第16章. 调试和分析

我们编写程序时，很少有能一次成功的，大部分情况下都会因各种各样的原因导致错误。而“调试（debugging）”就是找出并消除这些错误的过程。即便一个程序没有错误，依然有可能运行效率很低，此时就需要通过“分析（analysis）”来判断该程序的瓶颈在哪里，进而进行改进。调试与分析是每个编程者都需要掌握的技巧，然而用于调试与分析的手段和工具非常多，即便专门写上一本书都无法尽述。这一节讨论的是最基本的Python调试与分析技术，不依赖于任何第三方工具。当你成为一个有经验的编程者后，可以按照自己的喜好选择更方便的调试与分析工具，但它们的原理依然符合本章中讨论的内容。

16-1. 利用非调试专用特性实现调试

（标准库：Python开发模式）

事实上，本书前面介绍的很多语法特性虽然不是专门用于调试的，但都能被用于达到调试的目的。我们就从这里开始。

第7章介绍的I/O接口可用于输出某个变量的当前值，而这就构成了最原始的调试方法。举例来说，对于下面的代码：

```
l = []
i = 0
while i < 10:
    l.append(i*3)
    continue
    i += 1
```

通过在循环体的最开头插入“print(i)”，就可以跟踪循环变量i的变化，进而发现这是一个死循环。

第8章介绍的异常对调试也有极大的帮助。当一个Exception派生异常被抛出时，不论其是否会被传递，只要它没有被try语句捕获并处理，最终都会被写入标准出错，并使脚本的运行停止。通过回溯信息，我们可以了解到脚本的哪部分出错。

警告与异常的区别仅在于它不会使脚本的运行停止。我们同样可以通过标准出错获得警告携带的回溯信息。DeprecationWarning、ImportWarning、PendingDeprecationWarning和ResourceWarning默认会被忽略，如果想让这些警告被显示，则应在启动Python解释器时添加-Wdefault，或者启动“开发模式（development mode）”。

启动开发模式的方法是在启动解释器时添加-X dev选项或者将环境变量PYTHONDEVMODE设置为1。开发模式除了相当于添加了-Wdefault外，还相当于做了如下设置：

➤ 添加-X faulthandler，启用针对信号SIGSEGV、SIGFPE、SIGABRT、SIGBUS和SIGILL的自动处理。
➤ 设置环境变量PYTHONMALLOC为debug，使得会自动检查缓冲区下溢、缓冲区上溢、内存分配API的非法使用、以及对GIL的不安全使用。
➤ 设置环境变量PYTHONASYNCIODEBUG为1，即同时启用“异步I/O调试模式（asyncio debug mode）”。

最后，类型检查工具同样对调试有很大的帮助。但从第15章的讨论可知，静态类型检查工具的功能比较弱，而动态类型检查工具则很难实现。

16-2. __debug__常量

（标准库：内置常量）

下面开始介绍Python中专门用于调试的特性。第一种特性是内置常量__debug__：当正常启动Python解释器时__debug__引用True；而如果启动Python解释器时添加了-O选项或-OO选项，那么__debug__会引用False。

我们一般会以如下方式在脚本中嵌入专门用于调试的代码：

```
if __debug__:
    ... code for debug goes here ...
```

下面是一个例子：

```
import functools


def fibonacci(n):
    if __debug__:
        #这里是调试用的代码，以跟踪递归过程。
        print('fibo(' + str(n) + ')')
    if n < 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)


@functools.cache
def fibonacci_cache(n):
    if __debug__:
```

```
            #这里是调试用的代码，以跟踪递归过程。
            print('fibo(' + str(n) + ')')
        if n < 2:
            return 1
        else:
            return fibonacci_cache(n-1) + fibonacci_cache(n-2)
```

请将上述代码保存为debug1.py，然后通过如下命令行和语句验证：

```
$ python3 -i debug1.py

>>> fibonacci(1)
fibo(1)
1
>>> fibonacci_cache(1)
fibo(1)
1
>>> fibonacci(2)
fibo(2)
fibo(1)
fibo(0)
2
>>> fibonacci_cache(2)
fibo(2)
fibo(0)
2
>>> fibonacci(3)
fibo(3)
fibo(2)
fibo(1)
fibo(0)
fibo(1)
3
>>> fibonacci_cache(3)
fibo(3)
3
>>>

$ python3 -iO debug1.py

>>> fibonacci(1)
1
>>> fibonacci_cache(1)
1
>>> fibonacci(2)
2
>>> fibonacci_cache(2)
2
>>> fibonacci(3)
3
>>> fibonacci_cache(3)
3
>>>
```

从结果可以看出，该例子通过调试代码反映了递归计算斐波拉契数列的函数在使用缓存和不使用缓存时行为有什么不同，而这些调试代码可以通过-O选项或-OO选项屏蔽。

一般而言，如果在源代码中直接通过上述方式嵌入了调试代码，那么该脚本就不准备以源代码的形式发布，而会以-m选项结合-O选项执行相应模块，进而生成.pyc文件作为无源文件发型版（注意此时生成的.pyc文件的文件名中会包含“.opt”）。这样就可以避免最终用户使用该脚本时会执行调试代码。

16-3. 断言

（语言参考手册：7.3）

“断言（assertions）”是通过assert语句实现的，其实相当于结合使用了__debug__和异常，其语法为：

```
assert expression1[, expression2]
```

如果省略了expression2，则该语句等价于：

```
if __debug__:
    if not expression1: raise AssertionError
```

否则，该语句等价于：

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

换句话说，assert语句的功能可以概括为：当对expression1求值得到的对象的逻辑值检测结果为False时，抛出一个AssertionError异常。注意就像__debug__那样，assert语句也受-O选项和-OO选项的影响。

下面是一个使用断言的例子：

```
s = "********"
n = 0
while n < 5:
    n = n + 1
    #这里断言n不超过4。
    assert n < 5
    print(s[0:n])
```

请将上述代码保存为debug2.py，然后通过如下命令行验证：

```
$ python3 debug2.py
*
**
***
****
Traceback (most recent call last):
  File "/Users/wwy/debug2.py", line 6, in <module>
    assert n < 5
AssertionError

$ python3 -O debug2.py
*
**
***
****
*****
```

虽然assert语句本质上是__debug__用法中的一个特例，但当脚本中不存在错误时该语句什么也不会做，因此即使脚本包含assert语句也可以直接发布，没必要采用无源文件发型版。

16-4. 审计

（标准库：sys）

“审计（audit）”机制既可被用于性能分析，也可被用于调试。各种Python解释器在执行Python脚本的过程中，都会针对一些特别的操作自动触发相应的“审计事件（audit events）”。每个审计事件都具有一个字符串形式的名字和一个元组形式的参数列表（可以为空元组）。它们还注册了一些被称为“审计钩子（audit hooks）”的可调用对象。当一个审计事件被触发时，所有审计钩子都会按照注册的顺序依次被调用，调用时会传入该审计事件的名字和参数列表。本书不详细讨论审计机制的细节和CPython实现的审计事件，有兴趣的读者请阅读PEP 578[1]。

我们需要了解的是，可以通过表16-1列出的sys属性来触发（自定义）审计事件，以及注册自定义审计钩子。注意能够被注册为审计钩子的可调用对象的形式参数列表应该具有“(event, parameters)”的形式。

表16-1. 审计相关sys属性

属性	说明
sys.addaudithook(*hook*)	将通过hook参数传入的可调用对象注册为审计钩子。
sys.audit(*event*, **args*)	触发名字为event，具有通过args传入的参数列表的审计事件。

下面是对debug1.py的改写，通过审计来反映递归计算斐波拉契序列的函数的执行过程：

```
import functools, sys
```

```
#自定义审计钩子。
def myaudithook(event, parameters):
    if event == 'fibo':
        print('fibo(' + str(parameters[0]) +')')


#注册审计钩子。
sys.addaudithook(myaudithook)


def fibonacci(n):
    #触发自定义审计事件。
    sys.audit('fibo', n)
    if n < 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)


@functools.cache
def fibonacci_cache(n):
    #触发自定义审计事件。
    sys.audit('fibo', n)
    if n < 2:
        return 1
    else:
        return fibonacci_cache(n-1) + fibonacci_cache(n-2)
```

请将上述代码保存为debug3.py，然后通过下面的命令行和语句来验证：

```
$ python3 -i debug3.py

>>> fibonacci(1)
fibo(1)
1
>>> fibonacci_cache(1)
fibo(1)
1
>>> fibonacci(2)
fibo(2)
fibo(1)
fibo(0)
2
>>> fibonacci_cache(2)
fibo(2)
fibo(0)
2
>>> fibonacci(3)
fibo(3)
fibo(2)
fibo(1)
fibo(0)
fibo(1)
3
>>> fibonacci_cache(3)
fibo(3)
3
>>>
```

使用审计进行调试时需要注意两点。第一点是：sys.audit()并不受-O选项或-OO选项的控制，因此如果不想让正式发型版触发自定义的审计事件，那么最好联合使用__debug__和审计。第二点是：如果触发了一个审计事件，那么所有审计钩子都会被调用，而非仅我们注册的自定义审计钩子，这会对程序的执行速度造成负面影响。事实上，对审计机制最合理的使用是将一些特别重要的审计事件默默记录到日志中以供开发者查看，而最终用户不需要知道这些情况。

16-5. pdb基础

（标准库：pdb）

仅使用上面提到的特性进行手工调试是很笨拙的。使用“调试器（debugger）”是更好的办法，而标准库中的pdb模块就是Python官方提供的调试器。也许在你熟练后，会倾向于使用某种基于GUI的调试器，但对于初学者来说，掌握pdb模块是理解调试原理的最好途径。

为了说明如何使用pdb，请先编写debug4.py：

```
#!/usr/bin/env python3
#
#这是一个用来阐述如何进行调试的Python脚本。

"""This module illustrates how to debug with pdb.

    In this module, two functions to calculate factorial are defined, one
uses loop and the other uses recursion.
"""

import sys, pdb

__all__ = ['factorial1', 'factorial2']


#这是一个通过循环实现阶乘的函数。
def factorial1(n):
    """This function calculates n! using loop."""

    product = 1
    while n > 1:
        #pdb.set_trace(header="start loop")
        product = product * n
        n = n-1
    return product


#这是一个通过递归实现阶乘的函数。
def factorial2(n):
    """This function calculates n! using recursion."""

    #breakpoint()
    if n > 1:
        return n * factorial2(n-1)
    else:
        return 1


#入口点函数。
def main():
```

```
        """This function is called when this script runs in the top-level
environment."""

        result = factorial1(3)
        print(result)
        result = factorial2(3)
        print(result)

        return 0


    #判断脚本是否在顶层环境中运行。
    if __name__ == '__main__':
        sys.exit(main())
```

这个脚本具有第7章介绍的Python脚本最规范的格式。下面会用pdb来调试它。

pdb有两种用法。最基本的用法是通过-m选项直接执行pdb，后面跟被调试的代码，即：

```
python3 -m pdb {script | -m module | -c command} [args]
```

需要注意的是，如果module参数指向的模块只有无源文件发型版，则pdb无法获得源代码，也就无法完成调试。请执行如下命令行：

```
$ python3 -m pdb debug4.py
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb)
```

当看到“(Pdb)”提示符时，就表明pdb已经启动了，而前面的两行信息第一行表明正在调试哪个脚本，第二行信息表明执行到了该脚本的哪一行。

在“(Pdb)”提示符下，我们需使用表16-2列出的pdb命令来进行调试。注意这些命令的语法中圆括号内的内容可以省略，方括号内的内容则是可选的。此外，如果不输入任何pdb命令，直接按下Enter键，则表示重新执行上一次按下Enter键时被执行的pdb命令。接下来会详细讨论所有pdb命令的用法。

表16-2. pdb命令

范畴	命令	说明
重启和退出	**run [*args* ...]**	重启被调试的Python程序，以args列表作为执行该程序时的参数。
	restart [*args* ...]	
	q(uit)	退出调试器。
查看源代码	**l(ist) [*first*[, *last*]**	显示指定范围内的代码，并标明当前行。
	ll | longlist	显示当前帧对象绑定代码对象的代码，并标明当前行。

范畴	命令	说明
执行代码	**s(tep)**	执行当前行，在第一个可以停止的位置停下。
	n(ext)	执行当前行，遇到函数调用时执行到函数返回。
	unt(il) [*lineno*]	执行到具有指定行号的行所对应的语句。如果省略行号则执行完当前行所对应的语句。
	r(eturn)	执行到当前函数调用返回。
检查对象	**whatis *expression***	对expression表达式求值，并显示得到对象的类型。
	p *expression*	对expression表达式求值，并显示得到的对象。
	pp *expression*	与p命令类似，但使用pprint模块来显示得到的对象。
	source *expression*	对expression表达式求值，并显示得到的对象所关联的源代码。
	display [*expression*]	如果给出了expression表达式，则将该表达式添加到监视表达式列表。否则列出当前监视表达式列表。
	undisplay [*expression*]	如果给出了expression表达式，则将该表达式从监视表达式列表中删除。否则清空监视表达式列表。
检查函数调用	**w(here)**	显示函数栈的当前状态。
	u(p) [*count*]	将当前帧向上移动count级（count默认取1）。
	d(own) [*count*]	将当前帧向下移动count级（count默认取1）。
	a(rgs)	显示当前帧被传入的参数。
	retval	显示当前帧的返回值。
	j(ump) *lineno*	在当前帧对应的代码中基于行号跳转。
断点调试	**c(ont(inue))**	执行到遇到断点。
	b(reak) [([*filename*:]*lineno* | *function*)[, *condition*]]	如果给出lineno参数或function参数，则设置一个断点。否则列出当前已经设置的断点相关信息。
	tbreak [([*filename*:]*lineno* | *function*)[, *condition*]]	如果给出lineno参数或function参数，则设置一个临时断点。否则列出当前已经设置的临时断点相关信息。
	condition *bpnumber* [*condition*]	如果给出condition参数，则将指定断点的生效条件更新为该参数。否则删除指定断点的生效条件。
	commands [*bpnumber*]	为指定断点设置自动执行的pdb命令。省略bpnumber参数则作用于最后一个被设置的断点。
	disable [*bpnumber ...*]	如果给出bpnumber参数，则禁用指定的断点。否则禁用所有断点。
	enable [*bpnumber ...*]	如果给出bpnumber参数，则启用指定的断点。否则启用所有断点。
	ignore *bpnumber* [*count*]	将指定断点的忽略次数设置为count（count默认取0）。
	cl(ear) [*filename:lineno* | *bpnumber ...*]	如果给出filename:lineno参数或bpnumber参数，则清除指定的断点。否则清除所有断点。
额外语句	**[!]*statement***	在当前帧的上下文中执行statement参数指定的语句。
	interact	启动一个交互式解释器，以与用户交互。
	debug *code*	启动一个递归调试器，以调试code参数指定的代码。
别名	**alias [*name* [*command*]]**	定义和显示别名。

