Mybatis是支持定制化sql、存储过程及高级映射的优秀的持久层框架，其主要就完成了2件事情：

封装JDBC操作

利用反射打通Java类与Sql语句之间的相互转换

MyBatis的主要设计目的就是让我们对执行SQL语句时对输入输出的数据管理更加方便，所以方便地写出SQL和方便地获取SQL的执行结果才是MyBatis的核心竞争力

与原生JDBC的对比

原生JDBC的缺点：

原生的JDBC操作数据库时，需要频繁的开关链接

查询数据库的结果集，需要人为的进行封装

JDBC中没有缓存处理

JDBC的sql语句写到Java文件

Mybatis框架

内部提供数据库连接池不需要频繁开关链接

半自动对象关系映射、实现结果集自动封装，但是sql需要自己写

有缓存而且是二级缓存

mybatis把sql写到xml配置文件中

Mybatis的主要成员

Configuration：MyBatis所有的配置信息都保存在Configuration对象中，配置文件中的大部分配置都会 存储到该类中

SqlSession：作为MyBatis工作的主要顶层API，表示和数据库交互时的会话，完成必要数据库增删改查功能

根据statement id，在mybatis配置对象configuration中获取到对应的mappedstatement对象，然后调用执行器来执行具体操作

Executor：MyBatis执行器，是MyBatis调度的核心，负责sql语句的生成和查询缓存的维护

根据传递的参数，完成sql语句的动态解析，生成BoundSql对象，供StatementHandler使用

为查询创建缓存，以提高性能

创建JDBC的Statement链接对象，传递给StatementHandler对象，返回List查询结果

StatementHandler：封装了JDBC Statement操作，负责对JDBC statement的操作，如设置参数等

对于JDBC的preparedStatement类型的对象，创建过程中，sql语句字符串会包含若干个？占位符，然后再赋值。StatementHandler通过parameterize（statement）方法对statement进行设值

StatementHandler通过List query(Statement statement,ResultHandler resultHandler)方法来完成执行Statement，和将Statement对象返回的resultSet封装成List

