Cgroup - Linux内存资源管理

Hi，我是Zorro。这是我的微博地址，我会不定期在这里更新文章，如果你有兴趣，可以来关注我呦。

另外，我的其他联系方式：

Email: mini.jerry@gmail.com

QQ: 30007147

在聊cgroup的内存限制之前，我们有必要先来讲解一下：

Linux内存管理基础知识

free命令

无论从任何角度看，Linux的内存管理都是一坨麻烦的事情，当然我们也可以用一堆、一片、一块、一筐来形容这个事情，但是毫无疑问，用一坨来形容它简直恰当无比。在理解它之前，我甚至不会相信精妙的和恶心可以同时形容同一件事情，是的，在我看来它就是这样的。其实我只是做个铺垫，让大家明白，我们下面要讲的内容，绝不是一个成体系的知识，所以，学习起来也确实很麻烦。甚至，我写这个技术文章之前一度考虑了很久该怎么写？从哪里开始写？思考了半天，还是不能免俗，我们无奈，仍然先从free命令说起：

```
[root@zorrozou-pc ~]# free
             total       used       free     shared    buffers     cached
Mem:     131904480    6681612  125222868          0     478428    4965180
-/+ buffers/cache:    1238004  130666476
Swap:      2088956          0    2088956
```

这个命令几乎是每一个使用过Linux的人必会的命令，但越是这样的命令，似乎真正明白的人越少（我是说比例越少）。一般情况下，对此命令的理解可以分这几个阶段：

1. 我擦，内存用了好多，6个多G，可是我什么都没有运行啊？为什么会这样？Linux好占内存。

2. 嗯，根据我专业的眼光看出来，内存才用了1G多点，还有很多剩余内存可用。buffers/cache占用的较多，说明系统中有进程曾经读写过文件，但是不要紧，这部分内存是当空闲来用的。
3. free显示的是这样，好吧我知道了。神马？你问我这些内存够不够，我当然不知道啦！我特么怎么知道你程序怎么写的？

如果你的认识在第一种阶段，那么请你继续补充关于Linux的buffers／cache的知识。如果你处在第二阶段，好吧，你已经是个老手了，但是需要提醒的是，上帝给你关上一扇门的同时，肯定都会给你放一条狗的。是的，Linux的策略是：内存是用来用的，而不是用来看的。但是，只要是用了，就不是没有成本的。有什么成本，凭你对buffer/cache的理解，应该可以想的出来。一般我比较认同第三种情况，一般光凭一个free命令的显示，是无法判断出任何有价值的信息的，我们需要结合业务的场景以及其他输出综合判断目前遇到的问题。当然也可能这种人给人的第一感觉是他很外行，或者他真的是外行。

无论如何，free命令确实给我门透露了一些有用的信息，比如内存总量，剩余多少，多少用在了buffers／cache上，Swap用了多少，如果你用了其它参数还能看到一些其它内容，这里不做一一列举。那么这里又引申出另一些概念，什么是buffer？什么是cache？什么是swap？由此我们就直接引出另一个命令：

```
[root@zorrozou-pc ~]# cat /proc/meminfo
MemTotal:       131904480 kB
MemFree:        125226660 kB
Buffers:          478504 kB
Cached:          4966796 kB
SwapCached:            0 kB
Active:          1774428 kB
Inactive:        3770380 kB
Active(anon):     116500 kB
Inactive(anon):     3404 kB
Active(file):    1657928 kB
Inactive(file):  3766976 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       2088956 kB
SwapFree:        2088956 kB
Dirty:               336 kB
Writeback:             0 kB
AnonPages:         99504 kB
Mapped:            20760 kB
Shmem:             20604 kB
Slab:             301292 kB
SReclaimable:     229852 kB
SUnreclaim:        71440 kB
KernelStack:        3272 kB
PageTables:         3320 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    68041196 kB
Committed_AS:     352412 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      493196 kB
VmallocChunk:   34291062284 kB
HardwareCorrupted:     0 kB
AnonHugePages:     49152 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:      194816 kB
DirectMap2M:     3872768 kB
DirectMap1G:    132120576 kB
```

以上显示的内容都是些什么鬼？

其实这个问题的答案也是另一个问题的答案，即：Linux是如何使用内存的？了解清楚这个问题是很有必要的，因为只有先知道了Linux如何使用内存，我们在能知道内存可以如何限制，

以及，做了限制之后会有什么问题？我们在此先例举出几个常用概念的意义：

内存，作为一种相对比较有限的资源，内核在考虑其管理时，无非应该主要从以下出发点考虑：

1. 内存够用时怎么办？
2. 内存不够用时怎么办？

在内存够用时，内核的思路是，如何尽量提高资源的利用效率，以加快系统整体响应速度和吞吐量？于是内存作为一个CPU和I／O之间的大buffer的功能就呼之欲出了。为此，内核设计了以下系统来做这个功能：

