# Mybatis开发框架

## Mybatis简介

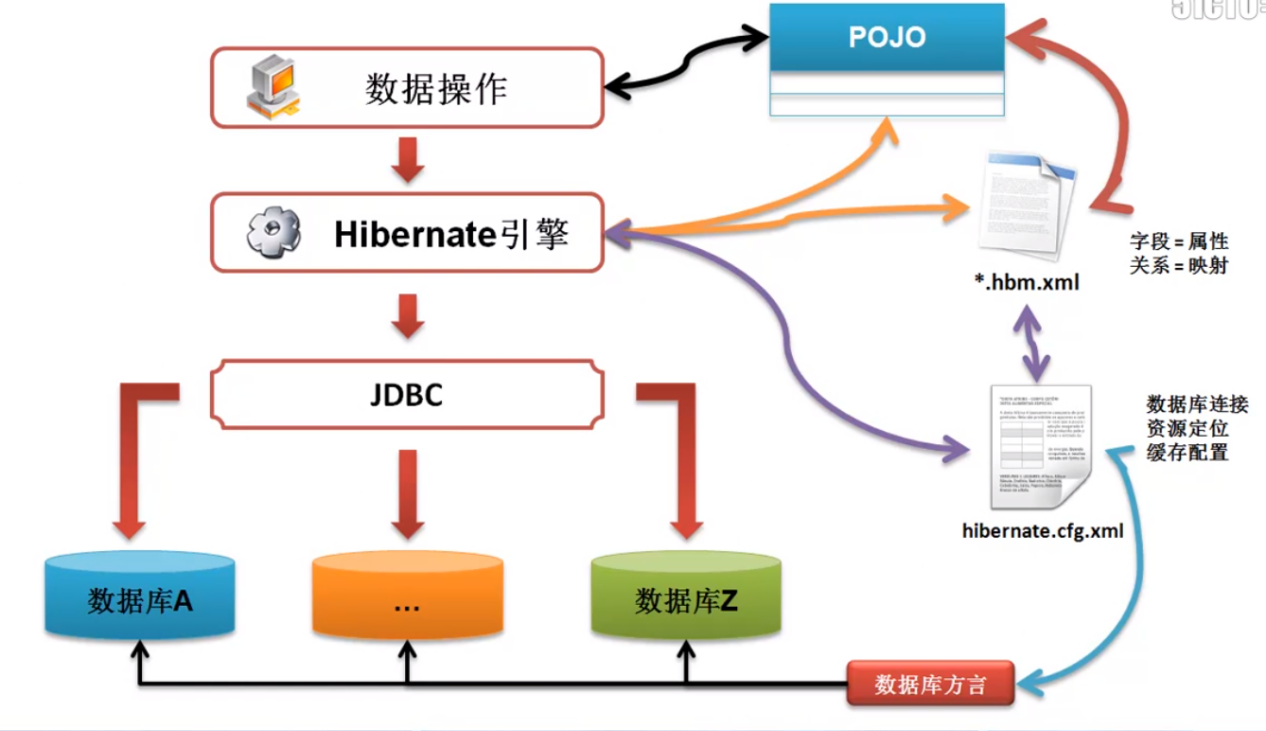
### Mybatis开发框架简介简介

JDBC是一个最原始的ORM操作组件（ORMapping），但是实现过于琐碎，但作为开发标准的操作就必须琐碎。后来为了解决这样的琐碎，开发者根据EJB（实体bean）的实现理念，推出了JDO、hibernate、Mybatis等。

这些组件中最具代表性的就是hibernate，但是hibernate存在缺点：其设计的最初动力来自于EJB组件，EJB中实体Bean的开发从标准来讲就是定义在数据层，如果使用hibernate开发，本质并没有带来多少便捷。

### Mybatis与hibernate区别

Hibernate设计理念：EJB的可移植思想。



Mybatis设计思想：让代码变得更简单，不过多的考虑那些复杂的数据库移植。

Hibernate整体优势如下：

1. 如果面对数据库移植问题，hibernate是不二选择。
2. Hibernate之中可以使用\*.hbm.xml文件及jpa实现关系配置。
3. 可以根据配置文件动态转换为SQL语句（数据库表结构）。

缺点：

1. Hibernate中需要处理（实现）的逻辑判断过于复杂。
2. 配置复杂，如果配置不当，或者操作不当，会直接带来严重的性能问题。

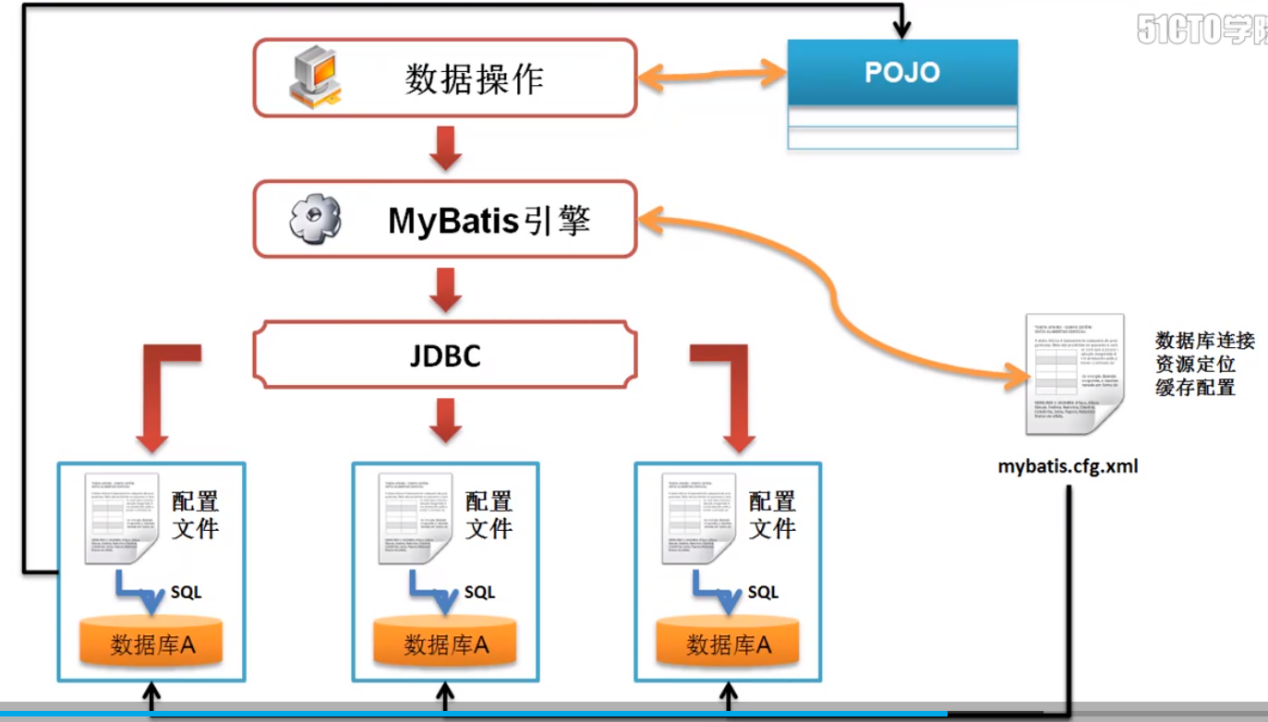
Hibernate最麻烦的一点在于：复杂更新以及数据查询的处理上，而且需要自己编写HQL（还和sql有点不一样）。如果直接使用SQL，就可能出现移植性消失的问题。

Mybatis没有考虑到hibernate中那些复杂的对象状态，它提供给用户高于jdbc的手工化，以及低于hibernate的全自动化的操作。即半自动化。

Mybatis框架有点：

1. 体积小，响应速度快；
2. 避免了复杂的逻辑性操作，可以完全实现数据层操作。
3. 提供了不亚于hibernate中的映射支持，入数据关联，识别器，annotation。

缺点：

1. 开发工具支持不到位，全程手写。
2. 不适合于数据库的移植操作。
3. 

