Mybatis动态sql

题目难度：★★★

知识点标签：Mybatis动态sql

学习时长：20分钟

题目描述

Mybatis动态sql是做什么的？都有哪些动态sql？能简述一下动态sql的执行原理吗？

解题思路

> 面试官问题可以从几个方面来回答：动态sql的应用、具体有哪些动态sql、动态sql执行原理

动态sql的应用

动态 SQL 是 MyBatis 的强大特性之一。如果你使用过 JDBC 或其它类似的框架，你应该能理解根据不同条件拼接 SQL 语句有多痛苦，例如拼接时要确保不能忘记添加必要的空格，还要注意去掉列表最后一个列名的逗号。利用动态 SQL，可以彻底摆脱这种痛苦。

使用动态 SQL 并非一件易事，但借助可用于任何 SQL 映射语句中的强大的动态 SQL 语言，MyBatis 显著地提升了这一特性的易用性。

如果你之前用过 JSTL 或任何基于类 XML 语言的文本处理器，你对动态 SQL 元素可能会感觉似曾相识。在 MyBatis 之前的版本中，需要花时间了解大量的元素。借助功能强大的基于 OGNL 的表达式，MyBatis 3 替换了之前的大部分元素，大大精简了元素种类，现在要学习的元素种类比原来的一半还要少。

具体有哪些动态sql

- if

  使用动态 SQL 最常见情景是根据条件包含 where 子句的一部分。比如：

```
<select id="findActiveBlogWithTitleLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title = #{title}
  </if>
</select>
```

这条语句提供了可选的查找文本功能。如果不传入 “title”，那么所有处于 “ACTIVE” 状态的 BLOG 都会返回；如果传入了 “title” 参数，那么就会对 “title” 一列进行匹配查找并返回对应的 BLOG 结果。

- choose , when, otherwise

  有时候，我们不想使用所有的条件，而只是想从多个条件中选择一个使用。针对这种情况，MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

  还是上面的例子，但是策略变为：传入了 “title” 就按 “title” 查找，传入了 “author” 就按 “author” 查找的情形。若两者都没有传入，就返回标记为 featured 的 BLOG（这可能是管理员认为，与其返回大量的无意义随机 Blog，还不如返回一些由管理员挑选的 Blog）。

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

- trim, where, set

  如果没有匹配的条件会怎么样？最终这条 SQL 会变成这样：

```
SELECT * FROM BLOG
WHERE
```

  这会导致查询失败。如果匹配的只是第二个条件又会怎样？这条 SQL 会是这样:

```
SELECT * FROM BLOG
WHERE
AND title like ‘someTitle’
```

  这个查询也会失败。这个问题不能简单地用条件元素来解决。这个问题是如此的难以解决，以至于解决过的人不会再想碰到这种问题。

  MyBatis 有一个简单且适合大多数场景的解决办法。而在其他场景中，可以对其进行自定义以符合需求。而这，只需要一处简单的改动：

```
<select id="findActiveBlogLike"
     resultType="Blog">
```

```
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

where 元素只会在子元素返回任何内容的情况下才插入 “WHERE” 子句。而且，若子句的开头为 “AND” 或 “OR”，*where* 元素也会将它们去除。

如果 *where* 元素与你期望的不太一样，你也可以通过自定义 trim 元素来定制 *where* 元素的功能。比如，和 *where* 元素等价的自定义 trim 元素为：

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>
```

prefixOverrides 属性会忽略通过管道符分隔的文本序列（注意此例中的空格是必要的）。上述例子会移除所有 *prefixOverrides* 属性中指定的内容，并且插入 *prefix* 属性中指定的内容。

用于动态更新语句的类似解决方案叫做 *set*。*set* 元素可以用于动态包含需要更新的列，忽略其它不更新的列。比如：

```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

这个例子中，*set* 元素会动态地在行首插入 SET 关键字，并会删掉额外的逗号（这些逗号是在使用条件语句给列赋值时引入的）。

来看看与 *set* 元素等价的自定义 *trim* 元素吧：

```
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```

注意，我们覆盖了后缀值设置，并且自定义了前缀值。

- foreach

  动态 SQL 的另一个常见使用场景是对集合进行遍历（尤其是在构建 IN 条件语句的时候）。比如：

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in
  <foreach item="item" index="index" collection="list"
      open="(" separator="," close=")">
        #{item}
  </foreach>
</select>
```

  foreach 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的集合项（item）和索引（index）变量。它也允许你指定开头与结尾的字符串以及集合项迭代之间的分隔符。这个元素也不会错误地添加多余的分隔符，看它多智能！

动态sql执行原理

1. SqlResource
   该接口含义是作为sql对象的来源，通过该接口可以获取sql对象。其唯一的实现类是XmlSqlResource，表示通过xml文件生成sql对象。
2. Sql
   该接口可以生成sql语句和获取sql相关的上下文环境(如ParameterMap、ResultMap等)，有三个实现类: RawSql表示为原生的sql语句，在初始化即可确定sql语句；SimpleDynamicSql表示简单的动态sql，即sql语句中参数通过$property$方式指定，参数在sql生成过程中会被替换，不作为sql执行参数；DynamicSql表示动态sql，即sql描述文件中包含isNotNull、isGreaterThan等条件标签。
3. SqlChild
   该接口表示sql抽象语法树的一个节点，包含sql语句的片段信息。该接口有两个实现类: SqlTag表示动态sql片段，即配置文件中的一个动态标签，内含动态sql属性值(如prepend、property值等)；SqlText表示静态sql片段，即为原生的sql语句。每条动态sql通过SqlTag和SqlText构成相应的抽象语法树。
4. SqlTagHandler
   该接口表示SqlTag(即不同的动态标签)对应的处理方式。比如实现类IsEmptyTagHandler用于处理标签，IsEqualTagHandler用于处理标签等。
5. SqlTagContext
   用于解释sql抽象语法树时使用的上下文环境。通过解释语法树每个节点，将生成的sql存入SqlTagContext。最终通过SqlTagContext获取完整的sql语句。

总结

从设计上看，dynamic sql的实现主要涉及三个模式:

- 解释器模式: 初始化过程中构建出抽象语法树，请求处理时根据参数对象解释语法树，生成sql语

句。

- 工厂模式：为动态标签的处理方式创建工厂类(SqlTagHandlerFactory)，根据标签名称获取对应的处理方式。
- 策略模式: 将动态标签处理方式抽象为接口，针对不同标签有相应的实现类。解释抽象语法树时，定义统一的解释流程，再调用标签对应的处理方式完成解释中的各个子环节。

最后，以一张图总结动态sql的实现原理: