# 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();

}

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。

(2)

# 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

}

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

# 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();

}

## selectMap

# 反射

Reflector

PropertyNamer

MethodInvoker

TypeParameterResolver

PropertyTokenizer

java反射