MyBatis（测试）

- 三层架构
  - 界面层（controller）：servlet，jsp，html，使用SpringMVC
  - 业务逻辑（service）：接受界面层获取的数据，调用数据库，获取数据，使用spring
  - 数据访问层（DAO）：访问数据库，执行数据查询，使用mybatis
- JDBC缺点
  - 代码复用性低，实现简单的功能却需要更多代码
  - 查询结果Result需要自己封装为list
  - 业务代码和数据库操作混在一起
- mybatis框架
  - 主要特点
    - SQL mapper：sql映射
      - 将数据库表视为java对象
    - Data Access Objects（DAOs）
      - 数据访问，对数据库执行增删改查
  - 提供的功能
    - 创建Connection，Statement，Result的能力
    - 提供执行sql的能力
    - 提供循环sql，将sql结果转为java对象，自动封装sql结果
    - 提供关闭资源的能力
  - 开发人员的作用
    - 提供sql语句
  - 开发过程
    - 提供sql--->mybatis处理sql--->开发人员获取mybatis结果
- 使用mybatis
  - （1）创建Maven项目添加mysql驱动和mybatis依赖

```
 <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.6</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-
java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.25</version>
        </dependency>
```

  - （2）编写用于存放表中数据的类，类名建议与表名一致

```
public class User {
    private String password;
    private String name;
}
```

- （3）编写用于操作数据类的接口

```
public interface User_dao {
    public List<User> select();
}
```

- 在操作接口下创建管理SQL的xml文件，在其中配置操作接口
- xml文件名建议与接口名相同
- xml和操作接口在同一目录下

```
<?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">
<!-- sql映射文件所约束的标签dtd文件-->

<mapper namespace="org.mybatis.example.BlogMapper">

<!-- sql映射根标签，namespace可以自定义但需唯一，建议使用dao接口的全限定名称-
->

    <select id="selectBlog" resultType="Blog">
        select *
        from Blog
        where id = #{id}
  </select>

<!--
sql语句编写

<select>表示select语句，以此类推
id属性:执行该sql语句的唯一标识，尽量使用dao接口中的方法名
resultType属性：sql执行后得到的java类型，也就是存放数据的类，也就是User，以该
类的全名作为该属性


 -->

</mapper>
```

- 编写后的SQL映射xml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="User_dao">
    <select id="select" resultType="User">
        select *
        from user
        where name = 'clearlove'
  </select>
</mapper>
```

- 编写mybatis主配置xml文件
  - 该xml文件是用于件简化JDBC中数据库连接，比如Connection创建等

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!--environments代表所有数据库的配置信息，其中default为默认指定数据库，必须和某一个environment id相同-->
    <environments default="user">

        <!--environment代表单个数据库的配置信息，id为该环境唯一标识，可以以数据库名为id-->
        <environment id="user">

            <!--transactionManager代表事务提交方式，type="JDBC"代表和connection中的事务提交方式相同-->
            <transactionManager type="JDBC"/>

            <!--dataSource代表该数据库的基本信息配置如url等，type="POOLED"表示采用连接池-->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/user"/>
                <property name="username" value="root"/>
                <property name="password" value="123"/>
            </dataSource>
        </environment>
    </environments>

<!--mappers代表所有SQL映射文件位置指定-->
    <mappers>
        <!--mapper代表单个SQL映射文件位置指定，位置从项目编译的class目录下开始。
        比如一个类的全包名为com.java.User_dao,则其映射文件位置就是com/java/User_dao.xml(如果映射文件与dao接口在同一目录下的话)-->
        <mapper resource="User_dao.xml" />
    </mappers>
</configuration>
```

- 编写一个测试SQL语句使用的类

```
public class mybatis_test {
    public static void main(String[] args) throws IOException {
        //设置mybatis主配置文件路径
        String config="mybatis_config.xml";
        //读取mybatis主配置文件
        InputStream in = Resources.getResourceAsStream(config);
        //创建SqlSessionFactoryBuilder
        SqlSessionFactoryBuilder builder = new
SqlSessionFactoryBuilder();
        //通过SqlSessionFactoryBuilder指定主配置文件读取流创建
SqlSessionFactory
        SqlSessionFactory factory=builder.build(in);
        //（重要）创建SQL执行对象
        SqlSession session = factory.openSession();
        //指定需要执行的sql语句的id，id=namespace+.+sql语句的id
        String sqlid="User_dao"+"."+"select";
        //获取查询结果
        List<User> list = session.selectList(sqlid);
        //输出查询结果
        list.forEach(s-> System.out.println(s));
        //关闭sqlsession
        session.close();
    }
}
```

