OOM与内存优化

内存管理基础

App内存组成以及限制

`Android`给每个`App`分配一个`VM`，让App运行在`dalvik`上，这样即使`App`崩溃也不会影响到系统。系统给`VM`分配了一定的内存大小，`App`可以申请使用的内存大小不能超过此硬性逻辑限制，就算物理内存富余，如果应用超出`VM`最大内存，就会出现内存溢出`crash`。

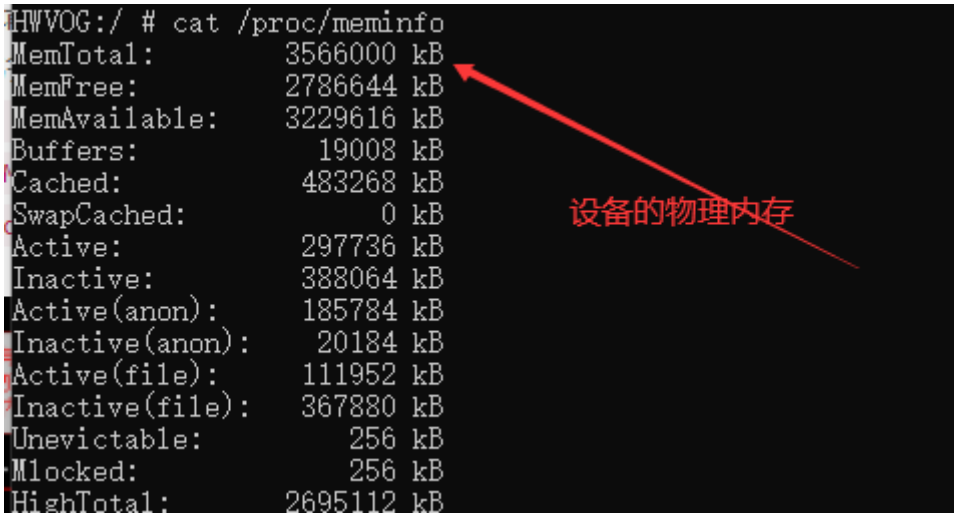
由程序控制操作的内存空间在`heap`上，分`java heapsize`和`native heapsize`

- Java申请的内存在`vm heap`上，所以如果`java`申请的内存大小超过`VM`的逻辑内存限制,就会出现内存溢出的异常。
- **native层内存申请不受其限制**,`native`层受`native process`对内存大小的限制

如何查看Android设备对App的内存限制

1. 主要查看系统配置文件`build.prop`，我们可以通过`adb shell`在 命令行窗口查看

`adb shell cat /system/build.prop`



2. 通过代码获取

```
ActivityManager activityManager =
(ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE)
activityManager.getMemoryClass();//以m为单位
```

3. 可以修改吗?

- 修改 \frameworks\base\core\jni\AndroidRuntime.cpp

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv, bool zygote)
  {
  /*

   * The default starting and maximum size of the heap.  Larger
   * values should be specified in a product property override.
     */
       parseRuntimeOption("dalvik.vm.heapstartsize", heapstartsizeOptsBuf,
"-Xms", "4m");
       parseRuntimeOption("dalvik.vm.heapsize", heapsizeOptsBuf, "-Xmx",
"16m");//修改这里
     * }
```

- 修改 platform/dalvik/+/eclair-release/vm/Init.c

