# 1. OOM

Oom的全称是out-of-memory，是内核在处理系统内存不足而又回收无果的情况下采取的一种措施，内核会经过选择杀死一些进程，以释放一些内存，满足当前内存申请的需求。

所以oom是一种系统行为，对应到memcg的oom，其原理和动机跟全局oom是一样的，区别只在于对象的不同，全局oom的对象是整个系统中所有进程，而memcg oom只针对memcg中的进程（如果使能了hierarchy，还包括所有子memcg中的进程），这里的对象主要是指oom时内核选择从哪些进程中杀死一些进程，所以memcg的oom**只**可能杀死**属于该memcg的进程**。

# 2. 猜测

linux 内存管理中oom killer机制存在于分配内存的 __alloc_pages_slowpath() 阶段

所以猜测 memcg的oom kill机制是在 charge (统计计数)阶段

# 3. 查看最初的代码

通过 tig mm/memcontrol.c 查看最开始代码，能得到oom最初的代码

## 3.1. 相关commit

Memory controller: OOM handling, c7ba5c9e8176704bfac0729875fa62798037584d, 2008-02-07, 08:42

> oom(Out of memory)处理用于当cgroup超过其限制. 从超过限制的cgroup中拿到一个进程, 利用现有的oom逻辑kill掉这个进程.

## 3.2. 代码分析

```
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -329,6 +329,7 @@ int mem_cgroup_charge(struct page *page, struct mm_struct *mm)
                }

                css_put(&mem->css);
+               mem_cgroup_out_of_memory(mem, GFP_KERNEL);
                goto free_pc;
        }
```

reset回这个commit查看.

```
// mm/memcontrol.c
int mem_cgroup_charge(struct page *page, struct mm_struct *mm)
{
        ......
        /*
         * If we created the page_cgroup, we should free it on exceeding
         * the cgroup limit.
         */
        // 下面逻辑都是创建了新的 page_cgroup, 即
        // 当前memcg使用的内存统计计数 + PAGE_SIZE
        // 如果大于这个memcg的limit, 则进入里面的流程
        while (res_counter_charge(&mem->res, PAGE_SIZE)) {
                // 回收释放这个memcg的page, 释放成功返回1(即进行下一次循环)
                if (try_to_free_mem_cgroup_pages(mem))
                        continue;

                /*
                 * try_to_free_mem_cgroup_pages() might not give us a full
                 * picture of reclaim. Some pages are reclaimed and might be
                 * moved to swap cache or just unmapped from the cgroup.
                 * Check the limit again to see if the reclaim reduced the
                 * current usage of the cgroup before giving up
                 */
                // 上面调用可能不会提供完整的回收信息
                // 一些页面被回收可能仅仅被移至交换缓存或仅从cgroup取消映射
                // 在放弃回收之前, 再次检查限制, 查看回收是否减少了cgroup的当前使用率
                // 在limit内返回true(下一次循环), 否则false(高于limit)
                if (res_counter_check_under_limit(&mem->res))
                        continue;
                        /*
                         * Since we control both RSS and cache, we end up with a
                         * very interesting scenario where we end up reclaiming
                         * memory (essentially RSS), since the memory is pushed
                         * to swap cache, we eventually end up adding those
                         * pages back to our list. Hence we give ourselves a
                         * few chances before we fail
                         */
                //5次的回收机会
                else if (nr_retries--) {
                        congestion_wait(WRITE, HZ/10);
                        continue;
                }

                css_put(&mem->css);
                // 回收失败则oom killer
                mem_cgroup_out_of_memory(mem, GFP_KERNEL);
                goto free_pc;
        }
        ......
}
```