- 通过mybatis编写insert语句
  - （1）DAO接口添加insert方法

```
public int insert(User user);//返回int类型，代表这次多了多少条记录
```

  - （2）编写mapper文件

```
<insert id="insert" >
    insert into data(name,password) values(#{name},#{password})
</insert>
```

    - 其中#{name}是mybatis中特有的格式，之后sqlsession执行该语句时会从user类中找到相应的变量并填入其中
  - （3）编写insert测试类

```
//设置mybatis主配置文件路径
String config="mybatis_config.xml";
//读取mybatis主配置文件
InputStream in = Resources.getResourceAsStream(config);
//创建SqlSessionFactoryBuilder
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
//通过SqlSessionFactoryBuilder指定主配置文件读取流创建SqlSessionFactory
SqlSessionFactory factory=builder.build(in);
//（重要）创建SQL执行对象
SqlSession session = factory.openSession();
//指定需要执行的sql语句的id，id=namespace+.+sql语句的id
```

```
User user = new User("root", "zhangsan");
String sqlid="User_dao.insert";
session.insert(sqlid, user);
session.commit();
//关闭sqlsession
session.close();
```

- 将需要增加的记录以类的形式传入session.insert（）方法中
- **注意：session中的insert方法默认取消自动事务自动提交，所以DDL语句均需要手动提交**
- 对于DDL语句，每次完成后我们并不知道sql最后是什么样子，sql的执行对程序员是透明的，但我们可以开启日志功能以便观察
  - 在mybatis主配置文件中的根标签下添加

```
<settings>
       <setting name="logImpl" value="STDOUT_LOGGING"/>
   </settings>
```

    - logImpl是日志功能，STDOUT_LOGGING是输出到控制台上

Mybatis中的常用类

Resources

- 作用：负责读取主配置文件

```
InputStream in = Resources.getResourceAsStream(主配置文件路径);
```

SqlSessionFactoryBuilder

- 作用：建立SqlSessionFactory

```
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory=builder.build(in);
```

  - SqlSessionFactory build(InputStream inputStream, String environment)
    - 可以指定environment

SqlSessionFactory

- 比较重要，本身是一个接口，消耗资源较多，其实现类是DefaultSqlSessionFactory

  SqlSessionFactory 有六个方法创建 SqlSession 实例。通常来说，当你选择其中一个方法时，你需要考虑以下几点：

  - **事务处理**：你希望在 session 作用域中使用事务作用域，还是使用自动提交（auto-commit）？（对很多数据库和/或 JDBC 驱动来说，等同于关闭事务支持）
  - **数据库连接**：你希望 MyBatis 帮你从已配置的数据源获取连接，还是使用自己提供的连接？
  - **语句执行**：你希望 MyBatis 复用 PreparedStatement 和/或批量更新语句（包括插入语句和删除语句）吗？

  基于以上需求，有下列已重载的多个 openSession() 方法供使用。

```
SqlSession openSession()
SqlSession openSession(boolean autoCommit)
SqlSession openSession(Connection connection)
SqlSession openSession(TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType,
TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCommit)
SqlSession openSession(ExecutorType execType, Connection connection)
```

默认的 openSession() 方法没有参数，它会创建具备如下特性的 SqlSession：

- 事务作用域将会开启（也就是不自动提交）。
- 将由当前环境配置的 DataSource 实例中获取 Connection 对象。
- 事务隔离级别将会使用驱动或数据源的默认设置。
- 预处理语句不会被复用，也不会批量处理更新。

相信你已经能从方法签名中知道这些方法的区别。向 `autoCommit` 可选参数传递 `true` 值即可开启自动提交功能。若要使用自己的 `Connection` 实例，传递一个 `Connection` 实例给 `connection` 参数即可。**注意，我们没有提供同时设置 `Connection` 和 `autoCommit` 的方法，这是因为 MyBatis 会依据传入的 Connection 来决定是否启用 autoCommit。**对于事务隔离级别，MyBatis 使用了一个 Java 枚举包装器来表示，称为 `TransactionIsolationLevel`，事务隔离级别支持 JDBC 的五个隔离级别（`NONE`、`READ_UNCOMMITTED`、`READ_COMMITTED`、`REPEATABLE_READ` 和 `SERIALIZABLE`），并且与预期的行为一致。

你可能对 `ExecutorType` 参数感到陌生。这个枚举类型定义了三个值:

- `ExecutorType.SIMPLE`：该类型的执行器没有特别的行为。它为每个语句的执行创建一个新的预处理语句。
- `ExecutorType.REUSE`：该类型的执行器会复用预处理语句。
- `ExecutorType.BATCH`：该类型的执行器会批量执行所有更新语句，如果 SELECT 在多个更新中间执行，将在必要时将多条更新语句分隔开来，以方便理解。

