

Mybatis 学习记录和总

一、概述

MyBatis 是 Apache 的一个开源项目 iBatis，2010年该项目由 Apache Software Foundation 迁移至 Google Code，并正式改名为 MyBatis，2013年11月迁移至GitHub。

MyBatis 是一个实现了数据持久化的开源框架，她对 JDBC 进行了封装。

MyBatis 的优点：

- 与JDBC 相比，减少了五成以上的代码量。
- MyBatis 是最简单的持久化框架，小巧且简单易学。
- MyBatis 相当灵活，不会对应用程序或者数据库的现有设计强加任何影响，所有的 SQL 语句都写在 XML 文件里面，从程序代码中彻底分离出来，大大降低了耦合度，便于统一管理和优化，并且可重用。
- 提供XML 标签，支持编写动态SQL语句。
- 提供映射标签，支持对象与数据库的ORM 字段关系映射。

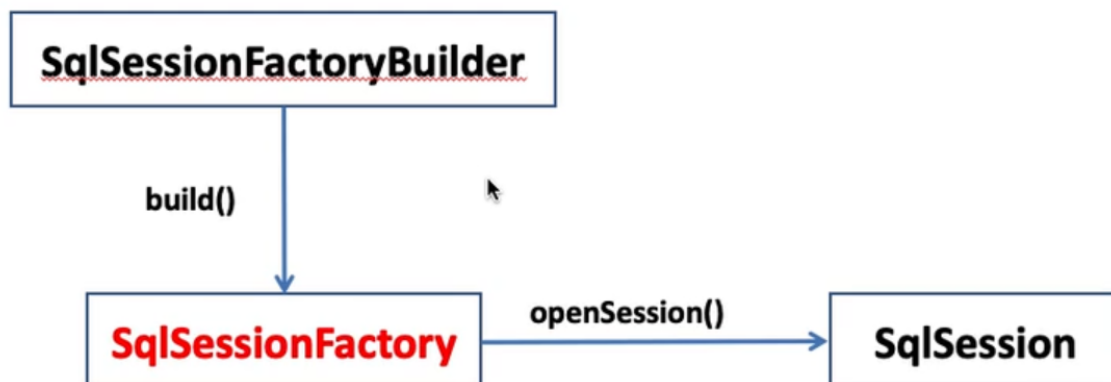
MyBatis的缺点：

- SQL语句的编写工作量大，尤其是字段多、关联表多时更是如此，对开发人员编写SQL语句的功底有一定要求。
- SQL语句过于依赖数据库，导致数据库移植性差，不能随意更换数据库。

MyBatis的开发方式：

- 使用原生接口
- Mapper代理实现自定义接口

MyBatis 核心接口和类



二、MyBatis 的使用

- 新建Maven工程, pom.xml 添加依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>mybatis_demo</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.5</version>
    </dependency>

    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-
java</artifactId>
      <version>8.0.20</version>
```

```

        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.12</version>
        </dependency>
    </dependencies>
</project>

```

- 创建对应数据表

```

CREATE TABLE account(
id INT PRIMARY KEY auto_increment,
username VARCHAR(11),
password VARCHAR(11),
age int
)

```

- 新建数据表对应的实体类

```

package com.gloryh.entity;

import lombok.Data;

/**
 * 实体类Account
 *
 * @author 黄光辉
 * @since 2020/9/17
 */
@Data
public class Account {
    private long id;
    private String username;
    private String password;
    private int age;
}

```

- 创建 MyBatis 的配置文件 config.xml（文件名可自定义）

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD
Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-
config.dtd">
<configuration>
    <!--配置 MyBatis的可运行环境-->
    <environments default="development">
        <environment id="development">
            <!-- 配置JDBC的事务管理 -->
            <transactionManager type="JDBC"/>
            <!-- 配置数据源（POOLED） -->
            <dataSource type="POOLED">
                <property name="driver"
value="com.mysql.cj.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3305/mybatis_demo?
useUnicode=true&characterEncoding=utf8&serv
erTimezone=UTC&allowMultiQueries=true"/>
                <property name="username"
value="admin"/>
                <property name="password"
value="123"/>
            </dataSource>
        </environment>
    </environments>
</configuration>

```

1、使用原生接口

MyBatis 框架需要开发者自定义SQL语句，写在 Mapper.xml 文件中，在实际开发中,会为每个实体类创建一个对应的 Mapper.xml 文件，定义管理该对象的SQL。

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper
3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.gloryh.mapper.AccountMapper">
    <insert id="save"
parameterType="com.gloryh.entity.Account">
        INSERT INTO account(username,password,age) VALUES
        (#{username},#{password},#{age})
    </insert>
</mapper>

```

- namespace 属性：通常设置为文件所在包+文件名
- insert 标签：添加操作
- select 标签：查询操作
- update 标签：更新操作
- delete 标签：删除操作
- id 属性：被调用时用于对应方法调用
- parameterType 属性：对应的实体类

写完mapper文件后要在全局配置文件 (config.xml)中注册该mapper文件。

```

<!-- 注册 mapper文件 -->
<mappers>
    <mapper
resource="com/gloryh/mapper/AccountMapper.xml"></mapper>
</mappers>

```

在pom.xml添加build标签用于加载资源，让程序能够读取到配置文件。

```

<build>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>
                    **/*.xml
                </include>
            </includes>
        </resource>
    </resources>
</build>

```

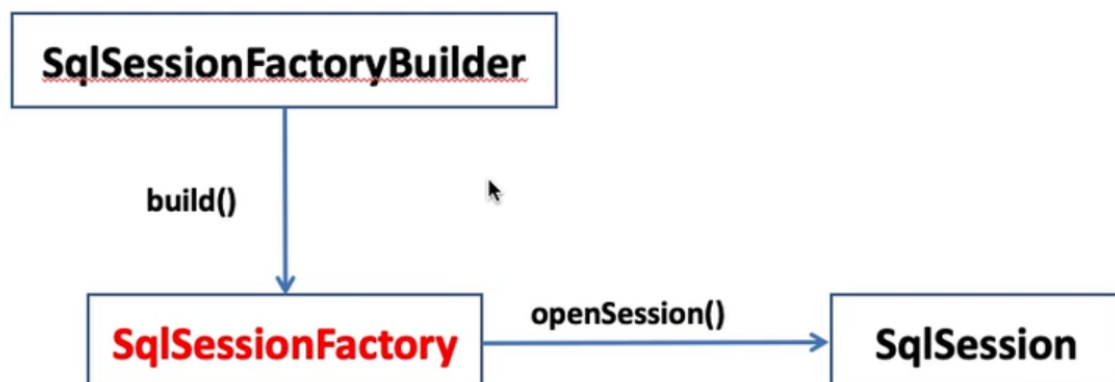
```

        </includes>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>
                *.xml
            </include>
            <include>
                *.properties
            </include>
        </includes>
    </resource>
</resources>
</build>

```

最后调用MyBatis 的原生接口执行添加操作。

MyBatis 核心接口和类



```

package com.gloryh.test;

import com.gloryh.entity.Account;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 测试类

```

```

*
* @author 黄光辉
* @since 2020/9/17
**/
public class Test {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("config.xml");

        SqlSessionFactoryBuilder factoryBuilder = new
SqlSessionFactoryBuilder();
        SqlSessionFactory
sessionFactory=factoryBuilder.build(stream);
        SqlSession session=sessionFactory.openSession();
        //namespace+id找到对应的方法
        String statement
="com.gloryh.mapper.AccountMapper.save";
        //实体化对象用于save
        Account account=new Account(1L,"张三","123",20);
        session.insert(statement,account);
        //执行save方法
        session.commit();
        //关闭资源
        session.close();
    }
}

```

运行结果:

对象

<

* 无标题 - 查询

account @mybatis_...

开始事务

文本

筛选

排序

导入

导出

id	username	password	age
1	张三	123	20

2、使用 Mapper 代理实现自定义接口

首先，要自定义接口，以及相关业务方法，然后编写与方法相对应的 mapper.xml。

- 自定义接口

