# 构建SqlSessionFactory的过程

## SqlSessionFactoryBuilder.build()

public SqlSessionFactory build(Reader reader)

public SqlSessionFactory build(Reader reader, String environment)

public SqlSessionFactory build(Reader reader, Properties properties)

public SqlSessionFactory build(Reader reader, String environment, Properties properties)

public SqlSessionFactory build(InputStream inputStream)

public SqlSessionFactory build(InputStream inputStream, String environment)

public SqlSessionFactory build(InputStream inputStream, Properties properties)

public SqlSessionFactory build(InputStream inputStream,Stringe nvironment,Properties properties)

//真正构建逻辑为

public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {

try {

XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);

return build(parser.parse());

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error building SqlSession.", e);

} finally {

ErrorContext.instance().reset();

try {

inputStream.close();

} catch (IOException e) {

// Intentionally ignore. Prefer previous error.

}

}

}

将最后的解析后的Configuration包装成为SqlSessionFactory

public SqlSessionFactory build(Configuration config) {

return new DefaultSqlSessionFactory(config);

}

# XMLConfigBuilder解析器

## XMLConfigBuilder是继承自BaseBuilder

public abstract class BaseBuilder {

protected final Configuration configuration;

protected final TypeAliasRegistry typeAliasRegistry;

protected final TypeHandlerRegistry typeHandlerRegistry;

}

## 构建XMLConfigBuilder

public XMLConfigBuilder(Reader reader)

public XMLConfigBuilder(Reader reader, String environment)

public XMLConfigBuilder(Reader reader, String environment, Properties props)

public XMLConfigBuilder(InputStream inputStream)

public XMLConfigBuilder(InputStream inputStream, String environment)

public XMLConfigBuilder(InputStream inputStream, String environment, Properties props)

真正构建方法：

public XMLConfigBuilder(InputStream inputStream, String environment, Properties props) {

this(new XPathParser(inputStream, true, props, new XMLMapperEntityResolver()), environment, props);

}

通过XPathParser进行创建

private XMLConfigBuilder(XPathParser parser, String environment, Properties props) {

super(new Configuration());

ErrorContext.instance().resource("SQL Mapper Configuration");

this.configuration.setVariables(props);

this.parsed = false;

this.environment = environment;

this.parser = parser;

}

## 解析配置

public Configuration parse() {

if (parsed) {

throw new BuilderException("Each XMLConfigBuilder can only be used once.");

}

parsed = true;

parseConfiguration(parser.evalNode("/configuration"));

return configuration;

}

private void parseConfiguration(XNode root) {

try {

//issue #117 read properties first

propertiesElement(root.evalNode("properties"));

Properties settings = settingsAsProperties(root.evalNode("settings"));

loadCustomVfs(settings);

loadCustomLogImpl(settings);

typeAliasesElement(root.evalNode("typeAliases"));

pluginElement(root.evalNode("plugins"));

objectFactoryElement(root.evalNode("objectFactory"));

objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));

reflectorFactoryElement(root.evalNode("reflectorFactory"));

settingsElement(settings);

// read it after objectFactory and objectWrapperFactory issue #631

environmentsElement(root.evalNode("environments"));

databaseIdProviderElement(root.evalNode("databaseIdProvider"));

typeHandlerElement(root.evalNode("typeHandlers"));

mapperElement(root.evalNode("mappers"));

} catch (Exception e) {

throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);

}

}

### 解析properties

propertiesElement(root.evalNode("properties"));

### 解析settings

Properties settings = settingsAsProperties(root.evalNode("settings"));

loadCustomVfs(settings);

loadCustomLogImpl(settings);

### 解析typeAliases

typeAliasesElement(root.evalNode("typeAliases"));

使用到的类TypeAliasRegistry typeAliasRegistry,一个是package，一个是alisa、type

### 解析插件

pluginElement(root.evalNode("plugins"));

使用到的类InterceptorChain、Interceptor

### 解析对象工厂objectFactory

objectFactoryElement(root.evalNode("objectFactory"));

### 解析对象加工工厂

objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));

<https://blog.csdn.net/ycxzuoxin/article/details/104843818/>对象加工工厂

### 解析反射reflectorFactory

reflectorFactoryElement(root.evalNode("reflectorFactory"));

<https://blog.csdn.net/weixin_39723544/article/details/88942249> 反射工厂

### 将一般的settings属性设置到Configuration

settingsElement(settings);

### 解析environments

配置如下：

<environments default="development">

<environment id="development">

<transactionManager type="JDBC">

<property name="..." value="..."/>

</transactionManager>

<dataSource type="POOLED">

<property name="driver" value="${driver}"/>

<property name="url" value="${url}"/>

<property name="username" value="${username}"/>

<property name="password" value="${password}"/>

</dataSource>

</environment>

</environments>

解析语句：

environmentsElement(root.evalNode("environments"));

包含解析transactionManager和datasource

TransactionFactorytxFactory= transactionManagerElement(child.evalNode("transactionManager"));

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

DataSource dataSource = dsFactory.getDataSource();

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

.transactionFactory(txFactory)

.dataSource(dataSource);

==>对应的是Environment类

public final class Environment {

private final String id;

private final TransactionFactory transactionFactory;

private final DataSource dataSource;

}

### 解析数据库厂商标识

配置：

<databaseIdProvider type="DB\_VENDOR">

<property name="SQL Server" value="sqlserver"/>

<property name="DB2" value="db2"/>

<property name="Oracle" value="oracle" />

</databaseIdProvider>

解析语句：

databaseIdProviderElement(root.evalNode("databaseIdProvider"));

==>对应的类是DatabaseIdProvider,默认实现是VendorDatabaseIdProvider

### typeHandlers解析

配置：

a.<typeHandlers>

<typeHandler handler="org.mybatis.example.ExampleTypeHandler"/>

</typeHandlers>

b.<typeHandlers>

<package name="org.mybatis.example"/>

</typeHandlers>

c.<typeHandlers>

<typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler" javaType="java.math.RoundingMode"/>

</typeHandlers>

解析语句：

typeHandlerElement(root.evalNode("typeHandlers"));

利用TypeHandlerRegistry进行注册

### mappers解析

配置：

<!-- 使用相对于类路径的资源引用 -->

<mappers>

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

<mapper resource="org/mybatis/builder/BlogMapper.xml"/>

<mapper resource="org/mybatis/builder/PostMapper.xml"/>

</mappers>

<!-- 使用完全限定资源定位符（URL） -->

<mappers>

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

<mapper url="file:///var/mappers/BlogMapper.xml"/>

<mapper url="file:///var/mappers/PostMapper.xml"/>

</mappers>

<!-- 使用映射器接口实现类的完全限定类名 -->

<mappers>

<mapper class="org.mybatis.builder.AuthorMapper"/>

<mapper class="org.mybatis.builder.BlogMapper"/>

<mapper class="org.mybatis.builder.PostMapper"/>

</mappers>

<!-- 将包内的映射器接口实现全部注册为映射器 -->

<mappers>

<package name="org.mybatis.builder"/>

</mappers>

解析语句：

mapperElement(root.evalNode("mappers"));

(1)使用resource或者url方式，使用XMLMapperBuilder进行解析

InputStream inputStream = Resources.getResourceAsStream(resource);

XMLMapperBuilder mapperParser = new XMLMapperBuilder(inputStream, configuration, resource, configuration.getSqlFragments());

mapperParser.parse();

public XMLMapperBuilder(InputStream inputStream, Configuration configuration, String resource, Map<String, XNode> sqlFragments) {

this(new XPathParser(inputStream, true, configuration.getVariables(), new XMLMapperEntityResolver()),

configuration, resource, sqlFragments);

}

private XMLMapperBuilder(XPathParser parser, Configuration configuration, String resource, Map<String, XNode> sqlFragments) {

super(configuration);

this.builderAssistant = new MapperBuilderAssistant(configuration, resource);

this.parser = parser;

this.sqlFragments = sqlFragments;

this.resource = resource;

}

(2)

#### 解析mapper.xml文件

public void parse() {

if (!configuration.isResourceLoaded(resource)) {

configurationElement(parser.evalNode("/mapper"));

configuration.addLoadedResource(resource);

bindMapperForNamespace();

}

parsePendingResultMaps();

parsePendingCacheRefs();

parsePendingStatements();

}

##### 解析并加载xml文件

private void configurationElement(XNode context) {

try {

String namespace = context.getStringAttribute("namespace");

if (namespace == null || namespace.equals("")) {

throw new BuilderException("Mapper's namespace cannot be empty");

}

builderAssistant.setCurrentNamespace(namespace);

cacheRefElement(context.evalNode("cache-ref"));

cacheElement(context.evalNode("cache"));

parameterMapElement(context.evalNodes("/mapper/parameterMap"));

resultMapElements(context.evalNodes("/mapper/resultMap"));

sqlElement(context.evalNodes("/mapper/sql"));

buildStatementFromContext(context.evalNodes("select|insert|update|delete"));

} catch (Exception e) {

throw new BuilderException("Error parsing Mapper XML. The XML location is '" + resource + "'. Cause: " + e, e);

}

}

###### 解析cache-ref

cacheRefElement(context.evalNode("cache-ref"));

###### 解析cache

cacheElement(context.evalNode("cache"));