SqlSession

- 正如之前所提到的，SqlSession 在 MyBatis 中是非常强大的一个类。它包含了所有执行语句、提交或回滚事务以及获取映射器实例的方法。

SqlSession 类的方法超过了 20 个，为了方便理解，我们将它们分成几种组别。

语句执行方法

- 这些方法被用来执行定义在 SQL 映射 XML 文件中的 SELECT、INSERT、UPDATE 和 DELETE 语句。你可以通过名字快速了解它们的作用，每一方法都接受语句的 ID 以及参数对象，参数可以是原始类型（支持自动装箱或包装类）、JavaBean、POJO 或 Map。

```
<T> T selectOne(String statement, Object parameter)

<E> List<E> selectList(String statement, Object parameter)

<K,V> Map<K,V> selectMap(String statement, Object parameter, String
mapKey)
```

- selectOne 和 selectList 的不同仅仅是 selectOne 必须返回一个对象或 null 值。如果返回值多于一个，就会抛出异常。如果你不知道返回对象会有多少，请使用 selectList。如

果需要查看某个对象是否存在，最好的办法是查询一个 count 值（0 或 1）。
    - selectMap 稍微特殊一点，它会将返回对象的其中一个属性作为 key 值，将对象作为 value 值，从而将多个结果集转为 Map 类型值。由于并不是所有语句都需要参数，所以这些方法都具有一个不需要参数的重载形式。
- insert、update 以及 delete 方法返回的值表示受该语句影响的行数。

```
int insert(String statement)

int update(String statement)

int delete(String statement)
```

最后，还有 select 方法的两个个高级版本，它们允许你限制返回行数的范围，通常在数据集非常庞大的情形下使用。

```
<E> List<E> selectList (String statement, Object parameter, RowBounds
rowBounds)
<K,V> Map<K,V> selectMap(String statement, Object parameter, String
mapKey, RowBounds rowbounds)
```

RowBounds 参数会告诉 MyBatis 略过指定数量的记录，并限制返回结果的数量。
RowBounds 类的 offset 和 limit 值只有在构造函数时才能传入，其它时候是不能修改的。

```
int offset = 100;
int limit = 25;
RowBounds rowBounds = new RowBounds(offset, limit);
```

- Insert语句执行后获取具有自增属性的主键的最新数据
    - 比如使用id代表注册网站的总人数，每注册一个增加一个，为了获取最新注册的人的id可以采用last_insert_id()函数

      ```
      select last_insert_id()
      ```

        - 但是这个方法有缺陷，如果使用单行的insert语句插入了多条数据的话，该函数值返回是这条insert语句第一次插入数据的id，比如表中有2条数据，最新的id为2，但如果我一次性插入三条记录，last_insert_id()返回的将是3,可以这样说last_insert_id()本质上是记录了insert的执行次数。
        - 同时还可以配合selectKey标签来获取这个值

          ```
          <insert id="insert">
          <selectKey keyProperty="id" resultType="int" order="AFTER">
              select last_insert_id()
          </selectKey>
              insert into user(name,password,age) values(#{name},#
          {password},#{age})
              </insert>
          ```

        - keyProperty：selectKey标签将值以User类的方式返回，同时将值存放在keyProperty所指的变量中（这个变量在User类中）。
        - resulitType：id的类型
        - order：指该标签中的内容在sql语句执行前还是执行后，AFTER为执行后。
    - 但是对于某些大型数据库而言，两个表甚至两个库中会有重复的id，这时可以使用UUID作为记录的唯一标识

```
UUID uuid=UUID.raandomUUID();
```

        - UUID是由36个字母或数字或划线组成的全球唯一的标识

事务控制方法

有四个方法用来控制事务作用域。当然，如果你已经设置了自动提交或你使用了外部事务管理器，这些方法就没什么作用了。然而，如果你正在使用由 Connection 实例控制的 JDBC 事务管理器，那么这四个方法就会派上用场：

```
void commit()
void commit(boolean force)
void rollback()
void rollback(boolean force)
```

默认情况下 MyBatis 不会自动提交事务，除非它侦测到调用了插入、更新或删除方法改变了数据库。如果你没有使用这些方法提交修改，那么你可以在 commit 和 rollback 方法参数中传入 true 值，来保证事务被正常提交（注意，在自动提交模式或者使用了外部事务管理器的情况下，设置 force 值对 session 无效）。**大部分情况下你无需调用 rollback()，因为 MyBatis 会在你没有调用 commit 时替你完成回滚操作。**不过，当你要在一个可能多次提交或回滚的 session 中详细控制事务，回滚操作就派上用场了。

