HOW 2026
Hello Open-source World

以开源之道·见致远之志
开源生态大会暨PostgreSQL高峰论坛

PG在云环境下的CG内存管理

目录
CONTENTS

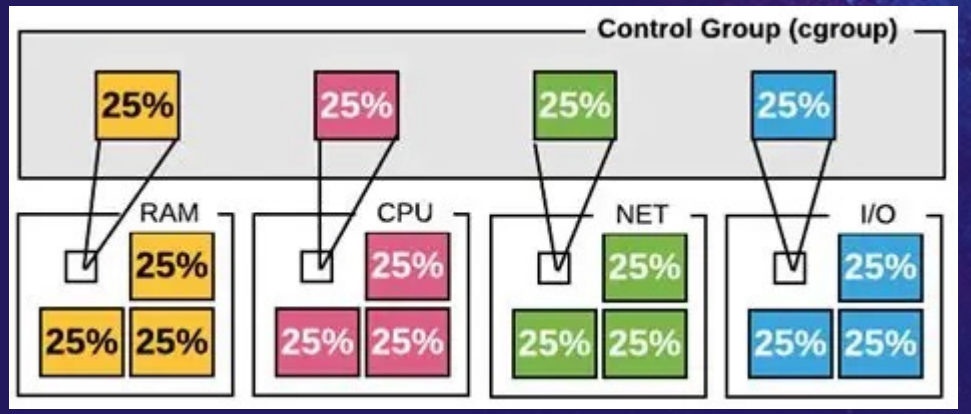
CGOUOP
资源管理

CGROUP管理内存和cpu资源

什么是CGROUP?
Cgroup（控制组）是 Linux 内核提供的进程组**资源管理**机制，用于对一组进程的 CPU 时间、物理内存页面、IO、网络等资源进行配额限制、隔离与使用统计。
CGROUP使用非常广泛，容器也是用的cgroup来管理资源的。对于PostgreSQL来说，无论是不是容器管理，都可以用CGROUP来管理资源，这也是云厂商的常见做法。

CGROUP对CPU和内存的管理
CGROUP可以管理和观察CPU、内存等等
- 对于CPU来说，最重要的是CPU的使用率
- 对于内存来说，可以管理/观察匿名页，文件页，swap，cache，共享内存，kernel mem的使用情况。

每个memcg都有独立的LRU，CG内回收内存时相互不影响。



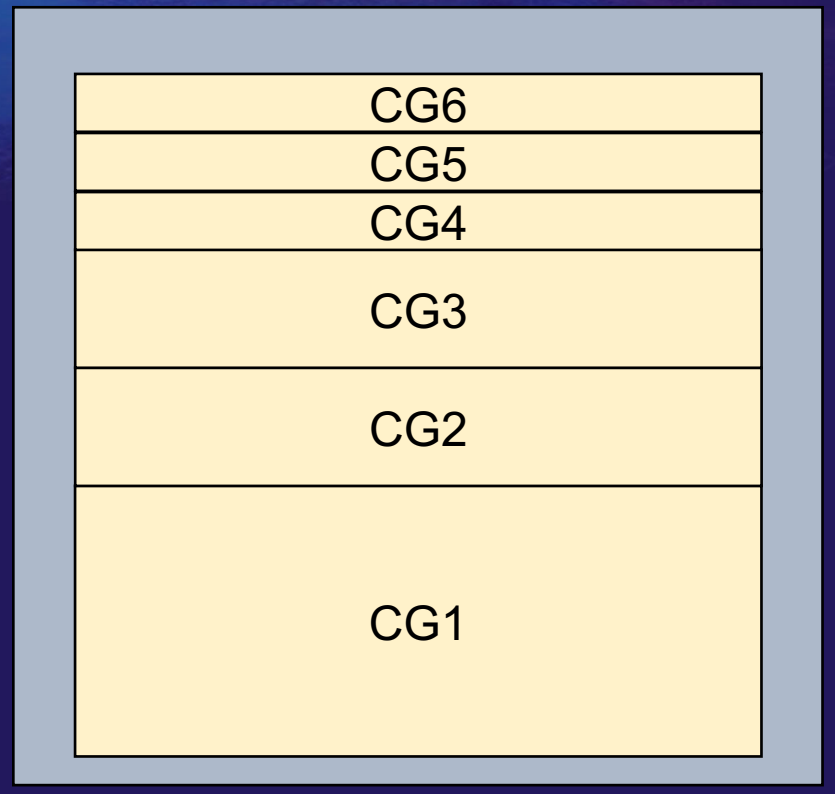
CGROUP管理cpu和内存资源的区别

- 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
- 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，达到上限只有回收。

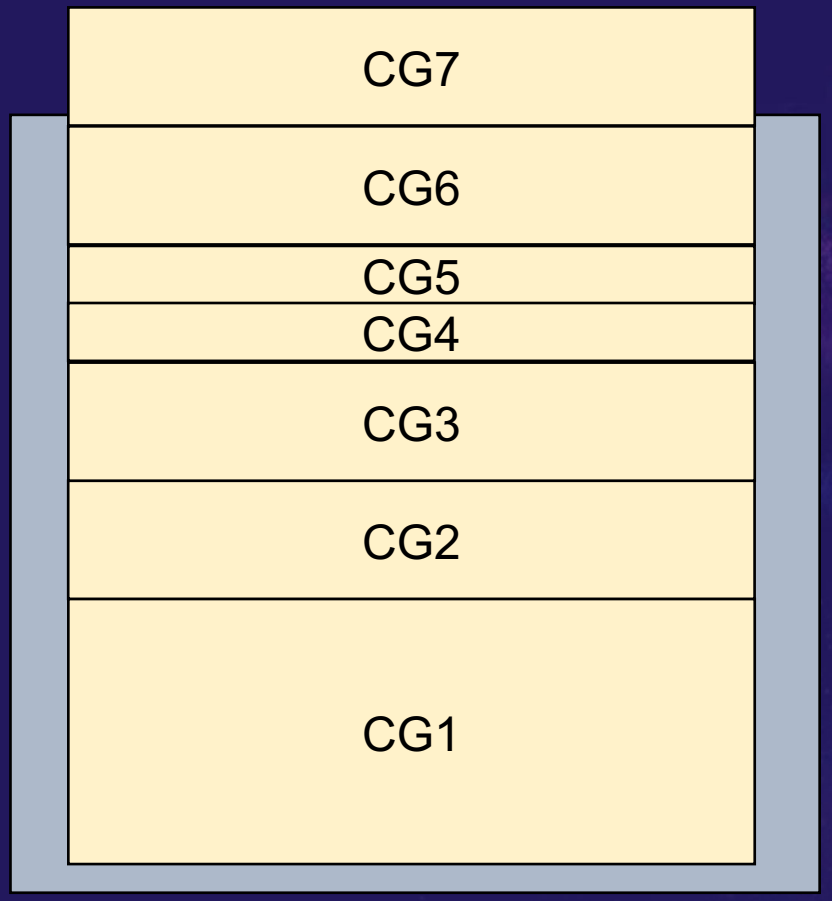
另外，Memory Control的核心是page计数，也就是说不是物理page分这些就是这些，这次申请的内存使用完后释放回free，下次申请基本上不会是同一个物理page