默认cache为PerpetualCache

默认缓存策略为LRU,其余缓存策略有

[FIFO", FifoCache

LRU", LruCache

SOFT", SoftCache

WEAK", WeakCache

]

最后configuration.addCache(cache);

###### 解析mapper/parameterMap

parameterMap元素为什么会被弃用

在mybatis的实现中，parameterMap元素配置的优先级比较低，它只会在一个语句完全没有通过其他途径配置映射的时候才会生效：行内参数映射

行内参数映射

parameterMap有两个必填的属性，其中id属性用于定义parameterMap在当前mapper文件中的唯一标志，type则表示parameterMap的具体类型。在parameterMap下可以出现一个或多个parameter子元素，parameter子元素的DTD定义如下

在parameterMap下可以出现一个或多个parameter子元素，parameter子元素的DTD定义如下

parameter元素有七个属性，除了property属性是必填的以外，其他参数均是非必填的：

property参数用于指定参数的名称,为必填项。

javaType用来指定参数的java类型。

jdbcType用来指定参数的JDBC类型。

mode用来配置参数的类型，其中IN表示入参，OUT表示出参,INOUT表示即为出参也为入参。

resultMap参数，当mode参数为OUT/INOUT，且jdbcType为CURSOR的时候，需要指定一个resultMap来映射参数结果集。

scale参数用来指定参数小数保留位数，比较有趣的是，虽然DTD中定义的是scale，但是Mybatis实际解析的却是numericScale.

typeHandler用来指定处理参数在java类型和jdbc类型之间互相转换的转换器类型。

每一个parameter元素，都控制着对应的java属性值如何转换为sql参数。

<https://www.jianshu.com/p/73602ce42709>

###### 解析mapper/resultMap

private ResultMap resultMapElement(XNode resultMapNode, List<ResultMapping> additionalResultMappings, Class<?> enclosingType) {

ErrorContext.instance().activity("processing " + resultMapNode.getValueBasedIdentifier());

String type = resultMapNode.getStringAttribute("type",

resultMapNode.getStringAttribute("ofType",

resultMapNode.getStringAttribute("resultType",

resultMapNode.getStringAttribute("javaType"))));

Class<?> typeClass = resolveClass(type);

if (typeClass == null) {

typeClass = inheritEnclosingType(resultMapNode, enclosingType);

}

Discriminator discriminator = null;

List<ResultMapping> resultMappings = new ArrayList<>(additionalResultMappings);

List<XNode> resultChildren = resultMapNode.getChildren();

for (XNode resultChild : resultChildren) {

if ("constructor".equals(resultChild.getName())) {

processConstructorElement(resultChild, typeClass, resultMappings);

} else if ("discriminator".equals(resultChild.getName())) {

discriminator = processDiscriminatorElement(resultChild, typeClass, resultMappings);

} else {

List<ResultFlag> flags = new ArrayList<>();

if ("id".equals(resultChild.getName())) {

flags.add(ResultFlag.ID);

}

resultMappings.add(buildResultMappingFromContext(resultChild, typeClass, flags));

}

}

String id = resultMapNode.getStringAttribute("id",

resultMapNode.getValueBasedIdentifier());

String extend = resultMapNode.getStringAttribute("extends");

Boolean autoMapping = resultMapNode.getBooleanAttribute("autoMapping");

ResultMapResolver resultMapResolver = new ResultMapResolver(builderAssistant, id, typeClass, extend, discriminator, resultMappings, autoMapping);

try {

return resultMapResolver.resolve();

} catch (IncompleteElementException e) {

configuration.addIncompleteResultMap(resultMapResolver);

throw e;

}

}

1. 解析constructor

配置:

column:数据库中的列名，或者是列的别名。一般情况下，这和传递给 resultSet.getString(columnName) 方法的参数一样。

javaType:一个 Java 类的完全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。 如果你映射到一个 JavaBean，MyBatis 通常可以推断类型。然而，如果你映射到的是 HashMap，那么你应该明确地指定 javaType 来保证行为与期望的相一致。

jdbcType:JDBC 类型，所支持的 JDBC 类型参见这个表格之前的“支持的 JDBC 类型”。 只需要在可能执行插入、更新和删除的且允许空值的列上指定 JDBC 类型。这是 JDBC 的要求而非 MyBatis 的要求。如果你直接面向 JDBC 编程，你需要对可能存在空值的列指定这个类型。

typeHandler :我们在前面讨论过默认的类型处理器。使用这个属性，你可以覆盖默认的类型处理器。 这个属性值是一个类型处理器实现类的完全限定名，或者是类型别名。

select :用于加载复杂类型属性的映射语句的 ID，它会从 column 属性中指定的列检索数据，作为参数传递给此 select 语句。具体请参考关联元素。

resultMap:结果映射的 ID，可以将嵌套的结果集映射到一个合适的对象树中。 它可以作为使用额外 select 语句的替代方案。它可以将多表连接操作的结果映射成一个单一的 ResultSet。这样的 ResultSet 将会将包含重复或部分数据重复的结果集。为了将结果集正确地映射到嵌套的对象树中，MyBatis 允许你 “串联”结果映射，以便解决嵌套结果集的问题。想了解更多内容，请参考下面的关联元素。

name:构造方法形参的名字。从 3.4.3 版本开始，通过指定具体的参数名，你可以以任意顺序写入 arg 元素。参看上面的解释。

方法:

processConstructorElement(resultChild, typeClass, resultMappings);

1. 解析鉴别器

discriminator = processDiscriminatorElement(resultChild, typeClass, resultMappings);

1. 解析id和result

resultMappings.add(buildResultMappingFromContext(resultChild, typeClass, flags))

###### 解析/mapper/sql

解析完成之后放在XMLMapperBuilder.sqlFragments Map<String, XNode> sqlFragments

###### 解析select|insert|update|delete

XMLMapperBuilder:

private void buildStatementFromContext(List<XNode> list) {

if (configuration.getDatabaseId() != null) {

buildStatementFromContext(list, configuration.getDatabaseId());

}

buildStatementFromContext(list, null);

}

private void buildStatementFromContext(List<XNode> list, String requiredDatabaseId) {

for (XNode context : list) {

final XMLStatementBuilder statementParser = new XMLStatementBuilder(configuration, builderAssistant, context, requiredDatabaseId);

try {

statementParser.parseStatementNode();

} catch (IncompleteElementException e) {

configuration.addIncompleteStatement(statementParser);

}

}

}

关键解析类:

XMLStatementBuilder:

public void parseStatementNode() {

String id = context.getStringAttribute("id");

String databaseId = context.getStringAttribute("databaseId");

if (!databaseIdMatchesCurrent(id, databaseId, this.requiredDatabaseId)) {

return;

}

String nodeName = context.getNode().getNodeName();

SqlCommandType sqlCommandType = SqlCommandType.valueOf(nodeName.toUpperCase(Locale.ENGLISH));

boolean isSelect = sqlCommandType == SqlCommandType.SELECT;

boolean flushCache = context.getBooleanAttribute("flushCache", !isSelect);

boolean useCache = context.getBooleanAttribute("useCache", isSelect);

boolean resultOrdered = context.getBooleanAttribute("resultOrdered", false);

// Include Fragments before parsing

XMLIncludeTransformer includeParser = new XMLIncludeTransformer(configuration, builderAssistant);

includeParser.applyIncludes(context.getNode());

String parameterType = context.getStringAttribute("parameterType");

Class<?> parameterTypeClass = resolveClass(parameterType);

String lang = context.getStringAttribute("lang");

LanguageDriver langDriver = getLanguageDriver(lang);

// Parse selectKey after includes and remove them.

processSelectKeyNodes(id, parameterTypeClass, langDriver);

// Parse the SQL (pre: <selectKey> and <include> were parsed and removed)

KeyGenerator keyGenerator;

String keyStatementId = id + SelectKeyGenerator.SELECT\_KEY\_SUFFIX;

keyStatementId = builderAssistant.applyCurrentNamespace(keyStatementId, true);

if (configuration.hasKeyGenerator(keyStatementId)) {

keyGenerator = configuration.getKeyGenerator(keyStatementId);

} else {

keyGenerator = context.getBooleanAttribute("useGeneratedKeys",

configuration.isUseGeneratedKeys() && SqlCommandType.INSERT.equals(sqlCommandType))

? Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;

}

SqlSource sqlSource = langDriver.createSqlSource(configuration, context, parameterTypeClass);

StatementType statementType = StatementType.valueOf(context.getStringAttribute("statementType", StatementType.PREPARED.toString()));

Integer fetchSize = context.getIntAttribute("fetchSize");

Integer timeout = context.getIntAttribute("timeout");

String parameterMap = context.getStringAttribute("parameterMap");

String resultType = context.getStringAttribute("resultType");

Class<?> resultTypeClass = resolveClass(resultType);

String resultMap = context.getStringAttribute("resultMap");

String resultSetType = context.getStringAttribute("resultSetType");

ResultSetType resultSetTypeEnum = resolveResultSetType(resultSetType);

if (resultSetTypeEnum == null) {

resultSetTypeEnum = configuration.getDefaultResultSetType();

}

String keyProperty = context.getStringAttribute("keyProperty");

String keyColumn = context.getStringAttribute("keyColumn");

