**概念:**

在wikipedia这样解读内存泄漏的：

* 在计算机科学中，内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

详情请看[wikipedia内存泄漏](https://zh.wikipedia.org/zh/%E5%86%85%E5%AD%98%E6%B3%84%E6%BC%8F)。

内存不是无穷无尽的，是有限的，如果申请了，使用了，用完没有释放，那么这块内存就一直无法被重新利用，最后再申请内存就找不到空闲的了（称为out of memory，**OOM**），可能导致程序逻辑错误崩溃。

以下用一个简单的例子说明内存泄漏：

#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char \*argv[])  
{  
 char \*p[1000];  
 int i;  
  
 for (i = 0; i < 1000; i++) {  
 p[i] = malloc(i \* 2 + 100);  
 if (p[i])  
 memset(p[i], 0, i \* 2 + 100);  
 }  
 for (i = 0; i < 1000; i++) {  
 printf("%p\n", p[i]);  
 }  
 return 0;  
}

上面的程序malloc申请的内存直至退出都没有被释放，这个就是内存泄漏了。

**分类**

内存泄漏可以根据发生的方式来分类：

* 常发性泄漏
  + 故障代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。
* 偶发性泄漏
  + 故障代码只有在特定场景或操作过程下才会被执行。常发性和偶发性是相对的。对于特定场景，偶发性的也许就变成了常发性的。所以测试场景和方法对检测内存泄漏至关重要。
* 一次性泄漏
  + 故障代码只会被执行一次或者由于逻辑的缺陷导致只有一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存。
* 隐式泄漏
  + 程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但对于daemon或需要长时间运行的程序来讲，不及时释放可能最终耗尽所有内存。

还可以根据泄漏类型来分类：

* 未引用泄漏
  + 这些泄漏点，已无任何指针指向，因此无法被找到，永远无法被释放。
  + native/kernel有这类问题，而java无此类问题，因为没被找到的，都可以gc。
* 引用泄漏
  + 可以被找到，等于隐式泄漏。引用泄漏像是缓存没有控制上限而导致耗尽内存。如果控制的好，则没问题，比如kernel里的slub也有per cpu缓存。
  + java也有这种问题存在。

另外不同的软件层都有可能存在内存泄漏：kernel层、native层和java层。不同的软件层泄漏的原理都差不多。

**危害**

内存泄漏是较难检测的异常之一，除了常发性泄漏，其他都是难以检查到的。从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为用户无法感觉到内存泄漏的存在，除非泄漏大量内存或一直累积直至消耗光内存，这时就有各种明显表现：

* 性能渐渐变差，因为要做各种内存回收工作。
* 直接出现逻辑错误崩溃，没有做好异常处理（error handling）。
* 本身没有问题，却引起其他程序异常。

既然内存泄漏这么难处理，那么是否有办法自动回收，不需要编程人员管理申请的内存呢？有的，java语言就有内存回收机制（垃圾回收，gc）。

不过即使用java编写的程序也可能出现泄漏，后面会讲解java内存泄漏的原理和调试方法。

**通用的调试方法**

需要对每一块申请的内存做标记，**记录**调用栈等信息，然后**监控**内存用量信息，感觉已经超出正常的内存用量很多时（相差不大时不见得能查到问题），**提取**这些信息，然后**分析**信息，找出可能的泄漏点，检查代码，分析逻辑，**修复**问题。

内存使用情况与监测

**(0). Android/Linux 内存分配的两个重要策略.**

Linux 在分配内存时, 为了节省内存, 按需分配, 使用了延时分配以及Copy-On-Write 的策略.

延时分配即针对user space 申请memory 时,  先只是明面上的分配虚拟空间, 等到真正操作memory 时, 才真正分配具体的物理内存,  这个需要借助MMU 的data abort 转换成page fault 来达成. 这样就可以极大的避免因user space 过度申请memory, 或者错误申请memory 造成的memory 浪费.

而Copy-On-Write 即是在进程fork 时,  子进程和父进程使用同一份memory, 只有当某块memory 被更新时, 才重新copy 出新的一份. 这个在android 上表现也非常显著,  上层app 包括system server 都由zygote fork 出来, 并且没重新exec 新的bin, ART VM/Lib 的memory 都是共享的, 可以极大的节省Memory 的使用.

