# UXDB中使用pstack打印fork子进程所有线程堆栈信息

by 刘海峰 20200324

UXDB数据库在并行查询中, 出现如下"stack depth limit exceeded"的错误, 因此想使用pstack来打印其堆栈信息, 依次来排查错误.

void  
check\_stack\_depth(void)  
{  
 if (stack\_is\_too\_deep())  
 {  
 ereport(ERROR,  
 (errcode(ERRCODE\_STATEMENT\_TOO\_COMPLEX),  
 errmsg("stack depth limit exceeded"),  
 errhint("Increase the configuration parameter \"max\_stack\_depth\" (currently %dkB), "  
 "after ensuring the platform's stack depth limit is adequate.",  
 max\_stack\_depth)));  
 }  
}

使用pstack pid 打印进程堆栈信息, 有2种方式, 一是在进程中 "pstack pid"来打印, 二是 "sudo pstack pid" 在外部打印, 外部打印需要sudo权限.

## 进程外部打印

下面select.sh脚本是想打印select count(\*) 时, fork出来的并行查询子进程(即: bgworker parallel )的堆栈信息, 构造了下面的脚本可以进行抓取.

**先创建表：**

create table t1(id int, info text);

**插入1千万行数据：**

insert into t1 select generate\_series(1,10000000),'digoal'||generate\_series(1,10000000);

注意：

1. 要触发并行查询, 首先要确保t1中的数据量足够的多, 如: 1千万条, 否则不会触发并行查询  
2. 并行查询的开关要打开  
3. 至少要2个客户端同时查询

**select.sh内容如下：**

#!/bin/bash

# 在select.sh 脚本中, 通过2条select count(\*)的后台执行, 来模拟客户端的并行查询  
./psql -p 5432 -c "select count(\*) from t1;" &  
echo "main wait 5432 1"  
  
./psql -p 5432 -c "select count(\*) from t1" &  
echo "main wait 5432 2"  
  
# 抓取并行子进程堆栈信息构造方法如下  
# 这里需要等待一点时间, 因为前面的select查询时异步执行的, 如果不等待,  
# 执行到下面 ps -ef 获取其进程pid时, 并行查询子进程可能还没fork出来,  
# list的结果会为空  
sleep 0.1  
  
# 打印出所有并行查询子进程, 存放在list中  
ps -ef|grep -v color|grep para  
list=`ps -ef|grep -v grep|grep para|awk '{print $2}'`  
echo "list is " $list  
  
# 启动一个循环, 逐次打印各个进程的堆栈信息  
# 在具体实践中, 这里有2个注意事项  
# 1 因为是sudo运行, 所以在执行select.sh时, 需要输入root密码,   
# 虽然多次循环, 但实际只需要输入一次root密码就够了, 我是在centos  
# 下, centos中, sudo 输入一次root密码, 后续短时间内如果再次sudo,   
# 默认是不需要root密码的  
#  
# 这里输入密码的速度要快, 否则 并行子进程可能会很快执行完并结束掉,   
# pstack 来不及打印  
#  
# 个人解决方法: 使用xshell, 在xshell中将输入root密码的操作添加到"  
# 快速命令集"中, 然后关联一个"Ctrl+F1"之类的快捷键, select.sh执行  
# 时, 按下"Ctrl+F1"即可快速输入密码.  
#  
# 这样就可以打印出其进程堆栈信息了  
  
for pid in $list  
do  
 echo ">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> pid = " $pid  
 sudo pstack $pid  
done  
  
wait

## 进程内部打印

在进程内部打印, 即在程序代码中调用 "pstack getpid()", 这种方式不需要sudo执行.

**示例如下：**

#include <sys/types.h>  
#include <unistd.h>  
  
bool  
stack\_is\_too\_deep(void)  
{  
 char stack\_top\_loc;  
 long stack\_depth;  
  
 /\*  
 \* Compute distance from reference point to my local variables  
 \*/  
 stack\_depth = (long) (stack\_base\_ptr - &stack\_top\_loc);  
  
 /\*  
 \* Take abs value, since stacks grow up on some machines, down on others  
 \*/  
 if (stack\_depth < 0)  
 stack\_depth = -stack\_depth;  
  
