文本

描述已自动生成

# Mybatis的配置文件解析

org.apache.ibatis.builder.xml.XMLConfigBuilder 开始解析xml文件

使用sax解析所以org.apache.ibatis.builder.xml.XMLMapperEntityResolver实现了dtd文件的查找可以避免请求网络。之后会调用XMLConfigBuilder的parse方法并调用parseConfiguration来正在解析配置文件mybatis-config.xml

文本

描述已自动生成

创建了了一个org.apache.ibatis.session.Configuration类，这个类存放了整个mybatis生命周期中的上下文，也就是所有的配置文件和一些默认的配置，这个配置会在很多地方使用，是核心配置类。

在Configuration的构造方法里会向TypeAliasRegistry中注册很多类的别名，其中包含默认的数据库DataSource的别名、日志别名等等，这些别名可以根据配置文件中配置的别名来获取真实对应的类。

# 初始化全局上下文Configuration

org.apache.ibatis.builder.BaseBuilder抽象的创建类（XMLConfigBuilder只是其中一种实现，还有一个非常重要的XMLMapperBuilder实现类），主要管理

Configuration : 全局上下文配置

TypeAliasRegistry : 数据类型别名和Java的类型映射表，比如string 对应的 String.class

TypeHandlerRegistry : 数据库字段类型和Java类类型类型转换器，主要是TypeHandler接口做的具体实现，这个接口通过传入的jdbc获取具体的字段并转换当前实现的类型，比如BooleanTypeHandler

XMLConfigBuilder 中执行parse方法然后调用parseConfiguration方法给已经实例好的Configuration填充配置文件mybatis-config.xml中的配置。如果使用了默认的数据源配置，会获取并自己实例化，比如使用了POOLED会使用自己实现的org.apache.ibatis.datasource.pooled.PooledDataSource作为连接池。具体看源码都配置哪些。

# Mapper文件的解析

再说BaseBuilder的另一个非常重要的子类XMLMapperBuilder。在执行parseConfiguration方法到最后的时候调用了处理mapper配置的方法mapperElement(root.evalNode("mappers"))这个方法内部实现mapper配置的读取以及mapper的xml文件的读取，读取mapper的xml文件时会新建一个XMLMapperBuilder然后调用parse方法后继续解析。同理，XMLMapperBuilder内部会调用configurationElement继续调用buildStatementFromContext方法来解析每一个真正的select等标签语句。

最后将解析到的Mapper配置的接口放到MapperRegistry中保存，并生成一个MapperProxyFactory的代理工厂，这个代理工厂会将Mapper接口的方法调用全部转发到MapperProxy类中，这个类就是Java的Proxy代理实现，最终都是在invoke方法中做实现。这就是为啥只有接口就能返回结果的原因。另外在注册到

MapperRegistry中时会解析Mapper接口类上的所有注解(包括注解方式配置SQL语句等等)。

# 解析select|insert|update|delete标签内容

上面说到BaseBuilder的子类解析Mapper文件，再看看BaseBuilder的另一个解析select|insert|update|delete 操作的子类XMLStatementBuilder，为啥会单独一个子类来执行解析呢，因为这里的解析操作属于承上启下了，一边需要解析xml文件一边需要创建MappedStatement类用于之后CURD操作时的语义解析以及所需要参数。

至此已经完成了好几个包的源码分析：

builder包：用来解析所有的配置文件并创建上下文Configuration

datasource包：包含了一个自定义的数据库连接池和JNDI和非连接池的实现

parsing包：提供SAX解析的一些工具类

binding包：提供接口的代理类实现以及接口代理类的注册获取

# Session创建过程

然后开始就是创建Session的过程

使用SqlSessionFactoryBuilder的build方法之后会创建一个SqlSessionFactory的实现。然后调用openSession方法，内部是调用了openSessionFromDataSource这个方法有三个参数：

执行器类型：SIMPLE, REUSE, BATCH三种

事务的隔离级别：就是数据库事务隔离级别

是否开启事务：默认是false也就是开启事务的

继续看看openSessionFromDataSource的代码实现：

文本

描述已自动生成

1.获取到上下文的环境配置之后然后又执行了方法getTransactionFactoryFromEnvironment获取一个TransactionFactory，这个接口有两个实现类：org.apache.ibatis.transaction.jdbc.JdbcTransactionFactory，和org.apache.ibatis.transaction.managed.ManagedTransactionFactory，这个接口是用来生产一个事务类Transaction，可以在包transaction下看到具体实现，而由工厂创建的事务本身很简单，获取一个Connection和事务的提交回滚等操作，为啥Mybatis会这么做，因为Mybatis自己并不管理事务，所以会将事务的管理交给其它容器，比如SpringTx或者JBoss服务器等等。

多有的事务管理器其实本身就是对Connection的管理，举个例子，我们可以通过一个Connection获取当前的Connection的事务是否开启，那么SpringTx在做事务的传播机制的时候只需要判断事务当前是否开启就行，所以Mybatis的一个Session中默认是开启事务的，Session关闭的时候提交事务，如果SpringTx需要事务不关闭这个时候就需要实现Spring版本的Transaction保证当前事务不关闭。