范畴	命令	说明
别名	**unalias** ***name***	删除别名。
帮助	**h(elp)** [***command***]	显示帮助信息。

使用pdb的第一步是知道如何退出，这是通过键入quit命令或Ctrl+D实现的，而被调试的程序也会自动停止运行（不论它被执行到了哪里）。而run命令和restart命令（两者等价）则在不退出pdb的前提下重新开始调试当前被调试的程序，且可以通过args参数改变启动被调试的程序时给出的参数。事实上，pdb启动时会自动执行run命令。请执行下列pdb命令和命令行来验证：

```
(Pdb) q

python3 -m pdb debug4.py
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) run
Restarting /Users/wwy/debug4.py with arguments:

> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) restart
Restarting /Users/wwy/debug4.py with arguments:

> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb)
```

16-6. 用pdb进行单步调试

（标准库：pdb）

属于表16-2中的“查看源代码”、“运行代码”、“检查对象”和“检查函数调用”这四个范畴的pdb命令提供了最基本的调试手段。本节将讨论它们。

进行调试时必须时刻清楚当前行的位置。所谓的当前行，也就是键入step命令后会被执行的行。虽然所有会导致当前行发生变化的pdb命令在反馈信息中都会标明新的当前行，而我们可以将它和脚本文件进行对比，但该反馈信息只包括当前行本身的内容，且不会显示当前行的行号，所以当调试比较长的脚本时通过反馈信息判断当前行的位置并不容易。此时就需要使用list命令或longlist命令。

list命令的参数存在如下组合：

➤ 没有给出参数：第一次执行显示当前行周围的11行（理想状态下当前行前后各5行，但当前行靠近脚本开头或末尾时做不到）。后续执行显示前一次list命令显示的行之后的11行。
➤ 仅给出了first参数：显示第first行周围的11行。特别的，first可以取值“.”以代表当前行。

➤ 给出了两个参数，且last≥first：显示第first行～第last行。
➤ 给出了两个参数，且last<first：显示第first行，以及后续的last行。

longlist命令则没有参数，显示的是当前帧绑定的代码对象的全部代码，也就是说如果当前行位于函数体或类体内，则仅会显示相应函数定义或类定义的全部代码；否则会显示当前模块（也就是整个脚本文件）的全部代码。

上述任何一种情况中，被显示的代码都会自动被添加行号，而如果包含当前行则会用“->”标明。

请执行下列pdb命令来验证（反馈信息略，↵代表按下Enter键）：

```
(Pdb) l
(Pdb) ↵
(Pdb) l 1
(Pdb) l
(Pdb) l 1, 20
(Pdb) l
(Pdb) l 10, 5
(Pdb) l
(Pdb) ll
```

step命令、next命令、until命令和return命令是执行被调试程序的基本方式，被统称为“单步调试（single-step debugging）”。

step命令会导致执行到下一个可停止的位置，一般而言是下一条语句，该语句的行号可能比当前行的行号小（例如当前行是某循环语句的循环体末尾），如果当前行是一次函数调用则停止在函数体的第一条语句处。

next命令与step命令的区别仅体现在函数调用中：当前行是一次函数调用的情况下，next命令会执行到该函数返回，而停止在函数调用后面那条语句。此外，如果执行停止在了某次函数调用刚刚返回的状态下，则使用step命令无法脱离该状态，应使用next命令，这将导致跳出该状态继续执行后续的程序。

until命令具有lineno参数时，会按照如下规则停止：

➤ 如果第lineno行位于一个函数体中，且目前该函数尚未定义，则执行到完成函数定义再停止。
➤ 如果第lineno行位于一个类体中，则执行完类定义再停止。
➤ 其他情况下，从第lineno行开始往下寻找第一条语句的开头，执行停止在该处。（如果第lineno行本身就是一条语句的开头，则停在第lineno行。）

而省略了lineno参数时，until命令会自动取当前行的行号加1。注意until命令必然导致当前行的行号递增，例如当前行是某循环语句的循环体末尾时，执行不带lineno参数的until命令会导致整个循环被执行完。这反过来要求lineno参数必须大于当前行的行号。

return命令使得执行到当前函数返回为止。需要注意的是，run命令和restart命令的本质是调用bdb模块中的run()，并执行到“exec(cmd, globals, locals)”语句（其中cmd参数会被传入被调试的代码，globals参数和locals参数则分别被传入run()被调用后的全局名字空间和本地名字空间）。这意味着对被调试程序的执行本身也被视为一次函数调用，因此即使没有调用被调试程序中的任何函数，依然可以执行return命令，只不过这将导致exec()返回，而run()在这种情况下继续执行会重新执行到“exec(cmd, globals, locals)”，即重新开始调试当前被调试的程序，等价于执行了run命令。

请执行下列pdb命令来验证：

```
(Pdb) run
Restarting /Users/wwy/debug4.py with arguments:

> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) s
> /Users/wwy/debug4.py(10)<module>()
-> import sys, pdb
(Pdb) ↵
> /Users/wwy/debug4.py(12)<module>()
-> __all__ = ['factorial1', 'factorial2']
(Pdb) ↵
> /Users/wwy/debug4.py(16)<module>()
-> def factorial1(n):
(Pdb) ↵
> /Users/wwy/debug4.py(28)<module>()
-> def factorial2(n):
(Pdb) unt 52
> /Users/wwy/debug4.py(52)<module>()
-> sys.exit(main())
(Pdb) s
--Call--
> /Users/wwy/debug4.py(39)main()
-> def main():
(Pdb) ↵
> /Users/wwy/debug4.py(42)main()
-> result = factorial1(3)
(Pdb) ↵
--Call--
> /Users/wwy/debug4.py(16)factorial1()
-> def factorial1(n):
(Pdb) n
> /Users/wwy/debug4.py(19)factorial1()
-> product = 1
(Pdb) ↵
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) r
--Return--
> /Users/wwy/debug4.py(24)factorial1()->6
-> return product
(Pdb) s
> /Users/wwy/debug4.py(43)main()
-> print(result)
```

```
(Pdb) n
6
> /Users/wwy/debug4.py(44)main()
-> result = factorial2(3)
(Pdb) ↵
> /Users/wwy/debug4.py(45)main()
-> print(result)
(Pdb) ↵
6
> /Users/wwy/debug4.py(47)main()
-> return 0
(Pdb) ↵
--Return--
> /Users/wwy/debug4.py(47)main()->0
-> return 0
(Pdb) ↵
SystemExit: 0
> /Users/wwy/debug4.py(52)<module>()
-> sys.exit(main())
(Pdb)
```

当通过单步调试（以及后面讨论的断点调试）使被调试的程序停止在仅执行了一部分的状态后，可以通过pdb命令whatis、p、pp和source检查被该程序定义和使用的那些标识符的状态。为了泛用性，这些pdb命令的参数都可以是一个（包含目标标识符）的表达式，当然最简单的表达式就仅具有一个标识符的名称。

whatis命令的功能是显示对象的类型，而p命令的功能则是显示对象本身，它们常结合在一起使用。pp命令是美化版的p命令，由于本书不讨论pprint模块，所以也不讨论pp命令。source命令用于显示对象关联的源代码，因此对expression表达式求值得到的对象必须是函数对象、方法对象、类对象、模块对象、回溯对象、帧对象或代码对象。

请执行下列pdb命令来验证（部分反馈信息被省略）：

```
(Pdb) run
(Pdb) unt 52
(Pdb) s
(Pdb) ↵
(Pdb) ↵
(Pdb) ↵
> /Users/wwy/debug4.py(19)factorial1()
-> product = 1
(Pdb) whatis n
<class 'int'>
(Pdb) p n
3
(Pdb) whatis factorial1
Function factorial1
(Pdb) p factorial1
<function factorial1 at 0x10567a4d0>
(Pdb) source factorial1
 16   def factorial1(n):
 17       """This function calculates n! using loop."""
 18
 19       product = 1
 20       while n > 1:
 21           #pdb.set_trace(header="start loop")
 22           product = product * n
```

```
 23           n = n-1
 24       return product
(Pdb)
```

而display命令和undisplay命令用于维护监视表达式列表。当通过单步调试（以及断点调试）使被调试的程序停止在仅执行了一部分的状态后，如果监视表达式列表中的某些表达式的值发生了变化，则会自动显示这些表达式。该机制比pdb命令whatis、p、pp和source更加智能。

请执行下列pdb命令来验证（部分反馈信息被省略）：

```
(Pdb) run
(Pdb) unt 52
(Pdb) s
(Pdb) ↵
(Pdb) ↵
(Pdb) ↵
(Pdb) ↵
(Pdb) ll
 16   def factorial1(n):
 17       """This function calculates n! using loop."""
 18
 19       product = 1
 20  ->   while n > 1:
 21           #pdb.set_trace(header="start loop")
 22           product = product * n
 23           n = n-1
 24       return product
(Pdb) display
Currently displaying:
(Pdb) display n
display n: 3
(Pdb) display product
display product: 1
(Pdb) display factorial1
display factorial1: <function factorial1 at 0x10132e710>
(Pdb) display
Currently displaying:
n: 3
product: 1
factorial1: <function factorial1 at 0x10132e710>
(Pdb) undisplay factorial1
(Pdb) display
Currently displaying:
n: 3
product: 1
(Pdb) s
(Pdb) ↵
> /Users/wwy/debug4.py(23)factorial1()
-> n = n-1
display product: 3  [old: 1]
(Pdb) ↵
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
display n: 2  [old: 3]
(Pdb) ↵
(Pdb) ↵
> /Users/wwy/debug4.py(23)factorial1()
-> n = n-1
```

```
display product: 6  [old: 3]
(Pdb) ↵
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
display n: 1  [old: 2]
(Pdb) ↵
(Pdb) ↵
--Return--
> /Users/wwy/debug4.py(24)factorial1()->6
-> return product
(Pdb)
```

Python程序的执行过程本质上是一系列的函数调用，因此检查函数调用对于调试相当有价值。下面讨论相关pdb命令。

where命令能够显示函数栈的当前状态，会输出多行信息。一般而言，函数栈中的每帧通过两行来描述，上面一行说明该帧是通过调用哪个函数产生的，下面一行（以“->”开头）说明即将执行该函数的函数体中的哪一行。特别的，如果该帧是调用eval()或exec()时额外产生的帧，则只用格式类似于“<string>(1)<module>()->3”的一行表示。

此外，where命令显示的信息中必然有某一帧的相关描述的首行以“>”开头，表明这是当前帧。当前帧只是在检查函数栈时定位用的，不会改变程序的执行状态，因此可以通过up命令和down命令自由地改变当前帧的位置。我们可以针对当前帧执行其他pdb命令，例如通过list或longlist显示当前帧对应的代码。

