

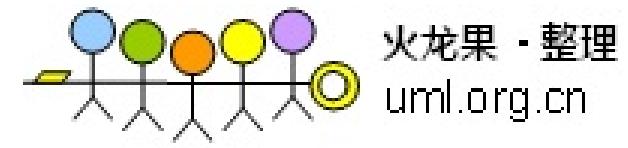
1. Mybatis 入门

从一个 jdbc 程序开始

```
public static void main(String[] args) {
    Connection connection = null ;
    PreparedStatement preparedStatement = null ;
    ResultSet resultSet = null ;

    try {
        // 加载数据库驱动
        Class.forName("com.mysql.jdbc.Driver");

        // 通过驱动管理类获取数据库链接
        connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8", "root", "mysql");
        // 定义 sql 语句 ? 表示占位符
        String sql = "select * from user where username = ?";
        // 获取预处理 statement
        preparedStatement = connection.prepareStatement(sql);
        // 设置参数，第一个参数为 sql 语句中参数的序号（从 1 开始），第二个参数为设置的参数值
        preparedStatement.setString(1, "王五");
        // 向数据库发出 sql 执行查询，查询出结果集
        resultSet = preparedStatement.executeQuery();
        // 遍历查询结果集
        while (resultSet.next()) {
            System.out.println(resultSet.getString("id") + " "
" + resultSet.getString("username"));
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        // 释放资源
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {
```



```
// TODOAuto-generated catch block
e.printStackTrace();

}

if (preparedStatement!= null ){

    try  {

        preparedStatement.close();

    } catch (SQLException e) {

        // TODOAuto-generated catch block

        e.printStackTrace();

    }

}

if (connection!= null ){

    try  {

        connection.close();

    } catch (SQLException e) {

        // TODOAuto-generated catch block

        e.printStackTrace();

    }

}

}
```

上边使用 jdbc 的原始方法（未经封装）实现了查询数据库表记录的操作。

jdbc 操作步骤总结如下：

- 1、加载数据库驱动
 - 2、创建并获取数据库链接
 - 3、创建 jdbc statement 对象
 - 4、设置 sql 语句
 - 5、设置 sql 语句中的参数 (使用 preparedStatement)
 - 6、通过 statement 执行 sql 并获取结果
 - 7、对 sql 执行结果进行解析处理
 - 8、释放资源 (resultSet、preparedstatement 、 connection)

jdbc 问题总结如下：

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

- 2、Sql 语句写在代码中造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。
- 3、向 sql 语句传参数麻烦，因为 sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应。
- 4、对结果集解析麻烦，sql 变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 pojo 对象解析比较方便。

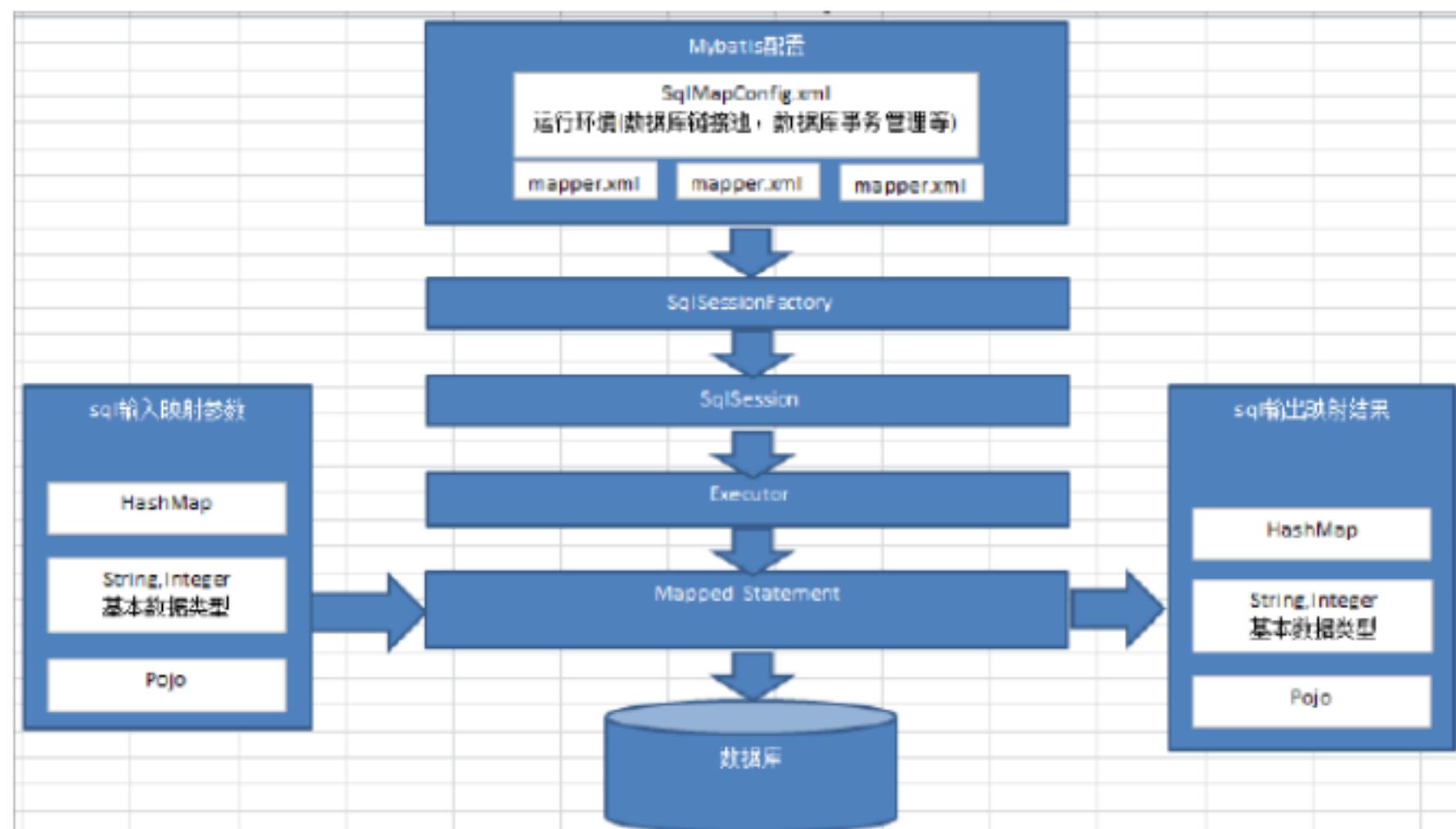
MyBatis 介绍

MyBatis 本是 apache 的一个开源项目 iBatis, 2010 年这个项目由 apache software foundation 迁移到了 google code，并且改名为 MyBatis。

MyBatis 是一个优秀的持久层框架，它对 jdbc 的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建 connection 、创建 statement 、手动设置参数、结果集检索等 jdbc 繁杂的过程代码。

Mybatis 通过 xml 或注解的方式将要执行的 statement 配置起来，并通过 java 对象和 statement 中的 sql 进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射成 java 对象并返回。

Mybatis 架构



1、mybatis 配置

SqlMapConfig.xml ,此文件作为 mybatis 的全局配置文件 , 配置了 mybatis 的运行环境等信息。 mapper.xml 文件即 sql 映射文件 , 文件中配置了操作数据库的 sql 语句。此文件需要在 SqlMapConfig.xml 中加载。

- 2、通过 mybatis 环境等配置信息构造 SqlSessionFactory 即会话工厂
- 3、由会话工厂创建 sqlSession 即会话 , 操作数据库需要通过 sqlSession 进行。
- 4、mybatis 底层自定义了 Executor 接口操作数据库 , Executor 接口有两个实现 , 一个是基本实现、一个是缓存实现。
- 5、Mapped Statement 也是 mybatis 一个底层对象 , 它包装了 mybatis 配置信息及 sql 映射信息等。 mapper.xml 文件中一个 sql 对应一个 Mapped Statement 对象 , sql 的 id 即是 Mapped statement 的 id。
- 6、Mapped Statement 对 sql 执行输入参数进行定义 , 包括 HashMap 、基本类型、 pojo ,Executor 通过 Mapped Statement 在执行 sql 前将输入的 java 对象映射至 sql 中 , 输入参数映射就是 jdbc 编程中对 preparedStatement 设置参数。
- 7、Mapped Statement 对 sql 执行输出结果进行定义 , 包括 HashMap 、基本类型、 pojo ,Executor 通过 Mapped Statement 在执行 sql 后将输出结果映射至 java 对象中 , 输出结果映射过程相当于 jdbc 编程中对结果的解析处理过程。

Mybatis 第一个程序

第一步：创建 JAVA 工程

使用 eclipse 创建 java 工程 , jdk 使用 1.6.

第二步：加入 jar 包

加入 mybatis 核心包、依赖包、数据驱动包。



第三步：log4j.properties

在 classpath 下创建 log4j.properties 如下：

```
# Global logging configuration
log4j.rootLogger      =DEBUG, stdout
# Console output...
log4j.appender.stdout    =org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout   =org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern      =%5p [%t] - %m%n
```

mybatis 默认使用 log4j 作为输出日志信息。

第四步：SqlMapConfig.xml

在 classpath 下创建 SqlMapConfig.xml , 如下：

```
<?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      >
    <!--<properties resource=""></properties> -->
    <environments    default  ="development"    >
        <environment   id  ="development"    >
            <transactionManager    type  ="JDBC" />
            <dataSource   type  ="POOLED">
                <property   name="driver"    value  ="com.mysql.jdbc.Driver"           />

                <property   name="url"     value  ="jdbc:mysql://localhost:3306/mybatis?charac
terEncoding=utf-8"           />
                <property   name="username"  value  ="root"    />
                <property   name="password"  value  ="mysql"    />
            </ dataSource  >
        </ environment   >
    </ environments   >
</ configuration      >
```

SqlMapConfig.xml 是 mybatis 核心配置文件，上边文件的配置内容为数据源、事务管理。

第五步：po 类

Po 类作为 mybatis 进行 sql 映射使用， po 类通常与数据库表对应， User.java 如下：

```
public class User {  
    private int id;  
    private String username; // 用户姓名  
    private String sex; // 性别  
    private Date birthday; // 出生日期  
    private String address; // 地址  
    private String detail; // 详细信息  
    private float score; // 成绩  
    get/set .....}
```

第六步：sql 映射文件

在 classpath 下的 sqlmap 目录下创建 sql 映射文件 User.xml：

```
<?xml version = "1.0" encoding = "UTF-8" ?>  
<!DOCTYPE mapper  
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >  
<mapper namespace = "test" >  
<!-- 根据 id 获取用户信息 -->  
    <select id = "selectUserById" parameterType = "int" resultType = "cn.itcast.mybatis.po.User" >  
        select * from user where id = #{id}  
    </select>  
<!-- 获取用户列表 -->  
    <select id = "selectUserList" resultType = "cn.itcast.mybatis.po.User" >  
        select * from user  
    </select>  
<!-- 添加用户 -->  
    <insert id = "insertUser" parameterType = "cn.itcast.mybatis.po.User" >  
        insert into user(username,birthday,sex,address,detail,score)  
        values(#{username},#{birthday},#{sex},#{address},#{detail},#{score});  
    </insert>  
<!-- 更新用户 -->  
    <update id = "updateUser" parameterType = "cn.itcast.mybatis.po.User" >
```

```
update user set
username=#{username},birthday=#{birthday},sex=#{sex},address=#{address}
,detail=#{detail},score=#{score}
    where id=#{id}
</ update >
<!-- 刪除用戶 -->
<delete id ="deleteUser"      parameterType   ="cn.itcast.mybatis.po.User"          >
    delete from user where id=#{id}
</ delete >
</ mapper >
```

namespace : 命名空间，用于隔离 sql 语句，后面会讲另一层非常重要的作用。
parameterType : 定义输入到 sql 中的映射类型， #{id} 表示使用 preparedstatement 设置占位符号并将输入变量 id 传到 sql 。
resultType : 定义结果映射类型。