```
package com.gloryh.repository;

import com.gloryh.entity.Account;

import java.util.List;

/**
 * 自动定义 repository 接口
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public interface AccountRepository {
    public int save(Account account );
    public int update(Account account);
    public int deleteById( long id);
    public List<Account> findAll();
    public Account findById(long id);
}
```

- 创建接口对应的 mapper.xml，定义接口方法对应的SQL语句。

statement 标签，例如 insert、update、delete、select，可根据 SQL 执行的业务来自行选择。

MyBatis 框架会根据规则自动创建接口实现类的代理对象。

规则：

mapper.xml 中 namespace 为接口的全类名。

mapper.xml 中 statement 标签的id为接口中对应的方法名。

mapper.xml 中 statement 标签的parameterType 和接口中对应方法的参数类型一致。

mapper.xml 中 statement 标签的 resultType 和接口中对应方法的返回值类型一致。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper
    namespace="com.gloryh.repository.AccountRepository"
>
    <insert id="save"
        parameterType="com.gloryh.entity.Account">
        INSERT INTO account(username,password,age)
        VALUES (#{username},#{password},#{age})
    </insert>
    <update id="update"
        parameterType="com.gloryh.entity.Account">
        UPDATE account SET username = #
        {username},password = #{password},age = #{age}
        WHERE id =#{id}
    </update>
    <delete id="deleteById"
        parameterType="java.lang.Long">
        DELETE FROM account WHERE id =#{id}
    </delete>

    <select id="findAll"
        resultType="com.gloryh.entity.Account">
        SELECT * FROM account
    </select>
    <select id="findById" parameterType="long"
        resultType="com.gloryh.entity.Account">
        SELECT * FROM account WHERE id = #{id}
    </select>

</mapper>
```

- 在 config.xml 中注册对应的 mapper.xml (和原生接口的注册方法一致)

```

<!-- 注册 mapper文件 -->
<mappers>
    <mapper
resource="com/gloryh/mapper/AccountMapper.xml">
</mapper>
    <mapper
resource="com/gloryh/repository/AccountRepository.xml"></mapper>
</mappers>

```

- 测试

```

package com.gloryh.test;

import com.gloryh.entity.Account;
import com.gloryh.repository.AccountRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;
import java.util.List;

/**
 * 测试mapper自定义接口
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public class Test2 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory
sessionFactory=factoryBuilder.build(stream);

```

```

SqlSession
session=sessionFactory.openSession();
    //获取实现接口的代理对象
    AccountRepository accountRepository
=session.getMapper(AccountRepository.class);
    //调用接口方法实现查询
    List<Account> accounts
=sessionRepository.findAll();
    for (Account account:accounts) {
        System.out.println(account);
    }
}
}

```

查询结果:

```

"E:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Account(id=1, username=张三, password=123, age=20)
Account(id=2, username=张三, password=123, age=20)

Process finished with exit code 0

```

数据库信息:

id	username	password	age
1	张三	123	20
2	张三	123	20

3、级联查询

- 一对多

需要的表: classes 和 student

对象	student @mybatis_demo (8....		
开始事务	文本	筛选	排序
id	name	c_id	
1	张三	2	
2	李四	2	
3	王五	2	

对象	classes @mybatis_demo (8....		
开始事务	文本	筛选	排序
id	name		
2	二班		

对象	classes @mybatis_demo (8....		
开始事务	文本	筛选	排序
id	name		
2	二班		

需要的体现级联的实体类：Student 和 Classes

```
package com.gloryh.entity;

import lombok.Data;

/**
 * 学生实体类
 */
```

```

* @author 黄光辉
* @since 2020/9/19
**/
@Data
public class Student {
    private long id;
    private String name;
    private Classes classes;
}

```

```

package com.gloryh.entity;

import lombok.Data;

import java.util.List;

/**
 * 班级实体类
 *
 * @author 黄光辉
 * @since 2020/9/19
 **/
@Data
public class Classes {
    private long id;
    private String name;
    private List<Student> students;
}

```

两表关联需要的 SQL 语句：

```

SELECT s.id,s.name,c.id,c.name FROM student s
,classes c WHERE s.id =1 AND s.c_id =c.id

```

查询的结果：

信息	Result 1	剖析	状态
id	name	id(1)	name(1)
1	张三	2	二班

定义接口类

```
package com.gloryh.repository;

import com.gloryh.entity.Student;

/**
 * 学生实体类操作接口
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public interface StudentRepository {
    public Student findById(long id);
}
```

定义对应mapper.xml，此时因为是级联查询，需要使用 resultMap 标签实现级联关系的对应，同时使用 as 字段对重复的字段进行区分：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper
    namespace="com.gloryh.repository.StudentRepository"
>
    <resultMap id="studentMap"
    type="com.gloryh.entity.Student">
        <id column="id" property="id"></id>
        <result column="name" property="name">
</result>
```

```

        <association property="classes"
javaType="com.gloryh.entity.Classes">
            <id column="cid" property="id"></id>
            <result column="cname" property="name">
</result>
        </association>
    </resultMap>
    <select id="findById" parameterType="long"
resultMap="studentMap">
        SELECT s.id,s.name,c.id AS cid,c.name AS
cname FROM student s ,classes c WHERE s.id =#{id}
AND s.c_id =c.id
    </select>
</mapper>

```

注册对应的mapper.xml

```

<!-- 注册 mapper文件 -->
<mappers>
    <mapper
resource="com/gloryh/mapper/AccountMapper.xml">
</mapper>
    <mapper
resource="com/gloryh/repository/AccountRepository.x
ml"></mapper>
    <mapper
resource="com/gloryh/repository/StudentRepository.x
ml"></mapper>
</mappers>

```

测试方法:

```

package com.gloryh.test;

import com.gloryh.repository.AccountRepository;
import com.gloryh.repository.StudentRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

```

```

import java.io.InputStream;

/**
 * 级联测试
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public class Test3 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        StudentRepository studentRepository =
session.getMapper(StudentRepository.class);
        //调用接口方法实现查询

        System.out.println(studentRepository.findById(1L))
;
        session.close();
    }
}

```

运行结果：

```

"E:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Student(id=1, name=张三, classes=Classes(id=2, name=二班, students=null))

进程已结束,退出代码0

```

- 多对一

两表关联需要的 SQL 语句:


```
SELECT s.id,s.name,c.id,c.name FROM student s
,classes c WHERE c.id =2 AND c.id=s.c_id
```

查询结果:

信息	Result 1	剖析	状态
id	name	id(1)	name(1)
1	张三	2	二班
2	李四	2	二班
3	王五	2	二班

定义接口类:

```
package com.gloryh.repository;

import com.gloryh.entity.Classes;

/**
 * 班级实体类操作接口
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public interface ClassesRepository {
    public Classes findById(long id);
}
```

定义对应mapper.xml, 此时因为是级联查询, 需要使用 resultMap 标签实现级联关系的对应, 同时使用 as 字段对重复的字段进行区分:

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

```

<mapper
namespace="com.gloryh.repository.ClassesRepository"
>
    <resultMap id="classesMap"
type="com.gloryh.entity.Classes">
        <id column="cid" property="id"></id>
        <result column="cname" property="name">
</result>
        <collection property="students"
ofType="com.gloryh.entity.Student">
            <id column="id" property="id"></id>
            <result column="name" property="name">
</result>
        </collection>
    </resultMap>
    <select id="findById" parameterType="long"
resultMap="classesMap" >
        SELECT s.id,s.name,c.id AS cid,c.name AS
cname FROM student s ,classes c WHERE c.id =#{id}
AND c.id=s.c_id
    </select>
</mapper>

```

注册对应的mapper.xml

```

<!-- 注册 mapper文件 -->
<mappers>
    <mapper
resource="com/gloryh/mapper/AccountMapper.xml">
</mapper>
    <mapper
resource="com/gloryh/repository/AccountRepository.x
ml"></mapper>
    <mapper
resource="com/gloryh/repository/StudentRepository.x
ml"></mapper>
    <mapper
resource="com/gloryh/repository/ClassesRepository.x
ml"></mapper>
</mappers>

```

测试方法

