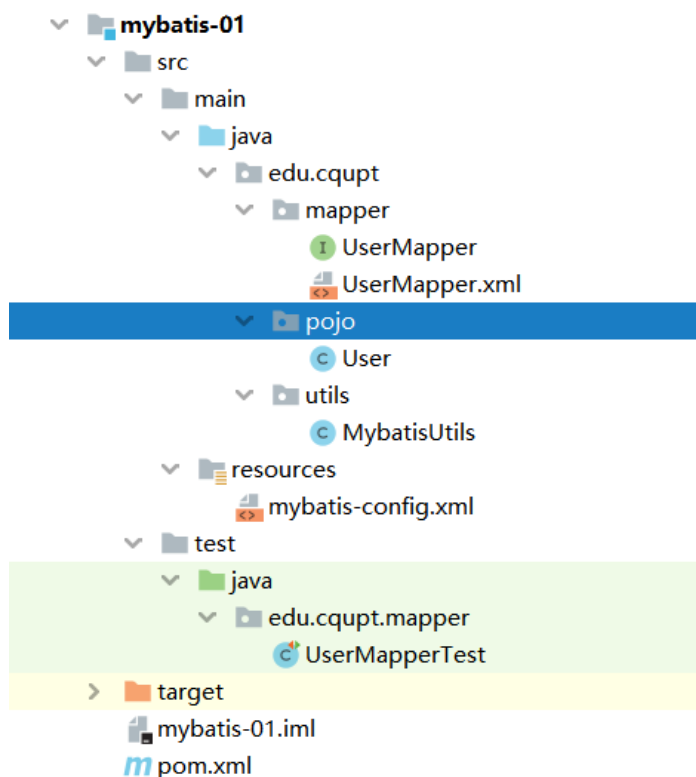


Mybatis

搭建第一个Mybatis程序



导入相应maven依赖,配置资源导出路径

pom.xml

```
1  <!--导入依赖-->
2  <dependencies>
3      <!--mysql驱动-->
4      <dependency>
5          <groupId>mysql</groupId>
6          <artifactId>mysql-connector-java</artifactId>
7          <version>5.1.47</version>
8      </dependency>
9      <!--mybatis-->
10     <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
11     <dependency>
12         <groupId>org.mybatis</groupId>
13         <artifactId>mybatis</artifactId>
14         <version>3.5.2</version>
```

```
15     </dependency>
16     <!--junit-->
17     <dependency>
18         <groupId>junit</groupId>
19         <artifactId>junit</artifactId>
20         <version>4.12</version>
21     </dependency>
22 </dependencies>
23 <!--在build中配置resources，来防止我们资源导出失败的问题-->
24 <build>
25     <resources>
26         <resource>
27             <directory>src/main/resources</directory>
28             <includes>
29                 <include>**/*.properties</include>
30                 <include>**/*.xml</include>
31             </includes>
32             <filtering>true</filtering>
33         </resource>
34         <resource>
35             <directory>src/main/java</directory>
36             <includes>
37                 <include>**/*.properties</include>
38                 <include>**/*.xml</include>
39             </includes>
40             <filtering>true</filtering>
41         </resource>
42     </resources>
43 </build>
```

编写实体类

```
1 package edu.cqupt.pojo;
2 //实体类
3 public class User {
4     private int id;
5     private String name;
6     private String pwd;
7
8     public User() {
9     }
10
11     public User(int id, String name, String pwd) {
12         this.id = id;
13         this.name = name;
14         this.pwd = pwd;
15     }
16
17     public int getId() {
```

```
18         return id;
19     }
20
21     public void setId(int id) {
22         this.id = id;
23     }
24
25     public String getName() {
26         return name;
27     }
28
29     public void setName(String name) {
30         this.name = name;
31     }
32
33     public String getPwd() {
34         return pwd;
35     }
36
37     public void setPwd(String pwd) {
38         this.pwd = pwd;
39     }
40
41     @Override
42     public String toString() {
43         return "User{" +
44             "id=" + id +
45             ", name='" + name + '\'' +
46             ", pwd='" + pwd + '\'' +
47             '}';
48     }
49 }
```

编写工具类

在 `edu.cqupt.utilutils` 路径下创建一个 `MybatisUtils.java`

```
1 package edu.cqupt.utils;
2
3 import org.apache.ibatis.io.Resources;
4 import org.apache.ibatis.session.SqlSession;
5 import org.apache.ibatis.session.SqlSessionFactory;
6 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
7
8 import java.io.IOException;
9 import java.io.InputStream;
10
11 public class MybatisUtils {
12     private static SqlSessionFactory sqlSessionFactory;
```

```

13     static{
14         try {
15             String resource = "mybatis-config.xml";
16             InputStream inputStream = Resources.getResourceAsStream(resource);
17             sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
18         } catch (IOException e) {
19             e.printStackTrace();
20         }
21     }
22     public static SqlSession getSqlSession(){
23         return sqlSessionFactory.openSession();
24     }
25
26 }

```

添加核心配置文件

在上面的工具类中需要一个**mybatis-config.xml** 文件，于是在resources目录下创建一个**mybatis-config.xml**

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5
6 <configuration>
7     <environments default="development">
8         <environment id="development">
9             <transactionManager type="JDBC"/>
10            <dataSource type="POOLED">
11                <property name="driver" value="com.mysql.jdbc.Driver"/>
12                <property name="url" value="jdbc:mysql://localhost:3306/mybatis?useSSL=true&";
13                <property name="username" value="root"/>
14                <property name="password" value="123456"/>
15            </dataSource>
16        </environment>
17    </environments>
18    <mappers>
19        <mapper resource="edu/cqupt/mapper/UserMapper.xml"/>
20    </mappers>
21 </configuration>

```

其中：

- environment：配置的是mysql数据库的参数
- mapper 添加的是Mapper.xml的映射

因此我们需要编写Mapper.xml文件

在`edu.cqpt.mapper`路径下创建：`UserMapper` 接口

编写Mapper接口并实现

```
1 package edu.cqpt.mapper;
2
3 import edu.cqpt.pojo.User;
4 import java.util.List;
5
6 public interface UserMapper {
7     List<User> getUserList();
8
9 }
```

用UserMapper.xml 实现接口

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

编写测试类

```
1 package edu.cqpt.mapper;
2
3 import edu.cqpt.pojo.User;
4 import edu.cqpt.utils.MybatisUtils;
5 import org.apache.ibatis.session.SqlSession;
6 import org.junit.Test;
7 import java.util.List;
8
9 public class UserMapperTest {
10     @Test
11     public void getUserList(){
12         SqlSession sqlSession = MybatisUtils.getSqlSession();
13         UserMapper mapper = sqlSession.getMapper(UserMapper.class);
14         List<User> userList = mapper.getUserList();
15         for (User user : userList) {
16             System.out.println(user);
17         }
18     }
19 }
```

↑

```
18     }  
19 }
```

```
User{id=1, name='zsl', pwd='1234'}  
User{id=2, name='chenxi', pwd='1244'}  
User{id=3, name='yup', pwd='32323'}
```

CUID

查询数据

```
1 User getUserById(int id);
```

```
1 <select id="getUserById" parameterType="int" resultType="edu.cqupt.pojo.User">  
2     select * from mybatis.user where id = #{id}  
3 </select>
```

```
1 @Test  
2 public void getUserById(){  
3     SqlSession sqlSession = MybatisUtils.getSqlSession();  
4     UserMapper mapper = sqlSession.getMapper(UserMapper.class);  
5     User user = mapper.getUserById(2);  
6     System.out.println(user);  
7 }
```

注：增删改需要提交事务： `sqlSession.commit();`

插入数据

```
1 int addUser(User user);
```

```
1 <insert id="addUser" parameterType="edu.cqupt.pojo.User">  
2     insert into mybatis.user(id,name,pwd) values(#{id},#{name},#{pwd})  
3 </insert>
```

```
1 @Test  
2 public void addUser(){  
3     SqlSession sqlSession = MybatisUtils.getSqlSession();
```

```

4      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
5      int res = mapper.addUser(new User(4, "why", "12344"));
6      sqlSession.commit();    //一定要注意提交事务，否则该操作是无效的
7      if(res > 0){
8          System.out.println("插入成功");
9      }
10     sqlSession.close();
11 }

```

修改数据

```

1  int updateUser(User user);

```

```

1  <update id="updateUser" parameterType="edu.cqupt.pojo.User">
2      update mybatis.user set name = #{name}, pwd = #{pwd} where id = #{id}
3  </update>

```

