

Contents

1	 	1
2	 	1
3	 	8
4	 	9
4.1	tangle	9
4.2	weave	11
5	 	13

1

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

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

- python有某种方法直接支持“line directive”。
- 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：

- 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
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。

Forwarding是为了捕捉异常，并将异常的堆栈信息转化为nw文件中的文件信息。因此LineDirective需要一个nw文件和输出文件的对应表，这个对应表自然应该在tangle时生成。

可是，我们该如何最自然的去使用LineDirective这个模块呢？这首先要分析一下我们一般是如何使用一个python的脚本的。使用python的脚本有两种方式：1) 作为模块引入（import module），2) 直接执行。在第一种情况下，脚本抛出的异常可能会，也可能不会被引入它的模块所处理，而第二种情况则必然不会被处理。我们当然希望，任何情况下，我们看到的异常的堆栈信息都已经对应好了。

2014年8月2日：万幸，python提供了sys.excepthook这个回掉接口，用于处理未捕捉的异常。不过这个文档还不是很明确，因此我还担心它在和其他IDE使用时，或者这个回掉被改写时，会是怎样的结果。不过不管怎么样，至少我找到了一个统一的入口来处理者系欸未处理的异常了。下面的代码是一个例子。

```
3 <testhook 3>≡
import sys
import functools
import traceback
import inspect
def onUnhandledExcept(oldHandler, type, value, tb):
    print '#####'
    print inspect.getfile(inspect.currentframe())
    print type(tb)
    print dir(tb)
    print tb
    print type
    print value
    print traceback.extract_tb(tb)
    value.args = value.args + ('aaa',)
    oldHandler(type, value, tb)
print sys.excepthook
sys.excepthook = functools.partial(onUnhandledExcept, sys.excepthook)
def bad():
    raise Exception('test', 'a tuple?')
bad()
```

Root chunk (not used in this document).

所以现在可以正式设计LineDirective了：

1. LineDirective和整个tangle过程紧密结合
2. LineDirective深度集成到生成的代码中
3. tangle的流程：
 - 3.1. 生成原始的py文件
 - 3.2. 用LineDirective扫描py文件，生成最终的py文件
 - 3.2.1. 扫描py文件，建立nw-py的行对应信息
 - 3.2.2. 用一个临时的py文件，试着import原始的py文件，以检测是否有语法错误
 - 3.2.3. 如果有语法错误，输出错误，停止生成，如果没有，则生成最终的py文件
 - 3.2.3.1. LineDirective在最终的py文件中的头部添加一些内容，用于设置sys.excepthook

于是我们的tangle就需要：

1. tangle LineDirective
2. 定义一个shell函数，以对单个的py文件tangle
 - 2.1. tangle一个源文件，其中包含对应的line directive
 - 2.2. 用LineDirective扫描这个源文件，生成目标文件。
 - 2.3. 处理过程中的错误应该被以log的形式写文件，以备修正

```
4  <LineDirective.py 4>≡
    import sys, re, pickle
    codeTemplate = """
    #-*-coding: utf-8-*-
    import functools, sys, inspect, re, traceback, pickle
    <替换异常打印的函数 7>
    sys.excepthook = functools.partial(replaceStackTrace, sys.excepthook, inspect.getfile(inspect.currentframe()))
    """

    <文件行转换器 5a>
    <LineDirective Definition 5b>
    if __name__ == '__main__':
        if (sys.argv) > 1:
            scan(sys.argv[1])
```

Root chunk (not used in this document).

```

5a  <文件行转换器 5a>≡
    class Chunks:
        def __init__(self):
            self.chunkList = []
        def insertChunk(self, srcLine, dstLine):
            pos = self.seekInsertPos(dstLine)
            self.chunkList.insert(pos, [srcLine, dstLine])
        def seekInsertPos(self, dstLine):
            arr = self.chunkList
            if len(self.chunkList) == 0:
                return -1
            if arr[0][1] > dstLine:
                return 0
            if arr[-1][1] < dstLine:
                return len(arr)
            lb = 0
            ub = len(self.chunkList)
            while True:
                if lb >= ub:
                    return lb
                mid = lb + (ub - lb)/2
                if dstLine > arr[mid][1]:
                    lb = mid + 1
                elif dstLine < arr[mid][1]:
                    ub = mid
                else:
                    return mid
        def sourceLine(self, dstLine):
            pos = self.seekInsertPos(dstLine)
            srcLine, inDst = self.chunkList[pos - 1]
            return srcLine + dstLine - inDst