## 配置并开发Mybatis程序

### 2.1 搭建Mybatis开发环境

1、将Mybatis开发包拷贝到项目中。最好是maven。

2、创建表

1. 在Mybatis也需要取得数据库连接及相关的资源配置，可以在src目录下建立一个mybatis-config.xml文件。

<?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"> <!-- 配置数据源的相关信息 -->

<environment id="development">

<!-- 此时的事务控制交由JDBC负责处理 -->

<transactionManager type="jdbc" /> <!-- 使用JDBC方式管理 -->

<dataSource type="POOLED"> <!-- 设置数据源类型，此时为POOLED -->

<property name="driver" value="org.gjt.mm.mysql.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mldn" />

<property name="username" value="root" />

<property name="password" value="mysqladmin" />

</dataSource>

</environment>

</environments>

</configuration>

### 2.2 数据的增加操作

在进行数据库连接配置的过程中，对于连接类型有如下几种：

* “POOLED”：表示所有的数据库连接都在数据库连接池中保存，性能是最好的。
* “UNPOOLED”：表示不使用数据库连接池；
* “JNDi”：表示使用服务器上配置好的数据库连接池组件；
* “jdbc”:这个操作指的是使用jdbc来实现数据库的事务控制;
* “MANAGED”：表示什么的不做，即不提交事务也不回滚事务，所有的事务交由容器控制。

1. 在cn.xxx.vo下创建Member类
2. 在cn.xxx.vo.mapping下创建映射文件MemberMapper.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">

<!-- 设置命名空间，可以与不同表的同类型操作进行区分，使用时以“空间名称.id”的方式调用 -->

<mapper namespace="cn.mldn.mapping.MemberNS">

<!-- 定义增加数据的操作配置，同时指定参数类型，建议与DAO的方法名称一致 -->

<insert id="doCreate" parameterType="cn.mldn.vo.Member">

INSERT INTO member(mid,name,age,birthday,salary)

VALUES (#{mid},#{name},#{age},#{birthday},#{salary}) ;

</insert>

</mapper>

1. 将映射文件配置到mybatis.xml文件中：

<mappers> <!-- 映射文件信息 -->

<mapper resource="cn/mldn/vo/mapping/News.xml" />

</mappers>

1. 进行具体的程序编写：

|-hibernate与mybatis操作类对比：

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 名称 | Hibernate | Mybatis |
| 1 | 读取配置 | Configuration | Reader、InputStream |
| 2 | 连接工厂 | SessionFactory | SqlSessionFactory |
| 3 | 连接 | Session | SqlSession |

对于以上给出的几个Mybatis操作类解释如下：

* 读取资源文件类：【抽象类】Reader、InputStream，都是IO;
* 通过CLASSPATH读取资源：【类】org.apache.ibatis.io.Resources，方法定义：

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法 | 类型 | 描述 |
| 1 | public static Reader getResourceAsReader(String resource) | 普通 | 读取指定的配置文件，以Reader的形式返回。 |
| 2 | public static InputStream getResourceAsStream(String resource) throws IOException | 普通 | 读取指定的配置文件，以InputStream的形式返回。 |

* 数据库连接工厂：【接口】org.apache.ibatis.session.SqlSessionFactory，定义有如下 方法：

|  |  |  |  |
| --- | --- | --- | --- |
| NO. | 方法 | 类型 | 描述 |
| 1 | SqlSession openSession(); | 普通 | 取得SqlSession对象 |

* 连接构造类：【类】org.apache.ibatis.session.SqlSessionFactoryBuilder，方法定义如下：

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法 | 类型 | 描述 |
| 1 | public SqlSessionFactory build(InputStream inputStream) | 普通 | 根据字节输入流创建SqlSesionFactory |
| 2 | public SqlSessionFactory build(Reader reader) | 普通 | 根据字符输入流创建SqlSesionFactory |

* 数据库操作：【接口】org.apache.ibatis.session.SqlSession，方法定义如下：

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法 | 类型 | 描述 |
| 1 | int insert(String statement); | 普通 | 执行指定命名空间的指定操作代码 |
| 2 | int insert(String statement, Object args); | 普通 | 执行指定命名空间的指定操作代码，并且传递操作的数据对象 |
| 3 | void commit(); | 普通 | 事务提交处理 |
| 4 | void rollback(); | 普通 | 事务的回滚处理 |

范例：实现数据的增加

**public static void** main(String[] args) **throws** Exception {  
 *//1、读取类路径下的Mybatis.config.xml进行解析* InputStream ins = Resources.*getResourceAsStream*(**"mybatis-config.xml"**);  
 *//2、创建SqlSessionFactory接口对象* SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(ins);  
 *//3、创建SqlSession接口对象* SqlSession sqlSession = sqlSessionFactory.openSession();  
 *//4、创建VO对象* Member member = **new** Member();  
 member.setName(**"杨幂"**);  
 member.setMid(**"yangmi"**);  
 member.setAge(18);  
 member.setBirthday(**new** Date());  
 member.setSalary(1000000.00);  
 *// 5、实现数据的保存处理* System.***out***.println(**"受影响行数："**+sqlSession.insert(**"cn.mldn.vo.MemberMapper.doCreate"**, member));  
 *// 6、关闭资源* sqlSession.commit();  
 sqlSession.close();  
 ins.close();  
}

### 2.3 配置log4j显示日志

1、将log4j.properties文件拷贝到类路径下：

*# Global logging configuration***log4j.rootLogger**=**INFO, stdout,logfile***# MyBatis logging configuration...  
#log4j.logger.org.mybatis.example.BlogMapper=TRACE  
# Console output...***log4j.appender.stdout**=**org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.layout**=**org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern**=**%5p [%t] - %m%n  
  
log4j.appender.logfile**=**org.apache.log4j.RollingFileAppender  
log4j.appender.logfile.File**=**logs/my.log  
log4j.appender.logfile.MaxFileSize**=**512KB***# Keep three backup files.***log4j.appender.logfile.MaxBackupIndex**=**3***# Pattern to output: date priority [category] - message***log4j.appender.logfile.layout**=**org.apache.log4j.PatternLayout  
log4j.appender.logfile.layout.ConversionPattern**=**%d %p [%c] - %m%n**

1. 添加日志依赖

<**dependency**>  
 <**groupId**>log4j</**groupId**>  
 <**artifactId**>log4j</**artifactId**>  
 <**version**>1.2.17</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-api</**artifactId**>  
 <**version**>1.7.25</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-log4j12</**artifactId**>  
 <**version**>1.7.25</**version**>  
 <**scope**>test</**scope**>  
</**dependency**>

<**dependency**>  
 <**groupId**>org.slf4j</**groupId**>  
 <**artifactId**>slf4j-simple</**artifactId**>  
 <**version**>1.7.25</**version**>  
</**dependency**>

1. 如果想要日志输出显示，就必须手动修改log4j.properties里面的配置;

*log4j.logger.命名没空间=TRACE*

以上是针对一个命名空间的，如果要配置多个：

*log4j.logger.包名=TRACE*

*针对于单个sql:*

<**mapper namespace="cn.mldn.vo.mapper.MemberMapper"**>  
 <**insert id="doCreate" parameterType="cn.mldn.vo.Member"**>  
 INSERT INTO member(mid,name,age,birthday,salary)VALUES(#{mid},#{name},#{age},#{birthday},#{salary})  
 </**insert**>  
</**mapper**>

log4j.logger.cn.mldn.vo.mapper.MemberMapper.doCreate=TRACE

1. 在mybatis.config.xml文件中添加：

<configuration>  
<settings>  
...  
**<setting name="logImpl" value="LOG4J"/>**  
...  
</settings>  
</configuration>