 /\*  
 \* Trouble?  
 \*  
 \* The test on stack\_base\_ptr prevents us from erroring out if called  
 \* during process setup or in a non-backend process. Logically it should  
 \* be done first, but putting it here avoids wasting cycles during normal  
 \* cases.  
 \*/  
 if (stack\_depth > max\_stack\_depth\_bytes &&  
 stack\_base\_ptr != NULL)  
 {  
 char str[128] = {0};  
 sprintf(str, "pstack %d", getpid());  
 system(str);  
  
 return true;  
 }  
  
}

# 附录

## LINUX 编程定位工具gstack,pstack

### pstack

pstack命令可显示每个进程的栈跟踪。

pstack 命令必须由相应进程的属主或 root 运行。

可以使用 pstack 来确定进程挂起的位置。

此命令允许使用的唯一选项是要检查的进程的 PID。

pstack 看活动的进程内的堆栈

用法：

root# pstack PID

### gstack

gstack -打印正在运行的进程的堆栈跟踪

使用方法：

gstack PID

描述

gstack连接到命令行中pid的活动进程

打印执行堆栈跟踪。如果ELF符号存在于二进制(usu -)中

如果你没有运行条带(1)，那么这个例子就会被打印出来

同样如此。

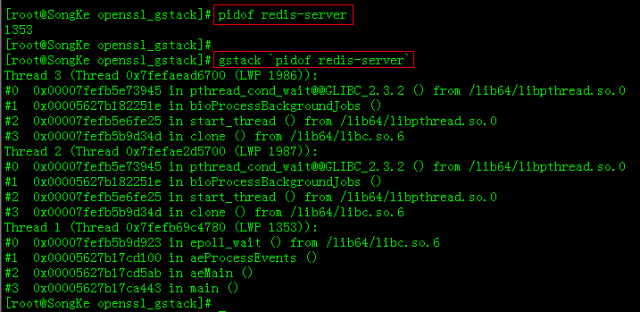
如果进程是线程组的一部分，那么gstack将打印出一个堆栈

对组中的每个线程进行跟踪。

### 思路分析

我们知道gdb的bt（backtrace）可以打印函数调用栈，但需要手动敲命令执行，不能批量多次运行，似乎不太方便。有没有更好的工具和方法搞定这个需求呢？有的，gstack就是一款用于方便查看函数调用栈的工具。gstack的用途是“print a stack trace of a running process”，即打印一个正在运行的进程的函数调用栈。下面以一个正在运行的redis-server进程为例，执行gstack `pidof redis-server`即可看到该进程当前正在运行的3个线程各自的函数调用栈。这其实与gdb bt看到的差不多，而且我们的目标是只需要函数名，而不需要地址信息，那么gstack有没有什么参数可以去掉每行的地址，以精简打印呢？

如下所示，gstack竟然没有帮助，这不像是一个正常的程序啊。用file `which gstack`查看，果然，它只是一个脚本，并不是一个正常的程序。



查看/usr/bin/gstack脚本源代码，发现它其实只是包装了gdb bt，并用sed对gdb bt的输出结果做了过滤而已。如下给出gstack脚本源代码的解读，该脚本分为五部分：

第一部分：检查是否提供一个入参，如果入参数量不是1，则打印用法提示，并退出脚本。

第二部分：检查入参必须是一个当前正在运行的程序的PID，如果不是，则退出脚本。

第三部分：判断内核是否支持gdb打印所有线程函数栈，如果不支持，则后续会将“bt”命令输入gdb中；如果支持，则后续会将“thread apply all bt”命令输入gdb中。

第四部分：执行gdb，通过“gdb [options] [executable-file] [process-id]”方式附着到指定PID的进程上，通过<<EOF方式为gdb传入多个命令，并将执行输出的结果通过管道“|”传给后续的sed命令。

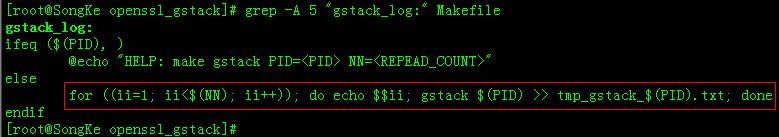
第五部分：用sed去掉gdb输出的无效行，只提取含有线程信息、函数信息的行。

通过上述对gstack脚本源代码的分析可知，gstack只是gdb bt的简单封装，与我们的目标还有一定差距。看来需要自己编写一些扩展脚本或程序，才能进一步达成目标。

