3.5. 内存跟踪调用

前面提到了关于 inode 和一些接口的事情 ,那么这一节主要想了解一下关于内存相关的，因为内存泄露的问题，在系统开发中会时不时地出现，尤其是因为 glusterfs 使用 C 语言进行开发的，并没有 java 和 python 这些语言天然自带的内存回收管理机制。

3.5.1. mem_acct

在前面介绍 xlator 的内容中，提到了关于_xlator 结构，这个就是每个功能模块的数据结构定义，在这个结构里面，有一个和该模块的内存相关的变量，如下所示。

```
1.  struct _xlator {
2.        ...
3.        int32_t (*mem_acct_init)(xlator_t *this);
4.        struct mem_acct *mem_acct;
5.        ...
6.  }
```

在这个结构里面的 mem_acc_init 是一个函数，在 xlator 初始化的时候和调用的时候，会记录该 xlator 的一些内存相关的信息，那么这里还有一个相关的数据结构，就是 men_acct 了，这个结构定义在 mem-pool.h 文件中，这里的定义如下所示。

```
1.  struct mem_acct_rec {
2.      const char *typestr;
3.      uint64_t size;
4.      uint64_t max_size;
5.      uint64_t total_allocs;
6.      uint32_t num_allocs;
7.      uint32_t max_num_allocs;
```

```
        gf_lock_t lock;
        ...
};

struct mem_acct {
        uint32_t num_types;
        gf_atomic_t refcnt;
        struct mem_acct_rec rec[0];
};
```

从上面的内容中可以看到，这里记录了总的内存容量，最大可申请的容量等变量，那么下面就来简单了解一下，每个 xlator 初始化的时候的函数调用。

3.5.2. xlator_init

每一个 xlator 在初始化的时候，需要调用 xlator_init 函数，这个函数定义在 xlator.c 中，在这个函数中，就会对该 xlator 的内存做一些初始化的工作，下面来简单了解一下。

```
int
xlator_init(xlator_t *xl)
{
        ...
        if (xl->mem_acct_init)
           xl->mem_acct_init(xl);
        ...
}
```

这里就是会调用 xlator 的 mem_acct_init 函数，那么这里和前面提到的 xlator_api 中的 mem_acct_init 就对应上了，这里会关联每个 xlator 的对应的函数，例如对于 afr 来说，这里就是 afr.c 中的 mem_acct_init 函数了，如下所示。

```
int32_t
```

```
2.  mem_acct_init(xlator_t *this)
3.  {
4.      int ret = -1;
5.
6.      if (!this)
7.          return ret;
8.
9.      ret = xlator_mem_acct_init(this, gf_afr_mt_end + 1);
10.
11.     if (ret != 0) {
12.         return ret;
13.     }
14.
15.     return ret;
16. }
```

那么这里调用的 xlator_mem_acct_init 中的第二个参数 gf_afr_mt_end 又是什么呢？这个是每个 xlator 定义的 mem types，因为对于不同的模块来说，需要关注的内存的类型是不同的，例如 afr 模块，这里需要考虑定义不同的内存类型的，这里的定义在 afr-mem-types.h 文件中， 那么接着这里进入到 xlator_mem_acct_init 函数中查看一下实现。

```
1.  int
2.  xlator_mem_acct_init(xlator_t *xl, int num_types)
3.  {
4.      int i = 0;
5.      int ret = 0;
6.      ...
7.      xl->mem_acct = MALLOC(sizeof(struct mem_acct) +
8.                            sizeof(struct mem_acct_rec) * n
    um_types);
9.
10.     ...
11.     xl->mem_acct->num_types = num_types;
12.     GF_ATOMIC_INIT(xl->mem_acct->refcnt, 1);
13.
14.     for (i = 0; i < num_types; i++) {
```

```
15.          memset(&xl->mem_acct->rec[i], 0, sizeof(struct me
   m_acct_rec));
16.          ret = LOCK_INIT(&(xl->mem_acct->rec[i].lock));
17.          if (ret) {
18.               fprintf(stderr, "Unable to lock..errno : %d",
    errno);
19.          }
20. #ifdef DEBUG
21.          INIT_LIST_HEAD(&(xl->mem_acct->rec[i].obj_list));

22. #endif
23.      }
24.
25.      return 0;
26. }
```

