# 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);

}

}

(1)解析properties

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

(2)解析settings

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

loadCustomVfs(settings);

loadCustomLogImpl(settings);

(3)解析typeAliases

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

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

(4)解析插件

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

使用到的类InterceptorChain、Interceptor

(5)解析对象工厂objectFactory

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

(6)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

(7)解析反射reflectorFactory

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

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

(8)将一般的settings属性设置到Configuration

settingsElement(settings);

(9)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;

}

(10)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

(11) 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

(12)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"));

利用MapperRegistry进行注册

# 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