[**记一次tcmalloc分配内存引起的coredump**](http://www.cppblog.com/kevinlynx/archive/2015/04/06/210257.html)

**现象**

线上的服务出现coredump，堆栈为：

#0 0x000000000045d145 in GetStackTrace(void\*\*, int, int) ()

#1 0x000000000045ec22 in tcmalloc::PageHeap::GrowHeap(unsigned long) ()

#2 0x000000000045eeb3 in tcmalloc::PageHeap::New(unsigned long) ()

#3 0x0000000000459ee8 in tcmalloc::CentralFreeList::Populate() ()

#4 0x000000000045a088 in tcmalloc::CentralFreeList::FetchFromSpansSafe() ()

#5 0x000000000045a10a in tcmalloc::CentralFreeList::RemoveRange(void\*\*, void\*\*, int) ()

#6 0x000000000045c282 in tcmalloc::ThreadCache::FetchFromCentralCache(unsigned long, unsigned long) ()

#7 0x0000000000470766 in tc\_malloc ()

#8 0x00007f75532cd4c2 in \_\_conhash\_get\_rbnode (node=0x22c86870, hash=30)

at build/release64/cm\_sub/conhash/conhash\_inter.c:88

#9 0x00007f75532cd76e in \_\_conhash\_add\_replicas (conhash=0x24fbc7e0, iden=<value optimized out>)

at build/release64/cm\_sub/conhash/conhash\_inter.c:45

#10 0x00007f75532cd1fa in conhash\_add\_node (conhash=0x24fbc7e0, iden=0) at build/release64/cm\_sub/conhash/conhash.c:72

#11 0x00007f75532c651b in cm\_sub::TopoCluster::initLBPolicyInfo (this=0x2593a400)

at build/release64/cm\_sub/topo\_cluster.cpp:114

#12 0x00007f75532cad73 in cm\_sub::TopoClusterManager::processClusterMapTable (this=0xa219e0, ref=0x267ea8c0)

at build/release64/cm\_sub/topo\_cluster\_manager.cpp:396

#13 0x00007f75532c5a93 in cm\_sub::SubRespMsgProcess::reinitCluster (this=0x9c2f00, msg=0x4e738ed0)

at build/release64/cm\_sub/sub\_resp\_msg\_process.cpp:157

...

查看了应用层相关数据结构，基本数据都是没有问题的。所以最初怀疑是tcmalloc内部维护了错误的内存，在分配内存时出错，这个堆栈只是问题的表象。几天后，线上的另一个服务，基于同样的库，也core了，堆栈还是一样的。

最初定位问题都是从最近更新的东西入手，包括依赖的server环境，但都没有明显的问题，所以最后只能从core的直接原因入手。

**分析GetStackTrace**

确认core的详细位置：

# core在该指令

0x000000000045d145 <\_Z13GetStackTracePPvii+21>: mov 0x8(%rax),%r9

(gdb) p/x $rip # core 的指令位置

$9 = 0x45d145

(gdb) p/x $rax

$10 = 0x4e73aa58

(gdb) x/1a $rax+0x8 # rax + 8 = 0x4e73aa60

0x4e73aa60: 0x0

该指令尝试从[0x4e73aa60]处读取内容，然后出错，这个内存单元不可读。但是具体这个指令在代码中是什么意思，**需要将这个指令对应到代码中**。获取tcmalloc的源码，发现GetStackTrace根据编译选项有很多实现，所以这里选择最可能的实现，然后对比汇编以确认代码是否匹配。最初选择的是stacktrace\_x86-64-inl.h，后来发现完全不匹配，又选择了stacktrace\_x86-inl.h。这个实现版本里也有对64位平台的支持。

stacktrace\_x86-inl.h里使用了一些宏来生成函数名和参数，精简后代码大概为：

int GET\_STACK\_TRACE\_OR\_FRAMES {

void \*\*sp;

unsigned long rbp;

\_\_asm\_\_ volatile ("mov %%rbp, %0" : "=r" (rbp));

sp = (void \*\*) rbp;

int n = 0;