```
gDvm.heapSizeStart = 2 * 1024 * 1024;   // Spec says 16MB; too big for
us.
gDvm.heapSizeMax = 16 * 1024 * 1024;    // Spec says 75% physical mem
```

内存指标概念

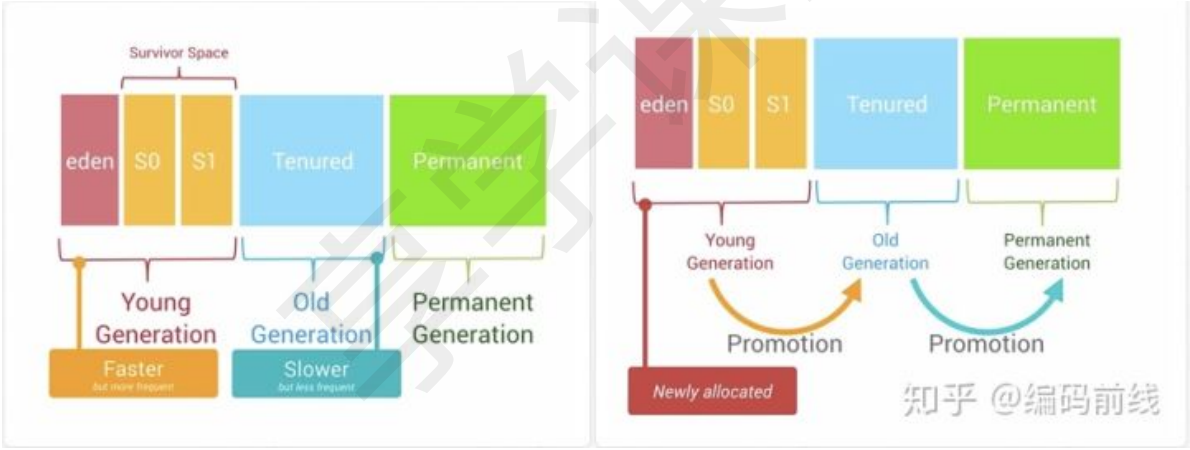
Item	全称	含义	等价
USS	Unique Set Size	物理内存	进程独占的内存
PSS	Proportional Set Size	物理内存	PSS= USS+ 按比例包含共享库
RSS	Resident Set Size	物理内存	RSS= USS+ 包含共享库
VSS	Virtual Set Size	虚拟内存	VSS= RSS+ 未分配实际物理内存

总结:VSS >= RSS >= PSS >= USS,但/dev/kgsl-3d0部份必须考虑VSS

Android内存分配与回收机制

- 内存分配

Android的Heap空间是一个`Generational Heap Memory`的模型，最近分配的对象会存放在`Young Generation`区域，当一个对象在这个区域停留的时间达到一定程度，它会被移动到`Old Generation`，最后累积一定时间再移动到`Permanent Generation`区域。



1、Young Generation

由一个Eden区和两个Survivor区组成，程序中生成的大部分新的对象都在Eden区中，当Eden区满时，还存活的对象将被复制到其中一个Survivor区，当次Survivor区满时，此区存活的对象又被复制到另一个Survivor区，当这个Survivor区也满时，会将其中存活的对象复制到年老代。

2、Old Generation

一般情况下，年老代中的对象生命周期都比较长。

3、Permanent Generation

用于存放静态的类和方法，持久代对垃圾回收没有显著影响。

总结：内存对象的处理过程如下：

- 1、对象创建后在Eden区。
- 2、执行GC后，如果对象仍然存活，则复制到S0区。
- 3、当S0区满时，该区域存活对象将复制到S1区，然后S0清空，接下来S0和S1角色互换。

- 4、当第3步达到一定次数（系统版本不同会有差异）后，存活对象将被复制到Old Generation。
- 5、当这个对象在Old Generation区域停留的时间达到一定程度时，它会被移动到Old Generation，最后累积一定时间再移动到Permanent Generation区域。

系统在Young Generation、Old Generation上采用不同的回收机制。每一个Generation的内存区域都有固定的大小。随着新的对象陆续被分配到此区域，当对象总的大小临近这一级别内存区域的阈值时，会触发GC操作，以便腾出空间来存放其他新的对象。

执行GC占用的时间与Generation和Generation中的对象数量有关：

- Young Generation < Old Generation < Permanent Generation
- Gener中的对象数量与执行时间成正比。

4、Young Generation GC

由于其对象存活时间短，因此基于Copying算法（扫描出存活的对象，并复制到一块新的完全未使用的控件中）来回收。新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在Young Generation区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。

5、Old Generation GC

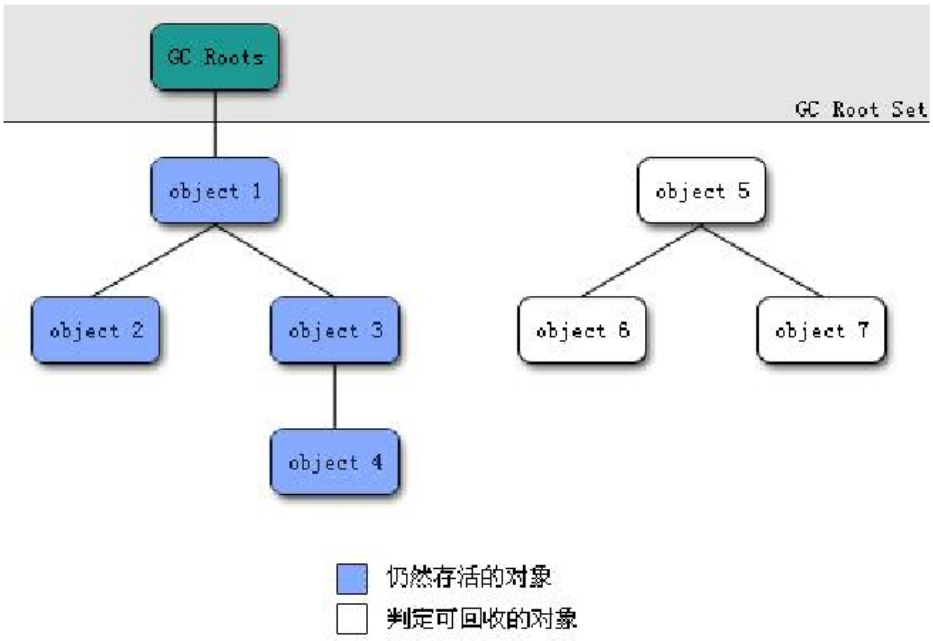
由于其对象存活时间较长，比较稳定，因此采用Mark（标记）算法（扫描出存活的对象，然后再回收未被标记的对象，回收后对空出的空间要么合并，要么标记出来便于下次分配，以减少内存碎片带来的效率损耗）来回收。

GC类型

在Android系统中，GC有三种类型：

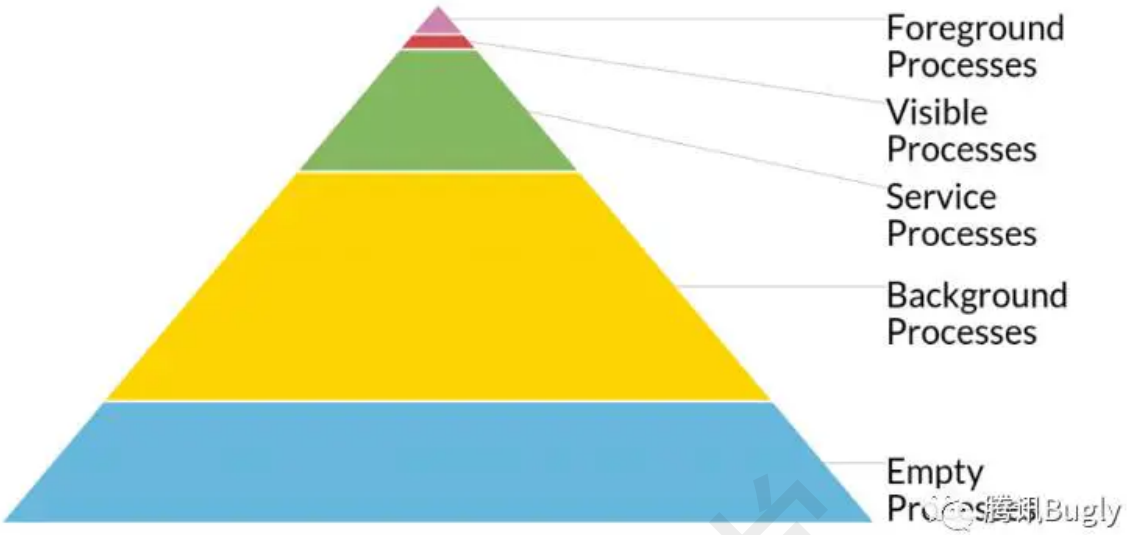
- kGcCauseForAlloc：分配内存不够引起的GC，会Stop World。由于是并发GC，其它线程都会停止，直到GC完成。
- kGcCauseBackground：内存达到一定阈值触发的GC，由于是一个后台GC，所以不会引起Stop World。
- kGcCauseExplicit：显示调用时进行的GC，当ART打开这个选项时，使用System.gc时会进行GC。

可达性分析与GCRoots



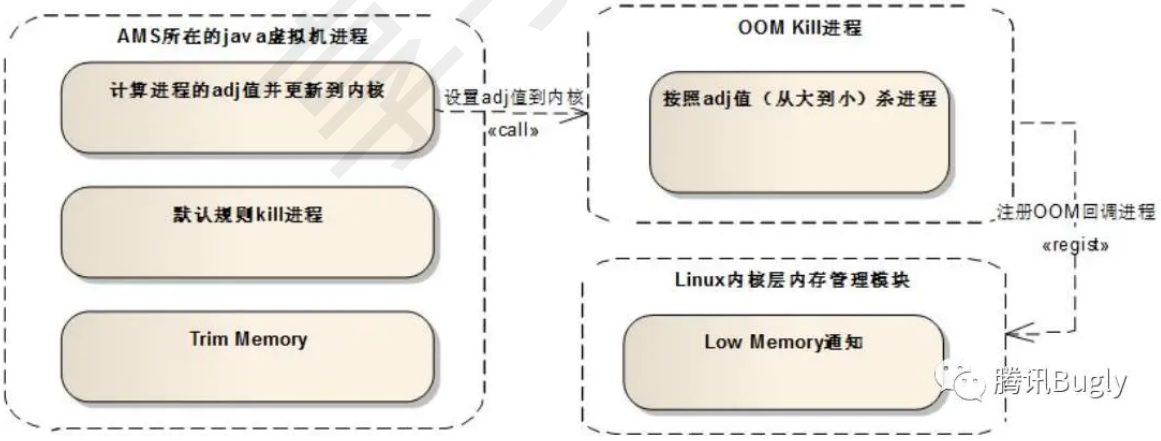
Android低内存杀进程机制

Anroid基于进程中运行的组件及其状态规定了默认的五个回收优先级：



- Empty process(空进程)
- Background process(后台进程)
- Service process(服务进程)
- Visible process(可见进程)
- Foreground process(前台进程)

系统需要进行内存回收时最先回收空进程,然后是后台进程，以此类推最后才会回收前台进程（一般情况下前台进程就是与用户交互的进程了,如果连前台进程都需要回收那么此时系统几乎不可用了）。



`ActivityManagerService`会对所有进程进行评分（存放在变量adj中），然后再讲这个评分更新到内核，由内核去完成真正的内存回收(`lowmemorykiller`, `Oom_killer`)。这里只是大概的流程，中间过程还是很复杂的

什么是OOM

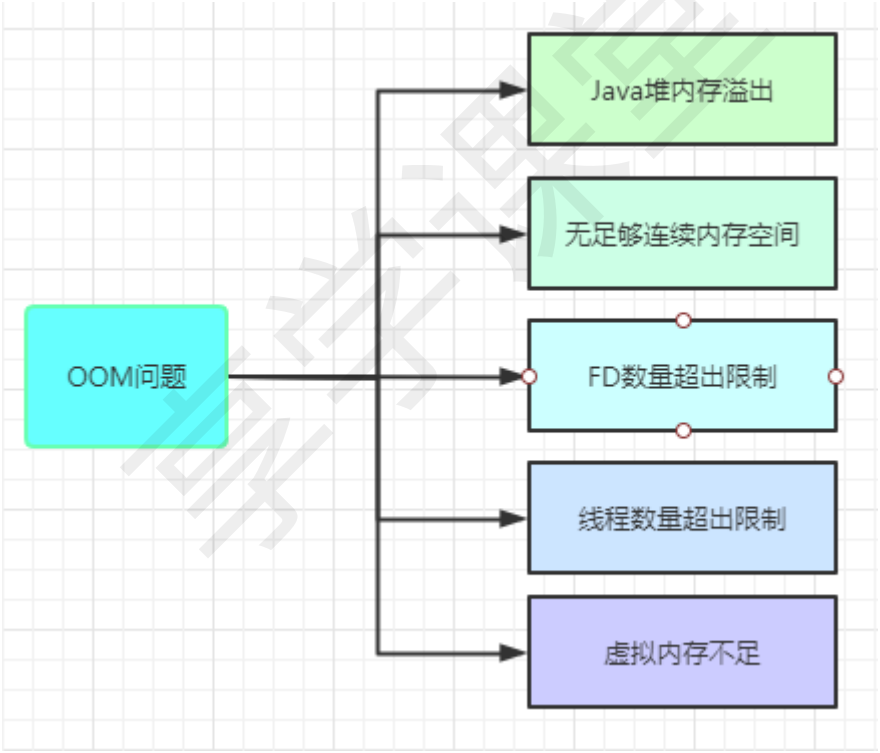
OOM（OutOfMemoryError）内存溢出错误，在常见的Crash疑难排行榜上，OOM绝对可以名列前茅并且经久不衰。因为它发生时的Crash堆栈信息往往不是导致问题的根本原因，而只是压死骆驼的最后一根稻草

