大厂Java面试题：从源码的角度分析MyBatis中#{}与${}的区别

大家好，我是[王有志](http://cxyroad.com/ ”https://www.yuque.com/wangyouzhi-u3woi/cr7d5y/uw8c5iyvpgnqpzmg”)。

今天我会通过源码来分析一道**京东，联储证券和爱奇艺**都考察过的 MyBatis 面试题：MyBatis 中“#{}”和“${}”有什么区别？是否可以使用“#{}”来传递 order by 的动态列？

“#{}”和“${}”有什么区别？

“#{}”在 MyBaits 中表示一个占位符，MyBatis 在解析 Mapper 文件的 SQL 语句时会将“#{}”的部分替换成占位符“?”，执行时使用 JDBC 编程中的 PreparedStatement 预编译语句，相当于如下代码：

```
Connection connection = DriverManager.getConnection(”jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8”, ”root”, ”123456”);

String sql = ”select * from user where user_id = ?”;
preparedStatement = connection.prepareStatement(sql);
preparedStatement.setInt(1, 1);
```

需要注意的是，在 PreparedStatement 中使用占位符时 Java 会明确要求数据类型，因此 MyBatis 在处理通过“#{}”设置的参数时会进行转义，这种方式带来的优点是，在 MyBatis 中使用“#{}”会有效的避免 SQL 注入，提高系统的安全性。

“${}”在 MyBatis 中表示字符串拼接，不会进行转移，无论传入的参数为哪种类型，最后都会被作字符串类型拼接到 SQL 语句中，这会导致传入特殊字符串时，SQL 注入的风险大大增加。

可以使用“#{}”**来传递 order by 语句的动态列吗？**

先说结论：**可以使用“#{}”传递参数，但是没有任何效果，如果想要起到 order by 效果，必须使用“${}”传递参数**。

使用“#{}”传递参数，MyBatis 会进行转义，当传入的类型为字符串类型时，最后设置到 SQL 语句中的参数前后会添加单引号，我们有如下的 Mapper 接口的方法：

```
List<UserDO> selectAll(@Param(”orderColumn”) String orderColumn);
```

Mapper 接口对应的 MyBatis 映射器为：

```
<select id=”selectAll” resultType=”com.wyz.entity.UserDO”>
  select * from user order by #{orderColumn} desc
</select>
```

为其编写测试方法：

```
@Test
public void testSelectAll() {
  Reader mysqlReader = Resources.getResourceAsReader(”mybatis-config.xml”);
  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(mysqlReader);
  SqlSession sqlSession = sqlSessionFactory.openSession();
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
  List<UserDO> users = userMapper.selectAll(”user_id”);
  log.info(”查询到的数据：{}”, JSON.toJSONString(users));
}
```

执行测试方法后，发现查询到的结果并没有按照“user_id”字段进行倒序排序，我们看到 MyBatis 的打印日志，映射器中#{orderColumn}被替换成了占位符“?”：

```
==>  Preparing: select * from user order by ? desc
==> Parameters: user_id(String)
```

而在最终执行时 Java 会使用转义后的 orderColumn 参数替换掉占位符，最终执行的 SQL 语句如下：

```
select * from user order by 'user_id' desc
```

使用“${}”传递参数，是直接将传递的参数拼接到 SQL 语句中，我们修改 `UserMapper#selectAll`方法的 SQL 语句映射，使用“${}”传递参数：

```
<select id="selectAll" resultType="com.wyz.entity.UserDO">
  select * from user order by ${orderColumn} desc
</select>
```

同样的，我们执行测试代码，可以看到控制台输出的 SQL 语句直接将参数“id”拼接到了 SQL 语句当中，并且查询结果也符合我们的预期。

```
==>  Preparing: select * from user order by user_id desc
==> Parameters:
```