以下是mybatis官方文档给出的内容：

Many environments ship Commons Logging as a part of the application server classpath (good  
examples include Tomcat and WebSphere). It is important to know that in such environments,  
MyBatis will use Commons Logging as the logging implementation. In an environment like  
WebSphere this will mean that your Log4J configuration will be ignored because WebSphere supplies  
its own proprietary implementation of Commons Logging. This can be very frustrating because it will  
appear that MyBatis is ignoring your Log4J configuration (in fact, MyBatis is ignoring your Log4J  
configuration because MyBatis will use Commons Logging in such environments). If your application  
is running in an environment where Commons Logging is included in the classpath but you would  
rather use one of the other logging implementations you can select a different logging implementation  
by adding a setting in mybatis-config.xml file as follows:  
<configuration>  
<settings>  
...  
<setting name="logImpl" value="LOG4J"/>  
...  
</settings>  
</configuration>  
Valid values are SLF4J, LOG4J, LOG4J2, JDK\_LOGGING, COMMONS\_LOGGING,  
STDOUT\_LOGGING, NO\_LOGGING or a full qualified class name that implements  
org.apache.ibatis.logging.Log and gets an string as a constructor parameter.

大致意思是说在使用WebSphere服务器时，会忽略log4j配置，需要使用COMMONS\_LOGGING,

### 2.4 自定义MyBatisSessionFactory

范例：模仿Hibernate定义工厂类

**public class** MyBatisSessionFactory {  
 *// 主要的功能定义公共资源文件的名称，方便Resources类的读取操作* **private static final** String ***CONFIG\_FILE*** = **"mybatis.config.xml"**;  
 *// 定义SqlSessionFactory接口对象，用于通过它操作二级缓存以及重新取得SqlSession的配置* **private static** SqlSessionFactory *sqlSessionFactory* = **null**;  
 *// 定义ThreadLocal类的对象，可以保存的是SqlSession接口对象，主要被不同的层做引用操作* **private static** ThreadLocal<SqlSession> *threadLocal* = **new** InheritableThreadLocal<>();  
 *// 主要是读取mybatis.config.xml文件* **private static** InputStream *inputStream* = **null**;  
  
 **static** {  
 *// 在类加载的时候创建SqlSessionFactory接口对象  
 rebuildSessionFactory*();  
 }  
  
 */\*\*  
 \* 取得当前连接的SqlSession对象，如果当前没有SqlSession，则需要通过SqlSessionFactory创建新的Session对象  
 \* 如果当前已经存在有SqlSession对象，则通过ThreadLocal取得返回  
 \*  
 \** ***@return*** *返回SqlSession对象  
 \*/* **public static** SqlSession getSession() {  
 *// 通过ThreadLocal取得SqlSession* SqlSession sqlSession = *threadLocal*.get();  
 **if** (sqlSession == **null**) {  
 **if** (*sqlSessionFactory* == **null**) {  
 *// 创建SqlSessionFactory  
 rebuildSessionFactory*();  
 }  
 *// 创建新的session对象* sqlSession = *sqlSessionFactory*.openSession();  
 *threadLocal*.set(sqlSession);  
 }  
 **return** sqlSession;  
 }  
  
 */\*\*  
 \* 返回sqlSessionfactory  
 \*  
 \** ***@return*** *\*/* **public static** SqlSessionFactory getSessionFactory() {  
 **return** *sqlSessionFactory*;  
 }  
  
 */\*\*  
 \* 重新建立新的SqlSessionFactory接口对象  
 \*/* **public static void** rebuildSessionFactory() {  
 **try** {  
 *// 读取配置文件  
 inputStream* = Resources.*getResourceAsStream*(***CONFIG\_FILE***);  
 *// 创建SqlSessionFactory接口对象  
 sqlSessionFactory* = **new** SqlSessionFactoryBuilder().build(*inputStream*);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* 关闭当前sqlSession  
 \*/* **public static void** close() {  
 SqlSession sqlSession = *threadLocal*.get();  
 *threadLocal*.set(**null**);  
 **if** (sqlSession != **null**) {  
 sqlSession.close();  
 **if** (*inputStream* != **null**) {  
 **try** {  
 *inputStream*.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

## Mybatis实现CRUD

### 3.1 数据增加深入操作

1、为VO类增加别名

在mybatis.config.xml追加配置：

*<!--type为别名名称-->*<**typeAlias type="news" alias="cn.mldn.vo.News"**/>

在News.xml文件中引用如下：

*<!--News为mybatis.config.xml中配置的别名-->*<**insert id="doCreate" parameterType="News"**>  
 INSERT INTO news(title,pub\_date) VALUES (#{title},#{pubdate}) ;  
</**insert**>

1. 取得自动增长主键（需手动配置，hibernate默认配置，这是消耗性能的做法）

<**insert id="doCreate" parameterType="News" keyProperty="nid" useGeneratedKeys="true"**>  
 INSERT INTO news(title,pub\_date) VALUES (#{title},#{pubdate}) ;  
</**insert**>

### 3.2 数据的修改操作

如果要修改，则还是需要使用SqlSession所定义的方法：

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 描述 |
| 1 | int update(String statemet); | 普通 | 实现指定语句的执行 |
| 2 | int update(String statement, Object agrs); | 普通 | 使用指定的内容实现数据的修改 |

理论上调用insert方法执行修改语句也可以成功，当然插入操作，也可以使用update方法操作成功，但最好还是根据标准来。

MyBatisSessionFactory.*getSession*().update(**"cn.mldn.vo.mapper.NewsMapper.doUpdate"**, vo)

MyBatisSessionFactory.*getSession*().insert(**"cn.mldn.vo.mapper.NewsMapper.doUpdate"**, vo)

### 3.3 数据删除操作

如果要删除，则还是需要使用SqlSession所定义的方法：

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 描述 |
| 1 | int delete(String statemet); | 普通 | 执行删除处理 |
| 2 | int delete(String statemet, Object args); | 普通 | 使用指定的内容实现数据的删除 |

MyBatisSessionFactory.*getSession*().delete(**"cn.mldn.vo.mapper.NewsMapper.doDelete"**, 3)

理论上调用insert、update方法执行删除语句也可以成功，但最好还是根据标准来。

其实上，在Mybatis实现的时候，底层本质还是jdbc封装，在执行所有更新操作时候都调用executeUpdate()方法，查询调用query()方法。但为啥Mybatis暴露给用户的都是对应的方法呢，因为接口是标准，应该按照标准来。

另外mybatisinsert、update、delete、查询等没有传递参数的方法，本质上调用的还是有参数方法：

**public int** update(String statement) {  
 **return this**.update(statement, **(Object)null**);  
}

### 3.4 数据的基本查询

对于数据的查询分为两种情况考虑：

1. 查询单个返回数据（单个值或vo对象）
2. 查询一组返回数据

SqlSession接口中定义的查询方法：

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 描述 |
| 1 | <E> List<E> selectList(String statement); | 普通 | 执行指定查询，返回一组对象 |
| 2 | <E> List<E> selectList(String statement, Object args); | 普通 | 查询指定内容，返回一组对象 |
| 3 | <T> T selectOne(String statement); | 普通 | 查询返回单个结果 |
| 4 | <E> List<E> selectList(String var1, Object var2); | 普通 | 查询返回单个结果，同时设置查询参数 |

范例：查询单个VO对象

如果存在对象属性名称与数据库字段不同，则需要处理：

方法一：如果属性少可以用别名的方式解决：select user\_name as username from xxx;

