# 基于ANTLR4的python式缩进语法分析器的实现

-本文系“stuPyd教学编程语言”项目开发过程中产生的成果文档之一，一方面旨在针对目前中国国内对ANTLR4的中文资料缺乏和相对外网应用尚未成熟的境况提供给各位开发者一个成熟的参考案例；另一方面，由于使用java做开发的代码已经比较成熟，因此我们使用python语言，为使用其他语言的各位程序员提供思路。

-开发工具：ANTLR4 PyCharm插件；Python 3.7.0解释器；PyCharm Professional

-未经允许禁止转载。

1. **杂项**

在<GitHub.com>上，国外同行已经根据Python Document中给出的文法使用ANTLR4制作了python的语法分析器。在本文所涉及的部分（仅指缩进语法的实现一块）代码逻辑皆参考于此。网址在<https://github.com/antlr/grammars-v4/blob/master/python3/Python3.g4> 。然而一方面ANTLR支持不同语言（包括C++和Python2和Python3），目前仅在java语言上技术较为成熟；另一方面这方面的中文资料少得可怜，并无搬运。因此本文对源码作出分析并改写为Python，同时详细介绍ANTLR4所生成的词法分析器（Lexer）和语法分析器（Parser）以及相关的类（Token，CommonToken）。

本文中实现代码放在最后，而将ANTLR源码解析放在前部且占大篇幅，主要原因是网上代码示例有余而原理解析不足，以致笔者在自己实现时遇到了很多困惑，同时由于目标语言不同时ANTLR生成的方法名、成员名也有所不同，导致代码可移植性不强，这也是撰写此文的原因之一，希望读到此文的读者能对此有更深刻的了解。

关于如何在IDE里使用ANTLR插件，以及ANTLR的基础语法请搜素其他网站。我们将可能在后续博文中给出教学，本文暂不涉及。

1. **内置函数和ANTLR分析器运行机制**

本段主要讲解在完成缩进文法过程中需要用到的类和方法，均为改写Lexer所必须。ANTLR在生成代码时，类和方法的名字多半不会改变，但由于语言特性的原因多少会有些不同。Java用户请参考前文中的源码，C++用户则请按本文介绍的思路和顺序阅读源码做出改写。

1. **Token和TokenStream**

ANTLR4产生的语法分析器（以语法起名stuPyd为例，其自动产生的语法分析器名为stuPydParser）继承自Parser类，用于分析源输入文件经词法处理后产生的Token流。举例说明，我们给出各个语法非终结符的定义（例如file\_input,INDENT等）并使ANTLR4生成分析器之后，我们可以在其中看到如下成员：

RULE\_file\_input = 0  
RULE\_stmt = 1  
RULE\_simple\_stmt = 2  
RULE\_small\_stmt = 3

...  
INDENT=38  
DEDENT=39

这些即为token的type属性。语法分析所分析的便是由type表示token内容的token流。

Token是由词法分析器生成的。词法分析结束后生成的token对象输出时如下：

[@-1,0:2='num',<3>,1:0]

其中我们用的到的属性有两个，一个是’num’，即token.text，存放token的原文，另一个是<3>，即token.type，存放词法分析的int型类型结果，语法分析器便在按次序存放的CommonTokenStream类里（默认情况下将Token放于其中将由程序自动完成，我们无需操心。但在INDENT和DEDENT生成的要求下还要考虑其原理的问题。后文有提及）针对type进行分析。

在Python Document给出的文法中可以看到，涉及代码分层的模块被命名为suite。Suite的推导如下：

suite ： NEWLINE INDENT stmt+ DEDENT

举例说明，在我们设计的stuPyd.g4里，suite用法形如

while\_stmt : ‘while’ bool ‘:’suite;

这就是一个模仿python型缩进语言的典型案例。写出来形如：

while True :

Code..

Code...

Other code...

其中NEWLINE是指换行符‘\n’等，INDENT指“更多的缩进个数”，DEDENT指缩进退格，用过python的用户可能更能理解其含义。若实现语法分析，他们都必须以Token的形式出现在CommonTokenStream里。但从我们希望达到的效果来看，INDENT和DEDENT并不能够以正则表达式来描述他们，也就是Parser不能通过正常的方式获取INDENT和DEDENT的token。因此我们要对词法分析器嵌入代码来生成这两个token。首先是在.g4文件的grammar语句下面写：

tokens{INDENT,DEDENT}

tokens是ANTLR4内置的关键字，意为对括号内的token符号进行声明，表明他们将会以非自动的方式由用户生成但将不在正文里做正则表达式解释。因此我们在写下tokens之后可以写出上文suite的产生式而避免报错，同时也会在stuPydParser里生成这两个Token的代号，也就是上文中stuPydParser中自动生成的的INDENT和DEDENT属性。接下来就是在代码中生成INDENT和DEDENT了。