Buffers／Cached

buffer和cache是两个在计算机技术中被用滥的名词，放在不通语境下会有不同的意义。在内存管理中，我们需要特别澄清一下，这里的buffer指Linux内存的：Buffer cache。这里的cache指Linux内存中的：Page cache。翻译成中文可以叫做缓冲区缓存和页面缓存。在历史上，它们一个（buffer）被用来当成对io设备写的缓存，而另一个（cache）被用来当作对io设备的读缓存，这里的io设备，主要指的是块设备文件和文件系统上的普通文件。但是现在，它们的意义已经不一样了。在当前的内核中，page cache顾名思义就是针对内存页的缓存，说白了就是，如果有内存是以page进行分配管理的，都可以使用page cache作为其缓存来使用。当然，不是所有的内存都是以页（page）进行管理的，也有很多是针对块（block）进行管理的，这部分内存使用如果要用到cache功能，则都集中到buffer cache中来使用。（从这个角度出发，是不是buffer cache改名叫做block cache更好？）然而，也不是所有块（block）都有固定长度，系统上块的长度主要是根据所使用的块设备决定的，而页长度在X86上无论是32位还是64位都是4k。

而明白了这两套缓存系统的区别，也就基本可以理解它们究竟都可以用来做什么了。

什么是page cache

Page cache主要用来作为文件系统上的文件数据的缓存来用，尤其是针对当进程对文件有read／write操作的时候。如果你仔细想想的话，作为可以映射文件到内存的系统调用：mmap是不是很自然的也应该用到page cache？如果你再仔细想想的话，malloc会不会用到page cache？

以上提出的问题都请自己思考，本文档不会给出标准答案。

在当前的实现里，page cache也被作为其它文件类型的缓存设备来用，所以事实上page cache也负责了大部分的块设备文件的缓存工作。

什么是buffer cache

Buffer cache则主要是设计用来在系统对块设备进行读写的时候，对块进行数据缓存的系统来使用。但是由于page cache也负责块设备文件读写的缓存工作，于是，当前的buffer cache实际上要负责的工作比较少。这意味着某些对块的操作会使用buffer cache进行缓存，比如我们在格式化文件系统的时候。

一般情况下两个缓存系统是一起配合使用的，比如当我们对一个文件进行写操作的时候，page cache的内容会被改变，而buffer cache则可以用来将page标记为不同的缓冲区，并记录是哪一个缓冲区被修改了。这样，内核在后续执行脏数据的回写（writeback）时，就不用将整个

page写回，而只需要写回修改的部分即可。

有搞大型系统经验的人都知道，缓存就像万金油，只要哪里有速度差异产生的瓶颈，就可以在哪里抹。但是其成本之一就是，需要维护数据的一致性。内存缓存也不例外，内核需要维持其一致性，在脏数据产生较快或数据量较大的时候，缓存系统整体的效率一样会下降，因为毕竟脏数据写回也是要消耗IO的。这个现象也会表现在这样一种情况下，就是当你发现free的时候，内存使用量较大，但是去掉了buffer／cache的使用之后剩余确很多。以一般的理解，都会认为此时进程如果申请内存，内核会将buffer／cache占用的内存当成空闲的内存分给进程，这是没错的。但是其成本是，在分配这部分已经被buffer／cache占用的内存的时候，内核会先对其上面的脏数据进行写回操作，保证数据一致后才会清空并分给进程使用。如果此时你的进程是突然申请大量内存，而且你的业务是一直在产生很多脏数据（比如日志），并且系统没有及时写回的时候，此时系统给进程分配内存的效率会很慢，系统IO也会很高。那么此时你还以为buffer／cache可以当空闲内存使用么？

> *思考题：Linux什么时候会将脏数据写回到外部设备上？这个过程如何进行人为干预？*

这足可以证明一点，以内存管理的复杂度，我们必须结合系统上的应用状态来评估系统监控命令所给出的数据，才是做评估的正确途径。如果你不这样做，那么你就可以轻而易举的得出“Linux系统好烂啊！“这样的结论。也许此时，其实是你在这个系统上跑的应用很烂的缘故导致的问题。

接下来，当内存不够用的时候怎么办？

我们好像已经分析了一种内存不够用的状态，就是上述的大量buffer／cache把内存几乎占满的情况。但是基于Linux对内存的使用原则，这不算是不够用，但是这种状态导致IO变高了。我们进一步思考，假设系统已经清理了足够多的buffer／cache分给了内存，而进程还在嚷嚷着要内存咋办？

此时内核就要启动一系列手段来让进程**尽量**在此时能够正常的运行下去。

请注意我在这说的是一种异常状态！我之所以要这样强调是因为，很多人把内存用满了当称一种正常状态。他们认为，当我的业务进程在内存使用到压力边界的情况下，系统仍然需要保证让业务进程有正常的状态！这种想法显然是缘木求鱼了。另外我还要强调一点，系统提供的是内存管理的机制和手段，而内存用的好不好，主要是业务进程的事情，责任不能本末倒置。

谁该SWAP?