本地缓存

Mybatis 使用到了两种缓存：本地缓存（local cache）和二级缓存（second level cache）。

每当一个新 session 被创建，MyBatis 就会创建一个与之相关联的本地缓存。**任何在 session 执行过的查询结果都会被保存在本地缓存中**，所以，当再次执行参数相同的相同查询时，就不需要实际查询数据库了。本地缓存将会在做出修改、事务提交或回滚，以及关闭 session 时清空。

默认情况下，**本地缓存数据的生命周期等同于整个 session 的周期。**由于缓存会被用来解决循环引用问题和加快重复嵌套查询的速度，所以无法将其完全禁用。但是你可以通过设置 localCacheScope=STATEMENT 来只在语句执行时使用缓存。

注意，如果 localCacheScope 被设置为 SESSION，对于某个对象，MyBatis 将返回在本地缓存中唯一对象的引用。对返回的对象（例如 list）做出的任何修改将会影响本地缓存的内容，进而将会影响到在本次 session 中从缓存返回的值。因此，不要对 MyBatis 所返回的对象作出更改，以防后患。

你可以随时调用以下方法来清空本地缓存：

```
void clearCache()
```

确保 SqlSession 被关闭

```
void close()
```

对于你打开的任何 session，你都要保证它们被妥善关闭，这很重要。保证妥善关闭的最佳代码模式是这样的：

```
SqlSession session = sqlSessionFactory.openSession();
try (SqlSession session = sqlSessionFactory.openSession()) {
    // 假设下面三行代码是你的业务逻辑
    session.insert(...);
    session.update(...);
    session.delete(...);
    session.commit();
}
```

- 有由于SqlSession本身是非线程安全的，只有通过以上逻辑才能保证SqlSession是线程安全的

编写工具类简化开发

- 对于mybatis而言仍然有很多步骤是没有得到复用的，所以可以编写一个工具类封装可以复用的过程

```
public class mybatis_util {
    static SqlSession getsqlSession(String config)
    {
       try {
           InputStream in = Resources.getResourceAsStream(config);
           SqlSessionFactoryBuilder builder = new
SqlSessionFactoryBuilder();
           return builder.build(in).openSession();
       } catch (IOException e) {
           e.printStackTrace();
           return null;
       }
     }
}
```

- 编写测试类

```
void util_test()
    {
        SqlSession sqlSession =
mybatis_util.getsqlSession("mybatis_config.xml");
        String sqlid="User_dao.select";
        List<User> list = sqlSession.selectList(sqlid);
        list.forEach(u-> System.out.println(u));
        sqlSession.close();
    }
```

使用动态代理简化DAO的执行

- 到目前为止User_dao类并没有任何的使用，没有参与任何程序的执行，我们可以编写一个类实现该接口，以达到对mybatis中sql语句执行过程的封装
  - 传统DAO执行
    - （1）编写一个类实现User_dao类并在其中封装sql语句调用的方法

```
public class dao_impl implements User_dao{
    @Override
    public List<User> select() {
        SqlSession sqlSession =
mybatis_util.getsqlSession("mybatis_config.xml");
        String sqlid="User_dao.select";
        List<User> list = sqlSession.selectList(sqlid);
        sqlSession.close();
        return list;
    }

    @Override
    public int insert(User user) {
        SqlSession sqlSession =
mybatis_util.getsqlSession("mybatis_config.xml");
        String sqlid="User_dao.select";
        int count = sqlSession.insert(sqlid,user);
        sqlSession.commit();
        sqlSession.close();
        return count;
    }
}
```

    - (2)在一个测试类中调用该类中的方法（不再演示）
- 而传统的DAO层执行仍存在代码没有复用的情况，所以我们可以借助动态代理的方式优化DAO的执行
  - 动态代理
    - 原理：
      - 若预先知道需要调用哪个方法，则sqlid=dao的执行类的全类名+.+方法名，这样就不必把不同的sql用不同的方法封装
      - 而mybatis可以在每次调用dao的执行方法时内部实现一个代理类并完成sql语句的执行，也就是mybatis在内部帮我们实现了dao_impl这个类，而且更加高效简洁
    - 实现
      - （1）mapper.xml文件中sql语句的id必须与DAO执行类中的方法名一致
      - （2）调用动态代理实现DAO执行

```
@Test
    void proxy_test()
    {
        SqlSession sqlSession =
mybatis_util.getsqlSession("mybatis_config.xml");
        User_dao dao =
sqlSession.getMapper(User_dao.class);
        List<User> list = dao.selectall();
        list.forEach(u-> System.out.println(u));
    }
```

- 此时的mapper文件就像是一个对于DAO执行接口中方法实现的说明，而mybatis看到mapper这份说明后为我们实现了DAO执行接口。
- 顺便一提，mybatis动态代理底层采用的是jdk动态代理（Spring有讲解，可以自己实现一下）