2. 接下来是通过Configuration获取一个Executor，Executor分为三种：

SimpleExecutor：就是普通的执行器，就是jdbc中的createStatement

ReuseExecutor：执行器会重用预处理语句，就是jdbc中的prepareStatement

BatchExecutor：执行器将重用语句并执行批量更新，举一反三jdbc中executeBatch

所有的实现在包executor下，具体的statement是通过StatementHandler创建的出来的。

3. 最后一步就是创建一个默认的DefaultSqlSession的SqlSession的实现类。

# 获取Mapper的实现类

代码接下来会session.getMapper(UserLoginMapper.class)获取UserLoginMapper的实例，内部是MapperRegistry中的MapperProxyFactory工厂new出来的MapperProxy类，这个类是所有的Mapper接口的实现类(使用Java的Proxy实现的)

# 开始执行查询方法

先调试看一下调用:

MapperProxy. invoke：methodCache的Map中获取当前执行的方法代理类

MapperMethod. execute：通过当前方法选择相应的select|insert|update|delete操作，select查询根据方法的返回值调用不同的查询方式

DefaultSqlSession. selectOne->selectList：从Configuration中获取当前对应的MappedStatement然后进入executor中

CachingExecutor. query：查询特有的类，也是其他的Executor实现的包装(代理类)，可以实现缓存相关的东西，这里通过BaseExecutor获取一个CacheKey，这个Key之后会用来一级缓存使用。

MappedStatement. getBoundSql：创建BoundSql对象并组建需要的参数值（这里的SqlSource接口是对每个sql语句具体的语义解析使用）

BaseExecutor. queryFromDatabase：先重置本地缓存也就是一级缓存，查询出来结果之后再添加查询后的数据到一级缓存中。

SimpleExecutor. doQuery：执行查询并返回最终结果，内部会根据Executor类型创建不同的Statement，也就是前面说的三种执行器类型：SIMPLE, REUSE, BATCH。

调用先到这里，接下来开始分析SQL语句是怎么生成的。

# BoundSql是怎么生成的

在MappedStatement中getBoundSql方法中可以看到是使用了SqlSource来获取到的，传入一个查询方法中传入的参数列表，这个参数列表会在生成sql的时候对#{}做映射，之后在jdbc的PreparedStatement中传入映射的参数下标。

SqlSource有几个实现类：

DynamicSqlSource：这个类是提供给动态sql解析用的，说直白点就是只要你的mapper中使用了比如逻辑判断之类的就会执行这个类，或者使用了${}

ProviderSqlSource：这个类是提供给注解配置的sql语句使用的

RawSqlSource：这个类是提供了类似静态sql的处理，并且会替换#{}的值

StaticSqlSource：这个类就纯粹的返回了sql本身

这些SqlSource的实现RawSqlSource和DynamicSqlSource都有一个SqlNode，这个接口实现使用了解释器模式，通过一个DynamicContext贯穿整个SqlNode树，实现类会将自己的sql片段加入到DynamicContext中去，如果有逻辑节点则通过逻辑条件使用ognl表达式执行并返回结果。

ognl表达式mybatis是直接将源码包拷贝进来了，这样就不需要额外的依赖。同样的javassist包也是直接拷贝进来的，用来在懒加载时做动态代理使用。

其中 #{} 和 ${} 是由工具类GenericTokenParser 做的解析操作。#{}的解析操作是在整个SqlNode操作完成后在类SqlSourceBuilder中将#{}替换成了?。

至此整个SQL语句的解析都已经完成了。

# 组合结果并最终返回

代码执行到DefaultResultSetHandler这个类之后剩下的就是对查询后的结果集进行组合并返回，这里做了几件事：

1. 对返回的多结果集进行处理也就是jdbc的getMoreResults来获取多结果集并进行处理
2. 创建并实例化Bean如果是懒加载则使用代理创建一个代理对象
3. 如果配置了数据库和字段驼峰低杠的转换需要先转换字段名称（单向转换）
4. 如果配置了前缀则对比的时候还要增加前缀
5. 通过storeObject对当前DefaultResultContext中的结果进行连接

调用的大致过程：

handleResultSets：处理并判断是否有多结果集，默认直接处理第一个，也可以在程序里指定resultSets对应的结果集名称，并使用handleResultSet方法执行进一步的操作

handleResultSet：判断当前结果是不是懒加载的结果，并调用handleRowValues进一步处理返回的数据

handleRowValues：判断当前是否有嵌套查询

handleRowValuesForSimpleResultMap：开始对每一行进行处理并调用getRowValue

getRowValue：创建每一行对应的映射类，并对没有映射的数据库字段处理，如果在resultMap中或者在Bean中找到了对应的字段这判断是否存在对应的Java类型处理的类TypeHandler，如果没有则什么都不做，如果有则加入处理列表，然后对创建好的映射类进行setter操作。

最后将结果返回给调用者，整个Mybatis的源码大致就是这样，有问题希望能指出来告知一下，谢谢。