和类名一致） -->
    <typeAlias type="cn.itcast.mybatis.pojo.User"
alias="user" />
</typeAliases>
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC" />
        <dataSource type="POOLED">
            <property name="driver" value="${driver}" />
            <property name="url" value="${url}" />
            <property name="username" value="${username}" />
            <property name="password" value="${password}" />
        </dataSource>
    </environment>
</environments>
<mappers>
    <mapper resource="UserMapper.xml" />
    <mapper resource="UserDaoMapper.xml" />
</mappers>
</configuration>

```

缺点：每个 pojo 类都要去配置。

7.3.2. 方式二：package

扫描指定包下的所有类，扫描之后的别名就是类名，大小写不敏感（不区分大小写），建议使用的时候和类名一致。

```

<typeAliases>
    <!-- 类型别名：type-pojo类的全路径，alias-别名名称（可随便

```

写，推荐和类名一致) -->

```
<!-- <typeAlias type="cn.itcast.mybatis.pojo.User"
alias="user" /> -->
```

<!-- 开启别名包扫描，name：包路径，扫描的别名就是类名，并且大

小写不敏感 -->

```
<package name="cn.itcast.mybatis.pojo"/>
</typeAliases>
```

在映射文件中使用类型别名：

```
<select id="queryUserAll" resultType="User">
  select * from tb_user
</select>

<select id="queryUserById" resultType="User" parameterType="Long">
  select * from tb_user where id=#{id}
</select>
```

包扫描别名

内置别名

已经为普通的 Java 类型内建了许多相应的类型别名。它们都是大小写不敏感的，需要注意的是由于重载原始类型的名称所做的特殊处理。

别名	映射的类型
<code>_byte</code>	byte
<code>_long</code>	long
<code>_short</code>	short
<code>_int</code>	int
<code>_integer</code>	int
<code>_double</code>	double
<code>_float</code>	float
<code>_boolean</code>	boolean

string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

7.4. typeHandlers (类型处理器)

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

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	<code>java.lang.Boolean, boolean</code>	任何兼容的布尔值
ByteTypeHandler	<code>java.lang.Byte, byte</code>	任何兼容的数字或字节类型
ShortTypeHandler	<code>java.lang.Short, short</code>	任何兼容的数字或短整型
IntegerTypeHandler	<code>java.lang.Integer, int</code>	任何兼容的数字和整型
LongTypeHandler	<code>java.lang.Long, long</code>	任何兼容的数字或长整型
FloatTypeHandler	<code>java.lang.Float, float</code>	任何兼容的数字或单精度浮点型
DoubleTypeHandler	<code>java.lang.Double, double</code>	任何兼容的数字或双精度浮点型
BigDecimalTypeHandler	<code>java.math.BigDecimal</code>	任何兼容的数字或十进制小数类型
StringTypeHandler	<code>java.lang.String</code>	CHAR 和 VARCHAR 类型
ClobTypeHandler	<code>java.lang.String</code>	CLOB 和 LONGVARCHAR 类型
NStringTypeHandler	<code>java.lang.String</code>	NVARCHAR 和 NCHAR 类型
NClobTypeHandler	<code>java.lang.String</code>	NCLOB 类型
ByteArrayTypeHandler	<code>byte[]</code>	任何兼容的字节流类型

<code>BlobTypeHandler</code>	<code>byte[]</code>	BLOB 和 LONGVARBINARY 类型
<code>DateTypeHandler</code>	<code>java.util.Date</code>	TIMESTAMP 类型
<code>DateOnlyTypeHandler</code>	<code>java.util.Date</code>	DATE 类型
<code>TimeOnlyTypeHandler</code>	<code>java.util.Date</code>	TIME 类型
<code>SqlTimestampTypeHandler</code>	<code>java.sql.Timestamp</code>	TIMESTAMP 类型
<code>SqlDateTypeHandler</code>	<code>java.sql.Date</code>	DATE 类型
<code>SqlTimeTypeHandler</code>	<code>java.sql.Time</code>	TIME 类型
<code>ObjectTypeHandler</code>	Any	其他或未指定类型
<code>EnumTypeHandler</code>	Enumeration Type	VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）
<code>EnumOrdinalTypeHandler</code>	Enumeration Type	任何兼容的 <code>NUMERIC</code> 或 <code>DOUBLE</code> 类型，作为位置存储（而不是代码本身）。

7.5. environments(环境)

MyBatis 可以配置成适应多种环境，例如，开发、测试和生产环境需要有不同的配置；

尽管可以配置多个环境，每个 `SqlSessionFactory` 实例只能选择其一。

虽然，这种方式也可以做到很方便的分离多个环境，但是实际使用场景下，我们更多的是选择使用 spring 来管理数据源，来做到环境的分离。

7.5.1. 方法一：default

添加一个 test（测试）环境，并在 default 参数中指向 test 环境。

```

<!-- 环境，可以配置多个，default: 指定采用哪个环境，值为环境id -->
<environments default="test">
  <!-- id: 唯一标识 -->
  <environment id="development">
    <!-- 事务管理器，JDBC类型事务管理器 -->
    <transactionManager type="JDBC" />
    <!-- 数据源，配置连接信息，池类型的数据源 -->
    <dataSource type="POOLED">
      <property name="driver" value="${driver}" />
      <property name="url" value="${url}" />
      <property name="username" value="${username}" />
      <property name="password" value="${password}" />
    </dataSource>
  </environment>
  <environment id="test">
    <!-- 事务管理器，JDBC类型事务管理器 -->
    <transactionManager type="JDBC" />
    <!-- 数据源，配置连接信息，池类型的数据源 -->
    <dataSource type="POOLED">
      <property name="driver" value="${driver}" />
      <property name="url" value="${url}" />
      <property name="username" value="${username}" />
      <property name="password" value="${password}" />
    </dataSource>
  </environment>
</environments>

```

7.5.2. 方法二：build 方法

通过 build 方法的重载方法

- build(Configuration config) : SqlSessionFactory - SqlSessionFactoryBuilder
- build(InputStream inputStream) : SqlSessionFactory - SqlSessionFactoryBuilder
- build(Reader reader) : SqlSessionFactory - SqlSessionFactoryBuilder
- build(InputStream inputStream, Properties properties) : SqlSessionFactory - SqlSessionFactoryBuilder
- build(InputStream inputStream, String environment) : SqlSessionFactory - SqlSessionFactoryBuilder
- build(Reader reader, Properties properties) : SqlSessionFactory - SqlSessionFactoryBuilder
- build(Reader reader, String environment) : SqlSessionFactory - SqlSessionFactoryBuilder
- build(InputStream inputStream, String environment, Properties properties) : SqlSessionFactory - SqlSessionFactoryBuilder
- build(Reader reader, String environment, Properties properties) : SqlSessionFactory - SqlSessionFactoryBuilder

使用：

```
// 指定全局配置文件
String resource = "mybatis-config.xml";
// 读取全局配置文件
InputStream inputStream = Resources.getResourceAsStream(resource);
// 构建sqlSessionFactory
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
// 打开sqlSession
SqlSession sqlSession = sqlSessionFactory.openSession();
```

7.6. Mappers

既然 MyBatis 的行为已经由上述元素配置完了，我们现在就要定义 SQL 映射语句了。但是首先我们需要告诉 MyBatis 到哪里去找到这些语句。Java 在自动查找这方面没有提供一个很好的方法，所以最佳的方式是告诉 MyBatis 到哪里去找映射文件。

7.6.1. 方式一：resource

在 mybatis-config.xml 引入项目目录下的映射文件：

```
<mappers>
  <mapper resource="MyMapper.xml" />
  <mapper resource="UserDaoMapper.xml"/>
  <mapper resource="UserMapper.xml"/>
</mappers>
```

缺点：每次都要在 mybatis-config.xml 中引入映射文件，麻烦

7.6.2. 方式二：file（不采用）

引入硬盘目录下的映射文件：

```
<!-- Using url fully qualified paths -->
<mappers>
  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>
```

缺点:

- 1、硬盘的位置可能随着项目的部署或迁移, 路径发生变化
- 2、每新增一个映射文件, 就要在全局配置文件中引入

7.6.3. 方式三: class

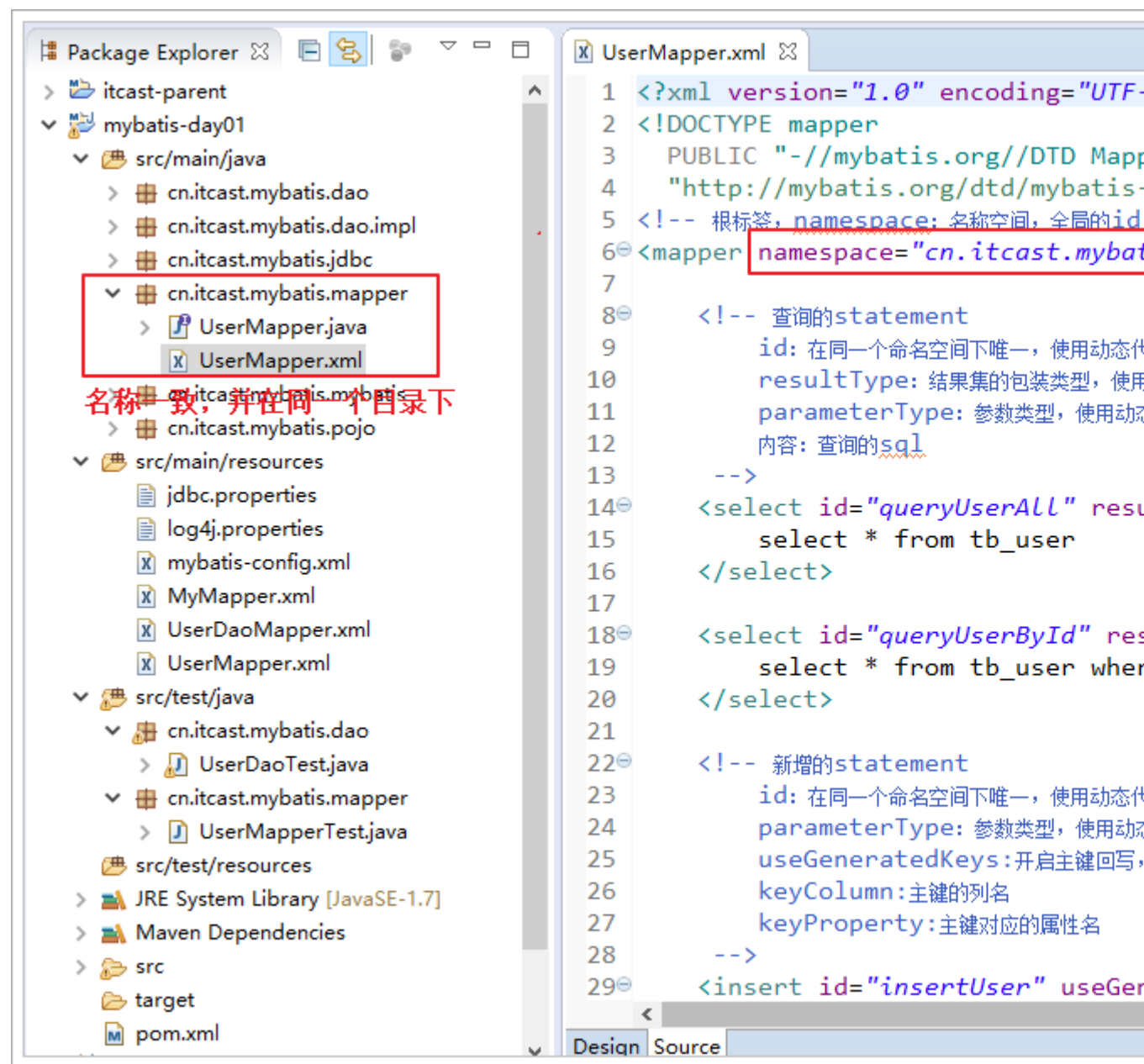
在 mybatis-config.xml 配置 mapper 接口的全路径:

```
<mappers>
  <!-- <mapper resource="MyMapper.xml" />
  <mapper resource="UserDaoMapper.xml"/>
  <mapper resource="UserMapper.xml"/> -->
  <!--
    1.映射文件必须和mapper接口在同一个目录下
    2.映射文件的namespace必须和mapper接口的全路径保持一致
    3.文件名要一致
  -->
  <mapper class="cn.itcast.mybatis.mapper.UserMapper" />
</mappers>
```

这种配置方式, 在全局配置文件中配置了 mapper 接口的全路径, 并没有配置 mapper 接口的映射文件的位置。如果要让 mybatis 找到对应的映射文件, 则必须满足一定的条件或规则:

- 1、映射文件和 mapper 接口在同一个目录下
- 2、文件名必须一致
- 3、映射文件的 namespace 必须和 mapper 接口的全路径保持一致

目录结构:



缺点:

- 1、java 文件和 xml 映射文件耦合
- 2、每新增一个映射文件，就要在全局配置文件中引入

7.6.4. 方式四：package

在 mybatis-config.xml 中，开启包扫描：

```
<mappers>
  <!-- <mapper resource="MyMapper.xml" />
  <mapper resource="UserDaoMapper.xml"/>
  <mapper resource="UserMapper.xml"/> -->
  <!--
    1.映射文件必须和mapper接口在同一个目录下
    2.映射文件的namespace必须和mapper接口的全路径保持一致
    3.文件名要一致
  -->
  <!-- <mapper class="cn.itcast.mybatis.mapper.UserMapper" /> -->
  <!-- 开启mapper接口的包扫描，基于class方式 -->
  <package name="cn.itcast.mybatis.mapper"/>
</mappers>
```

原理：扫描目标包目录下的 mapper 接口，并按照 class 的方式找到接口对应的映射文件。

缺点：

- 1、如果包的路径有很多
- 2、mapper.xml 和 mapper.java 没有分离。

8.Mapper XML 文件（映射文件）

- `cache` – 给定命名空间的缓存配置。
- `cache-ref` – 其他命名空间缓存配置的引用。
- `resultMap` – 是最复杂也是最强大的元素，用来描述如何从数据库结果集中来加载对象。
- `parameterMap` – 已废弃！老式风格的参数映射。内联参数是首选,这个元素可能在将来被移除，这里不会记
- `sql` – 可被其他语句引用的可重用语句块。
- `insert` – 映射插入语句
- `update` – 映射更新语句
- `delete` – 映射删除语句
- `select` – 映射查询语句

8.1. CRUD 标签

8.1.1. select

```
<!-- 查询的statement
    id: 在同一个命名空间下唯一，使用动态代理之后，要求和mapper接口的方法名保持一致，必须属性
    resultType: 结果集的安装类型，使用动态代理之后，要求和mapper接口方法的返回类型一致，resultMa
    parameterType: 参数类型，使用动态代理之后，要求和mapper接口方法的参数类型一致（不一定），可选属
    内容: 查询的sql
-->
<select id="queryUserAll" resultType="User">
    select * from tb_user
</select>
```

8.1.2. insert

<!-- 新增的statement

id: 在同一个命名空间下唯一，使用动态代理之后，要求和mapper接口的方法名保持一致，必须属性

parameterType: 参数类型，使用动态代理之后，要求和mapper接口方法的参数类型一致（不一定），可选

useGeneratedKeys: 开启主键回写，参数pojo类中，可选属性

keyColumn: 主键的列名，可选属性

keyProperty: 主键对应的属性名，可选属性

-->

<insert id="insertUser" useGeneratedKeys="true" keyColumn="id" keyProperty="id">

INSERT INTO tb_user (

id,

user_name,

password,

name,

age,

sex,

birthday,

created,

updated

)

VALUES

(

null,

#{userName},

#{password},

#{name},

#{age},

8.1.3. update

```
<!-- 更新的statement
    id: 在同一个命名空间下唯一，使用动态代理之后，要求和mapper接口的方法名保持一致，必须属性
    parameterType: 参数类型，使用动态代理之后，要求和mapper接口方法的参数类型一致（不一定），可选
    内容:更新的sql
-->
<update id="updateUser" parameterType="cn.itcast.mybatis.pojo.User">
    UPDATE tb_user
    SET
        user_name = #{userName},
        password = #{password},
        name = #{name},
        age = #{age},
        sex = #{sex},
        birthday = #{birthday},
        updated = NOW()
    WHERE
        (id = #{id});
</update>
```

8.1.4. delete

```
<!-- 删除的statement
    id: 在同一个命名空间下唯一，使用动态代理之后，要求和mapper接口的方法名保持一致
    parameterType: 参数类型，使用动态代理之后，要求和mapper接口方法的参数类型一致（不一定）
    内容:删除的sql
-->
<delete id="deleteUserById" parameterType="java.lang.Long">
    delete from tb_user where id=#{id}
</delete>
```

8.2. parameterType 传入参数

CRUD 标签都有一个属性 parameterType, statement 通过它指定接收的参数类型。

接收参数的方式有两种：

- 1、#{ }预编译
- 2、\${ }非预编译（直接的 sql 拼接，不能防止 sql 注入）

参数类型有三种：

- 1、基本数据类型
- 2、HashMap（使用方式和 pojo 类似）
- 3、Pojo 自定义包装类型

8.2.1. \${ }的用法

场景：数据库有两个一模一样的表。历史表，当前表

查询表中的信息，有时候从历史表中去查询数据，有时候需要去新的表去查询数据。希望使用 1 个方法来完成操作。

在 UserMapper 接口中，添加根据表名查询用户信息的方法：

```
/**
 * 根据表名查询用户信息
 * @param tableName
 * @return
 */
public List<User> queryUsersByTableName(String tableName);
```

在 UserMapper 映射文件中，添加方法对应的 statement：

```
<select id="queryUsersByTableName" resultType="User">
    select * from #{tableName}
</select>
```

输出 (报错):

```
org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException
### The error may exist in cn/itcast/mybatis/mapper/UserMapper.xml
### The error may involve defaultParameterMap
### The error occurred while setting parameters
### SQL: select * from ? sql语法错误
### Cause: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: You have an e
at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory
at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSe
at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSe
at org.apache.ibatis.binding.MapperMethod.executeForMany(MapperMethod.java:119)
at org.apache.ibatis.binding.MapperMethod.execute(MapperMethod.java:63)
at org.apache.ibatis.binding.MapperProxy.invoke(MapperProxy.java:52)
at com.sun.proxy.$Proxy3.queryUsersByTableName(Unknown Source)
at cn.itcast.mybatis.mapper.UserMapperTest.testQueryUsersByTableName(UserMapper
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl
at java.lang.reflect.Method.invoke(Method.java:606)
at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.
```

如果你要动态传入的字段名是表名, 并且 sql 执行是预编译的, 这显然是不行的, 所以你

必须改成非预编译的, 也就是这样:

```
<select id="queryUsersByTableName" resultType="User">
    select * from ${value}
</select>
```

注意:

使用`${}` 去接收参数信息, 在一个参数时, 默认情况下必须使用`$(value)`获取参数值,

而`#{}` 只是表示占位, 与参数的名字无关, 如果只有一个参数, 可以使用任意参数名接收参数值, 会自动对应。

但是这并不是一种稳妥的解决方案, 推荐使用`@Param` 注解指定参数名, 所以以上接口及

映射文件可以改成如下：



一个参数时，在使用`#{}传参`时，可以通过任意参数名接收参数；而`${}`，默认必须通过 `value` 来接收参数。

可以通过`@Param` 注解指定参数名

8.2.2. 多个参数

当 mapper 接口要传递多个参数时，有两种传递参数的方法：

- 1、默认规则获取参数{0,1,param1,param2}
- 2、使用@Param 注解指定参数名

案例：实现一个简单的用户登录，根据 username 和 password 验证用户信息

在 UserMapper 接口中，添加登陆方法：

```
/**
 * 根据用户名和密码登录
 * @param userName
 * @param password
 * @return
 */
public User login(String userName, String password);
```

在 UserMapper.xml 配置中，添加登陆方法对应的 Statement 配置：

```
<select id="login" resultType="User">
    select * from tb_user where user_name=#{userName} and password=#{password}
</select>
```

在 UserMapperTest 测试用例中，添加测试方法：

```
@Test
public void testLogin() {
    User user = this.userMapper.login("zhangsan", "123456");
    System.out.println(user);
}
```

运行报错：

```

org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause: org.apache.ibatis.binding.BindingException: Parameter 'userName' not found
### Cause: org.apache.ibatis.binding.BindingException: Parameter 'userName' not found
    at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:25)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:145)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:145)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:129)
    at org.apache.ibatis.binding.MapperMethod.execute(MapperMethod.java:68)
    at org.apache.ibatis.binding.MapperProxy.invoke(MapperProxy.java:52)
    at com.sun.proxy.$Proxy3.login(Unknown Source)
    at cn.itcast.mybatis.mapper.UserMapperTest.testLogin(UserMapperTest.java:124)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:62)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:59)
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:71)
    at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
    at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:26)
    at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:263)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:101)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:101)
    at org.junit.runners.ParentRunner$3.run(ParentRunner.java:231)

```

注意报错信息：没有找到参数 userName，可用的参数是:[1,0,param1,param2]，所以可有

以下解决方案：

解决方案一：

在映射文件里，通过#{0}，#{1}获取参数

```

<!--
    传入多个参数时，可以通过0,1,2,3...等方式获取参数
    0: 代表方法的第一个参数
    1: 代表方法的第二个参数
-->
<select id="login" resultType="User">
    select * from tb_user where user_name=#{0} and password=#{1}
</select>

```

解决方案二：

在映射文件里，通过 param1，param2 获取参数


```

<!--
    传入多个参数时
    还可以通过param1, param2,param3...等获取参数
    param1: 代表方法的第一个参数
    param2: 代表方法的第二个参数
-->
<select id="login" resultType="User">
    select * from tb_user where user_name=#{param1} and password=#{param2}
</select>

```

最终解决方案:

在接口方法中的参数前, 添加@Param 注解指定参数名

```

/**
 * 根据用户名和密码登录
 * @param userName
 * @param password
 * @return
 */
public User login(@Param("userName")String userName, @Param("password")String password)

```

```

<!--
    传入多个参数时
    最佳方案:
    通过在接口方法中使用@Param注解指定参数名
    这时, 在映射文件中的#{ }或者${ }就可以通过对应参数名获取参数
-->
<select id="login" resultType="User">
    select * from tb_user where user_name=#{userName} and password=#{password}
</select>

```

通常在方法的参数列表上加上一个注解@Param("xxxx") 表示参数的名字, 然后通过

\${ "xxxx" }或#{ "xxxx" }获取参数

注意:

单个参数时, #{ }与参数名无关的。

多个参数时，#{ }与参数名（@Param）有关。

什么时候需要加@Param 注解？什么时候不加？

单个参数不加，多个参数加

终极解决方案：都加。

8.2.3. HashMap

parameterType 有三种类型的输入参数：

- 1、基本数据类型
- 2、 hashMap
- 3、 pojo 包装类

前面已经使用了基本数据类型和 pojo 类型的参数，那么 hashMap 这种类型的参数怎么传递参数呢？

其实,它的使用方式和 pojo 有点类似,简单类型通过#{key}或者\${key},复杂类型通过\${key.属性名}或者#{key.属性名}

UserMapper 接口方法：

```
/**
 * 根据用户名和密码登录 (HashMap)
 * @param map
 * @return
 */
public User loginMap(Map<String, Object> map);
```

UserMapper 配置文件:

```
<!--
    获取HashMap中的参数
    简单对象: #{key}或者${key}
    pojo对象: #{key.属性名}或者${key.属性名}
-->
<select id="loginMap" resultType="User">
    select * from tb_user where user_name=#{userName} and password=#{pass
</select>
```

测试用例:

```
@Test
public void testLoginMap() {
    Map<String, Object> map = new HashMap<>();
    map.put("userName", "zhangsan");
    map.put("password", "123456");
    User user = this.userMapper.loginMap(map);
    System.out.println(user);
}
```

8.2.4. 面试题（#、\$区别）

在Mybatis的mapper中，参数传递有2种方式，一种是#{ }另一种是\${ }，两者有着很大的区别。

#{ } 实现的是sql语句的预处理参数，之后执行sql中用?号代替，使用时不需要关注数据类型，Mybatis自动实现数据类型的转换。并且可以防止SQL注入。

\${ } 实现是sql语句的直接拼接，不做数据类型转换，需要自行判断数据类型。不能防止SQL注入。

是不是\${ }就没用了呢？不是的，有些情况下就必须使用\${ }，举个例子：

在分表存储的情况下，我们从哪张表查询是不确定的，也就是说sql语句不能写死，表名是动态的，查询条件的固定的，这样：SELECT * FROM \${tableName} WHERE id = #{id}

总结：

#{ } 占位符，用于参数传递。

\${ }用于SQL拼接。

练习：根据用户名进行模糊查询

UserMapper 接口：

```
/**
 * 根据用户名模糊查询用户信息
 * @param userName
 * @return
 */
public List<User> queryUsersLikeUserName(@Param("userName")String userName)
```

UserMapper 映射文件：

```
<!-- <select id="queryUsersLikeUserName" resultType="User">
    select * from tb_user where user_name like '%${userName}%'
</select> -->

<select id="queryUsersLikeUserName" resultType="User">
    select * from tb_user where user_name like '%' #{userName} '%'
</select>
```

8.3. resultMap

ResultMap是Mybatis中最重要最强大的元素,使用**ResultMap**可以解决两大问题:

- POJO属性名和表结构字段名不一致的问题（有些情况下也不是标准的驼峰格式）
- 完成高级查询，比如说，一对一、一对多、多对多。

解决表字段名和属性名不一致的问题有两种方法：

- 1、如果是驼峰似的命名规则可以在Mybatis配置文件中设置 `<setting name="mapUnderscoreToCamelCase" value="true"/>`解决
- 2、使用**ResultMap**解决。

高级查询后面详细讲解。

8.3.1. 解决列名和属性名不一致

查询数据的时候，查不到 `userName` 的信息，原因：数据库的字段名是 `user_name`，而

POJO 中的属性名字是 `userName`

两端不一致，造成 mybatis 无法填充对应的字段信息。

解决方案 1: 在 sql 语句中使用别名

解决方案 2: 参考驼峰匹配 --- mybatis-config.xml 的时候

解决方案 3: resultMap 自定义映射

在 UserMapper.xml 中配置 resultMap

```
<!-- resultMap:自定义映射关系
    属性: type-结果集的封装类型, id-唯一标识, autoMapping-开启自动匹配, 如果开启了驼峰匹配, 就以
    id:指定主键映射的, 不要省。提高性能
    result: 其他的非主键普通字段
        子标签的属性: Column-表中的字段名, property-对应的java属性名
-->
<resultMap type="User" id="userMap" autoMapping="true">
    <id column="id" property="id"/>
    <!-- <result column="user_name" property="userName"/> -->
</resultMap>
```

在 UserMapper.xml 中使用 resultMap:

```
<!-- statement: 查询的statement
    id:在该映射文件下的唯一标识。在使用动态代理之后, 必须和接口的方法名一致。必须属性
    resultType:结果集的映射类型。在使用动态代理之后, 必须和接口方法的返回值类型一致。必须属性
    parameterType:参数类型。可省略
-->
<select id="queryUserById" resultMap="userMap">
    select * from tb_user where id = #{id}
</select>
```

8.3.2. resultMap 的自动映射

在上面讲到的 resultMap 中, 主键需要通过 id 子标签配置, 表字段和属性名不一致的普通字段需要通过 result 子标签配置。

那么，字段名称匹配的字段要不要配置那？

这个取决于 resultMap 中的 autoMapping 属性的值：

为 true 时：resultMap 中的没有配置的字段会自动对应。

为 false 时：只针对 resultMap 中已经配置的字段作映射。

并且 resultMap 会自动映射单表查询的结果集

8.4. sql 片段

在 java 代码中，为了提高代码的重用性，针对一些出现频率较高的代码，抽离出来一个共同的方法或者类

那么针对一些重复出现的 sql 片段，mybatis 有没有一个比较好的解决方案呢？

Mybatis 当然想到了这一点，它就是 sql 标签。

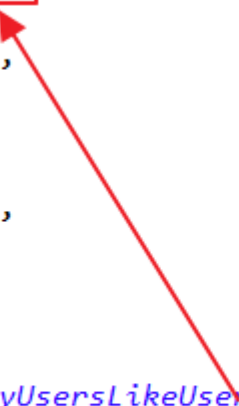
8.4.1. 用法一

sql 标签可以定义一个 sql 片段，在需要使用该 sql 片段的地方，通过 include 标签来使用，

如改造根据表名查询用户信息：

```
<sql id="commonSql">
    id,user_name
    password,
    name,
    age,
    sex,
    birthday,
    created,
    updated
</sql>

<select id="queryUsersLikeUserName" resultType="User">
    select <include refid="commonSql"></include> from tb_user where user
```



测试:

```
@Test
public void testQueryUsersLikeUserName(){
    List<User> users = this.userMapper.queryUsersLikeUserName("zhang");
    for (User user : users) {
        System.out.println(user);
    }
}
```

测试结果:

```
2017-07-26 10:18:08,622 [main]
[org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] PooledDataSource
forcefully closed/removed all connections.
2017-07-26 10:18:08,622 [main]
[org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] PooledDataSource
forcefully closed/removed all connections.
2017-07-26 10:18:08,622 [main]
[org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] PooledDataSource
forcefully closed/removed all connections.
2017-07-26 10:18:08,854 [main]
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-[DEBUG] Opening JDBC
Connection
2017-07-26 10:18:09,130 [main]
[org.apache.ibatis.datasource.pooled.PooledDataSource]-[DEBUG] Created connection
802102645.
2017-07-26 10:18:09,132 [main]
```

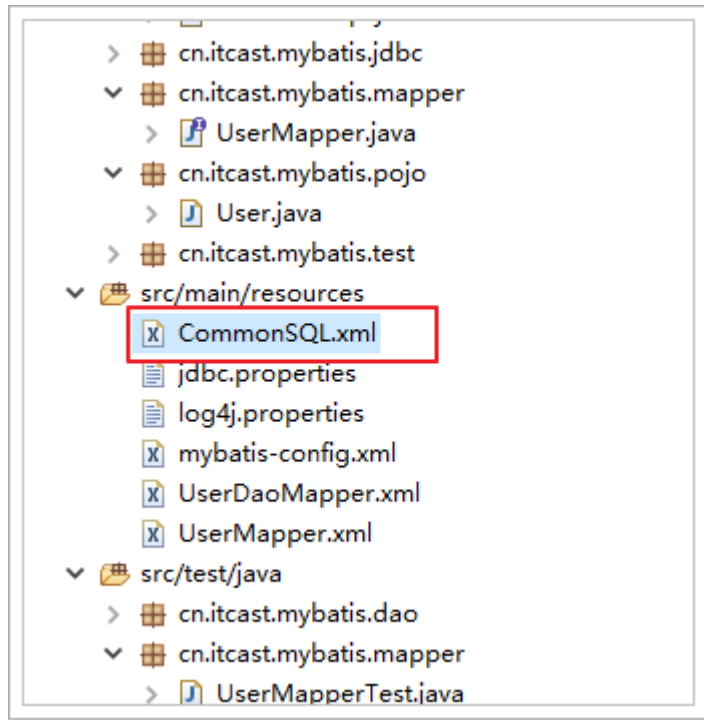


```
[cn.itcast.mybatis.mapper.UserMapper.queryUsersLikeUserName]-[DEBUG] ==>
Preparing: select id,user_name, password, name, age, sex, birthday, created,
updated from tb_user where user_name like '%' ? '%'
2017-07-26 10:18:09,185 [main]
[cn.itcast.mybatis.mapper.UserMapper.queryUsersLikeUserName]-[DEBUG] ==>
Parameters: zhang(String)
2017-07-26 10:18:09,213 [main]
[cn.itcast.mybatis.mapper.UserMapper.queryUsersLikeUserName]-[DEBUG] <==
Total: 2
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1,
birthday=Wed Aug 08 00:00:00 CST 1984, created=Fri Sep 19 16:56:04 CST 2014,
updated=Sun Sep 21 11:24:59 CST 2014]
User [id=4, userName=zhangwei, password=123456, name=张伟, age=20, sex=1,
birthday=Thu Sep 01 00:00:00 CDT 1988, created=Fri Sep 19 16:56:04 CST 2014,
updated=Fri Sep 19 16:56:04 CST 2014]
```

8.4.2. 用法二

很多时候同一个 sql 片段，可能在很多映射文件里都有使用，这就需要添加一个映射文件，用来统一定义 sql 片段。

如下，在 resource 目录下新增 CommonSQL.xml 文件：



内容如下：

```
<?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">

<!-- namespace(命名空间)：映射文件的唯一标识 -->
<mapper namespace="CommonSQL">

  <sql id="commonSql">
    id,user_name,
    password,
    name,
    age,
    sex,
    birthday,
    created,
    updated
  </sql>
</mapper>
```

定义好 sql 片段的映射文件之后，接下来就该使用它了，首先应该把该映射文件引入到 mybatis 的全局配置文件中（mybatis-config.xml）：

```

<mappers>
  <mapper resource="CommonSQL.xml"/>
  <mapper resource="UserMapper.xml" />
  <mapper resource="UserDaoMapper.xml" />
  <!--
    1.映射文件和mapper接口在同一个目录下
    2.命名一致
    3.命名空间必须是接口的全路径
  -->
  <!-- <mapper class="cn.itcast.mybatis.mapper.UserMapper" /> -->

```

最后在需要使用该 sql 片段的地方通过 include 标签的 refId 属性引用该 sql 片段: <include

refId=" 名称空间.sql 片段的 id" />

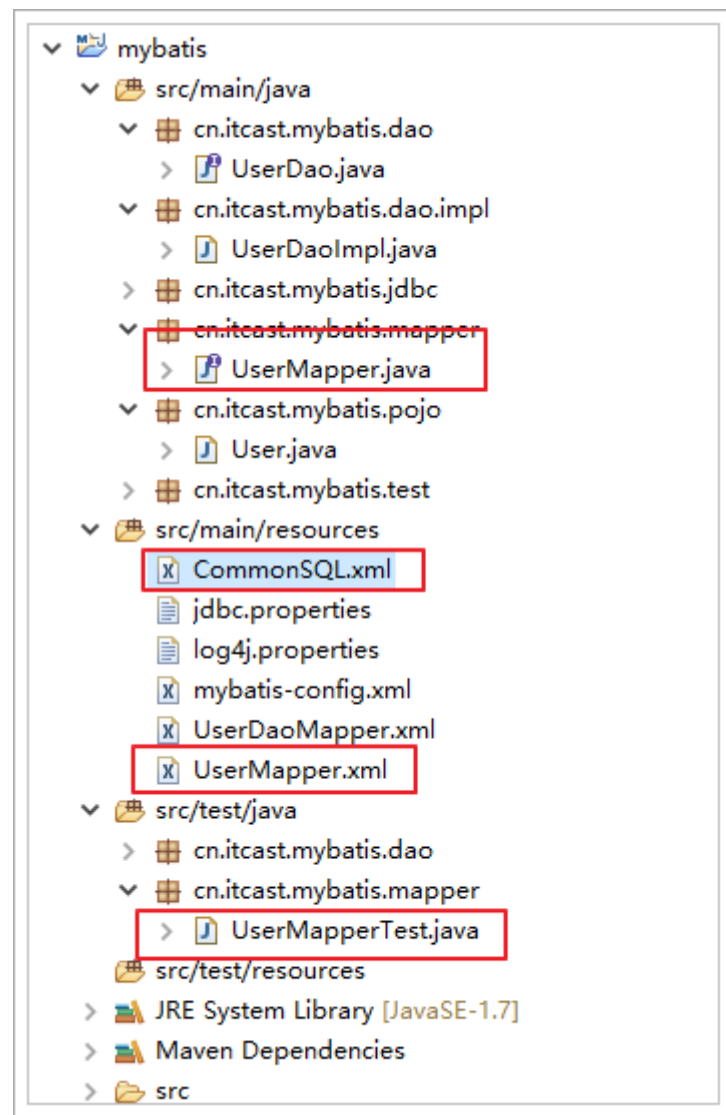
在 UserMapper.xml 的映射文件中, 进一步改造根据用户名查询用户信息

