CHINA POSTGRESQL ASSOCIATION

题目：PostgreSQL与Linux内存管理

刘智龙

CONTENTS

/01

VM内存管理

VM

linux内核virtual memory子系统， 虚拟内存的全局策略和行为控制 `ll /proc/sys/vm/*`

vm管理什么?
- 虚拟内存管理，比如地址映射，内存保护（只读、可执行权限等），共享内存管理等
- 物理内存管理，负责物理内存的分配，通过虚拟内存地址映射分配给进程使用（不负责直接管理物理内存）
- 内存回收和交换，比如内存回收机制、oom、swap等
- 页缓存PageCache管理，利用合并IO以减少IO操作，也包含写脏页机制。PG的double buffer第二层就是这个，所以pg是依赖VM的pagecache管理的

VM-内存压缩

内存压缩（Compaction）是Linux内核用于解决内存碎片化问题的机制，通过合并空闲物理页，提升大块内存页的分配。

参数名	功能定位	默认值/范围
compact_memory	手动触发全局内存压缩操作	写入1触发
compaction_proactiveness	控制主动压缩的触发频率	4.x才有的参数。0-100（默认20）
compact_unevictable_allowed	是否允许压缩不可回收页（如mlock锁定的内存）	4.x才有的参数。0（禁止）或1（允许）
defrag_mode	控制内存碎片整理的触发策略	4.x才有的参数。0-3，0表示关闭自动compaction特性，只能手动压缩；1表示defer开启被动压缩。3.10默认1
extfrag_threshold	大块内存不足时，触发压缩的阈值	0-1000（默认500）

VM-内存压缩

压缩有3种模式（跟内核版本是否支持相关）：

•被动压缩：extfrag_threshold是进程申请大块内存发现已经不足是否压缩，解决“已发生”的碎片问题。
•主动压缩：compaction_proactiveness是主动控制压缩的积极性，优化“未发生”但可能出现的碎片风险。
•手动压缩：compact_memory。

VM-内存压缩

extfrag_threshold是Linux内核中控制被动压缩的参数。当内核尝试分配高阶连续物理内存（如大页）失败时，会通过碎片化指数判断失败原因：
•-1：分配成功（满足水位线）
•0：因内存不足失败
•0-1：因碎片化失败

```
cat /sys/kernel/debug/extfrag/extfrag_index |grep Normal
Node 0, zone   Normal -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 0.995 0.998
```

