IT楠老师mybatis教程

> **B站：IT楠老师 公众号：IT楠说jiava QQ群： 1083478826 新知大数据**

制作不易、如果觉的好不妨打个赏：

B站IT楠老师

一、了解MyBatis

1、历史（百度百科）

- 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 。2013年11月迁移到Github。
- iBATIS一词来源于“internet”和“abatis”的组合，是一个基于Java的持久层框架。iBATIS提供的持久层框架包括SQL Maps和Data Access Objects（DAOs）

2、作用 （百度百科）

- **MyBatis 是一款优秀的持久层框架**，它支持定制化 SQL、存储过程以及高级映射。
- MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。
- MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs(Plain Ordinary Java Object,普通的 Java对象)映射成数据库中的记录。

3、说说持久化

持久化就是把数据存在磁盘而不是内存。

1. 程序产生的数据首先都是在内存。
2. 内存是不可靠的，他丫的一断电数据就没了。

3. 那可靠的存储地方是哪里，硬盘、U盘、光盘等。
4. 我们的程序在运行时说的持久化通常就是指将内存的数据存在硬盘。
5. 硬盘用哪些地方能存？数据库文件、xml文件、反正你将来能将数据读出来的文件都行。

高级一点：持久化就是将数据从瞬时态转化为持久态，长久保存。

4、说说持久层

- 业务是需要操作数据的
- 数据是在磁盘上的
- 具体业务调用具体的数据库操作，耦合度太高，复用性太差
- 将操作数据库的代码统一抽离出来，自然就形成了介于业务层和数据库中间的独立的层

5、持久层框架

- 写过jdbc没，里边的重复代码太多了，我们在讲jdbc的时候尝试带大家利用反射写了个简单的BaseDao，之后基本的crud就能不用写了，这就有了框架的影子了。

> **以下的代码，如果50个字段是不是写50次set**

```
perparedstatment=conn.prepareStatement("insert sysuser (uname,uphone) values (?,?) ");
perparedstatment.setString(1,var1);
perparedstatment.setString(2,var2);
perstmt.executeUpdate();
```

> **咱们写了basedao（mysql的jdbc讲解视频里有）**

```
public class BaseDaoImpl<T> implements IBaseDao<T> {}
public class UserDaoImpl extends BaseDaoImpl<User> implements UserDao {}
```

- 利用框架能够及其简单的帮助我们实现持久层的操作，从而规避了复杂且重复的操作。
- 实现了从实体类到数据库的映射关系，也就是orm映射。

6、mybatis的优点和缺点

- sql语句与代码分离，存放于xml配置文件中：

  优点：便于维护管理，不用在java代码中找这些语句；

  缺点： JDBC方式可以用打断点的方式调试，但是Mybatis不能，需要通过log4j日志输出日志信息帮助调试，然后在配置文件中修改。

- 用逻辑标签控制动态SQL的拼接：

  优点：用标签代替编写逻辑代码；

  缺点：拼接复杂SQL语句时，没有代码灵活，拼写比较复杂。不要使用变通的手段来应对这种复杂的语句。

- 查询的结果集与java对象自动映射：

  优点：保证名称相同，配置好映射关系即可自动映射或者，不配置映射关系，通过配置列名=字段名也可完成自动映射。

缺点：对开发人员所写的SQL依赖很强。

- 编写原生SQL：

  优点：接近JDBC，比较灵活。

  缺点：对SQL语句依赖程度很高；并且属于半自动，数据库移植比较麻烦，比如mysql数据库编程Oracle数据库，部分的sql语句需要调整。

- 最重要的一点，使用的人多！公司需要！但是应为用了反射，效率会下降，所有有些公司会使用原生的jdbc

7、代理设计模式

> **代理模式分为静态代理和动态代理。代理的核心功能是方法增强。**

（1）静态代理

静态代理角色分析

- 抽象角色 : 一般使用接口或者抽象类来实现
- 真实角色 : 被代理的角色
- 代理角色 : 代理真实角色 ; 代理真实角色后 , 一般会做一些附属的操作 .
- 客户 : 使用代理角色来进行一些操作 .

代码实现

写一个接口

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
public interface Singer {
    /**
     * 歌星都能唱歌
     */
    void sing();
}
```

定义男歌手

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
public class MaleSinger implements Singer{

    private String name;

    public MaleSinger(String name) {
        this.name = name;
    }
```

BIT IT楠老师

```
    @Override
    public void sing() {
        System.out.println(this.name + "开始唱歌了！");
    }
}
```

定义经纪人

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
public class Agent implements Singer {

    private Singer singer;

    public Agent(Singer singer) {
        this.singer = singer;
    }

    @Override
    public void sing() {
        System.out.println("节目组找过来！需要演出，谈好演出费用。。。。。");
        singer.sing();
        System.out.println("结算费用，下一次合作预约。。。。。。");
    }
}
```

Client . java 即客户

```
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
public class Client {

    public static void main(String[] args) {
        Singer singer = new MaleSinger("鹿晗");
        Singer agent = new Agent(singer);
        agent.sing();
    }
}
```

分析：在这个过程中，你直接接触的就是鹿晗的经济人，经纪人在鹿晗演出的前后跑前跑后发挥了巨大的作用。

优点

- 鹿晗还是鹿晗，没有必要为了一下前置后置工作改变鹿晗这个类
- 公共的统一问题交给代理处理
- 公共业务进行扩展或变更时，可以更加方便
- 这不就是更加符合开闭原则，单一原则吗?

B站IT楠老师

缺点：

- 每个类都写个代理，麻烦死了。

（2）动态代理

- 动态代理的角色和静态代理的一样 .
- 动态代理的代理类是动态生成的 . 静态代理的代理类是我们提前写好的
- 动态代理分为两类 : 一类是基于接口动态代理 , 一类是基于类的动态代理
- - 基于接口的动态代理----JDK动态代理
  - 基于类的动态代理--cglib（有兴趣自己研究）
  - 现在用的比较多的是 javasist 来生成动态代理 . 百度一下javasist
  - 我们这里使用JDK的原生代码来实现，其余的道理都是一样的！、

JDK的动态代理需要了解两个类

核心 : InvocationHandler 和 Proxy ， 打开JDK帮助文档看看

【InvocationHandler：调用处理程序】

```
Object invoke(Object proxy, 方法 method, Object[] args);
//参数
//proxy - 调用该方法的代理实例
//method -所述方法对应于调用代理实例上的接口方法的实例。方法对象的声明类将是该方法声明的接口，它可以是代理类继承该方法的代理接口的超级接口。
//args -包含的方法调用传递代理实例的参数值的对象的阵列，或null如果接口方法没有参数。原始类型的参数包含在适当的原始包装器类的实例中，例如java.lang.Integer或java.lang.Boolean 。
```

【Proxy : 代理】

```
//生成代理类
public Object getProxy(){
   return Proxy.newProxyInstance(this.getClass().getClassLoader(),
                                 rent.getClass().getInterfaces(),this);
}
```

代码实现

抽象角色和真实角色和之前的一样！

还是歌星和男歌星

Agent. java 即经纪人

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 * 经纪人
```

```
 */
public class Agent implements InvocationHandler {

    private Singer singer;

    /**
     * 设置代理的经济人
     * @param singer
     */
    public void setSinger(Singer singer) {
        this.singer = singer;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("---------------经纪人把把关--------------");
        Object returnObj = method.invoke(singer, args);
        System.out.println("---------------唱完了收收钱-----------------------");
        return returnObj;
    }

    /**
     * 获取一个代理对象
     * @return
     */
    public Object getProxy(){
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
                new Class[]{Singer.class},this);
    }
}
```

Client . java

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {

        MaleSinger luhan = new MaleSinger();

        Agent agent = new Agent();
        agent.setSinger(luhan);
        Singer singer = (Singer)agent.getProxy();

        singer.sing();
    }
}
```

核心：**一个动态代理，一般代理某一类业务，一个动态代理可以代理多个类，代理的是接口！、**

B站 IT楠老师

```
//该设置用于输出cglib动态代理产生的类
System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY, "D:\\class");
//该设置用于输出jdk动态代理产生的类
System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");
```

（3）万能代理

批量做代理，不用一个一个写

我们来使用动态代理实现代理我们后面写的UserService！

我们也可以编写一个通用的动态代理实现的类！所有的代理对象设置为Object即可！

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class ProxyAll implements InvocationHandler {
    private Object target;

    /**
     * 设置代理的经济人
     * @param target
     */
    public void setTarget(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("---------------之前干点啥！比如打印日志，开启事务？--------------
");
        Object returnObj = method.invoke(target, args);
        System.out.println("---------------之后干点啥！比如打印日志，关闭事务？--------------
----------");
        return returnObj;
    }

    /**
     * 获取一个代理对象
     * @return
     */
    public Object getProxy(){
        return Proxy.newProxyInstance(this.getClass().getClassLoader(),
                target.getClass().getInterfaces(),this);
    }
}
```