请执行下列pdb命令来验证（部分反馈信息被省略）：

```
(Pdb) run
(Pdb) w
  /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
bdb.py(597)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) unt 52
(Pdb) s
(Pdb) ↵
(Pdb) ↵
(Pdb) w
  /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
bdb.py(597)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()
  /Users/wwy/debug4.py(52)<module>()
-> sys.exit(main())
  /Users/wwy/debug4.py(42)main()
-> result = factorial1(3)
> /Users/wwy/debug4.py(16)factorial1()
-> def factorial1(n):
(Pdb) ll
 16  ->     def factorial1(n):
 17         """This function calculates n! using loop."""
 18
 19         product = 1
 20         while n > 1:
```

```
       21           #pdb.set_trace(header="start loop")
       22           product = product * n
       23           n = n-1
       24       return product
      (Pdb) u
      > /Users/wwy/debug4.py(42)main()
      -> result = factorial1(3)
      (Pdb) w
        /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
bdb.py(597)run()
      -> exec(cmd, globals, locals)
        <string>(1)<module>()
        /Users/wwy/debug4.py(52)<module>()
      -> sys.exit(main())
      > /Users/wwy/debug4.py(42)main()
      -> result = factorial1(3)
        /Users/wwy/debug4.py(16)factorial1()
      -> def factorial1(n):
      (Pdb) ll
       39   def main():
       40       """This function is called when this script runs in the top-level
environment."""
       41
       42  ->         result = factorial1(3)
       43       print(result)
       44       result = factorial2(3)
       45       print(result)
       46
       47       return 0
      (Pdb) u 3
      > /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
bdb.py(597)run()
      -> exec(cmd, globals, locals)
      (Pdb) w
      > /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
bdb.py(597)run()
      -> exec(cmd, globals, locals)
        <string>(1)<module>()
        /Users/wwy/debug4.py(52)<module>()
      -> sys.exit(main())
        /Users/wwy/debug4.py(42)main()
      -> result = factorial1(3)
        /Users/wwy/debug4.py(16)factorial1()
      -> def factorial1(n):
      (Pdb) l
      592           self.reset()
      593           if isinstance(cmd, str):
      594               cmd = compile(cmd, "<string>", "exec")
      595           sys.settrace(self.trace_dispatch)
      596           try:
      597  ->               exec(cmd, globals, locals)
      598           except BdbQuit:
      599               pass
      600           finally:
      601               self.quitting = True
      602               sys.settrace(None)
      (Pdb) d
      > <string>(1)<module>()
      (Pdb) w
        /Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
bdb.py(597)run()
      -> exec(cmd, globals, locals)
      > <string>(1)<module>()
        /Users/wwy/debug4.py(52)<module>()
      -> sys.exit(main())
```

```
  /Users/wwy/debug4.py(42)main()
-> result = factorial1(3)
  /Users/wwy/debug4.py(16)factorial1()
-> def factorial1(n):
(Pdb) ll
*** could not get source code
(Pdb)
```

args命令和retval命令是额外两个针对当前帧的pdb命令，前者显示调用相关函数时传入的参数，后者显示该函数返回时的返回值。需要注意的是，retval命令仅当当前帧进入返回状态但还没有被销毁时才有意义，而该状态通常是通过s命令进行单步调试才能达到的。

请执行下列pdb命令来验证（部分反馈信息被省略）：

```
(Pdb) run
(Pdb) unt 52
(Pdb) s
(Pdb) ↵
(Pdb) n
(Pdb) ↵
(Pdb) s
--Call--
> /Users/wwy/debug4.py(28)factorial2()
-> def factorial2(n):
(Pdb) args
n = 3
(Pdb) s
(Pdb) ↵
(Pdb) ↵
--Call--
> /Users/wwy/debug4.py(28)factorial2()
-> def factorial2(n):
(Pdb) args
n = 2
(Pdb) s
(Pdb) ↵
(Pdb) ↵
--Call--
> /Users/wwy/debug4.py(28)factorial2()
-> def factorial2(n):
(Pdb) args
n = 1
(Pdb) s
(Pdb) ↵
(Pdb) ↵
--Return--
> /Users/wwy/debug4.py(35)factorial2()->1
-> return 1
(Pdb) retval
1
(Pdb) s
--Return--
> /Users/wwy/debug4.py(33)factorial2()->2
-> return n * factorial2(n-1)
(Pdb) retval
2
(Pdb) s
--Return--
> /Users/wwy/debug4.py(33)factorial2()->6
```

```
-> return n * factorial2(n-1)
(Pdb) retval
6
(Pdb)
```

最后，jump命令使我们可以手工指定当前帧的执行顺序，并可以跳过一些语句。需要强调的是，该命令仅能在当前帧对应的代码中跳转，而不能跳入其他帧。

请执行下列pdb命令来验证（部分反馈信息被省略）：

```
(Pdb) run
(Pdb) unt 52
(Pdb) s
(Pdb) ↵
(Pdb) ↵
(Pdb) ↵
(Pdb) ↵
(Pdb) ll
 16   def factorial1(n):
 17       """This function calculates n! using loop."""
 18
 19       product = 1
 20  ->   while n > 1:
 21           #pdb.set_trace(header="start loop")
 22           product = product * n
 23           n = n-1
 24       return product
(Pdb) j 23
> /Users/wwy/debug4.py(23)factorial1()
-> n = n-1
(Pdb) s
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) ↵
> /Users/wwy/debug4.py(22)factorial1()
-> product = product * n
(Pdb) ↵
> /Users/wwy/debug4.py(23)factorial1()
-> n = n-1
(Pdb) p n
2
(Pdb) p product
2
(Pdb) j 19
> /Users/wwy/debug4.py(19)factorial1()
-> product = 1
(Pdb) s
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) ↵
> /Users/wwy/debug4.py(22)factorial1()
-> product = product * n
(Pdb) j 23
> /Users/wwy/debug4.py(23)factorial1()
-> n = n-1
(Pdb) s
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) ↵
> /Users/wwy/debug4.py(24)factorial1()
```

```
-> return product
(Pdb) p n
1
(Pdb) p product
1
(Pdb)
```

16-7. 用pdb进行断点调试

（标准库：pdb）

从上一节的讨论可以看出，单步调试比较麻烦，想要让程序运行到指定的位置时需要执行多条pdb命令。老练的编程者会更多地使用“断点调试（breakpoint debugging）”。下面将讨论断点调试相关pdb命令。

断点调试必然会用到的pdb命令是continue，它将使程序持续运行直到遇到一个断点或调用了sys.exit()。而pdb维护着一个断点列表，其余的断点调试pdb命令都是用于设置该列表的。

当不给出任何参数时，break命令会列出当前已经设置的所有断点的相关信息。当给出了参数时分如下三种情况：

- 给出了lineno参数但未给出filename参数：给当前文件的第lineno行设置一个断点。
- 同时给出了lineno参数和filename参数：给filename文件的第lineno行设置一个断点（filename文件可能尚未加载）。
- 给出了function参数：给function函数的函数体中的第一条语句设置一个断点。

对于上述任何一种情况，都可以通过condition参数设置该断点的生效条件，该参数是一个表达式，仅当对该表达式求值得到的对象的逻辑值检测结果为True时该断点才生效。我们可以给同一个位置设置多个断点，但这通常没有任何意义，因此应保证每个断点都指向代码的不同位置。

pdb模块维护着当前断点的最大编号n。n的初始值为0（代表没有断点）。每设置一个新断点就让其编号取n+1，然后n本身也增加1。这就使得断点的编号保持递增，新断点的编号总是大于已有的任何一个断点，哪怕某个曾经存在的断点被删除，相应编号也不会被重用。

请通过下面的命令行和pdb命令来验证结合使用break命令和continue命令进行断点调试的效果：

```
python3 -m pdb debug4.py
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) b
(Pdb) b 20
Breakpoint 1 at /Users/wwy/debug4.py:20
(Pdb) b factorial2, n == 1
```

```
Breakpoint 2 at /Users/wwy/debug4.py:28
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      stop only if n == 1
(Pdb) c
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) p n
3
(Pdb) c
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) p n
2
(Pdb) c
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) p n
1
(Pdb) c
6
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) p n
1
(Pdb) c
6
The program exited via sys.exit(). Exit status: 0
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 3 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      stop only if n == 1
      breakpoint already hit 3 times
(Pdb)
```

该例子说明了如下这些事实：

1. 当pdb刚启动时，断点列表是空的。

2. 在向断点列表添加了断点后，可以看到该断点的如下信息：断点号（Num）、断点类型（Type）、是否临时（Disp）、是否启用（Enb）、断点位置（Where）和断点生效条件（“stop only”行）。

3. 通过断点调试跟踪循环比通过单步调试跟踪循环要简单得多。只需要在循环语句的任意位置设置断点，一旦进入循环后，每次执行continue命令都会进行新一次循环，并停在相同的位置，直到该循环执行完成。

4. 如果一个断点被设置了生效条件，那么即使它被多次执行到，不满足生效条件的情况下程序也不会停止。

5. continue命令导致程序一直执行到sys.exit()后，会自动重新开始调试该程序。然而这不会导致断点列表被清空，反而可以通过break命令看到之前的调试过程中每个断点累计被执行到的总次数，即所谓的“命中次数”。

tbreak命令与break命令的语法完全相同，区别仅在于添加的断点是临时的，在命中一次之后就会自动被从断点列表中删除。请继续下面的pdb命令：

```
(Pdb) tbreak 43
Breakpoint 3 at /Users/wwy/debug4.py:43
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 3 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      stop only if n == 1
      breakpoint already hit 3 times
3   breakpoint   del  yes   at /Users/wwy/debug4.py:43
(Pdb) c
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) ↵
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) ↵
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) ↵
Deleted breakpoint 3 at /Users/wwy/debug4.py:43
> /Users/wwy/debug4.py(43)main()
-> print(result)
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 6 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      stop only if n == 1
      breakpoint already hit 3 times
(Pdb) tbreak 45
Breakpoint 4 at /Users/wwy/debug4.py:45
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 6 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      stop only if n == 1
      breakpoint already hit 3 times
4   breakpoint   del  yes   at /Users/wwy/debug4.py:45
(Pdb) c
6
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) ↵
Deleted breakpoint 4 at /Users/wwy/debug4.py:45
> /Users/wwy/debug4.py(45)main()
-> print(result)
(Pdb) tbreak
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 6 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      stop only if n == 1
      breakpoint already hit 6 times
(Pdb) c
6
The program exited via sys.exit(). Exit status: 0
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb)
```

该例子说明了即使一个断点被删除，它的断点号也不会被重用。注意断点列表的“Disp”列将为临时断点显示“del”而非“keep”。tbreak命令使我们在进行断点调试时可以根据需要临时性地添加一次性断点，这是非常好用的技巧。

condition命令可以改变已有断点的生效条件。注意如果不给出condition参数，则会使的原有生效条件被取消。请执行下列pdb命令：

```
(Pdb) condition 1 n > 3
New condition set for breakpoint 1.
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      stop only if n > 3
      breakpoint already hit 6 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      stop only if n == 1
      breakpoint already hit 6 times
(Pdb) condition 2
Breakpoint 2 is now unconditional.
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      stop only if n > 3
      breakpoint already hit 6 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      breakpoint already hit 6 times
(Pdb) c
6
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) p n
3
(Pdb) c
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) p n
2
(Pdb) c
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) p n
1
(Pdb) c
6
The program exited via sys.exit(). Exit status: 0
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb)
```

该例子给断点1添加了不可能满足的生效条件，使得该断点被直接跳过；而断点2的生效条件则被取消。

commands命令为指定断点设置自动执行的pdb命令。执行了commands命令后，提示符会从“(Pdb)”变成“(com)”，以表明后续键入的pdb命令是为断点设置的，而键入“end”表示结束输入。要取消某个断点自动执行的pdb命令，依然需要执行commands命

令，然后直接键入“end”即可。特别的，“silent”并不是一个pdb命令，但当添加了它时到达断点后调试器不再自动打印说明当前位置的信息。

请执行如下pdb命令：

```
(Pdb) condition 1
Breakpoint 1 is now unconditional.
(Pdb) commands 1
(com) display n
(com) display product
(com) silent
(com) end
(Pdb) b 24
Breakpoint 5 at /Users/wwy/debug4.py:24
(Pdb) commands
(com) undisplay
(com) end
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 9 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      breakpoint already hit 9 times
5   breakpoint   keep yes   at /Users/wwy/debug4.py:24
(Pdb)
```

该例子去掉了断点1的生效条件，并为其添加了两条display命令；然后添加了断点5，并为其添加了一条undisplay命令。注意在通过break命令或tbreak命令查看断点列表时，无法显示它们的自动执行pdb命令。

从上面的讨论可知，设置断点列表并不是一件容易的事。然而我们在调试过程中经常需要临时让某个断点不再生效。此时通过condition命令修改该断点的生效条件依然比较麻烦，更好的办法是使用disable命令和enable命令，它们可以有选择性的禁用/启用某些断点，也可以一次性禁用/启用所有断点。

请执行如下pdb命令：

```
(Pdb) disable 2
Disabled breakpoint 2 at /Users/wwy/debug4.py:28
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 9 times
2   breakpoint   keep no    at /Users/wwy/debug4.py:28
      breakpoint already hit 9 times
5   breakpoint   keep yes   at /Users/wwy/debug4.py:24
(Pdb) c
display n: 3
display product: 1
(Pdb) ↵
display n: 2
display product: 3
(Pdb) ↵
display n: 1
```

```
display product: 6
(Pdb) ↵
> /Users/wwy/debug4.py(24)factorial1()
-> return product
(Pdb) display
Currently displaying:
(Pdb) c
6
6
The program exited via sys.exit(). Exit status: 0
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) enable 2
Enabled breakpoint 2 at /Users/wwy/debug4.py:28
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 12 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      breakpoint already hit 9 times
5   breakpoint   keep yes   at /Users/wwy/debug4.py:24
      breakpoint already hit 1 time
(Pdb)
```

该例子先禁用了断点2，然后断点调试完了整个程序，在重新开始调试后又启用了断点2。注意在断点列表中被禁用断点的“Enb”列显示“no”。

如果我们预先知道某次调试中某个断点会被命中多少次，则可以通过ignore命令指定忽略该断点的次数，这样就可以代替联合使用disable命令和enable命令。如果ignore命令没有参数，则取消指定断点的忽略次数。当一个断点的忽略次数不为0时，查看断点列表时会通过“ignore”行说明。

请执行如下pdb命令：

```
(Pdb) ignore 2 2
Will ignore next 2 crossings of breakpoint 2.
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 12 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      ignore next 2 hits
      breakpoint already hit 9 times
5   breakpoint   keep yes   at /Users/wwy/debug4.py:24
      breakpoint already hit 1 time
(Pdb) ignore 2
Will stop next time breakpoint 2 is reached.
(Pdb) b
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 12 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      breakpoint already hit 9 times
5   breakpoint   keep yes   at /Users/wwy/debug4.py:24
      breakpoint already hit 1 time
(Pdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.
(Pdb) b
```

```
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      ignore next 2 hits
      breakpoint already hit 12 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      breakpoint already hit 9 times
5   breakpoint   keep yes   at /Users/wwy/debug4.py:24
      breakpoint already hit 1 time
(Pdb) c
display n: 1
display product: 6
(Pdb) c
> /Users/wwy/debug4.py(24)factorial1()
-> return product
(Pdb) c
6
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) c
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) c
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) c
6
The program exited via sys.exit(). Exit status: 0
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb)
```

