# gdb 如何调用函数?

CPP开发者 2018-05-01



> 编译：Linux 中国 / Lv Feng，英文： Julia Evans
> https://linux.cn/article-9588-1.html

在这周，我发现我可以从 gdb 上调用 C 函数。这看起来很酷，因为在过去我认为 gdb 最多只是一个只读调试工具。

我对 gdb 能够调用函数感到很吃惊。正如往常所做的那样，我在 Twitter 上询问这是如何工作的。我得到了大量的有用答案。我最喜欢的答案是 Evan Klitzke 的示例 C 代码，它展示了 gdb 如何调用函数。代码能够运行，这很令人激动!

我（通过一些跟踪和实验）认为那个示例 C 代码和 gdb 实际上如何调用函数不同。因此，在这篇文章中，我将会阐述 gdb 是如何调用函数的，以及我是如何知道的。

关于 gdb 如何调用函数，还有许多我不知道的事情，并且，在这儿我写的内容有可能是错误的。

## 从 gdb 中调用 C 函数意味着什么?

在开始讲解这是如何工作之前，我先快速的谈论一下我是如何发现这件令人惊讶的事情的。

假如，你已经在运行一个 C 程序（目标程序）。你可以运行程序中的一个函数，只需要像下面这样做:

- 暂停程序（因为它已经在运行中）
- 找到你想调用的函数的地址（使用符号表）
- 使程序（目标程序）跳转到那个地址
- 当函数返回时，恢复之前的指令指针和寄存器

通过符号表来找到想要调用的函数的地址非常容易。下面是一段非常简单但能够工作的代码，我在 Linux 上使用这段代码作为例子来讲解如何找到地址。这段代码使用 elf crate。如果我想找到 PID 为 2345 的进程中的 `foo` 函数的地址，那么我可以运行 `elf_symbol_value("/proc/2345/exe", "foo")`。

```rust
fn elf_symbol_value(file_name: &str, symbol_name: &str) -> Result<u64, Box<std::error::Error>> {
    // 打开 ELF 文件
    let file = elf::File::open_path(file_name).ok().ok_or("parse error")?;
    // 在所有的段 & 符号中循环，直到找到正确的那个
    let sections = &file.sections;
    for s in sections {
        for sym in file.get_symbols(&s).ok().ok_or("parse error")? {
            if sym.name == symbol_name {
                return Ok(sym.value);
            }
        }
    }
    None.ok_or("No symbol found")?
}
```

这并不能够真的发挥作用，你还需要找到文件的内存映射，并将符号偏移量加到文件映射的起始位置。找到内存映射并不困难，它位于 `/proc/PID/maps` 中。

总之，找到想要调用的函数地址对我来说很直接，但是其余部分（改变指令指针，恢复寄存器等）看起来就不这么明显了。

## 你不能仅仅进行跳转

我已经说过，你不能够仅仅找到你想要运行的那个函数地址，然后跳转到那儿。我在 gdb 中尝试过那样做（`jump foo`），然后程序出现了段错误。毫无意义。

## 如何从 gdb 中调用 C 函数

首先，这是可能的。我写了一个非常简洁的 C 程序，它所做的事只有 `sleep` 1000 秒，把这个文件命名为 `test.c`：

```c
#include <unistd.h>
int foo() {
    return 3;
}
int main() {
    sleep(1000);
}
```

接下来，编译并运行它：

```
$ gcc -o test  test.c
$ ./test
```

最后，我们使用 gdb 来跟踪 test 这一程序：

```
$ sudo gdb -p $(pgrep -f test)
(gdb) p foo()
$1 = 3
(gdb) quit
```

我运行 `pfoo()` 然后它运行了这个函数！这非常有趣。

**这有什么用?**

下面是一些可能的用途：

- 它使得你可以把 gdb 当成一个 C 应答式程序（REPL），这很有趣，我想对开发也会有用
- 在 gdb 中进行调试的时候展示/浏览复杂数据结构的功能函数（感谢 @invalidop）
- 在进程运行时设置一个任意的名字空间（我的同事 nelhage 对此非常惊讶）
- 可能还有许多我所不知道的用途

**它是如何工作的**

当我在 Twitter 上询问从 gdb 中调用函数是如何工作的时，我得到了大量有用的回答。许多答案是“你从符号表中得到了函数的地址”，但这并不是完整的答案。

有个人告诉了我两篇关于 gdb 如何工作的系列文章：原生调试：第一部分，原生调试：第二部分。第一部分讲述了 gdb 是如何调用函数的（指出了 gdb 实际上完成这件事并不简单，但是我将会尽力）。

步骤列举如下：

1. 停止进程
2. 创建一个新的栈框（远离真实栈）
3. 保存所有寄存器

4. 设置你想要调用的函数的寄存器参数
5. 设置栈指针指向新的栈框stack frame
6. 在内存中某个位置放置一条陷阱指令
7. 为陷阱指令设置返回地址
8. 设置指令寄存器的值为你想要调用的函数地址
9. 再次运行进程!

（LCTT 译注：如果将这个调用的函数看成一个单独的线程，gdb 实际上所做的事情就是一个简单的线程上下文切换）

我不知道 gdb 是如何完成这些所有事情的，但是今天晚上，我学到了这些所有事情中的其中几件。

## 创建一个栈框

如果你想要运行一个 C 函数，那么你需要一个栈来存储变量。你肯定不想继续使用当前的栈。准确来说，在 gdb 调用函数之前（通过设置函数指针并跳转），它需要设置栈指针到某个地方。

这儿是 Twitter 上一些关于它如何工作的猜测：

> 我认为它在当前栈的栈顶上构造了一个新的栈框来进行调用!