ParameterHandler：负责对用户传递的参数转换成JDBC Statement所对应的数据类型，对statement对象的？占位符进行赋值

ResultSetHandler：负责将JDBC返回的ResultSet结果集对象转换成List类型的集合

TypeHandler：负责java数据类型和jdbc数据类型（也可以说是数据表列类型）之间的映射和转换

MappedStatement：MappedStatement维护一条<select|update|delete|insert>节点的封装

SqlSource：负责根据用户传递的parameterObject，动态的生成SQL语句，将信息封装到BoundSql对象中并返回

BoundSql：表示动态生成的SQL语句以及相应的参数信息

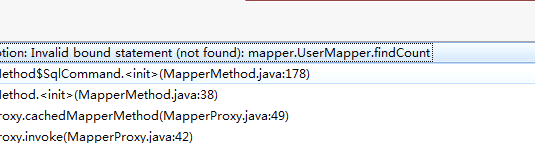
Mybatis接口调用原理

当程序执行时，通过接口方法调用

根据当前接口的路径匹配映射文件中的namespace

根据接口方法匹配映射文件中的id标识

如果执行正确则能成功将数据返回给接口，否则将报错



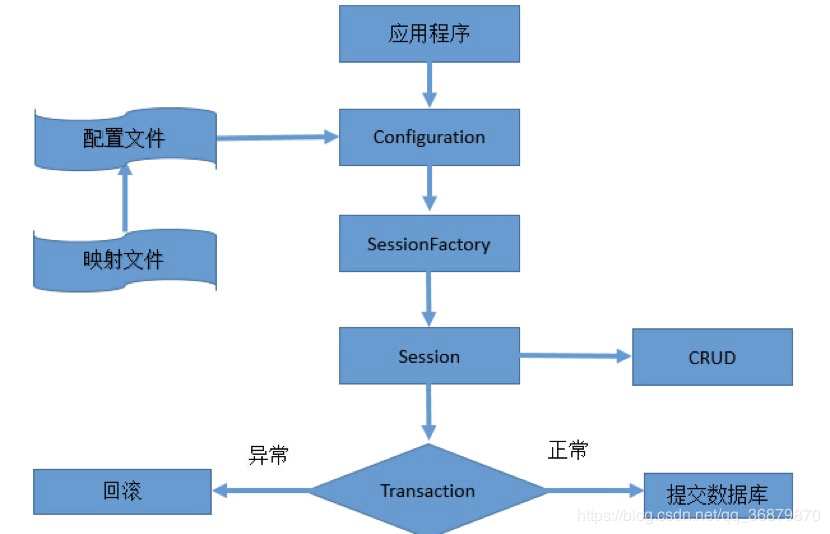
Mybatis的Sql执行顺序

简化版

创建sql会话工厂（sqlSessionFactory，这里用到了建造者模式），创建时需要使用到Mybatis的核心配置文件，在配置文件中需要制定映射配置文件

通过会话工厂得到会话对象

通过会话对象执行增删改查操作，在执行操作时需要找到对应的sql语句，而sql语句是存在于映射文件（mapper.xml）中的所以需要预先配置好映射文件（在映射文件中书写sql语句、装配参数和结果集映射相关操作）



详细1…

读取xml文件将属性和链接数据库的操作封装在Configuration对象中供后面的组件使用（namespace+Statementid）

创建sql会话工厂（sqlSessionFactory，这里用到了建造者模式）

通过通过sqlsesionfactory得到sqlsession(openSession)

为Mapper接口生成实现类(MapperProxy动态代理)

当代理类执行方法时，sqlsession执行SQL语句

StatementHandler预编译

ParameterHandler设置参数

Executor执行

ResultSetHandler封装结果集为List

Configuration文件的读取

其实就是XML文件Mapper信息的读取SAXReader

数据库连接信息以及所有Mapper的方法包括sql的类型、方法名、sql语句、返回类型和参数类型

方便理解，并不是源码

public class MapperBean {

private String interfaceName; //接口名

private List<Function> list; //接口下所有方法

}

public class Function {

private String sqltype;

private String funcName;

private String sql;

private Object resultType;

private String parameterType;

}

Configuration对象就是维护了一个Map<String,MapperBean>

源码

public interface CommonMapper {

List<String> findTimeList(CommonReport arg);

}

Mapper.xml的具体sql就不写了

Class XXX{

private static SqlSessionFactory sqlSessionFactory;

static{

String resource = "DbConfiguration.xml";

InputStream inputStream = null;

try {

inputStream = Resources.getResourceAsStream(resource);

} catch (IOException e) {

e.printStackTrace();

}

sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream,"fydb");

}

private static List<String> getTimeListFromDb(CommonReport arg) {

List<String> result = null;

SqlSession sqlSession = sqlSessionFactory.openSession();

CommonMapper commonMapper = sqlSession.getMapper(CommonMapper.class);

result = commonMapper.findTimeList(arg);

sqlSession.close();

return result;

}

}

我这里关注了sql的执行流程，XML的读取就不看了。

简单说明一下。MyBatis 在解析配置文件的节点的过程中，会调用 MapperRegistry 的 addMapper 方法将 Class 到 MapperProxyFactory 对象的映射关系存入到 knownMappers。

直接看getMapper操作

public class DefaultSqlSession implements SqlSession {

private Configuration configuration;

private Executor executor;

private boolean dirty;

public <T> T getMapper(Class<T> type) {

return this.configuration.getMapper(type, this);