```

This code is used in chunk 4.

```

5b  <LineDirective Definition 5b>≡
    def scan(srcPath):
        src = open(srcPath, 'r')
        lines = list(src)
        pattern = re.compile('^#line (\d+), (.+)')
        chunks = Chunks()
        for line, lineNum in zip(lines, range(len(lines))):
            m = pattern.match(line)
            if m != None:
                nwLine = m.group(1)
                nwName = m.group(2)
                chunks.insertChunk(int(nwLine), lineNum)
    <生成包裹后的代码 8a>

```

This code is used in chunk 4.

生成包裹后的代码是一个技术活。需要干的几件事依次是： 1) 生成替换异常打印的函数， 2) 计算包裹后的偏移量, 3) 将偏移量应用到包裹后的代码中

2014年8月7日: `excepthook`的第三个参数`traceback`是堆栈信息, 可以用`traceback`模块的`extract_tb`函数获得堆栈的列表。列表项是一个(文件路径, 行数, 函数, 语句)的四元组。

由于`excepthook`只处理那些未被捕捉的异常, 因此在这个函数被调用时, 异常已经到达了堆栈的最底层, 此时获得的堆栈信息必然是完整的。

而我们的替换函数设计成责任链的形式，会逐次调用被替换的`excepthook`。同时，我们不可能为每个文件编写不同的替换函数，必须以数据来控制函数的执行，而非用代码。所以，替换的流程这么设计：

1. 替换函数首先检查异常的`args`中是否有堆栈信息，如果没有，则使用`extract_tb`抽取堆栈信息，放入`args`中。
2. 遍历`args`中的堆栈信息，根据函数持有的源文件-目标文件对应表来替换其中可以替换的四元组。
3. 调用旧的`excepthook`。

7 <替换异常打印的函数>=

```
def revealLiterate(originFile, thisFile, chunks, stacks):
    replaced = []
    for dstPath, lineNumber, frame, source in stacks:
        if dstPath == thisFile:
            lineNumber = sourceLine(chunks, lineNumber - %d)
            dstPath = originFile
        replaced.append((dstPath.decode('utf-8'), lineNumber, frame, source))
    return replaced
def replaceStackTrace(nextHandler, thisFile, type, value, tb):
    chunks = %s
    if len(value.args) == 0:
        resultDict = {}
        resultDict["dictId"] = u"9D6B6AA1-92FC-453E-8B9A-91D0E02A17B1"
        resultDict["stackInfo"] = traceback.extract_tb(tb)
        value.args = value.args +(resultDict, )
    else:
        resultDict = value.args[-1]

    if type(resultDict) != type({}):
        resultDict = {}
        resultDict["dictId"] = u"9D6B6AA1-92FC-453E-8B9A-91D0E02A17B1"
        resultDict["stackInfo"] = traceback.extract_tb(tb)
        value.args = value.args +(resultDict, )

    if "dictId" not in resultDict or resultDict["dictId"] != u"9D6B6AA1-92FC-453E-8B9A-91D0E02A17B1":
        resultDict = {}
        resultDict["dictId"] = u"9D6B6AA1-92FC-453E-8B9A-91D0E02A17B1"
        resultDict["stackInfo"] = traceback.extract_tb(tb)
        value.args = value.args +(resultDict, )

    resultDict['stackInfo'] = revealLiterate("%s", thisFile, chunks, resultDict["stackInfo"])
    if '<built-in function excepthook>' == str(nextHandler):
        print 'Unhandled Exception, trace back:'
        for stackInfo in resultDict['stackInfo']:
            print ur' File "' + unicode(stackInfo[0]) + ur"', line ' + unicode(stackInfo[1]) + ur' in ' + unicode(stackInfo[2])
            print ur' ' + unicode(stackInfo[3])
        value.args = value.args[:-1]
        print re.compile(r"<type '([^\']+)>").match(str(type)).group(1)+":", value
    elif None != nextHandler:
        nextHandler(type, value, tb)
```



```

def seekInsertPos(arr, dstLine):
    if len(arr) == 0:
        return -1
    if arr[0][1] > dstLine:
        return 0
    if arr[-1][1] < dstLine:
        return len(arr)
    lb = 0
    ub = len(arr)
    while True:
        if lb >= ub:
            return lb
        mid = lb + (ub - lb)/2
        if dstLine > arr[mid][1]:
            lb = mid + 1
        elif dstLine < arr[mid][1]:
            ub = mid
        else:
            return mid

def sourceLine(arr, dstLine):
    pos = seekInsertPos(arr, dstLine)
    srcLine, inDst = arr[pos - 1]
    return srcLine + dstLine - inDst

```

This code is used in chunk 4.