多次回收失败则调用了 mem_cgroup_out_of_memory(mem, GFP_KERNEL); ,

```
//mm/oom_kill.c

#ifdef CONFIG_CGROUP_MEM_CONT
void mem_cgroup_out_of_memory(struct mem_cgroup *mem, gfp_t gfp_mask)
{
        unsigned long points = 0;
        struct task_struct *p;

        cgroup_lock();
        rcu_read_lock();
retry:
        // 找出该memcg下最该被kill的进程
        p = select_bad_process(&points, mem);
        if (PTR_ERR(p) == -1UL)
                goto out;

        if (!p)
                p = current;
        // 杀掉选中的进程及与其共用mm的进程
        // 杀进程的目的是释放内存，所以当然要把mm的所有引用都干掉
        // 里面的实现会优先kill子进程
        // 不成功，则重试
        if (oom_kill_process(p, gfp_mask, 0, points,
                                "Memory cgroup out of memory"))
                goto retry;
out:
        rcu_read_unlock();
        cgroup_unlock();
}
#endif
```

跟全局oom一样，memcg的oom也分成 select_bad_process 和 oom_kill_process 两个过程，而这两个都直接使用了内核的函数.

这里 select_bad_process() 只不过多加了个参数，用来兼容memcg.

```
--- a/mm/oom_kill.c
+++ b/mm/oom_kill.c
@@ -25,6 +25,7 @@
 #include <linux/cpuset.h>
 #include <linux/module.h>
 #include <linux/notifier.h>
+#include <linux/memcontrol.h>

 int sysctl_panic_on_oom;
 int sysctl_oom_kill_allocating_task;
@@ -50,7 +51,8 @@ static DEFINE_SPINLOCK(zone_scan_mutex);
  *    of least surprise ... (be careful when you change it)
  */

-unsigned long badness(struct task_struct *p, unsigned long uptime)
+unsigned long badness(struct task_struct *p, unsigned long uptime,
+                      struct mem_cgroup *mem)
 {
        unsigned long points, cpu_time, run_time, s;
        struct mm_struct *mm;
@@ -63,6 +65,13 @@ unsigned long badness(struct task_struct *p, unsigned long uptime)
                return 0;
        }
        // 关键部分
+#ifdef CONFIG_CGROUP_MEM_CONT
        // 在memcg情况下，如果mm的memcg不是当前这个，则不处理，返回
+       if (mem != NULL && mm->mem_cgroup != mem) {
+               task_unlock(p);
+               return 0;
+       }
+#endif
+
        /*
         * The memory size of the process is the basis for the badness.
         */
@@ -193,7 +202,8 @@ static inline enum oom_constraint constrained_alloc(struct zonelist
  *
  * (not docbooked, we don't want this one cluttering up the manual)
  */
-static struct task_struct *select_bad_process(unsigned long *ppoints)
+static struct task_struct *select_bad_process(unsigned long *ppoints,
+                                              struct mem_cgroup *mem)
 {
        struct task_struct *g, *p;
        struct task_struct *chosen = NULL;
@@ -247,7 +257,7 @@ static struct task_struct *select_bad_process(unsigned long *ppoints
                if (p->oomkilladj == OOM_DISABLE)
                        continue;

-               points = badness(p, uptime.tv_sec);
+               points = badness(p, uptime.tv_sec, mem);
```

```
                if (points > *ppoints || !chosen) {
                        chosen = p;
                        *ppoints = points;
```

至此第一版的memcg oom killer代码分析结束.

# 4. 最新的方案

基于:

```
VERSION = 5
PATCHLEVEL = 11
SUBLEVEL = 0
EXTRAVERSION = -rc4
```

```
static int try_charge(struct mem_cgroup *memcg, gfp_t gfp_mask,
                      unsigned int nr_pages)
{
        ......
        /*
         * keep retrying as long as the memcg oom killer is able to make
         * a forward progress or bypass the charge if the oom killer
         * couldn't make any progress.
         */
        // 只要memcg oom Killer能够取得前进，就可以持续重试
        // 如果oom killer无法取得任何进展，则绕开charge.
        oom_status = mem_cgroup_oom(mem_over_limit, gfp_mask,
                        get_order(nr_pages * PAGE_SIZE));
        switch (oom_status) {
        case OOM_SUCCESS:
                // oom成功，持续重试
                nr_retries = MAX_RECLAIM_RETRIES;
                goto retry;
        case OOM_FAILED:
                // oom失败
                goto force;
        default:
                goto nomem;
        }
nomem:
        if (!(gfp_mask & __GFP_NOFAIL))
                return -ENOMEM;
force:
        /*
         * The allocation either can't fail or will lead to more memory
         * being freed very soon.  Allow memory usage go over the limit
         * temporarily by force charging it.
         */
        page_counter_charge(&memcg->memory, nr_pages);
        if (do_memsw_account())
                page_counter_charge(&memcg->memsw, nr_pages);

        return 0;
        ......
```