测试！

B站 IT楠老师

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {

        MaleSinger luhan = new MaleSinger();

        ProxyAll agent = new ProxyAll();
        agent.setTarget(luhan);
        Singer singer = (Singer)agent.getProxy();

        singer.sing();
    }
}
```

测试，增删改查，查看结果，依然可以

```
---------------之前干点啥！比如打印日志，开启事务？--------------
我要唱歌了！！！
---------------之后干点啥！比如打印日志，关闭事务？---------------
```

> **思考那我们其他所有的类是不是都可以了?**

咱们的dao，service的类是不是都能代理了，批量去给这些方法加一些日志之类的是不是就可以了，或者统一加上开启事务，关系事务等。

(4) 只有接口的dao

还是要传入一个代理对象，那么我们能不能直接根据接口生成个代理啊！

> **能**

数据库相关

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
```

```
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class DBUtil {
    public static Connection getConnection(){
```

BIT IT楠老师

```
        String url = "jdbc:mysql://localhost:3306/ssm?
useUnicode=true&characterEncoding=utf-
8&relaxAutoCommit=true&zeroDateTimeBehavior=convertToNull&allowMultiQueries=true&useSSL
=false";
        String user = "root";
        String password = "root";
        String driverName = "com.mysql.jdbc.Driver";

        Connection conn = null;
        try {
            Class clazz = Class.forName(driverName);
            Driver driver = (Driver) clazz.newInstance();
            //3.注册驱动
            DriverManager.registerDriver(driver);
            //4.获取连接
            conn = DriverManager.getConnection(url, user, password);
        }catch (Exception e){
            e.printStackTrace();
        }
        return conn;
    }

    public static void closeAll(Connection connection, Statement statement, ResultSet
rs){
        if(connection != null){
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if(statement != null){
            try {
                statement.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if( rs != null ){
            try {
                rs.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

实体类

B站IT辅老师

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class User {

    private int id;
    private String username;
    private String password;

    ...
}
```

搞个接口

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public interface IUserDao {

    /**
     * 更新用户
     * @param user
     */
    void InsertUser(User user);

}
```

全国统一的插入代理

这个代理能针对所有的实体类（实体类和表名一致，字段名一致）生成对应的sql并形成代理。

简单感受即可，此代理功能简单不严谨。

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class InsertProxy implements InvocationHandler {

    @Override
    public Object invoke(Object proxy, Method method, Object[] args){
        //使用参数（就用第一个）实体类拼装sql（insert）
        StringBuilder sql = new StringBuilder("insert into ");
        sql.append(args[0].getClass().getSimpleName().toLowerCase())
                .append(" values (");
        Field[] fields = args[0].getClass().getDeclaredFields();
        for (Field field : fields) {
```

B站 IT楠老师

```
            field.setAccessible(true);
            if( field.getType() == int.class ){
                try {
                    sql.append(field.getInt(args[0])).append(",");
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
            }else if( field.getType() == String.class ){
                try {
                    sql.append("'").append(field.get(args[0])).append("',");
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
            }
        }

        //执行sql
        Connection connection = null;
        PreparedStatement preparedStatement = null;
        int rows = 0;
        try {
            connection = DBUtil.getConnection();
            sql.deleteCharAt(sql.length() - 1).append(")");
            preparedStatement = connection.prepareStatement(sql.toString());
            rows = preparedStatement.executeUpdate();
        }catch (SQLException e){
            e.printStackTrace();
        }finally {
            DBUtil.closeAll(connection,preparedStatement,null);
        }

        System.out.println("sql-------->" + sql);
        System.out.println("受影响行数--->: " + rows);
        return rows;
    }

    /**
     * 获取一个代理对象
     * @return
     */
    public static  <T> T getProxy(Class<T> tClass){
        InsertProxy insertProxy = new InsertProxy();
        return (T)Proxy.newProxyInstance(insertProxy.getClass().getClassLoader(),
                new Class[]{tClass},insertProxy);
    }
}
```

以后有插入需求，只需要写个接口生成代理即可，测试

```java
/**
 * @author IT楠老师
 * @date 2020/5/21
 */
public class Client {
    public static void main(String[] args) {
        //直接根据接口获取一个代理类
        IUserDao userDao = InsertProxy.getProxy(IUserDao.class);
        userDao.updateUser(new User(202,"2","23"));
    }
}
```

结果

```
sql-------->insert into user values (202,'2','23')
受影响行数--->: 1
```

真牛逼！！！！！

思考，比如有一个Admin的实体类和dao接口是不是也能进行插入啊！！！

二、搭建个环境

1、建立数据库

```sql
CREATE DATABASE `ssm`;
USE `ssm`;
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
`id` int(20) NOT NULL,
`username` varchar(30) DEFAULT NULL,
`password` varchar(30) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert  into `user`(`id`,`username`,`password`) values (1,'楠哥','123456'),(2,'老孙','abcdef'),(3,'磊哥','987654');
```

2、编写实体类

自行学习lombok

B站 IT楠老师

```java
/**
 * @author IT楠老师
 * @date 2020/5/28
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User implements Serializable {
    private int id;
    private String username;
    private String password;
}
```

3、maven配置

```xml
<dependencies>
    <!-- 单元测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.7</version>
        <scope>test</scope>
    </dependency>
    <!-- mybatis 核心 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.2</version>
    </dependency>
    <!-- 数据库确定 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.47</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.18</version>
    </dependency>
</dependencies>

<!-- 处理资源被过滤问题 -->
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
```

B站IT楠老师

```
                <source>1.8</source> <!-- 源代码使用的JDK版本 -->
                <target>1.8</target> <!-- 需要生成的目标class文件的编译版本 -->
                <encoding>UTF-8</encoding><!-- 字符集编码 -->
            </configuration>
        </plugin>
    </plugins>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>false</filtering>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>false</filtering>
        </resource>
    </resources>
</build>
```

4、编写MyBatis核心配置文件，mybatis-config.xml

> **小知识**

1. DTD(Document Type Definition)即文档类型定义，是一种XML约束模式语言，是XML文件的验证机制
2. 一个DTD文档包含：（1）元素的定义规则；（2）元素间关系的定义规则；（3）元素可使用的属性，可使用的实体或符号规则

当然我们还有可能接触其他类型的头文件，比如XML Schemal语言就是XSD，XML Schema描述了XML文档的结构，可以用一个指定的XML Schema来验证某个XML

1. XML Schema基于XML,没有专门的语法
2. XML Schema可以象其他XML文件一样解析和处理
3. XML Schema比DTD提供了更丰富的数据类型.
4. XML Schema提供可扩充的数据模型。
5. XML Schema支持综合命名空间

总之不管是dtd还是xsd文件都是用来约束我们的xml文件，保证我们的xml能够使用哪些标签，保证xml的有效性，在idea中工具还能根据头文件为我们提供强大的提示功能。

> **有兴趣的自行深入研究。**

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

```
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments 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/ssm?
useSSL=true&amp;useUnicode=true&amp;characterEncoding=utf8"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

三、CRUD来一套

说明

第一次使用mybatis，要注意mybatis，需要两个文件

- 一个接口类mapper（就是咱们写的dao）java文件，不需要有实现
- 一个与接口对应的xml文件
- 两个文件名字最好一样，比如一个UserMapper.java一个UserMapper.xml
- 框架要根据mapper和xml联合，形成代理对象

xxxMapper基本格式

```
<?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.xinzhi.dao.UserMapper">
     <select id="selectUser" resultType="com.xinzhi.entity.User">
        select id,username,password from user
     </select>
</mapper>
```

xxxmapper.xml需要在核心配置文件中注册

B站IT辉老师

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
       PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
       "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
   <environments default="development">
       ...省略
   </environments>
   <!-- 你要告诉我有哪些mapper，我统一管理 -->
   <mappers>
        <mapper resource="com/xinzhi/dao/userMapper.xml"/>
    </mappers>
</configuration>
```

简单理解namespace就是 维护mapper的对应关系

配置文件中namespace中的名称为对应Mapper接口或者Dao接口的完整包名,叫mapper或dao无所谓，但名字必须一致!