首先是Swap机制。Swap是交换技术，这种技术是指，当内存不够用的时候，我们可以选择性的将一块磁盘、分区或者一个文件当成交换空间，将内存上一些临时用不到的数据放到交换空间上，以释放内存资源给急用的进程。

哪些数据可能会被交换出去呢？从概念上判断，如果一段内存中的数据被经常访问，那么就不应该被交换到外部设备上，因为这样的数据如果交换出去的话会导致系统响应速度严重下降。内存管理需要将内存区分为活跃的（Active）和不活跃的（Inactive），再加上一个进程使用的用户空间内存映射包括文件影射（file）和匿名影射（anon），所以就包括了

Active（anon）、Inactive（anon）、Active（file）和Inactive（file）。你说神马？啥是文件影射（file）和匿名影射（anon）？好吧，我们可以这样简单的理解，匿名影射主要是诸如进程使用malloc和mmap的MAP_ANONYMOUS的方式申请的内存，而文件影射就是使用mmap影射的文件系统上的文件，这种文件系统上的文件既包括普通的文件，也包括临时文件系统（tmpfs）。这意味着，Sys V的IPC和POSIX的IPC（IPC是进程间通信机制，在这里主要指共享内存，信号量数组和消息队列）都是通过文件影射方式体现在用户空间内存中的。这两种影射的内存都会被算成进程的RSS，但是也一样会被显示在cache的内存计数中，在相关cgroup的另一项统计中，共享内存的使用和文件缓存（file cache）也都会被算成是cgroup中的cache使用的总量。这个统计显示的方法是：

```
[root@zorrozou-pc ~]# cat /cgroup/memory/memory.stat
cache 94429184
rss 102973440
rss_huge 50331648
mapped_file 21512192
swap 0
pgpgin 656572990
pgpgout 663474908
pgfault 2871515381
pgmajfault 1187
inactive_anon 3497984
active_anon 120524800
inactive_file 39059456
active_file 34484224
unevictable 0
hierarchical_memory_limit 9223372036854775807
hierarchical_memsw_limit 9223372036854775807
total_cache 94429184
total_rss 102969344
total_rss_huge 50331648
total_mapped_file 21520384
total_swap 0
total_pgpgin 656572990
total_pgpgout 663474908
total_pgfault 2871515388
total_pgmajfault 1187
total_inactive_anon 3497984
total_active_anon 120524800
total_inactive_file 39059456
total_active_file 34484224
total_unevictable 0
```

好吧，说了这么半天终于联系到一个cgroup的内存限制相关的文件了。在这需要说明的是，你之所以看见我废话这么多，是因为我们必须先基本理清楚Linux系统的内存管理方式，才能进一步对cgroup中的内存限制做规划使用，否则同样的名词会有很多的歧义。就比如我们在观察某一个cgroup中的cache占用数据的时候，我们究竟该怎么理解它？真的把它当成空闲空间来看么？

我们撒的有点远，回过头来说说这些跟Swap有什么关系？还是刚才的问题，什么内容该被从内存中交换出去呢？文件cache是一定不需要的，因为既然是cache，就意味着它本身就是硬盘上的文件（当然你现在应该知道了，它也不仅仅只有文件），那么如果是硬盘上的文件，就不用swap交换出去，只要写回脏数据，保持数据一致之后清除就可以了，这就是刚才说过的缓存清楚机制。但是我们同时也要知道，并不是所有被标记为cache的空间都能被写回硬盘的(是的，比如共享内存)。那么能交换出去内存应该主要包括有Inactive（anon）这部分内存。主要注意的是，内核也将共享内存作为计数统计近了Inactive（anon）中去了（是的，共享内存也可以被Swap）。还要补充一点，如果内存被mlock标记加锁了，则也不会交换，这是对内存加mlock锁的唯一作用。刚才我们讨论的这些计数，很可能会随着Linux内核的版本改变而产生变化，但是在比较长的一段时间内，我们可以这样理解。

我们基本搞清了swap这个机制的作用效果，那么既然swap是内部设备和外部设备的数据拷贝，那么加一个缓存就显得很有必要，这个缓存就是swapcache，在memory.stat文件中，swapcache是跟anon page被一起记录到rss中的，但是并不包含共享内存。另外再说明一下，HugePages也是不会交换的。显然，当前的swap空间用了多少，总共多少，这些我们也可以在相关的数据中找到答案。

> *以上概念中还有一些名词大家可能并不清楚其含义，比如RSS或HugePages。请自行查资料补上这些知识。为了让大家真的理解什么是RSS，请思考ps aux命令中显示的VSZ，RSS和cat /proc/**pid**/smaps中显示的：PSS这三个进程占用内存指标的差别?*

何时SWAP?

