第1章 MyBatis 框架配置文件详解

1.1 typeHandlers 类型转换器

每当 MyBatis 设置参数到 PreparedStatement 或者从 ResultSet 结果集中取得值时，就会使用 TypeHandler 来处理数据库类型与 java 类型之间转换。下表描述了默认 TypeHandlers

Type Handler	Java Types	JDBC Types
BooleanTypeHandler	Boolean, boolean	任何兼容的 BOOLEAN
ByteTypeHandler	Byte, byte	任何兼容的 NUMERIC 或 BYTE
ShortTypeHandler	Short, short	任何兼容的 NUMERIC 或 SHORT INTEGER
IntegerTypeHandler	Integer, int	任何兼容的 NUMERIC 或 INTEGER
LongTypeHandler	Long, long	任何兼容的 NUMERIC 或 LONG INTEGER
FloatTypeHandler	Float, float	任何兼容的 NUMERIC 或 FLOAT
DoubleTypeHandler	Double, double	任何兼容的 NUMERIC 或 DOUBLE
BigDecimalTypeHandler	BigDecimal	任何兼容的 NUMERIC 或 DECIMAL
StringTypeHandler	String	CHAR, VARCHAR
ClobTypeHandler	String	CLOB, LONGVARCHAR
NStringTypeHandler	String	NVARCHAR, NCHAR
NClobTypeHandler	String	NCLOB
ByteArrayTypeHandler	byte[]	任何兼容的 byte stream type
BlobTypeHandler	byte[]	BLOB, LONGVARBINARY
DateTypeHandler	Date (java.util)	TIMESTAMP
DateOnlyTypeHandler	Date (java.util)	DATE
TimeOnlyTypeHandler	Date (java.util)	TIME
SqlTimestampTypeHandler	Timestamp (java.sql)	TIMESTAMP
SqlDateTypeHadler	Date (java.sql)	DATE

1.1.1 自定义类型转换器

假设表中字段是 int 类型,而实体类与之对应的属性是 boolean 类型,此时可以采用自定义类型转换器进行对应

（1） 实体类

```
public class Dept {

    private Integer deptNo;
    private String dename;
    private String loc;
    private boolean flag;
    private String country;
```

（2） 表中字段

DEPTNO	DNAME	LOC	flag
10	ACCOUNTING	NEW YORK	<NULL>
20	RESEARCH	DALLAS	<NULL>
30	SALES	CHICAGO	<NULL>
40	OPERATIONS	BOSTON	<NULL>

（3） 开发自定义类型转换器

```
public class BooleanTypeHandler implements TypeHandler {

   public Object getResult(ResultSet rs, String columnName) throws SQLException {

           int flag=rs.getInt(columnName);//1 or 0
           Boolean obj = Boolean.FALSE;
           if(flag==1){
               obj=Boolean.TRUE;
           }
       return obj;
   }


public void setParameter(PreparedStatement ps, int i, Object parameter,
       if(parameter==null){
           ps.setInt(i, 0);
           return;
       }
       Boolean flag =(Boolean)parameter;
       Integer value=flag?1:0;
       ps.setInt(i, value);
}
```

（4） 在 MyBatis 核心配置文件注册自定义类型转换器

```
<typeHandlers>
  <typeHandler
      handler="com.kaikeba.util.BooleanTypeHandler"
      javaType="Boolean" jdbcType="NUMERIC"/>
</typeHandlers>
```

（5） 在 Mapper.xml 文件中指定使用自定义类型转换器场合

```
<resultMap type="Dept" id="DeptResultMap">
 <result column="flag" property="flag"
 typeHandler="com.kaikeba.util.BooleanTypeHandler"/>
</resultMap>
```

（6） 在查询 Statement 中指定对应的 ResultMap

```
<select id="deptFindAll" resultMap="DeptResultMap">
      select * from dept
</select>
```

1.2 objectFactory 对象工厂

MyBatis 每次创建结果对象的新实例时，它都会使用一个对象工厂（ObjectFactory）实例来完成。 默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。如果想覆盖对象工厂的默认行为，则可以通过创建自己的对象工厂来实现。

1.2.1 自定义对象工厂

（1） 继承与 DefaultObjectFactory

```
public class DeptObjectFactory extends DefaultObjectFactory {

    @Override
    public Object create(Class type) {
        Dept dept = null;
        if(type==Dept.class){
            dept=(Dept)super.create(type);
            dept.setCountry("China");
            return dept;
        }else{
            return super.create(type);
        }


    }
```

继承 DefaultObjectFactory 的原因,可以利用父类中 super.create 方法帮助我们创建一个实例对象

（2） 在 MyBatis 核心文件中注册自定义工厂

```
<objectFactory type="com.kaikeba.util.DeptObjectFactory"/>
```

