# 核心配置文件

## Environments（运行环境）

### 事物管理

在mybatis的核心配置文件中（mybatis-config.xml），我们都需要配置一个或者多个运行时环境。

<environments default="default\_pooled">  
 <environment id="default\_pooled">  
 <transactionManager type="JDBC">  
 <property name="" value=""/>  
 </transactionManager>  
 <dataSource type="POOLED">  
 <property name="driver" value="com.mysql.jdbc.Driver"/>  
 <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>  
 <property name="username" value="root"/>  
 <property name="password" value="lsh159"/>  
 </dataSource>  
 </environment>  
</environments>

事物管理器：transactionManager，在Mybatis中，transactionManager提供了两个实现类：1.org.apache.ibatis.transaction.jdbc.JdbcTransaction

2.org.apache.ibatis.transaction.managed.ManagedTransaction

于是这两个实现类对应着两个工厂：org.apache.ibatis.transaction.jdbc.JdbcTransactionFactory

org.apache.ibatis.transaction.managed.ManagedTransactionFactory

* JDBC使用JdbcTransactionFactory生产JdbcTransaction对象实现，它是jdbc方式对数据库的提交和回滚进行操作。
* MANAGED使用ManagedTransactionFactory生成ManagedTransaction对象的实现。它的提交和回滚方法不用任何操作，而是把事物交给容器处理。在默认的情况下，它会关闭连接，如果我们不希望这样，我们可以设置closeConnection属性值为false来阻止它默认的关闭行为。

我们也可以自定义事物管理器：

第一步：自定义事物工厂，需要实现TransactionFactory接口。

第二步：事物实现类，实现Transaction接口。

### 数据源环境

在Mybatis中，数据库通过PooledDataSourceFactory，UnpooledDataSourceFactory，JndiDataSourceFactory这三个工厂类来提供。前两个对应于PooledDataSource，UnpooledDateSource，而JndiDataSourceFactory则会根据JNDI的信息拿到外部容器实现的数据库连接对象。

* UNPOOLED采用非数据库池的管理方式，每一次请求都会打开一个新的数据库连接，所以创建比较慢。
* <dataSource type="UNPOOLED">  
   <!--数据库驱动名-->  
   <property name="driver" value=""/>  
   <!--数据库的URL-->  
   <property name="url" value=""/>  
   <!--用户名-->  
   <property name="username" value=""/>  
   <!--密码-->  
   <property name="password" value=""/>  
   <!--默认的连接事物隔离级别-->  
   <property name="defaultTransactionIsolationLevel" value=""/>  
  </dataSource>
* POOLED数据源POOLED利用池的概念将JDBC的Connection对象组织起来，它开始会有一些空置，并且已经连接好的数据库连接，所以请求时，无须再建立和验证。处理UNPOOLED属性外，会有更多属性用来配置POOLED的数据源：

1. poolMaximumActiveConnections是在任意时间都存在的活动连接数量，默认值为10。
2. poolMaximumIdleConnections是任意时间可能存在的空闲连接数。
3. poolMaximumCheckoutTime在强制返回之前，池中连接被检出的时间
4. poolTimeToWait是一个底层设置，如果获取连接花费相当长的时间，它会给连接池打印状态信息，并重新尝试获取一个连接。
5. poolPingQuery为发送到数据库的侦测查询，用来检验连接是否处在正常工作秩序中，并准备接受请求。默认“NO PING QUERY SET”，这会导致多数据库驱动失败时带有一个恰当的错误信息。
6. poolPingEnabled为是否启用侦测查询，若开启，也必须使用一个可执行的SQL语句设置poolPingQuery属性。
7. poolPingConnectionsNotUsedFor为设置poolPingQuery的使用额度。这可以设置成匹配具体的数据连接超时时间，来避免不必要的侦测，默认值为0（即所有连接每一刻都被侦测，仅当poolPingEnabled为true时适用）