1、select（查询）

select标签是mybatis中最常用的标签

1、在UserMapper中添加对应方法

```
public interface UserMapper {
   //根据id查询用户
   User selectUserById(int id);
}
```

2、在UserMapper.xml中添加Select语句

```
<select id="selectUserById" resultType="com.xinzhi.entity.User"  parameterType="int">
      select id,username,password from user where id = #{id}
</select>
```

知识点:

- resultType：制定返回类型，查询是有结果的，结果啥类型，你得告诉我
- parameterType：参数类型，查询是有参数的，参数啥类型，你得告诉我
- id：制定对应的方法，就是你的告诉我你这sql对应的是哪个方法
- #{id}：sql中的变量，要保证大括号的变量必须在User对象里有，
- 当然你可以使用map，这或啥也能干，这要你将来传进的map有id这个key就行
- #{}：占位符，其实就是咱们的【PreparedStatement】处理这个变量

除了#{}还有${}，看看有啥区别，面试常问

B站IT辅老师

- #{} 的作用主要是替换预编译语句(PrepareStatement)中的占位符? 【推荐使用】

```
INSERT INTO user (username) VALUES (#{username});
INSERT INTO user (username) VALUES (?);
```

- ${} 的作用是直接进行字符串替换

```
INSERT INTO user (username) VALUES ('${username}');
INSERT INTO user (username) VALUES ('楠哥');
```

3、测试类中测试

```
@Test
public void testFindUserById() {
     try {
        String resource = "mybatis-config.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        SqlSession session = sqlSessionFactory.openSession();

       //就问熟悉不熟悉，这不也是个代理?
       UserMapper mapper = session.getMapper(UserMapper.class);
       User user = mapper.selectUserById(1);
       System.out.println(user);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
private SqlSession session;

@Before
public void before() {
    try {
        String resource = "mybatis-config.xml";
        InputStream inputStream = Resources.getResourceAsStream(resource);
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        session = sqlSessionFactory.openSession();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

2、insert（插入）

Bili 楠老师

insert标签被用作插入操作

1、接口中添加方法

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
public interface UserMapper {

    /**
     * 新增user
     * @param user
     * @return
     */
    int addUser(User user);
}
```

2、xml中加入insert语句

```
<insert id="addUser" parameterType="com.xinzhi.entity.User">
    insert into user (id,username,password) values (#{id},#{username},#{password})
</insert>
```

3、测试

```
@Test
public void testAddUser() {
    UserMapper mapper = session.getMapper(UserMapper.class);
    User user = new User(5,"微微姐","12345678");
    int affectedRows = mapper.addUser(user);
    System.out.println(affectedRows);
}
```

注：增、删、改操作需要提交事务！此处引出，mybatis其实设置了事务的手动提交，其实不管如何，都应该提交事务，那么我们可以增强一下测试类。

```
public class TestUser {

    private SqlSession session;

    @Before
    public void createSession() {
        try {
            InputStream inputStream = Resources.getResourceAsStream("mybatis-
config.xml");
            SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
            session = sqlSessionFactory.openSession();
        } catch (IOException e) {
```

B站IT楠老师

```
            e.printStackTrace();
        }
    }

    @After
    public void commit() {
        //统一提交事务
        session.commit();
        session.close();
    }
}
```

思考，如果参数没有传实体类而是传了多个参数，，能不能执行

比如数据库为id，方式传入userId

1、在UserMapper中添加对应方法

```
/**
     * 新增用户
     * @param id
     * @param name
     * @param pws
     * @return
     */
int addUser(int id,String name,String pws);
```

2、在UserMapper.xml中添加Select语句

```
<insert id="addUser" parameterType="com.xinzhi.entity.User">
        insert into user (id,username,password) values (#{id},#{username},#{password})
</insert>
```

3、测试

```
@Test
public void testAddUser() {
    UserMapper mapper = session.getMapper(UserMapper.class);
    int affectedRows = mapper.addUser(6,"微微姐","12345678");
    System.out.println(affectedRows);
}
```

```
 Cause: org.apache.ibatis.binding.BindingException: Parameter 'id' not found. Available
parameters are [arg2, arg1, arg0, param3, param1, param2]
```

这就无法隐射了。

这就需要个注解了@Param

B站IT梅老师

```
 /**
     * 新增用户
     * @param userId
     * @param name
     * @param pws
     * @return
     */
int addUser(@Param("id") int id,@Param("username") String name,@Param("password")
String pws);
```

> **结论多个参数时，使用@Param注解，能制定对应的参数，当然如果能封装对象最好封装对象**

3、update（修改）

> **update标签用于更新操作**

1、写接口

```
/**
 * 修改用户
 * @param user
 * @return
 */
int updateUser(User user);
```

2、写SQL

```
<update id="updateUser" parameterType="com.xinzhi.entity.User">
    update user set username=#{username},password=#{password} where id = #{id}
</update>
```

3、测试

```
@Test
public void testUpdateUser() {
    UserMapper mapper = session.getMapper(UserMapper.class);
    User user = new User(5,"微微姐","12345678");
    int affectedRows = mapper.updateUser(user);
    System.out.println(affectedRows);
}
```

4、delete（删除）

> **delete标签用于做删除操作**

1、写接口

B站IT精选

```
/**
* 删除一个用户
* @param id
* @return
*/
int deleteUser(int id);
```

2、写SQL

```
<delete id="deleteUser" parameterType="int">
  delete from user where id = #{id}
</delete>
```

3、测试

```
@Test
public void testDeleteUser(){
    UserMapper mapper = session.getMapper(UserMapper.class);
    int affectedRows = mapper.deleteUser(5);
    System.out.println(affectedRows);
}
```

简单的梳理逻辑

1. SqlSessionFactoryBuilder().build(inputStream)-读取核心配置文件，
2. 核心配置文件负责核心配置，重要的一点是通过核心配置找到Mapper的xml文件。
3. xml文件通过 namespace 又能找到 对应的接口文件。
4. 有了方法，有了sql，会用动态代理，生成代理类，实现对象的方法。
5. session.getMapper(UserMapper.class) 获取的就是代理对象，当然就有了对应的实现。

5、模糊查询

方案一：在Java代码中拼串

```
string name = “%IT%”;
list<name> names = mapper.getUserByName(name);
```

```
<select id=”getUsersByName”>
    select * from user where name like #{name}
</select>
```

方案二：在配置文件中拼接

```
string name = “IT”;
list<User> users = mapper.getUserByName(name);
```

```
<select id=”getUsersByName”>
    select * from user where name like "%"#{name}"%"
</select>
```

为什么必须用双引号?

6、map的使用

> **map可以代替任何的实体类，所以当我们数据比较复杂时，可以适当考虑使用map来完成相关工作**

1、写sql

```
<select id="getUsersByParams" resultType="map">
    select id,username,password from user where username = #{name}
</select>
```

2、写方法

```
/**
* 根据一些参数查询
* @param map
* @return
*/
List<User> getUsersByParams(Map<String,String> map);
```

3、测试

```
@Test
public void findByParams() {
    UserMapper mapper = session.getMapper(UserMapper.class);
    Map<String,String> map = new HashMap<String, String>();
    map.put("name","磊磊哥");
    List<User> users = mapper.getUsersByParams(map);
    for (User user: users){
        System.out.println(user.getUsername());
    }
}
```

7、别名小插曲

> **思考:**

```
<select id="getUsersByParams" resultType="java.util.HashMap">
    select id,username,password from user where username = #{name}
</select>
```

resultType写成java.util.HashMap，也行写成map也行

B站 IT辅老师

说明mybatis内置很多别名：

Alias	Mapped Type
_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
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

B站IT涛老师

给自己类设定别名

在核心配置文件中加入

```
<typeAliases>
    <typeAlias type="com.xinzhi.entity.User"  alias="user"/>
</typeAliases>
```

`<typeAlias>` 标签中 有 `type` 和 `alias` 两个属性

`type` 填写 实体类的全类名， `alias` 可以不填，不填的话，默认是类名，不区分大小写，

`alias` 填了的话就以 alias里的值为准。

```
<typeAiases>
    <package name="cn.qingmings.mybatis.model"/>
</typeAliases>
```

`<package>` 标签 为某个包下的所有类起别名； `name` 属性填写包名。 别名默认是类名，不区分大小写。

```
@Alias` 注解    加在实体类上，为某个类起别名；例：`@Alias("User")
```

小结

- map工作中及其常用，特别在多表查询中，字段又多又复杂。
- 大家一定学会合理使用map传参，不要一味的修改实体类，导致某些实体类过于臃肿。

7、resultMap详解

数据库不可能永远是你所想或所需的那个样子

属性名和字段名不一致，我们一般都会按照约定去设计数据的，但确实阻止不了一些孩子，瞎比起名字。

1、Java中的实体类设计

```
public class User {

   private int id;              //id
   private String name;         //姓名，数据库为username
   private String password;     //密码，一致

   //构造
   //set/get
   //toString()
}
```

3、mapper

Bili IT博老师

```
//根据id查询用户
User selectUserById(int id);
```

4、mapper映射文件

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

5、测试

```
@Test
public void testSelectUserById() {
   UserMapper mapper = session.getMapper(UserMapper.class);
   User user = mapper.selectUserById(1);
   System.out.println(user);
   session.close();
}
```

结果:

- User{id=1, name='null', password='123'}
- 查询出来发现 name为空 . 说明出现了问题!