那么这里会做一些安全检查，并且调用 MALLOC,这个是一个宏定义，最后还是会调用 malloc 系统函数，然后申请一些内容，接着就是会进行 memset,也就是一些简单的信息设置。

那么做完了这些，对于一个 xlator 来说，内存初始化部分基本算完成了，而在系统运行中，这里要申请内存的话，就要使用到另外一个东西，叫做 GF_CALLOC,在下一节中将会讲解。

3.5.3. GF_CALLOC

这里 GF_CALLOC 是一个宏定义，这里的定义在 mem-pool.h 文件中，定义如下所示。

```
1.  #define GF_CALLOC(nmemb, size, type) __gf_calloc(nmemb, s
   ize, type, #type)
2.
3.  #define GF_MALLOC(size, type) __gf_malloc(size, type, #ty
   pe)
4.
5.  #define GF_REALLOC(ptr, size) __gf_realloc(ptr, size)
```

```
6.
7.  #define GF_FREE(free_ptr) __gf_free(free_ptr)
```

这里 MALLOC 和 CALLOC 其实是类似的，只是这里的参数不一样，多了一个 nmemb,这个 是调用系统函数 calloc 的时候的参数，表示元素的个数的。

```
1.  server = GF_CALLOC(1, sizeof(server_cmdline_t),gf_common_
    mt_server_cmdline_t);
```

那么这里调用的方式，随便中代码中找到的一个，从这里可以看到，就是传入了想要创建申请内存的元素个数为 1，然后大笑为 sizeof 判断的，最后的就是 type，也就是申请的类型。

那么这里 GF_CALLOC 这里的定义的一个函数__gf_calloc，那么这里就可以看看该函数的实现。

```
1.  void *
2.  __gf_calloc(size_t nmemb, size_t size, uint32_t type, con
    st char *typestr)
3.  {
4.      size_t tot_size = 0;
5.      size_t req_size = 0;
6.      void *ptr = NULL;
7.      xlator_t *xl = NULL;
8.
9.      if (!gf_mem_acct_enabled())
10.         return CALLOC(nmemb, size);
11.
12.     xl = THIS;
13.
14.     req_size = nmemb * size;
15.     tot_size = req_size + GF_MEM_HEADER_SIZE + GF_MEM_TRA
    ILER_SIZE;
16.
17.     ptr = calloc(1, tot_size);
18.
19.     if (!ptr) {
20.         gf_msg_nomem("", GF_LOG_ALERT, tot_size);
21.         return NULL;
```

```
    }

     return gf_mem_set_acct_info(xl->mem_acct, ptr, req_size, type, typestr);
}
```

那么这里可以看到，最终就是调用了系统函数 calloc 进行申请内存，而 return 之前，还会调用 gf_mem_set_acct_info 函数，这个函数就是用来设置一下内存相关的一些信息的，这里就和前面提到的 mem_acct 的结构这些有关联了。

```
 static void *
 gf_mem_set_acct_info(struct mem_acct *mem_acct, struct mem_header *header,
                      size_t size, uint32_t type, const char *typestr)
 {
     struct mem_acct_rec *rec = NULL;
     bool new_ref = false;

     if (mem_acct != NULL) {
         GF_ASSERT(type <= mem_acct->num_types);

         rec = &mem_acct->rec[type];
         LOCK(&rec->lock);
         {
             if (!rec->typestr) {
                 rec->typestr = typestr;
             }
             rec->size += size;
             new_ref = (rec->num_allocs == 0);
             rec->num_allocs++;
             rec->total_allocs++;
             rec->max_size = max(rec->max_size, rec->size);

             rec->max_num_allocs = max(rec->max_num_allocs, rec->num_allocs);

             ..
         }
```

```
26.         UNLOCK(&rec->lock);
27.
28.          ...
29.     }
30.
31.     header->type = type;
32.     header->mem_acct = mem_acct;
33.     header->magic = GF_MEM_HEADER_MAGIC;
34.
35.     return gf_mem_header_prepare(header, size);
36. }
```