以及

> 你确定是这样吗？它应该是分配一个伪栈，然后临时将 sp （栈指针寄存器）的值改为那个栈的地址。你可以试一试，你可以在那儿设置一个断点，然后看一看栈指针寄存器的值，它是否和当前程序寄存器的值相近？

我通过 gdb 做了一个试验：

```
(gdb) p $rsp
$7 = (void *) 0x7ffea3d0bca8
(gdb) break foo
Breakpoint 1 at 0x40052a
(gdb) p foo()
Breakpoint 1, 0x000000000040052a in foo ()
(gdb) p $rsp
$8 = (void *) 0x7ffea3d0bc00
```

这看起来符合“gdb 在当前栈的栈顶构造了一个新的栈框”这一理论。因为栈指针（`$rsp`）从 `0x7ffea3d0bca8` 变成了 `0x7ffea3d0bc00` —— 栈指针从高地址往低地址长。所以 `0x7ffea3d0bca8` 在 `0x7ffea3d0bc00` 的后面。真是有趣！

所以，看起来 gdb 只是在当前栈所在位置创建了一个新的栈框。这令我很惊讶！

## 改变指令指针

让我们来看一看 gdb 是如何改变指令指针的！

```
(gdb) p $rip
$1 = (void (*)()) 0x7fae7d29a2f0 <__nanosleep_nocancel+7>
(gdb) b foo
Breakpoint 1 at 0x40052a
(gdb) p foo()
Breakpoint 1, 0x000000000040052a in foo ()
(gdb) p $rip
$3 = (void (*)()) 0x40052a <foo+4>
```

的确是！指令指针从 `0x7fae7d29a2f0` 变为了 `0x40052a`（`foo` 函数的地址）。

我盯着输出看了很久，但仍然不理解它是如何改变指令指针的，但这并不影响什么。

## 如何设置断点

上面我写到 `break foo` 。我跟踪 gdb 运行程序的过程，但是没有任何发现。

下面是 gdb 用来设置断点的一些系统调用。它们非常简单。它把一条指令用 `cc` 代替了（这告诉我们 `int3` 意味着 `send SIGTRAP` https://defuse.ca/online-x86-assembler.html），并且一旦程序被打断了，它就把指令恢复为原先的样子。

我在函数 `foo` 那儿设置了一个断点，地址为 `0x400528` 。

`PTRACE_POKEDATA` 展示了 gdb 如何改变正在运行的程序。

```
// 改变 0x400528 处的指令
25622 ptrace(PTRACE_PEEKTEXT, 25618, 0x400528, [0x5d00000003b8e589]) = 0
25622 ptrace(PTRACE_POKEDATA, 25618, 0x400528, 0x5d00000003cce589) = 0
// 开始运行程序
25622 ptrace(PTRACE_CONT, 25618, 0x1, SIG_0) = 0
```

```
// 当到达断点时获取一个信号
25622 ptrace(PTRACE_GETSIGINFO, 25618, NULL, {si_signo=SIGTRAP, si_code=SI_KERNEL, si_value=
{int=-1447215360, ptr=0x7ffda9bd3f00}}) = 0
// 将 0x400528 处的指令更改为之前的样子
25622 ptrace(PTRACE_PEEKTEXT, 25618, 0x400528, [0x5d00000003cce589]) = 0
25622 ptrace(PTRACE_POKEDATA, 25618, 0x400528, 0x5d00000003b8e589) = 0
```

## 在某处放置一条陷阱指令

当 gdb 运行一个函数的时候，它也会在某个地方放置一条陷阱指令。这是其中一条。它基本上是用 `cc` 来替换一条指令（`int3`）。

```
5908  ptrace(PTRACE_PEEKTEXT, 5810, 0x7f6fa7c0b260, [0x48f389fd89485355]) = 0
5908  ptrace(PTRACE_PEEKTEXT, 5810, 0x7f6fa7c0b260, [0x48f389fd89485355]) = 0
5908 ptrace(PTRACE_POKEDATA, 5810, 0x7f6fa7c0b260, 0x48f389fd894853cc) = 0
```

`0x7f6fa7c0b260` 是什么？我查看了进程的内存映射，发现它位于 `/lib/x86_64-linux-gnu/libc-2.23.so` 中的某个位置。这很奇怪，为什么 gdb 将陷阱指令放在 libc 中?

让我们看一看里面的函数是什么，它是 `__libc_siglongjmp` 。其他 gdb 放置陷阱指令的地方的函数是 `__longjmp` 、`___longjmp_chk` 、`dl_main` 和 `_dl_close_worker` 。

为什么？我不知道！也许出于某种原因，当函数 `foo()` 返回时，它调用 `longjmp` ，从而 gdb 能够进行返回控制。我不确定。

## gdb 如何调用函数是很复杂的!

我将要在这儿停止了（现在已经凌晨 1 点），但是我知道的多一些了!

看起来"gdb 如何调用函数"这一问题的答案并不简单。我发现这很有趣并且努力找出其中一些答案，希望你也能够找到。

我依旧有很多未回答的问题，关于 gdb 是如何完成这些所有事的，但是可以了。我不需要真的知道关于 gdb 是如何工作的所有细节，但是我很开心，我有了一些进一步的理解。

阅读原文

喜欢此内容的人还喜欢

GNU & GCC 编译器的这些知识你都知道了吗?

技术让梦想更伟大

王健林的最后一搏

拆哪儿