该例子忽略了断点1的接下来两次命中，因此直接停在了第三次循环。

最后需要强调，如果断点列表不为空，即使进行单步调试遇到断点时也会停止。当然我们可以通过临时禁用所有断点来解决这个问题，但有时候我们更希望清空断点列表。从上面的例子可以看出，断点列表是独立于调试过程的，不能通过run命令或restart命令重置。当然，如果重新启动了pdb，则断点列表会被自动清空，但这是很笨拙的办法。另一方面，如果我们确定某个断点不再需要，则期望从断点列表中删除该断点。

清理断点列表的正规方法是使用clear命令，当存在参数时分如下两种情况：

- 同时给出了lineno参数和filename参数：清除filename文件的第lineno行的所有断点。
- 给出了bpnumber参数：清除断点号为bpnumber的断点。

而如果没有给出任何参数，则clear命令会清空断点列表，但在清空前会询问用户以求确认。

请执行如下pdb命令：

```
(Pdb) clear 5
Deleted breakpoint 5 at /Users/wwy/debug4.py:24
(Pdb) b
Num Type         Disp Enb   Where
```

```
1   breakpoint   keep yes   at /Users/wwy/debug4.py:20
      breakpoint already hit 15 times
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      breakpoint already hit 12 times
(Pdb) clear debug4.py:20
Deleted breakpoint 1 at /Users/wwy/debug4.py:20
(Pdb) b
Num Type         Disp Enb   Where
2   breakpoint   keep yes   at /Users/wwy/debug4.py:28
      breakpoint already hit 12 times
(Pdb) clear
Clear all breaks? y
Deleted breakpoint 2 at /Users/wwy/debug4.py:28
(Pdb) b
(Pdb)
```

16-8. 其他pdb命令

（标准库：pdb）

在“(Pdb)”提示符下，当输入的内容不能被识别为一条pdb命令时，就会被当成一条Python语句，并在当前帧的上下文中执行。然而为了明确地指定一条Python语句，我们可以在它前面添加“!”。

请执行如下命令行和pdb命令：

```
$ python3 -m pdb debug4.py
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) tbreak 20
Breakpoint 1 at /Users/wwy/debug4.py:20
(Pdb) c
Deleted breakpoint 1 at /Users/wwy/debug4.py:20
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) !n = 5
(Pdb) c
120
6
The program exited via sys.exit(). Exit status: 0
> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb)
```

在该例子中，我们在脚本的第20行设置了一个临时断点，并在脚本执行到该断点时，通过“!n = 5”插入了Python语句“n = 5”，这使得对factorial1()的调用返回了5! = 120，而非3! = 6。

上述方式只能一次插入一条Python语句，比较繁琐。当我们想要在脚本执行到某个断点时查看所有的本地变量/全局变量，使用interatct命令要更加方便，因为它会启动一个交互式Python解释器，并将该脚本的所有本地变量和全局变量拷贝到当前全局名字空间中的同名变量。退出该交互式Python解释器的方法是输入EOF（即Ctrl + D），这会导致回到原

“(Pdb)”提示符，而在该交互式Python解释器下的操作不会对脚本的运行状态造成任何影响。请继续执行下列pdb命令：

```
(Pdb) run
Restarting /Users/wwy/debug4.py with arguments:

> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) tbreak 20
Breakpoint 2 at /Users/wwy/debug4.py:20
(Pdb) c
Deleted breakpoint 2 at /Users/wwy/debug4.py:20
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) interact
*interactive*
>>> n
3
>>> product
1
>>> factorial1
<function factorial1 at 0x104546520>
>>> factorial2
<function factorial2 at 0x104546660>
>>> main
<function main at 0x1045465c0>
>>> __name__
'__main__'
>>> ^D
now exiting InteractiveConsole...
(Pdb)
```

在该例子中，启动交互式Python解释器后，可以查看本地变量n和product，以及全局变量factorial1、factorial2、main和__name__的值。

debug命令会启动一个递归调试器，以调试通过code参数指定的代码。而在该代码执行完成后，递归调试器会自动退出，回到原调试状态。请继续执行下列pdb命令：

```
(Pdb) run
Restarting /Users/wwy/debug4.py with arguments:

> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) tbreak 20
Breakpoint 4 at /Users/wwy/debug4.py:20
(Pdb) c
Deleted breakpoint 4 at /Users/wwy/debug4.py:20
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) ll
 16   def factorial1(n):
 17       """This function calculates n! using loop."""
 18
 19       product = 1
 20  ->   while n > 1:
 21           #pdb.set_trace(header="start loop")
 22           product = product * n
```

```
 23            n = n-1
 24        return product
(Pdb) debug factorial2(2)
ENTERING RECURSIVE DEBUGGER
> <string>(1)<module>()
((Pdb)) ll
*** could not get source code
((Pdb)) s
--Call--
> /Users/wwy/debug4.py(28)factorial2()
-> def factorial2(n):
((Pdb)) ll
 28  ->     def factorial2(n):
 29        """This function calculates n! using recursion."""
 30
 31        #breakpoint()
 32        if n > 1:
 33            return n * factorial2(n-1)
 34        else:
 35            return 1
((Pdb)) n
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
((Pdb)) r
--Return--
> /Users/wwy/debug4.py(33)factorial2()->2
-> return n * factorial2(n-1)
((Pdb)) n
--Return--
> <string>(1)<module>()->None
((Pdb)) ll
*** could not get source code
((Pdb)) n
LEAVING RECURSIVE DEBUGGER
(Pdb) ll
 16   def factorial1(n):
 17        """This function calculates n! using loop."""
 18
 19        product = 1
 20  ->    while n > 1:
 21            #pdb.set_trace(header="start loop")
 22            product = product * n
 23            n = n-1
 24        return product
(Pdb)
```

alias命令和unalias命令使我们可以为其他pdb命令（以及Python语句）定义别名。当不给alias提供任何参数时，会列出当前定义的所有别名；当只给出了name参数时，会显示name是哪条pdb命令的别名；当同时给出了name参数和command参数时，为command命令创建别名name，而command命令的参数可以通过%1，%2，……等表示，或用%*表示任意参数。而unalias命令则取消别名name。请继续执行下列pdb命令：

```
(Pdb) run
Restarting /Users/wwy/debug4.py with arguments:

> /Users/wwy/debug4.py(5)<module>()
-> """This module illustrates how to debug with pdb.
(Pdb) alias tb tbreak %*
(Pdb) alias tb
tb = tbreak %*
```

```
(Pdb) alias go continue
(Pdb) alias go
go = continue
(Pdb) alias
go = continue
tb = tbreak %*
(Pdb) tb 20
Breakpoint 9 at /Users/wwy/debug4.py:20
(Pdb) tb
Num Type         Disp Enb   Where
9   breakpoint   del  yes   at /Users/wwy/debug4.py:20
(Pdb) go
Deleted breakpoint 9 at /Users/wwy/debug4.py:20
> /Users/wwy/debug4.py(20)factorial1()
-> while n > 1:
(Pdb) unalias go
(Pdb) alias
tb = tbreak %*
(Pdb) unalias tb
(Pdb) alias
(Pdb)
```

help命令使我们能在“(Pdb)”提示符下查看在线帮助手册。没有给出command参数时，help命令将列出在线帮助手册有哪些主题。而当以其中某个主题作为command参数时，则会显示该主题的相关信息。请继续执行下列pdb命令：

```
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF    c          d        h         list      q        rv       undisplay
a      cl         debug    help      ll        quit     s        unt
alias  clear      disable  ignore    longlist  r        source   until
args   commands   display  interact  n         restart  step     up
b      condition  down     j         next      return   tbreak   w
break  cont       enable   jump      p         retval   u        whatis
bt     continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb) h run
run [args...]
        Restart the debugged python program. If a string is supplied
        it is split with "shlex", and the result is used as the new
        sys.argv.  History, breakpoints, actions and debugger options
        are preserved.  "restart" is an alias for "run".
(Pdb) h exec
(!) statement
        Execute the (one-line) statement in the context of the current
        stack frame.  The exclamation point can be omitted unless the
        first word of the statement resembles a debugger command.  To
        assign to a global variable you must always prefix the command
        with a 'global' command, e.g.:
        (Pdb) global list_options; list_options = ['-l']
        (Pdb)
(Pdb)
```

最后值得一提的是，我们可以在工作目录下创建.pdbrc文件，并在其内包含一些pdb命令（例如alias命令），这样当pdb启动时，会先自动执行该.pdbrc文件包含的pdb命令。

16-9. 通过脚本启动调试器

（标准库：内置函数、pdb、sys）

前面的讨论都只涉及pdb的第一种用法，即通过-m选项直接执行该模块。而如果我们通过import语句导入pdb模块，则可以使用它定义的函数和类。下面先讨论pdb定义的函数。

pdb.run()和pdb.runeval()分别用于调试一段代码和一个表达式，其语法分别为：

```
pdb.run(object, globals=None, locals=None)
pdb.runeval(expression, globals=None, locals=None)
```

pdb.run()的功能其实是启动一个调试器，并将其参数传入exec()以执行；而pdb.runeval()的功能则是启动一个调试器，并将其参数传入eval()以执行。

下面的例子说明了这两个函数的用法：

```
$ python3

>>> import debug4
>>> import pdb
>>> pdb.run('debug4.main()')
> <string>(1)<module>()
(Pdb) s
--Call--
> /Users/wwy/debug4.py(39)main()
-> def main():
(Pdb) ll
 39  ->     def main():
 40         """This function is called when this script runs in the top-level
environment."""
 41
 42         result = factorial1(3)
 43         print(result)
 44         result = factorial2(3)
 45         print(result)
 46
 47         return 0
(Pdb) q
>>> pdb.runeval('debug4.factorial1(3)')
> <string>(1)<module>()
(Pdb) s
--Call--
> /Users/wwy/debug4.py(16)factorial1()
-> def factorial1(n):
(Pdb) ll
 16  ->     def factorial1(n):
 17         """This function calculates n! using loop."""
 18
 19         product = 1
```

```
 20       while n > 1:
 21           #pdb.set_trace(header="start loop")
 22           product = product * n
 23           n = n-1
 24       return product
(Pdb) q
>>>
```

注意对于函数调用来说，既可以被exec()执行，也可以被eval()执行，所以在调试函数调用时pdb.run()和pdb.runcall()没有区别。

pdb.runcall()专门用于调试一次函数调用，其语法为：

```
pdb.runcall(function, *args, **kwargs)
```

其中function参数用于传入函数名，args和kwargs则用于传入实际参数。

请继续上面的例子，通过如下语句验证pdb.runcall()的用法：

```
>>> pdb.runcall(debug4.factorial2, 3)
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) ll
 28   def factorial2(n):
 29       """This function calculates n! using recursion."""
 30
 31       #breakpoint()
 32  ->   if n > 1:
 33           return n * factorial2(n-1)
 34       else:
 35           return 1
(Pdb) q
>>>
```

pdb.set_trace()的功能则是将一个断点硬编码到脚本中，使得该脚本执行到这里时自动启动pdb并进入调试状态。该函数的语法为：

```
pdb.set_trace(*, header=None)
```

如果header参数被传入了一个None之外的对象，则在进入调试状态后会自动将其显示到标准输出，相当于执行了“print(header)”。

请将debug4.py中第21行的“#”去掉，即在该位置插入如下语句：

```
pdb.set_trace(header="start loop")
```

然后通过如下命令行来验证：

```
$ python3 debug4.py
start loop
> /Users/wwy/debug4.py(22)factorial1()
-> product = product * n
(Pdb) ll
 16   def factorial1(n):
 17       """This function calculates n! using loop."""
 18
 19       product = 1
 20       while n > 1:
 21           pdb.set_trace(header="start loop")
 22  ->             product = product * n
 23           n = n-1
 24       return product
(Pdb)
```

此时debug4.py会在执行到第21行时自动进入调试状态。

pdb模块定义的Pdb类，其实也只是上述4个函数的面向对象接口，其实例化语法为：

```
class pdb.Pdb(completekey='tab', stdin=None, stdout=None, skip=None, nosigint=False, readrc=True)
```

Pdb类继承了cmd模块定义的Cmd类，其所有参数都会原封不动第传递给cmd.Cmd类，本书不深入讨论。Pdb对象具有属性run()、runeval()、runcall()和set_trace()，分别等价于上述四个函数。

除了上述四个函数，pdb模块还提供了两个函数来支持抛出异常时的“事后调试（post-mortem debugging）”。第一个函数是pdb.post_mortem()，其语法为：

```
pdb.post_mortem(traceback=None)
```

如果给traceback参数传入了一个回溯对象，则启动调试器后会针对该回溯对象进行调试。而如果省略了traceback参数，则pdb.post_mortem()只能在try语句的except子句中被调用，并自动传入与当前正被处理异常相关连的回溯对象。

下面的例子利用第8章中的例子exception1.py说明了如何用pdb.post_mortem()进行事后调试：

```
$ python3

>>> import pdb
>>> try:
```

```
...     import exception1
... except Exception:
...     pdb.post_mortem()
...
> /Users/wwy/exception1.py(2)f()
-> return 'str:' + a
(Pdb) ll
  1   def f(a):
  2  ->         return 'str:' + a
(Pdb) q
>>>
```