* JNDI是为了应用服务器中使用

MyBatis也支持第三方数据源，如：DBCP，c3p0等。

第一步：继承

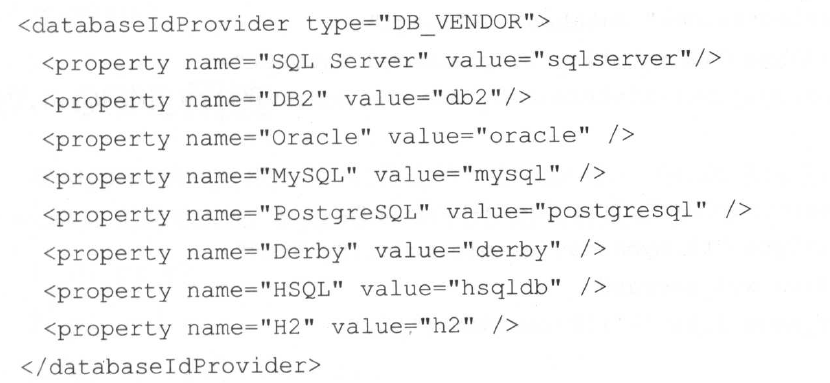
public class C3p0DataSourceFactory extends UnpooledDataSourceFactory  
{  
 protected DataSource dataSource;  
  
 public C3p0DataSourceFactory(DataSource dataSource)  
 {  
 this.dataSource = new ComboPooledDataSource() ;  
 }  
}

## 数据库厂商标识

DatabaseIdProvider元素主要是支持多种不同厂商的数据库。

<databaseIdProvider type="DB\_VENDOR">  
 <property name="Oracle" value="oracle"/>  
 <property name="MySQL" value="mysql" />  
 <property name="DB2" value="db2"></property>  
</databaseIdProvider>

如果不确定如何填写，那么可以使用JDBC创建其数据库连接对象Connection，然后通过代码connection.getMetaData().getDatabaseProductName()获取。



## Setting设置

* cacheEnabled：影响所有映射器中配置缓存的全局开关，默认为true。
* lazyLoadingEnable：延迟加载的全局开关，当开启时，所有关联对象都会延迟加载。在特定关联关系中可通过设置fetchType属性来覆盖该项的开关状态，默认为false。
* aggressiveLazyLoading：当启用时，对任意延迟属性的调用会使带有延迟加载属性的对象完整加载；反之，每种属性将会按需加载。
* autoMappingBehavior：指定MyBatis应如何自动映射列到字段或属性，NONE表示取消自动映射，PARTIAL表示只会自动映射，没有定义嵌套结果集和映射结果集。FULL会自动映射任意复杂的结果集。默认为：PARTIAL
* useGeneratedKeys：允许JDBC支持自动生成主键，需要驱动兼容。如果设置为true，则这个设置强制使用自动生成的主键，尽管一些驱动不能兼容但仍可正常工作。
* mapUnderscoreToCamelCase：是否开启自动驼峰命名规则映射，即从经典数据库列名A\_COLUMN到经典Java属性名aColumn的类似映射。
* defaultExecutorType：配置默认的执行器，SIMPLE是普通的执行器，REUSE会重用预处理语句，BATCH执行器将重用语句并执行批量更新。

## ObjectFactory（对象工厂）

当创建结果集时，MyBatis会使用一个对象工厂来完成创建这个结果集实例，在默认的情况下，MyBatis会使用其自定义的对象工厂---DefaultObjectFactory来完成对应的工作。

# 类型转换器

系统自定义类型转化器：

MyBatis中typeHandler都需要实现TypeHandler接口：

public interface TypeHandler<T> {  
  
 void setParameter(PreparedStatement ps, int i, T parameter, JdbcType jdbcType) throws SQLException;  
  
 T getResult(ResultSet rs, String columnName) throws SQLException;  
  
 T getResult(ResultSet rs, int columnIndex) throws SQLException;  
  
 T getResult(CallableStatement cs, int columnIndex) throws SQLException;  
  
}

setParameter方法是使用typeHandler通过PreparedStatement对象进行设置SQL参数的时候使用的具体方法，其中i是参数在SQL的下标，parameter是参数，JdbcType是数据库类型。

三个getResult的方法，它的作用是从JDBC结果集中获取数据进行转换，要么使用列名（columnName）要么使用下标（columnIndex）获取数据库的数据，而最后一个getResult方法是存储过程专用。

org.apache.ibatis.type.BaseTypeHandler继承了TypeHandler，我们在自定义类型转换器时可以继承BaseTypeHandler类，或者实现TypeHandler接口。

在MyBatis中使用org.apache.ibatis.type.TypeHandlerRegistry类的register方法进行类型转化器的注册。

## 枚举typeHandler

MyBatis在处理Java类型和数据库类型时，使用TypeHandler（类型处理器）对两者进行转换。MyBatis为Java和数据库JDBC中的基本类型和常用类型提供了TypeHandler的实现。