对应的我们在评估一个进程的memory 使用时, 我们往往就需要观察它使用的虚拟的memory 空间, 它真实的使用的物理memory, 它和其他进程有均摊多少memory, 即:

* VSS- Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）
* RSS- Resident Set Size 实际使用物理内存（包含共享库占用的内存）
* PSS- Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）
* USS- Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）
* **(1). 内存的整体使用情况.**
* 要分析memory leaks, 你需要知道总体的内存使用情况和划分. 以判断内存泄露是发生在user space, kernel space, mulit-media 等使用的memory, 从而进一步去判断具体的memory leaks.
* user space 使用的memory 即通常包括从进程直接申请的memory, 比如 malloc: 先mmap/sbrk 整体申请大块Memory 后再malloc 细分使用, 比如stack memory, 直接通过mmap 从系统申请; 以及因user space 进程打开文件所使用的page cache, 以及使用ZRAM 压缩 user space memory 存储所占用的memory.  
    
  kernel space 使用的memory 通常包括 kernel stack, slub, page table, vmalloc, shmem 等.  
    
  mulit-media 使用的memory 通常使用的方式包括 ion, gpu 等.  
    
  其他方式的memory 使用, 此类一般直接从buddy system 中申请出以page 为单位的memory, android 中比较常见如ashmem.  
    
  而从进程的角度来讲, 通常情况下进程所使用的memory, 都会通过mmap 映射到进程空间后访问使用(注: 也会一些非常特别异常的流程, 没有mmap 到进程空间), 所以进程的memory maps 资讯是至关重要的. 对应在AEE DB 里面的file 是 PROCESS\_MAPS  
    
  下面枚举一些关键的段:
* b1100000-b1180000 rw-p 00000000 00:00 0                                  [anon:libc\_malloc]

malloc 通过jemalloc 所管控的空间, 常见的malloc leaks 都会可以看到这种libc\_malloc段空间显著增长.

address           perms offset  dev    inode               pathname  
aefe5000-af9fc000 r-xp 00000000 103:0a 25039               /data/app/in.startv.hotstar-c\_zk-AatlkkDg2B\_FSQFuQ==/lib/arm/libAVEAndroid.so  
af9fc000-afa3e000 r--p 00a16000 103:0a 25039               /data/app/in.startv.hotstar-c\_zk-AatlkkDg2B\_FSQFuQ==/lib/arm/libAVEAndroid.so  
afa3e000-afad2000 rw-p 00a58000 103:0a 25039               /data/app/in.startv.hotstar-c\_zk-AatlkkDg2B\_FSQFuQ==/lib/arm/libAVEAndroid.so

则是只读并可执行的主体代码段. 第二段 "r--p" 则是这个lib 使用的只读变量段 , 第三段 "rw-p" 则是这个lib 使用的数据段.

7110f000-71110000 rw-p 00000000 00:00 0                                 [anon:.bss]  
71712000-71713000 rw-p 00000000 00:00 0                                 [anon:.bss]  
71a49000-71a4a000 rw-p 00000000 00:00 0                                 [anon:.bss]

BSS(Block Started by Symbol) 段, 存放进程未初始化的static 以及 gloal 变量, 默认初始化时全部为0. 通常此类不会有memory leaks, 基本上长度在程序启动时就已经决定了.

//java thread  
6f5b0b2000-6f5b0b3000 ---p 00000000 00:00 0 [anon:thread stack guard]  
6f5b0b3000-6f5b0b4000 ---p 00000000 00:00 0   
6f5b0b4000-6f5b1b0000 rw-p 00000000 00:00 0   
  
//native thread  
74d0d0e000-74d0d0f000 ---p 00000000 00:00 0 [anon:thread stack guard]  
74d0d0f000-74d0e0c000 rw-p 00000000 00:00 0

pthread stack 使用memory, 注意目前pthread create 时只标注了它底部的 "thread stack guard", 默认pthread stack 大小是1M - 16K. guard 是 4K.  注意的是java thread 在art 里面还会再隔离一个page, 判断收到的SIGSEGV 是否为StackOverflowError.

7e9cf16000-7e9cf17000 ---p 00000000 00:00 0                              [anon:thread signal stack guard]  
7e9cf17000-7e9cf1b000 rw-p 00000000 00:00 0                              [anon:thread signal stack]

对应Pthread signal stack, 大小为16K,同样底部有guard 保护.

7f31245000-7f31246000 ---p 00000000 00:00 0                              [anon:bionic TLS guard]  
7f31246000-7f31249000 rw-p 00000000 00:00 0                              [anon:bionic TLS]

对应Pthread 的TLS, 长度为12K, 同样底部有guard 保护.

 edce5000-edce6000 rw-s 00000000 00:05 1510969      /dev/ashmem/shared\_memory/443BA81EE7976CA437BCBFF7935200B2 (deleted)

此类是ashmem, 访问/dev/ashmem 然后申请的memory, 通常比较关键是要确认它的name, 一般从它的name 可以明确得知memory 的申请位置. 至于 (deleted) 标识, 是指 mmap 时有带MAP\_FILE flag, 并且对应的path file已经unlink 或者不存在.