```
Process: com.enjoy.memory, PID: 6472
java.lang.OutOfMemoryError: Failed to allocate a 2147483659 byte allocation with 4194304 free bytes and 247MB until OOM
    at com.enjoy.memory.MainActivity.onCreate(MainActivity.java:15)
    at android.app.Activity.performCreate(Activity.java:6100)
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1112)
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2481)
    at android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2614)
    at android.app.ActivityThread.access$800(ActivityThread.java:178)
    at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1470)
    at android.os.Handler.dispatchMessage(Handler.java:111)
    at android.os.Looper.loop(Looper.java:194)
    at android.app.ActivityThread.main(ActivityThread.java:5643) <2 internal calls>
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:960)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:755)
```

发生OOM的条件

- Android 2.x系统 GC LOG中的dalvik allocated + external allocated + 新分配的大小 >= getMemoryClass()值的时候就会发生OOM。 例如，假设有这么一段Dalvik输出的GC LOG：GC_FOR_MALLOC free 2K, 13% free 32586K/37455K, external 8989K/10356K, paused 20ms，那么32586+8989+(新分配23975)=65550>64M时，就会发生OOM。
- Android 4.x系统 Android 4.x的系统废除了external的计数器，类似bitmap的分配改到dalvik的java heap中申请，只要allocated + 新分配的内存 >= getMemoryClass()的时候就会发生OOM

OOM原因分类



OOM代码分析

Android 虚拟机最终抛出OutOfMemoryError的地方

> /art/runtime/thread.cc