MyBatis在启动时会加载所有的JDBC对应的类型处理器，在处理枚举类型时默认使用：

org.apache.ibatis.type.EnumTypeHandler处理器，这个处理器会将枚举类型转换成字符串类型的字面值并使用，如果数据库中使用非字符长类型，则转换一定失败。

Mybatis定义两个枚举类型的支持：

public enum SexEnum  
{  
 *MALE*(1,"男"),  
 *FEMALE*(0,"女");  
 private int id ;  
 private String name;  
 SexEnum(int id ,String name)  
 {  
 this.id = id ;  
 this.name =name;  
 }  
 public SexEnum getSexById(int id)  
 {  
 for(SexEnum sex :SexEnum.*values*())  
 {  
 if(sex.getId() == id)  
 {  
 return sex ;  
 }  
 }  
 return null;  
 }  
 public int getId()  
 {  
 return id ;  
 }  
}

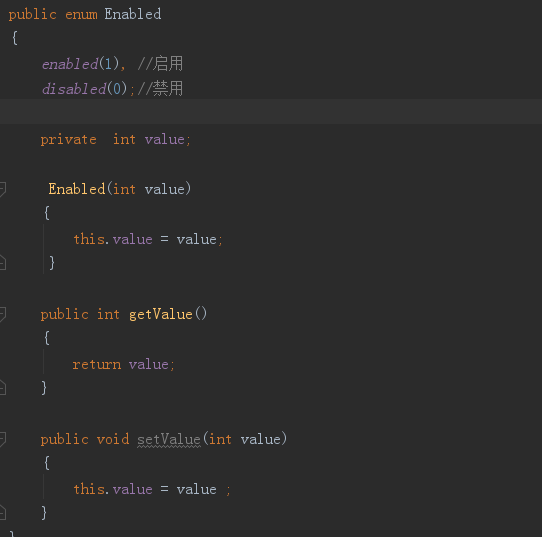
* EnumOrdinalTypeHandler，

按照Mybatis根据枚举数组下标索引的方式进行匹配的。，它要求数据库返回一个整数作为其下标。

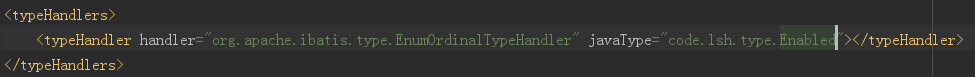
* EnumTypeHandler

会把使用的名称转化为对应的枚举。如：数据库返回的字段为“MALE”，进行Enum.valueOf(SexEnum.class,”MALE”);转换。

## 自定义枚举typeHandler



解决方法：在MyBatis配置文件中配置：



如果值既不是枚举的字面值，又不是枚举的索引值，我们可以实现自己的类型处理器：

* 实现TypeHandler接口
* 继承BaseTypeHandler类

具体实现可以参考官方已经实现的类型。

# MyBatis环境

## MyBatis核心组件

SqlSessionFactoryBuilder（构造器）：它会根据配置或者代码来生成SqlSessionFactory，采用的是分步构建的Builder模式。

SqlSessionFactory（工厂接口）：依靠它来生成SqlSession，使用的是工厂模式。

SqlSession（会话）：一个既可以发送SQL执行返回结果，也可以获取Mapper的接口，

SQL Mapper（映射器）：MyBatis新设计存在的组件，它由一个Java接口和XML文件

构成，需要给出对应的SQL和映射规则。它负责发送SQL

去执行并返回结果。

# 插件

Mybatis插件可以用来实现拦截器接口Interceptor，在实现类中对拦截器对象和方法进行处理。

package org.apache.ibatis.plugin;  
import java.util.Properties;public interface Interceptor {  
  
 Object intercept(Invocation invocation) throws Throwable;  
  
 Object plugin(Object target);  
  
 void setProperties(Properties properties);

setProperties方法，这个方法用来传递插件的参数，可以通过参数来改变插件的行为。参数的值是通过配置文件mybatis-config.xml进行设置:

<plugins>  
 <plugin interceptor="code.lsh.interceptor.XXInterceptor">  
 <property name="prop1" value="prop1"></property>  
 </plugin>  
</plugins>

plugin方法，这个方法参数target就是拦截器要拦截的对象，它的作用是给被拦截对象生成一个代理对象，方法会在创建被拦截器的接口实现类时被调用，我们只需要调用Plugin类的wrap静态方法就可以通过java的动态代理拦截目标对象。