**while** (sp && n < max\_depth) {

**if** (\*(sp+1) == **reinterpret\_cast**<void \*>(0)) {

**break**;

}

void \*\*next\_sp = NextStackFrame<!IS\_STACK\_FRAMES, IS\_WITH\_CONTEXT>(sp, ucp);

**if** (skip\_count > 0) {

skip\_count--;

} **else** {

result[n] = \*(sp+1);

n++;

}

sp = next\_sp;

}

**return** n;

}

NextStackFrame是一个模板函数，包含一大堆代码，精简后非常简单：

**template**<bool STRICT\_UNWINDING, bool WITH\_CONTEXT>

**static** void \*\*NextStackFrame(void \*\*old\_sp, **const** void \*uc) {

void \*\*new\_sp = (void \*\*) \*old\_sp;

**if** (STRICT\_UNWINDING) {

**if** (new\_sp <= old\_sp) **return** NULL;

**if** ((uintptr\_t)new\_sp - (uintptr\_t)old\_sp > 100000) **return** NULL;

} **else** {

**if** (new\_sp == old\_sp) **return** NULL;

**if** ((new\_sp > old\_sp)

&& ((uintptr\_t)new\_sp - (uintptr\_t)old\_sp > 1000000)) **return** NULL;

}

**if** ((uintptr\_t)new\_sp & (**sizeof**(void \*) - 1)) **return** NULL;

**return** new\_sp;

}

上面这个代码到汇编的对比过程还是花了些时间，其中汇编中出现的一些常量可以大大缩短对比时间，例如上面出现了100000，汇编中就有：

0x000000000045d176 <\_Z13GetStackTracePPvii+70>: cmp $0x186a0,%rbx # 100000=0x186a0

*注意NextStackFrame中的 if (STRICT\_UNWINDING)使用的是模板参数，这导致生成的代码中根本没有else部分，也就没有1000000这个常量*

在对比代码的过程中，可以**知道关键的几个寄存器、内存位置对应到代码中的变量，从而可以还原core时的现场环境**。分析过程中不一定要从第一行汇编读，可以从较明显的位置读，从而还原整个代码，**函数返回指令、跳转指令、比较指令、读内存指令、参数寄存器**等都是比较明显对应的地方。

另外注意GetStackTrace在RecordGrowth中调用，传入了3个参数：

GetStackTrace(t->stack, kMaxStackDepth-1, 3); // kMaxStackDepth = 31

以下是我分析的简单注解：

(gdb) disassemble

Dump of assembler code for function \_Z13GetStackTracePPvii:

0x000000000045d130 <\_Z13GetStackTracePPvii+0>: push %rbp

0x000000000045d131 <\_Z13GetStackTracePPvii+1>: mov %rsp,%rbp

0x000000000045d134 <\_Z13GetStackTracePPvii+4>: push %rbx

0x000000000045d135 <\_Z13GetStackTracePPvii+5>: mov %rbp,%rax

0x000000000045d138 <\_Z13GetStackTracePPvii+8>: xor %r8d,%r8d

0x000000000045d13b <\_Z13GetStackTracePPvii+11>: test %rax,%rax

0x000000000045d13e <\_Z13GetStackTracePPvii+14>: je 0x45d167 <\_Z13GetStackTracePPvii+55>

0x000000000045d140 <\_Z13GetStackTracePPvii+16>: cmp %esi,%r8d # while ( .. max\_depth > n ?

0x000000000045d143 <\_Z13GetStackTracePPvii+19>: jge 0x45d167 <\_Z13GetStackTracePPvii+55>

0x000000000045d145 <\_Z13GetStackTracePPvii+21>: mov 0x8(%rax),%r9 # 关键位置：\*(sp+1) -> r9, rax 对应 sp变量

0x000000000045d149 <\_Z13GetStackTracePPvii+25>: test %r9,%r9 # \*(sp+1) == 0 ?

