MyBatis中${}与#{}的区别

# $与#的区别

总结：

### #{param}与${param}的预编译处理机制不同，#{param}在预处理时将参数部分用一个占位符?代替；而${param}只是简单的字符串替换，在动态解析阶段就直接拼接成字符。

### #{param}参数替换发生在DBMS中，而${param}的参数替换发生在MyBatis的动态解析过程；

**动态 sql** 是 mybatis 的主要特性之一，在 mapper 中定义的参数传到 xml 中之后，在查询之前 **mybatis 会对其进行动态解析**。mybatis 为我们提供了两种支持动态 sql 的语法：**#{}以及${}**。

在下面的语句中，如果 username 的值为 zhangsan，则两种方式无任何区别：

select \* from user where name = **#{name}**;

select \* from user where name = **${name}**;

　　其解析之后的结果均为 select \* from user where name = **'zhangsan';**

## 区别之一：预编译处理不同

　　但是 **#{} 和 ${} 在预编译中的处理是不一样的**。#{} 在预处理时，会把参数部分用**一个占位符 ?** 代替，变成如下的 **sql 语句**：

select \* from user where name = **?**;

　　而 ${} 则只是**简单的字符串替换**，在动态解析阶段，该 sql 语句会被解析成

select \* from user where name = **'zhangsan'**;

## 区别之二：参数替换时期不同

　　以上，**#{} 的参数替换是发生在 DBMS 中，而 ${} 则发生在动态解析过程中。**

　　那么，在使用过程中我们应该使用哪种方式呢？

　　答案是，**优先使用 #{}**。因为 ${} 会导致 **sql 注入**的问题。

$将传入的数据直接显示生成在sql中，那么我们使用 ${}的时候:

select \* from user where name = ${name};

**${}在动态解析的时候，会将我们传入的参数当做String字符串填充到我们的语句中**，就会变成下面的语句: select \* from user where name = "dato";

**预编译之前的 SQL 语句**已经不包含变量了，完全已经是常量数据了。相当于我们普通没有变量的sql了。

**#{} 在动态解析的时候**， 会解析成一个**参数标记符**。

select \* from user where name = **#{name}**;

解析之后的语句是： select \* from user where name = **?**;

**${}**接受**从用户输出的内容并提供给语句中不变的字符串**，这样做是不安全的。这会导致潜在的SQL注入攻击。

${}接收输入参数，类型可以是简单类型，pojo、hashmap。

如果接收简单类型，${}中只能写成value。#{}中可以写成value或其它名称。

## MyBatis中$和#的区别

#{param} 代表param是属性值，map里面的key或者是你的pojo对象里面的属性, MyBatis会自动在它的**外面加上引号**,表现在sql语句是这样的 where xxx = 'value'。

#{},用于传递参数,相当于**一个占位符**,可以很大程度上防止sql注入;

${},用于sql的拼接,常用语group by,order by等不能使用占位符的语句中;

## 引号问题

#{param}将传入的参数值都当成**字符串**来解析，**会对自动传入的数据加一个双引号**。如：order by #{user\_id}，如果传入的值是111,那么解析成sql时的值为order by "111", 如果传入的值是id，则解析成的sql为order by "id"；<错误>

$将传入的数据直接显示生成在sql中。如：order by $user\_id$，如果传入的值是111,那么解析成sql时的值为order by 111, 如果传入的值是id，则解析成的sql为order by id.

## SQL注入问题

**#{param}方式**能够很大程度上防止SQL注入，但是**${param}方式**无法防止SQL注入。

如验证用户名时通过**t.NAME = ${abc or 1=1}**就可以实现sql注入的目的。

## 使用场景区别

使用原则：**能用#的就别用$；能用#的就别用$；能用#的就别用$；**

**知道了那些场景下必须使用$即可：**

**${param}**方式一般用于传入数据库对象，如传入数据库表名、列名；

${},用于SQL语句的拼接,常用于group by、order by、like '%${dctName}%'等不能使用占位符的语句中;

一般**模糊查询的时候**使用$, dct\_name like '%${dctName}%'。

因为order by **${param} 若传入id,编译为order by id;但是，**order by **${param}编译为order by 'id'是错误的。**

如果插入数据库是Date类型，则要用#，mybatis会把输入的Date类型自动装换为字符串类型。

ps:在使用mybatis中还遇到<![CDATA[]]>的用法，在该符号内的语句，将不会被当成字符串来处理，而是直接当成sql语句，比如要执行一个存储过程。

## 错误情况

用${}传入数据直接显示在生成的sql中，如上面的语句，用role\_id = ${roleId,jdbcType=INTEGER},那么sql在解析的时候值为role\_id = 100，执行时会报错;应该为role\_id = '100'

**${param}传递的参数会被当成sql语句中的一部分，比如传递表名，字段名**

例子：(传入值为id)

**order by ${param}**

则解析成的sql为：order by id

**#{parm}传入的数据都当成一个字符串，会对自动传入的数据加一个双引号**。

例子：(传入值为id)

select \* from table where name = #{param}

则解析成的sql为：select \* from table where name = **"id"**

防止SQL注入：

为了安全，**能用#的地方就用#方式传参**，这样可以有效的防止sql注入攻击

# 使用示例：

用法:

**#{}的用法：**

<if test="condition.userId != null and condition.userId != ''">

AND user\_id = **#{**condition.userId ,jdbcType=VARCHAR}

</if>

**${}的用法：**

<select id="get" parameterType="java.util.Map" resultMap="test">

SELECT

<include refid="sql\_column\_list" />

FROM test

<trim prefix="WHERE" prefixOverrides="AND">

<include refid="sql\_condition" />

</trim>

<if test="orderBy != null">

ORDER BY **${orderBy}**

</if>

LIMIT 1

</select>

# MyBatis中$的作用及使用规则

在MyBatis中，”**${xxx}**”这样格式的参数会直接参与SQL编译，从而不能避免注入攻击。但涉及到**动态表名和列名**时，**只能使用“${xxx}”**这样的参数格式。

# sql注入简介

直接上了**百度**的例子，感觉一看就清晰明了

某个网站的登录验证的SQL查询代码为：

strSQL = "SELECT \* FROM users WHERE (name = '" + userName + "') and (pw = '"+ passWord +"');"

恶意填入

userName = "1' OR '1'='1"; 与passWord = "1' OR '1'='1";

时，将导致原本的SQL字符串被填为

strSQL = "SELECT \* FROM users WHERE (**name = '1' OR '1'='1') and (pw = '1' OR '1'='1')**;"

也就是实际上运行的SQL命令会变成下面这样的

strSQL = "SELECT \* FROM users;"

这样在后台帐号验证的时候巧妙地绕过了检验，达到无账号密码，亦可登录网站。所以SQL注入攻击被俗称为**黑客的填空游戏**。

看下面的例子：

select \* from **${tableName}** where name = **#{name}**

　　在这个例子中，如果表名为

**user; delete user; --**

　　则动态解析之后 sql 如下：select \* from **user; delete user; --** where name = **?**;

　　--之后的语句被注释掉，而原本**查询用户的语句**变成了**查询所有用户信息+删除用户表的语句**，会对数据库造成重大损伤，极大可能导致服务器宕机。

**但是表名用参数传递进来的时候，只能使用 ${}** ，具体原因可以自己做个猜测，去验证。这也提醒我们在这种用法中要小心sql注入的问题。

# mybatis中#{}与${}的差别（如何防止sql注入）

默认情况下,使用#{}语法,MyBatis会产生PreparedStatement语句中，并且安全的设置PreparedStatement参数，这个过程中MyBatis会进行必要的安全检查和转义。

#相当于对数据 加上 双引号，$相当于直接显示数据

示例1：

执行SQL：select \* from emp where name = #{employeeName}

参数：employeeName=>Smith

解析后执行的SQL：select \* from emp where name = ？

示例2：

执行SQL：select \* from emp where name = ${employeeName}

参数：employeeName传入值为：Smith

解析后执行的SQL：Select \* from emp where name =Smith

综上所述、${}方式会引发SQL注入的问题、同时也会影响SQL语句的预编译，所以从安全性和性能的角度出发，能使用#{}的情况下就不要使用${}

Q:但是${}在什么情况下使用呢？

A:有时候可能需要直接插入一个不做任何修改的字符串到SQL语句中。这时候应该使用${}语法。

　　比如，动态SQL中的字段名，如：ORDER BY ${columnName}

重要：接受从用户输出的内容并提供给语句中不变的字符串，这样做是不安全的。这会导致潜在的SQL注入攻击，因此你不应该允许用户输入这些字段，或者通常自行转义并检查。

MyBatis框架作为一款半自动化的持久层框架，其SQL语句都要我们自己手动编写，这个时候当然需要防止SQL注入。其实，MyBatis的SQL是一个具有“输入+输出”的功能，类似于函数的结构，如下：

<select id="getBlogById" resultType="Blog" parameterType=”int”>

SELECT id,title,author,content

FROM blog

WHERE id=#{id}

</select>

　　这里，parameterType表示了输入的参数类型，resultType表示了输出的参数类型。回应上文，如果我们想防止SQL注入，理所当然地要在输入参数上下功夫。上面代码中黄色高亮即输入参数在SQL中拼接的部分，传入参数后，打印出执行的SQL语句，会看到SQL是这样的：

SELECT id,title,author,content FROM blog WHERE id = ?