}

~~~

}

public class Configuration {

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

return this.mapperRegistry.getMapper(type, sqlSession);

}

}

public class MapperRegistry {

private Configuration config;

private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap();

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

MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory)this.knownMappers.get(type);//从 knownMappers 中获取与 type 对应的 MapperProxyFactory

if (mapperProxyFactory == null) {

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

} else {

try {

//创建代理代理对象

return mapperProxyFactory.newInstance(sqlSession);

} catch (Exception var5) {

throw new BindingException("Error getting mapper instance. Cause: " + var5, var5);

}

}

}

}

Configuration类中有很多属性。。没有注解。。。

得到MapperProxyFactory对象后，即可调用工厂方法为Mapper创建代理对象

public class MapperProxyFactory<T> {

private final Class<T> mapperInterface;

private Map<Method, MapperMethod> methodCache = new ConcurrentHashMap();

protected T newInstance(MapperProxy<T> mapperProxy) {

return Proxy.newProxyInstance(this.mapperInterface.getClassLoader(), new Class[]{this.mapperInterface}, mapperProxy);//JDK动态代理

}

public T newInstance(SqlSession sqlSession) {

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

return this.newInstance(mapperProxy);

}

}

而JDK动态代理最后一个参数是关键，实现了InvocationHandler接口，然后将对象作为参数传给重载方法，并在重载方法中调用 JDK 动态代理接口为 Mapper 生成代理对象。

当代理对象执行接口方法时，会被Mapper内的invoke回调函数捕获

public class MapperProxy<T> implements InvocationHandler, Serializable {

private static final long serialVersionUID = -6424540398559729838L;

private final SqlSession sqlSession;

private final Class<T> mapperInterface;

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;

}

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

//如果是定义在Object类的方法，则直接执行，当时这一步没看懂。。多亏了田忠波前辈的注解。。

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

return method.invoke(this, args);

} else {

// 从缓存中获取 MapperMethod 对象，若缓存未命中，则创建 MapperMethod 对象

MapperMethod mapperMethod = this.cachedMapperMethod(method);

// 调用 execute 方法执行 SQL

return mapperMethod.execute(this.sqlSession, args);

}

}

private MapperMethod cachedMapperMethod(Method method) {

MapperMethod mapperMethod = (MapperMethod)this.methodCache.get(method);

if (mapperMethod == null) {

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

this.methodCache.put(method, mapperMethod);

}

return mapperMethod;

}

}

MapperMethod的创建

public class MapperMethod {

private final SqlCommand command;

private final MethodSignature method;

public MapperMethod(Class<?> mapperInterface, Method method, Configuration config) {

// 创建 SqlCommand 对象，该对象包含一些和 SQL 相关的信息

this.command = new SqlCommand(config, mapperInterface, method);

// 创建 MethodSignature 对象，由类名可知，该对象包含了被拦截方法的一些信息 this.method = new MethodSignature(config, mapperInterface, method);

}

}

最后通过该对象中的 execute 方法执行 SQL(方法中传入了sqlSession)，就是这一步进行数据库操作，那么先来看看关键的SqlSession对象

SqlSession重要的四个对象

Execute：调度执行StatementHandler、ParmmeterHandler、ResultHandler执行相应的SQL语句；

StatementHandler：使用数据库中Statement（PrepareStatement）执行操作，即底层是封装好了的prepareStatement；

ParammeterHandler：处理SQL参数；

ResultHandler：结果集ResultSet封装处理返回。

源码中当然有，但是找起来比较麻烦

package org.apache.ibatis.session.defaults;

public class DefaultSqlSession implements SqlSession {

private Configuration configuration;

private Executor executor;

private boolean dirty;

}

接着来看一下Executor

package org.apache.ibatis.executor;

public class SimpleExecutor extends BaseExecutor {

private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {

Connection connection = this.getConnection(statementLog);

Statement stmt = handler.prepare(connection);

handler.parameterize(stmt);

return stmt;

}

~~~//其他几个方法都有StatementHandler的出现

}

package org.apache.ibatis.executor.statement;

public abstract class BaseStatementHandler implements StatementHandler {

protected final Configuration configuration;

protected final ObjectFactory objectFactory;

protected final TypeHandlerRegistry typeHandlerRegistry;

protected final ResultSetHandler resultSetHandler;

protected final ParameterHandler parameterHandler;

protected final Executor executor;

protected final MappedStatement mappedStatement;

protected final RowBounds rowBounds;

protected BoundSql boundSql;

}

好的找齐了

这些对象归属结构也对应了sql执行的顺序，再来看看execute方法源码

public class MapperMethod {

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

Object result;

Object param;

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

param = this.method.convertArgsToSqlCommandParam(args);

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

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

param = this.method.convertArgsToSqlCommandParam(args);

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

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

param = this.method.convertArgsToSqlCommandParam(args);

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

} else {

if (SqlCommandType.SELECT != this.command.getType()) {

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

}

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

this.executeWithResultHandler(sqlSession, args);

result = null;

} else if (this.method.returnsMany()) {

result = this.executeForMany(sqlSession, args);

} else if (this.method.returnsMap()) {

result = this.executeForMap(sqlSession, args);

} else {

param = this.method.convertArgsToSqlCommandParam(args);

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

}

}

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

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

} else {

return result;

}

}

}

public class DefaultSqlSession implements SqlSession {

public <T> T selectOne(String statement, Object parameter) {

List<T> list = this.selectList(statement, parameter);

if (list.size() == 1) {

return list.get(0);

} else if (list.size() > 1) {

throw new TooManyResultsException("Expected one result (or null) to be returned by selectOne(), but found: " + list.size());

} else {

return null;

}

}

public <E> List<E> selectList(String statement) {

return this.selectList(statement, (Object)null);

}

public <E> List<E> selectList(String statement, Object parameter) {

return this.selectList(statement, parameter, RowBounds.DEFAULT);

}

public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {

List var6;

try {

MappedStatement ms = this.configuration.getMappedStatement(statement);