# 5.

通过 `struct mem_cgroup` 中oom相关的结构体变量以及 `mm/memcontrol.c` 中相关变量（ `memcg_oom_mutex` 等），查找最初的memcg oom killer代码

> 因为结构体位置有变化(从 mm/memcontrol.c 到了 include/linux/memcontrol.h )，以及代码覆盖情况(时间太久了)，所以有过多次reset动作

先是关注 struct mem_cgroup 的变量，如下

```
struct mem_cgroup {
    /* OOM-Killer disable */
    int             oom_kill_disable;

    /* For oom notifier event fd */
    struct list_head oom_notify;
}
```

得到最终patch set, 2010-05-26, 14:42

- memcg: oom wakeup filter, dc98df5a1b7be402a0e1c71f1b89ccf249ac15ee
- memcg: oom notifier, 9490ff275606da012d5b373342a49610ad61cb81
- memcg: oom kill disable and oom status, 3c11ecf448eff8f12922c498b8274ce98587eb74

但是查看git show

- memcg: fix oom kill behavior, 867578cbccb0893cc14fc29c670f7185809c90d6, 2010-03-10 15:22

最终代码:

- arch: mm: remove obsolete init OOM protection, 94bce453c78996cc4373d5da6cfabe07fcc6d9f9
- arch: mm: do not invoke OOM killer on kernel fault OOM, 871341023c771ad233620b7a1fb3d9c7031c4e5c
- arch: mm: pass userspace fault flag to generic fault handler, 759496ba6407c6994d6a5ce3a5e74937d7816208
- x86: finish user fault error path with fatal signal, 3a13c4d761b4b979ba8767f42345fed3274991b0
- mm: memcg: enable memcg OOM killer only for user faults, 519e52473ebe9db5cdef44670d5a97f1fd53d721
- mm: memcg: rework and document OOM waiting and wakeup, fb2a6fc56be66c169f8b80e07ed999ba453a2db2
- mm: memcg: do not trap chargers with full callstack on OOM, 3812c8c8f3953921ef18544110dafc3505c1ac62

improve memcg oom killer robustness (提升memcg oom killer的健壮性)

- v1
  - patch set: https://lkml.org/lkml/2013/7/25/653 ,
  - lwn: https://lwn.net/Articles/560868/
- v2
  - patch set: https://lore.kernel.org/lkml/1375549200-19110-1-git-send-email-hannes@cmpxchg.org/ , https://lkml.org/lkml/2013/8/3/81 ,
  - lwn: https://lwn.net/Articles/562091/

第一版代码分析

在分配内存失败的情况下，memcg代码会导致task trap，直到解决OOM情况为止。此时，它们可以持有各种锁（fs，mm），这容易导致死锁。

此系列patch将memcg OOM处理转换为在charge上下文中启动的两步过程，但是在完全解开错误堆栈后将进行任何等待。

1-4为支持新的memcg要求的体系结构处理程序做准备，但是这样做还可以消除旧的残废并统一整个体系结构的内存不足行为。

补丁5禁用了针对系统调用，预读，内核故障的memcg OOM处理，因为它们可以使用-ENOMEM正常展开堆栈。 OOM处理仅限于没有其他选择的用户触发的故障。

补丁6实现了由两部分组成的OOM处理，以使任务永远不会在OOM情况下被充满电荷的堆栈所困。