String resultSets = context.getStringAttribute("resultSets");

builderAssistant.addMappedStatement(id, sqlSource, statementType, sqlCommandType,

fetchSize, timeout, parameterMap, parameterTypeClass, resultMap, resultTypeClass,

resultSetTypeEnum, flushCache, useCache, resultOrdered,

keyGenerator, keyProperty, keyColumn, databaseId, langDriver, resultSets);

}

(1) Include Fragments before parsing(inclue代码块处理)

XMLIncludeTransformer includeParser = new XMLIncludeTransformer(configuration, builderAssistant);

includeParser.applyIncludes(context.getNode());

处理过程：

XMLIncludeTransformer：

public void applyIncludes(Node source) {

Properties variablesContext = new Properties();

Properties configurationVariables = configuration.getVariables();

Optional.ofNullable(configurationVariables).ifPresent(variablesContext::putAll);

applyIncludes(source, variablesContext, false);

}

private void applyIncludes(Node source, final Properties variablesContext, boolean included) {

if (source.getNodeName().equals("include")) {

Node toInclude = findSqlFragment(getStringAttribute(source, "refid"), variablesContext);

Properties toIncludeContext = getVariablesContext(source, variablesContext);

applyIncludes(toInclude, toIncludeContext, true);

if (toInclude.getOwnerDocument() != source.getOwnerDocument()) {

toInclude = source.getOwnerDocument().importNode(toInclude, true);

}

source.getParentNode().replaceChild(toInclude, source);

while (toInclude.hasChildNodes()) {

toInclude.getParentNode().insertBefore(toInclude.getFirstChild(), toInclude);

}

toInclude.getParentNode().removeChild(toInclude);

} else if (source.getNodeType() == Node.ELEMENT\_NODE) {

if (included && !variablesContext.isEmpty()) {

// replace variables in attribute values

NamedNodeMap attributes = source.getAttributes();

for (int i = 0; i < attributes.getLength(); i++) {

Node attr = attributes.item(i);

attr.setNodeValue(PropertyParser.parse(attr.getNodeValue(), variablesContext));

}

}

NodeList children = source.getChildNodes();

for (int i = 0; i < children.getLength(); i++) {

applyIncludes(children.item(i), variablesContext, included);

}

} else if (included && (source.getNodeType() == Node.TEXT\_NODE || source.getNodeType() == Node.CDATA\_SECTION\_NODE)

&& !variablesContext.isEmpty()) {

// replace variables in text node

source.setNodeValue(PropertyParser.parse(source.getNodeValue(), variablesContext));

}

}

(2)解析selectKey

对于不支持自动生成主键列的数据库和可能不支持自动生成主键的 JDBC 驱动，MyBatis 有另外一种方法来生成主键。

这里有一个简单（也很傻）的示例，它可以生成一个随机 ID（不建议实际使用，这里只是为了展示 MyBatis 处理问题的灵活性和宽容度）：

<insert id="insertAuthor">

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

select CAST(RANDOM()\*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1

</selectKey>

insert into Author

(id, username, password, email,bio, favourite\_section)

values

(#{id}, #{username}, #{password}, #{email}, #{bio}, #{favouriteSection,jdbcType=VARCHAR})

</insert>

在上面的示例中，首先会运行 selectKey 元素中的语句，并设置 Author 的 id，然后才会调用插入语句。这样就实现了数据库自动生成主键类似的行为，同时保持了 Java 代码的简洁。

selectKey 元素描述如下：

<selectKey

keyProperty="id"

resultType="int"

order="BEFORE"

statementType="PREPARED">

selectKey 元素的属性

属性 描述

keyProperty selectKey 语句结果应该被设置到的目标属性。如果生成列不止一个，可以用逗号分隔多个属性名称。

keyColumn 返回结果集中生成列属性的列名。如果生成列不止一个，可以用逗号分隔多个属性名称。

resultType 结果的类型。通常 MyBatis 可以推断出来，但是为了更加准确，写上也不会有什么问题。MyBatis 允许将任何简单类型用作主键的类型，包括字符串。如果生成列不止一个，则可以使用包含期望属性的 Object 或 Map。

order 可以设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它首先会生成主键，设置 keyProperty 再执行插入语句。如果设置为 AFTER，那么先执行插入语句，然后是 selectKey 中的语句 - 这和 Oracle 数据库的行为相似，在插入语句内部可能有嵌入索引调用。

statementType 和前面一样，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE 类型的映射语句，分别代表 Statement, PreparedStatement 和 CallableStatement 类型。

XMLStatementBuilder：

// Parse selectKey after includes and remove them.

processSelectKeyNodes(id, parameterTypeClass, langDriver);

private void processSelectKeyNodes(String id, Class<?> parameterTypeClass, LanguageDriver langDriver) {

List<XNode> selectKeyNodes = context.evalNodes("selectKey");

if (configuration.getDatabaseId() != null) {

parseSelectKeyNodes(id, selectKeyNodes, parameterTypeClass, langDriver, configuration.getDatabaseId());

}

parseSelectKeyNodes(id, selectKeyNodes, parameterTypeClass, langDriver, null);

removeSelectKeyNodes(selectKeyNodes);

}

private void parseSelectKeyNodes(String parentId, List<XNode> list, Class<?> parameterTypeClass, LanguageDriver langDriver, String skRequiredDatabaseId) {

for (XNode nodeToHandle : list) {

String id = parentId + SelectKeyGenerator.SELECT\_KEY\_SUFFIX;

String databaseId = nodeToHandle.getStringAttribute("databaseId");

if (databaseIdMatchesCurrent(id, databaseId, skRequiredDatabaseId)) {

parseSelectKeyNode(id, nodeToHandle, parameterTypeClass, langDriver, databaseId);

}

}

}

private void parseSelectKeyNode(String id, XNode nodeToHandle, Class<?> parameterTypeClass, LanguageDriver langDriver, String databaseId) {

String resultType = nodeToHandle.getStringAttribute("resultType");

Class<?> resultTypeClass = resolveClass(resultType);

StatementType statementType = StatementType.valueOf(nodeToHandle.getStringAttribute("statementType", StatementType.PREPARED.toString()));

String keyProperty = nodeToHandle.getStringAttribute("keyProperty");

String keyColumn = nodeToHandle.getStringAttribute("keyColumn");

boolean executeBefore = "BEFORE".equals(nodeToHandle.getStringAttribute("order", "AFTER"));

// defaults

boolean useCache = false;

boolean resultOrdered = false;

KeyGenerator keyGenerator = NoKeyGenerator.INSTANCE;

Integer fetchSize = null;

Integer timeout = null;

boolean flushCache = false;

String parameterMap = null;

String resultMap = null;

ResultSetType resultSetTypeEnum = null;

//➀获取SqlSource

SqlSource sqlSource = langDriver.createSqlSource(configuration, nodeToHandle, parameterTypeClass);

SqlCommandType sqlCommandType = SqlCommandType.SELECT;

//➁构建MappedStatement

builderAssistant.addMappedStatement(id, sqlSource, statementType, sqlCommandType,

fetchSize, timeout, parameterMap, parameterTypeClass, resultMap, resultTypeClass,

resultSetTypeEnum, flushCache, useCache, resultOrdered,

keyGenerator, keyProperty, keyColumn, databaseId, langDriver, null);

id = builderAssistant.applyCurrentNamespace(id, false);

MappedStatement keyStatement = configuration.getMappedStatement(id, false);

configuration.addKeyGenerator(id, new SelectKeyGenerator(keyStatement, executeBefore));

}

(3)获取SqlSource

SqlSource sqlSource = langDriver.createSqlSource(configuration, nodeToHandle, parameterTypeClass);

XMLLanguageDriver：

@Override

public SqlSource createSqlSource(Configuration configuration, XNode script, Class<?> parameterType) {

XMLScriptBuilder builder = new XMLScriptBuilder(configuration, script, parameterType);

return builder.parseScriptNode();

}

XMLScriptBuilder：

public SqlSource parseScriptNode() {

MixedSqlNode rootSqlNode = parseDynamicTags(context);

SqlSource sqlSource;

if (isDynamic) {

sqlSource = new DynamicSqlSource(configuration, rootSqlNode);

} else {

sqlSource = new RawSqlSource(configuration, rootSqlNode, parameterType);

}

return sqlSource;

}

protected MixedSqlNode parseDynamicTags(XNode node) {

List<SqlNode> contents = new ArrayList<>();

NodeList children = node.getNode().getChildNodes();

for (int i = 0; i < children.getLength(); i++) {

XNode child = node.newXNode(children.item(i));

if (child.getNode().getNodeType() == Node.CDATA\_SECTION\_NODE || child.getNode().getNodeType() == Node.TEXT\_NODE) {

String data = child.getStringBody("");

TextSqlNode textSqlNode = new TextSqlNode(data);

if (textSqlNode.isDynamic()) {

contents.add(textSqlNode);

isDynamic = true;

} else {

contents.add(new StaticTextSqlNode(data));

}

} else if (child.getNode().getNodeType() == Node.ELEMENT\_NODE) { // issue #628

String nodeName = child.getNode().getNodeName();

NodeHandler handler = nodeHandlerMap.get(nodeName);

if (handler == null) {

throw new BuilderException("Unknown element <" + nodeName + "> in SQL statement.");

}

handler.handleNode(child, contents);

isDynamic = true;

}

}

return new MixedSqlNode(contents);