```

1
2  @Test
3  public void updateUser(){
4      SqlSession sqlSession = MybatisUtils.getSqlSession();
5      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6      int res = mapper.updateUser(new User(2, "cx", "12356"));
7      sqlSession.commit();
8      if(res > 0){
9          System.out.println("更新成功");
10     }
11     sqlSession.close();
12 }

```

删除数据

```

1  int deleteUser(int id);

```

```

1  <delete id="deleteUser" parameterType="int">
2      delete from mybatis.user where id = #{id}
3  </delete>

```

```

1  @Test
2  public void deleteUser(){
3      SqlSession sqlSession = MybatisUtils.getSqlSession();

```

```

4      UserMapper mapper = sqlSession.getMapper(UserMapper.class);
5      int res = mapper.deleteUser(7);
6      sqlSession.commit();
7      if(res > 0){
8          System.out.println("删除成功");
9      }
10     sqlSession.close();
11 }

```

万能的Map

假设，我们的实体类，或者数据库中的表，字段或者参数过多，我们应当考虑使用Map！

```

1      //万能的Map
2      int addUser2(Map<String, Object> map);

```

```

1      <!--对象中的属性，可以直接取出来传递map的key-->
2      <insert id="addUser" parameterType="map">
3          insert into mybatis.user (id, pwd) values (#{userid},#{passWord});
4      </insert>

```

```

1      @Test
2      public void addUser2(){
3          SqlSession sqlSession = MybatisUtils.getSqlSession();
4          UserMapper mapper = sqlSession.getMapper(UserMapper.class);
5          Map<String, Object> map = new HashMap<String, Object>();
6          map.put("userid", 5);
7          map.put("passWord", "2222333");
8          mapper.addUser2(map);
9          sqlSession.close();
10     }

```

Map传递参数，直接在sql中取出key即可！ 【parameterType="map"】

对象传递参数，直接在sql中取对象的属性即可！ 【parameterType="Object"】

只有一个基本类型参数的情况下，可以直接在sql中取到！

多个参数用Map，或者注解！



模糊查询

```

1 List<User> getUserByLike(String value);

```


- 第一种SQL (推荐)

```
1 <select id="getUserByLike"  resultType="edu.cqupt.pojo.User">
2     select * from mybatis.user where name like #{value}
3 </select>
```

- 第二种SQL (不推荐, 会存在SQL注入)

```
1 <select id="getUserByLike"  resultType="edu.cqupt.pojo.User">
2     select * from mybatis.user where name like "%#{value}%"
3 </select>
```

```
1 @Test
2 public void getUserByLike(){
3     SqlSession sqlSession = MybatisUtils.getSqlSession();
4     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
5     List<User> userList = mapper.getUserByLike("%z%");
6     //List<User> userList = mapper.getUserByLike("z");
7     for (User user : userList) {
8         System.out.println(user);
9     }
10 }
```

配置解析

核心配置文件

- mybatis-config.xml
- MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息。
- configuration (配置)
- properties (属性)
- settings (设置)
- typeAliases (类型别名)
- typeHandlers (类型处理器)
- objectFactory (对象工厂)
- plugins (插件)
- environments (环境配置)
- environment (环境变量)
- transactionManager (事务管理器)
- dataSource (数据源)

- databaseldProvider (数据库厂商标识)
- mappers (映射器)

环境配置 (environments)

MyBatis 可以配置成适应多种环境

不过要记住：尽管可以配置多个环境，但每个 SqlSessionFactory 实例只能选择一种环境。

学会使用配置多套运行环境！

Mybatis默认的事务管理器就是 **JDBC** (还有一种是MANAGED) ， 连接池：**POOLED** (还有两种是UNPOOLED 和 JNDI)

属性 (properties)

我们可以通过properties属性来实现引用配置文件，这些属性都是可外部配置且可动态替换的，既可以在典型的 Java 属性文件中配置，亦可通过 properties 元素的子元素来传递。【db.properties】



编写一个配置文件

db.properties

```
1 driver=com.mysql.jdbc.Driver  
2 url=jdbc:mysql://localhost:3306/mybatis?useSSL=true&useUnicode=true&characterEncoding=UTF-8  
3 username=root  
4 password=123456
```

在核心配置文件中映入

```
1 <!--引入外部配置文件-->  
2 <properties resource="db.properties">  
3   <property name="username" value="root"/>  
4   <property name="pwd" value="1111"/>  
5 </properties>
```

- 可以直接引入外部文件
- 可以在其中增加一些属性配置
- 如果两个文件有同一个字段，优先使用外部配置文件的！

↑

类型别名 (typeAliases)

- 类型别名是为 Java 类型设置一个短的名字。
- 存在的意义：仅在于用来减少类完全限定名的冗余。

```
1 <!--可以给实体类起别名-->
2 <typeAliases>
3     <typeAlias type="com.kuang.pojo.User" alias="User"/>
4 </typeAliases>
```

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean，比如：扫描实体类的包，它的默认别名就为这个类的类名，首字母小写！

```
1 <!--可以给实体类起别名-->
2 <typeAliases>
3     <package name="com.kuang.pojo"/>
4 </typeAliases>
```

在实体类比较少的时候，使用第一种方式。如果实体类十分多，建议使用第二种。
第一种可以DIY别名，第二种则不行，如果非要改，需要在实体上增加注解

```
1 @Alias("user")
2 public class User {}
```

设置

这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。

cacheEnabled	全局地开启或关闭配置文件中的所有映射器已经配置的任何缓存。	true false	true
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。特定关联关系中可通过设置 <code>fetchType</code> 属性来覆盖该项的开关状态。	true false	false
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J LOG4J LOG4J2 JDK_LOGGING COMMONS_LOGGING STDOUT_LOGGING NO_LOGGING	未设置

其他配置

- typeHandlers（类型处理器） <<https://mybatis.org/mybatis-3/zh/configuration.html#typeHandlers>>
- objectFactory（对象工厂） <<https://mybatis.org/mybatis-3/zh/configuration.html#objectFactory>>
- plugins插件
 - mybatis-generator-core
 - mybatis-plus
 - 通用mapper

映射器（mappers）

MapperRegistry：注册绑定我们的Mapper文件；
方式一：【推荐使用】

```
1 <!--每一个Mapper.XML都需要在Mybatis核心配置文件中注册!-->
2 <mappers>
3     <mapper resource="com/kuang/dao/UserMapper.xml"/>
4 </mappers>
```

方式二：使用class文件绑定注册

```
1 <!--每一个Mapper.XML都需要在Mybatis核心配置文件中注册!-->
2 <mappers>
3     <mapper class="com.kuang.dao.UserMapper"/>
4 </mappers>
```

注意点：

- 接口和他的Mapper配置文件必须同名！
- 接口和他的Mapper配置文件必须在同一个包下！

方式三：使用扫描包进行注入绑定

