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. memcg priority oom killer

https://github.com/alibaba/cloud-kernel/commit/52e375fcb7a71d62566dc89764ce107e2f6af9ee#diff-8fa1dddd53606ceb933c5c6a12e714ed41e11d37a2b7bc48e91d15b54171d033

在内存压力下，将发生回收和oom. 在一个有多个cgroup的系统中，当有其他候选时，我们可能需要这些cgroup的一些内存或任务在回收和oom中幸存下来。

@memory.low 和 @memory.min已在回收期间发生这种情况，此补丁引入了memcg优先级oom来满足oom中的上述要求。

优先级是从0到12，数字越高优先级越高。 当oom发生时，它总是从低优先级的memcg中选择受害者。它既适用于memcg oom，也适用于全局oom，可以通过 @memory.use_priority_oom 启用/禁用，对于通过**根memcg**的 @memory.use_priority_oom 进行的全局缩放，默认情况下处于禁用状态。

每个mem_cgroup结构体引入了几个和memcg priority的变量

```
@@ -252,6 +255,12 @@ struct mem_cgroup {
        bool            oom_lock;
        int             under_oom;

        /* memcg priority */
        bool use_priority_oom;
        int priority;
        int num_oom_skip;
        struct mem_cgroup *next_reset;

        int     swappiness;
```

原有逻辑也是调用kernel 的 out_of_memory() ，然后调用 select_bad_process 和 oom_kill_process

在原有逻辑中， select_bad_process 阶段，如果是memcg，进行调用memcg自己的函数 mem_cgroup_scan_tasks

新方案，如果oom_control是memcg或者 root_memcg_use_priority_oom() root_memcg 使用priority_oom，则调用自己实现的 mem_cgroup_select_bad_process(oc);

> 注：所以可能在内存分配上下文(即非memcg的charge阶段)，可能也会调用到memcg的select bad process;
> 而在select中，如果是内存page分配上下文(oc->memcg为空)，
> 则 memcg = root_mem_cgroup ;

如果memcg(可能是当前memcg <在charge上下文> 或root_memcg)使用了 priority_oom ，先调用 mem_cgroup_select_victim_cgroup() 选择一个受害者memcgroup，然后调用之前的 mem_cgroup_scan_tasks 从这个受害者memcgroup中扫描进程(以前方案只有在memcg charge上下文会发生，所以只会当前memcg的扫描task)

> 注：
> 新方案只要开启root_memcg的priority_oom都会调用mem_cgroup的scan_tasks方法？是否合理
> 如果当前memcg没有开启priority_oom，则也不会根据priority选择mem_cgroup

> task_struct->css_set->cgroup_subsys_state->cgroup

在 mem_cgroup_select_victim_cgroup() 中，

1. 如果这个memcg没有hierarchy，则返回当前memcg
2. 获得memcg的subsystem(parent)
3. 获得parent css的memcg(parent_memcg)
4. while(parent)

- 如果parent的task数目小于等于 其对应的memcg不可kill的task数目(num_oom_skip)，跳出循环
- 受害者等于parent
- chosen_priority = 12 + 1 (最高优先级+1)
- 遍历parent subsystem的children(子链表串)css
  - 如果子css的task数目小于等于 其对应的memcg不可kill的task数目(num_oom_skip)，下一个子css
  - 子css的memcg的priority大于chosen_priority，下一个子css
  - 子css的memcg的priority小于chosen_priority，子css优先级更低，遍历子css的子css