目录

[一、 实现思路 2](#_Toc10132947)

[1.1 问题分析 2](#_Toc10132948)

[1.2 JavaCC实现分析 2](#_Toc10132949)

[1.3 正则匹配+递归实现 2](#_Toc10132950)

[1.4 结果集大小分析 3](#_Toc10132951)

[二、 实现分析 3](#_Toc10132952)

[2.1 规则存储数据结构 3](#_Toc10132953)

[2.2 核心类与方法 4](#_Toc10132954)

[三、 总结与思考 5](#_Toc10132955)

[3.1 总结 5](#_Toc10132956)

[3.2 思考 6](#_Toc10132957)

# 实现思路

## 问题分析

读完题目后，可以知道问题的输入为若干行自定义的语法规则，输出为符合规则的字符串，且能够无限生成。很明显需要先解析规则，然后定义相关函数实现规则间的元素组合，最终生成SQL语句。问题可分为两步：规则解析和规则间元素组合方式。

该题刚开始时想到了两种做法，JavaCC实现和正则匹配+递归两种做法。本文选择第二种实现方式。

## JavaCC实现分析

根据编译原理的相关知识，规则首先需要完成词法和语法解析，其中JavaCC是一个用Java开发的能生成词法和语法分析器的工具，它可以读取上下文无关且有特殊意义语法的文本输入并转换成可识别该语法的Java程序。

词法分析器就是将规则的每一行解析为一个个记号（Token），分析得到的记号后工语法分析使用，词法分析器是低级别的分析，就是将一系列字符分成一个个的Token，并标记Token的分类。

语法分析的任务就是在词法分析的基础上将单词序列组合成各类语法短语，判断输入在结构上是否正确，JavaCC采用自顶向下的分析方法，但因为没有回溯功能所以需要程序员自己解决冲突。

对于单行规则中的alt中可能包含其他elem，这是典型的左递归（如文法中有形如A→Aα的产生式），所以需要使用最左推导，对于分析过程可能陷入死循环的问题，可以使用通过显出左递归来实现。

由于对JavaCC中jj文件的解析规则有些淡忘，所以本文并未用JavaCC实现，虽然JavaCC可以完成更加复杂的规则解析，但本文的问题可以有相对简单的实现方式。

## 正则匹配+递归实现

本题的规则由一行query和若干行elem组成，而且query和elem的构成均符合相应规则，如query的表达式由若干个由双引号括住的字符串（strlit）和若干个独立的elem构成且由空格符分割，elem的表达式之间则是用“|”分割，“|”之间可以为由双引号括住的字符串、独立的elem或者**由strlit和独立的elem组成的混合alt**，我们将第二种情况称为**简单规则**，第三种情况称为**复合规则**，所以从规则解析来说，正则表达式能够解析本题的大多数规则情况。

当规则解析完成后，为了生成符合规则的SQL，需要根据query的规则对elem进行组合，如果每个elem全部由strlit构成，则只需在elem中的strlit随机选取一个即可，但alt中可以包含elem，且该elem可展开，这就意味着一个elem1中的strlit还是一个elem2，从elem2中获取strlit时可能获取到elem3……。为了解决这个问题，我们可以使用递归的方式来实现elem间的跳转，当从elem1中取出elem2时，递归的去elem2中获取元素，如果获取的是strlit，直接返回，如果是elem3，则递归的获取elem3中的元素，直到elem-n返回的是strlit即可。虽然存在一种极端，即每次的随机选取过程选中的都是elem，整个选取过程在所有的elem中轮转且一直无法选出strlit，这种情况取决于rule之间的关系和rule中elem所占的比例。

复合规则的处理和上述方法类似，如果复合规则是某个简单规则的一个alt，则它是有可能被随机选中的，当被选中后，我们需要将复合规则的内容逐个拼接，这种方法非常类似于重新处理一个query。