1.3 plugins 拦截器

拦截器的一个作用就是我们可以拦截某些方法的调用，我们可以选择在这些被拦截的方法执行前后加上某些逻辑，也可以在执行这些被拦截的方法时执行自己的逻辑而不再执行被拦截的方法。Mybatis 拦截器设计的一个初衷就是为了供用户在某些时候可以实现自己的逻辑而不必去动 Mybatis 固有的逻辑。打个比方，对于 Executor，Mybatis 中有几种实现：BatchExecutor、ReuseExecutor、SimpleExecutor 和 CachingExecutor。这个时候如果你觉得这几种实现对于 Executor 接口的 query 方法都不能满足你的要求，那怎么办呢？是要去改源码吗？当然不。我们可以建立一个 Mybatis 拦截器用于拦截 Executor 接口的 query 方法，在拦截之后实现自己的 query 方法逻辑，之后可以选择是否继续执行原来的 query 方法。

对于拦截器 Mybatis 为我们提供了一个 Interceptor 接口，通过实现该接口就可以定义我们自己的拦截器。我们先来看一下这个接口的定义：

```
public interface Interceptor {

  Object intercept(Invocation invocation) throws Throwable;

  Object plugin(Object target);

  void setProperties(Properties properties);

}
```

我们可以看到在该接口中一共定义有三个方法，intercept、plugin 和 setProperties。plugin 方法是拦截器用于封装目标对象的，通过该方法我们可以返回目标对象本身，也可以返回一个它的代理。当返回的是代理的时候我们可以对其中的方法进行拦截来调用 intercept 方法，当然也可以调用其他方法，这点将在后文讲解。setProperties 方法是用于在 Mybatis 配置文件中指定一些属性的。

定义自己的 Interceptor 最重要的是要实现 plugin 方法和 intercept 方法，在 plugin 方法中我们可以决定是否要进行拦截进而决定要返回一个什么样的目标对象。而 intercept 方法就是要进行拦截的时候要执行的方法。

对于 plugin 方法而言，其实 Mybatis 已经为我们提供了一个实现。Mybatis 中有一个叫做 Plugin 的类，里面有一个静态方法 wrap(Object target,Interceptor interceptor)，通过该方法可以决定要返回的对象是目标对象还是对应的代理。

对于实现自己的 Interceptor 而言有两个很重要的注解，一个是@Intercepts，其值是一个@Signature 数组。@Intercepts 用于表明当前的对象是一个 Interceptor，而@Signature 则表明要拦截的接口、方法以及对应的参数类型。来看一个自定义的简单 Interceptor：

```
@Intercepts({
            @Signature(method = "query", type = Executor.class, args = {
             MappedStatement.class, Object.class, RowBounds.class,
             ResultHandler.class })
           }
           )
public class MyInterceptor implements Interceptor {

    public Object intercept(Invocation invocation) throws Throwable {
          Object result = invocation.proceed();
          System.out.println("Invocation.proceed()");
          return result;
    }

    public Object plugin(Object target) {
           return Plugin.wrap(target, this);
    }
```

然后在 MyBatis 核心配置文件注册自定义拦截器

```
<plugins>
   <plugin interceptor="com.kaikeba.util.MyInterceptor"></plugin>
</plugins>
```

MyBatis 自定义拦截器，可以拦截接口只有四种.Executor.clas，StatementHandler.class,ParameterHandler.class,ResultSetHandler.class

第2章 MyBatis 框架 Mapper 配置文件详解

2.1 参数(#{参数名})

#{}实现的是向 prepareStatement 中的预处理语句中设置参数值，sql 语句中#{}表示一个占位符即?

```
<!-- 查询单个记录 -->
<select id="DeptFindByDeptNo" resultType="Dept">
   select * from  dept where deptno = #{deptNo}
</select>
```

使用#{参数名},将参数的内容添加到 sql 语句中指定位置.

如果当前 sql 语句中只有一个参数,此时参数名称可以随意定义
但是,如果当前 sql 语句有多个参数,此时参数名称应该是与当前表关联[实体类的属性名]或则[Map 集合关键字]

```
<insert id="insertDept">
   insert into dept(deptno,dname,loc)
   values
   (#{deptNo},#{dname},#{loc})
</insert>
```

上述 SQL 语句在调用时,我们可以分别采用如下两种方式输入参数

```
@Test
public void Test04(){
     Dept dept = new Dept();
     dept.setDname("金融事业部");
     dept.setLoc("北京");
     session.insert("insertDept",dept);
     session.commit();
}
```

使用#{}读取实体类对象属性内容

```
@Test
public void Test05(){
     Map map = new HashMap();
     map.put("dname", "保险事业部");
     map.put("loc", "深圳");
     session.insert("insertDept",map);
     session.commit();
}
```