}

（4）构建MappedStatement

builderAssistant.addMappedStatement(id, sqlSource, statementType, sqlCommandType,

fetchSize, timeout, parameterMap, parameterTypeClass, resultMap, resultTypeClass,

resultSetTypeEnum, flushCache, useCache, resultOrdered,

keyGenerator, keyProperty, keyColumn, databaseId, langDriver, null);

MapperBuilderAssistant:

public MappedStatement addMappedStatement(

String id,

SqlSource sqlSource,

StatementType statementType,

SqlCommandType sqlCommandType,

Integer fetchSize,

Integer timeout,

String parameterMap,

Class<?> parameterType,

String resultMap,

Class<?> resultType,

ResultSetType resultSetType,

boolean flushCache,

boolean useCache,

boolean resultOrdered,

KeyGenerator keyGenerator,

String keyProperty,

String keyColumn,

String databaseId,

LanguageDriver lang,

String resultSets) {

if (unresolvedCacheRef) {

throw new IncompleteElementException("Cache-ref not yet resolved");

}

id = applyCurrentNamespace(id, false);

boolean isSelect = sqlCommandType == SqlCommandType.SELECT;

MappedStatement.Builder statementBuilder = new MappedStatement.Builder(configuration, id, sqlSource, sqlCommandType)

.resource(resource)

.fetchSize(fetchSize)

.timeout(timeout)

.statementType(statementType)

.keyGenerator(keyGenerator)

.keyProperty(keyProperty)

.keyColumn(keyColumn)

.databaseId(databaseId)

.lang(lang)

.resultOrdered(resultOrdered)

.resultSets(resultSets)

.resultMaps(getStatementResultMaps(resultMap, resultType, id))

.resultSetType(resultSetType)

.flushCacheRequired(valueOrDefault(flushCache, !isSelect))

.useCache(valueOrDefault(useCache, isSelect))

.cache(currentCache);

ParameterMap statementParameterMap = getStatementParameterMap(parameterMap, parameterType, id);

if (statementParameterMap != null) {

statementBuilder.parameterMap(statementParameterMap);

}

MappedStatement statement = statementBuilder.build();

configuration.addMappedStatement(statement);

return statement;

}

1. 增加SelectKeyGenerator

configuration.addKeyGenerator(id, new SelectKeyGenerator (keyStatement, executeBefore));

说明:

1. executeBefore再真正语句执行前还是后
2. keyStatement selectKey执行语句信息

public class SelectKeyGenerator implements KeyGenerator {

public static final String SELECT\_KEY\_SUFFIX = "!selectKey";

private final boolean executeBefore;

private final MappedStatement keyStatement;

public SelectKeyGenerator(MappedStatement keyStatement, boolean executeBefore) {

this.executeBefore = executeBefore;

this.keyStatement = keyStatement;

}

@Override

public void processBefore(Executor executor, MappedStatement ms, Statement stmt, Object parameter) {

if (executeBefore) {

processGeneratedKeys(executor, ms, parameter);

}

}

@Override

public void processAfter(Executor executor, MappedStatement ms, Statement stmt, Object parameter) {

if (!executeBefore) {

processGeneratedKeys(executor, ms, parameter);

}

}

}

(5)解析完selectKey语句后，真正执行语句同解析selectKey语句

##### 动态语句处理

if

choose (when, otherwise)

trim (where, set)

foreach

相应的处理器（NodeHandler）

nodeHandlerMap.put("trim", new TrimHandler());

nodeHandlerMap.put("where", new WhereHandler());

nodeHandlerMap.put("set", new SetHandler());

nodeHandlerMap.put("foreach", new ForEachHandler());

nodeHandlerMap.put("if", new IfHandler());

nodeHandlerMap.put("choose", new ChooseHandler());

nodeHandlerMap.put("when", new IfHandler());

nodeHandlerMap.put("otherwise", new OtherwiseHandler());

nodeHandlerMap.put("bind", new BindHandler());

###### TrimHandler

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

...

</trim>

public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {

MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);

String prefix = nodeToHandle.getStringAttribute("prefix");

String prefixOverrides = nodeToHandle.getStringAttribute("prefixOverrides");

String suffix = nodeToHandle.getStringAttribute("suffix");

String suffixOverrides = nodeToHandle.getStringAttribute("suffixOverrides");

TrimSqlNode trim = new TrimSqlNode(configuration, mixedSqlNode, prefix, prefixOverrides, suffix, suffixOverrides);

targetContents.add(trim);

}

###### WhereHandler

public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {

MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);

WhereSqlNode where = new WhereSqlNode(configuration, mixedSqlNode);

targetContents.add(where);

}

###### SetHandler

同WhereSqlNode

###### ForEachHandler

public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {

MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);

String collection = nodeToHandle.getStringAttribute("collection");

String item = nodeToHandle.getStringAttribute("item");

String index = nodeToHandle.getStringAttribute("index");

String open = nodeToHandle.getStringAttribute("open");

String close = nodeToHandle.getStringAttribute("close");

String separator = nodeToHandle.getStringAttribute("separator");

ForEachSqlNode forEachSqlNode = new ForEachSqlNode(configuration, mixedSqlNode, collection, index, item, open, close, separator);

targetContents.add(forEachSqlNode);

}

###### IfHandler

public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {

MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);

String test = nodeToHandle.getStringAttribute("test");

IfSqlNode ifSqlNode = new IfSqlNode(mixedSqlNode, test);

targetContents.add(ifSqlNode);

}

###### ChooseHandler

public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {

List<SqlNode> whenSqlNodes = new ArrayList<>();

List<SqlNode> otherwiseSqlNodes = new ArrayList<>();

handleWhenOtherwiseNodes(nodeToHandle, whenSqlNodes, otherwiseSqlNodes);

SqlNode defaultSqlNode = getDefaultSqlNode(otherwiseSqlNodes);

ChooseSqlNode chooseSqlNode = new ChooseSqlNode(whenSqlNodes, defaultSqlNode);

targetContents.add(chooseSqlNode);

}

private void handleWhenOtherwiseNodes(XNode chooseSqlNode, List<SqlNode> ifSqlNodes, List<SqlNode> defaultSqlNodes) {

List<XNode> children = chooseSqlNode.getChildren();

for (XNode child : children) {

String nodeName = child.getNode().getNodeName();

NodeHandler handler = nodeHandlerMap.get(nodeName);

if (handler instanceof IfHandler) {

handler.handleNode(child, ifSqlNodes);

} else if (handler instanceof OtherwiseHandler) {

handler.handleNode(child, defaultSqlNodes);

}

}

}

private SqlNode getDefaultSqlNode(List<SqlNode> defaultSqlNodes) {

SqlNode defaultSqlNode = null;

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

defaultSqlNode = defaultSqlNodes.get(0);

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

throw new BuilderException("Too many default (otherwise) elements in choose statement.");

}

return defaultSqlNode;

}

###### When对应的IfHandler

###### OtherwiseHandler

public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {

MixedSqlNode mixedSqlNode = parseDynamicTags(nodeToHandle);

targetContents.add(mixedSqlNode);

}

###### BindHandler

public void handleNode(XNode nodeToHandle, List<SqlNode> targetContents) {

final String name = nodeToHandle.getStringAttribute("name");

final String expression = nodeToHandle.getStringAttribute("value");

final VarDeclSqlNode node = new VarDeclSqlNode(name, expression);

targetContents.add(node);

}

##### Xml tags

SqlNode

###### StaticTextSqlNode

public class StaticTextSqlNode implements SqlNode {

private final String text;

}

###### MixedSqlNode

public class MixedSqlNode implements SqlNode {

private final List<SqlNode> contents;

}

###### TextSqlNode

public class TextSqlNode implements SqlNode {

private final String text;

private final Pattern injectionFilter;

}

###### TrimSqlNode：

public class TrimSqlNode implements SqlNode {

private final SqlNode contents;

private final String prefix;

private final String suffix;

private final List<String> prefixesToOverride;

private final List<String> suffixesToOverride;

private final Configuration configuration;

}

###### VarDeclSqlNode

public class VarDeclSqlNode implements SqlNode {

private final String name;

private final String expression;

}

###### WhereSqlNode（继承TrimSqlNode）

public class WhereSqlNode extends TrimSqlNode {

private static List<String> prefixList = Arrays.asList("AND ","OR ","AND\n", "OR\n", "AND\r", "OR\r", "AND\t", "OR\t");

public WhereSqlNode(Configuration configuration, SqlNode contents) {

super(configuration, contents, "WHERE", prefixList, null, null);

}

}

ChooseSqlNode

ForEachSqlNode

IfSqlNode

###### SetSqlNode(继承TrimSqlNode)

public class SetSqlNode extends TrimSqlNode {

private static final List<String> COMMA = Collections.singletonList(",");

public SetSqlNode(Configuration configuration,SqlNode contents) {

super(configuration, contents, "SET", COMMA, null, COMMA);

}

}

##### SqlSource

###### DynamicSqlSource

public class DynamicSqlSource implements SqlSource {

private final Configuration configuration;

private final SqlNode rootSqlNode;

public DynamicSqlSource(Configuration configuration, SqlNode rootSqlNode) {

this.configuration = configuration;

this.rootSqlNode = rootSqlNode;

