

讲堂 > Linux性能优化实战 > 文章详情

18 | 案例篇：内存泄漏了，我该如何定位和处理？

2018-12-31 倪朋飞



18 | 案例篇：内存泄漏了，我该如何定位和处理？

朗读人：冯永吉 11'54" | 10.91M

你好，我是倪朋飞。

通过前几节对内存基础的学习，我相信你对 Linux 内存的工作原理，已经有了初步了解。

对普通进程来说，能看到的其实是内核提供的虚拟内存，这些虚拟内存还需要通过页表，由系统映射为物理内存。

当进程通过 `malloc()` 申请虚拟内存后，系统并不会立即为其分配物理内存，而是在首次访问时，才通过缺页异常陷入内核中分配内存。

为了协调 CPU 与磁盘间的性能差异，Linux 还会使用 Cache 和 Buffer，分别把文件和磁盘读写的数据缓存到内存中。

对应用程序来说，动态内存的分配和回收，是既核心又复杂的一个逻辑功能模块。管理内存的过程中，也很容易发生各种各样的“事故”，比如，

- 没正确回收分配后的内存，导致了泄漏。
- 访问的是已分配内存边界外的地址，导致程序异常退出，等等。

今天我就带你来看看，内存泄漏到底是怎么发生的，以及发生内存泄漏之后该如何排查和定位。

说起内存泄漏，这就要先从内存的分配和回收说起了。

内存的分配和回收

先回顾一下，你还记得应用程序中，都有哪些方法来分配内存吗？用完后，又该怎么释放还给系统呢？

进程的内存空间，用户空间内存包含只读段，数据段，堆，栈以及文件映射段等。

前面讲进程的内存空间时，我曾经提到过，用户空间内存包括多个不同的内存段，比如只读段、数据段、堆、栈以及文件映射段等。这些内存段正是应用程序使用内存的基本方式。

举个例子，你在程序中定义了一个局部变量，比如一个整数数组 `int data[64]`，就定义了一个可以存储 64 个整数的内存段。由于这是一个局部变量，它会从内存空间的栈中分配内存。

栈内存由系统自动分配和管理。一旦程序运行超出了这个局部变量的作用域，栈内存就会被系统自动回收，所以不会产生内存泄漏的问题。

再比如，很多时候，我们事先并不知道数据大小，所以你就要用到标准库函数 `malloc()`，在程序中动态分配内存。这时候，系统就会从内存空间的堆中分配内存。

堆内存由应用程序自己来分配和管理。除非程序退出，这些堆内存并不会被系统自动释放，而是需要应用程序明确调用库函数 `free()` 来释放它们。如果应用程序没有正确释放堆内存，就会造成内存泄漏。

这是两个栈和堆的例子，那么，其他内存段是否也会导致内存泄漏呢？经过我们前面的学习，这个问题并不难回答。

- 只读段，包括程序的代码和常量，由于是只读的，不会再去分配新的内存，所以也不会产生内存泄漏。
- 数据段，包括全局变量和静态变量，这些变量在定义时就已经确定了大小，所以也不会产生内存泄漏。
- 最后一个内存映射段，包括动态链接库和共享内存，其中共享内存由程序动态分配和管理。所以，如果程序在分配后忘了回收，就会导致跟堆内存类似的泄漏问题。

内存泄漏的危害非常大，这些忘记释放的内存，不仅应用程序自己不能访问，系统也不能把它们再次分配给其他应用。内存泄漏不断累积，甚至会耗尽系统内存。

虽然，系统最终可以通过 OOM（Out of Memory）机制杀死进程，但进程在 OOM 前，可能已经引发了一连串的反应，导致严重的性能问题。

比如，其他需要内存的进程，可能无法分配新的内存；内存不足，又会触发系统的缓存回收以及 SWAP 机制，从而进一步导致 I/O 的性能问题等等。

内存泄漏的危害这么大，那我们应该怎么检测这种问题呢？特别是，如果你已经发现了内存泄漏，该如何定位和处理呢。

接下来，我们就用一个计算斐波那契数列的案例，来看看内存泄漏问题的定位和处理方法。


斐波那契数列是一个这样的数列：0、1、1、2、3、5、8...，也就是除了前两个数是 0 和 1，其他数都由前面两数相加得到，用数学公式来表示就是 $F(n)=F(n-1)+F(n-2)$ ， $(n \geq 2)$ ， $F(0)=0$, $F(1)=1$ 。