0x000000000045d14c <\_Z13GetStackTracePPvii+28>: je 0x45d167 <\_Z13GetStackTracePPvii+55>

0x000000000045d14e <\_Z13GetStackTracePPvii+30>: mov (%rax),%rcx # new\_sp = \*old\_sp，这里已经是NextStackFrame的代码

0x000000000045d151 <\_Z13GetStackTracePPvii+33>: cmp %rcx,%rax # new\_sp <= old\_sp ?

0x000000000045d154 <\_Z13GetStackTracePPvii+36>: jb 0x45d170 <\_Z13GetStackTracePPvii+64> # new\_sp > old\_sp 跳转

0x000000000045d156 <\_Z13GetStackTracePPvii+38>: xor %ecx,%ecx

0x000000000045d158 <\_Z13GetStackTracePPvii+40>: test %edx,%edx # skip\_count > 0 ?

0x000000000045d15a <\_Z13GetStackTracePPvii+42>: jle 0x45d186 <\_Z13GetStackTracePPvii+86>

0x000000000045d15c <\_Z13GetStackTracePPvii+44>: sub $0x1,%edx # skip\_count--

0x000000000045d15f <\_Z13GetStackTracePPvii+47>: mov %rcx,%rax

0x000000000045d162 <\_Z13GetStackTracePPvii+50>: test %rax,%rax # while (sp ?

0x000000000045d165 <\_Z13GetStackTracePPvii+53>: jne 0x45d140 <\_Z13GetStackTracePPvii+16>

0x000000000045d167 <\_Z13GetStackTracePPvii+55>: pop %rbx

0x000000000045d168 <\_Z13GetStackTracePPvii+56>: leaveq

0x000000000045d169 <\_Z13GetStackTracePPvii+57>: mov %r8d,%eax # r8 存储了返回值，r8=n

0x000000000045d16c <\_Z13GetStackTracePPvii+60>: retq # return n

0x000000000045d16d <\_Z13GetStackTracePPvii+61>: nopl (%rax)

0x000000000045d170 <\_Z13GetStackTracePPvii+64>: mov %rcx,%rbx

0x000000000045d173 <\_Z13GetStackTracePPvii+67>: sub %rax,%rbx # offset = new\_sp - old\_sp

0x000000000045d176 <\_Z13GetStackTracePPvii+70>: cmp $0x186a0,%rbx # offset > 100000 ?

0x000000000045d17d <\_Z13GetStackTracePPvii+77>: ja 0x45d156 <\_Z13GetStackTracePPvii+38> # return NULL

0x000000000045d17f <\_Z13GetStackTracePPvii+79>: test $0x7,%cl # new\_sp & (sizeof(void\*) - 1)

0x000000000045d182 <\_Z13GetStackTracePPvii+82>: je 0x45d158 <\_Z13GetStackTracePPvii+40>

0x000000000045d184 <\_Z13GetStackTracePPvii+84>: jmp 0x45d156 <\_Z13GetStackTracePPvii+38>

0x000000000045d186 <\_Z13GetStackTracePPvii+86>: movslq %r8d,%rax # rax = n

0x000000000045d189 <\_Z13GetStackTracePPvii+89>: add $0x1,%r8d # n++

0x000000000045d18d <\_Z13GetStackTracePPvii+93>: mov %r9,(%rdi,%rax,8)# 关键位置：result[n] = \*(sp+1)

0x000000000045d191 <\_Z13GetStackTracePPvii+97>: jmp 0x45d15f <\_Z13GetStackTracePPvii+47>

分析过程比较耗时，同时还可以分析下GetStackTrace函数的实现原理，其实就是利用RBP寄存器不断回溯，从而得到整个调用堆栈各个函数的地址（严格来说是返回地址）。简单示意下函数调用中RBP的情况：

...