}

}

###### RawSqlSource

public class RawSqlSource implements SqlSource {

private final SqlSource sqlSource;

public RawSqlSource(Configuration configuration, SqlNode rootSqlNode, Class<?> parameterType) {

this(configuration, getSql(configuration, rootSqlNode), parameterType);

}

public RawSqlSource(Configuration configuration, String sql, Class<?> parameterType) {

SqlSourceBuilder sqlSourceParser = new SqlSourceBuilder(configuration);

Class<?> clazz = parameterType == null ? Object.class : parameterType;

sqlSource = sqlSourceParser.parse(sql, clazz, new HashMap<>());

}

private static String getSql(Configuration configuration, SqlNode rootSqlNode) {

DynamicContext context = new DynamicContext(configuration, null);

rootSqlNode.apply(context);

return context.getSql();

}

}

###### StaticSqlSource

public class StaticSqlSource implements SqlSource {

private final String sql;

private final List<ParameterMapping> parameterMappings;

private final Configuration configuration;

}

###### ProviderSqlSource

public class ProviderSqlSource implements SqlSource {

private final Configuration configuration;

private final Class<?> providerType;

private final LanguageDriver languageDriver;

private final Method mapperMethod;

private final Method providerMethod;

private final String[] providerMethodArgumentNames;

private final Class<?>[] providerMethodParameterTypes;

private final ProviderContext providerContext;

private final Integer providerContextIndex;

}

##### 解析与xml绑定的接口

private void bindMapperForNamespace() {

String namespace = builderAssistant.getCurrentNamespace();

if (namespace != null) {

Class<?> boundType = null;

try {

boundType = Resources.classForName(namespace);

} catch (ClassNotFoundException e) {

//ignore, bound type is not required

}

if (boundType != null) {

if (!configuration.hasMapper(boundType)) {

// Spring may not know the real resource name so we set a flag

// to prevent loading again this resource from the mapper interface

// look at MapperAnnotationBuilder#loadXmlResource

configuration.addLoadedResource("namespace:" + namespace);

configuration.addMapper(boundType);

}

}

}

}

Configuration:

public <T> void addMapper(Class<T> type) {

mapperRegistry.addMapper(type);

}

MapperRegistry:

public <T> void addMapper(Class<T> type) {

if (type.isInterface()) {

if (hasMapper(type)) {

throw new BindingException("Type " + type + " is already known to the MapperRegistry.");

}

boolean loadCompleted = false;

try {

knownMappers.put(type, new MapperProxyFactory<>(type));

// It's important that the type is added before the parser is run

// otherwise the binding may automatically be attempted by the

// mapper parser. If the type is already known, it won't try.

MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config, type);

parser.parse();

loadCompleted = true;

} finally {

if (!loadCompleted) {

knownMappers.remove(type);

}

}

}

}

重点是接口中基于注解的解析:

MapperAnnotationBuilder parser = new MapperAnnotationBuilder(config, type);

parser.parse();

MapperAnnotationBuilder：

public void parse() {

String resource = type.toString();

if (!configuration.isResourceLoaded(resource)) {

loadXmlResource();

configuration.addLoadedResource(resource);

assistant.setCurrentNamespace(type.getName());

//➀解析缓存CacheNamespace

parseCache();

//➁解析缓存引用CacheNamespaceRef

parseCacheRef();

Method[] methods = type.getMethods();

for (Method method : methods) {

try {

// issue #237

if (!method.isBridge()) {

parseStatement(method);

}

} catch (IncompleteElementException e) {

configuration.addIncompleteMethod(new MethodResolver(this, method));

}

}

}

parsePendingMethods();

}

MapperAnnotationBuilder：

void parseStatement(Method method) {

Class<?> parameterTypeClass = getParameterType(method);

LanguageDriver languageDriver = getLanguageDriver(method);

SqlSource sqlSource = getSqlSourceFromAnnotations(method, parameterTypeClass, languageDriver);

if (sqlSource != null) {

Options options = method.getAnnotation(Options.class);

final String mappedStatementId = type.getName() + "." + method.getName();

Integer fetchSize = null;

Integer timeout = null;

StatementType statementType = StatementType.PREPARED;

ResultSetType resultSetType = configuration.getDefaultResultSetType();

SqlCommandType sqlCommandType = getSqlCommandType(method);

boolean isSelect = sqlCommandType == SqlCommandType.SELECT;

boolean flushCache = !isSelect;

boolean useCache = isSelect;

KeyGenerator keyGenerator;

String keyProperty = null;

String keyColumn = null;

if (SqlCommandType.INSERT.equals(sqlCommandType) || SqlCommandType.UPDATE.equals(sqlCommandType)) {

// first check for SelectKey annotation - that overrides everything else

SelectKey selectKey = method.getAnnotation(SelectKey.class);

if (selectKey != null) {

keyGenerator = handleSelectKeyAnnotation(selectKey, mappedStatementId, getParameterType(method), languageDriver);

keyProperty = selectKey.keyProperty();

} else if (options == null) {

keyGenerator = configuration.isUseGeneratedKeys() ? Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;

} else {

keyGenerator = options.useGeneratedKeys() ? Jdbc3KeyGenerator.INSTANCE : NoKeyGenerator.INSTANCE;

keyProperty = options.keyProperty();

keyColumn = options.keyColumn();

}

} else {

keyGenerator = NoKeyGenerator.INSTANCE;

}

if (options != null) {

if (FlushCachePolicy.TRUE.equals(options.flushCache())) {

flushCache = true;

} else if (FlushCachePolicy.FALSE.equals(options.flushCache())) {

flushCache = false;

}

useCache = options.useCache();

fetchSize = options.fetchSize() > -1 || options.fetchSize() == Integer.MIN\_VALUE ? options.fetchSize() : null;

//issue #348

timeout = options.timeout() > -1 ? options.timeout() : null;

statementType = options.statementType();

if (options.resultSetType() != ResultSetType.DEFAULT) {

resultSetType = options.resultSetType();

}

}

String resultMapId = null;

ResultMap resultMapAnnotation = method.getAnnotation(ResultMap.class);

if (resultMapAnnotation != null) {

resultMapId = String.join(",", resultMapAnnotation.value());

} else if (isSelect) {

resultMapId = parseResultMap(method);

}

assistant.addMappedStatement(

mappedStatementId,

sqlSource,

statementType,

sqlCommandType,

fetchSize,

timeout,

// ParameterMapID

null,

parameterTypeClass,

resultMapId,

getReturnType(method),

resultSetType,

flushCache,

useCache,

// TODO gcode issue #577

false,

keyGenerator,

keyProperty,

keyColumn,

// DatabaseID

null,

languageDriver,

// ResultSets

options != null ? nullOrEmpty(options.resultSets()) : null);

}

}

获取SqlSource

MapperAnnotationBuilder：

private SqlSource getSqlSourceFromAnnotations(Method method, Class<?> parameterType, LanguageDriver languageDriver) {

try {

Class<? extends Annotation> sqlAnnotationType = getSqlAnnotationType(method);

Class<? extends Annotation> sqlProviderAnnotationType = getSqlProviderAnnotationType(method);

if (sqlAnnotationType != null) {

if (sqlProviderAnnotationType != null) {

throw new BindingException("You cannot supply both a static SQL and SqlProvider to method named " + method.getName());

}

Annotation sqlAnnotation = method.getAnnotation(sqlAnnotationType);

final String[] strings = (String[]) sqlAnnotation.getClass().getMethod("value").invoke(sqlAnnotation);

return buildSqlSourceFromStrings(strings, parameterType, languageDriver);

} else if (sqlProviderAnnotationType != null) {

Annotation sqlProviderAnnotation = method.getAnnotation(sqlProviderAnnotationType);

return new ProviderSqlSource(assistant.getConfiguration(), sqlProviderAnnotation, type, method);

}

return null;

} catch (Exception e) {

throw new BuilderException("Could not find value method on SQL annotation. Cause: " + e, e);

}

}

mybatise注解类型

SQL\_ANNOTATION\_TYPES.add(Select.class);

SQL\_ANNOTATION\_TYPES.add(Insert.class);

SQL\_ANNOTATION\_TYPES.add(Update.class);

SQL\_ANNOTATION\_TYPES.add(Delete.class);

SQL\_PROVIDER\_ANNOTATION\_TYPES.add(SelectProvider.class);

SQL\_PROVIDER\_ANNOTATION\_TYPES.add(InsertProvider.class);

SQL\_PROVIDER\_ANNOTATION\_TYPES.add(UpdateProvider.class);

SQL\_PROVIDER\_ANNOTATION\_TYPES.add(DeleteProvider.class);

private SqlSource buildSqlSourceFromStrings(String[] strings, Class<?> parameterTypeClass, LanguageDriver languageDriver) {

final StringBuilder sql = new StringBuilder();

for (String fragment : strings) {

sql.append(fragment);

sql.append(" ");

}

return languageDriver.createSqlSource(configuration, sql.toString().trim(), parameterTypeClass);

}

构建SqlSource对象

RawLanguageDriver, XMLLanguageDriver

public SqlSource createSqlSource(Configuration configuration, String script, Class<?> parameterType) {

// issue #3

if (script.startsWith("<script>")) {

XPathParser parser = new XPathParser(script, false, configuration.getVariables(), new XMLMapperEntityResolver());

return createSqlSource(configuration, parser.evalNode("/script"), parameterType);

} else {

// issue #127

script = PropertyParser.parse(script, configuration.getVariables());