- mybatis中的mapper传值
  - parameterType（用于告诉mybatis需要传值的类型）
    - 对于每次查询其实并不需要整个User类作为参数，也就是不需要对数据进行封装，比如只需要name即可，所以是否可以实现一个方法只传递一个name就能查询到数据库?
    - （1）编写DAO接口

```
public interface User_dao {
    public List<User> selectall();
    public int insert(User user);
    public User selectbyname(String n);

}
```

    - （2）编写User_dao.xml

```
<select id="selectbyname" resultType="User"
parameterType="java.lang.String">
        select name,password from data where name=#{n}
</select>

# {变量名}可以作为占位符，当mybatis碰到第一个#时会将其后面括号内部的都变成？，而mybatis底层采用的是preparestatement。
```

    - parameterType用来告知mybatis该数据的类型，不写的话会默认为传参的类型，如果编写的parameterType与实际需要的类型不符合时会报错。
    - 如果存在多个参数可以采用参数命名方式（这也是mybatis建议的）
      - 在DAO接口中添加一下方法

```
public User
selectbynameandpassword(@Param("name")String n,
@Param("password")String p);
```

        - @Param("在mapper中该参数的名字")
      - 在mapper中

```
<select id="selectbynameandpassword" resultType="User">
        select name,password from data where name=#
{name} and password=#{password}
</select>
```

- 同时多个参数也可以采用对象传递值
  - 只需要在mapper中指定即可

```
<select id="selectbynameandpassword"
resultType="User">
        select name,password from data where name=#
{name,javaType=java.lang.String,jdbcType=varchar}
and password=#
{password,javaType=java.lang.String,jdbcType=varcha
r}
</select>
具体格式：#{属性名,javaType=java类型全名称,jdbcType=数据库中类型名称}
```

    - 但这种方式开发中不常用，通常省略javaType和jdbcType，只有#{属性名称}
- 多个参数可以按顺序传值

```
<select id="selectbynameandpassword" resultType="User">
        select name,password from data where name=#
{arg0} and password=#{arg1}
</select>
```

  - 在mybatis3.3以及之前的版本中使用#{0},#{1}，而3.4之后采用以上方式，传值顺序以方法的参数从左到右决定，这种方式不常用，仅作了解
- 通过map集合传递多参数

```
public User syap(Map<String, Object> map);
```

```
<select id="syap" resultType="User" >
        select name,password from data where name=#
{key1} and password=#{key2}
    </select>
```

  - key1和key2代表的是map集合中key值，mybatis可以根据sql中的#{key1}中的key1自动获得map中的值
- ${}和#{}的区别
  - ${}采用的是statement，而#{}采用的是preparstatement，使用时statement需要添加单引号
    - 优缺点对比
      - "#{}":速度更快，更加安全，不够灵活
      - "${}":速度较慢，可以SQL注入，但是可以实现排序等函数，比较灵活

- （3）编写测试单元

```
@Test
void test_para(){
        SqlSession sqlSession =
mybatis_util.getsqlSession("mybatis_config.xml");
        User_dao dao = sqlSession.getMapper(User_dao.class);
        User user = dao.selectbyname("zhangsan");
        System.out.println(user);
}
```

- ResultType
  - 作为泛型传递，可以是java类型的全名，也可以是mybatis中的别名，比如Integer的别名是int，可以通过帮助文档查询
  - 除此以外，还可以自定义别名，对于一些自定义的类，通过typeAliases标签定义在主配置文件下的根标签下添加

    ```
    <typeAliases>
            <typeAlias type="User" alias="u"/>
    </typeAliases>
    ```

    - 除此以外还可以通过package标签起别名

      ```
      <typeAliases>
              <package/  name="包名">
      </typeAliases>
      ```

      - 这样表示该包下所有类的别名是类名（不区分大小写），比如com.java.User就可以表示为user或User等
      - 但不太建议使用别名，容易引起歧义
  - map作为返回值
    - select查询结果返回的数据封装在map中，key是字段名，value是字段值，但是map只能返回一条查询结果，但是可以通过以下代码使返回map集合同时返回多条语句，就是通过list封装map集合
    - DAO执行接口

      ```
      public List<HashMap<Object,Object>> selectformap();
      ```

    - mapperxml文件

      ```
      <select id="selectformap"
      resultType="java.util.HashMap">
          select name,age,password
          from test
      </select>
      ```