云服务器上的共享主机

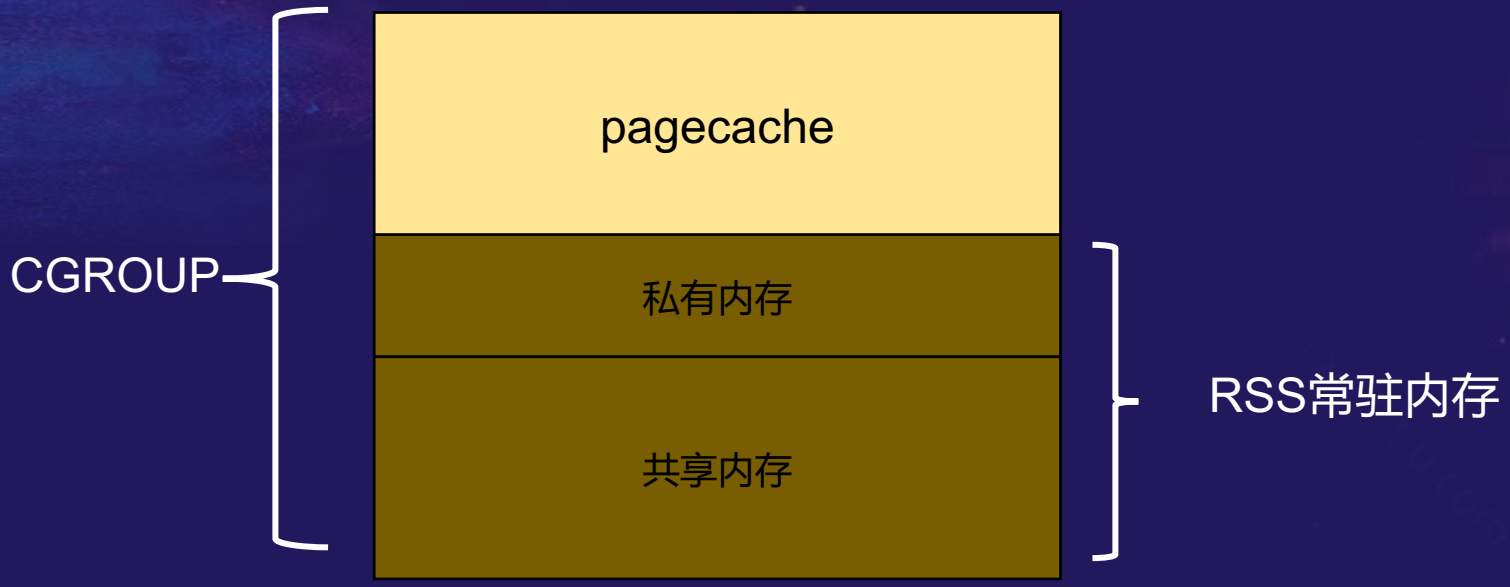
不超卖



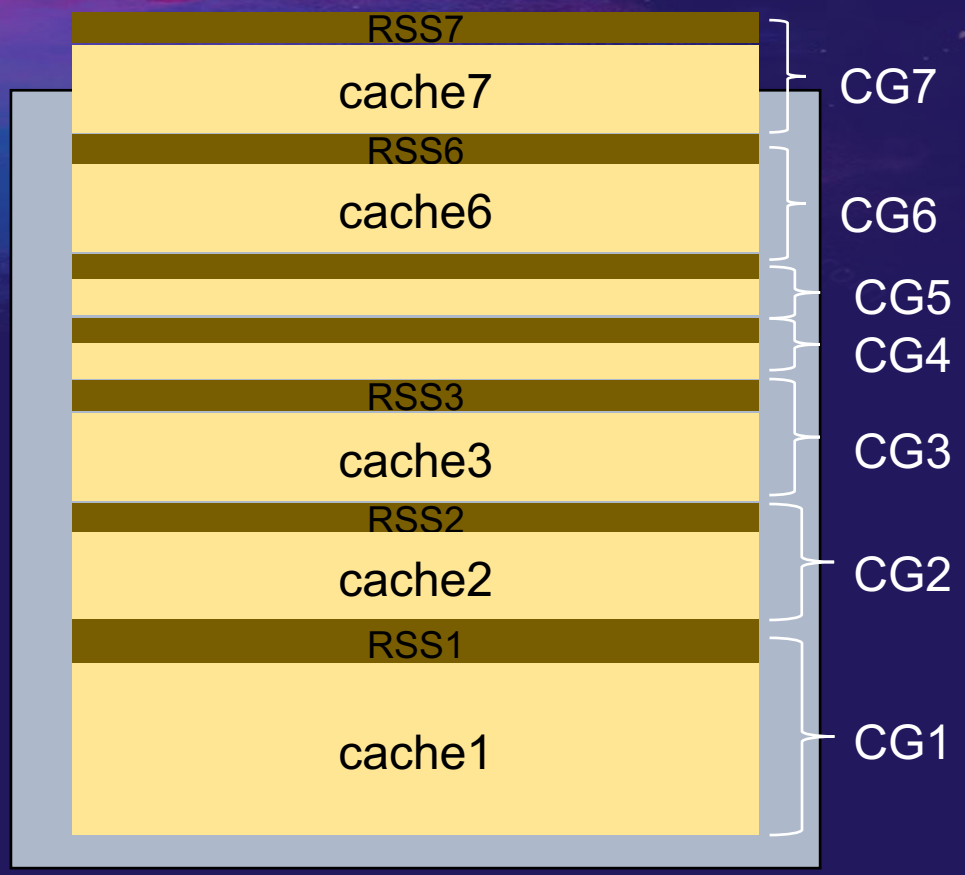
超卖

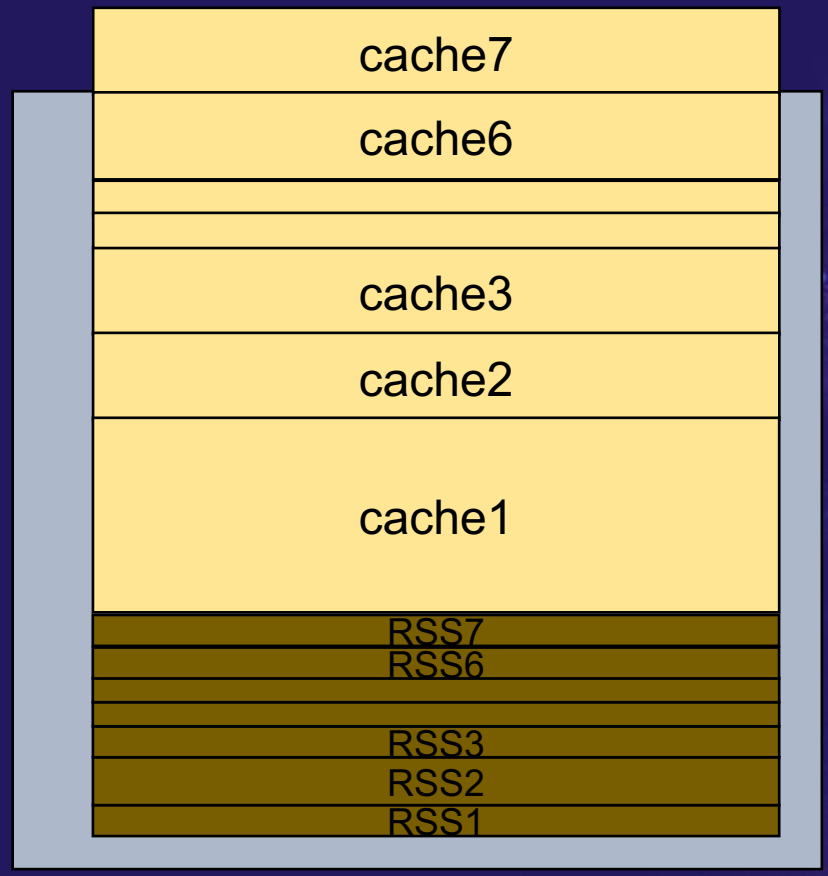


PG在OS上的内存结构

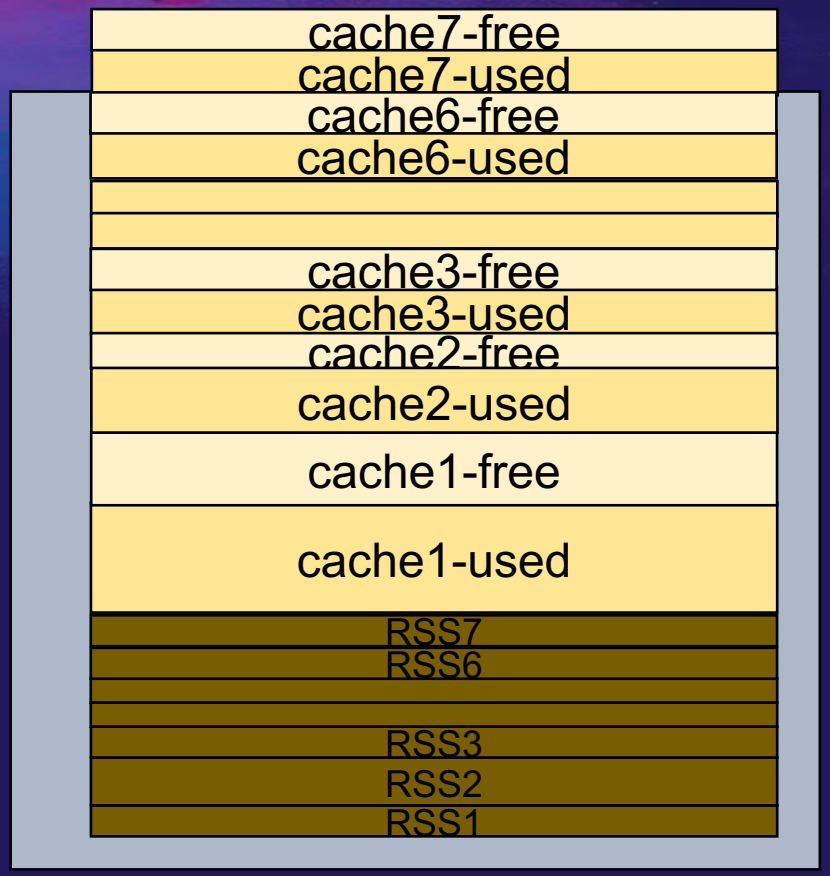


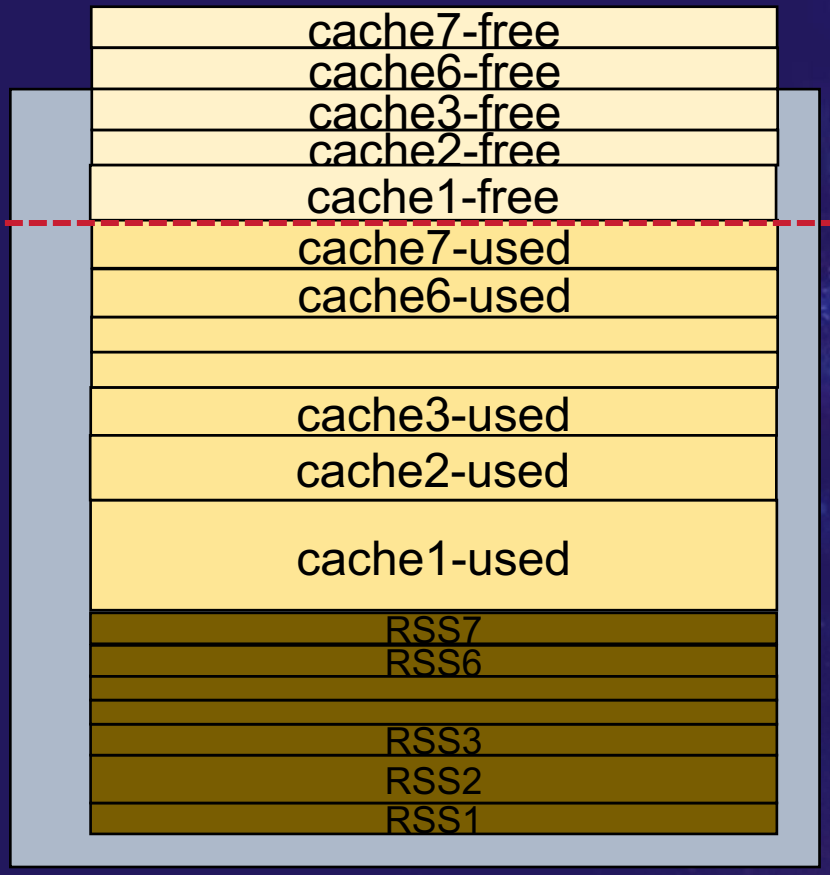
共享云主机上的double buffer PG



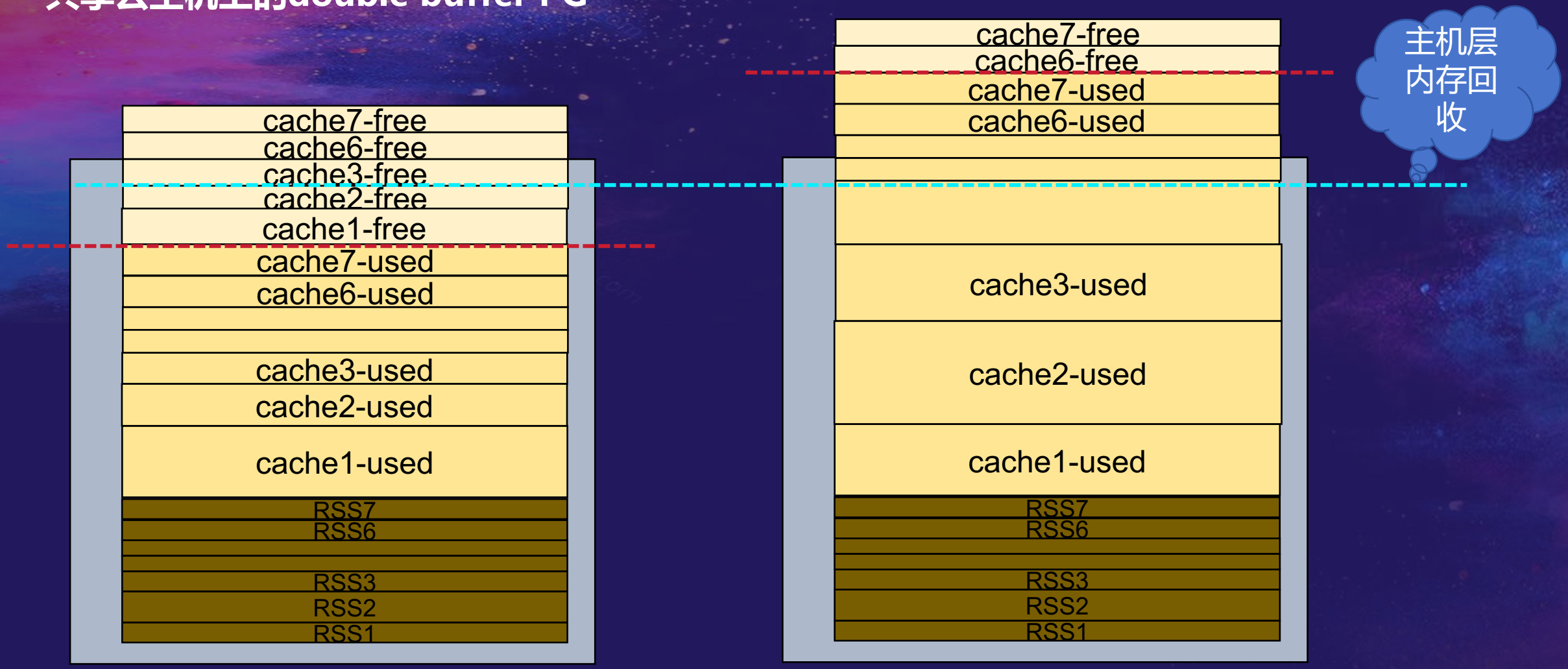


共享云主机上的double buffer PG





共享云主机上的double buffer PG



PAGES 的计算

CGROUP内存中观测文件

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

CGROUP内存中管理文件

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OOM killer（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

memory.stat文件

示例：一个PG库，shared_memory_type=mmap，shared_buffers=64，clients 800个左右，running

stat如下：

```
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
```

pages的计算

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_cache     : 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

```
#示例一个PG，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
 97634  97627 17932 postgres: lzlinst: walwriter
 97635  97627  2980 postgres: lzlinst: autovacuum launcher
 97638  97627  2376 postgres: lzlinst: logical replication launcher
 97630  97627  1592 postgres: lzlinst: logger
```

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

```
$ 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。
因为虚拟内存地址都是fork出来跟父进程是一致的，但RSS是实际使用的物理内存大小

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 --80MB**

mapped_file **52963262464 --50GB**

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。实际上，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

怎么正确模拟oom?

一般来说，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。
所以想要实现cg oom最好的办法是创建占用很多私有内存的会话；而不是加压读写数据，将cache填满打到cgroup memory.limit_in_bytes。

cgmem-cg oom