分析:

- select * from user where id = #{id} 可以看做

  select id,username,password from user where id = #{id}

- mybatis会根据这些查询的列名(会将列名转化为小写,数据库不区分大小写) , 利用反射去对应的实体类中查找相应列名的set方法设值 ， 当然找不大username;

解决方案

方案一：为列名指定别名 , 别名和java实体类的属性名一致 .

```
<select id="selectUserById" resultType="User">
  select id , username as name ,password from user where id = #{id}
</select>
```

方案二：使用结果集映射->ResultMap 【推荐】

B站IT杨老师

```
<resultMap id="UserMap" type="User">
   <!-- id为主键 -->
   <id column="id" property="id"/>
   <!-- column是数据库表的列名 , property是对应实体类的属性名 -->
   <result column="username" property="name"/>
   <result column="password" property="password"/>
</resultMap>

<select id="selectUserById" resultMap="UserMap">
  select id , username , password from user where id = #{id}
</select>
```

> **结论：**

这个地方我们手动调整了映射关系，称之为手动映射。

但如果不调整呢？mybatis当然会按照约定自动映射。

有了映射这种牛逼的事情之后：

我们的：

```
prepareStatement.setInt(1,21);
prepareStatement.setString(2,"IT楠老师");
```

还用写吗？两个字【**牛逼**】！

> **当然约定的最基本的操作就是全都都一样，还有就是下划线和驼峰命名的自动转化**

```
<settings>
    <!--开启驼峰命名规则-->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

B站IT楠老师

四、使用注解开发

```
DROP TABLE IF EXISTS `admin`;
CREATE TABLE `admin` (
`id` int(20) NOT NULL,
`username` varchar(30) DEFAULT NULL,
`password` varchar(30) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert  into `admin`(`id`,`username`,`password`) values (1,'楠哥','123456'),(2,'老孙','abcdef'),(3,'磊哥','987654');
```

- **mybatis最初配置信息是基于 XML ,映射语句(SQL)也是定义在 XML 中的。而到MyBatis 3提供了新的基于注解的配置。不幸的是，Java 注解的的表达力和灵活性十分有限。最强大的 MyBatis 映射并不能用注解来构建**
- sql 类型主要分成 :
- @select ()
- @update ()
- @Insert ()
- @delete ()

注意：利用注解开发就不需要mapper.xml映射文件了 .

1、接口中添加注解

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
public interface AdminMapper {

    /**
     * 保存管理员
     * @param admin
     * @return
     */
    @Insert("insert into admin (username,password) values (#{username},#{password})")
    int saveAdmin(Admin admin);

    /**
     * 跟新管理员
     * @param admin
     * @return
     */
    @Update("update admin set username=#{username} , password=#{password} where id = #{id}")
    int updateAdmin(Admin admin);

    /**
```

B站 IT楠老师

```
     * 删除管理员
     * @param admin
     * @return
     */
    @Delete("delete from admin where id=#{id}")
    int deleteAdmin(int id);

    /**
     * 根据id查找管理员
     * @param id
     * @return
     */
    @Select("select id,username,password from admin where id=#{id}")
    Admin findAdminById(@Param("id") int id);

    /**
     * 查询所有的管理员
     * @return
     */
    @Select("select id,username,password from admin")
    List<Admin> findAllAdmins();

}
```

2、核心配置文件中配置

```
<mappers>
    <mapper class="com.xinzhi.dao.AdminMapper"/>
</mappers>
```

3、进行测试

```
/**
 * author IT楠老师
 * date 2020/5/12
 */
public class TestAdmin {

    private SqlSession session;

    @Before
    public void before() {
        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream = Resources.getResourceAsStream(resource);
            SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
            session = sqlSessionFactory.openSession();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

B站 IT楠老师

```
    @Test
    public void testSaveAdmin() {
        AdminMapper mapper = session.getMapper(AdminMapper.class);
        Admin admin = new Admin(1,"微微姐","12345678");
        int i = mapper.saveAdmin(admin);
        System.out.println(i);
    }

    @Test
    public void testUpdateAdmin() {
        AdminMapper mapper = session.getMapper(AdminMapper.class);
        Admin user = new Admin(1,"磊磊哥","12345678");
        int i = mapper.updateAdmin(user);
        System.out.println(i);
    }

    @Test
    public void testDeleteAdmin(){
        AdminMapper mapper = session.getMapper(AdminMapper.class);
        int i = mapper.deleteAdmin(2);
        System.out.println(i);
    }

    @Test
    public void testGetAdminById(){
        AdminMapper mapper = session.getMapper(AdminMapper.class);
        Admin admin = mapper.findAdminById(1);
        System.out.println(admin);
    }

    @Test
    public void testGetAllAdmins(){
        AdminMapper mapper = session.getMapper(AdminMapper.class);
        List<Admin>  admins = mapper.findAllAdmins();
        for (Admin admin : admins) {
            System.out.println(admin);
        }
    }

    @After
    public void close(){
        session.commit();
        session.close();
    }
}
```

BIT IT博老师

五、配置文件解读

> **mybatis的配置文件分为核心配置文件和mapper配置文件**

1、核心配置文件

- mybatis-config.xml 系统核心配置文件
- 核心配置文件主要配置mybatis一些**基础组件和加载资源**，核心配置文件中的元素常常能影响mybatis的整个运行过程。
- 能配置的内容如下，**顺序不能乱**：

```
1.properties是一个配置属性的元素
2.settings设置，mybatis最为复杂的配置也是最重要的，会改变mybatis运行时候的行为
3.typeAliases别名（在TypeAliasRegistry中可以看到mybatis提供了许多的系统别名）
4.typeHandlers 类型处理器（比如在预处理语句中设置一个参数或者从结果集中获取一个参数时候，都会用到类型处理器，在TypeHandlerRegistry中定义了很多的类型处理器）
5.objectFactory 对象工厂（myabtis在构建一个结果返回的时候，会使用一个ObjectFactory去构建pojo）
6.plugins 插件
7.environments 环境变量
    environment 环境变量
      transactionManager 事务管理器
      dataSource 数据源
      databaseIdProvider 数据库厂商标识
8.mappers 映射器
```

> **找几个重要的讲解一下**

（1）environments元素

environments可以为mybatis配置多环境运行，将SQL映射到多个不同的数据库上，必须指定其中一个为默认运行环境（通过default指定），如果想切换环境修改default的值即可。

> **最常见的就是，生产环境和开发环境，两个环境切换必将导致数据库的切换。**

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

  <environment id="product">
```

```xml
        <transactionManager type="JDBC">
          <property name="..." value="..."/>
        </transactionManager>
        <dataSource type="POOLED">
          <property name="driver" value="${driver}"/>
          <property name="url" value="${url}"/>
          <property name="username" value="${username}"/>
          <property name="password" value="${password}"/>
        </dataSource>
  </environment>
</environments>
```

- dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。
- 数据源是必须配置的。
- 有三种内建的数据源类型

  ```
  type="[UNPOOLED|POOLED|JNDI]")
  - unpooled：这个数据源的实现只是每次被请求时打开和关闭连接。
  - pooled：这种数据源的实现利用"池"的概念将 JDBC 连接对象组织起来 ，这是一种使得并发 Web 应用快速响应请求的流行处理方式。
  - jndi：这个数据源的实现是为了能在如 Spring 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。
  ```

- 数据源也有很多第三方的实现，比如druid，hikari，dbcp，c3p0等等....
- 这两种事务管理器类型都不需要设置任何属性。
- 具体的一套环境，通过设置id进行区别，id保证唯一！
- 子元素节点：transactionManager - [事务管理器]

  ```xml
  <!-- 语法 -->
  <transactionManager type="[ JDBC | MANAGED ]"/>
  ```

- 子元素节点：**数据源（dataSource）**

（2）mappers元素

> **mappers的存在就是要对写好的mapper和xml进行统一管理**
>
> **要不然系统怎么知道我写了哪些mapper**

通常这么引入

```xml
<mappers>
    <!-- 使用相对于类路径的资源引用 -->
    <mapper resource="com/xinzhi/dao/userMapper.xml"/>
    <!-- 面向注解时使用全类名 -->
    <mapper class="com.xinzhi.dao.AdminMapper"/>
</mappers>

。。。还有其他的方式
```

Mapper文件

```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="com.xinzhi.mapper.UserMapper">

</mapper>
```

- namespace中文意思：命名空间，作用如下：
- namespace的命名必须跟某个接口同名，这才能找的到啊！

（3）Properties元素

数据库连接信息我们最好放在一个单独的文件中。

1、 在资源目录下新建一个db.properties

```properties
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/ssm?useSSL=true&useUnicode=true&characterEncoding=utf8
username=root
password=root
```

2、 将文件导入properties 配置文件

```xml
<configuration>
   <!--导入properties文件-->
   <properties resource="db.properties"/>

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

B站IT韩老师

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

(4) 设定别名

> 别名小插曲里已将讲了，这里就不重复说了。

(5) 其他配置浏览

> settings能对我的一些核心功能进行配置，如懒加载、日志实现、缓存开启关闭等

> 完整的 settings 元素：