    - 测试类

```
void test03()
    {
        SqlSession sqlSession =
util.getsqlSession("mybatis_config.xml");
        DAO mapper =
sqlSession.getMapper(org.mybatis_test.DAO.class);
        List<HashMap<Object, Object>> maps =
mapper.selectformap();
        for(HashMap<Object, Object> m:maps)
        {
            for(Map.Entry<Object, Object>
entry:m.entrySet())
            {

 System.out.println("key:"+entry.getKey()+"
value:"+entry.getValue());
            }
        }
    }
```

- 在开发中可能存在User类中变量名称和数据库中字段名不同的情况，对于这种情况，可以自定义ResultMap
- mapper文件配置

```
<resultMap id="map" type="org.mybatis_test.User">
        <id column="name" property="username" />
        <result column="password" property="userpassword"
/>
        <result column="age" property="userage"  />
    </resultMap>
<!--自定义resultMap,指定字段名中值赋值到类中的成员变量,resultMap可以重复利用-->

    <select id="selectreturnbymap" resultMap="map">
        select name,age,password
        from test
    </select>
```

- DAO执行接口方法设置

```
public List<User> selectreturnbymap();
```

- 测试类

```
void test02()
{
    SqlSession sqlSession =
util.getsqlSession("mybatis_config.xml");
    DAO mapper =
sqlSession.getMapper(org.mybatis_test.DAO.class);
    List<User> list= mapper.selectreturnbymap();
    list.forEach(u-> System.out.println(u));
}
```

- **注意：User类中必须要有set方法（最好也有get方法），而且必须要有无参构造方法（血与泪的教训），原理大概是mybatis在创建User时，只会通过set方法为成员变量赋值，所以必须有set方法，同时mybatis在创建User类时会调用无参构造，但如果编写了有参构造方法，但没有编写无参构造，mybatis就会报错**

- 使用collection标签完成多标查询
  - 业务场景1：两个表，一个表存放用户数据，一个存放用户订单，用户订单表含有外键,此时需要查询一个用户下的所有订单
  - 实现步骤
    - 编写用户类

```
package org.example.User2;

import java.util.List;

public class User2 {
    private int id;
    private String name;
    private String  password;
    private List<orders> o;
//省略构造方法和set方法等
}
```

      - 使用list集合封装该用户的订单
    - 编写订单类

```
package org.example.User2;

public class orders {
    private int userid;
    private String name;
    private int price;
    private int id;
//省略构造方法和set方法等
}
```

    - 编写DAO接口

```
public interface DAO2 {
    public User2 select(@Param("id") int id);
}
```

- 编写mapper文件

```
<resultMap id="map" type="org.example.User2.User2">
          <id column="uid" property="id"/>
      <result column="uname" property="name"/>
      <collection ofType="org.example.User2.orders"
property="o">
          <id column="oid" property="id"/>
          <result column="oname" property="name"/>
          <result column="oprice" property="price"/>
      </collection>
  </resultMap>

  <select id="select" resultMap="map">
      select
          u.id uid,u.name uname,o.id oid,o.name
oname,o.price oprice
      from user u inner join orders o on
          o.userid=u.id
      where u.id=#{id};
</select>
```

  - 返回类型采用resultMap，自定义resultMap中使用collection标签告诉mybatis如何为订单类传递值,ofType为该集合的泛型，property和resultMap中其他result标签汇中的property一样
  - 这里的sql语句采用内连接查询，只会显示符合条件的记录，如果使用外连接可以显示用户表的所有
- 编写测试类

```
void t2()
{
    DAO2 dao = util.getDAO("mybatis_config.xml");
    User2 u = dao.select(2);
    List<orders> o = u.getO();
    for(orders order : o)
    {
        System.out.println(order);
    }
}
```

- 业务场景2：两个表，一个表存放用户数据，一个存放用户订单，用户订单表含有外键,此时需要查询订单以及订单的用户的所有信息
  - 实现步骤
    - 编写订单类

```
public class orders {
    private String name;
    private int price;
    private int id;
    private User2 u;
//省略构造方法和set方法等
}
```

- 编写User2类

```
public class User2 {
    private int id;
    private String name;
    private String  password;

    //省略构造方法和set方法等
}
```

- DAO接口方法

```
public List<User2> select();
```

- mapper文件

```
<resultMap id="map" type="org.example.User2.orders">
         <id column="oid" property="id"/>
      <result column="oname" property="name"/>
      <result column="oprice" property="price"/>
      <association property="u"
javaType="org.example.User2.User2">
          <id column="uid" property="id"/>
          <result column="uname" property="name"/>
      </association>
  </resultMap>

  <select id="select" resultMap="map">
      select
          u.id uid,u.name uname,o.id oid,o.name
oname,o.price oprice
      from user u inner join orders o on
          o.userid=u.id

</select>
```