```

<select id="queryUsersLikeUserName" resultType="User">
  select <include refId="CommonSQL.commonSql"></include> from tb_user
</select>

```

8.5. 本章代码汇总



8.5.1. UserMapper.java

```
public interface UserMapper {  
  
    /**  
     * 根据用户名模糊查询用户信息  
     * @param userName  
     * @return  
     */  
    public List<User>  
    queryUsersLikeUserName(@Param("userName")String userName);  
}
```

```
/**
 * 登陆 (Map的方式)
 * @param map
 * @return
 */
public User loginByMap(Map<String,Object> map);

/**
 * 根据用户名和密码登陆
 * @param userName
 * @param password
 * @return
 */
public User login(@Param("userName")String userName,
@Param("password")String password);

/**
 * 根据表名查询用户信息
 * @param tableName
 * @return
 */
public List<User>
queryUsersByTableName(@Param("tableName")String tableName);

/**
 * 根据id查询用户信息
 * @param id
 * @return
 */
public User queryUserById(Long id);

/**
 * 查询所有用户信息
 * @return
 */
public List<User> queryUserAll();

/**
 * 新增用户信息
```

```

    * @param user
    */
    public void insertUser(User user);

    /**
     * 更新用户信息
     * @param user
     */
    public void updateUser(User user);

    /**
     * 根据id删除用户信息
     * @param id
     */
    public void deleteUserById(Long id);
}

```

8.5.2. 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">

<!-- namespace(命名空间): 映射文件的唯一标识 -->
<mapper namespace="cn.itcast.mybatis.mapper.UserMapper">

    <!-- <sql id="commonSql">
        id,user_name,
        password,
        name,
        age,
        sex,
        birthday,
        created,
        updated
    </sql> -->

    <select id="queryUsersLikeUserName" resultType="User">

```

```
select <include refid="CommonSQL.commonSql"></include>
from tb_user where user_name like '%' #{userName} '%'
</select>
```

```
<select id="loginByMap" resultType="User">
    select * from tb_user where user_name=#{userName} and
password=#{password}
</select>
```

```
<select id="login" resultType="User">
    select * from tb_user where user_name=#{userName} and
password=#{password}
</select>
```

```
<select id="queryUsersByTableName" resultType="User">
    select * from ${tableName}
</select>
```

<!-- resultMap:自定义映射关系

属性: type-结果集的封装类型, id-唯一标识, autoMapping-开启自动匹配, 如果开启了驼峰匹配, 就以驼峰匹配的形式进行匹配

id:指定主键映射的, 不要省。提高性能

result: 其他的非主键普通字段

子标签的属性: Column-表中的字段名, property-对应的java属性名

```
-->
<resultMap type="User" id="userMap" autoMapping="true">
    <id column="id" property="id"/>
    <!-- <result column="user_name" property="userName"/> -
->
</resultMap>
```

<!-- statement: 查询的statement

id:在该映射文件下的唯一标识。在使用动态代理之后, 必须和接口的方法名一致。必须属性

resultType:结果集的映射类型。在使用动态里之后, 必须和接口方法

的返回值类型一致。必须属性

parameterType:参数类型。可省略

-->

```
<select id="queryUserById" resultMap="userMap">
```

```
    select * from tb_user where id = #{id}
```

```
</select>
```

```
<select id="queryUserAll" resultType="User">
```

```
    select * from tb_user
```

```
</select>
```

<!-- 插入的statement

id:在该映射文件下的唯一标识。在使用动态代理之后, 必须和接口的

方法名一致。必须属性

parameterType:参数类型。可省略

useGeneratedKeys:开启主键回写,回写到参数的pojo对象里

keyColumn:主键列名

keyProperty: 主键对应的属性名

-->

```
<insert id="insertUser" useGeneratedKeys="true"
keyColumn="id" keyProperty="id" >
```

```
    INSERT INTO tb_user (
```

```
        user_name,
```

```
        password,
```

```
        name,
```

```
        age,
```

```
        sex,
```

```
        birthday,
```

```
        created,
```

```
        updated
```

```
    )
```

```
    VALUES
```

```
    (
```



```

        #{userName},
        #{password},
        #{name},
        #{age},
        #{sex},
        #{birthday},
        now(),
        now()
    );
</insert>

```

<!-- 更新的statement

id: 在该映射文件下的唯一标识。在使用动态代理之后，必须和接口的方法名一致。必须属性

parameterType:参数类型。可省略

```

-->
<update id="updateUser" >
    UPDATE tb_user
    SET
        user_name = #{userName},
        password = #{password},
        name = #{name},
        age = #{age},
        sex = #{sex},
        birthday = #{birthday},
        updated = now()
    WHERE
        (id = #{id});
</update>

```

<!-- 删除的statement

id: 在该映射文件下的唯一标识。在使用动态代理之后，必须和接口的方法名一致。必须属性

parameterType:参数类型。可省略

```

-->
<delete id="deleteUserById">

```

```
        delete from tb_user where id=#{id}
    </delete>

</mapper>
```

8.5.3. CommonSQL.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">

<!-- namespace(命名空间): 映射文件的唯一标识 -->
<mapper namespace="CommonSQL">

    <sql id="commonSql">
        id,user_name,
        password,
        name,
        age,
        sex,
        birthday,
        created,
        updated
    </sql>
</mapper>
```

8.5.4. Mybatis-config.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>

    <!-- 引入外部资源文件, resource: 相对路径, url: 绝对路径 -->
    <properties resource="jdbc.properties" />
    <settings>
```

```

    <!-- 行为参数, name:参数名, value: 参数值, 默认为false,
true: 开启驼峰匹配, 即从经典的数据库列名到经典的java属性名 -->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
<typeAliases>

    <!-- 类型别名: type: 类型的全路径, alias: 别名名称, 推荐和
class类名一致 -->

    <!-- <typeAlias type="cn.itcast.mybatis.pojo.User"
alias="User"/> -->

    <!-- 别名扫描, name:包的路径 -->

    <package name="cn.itcast.mybatis.pojo"/>
</typeAliases>

<!-- 环境: 说明可以配置多个, default:指定生效的环境 -->
<environments default="test">

    <!-- id:环境的唯一标识 -->
    <environment id="development">

        <!-- 事务管理器, type: 类型 -->
        <transactionManager type="JDBC" />

        <!-- 数据源: type-池类型的数据源 -->
        <dataSource type="POOLED">
            <property name="driver" value="${driver}" />
            <property name="url" value="${url}" />
            <property name="username" value="${username}" />
            <property name="password" value="${password}" />
        </dataSource>
    </environment>
    <environment id="test">

        <!-- 事务管理器, type: 类型 -->
        <transactionManager type="JDBC" />

        <!-- 数据源: type-池类型的数据源 -->
        <dataSource type="POOLED">
            <property name="driver" value="${driver}" />
            <property name="url" value="${url}" />

```

```

        <property name="username" value="${username}" />
        <property name="password" value="${password}" />
    </dataSource>
</environment>
</environments>

<!-- 映射文件 -->
<mappers>
    <mapper resource="CommonSQL.xml" />
    <mapper resource="UserMapper.xml" />
    <mapper resource="UserDaoMapper.xml" />
    <!--
        1.映射文件和mapper接口在同一个目录下
        2.命名一致
        3.命名空间必须是接口的全路径
    -->
    <!-- <mapper class="cn.itcast.mybatis.mapper.UserMapper"
/> -->
    <!-- <package name="cn.itcast.mybatis.mapper"/> -->
</mappers>
</configuration>

```

8.5.5. UserMapperTest

```

public class UserMapperTest {

    private UserMapper userMapper;

    @Before
    public void setUp() throws Exception {

        String resource = "mybatis-config.xml";

        // 读取配置文件

        InputStream inputStream =
Resources.getResourceAsStream(resource);

        // 构建sqlSessionFactory

        SqlSessionFactory sqlSessionFactory = new

```

```
SqlSessionFactoryBuilder().build(inputStream, "development");
```

```
// 获取sqlSession
```

```
SqlSession sqlSession =  
sqlSessionFactory.openSession(true);
```

```
    this.userMapper =  
sqlSession.getMapper(UserMapper.class);  
}
```

```
@Test
```

```
    public void testQueryUsersLikeUserName(){  
        List<User> users =  
this.userMapper.queryUsersLikeUserName("zhang");  
        for (User user : users) {  
            System.out.println(user);  
        }  
    }  
}
```

```
@Test
```

```
    public void testMap(){  
        Map<String, Object> map = new HashMap<>();  
        map.put("userName", "zhangsan");  
        map.put("password", "123456");  
        System.out.println(this.userMapper.loginByMap(map));  
    }  
}
```

```
@Test
```

```
    public void testLogin(){  
        System.out.println(this.userMapper.login("zhangsan",  
"123456"));  
    }  
}
```

```
@Test
```

```
    public void testQueryUsersByTableName(){  
        List<User> users =  
this.userMapper.queryUsersByTableName("tb_user");  
        for (User user : users) {  
            System.out.println(user);  
        }  
    }  
}
```

```
@Test
```

```
public void testQueryUserById() {
    System.out.println(this.userMapper.queryUserById(11));
}

@Test
public void testQueryUserAll() {
}

@Test
public void testInsertUser() {
    User user = new User();
    user.setAge(18);

    user.setName("柳岩");

    user.setPassword("123456");
    user.setUserName("yanyan1");
    user.setSex(3);
    user.setBirthday(new Date());
    this.userMapper.insertUser(user);
    System.out.println(user.getId());
}

@Test
public void testUpdateUser() {
}

@Test
public void testDeleteUserById() {
}

}
```

9.动态 sql

MyBatis 的一个强大的特性之一通常是它的动态 SQL 能力。提供了OGNL表达式动态生成SQL的功能。动态SQL有：

- 1、if
- 2、choose, when, otherwise
- 3、where, set
- 4、foreach

9.1. if

判断语句

案例：查询男性用户，如果输入了用户名，按用户名模糊查询

9.1.1. 定义接口

在 UserMapper 接口中定义方法：

```
/**
 * 查询男性用户，如果输入了用户名，按用户名模糊查询
 * @param userName
 * @return
 */
public List<User>
queryUserListLikeUserName(@Param("userName")String userName);
```

9.1.2. 编写 mapper.xml

在 UserMapper 映射文件中，定义接口方法对应的 Statement

```
<select id="queryUserListLikeUserName" resultType="User">
    select * from tb_user where sex=1

    <!-- if:判断

        test: OGNL表达式

    -->
    <if test="userName!=null and userName.trim()!=''">
        and user_name like '%' #{userName} '%'
    </if>
</select>
```

9.1.3. 测试

在 UserMapperTest 测试类中，添加测试用例

```
@Test
public void testQueryUserListLikeUserName(){
    List<User> users =
this.userMapper.queryUserListLikeUserName("zhang");
    for (User user : users) {
        System.out.println(user);
    }
}
```

9.2. choose when otherwise

查询男性用户，如果输入了用户名则按照用户名模糊查找，否则如果输入了年龄则按照年龄查找，否则查找用户名为 “zhangsan” 的用户。

9.2.1. 定义接口

在 UserMapper 接口中，定义接口方法：

```
/**
 * 查询男性用户，如果输入了用户名则按照用户名模糊查找，否则如果输入了年龄则按照年龄查找，否则查找用户名为“zhangsan”的用户。
 * @param userName
 * @param age
 * @return
 */
public List<User> queryUserListLikeUserNameOrAge(@Param("userName")String userName, @Param("age")Integer age);
```

9.2.2. 编写 mapper.xml

在 UserMapper.xml 中，定义接口方法对应的 Statement

```
<select id="queryUserListLikeUserNameOrAge"
resultType="User">
    select * from tb_user where sex=1

    <!-- choose:条件选择

    when: test-判断条件，一旦有一个when成立，后续的when都不再
执行

    otherwise: 所有的when都不成立时，才会执行
-->
    <choose>
        <when test="userName!=null and
userName.trim()!=''">and user_name like '%' #{userName}
'%'</when>
        <when test="age != null">and age = #{age}</when>
        <otherwise>and user_name = 'zhangsan' </otherwise>
    </choose>
</select>
```