```
package com.gloryh.test;

import com.gloryh.repository.AccountRepository;
import com.gloryh.repository.ClassesRepository;
import com.gloryh.repository.StudentRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 级联测试
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public class Test3 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        ClassesRepository classesRepository
=session.getMapper(ClassesRepository.class);
        //调用接口方法实现查询

        System.out.println(classesRepository.findById(2L))
;
        System.out.println();
    }
}
```

```

        session.close();
    }
}

```

运行结果：

```

"E:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Classes(id=2, name=班, students=[Student(id=1, name=张三, classes=null), Student(id=2, name=李四, classes=null), Student(id=3, name=王五, classes=null)])

```

- 多对多：由两个一对多形成一个多对多

需要的表，两个主表customer和goods，一个关联表customer_goods

对象	customer @mybatis_demo (...)
开始事务	文本 筛选 排序
id	name
1	张三
3	小明

对象	goods @mybatis_demo (8.0...)
开始事务	文本 筛选 排序
id	name
1	电视
2	电脑
3	洗衣机

对象		customer_goods @mybatis_...	
开始事务		文本	筛选 排序
	id	c_id	g_id
	1	1	1
	2	1	3
	3	3	2
▶	22	3	3

对应的实体类 Customer 和 Goods

```
package com.gloryh.entity;

import lombok.Data;

import java.util.List;

/**
 * 客户实体类
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
@Data
public class Customer {
    private long id;
    private String name;
    private List<Goods> goods;
}
```

```
package com.gloryh.entity;

import lombok.Data;

import java.util.List;

/**
```

```

* 商品实体类
*
* @author 黄光辉
* @since 2020/9/19
**/
@Data
public class Goods {
    private long id;
    private String name;
    private List<Customer> customers;
}

```

定义接口类:

```

package com.gloryh.repository;

import com.gloryh.entity.Customer;

/**
 * 客户实体类操作接口
 *
 * @author 黄光辉
 * @since 2020/9/19
 **/
public interface CustomerRepository {
    public Customer findById(long id);
}

```

需要用到的 SQL 语句:

```

SELECT c.id,c.name ,g.id,g.name FROM customer c
,goods g ,customer_goods cg WHERE c.id = 1 AND
cg.c_id=c.id AND cg.g_id=g.id

```

查询结果:

信息

Result 1

剖析

状态

id	name	id(1)	name(1)
1	张三	1	电视
1	张三	3	洗衣机

定义对应mapper.xml，此时因为是级联查询，需要使用 resultMap 标签实现级联关系的对应，同时使用 as 字段对重复的字段进行区分：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper
    namespace="com.gloryh.repository.CustomerRepository">
    <resultMap id="customerMap"
        type="com.gloryh.entity.Customer">
        <id column="cid" property="id"></id>
        <result column="cname" property="name">
    </result>
        <collection property="goods"
            ofType="com.gloryh.entity.Goods">
            <id column="id" property="id"></id>
            <result column="name" property="name">
    </result>
        </collection>
    </resultMap>
    <select id="findById" parameterType="long"
        resultMap="customerMap">
        SELECT c.id AS cid,c.name AS cname
        ,g.id,g.name FROM customer c ,goods g
        ,customer_goods cg WHERE c.id = #{id} AND
        cg.c_id=c.id AND cg.g_id=g.id
    </select>
</mapper>
```

注册对应的 mapper.xml

```
<!-- 注册 mapper文件 -->
<mappers>
    <mapper
resource="com/gloryh/mapper/AccountMapper.xml">
</mapper>
    <mapper
resource="com/gloryh/repository/AccountRepository.x
ml"></mapper>
    <mapper
resource="com/gloryh/repository/StudentRepository.x
ml"></mapper>
    <mapper
resource="com/gloryh/repository/ClassesRepository.x
ml"></mapper>
    <mapper
resource="com/gloryh/repository/CustomerRepository.
xml"></mapper>
</mappers>
```

测试方法:

```
package com.gloryh.test;

import com.gloryh.repository.AccountRepository;
import com.gloryh.repository.ClassesRepository;
import com.gloryh.repository.CustomerRepository;
import com.gloryh.repository.StudentRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 级联测试
 *
 * @author 黄光辉
 * @since 2020/9/19
```



```

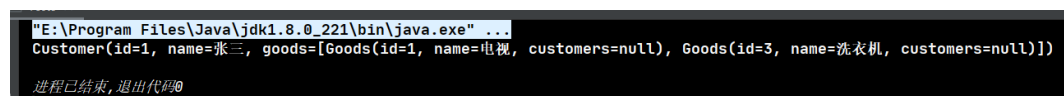
    /**/
    public class Test3 {
        public static void main(String[] args) {
            //加载MyBatis配置文件
            InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

            SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
            SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
            SqlSession session =
sessionFactory.openSession();
            CustomerRepository customerRepository
=session.getMapper(CustomerRepository.class);

            System.out.println(customerRepository.findById(1L)
);
            session.close();
        }
    }
}

```

运行结果：



```

"E:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Customer(id=1, name=张三, goods=[Goods(id=1, name=电视, customers=null), Goods(id=3, name=洗衣机, customers=null)])
进程已结束, 退出代码0

```

三、逆向工程

MyBatis 框架需要：实体类、自定义Mapper接口、Mapper.xml

传统的开发中，以上三个组件需要开发者手动创建，逆向工程可以帮助开发者开自动创建三个组件，减轻开发者的工作量，提高工作效率。

使用方法

MyBatis Generator，简称 MBG，是一个专门为 MyBatis 框架的开发者定制的代码生成器，可以自动生成 MyBatis 框架所需的实体类、Mapper接口、Mapper.xml 文件，支持基本的 CRUD 操作，但是一些相对复杂的SQL需要开发者自己来完成。

- 首先, 新建工程, pom.xml中引入相关依赖

```
<dependencies>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.5</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-
java</artifactId>
        <version>8.0.20</version>
    </dependency>

    <dependency>
        <groupId>org.mybatis.generator</groupId>
        <artifactId>mybatis-generator-
core</artifactId>
        <version>1.3.5</version>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.12</version>
    </dependency>
</dependencies>
```

- 创建 MBG 配置文件 generatorConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration PUBLIC
"-//mybatis.org//DTD MyBatis Generator
Configuration 1.0//EN"
"http://mybatis.org/dtd/mybatis-generator-
config_1_0.dtd">
<generatorConfiguration>
    <context id="testTables"
targetRuntime="MyBatis3">
```

```

<jdbcConnection

driverClass="com.mysql.cj.jdbc.Driver"

connectionURL="jdbc:mysql://localhost:3305/mybatis_demo?
useUnicode=true&characterEncoding=utf8&serverTimezone=UTC&allowMultiQueries=true"
        userId="admin"
        password="123"
    >
</jdbcConnection>
<javaModelGenerator
        targetPackage="com.gloryh.entity"
        targetProject="./src/main/java">
</javaModelGenerator>
<sqlMapGenerator

targetPackage="com.gloryh.repository"
        targetProject="./src/main/java">
</sqlMapGenerator>
<javaClientGenerator
        type="XMLMAPPER"

targetPackage="com.gloryh.repository"
        targetProject="./src/main/java">
</javaClientGenerator>
    <table tableName="user"
domainObjectName="User"></table>
    </context>
</generatorConfiguration>

```

需要的配置：

jdbcConnection ： 配置数据库连接信息

javaModelGenerator ： 配置JavaBean的生成策略

sqlMapGenerator ： 配置 SQL 映射文件的生成策略

javaClientGenerator ： 配置 Mapper 接口的生成策略

table ： 配置目标数据表（通过 table 配置表名，
domainObjName 配置 JavaBean 类名）

用到的数据表:

对象

user @mybatis_demo (8.0) -...

开始事务

文本

筛选

排序

	id	username	password	age
	1	张三	111	21
	2	李四	222	22
	3	王五	333	23
▶	4	赵六	444	24

- 创建 Generator 执行的类

