**Mybatis**

## 一 对原生态JDBC问题的总结

1. **数据库连接**，使用时就创建，不使用立即释放，对数据库进行频繁连接开启和关闭，造成数据库资源浪费，影响数据库性能。解决方案：使用数据库连接池管理数据库连接。

**2、将sql语句硬编码到java代码中**，如果sql 语句修改，需要重新编译java代码，不利于系统维护。解决方案：将sql语句配置在xml配置文件中，即使sql变化，不需要对java代码进行重新编译。

**3、向preparedStatement中设置参**数，对占位符号位置和设置参数值，硬编码在java代码中，不利于系统维护。解决方案：将sql语句及占位符号和参数全部配置在xml中。

**4、从resutSet中遍历结果集数据时**，存在硬编码，将获取表的字段进行硬编码，，不利于系统维护。解决方案：将查询的结果集，自动映射成java对象。

package com.utils;

import java.sql.Connection;import java.sql.DriverManager;import java.sql.PreparedStatement;import java.sql.ResultSet;import java.sql.SQLException;

/\*\*

\* @ClassName: JdbcTest

\* @Description: TODO(原始的JDBC操作数据库)

\* @author warcaft

\* @date 2015-6-27 下午3:31:22

\*

\*/public class JdbcTest {

public static void main(String[] args) {

// 数据库连接

Connection connection = null;

// 预编译的Statement，使用预编译的Statement提高数据库性能

PreparedStatement preparedStatement = null;

// 结果 集

ResultSet resultSet = null;

try {

// 加载数据库驱动

Class.forName("com.mysql.jdbc.Driver");

// 通过驱动管理类获取数据库链接

connection = DriverManager

.getConnection(

"jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8",

"root", "root");

// 定义sql语句 ?表示占位符

String sql = "select \* from t\_user where username = ?";

// 获取预处理statement

preparedStatement = connection.prepareStatement(sql);

// 设置参数，第一个参数为sql语句中参数的序号（从1开始），第二个参数为设置的参数值

preparedStatement.setString(1, "王五");

// 向数据库发出sql执行查询，查询出结果集

resultSet = preparedStatement.executeQuery();

// 遍历查询结果集

while (resultSet.next()) {

System.out.println(resultSet.getString("id") + " "

+ resultSet.getString("username"));

}

} catch (Exception e) {

e.printStackTrace();

} finally {

// 释放资源

if (resultSet != null) {

try {

resultSet.close();

} catch (SQLException e) {

// TODO Auto-generated catch block e.printStackTrace();

}

}

if (preparedStatement != null) {

try {

preparedStatement.close();

} catch (SQLException e) {

// TODO Auto-generated catch block e.printStackTrace();

}

}

if (connection != null) {

try {

connection.close();

} catch (SQLException e) {

// TODO Auto-generated catch block e.printStackTrace();

}

}

}

}

}

## 二：MyBatis框架

MyBatis 本是[apache](http://baike.baidu.com/view/28283.htm)的一个开源项目[iBatis](http://baike.baidu.com/view/628102.htm), 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis，实质上Mybatis对ibatis进行一些改进。 MyBatis是一个优秀的持久层框架，它对jdbc的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建connection、创建statement、手动设置参数、结果集检索等jdbc繁杂的过程代码。Mybatis通过xml或注解的方式将要执行的各种statement（statement、preparedStatemnt、CallableStatement）配置起来，并通过java对象和statement中的sql进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射成java对象并返回。

**MyBatis的#和$有什么区别？**

1.#将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #user\_id#，如果传入的值是111,那么解析成sql时的值为order by "111", 如果传入的值是id，则解析成的sql为order by "id".

2.$将传入的数据直接显示生成在sql中。如：order by $user\_id$，如果传入的值是111,那么解析成sql时的值为order by user\_id, 如果传入的值是id，则解析成的sql为order by id.

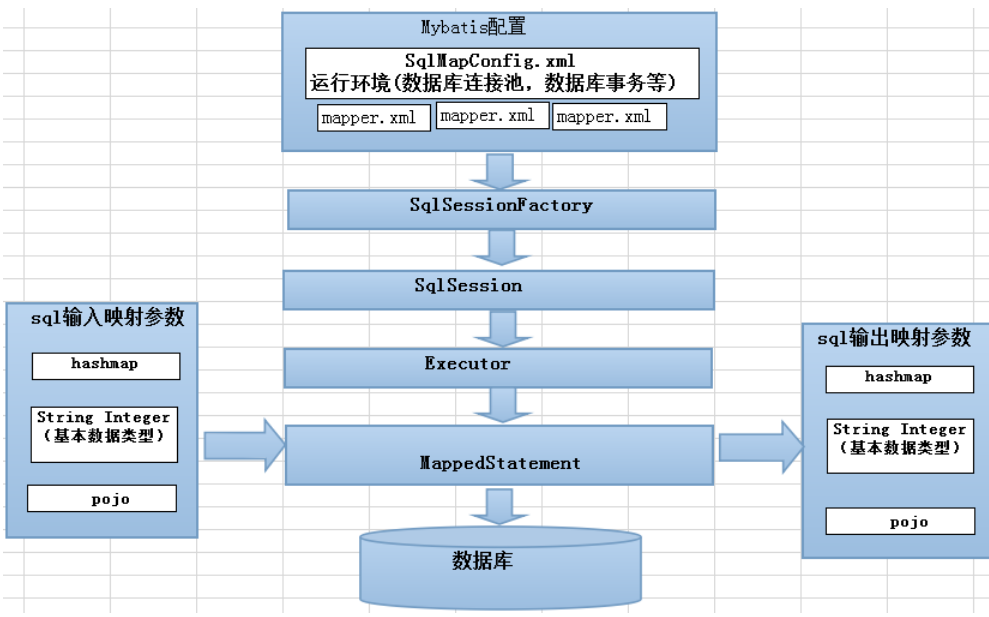
3.#方式能够很大程度防止sql注入。

4.$方式无法防止Sql注入。

5.$方式一般用于传入数据库对象，例如传入表名.

6.一般能用#的就别用$.