```
<settings>
    <!---->
     <setting name="cacheEnabled" value="true"/>
    <!---->
     <setting name="lazyLoadingEnabled" value="true"/>
    <!---->
     <setting name="multipleResultSetsEnabled" value="true"/>
     <setting name="useColumnLabel" value="true"/>
     <setting name="useGeneratedKeys" value="false"/>
     <setting name="autoMappingBehavior" value="PARTIAL"/>
     <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>
     <setting name="defaultExecutorType" value="SIMPLE"/>
     <setting name="defaultStatementTimeout" value="25"/>
     <setting name="defaultFetchSize" value="100"/>
     <setting name="safeRowBoundsEnabled" value="false"/>
     <setting name="mapUnderscoreToCamelCase" value="false"/>
     <setting name="localCacheScope" value="SESSION"/>
     <setting name="jdbcTypeForNull" value="OTHER"/>
     <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>
</settings>
```

> 设置一览表

设置参数	描述	有效值	默认值
cacheEnabled	该配置影响的所有映射器中配置的缓存的全局开关。	true | false	true
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。特定关联关系中可通过设置fetchType属性来覆盖该项的开关状态。	true | false	false
aggressiveLazyLoading	当启用时，带有延迟加载属性的对象的加载与否完全取决于对任意延迟属性的调用；反之，每种属性将会按需加载。	true | false	true
multipleResultSetsEnabled	是否允许单一语句返回多结果集（需要兼容驱动）。	true | false	true
useColumnLabel	使用列标签代替列名。不同的驱动在这方面会有不同的表现，具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。	true | false	true
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要驱动兼容。如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。	true | false	False
autoMappingBehavior	指定 MyBatis 是否以及如何自动映射指定的列到字段或属性。NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。FULL 会自动映射任意复杂的结果集（包括嵌套和其他情况）。	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）；BATCH 执行器将重用语句并执行批量更新。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置超时时间，它决定驱动等待数据库响应的秒数。	Any positive integer	Not Set (null)
safeRowBoundsEnabled	允许在嵌套语句中使用行分界（RowBounds）。	true | false	False
mapUnderscoreToCamelCase	是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A_COLUMN 到经典 Java 属性名 aColumn 的类似映射。	true | false	False
localCacheScope	MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。	SESSION | STATEMENT	SESSION
jdbcTypeForNull	当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。	JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER	OTHER
lazyLoadTriggerMethods	指定哪个对象的方法触发一次延迟加载。	A method name list separated by commas	equals,clone,hashCode,toString
defaultScriptingLanguage	指定动态 SQL 生成的默认语言。	A type alias or fully qualified class name.	org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver
callSettersOnNulls	指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这对于有 Map.keySet() 依赖或 null 值初始化的时候是有用的。注意原始类型（int、boolean等）是不能设置成 null 的。	true | false	false
logPrefix	指定 MyBatis 增加到日志名称的前缀。	Any String	Not set
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	Not set
proxyFactory	为 Mybatis 用来创建具有延迟加载能力的对象设置代理工具。		

六、动态sql

1、 概述

MyBatis提供了对SQL语句动态的组装能力，大量的判断都可以在 MyBatis的映射XML文件里面配置，以达到许多我们需要大量代码才能实现的功能，大大减少了我们编写代码的工作量。

动态SQL的元素

元素	作用	备注
if	判断语句	单条件分支判断
choose、when、otherwise	相当于Java中的 case when语句	多条件分支判断
trim、where、set	辅助元素	用于处理一些SQL拼装问题
foreach	循环语句	在in语句等列举条件常用

2、if元素（非常常用）

if元素相当于Java中的if语句，它常常与test属性联合使用。现在我们要根据name去查找学生，但是name是可选的，如下所示：

```
<select id="findUserById" resultType="com.xinzhi.entity.User">
    select id,username,password from user
    where 1 =1
    <if test="id != null and id != ''">
        AND id = #{id}
    </if>
    <if test="username != null and username != ''">
        AND username = #{username}
    </if>
    <if test="password != null and password != ''">
        AND password = #{password}
    </if>
</select>
```

3、choose、when、otherwise元素

选一个

有些时候我们还需要多种条件的选择，在Java中我们可以使用switch、case、default语句，而在映射器的动态语句中可以使用choose、when、otherwise元素。

B站IT辅老师

```
<!-- 有name的时候使用name搜索，没有的时候使用id搜索 -->
<select id="select" resultType="com.xinzhi.entity.User">
    SELECT * FROM user
    WHERE 1=1
    <choose>
        <when test="name != null and name != ''">
            AND username LIKE concat('%', #{username}, '%')
        </when>
        <when test="id != null">
            AND id = #{id}
        </when>
    </choose>
</select>
```

4、where元素

上面的select语句我们加了一个 `1=1` 的绝对true的语句，目的是为了防止语句错误，变成 `SELECT * FROM student WHERE` 这样where后没有内容的错误语句。这样会有点奇怪，此时可以使用 `<where>` 元素。

```
<select id="findUserById" resultType="com.xinzhi.entity.User">
    select id,username,password from user
    <where>
        <if test="id != null and id != ''">
            AND id = #{id}
        </if>
        <if test="username != null and username != ''">
            AND username = #{username}
        </if>
        <if test="password != null and password != ''">
            AND password = #{password}
        </if>
    </where>
</select>
```

5、trim元素

有时候我们要去掉一些特殊的SQL语法，比如常见的and、or，此时可以使用trim元素。trim元素意味着我们需要去掉一些特殊的字符串，prefix代表的是语句的前缀，而prefixOverrides代表的是你需要去掉的那种字符串，suffix表示语句的后缀，suffixOverrides代表去掉的后缀字符串。

Bili 小博老师

```
<select id="select" resultType="com.xinzhi.entity.User">
    SELECT * FROM user
    <trim prefix="WHERE" prefixOverrides="AND">
        <if test="username != null and username != ''">
            AND username LIKE concat('%', #{username}, '%')
        </if>
        <if test="id != null">
            AND id = #{id}
        </if>
    </trim>
</select>
```

6、set元素

在update语句中，如果我们只想更新某几个字段的值，这个时候可以使用set元素配合if元素来完成。**注意：set元素遇到,会自动把,去掉。**

```
<update id="update">
    UPDATE user
    <set>
        <if test="username != null and username != ''">
            username = #{username},
        </if>
        <if test="password != null and password != ''">
            password = #{password}
        </if>
    </set>
    WHERE id = #{id}
</update>
```

7、foreach元素

foreach元素是一个循环语句，它的作用是遍历集合，可以支持数组、List、Set接口。

```
<select id="select" resultType="com.xinzhi.entity.User">
    SELECT * FROM user
    WHERE id IN
    <foreach collection="ids" open="(" close=")" separator="," item="id">
        #{id}
    </foreach>
</select>
```

- collection配置的是传递进来的参数名称
- item配置的是循环中当前的元素。
- index配置的是当前元素在集合的位置下标。
- open和 close配置的是以什么符号将这些集合元素包装起来。
- separator是各个元素的间隔符。

8、SQL片段

B站IT辅老师

有时候可能某个 sql 语句我们用的特别多，为了增加代码的重用性，简化代码，我们需要将这些代码抽取出来，然后使用时直接调用。

提取SQL片段：

```
<sql id="if-title-author">
   <if test="title != null">
      title = #{title}
   </if>
   <if test="author != null">
      and author = #{author}
   </if>
</sql>
```

引用SQL片段：

```
<select id="queryBlogIf" parameterType="map" resultType="blog">
  select * from blog
   <where>
       <!-- 引用 sql 片段，如果refid 指定的不在本文件中，那么需要在前面加上 namespace -->
       <include refid="if-title-author"></include>
       <!-- 在这里还可以引用其他的 sql 片段 -->
   </where>
</select>
```

七、日志配置

> **配置日志的一个重要原因是想在调试的时候能观察到sql语句的输出，能查看中间过程**

1、标准日志实现

指定 MyBatis 应该使用哪个日志记录实现。如果此设置不存在，则会自动发现日志记录实现。

STD：standard out：输出

STDOUT_LOGGING：标准输出日志

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

这就好了，执行一下看看。

2、组合log4j完成日志功能（扩展）

使用步骤：

1、导入log4j的包

```
<dependency>
   <groupId>log4j</groupId>
   <artifactId>log4j</artifactId>
   <version>1.2.17</version>
</dependency>
```

2、配置文件编写 log4j.properties

```
#将等级为DEBUG的日志信息输出到console和file这两个目的地，console和file的定义在下面的代码
log4j.rootLogger=DEBUG,console,file

#控制台输出的相关设置
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.Target = System.out
log4j.appender.console.Threshold=DEBUG
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=[%c]-%m%n

#文件输出的相关设置
log4j.appender.file = org.apache.log4j.RollingFileAppender
log4j.appender.file.File=./log/xinzhi.log
log4j.appender.file.MaxFileSize=10mb
log4j.appender.file.Threshold=DEBUG
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%p][%d{yy-MM-dd}][%c]%m%n