第二个函数是pdb.pm()，其语法为：

```
pdb.pm()
```

它总是会针对sys.last_traceback引用的回溯对象进行调试，所以不需要传入回溯对象，也不局限于在try语句的except子句中被调用。

下面的例子同样利用exception1.py来说明如何用pdb.pm()进行事后调试：

```
$ python3

>>> import pdb
>>> import exception1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/exception1.py", line 4, in <module>
    f(1)
    ^^^^
  File "/Users/wwy/exception1.py", line 2, in f
    return 'str:' + a
           ~~~~~~~^~~
TypeError: can only concatenate str (not "int") to str
>>> pdb.pm()
> /Users/wwy/exception1.py(2)f()
-> return 'str:' + a
(Pdb) ll
  1   def f(a):
  2  ->         return 'str:' + a
(Pdb) q
>>>
```

至此我们已经讨论完了pdb模块提供的所有调试功能。这里需要补充说明的是，相较于使用pdb.set_trace()将一个断点硬编码到脚本中，不如使用内置函数breakpoint()，因为后者更加灵活。breakpoint()的语法为：

```
breakpoint(*args, **kwargs)
```

它其实会在内部调用sys.breakpointhook()。

表16-3列出了与调试相关的sys属性。sys.breakpointhook()总是由breakpoint()来调用的，会在内部调用一个函数，通过它来启动某个调试器。而breakpoint()被传入的实际参也会原封不动地被传入该函数。sys.breakpointhook()的返回值也会被作为breakpoint()的返回值。

表16-3. 调试相关sys属性

属性	说明
sys.breakpointhook()	由breakpoint()调用，在内部调用一个函数以启动某调试器。
sys.__breakpointhook__	当sys.breakpointhook()被改写时，引用其初始值。

sys.breakpointhook()的行为是这样的：它会先查找环境变量PYTHONBREAKPOINT，然后根据它的值决定后续的行为：

- 值为0：直接返回，以使断点失效。
- 值为“package.module.function”形式的字符串：自动加载指定的模块，然后调用指定的函数。（这通常会启动一个第三方调试器。）
- 值为空串：调用pdb.set_trace()。

而当PYTHONBREAKPOINT不存在时，也调用pdb.set_trace()。

综上所述，如果脚本通过breakpoint()硬编码了断点，则可以通过PYTHONBREAKPOINT来控制是否启动调试器，以及启动哪个调试器。请将debug4.py的21行重新注释掉，然后去掉第31行开头的“#”，使其成为通过breakpoint()插入的一个断点。然后通过如下命令行来验证：

```
$ python3 debug4.py
6
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb) ll
 28   def factorial2(n):
 29       """This function calculates n! using recursion."""
 30
 31       breakpoint()
 32  ->   if n > 1:
 33           return n * factorial2(n-1)
 34       else:
 35           return 1
(Pdb)

$ export PYTHONBREAKPOINT=0
$ python3 debug4.py
6
6
```

```
$ export PYTHONBREAKPOINT=""
$ python3 debug4.py
6
> /Users/wwy/debug4.py(32)factorial2()
-> if n > 1:
(Pdb)
```

请注意PYTHONBREAKPOINT环境变量对结果的影响。

16-10. 收集确定性剖析数据

（标准库：Python Profilers分析器）

现在让我们把注意力从调试转到分析。对软件进行性能分析的技术非常多，就算局限于Python标准库，也存在很多模块可用于分析Python脚本的性能，包括inspect、gc、traceback、trace、tracemalloc、timeit、faulthandler和logging。本书不可能详细讨论所有这些模块，更不可能涉及第三方工具。本书主要讨论cProfile扩展模块和其伴随模块profile的使用，附带介绍通过一些sys属性分析内存使用情况的技巧。

cProfile和profile对Python脚本进行“确定性剖析（deterministic profiling）”，即在脚本运行过程中监控函数调用、函数返回和抛出异常这三种事件，并记录下每个事件发生的时间。这就获得了针对该程序的运行过程的统计数据，然后这些数据将通过pstats模块提供的Stats类来分析。注意pstats与cProfile和profile是紧密联系的。

确定性剖析本质上是对程序运行过程中各函数被调用次数的计数。这些统计数据可以被用于达到如下目的：

➤ 查找“漏洞（bug）”：如果某个函数被调用的次数不符合预期，则表明与之相关的代码有很大概率存在漏洞。
➤ 探查“内联展开点（inline-expassion points）”：内联函数通常有很高的调用频率。但内联函数是C中的概念，仅当Python脚本的运行会访问扩展模块时才有可能涉及。
➤ 发现“热回路（hot loop）：热回路其实时模拟电路中的概念，这里指代程序被频繁执行的部分，对它们进行优化的收益是最大的。
➤ 判断算法的选择是否正确：通过对比实现相同功能但采用不同算法的函数的累积时间，可以比较这些算法用于解决当前问题时的效率，进而选择最合适的算法。

确定性剖析最大的问题是，为了记录被监控事件发生的时间，需要执行额外的代码，而这会给程序的运行带来额外开销，使得统计数据存在系统性偏差。与确定性剖析相对的是“统计剖析（statistical profiling）”，它通过随机采样“有效指令指针（effective instraction pointer）”来估算程序各部分的耗时程度。统计剖析的优点是带来的额外开销极小，因此统计数据的系统性偏差也极小；缺点则是统计数据本身远没有确定性剖析的精确，只能作为一种较模糊的提示。

Python的标准库只提供了实现确定性剖析的扩展模块cProfile和其伴随模块profile。这是因为Python脚本是被解释执行的，速度本身就比直接执行机器代码慢几个数量级。Python解释器在运行时自动为每个事件都提供了一个钩子函数，基于它们来记录函数调用、函数返

回和抛出异常的时间的额外开销相对较小，尤其是使用cProfile时该额外开销几乎可以忽略。在这种情况下，使用统计剖析的意义已经不大了。

需要强调的是，Python的确定性剖析存在下列局限性：

➤ cProfile和profile使用的默认计时器的精度是1ms，因此被记录的函数调用、函数返回和抛出异常的时间无法比1ms更加精确。然而很多简单函数调用的执行时间都不到1ms。如果一个函数被调用的次数足够多，可以通过求平均值的方法减少误差，但该方法并不通用。
➤ 记录函数调用、函数返回和抛出异常的时间的额外开销虽然很小，但可以累积，因此如果一个函数被调用的次数足够多，则会导致这种额外开销累积到不再能忽视。
因此，确定性剖析获得统计数据并不是准确的，不能代替“基准测试”。

此外，profile记录函数调用、函数返回和抛出异常的时间的额外开销要比cProfile大很多，不能忽略。为了提高统计数据的准确度，使用profile时最好先获取一个修正常数，即额外开销的平均耗费时间，然后profile记录的每个时间都将减去该修正常数，但这可能导致部分时间被记录为负数。一般而言，我们都会使用cProfile来进行确定性剖析，profile主要供希望编写自己的确定性剖析模块的用户参考。

下面介绍cProfile和profile的用法。最基本的使用方式是以指定的代码为参数调用它们定义的run()函数或runctx()函数，其语法分别为：

```
    profile.run(object, filename=None, sort=SortKey.STDNAME)
    profile.runctx(object, globals, locals, filename=None,
sort=SortKey.STDNAME)
```

注意该语法中“profile”既代表profile模块，也代表cProfile模块。调用run()时相当于执行了exec(object, __main__.__dict__, __main__.__dict__)；而调用runctx()时相当于执行了exec(object, globals, locals)。filename参数可以被传入一个代表文件路径的字符串，这样获得的统计数据就会以二进制的形式被保存到该路径指定的文件中，以供创建Stats对象时使用。如果filename参数被传入None，则会自动创建一个Stats对象来分析获得的统计数据，并基于这些数据形成一个表格。然后自动以sort参数传入的对象为实际参数调用Stats对象的sort_stats()，以对该表格中的行进行排序。最后自动调用Stats对象的print_stats()，以将该表格写入标准输出来显示。

为了说明如何进行性能分析，请将如下代码保存到profile1.py：

```
import time


def factorial1(n):
    """This function calculates n! using loop."""

    product = 1
    while n > 1:
```

```
            time.sleep(0.01)
            product = product * n
            n = n-1
        return product


    def factorial2(n):
        """This function calculates n! using recursion."""

        time.sleep(0.01)
        if n > 1:
            return n * factorial2(n-1)
        else:
            return 1
```

注意该例子其实就是debug4.py中的函数factorial1()和factorial2()，但通过在函数体内调用time.sleep()人为增加了调用它们所耗费的时间。请执行如下命令行和语句：

```
    $ python3

    >>> from profile1 import factorial1, factorial2
    >>> import cProfile
    >>> cProfile.run("factorial1(6)")
             9 function calls in 0.057 seconds

       Ordered by: standard name

       ncalls  tottime  percall  cumtime  percall filename:lineno(function)
            1    0.000    0.000    0.056    0.056 <string>:1(<module>)
            1    0.000    0.000    0.056    0.056 profile1.py:4(factorial1)
            1    0.000    0.000    0.057    0.057 {built-in method
builtins.exec}
            5    0.056    0.011    0.056    0.011 {built-in method time.sleep}
            1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}


    >>> cProfile.run("factorial2(6)")
             15 function calls (10 primitive calls) in 0.072 seconds

       Ordered by: standard name

       ncalls  tottime  percall  cumtime  percall filename:lineno(function)
            1    0.000    0.000    0.072    0.072 <string>:1(<module>)
          6/1    0.000    0.000    0.072    0.072 profile1.py:15(factorial2)
            1    0.000    0.000    0.072    0.072 {built-in method
builtins.exec}
            6    0.072    0.012    0.072    0.012 {built-in method time.sleep}
            1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}


    >>> cProfile.run("factorial1(6)", "profile_factorial1")
    >>> cProfile.run("factorial2(6)", "profile_factorial2")
    >>>
```

下面分析该结果。当没有指定文件路径时，会直接在标准输入显示如下信息：

1. 第一行为共计统计信息，包括函数调用总数（若包含递归调用，则会以“xx primitive calls”来标明原始调用数），以及总耗时。

2. 第二行表明表格基于什么排序。
3. 接下来是一个具有6列的表格，各列的含义如表16-4所示。

表16-4. 确定性剖析分析表格中各列的含义

列	说明
ncalls	该函数被调用的次数，如果是n/m格式，则表示被调用了n次，其中原始调用m次，递归调用n-m次。
tottime	执行该函数的函数体的代码（不包括子调用）耗费的总时间。
percall（第一个）	tottime除以ncalls的值。
cumtime	执行该函数的函数体的代码（包括子调用）耗费的总时间。
percall（第二个）	cumtime除以ncalls的值。
filename:lineno(function)	被调用函数的标准名，格式如该列的列名所示。但当调用的是内置函数或标准库中的函数时，该列将是用大括号括起立的说明信息。

当指定了文件路径时，会将统计数据保存到指定文件。对于上面的例子来说，将得到二进制文件profile_factorial1和profile_factorial2。

16-11. 分析确定性剖析数据

（标准库：Python Profilers分析器）

接下来讨论用于分析通过cProfile或profile得到的统计数据的Stats类，其实例化语法为：

```
class pstats.Stats(*filenames, stream=sys.stdout)
```

其中filenames参数可以是任意个（包括0个）指向储存有通过cProfile或profile得到统计数据的二进制文件的路径，以及Profile对象（将在后面讨论），而新建Stats对象会将所有这些数据源的数据整合起来。stream参数则用于指定调用Stats对象的print_stats()属性时，将相应的表格写入哪个文件，默认取标准输出。

表16-5总结了Stats对象的属性。下面依次讨论。

表16-5. Stats对象的属性

列	说明
print_stats()	将表格写入stream参数指定的文件。
print_callers()	将函数间的调用关系写入stream参数指定的文件，基于调用者整理。
print_callees()	将函数间的调用关系写入stream参数指定的文件，基于被调用者整理。
add()	整合来自其他文件的统计数据。

列	说明
strip_dirs()	去掉标准名中filename的目录部分，仅保留文件名。
dump_stats()	将当前整合的所有统计数据保存到指定的文件。
sort_stats()	将表格中的行按照指定的方式重新排序。
reverse_order()	将表格中的行的排列顺序反转。

print_stats()是输出分析结果的基本方式，其语法为：

```
Stats.print_stats(*restrictions)
```

其中restrictions参数可以接受如下类型的参数：

- 一个正整数n，表明只显示表格中的前n行。
- 一个0.0~1.0之间的浮点数p，表明只显示前100*p%行。
- 一个字符串s，表示只显示标准名中包含s的行。

注意1和2是相互矛盾的，但它们都能与3组合，而组合时会按照它们出现的顺序依次对行进行筛选。如果restrictions参数被省略，则显示整个表格。

下面的例子创建了一个Stats对象以整合profile_factorial1和profile_factorial2中的数据，然后输出了整个表格：

```
>>> import pstats
>>> sta1 = pstats.Stats("profile_factorial1", "profile_factorial2")
>>> sta1.print_stats()
Wed Aug 24 18:42:48 2022    profile_factorial1
Wed Aug 24 18:42:58 2022    profile_factorial2

         24 function calls (19 primitive calls) in 0.126 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
       11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
        1    0.000    0.000    0.057    0.057 /Users/wwy/
profile1.py:4(factorial1)
        2    0.000    0.000    0.126    0.063 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
      6/1    0.000    0.000    0.070    0.070 /Users/wwy/
profile1.py:15(factorial2)


<pstats.Stats object at 0x10f5e9010>
```

```
>>>
```