```
void Thread::ThrowOutOfMemoryError(const char* msg) {
  LOG(WARNING) << StringPrintf("Throwing OutOfMemoryError \"%s\"%s",
      msg, (tls32_.throwing_OutOfMemoryError ? " (recursive case)" : ""));
  if (!tls32_.throwing_OutOfMemoryError) {
    tls32_.throwing_OutOfMemoryError = true;
    ThrowNewException("Ljava/lang/OutOfMemoryError;", msg);
    tls32_.throwing_OutOfMemoryError = false;
  } else {
    Dump(LOG_STREAM(WARNING));  // The pre-allocated OOME has no stack, so help out and log one.
    SetException(Runtime::Current()->GetPreAllocatedOutOfMemoryError());
  }
}
```

堆内存分配失败

> /art/runtime/gc/heap.cc

```
void Heap::ThrowOutOfMemoryError(Thread* self, size_t byte_count, AllocatorType allocator_type) {
  // If we're in a stack overflow, do not create a new exception. It would require running the
  // constructor, which will of course still be in a stack overflow.
  if (self->IsHandlingStackOverflow()) {
    self->SetException(
        Runtime::Current()->GetPreAllocatedOutOfMemoryErrorWhenHandlingStackOverflow());
    return;
  }

  std::ostringstream oss;
  size_t total_bytes_free = GetFreeMemory();
    //为对象分配内存时达到进程的内存上限
  oss << "Failed to allocate a " << byte_count << " byte allocation with " << total_bytes_free
      << " free bytes and " << PrettySize(GetFreeMemoryUntilOOME()) << " until OOM,"
      << " target footprint " << target_footprint_.load(std::memory_order_relaxed)
      << ", growth limit "
      << growth_limit_;