```
package com.gloryh.test;

import org.mybatis.generator.api.MyBatisGenerator;
import org.mybatis.generator.config.Configuration;
import org.mybatis.generator.config.xml.ConfigurationParser;
import org.mybatis.generator.exception.InvalidConfigurationException;
import org.mybatis.generator.exception.XMLParserException;
import org.mybatis.generator.internal.DefaultShellCallback;

import java.io.File;
import java.io.IOException;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

/**
 * Generator 执行的类 == 测试方法
 */
```

```

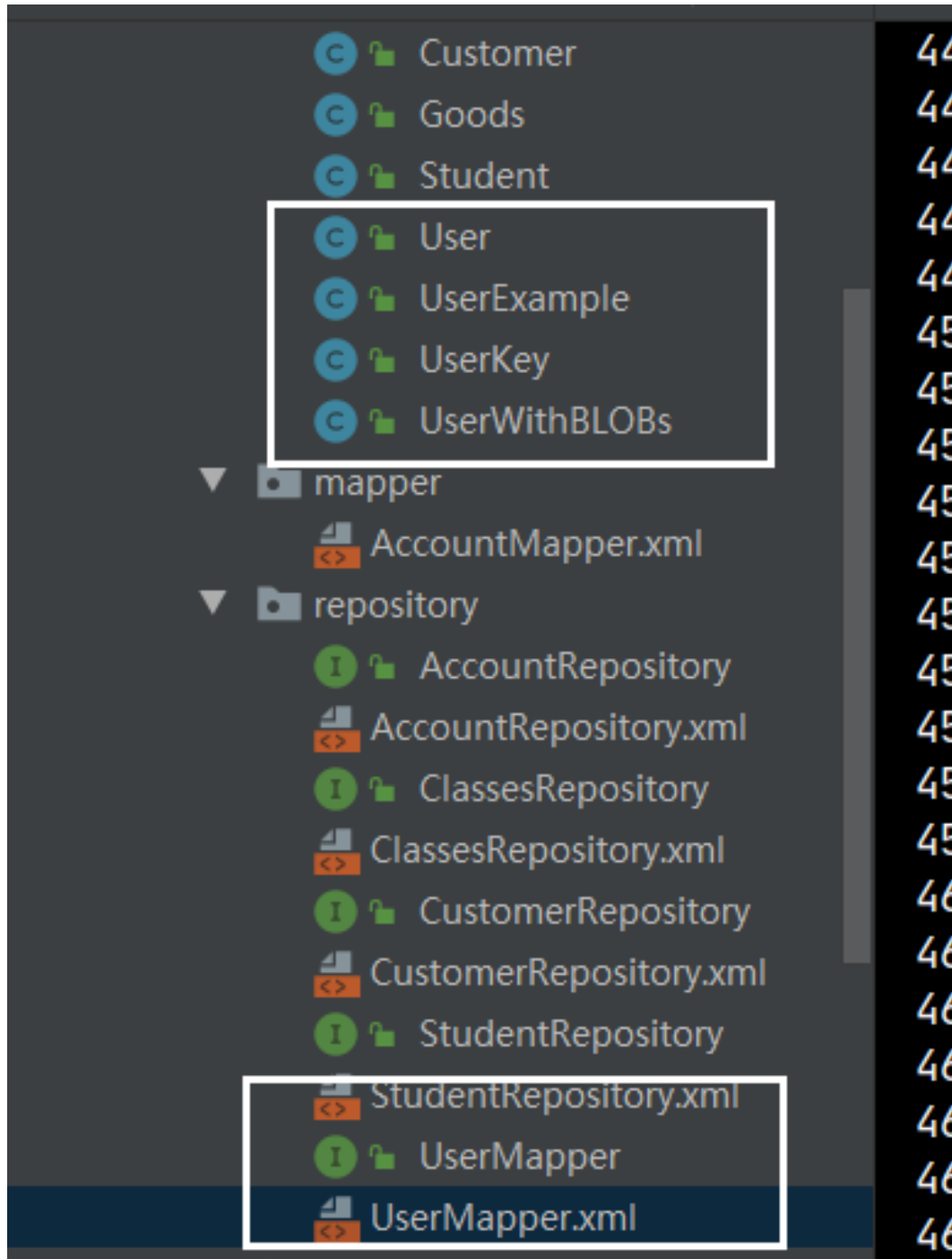
* @author 黄光辉
* @since 2020/9/19
**/
public class Test4 {
    public static void main(String[] args) {
        List<String> warings =new ArrayList<String>
();
        boolean overwirte =true;
        String
generatorConfig="/generatorConfig.xml";
        File configFile =new File("E:\\Program
Files\\IDEA
workspace\\mybatis_demo\\src\\main\\resources\\gene
ratorConfig.xml");
        ConfigurationParser configurationParser
=new ConfigurationParser(warings);
        Configuration configuration =null;
        try {

            configuration=configurationParser.parseConfigurati
on(configFile);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (XMLParserException e) {
            e.printStackTrace();
        }
        DefaultShellCallback callback =new
DefaultShellCallback(overwirte);
        MyBatisGenerator myBatisGenerator=null;
        try {
            myBatisGenerator=new
MyBatisGenerator(configuration,callback,warings);
        } catch (InvalidConfigurationException e) {
            e.printStackTrace();
        }
        try {
            myBatisGenerator.generate(null);
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
}
```

自动生成结果：



四、延迟加载

延迟加载：也被称为懒加载或者惰性加载，它可以提高程序的运行效率。

针对于数据持久层的操作，即在某些特定的情况下去访问特定的数据库时，有时并不需要访问某些表，从一定程度上减少了Java应用与数据库的交互次数。

例如之前我们在查询学生和班级信息时，由于两张表是一对多的级联关系，所以查询时会查询两张表的信息，而有时我们只需要获取学生信息即可满足需求，即只需要查询一张学生表里的对应信息，而不需要该学生信息对应的班级信息。

不同的业务需求，需要查询不同的表，根据具体的业务需求来动态地减少数据表的查询，这就是延迟加载要完成的工作。

- 拆表

之前查询学生信息的SQL为：

```
SELECT s.id,s.name,c.id as cid,c.name as cname FROM  
student s ,classes c WHERE s.id =#{id} AND s.c_id  
=c.id
```

之前查询学生信息的mapper.xml为：

```

<resultMap id="studentMap"
type="com.gloryh.entity.Student">
    <id column="id" property="id"></id>
    <result column="name" property="name">
</result>
    <association property="classes"
javaType="com.gloryh.entity.Classes">
        <id column="cid" property="id"></id>
        <result column="cname" property="name">
</result>
    </association>
</resultMap>
<select id="findById" parameterType="long"
resultMap="studentMap">
    SELECT s.id,s.name,c.id as cid,c.name as
cname FROM student s ,classes c WHERE s.id =#{id}
AND s.c_id =c.id
</select>

```

我们在学生实体类操作接口新建一个findByIdLazy接口用于实现延迟加载：

```

package com.gloryh.repository;

import com.gloryh.entity.Student;

/**
 * 学生实体类操作接口
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public interface StudentRepository {
    public Student findById(long id);
    public Student findByIdLazy(long id);
}

```

学生信息单表查询SQL：

```
SELECT * FROM student WHERE id =#{id}
```


在学生信息对应的 mapper.xml 文件中配置该查询操作：

```
<select id="findByIdLazy" parameterType="long"
resultType="com.gloryh.entity.Student">
    SELECT * FROM student WHERE id =#{id}
</select>
```

在班级实体类操作接口新建一个findByIdLazy接口用于实现延迟加载：

```
package com.gloryh.repository;

import com.gloryh.entity.Classes;

/**
 * 班级实体类操作接口
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public interface ClassesRepository {
    public Classes findById(long id);
    public Classes findByIdLazy(long id);
}
```

班级信息单表查询SQL：