9.2.3. 测试

在 UserMapperTest 测试类中，添加测试用例

```
@Test
public void testQueryUserListLikeUserNameOrAge(){
    List<User> users =
this.userMapper.queryUserListLikeUserNameOrAge(null, null);
    for (User user : users) {
        System.out.println(user);
    }
}
```

9.3. where

案例：查询所有用户，如果输入了用户名按照用户名进行模糊查询，如果输入年龄，按照年龄进行查询，如果两者都输入，两个条件都要成立。

9.3.1. 定义接口

在 UserMapper 接口中，定义接口方法：

```
/**
 * 查询所有用户，如果输入了用户名按照用户名进行模糊查询，如果输入
年龄，按照年龄进行查询，如果两者都输入，两个条件都要成立。
 * @param userName
 * @param age
 * @return
 */
public List<User>
queryUserListLikeUserNameAndAge(@Param("userName")String
userName, @Param("age")Integer age);
```

9.3.2. 编写 mapper.xml

在 UserMapper.xml 中，定义接口方法对应的 Statement

```
<select id="queryUserListLikeUserNameAndAge"
resultType="User">
    select * from tb_user
    <!--
        自动添加where关键字

        有一定的纠错功能：去掉sql语句块之前多余的一个and|or

        通常结合if或者choose使用
    -->
    <where>
        <if test="userName!=null and
userName.trim()!=''">user_name like '%' #{userName} '%"</if>
        <if test="age!=null">and age = #{age}</if>
    </where>
</select>
```

9.3.3. 测试

在 UserMapperTest 测试类中，添加测试用例

```
@Test
public void testQueryUserListLikeUserNameAndAge(){
    List<User> users =
this.userMapper.queryUserListLikeUserNameAndAge(null, 30);
    for (User user : users) {
        System.out.println(user);
    }
}
```

9.4. set

案例：修改用户信息，如果参数 user 中的某个属性为 null，则不修改。

9.4.1. 定义接口

在 UserMapper 接口中，定义接口方法：

```
/**
 * 修改用户信息，如果参数user中的某个属性为null，则不修改。
 * @param user
 */
public void updateUserSelective(User user);
```

9.4.2. 编写 mapper.xml

在 UserMapper.xml 中，定义接口方法对应的 Statement

```
<update id="updateUserSelective" >
    UPDATE tb_user
    <!--
        set自动添加set关键字

        也有一定的纠错功能：自动去掉sql语句块之后多余的一个逗号
    -->
    <set>
        <if test="userName!=null and
userName.trim()!=''">user_name = #{userName},</if>
        <if test="password!=null and
password.trim()!=''">password = #{password},</if>
        <if test="name!=null and name.trim()!=''">name =
#{name},</if>
        <if test="age!=null">age = #{age},</if>
        <if test="sex!=null">sex = #{sex},</if>
        updated = now(),
    </set>
    WHERE
```

```
        (id = #{id});  
</update>
```

9.4.3. 测试

在 UserMapperTest 测试类中，添加测试用例

```
@Test  
public void testUpdateUserSelective(){  
    User user = new User();  
    user.setAge(18);  
  
    user.setName("柳岩");  
  
    user.setPassword("123456");  
    user.setUserName("yanyan2");  
//    user.setSex(3);  
    user.setBirthday(new Date());  
    user.setId(121);  
    this.userMapper.updateUserSelective(user);  
}
```

9.5. foreach

案例：按照多个 id 查询用户信息

9.5.1. 接口

在 UserMapper 接口中，定义接口方法：

```
/**  
 * 根据多个id查询用户信息  
 * @param ids  
 * @return  
 */  
public List<User> queryUserListByIds(@Param("ids")Long[]  
ids);
```

9.5.2. 配置

在 UserMapper.xml 中，定义接口方法对应的 Statement

```
<select id="queryUserListByIds" resultType="User">
    select * from tb_user where id in
    <!--
        foreach:遍历集合

        collection: 接收的集合参数

        item: 遍历的集合中的一个元素

        separator:分隔符

        open:以什么开始

        close: 以什么结束

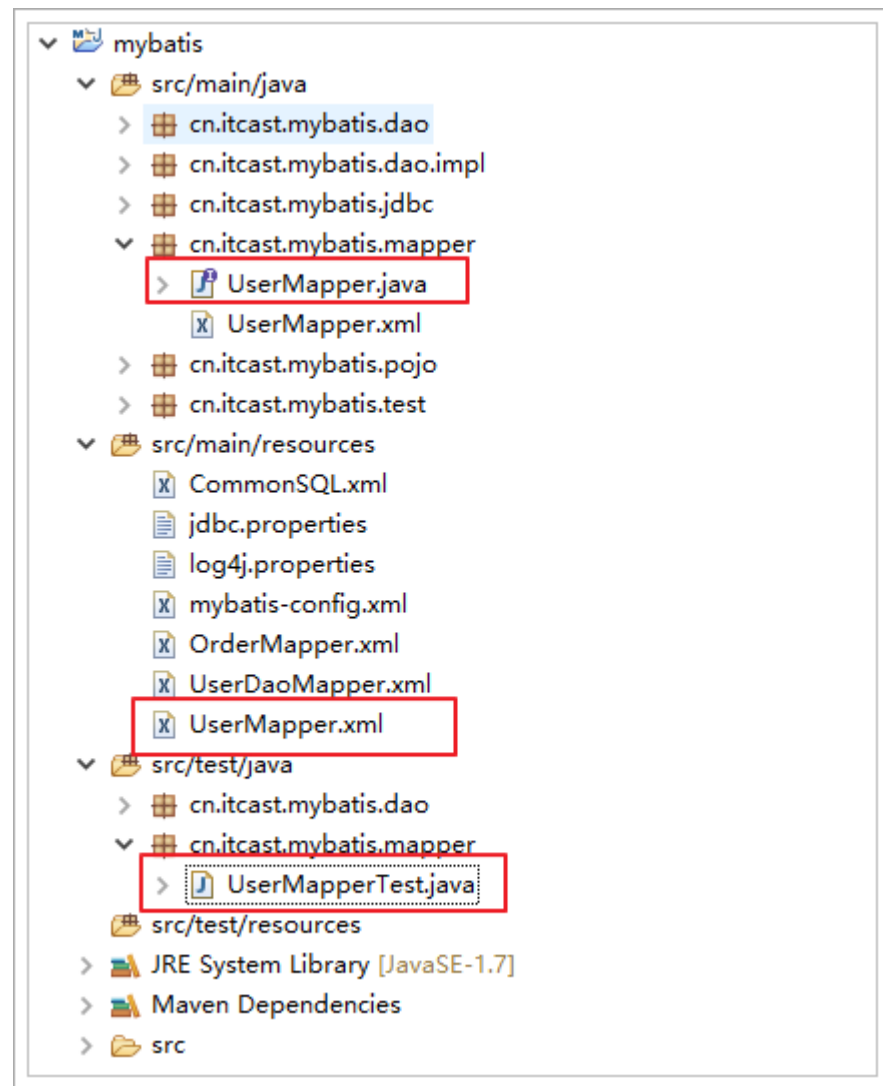
    -->
    <foreach collection="ids" item="id" separator=","
open="(" close=")">
        #{id}
    </foreach>
</select>
```

9.5.3. 测试

在 UserMapperTest 测试类中，添加测试用例

```
@Test
public void testQueryUserListByIds(){
    List<User> users =
this.userMapper.queryUserListByIds(new Long[]{11,21,31,41});
    for (User user : users) {
        System.out.println(user);
    }
}
```

9.6. 本章代码汇总



9.6.1. UserMapper.xml

```
<select id="queryUserListLikeUserName" resultType="User">
    select * from tb_user where sex < 2
    <!-- if:判断
        test: OGNL表达式或者简单的java代码
    -->
    <if test="userName!=null and !''.equals(userName)">
```

```

        and user_name like '%' #{userName} '%'
    </if>
</select>

<select id="queryUserListLikeUserNameOrAge"
resultType="User">
    select * from tb_user where sex=1

    <!-- choose:条件选择

        when: test-条件, 使用方式参考if的test属性, 一旦有一个成
立, 后续的when都不再执行

        otherwise: 所有的when都不成立时, 才会执行
-->
    <choose>
        <when test="userName!=null and
userName.trim()!=''">and user_name like '%' #{userName}
        '%'</when>
        <when test="age != null">and age = #{age}</when>
        <otherwise>and user_name = 'zhangsan' </otherwise>
    </choose>
</select>

<select id="queryUserListLikeUserNameAndAge"
resultType="User">
    select * from tb_user
    <!--
        自动添加where关键字

        有一定的纠错功能: 去掉sql语句块之前多余的一个and|or

        通常结合if或者choose使用
-->
    <where>
        <if test="userName!=null and
userName.trim()!=''">user_name like '%' #{userName} '%'</if>
        <if test="age!=null">and age = #{age}</if>
    </where>
</select>

<update id="updateUserSelective" >

```



```

UPDATE tb_user
<!--

    set自动添加set关键字

    也有一定的纠错功能：自动去掉sql语句块之后多余的一个逗号

-->
<set>
    <if test="userName!=null and
userName.trim()!=''">user_name = #{userName},</if>
    <if test="password!=null and
password.trim()!=''">password = #{password},</if>
    <if test="name!=null and name.trim()!=''">name =
#{name},</if>
    <if test="age!=null">age = #{age},</if>
    <if test="sex!=null">sex = #{sex}</if>
</set>
WHERE
    (id = #{id});
</update>

<select id="queryUserListByIds" resultType="User">
    select * from tb_user where id in
<!--

    foreach:遍历集合

    collection: 接收的集合参数

    item: 遍历的集合中的一个元素

    separator:分隔符

    open:以什么开始

    close: 以什么结束

-->
    <foreach collection="ids" item="id" separator=","
open="(" close=")">
        #{id}
    </foreach>
</select>

```

9.6.2. UserMapper.java

```
/**
 * 查询男性用户，如果输入了用户名，按用户名模糊查询
 * @param userName
 * @return
 */
public List<User>
queryUserListLikeUserName(@Param("userName")String userName);

/**
 * 查询男性用户，如果输入了姓名则按照姓名模糊查找，否则如果输入了
年龄则按照年龄查找，否则查找姓名为“张三”的用户。
 * @param userName
 * @param age
 * @return
 */
public List<User>
queryUserListLikeUserNameOrAge(@Param("userName")String
userName, @Param("age")Integer age);

/**
 * 查询所有用户，如果输入了姓名按照姓名进行模糊查询，如果输入年
龄，按照年龄进行查询，如果两者都输入，两个条件都要成立。
 * @param userName
 * @param age
 * @return
 */
public List<User>
queryUserListLikeUserNameAndAge(@Param("userName")String
userName, @Param("age")Integer age);

/**
 * 修改用户信息，如果参数user中的某个属性为null，则不修改。
 * @param user
 */
public void updateUserSelective(User user);
```

```

/**
 * 根据多个id查询用户信息
 * @param ids
 * @return
 */
public List<User> queryUserListByIds(@Param("ids")Long[]
ids);

```

9.6.3. UserMapperTest.java

```

@Test
public void testQueryUserListByIds(){
    List<User> users =
this.userMapper.queryUserListByIds(new Long[]{11,21,31,41});
    for (User user : users) {
        System.out.println(user);
    }
}

@Test
public void testUpdateUserSelective(){
    User user = new User();
    user.setAge(18);

    user.setName("柳岩");

    user.setPassword("123456");
    user.setUserName("yanyan2");
//    user.setSex(3);
    user.setBirthday(new Date());
    user.setId(121);
    this.userMapper.updateUserSelective(user);
}

@Test
public void testQueryUserListLikeUserNameAndAge(){
    List<User> users =
this.userMapper.queryUserListLikeUserNameAndAge(null, 30);
    for (User user : users) {
        System.out.println(user);
    }
}