7e8d008000-7e8d306000 rw-s 00000000 00:0a 7438                           anon\_inode:dmabuf  
7e8d306000-7e8d604000 rw-s 00000000 00:0a 7438                           anon\_inode:dmabuf  
7e8d604000-7e8d902000 rw-s 00000000 00:0a 7438                           anon\_inode:dmabuf  
7e8d902000-7e8dc00000 rw-s 00000000 00:0a 7438                           anon\_inode:dmabuf

ion memory 段. ion buffer 的 vma name 标注成dmabuf, 即已经mmap 的ion memory 可以从这个直接统计算出.  
  
注意的是, maps 打印的资讯只是地址空间资讯, 即是虚拟地址空间占用情况, 而实际的具体的memory 占用多少需要审查 proc/pid/smaps.  
比如:

7e8ea00000-7e8ee00000 rw-p 00000000 00:00 0                              [anon:libc\_malloc]  
Name:           [anon:libc\_malloc]  
Size:               4096 kB  
Rss:                 888 kB  
Pss:                 888 kB  
Shared\_Clean:          0 kB  
Shared\_Dirty:          0 kB  
Private\_Clean:         0 kB  
Private\_Dirty:       888 kB  
Referenced:          888 kB  
Anonymous:           888 kB  
AnonHugePages:         0 kB  
ShmemPmdMapped:        0 kB  
Shared\_Hugetlb:        0 kB  
Private\_Hugetlb:       0 kB  
Swap:                  0 kB  
SwapPss:               0 kB  
KernelPageSize:        4 kB  
MMUPageSize:           4 kB  
Locked:                0 kB  
VmFlags: rd wr mr mw me nr

比如这段jemalloc 使用的memory, 对应是一个4M 大小, 但实际目前使用的RSS=PSS=888K, 即还有大部分没有真实填充memory.  
  
同样人为的查看maps 比较耗时, 目前在android 里面有 procrank, showmap, pmap 等命令可供查看.  
procrank 根据进程使用的memory 进行排序统计系统中进程的memory 使用量, 不过它一般没有统计ion 等资讯. 注意的是这个命令默认只编译到了debug 版本.

k71v1\_64\_bsp:/ # procrank -h  
Usage: procrank [ -W ] [ -v | -r | -p | -u | -s | -h ]  
    -v  Sort by VSS.  
    -r  Sort by RSS.  
    -p  Sort by PSS.  
    -u  Sort by USS.  
    -s  Sort by swap.  
        (Default sort order is PSS.)  
    -R  Reverse sort order (default is descending).  
    -c  Only show cached (storage backed) pages  
    -C  Only show non-cached (ram/swap backed) pages  
    -k  Only show pages collapsed by KSM  
    -w  Display statistics for working set only.  
    -W  Reset working set of all processes.  
    -o  Show and sort by oom score against lowmemorykiller thresholds.  
    -h  Display this help screen.

showmap 根据进程的maps/smaps 进行统计排序, 注意的是这个命令默认只编译到了debug 版本.

 k71v1\_64\_bsp:/ # showmap  
showmap [-t] [-v] [-c] [-q]   
        -t = terse (show only items with private pages)  
        -v = verbose (don't coalesce maps with the same name)  
        -a = addresses (show virtual memory map)  
        -q = quiet (don't show error if map could not be read)

pmap 把maps 的每个段打印出来, 如果使用-x 则会使用smaps 中数据匹配, 统计PSS, SWAP 等.

OP46E7:/ # pmap --help  
usage: pmap [-xq] [pids...]  
  
Reports the memory map of a process or processes.  
  
-x Show the extended format  
-q Do not display some header/footer lines

从系统角度来看memory 的使用情况, 通常会习惯性简单的查看 proc/meminfo:  
下面简单和大家分享具体的含义.

k71v1\_64\_bsp:/ # cat proc/meminfo  
MemTotal:        3849612 kB      
MemFree:          206920 kB   
MemAvailable:    1836292 kB   
Buffers:           73472 kB   
Cached:          1571552 kB   
SwapCached:        14740 kB  
Active:          1165488 kB  
Inactive:         865688 kB  
Active(anon):     202140 kB  
Inactive(anon):   195580 kB  
Active(file):     963348 kB  
Inactive(file):   670108 kB  
Unevictable:        5772 kB  
Mlocked:            5772 kB  
SwapTotal:       1048572 kB  
SwapFree:         787780 kB  
Dirty:                32 kB  
Writeback:             0 kB  
AnonPages:        383924 kB  
Mapped:           248488 kB  
Shmem:              6488 kB  
Slab:             391060 kB  
SReclaimable:     199712 kB  
SUnreclaim:       191348 kB  
KernelStack:       22640 kB  
PageTables:        28056 kB  
NFS\_Unstable:          0 kB  
Bounce:                0 kB  
WritebackTmp:          0 kB  
CommitLimit:     2973376 kB  
Committed\_AS:   42758232 kB  
VmallocTotal:   258867136 kB  
VmallocUsed:           0 kB  
VmallocChunk:          0 kB  
CmaTotal:        2093056 kB  
CmaFree:           78916 kB