第七步：将 User.xml 添加在 SqlMapConfig.xml

在 SqlMapConfig.xml 中添加 mappers 如下：

```
<mappers >
    <mapper resource   ="sqlmap/user.xml"      />
</ mappers >
```

这里即告诉 mybatis Sql 映射文件在哪里。

第八步：程序编写

查询

```
/*
 * 第一个 mybatis 程序
 *
 * @author Thinkpad
 *
 */
public class Mybatis_select {
    public static void main(String[] args) throws IOException {
        //mybatis 配置文件
        String resource = "sqlMapConfig.xml";
    }
}
```

```

InputStream inputStream = Resources.getResourceAsStream(resource);
// 使用 SqlSessionFactoryBuilder 创建 sessionFactory
SqlSessionFactory sqlSessionFactory =
new SqlSessionFactoryBuilder()
    .build(inputStream);
// 通过 session 工厂获取一个 SqlSession , sqlSession 中包括了对数据库操作的
sql 方法
SqlSession session = sqlSessionFactory.openSession();
try {
    // 通过 sqlSession 调用 selectOne 方法获取一条结果集
    // 参数 1：指定定义的 statement 的 id, 参数 2：指定向 statement 中传递的参
数
    User user = session.selectOne("test.selectUserById", 1);
    System.out.println(user);
} finally {
    session.close();
}

}
}

```

添加

```

public class Mybatis_insert {
    public static void main(String[] args) throws IOException {
        //mybatis 配置文件
        String resource = "sqlMapConfig.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        // 使用 SqlSessionFactoryBuilder 创建 sessionFactory
        SqlSessionFactory sqlSessionFactory =
new SqlSessionFactoryBuilder()
    .build(inputStream);
        // 通过 session 工厂获取一个 SqlSession , sqlSession 中包括了对数据库操作的
sql 方法
        SqlSession session = sqlSessionFactory.openSession();
        try {
            User user = newUser();
            user.setUsername("张三");
            user.setBirthday(new Date());
            user.setSex("1");
            user.setAddress("北京市");
            user.setDetail("好同志");
            user.setScore(99.8f);
        }
    }
}

```

```

        session.insert("test.insertUser", user);
        session.commit();
    } finally {
        session.close();
    }

}
}

```

主键返回

通过修改 sql 映射文件，可以将 mysql 自增主键返回：

```

<insert id="insertUser" parameterType="cn.itcast.mybatisplus.po.User" >
    <!-- selectKey 将主键返回，需要再返回 -->

    <selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer" >
        select LAST_INSERT_ID()
    </selectKey >
    insert into user(username,birthday,sex,address,detail,score)

    values("#{username},#{birthday},#{sex},#{address},#{detail},#{score}");
</insert>

```

添加 selectKey 实现将主键返回

keyProperty: 返回的主键存储在 pojo 中的哪个属性

order : selectKey 的执行顺序，是相对与 insert 语句来说，由于 mysql 的自增原理执行完 insert 语句之后才将主键生成，所以这里 selectKey 的执行顺序为 after

resultType: 返回的主键是什么类型

LAST_INSERT_ID(): 是 mysql 的函数

Oracle 可采用序列完成：

首先自定义一个序列且于生成主键， selectKey 使用如下：

```

<selectKey resultType="java.lang.Integer" order="BEFORE"
keyProperty="id">
    SELECT 自定义序列 .NEXTVAL FROM DUAL
</selectKey>

```

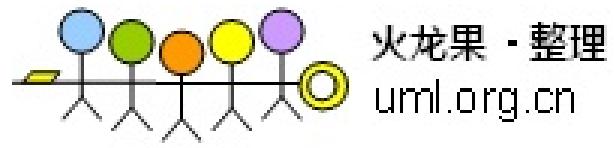
注意这里使用的 order 是 “ BEFORE ”

删除

```

public class Mybatis_delete {
    public static void main(String[] args) throws IOException {
        //mybatis 配置文件
    }
}

```



```
String resource = "sqlMapConfig.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
// 使用 SqlSessionFactoryBuilder 创建 sessionFactory
SqlSessionFactory sqlSessionFactory =
new SqlSessionFactoryBuilder()
    .build(inputStream);
// 通过 session 工厂获取一个 SqlSession ,sqlsession 中包括了对数据库操作的
sql 方法
SqlSession session = sqlSessionFactory.openSession();
try {
    session.delete("test.deleteUser", 4);
    session.commit();
} finally {
    session.close();
}
}
```

修改

```
public class Mybatis_update {  
    public static void main(String[] args) throws IOException {  
        // mybatis 配置文件  
        String resource = "sqlMapConfig.xml";  
        InputStream inputStream = Resources.getResourceAsStream(resource);  
        // 使用 SqlSessionFactoryBuilder 创建 sessionFactory  
        SqlSessionFactory sqlSessionFactory =  
            new SqlSessionFactoryBuilder()  
                .build(inputStream);  
        // 通过 session 工厂获取一个 SqlSession , sqlSession 中包括了对数据库操作的  
        // sql 方法  
        SqlSession session = sqlSessionFactory.openSession();  
        try {  
            User user = newUser();  
            user.setId(4);  
            user.setUsername("李四");  
            user.setBirthday(new Date());  
            user.setSex("1");  
            user.setAddress("北京市");  
            user.setDetail("好同志");  
            user.setScore(99.8f);  
            session.update("test.updateUser", user);  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            session.close();  
        }  
    }  
}
```

```
        session.commit();
    } finally {
        session.close();
    }
}
```

步骤总结：

- 1、创建 SqlSessionFactory
- 2、通过 SqlSessionFactory 创建 SqlSession
- 3、通过 sqlSession 执行数据库操作
- 4、调用 session.commit() 提交事务
- 5、调用 session.close() 关闭会话

Mybatis 解决 jdbc 编程的问题

- 1、数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。
解决：在 SqlMapConfig.xml 中配置数据链接池，使用连接池管理数据库链接。
- 2、Sql 语句写在代码中造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。
解决：将 Sql 语句配置在 XXXXmapper.xml 文件中与 java 代码分离。
- 3、向 sql 语句传参数麻烦，因为 sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应。
解决：Mybatis 自动将 java 对象映射至 sql 语句。
- 4、对结果集解析麻烦，sql 变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 pojo 对象解析比较方便。
解决：Mybatis 自动将 sql 执行结果映射至 java 对象。

与 hibernate 不同

Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句，不过 mybatis 可以通过 XML 或注解方式灵活配置要运行的 sql 语句，并将 java

对象和 sql 语句映射生成最终执行的 sql , 最后将 sql 执行的结果再映射生成 java 对象。

Mybatis 学习门槛低 , 简单易学 , 程序员直接编写原生态 sql , 可严格控制 sql 执行性能 , 灵活性高 , 非常适合对关系数据模型要求不高的软件开发 , 例如互联网软件、企业运营类软件等 , 因为这类软件需求变化频繁 , 一但需求变化要求成果输出迅速。但是灵活的前提是 mybatis 无法做到数据库无关性 , 如果需要实现支持多种数据库的软件则需要自定义多套 sql 映射文件 , 工作量大。

Hibernate 对象 / 关系映射能力强 , 数据库无关性好 , 对于关系模型要求高的软件 (例如需求固定的定制化软件) 如果用 hibernate 开发可以节省很多代码 , 提高效率。但是 Hibernate 的学习门槛高 , 要精通门槛更高 , 而且怎么设计 O/R 映射 , 在性能和对象模型之间如何权衡 , 以及怎样用好 Hibernate 需要具有很强的经验和能力才行。

总之 , 按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构 , 所以框架只有适合才是最好。

2. SqlSession

SqlSession 中封装了对数据库的 sql 操作 , 如 : 查询、插入、更新、删除等。通过 SqlSessionFactory 创建 SqlSession , 而 SqlSessionFactory 是通过 SqlSessionFactoryBuilder 进行创建。

SqlSessionFactoryBuilder

SqlSessionFacoty 是通过 SqlSessionFactoryBuilder 进行创建 , SqlSessionFactoryBuilder 只用于创建 SqlSessionFactory , 可以当成一个工具类 , 在使用时随时拿来使用不需要特殊处理为共享对象。

SqlSessionFactory

SqlSessionFactory 是一个接口 , 接口中定义了 openSession 的不同方式 , SqlSessionFactory 一旦创建后可以重复使用 , 实际应用时通常设计为单例模式。

SqlSession

SqlSession 是一个接口 , 默认使用 DefaultSqlSession 实现类 , sqlSession 中定义了数据库操作。

执行过程如下 :

1、 加载数据源等配置信息

Environment environment = configuration.getEnvironment();
2、 创建数据库链接
3、 创建事务对象
4、 创建 Executor , SqlSession所有操作都是通过 Executor 完成 , mybatis 源码如下：

```
if (ExecutorType.BATCH== executorType) {  
    executor = newBatchExecutor( this , transaction);  
} elseif (ExecutorType.REUSE== executorType) {  
    executor = new ReuseExecutor( this , transaction);  
} else {  
    executor = new SimpleExecutor( this , transaction);  
}  
if ( cacheEnabled ) {  
    executor = new CachingExecutor(executor, autoCommit);  
}
```

5、 SqlSession的实现类即 DefaultSqlSession , 此对象中对操作数据库实质上用的是 Executor

结论：每个线程都应该有它自己的 SqlSession 实例。 SqlSession 的实例不能共享使用，它也是线程不安全的。 因此最佳的范围是请求或方法范围。 绝对不能将 SqlSession 实例的引用放在一个类的静态字段甚至是实例字段中。

3. Namespace 的作用 (重要)

命名空间除了对 sql 进行隔离 , mybatis 中对命名空间有特殊的作用 , 用于定义 mapper 接口地址。

问题 :

没有使用接口编程 , java 是面向接口编程语言 , 对数据库的操作应该定义一些操作接口 , 如 : 用户添加、用户删除、用户查询等 , 调用 dao 接口完成数据库操作。