## 结果集大小分析

经过上述分析，**对于所有规则中的alt仅仅有两种可能的情况：strlit或者elem时，不包含复合规则，它的结果集大小是可以估算的**，分析如下：

1. 该问题的目的是能够无限生成符合规则的字符串，但是给定规则后，SQL的结果集大小基本就已经确定，分析案例的结果集大小：

query: "select " col op lit " from t;"

col: "c1" | "c2" | "c3"

op: "+" | "-"

lit: "65536" | "'2019'" | "'x'"

query中有3个elem，其中col有三个候选项，op有2个，lit有3个，所以非重复的SQL总数=3\*2\*3=18。

1. 如果修改案例如下（新增内容加粗显示）：

query: "select " col op lit " from t;"

col: "c1" | "c2" | "c3" | **lit**

op: "+" | "-"

lit: "65536" | "'2019'" | "'x'" | **col**

该情况，col和lit产生了相互依赖，所以计算elem的初始候选项大小时忽略elem：

**最终候选项大小=初始候选项大小+所有elem初始候选项大小**

分析：

1. col初始候选项大小=3
2. lit初始化大小=3
3. op最终候选项大小=2
4. col最终候选项大小= 3+lit初始候选项大小 = 3 + 3 = 6
5. lit最终候选项大小= 3+col初始候选项大小 = 3 + 3 = 6
6. **结果集大小= 6\*3\*6 = 108**

对于存在复合规则的情况，它的结果集是不确定的，最极端情况下可以存在无数多种，所以这里暂不分析复合规则的结果集大小。

# 实现分析

经过上面的分析后，需要选择合适的数据结构来保存规则，以及query表达式可能设计非常多的elem，且elem可能存在嵌套的情况，所以需要非常关注代码性能，如递归的中止条件和多线程生成SQL语句。接着分析本题解法中较重要的数据结构和方法

## 2.1 规则存储数据结构

程序中仅使用了5个重要的私有变量，分别保存query表达式和规则内容。

在实现时，我们定义了全局的rulesMap，用于保存所有规则（包括复合规则）的k，v映射，**这两种规则的k都是规则名，v都是组成它们的alt动态数组，但区别是复合规则会将它所有的alt进行拼接，而简单规则只是从alt中随机选取一个。**我们还定义了两个Set集合，simpleRuleNameSet和complexRuleNameSet，它们分别存储简单规则和复合规则的规则名。

*// 定义规则列表***private** List<String> **ruleList** = **new** ArrayList<>();  
  
*// 定义第一行查询后半部分规则内容***private** List<String> **queryExp** = **new** ArrayList<>();  
  
*// 定义除了第一行之外所有行的规则名：规则内容之间的映射（包括简单规则和复合规则）  
// 备注：  
// 简单规则名为单行规则开头的elem名称  
// 复合规则名为单行规则中的alt，且该alt为多个elem或者strlit组合而成  
// 例如cols: col | col ", " cols中的cols和col即为简单规则名；“col "," cols”即为复合规则名；***private** Map<String, ArrayList<String>> **rulesMap** = **new** HashMap<>();  
  
*// 保存所有简单规则的规则名***private** Set<String> **simpleRuleNameSet** = **new** HashSet<>();  
  
*// 保存所有复合规则的规则名***private** Set<String> **complexRuleNameSet** = **new** HashSet<>();

## 2.2 核心类与方法

程序的入口为generateSQL，只需要传递文件名和待生成的SQL条数即可，它会先调用init完成规则语法检查和相关数据结构的初始化，然后默认创建10个线程，将SQL条数均分到10个线程上，可以提高生成SQl的生成速度。

1. 下面代码为主程序入口：

**public static void** main(String[] args) {  
 SQLGen sg = **new** SQLGen();  
 **long** startTime=System.*currentTimeMillis*();  
 sg.generateSQL(**"conf\\second.case"**, 20000);  
 **long** endTime=System.*currentTimeMillis*();  
 *logger*.info(String.*format*(**"程序运行时间：%d ms"**,(endTime - startTime)));  
}