比如extfrag_threshold=600，则当碎片化指数>600时触发压缩。extfrag_index还是挺有用的，可以协助buddy观察碎片问题。

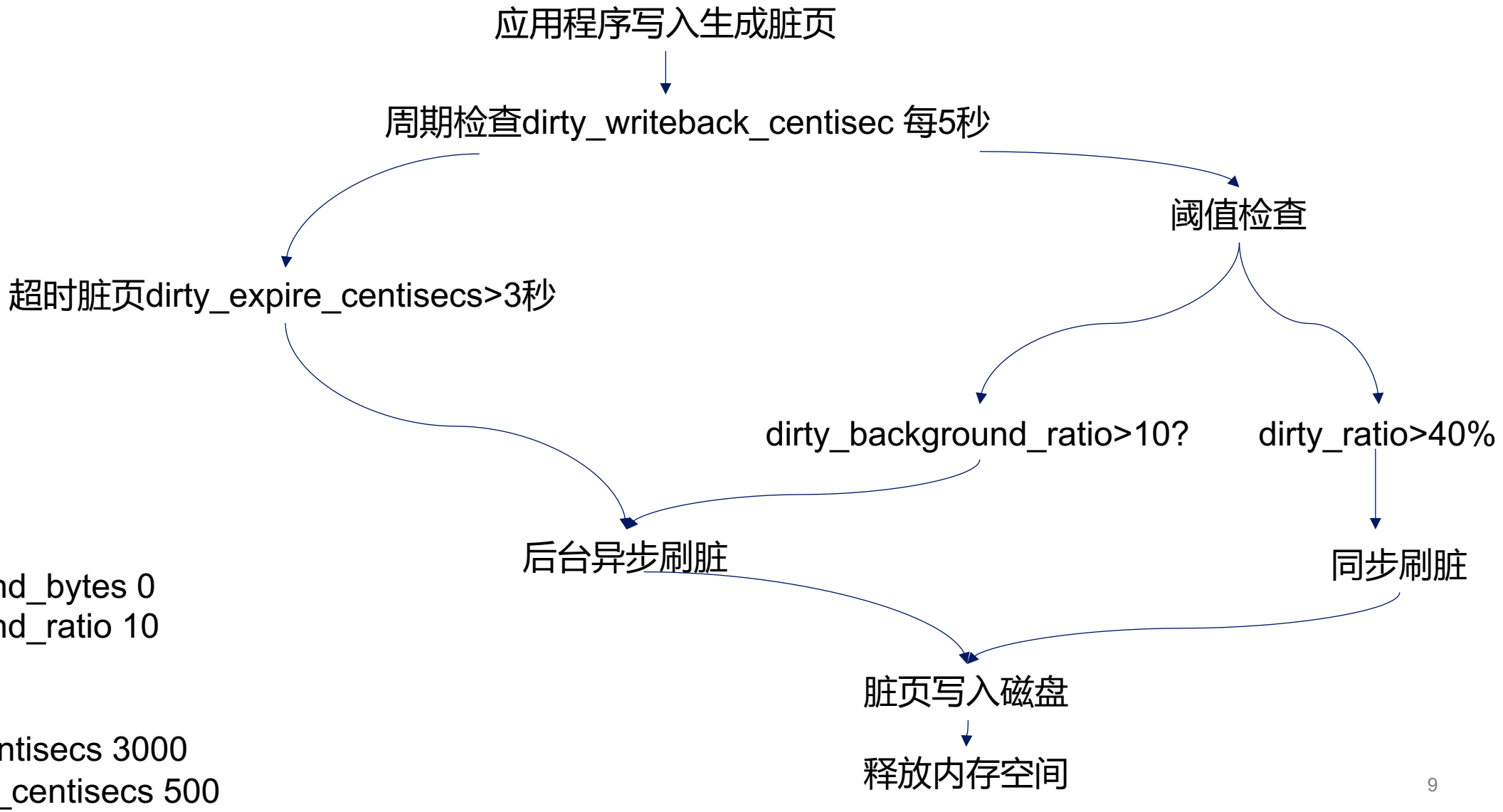
VM-刷脏

刷脏跟内存回收有点类似，也分异步和同步：
•异步刷脏：由pdflush/flush/kdmflush等后台线程执行，应用写入不受影响
•同步刷脏：直接阻塞应用进程，由发起写操作的进程自己刷脏

参数名称	作用描述	默认值
dirty_background_bytes	后台异步刷脏阈值，字节	0（未启用）
dirty_background_ratio	后台异步刷脏阈值，百分比	10%
dirty_bytes	同步刷脏阈值，字节	0（未启用）
dirty_ratio	同步刷脏阈值，百分比	20-40%
dirty_expire_centisecs	脏页在内存中的最长存活时间	3000（30秒）
dirty_writeback_centisecs	内核周期性检查脏页状态的频率	500（5秒）

xxx_bytes和xxx_ratio参数互斥。

VM-刷脏



dirty_background_bytes 0
dirty_background_ratio 10
dirty_bytes 0
dirty_ratio 40
dirty_expire_centisecs 3000
dirty_writeback_centisecs 500

VM-刷脏

观察脏页
cat /proc/meminfo |grep -E -w "Dirty" # os脏页
select isdirty,pinning_backends,count(*) from pg_buffercache where isdirty is true group by isdirty,pinning_backends; # pg脏页

测试用例
checkpoint;
begin;
--观察
insert into tlzl select generate_series(1,1000000);
--观察
commit;
--观察
checkpoint;
--观察

VM-刷脏

stage	dirty in pg	os dirty
干净状态	0	0.02-2M 浮动
insert完成	200M	上涨到1.7G，随后降落到20KB
commit提交	200M	0.02-2M 浮动
checkpoint刷脏	0	0.02-2M 浮动

pg dirty跟os dirty有点关系但又不全是相关的，pg插入数据时 os dirty确实会上升，不过os自己刷脏后，pg的脏页仍然是脏页。从原理和实践上看，共享内存中的脏页与os dirty无关。

VM-swapiness

swapiness参数代表什么?
swapiness参数控制系统从anonymous memory pool或 page cache 中回收内存的偏向性。

其实本质上是控制交换匿名页或者回收文件页，谁的代价对于上层应用来说更低。比如计算类的应用，动态分配或者私有内存使用更多，应该设置更低的swappiness，依赖数据的应用设置更高的swapiness以降低刷file page对数据访问的影响。不过这一切都建立在swap IO和filesystem IO的效率上。

一切都很美好，但是swap发生时很可能意味着性能降低。一般来说swapping意味着物理io，但file pages不一定，非脏页直接丢弃即可，不会产生io

VM-swapiness

swappiness=0时，内核会直到内存到达high水位线时，才有可能做swap。具体的策略跟内核版本和NUMA也有关系。
可以确认的是swappiness=0不代表关闭了swap，swapoff -a才是关闭swap功能。

swap动作是需要关注的

```
swapon --show
free -h |grep Swap
cat /proc/swaps
grep -E 'swap|none' /etc/fstab
cat /proc/meminfo|grep Swap
```

