MyBatis允许定义拦截器对SQL操作进行预处理。官方文档使用示例如下：

http://www.mybatis.org/mybatis-3/configuration.html#plugins

**plugins**

MyBatis allows you to intercept calls to at certain points within the execution of a mapped statement. By default, MyBatis allows plug-ins to intercept method calls of:

* Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
* ParameterHandler (getParameterObject, setParameters)
* ResultSetHandler (handleResultSets, handleOutputParameters)
* StatementHandler (prepare, parameterize, batch, update, query)

The details of these classes methods can be discovered by looking at the full method signature of each, and the source code which is available with each MyBatis release. You should understand the behaviour of the method you’re overriding, assuming you’re doing something more than just monitoring calls. If you attempt to modify or override the behaviour of a given method, you’re likely to break the core of MyBatis. These are low level classes and methods, so use plug-ins with caution.

Using plug-ins is pretty simple given the power they provide. Simply implement the Interceptor interface, being sure to specify the signatures you want to intercept.

// ExamplePlugin.java

@Intercepts({@Signature(

type= Executor.class,

method = "update",

args = {MappedStatement.class,Object.class})})

public class ExamplePlugin implements Interceptor {

public Object intercept(Invocation invocation) throws Throwable {

return invocation.proceed();

}

public Object plugin(Object target) {

return Plugin.wrap(target, this);

}

public void setProperties(Properties properties) {

}

}

<!-- mybatis-config.xml -->

<plugins>

<plugin interceptor="org.mybatis.example.ExamplePlugin">

<property name="someProperty" value="100"/>

</plugin>

</plugins>

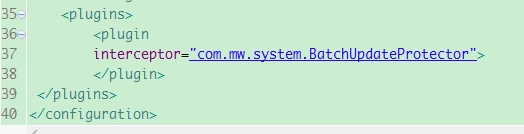
The plug-in above will intercept all calls to the "update" method on the Executor instance, which is an internal object responsible for the low level execution of mapped statements.

**NOTE** **Overriding the Configuration Class**

In addition to modifying core MyBatis behaviour with plugins, you can also override the Configuration class entirely. Simply extend it and override any methods inside, and pass it into the call to the sqlSessionFactoryBuilder.build(myConfig) method. Again though, this could have a severe impact on the behaviour of MyBatis, so use caution.

在实际项目中我是这样使用的：

mybatis配置文件定义plugins标签



编写拦截器类**BatchUpdateProtector**

package com.mw.system;

import java.util.Properties;

import org.apache.commons.lang.StringUtils;

import org.apache.ibatis.executor.Executor;

import org.apache.ibatis.mapping.BoundSql;

import org.apache.ibatis.mapping.MappedStatement;

import org.apache.ibatis.mapping.MappedStatement.Builder;

import org.apache.ibatis.mapping.SqlCommandType;

import org.apache.ibatis.mapping.SqlSource;

import org.apache.ibatis.plugin.Interceptor;

import org.apache.ibatis.plugin.Intercepts;

import org.apache.ibatis.plugin.Invocation;

import org.apache.ibatis.plugin.Plugin;

import org.apache.ibatis.plugin.Signature;

/\*\*

\* <p>

\* MyBatis语句禁止批量更新的拦截器

\* </p>

\*

\* <pre>

\* mapper示例：必须在update语句的最后面定义[primaryCloumn="orderNo"]，其中orderNo是能标识orders表的主键（逻辑主键或者业务主键）

\* <update id="updateOrder" parameterType="java.util.HashMap">

\* <![CDATA[

\* update

\* orders

\* set

\* status = #{currentStatus}

\* ]]>

\* <where>

\* <if test="orderNo != null and orderNo != ''">

\* and orderNo = #{orderNo, jdbcType=VARCHAR}

\* </if>

\* <if test="preStatus != null and preStatus != ''">

\* and status = #{preStatus, jdbcType=INTEGER}

\* </if>

\* </where>

\* [primaryCloumn="orderNo"]

\* </update>

\* </pre>

\*

\*

\*/

@Intercepts({ @Signature(type = Executor.class, method = "update", args = {

MappedStatement.class, Object.class }) })

public class BatchUpdateProtector implements Interceptor {

private final static String presentColumnTag = "primaryCloumn";// 定义where条件中必须出现的字段

/\*\*

\* <p>

\* 对update和delete语句进行拦截

\* </p>

\*

\* @see org.apache.ibatis.plugin.Interceptor#intercept(org.apache.ibatis.plugin

\* .Invocation)

\*/

public Object intercept(Invocation invocation) throws Throwable {

// 拦截update

if (isUpdateMethod(invocation)) {

invocation.getArgs()[0] = checkAndResetSQL(invocation);

}

// 拦截delete

else if (isDeleteMethod(invocation)) {

invocation.getArgs()[0] = checkAndResetSQL(invocation);

}

return invocation.proceed();

}

/\*\*

\* <p>

\* 判断该操作是否是update操作

\* </p>

\*

\* @param invocation

\* @return 是否是update操作

\*/

private boolean isUpdateMethod(Invocation invocation) {

if (invocation.getArgs()[0] instanceof MappedStatement) {

MappedStatement mappedStatement = (MappedStatement) invocation

.getArgs()[0];

return SqlCommandType.UPDATE.equals(mappedStatement

.getSqlCommandType());