    //没有足够大小的连续地址空间
 // There is no fragmentation info to log for large-object space.
    if (allocator_type != kAllocatorTypeLOS) {
      CHECK(space != nullptr) << "allocator_type:" << allocator_type
                              << " byte_count:" << byte_count
                              << " total_bytes_free:" << total_bytes_free;
      space->LogFragmentationAllocFailure(oss, byte_count);
    }
  }
```

创建线程失败

> /art/runtime/thread.cc

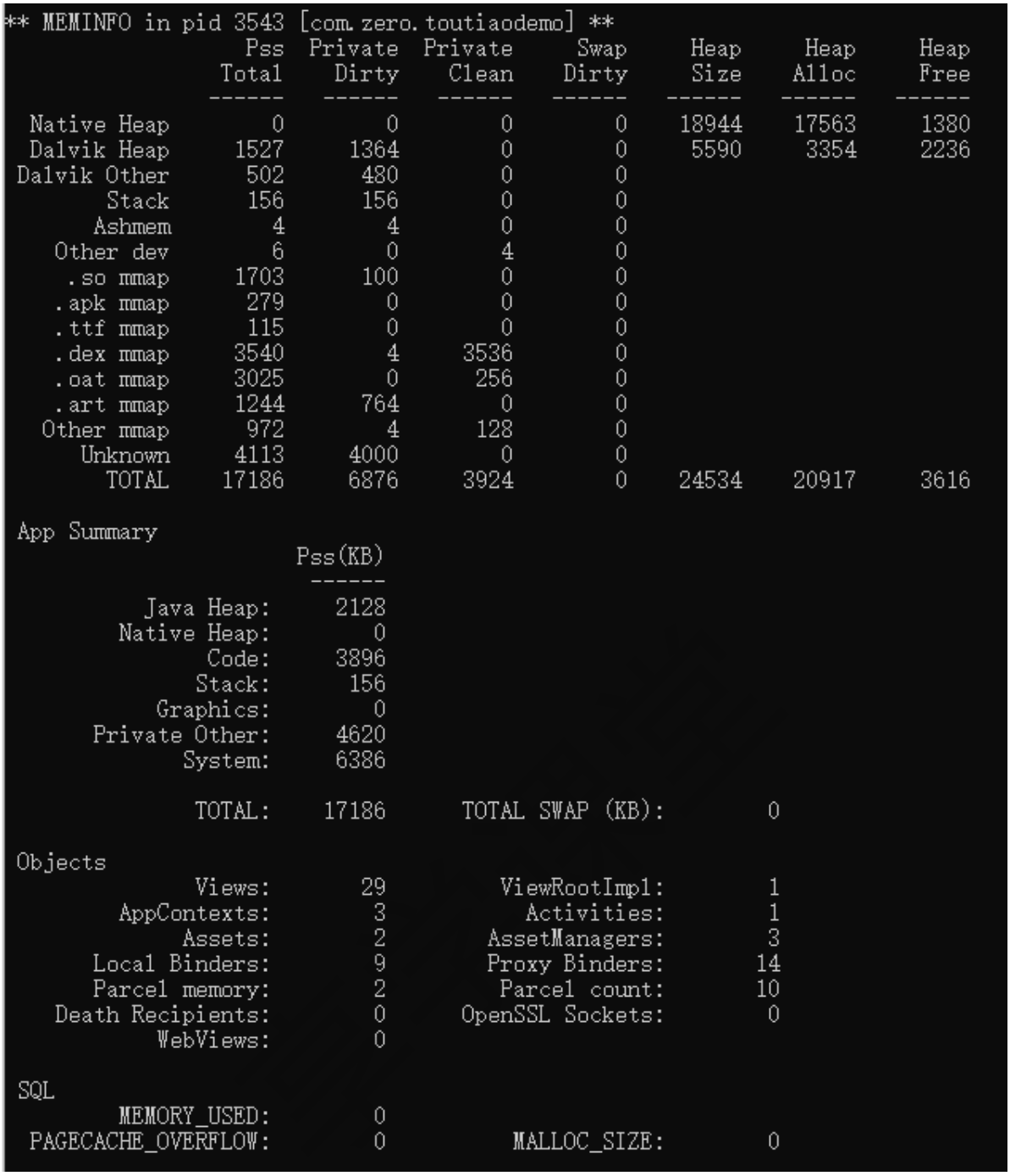
```cpp
void Thread::CreateNativeThread(JNIEnv* env, jobject java_peer, size_t stack_size, bool is_daemon) {
  CHECK(java_peer != nullptr);
  Thread* self = static_cast<JNIEnvExt*>(env)->GetSelf();

 // TODO: remove from thread group?
  env->SetLongField(java_peer, WellKnownClasses::java_lang_Thread_nativePeer, 0);
  {
    std::string msg(child_jni_env_ext.get() == nullptr ?
        StringPrintf("Could not allocate JNI Env: %s", error_msg.c_str()) :
        StringPrintf("pthread_create (%s stack) failed: %s",
                                 PrettySize(stack_size).c_str(), strerror(pthread_create_result)));
    ScopedObjectAccess soa(env);
    soa.Self()->ThrowOutOfMemoryError(msg.c_str());
  }
```

Android 内存分析命令介绍

常用的内存调优分析命令：

1. dumpsys meminfo
2. procrank
3. cat /proc/meminfo
4. free
5. showmap
6. vmstat

dumpsys meminfo



```
** MEMINFO in pid 3543 [com.zero.toutiaodemo] **
                   Pss  Private  Private     Swap     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap        0        0        0        0    18944    17563     1380
  Dalvik Heap     1527     1364        0        0     5590     3354     2236
 Dalvik Other      502      480        0        0
        Stack      156      156        0        0
      Ashmem         4        4        0        0
   Other dev         6        0        4        0
    .so mmap      1703      100        0        0
   .apk mmap       279        0        0        0
   .ttf mmap       115        0        0        0
   .dex mmap      3540        4     3536        0
   .oat mmap      3025        0      256        0
   .art mmap      1244      764        0        0
  Other mmap       972        4      128        0
     Unknown      4113     4000        0        0
       TOTAL     17186     6876     3924        0    24534    20917     3616