方法二：在对应映射文件中增加vo类与数据库字段的映射；

*<!-- 结果转换映射，一般都会在查询的时候使用 -->*<**resultMap id="NewsResultMap" type="News"**>  
 <**id property="nid" column="nid"**/>  
 <**result property="title" column="title"**/>  
 <**result property="pubdate" column="pub\_date"**/>  
</**resultMap**>

<**select id="queryNews" resultMap="NewsResultMap" parameterType="int"**>  
 select *\** from news where nid=#{id}  
</**select**>

查询单个数据，不止是单个对象，也有可能是统计，求和等。

范例：统计表中的全部数据量

<**select id="coutNews" resultType="int"**>  
 select *count*(1) from news  
</**select**>

MyBatisSessionFactory.*getSession*().selectOne(**"cn.mldn.vo.mapper.NewsMapper.coutNews"**);

**整个过程中，最大的亮点在于没有强制类型存在。**

范例：查询表中的全部数据

<**select id="queryAll" resultMap="NewsResultMap"**>  
 select *\** from news  
</**select**>

List<News> list = MyBatisSessionFactory.*getSession*().selectList(**"cn.mldn.vo.mapper.NewsMapper.queryAll"**);  
MyBatisSessionFactory.*getSession*().commit();  
MyBatisSessionFactory.*getSession*().close();  
  
Iterator<News> itre = list.iterator();  
**while** (itre.hasNext()) {  
 System.***out***.println(itre.next());  
}  
System.***out***.println(list.size());

### 3.5 数据分页查询

Mybatis对于参数的传递分为两种形式：

1. 以内容替换的方式设置参数：${替换变量},注意这种形式不参与预编译，极有可能发生sql注入;
2. 传递的是操作的数值：#{数值}；

SELECT \* FROM 表 WHERE 字段 LIKE ? LIMIT ?,?;

SELECT \* FROM 表 WHERE ${字段名称} LIKE #{内容} LIMIT #{起始值},#{偏移值};

范例：实现分页查询

<**select id="queryNewsPage" parameterType="java.util.Map" resultMap="NewsResultMap"**>  
 select *\** from news where ${column} like #{keyword} limit #{start},#{linesize}  
</**select**>  
  
<**select id="queryNewsPageRows" parameterType="java.util.Map" resultMap="NewsResultMap"**>  
 select *count*(1) from news where ${column} like #{keyword}  
</**select**>

String column = **"title"**;  
String keyword = **"%幂%"**;  
*//页码***int** pageNo = 1;  
*//每页显示条数***int** lineSize = 3;  
*//查询起始页***int** start = (pageNo - 1) \* lineSize;  
Map<String, Object> param = **new** HashMap<>(4);  
param.put(**"column"**, column);  
param.put(**"keyword"**, keyword);  
param.put(**"start"**, start);  
param.put(**"linesize"**, lineSize);  
List<News> list = MyBatisSessionFactory.*getSession*().selectList(**"cn.mldn.vo.mapper.NewsMapper.queryNewsPage"**, param);  
System.***out***.println(**"总记录数："**+MyBatisSessionFactory.*getSession*().selectOne(**"cn.mldn.vo.mapper.NewsMapper.queryNewsPageRows"**,param));  
MyBatisSessionFactory.*getSession*().commit();  
MyBatisSessionFactory.*getSession*().close();  
  
Iterator<News> itre = list.iterator();  
**while** (itre.hasNext()) {  
 System.***out***.println(itre.next());  
}  
System.***out***.println(list.size());

**注意若parameterType被写成parameterMap将报异常**：

### Error querying database. Cause: org.apache.ibatis.builder.IncompleteElementException: Could not find parameter map java.util.Map

### Cause: org.apache.ibatis.builder.IncompleteElementException: Could not find parameter map java.util.Map

at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:30)

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:150)

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:141)

at cn.mldn.test.TestNewsInsertDemo.main(TestNewsInsertDemo.java:37)

Caused by: org.apache.ibatis.builder.IncompleteElementException: Could not find parameter map java.util.Map

at org.apache.ibatis.builder.MapperBuilderAssistant.getStatementParameterMap(MapperBuilderAssistant.java:320)

at org.apache.ibatis.builder.MapperBuilderAssistant.addMappedStatement(MapperBuilderAssistant.java:296)

at org.apache.ibatis.builder.xml.XMLStatementBuilder.parseStatementNode(XMLStatementBuilder.java:109)

at org.apache.ibatis.session.Configuration.buildAllStatements(Configuration.java:788)

at org.apache.ibatis.session.Configuration.getMappedStatement(Configuration.java:719)

at org.apache.ibatis.session.Configuration.getMappedStatement(Configuration.java:714)

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectList(DefaultSqlSession.java:147)

... 2 more

Caused by: java.lang.IllegalArgumentException: Parameter Maps collection does not contain value for java.util.Map

at org.apache.ibatis.session.Configuration$StrictMap.get(Configuration.java:888)

at org.apache.ibatis.session.Configuration.getParameterMap(Configuration.java:660)

at org.apache.ibatis.builder.MapperBuilderAssistant.getStatementParameterMap(MapperBuilderAssistant.java:318)

... 8 more

Process finished with exit code 1

## Mybatis缓存策略

Mybatis里面提供有两种缓存：一级缓存（sqlsession级）和二级缓存（sqlsessionfactory级）。

### 4.1 一级缓存

一级缓存（SqlSession级）永远存在，，下面观察一级缓存：

* 前提是同一个SqlSession对象操作。

News voa = MyBatisSessionFactory.getSession().selectOne(

"cn.mldn.mapping.NewsNS.findById", 2);

System.out.println(voa);

System.out.println("=====================================");

News vob = MyBatisSessionFactory.getSession().selectOne(

"cn.mldn.mapping.NewsNS.findById", 2);

System.out.println(vob);

默认情况下一级缓存永远存在，将对同一个对象进行缓存以方便重复查询。

对对象进行修改，提交一次事务：

News voa = MyBatisSessionFactory.getSession().selectOne(

"cn.mldn.mapping.NewsNS.findById", 2);

System.out.println(voa);

MyBatisSessionFactory.getSession().commit();

System.out.println("=====================================");

News vob = MyBatisSessionFactory.getSession().selectOne(

"cn.mldn.mapping.NewsNS.findById", 2);

System.out.println(vob);

在进行一级缓存的处理过程中，如果之前的数据发生了事务的提交处理，mybatis会认为是该数据发生了变更，会自动将缓存中的数据清除。如果继续查询相同的数据，会发现重复出现了查询指令。

如果需要可以使用SqlSession接口中的方法手工处理：

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 描述 |
| 1 | void clearCache(); | 普通 | 清除缓存 |

范例：手动清除缓存

News voa = MyBatisSessionFactory.getSession().selectOne(

"cn.mldn.mapping.NewsNS.findById", 2);

System.out.println(voa);

MyBatisSessionFactory.getSession().clearCache();

System.out.println("=====================================");

News vob = MyBatisSessionFactory.getSession().selectOne(

"cn.mldn.mapping.NewsNS.findById", 2);

System.out.println(vob);

### 4.2 二级缓存

一级缓存只是针对于单个SqlSession的控制，二二级缓存可以针对于多个SqlSession进行控制（跨session），很明显二级缓存不是一直存在的，需要手动进行配置。

范例：观察二级缓存的问题

SqlSession sessionA = MyBatisSessionFactory.getSessionFactory()

.openSession();

News voa = sessionA.selectOne("cn.mldn.mapping.NewsNS.findById", 2);

System.out.println(voa);

sessionA.close();

System.out.println("=====================================");

SqlSession sessionB = MyBatisSessionFactory.getSessionFactory()

.openSession();

News vob = sessionB.selectOne("cn.mldn.mapping.NewsNS.findById", 2);

System.out.println(vob);

此时发现，查询统一条数据，进行两次sql执行，如果现在希望不同的session，查询同一对象使用缓存，需要实现二级缓存的配置：

范例：修改mybatis.config.xml文件，启用二级缓存

<?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>

<settings>

<!-- 启用二级缓存的配置 -->

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

</settings>

...

