Contents

1 导言

文学编程无疑是非常强大的工具。然而，要将这个强大的工具应用于python，却不是那么容易的。原因在于，python没有类似于c/c++的“line directive”预处理标记，因此无法直接判断python源文件中的一段代码究竟来源于*.nw文件中的哪一行。而不解决这个问题，就没法应付大规模的代码段，于是文学编程的长处便无法发挥。

所以，这篇文章的目的，就是为了解决这个问题。

2 几种设想

为了模拟“Line Directive”这种工具，按照经验，可能有这么几种途径：

1. python有某种方法直接支持“line directive”。
2. 通过异常处理，修改异常中的堆栈信息。

目前来看，似乎并不存在第一种方法，因此只能用第二种方法了。

第二种方法最直接的问题就是，什么时候捕捉异常？python中并没有类似“OnException”这种回调。因此也就没有一个集中的异常处理地点。于是，每一个处理异常的地方，恐怕都需要插入源文件转换的代码。

按照这样的思路，python的tangle过程可能就需要变成一种二段构造的方式：1. 生成基本的源代码，2. 扫描源代码，在异常处理处插入源文件转换。

为了验证这个想法，我们需要做一系列的实验：

1. 异常时扫描异常的堆栈信息

1a ⟨test.py 1a⟩≡

This definition is continued in chunk 2.

Root chunk (not used in this document).

1b ⟨在异常处理中获取堆栈信息 1b⟩≡

```
import traceback
def foo():
    raise Exception("an exception")

try:
    foo()
except Exception, e:
    stackStr = str(traceback.format_exc())
    lines = stackStr.split('\n')
    for line, index in zip(lines, range(len(lines))):
        print index, ':', line
```

Root chunk (not used in this document).

上面的代码段使用了traceback这个工具库，这意味着， 我们的Line Directive输出工具恐怕要变成单独的一个脚本。

同时，上面的代码段的打印信息揭示了， 1. 堆栈信息中的文件信息是以File起头的， 2. 基本上可以用正则表达式匹配。

这就为我们的Line Directive输出工具提供了执行基础： 我们可以知道，哪个文件的哪一行发生了错误。

紧接着，我们需要解决第二个问题： 通过什么样的方式来输出我们的出错信息？

python在捕获未处理的异常时会输出异常信息并结束程序。 最自然的当然是在这个时候输出我们的信息。

python的标准异常有args这个成员，是一个tuple， 可以被改写，于是我们可以利用这个玩意来输出我们的信息。 不过，美中不足的是，标准异常里， 打印args时是不带换行符的。 不过这个总比没有强就是了。

决定了如何输出我们的源代码行信息以后，我们就需要进一步解决细节问题了。第一个问题是，在嵌套raise的情况下，多次查询、插入堆栈信息是否会有问题？这需要下面的代码段来回答。在下面的代码段中，我们抛出一个异常，第一次捕获后打印当前堆栈信息，第二次捕获后依然打印堆栈信息，我们通过两次的比较，确认两次的堆栈信息是否一致。

2 ⟨test.py 1a⟩+≡

```
import traceback
import traceback
def foo():
    raise Exception("an exception")

def bar():
    print 'in bar'
    try:
        foo()
    except Exception, e:
        stackStr = str(traceback.format_exc())
        lines = stackStr.split('\n')
        for line, index in zip(lines, range(len(lines))):
            print index, ':', line
        e.args = (e.args[0], 'test')
        raise

try:
    bar()
except Exception, e:
    print 'in main'
    stackStr = str(traceback.format_exc())
    lines = stackStr.split('\n')
    for line, index in zip(lines, range(len(lines))):
        print index, ':', line
    print e
```

经过实验发现，上面的代码段揭示了，1) 如果raise带参数，堆栈信息会被改写，否则不会；2) 在不改写的情况下，每次raise都会层层加码; 3) re-raise抛出的依然是e，所以可以在修改e.args后re-raise。不过，在我们的场景中，层层加码，或者说嵌套地改写文件信息的情况不需要考虑太多。因为我们只需要在每次改写的时候替换args中对应的参数即可。

那么，LineDirective的方案就是：

1. LineDirective是一个单独的模块(module)
2. 它有“Forwarding”的功能，即，可以调用任意的模块，并且将命令行的参数传递给该模块
3. 用户程序可以使用LineDirective提供的工具对异常进行检测，获得出错信息对应的文件的位置。
4. 它提供一个扫描工具，可以使用扫描工具扫描tangle而成的源文件，从而实现py文件和nw文件之间的映射。
5. 这种映射关系只需要在tangle时扫描，而不是每次运行时都扫描。
6. 代码在主动抛出异常时，应该调用模块提供的工具来提供真正的堆栈信息。
7. 扫描工具会改写源文件，自动在代码中将要抛出异常的地方插入生成堆栈信息的代码。

3a ⟨tangle source codes 3a⟩≡

```
notangle -Rtest.py $file>test.py
```

This code is used in chunk 3c.