搞清楚了谁该swap，那么还要知道什么时候该swap。这看起来比较简单，内存耗尽而且cache也没什么可以回收的时候就应该触发swap。其实现实情况也没这么简单，实际上系统在内存压力可能不大的情况下也会swap，这种情况并不是我们今天要讨论的范围。

> *思考题：除了内存被耗尽的时候要swap，还有什么时候会swap？如何调整内核swap的行为？如何查看当前系统的swap空间有哪些？都是什么类型？什么是swap权重？swap权重有什么意义？*

其实绝大多数场景下，什么时候swap并不重要，而swap之后的事情相对却更重要。大多数的内存不够用，只是临时不够用，比如并发突增等突发情况，这种情况的特点是时间持续短，此时swap机制作为一种临时的中转措施，可以起到对业务进程的保护作用。因为如果没有swap，内存耗尽的结果一般都是触发oom killer，会杀掉此时积分比较高的进程。如果更严重的话，内存不够用还会触发进程D状态死锁，这一般发生在多个进程同时要申请内存的时候，此时oom killer机制也可能会失效，因为需要被干掉的积分比较高的进程很可能就是需要申请内存的进程，而这个进程本身因为正在争抢内存而导致陷入D状态，那么此时kill就可能是对它无效的。

但是swap也不是任何时候都有很好的保护效果。如果内存申请是长期并大量的，那么交换出去的数据就会因为长时间驻留在外部设备上，导致进程调用这段内存的几率大大增加，当进程很频繁的使用它已经被交换出去的内存时，就会让整个系统处在io繁忙的状态，此时进程的响应速度会严重下降，导致整个系统夯死。对于系统管理员来说，这种情况是完全不能接受的，因为故障之后的第一要务是赶紧恢复服务，但是swap频繁使用的IO繁忙状态会导致系统除了断电重启之外，没有其它可靠手段可以让系统从这种状态中恢复回来，所以这种情况是要尽力避免的。此时，如果有必要，我们甚至可以考虑不用swap，哪怕内存过量使用被oom，或者进程D状态都是比swap导致系统卡死的情况更好处理的状态。如果你的环境需求是这样的，那么可以考虑关闭swap。

进程申请内存的时候究竟会发生什么？

刚才我们从系统宏观的角度简要说明了一下什么是buffer／cache以及swap。下面我们从一个更加微观的角度来把一个内存申请的过程以及相关机制什么时候触发给串联起来。本文描述的过程是基于Linux 3.10内核版本的，Linux 4.1基本过程变化不大。如果你想确认在你的系统上究竟是什么样子，请自行翻阅相关内核代码。

进程申请内存可能用到很多种方法，最常见的就是malloc和mmap。但是这对于我们并不重要，因为无论是malloc还是mmap，或是其他的申请内存的方法，都不会真正的让内核去给进程分配一个实际的物理内存空间。真正会触发分配物理内存的行为是**缺页异常**。

缺页异常就是我们可以在memory.stat中看到的total_pgfault，这种异常一般分两种，一种叫major fault，另一种叫minor fault。这两种异常的主要区别是，进程所请求的内存数据是否会引发磁盘io？如果会引发，就是一个majfault，如果不引发，那就是minfault。就是说如果产生了major fault，这个数据基本上就意味着已经被交换到了swap空间上。

缺页异常的处理过程大概可以整理为以下几个路径：

首先检查要访问的虚拟地址是否合法，如果合法则继续查找和分配一个物理页，步骤如下：

1. 检查发生异常的虚拟地址是不是在物理页表中不存在？如果是，并且是匿名影射，则申请置0的匿名影射内存，此时也有可能是影射了某种虚拟文件系统，比如共享内存，那么就去影射相关的内存区，或者发生COW写时复制申请新内存。如果是文件影射，则有两种可能，一种是这个影射区是一个page cache，直接将相关page cache区影射过来即可，或者COW新内存存放需要影射的文件内容。如果page cache中不存在，则说明这个区域已经被交换到swap空间上，应该去处理swap。
2. 如果页表中已经存在需要影射的内存，则检查是否要对内存进行写操作，如果不写，那就直接复用，如果要写，就发生COW写时复制，此时的COW跟上面的处理过程不完全相同，在内核中，这里主要是通过do_wp_page方法实现的。

