

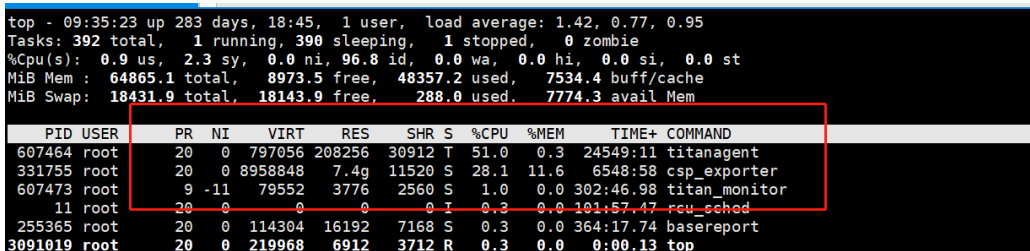
# 一次 Prometheus 采集器内存占用过高（内存泄漏）问题排查及解决

## 一、背景介绍

我参与的一个项目使用了腾讯的云存储组件 CSP。我们需要利用 Prometheus Exporter 对组件进行监控。由于没有找到相关的采集器，我们团队自己开发了一款 CSP Exporter。该采集器由我所在团队的一位大佬开发出了初版。采集器上线两个月后，用户反馈该采集器占用了大量的系统内存，重启后该问题消失。此时该大佬已经离职另谋高就，排查该问题的任务落到了我的头上。经过一周的不懈努力，最终定位到并解决了该问题，以下是排查及解决的过程。

## 二、问题定位

发现问题的过程是用户反馈该采集器占用了系统大量内存资源，如图 1 所示。



```
top - 09:35:23 up 283 days, 18:45, 1 user, load average: 1.42, 0.77, 0.95
Tasks: 392 total, 1 running, 390 sleeping, 1 stopped, 0 zombie
%Cpu(s): 0.9 us, 2.3 sy, 0.0 ni, 96.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 64865.1 total, 8973.5 free, 48357.2 used, 7534.4 buff/cache
MiB Swap: 18431.9 total, 18143.9 free, 288.0 used, 7774.3 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 607464 root        20   0  797056 208256 30912 T  51.0   0.3  24549:11 titanagent
 331755 root        20   0  8958848 7.4g 11520 S  28.1  11.6  6548:58 csp_exporter
 607473 root        9  -11  79552  3776  2560 S   1.0   0.0  302:46.98 titan_monitor
    11 root         0   0     0     0     0 T   0.3   0.0  101:57.47 rcu_sched
255365 root        20   0  114304  16192  7168 S   0.3   0.0  364:17.74 basereport
3091019 root        20   0  219968  6912  3712 R   0.3   0.0   0:00.13 top
```

图 1

刚接到这个问题时，第一反应想到，是不是因为代码里面某个地方申请了资源后没有释放而导致的。因为该采集器是由 GO 语言开发，而 GO 有自己的 GC 机制，所以我当时首先判断动态申请了内存后没有释放的概率是比较小的，因为这一部分 GO 自己完成了。

持续观察一段时间后，发现内存使用是在不断增加的。这一点很可疑，是什么会让资源不断增加？联想到采集器采集数据的机制，每隔 30 秒监控系统的 Prometheus 会向该采集器发起一次请求，采集器在收到请求后，会向在云端的 CSP 组件发起若干请求来获取数据。再回想起以往一次排查 ES 故障的经历，ES 由于打开的文件描述符太多，从而导致占用了过多的内存。这时突然灵光一现，

想到会不会是因为采集器每次与云上的 CSP 建立了 TCP 连接后没有释放。而每建立一次 TCP 连接，Linux 就会打开一个文件描述符，如果打开的 TCP 连接一直没有关闭，就会占用系统资源，包括内存资源。想到这里，立马开始查看该采集器打开的连接情况。首先使用 lsof 命令查看该进程打开的文件数量情况，如图 2。

```
[root@VM-7-157-linux ~]# lsof -c csp_exporter | wc -l
1406243
[root@VM-7-157-linux ~]#
```

图 2

果然，该采集共计打开了一百四十多万个文件，这显然是不正常的。再看打开的都是哪些文件（图 3）：

```
[root@VM-4-15-linux csp_exporter-outside]# lsof -c csp_exporter
COMMAND  PID USER  FD   TYPE    DEVICE SIZE/OFF      NODE NAME
csp_expor 19904 root   cwd   DIR      252,0    4096    1704666 /root/ming/bin/csp/csp_exporter-outside
csp_expor 19904 root   rtd   DIR      252,0    4096         2 /
csp_expor 19904 root   txt   REG     252,0   16126160   1704709 /root/ming/bin/csp/csp_exporter-outside/csp_exporter
csp_expor 19904 root   0r    CHR       1,3         0t0         1030 /dev/null
csp_expor 19904 root   1w    REG     252,0     191   1705186 /root/ming/bin/csp/csp_exporter-outside/nohup_out
csp_expor 19904 root   2w    REG     252,0     191   1705186 /root/ming/bin/csp/csp_exporter-outside/nohup_out
csp_expor 19904 root   3u    REG     252,0    3827   1705286 /root/ming/bin/csp/csp_exporter-outside/console_output.log
csp_expor 19904 root   4u    a_inode 0,13         0    12664 [eventpoll]
csp_expor 19904 root   5r    FIFO    0,12         0t0 376908476 pipe
csp_expor 19904 root   6w    FIFO    0,12         0t0 376908476 pipe
csp_expor 19904 root   7u    IPv6    376908480 0t0      TCP *:wap-wsp-s (LISTEN)
csp_expor 19904 root   8u    IPv6    376906631 0t0      TCP VM-4-15-linux:wap-wsp-s->:13496 (ESTABLISHED)
csp_expor 19904 root   9u    IPv4    376911299 0t0      TCP VM-4-15-linux:44548->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  10u    IPv4    376912242 0t0      TCP VM-4-15-linux:44564->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  11u    IPv4    376910390 0t0      TCP VM-4-15-linux:44590->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  12u    IPv4    376911327 0t0      TCP VM-4-15-linux:44604->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  13u    IPv4    376909459 0t0      TCP VM-4-15-linux:44624->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  14u    IPv4    376913077 0t0      TCP VM-4-15-linux:44638->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  15u    IPv4    376906605 0t0      TCP VM-4-15-linux:44652->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  16u    IPv4    376913088 0t0      TCP VM-4-15-linux:44666->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  17u    IPv4    376912270 0t0      TCP VM-4-15-linux:44684->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  18u    IPv4    376908539 0t0      TCP VM-4-15-linux:44702->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  19u    IPv4    376908549 0t0      TCP VM-4-15-linux:44716->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  20u    IPv4    376911359 0t0      TCP VM-4-15-linux:44734->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  22u    IPv4    376910433 0t0      TCP VM-4-15-linux:44756->169.254.170.2:80 (ESTABLISHED)
csp_expor 19904 root  23u    IPv4    376913104 0t0      TCP VM-4-15-linux:44776->169.254.170.2:80 (ESTABLISHED)
```

图 3

可以看到，输出中基本都是处于 ESTABLISHED 状态的 TCP 连接。为了进一步验证这些连接是会不断增加的，我在本地向该采集器多次发起请求。我观察到，一方面其 process\_open\_fd 指标值是不断增加的，另一方面，lsof -c csp\_exporter 的条目数量也是不断在增加的，且均为状态为 ESTABLISHED 的 TCP 连接，并且连接的地址就是云 CSP 的地址。如图 4、图 5 和图 6

```

process_open_fds 57
[root@VM-4-15-linux csp_exporter-outside]# curl http://localhost:9202/metrics | egrep process_open_fds | egrep -v "#"
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 8129 0 8129 0 0 62572 0 --:--:-- --:--:-- --:--:-- 63015
process_open_fds 58
[root@VM-4-15-linux csp_exporter-outside]# curl http://localhost:9202/metrics | egrep process_open_fds | egrep -v "#"
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 8140 0 8140 0 0 106k 0 --:--:-- --:--:-- --:--:-- 107k
process_open_fds 59
[root@VM-4-15-linux csp_exporter-outside]# curl http://localhost:9202/metrics | egrep process_open_fds | egrep -v "#"
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 8125 0 8125 0 0 80089 0 --:--:-- --:--:-- --:--:-- 79656
process_open_fds 60
[root@VM-4-15-linux csp_exporter-outside]# curl http://localhost:9202/metrics | egrep process_open_fds | egrep -v "#"
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 8130 0 8130 0 0 85343 0 --:--:-- --:--:-- --:--:-- 85673
process_open_fds 61
[root@VM-4-15-linux csp_exporter-outside]# curl http://localhost:9202/metrics | egrep process_open_fds | egrep -v "#"
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 8131 0 8131 0 0 101k 0 --:--:~ --:~:~ --:~:~ 101k
process_open_fds 62
[root@VM-4-15-linux csp_exporter-outside]# curl http://localhost:9202/metrics | egrep process_open_fds | egrep -v "#"
% Total % Received % Xferd Average Speed Time Time Time Current

```

图 4

```

[root@VM-4-15-linux csp_exporter-outside]# lsof -c csp_exporter | wc -l
66
You have mail in /var/spool/mail/root

```

图 5

```

process_open_fds 70
[root@VM-4-15-linux csp_exporter-outside]# lsof -c csp_exporter | wc -l
78
You have mail in /var/spool/mail/root
[root@VM-4-15-linux csp_exporter-outside]#

```

图 6

由此，我们基本可以确定，问题的发生是由于代码某个位置向云端的 CSP 发起请求后建立了 TCP 连接，请求完成后连接没有关闭引起的。定位到了问题，接下来只需找到向 CSP 发起请求的位置并增加一行关闭连接的代码即可。

### 三、 问题解决

该采集器共提供了九个指标，每一个指标数据的获取都是通过 COS SDK 请求云端的 CSP 服务得到，如图 7。

```
96      /* 方法:上传对象的大小/耗时
97      */
98      startTimeWrite := time.Now()
99      key := "exampleobject"
100     resPut, err := instance.Object.Put(
101         context.Background(),key,f,nil,
102     )
103     if err != nil {
104         level.Error(c.log).Log("msg", "读取对象失败", "err", err)
105         wflag = 0
106     }
107
108     /* 测试下载速度
109     * 方法:下载对象的大小/耗时
110     */
111
112     timeDurationWrite := time.Since(startTimeWrite).Seconds()
113     wflag = 1
114     writeBytePerSec := Decimal(float64(objectSize) / timeDurationWrite)
115
116     err = os.Remove("/tmp/csp_tmp.data")
117     if err != nil {
118         level.Error(c.log).Log("msg", "删除读写测试文件失败", "err", err)
119     }
120
121     startTimeRead := time.Now()
122     resGet, err := instance.Object.Get(context.Background(),key,nil)
123     if err != nil {
124         level.Error(c.log).Log("msg", "读取对象失败", "err", err)
125         rflag = 0
126     }
127
```

图 7

第 100 行代码向云端 CSP 服务发起请求获取数据。直到代码的末尾都没有看到客户端主动关闭连接的代码。从第 161 行到结束，我们增加了客户端主动断开连接的代码。每一次获取到云端 CSP 返回的数据后主动断开连接。

```
135     wflag,
136 )
137 ch <- prometheus.MustNewConstMetric(
138     CspWriteSpeed,
139     prometheus.GaugeValue,
140     writeBytePerSec,
141 )
142
143 ch <- prometheus.MustNewConstMetric(
144     CspReadSuccess,
145     prometheus.GaugeValue,
146     rflag,
147 )
148
149 ch <- prometheus.MustNewConstMetric(
150     CspReadSpeed,
151     prometheus.GaugeValue,
152     readBytePerSec,
153 )
154
155
156 resDelete_err := instance.Object.Delete(context.Background(),key)
157 if err != nil {
158     level.Error(c.log).Log("msg", "删除对象失败", "err", err)
159 }
160
161 resPut.Body.Close()
162 resGet.Body.Close()
163 resDelete.Body.Close()
164 return nil
165 }
166
```

图 8

在增加了这几行代码后，我们更新了生产环境的采集器，运行一段时间后我们观察到新采集器的文件描述符打开数一直稳定在 13（图 9），并且内存使用情况也趋于稳定。因此我判断我们的解决方法生效了。

```
[root@VM-7-157-linux ~]# lsof -p 2172708 | wc -l
13
[root@VM-7-157-linux ~]# 等待输入超时: 自动注销
```

图 9

至此，团队内部开发的 CSP 采集器内存资源占用过高的问题（内存泄漏）初步判断已解决。最终确定发生的原因是：采集器没有主动关闭与云端 CSP 组件

建立的 TCP 连接导致占用系统资源；解决方式是在代码中增加主动关闭连接的代码。