查看是否打开了swap

```
cat /proc/vmstat|grep swp
sar -W 1
```

监控swapping是否发生

VM-swapiness

inconsistent swap behavior：
os层的 /proc/sys/vm/swappiness 对 cgroups v1 的swap特性几乎没有影响（has little-to-no effect on the swap）。此问题可能导致不一致的swap行为。

发生场景（且）：
•vm.swappiness != cgroups memory.swappiness
•cgroups v1

发生原因：
systemd在启动初期创建cgroups，早于sysctl.service加载/etc/sysctl.conf ，vm.swappiness 无法限制 cgroup memory.swappiness。问题在于，当os的swap行为和cgroup不同时，到底哪个该生效的问题。

解决办法：
•for cgroup v1，设置vm.swappiness = all cgroups memory.swappiness
•for cgroup v1，很多解决办法，参考https://access.redhat.com/solutions/6785021
•使用cgroup v2。v2新增vm.force_cgroup_v2_swappiness参数，使cgroup的memory.swappiness失效

VM-overcommit

linux不是为每个虚拟地址预留物理内存，而是等到实际需要内存时才进行分配。overcommitment可以限制所有进程申请的总虚拟内存的大小，申请的内存超过限定的物理内存大小时，称为overcommit。

overcommit策略参数有三个：overcommit_memory，overcommit_ratio/overcommit_kbytes

有三种overcommitment策略，通过overcommit_memory参数控制：
0(默认): 启发式overcommitment策略，允许轻微的overcommit，CommitLimit=物理内存+swap 。适用于大多数普通系统，确保系统稳定性
1: 无overcommit检查 ，允许分配超过物理内存的内存，会提高内存利用率，但可能导致内存不足时系统崩溃或进程被终止。
2: 严格限制，禁止超过CommitLimit。内核允许分配超过物理内存和交换空间总和的内存，但会根据overcommit_ratio/overcommit_kbytes参数设置的比率来决定是否接受请求。如果vm.overcommit_ratio 设置为 50，那么内核允许分配的内存是物理内存和交换空间总和的 1.5 倍

VM-overcommit

overcommit accounting

Status

- We account **mmap** memory mappings
- We account mprotect changes in commit
- We account mremap changes in size
- We account **brk**
- We account munmap
- We report the commit status in /proc
- Account and check on **fork**
- Review stack handling/building on exec
- SHMfs accounting
- Implement actual limit enforcement

mmap、brk、fork都计算在内，明显对pg是有影响的

VM-overcommit


```
$ cat user_reserve_kbytes
131072
$ cat admin_reserve_kbytes
8192
```

reserve内存和overcommmit

user_reserve_kbytes：在overcommit_memory=2时，为普通用户进程预留的物理内存，在系统内存严重不足时，确保普通用户仍能执行基本操作（如启动新进程、处理内存分配请求）。默认值为min(3% of the current process size, 128M)。当设置为0时，单个进程可以分配（所有空闲内存-admin_reserve_kbytes)

admin_reserve_kbytes：为具备 CAP_SYS_ADMIN 权限的用户（通常是 root 用户）保留的物理内存，确保管理员恢复能力，确保系统管理员可以登陆并执行命令的保留物理内存。默认min(3%内存, 8MB)。严格限制overcommit模式时最好提高该参数。

VM-overcommit

CHINA POSTGRESQL ASSOCIATION

```
$ grep -E 'CommitLimit|Committed_AS' /proc/meminfo
CommitLimit:    203103492 kB
Committed_AS:   252170700 kB
$  sar -r 1
07:32:35 PM kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive
07:32:37 PM  25472180 370249056     93.56     14588 274485956 252242936     62.91 233866528
07:32:38 PM  25471904 370249332     93.56     14588 274487888 252242740     62.91 103570136
```

观测指标含义：
meminfo CommitLimit：由物理内存、Swap及overcommit参数共同计算得出的CommitLimit
meminfo Committed_AS：当前所有进程已申请的虚拟内存总量
sar -r kbcommit = Committed_AS
sar -r %commit =kbcommit/总物理内存

smaps或者status也可以查看总申请的虚拟内存，但直接算smaps/status的总计虚拟内存重复计算了共享库文件，映射文件（如mmap），而Committed_AS 仅统计mmap、brk、fork等申请的内存，且不会重复计算共享内存。
两者计算口径不同，看总虚拟内存还是看Committed_AS或者kbcommit就行。

VM-watermark

参数名称	作用描述	引入版本	默认值	单位/范围
min_free_kbytes	定义系统保留的最小空闲内存量，直接影响内存水位 watermark[min] 的计算，确保系统在内存紧张时保留足够内存供关键操作使用	早期内核版本	极小	KB
watermark_scale_factor	全局调节内存水位线间距（high-low 和 low-min）	Linux 内核4.6	10（0.1%物理内存）	最大3000（30%物理内存）
watermark_boost_factor	临时提升内存高水位线（high），触发积极内存回收以减少碎片化	Linux 内核4.6	15000（即 1.5 倍原始高水位线）	