```
$ 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 --oom killer off**

```
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。

CGROUP V1缺什么

- 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

CGROUP V2有什么

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

大页的管理

大页的优势?

大页对一些内存问题（内存碎片、CG内直接内存回收）有非常好的效果。

PG用大页的优势：

- 可以少许提升TPS
- 减少TLB刷新压力
- **极大的减少pagetable在主内存上的大小。这不仅减少了内存占用，在内存回收时也可更快的通过物理内存找到虚拟内存（pagetable是存映射的）**
- 大页在物理上是连续的。连续的物理内存访问比不连续的物理内存访问更优
- 当使用大页时，page是直接映射的，不会使用多级的pte条目
- **非大页的共享内存理论上可能回收，但大页不会**

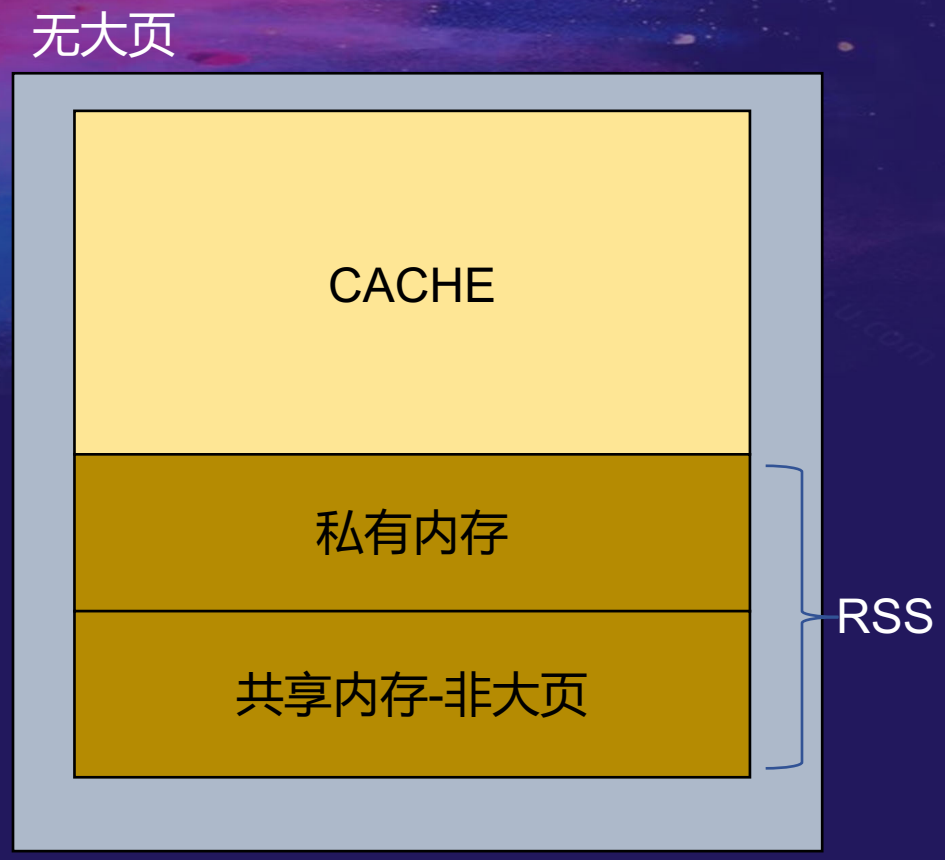
大页多少合适

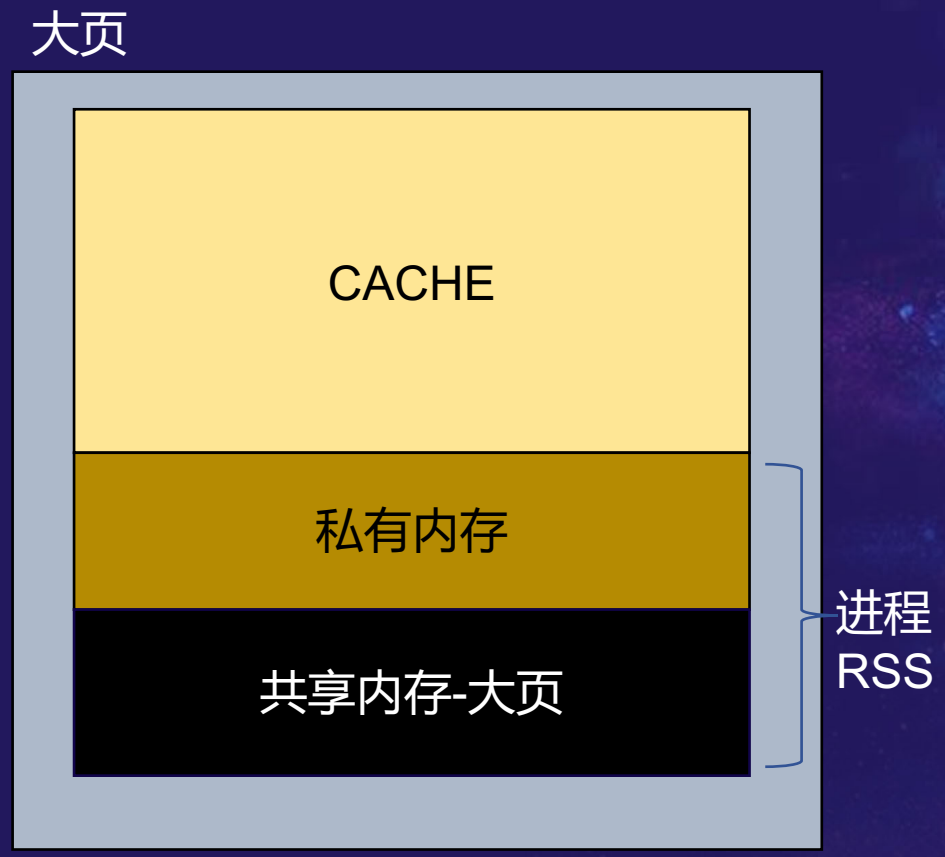
sharedbuffers=1/4 cgmem 似乎以成为行业标准，但是实际情况要复杂的多。理论上，把sharedbuffers调小可以增加一点pagecache，实际上略微增大了整个缓存大小，相反把sharedbuffers调大略微减小了整个缓存大小，但是提升了一定的sharedbuffer命中率。