8a 〈生成包裹后的代码 8a〉≡

```

srcString = codeTemplate % (len(codeTemplate.split("\n")), str(chunks.chunkList), nwName)
print srcString
for line in lines:
    print line[:-1]

```

This code is used in chunk 5b.

8b 〈tangle source codes 8b〉≡

```

echo '#-*-coding: utf-8 -*-'>LineDirective.py
../pytangle.py -RLineDirective.py -L'#line %L, %F%N' $file>>LineDirective.py
#notangle -Rtesthook $file>test.py

```

This definition is continued in chunk 9b.

This code is used in chunk 9d.

3 冒烟测试

测试方案:

1. 弄一个必然会出错的py文件
2. 用LineDirective扫描
3. 执行扫描生成的文件
4. 检查异常的结果是否正确

9a <error.py 9a>≡
 a=[1]
 print a[3]

Root chunk (not used in this document).

9b <tangle source codes 8b>+≡
 ../pytangle.py -Rerror.py -L'#line %L, %F%' \$file>error.py
 This code is used in chunk 9d.

4 action

9c <action 9c>≡
 <tangle_in_linux 9d>
 <weave 11>

Root chunk (not used in this document).

4.1 tangle

9d <tangle_in_linux 9d>≡
 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%"' \$2 | iconv -f utf-8 -t gbk > \$3
 ../pytangle.py -R"\$1" -L'#line %L "%F%"' \$2> \$3
 #iconv -f gbk -t utf-8 \$3 > \$3.utf-8
 }
 <tangle source codes 8b>
 <tangle_windows_part 9e>

This code is used in chunk 9c.

9e <tangle_windows_part 9e>≡
 notangle -R"action_in_win" -t4 \$file> action.bat

This code is used in chunk 9d.

```
10  <action_in_win 10>≡  
    @echo off  
    REM test.py test.py>test_dst.py  
    LineDirective.py LineDirective.py>temp.py  
    del LineDirective.py  
    rename temp.py LineDirective.py  
    LineDirective.py error.py>temp.py  
    del error.py  
    rename temp.py error.py  
    error.py  
    REM test.py test.py  
    pause  
    exit 0
```

Root chunk (not used in this document).

4.2 weave

```

11  <weave 11>≡
    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, xltextra}&/g'| \
    sed 's/\\usepackage{noweb}/\\usepackage{listings}&/g'| \
    sed 's/\\usepackage{noweb}/\\usepackage[120, ampersand]{easylst}&/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 -rf $ltx_file
    rm -rf $aux_file
    rm -rf $log_file

```

This code is used in chunk 9c.

12 <declare of literate programming 12>≡

/*

*

*

* 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).

5 代码块列表

⟨action 9c⟩ [9c](#)
⟨action_in_win 10⟩ [10](#)
⟨declare of literate programming 12⟩ [12](#)
⟨error.py 9a⟩ [9a](#)
⟨LineDirective Definition 5b⟩ [4](#), [5b](#)
⟨LineDirective.py 4⟩ [4](#)
⟨tangle source codes 8b⟩ [8b](#), [9b](#), [9d](#)
⟨tangle_in_linux 9d⟩ [9c](#), [9d](#)
⟨tangle_windows_part 9e⟩ [9d](#), [9e](#)
⟨test.py 1a⟩ [1a](#), [2](#)
⟨testhook 3⟩ [3](#)
⟨weave 11⟩ [9c](#), [11](#)
⟨在异常处理中获取堆栈信息 1b⟩ [1b](#)
⟨文件行转换器 5a⟩ [4](#), [5a](#)
⟨替换异常打印的函数 7⟩ [4](#), [7](#)
⟨生成包裹后的代码 8a⟩ [5b](#), [8a](#)