使用#{}读取 map 集合中关键字的值

2.2 #{}和${}区别

在 MyBatis 中提供了两种方式读取参数的内容到 SQL 语句中,分别是
#{参数名}:实体类对象或则 Map 集合读取内容
${参数名}:实体类对象或则 Map 集合读取内容

为了能够看到两种方式的区别,需要看到 MyBatis 执行时输送的 SQL 情况.因此
需要借助 Log4J 日志进行观察

第一步: 加载 Log4j 日志工具包到项目

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

第二步:将 Log4j 配置文件添加到 src/main/resources 下

```
log4j.appender.console=org.apache.log4j.ConsoleAppender
#The Target value is System.out or System.err
log4j.appender.console.Target=System.out
#set the layout type of the apperder
log4j.appender.console.layout=org.apache.log4j.PatternLayout
#set the layout format pattern
log4j.appender.console.layout.ConversionPattern=[%-5p] %m%n

##define a logger
log4j.rootLogger=debug,console
```

接下来,我们可以查看

```
<select id="find4" resultType="Dept">
    select * from Dept
    where dname = ${dname}  and loc = #{loc}
</select>
```

输出结果

```
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Co
==>  Preparing: select * from Dept where dname = 金融事业部 and loc = ?
==> Parameters: 北京(String)
```

从这里我们可以看出两者的区别:

#{} : 采用预编译方式,可以防止 SQL 注入

${}: 采用直接赋值方式,无法阻止 SQL 注入攻击

在大多数情况下,我们都是采用#{}读取参数内容.但是在一些特殊的情况下,我们还是需要使用${}读取参数的.

比如 有两张表,分别是 emp_2017 和 emp_2018 .如果需要在查询语句中动态指定表名,就只能使用${}

```
<select>
        select *    from emp_${year}
<select>
```

再比如.需要动态的指定查询中的排序字段,此时也只能使用${}

```
<select>
        select    *    from dept order by ${name}
</select>
```

简单来说,在 JDBC 不支持使用占位符的地方,都可以使用${}

2.3 resultMap

MyBatis 框架中是根据表中字段名到实体类定位同名属性的.如果出现了实体类属性名与表中字段名不一致的情况,则无法自动进行对应.此时可以使用 resultMap 来重新建立实体类与字段名之间对应关系.

```
<resultMap type="Dept" id="DeptResultMap">
  <id column="deptno" property="departNo"/>
  <result column="dname" property="departName"/>
  <result column="loc" property="departLoc"/>
</resultMap>

<!-- 查询结果返回List或则Map -->
<select id="DeptFind" resultMap="DeptResultMap">
   select * from dept
</select>
```

2.4 sql 标签

首先,我们如下两条 SQL 映射

```
<select id="findByDeptNo" resultType="dept">
     select  *  from dept where deptno = #{deptNo}
</select>

<select id="findByDname" resultType="dept">
    select  *   from dept where dname = #{dname}
</select>
```

这两条查询映射中要查询的表以及查询的字段是完全一致的.因此可以<sql>标签将[select * from dept]进行抽取.

```
<sql id="DeptFindSql">
    select * from dept
</sql>
```

在需要使用到这个查询的地方,通过<include>标签进行引用

```
<select id="findByDeptNo" resultType="dept">
     <include refid="DeptFindSql"></include>
     where deptno = #{deptNo}
</select>

<select id="findByDname" resultType="dept">
    <include refid="DeptFindSql"></include>
    where dname = #{dname}
</select>
```

第3章 MyBatis 动态 SQL

3.1 什么是 MyBatis 动态 SQL

根据用户提供的参数,动态决定查询语句依赖的查询条件或则 SQL 语句的内容

3.2 动态 SQL 依赖标签

3.2.1 if

3.2.2 choose、when、otherwise

3.2.3 trim、where、set

3.2.4 foreach

3.3 if 使用

```
select *  from dept
where 1=1
<if test="deptNo!=null and deptNo!=0">
    and deptno = #{deptNo}
</if>
<if test="dname!=null and  dname !=''">
   and dname like '%' #{dname} '%'
</if>
<if test="loc !=null and loc !=''">
   and loc = #{loc}
</if>
```

3.4 choose、when、otherwise

类似于 Java 中的 switch case default. 只有一个条件生效,也就是只执行满足的条件 when,没有满足的条件就执行otherwise,表示默认条件

```
select * from dept
where 1=1
<choose>
   <when test="dname != null and dname != ''">
       and dname like '%' #{dname} '%'
   </when>
   <when test="loc != null and loc != ''">
       and loc = #{loc}
   </when>
   <otherwise>
      and deptno = #{deptNo}
   </otherwise>
</choose>
```