很明显，sharedbuffers调大了不好，调小了也不好。sharedbuffers调太小，pg自己能工作的内存就太小了，相当于把内存管理工作扔给了OS，OS回收pagecache时对性能也会有影响；sharedbuffers调太大，不仅pagecache被挤占，还需要考虑pg sbs刷脏的影响，特别是写入较多的情况需要调整相应的bgwriter参数。

从粗糙的压测来看，一个建议值，至少比1/4 cgmem靠谱

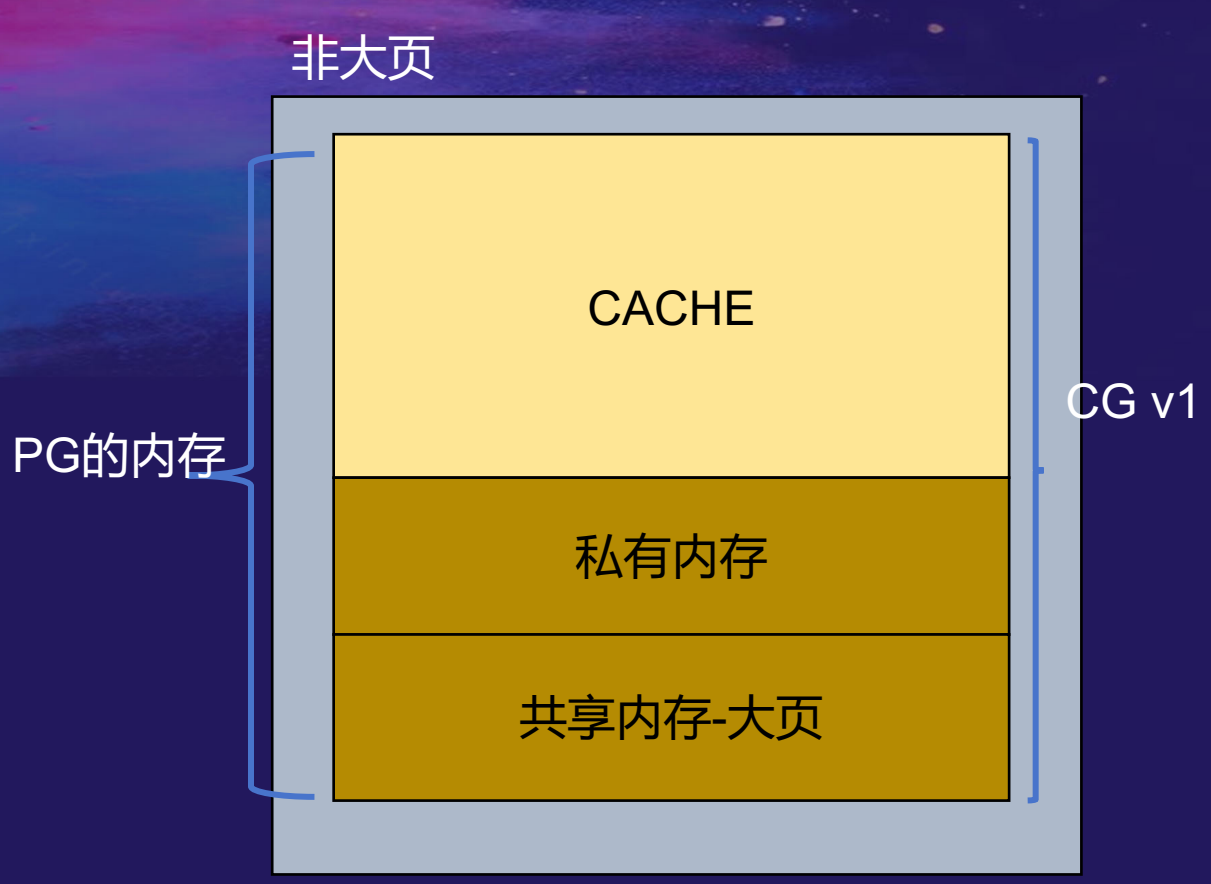
- **不开大页，shared buffers=min(1/4 MEM,20GB)**
- **开大页，shared buffers=min(1/4 MEM,60GB)**

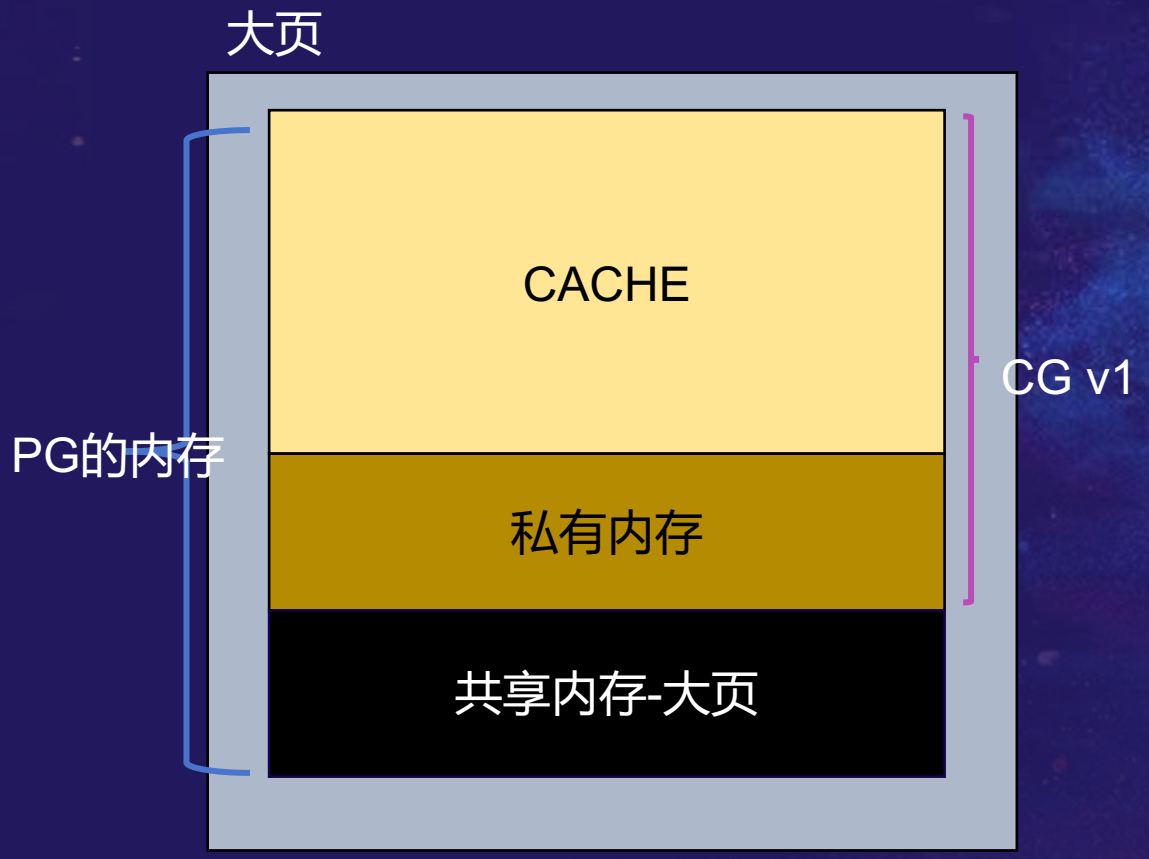
主机上的大页





主机上的大页-PG


非大页
CACHE
私有内存
共享内存-大页
PG的内存
CG v1


大页
CACHE
私有内存
共享内存-大页
PG的内存
CG v1

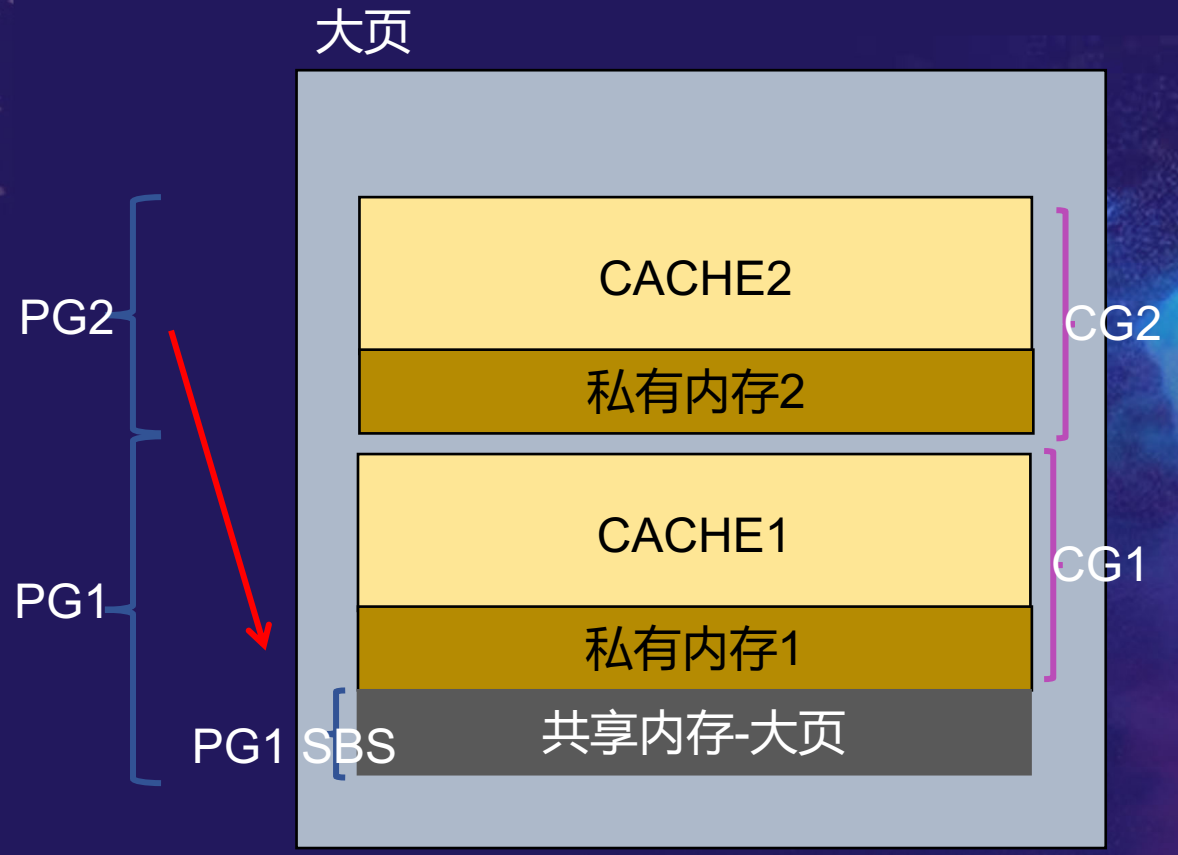
大页带来的管理问题

1. 监控，例如pages的计算，CG统计不到大页，RDS指标可能不像以前那么准确

2. 大页是随主机启动生效的，需要提前规划大小。更难的是如果不知道主机上要分配多少实例，分配什么类型的实例，怎么规划…