注意此时print_stats()的输出会先说明相应Stats对象的数据来源，在该例子中列出了两个文件的文件名，以及它们被创建的时间。而由于合并了两个文件的数据，所以采用的是随机排序。

下面的例子说明了给出限制参数3后的效果：

```
>>> sta1.print_stats(3)
Wed Aug 24 18:42:48 2022    profile_factorial1
Wed Aug 24 18:42:58 2022    profile_factorial2

         24 function calls (19 primitive calls) in 0.126 seconds

   Random listing order was used
   List reduced from 6 to 3 due to restriction <3>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
       11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
        1    0.000    0.000    0.057    0.057 /Users/wwy/
profile1.py:4(factorial1)


<pstats.Stats object at 0x10f5e9010>
>>>
```

下面的例子说明了给出限制参数0.3后的效果：

```
>>> sta1.print_stats(0.3)
Wed Aug 24 18:42:48 2022    profile_factorial1
Wed Aug 24 18:42:58 2022    profile_factorial2

         24 function calls (19 primitive calls) in 0.126 seconds

   Random listing order was used
   List reduced from 6 to 2 due to restriction <0.3>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
       11    0.126    0.011    0.126    0.011 {built-in method time.sleep}


<pstats.Stats object at 0x10f5e9010>
>>>
```

下面的例子说明了给出限制参数“factorial”后的效果：

```
>>> sta1.print_stats("factorial")
Wed Aug 24 18:42:48 2022    profile_factorial1
Wed Aug 24 18:42:58 2022    profile_factorial2
```

```
              24 function calls (19 primitive calls) in 0.126 seconds

         Random listing order was used
         List reduced from 6 to 2 due to restriction <'factorial'>

         ncalls  tottime  percall  cumtime  percall filename:lineno(function)
              1    0.000    0.000    0.057    0.057 /Users/wwy/
profile1.py:4(factorial1)
            6/1    0.000    0.000    0.070    0.070 /Users/wwy/
profile1.py:15(factorial2)


      <pstats.Stats object at 0x10f5e9010>
      >>>
```

下面的例子说明了给出限制参数3，“factorial”后的效果：

```
      >>> sta1.print_stats(3, "factorial")
      Wed Aug 24 18:42:48 2022    profile_factorial1
      Wed Aug 24 18:42:58 2022    profile_factorial2

               24 function calls (19 primitive calls) in 0.126 seconds

         Random listing order was used
         List reduced from 6 to 3 due to restriction <3>
         List reduced from 3 to 1 due to restriction <'factorial'>

         ncalls  tottime  percall  cumtime  percall filename:lineno(function)
              1    0.000    0.000    0.057    0.057 /Users/wwy/
profile1.py:4(factorial1)


      <pstats.Stats object at 0x10f5e9010>
```

而下面的例子则说明交换限制参数3和“factorial”的位置将得到不同结果：

```
      >>> sta1.print_stats("factorial", 3)
      Wed Aug 24 18:42:48 2022    profile_factorial1
      Wed Aug 24 18:42:58 2022    profile_factorial2

               24 function calls (19 primitive calls) in 0.126 seconds

         Random listing order was used
         List reduced from 6 to 2 due to restriction <'factorial'>

         ncalls  tottime  percall  cumtime  percall filename:lineno(function)
              1    0.000    0.000    0.057    0.057 /Users/wwy/
profile1.py:4(factorial1)
            6/1    0.000    0.000    0.070    0.070 /Users/wwy/
profile1.py:15(factorial2)


      <pstats.Stats object at 0x10f5e9010>
      >>>
```

print_callers()和print_callees()都用于反映函数间的调用关系，其语法分别为：

```
Stats.print_callers(*restrictions)
Stats.print_callees(*restrictions)
```

其中restrictions参数的含义与在print_stats()中相同。两者的区别是，print_callers()以调用者为目标进行整理，即列出每个被调用者，后面跟“<-”，然后是该被调用者的所有调用者（可能有多个），且为每个调用者列出ncalls、tottime和cumtime这三个统计数据。而print_callee()则以被调用者为目标进行整理，即列出每个调用者，后面跟“->”，然后是该调用者的所有被调用者（可能有多个），且也为每个被调用者列出ncalls、tottime和cumtime这三个统计数据。

下面的例子说明了print_callers()输出信息的格式：

```
      >>> sta1.print_callers()
         Random listing order was used

      Function                                          was called by...
                                                            ncalls  tottime
cumtime
      {built-in method builtins.exec}                   <-
      {built-in method time.sleep}                      <-       5    0.057
0.057  /Users/wwy/profile1.py:4(factorial1)
                                                                6    0.069
0.069  /Users/wwy/profile1.py:15(factorial2)
      /Users/wwy/profile1.py:4(factorial1)              <-       1    0.000
0.057  <string>:1(<module>)
      <string>:1(<module>)                              <-       2    0.000
0.126  {built-in method builtins.exec}
      {method 'disable' of '_lsprof.Profiler' objects}  <-
      /Users/wwy/profile1.py:15(factorial2)             <-     5/1    0.000
0.057  /Users/wwy/profile1.py:15(factorial2)
                                                                1    0.000
0.070  <string>:1(<module>)


      <pstats.Stats object at 0x10f5e9010>
      >>>
```

而下面的例子则说明了print_callee()输出信息的格式：

```
      >>> sta1.print_callees()
         Random listing order was used

      Function                                          called...
                                                            ncalls  tottime
cumtime
      {built-in method builtins.exec}                   ->       2    0.000
0.126  <string>:1(<module>)
      {built-in method time.sleep}                      ->
      /Users/wwy/profile1.py:4(factorial1)              ->       5    0.057
0.057  {built-in method time.sleep}
      <string>:1(<module>)                              ->       1    0.000
0.057  /Users/wwy/profile1.py:4(factorial1)
```

```
                                                          1    0.000
0.070  /Users/wwy/profile1.py:15(factorial2)
      {method 'disable' of '_lsprof.Profiler' objects}  ->
      /Users/wwy/profile1.py:15(factorial2)             ->     5/1    0.000
0.057  /Users/wwy/profile1.py:15(factorial2)
                                                          6    0.069
0.069  {built-in method time.sleep}


      <pstats.Stats object at 0x10f5e9010>
      >>>
```

从这两个结果都可以看出，time.sleep()被factorial1()调用了5次，被factorial2()调用了6次。

add()的功能是向已有的Stats对象添加统计数据，其语法为：

```
Stats.add(*filenames)
```

但这里的filenames参数只能被传入文件路径， 不能传入Profile对象。

下面的例子创建一个空的Stats对象，依次加载了profile_factorial1和profile_factorial2中的统计数据：

```
      >>> sta2 = pstats.Stats()
      >>> sta2.print_stats()
               0 function calls in 0.000 seconds

      <pstats.Stats object at 0x10f7c5c10>
      >>> sta2.add("profile_factorial1")
      <pstats.Stats object at 0x10f7c5c10>
      >>> sta2.print_stats()
      Wed Aug 24 18:42:48 2022    profile_factorial1

               9 function calls in 0.057 seconds

         Random listing order was used

         ncalls  tottime  percall  cumtime  percall filename:lineno(function)
              1    0.000    0.000    0.057    0.057 {built-in method
builtins.exec}
              5    0.057    0.011    0.057    0.011 {built-in method time.sleep}
              1    0.000    0.000    0.057    0.057 /Users/wwy/
profile1.py:4(factorial1)
              1    0.000    0.000    0.057    0.057 <string>:1(<module>)
              1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}


      <pstats.Stats object at 0x10f7c5c10>
      >>> sta2.add("profile_factorial2")
      <pstats.Stats object at 0x10f7c5c10>
      >>> sta2.print_stats()
      Wed Aug 24 18:42:48 2022    profile_factorial1
      Wed Aug 24 18:42:58 2022    profile_factorial2
```

```
              24 function calls (19 primitive calls) in 0.126 seconds

         Random listing order was used

         ncalls  tottime  percall  cumtime  percall filename:lineno(function)
              2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
             11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
              1    0.000    0.000    0.057    0.057 /Users/wwy/
profile1.py:4(factorial1)
              2    0.000    0.000    0.126    0.063 <string>:1(<module>)
              2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
            6/1    0.000    0.000    0.070    0.070 /Users/wwy/
profile1.py:15(factorial2)


      <pstats.Stats object at 0x10f7c5c10>
      >>>
```

strip_dirs()的语法为：

Stats.strip_dirs()

它只会影响表格的filename:lineno(function)列，使得其内各行在该列的值的filename部分只包含文件名，不包含目录部分。注意调用该属性后，Stats对象中储存的统计数据会丢失关于目录的信息，这是不可逆的。此外，如果表格是基于标准名排序的，则会变成随机排序。

下面的例子说明了strip_dirs()的作用：

```
      >>> sta2.strip_dirs()
      <pstats.Stats object at 0x10f7c5c10>
      >>> sta2.print_stats()
      Wed Aug 24 18:42:48 2022    profile_factorial1
      Wed Aug 24 18:42:58 2022    profile_factorial2

               24 function calls (19 primitive calls) in 0.126 seconds

         Random listing order was used

         ncalls  tottime  percall  cumtime  percall filename:lineno(function)
              2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
             11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
              1    0.000    0.000    0.057    0.057 profile1.py:4(factorial1)
              2    0.000    0.000    0.126    0.063 <string>:1(<module>)
              2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
            6/1    0.000    0.000    0.070    0.070 profile1.py:15(factorial2)


      <pstats.Stats object at 0x10f7c5c10>
      >>>
```

请特别关注对应factorial1()和factorial2()的行在filename:lineno(function)列的变化。

dump_stats()将Stats对象中的统计数据导出到文件，其语法为：

```
Stats.dump_stats(filename)
```

注意如果filename指定的文件已经存在，则它会被覆盖。

下面的例子将sta2中整合后再去掉了目录信息的数据保存到了profile_factorial12，然后再通过sta3读取它：

```
>>> sta2.dump_stats("profile_factorial12")
>>> sta3 = pstats.Stats("profile_factorial12")
>>> sta3.print_stats()
Thu Aug 25 15:15:26 2022    profile_factorial12

         24 function calls (19 primitive calls) in 0.126 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
       11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
        1    0.000    0.000    0.057    0.057 profile1.py:4(factorial1)
        2    0.000    0.000    0.126    0.063 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
      6/1    0.000    0.000    0.070    0.070 profile1.py:15(factorial2)


<pstats.Stats object at 0x10f87b490>
>>>
```

注意该结果说明profile_factorial12中已经不包含与目录相关的信息。

sort_stats()的功能是控制Stats对象生成的表格的排序方式，其语法为：

```
Stats.sort_stats(*keys)
```

其中keys参数可以取表16-6列出的预定义常量或字符串（推荐使用常量，因为它们的含义更加确定）。注意keys参数可以接受多个常量/字符串，此时会按照它们被传入的次序依次应用相应排序规则。特别的，SortKey.NFL就等价于（SortKey.NAME, SortKey.FILENAME, SortKey.LINE），即先基于函数名排序，当函数名相同时基于文件名排序，而当函数名和文件名都相同时基于行号排序。除了SortKey.Name、SortKey.LINE和SortKey.FILENAME会按照升序排列外，其余常量都按照降序排列。与strip_dirs()类似，sort_stats()会改变Stats对象中存储的统计数据。从run()函数和runctx()函数的sort参数的默认实参值可知，通过它们得到的统计数据默认是基于标准名排序的。

表16-6. Stats.sort_stats可取参数

常量	字符串	说明
SortKey.CALLS	**"calls"、"ncalls"**	基于调用次数（ncalls列中的n）排序。
SortKey.PCALLS	**"pcalls"**	基于原始调用次数（ncalls列中的m）排序。
SortKey.TIME	**"time"、"tottime"**	基于不包括子调用的耗费总时间（tottime列）排序。
SortKey.CUMULATIVE	**"cumulative"、"cumtime"**	基于包括子调用的耗费总时间（cumtime列）排序。
SortKey.STDNAME	**"stdname"**	基于标准名（filename:lineno(function)列）排序。
SortKey.FILENAME	**"filename"、"file"、"module"**	基于文件名（filename）排序。
SortKey.LINE	**"line"**	基于行号（lineno）排序。
SortKey.NAME	**"name"**	基于函数名（function）排序。
SortKey.NFL	**"nfl"**	基于（函数名，文件名，行号）排序。

下面的例子说明了SortKey.STDNAME和SortKey.NFL的区别：

```
     >>> sta3.sort_stats(pstats.SortKey.STDNAME)
     <pstats.Stats object at 0x10f87b490>
     >>> sta3.print_stats()
     Thu Aug 25 15:15:26 2022    profile_factorial12

              24 function calls (19 primitive calls) in 0.126 seconds

        Ordered by: standard name

        ncalls  tottime  percall  cumtime  percall filename:lineno(function)
             2    0.000    0.000    0.126    0.063 <string>:1(<module>)
           6/1    0.000    0.000    0.070    0.070 profile1.py:15(factorial2)
             1    0.000    0.000    0.057    0.057 profile1.py:4(factorial1)
             2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
            11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
             2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}


     <pstats.Stats object at 0x10f87b490>
     >>> sta3.sort_stats(pstats.SortKey.NFL)
     <pstats.Stats object at 0x10f87b490>
     >>> sta3.print_stats()
     Thu Aug 25 15:15:26 2022    profile_factorial12

              24 function calls (19 primitive calls) in 0.126 seconds

        Ordered by: name/file/line

        ncalls  tottime  percall  cumtime  percall filename:lineno(function)
             2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
```

```
             11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
              2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
              2    0.000    0.000    0.126    0.063 <string>:1(<module>)
              1    0.000    0.000    0.057    0.057 profile1.py:4(factorial1)
            6/1    0.000    0.000    0.070    0.070 profile1.py:15(factorial2)


      <pstats.Stats object at 0x10f87b490>
      >>>
```