3.5 where

<where>可以自动的将第一个条件前面的逻辑运算符(or ,and)去掉

```
  select * from dept
   <where>
   <if test="deptNo!=null and deptNo!=0">
      or deptno = #{deptNo}
  </if>
  <if test="dname!=null and  dname !=''">
     and dname like '%' #{dname} '%'
  </if>
  <if test="loc !=null and loc !=''">
     and loc = #{loc}
  </if>
   </where>
```

3.6 set

会在成功拼接的条件前加上 SET 单词且最后一个”,”号会被无视掉

```
update dept

<set>
     <if test="dname!=null and  dname !=''">
             dname = #{dname},
     </if>
     <if test="loc !=null and loc !=''">
             loc  = #{loc},
     </if>
</set>

where deptno = #{deptNo}
```

3.7 trim

```
select * from dept
where
<trim prefix="" prefixOverrides="and | or " suffix="" suffixOverrides=",">
    <if test="deptNo!=null and deptNo!=0">
       and deptno = #{deptNo}
    </if>
    <if test="dname!=null and  dname !=''">
        and dname like '%' #{dname} '%'
   </if>
   <if test="loc !=null and loc !=''">
       and loc = #{loc} ,
     </if>

</trim>
```

3.8 foreach

foreach 标签用于对集合内容进行遍历,将得到内容作为 SQL 语句的一部分.
在实际开发过程中主要用于 in 语句的构建和批量添加操作

foreach 元素的属性主要有 item，index，collection，open，separator，close。

```
item表示集合中每一个元素进行迭代时的别名，
index指 定一个名字，用于表示在迭代过程中，每次迭代到的位置，
open表示该语句以什么开始，
separator表示在每次进行迭代之间以什么符号作为分隔 符，
close表示以什么结束。
```

案例 1.使用 foreach 实现批处理添加

```
<insert id="insertDept">

    insert into dept (dname,loc)
    values
    <foreach collection="list" item="dept" separator=",">
      (#{dept.dname},#{dept.loc})
    </foreach>

</insert>
```

案例 2.使用 foreach 遍历 list 集合作为查询条件

```
<!-- foreach by List -->
<select id="findForeachByList" resultType="dept">
    select  *  from dept
    where deptno in
    <foreach collection="list" item="deptno" open="(" close=")" separator=",">
         #{deptno}
    </foreach>
</select>
```

案例 3.使用 foreach 遍历数组作为查询条件

```
 <!-- foreach by Array -->
<select id="findForeachByArray" resultType="dept">
    select  *  from dept
    where deptno in
    <foreach collection="array" item="deptno" open="(" close=")" separator=",">
         #{deptno}
    </foreach>
</select>
```

案例 4.使用 foreach 遍历 Map 作为查询条件

```
 <!-- foreach by Map -->
<select id="findForeachByMap" resultType="dept">
    select  *  from dept
    where deptno in
    <foreach collection="myMap.values" item="deptno" open="(" close=")" separator=",">
         #{deptno}
    </foreach>
</select>
```

第4章 MyBatis 级联操作

在实际开发中,我们操作的表往往不是一个独立个体.它们往往根据业务依赖关系,形成一对一,一对多,多对多关联关系.为了保证业务数据的完整性.我们在操作某一张表的时候也要对与这张表关联其它表进行操作.这样的操作就成为级联操作

4.1 级联操作分类

4.1.1 级联查询

4.1.2 级联删除

4.1.3 级联更新

4.1.4 级联添加

4.2 级联查询

4.2.1 一对多查询

4.2.2 多对一查询

4.2.3 多对多查询

第5章 MyBatis 注解开发

5.1 CRUD 操作

5.1.1 @Insert

5.1.2 @Update

5.1.3 @Delete

5.1.4 @Select

```
//添加部门信息
@Insert("insert into dept(deptno,dname,loc) values(#{deptNo},#{dname},#{loc})")
public int insertDept(Dept dept);

//删除部门信息
@Delete("delete from dept where deptno= #{deptNo}")
public int deleteDept(Integer deptno);

//更新部门信息
@Update("update dept set dname=#{dname},loc=#{loc} where deptno=#{deptNo}")
public int updateDept(Dept dept);

//查询所有部门信息
@Select("select * from dept")
public List<Dept> findAllDept();

//根据条件查询部门信息
@Select("select * from dept where dname = #{departName} and loc =#{departLoc}")
public Dept findDeptByCondition(@Param("departName")String dname,@Param("departLoc")String loc);
```

5.1.5 @SelectProvider

5.1.6 @InsertProvider

5.1.7 @DeleteProvider

5.1.8 @UpdateProvider