类CommonToken是用来创建新的token实例的，我们可以通过一些方法将他们送进CommonTokenStream。其构造函数如下：

**def** \_\_init\_\_(self, source:tuple = EMPTY\_SOURCE, type:int = **None**, channel:int=Token.DEFAULT\_CHANNEL, start:int=-1, stop:int=-1):

其中，source:tuple这个参数可以用Lexer内置成员填上，在下一节会介绍；type可以随意选择，如上文所示，我们将使用stuPydParser.INDENT或stuPydParser.DEDENT来填补这个参数；channel用以标识token存放位置，这一条不需额外修改。Start和stop将标识该token的文本开始和结束位置，随后text由这两个参数决定。

1. **Lexer**

用户生成的词法分析器stuPydLexer继承自Lexer，简单说只是增加了用户定义的关键字和嵌入的代码。因此要理解程序运行机制需要从Lexer开始研究。Lexer里有以下几个属性需要注意：

1） \_input

在调用lexer时可以为构造函数填入多种输入，包括字节流和文件流，也就是\_input即为输入流。在后文使用时主要是通过self.\_input.LA(1)这一方法。该方法可以在不移动文件指针的情况下获取输入流的下一个字符（char型，或者理解为int型的ascii码也可）。

1. \_tokenFactorySourcePair

这个无需太多注意。用来填写CommonToken类的tuple参数即可。

1. \_token

用来临时存储当前识别的token。在一个词法符号识别结束后发送给CommonTokenStream。在类内可以直接调用并修改。

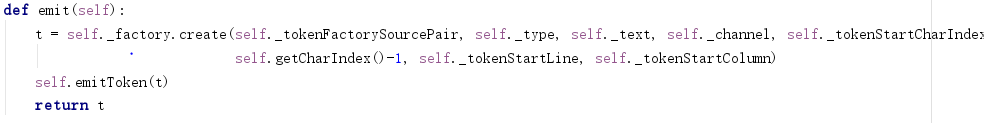
1. nextToken(self)

这是一个比较重要的函数。在我们的需求中要对它进行重写。它的功能摘取ANTLR4产生的代码注释：从自身来源中返回一个token。换言之，在输入流中向后匹配一个token。该方法指挥词法分析器跳过当前词法规则以寻找下一个token。当一个词法规则分析完一个被标记skip的token时nextToken() 会自动寻找下一个。注意： 如果在任何token规则结束时token为空，它将仍会创建一个并且将它送到token流。

在我们重写时，会考虑到它本身的功能以及配套函数emit的协同工作。

1. emit(self)

emit，即发送token的函数。函数本体声明如下：

它自动根据实例化的Lexer自身属性生成token（这些属性由其他类方法修改，我们不需关心），并发送到自身的“token缓冲区”self.\_token（这是由emitToken完成的。但请注意此时仅存放在self.\_token里，并没有发送到CommonTokenStream）。

1. **CommonTokenStream以及分析**

CommonTokenStream继承自BufferedTokenStream，在这个类中最重要方法是fetch()。其会循环调用输入源lexer的nextToken方法，对该方法返回的token按次序排列。

经过对以上源码的阅读和理解，我们就可以理清思路了，下面的代码也更容易理解。我们的方案是：

0.设置一个缩进栈以存储最近缩进个数；设置一个token栈以保存待发送的token符号（这点在后面做解释）

1.对NEWLINE后面紧跟的换行符号进行计数（即把生成INDENT的操作写在NEWLINE的嵌入代码里），而对未出现在换行符后的空格不作处理（在语法文件里skip掉），若缩进栈为空则默认栈顶为0.如果空格大于栈顶空格个数，就产生INDENT，并将此时的空格个数压入栈中；如果空格数与栈顶相等，则不作处理调用self.skip()；如果空格数小于栈顶，则产生DEDENT并栈顶出栈，继续判断是否相等。

这其中有一个至关重要的问题也是唯一难点：有可能两层缩进在同一行结束，这意味着分析同个词法符号NEWLINE时很可能不止产生一个token（即使该行只有一个缩进，也面临着产生NEWLINE和INDENT两个token的难题）。这使得CommonTokenStream无法正常工作，因为lexer的\_token属性仅能容纳一个token，而它将在CommonTokenStream调用lexer.nextToken()时返回。因此设置token栈的意义在于：通过改写nextToken()和emit()，使emit()发送token到栈中而nextToken()在token栈不为空时返回栈顶元素，在token栈为空时才调用父类的nextToken()去读取下一个token。