```

```

    }

    @Test
    public void testQueryUserListLikeUserNameOrAge(){
        List<User> users =
this.userMapper.queryUserListLikeUserNameOrAge(null, null);
        for (User user : users) {
            System.out.println(user);
        }
    }

    @Test
    public void testQueryUserListLikeUserName(){
        List<User> users =
this.userMapper.queryUserListLikeUserName("zhang");
        for (User user : users) {
            System.out.println(user);
        }
    }
}

```

10. 缓存(了解)

执行相同的 sql 语句和参数, mybatis 不重复发送执行 sql, 而是从缓存中命中返回。

10.1. 一级缓存

在 mybatis 中, 一级缓存默认是开启的, 并且一直无法关闭, 作用域: 在同一个 sqlSession

下

在 UserMapperTest 中添加测试一级缓存的方法:

```

@Test
public void testCache(){
    User user1 = this.userMapper.queryUserById(11);
    System.out.println(user1);

    System.out.println("=====第二次查询
=====");
    User user2 = this.userMapper.queryUserById(11);
}

```

```

        System.out.println(user2);
    }

```

由于一级缓存的存在，此时在 log 日志中，应该只会第一次查询是执行 sql 语句，第二次查询时直接从缓存中命中，即不再执行 sql 语句。

结果：

```

2017-04-12 21:35:50,728 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUserById(11)]
2017-04-12 21:35:50,776 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUserById(11)]
2017-04-12 21:35:50,798 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUserById(11)]
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, birthday=2017-04-12 21:35:50,728]
=====第二次查询=====
2017-04-12 21:35:50,801 [main] [cn.itcast.mybatis.mapper.UserMapper]-[DEBUG]
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, birthday=2017-04-12 21:35:50,801]

```

使用：`sqlSession.clearCache()`可以强制清除缓存

在测试方法中清空一级缓存：

```

@Test
public void testCache(){
    User user1 = this.userMapper.queryUserById(11);
    System.out.println(user1);
    this.sqlSession.clearCache();

    System.out.println("=====第二次查询=====");
    User user2 = this.userMapper.queryUserById(11);
    System.out.println(user2);
}

```

在执行第二次查询之前清空缓存，再去执行查询。这时无法从缓存中命中，便会去执行 sql 从数据库中查询。

```

2017-04-12 21:33:56,541 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 21:33:56,584 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 21:33:56,608 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, b
=====第二次查询=====
2017-04-12 21:33:56,611 [main] [cn.itcast.mybatis.mapper.UserMapper]-[DEBU
2017-04-12 21:33:56,612 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 21:33:56,612 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 21:33:56,614 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, b

```

执行 update、insert、delete 的时候，会清空缓存

在测试方法中添加更新的操作：

```

@Test
public void testCache(){
    User user1 = this.userMapper.queryUserById(11);
    System.out.println(user1);
    // this.sqlSession.clearCache();

    System.out.println("=====更新
=====");
    User user = new User();
    user.setAge(18);

    user.setName("柳岩");

    user.setPassword("123456");
    user.setUserName("yanyan2");
    // user.setSex(3);
    user.setBirthday(new Date());
    user.setId(121);
    this.userMapper.updateUser (user);

    System.out.println("=====第二次查询
=====");
    User user2 = this.userMapper.queryUserById(11);
    System.out.println(user2);
}

```

由于 insert、update、delete 会清空缓存，所以第二次查询时，依然会输出 sql 语句，即

从数据库中查询。

```
2017-04-12 21:39:51,517 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 21:39:51,569 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 21:39:51,594 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, b
=====更新=====
2017-04-12 21:39:51,635 [main] [cn.itcast.mybatis.mapper.UserMapper.update
2017-04-12 21:39:51,636 [main] [cn.itcast.mybatis.mapper.UserMapper.update
2017-04-12 21:39:51,636 [main] [cn.itcast.mybatis.mapper.UserMapper.update
=====第二次查询=====
2017-04-12 21:39:51,637 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 21:39:51,637 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
2017-04-12 21:39:51,639 [main] [cn.itcast.mybatis.mapper.UserMapper.queryl
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, b
```

10.2. 二级缓存

mybatis 的二级缓存的作用域：

- 1、 同一个 mapper 的 namespace, 同一个 namespace 中查询 sql 可以从缓存中命中。
- 2、 跨 sqlSession, 不同的 SqlSession 可以从二级缓存中命中

怎么开启二级缓存：

- 1、 在映射文件中, 添加<cache />标签
- 2、 在全局配置文件中, 设置 cacheEnabled 参数, 默认已开启。

注意：

由于缓存数据是在 sqlSession 调用 close 方法时, 放入二级缓存的, 所以第一个 sqlSession 必须先关闭

二级缓存的对象必须序列化, 例如: User 对象必须实现 Serializable 接口。

开启二级缓存，在映射文件(UserMapper.xml)中添加<cache />：

```
<!-- mybatis 二级缓存 -->
<mapper namespace="cn.itcast.mybatis.mapper.UserMapper">

    <cache />

    <select id="queryUserListLikeUserName" resultType="User">
        select * from tb_user where sex=1
    <!-- if:判断 -->
</mapper>
```

在 UserMapperTest 中添加二级缓存的测试方法：

```
@Test
public void testCache2(){
    User user1 = this.userMapper.queryUserById(11);
    System.out.println(user1);

    // 注意：关闭sqlSession
    sqlSession.close();

    System.out.println("=====第二次查询
=====");

    // 重新打开一个sqlSession会话
    SqlSession sqlSession2 =
this.sqlSessionFactory.openSession();

    // 通过sqlSession2重新实例化UserMapper
    this.userMapper =
sqlSession2.getMapper(UserMapper.class);
    User user2 = this.userMapper.queryUserById(11);
    System.out.println(user2);
}
```

运行测试用例：


```

1 x
org.apache.ibatis.cache.CacheException: Error serializing object. Cause: java.io.N
    at org.apache.ibatis.cache.decorators.SerializedCache.serialize(SerializedCache
    at org.apache.ibatis.cache.decorators.SerializedCache.putObject(SerializedCache
    at org.apache.ibatis.cache.decorators.LoggingCache.putObject(LoggingCache.java:
    at org.apache.ibatis.cache.decorators.SynchronizedCache.putObject(SynchronizedC
    at org.apache.ibatis.cache.decorators.TransactionalCache$AddEntry.commit(Transa
    at org.apache.ibatis.cache.decorators.TransactionalCache.commit(TransactionalCa
    at org.apache.ibatis.cache.TransactionalCacheManager.commit(TransactionalCacheM
    at org.apache.ibatis.executor.CachingExecutor.close(CachingExecutor.java:58)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.close(DefaultSqlSession
    at cn.itcast.mybatis.mapper.UserMapperTest.testCache2(UserMapperTest.java:64)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl
    at java.lang.reflect.Method.invoke(Method.java:606)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.i

```

说明二级缓存，必须序列化，使 User 类实现序列化接口：

```

public class User implements Serializable{
    private static final long serialVersionUID = -7125073066608012284L;
    private Long id;

```

给 User 对象实现序列化接口后，重新运行测试用例，日志：

```

2017-07-29 21:06:49,041 [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction
2017-07-29 21:06:49,869 [main] [org.apache.ibatis.datasource.pooled.PooledData
2017-07-29 21:06:49,871 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUser
2017-07-29 21:06:49,910 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUser
2017-07-29 21:06:49,971 [main] [cn.itcast.mybatis.mapper.UserMapper.queryUser
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, b
2017-07-29 21:06:49,983 [main] [org.apache.ibatis.transaction.jdbc.JdbcTransaction
2017-07-29 21:06:49,983 [main] [org.apache.ibatis.datasource.pooled.PooledData
=====第二次查询=====
2017-07-29 21:06:49,984 [main] [cn.itcast.mybatis.mapper.UserMapper]-[DEBUG]
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, b

```

执行 update、insert、delete 的时候，会清空缓存

```

@Test
public void testCache2(){
    User user1 = this.userMapper.queryUserById(11);
    System.out.println(user1);

```

```

        // 注意: 关闭sqlSession
        sqlSession.close();

        System.out.println("=====
");

        // 重新打开一个sqlSession会话
        SqlSession sqlSession2 =
this.sqlSessionFactory.openSession();

        // 通过sqlSession2重新实例化UserMapper

        this.userMapper =
sqlSession2.getMapper(UserMapper.class);
        User user = new User();
        user.setAge(18);

        user.setName("柳岩");

        user.setPassword("123456");
        user.setUserName("yanyan2");
        user.setBirthday(new Date());
        user.setId(121);
        this.userMapper.updateUserSelective(user);

        System.out.println("=====
");
        User user2 = this.userMapper.queryUserById(11);
        System.out.println(user2);
    }

```

关闭二级缓存:

不开启, 或者在全局的 mybatis-config.xml 中去关闭二级缓存

设置参数	描述	有效值	默认值
cacheEnabled	该配置影响的所有映射器中配置的缓存的全局开关。	true false	true

在 mybatis-config.xml 配置中:

```

<settings>
    <!-- 行为参数, name:参数名, value: 参数值, 默认为false,
true: 开启驼峰匹配, 即从经典的数据库列名到经典的java属性名 -->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
    <!-- 关闭二级缓存, 默认是开启, false: 关闭 -->
    <setting name="cacheEnabled" value="false"/>
</settings>

```

```

<cache
    eviction="FIFO"
    flushInterval="60000"
    size="512"
    readOnly="true"/>

```

这个更高级的配置创建了一个 FIFO 缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象的调用者之间修改它们会导致冲突。

可用的回收策略有:

- **LRU** – 最近最少使用的:移除最长时间不被使用的对象。
- **FIFO** – 先进先出:按对象进入缓存的顺序来移除它们。
- **SOFT** – 软引用:移除基于垃圾回收器状态和软引用规则的对象。
- **WEAK** – 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

默认的是 LRU。

flushInterval(刷新间隔)可以被设置为任意的正整数,而且它们代表一个合理的毫秒 形式的时间段。默认情况是 调用语句时刷新。

size(引用数目)可以被设置为任意正整数,要记住你缓存的对象数目和你运行环境的 可用内存资源数目。默认

readOnly(只读)属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此可以获得性能优势。可读写的缓存 会返回缓存对象的拷贝(通过序列化)。这会慢一些,但是安全,因此默认是 false

11. 高级查询(OrderMapper.xml)