现在表示整个项目中，允许你使用缓存的操作，但是并不意味着你现在可以操作缓存的了，因为缓存不是任何地方任何时候都需要，所以需要在要使用缓存的地方配置。

在映射文件中配置缓存：

<**mapper namespace="cn.mldn.vo.mapper.NewsMapper"**>  
 <**cache**/>

...

**可是要使用缓存需要注意两个事项：**

1. **第一个session必须关闭后才会写入缓存，如果不关闭，则缓存不存在。**
2. **要实现缓存的对象，其类必须实现java.io.Serializable接口。**

现在在映射文件内的所有查询都会二级缓存，某个查询不想使用缓存则需要单独配置：

<**select id="queryNews" resultMap="NewsResultMap" parameterType="int"** **useCache="false"**>  
 select *\** from news where nid=#{id}  
</**select**>

useCache="false"表示不使用缓存。

以上所进行的缓存操作，实际上都属于默认的缓存处理形式，她所使用的缓存处理算法叫：LRU（最近最少使用算法），当然如果有需要也可以改变缓存算法。

范例：在映射文件中配置缓存项

<**cache** **eviction="FIFO" flushInterval="10000" readOnly="true" size="512"**/>

在以上配置中参数作用如下：

1. eviction="FIFO"：设置缓存策略，有如下几种方式：
2. LRU（默认）：最近最少使用的算法，将不怎么使用的对象清除；
3. FIFO：先进先出，删除缓存队列中较早的对象；
4. SOFT：软引用，依靠java中GC处理；
5. WEAK:弱引用，依靠java中GC处理；
6. flushInterval="10000"：缓存的刷新时间，单位毫秒；
7. readOnly="true"：对于缓存的配置一定不要进行同步（和数据库），即只设置为只读缓存，如果设为false,表示要进行同步更新功能，性能差（数据更改就更新）；
8. size="512"：缓存开辟的空间，默认是1024字节。

## 动态SQL

### 5.1 if语句

判断值是否为空，拼接sql;

<if></if>

### 5.2 choose语句

If语句只能实现简单的判断操作，但是choose语句可以实现多条件的判断处理操作。if标签是与(and)的关系，而 choose 是或(or)的关系。choose标签是按顺序判断其内部when标签中的test条件出否成立，如果有一个成立，则 choose 结束。当 choose 中所有 when 的条件都不满则时，则执行 otherwise 中的sql。类似于Java 的 switch 语句，choose 为 switch，when 为 case，otherwise 则为 default。例如：

<select id="dynamicChooseTest" parameterType="Blog" resultType="Blog">

select \* from t\_blog where 1 = 1

<choose>

<when test="title != null">

and title = #{title}

</when>

<when test="content != null">

and content = #{content}

</when>

<otherwise>

and owner = "owner1"

</otherwise>

</choose>

</select>

“where”标签会知道如果它包含的标签中有返回值的话，它就插入一个‘where’，没有则不出现。此外，如果标签返回的内容是以 AND 或OR 开头的，则它会剔除掉。

<select id="dynamicWhereTest" parameterType="Blog" resultType="Blog">

select \* from t\_blog

<where>

<if test="title != null">

title = #{title}

</if>

<if test="content != null">

and content = #{content}

</if>

<if test="owner != null">

and owner = #{owner}

</if>

</where>

</select>

Where标签也可以和choose一起使用。

### 5.3 foreach语句 (在实现 mybatis in 语句查询时特别有用)

对于多个ID的数据往往会使用set集合来传递，但是foreach标签只支持java.util.List集合或者是数组。如果要传递Set集合那么将要使用Conllections集合的toArrays()来传递.

范例：实现数据的批量查询

1、数组方式实现，其参数名称为array,java.util.Arrays也可以替换为Object[]、Integer[]等。

<**select id="queryNews" resultMap="NewsResultMap" parameterType="java.util.Arrays" useCache="false"**>  
 select *\** from news  
 <**where**>  
 nid in  
 <**foreach collection="array" open="(" close=")" separator="," item="ele"**>  
 #{ele}  
 </**foreach**>  
 </**where**>  
</**select**>

SqlSession sessionA = MyBatisSessionFactory.*getSessionFactory*()  
 .openSession();  
Set<Integer> set = **new** HashSet<>();  
set.add(6);  
set.add(7);  
set.add(8);  
set.add(9);  
sessionA.selectList(**"cn.mldn.vo.mapper.NewsMapper.queryNews"**, set.toArray());  
sessionA.close();

2、一List方式实现，参数名称为list:

<**select id="queryNews" resultMap="NewsResultMap" parameterType="java.util.List" useCache="false"**>  
 select *\** from news  
 <**where**>  
 nid in  
 <**foreach collection="list" open="(" close=")" separator="," item="ele"**>  
 #{ele}  
 </**foreach**>  
 </**where**>  
</**select**>

SqlSession sessionA = MyBatisSessionFactory.*getSessionFactory*()  
 .openSession();  
List<Integer> list = **new** ArrayList<>();  
list.add(6);  
list.add(7);  
list.add(8);  
list.add(9);  
sessionA.selectList(**"cn.mldn.vo.mapper.NewsMapper.queryNews"**, list);  
sessionA.close();

### 5.4 Set语句

当 update 语句中没有使用 if 标签时，如果有一个参数为 null，都会导致错误。

当在 update 语句中使用if标签时，如果前面的if没有执行，则或导致逗号多余错误。使用set标签可以将动态的配置 SET 关键字，并剔除追加到条件末尾的任何不相关的逗号。使用 if+set 标签修改后，如果某项为 null 则不进行更新，而是保持数据库原值。如下示例：

<!-- if/set(判断参数) - 将实体 User类不为空的属性更新 -->

<update id="updateUser\_if\_set" parameterType="com.pojo.User">

UPDATE user

<set>

<if test="username!= null and username != '' ">

username = #{username},

</if>

<if test="sex!= null and sex!= '' ">

sex = #{sex},

</if>

<if test="birthday != null ">

birthday = #{birthday},

</if>

</set>

WHERE user\_id = #{userid};

</update>

再看看下面的一个示例：

<update id="dynamicSetTest" parameterType="Blog">

update t\_blog

<set>

<if test="title != null">

title = #{title},

</if>

<if test="content != null">

content = #{content},

</if>

<if test="owner != null">

owner = #{owner}

</if>

</set>

where id = #{id}

</update>

set 标签元素主要是用在更新操作的时候，它的主要功能和 where 标签元素其实是差不多的，主要是在包含的语句前输出一个 set，然后如果包含的语句是以逗号结束的话将会把该逗号忽略，如果 set 包含的内容为空的话则会出错。有了 set 元素就可以动态的更新那些修改了的字段。

### 5.5 trim语句

## **trim代替where/set标签**

trim 是更灵活用来去处多余关键字的标签，它可以用来实现 where 和 set 的效果。

<!-- 使用 if/trim 代替 where(判断参数) - 将 User 类不为空的属性作为 where 条件 -->

<select id="getUsertList\_if\_trim" resultMap="resultMap\_User">

SELECT \*

FROM user u

<trim prefix="WHERE" prefixOverrides="AND|OR">

<if test="username !=null ">

u.username LIKE CONCAT(CONCAT('%', #{username, jdbcType=VARCHAR}),'%')