```
1 <!--每一个Mapper.XML都需要在Mybatis核心配置文件中注册!-->
2 <mappers>
3     <package name="com.kuang.dao"/>
4 </mappers>
```

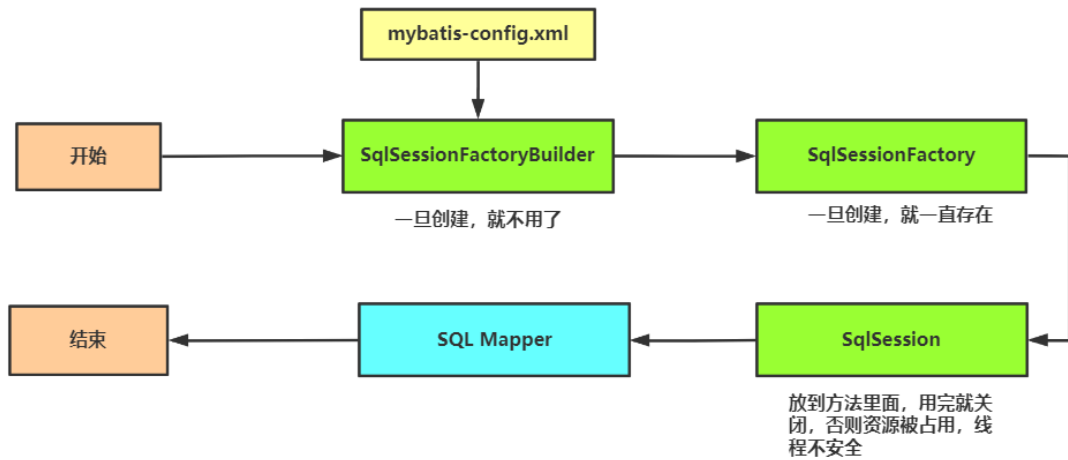
注意点：

- **接口和他的Mapper配置文件必须同名！**
- **接口和他的Mapper配置文件必须在同一个包下！**
-

练习时间：

- 将数据库配置文件外部引入
- 实体类别名
- 保证UserMapper 接口 和 UserMapper .xml 改为一致！并且放在同一个包下！

生命周期和作用域



生命周期，和作用域，是至关重要的，因为错误的使用会导致非常严重的**并发问题**。

SqlSessionFactoryBuilder:

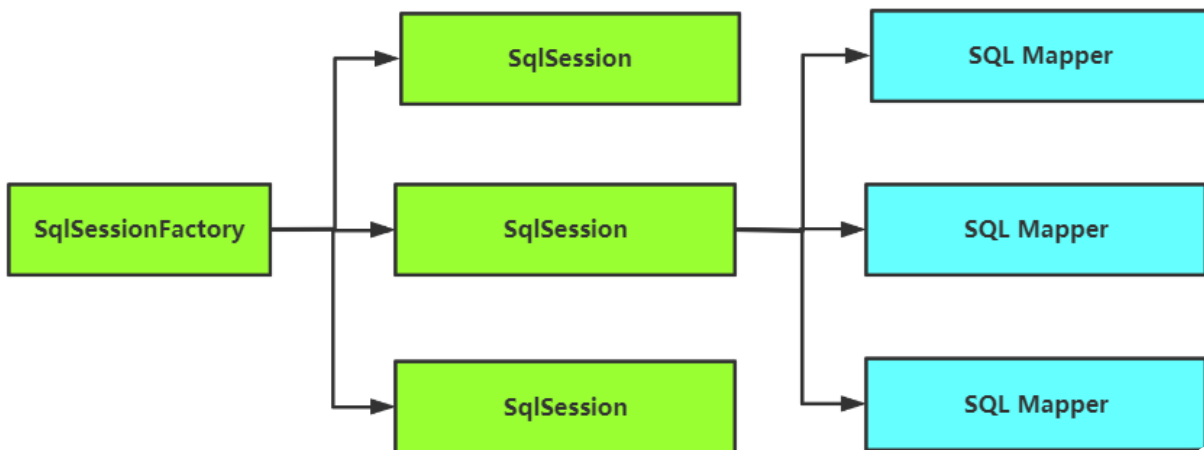
- 一旦创建了 SqlSessionFactory，就不再需要它了
- 局部变量

SqlSessionFactory:

- 说白了就是可以想象为：数据库连接池
- SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，**没有任何理由丢弃它或重新创建另一个实例。**
- 因此 SqlSessionFactory 的**最佳作用域是应用作用域**。
- 最简单的就是使用**单例模式**或者静态单例模式。

SqlSession

- 连接到连接池的一个请求！
- SqlSession 的实例不是线程安全的，因此是不能被共享的**，所以它的最佳的作用域是请求或方法作用域。
- 用完之后需要赶紧关闭，否则资源被占用！



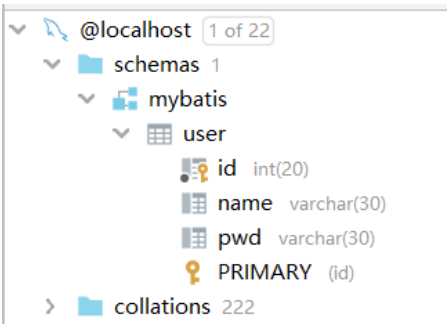
这里的每一个Mapper，就代表一个具体的业务！

↑

解决属性名和字段名不一致的问题

问题

数据库中的字段



新建一个项目，拷贝之前的，测试实体类字段不一致的情况

```
1 public class User {
2
3     private int id;
4     private String name;
5     private String password;
6 }
```

测试出现问题



```
1 //      select * from mybatis.user where id = #{id}
2 //类型处理器
3 //      select id,name,pwd from mybatis.user where id = #{id}
```

解决方法：

- 起别名

```
1 <select id="getUserById" resultType="com.kuang.pojo.User">
2     select id,name,pwd as password from mybatis.user where id = #{id}
3 </select>
```

resultMap

结果集映射

```
1 id    name    pwd
2 id    name    password
```

```

1 <!--结果集映射-->
2 <resultMap id="UserMap" type="User">
3     <!--column数据库中的字段，property实体类中的属性-->
4     <result column="id" property="id"/>
5     <result column="name" property="name"/>
6     <result column="pwd" property="password"/>
7 </resultMap>
8
9 <select id="getUserById" resultMap="UserMap">
10     select * from mybatis.user where id = #{id}
11 </select>

```

```

<resultMap id="UserMap" type="User">
    <result column="id" property="id"></result>
    <result column="name" property="name"></result>
    <result column="pwd" property="password"></result>
</resultMap> 数据库字段      实体类属性
<select id="getUserById" parameterType="int" resultMap="UserMap">
    select * from mybatis.user where id = #{id}
</select>

```

- resultMap 元素是 MyBatis 中最重要最强大的元素
- ResultMap 的设计思想是，对于简单的语句根本不需要配置显式的结果映射，而对于复杂一点的语句只需要描述它们的关系就行了。
- ResultMap 最优秀的地方在于，虽然你已经对它相当了解了，但是根本就不需要显式地用到他们。
- 如果世界总是这么简单就好了。

日志

日志工厂

如果一个数据库操作，出现了异常，我们需要排错。日志就是最好的助手！曾经：sout、debug。现在：日志工厂！

logImpl

指定 MyBatis 所用日志的具体实现，未指定时将自动查找。

SLF4J | LOG4J | 未设置
 LOG4J2 |
 JDK_LOGGING |
 COMMONS_LOGGING |
 | STDOUT_LOGGING |
 NO_LOGGING

- SLF4J
- **LOG4J 【掌握】**
- LOG4J2
- JDK_LOGGING
- COMMONS_LOGGING
- **STDOUT_LOGGING 【掌握】**
- NO_LOGGING

在Mybatis中具体使用那个一日志实现，在设置中设定！**STDOUT_LOGGING标准日志输出**。在mybatis核心配置文件中，配置我们的日志！

```

1 <settings>
2   <setting name="logImpl" value="STDOUT_LOGGING"/>
3 </settings>

```

```

Opening JDBC Connection
Created connection 832279283.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@319b92f3]
==> Preparing: select * from mybatis.user where id = ?
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 狂神, 123456
<== Total: 1
User{id=1, name='狂神', password='123456'}
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@319b92f3]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@319b92f3]
Returned connection 832279283 to pool.

```

Log4j

什么是Log4j?

- Log4j是[Apache <https://baike.baidu.com/item/Apache/8512995>](https://baike.baidu.com/item/Apache/8512995) 的一个开源项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台
https://baike.baidu.com/item/%E6%8E%A7%E5%88%B6%E5%8F%B0/2438626、文件、GUI
https://baike.baidu.com/item/GUI 组件
- 我们也可以控制每一条日志的输出格式；
- 通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程。
- 通过一个配置文件
https://baike.baidu.com/item/%E9%85%8D%E7%BD%AE%E6%96%87%E4%BB%B6/286550 来灵活地进行配置，而不需要修改应用的代码。

• 先导入log4j的包

```

1 <!-- https://mvnrepository.com/artifact/log4j/log4j -->
2 <dependency>
3   <groupId>log4j</groupId>
4   <artifactId>log4j</artifactId>
5   <version>1.2.17</version>
6 </dependency>

```

• log4j.properties

```

1 #将等级为DEBUG的日志信息输出到console和file这两个目的地，console和file的定义在下面的代码
2 log4j.rootLogger=DEBUG,console,file
3 #控制台输出的相关设置
4 log4j.appender.console = org.apache.log4j.ConsoleAppender
5 log4j.appender.console.Target = System.out
6 log4j.appender.console.Threshold=DEBUG
7 log4j.appender.console.layout = org.apache.log4j.PatternLayout
8 log4j.appender.console.layout.ConversionPattern=[%c]-%m%n

```



```

9 #文件输出的相关设置
10 log4j.appender.file = org.apache.log4j.RollingFileAppender
11 log4j.appender.file.File=./log/kuang.log
12 log4j.appender.file.MaxFileSize=10mb
13 log4j.appender.file.Threshold=DEBUG
14 log4j.appender.file.layout=org.apache.log4j.PatternLayout
15 log4j.appender.file.layout.ConversionPattern=[%p][%d{yy-MM-dd}][%c]%m%n
16 #日志输出级别
17 log4j.logger.org.mybatis=DEBUG
18 log4j.logger.java.sql=DEBUG
19 log4j.logger.java.sql.Statement=DEBUG
20 log4j.logger.java.sql.ResultSet=DEBUG
21 log4j.logger.java.sql.PreparedStatement=DEBUG

```

• 配置log4j为日志的实现

```

1 <settings>
2   <setting name="logImpl" value="LOG4J"/>
3 </settings>

```

Log4j的使用, 直接测试运行刚才的查询