如果需要申请新内存，则都会通过alloc_page_vma申请新内存，而这个函数的核心方法是__alloc_pages_nodemask，也就是Linux内核著名的内存管理系统**伙伴系统**的实现。

分配过程先会检查空闲页表中有没有页可以申请，实现方法是：get_page_from_freelist，我们并不关心正常情况，分到了当然一切ok。更重要的是异常处理，如果空闲中没有，则会进入__alloc_pages_slowpath方法进行处理。这个处理过程的主逻辑大概这样：

1. 唤醒kswapd进程，把能换出的内存换出，让系统有内存可用。
2. 继续检查看看空闲中是否有内存。有了就ok，没有继续下一步：

3. 尝试清理page cache，清理的时候会将进程置为D状态。如果还申请不到内存则：
4. 启动oom killer干掉一些进程释放内存，如果这样还不行则：
5. 回到步骤1再来一次！

当然以上逻辑要符合一些条件，但是这一般都是系统默认的状态，比如，你必须启用oom killer机制等。另外这个逻辑中有很多其它状态与本文无关，比如检查内存水印、检查是否是高优先级内存申请等等，当然还有关于numa节点状态的判断处理，我没有一一列出。另外，以上逻辑中，不仅仅只有清理cache的时候会使进程进入D状态，还有其它逻辑也会这样做。这就是为什么在内存不够用的情况下，oom killer有时也不生效，因为可能要干掉的进程正好陷入这个逻辑中的D状态了。

以上就是内存申请中，大概会发生什么的过程。当然，我们这次主要是真对本文的重点cgroup内存限制进行说明，当我们处理限制的时候，更多需要关心的是当内存超限了会发生什么？对边界条件的处理才是我们这次的主题，所以我并没有对正常申请到的情况做细节说明，也没有对用户态使用malloc什么时候使用sbrk还是mmap来申请内存做出细节说明，毕竟那是程序正常状态的时候的事情，后续可以另写一个内存优化的文章主要讲解那部分。

下面我们该进入正题了：

Cgroup内存限制的配置

当限制内存时，我们最好先想清楚如果内存超限了会发生什么？该怎么处理？业务是否可以接受这样的状态？这就是为什么我们在讲如何限制之前说了这么多基础知识的“废话”。其实最简单的莫过于如何进行限制了，我们的系统环境还是沿用上一次讲解CPU内存隔离的环境，使用cgconfig和cgred服务进行cgroup的配置管理。还是创建一个zorro用户，对这个用户产生的进程进行内存限制。基础配置方法不再多说，如果不知道的请参考这个文档。

环境配置好之后，我们就可以来检查相关文件了。内存限制的相关目录根据cgconfig.config的配置放在了/cgroup/memory目录中，如果你跟我做了一样的配置，那么这个目录下的内容应该是这样的：

```
[root@zorrozou-pc ~]# ls /cgroup/memory/
cgroup.clone_children  memory.failcnt                  memory.kmem.slabinfo
memory.kmem.usage_in_bytes  memory.memsw.limit_in_bytes
memory.oom_control          memory.usage_in_bytes  shrek
cgroup.event_control   memory.force_empty
memory.kmem.tcp.failcnt             memory.limit_in_bytes
memory.memsw.max_usage_in_bytes  memory.pressure_level
memory.use_hierarchy   tasks
cgroup.procs           memory.kmem.failcnt
memory.kmem.tcp.limit_in_bytes      memory.max_usage_in_bytes
memory.memsw.usage_in_bytes      memory.soft_limit_in_bytes           zorro
cgroup.sane_behavior   memory.kmem.limit_in_bytes
memory.kmem.tcp.max_usage_in_bytes  memory.meminfo
memory.move_charge_at_immigrate  memory.stat
notify_on_release
jerry                  memory.kmem.max_usage_in_bytes
memory.kmem.tcp.usage_in_bytes      memory.memsw.failcnt
memory.numa_stat                memory.swappiness           release_agent
```

其中，zorro、jerry、shrek都是目录概念跟cpu隔离的目录树结构类似。相关配置文件内容：

```
[root@zorrozou-pc ~]# cat /etc/cgconfig.conf    mount {
    cpu = /cgroup/cpu;
    cpuset  = /cgroup/cpuset;
    cpuacct = /cgroup/cpuacct;
    memory  = /cgroup/memory;
    devices = /cgroup/devices;
    freezer = /cgroup/freezer;
    net_cls = /cgroup/net_cls;
    blkio   = /cgroup/blkio;
}

group zorro {
    cpu {
        cpu.shares = 6000;
#       cpu.cfs_quota_us = "600000";
    }
    cpuset {
#       cpuset.cpus = "0-7,12-19";
#       cpuset.mems = "0-1";
    }
    memory {
    }
}
```