VM-watermark

在Linux 内核4.6前，min、low、high是固定比例，只能通过设置min_free_kbytes来改变low、high的值：
min：low：high=1：1.25：1.5
固定比例的问题：
本身应该提高low积极触发kswapd异步回收，降低min减少direct reclaim。在4.6前只能通过调整min来间接调整low/high，通过调整min来调整kswapd的delta工作缓冲。例如

	kswapd异步回收工作缓冲（low-min）	**kswapd异步回收工作量（high-low）**
min=1GB，low=1.25GB，high=1.5GB	0.25GB	0.25GB
min=10GB，low=12.5GB，high=15GB	2.5GB	2.5GB

提高min是为了提高low和high。
过低的min值，会导致kswapd还没来得及异步回收更多的内存，direct reclaim就触发了。过高的min，不仅浪费了内存，也会导致内存回收动作更频繁，sys cpu会更多。Linux中默认的low与min之间的差值确实显得小了点。

VM-watermark

watermark_scale_factor

如果可以直接调整min、low、high不是美滋滋吗？抱歉，linux内核没有（安卓有extra_free_kbytes）。
但是，
linux4.6内核以后新增watermark_scale_factor参数可以调节参数之间的比例，比例不再是固定的了。其默认值为10，对应内存占比0.1%(10/10000)，最大为3000。当它的值被设定为1000时，意味着"low"与"min"之间的差值，以及"high"与"low"之间的差值都将是内存大小的10%(1000/10000)。
0.1%明显是小了，1T的内存，scale才1GB。

watermark_boost_factor

watermark_boost_factor 用于优化内存外碎片化。它临时提高内存管理区的水位，即 zone->watermark_boost 从而提高内存管理区的高水位（ WMARK_HIGH），这样 kswapd 可以回收更多内存，内存规整模块（ compactd 内核线程）就比较容易合并大块的连续物理内存。 watermark_boost_factor 的默认值是 15000，表示会临时把原来的高水位提升到 150%。若把这个值设置为 0，则关闭临时提高内存管理区水位的机制

VM-watermark

```
# 从zoneinfo中计算zone的总min等值
cat /proc/zoneinfo | grep -E -w "min|low|high"|grep -E -v "high:"| awk '
/min/ { total_min += $2 }
/low/ { total_low += $2 }
/high/ { total_high += $2 }
END {
  printf "总min: %d KB\n总low: %d KB\n总high: %d KB\n",
        total_min * 4, total_low * 4, total_high * 4;
}'
总min: 15828844 KB
总low: 19786048 KB
总high: 23743260 KB

#当前系统min值
cat min_free_kbytes
15828849
```

VM-watermark

因为有其他zone，所有zone的总min加起来才差不多min_free_kbytes。Normal zone的min肯定比min_free_kbytes小一点点，只需要关注Normal zone就行了：

```
# Normal zone的min、low、high设置；page=4k
cat /proc/zoneinfo | grep -A 50 Normal | grep -E "min|low|high"
        min      3931615
        low      4914518
        high     5897422
```

VM-OOM

OOM killer 是在系统内存不足时触发的，通常与 overcommit机制有关，一般代表物理内存不足。cg oom 是另外一个故事。

参数名称	作用描述	默认值
panic_on_oom	控制OOM发生时系统行为： **0: 不触发panic，启动OOM Killer** 1: 触发panic并停机 2: 触发panic后尝试内存释放	0
oom_kill_allocating_task	是否优先杀死触发OOM的进程（而非遍历进程树选择最优目标）： 0: 禁用 1: 启用	0
oom_dump_tasks	OOM发生时是否转储所有任务信息（用于事后分析）： 0: 禁用 1: 启用	1

VM-OOM

oom发生时，系统需要通过oom score决定要kill哪个进程。 每个用户进程都有3个oom score接口文件

```
-rw-r--r-- 1 postgres postgres 0 May 24 16:39 /proc/63766/oom_adj
-r--r--r-- 1 postgres postgres 0 May 24 16:39 /proc/63766/oom_score
-rw-r--r-- 1 postgres postgres 0 May 24 16:39 /proc/63766/oom_score_adj
```

oom_score是系统自动计算的动态oom分数，至少受以下影响：

- 很多子进程的 +分
- 运行很长时间的 -分
- low nice value +分 (nice value代表进程的cpu时间片优先级。nice值越低，优先级越高，CPU时间片分配更多)
- 直接访问硬件的 -分

VM-OOM

除了linux自己算的oom分数，还可以自行调整（adj）oom分数。oom_adj是linux内核早期版本的，最好通过调整oom_score_adj接口文件来调整adj score。