因此，**当我们希望通过参数传递参与 order by 排序的字段时，我们应该使用“${}”去传递参数，同理，对于查询字段，以及条件语句中，如果想要动态的拼

接字段，我们都应该使用“${}”传递参数。**

SQL 注入案例

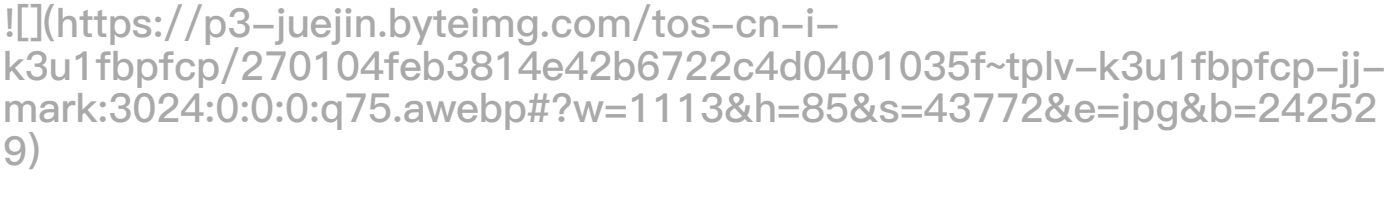
在互联网的早期时代，网站对于用户的密码存储并不规范，且登录权限校验并不完善，非常容易出现 SQL 注入的情况。例如，我们有如下用户表：

```
create table user (
  user_id      int          not null comment '用户Id' primary key,
  account      varchar(30) not null comment '账户',
  password     varchar(30) not null comment '密码',
  name         varchar(50) not null comment '用户名',
  age          int          not null comment '年龄',
  gender       varchar(50) not null comment '性别',
  id_type      int          not null comment '证件类型',
  id_number    varchar(50) not null comment '证件号'
);
```

在登录验证中，只做了账户和密码的非空验证后，就直接使用账户和密码查询用户信息，并且在 MyBatis 映射器中使用了“${}”来拼接参数，如下：

```
<select id="selectUserByPassword"
resultType="com.wyz.entity.UserDO">
  select * from user where account = ${account} and password =
${password}
</select>
```

我们为 user 表中插入两条数据：



接着我们来写一段测试代码：

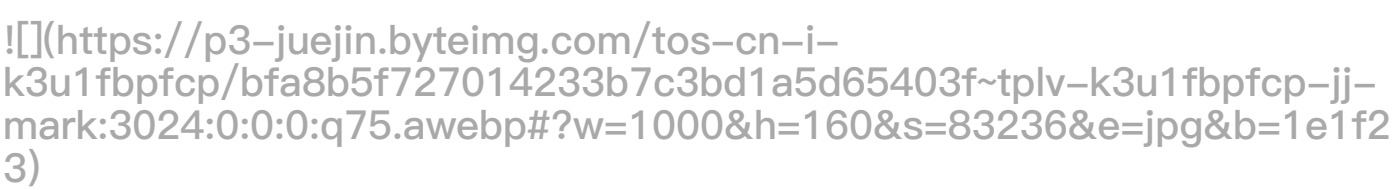
```
@Test
public void testSelectUserByPassword() {
  SqlSession sqlSession = sqlSessionFactory.openSession();
  UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

  String account = "'root' or 1 = 1";
  String password = "'大傻瓜'";
  UserDO user = userMapper.selectUserByPassword(account, password);

  System.out.println("root 用户信息：" + JSON.toJSONString(user));

  sqlSession.close();
}
```

执行这段测试代码后，可以看到控制台输出了账户为 root 的用户信息：



可以看到，最后生成的 SQL 语句是：

```
select * from user where account = 'root' or 1 = 1 and password = '大傻瓜'：
```

无论密码输入什么，都可以被轻松的绕过去，这就是 SQL 注入攻击。

源码分析

MyBatis 中 SQL 语句的处理可以分为两个部分：

1. MyBatis 构建 SqlSessionFactory 时，读取所有映射器文件，将 SQL 语句信息加载到 MyBatis 的运行环境中；
2. MyBatis 执行映射器方法时，查找 MyBatis 运行环境中的 SQL 语句，替换参数并执行 SQL 语句。

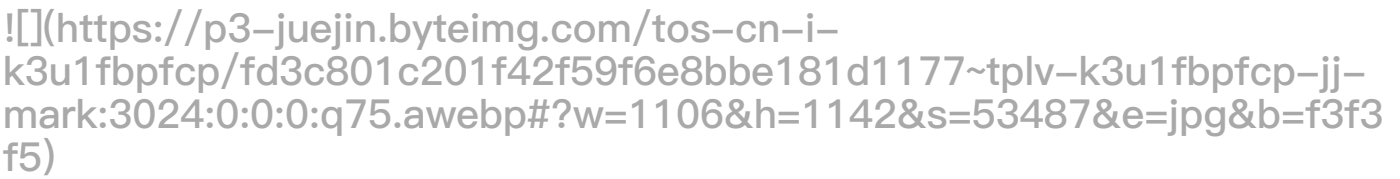
其中第一部分的 SQL 处理中不涉及到参数处理，只做映射器中 SQL 语句的配置解析，因此我们直接来看第二部分，MyBatis 在执行 SQl 语句时的处理流程。

我们写一条同时使用含“#{}”和“${}”的 SQL 语句：

```
<select id="selectById" resultType="com.wyz.entity.UserDO">
  select * from user where user_id = #{userId} order by ${orderColumn}
desc
</select>
```