配置中添加了一个真对memory的空配置项，我们稍等下再给里面添加配置。

```
[root@zorrozou-pc ~]# cat /etc/cgrules.conf
zorro       cpu,cpuset,cpuacct,memory   zorro
jerry       cpu,cpuset,cpuacct,memory   jerry
shrek       cpu,cpuset,cpuacct,memory   shrek
```

文件修改完之后记得重启相关服务：

```
[root@zorrozou-pc ~]# service cgconfig restart
[root@zorrozou-pc ~]# service cgred restart
```

让我们继续来看看真对内存都有哪些配置参数：

```
[root@zorrozou-pc ~]# ls /cgroup/memory/zorro/
cgroup.clone_children  memory.kmem.failcnt
memory.kmem.tcp.limit_in_bytes      memory.max_usage_in_bytes
memory.memsw.usage_in_bytes      memory.soft_limit_in_bytes
cgroup.event_control   memory.kmem.limit_in_bytes
memory.kmem.tcp.max_usage_in_bytes  memory.meminfo
memory.move_charge_at_immigrate  memory.stat
notify_on_release
cgroup.procs           memory.kmem.max_usage_in_bytes
memory.kmem.tcp.usage_in_bytes      memory.memsw.failcnt
memory.numa_stat                memory.swappiness           tasks
memory.failcnt         memory.kmem.slabinfo
memory.kmem.usage_in_bytes          memory.memsw.limit_in_bytes
memory.oom_control              memory.usage_in_bytes
memory.force_empty     memory.kmem.tcp.failcnt         memory.limit_in_bytes
memory.memsw.max_usage_in_bytes  memory.pressure_level
memory.use_hierarchy
```

首先我们已经认识了memory.stat文件了，这个文件内容不能修改，它实际上是输出当前cgroup相关内存使用信息的。常见的数据及其含义我们刚才也已经说过了，在此不再复述。

cgroup内存限制

memory.memsw.limit_in_bytes:内存＋swap空间使用的总量限制。

memory.limit_in_bytes：内存使用量限制。

这两项的意义很清楚了，如果你决定在你的cgroup中关闭swap功能，可以把两个文件的内容设置为同样的值即可。至于为什么相信大家都能想清楚。

OOM控制

memory.oom_control:内存超限之后的oom行为控制。 这个文件中有两个值:

oom_kill_disable 0

默认为0表示打开oom killer，就是说当内存超限时会触发干掉进程。如果设置为1表示关闭oom killer，此时内存超限不会触发内核杀掉进程。而是将进程夯住（hang／sleep），实际上内核中就是将进程设置为D状态，并且将相关进程放到一个叫做OOM-waitqueue的队列中。这时的进程可以kill杀掉。如果你想继续让这些进程执行，可以选择这样几个方法:

1. 增加内存，让进程有内存可以继续申请。
2. 杀掉一些进程，让本组内有内存可用。
3. 把一些进程移到别的cgroup中，让本cgroup内有内存可用。
4. 删除一些tmpfs的文件，就是占用内存的文件，比如共享内存或者其它会占用内存的文件。

说白了就是，此时只有当cgroup中有更多内存可以用了，在OOM-waitqueue队列中被挂起的进程就可以继续运行了。

under_oom 0

这个值只是用来看的，它表示当前的cgroup的状态是不是已经oom了，如果是，这个值将显示为1。我们就是通过设置和监测这个文件中的这两个值来管理cgroup内存超限之后的行为的。在默认场景下，如果你使用了swap，那么你的cgroup限制内存之后最常见的异常效果是IO变高，如果业务不能接受，我们一般的做法是关闭swap，那么cgroup内存oom之后都会触发kill掉进程，如果我们用的是LXC或者Docker这样的容器，那么还可能干掉整个容器。当然也经常会因为kill进程的时候因为进程处在D状态，而导致整个Docker或者LXC容器根本无法被杀掉。至于原因，在前面已经说的很清楚了。当我们遇到这样的困境时该怎么办？一个好的办法是，关闭oom killer，让内存超限之后，进程挂起，毕竟这样的方式相对可控。此时我们可以检查under_oom的值，去看容器是否处在超限状态，然后根据业务的特点决定如何处理业务。我推荐的方法是关闭部分进程或者重启掉整个容器，因为可以想像，容器技术所承载的服务应该是在整体软件架构上有容错的业务，典型的场景是web服务。容器技术的特点就是生存周期短，在这样的场景下，杀掉几个进程或者几个容器，都应该对整体服务的稳定性影响不大，而且容器的启动速度是很快的，实际上我们应该认为，容器的启动速度应该是跟进程启动速度可以相媲美的。你的业务会因为死掉几个进程而表现不稳定么？如果不会，请放心的干掉它们吧，大不了很快再启动起来就是了。但是如果你的业务不是这样，那么请根据自己的情况来制定后续处理的策略。