```
SELECT * FROM classes WHERE id =#{id}
```

在班级信息对应的 mapper.xml 文件中配置该查询操作：

```
<select id="findByIdLazy" parameterType="long"
resultType="com.gloryh.entity.Classes" >
    SELECT * FROM classes WHERE id =#{id}
</select>
```

测试分表是否完成：

```
package com.gloryh.test;

import com.gloryh.repository.AccountRepository;
```

```
import com.gloryh.repository.ClassesRepository;
import com.gloryh.repository.CustomerRepository;
import com.gloryh.repository.StudentRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 级联测试
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public class Test3 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        StudentRepository studentRepository =
session.getMapper(StudentRepository.class);
        ClassesRepository classesRepository
=session.getMapper(ClassesRepository.class);
        //调用接口方法实现查询

        System.out.println(studentRepository.findByIdLazy(
1L));

        System.out.println(classesRepository.findByIdLazy(
2L));
```

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

运行结果显示拆表已完成：

```
"E:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ..
Student(id=1, name=张三, classes=null)
Classes(id=2, name=二班, students=null)

Process finished with exit code 0
```

此时我们要完成通过学生信息查询班级信息需要配置resultMap接收对应的外键 `c_id`：

```
<resultMap id="studentMapLazy"
type="com.gloryh.entity.Student">
    <id column="id" property="id"></id>
    <result column="name" property="name"></result>
    <association property="classes"
javaType="com.gloryh.entity.Classes">
        <id column="c_id" property="id"></id>
        <result column="cname" property="name">
</result>
    </association>
</resultMap>
<select id="findByIdLazy" parameterType="long"
resultMap="studentMapLazy">
    SELECT * FROM student WHERE id =#{id}
</select>
```

通过 `student.getClasses().getId()` 来完成对对应班级信息的查询：

```
package com.gloryh.test;

import com.gloryh.entity.Student;
import com.gloryh.repository.AccountRepository;
import com.gloryh.repository.ClassesRepository;
import com.gloryh.repository.CustomerRepository;
import com.gloryh.repository.StudentRepository;
import org.apache.ibatis.session.SqlSession;
```

```
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 级联测试
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public class Test3 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        StudentRepository studentRepository =
session.getMapper(StudentRepository.class);
        ClassesRepository classesRepository
=session.getMapper(ClassesRepository.class);
        //调用接口方法实现查询
        Student student
=studentRepository.findByIdLazy(1L);
        System.out.println(student );

        System.out.println(classesRepository.findByIdLazy(
student.getClasses().getId()));
        session.close();
    }
}
```

运行结果：

```
"E:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...  
Student(id=1, name=张三, classes=Classes(id=2, name=null, students=null))  
Classes(id=2, name=二班, students=null)
```

再这样拆分之后，我们在只需要班级信息时，就只需要执行一次对学生信息表的查询即可满足需求。如果需要对应的班级信息时，我们再通过外键执行一次对应班级信息的查询来来满足需求。这就为延迟加载提供了实现的可能。

- 实现延迟加载

在config.xml添加设置观察执行的SQL语句：

```
<settings>  
    <!-- 打印 SQL 语句 -->  
    <setting name="logImp1" value="STDOUT_LOGGING"/>  
</settings>
```

我们要想让程序自动控制是否进行延迟加载，只需要将对应的 resultMap 中的 association 标签添加 对应的标签即可，例如我们现在执行的查询操作，需要使用 select 标签 来指定延迟加载要对应的查询接口和使用 column 标签 来对应的查询条件：

```
<resultMap id="studentMapLazy"  
type="com.gloryh.entity.Student">  
    <id column="id" property="id"></id>  
    <result column="name" property="name"></result>  
    <association property="classes"  
javaType="com.gloryh.entity.Classes"  
select="com.gloryh.repository.ClassesRepository.findByIdLazy" column="c_id">  
        </association>  
</resultMap>  
<select id="findByIdLazy" parameterType="long"  
resultMap="studentMapLazy">  
    SELECT * FROM student WHERE id =#{id}  
</select>
```

测试方法：

```
package com.gloryh.test;
```

```

import com.gloryh.entity.Student;
import com.gloryh.repository.AccountRepository;
import com.gloryh.repository.ClassesRepository;
import com.gloryh.repository.CustomerRepository;
import com.gloryh.repository.StudentRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 级联测试
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public class Test3 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        StudentRepository studentRepository =
session.getMapper(StudentRepository.class);
        //调用接口方法实现查询
        Student student
=studentRepository.findByIdLazy(1L);
        System.out.println(student );
        session.close();
    }
}

```

运行结果：

```
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2797d0c7]
==> Preparing: SELECT * FROM student WHERE id =?
==> Parameters: 1(Long)
<==      Columns: id, name, c_id
<==      Row: 1, 张三, 2
====> Preparing: SELECT * FROM classes WHERE id =?
====> Parameters: 2(Long)
<====      Columns: id, name
<====      Row: 2, 二班
<====      Total: 1
<==      Total: 1
Student(id=1, name=张三, classes=Classes(id=2, name=二班, students=null))
```

通过运行结果的日志可以观察到，此时我们获取学生信息执行了两个SQL，分别查询学生信息和对应的班级信息因为学生信息包括对应的班级信息。

但是如果只需要查询例如学生的姓名而不需要班级信息，此时我们再查询班级信息就会造成不必要的资源浪费，这个时候我们就需要用到延迟加载。

- 在 config.xml 文件内添加配置开启延迟加载

我们在使用 settings 属性内的 setting 标签 来进行延迟加载的配置：

```
<settings>
    <!-- 打印 SQL 语句 -->
    <setting name="logImpl" value="STDOUT_LOGGING"/>
    <!-- 开启延迟加载，默认为不开启，value 为 false -->
    <setting name="lazyLoadingEnabled"
value="true"/>
</settings>
```

查询学生的姓名而不需要学生信息对应的班级信息：

```
package com.gloryh.test;

import com.gloryh.entity.Student;
import com.gloryh.repository.AccountRepository;
import com.gloryh.repository.ClassesRepository;
import com.gloryh.repository.CustomerRepository;
import com.gloryh.repository.StudentRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;
```

```

import java.io.InputStream;

/**
 * 级联测试
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public class Test3 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        StudentRepository studentRepository =
session.getMapper(StudentRepository.class);
        //调用接口方法实现查询
        Student student
=studentRepository.findByIdLazy(1L);
        System.out.println(student.getName());
        session.close();
    }
}

```

开启前(执行两次 SQL 语句):


```

Setting autocommit to false on JDBC Connection [com.mys
==> Preparing: SELECT * FROM student WHERE id =?
==> Parameters: 1(Long)
<==      Columns: id, name, c_id
<==      Row: 1, 张三, 2
====> Preparing: SELECT * FROM classes WHERE id =?
====> Parameters: 2(Long)
<====      Columns: id, name
<====      Row: 2, 二班
<====      Total: 1
<==      Total: 1
张三

```

开启后（仅执行一次 SQL 语句）：

```

Opening JDBC Connection
Created connection 798981583.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2f9f7dcf]
==> Preparing: SELECT * FROM student WHERE id =?
==> Parameters: 1(Long)
<==      Columns: id, name, c_id
<==      Row: 1, 张三, 2
<==      Total: 1
张三
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2f9f7dcf]

```

当我们需要查询对应班级信息时：

```

package com.gloryh.test;

import com.gloryh.entity.Student;
import com.gloryh.repository.AccountRepository;
import com.gloryh.repository.ClassesRepository;
import com.gloryh.repository.CustomerRepository;
import com.gloryh.repository.StudentRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 级联测试
 *
 * @author 黄光辉
 * @since 2020/9/19
 */
public class Test3 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
    }
}

```

```

        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        StudentRepository studentRepository =
session.getMapper(StudentRepository.class);
        //调用接口方法实现查询
        Student student
=studentRepository.findByIdLazy(1L);
        System.out.println(student.getClasses());
        session.close();
    }
}

```

运行结果（仍执行两次 SQL 语句）：