#日志输出级别
log4j.logger.org.mybatis=DEBUG
log4j.logger.java.sql=DEBUG
log4j.logger.java.sql.Statement=DEBUG
log4j.logger.java.sql.ResultSet=DEBUG
log4j.logger.java.sql.PreparedStatement=DEBUG
```

3、setting设置日志实现

```
<settings>
   <setting name="logImpl" value="LOG4J"/>
</settings>
```

4、在程序中使用Log4j进行输出！

```
@Test
public void findAllUsers() {
   UserMapper mapper = session.getMapper(UserMapper.class);
   List<User> users = mapper.selectUser();
   for (User user: users){
       System.out.println(user);
  }
}
```

B站IT辅老师

5、测试，看控制台输出！

八、数据关系处理

- 部门和员工的关系，一个部门多个员工，一个员工属于一个部门
- 那我们可以采取两种方式来维护关系，一种在一的一方，一种在多的一方！

> **数据库设计**

```
CREATE TABLE `dept` (
`id` INT(10) NOT NULL,
`name` VARCHAR(30) DEFAULT NULL,
PRIMARY KEY (`id`)
)

INSERT INTO dept VALUES (1, '欣知开发四部');

CREATE TABLE `employee` (
`id` INT(10) NOT NULL,
`name` VARCHAR(30) DEFAULT NULL,
`did` INT(10) DEFAULT NULL,
PRIMARY KEY (`id`),
CONSTRAINT `fk_did` FOREIGN KEY (`did`) REFERENCES `dept` (`id`)
)


INSERT INTO employee VALUES (1, '邸智伟', 1);
INSERT INTO employee VALUES (2, '成虹', 1);
INSERT INTO employee VALUES (3, '康永亮', 1);
INSERT INTO employee VALUES (4, '杨春旺', 1);
INSERT INTO employee VALUES (5, '陈建强', 1);
```

1、自己手动维护

> **自己维护，多写几个语句，使用java组装，有时候使用mybatis维护关系反而复杂，不如自己维护。**

1、先搜索部门

2、根据部门搜索员工

3、手动拼装

2、在多的一方维护关系

1、编写实体类

```
/**
```

BIT IT辅老师

```
 * author IT楠老师
 * date 2020/5/15
 */
@Data
public class Dept {
    private int id;
    private String name;

}
/**
 * author IT楠老师
 * date 2020/5/15
 */
@Data
public class Employee {
    private int id;
    private String name;
    //维护关系
    private Dept dept;
}
```

2、编写实体类对应的Mapper接口

```
public interface DeptMapper {
}
public interface EmployeeMapper {
}
```

3、编写Mapper接口对应的 mapper.xml配置文件

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

</mapper>


<?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.xinzhi.dao.EmployeeMapper">

</mapper>
```

——

按查询嵌套，级联查询处理，就像SQL中的子查询

1、写方法

B站IT楠老师

```
/**
 * 获取所有的员工
 * @return
*/
List<Employee> findAllEmployees();
```

2、mapper处理

```
<?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.xinzhi.dao.EmployeeMapper">
   <resultMap id="EmployeeDept" type="com.xinzhi.entity.Employee">
       <!--association关联属性 property属性名 javaType属性类型 column在多的一方的表中的列名-->
       <association property="dept"  column="dId" javaType="com.xinzhi.entity.Dept"
select="getDept"/>
   </resultMap>


   <select id="findAllEmployees" resultMap="EmployeeDept">
    select * from employee
   </select>


   <select id="getDept" resultType="com.xinzhi.entity.Dept">
      select * from dept where id = #{id}
   </select>

</mapper>
```

3、编写完毕去Mybatis配置文件中，注册Mapper

4、测试

```
@Test
public void testFindAllEmployees() {
    EmployeeMapper mapper = session.getMapper(EmployeeMapper.class);
    List<Employee> allEmployees = mapper.findAllEmployees();
    for (Employee allEmployee : allEmployees) {
        System.out.println(allEmployee);
    }
}
```

5、结果

B站IT辅老师

```
[com.xinzhi.dao.EmployeeMapper.findAllEmployees]-==>  Preparing: select * from
employee
[com.xinzhi.dao.EmployeeMapper.findAllEmployees]-==> Parameters:
[com.xinzhi.dao.EmployeeMapper.getDept]-====>  Preparing: select * from dept where id
= ?
[com.xinzhi.dao.EmployeeMapper.getDept]-====> Parameters: 1(Integer)
[com.xinzhi.dao.EmployeeMapper.getDept]-<====      Total: 1
[com.xinzhi.dao.EmployeeMapper.findAllEmployees]-<==      Total: 5
Employee(id=1, name=邸智伟, dept=Dept(id=1, name=开发四部))
Employee(id=2, name=成虹, dept=Dept(id=1, name=开发四部))
Employee(id=3, name=康永亮, dept=Dept(id=1, name=开发四部))
Employee(id=4, name=杨春旺, dept=Dept(id=1, name=开发四部))
Employee(id=5, name=陈建强, dept=Dept(id=1, name=开发四部))
```

——

按结果嵌套处理，就像SQL中的联表查询

除了上面这种方式，还有其他思路吗?

我们还可以按照结果进行嵌套处理;

1、接口方法编写

```
List<Employee> findAllEmployees2();
```

2、编写对应的mapper文件

```
<select id="findAllEmployees2" resultMap="EmployeeDept2" >
    select e.id eid,e.name ename, d.id did,d.name dname
    from employee e,dept d
    where d.id = e.did
</select>

<resultMap id="EmployeeDept2" type="com.xinzhi.entity.Employee">
    <id property="id" column="eid"/>
    <result property="name" column="ename"/>
    <!--关联对象property 关联对象在Student实体类中的属性-->
    <association property="dept" javaType="com.xinzhi.entity.Dept">
        <id property="id" column="did"/>
        <result property="name" column="dname"/>
    </association>
</resultMap>
```

3、去mybatis-config文件中配置

```
<mapper resource="com/xinzhi/dao/DeptMapper.xml"/>
<mapper resource="com/xinzhi/dao/EmployeeMapper.xml"/>
```

4、测试

```
@Test
public void testFindAllEmployees2() {
    EmployeeMapper mapper = session.getMapper(EmployeeMapper.class);
    List<Employee> allEmployees = mapper.findAllEmployees2();
    for (Employee allEmployee : allEmployees) {
        System.out.println(allEmployee);
    }
}
```

3、在一的一方维护关系

在部门处维护关系，此处可以联想订单和订单详情。

实体类编写

```
/**
 * author IT楠老师
 * date 2020/5/15
 */
@Data
public class Dept {
    private int id;
    private String name;
    //用于关系维护
    List<Employee> employees;
}

/**
 * author IT楠老师
 * date 2020/5/15
 */
@Data
public class Employee {
    private int id;
    private String name;
}
```

按结果嵌套处理

1、写方法

```
/**
 * author IT楠老师
 * date 2020/5/15
 */
public interface DeptMapper {
    Dept findDeptById(int id);
}
```

B站 IT楠老师

2、写配置文件

```
 <select id="findDeptById" resultMap="DeptEmployee">
       select e.id eid,e.name ename, d.id did,d.name dname
       from employee e,dept d
       where d.id = e.did and e.id=#{id}
   </select>

   <resultMap id="DeptEmployee" type="com.xinzhi.entity.Dept">
       <id property="id" column="did"/>
       <result  property="name" column="dname"/>
       <collection property="employees" ofType="com.xinzhi.entity.Employee">
           <result property="id" column="eid" />
           <result property="name" column="ename" />
       </collection>
   </resultMap>
```

3、将Mapper文件注册到MyBatis-config文件中

```
<mappers>
  <mapper resource="com/xinzhi/dao/DeptMapper.xml"/>
</mappers>
```

4、测试

按查询嵌套处理

1、TeacherMapper接口编写方法

```
Dept findDeptById2(int id);
```

2、编写接口对应的Mapper配置文件

```
<resultMap id="DeptEmployee2" type="com.xinzhi.entity.Dept">
    <!--column是一对多的外键 ， 写的是一的主键的列名-->
    <collection property="employees" javaType="ArrayList"
ofType="com.xinzhi.entity.Employee" column="id" select="getEmployeeById"/>
</resultMap>

<select id="findDeptById2" resultMap="DeptEmployee2">
    select * from dept where id = #{id}
</select>
<select id="getEmployeeById" resultType="com.xinzhi.entity.Employee">
    select * from employee where did = #{id}
</select>
```

3、将Mapper文件注册到MyBatis-config文件中

B站 IT楠老师

```
<mappers>
  <mapper resource="com/xinzhi/dao/DeptMapper.xml"/>
</mappers>
```

4、测试

```
@Test
public void testFindDeptById() {
    DeptMapper mapper = session.getMapper(DeptMapper.class);
    Dept dept = mapper.findDeptById2(1);
    System.out.println(dept);
}
```

结果：

```
[com.xinzhi.dao.DeptMapper.findDeptById2]-==>  Preparing: select * from dept where id
= ?
[com.xinzhi.dao.DeptMapper.findDeptById2]-==> Parameters: 1(Integer)
[com.xinzhi.dao.DeptMapper.getEmployeeById]-====>  Preparing: select * from employee
where did = ?
[com.xinzhi.dao.DeptMapper.getEmployeeById]-====> Parameters: 1(Integer)
[com.xinzhi.dao.DeptMapper.getEmployeeById]-<====      Total: 5
[com.xinzhi.dao.DeptMapper.findDeptById2]-<==      Total: 1
Dept(id=0, name=开发四部, employees=[Employee(id=1, name=邸智伟, dept=null),
Employee(id=2, name=成虹, dept=null), Employee(id=3, name=康永亮, dept=null),
Employee(id=4, name=杨春旺, dept=null), Employee(id=5, name=陈建强, dept=null)])
```

九、Mybatis缓存

1、为什么要用缓存?