 App Summary
                       Pss(KB)
                        ------
           Java Heap:     2128
         Native Heap:        0
                Code:     3896
               Stack:      156
            Graphics:        0
       Private Other:     4620
              System:     6386

               TOTAL:    17186      TOTAL SWAP (KB):        0

 Objects
             Views:       29         ViewRootImpl:        1
       AppContexts:        3           Activities:        1
            Assets:        2        AssetManagers:        3
      Local Binders:        9        Proxy Binders:       14
      Parcel memory:        2         Parcel count:       10
   Death Recipients:        0      OpenSSL Sockets:        0
           WebViews:        0

 SQL
         MEMORY_USED:        0
  PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

相关参数的说明：

Pss Total：是一个进程实际使用的内存，该统计方法包括比例分配共享库占用的内存，即如果有三个进程共享了一个共享库，则平摊分配该共享库占用的内存。Pss Total统计方法的一个需要注意的地方是如果使用共享库的一个进程被杀死，则共享库的内存占用按比例分配到其他共享该库的进程中，而不是将内存资源返回给系统，这种情况下PssTotal不能够准确代表内存返回给系统的情况。

Private Dirty：进程私有的脏页内存大小，该统计方法只包括进程私有的被修改的内存。

Private Clear：进程私有的干净页内存大小，该统计方法只包括进程私有的没有被修改的内存。

Swapped Dirty：被交换的脏页内存大小，该内存与其他进程共享。

其中private Dirty + private Clean = Uss，该值是一个进程的使用的私有内存大小，即这些内存唯一被该进程所有。该统计方法真正描述了运行一个进程需要的内存和杀死一个进程释放的内存情况，是怀疑内存泄露最好的统计方法。

共享比例：sharing_proportion = (Pss Total - private_clean - private_dirty) / (shared_clean + shared_dirty)

能够被共享的内存：swappable_pss = (sharing_proportion * shared_clean) + private_clean

Native Heap：本地堆使用的内存，包括C/C++在堆上分配的内存

Dalvik Heap：dalvik虚拟机使用的内存

Dalvik other：除Dalvik和Native之外分配的内存，包括C/C++分配的非堆内存

Cursor：数据库游标文件占用的内存

Ashmem：匿名共享内存

Stack：Dalvik栈占用的内存

Other dev：其他的dev占用的内存

.so mmap：so库占用的内存

.jar mmap：.jar文件占用的内存

.apk mmap：.apk文件占用的内存

.ttf mmap：.ttf文件占用的内存

.dex mmap：.dex文件占用的内存

image mmap：图像文件占用的内存

code mmap：代码文件占用的内存

Other mmap：其他文件占用的内存

Graphics：GPU使用图像时使用的内存

GL：GPU使用GL绘制时使用的内存

Memtrack：GPU使用多媒体、照相机时使用的内存

Unknown：不知道的内存消耗

Heap Size：堆的总内存大小

Heap Alloc：堆分配的内存大小

Heap Free：堆待分配的内存大小

Native Heap | Heap Size : 从mallinfo usmblks获的，当前进程Native堆的最大总共分配内存

Native Heap | Heap Alloc : 从mallinfo uorblks获的，当前进程navtive堆的总共分配内存

Native Heap | Heap Free : 从mallinfo fordblks获的，当前进程Native堆的剩余内存

Native Heap Size ≈ Native Heap Alloc + Native Heap Free

mallinfo是一个C库，mallinfo()函数提供了各种各样通过malloc()函数分配的内存的统计信息。

Dalvik Heap | Heap Size : 从Runtime totalMemory()获得，Dalvik Heap总共的内存大小

Dalvik Heap | Heap Alloc : 从Runtime totalMemory() - freeMemory()获得，Dalvik Heap分配的内存大小

Dalvik Heap | Heap Free : 从Runtime freeMemory()获得，Dalvik Heap剩余的内存大小

Dalvik Heap Size = Dalvik Heap Alloc + Dalvik Heap Free

Obejcts当前进程中的对象个数

Views:当前进程中实例化的视图View对象数量

ViewRootImpl:当前进程中实例化的视图根ViewRootImpl对象数量

AppContexts:当前进程中实例化的应用上下文ContextImpl对象数量

Activities:当前进程中实例化的Activity对象数量

Assets:当前进程的全局资产数量

AssetManagers:当前进程的全局资产管理数量

Local Binders:当前进程有效的本地binder对象数量

Proxy Binders:当前进程中引用的远程binder对象数量

Death Recipients:当前进程到binder的无效链接数量

OpenSSL Sockets:安全套接字对象数量

SQL

MEMORY_USED:当前进程中数据库使用的内存数量，kb

PAGECACHE_OVERFLOW:页面缓存的配置不能够满足的数量，kb

MALLOC_SIZE: 向sqlite3请求的最大内存分配数量，kb

DATABASES

pgsz:数据库的页面大小

dbsz:数据库大小

Lookaside(b):后备使用的内存大小

cache:数据缓存状态

Dbname:数据库表名

Asset Allocations

资源路径和资源大小

procrank

功能： 获取所有进程的内存使用的排行榜，排行是以`Pss`的大小而排序。`procrank`命令比`dumpsys meminfo`命令，能输出更详细的VSS/RSS/PSS/USS内存指标。