我沿用kernel document 的注释:/kernel/Documentation/filesystems/proc.txt

MemTotal: Total usable ram (i.e. physical ram minus a few reserved  
              bits and the kernel binary code)  
     MemFree: The sum of LowFree+HighFree  
MemAvailable: An estimate of how much memory is available for starting new  
              applications, without swapping. Calculated from MemFree,  
              SReclaimable, the size of the file LRU lists, and the low  
              watermarks in each zone.  
              The estimate takes into account that the system needs some  
              page cache to function well, and that not all reclaimable  
              slab will be reclaimable, due to items being in use. The  
              impact of those factors will vary from system to system.  
     Buffers: Relatively temporary storage for raw disk blocks  
              shouldn't get tremendously large (20MB or so)  
      Cached: in-memory cache for files read from the disk (the  
              pagecache).  Doesn't include SwapCached  
  SwapCached: Memory that once was swapped out, is swapped back in but  
              still also is in the swapfile (if memory is needed it  
              doesn't need to be swapped out AGAIN because it is already  
              in the swapfile. This saves I/O)  
      Active: Memory that has been used more recently and usually not  
              reclaimed unless absolutely necessary.  
    Inactive: Memory which has been less recently used.  It is more  
              eligible to be reclaimed for other purposes  
   HighTotal:  
    HighFree: Highmem is all memory above ~860MB of physical memory  
              Highmem areas are for use by userspace programs, or  
              for the pagecache.  The kernel must use tricks to access  
              this memory, making it slower to access than lowmem.  
    LowTotal:  
     LowFree: Lowmem is memory which can be used for everything that  
              highmem can be used for, but it is also available for the  
              kernel's use for its own data structures.  Among many  
              other things, it is where everything from the Slab is  
              allocated.  Bad things happen when you're out of lowmem.  
   SwapTotal: total amount of swap space available  
    SwapFree: Memory which has been evicted from RAM, and is temporarily  
              on the disk  
       Dirty: Memory which is waiting to get written back to the disk  
   Writeback: Memory which is actively being written back to the disk  
   AnonPages: Non-file backed pages mapped into userspace page tables  
AnonHugePages: Non-file backed huge pages mapped into userspace page tables  
      Mapped: files which have been mmaped, such as libraries  
        Slab: in-kernel data structures cache  
SReclaimable: Part of Slab, that might be reclaimed, such as caches  
  SUnreclaim: Part of Slab, that cannot be reclaimed on memory pressure  
  PageTables: amount of memory dedicated to the lowest level of page  
              tables.  
NFS\_Unstable: NFS pages sent to the server, but not yet committed to stable  
          storage  
      Bounce: Memory used for block device "bounce buffers"  
WritebackTmp: Memory used by FUSE for temporary writeback buffers  
 CommitLimit: Based on the overcommit ratio ('vm.overcommit\_ratio'),  
              this is the total amount of  memory currently available to  
              be allocated on the system. This limit is only adhered to  
              if strict overcommit accounting is enabled (mode 2 in  
              'vm.overcommit\_memory').  
              The CommitLimit is calculated with the following formula:  
              CommitLimit = ([total RAM pages] - [total huge TLB pages]) \*  
                             overcommit\_ratio / 100 + [total swap pages]  
              For example, on a system with 1G of physical RAM and 7G  
              of swap with a `vm.overcommit\_ratio` of 30 it would  
              yield a CommitLimit of 7.3G.  
              For more details, see the memory overcommit documentation  
              in vm/overcommit-accounting.  
Committed\_AS: The amount of memory presently allocated on the system.  
              The committed memory is a sum of all of the memory which  
              has been allocated by processes, even if it has not been  
              "used" by them as of yet. A process which malloc()'s 1G  
              of memory, but only touches 300M of it will show up as  
          using 1G. This 1G is memory which has been "committed" to  
              by the VM and can be used at any time by the allocating  
              application. With strict overcommit enabled on the system  
              (mode 2 in 'vm.overcommit\_memory'),allocations which would  
              exceed the CommitLimit (detailed above) will not be permitted.  
              This is useful if one needs to guarantee that processes will  
              not fail due to lack of memory once that memory has been  
              successfully allocated.  
VmallocTotal: total size of vmalloc memory area  
 VmallocUsed: amount of vmalloc area which is used  
VmallocChunk: largest contiguous block of vmalloc area which is free

我们可以得到一些大体的"等式".