saved registers # i.e push rbx

local variabes # i.e sub 0x10, rsp

return address # call xxx

last func RBP # push rbp; mov rsp, rbp

saved registers

local variables

return address

last func RBP

... # rsp

总之，**一般情况下，任何一个函数中，RBP寄存器指向了当前函数的栈基址，该栈基址中又存储了调用者的栈基址，同时该栈基址前面还存储了调用者的返回地址**。所以，GetStackTrace的实现，简单来说大概就是：

sp = rbp *// 取得当前函数GetStackTrace的栈基址*

**while** (n < max\_depth) {

new\_sp = \*sp

result[n] = \*(new\_sp+1)

n++

}

以上，最终就知道了以下关键信息：

* r8 对应变量 n，表示当前取到第几个栈帧了
* rax 对应变量 sp，代码core在 \*(sp+1)
* rdi 对应变量 result，用于存储取得的各个地址

然后可以看看现场是怎样的：

(gdb) x/10a $rdi

0x1ffc9b98: 0x45a088 <\_ZN8tcmalloc15CentralFreeList18FetchFromSpansSafeEv+40> 0x45a10a <\_ZN8tcmalloc15CentralFreeList11RemoveRangeEPPvS2\_i+106>

0x1ffc9ba8: 0x45c282 <\_ZN8tcmalloc11ThreadCache21FetchFromCentralCacheEmm+114> 0x470766 <tc\_malloc+790>

0x1ffc9bb8: 0x7f75532cd4c2 <\_\_conhash\_get\_rbnode+34> 0x0

0x1ffc9bc8: 0x0 0x0

0x1ffc9bd8: 0x0 0x0

(gdb) p/x $r8

$3 = 0x5

(gdb) p/x $rax

$4 = 0x4e73aa58

**小结：**

GetStackTrace在取调用\_\_conhash\_get\_rbnode的函数时出错，取得了5个函数地址。当前使用的RBP为0x4e73aa58。

**错误的RBP**

RBP也是从堆栈中取出来的，既然这个地址有问题，首先想到的就是有代码局部变量/数组写越界。例如sprintf的使用。而且，**一般写越界破坏堆栈，都可能是把调用者的堆栈破坏了**，例如：

char s[32];

memcpy(s, p, 1024);

因为写入都是从低地址往高地址写，而调用者的堆栈在高地址。当然，也会遇到写坏调用者的调用者的堆栈，也就是跨栈帧越界写，例如以前遇到的：

len = vsnprintf(buf, sizeof(buf), fmt, wtf-long-string);

buf[len] = 0;

\_\_conhash\_get\_rbnode的RBP是在tcmalloc的堆栈中取的：

(gdb) f 7

#7 0x0000000000470766 in tc\_malloc ()

(gdb) x/10a $rsp

0x4e738b80: 0x4e73aa58 0x22c86870

0x4e738b90: 0x4e738bd0 0x85

0x4e738ba0: 0x4e73aa58 0x7f75532cd4c2 <\_\_conhash\_get\_rbnode+34> # 0x4e73aa58

所以这里就会怀疑是tcmalloc这个函数里有把堆栈破坏，这个时候就是读代码，看看有没有疑似危险的地方，未果。这里就陷入了僵局，怀疑又遇到了跨栈帧破坏的情况，这个时候就只能\_\_conhash\_get\_rbnode调用栈中周围的函数翻翻，例如调用\_\_conhash\_get\_rbnode的函数\_\_conhash\_add\_replicas中恰好有字符串操作：

void \_\_conhash\_add\_replicas(conhash\_t \*conhash, int32\_t iden)

{

node\_t\* node = \_\_conhash\_create\_node(iden, conhash->replica);

...

char buf[buf\_len]; *// buf\_len = 64*

...

snprintf(buf, buf\_len, VIRT\_NODE\_HASH\_FMT, node->iden, i);

uint32\_t hash = conhash->cb\_hashfunc(buf);

**if**(util\_rbtree\_search(&(conhash->vnode\_tree), hash) == NULL)

{

util\_rbtree\_node\_t\* rbnode = \_\_conhash\_get\_rbnode(node, hash);

...