  - association标签指的是类标签，在association中编写自定义类的注入方式

- 基于业务场景1的优化
  - 使用嵌套select语句完成

```
<resultMap id="map" type="org.example.User2.orders">
         <id column="oid" property="id"/>
      <result column="oname" property="name"/>
      <result column="price" property="price"/>
      <association property="u"
javaType="org.example.User2.User2"
select="org.example.DAO.DAO2.selectuser" column="uid"/>
  </resultMap>

<!--select属性指定引入的查询语句的结果集，由namespace+select语句的标签id定位到该语句，其中column代表是传入select属性中的select语句的值，-->
```

```
<!--先查询出订单的信息包括用户的ID-->
  <select id="select" resultMap="map">
      select o.id oid,o.name oname,o.price price,o.userid
uid
      from orders o
</select>


<!--通过用户id查询出用户信息-->
  <select id="selectuser" parameterType="int"
resultType="org.example.User2.User2">
      select u.name uname,u.password upassword from user u
where u.id=#{id}
  </select>
```

- 这样可以增加代码的复用成度。

动态SQL

- mybatis中内置了一些标签可以实现动态SQL的功能
  - if标签

```
<select id="select" resultType="org.example.User.User">
        select *
        from test where
        <if test="条件">
            满足条件后拼接的sql字符串
        </if>
        <if test="条件">
            满足条件后拼接的sql字符串
        </if>
  </select>
```

  - 自上而下的执行，满足一个if条件拼接一个sql语句

```
<select id="select" resultType="org.example.User.User">
        select *
        from test where
        <if test="age>0">
            age>#{age}
        </if>
        <if test="name!=null">
            or name=#{name}
        </if>
  </select>
```

  - 注意：在第一个if标签之后的if标签如果满足需要注意拼接时是否会出现语法错误（比如是否需要添加or或and），同时存在如果第一个if不满足，第二个if满足那么添加的or和and是否会产生语法错误，而这些问题在where中都可以解决
  - where标签

```
<select id="select" resultType="org.example.User.User">
        select *
        from test
            <where>
                多个if标签块
            </where>

  </select>
```

- where标签可以替代sql语句中的where，当有if符合条件时where标签会为sql语句后自动添加where和需要拼接的if语句中的sql语句，同时会舍去多余的or和and（如果在第二个if满足条件时而第一个不满足时）

```
<select id="select" resultType="org.example.User.User">
      select *
      from test
          <where>
              <if test="age>0">
                  age>#{age}
              </if>
              <if test="name!=null">
                  or name=#{name}
              </if>
          </where>

</select>
```

- foreach标签
  - 对于sql语句中某些时候需要大量添加拼接sql语句，比如in，可以使用foreach循环添加

```
<select id="selectforeach" resultType="org.example.User.User">
    select *from test where name in
    <foreach collection="循环集合类型，array为数组" item="代表java中以下foreach中的x，可以自定义" open="开始时添加的字符串" close="结束时添加的字符串" separator="每次循环之间拼接的字符串中间的分隔符">
        循环拼接的字符串
    </foreach>
</select>
```

    - 仿照java中的foreach对比

```
for(List x:list)
{
    循环内容
}
```

    - 实例

```
<select id="selectforeach" resultType="org.example.User.User">
    select *from test where name in
    <foreach collection="list" item="u" open="(" close=")"
separator=",">
        #{u.name}
    </foreach>
</select>
```

- 最后实现的sql语句为：select *from test where name in (? , ? , ?)
- 其中如果集合中存放的是User对象，可以采用以上写法，使用"."来访问对象中的成员
- foreach本身比较灵活，除了collection之外的属性都可以不指定，可以自己添加，比如以下实例

```
<select id="selectforeach" resultType="org.example.User.User">
    select *from test where name in
    (
    <foreach collection="list">
        1=1,
    </foreach>
    )
</select>
```

- 最后结果：select *from test where name in (1=1, 1=1, 1=1,)

- set标签
  - set用于update中，当我们只需要更新一个字段值时，而其他的字段值保持不变，可能会出现以下的bug

    ```
    User u=new User("name",123);
    ```

    ```
    <update id="update" parameterType="User">
        update test set name = #{name} ,password=#{password},
    age=#{age} where id=#{id}
    </update>
    ```

    - 如果User类中只有name和id进行了赋值操作，其他的均为空,那么这条记录除了name和id其他全部都会被赋值为null,所以需要使用set标签进行有选择的更新。
  - 代码如下

    ```
    <update id="update" parameterType="User">
        update test
        <set>
            <if test="name!=null">
                name = #{name},
            </if>

            <if test="password!=null">
                password=#{password},
            </if>
    ```

```
            <if test="age!=null">
                age=#{age},
            </if>
        </set>
        where id=#{id}
    </update>
```

