啊，我的程序为啥卡住啦

目录

正文

服务器程序员最怕的就是程序crash，不过有时候程序没有crash，但是“不工作”了也是够吓人的。所谓“不工作”就是指程序不再响应新的请求，处在了某种自娱自乐的状态，英语有一个很形象但的单词“hung”，但我不知道怎么翻译，姑且称之为“卡住”吧。本人遇到过的有两种情况，一种是卡在系统调用，如常见的磁盘IO或者网络、多线程锁；另一种就是代码进入了死循环。

在《日志的艺术》一文中，讨论了日志的重要性，如果日志恰当，也能帮助我们分许程序卡住的问题。如果狂刷重复的日志，那么很可能就是死循环，从日志内容就能分析出死循环的位置，甚至是死循环的原因。如果没有日志输出了，那么看看最后一条日志的内容，也许就会告诉我们即将进行IO操作，当然也可能是即将进入死循环。

如果日志无法提供充足的信息，那就得求助于其他的手段，在Linux下当仁不让的自然是gdb，gdb的功能很强大，陈皓大牛的用GDB调试程序系列是我见过的讲解gdb最好的中文文章。CPython是用C语言实现的，自然也是可以用gdb来调试的，只不过，默认只显示C栈，凡人如我是无法脑补出python栈来的，这个时候就需要使用到python－dbg与libpython了，前者帮助显示Python源码的符号信息，后者能让我们能在Python语言这个层面调试程序，比如打印Python调用栈。

本文简答介绍在linux环境下如何利用gdb来分析卡住的程序，本文使用的Python为Cpython2.7，操作系统为Debian。

本文地址：http://www.cnblogs.com/xybaby/p/8025435.html

阻塞在IO

回到顶部

程序被卡住，很可能是程序被阻塞了，即在等待（wait）等个系统调用的结束，比如磁盘IO与网络IO、多线程，默认的情况下很多系统调用都是阻塞的。多线程的问题复杂一下，后面专门介绍。下面举一个UDP Socket的例子（run_forever_block.py）：

复制代码

```
# -*- coding: utf-8 -*-
import socket

def simple_server():
        address = ('0.0.0.0', 40000)
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind(address)

        while True:
            data, addr = s.recvfrom(2048)
            if not data:
                print "client has exist"
```

```
13                 break
14             print "received:", data, "from", addr
15
16 if __name__ == '__main__':
17         simple_server()
```

复制代码

这是一个简单的UDP程序，代码在（0.0.0.0， 40000）这个地址上等待接收数据，核心就是第10行的recvfrom函数，这就是一个阻塞（blocking）的系统调用，只有当读到数据之后才会返回，因此程序会卡在第10行。当然，也可以通过fcntl设置该函数为非阻塞（non-blocking）。

我们来看看阻塞的情况，运行程序，然后通过top查看这个进程的状态

```
26466           20   0   41888   8996   5868 S   0.0  0.1   0:00.01 python run_forever_block.py
```

可以看到这个进程的pid是26466，进程状态为S(Sleep)，CPU为0.0。进程状态和CPU都暗示我们，当前进程正阻塞在某个系统调用。这个时候，有一个很好使的命令：strace，可以跟踪进程的所有系统调用，我们来看看

```
~$ strace -T -tt -e trace=all -p 26466
Process 26466 attached
19:21:34.746019 recvfrom(3,
```

可以看到，进程卡在了recvfrom这个系统调用，对应的文件描述符（file descriptor）是3，其实通过recvfrom这个名字，大致也能定位问题。关于文件描述符，还有一个更实用的命令，lsof（list open file），可以看到当前进程打开的所有文件，在linux下，一切皆文件，自然也就包括了socket。

```
lsof -p 26466
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

...
python 26466 xxxxxxx 3u IPv4 221871586 0t0 UDP *:40000
```

从上面可以看出这个文件描述符（3U）更详细的信息，比如是IPV4 UDP，监听在40000端口等等。

使用gdb调试Python程序

回到顶部

上面这个例子非常的简单，简单到我们直接从系统调用就能看出问题，但是实际的情况下可能更加复杂，即无法通过系统调用直接定位到处问题的代码。这个时候就可以使用gdb了，关于如何用gdb来调试python程序，可以参考了使用gdb调试Python进程这篇文章。以上面的代码为例，首先得做好准备条件

首先，参考DebuggingWithGdb，根据自己的Linux系统安装好python-dbg，在我的平台（Debian）上即 *sudo apt-get install gdb python2.7-dbg*