这里可以看到，每次申请了内存之后，num_allocs 和 total_allocs 这些都会自增，另外这里还可以看到有另外一个结构，叫做 header，这个结构是哪里来的呢，就是调用了系统函数 calloc 之后返回值 ptr,而 ptr 则是一个指针，这个指针指向的是分配内存的地址 ,而在释放内存的时候 ,也需要用到这个 ptr 的，否则是无法找到之前申请的内存的，而 glusterfs 中则是使用了 mem_header 这个结构来进行记录。另外处理完成了这个函数之后，可以看到最后还会调用 gf_mem_header_prepare 这个函数。

3.5.4. GF_FREE

那么这里有申请内存的，就肯定会有释放内存，而释放内存就是 GF_FREE，这里的定义比较简单，另外这里实际调用的函数是__gf_free，下面来简单了解一下。

```
1.  void
2.  __gf_free(void *free_ptr)
3.  {
4.      void *ptr = NULL;
5.      struct mem_acct *mem_acct;
6.      struct mem_header *header = NULL;
7.      bool last_ref = false;
8.
```

```
9.      if (!gf_mem_acct_enabled()) {
10.         FREE(free_ptr);
11.         return;
12.     }
13.
14.     if (!free_ptr)
15.         return;
16.
17.      ...
18.
19.     LOCK(&mem_acct->rec[header->type].lock);
20.     {
21.          mem_acct->rec[header->type].size -= header->size;

22.         mem_acct->rec[header->type].num_allocs--;
23.
24.         if (!mem_acct->rec[header->type].num_allocs) {
25.             last_ref = true;
26.             mem_acct->rec[header->type].typestr = NULL;
27.         }
28. #ifdef DEBUG
29.         list_del(&header->acct_list);
30. #endif
31.     }
32.     UNLOCK(&mem_acct->rec[header->type].lock);
33.
34.     if (last_ref) {
35.         xlator_mem_acct_unref(mem_acct);
36.     }
37.
38. free:
39. #ifdef DEBUG
40.     __gf_mem_invalidate(ptr);
41. #endif
42.
43.     FREE(ptr);
44. }
```

那么这里释放的函数是 FREE, 并且可以和前面申请时的函数 gf_mem_set_acct_info 进行对比，这里可以看到就是做相反的操作，把记录的申请数减少 1 等。

3.5.5. statedump

提到了可以内存相关的东西，那么这里接着就不得不讲到 statedump 了，这个是 glusterfs 提供的一个命令 ,就是用来获取当前 volume 的内存状态情况的 ,对于内存是否存在的泄露等提供非常大的帮助 , 那么下面先简单来感受一下这个命令的作用。

```
root@gfs01:~#  gluster --print-statedumpdir
/var/run/gluster
```

这里首先要知道 statedump 的报告输出目录在哪里，然后这里有两种方式获取 volume 的 statedump 状态信息，一种是找到每个 volume 对应节点的 brick 的 pid，可以通过 status 命令知道，然后使用 kill -USR1 <pid> 的方式，获取当前节点的 volume brick 的 statedump,而如果想获取该 volume 的全部 brick 的，则可以使用下面的命令。

```
root@gfs01:~# gluster volume statedump test-disperse
volume statedump: success

root@gfs01:~# ls /var/run/gluster
...
glusterfs-test-disperse.132218.dump.1624803759
```