MemAvailable = free - kernel reserved memory + ative file + inactive file + SReclaimable - 2 \* zone low water mark  
Cached = All file page - buffers - swapping = Active file + Inactive file + Unevictable file - Buffers  
Slab = SReclaimable + SUnreclaimable  
Active = Active(anon) + Active(file)  
Inactive = Inactive(anon) + Inactive(file)  
AnonPages + Buffers + Cached = Active + Inactive  
Buffers + Cached = Active(file) + Inactive(file)  
SwapTotal = SwapFree + SwapUsed(Not SwapCached)  
KernelStack = the number of kernel task \* Stack Size(16K)  
Kernel Memory Usage = KernelStack + Slab + PageTables + Shmem + Vmalloc  
Native Memory Usage = Mapped + AnonPages + Others

**(2). Android dumpsys meminfo 解析.**

从Android 的角度, Google 提供了dumpsys meminfo 命令来获取全局以及某个进程的memory 信息.  
android 在 AativityManagerService 里面提供了一个meminfo 的service , 可以来抓取process 的memory 使用概要,  这个慢慢成为了android 上层判断的主流.

adb shell dumpsys meminfo   ==> dump 全局的memory 使用情况.  
adb shell dumpsys meminfo pid ==> dump 单个process memory 使用情况.  
它的一个好处在于, 如果是user build 没有root 权限的时候, 可以借道sh ==> system\_server ==> binder ==> process 进行抓取操作, 规避了权限方面的风险.

对应的完整操作命令:

 OP46E7:/ # dumpsys meminfo -h  
meminfo dump options: [-a] [-d] [-c] [-s] [--oom] [process]  
  -a: include all available information for each process.  
  -d: include dalvik details.  
  -c: dump in a compact machine-parseable representation.  
  -s: dump only summary of application memory usage.  
  -S: dump also SwapPss.  
  --oom: only show processes organized by oom adj.  
  --local: only collect details locally, don't call process.  
  --package: interpret process arg as package, dumping all  
             processes that have loaded that package.  
  --checkin: dump data for a checkin  
  --proto: dump data to proto  
If [process] is specified it can be the name or  
pid of a specific process to dump.

下面我们将解析dumpsys meminfo 的数据来源, 便于大家读懂.

**(2.1) 系统memory 来源解析.**

Total RAM: 3,849,612K (status moderate)  
 Free RAM: 1,870,085K (   74,389K cached pss + 1,599,904K cached kernel +   195,792K free)  
 Used RAM: 1,496,457K (  969,513K used pss +   526,944K kernel)  
 Lost RAM:   686,331K  
     ZRAM:    48,332K physical used for   260,604K in swap (1,048,572K total swap)  
   Tuning: 384 (large 512), oom   322,560K, restore limit   107,520K (high-end-gfx)

Total RAM:  /proc/meminfo-MemTotal  
Free RAM:  cached pss = All  pss of process oom\_score\_adj >= 900  
           cached kernel = /proc/meminfo.Buffers + /proc/meminfo.Cached + /proc/meminfo.SlabReclaimable- /proc/meminfo.Mapped  
           free = /proc/meminfo.MemFree   
Used RAM:  used pss: total pss - cached pss  
Kernel: /proc/meminfo.Shmem + /proc/meminfo.SlabUnreclaim + VmallocUsed + /proc/meminfo.PageTables + /proc/meminfo.KernelStack  
Lost RAM: /proc/meminfo.memtotal - (totalPss - totalSwapPss) -  /proc/meminfo.memfree - /proc/meminfo.cached - kernel used - zram used

**(2.2) 单个Process 数据源解析**

单个process 则通过binder 接入app 来抓取.  接到ActivityThread 的 dumpMeminfo 来统计.  
  
Native Heap, 从jemalloc 取出,对应实现是 android\_os\_Debug\_getNativeHeapSize() ==> mallinfo() ==> jemalloc  
  
Dalvik Heap, 使用Runtime 从java heap 取出.  
  
同时根据process 的smaps 解析数据 Pss  Private Dirty  Private Clean  SwapPss.

\*\* MEMINFO in pid 1138 [system] \*\*  
                   Pss  Private  Private  SwapPss     Heap     Heap     Heap  
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free  
                ------   ------   ------   ------   ------   ------   ------  
  Native Heap    62318    62256        0        0   137216    62748    74467  
  Dalvik Heap    21549    21512        0        0    28644    16356    12288  
 Dalvik Other     4387     4384        0        0  
        Stack       84       84        0        0  
       Ashmem      914      884        0        0  
    Other dev      105        0       56        0  
     .so mmap    10995     1112     4576        0  
    .apk mmap     3912        0     2776        0  
    .ttf mmap       20        0        0        0  
    .dex mmap    60297       76    57824        0  
    .oat mmap     2257        0       88        0  
    .art mmap     3220     2788       12        0  
   Other mmap     1944        4      672        0  
    GL mtrack     5338     5338        0        0  
      Unknown     3606     3604        0        0  
        TOTAL   180946   102042    66004        0   165860    79104    86755  
  
 App Summary  
                       Pss(KB)  
                        ------  
           Java Heap:    24312  
         Native Heap:    62256  
                Code:    66452  
               Stack:       84  
            Graphics:     5338  
       Private Other:     9604  
              System:    12900  
  