案例

今天的案例基于 Ubuntu 18.04，当然，同样适用其他的 Linux 系统。

- 机器配置：2 CPU，8GB 内存
- 预先安装 sysstat、Docker 以及 bcc 软件包，比如：

```
1 # install sysstat docker
2 sudo apt-get install -y sysstat docker.io
3
4 # Install bcc
5 sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 4052245BD4284CDD
6 echo "deb https://repo.iovisor.org/apt/bionic bionic main" | sudo tee /etc/apt/sources.list.d/iov
7 sudo apt-get update
8 sudo apt-get install -y bcc-tools libbcc-examples linux-headers-$(uname -r)
```

 复制代码

其中，sysstat 和 Docker 我们已经很熟悉了。sysstat 软件包中的 vmstat，可以观察内存的变化情况；而 Docker 可以运行案例程序。

[bcc](#) 软件包前面也介绍过，它提供了一系列的 Linux 性能分析工具，常用来动态追踪进程和内核的行为。更多工作原理你先不用深究，后面学习我们会逐步接触。这里你只需要记住，按照上面步骤安装完后，它提供的所有工具都位于 /usr/share/bcc/tools 这个目录中。

注意：bcc-tools 需要内核版本为 4.1 或者更高，如果你使用的是 CentOS7，或者其他内核版本比较旧的系统，那么你需要手动[升级内核版本后再安装](#)。

打开一个终端，SSH 登录到机器上，安装上述工具。

同以前的案例一样，下面的所有命令都默认以 root 用户运行，如果你是用普通用户身份登录系统，请运行 `sudo su root` 命令切换到 root 用户。

如果安装过程中有什么问题，同样鼓励你先自己搜索解决，解决不了的，可以在留言区向我提问。如果你以前已经安装过了，就可以忽略这一点了。

安装完成后，再执行下面的命令来运行案例：

```
1 $ docker run --name=app -itd feisky/app:mem-leak
```

[复制代码](#)

案例成功运行后，你需要输入下面的命令，确认案例应用已经正常启动。如果一切正常，你应该可以看到下面这个界面：

```
1 $ docker logs app
2 2th => 1
3 3th => 2
4 4th => 3
5 5th => 5
6 6th => 8
7 7th => 13
```

[复制代码](#)

从输出中，我们可以发现，这个案例会输出斐波那契数列的一系列数值。实际上，这些数值每隔 1 秒输出一次。

知道了这些，我们应该怎么检查内存情况，判断有没有泄漏发生呢？你首先想到的可能是 `top` 工具，不过，`top` 虽然能观察系统和进程的内存占用情况，但今天的案例并不适合。**内存泄漏问题，我们更应该关注内存使用的变化趋势。**

所以，开头我也提到了，今天推荐的是另一个老熟人，`vmstat` 工具。

运行下面的 `vmstat`，等待一段时间，观察内存的变化情况。如果忘了 `vmstat` 里各指标的含义，记得复习前面内容，或者执行 `man vmstat` 查询。

```
1 # 每隔 3 秒输出一组数据
2 $ vmstat 3
3 procs -----memory----- ---swap-- -----io---- -system-- -----cpu-----
4 r  b  swpd   free   buff  cache   si   so    bi   bo    in   cs us sy id wa st
5 procs -----memory----- ---swap-- -----io---- -system-- -----cpu-----
6 r  b  swpd   free   buff  cache   si   so    bi   bo    in   cs us sy id wa st
7 0  0      0 6601824  97620 1098784    0    0     0    0   62  322  0  0 100  0  0
8 0  0      0 6601700  97620 1098788    0    0     0    0   57  251  0  0 100  0  0
9 0  0      0 6601320  97620 1098788    0    0     0    3   52  306  0  0 100  0  0
10 0  0      0 6601452  97628 1098788    0    0     0   27   63  326  0  0 100  0  0
11 2  0      0 6601328  97628 1098788    0    0     0   44   52  299  0  0 100  0  0
```

[复制代码](#)


```
12 0 0      0 6601080 97628 1098792    0    0    0    0 56 285 0 0 100 0 0
```

从输出中你可以看到，内存的 free 列在不停的变化，并且是下降趋势；而 buffer 和 cache 基本保持不变。

未使用内存存在逐渐减小，而 buffer 和 cache 基本不变，这说明，系统中使用的内存一直在升高。但这并不能说明有内存泄漏，因为应用程序运行中需要的内存也可能会增大。比如说，程序中如果用了一个动态增长的数组来缓存计算结果，占用内存自然会增长。

那怎么确定是不是内存泄漏呢？或者换句话说，有没有简单方法找出让内存增长的进程，并定位增长内存用在哪儿呢？

根据前面内容，你应该想到了用 top 或 ps 来观察进程的内存使用情况，然后找出内存使用一直增长的进程，最后再通过 pmap 查看进程的内存分布。

但这种方法并不太好用，因为要判断内存的变化情况，还需要你写一个脚本，来处理 top 或者 ps 的输出。

这里，我介绍一个专门用来检测内存泄漏的工具，memleak。memleak 可以跟踪系统或指定进程的内存分配、释放请求，然后定期输出一个未释放内存和相应调用栈的汇总情况（默认 5 秒）。

当然，memleak 是 bcc 软件包中的一个工具，我们一开始就装好了，执行 /usr/share/bcc/tools/memleak 就可以运行它。比如，我们运行下面的命令：

```
1 # -a 表示显示每个内存分配请求的大小以及地址
2 # -p 指定案例应用的 PID 号
3 $ /usr/share/bcc/tools/memleak -a -p $(pidof app)
4 WARNING: Couldn't find .text section in /app
5 WARNING: BCC can't handle sym look ups for /app
6     addr = 7f8f704732b0 size = 8192
7     addr = 7f8f704772d0 size = 8192
8     addr = 7f8f704712a0 size = 8192
9     addr = 7f8f704752c0 size = 8192
10    32768 bytes in 4 allocations from stack
11        [unknown] [app]
12        [unknown] [app]
13    start_thread+0xdb [libpthread-2.27.so]
```

[复制代码](#)


从 memleak 的输出可以看到，案例应用在不停地分配内存，并且这些分配的地址没有被回收。

这里有一个问题，Couldn't find .text section in /app，所以调用栈不能正常输出，最后的调用栈部分只能看到 [unknown] 的标志。

为什么会有这个错误呢？实际上，这是由于案例应用运行在容器中导致的。memleak 工具运行在容器之外，并不能直接访问进程路径 /app。

比方说，在终端中直接运行 ls 命令，你会发现，这个路径的确不存在：


```
1 $ ls /app
2 ls: cannot access '/app': No such file or directory
```

 复制代码

类似的问题，我在 CPU 模块中的 [perf 使用方法](#) 中已经提到好几个解决思路。最简单的方法，就是在容器外部构建相同路径的文件以及依赖库。这个案例只有一个二进制文件，所以只要把案例应用的二进制文件放到 /app 路径中，就可以修复这个问题。

比如，你可以运行下面的命令，把 app 二进制文件从容器中复制出来，然后重新运行 memleak 工具：


```
1 $ docker cp app:/app /app
2 $ /usr/share/bcc/tools/memleak -p $(pidof app) -a
3 Attaching to pid 12512, Ctrl+C to quit.
4 [03:00:41] Top 10 stacks with outstanding allocations:
5     addr = 7f8f70863220 size = 8192
6     addr = 7f8f70861210 size = 8192
7     addr = 7f8f7085b1e0 size = 8192
8     addr = 7f8f7085f200 size = 8192
9     addr = 7f8f7085d1f0 size = 8192
10    40960 bytes in 5 allocations from stack
11        fibonacci+0x1f [app]
12        child+0x4f [app]
13        start_thread+0xdb [libpthread-2.27.so]
```