这段代码最终发现是没有问题的，这里又耗费了不少时间。后来发现若干个函数里的RBP都有点奇怪，这个调用栈比较正常的范围是：0x4e738c90

(gdb) f 8

#8 0x00007f75532cd4c2 in \_\_conhash\_get\_rbnode (node=0x22c86870, hash=30)

(gdb) p/x $rbp

$6 = 0x4e73aa58 # 这个还不算特别可疑

(gdb) f 9

#9 0x00007f75532cd76e in \_\_conhash\_add\_replicas (conhash=0x24fbc7e0, iden=<value optimized out>)

(gdb) p/x $rbp

$7 = 0x4e738c60 # 这个也不算特别可疑

(gdb) f 10

#10 0x00007f75532cd1fa in conhash\_add\_node (conhash=0x24fbc7e0, iden=0) at build/release64/cm\_sub/conhash/conhash.c:72

(gdb) p/x $rbp # 可疑

$8 = 0x0

(gdb) f 11

#11 0x00007f75532c651b in cm\_sub::TopoCluster::initLBPolicyInfo (this=0x2593a400)

(gdb) p/x $rbp # 可疑

$9 = 0x2598fef0

**为什么很多函数中RBP都看起来不正常？** 想了想真要是代码里把堆栈破坏了，这错误得发生得多巧妙？

**错误RBP的来源**

然后转机来了，脑海中突然闪出-fomit-frame-pointer。编译器生成的代码中是可以不需要栈基址指针的，也就是RBP寄存器不作为栈基址寄存器。大部分函数或者说开启了frame-pointer的函数，其函数头都会有以下指令：

push %rbp

mov %rsp,%rbp

...

表示保存调用者的栈基址到栈中，以及设置自己的栈基址。看下\_\_conhash系列函数；

Dump of assembler code for function \_\_conhash\_get\_rbnode:

0x00007f75532cd4a0 <\_\_conhash\_get\_rbnode+0>: mov %rbx,-0x18(%rsp)

0x00007f75532cd4a5 <\_\_conhash\_get\_rbnode+5>: mov %rbp,-0x10(%rsp)

...

这个库是单独编译的，没有显示指定-fno-omit-frame-pointer，查阅[gcc手册](https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)，o2优化是开启了omit-frame-pinter 的。

在没有RBP的情况下，tcmalloc的GetStackTrace尝试读RBP取获取调用返回地址，自然是有问题的。但是，**如果整个调用栈中的函数，要么有RBP，要么没有RBP，那么GetStackTrace取出的结果最多就是跳过一些栈帧，不会出错。** 除非，这中间的某个函数把RBP寄存器另作他用（编译器省出这个寄存器肯定是要另作他用的）。所以这里继续追查这个错误地址0x4e73aa58的来源。

来源已经比较明显，肯定是\_\_conhash\_get\_rbnode中设置的，因为这个函数的RBP是在被调用者tcmalloc中保存的。

Dump of assembler code for function \_\_conhash\_get\_rbnode:

0x00007f75532cd4a0 <\_\_conhash\_get\_rbnode+0>: mov %rbx,-0x18(%rsp)

0x00007f75532cd4a5 <\_\_conhash\_get\_rbnode+5>: mov %rbp,-0x10(%rsp)

0x00007f75532cd4aa <\_\_conhash\_get\_rbnode+10>: mov %esi,%ebp # 改写了RBP

0x00007f75532cd4ac <\_\_conhash\_get\_rbnode+12>: mov %r12,-0x8(%rsp)

0x00007f75532cd4b1 <\_\_conhash\_get\_rbnode+17>: sub $0x18,%rsp

0x00007f75532cd4b5 <\_\_conhash\_get\_rbnode+21>: mov %rdi,%r12

0x00007f75532cd4b8 <\_\_conhash\_get\_rbnode+24>: mov $0x30,%edi

0x00007f75532cd4bd <\_\_conhash\_get\_rbnode+29>: callq 0x7f75532b98c8 <malloc@plt> # 调用tcmalloc，汇编到这里即可