- 如果缓存中有数据，就不用从数据库获取，大大提高系统性能。
- mybatis提供一级缓存和二级缓存

2、一级缓存:

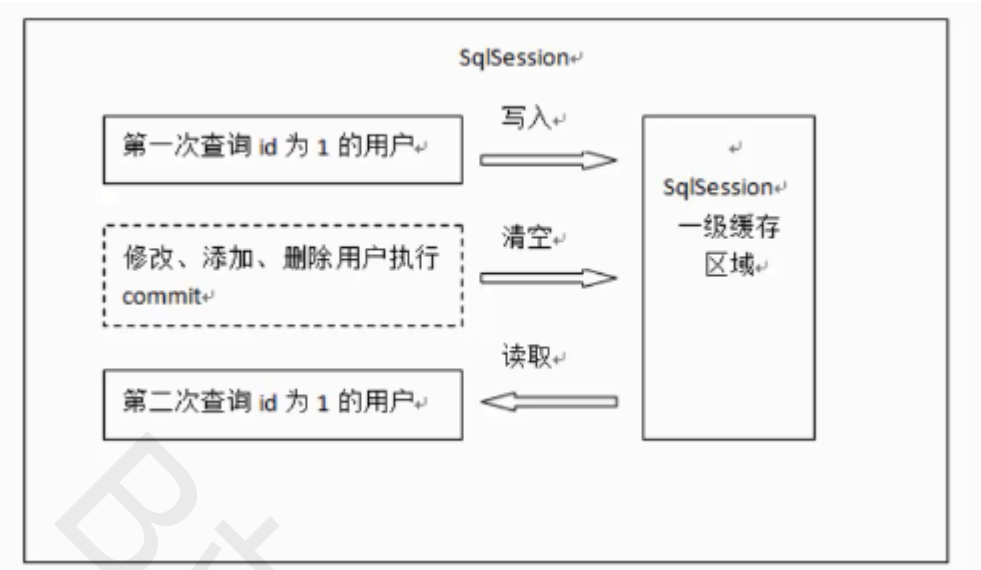
> **一级缓存是sqlsession级别的缓存**

- 在操作数据库时，需要构造sqlsession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据
- 不同的sqlsession之间的缓存区域是互相不影响的。

B站 小陶老师

一级缓存工作原理：

图解：



- 第一次发起查询sql查询用户id为1的用户，先去找缓存中是否有id为1的用户，如果没有，再去数据库查询用户信息。得到用户信息，将用户信息存储到一级缓存中。
- 如果sqlsession执行了commit操作（插入，更新，删除），会清空sqlsession中的一级缓存，避免脏读
- 第二次发起查询id为1的用户，缓存中如果找到了，直接从缓存中获取用户信息
- mybatis默认支持并开启一级缓存。

一级缓存演示

1、必须配置日志，要不看不见

2、编写接口方法

```
//根据id查询用户
User findUserById(@Param("id") int id);
```

3、接口对应的Mapper文件

```
<select id="findUserById" resultType="com.xinzhi.entity.User">
  select * from user where id = #{id}
</select>
```

4、测试

Bit 叶精老师

```
@Test
public void testFindUserById(){
    UserMapper mapper = session.getMapper(UserMapper.class);
    User user1 = mapper.findUserById(1);
    System.out.println(user1);
    User user2 = mapper.findUserById(3);
    System.out.println(user2);
    User user3 = mapper.findUserById(1);
    System.out.println(user3);
}
```

5、通过日志分析

```
[com.xinzhi.dao.UserMapper.findUserById]-==>  Preparing: select id,username,password
from user where id = ?
[com.xinzhi.dao.UserMapper.findUserById]-==> Parameters: 1(Integer)
[com.xinzhi.dao.UserMapper.findUserById]-<==      Total: 1
User{id=1, username='楠哥', password='123456'}         ---->ID为1，第一次有sql
[com.xinzhi.dao.UserMapper.findUserById]-==>  Preparing: select id,username,password
from user where id = ?
[com.xinzhi.dao.UserMapper.findUserById]-==> Parameters: 3(Integer)
[com.xinzhi.dao.UserMapper.findUserById]-<==      Total: 1
User{id=3, username='磊哥', password='987654'}         ---->ID为3，第一次有sql
User{id=1, username='楠哥', password='123456'}         ---->ID为1，第二次无sql，走缓存
```

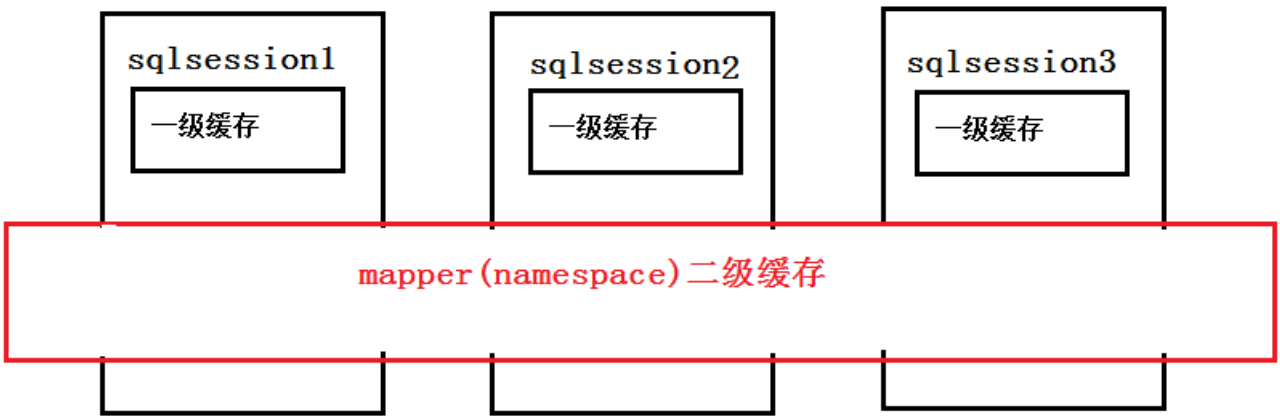
一级缓存失效

1. sqlSession不同
2. 当sqlSession对象相同的时候，查询的条件不同，，原因是第一次查询时候一级缓存中没有第二次查询所需要的数据
3. 当sqlSession对象相同,两次查询之间进行了插入的操作
4. 当sqlSession对象相同,手动清除了一级缓存中的数据

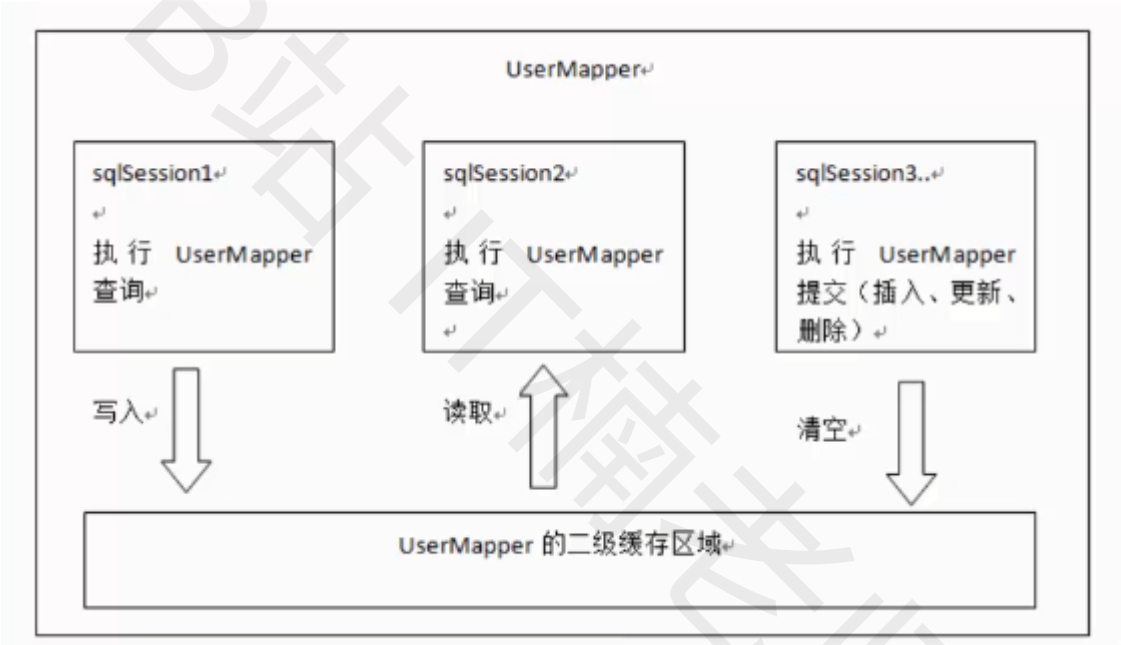
3、二级缓存:

二级缓存是mapper级别的缓存

- 多个sqlsession去操作同一个mapper的sql语句，多个sqlsession可以共用二级缓存，所得到的数据会存在二级缓存区域，
- 二级缓存是跨sqlsession的
- 二级缓存相比一级缓存的范围更大（按namespace来划分），多个sqlsession可以共享一个二级缓存



二级缓存实现原理



首先要手动开启mybatis二级缓存。

在config.xml设置二级缓存开关， 还要在具体的mapper.xml开启二级缓存

```
<settings>
    <!--开启二级缓存-->
    <setting name="cacheEnabled" value="true"/>
</settings>
<!-- 需要将映射的javabean类实现序列化 -->
```

class Student implements Serializable{}

```
<!--开启本Mapper的namespace下的二级缓存-->
<cache eviction="LRU" flushInterval="10000"/>
```

（1）cache属性的简介：

eviction回收策略（缓存满了的淘汰机制），目前MyBatis提供以下策略。

1. LRU（Least Recently Used）,最近最少使用的，最长时间不用的对象
2. FIFO（First In First Out）,先进先出，按对象进入缓存的顺序来移除他们
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，更积极的移除基于垃圾收集器状态和弱引用规则的对象。这里采用的是LRU， 移除最长时间不用的对形象

flushInterval:刷新间隔时间，单位为毫秒，

1. 这里配置的是100秒刷新，如果你不配置它，那么当SQL被执行的时候才会去刷新缓存。

size:引用数目，

1. 一个正整数，代表缓存最多可以存储多少个对象，不宜设置过大。设置过大会导致内存溢出。
   这里配置的是1024个对象

readOnly:只读，

1. 意味着缓存数据只能读取而不能修改，这样设置的好处是我们可以快速读取缓存，缺点是我们没有办法修改缓存，他的默认值是false，不允许我们修改

（2）操作过程：

sqlsession1查询用户id为1的信息，查询到之后，会将查询数据存储到二级缓存中。

如果sqlsession3去执行相同mapper下sql，执行commit提交，会清空该mapper下的二级缓存区域的数据

sqlsession2查询用户id为1的信息， 去缓存找 是否存在缓存，如果存在直接从缓存中取数据

禁用二级缓存：

在statement中可以设置useCache=false，禁用当前select语句的二级缓存，默认情况为true

```
<select id="getStudentById" parameterType="java.lang.Integer" resultType="Student"
useCache="false">
```

在实际开发中，针对每次查询都需要最新的数据sql，要设置为useCache="false" ，禁用二级缓存

flushCache标签：刷新缓存（清空缓存）

```
<select id="getStudentById" parameterType="java.lang.Integer" resultType="Student"
flushCache="true">
```

一般下执行完commit操作都需要刷新缓存，flushCache="true 表示刷新缓存，可以避免脏读

二级缓存应用场景

对于访问多的查询请求并且用户对查询结果实时性要求不高的情况下，可采用mybatis二级缓存，降低数据库访问量，提高访问速度，如电话账单查询

根据需求设置相应的**flushInterval**:刷新间隔时间，比如三十分钟，24小时等。

二级缓存局限性:

mybatis二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求:对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用mybatis的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为mybaits的二级缓存区域以mapper为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

二级缓存演示

先不进行配置

```
@Test
public void testFindUserCache() throws Exception {

    //使用不同的mapper
    UserMapper mapper1 = session.getMapper(UserMapper.class);
    User user1 = mapper1.findUserById(1);
    System.out.println(user1);
    //提交了就会刷到二级缓存，要不还在一级缓存，一定要注意
    session.commit();
    UserMapper mapper2 = session.getMapper(UserMapper.class);
    User user2 = mapper2.findUserById(1);
    System.out.println(user2);
    System.out.println(user1 == user2);
}
```

结果:

```
[com.xinzhi.dao.UserMapper.findUserById]-==>  Preparing: select id,username,password
from user where id = ?
[com.xinzhi.dao.UserMapper.findUserById]-==> Parameters: 1(Integer)
[com.xinzhi.dao.UserMapper.findUserById]-<==      Total: 1
User{id=1, username='楠哥', password='123456'}
[com.xinzhi.dao.UserMapper.findUserById]-==>  Preparing: select id,username,password
from user where id = ?
[com.xinzhi.dao.UserMapper.findUserById]-==> Parameters: 1(Integer)
[com.xinzhi.dao.UserMapper.findUserById]-<==      Total: 1
User{id=1, username='楠哥', password='123456'}
false               ---->两个对象不是一个，发了两个sql，说明缓存没有起作用
```

可以看见两次同样的sql，却都进库进行了查询。说明二级缓存没开。

配置二级缓存

1、开启全局缓存

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

2、使用二级缓存，这个写在mapper里

```
<!--开启本Mapper的namespace下的二级缓存-->
<cache eviction="LRU" flushInterval="10000" size="1024" readOnly="true"></cache>
<!--
创建了一个 LRU 缓存，每隔 100 秒刷新，最多可以存储 512 个对象，返回的对象是只读的。
-->
```

3、测试执行

```
[com.xinzhi.dao.UserMapper.findUserById]-==>  Preparing: select id,username,password
from user where id = ?
[com.xinzhi.dao.UserMapper.findUserById]-==> Parameters: 1(Integer)
[com.xinzhi.dao.UserMapper.findUserById]-<==      Total: 1
User{id=1, username='楠哥', password='123456'}
[com.xinzhi.dao.UserMapper]-Cache Hit Ratio [com.xinzhi.dao.UserMapper]: 0.5
User{id=1, username='楠哥', password='123456'}
true                  ---->两个对象一样了，就发了一个sql，说明缓存起了作用
```

3、第三方缓存--EhCache充当三级缓存

我们的三方缓存组件很对，最常用的比如ehcache，Memcached、redis等，我们以比较简单的ehcache为例。

1、引入依赖

```
<!-- https://mvnrepository.com/artifact/org.mybatis.caches/mybatis-ehcache -->
<dependency>
   <groupId>org.mybatis.caches</groupId>
   <artifactId>mybatis-ehcache</artifactId>
   <version>1.1.0</version>
</dependency>
```

2、修改mapper.xml中使用对应的缓存

```
<mapper namespace = “com.xinzhi.entity.User” >
       <cache type="org.mybatis.caches.ehcache.EhcacheCache" eviction="LRU"
flushInterval="10000" size="1024" readOnly="true"/>
</mapper>
```

Bili 小楠老师

3、添加ehcache.xml文件，ehcache配置文件，具体配置自行百度

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
        updateCheck="false">
   <!--
      diskStore：为缓存路径，ehcache分为内存和磁盘两级，此属性定义磁盘的缓存位置。参数解释如下：
      user.home – 用户主目录
      user.dir – 用户当前工作目录
      java.io.tmpdir – 默认临时文件路径
    -->
   <diskStore path="./tmpdir/Tmp_EhCache"/>

   <defaultCache
           eternal="false"
           maxElementsInMemory="10000"
           overflowToDisk="false"
           diskPersistent="false"
           timeToIdleSeconds="1800"
           timeToLiveSeconds="259200"
           memoryStoreEvictionPolicy="LRU"/>

</ehcache>
```

3、测试

```
[com.xinzhi.dao.UserMapper.findUserById]-==>  Preparing: select id,username,password
from user where id = ?
[com.xinzhi.dao.UserMapper.findUserById]-==> Parameters: 1(Integer)
[com.xinzhi.dao.UserMapper.findUserById]-<==      Total: 1
User{id=1, username='楠哥', password='123456'}
[com.xinzhi.dao.UserMapper]-Cache Hit Ratio [com.xinzhi.dao.UserMapper]: 0.5
User{id=1, username='楠哥', password='123456'}
true
```

全剧终，学到了就打个赏呗！加油，学习的每一天都是充实的一天。

推荐使用**微信支付**

Alm张楠(*楠)

微信支付

B站 IT楠老师