最后一行输出下面6个指标：

total	free	buffers	cached	shmem	slab
2857032K	998088K	78060K	78060K	312K	92392K

执行结果：

```
root@Phone:/# procrank
  PID       Vss      Rss      Pss      Uss  cmdline
 4395  2270020K  202312K  136099K  121964K  com.android.systemui
 1192  2280404K  147048K   89883K   84144K  system_server
29256  2145676K   97880K   44328K   40676K  com.android.settings
  501  1458332K   61876K   23609K    9736K  zygote
 4239  2105784K   68056K   21665K   19592K  com.android.phone
  479   164392K   24068K   17970K   15364K  /system/bin/mediaserver
  391   200892K   27272K   15930K   11664K  /system/bin/surfaceflinger
...
RAM: 2857032K total, 998088K free, 78060K buffers, c cached, 312K shmem, 92392K slab
```

cat /proc/meminfo

功能：能否查看更加详细的内存信息

```
指令： cat /proc/meminfo
```

输出结果如下(结果内存值不带小数点，此处添加小数点的目的是为了便于比对大小)：

```
root@phone:/ # cat /proc/meminfo
MemTotal:        2857.032 kB  //RAM可用的总大小 (即物理总内存减去系统预留和内核二进制代码大小)
MemFree:         1020.708 kB  //RAM未使用的大小
Buffers:           75.104 kB  //用于文件缓冲
Cached:           448.244 kB  //用于高速缓存
SwapCached:             0 kB  //用于swap缓存

Active:           832.900 kB  //活跃使用状态，记录最近使用过的内存，通常不回收用于其它目的
Inactive:         391.128 kB  //非活跃使用状态，记录最近并没有使用过的内存，能够被回收用于其他目的
Active(anon):     700.744 kB  //Active = Active(anon) + Active(file)
Inactive(anon):       228 kB  //Inactive = Inactive(anon) + Inactive(file)
Active(file):     132.156 kB
Inactive(file):   390.900 kB

Unevictable:            0 kB
Mlocked:                0 kB

SwapTotal:        524.284 kB  //swap总大小
SwapFree:         524.284 kB  //swap可用大小
Dirty:                  0 kB  //等待往磁盘回写的大小
Writeback:              0 kB  //正在往磁盘回写的大小

AnonPages:        700.700 kB  //匿名页，用户空间的页表，没有对应的文件
Mapped:           187.096 kB  //文件通过mmap分配的内存，用于map设备、文件或者库
Shmem:               .312 kB

Slab:              91.276 kB  //kernel数据结构的缓存大小，Slab=SReclaimable+SUnreclaim
SReclaimable:      32.484 kB  //可回收的slab的大小
SUnreclaim:        58.792 kB  //不可回收slab的大小

KernelStack:       25.024 kB
```

```
PageTables:         23.752 kB  //以最低的页表级
NFS_Unstable:           0 kB  //不稳定页表的大小
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:      1952.800 kB
Committed_AS:    82204.348 kB   //评估完成的工作量，代表最糟糕case下的值，该值也包含swap内存

VmallocTotal:  251658.176 kB  //总分配的虚拟地址空间
VmallocUsed:      166.648 kB  //已使用的虚拟地址空间
VmallocChunk:  251398.700 kB  //虚拟地址空间可用的最大连续内存块
```

对于cache和buffer也是系统可以使用的内存。所以系统总的可用内存为 MemFree+Buffers+Cached

free

主功能：查看可用内存，缺省单位KB。该命令比较简单、轻量，专注于查看剩余内存情况。数据来源于/proc/meminfo。

输出结果：

```
root@phone:/proc/sys/vm # free
             total         used         free       shared      buffers
Mem:       2857032      1836040      1020992            0        75104
-/+ buffers:            1760936      1096096
Swap:       524284            0       524284
```

- 对于`Mem`行，存在的公式关系： total = used + free;
- 对于`-/+ buffers`行： 1760936 = 1836040 - 75104(buffers); 1096096 = 1020992 + 75104(buffers);

showmap

主功能：用于查看虚拟地址区域的内存情况

```
用法：  showmap -a [pid]
```

该命令的输出每一行代表一个虚拟地址区域(vm area)

```
HWVOG:/ # showmap -a 3543
   start      end  virtual                 shared   shared  private  private
    addr     addr     size      RSS      PSS   clean    dirty    clean    dirty     swap object
-------- -------- -------- -------- -------- -------- -------- -------- -------- -------- ------------------------------
12c00000 12d2e000     1208      988      988        0        0        0      988        0 /dev/ashmem/dalvik-main space (deleted)
12d2e000 12e16000      928        0        0        0        0        0        0        0 /dev/ashmem/dalvik-main space (deleted)
```

- start addr和end addr:分别代表进程空间的起止虚拟地址；
- virtual size/ RSS /PSS这些前面介绍过；
- shared clean：代表多个进程的虚拟地址可指向这块物理空间，即有多少个进程共享这个库；
- shared: 共享数据
- private: 该进程私有数据
- clean: 干净数据，是指该内存数据与disk数据一致，当内存紧张时，可直接释放内存，不需要回写到disk
- dirty: 脏数据，与disk数据不一致，需要先回写到disk，才能被释放。

vmstat

主功能：不仅可以查看内存情况，还可以查看进程运行队列、系统切换、CPU时间占比等情况，另外该指令还是周期性地动态输出。

用法：

```
Usage: vmstat [ -n iterations ] [ -d delay ] [ -r header_repeat ]
    -n iterations     数据循环输出的次数
    -d delay          两次数据间的延迟时长(单位：S)
    -r header_repeat  循环多少次，再输出一次头信息行
```

