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，“|”之间只能为由双引号括住的字符串或者为独立的elem。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。

## 结果集大小分析

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**

# 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。

*// 定义规则列表***private** List<String> **ruleList** = **new** ArrayList<String>();  
*// 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***private** Map<String, ArrayList<String>> **rulesMap** = **new** HashMap<>();  
*// 保存所有规则的规则名***private** Set<String> **ruleNames** = **rulesMap**.keySet();  
*// 定义生成随机数的类*Random **randSeed** = **new** Random();

## 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. 下面的代码为递归生成strlit的方法，他会在规则的候选项中随机选择一个，如果选中的依旧是elem，则递归执行，该递归的结束条件为获取的候选项为strlit。

**public** String recursion(String subExp)  
{  
 *// 获取该规则名对应的规则内容* ArrayList<String> elemExp = **rulesMap**.get(subExp);  
 **int** expLen = elemExp.size();  
 **int** random = **randSeed**.nextInt(expLen);  
 String val = elemExp.get(random);  
 *// 如果这里取到的值还是elem，则继续递归获取，直到val为strlit时才终止* **if** (**ruleNames**.contains(val))  
 **return** recursion(val);  
 **if** (val.length() > 2)  
 **return** val.substring(1, val.length() - 1).replaceAll(**"\\\\"**, **""**);  
 **else  
 return ""**;  
}

1. 下面的代码为多线程运行的核心代码，它遍历query表达式并以此判断每个候选项是否为strlit还是elem，如果是strlit直接拼接在sb后，如果是elem调用递归函数，这里有个细节，如果两者都不是即表明该**候选项存在问题**，它会打印错误日志并阻塞当前线程等待父线程检测到时，终止程序。

**for** (**int** t = 1; t <= **times**; ++ t) {  
 StringBuilder sb = **new** StringBuilder();  
 *// 逐个遍历query中的表达式* **for** (**int** i = 0; i < queryExpLen; ++i) {  
 subExp = **queryExp**.get(i);  
 subExpLen = subExp.length();  
 **if** (isStrLegal(subExp))  
 {  
 **if** (subExpLen == 2)  
 sb.append(**""**);  
 **else** sb.append(subExp.substring(1, subExpLen - 1));  
  
 }  
 **else if** (**ruleNames**.contains(subExp))  
 sb.append(recursion(subExp));  
 **else** {  
 *logger*.error(String.*format*(**"Illegal subExp \"%s\" in query!"**, subExp));  
 Thread.*currentThread*().interrupt();  
 }  
 }  
 System.***out***.println(sb.toString());  
}

1. 这段代码负责开启多个线程加快SQL生成速度，其中主线程会sleep 10毫秒（这个时间其实会受query表达式遍历速度的影响），如果这10毫秒内子线程发出阻塞信号，即可被主线程检测到，从而终止程序。

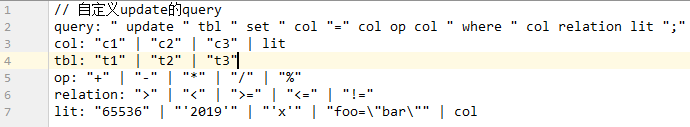
*// 这里默认线程数为10***int** threadSize = (sqlCount > 10) ? 10 : 1;  
CountDownLatch latch = **new** CountDownLatch(threadSize);  
**int** times = sqlCount / threadSize;  
**for** (**int** i = 0; i < threadSize; i++)  
{  
 Thread thread = **new** Thread(**new** multiGen(latch, times));  
 thread.start();  
 **try** {  
 Thread.*sleep*(10);  
 **if** (thread.isInterrupted())  
 **return**;  
 }**catch** (Exception e)  
 {  
 e.printStackTrace();  
 }  
}

# 总结与思考

## 3.1 总结

本次作业其实难度并不是很大，但是需要考虑的细节很多，如用正则表达式匹配时，无法匹配所有复杂的情况，如本文的解法就无法匹配在query表达式的双引号中加入反斜杠（如“select \”student\” ”）的语法，一方面是正则表达式能力有限，另一方面是不能精通。

此外，本文加入了CountDownLatch，来统计SQL的生成时间，下表为针对second.case的规则分别统计了不同数量SQL的生成时间对比：



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生成时间对对：



由上图可知，当数据量增大后可以发现多线程极大的提高了SQL的生成时间，但是具体开启多少线程可带来最大的性能提升，本文并未测试。

## 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。