```

Setting autocommit to false on JDBC Connection [com.mysql
==> Preparing: SELECT * FROM student WHERE id =?
==> Parameters: 1(Long)
<== Columns: id, name, c_id
<== Row: 1, 张三, 2
<== Total: 1
==> Preparing: SELECT * FROM classes WHERE id =?
==> Parameters: 2(Long)
<== Columns: id, name
<== Row: 2, 二班
<== Total: 1
Classes(id=2, name=二班, students=null)

```

五、MyBatis 缓存（重点）

MyBatis 缓存：使用 MyBatis 缓存可以减少 Java 应用与数据库之间的交互次数，从而提升程序的运行效率。

比如我们查询 id = 1 的对象，在第一次查询完成后，会自动将该对象保存到缓存中，当下次查询该对象时，会直接从缓存中将对象取出作为查询结果，而不需要再次对数据库进行访问和交互。

1、缓存的分类

- 一级缓存：SQLSession 级别，默认为开启状态，且不能关闭。

在我们操作数据库时，需要创建 SQLSession 对象，再改对象中有一个 HashMap 用于存储缓存数据，不同的 SQLSession 之间的缓存数据区域互不影响。

一级缓存作用域范围：SQLSession 范围。

当在同一个 SQLSession 中执行两次相同的 SQL 语句时，第一次执行回合数据库进行交互，但在第一次执行完毕后，会将结果保存到缓存中，第二次查询时会直接从缓存中读取。

注意

如果 SQLSession 执行了 DML 操作（即 insert、update、delete）之后，MyBatis 会将缓存清空以保证数据的准确性。

- 二级缓存：Mapper级别，默认为关闭状态，可以手动选择开启。

当二级缓存开启之后，多个 SQLSession 使用同一个 Mapper 的 SQL 语句操作数据库，得到的数据会缓存在二级缓存区，二级缓存同样也是使用 HashMap 进行数据存储，相比较于一级缓存，二级缓存的范围更大，优先级也更小。

多个 SQLSession 可以共用二级缓存，即二级缓存是跨 SQLSession 的。

二级缓存的作用域范围：由于二级缓存是多个 SQLSession 共享的，所以其作用域范围是 Mapper 文件的相同的

`<namespace/>`。

不同的 SQLSession 两次执行相同的 `<namespace/>` 下的 SQL 语句，参数也相等，则第一次执行成功之后会将数据保存到二级缓存中，第二次可直接从二级缓存中读取数据。

2、代码中的具体提现

- 一级缓存

对一个对象同时查询两次：

```
package com.gloryh.test;
```

```
import com.gloryh.entity.Account;
import com.gloryh.repository.AccountRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 测试 MyBatis 缓存
 *
 * @author 黄光辉
 * @since 2020/9/21
 */
public class Test5 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();

        AccountRepository accountRepository =
session.getMapper(AccountRepository.class);
        Account account =
accountRepository.findById(1L);
        System.out.println(account);
        Account account1 =
accountRepository.findById(1L);
        System.out.println(account1);
    }
}
```

查看结果（只执行了一次 SQL 语句）：

```
Created connection 798981585.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2f1e1e1e]
==> Preparing: SELECT * FROM account WHERE id = ?
==> Parameters: 1(Long)
<==      Columns: id, username, password, age
<==      Row: 1, 张三, 123, 20
<==      Total: 1
Account(id=1, username=张三, password=123, age=20)
Account(id=1, username=张三, password=123, age=20)
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2f1e1e1e]
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2f1e1e1e]
```

如果我们执行一次查询后关闭，然后再执行一次查询：

```
package com.gloryh.test;

import com.gloryh.entity.Account;
import com.gloryh.repository.AccountRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 测试 MyBatis 缓存
 *
 * @author 黄光辉
 * @since 2020/9/21
 */
public class Test5 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
    }
}
```

```

        SqlSession session =
sessionFactory.openSession();

        AccountRepository accountRepository =
session.getMapper(AccountRepository.class);
        Account account =
accountRepository.findById(1L);
        System.out.println(account);
        session.close();
        SqlSession session1 =
sessionFactory.openSession();
        accountRepository =
session1.getMapper(AccountRepository.class);
        Account account1 =
accountRepository.findById(1L);
        System.out.println(account1);
        session.close();
    }
}

```

查看结果（执行了两次同样的查询）：

```

Opening JDBC Connection
Created connection 798981583.
Setting autocommit to false on JDBC Connection [com.my
==> Preparing: SELECT * FROM account WHERE id = ?
==> Parameters: 1(Long)
<==      Columns: id, username, password, age
<==      Row: 1, 张三, 123, 20
<==      Total: 1
Account(id=1, username=张三, password=123, age=20)
Resetting autocommit to true on JDBC Connection [com.m
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionI
Returned connection 798981583 to pool.
Opening JDBC Connection
Checked out connection 798981583 from pool.
Setting autocommit to false on JDBC Connection [com.my
==> Preparing: SELECT * FROM account WHERE id = ?
==> Parameters: 1(Long)
<==      Columns: id, username, password, age
<==      Row: 1, 张三, 123, 20
<==      Total: 1
Account(id=1, username=张三, password=123, age=20)

```

- 二级缓存

MyBatis 自带的二级缓存

在 config.xml 中配置开启二级缓存。

```
<settings>
    <!-- 打印 SQL 语句 -->
    <setting name="logImpl"
value="STDOUT_LOGGING"/>
    <!-- 开启延迟加载，默认为不开启，value 为 false -->
    <setting name="lazyLoadingEnabled"
value="true"/>
    <!-- 开启二级缓存，默认为不开启，value 为 false -->
    <setting name="cacheEnabled" value="true"/>
</settings>
```

在对应的 mapper.xml 中配置二级缓存（添加 `<cache>` `</cache>`）。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper
3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-
mapper.dtd">
<mapper
namespace="com.gloryh.repository.AccountRepository"
>
    <cache></cache>
    <insert id="save"
parameterType="com.gloryh.entity.Account">
        INSERT INTO account(username,password,age)
VALUES (#{username},#{password},#{age})
    </insert>
    <update id="update"
parameterType="com.gloryh.entity.Account">
        UPDATE account SET username = #
{username},password = #{password},age = #{age}
WHERE id =#{id}
    </update>
    <delete id="deleteById"
parameterType="java.lang.Long">
        DELETE FROM account WHERE id =#{id}
    </delete>
```

```

        <select id="findAll"
resultType="com.gloryh.entity.Account">
            SELECT * FROM account
        </select>
        <select id="findById" parameterType="long"
resultType="com.gloryh.entity.Account">
            SELECT * FROM account WHERE id = #{id}
        </select>
    </mapper>

```

在对应的实体类中实现序列化接口（Serializable）。

```

package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.io.Serializable;

/**
 * 实体类Account
 *
 * @author 黄光辉
 * @since 2020/9/17
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Account implements Serializable {
    private long id;
    private String username;
    private String password;
    private int age;
}

```

配置完成后，我们刚刚在执行一次查询后关闭，然后再执行一次查询，会查询两次，我们再运行一次之前的测试，查看结果，会发现即使我们将 SQLSession关闭，由于开启了二级缓存，还是只执行了一次 SQL 语句：


```

Opening JDBC Connection
Created connection 1438030319.
Setting autocommit to false on JDBC Connection [com.m
==> Preparing: SELECT * FROM account WHERE id = ?
==> Parameters: 1(Long)
<==      Columns: id, username, password, age
<==      Row: 1, 张三, 123, 20
<==      Total: 1
Account(id=1, username=张三, password=123, age=20)
Resetting autocommit to true on JDBC Connection [com.r
Closing JDBC Connection [com.mysql.cj.jdbc.Connection]
Returned connection 1438030319 to pool.
Cache Hit Ratio [com.gloryh.repository.AccountRepository]
Account(id=1, username=张三, password=123, age=20)

```

第三方二级缓存：ehcache

在 pom.xml 中添加相关依赖。

```

<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.0.0</version>
</dependency>
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache-core</artifactId>
    <version>2.6.11</version>
</dependency>

