INDEX
- Python教程
  - Python简介
  - 安装Python
  - 第一个Python程序
  - Python基础
  - 函数
  - 高级特性
  - 函数式编程
  - 模块
  - 面向对象编程
  - 面向对象高级编程
  - 错误、调试和测试
    - 错误处理
    - 调试
    - 单元测试
    - 文档测试
  - IO编程
  - 进程和线程
  - 正则表达式
  - 常用内建模块
  - 常用第三方模块
  - virtualenv
  - 图形界面
  - 网络编程
  - 电子邮件
  - 访问数据库
  - Web开发
  - 异步IO
  - 实战
  - FAQ
  - 期末总结

关于作者

调试

程序能一次写完并正常运行的概率很小，基本不超过1%，总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

第一种方法简单直接粗暴有效，就是用`print()`把可能有问题的变量打印出来看看：

```
def foo(s):
    n = int(s)
    print('>>> n = %d' % n)
    return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
>>> n = 0
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

用`print()`最大的坏处是将来还得删掉它，想想程序里到处都是`print()`，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

断言

凡是用`print()`来辅助查看的地方，都可以用断言（assert）来替代：

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

`assert`的意思是，表达式`n != 0`应该是`True`，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败，`assert`语句本身就会抛出`AssertionError`：

```
$ python err.py
Traceback (most recent call last):
  ...
AssertionError: n is zero!
```

程序中如果到处充斥着`assert`，和`print()`相比也好不到哪去。不过，启动Python解释器时可以用`-O`参数来关闭`assert`：

```
$ python -O err.py
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

⚠ 注意：断言的开关“-O”是英文大写字母O，不是数字0。

关闭后，你可以把所有的`assert`语句当成`pass`来看。

logging

把`print()`替换为`logging`是第3种方式，和`assert`比，`logging`不会抛出错误，而且可以输出到文件：

```
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print(10 / n)
```

`logging.info()`就可以输出一段文本。运行，发现除了`ZeroDivisionError`，没有任何信息。怎么回事？

别急，在`import logging`之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这就是`logging`的好处，它允许你指定记录信息的级别，有`debug`，`info`，`warning`，`error`等几个级别，当我们指定`level=INFO`时，`logging.debug`就不起作用了。同理，指定`level=WARNING`后，`debug`和`info`就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

`logging`的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print(10 / n)
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<module>()
-> s = '0'
```

以参数`-m pdb`启动后，pdb定位到下一步要执行的代码`-> s = '0'`。输入命令`l`来查看代码：

```
(Pdb) l
  1     # err.py
  2  -> s = '0'
  3     n = int(s)
  4     print(10 / n)
```

输入命令`n`可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(4)<module>()
-> print(10 / n)
```

任何时候都可以输入命令`p 变量名`来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令`q`结束调试，退出程序：

```
(Pdb) q
```

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

pdb.set_trace()

这个方法也是用pdb，但是不需要单步执行，我们只需要`import pdb`，然后，在可能出错的地方放一个`pdb.set_trace()`，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print(10 / n)
```

运行代码，程序会自动在`pdb.set_trace()`暂停并进入pdb调试环境，可以用命令`p`查看变量，或者用命令`c`继续运行：

```
$ python err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<module>()
-> print(10 / n)
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这个方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有：

Visual Studio Code：https://code.visualstudio.com/，需要安装Python插件。

PyCharm：http://www.jetbrains.com/pycharm/

另外，Eclipse加上pydev插件也可以调试Python程序。

小结

写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。

虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

参考源码

do_assert.py

do_logging.py

do_pdb.py

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：

还可以分享给朋友：

分享到微博

Previous Page Next Page

Comments

Make a comment

Sign in to make a comment