这里自行补充下对应的接口方法以及测试案例即可，下面我们通过这句 SQL 语句，来看下 MyBatis 是如何处理“#{}”和“${}”的。

由于 MyBatis 是默认开启缓存的，因此在没有特殊配置的时候，MyBatis 的默认使用的执行器是 CachingExecutor，我们通过一张调用流程图来看下 CachingExecutor 是如何处理 SQL 语句的。



我们重点`DynamicSqlSource#getBoundSql`方法，源码如下：

```
public BoundSql getBoundSql(Object parameterObject) {
  DynamicContext context = new DynamicContext(configuration,
```

```
parameterObject);
  // 处理通过 ${} 设置的参数，直接拼接到 SQL 语句中
  rootSqlNode.apply(context);
  SqlSourceBuilder sqlSourceParser = new
SqlSourceBuilder(configuration);
  Class<?> parameterType = parameterObject == null ? Object.class :
parameterObject.getClass();
  // 处理通过 #{} 设置的参数，将其替换为占位符 ?
  SqlSource sqlSource = sqlSourceParser.parse(context.getSql(),
parameterType, context.getBindings());
  BoundSql boundSql = sqlSource.getBoundSql(parameterObject);
  context.getBindings().forEach(boundSql::setAdditionalParameter);
  return boundSql;
}
```

其中第 5 行调用的`rootSqlNode.apply(context)`会处理通过“${}”方式设置的参数，将参数值直接拼接到 SQL 语句中；而第 9 行中调用的`SqlSource sqlSource = sqlSourceParser.parse(context.getSql(), parameterType, context.getBindings())`会处理通过“#{}”方式设置的参数，将其替换为占位符“?”。

处理通过“${}”设置的参数

我们接着`rootSqlNode.apply(context)`这行代码往下看，这段调用的`MixedSqlNode#apply`方法，源码如下：

```
public boolean apply(DynamicContext context) {
  contents.forEach(node -> node.apply(context));
  return true;
}
```

注意，这里的入参 context 与 MixedSqlNode 的成员变量 contents 是不同的：

* context，存储传递到 SQL 语句中的参数；
* contents，存储 MyBatis 映射器中的 SQL 语句（未处理参数的）。

接着来看 forEach 中调用的`node.apply(context)`，由于我们的 SQL 语句非常简单，这里会调用到 SqlNode 的实现类 TextSqlNode 的 apply 方法中，源码如下：

```
public boolean apply(DynamicContext context) {
  GenericTokenParser parser = createParser(new
BindingTokenParser(context, injectionFilter));
  context.appendSql(parser.parse(text));
  return true;
}
```

首先是第 2 行构建 GenericTokenParser 实例对象的方法：

```
public class TextSqlNode implements SqlNode {
  private GenericTokenParser createParser(TokenHandler handler) {
    return new GenericTokenParser("${", "}", handler);
  }
}

public class GenericTokenParser {
  public GenericTokenParser(String openToken, String closeToken,
TokenHandler handler) {
    this.openToken = openToken;
    this.closeToken = closeToken;
    this.handler = handler;
  }
}
```

这段代码非常简单，我们可以清晰的看到创建的 GenericTokenParser 实例对象是用于处理“${}”的，这里需要注意，GenericTokenParser 实例对象中的成员变量 handler 中包含 context 对象，而该对象中存储着 SQL 语句的参数数据。

下面我们来看第 3 行调用的`parser.parse(text)`方法，该方法的部分源码如下：

```
public String parse(String text) {
// 省略部分代码
// 获取“${”在SQL语句中的位置
int start = text.indexOf(openToken);
// 省略部分代码
char[] src = text.toCharArray();
int offset = 0;
final StringBuilder builder = new StringBuilder();
StringBuilder expression = null;
do {
if (start > 0 && src[start - 1] == '\\') {
// 省略部分代码
} else {
if (expression == null) {
expression = new StringBuilder();
}
// 省略部分代码
builder.append(src, offset, start - offset);
offset = start + openToken.length();
// 获取与“${”对应的“}”在 SQL 语句中的位置，注意这里的 offset
int end = text.indexOf(closeToken, offset);
while (end > -1) {
if ((end <= offset) || (src[end - 1] != '\\')) {
expression.append(src, offset, end - offset);
break;
}
// 省略部分代码
}
if (end == -1) {
// 省略部分代码
} else {
// 拼接通过“${}”传入的参数
builder.append(handler.handleToken(expression.toString()));
offset = end + closeToken.length();
}
}
start = text.indexOf(openToken, offset);
} while (start > -1);
if (offset < src.length) {
builder.append(src, offset, src.length - offset);
}
return builder.toString();
}