TextSqlNode textSqlNode = new TextSqlNode(script);

if (textSqlNode.isDynamic()) {

return new DynamicSqlSource(configuration, textSqlNode);

} else {

return new RawSqlSource(configuration, script, parameterType);

}

}

}

处理注解中中的SelectKey

private KeyGenerator handleSelectKeyAnnotation(SelectKey selectKeyAnnotation, String baseStatementId, Class<?> parameterTypeClass, LanguageDriver languageDriver) {

String id = baseStatementId + SelectKeyGenerator.SELECT\_KEY\_SUFFIX;

Class<?> resultTypeClass = selectKeyAnnotation.resultType();

StatementType statementType = selectKeyAnnotation.statementType();

String keyProperty = selectKeyAnnotation.keyProperty();

String keyColumn = selectKeyAnnotation.keyColumn();

boolean executeBefore = selectKeyAnnotation.before();

// defaults

boolean useCache = false;

KeyGenerator keyGenerator = NoKeyGenerator.INSTANCE;

Integer fetchSize = null;

Integer timeout = null;

boolean flushCache = false;

String parameterMap = null;

String resultMap = null;

ResultSetType resultSetTypeEnum = null;

SqlSource sqlSource = buildSqlSourceFromStrings(selectKeyAnnotation.statement(), parameterTypeClass, languageDriver);

SqlCommandType sqlCommandType = SqlCommandType.SELECT;

assistant.addMappedStatement(id, sqlSource, statementType, sqlCommandType, fetchSize, timeout, parameterMap, parameterTypeClass, resultMap, resultTypeClass, resultSetTypeEnum,

flushCache, useCache, false,

keyGenerator, keyProperty, keyColumn, null, languageDriver, null);

id = assistant.applyCurrentNamespace(id, false);

MappedStatement keyStatement = configuration.getMappedStatement(id, false);

SelectKeyGenerator answer = new SelectKeyGenerator(keyStatement, executeBefore);

configuration.addKeyGenerator(id, answer);

return answer;

}

##### Script用法

@ResultMap("BaseResultMap")

@Select("<script>" +

"select \* from user " +

"<where>" +

"<if test=\"age != null\"> age = #{age}</if>" +

"</where>" +

"</script>")

List<User> getUser4(@Param("age") Integer age);

##### @Options注解

//注解能够设置缓存时间，能够为对象生成自增的主键值

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.METHOD})

public @interface Options {

boolean useCache() default true;

boolean flushCache() default false;

ResultSetType resultSetType() default ResultSetType.FORWARD\_ONLY;

StatementType statementType() default StatementType.PREPARED;

int fetchSize() default -1;

int timeout() default -1;

boolean useGeneratedKeys() default false;

String keyProperty() default "id";

String keyColumn() default "";

}

##### ParameterMapping的解析

private ParameterMapping buildParameterMapping(String content) {

Map<String, String> propertiesMap = parseParameterMapping(content);

String property = propertiesMap.get("property");

Class<?> propertyType;

if (metaParameters.hasGetter(property)) { // issue #448 get type from additional params

propertyType = metaParameters.getGetterType(property);

} else if (typeHandlerRegistry.hasTypeHandler(parameterType)) {

propertyType = parameterType;

} else if (JdbcType.CURSOR.name().equals(propertiesMap.get("jdbcType"))) {

propertyType = java.sql.ResultSet.class;

} else if (property == null || Map.class.isAssignableFrom(parameterType)) {

propertyType = Object.class;

} else {

MetaClass metaClass = MetaClass.forClass(parameterType, configuration.getReflectorFactory());

if (metaClass.hasGetter(property)) {

propertyType = metaClass.getGetterType(property);

} else {

propertyType = Object.class;

}

}

ParameterMapping.Builder builder = new ParameterMapping.Builder(configuration, property, propertyType);

Class<?> javaType = propertyType;

String typeHandlerAlias = null;

for (Map.Entry<String, String> entry : propertiesMap.entrySet()) {

String name = entry.getKey();

String value = entry.getValue();

if ("javaType".equals(name)) {

javaType = resolveClass(value);

builder.javaType(javaType);

} else if ("jdbcType".equals(name)) {

builder.jdbcType(resolveJdbcType(value));

} else if ("mode".equals(name)) {

builder.mode(resolveParameterMode(value));

} else if ("numericScale".equals(name)) {

builder.numericScale(Integer.valueOf(value));

} else if ("resultMap".equals(name)) {

builder.resultMapId(value);

} else if ("typeHandler".equals(name)) {

typeHandlerAlias = value;

} else if ("jdbcTypeName".equals(name)) {

builder.jdbcTypeName(value);

} else if ("property".equals(name)) {

// Do Nothing

} else if ("expression".equals(name)) {

throw new BuilderException("Expression based parameters are not supported yet");

} else {

throw new BuilderException("An invalid property '" + name + "' was found in mapping #{" + content + "}. Valid properties are " + PARAMETER\_PROPERTIES);

}

}

if (typeHandlerAlias != null) {

builder.typeHandler(resolveTypeHandler(javaType, typeHandlerAlias));

}

return builder.build();

}

private Map<String, String> parseParameterMapping(String content) {

try {

return new ParameterExpression(content);

} catch (BuilderException ex) {

throw ex;

} catch (Exception ex) {

throw new BuilderException("Parsing error was found in mapping #{" + content + "}. Check syntax #{property|(expression), var1=value1, var2=value2, ...} ", ex);

}

}

### Mybatise注解配置

|  |  |  |  |
| --- | --- | --- | --- |
| **注解** | **目标** | **相对应的 XML** | **描述** |
| CacheNamespace | Class | <cache> | 为给定的命名空间 (比如类) 配置缓存。 属性:implemetation,eviction, flushInterval,size 和 readWrite。 |
| CacheNamespaceRef | Class | <cacheRef> | 参照另外一个命名空间的缓存来使用。 属性:value,应该是一个名空间的字 符串值(也就是类的完全限定名) 。 |
| ConstructorArgs | Method | <constructor> | 收集一组结果传递给一个对象的构造方法。属性:value,是形式参数 的数组 |
| Arg | Method | <arg>  <idArg> | 单独的构造方法参数 , 是 ConstructorArgs 集合的一部分。属性: id,column,javaType,typeHandler。 id 属性是布尔值, 来标识用于比较的属 性,和<idArg>XML 元素相似。 |
| TypeDiscriminator | Method | <discriminator> | 一组实例值被用来决定结果映射的表 现。 属性: column, javaType, jdbcType, typeHandler,cases。cases 属性就是实 例的数组。 |
| Case |  | <case> | 单独实例的值和它对应的映射。属性: value,type,results。Results 属性是结 果数组,因此这个注解和实际的 ResultMap 很相似,由下面的 Results 注解指定。 |
| Results |  | <resultMap> | 结果映射的列表, 包含了一个特别结果 列如何被映射到属性或字段的详情。 属 性:value, id。value 属性是 Result 注解的数组。 The id attribute is the name of the result mapping. |
| Result |  | <result>  <id> | 在列和属性或字段之间的单独结果映 射。属 性:id,column, property, javaType ,jdbcType ,type Handler, one,many。id 属性是一个布尔值,表 示了应该被用于比较(和在 XML 映射 中的<id>相似)的属性。one 属性是单 独 的 联 系, 和 <association> 相 似 , 而 many 属 性 是 对 集 合 而 言 的 , 和 <collection>相似。 它们这样命名是为了 避免名称冲突。 |
| One |  | <association> | 复杂类型的单独属性值映射。属性: select,已映射语句(也就是映射器方 法)的完全限定名,它可以加载合适类 型的实例。注意:联合映射在注解 API 中是不支持的。这是因为 Java 注解的 限制,不允许循环引用。 fetchType, which supersedes the global configuration parameterlazyLoadingEnabled for this mapping. |
| Many |  | <collection> | A mapping to a collection property of a complex type. Attributes: select, which is the fully qualified name of a mapped statement (i.e. mapper method) that can load a collection of instances of the appropriate types,fetchType, which supersedes the global configuration parameterlazyLoadingEnabled for this mapping. NOTE You will notice that join mapping is not supported via the Annotations API. This is due to the limitation in Java Annotations that does not allow for circular references. |
| MapKey |  |  | 复 杂 类 型 的 集合 属 性 映射 。 属 性 : select,是映射语句(也就是映射器方 法)的完全限定名,它可以加载合适类 型的一组实例。注意:联合映射在 Java 注解中是不支持的。这是因为 Java 注 解的限制,不允许循环引用。 |
| Options |  |  | 映射语句的属性 这个注解提供访问交换和配置选项的 宽广范围, 它们通常在映射语句上作为 属性出现。 而不是将每条语句注解变复 杂,Options 注解提供连贯清晰的方式 来访问它们。属性:useCache=true , flushCache=FlushCachePolicy.DEFAULT , resultSetType=FORWARD\_ONLY , statementType=PREPARED , fetchSize=-1 , , timeout=-1 useGeneratedKeys=false , keyProperty=”id” , keyColumn=”” , resultSets=””。 理解 Java 注解是很 重要的,因为没有办法来指定“null” 作为值。因此,一旦你使用了 Options 注解,语句就受所有默认值的支配。要 注意什么样的默认值来避免不期望的 行为。 |
| @Insert  @Update  @Delete  @Select |  | <insert>  <update>  <delete>  <select> | 这些注解中的每一个代表了执行的真 实 SQL。 它们每一个都使用字符串数组 (或单独的字符串)。如果传递的是字 符串数组, 它们由每个分隔它们的单独 空间串联起来。这就当用 Java 代码构 建 SQL 时避免了“丢失空间”的问题。 然而,如果你喜欢,也欢迎你串联单独 的字符串。属性:value,这是字符串 数组用来组成单独的 SQL 语句。 |
| @InsertProvider  @UpdateProvider  @DeleteProvider  @SelectProvider |  | <insert>  <update>  <delete>  <select> | 这些可选的 SQL 注解允许你指定一个 类名和一个方法在执行时来返回运行 允许创建动态 的 SQL。 基于执行的映射语句, MyBatis 会实例化这个类,然后执行由 provider 指定的方法. 该方法可以有选择地接受参数对象.(In MyBatis 3.4 or later, it’s allow multiple parameters) 属性: type,method。type 属性是类。method 属性是方法名。 注意: 这节之后是对 类的 讨论,它可以帮助你以干净,容于阅读 的方式来构建动态 SQL。 |
| @Param |  |  | 如果你的映射器的方法需要多个参数, 这个注解可以被应用于映射器的方法 参数来给每个参数一个名字。否则,多 参数将会以它们的顺序位置来被命名 (不包括任何 RowBounds 参数) 比如。 #{param1} , #{param2} 等 , 这 是 默 认 的 。 使 用 @Param(“person”),参数应该被命名为 #{person}。 |
| @SelectKey |  | <selectKey> | This annotation duplicates the <selectKey> functionality for methods annotated with @Insert, @InsertProvider, @Update or@UpdateProvider. It is ignored for other methods. If you specify a@SelectKey annotation, then MyBatis will ignore any generated key properties set via the @Options annotation, or configuration properties. Attributes: statement an array of strings which is the SQL statement to execute, keyProperty which is the property of the parameter object that will be updated with the new value, before which must be either true orfalse to denote if the SQL statement should be executed before or after the insert, resultType which is the Java type of the keyProperty, andstatementType=PREPARED. |
| @ResultMap |  |  | This annotation is used to provide the id of a <resultMap> element in an XML mapper to a @Select or @SelectProvider annotation. This allows annotated selects to reuse resultmaps that are defined in XML. This annotation will override any @Results or @ConstructorArgs annotation if both are specified on an annotated select. |
| @ResultType |  |  | This annotation is used when using a result handler. In that case, the return type is void so MyBatis must have a way to determine the type of object to construct for each row. If there is an XML result map, use the @ResultMap annotation. If the result type is specified in XML on the<select> element, then no other annotation is necessary. In other cases, use this annotation. For example, if a @Select annotated method will use a result handler, the return type must be void and this annotation (or @ResultMap) is required. This annotation is ignored unless the method return type is void. |
| @Flush |  |  | If this annotation is used, it can be called theSqlSession#flushStatements() via method defined at a Mapper interface.(MyBatis 3.3 or above) |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 构建默认DefaultSqlSessionFactory