@Override  
public Object plugin(Object target)  
{  
 return Plugin.*wrap*(target,this);  
}

public static Object wrap(Object target, Interceptor interceptor) {  
 Map<Class<?>, Set<Method>> signatureMap = *getSignatureMap*(interceptor);  
 Class<?> type = target.getClass();  
 Class<?>[] interfaces = *getAllInterfaces*(type, signatureMap);  
 if (interfaces.length > 0) {  
 return Proxy.*newProxyInstance*(  
 type.getClassLoader(),  
 interfaces,  
 new Plugin(target, interceptor, signatureMap));  
 }  
 return target;  
}

intercept方法是MyBatis运行时要执行的拦截方法，通过该方法的参数invocation可以得到很多有用的信息：

@Override  
public Object intercept(Invocation invocation) throws Throwable  
{  
 //获取目标对象  
 Object target = invocation.getTarget() ;  
 //获取被拦截的方法  
 Method method = invocation.getMethod();  
 //获取被拦截方法的参数  
 Object[] args = invocation.getArgs();  
 //执行被拦截对象的真正方法（实际调用了Method.invoke(target,args)方法）  
 Object result = invocation.proceed();  
 return result;  
}

当配置多个拦截器时，MyBatis会遍历所有拦截器，按顺序执行拦截器的plugin方法，被拦截的对象就会被层层代理。在执行拦截对象的方法时，会一层层地调用拦截器，拦截器通过invocation.proceed()方法调用下一层的方法，直到真正的方法被执行。

如：如果存在按顺序配置A，B，C三个相同签名的拦截器，MyBatis会按照C->B->A->target.proceed()->A->B->C的顺序执行。如果A，B，C的签名不相同，就会按照MyBatis拦截对象的逻辑执行。

## 拦截器签名

除了实现Interceptor接口外，还需要给实现类配置拦截器注解。

@Intercepts和@Signature：用来配置拦截器要拦截的接口的方法。

@Intercepts 注解中的属性是一个@Signature(签名)数组，可以在同一个拦截器中

同时拦截不同的接口和方法。

以拦截ResultSetHandler 接口的handleResultSets 方法为例，配置签名如下。

@Intercepts({

@Signature (

type = ResultSetHandler.class ,

method = "handleResultSets",

args = {Statement.class})

})

public class ResultSetinterceptor implements Interceptor

**@Signature 注解包含以下三个属性：**

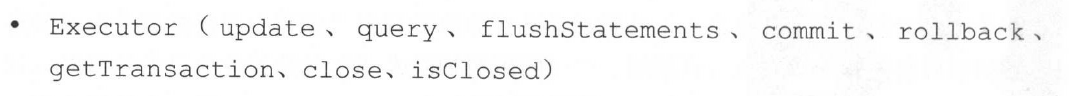
type：设置拦截的接口，可选值是前面提到的4 个接口。

Method：设置拦截接口中的方法名， 可选值是前面4 个接口对应的方法，需要和接口匹配。

args：设置拦截方法的参数类型数组， 通过方法名和参数类型可以确定唯一一个方法。

### Executor接口

Executor是一个执行器，它是一个真正执行Java和数据库交互的对象。



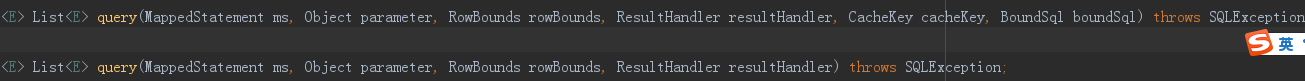
int update(MappedStatement ms, Object parameter) throws SQLException;

该方法会在INSERT，UPDATE，DELETE执行时被调用，如果想要拦截这三个操作，需要在我们实现的拦截器类加上以下注解。

@Intercepts({ @Signature(type=Executor.class,method="update",args={MappedStatement.class,Object.class})  
})

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

该方法会在所有的SELECT查询方法执行时调用，虽然接口中一个重载的方法，但是不可以被拦截。



<E> Cursor<E> queryCursor(MappedStatement ms, Object parameter, RowBounds rowBounds) throws SQLException;

当返回值为Cursor时被调用。

List<BatchResult> flushStatements() throws SQLException;

该方法只有通过SqlSession方法调用flushStatements方法或执行的接口方法中带有@Flush注解时才被调用。

void commit(boolean required) throws SQLException;

