

31 | pdb & cProfile : 调试和性能分析的法宝

2019-07-19 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 10:02 大小 9.20M



你好，我是景霄。

在实际生产环境中，对代码进行调试和性能分析，是一个永远都逃不开的话题。调试和性能分析的主要场景，通常有这么三个：

- 一是代码本身有问题，需要我们找到 root cause 并修复；
- 二是代码效率有问题，比如过度浪费资源，增加 latency，因此需要我们 debug；
- 三是在开发新的 feature 时，一般都需要测试。

在遇到这些场景时，究竟应该使用哪些工具，如何正确的使用这些工具，应该遵循什么样的步骤等等，就是这节课我们要讨论的话题。

用 pdb 进行代码调试

pdb 的必要性

首先，我们来看代码的调试。也许不少人会有疑问：代码调试？说白了不就是在程序中使用 `print()` 语句吗？

没错，在程序中相应的地方打印，的确是调试程序的一个常用手段，但这只适用于小型程序。因为你每次都得重新运行整个程序，或是一个完整的功能模块，才能看到打印出来的变量值。如果程序不大，每次运行都非常快，那么使用 `print()`，的确是很方便的。

但是，如果我们面对的是大型程序，运行一次的调试成本很高。特别是对于一些 tricky 的例子来说，它们通常需要反复运行调试、追溯上下文代码，才能找到错误根源。这种情况下，仅仅依赖打印的效率自然就很低了。

我们可以想象下面这个场景。比如你最常使用的极客时间 App，最近出现了一个 bug，部分用户无法登陆。于是，后端工程师们开始 debug。

他们怀疑错误的代码逻辑在某几个函数中，如果使用 `print()` 语句 debug，很可能出现的场景是，工程师们在他们认为的 10 个最可能出现 bug 的地方，都使用 `print()` 语句，然后运行整个功能块代码（从启动到运行花了 5min），看打印出来的结果值，是不是和预期相符。

如果结果值和预期相符，并能直接找到错误根源，显然是最好的。但实际情况往往是，

要么与预期并不相符，需要重复以上步骤，继续 debug；

要么虽说与预期相符，但前面的操作只是缩小了错误代码的范围，所以仍得继续添加 `print()` 语句，再一次运行相应的代码模块（又要 5min），进行 debug。

你可以看到，这样的效率就很低下了。哪怕只是遇到稍微复杂一点的 case，两、三个工程师一下午的时间可能就没了。

可能又有人会说，现在很多的 IDE 不都有内置的 debug 工具吗？

这话说的也没错。比如我们常用的 Pycharm，可以很方便地在程序中设置断点。这样程序只要运行到断点处，便会自动停下，你就可以轻松查看环境中各个变量的值，并且可以执行

相应的语句，大大提高了调试的效率。

看到这里，你不禁会问，既然问题都解决了，那为什么还要学习 pdb 呢？其实在很多大公司，产品的创造与迭代，往往需要很多编程语言的支持；并且，公司内部也会开发很多自己的接口，尝试把尽可能多的语言给结合起来。


这就使得，很多情况下，单一语言的 IDE，对混合代码并不支持 UI 形式的断点调试功能，或是只对某些功能模块支持。另外，考虑到不少代码已经挪到了类似 Jupyter 的 Notebook 中，往往就要求开发者使用命令行的形式，来对代码进行调试。

而 Python 的 pdb，正是其自带的一个调试库。它为 Python 程序提供了交互式的源代码调试功能，是命令行版本的 IDE 断点调试器，完美地解决了我们刚刚讨论的这个问题。

如何使用 pdb


了解了 pdb 的重要性与必要性后，接下来，我们就一起来看看，pdb 在 Python 中到底应该如何使用。

首先，要启动 pdb 调试，我们只需要在程序中，加入“import pdb”和“pdb.set_trace()”这两行代码就行了，比如下面这个简单的例子：

 复制代码


```
1 a = 1
2 b = 2
3 import pdb
4 pdb.set_trace()
5 c = 3
6 print(a + b + c)
```

当我们运行这个程序时时，它的输出界面是下面这样的，表示程序已经运行到了“pdb.set_trace()”这行，并且暂停了下来，等待用户输入。

 复制代码

```
1 > /Users/jingxiao/test.py(5)<module>()
2 -> c = 3
```

这时，我们就可以执行，在 IDE 断点调试器中可以执行的一切操作，比如打印，语法是"`p`
`<expression>`"：

 复制代码


```
1 (pdb) p a
2 1
3 (pdb) p b
4 2
```

你可以看到，我打印的是 `a` 和 `b` 的值，分别为 1 和 2，与预期相符。为什么不打印 `c` 呢？显然，打印 `c` 会抛出异常，因为程序目前只运行了前面几行，此时的变量 `c` 还没有被定义：