```

添加相关的配置文件 ehcache.xml。

```

<?xml version="1.0" encoding="UTF-8"?>
<ehcache
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

    xsi:noNamespaceSchemaLocation="../config/ehcache.xs
d">
    <!-- 磁盘保存路径 -->
    <diskStore/>

    <defaultCache
        maxElementsInMemory="1000"

```

```

        maxElementsOnDisk="10000000"
        eternal="false"
        overflowToDisk="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU">
    </defaultCache>
</ehcache>

```

在 config.xml 中配置开启二级缓存。

```

<settings>
    <!-- 打印 SQL 语句 -->
    <setting name="logImpl"
value="STDOUT_LOGGING"/>
    <!-- 开启延迟加载，默认为不开启，value 为 false -->
    <setting name="lazyLoadingEnabled"
value="true"/>
    <!-- 开启二级缓存，默认为不开启，value 为 false -->
    <setting name="cacheEnabled" value="true"/>
</settings>

```

在对应的 mapper.xml 中配置二级缓存（添加 `<cache>`
`</cache>`，并添加一些相应的配置信息）。

```

<cache
type="org.mybatis.caches.ehcache.EhcacheCache">
    <!-- 缓存创建之后最后一次访问缓存的时间 至 缓存失效
的时间间隔，以秒为单位 -->
    <property name="timeToIdleSeconds"
value="3600"/>
    <!-- 缓存自创建时间起 至 失效的时间间隔，疫苗为单位
-->
    <property name="timeToLiveSeconds"
value="3600"/>
    <!-- 垃圾回收策略，最近最少使用算法：LRU -->
    <property name="memoryStoreEvictionPolicy"
value="LRU"/>
</cache>

```

在对应的实体类中不需要实现序列化接口（Serializable）。

```
package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * 实体类Account
 *
 * @author 黄光辉
 * @since 2020/9/17
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Account {
    private long id;
    private String username;
    private String password;
    private int age;
}
```

配置完成后，我们刚刚在执行一次查询后关闭，然后再执行一次查询，会查询两次，我们再运行一次之前的测试，查看结果，会发现即使我们将 SQLSession关闭，由于开启了二级缓存，还是只执行了一次 SQL 语句：

```
Created connection 1757880885.
Setting autocommit to false on JDBC Connection [com.mysql
==> Preparing: SELECT * FROM account WHERE id = ?
==> Parameters: 1(Long)
<==      Columns: id, username, password, age
<==      Row: 1, 张三, 123, 20
<==      Total: 1
Account(id=1, username=张三, password=123, age=20)
Resetting autocommit to true on JDBC Connection [com.mys
Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImp
Returned connection 1757880885 to pool.
Cache Hit Ratio [com.gloryh.repository.AccountRepository
Account(id=1, username=张三, password=123, age=20)
```

六、MyBatis 动态 SQL

使用动态 SQL 可以简化代码的开发，减少开发者的工作量，程序可以自动根据业务参数来决定 SQL 语句的组成。

- 动态 SQL 的作用

新建一个按照 Account 全部属性来查询数据库内都与之相匹配的查询接口：

```
public Account findByAccount(Account account);
```

新建对应的 mapper.xml 参数：

```
<select id="findByAccount"
parameterType="com.gloryh.entity.Account"
resultType="com.gloryh.entity.Account">
    SELECT * FROM account WHERE id=#{id} AND
    username = #{username} AND password = #{password}
    AND age = #{age}
</select>
```

方法调用查询：

```
package com.gloryh.test;

import com.gloryh.entity.Account;
import com.gloryh.repository.AccountRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 动态 SQL 测试类
 *
 * @author 黄光辉
 * @since 2020/9/22
 */
public class Test6 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
```

```

        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        AccountRepository accountRepository =
session.getMapper(AccountRepository.class);
        //调用接口方法实现查询
        Account account = new Account(1L, "张三",
"123", 20);
        Account account1 =
accountRepository.findByAccount(account);
        System.out.println(account1);
    }
}

```

查询结果：

```

Cache hit ratio [com.gloryh.repository.AccountRepository]: 0.0
Opening JDBC Connection
Created connection 1757880885.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@68c72235]
==> Preparing: SELECT * FROM account WHERE id=? AND username = ? AND password = ? AND age = ?
==> Parameters: 1(Long), 张三(String), 123(String), 20(Integer)
<==      Columns: id, username, password, age
<==      Row: 1, 张三, 123, 20
<==      Total: 1
Account(id=1, username=张三, password=123, age=20)

```

当我们去掉一个赋值（例如password属性）：

```

package com.gloryh.test;

import com.gloryh.entity.Account;
import com.gloryh.repository.AccountRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

```

```

/**
 * 动态 SQL 测试类
 *
 * @author 黄光辉
 * @since 2020/9/22
 **/
public class Test6 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        AccountRepository accountRepository =
session.getMapper(AccountRepository.class);
        //调用接口方法实现查询
        Account account = new Account();
        account.setId(1L);
        account.setUsername("张三");
        account.setAge(20);
        Account account1 =
accountRepository.findByAccount(account);
        System.out.println(account1);
    }
}

```

此时因为未给password赋值，查询结果为空：

```

Opening JDBC Connection
Created connection 1757880885.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@68c72235]
==> Preparing: SELECT * FROM account WHERE id=? AND username = ? AND password = ? AND age = ?
==> Parameters: 1(Long), 张三(String), null, 20(Integer)
<==      Total: 0
null
Resetting autocommit to true on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@68c72235]

```

但此时我们其他三个条件都匹配，而password只是没有进行赋值，此时我们应该有数据进行匹配，但因为是 AND 连接的SQL语句，一个不成立，则无法得到数据，手动进行数据是否为被赋值的判断又很麻烦，此时我们就可以动态的识别未赋值字段，自动修改SQL语句，这就是 动态 SQL。

- 实现动态 SQL

if标签实现

我们可以修改对应 mapper.xml 文件内对应的 SQL语句配置关系，让他去自动识别是否被赋值或设置将它加入 SQL 语句 的条件，来进行处理：

```
<select id="findByAccount"
parameterType="com.gloryh.entity.Account"
resultType="com.gloryh.entity.Account">
    SELECT * FROM account WHERE
    <if test="id!=0">
        id=#{id}
    </if>
    <if test="username!=null">
        AND username = #{username}
    </if>
    <if test="password!=null">
        AND password = #{password}
    </if>
    <if test="age!=0">
        AND age = #{age}
    </if>
</select>
```

运行之前的测试方法（不给 password 属性赋值），得到结果（有数据）：

```
Created connection 574266151.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImp
==> Preparing: SELECT * FROM account WHERE id=? AND username = ? AND age = ?
==> Parameters: 1(Long), 张三(String), 20(Integer)
<== Columns: id, username, password, age
<== Row: 1, 张三, 123, 20
<== Total: 1
Account(id=1, username=张三, password=123, age=20)
```

但是此时如果 id为 0，SQL语句就会变成：

```
SELECT * FROM account WHERE AND username = ? AND  
age = ?
```

直接追加 AND 的话，SQL语句就会出错，此时我们可以提前追加 1=1 保证永真，然后 id 是否为 0 都不会有影响，即：

```
SELECT * FROM account WHERE 1 = 1 AND id=? AND  
username = ? AND age = ?
```

where 标签

除了 1=1 这种永真写法，MyBatis 还提供了 where 标签解决这个问题，我们可以将所有的 if 条件装进 where 标签内，它会自动处理 AND 被直接追加的情况：

```
<select id="findByAccount"  
parameterType="com.gloryh.entity.Account"  
resultType="com.gloryh.entity.Account">  
    SELECT * FROM account  
    <where>  
        <if test="id!=0">  
            id=#{id}  
        </if>  
        <if test="username!=null">  
            AND username = #{username}  
        </if>  
        <if test="password!=null">  
            AND password = #{password}  
        </if>  
        <if test="age!=0">  
            AND age = #{age}  
        </if>  
    </where>  
</select>
```