```

代码看起来很复杂，但实际上就做了一件事，反映到我们在源码分析开篇中的 SQL 语句中，就是将`${orderColumn}`替换为传入的参数。这段代码的复杂性主要来自于 SQL 语句中可能包含多个“”，处理时需要考虑到“{}”，处理时需要考虑到“”，处理时需要考虑到“{”与“}”的对应关系。

最后来看下第 33 行中调用的`handler.handleToken(expression.toString())`方法，需要注意这里的 handler 是 BindingTokenParser 的实例对象，源码如下：

```
public String handleToken(String content) {
Object parameter = context.getBindings().get("_parameter");
if (parameter == null) {
context.getBindings().put("value", null);
} else if (SimpleTypeRegistry.isSimpleType(parameter.getClass())) {
context.getBindings().put("value", parameter);
}
Object value = OgnlCache.getValue(content, context.getBindings());
String srtValue = value == null ? "" : String.valueOf(value);
checkInjection(srtValue);
return srtValue;
}
```

这段代码不难理解，方法的入参是“${}”中间的参数名，反映到开篇的 SQL 语句中，即是参数名“orderColumn”。
`BindingTokenParser#handleToken`**方法的核心作用是通过参数名在 context 对象中查找与之对应的参数值，需要特别注意的是，这里查找到参数值后直接“暴力”的将值的类型转换为了 String 后返回**，并直接通过`StringBuilder#append`直接拼接到 SQL 语句中。

处理通过“#{}”设置的参数

我们来看`SqlSourceBuilder#parser`方法的源码：

```
public SqlSource parse(String originalSql, Class<?> parameterType,
Map<String, Object> additionalParameters) {
ParameterMappingTokenHandler handler = new
ParameterMappingTokenHandler(configuration, parameterType,
```

```
additionalParameters);
GenericTokenParser parser = new GenericTokenParser("#{", "}",
handler);
String sql;
if (configuration.isShrinkWhitespacesInSql()) {
sql = parser.parse(removeExtraWhitespaces(originalSql));
} else {
sql = parser.parse(originalSql);
}
return new StaticSqlSource(configuration, sql,
handler.getParameterMappings());
}
```

这段代码会调用到第 8 行的`parser.parse(originalSql)`，该方法与之前我们看到的 parser 方法类似，这里我们就不展示其源码，我们直接找到 handleToken 方法，这里调用的是`SqlSourceBuilder#handleToken`方法，源码如下：

```
public String handleToken(String content) {
  parameterMappings.add(buildParameterMapping(content));
  return "?";
}
```

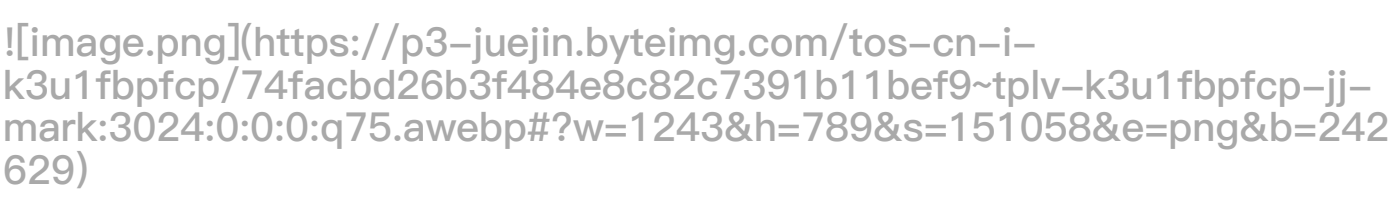
该方法有两个作用：

1. 第 2 行调用的buildParameterMapping(content)用于处理参数信息，获取该参数的 Java 类型，JDBC 类型以及映射关系等；
2. 第 3 行直接返回了占位符“?”，用于拼接到 SQL 语句中。

实际上到了这里 SQL 语句中使用到的“#{}”都已经被设置成了占位符“?”，例如，开头的的 SQL 语句实际上已经被转换成了如下的内容：

```
select * from user where user_id = ? order by user_id desc
```

通过断点调试源码也可以看到，在执行到`sql = parser.parse(originalSql)`时，originalSql 中的 SQL 语句还是未处理的`#{userId}`，而调用该方法之后返回的 SQL 已经将 originalSql 中的#{userId}替换为了占位符“?”。



好了，今天的内容就到这里了，如果本文对你有帮助的话，希望多多点赞支持，如果文章中出现任何错误，还请批评指正。**最后欢迎大家分享硬核 Java 技术的金融摸鱼侠**[王有志](http://cxyroad.com/”https://www.yuque.com/wangyouzhi-u3woi/wvkm9u/uw8c5iyvpgnqpzmg?singleDoc”)，我们下次再见！

原文链接: https://juejin.cn/post/7361261846089924658