参数/文件	作用	示例值
oom_score	内核计算的原始评分（动态变化）	0~1000
oom_score_adj	用户自定义调整值，直接影响最终评分	-1000~1000；-1000等价于禁用OOM
oom_adj（旧版）	旧版调整参数，范围-17~15	-17~15

VM-lowmem_reserve_ratio

除了min_free_kbytes，还有一个最低内存保留参数，会可能导致进程申请内存失败，但他们功能有较大差别。
lowmem_reserve_ratio 是一个用于保护低端内存（DMA、DMA32）不被高端内存分配请求过度占用内核参数。lowmem_reserve_ratio只是一个系数，并不是直接可用的数，内核会计算每个zone的保留页数。

```
#下面默认值
cat /proc/sys/vm/lowmem_reserve_ratio
256     256     32
```

内存区域（zone）按优先级从低到高排列：DMA → DMA32 → Normal → HighMem。高优先级区域的分配请求可“借用”低优先级区域的内存，但需按比例保留一定内存供低优先级区域使用。

VM-lowmem_reserve_ratio

```
cat /proc/zoneinfo |grep -Ew "Node 0|protection|free"
Node 0, zone      DMA
  pages free     3976
        protection: (0, 2484, 386430, 386430)
Node 0, zone    DMA32
  pages free     415741
        protection: (0, 0, 383946, 383946)
Node 0, zone   Normal
  pages free     5658528
        protection: (0, 0, 0, 0)
```

例如DMA的 protection 表示：

- 0:本区分配，没有跨区分配的限制
- 2484：DMA为来自DMA32区域保留的页数。
- 386430：DMA为来自Normal区域保留的页数
- 386430：保留扩展字段，此场景中无意义

以上设置

- 当DMA32区申请DMA区的内存时，3976>2484，是有可能成功的
- 当Normal区申请DMA区的内存时，3976<386430，是不会成功的
- 地区向高区申请内存，不会受到该限制

/02

PAGES的计算

pages的计算-inactive_anon+active_anon != anon

inactive_anon+active_anon != anon

why?
•主要：Shmem 会单独统计共享内存的页数。nr_anon_pages 未包含共享内存页，而 nr_inactive_anon 和 nr_active_anon包含匿名共享内存页
•次要：anon包含部分Unevictable （Mlocked也是Unevictable的子集）
•其他统计口径差异影响不大

pages的计算-anon的计算

粗糙但比较准确的算法：nr_inactive_anon +nr_active_anon +nr_unevictable -nr_shmem。该脚本大页下试用；numa下不适用

```
# /proc/meminfo,/proc/zoneinfo,/proc/vmstat都可以计算
#/proc/vmstat
echo -n "anon_computed        : ";cat /proc/vmstat|egrep -w
"nr_inactive_anon|nr_active_anon|nr_unevictable|nr_shmem"| awk 'NR==1 {a=$2} NR==2 {b=$2}
NR==3 {c=$2} NR==4 {d=$2; print (a+b+c-d)}' ;\
echo -n "anon_real            : ";cat /proc/vmstat|egrep -w "nr_anon_pages"|awk '{print $2}'
anon_computed        : 15776924
anon_real            : 15772671

##/proc/zoneinfo Normal
echo -n "anon_normal_computed        : "; cat /proc/zoneinfo |grep Normal -A 50|egrep -w
"nr_inactive_anon|nr_active_anon|nr_unevictable|nr_shmem"| awk 'NR==1 {a=$2} NR==2 {b=$2}
NR==3 {c=$2} NR==4 {d=$2; print (a+b+c-d)}' ;\
echo -n "anon_normal_real            : "; cat /proc/zoneinfo |grep Normal -A 50|egrep -w
"nr_anon_pages"|awk '{print $2}'
anon_normal_computed        : 15711170
anon_normal_real            : 15707402
```

pages的计算-cache的计算

free命令中的buff/cache，可以通过文件页或者cache本身计算出来。cache包含buffer+file页+共享内存+可回收的slab。注意shmem计算在cache中

```
echo -n "filepage+shmem:  ";cat /proc/meminfo |grep -Ew
"Buffers|Active\(file\)|Inactive\(file\)|Shmem|SReclaimable"| awk 'NR==1 {a=$2} NR==2 {b=$2} NR==3
{c=$2} NR==4 {d=$2} NR==5 {e=$2 ;print (a+b+c+d+e)}';\
echo -n "cached:          ";cat /proc/meminfo |grep -Ew "Buffers|Cached|SReclaimable" | awk 'NR==1
{a=$2} NR==2 {b=$2} NR==3 {c=$2 ;print (a+b+c)}';\
free -k;
```

#执行结果：

filepage+shmem: 289417584
cached   :   289419156

	total	used	free	shared	buff/cache	available
Mem:	395721236	79633516	26668564	84704912	**289419156**	178501152
Swap:	5242876	0	5242876			

pages的计算-shmem算不算cache?