}

return false;

}

/\*\*

\* <p>

\* 判断该操作是否是delete操作

\* </p>

\*

\* @param invocation

\* @return 是否是delete操作

\*/

private boolean isDeleteMethod(Invocation invocation) {

if (invocation.getArgs()[0] instanceof MappedStatement) {

MappedStatement mappedStatement = (MappedStatement) invocation

.getArgs()[0];

return SqlCommandType.DELETE.equals(mappedStatement

.getSqlCommandType());

}

return false;

}

/\*\*

\* <p>

\* 检查update语句中是否定义了presentColumn，并且删除presentColumn后重新设置update语句

\* </p>

\*

\* @param invocation

\* invocation实例

\* @return MappedStatement 返回删除presentColumn之后的MappedStatement实例

\*/

private Object checkAndResetSQL(Invocation invocation) {

MappedStatement mappedStatement = (MappedStatement) invocation

.getArgs()[0];

Object parameter = invocation.getArgs()[1];

mappedStatement.getSqlSource().getBoundSql(parameter);

BoundSql boundSql = mappedStatement.getBoundSql(parameter);

String resetSql = doCheckAndResetSQL(boundSql.getSql());

return getMappedStatement(mappedStatement, boundSql, resetSql);

}

/\*\*

\* <p>

\* 检查update语句中是否定义了presentColumn，并且删除presentColumn后重新设置update语句

\* </p>

\*

\* @param sql

\* mapper中定义的sql语句(带有presentColumn的定义)

\* @return 删除presentColumn之后的sql

\*/

private String doCheckAndResetSQL(String sql) {

if (sql.indexOf(presentColumnTag) > 0) {

// presentColumn的定义是否在sql的最后面

if (sql.indexOf("]") + 1 == sql.length()) {

int startIndex = sql.indexOf("[");

int endIndex = sql.indexOf("]");

String presentColumnText = sql.substring(startIndex,

endIndex + 1);// [presentColumn="orderNo"]

// 剔除标记逻辑主键相关内容之后的sql，该sql才是真正执行update的sql语句

sql = StringUtils.replace(sql, presentColumnText, "");

String[] subSqls = sql.toLowerCase().split("where");

String[] keyWords = presentColumnText.split("\"");

// 获取主键,比如orderNo

String keyWord = keyWords[1];

// 判断是否带有where条件并且在where条件中是否存在主键keyWord

if (subSqls.length == 2 && subSqls[1].indexOf(keyWord) == -1) {

throw new IllegalArgumentException("该update语句:" + sql

+ "是批量更新sql，不允许执行。因为它的的where条件中未包含能表示主键的字段"

+ keyWord + ",所以会导致批量更新。");

}

} else {

throw new IllegalArgumentException("[" + presentColumnTag

+ "=\"xxx\"\"]必须定义在update语句的最后面.");

}

} else {

throw new IllegalArgumentException("在mapper文件中定义的update语句必须包含"

+ presentColumnTag + "，它用于定义该sql的主键（逻辑主键或者业务主键），比如id");

}

return sql;

}

/\*\*

\* <p>

\* 通过验证关键字段不能为空之后的sql重新构建mappedStatement

\* </p>

\*

\* @param mappedStatement

\* 重新构造sql之前的mappedStatement实例

\* @param boundSql

\* 重新构造sql之前的boundSql实例

\* @param resetSql

\* 验证关键字段不能为空之后的sql

\* @return 重新构造之后的mappedStatement实例

\*/

private Object getMappedStatement(MappedStatement mappedStatement,

BoundSql boundSql, String resetSql) {

final BoundSql newBoundSql = new BoundSql(

mappedStatement.getConfiguration(), resetSql,

boundSql.getParameterMappings(), boundSql.getParameterObject());

Builder builder = new MappedStatement.Builder(

mappedStatement.getConfiguration(), mappedStatement.getId(),

new SqlSource() {

public BoundSql getBoundSql(Object parameterObject) {

return newBoundSql;

}

}, mappedStatement.getSqlCommandType());

String propertyStr = null;

if(mappedStatement.getKeyProperties() != null) {

for(String p : mappedStatement.getKeyProperties()) {

propertyStr = propertyStr + p + ",";

}

}

builder.cache(mappedStatement.getCache());

builder.fetchSize(mappedStatement.getFetchSize());

builder.flushCacheRequired(mappedStatement.isFlushCacheRequired());

builder.keyGenerator(mappedStatement.getKeyGenerator());

builder.keyProperty(propertyStr);

builder.resource(mappedStatement.getResource());

builder.resultMaps(mappedStatement.getResultMaps());

builder.resultSetType(mappedStatement.getResultSetType());

builder.statementType(mappedStatement.getStatementType());

builder.timeout(mappedStatement.getTimeout());

builder.useCache(mappedStatement.isUseCache());

return builder.build();

}

public Object plugin(Object target) {

return Plugin.wrap(target, this);

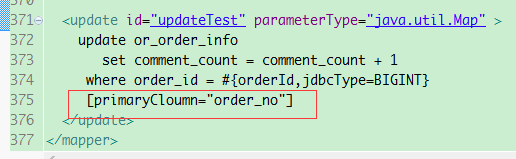
}

public void setProperties(Properties properties) {

}

}

在update标签的sql后面添加主键描述



如果写错或参数传入错误，则运行抛出异常