               TOTAL:   180946       TOTAL SWAP PSS:        0  
  
 Objects  
               Views:       11         ViewRootImpl:        2  
         AppContexts:       20           Activities:        0  
              Assets:       15        AssetManagers:        0  
       Local Binders:      528        Proxy Binders:     1134  
       Parcel memory:      351         Parcel count:      370  
    Death Recipients:      627      OpenSSL Sockets:        0  
            WebViews:        0  
  
 SQL  
         MEMORY\_USED:      384  
  PAGECACHE\_OVERFLOW:       86          MALLOC\_SIZE:      117  
  
 DATABASES  
      pgsz     dbsz   Lookaside(b)          cache  Dbname  
         4       64             85        12/29/8  /data/system\_de/0/accounts\_de.db  
         4       40                         0/0/0    (attached) ceDb: /data/system\_ce/0/accounts\_ce.db  
         4       20             27        54/17/3  /data/system/notification\_log.db

我给大家解释一下:

  Java Heap:    24312  dalvik heap + .art mmap  
         Native Heap:    62256      
                Code:    66452    .so mmap + .jar mmap + .apk mmap + .ttf mmap + .dex mmap + .oat mmap  
               Stack:       84  
            Graphics:     5338    Gfx dev + EGL mtrack + GL mtrack  
       Private Other:     9604  TotalPrivateClean + TotalPrivateDirty - java - native - code - stack - graphics  
              System:    12900    TotalPss - TotalPrivateClean - TotalPrivateDirty

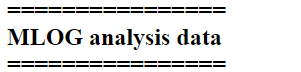
下面的解释来源于 https://developer.android.com/studio/profile/investigate-ram?hl=zh-cn

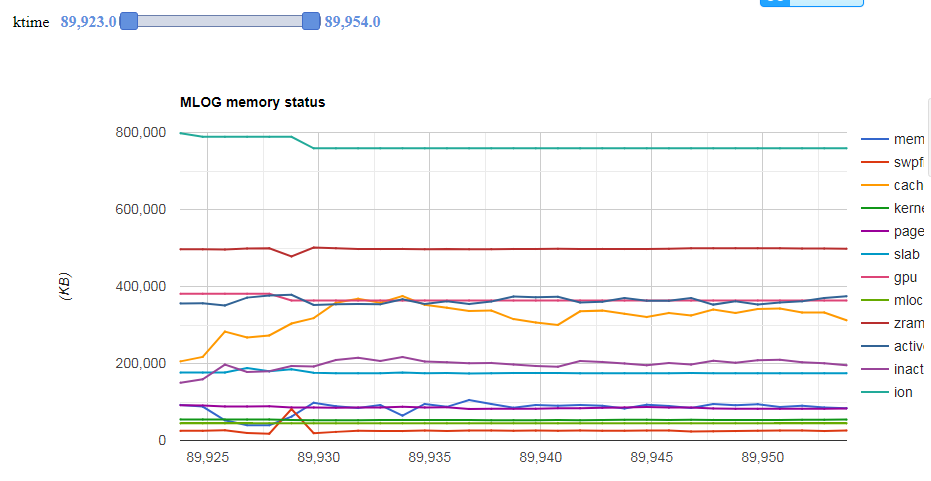
Dalvik Heap  
    您的应用中 Dalvik 分配占用的 RAM。Pss Total 包括所有 Zygote 分配（如上述 PSS 定义所述，通过进程之间的共享内存量来衡量）。Private Dirty 数值是仅分配到您应用的堆的实际 RAM，由您自己的分配和任何 Zygote 分配页组成，这些分配页自从 Zygote 派生应用进程以来已被修改。   
      
Heap Alloc 是 Dalvik 和原生堆分配器为您的应用跟踪的内存量。此值大于 Pss Total 和 Private Dirty，因为您的进程从 Zygote 派生，且包含您的进程与所有其他进程共享的分配。  
                
.so mmap 和 .dex mmap  
    映射的 .so（原生）和 .dex（Dalvik 或 ART）代码占用的 RAM。Pss Total 数值包括应用之间共享的平台代码；Private Clean 是您的应用自己的代码。通常情况下，实际映射的内存更大 - 此处的 RAM 仅为应用执行的代码当前所需的 RAM。不过，.so mmap 具有较大的私有脏 RAM，因为在加载到其最终地址时对原生代码进行了修改。   
.oat mmap  
    这是代码映像占用的 RAM 量，根据多个应用通常使用的预加载类计算。此映像在所有应用之间共享，不受特定应用影响。   
.art mmap  
    这是堆映像占用的 RAM 量，根据多个应用通常使用的预加载类计算。此映像在所有应用之间共享，不受特定应用影响。尽管 ART 映像包含 Object 实例，它仍然不会计入您的堆大小。