很明显，上面在计算cache的时候，把shmem也算进去了。shmem理论上应该不属于cache的部分
实际上内核社区也讨论过这个事情Why is Shmem included in Cached in /proc/meminfo?，想把共享内存从cache中抛出去

```
> -        cached =
global_node_page_state(NR_FILE_PAGES) -
> -                                 total_swapcache_pages() -
i.bufferram;
> +        cached =
global_node_page_state(NR_FILE_PAGES) -
> +                                 total_swapcache_pages()
> +                                 - i.bufferram - i.sharedram;
```

但是修改这个东西涉及向前兼容，问题就归结于向前兼容和修改一个信息表达的准确性谁更重要?
目前来看没有一个好的解决，反正现状就是这样了。

pages的计算-为什么内存页统计老对不上

计算内存页的时候，有些计算对不上，汇总原因如下：

- shmem算到cache中
- 看不到shmem中的文件映射和匿名映射pages
- nr_anon_pages 未包含共享内存页，而 nr_inactive_anon 和 nr_active_anon包含匿名共享内存页
- vm和cgroup的统计口径略有差别

/03

CGROUP内存管理（v1）

cg mem-cg mem管理概述

cgroup可以观察匿名页，文件页，swap cache，kernel mem的使用情况，并对资源使用做一些限制。每个memcg都有独立的LRU。
cgroup内存管理不同于cgroup cpu管理，1个任务可以申请很多cpu工作，达到cg cpu上限可延长执行时间来处理，但是这个任务占用的内存是工作内存，一个任务使用相同的物理内存。

cgroup管理cpu和内存的重要区别：
•内存必须通过复用和回收来管理，一个任务的工作内存是真实占用的不可被其他任务使用的；CPU通过时间分配来管理，其他任务或cg组可以用到
•内存需要即时可用，CPU通过时间片轮转，时间可以分散
•CPU control的核心是时间分配；Memory Control的核心是page计数

Memory Control的核心是page计数，也就是说不是物理page分这些就是这些，这次申请的内存使用完后释放回free，下次申请基本上不会是同一个物理page

cgmem-接口文件

cgroup通过接口文件配置和查看内存使用情况

```
ll /sys/fs/cgroup/memory/xxx/
```

内核内存和mem+swap可以单独设置或查看使用上限和使用情况：

memory.kmem.xxx #kernel mem

memory.memsw.xxx #mem+swap

接口文件可分为三类：

- 只读-显示使用情况，权限:-r--r--r--
- 读写-控制参数，权限:-rw-r--r--
- 其他-特殊设置，权限:其他

cgmem-只读文件

只读	**memory.stat**	**重要**，是主要的内存使用情况接口文件，有很多指标，后面单独分析
只读	memory.usage_in_bytes	usage_in_bytes is affected by the method and doesn't show 'exact' value of memory。不建议使用该文件查看cgroup的内存使用情况
只读	**memory.failcnt**	内存使用量超过memory.limit_in_bytes的次数，累计值
读写	memory.max_usage_in_bytes	cgroup使用峰值，属于观察指标

cgmem-控制参数

读写	**cgroup.procs**	用于管理当前 cgroup 中的 进程组（进程 ID，PID）。**对于多进程的pg来说，就是把pg的所有进程，包括管理进程和backend都写入procs文件**
读写	**memory.limit_in_bytes**	**cgroup内存上限**
读写	**memory.soft_limit_in_bytes**	优先回收的部分，约等价于v2的memory.low
读写	**memory.oom_control**	oom_kill_disable 0启用（默认），1禁用 oom under_oom 是否处于OOM状态，0为不是
读写	**memory.swappiness**	cg级别的swappiness

cgmem-stat文件

shared_memory_type=mmap，shared_buffers=64，clients 800个左右，running

```
cache 345587761152                  #page cache！！！
rss 27332608                        #匿名和swap cache内存大小，注意，跟OS的进程rss不同，这里明
显不包含PG的共享内存
rss_huge 0                          #of bytes of anonymous transparent hugepages，注意是透明大页
mapped_file 61491769344             # 文件共享内存大小，这里包含pg的共享内存
swap 0                              # swap分区上的
pgpgin 389395357                    # rss+cache的charge pages
pgpgout 305016672                   # rss+cache的uncharge pages
inactive_anon 165728256             # anonymous and swap cache memory on inactive LRU
active_anon 61549518848             # anonymous and swap cache memory on active LRU list
inactive_file 138240962560          # file-backed on inactive LRU list
active_file 145658613760            # file-backed memory on active LRU list
unevictable 0                       # 无法回收的内存
total_xxx                           # hierarchical
```

cgmem-内存计算

粗略来看（不看swap）cache+rss=inactive_anon+active_anon+inactive_file+active_file。
这上面的值还挺绕的，cache+rss跟[in]active_anon/file也难有直接对应关系，再加上mapped_file这个共享内存不知道该算到哪去，容易算晕。结合各种文档和测试，手搓脚本计算：