那么这里关于 statedump 的内容有都有哪些内容呢？下面可以来看看。

```
root@gfs01:~# cat /var/run/gluster/glusterfs-test-dispers
e.132218.dump.1624803759
DUMP-START-TIME: 2021-06-27 14:22:39.732433 +0000

[mallinfo]
mallinfo_arena=11046912
mallinfo_ordblks=431
```

```
7.  mallinfo_smblks=23
8.  mallinfo_hblks=8
9.  mallinfo_hblkhd=2105344
10. mallinfo_usmblks=0
11. mallinfo_fsmblks=2080
12. mallinfo_uordblks=1596384
13. mallinfo_fordblks=9450528
14. mallinfo_keepcost=54368
15. ...
```

首先这里的文件名称是有规范的，132218 就是当前 volume brick 对应的 pid 了，然后 mallinfo 的内容，这里是通过调用系统函数 mallinfo 获取的，这里 mallinfo_uordblks 表示进程总的分配的空间，单位是 bytes，而 mallinfo_hblks 则表示 mmaped 区域的数量，关于这部分的参数的详细解释，其实是和 mallinfo 这个系统函数对应的，可以使用命令 man mallinfo 查看一下。

```
1.  [global.glusterfs - Memory usage]
2.  num_types=125
3.
4.  [global.glusterfs - usage-type gf_common_mt_dnscache6 mem
    usage]
5.  size=16
6.  num_allocs=1
7.  max_size=16
8.  max_num_allocs=1
9.  total_allocs=1
10.
11. [global.glusterfs - usage-type gf_common_mt_event_pool me
    musage]
12. size=20696
13. num_allocs=3
14. max_size=20696
15. max_num_allocs=3
16. total_allocs=3
17. ....
```

这里第二部分是关于 glusterfs 这个模块的内存申请情况，这里下面会有不

同的数据类型 data type 的内存使用情况，这里可以多留意一下 num_allocs 的数值，如果这个数值一直在增加，那么可能会存在内存泄露的情况。

```
[mempool]
-----=-----
pool-name=struct saved_frame
xlator-name=test-disperse-quota
active-count=0
sizeof-type=88
padded-sizeof=128
size=0
shared-pool=0x7f7c127c4060
-----=-----
...
pool-name=inode_t
xlator-name=test-disperse-server
active-count=2
sizeof-type=168
padded-sizeof=256
size=512
shared-pool=0x7f7c127c4088
...
```

第三部分就是关于内存池的，内存池是一种旨在减少数据类型分配数量的优化。通过为一个数据类型创建一个包含 1024 个元素的内存池，只有当池中的 1024 个元素都处于活动状态时，才会使用 calloc 之类的系统调用从堆中分配该类型的新元素。这里的作用其实和大家常见的线程池是类似的。

那么这里可以看到每一个 pool-name 都是一部分，像 inode_t 这里，下面的 active-count 不为 0 ,就是表示目前有 2 个正在活跃或者使用的 inode_t 对象。

另外这里还有其他很多不同的内存信息，如 iobuf,fuse operation 等，具体地可以自行阅读，根据需要来查找排查问题等。

3.5.6. io thread

这里提到了内存相关的，那么 io thread,也就是 io 线程的问题也是需要关注的 ,这里涉及到 fop 操作的优先级 ,还有线程池等问题 ,这些都是值得关注的，下面来简单了解一下。

```
1.  struct iot_conf {
2.      pthread_mutex_t mutex;
3.      pthread_cond_t cond;
4.
5.       //最大的线程池可运行线程数量
6.      int32_t max_count;
7.     //现在实际正在运行的线程的数量
8.      int32_t curr_count;
9.      int32_t sleep_count;
10.
11.      ...
12.     //线程池的一个等待时间,如果超过这个时间没有 fop 进行处理，线程池会保持最小线程数运行
13.     int32_t idle_time;
14.      ...
15.
16.      //是否开启 least-priority,这个参数的作用是区分客户端内部的 fop 优先级这些的
17.     gf_boolean_t least_priority;
18.     ...
19. };
```