## 三、MyBatis架构图



**1、mybatis配置**

**（1）SqlMapConfig.xml，作为mybatis的全局配置文件**，配置了的运行环境等信息。

**（2）mapper.xml文件即sql映射文件**，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

 2、通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂

 3、由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。

 4、mybatis底层自定义了Executor执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器、一个是缓存执行器。

 5、Mapped Statement也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。

1. Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。

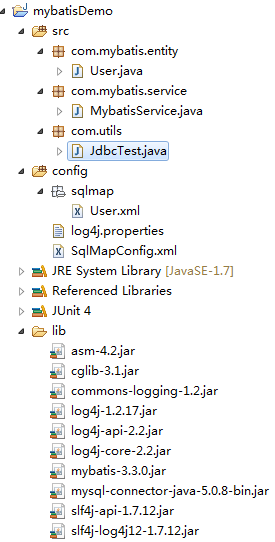
 7、Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

## 四、mybatis入门程序

1. **需求**
2. 根据用户id（主键）查询用户信息
3. 根据用户名称模糊查询用户信息
4. 添加用户
5. 删除用户
6. . 更新用户

**2.环境：java环境：JDK1.7，eclipse,Mysql5.1**

**3.工程目录结构**



**4.从mybatis的jar包结构可知mybatis用的是log4j记录日志**，所以log4j.properties文件内容如下：

# Global logging configuration

#在开发的环境下，日志级别要设置成DEBUG，生产环境设置成info或error

log4j.rootLogger=DEBUG, stdout

# Console output...

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

**5.SqlMapConfig.xml的配置文件内容**

<?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>

<!-- 和spring整合后 environments配置将废除-->

<environments default="development">

<environment id="development">

<!-- 使用jdbc事务管理,事务控制由mybatis管理-->

<transactionManager type="JDBC" />

<!-- 数据库连接池,由mybatis管理-->

<dataSource type="POOLED">

<property name="driver" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8" />

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

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

</dataSource>

</environment>

</environments><!-- 加载映射文件 -->

<mappers>

<mapper resource="sqlmap/User.xml"/>

</mappers></configuration>

**6.实体User.java内容**

package com.mybatis.entity;

import java.util.Date;

/\*\*

\* @ClassName: User

\* @Description: TODO(用户实体)

\* @author warcaft

\* @date 2015-6-27 下午1:56:02

\*

\*/public class User {

// 属性名称和数据库字段名称保持一致

private Integer id;

// 姓名

private String username;

// 性别

private String sex;

// 地址

private String address;

// 生日

private Date birthday;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getSex() {

return sex;

}

public void setSex(String sex) {

this.sex = sex;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

public Date getBirthday() {

return birthday;

}

public void setBirthday(Date birthday) {

this.birthday = birthday;

}

@Override

public String toString() {

return "User [id=" + id + ", username=" + username + ", sex=" + sex

+ ", address=" + address + ", birthday=" + birthday + "]";

}

}

**7.映射文件User.xml的内容**

<?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"><!-- namespace命名空间,作用就是对sql进行分类化的管理,理解为sql隔离

注意:使用mapper代理开发时，namespace有特殊作用

--><mapper namespace="test"><!-- 在映射文件中配置很多sql语句 --><!-- 需求：通过Id查询用户表的记录 --><!-- 通过SELECT执行数据库查询

id:标识映射文件中的sql，称为statement的id;

将sql语句封装在mapperStatement的对象中，所以Id称为Statement的id;

parameterType:指定输入参数的类型,这里指定int型

#{}:表示一个占位符;

#{id}:其中Id表示接收输入的参数，参数名称就是Id,如果输入参数是简单类型，#{}中的参数名可以任意，可以是value或者其它名称;

resultType:指定sql输出结果所映射的java对象类型,select指定resultType表示将单条记录映射成java对象。--><select id="findUserById" parameterType="int" resultType="com.mybatis.entity.User" >

select \* from t\_user where id=#{id}</select><!-- 根据用户名称模糊查询用户信息,可能返回多条数据

resultType:指定的就是单条记录所映射的java类型;

${}:表示拼接sql字符串，将接收到的参数内容不加任何修饰拼接在sql中.

使用${}拼接sql，可能会引起sql注入

${value}:接收输入参数的内容，如果传入的是简单类型,${}中只能使用value

--><select id="findUserByName" parameterType="java.lang.String" resultType="com.mybatis.entity.User" >

select \* from t\_user where username LIKE '%${value}%'</select><!-- 添加用户

parameterType:指定输入的参数类型是pojo(包括用户信息);

#{}中指定pojo的属性名称，接收到pojo对象的属性值 ,mybatis通过OGNL(类似struts2的OGNL)获取对象的属性值--><insert id="insertUser" parameterType="com.mybatis.entity.User" >

<!--

将insert插入的数据的主键返回到User对象中;

select last\_insert\_id():得到刚insert进去记录的主键值，只适用于自增主键;

keyProperty:将查询到的主键值，设置到parameterType指定的对象的那个属性

order:select last\_insert\_id()执行顺序，相对于insert语句来说它的执行顺序。

resultType:指定select last\_insert\_id()的结果类型;

-->

<selectKey keyProperty="id" order="AFTER" resultType="java.lang.Integer">

select last\_insert\_id()

</selectKey>

<!--

使用mysql的uuid()，实现非自增主键的返回。

执行过程:通过uuid()得到主键，将主键设置到user对象的Id的属性中，其次，在insert执行时，从user对象中取出Id属性值;

<selectKey keyProperty="id" order="BEFORE" resultType="java.lang.String">

select uuid()

</selectKey>

insert into t\_user (id,username,birthday,sex,address) values(#{id},#{username},#{birthday},#{sex},#{address})

-->

insert into t\_user (username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address})</insert><!-- 删除用户

根据ID删除用户,需要输入Id值-->

<delete id="deleteUser" parameterType="java.lang.Integer">

delete from t\_user where id=#{id}

</delete><!-- 更新用户

需要传入用户的Id和用户的更新信息

parameterType:指定User对象，包括Id和用户的更新信息,注意：Id是必须存在的

#{id}:从输入的User对象中获取Id的属性值--><update id="updateUser" parameterType="com.mybatis.entity.User">

update t\_user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address}

where id=#{id}</update>

</mapper>

8.测试程序MybatisService.java代码

package com.mybatis.service;

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

import org.apache.ibatis.io.Resources;import org.apache.ibatis.session.SqlSession;import org.apache.ibatis.session.SqlSessionFactory;import org.apache.ibatis.session.SqlSessionFactoryBuilder;import org.junit.Test;

import com.mybatis.entity.User;