 复制代码

这一次，我们终于看到了内存分配的调用栈，原来是 fibonacci() 函数分配的内存没释放。

定位了内存泄漏的来源，下一步自然就应该查看源码，想办法修复它。我们一起来看看案例应用的源代码 [app.c](#)：


```
1 $ docker exec app cat /app.c
2 \.\.\.
3 long long *fibonacci(long long *n0, long long *n1)
4 {
5     // 分配 1024 个长整数空间方便观测内存的变化情况
6     long long *v = (long long *) calloc(1024, sizeof(long long));
7     *v = *n0 + *n1;
8     return v;
9 }
10
11
```

 复制代码

```
12 void *child(void *arg)
13 {
14     long long n0 = 0;
15     long long n1 = 1;
16     long long *v = NULL;
17     for (int n = 2; n > 0; n++) {
18         v = fibonacci(&n0, &n1);
19         n0 = n1;
20         n1 = *v;
21         printf("%dth => %lld\n", n, *v);
22         sleep(1);
23     }
24 }
25 \.\.\.
```


你会发现，`child()` 调用了 `fibonacci()` 函数，但并没有释放 `fibonacci()` 返回的内存。所以，想要修复泄漏问题，在 `child()` 中加一个释放函数就可以了，比如：

```
1 void *child(void *arg)
2 {
3     \.\.\.
4     for (int n = 2; n > 0; n++) {
5         v = fibonacci(&n0, &n1);
6         n0 = n1;
7         n1 = *v;
8         free(v);    // 释放内存
9         printf("%dth => %lld\n", n, *v);
10        sleep(1);
11    }
12 }
```

 复制代码

我把修复后的代码放到了 [app-fix.c](https://github.com/feisky/app-fix.c)，也打包成了一个 Docker 镜像。你可以运行下面的命令，验证一下内存泄漏是否修复：

```
1 # 清理原来的案例应用
2 $ docker rm -f app
3
4 # 运行修复后的应用
5 $ docker run --name=app -itd feisky/app:mem-leak-fix
6
7 # 重新执行 memleak 工具检查内存泄漏情况
8 $ /usr/share/bcc/tools/memleak -a -p $(pidof app)
9 Attaching to pid 18808, Ctrl+C to quit.
10 [10:23:18] Top 10 stacks with outstanding allocations:
11 [10:23:23] Top 10 stacks with outstanding allocations:
```

 复制代码

现在，我们看到，案例应用已经没有遗留内存，证明我们的修复工作成功完成。

小结

总结一下今天的内容。

应用程序可以访问的用户内存空间，由只读段、数据段、堆、栈以及文件映射段等组成。其中，堆内存和内存映射，需要应用程序来动态管理内存段，所以我们必须小心处理。不仅要会用标准库函数 `malloc()` 来动态分配内存，还要记得在用完内存后，调用库函数 `_free()` 来释放它们。

今天的案例比较简单，只用加一个 `free()` 调用就能修复内存泄漏。不过，实际应用程序就复杂多了。比如说，

- `malloc()` 和 `free()` 通常并不是成对出现，而是需要你，在每个异常处理路径和成功路径上都释放内存。
- 在多线程程序中，一个线程中分配的内存，可能会在另一个线程中访问和释放。
- 更复杂的是，在第三方的库函数中，隐式分配的内存可能需要应用程序显式释放。

所以，为了避免内存泄漏，最重要的一点就是养成良好的编程习惯，比如分配内存后，一定要先写好内存释放的代码，再去开发其他逻辑。还是那句话，有借有还，才能高效运转，再借不难。

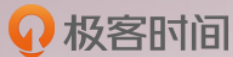
当然，如果已经完成了开发任务，你还可以用 `memleak` 工具，检查应用程序的运行中，内存是否泄漏。如果发现了内存泄漏情况，再根据 `memleak` 输出的应用程序调用栈，定位内存的分配位置，从而释放不再访问的内存。