输入结果：

```
root@phone:/ # vmstat
procs  memory                         system          cpu
 r  b   free  mapped   anon   slab    in   cs  flt  us ni sy id wa ir
 2  0  663436 232836 915192 113960   196  274    0   8  0  2 99  0  0
 0  0  663444 232836 915108 113960   180  260    0   7  0  3 99  0  0
 0  0  663476 232836 915216 113960   154  224    0   2  0  5 99  0  0
 1  0  663132 232836 915304 113960   179  259    0  11  0  3 99  0  0
 2  0  663124 232836 915096 113960   110  175    0   4  0  3 99  0  0
```

参数列总共15个参数，分为4大类：

- procs(进程)
  - r: Running队列中进程数量
  - b: IO wait的进程数量
- memory(内存)
  - free: 可用内存大小
  - mapped：mmap映射的内存大小
  - anon: 匿名内存大小
  - slab: slab的内存大小
- system(系统)
  - in: 每秒的中断次数(包括时钟中断)
  - cs: 每秒上下文切换的次数
- cpu(处理器)
  - us: user time
  - ni: nice time
  - sy: system time
  - id: idle time
  - wa: iowait time
  - ir: interrupt time

总结

1. `dumpsys meminfo` 适用场景： 查看进程的oom adj，或者dalvik/native等区域内存情况，或者某个进程或apk的内存情况，功能非常强大；
2. `procrank` 适用场景： 查看进程的VSS/RSS/PSS/USS各个内存指标；
3. `cat /proc/meminfo` 适用场景： 查看系统的详尽内存信息，包含内核情况；
4. `free` 适用场景： 只查看系统的可用内存；
5. `showmap` 适用场景： 查看进程的虚拟地址空间的内存分配情况；

6. `vmstat`适用场景： 周期性地打印出进程运行队列、系统切换、CPU时间占比等情况；

Android内存泄漏分析工具

MAT

课上重点讲解

Android Studio Memory-profiler

https://developer.android.com/studio/profile/memory-profiler#performance

LeakCanary

https://github.com/square/leakcanary

Android内存泄漏常见场景以及解决方案

1、资源性对象未关闭

对于资源性对象不再使用时，应该立即调用它的close()函数，将其关闭，然后再置为null。例如Bitmap等资源未关闭会造成内存泄漏，此时我们应该在Activity销毁时及时关闭。

2、注册对象未注销

例如BraodcastReceiver、EventBus未注销造成的内存泄漏，我们应该在Activity销毁时及时注销。

3、类的静态变量持有大数据对象

尽量避免使用静态变量存储数据，特别是大数据对象，建议使用数据库存储。

4、单例造成的内存泄漏

优先使用Application的Context，如需使用Activity的Context，可以在传入Context时使用弱引用进行封装，然后，在使用到的地方从弱引用中获取Context，如果获取不到，则直接return即可。

5、非静态内部类的静态实例

该实例的生命周期和应用一样长，这就导致该静态实例一直持有该Activity的引用，Activity的内存资源不能正常回收。此时，我们可以将该内部类设为静态内部类或将该内部类抽取出来封装成一个单例，如果需要使用Context，尽量使用Application Context，如果需要使用Activity Context，就记得用完后置空让GC可以回收，否则还是会内存泄漏。

6、Handler临时性内存泄漏

Message发出之后存储在MessageQueue中，在Message中存在一个target，它是Handler的一个引用，Message在Queue中存在的时间过长，就会导致Handler无法被回收。如果Handler是非静态的，则会导致Activity或者Service不会被回收。并且消息队列是在一个Looper线程中不断地轮询处理消息，当这个Activity退出时，消息队列中还有未处理的消息或者正在处理的消息，并且消息队列中的Message持有Handler实例的引用，Handler又持有Activity的引用，所以导致该Activity的内存资源无法及时回收，引发内存泄漏。解决方案如下所示：

- 1、使用一个静态Handler内部类，然后对Handler持有的对象（一般是Activity）使用弱引用，这样在回收时，也可以回收Handler持有的对象。

- 2、在Activity的Destroy或者Stop时，应该移除消息队列中的消息，避免Looper线程的消息队列中有待处理的消息需要处理。

需要注意的是，AsyncTask内部也是Handler机制，同样存在内存泄漏风险，但其一般是临时性的。对于类似AsyncTask或是线程造成的内存泄漏，我们也可以将AsyncTask和Runnable类独立出来或者使用静态内部类。

7、容器中的对象没清理造成的内存泄漏

在退出程序之前，将集合里的东西clear，然后置为null，再退出程序

8、WebView

WebView都存在内存泄漏的问题，在应用中只要使用一次WebView，内存就不会被释放掉。我们可以为WebView开启一个独立的进程，使用AIDL与应用的主进程进行通信，WebView所在的进程可以根据业务的需要选择合适的时机进行销毁，达到正常释放内存的目的。

9、使用ListView时造成的内存泄漏

在构造Adapter时，使用缓存的convertView。