```

[org.apache.ibatis.io.VFS]-Using VFS adapter org.apache.ibatis.io.DefaultVFS
[org.apache.ibatis.io.DefaultVFS]-Find JAR URL: file:/E:/CodePlace/Java/idea/%e7%8b%82%e7%a5%9e%e8%af%b4Ja
[org.apache.ibatis.io.DefaultVFS]-Not a JAR: file:/E:/CodePlace/Java/idea/%e7%8b%82%e7%a5%9e%e8%af%b4Ja/
[org.apache.ibatis.io.DefaultVFS]-Reader entry: User.class
[org.apache.ibatis.io.DefaultVFS]-Listing file:/E:/CodePlace/Java/idea/%e7%8b%82%e7%a5%9e%e8%af%b4Ja/Mav
[org.apache.ibatis.io.DefaultVFS]-Find JAR URL: file:/E:/CodePlace/Java/idea/%e7%8b%82%e7%a5%9e%e8%af%b4Ja
[org.apache.ibatis.io.DefaultVFS]-Not a JAR: file:/E:/CodePlace/Java/idea/%e7%8b%82%e7%a5%9e%e8%af%b4Ja/
[org.apache.ibatis.io.DefaultVFS]-Reader entry: 1 @
[org.apache.ibatis.io.ResolverUtil]-Checking to see if class edu.cqupt.pojo.User matches criteria [is assi
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all conn
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all conn
[org.apache.ibatis.datasource.pooled.PooledDataSource]-PooledDataSource forcefully closed/removed all conn
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Opening JDBC Connection
[org.apache.ibatis.datasource.pooled.PooledDataSource]-Created connection 1095293768.
[org.apache.ibatis.transaction.jdbc.JdbcTransaction]-Setting autocommit to false on JDBC Connection [com.m
[edu.cqupt.mapper.UserMapper.getUserById]==> Preparing: select * from mybatis.user where id=?
[edu.cqupt.mapper.UserMapper.getUserById]==> Parameters: 3(Integer)
[edu.cqupt.mapper.UserMapper.getUserById]-<==      Total: 1
User{id=3, name='yup', password='12356'}

```

简单使用

1. 在要使用Log4j的类中, 导入包 `import org.apache.log4j.Logger;`
2. 日志对象, 参数为当前类的class

```
static Logger logger = Logger.getLogger(UserDaoTest.class);
```

3. 日志级别

```

1 logger.info("info:进入了testLog4j");
2 logger.debug("debug:进入了testLog4j");

```

```
3 logger.error("error:进入了testLog4j");
```

分页

思考：为什么要分页？

- 减少数据的处理量

使用Limit分页

语法： `SELECT * from user limit startIndex,pageSize;`

```
SELECT * from user limit 3; #[0,n]
```

使用Mybatis实现分页，核心SQL

1. 接口

```
1 //分页
2 List<User> getUserByLimit(Map<String,Integer> map);
```

UserMapper.xml

```
1 <!--//分页-->
2 <select id="getUserByLimit" parameterType="map" resultMap="UserMap">
3     select * from mybatis.user limit #{startIndex},#{pageSize}
4 </select>
```

测试

```
1 @Test
2 public void getUserByLimit(){
3     SqlSession sqlSession = MybatisUtils.getSqlSession();
4     UserMapper mapper = sqlSession.getMapper(UserMapper.class);
5     HashMap<String, Integer> map = new HashMap<String, Integer>();
6     map.put("startIndex",1);
7     map.put("pageSize",2);
8     List<User> userList = mapper.getUserByLimit(map);
9     for (User user : userList) {
10         System.out.println(user);
11     }
12     sqlSession.close();
13 }
```

RowBounds分页

不再使用SQL实现分页

接口

```
1 //分页2
2 List<User> getUserByRowBounds();
```

UserMapper.xml

```
1 <!--分页2-->
2 <select id="getUserByRowBounds" resultMap="UserMap">
3     select * from mybatis.user
4 </select>
```

测试

```
1 @Test
2 public void getUserByRowBounds(){
3     SqlSession sqlSession = MybatisUtils.getSqlSession();
4     //RowBounds实现
5     RowBounds rowBounds = new RowBounds(1, 2);
6     //通过Java代码层面实现分页
7     List<User> userList = sqlSession.selectList("com.kuang.dao.UserMapper.getUserByRowBounds", null, rowBounds);
8     for (User user : userList) {
9         System.out.println(user);
10    }
11    sqlSession.close();
12 }
```

分页插件

MyBatis 分页插件 PageHelper

如果你也在用 MyBatis，建议尝试该分页插件，这一定是最方便使用的分页插件。分页插件支持任何复杂的单表、多表分页。

[View on Github](#)[View on GitOsc](#)

maven central 5.1.10

了解即可，需要使用，需要知道它是什么东西！

使用注解开发

面向接口编程

- 大家之前都学过面向对象编程，也学习过接口，但在真正的开发中，很多时候我们会选择面向接口编程
- 根本原因：**解耦**，可拓展，提高复用，分层开发中，上层不用管具体的实现，大家都遵守共同的标准，使得开发变得容易，规范性更好
- 在一个面向对象的系统中，系统的各种功能是由许许多多的不同对象协作完成的。在这种情况下，各个对象内部是如何实现自己的,对系统设计人员来讲就不那么重要了；
- 而各个对象之间的协作关系则成为系统设计的关键。小到不同类之间的通信，大到各模块之间的交互，在系统设计之初都是要着重考虑的，这也是系统设计的主要工作内容。面向接口编程就是指按照这种思想来编程。

关于接口的理解

- 接口从更深层次的理解，应是定义（规范，约束）与实现（名实分离的原则）的分离。
- 接口的本身反映了系统设计人员对系统的抽象理解。
- 接口应有两类：
- 第一类是**对一个个体的抽象**，它可对应为一个抽象体(**abstract class**)；
- 第二类是**对一个个体某一方面的抽象**，即形成一个抽象面 (**interface**) ；
- 一个体有可能有多个抽象面。抽象体与抽象面是有区别的。

三个面向区别

- 面向对象是指，我们考虑问题时，以对象为单位，考虑它的属性及方法。
- 面向过程是指，我们考虑问题时，以一个具体的流程（事务过程）为单位，考虑它的实现。
- 接口设计与非接口设计是针对复用技术而言的，与面向对象（过程）不是一个问题.更多的体现就是对系统整体的架构

使用注解开发



1. 注解在接口上实现