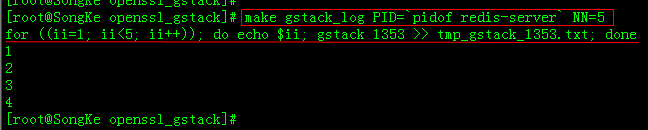
首先，需要编写一个脚本，重复运行多次gstack，采集目标程序足够多次函数调用栈；其次，需要进一步净化数据，比如函数地址信息就需要过滤掉；还有，需要归并出不同的函数调用栈，找到不同的函数调用链，因为gstack输出的函数栈是用Thread行分隔的，可以编写一个程序来解析Thread行，将每个Thread块（多行）放到哈希桶中排重（即，排除重复项），从而得到唯一不同的函数调用链。

### 扩展编程

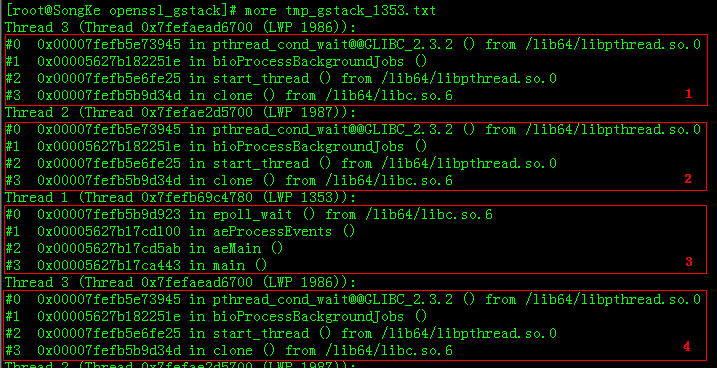
首先，编写一个makefile脚本，用shell for循环不断调用gstack，将输出结果追加到临时文本文件中。



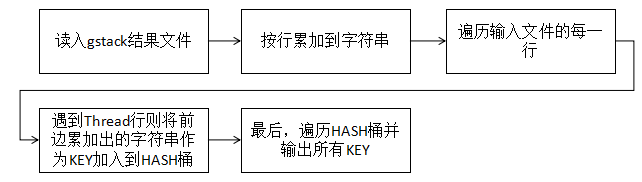
仍以redis-server为例，执行 make gstack\_log PID=`pidof redis-server` NN=5，即可对redis-server连续运行5次gstack，并将结果保存到一个临时文件tmp\_gstack\_1353.txt中。在正式采集时，可以将NN设置为很大，比如NN=2000次，以采集到足够多的不同的函数调用栈信息。



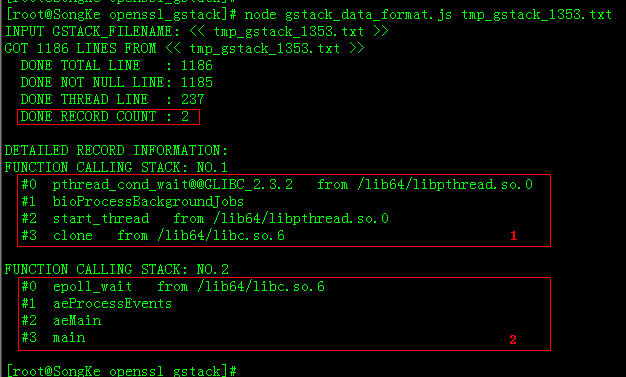
然后，查看一下输出的临时文件的内容，即多次gstack输出结果的罗列。下一步需要将每个Thread行所分隔的块（多行），如块1、块2、块3、块4、、、进行净化和排重。



编写一个Node.js小程序gstack\_data\_format.js，用于对gstack输出结果净化并排重。程序读入gstack结果文件（如：tmp\_gstack\_1353.txt），一行一行地读入并累加到一个字符串变量中，遇到Thread行则停止累加，并将该字符串作为KEY添加到一个HASH桶中，因为HASH KEY天然不会重复，利用这个特点进行排重；遇到Thread行后，清空该字符串变量，重新开始累加；依次往复，直到读完整个文件。程序基本流程如下，具体源代码请见本文附录。



如下给出gstack\_data\_format.js的运行效果。该gstack结果文件为1186行，采集到237个函数栈，进行净化、排重后，得到2个唯一不同的函数调用栈。



### 参考

<https://blog.csdn.net/O4dC8OjO7ZL6/article/details/78954755>