下面的例子说明了传入多个SortKey系列常量时，它们的排列顺序会影响结果：

```
      >>> sta3.sort_stats(pstats.SortKey.CUMULATIVE, pstats.SortKey.CALLS)
      <pstats.Stats object at 0x10f87b490>
      >>> sta3.print_stats()
      Thu Aug 25 15:15:26 2022    profile_factorial12

               24 function calls (19 primitive calls) in 0.126 seconds

         Ordered by: cumulative time, call count

         ncalls  tottime  percall  cumtime  percall filename:lineno(function)
              2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
              2    0.000    0.000    0.126    0.063 <string>:1(<module>)
             11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
            6/1    0.000    0.000    0.070    0.070 profile1.py:15(factorial2)
              1    0.000    0.000    0.057    0.057 profile1.py:4(factorial1)
              2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}


      <pstats.Stats object at 0x10f87b490>
      >>> sta3.sort_stats(pstats.SortKey.CALLS, pstats.SortKey.CUMULATIVE)
      <pstats.Stats object at 0x10f87b490>
      >>> sta3.print_stats()
      Thu Aug 25 15:15:26 2022    profile_factorial12

               24 function calls (19 primitive calls) in 0.126 seconds

         Ordered by: call count, cumulative time

         ncalls  tottime  percall  cumtime  percall filename:lineno(function)
             11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
            6/1    0.000    0.000    0.070    0.070 profile1.py:15(factorial2)
              2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
              2    0.000    0.000    0.126    0.063 <string>:1(<module>)
              2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
              1    0.000    0.000    0.057    0.057 profile1.py:4(factorial1)


      <pstats.Stats object at 0x10f87b490>
      >>>
```

最后，reverse_order()的功能是反转Stats对象生成的表格的排序顺序，其语法为：

```
Stats.reverse_order()
```

它同样会改变Stats对象中存储的统计数据。

下面的例子说明了reverse_order()的作用：

```
>>> sta3.reverse_order()
<pstats.Stats object at 0x10f87b490>
>>> sta3.print_stats()
Thu Aug 25 15:15:26 2022    profile_factorial12

         24 function calls (19 primitive calls) in 0.126 seconds

   Ordered by: call count, cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.057    0.057 profile1.py:4(factorial1)
        2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
        2    0.000    0.000    0.126    0.063 <string>:1(<module>)
        2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
      6/1    0.000    0.000    0.070    0.070 profile1.py:15(factorial2)
       11    0.126    0.011    0.126    0.011 {built-in method time.sleep}


<pstats.Stats object at 0x10f87b490>
>>> sta3.reverse_order()
<pstats.Stats object at 0x10f87b490>
>>> sta3.print_stats()
Thu Aug 25 15:15:26 2022    profile_factorial12

         24 function calls (19 primitive calls) in 0.126 seconds

   Ordered by: call count, cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       11    0.126    0.011    0.126    0.011 {built-in method time.sleep}
      6/1    0.000    0.000    0.070    0.070 profile1.py:15(factorial2)
        2    0.000    0.000    0.126    0.063 {built-in method
builtins.exec}
        2    0.000    0.000    0.126    0.063 <string>:1(<module>)
        2    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
        1    0.000    0.000    0.057    0.057 profile1.py:4(factorial1)


<pstats.Stats object at 0x10f87b490>
>>>
```

16-12. 确定性剖析的精确控制

（标准库：Python Profilers分析器）

至此，你已经掌握了用cProfile或profile对Python脚本进行性能分析的基本技术。如果你想对统计信息的收集进行更精确的控制，则应使用cProfile或profile提供的Profile类，其实例化语法为：

```
class profile.Profile(timer=None, timeunit=0.0, subcalls=True, builtins=True)
```

我们可以通过timer参数指定使用自定义计时器来获取时间，它应是一个返回数值的函数。这些自定义计时器通常会被设计为具有更高的精度，例如使用time.perf_counter()或者time.perf_counter_ns()。当该自定义计时器返回一个整数时，可以通过timeunit参数来指定该整数的单位是多少秒，例如0.000001表示单位是微秒。如果subcalls参数是False，则意味着表格中只包括那些没有调用者的函数调用。如果builtins参数是False，则意味着表格中不包括对内置函数或标准库中函数的调用。

表16-7列出了Profile对象具有的属性。run()属性和runctx()属性和其同名函数的功能几乎是一样的，区别仅在于统计数据会保存到Profile对象中。runcall()属性则专门用于分析一次函数调用，同样将统计数据保存到Profile对象中。print_stats()属性会以该Profile对象为参数隐式创建一个Stats对象，然后调用后者的sort_stats()属性以完成排序，最后调用后者的print_stats()以输出表格。dump_stats()属性的功能与在Stats对象中相同。Profile对象还可以被当成上下文管理器来使用，而create_stats()属性则用于控制对哪些代码进行性能分析。enable()和disable()只有cProfile支持，其用法是先调用前者，再调用后者，以对两者间的代码进行性能分析。

表16-7. Profile对象的属性

属性	说明
`run()`	执行指定的代码，并保存获得的统计数据。
`runctx()`	执行指定的代码，可以指定全局/本地名字空间，并保存获得的统计数据。
`runcall()`	调用指定的函数，并保存获得的统计数据。
`print_stats()`	隐式创建一个Stats对象来分析获得的统计数据，然后将表格写入标准输出。
`dump_stats()`	将统计数据保存到指定的文件。
`create_stats()`	停止收集统计数据。
`enable()`	开始收集统计数据。仅cProfile支持。
`disable()`	停止收集统计数据。仅cProfile支持。

runcall()是Profile对象特有的，其语法为：

```
Profile.runcall(func, /, *args, **kwargs)
```

相当于收集func(*args, **kwargs)的统计数据。

Profile对象的print_stats()与Stats对象的print_stats()语法不同，其语法为：

```
Profile.print_stats(sort=SortKey.STDNAME)
```

注意它不具有用于筛选行的restrictions参数，而sort参数则被用于指定表格中的行的排序方法。

下面的例子创建了一个Profile对象，使用time.perf_counter_ns()为计时器，忽略对内置函数和标准库中函数的调用，通过runcall()先后收集“factorial1(6)”和“factorial2(6)”的统计数据，并通过print_stats()显示分析结果：

```
$ python3

>>> from profile1 import factorial1, factorial2, time
>>> import cProfile
>>> from pstats import SortKey
>>> pro = cProfile.Profile(time.perf_counter_ns, 0.000000001, True, False)
>>> pro.runcall(factorial1, 6)
720
>>> pro.print_stats()
         1 function calls in 0.057 seconds

   Ordered by: name/file/line

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.057    0.057    0.057    0.057 profile1.py:4(factorial1)


>>> pro.runcall(factorial2, 6)
720
>>> pro.print_stats(SortKey.NFL)
         7 function calls (2 primitive calls) in 0.125 seconds

   Ordered by: name/file/line

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.057    0.057    0.057    0.057 profile1.py:4(factorial1)
      6/1    0.068    0.011    0.068    0.068 profile1.py:15(factorial2)


>>> pro.print_stats(SortKey.TIME)
         7 function calls (2 primitive calls) in 0.125 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      6/1    0.068    0.011    0.068    0.068 profile1.py:15(factorial2)
        1    0.057    0.057    0.057    0.057 profile1.py:4(factorial1)


>>>
```

下面的例子说明了如何将Profile对象当成上下文管理器来收集统计数据，以及在这一过程中create_stats()的作用。同时它也说明了如何将Profile对象中的统计数据导出到文件：

```
    from profile1 import factorial1, factorial2
    import cProfile, pstats

    #将Profile对象当成上下文管理器。
    with cProfile.Profile(builtins=False) as pro:
        #从这里开始收集统计数据。
        factorial2(6)
        pro.create_stats()
        #在这里终止收集统计数据。
        factorial1(6)

    #将收集到的统计数据保存到文件。
    pro.dump_stats("profile_factorial21")

    #创建一个Stats对象，并显示分析结果。
    sta = pstats.Stats("profile_factorial21")
    sta.strip_dirs()
    sta.print_callers()
```

请将上述代码保存为profile2.py，然后通过如下命令行验证：

```
    $ python3 profile2.py
       Random listing order was used

    Function                        was called by...
                                        ncalls  tottime  cumtime
    cProfile.py:50(create_stats)  <-
    profile1.py:15(factorial2)    <-     5/1    0.058    0.058
profile1.py:15(factorial2)
```

接下来注释掉“pro.create_stats()”行，然后再次验证：

```
    $ python3 profile2.py
       Random listing order was used

    Function                      was called by...
                                      ncalls  tottime  cumtime
    profile1.py:4(factorial1)   <-
    profile1.py:15(factorial2)  <-     5/1    0.055    0.055
profile1.py:15(factorial2)
    cProfile.py:117(__exit__)   <-
```

从上述结果可以看出，当将Profile对象当成上下文管理器来使用时，默认收集整个with语句的代码块以及调用Profile对象的__exit__这段代码的统计数据，但可以通过在with语句的代码块中插入对create_stats()的调用来提前终止收集。

下面的例子说明了如何使用Profile对象的enable()和disable()来收集统计数据。它同时说明了Profile对象可以直接作为Stats类实例化时的filenames参数：

```
from profile1 import factorial2
import cProfile, pstats

#创建一个空的Profile对象。
pro = cProfile.Profile(builtins=False)

pro.enable()
#从这里开始收集统计数据。
factorial2(6)
#在这里终止收集统计数据。
pro.disable()

#创建一个Stats对象，并显示分析结果。    注意这里直接以Profile对象为参数。
sta = pstats.Stats(pro)
sta.strip_dirs()
sta.print_callees()
```

请将上述代码保存为profile3.py，然后通过如下命令行验证：

```
$ python3 profile3.py
   Random listing order was used

Function                        called...
                                    ncalls  tottime  cumtime
profile1.py:15(factorial2)  ->      5/1    0.060    0.060
profile1.py:15(factorial2)
```

16-13. 分析内存使用情况

（标准库：sys）

确定性剖析存在一个缺陷，即只能分析程序的执行速度，却不能分析程序对内存的使用情况。本节介绍两个可用于分析内存使用情况的sys属性，它们被总结在表16-8中。

表16-8. 分析内存使用情况相关sys属性

属性	说明
sys.getsizeof()	返回指定对象占用的字节数。
sys.getallocatedblocks()	返回解释器当前已分配的内存块数。

sys.getsizeof()能够返回任何对象在内存中占用的字节数，其语法为：

```
sys.getsizeof(obj[, default])
```

对于CPython来说，该函数会调用obj参数传入对象的函数属性__sizeof__以获得所需信息，如果该对象并非通过第三方扩展模块定义的类创建，那么该结果将是准确的。如果该对象不具有__sizeof__，那么给default参数传入了一个对象时就返回该对象作为默认值，否则会抛出TypeError异常。其它Python解释器对sys.getsizeof()的实现可能采用其他方式，因此其准确性也不能保证。

注意在Python中，一个对象经常会通过属性引用其他对象。而在计算该对象占用的内存时，只会计入每个引用占用的字节数，而不会计入被引用对象占用的字节数。如果想知道一个对象和它所引用的所有对象占用的总字节数，则需要编写一个函数手工跟踪该对象引用的所有对象，以及后者引用引用的所有对象，……依此类推。通常，该函数应使用递归以简化编写难度。

sys.getallocatedblocks()则不针对某个特定的对象，而是计算解释器当前分配的内存块的数量，其语法为：

```
sys.getallocatedblocks()
```

Python解释器在启动过程中会分配大量内存块，而在执行脚本的过程中也会动态分配内存块。这些内存块的大小与操作系统有关，但必然是512B的倍数（在Unix和类Unix操作系统上通常是4096B）。

以内存块为单位显然不如以字节为单位精确，但由于sys.getallocatedblocks()不需要指定对象，所以能更好地发现“内存泄漏（memory leak）”的情况。我们已经知道，Python解释器会通过垃圾回收机制自动管理内存，因此发生内存泄漏的概率是很小的。然而任何解释器的垃圾回收机制都不是完美的，而有许许多多复杂的情况可能绕过垃圾回收机制导致内存泄漏。举例来说，当解释器使用的垃圾回收机制不够智能，无法识别循环引用时，只要对象A和对象B相互引用对方，就永远不会被销毁。（CPython已经较好地解决了循环引用的问题，但依旧存在无法解决的其他问题，例如使用某些调试工具，或通过try语句处理异常时都有可能导致某些本该被销毁的对象继续存活。）

当通过sys.getallocatedblocks()发现Python解释器分配的内存块数量持续增加就可以判断发生了内存泄漏，不需要特别精确。此时应使用gc模块来进一步分析导致内存泄露的原因是什么，但对该模块的讨论超出了本书的范围。如果想通过sys.getallocatedblocks()跟踪一个脚本执行过程，则需要更精确的结果，此时通常需要调用sys._clear_type_cache()来清空解释器内部的类型缓存，以避免它对sys.getallocatedblocks()结果的影响。但一般而言，sys._clear_type_cache()只应由Python解释器在内部自动调用。