解决 :

```
publicinterface UserDao {  
    public User getUserById( int id) throws Exception;  
    publicvoid insertUser(User user) throws Exception;  
}
```

```
public class UserDaoImpl implements UserDao {  
  
    public UserDaoImpl(SqlSessionFactory sqlSessionFactory){  
        this.setSqlSessionFactory(sqlSessionFactory);  
    }  
  
    private SqlSessionFactory sqlSessionFactory ;  
    @Override  
    public User getUserById( int id) throws Exception {  
        SqlSession session = sqlSessionFactory.openSession();  
        User user = null ;  
        try {  
            // 通过 sqlsession 调用 selectOne 方法获取一条结果集  
            // 参数 1：指定定义的 statement 的 id, 参数 2：指定向 statement 中传递的参  
数  
            user = session.selectOne("selectUserById" , 1);  
            System.out.println(user);  
            // 获取 List  
            List<User> list = session.selectList("selectUserList" );  
            System.out.println(list);  
  
        } finally {  
            session.close();  
        }  
        return user;  
    }  
  
    @Override  
    public void insertUser(User user) throws Exception {  
        SqlSession session = sqlSessionFactory.openSession();  
        try {  
            session.insert("insertUser" , user);  
            session.commit();  
        } finally {  
            session.close();  
        }  
    }  
  
    public SqlSessionFactory getSqlSessionFactory() {  
        return sqlSessionFactory ;  
    }  
    public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory)  
{
```

```
        this .sqlSessionFactory = sqlSessionFactory;
    }
}
```

问题：

第一个例子中，在访问 sql 映射文件中定义的 sql 时需要调用 sqlSession 的 selectOne 方法， 并将 sql 的位置（命名空间 +id）和参数传递到 selectOne 方法中，且第一个参数是一个长长的字符串， 第二个参数是一个 object 对象， 这对于程序编写有很大的不方便， 很多问题无法在编译阶段发现。

虽然上边对提出的面向接口编程问题进行解决，但是 dao 实现方法中仍然是调用 sqlSession 的 selectOne 方法， 重复代码多。

改为 mapper 接口实现：

第一步：定义 mapper.xml

Mapper.xml 文件不变还用原来的。

第二步：定义 mapper 接口

```
/*
 *  用户管理 mapper
 *  @author Thinkpad
 *
 */
publicinterface UserMapper {
    public User selectUserById( int id) throws Exception;
    public List<User> selectUserList() throws Exception;
    publicvoid insertUser(User user) throws Exception;
    publicvoid updateUser(User user) throws Exception;
    publicvoid deleteUser( int id) throws Exception;
}
```

接口定义有如下特点：

- 1、Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同

- 2、Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同
- 3、Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同

第三步：修改 namespace

Mapper.xml 映射文件中的 namespace 改为如下：

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

修改后 namespace 即是 mapper 接口的地址。

第四步：通过 mapper 接口调用 statement

```
public class UserMapperTest extends TestCase {  
  
    private SqlSessionFactory sqlSessionFactory;  
  
    protected void setUp() throws Exception {  
        // mybatis 配置文件  
        String resource = "sqlMapConfig.xml";  
        InputStream inputStream = Resources.getResourceAsStream(resource);  
        // 使用 SqlSessionFactoryBuilder 创建 sessionFactory  
        sqlSessionFactory = new  
        SqlSessionFactoryBuilder().build(inputStream);  
    }  
  
    public void testSelectUserById() throws Exception {  
        // 获取 session  
        SqlSession session = sqlSessionFactory.openSession();  
        // 获取 mapper 接口实例  
        UserMapper userMapper = session.getMapper(UserMapper.class);  
        // 通过 mapper 接口调用 statement  
        User user = userMapper.selectUserById(1);  
        System.out.println(user);  
        // 关闭 session  
        session.close();  
    }  
    public void testSelectUserList() throws Exception {  
    }
```

```
// 获取 session
SqlSession session = sqlSessionFactory.openSession();
// 获取 mapper 接口实例
UserMapper userMapper = session.getMapper(UserMapper.class);
// 通过 mapper 接口调用 statement
List<User>list = userMapper.selectUserList();
System.out.println(list);
// 关闭 session
session.close();

}

public void testInsertUser() throws Exception {
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获得 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 要添加的数据
    User user = newUser();
    user.setUsername("张三");
    user.setBirthday(new Date());
    user.setSex("1");
    user.setAddress("北京市");
    user.setDetail("好同志");
    user.setScore(99.8f);
    // 通过 mapper 接口添加用户
    userMapper.insertUser(user);
    // 提交
    session.commit();
    // 关闭 session
    session.close();
}

public void testUpdateUser() throws Exception {
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获得 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 要更新的数据
    User user = newUser();
    user.setId(7);
    user.setUsername("李四");
    user.setBirthday(new Date());
    user.setSex("1");
    user.setAddress("北京市");
    user.setDetail("好同志");
```

```
        user.setScore(99.8f);
        // 通过 mapper 接口调用 statement
        userMapper.updateUser(user);
        // 提交
        session.commit();
        // 关闭 session
        session.close();
    }
    public void testDeleteUser() throws Exception {
        // 获取 session
        SqlSession session = sqlSessionFactory.openSession();
        // 获限 mapper 接口实例
        UserMapper userMapper = session.getMapper(UserMapper.class);
        // 通过 mapper 接口删除用户
        userMapper.deleteUser(6);
        // 提交
        session.commit();
        // 关闭 session
        session.close();
    }
}
```

session.getMapper(UserMapper.class) 生成一个代理对象作为 UserMapper 的接口实现对象。

总结：

使用 mapper 接口不用写接口实现类即可完成数据库操作，简单方便，此方法为官方推荐方法。

使用 mapper 接口调用必须具备如下条件：

- 1、Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同
- 2、Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同
- 3、Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同
- 4、Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

至此，mybatis 的 mapper 包括 mapper.xml 和 mapper 接口两种文件。

4. SqlMapConfig.xml

配置内容

SqlMapConfig.xml 中配置的内容和顺序如下：

- properties (属性)
- settings (全局配置参数)
- typeAliases (类型别名)
- typeHandlers (类型处理器)
- objectFactory (对象工厂)
- plugins (插件)
- environments (环境集合属性对象)
 - environment (环境子属性对象)
 - transactionManager (事务管理)
 - dataSource (数据源)
- mappers (映射器)

properties (属性)

SqlMapConfig.xml 可以引用 java 属性文件中的配置信息如下：

在 classpath 下定义 db.properties 文件，

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

SqlMapConfig.xml 引用如下：

```
<properties resource = "db.properties"  />
<environments default = "development" >
  <environment id = "development" >
    <transactionManager type = "JDBC" />
    <dataSource type = "POOLED">
      <property name="driver" value = "${jdbc.driver}"      />
      <property name="url"   value = "${jdbc.url}"        />
      <property name="username" value = "${jdbc.username}" " />
      <property name="password" value = "${jdbc.password}" " />
```

```

</ dataSource >
</ environment >
</ environments >

```

settings(配置)

mybatis 全局配置参数，全局参数将会影响 mybatis 的运行行为。

详细参见“学习资料 /mybatis-settings.xlsx ”文件

Setting(设置)	Description(描述)	Valid Values(验证值组)	Default(默认值)
cacheEnabled	在全局范围内启用或禁用缓存配置任何映射器在此配置下。	true false	TRUE
lazyLoadingEnabled	在全局范围内启用或禁用延迟加载。禁用时，所有协会将被加载。	true false	TRUE
aggressiveLazyLoading	启用时，有延迟加载属性的对象将被完全加载后调用懒惰的任何属性。否则，每一个属性是按需加载。	true false	TRUE
multipleResultSetsEnabled	允许或不允许从一个单独的语句（需要兼容的驱动程序）要返回多个结果集。	true false	TRUE
useColumnLabel	使用列标签，而不是列名。在这方面，不同的驱动有不同的行为。参考驱动文档或测试两种方法来决定你的驱动程序的行为如何。	true false	TRUE
useGeneratedKeys	允许JDBC支持生成的键。兼容的驱动程序是必需的。此设置限制生成的键的使用。如果设置为true，一些驱动会不兼容，但仍然可以工作。	true false	FALSE

autoMappingBehavior	指定MyBatis应如何自动映射到数据库属性。NONE自动映射。PARTIAL只会自动映射结果没有的属性映射到定义在里面。FULL会自动映射的结果映射到任何剩余的（包含嵌套或其他）。	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	配置默认执行人。SIMPLE执行人确实没有什么特别的。REUSE执行器重用准备好的语句。BATCH执行器重用语句和批处理更新。	SIMPLE, REUSE, BATCH	SIMPLE
defaultStatementTimeout	设置驱动程序等待一个数据库响应的秒数。	Any positive integer	Not Set (null)
safeRowBoundsEnabled	允许使用嵌套的语句RowBounds。	true false	FALSE
mapUnderscoreToCamelCase	从经典的数据库列名A_COLUMN应用自动映射到驼峰标记的经典Java属性名aColumn。	true false	FALSE
localCacheScope	MyBatis的使用本地缓存，以防止循环引用，并加快反复的查询。默认情况下(SESSION)会话期间执行的所有查询缓存。如果localCacheScope=STATEMENT本地会话将就用于语句的执行，只是没有将数据共享之间的两个不同的语句相同的语句Session。	SESSION STATEMENT	SESSION
jdbcTypeForNull	指定为空值时，没有特定的JDBC类型的参数的JDBC类型。有些驱动需要指定的JDBC类型，但将就像NULL, VARCHAR或OTHER的工作与通用值。	JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER	OTHER

lazyLoadTriggerMethods	指定触发延迟加载的对象的方法。	A method name list separated by commas	equals, clone, hashCode, toString
defaultScriptingLanguage	指定所使用的语言默认为动态SQL生成。	A type alias or fully qualified class name.	org.apache.ibatis.scripting.xmltags.XMLDynamicLanguage
callSettersOnNulls	指定如果setter方法或地图的put方法时，将调用检查到的值是null。它是有用的，当你依靠Map.keySet()或null初始化。注意原语（如整型，布尔等）不会被设置为null。	true false	FALSE
logPrefix	指定的前缀字符串，MyBatis将会增加记录器的名称。	Any String	Not set
logImpl	指定MyBatis的日志实现使用。如果此设置不存在的记录的实现将自动查找。	SLF4J LOG4J LOG4J2 JDE_LOGGING COMMONS_LOGGING STUBBY noLogger	Not set
proxyFactory	指定代理工具，MyBatis将会使用创建增强功能的对象。	CGLIB JAVASSIST	

typeAliases(类型别名)

mybatis 支持别名：

别名	映射的类型
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	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

自定义别名：

在 **SqlMapConfig.xml** 中配置：

```
<typeAliases>
    <!-- 单个别名定义 -->
    <typeAlias alias="user" type="cn.itcast.mybatis.po.User" />
    <!-- 批量别名定义，扫描整个包下的类 -->
    <package name="cn.itcast.mybatis.po" />
</typeAliases>
```

typeHandlers (类型处理器)

类型处理器在将 java 类型和 sql 映射文件进行映射时使用，如下：

```
<select id="selectUserById" parameterType="int" resultType="user">
    select * from user where id = #{id}
</select>
```

parameterType：指定输入数据类型为 int，即向 statement 设置值

resultType：指定输出数据类型为自定义 User，即将 resultset 转为 java 对象

mybatis 自带的类型处理器基本上满足日常需求，不需要单独定义。

mybatis 支持类型处理器：

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	Boolean, boolean	任何兼容的布尔值
ByteTypeHandler	Byte, byte	任何兼容的数字或字节类型
ShortTypeHandler	Short, short	任何兼容的数字或短整型
IntegerTypeHandler	Integer, int	任何兼容的数字和整型
LongTypeHandler	Long, long	任何兼容的数字或长整型
FloatTypeHandler	Float, float	任何兼容的数字或单精度浮点型
DoubleTypeHandler	Double, double	任何兼容的数字或双精度浮点型
BigDecimalTypeHandler	BigDecimal	任何兼容的数字或十进制小数类型
StringTypeHandler	String	CHAR 和 VARCHAR 类型
ClobTypeHandler	String	CLOB 和 LONGVARCHAR