对于 io 线程，这里首先需要简单了解一下 iot_conf 数据结构，这个结构定义的就是线程的 conf,也包括线程池的一些配置等等，在初始化的时候，这里需把一些参数进行设置然后运行。下面来简单了解一下 io thread 初始化的函

数。

```
1.  int
2.  init(xlator_t *this)
3.  {
4.      iot_conf_t *conf = NULL;
5.      int ret = -1;
6.      int i = 0;
7.
8.      //下面进行一些安全检查
9.      if (!this->children || this->children->next) {
10.         gf_smsg("io-threads", GF_LOG_ERROR, 0,
11.                 IO_THREADS_MSG_XLATOR_CHILD_MISCONFIGURED,
    NULL);
12.         goto out;
13.     }
14.     ...
15.
16.     //下面对一些参数配置进行初始化设置
17.      GF_OPTION_INIT("thread-count", conf->max_count, int32, o
    ut);
18.      ...
19.
20.     if (!this->pass_through) {
21.         //这里就是对线程池进行调整线程.对于初始化，就是创建线程
22.         ret = iot_workers_scale(conf);
23.         ...
24.     }
25.
26.     this->private = conf;
27.
28.     conf->watchdog_secs = 0;
29.      GF_OPTION_INIT("watchdog-secs", conf->watchdog_secs, in
    t32, out);
30.     if (conf->watchdog_secs > 0) {
31.         start_iot_watchdog(this);
32.     }
33.
34.     ret = 0;
35. out:
36.     if (ret)
```

```
37.         GF_FREE(conf);
38.
39.     return ret;
40. }
```

上面的代码是在 xlator/performance/io-threads/io-threads.c 这个文件中,这里主要就是初始化设置一些参数,然后就进行创建一些线程了。

那么每一个 xlator 都会有对应的 xlator_api 结构，在这里有一个 fops 的参数，对应的就是当前 xlator 的 xlator_fops，对于 io threads 模块来说，这里有很多的 fop,其中包括 iot_open,iot_create,iot_writev 等，而观察这些 fop 的实现，可以知道这里大部分调用的都是 IOT_FOP，如下所示。

```
1.  int
2.  iot_open(call_frame_t *frame, xlator_t *this, loc_t *loc, int32_t
    flags,
3.           fd_t *fd, dict_t *xdata)
4.  {
5.      IOT_FOP(open, frame, this, loc, flags, fd, xdata);
6.      return 0;
7.  }
8.
9.  int
10. iot_create(call_frame_t *frame, xlator_t *this, loc_t *loc, int32_
    t flags,
11.            mode_t mode, mode_t umask, fd_t *fd, dict_t *xdata)
12. {
13.     IOT_FOP(create, frame, this, loc, flags, mode, umask, fd, xd
    ata);
14.     return 0;
15. }
```

这里调用的时候可以看到，IOT_FOP 的第一个参数是用来区分不同的 fop 的，下面再进入其中了解一下。

```
1.  #define
    IOT_FOP(name,frame,this,args...)
```

```
2.  do{
3.    call_stub_t *__stub = NULL;
4.      int __ret = -1;
5.
6.    __stub = fop_##name##_stub(frame, default_##name##_r
    esume, args);
7.      if(!__stub){
8.          __ret = -ENOMEM;
9.          goto out;
10.     }
11.
12.     __ret = iot_schedule(frame, this, __stub);
13.
14. out:
15.     if (__ret < 0) {
16.         default_##name##_failure_cbk(frame, -__ret);
17.         if(__stub!=NULL){
18.             call_stub_destroy(__stub);
19.         }
20.     }
21. } while (0)
```

这里有一个 fop##name##_stub 的东西，这个就是根据传入的 fop 进行动态替换的，如传入的 name 是 writev,那么这里调用的函数就是 fop_writev_stub,这个函数的定义在 libglusterfs/src/call-stub.c 中定义。另外 stub 又是什么呢？这个可以理解为封装的一个数据结构,封装了一些 fop 的，感兴趣的可以查看 call-stub.h 文件中的定义。