SqlSessionFactoryBuilder中最后一个方法:

public SqlSessionFactory build(Configuration config) {

return new DefaultSqlSessionFactory(config);

}

# 创建SqlSession

public interface SqlSessionFactory {

SqlSession openSession();

SqlSession openSession(boolean autoCommit);

SqlSession openSession(Connection connection);

SqlSession openSession(TransactionIsolationLevel level);

SqlSession openSession(ExecutorType execType);

SqlSession openSession(ExecutorType execType, boolean autoCommit);

SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level);

SqlSession openSession(ExecutorType execType, Connection connection);

Configuration getConfiguration();

}

默认实现DefaultSqlSessionFactory

相应实现方法如下：

@Override

public SqlSession openSession() {

return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, false);

}

@Override

public SqlSession openSession(boolean autoCommit) {

return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, autoCommit);

}

@Override

public SqlSession openSession(ExecutorType execType) {

return openSessionFromDataSource(execType, null, false);

}

@Override

public SqlSession openSession(TransactionIsolationLevel level) {

return openSessionFromDataSource(configuration.getDefaultExecutorType(), level, false);

}

@Override

public SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level) {

return openSessionFromDataSource(execType, level, false);

}

@Override

public SqlSession openSession(ExecutorType execType, boolean autoCommit) {

return openSessionFromDataSource(execType, null, autoCommit);

}

@Override

public SqlSession openSession(Connection connection) {

return openSessionFromConnection(configuration.getDefaultExecutorType(), connection);

}

@Override

public SqlSession openSession(ExecutorType execType, Connection connection) {

return openSessionFromConnection(execType, connection);

}

@Override

public Configuration getConfiguration() {

return configuration;

}

//最后都汇总调用两个方法openSessionFromDataSource、openSessionFromConnection

Eg:openSessionFromDataSource

private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {

Transaction tx = null;

try {

final Environment environment = configuration.getEnvironment();

final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);

tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);

final Executor executor = configuration.newExecutor(tx, execType);

return new DefaultSqlSession(configuration, executor, autoCommit);

} catch (Exception e) {

closeTransaction(tx); // may have fetched a connection so lets call close()

throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);

} finally {

ErrorContext.instance().reset();

}

}

说明：

TransactionFactory的默认实现是ManagedTransactionFactory，别名managed

另外一个实现是JdbcTransactionFactory，别名是JDBC

步骤：

(1)获取事务工厂

TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);

(2)创建事务

tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);

(3)创建执行器

Executor executor = configuration.newExecutor(tx, execType);

(4)创建SqlSession默认实现

new DefaultSqlSession(configuration, executor, autoCommit);

# 事务Transaction

public interface Transaction {

/\*\*

\* Retrieve inner database connection.

\* @return DataBase connection

\* @throws SQLException

\*/

Connection getConnection() throws SQLException;

/\*\*

\* Commit inner database connection.

\* @throws SQLException

\*/

void commit() throws SQLException;

/\*\*

\* Rollback inner database connection.

\* @throws SQLException

\*/

void rollback() throws SQLException;

/\*\*

\* Close inner database connection.

\* @throws SQLException

\*/

void close() throws SQLException;

/\*\*

\* Get transaction timeout if set.

\* @throws SQLException

\*/

Integer getTimeout() throws SQLException;

}

默认实现ManagedTransaction另一个实现是JdbcTransaction

public class ManagedTransaction implements Transaction {

private DataSource dataSource;

private TransactionIsolationLevel level;

private Connection connection;

private final boolean closeConnection;

}

# 事务隔离级别TransactionIsolationLevel

public enum TransactionIsolationLevel {

NONE(Connection.TRANSACTION\_NONE),

READ\_COMMITTED(Connection.TRANSACTION\_READ\_COMMITTED),

READ\_UNCOMMITTED(Connection.TRANSACTION\_READ\_UNCOMMITTED),

REPEATABLE\_READ(Connection.TRANSACTION\_REPEATABLE\_READ),

SERIALIZABLE(Connection.TRANSACTION\_SERIALIZABLE);

private final int level;

TransactionIsolationLevel(int level) {

this.level = level;

}

public int getLevel() {

return level;

}

}

# 执行器Executor

public enum ExecutorType {

SIMPLE, REUSE, BATCH

}

对应的三个执行器分别是SimpleExecutor，ReuseExecutor，BatchExecutor

创建执行器代码

public Executor newExecutor(Transaction transaction, ExecutorType executorType) {

executorType = executorType == null ? defaultExecutorType : executorType;

executorType = executorType == null ? ExecutorType.SIMPLE : executorType;

Executor executor;

if (ExecutorType.BATCH == executorType) {

executor = new BatchExecutor(this, transaction);

} else if (ExecutorType.REUSE == executorType) {

executor = new ReuseExecutor(this, transaction);

} else {

executor = new SimpleExecutor(this, transaction);

}

if (cacheEnabled) {

executor = new CachingExecutor(executor);

}

executor = (Executor) interceptorChain.pluginAll(executor);

return executor;

}

其中用到了：

if (cacheEnabled) {

executor = new CachingExecutor(executor);//装饰者模式包装Excuter

}

executor = (Executor) interceptorChain.pluginAll(executor);//通过动态代理方式用到配置中的插件

## defaultExecutorType的设置

1. 通过mybatise配置文件的setting配置中的defaultExecutorType参数进行设置
2. 如果配置文件中未设置则默认使用SimpleExecutor
3. 如果opneSession中设置参数ExecutorType execType则使用相应的executor

## 三个执行器的区别

# BoundSql

public class BoundSql {

private final String sql;

private final List<ParameterMapping> parameterMappings;

private final Object parameterObject;

private final Map<String, Object> additionalParameters;

private final MetaObject metaParameters;

}

# SqlSession API

public interface SqlSession extends Closeable {

/\*\*

\* Retrieve a single row mapped from the statement key.

\* @param <T> the returned object type

\* @param statement

\* @return Mapped object

\*/

<T> T selectOne(String statement);

/\*\*

\* Retrieve a single row mapped from the statement key and parameter.

\* @param <T> the returned object type

\* @param statement Unique identifier matching the statement to use.

\* @param parameter A parameter object to pass to the statement.