		类型
StringTypeHandler	String	NVARCHAR 和NCHAR 类型
NClobTypeHandler	String	NCLOB 类型
ByteArrayTypeHandler	byte[]	任何兼容的字节流类型
BlobTypeHandler	byte[]	BLOB 和 LONGVARBINARY 类型
DateTypeHandler	Date (java.util)	TIMESTAMP 类型
DateOnlyTypeHandler	Date (java.util)	DATE 类型
TimeOnlyTypeHandler	Date (java.util)	TIME 类型
SqlTimestampTypeHandler	Timestamp (java.sql)	TIMESTAMP 类型
SqlDateTypeHandler	Date (java.sql)	DATE 类型
SqlTimeTypeHandler	Time (java.sql)	TIME 类型
ObjectTypeHandler	任意	其他或未指定类型
EnumTypeHandler	Enumeration 类型	VARCHAR- 任何兼容的字符串类型，作为代码存储（而不是索引）。

mappers (映射器)

Mapper 配置的几种方法：

```
<mapper resource=" " />
```

使用相对于类路径的资源

如：`<mapper resource="sqlmap/user.xml" />`

```
<mapper url=" " />
```

使用完全限定路径

如：`<mapper url="file:///D:/workspace_springmvc/mybatis_01/config/sqlmap/user.xml" />`

```
<mapper class=" " />
```

使用 mapper 接口类路径

如：`<mapper class="cn.itcast.mybatis.mapper.UserMapper"/>`

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

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

注册指定包下的所有 mapper 接口

如： <package name="cn.itcast.mybatis.mapper"/>

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。

5. Mapper.xml

Mapper.xml 映射文件中定义了操作数据库的 sql，每个 sql 是一个 statement，映射文件是 mybatis 的核心。

parameterType(输入类型)

#{} 与 \${}

#{} 实现的是向 prepareStatement 中的预处理语句中设置参数值，sql 语句中 #{} 表示一个占位符即 ?。

```
<!-- 根据 id 查询用户信息 -->
<select id ="selectUserById"      parameterType   ="int"    resultType   ="user"  >
    select * from user where id = #{id}
</ select >
```

使用占位符 #{} 可以有效防止 sql 注入，在使用时不需要关心参数值的类型，mybatis 会根据参数值的类型调用不同的 statement 设置参数值的方法。可以想象为：如果参数值是一个字符串则自动映射生成的 sql 中参数值两边自动有单引号，如果参数值是一个数字型则自动映射生成的 sql 中参数值两边没有单引号。

注意：当传递单个值时 #{} 中的参数名称通常和 mapper 接口的形参名称相同，也可以设置成任意值。

{} 和 #{} 不同， {} 是将参数值不加修饰的拼在 sql 中，相当中用 jdbc 的 statement 拼接 sql，使用 {} 不能防止 sql 注入，但是有时用 {} 会非常方便，如下的例子：

```
<!-- 根据名称模糊查询用户信息 -->
<select id ="selectUserByName"      parameterType   ="string"    resultType   ="user"  >
    select * from user where username like '%${value}%'
</ select >
```

如果本例子使用 #{} 则传入的字符串中必须有 % 号，而 % 是人为拼接在参数中，显然有点麻烦，如果采用 \${} 在 sql 中拼接为 % 的方式则在调用 mapper 接口传递参数就方便很多。

```
// 如果使用占位符号则必须人在传参数中加 %
List<User> list = userMapper.selectUserByName("%管理员 %");
```

```
// 如果使用 ${} 原始符号则不用人在参数中加 %
List<User> list = userMapper.selectUserByName(" 管理员 ");
```

再比如 order by 排序，如果将列名通过参数传入 sql，根据传的列名进行排序，应该写为：
 ORDER BY \${columnName}
 如果使用 #{} 将无法实现此功能。

注意： \${} 不能防止 sql 注入，对系统安全性有很大的影响，如果使用 \${} 建议传入参数尽量不让用户自动填写，即使要用户填写也要对填写的数据进行校验，保证安全性。
 另外，当传递单个值时 \${} 中填写的参数名称经过测试填写 value 不错报。

@Param(org.apache.ibatis.annotations.Param)

接口如下

```
Public User selectUser(@param( "userName" )Stringname,@param( "userpassword" )String
password);
```

Mapper.xml：

```
1. <select id =" selectUser" resultMap = "BaseResultMap" >
2.   select * from user_user_t where user_name = #{userName} and user_pass
word =#{userPassword }
3. </select>
```

传递简单类型

参考上边的例子。

传递 pojo 对象

Mybatis 使用 ognl 表达式解析对象字段的值，如下例子：

```
<!—传递 pojo 对象综合查询用户信息 —->
<select id ="selectUserByUser"      parameterType = "user"    resultType = "user" >
  select * from user where id=#{          id } and username like '%${      username }%'
```

上边红色标注的是 user 对象中的字段名称。

测试：

```
public void testselectUserByUser() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获限 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 构造查询条件 user 对象
    User user = newUser();
    user.setId(1);
    user.setUsername("管理员");
    // 传递 user 对象查询用户列表
    List<User> list = userMapper.selectUserByUser(user);
    // 关闭 session
    session.close();
}
```

异常测试：

Sql 中字段名输入错误后测试，username 输入 dusername 测试结果报错：

```
org.apache.ibatis.exceptions.PersistenceException:
### Error querying database. Cause: org.apache.ibatis.reflection.ReflectionException: There is
no getter for property named 'dusername' in 'class cn.itcast.mybatisplus.po.User'
### Cause: org.apache.ibatis.reflection.ReflectionException: There is no getter for property
named 'dusername' in 'class cn.itcast.mybatisplus.po.User'
```

传递 hashmap

Sql 映射文件定义如下：

```
<!-- 传递 hashmap 综合查询用户信息 -->
<select id ="selectUserByHashmap" parameterType ="hashmap" resultType ="user" >
    select * from user where id=#{id} and username like '%${username}%'
</ select >
```

上边红色标注的是 hashmap 的 key。

测试：

```
public void testselectUserByHashmap() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获限 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 构造查询条件 Hashmap 对象
    HashMap<String, Object> map = new HashMap<String, Object>();
    map.put("id", 1);
    map.put("username", "管理员");

    // 传递 Hashmap 对象查询用户列表
    List<User> list = userMapper.selectUserByHashmap(map);
    // 关闭 session
    session.close();
}
```

异常测试：

传递的 map 中的 key 和 sql 中解析的 key 不一致。

测试结果没有报错，只是通过 key 获取值为空。

resultType(输出类型)

输出简单类型

参考 getnow 输出日期类型，看下边的例子输出整型：

Mapper.xml 文件

```
<!-- 获取用户列表总数 -->
<select id ="selectUserCount"      parameterType ="user"   resultType ="int" >
    select count(1) from user
</ select >
```

Mapper 接口

```
public int selectUserCount(User user) throws Exception;
```

调用：

```
public void testselectUserCount() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获取 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);

    User user = newUser();
    user.setUsername("管理员");

    // 传递 Hashmap 对象查询用户列表
    int count = userMapper.selectUserCount(user);

    // 使用 session 实现
    //int count =
    session.selectOne("cn.itcast.mybatis.mapper.UserMapper.selectUserCount",
    user);
    // 关闭 session
    session.close();
}
```

总结：

输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型。

使用 session 的 selectOne 可查询单条记录。

输出 pojo

输出 pojo 对象

参考 selectUserById 的定义：

Mapper.xml

```
<!-- 根据 id 查询用户信息 -->
<select id ="selectUserById" parameterType ="int" resultType ="user" >
    select * from user where id = #{id}
</ select >
```

Mapper 接口：

```
public User selectUserById(int id) throws Exception;
```

测试：

```
public void testSelectUserById() throws Exception {
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获限 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 通过 mapper 接口调用 statement
    User user = userMapper.selectUserById(1);
    System.out.println(user);
    // 关闭 session
    session.close();
}
```

使用 session 调用 selectOne 查询单条记录。

输出 pojo 列表

参考 selectUserByName 的定义：

Mapper.xml

```
<!-- 根据名称模糊查询用户信息 -->
<select id ="selectUserByName" parameterType ="string" resultType ="user" >
    select * from user where username like '%${value}%'
</ select >
```

Mapper 接口：

```
public List<User> selectUserByName(String username) throws Exception;
```

测试：

```
public void testselectUserByName() throws Exception{}
```

```
// 获取 session
SqlSession session = sqlSessionFactory.openSession();
// 获得 mapper 接口实例
UserMapper userMapper = session.getMapper(UserMapper.class);
// 如果使用占位符号则必须人在传参数中加 %
//List<User> list = userMapper.selectUserByName("% 管理员 %");
// 如果使用 ${} 原始符号则不用人在参数中加 %
List<User> list = userMapper.selectUserByName(" 管理员 ");
// 关闭 session
session.close();
}
```

使用 session 的 selectList 方法获取 pojo 列表。

总结：

输出 pojo 对象和输出 pojo 列表在 sql 中定义的 resultType 是一样的。

返回单个 pojo 对象要保证 sql 查询出来的结果集为单条，内部使用 session.selectOne 方法调用， mapper 接口使用 pojo 对象作为方法返回值。

返回 pojo 列表表示查询出来的结果集可能为多条，内部使用 session.selectList 方法调用， mapper 接口使用 List<pojo> 对象作为方法返回值。

输出 hashmap

输出 pojo 对象可以改用 hashmap 输出类型，将输出的字段名称作为 map 的 key , value 为字段值。

动态 sql(重点)

Mybatis 提供使用 ognl 表达式动态生成 sql 的功能。

If

```
<!-- 传递 pojo 综合查询用户信息 -->
<select id ="selectUserByUser" parameterType ="user" resultType ="user" >
```

```
select * from user
where 1=1
<if test ="id!=null and id!="">
and id=#{id}
</if>
<if test ="username!=null and username!="">
and username like '%${username}%'
</if>
</select>
```

注意要做不等于空字符串校验。

Where

上边的 sql 也可以改为：

```
<select id ="selectUserByUser" parameterType ="user" resultType ="user" >
    select * from user
    <where>
        <if test ="id!=null and id!="">
        and id=#{id}
        </if>
        <if test ="username!=null and username!="">
        and username like '%${username}%'
        </if>
    </where>
</select>
```

<where /> 可以自动处理第一个 and。

foreach

向 sql 传递数组或 List，mybatis 使用 foreach 解析，如下：

传递 List

传递 List 类型在编写 mapper.xml 没有区别，唯一不同的是只有一个 List 参数时它的参数名为 list。

如下：

Mapper.xml

```
<select id = "selectUserByList" parameterType = "java.util.List" resultType = "user" >
    select * from user
    <where>
        <!-- 传递 List , List 中是 pojo -->
        <if test = "list!=null" >
            <foreach collection = "list" item = "item" open = "and id in(" separator = "," close = ")" >
                #{item.id}
            </foreach>
        </if>
        </where>
    </select>
```