</if>

<if test="sex != null and sex != '' ">

AND u.sex = #{sex, jdbcType=INTEGER}

</if>

<if test="birthday != null ">

AND u.birthday = #{birthday, jdbcType=DATE}

</if>

</trim>

</select>

### **trim 代替 set**

<!-- if/trim代替set(判断参数) - 将 User 类不为空的属性更新 -->

<update id="updateUser\_if\_trim" parameterType="com.yiibai.pojo.User">

UPDATE user

<trim prefix="SET" suffixOverrides=",">

<if test="username != null and username != '' ">

username = #{username},

</if>

<if test="sex != null and sex != '' ">

sex = #{sex},

</if>

<if test="birthday != null ">

birthday = #{birthday},

</if>

</trim>

WHERE user\_id = #{user\_id}

</update>

****trim (对包含的内容加上 prefix,或者 suffix 等，前缀，后缀)****

<select id="dynamicTrimTest" parameterType="Blog" resultType="Blog">

select \* from t\_blog

<trim prefix="where" prefixOverrides="and |or">

<if test="title != null">

title = #{title}

</if>

<if test="content != null">

and content = #{content}

</if>

<if test="owner != null">

or owner = #{owner}

</if>

</trim>

</select>

trim 元素的主要功能是可以在自己包含的内容前加上某些前缀，也可以在其后加上某些后缀，与之对应的属性是 prefix 和 suffix；可以把包含内容的首部某些内容覆盖，即忽略，也可以把尾部的某些内容覆盖，对应的属性是 prefixOverrides 和 suffixOverrides；正因为 trim 有这样的功能，所以我们也可以非常简单的利用 trim 来代替 where 元素的功能。

总结：

条件判断标签是针对JAVABEAN或者集合的，STRING（基本类型）不能用标签

### 5.6 mybatis中的sql片段

定义SQL片段：

<**sql id="queryNewsColumn"**>  
 nid,title,pub\_date  
</**sql**>  
<**sql id="queryNewsWhere"**>  
 <**if test="title!= null"**>  
 and title=#{title},  
 </**if**>  
</**sql**>  
<**select id="queryNews" resultMap="NewsResultMap" parameterType="News" useCache="false"**>  
 select  
 <**include refid="queryNewsColumn"**/>  
 from news  
 <**trim prefix="where" prefixOverrides="and|or" suffixOverrides=","**>  
 <**include refid="queryNewsWhere"**/>  
 </**trim**>  
</**select**>

## Mybatis高级配置

### 6.1 注解配置

在mybatis里面，每一个实体表都应该有一个对应的\*.xml文件，在xml文件编写的时候会感觉比较麻烦：

1. 如果项目中的数据表过多，那么维护的时候就会比较麻烦。

|- 相对于hibernate中的复杂关系，mybatis的维护要方便许多。

1. 每一次都要编写很长的命名空间;

范例：定义INewsDao,此时不要实现子类

**public interface** INewsDAO {  
 @Insert(**"INSERT INTO news(title,pub\_date) VALUES (#{title},#{pubdate})"**)  
 @SelectKey(before = **false**, keyProperty = **"nid"**, resultType = java.lang.Integer.**class**, statement = **"SELECT LAST\_INSERT\_ID()"**)  
 **public boolean** doCreate(News vo) **throws** Exception;  
  
 @Update(**"UPDATE news SET title=#{title},pub\_date=#{pubdate} WHERE nid=#{nid}"**)  
 **public boolean** doUpdate(News vo) **throws** Exception;  
  
 @Delete(**"DELETE FROM news WHERE nid=#{nid}"**)  
 **public boolean** doRemove(Integer id) **throws** Exception;  
  
 @Select(**"SELECT nid,title,pub\_date AS pubdate FROM news WHERE nid=#{nid}"**)  
 **public** News findById(Integer id) **throws** Exception;  
  
 @Select(**"SELECT nid,title,pub\_date AS pubdate FROM news"**)  
 **public** List<News> findAll() **throws** Exception;  
  
 @Select(**"SELECT nid,title,pub\_date AS pubdate FROM news WHERE #{column} LIKE #{keyWord} LIMIT #{start},#{lineSize}"**)  
 **public** List<News> findAllSplit(  
 @Param(**"column"**) String column,  
 @Param(**"keyWord"**) String keyWord,  
 @Param(**"start"**) Integer start,  
 @Param(**"lineSize"**) Integer lineSize) **throws** Exception;  
  
 @Select(**"SELECT COUNT(\*) FROM nid WHERE #{column} LIKE #{keyWord}"**)  
 **public** Integer getAllCount(  
 @Param(**"column"**) String column,  
 @Param(**"keyWord"**) String keyWord) **throws** Exception;  
}

如果要使用这个接口进行数据操作，则需要修改自定义的mybatisSessionFactory工厂类。

首先对于接口文件的绑定处理操作，可以直接通过SessionFactory接口完成，此接口中定义了可以取得全部配置文件的方法：Configuration getConfiguration()。

这个方法返回的是：”org.apache.ibatis.session.Configuration”类对象，这个方法定义有如下两个方法：

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 描述 |
| 1 | public <T> void addMapper(Class<T> type) | 普通 | 追加一个类型的定义，包含有annotation的接口 |
| 2 | public void addMappers(String packageName) | 普通 | 将指定包下的annotation接口自动加载 |

范例修改MybatisSessionFactory程序类

*/\*\*  
 \* 重新建立新的SqlSessionFactory接口对象  
 \*/***public static void** rebuildSessionFactory() {  
 **try** {  
 *// 读取配置文件  
 inputStream* = Resources.*getResourceAsStream*(***CONFIG\_FILE***);  
 *// 创建SqlSessionFactory接口对象  
 sqlSessionFactory* = **new** SqlSessionFactoryBuilder().build(*inputStream*);  
 *//取得指定包下annotation配置的接口文件  
 sqlSessionFactory*.getConfiguration().addMappers(**"cn.mldn.dao"**);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
}

如果想要利用annotation定义的接口实现数据库的操作，则必须利用SqlSession接口所定义的方法取得指定的接口对象。方法：<T> T getMapper(Class<T> var1);

范例：实现数据增加

News vo = **new** News();  
vo.setTitle(**"杨幂，我爱你！"**);  
vo.setPubdate(**new** Date());  
INewsDAO newsDAO = MyBatisSessionFactory.*getSession*().getMapper(INewsDAO.**class**);  
newsDAO.doCreate(vo);  
MyBatisSessionFactory.*getSession*().commit();  
MyBatisSessionFactory.*getSession*().close();  
System.***out***.println(**"新增记录的ID："** + vo.getNid());

测试结果：

2018-05-10 22:46:25,643 DEBUG [cn.mldn.dao.INewsDAO.doCreate] - ==> Preparing: INSERT INTO news(title,pub\_date) VALUES (?,?)