当我们进行了内存限制之后，内存超限的发生频率要比使用实体机更多了，因为限制的内存量一般都是小于实际物理内存的。所以，使用基于内存限制的容器技术的服务应该多考虑自己内存使用的情况，尤其是内存超限之后的业务异常处理应该如何让服务受影响的程度降到更低。在系统层次和应用层次一起努力，才能使内存隔离的效果达到最好。

内存资源审计

memory.memsw.usage_in_bytes:当前cgroup的内存＋swap的使用量。

memory.usage_in_bytes:当前cgroup的内存使用量。

memory.max_usage_in_bytes:cgroup最大的内存＋swap的使用量。

memory.memsw.max_usage_in_bytes:cgroup的最大内存使用量。

这些文件都是只读的，用来查看相关状态信息，只能看不能改。

如果你的内核配置打开了CONFIG_MEMCG_KMEM选项的话，那么可以看到当前cgroup的内核内存使用的限制和状态统计信息，他们都是以memory.kmem开头的文件。你可以通过memory.kmem.limit_in_bytes来限制内核使用的内存大小，通过memory.kmem.slabinfo来查看内核slab分配器的状态。现在还能通过memory.kmem.tcp开头的文件来限制cgroup中使用tcp协议的内存资源使用和状态查看。

所有名字中有failcnt的文件里面的值都是相关资源超限的次数的计数，可以通过echo 0将这些计数重置。如果你的服务器是NUMA架构的话，可以通过memory.numa_stat这个文件来查看cgroup中的NUMA相关状态。memory.swappiness跟/proc/sys/vm/swappiness的概念一致，用来调整cgroup使用swap的状态，如果大家认真做了本文前面的思考题的话，应该知道这个文件是干嘛的，本文不会详细解释关于swappiness的细节算法，以后将在性能调整系列文章中详细解释相关参数。

内存软限制以及内存超卖

memory.soft_limit_in_bytes:内存软限制。

如果超过了memory.limit_in_bytes所定义的限制，那么进程会被oom killer干掉或者被暂停，这相当于硬限制，因为进程无法申请超过自身cgroup限制的内存，但是软限制确是可以突破的。我们假定一个场景，如果你的实体机上有四个cgroup，实体机的内存总量是64G，那么一般情况我们会考虑给每个cgroup限制到16G内存。但是现实情况并不会这么理想，首先实体机上其他进程和内核会占用部分内存，这将导致实际上每个cgroup都不会真的有16G内存可用，如果四个cgroup都尽量占用内存的话，他们可能谁都不会到达内存的上限触发超限的行为，这可能将导致进程都抢不到内存而被饿死。类似的情况还可能发上在内存超卖的环境中，比如，我们仍然只有64G内存，但是确开了8个cgroup，每个都限制了16G内存。这样每个cgroup分配的内存之和达到了128G，但是实际内存量只有64G。这种情况是出于绝大多数应用可能不会占用满所有的内存来考虑的，这样就可以把本来属于它的那份内存“借用”给其它cgroup。以上这样的情况都会出现类似的问题，就是，如果全局内存已经耗尽了，但是某些cgroup还没达到他的内存使用上限，而它们此时如果要申请内存的话，此时该从哪里回收内存？如果我们配置了memory.soft_limit_in_bytes，那么内核将去回收那些内存超过了这个软限制的cgroup的内存，尽量缩减它们的内存占用达到软限制的量以下，以便让没有达到软限制的cgroup有内存可以用。当然，在没有这样的内存竞争以及没有达到硬限制的情况下，软限制是不会生效的。还有就是，软限制的起作用时间可能会比较长，毕竟内核要平衡多个cgroup的内存使用。

根据软限制的这些特点，我们应该明白如果想要软限制生效，应该把它的值设置成小于硬限制。

进程迁移时的内存charge

memory.move_charge_at_immigrate:打开或者关闭进程迁移时的内存记账信息。

进程可以在多个cgroup之间切换，所以内存限制必须考虑当发生这样的切换时，进程进入的新cgroup中记录的内存使用量是重新从0累计还是把原来cgroup中的信息迁移过来？当这个开关设置为0的时候是关闭这个功能，相当于不累计之前的信息，默认是1，迁移的时候要在新的cgroup中累积（charge）原来信息，并把旧group中的信息给uncharge掉。如果新cgroup中没有足够的空间容纳新来的进程，首先内核会在cgroup内部回收内存，如果还是不够，就会迁移失败。