/\*\*

\* @ClassName: MybatisService

\* @Description: TODO(mybatis入门程序)

\* @author warcaft

\* @date 2015-6-27 下午4:49:49

\*

\*/public class MybatisService {

// 根据Id查询用户信息，得到一条记录结果 @Test

public void findUserByIdTest() {

// mybatis的配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = null;

SqlSession sqlSession = null;

try {

inputStream = Resources.getResourceAsStream(resource);

// 1.创建会话工场,传入mybatis的配置文件信息

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()

.build(inputStream);

// 2.通过工厂得到SqlSession

sqlSession = sqlSessionFactory.openSession();

// 3.通过sqlSession操作数据库

// 第一个参数：映射文件中的statement的Id,等于namespace + "." + statement的id;

// 第二个参数:指定和映射文件中所匹配的parameterType类型的参数;

// sqlSession.selectOne结果是与映射文件所匹配的resultType类型的对象;

// selectOne：查询一条结果

User user = sqlSession.selectOne("test.findUserById", 1);

System.out.println(user.toString());

} catch (IOException e) {

e.printStackTrace();

} finally {

if (sqlSession != null) {

sqlSession.close();

}

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

// 根据姓名模糊查询用户信息，得到一条或多条记录结果 @Test

public void findUserByNameTest() {

// mybatis的配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = null;

SqlSession sqlSession = null;

try {

inputStream = Resources.getResourceAsStream(resource);

// 1.创建会话工场,传入mybatis的配置文件信息

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()

.build(inputStream);

// 2.通过工厂得到SqlSession

sqlSession = sqlSessionFactory.openSession();

// 3.通过sqlSession操作数据库

// 第一个参数：映射文件中的statement的Id,等于namespace + "." + statement的id;

// 第二个参数:指定和映射文件中所匹配的parameterType类型的参数;

// sqlSession.selectOne结果是与映射文件所匹配的resultType类型的对象;

// list中的user和resultType类型一致

List<User> list = sqlSession.selectList("test.findUserByName", "小");

System.out.println(list);

} catch (IOException e) {

e.printStackTrace();

} finally {

if (sqlSession != null) {

sqlSession.close();

}

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

// 添加用户 @Test

public void insertUserTest() {

// mybatis的配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = null;

SqlSession sqlSession = null;

try {

inputStream = Resources.getResourceAsStream(resource);

// 1.创建会话工场,传入mybatis的配置文件信息

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()

.build(inputStream);

// 2.通过工厂得到SqlSession

sqlSession = sqlSessionFactory.openSession();

//插入用户的对象

User user = new User();

user.setUsername("小红");

user.setBirthday(new Date());

user.setSex("1");

user.setAddress("上海");

// 3.通过sqlSession操作数据库

// 第一个参数：映射文件中的statement的Id,等于namespace + "." + statement的id;

// 第二个参数:指定和映射文件中所匹配的parameterType类型的参数;

// sqlSession.selectOne结果是与映射文件所匹配的resultType类型的对象;

sqlSession.insert("test.insertUser", user);

//执行提交事务 sqlSession.commit();

//项目中经常需要 获取新增的用户的主键 System.out.println(user.getId());

} catch (IOException e) {

e.printStackTrace();

} finally {

if (sqlSession != null) {

sqlSession.close();

}

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

// 根据Id删除用户 @Test

public void deleteUserTest() {

// mybatis的配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = null;

SqlSession sqlSession = null;

try {

inputStream = Resources.getResourceAsStream(resource);

// 1.创建会话工场,传入mybatis的配置文件信息

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()

.build(inputStream);

// 2.通过工厂得到SqlSession

sqlSession = sqlSessionFactory.openSession();

// 3.通过sqlSession操作数据库

// 第一个参数：映射文件中的statement的Id,等于namespace + "." + statement的id;

// 第二个参数:指定和映射文件中所匹配的parameterType类型的参数;

// sqlSession.selectOne结果是与映射文件所匹配的resultType类型的对象;

//传入Id，删除用户

sqlSession.delete("test.deleteUser", 7);

//执行提交事务 sqlSession.commit();

} catch (IOException e) {

e.printStackTrace();

} finally {

if (sqlSession != null) {

sqlSession.close();

}

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

// 根据Id更新用户信息 @Test

public void updateUserTest() {

// mybatis的配置文件

String resource = "SqlMapConfig.xml";

InputStream inputStream = null;

SqlSession sqlSession = null;

try {

inputStream = Resources.getResourceAsStream(resource);

// 1.创建会话工场,传入mybatis的配置文件信息

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder()

.build(inputStream);

// 2.通过工厂得到SqlSession

sqlSession = sqlSessionFactory.openSession();

//更新用户的信息

User user = new User();

user.setId(2);

user.setUsername("小黑");

user.setBirthday(new Date());

user.setSex("2");

user.setAddress("上海");

// 3.通过sqlSession操作数据库

// 第一个参数：映射文件中的statement的Id,等于namespace + "." + statement的id;

// 第二个参数:指定和映射文件中所匹配的parameterType类型的参数;

// sqlSession.selectOne结果是与映射文件所匹配的resultType类型的对象;

//更具Id更新用户

sqlSession.update("test.updateUser", user);

//执行提交事务 sqlSession.commit();

} catch (IOException e) {

e.printStackTrace();

} finally {

if (sqlSession != null) {

sqlSession.close();

}

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

**9.数据库所用sql脚本**

CREATE TABLE t\_user (

id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(30) NOT NULL COMMENT '用户名称',

birthday DATE DEFAULT NULL COMMENT '生日',

sex CHAR(2) DEFAULT NULL COMMENT '性别',

address VARCHAR(256) DEFAULT NULL COMMENT '地址'

);INSERT INTO t\_user (username,birthday,sex,address)

VALUES

('小A','2015-06-27','2','北京'),

('小B','2015-06-27','2','北京'),

('小C','2015-06-27','1','北京'),

('小D','2015-06-27','2','北京');

**10 Mybatis缓存机制**

@一级缓存: 基于PerpetualCache 的 HashMap本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该Session中的所有 Cache 就将清空。

@二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。

@ 对于缓存数据更新机制，当某一个作用域(一级缓存Session/二级缓存Namespaces)的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被clear。