2018-05-10 22:46:25,728 DEBUG [cn.mldn.dao.INewsDAO.doCreate] - ==> Parameters: 杨幂，我爱你！(String), 2018-05-10 22:46:24.222(Timestamp)

2018-05-10 22:46:25,733 DEBUG [cn.mldn.dao.INewsDAO.doCreate] - <== Updates: 1

2018-05-10 22:46:25,736 DEBUG [cn.mldn.dao.INewsDAO.doCreate!selectKey] - ==> Preparing: SELECT LAST\_INSERT\_ID()

2018-05-10 22:46:25,736 DEBUG [cn.mldn.dao.INewsDAO.doCreate!selectKey] - ==> Parameters:

2018-05-10 22:46:25,768 TRACE [cn.mldn.dao.INewsDAO.doCreate!selectKey] - <== Columns: LAST\_INSERT\_ID()

2018-05-10 22:46:25,768 TRACE [cn.mldn.dao.INewsDAO.doCreate!selectKey] - <== Row: 13

2018-05-10 22:46:25,771 DEBUG [cn.mldn.dao.INewsDAO.doCreate!selectKey] - <== Total: 1

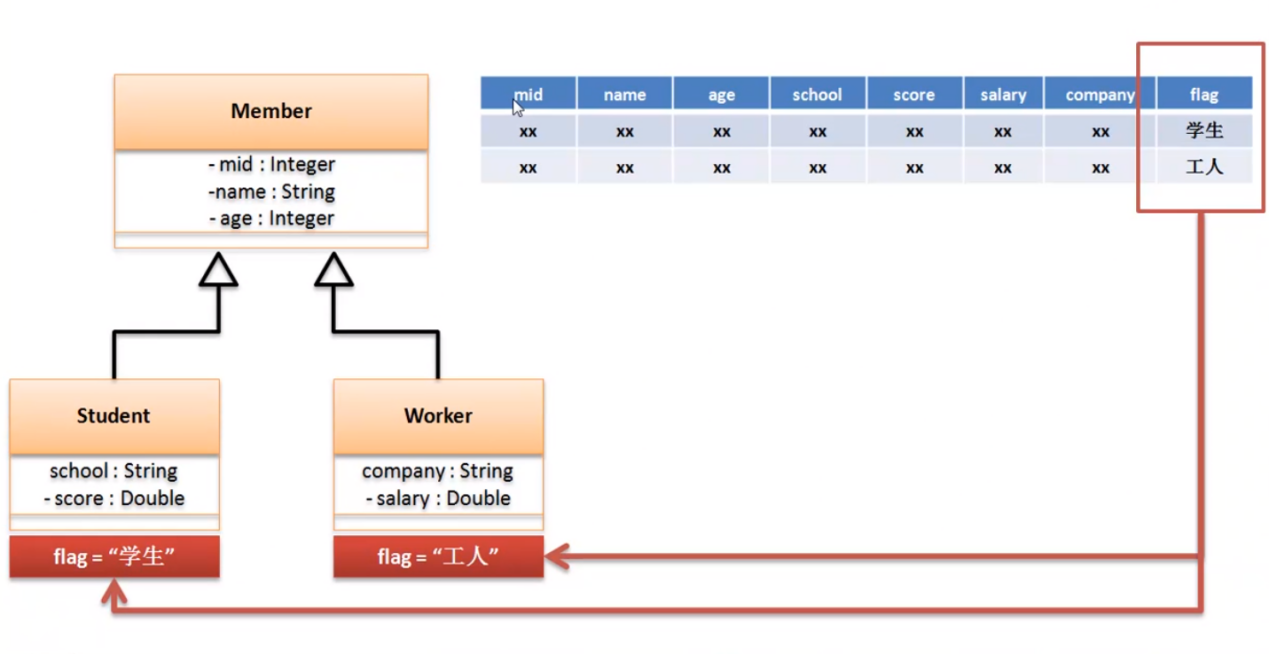
新增记录的ID：13

Process finished with exit code 0

整个操作过程中，用户不在关注于具体的Dao子类该如何实现，因为这里面只是模拟了一个方法的调用，真正操作的还是上面注解语句的执行。

### 6.2 识别器或**鉴别器（discriminator）**

所谓的识别器，指的是在一张数据表中可以描述多种实体关系，这样可以在程序中，可以用不同实体模式，来描述同一张数据表的数据。比如member为公共信息，学生则没有工人信息，则相应属性设为空，反之工人也是如此。



在mybatis开发框架中支持有这种识别器的应用，要想实现这一功能，我们必须以手工的模式进行识别字段内容的设置。

1. 准备数据库脚本：

CREATE TABLE member (

mid VARCHAR(50) ,

name VARCHAR(50) ,

age INT ,

**school VARCHAR(50) ,**

**score DOUBLE ,**

**company VARCHAR(50) ,**

**salary DOUBLE ,**

**flag VARCHAR(50) ,**

CONSTRAINT pk\_mid PRIMARY KEY(mid)

) ;

此时最大的关系体现在类的继承关系上，应该有父子类的关系。

1. 定义member类：

**public class** Member **implements** Serializable {  
 **private** String **mid** ;  
 **private** String **name** ;  
 **private** Integer **age** ;  
 **private** String **flag** ;  
 **public** String getFlag() {  
 **return flag**;  
 }  
 **public void** setFlag(String flag) {  
 **this**.**flag** = flag;  
 }  
 **public** String getMid() {  
 **return mid**;  
 }  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
 **public** String getName() {  
 **return name**;  
 }  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
 **public** Integer getAge() {  
 **return age**;  
 }  
 **public void** setAge(Integer age) {  
 **this**.**age** = age;  
 }  
}

随后分别定义Student和Worker子类：

**public class** Student **extends** Member {  
 **private** String **school** ;  
 **private** Double **score** ;  
 **public** String getSchool() {  
 **return school**;  
 }  
 **public void** setSchool(String school) {  
 **this**.**school** = school;  
 }  
 **public** Double getScore() {  
 **return score**;  
 }  
 **public void** setScore(Double score) {  
 **this**.**score** = score;  
 }  
 @Override  
 **public** String toString() {  
 **return "Student [school="** + **school** + **", score="** + **score** + **"]"**;  
 }  
}

**public class** Worker **extends** Member {  
 **private** String **company** ;  
 **private** Double **salary** ;  
 **public** String getCompany() {  
 **return company**;  
 }  
 **public void** setCompany(String company) {  
 **this**.**company** = company;  
 }  
 **public** Double getSalary() {  
 **return salary**;  
 }  
 **public void** setSalary(Double salary) {  
 **this**.**salary** = salary;  
 }  
}

1. 在mybatis.config.xml中定义类的别名：

<typeAlias type="cn.mldn.vo.Member" alias="Member"/>

<typeAlias type="cn.mldn.vo.Student" alias="Student"/>

<typeAlias type="cn.mldn.vo.Worker" alias="Worker"/>

1. 在Member.xml文件中进行识别器的维护操作：

<mapper namespace="cn.mldn.mapping.MemberNS">

<!-- resultMap是在查询返回结果的时候才会起作用，那么也就意味着数据的更新跟此配置无关 -->

<resultMap type="Member" id="MemberResultMap">

<id property="mid" column="mid" />

<result property="name" column="name" />

<result property="age" column="age" />

<discriminator javaType="String" column="flag">

<case value="学生" resultType="Student">

<result property="score" column="score" />

<result property="school" column="school" />

</case>

<case value="工人" resultType="Worker">

<result property="salary" column="salary" />

<result property="company" column="company" />

</case>

</discriminator>

</resultMap>

<insert id="doCreateStudent" parameterType="Student">

INSERT INTO member(mid,name,age,flag,school,score) VALUES (#{mid},#{name},#{age},'学生',#{school},#{score}) ;

</insert>

<insert id="doCreateWorker" parameterType="Worker">

INSERT INTO member(mid,name,age,flag,company,salary) VALUES (#{mid},#{name},#{age},'工人',#{company},#{salary}) ;

</insert>

<select id="findByStudentId" parameterType="String" resultMap="MemberResultMap">

SELECT mid,name,age,flag,school,score FROM member WHERE mid=#{pid} AND flag='学生' ;

</select>

<select id="findByWorkerId" parameterType="String" resultMap="MemberResultMap">

SELECT mid,name,age,flag,company,salary FROM member WHERE mid=#{pid} AND flag='工人' ;

</select>

</mapper>

1. 实现学生数据的增加

public static void main(String[] args) {

Student stu = new Student();

stu.setMid("mldn-stu");

stu.setName("瞌睡田");

stu.setAge(18);

stu.setSchool("MLDN");

stu.setScore(80.0);

System.out.println(MyBatisSessionFactory.getSession().insert(

"cn.mldn.mapping.MemberNS.doCreateStudent", stu));

MyBatisSessionFactory.getSession().commit();