那么这里回到 IOT_FOP 中 ,这里下面就是进入到 iot_schedule 这个函数中了 ,这个函数主要是弄一些 fop 的优先级的 ,因为对于不同的 fop 操作，优先级是有区别的,设置优先级的原因是 ,对于一些客户端的查询响应希望能够快一点，那样能提高客户端的体验。

在 iot_schedule 函数中，这里后面会执行 do_iot_schedule，从这个

函数开始，就是处理和线程有关的事情了，下面进入简单了解一下。

```
1.  int
2.  do_iot_schedule(iot_conf_t *conf, call_stub_t *stub, int pri)
3.  {
4.      int ret = 0;
5.
6.      pthread_mutex_lock(&conf->mutex);
7.      {
8.          __iot_enqueue(conf, stub, pri);
9.
10.         pthread_cond_signal(&conf->cond);
11.
12.         ret = __iot_workers_scale(conf);
13.     }
14.     pthread_mutex_unlock(&conf->mutex);
15.
16.     return ret;
17. }
```

这里的__iot_enqueue 函数就是通过配置 conf,还有封装的 stub 和优先级 pri 来设置队列情况了，这里每个 xlator 模块都有一个队列的，然后会把任务根据优先级插入到队列中。接着下面的__iot_workers_scale 就根据配置和实际的任务情况，来调整线程池的线程数量了。下面继续进入到__iot_workers_scale 中了解一下。

```
1.  int
2.  __iot_workers_scale(iot_conf_t *conf)
3.  {
4.      int scale = 0;
5.      int diff = 0;
6.      pthread_t thread;
7.      int ret = 0;
8.      int i = 0;
9.
10.     for (i = 0; i < GF_FOP_PRI_MAX; i++)
11.         scale += min(conf->queue_sizes[i], conf->ac_iot_limit[i]);
```

```
12.
13.    if (scale < IOT_MIN_THREADS)
14.        scale = IOT_MIN_THREADS;
15.
16.    if (scale > conf->max_count)
17.        scale = conf->max_count;
18.
19.    if (conf->curr_count < scale) {
20.        diff = scale - conf->curr_count;
21.    }
22.
23.    while (diff) {
24.        diff--;
25.
26.     ret = gf_thread_create(&thread, &conf->w_attr, iot_worker,
    conf,
27.                            "iotwr%03hx", conf->curr_count & 0x3ff);
28.        if (ret == 0) {
29.            pthread_detach(thread);
30.            conf->curr_count++;
31.            gf_msg_debug(conf->this->name, 0,
32.                         "scaled threads to %d (queue_size=%d/%d)",
33.                         conf->curr_count, conf->queue_size, scale);
34.        } else {
35.            break;
36.        }
37.    }
38.
39.    return diff;
40.}
```

这里可以看到首先是判断要调整的线程数据和不同，然后进入 while 循环进行调整，在这里调用的是 gf_thread_create 函数，这个函数里面有一个参数 iot_worker,iot_worker 是每个 io-thread 执行的函数。这是一个线程池实现。大致是这样的工作：

1. 每个线程从第一个客户端的优先级队列的 fop 列表中取出一个 fop。

2. 在使 fop 出列后，下一个客户端会到达客户端列表的头部，以便下一个线程将从该客户端队列中选择 fop。

3. 它执行 fop 并转到第 1 步

4. 如果在任何时候队列中没有 fops，则线程等待 idle_time（默认配置为 120 秒），如果在此期间没有任何操作，只要最小线程数仍然存在，线程就会终止跑步。

5. 最低优先级用于内部客户端，如自我修复守护程序和重新平衡。只有当没有来自用户的 I/O 时，才会接收来自这些内部客户端的 I/O。

所以从这里就不难理解，为什么有时候设置关于 threads 的参数，例如 config.client-threads 和 config.brick-threads 这些参数 ,并没有看到明显的效果，因为这里可能当前的 IO 任务不多的时候，并不会扩大线程池的情况。