11.1. 表关系说明

Mybatis作为一个ORM框架，也对SQL的高级查询做了支持，下面我们学习Mybatis下的一对一、一对多、多对多的查询。

案例说明：

此案例的业务关系是用户、订单、订单详情、商品之间的关系，其中，
一个订单只能属于一个人。
一个订单可以有多个订单详情。
一个订单详情中包含一个商品信息。

它们的关系是：

订单和人是一**对一**的关系。
订单和订单详情是一**对多**的关系。
订单和商品是多**对多**的关系。

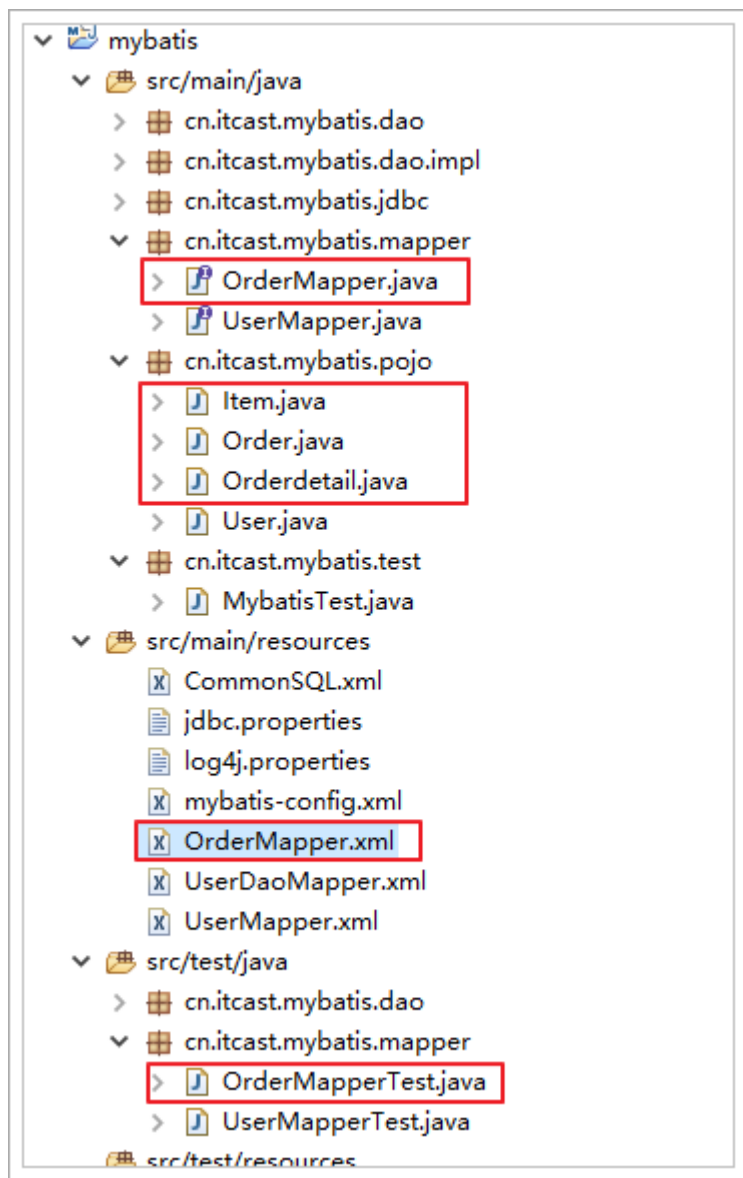
需求说明：

一对一查询：查询订单，并且查询出下单人的信息。

一对多查询：查询订单，查询出下单人信息并且查询出订单详情。

多对多查询：查询订单，查询出下单人信息并且查询出订单详情中的商品数据。

目录结构：

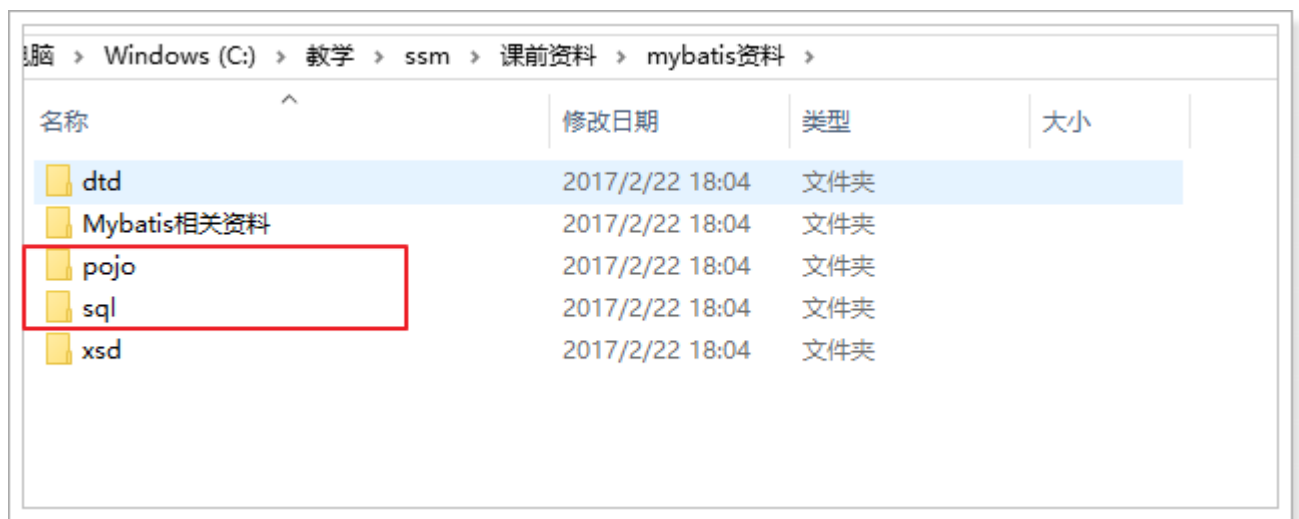


创建 OrderMapper.java 接口，参考 UserMapper.java。

添加 OrderMapper.xml 映射文件，参考 UserMapper.xml。

创建 OrderMapperTest 测试用例，参考 UserMapperTest。

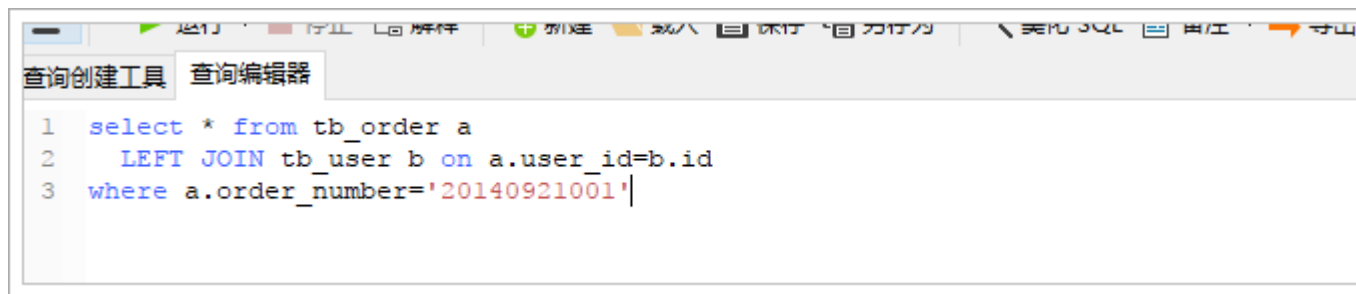
Pojo 参考课前资料：



11.2. 一对一查询

需求：查询出订单信息，并查询出下单人信息

Sql:



11.2.1. 扩展 Order 类的实现方式（了解）

思路：

创建新的 pojo 对象 OrderUser 继承 Order 对象，这样 OrderUser 就从 Order 中继承了订单信息。

再把 User 对象中的属性 copy 到 OrderUser 对象中，这样 OrderUser 就有了 User 中的用户信息。

把 sql 语句的结果集封装成 OrderUser 对象，那么结果中就既包含了订单信息又包含了下单人信息。

新建 OrderUser 实体类继承 Order，在把 User 中的属性 copy 到 OrderUser 中，并添加 get、set 方法：

```
public class OrderUser extends Order{
    // 用户名
    private String userName;

    // 密码
    private String password;

    // 姓名
    private String name;

    // 年龄
    private Integer age;
```

在 OrderMapper 接口中，添加查询方法：

```
public OrderUser queryOrderUserByOrderNumber(@Param("number") String number)
```

在 OrderMapper.xml 中，配置对应的 Statement，并把结果集封装成 OrderUser 对象：

```
<mapper namespace="cn.itcast.mybatis.mapper.OrderMapper">
    <select id="queryOrderUserByOrderNumber" resultType="OrderUser">
        select * from tb_order o left join tb_user u on o.user_id=u.id where
    </select>
</mapper>
```

在 OrderMapperTest 中添加测试方法：

```
@Test
public void testQueryOrderUserByOrderNumber() {
    OrderUser orderUser=orderMapper.queryOrderUserByOrderNumber("20140921001");
    System.out.println(orderUser);
}
```

11.2.2. 添加 User 属性的实现方式

思路：

从订单的角度来看，订单和用户的关系是一对一的关系，并且通过 tb_order 表中的 id 进行关联。

用面向对象的思想实现，就是在 Order 对象中添加 user 属性。

在 Order 对象中添加 User 属性，并添加 user 的 get、set 方法：

```
public class Order {  
  
    private Integer id;  
  
    private Long userId;  
  
    private String orderNumber;  
  
    private User user;  
  
    public User getUser() {  
        return user;  
    }  
  
    public void setUser(User user) {  
        this.user = user;  
    }  
  
    public Integer getId() {  
        return id;  
    }  
}
```

在 OrderMapper 接口中，添加接口方法：

```
/**  
 * 根据订单号查询订单信息，并且查询出下单人信息  
 * @param number  
 * @return  
 */
```



```
public Order queryOrderWithUser(@Param("number")String number);
```

使用 `resultType` 不能完成 `user` 对象的自动映射，需要手动完成结果集映射，即使用 `resultMap` 标签自定义映射。

在 `OrderMapper.xml` 中配置，结果集的映射，这时必须使用 `resultMap`：

```
<resultMap type="Order" id="orderUserMap"
autoMapping="true">
    <id column="id" property="id"/>
    <!--
        association:一对一的映射
        property: java的属性名
        javaType: 属性名对应的java类型
        autoMapping:开启自动映射
        子标签: 参照resultMap
    -->
    <association property="user" javaType="User"
autoMapping="true">
        <id column="user_id" property="id"/>
    </association>
</resultMap>

<!-- resultType不能完成user信息的映射，必须使用resultMap，
resultMap的值对应resultMap标签的id，resultMap和resultType必须二选一 -->

<select id="queryOrderWithUser" resultMap="orderUserMap">
    select * from tb_order a
        LEFT JOIN tb_user b on a.user_id=b.id
    where a.order_number = #{number}
</select>
```

在 OrderMapperTest 添加测试：

```
public class OrderMapperTest {

    private OrderMapper orderMapper;

    @Before
    public void setUp() throws Exception {
        String resource = "mybatis-config.xml";

        // 读取配置文件
        InputStream inputStream =
Resources.getResourceAsStream(resource);

        // 构建sqlSessionFactory
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream, "development");

        // 获取sqlSession
        SqlSession sqlSession =
sqlSessionFactory.openSession(true);

        this.orderMapper =
sqlSession.getMapper(OrderMapper.class);
    }

    @Test
    public void testQueryOrderWithUser() {

        System.out.println(this.orderMapper.queryOrderWithUser("201
40921001"));
    }
}
```

11.3. 一对多查询

一对多查询： 查询订单，查询出下单人信息并且查询出订单详情。

思路:

订单 : 订单详情 = 1:n (体现在 pojo 对象中, 就是在 Order 对象中添加 OrderDetail

对象的集合)

SQL:

```
1
2 select * from tb_order o
3   left join tb_user u on o.user_id=u.id
4   left join tb_orderdetail od on o.id=od.order_id
5 where o.order_number='20140921001'
6
```

信息	结果1	概况	状态							
id	user_id	order_number	id1	user_name	password	name	age	sex	birthday	
1	1	20140921001	1	zhangsan	123456	张三	19	1	1984-08-08	
1	1	20140921001	1	zhangsan	123456	张三	19	1	1984-08-08	

在 Order 类添加 List<OrderDetail>属性, 并添加 get、set 方法:

```
public class Order {

    private Integer id;

    private Long userId;

    private String orderNumber;

    private User user;

    private List<Orderdetail> detailList;

    public List<Orderdetail> getDetailList() {
        return detailList;
    }

    public void setDetailList(List<Orderdetail> detailList) {
        this.detailList = detailList;
    }

}
```

OrderMapper 接口:

```

/**
 * 查询订单，查询出下单人信息并且查询出订单详情。
 * @param number
 * @return
 */
public Order queryOrderWithUserDetail(@Param("number")String
number);

```

OrderMapper 映射:

```

<resultMap type="Order" id="orderUserDetailMap"
autoMapping="true">
    <id column="id" property="id"/>
    <association property="user" javaType="User"
autoMapping="true">
        <id column="user_id" property="id"/>
    </association>
<!--
    collection:一对多的查询

    property:属性名

    javaType: 集合类型

    ofType: 集合中的元素类型

    autoMapping: 开启自动映射

    子标签: 参照resultMap
-->
    <collection property="detailList" javaType="list"
ofType="Orderdetail" autoMapping="true">
        <id column="detail_id" property="id"/>
    </collection>
</resultMap>

<select id="queryOrderWithUserDetail"
resultMap="orderUserDetailMap">
    select *,c.id as detail_id from tb_order a
    LEFT JOIN tb_user b on a.user_id=b.id
    LEFT JOIN tb_orderdetail c on a.id=c.order_id