1. 下面为多线程生成SQL的核心代码，它包括两个部分，run函数和SQL生成函数。**其中默认开启10个线程执行，多线程能极大的提高SQL生成速度。**  
   @Override  
   **public void** run() {  
    *// 该线程执行制定次数，生成制定数量条SQL* **for** (**int** t = 1; t <= **times**; ++t) {  
    String result = getSubString(**queryExp**);  
    System.***out***.println(result);  
    }  
    **latch**.countDown();  
   }

*// 传入query表达式***public** String getSubString(List<String> subQueryExp) {  
 **int** queryExpLen = subQueryExp.size();  
 String subExp = **""**;  
 **int** subExpLen = 0;  
 StringBuffer sbTmp = **new** StringBuffer();  
 **for** (**int** i = 0; i < queryExpLen; ++i) {  
 subExp = subQueryExp.get(i);  
 subExpLen = subExp.length();  
  
 *// 字符串，处理后返回* **if** (isStrLegal(subExp))  
 sbTmp.append(subExpLen > 2 ?  
 subExp.substring(1, subExpLen - 1).replaceAll(**"\\\\"**, **""**) : **""**);  
 *// 简单规则名，递归拼接* **else if** (**simpleRuleNameSet**.contains(subExp))  
 sbTmp.append(simpleRecursion(subExp));  
 *// 复合规则名，递归拼接（一般query中不存在复合规则名）* **else if** (**complexRuleNameSet**.contains(subExp))  
 sbTmp.append(complexRecursion(subExp));  
 *// 属于未定义的规则名，直接结束程序* **else** {  
 *logger*.error(String.*format*(**"[%s] has not been defined!"**, subExp));  
 System.*exit*(-1);  
 }  
 }  
 **return** sbTmp.toString();  
}

1. 我们定义了两个递归函数，它们是处理规则拼接的核心函数，其中simpleRecursion会随机从它的alt动态数组中随机选取一个，可能会碰到strlit，简单规则和复合规则，所以有三种处理情况，compleRecursion同理。这里有个细节需要注意，**java.util.Random是多线程安全的，所以它产生随机数的速度慢，而ThreadLocalRandom是专门为多线程生成随机数准备的，它的生成速度快。**

*// 简单规则名, 随机从规则列表中选择一个***public** String simpleRecursion(String subExp) {  
 ArrayList<String> subElemList = **rulesMap**.get(subExp);  
 **int** subElemSize = subElemList.size();  
  
 *// 多线程并发生成随机数* **int** randomIdx = ThreadLocalRandom.*current*().nextInt(subElemSize);  
 String val = subElemList.get(randomIdx);  
  
 *// 当前alt已经是strlit了，处理后返回* **if** (isStrLegal(val))  
 **return** val.length() > 2 ?  
 val.substring(1, val.length() - 1).replaceAll(**"\\\\"**, **""**) : **""**;  
  
 *// 当前alt是简单规则名，继续递归获取，直到val为strlit时终止* **if** (**simpleRuleNameSet**.contains(val))  
 **return** simpleRecursion(val);  
  
 *// 当前alt是复合规则名，调用复合规则函数* **if** (**complexRuleNameSet**.contains(val))  
 **return** complexRecursion(val);  
  
 **return** val;  
}

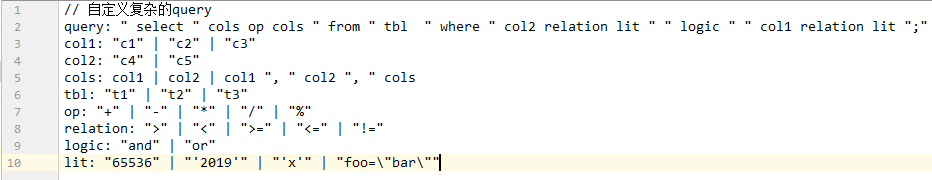
*// 复杂规则名，从头到尾依次拼接***public** String complexRecursion(String subElemName) {  
 ArrayList<String> subElemList = **rulesMap**.get(subElemName);  
 **int** subElemSize = subElemList.size();  
 StringBuffer sb = **new** StringBuffer();  
 **for** (**int** i = 0; i < subElemSize; ++i) {  
 String tmp = subElemList.get(i);  
  