3 action

3b ⟨action 3b⟩≡
```
⟨tangle_in_linux 3c⟩
⟨weave 4b⟩
```
Root chunk (not used in this document).

3.1 tangle

3c ⟨tangle_in_linux 3c⟩≡
```
fileName=使用noweb对python进行文学编程
file=$fileName.nw
ltx_file=$fileName.ltx
aux_file=$fileName.aux
log_file=$fileName.log
function tangleSource
{

    notangle -R"$1" -t4 -L'#line %L "%F"%N' $2 | iconv -f utf-8 -t gbk > $3
    astyle --style=ansi --mode=c $3
    iconv -f gbk -t utf-8 $3 > $3.utf-8
}
⟨tangle source codes 3a⟩
⟨tangle_windows_part 3d⟩
```
This code is used in chunk 3b.

3d ⟨tangle_windows_part 3d⟩≡
```
notangle -R"action\\_in\\_win" -t4 $file> action.bat
```
This code is used in chunk 3c.

4a ⟨action_in_win 4a⟩≡
```
@echo off
test.py
pause
exit 0
```
Root chunk (not used in this document).

3.2 weave

4b ⟨weave 4b⟩≡

```
noweave -x $file| \
sed 's/\\usepackage{noweb}/\\usepackage[top=1.2in,bottom=1.2in,left=1.2in,right=1in]{geometry}&/g'| \
sed 's/\\usepackage{noweb}/\\usepackage{fontspec, xunicode, xltxtra}&/g'| \
sed 's/\\usepackage{noweb}/\\usepackage{listings}&/g'| \
sed 's/\\usepackage{noweb}/\\usepackage[120, ampersand]{easylist}&/g'| \
sed 's/\\usepackage{noweb}/\\usepackage{paralist}&/g'| \
sed 's/\\usepackage{noweb}/\\usepackage{color}&/g'| \
sed 's/\\usepackage{noweb}/\\usepackage{hyperref}&/g'| \
sed 's/\\usepackage{noweb}/\\usepackage{underscore}&/g'| \
sed 's/\\usepackage{noweb}/&\\noweboptions{longxref}/g'| \
sed 's/\\usepackage{noweb}/&\\noweboptions{smallcode}/g'| \
sed 's/\\usepackage{noweb}/&\\noweboptions{alphasubpage}/g'| \
sed 's/\\usepackage{noweb}/&\\noweboptions{longchunks}/g'| \
sed 's/\\usepackage{noweb}/&\\XeTeXlinebreaklocale "zh-cn"/g'| \
sed 's/\\usepackage{noweb}/&\\pagecolor{grayyellow}/g'| \
sed 's/\\usepackage{noweb}/&\\definecolor{grayyellow}{RGB}{255, 255, 200}/g'| \
sed 's/\\usepackage{noweb}/&\\XeTeXlinebreakskip = 0pt plus 1pt minus 0.1pt/g'| \
sed 's/\\usepackage{noweb}/&\\setmainfont[BoldFont={Adobe Heiti Std}]{Adobe Song Std}/g'| \
sed 's/\\usepackage{noweb}/&\\setmonofont[Color=0000FF99]{Microsoft YaHei UI Light}/g'| \
sed 's/\\usepackage{noweb}/\\usepackage{amsmath}&/g'| \
sed 's/\\usepackage{noweb}/\\usepackage{amssymb}&/g'| \
sed 's/\\begin{document}/&\\tableofcontents/g'| \
sed 's/\\begin{document}/&\\setcounter{tocdepth}{7}/g'| \
sed 's/\\documentclass[11pt]/&[11pt]/g'|
sed 's/  /  /g'> $ltx_file &2|iconv -f utf-8 -t gbk
xelatex $ltx_file
xelatex $ltx_file
echo $ltx_file|sed 's/ltx$/aux/g'|xargs rm -rf
echo $ltx_file|sed 's/ltx$/toc/g'|xargs rm -rf
echo $ltx_file|sed 's/ltx$/out/g'|xargs rm -rf
rm $ltx_file
rm $aux_file
rm $log_file
```

This code is used in chunk 3b.

5 ⟨declare of literate programming 5⟩≡

```
/*

    ****************************************
    *                  *
    *       注意事项        *
    *                  *
    ****************************************

  你看到的这份源码文件不是直接生成的,而是使用noweb工具，从*.nw文件中将代码抽取出来组织而成的。
  因此请不要直接编辑这些源文件，否则它们会被*.nw文件中的内容覆盖掉。
  如果想了解如何使用noweb工具抽取代码和生成pdf文档，请联系huangyangkun@gmail.com。
  noweb是一个"文学编程（literate programming）"工具。
  关于文学编程: http://zh.wikipedia.org/wiki/%E6%96%87%E5%AD%A6%E7%BC%96%E7%A8%8B
  关于noweb: http://en.wikipedia.org/wiki/Noweb
  */
```

Root chunk (not used in this document).

4 代码块列表