#shared_buffers=2GB的库
shared_mem_mapped : 1.69063
shared_mem_anon : 1.69828
pagecache : 5.94717
pagecache_cache-share : 4.25654
file_cache : 4.24889
anon_cache : 3.23096
total_used_rss+map : 3.2233
total_mem_file+rss+map: 7.47219
total_mem_rss+cache : 7.47984
total_mem_anon+file : 7.47984
hard_limit : 8

shared_mem_mapped= inactive_anon+active_anon-rss
cache-shared_mem_mapped=inactive_file+active_file
rss+mapped_file=inactive_anon+active_anon

inactive_file+active_file+rss+mapped_file=
inactive_file+active_file_inactive_anon+active_anon=
rss+cache

cgmem-cg rss和process rss的差异

CHINA POSTGRESQL ASSOCIATION

#shared_buffers= 64GB,所有pg进程的rss排序

```
ps -eo pid,ppid,rss,args |grep `cat $PGDATA/postmaster.pid|head -1`|sort -k3 -rn
 97632  97627 61103720 postgres: lzlinst: checkpointer
 97633  97627 59045152 postgres: lzlinst: background writer
 97627      1  2322820 /paic/postgres/base/11.3/bin/postgres -D /paic/pg6888/data
 97637  97627 85116 postgres: lzlinst: pgsentinel
 97697  97627 19620 postgres: lzlinst: dbmgr users [local] idle
 97634  97627 17932 postgres: lzlinst: walwriter
250063  97627 14508 postgres: lzlinst: dbmon postgres [local] idle
 97636  97627 13220 postgres: lzlinst: stats collector
248777  97627 11576 postgres: lzlinst: dbmon postgres [local] idle
 97635  97627  2980 postgres: lzlinst: autovacuum launcher
 97638  97627  2376 postgres: lzlinst: logical replication launcher
 97630  97627  1592 postgres: lzlinst: logger
250185  39130   972 grep --color=auto 97627
```

一般来说，pg rss值最多的是checkpointer和bgwriter进程，因为rss代表的真实使用的内存，含共享内存部分，这2个要刷shared buffer脏页的进程占用最多。也有查数据过多的backend也可能rss值较高，不过一般是抽数或者全表扫的慢sql导致。

postmaster为什么很少？因为postmaster本身不需要做太多shared_buffer的操作，它只需要把共享内存的虚拟地址开辟下来，fork给其他进程用即可。

cgmem-cg rss和process rss的差异

CHINA POSTGRESQL ASSOCIATION

```
$ cat /proc/97632/smaps |grep -A 3 "zero" #checkpointer
2b4fd87cf000-2b60a2143000 rw-s 00000000 00:04 15925397                   /dev/zero (deleted)
Size:          70411728 kB
Rss:           61087812 kB
Pss:           31429895 kB
$ cat /proc/97633/smaps |grep -A 3 "zero" #bgwriter
2b4fd87cf000-2b60a2143000 rw-s 00000000 00:04 15925397                   /dev/zero (deleted)
Size:          70411728 kB
Rss:           59043388 kB
Pss:           29394787 kB
$ cat /proc/97627/smaps |grep -A 3 "zero" #postmaster
2b4fd87cf000-2b60a2143000 rw-s 00000000 00:04 15925397                   /dev/zero (deleted)
Size:          70411728 kB
Rss:            2318408 kB
Pss:            1741764 kB
```

pm的子进程的共享内存地址是相同的，rss就不一定了

cgmem-cg rss和process rss的差异

前面看到，ckpt和bgwrite的rss都有60GB了，但是，cgroup中的rss只有几十MB，远小于进程的rss

```
cat /sys/fs/cgroup/memory/lzlinst/memory.stat |egrep -w "rss|mapped_file"
rss 88997888
mapped_file 52963262464
```

可以看出来，pg共享内存没有在cgroup stat rss的统计中。cgroup的rss没有计算file page以及shared file page。cg把pg mmap的内存算成mapped_file。

观察sysv和大页的情况，pg的memory.stat相关指标总结：
- stat中的rss不包含file map共享内存。观察来看，无论是mmap还是sysv，rss都不含pg的共享内存
- 同理，rss_huge也不含file map共享大页内存。观察来看，即便开启大页，stat也不含pg的共享内存
- 无大页时，pg的共享内存（mmap or sysv）均统计在memory.stat mapped_file下；有大页时，不在stat中的任何指标中，包括rss_huge