这里打印RSI寄存器的值可能会被误导，因为任何时候打印寄存器的值可能都是错的，除非它有被显示保存。不过这里可以看出RSI的值来源于参数(RSI对应第二个参数)：

void \_\_conhash\_add\_replicas(conhash\_t \*conhash, int32\_t iden)

{

node\_t\* node = \_\_conhash\_create\_node(iden, conhash->replica);

...

char buf[buf\_len]; *// buf\_len = 64*

...

snprintf(buf, buf\_len, VIRT\_NODE\_HASH\_FMT, node->iden, i);

uint32\_t hash = conhash->cb\_hashfunc(buf); *// hash值由一个字符串哈希函数计算*

**if**(util\_rbtree\_search(&(conhash->vnode\_tree), hash) == NULL)

{

util\_rbtree\_node\_t\* rbnode = \_\_conhash\_get\_rbnode(node, hash); *// hash值*

...

追到\_\_conhash\_add\_replicas：

0x00007f75532cd764 <\_\_conhash\_add\_replicas+164>: mov %ebx,%esi # 来源于rbx

0x00007f75532cd766 <\_\_conhash\_add\_replicas+166>: mov %r15,%rdi

0x00007f75532cd769 <\_\_conhash\_add\_replicas+169>: callq 0x7f75532b9e48 <\_\_conhash\_get\_rbnode@plt>

(gdb) p/x $rbx

$11 = 0x4e73aa58

(gdb) p/x hash

$12 = 0x4e73aa58 # 0x4e73aa58

找到了0x4e73aa58的来源。这个地址值竟然是一个字符串哈希算法算出来的！这里还可以看看这个字符串的内容：

(gdb) x/1s $rsp

0x4e738bd0: "conhash-00000-00133"

这个碉堡的哈希函数是conhash\_hash\_def。

**coredump的条件**

以上，既然只要某个库omit-frame-pointer，那tcmalloc就可能出错，为什么发生的频率并不高呢？这个可以回到GetStackTrace尤其是NextStackFrame的实现，其中包含了几个合法RBP的判定：

**if** (new\_sp <= old\_sp) **return** NULL; *// 上一个栈帧的RBP肯定比当前的大*

**if** ((uintptr\_t)new\_sp - (uintptr\_t)old\_sp > 100000) **return** NULL; *// 指针值范围还必须在100000内*

...

**if** ((uintptr\_t)new\_sp & (**sizeof**(void \*) - 1)) **return** NULL; *// 由于本身保存的是指针，所以还必须是sizeof(void\*)的整数倍，对齐*

有了以上条件，才使得这个core几率变得很低。

**总结**

最后，如果你很熟悉tcmalloc，整个问题估计就被秒解了：[tcmalloc INSTALL](http://gperftools.googlecode.com/svn/trunk/INSTALL)

**附**

另外附上另一个有意思的东西。

在分析\_\_conhash\_add\_replicas时，其内定义了一个64字节的字符数组，查看其堆栈：

(gdb) x/20a $rsp

0x4e738bd0: 0x2d687361686e6f63 0x30302d3030303030 # 这些是字符串conhash-00000-00133

0x4e738be0: 0x333331 0x0

0x4e738bf0: 0x0 0x7f75532cd69e <\_\_conhash\_create\_node+78>

0x4e738c00: 0x24fbc7e0 0x4e738c60

0x4e738c10: 0x24fbc7e0 0x7f75532cd6e3 <\_\_conhash\_add\_replicas+35>

0x4e738c20: 0x0 0x24fbc7e8

0x4e738c30: 0x4e738c20 0x24fbc7e0

0x4e738c40: 0x22324360 0x246632c0

0x4e738c50: 0x0 0x0

0x4e738c60: 0x0 0x7f75532cd1fa <conhash\_add\_node+74>

最开始我觉得buf占64字节，也就是整个[0x4e738bd0, 0x4e738c10)内存，但是这块内存里居然有函数地址，这一度使我怀疑这里有问题。后来醒悟这些地址是定义buf前调用\_\_conhash\_create\_node产生的，调用过程中写到堆栈里，调用完后栈指针改变，但并不需要清空栈中的内容。