```
1 @Select("select * from user")
2 List<User> getUsers();
```

2. 需要再核心配置文件中绑定接口！

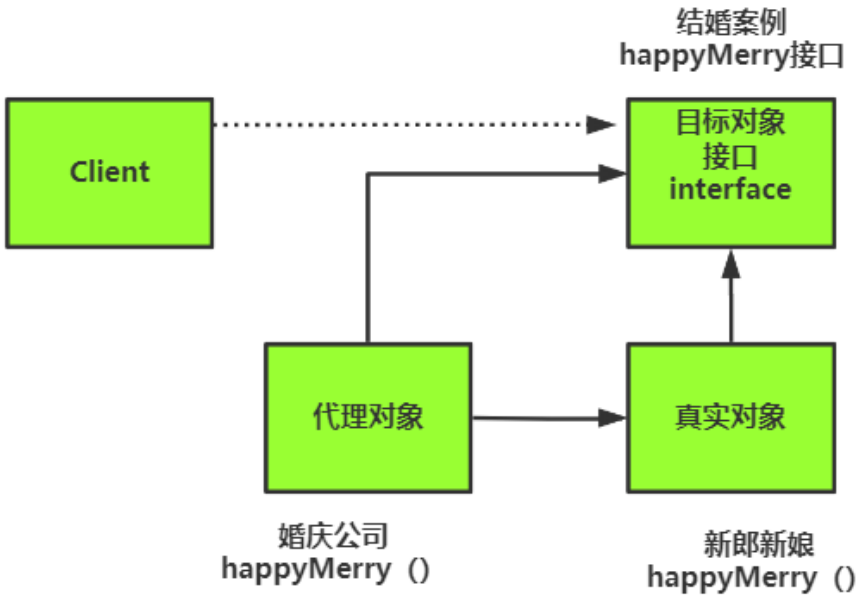
```
1 <!--绑定接口-->
2 <mappers>
3     <mapper class="com.kuang.dao.UserMapper"/>
4 </mappers>
```

3. 测试

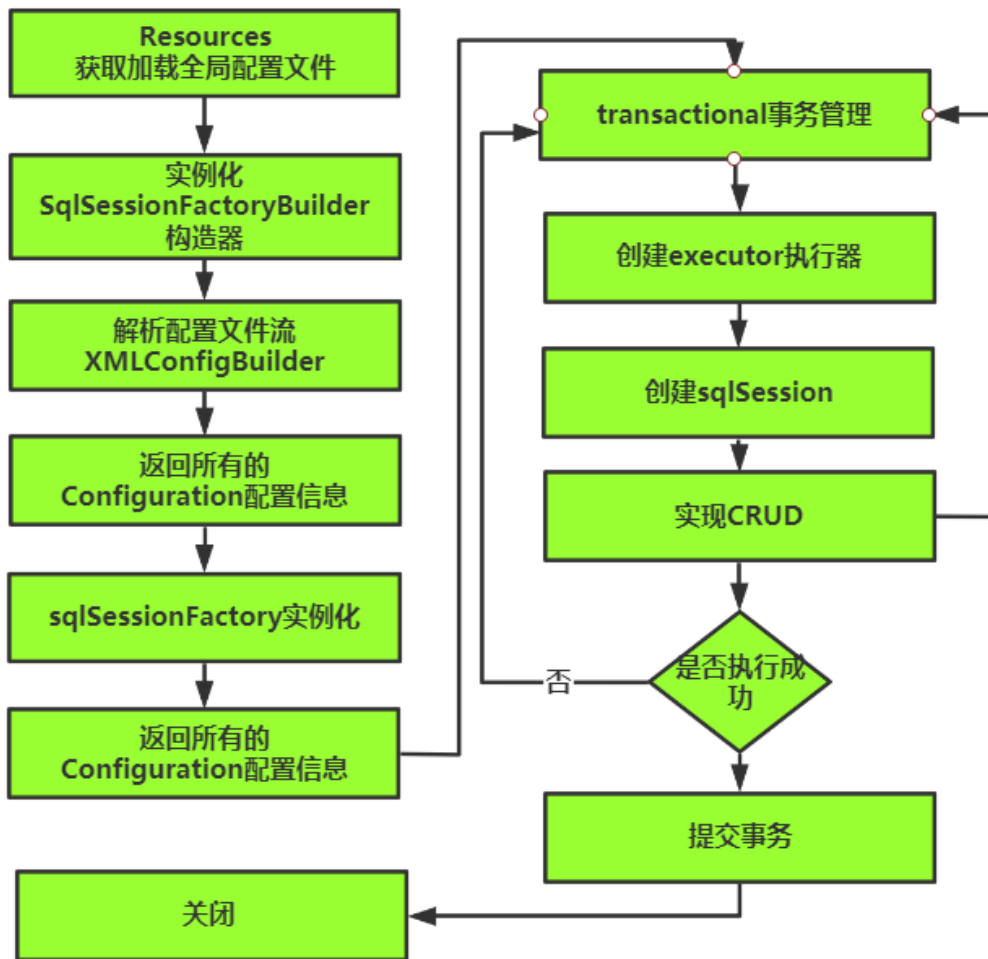
```
Opening JDBC Connection
Created connection 1773206895.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@69b0fd6f]
==> Preparing: select * from user
==> Parameters:
<== Columns: id, name, pwd
<== Row: 1, zsl, 12356
<== Row: 2, cx, 12356
<== Row: 3, yup, 12356
<== Row: 4, why, 12356
<== Row: 5, lw, 12356
<== Row: 6, xm, 12356
<== Row: 7, zack, 123456
<== Row: 8, nick, 123456
<== Total: 8
User{id=1, name='zsl', password='null'}
User{id=2, name='cx', password='null'}
User{id=3, name='yup', password='null'}
User{id=4, name='why', password='null'}
User{id=5, name='lw', password='null'}
User{id=6, name='xm', password='null'}
User{id=7, name='zack', password='null'}
User{id=8, name='nick', password='null'}
```

本质：反射机制实现

底层：动态代理！



Mybatis详细的执行流程！



CRUD

我们可以在工具类创建的时候实现**自动提交事务!**

```

1 public static SqlSession getSqlSession(){
2     return sqlSessionFactory.openSession(true);
3 }

```

编写接口，增加注解

```

1 public interface UserMapper {
2
3     @Select("select * from user")
4     List<User> getUsers();
5
6     // 方法存在多个参数，所有的参数前面必须加上 @Param("id")注解
7     @Select("select * from user where id = #{id}")
8     User getUserByID(@Param("id") int id);
9
10
11     @Insert("insert into user(id,name,pwd) values (#{id},#{name},#{password})")
12     int addUser(User user);
13 }

```

```
14
15     @Update("update user set name=#{name},pwd=#{password} where id = #{id}")
16     int updateUser(User user);
17
18
19     @Delete("delete from user where id = #{uid}")
20     int deleteUser(@Param("uid") int id);
21 }
```

测试类

```
1 public class UserMapperTest {
2     @Test
3     public void getUsers(){
4         SqlSession sqlSession = MybatisUtils.getSqlSession();
5         UserMapper mapper = sqlSession.getMapper(UserMapper.class);
6         List<User> userList = mapper.getUsers();
7         for (User user : userList) {
8             System.out.println(user);
9         }
10    }
11    @Test
12    public void getUsersByIdName(){
13        SqlSession sqlSession = MybatisUtils.getSqlSession();
14        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
15        User user = mapper.getUserByIdName(1, "zsl");
16        System.out.println(user);
17    }
18    }
19    @Test
20    public void addUser(){
21        SqlSession sqlSession = MybatisUtils.getSqlSession();
22        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
23        int hello = mapper.addUser(new User(9, "hello", "123456"));
24        System.out.println(hello);
25    }
26    @Test
27    public void updateUser(){
28        SqlSession sqlSession = MybatisUtils.getSqlSession();
29        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
30        int hello = mapper.updateUser(new User(9, "haha", "123456"));
31        System.out.println(hello);
32    }
33    @Test
34    public void deleteUser(){
35        SqlSession sqlSession = MybatisUtils.getSqlSession();
36        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
37        int hello = mapper.deleteUser(9);
38        System.out.println(hello);
39    }
40 }
```

```
39     }
40 }
```

【注意：我们必须要讲接口注册绑定到我们的核心配置文件中！】

关于@Param() 注解

- 基本类型的参数或者String类型，需要加上
- 引用类型不需要加
- 如果只有一个基本类型的话，可以忽略，但是建议大家都加上！
- 我们在SQL中引用的就是我们这里的 @Param() 中设定的属性名！
-

mybatis中的#和\$的区别：

1、#将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。 如：where username=#{username}，如果传入的值是111,那么解析成sql时的值为where username="111",如果传入的值是id，则解析成的sql为where username="id". 2、\$将传入的数据直接显示生成在sql中。

如：where username=\${username}，如果传入的值是111,那么解析成sql时的值为where username=111;

如果传入的值是;drop table user;, 则解析成的sql为：select id, username, password, role from user where username=;drop table user;

3、#方式能够很大程度防止sql注入，\$方式无法防止Sql注入。

4、\$方式一般用于传入数据库对象，例如传入表名。

5、一般能用#的就别用\$，若不得不使用“\${xxx}”这样的参数，要手工地做好过滤工作，来防止sql注入攻击。

6、在MyBatis中，“\${xxx}”这样格式的参会会直接参与SQL编译，从而不能避免注入攻击。但涉及到动态表名和列名时，只能使用“\${xxx}”这样的参数格式。所以，这样的参数需要我们在代码中手工进行处理来防止注入。

【结论】在编写MyBatis的映射语句时，尽量采用“#{xxx}”这样的格式。若不得不使用“\${xxx}”这样的参数，要手工地做好过滤工作，来防止SQL注入攻击。参考：

<https://www.cnblogs.com/myseries/p/10821372.html>

<<https://www.cnblogs.com/myseries/p/10821372.html>>

Lombok

- 1 Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java.
- 2 Never write another getter or equals method again, with one annotation your `class` has a `fully featured builder`, Automate your logging variables, and much more.

- java library
- plugs

- build tools
- with one annotation your class

使用步骤:

1. 在IDEA中安装Lombok插件!
2. 在项目中导入lombok的jar包

```
1 <dependency>
2   <groupId>org.projectlombok</groupId>
3   <artifactId>lombok</artifactId>
4   <version>1.18.10</version>
5 </dependency>
```

3. 在实体类上加注解即可!