 **if** (isStrLegal(tmp))  
 sb.append(tmp.length() > 2 ?  
 tmp.substring(1, tmp.length() - 1).replaceAll(**"\\\\"**, **""**) : **""**);  
  
 *// 当前alt是简单规则名，继续递归获取，直到val为strlit时终止* **if** (**simpleRuleNameSet**.contains(tmp))  
 sb.append(simpleRecursion(tmp));  
  
 *// 当前alt是复合规则名，调用复合规则函数  
 // 无论是否存在复合规则名嵌套复合规则名的情况，这里都支持* **if** (**complexRuleNameSet**.contains(tmp))  
 sb.append(complexRecursion(tmp));  
 }  
 **return** sb.toString();  
}

# 总结与思考

## 3.1 总结

### 3.1.1 多线程效率分析

为了分析多线程带来的性能提升，我们构造了一个复杂的测试用例，有八个简单规则和一组复合规则，分别对比了生成四个数量级在单线程和多线程情况下的时间，下图为测试规则：



当数据量很大时无论是将SQL语句输出到文件中还是输出到终端上显示，都会带来极大的IO开销，我们将打印语句注释掉，默认它们不消耗时间，下图为SQL生成时间对比：



可以看出，当生成的SQL量很大的时候，多线程带来的效率提升明显，这里仅测试了10个线程情况下的时间，具体开启多少线程，性能最优本文并未测试。

### 3.1.2 方法优缺点

本文使用了正则表达式+递归的方法实现，在实现起来较为容易，本文将规则分为简单规则和复合规则，且两种规则在实现思路上都是基于递归的技术，思路清晰，即使构造更加复杂的规则，递归在很大程度上依旧可以解决。

当然本文的方法还存在许多不足，其中最明显的不足就是规则解析方面，由于正则表达式书写难度大，匹配规则复杂，所以对于query或者alt字符串中嵌套字符串支持上存在一定缺陷，如无法正确发现下列书写错误：

query: “select ” “\”col\”” “ from ” t1;

query: “select ” “\”col2” “ from ” t2;

这是由于双引号匹配时存在歧义，且本人在正则的掌握上存在疏漏。

也许本文在其他性能优化或其他细节上还存在问题，但本人已尽力优化了代码。

## 3.2 思考

由于本文是一个相对简单的SQL生成工具，对于更加复杂的SQL生成器应该满足能够支持哪些功能呢？我个人认为应该包含以下若干方面：

1. 灵活的数据特征生成，如指定属性列上的数据特征指定（数值型：最大最小值、字符型：平均长度、最大长度、基数（非重复值个数））。
2. 丰富的数据访问分布，如唯一值分布、随机分布、高斯分布和zipfian分布等。
3. 灵活的多表间join的SQL语句自动生成？
4. 如何根据规则生成合法的SQL？

其中第1、2两项对于数据库的功能和性能都有很大影响，尤其是性能测试，不同的数据特征直接影响分布式事务操作的结果集大小以及结果集的遍历时间，影响二级索引的效率等；不同的访问分布与内存缓存关系密切，高Skew的缓存效果要比均匀分布好。

第三点任务相对于前两者更加复杂，它需要考虑每个Schema间的主外键关系，对于混合主键的情况相比于单主键情况更加多样化，但是多表join的SQL自动化生成能极高的节约测试成本。

如何生成合法的SQL语句相比于本题研究的生成无限规则字符串更加具有意义，如果生成的SQL语句中有大量的无效SQL则他们在SQL语法解析的时候就会被过滤掉，这极大的浪费了数据库资源，所以如何生成合法的SQL也具有挑战性。