**（1）、一级缓存**

MyBatis 默认开启了一级缓存，一级缓存是在SqlSession 层面进行缓存的。即，同一个SqlSession ，多次调用同一个Mapper和同一个方法的同一个参数，只会进行一次数据库查询，然后把数据缓存到缓冲中，以后直接先从缓存中取出数据，不会直接去查数据库。但是不同的SqlSession对象，因为不用的SqlSession都是相互隔离的，所以相同的Mapper、参数和方法，他还是会再次发送到SQL到数据库去执行，返回结果。

// 获得SqlSession对象

SqlSession sqlSession = factory.openSession();

// 获得dao实体

UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

// 进行两次相同的查询操作

userMapper.selectByPrimaryKey(1);

userMapper.selectByPrimaryKey(1);

// 获得一个新的SqlSession 对象

SqlSession sqlSession1 = factory.openSession();

// 进行相同的查询操作

sqlSession1.getMapper(UserMapper.class).selectByPrimaryKey(1);**可以发现，第一次的两个相同操作，只执行了一次数据库。后来的那个操作又进行了数据库查询。**

**（2）二级缓存**

在userMapper.xml文件中添加如下配置

<mapper namespace="me.gacl.mapping.userMapper">

<!-- 开启二级缓存 -->

<cache/>

　　1. 映射语句文件中的所有select语句将会被缓存。

　　2. 映射语句文件中的所有insert，update和delete语句会刷新缓存。

　　3. 缓存会使用Least Recently Used（LRU，最近最少使用的）算法来收回。

　　4. 缓存会根据指定的时间间隔来刷新。

5. 缓存会存储1024个对象

**cache标签常用属性：**

<cache

eviction="FIFO" <!--回收策略为先进先出-->

flushInterval="60000" <!--自动刷新时间60s-->

size="512" <!--最多缓存512个引用对象-->

readOnly="true"/> <!--只读-->

**收回策略**

LRU——默认，最近最少使用的：移除最长时间不被使用的对象

FIFO——先进先出的：按对象进入缓存的顺序来移除他们

SOFT——软引用：移除基于垃圾回收器状态和软引用规则的对象

WEAK——弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。

## **二级缓存使用redis**

**spring-context-jedis.xml**

<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">

<property name="maxIdle" value="300"/>

<property name="maxTotal" value="60000"/>

<property name="testOnBorrow" value="true"/>

</bean>

<bean id="jedisPool" class="redis.clients.jedis.JedisPool">

<constructor-arg index="0" ref="jedisPoolConfig"/>

<constructor-arg index="1" value="${redis.host}"/>

<constructor-arg index="2" value="${redis.port}" type="int"/>

</bean>

**MyBatis配置文件（SqlMapConfig.xml）打开二级缓存**

<settings>

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

</settings>

**Mapper配置文件添加cache标签**

import org.apache.ibatis.cache.Cache;

public class RedisCache implements Cache

<cache type="com.hiya.ee.RedisCache "/>

**缓存结果继承序列化接口**

public class User implements Serializable

**测试二级缓存**

SqlSessionFactory factory=new SqlSessionFactoryBuilder().build(Test.class.getClassLoader().getResourceAsStream("SqlMapConfig.xml"));

SqlSession session = factory.openSession();

UserMapper mapper = session.getMapper(UserMapper.class);

User user = mapper.findById(1);

System.out.println(user.toString());

session.commit();

session.close();

SqlSession session2 = factory.openSession();

UserMapper mapper2 = session2.getMapper(UserMapper.class);

User user2 = mapper2.findById(1);

System.out.println(user2.toString());

session2.commit();

session2.close();

## 五、MyBatis框架的优点

1. 与JDBC相比，减少了50%以上的代码量。

2. MyBatis是最简单的持久化框架，小巧并且简单易学。

3. MyBatis相当灵活，不会对应用程序或者数据库的现有设计强加任何影响，SQL写在XML里，从程序代码中彻底分离，降低耦合度，便于统一管理和优化，并可重用。

4. 提供XML标签，支持编写动态SQL语句。

5. 提供映射标签，支持对象与[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "https://blog.csdn.net/u014788653/article/details/68489301/_blank)的ORM字段关系映射。

## 六、MyBatis框架的缺点

1. SQL语句的编写工作量较大，尤其是字段多、关联表多时，更是如此，对开发人员编写SQL语句的功底有一定要求。

2. SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

七、MyBatis框架适用场合**：**

MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案。

对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis将是不错的选择。

## 八、Hibernate与MyBatis对比

**1 相同点：**

（1）Hibernate与MyBatis都是通过SessionFactoryBuider由XML配置文件生成SessionFactory，由SessionFactory 生成Session，由Session来开启执行事务和SQL(Structured Query Language，结构化查询语言)语句。

（2）Hibernate和MyBatis都支持JDBC（Java DataBase Connectivity，java数据库连接）和JTA（Java Transaction API，Java事务API（Application Programming Interface，应用程序编程接口））事务处理。注：jdbc和jta的主要作用是增强数据访问能力。

（3）基于ORM（Object Relational Mapping， 对象关系映射）思想解决了entity和数据库的映射问题

**2 不同点：**

（1）sql方面：mybaits通过mapper.xml维护映射结果，程序员手动编写sql相比hibernate自动生成hql（hibernate sql）更加灵活，sql调优更加容易（hibernate因为更好的封装性，开发效率提高的同时，sql语句调优要更费力，当然可以手动修改sql来优化，但是同时也会影响开发效率）；hibernate的hql数据库移植性更好，体现在强壮性。hibernate在级联删除的时候效率低；数据量大， 表多的时候，基于关系操作会变得复杂。

（2）缓存方面：mybatis和hibernate都可以使用第三方缓存，而hibernate相比maybatis有更好的二级缓存机制。

## 九、Mybatis体系架构

**1 概述**

MyBatis并不是一个完整的ORM框架，其官方首页是这么介绍自己

The MyBatis data mapper framework makes it easier to use a relational database with object-oriented applications. MyBatis couples objects with stored procedures or SQL statements using a XML descriptor or annotations. Simplicity is the biggest advantage of the MyBatis data mapper over object relational mapping tools.