MyBatisSessionFactory.close();

}

1. 增加工人数据

public static void main(String[] args) {

Worker wkr = new Worker();

wkr.setMid("mldn-wkr");

wkr.setName("万千总");

wkr.setAge(18);

wkr.setCompany("西伯利亚难民营");

wkr.setSalary(3.00);

System.out.println(MyBatisSessionFactory.getSession().insert(

"cn.mldn.mapping.MemberNS.doCreateWorker", wkr));

MyBatisSessionFactory.getSession().commit();

MyBatisSessionFactory.close();

}

Mybatis中的识别器，需要手工处理，而关键性的问题在于数据的查询操作上。

1. 取得学生信息

public static void main(String[] args) {

Student stu = MyBatisSessionFactory.getSession().selectOne(

"cn.mldn.mapping.MemberNS.findByStudentId", "mldn-stu");

System.out.println(stu);

System.out.println(stu.getClass());

MyBatisSessionFactory.close();

}

Mybatis识别器只能够针对于查询结果转换到指定的VO类对象做一个区分，但是进行更新处理操作的时候，识别器的字段只能够由用户自己来维护。

## 查询关联操作

### 7.1 一对一映射

一对一的概念，实际上是有些鸡肋的,一对一实际上是一条完整的数据记录。

1. 定义数据库脚本：

CREATE TABLE member\_login (

mid VARCHAR(50) ,

password VARCHAR(50) ,

CONSTRAINT pk\_mid PRIMARY KEY(mid)

) ;

CREATE TABLE member\_details (

mid VARCHAR(50) ,

name VARCHAR(50) ,

age INT ,

CONSTRAINT pk\_mid2 PRIMARY KEY(mid) ,

CONSTRAINT fk\_mid FOREIGN KEY(mid) REFERENCES member\_login(mid) ON DELETE CASCADE

) ;

1. 定义MemberLogin.java类：

public class MemberLogin implements Serializable {

private String mid ;

private String password ;

private MemberDetails details ;

public MemberDetails getDetails() {

return details;

}

public void setDetails(MemberDetails details) {

this.details = details;

}

public String getMid() {

return mid;

}

public void setMid(String mid) {

this.mid = mid;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

}

1. 定义MemberDetails类：

public class MemberDetails implements Serializable {

private String mid ;

private String name ;

private Integer age ;

private MemberLogin login ;

public String getMid() {

return mid;

}

public void setMid(String mid) {

this.mid = mid;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

public MemberLogin getLogin() {

return login;

}

public void setLogin(MemberLogin login) {

this.login = login;

}

}

1. 修改在mybatis配置文件中，设置操作类别名：

<typeAliases>

<typeAlias type="cn.mldn.vo.MemberLogin" alias="MemberLogin"/>

<typeAlias type="cn.mldn.vo.MemberDetails" alias="MemberDetails"/>

</typeAliases>

1. 准备两个操作类的映射文件：

<mappers> <!-- 映射文件信息 -->

<mapper resource="cn/mldn/vo/mapping/MemberLogin.xml" />

<mapper resource="cn/mldn/vo/mapping/MemberDetails.xml" />

</mappers>

定义MemberLogin.xml文件：

*<?***xml version="1.0" encoding="UTF-8"***?>***<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"   
"http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>****<!-- 设置命名空间，可以与不同表的同类型操作进行区分，使用时以“空间名称.id”的方式调用 -->*<**mapper namespace="cn.mldn.mapping.MemberLoginNS"**>  
 *<!-- resultMap是在查询返回结果的时候才会起作用，那么也就意味着数据的更新跟此配置无关 -->* <**resultMap type="MemberLogin" id="MemberLoginResultMap"**>  
 <**id property="mid" column="mid"** />  
 <**result property="password" column="password"** />  
 *<!-- 这个级联操作发生在数据查询的时候进行，表示查询返回的是MemberDetailsResultMap时，执行级联 -->*

*<!--MemberDetails为类的别名-->* <**association property="details" javaType="MemberDetails" column="mid"  
 select="cn.mldn.mapping.MemberDetailsNS.findById"**/>  
 </**resultMap**>  
 <**select id="findById" parameterType="String" resultMap="MemberLoginResultMap"**>  
 SELECT mid,password FROM member\_login WHERE mid=#{pmid} ;  
 </**select**>  
 <**insert id="doCreate" parameterType="MemberLogin"**>  
 INSERT INTO member\_login(mid,password) VALUES (#{mid},#{password}) ;  
 </**insert**>  
</**mapper**>

定义MemberDetails.xml文件：

*<?***xml version="1.0" encoding="UTF-8"***?>***<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"   
"http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>****<!-- 设置命名空间，可以与不同表的同类型操作进行区分，使用时以“空间名称.id”的方式调用 -->*<**mapper namespace="cn.mldn.mapping.MemberDetailsNS"**>  
 *<!-- resultMap是在查询返回结果的时候才会起作用，那么也就意味着数据的更新跟此配置无关 -->* <**resultMap type="MemberDetails" id="MemberDetailsResultMap"**>  
 <**id property="mid" column="mid"** />   
 <**result property="name" column="name"** />  
 <**result property="age" column="age"** />  
 </**resultMap**>  
 <**select id="findById" parameterType="String" resultMap="****MemberDetailsResultMap"**>  
 SELECT mid,name,age FROM member\_details WHERE mid=#{pmid} ;  
 </**select**>  
 <**insert id="doCreate" parameterType="MemberDetails"**>  
 INSERT INTO member\_details(mid,name,age) VALUES (#{mid},#{name},#{age}) ;  
 </**insert**>  
</**mapper**>

1. 实现数据增加：

MemberLogin login = new MemberLogin() ;

login.setMid("mldn - hello");

login.setPassword("www.mldn.cn");

MemberDetails details = new MemberDetails() ;

details.setMid("1");

details.setName("杨幂");

details.setAge(24);

System.out.println(MyBatisSessionFactory.getSession().insert("cn.mldn.mapping.MemberLoginNS.doCreate", login));

System.out.println(MyBatisSessionFactory.getSession().insert("cn.mldn.mapping.MemberDetailsNS.doCreate", details));

MyBatisSessionFactory.getSession().commit();

MyBatisSessionFactory.close();

}

1. 数据查询：

public static void main(String[] args) {

MemberLogin login = MyBatisSessionFactory.getSession().selectOne(

"cn.mldn.mapping.MemberLoginNS.findById", "mldn - hello");

// System.out.println(login.getDetails());

MyBatisSessionFactory.close();

}

在执行查询的时候发现，执行了两次查询，包括级联关系查询。原因是设置了“select="cn.mldn.mapping.MemberDetailsNS.findById”属性，如果取消此属性，不会则不会进行级联查询。所有级联关系只会发生在查询中，更新不会有关系。

### 7.2 一对多数据关联

一对多的数据映射是mybatis本身默认支持的，只需配置实现即可。关系只能在返回的结果上配置。

1. 定义数据库脚本
2. 定义Type.java类：

定义SubType.java类：

1. 设置别名，并配置映射路径：

Mybatis关联关系只有在查询的时候关注，对于更新处理的时候只需要先添加type、在分别添加多个子Type即可。

1. 定义映射文件

4.1 定义Type.Xml文件:

4.2 定义SubType.xml：

1. 增加type数据
2. 增加子分类数据：
3. 实现type查询

此时没有任何的SubType数据返回，也就是说没有发生任何的级联配置关系。如果此时有级联配置：

这个时候发现配置了级联操作后，会查询两张数据表，很明显在大数据量下，这样的级联操作不可取，存在严重的“1+N”次查询。

1. 实现子分类查询

想要实现这些关系的自动匹配，必须有个前提，查询返回参数配置的是ResultMap而不是ResultType。