- SQL代码片段
  - 将重复的代码封装定义声明在xml文件中，提高代码复用，比如以下的语句

```
<select id="usesql" resultType="org.example.User.User">
        select name,password,age from test where name =#{name}
    </select>
```

    - 如果"select name,password,age from test where name ="这一片段使用多次则可以使用sql标签在mapper下定义封装

```
<sql id="headofselectbyname">
        select name,password,age from test where name =
</sql>
```

    - id为该复用代码块唯一指定标识
    - select标签编写

```
<select id="usesql" resultType="org.example.User.User">
    <include refid="headofselectbyname"/>
     #{name}
</select>
```

      - refid表示引用的sql代码片段id
      - 最后的sql为：select name,password,age from test where name = ?

Mybatis主配置文件

- transactionManger：事务的提交类型
  - JDBC：底层调用JDBC中的Connection对象的commit和rollback
  - MANAGED：把mybatis的事务处理委托给其他容器，比如服务器或者框架（spring）
- dataSource：数据源
  - 在java中规定时限了javax.sql.DataSource的都是数据源，数据源表示connection对象
  - dataSource本身是一个接口，接口中定义了getconnection方法，返回connection对象
  - 在mybatis中有三个实现类分别为：PooledDataSource,UnpooledDataSource,MysqlDataSource
  - type属性
    - POOLED
      - 采用PooledDataSource实现类，使用连接池管理connection，每当需要执行sql时从连接池中获取connection，使用后放回
      - PooledDataSource和UnpooledDataSource都是mybatis中自带的
    - UNPOOLED

        - 不使用连接池，采用UnpooledDataSource，每次执行sql语句时，先创立连接，使用完后关闭connection，实际类似于传统JDBC的执行的封装，单元测试时使用较多
    - JNDI
        - java命名和目录服务（windows注册表）
        - 使用较少，较为复杂
- 采用properties文件保存JDBC中必要的信息（url，用户名等）
    - 在Resource目录下创建JDBC.properties文件
    - 在xml文件中指定配置文件

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

    - xml文件通过${key}来获取值

```
<dataSource type="POOLED">
        <property name="driver" value="${jdbc.mysql.driver}"/>
        <property name="url" value="${jdbc.mysql.url}"/>
        <property name="username" value="${jdbc.mysql.username}"/>
        <property name="password" value="${jdbc.mysql.password}"/>
      </dataSource>
```

- 简化mapper文件的配置
    - 开发过程中可能会有很多mapper文件
    - 在主配置文件中的mappers标签下

```
<package name="org.example.DAO"/>
```

        - 这种格式会一次性读取name包下所有mapper
        - 这种格式的要求是mapper文件的名字需要和DAO接口的名称一样
- PageHepler插件
    - 某些时候需要查询得到的数据量很大，对数据的分页有很有必要，PageHepler插件可以做到
        - 步骤
            - 在pom.xml中添加依赖
            - 在主配置文件中添加plugin标签

```
<plugins>
  <plugin
interceptor="com.github.pagehelper.PageInterceptor">
</plugin>
</plugins>
```

                - 在环境前添加该标签
            - 编写测试类

```
DAO dao=util.getDAO("mybatis_config.xml");
PageHelper.startPage(1,1);
//PageHelper.startPage(1,1)启动分页，第一个参数代表第几页，第二个参数代表每一页多少条数据
List<User> list = dao.selectall();
for(User u:list)
{
    System.out.println(u);
}
```

- **Mybatis缓存机制**
  - 缓存机制：大体和计算机组成原理相同
    - 第一次访问先找缓存，找到了就直接返回，没找到就到数据库中找，并同时在缓存中添加
    - 不同点：如果发生crud（增删改），缓存全部清空
  - mybatis缓存
    - 一级缓存
      - 默认开启，作用域在sqlsession中，也就是创建sqlsession时为其默认开辟空间作为缓存
    - 二级缓存
      - 需要手动开启，其作用域在mapper中
      - 步骤
        - (1)在主配置文件中添加

```
<settings>
    <setting name="cacheEnabled" value="value"/>
</settings>
```

        - (2)在需要二级缓存的mapper文件中添加cache标签

```
<cache></cache>
```

        - (3)所有数据实体类均需要继承可序列化接口

```
public class User implements Serializable{

}
```

- ORM概念
  - ORM:对象关系映射
    - java将数据以类的方式暂时存放在内存中，程序结束后释放内存
    - 数据库将数据以表的方式存储硬盘中，永久保存
    - 而将java对象中的数据转为数据库中表的数据，这一过程称为持久化，而反过来叫瞬时化，完成这两个操作的过程叫映射，而mybatis框架是一个具有良好的ORM框架