```

```
        where a.order_number=#{number}
    </select>
```

OrderMapperTest 测试:

```
@Test
public void testQueryOrderWithUserDetail(){

    System.out.println(this.orderMapper.queryOrderWithUserDetail("20140921001"));
}
```

11.4. 多对多查询

多对多查询: 查询订单，查询出下单人信息并且查询出订单详情中的商品数据。

思路:

订单：订单详情 = 1 : n (体现在 pojo 对象中就是在 Order 对象中添加 OrderDetail 对象的集合)

订单详情：商品 = 1 : 1 (体现在 pojo 对象中就是在 OrderDetail 对象中添加 Item 对象)

Sql:

```

2  select * from tb_order o
3      left join tb_user u on o.user_id=u.id
4      left join tb_orderdetail od on o.id=od.order_id
5      left join tb_item i on od.item_id=i.id
6  where o.order_number='20140921001'
7
8
9  |

```

信息	结果1	概况	状态						
id	user_id	order_number	id1	user_name	password	name	age	sex	birthday
1	1	20140921001	1	zhangsan	123456	张三	19	1	1984-08-0
1	1	20140921001	1	zhangsan	123456	张三	19	1	1984-08-0

在 Orderdetail 添加 Item 属性，并添加 get、set 方法：

```

public class Orderdetail {

    private Integer id;

    private Double totalPrice;

    private Integer status;

    private Item item;

    public Item getItem() {
        return item;
    }

    public void setItem(Item item) {
        this.item = item;
    }
}

```

OrderMapper 接口：

```

/**
 * 查询订单，查询出下单人信息并且查询出订单详情中的商品数据。
 * @param number
 * @return
 */
public Order queryOrderWithUserDetailItem(@Param("number")String number);

```

OrderMapper 配置 (通过在 collection 标签中嵌套使用 association 标签):

```
<resultMap type="Order" id="orderUserDetailItemMap"
autoMapping="true">
    <id column="id" property="id"/>
    <association property="user" javaType="User"
autoMapping="true">
        <id column="user_id" property="id"/>
    </association>
    <collection property="detailList" javaType="list"
ofType="Orderdetail" autoMapping="true">
        <id column="detail_id" property="id"></id>
        <association property="item" javaType="Item"
autoMapping="true">
            <id column="item_id" property="id"/>
        </association>
    </collection>
</resultMap>

<select id="queryOrderWithUserDetailItem"
resultMap="orderUserDetailItemMap">
    select *,c.id as detail_id from tb_order a
    LEFT JOIN tb_user b on a.user_id=b.id
    LEFT JOIN tb_orderdetail c on a.id=c.order_id
    LEFT JOIN tb_item d on c.item_id=d.id
    where a.order_number=#{number}
</select>
```

OrderMapperTest 测试:

```
@Test
public void testQueryOrderWithUserDetailItem(){

    System.out.println(this.orderMapper.queryOrderWithUserDetailItem("20140921001"));
}
```

11.5. resultMap 的继承

```
<resultMap type="Order" id="OrderUserResultMap" autoMapping="true">
  <id column="id" property="id"/>
  <!--
```

association: 完成子对象的映射

property: 子对象在父对象中的属性名称

javaType: 子对象的java数据类型

autoMapping: 完成子对象的自动映射, 若开启驼峰, 则按驼峰映射。

内容: 完成子对象的映射, 数据库的数据列对应子对象的属性名, 使用方式参考resultMap。

如果子对象的id不固定, 需要用别名固定

-->

```
<association property="user" javaType="User" autoMapping="true">
```

```
  <id column="user_id" property="id"/>
```

```
</association>
```

```
</resultMap>
```

```
<resultMap type="Order" id="OrderUserDetailResultMap" autoMapping="true" extends="OrderUserResultMap">
  <!--
```

collection: 定义子对象集合的映射

property: 子对象集合在父对象中的属性名

javaType: 集合类型

ofType: 子对象的java数据类型

autoMapping: 开启自动映射

内容: 子对象的属性映射, 参考resultMap

-->

```
<collection property="detailList" javaType="List" ofType="Orderdetail" autoMapping="true">
```

```
  <id column="detail_id" property="id"/>
```

```
</collection>
```

```
</resultMap>
```

继承了之后便有了一对一以及id的映射配置

11.6. 高级查询的整理

resultType 无法帮助我们自动的去完成映射, 所以只有使用 resultMap 手动的进行映射

resultMap:

type 结果集对应的数据类型

id 唯一标识, 被引用的时候, 进行指定

autoMapping 开启自动映射

extends 继承

子标签:

association: 一对一的映射

property 定义对象的属性名

javaType 属 性 的 类 型

autoMapping 开启自动映射

collection: 一对多的映射

property 定义对象的属性名

javaType 集合的类型

ofType 集合中的元素类型

autoMapping 开启自动映射

12. 延迟加载

设置参数	描述
cacheEnabled	该配置影响的所有映射器中配置的缓存的全局开关。
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 可通过设置 fetchType 属性来覆盖该项的开关状态。
aggressiveLazyLoading	当启用时，带有延迟加载属性的对象的加载与否完全取决于对 用；反之，每种属性将会按需加载。

mapUnderscoreToCamelCase	是否开启自动驼峰命名规则 (camel case) 映射, 即从经典类 A_COLUMN 到经典 Java 属性名 aColumn 的类似映射。
--------------------------	--

分析:

采用之前的配置方式 (参考高级查询), 肯定是不能做到延迟加载的, 因为咱是通过一个查询 sql 直接查询出所有的数据。为了测试延迟加载的效果, 必须改造高级查询的配置, 使 Order 的查询和 User 或者 OrderDetail 的查询分开。只有当我们访问 Order 对象的 User 或者 OrderDetail 属性时, 才去执行 User 或者 OrderDetail 的查询。

12.1. 改造一对一查询

在 OrderMapper 接口中, 编写接口方法:

```
/**
 * 测试延迟加载
 * @param number
 * @return
 */
public Order queryOrderLazy(@Param("number")String number);
```

OrderMapper 配置:

```

<resultMap id="orderUserLazyMap" type="Order" autoMapping="true">
  <id column="id" property="id"/>
  <!-- 一对一的映射 -->
  <association property="user" javaType="User" select="queryUser" column="id"/>
</resultMap>

<!-- 查询一，获取Order信息，如要获取Order对象中的User信息，需要关联查询，所以必须使用resultMap -->
<select id="queryOrderLazy" resultMap="orderUserLazyMap">
  select * from tb_order where order_number=#{number}
</select>

<!-- 查询二，获取User信息，不需要关联查询，只需要根据id查询出User信息，使用resultType就行 -->
<select id="queryUser" resultType="User">
  select * from tb_user where id=#{id}
</select>

```

指定延迟加载要执行的statementId

指定结果集中的外键

由于只有一个参数，所以可以任意

处理组合键时，需要传递多个参数，可以使用 `column="{prop1=col1, prop2=col2, prop3=col3...}"`，设置多个列名传入到嵌套查询语句，mybatis 会把 prop1,prop2,prop3 设置到目标嵌套的查询语句中的参数对象中。

子查询中，必须通过 prop1,prop2,prop3 获取对应的参数值，你也可以使用这种方式指定参数名例如：

```

<resultMap id="orderUserLazyMap" type="Order" autoMapping="true">
  <id column="id" property="id"/>
  <!-- 一对一的映射 -->
  <association property="user" javaType="User" select="queryUser" column="id"/>
</resultMap>

<!-- 查询一，获取Order信息，如要获取Order对象中的User信息，需要关联查询，所以必须使用resultMap -->
<select id="queryOrderLazy" resultMap="orderUserLazyMap">
  select * from tb_order where order_number=#{number}
</select>

<!-- 查询二，获取User信息，不需要关联查询，只需要根据id查询出User信息，使用resultType就行 -->
<select id="queryUser" resultType="User">
  select * from tb_user where id=#{user_id}
</select>

```

此时必须通过#{user_id}获取

测试：

```
@Test
public void testQueryOrderLazy(){
    Order order = this.orderMapper.queryOrderLazy("20140921001");
    System.out.println(order.getOrderNumber());
    System.out.println(order.getUser());
}
```

日志信息：

```
2017-04-13 00:11:14,982 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryOrderLazy]-[DEBUG] ==> Pre
2017-04-13 00:11:15,028 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryOrderLazy]-[DEBUG] ==> Para
2017-04-13 00:11:15,049 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryUser]-[DEBUG] ==> Prepar
2017-04-13 00:11:15,049 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryUser]-[DEBUG] ==> Paramet
2017-04-13 00:11:15,057 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryUser]-[DEBUG] <==== Tot
2017-04-13 00:11:15,057 [main] [cn.itcast.mybatis.mapper.OrderMapper.queryOrderLazy]-[DEBUG] <==
Order [id=1, userId=null, orderNumber=20140921001, user=User [id=1, userName=zhangsan, password=12345
Sep 19 16:56:04 CST 2014, updated=Sun Sep 21 11:24:59 CST 2014], detailList=null]
20140921001
User [id=1, userName=zhangsan, password=123456, name=张三, age=30, sex=1, birthday=Wed Aug 08 00:00:00
CST 2014]
```

分析：红色部分为查询 Order 信息的日志，绿色为查询 Order 中的 User 信息的日志。已

经成功拆分成两个子查询，并且查询出了带有 User 信息的 Order 信息，说明改造 OK。

接下来开启延迟加载

12.2. 开启延迟加载

在 mybatis-config.xml 中配置行为参数：

```

<settings>
    <!-- 开启驼峰匹配，经典的数据库列名（多个单词时，以下划线连接）到经典java属性名（多个单词时，以驼峰命名） -->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
    <!-- 开启延迟加载 -->
    <setting name="lazyLoadingEnabled" value="true"/>
    <!-- 开启按需加载 -->
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>

```

修改测试用例，注释掉打印 Order 信息的这行代码：

```

@Test
public void testQueryOrderLazy() {
    Order order = this.orderMapper.queryOrderLazy("20140921001");
    // System.out.println(order);
    System.out.println(order.getOrderNumber());
    System.out.println(order.getUser());
}

```

执行，报错：

```

org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause: java.lang.IllegalStateException: Cannot enable lazy loading because CGLIB is not available
### The error may exist in cn/itcast/mybatis/mapper/OrderMapper.xml
### The error may involve cn.itcast.mybatis.mapper.OrderMapper.queryOrderLazy
### The error occurred while handling results
### SQL: select * from tb_order where order number=?
### Cause: java.lang.IllegalStateException: Cannot enable lazy loading because CGLIB is not available
    at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:26)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:111)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:102)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:66)
    at org.apache.ibatis.binding.MapperMethod.execute(MapperMethod.java:68)
    at org.apache.ibatis.binding.MapperProxy.invoke(MapperProxy.java:52)
    at com.sun.proxy.$Proxy3.queryOrderLazy(Unknown Source)
    at cn.itcast.mybatis.mapper.OrderMapperTest.testQueryOrderLazy(OrderMapperTest.java:30)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

```

说明延迟加载需要 cglib 的支持。

在 pom.xml 中，添加 cglib 的依赖：

```

<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.1</version>
</dependency>

```

执行：

```
2017-04-13 00:29:38,878 [main] [cn.itcast.mybatis.mapper.OrderMapper.query
2017-04-13 00:29:38,934 [main] [cn.itcast.mybatis.mapper.OrderMapper.query
2017-04-13 00:29:39,068 [main] [cn.itcast.mybatis.mapper.OrderMapper.query
20140921001 访问order_number时，user信息并没有加载
2017-04-13 00:29:39,069 [main] [cn.itcast.mybatis.mapper.OrderMapper.query
2017-04-13 00:29:39,069 [main] [cn.itcast.mybatis.mapper.OrderMapper.query
2017-04-13 00:29:39,076 [main] [cn.itcast.mybatis.mapper.OrderMapper.query
User [id=1, userName=zhangsan, password=123456, name=1, age=20] 访问order中的user属性时，执
```

13. 如果 sql 语句中出现' <' 的解决方案

1 、 使 用 xml 中 的 字 符 实 体

<	<	小于
>	>	大于
&	&	和号
'	'	省略号
"	"	引号

2、使用<![CDATA[<]]>