下面通过一个例子来说明如何通过sys.getsizeof()和sys.getallocatedblocks()来跟踪脚本运行过程中内存的使用情况：

```
import functools
import sys
```

```
#定义通过递归计算斐波那契数列的函数。
def fibonacci(n):
    if n < 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)


#生成斐波那契数列的前15项，并记录内存使用情况。
seq = []
n = 0
#清理内部的类型缓存。
sys._clear_type_cache()
#显示初始状态下数列的值、占用内存和已分配块数。
print(f'seq: {seq}')
print(f'seq size: {sys.getsizeof(seq)}')
print(f'block number: {sys.getallocatedblocks()}')
while n < 15:
    #向数列中插入一项。
    seq.append(fibonacci(n))
    #显示当前状态下数列的值、占用内存和已分配块数。
    print(f'seq: {seq}')
    print(f'seq size: {sys.getsizeof(seq)}')
    print(f'block number: {sys.getallocatedblocks()}')
    n += 1
```

请将上述代码保存为profile4.py，然后通过如下命令行验证：

```
$ python3 -i profile4.py
seq: []
seq size: 56
block number: 26114
seq: [1]
seq size: 88
block number: 26115
seq: [1, 1]
seq size: 88
block number: 26115
seq: [1, 1, 2]
seq size: 88
block number: 26115
seq: [1, 1, 2, 3]
seq size: 88
block number: 26115
seq: [1, 1, 2, 3, 5]
seq size: 120
block number: 26115
seq: [1, 1, 2, 3, 5, 8]
seq size: 120
block number: 26115
seq: [1, 1, 2, 3, 5, 8, 13]
seq size: 120
block number: 26115
seq: [1, 1, 2, 3, 5, 8, 13, 21]
seq size: 120
block number: 26115
seq: [1, 1, 2, 3, 5, 8, 13, 21, 34]
seq size: 184
block number: 26115
seq: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
seq size: 184
```

```
block number: 26115
seq: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
seq size: 184
block number: 26115
seq: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
seq size: 184
block number: 26115
seq: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
seq size: 184
block number: 26115
seq: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
seq size: 184
block number: 26116
seq: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
seq size: 184
block number: 26117
>>> sys.getallocatedblocks()
26072
>>>
```

从该结果可以看出如下三点：

1. Python解释器启动后就已经分配了26000多个内存块。
2. 调用递归函数fibonacci()时，随着传入参数的增大，需要为其动态分配的内存块的数量也变多，但调用完后这些内存块会被自动回收。
3. 为列表seq动态添加元素时，其占用的内存最初以32B为单位，后来变成以64B为单位，但一个较小的数字占不满32B或64B内存空间，因此插入几个数字后占用内存才增长一次。

16-14. 调试和分析异步程序

（标准库：asyncio、sys）

本章到目前为止所有的讨论都是以同步程序为例子的。事实上这些技巧也可用于调试和分析异步程序。

下面以async_generator.py为例说明如何用pdb调试异步程序：

```
$ python3 -m pdb async_generator.py
> /Users/wwy/async_generator.py(1)<module>()
-> import random
(Pdb) b 8
Breakpoint 1 at /Users/wwy/async_generator.py:8
(Pdb) c
ag1:
<class 'async_generator'>
> /Users/wwy/async_generator.py(8)async_gen1()
-> await asyncio.sleep(0.3)
(Pdb) n
> /Users/wwy/async_generator.py(9)async_gen1()
-> print("A")
(Pdb)
A
> /Users/wwy/async_generator.py(10)async_gen1()
-> yield 1
(Pdb)
Internal StopIteration: 1
```

```
> /Users/wwy/async_generator.py(78)main()
-> print(await anext(ag1))
(Pdb) r
1
Internal StopIteration
> /Users/wwy/async_generator.py(11)async_gen1()
-> await asyncio.sleep(0.3)
(Pdb) n
> /Users/wwy/async_generator.py(12)async_gen1()
-> print("B")
(Pdb)
B
> /Users/wwy/async_generator.py(13)async_gen1()
-> yield 2
(Pdb)
Internal StopIteration: 2
> /Users/wwy/async_generator.py(78)main()
-> print(await anext(ag1))
(Pdb) r
2
Internal StopIteration
> /Users/wwy/async_generator.py(14)async_gen1()
-> await asyncio.sleep(0.3)
(Pdb) q
```

可以看出用pdb调试异步程序的方法与调试同步程序相同。

下面则以async_generator.py为例说明如何用cProfile分析异步程序：

```
$ python3

>>> from async_generator import main, asyncio
>>> import cProfile
>>> cProfile.run("asyncio.run(main())")
...
```

注意由于本例子输出的信息太多，所以被省略。

然而相对于同步程序而言，异步程序要复杂得多，除了前面介绍的技巧外，还需要一些专门的技巧来调试和分析异步程序。本章的最后一节将介绍这些技巧。

首先，前面已经提到了“异步I/O调试模式”，而开启它的方法包括：

- 在启动Python解释器前将环境变量PYTHONASYNCIODEBUG设置为1。（启动开发模式会自动执行这一步骤。）
- 调用asyncio.run()时给debug参数传入True。
- 实例化asyncio.Runner类时以True作为debug参数。

当开启了异步I/O调试模式之后，会增加如下额外操作：

- 当执行非线程安全的异步API时，如果从错误的线程调用，则抛出异常。
- 自动检查是否存在未被await的可等待对象，如果有则将其记录到日志。
- 如果某异步I/O操作的执行时间太长，则将其记录到日志。
- 如果某回调函数的执行时间太长，则将其记录到日志。

而Python日志的设置是通过logging模块完成的，对其的详细讨论超出了本书的范围。

然后，第14章已经提到协程具有cr_origin属性，事实上该属性是用于支持“协程溯源（coroutine origin tracking）”的。我们可以通过表16-9列出的sys属性来控制协程溯源的深度：当深度为默认值0时，cr_origin会引用None；而当深度为一个正整数n时，cr_origin会引用一个至多具有n个元素的元组，每个元素都是一个格式为“（filename, line_number, function_name）”的元组，指向了一条与创建该协程相关的语句，而这些元素会按照这些语句被执行的顺序排序，这样我们就可以了解到该协程是如何被创建的。显然，启用了协程溯源后每个协程都会占用更多内存，因此该功能只应被用于调试和分析。

表16-9. 协程溯源相关sys属性

属性	说明
sys.set_coroutine_origin_tracking_depth()	接受一个表示协程溯源深度的整数。0表示禁用。
sys.get_coroutine_origin_tracking_depth()	获得当前协程溯源的深度。

需要强调的是，仅当一个协程被包装为任务来调度时才能对其进行协程溯源，否则不论协程溯源的深度被设置为多少，该协程的cr_origin都总是引用None。而第14章也提到了，可以通过Task对象的get_coro()属性取得它包装的协程，该函数的语法为：

```
Task.get_coro()
```

下面的例子说明了如何使用协程溯源：

```
#!/usr/bin/env python3

import asyncio
import sys


async def coro():
    pass


async def main():
    if len(sys.argv) < 2:
        sys.exit(1)
    try:
        #根据第一个命令行参数设置协程溯源的深度。
```

```
            d = int(sys.argv[1])
            sys.set_coroutine_origin_tracking_depth(d)
            #创建一个任务并执行。
            task = asyncio.create_task(coro())
            await task
            #取得该任务包装的协程。
            cr = task.get_coro()
            #取得并显示当前协程溯源的深度。
            otd = sys.get_coroutine_origin_tracking_depth()
            print(f"origin tracking depth: {otd}")
            #显示协程溯源信息。
            print(f"cr_origin: {cr.cr_origin}")
            sys.exit(0)
        except Exception:
            sys.exit(1)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为async_debug1.py，然后通过如下命令行验证：

```
    $ python3 async_debug1.py 0
    origin tracking depth: 0
    cr_origin: None

    $ python3 async_debug1.py 1
    origin tracking depth: 1
    cr_origin: (('/Users/wwy/async_debug1.py', 19, 'main'),)

    $ python3 async_debug1.py 2
    origin tracking depth: 2
    cr_origin: (('/Users/wwy/async_debug1.py', 19, 'main'), ('/Library/
Frameworks/Python.framework/Versions/3.11/lib/python3.11/asyncio/events.py', 80,
'_run'))

    $ python3 async_debug1.py 3
    origin tracking depth: 3
    cr_origin: (('/Users/wwy/async_debug1.py', 19, 'main'), ('/Library/
Frameworks/Python.framework/Versions/3.11/lib/python3.11/asyncio/events.py', 80,
'_run'), ('/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
asyncio/base_events.py', 1909, '_run_once'))
```

最后，第14章提到的Task对象的属性get_stack()和print_stack()可用于分析它所包装的协程关联的栈。get_stack()的语法为：

```
Task.get_stack(*, limit=None)
```

它总是返回一个列表，而列表中的元素按照如下规则设置：

➤ 当任务处于“未完成”状态，而协程尚未运行或被挂起时，列表将按照从旧到新的顺序包含关联到该协程的栈的帧对象。
➤ 当任务处于“已完成”状态且协程没有抛出异常，或者任务处于“已取消”状态时，列表将是空的。
➤ 当任务处于“已完成”状态且协程抛出了异常时，列表将按照从旧到新的顺序包含关联到被抛出异常的回溯对象引用的帧对象。

而如果给limit参数传入一个正整数n，则列表至多包含n个帧对象，而从第n+1个帧对象开始直到最新的帧对象都被忽略。

print_stack()的功能则是将通过get_stack()获得的栈框架或回溯框架转化为更易理解的信息来显示，其语法为：

```
Task.print_stack(*, limit=None, file=None)
```

其中file参数用于指定将信息写入哪个文件，传入None则默认写入标准出错。该函数会在内部调用get_stack()，而通过limit参数传入的对象则会被直接传给get_stack()。特别的，对于规则3中的情况，print_stack()将显示被抛出异常的回溯信息。

下面的例子说明了如何使用Task对象的上述属性：

```
#!/usr/bin/env python3

import asyncio
import sys

#该全局变量控制着协程是否抛出异常。
fail = False
#fail = True


#该协程可能执行完成，也可能抛出异常。
async def coro():
    if fail:
        raise RuntimeError()


#该函数分析任务包装协程的栈。
def show_stack(task, l):
    #获得该任务包装协程的栈框架并显示。
    stack = task.get_stack(limit=l)
    print(f"{task.get_name()}'s stack frames:")
    print(stack)
    print("")
    #显示该任务包装协程的栈框架相关信息。
    print(f"{task.get_name()}'s stack info:")
    task.print_stack(limit=l)
    print("")
```

```
    async def main():
        #创建任务task1，但未执行。
        task1 = asyncio.create_task(coro(), name="task1")
        show_stack(task1, 3)
        #执行该任务到它正常返回或抛出异常。
        try:
            await task1
            show_stack(task1, 3)
        except Exception:
            show_stack(task1, 3)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为async_debug2.py，然后通过如下命令行验证：

```
    $ python3 async_debug2.py
    task1's stack frames:
    [<frame at 0x10eebd000, file '/Library/Frameworks/Python.framework/
Versions/3.11/lib/python3.11/asyncio/events.py', line 80, code _run>, <frame at
0x10ee5f400, file '/Users/wwy/async_debug2.py', line 33, code main>, <frame at
0x10ef58a40, file '/Users/wwy/async_debug2.py', line 12, code coro>]

    task1's stack info:
    Stack for <Task pending name='task1' coro=<coro() running at /Users/wwy/
async_debug2.py:12>> (most recent call last):
      File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
asyncio/events.py", line 80, in _run
        self._context.run(self._callback, *self._args)
      File "/Users/wwy/async_debug2.py", line 33, in main
        show_stack(task1, 3)
      File "/Users/wwy/async_debug2.py", line 12, in coro
        async def coro():

    task1's stack frames:
    []

    task1's stack info:
    No stack for <Task finished name='task1' coro=<coro() done, defined at /
Users/wwy/async_debug2.py:12> result=None>
```

该结果对应规则1和2。

接下来将全局变量fail设置为True，再次验证：

```
    $ python3 async_debug2.py
    task1's stack frames:
    [<frame at 0x10d279000, file '/Library/Frameworks/Python.framework/
Versions/3.11/lib/python3.11/asyncio/events.py', line 80, code _run>, <frame at
0x10d1cb400, file '/Users/wwy/async_debug2.py', line 33, code main>, <frame at
0x10d2c8a40, file '/Users/wwy/async_debug2.py', line 12, code coro>]

    task1's stack info:
    Stack for <Task pending name='task1' coro=<coro() running at /Users/wwy/
async_debug2.py:12>> (most recent call last):
```

```
        File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
asyncio/events.py", line 80, in _run
          self._context.run(self._callback, *self._args)
        File "/Users/wwy/async_debug2.py", line 33, in main
          show_stack(task1, 3)
        File "/Users/wwy/async_debug2.py", line 12, in coro
          async def coro():

      task1's stack frames:
      [<frame at 0x10d1cb400, file '/Users/wwy/async_debug2.py', line 39, code
main>, <frame at 0x10d2c8a40, file '/Users/wwy/async_debug2.py', line 14, code
coro>]

      task1's stack info:
      Traceback for <Task finished name='task1' coro=<coro() done, defined at /
Users/wwy/async_debug2.py:12> exception=RuntimeError()> (most recent call last):
        File "/Users/wwy/async_debug2.py", line 39, in main
          show_stack(task1, 3)
        File "/Users/wwy/async_debug2.py", line 14, in coro
          raise RuntimeError()
      RuntimeError
```

该结果对应规则1和3。

---

[1] "PEP 578 – Python Runtime Audit Hooks". https://peps.python.org/pep-0578/.