List<E> result = this.executor.query(ms, this.wrapCollection(parameter), rowBounds, Executor.NO\_RESULT\_HANDLER);

var6 = result;

} catch (Exception var10) {

throw ExceptionFactory.wrapException("Error querying database. Cause: " + var10, var10);

} finally {

ErrorContext.instance().reset();

}

return var6;

}

}

可以看到selectOne方法也很真实。。最后还是调用了selectList方法。。。

接下来执行前的设置参数等等问题下回再学了。。。

其他一些基础实操问题

#和$的区别

#{}含有预编译的效果，能够防止sql注入共计，为参数添加了一堆""

对传递进来的参数直接拼接在sql中以列名会参数时使用 {}对传递进来的参数直接拼接在sql中以列名会参数时使用对传递进来的参数直接拼接在sql中以列名会参数时使用，以及需要使用declare的时候，因为declare需要在sql执行前先声明

例子可以看我之前的博客https://blog.csdn.net/qq\_36879870/article/details/89919572

实体类与表中的字段名不一样怎么办

字段定义别名

resultMap映射

如何获取自动生成的（主）键值

mysql

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

<selectKey keyProperty="id" order="AFTER" resultType="int">

select LAST\_INSERT\_ID()

</selectKey>

INSERT INTO USER(username,birthday,sex,address) VALUES(#{username},#{birthday},#{sex},#{address})

</insert>

oracle

先查询序列得到主键，将主键设置到对象中，再将对象插入数据库

<!-- oracle

在执行insert之前执行select 序列.nextval() from dual取出序列最大值，将值设置到user对象 的id属性

-->

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

<selectKey keyProperty="id" order="BEFORE" resultType="int">

select 序列.nextval() from dual

</selectKey>

INSERT INTO USER(id,username,birthday,sex,address) VALUES( 序列.nextval(),#{username},#{birthday},#{sex},#{address})

</insert>

如何传递多个参数

顺序传参

//对应的xml,#{0}代表接收的是dao层中的第一个参数，#{1}代表dao层中第二参数，更多参数一致往后加即可。

<select id="selectUser"resultMap="BaseResultMap">

select \* fromuser\_user\_t whereuser\_name = #{0} anduser\_area=#{1}

</select>

使用@param注解来命名参数

public interface usermapper {

user selectuser(@param(“username”) string username,

@param(“hashedpassword”) string hashedpassword);

}

<select id=”selectuser” resulttype=”user”>

select id, username, hashedpassword

from some\_table

where username = #{username}

and hashedpassword = #{hashedpassword}

</select>

使用Map来装载，mybatis根据key自动找到对应Map中value

list，动态sql

对象

动态sql？有哪些？执行原理？

mybatis动态sql可以让我们在xml映射文件内，以标签的形式编写动态sql，完成逻辑判断和动态拼接sql的能力。

trim|where|set|foreach|if|choose|when|otherwise|bind

<where>

<if test="shopCategoryId!=null">sql语句</if>

<foreach item="numberList" collection="list" open="(" separator="," close=")">

#{numberList.num}

</foreach>

</where>

执行原理：使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接sql，以此来完成动态sql的功能

Mybatis的XML映射文件中，不通的XML映射文件，id是否可以重复

如果配置了namespace的话当然可以重复，因为我们的statement实际上就是namespace+id

但是如果没有配置namespace的话，那么相同的id就会导致覆盖了

为什么说Mybatis是半自动ORM映射工具？与全自动的区别在哪里

Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。

而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql，所以，称之为半自动ORM映射工具。

通常一个xml映射文件，都会写一个Dao接口与之对应，请问这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗？

Dao接口，就是我们说的Mapper接口，接口的全限名，就是映射文件中的namespace的值，接口的方法名就是映射文件中MappedStatement的id值，接口方法内的参数，就是传递给sql的参数

mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法拼接字符串作为key值，可唯一定位一个MappedStatement。

Dao接口里的方法，是不能重载的，因为是全限名+方法名的保存和寻找策略。

Dao接口里的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Dao接口生成代理proxy对象，代理对象proxy会拦截接口方法，转而执行MappedStatement所代表的sql，然后将sql执行结果返回。

Mybatis有哪些Executor执行器？他们之间的区别是什么？

Mybatis有三种基本的Executor执行器

SimpleExecutor：没执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象

ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置在Map<String,Statement>内，供下一次使用

BatchExecutor：执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理（addBatch）中，等待统一执行（execute），它缓存了多个Statement对象，每个Statement对象都是addBatch完毕后，等待逐一执行executeBatch批处理，与JDBC批处理相同