Mapper 接口

```
public List<User> selectUserByList(List userlist) throws Exception;
```

测试：

```
public void testselectUserByList() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获得 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 构造查询条件 List
    List<User> userlist = new ArrayList<User>();
    User user = newUser();
    user.setId(1);
    userlist.add(user);
    user = new User();
    user.setId(2);
```

```

        userlist.add(user);
        // 传递 userlist    列表查询用户列表
        List<User>list = userMapper.selectUserByList(userlist);
        // 关闭 session
        session.close();
    }
}

```

useGeneratedKeys 和 keyProperty

`useGeneratedKeys="true"` 自增主键值
`keyProperty="user_id"`keyProperty 是 Java 对象的属性名！

传递数组（数组中是 pojo）：

请阅读文档学习。

Mapper.xml

```

<!-- 传递数组综合查询用户信息 -->
<select id ="selectUserByArray"      parameterType  ="Object[]"      resultType   ="use
r" >
    select * from user
    <where >
        <!-- 传递数组 -->
        <if test  ="array!=null"      >
            <foreach collection      ="array"      index  ="index"      item  ="item"      open ="and id
in(" separator     =",," close  =")" >
                #{item.id}
            </ foreach >
        </ if >
    </ where >
</ select >

```

sql 只接收一个数组参数，这时 sql 解析参数的名称 mybatis 固定为 array，如果数组是通过一个 pojo 传递到 sql 则参数的名称为 pojo 中的属性名。

`index` : 为数组的下标。

`item` : 为数组每个元素的名称，名称随意定义

`open` : 循环开始

`close` : 循环结束

`separator` : 中间分隔输出

Mapper 接口：

```
public List<User> selectUserByArray(Object[] userlist) throws Exception;
```

测试：

```
public void testselectUserByArray() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获限 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 构造查询条件 List
    Object[] userlist = new Object[2];
    User user = newUser();
    user.setId(1);
    userlist[0] = user;
    user = new User();
    user.setId(2);
    userlist[1] = user;
    // 传递 user 对象查询用户列表
    List<User> list = userMapper.selectUserByArray(userlist);
    // 关闭 session
    session.close();
}
```

传递数组（数组中是字符串类型）：

请阅读文档学习。

Mapper.xml

```
<!-- 传递数组综合查询用户信息 -->
<select id ="selectUserByArray" parameterType ="Object[]" resultType ="User" >
    select * from user
    <where >
        <!-- 传递数组 -->
        <if test ="array!=null" >
            <foreach collection ="array" index ="index" item ="item" open ="and id" >
```

```
in(" separator =," close =") >
    #{item}
    </ foreach >
    </ if >
    </ where >
</ select >
```

如果数组中是简单类型则写为 `#{item}` , 不用再通过 ognl 获取对象属性值了。

Mapper 接口 :

```
public List<User> selectUserByArray(Object[] userlist) throws Exception;
```

测试 :

```
public void testselectUserByArray() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获得 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 构造查询条件 List
    Object[] userlist = new Object[2];
    userlist[0] = "1";
    userlist[1] = "2";
    // 传递 user 对象查询用户列表
    List<User> list = userMapper.selectUserByArray(userlist);
    // 关闭 session
    session.close();
}
```

set

参考 pdf 文档自行学习

Sql 片段

需求

Sql 中可将重复的 sql 提取出来，使用时用 include 引用即可，最终达到 sql 重用的目的，如下：

```
<!-- 传递 pojo 综合查询用户信息 -->
<select id ="selectUserByUser"      parameterType  ="user"  resultType   ="user" >
    select * from user
    <where >
        <if test  ="id!=null and id!=""          >
            and id=#{id}
        </ if >
        <if test  ="username!=null and username!=""      >
            and username like '%${username}%'
        </ if >
    </ where >
</ select >
```

将 where 条件抽取出来：

```
<sql id ="query_user_where"      >
    <if test  ="id!=null and id!=""          >
        and id=#{id}
    </ if >
    <if test  ="username!=null and username!=""      >
        and username like '%${username}%'
    </ if >
</ sql >
```

使用 include 引用：

```
<select id ="selectUserByUser"      parameterType  ="user"  resultType   ="user" >
    select * from user
    <where >
        <include refid  ="query_user_where"      />
```

```
</ where >  
</ select >
```

注意：如果引用其它 mapper.xml 的 sql 片段，则在引用时需要加上 namespace，如下：

```
<include refid ="namespace.sql 片段 ">
```

resultMap

当输出 pojo 的字段和 sql 查询出来的字段名称不对应时而还想用这个 pojo 类作为输出类型这时就需要使用 resultMap 了。

另外，resultMap 也解决了一对一关联查询、一对多关联查询等常见需求。

创建 Person 类

```
public class Person {  
    private int id ;  
    private String name; // 用户姓名，名称和 User 表的字段名称不一样  
    private String sex ; // 性别  
    private Date birthday ; // 出生日期  
    private String addr ;// 地址，名称和 User 表的字段名称不一样  
    private String detail ; // 详细信息  
    private float score ; // 成绩  
get/set ....
```

定义 resultMap

在 mapper.xml 文件中定义 resultMap：

```
<!-- resultMap 定义 -->  
<resultMap type ="cn.itcast.mybatis.po.Person" id ="personmap" >  
<id property ="id" column ="id" />  
<result property ="name" column ="username" />  
<result property ="addr" column ="address" />  
</ resultMap >
```

<id /> : 此属性表示查询结果集的唯一标识，非常重要。如果是多个字段为复合唯一约束则定义多个 <id />。

Property : 表示 person 类的属性。

Column : 表示 sql 查询出来的字段名。

Column 和 property 放在一块儿表示将 sql 查询出来的字段映射到指定的 pojo 类属性上。

<result /> : 普通结果，即 pojo 的属性。

这里只将 sql 查询出来的字段与 pojo 属性名不一致的进行了定义，通过后边的测试 pojo 属性名和 sql 字段相同的自动进行映射。

Mapper.xml 定义

```
<!-- 获取用户列表返回 resultMap -->
<select id ="selectUserListResultMap"      resultMap  ="personmap"  >
    select * from user
</ select >
```

使用 resultMap 指定上边定义的 personmap。

Mapper 接口定义

```
public List<Person> selectUserListResultMap() throws Exception;
```

实际返回的类型是 Person 类型。

测试：

```
public void testselectUserListResultMap() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获限 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);

    User user = newUser();
    user.setUsername("管理员");

    // 查询用户列表返回 resultMap
```

```
List<Person> list = userMapper.selectUserListResultMap();
System.out.println(list);
// 关闭 session
session.close();
}
```

一对一查询

案例：查询所有订单信息，订单信息中显示下单人信息。

注意：因为一个订单信息只会是一个人下的订单，所以从查询订单信息出发关联查询用户信息为一对一查询。如果从用户信息出发查询用户下的订单信息则为一对多查询，因为一个用户可以下多个订单。

方法一：

使用 resultType，定义订单信息 po 类，此 po 类中包括了订单信息和用户信息：

Sql语句：

```
SELECT
orders.*,
user.username,
user.address
FROM
orders,
USER
WHERE orders.user_id = user.id
```

定义 po 类

Po 类中应该包括上边 sql 查询出来的所有字段，如下：

```
public class UserOrder extends Orders {
```

```
private String username ; // 用户姓名  
private String address ; // 地址  
get/set . . . .
```

UserOrder 类继承 Orders 类后 UserOrder 类包括了 Orders 类的所有字段，只需要定义用户的信息字段即可。

Mapper.xml

```
<!-- 查询所有订单信息 -->  
<select id ="findOrdersList"      resultType  ="cn.itcast.mybatis.po.UserOrder"  
" >  
    SELECT  
        orders.*,  
        user.id user_id,  
        user.username,  
        user.address  
    FROM  
        orders,  USER  
    WHERE orders.user_id = user.id  
</ select >
```

Mapper 接口：

```
public List<UserOrder> findOrdersList() throws Exception;
```

测试：

```
public void testfindOrdersList() throws Exception{  
    // 获取 session  
    SqlSession session = sqlSessionFactory.openSession();  
    // 获得 mapper 接口实例  
    UserMapper userMapper = session.getMapper(UserMapper.class);  
    // 查询订单信息  
    List<UserOrder> list = userMapper.findOrdersList();  
    System.out.println(list);  
    // 关闭 session
```

```
    session.close();  
}
```

总结：

定义专门的 po 类作为输出类型，其中定义了 sql 查询结果集所有的字段。此方法较为简单，企业中使用普遍。

方法二：

使用 resultMap，定义专门的 resultMap 用于映射一对一查询结果。

Sql语句：

```
SELECT  
orders.*,  
user.username,  
user.address  
FROM  
orders,  
USER  
WHERE orders.user_id = user.id
```

定义 po 类

在 Orders 类中加入 User 属性。

Mapper.xml

```
<select id ="findOrdersList2"      resultMap  ="userordermap"   >  
    SELECT  
    orders.*,  
    user.username,  
    user.address  
    FROM  
    orders,  USER
```

```
WHERE orders.user_id = user.id
</ select >
```

这里 resultMap 指定 userordermap。

定义 resultMap

```
<!-- 订单信息 resultMap -->
<resultMap type = "cn.itcast.mybatis.po.Orders" id = "userordermap" >
    <!-- 这里的 id 是 mybatis 在进行一对一查询时将 user 字段映射为 user 对象时要使用，必须写 -->
    <id property = "id" column = "id" />
    <result property = "user_id" column = "user_id" />
    <result property = "order_number" column = "order_number" />
    <association property = "user" javaType = "cn.itcast.mybatis.po.User" >
        <!-- 这里的 id 为 user 的 id，如果写上表示给 user 的 id 属性赋值 -->
        <id property = "id" column = "user_id" />
        <result property = "username" column = "username" />
        <result property = "address" column = "address" />
    </ association >
</ resultMap >
```

association : 表示进行关联查询单条记录

property : 表示关联查询的结果存储在 cn.itcast.mybatis.po.Orders 的 user 属性中

javaType : 表示关联查询的结果类型

<id property = "id" column = "user_id" /> : 查询结果的 user_id 列对应关联对象的 id 属性，这里是 <id /> 表示 user_id 是关联查询对象的唯一标识。

<result property = "username" column = "username" /> : 查询结果的 username 列对应关联对象的 username 属性。

Mapper 接口：

```
public List<Orders> findOrdersList2() throws Exception;
```

测试：

```
public void testfindOrdersList2() throws Exception{
    // 获取 session
```

```
SqlSession session = sqlSessionFactory.openSession();
// 获限 mapper 接口实例
UserMapper userMapper = session.getMapper(UserMapper.class);
// 查询订单信息
List<Orders> list = userMapper.findOrdersList2();
System.out.println(list);
// 关闭 session
session.close();
}
```

总结：

此种方法使用了 mybatis 的 association 标签用于一对关联查询，将查询结果映射至对象中。

一对多查询

案例：查询所有订单信息及订单下的订单明细信息。

订单信息与订单明细为一对多关系，一个订单包括多个商品信息。

使用 resultMap 实现如下：

Sql语句：

```
SELECT
orders.*,
user.username,
user.address,
orderdetail.id orderdetail_id,
orderdetail.item_id,
orderdetail.item_num,
orderdetail.item_price
FROM
orders,USER ,orderdetail
```

```
WHERE orders.user_id = user.id  
AND orders.id = orderdetail.orders_id
```