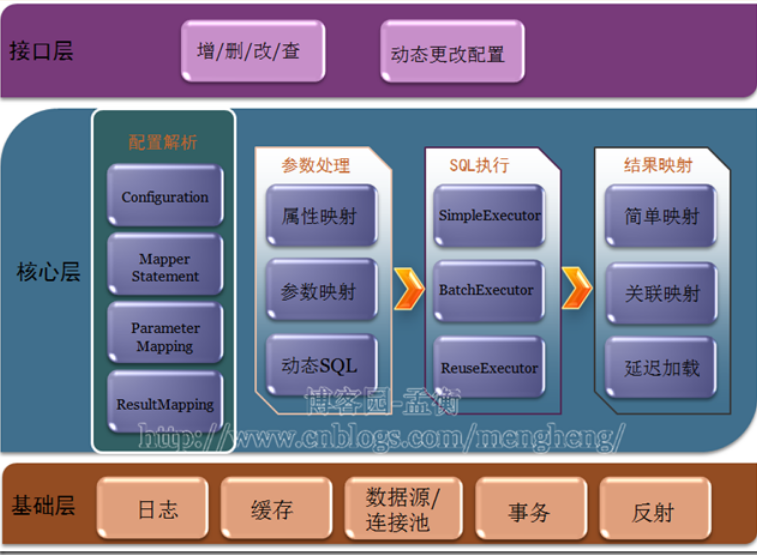
而在其官方文档中介绍“What is MyBaits”中说到

MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results. MyBatis can use simple XML or Annotations for configuration and map primitives, Map interfaces and Java POJOs (Plain Old Java Objects) to database records.

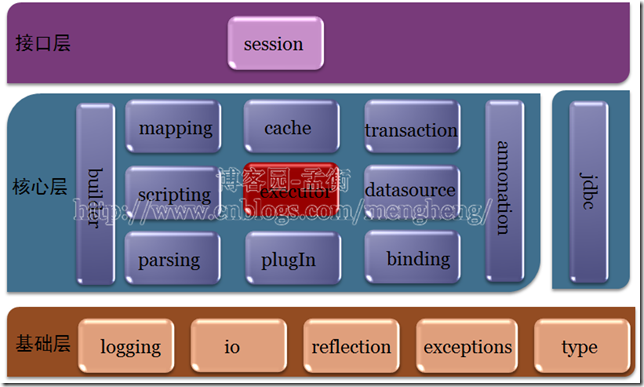
ORM是Object和Relation之间的映射，包括Object->Relation和Relation->Object两方面。Hibernate是个完整的ORM框架，而MyBatis完成的是Relation->Object，也就是其所说的data mapper framework。关于ORM的一些设计思路和细节可以参见Martin Flow《企业应用架构模式》一书中的ORM章节，MyBatis并不刻意于完成ORM(对象映射)的完整概念，而是旨在更简单、更方便地完成数据库操作功能，减轻开发人员的工作量，我想这对于应用系统来说也是最实用的，相信用Hibernate的都受过它的痛苦，而用过MyBatis的都会感觉它很简捷轻松。

**2 整体架构**

下面是从功能流程层次描述MyBatis的整体架构图



而下面是MyBatis源码包对应的架构图

[](https://images0.cnblogs.com/blog/571766/201405/202018523246078.png)

下面以“功能流程角度的架构图”来简要地分析下各层的架构，在后面系列文章中将有专题来深入解析MyBatis重要的功能点。

**3 接口层**

我们知道，在不考虑与Spring集成的情况下,使用MyBatis执行数据库操作的代码如下

String resource = "org/mybatis/example/mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

try {

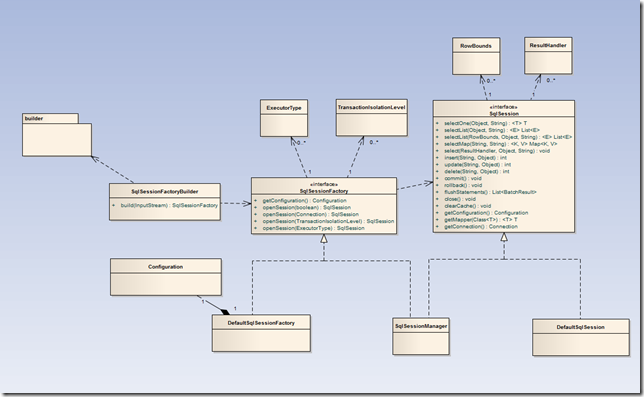
Blog blog = session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);

} finally {

session.close();

}

SqlSessionFactory、SqlSession这是MyBatis接口层的核心类，尤其是SqlSession，是实现所有数据库操作的API，这几个类都是org.apache.ibatis.session包下的，这个包的主体类结构图如下

[](https://images0.cnblogs.com/blog/571766/201405/202018597153868.png)

Configuration是MyBatis中相当重要的一个类，可以这么说，如果理解了其中的所有参数的意义，不仅清楚地知道MyBatis提供的所有配置项，还理解了MyBatis的内部核心运行原理，当然要真正理解这些参数的意义及实现，还需要阅读完完整的MyBatis框架之后才能做到。

由上图可以看到，Configuration对象与DefaultSqlSessionFactory是1:1的关联关系，这也就意味着在一个DefaultSqlSessionFactory衍生出来的所有SqlSession作用域里，Configuration对象是全局唯一的。同时SqlSessionFactory提供了getConfiguration()接口来公开Configuration对象，因此开发者除了配置文件之外，还可以在程序里动态更改Configuration的属性项以达到动态调整的目的，但此时不仅要考虑到执行完reset，同时还要考虑在修改过程中会可能影响到其他SqlSession的执行。