思考

最后，给你留一个思考题。

今天的案例，我们通过增加 `free()` 调用，释放函数 `fibonacci()` 分配的内存，修复了内存泄漏的问题。就这个案例而言，还有没有其他更好的修复方法呢？结合前面学习和你自己的工作经验，相信你一定能有更多更好的方案。

欢迎留言和我讨论，写下你的答案和收获，也欢迎你把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。



Linux 性能优化实战

10 分钟帮你找到系统瓶颈

倪朋飞

微软资深工程师
Kubernetes 项目维护者



新版升级：点击「👤请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

©版权归极客邦科技所有，未经许可不得转载

上一篇 17 | 案例篇：如何利用系统缓存优化程序的运行效率？

下一篇 19 | 案例篇：为什么系统的Swap变高了（上）

写留言

精选留言



Scott

👍 8

我比较关心老版本的Linux怎么做同样的事，毕竟没有办法升级公司服务器的内核。

2019-01-01

作者回复

另一个用的比较多的是valgrind

2019-01-02



郭江伟

👍 3

本例中将动态分配内存改为使用数组，然后就不需要自己free了；
将app.c拷贝为app2.c 做如下修改，因为篇幅有限没法贴完全代码：

```
long long fibonacci(long long *n0, long long *n1)
```

```
{
```

```
//分配1024个长整数空间方便观测内存的变化情况
```

```
// long long *v = (long long *) calloc(1024, sizeof(long long));
```

```
long long v[1024];
```

然后执行memleak

```
gjl@gjl:~$ sudo /usr/share/bcc/tools/memleak -p $(pidof app2c)
```

Attaching to pid 3463, Ctrl+C to quit.

[13:02:24] Top 10 stacks with outstanding allocations:

[13:02:29] Top 10 stacks with outstanding allocations:

```
^Cgjl@gjl:~$ sudo /usr/share/bcc/tools/memleak -p $(pidof app2c)
```

Attaching to pid 3463, Ctrl+C to quit.

[13:02:43] Top 10 stacks with outstanding allocations:

[13:02:48] Top 10 stacks with outstanding allocations:

[13:02:53] Top 10 stacks with outstanding allocations:

[13:02:58] Top 10 stacks with outstanding allocations:

2018-12-31



我来也

👍 2

[D18打卡]

想不到又有神器可以直接分析出是哪个函数导致了内存泄露。

以前都是在申请和释放的地方加标记，然后用工具去分析。

思考题：

一般能预分配的空间都没必要去动态申请。

这个案例可以把存放结果的值先定义好，函数参数中用指针过去，这样就没必要申请内存了。

2018-12-31



夜空中最亮的星（华仔）

👍 1

老师，代码段里面可否把 代码前面的 \$ 或 # 号，去掉。带着还的手动去掉下才能执行代码

2019-01-02

作者回复

还是需要留着，去掉就不容易区分注释、命令和输出了

2019-01-03



划时代

👍 1

memleak好像要比valgrind进行内存泄漏检测要方便很多。

2019-01-02

作者回复

是的

2019-01-02



mj4ever

👍 1

老师：

遇到了个问题，google也查不出所以然：

1、ubuntu 18.04，内核4.15.0-29-generic

2、运行 memleak -a -p \$(pidof app)，报错：

Attaching to pid 14069, Ctrl+C to quit.

```
perf_event_open(/sys/kernel/debug/tracing/events/uprobes/p_lib_x86_64_linux_gnu_
_libc_2_27_so_0x97070_14069_bcc_14199/id): Input/output error
```

Traceback (most recent call last):

File "/usr/share/bcc/tools/memleak", line 416, in <module>

attach_probes("malloc")

File "/usr/share/bcc/tools/memleak", line 406, in attach_probes

pid=pid)

File "/usr/lib/python2.7/dist-packages/bcc/__init__.py", line 952, in attach_uprobe

raise Exception("Failed to attach BPF to uprobe")