内存压力通知机制

最后，内存的资源隔离还提供了一种压力通知机制。当cgoup内的内存使用量达到某种压力状态的时候，内核可以通过eventfd的机制来通知用户程序，这个通知是通过**cgroup.event_control**和**memory.pressure_level**来实现的。使用方法是：

使用eventfd()创建一个eventfd，假设叫做efd，然后open()打开memory.pressure_level的文件路径，产生一个另一个fd，我们暂且叫它cfd，然后将这两个fd和我们要关注的内存压力级别告诉内核，让内核帮我们关注条件是否成立，通知方式就是把以上信息按这样的格式:"<event_fd：efd> "写入cgroup.event_control。然后就可以去等着efd是否可读了，如果能读出信息，则代表内存使用已经触发相关压力条件。

压力级别的level有三个：

“low”：表示内存使用已经达到触发内存回收的压力级别。

“medium”：表示内存使用压力更大了，已经开始触发swap以及将活跃的cache写回文件等操作了。

“critical”：到这个级别，就意味着内存已经达到上限，内核已经触发oom killer了。

程序从efd读出的消息内容就是这三个级别的关键字。我们可以通过这个机制，建立一个内存压力管理系统，在内存达到相应级别的时候，触发响应的管理策略，来达到各种自动化管理的目的。

下面给出一个监控程序的例子：

```
#include <assert.h>
#include <err.h>
#include <errno.h>
#include <fcntl.h>
#include <libgen.h>
#include <limits.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <sys/eventfd.h>

#define USAGE_STR "Usage: cgroup_event_listener <path-to-control-file>
<args>"

int main(int argc, char **argv)
{
    int efd = -1;
    int cfd = -1;
    int event_control = -1;
    char event_control_path[PATH_MAX];
    char line[LINE_MAX];
    int ret;

    if (argc != 3)
        errx(1, "%s", USAGE_STR);

    cfd = open(argv[1], O_RDONLY);
    if (cfd == -1)
        err(1, "Cannot open %s", argv[1]);

    ret = snprintf(event_control_path, PATH_MAX, "%s/cgroup.event_control",
            dirname(argv[1]));
    if (ret >= PATH_MAX)
        errx(1, "Path to cgroup.event_control is too long");

    event_control = open(event_control_path, O_WRONLY);
    if (event_control == -1)
```

```
        err(1, "Cannot open %s", event_control_path);

    efd = eventfd(0, 0);
    if (efd == -1)
        err(1, "eventfd() failed");

    ret = snprintf(line, LINE_MAX, "%d %d %s", efd, cfd, argv[2]);
    if (ret >= LINE_MAX)
        errx(1, "Arguments string is too long");

    ret = write(event_control, line, strlen(line) + 1);
    if (ret == -1)
        err(1, "Cannot write to cgroup.event_control");

    while (1) {
        uint64_t result;

        ret = read(efd, &result, sizeof(result));
        if (ret == -1) {
            if (errno == EINTR)
                continue;
            err(1, "Cannot read from eventfd");
        }
        assert(ret == sizeof(result));

        ret = access(event_control_path, W_OK);
        if ((ret == -1) && (errno == ENOENT)) {
            puts("The cgroup seems to have removed.");
            break;
        }

        if (ret == -1)
            err(1, "cgroup.event_control is not accessible any more");

        printf("%s %s: crossed\n", argv[1], argv[2]);
    }

    return 0;
}
```

最后

Linux的内存限制要说的就是这么多了，当我们限制了内存之后，相对于使用实体机，实际上对于应用来说可用内存更少了，所以业务会相对更经常地暴露在内存资源紧张的状态下。相对于虚拟机（kvm，xen），多个cgroup之间是共享内核的，我们可以从内存限制的角度思考一些关于“容器”技术相对于虚拟机和实体机的很多特点：

1. 内存更紧张，应用的内存泄漏会导致相对更严重的问题。
2. 容器的生存周期时间更短，如果实体机的开机运行时间是以年计算的，那么虚拟机则是

以月计算的，而容器应该跟进程的生存周期差不多，顶多以天为单位。所以，容器里面要跑的应用应该可以被经常重启。

3. 当有多个cgroup（容器）同时运行时，我们不能再以实体机或者虚拟机对资源的使用的理解来规划整体运营方式，我们需要更细节的理解什么是cache，什么是swap，什么是共享内存，它们会被统计到哪些资源计数中？在内核并不冲突的环境，这些资源都是独立给某一个业务使用的，在理解上即使不是很清晰，也不会造成歧义。但是在cgroup中，我们需要彻底理解这些细节，才能对遇到的情况进行预判，并规划不同的处理策略。

也许我们还可以从中得到更多的理解，大家一起来想喽？