**4 核心层**

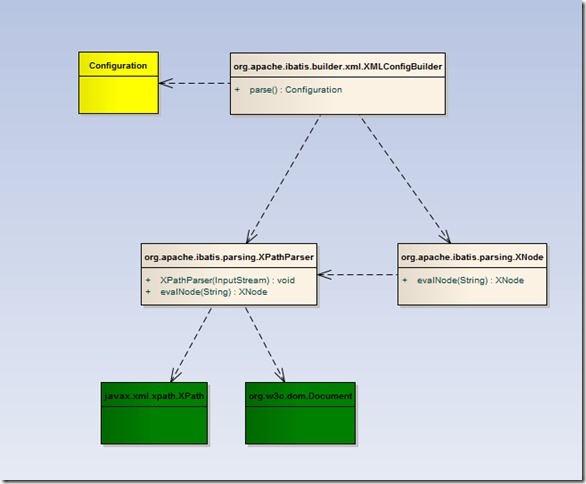
**（1）配置解析**

在应用启动的时候，MyBatis解析两种配置文件

SqlMapConfig.xml

SqlMap.xml

SqlMapConfig.xml是在XMLConfigBuilder类中完成解析的，其类图关系大致如下

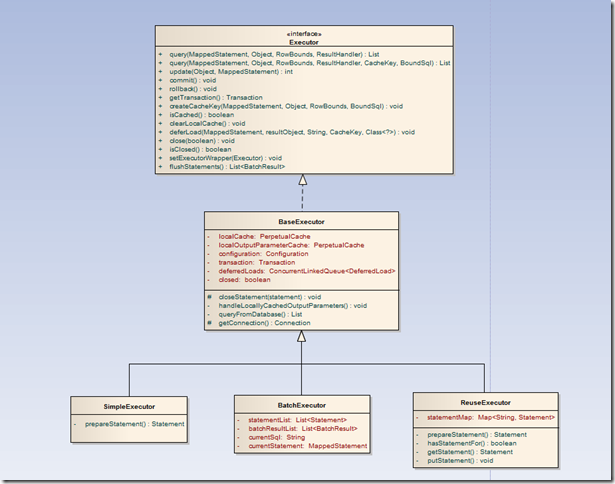
[](https://images0.cnblogs.com/blog/571766/201405/202019034496243.jpg)

我们知道XML有两种解析方式：一是DOM，另一个是SAX，MyBatis使用的是org.wrc.dom——JDK提供的文档对象模型(DOM)接口(SqlMapConfig.xml并不大，所以DOM方式并没有什么效率损耗，JDK也提供了SAX模型接口org.xml.sax，这两个都是JAXP的组件API)，以及JDK官方提供的javax.xml.xpath.XPath来作为XML路径寻找组件。

SqlMap.xml是在XMLMapperBuilder中解析完成的，其中把对Statement的解析(即SqlMap.xml中SELECT|INSERT|UPDATE|DELETE定义部分)委托给XMLStatementBuilder来完成。SqlMap.xml的解析比较复杂的，涉及到PreparedMapping、ResultMapping、LanguageDriver、Discriminator、缓存、自动映射等一系列对象的构造，这里暂时略过，后面专题分析。

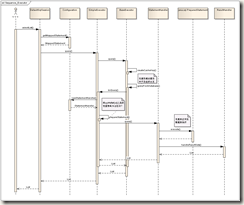
**（2） SQL执行**

MyBatis中Executor是的核心，围绕着它完成了数据库操作的完整过程。下面是Executor的类图

[](https://images0.cnblogs.com/blog/571766/201405/202019091371247.png)

在上图中我列出了Executor中方法的参数，而在其子类中就没有明确写出。从上图中可以看到，Executor主要提供了QUERY|UPDATE(INSERT和DELETE也是使用UPDATE)，从方法定义中可看到，它需要MappedStatement、parameter、resultHandler这几个实例对象，这几个也是SQL执行的主要部分，详细实现在后面专题中再介绍。事务提交/回滚，这委托给Transaction对象来完成。缓存，createCacheKey()/isCached()。延迟加载，deferload()。关闭，close()，主要是事务回滚/关闭。BaseExecutor的属性表明：它内部维护了localCache来localOutputParameterCache来处理缓存，至于这缓存保存的是什么，这后面专题再说。以及线程安全的延迟加载列表deferredLoads、事务对象Transaction。

BatchExecutor的属性已经表明：它内部维护了StatementList批量提交并通过batchResultList保存执行结果。ResueExecutor的属性及方法表明：它内部维护了java.sql.Statement对象缓存，以重用Statement对象(对于支持预编译的数据库而言，在创建PreparedStatement时需要发送一次数据库请求预编译，而重用Statement对象主要是减少了这次预编译的网路开销)。下面以SqlSession.selectList为例，画出SQL执行的时序图(点击下方的图片查看大图，部分分支有所简化)

[](http://images.cnblogs.com/cnblogs_com/mengheng/581225/o_æªå½å.PNG)

**5 基础层**

**（1）logging**

MyBatis使用了自己定义的一套logging接口，根据开发者常使用的日志框架——Log4j、Log4j2、Apache Commons Log、java.util.logging、slf4j、stdout(控制台)——分别提供了适配器。由于各日志框架的Log级别分类法有所不同(比如java.util.logging.Level提供的是All、FINEST、FINER、FINE、CONFIG、INFO、WARNING、SEVERE、OFF这九个级别，与通常的日志框架分类法不太一样)，MyBatis统一提供trace、debug、warn、error四个级别，这基本与主流框架分类法是一致的(相比而言缺少Info，也许MyBatis认为自己的日志要么是debug需要的，要么就至少是warn，没有Info的必要)。在org.apache.ibatis.logging里还有个比较特殊的包jdbc，这不是按字面意义理解把日志通过jdbc记录到数据库里，而是将jdbc操作以开发者配置的日志框架打印出来，这也就是我们在开发阶段常用的跟踪SQL语句、传入参数、影响行数这些重要的调试信息。

**（2）IO**

MyBatis里的IO主要是包含两大功能：提供读取资源文件的API、封装MyBatis自身所需要的ClassLoader和加载顺序。

**（3）reflection**

在MyBatis如参数处理、结果映射这些大量地使用了反射，需要频繁地读取Class元数据、反射调用get/set，因此MyBatis提供了org.apache.ibatis.reflection对常见的反射操作进一步封装，以提供更简洁方便的API。比如我们reflect时总是要处理异常(IllegalAccessException、NoSuchMethodException)，MyBatis统一处理为自定义的RuntimeException，减少代码量。

**（4）exceptions**

在以Spring为代表的开源框架中，对于应用程序中无法进一步处理的异常大都转成RuntimeException来方便调用者操作，另外如频繁遇到的SQLException，JDK约定其是个Exception，从JDK的角度考虑，强制要求开发者捕获SQLException是为了能在catch/finally中关闭数据库连接，而Spring之类的框架为开发者做了资源管理的事情，自然就不需要开发者再烦心SQLException，因此封装转换成RuntimeException。MyBatis的异常体系不复杂，org.apache.ibatis.exceptions下就几个类，主要被使用的是PersistenceException。

**（5）缓存**

缓存是MyBatis里比较重要的部分，有两种缓存：

SESSION或STATEMENT作用域级别的缓存，默认是SESSION，BaseExecutor中根据MappedStatement的Id、SQL、参数值以及rowBound(边界)来构造CacheKey，并使用BaseExccutor中的localCache来维护此缓存。