测试方法中不给 id 属性赋值：

```
package com.gloryh.test;  
  
import com.gloryh.entity.Account;  
import com.gloryh.repository.AccountRepository;
```



```

import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 动态 SQL 测试类
 *
 * @author 黄光辉
 * @since 2020/9/22
 */
public class Test6 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        AccountRepository accountRepository =
session.getMapper(AccountRepository.class);
        //调用接口方法实现查询
        Account account = new Account();
        /*account.setId(1L);*/
        account.setUsername("张三");
        account.setAge(20);
        Account account1 =
accountRepository.findByAccount(account);
        System.out.println(account1);
        session.close();
    }
}

```

运行结果（有数据）：

```
Created Connection 134317740.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@932bc4a]
==> Preparing: SELECT * FROM account WHERE username = ? AND age = ?
==> Parameters: 张三(String), 20(Integer)
<== Columns: id, username, password, age
<== Row: 1, 张三, 123, 20
<== Total: 1
Account(id=1, username=张三, password=123, age=20)
```

choose 标签和 when 标签

when 标签、choose 标签和 when 标签一起使用也可以解决类似的问题，此时判断条件不需要追加 AND 字段，MyBatis 会自动完成对 SQL 语句的处理：

```
<select id="findByAccount"
parameterType="com.gloryh.entity.Account"
resultType="com.gloryh.entity.Account">
    SELECT * FROM account
    <where>
        <choose>
            <when test="id!=0">
                id = #{id}
            </when>
            <when test="username!=null">
                username = #{username}
            </when>
            <when test="password!=null">
                password = #{password}
            </when>
            <when test="age!=0">
                age = #{age}
            </when>
        </choose>
    </where>
</select>
```

运行之前的测试（不给 id 赋值），查看结果（有数据）：

```
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@
==> Preparing: SELECT * FROM account WHERE username = ?
==> Parameters: 张三(String)
<== Columns: id, username, password, age
<== Row: 1, 张三, 123, 20
<== Total: 1
Account(id=1, username=张三, password=123, age=20)
```

trim 标签

trim 标签中的 prefix 和 suffix 属性 会被用于生成实际的 SQL 语句，回合标签内部的语句进行拼接，如果语句前后出现了 prefixOverrides 或者 suffixOverrides 属性中指定的值，MyBatis 框架会自动将其删除。

trim 标签 和 where 标签的功能相同。

```
<select id="findByAccount"
parameterType="com.gloryh.entity.Account"
resultType="com.gloryh.entity.Account">
    SELECT * FROM account
    <trim prefix="where" prefixOverrides="and">
        <if test="id!=0">
            id=#{id}
        </if>
        <if test="username!=null">
            AND username = #{username}
        </if>
        <if test="password!=null">
            AND password = #{password}
        </if>
        <if test="age!=0">
            AND age = #{age}
        </if>
    </trim>
</select>
```

仍然不给 id 赋值，查看运行结果（有数据）：

```
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@5
==> Preparing: SELECT * FROM account where username = ? AND age = ?
==> Parameters: 张三(String), 20(Integer)
<== Columns: id, username, password, age
<== Row: 1, 张三, 123, 20
<== Total: 1
Account(id=1, username=张三, password=123, age=20)
```

set 标签

set 标签用于 update 操作，会自动根据参数选择生成 SQL 语句。

改写之前的 update 方法对应的 mapper.xml.

改写前：

```

<update id="update"
parameterType="com.gloryh.entity.Account">
    UPDATE account SET username = #
{username},password = #{password},age = #{age}
WHERE id =#{id}
</update>

```

测试修改 username属性为张三的数据，将 张三修改为王五：

```

package com.gloryh.test;

import com.gloryh.entity.Account;
import com.gloryh.repository.AccountRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import
org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;

/**
 * 动态 SQL 测试类
 *
 * @author 黄光辉
 * @since 2020/9/22
 */
public class Test6 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象

```

```

        AccountRepository accountRepository =
session.getMapper(AccountRepository.class);
        //调用接口方法实现查询
        Account account = new Account();
        account.setId(1L);
        account.setUsername("张三");
        /*account.setAge(20);*/
        accountRepository.update(account);
        session.commit();
        session.close();
    }
}

```

修改前：

	id	username	password	age
I	1	张三	123	20
	2	李四	123	20

运行结果（发现其实不仅 username 属性 被修改了，其他的也被修改，被置为空或null）：

开始事务

文本

筛选

排序

	id	username	password	age
▶	1	王五	(Null)	0
	2	李四	123	20

set 标签改写后就可以解决这个问题，动态修改被改变的值，未被赋值则不被修改：

```

<update id="update"
parameterType="com.gloryh.entity.Account">
    UPDATE account
    <set>
        <if test="username!=null">
            username = #{username},
        </if>
        <if test="password!=null">
            password = #{password},
        </if>
    </set>
</update>

```

```
<if test="age!=0">
    age = #{age}
</if>
WHERE id =#{id}
</set>
</update>
```

运行前：

	id	username	password	age
▶	1	张三	123	20
	2	李四	123	20

控制台：

```
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@48974e45]
==> Preparing: UPDATE account SET username = ? WHERE id = ?
==> Parameters: 王五(String), 1(Long)
<== Updates: 1
Committing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@48974e45]
```

运行后：

	id	username	password	age
▶	1	王五	123	20
	2	李四	123	20

foreach 标签

foreach 标签 可以迭代生成一系列值，主要用于 SQL 语句 中的 in 语句。

例如，我一次查询多个 id 相匹配的 Account。

首先，Account中添加属性 ids：

```
package com.gloryh.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.List;
```

```

/**
 * 实体类Account
 *
 * @author 黄光辉
 * @since 2020/9/17
 **/
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Account {
    private long id;
    private String username;
    private String password;
    private int age;
    private List<Long> ids;
}

```

提供一个接口用于返回 Account 的 List 集合：

```

package com.gloryh.repository;

import com.gloryh.entity.Account;

import java.util.List;

/**
 * 自动定义 repository 接口
 *
 * @author 黄光辉
 * @since 2020/9/19
 **/
public interface AccountRepository {
    public int save(Account account );
    public int update(Account account);
    public int deleteById( long id);
    public List<Account> findAll();
    public Account findById(long id);
    public Account findByAccount(Account account);
    public List<Account> findByIds(Account
account);
}

```

对应 mapper.xml 文件实现：

```
<select id="findByIds"
parameterType="com.gloryh.entity.Account"
resultType="com.gloryh.entity.Account">
    SELECT * FROM account
    <where>
        <foreach collection="ids" open="id in("
close=*)" item="id" separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

测试方法调用接口：

```
package com.gloryh.test;

import com.gloryh.entity.Account;
import com.gloryh.repository.AccountRepository;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;

/**
 * 动态 SQL 测试类
 *
 * @author 黄光辉
 * @since 2020/9/22
 */
public class Test6 {
    public static void main(String[] args) {
        //加载MyBatis配置文件
        InputStream stream =
Test.class.getClassLoader().getResourceAsStream("co
nfig.xml");
```



```

        SqlSessionFactoryBuilder factoryBuilder =
new SqlSessionFactoryBuilder();
        SqlSessionFactory sessionFactory =
factoryBuilder.build(stream);
        SqlSession session =
sessionFactory.openSession();
        //获取实现接口的代理对象
        AccountRepository accountRepository =
session.getMapper(AccountRepository.class);
        //调用接口方法实现查询
        Account account = new Account();
        List<Long> ids=new ArrayList<Long>();
        ids.add(1L);
        ids.add(2L);
        account.setIds(ids);
        System.out.println(
accountRepository.findByIds(account));
        session.close();
    }
}

```

运行结果：

```

Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@932bc4a]
==> Preparing: SELECT * FROM account WHERE id in( ?, ? )
==> Parameters: 1(Long), 2(Long)
<== Columns: id, username, password, age
<== Row: 1, 王五, 123, 20
<== Row: 2, 李四, 123, 20
<== Total: 2
[Account(id=1, username=王五, password=123, age=20, ids=null), Account(id=2, username=李四, password=123, age=20, ids=null)]
Resetting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@932bc4a]

```