　　不管输入什么参数，打印出的SQL都是这样的。这是因为MyBatis启用了预编译功能，在SQL执行前，会先将上面的SQL发送给数据库进行编译；执行时，直接使用编译好的SQL，替换占位符“?”就可以了。因为SQL注入只能对编译过程起作用，所以这样的方式就很好地避免了SQL注入的问题。

　　【底层实现原理】MyBatis是如何做到SQL预编译的呢？其实在框架底层，是JDBC中的PreparedStatement类在起作用，PreparedStatement是我们很熟悉的Statement的子类，它的对象包含了编译好的SQL语句。这种“准备好”的方式不仅能提高安全性，而且在多次执行同一个SQL时，能够提高效率。原因是SQL已编译好，再次执行时无需再编译。

　　在MyBatis中，${xxx}这样格式的参数会直接参与SQL编译，从而不能避免注入攻击。但涉及到动态表名和列名时，只能使用${xxx}这样的参数格式。所以，这样的参数需要我们在代码中手工进行处理来防止注入。

【结论】

在编写MyBatis的映射语句时，尽量采用#{xxx}这样的格式。若不得不使用${xxx}这样的参数，要手工地做好过滤工作，来防止SQL注入攻击。

#{}：相当于JDBC中的PreparedStatement

${}：是输出变量的值

简单说，#{}是经过预编译的，是安全的；${}是未经过预编译的，仅仅是取变量的值，是非安全的，存在SQL注入。

如果我们order by语句后用了${}，那么不做任何处理的时候是存在SQL注入危险的。你说怎么防止，那我只能悲惨的告诉你，你得手动处理过滤一下输入的内容。如判断一下输入的参数的长度是否正常（注入语句一般很长）

原理介绍：

myBatis 对于#{} 在boundSql 的时候，会将sql 语句解析成为？；对于${} 则本来的值进行替换。

对于 select \* from emp where name = #{employeeName}，myBatis 会解析sql为：select \* from emp where name = #{employeeName}，然后进行prepareStatment 预编译处理。

对于 select \* from emp where name = ${employeeName}，myBatis 会解析sql为：select \* from emp where name = 'employeeName'，然后进行prepareStatment 预编译处理。

myBatis 其实底层防止sql注入，使用的是prepareStatment语句。表现为#{employeeName}和${employeeName}的两种参数注入方法。

a专区 数据库专区

MyBatis mapper文件中的变量引用方式#{}与${}的差别

默认情况下,使用#{}语法,MyBatis会产生PreparedStatement语句中，并且安全的设置PreparedStatement参数，这个过程中MyBatis会进行必要的安全检查和转义。

示例1：

执行SQL：Select \* from emp where name = #{employeeName}

参数：employeeName=>Smith

解析后执行的SQL：Select \* from emp where name = ？

执行SQL：Select \* from emp where name = ${employeeName}

参数：employeeName传入值为：Smith

解析后执行的SQL：Select \* from emp where name =Smith

综上所述、${}方式会引发SQL注入的问题、同时也会影响SQL语句的预编译，所以从安全性和性能的角度出发，能使用#{}的情况下就不要使用${}

但是${}在什么情况下使用呢？

有时候可能需要直接插入一个不做任何修改的字符串到SQL语句中。这时候应该使用${}语法。

比如，动态SQL中的字段名，如：ORDER BY ${columnName}

注意：当使用${}参数作为字段名或表名时、需指定statementType为“STATEMENT”，如：

<select id="queryMetaList" resultType="Map" statementType="STATEMENT">Select \* from emp where name = ${employeeName} ORDER BY ${columnName}</select>

# MyBatis动态传入表名，字段名参数的解决办法

要实现动态传入**表名、列名**，需要做如下修改

添加属性statementType="STATEMENT"

同时sql里的属有变量取值都改成${xxxx}，而不是#{xxx}

<delete id="deleteTableData" parameterType="java.util.Map" statementType="STATEMENT">

<![CDATA[

delete from ${table} where

${col} < ${date}]]>

</delete>

1. statementType：STATEMENT（非预编译），PREPARED（预编译）或CALLABLE中的任意一个，这就告诉 MyBatis 分别使用Statement，PreparedStatement或者CallableStatement。默认：PREPARED。这里显然不能使用预编译，要改成非预编译。

2. ${xxxx}：$将传入的数据直接显示生成在sql中，对于字符串数据，需要手动加上引号。

String dateStr = DateFormatUtils.format(date.getTime(), "yyyy-MM-dd HH:mm:ss");

dateStr = "'" + dateStr + "'";

因为parameterType这种种类型在执行sql语句之前先进行了编译，而没有表名的查询语句编译是无法通过的，所以要用执行时才进行编译的类型"STATEMENT"。

分表：自动创建表格

<update id="createNewTable" statementType="STATEMENT">

CREATE TABLE ${tableName} (

ID int(8) NOT NULL AUTO\_INCREMENT,

NAME varchar(12) NOT NULL,

OPEN\_ID varchar(50) NOT NULL,

NICK\_NAME varchar(50) NOT NULL,

SEX int(1) NOT NULL

</update>

使用了statementType="STATEMENT"字段也实现了传入表名进行建表。

2.异常捕获，触发建表

Map<String, Object> condition = new HashMap<String, Object>();

condition.put("tableName", "user\_info");

condition.put("id", 1);

try {

UserInfo userInfo = userInfoSrv.testSelect(condition);

} catch (BadSqlGrammarException e) {

logger.error("Table not exist,now creat new table!", e);

**int gt = userInfoSrv.createNewTable(condition);**

}

当捕获异常BadSqlGrammarException时自动建表，当然这个异常不仅仅是标示表不存在的异常，还有其他的情况，所以其他可能触发建表的异常都要提前用代码排除掉，以上就是我解决这个问题的全部思路。