```
1 @Data
2 @AllArgsConstructor
3 @NoArgsConstructor
```

```
1 @Getter and @Setter
2 @FieldNameConstants
3 @ToString
4 @EqualsAndHashCode
5 @AllArgsConstructor, @RequiredArgsConstructor and @NoArgsConstructor
6 @Log, @Log4j, @Log4j2, @Slf4j, @XSlf4j, @CommonsLog, @JBossLog, @Flogger
7 @Data
8 @Builder
9 @Singular
10 @Delegate
11 @Value
12 @Accessors
13 @Wither
14 @SneakyThrows
```

说明:

```
1 @Data: 无参构造, get、set、toString、hashCode, equals
2 @AllArgsConstructor
3 @NoArgsConstructor
4 @EqualsAndHashCode
5 @ToString
6 @Getter
```

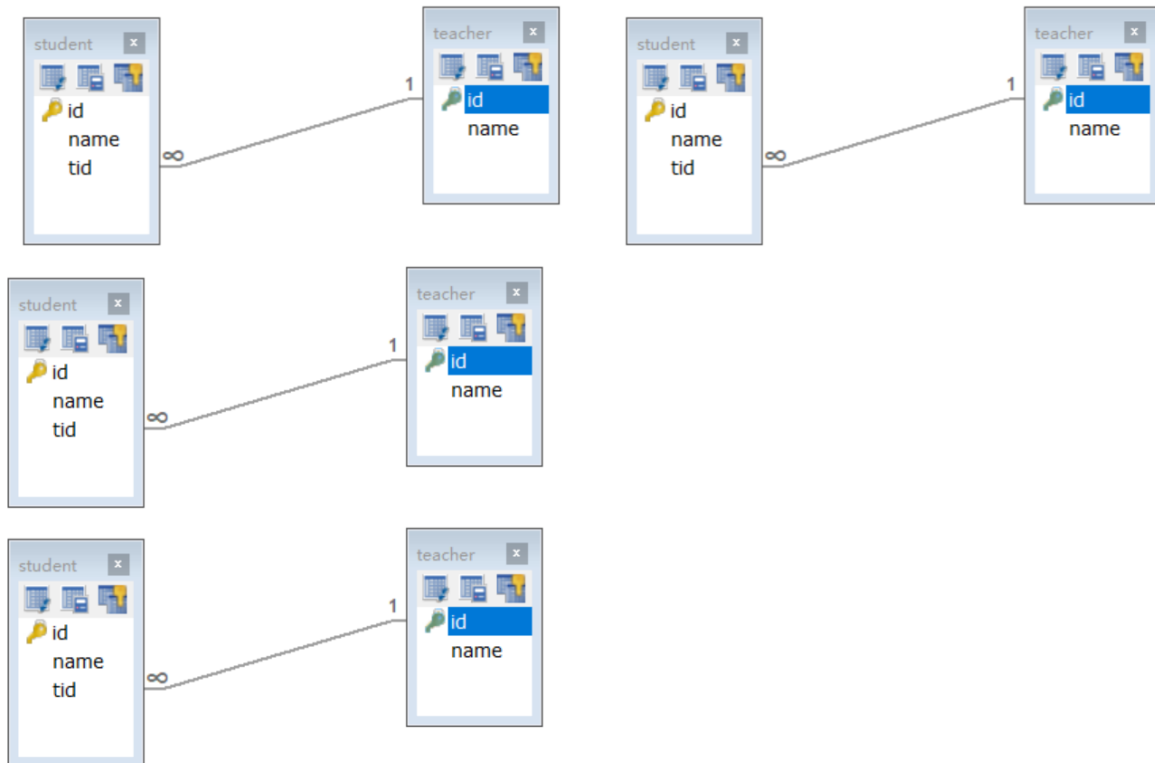


多对一处理

多对一处理

多对一：

- 多个学生，对应一个老师
- 对于学生这边而言， **关联** .. 多个学生，关联一个老师 【多对一】
- 对于老师而言， **集合** ， 一个老师，有很多学生 【一对多】



SQL:

```

1 CREATE TABLE `teacher` (
2   `id` INT(10) NOT NULL,
3   `name` VARCHAR(30) DEFAULT NULL,
4   PRIMARY KEY (`id`)
5 ) ENGINE=INNODB DEFAULT CHARSET=utf8
6
7 INSERT INTO teacher(`id`, `name`) VALUES (1, '秦老师');
8
9 CREATE TABLE `student` (
10  `id` INT(10) NOT NULL,
11  `name` VARCHAR(30) DEFAULT NULL,
12  `tid` INT(10) DEFAULT NULL,
13  PRIMARY KEY (`id`),
14  KEY `fk tid` (`tid`),
15  CONSTRAINT `fk tid` FOREIGN KEY (`tid`) REFERENCES `teacher` (`id`)

```

```
16 ) ENGINE=INNODB DEFAULT CHARSET=utf8
17
18
19 INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('1', '小明', '1');
20 INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('2', '小红', '1');
21 INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('3', '小张', '1');
22 INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('4', '小李', '1');
23 INSERT INTO `student` (`id`, `name`, `tid`) VALUES ('5', '小王', '1');
```

测试环境搭建

1. 导入lombok
2. 新建实体类 Teacher, Student
3. 建立Mapper接口
4. 建立Mapper.XML文件
5. 在核心配置文件中绑定注册我们的Mapper接口或者文件! 【方式很多, 随心选】
6. 测试查询是否能够成功!

按照查询嵌套处理

```
1 <!--
2     思路:
3         1. 查询所有的学生信息
4         2. 根据查询出来的学生的tid, 寻找对应的老师!    子查询
5     -->
6
7 <select id="getStudent" resultMap="StudentTeacher">
8     select * from student
9 </select>
10
11 <resultMap id="StudentTeacher" type="Student">
12     <result property="id" column="id"/>
13     <result property="name" column="name"/>
14     <!--复杂的属性, 我们需要单独处理 对象: association 集合: collection -->
15     <association property="teacher" column="tid" javaType="Teacher" select="getTeacher"/>
16 </resultMap>
17
18 <select id="getTeacher" resultType="Teacher">
19     select * from teacher where id = #{id}
20 </select>
```

```
Student(id=1, name=小明, teacher=Teacher(id=1, name=秦老师))
Student(id=2, name=小红, teacher=Teacher(id=1, name=秦老师))
Student(id=3, name=小张, teacher=Teacher(id=1, name=秦老师))
Student(id=4, name=小李, teacher=Teacher(id=1, name=秦老师))
Student(id=5, name=小王, teacher=Teacher(id=1, name=秦老师))
```

按照结果嵌套处理

```
1 <!--按照结果嵌套处理-->
2 <select id="getStudent2" resultMap="StudentTeacher2">
3     select s.id sid,s.name sname,t.name tname
4     from student s,teacher t
5     where s.tid = t.id;
6 </select>
7
8 <resultMap id="StudentTeacher2" type="Student">
9     <result property="id" column="sid"/>
10    <result property="name" column="sname"/>
11    <association property="teacher" javaType="Teacher">
12        <result property="name" column="tname"/>
13    </association>
14 </resultMap>
```

回顾Mysql 多对一查询方式：

- 子查询
- 联表查询

一对多处理

比如：一个老师拥有多个学生！对于老师而言，就是一对多的关系！

环境搭建

1. 环境搭建，和刚才一样

实体类

```
1 @Data
2 public class Student {
3
4     private int id;
5     private String name;
6     private int tid;
7
8 }
```

```
1 @Data
2 public class Teacher {
3     private int id;
4     private String name;
5
6     //一个老师拥有多个学生
```

↑

```

7     private List<Student> students;
8 }

```

按照结果嵌套处理

```

1     <!--按结果嵌套查询-->
2     <select id="getTeacher" resultMap="TeacherStudent">
3         select s.id sid, s.name sname, t.name tname,t.id tid
4         from student s,teacher t
5         where s.tid = t.id and t.id = #{tid}
6     </select>
7
8     <resultMap id="TeacherStudent" type="Teacher">
9         <result property="id" column="tid"/>
10        <result property="name" column="tname"/>
11        <!--复杂的属性，我们需要单独处理 对象： association 集合： collection
12        javaType="" 指定属性的类型！
13        集合中的泛型信息，我们使用ofType获取
14        -->
15        <collection property="students" ofType="Student">
16            <result property="id" column="sid"/>
17            <result property="name" column="sname"/>
18            <result property="tid" column="tid"/>
19        </collection>
20    </resultMap>

```

按照查询嵌套处理

```

1 <select id="getTeacher2" resultMap="TeacherStudent2">
2     select * from mybatis.teacher where id = #{tid}
3 </select>
4
5 <resultMap id="TeacherStudent2" type="Teacher">
6     <collection property="students" javaType="ArrayList" ofType="Student" select="getStudentId"
7 </resultMap>
8
9 <select id="getStudentByTeacherId" resultType="Student">
10     select * from mybatis.student where tid = #{tid}
11 </select>

```

小结

1. 关联 - association 【多对一】
2. 集合 - collection 【一对多】
3. javaType & ofType

- a. `JavaType` 用来指定实体类中属性的类型
- b. `ofType` 用来指定映射到List或者集合中的 pojo类型，泛型中的约束类型！

注意点：

- 保证SQL的可读性，尽量保证通俗易懂
- 注意一对多和多对一中，属性名和字段的问题！
- 如果问题不好排查错误，可以使用日志，建议使用 Log4j

慢SQL 1s 1000s

面试高频

- Mysql引擎
- InnoDB底层原理
- 索引
- 索引优化

动态 SQL

什么是动态SQL：动态SQL就是指根据不同的条件生成不同的SQL语句

利用动态 SQL 这一特性可以彻底摆脱这种痛苦。

```

1 动态 SQL 元素和 JSTL 或基于类似 XML 的文本处理器相似。在 MyBatis 之前的版本中，有很多元素需要花
2
3 if
4 choose (when, otherwise)
5 trim (where, set)
6 foreach

```

搭建环境

```

1 CREATE TABLE `blog` (
2   `id` varchar(50) NOT NULL COMMENT '博客id',
3   `title` varchar(100) NOT NULL COMMENT '博客标题',
4   `author` varchar(30) NOT NULL COMMENT '博客作者',
5   `create_time` datetime NOT NULL COMMENT '创建时间',
6   `views` int(30) NOT NULL COMMENT '浏览量'
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8

```

↑

创建一个基础工程

1. 导包
2. 编写配置文件
3. 编写实体类

```
1 @Data
```

```
2 public class Blog {
3     private int id;
4     private String title;
5     private String author;
6     private Date createTime;
7     private int views;
8
9
10 }
```

4. 编写实体类对应Mapper接口 和 Mapper.XML文件

IF

```
1 <select id="queryBlogIF" parameterType="map" resultType="blog">
2     select * from mybatis.blog where 1=1
3     <if test="title != null">
4         and title = #{title}
5     </if>
6     <if test="author != null">
7         and author = #{author}
8     </if>
9 </select>
```

choose (when, otherwise)

```
1     <select id="queryBlogChoose" parameterType="map" resultType="blog">
2         select * from mybatis.blog
3         <where>
4             <choose>
5                 <when test="title != null">
6                     title = #{title}
7                 </when>
8                 <when test="author != null">
9                     and author = #{author}
10                </when>
11                <otherwise>
12                    and views = #{views}
13                </otherwise>
14            </choose>
15        </where>
16    </select>
```

trim (where,set)

```
1 select * from mybatis.blog
2 <where>
```

```

3   <if test="title != null">
4       title = #{title}
5   </if>
6   <if test="author != null">
7       and author = #{author}
8   </if>
9 </where>

```

```

1 <update id="updateBlog" parameterType="map">
2     update mybatis.blog
3     <set>
4         <if test="title != null">
5             title = #{title},
6         </if>
7         <if test="author != null">
8             author = #{author}
9         </if>
10    </set>
11    where id = #{id}
12 </update>

```

所谓的动态SQL，本质还是SQL语句，只是我们可以在SQL层面，去执行一个逻辑代码

if, where, set, choose, when

SQL片段

有的时候，我们可能会将一些功能的部分抽取出来，方便复用！

1. 使用SQL标签抽取公共的部分

```

1 <sql id="if-title-author">
2     <if test="title != null">
3         title = #{title}
4     </if>
5     <if test="author != null">
6         and author = #{author}
7     </if>
8 </sql>

```

2. 在需要使用地方使用Include标签引用即可

```

1 <select id="queryBlogIF" parameterType="map" resultType="blog">
2     select * from mybatis.blog
3     <where>
4         <include refid="if-title-author"></include>
5     </where>
6 </select>

```

↑

注意事项：

- 最好基于单表来定义SQL片段！
- 不要存在where标签

Foreach

```
1 select * from user where 1=1 and
2   <foreach item="id" collection="ids"
3     open="(" separator="or" close=")">
4     #{id}
5   </foreach>
6 (id=1 or id=2 or id=3)
```

动态 SQL 的另外一个常用的操作需求是对一个集合进行遍历，通常是在构建 IN 条件语句的时候。比如：

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
  <foreach item="item" index="index" collection="list"
    open="(" separator="," close=")">
    #{item}
  </foreach>
</select>
```

(1, 2, 3, 4, 5)

`foreach` 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的集合项（item）和索引（index）变量。它允许你指定开头与结尾的字符串以及在迭代结果之间放置分隔符。这个元素是很智能的，因此它不会偶然地附加多余的分隔符。

注意 你可以将任何可迭代对象（如 List、Set 等）、Map 对象或者数组对象传递给 `foreach` 作为集合参数。当使用可迭代对象或者数组时，index 是当前迭代的次数，item 的值是本次迭代获取的元素。当使用 Map 对象（或者 Map.Entry 对象的集合）时，index 是键，item 是值。

到此我们已经完成了涉及 XML 配置文件和 XML 映射文件的讨论。下一章将详细探讨 Java API，这样就能提高已创建的映射文件的利用效率。

	id	title	author	create_time	views
1	1	Mybatis如此简单	狂神说	2019-10-01 16:33:38	9999
2	2	Java如此简单	狂神说	2019-10-01 16:33:38	1000
3	3	Java如此简单2	狂神说2	2019-10-01 16:33:38	9999
4	4	微服务如此简单	狂神说	2019-10-01 16:33:38	9999

```
1 <select id="queryBlogForeach" parameterType="map" resultType="Blog">
2   select * from blog
3   <where>
4     <foreach item="id" collection="ids"
5       open="and (" separator="or" close=")">
6       id = #{id}
7     </foreach>
8   </where>
9 </select>
```

```
1 @Test
2 public void queryBlogForeach(){
```

```
3      SqlSession sqlSession = MybatisUtils.getSqlSession();
4      BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
5      Map map = new HashMap();
6      List<String> ids = new ArrayList<String>();
7      ids.add("bce7cc1ca483454eb925c1c0e6037d5f");
8      ids.add("3a0b7bbb3faa4bbaad1dcc151fb29769");
9      ids.add("c6c0616e8f82403cb336696a6f6729af");
10     map.put("ids",ids);
11     mapper.queryBlogForeach(map);
12     sqlSession.close();
13 }
```

动态SQL就是在拼接SQL语句，我们只要保证SQL的正确性，按照SQL的格式，去排列组合就可以了。建议：

- 现在Mysql中写出完整的SQL,再对应的去修改成为我们的动态SQL实现通用即可！

缓存（了解）

简介

- 1 查询 ： 连接数据库 ， 耗资源！
- 2 一次查询的结果，给他暂存在一个可以直接取到的地方！--> 内存 ： 缓存
- 3 我们再次查询相同数据的时候，直接走缓存，就不用走数据库了

1. 什么是缓存 [Cache]?

- 存在内存中的临时数据。
- 将用户经常查询的数据放在缓存（内存）中，用户去查询数据就不用从磁盘上(关系型数据库数据文件)查询，从缓存中查询，从而提高查询效率，解决了高并发系统的性能问题。

2. 为什么使用缓存？

- 减少和数据库的交互次数，减少系统开销，提高系统效率。

3. 什么样的数据能使用缓存？

- 经常查询并且不经常改变的数据。【可以使用缓存】

Mybatis缓存

- MyBatis包含一个非常强大的查询缓存特性，它可以非常方便地定制和配置缓存。缓存可以极大的提升查询效率。
- MyBatis系统中默认定义了两级缓存：**一级缓存**和**二级缓存**
 - 默认情况下，只有一级缓存开启。（SqlSession级别的缓存，也称为本地缓存）
 - 二级缓存需要手动开启和配置，他是基于namespace级别的缓存。

- 为了提高扩展性，MyBatis定义了缓存接口Cache。我们可以通过实现Cache接口来自定义二级缓存

一级缓存

- 一级缓存也叫本地缓存： SqlSession
 - 与数据库同一次会话期间查询到的数据会放在本地缓存中。
 - 以后如果需要获取相同的数据，直接从缓存中拿，没必要再去查询数据库；

测试步骤：

- 开启日志！
- 测试在一个Session中查询两次相同记录
- 查看日志输出

```

PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 811760110.
==> Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 狂神, 123456
<== Total: 1
User(id=1, name=狂神, pwd=123456)
=====
User(id=1, name=狂神, pwd=123456)
true
Closing JDBC Connection [com.mysql.jdbc.jdbc4Connection@306279ee]
Returned connection 811760110 to pool.
Process finished with exit code 0
  
```

SqlSession

缓存失效的情况：

- 查询不同的东西
- 增删改操作，可能会改变原来的数据，所以必定会刷新缓存！

```

Tests passed: 1 of 1 test - 1 s 481 ms
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 811760110.
==> Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 狂神, 123456
<== Total: 1
User(id=1, name=狂神, pwd=123456)
==> Preparing: update mybatis.user set name=?,pwd=? where id = ?;
==> Parameters: aaaa(String), bbbbbb(String), 2(Integer)
<== Updates: 1
=====
==> Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<== Columns: id, name, pwd
<== Row: 1, 狂神, 123456
<== Total: 1
User(id=1, name=狂神, pwd=123456)
false
Closing JDBC Connection [com.mysql.jdbc.jdbc4Connection@306279ee]
Returned connection 811760110 to pool.
  