该方法在SqlSession调用commit方法时才被调用。

void rollback(boolean required) throws SQLException;

该方法在SqlSession调用rollback方法时才被调用。

Transaction getTransaction();

该方法只在通过SqlSession方法获取数据库连接时才被调用。

void close(boolean forceRollback);

该方法只在延迟加载获取新的Executor后才会被执行。

boolean isClosed();

该方法只在延迟加载执行查询方法前被执行。

### ParameterHandler接口

Object getParameterObject();

该方法只在执行存储过程处理处参的时候被调用。

void setParameters(PreparedStatement ps) throws SQLException;

该方法在所有的数据库方法设置SQL参数时被调用。

### ResultSetHandler接口

<E> List<E> handleResultSets(Statement stmt) throws SQLException;

该方法会在除存储过程及返回值类型为Cursor<T>以外的查询方法中被调用。

包装

<E> Cursor<E> handleCursorResultSets(Statement stmt) throws SQLException;

该方法只在返回值类型为Cursor<T>的查询方法中被调用。

void handleOutputParameters(CallableStatement cs) throws SQLException;

该方法只在使用存储过程处理出参时被调用。

### StatementHandler接口

StatementHandler数据库会话器，是专门处理数据库会话的

Statement prepare(Connection connection, Integer transactionTimeout)  
 throws SQLException;

该方法会在数据库执行前被调用，优先于当前接口中的其他方法而被执行。

void parameterize(Statement statement) throws SQLException;

该方法在prepare方法之后执行，用于处理参数信息。

void batch(Statement statement) throws SQLException;

在全局设置配置defaultExecutorType=”BATCH”时，执行数据操作才会调用该方法。

<E> List<E> query(Statement statement, ResultHandler resultHandler)  
 throws SQLException;

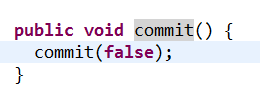
执行SELECT方法时调用。

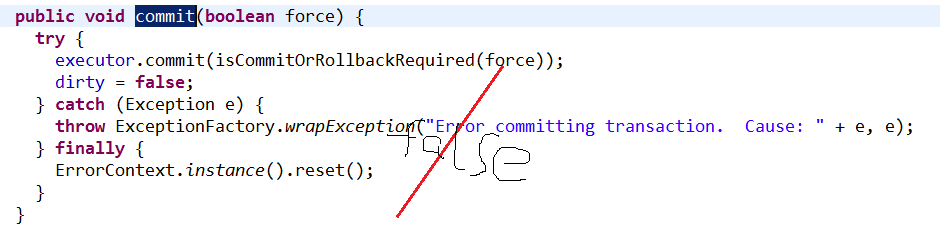
<E> Cursor<E> queryCursor(Statement statement)  
 throws SQLException;