问题解决了，下面是代码实现。

1. **代码实现及注释**

# grammar stuPyd;

//decls: waiting for write

@lexer::header{

from stuPydParser import stuPydParser

from antlr4.Token import CommonToken

}

@lexer::members{

self.lastToken = None

self.tokens = []

self.indentStack = []

def emit(self,t=None):

if t == None :

s = self.\_factory.create(self.\_tokenFactorySourcePair, self.\_type, self.\_text, self.\_channel, self.\_tokenStartCharIndex,

self.getCharIndex()-1, self.\_tokenStartLine, self.\_tokenStartColumn)

self.emitToken(s)

print(s)

self.tokens.append(s)

return s

else:

self.emitToken(t)

self.tokens.append(t)

print(t)

return t

def commonToken(self,type,text):

stop = self.getCharIndex()-1

if len(text)==0 :

start = stop

else:

start = stop - len(text)+1

return CommonToken(self.\_tokenFactorySourcePair,type,self.DEFAULT\_TOKEN\_CHANNEL,start,stop)

def createDedent(self):

dedent = self.commonToken(stuPydParser.DEDENT,'')

dedent.line = self.lastToken.line

return dedent

@staticmethod

def getIndentationCount(spaces):

count = 0

for ch in spaces:

if ch == '\t':

count +=( 4 - (count%4))

elif ch == ' ':

count += 1

else :

pass

# print(count)

return count

def nextToken(self):

# Check if the end-of-file is ahead and there are still some DEDENTS expected

if(self.\_input.LA(1)==Token.EOF and len(self.indentStack)!=0):

# Remove ant trailing EOF tokens from our buffer

i = len(self.tokens)-1

while i>= 0 :

if(self.tokens[i].getType()==Token.EOF):

self.tokens.pop(i)

i-=1

# First emit an extra line break that serves as the end of the stmt

self.emit(self.commonToken(stuPydParser.NEWLINE,'\n'))

# Now emit as much DEDENT tokens as needed

while len(self.indentStack)!=0 :

self.emit(self.createDedent())

self.indentStack.pop()

# put the EOF back on the token stream .

self.emit(self.commonToken(stuPydParser.EOF,'<EOF>'))

next = super().nextToken()

if (next.channel == Token.DEFAULT\_CHANNEL):

# Keep track of the last token on the default channel

self.lastToken = next

if len(self.tokens) == 0 :

# print(next)

return next

else :

temp = self.tokens[0]

self.tokens.pop(0)

#print(temp)

return temp

def atStartOfInput(self):

if super().getCharIndex()==0 and super().line==1 :

# print('True')

return True

else:

# print('False')

return False

}

tokens{INDENT,DEDENT}

fragment SPACES : [ \t]+;

NEWLINE

:

( {self.atStartOfInput()}? SPACES

| ( '\r'? '\n' | '\r' ) SPACES?

)

{

newLine = self.text

for i in newLine :

if i == '\r' or i == '\n' or i == '\f':

pass

else:

newLine.replace(str(i),'')

spaces = self.text.replace('\r','')

spaces = self.text.replace('\n','')

spaces = self.text.replace('\f','')

next = self.\_input.LA(1)

if(next=='\r' or next == '\n' or next == '\f' or next == '<EOF>'):

self.skip()

else:

self.emit(self.commonToken(self.NEWLINE,newLine))

indent = self.getIndentationCount(spaces)

previous = 0

if len(self.indentStack) != 0 :

previous = self.indentStack.pop()

self.indentStack.append(previous)

# it is equal to indentStack.peek()

if indent==previous :

# skip indents of the same size as the present indent-size

self.skip()

elif indent > previous :

self.indentStack.append(indent)

self.emit(self.commonToken(stuPydParser.INDENT,spaces))

else:

# Possibly emit more than 1 DEDENT token

while len(self.indentStack)!=0 and self.indentStack[len(self.indentStack)-1]>indent:

self.emit(self.createDedent())

self.indentStack.pop()

}

;

BLANK : (SPACES|COMMENT)+ ->skip

;

通过这几个即可实现缩进式语法。不要忘记

Suite：NEWLINE INDENT stmt+ DEDENT

其他的语法细节读者可自行添加。