```

- 查询不同的Mapper.xml

4. 手动清理缓存!

```
SqlSession sqlSession = MybatisUtils.getSqlSession();
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
User user = mapper.getUserByIdName( id: 1, name: "zs1");
System.out.println(user);
sqlSession.clearCache();
User user2 = mapper.getUserByIdName( id: 1, name: "zs1");
System.out.println(user2);
```

小结：一级缓存默认是开启的，只在一次SqlSession中有效，也就是拿到连接到关闭连接这个区间段！一级缓存就是一个Map。

二级缓存

- 二级缓存也叫全局缓存，一级缓存作用域太低了，所以诞生了二级缓存
- 基于namespace级别的缓存，一个名称空间，对应一个二级缓存；
- 工作机制
 - 一个会话查询一条数据，这个数据就会被放在当前会话的一级缓存中；
 - 如果当前会话关闭了，这个会话对应的一级缓存就没了；但是我们想要的是，会话关闭了，一级缓存中的数据被保存到二级缓存中；
 - 新的会话查询信息，就可以从二级缓存中获取内容；
 - 不同的mapper查出的数据会放在自己对应的缓存（map）中；

步骤：

1. 开启全局缓存

```
1 <!--显示的开启全局缓存-->
2 <setting name="cacheEnabled" value="true"/>
```

2. 在要使用二级缓存的Mapper中开启

```
1 <!--在当前Mapper.xml中使用二级缓存-->
2 <cache/>
```

2. 也可以自定义参数

```
1 <!--在当前Mapper.xml中使用二级缓存-->
2 <cache eviction="FIFO"
3         flushInterval="60000"
4         size="512"
5         readOnly="true"/>
```

3. 测试

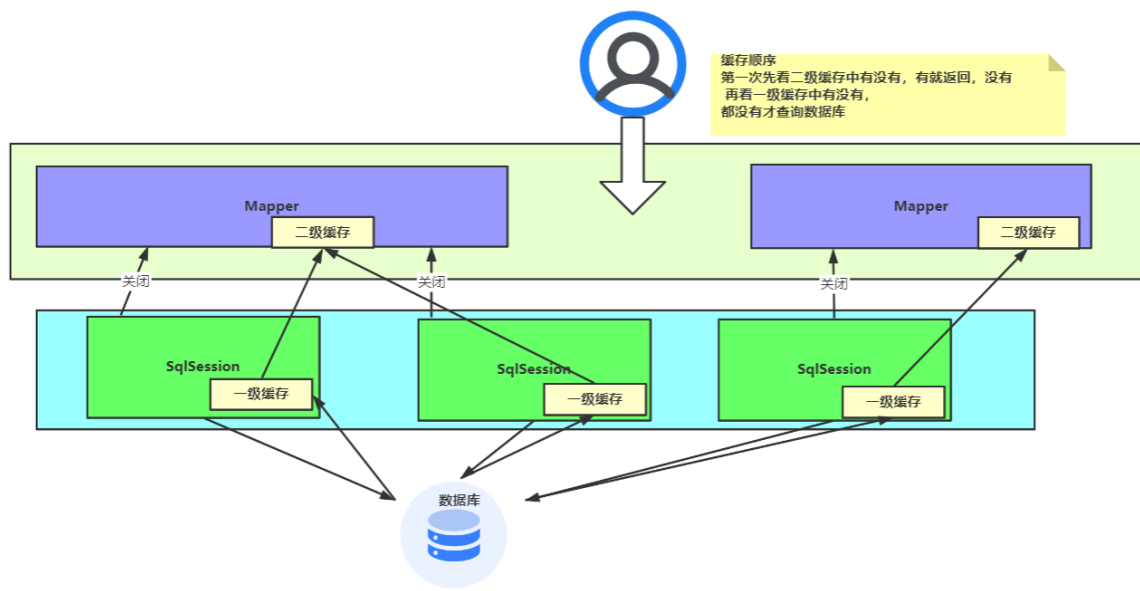
- a. 问题:我们需要将实体类序列化! 否则就会报错!

```
1 Caused by: java.io.NotSerializableException: com.kuang.pojo.User
```

小结：

- 只要开启了二级缓存，在同一个Mapper下就有效
- 所有的数据都会先放在一级缓存中；
- 只有当会话提交，或者关闭的时候，才会提交到二级缓存中！

缓存原理



自定义缓存-ehcache

- 1 Ehcache是一种广泛使用的开源Java分布式缓存。主要面向通用缓存

要在程序中使用ehcache，先要导包！

```
1 <!-- https://mvnrepository.com/artifact/org.mybatis.caches/mybatis-ehcache -->
2 <dependency>
3   <groupId>org.mybatis.caches</groupId>
4   <artifactId>mybatis-ehcache</artifactId>
5   <version>1.1.0</version>
6 </dependency>
```

在mapper中指定使用我们的ehcache缓存实现！

```

1 <!--在当前Mapper.xml中使用二级缓存-->
2 <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

```

ehcache.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3         xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
4         updateCheck="false">
5     <!--
6         diskStore: 为缓存路径，ehcache分为内存和磁盘两级，此属性定义磁盘的缓存位置。参数解释如下
7         user.home - 用户主目录
8         user.dir   - 用户当前工作目录
9         java.io.tmpdir - 默认临时文件路径
10    -->
11    <diskStore path="./tmpdir/Tmp_EhCache"/>
12
13    <defaultCache
14        eternal="false"
15        maxElementsInMemory="10000"
16        overflowToDisk="false"
17        diskPersistent="false"
18        timeToIdleSeconds="1800"
19        timeToLiveSeconds="259200"
20        memoryStoreEvictionPolicy="LRU"/>
21
22    <cache
23        name="cloud_user"
24        eternal="false"
25        maxElementsInMemory="5000"
26        overflowToDisk="false"
27        diskPersistent="false"
28        timeToIdleSeconds="1800"
29        timeToLiveSeconds="1800"
30        memoryStoreEvictionPolicy="LRU"/>
31    <!--
32        defaultCache: 默认缓存策略，当ehcache找不到定义的缓存时，则使用这个缓存策略。只能定义一个
33    -->
34    <!--
35        name:缓存名称。
36        maxElementsInMemory:缓存最大数目
37        maxElementsOnDisk: 硬盘最大缓存个数。
38        eternal:对象是否永久有效，一旦设置了，timeout将不起作用。
39        overflowToDisk:是否保存到磁盘，当系统当时时
40        timeToIdleSeconds:设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用。
41        timeToLiveSeconds:设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。
42        diskPersistent: 是否缓存虚拟机重启期数据 Whether the disk store persists between restarts

```


```
43     diskSpoolBufferSizeMB: 这个参数设置DiskStore（磁盘缓存）的缓存区大小。默认是30MB。每个Ca
44     diskExpiryThreadIntervalSeconds: 磁盘失效线程运行时间间隔，默认是120秒。
45     memoryStoreEvictionPolicy: 当达到maxElementsInMemory限制时，Ehcache将会根据指定的策略去
46     clearOnFlush: 内存数量最大时是否清除。
47     memoryStoreEvictionPolicy:可选策略有：LRU（最近最少使用，默认策略）、FIFO（先进先出）、L
48     FIFO, first in first out, 这个是大家最熟的，先进先出。
49     LFU, Less Frequently Used, 就是上面例子中使用的策略，直白一点就是讲一直以来最少被使用的
50     LRU, Least Recently Used, 最近最少使用的，缓存的元素有一个时间戳，当缓存容量满了，而又需
51     -->
52
53 </ehcache>
```

Redis数据库来做缓存！ K-V

练习：29道练习题实战！




关注作者和知识库后续更新



青梅换了酒钱

关注



狂神说Java
跟着小狂神一起学Java

关注

推荐阅读

超市订单管理系统（SMBMS）

项目资源链接:系统功能架构数据库模型设计数据库SQL文件...

JavaWeb

Java Web 知识： web服务器 HTTP协议、Maven、JSP 、Ser...

Spring

spring 学习： 主要包括IOC、依赖注入、AOP