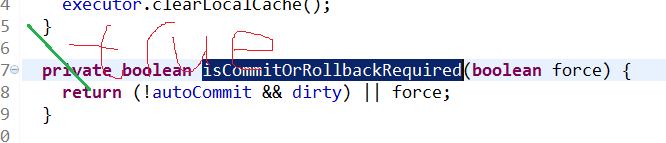
只会在返回值类型为Cursor<T>的查询中被调用。

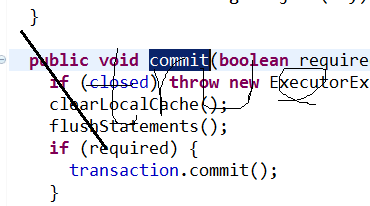
# 问题

1．调用SqlSessionFactory中的commit为什么会自动提交事物



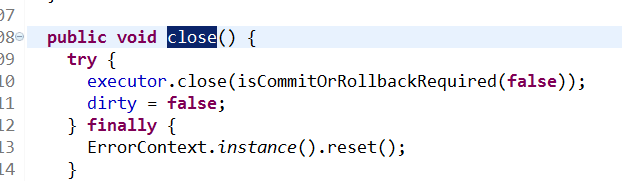


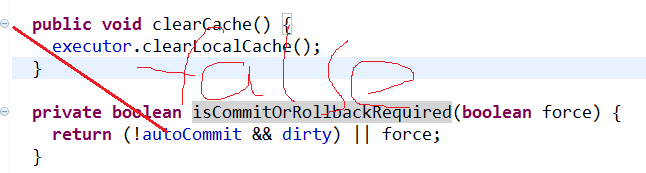


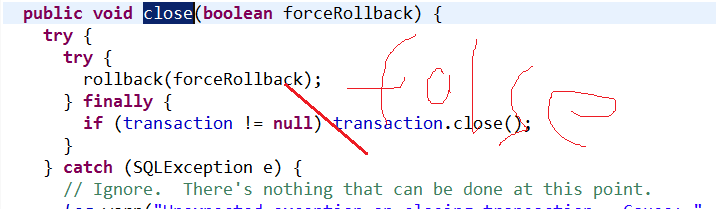


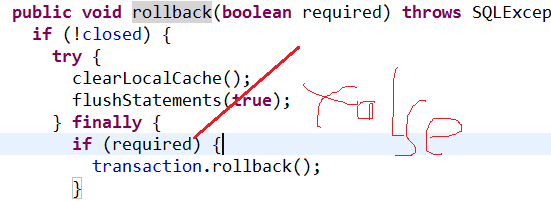
2．SqlSessionFactoy调用close为什么不需要加上事物回滚

答:如果调用了commit,则当关闭SqlSession时不进行事物回滚







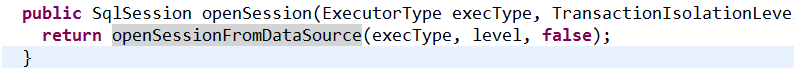


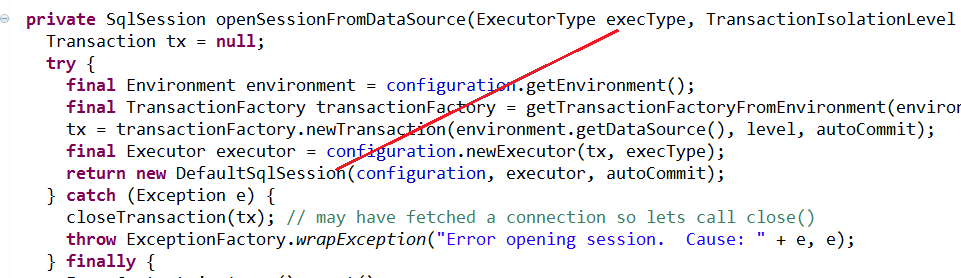
3．为什么不需要关闭IO

答：在调用build之后，框架会自动关闭

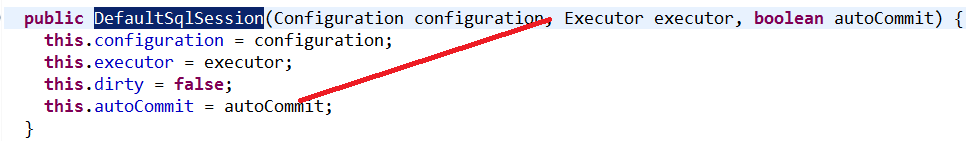
4．调用SqlSessionFacoty中openSession完成了什么工作？

答:完成成员变量的初始化





DefautlSqlSession是线程不安全的类。



5，加载Mapper文件一共有几种方式和优先级。

答：一共有四种方式：

<mapper package="" /> 优先级最高

<mapper resource=""/>

<mapper url="""/>

<mapper class=""/>优先级最低

在Xml

private void mapperElement(XNode parent) throws Exception {

if (parent != null) {

for (XNode child : parent.getChildren()) {

if ("package".equals(child.getName())) {

String mapperPackage = child.getStringAttribute("name");

configuration.addMappers(mapperPackage);

} else {

String resource = child.getStringAttribute("resource");

String url = child.getStringAttribute("url");

String mapperClass = child.getStringAttribute("class");

if (resource != null && url == null && mapperClass == null) {

ErrorContext.instance().resource(resource);

//获取Mapper文件的输入流

InputStream inputStream = Resources.getResourceAsStream(resource);

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

mapperParser.parse();

} else if (resource == null && url != null && mapperClass == null) {

ErrorContext.instance().resource(url);

InputStream inputStream = Resources.getUrlAsStream(url);

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

mapperParser.parse();

} else if (resource == null && url == null && mapperClass != null) {

Class<?> mapperInterface = Resources.classForName(mapperClass);

configuration.addMapper(mapperInterface);

} else {

throw new BuilderException("A mapper element may only specify a url, resource or class, but not more than one.");

}

}

}

}

}