定义 po 类

在 Orders 类中加入 User 属性。
在 Orders 类中加入 List<Orderdetail> orderdetails 属性

Mapper.xml

```
<select id = "findOrdersDetailList" resultMap = "userorderdetailmap" >  
    SELECT  
        orders.*,  
        user.username,  
        user.address,  
        orderdetail.id orderdetail_id,  
        orderdetail.item_id,  
        orderdetail.item_num,  
        orderdetail.item_price  
    FROM orders,USER ,orderdetail  
    WHERE orders.user_id = user.id  
    AND orders.id = orderdetail.orders_id  
</ select >
```

定义 resultMap

```
<!-- 订单信息 resultMap -->  
<resultMap type ="cn.itcast.mybatis.po.Orders" id = "userorderdetailmap" >  
    <id property = "id" column = "id" />  
    <result property = "user_id" column = "user_id" />  
    <result property = "order_number" column = "order_number" />  
    <association property = "user" javaType = "cn.itcast.mybatis.po.User" >  
        <id property = "id" column = "user_id" />  
        <result property = "username" column = "username" />  
        <result property = "address" column = "address" />  
    </ association >  
<collection property = "orderdetails" ofType = "cn.itcast.mybatis.po.Orderdet
```

```
ail" >
    <id property = "id" column = "orderdetail_id"      />
    <result property = "item_id"   column = "item_id"    />
    <result property = "item_num"  column = "item_num"   />
    <result property = "item_price" column = "item_price" />
</ collection >
</ resultMap >
```

黄色部分和上边一对查询订单及用户信息定义的 `resultMap` 相同，
`collection` 部分定义了查询订单明细信息。
`collection` : 表示关联查询结果集
`property = "orderdetails"` : 关联查询的结果集存储在 `cn.itcast.mybatis.po.Orders` 上哪个属性。
`ofType = "cn.itcast.mybatis.po.Orderdetail"` : 指定关联查询的结果集中的对象类型即 `List` 中的对象类型。
`<id />` 及 `<result />` 的意义同一对一查询。

Mapper 接口：

```
public List<Orders>findOrdersDetailList () throws Exception;
```

测试：

```
public void testfindOrdersDetailList() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获得 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 查询订单信息
    List<Orders> list = userMapper.findOrdersDetailList();
    System.out.println(list);
    // 关闭 session
    session.close();
}
```

总结：

此种方法使用了 mybatis 的 collection 标签用于一对多关联查询，将查询结果映射至集合对象中。

resultMap 使用继承

上边定义的 resultMap 中黄色部分和一对一查询订单信息的 resultMap 相同，这里使用继承可以不再填写重复的内容，如下：

```
<resultMap type ="cn.itcast.mybatis.po.Orders" id ="userorderdetailmap2" extends ="userordermap" >
<collection property ="orderdetails" ofType ="cn.itcast.mybatis.po.Orderdetail" >
    <id property ="id" column ="orderdetail_id" />
    <result property ="item_id" column ="item_id" />
    <result property ="item_num" column ="item_num" />
    <result property ="item_price" column ="item_price" />
</ collection >
</ resultMap >
```

使用 extends 继承订单信息 userordermap 。

多对多查询

案例：查询所有订单信息及订单明细的商品信息。

订单信息与商品信息为多对多关系，因为一个订单包括多个商品信息，一个商品可以在多个订单中存在，订单信息与商品信息的多对多关系是通过订单明细表进行关联。

Sql语句：

```
SELECT
orders.*,
user.username,
```

```
user.address,  
orderdetail.id orderdetail_id,  
orderdetail.item_id,  
orderdetail.item_num,  
orderdetail.item_price,  
items.item_name,  
items.item_detail  
  
FROM  
orders,USER ,orderdetail,items  
  
WHERE orders.user_id = user.id  
AND orders.id = orderdetail.orders_id  
AND orderdetail.item_id = items.id
```

定义 po 类

在 Orders 类中加入 User 属性。

在 Orders 类中加入 List<Orderdetail> orderdetails 属性，存储订单明细信息

在 Orderdetail 类中加入 Items items 属性存储商品信息

Mapper.xml

```
<select id ="findOrdersItemsList" resultMap ="userorderitemsmap" >  
    SELECT  
        orders.*,  
        user.username,  
        user.address,  
        orderdetail.id orderdetail_id,  
        orderdetail.item_id,  
        orderdetail.item_num,  
        orderdetail.item_price,  
        items.item_name,  
        items.item_detail  
    FROM  
        orders,USER ,orderdetail,items  
  
    WHERE orders.user_id = user.id  
    AND orders.id = orderdetail.orders_id
```

```
    AND orderdetail.item_id = items.id  
</ select >
```

定义 resultMap

```
<!-- 订单商品信息 resultMap -->  
<resultMap type = "cn.itcast.mybatis.po.Orders" id = "userorderitemsmap"  
extends = "userordermap" >  
  
<collection property = "orderdetails" ofType = "cn.itcast.mybatis.po.Order  
detail" >  
    <id property = "id" column = "orderdetail_id" />  
    <result property = "item_id" column = "item_id" />  
    <result property = "item_num" column = "item_num" />  
    <result property = "item_price" column = "item_price" />  
    <!-- 商品信息 -->  
  
<association property = "items" javaType = "cn.itcast.mybatis.po.Items" >  
    <id property = "id" column = "item_id" />  
    <result property = "item_name" column = "item_name" />  
    <result property = "item_detail" column = "item_detail" />  
    </ association >  
    </ collection >  
</ resultMap >
```

在 collection 中加入 association 通过订单明细表关联查询商品信息

Mapper 接口：

```
public List<Orders>findOrdersItemsList () throws Exception;
```

测试：

```
public void findOrdersItemsList() throws Exception{  
    // 获取 session  
    SqlSession session = sqlSessionFactory.openSession();
```

```

// 获限 mapper 接口实例
UserMapper userMapper = session.getMapper(UserMapper.class);
// 查询订单信息
List<Orders> list = userMapper.findOrdersItemsList();
System.out.println(list);
// 关闭 session
session.close();
}

```

总结：

所谓一对多查询、 多对多查询都对于具体的业务分析来说， 使用 mybatis 提交的 collection 和 association 可以完成不同的关联查询需求，通常在实际应用时 association 用自定义 pojo 方式代替，关联查询结果集使用 collection 完成。

延迟加载

需要查询关联信息时，使用 mybatis 延迟加载特性可有效的减少数据库压力，首次查询只查询主要信息，关联信息等用户获取时再加载。

打开延迟加载开关

在 mybatis 核心配置文件中配置：

lazyLoadingEnabled、 aggressiveLazyLoading

设置项	描述	允许值	默认值
lazyLoadingEnabled	全局性设置懒加载。如果设为‘ false ’，则所有相关联的都会被初始化加载。	true false	false
aggressiveLazyLoading	当设置为‘ true ’的时候，懒加载的对象可能被任何懒属性全部加载。否则，每个属性都按需加载。	true false	true

```

<settings>
    <setting name="lazyLoadingEnabled" value="true" />
    <setting name="aggressiveLazyLoading" value="false" />

```

```
</ settings >
```

一对一查询延迟加载

Sql语句：

```
SELECT  
orders.*  
FROM  
orders
```

定义 po 类

在 Orders 类中加入 User 属性。

定义 resultMap

```
<!-- 订单信息 resultMap -->  
<resultMap type = "cn.itcast.mybatis.po.Orders" id ="userordermap2" >  
<id property = "id" column = "id" />  
<result property = "user_id" column = "user_id" />  
<result property = "order_number" column = "order_number" />  
<association property = "user" javaType = "cn.itcast.mybatis.po.User" select = "  
selectUserById" column = "user_id" />  
</ resultMap >
```

association :

select = "selectUserById" : 指定关联查询 sql 为 selectUserById
column = "user_id" : 关联查询时将 user_id 列的值传入 selectUserById
最后将关联查询结果映射至 cn.itcast.mybatis.po.User 。

Mapper.xml

```
<select id = "findOrdersList3"      resultMap  = "userordermap2" >
    SELECT
        orders.*
    FROM
        orders
</ select >
```

Mapper 接口：

```
public List<Orders> findOrdersList3() throws Exception;
```

测试：

```
public void testfindOrdersList3() throws Exception{
    // 获取 session
    SqlSession session = sqlSessionFactory.openSession();
    // 获限 mapper 接口实例
    UserMapper userMapper = session.getMapper(UserMapper.class);
    // 查询订单信息
    List<Orders> list = userMapper.findOrdersList3();
    System.out.println(list);
    // 开始加载，通过 orders.getUser 方法进行加载
    for(Orders orders:list){
        System.out.println(orders.getUser());
    }
    // 关闭 session
    session.close();
}
```

总结：

使用延迟加载提高数据库查询性能，默认不查询关联数据，按需要发出

sql 请求关联查询信

息。

6. 缓存

一级缓存

Mybatis 一级缓存的作用域是同一个 `SqlSession`，
一级缓存默认就开启，无须人为配置。

第一个例子：

```
// 获取 session
SqlSession session = sqlSessionFactory.openSession();
// 获得 mapper 接口实例
UserMapper userMapper = session.getMapper(UserMapper.class);
// 第一次查询
User user1 = userMapper.selectUserById(1);
System.out.println(user1);
// 第二次查询，由于是同一个 session 则不再向数据发出语句直接从缓存取出
User user2 = userMapper.selectUserById(1);
System.out.println(user2);
```

原理：

Mybatis 首先去缓存中查询结果集，如果没有则查询数据库，如果有则从缓存取出返回结果集就不走数据库。

Mybatis 内部存储缓存使用一个 `HashMap`，key 为 `hashCode+sqlId+Sql` 语句。value 为从查询出来映射生成的 `java` 对象

如果对数据进行修改，针对 `update`、`insert`、`delete` 一级缓存自动清空。

第二个例子：

```
// 获取 session
SqlSession session = sqlSessionFactory.openSession();
// 获得 mapper 接口实例
```

```
UserMapper userMapper = session.getMapper(UserMapper.class);
// 第一次查询
User user1 = userMapper.selectUserById(1);
System.out.println(user1);
// 在同一个 session 执行更新
User user_update = newUser();
user_update.setId(1);
user_update.setUsername("李奎");
userMapper.updateUser(user_update);
session.commit();

// 第二次查询，虽然是同一个 session 但是由于执行了更新操作 session 的缓存被清空，这里重新发出 sql 操作
User user2 = userMapper.selectUserById(1);
System.out.println(user2);
```

原理

该例子与第一个例子不同的是在两次查询中间加入了更新，更新操作执行后 mybatis 执行了清除缓存即清空 HashMap。

二级缓存

Mybatis 的二级缓存即查询缓存，它的作用域是一个 mapper 的 namespace，即在同一个 namespace 中查询 sql 可以从缓存中获取数据。
二级缓存是可以跨 SqlSession 的。

开启二级缓存：

1. 在核心配置文件 SqlMapConfig.xml 中加入

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

	描述	允许值	默认值
cacheEnabled	对在此配置文件下的所有 cache 进行全局性开 / 关设置。	true false	true

2. 要在你的 Mapper 映射文件中添加一行：
`<cache />`

3. 在 select 语句中 **useCache=false** 可以禁用当前的语句的二级缓存，即每次查询对 session 的查询都会发出 sql 去查询，默认情况是 true，即该 sql 使用二级缓存。