然后，下载libpython，放在自己的HOME目录下

接下来就可以使用gdb进行分析了：*gdb -p 26466*

在gdb的交互式环境中，使用bt命令，看到的是C栈，意义不大，我们直接来看Python栈

```
(gdb) python
>import sys
>sys.path.insert(0, '/home/xxx')
>import libpython
>end


(gdb) py-bt
```

```
Traceback (most recent call first):
  File "run_forever_block.py", line 10, in simple_server
    data, addr = s.recvfrom(2048)
  File "run_forever_block.py", line 17, in <module>
    simple_server()
```

可以看到，通过py-bt就能显示出完整的python调用栈，能帮助我们定位问题。还有很多py开头的命令，具体可见libpython.py中（所有gdb.Command的子类都是一个命令），这篇文章中总结了几个常用的：

- py-list 查看当前python应用程序上下文
- py-bt 查看当前python应用程序调用堆栈
- py-bt-full 查看当前python应用程序调用堆栈，并且显示每个frame的详细情况
- py-print 查看python变量
- py-locals 查看当前的scope的变量
- py-up 查看上一个frame
- py-down 查看下一个frame

死循环

回到顶部

死循环很令人讨厌，死循环是预期之外的无限循环。最典型的预期之内的无限循环是socketserver，进程不死，服务不止。而死循环看起来很忙（CPU100%），但是没有任何实质的作用。死循环有不同的粒度，最粗的粒度是两个进程之间的相互调用，比如RPC；其次是函数级别，较为常见的是没有边界条件的递归调用，或者在多个函数之间的相互调用；最小的粒度是在一个函数内部，某一个代码块（code block）的死循环，最为常见的就是for，while语句。在Python中，函数级别是不会造成无限循环的，如代码所示：

复制代码

```
# -*- coding: utf-8 -*-
def func1():
    func()

def func():
    func1()

if __name__ == '__main__':
    func()
```

复制代码

运行代码，很快就会抛出一个异常：*RuntimeError: maximum recursion depth exceeded*。显然，python内部维护了一个调用栈，限制了最大的递归深度，默认约为1000层，也可以通过*sys.setrecursionlimit(limit)*来修改最大递归深度。在Python中，虽然出现这种函数级别的死循环不会导致无限循环，但是也会占用宝贵的CPU，也是决不应该出现的。

而代码块级别的死循环，则会让CPU转到飞起，如下面的代码

复制代码

```
# -*- coding: utf-8 -*-

def func():
    while True:
        pass

if __name__ == '__main__':
    func()
```

复制代码

这种情况，通过看CPU还是很好定位的

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
24943           20   0   32848   6764   4212 R  99.8  0.1   0:21.32 python run_forever.py
```

从进程状态R（run），以及100%的CPU，基本上就能确定是死循环了，当然也不排除是CPU密集型，这个跟代码的具体逻辑相关。这个时候，也是可以通过gdb来看看当前调用栈的，具体的准备工作如上，这里直接给出py-bt结果

```
(gdb) py-bt
Traceback (most recent call first):
      File "run_forever.py", line 5, in func
      File "run_forever.py", line 8, in <module>
(gdb)
```

在《无限“递归”的python程序》一文中，提到过使用协程greenlet能产生无限循环的效果，而且看起来非常像函数递归，其表现和gdb调试结果与这里的死循环是一样的

多线程死锁

回到顶部

由于Python的GIL，在我们的项目中使用Python多线程的时候并不多。不过多线程死锁是一个非常普遍的问题，而且一般来说又是一个低概率的事情，复现不容易，多出现在高并发的线上环境。这里直接使用《飘逸的python - 演示一种死锁的产生》中的代码，然后分析这个死锁的多线程程序

```
#coding=utf-8
import time
import threading
class Account:
    def __init__(self, _id, balance, lock):
        self.id = _id
        self.balance = balance
        self.lock = lock

    def withdraw(self, amount):
        self.balance -= amount

    def deposit(self, amount):
        self.balance += amount


def transfer(_from, to, amount):
    if _from.lock.acquire():#锁住自己的账户
        _from.withdraw(amount)
        time.sleep(1)#让交易时间变长，2个交易线程时间上重叠，有足够时间来产生死锁
        print 'wait for lock...'
        if to.lock.acquire():#锁住对方的账户
            to.deposit(amount)
            to.lock.release()
        _from.lock.release()
    print 'finish...'

a = Account('a',1000, threading.Lock())
b = Account('b',1000, threading.Lock())
threading.Thread(target = transfer, args = (a, b, 100)).start()
threading.Thread(target = transfer, args = (b, a, 200)).start()
```

线程死锁代码实例

运行代码，可以看见，该进程（进程编号26143）也是出于Sleep状态，因为本质上进程也是阻塞在了某个系统调用，因此，同样可以使用strace