3.超卖怎么设置，大页分配后，除了SHARED BUFFER，其他都用不到大页。如果主机物理内存是云资源的基，这个基是否需要改变，调动算法是否需要改变

大页带来的管理问题

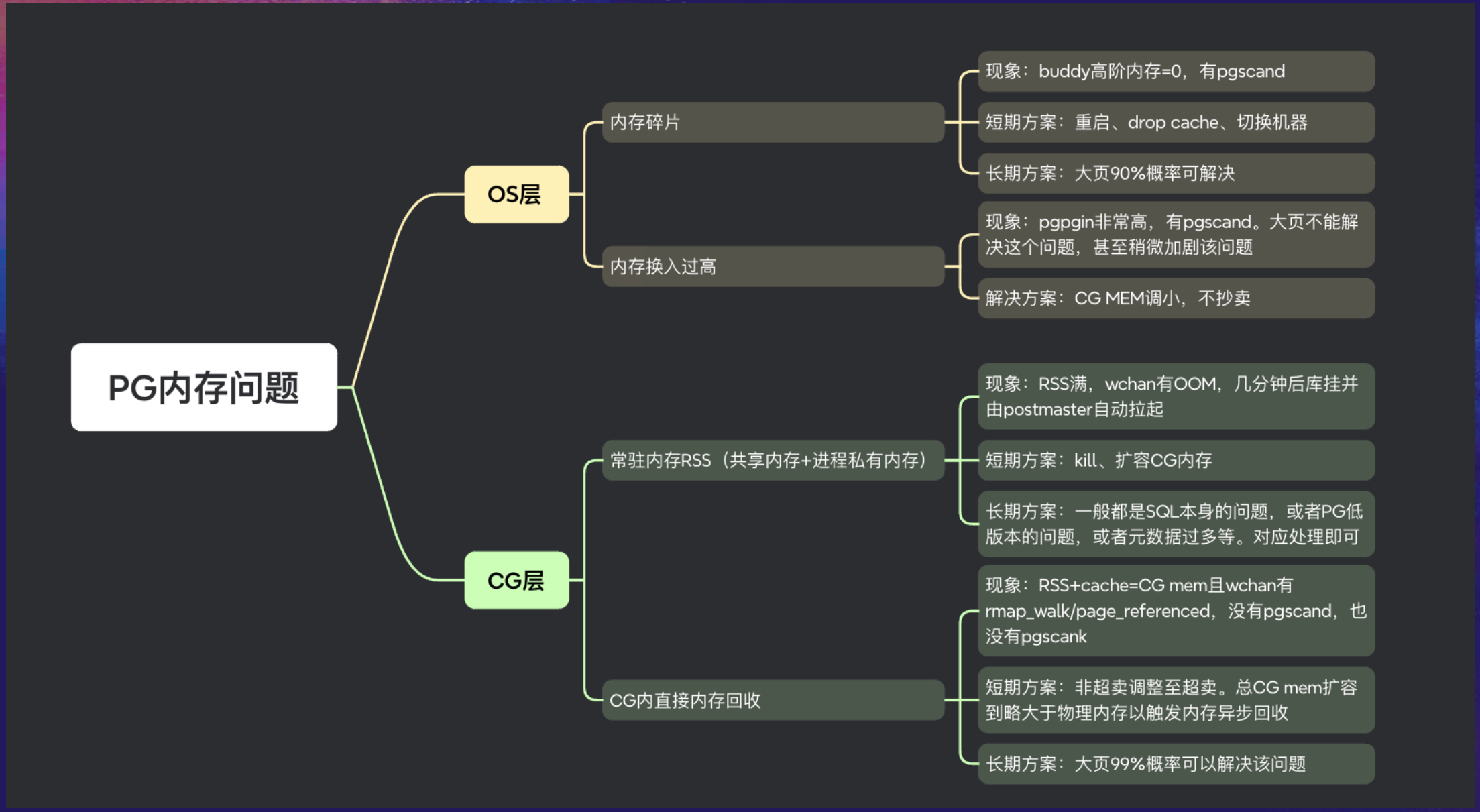


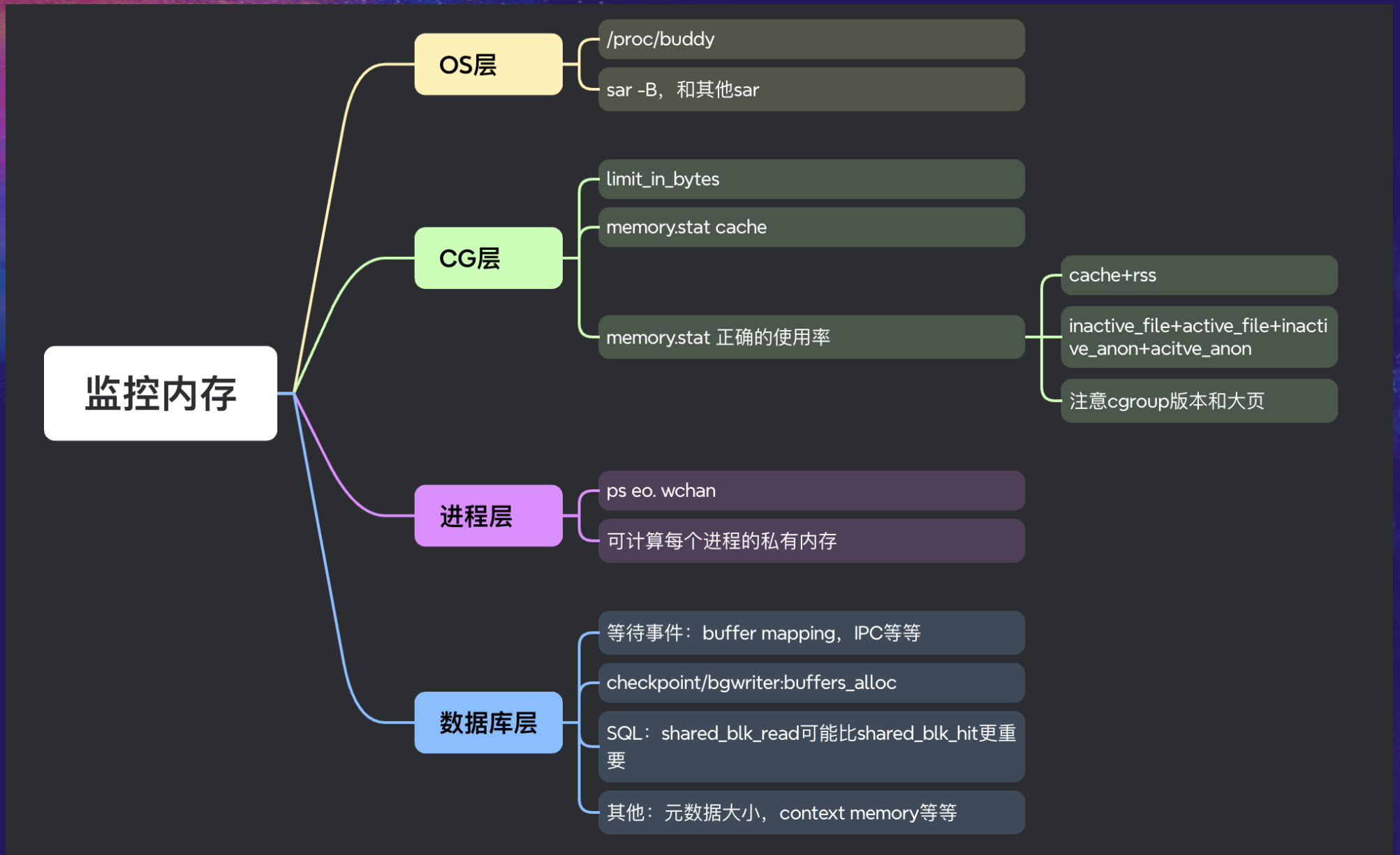
后续的实例分配不到大页，大页不足

大页带来的管理问题

大页过大带来内存浪费

内存故障模型


PG内存问题
OS层
内存碎片
现象：buddy高阶内存=0，有pgscand
短期方案：重启、drop cache、切换机器
长期方案：大页90%概率可解决
内存换入过高
现象：pgpgin非常高，有pgscand。大页不能解决这个问题，甚至稍微加剧该问题
解决方案：CG MEM调小，不抄卖
CG层
常驻内存RSS（共享内存+进程私有内存）
现象：RSS满，wchan有OOM，几分钟后库挂并由postmaster自动拉起
短期方案：kill、扩容CG内存
长期方案：一般都是SQL本身的问题，或者PG低版本的问题，或者元数据过多等。对应处理即可
CG内直接内存回收
现象：RSS+cache=CG mem且wchan有rmap_walk/page_referenced，没有pgscand，也没有pgscank
短期方案：非超卖调整至超卖。总CG mem扩容到略大于物理内存以触发内存异步回收
长期方案：大页99%概率可以解决该问题


监控内存
OS层
/proc/buddy
sar -B，和其他sar
CG层
limit_in_bytes
memory.stat cache
memory.stat 正确的使用率
cache+rss
inactive_file+active_file+inactive_anon+acitve_anon
注意cgroup版本和大页
进程层
ps eo. wchan
可计算每个进程的私有内存
数据库层
等待事件：buffer mapping，IPC等等
checkpoint/bgwriter:buffers_alloc
SQL：shared_blk_read可能比shared_blk_hit更重要
其他：元数据大小，context memory等等


内存相关wchan
直接内存回收
page
page_referenced:active和inactive的转换用reference的进行标记
lock_page_killable:可中断方式的锁定物理内存page
wait_on_page_bit:等待page标识位改变
rmap
rmap_walk:通过物理内存找到页表项
刷盘
blk_dev_issue_flush:刷缓存
ima_file_check:open()
其他： blk*,write*
等待信号/位的改变
rpc_wait_bit_killable:远程调用RPC等待内核中的标识位
call_rwsem_down_write_failed:写锁竞争
OOM
mem_cgroup_oom_synchroize:oom_control_under_oom=1
mem_cgroup_oom:同mem_cgroup_oom_synchroize

HOW 2026
Hello Open-source World

以开源之道·见致远之志
开源生态大会暨PostgreSQL高峰论坛

Thanks

谢谢