实现序列化：

注意：将查询结果的 pojo 对象进行序列化实现 `java.io.Serializable` 接口

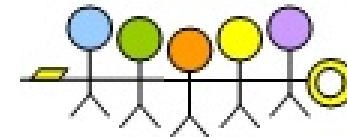
例子：

```
// 获取 session1
    SqlSession session1 = sqlSessionFactory.openSession();
    UserMapper userMapper = session1.getMapper(UserMapper.class);
    // 使用 session1 执行第一次查询
    User user1 = userMapper.selectUserById(1);
    System.out.println(user1);
    // 关闭 session1
    session1.close();
    // 获取 session2
    SqlSession session2 = sqlSessionFactory.openSession();
    UserMapper userMapper2 = session2.getMapper(UserMapper.class);
    // 使用 session2 执行第二次查询，由于开启了二级缓存这里从缓存中获取数据不再向数据库发出 sql
    User user2 = userMapper2.selectUserById(1);
    System.out.println(user2);
    // 关闭 session2
    session2.close();
```

刷新缓存

在 mapper 的同一个 namespace 中，如果有其它 insert、update、delete 操作数据后需要刷新缓存，如果不执行刷新缓存会出现脏读。

4. sql 中的 **flushCache = "true"** 属性，默认情况下为 true 即刷新缓存，如果改成 false 则不会刷新。使用缓存时如果手动修改数据库表中的查询数据会出现脏读。



如下：

```
<insert id="insertUser" parameterType="cn.itcast.mybatis.po.User" flushCache="true" >
```

cache 的其它参数：

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。默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新。

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。默认值是 1024。

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。因此这些对象不能被修改。这提供了很重要的性能优势。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。

如下例子：

```
<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，而且返回的对象被认为是只读的，因此在不同线程中的调用者之间修改它们会导致冲突。

可用的收回策略有，默认的是 LRU：

1. LRU – 最近最少使用的：移除最长时间不被使用的对象。
2. FIFO – 先进先出：按对象进入缓存的顺序来移除它们。
3. SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。
4. 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。
- 5.

应用场景：

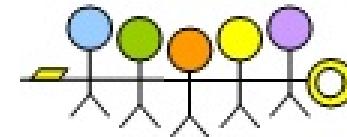
1、针对复杂的查询或统计的功能，用户不要求每次都查询到最新信息，使用二级缓存，通过刷新间隔 flushInterval 设置刷新间隔时间，由 mybatis 自动刷新。

比如：实现用户分类统计 sql，该查询非常耗费时间。

将用户分类统计 sql 查询结果使用二级缓存，同时设置刷新间隔时间： flushInterval（一般设置时间较长，比如 30 分钟，60 分钟，24 小时，根据需求而定）

2、针对信息变化频率高，需要显示最新的信息，使用二级缓存。

将信息查询的 statement 与信息的增、删、改定义在一个 mapper.xml 中，此 mapper 实现二级缓存，当执行增、删、修改时，由 mybatis 及时刷新缓存，满足用户从缓存查询到最新的数据。



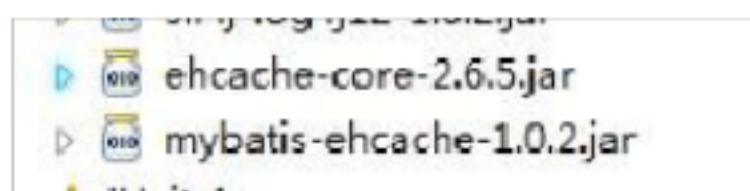
比如：新闻列表显示前 10 条，该查询非常快，但并发对数据也有压力。

将新闻列表查询前 10 条的 sql 进行二级缓存，这里不用刷新间隔时间，当执行新闻添加、删除、修改时及时刷新缓存。

二级缓存使用 Ehcache

Mybatis 与缓存框架 ehcache 进行了整合，采用 ehcache 框架管理缓存数据。

第一步：引入缓存的依赖包



第二步：引入缓存配置文件

ehcache.xml

defaultCache 配置说明：

maxElementsInMemory 内存中最大缓存对象数 . 当超过最大对象数的时候 ,ehcache 会按指定的策略去清理内存

eternal 缓存对象是否永久有效 ,一但设置了 ,timeout 将不起作用 .

timeToIdleSeconds 设置 Element 在失效前的允许闲置时间 .仅当 element 不是永久有效时使用 ,可选属性 ,默认值是 0, 也就是可闲置时间无穷大 .

timeToLiveSeconds : 设置 Element 在失效前允许存活时间 .最大时间介于创建时间和失效时间之间 .仅当 element 是永久有效时使用 ,默认是 0., 也就是 element 存活时间无穷大 .

overflowToDisk 配置此属性 ,当内存中 Element 数量达到 maxElementsInMemory 时,Ehcache 将会 Element 写到磁盘中 .

diskSpoolBufferSizeMB 这个参数设置 DiskStore(磁盘缓存) 的缓存区大小 .默认是 30MB. 每个 Cache 都应该有自己的一个缓冲区 .

maxElementsOnDisk 磁盘中最大缓存对象数 ,若是 0 表示无穷大 .

diskPersistent 是否在重启服务的时候清楚磁盘上的缓存数据 .true 不清除 .

diskExpiryThreadIntervalSeconds 磁盘失效线程运行时间间隔 .

memoryStoreEvictionPolicy : 当达到 maxElementsInMemory 限制时 ,Ehcache 将会根据指定的策略去清理内存 .默认策略是 LRU(最近最少使用).你可以设置为 FIFO(先进先出) 或是 LFU(较少使用).

第三步：修改 mapper 文件中缓存类型

在 cache 中指定 type。

```
<cache type ="org.mybatis.caches.ehcache.EhcacheCache" />
```

7. Mybatis 与 springmvc 整合

Dao

Spring 配置文件：

applicationContext.xml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"           xmlns:mvc = "http
://www.springframework.org/schema/mvc"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:aop = "http://www.springframework.org/schema/aop"           xmlns:tx = "http:
//www.springframework.org/schema/tx"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.1.xsd" >
```

```

<!-- 引用配置文件 -->
<context:property-placeholder          location  ="classpath:db.properties"      />
<bean id ="dataSource"    class  ="org.apache.commons.dbcp.BasicDataSource"
      destroy-method   ="close"    >
    <property   name="driverClassName"      value  ="${mysql.driver}"        />
    <property   name="url"    value  ="${mysql.url}"        />
    <property   name="username"  value  ="${mysql.username}"      />
    <property   name="password"  value  ="${mysql.password}"      />
    <property   name="maxActive"  value  ="30"      />
    <property   name="maxIdle"   value  ="5"       />
</ bean >

</ beans >

```

applicationContext-dao.xml

```

<?xml version  ="1.0"  encoding  ="UTF-8"  ?>
<beans xmlns ="http://www.springframework.org/schema/beans"
       xmlns:xsi  ="http://www.w3.org/2001/XMLSchema-instance"           xmlns:mvc  ="http
://www.springframework.org/schema/mvc"
       xmlns:context     ="http://www.springframework.org/schema/context"
       xmlns:aop      ="http://www.springframework.org/schema/aop"           xmlns:tx      ="http:
//www.springframework.org/schema/tx"
       xsi:schemaLocation      ="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.1.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.1.xsd "      >

<!-- 会话工厂 -->
<bean id ="sqlSessionFactory"      class  ="org.mybatis.spring.SqlSessionFacto
ryBean"  >
    <property   name="dataSource"    ref  ="dataSource"    ></ property >
    <!-- 加载 mybatis 的配置文件 -->

```

```
<property name="configLocation" value ="classpath:mybatis/sqlMapConfig.xml" /></ property >
</ bean >

<!-- mapper 扫描器，这里由于没有在 sqlMapConfig 配置 mapper，所以必须保证 mapper 和 dao 接口在同一个目录且同名 -->
<bean class ="org.mybatis.spring.mapper.MapperScannerConfigurer" >

<property name="basePackage" value ="yycg.**.dao.mapper" /></ property >

<property name="sqlSessionFactoryBeanName" value ="sqlSessionFactory" />
</ bean >

<!-- 如果采用自动扫描器则不用手动设置工厂 bean
<bean id="useryyMapper2"
class="org.mybatis.spring.mapper.MapperFactoryBean">
<property name="mapperInterface"
value="yycg.dao.mapper.UserMapper" />
<property name="sqlSessionFactory" ref="sqlSessionFactory" />
</bean>-->

</ beans >
```

sqlmapConfig.xml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd" >
<configuration >

<! - 使用自动扫描器时， mapper.xml 文件如果和 mapper.java 接口在一个目录则此处不用
定义 mappers -->
<mappers >
<package name="cn.itcast.mybatis.mapper" />
</ mappers >
</ configuration >
```

Mapper 编写的三种方法

1 接口实现类继承 SqlSessionDaoSupport

使用此种方法需要编写 mapper 接口， mapper 接口实现类、 mapper.xml 文件

1、 在 sqlMapConfig.xml 中配置 mapper.xml 的位置

```
<mappers>
<mapper resource= "mapper.xml 文件的地址 " />
<mapper resource= "mapper.xml 文件的地址 " />
</mappers>
```

2、 定义 mapper 接口

3、 实现类集成 SqlSessionDaoSupport

mapper 方法中可以 this.getSqlSession() 进行数据增删改查。

4、 spring 配置

```
<bean id = " " class = "mapper 接口的实现 " >
<property name="sqlSessionFactory" ref = "sqlSessionFactory" ></ property >
</ bean >
```

2 使用 org.mybatis.spring.mapper.MapperFactoryBean

1、 在 sqlMapConfig.xml 中配置 mapper.xml 的位置

如果 mapper.xml 和 mappre 接口的名称相同且在同一个目录，这里可以不用配置

```
<mappers>
<mapper resource= "mapper.xml 文件的地址 " />
<mapper resource= "mapper.xml 文件的地址 " />
</mappers>
```

2、 定义 mapper 接口

注意

1、 mapper.xml 中的 namespace 为 mapper 接口的地址

2、 mapper 接口中的方法名和 mapper.xml 中定义的 statement 的 id 保持一致

3、 Spring 中定义

```
<bean id = " " class = "org.mybatis.spring.mapper.MapperFactoryBean" >
<property name="mapperInterface" value = "mapper 接口地址 " />
<property name="sqlSessionFactory" ref = "sqlSessionFactory" />
</ bean >
```

3 使用 mapper 扫描器

1、 mapper.xml 文件编写，

注意：

mapper.xml 中的 namespace 为 mapper 接口的地址

mapper 接口中的方法名和 mapper.xml 中的定义的 statement 的 id 保持一致

如果将 mapper.xml 和 mapper 接口的名称保持一致则不用在 sqlMapConfig.xml 中进行配置

2、 定义 mapper 接口

注意 mapper.xml 的文件名和 mapper 的接口名称保持一致，且放在同一个目录

3、 配置 mapper 扫描器