 复制代码


```
1 (pdb) p c
2 *** NameError: name 'c' is not defined
```

除了打印，常见的操作还有"`n`"，表示继续执行代码到下一行，用法如下：

 复制代码

```
1 (pdb) n
2 -> print(a + b + c)
```

而命令"`l`"，则表示列举出当前代码行上下的 11 行源代码，方便开发者熟悉当前断点周围的代码状态：


 复制代码

```
1 (pdb) l
2 1      a = 1
3 2      b = 2
4 3      import pdb
5 4      pdb.set_trace()
6 5  -> c = 3
```

```
7 6 print(a + b + c)
```

命令“s”，就是 step into 的意思，即进入相对应的代码内部。这时，命令行中会显示“--Call--”的字样，当你执行完内部的代码块后，命令行中则会出现“--Return--”的字样。

我们来看下面这个例子：

 复制代码

```
1 def func():
2     print('enter func()')
3
4 a = 1
5 b = 2
6 import pdb
7 pdb.set_trace()
8 func()
9 c = 3
10 print(a + b + c)
11
12 # pdb
13 > /Users/jingxiao/test.py(9)<module>()
14 -> func()
15 (pdb) s
16 --Call--
17 > /Users/jingxiao/test.py(1)func()
18 -> def func():
19 (Pdb) l
20 1 -> def func():
21 2     print('enter func()')
22 3
23 4
24 5     a = 1
25 6     b = 2
26 7     import pdb
27 8     pdb.set_trace()
28 9     func()
29 10    c = 3
30 11    print(a + b + c)
31
32 (Pdb) n
33 > /Users/jingxiao/test.py(2)func()
34 -> print('enter func()')
35 (Pdb) n
36 enter func()
```

```
37 --Return--
38 > /Users/jingxiao/test.py(2)func()->None
39 -> print('enter func()')
40
41 (Pdb) n
42 > /Users/jingxiao/test.py(10)<module>()
43 -> c = 3
```

这里，我们使用命令“s”进入了函数 func() 的内部，显示“--Call--”；而当我们执行完函数 func() 内部语句并跳出后，显示“--Return--”。

另外，

与之相对应的命令“r”，表示 step out，即继续执行，直到当前的函数完成返回。

命令“b [([filename:]lineno | function) [, condition]]”可以用来设置断点。比方说，我想要在代码中的第 10 行，再加一个断点，那么在 pdb 模式下输入“b 11”即可。

而“c”则表示一直执行程序，直到遇到下一个断点。

当然，除了这些常用命令，还有许多其他的命令可以使用，这里我就不在一一赘述了。你可以参考对应的官方文档（<https://docs.python.org/3/library/pdb.html#module-pdb>），来熟悉这些用法。

用 cProfile 进行性能分析

关于调试的内容，我主要先讲这么多。事实上，除了要对程序进行调试，性能分析也是每个开发者的必备技能。


日常工作中，我们常常会遇到这样的问题：在线上，我发现产品的某个功能模块效率低下，延迟（latency）高，占用的资源多，但却不知道是哪里出了问题。

这时，对代码进行 profile 就显得异常重要了。

这里所谓的 profile，是指对代码的每个部分进行动态的分析，比如准确计算出每个模块消耗的时间等。这样你就可以知道程序的瓶颈所在，从而对其进行修正或优化。当然，这并不


需要你花费特别大的力气，在 Python 中，这些需求用 cProfile 就可以实现。

举个例子，比如我想计算斐波拉契数列，运用递归思想，我们很容易就能写出下面这样的代码：

 复制代码


```
1 def fib(n):
2     if n == 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         return fib(n-1) + fib(n-2)
8
9 def fib_seq(n):
10     res = []
11     if n > 0:
12         res.extend(fib_seq(n-1))
13     res.append(fib(n))
14     return res
15
16 fib_seq(30)
```

接下来，我想要测试一下这段代码总的效率以及各个部分的效率。那么，我就只需在开头导入 cProfile 这个模块，并且在最后运行 cProfile.run() 就可以了：

 复制代码

```
1 import cProfile
2 # def fib(n)
3 # def fib_seq(n):
4 cProfile.run('fib_seq(30)')
```

或者更简单一些，直接在运行脚本的命令中，加入选项“-m cProfile”也很方便：

 复制代码

```
1 python3 -m cProfile xxx.py
```

运行完毕后，我们可以看到下面这个输出界面：

```
7049218 function calls (96 primitive calls) in 1.417 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
7049123/31  1.417    0.000    1.417    0.046 test.py:1(fib)
31/1      0.000    0.000    1.417    1.417 test.py:9(fib_seq)
1         0.000    0.000    1.417    1.417 {built-in method builtins.exec}
31        0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
1         0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
30        0.000    0.000    0.000    0.000 {method 'extend' of 'list' objects}
```