\* @return Mapped object

\*/

<T> T selectOne(String statement, Object parameter);

/\*\*

\* Retrieve a list of mapped objects from the statement key and parameter.

\* @param <E> the returned list element type

\* @param statement Unique identifier matching the statement to use.

\* @return List of mapped object

\*/

<E> List<E> selectList(String statement);

/\*\*

\* Retrieve a list of mapped objects from the statement key and parameter.

\* @param <E> the returned list element type

\* @param statement Unique identifier matching the statement to use.

\* @param parameter A parameter object to pass to the statement.

\* @return List of mapped object

\*/

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

/\*\*

\* Retrieve a list of mapped objects from the statement key and parameter,

\* within the specified row bounds.

\* @param <E> the returned list element type

\* @param statement Unique identifier matching the statement to use.

\* @param parameter A parameter object to pass to the statement.

\* @param rowBounds Bounds to limit object retrieval

\* @return List of mapped object

\*/

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

/\*\*

\* The selectMap is a special case in that it is designed to convert a list

\* of results into a Map based on one of the properties in the resulting

\* objects.

\* Eg. Return a of Map[Integer,Author] for selectMap("selectAuthors","id")

\* @param <K> the returned Map keys type

\* @param <V> the returned Map values type

\* @param statement Unique identifier matching the statement to use.

\* @param mapKey The property to use as key for each value in the list.

\* @return Map containing key pair data.

\*/

<K, V> Map<K, V> selectMap(String statement, String mapKey);

/\*\*

\* The selectMap is a special case in that it is designed to convert a list

\* of results into a Map based on one of the properties in the resulting

\* objects.

\* @param <K> the returned Map keys type

\* @param <V> the returned Map values type

\* @param statement Unique identifier matching the statement to use.

\* @param parameter A parameter object to pass to the statement.

\* @param mapKey The property to use as key for each value in the list.

\* @return Map containing key pair data.

\*/

<K, V> Map<K, V> selectMap(String statement, Object parameter, String mapKey);

/\*\*

\* The selectMap is a special case in that it is designed to convert a list

\* of results into a Map based on one of the properties in the resulting

\* objects.

\* @param <K> the returned Map keys type

\* @param <V> the returned Map values type

\* @param statement Unique identifier matching the statement to use.

\* @param parameter A parameter object to pass to the statement.

\* @param mapKey The property to use as key for each value in the list.

\* @param rowBounds Bounds to limit object retrieval

\* @return Map containing key pair data.

\*/

<K, V> Map<K, V> selectMap(String statement, Object parameter, String mapKey, RowBounds rowBounds);

/\*\*

\* A Cursor offers the same results as a List, except it fetches data lazily using an Iterator.

\* @param <T> the returned cursor element type.

\* @param statement Unique identifier matching the statement to use.

\* @return Cursor of mapped objects

\*/

<T> Cursor<T> selectCursor(String statement);

/\*\*

\* A Cursor offers the same results as a List, except it fetches data lazily using an Iterator.

\* @param <T> the returned cursor element type.

\* @param statement Unique identifier matching the statement to use.

\* @param parameter A parameter object to pass to the statement.

\* @return Cursor of mapped objects

\*/

<T> Cursor<T> selectCursor(String statement, Object parameter);

/\*\*

\* A Cursor offers the same results as a List, except it fetches data lazily using an Iterator.

\* @param <T> the returned cursor element type.

\* @param statement Unique identifier matching the statement to use.

\* @param parameter A parameter object to pass to the statement.

\* @param rowBounds Bounds to limit object retrieval

\* @return Cursor of mapped objects

\*/

<T> Cursor<T> selectCursor(String statement, Object parameter, RowBounds rowBounds);

/\*\*

\* Retrieve a single row mapped from the statement key and parameter

\* using a {@code ResultHandler}.

\* @param statement Unique identifier matching the statement to use.

\* @param parameter A parameter object to pass to the statement.

\* @param handler ResultHandler that will handle each retrieved row

\*/

void select(String statement, Object parameter, ResultHandler handler);

/\*\*

\* Retrieve a single row mapped from the statement

\* using a {@code ResultHandler}.

\* @param statement Unique identifier matching the statement to use.

\* @param handler ResultHandler that will handle each retrieved row

\*/

void select(String statement, ResultHandler handler);

/\*\*

\* Retrieve a single row mapped from the statement key and parameter

\* using a {@code ResultHandler} and {@code RowBounds}.

\* @param statement Unique identifier matching the statement to use.

\* @param rowBounds RowBound instance to limit the query results

\* @param handler ResultHandler that will handle each retrieved row

\*/

void select(String statement, Object parameter, RowBounds rowBounds, ResultHandler handler);

/\*\*

\* Execute an insert statement.

\* @param statement Unique identifier matching the statement to execute.

\* @return int The number of rows affected by the insert.

\*/

int insert(String statement);

/\*\*

\* Execute an insert statement with the given parameter object. Any generated

\* autoincrement values or selectKey entries will modify the given parameter

\* object properties. Only the number of rows affected will be returned.

\* @param statement Unique identifier matching the statement to execute.

\* @param parameter A parameter object to pass to the statement.

\* @return int The number of rows affected by the insert.

\*/

int insert(String statement, Object parameter);

/\*\*

\* Execute an update statement. The number of rows affected will be returned.

\* @param statement Unique identifier matching the statement to execute.

\* @return int The number of rows affected by the update.

\*/

int update(String statement);

/\*\*

\* Execute an update statement. The number of rows affected will be returned.

\* @param statement Unique identifier matching the statement to execute.

\* @param parameter A parameter object to pass to the statement.

\* @return int The number of rows affected by the update.

\*/

int update(String statement, Object parameter);

/\*\*

\* Execute a delete statement. The number of rows affected will be returned.

\* @param statement Unique identifier matching the statement to execute.

\* @return int The number of rows affected by the delete.

\*/

int delete(String statement);

/\*\*

\* Execute a delete statement. The number of rows affected will be returned.

\* @param statement Unique identifier matching the statement to execute.

\* @param parameter A parameter object to pass to the statement.

\* @return int The number of rows affected by the delete.

\*/

int delete(String statement, Object parameter);

/\*\*

\* Flushes batch statements and commits database connection.

\* Note that database connection will not be committed if no updates/deletes/inserts were called.

\* To force the commit call {@link SqlSession#commit(boolean)}

\*/

void commit();

/\*\*

\* Flushes batch statements and commits database connection.

\* @param force forces connection commit

\*/

void commit(boolean force);

/\*\*

\* Discards pending batch statements and rolls database connection back.

\* Note that database connection will not be rolled back if no updates/deletes/inserts were called.

\* To force the rollback call {@link SqlSession#rollback(boolean)}

\*/

void rollback();

/\*\*

\* Discards pending batch statements and rolls database connection back.

\* Note that database connection will not be rolled back if no updates/deletes/inserts were called.

\* @param force forces connection rollback

\*/

void rollback(boolean force);

/\*\*

\* Flushes batch statements.

\* @return BatchResult list of updated records

\* @since 3.0.6

\*/

List<BatchResult> flushStatements();

/\*\*

\* Closes the session.

\*/

@Override

void close();

/\*\*

\* Clears local session cache.

\*/

void clearCache();

/\*\*

\* Retrieves current configuration.

\* @return Configuration

\*/

Configuration getConfiguration();

/\*\*

\* Retrieves a mapper.

\* @param <T> the mapper type

\* @param type Mapper interface class

\* @return a mapper bound to this SqlSession

\*/

<T> T getMapper(Class<T> type);

/\*\*

\* Retrieves inner database connection.

\* @return Connection

\*/

Connection getConnection();

}

## selectOne

<T> T selectOne(String statement);

<T> T selectOne(String statement, Object parameter);

实际上调用的是selectList

## selectList

真正执行逻辑

try {

MappedStatement ms = configuration.getMappedStatement(statement);

return executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO\_RESULT\_HANDLER);

} catch (Exception e) {

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

} finally {

ErrorContext.instance().reset();

}

BaseExecutor：

@Override

public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) throws SQLException {

BoundSql boundSql = ms.getBoundSql(parameter);

CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);

return query(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

@Override

public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {

ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId());

if (closed) {

throw new ExecutorException("Executor was closed.");

}

if (queryStack == 0 && ms.isFlushCacheRequired()) {

clearLocalCache();

}

List<E> list;

try {

queryStack++;

list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;

if (list != null) {

handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);

} else {

list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);

}

} finally {

queryStack--;

}

if (queryStack == 0) {

for (DeferredLoad deferredLoad : deferredLoads) {

deferredLoad.load();

}

// issue #601

deferredLoads.clear();

if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {

// issue #482

clearLocalCache();

}

}

return list;

}

## selectMap

# StatementHandler

# InterceptorChain Interceptor

public class InterceptorChain {

private final List<Interceptor> interceptors = new ArrayList<>();

public Object pluginAll(Object target) {

for (Interceptor interceptor : interceptors) {

target = interceptor.plugin(target);

}

return target;

}

public void addInterceptor(Interceptor interceptor) {

interceptors.add(interceptor);

}

public List<Interceptor> getInterceptors() {

return Collections.unmodifiableList(interceptors);

}

}

InterceptorChain使用装饰者模式

# 反射

Reflector

PropertyNamer

MethodInvoker

TypeParameterResolver

PropertyTokenizer

java反射