```
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="mapper 接口包地址" />
    <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />
</bean>
```

4、 使用扫描器后从 spring 容器中获取 mapper 的实现对象

扫描器将接口通过代理方法生成实现对象，要 spring 容器中自动注册，名称为 mapper 接口的名称。

Service

UserManager 接口

编写 UserManagerService 接口，如下：

```
public interface UserManagerService {
    /**
     * 根据 id 查询用户
}
```

```
/*
public User findUserById(String id) throws Exception;
}
```

```
public class UserManagerServiceImpl implements UserManagerService {

    @Autowired
    UserMapper userMapper;

    @Override
    public User findUserById(int id) throws Exception {
        return userMapper.selectUserById(id);
    }

}
```

Spring配置文件：

将 userManager 在 spring 配置文件进行配置

applicationContext--service.xml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<beans xmlns = "http://www.springframework.org/schema/beans"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc = "http://www.springframework.org/schema/mvc"
       xmlns:context = "http://www.springframework.org/schema/context"
       xmlns:aop = "http://www.springframework.org/schema/aop"
       xmlns:tx = "http://www.springframework.org/schema/tx"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
                           http://www.springframework.org/schema/mvc
                           http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.1.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-3.1.xsd" >
```

```
<!-- 用户管理 -->
<bean id ="userManagerService"      class  ="cn.itcast.mybatis.service.impl.UserM
anagerServiceImpl"      />

</ beans >
```

Service 测试：

```
ApplicationContext      applicationContext      ;

protected void      setUp()      throws Exception {
    applicationContext      = new ClassPathXmlApplicationContext(
        new String[]{
            "spring/applicationContext.xml"      ,
            "spring/applicationContext-dao.xml"      ,
            "spring/applicationContext-service.xml"
        }
    );
}

public void      testFindUserById()      throws Exception {
    UserManagerService userManagerService =
(UserManagerService)      applicationContext      .getBean(      "userManagerService"      );
    System.out.println(userManagerService.findUserById(1));
}
```

事务控制：

配置

在 applicationContext.xml 中配置事务管理器

```
<!-- 事务控制 -->
<bean id ="txManager-base"
class  ="org.springframework.jdbc.datasource.DataSourceTransactionMana
ger" >
    <property      name="dataSource"      ref  ="dataSource"      ></ property      >
</ bean >
<tx:advice      id = "txAdvice-base"      transaction-manager      ="txManager-base"      >
```

```

<tx:attributes      >
    <tx:method name="save*"   propagation  ="REQUIRED" />
    <tx:method name="insert*"  propagation  ="REQUIRED" />
    <tx:method name="update*"  propagation  ="REQUIRED" />
    <tx:method name="delete*"  propagation  ="REQUIRED" />
    <tx:method name="get*"    read-only   ="true"   />
    <tx:method name="select*"  read-only   ="true"   />
    <tx:method name="find*"   read-only   ="true"   />
</ tx:attributes      >
</ tx:advice     >

<aop:config proxy-target-class      ="true"   >
    <aop:advisor
        pointcut   ="execution(* cn.itcast.**.service.impl.*.*(..))"
        advice-ref  ="txAdvice-base"      />
</ aop:config     >

```

事务测试

在一个 service 方法中先执行更新，再执行插入，插入一个违反唯一约束的记录，如果数据不回滚则说明事务没有控制。

Action

springmvc.xml 配置文件

```

<?xml version  ="1.0"  encoding  ="UTF-8"  ?>
<beans xmlns ="http://www.springframework.org/schema/beans"
    xmlns:xsi   ="http://www.w3.org/2001/XMLSchema-instance"           xmlns:mvc  ="http
://www.springframework.org/schema/mvc"
    xmlns:context  ="http://www.springframework.org/schema/context"
    xmlns:aop    ="http://www.springframework.org/schema/aop"           xmlns:tx   ="http:
//www.springframework.org/schema/tx"
    xsi:schemaLocation      ="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.1.xsd
        http://www.springframework.org/schema/aop

```

```
http://www.springframework.org/schema/aop/spring-aop-3.1.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.1.xsd "      >

<!-- 注解驱动 -->
<mvc:annotation-driven      />
<!-- 组件扫描，用于控制层 -->
<context:component-scan      base-package  ="cn.itcast.mybatis.action"      />

<!-- 视图解析器 -->
<bean

    class  ="org.springframework.web.servlet.view.InternalResourceViewReso
lver"  >
        <property   name="prefix"      value  ="/WEB-INF/jsp"      ></ property     >
        <property   name="suffix"      value  =".jsp"      ></ property     >
</ bean >

<!-- 拦截器 -->
<!--<mvc:interceptors>
    多个拦截器，顺序执行
    <mvc:interceptor>
        <mvc:mapping path="/" />
        <bean
class="cn.itcast.project.yycg.base.filter.LoginInterceptor"></bean>
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/" />
            <bean
class="cn.itcast.project.yycg.base.filter.PermissionInterceptor"></bean>
        </mvc:interceptor>
    </mvc:interceptors> -->

</ beans >
```

编写 UserAction.java

```
/**
```

```
* 用户管理
* @author Thinkpad
*
*/
@Controller
@RequestMapping ( "/user" )
public class UserAction {

    @Autowired
    UserManagerService userManagerService;

    /**
     * 用户修改
     * @param model
     * @param id
     * @return
     * @throws Exception
     */
    @RequestMapping ( "/useredit" )
    public String useredit(Model model, int id) throws Exception{
        User user = userManagerService.findUserById(id);
        model.addAttribute("user", user);
        return "useredit";
    }
    /**
     * 用户修改提交
     * @param user
     * @return
     * @throws Exception
     */
    @RequestMapping ( "/usereditsubmit" )
    public String usereditsubmit(User user) throws Exception{
        userManagerService.saveUser(user);
        return "success";
    }

    // 其它方法略
    // .....
}
```

注意：学会如果在 action 中调用 service，处理结果返回用户。

web.xml

```
<?xml version  ="1.0"  encoding  ="UTF-8"  ?>
<web-app xmlns:xsi  ="http://www.w3.org/2001/XMLSchema-instance"          xmlns = "htt
p://java.sun.com/xml/ns/javaee"           xsi:schemaLocation  ="http://java.sun.com/
xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"           id ="WebApp_ID"  version
="3.0"  >
<display-name  >mybatis_03</ display-name  >
<context-param  >
<param-name >contextConfigLocation      </ param-name >
<param-value >/WEB-INF/classes/spring/applicationContext.xml,/WEB-INF/cl
asses/spring/applicationContext-*.xml      __</ param-value >
</ context-param  >
<listener  >
<listener-class  >org.springframework.web.context.ContextLoaderListener      </
listener-class  >
</ listener  >

<filter  >
<filter-name  >SpringCharacterEncodingFilter      </ filter-name >
<filter-class  >org.springframework.web.filter.CharacterEncodingFilter      </
filter-class  >
<init-param  >
<param-name >encoding  </ param-name >
<param-value >UTF-8 </ param-value >
</ init-param  >
</ filter  >
<filter-mapping  >
<filter-name  >SpringCharacterEncodingFilter      </ filter-name >
<url-pattern  >/* </ url-pattern >
</ filter-mapping  >
<servlet  >
<servlet-name  >springmvc</ servlet-name >
<servlet-class  >org.springframework.web.servlet.DispatcherServlet      </
servlet-class  >
<init-param  >
<param-name >contextConfigLocation      </ param-name >
<param-value >classpath:spring/springmvc-servlet.xml      </ param-value >
</ init-param  >
</ servlet  >
```

```
<servlet-mapping>
< servlet-name >springmvc </ servlet-name >
< url-pattern >*.action </ url-pattern >
</ servlet-mapping >

< welcome-file-list >
< welcome-file >index.jsp </ welcome-file >
</ welcome-file-list >
</ web-app >
```

测试

将工程部署在 tomcat 运行，输入：http://localhost:8080/mybatis_03/user/usredit.aciton?id=1，
进入首页

8. Mybatis 逆向工程

使用官方网站的 mapper 自动生成工具 mybatis-generator-core-1.3.2 来生成 po 类和 mapper 映射文件。

第一步 mapper 生成配置文件：

在 generatorConfig.xml 中配置 mapper 生成的详细信息，注意改下几点：

- 1、添加要生成的数据库表
- 2、po 文件所在包路径
- 3、mapper 文件所在包路径

配置文件如下：

详见 generatorSqlmapCustom 工程

第二步使用 java 类生成 mapper 文件：

```
public void generator() throws Exception{
    List<String> warnings = new ArrayList<String>();
    boolean overwrite = true ;
    File configFile = newFile( "generatorConfig.xml" );
    ConfigurationParser cp = newConfigurationParser(warnings);
    Configuration config = cp.parseConfiguration(configFile);
    DefaultShellCallback callback = new
DefaultShellCallback(overwrite);
    MyBatisGenerator myBatisGenerator =
newMyBatisGenerator(config,
                    callback, warnings);
    myBatisGenerator.generate( null );
}

public static void main(String[] args) throws Exception {
    try {
        GeneratorSqlmap generatorSqlmap = newGeneratorSqlmap();
        generatorSqlmap.generator();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

第三步：拷贝生成的 **mapper** 文件到工程中指定的目录中

Mapper.xml

Mapper.xml 的文件拷贝至 mapper 目录内

Mapper.java

Mapper.java 的文件拷贝至 mapper 目录内

注意： mapper xml 文件和 mapper.java 文件在一个目录内且文件名相同。

第四步 Mapper 接口测试

学会使用 mapper 自动生成的增、删、改、查方法。

```
// 删除符合条件的记录
int deleteByExample(UserExample example);
// 根据主键删除
int deleteByPrimaryKey(String id);
// 插入对象所有字段
int insert(User record);
// 插入对象不为空的字段
int insertSelective(User record);
// 自定义查询条件查询结果集
List<User>selectByExample(UserExample example);
// 根据主键查询
User selectByPrimaryKey(String id);
// 根据主键将对象中不为空的值更新至数据库
int updateByPrimaryKeySelective(User record);
// 根据主键将对象中所有字段的值更新至数据库
int updateByPrimaryKey(User record);
```

注意：

Mapper 文件内容不覆盖而是追加

XXXMapper.xml 文件已经存在时，如果进行重新生成则 mapper.xml 文件内容不被覆盖而是进行内容追加，结果导致 mybatis 解析失败。

解决方法：删除原来已经生成的 mapper.xml 文件再进行生成。

Mybatis 自动生成的 po 及 mapper.java 文件不是内容而是直接覆盖没有此问题。

Table schema 问题

下边是关于针对 oracle 数据库表生成代码的 schema 问题：

Schma 即数据库模式，oracle 中一个用户对应一个 schema，可以理解为用户就是 schema。当 Oracle 数据库存在多个 schema 可以访问相同的表名时，使用 mybatis 生成该表的 mapper.xml 将会出现 mapper.xml 内容重复的问题，结果导致 mybatis 解析错误。

解决方法：在 table 中填写 schema，如下：

```
<table schema="XXXX" tableName=" " >
```

XXXX即为一个 schema 的名称，生成后将 mapper.xml 的 schema 前缀批量去掉，如果不去掉当 oracle 用户变更了 sql 语句将查询失败。

快捷操作方式： mapper.xml 文件中批量替换：“ from XXXX.” 为空

Oracle 查询对象的 schema 可从 dba_objects 中查询，如下：

```
select * from dba_objects
```