全局的二级缓存，通过CacheExecutor来实现，其委托TransactionalCacheManager来保存/获取缓存，这个全局二级缓存比较复杂，后面还需要专题分析，至于其缓存的效率以及应用场景也留到那时候再分析。

**（6）数据源/连接池**

MyBatis自身提供了一个简易的数据源/连接池，在org.apache.ibatis.datasource下，后面专题分析。主要实现类是PooledDataSource，包含了最大活动连接数、最大空闲连接数、最长取出时间(避免某个线程过度占用)、连接不够时的等待时间，虽然简单，却也体现了连接池的一般原理。阿里有个“druid”项目，据他们说比proxool、c3p0的效率还要高，可以学习一下。

**（7）事务**

MyBatis对事务的处理相对简单，TransactionIsolationLevel中定义了几种隔离级别，并不支持内嵌事务这样较复杂的场景，同时由于其是持久层的缘故，所以真正在应用开发中会委托Spring来处理事务实现真正的与开发者隔离。分析事务的实现是个入口，借此可以了解不扫JDBC规范方面的事情。后续将对MyBatis各个部分做详细的设计及源代码分析，由于读取和解析SqlMapConfig.xml和SqlMap.xml的逻辑与各个模块的相关性较强，因此将把这部分内容与在各模块组合在一起分析。

## 十、Mapper的动态代理

**1 定义**

采用Mapper动态代理方法只需要编写相应的Mapper接口（相当于Dao接口），那么Mybatis框架根据接口定义创建接口的动态代理对象，代理对象的方法体同Dao接口实现类方法。

Mapper接口开发需要遵循以下规范：

1、Mapper.xml文件中的namespace与mapper接口的全类名相同。

2、Mapper接口方法名和Mapper.xml中定义的每个statement的id相同。

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

4、Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同。

**2当定义好一个Mapper接口(UserDao)里**，我们并不需要去实现这个类，但sqlSession.getMapper()最终会返回一个实现该接口的对象。这个对象是Mybatis利用jdk的动态代理实现的。这里将介绍这个代理对象的生成过程及其方法的实现过程。

**3 Mapper代码对象的生成过程**

DefaultSqlSession.getMapp()方法最终会调用MapperRegistry.getMapper()方法

public <T> T getMapper(Class<T> type, SqlSession sqlSession) {

//这个MapperProxyFactory是调用addMapper方法时加到knownMappers中的，

final MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory<T>) knownMappers.get(type);

if (mapperProxyFactory == null)

//说明这个Mapper接口没有注册

throw new BindingException("Type " + type + " is not known to the MapperRegistry.");

try {

//生成一个MapperProxy对象

return mapperProxyFactory.newInstance(sqlSession);

} catch (Exception e) {

throw new BindingException("Error getting mapper instance. Cause: " + e, e);

}

}

下面是MapperProxyFactory的newInstance方法

public T newInstance(SqlSession sqlSession) {

//创建一个Mapperxy对象，这个方法实现了JDK动态代理中的InvocationHandler接口

final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);

return newInstance(mapperProxy);

}

protected T newInstance(MapperProxy<T> mapperProxy) {

//mapperInterface，说明Mapper接口被代理了，这样子返回的对象就是Mapper接口的子类，方法被调用时会被mapperProxy拦截,也就是执行mapperProxy.invoke()方法

return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapperInterface }, mapperProxy);

}

**4 MapperProxy**

详细分析一下MapperProxy类

public class MapperProxy<T> implements InvocationHandler, Serializable {

private static final long serialVersionUID = -6424540398559729838L;

private final SqlSession sqlSession;

//Mapper接口

private final Class<T> mapperInterface;

//Mapper接口中的每个方法都会生成一个MapperMethod对象, methodCache维护着他们的对应关系

//这个methodCache是在MapperProxyFactory中持有的，MapperProxyFactory又是在Configuration中持有的

//所以每个Mapper接口类对应的MapperProxyFactory和methodCache在整个应用中是共享的，一般只会有一个实例

private final Map<Method, MapperMethod> methodCache;

public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {

this.sqlSession = sqlSession;

this.mapperInterface = mapperInterface;

this.methodCache = methodCache;

}

//这里会拦截Mapper接口(UserDao)的所有方法

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

//如果是Object中定义的方法，直接执行。如toString(),hashCode()等待。

if (Object.class.equals(method.getDeclaringClass())) {

try {

return method.invoke(this, args);

} catch (Throwable t) {

throw ExceptionUtil.unwrapThrowable(t);

}

}

//其他Mapper接口定义的方法交由mapperMethod来执行。

final MapperMethod mapperMethod = cachedMapperMethod(method);

return mapperMethod.execute(sqlSession, args);

}

private MapperMethod cachedMapperMethod(Method method) {

MapperMethod mapperMethod = methodCache.get(method);

if (mapperMethod == null) {

mapperMethod = new MapperMethod(mapperInterface, method, sqlSession.getConfiguration());

methodCache.put(method, mapperMethod);

}

return mapperMethod;

}

}

**5 MapperMethod**

//所有Mapper接口中方法被调用里，都会执行这个方法.这里实际上是调用SqlSession中的相关方法,

public Object execute(SqlSession sqlSession, Object[] args) {

Object result;

//判断这个方法被注解里的Sql类型

if (SqlCommandType.INSERT == command.getType()) {

//执行insert

Object param = method.convertArgsToSqlCommandParam(args);

result = rowCountResult(sqlSession.insert(command.getName(), param));

} else if (SqlCommandType.UPDATE == command.getType()) {

//执行update

Object param = method.convertArgsToSqlCommandParam(args);

result = rowCountResult(sqlSession.update(command.getName(), param));

} else if (SqlCommandType.DELETE == command.getType()) {

/delete

Object param = method.convertArgsToSqlCommandParam(args);

result = rowCountResult(sqlSession.delete(command.getName(), param));

} else if (SqlCommandType.SELECT == command.getType()) {

//select ,查询

if (method.returnsVoid() && method.hasResultHandler()) {

//没有返回值，并且有ResultHandler的情况

executeWithResultHandler(sqlSession, args);

result = null;

} else if (method.returnsMany()) {

//返回一个List

result = executeForMany(sqlSession, args);