```
p$ strace -T -tt -e trace=all -p 26143
Process 26143 attached
19:29:29.289042 futex(0x1286060, FUTEX_WAIT_PRIVATE, 0, NULL
```

可以看见，进程阻塞在futex这个系统调用，参数的意义可以参见manpage。

gdb也非常适用于调试多线程程序，对于多线程，有几个很使用的命名

- info thread： 列出所有的线程，以及所在线程
- thread x： 切换到第X号线程
- thread apply all bt： 打印所有线程的调用栈

下面是简化后的结果

```
(gdb) info thread
Id Target Id Frame
3 Thread 0x7fb6b2119700 (LWP 26144) "python" 0x00007fb6b38d8050 in sem_wait () from
/lib/x86_64-linux-gnu/libpthread.so.0
2 Thread 0x7fb6b1918700 (LWP 26145) "python" 0x00007fb6b38d8050 in sem_wait () from
/lib/x86_64-linux-gnu/libpthread.so.0
* 1 Thread 0x7fb6b3cf8700 (LWP 26143) "python" 0x00007fb6b38d8050 in sem_wait () from
/lib/x86_64-linux-gnu/libpthread.so.0
(gdb) thread apply all bt

Thread 3 (Thread 0x7fb6b2119700 (LWP 26144)):
#0 0x00007fb6b38d8050 in sem_wait () from /lib/x86_64-linux-gnu/libpthread.so.0
#1 0x000000000057bd20 in PyThread_acquire_lock (waitflag=1, lock=0x126c2f0) at
../Python/thread_pthread.h:324
#2 lock_PyThread_acquire_lock.lto_priv.2551 () at ../Modules/threadmodule.c:52
...

Thread 2 (Thread 0x7fb6b1918700 (LWP 26145)):
#0 0x00007fb6b38d8050 in sem_wait () from /lib/x86_64-linux-gnu/libpthread.so.0
#1 0x000000000057bd20 in PyThread_acquire_lock (waitflag=1, lock=0x131b3d0) at
../Python/thread_pthread.h:324
...

Thread 1 (Thread 0x7fb6b3cf8700 (LWP 26143)):
#0 0x00007fb6b38d8050 in sem_wait () from /lib/x86_64-linux-gnu/libpthread.so.0
#1 0x000000000057bd20 in PyThread_acquire_lock (waitflag=1, lock=0x1286060) at
../Python/thread_pthread.h:324
#2 lock_PyThread_acquire_lock.lto_priv.2551 () at ../Modules/threadmodule.c:52
...
```

这里推荐一篇非常不错的文章，用gdb调试python多线程代码-记一次死锁的发现，记录了一个在真实环境中遇到的多线程死锁问题，感兴趣的同学可以细读。

Coredump

回到顶部

当进程被卡主，我们需要尽快恢复服务，被卡主的原因可能是偶然的，也有可能是必然的，而且实际中导致进程卡主的真正原因不一定那么清晰明了。在我们分析卡主的具体原因的时候，我们可能需要先尽快重启服务，这个时候就需要保留现场了，那就是coredump。

按照我的经验，有两种方式。

第一种，kill -11 pid，11代表信号SIGSEGV，在kill这个进程的同时产生coredump，这样就可以迅速重启程序，然后慢慢分析

第二种，在gdb中使用gcore（generate-core-file），这个也很有用，一些bug我们是可以通过gdb线上修复的，但问题还是需要时候继续查看，这个时候就可以用这个命令先产生coredump，然后退出gdb，让修复后的程序继续执行。

总结

回到顶部

当一个进程不再响应新的请求时，首先看日志，很多时候日志里面会包含足够的信息。

其次，看进程的信息，Sleep状态，以及100%的CPU都能给我们很多信息，也可以用pidstat查看进程的各种信息

如果怀疑进程被阻塞了，那么可以使用strace确认

linux下，gdb是很好的调试武器，建议平时多试试，coredump也是一定会遇到的

linux下运行的Python程序，可以配合使用python-dbg和libpython分析程序。

references

回到顶部

DebuggingWithGdb

GDB: The GNU Project Debugger

libpython.py

EasierPythonDebugging

用GDB调试程序

使用gdb调试Python进程

用gdb调试python多线程代码-记一次死锁的发现

标签: python, linux

« 上一篇：初识分布式计算：从MapReduce到Yarn&Fuxi
» 下一篇：我的进程去哪儿了，谁杀了我的进程

posted @ 2017-12-17 13:25 xybaby 阅读(2980) 评论(5) 编辑 收藏