这里有一些参数你可能比较陌生，我来简单介绍一下：

`ncalls`，是指相应代码 / 函数被调用的次数；

`tottime`，是指对应代码 / 函数总共执行所需要的时间（注意，并不包括它调用的其他代码 / 函数的执行时间）；


`tottime percall`，就是上述两者相除的结果，也就是 `tottime / ncalls`；

`cumtime`，则是指对应代码 / 函数总共执行所需要的时间，这里包括了它调用的其他代码 / 函数的执行时间；

`cumtime percall`，则是 `cumtime` 和 `ncalls` 相除的平均结果。

了解这些参数后，再来看这张图。我们可以清晰地看到，这段程序执行效率的瓶颈，在于第二行的函数 `fib()`，它被调用了 700 多万次。

有没有什么办法可以提高改进呢？答案是肯定的。通过观察，我们发现，程序中有很多对 `fib()` 的调用，其实是重复的，那我们就可以用字典来保存计算过的结果，防止重复。改进后的代码如下所示：

 复制代码

```
1 def memoize(f):
2     memo = {}
3     def helper(x):
4         if x not in memo:
5             memo[x] = f(x)
6         return memo[x]
7     return helper
8
```



```

9 @memoize
10 def fib(n):
11     if n == 0:
12         return 0
13     elif n == 1:
14         return 1
15     else:
16         return fib(n-1) + fib(n-2)
17
18
19 def fib_seq(n):
20     res = []
21     if n > 0:
22         res.extend(fib_seq(n-1))
23         res.append(fib(n))
24     return res
25
26 fib_seq(30)

```

这时，我们再对其进行 profile，你就会得到新的输出结果，很明显，效率得到了极大的提高。

216 function calls (128 primitive calls) in 0.000 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	test.py:1(<module>)
1	0.000	0.000	0.000	0.000	test.py:1(memoize)
31/1	0.000	0.000	0.000	0.000	test.py:18(fib_seq)
89/31	0.000	0.000	0.000	0.000	test.py:3(helper)
31	0.000	0.000	0.000	0.000	test.py:9(fib)
1	0.000	0.000	0.000	0.000	{built-in method builtins.exec}
31	0.000	0.000	0.000	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}
30	0.000	0.000	0.000	0.000	{method 'extend' of 'list' objects}

这个简单的例子，便是 cProfile 的基本用法，也是我今天想讲的重点。当然，cProfile 还有很多其他功能，还可以结合 stats 类来使用，你可以阅读相应的 [官方文档](#) 来了解。

总结

这节课，我们一起学习了 Python 中常用的调试工具 pdb，和经典的性能分析工具 cProfile。pdb 为 Python 程序提供了一种通用的、交互式的高效率调试方案；而 cProfile 则是为开发者提供了每个代码块执行效率的详细分析，有助于我们对程序的优化与提高。

关于它们的更多用法，你可以通过它们的官方文档进行实践，都不太难，熟能生巧。

思考题

最后，留一个开放性的交流问题。你在平时的工作中，常用的调试和性能分析工具是什么呢？有发现什么独到的使用技巧吗？你曾用到过 pdb、cProfile 或是其他相似的工具吗？

欢迎在下方留言与我讨论，也欢迎你把这篇文章分享出去。我们一起交流，一起进步。

 极客时间

Python 核心技术与实战

系统提升你的 Python 能力



景霄
Facebook 资深工程师

新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 真的有必要写单元测试吗？

精选留言 (12)

写留言



return

2019-07-19

全文这个装饰器最牛逼。

展开 ∨



5



AllenGFLiu

2019-07-19

哥，不怀好意的问一下，你们在谷歌会用微软的vscode 吗？[奸笑脸]
每次看你都是提到pycharm ,我是从pycharm 转到vscode 上的，感觉整个世界都安静了。

作者回复: 哈哈，我是fb的，不用vscode的哈



1

3



(_ _)

2019-07-19

和gdb的命令差不多

展开 ∨



2



小侠龙旋风

2019-07-21

我要把这个装饰器保存下来

展开 ∨



leixin

2019-07-20

ipython 好像也带debug模式

展开 ∨



Claywoow

2019-07-20

最近工作的一段时间一直在用pdb调试。。。pdb大法好啊：)





CN

2019-07-19

老师您好，最近使用requests.get方法遇到一个问题，返回结果部分内容是“<noscript>You need to enable JavaScript to run this app.</noscript>”，请问您遇到过这种情况吗，怎么解决呢。谢谢



未来已来

2019-07-19

被最后那个装饰器惊艳到了，以前只知道用循环，没想到 Python 还可以这么玩



JackDong

2019-07-19

还有一个ipdb是pdb的加强版，用法比较相近，不过需要pip安装一下



lipan

2019-07-19

最近在用js撸小程序，一个console.log()搞定所有调试。



321

2019-07-19

web 应用怎么调试？譬如flask或django框架开发的应用，该如何调试。

作者回复: 看你侧重点是什么？如果是性能，有类似cProfile的工具，如果只是简单的debug，可以通过设置breakpoint或者print



ikimiy

2019-07-19

python的pdb package和Linux下的pdb debug工具很类似