} else if (method.returnsMap()) {

//返回一个Map

result = executeForMap(sqlSession, args);

} else {

//返回一个对象

Object param = method.convertArgsToSqlCommandParam(args);

result = sqlSession.selectOne(command.getName(), param);

}

} else {

throw new BindingException("Unknown execution method for: " + command.getName());

}

if (result == null && method.getReturnType().isPrimitive() && !method.returnsVoid()) {

throw new BindingException("Mapper method '" + command.getName()

+ " attempted to return null from a method with a primitive return type (" + method.getReturnType() + ").");

}

return result;

}

**6 小结**

（1）在Mybatis提供的编程接口中，开发人员只需要定义好Mapper接口(如：UserDao)，开发人员无需去实现。Mybatis会利用JDK的动态代理实现 Mapper接口。

（2）在Mybatis中，每个Mapper接口都会对应一个MapperProxyFactory对象实例，这个对应关系在Configuration.mapperRegistry.knownMappers中。

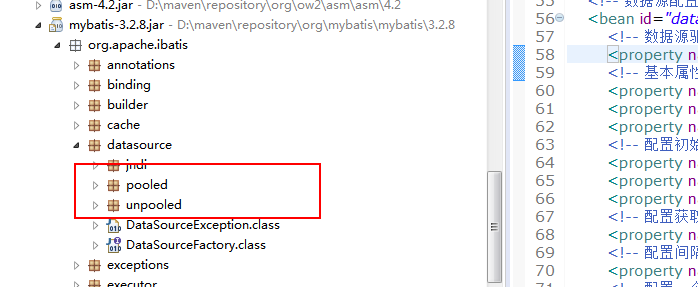
（3）当getMapper()方法被调用时，Mybatis会找到相对应的MapperProxyFactory对象实例，利用这个工厂来创建一个jdk动态代理对象，是这个Mapper接口的实现类,当Mapper定义的方法被调用时，会调用MapperProxy来处理。

（4）MapperProxy会根据方法找到对应的MapperMethod对象来实现这次调用。

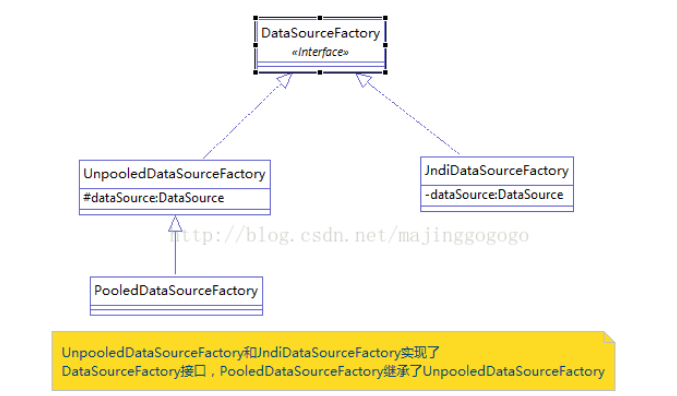
（5）MapperMethod对应会读取方法中的注解，从Configuration中找到相对应的MappedStatement对象，再执行。

## 十一、Mybatis中的数据源与连接池

**1 Mybatis中支持三种形式数据源的配置，分别为：UNPOOLED、POOLED和JNDI**



**2 在Mybatis内部定义了一个接口DataSourceFactory，而支持的三种形式都需要实现这个接口**



**3 MyBatis创建了DataSource实例后，会将其放到Configuration对象内的Environment对象中， 供以后使用.**

DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));

DataSource dataSource = dsFactory.getDataSource();

Environment.Builder environmentBuilder = new Environment.Builder(id)

.transactionFactory(txFactory)

.dataSource(dataSource);

configuration.setEnvironment(environmentBuilder.build());

**4 数据源DataSource对象什么时候创建数据库连接**

当我们需要创建SqlSession对象并需要执行SQL语句时，这时候MyBatis才会去调用dataSource对象来创建java.sql.Connection对象。也就是说，java.sql.Connection对象的创建一直延迟到执行SQL语句的时候。

**5 Mybatis底层执行SQL语句的过程是什么？接口实现类怎么生成的？**

（1）配置spring集成mybatis

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="configLocation" value="classpath:conf/mybatis/config.xml"/>

</bean>

要实现对数据库的操作就要有sqlSession,而sqlSession就是有sqlSessionFactory创建的。

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="basePackage" value="com.lzz.aspp.\*\*.dao,com.lzz.lsp.\*\*.dao" />

</bean>

这段配置主要是配置映射文件的路径，这样做的好处就是不用写Dao的实现了，简单的说就是接好接口，写好映射文件 会自动映射到方法和sql语句。

（2）底层原理：

mybatis通过JDK的动态代理方式，在启动加载配置文件时，根据配置mapper的xml去生成Dao的实现。session.getMapper()使用了代理，当调用一次此方法，都会产生一个代理class的instance,看看这个代理class的实现.

public class MapperProxy implements InvocationHandler {

...

public static <T> T newMapperProxy(Class<T> mapperInterface, SqlSession sqlSession) {

ClassLoader classLoader = mapperInterface.getClassLoader();

Class<?>[] interfaces = new Class[]{mapperInterface};

MapperProxy proxy = new MapperProxy(sqlSession);

return (T) Proxy.newProxyInstance(classLoader, interfaces, proxy);

}

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

if (!OBJECT\_METHODS.contains(method.getName())) {

final Class<?> declaringInterface = findDeclaringInterface(proxy, method);

final MapperMethod mapperMethod = new MapperMethod(declaringInterface, method, sqlSession);

final Object result = mapperMethod.execute(args);

if (result == null && method.getReturnType().isPrimitive()) {

throw new BindingException("Mapper method '" + method.getName() + "' (" + method.getDeclaringClass() + ") attempted to return null from a method with a primitive return type (" + method.getReturnType() + ").");

}

return result;

}

return null;

}

这里是用到了JDK的代理Proxy。 newMapperProxy()可以取得实现interfaces 的class的代理类的实例。

当执行interfaces中的方法的时候，会自动执行invoke()方法，其中public Object invoke(Object proxy, Method method, Object[] args)中 method参数就代表你要执行的方法.

MapperMethod类会使用method方法的methodName 和declaringInterface去取 sqlMapxml 取得对应的sql，也就是拿declaringInterface的类全名加上 sqlid..