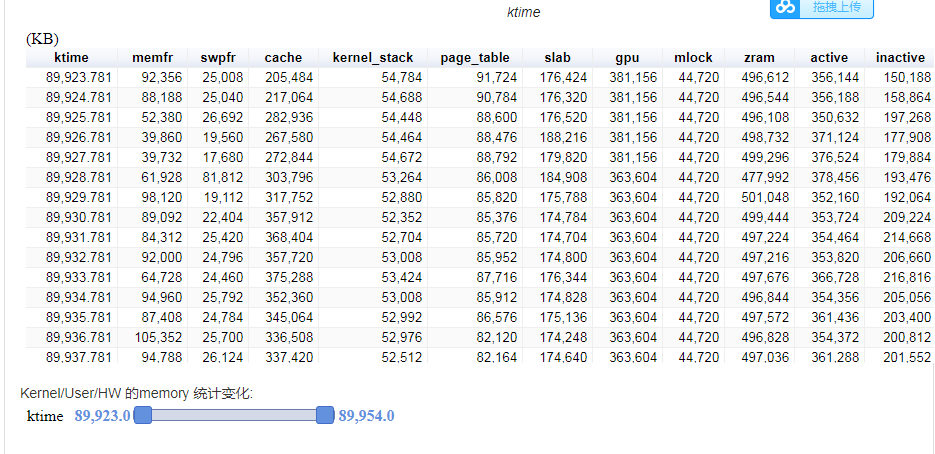
**(3). 内存使用情况监测**

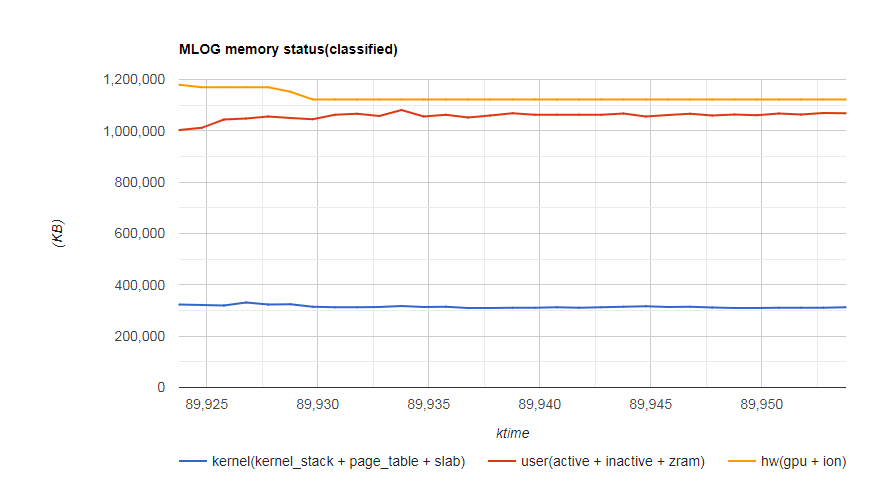
我们说通常的监测机制有两种. 一种是轮询, 周期性的查看memory 的使用情况, 通常是通过脚本或者daemon 程序周期性的监测. 监测的数据一般包括:  
/proc/meminfo 系统总的memory 使用情况.  
/proc/zoneinfo 每个zone 的memory 使用情况.  
/proc/buddyinfo  buddy system 的memory 情况.  
/proc/slabinfo  slub 的memory 使用分布.  
/proc/vmallocinfo vmalloc 的memory 使用情况.  
/proc/zraminfo zram 的使用情况, 以及占用memory 情况.  
/proc/mtk\_memcfg/slabtrace slab memory 的具体分布.  
/proc/vmstat 系统memory 根据使用类型的分布.  
/sys/kernel/debug/ion/ion\_mm\_heap mtk multi-media ion memory 使用情况.  
/sys/kernel/debug/ion/client\_history ion 各个clients 使用的ion 情况粗略统计.  
/proc/mali/memory\_usage  arm mali gpu 使用memory 按进程统计  
/sys/kernel/debug/mali0/gpu\_memory arm mali gpu 使用memory 按进程统计  
ps -A -T 打印系统所有进程/线程资讯, 可观察每个进程的线程量, 以及VSS/RSS  
dumpsys meminfo 从Android 角度观察系统memory 的使用情况.  
/sys/kernel/debug/mlog mtk 统计系统一段时间(约60s) 系统memory的使用情况, 包括kernel, user space, ion, gpu 等的分布.  
  
大家可以写脚本周期性的抓取.   
  
这里单独把mlog 抓出来说明, mlog 是MTK 开发的轻量级的memory log, 一体式抓取常见的memory 统计资讯, 包括kernel(vmalloc, slub...), user space (进程VSS,RSS...), ion, gpu 等在一段时间内部的概要使用情况. 并且提供了图形化的tool 来展示具体的memory 分布, 使用情况, 非常方便, 请大家优先使用(tool\_for\_memory\_analysis).  
  