cgmem-cg oom

正常来说，sharedbuffer=1/4的cg mem，那么在不计算私有内存的情况下，pagecache最大可以到3/4的cg mem。一般来说，正常的业务私有内存占用不会很多，如果cg mem打满是可以从cg pagecache中回收内存的。所以测试cg oom最好的办法是用占用很多私有内存的会话而不是加压。

```
# vm oom打开; 0:不触发panic，启动OOM Killer
# memory.oom_control oom_kill_disable 1或者0，关闭和启用cg oom

# 一个可以占用很多私有内存的sql，很多union all有很多plan node
psql -d lzldb -tX -c "create table lzl1(col1 varchar(1));"
psql -tX -c "\o sqltext.sql" -c "
SELECT 'select col1 from lzl1' || ' union all'
FROM generate_series(1, 100000)
UNION ALL
SELECT 'select col1 from lzl1;'
FROM generate_series(1, 1);
"

#调整stack参数不然sql会被掐掉
psql -d lzldb -c "set max_stack_depth=1024000" -f sqltext.sql
```

cgmem-cg oom

oom off

```
$ ps -eo user,ppid,pid,state,%cpu,%mem,stime,wchan:14,args,rss,vsz |grep `head -1
$PGDATA/postmaster.pid` |grep -v grep
postgres  19005    870 D  0.0  0.0 10:54 mem_cgroup_oom postgres: pg3ymhp2: lzluser  7216 2807460
postgres  19005   3417 S  0.0  0.0 10:55 pipe_wait      postgres: pg3ymhp2: lzluser 22944 2808540
postgres  19005  13069 D  0.0  0.0 11:10 mem_cgroup_oom postgres: pg3ymhp2: lzluser 11944 2808348
postgres  19005  13104 D  0.0  0.0 11:10 mem_cgroup_oom postgres: pg3ymhp2: lzluser 12224 2808348
postgres  19005  14352 D  0.0  0.0 11:10 mem_cgroup_oom postgres: pg3ymhp2: lzluser 11680 2808348
```

```
cg memory.oom_control under_oom 1
/proc/97994/oom_score 11
```

```
total_used_rss+map          : 7.99789
total_mem_file+rss+map      : 7.99789
total_mem_rss+cache         : 8
total_mem_anon+file         : 8
total_memsw                 : 8
hard_limit                  : 8
```

cgmem-cg oom

oom on

用户进程因为oom score高而被kill，发送的是kill -9，pg绝大部分进程奔溃，postmaster reset_shared()后自动拉起其他进程。mesage和dmesg都有out of memory相关信息：
pg log：server process (PID 236413) was terminated by signal 9: Killed","Failed process was running: select col1 from lzl1 union all
message/dmesg: lzlhost kernel: Memory cgroup out of memory: Kill process 236413 (postgres) score 497 or sacrifice child

cg oom on和off对于pg库管理上的区别：
•on，cg oom killer会kill oom score高的进程，一般来说是用户进程
•off，cg oom killer不会启动。pg进程会hang，当然也可能自己恢复，但pg的关键进程（如walwriter）可能因内存不足而跑崩，实例一样可能挂掉。

注意这里说的cg oom，不是vm oom。系统级的vm oom主要由系统级的vm overcommit机制来判断。

cgmem-缺啥

没有统计cg pagetable
没有统计cg slab
没有统计cg hugepage（hugepage是没有charge，还不是没有算进去）
没有统计cg异步、同步回收pages
cg rss与process rss统计口径不统一
shmem统计口径比较乱

/04

what’s new in CG V2

cg mem v2

V2 Officially released in Linux 4.5 (March 2016)

v2 cg mem在管理上有如下优势：
•相对v1，v2有更简单明了的层级管理
•v1只有OOM kill or freeze，v2有更多手段控制内存大小(如memory.min/low/high)
•v2更容易控制突刺负载
•移除直接关闭cg oom killer的接口文件
•增加memory_hugetlb_accounting

v2 cg mem在观测上有如下优势：
•新增slab、pagetable、pgscank/pgsand/pgsteal、大页信息，这些都是v1没有的
•更多具体特性相关的观测指标，比如sock 、vmalloc、透明大页、zswap压缩交换、swap_zero全零交换等
•共享内存shmem和file_mapped指标分开

REF

V2 Officially released in Linux 4.5 (March 2016)

v2 cg mem在管理上有如下优势：
•相对v1，v2有更简单明了的层级管理
•v1只有OOM kill or freeze，v2有更多手段控制内存大小(如memory.min/low/high)
•v2更容易控制突刺负载
•移除直接关闭cg oom killer的接口文件
•增加memory_hugetlb_accounting

v2 cg mem在观测上有如下优势：
•新增slab、pagetable、pgscank/pgsand/pgsteal、大页信息，这些都是v1没有的
•更多具体特性相关的观测指标，比如sock 、vmalloc、透明大页、zswap压缩交互、swap_zero全零交互等
•共享内存shmem和file_mapped指标分开

THANKS

sharedbuffers最佳实践
内存碎片和多cg组的关系
碎片管理、大页管理
大内存主机和大内存库的管理

刘智龙
DBA