# 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);

}

# API

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

a.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