下面是一些精美的截图:

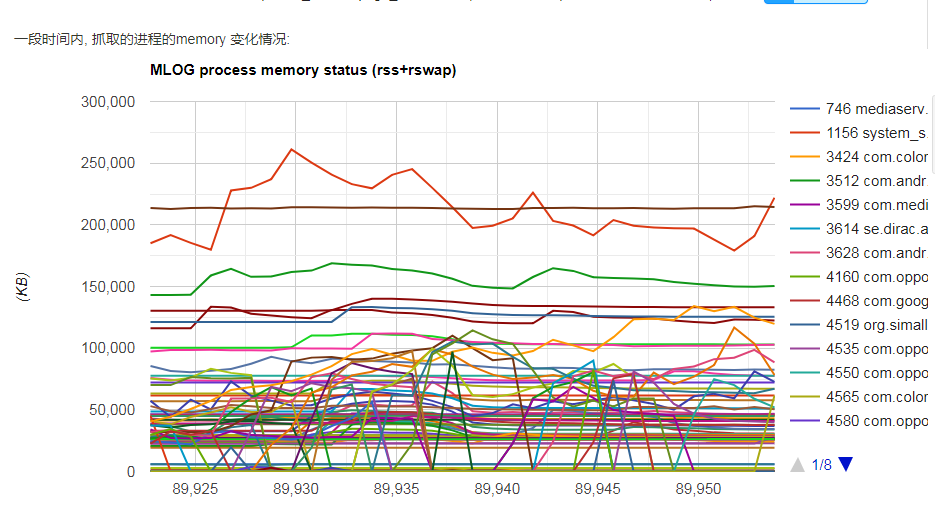
各类memory 在一段时间内的变化情况:











欢迎大家手工使用.

另外一种熔断, 即限制memory 的使用, 当到一定程度时, 主动发生异常, 回报错误.  
通常情况下, 系统memory leaks , 就会伴随OOM 发生, 严重是直接KE. 而单个进程 memory leaks, 如果它的oom adj < 0, 即daemon service 或者 persist app, 通常它的memory leaks 也会触发系统OOM , 因为lmk 难以杀掉. 如果是普通app 发生memory leaks, 则往往直接被LMK 杀掉. 难以对系统产生直接异常. 当然进程也可能无法申请到memory 发生JE, NE 等异常.  
  
针对总的系统的memory 使用, 我们可以通过设定, 限制系统总体的memory, 比如设置系统最大2GB:  
(1). ProjectConfig.mk  
CUSTOM\_CONFIG\_MAX\_DRAM\_SIZE = 0x80000000  
注意: CUSTOM\_CONFIG\_MAX\_DRAM\_SIZE must be included by AUTO\_ADD\_GLOBAL\_DEFINE\_BY\_NAME\_VALUE  
  
(2). preloader project config file  
vendor/mediatek/proprietary/bootable/bootloader/preloader/custom/{project}/{project}.mk  
CUSTOM\_CONFIG\_MAX\_DRAM\_SIZE = 0x80000000  
注意: CUSTOM\_CONFIG\_MAX\_DRAM\_SIZE must be exported  
  
针对某个进程使用的memory, 我们可以通过setrlimit 来进行限制, 如: 针对camerahalserver, 使用init 的setrlimit 进行限制.

service camerahalserver /vendor/bin/hw/camerahalserver  
    class main  
    user cameraserver  
    group audio camera input drmrpc sdcard\_rw system media graphics  
    ioprio rt 4  
    capabilities SYS\_NICE  
    writepid /dev/cpuset/camera-daemon/tasks /dev/stune/top-app/tasks  
    #limit VSS to 4GB  
    rlimit as 0x100000000 0x100000000  
    #limit malloc to 1GB  
    rlimit data 0x40000000 0x40000000

把camerahalserver 的VSS 限制到4GB,  把malloc 的大小限制到1GB,  一旦超出就会返回 ENOMEM,  通常情况下，这样可自动产生NE.  以便抓到camerahalserver 的更多信息.  
  
注意的是因为vendor 下面的service 是由 vendor\_init 拉起来的,  需要给vendor\_init 设置sepolicy. 以免无法设定成功.

 /device/mediatek/sepolicy/basic/non\_plat/vendor\_init.te  
   
allow vendor\_init self:global\_capability\_class\_set sys\_resource;

也可以直接在代码里面写死, 参考如/frameworks/av/media/libmedia/MediaUtils.cpp  
  
针对APP java heap的memory leaks, 我们可以通过设定 dalvik 的heap size 进行限制, 通过system property 设定, 注意的是, 目前的做法会影响到所有的java process.

[dalvik.vm.heapgrowthlimit]: [384m]  
[dalvik.vm.heapsize]: [512m]