Exception: Failed to attach BPF to uprobe

2019-01-01



code2

👍 1

防止内存泄露，在c中最好让malloc和free成对出现，不要在函数中分配，在函数外释放，这样一不留神就忘了，检查时也不容易发现。也可使用一些源代码内存泄露检测工具。在C++中除了成对出现外还要注意new和delete使用的一些要点。曾遇到过一个投资数千万的大项目，java做的，因内存泄露不能查明原因，服务器不得不每月杀掉服务进程，重新启动。

2019-01-01



萧董

👍 1

memleak输出中一直有addr就是内存没有释放吗

2018-12-31



付盼星

👍 1

老师好，我有个问题想请教下，这里的堆栈和java虚拟机的堆栈是对应起来的么？

2018-12-31



nomoshen

👍 0

目前我司用的内核版本还是2.6的；而且用valgrind会对线上正在执行的程序有很大性能影响吧；对内存泄露这块还是很难把握的；希望老师能x细聊这块

2019-01-03



walker

👍 0

valgrind不是实时的查看命令，把检查结果存入到一个文本中。

```
valgrind --tool=memcheck --leak-check=full --xtree-leak=yes --show-mismatched-frees=yes test.txt
```

2019-01-03



Orcsir

👍 0

Flag

2019-01-02





腾达

0

addr = 7f5d902f66d0 size = 8192

有什么工具能知道7f5d902f66d0这个地址上是什么内容吗?

2019-01-02



腾达

0

在通过make run 启动运行docker后, 尝试使用memleak出现" Failed to attach BPF to uprobe ", 这个错误, 自己google了一下, 未发现明显的解决方案, 请问这个要如何处理? 如果自己直接 gcc编译 app.c, 不用docker, 可以运行memleak

```
root@abbott-VirtualBox:~# memleak -a -p $(pidof app)
```

Attaching to pid 5877, Ctrl+C to quit.

```
perf_event_open(/sys/kernel/debug/tracing/events/uprobes/p_lib_x86_64_linux_gnu_libc_2_27_so_0x97070_5877_bcc_5949/id): Input/output error
```

Traceback (most recent call last):

File "/usr/share/bcc/tools/memleak", line 416, in <module>

attach_probes("malloc")

File "/usr/share/bcc/tools/memleak", line 406, in attach_probes

pid=pid)

File "/usr/lib/python2.7/dist-packages/bcc/__init__.py", line 952, in attach_uprobe

raise Exception("Failed to attach BPF to uprobe")

Exception: Failed to attach BPF to uprobe

2019-01-02



小老鼠

0

用java就不要用free了吧

2019-01-02

作者回复

是的, 但要注意释放对象的引用

2019-01-03



Leon

0

老师, 现在大部分企业公司线上工程运行的都是centos系统, centos一般用valgrind来排查, 我之前遇到一个因为tcp连接的fd遇到异常情况下没有close导致内存泄露, 但是valgrind无法检测出来, 请老师指点下遇到这种情况下怎么排查

2019-01-02



清晓

0

老师您好! 您在文中从前面几篇降到函数, 这些函数是在你打包的文件中吗? 该怎么去查看分析?

2019-01-02

作者回复

是的 可以到github查看案例源码 <https://github.com/feiskyer/linux-perf-examples>

2019-01-02



清晓



老师您好！关于memleak或其他工具，输出的内容里，您可以标注一下重要语句、参数、代码的功能吗？比如：这次的memleak，\$(pidof app)为什么这么写？输出的内容中，从哪里查看分析内存分配回收。

2019-01-02

作者回复

嗯 不错的建议。案例和工具里面有很多基础的东西没有展开细说，不过关键的地方的确标注下更有助于初学者理解

2019-01-02



Brown羊羊



工具对内核要求还是蛮高的

2019-01-02



walker



有没有低端版本的工具来检测内存泄漏的。现在线上的服务器，包括开发测试的服务器都不可能去升级内核。

2019-01-02

作者回复

有的 可以用valgrind

2019-01-02